1.整合SpringMVC

刚才案例已经能实现mvc自动配置，这里我们主要解决以下3个问题

- 修改端口
- 静态资源
- 拦截器配置

修改端口

查看SpringBoot的全局属性可知，端口通过以下方式配置：

```
# 映射端口
server.port=8001
```

重启服务后测试：

访问静态资源

ResourceProperties的类，里面就定义了静态资源的默认查找路径：

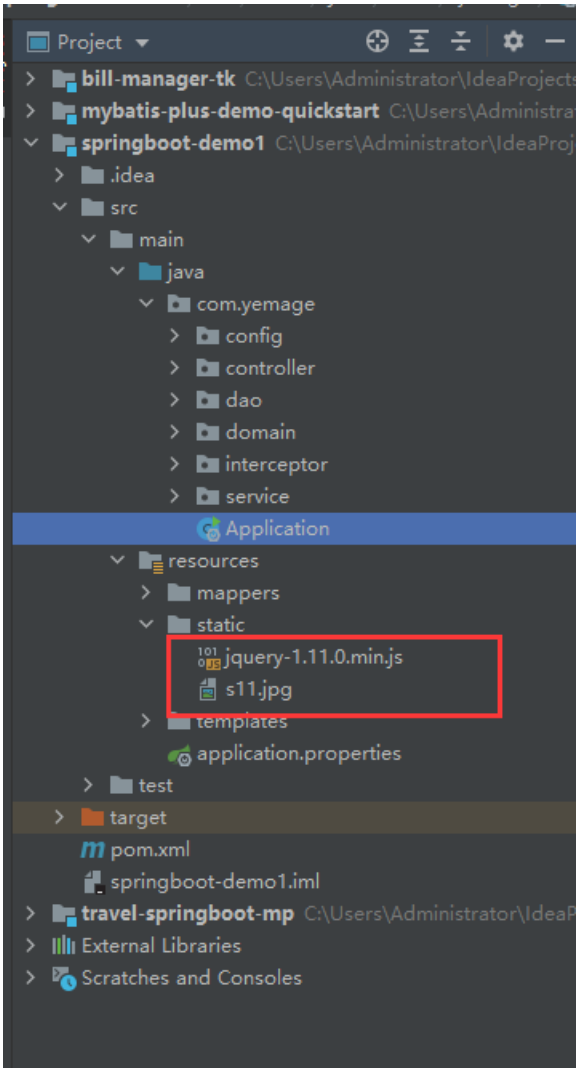
```
*/
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties {

    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
            "classpath:/META-INF/resources/", "classpath:/resources/",
            "classpath:/static/", "classpath:/public/" };
```

默认的静态资源路径为：

- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public

只要静态资源放在这些目录中任何一个，SpringMVC都会帮我们处理。

我们习惯会把静态资源放在`classpath:/static/`目录下。我们创建目录，并且添加一些静态资源：



重启项目后测试：

添加拦截器

> 首先我们定义一个拦截器:

```
public class LoginInterceptor implements HandlerInterceptor {

    private Logger logger = LoggerFactory.getLogger(LoginInterceptor.class);
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) {
        logger.debug("处理器执行前执行!");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) {
        logger.debug("处理器执行后执行!");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) {
        logger.debug("跳转后执行!");
    }
}
```

> 通过实现`WebMvcConfigurer`并添加`@Configuration`注解来实现自定义部分SpringMvc配置:

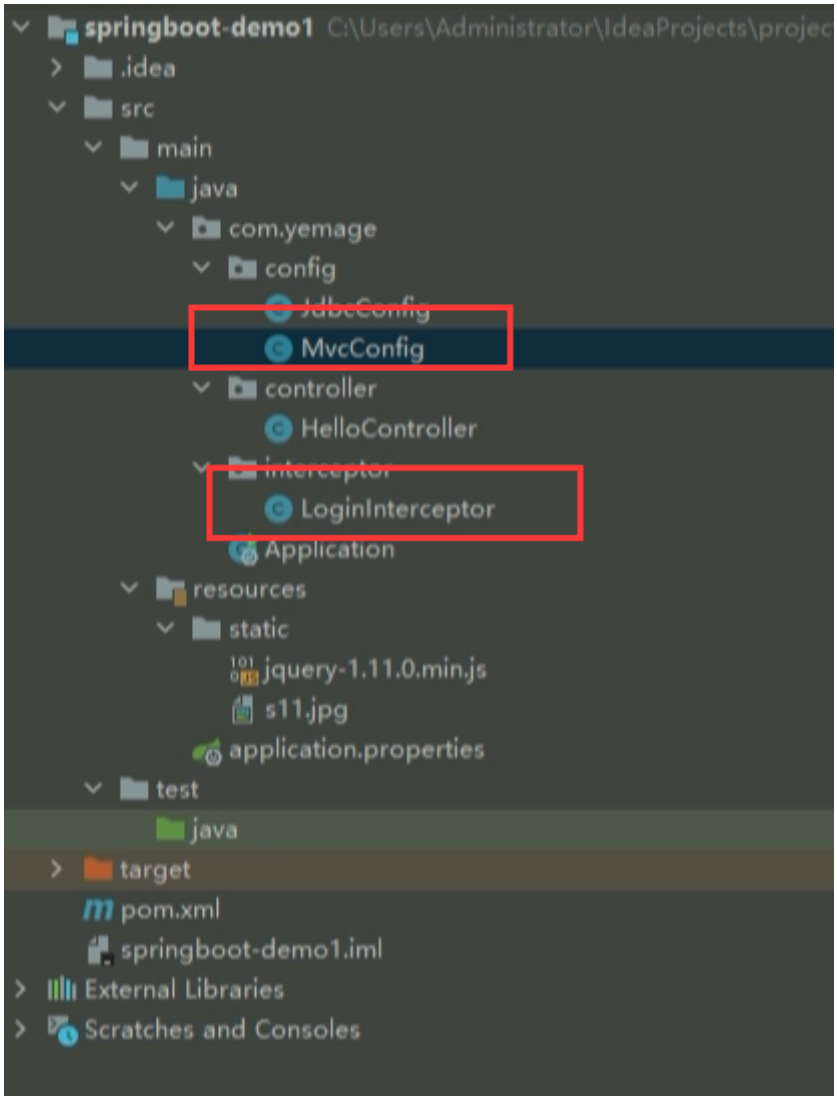
```
@Configuration
public class MvcConfig implements WebMvcConfigurer{
```

```
    /**
     * 通过@Bean注解，将我们定义的拦截器注册到Spring容器
     * @return
     */
    @Bean
    public LoginInterceptor loginInterceptor(){
        return new LoginInterceptor();
    }

    /**
     * 重写接口中的addInterceptors方法，添加自定义拦截器
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // 通过registry来注册拦截器，通过addPathPatterns来添加拦截路径
        registry.addInterceptor(this.loginInterceptor()).addPathPatterns("/**");
    }
}
```

ant path路径匹配通配符

- '?' 匹配任何单字符
- '*' 匹配0或者任意数量的字符
- '/**' 匹配0或者更多的目录

结构如下：



接下来运行并查看日志：

你会发现日志中什么都没有，因为我们记录的log级别是debug，默认是显示info以上，我们需要进行配置。

SpringBoot通过`logging.level.*=debug`来配置日志级别，*填写包名

```
# 设置com.yemage包的日志级别为debug
logging.level.com.yemage=debug
```

再次运行查看：

2.整合jdbc

> 导入资料中的t_user.sql文件

引入依赖

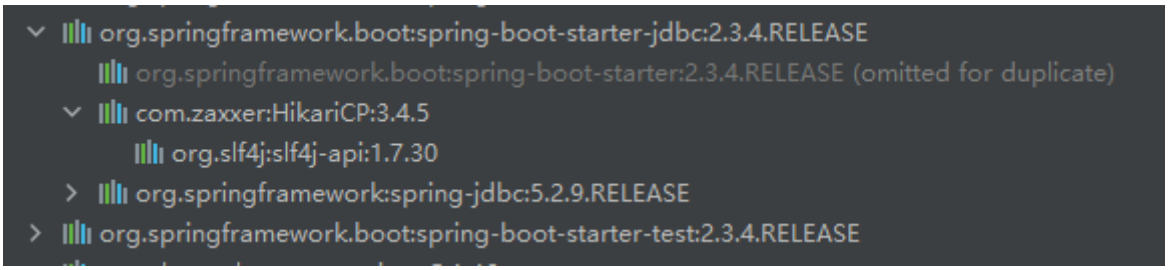
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

MySQL数据库驱动

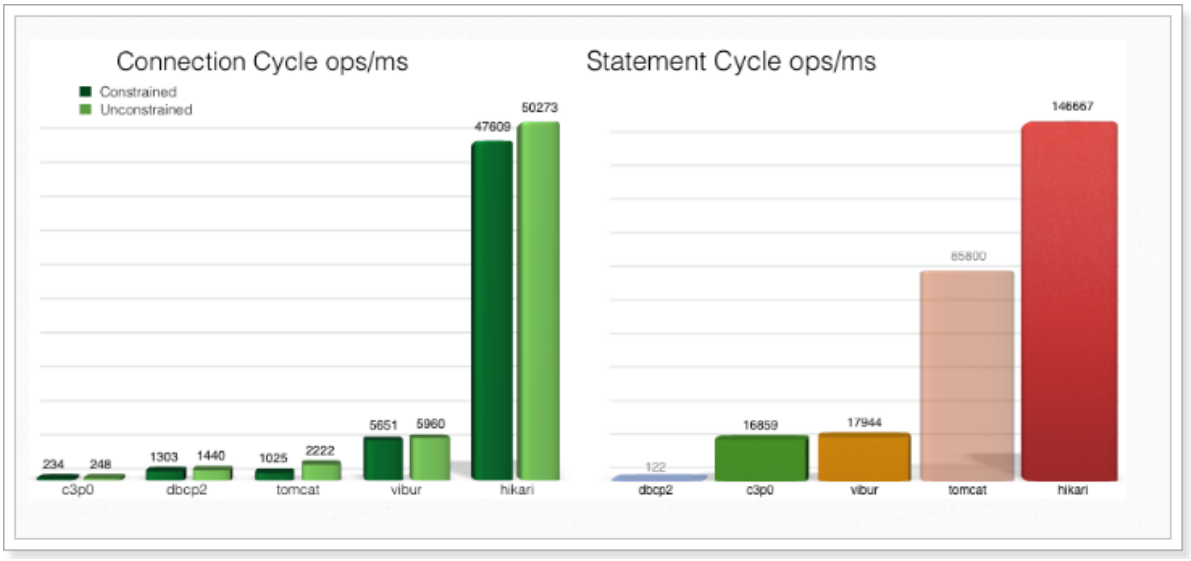
```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.46</version>
</dependency>
```

> 配置连接池

SpringBoot自动帮我们引入了一个连接池：



HikariCP应该是目前速度最快的连接池了，我们看看它与c3p0的对比：



因此，我们只需要指定连接池参数即可：

```
# 连接四大参数
spring.datasource.url=jdbc:mysql://localhost:3306/springboot
spring.datasource.username=root
spring.datasource.password=123456
# 可省略，SpringBoot自动推断
spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.hikari.idle-timeout=60000
spring.datasource.hikari.maximum-pool-size=30
spring.datasource.hikari.minimum-idle=10
```

> 实体类

```
public class User implements Serializable {

    private Long id;

    // 用户名
    //自动转换下换线到驼峰命名user_name -> userName
    private String userName;

    // 密码
    private String password;

    // 姓名
    private String name;

    // 年龄
    private Integer age;

    // 性别，1男性，2女性
    private Integer sex;

    // 出生日期
    private Date birthday;

    // 创建时间
    private Date created;

    // 更新时间
    private Date updated;

    // 备注
    private String note;
```

> dao

```
@Repository
public class JdbcDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List<User> findAll() {
        return jdbcTemplate.query("select * from tb_user", new
BeanPropertyRowMapper<>(User.class));
    }

}
```

> 测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class JdbcDaoTest {

    @Autowired
    private JdbcDao jdbcDao;

    @Test
    public void findAll() {
        List<User> list = jdbcDao.findAll();
        for (User user : list) {
            System.out.println(user);
        }
    }
}
```

3.整合mybatis

mybatis

引入pom.xml：

```
<!--mybatis -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>
```

application.properties配置：

```
# mybatis 别名扫描
mybatis.type-aliases-package=com.yemage.domain
# mapper.xml文件位置,如果没有映射文件，请注释掉
mybatis.mapper-locations=classpath:mappers/*.xml
```

> 实体类,直接使用jdbc用到的实体类

```
public class User {

    private Long id;

    // 用户名
    //自动转换下换线到驼峰命名user_name -> userName
    private String userName;

    // 密码
    private String password;

    // 姓名
    private String name;

    // 年龄
    private Integer age;

    // 性别，1男性，2女性
    private Integer sex;

    // 出生日期
    private Date birthday;

    // 创建时间
    private Date created;

    // 更新时间
    private Date updated;

    // 备注
    private String note;

    //getter和setter方法
```

接口

```
public interface UserDao {

    public List<User> findAll();

}
```

映射文件

```
<?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.yemage.demo.dao.UserDao">

    <select id="findAll" resultType="user">
        select * from tb_user
    </select>

</mapper>
```

Mapper的加载接口代理对象方式有2种

第一种：使用@Mapper注解(不推荐)

需要注意，这里没有配置mapper接口扫描包，因此我们需要给每一个Mapper接口添加`@Mapper`注解，才能被识别。

```
@Mapper
public interface UserMapper {
}
```

第二种设置MapperScan,注解扫描的包(推荐)

> @MapperScan("dao所在的包")，自动搜索包中的接口，产生dao的代理对象

```
@SpringBootApplication
@MapperScan("com.yemage.demo.dao")
public class Application {

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

}
```

测试

引入测试构建

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

测试代码

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserDaoTest {

    @Autowired
    private UserDao userDao;

    @Test
    public void testFindAll() {
        List<User> list = userDao.findAll();
    }

}
```

通用mapper

> 避免重复书写CRUD映射的框架有两个

- 通用mybatis（tk mybatis）
- mybatis plus，功能更加强大

通用Mapper的作者也为自己的插件编写了启动器，我们直接引入即可：

```
<!-- 通用mapper -->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.0.2</version>
</dependency>
```

实体类

> tk mybatis 实体类使用的注解是jpa注解

```
@Table(name = "tb_user")
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    // 用户名
    private String userName;

    ....
```

注意事项：

```
1. 默认表名=类名，字段名=属性名
    2. 表名可以使用`@Table(name = "tableName")`进行指定
    3. `@Column(name = "fieldName")`指定
    4. 使用`@Transient`注解表示跟字段不进行映射
```

不需要做任何配置就可以使用了。

```
@Mapper
public interface UserMapper extends tk.mybatis.mapper.common.Mapper<User>{

    public List<User> findByUser(User user);

}
```

自定义映射文件

> 映射复杂方法 resources/mappers/UserMapper.xml

```
<?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.yemage.demo.dao.UserMapper">

<select id="findByUser" resultType="user">
    SELECT
    *
    FROM
    tb_user
    <where>
        <if test="name != null">
            name like '%${name}%'
        </if>
        <if test="note != null">
            and note like '%${note}%'
        </if>
    </where>
</select>

</mapper>
```

Mapper所有的通用方法：

Select 方法：`List<T> select(T record);` 说明：根据实体中的属性值进行查询，查询条件使用等号

方法：`T selectByPrimaryKey(Object key);` 说明：根据主键字段进行查询，方法参数必须包含完整的主键属性，查询条件使用等号

方法：`List<T> selectAll();` 说明：查询全部结果，select(null)方法能达到同样的效果

方法：`T selectOne(T record);` 说明：根据实体中的属性进行查询，只能有一个返回值，有多个结果是抛出异常，查询条件使用等号

方法：`int selectCount(T record);` 说明：根据实体中的属性查询总数，查询条件使用等号

Insert 方法：`int insert(T record);` 说明：保存一个实体，null的属性也会保存，不会使用数据库默认值

方法：`int insertSelective(T record);` 说明：保存一个实体，null的属性不会保存，会使用数据库默认值

Update 方法：`int updateByPrimaryKey(T record);` 说明：根据主键更新实体全部字段，null值会被更新

方法：`int updateByPrimaryKeySelective(T record);` 说明：根据主键更新属性不为null的值

Delete 方法：`int delete(T record);` 说明：根据实体属性作为条件进行删除，查询条件使用等号

方法：`int deleteByPrimaryKey(Object key);` 说明：根据主键字段进行删除，方法参数必须包含完整的主键属性

Example方法 方法：`List<T> selectByExample(Object example);` 说明：根据Example条件进行查询 重点：这个查询支持通过`Example`类指定查询列，通过`selectProperties`方法指定查询列

方法：`int selectCountByExample(Object example);` 说明：根据Example条件进行查询总数

方法：`int updateByExample(@Param("record") T record, @Param("example") Object example);` 说明：根据Example条件更新实体`record`包含的全部属性，null值会被更新

方法：`int updateByExampleSelective(@Param("record") T record, @Param("example") Object example);` 说明：根据Example条件更新实体`record`包含的不是null的属性值

方法：`int deleteByExample(Object example);` 说明：根据Example条件删除数据

> 注意要把MapperScan类改成tk-mybatis构件的类

```
import tk.mybatis.spring.annotation.MapperScan;

@SpringBootApplication
@EnableConfigurationProperties
@MapperScan("com.yemage.demo.dao")
public class Application {
```

> 注意：必须使用tk mybatis的MapperScan

启动测试

> 测试类

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserDaoTest {

    @Autowired
    private UserDao userDao;

    @Test
    public void testFindByUser() {
```

```
        User condition = new User();
        condition.setName("a");
        List<User> list = userMapper.findByUser(condition);
        for (User user : list) {
            System.out.println(user);
        }
    }

    @Test
    public void testFindAll() {
        List<User> list = userDao.selectAll();
        for (User user : list) {
            System.out.println(user);
        }
    }

    @Test
    public void testFindById() {
        User user  = userDao.selectByPrimaryKey(4);
            System.out.println(user);
    }

    @Test
    public void testFindByExample() {
        Example example = new Example(User.class);
        example.createCriteria().andLike("name", "%a%");
        userMapper.selectByExample(example).forEach(user -> {
            System.out.println(user);
        });
    }

    @Test
    public void testInsert() {
        User user = new User();
        user.setAge(18);
        user.setBirthday(new Date());
        user.setCreated(new Date());
        user.setName("周星驰");
        userDao.insert(user);
    }
```

4.Thymeleaf

> 概念

Thymeleaf 是一个跟 FreeMarker 类似的模板引擎，它可以完全替代 JSP 。相较与其他的模板引擎，它有如下特点：

- 动静结合：Thymeleaf 在有网络和无网络的环境下皆可运行，无网络显示静态内容，有网络用后台得到数据替换静态内容
- 与SpringBoot完美整合，springboot默认整合thymeleaf

4.1 入门案例

编写接口

UserService

```
@Service
public class UserService {

    @Autowired
    private UserDao userDao;

    public List<User> queryAll() {
        return this.userDao.selectAll();
    }
}
```

Controller

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping("/all")
    public String all(Model model) {
        List<User> list = userService.findAll();
        model.addAttribute("users", list);
        // 返回模板名称（就是classpath:/templates/目录下的html文件名）
        return "users";
    }

}
```

引入启动器

直接引入启动器：

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

SpringBoot会自动为Thymeleaf注册一个视图解析器：

```
 * </p>
 *
 * @author Daniel Fern&aacute;ndez
 *
 * @since 3.0.3
 *
 */
public class ThymeleafViewResolver
        extends AbstractCachingViewResolver
        implements Ordered {
```

与解析JSP的InternalViewResolver类似，Thymeleaf也会根据前缀和后缀来确定模板文件的位置：

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

    private static final Charset DEFAULT_ENCODING = StandardCharsets.UTF_8;

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

    public static final String DEFAULT_SUFFIX = ".html";

    /**
```

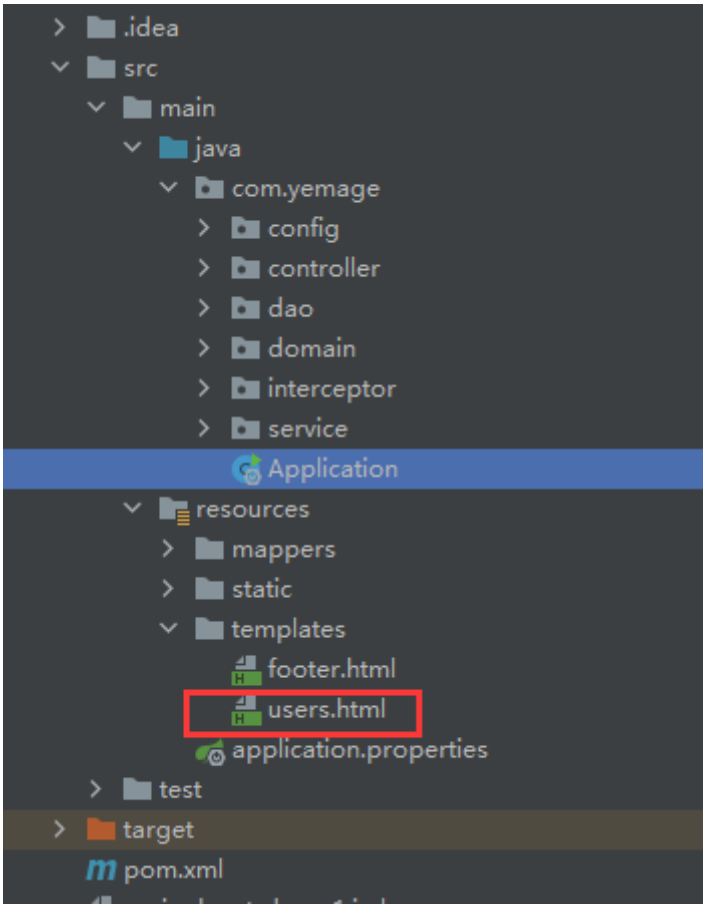
- 默认前缀：`classpath:/templates/`
- 默认后缀：`.html`

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

一般我们无需进行修改，默认即可。

静态页面

根据上面的文档介绍，模板默认放在classpath下的templates文件夹，我们新建一个html文件放入其中：



编写html模板，渲染模型中的数据：

注意，把html 的名称空间，改成：`xmlns:th="http://www.thymeleaf.org"` 会有语法提示

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>首页</title>
    <style type="text/css">
        table {border-collapse: collapse; font-size: 14px; width: 80%; margin:
auto}
        table, th, td {border: 1px solid darkslategray;padding: 10px}
    </style>
</head>
<body>
<div style="text-align: center">
    <span style="color: darkslategray; font-size: 30px">欢迎光临！</span>
    <hr/>
    <table class="list">
        <tr>
            <th>id</th>
            <th>姓名</th>
            <th>用户名</th>
            <th>年龄</th>
            <th>性别</th>
            <th>生日</th>
            <th>备注</th>
            <th>操作</th>
        </tr>
        <tr th:each="user, status : ${users}" th:object="${user}">
            <td th:text="${user.id}">1</td>
```

```
            <td th:text="*{name}">张三</td>
            <td th:text="*{userName}">zhangsan</td>
            <td th:text="${user.age}">20</td>
            <td th:text="${user.sex} == 1 ? '男': '女'">男</td>
            <td th:text="${#dates.format(user.birthday, 'yyyy-MM-dd')}">1980-02-
30</td>
            <td th:text="${user.note}">1</td>
            <td>
                <a th:href="@{/delete(id=${user.id}, userName=*{userName})}">删除
</a>
                <a th:href="|/update/${user.id}|">修改</a>
                <a th:href="'/approve/' + ${user.id}">审核</a>
            </td>
        </tr>

    </table>
</div>
</body>
</html>
```

我们看到这里使用了以下语法：

- `${}`：这个类似与el表达式，但其实是ognl的语法，比el表达式更加强大
- `th-`指令：`th-`是利用了Html5中的自定义属性来实现的。如果不支持H5，可以用`data-th-`来代替
  - `th:each`：类似于`c:foreach`遍历集合，但是语法更加简洁
  - `th:text`：声明标签中的文本
    - 例如`<td th-text='${user.id}'>1</td>`，如果user.id有值，会覆盖默认的1
    - 如果没有值，则会显示td中默认的1。这正是thymeleaf能够动静结合的原因，模板解析失败不影响页面的显示效果，因为会显示默认值！

测试

接下来，我们打开页面测试一下：

模板缓存

Thymeleaf会在第一次对模板解析之后进行缓存，极大的提高了并发处理能力。但是这给我们开发带来了不便，修改页面后并不会立刻看到效果，我们开发阶段可以关掉缓存使用：

```
# 开发阶段关闭thymeleaf的模板缓存
spring.thymeleaf.cache=false
```

注意：

在Idea中，我们需要在修改页面后按快捷键：`Ctrl + Shift + F9` 对项目进行rebuild才可以。

我们可以修改页面，测试一下。

4.2 thymeleaf详解

表达式

它们分为三类

1. 变量表达式
2. 选择或星号表达式
3. URL表达式

> 变量表达式

变量表达式即OGNL表达式或Spring EL表达式(在Spring中用来获取model attribute的数据)。如下所示：

```
${session.user.name}
```

它们将以HTML标签的一个属性来表示：

```
    <h5>表达式</h5>
    <span>${text}</span>
    <span th:text="${text}">你好 thymleaf</span>
```

> 选择(星号)表达式

选择表达式很像变量表达式，不过它们用一个预先选择的对象来代替上下文变量容器(map)来执行，如下：`*{customer.name}`

被指定的object由th:object属性定义：

users.html

```
<tr th:each="user : ${users}" th:object="${user}">
    <td th:text="${user.id}">1</td>
    <td th:text="*{name}">张三</td>
    <td th:text="*{userName}">zhangsan</td>
    ....
```

> URL表达式

URL表达式指的是把一个有用的上下文或回话信息添加到URL，这个过程经常被叫做URL重写。`@{/order/list}` URL还可以设置参数：`@{/order/details(id=${orderId}, name=*{name})}` 相对路径：`@{../documents/report}`

让我们看这些表达式：

```
    <form th:action="@{/createOrder}">
    <a href="main.html" th:href="@{/main}">
```

url表达式

```
<a th:href="@{/delete(id=${user.id}, userName=*{userName})}">删除</a>
```

文本替换

```
<a th:href="|/update/${user.id}|">修改</a>
```

字符串拼接

```
<a th:href="'/approve/' + ${user.id}">审核</a>
```

表达式常见用法

> 字面（Literals）

- 文本文字（Text literals）：`'one text', 'Another one!',…`
- 数字文本（Number literals）：`0, 34, 3.0, 12.3,…`
- 布尔文本（Boolean literals）：`true, false`
- 空（Null literal）：`null`
- 文字标记（Literal tokens）：`one, sometext, main,…`

> 文本操作（Text operations）

- 字符串连接(String concatenation): `+`
- 文本替换（Literal substitutions）：`|The name is ${name}|`

> 算术运算（Arithmetic operations）

- 二元运算符（Binary operators）：`+, -, *, /, %`
- 减号（单目运算符）Minus sign (unary operator): `-`

> 布尔操作（Boolean operations）

- 二元运算符（Binary operators）：`and, or`
- 布尔否定（一元运算符）Boolean negation (unary operator): `!, not`

> 比较和等价(Comparisons and equality)

- 比较（Comparators）：`>, <, >=, <= (gt, lt, ge, le)`
- 等值运算符（Equality operators）：`==, != (eq, ne)`

> 条件运算符（Conditional operators）

- If-then: `(if) ? (then)`
- If-then-else: `(if) ? (then) : (else)`
- Default: (value) ?: `(defaultvalue)`

常用th标签

关键字	功能介绍	案例
th:text	文本替换	< p th:text="${collect.description}">description< /p >
th:id	替换id	< input th:id="'xxx' + ${collect.id}"/ >
th:utext	支持html的文本替换	< p th:utext="${htmlcontent}">conten< /p >
th:object	替换对象	< div th:object="${session.user}">
th:value	属性赋值	< input th:value="${user.name}" />
th:with	变量赋值运算	< div th:with="isEven=${prodStat.count}%2==0">< /div>
th:style	设置样式	th:style="'display:' + @{(${sitrue} ? 'none' : 'inline-block')} + '"'
th:onclick	点击事件	th:onclick="'getCollect()'"
th:each	属性赋值	tr th:each="user,userStat:${users}">
th:if	判断条件	< a th:if="${userId == collect.userId}" >
th:unless	和th:if判断相反	<a th:href="@{/login}" th:unless=${session.user != null}>Login
th:href	链接地址	<a th:href="@{/login}" th:unless=${session.user != null}>Login />
th:switch	多路选择配合 th:case 使用	< div th:switch="${user.role}">
th:case	th:switch的一个分支	< p th:case="'admin'">User is an administrator< /p>
th:fragment	布局标签，定义一个代码片段，方便其它地方引用	< div th:fragment="alert">
th:include	布局标签，替换内容到引入的文件	< head th:include="layout :: htmlhead" th:with="title='xx'">< /head>
th:replace	布局标签，替换整个标签到引入的文件	< div th:replace="fragments/header :: title">< /div>

关键字	功能介绍	案例
th:selected	selected 选择框 选中	th:selected="($xxx.id$ ==${configObj.dd})"
th:src	图片类地址引入	< img class="img-responsive" alt="App Logo" th:src="@{/img/logo.png}" />
th:inline	定义js脚本可以使用变量	< script type="text/javascript" th:inline="javascript">
th:action	表单提交的地址	< form action="subscribe.html" th:action="@{/subscribe}">
th:remove	删除某个属性	< tr th:remove="all"> 1.all:删除包含标签和所有的孩子。2.body:不包含标记删除,但删除其所有的孩子。3.tag:包含标记的删除,但不删除它的孩子。4.all-but-first:删除所有包含标签的孩子,除了第一个。5.none:什么也不做。这个值是有用的动态评估。
th:attr	设置标签属性，多个属性可以用逗号分隔	比如 th:attr="src=@{/image/aa.jpg},title=#{logo}"，此标签不太优雅，一般用的比较少。

还有非常多的标签，这里只列出最常用的几个

基本用法

1. 赋值、字符串拼接

字符串拼接还有另外一种简洁的写法

```
<a th:href="|/update/${user.id}|">修改</a>
<a th:href="'/approve/' + ${user.id}">审核</a>
```

2. 条件判断 If/Unless

Thymeleaf中使用th:if和th:unless属性进行条件判断，下面的例子中，`<a>`标签只有在`th:if`中条件成立时才显示:

```
    <h5>if指令</h5>
    <a th:if="${users.size() > 0}">查询结果存在</a><br>
    <a th:if="${users.size() <= 0}">查询结果不存在</a><br>
    <a th:unless="${session.user != null}" href="#">登录</a><br>
```

th:unless于th:if恰好相反，只有表达式中的条件不成立，才会显示其内容。

也可以使用`(if) ? (then) : (else)`这种语法来判断显示的内容

3. for 循环

```
        <tr th:each="user, status : ${users}" th:object="${user}"
th:class="${status.even} ? 'grey'">
            <td th:text="${status.even}"></td>
            <td th:text="${user.id}">1</td>
            <td th:text="*{name}">张三</td>
            <td th:text="*{userName}">zhangsan</td>
            <td th:text="${user.age}">20</td>
            <td th:text="${user.sex} == 1 ? '男' : '女'">男</td>
            <td th:text="${#dates.format(user.birthday, 'yyyy-MM-dd')}">1980-02-
30</td>
            <td th:text="${user.note}">1</td>
            <td>
                <a th:href="@{/delete(id=${user.id}, userName=*{userName})}">删除
</a>
                <a th:href="|/update/${user.id}|">修改</a>
                <a th:href="'/approve/' + ${user.id}">审核</a>
            </td>
        </tr>
```

status称作状态变量，属性有：

- index:当前迭代对象的index（从0开始计算）
- count: 当前迭代对象的index(从1开始计算)
- size:被迭代对象的大小
- current:当前迭代变量
- even/odd:布尔值，当前循环是否是偶数/奇数（从0开始计算）
- first:布尔值，当前循环是否是第一个
- last:布尔值，当前循环是否是最后一个

4. 内联文本

内联文本：[[...]]内联文本的表示方式，使用时，必须先用th:inline="text/javascript/none"激活，th:inline可以在父级标签内使用，甚至作为body的标签。内联文本尽管比th:text的代码少，不利于原型显示。

> 在thymeleaf指令中显示

```
<h6 th:text="${text}">静态内容</h6>
```

> 使用内联文本显示model attribute

```
    <h5>内联文本</h5>
    <div>
        <h6 th:inline="text">[[${text}]]</h6>
        <h6 th:inline="none">[[${text}]]</h6>
        <h6>[[${text}]]</h6>
    </div>
```

> 原则能用指令就用th指令

5. 内联js

内联文本：[[...]]内联文本的表示方式，使用时，必须先用th:inline="text/javascript/none"激活，th:inline可以在父级标签内使用，甚至作为body的标签。内联文本尽管比th:text的代码少，不利于原型显示。

```
    <h5>内联js</h5>
    <script th:inline="javascript">
        /*<![CDATA[*/
            var text = '[[${text}]]';
            alert(text);
        /*]]>*/
    </script>
```

6. 内嵌变量

为了模板更加易用，Thymeleaf还提供了一系列Utility对象（内置于Context中），可以通过#直接访问：

- dates： java.util.Date的功能方法类。
- calendars : 类似#dates，面向java.util.Calendar
- numbers : 格式化数字的功能方法类
- strings : 字符串对象的功能类，contains,startWiths,prepending/appending等等。
- objects: 对objects的功能类操作。
- bools: 对布尔值求值的功能方法。
- arrays：对数组的功能类方法。
- lists: 对lists功能类方法
- sets
- maps ...

下面用一段代码来举例一些常用的方法：

> dates

```
    <h5>内置变量</h5>
    <h6 th:text="${#dates.createNow()}">获取当前日期</h6>
```

> strings

```
    <h5>内置变量</h5>
    <h6 th:text="${#dates.createNow()}">获取当前日期</h6>
    <h6 th:text="${#strings.substring(text, 6, 9)}">截取字符串</h6>
    <h6 th:text="${#strings.length(text)}">获得长度</h6>
    <h6 th:text="${#strings.randomAlphanumeric(6)}">随机字符串</h6>
    <h6 th:text="${#strings.equals(text, 'hello text....')}"></h6>
```

使用thymeleaf布局

使用thymeleaf布局非常的方便

在/resources/templates/目录下创建footer.html，内容如下

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <footer th:fragment="copy(title)">
        &copy; 2021 野马哥版权所有<br>
        <span th:text="${title}">title footer</span>
    </footer>
</body>
</html>
```

在页面任何地方引入：

```
    <h5>thymeleaf布局</h5>
    <div th:insert="footer :: copy('野马哥1')"></div>
    <div th:replace="footer :: copy('野马哥2')"></div>
    <div th:include="footer :: copy('野马哥3')"></div>
```

- th:insert：保留自己的主标签，保留th:fragment的主标签。
- `th:replace`：不要自己的主标签，保留th:fragment的主标签。
- `th:include`：保留自己的主标签，不要th:fragment的主标签。（官方3.0后不推荐）

返回的HTML如下：

```
    <h5>thymeleaf布局</h5>
    <div><footer>
        &copy; 2021 野马哥版权所有<br>
        <span>野马哥1</span>
    </footer></div>
    <footer>
        &copy; 2021 野马哥版权所有<br>
        <span>野马哥2</span>
    </footer>
    <div>
        &copy; 2021 野马哥版权所有<br>
        <span>野马哥3</span>
    </div>
```

5. Mybatis Plus

> 简介

> Mybatis-Plus（简称MP）是一个 Mybatis 的增强工具，在 Mybatis 的基础上只做增强不做改变，避免了我们重复CRUD语句。

快速入门

> 创建工程，引入依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<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>

    <groupId>com.yemage</groupId>
    <artifactId>mybatis-plus-demo-quickstart</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.0.RELEASE</version>
        <relativePath/>
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
        <mybatisplus.version>3.3.2</mybatisplus.version>
        <skipTests>true</skipTests>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
```

```
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-boot-starter</artifactId>
            <version>${mybatisplus.version}</version>
        </dependency>

        <dependency>
            <groupId>org.assertj</groupId>
            <artifactId>assertj-core</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>


    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

> 配置文件application.yml

yml配置简介

在Springboot中，推荐使用properties或者YAML文件来完成配置，但是对于较复杂的数据结构来说，YAML又远远优于properties。我们快速介绍YAML的常见语法格式。

先来看一个Springboot中的properties文件和对应YAML文件的对比：

```
#properties(示例来源于Springboot User guide):

environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```

可以明显的看到，在处理层级关系的时候，properties需要使用大量的路径来描述层级（或者属性），比如environments.dev.url和environments.dev.name。其次，对于较为复杂的结构，比如数组（my.servers），写起来更为复杂。而对应的YAML格式文件就简单很多：

```
#YAML格式
environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My Cool App
my:
    servers:
        - dev.bar.com
        - foo.bar.com
```

application.yml

```
# DataSource Config
spring:
  datasource:
    driver-class-name: org.h2.Driver
    schema: classpath:db/schema-h2.sql
    data: classpath:db/data-h2.sql
    url: jdbc:h2:mem:test
    username: root
    password: test

# Logger Config
logging:
  level:
    com.yemage.quickstart: debug
```

数据库脚本文件/db/data-h2.sql和/db/schema-h2.sql（拷贝）

h2数据库是一个基于内存的数据库，在jvm启动时，自动执行脚本加载相应的数据

springboot 中使用h2数据库直接按照上面配置，配置schema表结构脚本和data数据脚本即可

注意这里用户名密码可以省略不写，或者随意设定

> 启动类

```
@SpringBootApplication
@MapperScan("com.yemage.quickstart.mapper")
public class QuickstartApplication {

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

}
```

实体类

```
@Data
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

> dao

```
public interface UserMapper extends BaseMapper<User> {

}
```

> 测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SampleTest {

    @Resource
    private UserMapper userMapper;

    @Test
    public void testSelect() {
        System.out.println(("----- selectAll method test ------"));
        List<User> userList = userMapper.selectList(null);
        Assert.assertEquals(6, userList.size());
        userList.forEach(System.out::println);
    }
}
```

常用注解

> MyBatisPlus提供了一些注解供我们在实体类和表信息出现不对应的时候使用。通过使用注解完成逻辑上匹配。

注解名称	说明
`@TableName`	实体类的类名和数据库表名不一致
`@TableId`	实体类的主键名称和表中主键名称不一致
`@TableField`	实体类中的成员名称和表中字段名称不一致>

> mybatis plus注解策略配置

如果mysql自增主键注解策略设置如下

```
@TableId(type = IdType.AUTO)
private Long id;
```

默认主键策略

```
/**
* 采用雪花算法生成全局唯一主键
**/
ASSIGN_ID(3),
```

> 排除实体类中非表字段

使用`@TableField(exist = false)`注解

> 主键策略参考源码IdType

内置增删改查

> 测试

```
    @Test
    public void testInsert() {
        User user = new User();
        user.setName("野马哥");
        user.setEmail("yemage@163.com");
        user.setAge(3);
        Assert.assertTrue(mapper.insert(user) > 0);
        mapper.selectList(null).forEach(System.out :: println);
    }

    @Test
    public void testDelete() {
//        //主键删除
//        mapper.deleteById(3l);
//        mapper.selectList(null).forEach(System.out :: println);

//        //批量删除：1
//        mapper.delete(new QueryWrapper<User>().like("name", "J"));
//        mapper.selectList(null).forEach(System.out :: println);

//        //批量删除：2
//        mapper.delete(Wrappers.<User>query().like("name", "J"));
//        mapper.selectList(null).forEach(System.out :: println);

        //批量删除：2
        mapper.delete(Wrappers.<User>query().lambda().like(User::getName, "J"));
        mapper.selectList(null).forEach(System.out :: println);
    }

    @Test
    public void testUpdate() {
//        //基本修改
//        mapper.updateById(new User().setId(1l).setName("慧科"));
//        mapper.selectList(null).forEach(System.out :: println);

//        //批量修改：1
//        mapper.update(null, Wrappers.<User>update().set("email",
"huike@163.com").like("name", "J"));
//        mapper.selectList(null).forEach(System.out :: println);

        //批量修改：2
```

```
        mapper.update(new User().setEmail("huike@163.com"), Wrappers.
<User>update().like("name", "J"));
        mapper.selectList(null).forEach(System.out :: println);

    }

    @Test
    public void testSelect() {
//        //基本查询
//        System.out.println(mapper.selectOne(Wrappers.<User>query().eq("name",
"Tom")));

        //投影查询
        mapper.selectList(new QueryWrapper<User>().select("id",
"name")).forEach(user -> {
            System.out.println(user);
        });
    }
```

分页

内置分页

```
@Configuration
public class MybatisPlusConfig {

    /**
     * 分页插件
     */
    @Bean
    public PaginationInterceptor paginationInterceptor() {
        // 开启 count 的 join 优化,只针对 left join !!!
        return new PaginationInterceptor().setCountSqlParser(new
JsqlParserCountOptimize(true));
    }

}
```

> 优化left join count场景

在一对一join操作时，也存在优化可能，看下面sql

```
select u.id,ua.account from user u left join user_account ua on u.id=ua.uid
#本来生成的count语句像这样
select count(1) from (select u.id,ua.account from user u left join user_account
ua on u.id=ua.uid)
```

这时候分页查count时，其实可以去掉left join直查user，因为user与user_account是1对1关系，如下：

```
查count:
select count(1) from user u
查记录:
select u.id,ua.account from user u left join user_account ua on u.id=ua.uid limit
0,50
```

> 测试

```
    @Test
    public void testPage() {
        System.out.println("------ baseMapper 自带分页 ------");
        Page<User> page = new Page<>(1, 5);
        IPage<User> pageResult = mapper.selectPage(page, new QueryWrapper<User>
().eq("age", 20));
        System.out.println("总条数 ------> " + pageResult.getTotal());
        System.out.println("当前页数 ------> " + pageResult.getCurrent());
        System.out.println("当前每页显示数 ------> " + pageResult.getSize());
        pageResult.getRecords().forEach(System.out :: println);
    }
```

自定义xml分页

application.yml配置文件

```
# 配置mybatis plus
mybatis-plus:
  type-aliases-package: com.yemage.crud.entity #别名搜索
  mapper-locations: classpath:/mappers/*.xml #加载映射文件
```

UserMapper接口

```
public interface UserMapper extends BaseMapper<User> {

    /**
     * 如果映射的接口方法有2个参数需要@Param定义参数名，定义参数名后，映射文件中使用p.属性 c.
属性，具体访问
     *
     * @param page
     * @param conditioin
     * @return
     */
    public IPage<User> selectUserByPage(@Param("p") IPage<User> page,
@Param("c") User conditioin);

}
```

UserMapper.xml映射文件

```
<?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.yemage.mybatisplus.samples.crud.mapper.UserMapper">

    <sql id="selectSql">
        SELECT
            *
        FROM
            user
    </sql>
```

```
    <select id="selectUserByPage" resultType="user">
        <include refid="selectSql"></include>
        <where>
            <if test="c.age !=null">
                age = #{c.age}
            </if>
            <if test="c.email !=null">
                and email like '%${c.email}%'
            </if>
        </where>
    </select>

</mapper>
```

测试

```
@Test
public void testXmlPage() {
    System.out.println("------ baseMapper 自定义xml分页 ------");
    Page<User> page = new Page<>(1, 5);
    User user = new User();
    user.setAge(20);
    user.setEmail("test");
    IPage<User> pr = mapper.selectUserByPage(page, user);
    System.out.println("总条数 ------> " + pr.getTotal());
    System.out.println("当前页数 ------> " + pr.getCurrent());
    System.out.println("当前每页显示数 ------> " + pr.getSize());
    pr.getRecords().forEach(System.out :: println);
}
```

pageHelper分页

> 引入pageHelper依赖

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>5.1.11</version>
</dependency>
```

> mybatis plus 整合pageHelper的配置类

```
@Configuration
@MapperScan("com.yemage.mybatisplus.samples.crud.mapper")
public class MybatisPlusConfig {

    /**
     * mp分页插件
     */
    @Bean
    public PaginationInterceptor paginationInterceptor() {
        // 开启 count 的 join 优化,只针对 left join !!!
```

```
        return new PaginationInterceptor().setCountSqlParser(new
JsqlParserCountOptimize(true));
    }

    /**
     * 两个分页插件都配置,不会冲突
     * pagehelper的分页插件
     */
    @Bean
    public PageInterceptor pageInterceptor() {
        return new PageInterceptor();
    }

}
```

映射文件

```
<select id="selectUserByPage2" resultType="user">
    <include refid="selectSql"></include>
    <where>
        <if test="age !=null">
            age = #{age}
        </if>
        <if test="email !=null">
            and email like '%${email}%'
        </if>
    </where>
</select>
```

测试

```
    @Test
    public void testPageHelper() {
        // pagehelper
//        PageInfo<User> page = PageHelper.startPage(1, 2).doSelectPageInfo(() ->
mapper.selectList(Wrappers.<User>query()));
        PageHelper.startPage(1,2);
//        PageInfo<User> page = new PageInfo<>(mapper.selectList(Wrappers.
<User>query()));

        User u = new User();
        u.setAge(20);
        PageInfo<User> page = new PageInfo<User>(mapper.selectUserByPage2(u));

        List<User> list = page.getList();

        System.out.println("总行数=" + page.getTotal());
        System.out.println("当前页=" + page.getPageNum());
        System.out.println("每页行数=" + page.getPageSize());
        System.out.println("总页数=" + page.getPages());
        System.out.println("起始行数=" + page.getStartRow());

        System.out.println("是第一页=" + page.isIsFirstPage());
        System.out.println("是最后页=" + page.isIsLastPage());

        System.out.println("还有下一页=" + page.isHasNextPage());
```

```
        System.out.println("还有上一页=" + page.isHasPreviousPage());
        System.out.println("页码列表" +
Arrays.toString(page.getNavigatepageNums()));

    }
```