日志在应用程序中是非常非常重要的，好的日志信息能有助于我们在程序出现 BUG 时能快速进行定位，并能找出其中的原因。

但是，很多介绍 AOP 的地方都采用日志来作为介绍，实际上日志要采用切面的话是极其不科学的！对于日志来说，只是在方法开始、结束、异常时输出一些什么，那是绝对不够的，这样的日志对于日志分析没有任何意义。如果在方法的开始和结束整个日志，那方法中呢？如果方法中没有日志的话，那就完全失去了日志的意义！如果应用出现问题要查找由什么原因造成的，也没有什么作用。这样的日志还不如不用！

希望藉以本文能让应用程序的开发人员能更加重视日志，能在应用中输出有意义的日志。

**日志基本格式**

日志输出主要在文件中，应包括以下内容：

* 时间
* 日志级别主要使用
* 调用链标识（可选）
* 线程名称
* 日志记录器名称
* 日志内容
* 异常堆栈（不一定有）

11:44:44.827 WARN [93ef3E0120160803114444-1.2] [main] [ClassPathXmlApplicationContext] Exception encountered during context initialization - cancelling refresh attempt

**日志时间**

作为日志产生的日期和时间，这个数据非常重要，一般精确到毫秒。由于一般按天滚动日志文件，日期不需要放在这个时间中，使用 HH:mm:ss.SSS 格式即可。

**日志级别**

日志级别主要使用 DEBUG、INFO、WARN、ERROR。

**DEBUG**

DEUBG 级别的主要输出调试性质的内容，该级别日志主要用于在开发、测试阶段输出。该级别的日志应尽可能地详尽，便于在开发、测试阶段出现问题或者异常时，对其进行分析。

**INFO**

INFO 级别的主要输出提示性质的内容，该级别日志主要用于生产环境的日志输出。该级别或更高级别的日志不要出现在循环中，可以在循环开始或者结束后输出循环的次数，以及一些其他重要的数据。

* 应用启动时所加载的配置参数值（比如：连接参数、线程池参数、超时时间等，以及一些与环境相关的配置，或者是整个配置参数）
* 一些重要的依赖注入对象的类名
* 方法（服务方法）的输入参数值、返回值，由于一些方法入参的值非常多，只在入口处输出一次就可以了，在服务方法内部或者调用非服务方法时就不需要再输出了
* 方法中重要的部分，比如：从数据库中所获取较为重要的数据，以及调用第三方接口的输入参数值和接口返回值

INFO 级别日志原则是在生产环境中，通过 INFO 和更高级别的日志，可以了解系统的运行状况，以及出现问题或者异常时，能快速地对问题进行定位，还原当时调用的上下文数据，能重现问题。

建议在项目完成后，在测试环境将日志级别调成 INFO，然后通过 INFO 级别的信息看看是否能了解这个应用的运用情况，如果出现问题后是否这些日志能否提供有用的排查问题的信息。

**WARN**

WARN 级别的主要输出警告性质的内容，这些内容是可以预知且是有规划的，比如，某个方法入参为空或者该参数的值不满足运行该方法的条件时。在 WARN 级别的时应输出较为详尽的信息，以便于事后对日志进行分析，不要直接写成：

不好的日志

log.warn( "name is null" );

除了输出警告的原因之外，还需要将其他参数内容都输出，以便于有更多的信息供为日志分析的参考。

推荐的日志

log.warn( "[{}] name is null, ignore the method, arg0: {}, arg1: {}" , methodName , arg0 , arg1 );

**ERROR**

ERROR 级别主要针对于一些不可预知的信息，诸如：错误、异常等，比如，在 catch 块中抓获的网络通信、数据库连接等异常，若异常对系统的整个流程影响不大，可以使用 WARN 级别日志输出。在输出 ERROR 级别的日志时，尽量多地输出方法入参数、方法执行过程中产生的对象等数据，在带有错误、异常对象的数据时，需要将该对象一并输出：

推荐的日志

log.error( "Invoking com.service.UserService cause error, username: {}" , username , e );

不要写成（下面这种会将 e 作为日志内容参数中的一个，效果与使用 e.toString() 一致，不会输出异常堆栈）：

不好的日志

log.error( "Invoking com.service.UserService cause error, username: {}, e: {}" , username , e );

不要在日志中输出下面这样的日志，在异常堆栈 e 中本身就会输出 e.getMessage 的内容，没必要在日志行中输出一遍，这样的日志对于问题的追踪毫无意义！

不好的日志

log.error( e.getMessage() , e );

**调用链标识**

在分布式应用中，用户的一个请求会调用若干个服务完成，这些服务可能还是嵌套调用的，因此完成一个请求的日志并不在一个应用的日志文件，而是分散在不同服务器上不同应用节点的日志文件中。该标识是为了串联一个请求在整个系统中的调用日志。

调用链标识格式：

* 唯一字符串（trace ID）
* 调用层级（span ID）

调用链标识作为可选项，无该数据时只输出 [] 即可。

**线程名称**

输出该日志的线程名称，一般在一个应用中一个同步请求由同一线程完成，输出线程名称可以在各个请求产生的日志中进行分类，便于分清当前请求上下文的日志。

**日志记录器名称**

日志记录器名称一般使用类名，日志文件中可以输出简单的类名即可，看实际情况是否需要使用包名。主要用于看到日志后到哪个类中去找这个日志输出，便于定位问题所在。

**日志内容**

**注意事项**

*禁用 System.out.println*

src/main 的代码中严禁使用 System.out.println 进行输出，因为生产环境一般不会将标准输出和错误输出重定向到文件中去，如果代码中使用该方式输出日志，可能会导致该输出丢失。

*变参替换日志拼接*

使用 slf4j 的 Logger 进行处理，使用其变参功能进行日志输出，不要在日志中进行字符串的拼接，比如：

推荐的日志

log.debug( "Load No.{} object, {}" , i , object );

不要写成 log.debug ( "Load No." + i + " object, " + object ); 这是因为将日志级别调至 INFO 或以上级别时，这样会增加无畏的字符串拼接。

*实现 toString()*

需要输出日志的对象，应在其类中实现快速的 toString 方法，以便于在日志输出时仅输出这个对象类名和 hashCode。该 toString 方法应该处理类中所有的字段。toString 方法可以通过 IDE 的自动功能 toString 功能生成。toString 方法建议不要通过反射或者一些 toString 工具类生成，也不要直接使用 JSON 序列化工具转为 JSON 字符串，这两者均使用反射进行处理的，仅为了输出日志较为影响应用的性能。

*预防空指针*

不要在日志中调用对象的方法获取值，除非确保该对象肯定不为 null，否则很有可能会因为日志的问题而导致应用产生空指针异常。

不好的日志

log.debug( "Load student(id={}), name: {}" , id , student.getName() );

可以改为（当 student 为 null 时，这样也不会产生空指针异常）：

推荐的日志

log.debug( "Load student(id={}), student: {}" , id , student );

对于一些一定需要进行拼接字符串，或者需要耗费时间、浪费内存才能产生的日志内容作为日志输出时，应使用 log.isXxxxxEnable() 进行判断后再进行拼接处理，比如：

推荐的代码

|  |
| --- |
| if ( log.isDebugEnable() ) {      StringBuilder builder = new StringBuilder();      for ( Student student : students ) {          builder.append( "student: " ).append( student );      }      builder.append( "value: " ).append( JSON.toJSONString(object) );      log.debug( "debug log example, detail: {}" , builder );  } |

**信息安全**

切记不要 log 密码及个人信息相关的内容！为了便于进行问题定位，以下是涉及敏感信息日志输出时最为宽松（明文显示的数据只能更少，不能更多）的要求：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **要求** | **示例** | **说明** |
| 密码 | 不输出 | \*\*\*\*\*\* | 登录密码、支付密码等各种类型的密码 |
| 信用卡 CVV2 | 不输出 | \*\*\* |  |
| 信用卡有效期 | 不输出 | \*\*\*\* |  |
| 验证码 | 不输出 | \*\*\*\*\*\* | 图形验证码、短信验证码、邮件验证码等 |
| 密钥、盐 | 不输出 | \*\*\*\*\*\* | 用于加解密算法的密钥，消息摘要的盐，以及数字签名及签名验证算法所使用的公私钥对等 |
| 会话 ID  设备指纹 (ID)  指纹 token  密文数据 | 前 5 后 5 | 7SuA8\*\*\*TtslB | 主要有以下类型：  1. 应用的会话标识，比如：Web、APP、H5 等用于识别会话状态信息的标识  2. APP 标识设备的设备指纹或者设备 ID  3. APP 用于指纹验证的 token  4. 密文数据指的是加密后的数据  被掩码的字符无论多少位都输出 3 个 \* |
| 银行卡卡号 | 前 6 后 4 | 622666\*\*\*\*\*\*0831 | 银行卡卡号最多 19 位数字 |
| 手机号 | 前 3 后 4 | 137\*\*\*\*9574 | 定长 11 位数字 |
| 身份证号 | 前 1 后 1 | 3\*\*\*\*\*\*X | 定长 18 位 |
| 姓名 | 隐姓 | \*世仁 | 将姓氏隐藏 |
| IP 地址 | 前 1 后 1 | 10.\*.\*.27 | 隐藏 IP 地址的第 2、第 3 段 |
| 邮箱地址 | 前 1 后 1 | w\*\*3@gmail.com | 仅对 @ 之前的邮箱名称进行掩码，掩码部分不管多少位均输出 \*\*\* |
| 地址 | 隐号码 | 上海市西藏北路 \*\*\* 号 \*\*\* 楼 \*\*\* 室 |  |

上述仅列取出部分数据的显示要求，其他的显示原则为通过掩码后的数据无法得知原始数据。

实现了如上掩码的工具类，参考：https://github.com/frankiegao123/mask-utils

**异常堆栈**

异常堆栈一般会出现在 ERROR 或者 WARN 级别的日志中，异常堆栈含有方法调用链的系统，以及异常产生的根源。异常堆栈的日志属于上一行日志的，在日志收集时需要将其划至上一行中。

**日志文件**

日志文件放置于固定的目录中，按照一定的模板进行命名，推荐的日志文件名称：

* 当前正在写入的日志文件名：<应用名>[-<功能名>].log
* 已经滚入历史的日志文件名：<应用名>[-<功能名>].log.<yyyy-MM-dd>

**日志配置**

**输出**

根据不同的环境配置不同的日志输出方式：

* 本地调试可以将日志输出到控制台上
* 测试环境或者生产环境输出到文件中，每天产生一个文件，如果日志量庞大可以每个小时产生一个日志文件
* 生产环境中的文件输出，可以考虑使用异步文件输出，该种方式日志并不会马上刷新到文件中去，会产生日志延时，在停止应用时可能会导致一些还在内存中的日志未能及时刷新到文件中去而产生丢失，如果对于应用的要求并不是非常高的话，可暂不考虑异步日志

logback 日志工具可以在日志文件滚动后将前一文件进行压缩，以减少磁盘空间占用，若使用 logback 对于日志量庞大的应用建议开启该功能。