**智慧海关应用云平台最佳实践及关键技术方案**

目录

[1 前言 7](#_Toc522392335)

[1.1 文档介绍 7](#_Toc522392336)

[1.1.1 文档目的 7](#_Toc522392337)

[1.1.2 预期读者 8](#_Toc522392338)

[1.1.3 名词解释 9](#_Toc522392339)

[2 工程结构规约 11](#_Toc522392340)

[2.1 工程结构 11](#_Toc522392341)

[2.2 依赖库 13](#_Toc522392342)

[2.3 服务器 15](#_Toc522392343)

[3 编程规约（Java） 16](#_Toc522392344)

[3.1 源文件结构 16](#_Toc522392345)

[3.2 命名风格 17](#_Toc522392346)

[3.3 常量定义 20](#_Toc522392347)

[3.4 代码格式 22](#_Toc522392348)

[3.5 OOP规约 23](#_Toc522392349)

[3.6 集合处理 27](#_Toc522392350)

[3.7 并发处理 29](#_Toc522392351)

[3.8 控制语句 31](#_Toc522392352)

[3.9 注释规约 33](#_Toc522392353)

[3.10 其他 34](#_Toc522392354)

[4 编程规约（JavaScript） 35](#_Toc522392355)

[4.1 函数声明 35](#_Toc522392356)

[4.2 函数表达式 36](#_Toc522392357)

[4.3 调用函数 38](#_Toc522392358)

[4.4 函数作用域 40](#_Toc522392359)

[4.5 作用域和函数堆栈 42](#_Toc522392360)

[4.5.1 递归 42](#_Toc522392361)

[4.5.2 嵌套函数和闭包 44](#_Toc522392362)

[4.5.3 保存变量 46](#_Toc522392363)

[4.5.4 多层嵌套函数 46](#_Toc522392364)

[4.5.5 命名冲突 47](#_Toc522392365)

[4.6 arguments 对象 48](#_Toc522392366)

[4.7 函数参数 50](#_Toc522392367)

[4.7.1 默认参数 50](#_Toc522392368)

[4.7.2 Link to section剩余参数 51](#_Toc522392369)

[4.8 箭头函数 51](#_Toc522392370)

[4.8.1 更简洁的函数 51](#_Toc522392371)

[4.8.2 this 的词法 52](#_Toc522392372)

[4.9 运算符 54](#_Toc522392373)

[4.9.1 赋值运算符 55](#_Toc522392374)

[4.9.2 比较运算符 57](#_Toc522392375)

[4.9.3 Link to section算术运算符 59](#_Toc522392376)

[4.9.4 位运算符 60](#_Toc522392377)

[4.9.5 逻辑运算符 64](#_Toc522392378)

[4.9.6 字符串运算符 66](#_Toc522392379)

[4.9.7 条件（三元）运算符 66](#_Toc522392380)

[4.9.8 逗号操作符 66](#_Toc522392381)

[4.9.9 一元操作符 67](#_Toc522392382)

[4.9.10 关系操作符 71](#_Toc522392383)

[4.9.11 运算符优先级 73](#_Toc522392384)

[4.10 部分主要预定义函数 74](#_Toc522392385)

[5 测试规约 76](#_Toc522392386)

[6 异常规约 79](#_Toc522392387)

[6.1 异常处理 79](#_Toc522392388)

[6.2 日志规约 81](#_Toc522392389)

[7 安全规约 82](#_Toc522392390)

[8 规约执行 83](#_Toc522392391)

[9 工具推荐 83](#_Toc522392392)

[9.1 编译文件检测工具 84](#_Toc522392393)

[9.2 源文件语法类检测工具 84](#_Toc522392394)

[9.3 源文件格式类检测工具 86](#_Toc522392395)

**版本历史**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 更新日期 | 更新说明 | 更新人员 | 备注 |
| 1.0.0 | 2018.5.18 | 文档创建 | AC |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 前言

## 文档介绍

### 文档目的

随着IT行业的高速发展以及企业快速的数字化转型，软件已经成为一个企业的重要资产。纵观一个软件整个生命周期，其80%的时间在维护过程中，而由于软件行业人员流动频繁，导致大部分软件不会由最初人员负责维护。因此软件的易读性对于软件的交付、维护、安全有决定性作用。

本文档以Java 开发者为中心视角，从软件易读性、可维护性、安全性、美观角度出发，整体划分为工程结构规约、编程规约、测试规约、异常规约、安全规约、工具推荐六个维度，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**军规、建议、参考**三大类。

**【军规】**：规约对系统有重大故障影响，必须强制执行

**【建议】**：规约约束力一般对系统有一定影响，开发过程中尽量执行

**【参考】**：规约仅为参考，不做强制性约束

“无规矩不成方圆”，本手册旨在形成企业内部统一的编码规范，尽量减少各类“无边际”的个性化编码，加强各团队的协同开发，提升工作效率码出高效。

**适用范围**

本文档包括智慧海关应用云平台涉及到的所有内容。

### 预期读者

本文档适用于系统项目组所有相关人员，包括项目管理人员、需求分析人员、架构设计人员、开发人员、测试人员、用户代表，阅读前应充分理解软件架构和开发流程。

#### 参考文档

#### 编制依据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参考文档(SP11) | | 发布单位 | 参考类别(PA061) | 其它说明 |
| 编号 | 名称 |
| 1 | 中国海关科技应用项目管理办法 | 海关总署科技司 | 强制执行标准 |  |
| 2 | 海关科技应用项目交办承办实施管理办法 | 海关总署科技司 | 执行标准 |  |
| 3 | H2018工程业务分析报告 |  |  |  |
| 4 | H2018工程技术需求报告 |  |  |  |
| 5 | H2018工程总体方案 | 海关总署科技司 |  |  |
|  |  |  |  |  |

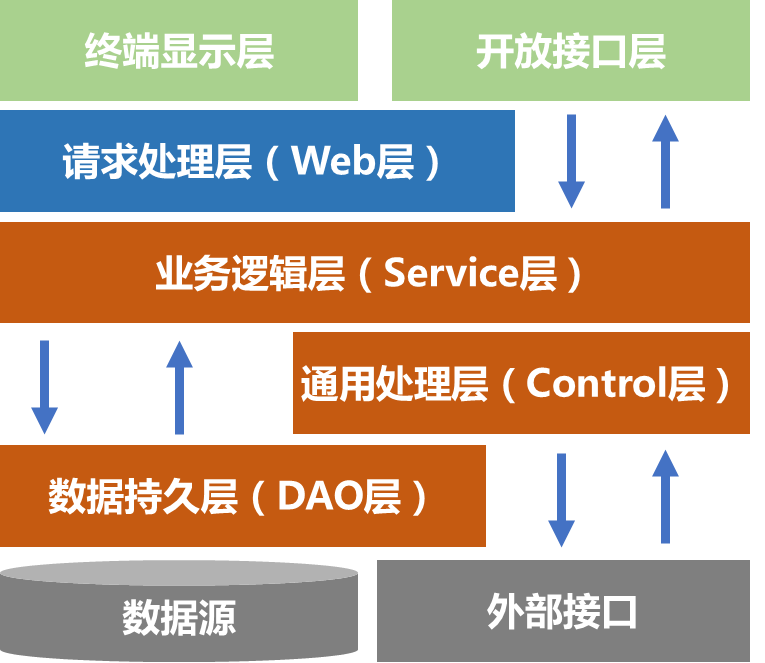
### 名词解释

|  |  |  |
| --- | --- | --- |
| 序号 | 名词 | 解释 |
| 1 | POJO | POJO（Plain Ordinary Java Object）专指只有setter / getter / toString的简单类，包括DO/DTO/BO/VO等。 |
| 2 | GAV | （GroupId、ArtifactctId、Version）,Maven坐标，是用来唯一标识jar包。 |
| 3 | OOP | OOP（Object Oriented Programming）泛指类、对象的编程处理方式。 |
| 4 | ORM | 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。 |
| 5 | NPE | NPE（java.lang.NullPointerException）空指针异常 |
| 6 | SOA | 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。 |
| 7 | 内置库 | 本工程内部子项目模块依赖的库（jar包）。 |
| 8 | 依赖库 | 公司内部发布到制件库，可供公司内部其它应用依赖的库（jar包）。 |
| 9 | 三方库 | 公司之外的第三方开源库（jar包）。 |
| 10 | IDE | IDE（Integrated Development Environment）用于提供程序开发环境的应用程序，一般包括代码编辑器、编译器、调试器和图形用户界面等工具，本《手册》泛指IntelliJ IDEA和eclipse。 |

# 工程结构规约

## 工程结构

1. **【建议】**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的层。当前主要是velocity渲染，JS渲染，JSP渲染，移动端展示等。

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。

Control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，对多个DAO的组合复用。

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

外部接口：包括其它部门的开放接口。

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Control/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。如果Control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层不可继续往上抛异常，因为已经处于顶层，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，加上用户容易理解的错误提示信息。

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或Control向外传输的对象。

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

AO（Application Object）：应用对象。在Web层与Service层之间抽象的复用对象模型，极为贴近展示层，复用度不高。

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。

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

## 依赖库

1. 【军规】定义GAV遵从以下规则：

1） GroupID格式：com.{公司}.业务项目.[子业务项目]，最多4级。

**正例**：com.*company*.devops 或 com.*company*.devops.autotest。

2） ArtifactID格式：业务项目名-模块名。语义不重复不遗漏，先到制件库查证一下。 正例：devops-api / devops-tool。

3） Version：详细规定参考下方。

2. 【军规】依赖库版本号命名方式：主版本号.次版本号.修订号

1） 主版本号：产品方向改变，或者大规模API不兼容，或者架构不兼容升级。

2） 次版本号：保持相对兼容性，增加主要功能特性，影响范围极小的API不兼容修改。

3） 修订号：保持完全兼容性，修复BUG、新增次要功能特性等。 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制件库仓库进行查证，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. 【军规】线上应用不可依赖第三方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。

4. 【军规】依赖库的新增或升级，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. 【军规】依赖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. 【军规】依赖于一个依赖库群时，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. 【军规】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。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. 【建议】所有pom文件中的依赖声明放在<dependencies>语句块中，所有版本放在<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. 【建议】依赖库不要有配置项，最低限度不要再增加配置项。

10. 【参考】为避免应用依赖库的依赖冲突问题，依赖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。如果依赖其它依赖库，尽量是provided引入，让依赖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，依赖库由谁维护，源码在哪里，都需要能方便查到。除非用户主动升级版本，否则公共依赖库的行为不应该发生变化。

## 服务器

1. 【建议】高并发服务器建议调小TCP协议的time\_wait超时时间。

**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. 【建议】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“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. 【建议】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. 【建议】在线上生产环境，JVM的Xms和Xmx设置一样大小的内存容量，避免在GC 后调整堆大小带来的压力。

5. 【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。

# 编程规约（Java）

## 源文件结构

一个源文件包含(按照顺序)：

1. 许可证或版权信息(如有需要)

2. package语句

3. import语句

4. 一个顶级类(只有一个)

以上每个部分之间用一个空行隔开。

1. 【军规】许可证或版权信息应当被放在文件最前面。

2. 【军规】package语句不换行，列限制并不适用于package语句。

示例：com.chinacustom.infocenter.（折行）

tongguan2.connection.json

3. 【军规】import不要使用通配符，不要出现类似这样的import语句：

示例： import java.util.\*;。

4. 【军规】import语句不换行，列限制并不适用于import语句。(每个import语句独立成行)。

5. 【军规】import语句可分为以下几组，按照这个顺序，每组由一个空行分隔

示例：

1)所有的静态导入独立成组

2)cn.gov.customs imports(仅当这个源文件是在 cn.gov.customs 包下)

cn.gov.customs

3)第三方的包。每个顶级包为一组，字典序。例如：android, com, junit, org, sun

组内不空行，按字典序排列。

6. 【军规】只有一个顶级类声明,每个顶级类都在一个与它同名的源文件中(当然，还包含 .java 后缀)。

## 命名风格

1. 【军规】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. 【军规】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：huawei/ shenzhen/ beijing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. 【军规】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. 【军规】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. 【军规】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。

**正例**：MAX\_STOCK\_COUNT **反例**：MAX\_COUNT

6. 【军规】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Tests结尾。

7. 【军规】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. 【军规】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. 【军规】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.company.open.util、类名为MessageUtils（此规则参考spring的框架结构）

10. 【军规】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。

11. 【建议】为了达到代码自解释的目标，任何定义编程元素在命名时使用尽量完整单词 组合来表达其意。

**正例**： 从远程仓库拉取代码的类命名为 PullCodeFromRemoteRepository。

**反例**： 变量int a;的随意命名方式。

12. 【建议】如果模块、接口、类、方法使用了设计模式，在命名时体现出具体模式。

13**. 【建议】**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 = "China Customs";

**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。

14. 接口和实现类的命名有两套规则：

1）**【军规】**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）**【建议】** 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。

15. **【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**：枚举名字为ProcessStatusEnum的成员名称：SUCCESS / UNKOWN\_REASON。

16. 【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) 领域模型命名规约 1） 数据对象：xxxDO，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。

## 常量定义

1. **【军规】**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#taobao\_" + tradeId; cache.put(key, value);

2. **【军规】**long或者Long初始赋值时，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. **【建议】**不要使用一个常量类维护所有常量，按常量功能进行归类，分开维护。 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。

**正例**：缓存相关常量放在类CacheConsts下；系统配置相关常量放在类ConfigConsts下。

4. **【建议】**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：放置在依赖库中，通常是client.jar中的constant目录下。

2） 应用内共享常量：放置在内置库中，通常是modules中的constant目录下。

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. **【建议】**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);}

## 代码格式

1. 【军规】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；表示终止的右大括号后必须换行。

2. 【军规】 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。

反例：if (空格a == b空格)

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

4. 【军规】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. 【军规】采用2个空格缩进，禁止使用tab字符。

说明： 如果使用 tab 缩进，必须设置缩进，必须设置 1个 tab 为 2个空格。 IDEA 设置 tab 为 2个空格时， 请勿勾选 Use tab character ；而在 eclipse 中，必须勾选 insert spaces for tabs 。

6. 【军规】注释的双斜线与注释内容之间有且仅有一个空格。

正例：// 注释内容，注意在//和注释内容之间有一个空格。

7. 【军规】单行字符数限不超过 120 个，超出需要换行时，遵循如下原则： 1） 第二行相对一缩进 2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） 方法调用时，多个参数，需要换行时，在逗号后进行。

5） 在括号前不要换行。

8. 【军规】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");

9. 【军规】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。

10. 【建议】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。

11. 【建议】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。

## OOP规约

1. 【军规】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. 【军规】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. 【军规】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。

4. 【军规】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. 【军规】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。

6. 【军规】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. 【军规】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。

8. 关于基本数据类型与包装数据类型的使用标准如下：

1）【军规】所有的POJO类属性必须使用包装数据类型。

2）【军规】RPC方法的返回值和参数必须使用包装数据类型。

3）【建议】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，页面显示为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. 【军规】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. 【军规】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. 【军规】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. 【军规】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. 【建议】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。

14. 【建议】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，此条规则优先于第15条规则。

15. 【建议】 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。

16. 【建议】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。

17. 【建议】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。

18. 【建议】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： 1） 不允许被继承的类，如：String类。 2） 不允许修改引用的域对象，如：POJO类的域变量。 3） 不允许被重写的方法，如：POJO类的setter方法。 4） 不允许运行过程中重新赋值的局部变量。 5） 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. 【建议】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. 【建议】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，无限制的到处跑，那么你会担心的。

## 集合处理

1. 【军规】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。 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. 【军规】 ArrayList的subList结果不可强转成ArrayList，否则会抛出ClassCastException异常，即java.util.RandomAccessSubList cannot be cast to java.util.ArrayList。

3. 【军规】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、删除均会产生ConcurrentModificationException 异常。

4. 【军规】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()。

5. 【军规】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。

6. 【军规】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：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>。

7. 【军规】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。

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

9. 【建议】集合初始化时，指定集合初始值大小。

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

11. 【建议】高度注意Map类集合K/V能不能存储null值的情况。

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。

## 并发处理

1. 【军规】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. 【军规】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。

3.【军规】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. 【军规】线程池不允许使用 【军规】线程池不允许使用 Executors去创建，而是通过 去创建，而是通过 去创建，而是通过 ThreadPoolExecutor的方式。

5. 【军规】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。

6. 【军规】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. 【军规】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. 【军规】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. 【军规】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. 【建议】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。 说明：注意，子线程抛出异常堆栈，不能在主线程try-catch到。

11. 【建议】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. 【建议】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。

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。

## 控制语句

1. 【军规】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. 【军规】在if/else/for/while/do语句中必须使用大括号。即使只有一行代码，避免采用单行的编码方式。

3. 【建议】表达异常的分支时，少用if-else方式。

4. 【建议】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。

5. 【建议】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。

6. 【建议】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： 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。

8. 【参考】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。但在方法说明里必须注明外部参数检查要求。 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。

## 注释规约

1. 【军规】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. 【军规】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. 【军规】所有的类都必须添加创建者和创建日期。

4. 【军规】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. 【军规】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. 【建议】特定场景直接使用中文注释。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”。

7. 【建议】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。

8. 【参考】谨慎注释掉代码。在上方详细说明，而不是简单地注释掉。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

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。

11. 【参考】特殊注释标记，请注明标记人与标记时间。

## 其他

1. 【军规】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。

2. 【军规】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。

3. 【军规】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。

4. 【军规】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。

5. 【建议】不要在视图模板中加入任何复杂的逻辑。

6. 【建议】任何数据结构的构造或初始化，都应指定大小，避免内存过度消耗。

7. 【建议】及时清理不再使用的代码段或配置信息。

说明：对于垃圾代码或过时配置，坚决清理干净，避免程序过度臃肿，代码冗余。

# 编程规约（JavaScript）

## 函数声明

**【军规】一个函数定义（也称为函数声明，或函数语句）由一系列的**[**function**](https://developer.mozilla.org/zh-CN/docs/JavaScript/Reference/Statements/function)**关键字组成**

**构造：**

* 函数的名称。
* 函数参数列表，包围在括号中并由逗号分隔。
* 定义函数的 JavaScript 语句，用大括号{}括起来。

例如，以下的代码定义了一个简单的square函数：

function square(number) {

return number \* number;

}

函数square使用了一个参数，叫作number。这个函数只有一个语句，它说明该函数将函数的参数（即number）自乘后返回。函数的[return](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/return)语句确定了函数的返回值。

return number \* number;

原始参数（比如一个具体的数字）被作为**值**传递给函数；值被传递给函数，如果被调用函数改变了这个参数的值，这样的改变不会影响到全局或调用函数。

如果你传递一个对象（即一个非原始值，例如[Array](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Array)或用户自定义的对象）作为参数，而函数改变了这个对象的属性，这样的改变对函数外部是可见的，如下面的例子所示：

function myFunc(theObject) {

theObject.make = "Toyota";

}

var mycar = {make: "Honda", model: "Accord", year: 1998};

var x, y;

x = mycar.make; // x获取的值为 "Honda"

myFunc(mycar);

y = mycar.make; // y获取的值为 "Toyota"

// (make属性被函数改变了)

## 函数表达式

**【军规】函数声明在语法上是一个语句，但函数也可以由函数表达式创建。函数可以是匿名**。

例如，函数square也可这样来定义：

var square = function(number) { return number \* number; };

var x = square(4); // x gets the value 16

函数表达式也可以提供函数名，可以用于在函数内部代指其本身，或在调试器堆栈跟踪中识别该函数：

var factorial = function fac(n) {return n<2 ? 1 : n\*fac(n-1)};

console.log(factorial(3));

将函数作为参数传递给另一个函数时，函数表达式很方便。如下演示了一个叫map的函数如何被定义，而后使用一个表达式函数作为其第一个参数进行调用：

function map(f,a) {

var result = [], // 创建一个新的数组

i;

for (i = 0; i != a.length; i++)

result[i] = f(a[i]);

return result;

}

代码：

function map(f, a) {

var result = []; // 创建一个数组

var i; // 声明一个值，用来循环

for (i = 0; i != a.length; i++)

result[i] = f(a[i]);

return result;

}

var f = function(x) {

return x \* x \* x;

}

var numbers = [0,1, 2, 5,10];

var cube = map(f,numbers);

console.log(cube);

返回 [0, 1, 8, 125, 1000]。

在 JavaScript 中，可以根据条件来定义一个函数。比如下面的代码，当num 等于 0 的时候才会定义 myFunc ：

var myFunc;

if (num == 0){

myFunc = function(theObject) {

theObject.make = "Toyota"

}

}

除上述的定义函数方法外，在运行时用 [Function](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Function) 构造器由一个字符串来创建一个函数 ，类似 [eval()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/eval()) 函数。

当一个函数是一个对象的属性时，称之为**方法**。了解更多关于对象和方法的知识 [使用对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects)。

## 调用函数

【军规】定义一个函数并不会自动的执行它。定义了函数仅仅是赋予函数以名称并明确函数被调用时该做些什么。**调用**函数才会以给定的参数真正执行这些动作。

例如，一旦你定义了函数square，你可以如下这样调用它：

square(5);

上述语句通过提供参数 5 来调用函数。函数执行完它的语句会返回值25。

函数一定要处于调用它们的域中，但是函数的声明可以被提升(出现在调用语句之后)，如下例：

console.log(square(5));

/\* ... \*/

function square(n) { return n\*n }

函数域是指函数声明时的所在的地方，或者函数在顶层被声明时指整个程序。

**提示：**注意只有使用如上的语法形式（即 function funcName(){}）才可以。而下面的代码是无效的。就是说，函数提升仅适用于函数声明，而不适用于函数表达式。

console.log(square); // square is hoisted with an initial value undefined.

console.log(square(5)); // TypeError: square is not a function

var square = function (n) {

return n \* n;

}

函数的参数并不局限于字符串或数字。你也可以将整个对象传递给函数。函数 show\_props（其定义参见 [用对象编程](https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Working_with_Objects#Objects_and_Properties)）就是一个将对象作为参数的例子。

函数可以被递归，就是说函数可以调用其本身。例如，下面这个函数就是用递归计算阶乘：

function factorial(n){

if ((n == 0) || (n == 1))

return 1;

else

return (n \* factorial(n - 1));

}

你可以计算1-5的阶乘如下：

var a, b, c, d, e;

a = factorial(1); // 1赋值给a

b = factorial(2); // 2赋值给b

c = factorial(3); // 6赋值给c

d = factorial(4); // 24赋值给d

e = factorial(5); // 120赋值给e

还有其它的方式来调用函数。常见的一些情形是某些地方需要动态调用函数，或者函数的实参数量是变化的，或者调用函数的上下文需要指定为在运行时确定的特定对象。显然，函数本身就是对象，因此这些对象也有方法（参考[Function](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Function) ）。作为此中情形之一，[apply()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Function/apply)方法可以实现这些目的。

## 函数作用域

【军规】在函数内定义的变量不能在函数之外的任何地方访问，因为变量仅仅在该函数的域的内部有定义。相对应的，一个函数可以访问定义在其范围内的任何变量和函数。换言之，定义在全局域中的函数可以访问所有定义在全局域中的变量。在另一个函数中定义的函数也可以访问在其父函数中定义的所有变量和父函数有权访问的任何其他变量。

// 下面的变量定义在全局作用域(global scope)中

var num1 = 20,

num2 = 3,

name = "Chamahk";

// 本函数定义在全局作用域

function multiply() {

return num1 \* num2;

}

multiply(); // 返回 60

// 嵌套函数的例子

function getScore() {

var num1 = 2,

num2 = 3;

function add() {

return name + " scored " + (num1 + num2);

}

return add();

}

getScore(); // 返回 "Chamahk scored 5"

## 作用域和函数堆栈

### 递归

一个函数可以指向并调用自身。有三种方法可以达到这个目的：

函数名

[arguments.callee](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments/callee)

作用域下的一个指向该函数的变量名

例如，思考一下如下的函数定义：

var foo = function bar() {

// statements go here

};

在这个函数体内，以下的语句是等价的：

bar()

arguments.callee()

foo()

调用自身的函数我们称之为*递归函数*。在某种意义上说，递归近似于循环。两者都重复执行相同的代码，并且两者都需要一个终止条件（避免无限循环或者无限递归）。例如以下的循环：

var x = 0;

while (x < 10) { // "x < 10" 是循环条件

// do stuff

x++;

}

可以被转化成一个递归函数和对其的调用：

function loop(x) {

if (x >= 10) // "x >= 10" 是退出条件（等同于 "!(x < 10)"）

return;

// 做些什么

loop(x + 1); // 递归调用

}

loop(0);

不过，有些算法并不能简单的用迭代来实现。例如，获取树结构中所有的节点时，使用递归实现要容易得多：

function walkTree(node) {

if (node == null) //

return;

// do something with node

for (var i = 0; i < node.childNodes.length; i++) {

walkTree(node.childNodes[i]);

}

}

跟loop函数相比，这里每个递归调用都产生了更多的递归。

将递归算法转换为非递归算法是可能的，不过逻辑上通常会更加复杂，而且需要使用堆栈。事实上，递归函数就使用了堆栈：函数堆栈。

这种类似堆栈的行为可以在下例中看到：

function foo(i) {

if (i < 0)

return;

console.log('begin:' + i);

foo(i - 1);

console.log('end:' + i);

}

foo(3);

### 嵌套函数和闭包

你可以在一个函数里面嵌套另外一个函数。嵌套（内部）函数对其容器（外部）函数是私有的。它自身也形成了一个闭包。一个闭包是一个可以自己拥有独立的环境与变量的的表达式（通常是函数）。

既然嵌套函数是一个闭包，就意味着一个嵌套函数可以”继承“容器函数的参数和变量。换句话说，内部函数包含外部函数的作用域。

可以总结如下：

内部函数只可以在外部函数中访问。

内部函数形成了一个闭包：它可以访问外部函数的参数和变量，但是外部函数却不能使用它的参数和变量。

下面的例子展示了嵌套函数：

function addSquares(a, b) {

function square(x) {

return x \* x;

}

return square(a) + square(b);

}

a = addSquares(2, 3); // returns 13

b = addSquares(3, 4); // returns 25

c = addSquares(4, 5); // returns 41

由于内部函数形成了闭包，因此你可以调用外部函数并为外部函数和内部函数指定参数：

function outside(x) {

function inside(y) {

return x + y;

}

return inside;

}

fn\_inside = outside(3); // Think of it like: give me a function that adds 3 to whatever you give it

result = fn\_inside(5); // returns 8

result1 = outside(3)(5); // returns 8

### 保存变量

注意到上例中 inside 被返回时 x 是怎么被保留下来的。一个闭包必须保存它可见作用域中所有参数和变量。因为每一次调用传入的参数都可能不同，每一次对外部函数的调用实际上重新创建了一遍这个闭包。只有当返回的 inside 没有再被引用时，内存才会被释放。

这与在其他对象中存储引用没什么不同，但是通常不太明显，因为并不能直接设置引用，也不能检查它们。

### 多层嵌套函数

函数可以被多层嵌套。例如，函数A可以包含函数B，函数B可以再包含函数C。B和C都形成了闭包，所以B可以访问A，C可以访问B和A。因此，闭包可以包含多个作用域；他们递归式的包含了所有包含它的函数作用域。这个称之为作用*域链*。（稍后会详细解释）

思考一下下面的例子：

function A(x) {

function B(y) {

function C(z) {

console.log(x + y + z);

}

C(3);

}

B(2);

}

A(1); // logs 6 (1 + 2 + 3)

在这个例子里面，C可以访问B的y和A的x。这是因为：

B形成了一个包含A的闭包，B可以访问A的参数和变量

C形成了一个包含B的闭包

B包含A，所以C也包含A，C可以访问B和A的参数和变量。换言之，C用这个顺序链接了B和A的作用域

反过来却不是这样。A不能访问C，因为A看不到B中的参数和变量，C是B中的一个变量，所以C是B私有的。

### 命名冲突

当同一个闭包作用域下两个参数或者变量同名时，就会产生命名冲突。更近的作用域有更高的优先权，所以最近的优先级最高，最远的优先级最低。这就是作用域链。链的第一个元素就是最里面的作用域，最后一个元素便是最外层的作用域。

看以下的例子：

function outside() {

var x = 5;

function inside(x) {

return x \* 2;

}

return inside;

}

outside()(10); // returns 20 instead of 10

命名冲突发生在return x上，inside的参数x和outside变量x发生了冲突。这里的作用链域是{inside, outside, 全局对象}。因此inside的x具有最高优先权，返回了20（inside的x）而不是10（outside的x）。

## arguments 对象

函数的实际参数会被保存在一个类似数组的arguments对象中。在函数内，你可以按如下方式找出传入的参数：

arguments[i]

i是参数的序数编号（译注：数组索引），以0开始。所以第一个传来的参数会是arguments[0]。参数的数量由arguments.length表示。

使用arguments对象，你可以处理比声明的更多的参数来调用函数。这在你事先不知道会需要将多少参数传递给函数时十分有用。你可以用arguments.length来获得实际传递给函数的参数的数量，然后用arguments对象来取得每个参数。

例如，设想有一个用来连接字符串的函数。唯一事先确定的参数是在连接后的字符串中用来分隔各个连接部分的字符（译注：比如例子里的分号“；”）。该函数定义如下：

function myConcat(separator) {

var result = ''; // 把值初始化成一个字符串，这样就可以用来保存字符串了！！

var i;

// iterate through arguments

for (i = 1; i < arguments.length; i++) {

result += arguments[i] + separator;

}

return result;

}

你可以给这个函数传递任意数量的参数，它会将各个参数连接成一个字符串“列表”：

// returns "red, orange, blue, "

myConcat(", ", "red", "orange", "blue");

// returns "elephant; giraffe; lion; cheetah; "

myConcat("; ", "elephant", "giraffe", "lion", "cheetah");

// returns "sage. basil. oregano. pepper. parsley. "

myConcat(". ", "sage", "basil", "oregano", "pepper", "parsley");

**提示：**arguments变量只是 *”***类数组对象**“，并不是一个数组，类数组对象有一个索引编号和length属性。并不拥有全部的Array对象的操作方法。

## 函数参数

从ECMAScript 6开始，有两个新的类型的参数：默认参数，剩余参数。

### 默认参数

在JavaScript中，函数参数的默认值是undefined。然而，在某些情况下设置不同的默认值是有用的。这时默认参数可以提供帮助。

在过去，用于设定默认的一般策略是在函数的主体测试参数值是否为undefined，如果是则赋予一个值。如果在下面的例子中，调用函数时没有实参传递给b，那么它的值就是undefined，于是计算a\*b得到、函数返回的是 NaN：

function multiply(a, b) {

b = (typeof b !== 'undefined') ? b : 1;

return a\*b;

}

multiply(5); // 5

使用默认参数，在函数体的检查就不再需要了。现在，你可以在函数头简单地把1设定为b的默认值：

function multiply(a, b = 1) {

return a\*b;

}

multiply(5); // 5

了解更多[默认参数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters)的信息。

### [Link to section](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Functions#%E5%AE%9A%E4%B9%89%E5%87%BD%E6%95%B0#%E5%89%A9%E4%BD%99%E5%8F%82%E6%95%B0)剩余参数

[剩余参数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/rest_parameters)语法允许将不确定数量的参数表示为数组。在下面的例子中，使用剩余参数收集从第二个到最后参数。然后，我们将这个数组的每一个数与第一个参数相乘。这个例子是使用了一个箭头函数，这将在下一节介绍。

function multiply(multiplier, ...theArgs) {

return theArgs.map(x => multiplier \* x);

}

var arr = multiply(2, 1, 2, 3);

console.log(arr); // [2, 4, 6]

## 箭头函数

[箭头函数表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions)（也称胖箭头函数）相比函数表达式具有较短的语法并以词法的方式绑定 this。箭头函数总是匿名的。另见 hacks.mozilla.org 的博文：“[深度了解ES6：箭头函数](https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/)”。

有两个因素会影响引入箭头函数：更简洁的函数和 this。

### 更简洁的函数

有一些函数模式，更简洁的函数很受欢迎。对比一下：

var a = [

"Hydrogen",

"Helium",

"Lithium",

"Beryllium"

];

var a2 = a.map(function(s){ return s.length });

console.log(a2); // logs [ 8, 6, 7, 9 ]

var a3 = a.map( s => s.length );

console.log(a3); // logs [ 8, 6, 7, 9 ]

### this 的词法

在箭头函数出现之前，每一个新函数都重新定义了自己的 [this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this) 值（在严格模式下，一个新的对象在构造函数里是未定义的，以“对象方法”的方式调用的函数是上下文对象等）。以面向对象的编程风格，这样着实有点恼人。

function Person() {

// The Person() constructor defines `this` as itself.

this.age = 0;

setInterval(function growUp() {

// In nonstrict mode, the growUp() function defines `this`

// as the global object, which is different from the `this`

// defined by the Person() constructor.

this.age++;

}, 1000);

}

var p = new Person();

在ECMAScript 3/5里，通过把this的值赋值给一个变量可以修复这个问题。

function Person() {

var self = this; // Some choose `that` instead of `self`.

// Choose one and be consistent.

self.age = 0;

setInterval(function growUp() {

// The callback refers to the `self` variable of which

// the value is the expected object.

self.age++;

}, 1000);

}

另外，创建一个[约束函数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Function/bind)可以使得 this值被正确传递给 growUp() 函数。

箭头功能捕捉闭包上下文的this值，所以下面的代码工作正常。

function Person(){

this.age = 0;

setInterval(() => {

this.age++; // |this| properly refers to the person object

}, 1000);

}

var p = new Person();

## 运算符

JavaScript 拥有如下类型的运算符。本节描述了运算符和运算符的优先级。JavaScript 拥有二元和一元运算符， 和一个特殊的三元运算符（条件运算符）。一个二元运算符需要两个操作数，分别在运算符的前面和后面：

操作数1 运算符 操作数2

例如, 3+4 或 x\*y。

一个一元运算符需要一个操作数，在运算符前面或后面：

运算符 操作数

或

操作数 运算符

例如, x++ 或 ++x。

### 赋值运算符

一个 [赋值运算符(assignment operator)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Assignment_Operators) 将它右边操作数的值赋给它左边的操作数。最简单的赋值运算符是等于（=），它将右边的操作数值赋给左边的操作数。那么 x = y 就是将 y 的值赋给 x。

还有一些复合赋值操作符，它们是下表列出的这些操作的缩写：

| 复合赋值运算符 | | |
| --- | --- | --- |
| 名字 | 简写的操作符 | 含义 |
| [赋值(Assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Assignment) | x = y | x = y |
| [加法赋值(Addition assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Addition_assignment) | x += y | x = x + y |
| [减法赋值(Subtraction assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Subtraction_assignment) | x -= y | x = x - y |
| [乘法赋值(Multiplication assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Multiplication_assignment) | x \*= y | x = x \* y |
| [除法赋值(Division assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Division_assignment) | x /= y | x = x / y |
| [求余赋值(Remainder assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Remainder_assignment) | x %= y | x = x % y |
| [求幂赋值(Exponentiation assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Exponentiation_assignment) | x \*\*= y | x = x \*\* y |
| [左移位赋值(Left shift assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Left_shift_assignment) | x <<= y | x = x << y |
| [右移位赋值(Right shift assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Right_shift_assignment) | x >>= y | x = x >> y |
| [无符号右移位赋值(Unsigned right shift assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Unsigned_right_shift_assignment) | x >>>= y | x = x >>> y |
| [按位与赋值(Bitwise AND assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Bitwise_AND_assignment) | x &= y | x = x & y |
| [按位异或赋值(Bitwise XOR assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Bitwise_XOR_assignment) | x ^= y | x = x ^ y |
| [按位或赋值(Bitwise OR assignment)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment_Operators#Bitwise_OR_assignment) | x |= y | x = x | y |

解构

对于更复杂的赋值，[解构赋值](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment)语法是一个能从数组或对象对应的数组结构或对象字面量里提取数据的 Javascript 表达式。

var foo = ["one", "two", "three"];

// 不使用解构

var one = foo[0];

var two = foo[1];

var three = foo[2];

// 使用解构

var [one, two, three] = foo;

### 比较运算符

[比较运算符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Comparison_Operators)比较它的操作数并返回一个基于表达式是否为真的逻辑值。操作数可以是数字，字符串，逻辑，对象值。字符串比较是基于标准的字典顺序，使用Unicode值。在多数情况下，如果两个操作数不是相同的类型， JavaScript 会尝试转换它们为恰当的类型来比较。这种行为通常发生在数字作为操作数的比较。类型转换的例外是使用 === 和 !== 操作符，它们会执行严格的相等和不相等比较。这些运算符不会在检查相等之前转换操作数的类型。下面的表格描述了该示例代码中的各比较运算符

var var1 = 3;

var var2 = 4;

| 比较运算符 | | |
| --- | --- | --- |
| 运算符 | 描述 | 返回true的示例 |
| [等于 Equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Equality) (==) | 如果两边操作数相等时返回true。 | 3 == var1  "3" == var1  3 == '3' |
| [不等于 Not equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Inequality) (!=) | 如果两边操作数不相等时返回true | var1 != 4 var2 != "3" |
| [全等 Strict equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Identity) (===) | 两边操作数相等且类型相同时返回true。 参见 [Object.is](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/is) and [sameness in JS](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness). | 3 === var1 |
| [不全等 Strict not equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Nonidentity) (!==) | 两边操作数不相等或类型不同时返回true。 | var1 !== "3" 3 !== '3' |
| [大于 Greater than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_operator) (>) | 左边的操作数大于右边的操作数返回true | var2 > var1 "12" > 2 |
| [大于等于 Greater than or equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Greater_than_or_equal_operator) (>=) | 左边的操作数大于或等于右边的操作数返回true | var2 >= var1 var1 >= 3 |
| [小于 Less than](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than_operator) (<) | 左边的操作数小于右边的操作数返回true | var1 < var2 "2" < 12 |
| [小于等于 Less than or equal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators#Less_than_or_equal_operator) (<=) | 左边的操作数小于或等于右边的操作数返回true | var1 <= var2 var2 <= 5 |

**注意:**（**=>**） 不是运算符，而是[箭头函数](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)的标记符号 。

### [Link to section](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Expressions_and_Operators#%E7%AE%97%E6%9C%AF%E8%BF%90%E7%AE%97%E7%AC%A6)算术运算符

算术运算符使用数值(字面量或者变量)作为操作数并返回一个数值.标准的算术运算符就是加减乘除(+ - \* /)。当操作数是浮点数时，这些运算符表现得跟它们在大多数编程语言中一样（特殊要注意的是，除零会产生[Infinity](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Infinity)）。例如：

1 / 2; // 0.5

1 / 2 == 1.0 / 2.0; // true

除了标准的算术运算符（+， - ，\* /），JavaScript还提供了下表中的算术运算符。

| 表 3.3 算术运算符 | | |
| --- | --- | --- |
| Operator | Description | Example |
| 求余(%) | 二元运算符. 返回相除之后的余数. | 12 % 5 返回 2。 |
| 自增(++) | 一元运算符. 将操作数的值加一. 如果放在操作数前面 (++x), 则返回加一后的值; 如果放在操作数后面 (x++), 则返回操作数原值,然后再将操作数加一. | var x=3;  console.log(++x); //4  console.log(x); //4  var y=3;  console.log(y++); //3  console.log(y); //4 |
| 自减(--) | 一元运算符. 将操作数的值减一. 前后缀两种用法的返回值类似自增运算符. | var x=3; console.log(--x); //输入2,x=2  var y=3;console.log(y--);//输出3,x=2; |
| 一元负值符(-) | 一元运算符,返回操作数的负值. | var x=3; console.log(-x); //输入-3 |
| 一元正值符(+) | 一元运算符, 如果操作数在之前不是number，试图将其转换为number | console.log( +'3' ); // 3  console.log( '3' ); // '3'  console.log(+true); // 1 |
| 指数运算符(\*\*) | 计算 base(底数) 的 exponent(指数)次方, 表示为baseexponent | 2 \*\* 3 returns 8. 10 \*\* -1 returns 0.1. |
|  |  |  |

### 位运算符

位运算符将它的操作数视为32位元的二进制串（0和1组成）而非十进制八进制或十六进制数。例如：十进制数字9用二进制表示为1001，位运算符就是在这个二进制表示上执行运算，但是返回结果是标准的JavaScript数值。

下表总结了 JavaScript 的位运算符。

| 位运算符 | | |
| --- | --- | --- |
| Operator | Usage | Description |
| 按位与[AND](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_AND) | a & b | 在a,b的位表示中，每一个对应的位都为1则返回1， 否则返回0. |
| 按位或[OR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_OR) | a | b | 在a,b的位表示中，每一个对应的位，只要有一个为1则返回1， 否则返回0. |
| 按位异或[XOR](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_XOR) | a ^ b | 在a,b的位表示中，每一个对应的位，两个不相同则返回1，相同则返回0. |
| 按位非[NOT](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Bitwise_NOT) | ~ a | 反转被操作数的位。 |
| 左移[shift](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators#Left_shift) | a << b | 将a的二进制串向左移动b位,右边移入0. |
| 算术右移 | a >> b | 把a的二进制表示向右移动b位，丢弃被移出的所有位.  (译注:算术右移左边空出的位是根据最高位是0和1来进行填充的) |
| 无符号右移  (左边空出位用0填充) | a >>> b | 把a的二进制表示向右移动b位，丢弃被移出的所有位，并把左边空出的位都填充为0 |

* **位逻辑运算符**

概念上来讲, 位逻辑运算符工作流程如下:

操作数被转换为32bit整數，以位序列（0和1组成）表示.若超過32bits，則取低位32bit，如下所示：

Before: 11100110111110100000000000000110000000000001

After: 10100000000000000110000000000001

第一个操作数的每一位都与第二个操作数的对应位组对: 第一位对应第一位,第二位对应第二位,以此类推.

运算符被应用到每一对"位"上, 最终的运算结果由每一对“位”的运算结果组合起来.

例如,十进制数9的二进制表示是1001,十进制数15的二进制表示是1111.因此,当位运算符应用到这两个值时,结果如下:

| 位运算符范例 | | |
| --- | --- | --- |
| 表达式 | 结果 | 二进制描述 |
| 15 & 9 | 9 | 1111 & 1001 = 1001 |
| 15 | 9 | 15 | 1111 | 1001 = 1111 |
| 15 ^ 9 | 6 | 1111 ^ 1001 = 0110 |
| ~15 | -16 | ~00000000...00001111 = 11111111...11110000 |
| ~9 | -10 | ~00000000...00001001 = 11111111...11110110 |

注意位运算符“非”将所有的32位取反，而值的最高位(最左边的一位)为1则表示负数(2-补码表示法)。

* **移位运算符**

移位运算符带两个操作数：第一个是待移位的数，第二个是指定第一个数要被移多少位的数。移位的方向由运算符来控制.

移位运算符把操作数转为32bit整数，然后得出一个与待移位数相同种类的值。

移位运算符列表如下。

**移位运算符**

| 运算符 | 描述 | 范例 |
| --- | --- | --- |
| << (左移位) | 将第一个操作数向左移动指定数量的位. 左边移出位被抛弃. 左边移出的几位被丢弃.右边多出的空位由0补齐 | 9<<2产生36，因为1001移位2比特向左变为100100，它是36。 |
| >> (带符号右移) | 将第一个操作数向右移动指定数量的位. 右边移出位被抛弃. 左边多出的空位由原值的最左边数字补齐. | 9>>2产生2，因为1001移位2位向右变为10，其是2。同样，-9>>2产生-3，由于符号被保留。 |
| >>> (补零右移) | 将第一个操作数向右移动指定数量的位. 右边移出位被抛弃. 左边多出的空位由0补齐. | 19>>>2产生4，因为10011移位2位向右变为100，它是4。对非负数值，补零右移和带符号右移产生相同结果。 |

### 逻辑运算符

逻辑运算符常用于布尔（逻辑）值之间; 当操作数都是布尔值时，返回值也是布尔值。 不过实际上&&和||返回的是一个特定的操作数的值，所以当它用于非布尔值的时候，返回值就可能是非布尔值。 逻辑运算符的描述如下。

**逻辑运算符**

| 运算符 | 范例 | 描述 |
| --- | --- | --- |
| [逻辑与](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_AND) (&&) | expr1 && expr2 | (逻辑与) 如果expr1能被转换为false，那么返回expr1；否则，返回expr2。因此，&&用于布尔值时，当操作数都为true时返回true；否则返回false. |
| [逻辑或](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_OR)(||) | expr1 || expr2 | (逻辑或) 如果expr1能被转换为true，那么返回expr1；否则，返回expr2。因此，||用于布尔值时，当任何一个操作数为true则返回true；如果操作数都是false则返回false。 |
| [逻辑非](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators#Logical_NOT) (!) | !expr | (逻辑非) 如果操作数能够转换为true则返回false；否则返回true。 |

能被转换为false的值有null, 0, NaN, 空字符串("")和undefined。（译者注：也可以称作”falsy“）

下面是&&（逻辑"与"）操作符的示例。

var a1 = true && true; // t && t returns true

var a2 = true && false; // t && f returns false

var a3 = false && true; // f && t returns false

var a4 = false && (3 == 4); // f && f returns false

var a5 = "Cat" && "Dog"; // t && t returns Dog

var a6 = false && "Cat"; // f && t returns false

var a7 = "Cat" && false; // t && f returns false

下面是||（逻辑"或"）操作符的示例。

var o1 = true || true; // t || t returns true

var o2 = false || true; // f || t returns true

var o3 = true || false; // t || f returns true

var o4 = false || (3 == 4); // f || f returns false

var o5 = "Cat" || "Dog"; // t || t returns Cat

var o6 = false || "Cat"; // f || t returns Cat

var o7 = "Cat" || false; // t || f returns Cat

下面是！（逻辑"非"）操作符的示例。

var n1 = !true; // !t returns false

var n2 = !false; // !f returns true

var n3 = !"Cat"; // !t returns false

作为逻辑表达式进行求值是从左到右，它们是为可能的“短路”的出现而使用以下规则进行测试：

false && *anything*    // 被短路求值为false

true || *anything*       // 被短路求值为true

逻辑的规则，保证这些评估是总是正确的。请注意，上述表达式的任何一部分不评估，所以这样做的任何副作用不生效。

### 字符串运算符

除了比较操作符，它可以在字符串值中使用，连接操作符（+）连接两个字符串值相连接，返回另一个字符串，它是两个操作数串的结合。

例如，

console.log("my " + "string"); // console logs the string "my string".

简写操作符 += 也可以用来拼接字符串，例如：

var myString = "alpha";

myString += "bet"; // 返回 "alphabet"

### 条件（三元）运算符

[条件运算符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)是JavaScript中唯一需要三个操作数的运算符。运算的结果根据给定条件在两个值中取其一。语法为：

条件 ? 值1 : 值2

如果条件为真，则结果取值1。否则为值2。你能够在任何允许使用标准运算符的地方使用条件运算符。

例如，

var status = (age >= 18) ? "adult" : "minor";

当 age 大于等于18的时候，将“adult”赋值给 status；否则将“minor”赋值给 status。

### 逗号操作符

[逗号操作符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Comma_Operator)（,）对两个操作数进行求值并返回最终操作数的值。它常常用在 for 循环中，在每次循环时对多个变量进行更新。

例如，假如 a 是一个二维数组，每个维度各有10个元素，以下代码利用逗号操作符来同时改变两个变量的值。这段代码的功能是打印出该二维数组的对角线元素的值：

var x = [0,1,2,3,4,5,6,7,8,9]

var a = [x, x, x, x, x];

for (var i = 0, j = 9; i <= j; i++, j--)

console.log('a[' + i + '][' + j + ']= ' + a[i][j]);

### 一元操作符

一元操作符仅对应一个操作数。

* delete

[delete](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete)操作符，删除一个对象或一个对象的属性或者一个数组中某一个键值。语法如下:

delete objectName;

delete objectName.property;

delete objectName[index];

delete property; // legal only within a with statement

objectName是一个对象名，property 是一个已经存在的属性，index是数组中的一个已经存在的键值的索引值。

第四行的形式只在[with](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with)声明的状态下是合法的， 从对象中删除一个属性。

你能使用 delete 删除各种各样的隐式声明， 但是被var声明的除外。

如果 delete 操作成功，属性或者元素会变成 undefined。如果 delete可行会返回true，如果不成功返回false。

x = 42;

var y = 43;

myobj = new Number();

myobj.h = 4; // create property h

delete x; // returns true (can delete if declared implicitly)

delete y; // returns false (cannot delete if declared with var)

delete Math.PI; // returns false (cannot delete predefined properties)

delete myobj.h; // returns true (can delete user-defined properties)

delete myobj; // returns true (can delete if declared implicitly)

删除数组元素

删除数组中的元素时，数组的长度是不变的，例如删除a[3], a[4]，a[4]和a[3] 仍然存在变成了undefined。

delete 删除数组中的一个元素，这个元素就不在数组中了。例如，trees[3]被删除，trees[3] 仍然可寻址并返回undefined。

var trees = new Array("redwood", "bay", "cedar", "oak", "maple");

delete trees[3];

if (3 in trees) {

// 不会被执行

}

如果想让数组中存在一个元素但是是undefined值，使用undefined关键字而不是delete操作. 如下： trees[3]分配一个undefined,但是这个数组元素仍然存在:

var trees = new Array("redwood", "bay", "cedar", "oak", "maple");

trees[3] = undefined;

if (3 in trees) {

// this gets executed（会被执行）

}

* typeof

[typeof操作符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/typeof) 可通过下面2种方式使用：

typeof operand

typeof (operand)

typeof 操作符返回一个表示 operand 类型的字符串值。operand 可为字符串、变量、关键词或对象，其类型将被返回。operand 两侧的括号为可选。

假设你定义了如下的变量：

var myFun = new Function("5 + 2");

var shape = "round";

var size = 1;

var today = new Date();

typeof 操作符将会返回如下的结果：

typeof myFun; // returns "function"

typeof shape; // returns "string"

typeof size; // returns "number"

typeof today; // returns "object"

typeof dontExist; // returns "undefined"

对于关键词 true 和 null， typeof 操作符将会返回如下结果：

typeof true; // returns "boolean"

typeof null; // returns "object"

对于一个数值或字符串， typeof 操作符将会返回如下结果：

typeof 62; // returns "number"

typeof 'Hello world'; // returns "string"

对于属性值，typeof 操作符将会返回属性所包含值的类型：

typeof document.lastModified; // returns "string"

typeof window.length; // returns "number"

typeof Math.LN2; // returns "number"

对于方法和函数，typeof 操作符将会返回如下结果：

typeof blur; // returns "function"

typeof eval; // returns "function"

typeof parseInt; // returns "function"

typeof shape.split; // returns "function"

对于预定义的对象，typeof 操作符将会返回如下结果：

typeof Date; // returns "function"

typeof Function; // returns "function"

typeof Math; // returns "object"

typeof Option; // returns "function"

typeof String; // returns "function"

* void

void 运算符运用方法如下：

void (expression)

void expression

void运算符,表明一个运算没有返回值。expression是javaScript表达式，括号中的表达式是一个可选项，当然使用该方式是一种好的形式。

你可以使用void运算符指明一个超文本链接。该表达式是有效的，但是并不会在当前文档中进行加载。

如下创建了一个超链接文本，当用户单击该文本时，不会有任何效果。

<a href="javascript:void(0)">Click here to do nothing</a>

下面的代码创建了一个超链接，当用户单击它时，提交一个表单。

<a href="javascript:void(document.form.submit())">

Click here to submit</a>

### 关系操作符

关系操作符对操作数进行比较，根据比较结果真或假，返回相应的布尔值。

[in操作符](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in)，如果所指定的**属性**确实存在于所指定的对象中，则会返回true，语法如下：

propNameOrNumber in objectName

在这里 propNameOrNumber可以是一个代表着属性名的字符串或者是一个代表着数组索引的数值表达式，而objectName则是一个对象名。

下面的例子是 in 操作的常见用法。

// Arrays

var trees = new Array("redwood", "bay", "cedar", "oak", "maple");

0 in trees; // returns true

3 in trees; // returns true

6 in trees; // returns false

"bay" in trees; // returns false (you must specify the index number,

// not the value at that index)

"length" in trees; // returns true (length is an Array property)

// Predefined objects

"PI" in Math; // returns true

var myString = new String("coral");

"length" in myString; // returns true

// Custom objects

var mycar = {make: "Honda", model: "Accord", year: 1998};

"make" in mycar; // returns true

"model" in mycar; // returns true

instanceof

如果所判别的对象确实是所指定的类型，则返回true。其语法如下：

objectName instanceof objectType

objectName 是需要做判别的对象的名称,而objectType是假定的对象的类型, 例如[Date](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Date)或 [Array](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Array).

当你需要确认一个对象在运行时的类型时，可使用instanceof. 例如，需要 catch 异常时，你可以针对抛出异常的类型，来做不同的异常处理。

例如, 下面的代码使用instanceof去判断 theDay是否是一个 Date 对象. 因为theDay是一个Date对象, 所以if中的代码会执行.

var theDay = new Date(1995, 12, 17);

if (theDay instanceof Date) {

// statements to execute

}

### 运算符优先级

运算符的优先级，用于确定一个表达式的计算顺序。在你不能确定优先级时，可以通过使用括号显式声明运算符的优先级。

下表列出了描述符的优先级，从最高到最低。

**运算符优先级**

| Operator type | Individual operators |
| --- | --- |
| member | . [] |
| call / create instance | () new |
| negation/increment | ! ~ - + ++ -- typeof void delete |
| multiply/divide | \* / % |
| addition/subtraction | + - |
| bitwise shift | << >> >>> |
| relational | < <= > >= in instanceof |
| equality | == != === !== |
| bitwise-and | & |
| bitwise-xor | ^ |
| bitwise-or | | |
| logical-and | && |
| logical-or | || |
| conditional | ?: |
| assignment | = += -= \*= /= %= <<= >>= >>>= &= ^= |= |
| comma | , |

## 部分主要预定义函数

**JavaScript语言有好些个顶级的内建函数：**

* **eval()**

**eval()**方法会对一串字符串形式的JavaScript代码字符求值。

* **uneval()**

**uneval()**方法创建的一个[Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object)的源代码的字符串表示。

* **isFinite()**

**isFinite()**函数判断传入的值是否是有限的数值。 如果需要的话，其参数首先被转换为一个数值。

* **isNaN()**

**isNaN()**函数判断一个值是否是[NaN](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/NaN)。注意：isNaN函数内部的[强制转换规则](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/isNaN#Description)十分有趣； 另一个可供选择的是ECMAScript 6 中定义[Number.isNaN()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Number/isNaN) , 或者使用 [typeof](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/typeof)来判断数值类型。

* **parseFloat()**

**parseFloat()** 函数解析字符串参数，并返回一个浮点数。

* **parseInt()**

**parseInt()** 函数解析字符串参数，并返回指定的基数（基础数学中的数制）的整数。

* **decodeURI()**

**decodeURI()** 函数对先前经过[encodeURI](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/encodeURI)函数或者其他类似方法编码过的字符串进行解码。

* **decodeURIComponent()**

**decodeURIComponent()**方法对先前经过[encodeURIComponent](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent)函数或者其他类似方法编码过的字符串进行解码。

* **encodeURI()**

**encodeURI()**方法通过用以一个，两个，三个或四个转义序列表示字符的UTF-8编码替换统一资源标识符（URI）的某些字符来进行编码（每个字符对应四个转义序列，这四个序列组了两个”替代“字符）。

* **encodeURIComponent()**

**encodeURIComponent()** 方法通过用以一个，两个，三个或四个转义序列表示字符的UTF-8编码替换统一资源标识符（URI）的每个字符来进行编码（每个字符对应四个转义序列，这四个序列组了两个”替代“字符）。

* **escape()**

已废弃的 **escape()** 方法计算生成一个新的字符串，其中的某些字符已被替换为十六进制转义序列。使用 [encodeURI](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/encodeURI)或者[encodeURIComponent](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/encodeURIComponent)替代本方法。

* **unescape()**

已废弃的 **unescape()** 方法计算生成一个新的字符串，其中的十六进制转义序列将被其表示的字符替换。上述的转义序列就像[escape](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/escape)里介绍的一样。因为 unescape 已经废弃，建议使用[decodeURI()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/decodeURI)或者[decodeURIComponent](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/decodeURIComponent) 替代本方法。

# 测试规约

1. 【军规】单元测试必须遵守AIR原则。 说明：单元测试在线上运行时，感觉像空气（AIR）一样并不存在，但在测试质量的保障上，却是非常关键的。好的单元测试宏观上来说，具有自动化、独立性、可重复执行的特点。 

A：Automatic AutomaticAutomatic Automatic Automatic （自动化）

I：Independent IndependentIndependent Independent Independent Independent （独立性）

 R：Repeatable RepeatableRepeatable Repeatable Repeatable Repeatable（可重复）

2. 【军规】单元测试应该是全自动执行的，并且非交互式的。测试框架通常是定期执行的，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。

3. 【军规】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。 反例：method2需要依赖method1的执行，将执行结果做为method2的输入。

4. 【军规】单元测试是可以重复执行的，不能受到外界环境的影响。 说明：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。

5. 【军规】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。 说明：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. 【军规】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。 说明：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. 【军规】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。 说明：源码构建时会跳过此目录，而单元测试框架默认是扫描此目录。

8. 【建议】单元测试的基本目标：语句覆盖率达到70%；核心模块的语句覆盖率和分支覆盖率都要达到100% 说明：在工程规约的应用分层中提到的DAO层，Control层，可重用度高的Service，都应该进行单元测试。

9. 【建议】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。  B：Border，边界值测试，包括循环、特殊取时间点数据顺序等。 ，边界值测试包括循环、特殊取时间点数据顺序等。  C：Correct，正确的输入，并得到预期结果。 ，正确的输入并得到预期结果。  D：Design，与设计文档相结合，来编写单元测试。 ，与设计文档相结合来编写单元测试。  E：Error，强制错误信息输入（如：非法数据、异常流程业务允许等），并得到预期的结果。

10. 【建议】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RDC内部单元测试中，使用RDC\_UNIT\_TEST\_的前缀标识数据。

11. 【建议】对于不可测的代码建议做必要的重构，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. 【建议】在设计评审阶段，开发人员需要和测试人员一起评审单元测试范围，单元测试最好覆盖所有测试用例（UC）。

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

14. 【参考】为了更方便地进行单元测试，业务代码不应存在过多的全局 变量和静态方法、不应存在过多的外部依赖，不应存在过多的条件语句。

# 异常规约

## 异常处理

1. 【军规】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。

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) null) {...}

反例：try { obj.method() } catch (NullPointerException e) {...}

2. 【军规】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. 【军规】禁止对大段代码进行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。

4. 【军规】尽可能的处理捕获异常，如果不想处理，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. 【军规】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. 【军规】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. 【军规】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. 【军规】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. 【建议】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. 【建议】注意防止NPE的场景

1）返回类型为基本数据，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。

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） 远程调用返回对象时，一律要求进行空指针判断，防止NPE。

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。

正例：使用JDK8的Optional类来防止NPE问题。

11. 【建议】定义时区分unchecked / checked 异常，避免直接抛出new RuntimeException()，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】在代码中使用“抛异常”还是“返回错误码”，对于外部的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()方法、“错误码”、“错误简短信息”。

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，甚至是组件化。

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) {...}

## 日志规约

1. 【军规】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。

2. 【军规】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。

3. 【军规】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 + " and 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。

4. 【军规】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">

5. 【军规】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);

6. 【建议】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。

7. 【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。

# 安全规约

1. 【军规】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，比如查看他人的资料、修改他人的订单。

2. 【军规】用户敏感数据禁止直接展示，必须对展示数据进行脱敏。 说明：查看个人手机号码会显示成:131\*\*\*\*1234，隐藏中间4位，防止隐私泄露。

3. 【军规】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. 【军规】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：pagesize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。

5. 【军规】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. 【军规】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. 【军规】在使用基础通用能力，譬如短信、邮件、电话，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，并造成短信资源浪费。

# 规约执行

# 工具推荐

代码规约是组织内成员共同遵守的约定，同时也是管理者对代码质量、代码的可维护性、可读性的监管手段，规约的遵守和监管需要借助相关工具降低管理难度，提升检查效率。推荐面向编译文件、源文件语法、源文件编码格式的三类静态代码检查工具。

(增加sonar)

## 编译文件检测工具

【推荐工具】FindBugs

【工具介绍】FindBugs是一个开源的静态代码分析工具，基于LGPL开源协议，无需运行就能对代码进行分析的工具。不注重style及format，注重检测真正的bug及潜在的性能问题，尽可能抑制误检测(false positives)的发生。以bytecode（\*.class、\*.jar）为对象进行检查。除了单独运行，还可以用作Eclipse 的plug-in，以及嵌入Ant作为task之一进行利用。

【工具检查项】主要检查bytecode中的bug patterns，如NullPoint空指针检查、没有合理关闭资源、字符串相同判断错（==，而不是equals）等 。

【工具特点】

1)FindBugs主要着眼于寻找代码中的缺陷，直接操作类文件（class文件）而不是源代码。

2)FindBugs可以通过命令行、各种构建工具（如Ant、Maven等）、独立的Swing GUI或是以Eclipse和NetBeans IDE插件的方式来运行。

3)FindBugs输出结果既可以是XML的，也可以是文本形式的。

4)开发者可以通过多种方式来使用FindBugs，最常见的是在新编写模块的代码分析以及对现有代码进行更大范围的分析。

5)不注重style及format，注重检测真正的bug及潜在的性能问题，尤其注意了尽可能抑制误检测(false positives)的发生。

## 源文件语法类检测工具

【推荐工具】PMD

【工具介绍】PMD是一个代码检查工具，它用于分析Java源代码，找出潜在的问题：

1)潜在的bug：空的try/catch/finally/switch语句

2)未使用的代码：未使用的局部变量、参数、私有方法等

3)可选的代码：String/StringBuffer的滥用

4)复杂的表达式：不必须的if语句、可以使用while循环完成的for循环

5)重复的代码：拷贝/粘贴代码意味着拷贝/粘贴bugs。

【工具检查项】主要包括：

空try/catch/finally/switch语句块

未使用的局部变量、参数和private方法

空if/while语句

过于复杂的表达式，如不必要的if语句等

复杂类。

【工具特点】

1) PMD通过静态分析获知代码错误。也就是说，在不运行Java程序的情况下报告错误。

2)PMD附带了许多可以直接使用的规则，利用这些规则可以找出Java源程序的许多问题。

3)PMD已经与JDeveloper、Eclipse、jEdit、JBuilder、BlueJ、CodeGuide、NetBeans、Sun JavaStudio Enterprise/Creator、IntelliJ IDEA、TextPad、Maven、Ant、Gel、JCreator以及Emacs集成在一起。

4)PMD规则是可以定制的: 可用的规则并不仅限于内置规则。您可以添加新规则：可以通过编写Java代码并重新编译PDM，或者更简单些，编写XPath表达式，它会针对每个Java类的抽象语法树进行处理。

## 源文件格式类检测工具

【推荐工具】Checkstyle

【工具介绍】Checkstyle是一款检查Java程序源代码样式的工具。

【工具检查项】主要包括：

Javadoc注释

命名规范

多余没用的Imports

Size度量，如过长的方法

缺少必要的空格Whitespace

重复代码。

【工具特点】

1)它可以有效的帮助我们检视代码以便更好的遵循代码编写标准，特别适用于小组开发时彼此间的样式规范和统一。

2)Checkstyle提供了高可配置性，以便适用于各种代码规范，所以除了使用它提供的几种常见标准之外，你也可以定制自己的标准。

3)Checkstyle提供了支持大多数常见IDE的插件，大部分插件中就含有最新的Checkstyle，就不用费心再部署一份了。

4)Checkstyle可以检查代码的很多方面，从传统观点看，它主要是用来检查代码层面的，自从第三版以后，它的内部架构作了重大改变，很多其它意图的检测加了进来，现在Checkstyle可以检查像类设计的问题，重复代码，如锁的双重检查的bug模式。