**最佳日志实践**

January 7th, 2014

<http://www.bitstech.net/2014/01/07/log-best-practice/>

**前言**

日志用来记录用户操作、系统运行状态等，是一个系统的重要组成部分。然而由于日志并非系统核心功能，通常情况下并不受团队的重视。在出现问题需要通过日志来定位时，才发现日志还存在很多问题。  
日志记录的好坏直接关系到系统出现问题时定位的速度，同时可以通过对日志的观察和分析，提前发现系统可能的风险，避免线上事故的发生。  
我们在开发和运维NOS（网易对象存储，Netease Object Storage）的过程中，对整个系统的日志进行了分析优化，积累出一些经验，归纳如下。

**相关问题经验整理**

**################################**

**1. 关于日志级别**

我们通常使用的日志库（如log4j等），将日志基本分为以下几类（从低到高）：  
**TRACE**- The TRACE Level designates finer-grained informational events than the DEBUG  
**DEBUG** – The DEBUG Level designates fine-grained informational events that are most useful to debug an application.  
**INFO**- The INFO level designates informational messages that highlight the progress of the application at coarse-grained level.  
**WARN**- The WARN level designates potentially harmful situations.  
**ERROR**- The ERROR level designates error events that might still allow the application to continue running.  
**FATAL**- The FATAL level designates very severe error events that will presumably lead the application to abort.

尽管[log4j官方文档](http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html)对各个日志级别进行了简单定义。然而在实践中，究竟哪些操作需要记入日志，哪种错误应该记为WARN级别，而哪种错误又为ERROR级别，还需要进行进一步讨论。

关于该问题，在StackOverflow上有一个[讨论贴](http://stackoverflow.com/questions/2031163/when-to-use-log-level-warn-vs-error)进行过讨论。

此处对贴子中的一些观点，加上我们在平时运维过程中遇到的相关问题进行归纳：

* 一个项目各个log级别的定义应该是清楚明确的，是每个开发人员所遵循的；
* 即使是TRACE或者DEBUG级别的日志，也应该有一定的规范，要保证除了开发人员自己以外，包括测试人员和运维人员都可以方便地通过日志定位问题；
* 对于日志级别的分类，有以下参考：  
  **FATAL** — 表示需要立即被处理的系统级错误。当该错误发生时，表示服务已经出现了某种程度的不可用，系统管理员需要立即介入。这属于最严重的日志级别，因此**该日志级别必须慎用，如果这种级别的日志经常出现，则该日志也失去了意义。通常情况下，一个进程的生命周期中应该只记录一次FATAL级别的日志，即该进程遇到无法恢复的错误而退出时。当然，如果某个系统的子系统遇到了不可恢复的错误，那该子系统的调用方也可以记入FATAL级别日志**，以便通过日志报警提醒系统管理员修复；  
  **ERROR** — 该级别的错误也需要马上被处理，但是紧急程度要低于FATAL级别。当ERROR错误发生时，已经影响了用户的正常访问。从该意义上来说，**实际上ERROR错误和FATAL错误对用户的影响是相当的。FATAL相当于服务已经挂了，而ERROR相当于好死不如赖活着，然而活着却无法提供正常的服务，只能不断地打印ERROR日志**。特别需要注意的是，**ERROR和FATAL都属于服务器自己的异常，是需要马上得到人工介入并处理的。而对于用户自己操作不当，如请求参数错误等等，是绝对不应该记为ERROR日志的**；  
  **WARN** — 该日志表示系统**可能出现问题，也可能没有，这种情况如网络的波动**等。对于那些目前还不是错误，然而**不及时处理也会变为错误的情况，也可以记为WARN**日志，例如一个存储系统的**磁盘使用量超过阀值，或者系统中某个用户的存储配额快用完等等**。对于WARN级别的日志，虽然不需要系统管理员马上处理，也是需要即使查看并处理的。因此此种级别的日志也不应太多，能不打WARN级别的日志，就尽量不要打；  
  **INFO** — 该种日志记录系统的正常运行状态，例如某个子系统的初始化，某个请求的成功执行等等。通过查看INFO级别的日志，可以很快地对系统中出现的WARN,ERROR,FATAL错误进行定位。INFO日志不宜过多，通常情况下，INFO级别的日志应该不大于TRACE日志的10%；  
  **DEBUG** or **TRACE** — 这两种日志具体的规范应该由项目组自己定义，该级别日志的主要作用是对系统每一步的运行状态进行精确的记录。通过该种日志，可以查看某一个操作每一步的执行过程，可以准确定位是何种操作，何种参数，何种顺序导致了某种错误的发生。可以保证在不重现错误的情况下，也可以通过DEBUG（或TRACE）级别的日志对问题进行诊断。需要注意的是，DEBUG日志也需要规范日志格式，应该保证除了记录日志的开发人员自己外，其他的如运维，测试人员等也可以通过DEBUG（或TRACE）日志来定位问题；

**Rule 1：整个团队（包括运维人员）需要对日志级别有明确的规定，什么日志记入什么级别的日志，什么级别的错误出现要如何处理等**

**######################################**

2. 对记录的日志要进行更新维护

由于DEBUG（或TRACE）级别的日志对于定位问题至关重要，因此该种日志记录是否完备且不冗余、格式是否规范等也需要花费大量精力来优化。此处有以下几个比较好的实践：

* 定义好整个团队记录DEBUG（或TRACE）日志的规范，保证每个开发记录的日志格式统一；
* 整个团队（包括开发，运维和测试）定期对记录的日志内容进行Review；
* 开发做运维，通过在查问题的过程来优化日志记录的方式；
* 运维或测试在日志中发现的问题，都需要及时向开发人员反映；

Rule 2：需要定期对日志内容进行优化更新，目的就是通过日志快速准确的定位问题

######################################

**3. 关于日志分类**

日志从功能来说，可分为诊断日志、统计日志、审计日志。

**诊断日志**， 典型的有：

* 请求入口和出口
* 外部服务调用和返回
* 资源消耗操作: 打开文件等
* 容错行为： 譬如云硬盘的副本修复操作
* 程序异常： 譬如数据库无法连接
* 后台操作：清理程序
* 启动、关闭、配置加载
* 抛出异常时，不记录日志

**统计日志：**

* **用户访问统计**
* **计费日志（如记录用户使用的网络资源或磁盘占用，格式较为严格，便于统计）**

**审计日志：**

* **管理操作**

将不同需求的日志记入到不同的日志文件中，可以方便相关问题（管理平台操作审计，用户操作计费等）的处理。针对每一种需求，需要对日志的格式，日志记录的内容等进行特别的记录。

**Rule 3：要明确不同日志的用途，对日志内容进行分类**

**######################################**

**4. 日志中不要记录无用信息**

在很多应用中，用户都需要通过Fuse方式来挂载使用NOS。

POSIX标准中文件系统接口不允许文件 **/a** 与目录**/a/** 同时存在，而NOS作为对象存储系统，**/a** 和 **/a/** 是不同的对象，是能够同时存在的，一般地，NOS 中我们会规定**/a/** 是目录，**/a** 是文件，目录对象大小为0。

POSIX标准对文件的getattr操作，无论是**/a** 还是 **/a/**，对应的请求都是 **/a**。为了避免遗漏，需分别向 NOS 请求 HeadObject(“**/a**“)和 HeadObject(“**/a/**“)。如果命中**/a**，说明**/a** 是一个文件，不用再请求 getattr(“**/a/**“)。

因此当用户访问 \*/a/b/c.txt\* 时，实际上向NOS发送了以下请求：  
# HeadObject(“/a”)  
# HeadObject(“/a/”)  
# HeadObject(“/a/b”)  
# HeadObject(“/a/b/”)  
# HeadObject(“/a/b/c.txt”)

对于上面的请求，实际上HeadObject(“/a”)和HeadObject(“/a/b”)都会返回NoSuchKey错误，而Fuse正是该错误来判断该文件不存在，而可能是个目录的。

然而对于NOS来说，这将导致产生大量无意义的NoSuchKey日志（整个日志文件的80%都是该错误日志）。这些日志对于开发人员进行日志观察，运维人员定位问题，日志监控等都造成了困难。

**Rule 4: 绝不要打印没有用的日志，防止无用日志淹没重要信息**

**解决办法：**Fuse请求时，在Http头部加入 User-Agent 字段，当NOS发现请求是 Fuse发过来的且为HeadObject操作且为NoSuchKey错误时，则不打印错误日志。

#####################################

**5. 日志记录信息要完整**

**问题描述：**

NOS提供分块上传的接口，用户可以通过以下的调用序列，来实现一次分块上传的流程：

* InitMultiUpload（生成一个UploadID）
* UploadPart
* UploadPart
* ……
* UploadPart
* CompleteMultiUpload（AbortMultiUpload）

之前在某个产品上线初期，由于其开发人员对NOS的熟悉程度不够等原因。出现过如下问题：客户端常常会收到NoSuchUpload的错误。该错误出现的原因是，用户在未调用InitMultiUpload之前，或者在调用了CompleteMultiUpload（AbortMultiUpload）之后再次调用UploadPart。

然而当我们查日志，希望可以看到该UploadPart请求对哪个UploadID进行操作，该UploadID又对应哪些操作时，却发现我们的日志中没有记录UploadPart请求对应的UploadID。

类似的问题还有很多，很多针对特定请求的日志缺失，导致很多问题无法定位。

因此，需要进一步对日志中需要记录哪些内容进行规定，**此处推荐的需要在日志中记录的内容有：**

* 在系统启动或初始化时记录重要的系统初始化参数
* 记录系统运行过程中的所有的错误
* 记录系统运行过程中的所有的警告
* 在持久化数据修改时记录修改前和修改后的值
* 记录系统各主要模块之间的请求和响应（如在NOS中的视频处理模块在接收到请求和发送应答时，或者向客户端发送回调请求时）
* 重要的状态变化（如NOS中对系统白名单的修改等）
* 系统中一些长期执行的任务的执行进度

**而不推荐记录日志的内容有：**

* 函数入口信息 —— 除非该函数入口表示了一个重要事件的开始，或者将该信息记入DEBUG级别日志
* 文件内容或者一大段消息的内容 —— 如果实在需要记录，则可以截取其中一些重要的信息来记入日志
* “良性”错误 —— 有时候虽然出现了错误，然而错误处理的流程可以正确解决这种情况，例如插入数据库时有重复的记录，尽管是个错误，然而错误处理流程可以对这种情况进行处理

**Rule 5：日志信息要准确全面，能做到仅凭日志就可以定位问题**

**解决办法：**整理所有的请求处理流程，针对每一个操作（去重，分块上传……）打印特定的日志。

######################################

**6. 测试的日志**

测试代码（单元测试，接口测试……）的日志同样重要。特别是，当一个测试失败时，可以通过日志很快确定是测试代码有问题，还是系统出现了故障，如果做不到这一点，那就需要优化测试的日志了。

测试日志应该包含以下内容：

* 测试执行的环境
* 测试执行前的初始状态
* 测试的详细步骤
* 测试和系统的交互信息
* 测试期望的返回结果
* 测试实际的返回结果

**Rule 6：要以同样严格的要求对待测试程序的日志**

**#######################################**

**7. 从问题中完善日志**

在线上出现问题的时候，需要尽快发现问题并解决，而同时，需要借此机会好好思考一下当前系统的日志是否合理。需要考虑以下问题：

* 如果定位问题花费了很长时间，那就说明系统日志还存在问题，需要进一步完善和优化
* 需要思考是否可以通过优化日志，来提前预判该问题是否可能发生（如某种资源耗尽而导致的错误，可以对资源的使用情况进行记录）

通过系统出现的问题来优化日志，应该是一项长期的实践，不断地从日志发现系统的问题，不断地从系统异常发现日志的问题。

**Rule 7：日志的优化是一件持续不断需要投入精力的事，需要不断从错误中学习**

**######################################**

**8. 关于RequestID**

**RequestID的生成：**

如今NOS有8台机器，共40个tomcat对外提供服务。通常用户在请求出错的时候，我们都希望用户告诉我们请求的RequestID，以此我们可以确定请求是在哪台机器上进行处理的。

NOS通过以下信息生成一个请求的RequestID：

* 收到请求的时间
* 处理请求的服务器ip地址
* 随机数

因此我们可以通过一个简单的程序从RequestID中得到该请求的处理时间和处理请求的服务器地址，更方便的去查看日志：

**./decode.sh 4b2c009a0a7800000142789f42b8ca96**

**Thu Nov 21 11:06:12 CST 2013**

**10.120.202.150**

**4b2c009a**

**Rule 8：在RequestID中尽量编码更多的信息**

**用RequestID将请求的处理流程关联起来：**

在NOS性能测试中，之前存在的一个问题是，由于在打印错误堆栈的地方，并没有打印请求的RequestID，因此当一个请求出现错误时，很难（日志量太大）将该请求的错误堆栈和具体的请求关联起来。

另一个问题是，NOS后端有视频服务器集群和图片处理服务器集群。因此我们可能会有以下需求：当用户视频截图失败时，用户会告诉我们请求的RequestID，由于NOS并没有将该RequestID转发到后端的图片处理服务器，因此无法利用该信息去查看视频处理服务器上的日志，而需要通过用户请求的URL进行查找。同时，由于我们无法知道该请求是在哪个具体的视频处理的worker上进行，进一步导致查找日志的困难。

还有一个潜在的问题是：如果NOS将所有的日志收集起来（tomcat，图片处理集群，视频处理集群……），我们无法做到通过requestID来查找一个请求的处理流程。

**Rule 9：将一个请求的整个处理流程和唯一的requestID关联起来**

**######################################**

**9. 关于线上机器的日志级别**

**问题描述：**

NOS的DEBUG日志非常详细的记录了请求处理相关信息，然而由于DEBUG日志量太大，因此通常线上只开INFO级别日志。然而INFO级别的日志却有可能导致部分问题无法定位。NOS线上一个请求可能随机地分发到4台机器进行处理，因此如果某一种错误在一段时间内多次出现，它也会在4台服务器上都出现。

因此我们推荐的做法是，选择一台机器开启DEBUG级别的日志，方便定位问题。其实该做法背后的目的是，在线上任何问题的时候，都可以通过日志最快的找到问题的根源。

**Rule 10：让一台机器开启DEBUG日志**

**########################################**

**10. 上线后的日志观察**

随着NOS开始服务越来越多的产品，NOS每次版本升级之后，通过对日志的观察来确定服务是否正常变得至关重要。同时在上线新功能时，来发人员需要通过观察一些特定的日志，来确定新功能是否工作正常。

**举例来说：**

NOS在实现了桶表缓存的功能之后，首先上线一台服务器，并对该功能是否工作正常进行观察。通过将桶缓存的所有操作（如插入，查找，过期删除等）以及桶缓存的状态（如缓存桶数量）都记录在DEBUG级别的日志中。将新上线的机器的日志级别调为DEBUG，并对桶缓存的相关操作是否正确，缓存桶数量等信息进行观察，确认一切正常之后再上线其他机器。

**Rule 11：新上线服务器后一定要对日志进行观察，特别地，开发人员可以通过观察日志来确认新功能是否工作正常**

**#########################################**

**11. 慢操作日志**

NOS在接收到一个请求的时候，会记录请求的接收时间（T1），在请求处理完成待发送的时候，会记录请求发送时间（T2），通常一个请求的日志都记为INFO级别，然而当出现请求处理时间（T2-T1）超过一定时间（如10s）时，会将该日志提升为WARN级别。通过该方法，可以预先发现系统可能存在的一些问题。

同样的慢操作日志还可以用来记录系统一些外部依赖的处理时间，如NOS依赖外部认证服务器来进行认证。我们会记录每个请求的认证时间，如果认证时间超过某个值，也需要将该事件的日志级别进行提升，这样我们可以尽早发现认证服务器是不是需要扩容等问题。

慢日志的时间阀值应该是可以动态调整的，这样在进行系统优化时，可以将该报警时间阀值逐渐调小，不断地对系统进行优化。

**Rule 12：通过日志级别的提升来发现潜在问题**

**######################################**

**12. 日志报警**

错误日志报警：

NOS通过[运维平台|https://m.hz.netease.com/]设置了日志监控报警，周期性的（1分钟，5分钟）对服务器新产生的日志进行监控，如果发现错误数超过某个阀值，则进行报警。这类报警通常不一定是我们服务本身的问题，也有可能是用户使用NOS不当造成的。

此处需要注意的问题是，日志报警相当于grep操作，如果日志量过大，或者匹配规则过多，可能对线上的服务产生影响。因此在设置好日志报警后，需要周期性的关注每次日志扫描的时间，评估日志监控是否对服务产生影响。

**Rule 13：对日志进行监控报警，比客户先发现系统问题**

关键字报警：

NOS为每个用户分配了一定量的存储配额，当用户容量超限时，会限制用户的上传操作。通过在日志中记录关键字，如“Quota Warning”等，可以及时提醒用户进行扩容，避免用户服务中断。

类似的关键字报警还有很多：如对InternalError的数量进行监控，对缓存的桶数量进行监控等等。

**Rule 14：通过日志中的关键字来确定系统的运行状态**

**#######################################**

**13. 关于日志格式**

日志格式一定要统一，不能任由开发人员的喜好来。举例来说，对于NOS视频截图超时的ERROR日志，有以下几种方式打印：

第一种：  
**logger.error(“Gearman timeout exception for request ” + getRequestID() + ” value: ” + value, e);**

第二种：  
**logger.error(“RequestID: ” + getRequestID() + “, Error Message: Gearman timeout exception: ” + e);**

第三种：  
**logger.error(getErrorMessage(getRequestID(), getErrorMessage(), e));**

第一种方式打印日志即是开发人员按照自己的喜好来的，这种方法带来的问题是：

* 系统中日志格式不统一，不利于自动化处理
* 有些日志可能只有开发人员自己才能看懂
* 代码规范性不好

而第三种方式，通过一个函数来规范日志格式，所有开发人员便可以通过该接口实现统一的日志。

**Rule 15：日志格式要统一规范**

**######################################**

**14. 错误日志输出到不同文件**

在性能测试中遇到的另一个问题是，当并发量很大时，可能会有一些请求处理失败（如0.5%），为了对这些错误进行分析，需要去查这些错误请求的日志。而由于这种情况下并发量很大，使得对错误日志的分析变得困难。

这种情况下可以将所有的错误日志同时输出到一个单独的文件之中。

**Rule 16：将错误日志输出到一个单独的文件中进行分析**

**#######################################**

**15. 关于日志文件大小**

日志文件不宜过大，过大的日志文件对于日志监控，问题定位等都会带来不便。因此需要进行日志文件的切分，日志文件的切分可以通过log4j等日志工具来配置，日志文件应该按天来分割，还是按照小时来分割，应该根据日志量来决定，原则就是方便开发或运维人员能快速查找日志。

为了防止日志文件将整个磁盘空间占满，需要定期对日志文件进行删除。例如，在收到磁盘报警时，可以将两个月以前的日志文件删除。此处比较好的实践是：

* 将所有日志文件收集起来，这样即使在记录日志的机器上删除，也可以通过收集的日志对之前的问题进行定位；
* 每天通过定时任务来删除过期日志，如每天在凌晨4点删除60天前的日志

log4j关于日志切分的相关配置，可以参考[这篇文章](http://www.cnblogs.com/baibaluo/archive/2011/06/03/2072091.html)。

**Rule 17：要把日志的大小，如何切分，如何删除等作为规范建立起来**

**############################################**

**经验汇总**

此处对以上总结的所有经验进行汇总：

* 整个团队（包括运维人员）需要对日志级别有明确的规定，什么日志记入什么级别的日志，什么级别的错误出现要如何处理等
* 需要定期对日志内容进行优化更新，目的就是通过日志快速准确的定位问题
* 要明确不同日志的用途，对日志内容进行分类
* 绝不要打印没有用的日志，防止无用日志淹没重要信息
* 日志信息要准确全面，努力做到仅凭日志就可以定位问题
* 要以同样严格的要求对待测试程序的日志
* 日志的优化是一件持续不断需要投入精力的事，需要不断从错误中学习
* 在RequestID中尽量编码更多的信息
* 将一个请求的整个处理流程和唯一的requestID关联起来
* 让一台机器开启DEBUG日志
* 新上线服务器后一定要对日志进行观察，特别地，开发人员可以通过观察日志来确认新功能是否工作正常
* 通过日志级别的提升来发现潜在问题
* 对日志进行监控报警，比客户先发现系统问题
* 通过日志中的关键字来确定系统的运行状态
* 日志格式要统一规范
* 将错误日志输出到一个单独的文件中进行分析
* 要把日志的大小，如何切分，如何删除等作为规范建立起来

**参考文献**

[1] *”Optimal Logging” Anthony Vallone from Google*[*http://googletesting.blogspot.jp/2013/06/optimal-logging.html*](http://googletesting.blogspot.jp/2013/06/optimal-logging.html)