

第7章 SSMP框架整合及 企业级实战

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 掌握**SSMP**框架快速整合方法。
- 掌握**MyBatis-Plus**多表联合分页查找方法。
- 掌握前后端数据格式统一方法。
- 掌握前后端分离开发方法。
- 掌握**SSMP**框架业务层缓存管理方法。



主要内容

- 7.1 SSMP框架快速整合
- 7.2 SSMP框架整合案例
- 7.3 SSMP框架业务层缓存管理



7.1 SSMP框架快速整合

■ **SSMP框架整合**，即基于**Spring Boot**的**Spring + Spring MVC + Mybatis Plus**框架的整合，本小节以上一章的springbootdata为数据源，利用**Mybatis-Plus**的自动代码生成器快速构建**SSMP**整合框架。

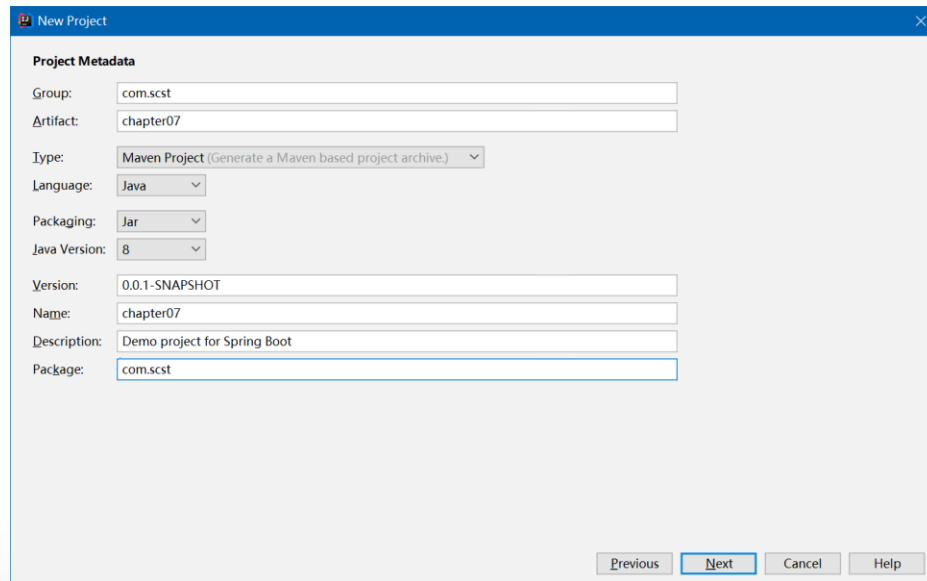
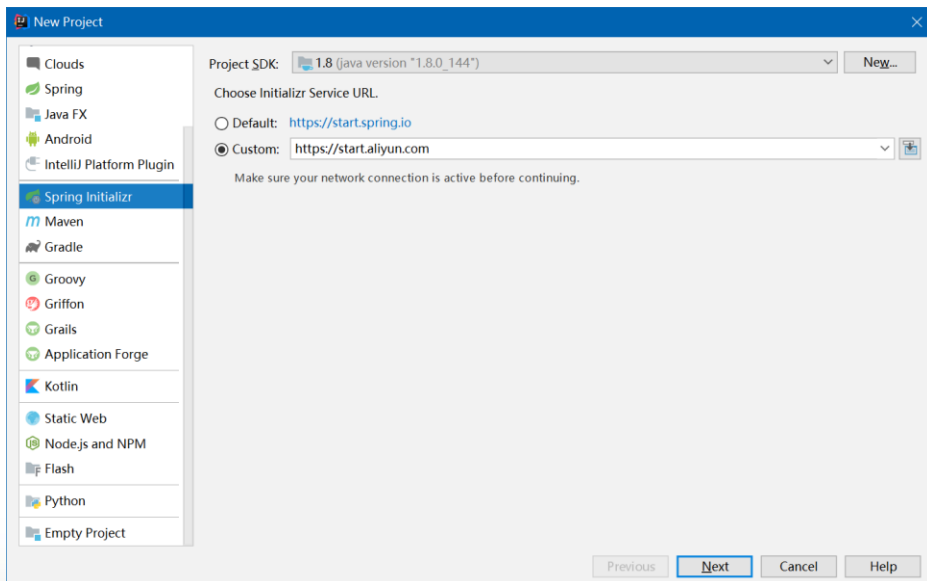
■ **整合步骤：**

- ✧①创建**Spring Boot**项目
- ✧②引入依赖启动器
- ✧③配置代码生成器
- ✧④执行并查看效果



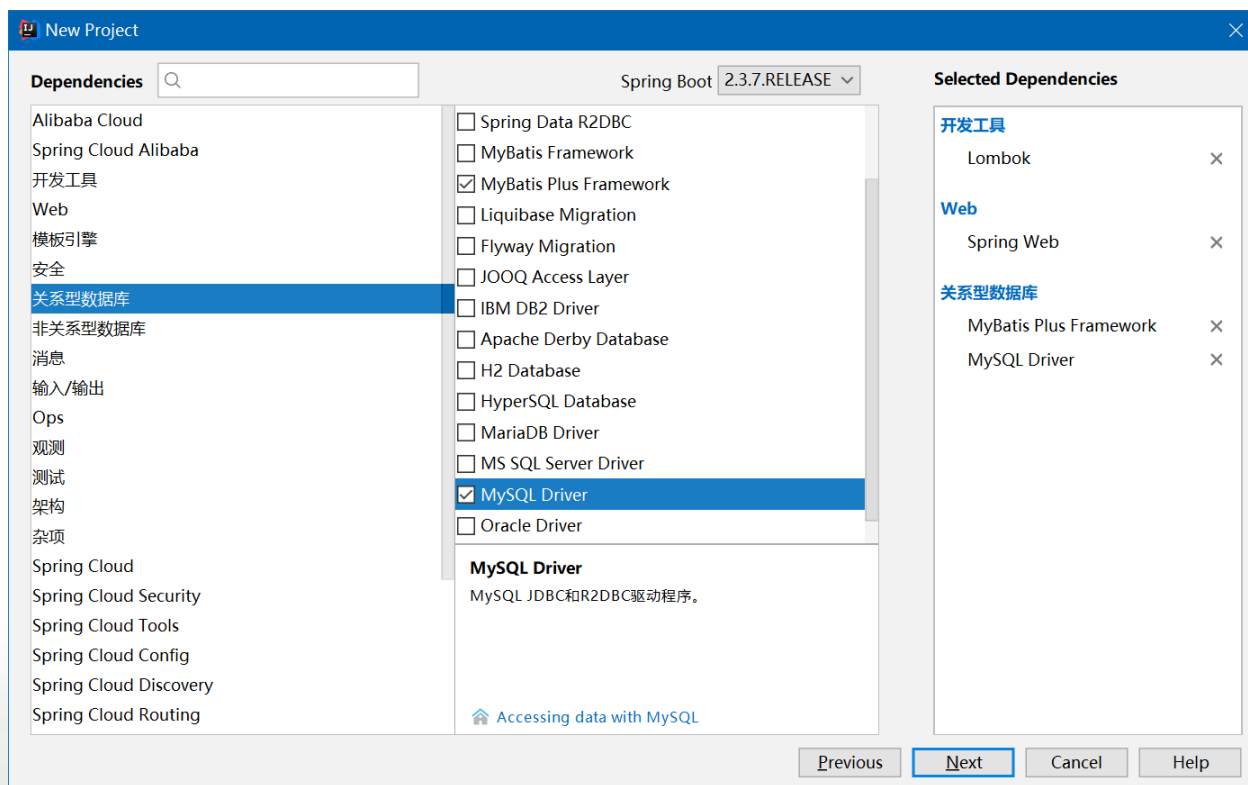
①创建Spring Boot项目

■使用Spring Initializr方式基于阿里云创建一个Spring Boot项目chapter07。



②引入依赖启动器

- 在**Dependencies**依赖选择中选择相关依赖，并在**pom.xml**中手动引入所需依赖。



手动引入的依赖启动器

```
<!--代码生成器-->
```

```
<dependency>
```

```
  <groupId>com.baomidou</groupId>
```

```
  <artifactId>mybatis-plus-generator</artifactId>
```

```
  <version>3.3.1</version>
```

```
</dependency>
```

```
<!--模板引擎依赖-->
```

```
<dependency>
```

```
  <groupId>org.apache.velocity</groupId>
```

```
  <artifactId>velocity-engine-core</artifactId>
```

```
  <version>2.2</version>
```

```
</dependency>
```

③配置代码生成器

- 为了防止被打包，在**chapter07**项目的**test**目录下的**com.scst**包中新建一个**CodeGenerator**类，通过配置**MP**的自动代码生成器**AutoGenerator**实现代码自动生成。



CodeGenerator类

```
import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.generator.AutoGenerator;
import com.baomidou.mybatisplus.generator.config.*;
import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;

public class CodeGenerator {
    public static void main(String[] args) {
        // 1、创建代码生成器
        AutoGenerator mpg = new AutoGenerator();
```



CodeGenerator类

// 2、全局配置

```
GlobalConfig gc = new GlobalConfig();
```

```
String projectPath = System.getProperty("user.dir");
```

```
gc.setOutputDir(projectPath + "/src/main/java");
```

```
gc.setAuthor("LUO");
```

```
gc.setOpen(true); //生成后是否打开资源管理器
```

```
gc.setFileOverride(true); //重新生成时文件是否覆盖
```

```
gc.setServiceName("%sService"); //去掉Service接口的首字母I
```

```
mpg.setGlobalConfig(gc);
```



CodeGenerator类

// 3、数据源配置

```
DataSourceConfig dsc = new DataSourceConfig();  
dsc.setUrl("jdbc:mysql://192.168.31.173:3306/springbootdata?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=utf8");  
dsc.setDriverName("com.mysql.cj.jdbc.Driver");  
dsc.setUsername("root");  
dsc.setPassword("root");  
dsc.setDbType(DbType.MYSQL);  
mpg.setDataSource(dsc);
```



CodeGenerator类

// 4、包配置

```
PackageConfig pc = new PackageConfig();  
pc.setParent("com.scst");  
pc.setController("controller");  
pc.setEntity("domain");  
pc.setService("service");  
pc.setMapper("mapper");  
mpg.setPackageInfo(pc);
```



CodeGenerator类

// 5、策略配置

```
StrategyConfig strategy = new StrategyConfig();
```

```
strategy.setInclude("t_comment"); //需要生成代码的数据库表
```

```
strategy.setNaming(NamingStrategy.underline_to_camel); //数据库表映射到实体的命名策略
```

```
strategy.setTablePrefix("t_"); //生成实体时去掉表前缀t_
```

```
strategy.setColumnNaming(NamingStrategy.underline_to_camel); //数据库表字段映射到实体  
属性的命名策略
```

```
strategy.setEntityLombokModel(true); // 使用lombok模型
```

```
strategy.setRestControllerStyle(true); //使用restful api风格
```

```
strategy.setControllerMappingHyphenStyle(true); //url中驼峰转连字符
```

```
mpg.setStrategy(strategy);
```

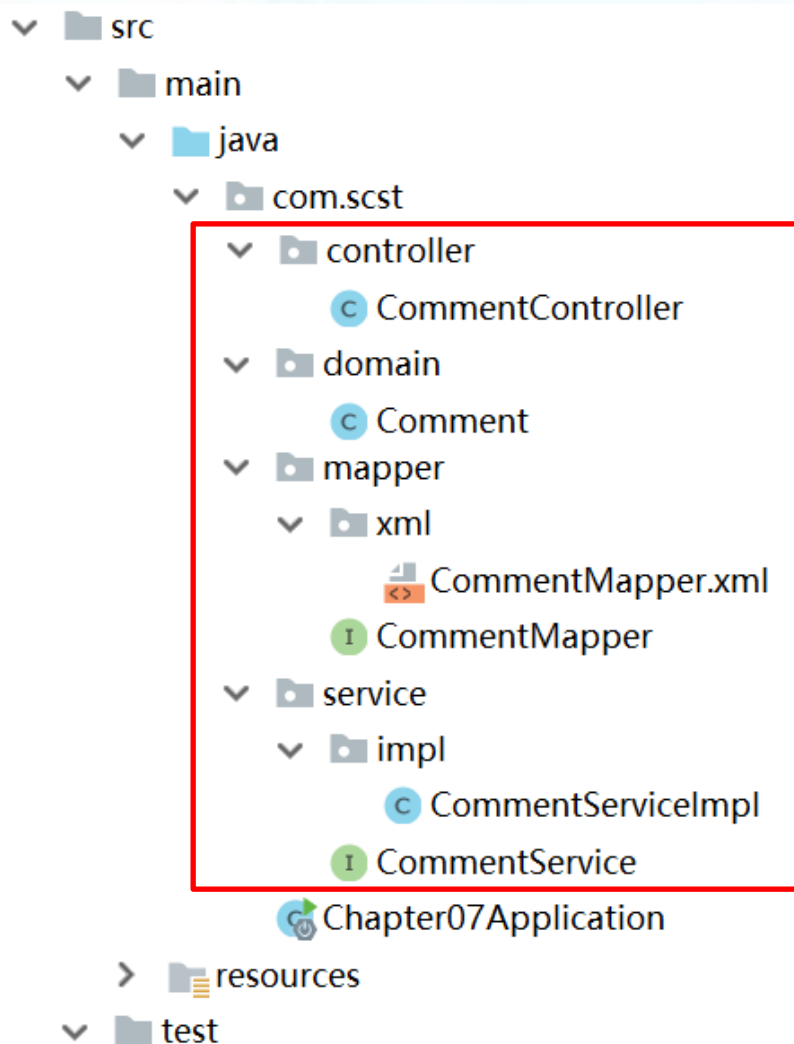
// 6、执行

```
mpg.execute();
```

```
}
```

```
}
```

④执行并查看效果



自动生成的类和类接口

```
@RestController
```

```
public class CommentController {  
}
```

```
public class Comment {  
}
```

```
public interface CommentMapper extends BaseMapper<Comment> {  
}
```

```
@Service
```

```
public class CommentServiceImpl extends ServiceImpl<CommentMapper, Comment> implements  
CommentService {  
}
```









































```
public interface CommentService extends IService<Comment> {  
}
```

内置的业务层通用接口IService

```
▼ 1 IService
  (m) save(T): boolean
  (m) saveBatch(Collection<T>): boolean
  (m) saveBatch(Collection<T>, int): boolean
  (m) saveOrUpdateBatch(Collection<T>): boolean
  (m) saveOrUpdateBatch(Collection<T>, int): boolean
  (m) removeById(Serializable): boolean
  (m) removeByMap(Map<String, Object>): boolean
  (m) remove(Wrapper<T>): boolean
  (m) removeByIds(Collection<? extends Serializable>): boolean
  (m) updateById(T): boolean
  (m) update(T, Wrapper<T>): boolean
  (m) update(Wrapper<T>): boolean
  (m) updateBatchById(Collection<T>): boolean
  (m) updateBatchById(Collection<T>, int): boolean
  (m) saveOrUpdate(T): boolean
  (m) getById(Serializable): T
  (m) listByIds(Collection<? extends Serializable>): Collection<T>
  (m) listByMap(Map<String, Object>): Collection<T>
  (m) getOne(Wrapper<T>): T
  (m) getOne(Wrapper<T>, boolean): T
  (m) getMap(Wrapper<T>): Map<String, Object>
```



内置的业务层通用接口IService

- (m)   getObj(Wrapper<T>, Function<? super Object, V>): V
- (m)   count(Wrapper<T>): int
- (m)   count(): int
- (m)   list(Wrapper<T>): List<T>
- (m)   list(): List<T>
- (m)   page(IPage<T>, Wrapper<T>): IPage<T>
- (m)   page(IPage<T>): IPage<T>
- (m)   listMaps(Wrapper<T>): List<Map<String, Object>>
- (m)   listMaps(): List<Map<String, Object>>
- (m)   listObjs(): List<Object>
- (m)   listObjs(Function<? super Object, V>): List<V>
- (m)   listObjs(Wrapper<T>): List<Object>
- (m)   listObjs(Wrapper<T>, Function<? super Object, V>): List<V>
- (m)   pageMaps(IPage<T>, Wrapper<T>): IPage<Map<String, Object>>
- (m)   pageMaps(IPage<T>): IPage<Map<String, Object>>
- (m)   getBaseMapper(): BaseMapper<T>
- (m)   query(): QueryChainWrapper<T>
- (m)   lambdaQuery(): LambdaQueryChainWrapper<T>
- (m)   update(): UpdateChainWrapper<T>
- (m)   lambdaUpdate(): LambdaUpdateChainWrapper<T>












内置的业务层通用实现类ServiceImpl

```
▼ c ServiceImpl
  m ServiceImpl()
  m getBaseMapper(): M ↑Service
  m retBool(Integer): boolean
  m currentModelClass(): Class<T>
  m sqlSessionBatch(): SqlSession
  m closeSqlSession(SqlSession): void
  m sqlStatement(SqlMethod): String
  m save(T): boolean ↑Service
  m saveBatch(Collection<T>, int): boolean ↑Service
  m saveOrUpdate(T): boolean ↑Service
  m saveOrUpdateBatch(Collection<T>, int): boolean ↑Service
  m removeById(Serializable): boolean ↑Service
  m removeByMap(Map<String, Object>): boolean ↑Service
  m remove(Wrapper<T>): boolean ↑Service
  m removeByIds(Collection<? extends Serializable>): boolean ↑Service
  m updateById(T): boolean ↑Service
  m update(T, Wrapper<T>): boolean ↑Service
  m updateBatchById(Collection<T>, int): boolean ↑Service
  m getById(Serializable): T ↑Service
  m listByIds(Collection<? extends Serializable>): Collection<T> ↑Service
  m listByMap(Map<String, Object>): Collection<T> ↑Service
```



内置的业务层通用实现类ServiceImpl

- m  `getOne(Wrapper<T>, boolean): T ↑IService`
- m  `getMap(Wrapper<T>): Map<String, Object> ↑IService`
- m  `count(Wrapper<T>): int ↑IService`
- m  `list(Wrapper<T>): List<T> ↑IService`
- m  `page(IPage<T>, Wrapper<T>): IPage<T> ↑IService`
- m  `listMaps(Wrapper<T>): List<Map<String, Object>> ↑IService`
- m  `listObjs(Wrapper<T>, Function<? super Object, V>): List<V> ↑IService`
- m  `pageMaps(IPage<T>, Wrapper<T>): IPage<Map<String, Object>> ↑IService`
- m  `getObj(Wrapper<T>, Function<? super Object, V>): V ↑IService`



7.2 SSMP框架整合案例

- 7.2.1 效果演示
- 7.2.2 开发流程
- 7.2.3 案例实现



7.2.1 效果演示

基于SpringBoot的SSMP整合案例

localhost:8080/pages/comments.html

阅读清单

评论管理

文章标题

用户名

评论内容

查询

新建

序号	文章标题	用户名	评论内容	操作
1	Spring Boot基础入门	zhangsan	很全、很详细	<div>编辑</div> <div>删除</div>
2	Spring Boot基础入门	lisi	超赞!	<div>编辑</div> <div>删除</div>

共 5 条

< 1 2 3 >

前往 1 页

效果演示

基于SpringBoot的SSMP整合案例

+

localhost:8080/pages/comments.html

阅读清单

市民网站

教育网站

创新创业

样书申请

学习网站

开发天地

金融基金

高考报考

评论管理

文章标题

用户名

序号	文章标题
1	Spring Boot基础入门
2	Spring Boot基础入门

新增评论

* 文章ID

* 用户ID

评论内容

取消

确定

操作

编辑

删除

编辑

删除

<

1

2

3

>

前往

1

页

效果演示

基于SpringBoot的SSMP整合案例

+

localhost:8080/pages/comments.html

阅读清单

市民网站

教育网站

创新创业

样书申请

学习网站

开发天地

金融基金

高考报考

评论管理

序号	文章标题	操作
1	Spring Boot基础入门	<div>编辑删除</div>
2	Spring Boot基础入门	<div>编辑删除</div>

编辑检查项

* 文章ID

1

* 用户ID

1

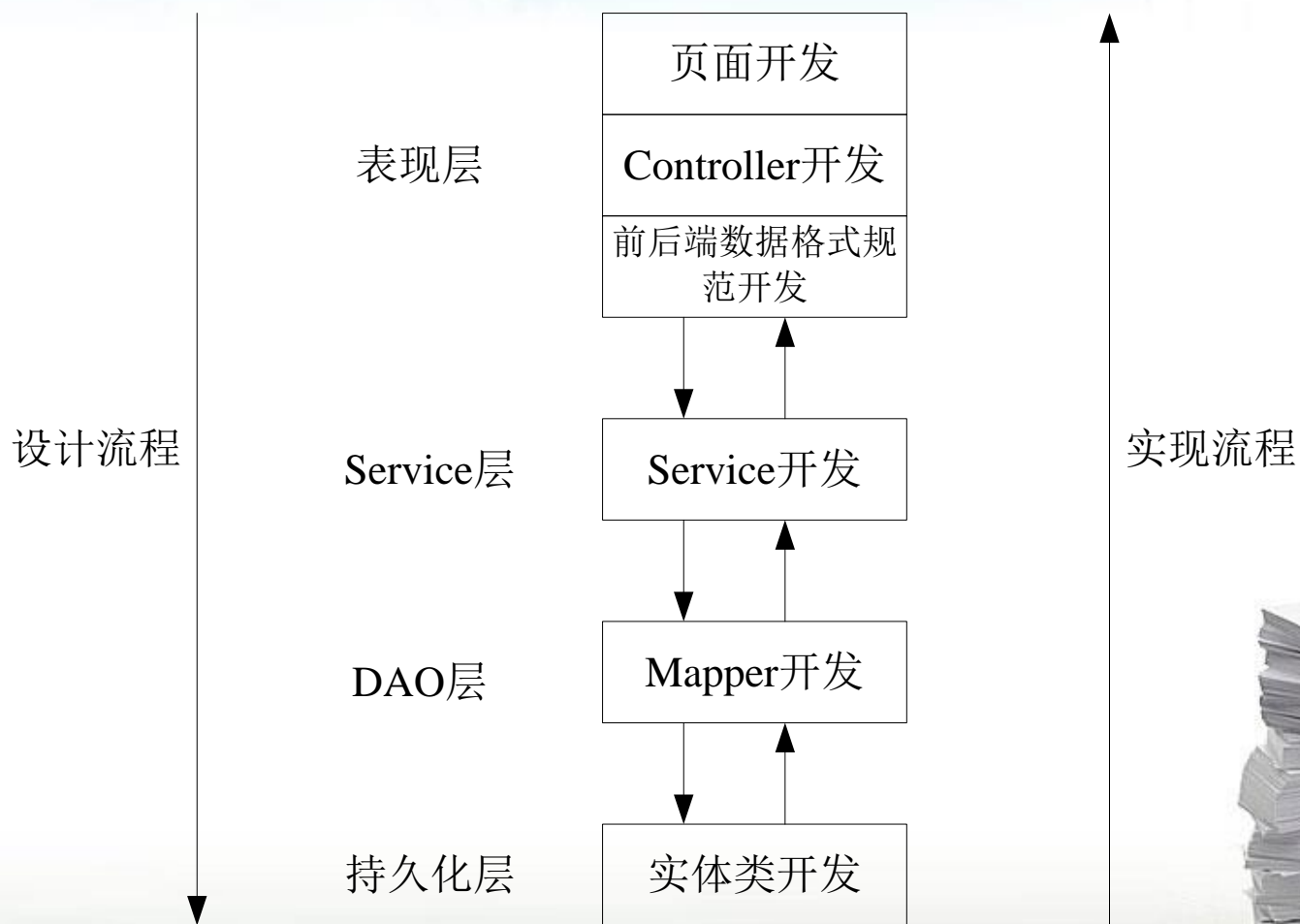
评论内容

很全、很详细

取消

确定

7.2.2 开发流程



7.2.3 案例实现

■ 本案例在7.1小节中快速生成的**SSMP**框架上进行实现。

■ 实现步骤：

✧①实体类开发

✧②**Mapper**开发

✧③**Service**开发

✧④前后端数据格式规范开发

✧⑤ **Controller**开发

✧⑥页面开发



①实体类开发

- 在chapter07项目的com.scst.domain包中完善自动生成的Comment类，并创建与分页页面中记录对应的实体类CommentVO。



Comment类

```
import lombok.Data;
```

```
@Data
```

```
public class Comment {
```

```
    private Integer id;
```

```
    private String content;
```

```
    private Integer uId;
```

```
    private Integer aId;
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Comment{" + "id=" + id + ", content='" + content + "'" + ", uId=" + uId + ", aId=" +  
aId + '}';
```

```
    }
```

```
}
```

CommentVO类

```
import lombok.Data;

@Data
public class CommentVO {
    private Integer id;
    private String content;
    private String userName;
    private String title;
}
```



配置策略

- 在项目的全局配置文件 **application.properties** 中配置数据源以及全局的表名前缀和全局id生成策略。



配置策略

数据源连接配置

spring.datasource.druid.driver-class-name: com.mysql.cj.jdbc.Driver

spring.datasource.druid.url:

jdbc:mysql://192.168.31.173:3306/springbootdata?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=utf8

spring.datasource.druid.username: root

spring.datasource.druid.password: root

spring.datasource.druid.initialSize=20

spring.datasource.druid.minIdle=10

spring.datasource.druid.maxActive=100



配置策略

#配置全局的表名前缀

mybatis-plus.global-config.db-config.table-prefix=t_

#配置全局的id生成策略

mybatis-plus.global-config.db-config.id-type=auto

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

mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdoutImpl



②Mapper开发

- 在chapter07项目的com.scst.mapper包中完善自动生成的CommentMapper接口类，并添加多表条件分页查询方法。



CommentMapper接口类

```
import com.baomidou.mybatisplus.core.conditions Wrapper;  
import com.baomidou.mybatisplus.core.metadata.IPage;  
import com.baomidou.mybatisplus.core.toolkit.Constants;  
import com.scst.domain.Comment;  
import com.baomidou.mybatisplus.core.mapper.BaseMapper;  
import com.scst.domain.CommentVO;  
import org.apache.ibatis.annotations.Mapper;  
import org.apache.ibatis.annotations.Param;  
import org.apache.ibatis.annotations.Select;
```

@Mapper

```
public interface CommentMapper extends BaseMapper<Comment> {
```



CommentMapper接口类

```
@Select("SELECT c.id,c.content,u.user_name,a.title FROM t_comment c " +  
        "LEFT JOIN t_user u ON c.u_id = u.id " +  
        "LEFT JOIN t_article a ON c.a_id = a.id " +  
        "${ew.customSqlSegment}")
```

// 多表条件分页查询

```
public IPage<CommentVO> selectCommentVOByPage(IPage<CommentVO> page,  
@Param(Constants.WRAPPER) Wrapper<CommentVO> wrapper);  
}
```



③Service开发

- 在chapter07项目的com.scst.service包及其子包impl中完善自动生成的CommentService接口类和CommentServiceImpl实现类，并添加多表条件分页查询方法。
- 为了实现分页查询和SQL性能分析，还需在chapter07项目中新建一个com.scst.config包，并在包中创建配置类MybatisPlusConfig。



CommentService接口类

```
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.scst.domain.Comment;
import com.baomidou.mybatisplus.extension.service.IService;
import com.scst.domain.CommentVO;

public interface CommentService extends IService<Comment> {
    IPage<CommentVO> selectCommentVOByPage(Long currentPage, Long pageSize,
    CommentVO commentVO);
}
```



CommentServiceImpl实现类

```
import com.baomidou.mybatisplus.core.conditions.query.QueryWrapper;
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.baomidou.mybatisplus.extension.plugins.pagination.Page;
import com.scst.domain.Comment;
import com.scst.domain.CommentVO;
import com.scst.mapper.CommentMapper;
import com.scst.service.CommentService;
import com.baomidou.mybatisplus.extension.service.impl.ServiceImpl;
import org.apache.logging.log4j.util.Strings;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class CommentServiceImpl extends ServiceImpl<CommentMapper, Comment> implements
CommentService {

    @Autowired
    CommentMapper commentMapper;
```

CommentServiceImpl实现类

// 多表条件分页查询

```
public IPage<CommentVO> selectCommentVOByPage(Long currentPage, Long pageSize, CommentVO
commentVO){
    IPage<CommentVO> page=new Page<>(currentPage,pagesize); //设置当前页号和分页大小
    QueryWrapper<CommentVO> wrapper = new QueryWrapper<>();
    String title = commentVO.getTitle();
    String userName = commentVO.getUserName();
    String content = commentVO.getContent();
    if (Strings.isEmpty(title)) wrapper.like("a.title",title);
    if (Strings.isEmpty(userName)) wrapper.like("u.user_name",userName);
    if (Strings.isEmpty(content)) wrapper.like("c.content",content);
    return commentMapper.selectCommentVOByPage(page,wrapper);
}
}
```



MybatisPlusConfig配置类

```
import com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor;
import com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MybatisPlusConfig {

    @Bean //配置分页插件
    public PaginationInterceptor paginationInterceptor(){
        return new PaginationInterceptor();
    }
}
```



MybatisPlusConfig配置类

@Bean //配置SQL性能分析插件，该插件只用于开发环境，不建议生产环境使用。

```
public PerformanceInterceptor performanceInterceptor(){  
    PerformanceInterceptor performanceInterceptor = new PerformanceInterceptor();  
    //设置SQL是否格式化  
    performanceInterceptor.setFormat(true);  
    //设置SQL最大执行时间（ms），超过时间会抛出异常。  
    performanceInterceptor.setMaxTime(100);  
    return performanceInterceptor;  
}
```



④前后端数据格式规范开发

- 为了规范前端和后端数据格式的统一，在 **chapter07** 项目的 **com.scst.controller** 包中新建子包 **utils**，并在子包中新建控制层返回结果的模型类 **R**。

✧注：**R** 中的后端对象 **data** 以 **JSON** 格式被传到前端后其成员变量属性名可能被转变为小写。

- 由此，为了统一异常处理，在子包 **utils** 中新建 **GlobalExceptionHandler** 类，统一以 **R** 格式返回异常提示信息。



模型类R

```
import lombok.Data;
```

```
@Data
```

```
//控制层返回结果的模型类，用于后端与前端进行数据格式统一
```

```
public class R {
```

```
    private Boolean flag; //成功或异常标识
```

```
    private Object data; //数据内容
```

```
    private String msg; //成功或异常信息
```

```
    public R(){}
```



模型类R

```
public R(Boolean flag){  
    this.flag = flag;  
}  
public R(Boolean flag,Object data){  
    this.flag = flag;  
    this.data = data;  
}  
public R(Boolean flag,String msg){  
    this.flag = flag;  
    this.msg = msg;  
}  
}
```

GlobalExceptionHandler类

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
//声明一个增强式控制器类
@RestControllerAdvice
public class GlobalExceptionHandler {
    //声明一个异常统一处理方法，用于拦截所有异常
    @ExceptionHandler(Exception.class)
    public R handleException(Exception ex){
        //省略（记录日志，发送消息给运维，发送邮件给开发人员，ex对象发送给开发人员
    )
        ex.printStackTrace();
        return new R(false,"服务器故障，请稍后再试！");
    }
}
```


⑤Controller开发

- 在chapter07项目的com.scst.controller包中完善自动生成的CommentController类，并添加供前端异步访问的增删改查接口方法。



CommentController类

```
import com.baomidou.mybatisplus.core.metadata.IPage;
import com.scst.controller.utils.R;
import com.scst.domain.Comment;
import com.scst.domain.CommentVO;
import com.scst.service.impl.CommentServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.io.IOException;

@RestController
@RequestMapping("/comment")
public class CommentController {

    @Autowired
    private CommentServiceImpl commentService;
```

CommentController类

```
@GetMapping("/{currentPage}/{pageSize}")  
public R getAll(@PathVariable Long currentPage, @PathVariable Long pageSize,  
CommentVO commentVO)  
{  
    IPage page =  
commentService.selectCommentVOByPage(currentPage,pageSize,commentVO);  
    //如果当前页码值大于了总页码值，那么重新执行查询操作，使用最大页码值作为当前页码值  
    if( currentPage > page.getPages()){  
        page =  
commentService.selectCommentVOByPage(page.getPages(),pageSize,commentVO);  
    }  
    return new R(page!=null,page);  
}
```

CommentController类

```
@GetMapping("/{id}")
```

```
public R getById(@PathVariable Integer id){  
    Comment comment = commentService.getById(id);  
    boolean flag = comment!=null;  
    return new R(flag, flag ? comment : "数据同步失败-_-!");  
}
```

```
@PostMapping
```

```
public R save(@RequestBody Comment comment) throws IOException {  
    if (comment.getContent().equals("123")) throw new IOException();  
    boolean flag = commentService.save(comment);  
    return new R(flag, flag ? "添加成功^_^" : "添加失败-_-!");  
}
```

CommentController类

@PutMapping

```
public R update(@RequestBody Comment comment) throws IOException {  
    if (comment.getContent().equals("123") ) throw new IOException();  
    boolean flag = commentService.saveOrUpdate(comment);  
    return new R(flag, flag ? "修改成功^^" : "修改失败-_-!");  
}
```

@DeleteMapping("/{id}")

```
public R delete(@PathVariable Integer id){  
    boolean flag = commentService.removeById(id);  
    return new R(flag, flag ? "删除成功^^" : "删除失败-_-!");  
}  
}
```

⑥页面开发

- 前后端分离结构设计中页面归属前端服务器
- 单体工程中页面放置在**resources**目录下的**static**目录中（建议执行**Maven clean**）



静态资源结构

- ▼ resources
 - ▼ static
 - ▼ css
 - style.css
 - ▼ js
 - axios-0.18.0.js
 - index.js
 - jquery.min.js
 - vue.js
 - ▼ pages
 - comments.html
 - ▼ plugins
 - > elementui
 - > font-awesome
 - templates
 - application.properties



comments.html页面

```
comments.html x
3 <head...>
14 <body class="hold-transition">
15 <div id="app">
16   <div class="content-header"...>
19   <div class="app-container">
20     <div class="box">
21       <!-- 使用elementui 中的输入框和按钮组件-->
22       <div class="filter-container"...>
30       <!-- 使用elementui 中的表格组件-->
31       <el-table size="small" current-row-key="id" :data="dataList" stripe highlight-current-row...>
44       <!-- 使用elementui 中的分页组件-->
45       <div class="pagination-container"...>
55       <!-- “新建”标签弹层 -->
56       <div class="add-form"...>
85       <!-- “编辑”标签弹层 -->
86       <div class="add-form"...>
115     </div>
116   </div>
117 </div>
118 </body>
119 <!-- 引入组件库 -->
120 <script src="../js/vue.js"></script>
121 <script src="../plugins/elementui/index.js"></script>
122 <script type="text/javascript" src="../js/jquery.min.js"></script>
123 <script src="../js/axios-0.18.0.js"></script>
124 <script>
125   var vue = new Vue({el: '#app'...})
262 </script>
```

head

```
<head>
  <!-- 页面meta -->
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>基于SpringBoot的SSMP整合案例</title>
  <meta content="width=device-width,initial-scale=1,maximum-scale=1,user-scalable=no"
name="viewport">
  <!-- 引入样式 -->
  <link rel="stylesheet" href="../plugins/elementui/index.css">
  <link rel="stylesheet" href="../plugins/font-awesome/css/font-awesome.min.css">
  <link rel="stylesheet" href="../css/style.css">
</head>
```

使用el中的输入框和按钮组件

```
<!--使用elementui中的输入框和按钮组件-->
```

```
<div class="filter-container">
```

```
  <!--通过v-model实现数据双向绑定-->
```

```
  <el-input placeholder="文章标题" v-model="pagination.title" style="width: 200px;"
```

```
  class="filter-item"></el-input>
```

```
  <el-input placeholder="用户名" v-model="pagination.userName" style="width: 200px;"
```

```
  class="filter-item"></el-input>
```

```
  <el-input placeholder="评论内容" v-model="pagination.content" style="width: 200px;"
```

```
  class="filter-item"></el-input>
```

```
  <el-button @click="getAll()" class="dalfBut">查询</el-button>
```

```
  <el-button type="primary" class="butT" @click="handleCreate()">新建</el-button>
```

```
</div>
```



使用el中的表格组件

```
<!--使用elementui中的表格组件-->
```

```
<el-table size="small" current-row-key="id" :data="dataList" stripe highlight-current-row>
```

```
  <el-table-column type="index" align="center" label="序号"></el-table-column>
```

```
  <!--绑定dataList.title、dataList.userName、dataList.content-->
```

```
  <el-table-column prop="title" label="文章标题" align="center"></el-table-column>
```

```
  <el-table-column prop="userName" label="用户名" align="center"></el-table-column>
```

```
  <el-table-column prop="content" label="评论内容" align="center"></el-table-column>
```

```
  <el-table-column label="操作" align="center">
```

```
    <template slot-scope="scope">
```

```
      <el-button type="primary" size="mini" @click="handleEdit(scope.row)">编辑</el-button>
```

```
      <el-button type="danger" size="mini" @click="handleDelete(scope.row)">删除</el-button>
```

```
    </template>
```

```
  </el-table-column>
```

```
</el-table>
```

使用el中的分页组件

```
<!--使用elementui中的分页组件-->
<div class="pagination-container">
  <el-pagination
    class="pagiantion"
    @current-change="handleCurrentChange"
    :current-page="pagination.currentPage"
    :page-size="pagination.pageSize"
    :total="pagination.total"
    layout="total, prev, pager, next, jumper">
  </el-pagination>
</div>
```




“新建”标签弹层

```
<!-- “新建” 标签弹层 -->
<div class="add-form">
  <el-dialog title="新增评论" :visible.sync="dialogFormVisible4Add">
    <el-form ref="dataAddForm" :model="formData" :rules="rules" label-position="right" label-width="100px">
      <el-row>
        <el-col :span="12">
          <el-form-item label="文章ID" prop="aid">
            <el-input v-model="formData.aid"/>
          </el-form-item>
        </el-col>
        <el-col :span="12">
          <el-form-item label="用户ID" prop="uid">
            <el-input v-model="formData.uid"/>
          </el-form-item>
        </el-col>
      </el-row>
      <el-row>
        <el-col :span="24">
          <el-form-item label="评论内容">
            <el-input v-model="formData.content" type="textarea"></el-input>
          </el-form-item>
        </el-col>
      </el-row>
    </el-form>
    <div slot="footer" class="dialog-footer">
      <el-button @click="cancel()">取消</el-button>
      <el-button type="primary" @click="handleAdd()">确定</el-button>
    </div>
  </el-dialog>
</div>
```

“编辑”标签弹层

```
<!-- “编辑” 标签弹层 -->
<div class="add-form">
  <el-dialog title="编辑检查项" :visible.sync="dialogFormVisible4Edit">
    <el-form ref="dataEditForm" :model="formData" :rules="rules" label-position="right" label-width="100px">
      <el-row>
        <el-col :span="12">
          <el-form-item label="文章ID" prop="aid">
            <el-input v-model="formData.aid"/>
          </el-form-item>
        </el-col>
        <el-col :span="12">
          <el-form-item label="用户ID" prop="uid">
            <el-input v-model="formData.uid"/>
          </el-form-item>
        </el-col>
      </el-row>
      <el-row>
        <el-col :span="24">
          <el-form-item label="评论内容">
            <el-input v-model="formData.content" type="textarea"></el-input>
          </el-form-item>
        </el-col>
      </el-row>
    </el-form>
    <div slot="footer" class="dialog-footer">
      <el-button @click="cancel()">取消</el-button>
      <el-button type="primary" @click="handleUpdate()">确定</el-button>
    </div>
  </el-dialog>
</div>
```

Vue对象

```
var vue = new Vue({
  el: '#app',
  data: {
    dataList: [], // 当前页要展示的列表数据
    dialogFormVisible4Add: false, // "新增" 表单是否可见
    dialogFormVisible4Edit: false, // "编辑" 表单是否可见
    formData: {}, // 表单数据
    rules: { // 表单数据校验规则
      uid: [{ required: true, message: '用户ID为必填项', trigger: 'blur' }],
      aid: [{ required: true, message: '文章ID为必填项', trigger: 'blur' }]
    },
    pagination: { // 分页相关模型数据
      currentPage: 1, // 当前页码
      pageSize: 2, // 每页显示的记录数
      total: 0, // 总记录数
      title: '', // 分页查询实体条件
      userName: '', // 分页查询实体条件
      content: '' // 分页查询实体条件
    }
  },
  // 钩子函数, VUE 对象初始化完成后自动执行
  created() {
    // 调用查询全部数据的操作
    this.getAll();
  },
  methods: {
  })
```

Vue对象中的方法

//分页查询

`getAll()` {

 //组织参数，拼接url请求地址

 param = "?title="+this.pagination.title;

 param += "&userName="+this.pagination.userName;

 param += "&content="+this.pagination.content;

 //发送异步请求

 axios.get("/comment/"+this.pagination.currentPage+"/"+this.pagination.pageSize+param).then((res)=>{

 this.pagination.pageSize = res.data.data.size;

 this.pagination.currentPage = res.data.data.current;

 this.pagination.total = res.data.data.total;

 this.dataList = res.data.data.records;

 });

},



Vue对象中的方法

//切换页码

```
handleCurrentChange(currentPage) {  
    //修改页码值为当前选中的页码值  
    this.pagination.currentPage = currentPage;  
    //执行查询  
    this.getAll();  
},
```

//弹出"新建"窗口

```
handleCreate() {  
    this.dialogFormVisible4Add = true;  
    this.resetForm();  
},
```

//重置表单

```
resetForm() {  
    this.formData = {};  
},
```

Vue对象中的方法

//确认添加

```
handleAdd () {  
  axios.post("/comment",this.formData).then((res)=>{  
    //判断当前操作是否成功  
    if(res.data.flag){  
      //1.关闭弹层  
      this.dialogFormVisible4Add = false;  
      this.$message.success(res.data.msg);  
    }else{  
      this.$message.error(res.data.msg);  
    }  
  }).finally(()=>{  
    //2.重新加载数据  
    this.getAll();  
  });  
},
```


Vue对象中的方法

//取消

```
cancel(){  
  this.dialogFormVisible4Add = false;  
  this.dialogFormVisible4Edit = false;  
  this.$message.info("当前操作取消");  
},
```



Vue对象中的方法

//确认修改

```
handleUpdate() {  
  axios.put("/comment",this.formData).then((res)=>{  
    //判断当前操作是否成功  
    if(res.data.flag){  
      //1.关闭弹层  
      this.dialogFormVisible4Edit = false;  
      this.$message.success(res.data.msg);  
    }else{  
      this.$message.error(res.data.msg);  
    }  
  }).finally(()=>{  
    //2.重新加载数据  
    this.getAll();  
  });  
},
```

Vue对象中的方法

//弹出“编辑”窗口

```
handleEdit(row) {  
  // row.id为该行记录在原数据库表中的id  
  axios.get("/comment/"+row.id).then((res)=>{  
    if(res.data.flag && res.data.data != null ){  
      //1.关闭弹层  
      this.dialogFormVisible4Edit = true;  
      this.formData = res.data.data;  
    }else{  
      this.$message.error(res.data.msg);  
    }  
  }).finally(()=>{  
    //2.重新加载数据  
    this.getAll();  
  });  
},
```

Vue对象中的方法

// 弹出“删除”确认框

```
handleDelete(row) {  
  this.$confirm("此操作永久删除当前信息，是否继续？","提示",{type:"info"}).then(()=>{  
    axios.delete("/comment/"+row.id).then((res)=>{  
      if(res.data.flag){  
        this.$message.success(res.data.msg);  
      }else{  
        this.$message.error(res.data.msg);  
      }  
    }).finally(()=>{  
      //2.重新加载数据  
      this.getAll();  
    });  
  }).catch(()=>{  
    this.$message.info("当前操作取消");  
  });  
}
```

7.3 SSMP框架业务层缓存管理

- 7.3.1 Spring Boot缓存管理概述
- 7.3.2 使用Redis缓存中间件
- 7.3.3 高并发下缓存失效问题



7.3.1 Spring Boot缓存管理概述

- 缓存是分布式系统中的重要组件，主要解决数据库数据的高并发访问。

- ✧ 缓存中数据格式：<key,value>

- Spring Boot对缓存提供了良好的支持。

- ✧ 其管理核心是将缓存应用于操作数据的方法（一般是业务类或其方法）中，从而减少访问数据库操作数据的次数，同时不会对程序本身造成任何干扰。

- ✧ Spring Boot默认在内存中缓存数据，而在大型系统或分布式系统中常采用Redis缓存中间件缓存数据。



Spring Boot默认缓存底层结构

- 在诸多的缓存自动配置类中，Spring Boot默认装配的是SimpleCacheConfiguration，它使用的缓存管理器CacheManager是ConcurrentMapCacheManager，使用ConcurrentHashMap作为底层的数据结构存放<cacheName, Cache>，根据cacheName可查询出Cache，而每一个Cache中存在多个key-value缓存键值对。



关于缓存管理器的自动选择

■ Spring Boot支持的缓存组件或中间件（都有相应缓存管理器）有：

- ✧ (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）

■ 当开启缓存管理后，如果不指定缓存管理器，Spring Boot会按上述顺序查找有效的缓存管理器进行缓存管理。



Spring Boot相关缓存注解

■ 在Spring Boot中，缓存相关注解主要有：

✧ ① **@EnableCaching**注解

✧ ② **@Cacheable**注解

✧ ③ **@CachePut**注解

✧ ④ **@CacheEvict**注解

✧ ⑤ **@Caching**注解

✧ ⑥ **@CacheConfig**注解



① @EnableCaching注解

- @EnableCaching注解用于开启基于注解的缓存支持。该注解需要配置在类上（在Spring Boot中，通常配置在项目启动类或配置类上）

```
// 开启Spring Boot基于注解的缓存管理支持
@EnableCaching
@SpringBootApplication
public class chapter07Application {
    public static void main(String[] args) {
        SpringApplication.run(chapter07Application.class, args);
    }
}
```

② @Cacheable注解

- @Cacheable注解用于对方法结果进行缓存存储，可以作用于类或方法（通常用在数据查询方法上）
- @Cacheable注解的执行顺序是，先进行缓存查询，如果为空则进行方法查询，并将结果进行缓存；如果缓存中有数据，不进行方法查询（不执行其作用于的方法），而是直接使用缓存数据。



@Cacheable注解的属性

属性名	说明
value/cacheNames	指定缓存空间的名称，必配属性。这两个属性二选一使用
key	指定缓存数据的key，默认使用方法参数值，可以使用SpEL表达式
keyGenerator	指定缓存数据的key的生成器，与key属性二选一使用
cacheManager	指定缓存管理器
cacheResolver	指定缓存解析器，与cacheManager属性二选一使用
condition	指定在符合某条件下，进行数据缓存
unless	指定在符合某条件下，不进行数据缓存
sync	指定是否单线程查数据库。默认false

Cache缓存支持的SpEL表达式

表达式示例	说明
<code>#root.methodName</code>	表示当前被调用的方法名
<code>#root.method.name</code>	表示当前被调用的方法名
<code>#root.target</code>	表示当前被调用的目标对象实例
<code>#root.targetClass</code>	表示当前被调用的目标对象的类
<code>#root.args[0]</code>	表示当前被调用的方法的参数列表的0号元素
<code>#root.caches[0].name</code>	表示当前被调用的方法的缓存列表的0号元素
<code>#id</code> 、 <code>#a0</code> 、 <code>#p0</code>	表示当前被调用的方法参数，可以用#参数名、#a0、#p0的形式表示（0代表参数索引，从0开始）
<code>#result</code>	表示当前被调用的方法执行后的返回结果

上述表达式都可作为**key**的属性值，但需保证**key**对每个不同请求具有唯一性。



③ @CachePut注解

- @CachePut注解作用是更新缓存数据，可以作用于类或方法（通常用在数据更新方法上）
- @CachePut注解的执行顺序是，先进行方法调用，然后将方法结果更新到缓存中。
- @CachePut注解也提供了多个属性，这些属性与@Cacheable注解的属性完全相同。



④ @CacheEvict注解

- **@CacheEvict**注解作用是**删除缓存数据**，可以作用于类或方法（通常用在数据删除方法上）。
- **@CacheEvict**注解的默认执行顺序是，先进行方法调用，然后将缓存进行清除。
- **@CacheEvict**注解也提供了多个属性，这些属性与**@Cacheable**注解的属性基本相同，除此之外，还额外提供了两个特殊属性**allEntries**和**beforeInvocation**。



@CacheEvict注解的特殊属性

■ allEntries属性

✧表示是否清除指定缓存空间中的所有缓存数据，默认值为**false**（即默认只删除指定**key**对应的缓存数据）。

■ beforeInvocation属性

✧表示是否在方法执行之前进行缓存清除，默认值为**false**（即默认在执行方法后再进行缓存清除）。



⑤ @Caching注解

- **@Caching**注解用于针对复杂规则的数据缓存管理，可以作用于类或方法，在**@Caching**注解内部包含有**cacheable**、**put**和**evict**三个属性，分别对应于**@Cacheable**、**@CachePut**和**@CacheEvict**三个注解。



示例代码

```
@Caching(cacheable={ @Cacheable(cacheNames ="comment", key = "#id")},  
          put = { @CachePut(cacheNames = "comment", key = "#result.author")})  
public Comment getComment(int comment_id){  
    .....  
}
```



⑥ @CacheConfig注解

■ **@CacheConfig**注解使用在类上，主要用于统筹管理类中所有使用**@Cacheable**、**@CachePut**和**@CacheEvict**注解标注方法中的公共属性，这些公共属性包括有**cacheNames**、**keyGenerator**、**cacheManager**和**cacheResolver**。

✧ 这样，在该类中所有方法上使用缓存注解时可以省略这些公共属性。



示例代码

```
@CacheConfig(cacheNames = "comment")
@Service
public class CommentService {
    @Autowired
    private CommentRepository commentRepository;
    @Cacheable
    public Comment findById(int comment_id){
        .....
    }
    ...
}
```


7.3.2 使用Redis缓存中间件

- 1.Redis简介
- 2.Redis的安装
- 3.Redis的整合支持
- 4.基于注解的缓存实现案例



1.Redis简介

■ Redis 是一个**开源**（**BSD**许可）的、内存中的**数据结构**存储系统，一种非关系型数据库系统，它可以用作**数据库**、**缓存**和**消息中间件**，并提供多种语言的**API**。



Redis优点

■ 1.存取速度快:

✧ **Redis**速度非常快，每秒可执行大约**110000**次的设值操作，或者执行**81000**次的读取操作。

■ 2支持丰富的数据类型:

✧ **Redis**支持开发人员常用的大多数数据类型，例如列表、集合、排序集和散列等。

■ 3.操作具有原子性:

✧ 所有**Redis**操作都是原子操作，这确保如果两个客户端并发访问，**Redis**服务器能接收更新后的值。

■ 4.提供多种功能:

✧ **Redis**提供了多种功能特性，可用作非关系型数据库、缓存中间件、消息中间件等。



2.Redis的安装

■ 下载最新版Redis:

✧ <https://github.com/MSOpenTech/redis/tags>

✧ Redis-x64-3.2.100.msi

■ 下载完成后，将安装包安装到默认目录下，Redis将以Windows服务自动启动。

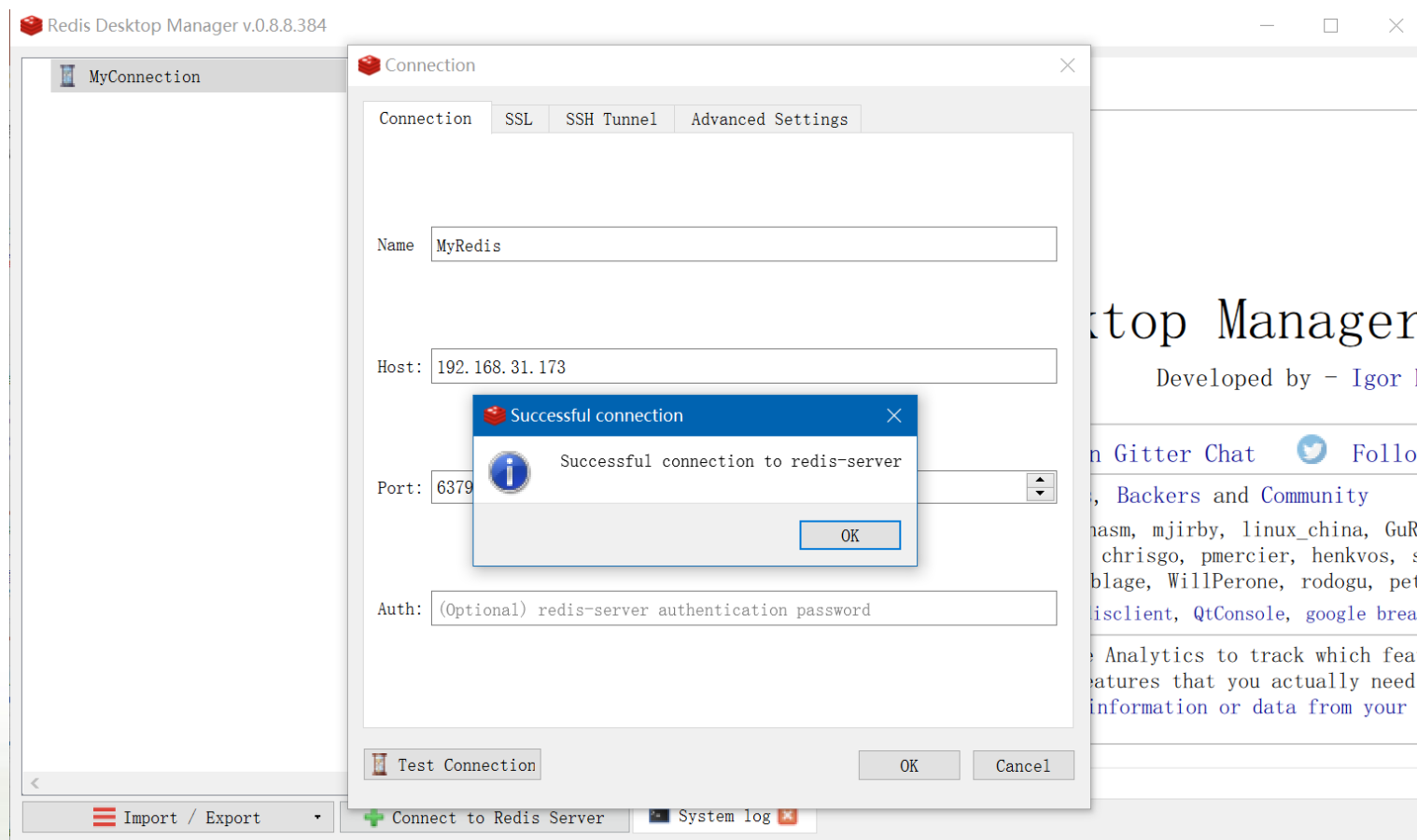
■ 为了能让远程客户机访问Redis服务，修改配置文件redis.windows-service.conf中的以下语句：

✧ bind 127.0.0.1修改为：bind
192.168.31.173(Redis所在主机地址)



安装Redis客户端

■ 安装Redis Desktop Manager, 连接Redis



3.Redis的整合支持

■ **Spring Boot与Redis整合所需依赖启动器如下（不需编写对应版本号信息，其版本号信息由Spring Boot统一管理）。**

```
<!-- Spring Data Redis依赖启动器 -->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>
```



4.基于注解的缓存实现案例

■ 本案例使用**Redis**缓存中间件，对7.2节的业务类**CommentServiceImpl**进行包装，使其具有业务数据缓存管理功能。

■ 搭建步骤：

- ✧①引入依赖启动器
- ✧②配置中间件连接
- ✧③配置序列化方式
- ✧④创建带缓存业务类
- ✧⑤改写控制器类
- ✧⑥效果测试



①引入启动器依赖

- 在chapter07项目的pom.xml中引入Spring Data Redis依赖启动器，提供对数据库Redis的数据操作支持。

```
<!-- Spring Data Redis依赖启动器 -->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>
```



②配置中间件连接

■ 在项目的全局配置文件 **application.properties** 中添加 **Redis** 中间件的连接配置，示例代码如下。

```
spring.redis.database = 0          #数据库名db0
spring.redis.host=192.168.31.173   # Redis服务器地址
spring.redis.port=6379             # Redis服务器连接端口
spring.redis.password=            # Redis服务器连接密码（默认为空）
spring.cache.type=redis            #启用redis/simple...缓存
spring.cache.redis.use-key-prefix=true
spring.redis.jedis.pool.max-idle=8  # 连接池的最大连接数
spring.redis.jedis.pool.min-idle=0  # 连接池的最小连接数
spring.redis.jedis.pool.max-active=8 #在给定时间连接池可以分配的最大连接数
spring.redis.jedis.pool.max-wait=-1ms # 当池被耗尽时等待分配连接的最长时间（毫秒）
```

③配置序列化方式

- 在chapter07项目com.scst.config包中创建一个自定义配置类RedisConfig，为RedisCache设置序列化方式，分别对缓存数据的key和value进行序列化方式定制，其中key定制为StringRedisSerializer（即String格式），value定制为Jackson2JsonRedisSerializer（即JSON格式）。



RedisConfig类

```
package com.scst.config;

import org.springframework.boot.autoconfigure.cache.CacheProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.cache.RedisCacheConfiguration;
import org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.RedisSerializationContext;
import org.springframework.data.redis.serializer.StringRedisSerializer;

@EnableConfigurationProperties(CacheProperties.class)
@EnableCaching
@Configuration // 定义一个配置类
public class RedisConfig {
```

RedisConfig类

@Bean

```
RedisCacheConfiguration redisCacheConfiguration(CacheProperties cacheProperties) {  
    RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig();  
    config = config.serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(new StringRedisSerializer()));  
    config = config.serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new GenericJackson2JsonRedisSerializer()));  
    CacheProperties.Redis redisProperties = cacheProperties.getRedis();  
    if (redisProperties.getTimeToLive() != null) {  
        config = config.entryTtl(redisProperties.getTimeToLive());  
    }  
    if (redisProperties.getKeyPrefix() != null) {  
        config = config.prefixKeysWith(redisProperties.getKeyPrefix());  
    }  
    if (!redisProperties.isCacheNullValues()) {  
        config = config.disableCachingNullValues();  
    }  
    if (!redisProperties.isUseKeyPrefix()) {  
        config = config.disableKeyPrefix();  
    }  
    return config;  
}
```

关于序列化

■ 在进行缓存管理时， **RedisCache**需要采用 **JdkSerializationRedisSerializer**默认方式对缓存对象（包括**key**和**value**）进行序列化，结果是以**HEX**格式进行存储，这种格式不方便缓存数据的可视化查看和管理，所以在实际开发中通常会为**RedisCache**另配置数据序列化方式。

✧如：采用**Jackson2JsonRedisSerializer**序列化，结果以**JSON**格式存储。



④创建带缓存业务类

- 在chapter07项目的com.scst.service.impl包中新建一个业务类CachedCommentService，在其中实现基于注解的业务数据缓存管理，包括查询缓存、更新缓存、删除缓存等。

✧ 缓存有效期可在全局配置文件中配置：

- `spring.cache.redis.time-to-live=60000ms` #统一设置有效期为1min

- 当对数据进行缓存管理时，为了避免与其他业务的缓存数据混淆，在对Comment数据缓存管理时，指定缓存的命名空间（`cacheName`）为“comment”。

CachedCommentService类

```
package com.scst.service.impl;

import com.baomidou.mybatisplus.core.metadata.IPage;
import com.scst.domain.Comment;
import com.scst.domain.CommentVO;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class CachedCommentService {

    @Autowired
    CommentServiceImpl commentService;
```


CachedCommentService类

// 多表条件分页查询

@Cacheable(cacheNames = "comment", unless = "#result==null")

```
public IPage<CommentVO> selectCommentVOByPage(Long currentPage, Long pageSize,
CommentVO commentVO){
    return commentService.selectCommentVOByPage(currentPage,pageSize,commentVO);
}
```

//查找指定id的记录详细信息

//只允许单线程查，防范缓存击穿

@Cacheable(cacheNames = "comment", sync = true)

```
public Comment getById(Integer id){
    return commentService.getById(id);
}
```

CachedCommentService类

//修改指定记录字段值

```
@CachePut(cacheNames = "comment")
```

```
public boolean saveOrUpdate(Comment comment){  
    return commentService.saveOrUpdate(comment);  
}
```

//存储记录

```
public boolean save(Comment comment){  
    return commentService.save(comment);  
}
```

//删除指定id的记录

```
@CacheEvict(cacheNames = "comment")
```

```
public boolean removeById(Integer id){  
    return commentService.removeById(id);  
}
```

```
}
```

⑤改写控制器类

- 将7.2节**CommentController**类中注入的**CommentServiceImpl**类型Bean实例改为**CachedCommentService**类型Bean实例。

@Autowired

```
//private CommentServiceImpl commentService;
```

```
private CachedCommentService commentService;
```



⑥效果测试

■ 启动项目，通过浏览器访问

`http://localhost:8080/comment/1/2`或

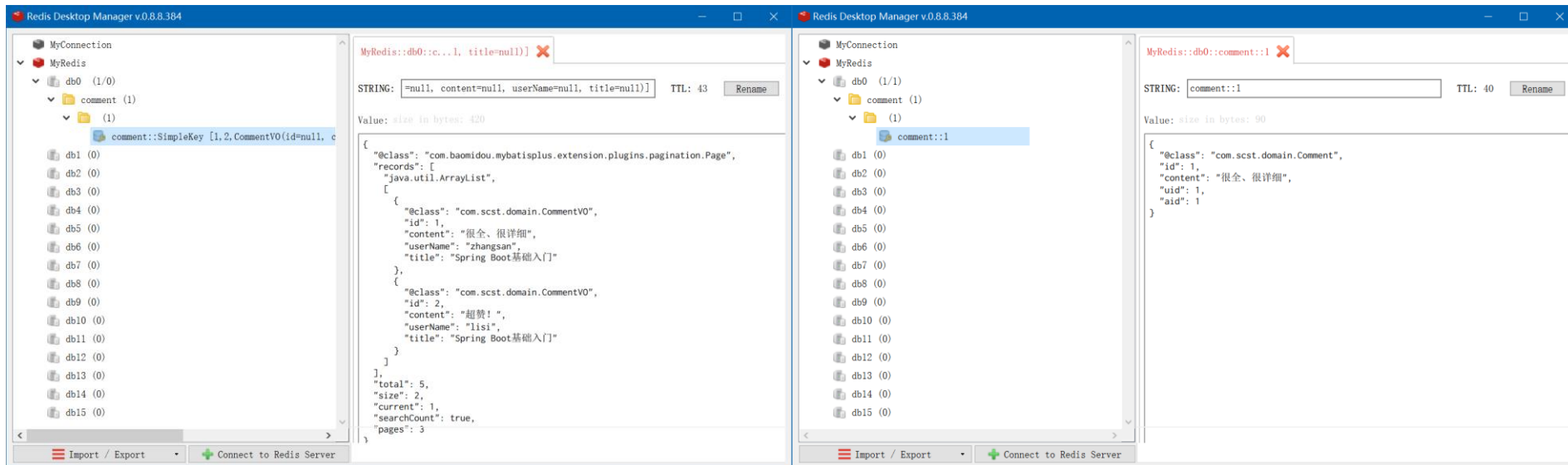
`http://localhost:8080/comment/1`，然后重复刷新浏览器或换用其他浏览器查询同一条数据信息，将观察到数据库只执行了一次SQL语句。

✧ 为了让**JSON**格式数据在浏览器中显示更美观，先在项目全局配置文件中配置以下属性：

- **`spring.jackson.serialization.indent-output=true`**



缓存结果



7.3.3 高并发下缓存失效问题

- 1.缓存穿透
- 2.缓存雪崩
- 3.缓存击穿
- 4.缓存数据一致性



1.缓存穿透

- **缓存穿透**是指查询一个一定不存在的数据，由于缓存是不命中，导致每次请求将去数据库查询，从而失去了缓存的意义。
- **风险：**
 - ✧ 利用不存在的数据进行攻击，数据库瞬时压力增大，最终导致崩溃
- **解决办法：**
 - ✧ 1) **null**结果缓存，并加入短暂过期时间
 - `spring.cache.redis.cache-null-values=true`
 - `spring.cache.redis.time-to-live=60000ms`
 - ✧ 2) 使用布隆过滤器，将所有可能存在的数据哈希到一个足够大的**bitmap**中，一个一定不存在的数据会被这个**bitmap**拦截掉，从而避免了对底层存储系统的查询压力。



2.缓存雪崩

■ **缓存雪崩**是指在我们设置缓存时**key**采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到**DB**，**DB**瞬时压力过重雪崩。

■ **解决办法：**

✧原有的失效时间基础上增加一个随机值，比如**1-5分钟**随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。



3.缓存击穿

■ 对于一些设置了过期时间的**key**，如果这些**key**可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。如果这个**key**在大量请求同时进来前正好失效，那么所有对这个**key**的数据查询都落到**db**，我们称为**缓存击穿**。

■ 解决办法：加锁

✧ 大量并发只让一个线程去查**db**，而其他线程等待，查到以后释放锁，其他线程获取到锁，先查缓存，就会有数据，这样就不用去**db**查询

- 例如：在@Cacheable注解属性中使用syn = true



4.缓存数据一致性

■ 缓存数据一致性问题即缓存中的数据与数据库中的数据不一致性问题。

■ 解决办法：

◇1) 双写模式：写完数据库同时更新缓存

◇2) 失效模式：写完数据库同时删除缓存，或数据过期删除缓存，下一次查询触发主动更新。

■ 在多并发不加锁条件下，上述两种模式只能保证最终一致性，但可使用Redisson分布式锁功能对缓存加读写锁，保证多线程一定能读到缓存中最新数据。

◇不过，实时性、一致性要求高的数据应直接查数据库，而不是采用加锁这样笨重方法。

关于Redisson

■ **Redisson**提供了强大的分布式锁功能，包括读写锁、公平锁、闭锁、信号量（**Semaphore**）等。

✧ 实现了看门狗机制，不用担心死锁问题。

- 如果业务超长，可对锁每隔**10s**再次自动续期至**30s**，直至业务完成后解锁
- 如果业务崩溃或不手动解锁，**30s**后自动解锁

✧ 推荐采用手动设置过期时间和手动解锁，效率更高

■ 可通过**Redisson**加读写锁保证多线程读写时排斥等待，写写时按顺序排队，读读时互不影响，即：

✧ 读+写：等待读锁释放才能写

✧ 写+读：等待写锁释放才能读

✧ 写+写：阻塞方式

✧ 读+读：相当于无锁，会并发读



本章小结

■ 本章具体讲解了：

✧ **7.1 SSMP**框架快速整合

✧ **7.2 SSMP**框架整合案例

✧ **7.3 SSMP**框架业务层缓存管理



