后端日志添加说明

新品孵化团队

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **修订内容** | **修订人** | **修订时间** |
| V0.01 | 添加日志规范及注意事项 | 许远航 | 20191031 |

目录

[1、记录日志的目的 2](#_Toc23499303)

[2、日志的记录内容 2](#_Toc23499304)

[3、日志记录规范 3](#_Toc23499305)

[4、注意事项 4](#_Toc23499306)

# 1、记录日志的目的

* 1. 调试开发

目的是开发期调试程序使用，这种日志量比较大，且没有什么实质性的意义，**只应该出现在开发期，而不应该在项目上线之后输出**。

* 1. 用户行为日志

这种类型的日志，记录用户的操作行为，用于大数据分析，比如监控、风控、推荐等等。这种日志，一般是给其他团队分析使用，而且可能是多个团队，因此一般会有一定的格式要求，开发者应该按照这个格式来记录，便于其他团队的使用。当然，要记录哪些行为、操作，一般也是约定好的，因此，开发者主要是执行的角色。

* 1. **程序运行日志**

记录程序的运行状况，特别是非预期的行为、异常情况，这种日志，主要是给开发、维护人员使用。什么时候记录，记录什么内容，完全取决于开发人员，开发者具有高度自主性。本文讨论的主要也是指这种类型的日志，因为作为一个服务端开发、运维人员，程序运行日志往往是解决线上问题的救命稻草。

* 1. 记录系统或者机器的状态

比如网络请求、系统CPU、内存、IO使用情况等等，这种日志主要是给运维人员使用，生成各种更直观的展现形式，在系统出问题的时候报警。

# 2、日志的记录内容

确定记录内容最要重要的原则：日志加上哪些信息能定位问题发生的原因

1. when 事件发生时间

时间指的是日志记录的事件的发生时间，而不是日志被最终输出的时间。

1. where 事件产生的地点

记录地点的意义在于能够让自己或者其他程序员一眼就看出这条日志是在哪里产生的

1. how 事件的重要性

重要性即日志的级别：TRACE, DEBUG, INFO, WARN, ERROR,FATAL

1. what 事件的内容

主体内容应该简明扼要的描述发生的什么事情。要求可以通过日志本身，而不是重新阅读产生日志的代码。

正例：logger.warn('user\_login failed due to unvalid\_username')；

反例：logger.warn('Error Happened')；

1. who 事件产生者

事件产生者的唯一标识（identity），用于区分同样的事件。

正例：logger.warn('user\_login failed due to password, username = [{}]', username)；

反例：logger.warn('user\_login failed due to password')；

# 3、日志记录规范

1. 【强制】应用中不可直接使用日志系统（Log4j、Logback）中的API，而应依赖使用日志框架 SLF4J中的API，使用门面模式的日志框架，有利于维护和各个类的日志处理方式统一。

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

private static final Logger logger = LoggerFactory.getLogger(Test.class);//将Test.class更换为需要记录日志的类名

1. 【强制】在日志输出时，字符串变量之间的拼接使用占位符的方式。

说明：因为String字符串的拼接会使用StringBuilder的append()方式，有一定的性能损耗。使用占位符仅是替换动作，可以有效提升性能。【推荐】使用[{}]进行变量隔离，可读性更好

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

1. 【强制】异常信息应该包括两类信息：案发现场信息和异常堆栈信息。如果不处理，那么通过关键字throws往上抛出。

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

1. 【推荐】对于trace/debug级别的日志输出，是具体情况进行日志级别的开关判断。

debug 参数复杂，响应的时间长，考虑并发的情况下，建议使用if(logger.isDebugEnabled()) 先判断；

debug 参数简单，响应时间短，建议直接使用 logger.debug()；

正例：

// 如果判断为真，那么可以输出trace和debug级别的日志

if (logger.isDebugEnabled()) {

logger.debug("这是个复杂的参数调用:"+ costLongLongTime());

}

# 4、注意事项

1. 记录日志的过程不应该引入新的问题，不能抛出新的异常

例如：log.debug("Processing request with id: {}", request.getId());

该日志可能会因为request为空导致空指针异常；

例如：log.debug("Returning users: {}", users);

该日志记录从数据库读取的users集合，可能会因为记录数据过多导致内存溢出、日志存储空间用完了等问题

1. 记录认为一定不会发生的情况

当认为某种情况一定不会发生，按照墨菲定律，它还是很可能会发生，那么就应该在万一发生的情况记录log，Error（Fatal）级别的log。尤其是在异步，并发的情况下，程序员任务的不可能都会成为可能。

1. 不同级别的使用

**ERROR:**

基本概念

影响到程序正常运行、当前请求正常运行的异常情况:

1. 打开配置文件失败
2. 所有第三方对接的异常(包括第三方返回错误码)
3. 所有影响功能使用的异常，包括:SQLException和除了业务异常之外的所有异常(RuntimeException和Exception)

如果有Throwable信息，需要记录完成的堆栈信息:

log.error("获取用户[{}]的用户信息时出错",userName,e);

说明

如果进行了抛出异常操作，请不要记录error日志，由最终处理方进行处理。

反例(不要这么做):

try{

}catch(Exception ex){

String errorMessage=String.format("Error while reading information of user [{}]",userName);

logger.error(errorMessage,ex);

throw new UserServiceException(errorMessage,ex);

}

**WARN:**

基本概念

不应该出现但是不影响程序、当前请求正常运行的异常情况:

1. 有容错机制的时候出现的错误情况
2. 找不到配置文件，但是系统能自动创建配置文件

即将接近临界值的时候，例如：

缓存池占用达到警告线

业务异常的记录,比如:

当接口抛出业务异常时，应该记录此异常

**INFO:**

基本概念

系统运行信息

1. Service方法中对于系统/业务状态的变更
2. 主要逻辑中的分步骤

外部接口部分

1. 客户端请求参数(REST/WS)
2. 调用第三方时的调用参数和调用结果

说明

1. 并不是所有的service都进行出入口打点记录,单一、简单service是没有意义的(job除外,job需要记录开始和结束,)。
2. 对于复杂的业务逻辑，需要进行日志打点，以及埋点记录，比如电商系统中的下订单逻辑，以及OrderAction操作(业务状态变更)。
3. 对于整个系统的提供出的接口(REST/WS)，使用info记录入参
4. 调用其他第三方服务时，所有的出参和入参是必须要记录的(因为你很难追溯第三方模块发生的问题)