Java编码规范 V1.0

一、命名规范

1.包名

由一组以“.”连接的标识符构成，通常第一个标识符为符合网络域名的两个或者三个英文小写字母.

例如：com, edu,gov, mil, net, org, uk,jp.

举例：

```
com.JavaSoft.jag.Oak（错误示例:应该用小写）
uk.ac.city.rugby.name
org.npr.plddge.driver
要注意的是，根据Sun公司的规定，“java”不能用于第一个标识符。
```

注意：

机会多业务包名命名规范为：以com.hans.jhd为基础，后接不同业务模块。

示例如下：

```
com.hans.jhd.security
com.hans.jhd.ucenter
```

2.类和接口命名

2.1 整体说明

类名或者接口名由一组描述性的名词或者短语构成，不应该过长，每个单词的首字母都应该大写

举例：

```
ClassLoader
SecurityManager
Thread
Dictionary
```

类名必须使用名词，如果一个类名内含多个单词，那么各个单词第一个字母大写，后续字母小写，起伏呈驼峰状，人称驼峰式命名。给类名命名时，必须保证准确、简洁且容易理解。尽量使用完整单词，避免使用缩写词（除了大家公认的）

（1）选择有意义的名字，能快速地传达该类的用途。

（2）参照java驼峰命名法，类名的首字母必须采用大写的形式，如果类名为多词组合而成的话，那么每个词的首字母必须采用大写。

• 如：StudentAnswer.java

（3）**为了区别接口类和实现类，在接口前面加上前缀I。注意接口实现类无需再加Impl后缀。**

• 如：接口类：IUserxxx.java　接口实现类：Userxxx

（4）领域类不就需要加后缀名。

（5）为了区分枚举和类，在枚举前加上前缀E

• 如：实体状态枚举EEntityState

```
/**
 * 实体状态枚举（对应数据库中记录）
 * Created by Albert.Liu on 15/9/14.
 */
public enum EEntityState {
    VALID(0, "有效"),
    INVALID(1, "无效");

    public Integer val;
    public String desc;

    EEntityState(Integer val, String desc) {
        this.val = val;
        this.desc = desc;
    }

    public static EEntityState getTypeByVal(Integer val) {
        EEntityState defaultState = VALID;
        for (EEntityState eEntityState : EEntityState.values()) {
            if (eEntityState.val.equals(val)) {
                return eEntityState;
            }
        }
        return defaultState;
    }

    public static String getDescByVal(Integer val) {
        return getTypeByVal(val).desc;
    }

}
```

2.2 各层中类和接口的命名

2.2.1.概述

所谓领域驱动DDD，是以一种领域专家、设计人员、开发人员都能理解的通用语言作为相互交流的

工具，

在交流的过程中发现领域概念，然后将这些概念设计成一个领域模型；

由领域模型驱动软件设计，用代码来实现该领域模型领域驱动设计思想中，每一个项目会按一下层次进

行划分，下面分层介绍每一层的命名规范

通用的领域驱动模型可分为：

- infra基础设施层
- domain领域层
- application应用层
- facade门面接口层
- facade-impl门面实现层
- webapp用户接口层（采用web形式）

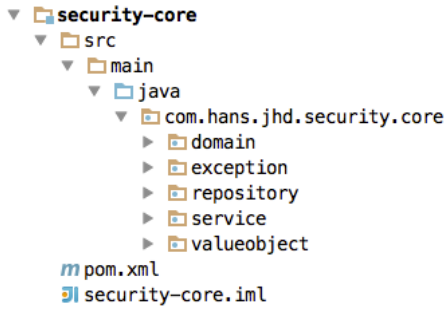
通用项目截图例如下图：

2.2.2.领域层

2.2.2.1 概述

负责表达业务概念，业务状态信息以及业务规则，领域模型处于这一层，是业务软件的核心。

领域层实例图如下:



整体包名为：com.hans.jhd.security.core(hans为大族集团简写，jhd为机会多简写，security为业务模块)

2.2.2.2 Domain领域实体

领域实体包命名格式 xxx.xxx.core.domain

> 例:com.hans.jhd.security.core.domain

该包为所有实体对象的位置，一般继承自对应实体DTO（数据传输对象，后边介绍），一般不包含数据库对应的属性，

含有该对象所具有的方法，区别于传统MVC中domain实体对象只包含对象属性，用于数据库简单映射这种“毫无灵魂”

的设计。例如：

> 汽车Car的基本属性在CarDTO中，领域实体Car继承自CarDTO，在Car中包含汽车所具有的方法，start()、
> stop()等功能。

注意：实体内容编写顺序，字段->构造函数->方法(动->静)->业务主键->属性

2.2.2.3 Exception异常

异常类放在xxx.xxx.core下，以xxxException结尾，必须是RuntimeException 的子类

例如：

```
public class UserAccountIsExistedException extends RuntimeException {}
```

该包为所有自定义异常所在的位置，用于在常用的业务逻辑出现异常时，进行获取异常，常见的如用户密码错误异常、用户账户已存在异常等。

完整实例如下：

```
public class UserAccountIsExistedException extends RuntimeException {

    private static final long serialVersionUID = 2631317949499926794L;

    public UserAccountIsExistedException() {
    }

    public UserAccountIsExistedException(String message) {
        super(message);
    }

    public UserAccountIsExistedException(Throwable cause) {
        super(cause);
    }

    public UserAccountIsExistedException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

2.2.2.4 Repository仓储

该包为Domain进行操作数据库的最外层数据接口，包含CRUD方法，该接口名以I开头Repository结尾，例如：

```
com.hans.jhd.security.core.repository.IUserRepository
```

IUserRepository表示User操作数据的仓储接口，可以对用户进行数据操作。

仓储接口方法命名规范：

包含最简单的CRUD操作，使用insert、delete、update、select这些字母开头，不应包含业务逻辑

完整接口如下：

```
public interface IUserRepository {

    int deleteByPrimaryKey(Long id);

    int insertSelective(Actor record);

    Actor selectByPrimaryKey(Long id);

    int updateByPrimaryKeySelective(Actor record);
}
```

> 注意：在实现仓储的过程中，不准使用关联查询。
>
> 关联查询通过多次单表查询实现。比如：先从一个表中获取相关数据，然后通过该相关数据去另一个表中进行查找。

2.2.2.5 Service服务层

该包为Service，主要用于当有多个实体进行相互访问的时候，进行的关联操作。

命名为三段式：所获取实体+服务类别+Service，类名例如：

```
AuthorityRelationQueryService
```

表示关联查询服务，最终查询得到的实体为Authority，即多个实体之间相互访问，最终获得是Authority.

完整实例如下:

```
@Named
public class AuthorityRelationQueryService {

    @Inject
    private User user;
```

```
    @Inject
    private Role role;

    @Inject
    private Authorization authorization;

    public List<Authority> findAllAuthoritysBy(String userAccount) {
        User user = this.user.getByUserAccount(userAccount);
        return listAllAuthoritysByUser(user);
    }

    public Set<Authority> listAllRolesByUser(User user) {
        List<Authorization> authorizations = this.authorization.findByActor(user);
        Set<Long> authorityIdSet = Authorization.listAuthorityId(authorizations);
        Set<Authority> roleSet = this.role.listRoleBySelfIds(authorityIdSet);
        return roleSet;
    }
}
```

可以看出，通过User、Role、Authorization实体类的帮助最终获得所需要的Authority实体对象。

因此命名规范上便于排序，如果后期还有其他类别的服务类型，根据三段式的方式，所属实体在最前边，

便于看出各实体的服务。

Service方法命名规范：

• 获取单个对象：get

• 获取列表文本 ：list

• 获取分页 ：page

• 搜索：支持模糊查询的用search，不支持模糊查询的用query

• 添加：add、addBatch ---- 批量添加

• 创建：create与insert的区别在于insert中只是直接的入数据库插入一条记录，而create不仅包含了insert这个操作，还含有通过一定业务逻辑组装要存储对象的过程

• 先删除后保存：add 、addBatch---- 先删除后保存

• 删除：delete，deleteBatch,remove与delete的区别是,remove包含其他的业务逻辑处理,delete则不包含

• 修改：update，updateBatch

• 存在插入，不存在更新：save、saveBatch

注意：不建议使用get、set的方式命名，以避免和领域实体中属性的get、set方法向冲突。

2.2.2.6 ValueObject值对象

在领域中，并不是每一个事物都必须有一个唯一标识，即并不是每一个事物都会对应一个实体对象，也就是说我们不关心对象是哪个，而只关心对象是什么。

ValueObject的使用场景有：

1、代表查询数据库中的条件

• 例：QueryCondSecurityResource表示查询SecurityResource数据库的条件，该条件可能并不会和SecurityResource实体的属性全部对应，比如，在查询条件中会有page分页的功能，都有可能添加在该类中。

示例如下：

```
public class QueryCondUser extends UserDTO {
    /**
     * 分页相关信息
     */
    private Page page = new Page();

    public Page getPage() {
        if (page == null) {
            page = new Page();
        }
        return page;
    }

    public void setPage(Page page) {
        this.page = page;
    }
}
```

2、代表存储数据库查询的结果

• 在进行关联查询，或者多表查询的时候，查询到的结果，可能并不会和任一实体对象相对应，因此可以单独一个对象，用于存储该查询结果，完整实例如下：

```
public class User {
    private String name;
    private String sex;
    private String age;
}
```

```
public class Address {
    private String country;
    private String province;
    private String city;
}
```

```
public class QueryResultOfUserAndAddress {
    private String country;
    private String province;
    private String city;
    private String name;
    private String sex;
    private String age;
}
```

表示User和Address，但是查询User的信息时，可以使用QueryResultOfUserAndAddress对象进行存储。

2.2.3.基础设施层

2.2.3.1 概述

infra包含工具类、常量信息、数据仓储等包。

整体包名为：com.hans.jhd.security.infra(hans为大族集团简写，jhd为机会多简写，security为业务模块)

2.2.3.2 Common公共包

common用于存放常量信息，工具类等，属于项目公共的部分

• constant表示常量，命名格式为：xxxConstant

例：RedisKeyIdNumConstant

• utils表示工具类，命名格式为：xxxUtil

例：RedisUtil

2.2.3.3 Repository仓储

repository下分三个包：nosql、sql、openapi

• nosql表示数据的来源方式为nosql，如：缓存Redis等

命名格式为：xxxRedis

• sql表示数据的来源方式为sql，如：mysql数据库等

命名格式为：xxxMapper(这里以使用MyBatis做持久层框架为例)

• openapi表示数据的来源为第三方api，如：聚合数据的api等

命名格式为：xxxAPI

• ActorRepository为领域层repository的接口实现类,

命名格式为：xxxRepository

示例代码:

```
@Named
public class ActorRepository implements IActorRepository {

    @Inject
    private ActorMapper actorMapper;

    @Inject
    private ActorRedis actorRedis;

    @Inject
    private ActorApi actorApi;

    public int deleteByPrimaryKey(Long id) {
        //删除缓存数据
        actorRedis.deleteByPrimaryKey(id);
        return actorMapper.deleteByPrimaryKey(id);

    }

    public Actor selectByPrimaryKey(Long id) {
        //从缓存数据中选择数据
        Actor actor = new Actor();
        actor = actorRedis.selectByPrimaryKey(id);
        //如果缓存中没有数据的话，择取数据库中进行读取
        if(actor == null){
            actor = actorMapper.selectByPrimaryKey(id);
        }
        return actor;
    }
```

该分层结构的优点：

根据领域驱动的规范，在Domain层调用Repository，实体对象的操作固然包含CRUD，既是我们需要进行对

数据库的操作，当我们只有一个数据源的时候，很简单，但是后期项目中数据源可能会增加，可能会添加缓存

等，这样的话使用原来的MVC模式的话，我们可能需要修改很多，如下图：

上图中包含了领域驱动的调用过程，我们在后期如果添加多个数据源的话，我们只需要修改Repository的实现即可，无需

改动领域层，以实现对整个领域层的透明。

> 注意：在实现仓储的过程中，不准使用关联查询。

可以使用：现在一个表中获取相关数据，然后通过该相关数据去另一个表中进行查找。

2.2.4.应用层

2.2.4.1 概述

这一层主要是操作实体对象的，是于数据更近一层的操作，主要定义了用户所拥有的方法和属性，操作实体对象层，将实体对象所有的方法展示出来供用户使用；

整体包名为：com.hans.jhd.security.application(hans为大族集团简写，jhd为机会多简写，security为业务模块)

2.2.4.2 命名规范

整体包名为：com.hans.jhd.security.application(hans为大族集团简写，jhd为机会多简写，security为业务模块)

1、com.hans.jhd.security.application下为接口

• 例：IAuthorityApplication表示接口，格式为IxxxApplication

2、com.hans.jhd.security.application.impl为接口的实现类

• 例：AuthorityApplication表示接口的实现类，格式为xxxApplication

3、方法命名规范

• 获取单个对象：select

• 获取这里是列表文本列表 ：list

• 获取分页 ：page

• 搜索：支持模糊查询的用search，不支持模糊查询的用query

• 添加：add、addBatch（批量添加）

• 创建：create与insert的区别在于insert中只是直接的入数据库插入一条记录，而create不仅包含了insert这个操作，还含有通过一定业务逻辑组装要存储对象的过程

• 先删除后保存：add 、addBatch（先删除后保存）

• 删除：delete，deleteBatch,remove与delete的区别是,remove包含其他的业务逻辑处理,delete则不包含

• 修改：update，updateBatch（批量修改）

• 存在插入，不存在更新：save、saveBatch

> 注意：不要使用get、set开头的方法，以避免和领域实体中的get、set方法冲突带来的问题。

4、其他注意事项

• 接口必须编写详细的注释，包括创建时间，创建人，类的意义以及每个方法的意义

• 接口方法必须编写相对应的单元测试，并且有较高的覆盖率

• 事务标注在接口实现类

• 接口定义不需要写作用范围，默认 public

完整示例：

• Application接口：

```
public interface IAuthorityApplication {

    /**
     * 根据用户账户查找该用户拥有的所有角色。
     * @param userAccount 用户账户
     * @return 用户的所有角色集合
     */
    List<Role> findAllRolesByUserAccount(String userAccount);

    /**
     * 根据角色名称得到角色。
     * @param roleName 角色名称
     * @return 角色
     */
    Role getRoleBy(String roleName);
}
```

• Application接口实现类：

```
@Named
public class AuthorityApplication implements IAuthorityApplication {

    @Inject
    private AuthorityRelationQueryService authorityRelationQueryService;

    @Inject
    private Permission permission;

    @Inject
    private Role role;

    public List<Role> findAllRolesByUserAccount(String userAccount) {
        Set<Role> roleSet =
authorityRelationQueryService.findAllRolesByUserAccount(userAccount);
        return Lists.newArrayList(roleSet);
    }

    public Role getRoleBy(String roleName) {
        return role.selectByName(roleName);
    }
```

```
}
```

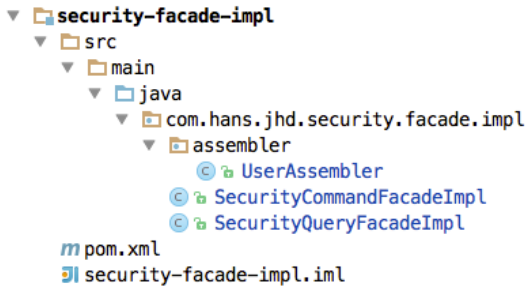
可以看出在Application应用层中使用到了领域层实体对象和领域层的服务，这也是领域驱动中推荐使用的调用逻辑方式。

2.2.5.门面层

2.2.5.1 概述

1、门面层：

2、门面层的实现层：



门面层主要分为两层，一层是门面接口的定义，另一层是门面接口定义的实现。

门面层接口可以看做是暴露给外部用户或系统（或者是web层）使用的接近底层数据最外边的一个层次结构，该层

通过调用应用层，通过应用层将领域实体所具有的能力（也就是方法属性）展示位外部用户，而使用该门面接口的

用户或系统，完全不用考虑具体内部的实现机制，只需要根据不同方法，解决不同的用户需求，应用层以下对用户

或系统完全透明。

2.2.5.2 门面接口层

门面接口层，主要包含dto包和门面层方法

1、dto包

（a）dto主要进行数据的请求Command（request包中）和数据实体对象属性的存放位置，例如：

> 汽车Car的基本属性在CarDTO中，领域实体Car继承自CarDTO，在Car中包含汽车所具有的方法，start()、
> stop()等功能。

这是领域层的一个实例，可以看出这种方式的优点，同样可以实现属性和方法分离开来，通过继承的关系来

达到领域实体的完整。

完整包名为：com.hans.jhd.security.facade.dto.xxxDTO

（b）Command主要是在Controller中使用的http和参数的映射，例如：

```
public class LoginCommand {

    private String username;

    private String password;

    private String rememberMe;

    //get、set方法
}
```

命名规范为：xxxCommand,xxx表示命令的行动类型。

2、门面层

之所以成为门面层，我们暂可认为是我们用户可以看到的整个系统的东西，门面接口的定义又分为：**读和写**，

主要用于后期项目数据量较大时，实现读写分离。

- SecurityQueryFacade表示读的门面接口，格式为xxxQueryFacade的三段式方式
- SecurityCommandFacade表示写的门面接口，格式为xxxCommandFacade的三段式方式

另外，使用Dubbo或者Dubbox的方式，将该接口暴露给其他系统使用者，也是该门面层接口，以实现底层的完全透明。

2.2.5.3 门面实现层

1、Assember

领域层实体和数据传输对象DTO的转换，例如：User和UserAssember，前边用于和数据打交道，UserDTO由

于只包含领域实体的属性所以更擅长与网络请求打交道，用与在网络上进行数据的传输。

命名规范:xxxAssember,例如：UserAssembler

方法命名规范：

- 领域实体转化为DTO：toDto()
- DTO转化为领域实体：toXxx(),例如：toUser()
- 领域实体集合转化为DTO：toDtos()
- 多个DTO集合转化为领域实体：toXxxs(),例如：toUsers()

2、门面接口的实现类

命名方式为：xxxFacadeImpl，同样分为读写，例如：

SecurityQueryFacadeImpl和SecurityCommandFacadeImpl

2.2.6.用户界面层

2.2.6.1 概述

用户界面层一般为Web层，即使用SpringMVC的方式。

完整包名为：com.hans.jhd.security.controller

同样，对于有不同的业务逻辑模块，需要进行单独的分开，例如：

```
com.hans.jhd.security.controller
com.hans.jhd.ucenter.controller
```

2.2.6.2 Controller

1、当采用SpringMVC的时候，所有的Controller都放在此包中，格式为：xxxController

完整实例如下：

```
@Controller
@RequestMapping("/auth/user")
@SuppressWarnings("unused")
public class UserController {

    @Inject
    private SecurityCommandFacade securityCommandFacade;

    /**
     * 添加用户
     */
    @ResponseBody
    @RequestMapping(value = "/add", method = RequestMethod.POST)
    public Response add(CreateUserCommand command) {
        String createOwner = CurrentUser.getUserAccount();
        command.setCreateOwner(createOwner);
        return securityCommandFacade.createUser(command);
    }
}
```

可以看出在Controller中使用到了SecurityCommandFacade即使用到了门面层的方法，这也是领域驱动规范

的使用方式。

2、Controller方法命名规范：

- 获取单个对象：select
- 获取这里是列表文本列表 ：list
- 获取分页 ：page
- 搜索：支持模糊查询的用search，不支持模糊查询的用query

• 添加：add、addBatch ---- 批量添加

• 创建：create与insert的区别在于insert中只是直接的入数据库插入一条记录，而create不仅包含了insert这个操作，还含有通过一定业务逻辑组装要存储对象的过程

• 先删除后保存：add 、addBatch---- 先删除后保存

• 删除：delete，deleteBatch,remove与delete的区别是,remove包含其他的业务逻辑处理,delete则不包含

• 修改：updat，updateBatch

• 存在插入，不存在更新：save、saveBatch

方法命名原则：能够突出该方法所代表的方法意义

2.2.6.3 Filter

过滤器，格式为：xxxFilter

2.2.6.4 Listener

系统Listener，命名格式为：xxxListener

继承自:ServletContextListener,示例如下：

```
public class SystemEvenmentListener implements ServletContextListener {

    private static final Logger LOGGER =
LoggerFactory.getLogger(SystemEvenmentListener.class);

    public void contextInitialized(ServletContextEvent sce) {
        WebApplicationContext applicationContext =
WebApplicationContextUtils.getRequiredWebApplicationContext(sce.getServletContex
t());
        CustomDefaultFilterChainManager filterChainManager =
applicationContext.getBean(CustomDefaultFilterChainManager.class);
        ShiroFilterChainManager shiroFilterChainManager =
applicationContext.getBean(ShiroFilterChainManager.class);
        filterChainManager.init();
        shiroFilterChainManager.init();
        shiroFilterChainManager.initFilterChain();
        LOGGER.info("init System Evenment.");
    }

    public void contextDestroyed(ServletContextEvent sce) {
    }

}
```

2.2.6.5.Resources

2.2.6.5.1 概述

该部分为Spring框架设置、数据源设置、MyBatis等，目录结构分为:

- mybatis
- props
- spring
- Spring的启动文件root.xml
- SpingMVC的启动文件springmvc-servlet.xml

2.2.6.5.2 mybatis

mybatis下分为三部分：mappers、db-mybatis.xml、脚本文件

1、mapper

mapper下分为security和ucenter两个不同的业务模块，MyBatis的映射文件格式为：xxxMapper

2、db-mybatis.xml

为sessionFactory，代码为：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="sessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
      <property name="dataSource" ref="dataSource"/>
      <property name="mapperLocations" value="classpath:mybatis/mappers/*.xml"/>
   </bean>

   <bean id="mapperScannerConfigurer"
class="org.mybatis.spring.mapper.MapperScannerConfigurer">
      <property name="basePackage" value="org.albert.security.infra.repository.sql"/>
      <property name="sqlSessionFactoryBeanName" value="sessionFactory"/>
   </bean>
</beans>
```

3、脚本文件

为映射数据库文件的脚本文件。

2.2.6.5.2 props

该文件为配置文件的所在位置

可以使用下边的方式进行引入：

```
<context:property-placeholder location="classpath*:/props/*.properties" ignore-unresolvable="true"/>
```

常见的prop配置如下：

• MySQL数据库配置

```
db.jdbc.driver=com.mysql.jdbc.Driver
db.jdbc.connection.url=jdbc:mysql://127.0.01:3306/mypip_security?useUnicode=true&amp;characterEncoding=UTF-8
db.jdbc.username=你的名字
db.jdbc.password=你的密码
db.jdbc.dialect=org.hibernate.dialect.MySQL5Dialect
db.jdbc.testsql=select 1
hibernate.hbm2ddl.auto=update
db.jdbc.show_sql=true
db.jdbc.database.Type=MYSQL
db.jdbc.generateDdl=true
db.jdbc.maximumConnectionCount=200
db.jdbc.minimumConnectionCount=20
```

注：如果有多个数据源，使用db.security.jdbc.driver的方式，其中security为业务模块

• Redis缓存数据库配置

```
redis.ip=127.0.0.1
redis.port=6379
redis.password=你的密码
redis.pool.maxTotal=10
redis.pool.minIdle=4
redis.pool.maxIdle=8
redis.pool.testOnBorrow=true
```

• Log4j配置文件

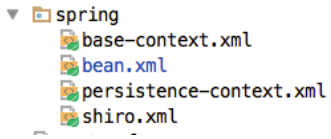
```
log4j.rootLogger=DEBUG,rolling,errlog,stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d{HH:mm:ss}][%X{traceId}][%-5p][%c{1}.%M:%L] %m%n
```

```
common log
log4j.appender.rolling=org.apache.log4j.DailyRollingFileAppender
log4j.appender.rolling.File=${catalina.base}/logs/security.log
log4j.appender.rolling.DatePattern='.'yyyy-MM-dd-HH
log4j.appender.rolling.layout=org.apache.log4j.PatternLayout
log4j.appender.rolling.layout.ConversionPattern=[%d{HH:mm:ss.SSS}][%X{traceId}][%-5p][%c{1}.%M:%L] %m%n

error log
log4j.appender.errlog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.errlog.Threshold=ERROR
log4j.appender.errlog.File=${catalina.base}/logs/error_security.log
log4j.appender.errlog.DatePattern='.'yyyy-MM-dd
log4j.appender.errlog.layout=org.apache.log4j.PatternLayout
log4j.appender.errlog.layout.ConversionPattern=[%d{HH:mm:ss.SSS}][%X{traceId}][%-5p][%c{1}.%M:%L] %m%n
```

2.2.6.5.3 spring



该文件存Spring框架的配置文件，可以根据不同的功能进行划分

基础的Spring配置文件如下：

1、base-context.xml

基础的加载properties配置，包的扫描配置等

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="classpath*:/props/*.properties" ignore-unresolvable="true"/>

    <context:component-scan base-package="org.albert.security.*">
```

```
        <context:exclude-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

</beans>
```

2、bean.xml

一些需要手动进行注入的bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

    <bean id="qqOauth" class="com.qq.connect.oauth.Oauth"/>

    <bean id="securityQueryFacade"
class="com.hans.jhd.security.facade.impl.SecurityQueryFacadeImpl"/>

    <bean id="springGetBeanUtil" class="org.albert.security.utils.SpringGetBeanUtil"/>

</beans>
```

3、persistence-context.xml

持久层配置，用于配置数据源等

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="dataSourceTransactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
          init-method="init" destroy-method="close">
        <!-- 数据库基本信息配置 -->
        <property name="driverClassName" value="${db.jdbc.driver}"/>
        <property name="url" value="${db.jdbc.connection.url}"/>
```

```
        <property name="username" value="${db.jdbc.username}"/>
        <property name="password" value="${db.jdbc.password}"/>
        <!-- 初始化连接数量 -->
        <property name="initialSize" value="10"/>
        <!-- 最大并发连接数 -->
        <property name="maxActive" value="100"/>
        <!-- 最小空闲连接数 -->
        <property name="minIdle" value="20"/>
        <!-- 配置获取连接等待超时的时间 -->
        <property name="maxWait" value="5000"/>
        <!-- 超过时间限制是否回收 -->
        <property name="removeAbandoned" value="true"/>
        <!-- 超过时间限制多长； -->
        <property name="removeAbandonedTimeout" value="120000"/>
        <!-- 配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒 -->
        <property name="timeBetweenEvictionRunsMillis" value="60000"/>
        <!-- 配置一个连接在池中最小生存的时间，单位是毫秒 -->
        <property name="minEvictableIdleTimeMillis" value="40000"/>
        <!-- 用来检测连接是否有效的sql，要求是一个查询语句-->
        <property name="validationQuery" value="select 1"/>
        <!-- 申请连接的时候检测 -->
        <property name="testWhileIdle" value="true"/>
        <!-- 申请连接时执行validationQuery检测连接是否有效，配置为true会降低性能 -->
        <property name="testOnBorrow" value="false"/>
        <!-- 归还连接时执行validationQuery检测连接是否有效，配置为true会降低性能  -->
        <property name="testOnReturn" value="false"/>
        <!-- 打开PSCache，并且指定每个连接上PSCache的大小 -->
        <property name="poolPreparedStatements" value="true"/>
        <property name="maxPoolPreparedStatementPerConnectionSize"
                value="50"/>
        <!--属性类型是字符串，通过别名的方式配置扩展插件，常用的插件有：
                监控统计用的filter:stat
                日志用的filter:log4j
               防御SQL注入的filter:wall -->
        <property name="filters" value="stat"/>
    </bean>
</beans>
```

2.2.7.各层次调用顺序

2.2.7.1 各层在运行时对象调用关系

2.2.7.2 各层在编译时的类依赖关系

3.方法命名

1、一般的方法命名规范如下：

• 应该由动词或者动词短语混合构成，第一个单词小写，此后每个单词的首字母大写，驼峰式的命方式

• 设置/获取某个值的Method，应该遵循selectV/findV规范

• 返回长度的Method，应该命名为length

• 测试某个布尔值的Method，应该命名为isV

• 将对象转换为某个特定类型的Mehod应该命名为toF

2、要求如下：

• 1、方法表示一种行为，它代表一种动作，最好是一个动词或者动词词组或者第一个单词为一个动词。

• 2、属性方法：以get/set开头，其后跟字段名称，字段名称首字母大写。如：getUserName()

• 3、数据层方法：只能以insert（插入）,delete（删除）,update（更新）,select（查找）,count（统计）开头，其他层方法避免以这个5个单词开头，以免造成误解。

• 4、服务层方法：根据方法的行为命名，只描述方法的意义，而不采用方法的目的命名。比如系统的添加新用户，用户可以前台注册，也可以管理员后台添加，方法会被重用，所以最好不要用使用register，采用add会更好写。避免使用与web层相关的方法。

• 5、Web层方法：最好是贴近web的语言，如register，login，logout等方法。

3、举例如下：

```
getDate();
length();
isReady();
toOracleFormat();
```

4、总结如下：

• 获取单个对象：get

• 获取这里是列表文本列表 ：list

• 获取分页 ：page

• 搜索：支持模糊查询的用search，不支持模糊查询的用query

• 添加：add、addBatch ---- 批量添加

• 创建：create与insert的区别在于insert中只是直接的入数据库插入一条记录，而create不仅包含了insert这个操作，还含有通过一定业务逻辑组装要存储对象的过程

• 先删除后保存：add 、addBatch---- 先删除后保存

• 删除：delete，deleteBatch,remove与delete的区别是,remove包含其他的业务逻辑处理,delete则不包含

• 修改：updat，updateBatch

• 存在插入，不存在更新：save、saveBatch

4.属性命名

由名词、名词短语或者名词的缩写构成第一个单词小写，此后每个单词的首字母大写，采用驼峰式命名规范

举例：

```
java.io.ByteArrayInputStream中的buf, pos,count
java.io.InterruptedIOException中的bytesTransferred
```

5.常量命名

规范：由多个单词或缩写组合而成，所有字母都要求大写，并且以“_”连接

举例如下：

```
MIN_VALUE
MAX_VALUE
MIN_RADIX
MAX_RADIX
```

完整实例如下：

```
public interface ConstOrder {

    String PREFIX = "ORDER";

    String PREFIX_ = PREFIX + ConstPunctuation.UNDERLINE;

    String PAGE = PREFIX_ + "PAGE";

    String GET_DETAIL = PREFIX_ + "GET_DETAIL";

    String UPDATE = PREFIX_ + "UPDATE";

    String REMOVE = PREFIX_ + "REMOVE";

    String APPLY_TICKET = PREFIX_ + "APPLY_TICKET";
```

```
    String BILL = PREFIX_ + "BILL";

    String INVALID = PREFIX_ + "INVALID";
}
```

6.局部变量和参数命名

由意义明确的短词构成，通常均为小写，而且不是完整的单词

```
cp代表colorPoint
buf代表buffer
off代表offset
len代表length
```

除非是在循环中，否则一般不推荐使用单个字母作为变量名，不过也有例外，即约定俗成的单个字母

```
b代表byte
c代表char
d代表double
e代表Exception
f代表float
i, j, k代表整数
l代表long
o代表Object
s代表String
v代表某些类型的特定值
```

二、代码注释规范

1.概述

> **代码注释原则**
>
> 好的编码规范就是最好的注释

如果遵循上述的命名规范，基本的代码都可以见名知意，但对于特殊方法，负责业务逻辑的方法需要对其进行注释。

2.基本注释

- 类（接口）的注释
- 构造函数的注释

- 方法的注释
- 全局变量的注释
- 字段/属性的注释

备注：简单的代码做简单注释，注释内容不大于10个字即可，另外，持久化对象或VO对象的getter、setter方法不需加注释。

具体的注释格式请参考下面举例。

3.特殊必加注释

- 典型算法必须有注释。
- 在代码不明晰处必须有注释。
- 在代码修改处加上修改标识的注释。
- 在循环和逻辑分支组成的代码中加注释。
- 为他人提供的接口必须加详细注释。
- 复杂逻辑或者业务加注释。

备注：此类注释格式暂无举例。

具体的注释格式自行定义，要求注释内容准确简洁。

注释格式：

1、单行(single-line)注释： “//……”

2、块(block)注释： “/……/”

3、文档注释： “/*……/”

4、javadoc 注释标签语法

```
@author 对类的说明 标明开发该类模块的作者
@version 对类的说明 标明该类模块的版本
@see 对类、属性、方法的说明 参考转向，也就是相关主题
@param 对方法的说明 对方法中某参数的说明
@return 对方法的说明 对方法返回值的说明
@exception 对方法的说明 对方法可能抛出的异常进行说明
```

4.参考举例

1.类（接口）注释 例如：

```
/**
* 类的描述
* @author Administrator
* @Time 2012-11-2014:49:01
```

```
*/
public classTest extends Button {
  ……
}
```

2.构造方法注释 例如:

```
public class Test extends Button {
  /**
   * 构造方法 的描述
   * @param name 按钮的上显示的文字
   */
  public Test(String name){
     ……
  }
}
```

3.方法注释 例如：

```
public class Test extends Button {
  /**
   * 为按钮添加颜色
   *@param color 按钮的颜色
   *@return
   *@exception  (方法有异常的话加)
   *@author Administrator
   *@Time2012-11-20 15:02:29
   */
  public voidaddColor(String color){
     ……
  }
}
```

4.全局变量注释 例如：

```
public final class String  implements java.io.Serializable,
Comparable<String>,CharSequence
{
   /** The value is used for characterstorage. */
   private final char value[];
   /** The offset is the first index of thestorage that is used. */
   private final int offset;
   /** The count is the number of charactersin the String. */
   private final int count;
```

```
    /** Cache the hash code for the string */
    private int hash; // Default to 0
……
}
```

5.字段/属性注释 例如：

```
public class EmailBody implements Serializable{
    private String id;
    private String senderName;//发送人姓名
    private String title;//不能超过120个中文字符
    private String content;//邮件正文
    private String attach;//附件，如果有的话
    private String totalCount;//总发送人数
    private String successCount;//成功发送的人数
    private Integer isDelete;//0不删除 1删除
    private Date createTime;//目前不支持定时 所以创建后即刻发送
    privateSet<EmailList> EmailList;
……
}
```