优秀日志实践准则

刘持 zhisheng 昨天

> 程序员的日常离不开日志，日志就好比是程序员的私人秘书，负责运行周期一切trace工作。优秀的日志实践能极大帮助程序员快速定位问题，减少在线错误报警。本文从日志书写时各方面来做阐述，依据日志推荐的日志等级，做相应优秀日志实践的推荐。

重新认识日志

1.日志级别概述

- **ERROR**

  ERROR是最高级别错误，反映系统发生了非常严重的故障，无法自动恢复到正常态工作，需要人工介入处理。系统需要将错误相关痕迹以及错误细节记录ERROR日志中，方便后续人工回溯解决。

- **WARN**

  WARN是低级别异常日志，反映系统在业务处理时触发了异常流程，但系统可恢复到正常态，下一次业务可以正常执行。但WARN级别问题需要开发人员给予足够关注，往往表示有参数校验问题或者程序逻辑缺陷，当功能逻辑走入异常逻辑时，应该考虑记录WARN日志。

- **INFO**

  INFO日志主要记录系统关键信息，旨在保留系统正常工作期间关键运行指标，开发人员可以将初始化系统配置、业务状态变化信息，或者用户业务流程中的核心处理记录到INFO日志中，方便日常运维工作以及错误回溯时上下文场景复现。

- **DEBUG**

  DEBUG日志是INFO日志的好帮手，开发人员可以将各类详细信息记录到DEBUG里，起到调试的作用，包括参数信息，调试细节信息，返回值信息等等。其他等级不方便显示的信息都可以通过DEBUG日志来记录。

2.记录日志的时机

在看线上日志的时候，我们可曾陷入到日志泥潭？该出现的日志没有，无用的日志一大堆，或者需要的信息分散在各个角落，特别是遇到紧急的在线bug时，有效的日志被大量无意义的日志信息淹没，焦急且无奈地浪费大量精力查询日志。那什么是记录日志的合适时机呢？

总结几个需要写日志的点：

编程语言提示异常：如今各类主流的编程语言都包括异常机制，业务相关的流行框架有完整的异常模块。这类捕获的异常是系统告知开发人员需要加以关注的，是质量非常高的报错。应当适当记录日志，根据实际结合业务的情况使用warn或者error级别。

业务流程预期不符：除开平台以及编程语言异常之外，项目代码中结果与期望不符时也是日志场景之一，简单来说所有流程分支都可以加入考虑。取决于开发人员判断能否容忍情形发生。常见的合适场景包括外部参数不正确，数据处理问题导致返回码不在合理范围内等等。

系统核心角色，组件关键动作：系统中核心角色触发的业务动作是需要多加关注的，是衡量系统正常运行的重要指标，建议记录INFO级别日志，比如电商系统用户从登录到下单的整个流程；微服务各服务节点交互；核心数据表增删改；核心组件运行等等，如果日志频度高或者打印量特别大，可以提炼关键点INFO记录，其余酌情考虑DEBUG级别。

系统初始化：系统或者服务的启动参数。核心模块或者组件初始化过程中往往依赖一些关键配置，根据参数不同会提供不一样的服务。务必在这里记录INFO日志，打印出参数以及启动完成态服务表述。

3.警惕日志性能代价

不管是多么优秀的日志工具，在日志输出时总会对性能产生或多或少的影响，为了将影响降低到最低，有以下几个准则需要遵守：

- 根本原则：有必要才记录日志，频繁过量日志对性能是有损耗的，并且这种风险不常在系统正常时出现，系统出现问题时大量ERROR，INFO等问题相关日志有可能产生连锁反应，造成严重的后果。将关键信息保存到日志，同时考虑极端场景日志爆发。
- Logger获取: 根据系统使用的日志框架组合，确定正确的实例获取方式。在log4j的早期版本，一般要求使用static，而在高版本以及后来的slf4j等一些框架封装中，该问题已经得到优化，获取（创建）logger实例的成本已经很低。但对于多例，尤其是需要频繁创建的class，推荐添加static前缀。

- 输出等级校验: 在log4j 1.x版本，对于可以预见的会频繁产生的日志输出，先判断一下(logger.isXXXEnabled()，对于性能有很大提升，在其他外观框架或者log4j 2.x中已经自动实现。
- 输出格式：禁止使用字符串拼接，使用参数方式。
- 样式配置: 布局配置输出的信息也会影响到性能，需要根据logger的具体使用场景来选择输出合适信息。

  上述几点可以看出，核心都是减少日志量 ，前两点偏向设计，后四点偏向日志框架及习惯，并且这四点目前一些框架组合已经能帮开发人员减少不少工作，比如log4j2.x在实例获取，输出等级判断都有优化。除开减少日志量，还需要注意多线程以及高并发情况下的日志输出。日志输出本身是写磁盘操作，自然会有性能瓶颈。更多属于日志框架选择及优化方面，选择日志框架时除了考虑正常功能使用，务必关注该日志框架影响性能的细节，日志的出发点是帮助处理问题，如果成为隐患就得不偿失了。介绍日志框架选择的文章很多，在此就不做详细介绍了。

优秀的INFO,DEBUG实践

INFO和DEBUG级别日志描述的是系统正常运行时的表征。在监控程序正常执行，处理客服反馈，分析用户行为时起到重要作用。因此优秀的INFO,DEBUG日志能帮助开发人员快速了解运行时的各个细节。

使用场景：

- **线上问题跟踪**

  客服反馈或者线上问题很难解决时，需要更多细节信息，这时会寻求日志的帮助，error，warn，info都可能对解决问题有所帮助。必要时还需要开启debug日志帮忙在线定位问题。
- **场景还原**

  随着数据重要性越来越高，日志的作用不再单单帮助纠错，构建上下文也成为日志大放光彩的地方，还原一个会话，需要在日志中需要加入会话标识的概念，可以是简单的ip或者复杂会话痕迹：常见包括以下两个角度：**模块维度记录**：登录模块，商品详情模块，下单模块，支付模块，派发模块等；**以行为维度记录**：在什么时间，在什么地方，在干什么，结果是什么样；统一加上会话标识以及时间属性即可。这也是微服务日志以及数据分析的基础。

- **监控调优**

  对于系统的监控调优也是日常工作之一，主要通过日志进行信息记录，主要关注以下3个部分的日志：

  请求处理时间：
  模块操作处理的时间
  核心组件操作的时间（db，缓存，磁盘io等等）

日志内容

日志应当提供如下内容：

- **时间**，包含时区信息和毫秒，这个工作往往日志框架足以支持。核心属性之一。
- **日志级别**，例如 debug、info 以及warn、error
- **会话标识**，能知道是哪个客户端或者是哪个用户触发，登陆账号，seesion信息等
- **功能标识**，功能标识的意义在于方便日志搜索，跟踪指定功能的完整轨迹，是INFO，DEBUG日志的常见技巧。跟logger分类同一道理，更细分功能标识则是方法标识，更多使用在DEBUG做在线调试使用。
- **精炼的内容**，内容永远是日志的核心，结合上述使用场景，简单来说包括场景信息（谁，什么功能等），状态信息(开始，中断，结束)以及重要参数.
- **其他信息**，其他可能的有用信息包括：版本号，线程号等等。

INFO和DEBUG的选择

DEBUG级别比INFO低，包含调试时更详细的了解系统运行状态的东西，比如变量的值等等，都可以输出到DEBUG日志里。 INFO是在线日志默认的输出级别，反馈系统的当前状态给最终用户看的。输出的信息，应该对最终用户具有实际意义的。从功能角度上说，Info输出的信息可以看作是软件产品的一部分，所以需要谨慎对待，不可随便输出。尝试记录INFO日志时不妨在头脑中模拟线上运行，如果这条日志会被频繁打印或者大部分时间对于纠错起不到作用，就应当考虑下调为DEBUG级别。

注意事项

由于info及debug日志打印量远大于ERROR，出于前文日志性能的考虑，如果代码为核心代码，执行频率非常高，务必推敲日志设计是否合理，是否需要下调为DEBUG级别日志。

注意日志的可读性，不妨在写完代码review这条日志是否通顺，能否提供真正有意义的信息。
日志输出是多线程公用的，如果有另外一个线程正在输出日志，上面的记录就会被打断，最终显示输出和预想的就会不一致。
线上代码禁止出现各类print等
不要认为记录DEBUG就等同于在线忽略。很多情况因为担心线上出问题时缺少有用信息而对于等级犹豫不决，担心记录为DEBUG对于在线调试起不到作用。你需要的不是研究日志级别，而是实时日志等级调整工具。尝试记录为DEBUG，线上有必要使用DEBUG级别调试时再调整对应logger级别。

优秀的WARN,ERROR实践

使用时机和思路

当方法或者功能处理过程中产生不符合预期结果或者有框架报错时可以考虑使用，常见问题处理方法包括：

- 增加判断处理逻辑，尝试本地解决
- 抛出异常，交给上层逻辑解决
- 记录日志，报警提醒
- 使用返回码包装错误做返回

几个成长阶段

- 不记录日志，顺其自然，tomcat或者框架来捕获最后的异常
- 方法出入口try catch， e.printstack()， system.out， log混合使用
- 有判断校验减少异常可能性，能合理小范围使用try catch，使用log作为唯一记录方式
- 灵活运用处理办法，合理抛留异常。报警少而精

方法和准则

- 增加逻辑判断吞掉报警永远是最优选择。
- 不在预期范围内的情景，则考虑返回码包装或者抛出异常，需要依情况而定：

返回码的缺点：
不直观，不友好，处处都需要进行显示判断，返回码都有具体含义，但字面不体现，持续维护时代码理解成本高。

异常机制的缺点：

必须在系统内部，多系统交互，前后台交互场景等无法使用
每一个异常分支都可能需要单独一个类作为描述和控制，大量错误类型造成代码量增加。

- 一旦抛出异常，必须catch处理，挑选正确方式：
  打印日志：当前逻辑就能处理掉的，不需要上层再处理的，或者本身就是最上层。
  重抛异常：判断异常当前无法处理，需要继续向上抛出，可以经过异常包装转义。需要注意当前是否为最上层。
- 避免过大的try块，尽量保持一个try块对应一个或多个异常。
- 细化异常的类型，避免不顾细节抛出Excetpion。异常类的作用就是告诉Java编译器我们想要处理的是哪一种异常，然后针对具体的异常类进行不同的处理。
- 函数返回值能表达错误含义，则不应该打印ERROR日志，防止ERROR日志泛滥。错误不一定到边界才能终止，只要返回到能处理它的地方就应当终止。

使用WARN和统计报警

一般来说，WARN级别不会短信报警，ERROR级别则会短信报警甚至电话报警，ERROR级别的日志意味着系统中发生了非常严重的问题，必须有人马上处理，比如数据库不可用，系统的关键业务流程走不下去等等。错误的使用反而带来严重的后果，不区分问题的重要程度，只要有问题就error记录下来，其实这样是非常不负责任的，因为对于成熟的系统，都会有一套完整的报错机制，那这个错误信息什么时候需要发出来，很多都是依据单位时间内ERROR日志的数量来确定的。因此如果我们不分轻重缓急，一律ERROR对待，就会徒增报错的频率，久而久之，我们的救火队员对错误警报就不会那么在意，这个警报也就失去了原始的意义。

WARN代表可恢复的异常，此次失败不影响下次业务的执行，开发人员会苦恼某些场景下几次失败可容忍，频率高的时候需要提醒，记录ERROR的结果是线上时不时出现容忍范围内的报警，这时报警是无意义的。但反之不记录ERROR日志，真正出现问题则不会有实时报警，错过最佳处理时机。

- 请使用WARN级别配合统计报警，统计报警的实现方法有很多种，代码统计，脚本统计，zabbix统计等等，挑选合适的统计方式结合warn级别，建立频次报警机制，做

到适时，适量提醒。

强调ERROR报警

ERROR的报出应该伴随着业务功能受损，即上面提到的系统中发生了非常严重的问题，必须有人马上处理。

ERROR日志目标

给处理者直接准确的信息：error信息形成自身闭环。

问题定位：发生了什么问题，哪些功能受到影响
获取帮助信息：直接帮助信息或帮助信息的存储位置
通过报警知道解决方案或者找何人解决

实用模板

日志模板2选1：

```
1   log.error("[接口名或操作名] [Some Error Msg] happens. [Probably Because]. [Proba
2   log.error("[接口名或操作名] [Some Error Msg] happens. [Probably Because]. [pleas
```

请尽量按上述模板完成，如果实施起来有难度，至少ERROR 日志打印时需要在做一个自我问答，能非常有效的帮助评估这条报警是否有意义：这条报警看到之后我能处理吗？ 应该怎么处理？如果是同事看到能处理或者及时通知联系人呢吗？ 因为你不可能保证随时都处在工作状态，但报警时随时有可能出现的。

注意事项

- 参数检查不是异常

  不要将属于你的检查工作变成ERROR日志。参数检查属于开发人员的工作，而不是全部交给日志。

- 留意业务相关性

  技术相关异常是你需要记录并为此做出反应的。比如内存不足，接口访问超时。涉业务过深的返回需要结合实际情况考虑，比如用户操作错误大多属于业务范畴：用户无

权限参与这次活动返回码。在大多数场景下这个日志作用不大。

- 异常增加结构化参数

  异常代表一类错误，但仅仅是异常类型无法帮助解决问题。保留异常现场参数，保证所有相关的堆栈追踪信息的开始处记录在你的日志中。

- 不要记录日志又重新向外抛出

总结

实际项目中清晰的日志能带来的好处想必不用多说。本文除开介绍常见日志等级以及实践准则之外，更希望DEBUG,WARN两种级别更多，更灵活的利用起来，在项目中形成完整的日志体系。

相关资料

- https://www.slf4j.org/manual.html
- http://logging.apache.org/log4j/2.x/

附录：阿里java编程规范-日志部分：

- **日志规约**

【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。

```
1   import org.slf4j.Logger;
2   import org.slf4j.LoggerFactory;
3   private static final Logger logger = LoggerFactory.getLogger(Abc.class);
```

slf4j 是日志门面框架，其仅提供日志记录的API，而不实现日志记录的功能，slf4j需要通过适配库适配到log4j或logback等日志系统来实现日志的记录。使用slf4j api能够提升代码和应用的可移植性，在使用不同日志系统的应用之间能够做到无缝的适配。同时，使用slf4j api的应用，在切换日志系统时（比如从logback切换到log4j2，不需要代码改造）

【强制】日志文件推荐至少保存15天，因为有些异常具备以“周”为频次发生的特点。

【强制】应用中的扩展日志（如打点、临时监控、访问日志等）命名方式：appName_logType_logName.log。

1、logType:日志类型，推荐分类有stats/desc/monitor/visit等；
2、logName:日志描述。这种命名的好处：通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。
正例：mppserver应用中单独监控时区转换异常，如：mppserver_monitor_timeZoneConvert.log
说明：推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于通过日志对系统进行及时监控。

【强制】对trace/debug/info级别的日志输出，必须使用条件输出形式或者使用占位符的方式。
说明：logger.debug("Processing trade with id: " + id + " symbol: " + symbol); 如果日志级别是warn，上述日志不会打印，但是会执行字符串拼接操作，如果symbol是对象，会执行toString()方法，浪费了系统资源，执行了上述操作，最终日志却没有打印。
正例：（条件）

```
1   if (logger.isDebugEnabled()) {
2     logger.debug("Processing trade with id: " + id + " symbol: " + symbol);
3    }
```

正例：（占位符）

logger.debug("Processing trade with id: {} symbol : {} ", id, symbol);
占位符方式，log4j2/logback支持，log4j1.x是不直接支持的，只能通过slf4j库适配

【强制】避免重复打印日志，浪费磁盘空间，务必在log4j.xml中设置additivity=false。

正例： additivity默认为true，即通过该logger输出的日志会同时输出到root logger，如果还为该logger指定了独立的appender，就会导致这部分日志重复输出

【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：

```
1   logger.error(各类参数或者对象toString + "_" + e.getMessage(), e);
2   记录异常日志的常见错误：
3   logger.error(e);
4   logger.error(e.getMessage());
5   logger.error("上下文"+e.getMessage());
```

上面这几种都是错的！请确保使用的是两个入参的API，如error(String s, Throwable t)

【推荐】谨慎地记录日志。生产环境禁止输出debug日志；有选择地输出info日志；如果使用warn来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘撑爆，并记得及时删除这些观察日志。

说明：大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考：这些日志真的有人看吗？看到这条日志你能做什么？能不能给问题排查带来好处？不要认为日志记录不怎么消耗性能，我见过不少事无巨细式的日志把系统性能严重拖慢的案例

【参考】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 error 级别。