目錄

程序员你为什么这么累？

背景

大家好，我是晓风轻，《程序员你为什么这么累？》 系列贴是我最早在知乎（链接）上发表的一系列关于编码规范的帖子，帖子里面讲解的怎么样定义接口和代码模板，怎么样把业务代码写简单清晰，得到很多同行的关注和支持，特此整理成文档供大家参考。

作者联系方式

群名称：java技术交流
群　号：607679993

github https://xwjie.github.io/

导读

大家一提到程序员，首先想到的是以下标签：苦逼，加班，熬夜通宵。但是，但凡工作了的同学都知道，其实大部分程序员做的事情都很简单，代码CRUD可以说毫无技术含量，就算什么不懂依葫芦画瓢很多功能也能勉强做出来，做个多线程并发就算高科技了，程序员这行的门槛其实还是比较低的。（这里说的是大部分，有些牛逼的，写算法、jvm等的请自动跳过）

是不是觉得很矛盾，一方面工作不复杂，一方面却累成狗。有没有想过问题出在哪里？有没有想过时间都花在哪里呢？

对于我个人来说，编码还是一个相对轻松的活（我是负责公司it系统的，没有太多技术含量，数据量大，但并发量不大）。从工作到现在，我加班编码的时间还是比较少的，我到现在为止每天还会编码，很少因为编码工作加班。

大家写的东西都是一些crud的业务逻辑代码，为什么大家这么累，加班加点天天都是奋斗者？我从自己带的项目中观察中发现，大部分人的大部分时间都是在 定位问题 + 改代码，真正开发的时间并不多。定位问题包括开发转测试的时候发现问题和上线后发现问题，改代码的包括改bug和因为需求变动修改代码（后面专门开一贴说如何应对需求改动）。

所以说，simple is not easy。很多人就是因为觉得简单，所以功能完成自己测试ok了就算了，没有思考有没有更加好的方式。归根到底是因为编码习惯太糟糕，写的代码太烂，导致无法定位频繁修改频繁出问题。（后面我会详细讲一些我看到的大部分的编码问题。）

其实，对于个人来说，技术很重要，但是对于工作来说，编码的习惯比技术更加主要。工作中你面试的大部分技术都不需要用到的。工作中，因为你的编码习惯不好，写的代码质量差，代码冗余重复多，很多无关的代码和业务代码搅在一起，导致了你疲于奔命应付各种问题。

所以我作为SE，不管接手任何项目组，第一步就是制定代码框架，制定项目组的开发规范，把代码量减下去。事实上证明，这一步之后，大家的代码量能下去最少1/3，后台的问题数下降比较明显，大家的加班会比之前少。

给大家一个直观的例子。下面是controller的一个删除数据的接口，我来之前大家写的这个样子的（其实一开始比这个还差很多），功能很简单，输入一个对象id执行删除返回是否删除成功。大家有没有觉得有什么问题？

```
@PostMapping("/delete")
public Map<String, Object> delete(long id, String lang) {
  Map<String, Object> data = new HashMap<String, Object>();

  boolean result = false;
  try {
    // 语言（中英文提示不同）
    Locale local = "zh".equalsIgnoreCase(lang) ? Locale.CHINESE : Locale.ENGLISH;

    result = configService.delete(id, local);
```

```
    data.put("code", 0);

  } catch (CheckException e) {
    // 参数等校验出错，这类异常属于已知异常，不需要打印堆栈，返回码为-1
    data.put("code", -1);
    data.put("msg", e.getMessage());
  } catch (Exception e) {
    // 其他未知异常，需要打印堆栈分析用，返回码为99
    log.error(e);

    data.put("code", 99);
    data.put("msg", e.toString());
  }

  data.put("result", result);

  return data;
}
```

其实上面的代码也没有大问题。而我接手之后，我会开发自己的代码框架，最后制定代码框架交付的代码如下（这是controller的部分）：

```
@PostMapping("/delete")
public ResultBean<Boolean> delete(long id) {
  return new ResultBean<Boolean>(configService.delete(id));
}
```

用到的技术就是**AOP**，也不是什么高深技术。怎么样？代码量就一行，特性一个都没有丢。这就是我们项目组现在的controller的样子！（如果恰好有我带过的项目组的人，看到ResultBean<>应该很熟悉应该知道我是谁了）

所以说技术无所谓高低，看你怎么样用。上面的代码简单说一下问题，第一，lang和业务没有什么关系，我后面的代码框架去掉了（不是说我后面的代码没有这个功能，是把他隐藏起来对开发人员透明了，使用的技术就是ThreadLocal）。第二，前面那个代码，实际上干活的就只有一行，其他都和业务代码没有一毛钱关系，我的代码框架里面完全看不到了。

使用的技术真的很简单，但是编码效果非常好，因为大家不要因为使用的技术初级就觉得不重要！！使用这套框架后，大家再也不需要大部分时间都写一些无聊的代码，可以有更加多时间学习其他技术。说实话，在我项目组的开发人员都是比较幸运的，觉得能学到东西，不是像其他项目组，写了几年都是一样的CRUD代码，虽然我比较严厉，但是还是愿意待在我项目组，毕竟加班比其他项目组少啊。

这就是我说的工作中，编码习惯（或者说编码风格）比技术更加重要。我工作了也有很长时间了，我觉得我个人价值最大的地方就是这些，技术上其实我懂的也和大家差不多，但编码上我还是觉得可以超过大部分人的。后面我会把我们这些业务系统中大家编码的问题一个一个写出来，并把我的

解决办法分享出来。谢谢大家关注！

接口定义常见问题

工作中，少不了要定义各种接口，系统集成要定义接口，前后台掉调用也要定义接口。接口定义一定程度上能反应程序员的编程功底。列举一下工作中我发现大家容易出现的问题：

返回格式不统一

同一个接口，有时候返回数组，有时候返回单个；成功的时候返回对象，失败的时候返回错误信息字符串。工作中有个系统集成就是这样定义的接口，真是辣眼睛。这个对应代码上，返回的类型是map，json，object，都是不应该的。实际工作中，我们会定义一个统一的格式，就是ResultBean，分页的有另外一个PageResultBean

> 错误范例
>
> 返回map可读性不好，不知道里面是什么

```
@PostMapping("/delete")
public Map<String, Object> delete(long id, String lang) {

}
```

> 错误范例
>
> 成功返回boolean，失败返回string，大忌

```
@PostMapping("/delete")
public Object delete(long id, String lang) {
  try {
    boolean result = configService.delete(id, local);
    return result;
  } catch (Exception e) {
    log.error(e);
    return e.toString();
  }
}
```

没有考虑失败情况

一开始只考虑成功场景，等后面测试发现有错误情况，怎么办，改接口呗，前后台都改，劳民伤财无用功。

> 错误范例
>
> 不返回任何数据，没有考虑失败场景，容易返工

```
@PostMapping("/update")
public void update(long id, xxx) {

}
```

出现和业务无关的输入参数

如lang语言，当前用户信息 都不应该出现参数里面，应该从当前会话里面获取。后面讲ThreadLocal会说到怎么样去掉。除了代码可读性不好问题外，尤其是参数出现当前用户信息的，这是个严重问题。

> 错误范例
>
> 当前用户删除数据）参数出现lang和userid，尤其是userid，大忌

```
@PostMapping("/delete")
public Map<String, Object> delete(long id, String lang, String userId) {

}
```

出现复杂的参数

一般情况下，不允许出现例如json字符串这样的参数，这种参数可读性极差。应该定义对应的bean。

> 错误范例
>
> 参数出现json格式，可读性不好，代码也难看

```
@PostMapping("/update")
public Map<String, Object> update(long id, String jsonStr) {

}
```

没有返回应该返回的数据

例如，新增接口一般情况下应该返回新对象的id标识，这需要编程经验。新手定义的时候因为前台没有用就不返回数据或者只返回true，这都是不恰当的。别人要不要是别人的事情，你该返回的还是应该返回。

> 错误范例
>
> 约定俗成，新建应该返回新对象的信息(对象或者ID)，只返回boolean容易导致返工

```
@PostMapping("/add")
public boolean add(xxx) {
  //xxx
  return configService.add();
}
```

> 相关讨论
>
> 当时在知乎上发帖的时候，有人认为返回给前台的信息应该越少越好，避免出现信息泄露问题。这个觉悟是对的，但对象ID并不是敏感信息，这里返回没有问题。

很多人看了我的这篇文章`程序员你为什么这么累？`，都觉得里面的技术也很简单，没有什么特别的地方，但是，实现这个代码框架之前，就是要你的接口的统一的格式ResultBean，aop才好做。有些人误解了，我那篇文章说的都不是技术，重点说的是编码习惯工作方式，如果你重点还是放在什么技术上，那我也帮不了你了。同样，如果我后面的关于习惯和规范的帖子，你重点还是放在技术上的话，那是丢了西瓜捡芝麻，有很多贴还是没有任何技术点呢。

> 晓风轻总结
>
> 统一的接口规范，能帮忙规避很多无用的返工修改和可能出现的问题。能使代码可读性更加好，利于进行aop和自动化测试这些额外工作。大家一定要重视。

Controller规范

第一篇文章中，我贴了2段代码，第一个是原生态的，第2段是我指定了接口定义规范，使用AOP技术之后最终交付的代码，从15行到1行，自己感受一下。今天来说说大家关注的AOP如何实现。

先说说Controller规范，主要的内容是就是接口定义里面的内容，你只要遵循里面的规范，controller就问题不大，除了这些，还有另外的几点：

统一返回ResultBean对象

所有函数返回统一的ResultBean/PageResultBean格式，原因见我的接口定义这个贴。没有统一格式，AOP无法玩，更加重要的是前台代码很不好写。当然类名你可以按照自己喜好随便定义，如就叫Result。

大家都知道，前台代码很难写好做到重用，而我们返回相同数据结构后，前台代码可以这样写（方法handlerResult的重用）：

```
// 查询所有配置项记录
function fetchAllConfigs() {
  $.getJSON('config/all', function(result) {
    handlerResult(result, renderConfigs);
  });
}

// 根据id删除配置项
function deleteConfig(id) {
  $.post('config/delete', {
    id : id
  }, function(result) {
    console.log('delete result', result);
    handlerResult(result, fetchAllConfigs);
  });
}

/**
  * 后台返回相同的数据结构，前台的代码才好写才能重用
  * @param result:  ajax返回的结果
  * @param fn:  成功的处理函数（传入data）
  */
function handlerResult(result, fn) {
  // 成功执行操作，失败提示原因
  if (result.code == 0) {
    fn(result.data);
  }
  // 没有登陆异常，重定向到登陆页面
```

```
    else if (result.code == -1) {
      showError("没有登录");
    }
    // 参数校验出错，直接提示
    else if (result.code == 1) {
      showError(result.msg);
    }
    // 没有权限，显示申请权限电子流
    else if (result.code == 2) {
      showError("没有权限");
    } else {
      // 不应该出现的异常，应该重点关注
      showError(result.msg);
    }
  }
```

ResultBean不允许往后传

ResultBean/PageResultBean是controller专用的，不允许往后传！往其他地方传之后，可读性立马下降，和传map，json好不了多少。

Controller只做参数格式的转换

Controller做参数格式的转换，不允许把json，map这类对象传到services去，也不允许services返回json、map。写过代码都知道，map，json这种格式灵活，但是可读性差（ 编码一时爽，重构火葬场）。如果放业务数据，每次阅读起来都十分困难，需要从头到尾看完才知道里面有什么，是什么格式。定义一个bean看着工作量多了，但代码清晰多了。

参数不允许出现Request，Response 这些对象

和json/map一样，主要是可读性差的问题。一般情况下不允许出现这些参数，除非要操作流。

不需要打印日志

日志在AOP里面会打印，而且我的建议是大部分日志在Services这层打印。

> Contorller只做参数格式转换，如果没有参数需要转换的，那么就一行代码。日志/参数校验/权限判断建议放到service里面，毕竟controller基本无法重用，而service重用较多。而我们的单元测试也不需要测试controller，直接测试service即可。

规范里面大部分是 不要做的项多，要做的比较少，落地比较容易。

AOP实现

我们需要在AOP里面统一处理异常，包装成相同的对象ResultBean给前台。

ResultBean定义

ResultBean定义带泛型，使用了**lombok**。

```
@Data
public class ResultBean<T> implements Serializable {

  private static final long serialVersionUID = 1L;

  public static final int NO_LOGIN = -1;

  public static final int SUCCESS = 0;

  public static final int FAIL = 1;

  public static final int NO_PERMISSION = 2;

  private String msg = "success";

  private int code = SUCCESS;

  private T data;

  public ResultBean() {
    super();
  }

  public ResultBean(T data) {
    super();
    this.data = data;
  }

  public ResultBean(Throwable e) {
    super();
    this.msg = e.toString();
    this.code = FAIL;
  }
}
```

AOP实现

AOP代码，主要就是打印日志和捕获异常，异常要区分已知异常和未知异常，其中未知的异常是我们重点关注的，可以做一些邮件通知啥的，已知异常可以再细分一下，可以不同的异常返回不同的返回码：

```
/**
 * 处理和包装异常
 */
public class ControllerAOP {
  private static final Logger logger = LoggerFactory.getLogger(ControllerAOP.class);

  public Object handlerControllerMethod(ProceedingJoinPoint pjp) {
    long startTime = System.currentTimeMillis();

    ResultBean<?> result;

    try {
      result = (ResultBean<?>) pjp.proceed();
      logger.info(pjp.getSignature() + "use time:" + (System.currentTimeMillis() - star
tTime));
    } catch (Throwable e) {
      result = handlerException(pjp, e);
    }

    return result;
  }

  /**
    * 封装异常信息，注意区分已知异常（自己抛出的）和未知异常
    */
  private ResultBean<?> handlerException(ProceedingJoinPoint pjp, Throwable e) {
    ResultBean<?> result = new ResultBean();

    // 已知异常
    if (e instanceof CheckException) {
      result.setMsg(e.getLocalizedMessage());
      result.setCode(ResultBean.FAIL);
    } else if (e instanceof UnloginException) {
      result.setMsg("Unlogin");
      result.setCode(ResultBean.NO_LOGIN);
    } else {
      logger.error(pjp.getSignature() + " error ", e);
      //TODO 未知的异常，应该格外注意，可以发送邮件通知等
      result.setMsg(e.toString());
      result.setCode(ResultBean.FAIL);
    }

    return result;
  }
```

```
}
```

> 对于未知异常，给相关责任人发送邮件通知，第一时间知道异常，实际工作中非常有意义。

返回码定义

关于怎么样定义返回码，个人经验是 粗分异常，不能太细。如没有登陆返回-1，没有权限返回-2，参数校验错误返回1，其他未知异常返回-99等。需要注意的是，定义的时候，需要调用方单独处理的异常需要和其他区分开来，比如没有登陆这种异常，调用方不需要单独处理，前台调用请求的工具类统一处理即可。而参数校验异常或者没有权限异常需要调用方提示给用户，没有权限可能除了提示还会附上申请权限链接等，这就是异常的粗分。

> 返回码不要太细，千万不要标题为空返回1，描述为空返回2，字段X非法返回3，这种定义看上去很专业，实际上会把前台和自己累死。

AOP配置

关于用java代码还是xml配置，这里我倾向于xml配置，因为这个会不定期改动

```
<!-- aop -->
  <aop:aspectj-autoproxy />
  <beans:bean id="controllerAop" class="xxx.common.aop.ControllerAOP" />
  <aop:config>
    <aop:aspect id="myAop" ref="controllerAop">
      <aop:pointcut id="target"
        expression="execution(public xxx.common.beans.ResultBean *(..))" />
      <aop:around method="handlerControllerMethod" pointcut-ref="target" />
    </aop:aspect>
  </aop:config>
```

现在知道为什么要返回统一的一个ResultBean了：

- 为了统一格式
- 为了应用AOP
- 为了包装异常信息

分页的PageResultBean大同小异，大家自己依葫芦画瓢自己完成就好了。

简单示例

贴一个简单的controller。请对比 程序员你为什么这么累？ 里面原来的代码查看，没有对比就没有伤害。

```
/**
```

```
 * 配置对象处理器
 *
 * @author 晓风轻 https://github.com/xwjie/PLMCodeTemplate
 */
@RequestMapping("/config")
@RestController
public class ConfigController {

  private final ConfigService configService;

  public ConfigController(ConfigService configService) {
    this.configService = configService;
  }

  @GetMapping("/all")
  public ResultBean<Collection<Config>> getAll() {
    return new ResultBean<Collection<Config>>(configService.getAll());
  }

  /**
   * 新增数据，返回新对象的id
   *
   * @param config
   * @return
   */
  @PostMapping("/add")
  public ResultBean<Long> add(Config config) {
    return new ResultBean<Long>(configService.add(config));
  }

  /**
   * 根据id删除对象
   *
   * @param id
   * @return
   */
  @PostMapping("/delete")
  public ResultBean<Boolean> delete(long id) {
    return new ResultBean<Boolean>(configService.delete(id));
  }

  @PostMapping("/update")
  public ResultBean<Boolean> update(Config config) {
    configService.update(config);
    return new ResultBean<Boolean>(true);
  }
}
```

为什么不用**ExceptionHandler**

这是我发帖后问的最多的一个问题，很多人说为什么不用 ControllerAdvice + ExceptionHandler 来处理异常？觉得是我在重复发明轮子。首先，这2这都是AOP，本质上没有啥区别。而最重要的是ExceptionHandler只能处理异常，而我们的AOP除了处理异常，还有一个很重要的作用是打印日志，统计每一个controller方法的耗时，这在实际工作中也非常重要和有用的特性！

> 就算你使用ExceptionHandler，也不要成功和失败的时候返回不一样的数据格式，否则前台很难写好代码。

为什么不用**Restful**风格

这也是问的比较多的一个问题。如果你提供的接口是给前台调用的，而你又在实际工作中前后台开发都负责的话，我觉得你应该不会问这个问题。诚然，restful风格的定义很优雅，但是在前台调用起来却非常的麻烦，前台通过返回的ResultBean的code来判断成功失败显然比通过http状态码来判断方便太多。第2个原因，使用http状态码返回出错信息也值得商榷。系统出错了返回400我觉得没有问题，但一个参数校验不通过也返回400，我个人觉得是很不合理的，是无法接受的。

> 有统一的接口定义规范，然后有**AOP**实现，先有思想再有技术。技术不是关键，AOP技术也很简单，这个帖子的关键点不是技术，而是习惯和思想，不要捡了芝麻丢了西瓜。网络上讲技术的贴多，讲习惯、风格的少，这些都是我工作多年的行之有效的经验之谈，望有缘人珍惜。

日志打印

开发中日志这个问题，每个公司都强调，也制定了一大堆规范，但根据实际情况看，效果不是很明显，主要是这个东西不好测试和考核，没有日志功能一样跑啊。

但编程活久见，开发久了，总会遇到“这个问题生产环境上能重现，但是没有日志，业务很复杂，不知道哪一步出错了？”这个时候，怎么办？ 还能怎么办，发个版本，就是把所有地方加上日志，没有任何新功能，然后在让用户重现一遍，拿下日志来看，哦，原来是这个问题。

有没有很熟悉的感觉？

还有一种情况，我们系统有3*5=15个节点，出了问题找日志真是痛苦，一个一个机器翻，N分钟后终于找到了，找到了后发现好多相似日志，一个一个排查；日志有了，发现逻辑很复杂，不知道走到那个分支，只能根据逻辑分析，半天过去了，终于找到了原因。。。一个问题定位就过去了2个小时，变更时间过去了一半。。。

日志要求

所以我对日志的最少有以下2点要求：

1 能找到那个机器

2 能找到用户做了什么

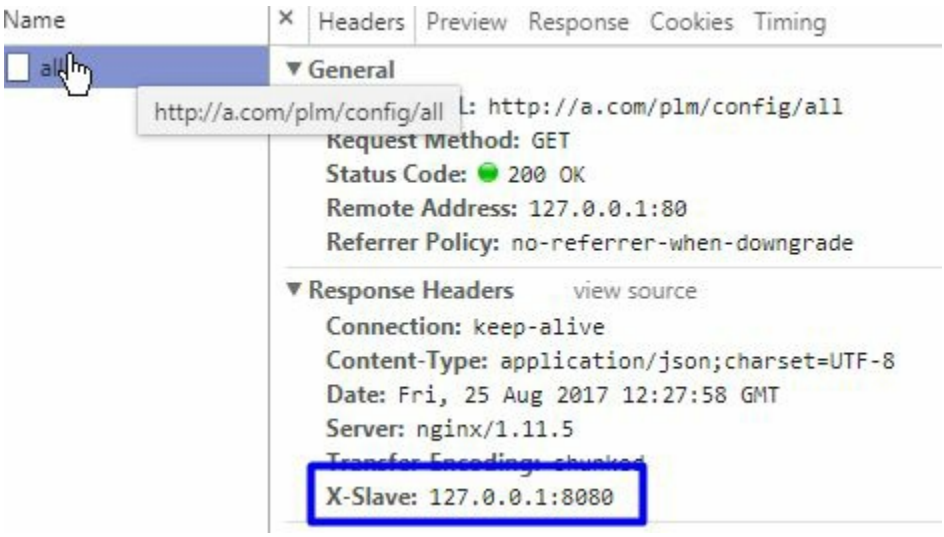
配置**nginx**

针对第一点，我修改了一下nginx的配置文件，让返回头里面返回是那个机器处理的。

nginx的基本配置，大家查阅一下资料就知道。简单配置如下（生产环境比这个完善）

```
upstream serverlist{
  server localhost:8080;
  server localhost:8081;
}

server {
  listen       80;
  server_name  a.com;

  location / {
    proxy_pass http://serverlist/;
    add_header X-Slave $upstream_addr;
  }
}
```

效果如图，返回了处理的节点：



第二点，要知道用户做了什么。用户信息是很重要的一个信息，能帮助海量日志里面能快速找到目标日志。一开始要求开发人员打印的时候带上用户，但是发现这个落地不容易，开发人员打印日志都经常忘记，更加不用说日志上加上用户信息，我也不可能天天看代码。所以找了一下log4j的配置，果然log4j有个叫MDC(Mapped Diagnostic Context)的类（技术上使用了ThreadLocal实现，重点技术）。具体使用方法请自行查询。具体使用如下：

UserFilter

filter中得到用户信息，并放入MDC，记住filter后要清理掉（因为tomcat线程池线程重用的原因）。

```
/**
 * 用户信息相关的filter
```

```
 *
 * @author 晓风轻 https://xwjie.github.io/PLMCodeTemplate/
 *
 */
public class UserFilter implements Filter {

  @Override
  public void init(FilterConfig filterConfig) throws ServletException {

  }

  @Override
  public void doFilter(ServletRequest request, ServletResponse response,
      FilterChain chain) throws IOException, ServletException {

    // 得到用户个人相关的信息（登陆的用户，用户的语言）
    fillUserInfo((HttpServletRequest) request);

    try {
      chain.doFilter(request, response);
    } finally {
      // 由于tomcat线程重用，记得清空
      clearAllUserInfo();
    }
  }

  private void clearAllUserInfo() {
    UserUtil.clearAllUserInfo();
  }

  private void fillUserInfo(HttpServletRequest request) {
    // 用户信息
    String user = getUserFromSession(request);

    if (user != null) {
      UserUtil.setUser(user);
    }

    // 语言信息
    String locale = getLocaleFromCookies(request);

    // 放入到threadlocal，同一个线程任何地方都可以拿出来
    if (locale != null) {
      UserUtil.setLocale(locale);
    }
  }

  private String getLocaleFromCookies(HttpServletRequest request) {
    Cookie[] cookies = request.getCookies();
```

```
    if (cookies == null) {
      return null;
    }

    for (int i = 0; i < cookies.length; i++) {
      if (UserUtil.KEY_LANG.equals(cookies[i].getName())) {
        return cookies[i].getValue();
      }
    }

    return null;
  }

  private String getUserFromSession(HttpServletRequest request) {
    HttpSession session = request.getSession(false);

    if (session == null) {
      return null;
    }

    // 从session中获取用户信息放到工具类中
    return (String) session.getAttribute(UserUtil.KEY_USER);
  }

  @Override
  public void destroy() {

  }

}
```

用户工具类

用户信息放入MDC：

```
/**
 * 用户工具类
 *
 * @author 晓风轻 https://xwjie.github.io/PLMCodeTemplate/
 *
 */
public class UserUtil {

  private final static ThreadLocal<String> tlUser = new ThreadLocal<String>();

  private final static ThreadLocal<Locale> tlLocale = new ThreadLocal<Locale>() {
```

```
    protected Locale initialValue() {
      // 语言的默认值
      return Locale.CHINESE;
    };
  };

  public static final String KEY_LANG = "lang";

  public static final String KEY_USER = "user";

  public static void setUser(String userid) {
    tlUser.set(userid);

    // 把用户信息放到log4j
    MDC.put(KEY_USER, userid);
  }

  /**
   * 如果没有登录，返回null
   *
   * @return
   */
  public static String getUserIfLogin() {
    return tlUser.get();
  }

  /**
   * 如果没有登录会抛出异常
   *
   * @return
   */
  public static String getUser() {
    String user = tlUser.get();

    if (user == null) {
      throw new UnloginException();
    }

    return user;
  }

  public static void setLocale(String locale) {
    setLocale(new Locale(locale));
  }

  public static void setLocale(Locale locale) {
    tlLocale.set(locale);
  }
```

```
  public static Locale getLocale() {
    return tlLocale.get();
  }

  public static void clearAllUserInfo() {
    tlUser.remove();
    tlLocale.remove();

    MDC.remove(KEY_USER);
  }
}
```

log4j配置

增加用户信息变量，%X{user}

```
<!-- Appenders -->
<appender name="console" class="org.apache.log4j.ConsoleAppender">
  <param name="Target" value="System.out" />
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern"
      value="[%t]%-d{MM-dd HH:mm:ss,SSS} %-5p:%X{user} - %c - %m%n" />
  </layout>
</appender>
```

日志要求

我做好上面2步后，对开发人员的日志只有3点要求：

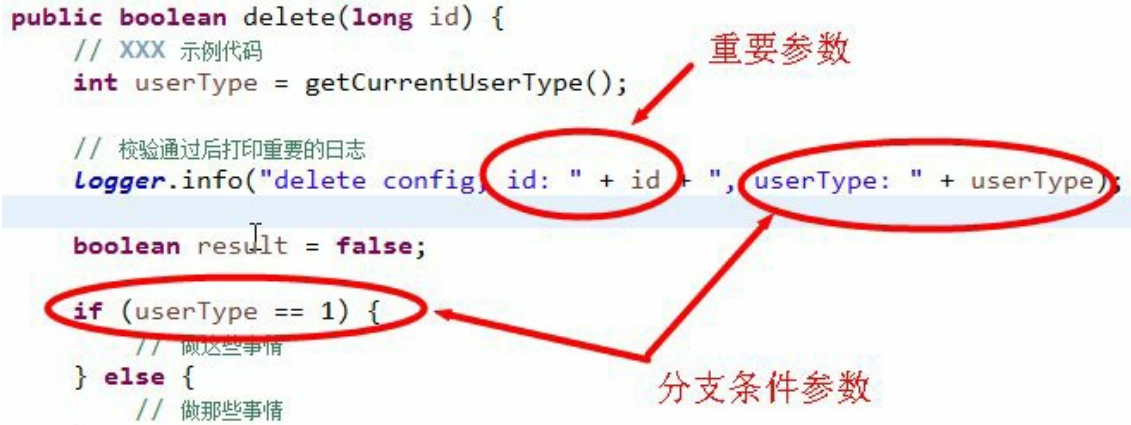
1. 修改（包括新增）操作必须打印日志

大部分问题都是修改导致的。数据修改必须有据可查。

1. 条件分支必须打印条件值，重要参数必须打印

尤其是分支条件的参数，打印后就不用分析和猜测走那个分支了，很重要！如下面代码里面的userType，一定要打印值，因为他决定了代码走那个分支。

```java
public boolean delete(long id) {
    // XXX 示例代码
    int userType = getCurrentUserType();

    // 校验通过后打印重要的日志
    logger.info("delete config, id: " + id + ", userType: " + userType);

    boolean result = false;

    if (userType == 1) {
        // 做这些事情
    } else {
        // 做那些事情
    }

    // 修改操作需要打印操作结果
    logger.info("delete config success, id:" + id + ", result:" + result);

    return result; // 示例代码
}

private int getCurrentUserType() {
    return 2;
}
```

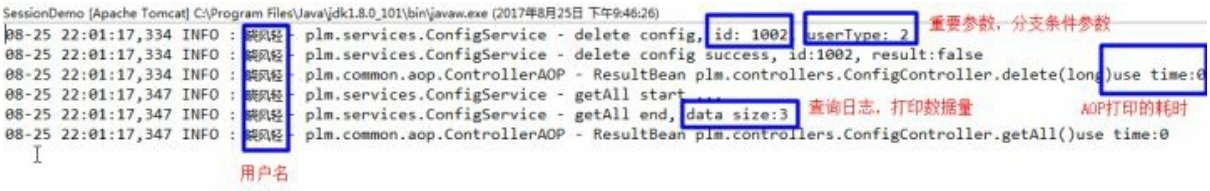


1. 数据量大的时候需要打印数据量

前后打印日志和最后的数据量，主要用于分析性能，能从日志中知道查询了多少数据用了多久。这点是建议。自己视情况而决定是否打印，我一般建议打印。

日志效果图

加上 我的编码习惯 - Controller规范 这篇文章的AOP，最后的日志如下：



其实日志的级别我到不是很关注，还没有到关注这步到时候。开发组长需要做好后勤工作（前面2步），然后制定简单规则，规则太多太能落实了。

新手建议

日志这个东西，更多是靠自觉，项目组这么多人，我也不可能一个一个给大家看代码，然后叫你加日志。我分析了一下，为什么有些人没有打印日志的习惯，说了多次都改不过来。我建议大家养成下面的习惯，这样你的日志就会改善多了！

1.不要依赖debug，多依赖日志。

别人面对对象编程，你面对debug编程。有些人无论什么语言，最后都变成了面对debug编程。哈哈。这个习惯非常非常不好！debug会让你写代码的时候偷懒不打日志，而且很浪费时间。改掉这个恶习。

1. 代码开发测试完成之后不要急着提交，先跑一遍看看日志是否看得懂。

日志是给人看的，只要热爱编程的人才能成为合格程序员，不要匆匆忙忙写完功能测试ok就提交代码，日志也是功能的一部分。要有精益求精的工匠精神！

异常处理

对于大型IT系统，最怕的事情第一是系统出现了异常我不知道，等问题闹大了用户投诉了才知道出问题了。第二就是出了问题之后无法找到出错原因。针对这2个问题，说说我们项目组是怎么样规定异常处理的。

再次声明我的观点，我这系列贴里面，没有什么技术点，都是一些编程的经验之谈，而且是建立在项目背景是大部分代码都是简单的CRUD、开发人员流动大水平一般的情况下。希望读者的重点不要再关注技术点。大部分工作中不需要什么技术，你只要把代码写好，足够你轻松面对！

言归正传，说回第一个问题，系统出异常了我不知道，等问题闹大了用户投诉了才知道。这个问题出现非常多，而且非常严重。我不知道其他公司有没有这种场景，对我们公司而言，经常会出现用户反馈、投诉过来说某个功能不可用，开发人员定位分析之后，才发现之前的某一步出错了。公司业务流程非常复杂，和周边系统一堆集成，一堆的后台队列任务，任何一部都可能出问题。

举几个今年真实的案例：

1 某系统销户无法成功，最后定位发现前段时间ldap密码修改没有更新

1. 某个流程失败，最后发现集成的系统新增加了NAS盘，防火墙不通无法访问导致报错。

3、某个功能无法使用，查看日志发现后台定时任务已经停了好几天。

针对这些功能，在流程上当然可以采取相对的策略来保证，但从开发的角度来说，任何规定都无法保证一定不会发生错误，老虎也有打盹的时候，我只相信代码。

错误范例

贴一段非常常见的代码，大家觉得这段代码有没有问题？

在我看来，这段代码很多时候问题特别大！

- 丢掉了异常

异常就算打印了堆栈，也不会有人去看的！除非用户告诉你出问题了，你才会去找日志！所以，看着好像很严谨的代码，其实作用并不大。

- 空判断隐藏了错误

  异常处理再加上框框2处的空判断，天衣无缝的避开了所有正确答案。本来需要更新文档，结果什么错误没有报，什么也没有做。你后台就算打了日志堆栈又怎么样？

所以，我对开发人员的要求就是，绝大部分场景，不允许捕获异常，不要乱加空判断。只有明显不需要关心的异常，如关闭资源的时候的io异常，可以捕获然后什么都不干，其他时候，不允许捕获异常，都抛出去，到controller处理。空判断大部分时候不需要，你如果写了空判断，你就必须测试为空和不为空二种场景，要么就不要写空判断。

强调，有些空判断是要的，如：参数是用户输入的情况下。但是，大部分场景是不需要的（我们的IT系统里面，一半以上不需要），如参数是其它系统传过来，或者其他地方获取的传过来的，99.99%都不会为空，你判断来干嘛？就抛一个空指针到前台怎么啦？何况基本上不会出现。

新手最容易犯的错误，到处捕获异常，到处加空判断，自以为写出了“健壮”的代码，实际上完全相反。导致的问题，第一代码可读性很差，你如果工作了看到一半代码是try-catch和空判断你会同意我的观点的，第二更加重要的掩盖了很多错误，如上面图片的例子！日志是不会有人看的，我们的目的是尽早让错误抛出来，还有，你加了空判断，那你测试过为空的场景吗？

web请求上的异常，不允许开发人员捕获，直接抛到前台，会有controller处理！见之前的Controller规范

所以上面的代码，我来写的话是这样的，清晰明了。

```
private void updateDocInfo(long id) {
    Document doc = getDocById(id);

    log.info("update doc, id=" + id);

    // dosomething
    doUpdateDoc(doc);
}
```

另外一种后台定时任务队列的异常，其实思路是一样的，有个统一的地方处理异常，里面的代码同样不准捕获异常！然后异常的时候邮件通知到我和开发人员，开发组长必须知道后台的任何异常，不要等用户投诉了才知道系统出问题了。

另外，开发组长需要自己定义好系统里面的异常，其实能定义的没有几种，太细了很难落地，还有，异常不要继承Exception，而是继承RuntimeException，否则到时候从头改到尾就为了加个异常声明你就觉得很无聊。

异常处理要求

- 开发组长定义好异常，异常继承RuntimeException。
- 不允许开发人员捕获异常。

异常上对开发人员就这点要求！异常都抛出到controller上用AOP处理。后台（如队列等）异常一定要有通知机制，要第一时间知道异常。

- 少加空判断，加了空判断就要测试为空的场景！

这篇文章，我估计一定有很多争议，这些规则都和常见的认识相反，我在公司里面推广和写贴分享的时候也有人反对。但是，你要知道你遇到的是什么问题，要解决的是什么问题？我遇到是很多异常本来很简单，但由于一堆“健壮”的try-catch和空判断，导致问题发现很晚，可能很小一个问题最后变成了一个大事件，在一些IT系统里面，尤其常见。大家不要理解为不能加空判断，大家见仁见智吧。反正我是这样写代码的，我发现效果很好，我很少花时间在调试代码和改bug上，更加不会出现前台返回成功，后台有异常什么也没有做的场景。

最后对新手说一句，不要养成到处**try-catch**和加空判断的恶习，你这样会掩盖掉很多错误，给人埋很多坑的！

参数校验和国际化

今天我们说说参数校验和国际化，这些代码没有什么技术含量，却大量充斥在业务代码上，很可能业务代码只有几行，参数校验代码却有十几行，非常影响代码阅读，所以很有必要把这块的代码量减下去。

今天的目的主要是把之前例子里面的和业务无关的国际化参数隐藏掉，以及如何封装好校验函数。

修改前代码

- controller代码

```
/**
* ！！！ 错误范例
*
*  根据id删除对象
*
* @param id
* @param lang
* @return
*/
@PostMapping("/delete")
public Map<String, Object> delete(long id, String lang) {
Map<String, Object> data = new HashMap<String, Object>();

boolean result = false;
try {
  // 语言（中英文提示不同）
  Locale local = "zh".equalsIgnoreCase(lang) ? Locale.CHINESE
      : Locale.ENGLISH;

  result = configService.delete(id, local);

  data.put("code", 0);

} catch (CheckException e) {
  // 参数等校验出错，已知异常，不需要打印堆栈，返回码为-1
  data.put("code", -1);
  data.put("msg", e.getMessage());
} catch (Exception e) {
  // 其他未知异常，需要打印堆栈分析用，返回码为99
  log.error("delete config error", e);

  data.put("code", 99);
  data.put("msg", e.toString());
```

```
    }

    data.put("result", result);

    return data;
    }
```

其中的lang参数我们需要去掉

- service代码

```
/**
 * ！！！错误示范
 *
 * 出现和业务无关的参数local
 *
 * @param id
 * @param locale
 * @return
 */
public boolean delete(long id, Locale locale) {
  // 参数校验
  if (id <= 0L) {
    if (locale.equals(Locale.CHINESE)) {
      throw new CheckException("非法的ID：" + id);
    } else {
      throw new CheckException("Illegal ID:" + id);
    }
  }

  boolean result = dao.delete(id);

  // 修改操作需要打印操作结果
  logger.info("delete config success, id:" + id + ", result:" + result);

  return dao.delete(id);
}
```

修改后代码

- controller代码

```
/**
 * 根据id删除对象
 *
 * @param id
```

```
 * @return
 */
@PostMapping("/delete")
public ResultBean<Boolean> delete(long id) {
  return new ResultBean<Boolean>(configService.delete(id));
}
```

- service代码

```
public boolean delete(long id) {
  // 参数校验
  check(id > 0L, "id.error", id);

  boolean result = dao.delete(id);

  // 修改操作需要打印操作结果
  logger.info("delete config success, id: {}, result: {}", id, result);

  return result;
}
```

Controll的非业务代码如何去掉参考 Controller规范，下面说说去掉Local参数。

> 业务代码里面不要出现和业务无关的东西，如local，MessageSource 。

去掉国际化参数还是使用的技术还是**ThreadLocal**。国际化信息可以放好几个地方，但建议不要放在每一个url上，除了比较low还容易出很多其他问题。这里演示的是放在cookie上面的例子：

用户工具类**UserUtil**

需要保存用户的国际化信息。

```
public class UserUtil {

  private final static ThreadLocal<String> tlUser = new ThreadLocal<String>();

  private final static ThreadLocal<Locale> tlLocale = new ThreadLocal<Locale>() {
    protected Locale initialValue() {
      // 语言的默认值
      return Locale.CHINESE;
    };
  };

  public static final String KEY_LANG = "lang";

  public static final String KEY_USER = "user";
```

```
  public static void setUser(String userid) {
    tlUser.set(userid);

    // 把用户信息放到log4j
    MDC.put(KEY_USER, userid);
  }

  public static String getUser() {
    return tlUser.get();
  }

  public static void setLocale(String locale) {
    setLocale(new Locale(locale));
  }

  public static void setLocale(Locale locale) {
    tlLocale.set(locale);
  }

  public static Locale getLocale() {
    return tlLocale.get();
  }

  public static void clearAllUserInfo() {
    tlUser.remove();
    tlLocale.remove();

    MDC.remove(KEY_USER);
  }
}
```

校验工具类**CheckUtil**

这里需要调用用户工具类得到用户的语言。还有就是提示信息里面，需要支持传入变量。

```
package plm.common.utils;

import org.springframework.context.MessageSource;

import plm.common.exceptions.CheckException;

public class CheckUtil {
  private static MessageSource resources;

  public static void setResources(MessageSource resources) {
    CheckUtil.resources = resources;
```

```
  }

  public static void check(boolean condition, String msgKey, Object... args) {
    if (!condition) {
      fail(msgKey, args);
    }
  }

  public static void notEmpty(String str, String msgKey, Object... args) {
    if (str == null || str.isEmpty()) {
      fail(msgKey, args);
    }
  }

  public static void notNull(Object obj, String msgKey, Object... args) {
    if (obj == null) {
      fail(msgKey, args);
    }
  }

  private static void fail(String msgKey, Object... args) {
    throw new CheckException(resources.getMessage(msgKey, args, UserUtil.getLocale()));
  }
}
```

这里有几个小技术点：

spring的静态方法注入

工具类里面使用spring的bean，使用了MethodInvokingFactoryBean的静态方法注入：

```
<!-- 国际化 -->
<bean id="messageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>format</value>
      <value>exceptions</value>
      <value>windows</value>
    </list>
  </property>
</bean>

<bean
  class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="staticMethod" value="plm.common.utils.CheckUtil.setResources" />
  <!-- 这里配置参数 -->
```

```
    <property name="arguments" ref="messageSource">
    </property>
</bean>
```

jdk 的 import static

server里面调用 `check` 方法的时候没有出现类名。这里使用的jdk的import static 特性，可以在ide上配置，请自行google。

```
import static plm.common.utils.CheckUtil.*;
```

还有一小点注意，我建议参数非法的时候，把非法值打印出来，否则你又要浪费时间看是没有传呢还是传错了，时间就是这样一点点浪费的。

```
check(id > 0L, "id.error", id); // 当前非法的id也传入提示出去
```

另外有些项目用valid来校验，从我实际接触来看，用的不多，可能是有短木板吧。如果你的项目valid就能满足，那就更加好了，不需要看了。但是大部分场景，校验比例子复杂N多，提示也千变万化，所以我们还是自己调用函数校验。

做了这几步之后，代码会漂亮很多，记住，代码最主要的不是性能，而是可读性，有了可读性才有才维护性。而去掉无关的代码后的代码，和之前的代码对比一下，自己看吧。

工具类编写

一个项目不可能没有工具类，工具类的初衷是良好的，代码重用，但到了后面工具类越来越乱，有些项目工具类有几十个，看的眼花缭乱，还有不少重复。如何编写出好的工具类，我有几点建议：

隐藏实现

就是要定义自己的工具类，尽量不要在业务代码里面直接调用第三方的工具类。这也是解耦的一种体现。

如果我们不定义自己的工具类而是直接使用第三方的工具类有2个不好的地方：

- 不同的人会使用不同的第三方工具库，会比较乱。
- 将来万一要修改工具类的实现逻辑会很痛苦。

以最简单的字符串判空为例，很多工具库都有 `StringUtils` 工具类，如果我们使用 `commons` 的工具类，一开始我们直接使用 `StringUtils.isEmpty` ，字符串为空或者空串的时候会返回为true，后面业务改动，需要改成如果全部是空格的时候也会返回true，怎么办？我们可以改成使用 `StringUtils.isBlank` 。看上去很简单，对吧？ 如果你有几十个文件都调用了，那我们要改几十个文件，是不是有点恶心？再后面发现，不只是英文空格，如果是全角的空格，也要返回为true，怎么办？StringUtils上的方法已经不能满足我们的需求了，真不好改了。。。

所以我的建议是，一开始就自己定义一个自己项目的 `StringUtil` ，里面如果不想自己写实现，可以直接调用 `commons` 的方法，如下：

```
public static boolean isEmpty(String str) {
  return org.apache.commons.lang3.StringUtils.isEmpty(str);
}
```

后面全部空格也返回true的时候，我们只需要把isEmpty改成isBlank；再后面全部全角空格的时候也返回true的话，我们增加自己的逻辑即可。我们只需要改动和测试一个地方。

再举一个真实一点的例子，如复制对象的属性方法。

一开始，如果我们自己不定义工具类方法，那么我们可以使用 `org.springframework.beans.BeanUtils.copyProperties(source, dest)` 这个工具类来实现，就一行代码，和调用自己的工具类没有什么区别。看上去很OK，对吧？

随着业务发展，我们发现这个方式的性能或者某些特性不符合我们要求，我们需要修改改成 `commons-beanutils` 包里面的方法， `org.apache.commons.beanutils.BeanUtils.copyProperties(dest,source)` ，这个时候问题来了，第一个问题，它的方法的参数顺序和之前spring的工具类是相反的，改起来非常容易出错！第

二个问题，这个方法有异常抛出，必须声明，这个改起来可要命了！结果你发现，一个看上去很小的改动，改了几十个文件，每个改动还得测试一次，风险不是那么得小。有一点小崩溃了，是不是？

等你改完之后测试完了，突然有一天需要改成，复制参数的时候，有些特殊字段需要保留（如对象id）或者需要过滤掉（如密码）不复制，怎么办？这个时候我估计你要崩溃了吧？不要觉得我是凭空想象，编程活久见，你总会遇到的一天！

所以，我们需要定义自己的工具类函数，一开始我定义成这样子。

```
public static void copyAttribute(Object source, Object dest) {
  org.springframework.beans.BeanUtils.copyProperties(source, dest);
}
```

后面需要修改为 `commons-beanutis` 的时候，我们改成这样即可，把参数顺序掉过来，然后处理了一下异常，我使用的是 `Lombok` 的 `@SneakyThrows` 来保证异常，你也可以捕获掉抛出运行时异常，个人喜好。

```
@SneakyThrows
public static void copyAttribute(Object source, Object dest) {
  org.apache.commons.beanutils.BeanUtils.copyProperties(dest, source);
}
```

再后面，复制属性的时候需要保留某些字段或者过滤掉某些字段，我们自己参考其他库实现一次即可，只改动和测试一个文件一个方法，风险非常可控。

还记得我之前的帖子里说的需求变更吗？你可以认为这算需求变更，但同样的需求变更，我一个小时改完测试（因为我只改一个文件），没有任何风险轻轻松松上线，你可能满头大汗加班加点还担心出问题。。。

使用父类/接口

上面那点隐藏实现，说到底是封装/解耦的思想，而现在说的这点是抽象的思想，做好了这点，我们就能编写出看上去很专业的工具类。这点很好理解也很容易做到，但是我们容易忽略。

举例，假设我们写了一个判断arraylist是否为空的函数，一开始是这样的。

```
public static boolean isEmpty(ArrayList<?> list) {
  return list == null || list.size() == 0;
}
```

这个时候，我们需要思考一下参数的类型能不能使用父类。我们看到我们只用了size方法，我们可以知道size方法再list接口上有，于是我们修改成这样。

```
public static boolean isEmpty(List<?> list) {
  return list == null || list.size() == 0;
}
```

后面发现，size方法再list的父类/接口Collection上也有，那么我们可以修改为最终这样。

```
public static boolean isEmpty(Collection<?> list) {
  return list == null || list.size() == 0;
}
```

到了这部，Collection没有父类/接口有size方法了，修改就结束了。最后我们需要把参数名字改一下，不要再使用list。改完后，所有实现了Collection都对象都可以用，最终版本如下：

```
public static boolean isEmpty(Collection<?> collection) {
  return collection == null || collection.size() == 0;
}
```

是不是看上去通用多了，看上去也专业多了？上面的string相关的工具类方法，使用相同的思路，我们最终修改一下，把参数类类型由String修改为CharSequence，参数名str修改为cs。如下：

```
public static boolean isEmpty(CharSequence cs) {
  return org.apache.commons.lang3.StringUtils.isEmpty(cs);
}
```

思路和方法很简单，但效果很好，写出来的工具类也显得很专业！总结一下，思路是抽象的思想，主要是修改参数类型，方法就是往上找父类/接口，一直找到顶为止，记得修改参数名。

使用重载编写衍生函数组

开发过的兄弟都知道，有一些工具库，有一堆的重载函数，调用起来非常方便，经常能直接调用，不需要做参数转换。这些是怎么样编写出来的呢？我们举例说明。

现在需要编写一个方法，输入是一个utf-8格式的文件的文件名，把里面内容输出到一个list<String>。我们刚刚开始编写的时候，是这个样子的

```
public static List<String> readFile2List(String filename) throws IOException {
  List<String> list = new ArrayList<String>();

  File file = new File(filename);

  FileInputStream fileInputStream = new FileInputStream(file);

  BufferedReader br = new BufferedReader(new InputStreamReader(fileInputStream,
```

```
      "UTF-8"));

  // XXX操作

  return list;
}
```

我们先实现，实现完之后我们做第一个修改，很明显，utf-8格式是很可能要改的，所以我们先把它做为参数提取出去，方法一拆为二，就变成这样。

```
public static List<String> readFile2List(String filename) throws IOException {
  return readFile2List(filename, "UTF-8");
}

public static List<String> readFile2List(String filename, String charset)
  throws IOException {
  List<String> list = new ArrayList<String>();

  File file = new File(filename);
  FileInputStream fileInputStream = new FileInputStream(file);

  BufferedReader br = new BufferedReader(new InputStreamReader(fileInputStream,
    charset));

  // XXX操作

  return list;
}
```

多了一个方法，直接调用之前的方法主体，主要的代码还是只有一份，之前的调用地方不需要做任何修改！可以放心修改。

然后我们在看里面的实现，下面这2行代码里面，String类型的filename会变化为File类型，然后在变化为FileInputStream 类型之后才使用。

```
File file = new File(filename);
FileInputStream fileInputStream = new FileInputStream(file);
```

这里我们就应该想到，用户可能直接传如File类型，也可能直接传入FileInputStream类型，我们应该都需要支持，而不需要用户自己做类型的处理！在结合上一点的使用父类，把FileInputStream改成父类InputStream，我们最终的方法组如下：

```
package plm.common.utils;

import java.io.BufferedReader;
import java.io.File;
```

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

import org.apache.commons.io.IOUtils;

/**
 * 工具类编写范例，使用重载编写不同参数类型的函数组
 *
 * @author 晓风轻 https://github.com/xwjie/PLMCodeTemplate
 *
 */
public class FileUtil {

  private static final String DEFAULT_CHARSET = "UTF-8";

  public static List<String> readFile2List(String filename) throws IOException {
    return readFile2List(filename, DEFAULT_CHARSET);
  }

  public static List<String> readFile2List(String filename, String charset)
    throws IOException {
    FileInputStream fileInputStream = new FileInputStream(filename);
    return readFile2List(fileInputStream, charset);
  }

  public static List<String> readFile2List(File file) throws IOException {
    return readFile2List(file, DEFAULT_CHARSET);
  }

  public static List<String> readFile2List(File file, String charset)
    throws IOException {
    FileInputStream fileInputStream = new FileInputStream(file);
    return readFile2List(fileInputStream, charset);
  }

  public static List<String> readFile2List(InputStream fileInputStream)
    throws IOException {
    return readFile2List(fileInputStream, DEFAULT_CHARSET);
  }

  public static List<String> readFile2List(InputStream inputStream, String charset)
    throws IOException {
    List<String> list = new ArrayList<String>();

    BufferedReader br = null;
```

```
    try {
      br = new BufferedReader(new InputStreamReader(inputStream, charset));

      String s = null;
      while ((s = br.readLine()) != null) {
        list.add(s);
      }
    } finally {
      IOUtils.closeQuietly(br);
    }

    return list;
  }

}
```

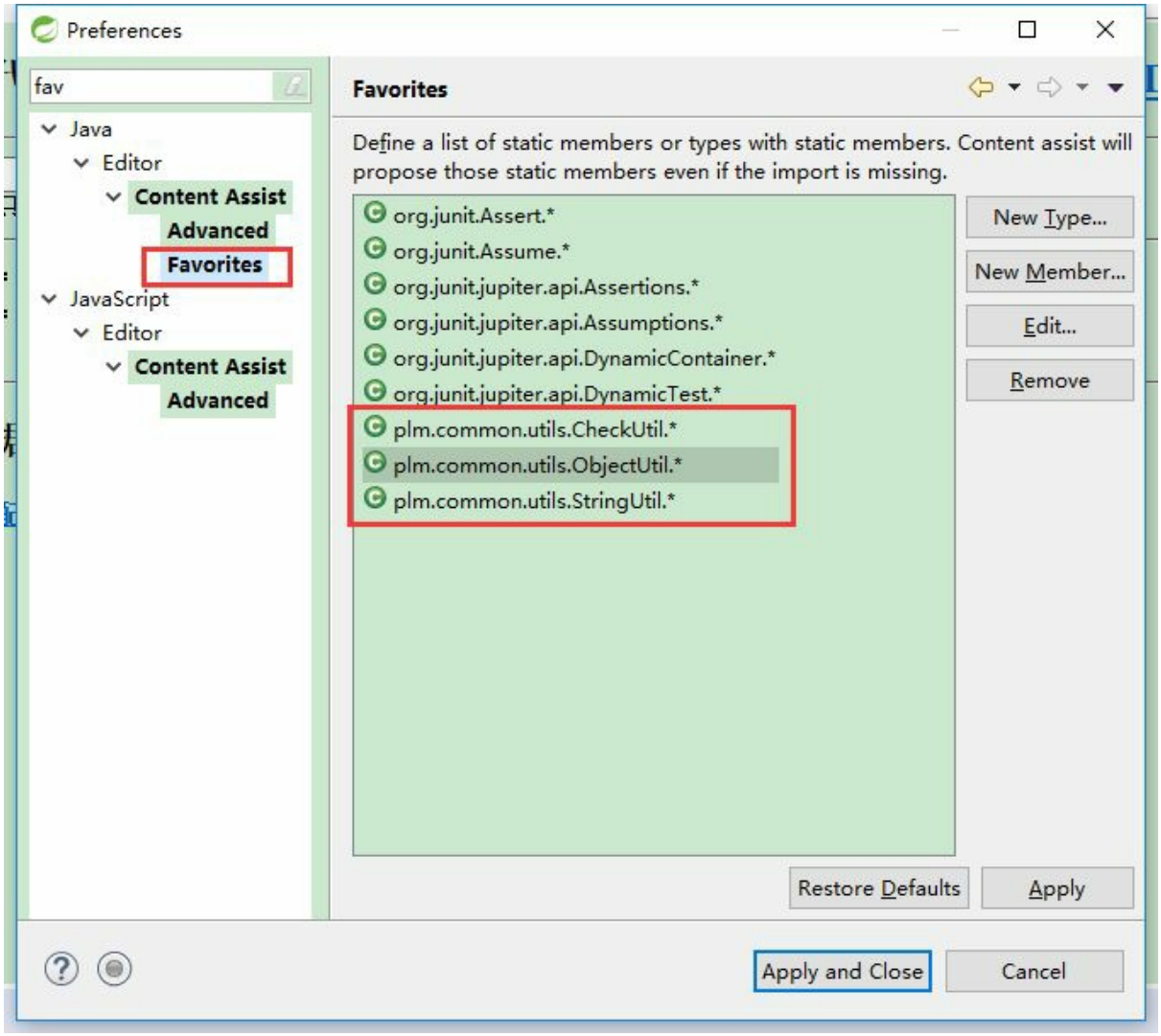
怎么样？6个方法，实际上代码主体只有一份，但提供各种类型的入参，调用起来很方便。开发组长编写的时候，多费一点点时间，就能写来看上去很专业调用起来很方便的代码。如果开发组长不写好，开发人员发现现有的方法只能传String，她要传的是InputStream，她又不敢改原来的代码，就会copy一份然后修改一下，就多了一份重复代码。代码就是这样烂下去了。。。

> 晓风轻建议
>
> 多想一步，根据参数变化编写各种类型的入参函数，需要保证函数主要代码只有一份。

使用静态引入

工具类的一个问题就是容易泛滥，主要原因是开发人员找不到自己要用的方法，就自己写一个，开发人员很难记住类名，你也不可能天天代码评审。

所以要让开发人员容易找到，我们可以使用静态引入，在Eclipse里面这样导入：

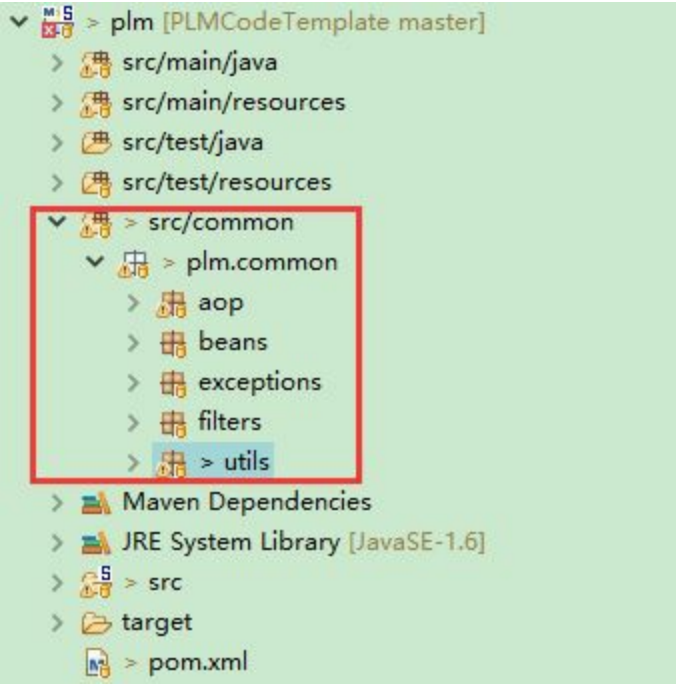


这样，任何地方开发人员只要一敲就可以出来，然后再约定一下项目组方法名规范，这样工具类的使用就会简单很多！

物理上独立存放

这点是我的习惯，我习惯把和业务无关的代码放到独立的工程或者目录，在物理上要分开，专人维护。不是所有人都有能力写工具类，独立存放专门维护，专门的权限控制有助于保证代码的纯洁和质量。这样普通的开发人员就不会随意修改。

例如我的范例工程里面，专门建立了一个source目录存放框架代码，工具类也在里面，这里的代码，只有我一个人会去修改：



> 晓风轻总结
>
> 几乎所有人都知道面向对象的思想有抽象封装，但几个人真正能做到，其实有心的话，处处都能体现出这些思想。编写工具类的时候需要注意参数的优化，而且大型项目里面不要在业务代码里面直接调用第三方的工具类，然后就是多想一步多走一步，考虑各种类型的入参，这样你也能编写出专业灵活的工具类！

函数编写建议

> 傻瓜都能写出计算机可以读懂的代码，只有优秀的程序员才能写出人能读懂的代码！

在我看来，编写简单的函数是一件简单又困难的事情。简单是因为这没有什么技术难点，困难是因为这是一种思维习惯，很难养成，不写个几年代码，很难写出像样的代码。

大部分的程序员写的都是CRUD、一些业务逻辑的代码，谁实现不了？对于我来说，如果业务逻辑的代码评审，需要人来讲每一个代码做了什么，这样的代码就是不合格的，合格的代码写出来应该像人说话那么简单有条理，基本上是业务怎么样描述需求，写出来的代码就是怎么样的。编写出非开发人员都能看懂的代码，才是我们追求的目标。不要以写出了一些非常复杂的代码而沾沾自喜。好的代码应该是看起来平淡无奇觉得很简单自然，而不是看得人云里雾里的觉得很高深很有技术含量。

如果你做好了我前面几篇文章的要求，编写简单的函数就容易的多，如果你觉得我之前说的去掉local，去掉用户参数这些没有什么必要是小题大做，那么我觉得你写不出简单的函数。从个人经验来说，函数编写的建议有以下几点：

不要出现和业务无关的参数

参考我之前的帖子，参数校验和国际化规范，函数参数里面不要出现local，messagesource，request，Response这些参数，第一非常干扰阅读，一堆无关的参数把业务代码都遮掩住了，第二导致你的函数不好测试，如你要构建一个request参数来测试，还是有一定难度的。

避免使用**Map，Json**这些复杂对象作为参数和结果

这类参数看着灵活方便，但是灵活的同义词（代价）就是复杂，最终的结果是可变数多bug多质量差。就好比刻板的同义词就是严谨，最终的结果就是高质量。千万不要为了偷懒少几行代码，就到处把map，json传来传去。其实定义一个bean也相当简单，加上lombok之后，代码量也没有几行，但代码可读性就不可同日而语了。做过开发的人应该很容易体会，你如果接手一个项目，到处的输入输出都是map的话，request从头传到尾，看到这样的代码你会哭的，我相信你会马上崩溃很快离职的。

还有人说用bean的话后面加字段改起来麻烦，你用map还不是一样要加一个key，不是更加麻烦吗？说到底就是懒！

如果一个项目的所有代码都如下面这样，我是会崩溃的！

```
/**
 * ！！！错误代码示例
 * 1. 和业务无关的参数locale，messagesource
 * 2. 输入输出都是map，根本不知道输入了什么，返回了什么
 *
```

```
 * @param params
 * @param local
 * @param messageSource
 * @return
 */
public Map<String, Object> addConfig(Map<String, Object> params,
    Locale locale, MessageSource messageSource) {

  Map<String, Object> data = new HashMap<String, Object>();

  try {
    String name = (String) params.get("name");
    String value = (String) params.get("value");

    //示例代码，省略其他代码
  }
  catch (Exception e) {
    logger.error("add config error", e);

    data.put("code", 99);
    data.put("msg", messageSource.getMessage("SYSTEMERROR", null, locale));
  }

  return data;
}
```

有明确的输入输出和方法名

尽量有清晰的输入输出参数，使人一看就知道函数做了啥。举例：

```
public void updateUser(Map<String, Object> params){
  long userId = (Long) params.get("id");
  String nickname = (String) params.get("nickname");

  //更新代码
}
```

上面的函数，看函数定义你只知道更新了用户对象，但你不知道更新了用户的什么信息。建议写成下面这样：

```
public void updateUserNickName(long userId, String nickname){
  //更新代码
}
```

你就算不看方法名，只看参数就能知道这个函数只更新了nickname一个字段。多好啊！这是一种思路，但并不是说每一个方法都要写成这样。

把可能变化的地方封装成函数

编写函数的总体指导思想是抽象和封装，你要把代码的逻辑抽象出来封装成为一个函数，以应对将来可能的变化。以后代码逻辑有变更的时候，单独修改和测试这个函数即可。

这一点相当重要，否则你会觉得怎么需求老变？改代码烦死了。

如何识别可能变的地方，多思考一下就知道了，工作久了就知道了。比如，开发初期，业务说只有管理员才可以删除某个对象，你就应该考虑到后面可能除了管理员，其他角色也可能可以删除，或者说对象的创建者也可以删除，这就是将来潜在的变化，你写代码的时候就要埋下伏笔，把是否能删除做成一个函数。后面需求变更的时候，你就只需要改一个函数。

举例，删除配置项的逻辑，判断一下只有是自己创建的配置项才可以删除，一开始代码是这样的：

```
/**
 * 删除配置项
 */
@Override
public boolean delete(long id) {
  Config config = configs.get(id);

  if(config == null){
    return false;
  }

  // 只有自己创建的可以删除
  if (UserUtil.getUser().equals(config.getCreator())) {
    return configs.remove(id) != null;
  }

  return false;
}
```

这里我会识别一下，是否可以删除这个地方就有可能会变化，很有可能以后管理员就可以删除任何人的，那么这里就抽成一个函数：

```
/**
 * 删除配置项
 */
@Override
public boolean delete(long id) {
  Config config = configs.get(id);
```

```
    if(config == null){
      return false;
    }

    // 判断是否可以删除
    if (canDelete(config)) {
      return configs.remove(id) != null;
    }

    return false;
}

/**
 * 判断逻辑变化可能性大，抽取一个函数
 *
 * @param config
 * @return
 */
private boolean canDelete(Config config) {
    return UserUtil.getUser().equals(config.getCreator());
}
```

后来想了一下，没有权限应该抛出异常，再次修改为：

```
/**
 * 删除配置项
 */
@Override
public boolean delete(long id) {
    Config config = configs.get(id);

    if (config == null) {
      return false;
    }

    // 判断是否可以删除
    check(canDelete(config), "no.permission");

    return configs.remove(id) != null;
}
```

这就是简单的抽象和封装的艺术。看这些代码，参数多么的简单，很容易理解吧。

这一点非常重要，做好了这点，大部分的小的需求变更对程序员的伤害就会降到最低了！毕竟需求变更大部分都是这些小逻辑的变更。

编写能测试的函数

> 程序猿不招妹子们喜爱的根本原因在于追求了错误的目标：更短、更小、更快。

这个非常重要，当然很难实现，很多人做技术之前都觉得代码都会做单元测试，实际上和业务相关的代码单元测试是很难做的。

我觉得要编写能测试的函数主要有以下几点：

第一不要出现乱七八糟的参数，如参数里面有request，response就不好测试，

第二你要把函数写小一点。如果一个功能你service代码只有一个函数，那么你想做单元测试是很难做到的。我的习惯是尽量写小一点，力求每一个函数都可以单独测试（用junit测试或者main函数测试都没有关系）。这样会节约大量的时间，尤其是代码频繁改动的时候。我们应用重启一次需要15分钟以上。新手可以写一个功能可能需要重启10几次，我可能只需要重启几次，节约的时候的很可观的。

第三你要有单独测试每一个函数的习惯。不要一上来就测试整个功能，应该一行一行代码、一个一个函数测试，有了这个习惯，自然就会写出能测试的小函数。所以说，只有喜欢编码的人才能写出好代码。

如我的编码习惯 - 配置规范这篇文章了，我的配置相关代码，都是可以单独测试的，所以配置项的改动不需要测试业务功能，应用都不需要重启。

配置规范

工作中少不了要制定各种各样的配置文件，这里和大家分享一下工作中我是如何制定配置文件的，这是个人习惯，结合强大的spring，效果很不错。

需求

如我们现在有一个这样的配置需求，顶层是Server，有port和shutdown2个属性，包含一个service集合，service对象有name一个属性，并包含一个connector集合，connector对象有port和protocol2个属性。

我一上来不会去考虑是用xml还是json还是数据库配置，我会第一步写好对应的配置bean。如上面的需求，就写3个bean。bean和bean之间的包含关系要体现出来。（使用了lombok）

```
@Data
public class ServerCfg {
  private int port = 8005;
  private String shutDown = "SHUTDOWN";
  private List<ServiceCfg> services;
}

@Data
public class ServiceCfg {
  private String name;
  private List<ConnectorCfg> connectors;
}

@Data
public class ConnectorCfg {
  private int port = 8080;
  private String protocol = "HTTP/1.1";
}
```

然后找一个地方先用代码产生这个bean：

```
@Configuration
public class Configs {

  @Bean
  public ServerCfg createTestBean() {
    ServerCfg server = new ServerCfg();

    //
    List<ServiceCfg> services = new ArrayList<ServiceCfg>();
```

```
    server.setServices(services);

    //
    ServiceCfg service = new ServiceCfg();
    services.add(service);

    service.setName("Kitty");

    //
    List<ConnectorCfg> connectors = new ArrayList<ConnectorCfg>();
    service.setConnectors(connectors);

    //
    ConnectorCfg connectorhttp11 = new ConnectorCfg();

    connectorhttp11.setPort(8088);
    connectorhttp11.setProtocol("HTTP/1.1");

    connectors.add(connectorhttp11);

    //
    ConnectorCfg connectorAJP = new ConnectorCfg();

    connectorAJP.setPort(8089);
    connectorAJP.setProtocol("AJP");

    connectors.add(connectorAJP);

    return server;
  }
}
```

然后先测试，看看是否ok。为了演示，我就直接在controller里面调用一下

```
@Autowired
ServerCfg cfg;

@GetMapping(value = "/configTest")
@ResponseBody
public ResultBean<ServerCfg> configTest() {
  return new ResultBean<ServerCfg>(cfg);
}
```

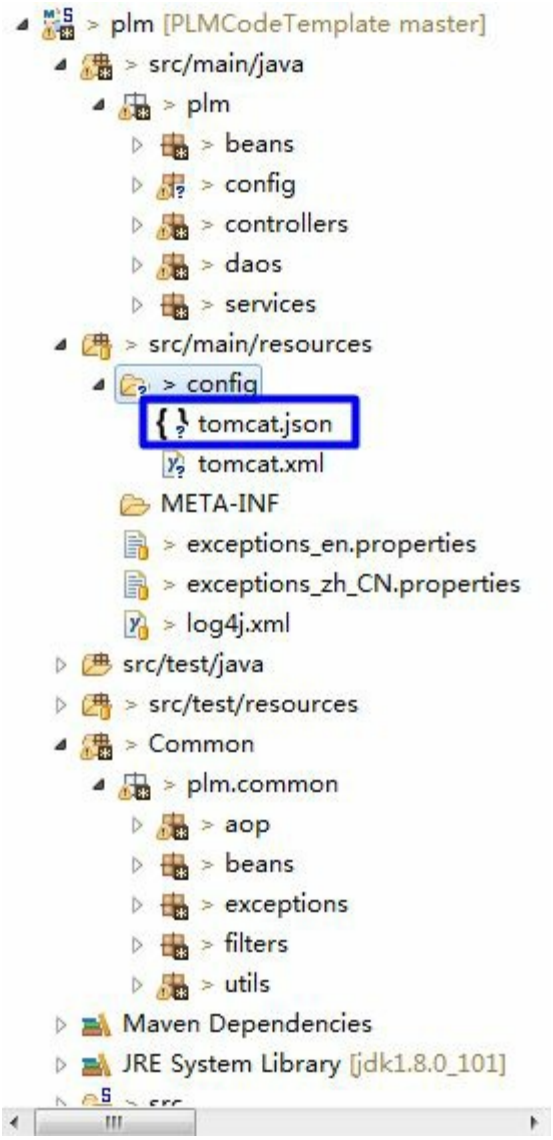
测试一下，工作正常。

然后进行业务代码编写，等到所有功能测试完毕，就是【开发后期】，再来定义配置文件。中途当然少不了修改格式，字段等各种修改，对于我们来说只是修改bean定义，so easy。

都ok了，再决定使用哪种配置文件。如果是json，我们这样：

JSON格式

把上面接口调用的json复制下来，报存到配置文件。



json内容

```
{
  "port": 8005,
  "shutDown": "SHUTDOWN",
  "services": [
    {
      "name": "Kitty",
      "connectors": [
        {
          "port": 8088,
          "protocol": "HTTP/1.1",
          "executor": null
        },
        {
          "port": 8089,
          "protocol": "AJP",
          "executor": null
```

```
        }
      ]
    }
  ]
}
```

然后修改config的bean生成的代码为：

```
import com.fasterxml.jackson.databind.ObjectMapper;

@Configuration
public class Configs {
  @Value("classpath:config/tomcat.json")
  File serverConfigJson;

  @Bean
  public ServerCfg readServerConfig() throws IOException {
    return new ObjectMapper().readValue(serverConfigJson, ServerCfg.class);
  }
}
```

代码太简洁了，有没有？！

XML格式

如果使用XML，麻烦一点，我这里使用XStream序列化和反序列化xml。

首先在bean上增加XStream相关注解

```
@Data
@XStreamAlias("Server")
public class ServerCfg {

  @XStreamAsAttribute
  private int port = 8005;

  @XStreamAsAttribute
  private String shutDown = "SHUTDOWN";

  private List<ServiceCfg> services;

}

@Data
@XStreamAlias("Service")
public class ServiceCfg {
```

```
  @XStreamAsAttribute
  private String name;

  private List<ConnectorCfg> connectors;
}

@Data
@XStreamAlias("Connector")
public class ConnectorCfg {
  @XStreamAsAttribute
  private int port = 8080;

  @XStreamAsAttribute
  private String protocol = "HTTP/1.1";
}
```

然后修改产品文件的bean代码如下：

```
@Configuration
public class Configs {
  @Value("classpath:config/tomcat.xml")
  File serverConfigXML;

  @Bean
  public ServerCfg readServerConfig() throws IOException {
    return XMLConfig.toBean(serverConfigXML, ServerCfg.class);
  }
}
```

XMLConfig工具类相关代码：

```
public class XMLConfig {

  public static String toXML(Object obj) {
    XStream xstream = new XStream();

    xstream.autodetectAnnotations(true);
    // xstream.processAnnotations(Server.class);

    return xstream.toXML(obj);
  }

  public static <T> T toBean(String xml, Class<T> cls) {
    XStream xstream = new XStream();

    xstream.processAnnotations(cls);
```

```
    T obj = (T) xstream.fromXML(xml);

    return obj;
  }

  public static <T> T toBean(File file, Class<T> cls) {
    XStream xstream = new XStream();

    xstream.processAnnotations(cls);
    T obj = (T) xstream.fromXML(file);

    return obj;
  }

}
```

XStream库需要增加以下依赖：

```
<dependency>
  <groupId>com.thoughtworks.xstream</groupId>
  <artifactId>xstream</artifactId>
  <version>1.4.10</version>
</dependency>
```

所以个人爱好，格式推荐json格式配置。

配置文件编码禁忌

- 读取配置的代码和业务代码耦合在一起

大忌！千万千万不要！如下，业务代码里面出现了json的配置代码。

```
public void someServiceCode() {
  // 使用json配置，这里读取到了配置文件，返回的是json格式
  JSONObject config = readJsonConfig();

  // 如果某个配置了
  if(config.getBoolean("somekey")){
    // dosomething
  }
  else{

  }
}
```

- 开发初期就定配置文件

毫无意义，还导致频繁改动！先定义bean，改bean简单多了。我的习惯是转测试前一天才生成配置文件。

- 手工编写配置文件

应该先写完代码，根据代码生成配置序列化成对应的格式，而不是自己编写配置文件然后用代码读出来。不要做反了。

重要思想

最主要的思想是，不要直接和配置文件发生关系，一定要有第三者（这里是配置的bean）。你可以说是中间件，中介都行。 否则，一开始说用xml配置，后面说用json配置，再后面说配置放数据库？这算不算需求变更？你们说算不算？算吗？不算吗？何必这么认真呢？只是1,2行代码的问题，这里使用xml还是json，代码修改量是2行。而且改了测试的话，写个main函数或者junit测试即可，不需要测试业务，工程都不用起，你自己算算节约多少时间。

另外，代码里面是使用spring的习惯，没有spring也是一样的，或者配置的bean你不用spring注入，而用工具类获取也是一样，区别不大。

如何应对需求变更

我之前的文章 程序员你为什么这么累？ 中，我个人观点是加班原因是编码质量占了大部分因素，但是不少同学都不认为是代码质量导致的加班，都认为是不断的需求改动导致的加班。这位同学，说的好像别人的需求就不会变动似的！谁的需求不改动啊？不改动的能叫需求吗？哈哈。

先看个程序员的段子娱乐一下

```
客户被绑，蒙眼，惊问：“想干什么?”
对方不语，鞭笞之，客户求饶：“别打，要钱?”
又一鞭，“十万够不?”
又一鞭，“一百万?”
又一鞭。客户崩溃：“你们TMD到底要啥?”
“要什么?我帮你做项目，写代码的时候也很想知道你TMD到底想要啥!”
```

有没有可能存在明确的、不再改动的需求呢？其实很难的。以前我们公司是瀑布开发模式，需求阶段时间较长，需要输出完整的需求规范，还要评审几次然后才进入开发，这个时候，需求变更就比较少，但还是有；后来公司赶时髦改成了敏捷开发模式，文档大量简化，于是需求没有考虑清楚就开始开发，需求是边开发边变动。敏捷开发模式间接变成了加班开发模式。

关于需求变动，不同的角色定义很不一样。BA觉得这个改动很正常，开发人员觉得就是个需求变更，两边各执一词，这种矛盾长期存在。

我列举几种场景，大家觉得算不算需求变更？

1. 删除对象功能

一开始只能创建者删除，后面变更为管理员也可以删除，再后面变更了某个角色也可以删除。

1. 配置功能

一开始使用xml配置，后面修改为json格式，又或者修改为使用数据库配置。

1. 导出功能

一开始导出为excel格式，后面变更为导出json格式或者pdf格式。或者一开始导出20个字段，后面变更为导出30个字段。

这些当然都是变更了，但这些真的就是我们加班加点的原因吗？！我们就没有办法只能任人宰割吗？！而我的观点刚好是，正是因为需求变更不可避免，所以我们才更应该把代码写简单，以对付各种各样的需求变化。有以下几点心得建议：

把代码写到最简单

最起码的要求，我之前一系列的文章说的就是这个。重要程度不需要再讲了。改1行简单代码和改10行复杂代码，工作量能一样吗？！测试一个20行的函数和测试一个2行的函数工作量能一样吗？！

好比一个房子，你打扫的干干净净收拾得井井有条，将来房子里面的东西搬来搬去都比较简单；但如果你的房子垃圾堆一样，走进去都难（代码无法看），就更加不用说把东西搬动了（改代码）。

把可能变化的封装成函数

请阅读：函数编写建议。很重要的习惯，多思考多抽象和封装，小变更将无法伤害到你。主动思考，主动思考将来可能的各种场景。其实这个不难，你只要有这个意识就成功了一大步！

先做确定的需求

多个功能中先做不会变的功能，一个功能中先做不会变的部分，兵法中叫攻其必救之地。你要知道哪些需求是所有人都明白看上去就很合理的需求，就先开始做，你觉得有争议的需求你可以放后面一点。同样，一个功能中你要知道哪些会变的，哪些是不会变的，不变的先做。

> 需求实现先后顺序应该难的/确定的先做。先做难的是需要把周期拉长，更多的时间设计；先做确定的是为了避免频繁的改动。

举例：每个系统都有导出功能，我们实现功能之前，先要考虑哪些功能是确定的，哪些功能是很可能变化的？简单分析之后可以知道，从数据库库查询出来然后处理包装数据这是肯定要做的而且不会变的，这个应该先做；而导出为什么格式（xls还是pdf），导出的具体完整字段，字段的格式如何展示这些是会变的，这些你开始甚至都不需要仔细看需求，等要做的时候在确认可能需求都有不同的变化。你完全可以边做前面确定的导出功能边确认其他的细节，确认需求的时间越多，需求就越清晰，变更的概率就越小。

多个功能中，我的习惯是先做最难的功能，最少要开始设计和思考，拉长功能开发周期。有些同学喜欢先做简单的，导致难的问题开发周期不够，后面加班加点也解决不了。加班其实是解决不了太多问题的。

> 拖延症的一个好处就是，很多需求拖着拖着就不用做了，因为提出人发现了这个需求的不合理。所以先做合理确定的需求。

解耦！解耦！

个人认为，解耦是编程里面重要的思想，解耦的关键在于：多引入“第三者”，不要直接发生关系。spring的IoC最重要的价值不就是解耦吗？spring的容器不就是“第三者”吗？就像mvc一样，数据和视图要彻底的分离，否则业务代码里面有视图代码改起来是很痛苦的。

我上面的 `配置规范` 里面的举例，bean的定义就是第三者，就是为了解耦。如导出功能里面，也要有中介。不要把查询数据，处理数据和导出数据都在一个函数一个循环里面做了。否则导出格式由xls改成pdf的时候，你相当于重写做了一遍功能。jms这些基于消息的都是解耦的思想，架构设计

上要多用这些松耦合的设计。

数据结构上要考虑扩展

由于是牵涉到表设计的时候，大家都知道改表结构很痛苦。很多时候，由于时间关系，一开始只做简单的功能，后面会慢慢丰富功能。这虽然不是变更，但是如果你一开始的时候不设计好，很可能后面版本需要大改动，数据库表都要推倒重来，比全新做还痛苦，相信大家会有体会。所以，作为开发组长，做任何一个功能都要想到将来的发展，功能现在可以不做，但必须对将来的变化做到了然于胸。

我举几个例子。

- 下载功能

工作量问题当前版本只需要显示总下载量。你要考虑将来会不会要列出所有的下载过的用户？如果不需要，可能用一个字段记录总数就可以；如果需要，那么就要用新表，就算现在做起来麻烦一点也不要后面来推翻数据库表设计。

- 关系表相关的功能

牵涉到link的，现在是1对1，要考虑将来有没有可能1对n或者n对n。1对1用个外键就可以了，否则一开始就单独用一张link表。

- 系统集成

现在只对接一个系统，要考虑将来会不会相同的业务对接多个系统？如果会，你应该直接上jms这种（虽然工作量加大了），不上jms这种的话，也要设计成被动接受的集成方式，那么在增加新系统你都不需要怎么样改。（如果你主动请求的交互方式，多一个系统你就要多一个配置，多测试一遍，如果设计成被动接受的，就不需要什么配置和测试了。而很多时候，2个系统集成设计成主动被动都可以实现需求）

总结

其实，我上面说的这些，概括起来，就是要主动思考，多走一步，不要被动接受看到的需求，要对需求的将来变化做好心中有数。当然，你可以说这些变更都是小变，大变怎么办？大变还不给你加工作量，你就走人不干了吧，哪里有这么无良的老板！

> 每一个开发人员都应该思考：需求变动真的是我加班的最重要原因吗？我的代码是否写得足够好？需求变更里面，我能控制是什么，我不能控制的是什么？我应该做好什么的准备来拥抱需求的变更？

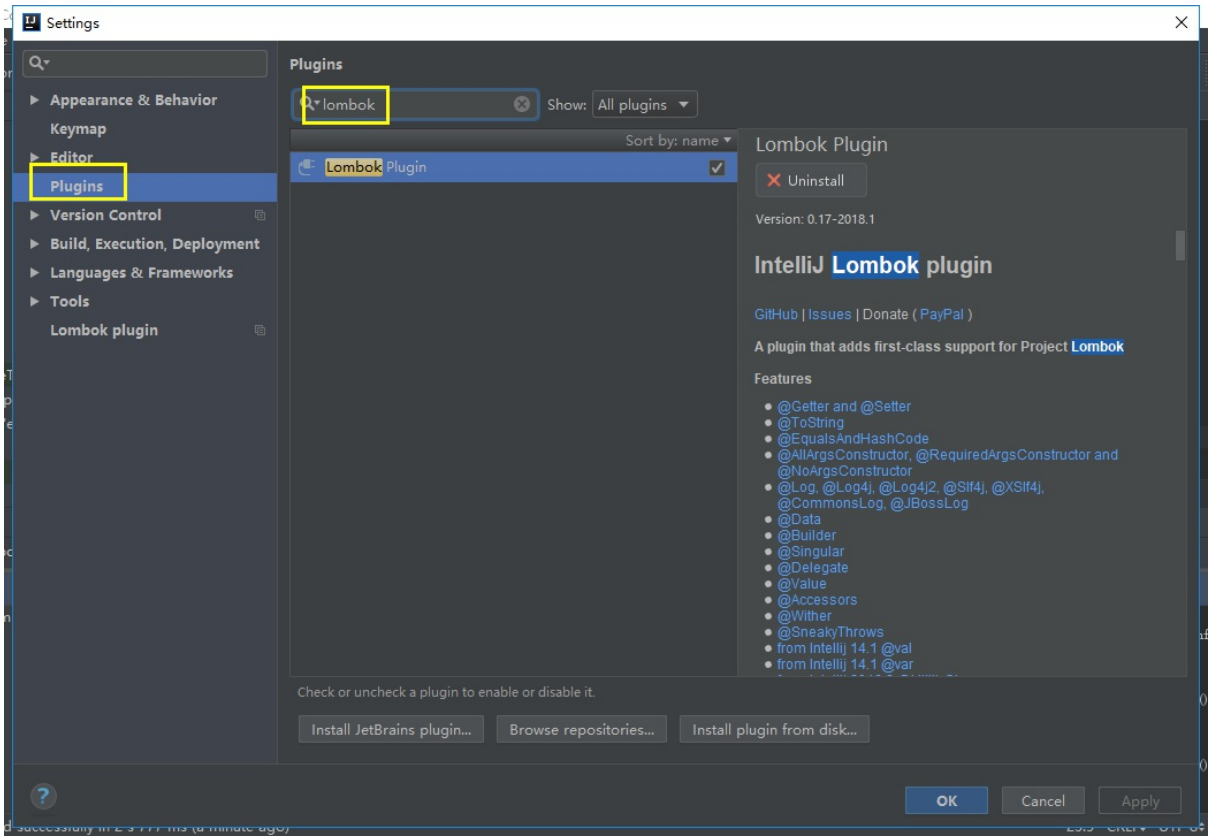
工程使用说明

jdk

jdk6+。

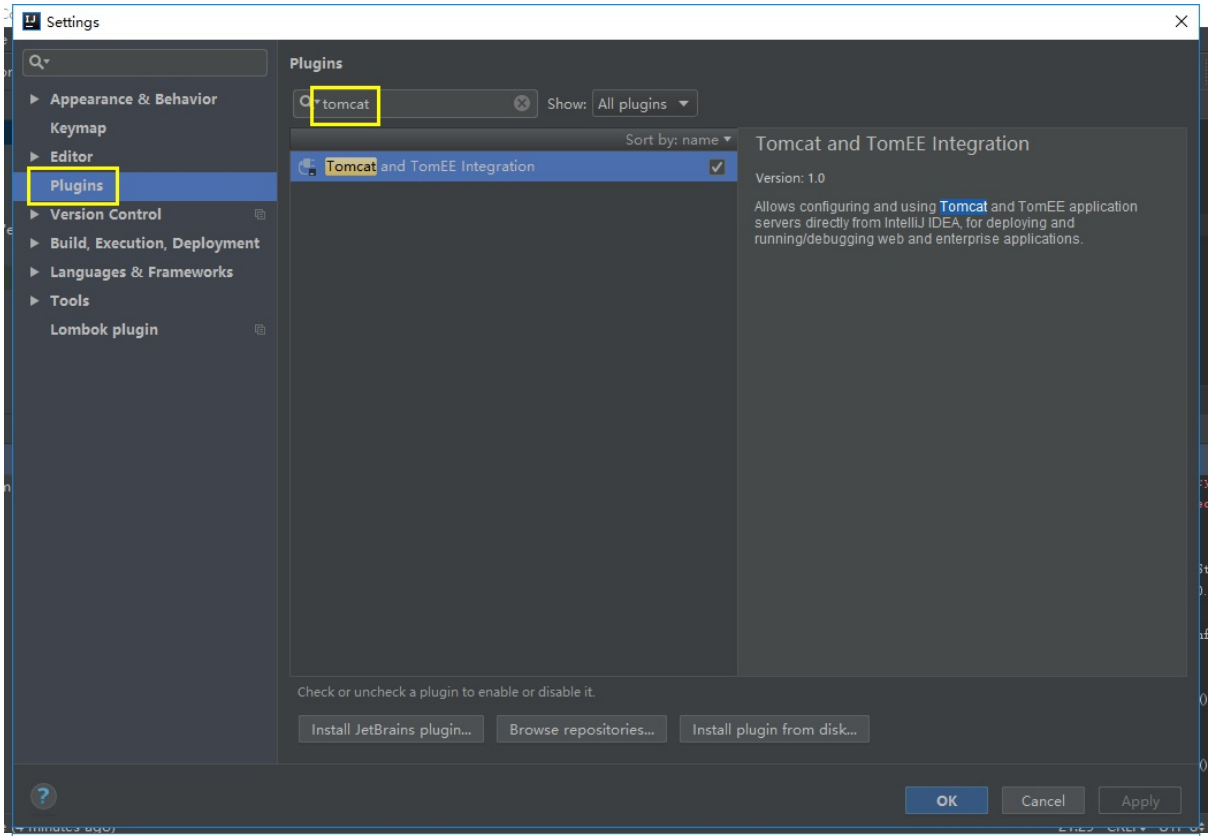
idea

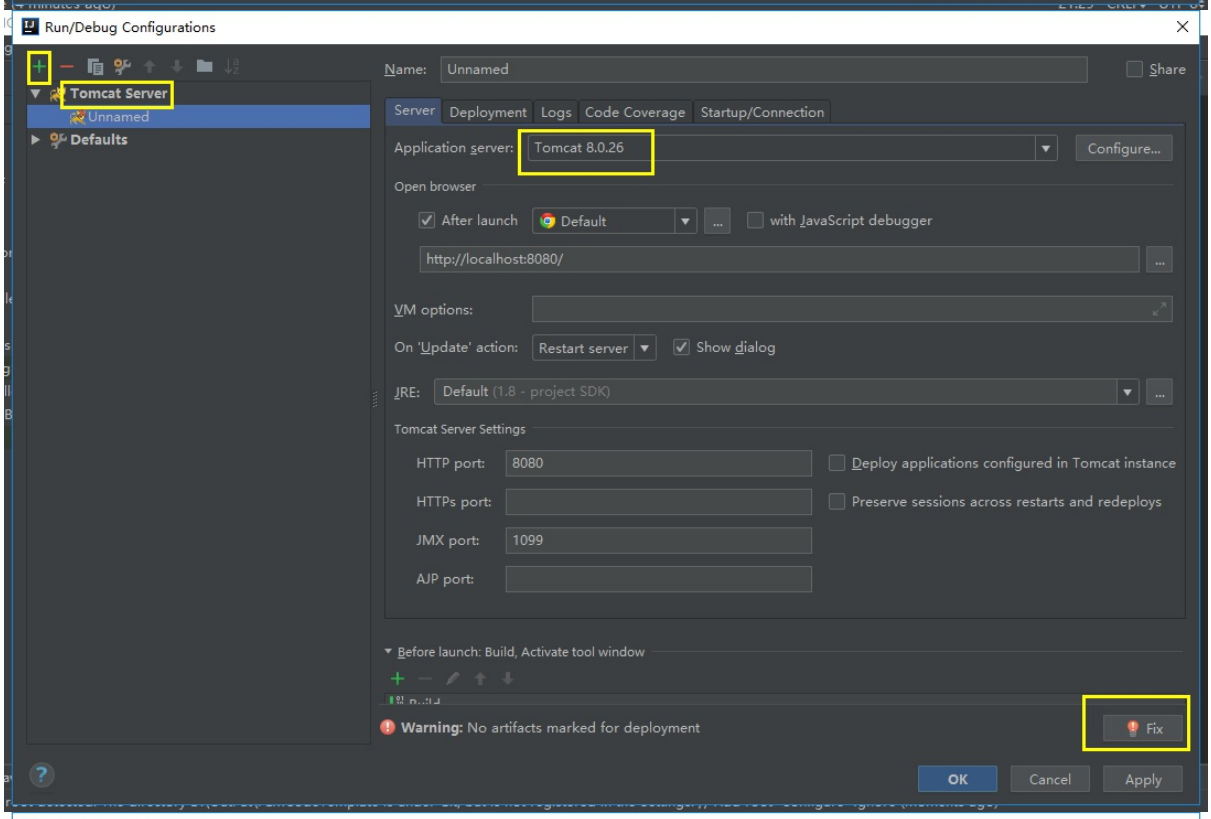
使用了lombok，需要在IDE里面先安装插件。idea中在 `plugins` 里面安装 `lombok` 插件重启即可。

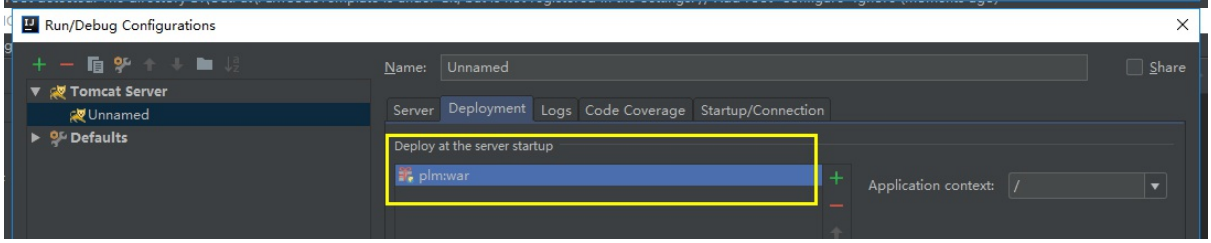


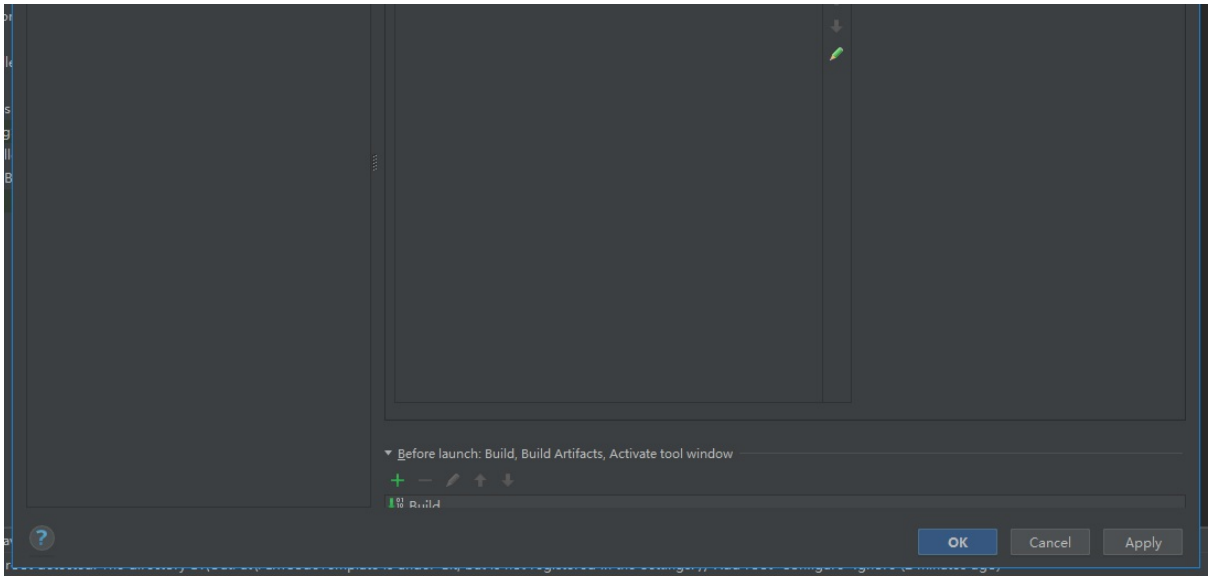
在 Idea 里面选择 `source` 目录导入 `Maven` 工程，然后在Tomcat里面运行工程即可。

idea 中需要先安装tomcat插件。


Settings
Plugins
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Languages & Frameworks
Tools
Lombok plugin
tomcat
Show: All plugins
Sort by: name
Tomcat and TomEE Integration
Tomcat and TomEE Integration
Version: 1.0
Allows configuring and using Tomcat and TomEE application servers directly from IntelliJ IDEA, for deploying and running/debugging web and enterprise applications.
Check or uncheck a plugin to enable or disable it.
Install JetBrains plugin...
Browse repositories...
Install plugin from disk...
OK
Cancel
Apply


Run/Debug Configurations
Tomcat Server
Unnamed
Defaults
Name: Unnamed
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Application server: Tomcat 8.0.26
Configure...
Open browser
After launch
Default
with JavaScript debugger
http://localhost:8080/
VM options:
On 'Update' action: Restart server
Show dialog
JRE: Default (1.8 - project SDK)
Tomcat Server Settings
HTTP port: 8080
Deploy applications configured in Tomcat instance
HTTPs port:
Preserve sessions across restarts and redeploys
JMX port: 1099
AJP port:
Before launch: Build, Activate tool window
Warning: No artifacts marked for deployment
Fix
OK
Cancel
Apply


Run/Debug Configurations
Tomcat Server
Unnamed
Defaults
Name: Unnamed
Share
Server
Deployment
Logs
Code Coverage
Startup/Connection
Deploy at the server startup
plm:war
Application context: /



启动项目，访问地址 `http://localhost:8080/+[应用名（可为空）]` 即可。

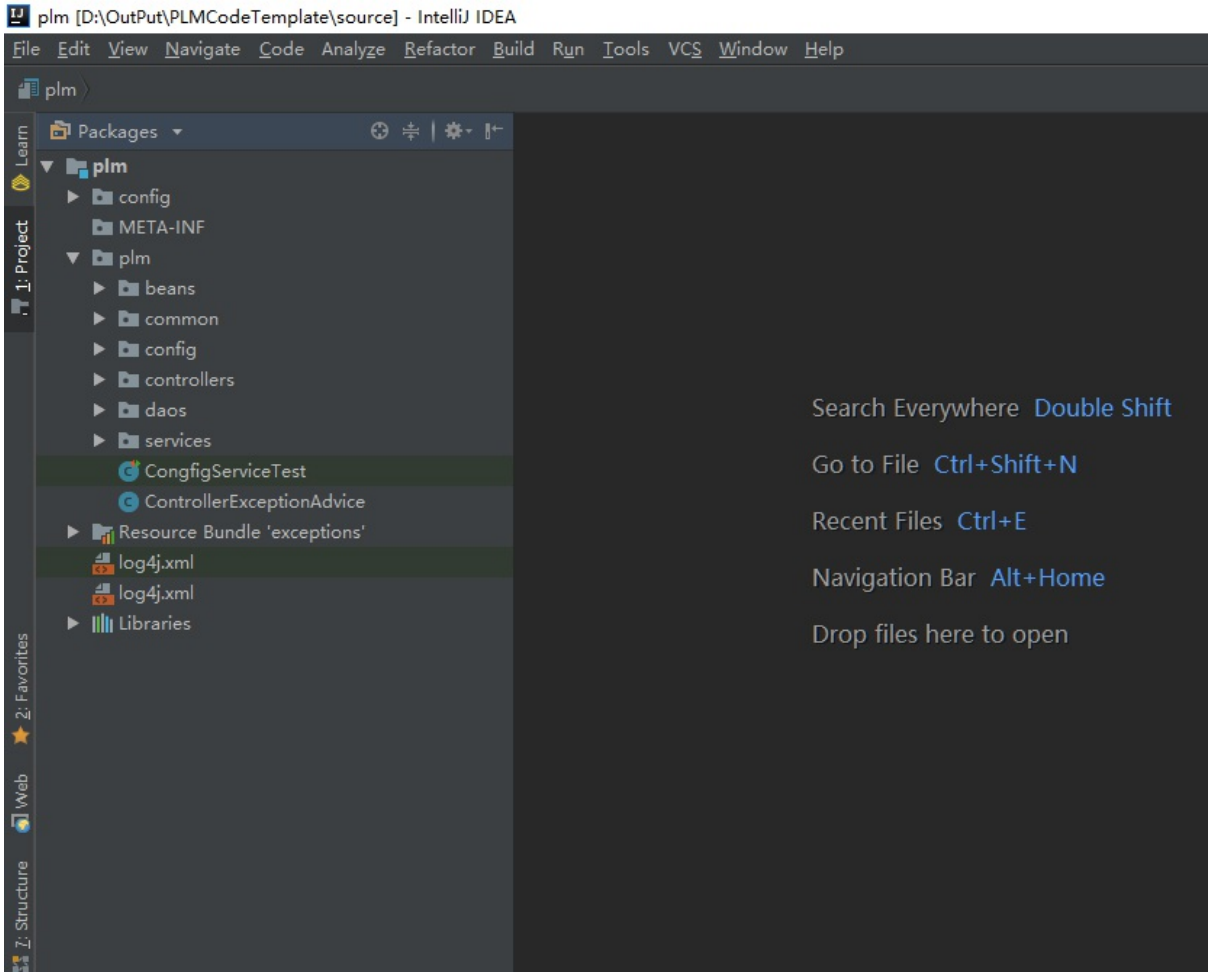
eclipse / sts

使用了lombok，需要在IDE里面先安装插件。

然后需要把 `source\src\common` 导入为源代码目录即可。（Eclipse里面在该目录选择右键 "Build Path" -> "Use as Source Folder"）

启动项目，访问地址 `http://localhost:8080/plm` 即可。

工程目录结构


plm [D:\OutPut\PLMCodeTemplate\source] - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
plm
Packages
plm
config
META-INF
plm
beans
common
config
controllers
daos
services
CongfigServiceTest
ControllerExceptionAdvice
Resource Bundle 'exceptions'
log4j.xml
log4j.xml
Libraries
Search Everywhere Double Shift
Go to File Ctrl+Shift+N
Recent Files Ctrl+E
Navigation Bar Alt+Home
Drop files here to open
1: Project
2: Favorites
7: Structure
Learn
Web

工程界面


晓风轻
GitHub
老的首页
读取json配置例子
相关链接
Search
Submit
登陆
语言
晓风轻代码模板
编写简单可读的代码！欢迎交流，QQ群： 607679993
Learn more
新增配置项
配置项名称
晓风轻技术小站
配置项取值
99
配置项描述
描述
新增配置项

GITHUB地址

工程地址 https://github.com/xwjie/PLMCodeTemplate

相关链接

- 知乎
- 慕课网