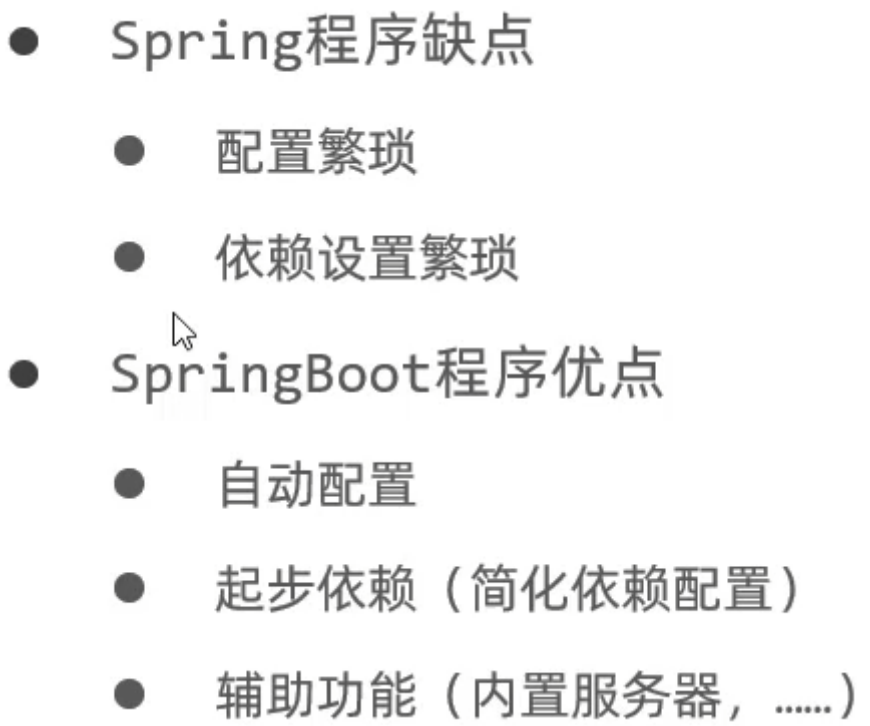
SpringBoot和Mybatis-Plus

SpringBoot

目的：简化Spring应用的初始搭建以及开发过程



- 起步依赖

团队项目开发要指定好使用哪个版本号

- SpringBoot在创建项目时，采用jar的打包方式
- SpringBoot的引导类是项目的入口，运行main方法就可以启动项目

基础配置

修改服务器端口号

加载配置文件，properties优先级最高

yaml、yml格式

- YAML（YAML Ain't Markup Language），一种数据序列化格式
- 优点：
  - 容易阅读
  - 容易与脚本语言交互
  - 以数据为核心，重数据轻格式
- YAML文件扩展名
  - **.yml（主流）**
  - .yaml

```
enterprise.name=itcast
enterprise.age=16
enterprise.tel=4006184000
```
Properties

```
enterprise:
  name: itcast
  age: 16
  tel: 4006184000
```
yaml

语法规范使用

- 大小写敏感
- 属性层级关系使用多行描述，每行结尾使用冒号结束
- 使用缩进表示层级关系，同层级左侧对齐，只允许使用空格（不允许使用Tab键）
- 属性值前面添加空格（属性名与属性值之间使用冒号+空格作为分隔）
- # 表示注释

数组数据

- 数组数据在数据书写位置的下方使用减号作为数据开始符号，每行书写一个数据，减号与数据间空格分隔

读取配置文件的快捷方法（熟记）

Yaml

1. 第一种：使用@Value注解加${}去获取
2. 定义一个environment类

使用他的方法来直接读取里面的内容，遍历所有东西

1. pojo类

添加到Bean中，读取配置文件，作用：做数据配置

实体类上加上这两个注解

```
@Component
@ConfigurationProperties(prefix = "user")
```

在yaml中搞出三种配置环境：

```yaml
spring:
  profiles:
    active: pro
---
spring:
  profiles: pro

server:
  port: 80
---
spring:
  profiles: dev

server:
  port: 81
---
spring:
  profiles: test

server:
  port: 82
```

启动指定环境

设置生产环境

生产环境具体参数设定

设置开发环境

开发环境具体参数设定

设置测试环境

测试环境具体参数设定

properties文件多环境启动

- 主启动配置文件application.properties

```properties
spring.profiles.active=pro
```

- 环境分类配置文件application-pro.properties

```properties
server.port=80
```

- 环境分类配置文件application-dev.properties

```properties
server.port=81
```

- 环境分类配置文件application-test.properties

```properties
server.port=82
```

参数加载优先顺序

发现出来的值和自己预想的值不一样，就可以想是不是加载的顺序不一样

命令行启动时可以临时设置一些参数

Maven与SpringBoot多环境关联

maven为主，boot为辅

- 解决思路：对于源码中非java类的操作要求加载Maven对应的属性，解析${}占位符

④：对资源文件开启对默认占位符的解析

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-resources-plugin</artifactId>
            <configuration>
                <encoding>utf-8</encoding>
                <useDefaultDelimiters>true</useDefaultDelimiters>
            </configuration>
        </plugin>
    </plugins>
</build>
```

配置文件分类

- SpringBoot中4级配置文件

  1级： file ： config/application.yml 【最高】

  2级： file ： application.yml

  3级：classpath：config/application.yml

  4级：classpath：application.yml 【最低】

- 作用：
  - 1级与2级留做系统打包后设置通用属性
  - 3级与4级用于系统开发阶段设置通用属性

整合第三方

整合JUnit

自动生成的，其实什么也不用管

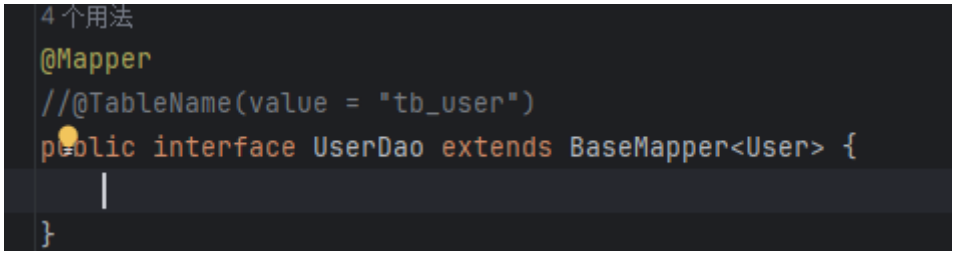
主启动类的级别要最高才可以

Mybatis-plus

导入

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.1</version>
</dependency>
```

继承



MyBatisPlus特性

- 无侵入：只做增强不做改变，不会对现有工程产生影响
- 强大的 CRUD 操作：内置通用 Mapper，少量配置即可实现单表CRUD 操作
- 支持 Lambda：编写查询条件无需担心字段写错
- 支持主键自动生成
- 内置分页插件
- ……

标准数据层CRUD功能

功能	自定义接口	MP接口
新增	`boolean save(T t)`	`int insert(T t)`
删除	`boolean delete(int id)`	`int deleteById(Serializable id)`
修改	`boolean update(T t)`	`int updateById(T t)`
根据id查询	`T getById(int id)`	`T selectById(Serializable id)`
查询全部	`List<T> getAll()`	`List<T> selectList()`
分页查询	`PageInfo<T> getAll(int page, int size)`	`IPage<T> selectPage(IPage<T> page)`
按条件查询	`List<T> getAll(Condition condition)`	`IPage<T> selectPage(Wrapper<T> queryWrapper)`

update的好处

可以自动动态修改，有改动的数据才会修改，没有的话是不会修改为null的

Lombok

jar包，快速开发实体类

@Data不包含构造方法的两个注解，但是在我自己的程序上面包含了

分页功能

分页拦截器，需要配置

```
@Configuration
public class MpConfig {
    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor(){
        //1.定义Mp拦截器
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        //2.添加具体的拦截器
        interceptor.addInnerInterceptor(new PaginationInnerInterceptor());
        return interceptor;
    }
}
```

开启Mybatis-plus的日志

在yaml、yml文件中输入

```
#开启mp的日志到控制台
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

查询语句

支持链接编程，推荐Lambda方式

```
    public void testSelect(){
//        QueryWrapper wrapper = new QueryWrapper();
//        wrapper.lt("age",1000);
//        List<User> list = userDao.selectList(wrapper);
//        System.out.println(list);

        //Lambda方式
        LambdaQueryWrapper<User> lambdaQueryWrapper  = new LambdaQueryWrapper<>
();
//        lambdaQueryWrapper.gt(User::getAge, 1000).lt(User::getAge,10000);//并且
关系
        lambdaQueryWrapper.lt(User::getAge, 1000).or().gt(User::getAge,10000);//
或者关系
        List<User> users = userDao.selectList(lambdaQueryWrapper);
        System.out.println(users);
    }
```

条件查询的null值判定

要创建一个多UserQuery，

```
lambdaQueryWrapper.eq(User::getAge, 999);//相当于where age == 999
```

范围查询

lt、gt不带等号，le，ge带等号，eq是等于

模糊查询，like

字段映射与表名映射

- 名称：@TableField
- 类型：**属性注解**
- 位置：模型类属性定义上方
- 作用：设置当前属性对应的数据库表中的字段关系
- 范例：

```
public class User {
    @TableField(value="pwd")
    private String password;
}
```

- 相关属性
  - value（默认）：设置数据库表字段名称

如果编码中添加了数据库中未定义的属性

要用@TableField（exist = false）

- 范例：

```
public class User {
    @TableField(value="pwd",select = false)
    private String password;
}
```

- 相关属性
  - value：设置数据库表字段名称
  - exist：设置属性在数据库表字段中是否存在，默认为true。此属性无法与value合并使用
  - **select：设置属性是否参与查询，此属性与select()映射配置不冲突**

增删改

id生成策略控制

- 不同的表应用不同的id生成策略
  - 日志：自增（1,2,3,4，……）
  - 购物订单：特殊规则（FQ23948AK3843）
  - 外卖单：关联地区日期等信息（10 04 20200314 34 91）
  - 关系表：可省略id
  - ……
- AUTO(0)：使用数据库id自增策略控制id生成
- NONE(1)：不设置id生成策略
- INPUT(2)：用户手工输入id
- ASSIGN_ID(3)：雪花算法生成id（可兼容数值型与字符串型）
- ASSIGN_UUID(4)：以UUID生成算法作为id生成策略

雪花算法

批量删除

```
userDao.deleteBatchIds(list);
```

好像只能根据集合，不能数组

逻辑删除

- 删除操作业务问题：业务数据从数据库中丢弃
- 逻辑删除：为数据设置是否可用状态字段，删除时设置状态字段为不可用状态，数据保留在数据库中

实体类中

```
//逻辑删除字段
@TableLogic(value = "0" , delval = "1")
private int deleted;
```

也可以直接在配置文件中通用字段逻辑

```
mybatis-plus:
  global-config:
    db-config:
      logic-delete-field: deleted
      logic-not-delete-value: 0
      logic-delete-value: 1
```

推荐通用的

乐观锁

业务并发现象带来的问题：秒杀

关键：使用拦截器

实体类中

```
//用于乐观锁
@Version
private int version;
```

拦截器

```
//乐观锁
interceptor.addInnerInterceptor(new OptimisticLockerInnerInterceptor());
```