

# Part 3. SpringBoot数据访问

## 1. SpringBoot 整合 MyBatis

SpringBoot 默认采用整合 SpringData 的方式统一处理数据访问层，通过添加大量的自动配置，引入各种数据访问模版（xxxTemplate）以及统一的 Repository Interface，从而达到简化数据访问层的操作。



SpringData 提供了多种类型数据库支持，对支持的数据库进行了整合管理，提供了各种依赖启动器，常见的有：

名称	描述
spring-boot-starter-data-jpa	使用Spring Data JPA与Hibernate
spring-boot-starter-data-mongodb	使用MongoDB和Spring Data MongoDB
spring-boot-starter-data-neo4j	使用Neo4j图数据库和Spring Data Neo4j
spring-boot-starter-data-redis	使用Redis键值数据存储与Spring Data Redis和Jedis客户端

当然第三方也能开发自己的starter (e.g. mybatis)。

### 1.1 项目的基础环境搭建

#### Step 1. Prepare Table yu\_article and yu\_comment

Query 1	yu_article	yu_comment
Result Grid   Filter Rows: <input type="text" value="Search"/>		
id	title	content
▶ 1	Spring Boot基础入门	从入门到放弃...
2	Spring Cloud基础入门	从入门到成佛...
NULL	NULL	NULL

⚡ Query 1

⚡ yu\_article

⚡ yu\_comment

Result Grid





Filter Rows:

	id	content	author	a_id	
▶	1	很全、很详细	lucy	1	
	2	赞一个	tom	1	
	3	很详细	eric	1	
	4	666	张三	1	
	5	Fantastic	莉丝	2	
	NULL	NULL	NULL	NULL	

yu\_comment's a\_id = yu\_article's id

## Step 2. Create project by Spring Initializr

Choose "SQL":

1. MyBatis Framework
2. MySQL Driver

## Step 3. Create Comment class and Article class

Match DB table yu\_comment and yu\_article respectively.

```
public class Comment {

    private Integer id; // 评论ID
    private String content;
    private String author;
    private Integer aId; // 被评论的文章的ID => 表中column是"a_id"
```

这里需要在 application.properties 中开启 驼峰命名匹配映射配置：

```
mybatis.configuration.map-underscore-to-camel-case=true
```

```
public class Article {

    private Integer id; // 文章ID
    private String title;
    private String content;
```

## Step 4. Create Configuration file "application.properties"

```
# Mysql DB connection
spring.datasource.url=jdbc:mysql://localhost:3306/springbootdata?
serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=rootroot
# No driver class here cause this is spirngboot project based on SPI.
# Current driver is com.mysql.cj.jdbc.Driver
```

## 1.2 整合 MyBatis 进 SpringBoot

### 1.2.1 注解方式来整合

Step 1. 创建CommentMapper Interface 用于操作yu\_comment table

```
@Mapper // 标示该Interface是Mybatis的文件，并且能让SpringBoot能扫描到并生成该接口代理对象，存入容器
public interface CommentMapper {
    @Select("select * from yu_comment where id = #{suibian}")
    public Comment findById(Integer id);
}
```

【注意】添加 驼峰命名匹配映射到 application.properties 文件中。

Step 2. Test

```
// 这里有IDE的红线报错，但其实commentMapper是在runtime被autowired到的，所以可忽略该报错
@Autowired
private CommentMapper commentMapper;

@Test
public void testCommentMapper() {
    Comment result = commentMapper.findById(1);
    System.out.println(result);
}
```

### 1.2.2 配置文件方式来整合

### Step 1. 创建 ArticleMapper Interface

```
@Mapper
public interface ArticleMapper {

    public Article findArticle(Integer id);
}
```

### Step 2. 创建xml文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="com.lagou.mapper.ArticleMapper">

    <select id="findArticle" parameterType="int" resultType="article">
        select * from yu_article where id = #{id}
    </select>

</mapper>
```

### Step 3. 配置xml文件路径进入 application.properties

```
# 配置mybatis 映射配置文件路径
mybatis.mapper-locations=classpath:mapper/*.xml
# 配置mybatis 映射配置文件中的实体类别名 = 类名全小写
mybatis.type-aliases-package=com.lagou.pojo
```

### Step 4. Test

Autowired 同样有红线，可以不用管。

使用 @MapperScan 去批量扫整个mapper pkg，这样就可以不用在每一个Interface上写 @Mapper 了：

```
@MapperScan("com.lagou.mapper")
@SpringBootApplication
public class Springboot03DataApplication {...}
```

这样在test中也能正常执行。

## 2. SpringBoot 整合 JPA

Step 1. Add JPA dependency

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

Step 2. Create ORM Pojo class

```
@Entity
@Table(name = "yu_comment")
public class Comment {

    @Id // 映射主键id
    @GeneratedValue(strategy = GenerationType.IDENTITY) // 主键自增
    private Integer id;
    private String content;
    private String author;
    @Column(name = "a_id") // attribute名称和column的名称不一致时，才专门加@Column 注解
    private Integer aId;
```

Step 3. Create CommentRepository Interface which extending JpaRepository

```
// 实体类，主键类型
public interface CommentRepository extends JpaRepository<Comment, Integer> {
}
```

Step 4. Test

```

@Autowired
private CommentRepository commentRepository;

@Test
public void testJPAFindComment() {
    Optional<Comment> optional = commentRepository.findById(1);
    if (optional.isPresent()) {
        System.out.println(optional.get());
    }
}

```

### 3. SpringBoot 整合 Redis

Step 1. Add Redis dependency

```

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

```

Step 2. Create Person class and Address class

```

@RedisHash(value = "persons") // 指定实体类对象在redis中的存储空间名字
public class Person {

    @Id // 主键, Redis会自动使用String形式的hashkey来标示唯一的实体类对象id, 当然也可手动设置
    private String id;

    @Indexed // 二级索引 in Redis, 名称即属性名
    private String firstname;

    @Indexed
    private String lastname;

    private Address address;
}

```

```
public class Address {

    @Indexed // 可以在address的基础上去查找person
    private String city;

    @Indexed
    private String country;
}
```

### Step 3. Create Repository Interface extending CrudRepository

这里直接去继承最底层的 CrudRepository Interface。JpaRepository是SpringBoot整合JPA特有的。如果非要继承JpaRepository，在pom中加入JPA依赖后方可继承（但没必要）。

```
// 要操作的实体类，主键类型
public interface PersonRepository extends CrudRepository<Person, String> {

    // Find all people living in the city
    List<Person> findByAddress_City(String cityName);
}
```

### Step 4. Configure Redis DB connection in application.properties

By default `redis-cli` connects to the server at 127.0.0.1 port 6379 ([link](#))

```
# Redis DB connection
spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.password=
```

### Step 5. Test

```
@Autowired
private PersonRepository personRepository;

@Test
public void testRedis() {
    Person person = new Person();
    person.setFirstname("Jenny");
    person.setLastname("Kim");
}
```

```

        Address address = new Address();
        address.setCity("Soul");
        address.setCountry("Korean");
        person.setAddress(address);

        personRepository.save(person); // Add the person into Redis
    }

```

Then go to check Redis with a city name:

```

@Test
public void checkPersonInRedis() {
    List<Person> result = personRepository.findByAddress_City("Soul");
    for (Person person : result) {
        System.out.println(person);
    }
}

/*
 * Person{id='a065ee3e-6f0a-4efe-9aef-aae186a2ea64',
 *     firstname='Jenny', lastname='Kim', address=Address{city='Soul',
 * country='Korean'}}
 * */

```

由于在Redis数据库中生成了相应的二级索引，所以可以通过二级索引来做查询。比如上面的city name对于Redis DB “persons” 来说就是二级索引。如果没有设置二级索引，那么该查询结果将为空。

## Part 4. SpringBoot视图技术

### 4.1 支持的视图技术（了解）

前端模版引擎技术实现了前后端分离开发。SpringBoot提供整合支持多种常用模版引擎技术：FreeMarker、Thymeleaf、Mustache等。

但SpringBoot不太支持JSP模版，因为存在以下限制：

- SpringBoot默认使用嵌入式Servlet容器以Jar包的方式来进行项目打包部署，而Jar包方式并不支持JSP模版。
- 若使用Undertow嵌入式容器部署SpringBoot项目，也不支持JSP模版。
- SpringBoot默认提供了一个处理请求路径“/error”但统一错误处理器，返回具体异常信息。使用JSP模版时，无法对默认的错误处理器进行覆盖，只能根据SpringBoot的要求在指定位置定制error页面。



## 4.2 Thymeleaf

是在使用SpringBoot框架进行页面设计时，常被选择的模版引擎技术（基于服务器端的Java）。

### 4.2.1 Thymeleaf 语法

Thymeleaf 标签能够动态地替换掉静态内容。

```
<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">
  <head>

    <meta charset="UTF-8">
    <link rel="stylesheet" type="text/css" media="all" href="../../css/gtvv.css"
th:href="@{/css/gtvv.css}" />

    <title>Title</title>

  </head>
  <body>

    <p th:text="${hello}">只有当${hello}没有值的时候才会显示这句话</p>
  </body>

</html>
```

1. xmlns:th="<http://www.thymeleaf.org>" 用于引入Thymeleaf标签，使用关键字“th”标注标签。
2. 常用标签：

th: 标签	说明
th:insert	布局标签，替换内容到引入的文件
th:replace	页面片段包含（类似JSP中的include标签）
th:each	元素遍历（类似JSP中的c:forEach标签）
th:if	条件判断，如果为真
th:unless	条件判断，如果为假
th:switch	条件判断，进行选择性匹配
th:case	条件判断，进行选择性匹配
th:value	属性值修改，指定标签属性值
th:href	用于设定链接地址
th:src	用于设定链接地址
th:text	用于指定标签显示的文本内容

### 3. 标准表达式（5种）

看讲义：Page 47

#### （1）变量表达式 \${}

内置对象，这里是#locale

```
The locale country is: <span th:text="${#locale.country}">US</span>
```

这里会展示从 #locale 中拿到的country值，而不是hardcode的 US。

#### （2）选择变量表达式 \*{}

```
<div th:object="${book}">
  <p>title: <span th:text="*{title}">标题</span>.</p>
</div>
```

\*{title} 获取当前指定对象book的title 属性值。

#### （3）消息表达式 #{}，见4.2.4

用于读取国际化内容从而进行动态替换和展示。

#### （4）链接表达式 @{}

有参表达式中，要按照格式：@{路径（参数名称=参数值，参数名称=参数值）}，同时该参数值可以使用变量表达式来传递动态参数值。

```
<a th:href="@{http://localhost:8080/order/details(orderId=${o.id})}">view</a> <a  
th:href="@{/order/details(orderId=${o.id})}">view</a>
```

#### (5) 片段表达式 ~{}

最常见的用法是使用 th:insert 或者 th:replace 属性插入片段：

```
<div th:insert="~{thymeleafDemo::title}"></div>
```

这里 Thymeleaf 会自动查找 “/resources/templates/” 目录下的 以thymeleafDemo 为模版、以title 为片段名的片段。

## 4.2.2 基本使用

1. pom文件导入
2. application.properties 中配置：

```
spring.thymeleaf.cache = true #启用模版缓存，默认为true。但在开发过程中通常会关闭缓存，以便修改能及时展现出来  
spring.thymeleaf.encoding = UTF-8  
spring.thymeleaf.mode = HTML5 #应用与模版的模版模式  
spring.thymeleaf.prefix = classpath:/templates/ #指定模版页面存放路径  
spring.thymeleaf.suffix = .html #指定模版页面名称的后缀
```

3. 关于静态资源访问

SpringBoot默认设置的静态资源访问路径是 resources 目录中的 public、static、resources 三个子目录。

## 4.2.3 SpringBoot 整合 Thymeleaf

Step 1. 创建SpringBoot项目，引入Thymeleaf依赖

选 Spring Web 和 Thymeleaf

Step 2. 添加配置进 application.properties

```
# thymeleaf页面缓存设置  
spring.thymeleaf.cache=false
```

### Step 3. 创建 Controller

向登录页面 login.html 跳转的同时，携带当前年份信息。

```
@Controller
public class LoginController {

    @RequestMapping("/toLoginPage")
    public String toLoginPage(Model model){

        model.addAttribute("currentYear",
            Calendar.getInstance().get(Calendar.YEAR));

        return "login";
    }
}
```

### Step 4. 创建模版页面 + 引入静态资源文件

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>用户登录界面</title>
    <link th:href="@{/login/css/bootstrap.min.css}" rel="stylesheet">
    <link th:href="@{/login/css/signin.css}" rel="stylesheet">
</head>
<body class="text-center">
<!-- 用户登录form表单 -->
<form class="form-signin">
    
    <h1 class="h3 mb-3 font-weight-normal">请登录</h1>
    <input type="text" class="form-control"
        th:placeholder="用户名" required="" autofocus="">
    <input type="password" class="form-control"
        th:placeholder="密码" required="">
    <div class="checkbox mb-3">
        <label>
            <input type="checkbox" value="remember-me"> 记住我
        </label>
    </div>
</form>
```

```

</div>
<button class="btn btn-lg btn-primary btn-block" type="submit">登录</button>
<p class="mt-5 mb-3 text-muted">© <span th:text="${currentYear}">2019</span>-
<span th:text="${currentYear}+1">2020</span></p>

</form>
</body>
</html>

```

其中：

```

<span th:text="${currentYear}">2019</span>-<span
th:text="${currentYear}+1">2020</span>

```

不再显示 hardcode 的 2019-2020，而是显示从后端拿上来的 currentYear-currentYear+1

静态资源放到 "/resources/static/login/css" 和 "/resources/static/login/img" 中。

Step 5. 效果测试

## 4.2.4 配置国际化页面

即在页面中支持中英文的切换。

### 1. 编写多语言国际化配置文件

创建 i18n folder，其中包含3个配置文件，并且，由于Springboot 默认识别的语言配置文件为类路径 resources 下的 messages.properties，其他语言国际化文件的名称必须严格按照格式 文件前缀名\_语言代码\_国家代码 来命名。

1. login.properties
2. login\_zh\_CN.properties
3. login\_en\_US.properties

### 2. 编写全局配置文件

由于使用 login.properties 来作为默认语言配置文件，所以需要在全局配置文件中设置一下才能引用自定义国际化配置。

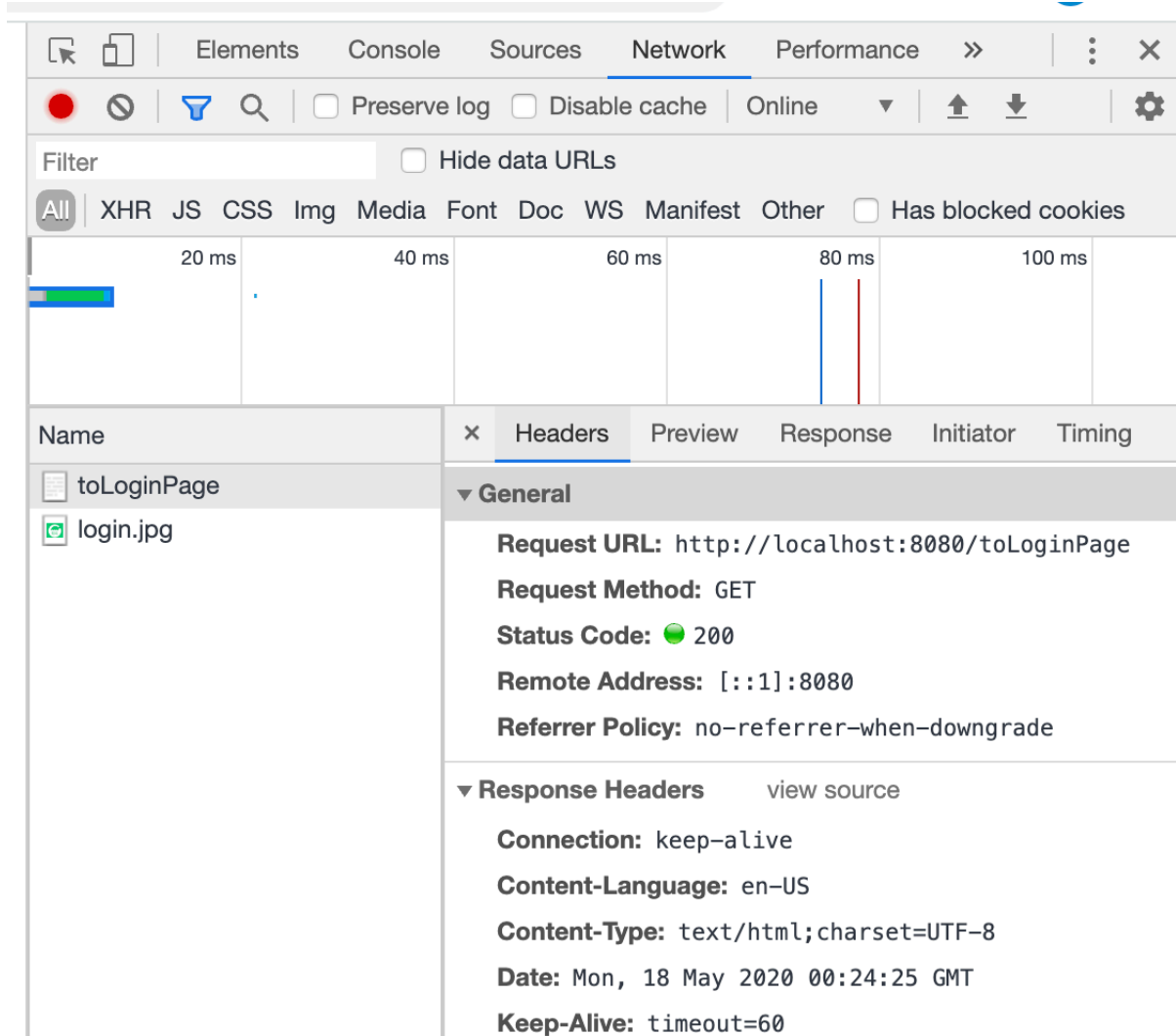
```

# 配置国际化文件基础名，这里的 login 是文件前缀名
spring.messages.basename=i18n.login

```

### 3. 为了实现手动语言切换功能，创建自定义区域信息解析器

SpringBoot中有个 `LocaleResolver`，默认情况下它会根据 http request 中的 header 来选择支持显示的语言配置文件（header见下，这里可以看到 `content-language` 是 `en-US`）



创建 `com.lagou.config` pkg,

```
package com.lagou.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.LocaleResolver;
import org.thymeleaf.util.StringUtils;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.Locale;

@Configuration
public class MyLocaleResolver implements LocaleResolver {
```

// 将自定义的 MyLocaleResolver 重新注册成为一个类型为 LocaleResolver 的bean组件, 从而可以覆盖默认的 LocaleResolver。

```
@Bean
public LocaleResolver localeResolver() {
    return new MyLocaleResolver();
}

// 完成自定义区域解析方式
@Override
public Locale resolveLocale(HttpServletRequest httpServletRequest) {
    Locale locale = null;
    // Obtain the value of "lan" in request: zh_CN, en_US or ''
    String lan = httpServletRequest.getParameter("lan");
    if (!StringUtils.isEmpty(lan)) {
        String[] strings = lan.split("_");
        locale = new Locale(strings[0], strings[1]);
    } else {
        // Obtain info from header "Accept-Language: zh-CN ,zh;q=0.9"
        String header = httpServletRequest.getHeader("Accept-Language");
        String[] split = header.split(",");
        String[] strings = split[0].split("-");
        locale = new Locale(strings[0], strings[1]);
    }

    return locale;
}

@Override
public void setLocale(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse, Locale locale) {
}
}
```

#### 4. login page 添加国际化信息

为了让后端明白到底现在需要切换到中文还是英文, 在url中添加了参数 lan

```
<a class="btn btn-sm" th:href="@{/toLoginPage(lan='zh_CN')}">中文</a>
<a class="btn btn-sm" th:href="@{/toLoginPage(lan='en_US')}">English</a>
```

# Part 5. SpringBoot缓存管理

---

## 1. 默认缓存管理

---

SpringBoot 继承了Spring framework的缓存管理功能，使用 @EnableCaching 开启基于注解的缓存支持。

### 1.1 基础环境搭建

数据准备：2个表yu\_article 和 yu\_comment

需求：Dao <= Service <= Controller

Step 1. Create a SpringBoot project.

Based on SQL dependency (Spring Data JPA + MySQL Driver) and Web dependency (Spring Web).

Step 2. Create pojo Comment class with JPA annotations

Step 3. Create CommentRepository and a query inside.

- 方式一: @Entity + @Table(name) + @Query(value="sql", nativeQuery=true), 这样原生SQL语句才能被识别

```
@Entity
@Table(name = "yu_comment")
public class Comment {

}
```



```

public interface CommentRepository extends JpaRepository<Comment, Integer> {

    // 根据评论id修改评论作者
    @Transactional
    @Modifying // Cause this is an Update operation
    @Query(value = "update yu_comment c set c.author = ?1 where c.id = ?2",
nativeQuery = true)
    public int updateComment(String author, Integer id);
}

```

- 方式二：@Entity(name) + @Query(value="sql"), 不需要特别标注使用原生SQL与否
- 其他：@Entity(name) + @Table(name), 这样可以去编写HQL或者JPQL

Step 4. Create CommentService class and findByCommentId method inside

Step 5. Create CommentController to call CommentService.

Step 6. DB connection configuration in application.properties

Step 7. Test

每次Query都直接要去Query DB，因为没有使用Cache。

## 1.2 默认缓存体验

Step 1. 在项目启动类上添加 @EnableCaching，从而开启SpringBoot基于注解的缓存管理支持

```

@EnableCaching
@SpringBootApplication
public class Springboot05CacheYuApplication {

```

Step 2. 设置哪些DB 操作需要缓存。@Cacheable 一般加在业务层（Service）

```

@Service
public class CommentService {

```

```

@Autowired
private CommentRepository commentRepository;

/*
 * springBoot默认装配的是SimpleCacheConfiguration, 其底层实现是
ConcurrentMap<String, Cache>。
 * ConcurrentMap.String 是 cacheNames
 * ConcurrentMap.Cache 是 key-value pair:
 *   ConcurrentMap.Cache.key = 默认是方法参数 (这里是id)。如果方法有1+个或者没有参数,
会用SpringBoot中的simpleKeyGenerate来生成key
 *   ConcurrentMap.Cache.value = 缓存结果
 * */
@Cacheable(cacheNames = "comment") // 将该方法查询结果以"comment"命名并存入
SpringBoot默认缓存中
public Comment findCommentById(Integer id) {
    Optional<Comment> optional = commentRepository.findById(id);
    if (optional.isPresent()) {
        return optional.get();
    }
    return null;
}
}

```

### Step 3. Test

现在不停的发相同id的请求, 只有第一次发的时候, 由于不能通过cacheNames找到Cache, 于是会对DB发sql来query, 其他时候都是从Cache中取。

## 1.3 缓存管理相关注解

### @EnableCaching

由Spring Framework提供, 通常配置在项目启动类上, 用于开启基于注解的缓存支持。

### @Cacheable

由Spring Framework提供, 通常放在Service层的DB查询的某个方法上, 从而可以缓存方法查询结果。

若缓存中有, 就缓存取; 若缓存没有, 查数据库, 然后把结果放缓存中。

```

/*
 * springBoot默认装配的是SimpleCacheConfiguration，其底层实现是
ConcurrentMap<String, Cache>
 * ConcurrentMap.String 是 cacheNames
 * ConcurrentMap.Cache 是 key-value pair:
 *   ConcurrentMap.Cache.key = 默认是方法参数（这里是id）。如果方法有1+个或者没有参数，
   会用SpringBoot中的simpleKeyGenerate来生成key。或者直接用自定义的key属性值(e.g. "mykey")
 *   ConcurrentMap.Cache.value = 缓存结果
 * */
@Cacheable(cacheNames = "comment", key = "mykey") // 将该方法查询结果
以"comment"命名并存入SpringBoot默认缓存中
public Comment findCommentById(Integer id) {
    ...
}

```

"condition": 仅在该条件下才缓存

"unless": 在该条件下就不缓存

## 执行流程&时机

1. Service层的DB查询方法运行之前，先去缓存中按照 cacheNames 查（CacheManager来做），若此时没有相应Cache，就自动创建。
2. 去Cache中继续通过key来查缓存data，这个key 默认是方法参数（这里是id）。如果方法有1+个或者没有参数，会用SpringBoot中的SimpleKeyGenerator class来生成key。或者直接用自定义的key属性值(e.g. "mykey")

SimpleKeyGenerator生成key的默认策略：

参数个数	key
没有参数	new SimpleKey()
有一个参数	参数值
多个参数	new SimpleKey(params)

## @CachePut

目标方法执行结束之后才生效。

场景：通过某方法将DB中数据更新，但cache中是旧数据。可以通过在该方法上加 @CachePut 来自动更新缓存中的数据。

## @CacheEvict

由Spring Framework提供，通常是用来删除缓存数据。默认执行顺序是：先通过方法调用删除DB中数据，再回来清理缓存数据。

## 2. SpringBoot 整合 Redis 缓存实现

### 2.1 SpringBoot支持的缓存组件

数据缓存管理存储依赖于Spring Framework中的 Cache 和 CacheManager interface

SpringBoot有9个缓存组件（按照顺序）可选：

1. Generic
2. JCache (JSR-107) (EhCache 3、Hazelcast、Infinispan 等)
3. EhCache 2.x
4. Hazelcast
5. Infinispan
6. Couchbase
7. Redis
8. Caffeine
9. Simple 【默认】

如果用了多个缓存组件，又没有指定具体CacheManager，就会按照上面顺序来查找有效的缓存组件来进行缓存管理。默认的是 SimpleCache 和 SimpleCacheManager。

### 2.2 基于注解的Redis缓存实现

Step 1. Add dependency in pom.xml

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

Now Springboot is using RedisCacheConfiguration.java and RedisCacheManager.java (instead of Simplexxxx)

## Step 2. Redis local DB connection

```
spring.redis.host=127.0.0.1
spring.redis.port=6379
spring.redis.password=
```

## Step 3. Modify CommentService with @Cacheable（用于缓存存储）, @CachePut（用于缓存更新） and @CacheEvict（用于缓存删除）

```
@Service
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    /*
     * 底层实现是 ConcurrentMap, 按照cache对名字查出cache data。
     * ConcurrentMap.key = 即方法的参数值。如果方法有多个或者没有参数, 会用SpringBoot中的
     simpleKeyGenerate来生成key
     * ConcurrentMap.value = 缓存结果
     *
     * unless = "#result==null" 若当前从DB查询的结果为空, 就不用将其存入缓存
     * */
    @Cacheable(cacheNames = "comment", unless = "#result==null") // 将该方法查询结果以"comment"命名并存入SpringBoot默认缓存中
    public Comment findCommentById(Integer id) {
        Optional<Comment> optional = commentRepository.findById(id);
        if (optional.isPresent()) {
            return optional.get();
        }
        return null;
    }

    /*
     * Update data in DB and Redis (最好别改变原本对应的key)
     *
     * key = "#result.id" 即把作为结果返回的comment 的id 当作其在Redis中的key
     * */
    @CachePut(cacheNames = "comment", key = "#result.id")
    public Comment updateCommentAuthor(Comment comment) {
        commentRepository.updateComment(comment.getAuthor(), comment.getId());
        return comment; // 返回 拿去做修改的comment
    }
}
```

```

    }

    /**
     * Delete data in DB and Redis
     *
     * Use cacheNames to delete record in Redis
     * */
    @CacheEvict(cacheNames = "comment")
    public void deleteCommentById(Integer id) {
        commentRepository.deleteById(id);
    }
}

```

Step 4. Add Update method and Delete method inside CommentService

```

@RequestMapping("/updateCommentAuthor")
public Comment updateCommentAuthor(Comment comment) {
    // 如果直接用input comment去update, 可能会出现 author 有值但其他attribute没值的情况
    // 所以这里先做查询, 然后去修改查询上来的existed comment的author值即可
    Comment existedComment = commentService.findCommentById(comment.getId());
    existedComment.setAuthor(comment.getAuthor());

    Comment result = commentService.updateCommentAuthor(existedComment);
    return result;
}

@RequestMapping("/deleteComment")
public void deleteComment(Integer id) {
    commentService.deleteCommentById(id);
}

```

Step 5. Don't forget to let pojo Comment class implements Serializable Interface so that it can be parsed by the frontend:

```

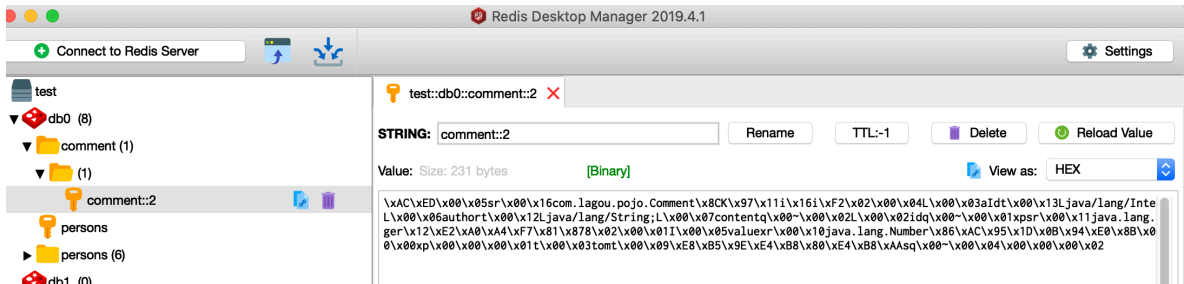
@Entity
@Table(name = "yu_comment")
public class Comment implements Serializable {
    // Implement Serializable Interface for frontend recognition
}

```

Step 6. Test (记得先开启Redis)

- 缓存查询

- URL = <http://localhost:8080/findCommentById?id=2>
- DB: query DB 来获取result
- Redis: One more record whose cacheNames = "comment", key = "comment::2" (cacheNames::id)。不过value是团mess，因为value值是经过JDK默认序列格式化后的 HEX 格式存储。这种 HEX 格式不易读，可通过后续的自定义数据的序列化格式来使其易读：



- 缓存更新

- URL = <http://localhost:8080/updateCommentAuthor?id=1&author=Jenny>
- DB: id 为 1 的记录的Author从 “lucy” 变为 “Jenny”
- Redis: 增加一条记录，其 cacheNames = "comment", key = “comment::1”
- 此时再发送 findCommentById请求id=1的Comment (<http://localhost:8080/findCommentById?id=1>)，后端不会再发 Hibernate SQL 去 query DB而是直接从 Redis中取value。

- 缓存删除

- URL = <http://localhost:8080/deleteComment?id=1>
- DB: id 为 1 的记录被删
- Redis: cacheNames = "comment", key = “comment::1” 的记录被删

## 2.3 Redis 缓存有效期

对基于注解对Redis缓存数据，可在 application.properties 文件中统一设置有效期为 1分钟（= 60000 ms），但这太过于一刀砍了，更灵活的缓存有效期设置可以使用Redis 的API来实现。：

```
# 对基于注解对Redis缓存数据 统一设置有效期为 1分钟 （= 60000 ms）
spring.cache.redis.time-to-live=60000
```

## 2.4 基于API的Redis缓存实现

Step 1. Modify CommentService to ApiCommentService

```
@Service
```

```

public class ApiCommentService {

    @Autowired
    private CommentRepository commentRepository;

    @Autowired
    private RedisTemplate redisTemplate;

    /*
     * 先查缓存，有就直接返回；没有就去查数据库，并且也存一份进缓存（同时还可设置有效期）
     */
    public Comment findCommentById(Integer id) {
        Object obj = redisTemplate.opsForValue().get("comment_" + id);
        if (obj != null) {
            return (Comment) obj;
        } else {
            Optional<Comment> optional = commentRepository.findById(id);
            if (optional.isPresent()) {
                Comment comment = optional.get();
                // 也存一份进缓存,同时还设置了有效期为 1天
                redisTemplate.opsForValue().set("comment_" + id, comment, 1,
TimeUnit.DAYS);
                return comment;
            }
        }

        return null;
    }

    /*
     * 完成DB更新之后，也去更新一下缓存
     */
    public Comment updateCommentAuthor(Comment comment) {
        commentRepository.updateComment(comment.getAuthor(), comment.getId());

        redisTemplate.opsForValue().set("comment_" + comment.getId(), comment);

        return comment; // 返回 拿去做修改的comment
    }

    /*
     * 根据 key 来删除 Redis 中的数据
     */
    public void deleteCommentById(Integer id) {
        commentRepository.deleteById(id);
    }
}

```



```
        redisTemplate.delete("comment_" + id);
    }
}
```

## Step 2. Modify CommentController to ApiCommentController

```
@RestController
@RequestMapping("api") // 避免与 CommentController使用相同的url
public class ApiCommentController {

    @Autowired
    private ApiCommentService commentService;

    @RequestMapping("/findCommentById")
    public Comment findCommentById(Integer id) {
        Comment comment = commentService.findCommentById(id);
        return comment;
    }

    @RequestMapping("/updateCommentAuthor")
    public Comment updateCommentAuthor(Comment comment) {
        // 如果直接用input comment去update, 可能会出现 author 有值但其他attribute没值的
        // 情况
        // 所以这里先做查询, 然后去修改查询上来的existed comment的author值即可
        Comment existedComment = commentService.findCommentById(comment.getId());
        existedComment.setAuthor(comment.getAuthor());

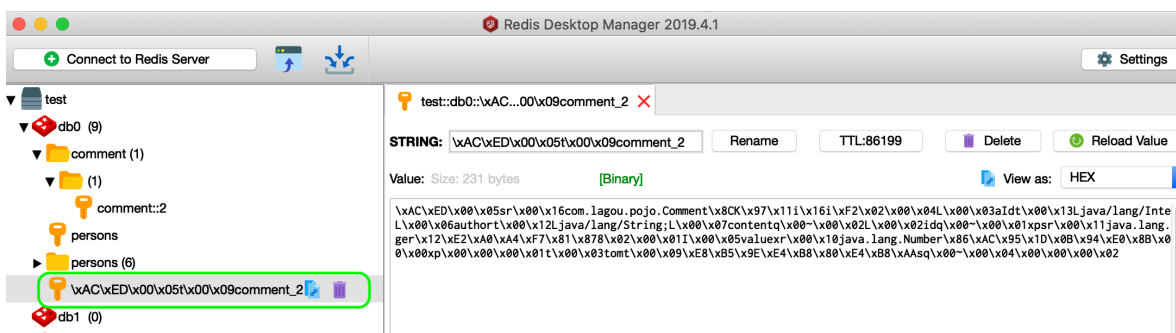
        Comment result = commentService.updateCommentAuthor(existedComment);
        return result;
    }

    @RequestMapping("/deleteComment")
    public void deleteComment(Integer id) {
        commentService.deleteCommentById(id);
    }
}
```

## Step 3. Test

- 缓存查询
  - URL = <http://localhost:8080/api/findCommentById?id=2>
  - DB: query 了DB来获取result

- Redis: 这里虽然有之前用annotation存进来的key为“comment::2”的record, 但Redis并未将这个现有的返回给browser, 而是依然向DB发了Query, 获取result之后还向 Redis 中存了一份(即绿框):



- 缓存更新
  - URL = <http://localhost:8080/api/updateCommentAuthor?id=2&author=Colde>
  - DB: Updated by the query.
  - Redis: Update 之后, 再次发送 查询 请求 (<http://localhost:8080/api/findCommentById?id=2>), 后端不会发query给DB, 说明是从 Redis 中取值返回了。
- 缓存删除
  - URL = <http://localhost:8080/api/deleteComment?id=2>
  - DB: id = 2 的记录被删除
  - Redis: 上图绿框被删除

## 3. 自定义Redis缓存序列化机制

需求: 修改序列化方式, 采用自定义JSON格式的数据序列化方式进行数据缓存管理。

Code: `springboot05_cache_redisapi`

### 3.1 自定义 RedisTemplate (仅限于 Redis API)

#### 3.1.1 Redis API 默认序列化机制

基于API的Redis缓存实现是使用 RedisTemplate 来操作的。

- RedisTemplate默认使用 JdkSerializationRedisSerializer 序列化方式。所以要进行数据缓存的pojo实体类, 必须实现JDK自带的序列化接口 (如 Serializable Interface)
- 如果自定义了缓存序列化方式defaultSerializer, 那么Redis就会用自定义的。
- RedisSerializer 是一个 Redis 序列化Interface, 默认有 6 个实现类, 分别表示6种不同的 数据序列化方式 (其中 JdkSerializationRedisSerializer 是JDK自带且默认被RedisTemplate用):

Choose Implementation of RedisSerializer (7 found)		
Ⓢ	ByteArrayRedisSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	GenericJackson2JsonRedisSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	GenericToStringSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	Jackson2JsonRedisSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	JdkSerializationRedisSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	OxmSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2
Ⓢ	StringRedisSerializer (org.springframework.data.redis.serializer)	Maven: org.springframework.data:spring-data-redis:2

### 3.1.2 对于RedisAPI，自定义 RedisTemplate 序列化机制

- 查源码：SpringBoot 提供的 RedisAutoConfiguration 自动配置类中，通过 RedisConnectionFactory 初始化了一个 RedisTemplate，当名称为“redisTemplate”的bean对象不存在时，才会trigger这个初始化。
- 思路：只要自定义的时候搞出一个名称为“redisTemplate”的bean，就会走该自定义的bean，那么在该自定义bean中实现JSON序列化方式即可。
- 实现：创建一个自己的RedisConfig 并 注入自己的 serializer：

```
@Configuration
public class RedisConfig {

    @Bean
    public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
        RedisTemplate<Object, Object> template = new RedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);

        /*
        * 创建 JSON 格式序列化对象，对缓存数据的key和value进行转换
        * */
        Jackson2JsonRedisSerializer mySerializer = new
Jackson2JsonRedisSerializer(Object.class);

        // 解决查询缓存转换异常问题 [这是工具类代码。。。]
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL,
JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);

        mySerializer.setObjectMapper(om);

        // 设置redisTemplate API 的序列化方式为 JSON
        template.setDefaultSerializer(mySerializer);

        return template;
    }
}
```

```
}
```

### 3.1.3 测试（仅限于Redis API）

URL = <http://localhost:8080/api/findCommentById?id=3>

DB: query DB来获取新的查询result

Redis: id为3的记录以JSON格式存储了！（选View as HEX或者 View as JSON都能显示JSON）



## 3.2 自定义 RedisCacheManager

现在来针对基于 注解 的Redis 缓存机制来做自定义的序列化方式。

### 3.2.1 Redis 注解 默认序列化机制

缓存自动配置类 org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration 中，

```
class RedisCacheConfiguration {
    RedisCacheConfiguration() {
    }

    @Bean
    RedisCacheManager cacheManager(CacheProperties cacheProperties,
    CacheManagerCustomizers cacheManagerCustomizers,
    ObjectProvider<org.springframework.data.redis.cache.RedisCacheConfiguration>
    redisCacheConfiguration, ObjectProvider<RedisCacheManagerBuilderCustomizer>
    redisCacheManagerBuilderCustomizers, RedisConnectionFactory
    redisConnectionFactory, ResourceLoader resourceLoader) {
```

```

        RedisCacheManagerBuilder builder =
RedisCacheManager.builder(redisConnectionFactory).cacheDefaults(this.determineCon
figuration(cacheProperties, redisCacheConfiguration,
resourceLoader.getClassLoader()));
        List<String> cacheNames = cacheProperties.getCacheNames();
        ...
    }

    private org.springframework.data.redis.cache.RedisCacheConfiguration
determineConfiguration(CacheProperties cacheProperties,
ObjectProvider<org.springframework.data.redis.cache.RedisCacheConfiguration>
redisCacheConfiguration, ClassLoader classLoader) {
        return
(org.springframework.data.redis.cache.RedisCacheConfiguration)redisCacheConfigura
tion.getIfAvailable(() -> {
            return this.createConfiguration(cacheProperties, classLoader);
        });
    }

    private org.springframework.data.redis.cache.RedisCacheConfiguration
createConfiguration(CacheProperties cacheProperties, ClassLoader classLoader) {
        Redis redisProperties = cacheProperties.getRedis();
        org.springframework.data.redis.cache.RedisCacheConfiguration config =
org.springframework.data.redis.cache.RedisCacheConfiguration.defaultCacheConfig()
;
        config = config.serializeValuesWith(SerializationPair.fromSerializer(new
JdkSerializationRedisSerializer(classLoader)));
        ...
    }

```

- 查源码：在创建RedisCacheManager时候，在determineConfiguration 时，默认使用了JdkSerializationRedisSerializer 作为序列化方式。
- 思路：创建自己的 cacheManager 作为bean，然后在其中设置想要的序列化方式即可。

### 3.2.2 自定义 RedisCacheManager

[工具类代码] 在 RedisConfig.java 中添加bean：

```

@Bean
public RedisCacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
    // 分别创建String和JSON格式序列化对象，对缓存数据key和value进行转换
    RedisSerializer<String> strSerializer = new StringRedisSerializer();
    Jackson2JsonRedisSerializer jacksonSeial =

```

```

        new Jackson2JsonRedisSerializer(Object.class);

// 解决查询缓存转换异常的问题
ObjectMapper om = new ObjectMapper();
om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
jacksonSeial.setObjectMapper(om);

// 定制缓存数据序列化方式及时效
RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
    .entryTtl(Duration.ofDays(1))
    .serializeKeysWith(RedisSerializationContext.SerializationPair
        .fromSerializer(strSerializer))
    .serializeValuesWith(RedisSerializationContext.SerializationPair
        .fromSerializer(jacksonSeial))
    .disableCachingNullValues();
RedisCacheManager cacheManager = RedisCacheManager
    .builder(redisConnectionFactory).cacheDefaults(config).build();
return cacheManager;
}

```

Test:

缓存查询 URL = <http://localhost:8080/findCommentById?id=4>

DB: 第一次查询, 发query去DB拿result

Redis: id为4的记录以JSON格式存储了!

