**1. 添加必要的注释**

* 所有的类都必须添加创建者和创建日期，以及简单的注释描述
* 方法内部的复杂业务逻辑或者算法，需要添加清楚的注释
* 一般情况下，注释描述类、方法、变量的作用
* 任何需要提醒的警告或TODO，也要注释清楚
* 如果是注释一行代码的，就用//;如果注释代码块或者接口方法的，有多行/\* \*\*/
* **一块代码逻辑如果你站在一个陌生人的角度去看,第一遍看不懂的话,就需要添加注释了**

## 2.日志打印规范

日志是快速定位问题的好帮手，是撕逼和甩锅的利器！打印好日志非常重要。**这些日志规范没遵守，就需要修改**：

* 日志级别选择不对。常见的日志级别有error、warn、info、debug四种，不要反手就是info哈
* 日志没打印出**调用方法**的入参和响应结果，尤其是跨系统调用的时候。
* 业务日志没包含关键参数,如userId,bizSeq等等,不方便问题排查
* 如果日志包含关键信息，比如手机号、身份证等，需要**脱敏处理**
* 一些不符合预期的情况，如一些未知异常（数据库的数据异常等），又或者**不符合业务预期的特殊场景，都需要打印相关的日志**

## 3. 命名规范

Java代码的命名应该**清晰、简洁和易于理解**。要注意**是否有命名不规范，不清晰的代码**。下面是一些命名规范的建议：

* 类和接口应该使用首字母大写的驼峰命名法
* 方法和变量应该使用小写的驼峰命名法
* 常量应该使用全大写字母和下划线
* 开发者是不是选择易于理解的名称给变量、类和方法进行命名

## 4.参数校验

要注意参数是否都做了校验，如userId非空检查、金额范围检查、userName长度校验等等。一般我们在处理业务逻辑的时候，要遵循先检查、后处理的原则。

如果你的数据库字段userName设置为varchar(16),对方传了一个32位的字符串过来，你不校验参数，**插入数据库直接异常**了。

很多bug都是因为**没做参数校验**造成的，这一项，**是重点关注的哈**：

## 5. 判空处理

* 获取对象的属性时，都要判空处理。要不然很多时候会出现空指针异常。

## 6. 异常处理规范

良好的异常处理可以确保代码的可靠性和可维护性。因此，异常处理也是一项重要规范。以下是一些异常处理的建议：

* 不要捕获通用的Exception异常，而应该尽可能捕获特定的异常
* 在捕获异常时，应该记录异常信息以便于调试
* 内部异常要确认最终的处理方式，**避免未知异常当作失败处理**。
* 在finally块中释放资源，或者使用try-with-resource
* 不要使用e.printStackTrace(),而是使用log打印。
* catch了异常，要打印出具体的exception，否则无法更好定位问题
* 捕获异常与抛出异常必须是完全匹配，或者捕获异常是抛异常的父类
* 捕获到的异常，不能忽略它，要打印相对应的日志
* 注意异常对你的代码层次结构的侵染（早发现早处理）
* 自定义封装异常，不要丢弃原始异常的信息Throwable cause
* 注意异常匹配的顺序，优先捕获具体的异常
* 对外提供APi时，要提供对应的错误码
* 系统内部应该抛出有业务含义的自定义异常，而不是直接抛出RuntimeException，或者直接抛出Exception\Throwable。

## 7. 模块化，可扩展性

**代码编写设计是否满足模块话，接口是否具有可扩展性**

比如你的需求是酱紫：是用户添加或者修改员工时，需要刷脸。那你是反手提供一个员工管理的提交刷脸信息接口？还是先思考：提交刷脸是不是通用流程呢？比如转账或者一键贴现需要接入刷脸的话，你是否需要重新实现一个接口呢？还是当前按业务类型划分模块，复用这个接口就好，保留接口的可扩展性。

如果按模块划分的话，未来如果其他场景比如一键贴现接入刷脸的话，不用再搞一套新的接口，只需要新增枚举，然后复用刷脸通过流程接口，实现一键贴现刷脸的差异化即可。

## 8. 并发控制规范

* 在使用并发集合时，应该注意它们的线程安全性和并发性能,如ConcurrentHashMap是线性安全的,HashMap就是非线性安全的
* 乐观锁,悲观锁防止数据库并发.乐观锁一般用版本号version控制,悲观锁一般用select …for update
* 如果是单实例的多线程并发处理,一般通过Java锁机制,比如sychronized ,reentrantlock
* 如果是同一集群的多线程并发处理,可以用Redis分布式锁或者走zookeeper
* 如果是跨集群的多线程并发处理,则考虑数据库实现的分布式锁。
* 在使用分布式锁的时候,要注意有哪些坑,比如redis一些经典的坑.

## 9. 单元测试规范

* 测试类的命名,一般以测试的类+Test,如:CalculatorTest.
* 测试方法的命名,一般以test开头+ 测试的方法,如testAdd.
* 单测行覆盖率一般要求大于75%.
* 单测一般要求包含主流程用例、参数边界值等校验用例
* 单测一般也要求包含中间件访问超时、返回空、等异常的用例,比如访问数据库或者Redis异常.
* 单测用例要求包含**并发、防重、幂等**等用例.

## 10. 代码格式规范

良好的代码格式，可以使代码更**容易阅读和理解**。下面是一些常见的代码格式化建议：

* 缩进使用四个空格
* 代码块使用花括号分隔
* 每行不超过80个字符
* 每个方法应该按照特定的顺序排列，例如：类变量、实例变量、构造函数、公共方法、私有方法等。

## 11. 接口兼容性

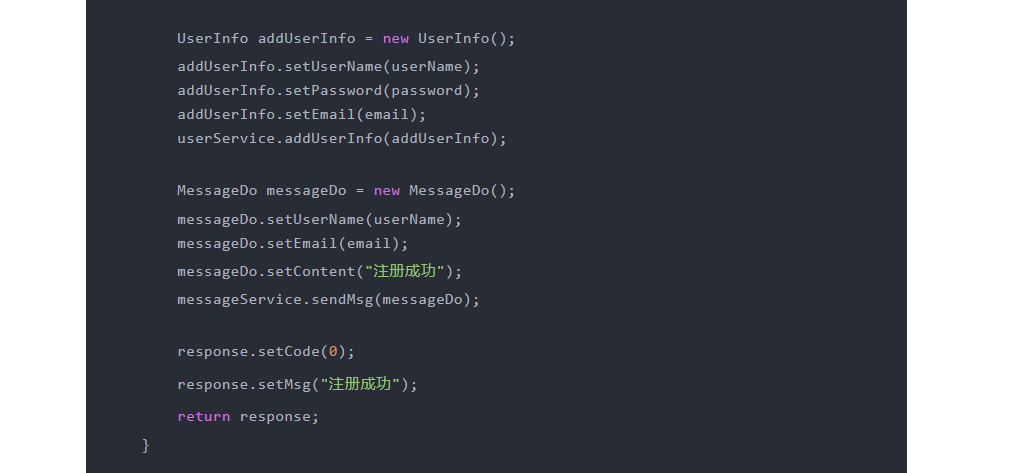
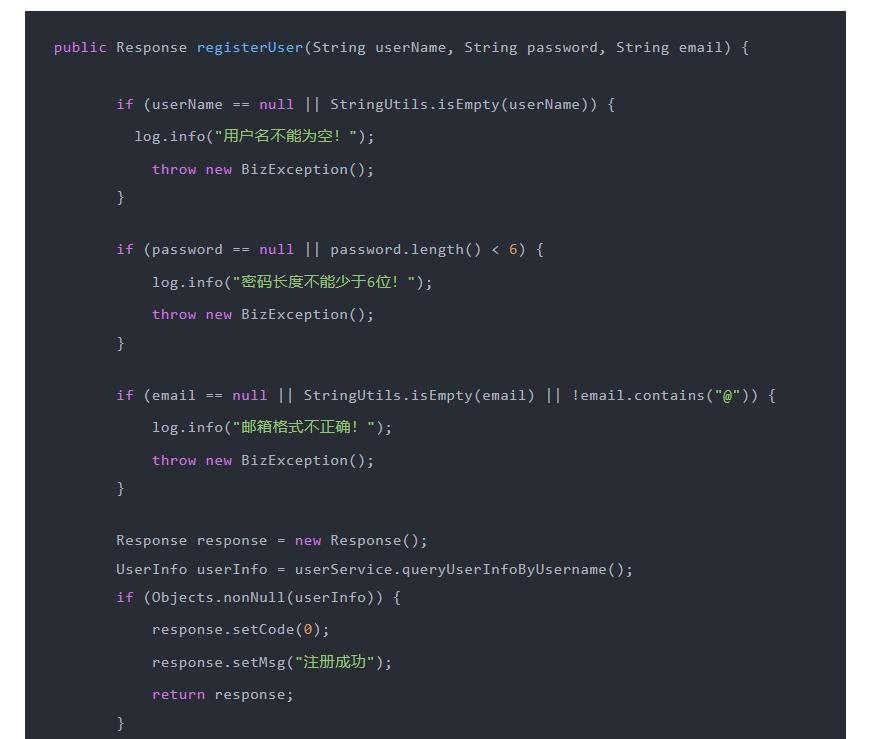
是否考虑到了**接口的兼容性**.因为很多bug都是因为修改了对外旧接口，但是却不做兼容导致的。关键这个问题多数是比较严重的，可能直接导致系统发版失败的。新手程序员很容易犯这个错误哦~

所以，如果你的需求是在原来接口上修改，尤其这个接口是对外提供服务的话，一定要考虑接口兼容。举个例子吧，比如dubbo接口，原本是只接收A，B参数，现在你加了一个参数C，就可以考虑这样处理：

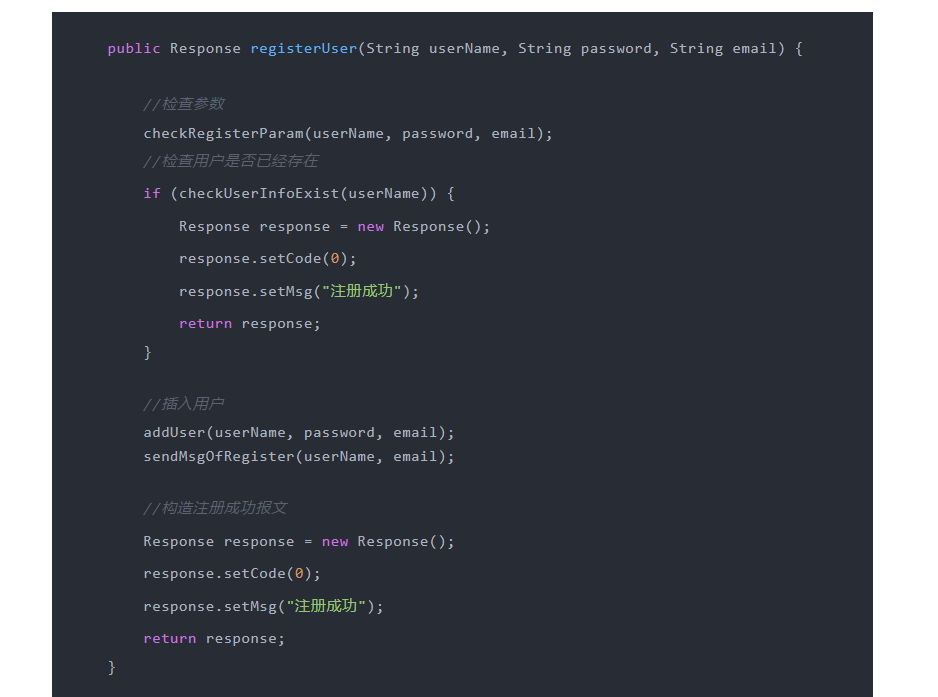


## 12. 程序逻辑是否清晰,主次是否够分明

关注程序逻辑是否清晰。比如，你的一个注册接口，**有参数校验、判断用户是否已经注册、插入用户记录、发送注册成功通知等功能**。如果你把所有所有功能代码塞到一个方法里面，程序逻辑就不清晰，**主次不够分明**，反例如下:



其实，以上这块代码，主次不够分明的点:参数校验就占registerUser方法很大一部分。正例可以划分主次，抽一下小函数，如下：



## 13. 安全规范

**代码是否存在安全性问题**。比如：

* **输入校验**：应该始终对任何来自外部的输入数据进行校验，以确保它们符合预期并且不会对系统造成伤害。校验应该包括检查数据的类型、大小和格式。
* **防范SQL注入攻击**:在使用SQL查询时，应该始终使用参数化查询或预处理语句，以防止SQL注入攻击。
* **防范跨站脚本攻击（XSS）**: 在Web应用程序中，应该始终对输入的HTML、JavaScript和CSS进行校验，并转义特殊字符，以防止XSS攻击。
* **避免敏感信息泄露**: 敏感信息（如密码、密钥、会话ID等）应该在传输和存储时进行加密，以防止被未经授权的人访问。同时，应该避免在日志、调试信息或错误消息中泄露敏感信息。
* **防范跨站请求伪造（CSRF）**: 应该为所有敏感操作（如更改密码、删除数据等）添加CSRF令牌，以防止未经授权的人员执行这些操作。
* **防范安全漏洞**: 应该使用安全性高的算法和协议（如HTTPS、TLS）来保护敏感数据的传输和存储，并定期对系统进行漏洞扫描和安全性审计。

## 14. 事务控制规范

* 一般推荐使用编程式事务，而不是一个注解 @Transactional的声明式事务。因为 @Transactional有很多场景，可能导致事务不生效。
* 事务范围要明确，数据库操作必须在事务作用范围内，**如果是非数据库操作，尽量不要包含在事务内。**
* 不要在事务内进行远程调用（可能导致数据不一致，比如本地成功了，但是远程方法失败了，这时候需要用分布式事务解决方案）
* 事务中避免处理太多数据，一些查询相关的操作，尽量放到事务之外（避免大事务问题）

## 15. 幂等处理规范

什么是幂等？

计算机科学中，幂等表示一次和多次请求某一个资源应该具有同样的副作用，或者说，多次请求所产生的影响与一次请求执行的影响效果相同。

要关注接口是否考虑幂等。**比如开户接口，多次请求过来的时候，需要先查一下该客户是否已经开过户，如果已经开户成功，直接返回开户成功的报文。如果还没开户，就先开户，再返回开户成功的报文。这就是幂等处理。**

**一般情况有这几种幂等处理方案：**

* select+insert+主键/唯一索引冲突
* 直接insert + 主键/唯一索引冲突
* 状态机幂等
* 抽取防重表
* token令牌
* 悲观锁
* 乐观锁
* 分布式锁

幂等要求有个**唯一标记**，比如数据库防重表的一个**业务唯一键**。同时强调多次请求和一次请求所产生影响是一样的。

## 16. 中间件注意事项 （数据库，redis）

如果用数据库、Redis、RocketMq等的中间件时，我们需要关注这些中间件的一些注意事项哈。

**比如数据库**：

* 关注数据库连接池参数设置、超时参数设置是否合理
* 避免循环调用数据库操作
* 如果不分页，查询SQL时，如果条数不明确，是否加了limit限制限制
* 数据库的返回是否判空处理
* 数据库慢SQL是否有监控
* 表结构更新是否做兼容，存量表数据是否涉及兼容问题考虑
* 索引添加是否合理
* 是否连表过多等等

比如Redis:

* Redis的key使用是否规范
* Redis 异常捕获以及处理逻辑是否合理
* Redis连接池、超时参数设置是否合理
* Redis 是否使用了有坑的那些命令，如hgetall、smember
* 是否可能会存在缓存穿透、缓存雪奔、缓存击穿等问题。

## 17. 注意代码坏味道问题

理解几个常见的代码坏味道，需要关注一些哈：

* 大量重复代码（抽公用方法，设计模式）
* 方法参数过多（可封装成一个DTO对象）
* 方法过长（抽小函数）
* 判断条件太多（优化if...else）
* 不处理没用的代码（没用的import）
* 避免过度设计

## 18. 远程调用

* **不要把超时当作失败处理**: 远程调用可能会失败，比如**网络中断、超时**等等。开发者需要注意远程调用返回的错误码，除非是明确的失败，**如果仅仅是超时等问题，不能当作失败处理**！而是应该发起查询，确认是否成功，再做处理。
* 异常处理：远程调用可能会抛出异常，例如由于服务端错误或请求格式不正确等。因此，开发人员需要确保能够捕获和处理这些异常，以避免系统崩溃或数据丢失。
* **网络安全**：由于远程调用涉及网络通信，因此开发人员需要考虑网络安全的问题，例如**数据加密、认证、访问控制**等。尽可能使用安全的协议，例如HTTPS 或 SSL/TLS。
* **服务质量**：远程调用可能会影响系统的性能和可用性。因此，开发人员需要确保服务的质量，例如**避免过度使用远程调用、优化数据传输、实现负载均衡等。**
* **版本兼容**：由于远程调用涉及不同的进程或计算机之间的通信，因此开发人员需要注意服务端和客户端之间的版本兼容性。尽可能使用相同的接口和数据格式，避免出现不兼容的情况。
* **尽量避免for循环远程调用**: 尽量避免for循环远程调用，而应该考虑实现了批量功能的接口。