编码规范

版 本 历 史

版本	内容	日期	备注
V1.0	编码规范	2011-12-4	
V2.0		2016-3-1	翟冲修改
V2.1		2017-4-18	翟冲修改
V3.0		2019-4-13	熊跃辉、王寒冰、翟冲、张立强，本版本文档适用于基于 gboat3 开发的产品和项目。
V3.1		2020-04	参与人：周兵、王寒冰、袁小芬、熊跃辉、翟冲
V4.0		2021-03	内容：将阿里巴巴代码规范 1.3.0 中适合我们规范补充进来，主要包括并发、异常、日志等、并将 SQL 规范合并进来； 参与人：周兵、王寒冰、袁小芬、熊跃辉、兰时军、付春阳、刘红路、柳海军、翟冲

目录

1. 引言

软件开发涉及到各方面人员的交互、协作，为了有效地进行项目开发的沟通，完善代码的维护和交付，有必要在一个小组中采用统一的软件开发标准。一般来说，制定这样的标准有下列好处：

- 方便软件维护。据统计，80%的软件开发费用在维护，规范化的代码才方便维护，降低维护成本。
- 在软件的整个生命期内，期望一个编码人员从开始到该软件报废一致维护其代码是不现实的，必然需要不断地交付、协同
- 好的编码规范能够大大增强代码的可读性，便于开发人员快速的理解新代码。
- 任何产品都需要好的包装。我们可以把代码本身看作是一种产品，那么按照规范编程也是对这个“产品”的包装
- 规范化的代码也是软件质量的保证手段之一，也是软件过程能够流畅的基础。

我们每个人必须牢牢树立这样的观念：无论从时间跨度还是从工作量来说，一个软件系统生存期的大部分是维护，不是开发。你今天所编写的代码，会一直使用很多年，并且很有可能被其他人维护和改进。所以，我们必须努力写出“干净”和易读的代码。

本文档适用于软件开发过程中开发人员，主要包括编码人员、测试人员、开发人员，规范必须严格遵守，否则程序被视为不合格程序。

2. Java 命名规范

2.1. 字典说明

我们的业务简写包括以下类型，我们在给系统起名，给 package 起名简写都遵循这个简写规则：

- 业务简称：

  政府采购：gp

  建设工程：gb

  产权：gc

  土矿：le

  竞价：aus

  企业库：gfm

  保证金：gtm

  场地：gvm

  编标：geb

开评标：gbes

大数据：gst

2.2.组件的命名

- 某个模块的接口组件命名格式是：项目或者产品名称-模块名称。比如政府采购里的项目组件，名称是：gp-project；
- 某个模块的实现组件命名格式是：项目或者产品名称-模块名称-provider。比如政府采购项目组件，名称是：gp-project-provider；
- 原则上每个业务发布一个微服务 API 组件，这个组件是服务提供者自己提供的，格式为：项目或者产品名称-api，例如企业库对外提供的微服务 api 组件名称是：gfm-api，竞价系统对外提供的 api 组件的名称是：aus-api；
- 组件的 groupId 到项目的名称一级，例如：com.glodon.gp，组件的 artifactId 格式是：项目或者产品名称-组件名称,例如 model 组件是：gp-project，provider 组件是：gp-project-provider

2.3.package 的命名

- 【强制】产品模块导出的接口定义放在 com.glodon.项目或者产品名称.模块.service(放的全是对外接口）；
- 【强制】对外服务的实现类 com.glodon.项目或者产品名称.模块.service.impl(放的全是对外接口实现，依赖 business)
- 【推荐】所有 service.impl 里面仅实现很小一部分业务逻辑，内部核心业务逻辑放在 business 包中
- 【强制】业务类代码接口放在 com.glodon.项目或者产品名称.模块.business,可继承 IBaseService
- 【强制】后台业务类代码 com.glodon.项目或者产品名称.模块.business.impl,可继承 BaseService
- 【强制】domain 命名 com.glodon.项目或者产品名称.模块.model(域模型层 PO+VO+Constants)
- 【强制】controller 命名 com.glodon.项目或者产品名称.模块.controller(直接调用 business 进行处理)
- 【强制】util 命名 com.glodon.项目或者产品名称.模块.utils
- 【强制】微服务包命名 com.glodon.项目或者产品名称.api.模块
- 【推荐】命名简短，常采用约定俗成的缩写，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其意
- 【强制】package 的所有字符都为小写
- 【强制】严禁使用 java, javax 作为自定义包的前缀

2.4. 类、接口、枚举的命名

【强制】类、接口的名字必须由大写字母开头，而其他字母都小写的单词组成,对于所有标识符，其中包含的所有单词都应紧靠在一起，而且大写中间单词的首字母，一般使用名词命名。

```
public abstract class AbstractWebService {
    /*内容*/
}
```

- 【强制】本地接口的第一个字符用“I”开头，实现类的命名统一以“Impl”结尾。

```
public interface IUserInfoQryService {
    /**内容*/
}
public class UserInfoQryServiceImpl implements IUserInfoQryService {
    /**内容*/
}
```

- 【强制】抽象类以 Abstract 开头，例如：

```
public class AbstractController {
    /**内容*/
}
```

- 【强制】如果使用了设计模式，以设计模式的名称结尾，例如：

```
public class ProcessSelectFactory {
    /**内容*/
}
public class ControllerProxy {
    /**内容*/
}
```

- 【强制】枚举类以 Enum 结尾。

```
public enum AssignerTypeEnum {
    **内容*/
}
```

- 【强制】所有 VO 以大写 VO 结尾

```
public class ProjectVO {
    **内容*/
}
```

2.5. 微服务接口规范

【强制】调用远端的接口，比如我们这里的微服务接口，都是 RI 开头，例如；

Public class RIUserInfoService();

一般我们微服务的接口名字是 RIUserInfoService,则其对应的 Controller 则是：UserInfoController

2.6. 变量及方法命名

【强制】变量的名字必须用一个小写字母开头，后面的单词用大写字母开头。变量一般使用名词命名，方法名一般使用动词命名，并带有一定的意义，让人一读就懂；再者对于业界认可的单词或固有名词，可直接使用。

变量的命名包括实例变量，静态变量，函数参数的命名。

- 【强制】避免在命名中采用数字，除非命名意义明确，程序更加清晰，对实例变量的命名中不应该有数字
- 【推荐】变量名称是名词意义
- 【强制】采用有符合问题领域意义的单词或单词组合。第一个单词全部小写，后续的每个单词采用首字母大写，其余小写(特殊单词除外，如 URL)
- 【推荐】命名尽量简短，不要超过 16 个字符
- 【推荐】除了生命周期很短的临时变量外，避免采用单字符作为变量名，实例变量的命名不要用单字符。常用的单字符变量如整型用 i、j、 k、 m、 n 字符型用 c、d、 e，坐标用 x、y、z。
- 【推荐】变量尽量放在类的底端，使用块级注释说明该变量的作用

方法命名多数为动词结构

- 【推荐】采用有符合问题领域意义的单词或单词组合。第一个单词采用小写，后续的每个单词采用首字母大写，其余小写（特殊字除外如 URL），没有特别理由不用下划线作为分隔符
- 【强制】在 Java 中对属性方法命名遵循 JavaBean 的标准：

  1） getter 方法： get+属性名，对 boolean 型采用 is+属性名，有些特定的属性名用 has，can 代替 is 可能更好

  2） setter 方法： set+属性名
- 【强制】构造方法的命名与类名一致
- 【推荐】获取单个对象的方法以 get 做前缀
- 【推荐】获取多个对象的方法以 list 做前缀
- 【推荐】获取数量统计的方法以 count 做前缀
- 【推荐】保存方法以 save 做前缀
- 【推荐】修改方法以 update 做前缀

- 【推荐】删除方法以 delete 做前缀

2.7. 常量的命名

- 【强制】所有的字符都必须大写。采用有意义的单词组合表达，单词与单词之间以“下划线”隔开。
- 【推荐】命名尽量简短，不要超过 16 个字符

```
//  DBConfig PATH
public static final String DB_CONFIG_FILE_PATH ="com.glodon.gboat.dbconfig";
```

2.8. 数组的定义及命名

【强制】数组应该总是用下面的方式来命名：

```
byte[] buffer;   而不是： byte buffer[];
```

2.9. 方法的参数

【推荐】使用有意义的参数命名，如果可能的话，使用和要赋值的字段一样的名字：

```
setCounter(int size){
   this.size = size;
}
```

2.10. 内部循环变量的命名

- 【强制】请不要用 a、b、c、i、j、n 等没有意义的变量命名；
- 【推荐】不要怕麻烦，请使用带有意义的单词命名；如：userListSize、rowLength 等来命名。

3. java 注释规范

3.1. 基本规则

- 【推荐】注释应该使代码更加清晰易懂；
- 【推荐】注释要简单明了，只要提供能够明确理解程序所必要的信息就可以了。如果注释太复杂说明程序需要修改调整，使设计更加合理；
- 【推荐】注释不仅描述程序做了什么，还要描述为什么要这样做,以及约束；
- 【推荐】对于一般的 getter、setter 方法不用注释；

- 【推荐】注释不能嵌套；
- 【推荐】生成开发文档的需要用中文编写；
- 【推荐】如果需要注释的内容太多，需附加文档进行说明， 注释时加上“参见《****》”

3.2. Java 中有三种注释方式说明

不要写过于纯技术的注释，比如“去掉结尾的斜杠”等等，主要要写算法描述、业务场景等需要协助自己或者其他人理解。总之，写代码时换位思考，来个不了解我们业务的人也可以通过这些注解快速理解业务。

1） 文档注释 / */**

可以对用多行，一般用来对类、接口、成员方法、成员变量、静态字段、静态方法、常量进行说明。Javadoc 可以用它来产生代码的文档。为了可读性，可以有缩进和格式控制。

文档注释常采用一些标签进行文档的特定用途描述,用于帮助 Javadoc 产生文档,常用的有：

标签	Used for	目的
@author name	类/接口	描述代码的作者，每个作者对应一个这样的标签
@deprecated	类 成员方法	说明该段 API 已经被废除
@exception name description 或 @throws name description	成员方法	描述方法抛出的异常 每个异常一个对应一个这样的标签
@param name description	成员方法	描述成员方法中的参数用途和意义，一个参数对应一个这样的标签
@return description	成员方法	描述成员方法的返回值的意义
@since	类/接口 成员方法	描述该段 API 开始的时间
@see ClassName	类/接口 成员方法 成员变量	用于引用特定的类描述，一般 ClassName 用包括包名的全名
@see ClassName#memberfunction	类/接口 成员方法 成员变量	用于引用特定的类的成员方法的描述，一般 ClassName 用包括包名的全名
@version text	类/接口	版本
@inheritDoc	类/接口	继承的文档

	成员方法	

2） 行注释 //

【强制】一次只能注释一行，一般用来简短的描述某一个局部变量，程序块的作用。

3） 注释： /* */

【强制】可以用多行，一般用来对程序块、算法实现、类的实现进行说明。为了可读性，可以有缩进和格式控制。一般在行注释不能满足注释需要的时候采用。**一般用来作为文件头、复杂算法的说明，方法体内的复杂过程说明等；另外成员变量也用块注释来进行注释**。

3.3. 单行注释

【推荐】注释要简单明了，注释斜杠和内容之间有个空格，例如：

```
// 用户名
String userName = null;
```

3.4. 类注释

【推荐】类/接口描述，一般比较详细。按照常用的说明顺序排列，主要包括

- 类的主要说明，以“。”或“.”结束；
- 类设计的目标，完成什么样的功能；
- <Strong>主要的类使用</Strong>如何使用该类，包括环境要求,如是否线程安全,并发性要求，以及使用约束；
- 描述类的修改历史:<Strong>修改人+日期＋简单说明</Strong>；
- @author 作者、@version 版本，@see 参照,@since 开始版本等信息；

为了使形成的文档可读性好，注释中经常带有缩进和格式控制。类描述放在类的类定义的紧前面，不能有任何的空行。

```
/**
 *
 * 功能描述：写上你的描述，至少能看懂本类是做什么的，有哪些功能
 * <p>
 * <Strong>类使用说明</Strong>
 *<p>
* <Strong>修改历史<Strong>
 * @see
 *    与该类相关的类，写出具体的路径：包括完整的包名和类名.java
 *
```

```
 * @author (作者)  写上你的姓名
*
 * @since (该版本支持的 JDK 版本) ：  1.5
*
 * @date (开发日期)   写上编写日期
 *
 * @modify (修改)
*                       <p>第一次修改：时间、修改人;修改内容简介 </p>
*                       <p>第二次修改：时间、修改人;修改内容简介 </p>
*                       <p>第三次修改：时间、修改人;修改内容简介 </p>
 *
 */
```

3.5. 方法注释

【推荐】描述函数的功能，对成员方法，静态方法一般采用文档描述，特别是公开的方法。注释可以很详细，为了可读性强也可包含格式控制，如下面说明含有缩进：

方法注释一般包括:

- 方法的主要说明，以。或. 结束
- 描述方法完成什么样的功能, 方法的目标, 用该方法的原因
- 描述方法的使用方法, 包括使用的环境要求, 如前置条件, 后置条件和并发性要求
- @param c elements to be inserted into this list.（参数说明）
- @return <tt>true</tt> if this list changed as a result of the call.(返回值说明)
- @throws NullPointerException if the specified Collection is null.（异常说明）
- @see 如果参考的信息

```
/**
* 方法描述
* @param args array of string arguments
* @return No return value
* @exception exceptions No exceptions thrown
*/
public static void main(String[] args)  {
```

```
        System.out.println("Hello  world !");
    }
```

3.6. 变量注释

【强制】成员变量、类静态变量采用文档注释，对成员变量的注释通常包括：

1）变量的意义

2）变量的合法值域

3）对并发访问的限制

```
/**  XXXXXX */
String username = "xyz";
```

局部变量，如算法相关的变量采用块或行注释

```
String username = "xyz"; // 用户姓名
// 用户姓名
String username = "xyz";
```

【强制】参数变量注释一般用文档注释,并且用@param 来说明为参数,一般包括:

1）参数描述

2）对参数值范围的要求

3.7. 算法注释

【推荐】算法描述指在实现级别的描述注释，如在方法内的注释，对类功能的注释,这样使得程序更加易懂,方便程序算法的修改和 BUG 的修复。一般采用块/行注释，对于简短的描述采用行注释，不要用文档注释。注释的主要内容包括：

- 某些局部变量的意义和用途
- 复杂的控制结构的注释，如循环，分枝，条件表达式。说明控制所要达到的目标
- 复杂的代码段的描述，说明代码完成的功能，以及为什么这样做。
- 使用@ inheritDoc 开始

4. java 编码排版规范

4.1. 空格

- 【推荐】关键词和变量，变量和操作符之间加一个的空格

  ```
  Options opt1 = null;
  ```

- 【推荐】左小括号、右小括号和字符之间不出现空格

  if(a > b)，而不是 if(a > b)；

- 【推荐】任何二目、 三目运算符的左右两边都需要加一个空格，比如：=、&&、+ 、- 、*等

- 【推荐】方法参数在定义和传入时，多个参数逗号后边必须加空格

  是 method("a", "b", "c");，而不是 method("a","b","c");

4.2. 空行

【推荐】空行将逻辑相关的代码段分隔开，以提高可读性。

下列情况应该总是使用两个空行：

- 一个源文件的两个片段(section)之间
- 类声明和接口声明之间

下列情况应该总是使用一个空行：

两个方法之间

- 方法内的局部变量和方法的第一条语句之间
- 块注释或单行注释之前
- 一个方法内的两个逻辑段之间，用以提高可读性

4.3. 换行

【推荐】当一个表达式无法容纳在一行内时，可以依据如下一般规则断开之：

- 在一个逗号后面断开
- 在一个操作符前面断开
- 宁可选择较高级别(higher-level)的断开，而非较低级别(lower-level)的断开

4.4. 缩进

【强制】对不同级别缩进一个 TAB。

4.5. 声明

【强制】推荐一行一个声明，因为这样以利于写注释。亦即，

```
// indentation level
int level;


// size of table
int size;
```

而不是，

```
int level, size;
```

更不要将不同类型变量的声明放在同一行，例如：

```
int foo,  fooarray[];   // 错误写法
```

4.6. pom

- 【推荐】pom 中标签顺序按照 parent、当前组件的 groupId、artifactId、version、packaging、properties、dependencies、build 顺序排布
- 【推荐】dependencies 标签中，坐标从上到下按照业务组件、平台、第三方组件三组排列
- 【推荐】如果存在多个子标签，则按照 artifactId 的首字母顺序排列

5. 并发规范要求

- 【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。比如：资源驱动类、工具类、单例工厂类都需要注意
- 【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。例如：

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

- 【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。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

- 【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。例如： 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 SimpleDateFormat，官方给出的解释： simple beautiful strong immutable thread-safe。

- 【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。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
- 【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。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
- 【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。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
- 【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
- 【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 方法被执行到，避免主线程无法执行至 await 方法，直到超时才返回结果。注意，子线程抛出异常堆栈，不能在主线程 try-catch 到
- 【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。Random 实例包括 java.util.Random 的实例或者 Math.random()的方式。在 JDK7 之后，可以直接使用 API ThreadLocalRandom， 而在 JDK7 之前， 需要编码保正例：证每个线程持有一个实例
- 【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 Singleton {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) synchronized(this) {
            if (helper == null)
```

```
            helper = new Helper();
        }
        return helper;
    }
    // other methods and fields...
}
```

- 【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）
- 【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
- 【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享的，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

6. java 编码原则约定

6.1. 基本原则

- 【推荐】一个程序文件最好不要超过 2000 行
- 【推荐】尽可能缩小对象的作用域，这样对象的可见范围和生存期也都会尽可能地小
- 【推荐】一个方法所完成的功能要单一,不同的功能封装为不同的方法，方法代码行数如果超过 50 行，则考虑再抽取个新的方法；
- 【推荐】尽可能的处理异常或转换异常，不要一味的包装异常
- 【推荐】如果对象在某个特定范围内必须被清理（而不是作为垃圾被回收），请使用带有 finally 子句的 try 块，在 finally 子句中进行清理。
- 【推荐】对于把一些逻辑相关的类组织在一起，可以考虑把一个类的定义放在另一个类的定义中，这种情况推荐使用内部类（比如界面层中的事件响应等）。内部类拥有所有外围类所有成员的访问权。
- 【强制】对成员变量的访问最好通过 getter/setter 方法,这样能够保证访问的合法性,以及代码调整
- 【推荐】优先选择接口而不是抽象类或具体类。如果你知道某些东西将成为基类，你应当优先把它们设计成接口；只有在必须放进方法定义或成员变量时，才把它修改为具体或抽象类。接口只和客户希望的动作有关（协议），而类则倾向于关注实现细节。
- 【推荐】在处理可变 String 的时候要必须使用 StringBuffer 或者 StringBuilder 类，前者适合多线程安

全的情况，如果不存在线程安全考虑则使用后者，可变范围在 5 次以内可以使用 String 字符串连接。

- 【推荐】使用 java 标准库提供的容器。精通他们的用法，将极大地提高工作效率。例如：优先选择 ArrayList 来处理顺序结构，选择 HashSet 来处理集合，选择 HashMap 来处理关联数组，选择 LinkedList 来处理堆栈和队列，它对顺序访问进行了优化，向 List 中间插入与删除的开销小，但随机访问则较慢。当使用前三个的时候，应该把他们向上转型为 List、Set 和 Map，这样就可以在必要的时候以其它方式实现。尽量优先选择 Apache 的 common 工具包，例如在做字符串为空判断时使用 StringUtils.isEmpty，做类似于 bean 操作的时候可以使用 BeanUtils 等。
- 【推荐】数组是一种效率最高的存储和随机访问对象引用序列的方式，但是当创建了一个数组对象，数组的大小就被固定了，如果在空间不足时再创建新的数组进行复制，这样效率就比 ArrayList 开销大了。所以必须明确使用场景。
- 【强制】尽量使用”private”、”protected”关键字。一旦你把库的特征（包括类、方法、字段）标记为 public，你就再也不可能去掉他们。在这种方式下，实现的变动对派生类造成的影响最小，在处理多线程问题的时候，保持私有性尤其重要，因为只有 private 的字段才会受到保护，而不用担心被未受同步控制的使用所破坏。
- 采用类而不是对象引用静态变量和方法。
- 【推荐】使用 Boolean 必须采用 Boolean.TRUE/Boolean.FALSE 进行构造，Boolean.valueof(true|false|y|n|Y|N)的形式进行构造。
- 【推荐】禁止后台业务代码使用如下代码，要输出日志，同时进行异常包装抛出。

```
try {
    something();
} catch (Exception ex){
    e.printStackTrace();
}
```

- 【推荐】类型尽量都使用包装类型，比如布尔类型，采用 Boolean，而不是 boolean，尽量采用 Integer，而不是 int。

6.2. 类编写规范

- 【强制】所有 PO 和 VO 都要实现 Seralizable；（可能需要放入分布式缓存）
- 【推荐】类的结构组织，一般按照如下的顺序：

1. 构造函数部分
2. finalize 部分
3. 成员方法部分

4. 静态方法部分

5. 成员变量声明

6. 静态变量声明

7. 常量声明

这种顺序是推荐的，在实际开发中可以按照一定的尺度修改，原则是程序更易读。如对方法的排序按照重要性，或按照字母顺序排列或按照方法之间的关系排列。

【推荐】每个方法（包括构造与 finalize）都是一个段。多个变量声明按照逻辑共同组成一个段，段与段之间以空行分隔。

【强制】类声明时，要指出其访问控制，一般为没有修饰符、public 和 private。

【推荐】方法与方法之间，都需要以空行隔离。

【推荐】主要的方法放在类的上方，其他辅助的方法放类的下方。public 的方法放在上方，privated 的方法放在下方，get/set 方法放在最后，因为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； 因为承载的信息价值较低，所有 Service 和 DAO 的 getter/setter 方法放在类体最后

【强制】编写通用性的类时，请遵守标准形式。包括定义 equals()、hasCode()、toString()、Clone(实现 Cloneable 接口)，并实现 Comparable 和 Serialiable 接口。

【推荐】对于设计期间不需要继承的类，尽量使用 final。

6.3. 变量

- 【强制】对成员变量，尽量采用 private。
- 【强制】每一个变量声明/定义占一行（参数变量除外），如：

```
Integer a;
Integer b;
```

比 Integer a,b; 更容易读，更容易查找 bug。

- 【强制】局部变量在使用前必须初始化，一般在声明时初始化。
- 【推荐】变量声明的位置尽量保证生命周期短，声明的位置尽量靠近使用的位置。

如：

```
private void myMethod() {
    int int1 = 0; // beginning of method block
    if (condition) {
        int int2 = 0; // beginning of "if" block
        ...
```

```
    }
}
```

【推荐】一种例外情况是在 for 语句中，定义声明不仅不占一行，还在表达式内部，完全采用 Eclipse 生成, 如：

```
for(int i = 0; i<100; i++)
```

- 【强制】数组的申明采用 <数据类型[] + 变量名>方式如：

```
char[] buffer;
```

而不是：

```
char buffer[];
```

6.4. 常量

- 【推荐】常量的复用层次有四层：跨应用共享常量、跨组件共享常量、组件内共享常量、类内共享常量。
  1） 跨应用共享常量：需要单独抽取一个组件放静态类，该组件被两个应用共用
  2） 跨组件共享常量：需要单独抽取一个组件放静态类，该组件被两个应用共用
  3） 组件内共享常量：在该组件下创建静态类
  4） 类内共享常量：直接在类内部 private static final 定义。
- 【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。比如：

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

6.5. 方法

- 【强制】对成员方法, 不要轻易的采用 public 的成员变量。
- 【强制】空方法中方法声明和函数体可都在一行。如：private void func(){}。
- 【强制】方法和方法之间空一行。
- 【强制】方法的文档注释放在方法的紧前面，不能空一行。
- 【推荐】避免过多的参数列表，尽量控制在 5 个以内，若需要传递多个参数时，当使用一个容纳这些参数的对象进行传递，以提高程序的可读性和可扩展性。
- 【强制】方法中的循环嵌套不能超过 3 层。
- 【强制】对于设计期间不需要子类来重载的方法，尽量使用 final。
- 【推荐】每个方法尽量代码行数尽量不要超过 50 行（有效代码行，不包括注释），但必须保证逻辑的完整

性。

6.6. 注解

- 【强制】PO 的@Table、@Column、@Id 注解统一使用 javax.persistence 包下的
- 【强制】PO 的 ID 生成策略统一使用 Hibernate 的 uuid.hex 格式
- 【强制】PO 的@Column 注解都在 get 方法上
- 【强制】PO 类的@Comment 注解放在类上方，字段的@Comment 注解在该字段上
- 【强制】PO 上关于数据库表、字段都是大写，单词之间用英文下划线分割
- 【强制】如果是业务的 Service、Business 使用 Spring 的@Service 注解
- 【强制】如果没有业务逻辑，而是平台扩展或者第三方组件扩展使用@Component
- 【强制】所有覆写的方法上都必须加上@Override 注解

6.7. OOP

- 【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
- 【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
- 【强制】不能使用过时的类或方法。

  说明： java.net.URLDecoder 中的方法 decode(String encodeStr) 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
- 【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
-

6.8. 其他

- 【强制】避免变量的定义与上一层作用域的变量同名。
- 【强制】局部变量在使用时刻声明，局部变量/静态变量在声明时同时初始化。
- 【推荐】在与常数作比较时常数放在比较表达式的前面如：

```
if("simpleCase".equals(obj))…
```

```
if(null == obj)….
```

- 【强制】return 语句中，不要有复杂的运算。
- 【强制】switch 语句，需要一个缺省的分支。
- 【强制】for、if、try 等嵌套不要超过 3 层，如果超出了考虑提取新的方法

7. 日志

- 【强制】严禁在代码中使用 System.out 的方式输出日志到控制台
- 【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(ProjectController.class);
```

- 【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

反例说明：

```
logger.debug("Processing trade with id: " + id + " and symbol: " + symbol);
```

如果日志级别是 warn，上述日志虽然不会打印，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 + " and symbol: " + symbol);
}
```

正例：（占位符）

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

- 【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

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

- 【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？**

8. 异常

- 【推荐】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。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。

  是： if (obj != null) {...}

  而不是： `try { obj.method() } catch (NullPointerException e) {...}`
- 【强制】不允许对大段代码进行 try-catch，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。
- 【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（吃掉异常），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。
- 【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。如果 JDK7 及以上，可以使用 try-with-resources 方式
- 【强制】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； Controller 提供前端的服务调用优先考虑使用平台封装的 Result 方式。Controller 不允许直接抛出异常，对外的微服务也不允许直接抛出异常，而需要进行业务包装
- 【推荐】**各个产品、项目需要根据自己的业务需要，定义全局异常处理**

9. 数据库编码命名规范

9.1. 大小写说明

有关数据库的命名都是用大写。

9.2. 数据库表命名

数据库表的命名采用如下规则：

1)表名用 T_开头，表名长度不能超过 30 个字符，表名中含有单词要大写。如果表为表之间的关系表采用 R_开头。

所有配置相关的以 C_开头，例如：

```
gb_T_***  //中间字母'T'表示业务表
gb_C_***  //中间字母'C'表示字典表
gb_R_***  //中间字母'R'表示关联表
```

具体说明：

数据库命名规则=子系统关键字头+业务表类型+业务表名称

每个子系统都有自己的命名关键字“头”：

平台系统 G3_
工程交易 GB_(Bidding Management)
政府采购 GP_
产权 GC_
土矿 LE_
竞价系统 AUS_
主体库 GFM_(Faithfulness Management)
场地系统 GVM_(Venue Management)
交易资金系统 GTM_ (Trading Money Management)
电子档案 GA_(Archives Management)
专家系统 GE_(Expert Management)
电子监察 GES_(Electronic Supervision Management)
数据同步系统 GSD_(Synch Data Management)

2）表中含有的单词建议用完整的单词。**如果导致表名长度超过 30 个字符**，则从最后一个单词开始，依次向前采用该单词的缩写。

例如：SSO2_T_USER （系统 SSO2 的用户表）

9.3. 视图命名

原则上严禁使用视图，如果必须，需要经过所在项目、产品的研发团队评审

9.4. 函数/存储过程

原则上严禁使用存储过程，如果必须，需要经过所在项目、产品的研发团队评审

9.5. 字段名命名

- 以英文名命名；
- 对于多个单词组合的情况，以“_”分隔；
- 单词长度大的使用标准简称；
- **字段名应该在 30 字母以内；**
- 示例：USER_NAME（用户名）
  USER_PWD（用户密码）

9.6. 主键命名

主键的命名采用如下规则：

主键名用_ID 结尾，前缀为该主键所在的表名(去掉“T_系统编码_”)。主键名长度不能超过 18 个字符。如果过长，可对表名进行缩写。缩写规则同表名的缩写规则。

9.7. 外键命名

和别的 PO 中的主键名称保持一致

9.8. 索引

索引的命名采用如下规则：
1）索引名用小写的英文字母和数字表示。索引名的长度不能超过 30 个字符。
2）主键对应的索引和主键同名。
3）每类索引都用_结束。
4）唯一性索引用 uni_开头，后面跟表名。一般性索引用 idx_开头，后面跟表名。
5）如果索引长度过长，可对表名进行缩写。缩写规则同表名的缩写规则。

9.9. 表间约束关系

表间约束关系不在数据库层面考虑，通过代码维护数据完整性

10. 数据库 SQL 语句格式

10.1. 大小写说明

SQL 语句必须全部用大写字母编写。

10.2. INSERT INTO 语句

```
INSERT INTO 表名(字段 1, 字段 2, 字段 3)
VALUES(值 1, 值 2, 值 3)
```

注意要求：

- 第一行为：INSERT INTO 表(表字段)
- 第二行为：VALUES(字段所对应的值)
- “,”后请打一个空格

10.3. UPDATE 语句

```
UPDATE 表名
  SET 字段 1 = 值 1, 字段 2 = 值 2, 字段 3 = 值 3
WHERE
    条件 1
  AND(OR) 条件 2
  AND(OR) 条件 3
```

注意要求：

- 第一行 ：UPDATE 表名
- SET 设置字段值（注：如果太长，请换行）
- “=”两头请都打空格
- “,”后请打一个空格
- 若带条件，单独一行写 WHERE
- 空四个空格符，写上第一个条件

- 若带多个条件，换行，敲两个空格，写 AND 条件 N
- 一行只写一个 AND 条件

10.4. SELECT 语句

```
SELECT
    字段 1,
    字段 2,
    字段 3
FROM   表 1, 表 2, 表 3
WHERE
    条件 1
  AND(OR) 条件 2
  AND(OR) 条件 3
ORDER BY 排列字段
GROUP BY 分组字段 HAVING 过滤条件
UNION[ALL]
SELECT ……
```

或者(SQL 标准的关联查询写法)：

```
SELECT
    字段 1,
    字段 2
    字段 3
FROM   表 1
INNER(LEFT、RIGHT 、FULL) JOIN 表 1 ON 条件
INNER(LEFT、RIGHT 、FULL) JOIN 表 2 ON 条件
CROSS JION 表 3 (注：交叉查询是不带 ON 条件的)
WHERE  条件 1
  AND(OR) 条件 2
  AND(OR) 条件 3
ORDER BY 排列字段
GROUP BY 分组字段 HAVING 过滤条件
UNION[ALL]
SELECT ……
```

注意要求：

请注意以上的两种格式，该换行的请换行、该空格的请空格！建议采用第二种格式写法(相关的链接查询请正确的理解)

知识提醒：

INNER JOIN 表 ON 条件

同等于 oracle 中的 T1.A = T2.A

LEFT JOIN 表 1 ON 条件

同等于 oracle 中的 T1.A = T2.A(+)

RIGHT JOIN 表 1 ON 条件

同等于 oracle 中的 (+)T1.A = T2.A

11. 查询优化说明

1) 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2) 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：
select id from t where num is null
可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：
select id from t where num=0

3) .应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

4) .应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：
select id from t where num=10 or num=20
可以这样查询：
select id from t where num=10
union all
select id from t where num=20

5) in 和 not in 也要慎用，否则会导致全表扫描，如：
select id from t where num in(1,2,3)
对于连续的数值，能用 between 就不要用 in 了：
select id from t where num between 1 and 3

6) 下面的查询也将导致全表扫描：
select id from t where name like '%abc%'
若要提高效率，可以考虑全文检索。

7) 如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：
select id from t where num=@num
可以改为强制查询使用索引：
select id from t with(index(索引名)) where num=@num

8) 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：
select id from t where num/2=100
应改为:
select id from t where num=100*2

9) 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：
select id from t where substring(name,1,3)='abc'--name 以 abc 开头的 id
select id from t where datediff(day,createdate,'2005-11-30')=0--‘2005-11-30’生成的 id
应改为:
select id from t where name like 'abc%'
select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'

10) 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11) 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12) 不要写一些没有意义的查询，如需要生成一个空表结构：
select col1,col2 into #t from t where 1=0
这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：
create table #t(...)

13) 很多时候用 exists 代替 in 是一个好的选择：
select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

14) 并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，male、female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

15) 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

16) 应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17) 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

18) 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

19) 任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

20) 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

21) 避免频繁创建和删除临时表，以减少系统表资源的消耗。

22) 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

23) 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。

24) 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25) .尽量避免大事务操作，提高系统并发能力。

26) 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

12. 跨数据库

- 优先使用技术平台提供的 VO 查询，如果无法满足需要考虑是否 PO 划分的不够科学，考虑使用 HQL 查询，如果 HQL 还是不行，再考虑使用 SQL
- SQL 需要是标准 SQL，如果不是则需要做数据库判断，暂时不支持的数据库需要抛出异常；