目录

[一、迪米特法则 1](#_Toc26125)

[1. 原则说明 1](#_Toc17318)

[2. 为什么重要 1](#_Toc30191)

[3. 典型违规场景解析 1](#_Toc23302)

[4. 博客系统实践场景：文章发布流程 1](#_Toc24674)

[二、依赖倒转原则 3](#_Toc30998)

[1. 原则说明 3](#_Toc8332)

[2. 为什么重要 3](#_Toc20902)

[3. 典型违规场景解析 3](#_Toc4948)

[4. 博客系统实践场景：消息通知模块 3](#_Toc11527)

[三、合成复用原则 5](#_Toc27305)

[1. 原则说明 5](#_Toc23752)

[2. 为什么重要 5](#_Toc25548)

[3. 典型违规场景解析 5](#_Toc23372)

[4. 博客系统实践场景：评论功能复用 5](#_Toc8472)

[总结 7](#_Toc5373)

# 一、迪米特法则

## 1. 原则说明

迪米特法则，又称“最少知识原则”（Least Knowledge Principle），强调：一个对象应只与其直接的“朋友”通信，不应与陌生的对象打交道。换句话说，调用链不应过长，不应通过一个对象再调用另一个对象的内部对象的方法。

该原则核心目标是减少对象之间的耦合度，提高系统模块的独立性和可维护性。

## 为什么重要

降低耦合性：减少对象之间的依赖，使模块间的变化不会引起连锁反应。增强封装性：对象只暴露必要的接口，隐藏内部实现细节。提升系统健壮性：减少因对象结构变化导致的连锁错误。方便测试和维护：局部改动不会影响调用该对象的其他模块。

## 3. 典型违规场景解析

很多系统中经常出现“链式调用”的情况，比如a.getB().getC().doSomething()，这种写法让a暴露了过多内部结构，破坏了封装。当内部结构变化时，所有调用该链的代码都必须改动，增加维护难度。

## 4. 博客系统实践场景：文章发布流程

文章发布控制器直接访问了用户的属性和通知细节，违背了迪米特法则，导致控制器对底层细节过度依赖，耦合度高。

错误示例

public class ArticleController {

public void publish(ArticleDto dto) {

User user = userService.getById(dto.getUserId());

if (!user.isVerified()) {

throw new RuntimeException("未认证用户不能发文");

}

articleRepository.save(new Article(dto));

notificationService.send(user.getEmail(), "文章发布成功！");

}

}

控制器既知道User内部状态，又直接调用通知服务，职责过多且耦合。

优化后的设计

public class ArticleController {

public void publish(ArticleDto dto) {

articleService.publish(dto);

}

}

public class ArticleService {

public void publish(ArticleDto dto) {

if (!userService.isUserVerified(dto.getUserId())) {

throw new RuntimeException("未认证用户不能发文");

}

articleRepository.save(new Article(dto));

notificationService.notifyUser(dto.getUserId(), "文章发布成功！");

}

}

控制器只负责调用服务层接口，不了解内部实现细节。ArticleService负责处理用户验证和通知逻辑，封装了间接依赖。变更用户验证规则或通知逻辑时，不影响控制器代码。提高了模块的独立性和职责单一原则。

# 二、依赖倒转原则

## 1. 原则说明

依赖倒转原则是面向对象设计的核心原则之一，核心内容包括两点：高层模块不应该依赖低层模块，二者都应该依赖抽象（接口或抽象类）;抽象不应该依赖细节，细节应该依赖抽象.通俗理解就是：程序设计中应面向接口编程，而非面向实现编程。

## 2. 为什么重要

降低耦合：高层业务逻辑不依赖具体实现，方便替换。增强灵活性：通过依赖抽象，可以轻松切换不同实现，如邮件、短信、推送等。持单元测试：通过接口可以方便地Mock具体实现，独立测试业务逻辑。扩展方便：新增功能只需新增实现类，无需修改现有业务代码，符合开闭原则。

## 3. 典型违规场景解析

系统中直接创建具体类实例，导致高层模块对低层细节直接依赖，后续如果需要切换实现，需修改大量代码。

## 4. 博客系统实践场景：消息通知模块

通知服务直接依赖EmailSender，未来如果想支持短信、站内信就需要修改通知服务代码，扩展性差。

错误示例

public class NotificationService {

private EmailSender sender = new EmailSender();

public void notifyUser(String email, String content) {

sender.send(email, content);

}

}

优化方案

java

public interface MessageSender {

void send(String target, String content);

}

public class EmailSender implements MessageSender {

public void send(String email, String content) {

// 邮件发送逻辑

}

}

public class NotificationService {

private final MessageSender sender;

public NotificationService(MessageSender sender) {

this.sender = sender;

}

public void notifyUser(String receiver, String content) {

sender.send(receiver, content);

}

}

依赖抽象接口MessageSender而非具体实现。通过构造函数注入不同的实现（邮件、短信等）。满足开闭原则：新增通知方式不修改通知服务代码。方便单元测试时Mock不同的发送实现。

# 三、合成复用原则

## 1. 原则说明

合成复用原则建议：优先使用对象组合（Has-A关系）来实现代码复用，而非通过类继承（Is-A关系）。组合通过“拥有”关系，让一个类内部包含另一个类的实例，以实现功能复用；继承是强耦合，容易引发类层次复杂和脆弱。

## 2. 为什么重要

更灵活：组合在运行时可动态替换组合对象，继承是静态绑定，灵活性差。降低耦合：组合只暴露有限接口，避免子类过度依赖父类实现。防止继承污染：继承可能导致子类继承了不必要的方法或状态，影响子类设计。支持职责分离：将功能模块拆成独立类，通过组合复用。

## 3. 典型违规场景解析

过度继承导致继承层次深、修改影响面大、不易理解和维护。

## 4. 博客系统实践场景：评论功能复用

如果所有需要评论功能的类都继承自Commentable，会造成耦合和扩展限制。

错误示例

java

public class Article extends Commentable { }

public class MessageBoard extends Commentable { }

继承限制了Article和MessageBoard的继承自由，也无法灵活切换评论实现。

优化方案=

java

public class CommentService {

public void addComment(String targetId, String userId, String content) {

// 保存评论

}

public List<Comment> listComments(String targetId) {

return commentRepository.findByTargetId(targetId);

}

}

public class ArticleService {

private final CommentService commentService;

public ArticleService(CommentService commentService) {

this.commentService = commentService;

}

public void commentArticle(String articleId, String userId, String content) {

commentService.addComment(articleId, userId, content);

}

}

设计亮点

评论功能作为独立服务，复用性强。通过组合为不同业务模块提供评论能力。避免了继承限制，增强代码灵活性和扩展性。代码职责更加清晰，维护成本更低。

# 总结

迪米特法则专注于减少对象间不必要的依赖，强调“只与直接朋友通信”。

依赖倒转原则强调面向接口编程，避免高层依赖具体实现。

合成复用原则建议用组合优于继承，实现更灵活的代码复用。