**序言**

DIP是集成业界流行和成熟的技术框架和良好用户体验的平台应用快速开发框架。 它具有以下特点：

* 轻量 ： 技术选型及架构上避免选择重型的框架和架构
* 前后端分离 ： 前端完成UI相关的动作， 服务端只负责数据处理
* Spring集成 ：集成较新的Spring Boot框架
* Rest风格 ： 标准的restful风格架构
* 约定俗成 ： 约定优先， 规则优先（ 接口定义， 命名等等） 。 统一代码风格。
* 移动互联 ： 兼容PC、 移动端。
* 支持分布式部署 ： 负载均衡， 服务器节点水平伸缩扩容。
* 面向接口 ： 规范约束， 分离设计与实现， 解耦合。
* 高内聚低耦合 ： 易维护、 易扩展。

**背景**

随着传统互联网， 移动互联网， 互联网+等互联网形态演进， 新技术层出不穷， 都为一个目标， 为适应瞬息万变的市场。 传统的企业应用业务越来越庞大和复杂， 而交付周期却越来越快。 国内各大企业都在打造适应自己业务需求的应用框架和平台。 构建成熟合适的平台， 有利于整个系统的快速开发和良好实施， 也有利于企业内部技术体系的构建和资源的共享。 不是简单的技术框架堆砌而成的脚手架， 而是提供高效的业务开发平台。 把应用系统从纷繁复杂的框架选型， 基础功能开发， 系统集成等重复劳动中解放出来， 从而聚焦业务本身的功能。

**简介**

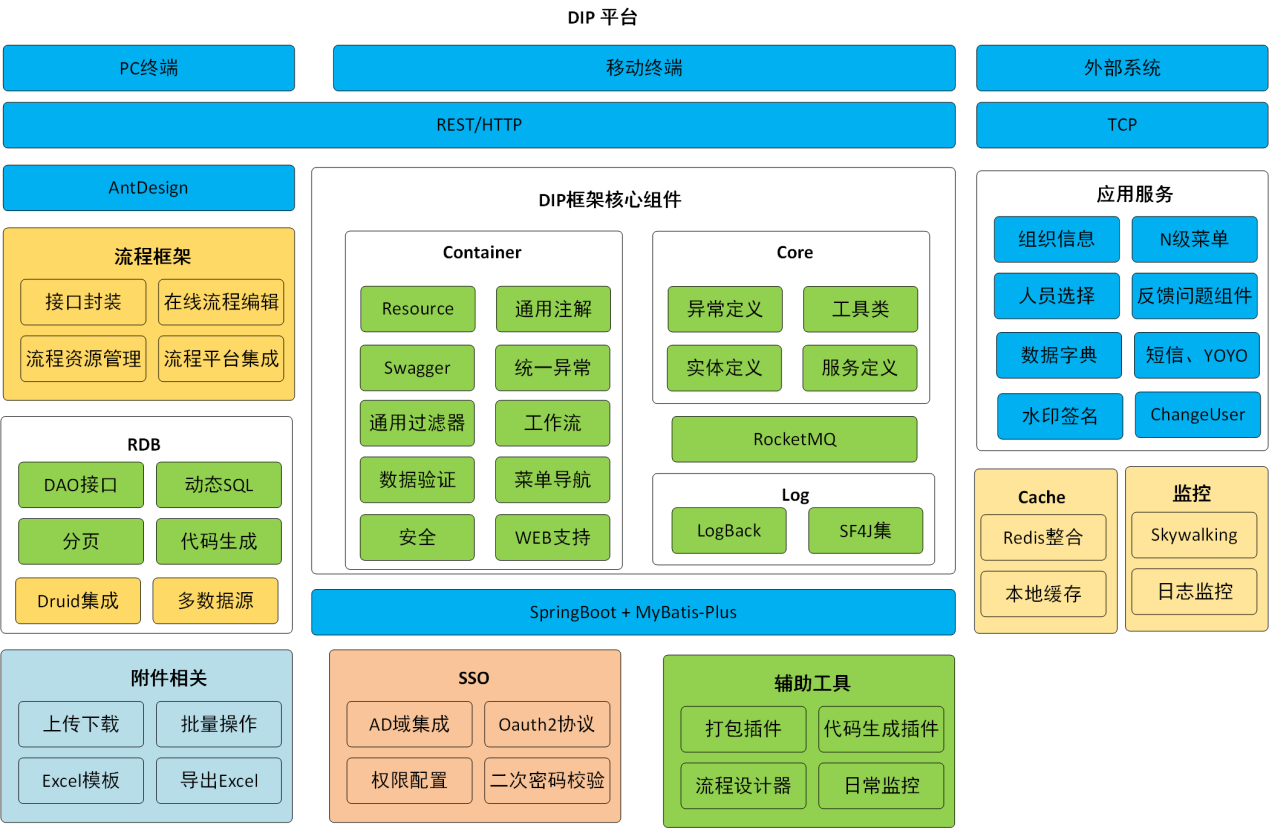
整个架构体系以轻量为指导原则， 大道至简的架构才能更顺应技术变革和满足不同行业应用的实施。 基于底层云计算平台， 打造支持分布式部署的， 手机， pad， pc接入的， 良好用户体验的应用快熟开发框架。 集成业界流行和成熟的技术框架， 并通过应用系统使用， 反馈的情况不断完善应用框架的通用功能， 满足业务系统快熟构建的目标。

整个架构体系以轻量为指导原则， 大道至简的架构才能更顺应技术变革和满足不同行业应用的实施。

* 开发简单
* 技术栈灵活
* 服务独立无依赖
* 独立按需扩展
* 可用性高

**基础框架**

1. **总体架构**

****

总体架构图

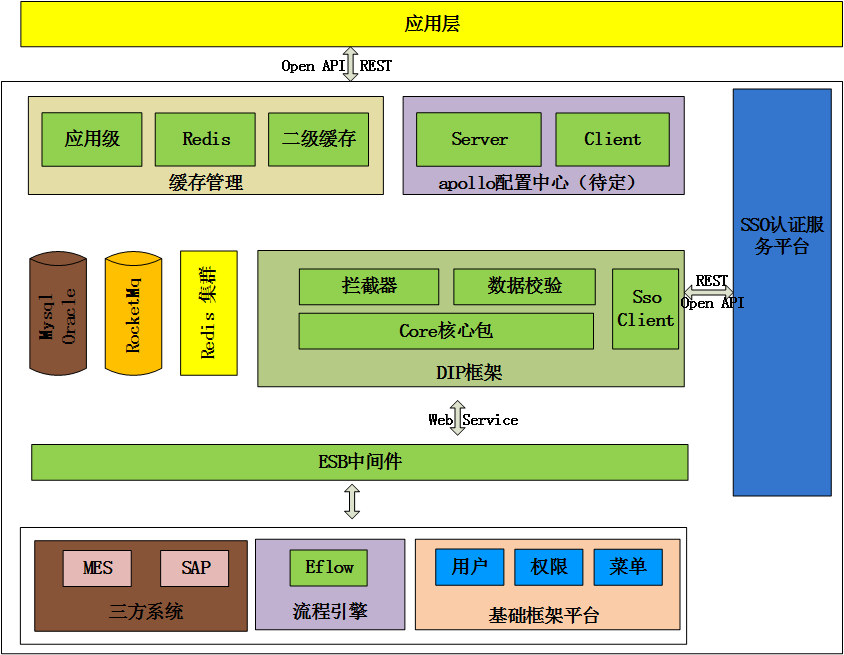
本平台全栈采用开源组件，核心基于SpringBoot+Mybatis-Plus+Vue 打造前后端分离的系统架构。通过提供代码生成插件及核心基类便于业务系统快速进行业务的迭代开发。已有的流程、SSO认证、Redis、RocketMQ等系统或组件，通过平台提供的工具类可快速进行集成开发。

通过抽象封装业务系统中常用的组件，可以进一步体现框架的易用及便捷性。我们提供的组件包含组织信息、N级菜单、人员选择、反馈问题、数据字典、短信YOYO、水印签名、ChangeUser等。同时对于一些通用功能如上传下载、Excel导出等也提供了相应的接口调用。

相关技术栈如下：

* 后端：Spring Boot、Mybatis、Mybatis-Plus、Druid、Dynamic-datasource、FastJson、LogBack、Redis、Dom4j、Axis2、Cas、Jwt、Swagger、Hutool、AspectJ、Poi、RocketMQ、SkyWalking等
* 前端：Html5、Js、Css、Es2015、Vue、Vue Router、Ant-Design、Axios、Lodash、Moment、MD5等

1. **系统集成架构**



考虑到公司内使用的三方系统及应用较多，平台框架开发过程中同步需要考虑与三方系统的集成内容。主要涉及的系统可分为如下三部分：

* ESB相关。子系统注册在ESB中间件中，框架通过提供webService接口的方式，与各个子系统进行交互。涉及的子系统主要包含SAP、MES、流程引擎、基础框架平台等。
* 单点登录相关。框架需要提供基于AD域或用户名、密码的方式进行SSO实现。利用CAS框架达到各个基于框架开发的子系统，只需登录一次，访问各个子系统免登录的效果。
* 中间件相关。框架需要提供基于redis、rocketmq等中间件的公共方法调用。

**应用开发**

本章包含如何快速开发DIP平台下代码, 了解业务开发相关的基础技术栈, 以及平台提供的开发工具包

**环境篇**

开发环境搭建

**Java环境**

以下内容为建议安装， 如本机环境同建议环境不同， 可自行调整

1. **Java安装**

1. 要安装 JDK(JDK8+)， 请转至 http://java.sun.com/javase/downloads/index.jsp。

安装JDK 选择安装目录 安装过程中会出现两次 安装提示 。 第一次是安装 jdk ， 第二次是安装 jre 。 建议两个都安装在同一个java文件夹中的不同文件夹中,推荐路径 d:\Java\jdk8 下 （ 注意安装路径不要有空格） 。

一旦安装了 JDK， 您即可设置 JAVA\_HOME。

1. 要设置 JDK， 请右键单击“我的电脑”， 然后选择“属性”。

2. 在“高级”选项卡上， 选择“环境变量”， 然后编辑 JAVA\_HOME 以指向 JDK 所在的位置（ 例如： d:\Java\jdk8） 。

3. 添加Path路径， 系统变量→寻找 Path 变量→编辑

在变量值最后输入 d:\Java\jdk8\bin;

（ 注意原来Path的变量值末尾有没有;号， 如果没有先输入;号再输入上面的代码）

1. **Maven安装**

1. 要安装Maven, 请转至http://maven.apache.org/download.cgi

2. 加入maven环境到Path， 系统变量→寻找 Path 变量→编辑

在变量值最后输入 d:\maven\bin;

3. 配置maven仓库地址

<localRepository>d:/m2/repository</localRepository>

...

<mirrors>

...

<mirror>

<id>alimaven</id>

<name>aliyun maven</name>

<url>http://maven.aliyun.com/nexus/content/groups/public/</url>

<mirrorOf>central</mirrorOf>

</mirror>

</mirrors>

1. **Eclipse安装**

1. 要安装Eclipse， 请转至http://www.eclipse.org/downloads/

2. 配置文字编码UTF-8(Window->Preferences->General->Workspace)

3. 导入代码格式化 (Window->Preferences->Code Style->Formatter) eclipse-java-smart-style.xml

4. Maven配置

1. **Git安装**

DIP的源码、 配置文件、 demo、 文档等都保存在git环境中， 应用开发首先必须要安装git环境

**1.** 要安装 **Git**

请转至https://git-scm.com/download/win。

**2.** 要安装 **TortoiseGit**

请转至 https://tortoisegit.org/download/。

1. **工具**

除标准开发工具外的其他辅助开发工具, 可按需安装

**1. MySQL**客户端

**2. Redis**客户端 **RedisDesktopManager**

**3. SSH**客户端 **MobaXterm**

**4. NodeJS**

**5.** 代码比较工具

**6.** 文本编辑工具 **notepad++**

**7.** 浏览器 **Chrome, Firefox**

**基础篇**

应用基础开发涵盖的技术要点

**Annotation**注解

java 注解， 从名字上看是注释， 解释， 但功能却不仅仅是注释那么简单。

注解（ Annotation） 为我们在代码中添加信息提供了一种形式化的方法。 通过注解机制， 我们可以更好的实现一些特性。 比如： JSON序列化增强， 及配置信息等。

@GET,@POST,@Component,@Service,@Autowired,@Mapper,@Produces,@Consumes 等用法可参考官方文档接下来， 我们会详细介绍框架提供的注解及作用和用法。

**1. @CrudConfig** （基础增删改查注释）

该注解只能作用于类class,在需要进行控制的资源类上添加该注释 。 该注解控制该controller的基础增删改查方法的特定属性，如修改是否更新所有字段、查询根据什么属性进行排序等。 属性如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 属性类型 | 属性说明 | 默认值 |
| updateAllColumn | Boolean | 是否更新所有字段 | False |
| properties | String | 查询字段 | “” |
| simpleProperties | String | 简单的查询字段 | “” |
| sortField | String | 默认排序字段 | id |
| isAsc | Boolean | 默认是否升序 | true |
| updateIgnoreProperties | String | 更新排除字段 | “” |

使用方式：

@CrudConfig(updateAllColumn = **true**, updateIgnoreProperties = {"pwd","salt"})

**public** **class** UserController **extends** BaseCrudController<User, IUserService, Long> {

....

}

**2. @Log** （日志）

该注解可作用于类class或方法上，用于该方法请求时需要记录操作日志时使用 。 在方法上加上该注解后，方法的请求会自动写入log日志。写入操作为异步操作，成功与否均不影响业务方法的正常执行。且写入方法可自定义，分为写入文件或数据库。 属性如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 属性类型 | 属性说明 | 默认值 |
| value | String | 日志描述 | “” |
| outType | Enum | 写入文件或数据库，分为FILE、DB | FILE |

使用方式：

@Log(value="图书管理")

**public** **class** BookServiceImpl **extends** CrudServiceImpl<BookMapper, Book, Long> **implements** BookService {

@Log(outType=LogOutType.***FILE***, value="测试多数据库查询")

**public** CommonResult list4Salve(){

....

}

}

**3. @Logined** （校验登录态）

该注解只能作用于方法上，用于校验该方法请求时是否必须登录态 。 在方法上加上该注解后，方法的请求必须是登录态，否则返回未登录异常。 属性如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性名 | 属性类型 | 属性说明 | 默认值 |
| notEffectSelf | Boolean | 是否对自己生效（false只对基类生效） | False |

使用方式：

**public** **class** BookServiceImpl **extends** CrudServiceImpl<BookMapper, Book, Long> **implements** BookService {

@Logined

**public** CommonResult list4Salve(){

....

}

}

**Filter 过滤器**

在开发 Web 项目的时候，经常需要过滤器来处理一些请求，包括字符集转换什么的，记录请求日志什么的等等。在Springboot中，可以通过实现Filter接口的方法实现自定义过滤器。框架提供跨域及xss攻击两种过滤器实现。

1. **CrossFilter**（跨域过滤器）

当一个资源从与该资源本身所在的服务器不同的域或端口请求一个资源时，资源会发起一个跨域 HTTP 请求。出于安全原因，浏览器限制从脚本内发起的跨源HTTP请求。

CrossFilter过滤器用于解决如上问题，实现允许跨域请求。（尤其在前后端分离的项目开发过程中，跨域请求非常常见）

**public** **class** CrossFilter **implements** Filter {

@Override

**public** **void** init(FilterConfig filterConfig) **throws** ServletException {

}

@Override

**public** **void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {

HttpServletResponse response = (HttpServletResponse) servletResponse;

HttpServletRequest reqs = (HttpServletRequest) servletRequest;

response.setHeader("Access-Control-Allow-Origin",reqs.getHeader("Origin"));

response.setHeader("Access-Control-Allow-Credentials", "true");

response.setHeader("Access-Control-Allow-Methods", "\*");

response.setHeader("Access-Control-Max-Age", "3600");

response.setHeader("Access-Control-Allow-Headers", "X-Requested-With,Content-Type,Authorization");

filterChain.doFilter(servletRequest, servletResponse);

}

@Override

**public** **void** destroy() {

}

}

1. XssFilter（Xss攻击过滤器）

XSS : 跨站脚本攻击(Cross Site Scripting)，为不和层叠样式表(Cascading Style Sheets, CSS)的缩写混淆，故将跨站脚本攻击缩写为XSS。恶意攻击者往Web页面里插入恶意html代码，当用户浏览该页之时，嵌入其中Web里面的html代码会被执行，从而达到恶意攻击用户的特殊目的。

XssFilter用于解决如上恶意攻击问题，过滤器中会处理script代码，避免恶意植入。

**class** XssFilterWrapper **extends** HttpServletRequestWrapper {

**public** XssFilterWrapper(HttpServletRequest request) {

**super**(request);

}

/\*\*

\* 对数组参数进行特殊字符过滤

\*/

@Override

**public** String[] getParameterValues(String name) {

**if**("content".equals(name)){//不想过滤的参数，此处content参数是 富文本内容

**return** **super**.getParameterValues(name);

}

String[] values = **super**.getParameterValues(name);

**if**(values == **null**){

**return** **null**;

}

String[] newValues = **new** String[values.length];

**for** (**int** i = 0; i < values.length; i++) {

newValues[i] = HtmlUtils.*htmlEscape*(values[i]);//spring的HtmlUtils进行转义

}

**return** newValues;

}

}

**AOP 切面编程**

框架利用AspectJ实现切面编程效果，目前实现日志和事物两个切面处理。

1. **LogAspect**（日志切面）

在方法请求时，通过反射机制扫描类及方法的是否有Log注解，如果有则利用spring的EventListener机制进行异步日志输出处理，并通过Log注解的输出方式进行文件或数据库的写入操作。

@Around(value = "@annotation(com.bs.platform.core.annotations.Log)")

**public** Object saveLog(ProceedingJoinPoint joinPoint) **throws** Throwable {

/\*\*

\* 解析Log注解

\*/

Method currentMethod = currentMethod(joinPoint);

Log log = currentMethod.getAnnotation(Log.**class**);

Log classLog = joinPoint.getTarget().getClass().getAnnotation(Log.**class**);

/\*\*

\* 日志入库

\*/

String logStr = **null**;

**if** (classLog != **null**) {

logStr = classLog.value() + "-" + log.value();

} **else** {

logStr = log.value();

}

HttpServletRequest request = HttpServletUtils.*getRequest*();

**if** (log != **null** && logPoint != **null**) {

//发布LogRegisterEvent事件，异步处理日志入库事件

applicationContext.publishEvent(**new** LogRegisterEvent(**this**, log.outType(), joinPoint, currentMethod.getName(), logStr, request, getLoginUser()));

}

/\*\*

\* 方法执行

\*/

**return** joinPoint.proceed();

}

1. **TransactionalAspect**（事务切面）

在方法请求时，通过判断方法名自动启动事务。约定俗成save、insert、add、create、delete方法名开头的启动REQUIRED事务，get、query、find、list、count方法名开头的启动READONLY事务。

@Bean

**public** TransactionInterceptor txAdvice() {

DefaultTransactionAttribute txAttr\_REQUIRED = **new** DefaultTransactionAttribute();

txAttr\_REQUIRED.setPropagationBehavior(TransactionDefinition.***PROPAGATION\_REQUIRED***);

DefaultTransactionAttribute txAttr\_REQUIRED\_READONLY = **new** DefaultTransactionAttribute();

txAttr\_REQUIRED\_READONLY.setPropagationBehavior(TransactionDefinition.***PROPAGATION\_REQUIRED***);

txAttr\_REQUIRED\_READONLY.setReadOnly(**true**);

NameMatchTransactionAttributeSource source = **new** NameMatchTransactionAttributeSource();

source.addTransactionalMethod("save\*", txAttr\_REQUIRED);

source.addTransactionalMethod("insert\*", txAttr\_REQUIRED);

source.addTransactionalMethod("add\*", txAttr\_REQUIRED);

source.addTransactionalMethod("create\*", txAttr\_REQUIRED);

source.addTransactionalMethod("update\*", txAttr\_REQUIRED);

source.addTransactionalMethod("delete\*", txAttr\_REQUIRED);

source.addTransactionalMethod("get\*", txAttr\_REQUIRED\_READONLY);

source.addTransactionalMethod("query\*", txAttr\_REQUIRED\_READONLY);

source.addTransactionalMethod("find\*", txAttr\_REQUIRED\_READONLY);

source.addTransactionalMethod("list\*", txAttr\_REQUIRED\_READONLY);

source.addTransactionalMethod("count\*", txAttr\_REQUIRED\_READONLY);

**return** **new** ~~TransactionInterceptor~~(transactionManager, source);

}

**Model**

Model可以理解为面向对象的一个实体类， 这个类对应数据库中的一张表， 类的属性对应数据库表中字段

**1. Model**数据实体

DIP框架为业务表定义了一些公共字段， 例如ID、创建人ID、创建时间、修改人id、修改时间等， 并对这些公共信息进行了封装为基础BaseEntity，BaseEntity继承自MyBatis-Plus的Model，因此默认拥有MyBatis-Plus的相关能力。

BaseEntity ：

**public** **class** BaseEntity<T **extends** Model> **extends** Model<T> **implements** Serializable {

@TableField(exist = **false**)

**private** **static** **final** **long** ***serialVersionUID*** = -34115333603863619L;

/\*\*

\* 主键Id

\*/

@TableId(value = "id", type = IdType.***AUTO***)

@ApiModelProperty(name = "id", value = "主键", required = **true**, example = "")

**protected** Long id;

/\*\*

\* 创建人

\*/

@TableField("creator\_id")

@ApiModelProperty(name = "creatorId", value = "创建人", required = **true**, example = "")

**private** Long creatorId;

/\*\*

\* 创建时间

\*/

@TableField(value = "create\_time", fill = FieldFill.***INSERT***)

@ApiModelProperty(name = "createTime", value = "创建时间", required = **true**, example = "")

**private** String createTime;

/\*\*

\* 最后修改时间

\*/

@TableField(value = "last\_modify\_time", fill = FieldFill.***INSERT\_UPDATE***, update = "now()")

@ApiModelProperty(name = "lastModifyTime", value = "最后修改时间", required = **true**, example = "")

**private** String lastModifyTime;

/\*\*

\* 最后修改人

\*/

@TableField("last\_modifier\_id")

@ApiModelProperty(name = "lastModifierId", value = "最后修改人", required = **true**, example = "")

**private** Long lastModifierId;

**Dao**

数据库持久层使用的是开源 Mybatis + Plus和扩展方式实现， Dao层只需要提供API， 通过mybatis-spring的扫描机制， 自动组装实现类。业务系统Dao只需继承MyBatis-Plus的BaseMapper即可，自动继承了基础增删改查的通用能力。

**Service**

Service层是DIP框架的服务层， 在该接口中定义服务相关的方法名。 Service层提供事物支持， 对外提供原子操作的服务。 框架提供两种Service，分别为BaseService和CurdServiceImpl ,前者不提供增删改查功能，后者提供基于MyBatis-Plus的增删改查功能。服务层接口如果继承该接口通用功能就不必再重写基础方法。 只需根据业务需求对服务层接口方法进行扩展。

**1.** 框架**Service**接口

DIP框架在服务层提供父接口CurdServiceImpl<T> ， 该接口继承了MyBatis-Plus的ServiceImpl 。

在该接口中集成了通用的服务， 主要分为查询服务， 存储服务， 更新服务， 删除服务。 且对外提供原子操作的服务， 在方法定义上一行加注解 @Transactional() 保证服务操作的原子性， 即要不服务执行成功否则回到之前没有任何改变。

**1)** 查询服务

@Override

**public** T findById(ID id) {

}

@Override

**public** CommonResult findListByPage(T m, Page<T> page) {

}

@Override

**public** CommonResult findListByPage(T m, Page<T> page, String[] properties) {

}

@Override

**public** CommonResult list(T m, String[] properties) {

}

@Override

**public** List<T> selectList(T m, String[] properties) {

}

**2)** 存储服务

提供了createBefore和createAfter两个方法，供业务实现存储的前后扩展事件。

**public** CommonResult create(T m) {

CommonResult result = **new** CommonResult(**false**);

CommonResult beforeResult = createBefore(m);

**if** (beforeResult != **null**) {

**return** beforeResult;

}

**try** {

**if** (insert(m)) {

CommonResult afterResult = createAfter(m);

**if** (afterResult != **null**) {

**return** afterResult;

}

result.setData(m.getId());

result.setSuccess(**true**);

} **else** {

result.setMessage("添加失败!");

**return** result;

}

} **catch** (Exception e) {

*logger*.error(e.getMessage());

e.printStackTrace();

result.setMessage("添加数据出现异常!");

**return** result;

}

**return** result;

}

**3)** 修改服务

提供了updateBefore和updateAfter两个方法，供业务实现修改的前后扩展事件;

**4)** 删除服务

提供了deleteBefore和deleteAfter两个方法，供业务实现删除的前后扩展事件，并提供批量删除和单个删除方法。

public CommonResult delById(ID id) {

}

public CommonResult mulDelete(String ids) {

}

**Resource**层

Resource层提供rest风格的网络服务， 支持登录验证，日志控制等。框架提供两种Controller，分别为BaseController和BaseCurdController ,前者不提供增删改查功能，后者提供基于MyBatis-Plus的增删改查功能。 如果方法需要验证登录， 只需要添加Logined注解。

**1.** 实体相关

通用Resource层，提供增删改查基础方法

1. **BaseCrudController**

提供能满足大部分业务需求的数据库操作， API具体如下：

public abstract class BaseCrudController<T extends BaseEntity, S extends ICrudService<T, ID>, ID> extends BaseController {

@ApiOperation(value = "列表分页查询", notes = "")

@ResponseBody

@RequestMapping(value = "/list", method = RequestMethod.***POST***)

@Permission(value = "list")

**public** CommonResult list(T m, Integer pageSize, Integer page, Boolean simple, String sortField, String sortOrder) {

}

/\*\*

\* 增加

\*

\* **@param** m

\* **@return**

\*/

@ApiOperation(value = "新增", notes = "")

@ResponseBody

@RequestMapping(value = "/create", method = RequestMethod.***POST***)

@Permission(value = "create")

**public** CommonResult create(@Valid T m, BindingResult bindingResult) {

}

\*\*

\* 更新

\*

\* **@param** m

\* **@return**

\*/

@ApiOperation(value = "修改", notes = "")

@ResponseBody

@RequestMapping(value = "/update", method = RequestMethod.***POST***)

@Permission(value = "update")

**public** CommonResult update(@Valid T m, BindingResult bindingResult) {

}

/\*\*

\* 删除

\*

\* **@param** ids

\* **@return**

\*/

@ApiOperation(value = "删除", notes = "")

@ResponseBody

@RequestMapping(value = "/delete", method = RequestMethod.***POST***)

@Permission(value = "delete")

**public** CommonResult delete(String ids) {

}

**2.** 非实体相关

一些不需要写数据库， 不绑定实体或不需要提供标准增删查改服务的Resource类

继承**BaseController**即可

public class xxxResource extends BaseController

**Exception**

DIP框架对业务操作过程中的业务异常进行处理， 保证代码的健壮性。

**1.** 异常简介

框架提供业务异常处理类 BizException ， 它是所有业务操作异常结构体系的根对象， 该异常类继承自RuntimeException。同时框架在ResultCode类中定义了多种常见异常code及异常描述，业务系统用户可以根据使用ResultCode中的code抛出BizException或自定义业务code抛出异常即可。

BizException如下：

**public** **class** BizException **extends** RuntimeException {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **final** **long** errorCode;

**public** BizException(**long** ec, String message, Throwable cause) {

**super**(message, cause);

errorCode = ec;

}

**public** BizException(**long** ec, String message) {

**this**(ec, message, **null**);

}

**public** BizException(**long** ec, Throwable cause) {

**this**(ec, **null**, cause);

}

**public** **long** getErrorCode() {

**return** errorCode;

}

}

ResultCode中平台定义的异常如下所示：

/\*\*

\* 成功返回

\*/

***SUCCESS***(200, "成功"),

/\*\*

\* 未登录系统

\*/

***NOTLOGIN***(1000, "未登录系统！"),

/\*\*

\* 操作失败

\*/

***ERROR***(1001, "操作失败！"),

/\*\*

\* 权限不足

\*/

***PERMISSION\_DENIED***(1002, "权限不足"),

/\*\*

\* 不存在

\*/

***INEXISTENCE***(1003, "不存在"),

/\*\*

\* 存在

\*/

***EXIST***(1004, "存在"),

/\*\*

\* 参数为空

\*/

***PARAMNULL***(1005, "参数为空"),

/\*\*

\* 参数绑定校验异常

\*/

***BIND\_ERROR***(1006, "参数为空"),

/\*\*

\* 页面不存在

\*/

***NOT\_FOUND***(404, "Resource not found"),

/\*\*

\* 参数错误

\*/

***INVALID\_ARGUMENT***(11001, "Invalid argument"),

/\*\*

\* 未知错误

\*/

***UNKNOWN***(-1, "Unknown error"),

/\*\*

\* 服务内部错误

\*/

***SERVICE\_ERROR***(500, "服务内部错误"),

/\*\*

\* oauth2 code信息错误

\*/

***OAUTH2\_CODE\_ERROR***(1006, "CODE错误！");

**2.** 处理异常

在数据访问层、 服务实现层、 UI层遇到要处理的异常则抛出异常， 如下代码所示：

throw new BizException(ResultCode.INVALID\_ARGUMENT.getCode(), ResultCode.INVALID\_ARGUMENT.getMessage());;

**Util** 工具类

常用工具类可优先查看apache,spring文档

**1. OperatorUtils**

当前登录用户信息工具类， 提供API如下：

登录HttpServletRequest信息

private HttpServletRequest getRequest() {

登录用户信息（ 可以为null）

public OperatorBase getTokenUser() {

创建登录会话

public <T extends OperatorBase> String createToken(T loginUser, long seconds) {

移除登录用户

public void removeLoginUser(String token) {

获取后台管理登录用户id

public Long getLoginUserId() {

**2. CodeUtil**

SHA、MD5加密处理

// MD5加密

public static byte[] encryptMD5(byte[] data) throws Exception {

// SHA加密

public static byte[] encryptSHA(byte[] data) throws Exception {

**3. FileUtils**

文件处理， 提供API如下：

// 下载文件

public static void downloadFile(HttpServletResponse response,

String filePath, String fileName) {

// 删除文件

public static void deleteFile(String filePath){

**4. RestTemplateUtils**

RestTemplateUtils RestTemplate 远程调用工具类。 API如下代码块所示：

public static <T> ResponseEntity<T> get(String url, Class<T> responseType) {

public static <T> ResponseEntity<T> get(String url, Class<T> responseType, Object... uriVariables) {

public static <T> ResponseEntity<T> post(String url, HttpHeaders headers, Object requestBody, Class<T> responseType, Map<String, ?> uriVariables) {

…

**5. SpringContextUtils**

Spring上下文工具类。 API如下代码块所示：

public void setApplicationContext(ApplicationContext applicationContext)

public static Object getBean(String name) {

public static <T> T getBean(String name, Class<T> requiredType) {

public static boolean containsBean(String name) {

public static boolean isSingleton(String name) {

public static Class<? extends Object> getType(String name) {

…

**6. XMLUtil**

xml工具类主要用于XML文件的读取和解析、 将node节点转化成xml字符串、 通过xpath表达式查找节点， 并返回第一个符合条件节点、 saveXML。

// java对象转换为xml文件

public static String beanToXml(Object obj, Class<?> clazz) throws JAXBException {

// xml文件配置转换为对象

public static <T> T xmlToBean(String xmlPath, Class<T> clazz) throws Exception {

// xml转换成JavaBean

public static <T> T convertToJavaBean(String xml, Class<T> clazz) {

// JavaBean转换成xml

public static String convertToXml(Object obj) {

// JavaBean转换成xml

public static String convertToXml(Object obj, String encoding) {

// JavaBean转换成xml去除xml声明部分

public static String convertToXmlIgnoreXmlHead(Object obj, String encoding) {

**7. FStringUtils**

处理String类型的工具类， 具体的应用场景为如下所示：

将字符串source根据separator分割成字符串数组

public static List<String> listSplit(String source, String separator) {

将字符串source根据separator分割成字符串数组

public static String[] split(String source, String separator) {

判断str最后一个字符是否包含str2是的话删除

public static String removeLast(String str, String str2) {

将字符串source根据全局变量GobelConstants.SPLIT\_SEPARATOR分割成字符串数组

public static String[] split(String source) {

**8. DateUtils**

日期处理函数， API如下：

// 计算两日期相差的天数

public static int daysOfTwo(Date fDate, Date oDate) {

// 计算两日期相差的分钟数

public static long minutesOfTwo(Date oDate, Timestamp fDate) {

// 按pattern格式化输出Date

public static String dateToString(Date date, String pattern) {

// 按pattern格式化输出Date

public static String dateToString(Date date) {

// 将类型是pattern的日期字符串转化成Date型数据

public static Date toDate(String pattern, String dateString)

// 将类型是yyyy-MM-dd的日期字符串转化成Date型数据

public static Date toDate(String dateString) throws ParseException {

…

**开发规范**

本章节主要介绍使用**DIP**平台时必须遵循的一些开发规范！ ！

一、 包名命名规范

包名必须为： com.bs.it.xxx\*\*

二、 启动类命名规范

1. 启动类命名必须为xxxApplication

2. 启动类必须放到包的最顶层

三、 启动类的使用规范

启动类必须使用DipPlatform**.run(xxxApplication.class, args)**;这种形式； 参考下面实例：

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

DipPlatformApp.run(DemoApplication .class, args);

}

}