# Java高级工程师

项目实践第二讲

架构升级

# 下面开始代码实现

先从Model即我们先前说的领域对象开始

# 实现用户注册的前端请求领域对象

```java
package com.jzsf.tuitor.rpcDomain.reg;

import java.io.Serializable;

/**
 * @author by
 * @since 2020/04/12
 * 携带注册信息请求的entity
 */
public class RegisterReq implements Serializable {

    private static final long serialVersionUID = -690477244058984705L;
    /**
     * 用户名
     */
    private String username;
    /**
     * 密码
     */
    private String password;
    /**
     * 邮箱
     */
    private String email;
    /**
     * 验证码
     */
    private String captcha;
    /**
     * Constructor.
```

# 添加cotorller对象承载前端领域对象

针对用户的操作,都收口到UserController里

```java
package com.jzsf.tuitor.control;

import ...

/**
 * @author by
 * @since 2020/04/12
 * 用户登录操作核心控制器
 */
@Controller
@RequestMapping(value = "/user")
public class UserController {
```

# 添加cotorller对象承载前端领域对象

```java
package com.jzsf.tuitor.control;

import ...

/**
 * @author by
 * @since 2020/04/12
 * 用户登录操作核心控制器
 */
@Controller
@RequestMapping(value = "/user")
public class UserController {
```

@Controller 注解该类在spring boot上下文中被识别为contorller对象

# 添加cotorller对象承载前端领域对象

```java
package com.jzsf.tuitor.control;

import ...

/**
 * @author by
 * @since 2020/04/12
 * 用户登录操作核心控制器
 */
@Controller
@RequestMapping(value = "/user")
public class UserController {
```

@Controller 注解该类在spring boot上下文中被识别为contorller对象

# @RequestMapping的用法

将 HTTP 请求映射到 MVC 和 REST 控制器的处理方法上。

# @RequestMapping的用法

在控制器类的级别和/或其中的方法的级别上使用。
在类的级别上的注解会将一个特定请求或者请求模式映射到一个控制器之上。之后你还可以另外添加方法级别的注解来进一步指定到处理方法的映射关系。

# @RequestMapping的用法

简单的来说,刚刚这个注解说明,前端发来的后缀为/user的请求,都会被定位到这个controller中处理

# 实现register注册方法

```java
@PostMapping(value = "/register", produces = {MediaType.APPLICATION_JSON_VALUE})
@ResponseBody
@JwtIgnore
public RespResult register(@RequestBody RegisterReq reqInfo) {
    // 先校验验证码
    if (!userService.checkCaptcha(reqInfo)) {
        return new RespResult(ResultCode.WRONG_CAPTCHA);
    }
    // 执行注册
    return userService.registerUser(reqInfo);
}
```

# @PostMapping

将请求后缀为/user/register 的post请求映射到
public RespResult register(@RequestBody RegisterReq reqInfo)
来进行处理

# MediaType

表示该方法可以接受的参数类型

# 直接看源码,选择自己需要的参数类型

```
package org.springframework.http;

import ...

public class MediaType extends MimeType implements Serializable {
    private static final long serialVersionUID = 2069937152339670231L;
    public static final MediaType ALL = valueOf("*/*");
    public static final String ALL_VALUE = "*/*";
    public static final MediaType APPLICATION_ATOM_XML = valueOf("application/atom+xml");
    public static final String APPLICATION_ATOM_XML_VALUE = "application/atom+xml";
    public static final MediaType APPLICATION_FORM_URLENCODED = valueOf("application/x-www-form-urlencoded");
    public static final String APPLICATION_FORM_URLENCODED_VALUE = "application/x-www-form-urlencoded";
    public static final MediaType APPLICATION_JSON = valueOf("application/json");
    public static final String APPLICATION_JSON_VALUE = "application/json";
    public static final MediaType APPLICATION_JSON_UTF8 = valueOf("application/json;charset=UTF-8");
    public static final String APPLICATION_JSON_UTF8_VALUE = "application/json;charset=UTF-8";
    public static final MediaType APPLICATION_OCTET_STREAM = valueOf("application/octet-stream");
    public static final String APPLICATION_OCTET_STREAM_VALUE = "application/octet-stream";
    public static final MediaType APPLICATION_PDF = valueOf("application/pdf");
    public static final String APPLICATION_PDF_VALUE = "application/pdf";
    public static final MediaType APPLICATION_PROBLEM_JSON = valueOf("application/problem+json");
    public static final String APPLICATION_PROBLEM_JSON_VALUE = "application/problem+json";
    public static final MediaType APPLICATION_PROBLEM_JSON_UTF8 = valueOf("application/problem+json;charset=UTF-8");
    public static final String APPLICATION_PROBLEM_JSON_UTF8_VALUE = "application/problem+json;charset=UTF-8";
    public static final MediaType APPLICATION_PROBLEM_XML = valueOf("application/problem+xml");
    public static final String APPLICATION_PROBLEM_XML_VALUE = "application/problem+xml";
    public static final MediaType APPLICATION_RSS_XML = valueOf("application/rss+xml");
    public static final String APPLICATION_RSS_XML_VALUE = "application/rss+xml";
    public static final MediaType APPLICATION_STREAM_JSON = valueOf("application/stream+json");
    public static final String APPLICATION_STREAM_JSON_VALUE = "application/stream+json";
    public static final MediaType APPLICATION_XHTML_XML = valueOf("application/xhtml+xml");
    public static final String APPLICATION_XHTML_XML_VALUE = "application/xhtml+xml";
    public static final MediaType APPLICATION_XML = valueOf("application/xml");
    public static final String APPLICATION_XML_VALUE = "application/xml";
    public static final MediaType IMAGE_GIF = valueOf("image/gif");
```

# @RequestBody

用来接收前端请求体中的数据并自动映射为前端请求的领域模型

# @RequestBody的使用规范

GET方式无请求体，所以使用@RequestBody接收数据时，前端不能使用GET方式提交数据，而是用POST方式进行提交。

# @RequestBody的使用规范

根据json字符串中的key来匹配对应实体类的属性，如果匹配一致且json中的该key对应的值符合

# @RequestBody的使用规范

实体类的对应属性的类型要求时,会调用实体类的setter方法将值赋给该属性。

换言之,你的领域实体bean不能正确注入时,注意检查是否有完整的setter,getter方法

# 继续看controller的实现

```java
@PostMapping(value = "/register", produces = {MediaType.APPLICATION_JSON_VALUE})
@ResponseBody
@JwtIgnore
public RespResult register(@RequestBody RegisterReq reqInfo) {
    // 先校验验证码
    if (!userService.checkCaptcha(reqInfo)) {
        return new RespResult(ResultCode.WRONG_CAPTCHA);
    }
    // 执行注册
    return userService.registerUser(reqInfo);
}
```

# 可以看出controller直接使用service的能力来实现功能

# 关注userService

```
@Autowired
private UserService userService;
```

@Autowired 注释，它可以对类成员变量、方法及构造函数进行标注，完成自动装配的工作

# @Autowired

对类成员变量、方法及构造函数进行标注，完成自动装配的工作

# @Autowired

该注解可以消除 set， get方法,降低代码冗余度.

# @Autowired

对比不用注解的场景:

<property name="属性名" value=" 属性值"/>

可以节省大量的配置工作量

# 声明一个接口,规约行为

虽然spring boot可以直接按component的方式注入一个实例,但是使用接口进行服务规约是面向服务架构甚至后续提升到微服务架构时强依赖的技术能力,因此,使用接口规约来实现系统架构的集成是工业级软件开发的基本原则.

# UserService的实现

```java
@Service
public interface UserService extends BaseService<User, String> {
```

在用户注册场景中使用到的接口方法:

```java
    /**
     * 校验验证码
     *
     * @param reqInfo
     * @return
     */
    boolean checkCaptcha(RegisterReq reqInfo);

    /**
     * 注册新用户
     *
     * @param reqInfo
     * @return 注册动作信息
     */
    RespResult registerUser(RegisterReq reqInfo);
```

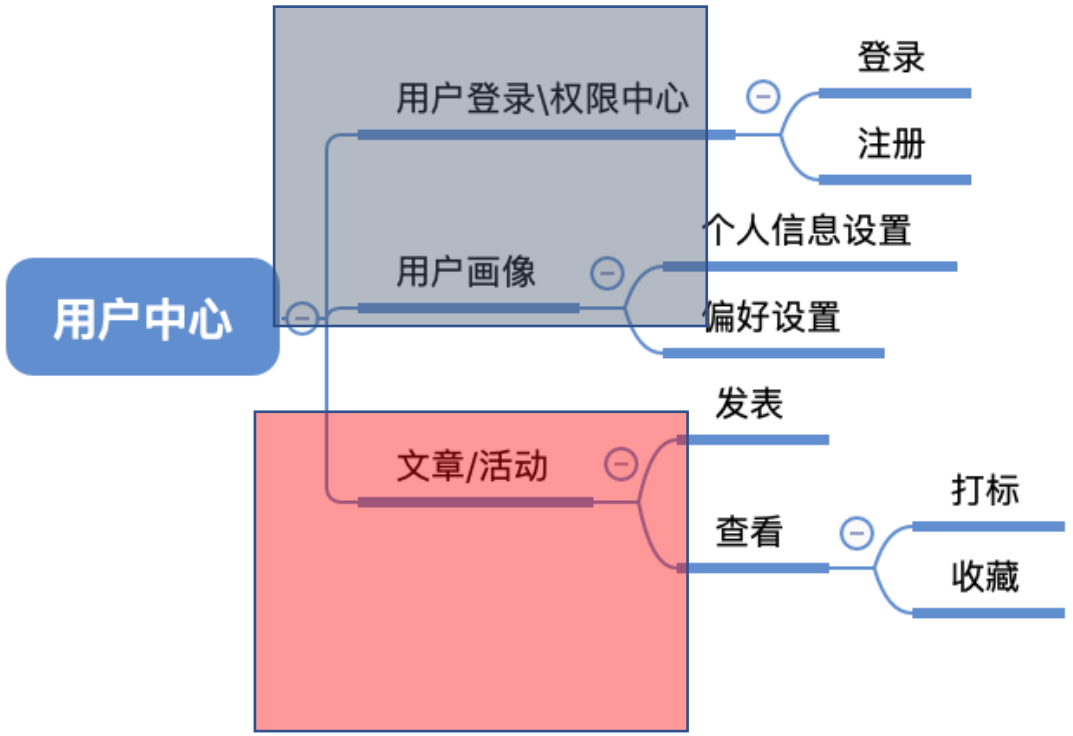
# 问题:

为什么在项目中Service类需要类似BaseService这样基础接口进行抽象?

# BaseService 基础能力收口

```java
@Service
public interface BaseService<T, ID> {

    /**
     * 保存单个实体
     *
     * @param entity
     * @param <S>
     * @return
     */
    <S extends T> S save(S entity);

    /**
     * 保存所有实体
     *
     * @param entitys
     * @param <S>
     * @return
     */
    <S extends T> Iterable<S> saveAll(Iterable<S> entitys);

    /**
     * 根据id获取对象
     *
     * @param id
     * @return
     */
    Optional<T> findById(ID id);

    /**
```

# 服务也是可以进行类型划分和聚合的



# 既然能分类,就一定有共性可以沉淀

所谓的架构师,就是能快速的识别这样共性并能优雅的沉淀到底层.

# 底层怎么定义？

架构上的底层
沉淀为底层组件
沉淀为基类、上层接口

# 编写注册用户的逻辑

```
@PostMapping(value = "/getCaptcha", produces = {MediaType.APPLICATION_JSON_VALUE})
@ResponseBody
@JwtIgnore
public RespResult getCaptcha(@RequestBody RegisterReq reqInfo) { return userService.beforeRegister(reqInfo); }
```

```
@Override
public RespResult registerUser(RegisterReq reqInfo) {
    User user = userDao.getByUsername(reqInfo.getUsername());
    user.setIsVerified("1");
    userDao.save(user);
    initUserInfo(user);
    return new RespResult(ResultCode.REGISTERED_SUCCESS);
}
```

```java
@Override
public RespResult beforeRegister(RegisterReq reqInfo) {

    // 无效的注册
    if (StringUtils.isNotBlank(reqInfo.getCaptcha())
            || StringUtils.isBlank(reqInfo.getEmail())
            || StringUtils.isBlank(reqInfo.getPassword())
            || StringUtils.isBlank(reqInfo.getUsername())) {
        return new RespResult(ResultCode.REG_DATA_IS_WRONG);
    }

    // 检查是否注册
    int registerCode = registerRecordService.checkRegister(reqInfo.getEmail(), reqInfo.getUsername());
    if (registerCode == 1) {
        return new RespResult(ResultCode.USERNAME_HAS_USED);
    } else if (registerCode == 2) {
        return new RespResult(ResultCode.MAIL_HAS_USED);
    }

    // 获取验证码
    String captcha = RandomCaptcha.get();

    // 发送邮件
    StringBuilder contentBuilder = new StringBuilder();
    contentBuilder.append("您好, ").append(reqInfo.getUsername()).append(", 您的验证码是 : ").append(captcha);
    String subject = "新用户注册";
    boolean isSend = mailService.sendSimpleMail(reqInfo.getEmail(), subject, contentBuilder.toString());
    if (isSend) {
        // 保存注册记录
        RegisterRecord registerRecord = new RegisterRecord();
        registerRecord.setId(UUIDUtil.getUUID());
        registerRecord.setUsername(reqInfo.getUsername());
        registerRecord.setEmail(reqInfo.getEmail());
        registerRecord.setCaptcha(captcha);
        registerRecord.setSendTime(new Date());
        registerRecordService.save(registerRecord);

        // 保存用户
        User user = new User();
        user.setUsername(reqInfo.getUsername());
        user.setPassword(MD5Utils.getMD5(reqInfo.getPassword()));
        user.setId(UUIDUtil.getUUID());
        userDao.save(user);
        return new RespResult(ResultCode.REGISTER_CAPTCHA_SEND);
    } else {
        return new RespResult(ResultCode.MAIL_SEND_FAIL);
    }
}
```

# 编写注册用户的逻辑

```java
@PostMapping(value = "/register", produces = {MediaType.APPLICATION_JSON_VALUE})
@ResponseBody
@JwtIgnore
public RespResult register(@RequestBody RegisterReq reqInfo) {
    // 先校验验证码
    if (!userService.checkCaptcha(reqInfo)) {
        return new RespResult(ResultCode.WRONG_CAPTCHA);
    }
    // 执行注册
    return userService.registerUser(reqInfo);
```

```java
@Override
public RespResult registerUser(RegisterReq reqInfo) {
    User user = userDao.getByUsername(reqInfo.getUsername());
    user.setIsVerified("1");
    userDao.save(user);
    initUserInfo(user);
    return new RespResult(ResultCode.REGISTERED_SUCCESS);
}
```

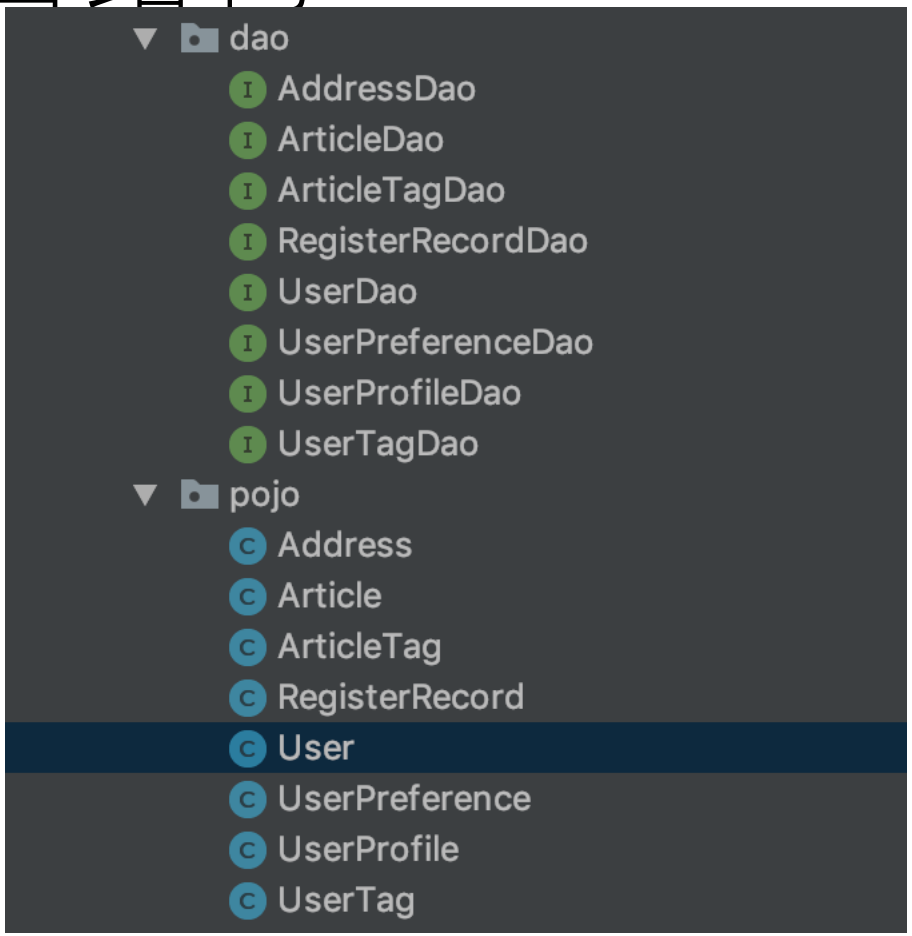
# Jpa数据库接口

```java
@Repository
public interface UserDao extends JpaRepository<User, String> {

    /**
     * 根据用户名，获取entity
     *
     * @param username 用户名
     * @return 当前User
     */
    User getByUsername(String username);
}
```

# DAO层编写

DAO层的实现有两个方面:
Dao接口
POJO持久化bean

# DAO层的项目结构


dao
AddressDao
ArticleDao
ArticleTagDao
RegisterRecordDao
UserDao
UserPreferenceDao
UserProfileDao
UserTagDao
pojo
Address
Article
ArticleTag
RegisterRecord
User
UserPreference
UserProfile
UserTag

# DAO接口

Dao接口就是所谓的Mapper接口

# DAO接口

将对实体POJO的操作映射为实际上的数据库操作

# DAO接口

```java
@Repository
public interface UserDao extends JpaRepository<User, String> {

    /**
     * 根据用户名，获取entity
     *
     * @param username 用户名
     * @return 当前User
     */
    User getByUsername(String username);
}
```

通过继承JpaRepository将底层的数据库访问能力开放到业务层供访问

# 面试问题

你使用spring boot来开发项目,是看中了它的什么能力,又是怎样使用它的呢?

频度:高

难度:中

通过率:中

# 答案

要点1:整体上,用spring mvc的能力做了控制器-view和后端service的解耦

# 答案

要点2:用spring的bean管理能力轻松实现架构扩展

# key

项目实战讨论其实就是对候选人架构能力和架构思想的考核

# key

架构能力两个关键点:

解耦

沉淀

# 问题:骨架已经出来了,是否业务代码写完就完事了呢?

Spring boot的结构是基于框架能力的,真正面向实际场景,我们还要进行下一层面的设计:

# 不适感是架构升级和技术升级的推动力

现在看一看,有哪些代码现在或者是将来会带给我们不适

# 代码走读

用架构师的眼光走读源码

# 大家来找茬



```java
@Override
public RespResult beforeRegister(RegisterReq reqInfo) {

    // 无效的注册
    if (StringUtils.isNotBlank(reqInfo.getCaptcha())
            || StringUtils.isBlank(reqInfo.getEmail())
            || StringUtils.isBlank(reqInfo.getPassword())
            || StringUtils.isBlank(reqInfo.getUsername())) {
        return new RespResult(ResultCode.REG_DATA_IS_WRONG);
    }

    // 检查是否注册
    int registerCode = registerRecordService.checkRegister(reqInfo.getEmail(), reqInfo.getUsername());
    if (registerCode == 1) {
        return new RespResult(ResultCode.USERNAME_HAS_USED);
    } else if (registerCode == 2) {
        return new RespResult(ResultCode.MAIL_HAS_USED);
    }

    // 获取验证码
    String captcha = RandomCaptcha.get();

    // 发送邮件
    StringBuilder contentBuilder = new StringBuilder();
    contentBuilder.append("您好, ").append(reqInfo.getUsername()).append(", 您的验证码是 : ").append(captcha);
    String subject = "新用户注册";
    boolean isSend = mailService.sendSimpleMail(reqInfo.getEmail(), subject, contentBuilder.toString());
    if (isSend) {
        // 保存注册记录
        RegisterRecord registerRecord = new RegisterRecord();
        registerRecord.setId(UUIDUtil.getUUID());
```

一个方法只做一件事,这里一个方法做了多少件事呢?

从userService 接口方法定义来看,beforeRegister应该是实现注册前包围的一种设计,取验证码时候调用前包围function就很奇怪了

```
@PostMapping(value = "/getCaptcha", produces = {MediaType.APPLICATION_JSON_VALUE})
@ResponseBody
@JwtIgnore
public RespResult getCaptcha(@RequestBody RegisterReq reqInfo) { return userService.beforeRegister(reqInfo); }
```

# 这里是表单校验的过程,我们现在给架构赋能实现



```
// 无效的注册
if (StringUtils.isNotBlank(reqInfo.getCaptcha())
        || StringUtils.isBlank(reqInfo.getEmail())
        || StringUtils.isBlank(reqInfo.getPassword())
        || StringUtils.isBlank(reqInfo.getUsername())) {
    return new RespResult(ResultCode.REG_DATA_IS_WRONG);
}
```

# 新建一个表单校验能力模板接口



```java
public interface FromValitor<T,E> {
    /**
     *
     * @param inPkg
     * @return
     */
    boolean canAccept(T inPkg);

    /**
     *
     * @param aInPkg
     * @param aOutPkg
     */
    void validate(T aInPkg,E aOutPkg);

}
```

```java
@Service
public class RegisterFormValidator implements FormValidator {

    /**
     * @param inPkg
     * @return
     */
    @Override
    public boolean canAccept(Object inPkg) {
        if (inPkg instanceof RegisterReq) {
            return true;
        }
        return false;
    }

    /**
     * @param aInPkg
     */
    @Override
    public RespResult validate(Object aInPkg) {
        RegisterReq reqInfo = (RegisterReq) aInPkg;
        RespResult rspInfo = null;
        // 无效的注册
        if (StringUtils.isNotBlank(reqInfo.getCaptcha())
                || StringUtils.isBlank(reqInfo.getEmail())
                || StringUtils.isBlank(reqInfo.getPassword())
                || StringUtils.isBlank(reqInfo.getUsername())) {
            rspInfo = new RespResult(ResultCode.REG_DATA_IS_WRONG);
        }
        rspInfo = new new RespResult(ResultCode.SUCCESS);
        return rspInfo;
    }
}
```

beforeRegiter是controller层的方法,不应该穿透到common层调用

```
// 检查是否注册
int registerCode = registerRecordService.checkRegister(reqInfo.getEmail(), reqInfo.getUsername());
if (registerCode == 1) {
    return new RespResult(ResultCode.USERNAME_HAS_USED);
} else if (registerCode == 2) {
    return new RespResult(ResultCode.MAIL_HAS_USED);
}
```

# 改进方法

验证码发送是公共能力,应该作为独立服务向前端开放

Service层新增一个接口，

```
/**...*/
package com.jzsf.tuitor.service;

/**
 *
 * @author senyang
 * @version $Id: ToolService.java, v 0.1 2020年
 */
public interface ToolService {
    String getCaptcha();
}
```

# 实现这个接口,将底层工具能力引到service层来

```
@Service
public class ToolServiceImpl implements ToolService {
    @Override
    public String getCaptcha() {
        return RandomCaptcha.get();
    }
}
```

```
// 获取验证码
String captcha = toolService.getCaptcha();
```

# 注意

这里用了个注解 @Service,引出一个知识点

# 面试题

Spring boot的bean注解有哪些?它们有什么区别?

难度:低

频度:高

通过率:中

| 注解 | 含义 |
| --- | --- |
| @Component | 最普通的组件，可以被注入到spring容器进行管理 |
| @Repository | 作用于持久层 |
| @Service | 作用于业务逻辑层 |
| @Controller | 作用于表现层（spring-mvc的注解） |

# 扩展

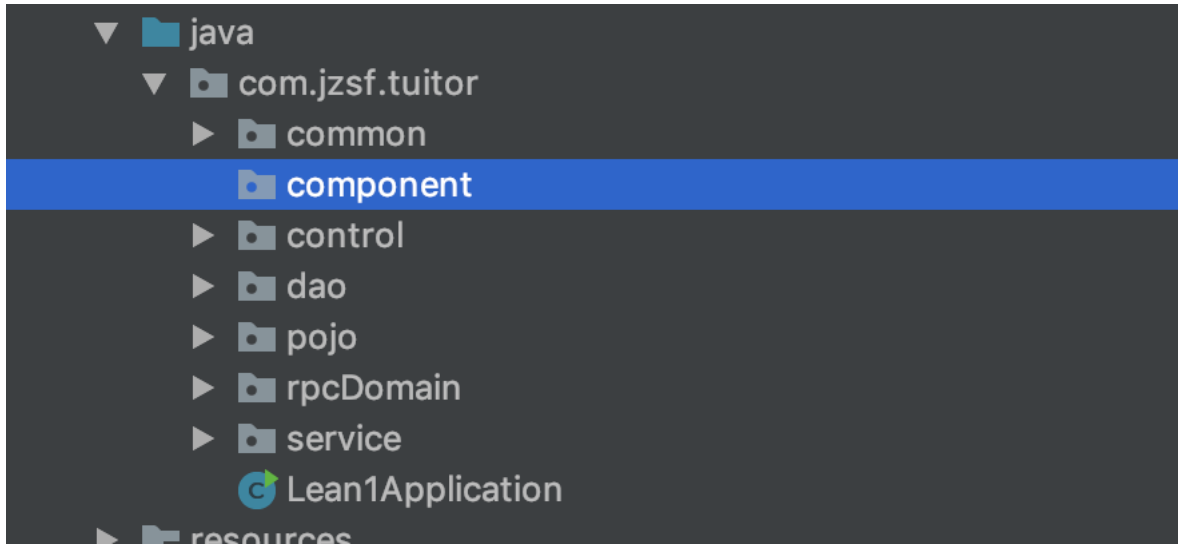
用于微服务架构时,@Service注解意味着能够支持微服务模块间的调用

# 改进方法

如果beforeRegister既然作为前包围方法,必然涉及到多种前置操作
这些操作细节必然需要通过架构优化的方式进行收口

# 增加一个component层,把之前复杂的逻辑封装为独立的组件能力





# 如:表单校验,就应该这样进行抽象处理



```java
public interface FormValidator<T> {
    /**
     *
     * @param inPkg
     * @return
     */
    boolean canAccept(T inPkg);

    /**
     *
     * @param aInPkg
     */
    RespResult validate(T aInPkg );


}
```

# 思考,从源码上看,我们的表单校验是分主题的



- 非空校验
- 注册状态校验
- 验证码校验

# 抽象一个通用表单校验接口



```java
public interface FormValidator<T> {
    /**
     *
     * @param inPkg
     * @return
     */
    boolean canAccept(T inPkg);

    /**
     *
     * @param aInPkg
     */
    RespResult validate(T aInPkg );

}
```

# 以非空校验为例



```java
@Service
public class RegisterFormValidator implements FormValidator {

    /**
     * @param inPkg
     * @return
     */
    @Override
    public boolean canAccept(Object inPkg) {
        if (inPkg instanceof RegisterReq) {
            return true;
        }
        return false;
    }

    /**
     * @param aInPkg
     */
    @Override
    public RespResult validate(Object aInPkg) {
        RegisterReq reqInfo = (RegisterReq) aInPkg;
        RespResult rspInfo = null;
        // 无效的注册
        if (StringUtils.isNotBlank(reqInfo.getCaptcha())
                || StringUtils.isBlank(reqInfo.getEmail())
                || StringUtils.isBlank(reqInfo.getPassword())
                || StringUtils.isBlank(reqInfo.getUsername())) {
            rspInfo = new RespResult(ResultCode.REG_DATA_IS_WRONG);
        }
        rspInfo =  new RespResult(ResultCode.SUCCESS);
        return rspInfo;

    }

}
```

# 思考



- Validate 可以理解

- 为何要有canAccept?

# 观察我们的请求领域模型对象



# Key

每一次在架构上进行突破都要考虑架构能力是否可以做到全覆盖了

# canAccept

看到这个方法,一般可以判定我们要选用责任链模式的能力了

# 回顾一下责任链模式

# 要点

一个接口
一个迭代
一组实现
一个准入

# 特点

整组的实现平等执行

# 实现一个Manager对象统一对这些校验器进行管理



```
 */
public class ReqValiadateManagerImpl implements ReqValiadateManager {

    @Autowired
    private List<FormValidator> validators;

    @Override
    public void doExeute(Object aInPkg) {

        for(FormValidator validator : validators){
            if(validator.canAccept(aInPkg)){
                validator.validate(aInPkg);
            }
        }

    }
}
```

# 这是一个大体的实现

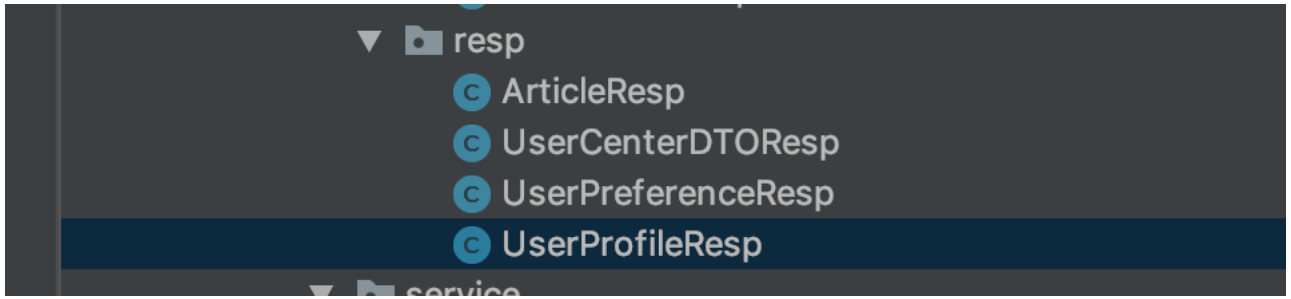
但遇上了一个很不舒适的block

# 这里**校验完,结果回不去了**

```
 */
public class ReqValiadateManagerImpl implements ReqValiadateManager {

    @Autowired
    private List<FormValidator> validators;

    @Override
    public void doExeute(Object aInPkg) {

        for(FormValidator validator : validators){
            if(validator.canAccept(aInPkg)){
                validator.validate(aInPkg);
            }
        }

    }
}
```

# Why?

回应领域对象没有做架构收口




用我们之前的知识,简单的做法就是放一个基类来进行架构收口

```
 */
public class ReqValiadateManagerImpl implements ReqValiadateManager {

    @Autowired
    private List<FormValidator> validators;

    @Override
    public void doExeute(Object aInPkg) {

        for(FormValidator validator : validators){
            if(validator.canAccept(aInPkg)){
                validator.validate(aInPkg);
            }
        }

    }
}
```

用基类接住

然后再去强制转换为场景需要的子对象类型

然而,臃肿且不优雅

# 架构困境

不断走向统一的架构和复杂多样的前端感知层之间的矛盾

# 架构困境

现实就是前端需要针对不同的请求场景订制不同的返回包.

# 解法一

继承收口法

# 解法二

针对这样强感知成功与否这样的逻辑链路,用异常处理进行架构收口是很有效的

# 为表单验证场景订制专用的异常

```
public class ValidateException extends RuntimeException {

    public ValidateException(ResultCode code, String msg){
        resultCode = code;
        errMsg = msg;
    }
    public String getErrLogStr() { return "{"+resultCode+"}:" + errMsg; }
```

# 一个技巧

继承于RuntimeException

1、把对业务代码的侵入降到最小

2、避免对不感知该异常的逻辑流造成阻断

# 现在校验器的代码就很轻量了

```
@Override
public void validate(Object aInPkg) {
    // 检查是否注册
    RegisterReq reqInfo = (RegisterReq) aInPkg;
    int registerCode = registerRecordService.checkRegister(reqInfo.getEmail(), reqInfo.getUsername());
    if (registerCode == 1) {
        throw new ValidateException(ResultCode.USERNAME_HAS_USED,"用户名已被占用");
    } else if (registerCode == 2) {
        throw new ValidateException(ResultCode.MAIL_HAS_USED,"邮箱已被占用");

    }
```

# 优势

服务层不再需要感知Controller层的东西

# 优势

只要定制异常的捕获机制,就可以轻松的让其上穿到我们需要的业务层进行处理

# 对manager主控没有影响,直接穿到上一层

```
public class RegValiadateManagerImpl implements ReqValiadateManager {

    @Autowired
    private List<FormValidator> validators;

    @Override
    public void doExeute(Object aInPkg) {

        for(FormValidator validator : validators){
            if(validator.canAccept(aInPkg)) {
                validator.validate(aInPkg);
            }
        }
    }
}
```

# 非常干净的实现

```
/**
 * 调用valiadate manager
 */
try{
    reqValiadateManager.doExeute(reqInfo);
}catch (ValidateException e){
    return new RespResult(e.getResultCode());
}
```

# 继续往下看代码

```
// 获取验证码
String captcha = toolService.getCaptcha();

// 发送邮件
StringBuilder contentBuilder = new StringBuilder();
contentBuilder.append("您好, ").append(reqInfo.getUsername()).append(", 您的验证码是 : ").append(captcha);
String subject = "新用户注册";
boolean isSend = mailService.sendSimpleMail(reqInfo.getEmail(), subject, contentBuilder.toString());
if (isSend) {
    // 保存注册记录
    RegisterRecord registerRecord = new RegisterRecord();
```

# 通用能力的沉淀

验证码和邮件发送都不仅仅是在这个场景里面有用,可以预见是整个系统的底盘元件

# 沉淀的难度在于敏感度

架构师的经济价值在于沉淀共性,不重复造轮子

# ToolService中沉淀底层能力

```
@Service
public class ToolServiceImpl implements ToolService {
    @Autowired
    private MailService mailService;

    @Override
    public String getCaptcha() {
        return RandomCaptcha.get();
    }

    @Override
    public boolean sendRegisterMail(RegisterReq req {
        String captcha = this.getCaptcha();
        StringBuilder contentBuilder = new StringBuilder();
        contentBuilder.append("您好, ").append(req.getUsername()).append(", 您的验证码是 : ").append(captcha);
        String subject = "新用户注册";
        return this.mailService.sendSimpleMail(req.getEmail(), subject, contentBuilder.toString());
    }

}
```

# 现在看看调用层的代码

```
/**
 * 调用valiadate manager
 */
try{
    reqValiadateManager.doExeute(reqInfo);
}catch (ValidateException e){
    return new RespResult(e.getResultCode());
}

boolean isSend = toolService.sendRegisterMail(reqInfo);
```

# 对比之前

```java
@Override
public RespResult beforeRegister(RegisterReq reqInfo) {

    // 无效的注册
    if (StringUtils.isNotBlank(reqInfo.getCaptcha())
            || StringUtils.isBlank(reqInfo.getEmail())
            || StringUtils.isBlank(reqInfo.getPassword())
            || StringUtils.isBlank(reqInfo.getUsername())) {
        return new RespResult(ResultCode.REG_DATA_IS_WRONG);
    }

    // 检查是否注册
    int registerCode = registerRecordService.checkRegister(reqInfo.getEmail(), reqInfo.getUsername());
    if (registerCode == 1) {
        return new RespResult(ResultCode.USERNAME_HAS_USED);
    } else if (registerCode == 2) {
        return new RespResult(ResultCode.MAIL_HAS_USED);
    }

    // 获取验证码
    String captcha = RandomCaptcha.get();

    // 发送邮件
    StringBuilder contentBuilder = new StringBuilder();
    contentBuilder.append("您好, ").append(reqInfo.getUsername()).append(", 您的验证码是 : ").append(captcha);
    String subject = "新用户注册";
    boolean isSend = mailService.sendSimpleMail(reqInfo.getEmail(), subject, contentBuilder.toString());
    if (isSend) {
        // 保存注册记录
        RegisterRecord registerRecord = new RegisterRecord();
        registerRecord.setId(UUIDUtil.getUUID());
```

# 下面对这一段逻辑下手

```
if (isSend) {
    // 保存注册记录
    RegisterRecord registerRecord = new RegisterRecord();
    registerRecord.setId(UUIDUtil.getUUID());
    registerRecord.setUsername(reqInfo.getUsername());
    registerRecord.setEmail(reqInfo.getEmail());
    registerRecord.setCaptcha(toolService.getCaptcha());
    registerRecord.setSendTime(new Date());
    registerRecordService.save(registerRecord);

    // 保存用户
    User user = new User();
    user.setUsername(reqInfo.getUsername());
    user.setPassword(MD5Utils.getMD5(reqInfo.getPassword()));
    user.setId(UUIDUtil.getUUID());
    userDao.save(user);
    return new RespResult(ResultCode.REGISTER_CAPTCHA_SEND);
} else {
    return new RespResult(ResultCode.MAIL_SEND_FAIL);
}
```

# 思考

这一段代码如何把它“清理掉”

# 高级程序员的思维

降为底层方法供上层调用

# 架构师思维

看本质

# 发送成功与否是一个策略

```
if (isSend) {
    // 保存注册记录
    RegisterRecord registerRecord = new RegisterRecord();
    registerRecord.setId(UUIDUtil.getUUID());
    registerRecord.setUsername(reqInfo.getUsername());
    registerRecord.setEmail(reqInfo.getEmail());
    registerRecord.setCaptcha(toolService.getCaptcha());
    registerRecord.setSendTime(new Date());
    registerRecordService.save(registerRecord);

    // 保存用户
    User user = new User();
    user.setUsername(reqInfo.getUsername());
    user.setPassword(MD5Utils.getMD5(reqInfo.getPassword()));
    user.setId(UUIDUtil.getUUID());
    userDao.save(user);
    return new RespResult(ResultCode.REGISTER_CAPTCHA_SEND);
} else {
    return new RespResult(ResultCode.MAIL_SEND_FAIL);
}
```

# 策略模式实现架构升级

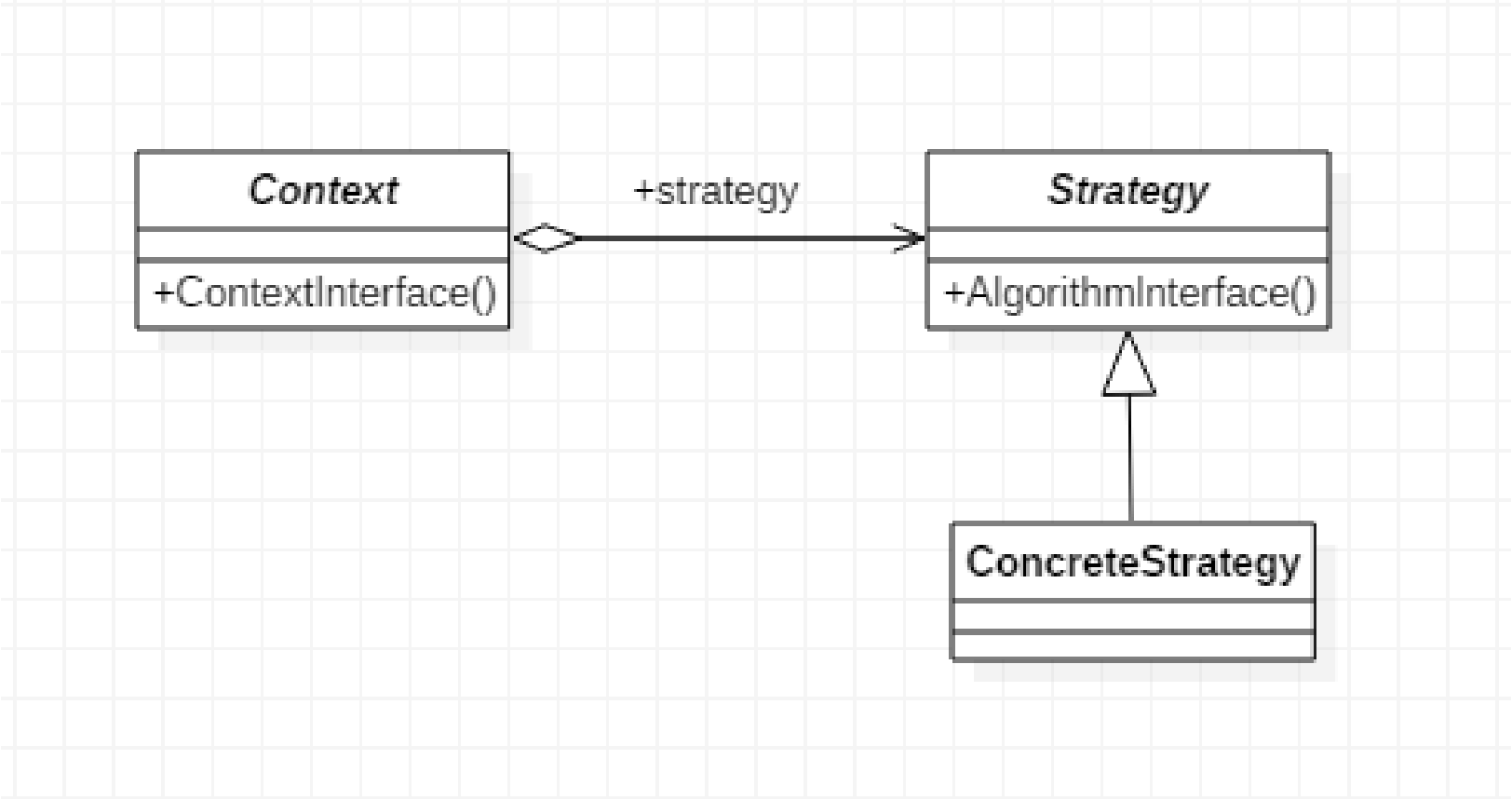
## 什么是策略模式

# 策略模式

一个策略接口

一个上下文枚举

一组策略实现


Context
+ContextInterface()
+strategy
Strategy
+AlgorithmInterface()
ConcreteStrategy

# 活学活用

这里是根据前面的运行结果执行一种策略
运行的结果正好和枚举完美匹配

# 构建Context

```
 */
public enum OperatorStrategeEnum {
    /** 成功 */
    SUCCESS("SUCCESS", "成功"),

    /** 失败 */
    FAIL("FAIL", "失败"),

    /** 未知 */
    UNKNOWN("UNKNOWN", "未知"),

    /** 验证码失败 */
    EMAILFAIL("EMAILFAIL", "验证码发送失败");

    /** code */
    private String code;

    /** 描述 */
    private String desc;


    OperatorStrategeEnum(String code, String desc) {
        this.code = code;
        this.desc = desc;
    }
}
```

# 定义策略接口

```
public interface UserStratege {
    RespResult doProcessor(RegisterReq req, OperatorStrategeEnum context);
}
```

# 实现策略接口

```
@Service
public class RegestProcessingStrategeImpl implements UserStratege{

    @Autowired
    private UserDao userDao;

    @Autowired
    private RegisterRecordService registerRecordService;

    @Override
    public RespResult doProcessor(RegisterReq req,  OperatorStrategeEnum context) {
        if (context == OperatorStrategeEnum.SUCCESS) {
            // 保存注册记录
            RegisterRecord registerRecord = new RegisterRecord();
            registerRecord.setId(UUIDUtil.getUUID());
            registerRecord.setUsername(req.getUsername());
            registerRecord.setEmail(req.getEmail());
            registerRecord.setCaptcha(req.getCaptcha());
            registerRecord.setSendTime(new Date());
            registerRecordService.save(registerRecord);

            // 保存用户
            User user = new User();
            user.setUsername(req.getUsername());
            user.setPassword(MD5Utils.getMD5(req.getPassword()));
            user.setId(UUIDUtil.getUUID());
            userDao.save(user);
            return new RespResult(ResultCode.REGISTER_CAPTCHA_SEND);
        } else {
            return new RespResult(ResultCode.MAIL_SEND_FAIL);
        }
    }
}
```

# 策略的收口

原来在前面杂乱的代码收口到策略点里

# 简单定义一个Strategy策略映射接口

```
public interface ContextMapper {
    UserStratege loadProcessor(OperatorStrategeEnum context);
}
```

# 实现这个接口

```
@Service
public class ContextMapperImpl implements ContextMapper {

    @Autowired
    private RegestProcessingStrategeImpl userStratege;

    @Override
    public UserStratege loadProcessor(OperatorStrategeEnum context) {
        if(context == OperatorStrategeEnum.SUCCESS || context == OperatorStrategeEnum.EMAILFAIL){
            return this.userStratege;
        }
        return null;
    }
```

# 看看经过架构升级的成果

```
@Override
public RespResult beforeRegister(RegisterReq reqInfo) {

    /**
     * 调用validate manager
     */
    try {
        reqValidateManager.doExeute(reqInfo);
    } catch (ValidateException e) {
        return new RespResult(e.getResultCode());
    }

    boolean isSend = toolService.sendRegisterMail(reqInfo);
    OperatorStrategeEnum context;
    if(isSend){
        context = OperatorStrategeEnum.SUCCESS;
    }else{
        context = OperatorStrategeEnum.EMAILFAIL;
    }

    return contextMapper.loadProcessor(context).
            doProcessor(reqInfo, context);

}
```

# 问题

做了很多的架构优化,感觉代码变多了,优化的意义在哪里呢?

# 清晰的划分

集中处理好一个专题是容易的

# 便捷的横向扩展

尽可能的覆盖多样化的场景

# 便捷的横向扩展

可配置

统一的组件编写规则

# 总结

本次课程重点:
从场景分析到系统分析
从技术框架选型到简单的实现
最简单的项目,也有最致命的毛病

# 总结

再简单的项目,面试官也能准确的识别一个候选人的能力
细节之间显示一个人的真功夫
架构思维才是一个高级工程师的真正价值.