**民生2020新核心开发规范**

目录

[一、工程结构 2](#_Toc50216590)

[1、DDD-领域模型 2](#_Toc50216591)

[2、DDD-分层详解 3](#_Toc50216592)

[3、DDD-调用规范 3](#_Toc50216593)

[4、DDD-工程结构 4](#_Toc50216594)

[二、编程规范 5](#_Toc50216595)

[1. 命名风格 5](#_Toc50216596)

[2. 常量定义 11](#_Toc50216597)

[3. 代码格式 11](#_Toc50216598)

[4. 日期时间 13](#_Toc50216599)

[5. 集合 14](#_Toc50216600)

[6. 并发处理 20](#_Toc50216601)

[7. 条件判断控制 24](#_Toc50216602)

[8. 分布式锁 27](#_Toc50216603)

[三、异常处理 29](#_Toc50216604)

[1、统一异常结果包装 29](#_Toc50216605)

[2、统一接口返回结果包装 33](#_Toc50216606)

[四、注释规范 39](#_Toc50216607)

[五、单元测试 41](#_Toc50216608)

## 一、工程结构

### 1、DDD-领域模型

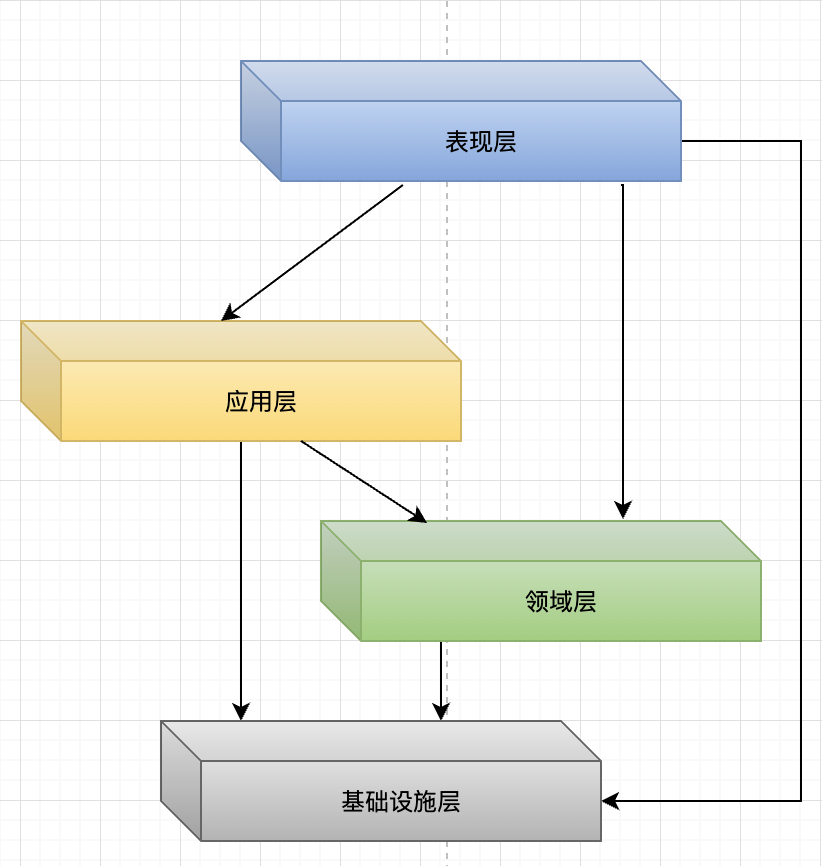
根据DDD领域驱动设计原则，对应的软件架构做出相应的调整。 我们常用的三层架构模型划分为表现层，业务逻辑层，数据访问层等，在DDD分层结构中既有联系又有区别，主要有如下异同：

1. 在架构设计上，在DDD分层结构中将传统三层架构的业务逻辑层拆解为应用层和领域层， 其中application划分为很薄的一层服务，非核心的逻辑放到此层去实现，核心的业务逻辑表现下沉到领域层去实现，凝练为更为精确的业务规则集合。
2. 在建模方式上，DDD分层的建模思维方式有别于传统三层传统三层通常是以数据库为起点进行数据库分析设计，而DDD则需要以业务领域模型为核心建模（即面向对象建模方式），更能体现对现实世界的抽象。 故**在DDD分层凸显领域层的重要作用，领域层为系统的核心，包括所有的业务领域模型的抽象表达**。
3. 在职责划分上，基础设施层涵盖了2方面内容
   1. 持久化功能，其中原三层架构的数据访问层下沉到基础设施层的持久化机制实现
   2. 通用技术支持，一些公共通用技术支持也放到基础设施层去实现。

### 2、DDD-分层详解

|  |  |  |
| --- | --- | --- |
| 分层 | 名称 | 描述 |
| 表现层 | Interface | 表现层，或者用户界面层，负责向用户显示解释用户命令 |
| 应用层 | application | 定义软件要完成的任务，并只会协调领域对象进行不同的操作，该层不包含业务领域知识。 |
| 领域层 | domain | 或称为模型层，系统的核心，负责表达业务概念，业务状态信息以及业务规则，即包含了该领域（问题域）所有复杂的业务知识抽象和规则定义，该层主要作用于领域对象的分析上，可以从实体，值对象，聚合根，领域服务，领域事件，仓储，工厂等方面入手 |
| 基础设施层 | Infrastructure | 主要是两方面的内容：  1、为领域模型提供持久化机制，  2、对其他层提供通用的技术支持能力，如消息通信，通用工具，配置等的实现 |

### 3、DDD-调用规范



图（1）四层架构图

手机屏幕截图

描述已自动生成

图（2）详细架构图

在上图（1）中可以看出来，上层模块可以调用下层模块，反之不行。即

* interface ——> application | domain | infrastructure
* application ——>domain | infrastructure
* domain ——>infrastructure

### 4、DDD-工程结构

├─main

│ ├─java

│  │ ├─com.galaxy.()

│  │ │    ├─interface  API接口层

│  │ │    │ ├─dto   视图模型,数据模型定义 vo、dto（大多数情況一样的）

│  │ │    │ ├─assembler 装配器，实现模型转换 eg.apiModel<=>domainModel

│  │ │    │ └─facade   控制器，对外提供（Restful）接口

│  │ │    ├─application  应用层

│  │ │    │  ├─impl  应用服务实现类

│  │ │    │   └─xxxservice.java  应用服务，非核心服务

│  │ │    ├─domain  领域层

│  │ │    │   ├─common       公共代码抽取，限于领域层有效

│  │ │    │   ├─model   领域模型

│  │ │    │   │    ├─vo            值对象  eg：xxxVO.java

│  │ │    │   │    ├─dos           实体类，充血的领域模型   eg：xxxDO.java

│  │ │    │   │    └─entity   通常表现为实体的聚合，需要有聚合根 eg： xxxPoJo.java 聚合类，

│  │ │    │   ├─service      领域服务类，一些不能归属某个具体领域模型的行为

│  │ │  │  ├─event          领域事件，事件发布 + 事件存储 + 事件分发+ 事件处理

│  │ │    │  ├─factory      工厂类，负责复杂领域对象创建，封装细节

│  │ │  │  └─repository  仓储类似于传统三层架构的DAO接口，只是接口不包括实现

│  │ │    ├─infrastructure  基础设施层

│  │ │    │   ├─persistent   持久化机制

│  │ │    │   │    ├─po           持久化对象

│  │ │    │   │    ├─mapper         MyBatis 的 Mapper 类

│  │ │    │   │    └─repository   仓储类，持久化接口&实现，可与ORM映射框架结合。

│  │ │    │   ├─general      通用技术支持，向其他层输出通用服务

│  │ │    │   │    ├─config       配置类

│  │ │    │   │    ├─utils      工具类

│  │ │    │   │    └─common    基础公共模块等

│  │ │    │   ├─external

│  │ │    │ │ └─client外部服务调用

│  └─resources

│  ├─bootstrap.yml  全局配置文件

│  ├─logback.xml     日志文件

│  └─application.yml   配置文件

├─test

## 二、编程规范

### 命名风格

（1）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。

错误：\_name 、 \_\_name 、 $name 、 name\_ 、 name$ 、 name\_\_

（2）所有编程相关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。

错误：DaZhePromotion [打折] 、 getPingfenByName() [评分] 、 String fw[福娃] 、 int 某变量 = 3

（3）类名使用UpperCamelCase风格，但以下情形例外：DO 、 BO 、 DTO 、 VO 、 AO 、 PO 、 UID等。

eg：ForceCode 、 UserDO 、 HtmlDTO 、 XmlService 、 TcpUdpDeal 、 TaPromotion

错误：forcecode 、 UserDo 、 HTMLDto 、 XMLService 、 TCPUDPDeal 、 TAPromotion

（4）方法名、参数名、成员变量、局部变量都统一使用lowerCamelCase风格。

eg： localValue 、 getHttpMessage() 、 inputUserId

（5）常量命名全部大写，单词间用下划线隔开，语义表达完整清楚，允许名字长。

eg：MAX\_STOCK\_COUNT 、 CACHE\_EXPIRED\_TIME

错误：MAX\_COUNT 、 EXPIRED\_TIME

（6）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）类型与中括号紧挨相连来表示数组。

eg：定义整形数组int[] arrayDemo。

（8）POJO类中的任何布尔类型的变量，都不要加is前缀，否则部分框架解析会引起序列化错误。 注：在本文MySQL规约中的建表约定第一条，表达是与否的变量采用is\_xxx的命名方式，所以，需要在<resultMap>设置从is\_xxx到xxx的映射关系。 错误：定义为基本数据类型Boolean isDeleted的属性，它的方法也是isDeleted()，框架在反向解析的时候，“误以为”对应的属性名称是deleted，导致属性获取不到，进而抛出异常。

（9）包名统一使用小写，点分隔符之间有且仅有一个语义的英语单词。包名统一使用单数形式，类名可以使用复数形式。

eg：应用工具类包名为com.galaxy.dorado..util、类名为MessageUtils

（10）避免在子父类的成员变量之间、或者不同代码块的局部变量之间采用完全相同的命名，使可理解性降低。

注：子类、父类成员变量名相同，即使是public类型的变量也能够通过编译，另外，局部变量在同一方法内的不同代码块中同名也是合法的，这些情况都要避免。对于非setter、getter的参数名称也要避免与成员变量名称相同。

错误：

public class ConfusingName {

public int stock;

//非setter、getter的参数名称，不允许与本类成员变量同名

public void get(String alibaba) {

if (condition) {

final int money = 666;

// ...

}

for (int i = 0; i < 10; i++) {

//在同一方法体中，不允许与其它代码块中的money命名相同

final int money = 15978;

// ...

}

}

}

class Son extends ConfusingName {

//不允许与父类的成员变量名称相同

public int stock;

}

（11）杜绝完全不规范的缩写，避免望文不知义。

错误：AbstractClass“缩写”成AbsClass；condition“缩写”成 condi；Function缩写”成Fu，此类随意缩写严重降低了代码的可阅读性。

（12）为了达到代码自解释的目标，任何自定义编程元素在命名时，使用尽量完整的单词组合来表达。

正例：对某个对象引用的volatile字段进行原子更新的类名为 AtomicReferenceFieldUpdater。 错误：常见的方法内变量为int a;的定义方式。

（13）在常量与变量的命名时，表示类型的名词放在词尾，以提升辨识度。

eg：startTime、workQueue 、 nameList 、TERMINATED\_THREAD\_COUNT

not：startedAt 、 QueueOfWork 、 listName 、 COUNT\_TERMINATED\_THREAD

（14）如果模块、接口、类、方法使用了设计模式，在命名时需体现出具体模式。

eg： public class OrderFactory; public class LoginProxy; public class ResourceObserver;

（15）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，确定与接口方法相关，并且是整个应用的基础常量。 正例：接口方法签名 void commit(); 接口基础常量 String COMPANY = "minsheng"; 错误：接口方法定义 public abstract void f(); 注：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。

（16） 接口和实现类的命名有两套规则：

1）对于Service和mapper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。

eg：CacheServiceImpl实现CacheService接口。

2）如果是表示能力的接口名称，取对应的形容词为接口名（通常是–able的形容词）。

eg：AbstractTranslator实现 Translatable接口。

（17） 枚举类名带上Enum后缀，枚举成员名称需要全大写，单词间用下划线隔开。

注意：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。

eg：枚举名字为ProcessStatusEnum的成员名称：SUCCESS 、 UNKNOWN\_REASON。

（18）各层对象命名规约：

A) Service、mapper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，复数结尾，如：listObjects。

3） 获取统计值的方法用count做前缀。

4） 插入的方法用save、insert做前缀。

5） 删除的方法用remove、delete做前缀。

6） 修改的方法用update做前缀。

B) 领域模型层命名规约

1） 领域对象：xxxDO，xxx即为业务数据表名。

2） 数据传象：xxxDTO，xxxVO（此处规范一下，在interface层统一用DTO，在其他层对象转换用VO）

C）展示层：

1） POJO是DO、DTO、BO、VO的统称，禁止命名成xxxPOJO。

2） xxxDTO，xxx一般为前端请求过来的页面名称。

### 常量定义

（1）不允许任何魔法值（即未经预先定义的常量）直接出现在代码中。

// 本例中，开发者A定义了缓存的key，然后开发者B使用缓存时少了下划线，即key是"Id#taobao"+tradeId，导致出现故障 \_

String key = "Id#taobao\_" + tradeId;

cache.put(key, value);

（2）在long或者Long赋值时，数值后使用大写字母L，不能是小写字母l，小写容易跟数字混淆，造成误解。

（3）不要使用一个常量类维护所有常量，要按常量功能进行归类，分开维护。 \_注：大而全的常量类，杂乱无章，使用查找功能才能定位到修改的常量，不利于理解，也不利于维护。

eg：缓存相关常量放在类CacheConsts下；系统配置相关常量放在类SystemConfigConsts下。

（4）如果变量值仅在一个固定范围内变化用enum类型来定义。

### 代码格式

（1）如果是大括号内为空，则简洁地写成{}即可，大括号中间无需换行和空格；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；表示终止的右大括号后必须换行。

（2）左小括号和右边相邻字符之间不出现空格；右小括号和左边相邻字符之间也不出现空格；而左大括号前需要加空格。详见第5条下方正例提示。

错误：if (空格a == b空格)

（3）if/for/while/switch/do等保留字与括号之间都必须加空格。

（4）任何二目、三目运算符的左右两边都需要加一个空格。 \_注：包括赋值运算符=、逻辑运算符&&、加减乘除符号等。

（5）采用4个空格缩进，禁止使用Tab字符。

注：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。

public class test {

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）没有必要增加若干空格来使变量的赋值等号与上一行对应位置的等号oop规约

### 日期时间

1. 日期格式化时，传入pattern中表示年份统一使用小写的y。 \_注：日期格式化时，yyyy表示当天所在的年，而大写的YYYY代表是week in which year（JDK7之后引入的概念），意思是当天所在的周属于的年份，一周从周日开始，周六结束，只要本周跨年，返回的YYYY就是下一年。

eg：表示日期和时间的格式如下所示： new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

1. 在日期格式中分清楚大写的M和小写的m，大写的H和小写的h分别指代的意义。

注：日期格式中的这两对字母表意如下：

1） 表示月份是大写的M；

2） 表示分钟则是小写的m；

3） 24小时制的是大写的H；

4） 12小时制的则是小写的h。

（3）时间日期使用jdk1.8的新特性，表示日期的LocalDate、表示时间的LocalTime、表示日期时间的LocalDateTime

（4）不允许在程序任何地方中使用：

1）java.sql.Date。

2）java.sql.Time。

3）java.sql.Timestamp。

注：第1个不记录时间，getHours()抛出异常；第2个不记录日期，getYear()抛出异常；第3个在构造方法super((time/1000)\*1000)，在Timestamp 属性fastTime和nanos分别存储秒和纳秒信息。

错误： \_java.util.Date.after(Date)进行时间比较时，当入参是java.sql.Timestamp时，会触发JDK BUG(JDK9已修复)，可能导致比较时的意外结果。 \_

（5）不要在程序中写死一年为365天，避免在公历闰年时出现日期转换错误或程序逻辑错误。

（6）避免公历闰年2月问题。闰年的2月份有29天，一年后的那一天不可能

（7）使用枚举值来指代月份。如果使用数字，注意Date，Calendar等日期相关类的月份month取值在0-11之间。

注：参考JDK原生注释，Month value is 0-based. e.g., 0 for January.

eg：Calendar.JANUARY，Calendar.FEBRUARY，Calendar.MARCH等来指代相应月份来进行传参或比较。

### 集合

（1）关于hashCode和equals的处理，遵循如下规则：

1） 只要覆写equals，就必须覆写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存储的对象必须覆写这两种方法。

3）如果自定义对象作为Map的键，那么必须覆写hashCode和equals。 注：String因为覆写了hashCode和equals方法，所以可以愉快地将String对象作为key来使用。

（2）判断所有集合内部的元素是否为空，使用isEmpty()方法，而不是size()==0的方式。 注：在某些集合中，前者的时间复杂度为O(1)，而且可读性更好。

eg： Map<String, Object> map = new HashMap<>(16);

if(map.isEmpty()) {

System.out.println("no element in this map.");

}

（3）在使用java.util.stream.Collectors类的toMap()方法转为Map集合时，一定要使用含有参数类型为BinaryOperator，参数名为mergeFunction的方法，否则当出现相同key值时会抛出IllegalStateException异常。

注：参数mergeFunction的作用是当出现key重复时，自定义对value的处理策略。

eg：

List<Pair<String, Double>> pairArrayList = new ArrayList<>(3);

pairArrayList.add(new Pair<>("version", 12.10));

pairArrayList.add(new Pair<>("version", 12.19));

pairArrayList.add(new Pair<>("version", 6.28));

Map<String, Double> map = pairArrayList.stream().collect(

// 生成的map集合中只有一个键值对：{version=6.28}

Collectors.toMap(Pair::getKey, Pair::getValue, (v1, v2) -> v2));

（4）在使用java.util.stream.Collectors类的toMap()方法转为Map集合时，一定要注意当value为null时会抛NPE异常。

注：在java.util.HashMap的merge方法里会进行如下的判断：

if (value == null || remappingFunction == null)

throw new NullPointerException();

（5）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。

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

（6）使用Map的方法keySet()/values()/entrySet()返回集合对象时，不可以对其进行添加元素操作，否则会抛出UnsupportedOperationException异常。 \_

（7）Collections类返回的对象，如：emptyList()/singletonList()等都是immutable list，不可对其进行添加或者删除元素的操作。 \_错误：如果查询无结果，返回Collections.emptyList()空集合对象，调用方一旦进行了添加元素的操作，就会触发UnsupportedOperationException异常。

（8）在subList场景中，高度注意对父集合元素的增加或删除，均会导致子列表的遍历、增加、删除产生ConcurrentModificationException 异常。

（9）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致、长度为0的空数组。 \_错误：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。

eg：List<String> list = new ArrayList<>(2);

list.add("guan");

list.add("bao");

String[] array = list.toArray(new String[0]);

注：使用toArray带参方法，数组空间大小的length：

1） 等于0，动态创建与size相同的数组，性能最好。

2） 大于0但小于size，重新创建大小等于size的数组，增加GC负担。

3） 等于size，在高并发情况下，数组创建完成之后，size正在变大的情况下，负面影响与2相同。

4） 大于size，空间浪费，且在size处插入null值，存在NPE隐患。

（10）在使用Collection接口任何实现类的addAll()方法时，都要对输入的集合参数进行NPE判断。

注：在ArrayList#addAll方法的第一行代码即Object[] a = c.toArray(); 其中c为输入集合参数，如果为null，则直接抛出异常。

（11）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。

注：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[] { "chen", "yang", "hao" }; List list = Arrays.asList(str);

第一种情况：list.add("yangguanbao"); 运行时异常。

第二种情况：str[0] = "change"; 也会随之修改，反之亦然。

（12）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，两者在接口调用赋值的场景中容易出错。

注：扩展说一下PECS(Producer Extends Consumer Super)原则：

第一、频繁往外读取内容的，适合用<? extends T>。

第二、经常往里插入的，适合用<? super T>

（13）在无泛型限制定义的集合赋值给泛型限制的集合时，在使用集合元素时，需要进行instanceof判断，避免抛出ClassCastException异常。

注：毕竟泛型是在JDK5后才出现，考虑到向前兼容，编译器是允许非泛型集合与泛型集合互相赋值。

（14）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。

eg： List<String> list = new ArrayList<>();

list.add("1");

list.add("2");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (删除元素的条件) {

iterator.remove();

}

}

（15）在JDK7版本及以上，Comparator实现类要满足如下三个条件，不然Arrays.sort，Collections.sort会抛IllegalArgumentException异常。

注：三个条件如下

1） x，y的比较结果和y，x的比较结果相反。

2） x>y，y>z，则x>z。

3） x=y，则x，z比较结果和y，z比较结果相同。

（16）集合泛型定义时，在JDK7及以上，使用diamond语法或全省略。

注：菱形泛型，即diamond，直接使用<>来指代前边已经指定的类型。

（17）集合初始化时，指定集合初始值大小。 \_注：HashMap使用HashMap(int initialCapacity) 初始化，如果暂时无法确定集合大小，那么指定默认值（16）即可。

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（即默认值）。

错误： \_HashMap需要放置1024个元素，由于没有设置容量初始大小，随着元素增加而被迫不断扩容，resize()方法总共会调用8次，反复重建哈希表和数据迁移。当放置的集合元素个数达千万级时会影响程序性能。 \_

（18）使用entrySet遍历Map类集合KV，而不是keySet方式进行遍历。

注：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。

（19）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 | 注 |
| 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 | 锁分段技术（JDK8:CAS） |
| 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 |

### 并发处理

1. 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：自定义线程工厂，并且根据外部特征进行分组，比如，来自同一机房的调用，把机房编号赋值给whatFeatureOfGroup

public class UserThreadFactory implements ThreadFactory {

private final String namePrefix;

private final AtomicInteger nextId = new AtomicInteger(1);

// 定义线程组名称，在利用jstack来排查问题时，非常有帮助 \_

UserThreadFactory(String whatFeatureOfGroup) {

namePrefix = "From UserThreadFactory's " + whatFeatureOfGroup + "-Worker-";

}

@Override

public Thread newThread(Runnable task) {

String name = namePrefix + nextId.getAndIncrement();

Thread thread = new Thread(null, task, name, 0, false);

System.out.println(thread.getName());

return thread;

}

}

（2）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。注：线程池的好处是减少在创建和销毁线程上所消耗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。

（3）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。

注：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： 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。

（4）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。

eg：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");

}

};

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

（5）必须回收自定义的ThreadLocal变量，尤其在线程池场景下，线程经常会被复用，如果不清理自定义的 \_ThreadLocal变量，可能会影响后续业务逻辑和造成内存泄露等问题。尽量在代理中使用try-finally块进行回收。

eg：objectThreadLocal.set(userInfo);

try {

// ...

} finally {

objectThreadLocal.remove();

}

（6）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。

注：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）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。

注：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）在使用阻塞等待获取锁的方式中，必须在try代码块之外，并且在加锁方法与try代码块之间没有任何可能抛出异常的方法调用，避免加锁成功后，在finally中无法解锁。

原因：

1）：如果在lock方法与try代码块之间的方法调用抛出异常，那么无法解锁，造成其它线程无法成功获取锁。

2）：如果lock方法在try代码块之内，可能由于其它方法抛出异常，导致在finally代码块中，unlock对未加锁的对象解锁，它会调用AQS的tryRelease方法（取决于具体实现类），抛出IllegalMonitorStateException异常。

3）：在Lock对象的lock方法实现中可能抛出unchecked异常，产生的后果与注二相同。

eg： Lock lock = new XxxLock();

// ...

lock.lock();

try {

doSomething();

doOthers();

} finally {

lock.unlock();

}

（9）在使用尝试机制来获取锁的方式中，进入业务代码块之前，必须先判断当前线程是否持有锁。锁的释放规则与锁的阻塞等待方式相同。

注：Lock对象的unlock方法在执行时，它会调用AQS的tryRelease方法（取决于具体实现类），如果当前线程不持有锁，则抛出IllegalMonitorStateException异常。

eg： Lock lock = new XxxLock();

boolean isLocked = lock.tryLock();

if (isLocked) {

try {

doSomething();

doOthers();

} finally {

lock.unlock();

}

}

（10）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。

注：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。

（11）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。

（12）资金相关的金融敏感信息，使用悲观锁策略。 \_注：乐观锁在获得锁的同时已经完成了更新操作，校验逻辑容易出现漏洞，另外，乐观锁对冲突的解决策略有较复杂的要求，处理不当容易造成系统压力或数据异常，所以资金相关的金融敏感信息不建议使用乐观锁更新。

eg：悲观锁遵循一锁、二判、三更新、四释放的原则。

（13）使用CountDownLatch进行异步转同步操作，每个线程退出前必须调用·

注：注意，子线程抛出异常堆栈，不能在主线程try-catch到。

（14）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。

注：Random实例包括java.util.Random 的实例或者 \_Math.random()的方式。

eg：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单独的Random实例。

（15）通过双重检查锁（double-checked locking）（在并发场景下）存在延迟初始化的优化问题隐患（可参考 The "Double-Checked Locking is Broken" Declaration），推荐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，比如将helper的属性声明修改为

private volatile Helper helper = null;

public class LazyInitDemo {

private volatile Helper helper = null;

public Helper getHelper() {

if (helper == null) {

synchronized (this) {

if (helper == null) { helper = new Helper(); }

}

}

return helper;

}

// other methods and fields...

}

（16）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。

注：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）

### 条件判断控制

1. 在一个switch块内，每个case要么通过continue/break/return等来终止，要么注释注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。

注：注意break是退出switch语句块，而return是退出方法体。

（2）当switch括号内的变量类型为String并且此变量为外部参数时，必须先进行null判断。 \_错误：如下的代码输出是什么？ \_

（3）在if/else/for/while/do语句中必须使用大括号。

注：即使只有一行代码，也禁止不采用大括号的编码方式：if (condition) statements;

（4）三目运算符condition? 表达式1 : 表达式2中，高度注意表达式1和2在类型对齐时，可能抛出因自动拆箱导致的NPE异常。

注：以下两种场景会触发类型对齐的拆箱操作：

1） 表达式1或表达式2的值只要有一个是原始类型。

2） 表达式1或表达式2的值的类型不一致，会强制拆箱升级成表示范围更大的那个类型。

（5）在高并发场景中，避免使用”等于”判断作为中断或退出的条件。

注：如果并发控制没有处理好，容易产生等值判断被“击穿”的情况，使用大于或小于的区间判断条件来代替。

（6）当某个方法的代码总行数超过10行时，return / throw 等中断逻辑的右大括号后均需要加一个空行。

（7）表达异常的分支时，少用if-else方式，这种方式可以改写成：

if (condition) {

...

return obj;

}

（8）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。

（9）不要在其它表达式（尤其是条件表达式）中，插入赋值语句。 \_注：赋值点类似于人体的穴位，对于代码的理解至关重要，所以赋值语句需要清晰地单独成为一行。

（10）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）。

（1 1）避免采用取反逻辑运算符。

（12）公开接口需要进行入参保护，尤其是批量操作的接口。

（13）下列情形，需要进行参数校验：

1） \_调用频次低的方法。

2） 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。

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。

（14）下列情形，不需要进行参数校验：

1） 极有可能被循环调用的方法。但在方法注里必须注明外部参数检查。

2） 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。

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。

### 分布式锁

（1）系统提供统一的 Redis 分布式锁封装注解，对于业务服务，只允许使用统一的 Redis 分布式锁注解，系统提供统一的 Redis 分布式锁注解 @Lockable

（2）使用方式

1) 注解 Lockable

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

public @interface Lockable {

String[] key() default "";

long maxWait() default 2000;

long expiration() default 1000;

LockMode mode() default LockMode.Fair;

TimeUnit unit() default TimeUnit.MILLISECONDS;

}

2）字段注如下：

|  |  |  |
| --- | --- | --- |
| **序号** | **字段** | **描述** |
| 1 | key | 定义 redis 锁的 key |
| 2 | maxWait | 获取锁时最大等待时间 |
| 3 | expiration | 锁的过期时间 |
| 4 | mode | 锁的类型，默认公平锁 |
| 5 | unit | 时间单位 |

3）添加依赖

<dependency>

<groupId>io.ifa.chaos.starters</groupId>

<artifactId>chaos-starter-lock</artifactId>

<version>1.0</version>

</dependency>

4）配置文件

单机模式

spring:

redis:

host: 192.168.206.144

port: 6379

哨兵模式

spring:

redis:

sentinel:

master: mymaster

nodes: 192.168.10.111:26379

（3）使用场景

对于新开发的接口，在 Service 层的方法上添加 @Lockable 注解，样例如下：

@Service

public class SecKillServiceImpl implements SecKillService {

@Lockable(expiration = 60000,maxWait = 1000,key = {"#productId"})

public void orderProductMockDiffUser(String productId) {

//1.查询商品库存，为 0 则活动结束

int stockNum = stock.get(productId);

if (stockNum == 0) {

throw new SecKillException(100, "活动结束");

} else {

//2.下单（模拟不同用户 id 不同）

orders.put(KeyUtil.genUniqueKey(), productId);

//3.减库存

stockNum = stockNum - 1;

try {

// 模拟耗时请求

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

stock.put(productId, stockNum);

}

}

}

## 三、异常处理

### 1、统一异常结果包装

（1）、约定

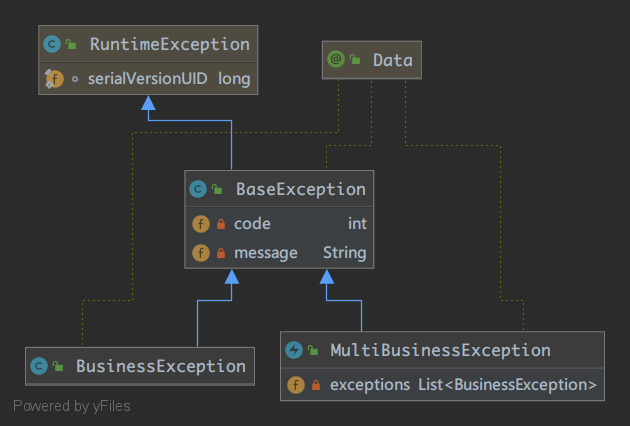
系统提供统一的异常封装类，对于业务过程中，可被识别的业务异常如核保失败、产品停售、代理人信息不存在等，均以抛出业务异常方式结束调用，特别在 domain 层，方法的返回值仅允许使用业务模型，业务校验不通过等其他结果，只允许使用异常方式抛出

* 系统提供统一的业务异常类 BusinessException，该类接收 ResultCode 及其子类作为参数，详细见 [错误码定义类](http://mslife.sinosoft.co/002-common/2002-result-wrapper.html#%E9%94%99%E8%AF%AF%E7%A0%81%E5%AE%9A%E4%B9%89%E7%B1%BB)，该异常类仅允许 application 和 domain 层抛出
* 系统提供统一的多业务异常类 MultiBusinessException，该异常类用于投保规则校验等会发生多个业务异常，需要一次性响应的场景
* 系统提供统一的异常基类 BaseException，infrastructure 及 interfaces 层的异常可以继承该类

异常类可结合 [统一接口返回结果包装](http://mslife.sinosoft.co/002-common/2002-result-wrapper.html) 使用，当 Controller 中方法未正常执行，抛出异常时，系统将自动捕获异常信息，并封装统一的返回结果

（2）、使用注

1）类关系图

2）BusinessException 的使用

a）领域层和应用层

当 domain 层的业务过程发生异常，使用该类，参考如下：

public class InsuringSerivce{

public void apply(ApplicationFormCreateCommand command){

// 代理人校验

Agent agent = agentService.get(agentCode);

if(!agent.getManageCom().equals(command.getOrganizationCode())){

//代理人所属机构与当前机构不符，抛出异常

throw new BusinessException(DoradoResultCode.AGENT\_NOT\_FOUND\_IN\_CURRENT\_ORGANIZATION);

}

//后续业务处理

}

}

b）接口层

Controller 层调用当前方法，有两种处理方式

* 不捕获，系统自动处理，对于该场景，Controller 层需要在方法或者类上添加 @ResultWrapped，参考代码如下：

public class SomeController{

@Resource InsuringSerivce insuringService;

@GetMappings("/some/path")

@ResultWrapped

public String someMethod(){

insuringService.apply(command);

String result = "some string";

return result;

}

}

当 insuringService.apply 抛出 BusinessException 时，系统将自动封装当前方法的返回值为如下内容：

{

"code":51002,

"message":"代理人不属于当前机构",

"elapsed":108,

"data":null

}

* 捕获异常，人工处理

对于该场景，Controller 层需要在方法上添加 @IgnoreWrapped 注解，随后捕获异常并处理

public class SomeController{

@Resource InsuringSerivce insuringService;

@GetMappings("/some/path")

@IgnoreWrapped

public Result<String> someMethod(){

try{

insuringService.apply(command);

}catch(BusinessException e){

return Result.failure(e.getCode(),e.getMessage());

}

String result = "some string";

return Result.success(result);

}

}

该方式返回结果和第一种相同。多异常处理

部分场景要求发生异常不中断，继续执行业务处理，比如投核保规则等，系统提供多异常包装类 MultiBusinessException，使用方式如下：

public class InsuringSerivce{

public void apply(ApplicationFormCreateCommand command){

// 投保规则校验

InsuringRuleRes insuringRes = null;//规则引擎调用

MultiBusinessException multiBusinessException = new MultiBusinessException();

for(InsuringRuleRes.CheckResult result:insuringRes.getCheckResults()){

//这里包装了多个投保规则校验异常

multiBusinessException.addException(new BusinessException(result.getRuleCode(),result.getResult()));

}

//后续业务处理

SomeModel model = someService.someThings();

if(model.someProperty != someCondition){

//包装更多的异常

multiBusinessException.addException(new BusinessException(DoradoResultCode.SOME\_ERROR\_CODE));

}

//检查是否需要抛出异常

if(multiBusinessException.needThrowOut()){

throw multiBusinessException;

}

}

}

对于 domain 及 application 抛出的多异常类，系统暂无自动处理，需要参照 捕获异常，人工处理 的方式捕获异常后，人工处理，MultiBusinessException 类的定义如下：

public class MultiBusinessException extends BaseException {

private List<BusinessException> exceptions = new ArrayList<>();

}

在 Controller 中需要循环其属性 exceptions，遍历将异常信息放置到响应结果中。

### 2、统一接口返回结果包装

（1）约定

为统一系统的接口返回报文格式，现做如下约定：

* 接口层使用统一的返回值包装类
* 定义统一的异常处理机制，自动包装异常类为响应报文
* 各服务按照分配的异常码范围定义服务内异常信息
  1. 使用方式

a）统一的报文返回格式

{

"code":0,

"message":"请求成功",

"elapsed":183,

"data":""

}

b）字段注如下：

| **序号** | **字段** | **描述** |
| --- | --- | --- |
| 1 | code | 接口请求返回的状态码，约定以 **0** 作为接口正常响应的状态码，**-1** 作为未捕获的系统异常状态码，其他状态码由各服务定义 各服务错误码定义 |
| 2 | message | 返回的状态码对应的消息文字 |
| 3 | elapsed | 从服务接受请求到响应请求所花的时间，毫秒 |
| 4 | data | 返回的数据，根据接口定义，返回的可能是字符串、数组、对象等 |

* 1. 添加依赖

<dependency>

<groupId>io.ifa.chaos.starters</groupId>

<artifactId>chaos-starter-api</artifactId>

<version>1.0</version>

</dependency>

* 1. 使用场景

a、使用注解的方式，对于新开发的接口，在 Controller 或者方法上添加 @ResultWrapped 注解，样例如下：

@RestController

@ResultWrapped

public class SegmentController {

}

或者

@RestController

public class SegmentController {

@GetMapping("/api/segment/{segment}/{prefix}")

@ResultWrapped

public String id(@PathVariable String segment, @PathVariable String prefix){

return generateService.generate(segment,prefix);

}

@GetMapping("/api/segment/{segment}/{prefix}")

@ResultWrapped

public Segment id(@PathVariable String segment, @PathVariable String prefix){

return generateService.generate(segment,prefix);

}

}

当添加了注解后，方法的返回值可以是实际需要返回的数据，比如需要返回字符串，则使用 String，需要返回投保人信息，则使用 ApplicantDTO，系统会自动封装返回内容到报文中的 data 字段

b）使用代码方式

当特殊情况注解无法满足需求时，可使用代码方式构建返回值，此时无需添加注解，如果在当前 Controller 类添加了注解，请按照[这里](http://mslife.sinosoft.co/002-common/2002-result-wrapper.html" \l "%E6%B3%A8%E8%A7%A3%E6%96%B9%E5%BC%8F%E5%BF%BD%E7%95%A5)忽略当前类的包装，参考代码如下：

@GetMapping("/api/path")

public Result<String> method(@PathVariable String segment, @PathVariable String prefix){

String value;

return Result.success(value);

//或者

//return Result.failure(DoradoResultCode.INSURING\_RULE\_VALIDATE\_FAIL,value);

}

@GetMapping("/api/path")

public Result<ApplicantDTO> method(@PathVariable String segment, @PathVariable String prefix){

ApplicantDTO applicant;

return Result.success(applicant);

//或者

//return Result.failure(DoradoResultCode.INSURING\_RULE\_VALIDATE\_FAIL,applicant);

}

4）忽略包装

a)注解方式忽略

当在 Controller 上添加了 @ResultWrapped 注解，但是当前 Controller 下有某个方法不想被包装，或者需要使用代码方式返回结果时，可在当前方法上添加 IgnoreWrapped 注解，系统自动包装返回值时将忽略该方法。

@GetMapping("/api/segment/{segment}/{prefix}")

@IgnoreWrapped

public Segment id(@PathVariable String segment, @PathVariable String prefix){

return generateService.generate(segment,prefix);

}

b）配置方式忽略

当有存量代码暂时不想修改返回值方式，但是需要忽略返回值包装时，系统提供如下配置

chaos:

wrapped:

# 需要忽略的类

ignored-classes:

- com.galaxy.dorado.interfaces.insuring.InsuringController

- com.galaxy.dorado.other.OtherClass

# 需要忽略的包

ignored-packages:

- com.galaxy.dorado.interfaces.other.package

以如上配置将忽略 InsuringController、OtherClass 及 other.package 下的所有类，不做返回值包装

（2）各服务错误码范围定义

1）错误码定义类

各服务在 interface 层新建本服务的异常码定义类，继 承 ResultCode 类，参照该类方式定义异常码，不允许在代码中以非常 量方式使用异常码和异常消息，ResultCode 类定义如下：

public class ResultCode implements Serializable {

private int code;

private String message;

public ResultCode(int code, String message){

this.code = code;

this.message = message;

}

/\*\*

\* 请求响应正常

\*/

public static ResultCode SUCCESS = new ResultCode(0,"请求成功");

/\*\*

\* 未知异常响应

\*/

public static ResultCode UNKNOWN\_ERROR = new ResultCode(-1,"服务器打了个盹，马上回来");

}

各服务的异常码定义类应当如下：

public class DoradoResultCode extends ResultCode implements Serializable {

/\*\*

\* 投保规则校验失败

\*/

public static DoradoResultCode INSURING\_RULE\_VALIDATE\_FAIL = new ResultCode(51001,"投保规则校验失败");

}

2) 错误码长度及范围定义

除请求响应成功和未知异常的返回码外，其他服务的返回码均为 5-6 位数字下表是各服务的异常码范围：

|  |  |
| --- | --- |
| 序号 | 范围 |
| 客户 | P030000- P039999 |
| 产品 | P040000- P049999 |
| 运营 | P050000- P059999 |
| 风控 | P060000- P069999 |
| 财务 | P070000- P079999 |
| 合同 | P080000- P089999 |
| 数据 | P0100000- P0109999 |
| 统一接入 | P0110000- P0119999 |

## 四、注释规范

1. 类、类属性、类方法的注释必须使用Javadoc规范：

使用/\*\*xxx\*/格式，不得使用// xxx方式。

2. 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、异常注外，还必须指出该方法做什么事情，实现什么功能。

注：对子类的实现要求，或者调用注意事项，请一并注。

3. 所有的类都必须添加创建者、创建日期、功能描述（包括功能注意事项等信息的描述）

注意：（1）IDEA模板设置如下：

电脑屏幕的截图

描述已自动生成

截图里有图片

描述已自动生成

（2）eclipse模板设置如下：

4. 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. 所有的枚举类型字段必须要有注释，注每个数据项的用途。

6. 最好使用英文来注释，不行就用中文注释把问题描述清楚。可以允许专有名词与关键字保持英文原文。

7. 代码修改的同时，注释也要进行相应的修改，特别是参数、返回值、异常、核心逻辑等的修改。（应该注释上修改的author、time、reason）

8. 在类中删除未使用的任何字段、方法、内部类；

在方法中删除未使用的任何参数声明与内部变量。

9. 谨慎注释掉代码。在上方详细注，而不是简单地注释掉。如果无用，则删除。

10. 注释的双斜线与注释内容之间有且仅有一个空格。

## 五、单元测试

1、好的单元测试必须遵守AIR原则。 注：

⚫ A：Automatic（自动化）

⚫ I： Independent（独立性）

⚫ R：Repeatable（可重复）

2、单元测试应该是全自动执行的，并且非交互式的。测试用例通常是被定期执行的，执行过程必须完全自动化才有意义。输出结果需要人工检查的测试不是一个好的单元测试。单元测试中不准使用System.out来进行人肉验证，必须使用assert来验证。

注：（1）assertEquals

* 函数原型1：Assert.assertEquals([String message],expected,actual)

参数注：

message（可选）：将会在发生错误时报告这个消息。

expected（必填）：期望值，通常都是用户指定的内容。

actual（必填）：是被测试的代码返回的实际值。

例：Assert.assertEquals("equals","1","1");

* 函数原型2 ：Assert.assertEquals([Stringmessage],expected,actual,tolerance)

参数注：

message（可选）：将会在发生错误时报告这个消息。

expected（必填）：期望值，通常都是用户指定的内容。

actual（必填）：被测试的代码返回的实际值。

tolerance（选填）：误差参数，参加比较的两个浮点数在这个误差之内则会被认为是相等的。

例： Assert.assertEquals ("yes",5.8,11.0/2.0,0.5);

（2）assertTrue

* 函数原型：Assert.assertTrue ([String message],Boolean condition)

参数注：

message(可选)：将会在发生错误时报告这个消息。

condition（必填）：待验证的布尔型值。

该断言用来验证给定的布尔型值是否为真，假如结果为假，则验证失败。当然，更有验证为假的测试条件：

* 函数原型： Assert.assertFalse([String message],Boolean condition)

该断言用来验证给定的布尔型值是否为假，假如结果为真，则验证失败。

例：Assert.assertTrue("true",1==1);

Assert.assertFalse("false",2==1);

（3）assertNull

* 函数原型： Assert.assertNull([String message],Object object)

参数注：

message（可选）：将会在发生错误时报告这个消息。

object（必填）：待验证的对象。

该断言用来验证给定的对象是否为null，假如不为null，则验证失败。相应地，还存在能够验证非null的断言：

* 函数原型：

Assert.assertNotNull([String message],Object object)

该断言用来验证给定的对象是否为非null，假如为null，则验证失败。

例：Assert.assertNull("null",null);

Assert.assertNotNull("not null",new String());

（4）、assertSame

* 函数原型： Assert.assertSame ([String message], expected, actual)

参数注：

message是个可选的消息，假如提供，将会在发生错误时报告这个消息。

expected是期望值。

actual是被测试的代码返回的实际值。

该断言用来验证expected参数和actual参数所引用的是否是同一个对象，假如不是，则验证失败。相应地，也存在验证不是同一个对象的断言：

* 函数原型： Assert.assertNotSame ([String message], expected, actual)

该断言用来验证expected参数和actual参数所引用的是否是不同对象，假如所引用的对象相同，则验证失败。

例：Assert.assertSame("same",2,4-2);

Assert.assertNotSame("not same",2,4-3);

（5）、fail

* 函数原型：Assert.fail(actual, expected, [String message],operator)

参数注：

actual（必填）：被测试的代码返回的实际值。

expected（必填）：期望值，通常都是用户指定的内容。

message（可选）：会在发生错误时报告这个消息。

operator（必填）：在发生错误时，actual值和expexted中间的分隔符。

抛出一个 AssertionError。如果 message 是假值，错误信息会被设置为被 operator 分隔在两边 actual 和 expected 的值。否则，该错误信息会是 message 的值。该断言会使测试立即失败，通常用在测试不能达到的分支上（如异常）。

例：Assert.fail(1, 2, undefined, '>');// AssertionError: 1 > 2

3、保持单元测试的独立性。为了保证单元测试稳定可靠且便于维护，单元测试用例之间决不能互相调用，也不能依赖执行的先后次序。

4、单元测试是可以重复执行的，不能受到外界环境的影响。

注：单元测试通常会被放到持续集成中，每次有代码check in时单元测试都会被执行。如果单测对外部环境（网络、服务、中间件等）有依赖，容易导致持续集成机制的不可用。

eg：为了不受外界环境影响，要求设计代码时就把SUT的依赖改成注入，在测试时用spring 这样的DI框架注入一个本地（内存）实现或者Mock实现。

5、对于单元测试，要保证测试粒度足够小，有助于精确定位问题。单测粒度至多是类级别，一般是方法级别。 \_注：只有测试粒度小才能在出错时尽快定位到出错位置。单测不负责检查跨类或者跨系统的交互逻辑，那是集成测试的领域。

6、核心业务、核心应用、核心模块的增量代码确保单元测试通过。

注：新增代码及时补充单元测试，如果新增代码影响了原有单元测试，请及时修正。

7、单元测试代码必须写在如下工程目录：src/test/java，不允许写在业务代码目录下。

注：源码编译时会跳过此目录，而单元测试框架默认是扫描此目录。

8、编写单元测试代码遵守BCDE原则，以保证被测试模块的交付质量。

注：

⚫ Border，边界值测试，包括循环边界、特殊取值、特殊时间点、数据顺序等。

⚫ Correct，正确的输入，并得到预期的结果。

⚫ Design，与设计文档相结合，来编写单元测试。

⚫ Error，强制错误信息输入（如：非法数据、异常流程、业务允许外等），并得到 预期的结果。

9、对于数据库相关的查询，更新，删除等操作，不能假设数据库里的数据是存在的，或者直接操作数据库把数据插入进去，请使用程序插入或者导入数据的方式来准备数据。

10、和数据库相关的单元测试，可以设定自动回滚机制，不给数据库造成脏数据。或者对单元测试产生的数据有明确的前后缀标识。 \_正例：在阿里巴巴企业智能事业部的内部单元测试中，使用ENTERPRISE\_INTELLIGENCE \_UNIT\_TEST\_的前缀来标识单元测试相关代码。

11、对于不可测的代码在适当的时机做必要的重构，使代码变得可测，避免为了达到测试要求而书写不规范测试代码。

12、在设计评审阶段，开发人员需要和测试人员一起确定单元测试范围，单元测试最好覆盖所有测试用例（UC）。

13、单元测试作为一种质量保障手段，在项目提测前完成单元测试，不建议项目发布后补充单元测试用例。