云课堂服务端开发规约（内容主要基于阿里巴巴Java开发手册）

一、编程规约

(一)命名规约

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。

反例:

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

正例:

```
alibaba / taobao / youku / hangzhou
```

等国际通用的名称，可视同英文。

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：（领域模型的相关命名） DTO / VO等。

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。

[特例]对于Logger对象，统一使用小写log

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 UTest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：请勿使用 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 isSuccess；的属性，它的方法也是 isSuccess()，RPC框架在反向解析的时候，“以为”对应的属性名称是 success，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.alibaba.open.util、类名为 MessageUtils（此规则参考spring的框架结构），util类统一使用Utils结尾

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 ResourceObserver;
```

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 = "alibaba";
```

反例：接口方法定义：

```
public abstract void f();
```

说明：JDK8中接口允许有默认实现，那么这个 default方法，是对所有实现类都有价值的默认实现。

13.接口和实现类的命名有两套规则：

- 1）【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。

- 2）【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。Domain定义相应的Constants，将可枚举的字段全部列出来。 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

- 1） 获取单个对象的方法用 get做前缀。
- 2） 获取多个对象的方法用 list做前缀。
- 3） 获取分页的方法用 page做前缀。
- 4） 获取统计值的方法用 count做前缀。
- 5） 插入的方法用 save（推荐）或 insert做前缀。
- 6） 删除的方法用 remove（推荐）或 delete做前缀。
- 7） 修改的方法用 update做前缀。

B) 领域模型命名规约

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

(二)常量定义

1.【强制】不允许出现任何魔法值（即未经定义的常量）直接出现在代码中。

反例：

```
String key="Id#taobao_"+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.【参考】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。

- 1） 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。
- 2） 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，两位攻城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，导致产生线上问题。

- 3） 子工程内部共享常量：即在当前子工程的 constant目录下。
- 4） 包内共享常量：即在当前包下单独的 constant目录下。
- 5） 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用 Enum类。如果还带有名称之外的延伸属性，必须 使用 Enum类，下面正例中的数字就是延伸信息，表示星期几。

正例：

```
publicEnum{MONDAY(1),TUESDAY(2),WEDNESDAY(3),THURSDAY(4),FRIDAY(5),SATURDAY(6),
SUNDAY(7);
```

(三)格式规约

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：

- 1） 左大括号前不换行。
- 2） 左大括号后换行。
- 3） 右大括号前换行。
- 4） 右大括号后还有else等代码则不换行；表示终止右大括号后必须换行。
- 2.【强制】 左括号和后一个字符之间不出现空格；同样，右括号和前一个字符之间也不出现空格。详见第 5条下方正例提示。

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

4.【强制】任何运算符左右必须加一个空格。 说明：运算符包括赋值运算符=、逻辑运算符&&、加减乘除符号、三目运行符等。

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：

- 1） 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。
- 2） 运算符与下文一起换行。
- 3） 方法调用的点符号与下文一起换行。
- 4） 在多个参数超长，逗号后进行换行。
- 5） 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的相应字符对齐。

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：没有必要插入多行空格进行隔开。

(四)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。

反例：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。（提倡同学们尽量不用可变参数编程）

正例：

```
public User getUsers(String type, Integer... ids)
```

4.【强制】对外暴露的接口签名，原则上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之间的赋值，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.【强制】关于基本数据类型与包装数据类型的使用标准如下：

- 1） 所有的 POJO类属性必须使用包装数据类型。
- 2） RPC方法的返回值和参数必须使用包装数据类型。
- 3） 所有的局部变量【推荐】使用基本数据类型。
- 4） 在枚举类里面，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。实现 InitializingBean接口。

12.【推荐】POJO类必须写 toString方法。使用 IDE的中工具：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，尽量不要增加业务逻辑，增加排查问题的难度。

反例：

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

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

反例：

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

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。

18.【推荐】final可提高程序响应效率，声明成 final的情况：

- 1） 不需要重新赋值的变量，包括类属性、局部变量。
- 2） 对象参数前加 final，表示不允许修改引用的指向。
- 3） 类方法确定不允许被重写。

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：

- 1） 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。
- 2） 工具类不允许有 public或 default构造方法。
- 3） 类非 static成员变量并且与子类共享，必须是 protected。
- 4） 类非 static成员变量并且仅在本类使用，必须是 private。
- 5） 类 static成员变量如果仅在本类使用，必须是 private。
- 6） 若是 static成员变量，必须考虑是否为 final。
- 7） 类成员方法只供类内部调用，必须是 private。
- 8） 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，如果无限制的到处跑，那么你会担心的。

(五)集合处理

1.【强制】关于 hashCode和 equals的处理，遵循如下规则：

- 1） 只要重写 equals，就必须重写 hashCode。
- 2） 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。
- 3） 如果自定义对象做为 Map的键，那么必须重写 hashCode和 equals。

正例：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()。

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。

正例：

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

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。

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]= "gujin"; 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。

说明：苹果装箱后返回一个<? extends Fruits>对象，此对象就不能往里加任何水果，包括 苹果。

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。

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?

正例：

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

8.【强制】在 JDK7版本以上，Comparator要满足自反性，传递性，对称性，不然 Arrays.sort，Collections.sort会报 IllegalArgumentException异常。 说明：

- 1） 自反性：x，y的比较结果和 y，x的比较结果相反。
- 2） 传递性：x>y,y>z,则 x>z。
- 3） 对称性：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，尽量指定集合初始值大小。 说明：ArrayList尽量使用 ArrayList(int initialCapacity) 初始化。

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，注意存储 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：稳定性指集合每次遍历的元素次序是一定的。有序性是指遍历的结果是按某种比较规则 依次排列的。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。

(六)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.【推荐】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 的处理方式让写的同学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：

- 1）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。
- 2）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：simplebeautifulstrong 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。

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，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 countDown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()实例。

正例：在 JDK7之后，可以直接使用 API ThreadLocalRandom，在 JDK7之前，可以做到每个 线程一个实例。

12.【推荐】通过双重检查锁（double-checked locking）（在并发场景）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：

```
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 开发过程中注意规避此风险。

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。

(七)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 下面的形式：if (condition) statements;

3.【推荐】推荐尽量少用 else， if-else的方式可以改写成：

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

说明：如果非得使用 if()...else if()...else...方式表达逻辑，【强制】请勿超过 3层， 超过请使用状态设计模式。

正例：逻辑上超过 3 层的 if-else代码可以使用switch语句，或者状态模式来实现。

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
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。

- 1） Dao不应该做参数验证。
- 2） private方法可以不做参数验证。
- 3） 对外暴露的接口 / 方法必须做参数验证。

7.【参考】方法中需要进行参数校验的场景：

- 1） 调用频次低的方法。
- 2） 执行时间开销很大的方法，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。

- 3） 需要极高稳定性和可用性的方法。
- 4） 对外提供的开放接口，不管是 RPC/API/HTTP接口。
- 5） 敏感权限入口。

8.【参考】方法中不需要参数校验的场景：

- 1） 极有可能被循环调用的方法，不建议对参数进行校验。但在方法说明里必须注明外部参数检查。
- 2） 底层的方法调用频度都比较高，一般不校验。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。
- 3） 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。

(八)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信息。

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.【推荐】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】注释掉的代码尽量要配合说明，而不是简单的注释掉。

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（代码仓库保存了历史代码）。

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。

1） 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）。

2） 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。

(九)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，用 System.nanoTime()。在 JDK8中，针对统计 时间等场景，推荐使用 Instant类。

6.【推荐】尽量不要在 vm中加入变量声明、逻辑运算符，更不要在 vm模板中加入任何复杂的逻 辑。

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。

二、异常日志

(一)异常处理

1.【强制】不要捕获 Java类库中定义的继承自 RuntimeException的运行时异常类，如：IndexOutOfBoundsException / NullPointerException，这类异常由程序员预检查 来规避，保证程序健壮性。

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.【推荐】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.【推荐】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，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：本规约明确防止 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，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：

- 1） 返回类型为包装数据类型，有可能是 null，返回 int值时注意判空。

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

- 2） 数据库的查询结果可能为 null。
- 3） 集合里的元素即使 isNotEmpty，取出的数据元素也可能为 null。
- 4） 远程调用返回对象，一律要求进行 NPE判断。
- 5） 对于 Session中获取的数据，建议 NPE检查，避免空指针。
- 6） 级联调用 obj.getA().getB().getC()；一连串调用，易产生 NPE。

11.【推荐】在代码中使用“抛异常”还是“返回错误码”，对于公司外的 http/api开放接口必须 使用“错误码”；而应用内部推荐异常抛出；跨应用间 RPC调用优先考虑使用 Result方式，封 装 isSuccess、“错误码”、“错误简短信息”。

说明：关于 RPC方法返回方式使用 Result方式的理由：

- 1）使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。
- 2）如果不加栈信息，只是 new自定义异常，加入自己的理解的 error message，对于调用端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输的性能损耗也是问题。

12.【推荐】定义时区分 unchecked/checked 异常，避免直接使用 RuntimeException抛出， 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。

13.【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){...}
```

(二)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.【参考】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.【推荐】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，错误日志和业务日志尽量分开存放，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.【推荐】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.taobao.dubbo.config" additivity="false">
```

6.【推荐】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，那么往上 抛。

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

7.【推荐】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。

8.【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。记录日志时请 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？

三、数据库设计规范

(一)基础规范

1. 命名规范：库名、表名、列名均使用小写+下划线分隔，命名要见名知义，编码使用无明确含义的命名
2. 存储引擎使用innoDB
3. 字符集使用utf8,在涉及表情符号时，可使用utf8mb4
4. 禁止使用视图、触发器、存储过程
5. 避免在数据库中存储大文件，比如照片

(二)表设计规范

1. 表必须有主键，建议使用整型作为主键
2. 禁止使用外键，表之间的关联性和完整性通过应用层控制
3. 表在设计之初，应考虑到大致的数量级，若表记录数低于1000W，尽量使用单表，不建议分表
4. 建议将大字段、访问频率低及不需要筛选的字段拆分到拓展表中（做好垂直拆分）
5. 控制单实例中表的总数（一个微服务实例不要超过1000个）、分表个数控制在1024以内

(三)列设计规范

1. 合理使用字段类型，如tinyint、int、bigint
2. 所有字段应定义为NOT NULL并设置默认值，存储NULL需要更多的空间，并且使得索引和统计变得更复杂
3. 使用varchar(20)存储手机号，不要使用整数
4. 使用INT UNSIGNED存储IPv4，不要用char(15)

(四)索引规范

1. 唯一索引使用uniq_[字段名]来命名
2. 非唯一索引使用idx_[字段名]来命名
3. 不建议在频繁更新的字段上建立索引
4. 非必要不要进行JOIN查询，如果要进行JOIN查询，被JOIN的字段必须类型相同，并建立索引
5. 单张表索引数量建议控制在5个以内,索引过多，不仅会导致插入更新性能下降，还可能导致MySQL使用错误的索引，可在语句中加上force index来强制使用某个索引
6. 组合索引字段数不建议超过5个
7. 理解组合索引最左前缀原则，避免重复建设索引，如果建立了(a,b,c)，相当于建立了(a), (a,b), (a,b,c)

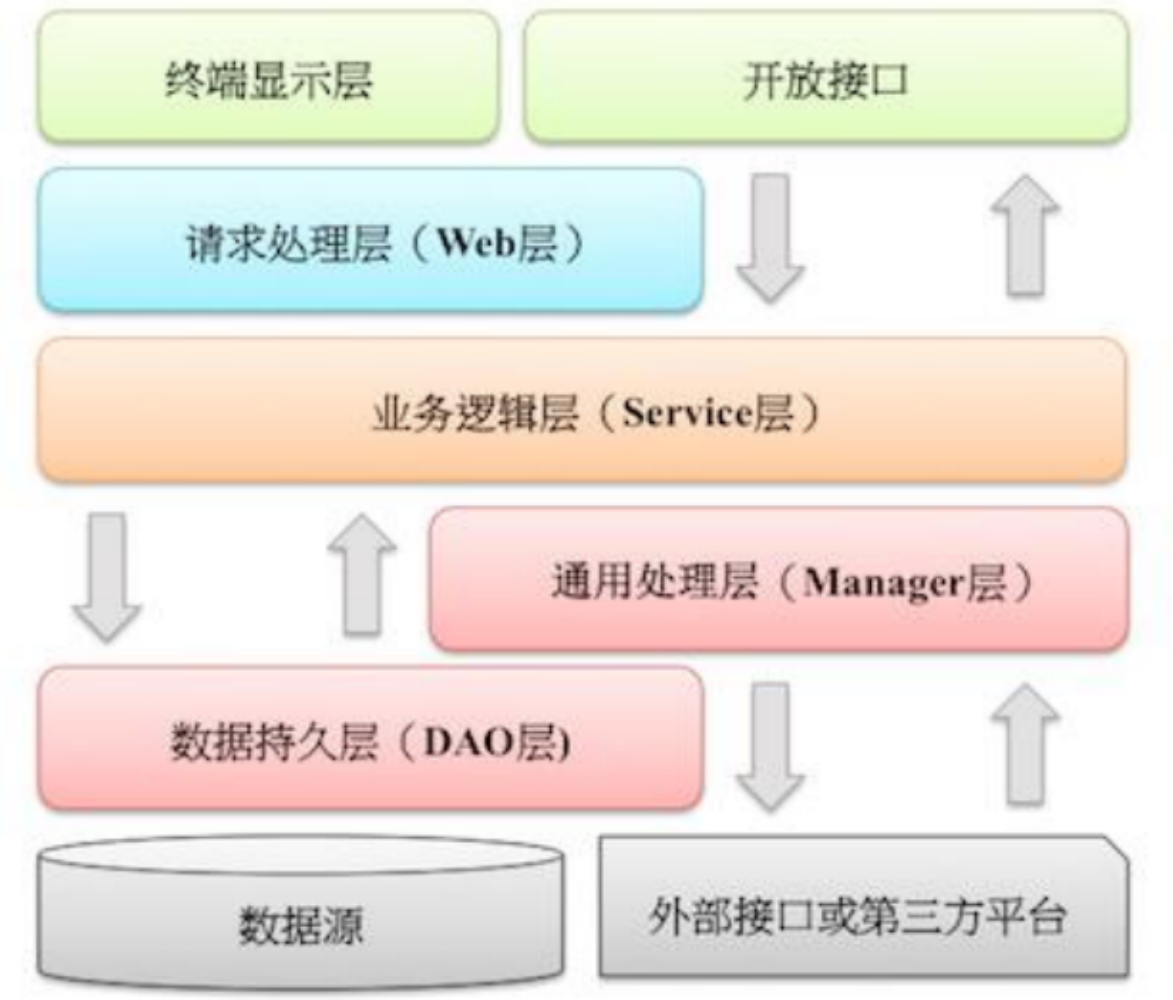
(五)sql规范

1. 禁止使用select *，只获取必要字段，select *会增加cpu/io/内存/带宽的消耗，指定字段能有效利用索引覆盖指定字段查询，在表结构变更时，能保证对应用程序无影响
2. insert必须指定字段，禁止使用insert into T values()，指定字段插入，在表结构变更时，能保证对应用程序无影响
3. 隐式类型转换会使索引失效，导致全表扫描
4. 禁止在where条件列使用函数或者表达式，导致不能命中索引，全表扫描
5. 禁止负向查询以及%开头的模糊查询，导致不能命中索引，全表扫描
6. 避免直接返回大结果集造成内存溢出，可采用分段和游标方式
7. 返回结果集时尽量使用limit分页显示
8. 避免出现较大的limit和offset值
9. 使用group by或order by的语句，即使用了limit offset，如果没有合适的索引做排序操作，也会遍历所有满足where条件的结果。
10. 大表扫描操作尽量放到镜像库上做
11. 禁止大表JOIN和子查询
12. 同一个字段上的OR必须改写问IN，IN的值必须少于50个
13. 应用程序必须捕获SQL异常，方便定位线上问题

四、工程规约

(一)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：



- 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。
- 终端显示层：各个端的模板渲染并执行显示层。当前主要是 velocity渲染，JS渲染，JSP渲染，移动端展示层等。
- 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。
- Service层：相对具体的业务逻辑服务层。
- Manager层：通用业务处理层，它有如下特征：

1. 对第三方平台封装的层，预处理返回结果及转化异常信息；
2. 对 Service层通用能力的下沉，如缓存方案、中间件通用处理；
3. 与 DAO层交互，对 DAO的业务通用能力的封装。

- DAO层：数据访问层，与底层 MySQL、Oracle、Hbase进行数据交互。
- 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，无法用细粒度异常进行catch，使用catch(Exceptione)方式，并thrownewDAOException(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，必须记录日志信息到磁盘，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，尽量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：

- 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。
- DTO（Data Transfer Object）： 数据传输对象，Service和 Manager向外传输的对象。
- BO（Business Object）： 业务对象。可以由 Service层输出的封装业务逻辑的对象。
- 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。
- 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。

(二)二方库规约

1.【强制】定义 GAV遵从以下规则： 1） 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：alibaba/taobao/tmall/aliexpress等 BU一级；子业务线可选。

正例：

```
com.taobao.jstorm 或 com.alibaba.dubbo.register
```

2） ArtifactID格式：产品线名-模块名。语义不重复不遗漏，先到仓库中心去查证一下。

正例：

```
dubbo-client / fastjson-api / jstorm-tool
```

3） 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

- 1） 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。
- 2） 次版本号：当做了向下兼容的功能性新增（新增类、接口等）。
- 3） 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：起始版本号必须为：1.0.0，而不是 0.0.1

3.【强制】线上应用不要依赖 SNAPSHOT版本（安全包除外）；正式发布的类库必须使用 RELEASE 版本号升级+1的方式，且版本号不允许覆盖升级，必须去中央仓库进行查证。 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。返回值使用具体的值，同时提供方法，能够根据值获取枚举类型。

6.【强制】依赖于一个二方库群时，必须定义一个统一版本变量，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，但是不同的 Version。

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。曾经出现过线下调试是正确的，发布到线上出故障的先例。

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

说明：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。而所有声明在主 pom的 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】二方库尽量不要有配置项，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：

- 1）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。
- 2）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。

(三)服务器规约

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很 容易因为 fd不足而出现“opentoomanyfiles”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。

五、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。

2.【强制】用户敏感数据禁止直接展示，必须对展示数据脱敏。

说明：查看个人手机号码会显示成:1589119，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：

- page size过大导致内存溢出
- 恶意 order by导致数据库慢查询
- 任意重定向
- SQL注入
- 反序列化注入
- 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效果。

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它用户，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。