ICS 点击此处添加 ICS 号
CCS 点击此处添加 CCS 号

T/

江 苏 省 软 件 行 业 协 会 团 体 标 准

T/JSIA 2026—XXXX

软件开发 Java 编码规范

（征求意见稿）

在提交反馈意见时，请将您知道的相关专利连同支持性文件一并附上。

XXXX－XX－XX 发布　　XXXX－XX－XX 实施

江苏省软件行业协会　　发 布

前　言

本文件按照GB/T 1.1—2020《标准化工作导则　第1部分：标准化文件的结构和起草规则》的规定起草。

本文件由××××提出。
本文件由××××归口。
本文件起草单位：
本文件主要起草人：

软件开发 Java 编码规范

1 范围

本标准规定了在Java开发过程中，针对面向对象编程（OOP）、服务化架构（SOA）、对象关系映射（ORM）等技术场景的编码质量检测要求。

本标准适用于以下对象：

a)企业级单体应用、微服务架构及云原生应用的开发；

b)使用Spring、MyBatis、Dubbo等主流框架的软件项目；

c)涉及一方库、二方库、三方库依赖管理的软件工程。

本标准适用于软件生命周期的以下阶段：

——软件设计；

——编码实现；

——测试验证；

——运行维护；

——代码审查。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

ISO/IEC25010:2011《系统和软件质量模型》

GB/T25000.51-2016《系统与软件工程系统与软件质量要求和评价（SQuaRE）第10部分：系统与软件质量模型》

GB/T8567-2006《计算机软件文档编制规范》

JR/T0167-2018《金融行业应用系统非功能需求规范》

YD/T3169-2016《电信和互联网服务软件代码安全要求》

T/CAS1-2017《软件工程开发规范》

T/CSIA003-2019《企业Java应用开发安全编码指南》

3 术语和定义

下列术语和定义适用于本文件。

3.1 POJO（PlainOrdinaryJavaObject）

特指仅包含setter、getter、toString方法的简单类，包括DO、DTO、BO、VO等类型，禁止使用“xxxPOJO”的形式命名。

3.2 GAV（GroupId、ArtifactctId、Version）

Maven坐标，包含GroupId、ArtifactId和Version三个元素，用于唯一标识依赖组件。

3.3 OOP（ObjectOrientedProgramming）

面向对象编程，一种基于“类”和“对象”的程序设计范式。

3.4 ORM（ObjectRelationMapping）

对象关系映射，指将对象模型与关系数据库表结构进行映射的框架，如MyBatis

3.5 NPE（java.lang.NullPointerException）

空指针异常，在调用空对象的属性或方法时抛出，应通过判空或Optional机制避免。

3.6 SOA（Service-OrientedArchitecture）

面向服务架构，一种将系统功能以松耦合服务形式设计、部署和复用的分布式架构风格。

4 总则

4.1 检测目的

Java程序编码质量检测的核心目的是通过对软件代码的规范性、稳定性、可维护性、结构清晰性及资源利用效率等维度进行系统评估，确保编码质量满足相应标准。通过该检测，能够有效识别潜在代码缺陷，并为软件质量的量化评估与持续改进提供客观依据。

4.2 检测类别

依据GB/T25000.10-2016（无对应引用标准），本标准明确以下检测类别，并对其作详细规定：

——编码规范性
——资源利用率
——代码稳定性
——安全性
——可维护性
——兼容性

在实际检测过程中，可根据软件的规模、类型及完整性级别，对上述检测类别进行选择性执行。上述维度应贯穿于软件生命周期的全过程，包括集成、验收、部署、运维及迁移等阶段，并可根据具体场景开展多类别联合检测。

4.3 检测需求

在检测启动前，应对检测对象及其运行环境进行全面分析，以形成清晰的检测需求，具体包括：

a） 了解业务场景，明确检测目标与重点方向；
b） 分析检测对象的代码规模与技术路线，初步确定适用的检测类别；
c） 结合其软件架构类型，识别潜在问题类别；
d） 收集生产环境硬件配置及依赖组件信息，为检测提供环境上下文；
e） 获取系统负载特征与趋势数据，辅助确定检测重点；
f） 综合上述维度的分析结果，编制详细的检测需求文档，涵盖检测方向、软件规模、架构特征、运行环境及负载情况等内容。

4.4 检测过程

4.4.1 概述

检测过程一般包含检测策划、检测准备、检测执行、检测汇总四项活动，各项活动应按顺序开展。

4.4.2 检测策划

检测策划的主要内容包括：

a） 检测场景分析：根据检测场景、规模、所处生命周期阶段、生产环境网络条件及检测目标，明确检测类别与内容；
b） 确定检测充分性要求；
c） 确定检测基本方式；
d） 确定检测工具与技术路线；
e） 开展风险识别、分析与评估；
f） 明确检测执行人员、监督人员与复核人员的能力与资格要求。

4.4.3 检测准备

检测准备应以检测需求与检测策划结果为输入，并包括人员配置与职责分工与检测资源配置与参数设置工作。

4.4.3.1 人员配置

人员配置如表1所示

表 1 人员配置表

人员类型	人数	具体职责
检测项目负责人	1	整个检测项目的主要责任人，负责资源调度，人员协调、以及检测结果的完备性、准确性和有效性。
检测需求计划复核员	1	复核提报的检测需求计划，分析需求计划是否合理，选择检测方向是否准确、检测范围是否有效覆盖。
检测配置员	1	合理编排检测内容，根据不同检测模块的要求以及检测对象的场景因素来配置合适的参数；精准有效地使用工具来满足检测需求
检测执行员	1	根据检测任务的编排内容与参数，执行对应的检测工具和模块，程序进行检测，同时采集检测结果并规范化。
结果审核员	1	对检测结果进行检测，避免参数配置或者潜在因素产生检测结果的偏差。
报告整合员	1	根据检测类型和场景生成相应的检测报告，最终交付给客户

注：其中检测项目负责人和执行人员是必须人员，其他人员可以由其他岗位人员兼职。

4.4.3.2 资源配置

资源配置应根据已明确的检测需求进行，主要涵盖两方面：

a) 硬件资源配置：明确检测所需设备的型号、并发处理能力上限，以及可支持的软件规模。
b) 软件资源配置：确定选用的检测模块与工具，明确多模块协同工作对硬件的具体要求，并规定结果数据的采集方式。

4.4.3.3 参数配置

参数配置应依据检测场景与具体需求进行，主要包括以下活动：

a)根据待测代码规模，配置并发执行的检测任务数量。
b)根据各检测模块的技术特性，配置其输入参数。
c)根据检测目标，编排各检测任务的执行顺序与参数传递关系。
d)为检测模块设定评估数据所需的基准或阈值。
e)配置模拟或匹配预期生产环境的运行参数。

4.4.4 检测实施

检测实施应按照以下程序有序开展：

a) **任务复核**：由检测计划复核人员对检测需求和待检代码进行复核，确认二者内容准确且相互一致；
b) **代码准备**：获取待测代码压缩包，进行一致性验证。验证通过后，通过文件指纹生成工具生成唯一特征码，作为本次检测的唯一标识（检测 ID）；
c) **环境准备**：根据检测策划方案，申请并初始化所需硬件、工具、网络等资源；
d) **代码上传**：通过指定的安全传输通道，将待测代码上传至检测设备；
e) **代码部署**：在检测设备指定目录中解压待测代码；
f) **模块初始化**：根据编排计划，使用预设模块参数对检测模块进行初始化；
g) **模块执行**：按计划执行各检测模块；
h) **结果采集**：通过结果采集工具获取各模块的检测结果；
i) **结果标注**：为检测结果标注检测环境、检测目的、检测维度等信息，形成待审核的结果数据；
j) **结果审核**：审核人员对检测结果进行审核，审核内容包括：
    1. 检测模块是否符合检测需求；
    2. 检测过程是否存在漏检；
    3. 检测结果是否准确；
    4. 结果格式是否规范；
    5. 正反例使用是否恰当；
    6. 适用规则是否完整；
    7. 问题原因是否明确；
    8. 问题代码定位是否清晰；

k） **问题整理**：整理检测结果中的问题，形成问题列表。若问题列表为空，则直接进入报告生成步骤；
l） **问题复测**：实施人员根据问题列表重新检测并调整相关代码；
m） **复核确认**：对复测结果进行复核；
n） **报告生成**：复核无问题后，报告整合员根据标准模板生成检测报告；
o） **报告签批**：依次由审核人员、项目负责人签字确认；
p） **报告用印**：按流程完成报告用印；
q） **报告交付**：通知项目单位领取检测报告；
r） **资料归档**：将代码压缩包与检测报告进行归档，并与检测 ID 关联，保存期满 30 天后自动删除。

4.4.5 检测报告

检测报告应包括概述、检测场景与目的、检测内容引用、具体检测项列表、附件与案例引用等组成部分。

4.4.5.1 概述

概述部分应至少包含以下信息：
——**唯一标识码**：为本次检测分配的唯一 ID。该标识码与检测对象的代码文件特征码进行强关联具有唯一性、可追溯性与防篡改性，用于将检测报告与被检测文件进行绑定。
——**申请单位**：提出检测需求的单位，负责明确检测目标和质量要求。
——**项目单位**：提供被检测代码的单位。
——**联系方式**：申请单位与项目单位指定的负责人及联系方式，包括但不限于电话、微信、QQ、电子邮箱等。

4.4.5.2 检测场景与目的

报告应清晰说明被检测代码的运行场景，包括但不限于请求规模、硬件配置、网络环境等信息，并阐述本次检测的目的，例如：
——验收阶段对代码规范性、可维护性进行评估；
——验证在大数据、高负载场景下的系统稳定性。

4.4.5.3 检测内容引用

检测内容应包含以下方面：
——**基础信息**：代码开发语言类型、所用框架等；
——**代码位置**：被检测代码在项目中的具体位置，包括文件相对路径及其在文件中的起止位置；
——**代码片段**：包含具体检测点的源代码片段；
——**检测依据**：判定检测结果所依据的标准或规则，可来源于公开标准、行业规范、企业规范或组织内部规范；
——**规则说明**：应对检测依据与判定规则进行具体说明，包含检测内容、判定方法及推荐的整改或优化建议。

4.4.5.4 具体检测项目

检测结果项应至少包含以下五个部分：代码片段位置、被检测代码内容、规则内容或依据、检测结果、完善建议。在实际检测中，可根据具体检测场景对结果项进行适应性调整。

4.4.5.4.1 代码片段位置

代码片段位置信息应包括：
——文件路径：被检测代码文件在项目中的相对路径；
——代码行号：被检测代码在文件中的起始行与结束行号；
——地址链接（可选）：指向被检测代码的直接访问链接或下载地址。

4.4.5.4.2 被检测代码内容

被检测代码内容可采用下列任一形式呈现：
——直接显示代码片段（适用于代码量较小的情况）；
——文件级高亮显示，突出展示被检测代码段；
——提供在线查看或下载链接。

4.4.5.4.3 规则内容或依据

规则内容应包含：
——具体描述：明确代码应遵循的要求、禁止的特征、违反的原则及可能引发的后果；
——可复现性：规则描述应与代码实现高度相关，支持通过人工走查等方式复现验证。

4.4.5.4.4 检测结果

检测结果应包括：
——检测结果包含被检测内容片段，代码中存在的问题项，具体问题项详细描述了代码存在的问题。
——文本描述：通过文本描述表示代码在指定维度上检测的结果，被检测单位可以通过文本信息了解该维度的代码质量。
——量化指标：列举指定场景下，该维度的指标并量化，如 CPU 占用率，单位时间的吞吐率。
——问题示例：列举改代码在指定维度上可能出现的现象，可以给出相关实现。
——趋势分析：对指定维度上的趋势分析和展示，为后续预防提供依据。

4.4.5.4.5 完善建议

根据问题严重程度，完善建议分为以下四类：
——强制修改：问题发生概率高或影响严重，必须在发布前修复；
——建议修改：问题影响较大，建议修复；如因特殊场景或客观限制无法修改，需说明原因；
——推荐优化：代码功能满足要求，但存在非最优实现，可能引发累积性风险（如内存占用过高），建议结合场景评估是否优化；
——参考建议：代码满足要求，风险较低，提供更佳实践作为参考。

4.4.5.4.6 附件与案例引用

——附件：检测过程中产生的原始代码、文本说明、引用文件等内容应作为附件保存，并分配唯一编号，在正文中通过编号引用；
——案例引用：针对具体检测项，可提供正例与反例。案例内容如无法在报告中完整呈现，应以文件形式保存并编号，与报告中相应引用点关联。

5 结果评价

检测结果的评判可采用以下三种方式之一或组合：
——分级制
根据预设的等级标准（如高、较高、中、低）对检测结果进行分级，不同等级对应不同的质量接收条件。
——通过制
由检测单位设定或与需求单位共同约定明确的通过条件。检测结果符合条件则评判为“通过”，否则为“不通过”。
——量化制
针对无明显质量界限的场景，可对相关质量维度进行相对量化。量化结果可作为决策参考或用于描述质量概况。

5.1 分级制

检测报告应包含综合评价结论，对被测软件的代码规范与实施质量进行等级评定。综合评价方法如表2。综合评价等级为“高”或“较高”时，视为满足系统代码总体软件质量要求，检测报告应给出各检测项的缺陷结论，并最终输出信息系统的代码质量综合评价等级，强制类缺陷为必须修改项。

表 2 分级制评价表

安全性结论	评价要求
高	在静态代码质量检测或整改完成后，若系统中无强制类缺陷，且建议类缺陷数≤10（不含强制类缺陷），则软件质量综合评价等级为**“高”**
较高	在静态代码质量检测或整改完成后，若系统中无强制类缺陷，且建议类缺陷数>10但≤20，则软件质量综合评价等级为**“较高”**
中	在静态代码质量检测或整改完成后，若系统满足无强制类缺陷，且建议类缺陷数>20但≤25或强制类缺陷数≤3，且建议类缺陷数≤25，则软件质量综合评价等级为“中”
低	在静态代码检测或整改完成后，若系统满足强制类缺陷数>3或建议类缺陷数>25，则软件质量综合评价等级为**“低”**

5.2 通过制

“通过制”是指由检测方提供包含具体检测项、覆盖维度及结果说明的检测方案，由委托方（或需求方）根据自身质量目标，从中选定所需的检测项，并为每个检测项的检测结果属性设定明确的通过与不通过条件，最终依据这些预定条件对整体检测结果进行二元化（通过/不通过）判定的方法。

5.3 量化制

“量化制”适用于将检测结果转换为具体量化数据的场景。通过对各项质量指标（如缺陷密度、复杂度、测试覆盖率等）进行量化，可以对代码实现的质量影响进行直观、客观的评估。量化结果便于进行趋势分析、资源投入决策、制定预防与改进方案、人员绩效管理以及优化内部编码规则的制定。

6 代码质量检测工具

代码质量检测工具是一类通过静态或者动态分析代码，自动发现程序中的缺陷、漏洞、规范问题与性能风险，从而提升软件质量、安全性和可维护性的工具。代码质量检测工具包含内容详见表3。

表 3 代码质量检测工具

工具类型	功能与特征说明	举例	备注
静态检测工具	对软件编码质量、结构设计、资源消耗、稳定性、可维护性等维度进行检测的工具	复杂度分析、规范性分析、可维护性分析、可靠性分析、稳定性分析等工具	针对软件编码质量、结构设计、可靠性、稳定性、资源利用率、可维护性等维度检测的工具很少
支持检测过程活动的其他工具	支持静态解析、语法语义解析、结构访问的工具	检测策划、资源配置、人员安排、规则加载、检测数据一致性等工具	抽象语法树结构访问工具可以帮助引擎执行具体检测规则。

附 录 A
（资料性）
检测服务内容

A.1 设计规约

【强制】存储方案和底层数据结构的设计获得评审一致通过，并沉淀成为文档。

说明：有缺陷的底层数据结构容易导致系统风险上升，可扩展性下降，重构成本也会因历史数据迁移和系统平滑过渡而陡然增加，所以，存储方案和数据结构需要认真地进行设计和评审，生产环境提交执行后，需要进行二次确认。

【强制】在需求分析阶段，如果与系统交互的User超过一类并且相关的UserCase超过5个，使用用例图来表达更加清晰的结构化需求。

说明：在需求分析阶段，若与系统交互的用户类别多于一种，且所有用户关联的用例总数超过5个，此时仅靠文字或列表难以清晰体现系统边界和角色权限，必须采用用例图来结构化地表达需求，以便直观展示不同参与者可执行的功能，有效避免遗漏、重复和角色分配错误。

【强制】如果某个业务对象的状态超过3个，使用状态图来表达并且明确状态变化的各个触发条件。

说明：状态图的核心是对象状态，首先明确对象有多少种状态，然后明确两两状态之间是否存在直接转换关系，再明确触发状态转换的条件是什么。

【强制】如果系统中某个功能的调用链路上的涉及对象超过3个，使用时序图来表达并且明确各调用环节的输入与输出。

说明：时序图反映了一系列对象间的交互与协作关系，清晰立体地反映系统的调用纵深链路。

【强制】如果系统中模型类超过5个，并且存在复杂的依赖关系，使用类图来表达并且明确类之间的关系。

说明：类图像建筑领域的施工图，如果搭平房，可能不需要，但如果建造空间大楼，肯定需要详细的施工图。

【强制】如果系统中超过2个对象之间存在协作关系，并且需要表示复杂的处理流程，使用活动图来表示。

说明：活动图是流程图的扩展，增加了能够体现协作关系的对象泳道，支持表示并发等。

【推荐】系统架构设计时明确以下目标：

确定系统边界。确定系统在技术层面上的做与不做。

确定系统内模块之间的关系。确定模块之间的依赖关系及模块的宏观输入与输出。

确定指导后续设计与演化的原则。使后续的子系统或模块设计在一个既定的框架内和技术方向上继续演化。

确定非功能性需求。非功能性需求是指安全性、可用性、可扩展性等。

【推荐】需求分析与系统设计在考虑主干功能的同时，需要充分评估异常流程与业务边界。

说明：在需求分析与系统设计中，任何一个主干功能都不能只停留在理想的正向流程上，而必须同步深入评估所有可能的异常场景和明确的业务边界，将超时、错误、并发、数据限制等非正常情况下系统的行为固化进文档，这样才能避免后期因边界不清导致的返工和漏洞，真正构建出稳健、可靠的产品逻辑。

【推荐】类在设计与实现时要符合单一原则。

说明：单一原则最易理解却是最难实现的一条规则，随着系统演进，很多时候，忘记了类设计的初衷。

【推荐】谨慎使用继承的方式来进行扩展，优先使用聚合/组合的方式来实现。

说明：不得已使用继承的话，必须符合里氏代换原则，此原则说父类能够出现的地方子类一定能够出现，比如，“把钱交出来”，钱的子类美元、欧元、人民币等都可以出现。

【推荐】系统设计阶段，根据依赖倒置原则，尽量依赖抽象类与接口，有利于扩展与维护。

说明：低层次模块依赖于高层次模块的抽象，方便系统间的解耦。

【推荐】系统设计阶段，注意对扩展开放，对修改闭合。

说明：极端情况下，交付的代码是不可修改的，同一业务域内的需求变化，通过模块或类的扩展来实现。

【推荐】系统设计阶段，共性业务或公共行为抽取出来公共模块、公共配置、公共类、公共方法等，在系统中不出现重复代码的情况。

说明：随着代码的重复次数不断增加，维护成本指数级上升。

【推荐】避免如下误解：敏捷开发=讲故事+编码+发布。

说明：敏捷开发是快速交付迭代可用的系统，省略多余的设计方案，摒弃传统的审批流程，但核心关键点上的必要设计和文档沉淀是需要的。

【参考】设计文档的作用是明确需求、理顺逻辑、后期维护，次要目的用于指导编码。

说明：避免为了设计而设计，系统设计文档有助于后期的系统维护和重构，所以设计结果需要进行分类归档保存。

【参考】可扩展性的本质是找到系统的变化点，并隔离变化点。

说明：世间众多设计模式其实就是一种设计模式即隔离变化点的模式。

【参考】设计的本质就是识别和表达系统难点。

说明：识别和表达完全是两回事，很多人错误地认为识别到系统难点在哪里，表达只是自然而然的事情，但是大家在设计评审中经常出现语焉不详，甚至是词不达意的情况。准确地表达系统难点需要具备如下能力：表达规则和表达工具的熟练性。抽象思维和总结能力的局限性。基础知识体系的完备性。深入浅出的生动表达力。

【参考】代码即文档的观点是错误的，清晰的代码只是文档的某个片断，而不是全部。

说明：代码的深度调用，模块层面上的依赖关系网，业务场景逻辑，非功能性需求等问题是需要相应的文档来完整地呈现的。

【参考】在做无障碍产品设计时，需要考虑到以下元素：

所有可交互的控件元素必须能被tab键聚焦，并且焦点顺序需符合自然操作逻辑。

用于登陆校验和请求拦截的验证码均需提供图形验证以外的其它方式。

自定义的控件类型需明确交互方式。

A.2 编程规约

A.2.1 命名规范

【强制】命名禁止使用特殊符号，禁止以下划线（_）或美元符号（$）开头或结尾。

【强制】禁止混合语言命名，严禁使用拼音与英文混合命名（如daZhePromotion），禁止纯中文命名。国际通用术语除外。

【强制】杜绝不规范缩写，禁止使用非行业共识的缩写（如condition缩写为cond）。

允许缩写：

国际通用缩写：MAX（最大值） /MIN（最小值）

领域术语缩写：HTTP（超文本传输协议）

【强制】类名格式

采用UpperCamelCase风格，如UserService。

例外：领域模型后缀需全大写（UserDO/OrderDTO）。

【强制】抽象类与异常类

抽象类命名以Abstract或Base开头（如AbstractController）。

异常类命名以Exception结尾（如ValidationException）。

【强制】接口与实现类

服务接口与实现类：

接口：UserService

实现类：UserServiceImpl

能力型接口以-able结尾（如Runnable/Cloneable）。

【强制】测试类命名

测试类需以Test结尾（如UserServiceTest）。

【强制】方法与变量格式，方法名、参数名、变量名统一使用lowerCamelCase风格。
正例：getUserInfo()/localValue
【强制】布尔变量命名，POJO类中布尔类型属性需直接表达状态，禁止添加is前缀。
【强制】数组声明格式，类型声明后接中括号
【强制】常量命名，全大写，单词间以下划线分隔，需完整表达语义。
【推荐】枚举规范
枚举类名以Enum结尾（如WeekdayEnum）。
枚举成员全大写，下划线分隔（如MONDAY/TUESDAY）。

【强制】Service/DAO层方法前缀

操作类型	方法前缀	示例
获取单个对象	get	getUserById()
获取多个对象	list	listActiveOrders()
统计操作	count	countInvalidLogs()

【强制】领域模型后缀
【推荐】命名体现设计模式，在类名中标识模式类型。
【推荐】接口规范，接口方法不加public修饰符，Javadoc 需说明方法契约。
【强制】包名格式全小写，单数形式，点分隔符间为单个语义单词。
常量规约
【强制】禁止魔法值，所有字面常量必须通过常量定义显式声明，禁止直接硬编码。
引用标准：GB/T 25000.51-2016 第5.2.3条（可维护性要求）
【强制】Long类型字面量格式，初始化long或Long类型时，必须使用大写L后缀。
【推荐】常量分类维护，禁止集中式常量类（如GlobalConstants），按功能模块拆分。
示例结构：

```
src
├── constants
│   ├── OrderConstants.java      // 订单相关常量
│   ├── PaymentConstants.java    // 支付相关常量
│   └── SystemEnv.java           // 系统环境变量
```

引用标准：T/CAS 1-2017 第8.6条（模块化设计原则）
【推荐】常量复用层级

层级	存储位置	示例
跨应用共享常量	二方库client.jar的com.xxx.constant包下	GlobalErrorCode
应用内共享常量	一方库模块的constants目录	OrderStatusEnum
子工程内共享常量	子工程src/main/resources/constants目录	SubProjectConfig
包内共享常量	当前包下的constant子包	com.xxx.module.util.constant
类内私有常量	类内部private static final定义	MAX_RETRY_COUNT = 3

【推荐】枚举使用场景，当变量值有明确范围且需附加属性时，必须使用枚举类。

代码格式规范

【强制】大括号格式，空代码块：直接写成{}，无需换行。

【强制】括号与字符间距，括号与相邻字符间不得添加空格。

【强制】关键字与括号间距，if/for/while/switch/do等关键字与括号间需加空格。

【强制】运算符间距，所有二目、三目运算符（如=,&&,+,?:）两侧需加空格。

【强制】注释格式，双斜线//与注释内容间保留一个空格。

【强制】单行字符限制与换行规则，单行不得超过120 字符，换行需遵循：

场景	换行规则	示例
长表达式	第二行缩进4空格，运算符与下文换行	String name = firstName + lastName + middleName;
方法调用	点符号与下文换行	userService.getUserById(userId) .thenApply(...);
多参数	逗号后换行	sendMessage(param1, param2, param3, param4);
禁止在括号前换行	括号必须紧跟内容	反例：execute(param1, param2);

【强制】参数逗号间距，方法定义或调用时，多个参数逗号后需加空格。

【强制】IDE全局设置，文件编码：UTF-8，换行符：Unix格式（\n），禁止使用Windows格式（\r\n）。

【推荐】禁止对齐空格，无需通过额外空格对齐不同行的字符。

【推荐】代码块空行规则，需加空行情况：

变量定义组与执行语句组之间；

不同业务逻辑或语义的代码段之间。

禁止空行：相同逻辑的连续代码段。

方法和实现规约

【强制】访问静态成员时禁止通过对象实例调用，必须使用类名直接访问

优化：直接通过类名访问静态变量/方法可避免JVM多级解析过程，提升执行效率。编译器对类名访问方式有明确优化路径，而对象引用需要多一步解析实例所属类。

【强制】覆写方法必须显式添加@Override注解

优化：该注解会触发编译期严格校验，确保方法签名与父类完全一致，避免因拼写错误导致无效覆盖。例如getObjec()（字母o）与getObject()（数字0）的拼写差异可通过注解快速识别。

【强制】可变参数使用规范升级

优化：可变参数仅允许在相同类型、相同业务场景下使用，且必须作为参数列表末位。禁止定义Object...类型的可变参数，否则会导致类型不安全问题。建议优先使用集合类型替代可变参数。

【强制】接口兼容性升级方案

优化：对外暴露的接口过时后，必须提供新旧版本共存期（如3个版本迭代周期），并在@Deprecated注解中明确标注替代接口的完整类路径和迁移指南。重要接口需提供自动化迁移工具。

【强制】过时API替换流程

优化：使用@Deprecated标记的方法/类必须在代码审查阶段记录替换方案。例如URLDecoder.decode(String)已过时，需替换为双参数版本decode(String source, String enc)，并在替换时显式处理UnsupportedEncodingException。

【强制】equals方法空指针防护

优化：推荐使用java.util.Objects.equals(Object a, Object b)进行对象比较，该方法已内置空指针防护机制。禁止将可能为null的对象放在equals方法调用方，如variable.equals("literal")应改为"literal".equals(variable)。

【强制】包装类比较统一规范

优化：所有包装类比较必须使用equals方法，包括在-128~127范围内的Integer对象。虽然该区间内==比较可能成立，但会导致代码可读性下降和缓存机制依赖风险。

【强制】数据类型使用标准细化

POJO属性必须使用包装类型：应对数据库NULL值场景，例如Long userId允许为null表示未分配，而基本类型long会默认0导致数据误解

RPC参数/返回值规范：包装类型的null可明确表示远程调用失败等异常状态，而基本类型默认值会掩盖问题

局部变量推荐基本类型：栈内存访问效率更高，避免自动装箱带来的性能损耗

【强制】POJO属性默认值禁令

优化：属性默认值会导致数据污染，例如gmt_create字段默认new Date()会造成全表更新时意外修改创建时间。应通过数据库DEFAULT约束或业务层显式赋值实现。

【强制】序列化版本号管理规范

优化：兼容性升级时保持serialVersionUID不变，非兼容升级必须变更UID并同步更新所有序列化/反序列化端。重大变更需通过版本路由机制逐步迁移。

【强制】构造方法纯净性原则

优化：构造方法仅允许进行参数校验和简单赋值操作，复杂初始化逻辑必须封装在init()方法中，并通过状态标识防止重复初始化。

【强制】toString实现规范

优化：POJO必须实现包含所有核心字段的toString()，继承场景需先调用super.toString()。建议使用IDE生成模板并添加@ToString(callSuper=true)注解。

【推荐】字符串分割安全校验

优化：对String.split()结果进行长度校验，特别是通过str.split(";", -1)保留空元素，避免直接访问分割后的数组索引。

【推荐】方法组织规范

优化：重载方法应按参数数量升序排列，相同功能方法集中放置。例如构造方法按参数数量从少到多排列，便于阅读时理解默认参数逻辑。

【推荐】类成员方法排序策略

优化：公有方法应遵循「阅读动线」原则，将核心业务方法置于类首部，工具方法次之，最后放置getter/setter。DAO层类可把数据访问方法前置。

【推荐】属性存取方法规范

优化：setter方法参数名必须与字段名完全一致（含大小写），禁止添加参数转换等业务逻辑。例如setAmount(BigDecimal amount)中直接this.amount = amount。

【推荐】字符串拼接性能优化

优化：循环体内字符串操作必须使用StringBuilder，预估初始容量（如new StringBuilder(128)）避免扩容。单次操作可直接用+连接。

【推荐】final关键字适用场景

优化：

工具类声明为final class

POJO字段添加final修饰时需配合构造方法初始化

方法参数final仅用于Lambda表达式等特殊场景

【推荐】对象拷贝规范

优化：深拷贝推荐使用构造函数复制、序列化方案或工具类（如BeanUtils.copyProperties），禁用默认clone机制。必须实现Cloneable接口时，需递归调用子对象clone。

【推荐】访问控制层级规范

优化：工具类应声明private构造方法并通过抛出AssertionError防止反射实例化。protected修饰符仅用于明确需要子类继承的成员。

集合规约

【强制】hashCode与equals契约的深度规范

重写equals必须同步重写hashCode：违反此规则会导致HashSet/HashMap等哈希集合无法正确识别对象唯一性。例如两个逻辑相等的对象若hashCode不同，会被存入哈希表的不同桶中，破坏集合核心机制

Set存储对象强制实现双方法：HashSet底层依赖HashMap实现，添加元素时先比较hashCode再调用equals，缺一不可

Map键对象特殊要求：作为键的对象若未正确实现这两个方法，会导致无法检索已有键值对。反例：map.put(new Key(1), "A"); map.get(new Key(1))返回null

技术验证：使用IntelliJ IDEA的Generate equals() and hashCode()工具自动生成，确保符合《Effective Java》第10、11条规范

【强制】subList类型安全禁区

禁止强转subList结果：ArrayList.subList()返回的是RandomAccessSubList实例（JDK8+为SubList），与ArrayList无继承关系。强制转换会触发ClassCastException

视图层危险操作：SubList直接引用原列表的modCount字段，任何结构性修改（如原列表增减元素）都会导致子列表遍历时抛出ConcurrentModificationException

解决方案：需独立使用子列表时应创建新集合List<Integer> sub = new ArrayList<>(list.subList(0, 5)); // 防御性复制

【强制】subList并发修改防御策略

原列表冻结原则：获取subList后，原列表应视为不可变对象。必须通过子列表进行修改操作

事务性操作规范：批量修改需通过subList的clear()/addAll()等方法进行，避免直接操作原列表

【强制】集合转数组类型安全规范

必须使用类型化数组：toArray(T[] a)方法通过反射获取数组类型信息，无参toArray()返回Object[]可能引发类型转换异常。

【强制】Arrays.asList陷阱规避

固定大小限制：返回的ArrayList（Arrays内部类）继承AbstractList但未重写add/remove方法，调用时抛出UnsupportedOperationException

安全转换方案：

List<String> list = new ArrayList<>(Arrays.asList("a", "b")); // 包装创建可修改集合

【强制】泛型通配符PECS原则强化

生产者使用<? extends T>：仅允许读取，禁止写入（编译器阻止add(null)外的所有插入操作）

消费者使用<? super T>：允许写入T及其子类，读取时只能转为Object

【强制】集合遍历修改规范

fail-fast机制：ArrayList等集合的迭代器直接检测modCount，快速失败而非冒险继续

安全删除模式：

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
if (it.next() == 1) {
it.remove();  // 通过迭代器自身方法删除
}
}
```

并发场景方案：使用CopyOnWriteArrayList等线程安全集合

【强制】Comparator可比性三定律

自反性违反示例：

```
Comparator<Integer> broken = (x, y) -> (x > y) ? 1 : -1;
// 当x=y时返回-1，违反x.compareTo(y) == -y.compareTo(x)
```

正确实现模式：

```
Comparator<Integer> valid = Comparator.comparingInt(x -> x);
```

排序稳定性：JDK8+推荐使用thenComparing构建链式比较器

【推荐】Map遍历性能优化方案

keySet二次查询损耗：map.get(key)需要重新计算hash定位桶位置

entrySet效率优势：直接遍历Node<K,V>数组，避免重复哈希计算

【推荐】Map null值处理矩阵

实现类 Key允许null Value允许null线程安全 备注
HashMap是 是 否 链表长度>8转红黑树
ConcurrentHashMap 否 否 是 分段锁(CAS in JDK8+)
TreeMap 否 是 否 基于红黑树实现有序
LinkedHashMap 是 是 否 保持插入顺序或访问顺序
【参考】集合有序性分类表
集合类型	排序性	稳定性	实现原理
ArrayList	无序	稳定	插入顺序
LinkedList	无序	稳定	插入顺序
TreeSet	有序	不稳定	比较器排序
LinkedHashSet	无序	稳定	插入顺序+哈希表
【参考】集合去重性能对比
List.contains()去重：时间复杂度O(n²)，1万元素需1亿次比较
HashSet去重：时间复杂度O(n)，基于哈希表快速定位
【推荐】数组边界防御性编程
并发规约
【强制】单例模式线程安全实现方案，双重检查锁+volatile（JDK5+适用）：
原理说明：
volatile关键字确保对象初始化完成前对其他线程可见，防止DCL失效。枚举类天然线程安全且能防御反射破坏单例。
【强制】线程命名规范，使用Guava线程工厂或自定义命名规则：
ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
.setNameFormat("order-process-%d").build();
ExecutorService pool = new ThreadPoolExecutor(..., namedThreadFactory);
【强制】线程池资源管控，阻塞队列选择：任务具有依赖性时使用SynchronousQueue，无依赖用ArrayBlockingQueue, 拒绝策略： new ThreadPoolExecutor.AbortPolicy() // 默认抛出RejectedExecutionException new ThreadPoolExecutor.CallerRunsPolicy() // 主线程执行
【强制】线程池创建禁令
【强制】日期处理线程安全
【强制】锁性能优化策略
锁粒度控制：
使用StampedLock替代synchronized实现乐观读锁
分段锁实践（如ConcurrentHashMap的分段机制）
无锁数据结构：AtomicReferenceFieldUpdater
【强制】定时任务异常处理.
【推荐】异步转同步控制,确保异常处理。
【推荐】随机数生成优化。
【推荐】注意避免DCL缺陷
【参考】计数器选型策略
性能对比：

场景	选型	QPS（万次/秒）
低并发写	AtomicLong	12.5
高并发写	LongAdder	23.8
一写多读	volatile	98.6

原理差异：
LongAdder采用分段计数机制，通过分散热点提升写性能。

【参考】HashMap并发死链规避。

【参考】ThreadLocal内存泄漏防护，使用后必须清理，内存泄漏根源：ThreadLocalMap的Entry使用弱引用解决key泄漏，但value仍需要手动remove。

控制流规约

【强制】switch语句安全规范

技术实现：

```
switch (status) {
case NEW -> {
startOrder();
break;  // 显式终止
}
case PAID -> processPayment(); // 单行无需break（Java14+箭头语法特性）
case CLOSED:  // 穿透场景需注释
logClosed();
// fall through 明确标注穿透到DELETED
case DELETED:
cleanup();
break;
default -> throw new IllegalStateException("Unexpected value: " + status);
}
```

风险案例：

未添加default的分支在新增枚举值时可能导致逻辑漏洞（如订单状态新增REFUNDED但未处理）

【强制】控制流语句大括号规范

避免dangling else问题（else与最近的if匹配）

代码diff时更清晰（Git等版本工具的行级变更识别）

【推荐】多条件逻辑优化策略，卫语句重构或设计模式应用：

【推荐】条件表达式可读性优化，坏味道代码，条件语句包含很多逻辑，应该封装方法，生成判断对象，使用计算结果作为判断对象。

【推荐】条件判断防误写方案。

【推荐】方法覆盖规范，静态方法隐藏问题，私有方法限制。

【推荐】静态属性初始化。

【推荐】循环体性能优化，资源外提，异常处理。

【推荐】接口入参防护，批量操作校验，防御性复制。

【参考】参数校验决策矩阵

校验场景	必须校验	建议方式	例外情况
公开API参数	✓	JSR 303注解校验	已通过网关校验的参数
核心业务方法	✓	Preconditions.checkArgument	内部循环调用且参数可控
数据库写操作	✓	空值/长度校验	DAO层基础校验已覆盖
工具类方法	✓	Objects.requireNonNull	private辅助方法且调用方可靠
高频调用方法	△	文档约束+单元测试	性能敏感场景

注释规约
【强制】Javadoc注释规范
技术实现：
```
/**
* 订单状态变更处理器
*
* @author 张三 (zhangsan@xxx.com)
* @since 2023-08-01
*/
public class OrderStateHandler {
/**
* 计算订单优惠金额
*
* @param order 订单实体，不可为空
* @param user 用户信息，包含会员等级
* @return 优惠金额（单位：分），始终为非负数
* @throws IllegalArgumentException 当订单金额不合法时抛出
*/
public int calculateDiscount(Order order, User user) {
// ...
}
}
```

原理说明：
IDE智能提示：在IntelliJ/VSCode中悬停方法名可查看参数说明
文档生成：通过mvn javadoc:javadoc生成标准API文档
代码溯源：@author标注责任人，@since标记功能引入版本

【强制】抽象方法注释规范
示例模板：
```
public interface PaymentService {
/**
* 执行支付操作
*
* <p>实现类需保证接口幂等性，支持重复调用</p>
*
* @param request 支付请求参数，必须包含有效的订单号
* @return 支付流水号，可用于后续查询
* @throws PaymentException 当支付通道不可用时抛出
* @see RetryPolicy 重试策略配置
*/
String pay(PaymentRequest request) throws PaymentException;
}
```

关键要素：
<p>标签描述实现约束
@see关联相关类
异常类型需明确业务场景

【强制】类创建信息标注
现代工程实践：

```
/**
* 分布式锁工具类
*
* @author 李四（lisi@xxx.com）
* @since 2022-05-30 适配Redis 6.0新特性
*/
public class DistributedLockUtil {
// ...
}
```

注意事项：
多人协作时使用团队邮箱：@author Infrastructure Team (infra@xxx.com)
大版本变更时更新@since日期及说明

【强制】代码内注释格式
正确示例：

```
public void processData() {
// 步骤1：数据清洗（耗时操作）
cleanRawData();

/*
* 以下为特征计算逻辑：
* 1. 计算用户偏好得分
* 2. 生成时效性权重
*/
calculateFeature();
}
```

排版规范：
单行注释与代码间隔一个空行
多行注释每行缩进与代码块对齐
避免行尾注释（超过40字符换行）

【强制】枚举注释标准
业务枚举示例：

```
/**
* 工单处理状态
*/
public enum TicketStatus {
/** 已创建待分配 */
CREATED,

/** 已分配处理中（超时30分钟自动升级） */
PROCESSING,

/** 已解决待确认 */
RESOLVED,

/** 已关闭（终结状态） */
CLOSED
}
```

特殊状态说明：

状态流转规则
超时策略
终结状态标识

【推荐】中英文注释原则
正确示范：
// 使用快速排序算法优化性能（基准值pivot选择首中尾中位数）
quicksort(data);

// 调用ERP接口同步库存（ERP: Enterprise Resource Planning）
syncStockFromERP();

术语处理：
保留技术术语：DAO、RPC、JSON
中文解释专业缩写：GC（垃圾回收）

【参考】高质量注释标准
好注释特征：
// 采用Brent算法替代二分法，解决导数不连续问题（参见：XXX论文第5章）
double root = findRoot(f, a, b);

坏注释案例：
// 循环处理数据
for (Data d : dataList) { // 冗余注释，未说明业务目的
process(d);
}
【参考】注释精简原则
自解释代码优先：
// 优化前：
// 检查用户是否是VIP且余额大于100元
if (user.isVip() && user.getBalance() > 10000) {

// 优化后（通过方法名自解释）：
if (user.isEligibleForDiscount()) {

必要注释场景：
复杂算法实现原理
临时解决方案（hack）
第三方接口兼容逻辑

【参考】特殊标记管理
标记规范：
// TODO (#2345) 李四 2023-08-01: 替换为高性能解析库
private void parseData() {
// ...
}

// FIXME (王五 2023-08-02): 多线程环境下存在竞态条件
public void updateCounter() {
// ...
}
管理流程：

在IDE中配置TODO面板（IntelliJ: View → Tool Windows → TODO）
每日构建扫描特殊标记并生成报告迭代计划中分配标记处理任务
其他：
【强制】正则表达式预编译优化
【强制】随机数生成安全实践
【强制】时间获取最佳实践
System.nanoTime()不保证跨JVM实例的单调递增性
分布式系统应使用NTP服务同步时间
异常处理
【推荐】正确处理可预见的RuntimeException（如IndexOutOfBoundsException）
核心要点：
应通过预先检查避免（如检查数组索引范围），而非依赖try-catch。
【推荐】异常不能替代条件控制或流程控制
关键原因：
性能损耗：异常处理涉及栈跟踪生成，比if-else慢 1~2 个数量级。
【推荐】合理划分try-catch块的范围？
最佳实践：
稳定代码：如System.out.println无需包裹在try中。
非稳定代码：如 I/O、网络调用，需细分异常类型：
【强制】捕获异常后未处理问题。
风险：
错误被静默忽略，导致后续逻辑不可控（如数据不一致）。
【强制】事务代码中捕获异常后保证回滚
关键步骤：
在catch中标记事务回滚（如TransactionAspectSupport.currentTransactionStatus().setRollbackOnly()）。
重新抛出异常，确保事务管理器感知。
【强制】传统与JDK7+正确关闭资源的方式
【强制】禁止在finally块中使用return
【强制】强制要求，捕获的异常类型必须与方法抛出的异常完全匹配，或其父类。
【推荐】方法返回null时应避免调用方 NPE。
【强制】预防常见的 NPE 场景
核心场景与方案：

场景	解决方案
自动拆箱	使用包装类型或检查null
数据库查询结果	使用Optional包装结果
集合元素空值	遍历前检查元素非空
远程调用返回值	防御性空判断
级联调用（如a.b.c）	使用Optional或分段检查

【推荐】异常与错误码使用场景的决策矩阵：

场景	方案	原因
对外HTTP/API 接口	错误码（如HTTP 状态码）	客户端需明确处理结果，避免解析异常信息
应用内部	抛出业务异	快速定位问题，结合全

	常	局异常处理
跨应用 RPC 调用	Result 封 装结果	避免序列化异常对象的性能损耗

【强制】通过 DRY 原则避免重复代码
方法：抽象公共类/接口：如模板方法模式。
工具类封装：如校验逻辑、日志处理。
日志规约
【强制】统一日志门面规范
门面模式优势：
实现解耦：无需修改代码即可切换Log4j2/Logback/JUL
统一日志行为：通过logback.xml统一控制格式、过滤策略
【推荐】日志保留策略优化
日志切割配置：
环境 保留天数 存储方案
生产环境 ≥30天 本地+ELK集群归档
测试环境 15天 按需保留
开发环境 7天 本地存储
【推荐】日志输出性能优化
【推荐】日志去重配置
【推荐】异常日志规范
日志分析标准：
每条ERROR日志必须包含：
业务场景标识（如订单ID）
操作类型（如支付/退款）
关键参数摘要
完整异常堆栈
【推荐】生产环境日志分级管控
日志级别配额：
级别 允许场景 容量警戒线
ERROR 系统级故障（数据库宕机等） 无限制
WARN 业务异常（参数错误等） ≤1GB/天
INFO 核心业务流程 ≤10GB/天
DEBUG 仅临时排查时开启 立即关闭
【参考】日志级别语义规范
分级使用指南：
级别 使用场景 报警策略
ERROR 系统不可用（服务启动失败等） 触发PagerDuty呼叫
WARN 业务异常（用户输入校验失败等） 企业微信通知
INFO 关键业务节点（订单创建成功等） 仅记录，不报警
DEBUG 问题排查时临时开启 禁止生产环境开启
安全规约
【强制】 隶属于用户个人的页面或者功能必须进行权限控制校验。
说明： 防止没有做水平权限校验就可随意访问、修改、删除别人的数据，比如查看他人的私信内容。
【强制】 用户敏感数据禁止直接展示，必须对展示数据进行脱敏。
说明： 中国大陆个人手机号码显示为:1 37****0969， 隐藏中间 4 位， 防止隐私泄露。

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

反例： 某系统签名大量被恶意修改， 即是因为对于危险字符 # --没有进行转义， 导致数据库更新时， where后边的信息被注释掉， 对全库进行更新。

【强制】 用户请求传入的任何参数必须做有效性验证。

说明： 忽略参数校验可能导致：

page size 过大导致内存溢出

恶意 order by 导致数据库慢查询

缓存击穿

SSRF

任意重定向

SQL 注入， Shell 注入， 反序列化注入

正则输入源串拒绝服务 ReDoS

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

【强制】 禁止向 HTML 页面输出未经安全过滤或未正确转义的用户数据。

【强制】 表单、 AJAX 提交必须执行 CSRF 安全验证。

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

【强制】 URL 外部重定向传入的目标地址必须执行白名单过滤。

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

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

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

附 录 B
（资料性）
编码示例

B.1 编码示例

编码示例详见表B.1

表 B.1 编码示例

检查项	正例	反例

命名禁止使用特殊符号，禁止以下划线（_）或美元符号（$）开头或结尾	String userName;
	List<Order> orderList;
		String _user; // 下划线开头
		String $order; // 美元符号开头
		String name_; // 下划线结尾
		Object user@info; // 包含非法符号@

禁止混合语言命名，严禁使用拼音与英文混合命名	String taobaoOrder; // 国际品牌名
	String wechatPay; // 通用产品名
		int daZhePromotion; // 拼音+英文混合
		String 用户ID; // 纯中文命名
		String shangHaiUser; // 拼音首字母大写错误（应为shanghaiUser）

杜绝不规范缩写，禁止使用非行业共识的缩写	int maxRetryCount = MAX_RETRIES; // MAX为国际通用缩写
	String httpRequest; // HTTP为领域术语
		void calcCond() { ... } // condition缩写为cond
		String xmlParser; // XML应全大写（XMLParser）

类名格式 采用 UpperCamelCase 风格，如UserService。 **例外**：领域模型后缀需全大写（UserDO/OrderDTO）	class UserService { ... } // 大驼峰风格 class OrderDTO { ... } // 领域模型后缀全大写	class userService { ... } // 首字母未大写 class OrderDo { ... } // 后缀未全大写（应为OrderDO）
抽象类与异常类	abstract class AbstractController { ... } // Abstract前缀 class ValidationException extends Exception { ... } // Exception后缀	class ControllerBase { ... } // 后缀Base位置错误 class NetworkError { ... } // 未使用Exception后缀

表B.1 编码示例（续）

检查项	正例	反例

接口与实现类	interface UserService { ... } // 接口无后缀
	class UserServiceImpl implements UserService { ... } // Impl后缀
	@FunctionalInterface
	interface Runnable { ... } // -able结尾的能力接口
		interface UserServiceInterface { ... } // 冗余Interface后缀
		class UserServiceImp { ... } // Impl缩写错误

测试类命名	class UserServiceTest { ... } // Test结尾	class TestUserService { ... } // Test前缀位置错误 class UserServiceTesting { ... } // 非常规后缀
方法与变量格式	public void getUserInfo() { ... } // 方法名小驼峰 private String localValue; // 变量名小驼峰	public void GetUserInfo() { ... } // 首字母大写 String Local_Value; // 下划线分隔

布尔变量命名	public class User {
	private boolean active; // 直接表达状态
	}
		public class User {
		private boolean isActive; // 冗余is前缀
		}

数组声明格式	String[] validDomains = {"com", "cn"}; // 规范声明	String args[] = new String[10]; // 中括号位置错误

常量命名	public static final int MAX_RETRY_COUNT = 3; // 完整语义	public static final int maxCount = 3; // 未全大写
		public static final int MAX_CNT = 3; // 不规范缩写（CNT）

枚举规范	public enum WeekdayEnum { // Enum后缀
	MONDAY, TUESDAY // 全大写+下划线
	}
		public enum Weekday { // 缺少Enum后缀
		Monday // 未全大写
		}

Service/DAO方法前缀	User getUserById(Long id); // get前缀
	List<Order> listActiveOrders(); // list前缀
		User fetchUserById(Long id); // 未用get
		List<Order> getAllOrders(); // 未用list

领域模型后缀	public class UserVO { ... } // 视图对象 public class OrderDTO { ... } // 数据传输对象	public class UserVo { ... } // 后缀未全大写
命名体现设计模式	class OrderFactory { ... } // 工厂模式 class UserProxy { ... } // 代理模式	class OrderCreator { ... } // 未明确模式类型

表B.1 编码示例（续）

检查项	正例	反例

接口规范，接口方法不加public修饰符，Javadoc需说明方法契约。	public interface PaymentService {
	/**
	* 执行支付操作（需预先校验金额）
	*/
	void executePayment(BigDecimal amount); // 方法契约说明
	}
		public interface PaymentService {
		public void pay(); // 冗余public修饰符
		}

包名格式	package com.example.service; // 单数形式	package com.example.Services; // 首字母大写
		package com.example.utils; // 复数形式

禁止魔法值	public static final int MAX_RETRY_COUNT = 3; if (retryCount > MAX_RETRY_COUNT) { ... }	if (retryCount > 3) { ... } // 魔法值3未定义 String url = "http://10.0.0.1:8080/api"; // IP和端口未提取为常量

Long类型字面量格式	long userId = 1000L;
	Long orderId = 2000L;
		long userId = 1000; // 未加L后缀（编译警告）
		Long orderId = 2000l; // 小写l易与数字1混淆

常量按功能模块拆分	public final class OrderConstants {
	public static final int MAX_ORDER_QUANTITY = 100;
	}
		public class GlobalConstants { // 违反模块化原则
		public static final int MAX_ORDER = 100;
		public static final String PAYMENT_URL = "...";
		}

枚举替代离散值	public enum OrderStatusEnum {
	PENDING(1, "待支付"),
	PAID(2, "已支付"),
	CANCELLED(3, "已取消");
	
	private final int code;
	private final String desc;
	
	OrderStatusEnum(int code, String desc) { ... }
	}
		public class Order {
		private int status; // 1-待支付, 2-已支付（需查文档才能理解含义）
		}

表B.1 编码示例（续）

检查项	正例	反例

枚举方法封装	public enum FileTypeEnum {
	PDF("pdf", "application/pdf") {
	@Override
	public boolean isEditable() { return false; }
	},
	DOCX("docx", "application/msword") {
	@Override
	public boolean isEditable() { return true; }
	};
	
	public abstract boolean isEditable();
	}
		public class FileUtils { // 状态与行为分离，违反高内聚原则
		public static boolean isEditable(String fileType) { ... }
		}

大括号格式，空代码块：直接写成{}，无需换行	public void emptyMethod() {}	public void emptyMethod() {
		}
		public void emptyMethod() { }
		// 内部含空格

括号与字符间距	List<String> list = new ArrayList<>();	List<String> list = new ArrayList< >(); // 泛型括号加空格
		if (condition) { ... }
		// 条件括号两侧空格

关键字与括号间距	if (isValid) { ... }
	for (int i = 0; i < 10; i++) { ... }
		if(isValid) { ... } // if与(紧贴
		while(true){ ... } // while与(无空格

运算符间距	int sum = a + b;
	boolean flag = (a > b) ? true : false;
		int sum=a+b; // 等号与加减符无空格
		String name=firstName+lastName; // 运算符粘连

注释格式	// 用户状态校验 public void checkStatus() { ... }	//用户状态校验 // 无空格 // 用户状态校验 // 多余空格

单行字符限制与换行规则	/ 场景1：长表达式换行（缩进4空格，运算符下行）
	String fullName = firstName + lastName
	+ middleName;
	
	// 场景2：方法调用链换行（点符号下行）
	userService.getUserById(userId)
	.thenApply(...);
	
	// 场景3：多参数换行（逗号后换行）
	sendMessage(argument1, argument2,
	argument3, argument4);
		execute(// 括号前换行
		param1, param2);
		String name = firstName + // 运算符未下行
		lastName;

表B.1 编码示例（续）

检查项	正例	反例

参数逗号间距	public void send(String msg, int priority) { ... } // 定义时
	send(message, 1);
	// 调用时
		public void send(String msg,int priority) { ... } // 逗号后无空格
		send(message ,1);

IDE全局设置	文件编码：UTF-8 换行符：LF (\n)	换行符：CRLF (\r\n)

禁止对齐空格	private int userId;
	private String userName;
		private int userId; // 添加多余空格对齐
		private String userName;

代码块空行	public void processOrder() {
	// 变量定义组
	int orderId = 123;
	String status = "PAID";
	
	// 执行校验逻辑（空行分隔）
	validateOrder(orderId);
	
	// 执行支付逻辑（空行分隔）
	payOrder(orderId, status);
	}
		public void processOrder() {
		int orderId = 123;
		validateOrder(orderId); // 变量与执行语句间无空行
		
		payOrder(orderId); // 相同逻辑间冗余空行
		}

禁止通过对象实例访问静态成员	public class UserUtil {
	public static final int MAX_AGE = 120;
	}
	int maxAge = UserUtil.MAX_AGE; // 直接类名调用
		UserUtil userUtil = new UserUtil();
		int maxAge = userUtil.MAX_AGE; // 通过实例访问静态变量（编译警告）

@Override注解使用	@Override
	public void run() {
	// 方法实现
	}
		public void Run() {
		// 未使用@Override，字母大小写错误导致无效覆盖
		}

类型安全参数定义	public void logMessages(String... messages) { // 相同类型可变参数
	
	Arrays.stream(messages).forEach(System.out::println);
	}
		public void processParams(Object... params) { // 类型不安全
		// 需强制类型转换，可能引发ClassCastException
		}

过时接口声明	@Deprecated(since = "2.0", forRemoval = true)
	@see com.example.NewService#processOrder
	public interface OldService { // 明确替代方案
	void process();
	}
		@Deprecated
		public void oldMethod() { // 未提供迁移路径
		// 调用方无法知晓替代方案
		}

过时方法替换	String decoded = URLDecoder.decode(source, StandardCharsets.UTF_8); // 显式编码	String decoded = URLDecoder.decode(source); // 已废弃的单参数方法

表B.1 编码示例（续）

检查项	正例	反例

Null安全比较	if (Objects.equals(user.getName(), "admin")) { // 空安全
	// 业务逻辑
	}
		if (user.getName().equals("admin")) { // user.getName()可能为null
		// 可能抛出NPE
		}

equals方法使用	Integer a = 127, b = 127; boolean valid = a.equals(b); // 始终可靠	Integer c = 128, d = 128; boolean invalid = (c == d); // 超出缓存范围返回false

POJO属性定义	public class UserDTO {
	private Long id; // 允许null值
	private String name;
	}
		public class UserVO {
		private long id; // 默认0可能误导业务
		private String name;
		}

属性初始化禁令	public class OrderDO {
	private Date createTime; // 由数据库填充
	}
		public class OrderDO {
		private Date createTime = new Date(); // 全表更新时数据污染
		}

版本号管理	public class SerializedData implements Serializable {
	private static final long serialVersionUID = 1L; // 保持UID不变
	}
		public class UpdatedData implements Serializable {
		private static final long serialVersionUID = 2L; // 未同步修改反序列化端
		}

初始化逻辑分离	public class PaymentService {
	public PaymentService(Config config) {
	validateConfig(config); // 仅参数校验
	}
	private void init() { /* 复杂初始化 */ }
	}
		public class OrderService {
		public OrderService() {
		loadAllOrders(); // 构造方法执行IO操作
		}
		}

toString实现	@Override
	public String toString() {
	return "User{" + "id=" + id + ", name='" + name + '\'' + '}';
	}
		public String toString() {
		return "User@" + hashCode(); // 无法查看具体属性
		}

字符串分割安全校验	// 正例（长度检查）
	String[] parts = str.split(",", -1); // 保留空元素
	if (parts.length > 3) {
	String value = parts[3];
	}
		String[] parts = str.split(",");
		String value = parts[3]; // 可能越界

参数排序策略	public class FileUtils {
	public static void read(String path) { ... }
	public static void read(String path, Charset charset) { ... }
	}
		public class StringUtil {
		public static void format(String str, Object... args) { ... }
		public static void format(Locale locale, String str) { ... } // 参数顺序混乱
		}

表B.1 编码示例（续）

检查项	正例	反例

类成员排序核心方法前置	public class OrderService {
	// 核心业务方法
	public void createOrder() { ... }
	
	// 工具方法
	private void validateStock() { ... }
	}
		public class UserService {
		private void sendEmail() { ... }
		// 工具方法前置干扰
		public void register() { ... }
		}

存取方法规范参数命名一致	public void setUserName(String userName) { // 参数与字段同名
	this.userName = userName;
	}
		public void setUsername(String name) { // 参数名与字段不一致
		this.userName = name; // 可能引发序列化问题
		}

字符串拼接性能	StringBuilder sb = new StringBuilder(128); // 避免扩容
	for (String item : list) {
	sb.append(item);
	}
		String result = "";
		for (String item : list) { // 每次循环创建新对象
		result += item;
		}

对象拷贝规范深拷贝实现	public User copyUser(User source) {
	return new User(source.getId(), source.getName()); // 安全复制
	}
		public User cloneUser(User source) {
		return (User) source.clone(); // 未重写clone方法
		}

访问控制规范	public final class DateUtils {
	private DateUtils() {
	throw new AssertionError(); // 防止反射创建
	}
	public static LocalDate parse(String date) { ... }
	}
		public class StringUtil {
		public StringUtil() {} // 允许实例化无意义对象
		}

表B.1 编码示例（续）

检查项	正例	反例

hashCode 与 equals双方法重写	class User {
	private Long id;
	
	@Override
	public boolean equals(Object o) {
	if (this == o) return true;
	if (!(o instanceof User)) return false;
	return Objects.equals(id, ((User) o).id);
	}
	
	@Override
	public int hashCode() {
	return Objects.hash(id); // 相同id生成相同hashCode
	}
	}
		class ErrorUser {
		private Long id;
		
		@Override
		public boolean equals(Object o) { ... } // 未重写hashCode
		
		// 存入HashSet会出现重复元素
		public static void main(String[] args) {
		Set<ErrorUser> set = new HashSet<>();
		set.add(new ErrorUser(1L));
		set.add(new ErrorUser(1L)); // 两个对象会被认为不同
		System.out.println(set.size()); // 输出2
		}
		}

subList 类型安全禁区	List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3,4,5)); List<Integer> safeSub = new ArrayList<>(list.subList(0, 3)); // 创建新集合	List<String> origin = new ArrayList<>(Arrays.asList("A","B","C")); List<String> sub = origin.subList(0, 2); ArrayList<String> errorCast = (ArrayList<String>) sub; // 抛出ClassCastException

subList 并发修改防御	List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3));
	List<Integer> sub = list.subList(0, 2);
	sub.add(4); // 合法操作，原列表变为[1,2,4,3]
		List<Integer> origin = new ArrayList<>(Arrays.asList(1,2,3));
		List<Integer> sub = origin.subList(0, 2);
		origin.add(4); // 结构性修改
		sub.get(0); // 抛出ConcurrentModificationException

集合转数组类型安全转换	List<String> list = Arrays.asList("A", "B"); String[] arr = list.toArray(new String[0]); // 安全转换	Object[] objArr = list.toArray(); String[] errorArr = (String[]) objArr; // 运行时ClassCastException
Arrays.asList 陷阱固定大小限制	List<String> safeList = new ArrayList<>(Arrays.asList("A", "B")); safeList.add("C"); // 允许操作	List<String> fixedList = Arrays.asList("A", "B"); fixedList.add("C"); // 抛出UnsupportedOperationException

表B.1 编码示例（续）

检查项	正例	反例

泛型通配符PECS原则生产者消费者规则	public static <T> void copy(List<? super T> dest, List<? extends T> src) {
	dest.addAll(src); // src生产数据，dest消费数据
	}
		List<Number> numbers = new ArrayList<>();
		List<Integer> integers = Arrays.asList(1,2);
		numbers.addAll(integers); // 正确写法应使用通配符

集合遍历修改，安全删除模式	List<Integer> list = new ArrayList<>(Arrays.asList(1,2,3));
	Iterator<Integer> it = list.iterator();
	while (it.hasNext()) {
	if (it.next() == 2) {
	it.remove(); // 安全删除
	}
	}
		for (Integer num : list) {
		if (num == 2) {
		list.remove(num); // 抛出 ConcurrentModificationException
		}
		}

Comparator可比性三定律，自反性违反	// 正例（正确实现） Comparator<Integer> valid = Integer::compare;	Comparator<Integer> broken = (a, b) -> (a > b) ? 1 : -1; // a 等于 b 时返回 -1，违反 a.compareTo(b) == -b.compareTo(a)

entrySet高效遍历	Map<String, Integer> map = new HashMap<>();
	for (Map.Entry<String, Integer> entry : map.entrySet()) { // 直接获取键值对
	System.out.println(entry.getKey() + ": " + entry.getValue());
	}
		for (String key : map.keySet()) { // 每次get重新计算哈希
		System.out.println(key + ": " + map.get(key));
		}

数组边界防御编程	public static void safeAccess(int[] arr, int index) {
	if (index >= 0 && index < arr.length) {
	System.out.println(arr[index]);
	}
	}
		public static void riskyAccess(int[] arr, int index) {
		System.out.println(arr[index]);
		// 可能抛出 ArrayIndexOutOfBoundsException
		}

表B.1 编码示例（续）

检查项	正例	反例

单例模式线程安全双重检查锁+volatile	public class Singleton {
	private static volatile Singleton instance;
	
	private Singleton() {}
	
	public static Singleton getInstance() {
	if (instance == null) {
	synchronized (Singleton.class) {
	if (instance == null) {
	instance = new Singleton(); // volatile防止指令重排序
	}
	}
	}
	return instance;
	}
	}
	
	public enum SafeSingleton {
	INSTANCE; // JVM保证线程安全
	public void service() { ... }
	}
		public class BrokenSingleton {
		private static Singleton instance; // 缺失volatile
		
		public static Singleton getInstance() {
		if (instance == null) { // 可能返回未初始化对象
		synchronized (Singleton.class) {
		if (instance == null) {
		instance = new Singleton();
		}
		}
		}
		return instance;
		}
		}

Guava 线程工厂	ThreadFactory namedFactory = new ThreadFactoryBuilder() .setNameFormat("payment-service-%d").build(); ExecutorService pool = new ThreadPoolExecutor(..., namedFactory);	ExecutorService pool = Executors.newFixedThreadPool(4); // 线程名类似"pool-1-thread-2"，难以追踪问题
线程池资源阻塞队列选择	ExecutorService dependentPool = new ThreadPoolExecutor(4, 4, 0L, TimeUnit.MILLISECONDS, new SynchronousQueue<>()); ExecutorService batchPool = new ThreadPoolExecutor(4, 4, 0L, TimeUnit.MILLISECONDS, new ArrayBlockingQueue<>(1000));	

线程池创建禁令	ExecutorService safePool = new ThreadPoolExecutor(
	4, 8, 60L, TimeUnit.SECONDS,
	new ArrayBlockingQueue<>(1000),
	new ThreadPoolExecutor.AbortPolicy()
	);
		ExecutorService unmanagedPool = Executors.newCachedThreadPool(); // 无界线程数风险
		ExecutorService fixedPool = Executors.newFixedThreadPool(10); // 使用无界队列

表B.1 编码示例（续）

检查项	正例	反例

日期处理线程安全	public class SafeDateUtil {
	private static final ThreadLocal<DateFormat> localFormat = ThreadLocal.withInitial(
	() -> new SimpleDateFormat("yyyy-MM-dd")
	);
	
	public static String format(Date date) {
	return localFormat.get().format(date);
	}
	}
	
	// 正例（JDK8+）
	DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE;
	LocalDate date = LocalDate.parse("2023-01-01", formatter);
		public class DateUtil {
		private static final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd"); // 多线程format报错
		}

锁性能优化策略	StampedLock lock = new StampedLock();
	double read() {
	long stamp = lock.tryOptimisticRead();
	double v1 = x, v2 = y;
	if (!lock.validate(stamp)) {
	stamp = lock.readLock(); // 降级为悲观锁
	try { v1 = x; v2 = y; }
	finally { lock.unlockRead(stamp); }
	}
	return v1 + v2;
	}
	ConcurrentHashMap<String, Long> counterMap = new ConcurrentHashMap<>();
	counterMap.compute(key, (k, v) -> (v == null) ? 1 : v + 1);
		Hashtable<String, Long> unsafeMap = new Hashtable<>(); // 全局锁导致低吞吐量

定时任务异常处理	ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1); scheduler.scheduleAtFixedRate(() -> { try { processTask(); } catch (Exception e) { log.error("Task failed", e); } }, 0, 1, TimeUnit.SECONDS);	scheduler.scheduleAtFixedRate(this::processTask, 0, 1, TimeUnit.SECONDS); // 异常导致任务终止

异步转同步控制	CountDownLatch latch = new CountDownLatch(1);
	asyncTask().whenComplete((r, e) -> latch.countDown());
	boolean success = latch.await(5, TimeUnit.SECONDS);
		latch.await(); // 可能导致主线程永久阻塞

表B.1 编码示例（续）

检查项	正例	反例
随机数生成优化	ThreadLocalRandom.current().nextInt(100);	Random sharedRandom = new Random(); // 多线程竞争导致性能下降

ThreadLocal内存泄漏防护	try {
	threadLocal.set(user);
	process();
	} finally {
	threadLocal.remove(); // 强制清除
	}
		threadLocal.set(user); // static修饰时value会持续强引用user对象

Switch语句安全规范	// 正例（Java14+箭头语法）
	OrderStatus status = OrderStatus.PAID;
	switch (status) {
	case NEW -> {
	startOrder();
	break; // 显式终止
	}
	case PAID -> processPayment(); // 单行无需break
	case CLOSED: // 穿透场景需注释
	logClosed();
	// fall through 明确标注穿透到DELETED
	case DELETED:
	cleanup();
	break;
	default -> throw new IllegalStateException("Unexpected value: " + status); // 必须包含default
	}
	
	// 正例（枚举类型处理）
	enum Color { RED, GREEN, BLUE }
	switch (color) {
	case RED -> System.out.println("Stop");
	case GREEN -> System.out.println("Go");
	default -> handleUnknownColor();
	}
		// 反例（缺失default分支）
		switch (status) {
		case NEW: startOrder(); break;
		case PAID: processPayment(); break;
		// 新增REFUNDED状态时无处理逻辑
		}
		
		// 反例（未注释穿透）
		switch (status) {
		case CLOSED:
		logClosed(); // 无注释说明穿透意图
		case DELETED:
		cleanup(); // 易导致维护者误解逻辑
		break;
		}

控制流语句大括号规范	// 正例（避免dangling else）
	if (condition1) {
	if (condition2) {
	handleCaseA();
	}
	} else {
	handleCaseB(); // else明确对应外层if
	}
	
	// 正例（Git Diff友好）
	if (isValid) {
	execute(); // 大括号保证行级变更清晰
		// 反例（单行省略大括号）
		if (isValid) execute(); // 违反强制规范
		else logError();
		
		// 反例（缩进误导）
		if (condition1)
		if (condition2) callA();
		else callB(); // else实际匹配第二个if，易产生歧义

	}	

卫语句重构	public void process(Order order) {
	if (order == null) return; // 快速失败
	if (order.isDeleted()) return;
	
	// 核心逻辑
	calculateAmount(order);
	}
		if (user != null) {
		if (order != null) {
		if (payment != null) {
		// 业务逻辑...
		}
		}
		}

条件表达式可读性优化	boolean isEligible = age >= 18 && hasLicense && !isExpired;
	if (isEligible) { ... } // 提升可读性
	// 正例（方法封装
	if (isValidOrder(order)) { ... }
	
	private boolean isValidOrder(Order order) {
	return order != null
	&& order.getAmount() > 0
	&& !order.isCanceled();
	}
		if (user != null && user.getAge() > 18
		&& (user.getVipLevel() > 2 || couponCount > 0)) {
		// 难以快速理解
		}

条件判断防误写常量前置	// 正例（避免空指针）
	if ("SUCCESS".equals(status)) { ... } // 推荐写法
	
	// 正例（枚举比较）
	if (OrderStatus.PAID == status) { ... }
		// 反例（可能NPE）
		if (status.equals("SUCCESS")) { ... } // status为null时崩溃
		
		// 反例（赋值误写）
		if (isValid = true) { ... } // 实际是赋值操作

方法覆盖规范	class Parent {
	public void execute() { ... }
	}
	
	class Child extends Parent {
	@Override // 显式注解
	public void execute() { ... }
	}
		class Parent {
		public static void log() { ... }
		}
		
		class Child extends Parent {
		public static void log() { ... }
		// 静态方法隐藏，非覆盖
		}

静态属性初始化	class ConfigLoader {
	private static volatile Config instance;
	
	public static Config getInstance() {
	if (instance == null) {
	synchronized (ConfigLoader.class) {
	if (instance == null) {
	instance = loadConfig();
	}
	}
	}
	return instance;
	}
	}
		class UnsafeConfig {
		private static Config instance;
		
		public static Config getInstance() {
		if (instance == null) { // 多线程可能重复初始化
		instance = loadConfig();
		}
		return instance;
		}
		}

循环体性能优化	// 正例（避免重复创建对象）
	StringBuilder sb = new StringBuilder(1024);
	for (String item : list) {
	sb.append(item); // 资源复用
	}
		// 反例（循环内创建对象）
		for (int i = 0; i < 1000; i++) {
		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd"); // 应提到循环外

	// 正例（批量提交）
	int BATCH_SIZE = 100;
	List<Order> buffer = new ArrayList<>(BATCH_SIZE);
	for (Order order : orders) {
	buffer.add(order);
	if (buffer.size() >= BATCH_SIZE) {
	batchInsert(buffer);
	buffer.clear();
	}
	}
		sdf.format(new Date());
		}

接口入参防护防御性复制	// 正例（不可变集合）
	public void batchProcess(List<Order> orders) {
	List<Order> copy = Collections.unmodifiableList(
	new ArrayList<>(Objects.requireNonNull(orders))
	);
	// 使用copy进行操作
	}
	
	// 正例（深度校验）
	public void updateUser(User user) {
	if (user == null || user.getId() == null) {
	throw new IllegalArgumentException();
	}
	// 核心逻辑...
	}
		public void process(List<String> list) {
		list.remove(0); // 修改原始数据
		// 业务逻辑...
		}

Javadoc 注释规范	/**
	* 订单状态变更处理器
	*
	* @author 张三 (zhangsan@xxx.com)
	* @since 2023-08-01
	*/
	public class OrderStateHandler {
	/**
	* 计算订单优惠金额
	*
	* @param order 订单实体，不可为空
	* @param user 用户信息，包含会员等级
	* @return 优惠金额（单位：分），始终为非负数
	* @throws IllegalArgumentException 当订单金额不合法时抛出
	*/
	public int calculateDiscount(Order order, User user) {
	// ...
	}
	}
		public class OrderHandler {
		// 计算折扣
		public int calc(Order o, User u) {
		/* 参数校验 */
		if (o == null) throw new Exception();
		// ...计算逻辑
		}
		}

抽象方法注释规范	public interface PaymentService { /**	interface UserService { /**

	* 执行支付操作
	*
	* 实现类需保证接口幂等性，支持重复调用
	*
	* @param request 支付请求参数，必须包含有效的订单号
	* @return 支付流水号，可用于后续查询
	* @throws PaymentException 当支付通道不可用时抛出
	* @see RetryPolicy 重试策略配置
	*/
	String pay(PaymentRequest request) throws PaymentException;
	}
		* 更新用户信息
		* @param user 用户对象
		*/
		void update(User user);
		}

类创建信息标注	/**
	* 分布式锁工具类
	*
	* @author 李四（lisi@xxx.com）
	* @since 2022-05-30 适配Redis 6.0新特性
	*/
	public class DistributedLockUtil {
	// ...
	}
		public class CacheUtil { // 无@author和@since
		// ...
		}

代码内注释格式	public void processData() {
	// 步骤1：数据清洗（耗时操作）
	cleanRawData();
	
	/*
	* 以下为特征计算逻辑：
	* 1. 计算用户偏好得分
	* 2. 生成时效性权重
	*/
	calculateFeature();
	}
		void process() {
		cleanData(); //数据清洗
		/*计算特征开始*/
		calcFeature(); //步骤1
		//步骤2
		//...
		}

枚举注释标准	/**
	* 工单处理状态
	*/
	public enum TicketStatus {
	/** 已创建待分配（超时30分钟自动升级） */
	CREATED,
	
	/** 已分配处理中（优先级：P0<30min, P1<2h） */
	PROCESSING
	}
		public enum Status {
		NEW, // 新状态
		OLD // 旧状态
		}

中英文注释原则	// 使用快速排序优化性能（pivot选择首中尾中位数）
	quicksort(data);
	
	// 调用ERP接口同步库存（ERP: Enterprise Resource Planning）
		// 使用kuaisu排序
		quickSort(data); // 快速排序
		
		// Call ERP API
		syncERP();

	syncStockFromERP();	

高质量注释标准	// 采用Brent算法替代二分法（解决导数不连续问题，参见《数值分析》第5章）
	double root = findRoot(f, a, b);
		// 循环处理数据
		for (Data d : dataList) {
		process(d);
		}

注释精简原则	// 优化后：
	if (user.isEligibleForDiscount()) { ... }
		// 优化前：
		// 检查用户是否是VIP且余额大于100元
		if (user.isVip() && user.getBalance() > 10000) { ... }

特殊标记管理	// TODO (#JIRA-123) 李四 2025-06-30: 替换为OAuth2.0协议
	void auth() { ... }
	
	// FIXME (王五 2025-05-28): 高并发下计数器溢出风险
	AtomicInteger counter = new AtomicInteger();
		// TODO 这里需要优化
		// FIXME 有问题

正则表达式预编译优化	// 预编译正则表达式（提升10倍+性能）
	public class RegexUtils {
	private static final Pattern EMAIL_PATTERN = Pattern.compile("^[A-Za-z0-9+_.-]+@(.+)$");
	
	public static boolean isValidEmail(String email) {
	return EMAIL_PATTERN.matcher(email).matches();
	}
	}
	// 使用示例
	boolean valid = RegexUtils.isValidEmail("test@example.com");
		// 每次调用重新编译（性能杀手）
		public boolean validatePhone(String phone) {
		return phone.matches("^1[3-9]\\d{9}$");
		// 内部每次new Pattern()
		}
		
		// 多线程场景隐患
		Pattern dynamicPattern = Pattern.compile(userInput);
		// 未处理特殊字符如"(?"导致ReDOS攻击

随机数生成安全实践	// 明确种子初始化（避免可预测性）
	private static final ThreadLocal<Random> secureRandom = ThreadLocal.withInitial(() -> {
	return new SecureRandom(); // 使用SecureRandom
	});
	
	// 生成会话Token
	String token = UUID.randomUUID().toString().replace("-", "");
		// 可预测随机序列（安全漏洞）
		int code = new Random().nextInt(9000) + 1000; // 验证码生成
		
		// 共享Random实例（线程竞争）
		public static Random globalRandom = new Random();
		// 多线程调用导致性能下降和序列重叠

时间获取最佳实践	// 耗时统计（纳秒级精度）
	long start = System.nanoTime();
	performComplexCalculation();
	long durationNs = System.nanoTime() - start;
	
	// JDK8时间API（明确语义）
	Instant transactionTime = Instant.now(); // 带时区信息的时间点
		// 错误时间计算方式
		long endTime = System.currentTimeMillis() + 60000; // 易受系统时间回拨影响
		
		// 误用nanoTime跨实例
		long clusterTime = System.nanoTime(); // 不同

		JVM实例的nanoTime无关联性

正确处理可预见的RuntimeException	// 数组索引检查
	List<String> data = getData();
	int index = 5;
	if (index >= 0 && index < data.size()) { // 前置校验
	String value = data.get(index);
	}
	
	// Map键值检查
	Map<String, User> userMap = loadUsers();
	String key = "user_1001";
	if (userMap.containsKey(key)) { // 显式存在性检查
	User target = userMap.get(key);
	}
		try {
		String value = data.get(10);
		// 高风险操作
		} catch (IndexOutOfBoundsException e) { // 异常兜底
		log.error("索引越界", e);
		}

异常不能替代条件控制	public void processFile(File file) {
	if (file.exists() && file.canRead()) { // 前置检查
	readContent(file);
	} else {
	handleMissingFile();
	}
	}
		for (int i = 0; i < 100; i++) {
		try {
		process(dataArray[i]);
		// 依赖异常终止循环
		} catch (ArrayIndexOutOfBoundsException e) {
		break;
		}
		}

合理划分try-catch范围	try (Connection conn = dataSource.getConnection()) { // 资源获取
	Statement stmt = conn.createStatement();
	ResultSet rs = stmt.executeQuery("SELECT...");
	// 结果集处理（稳定操作无需try）
	while (rs.next()) {
	System.out.println(rs.getString(1)); // 稳定输出
	}
	} catch (SQLException e) { // 精确捕获I/O异常
	throw new DataAccessException(e);
	}
		try { // 过度包裹
		String name = user.getName();
		System.out.println(name);
		// 稳定操作
		saveToDatabase(user);
		// 高风险操作
		} catch (Exception e) { // 过度宽泛
		// 无法区分错误类型
		}

禁止静默忽略异常	try {
	remoteService.call();
	} catch (ServiceException e) {
	// 记录完整上下文
	log.error("调用失败，参数：{}", request, e);
	// 恢复现场
	rollbackTransaction();
	// 重新抛出
	throw new BusinessException("业务处理失败", e);
	}
		try {
		saveOrder(order);
		} catch (SQLException e) {
		// 仅打印无处理（导致数据不一致）
		e.printStackTrace();
		}

事务异常回滚规范	@Transactional
	public void placeOrder(Order order) {
	try {
		@Transactional
		public void process() {
		try {

	inventoryService.deductStock(order);
	paymentService.charge(order);
	} catch (BusinessException e) {
	// 标记回滚
	
	TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
	// 重新抛出
	throw e;
	}
	}
		// 业务操作
		} catch (Exception e) {
		// 未标记回滚导致部分提交
		log.error("操作失败", e);
		}
		}

资源关闭规范	try (FileInputStream fis = new FileInputStream("data.txt");
	BufferedReader reader = new BufferedReader(new InputStreamReader(fis))) {
	// 自动关闭资源
	} catch (IOException e) {
	// 异常处理
	}
	BufferedReader br = null;
	try {
	br = new BufferedReader(new FileReader("file.txt"));
	// 读写操作
	} catch (IOException e) {
	// 处理异常
	} finally {
	if (br != null) {
	try { br.close(); } catch (IOException e) { /* 日志记录 */ }
	}
	}
		FileOutputStream fos = new FileOutputStream("temp");
		fos.write(data); // 未关闭流导致资源泄漏

finally块规范	public String readData() {
	String result = "";
	BufferedReader br = null;
	try {
	br = new BufferedReader(...);
	result = br.readLine();
	} catch (IOException e) {
	// 异常处理
	} finally {
	// 仅执行清理
	if (br != null) try { br.close(); } catch (IOException e) {}
	}
	return result;
	}
		public int calculate() {
		try {
		return complexCalc();
		} finally {
		return 0; // 覆盖正常返回值
		}
		}

异常类型精确匹配	try {
	parseJson(input);
	} catch (JsonSyntaxException e) { // 特定异常
	handleInvalidFormat();
	}
		try {
		// 业务代码
		} catch (Exception e) { // 捕获所有异常
		// 无法针对性处理
		}

空值安全实践	public List<String> getNames() {	public List<String> fetchData() {

	List<String> result = queryFromDB();
	return result != null ? result : Collections.emptyList(); // 空集合代替null
	}
	
	// Optional方案
	public Optional<User> findUser(String id) {
	return Optional.ofNullable(userRepository.findById(id));
	}
		// 可能返回null
		return dao.query();
		}

DRY 原则实施	public abstract class ReportGenerator {
	// 公共流程
	public final void generate() {
	init();
	loadData();
	format();
	export();
	}
	// 子类实现差异化步骤
	protected abstract void format();
	}
	public final class ValidationUtils {
	public static void checkArgument(boolean expr, String msg) {
	if (!expr) throw new IllegalArgumentException(msg);
	}
	}
		// ServiceA
		public void processOrder(Order o) {
		if (o == null) throw new IllegalArgumentException();
		if (o.getAmount() <=0) throw new IllegalArgumentException();
		// 业务逻辑...
		}
		
		// ServiceB
		public void updateUser(User u) {
		if (u == null) throw new IllegalArgumentException();
		if (u.getId() == null) throw new IllegalArgumentException();
		// 重复校验逻辑...
		}

统一日志门面规范	import org.slf4j.Logger;
	import org.slf4j.LoggerFactory;
	
	public class OrderService {
	// 使用门面API
	private static final Logger logger = LoggerFactory.getLogger(OrderService.class);
	
	public void process() {
	logger.info("订单创建成功，ID：{}", orderId);
	try {
	paymentService.charge();
	} catch (PaymentException e) {
	logger.error("支付失败，订单ID：{}，金额：{}", orderId, amount, e);
	}
	}
	}
		import org.apache.logging.log4j.LogManager;
		
		public class UserService {
		// 直接绑定Log4j2
		private static final org.apache.logging.log4j.Logger logger =
		
		LogManager.getLogger(UserService.class);
		}

日志保留策略优化	<!-- 生产环境配置 --> <appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender"> <file>logs/app.log</file> <rollingPolicy	<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender"> <encoder> <pattern>%msg%n</pattern>

	class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy"> <fileNamePattern>logs/archived/app-%d{yyyy-MM-dd}.%i.log.gz</fileNamePattern> <maxHistory>30</maxHistory> <!-- 保留30天 --> <totalSizeCap>50GB</totalSizeCap> </rollingPolicy> </appender>	</encoder> <!-- 未配置滚动策略，日志无限增长 --> </appender>
日志输出性能优化	<appender name="ASYNC" class="ch.qos.logback.classic.AsyncAppender"> <queueSize>1024</queueSize> <discardingThreshold>80</discardingThreshold> <!-- 队列剩余20%时丢弃DEBUG日志 --> <appender-ref ref="FILE"/> </appender>	// 同步输出到控制台 logger.debug("Debug信息：" + complexObject.toString()); // 字符串拼接损耗

日志去重配置	try (MDC.MDCCloseable closeable = MDC.putCloseable("traceId", UUID.randomUUID().toString())) {
	logger.error("订单处理失败，ID：{}", orderId); // 相同错误带不同traceId
	}
		for (int i = 0; i < 1000; i++) {
		logger.error("网络连接失败"); // 生成1000条相同日志
		}

异常日志规范	try {
	database.insert(order);
	} catch (SQLException e) {
	logger.error("[订单服务] 数据库写入失败，订单ID：{}，SQL：{}，参数：{}",
	orderId, sql, params, e); // 包含四要素
	}
		catch (IOException e) {
		logger.error("文件读取错误"); // 缺少上下文和堆栈
		}

生产环境日志分级管控	# Logback配置文件
	<logger name="com.example" level="INFO"/>
	<logger name="com.example.debug" level="DEBUG" additivity="false">
	<appender-ref ref="DEBUG_FILE"/>
	</logger>
	
	# 报警规则（ELK示例）
	PUT _ilm/policy/log_retention_policy {
	"policy": {
	"phases": {
	"hot": {
	"actions": {
	"rollover": {
	"max_size": "10GB" # INFO级别日志限额
	}
	}
	}
	}
	}
	}
		logger.debug("用户登录成功，ID：{}", userId); # 生产环境DEBUG输出