文档名称	密级
Java开发手册	机密
文档版本	共31页
1.0	

Java开发手册

拟制		日期	2017-06-30
审核		日期	yyyy-mm-dd
批准		日期	yyyy-mm-dd

北京君德财富投资管理股份有限公司

北京微金客科技有限公司

修订记录

日期	修订版本	CR号	修改章节	修改描述	作者
2017-06-30	1.00			初稿完成	

目录

Java开发手册

关键词：君德财富，微金客，java，开发，手册

摘　要：本文档针对北京君德财富投资管理股份有限公司和北京微金客科技有限公司Java开发相关人员制定开发手册，可作为公司研发部门人员的参考文档。

缩略语清单：

缩略语	英文全名	中文解释
DO	Data Object	本手册指数据库表一一对应的POJO类
GAV	GroupId、ArtifactctId、Version	Maven坐标，是用来唯一标识jar包
NPE	java.lang.NullPointerException	空指针异常
OOP	Object Oriented Programming	面向对象编程，本手册泛指类、对象的编程处理方式
ORM	Object Relation Mapping	对象关系映射，对象领域模型与底层数据之间的转换，本文泛指iBATIS, mybatis等框架
POJO	Plain Ordinary Java Object	在本手册中，POJO专指只有setter / getter / toString的简单类，包括DO/DTO/BO/VO等
SOA	Service-Oriented Architecture	面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用，有利于提升组件可重用性，可维护性
一方库	/	本工程内部子项目模块依赖的库（jar包）
二方库	/	公司内部发布到中央仓库，可供公司内部其它应用依赖的库（jar包）
三方库	/	公司之外的开源库（jar包）

1 前言

现代软件行业的高速发展对开发者的综合素质要求越来越高，因为不仅是编程知识点，其它维度的知识点也会影响到软件的最终交付质量。比如：数据库的表结构和索引设计缺陷可能带来软件上的架构缺陷或性能风险；工程结构混乱导致后续维护艰难；没有鉴权的漏洞代码易被黑客攻击等等。所以本手册以 Java开发者为中心视角，划分为编程规约、异常日志、工程结构、安全规约四大块，再根据内容特征，细分成若干二级子目录。根据约束力强弱及故障敏感性，规约依次分为强制、推荐、参考三大类。对于规约条目的延伸信息中，“说明”对内容做了引申和解释；“正例”提倡什么样的编码和实现方式；“反例”说明需要提防的雷区，以及真实的错误案例。

本手册的愿景是码出高效、码出质量。代码的字里行间流淌的是软件生命中的血液，质量的提升是尽可能少踩坑，杜绝踩重复的坑，切实提升质量意识。另外，现代软件架构都需要协同开发完成，高效协作即降低协同成本，提升沟通效率，所谓无规矩不成方圆，无规范不能协作。众所周知，制订交通法规表面上是要限制行车权，实际上是保障公众的人身安全。试想如果没有限速，没有红绿灯，谁还敢上路行驶。对软件来说，适当的规范和标准绝不是消灭代码内容的创造性、优雅性，而是限制过度个性化，以一种普遍认可的统一方式一起做事，提升协作效率。

2 编程规约

2.1 命名风格

1. 【强制】代码中的命名均不能以下划线或美元符号开始，也不能以下划线或美元符号结束。

   反例：_name / __name / $Object / name_ / name$ / Object$

2. 【强制】代码中的命名严禁使用拼音与英文混合的方式，更不允许直接使用中文的方式。

   说明：正确的英文拼写和语法可以让阅读者易于理解，避免歧义。注意：即使纯拼音命名方式也要避免采用。

   正例：beijing / junde / weijinke / youku 等国际通用的名称，可视同英文。

   反例：DaZhePromotion [打折] / getPingfenByName() [评分] / int 某变量 = 3

3. 【强制】类名使用UpperCamelCase风格，必须遵从驼峰形式，但以下情形例外：DO / BO / DTO / VO / AO

   正例：MarcoPolo / UserDO / XmlService / TcpUdpDeal / TaPromotion

   反例：macroPolo / UserDo / XMLService / TCPUDPDeal / TAPromotion

4. 【强制】方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase风格，必须遵从驼峰形式。

   正例：localValue / getHttpMessage() / inputUserId

5. 【强制】常量命名全部大写，单词间用下划线隔开，力求语义表达完整清楚，不要嫌名字长。

   正例：MAX_STOCK_COUNT

   反例：MAX_COUNT

6. 【强制】抽象类命名使用Abstract或Base开头；异常类命名使用Exception结尾；测试类命名以它要测试的类的名称开始，以Test结尾。

7. 【强制】中括号是数组类型的一部分，数组定义如下：String[] args;

   反例：使用 String args[]的方式来定义。

8. 【强制】POJO类中布尔类型的变量，都不要加is，否则部分框架解析会引起序列化错误。

   反例：定义为基本数据类型 Boolean isDeleted；的属性，它的方法也是 isDeleted()，RPC框架在反向解析的时候，“以为”对应的属性名称是deleted，导致属性获取不到，进而抛出异常。

9. 【强制】包名统一使用小写，点分隔符之间有且仅有一个自然语义的英语单词。包名统一使用单数形式，但是类名如果有复数含义，类名可以使用复数形式。

   正例：应用工具类包名为com.junde.open.util、类名为MessageUtils（此规则参考spring的框架结构）

10. 【强制】杜绝完全不规范的缩写，避免望文不知义。

    反例：AbstractClass“缩写”命名成AbsClass；condition“缩写”命名成 condi，此类随意缩写严重降低了代码的可阅读性。

11. 【推荐】如果使用到了设计模式，建议在类名中体现出具体模式。

    说明：将设计模式体现在名字中，有利于阅读者快速理解架构设计思想。

    正例：public class OrderFactory; public class LoginProxy; public class SystemObserver;

12. 【推荐】接口类中的方法和属性不要加任何修饰符号（public也不要加），保持代码的简洁性，并加上有效的Javadoc注释。尽量不要在接口里定义变量，如果一定要定义变量，肯定是与接口方法相关，并且是整个应用的基础常量。

    正例：接口方法签名：void f(); 接口基础常量表示：String COMPANY = "weijinke";

反例：接口方法定义：public abstract void f();

13. 接口和实现类的命名有两套规则：

A. 【强制】对于Service和DAO类，基于SOA的理念，暴露出来的服务一定是接口，内部的实现类用Impl的后缀与接口区别。

正例：CacheServiceImpl实现CacheService接口。

B. 【推荐】如果是形容能力的接口名称，取对应的形容词做接口名（通常是-able的形式）。

正例：AbstractTranslator实现Translatable。

14. 【参考】枚举类名建议带上Enum后缀，枚举成员名称需要全大写，单词间用下划线隔开。

说明：枚举其实就是特殊的常量类，且构造方法被默认强制是私有。

正例：枚举名字：DealStatusEnum，成员名称：SUCCESS / UNKOWN_REASON。

15. 【参考】各层命名规约：

A. Service/DAO层方法命名规约

- 获取单个对象的方法用get做前缀。
- 获取多个对象的方法用list做前缀。
- 获取统计值的方法用count做前缀。
- 插入的方法用save（推荐）或insert做前缀。
- 删除的方法用remove（推荐）或delete做前缀。
- 修改的方法用update做前缀。

B. 领域模型命名规约

- 数据对象：xxxDO，xxx即为数据表名。
- 数据传输对象：xxxDTO，xxx为业务领域相关的名称。
- 展示对象：xxxVO，xxx一般为网页名称。
- POJO是 DO/DTO/BO/VO的统称，禁止命名成 xxxPOJO。

2.2 常量定义

1. 【强制】不允许任何魔法值（即未经定义的常量）直接出现在代码中。

反例：String key = "Id#lend_" + tradeId; cache.put(key, value);

2. 【强制】long或者Long初始赋值时，必须使用大写的L，不能是小写的l，小写容易跟数字

1混淆，造成误解。

说明：Long a = 2l; 写的是数字的21，还是Long型的2?

3. 【推荐】不要使用一个常量类维护所有常量，应该按常量功能进行归类，分开维护。如：缓存相关的常量放在类：CacheConsts下；系统配置相关的常量放在类：ConfigConsts下。

说明：大而全的常量类，非得使用查找功能才能定位到修改的常量，不利于理解和维护。

4. 【推荐】常量的复用层次有五层：跨应用共享常量、应用内共享常量、子工程内共享常量、包内共享常量、类内共享常量。

A. 跨应用共享常量：放置在二方库中，通常是client.jar中的constant目录下。

B. 应用内共享常量：放置在一方库的modules中的constant目录下。

反例：易懂变量也要统一定义成应用内共享常量，两位工程师在两个类中分别定义了表示“是”的变量：

类 A中：public static final String YES = "yes";

类 B中：public static final String YES = "y";

A.YES.equals(B.YES)，预期是true，但实际返回为false，导致线上问题。

C. 子工程内部共享常量：即在当前子工程的constant目录下。

D. 包内共享常量：即在当前包下单独的constant目录下。

E. 类内共享常量：直接在类内部private static final定义。

5. 【推荐】如果变量值仅在一个范围内变化，且带有名称之外的延伸属性，定义为枚举类。下面正例中的数字就是延伸信息，表示星期几。

正例：public Enum { MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7); }

2.3 代码格式

1. 【强制】大括号的使用约定。如果是大括号内为空，则简洁地写成{}即可，不需要换行；如果是非空代码块则：

A. 左大括号前不换行。

B. 左大括号后换行。

C. 右大括号前换行。

D. 右大括号后还有else等代码则不换行；表示终止的右大括号后必须换行。

2. 【强制】左小括号和字符之间不出现空格；同样，右小括号和字符之间也不出现空格。详见第5条下方正例提示。

反例：if (空格a == b空格)

3. 【强制】if/for/while/switch/do等保留字与括号之间都必须加空格。

4. 【强制】任何二目、三目运算符的左右两边都需要加一个空格。

说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

5. 【强制】缩进采用4个空格，禁止使用tab字符。

说明：如果使用tab缩进，必须设置1个tab为4个空格。IDEA设置tab为4个空格时，请勿勾选Use tab character；而在eclipse中，必须勾选insert spaces for tabs。

正例：（涉及1-5点）

```
public static void main(String[] args) {
    // 缩进 4个空格
    String say = "hello";
    // 运算符的左右必须有一个空格
    int flag = 0;
    // 关键词if与括号之间必须有一个空格，括号内的f与左括号，0与右括号不需要空格
    if (flag == 0) {
        System.out.println(say);
    }
    // 左大括号前加空格且不换行；左大括号后换行
    if (flag == 1) {
        System.out.println("world");
    // 右大括号前换行，右大括号后有 else，不用换行
    } else {
        System.out.println("ok");
    // 在右大括号后直接结束，则必须换行
    }
}
```

6. 【强制】单行字符数限制不超过120个，超出需要换行，换行时遵循如下原则：

A. 第二行相对第一行缩进4个空格，从第三行开始，不再继续缩进，参考示例。

B. 运算符与下文一起换行。

C. 方法调用的点符号与下文一起换行。

D. 在多个参数超长，在逗号后换行。

E. 在括号前不要换行，见反例。

正例：

```
StringBuffer sb = new StringBuffer();
//超过120个字符的情况下，换行缩进4个空格，并且方法前的点符号一起换行
sb.append("zi").append("xin")...
        .append("huang")...
        .append("huang")...
        .append("huang");
```

反例：

```
StringBuffer sb = new StringBuffer();
//超过120个字符的情况下，不要在括号前换行
sb.append("zi").append("xin")...append
        ("huang");
//参数很多的方法调用可能超过120个字符，不要在逗号前换行
method(args1, args2, args3, ...
        , argsX);
```

7. 【强制】方法参数在定义和传入时，多个参数逗号后边必须加空格。

   正例：下例中实参的"a",后边必须要有一个空格。

   ```
   method("a", "b", "c");
   ```

8. 【强制】IDE的text file encoding设置为UTF-8; IDE中文件的换行符使用Unix格式，不要使用windows格式。

9. 【推荐】没有必要增加若干空格来使某一行的字符与上一行对应位置的字符对齐。

   正例：

   ```
   int a = 3;
   long b = 4L;
   float c = 5F;
   StringBuffer sb = new StringBuffer();
   ```

   说明：增加sb这个变量，如果需要对齐，则给a、b、c都要增加几个空格，在变量比较多的情况下，是一种累赘的事情。

10. 【推荐】方法体内的执行语句组、变量的定义语句组、不同的业务逻辑之间或者不同的语义之间插入一个空行。相同业务逻辑和语义之间不需要插入空行。

    说明：没有必要插入多个空行进行隔开。

2.4 OOP规约

1. 【强制】避免通过一个类的对象引用访问此类的静态变量或静态方法，无谓增加编译器解析成本，直接用类名来访问即可。

2. 【强制】所有的覆写方法，必须加@Override注解。

   说明：getObject()与get0bject()的问题。一个是字母的O，一个是数字的0，加@Override可以准确判断是否覆盖成功。另外，如果在抽象类中对方法签名进行修改，其实现类会马上编译报错。

3. 【强制】相同参数类型，相同业务含义，才可以使用Java的可变参数，避免使用Object。

   说明：可变参数必须放置在参数列表的最后。（提倡大家尽量不用可变参数编程）

   正例：public User getUsers(String type, Integer... ids) {...}

4. 【强制】外部正在调用或者二方库依赖的接口，不允许修改方法签名，避免对接口调用方产生影响。接口过时必须加@Deprecated注解，并清晰地说明采用的新接口或者新服务是什么。

5. 【强制】不能使用过时的类或方法。

   说明：java.net.URLDecoder中的方法decode(StringencodeStr)这个方法已经过时，应该使用双参数decode(String source, String encode)。接口提供方既然明确是过时接口，那么有义务同时提供新的接口；作为调用方来说，有义务去考证过时方法的新实现是什么。

6. 【强制】Object的equals方法容易抛空指针异常，应使用常量或确定有值的对象来调用equals。

   正例："test".equals(object);

   反例：object.equals("test");

   说明：推荐使用java.util.Objects#equals（JDK7引入的工具类）

7. 【强制】所有的相同类型的包装类对象之间值的比较，全部使用equals方法比较。

   说明：对于Integer var = ?在-128至127范围内的赋值，Integer对象是在IntegerCache.cache产生，会复用已有对象，这个区间内的Integer值可以直接使用==进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用equals方法进行判断。

8. 关于基本数据类型与包装数据类型的使用标准如下：

   A. 【强制】所有的POJO类属性必须使用包装数据类型。

   B. 【强制】RPC方法的返回值和参数必须使用包装数据类型。

   C. 【推荐】所有的局部变量使用基本数据类型。

      说明：POJO类属性没有初值是提醒使用者在需要使用时，必须自己显式地进行赋值，

任何NPE问题，或者入库检查，都由使用者来保证。

正例：数据库的查询结果可能是null，因为自动拆箱，用基本数据类型接收有NPE风险。

反例：比如显示成交总额涨跌情况，即正负x%，x为基本数据类型，调用的RPC服务，调用不成功时，返回的是默认值，页面显示：0%，这是不合理的，应该显示成中划线-。所以包装数据类型的null值，能够表示额外的信息，如：远程调用失败，异常退出。

9. 【强制】定义DO/DTO/VO等POJO类时，不要设定任何属性默认值。

反例：POJO类的gmtCreate默认值为new Date();但是这个属性在数据提取时并没有置入具体值，在更新其它字段时又附带更新了此字段，导致创建时间被修改成当前时间。

10. 【强制】序列化类新增属性时，请不要修改serialVersionUID字段，避免反序列失败；如果完全不兼容升级，避免反序列化混乱，那么请修改serialVersionUID值。

说明：注意serialVersionUID不一致会抛出序列化运行时异常。

11. 【强制】构造方法里面禁止加入任何业务逻辑，如果有初始化逻辑，请放在init方法中。

12. 【强制】POJO类必须写toString方法。使用IDE中的工具：source>generate toString时，如果继承了另一个POJO类，注意在前面加一下super.toString。

说明：在方法执行抛出异常时，可以直接调用POJO的toString()方法打印其属性值，便于排查问题。

13. 【推荐】使用索引访问用String的split方法得到的数组时，需做最后一个分隔符后有无内容的检查，否则会有抛IndexOutOfBoundsException的风险。

说明：

```
String str = "a,b,c,,";
String[] ary = str.split(",");
//预期大于 3，结果是 3
System.out.println(ary.length);
```

14. 【推荐】当一个类有多个构造方法，或者多个同名方法，这些方法应该按顺序放置在一起，便于阅读。

15. 【推荐】类内方法定义顺序依次是：公有方法或保护方法 > 私有方法 > getter/setter方法。

说明：公有方法是类的调用者和维护者最关心的方法，首屏展示最好；保护方法虽然只是子类关心，也可能是“模板设计模式”下的核心方法；而私有方法外部一般不需要特

别关心，是一个黑盒实现；因为方法信息价值较低，所有Service和DAO的getter/setter方法放在类体最后。

16. 【推荐】setter方法中，参数名称与类成员变量名称一致，this.成员名 = 参数名。在getter/setter方法中，不要增加业务逻辑，增加排查问题的难度。

反例：

```
public Integer getData() {
    if (true) {
        return this.data + 100;
    } else {
        return this.data - 100;
    }
}
```

17. 【推荐】循环体内，字符串的连接方式，使用StringBuilder的append方法进行扩展。

说明：反编译出的字节码文件显示每次循环都会new出一个StringBuilder对象，然后进行append操作，最后通过toString方法返回String对象，造成内存资源浪费。

反例：

```
String str = "start";
for (int i = 0; i < 100; i++) {
    str = str + "hello";
}
```

18. 【推荐】final可以声明类、成员变量、方法、以及本地变量，下列情况使用final关键字：

A. 不允许被继承的类，如：String类。

B. 不允许修改引用的域对象，如：POJO类的域变量。

C. 不允许被重写的方法，如：POJO类的setter方法。

D. 不允许运行过程中重新赋值的局部变量。

E. 避免上下文重复使用一个变量，使用final描述可以强制重新定义一个变量，方便更好地进行重构。

19. 【推荐】慎用Object的clone方法来拷贝对象。

说明：对象的clone方法默认是浅拷贝，若想实现深拷贝需要重写clone方法实现属性对象的拷贝。

20. 【推荐】类成员与方法访问控制从严：

A. 如果不允许外部直接通过new来创建对象，那么构造方法必须是private。

B. 工具类不允许有public或default构造方法。

C. 类非static成员变量并且与子类共享，必须是protected。

D. 类非static成员变量并且仅在本类使用，必须是private。

E. 类static成员变量如果仅在本类使用，必须是private。

F. 若是static成员变量，必须考虑是否为final。

G. 类成员方法只供类内部调用，必须是private。

H. 类成员方法只对继承类公开，那么限制为protected。

说明：任何类、方法、参数、变量，严控访问范围。过于宽泛的访问范围，不利于模块解耦。

思考：如果是一个private的方法，想删除就删除，可是一个public的service方法，或者一个public的成员变量，删除一下，不得手心冒点汗吗？变量像自己的小孩，尽量在自己的视线内，变量作用域太大，如果无限制地到处跑，那么你会担心的。

2.5 集合处理

1. 【强制】关于hashCode和equals的处理，遵循如下规则：

   A. 只要重写equals，就必须重写hashCode。

   B. 因为Set存储的是不重复的对象，依据hashCode和equals进行判断，所以Set存储的对象必须重写这两个方法。

   C. 如果自定义对象做为Map的键，那么必须重写hashCode和equals。

   说明：String重写了hashCode和equals方法，所以我们可以非常愉快地使用String对象作为key来使用。

2. 【强制】ArrayList的subList结果不可强转成ArrayList，否则会抛出ClassCastException异常：java.util.RandomAccessSubList cannot be cast to java.util.ArrayList ;

   说明：subList返回的是ArrayList的内部类SubList，并不是ArrayList ，而是ArrayList的一个视图，对于SubList子列表的所有操作最终会反映到原列表上。

3. 【强制】在subList场景中，高度注意对原集合元素个数的修改，会导致子列表的遍历、增加、删除均产生ConcurrentModificationException异常。

4. 【强制】使用集合转数组的方法，必须使用集合的toArray(T[] array)，传入的是类型完全一样的数组，大小就是list.size()。

   说明：使用toArray带参方法，入参分配的数组空间不够大时，toArray方法内部将重新分

配内存空间，并返回新数组地址；如果数组元素大于实际所需，下标为[list.size()]的数组元素将被置为null，其它数组元素保持原值，因此最好将方法入参数组大小定义与集合元素个数一致。

正例：（涉及1-5点）

```
List<String> list = new ArrayList<String>(2);
list.add("junde");
list.add("weijinke");
String[] array = new String[list.size()];
array = list.toArray(array);
```

反例：直接使用toArray无参方法存在问题，此方法返回值只能是Object[]类，若强转其它类型数组将出现ClassCastException错误。

5. 【强制】使用工具类Arrays.asList()把数组转换成集合时，不能使用其修改集合相关的方法，它的add/remove/clear方法会抛出UnsupportedOperationException异常。

说明：asList的返回对象是一个Arrays内部类，并没有实现集合的修改方法。Arrays.asList体现的是适配器模式，只是转换接口，后台的数据仍是数组。

```
String[] str = new String[] { "a", "b" };
List list = Arrays.asList(str);
```

第一种情况：list.add("c"); 运行时异常。

第二种情况：str[0] = "beijing"; 那么list.get(0)也会随之修改。

6. 【强制】泛型通配符<? extends T>来接收返回的数据，此写法的泛型集合不能使用add方法，而<? super T>不能使用get方法，做为接口调用赋值时易出错。

说明：扩展说一下PECS(Producer Extends Consumer Super)原则：1）频繁往外读取内容的，适合用上界Extends。2）经常往里插入的，适合用下界Super。

7. 【强制】不要在foreach循环里进行元素的remove/add操作。remove元素请使用Iterator方式，如果并发操作，需要对Iterator对象加锁。

正例：

```
Iterator<String> it = a.iterator();
while (it.hasNext()) {
    String temp = it.next();
    if (删除元素的条件) {
        it.remove();
    }
}
```

反例：

```
List<String> a = new ArrayList<String>();
a.add("1");
a.add("2");
for (String temp : a) {
    if ("1".equals(temp)) {
        a.remove(temp);
    }
}
```

说明：以上代码的执行结果肯定会出乎大家的意料，那么试一下把“1”换成“2”，会是同样的结果吗？

8. 【强制】在JDK7版本及以上，Comparator要满足如下三个条件，不然Arrays.sort，Collections.sort会报IllegalArgumentException异常。

   A. x，y的比较结果和y，x的比较结果相反。

   B. x>y，y>z，则 x>z。

   C. x=y，则 x，z比较结果和 y，z比较结果相同。

   反例：下例中没有处理相等的情况，实际使用中可能会出现异常：

```
new Comparator<Student>() {
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getId() > o2.getId() ? 1 : -1;
    }
};
```

9. 【推荐】集合初始化时，指定集合初始值大小。

   说明：HashMap使用HashMap(int initialCapacity)初始化。

   正例：initialCapacity = (需要存储的元素个数 / 负载因子) + 1。注意负载因子（即loader factor）默认为0.75，如果暂时无法确定初始值大小，请设置为16。

   反例：HashMap需要放置1024个元素，由于没有设置容量初始大小，随着元素不断增加，容量7次被迫扩大，resize需要重建hash表，严重影响性能。

10. 【推荐】使用entrySet遍历Map类集合KV，而不是keySet方式进行遍历。

    说明：keySet其实是遍历了2次，一次是转为Iterator对象，另一次是从hashMap中取出key所对应的value。而entrySet只是遍历了一次就把key和value都放到了entry中，效率更高。如果是JDK8，使用Map.foreach方法。

    正例：values()返回的是V值集合，是一个list集合对象；keySet()返回的是K值集合，是一个Set集合对象；entrySet()返回的是K-V值组合集合。

11. 【推荐】高度注意Map类集合K/V能不能存储null值的情况，如下表格：

集合类	Key	Value	Super	说明
Hashtable	不允许为null	不允许为null	Dictionary	线程安全
ConcurrentHashMap	不允许为null	不允许为null	AbstractMap	分段锁技术
TreeMap	不允许为null	允许为null	AbstractMap	线程不安全
HashMap	允许为null	允许为null	AbstractMap	线程不安全

反例：由于HashMap的干扰，很多人认为ConcurrentHashMap是可以置入null值，而事实上，存储null值时会抛出NPE异常。

12. 【参考】合理利用好集合的有序性(sort)和稳定性(order)，避免集合的无序性(unsort)和不稳定性(unorder)带来的负面影响。

说明：有序性是指遍历的结果是按某种比较规则依次排列的。稳定性指集合每次遍历的元素次序是一定的。如：ArrayList是order/unsort；HashMap是unorder/unsort；TreeSet是order/sort。

13. 【参考】利用 Set元素唯一的特性，可以快速对一个集合进行去重操作，避免使用List的contains方法进行遍历、对比、去重操作。

2.6 并发处理

1. 【强制】获取单例对象需要保证线程安全，其中的方法也要保证线程安全。

说明：资源驱动类、工具类、单例工厂类都需要注意。

2. 【强制】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。

正例：

```
public class TimerTaskThread extends Thread {
public TimerTaskThread() {
    super.setName("TimerTaskThread"); ...
}
```

3. 【强制】线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。

说明：使用线程池的好处是减少在创建和销毁线程上所花的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

4. 【强制】线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方式让写的人员更加明确线程池的运行规则，规避资源耗尽的风险。

说明：Executors返回的线程池对象的弊端如下：

A. FixedThreadPool和SingleThreadPool：

允许的请求队列长度为Integer.MAX_VALUE，可能会堆积大量的请求，从而导致OOM。

B. CachedThreadPool和ScheduledThreadPool：

允许的创建线程数量为Integer.MAX_VALUE，可能会创建大量的线程，从而导致OOM。

5. 【强制】SimpleDateFormat是线程不安全的类，一般不要定义为static变量，如果定义为static，必须加锁，或者使用DateUtils工具类。

正例：注意线程安全，使用DateUtils。亦推荐如下处理：

```
private static final ThreadLocal<DateFormat> df = new ThreadLocal<DateFormat>() {
    @Override
    protected DateFormat initialValue() {
        return new SimpleDateFormat("yyyy-MM-dd");
    }
};
```

说明：如果是JDK8的应用，可以使用Instant代替Date，LocalDateTime代替Calendar，DateTimeFormatter代替Simpledateformatter，官方给出的解释：simple beautiful strong immutable thread-safe。

6. 【强制】高并发时，同步调用应该去考量锁的性能损耗。能用无锁数据结构，就不要用锁；能锁区块，就不要锁整个方法体；能用对象锁，就不要用类锁。

说明：尽可能使加锁的代码块工作量尽可能的小，避免在锁代码块中调用RPC 方法。

7. 【强制】对多个资源、数据库表、对象同时加锁时，需要保持一致的加锁顺序，否则可能会造成死锁。

说明：线程一需要对表A、B、C依次全部加锁后才可以进行更新操作，那么线程二的加锁顺序也必须是A、B、C，否则可能出现死锁。

8. 【强制】并发修改同一记录时，避免更新丢失，需要加锁。要么在应用层加锁，要么在缓存加锁，要么在数据库层使用乐观锁，使用version作为更新依据。

说明：如果每次访问冲突概率小于20%，推荐使用乐观锁，否则使用悲观锁。乐观锁的重试次数不得小于 3次。

9. 【强制】多线程并行处理定时任务时，Timer运行多个TimeTask时，只要其中之一没有捕获抛出的异常，其它任务便会自动终止运行，使用ScheduledExecutorService则没有这个问题。

10. 【推荐】使用CountDownLatch进行异步转同步操作，每个线程退出前必须调用countDown方法，线程执行代码注意catch异常，确保countDown方法可以执行，避免主线程无法执行至await方法，直到超时才返回结果。

    说明：注意子线程抛出异常堆栈，不能在主线程try-catch到。

11. 【推荐】避免Random实例被多线程使用，虽然共享该实例是线程安全的，但会因竞争同一seed导致的性能下降。

    说明：Random实例包括java.util.Random的实例或者Math.random()的方式。

    正例：在 JDK7之后，可以直接使用API ThreadLocalRandom，而在JDK7之前，需要编码保证每个线程持有一个实例。

12. 【推荐】在并发场景下，通过双重检查锁（double-checked locking）实现延迟初始化的优化问题隐患（可参考The "Double-Checked Locking is Broken" Declaration），推荐问题解决方案中较为简单一种（适用于JDK5及以上版本），将目标属性声明为volatile型。

    反例：注意线程安全，使用DateUtils。亦推荐如下处理：

```
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) synchronized(this) {
            if (helper == null)
            helper = new Helper();
        }
        return helper;
    }
    // other functions and members...
}
```

13. 【参考】volatile解决多线程内存不可见问题。对于一写多读，是可以解决变量同步问题，但是如果多写，同样无法解决线程安全问题。如果是count++操作，使用如下类实现：AtomicInteger count = new AtomicInteger(); count.addAndGet(1);如果是JDK8，推荐使用LongAdder对象，比AtomicLong性能更好（减少乐观锁的重试次数）。

14. 【参考】HashMap在容量不够进行resize时由于高并发可能出现死链，导致CPU飙升，在开发过程中可以使用其它数据结构或加锁来规避此风险。

15. 【参考】ThreadLocal无法解决共享对象的更新问题，ThreadLocal对象建议使用static修饰。这个变量是针对一个线程内所有操作共有的，所以设置为静态变量，所有此类实例共享此静态变量 ，也就是说在类第一次被使用时装载，只分配一块存储空间，所有此类

的对象（只要是这个线程内定义的）都可以操控这个变量。

2.7 控制语句

1. 【强制】在一个 switch块内，每个case要么通过break/return等来终止，要么注释说明程序将继续执行到哪一个case为止；在一个switch块内，都必须包含一个default语句并且放在最后，即使它什么代码也没有。
2. 【强制】在if/else/for/while/do语句中必须使用大括号。即使只有一行代码，避免使用单行的形式：if (condition) statements;
3. 【推荐】SimpleDateFormat是线程不安全的类，一般不要定义为static变量，如果定义为static，必须加锁，或者使用DateUtils工具类。

```
if (condition) {
    ...
    return obj;
}
//接着写 else的业务逻辑代码
```

说明：如果非得使用if()...elseif()...else...方式表达逻辑，【强制】避免后续代码维护困难，请勿超过3层。

正例：逻辑上超过3 层的if-else代码可以使用卫语句，或者状态模式来实现。卫语句示例如下：

```
public void today() {
    if (isBusy()) {
        System.out.println(“change time.”);
        return;
    }
    if (isFree()) {
        System.out.println(“go to travel.”);
        return;
    }
    System.out.println(“stay at home to learn Java Coding Guideline.”);
    return;
}
```

4. 【推荐】除常用方法（如getXxx/isXxx）等外，不要在条件判断中执行其它复杂的语句，将复杂逻辑判断的结果赋值给一个有意义的布尔变量名，以提高可读性。

说明：很多if语句内的逻辑相当复杂，阅读者需要分析条件表达式的最终结果，才能明确什么样的条件执行什么样的语句，那么，如果阅读者分析逻辑表达式错误呢？

正例：

```
//伪代码如下
final boolean existed = (file.open(fileName, "w") != null) && (...) || (...);
if (existed) {
    ...
}
```

反例：

```
if ((file.open(fileName, "w") != null) && (...) || (...)) {
    ...
}
```

5. 【推荐】循环体中的语句要考量性能，以下操作尽量移至循环体外处理，如定义对象、变量、获取数据库连接，进行不必要的try-catch操作（这个try-catch是否可以移至循环体外）。
6. 【推荐】接口入参保护，这种场景常见的是用于做批量操作的接口。
7. 【参考】下列情形，需要进行参数校验：
   A. 调用频次低的方法。
   B. 执行时间开销很大的方法。此情形中，参数校验时间几乎可以忽略不计，但如果因为参数错误导致中间执行回退，或者错误，那得不偿失。
   C. 需要极高稳定性和可用性的方法。
   D. 对外提供的开放接口，不管是RPC/API/HTTP接口。
   E. 敏感权限入口。
8. 【参考】下列情形，不需要进行参数校验：
   A. 极有可能被循环调用的方法。但在方法说明里必须注明外部参数检查要求。
   B. 底层调用频度比较高的方法。毕竟是像纯净水过滤的最后一道，参数错误不太可能到底层才会暴露问题。一般DAO层与Service层都在同一个应用中，部署在同一台服务器中，所以DAO的参数校验可以省略。
   C. 被声明成private只会被自己代码所调用的方法，如果能够确定调用方法的代码传入参数已经做过检查或者肯定不会有问题，此时可以不校验参数。

2.8 注释规约

1. 【强制】类、类属性、类方法的注释必须使用Javadoc规范，使用/**内容*/格式，不得使用//xxx方式。

说明：在IDE编辑窗口中，Javadoc方式会提示相关注释，生成Javadoc可以正确输出相应注释；在IDE中，工程调用方法时，不进入方法即可悬浮提示方法、参数、返回值的意义，提高阅读效率。

2. 【强制】所有的抽象方法（包括接口中的方法）必须要用Javadoc注释、除了返回值、参数、异常说明外，还必须指出该方法做什么事情，实现什么功能。

   说明：对子类的实现要求，或者调用注意事项，请一并说明。

3. 【强制】所有的类都必须添加创建者和创建日期。

4. 【强制】方法内部单行注释，在被注释语句上方另起一行，使用//注释。方法内部多行注释使用/* */注释，注意与代码对齐。

5. 【强制】所有的枚举类型字段必须要有注释，说明每个数据项的用途。

6. 【推荐】与其“半吊子”英文来注释，不如用中文注释把问题说清楚。专有名词与关键字保持英文原文即可。

   反例：“TCP连接超时”解释成“传输控制协议连接超时”，理解反而费脑筋。

7. 【推荐】代码修改的同时，注释也要进行相应的修改，尤其是参数、返回值、异常、核心逻辑等的修改。

   说明：代码与注释更新不同步，就像路网与导航软件更新不同步一样，如果导航软件严重滞后，就失去了导航的意义。

8. 【参考】合理处理注释掉的代码。在上方详细说明，而不是简单的注释掉。如果无用，则删除。

   说明：代码被注释掉有两种可能性：1）后续会恢复此段代码逻辑。2）永久不用。前者如果没有备注信息，难以知晓注释动机。后者建议直接删掉（代码库保存了历史代码）。

9. 【参考】对于注释的要求：第一、能够准确反应设计思想和代码逻辑；第二、能够描述业务含义，使别的程序员能够迅速了解到代码背后的信息。完全没有注释的大段代码对于阅读者形同天书，注释是给自己看的，即使隔很长时间，也能清晰理解当时的思路；注释也是给继任者看的，使其能够快速接替自己的工作。

10. 【参考】好的命名、代码结构是自解释的，注释力求精简准确、表达到位。避免出现注释的一个极端：过多过滥的注释，代码的逻辑一旦修改，修改注释是相当大的负担。

    反例：

```
// put elephant into fridge
put(elephant, fridge);
```

方法名put，加上两个有意义的变量名elephant和fridge，已经说明了这是在干什么，语义清晰的代码不需要额外的注释。

11. 【参考】特殊注释标记，请注明标记人与标记时间。注意及时处理这些标记，通过标记扫描，经常清理此类标记。线上故障有时候就是来源于这些标记处的代码。

    A. 待办事宜（TODO）:（ 标记人，标记时间，[预计处理时间]）

       表示需要实现，但目前还未实现的功能。这实际上是一个Javadoc的标签，目前的Javadoc还没有实现，但已经被广泛使用。只能应用于类，接口和方法（因为它是一个Javadoc标签）。

    B. 错误，不能工作（FIXME）:（标记人，标记时间，[预计处理时间]）

       在注释中用FIXME标记某代码是错误的，而且不能工作，需要及时纠正的情况。

2.9 其它

1. 【强制】在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。

   说明：不要在方法体内定义：Pattern pattern = Pattern.compile(规则);

2. 【强制】velocity调用POJO类的属性时，建议直接使用属性名取值即可，模板引擎会自动按规范调用POJO的getXxx()，如果是boolean基本数据类型变量（boolean命名不需要加is前缀），会自动调用isXxx()方法。

   说明：注意如果是Boolean包装类对象，优先调用getXxx()的方法。

3. 【强制】后台输送给页面的变量必须加$!{var}——中间的感叹号。

   说明：如果var=null或者不存在，那么${var}会直接显示在页面上。

4. 【强制】注意Math.random()这个方法返回是double类型，注意取值的范围0≤x<1（能够取到零值，注意除零异常），如果想获取整数类型的随机数，不要将x放大10的若干倍然后取整，直接使用Random对象的nextInt或者nextLong方法。

5. 【强制】获取当前毫秒数System.currentTimeMillis(); 而不是new Date().getTime();

   说明：如果想获取更加精确的纳秒级时间值，使用System.nanoTime()的方式。在JDK8中，针对统计时间等场景，推荐使用Instant类。

6. 【推荐】不要在视图模板中加入任何复杂的逻辑。

   说明：根据MVC理论，视图的职责是展示，不要抢模型和控制器的活。

7. 【推荐】任何数据结构的构造或初始化，都应指定大小，避免数据结构无限增长吃光内

存。

8. 【推荐】对于“明确停止使用的代码和配置”，如方法、变量、类、配置文件、动态配置属性等要坚决从程序中清理出去，避免造成过多垃圾。

3 异常日志

3.1 异常处理

1. 【强制】Java 类库中定义的一类RuntimeException可以通过预先检查进行规避，而不应该通过catch来处理，比如：IndexOutOfBoundsException，NullPointerException等等。

   说明：无法通过预检查的异常除外，如在解析一个外部传来的字符串形式数字时，通过catch NumberFormatException来实现。

   正例：if (obj != null) {...}

   反例：try { obj.method() } catch (NullPointerException e) {...}
2. 【强制】异常不要用来做流程控制，条件控制，因为异常的处理效率比条件分支低。
3. 【强制】对大段代码进行try-catch，这是不负责任的表现。catch时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。对于非稳定代码的catch尽可能进行区分异常类型，再做对应的异常处理。
4. 【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的内容。
5. 【强制】有try块放到了事务代码中，catch异常后，如果需要回滚事务，一定要注意手动回滚事务。
6. 【强制】finally块必须对资源对象、流对象进行关闭，有异常也要做try-catch。

   说明：如果JDK7及以上，可以使用try-with-resources方式。
7. 【强制】不能在 finally块中使用return，finally块中的return返回后方法结束执行，不会再执行try块中的return语句。
8. 【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。

   说明：如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。
9. 【推荐】方法的返回值可以为null，不强制返回空集合，或者空对象等，必须添加注释充分说明什么情况下会返回null值。调用方需要进行null判断防止NPE问题。

说明：本手册明确防止NPE是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回null的情况。

10. 【推荐】防止NPE，是程序员的基本修养，注意NPE产生的场景：

    A. 返回类型为基本数据类型，return包装数据类型的对象时，自动拆箱有可能产生NPE。

       反例：public int f() { return Integer对象}， 如果为null，自动解箱抛NPE。

    B. 数据库的查询结果可能为null。

    C. 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。

    D. 远程调用返回对象时，一律要求进行空指针判断，防止NPE。

    E. 对于Session中获取的数据，建议NPE检查，避免空指针。

    F. 级联调用obj.getA().getB().getC()；一连串调用，易产生NPE。

    正例：使用JDK8的Optional类来防止NPE问题。

11. 【推荐】定义时区分unchecked/checked异常，避免直接抛出newRuntimeException()，更不允许抛出Exception或者Throwable，应使用有业务含义的自定义异常。推荐业界已定义过的自定义异常，如：DAOException / ServiceException等。

12. 【参考】避免出现重复的代码（Don’t Repeat Yourself），即DRY原则。

    说明：随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是共用模块。

    正例：一个类中有多个public方法，都需要进行数行相同的参数校验操作，这个时候请抽取：private boolean checkParam(DTO dto) {...}

3.2 日志规约

1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的API，而应依赖使用日志框架SLF4J中的API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

2. 【强制】日志文件推荐至少保存15天，因为有些异常具备以“周”为频次发生的特点。

3. 【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：

appName_logType_logName.log 。 logType ： 日 志 类 型 ， 推 荐 分 类 有 stats/desc/monitor/visit等；logName：日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。

正 例 ： mppserver 应 用 中 单 独 监 控 时 区 转 换 异 常 ， 如 ： mppserver_monitor_timeZoneConvert.log

说明：推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。

4. 【强制】对trace/debug/info级别的日志输出，必须使用条件输出形式或者使用占位符的方式。

说明：logger.debug("Processing trade with id: " + id + " symbol: " + symbol); 如果日志级别是warn，上述日志不会打印，但是会执行字符串拼接操作，如果symbol是对象，会执行toString()方法，浪费了系统资源，执行了上述操作，最终日志却没有打印。

正例：（条件）

```
if (logger.isDebugEnabled()) {
    logger.debug("Processing trade with id: " + id + " symbol: " + symbol);
}
```

正例：（占位符）

```
logger.debug("Processing trade with id: {} symbol : {} ", id, symbol);
```

5. 【强制】避免重复打印日志，浪费磁盘空间，务必在log4j.xml中设置additivity=false。

正例：<logger name="com.weijinke.dubbo.config" additivity="false">

6. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字throws往上抛出。

正例：logger.error(各类参数或者对象.toString + "_" + e.getMessage(), e);

7. 【推荐】谨慎地记录日志。生产环境禁止输出debug日志；有选择地输出info日志；如果使用warn来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。

说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。

思考：这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？

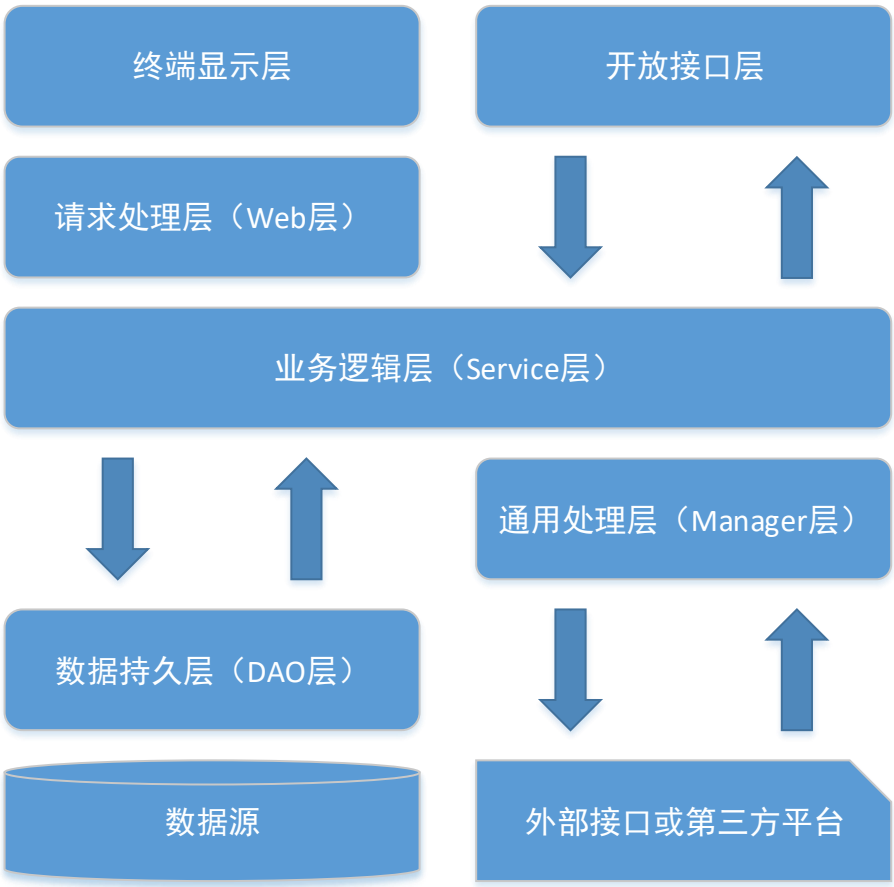
8. 【参考】可以使用warn日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，error级别只记录系统逻辑出错、异常等重要的错误信息。

如非必要，请不要在此场景打出error级别。

4 工程结构

4.1 应用分层

1. 【推荐】图中默认上层依赖于下层，箭头关系表示可直接依赖，如：开放接口层可以依赖于Web层，也可以直接依赖于Service层，依此类推：



A. 开放接口层：可直接封装Service方法暴露成RPC接口；通过Web封装成http接口；进行网关安全控制、流量控制等。

B. 终端显示层：各个端的模板渲染并执行显示的层。当前主要是velocity渲染，JS渲染，JSP渲染，移动端展示等。

C. Web层：主要是对访问控制进行转发，各类基本参数校验，或者不复用的业务简单处理等。

D. Service层：相对具体的业务逻辑服务层。

E. Manager层：通用业务处理层，它有如下特征：

- 对第三方平台封装的层，预处理返回结果及转化异常信息；
- 对Service层通用能力的下沉，如缓存方案、中间件通用处理；
- 与DAO层交互，对多个DAO的组合复用。

F. DAO层：数据访问层，与底层MySQL、Oracle、Hbase等进行数据交互。

G. 外部接口或第三方平台：包括其它部门RPC开放接口，基础平台，其它公司的HTTP接口。

2. 【参考】（分层异常处理规约）在DAO层，产生的异常类型有很多，无法用细粒度的异常进行catch，使用catch(Exception e)方式，并throw new DAOException(e)，不需要打印日志，因为日志在Manager/Service层一定需要捕获并打到日志文件中去，如果同台服务器再打日志，浪费性能和存储。在Service层出现异常时，必须记录出错日志到磁盘，尽可能带上参数信息，相当于保护案发现场。如果Manager层与Service同机部署，日志方式与DAO层处理一致，如果是单独部署，则采用与Service一致的处理方式。Web层绝不应该继续往上抛异常，因为已经处于顶层，无继续处理异常的方式，如果意识到这个异常将导致页面无法正常渲染，那么就应该直接跳转到友好错误页面，加上友好的错误提示信息。开放接口层要将异常处理成错误码和错误信息方式返回。

3. 【参考】分层领域模型规约：

A. DO（Data Object）：与数据库表结构一一对应，通过DAO层向上传输数据源对象。

B. DTO（Data Transfer Object）：数据传输对象，Service和Manager向外传输的对象。

C. BO（Business Object）：业务对象。可以由Service层输出的封装业务逻辑的对象。

D. Query：数据查询对象，各层接收上层的查询请求。注：超过2个参数的查询封装，禁止使用Map类来传输。

E. VO（View Object）：显示层对象，通常是Web向模板渲染引擎层传输的对象。

4.2 二方库依赖

1. 【强制】定义GAV遵从以下规则：

A. GroupID格式：com.{公司/BU}.业务线.[子业务线]，最多4级。

说明：{公司/BU}例如：junde/weijinke等BU一级；子业务线可选。

正例：com.junde.fund或com.weijinke.dubbo.register

B. ArtifactID格式：产品线名-模块名。语义不重复不遗漏。

正例：dubbo-client / fastjson-api / jstorm-tool

C. Version：详细规定参考下方。

2. 【强制】二方库版本号命名方式：主版本号.次版本号.修订号

A. 主版本号：当做了不兼容的API 修改，或者增加了能改变产品方向的新功能。

B. 次版本号：当做了向下兼容的功能性新增（新增类、接口等）。

C. 修订号：修复bug，没有修改方法签名的功能加强，保持API 兼容性。

说明：注意：起始版本号必须为：1.0.0，而不是0.0.1。正式发布的类库必须先去中央仓库进行查证，使版本号有延续性，正式版本号不允许覆盖升级。如当前版本：1.3.3，那么下一个合理的版本号：1.3.4 或1.4.0或2.0.0。

3. 【强制】线上应用不要依赖SNAPSHOT版本（安全包除外）。

说明：不依赖SNAPSHOT版本是保证应用发布的幂等性。另外，也可以加快编译时的打包构建。

4. 【强制】二方库的新增或升级，保持除功能点之外的其它jar包仲裁结果不变。如果有改变，必须明确评估和验证，建议进行dependency:resolve前后信息比对，如果仲裁结果完全不一致，那么通过dependency:tree命令，找出差异点，进行<excludes>排除jar包。

5. 【强制】二方库里可以定义枚举类型，参数可以使用枚举类型，但是接口返回值不允许使用枚举类型或者包含枚举类型的POJO对象。

6. 【强制】依赖于一个二方库群时，必须定义一个统一的版本变量，避免版本号不一致。

说明：依赖springframework-core,-context,-beans，它们都是同一个版本，可以定义一个变量来保存版本：${spring.version}，定义依赖的时候，引用该版本。

7. 【强制】禁止在子项目的pom依赖中出现相同的GroupId，相同的ArtifactId。

说明：在本地调试时会使用各子项目指定的版本号，但是合并成一个war，只能有一个版本号出现在最后的lib目录中。可能出现线下调试是正确的，发布到线上却出故障的问题。

8. 【推荐】所有pom文件中的依赖声明放在<dependencies>语句块中，所有版本仲裁放在<dependencyManagement>语句块中。

说明：<dependencyManagement>里只是声明版本，并不实现引入，因此子项目需要显式的声明依赖，version和scope都读取自父pom。而<dependencies>所有声明在主pom的<dependencies>里的依赖都会自动引入，并默认被所有的子项目继承。

9. 【推荐】二方库不要有配置项，最低限度不要再增加配置项。

10. 【参考】为避免应用二方库的依赖冲突问题，二方库发布者应当遵循以下原则：

    A. 精简可控原则。移除一切不必要的API和依赖，只包含Service API、必要的领域模型对象、Utils类、常量、枚举等。如果依赖其它二方库，尽量是provided引入，让二方库使用者去依赖具体版本号；无log具体实现，只依赖日志框架。

    B. 稳定可追溯原则。每个版本的变化应该被记录，二方库由谁维护，源码在哪里，都需要能方便查到。除非用户主动升级版本，否则公共二方库的行为不应该发生变化。

4.3 服务器

1. 【推荐】高并发服务器建议调小TCP协议的time_wait超时时间。

   说明：操作系统默认240秒后，才会关闭处于time_wait状态的连接，在高并发访问下，服务器端会因为处于time_wait的连接数太多，可能无法建立新的连接，所以需要在服务器上调小此等待值。

   正例：在linux服务器上请通过变更/etc/sysctl.conf文件去修改该缺省值（秒）：net.ipv4.tcp_fin_timeout = 30

2. 【推荐】调大服务器所支持的最大文件句柄数（File Descriptor，简写为fd）。

   说明：主流操作系统的设计是将TCP/UDP连接采用与文件一样的方式去管理，即一个连接对应于一个fd。主流的linux服务器默认所支持最大fd数量为1024，当并发连接数很大时就容易因为fd不足而出现“open too many files”错误，导致新的连接无法建立。 建议将linux服务器所支持的最大句柄数调高数倍（与服务器的内存数量相关）。

3. 【推荐】给JVM设置-XX:+HeapDumpOnOutOfMemoryError参数，让JVM碰到OOM场景时输出dump信息。

   说明：OOM的发生是有概率的，甚至有规律地相隔数月才出现一例，出现时的现场信息对查错非常有价值。

4. 【参考】服务器内部重定向使用forward；外部重定向地址使用URL拼装工具类来生成，否则会带来URL维护不一致的问题和潜在的安全风险。

5 安全规约

1. 【强制】隶属于用户个人的页面或者功能必须进行权限控制校验。

   说明：防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容、修改他人的订单等。

2. 【强制】用户敏感数据禁止直接展示，必须对展示数据进行脱敏处理。

   说明：查看个人手机号码会显示成：158****9119，隐藏中间4位，防止隐私泄露。

3. 【强制】用户输入的SQL参数严格使用参数绑定或者METADATA字段值限定，防止SQL注入，禁止字符串拼接SQL访问数据库。

4. 【强制】用户请求传入的任何参数必须做有效性验证。

   说明：忽略参数校验可能导致：

   A. page size过大导致内存溢出

   B. 恶意order by导致数据库慢查询

   C. 任意重定向

   D. SQL注入

   E. 反序列化注入

   F. 正则输入源串拒绝服务ReDoS

      说明：Java代码用正则来验证客户端的输入，有些正则写法验证普通用户输入没有问题，但是如果攻击人员使用的是特殊构造的字符串来验证，有可能导致死循环的结果。

5. 【强制】禁止向HTML页面输出未经安全过滤或未正确转义的用户数据。

6. 【强制】表单、AJAX提交必须执行CSRF安全过滤。

   说明：CSRF（Cross-site request forgery）跨站请求伪造是一类常见编程漏洞。对于存在CSRF漏洞的应用/网站，攻击者可以事先构造好URL，只要受害者用户一访问，后台便在用户不知情的情况下对数据库中用户参数进行相应修改。

7. 【强制】在使用平台资源，譬如短信、邮件、电话、下单、支付，必须实现正确的防重放限制，如数量限制、疲劳度控制、验证码校验，避免被滥刷、资损。

   说明：如注册时发送验证码到手机，如果没有限制次数和频率，那么可以利用此功能骚扰到其他用户，并造成短信平台资源浪费。

8. 【推荐】发贴、评论、发送即时消息等用户生成内容的场景必须实现防刷、文本内容违禁词过滤等风控策略。