Log规范

一、概要

1.1 什么是日志?

日志，维基百科中对其的定义是**一个或多个由服务器自动创建和维护的日志文件，其中包含其所执行活动的列表。**

一个打印良好的日志文件可为开发人员提供精确的系统记录，可辅助开发人员定位到系统错误发生的详情及根源。在 Java 应用程序中，通常使用日志文件来记录应用程序运行过程中的重要逻辑参数及异常错误，辅之日志采集系统(ELK、DTM)构建系统监控体系。

1.2 为什么要记录日志?

上文中提到日志可以提供精准的系统记录方便根因分析，那为什么要记录日志，记录日志有哪些作用呢?

- **打印调试：**用日志来记录变量或者某一段逻辑，记录程序运行的流程，即程序运行了哪些代码，方便排查逻辑问题。
- **问题定位：**程序出异常或者出故障时快速的定位问题，方便后期解决问题。因为线上生产环境无法 debug，在测试环境去模拟一套生产环境费时费力。所以依靠日志记录的信息定位问题，这点非常重要。
- **监控告警 & 用户行为审计：**格式化后日志可以通过相关监控系统(AntMonitor)配置多维度的监控视图，让我们可以掌握系统运行情况或者记录用户的操作行为并对日志采集分析，用于建设业务大盘使用。

1.3 什么时候记录日志?

上文说了日志的重要性，那么什么时候需要记录日志。

- **代码初始化时或进入逻辑入口时：**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。
- **编程语言提示异常：**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：**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/业务核心逻辑的关键动作：**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。
- **第三方服务远程调用：**微服务架构体系中有一个重要的点就是第三方永远不可信，对于第三方服务远程调用建议打印请求和响应的参数，方便在和各个终端定位问题，不会因为第三方服务日志的缺失变得手足无措。

二、基本规范

2.1 日志记录原则

- **隔离性：**日志输出不能影响系统正常运行；
- **安全性：**日志打印本身不能存在逻辑异常或漏洞，导致产生安全问题；
- **数据安全：**不允许输出机密、敏感信息，如用户联系方式、身份证号码、token 等；
- **可监控分析：**日志可以提供给监控进行监控，分析系统进行分析；
- **可定位排查：**日志信息输出需有意义，需具有可读性，可供日常开发同学排查线上问题。

2.2 日志等级设置规范

在我们日常开发中有四种比较常见的日志打印等级，不同的等级适合在不同的时机下打印日志。

主要使用的有以下四个等级：

- **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，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，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。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**

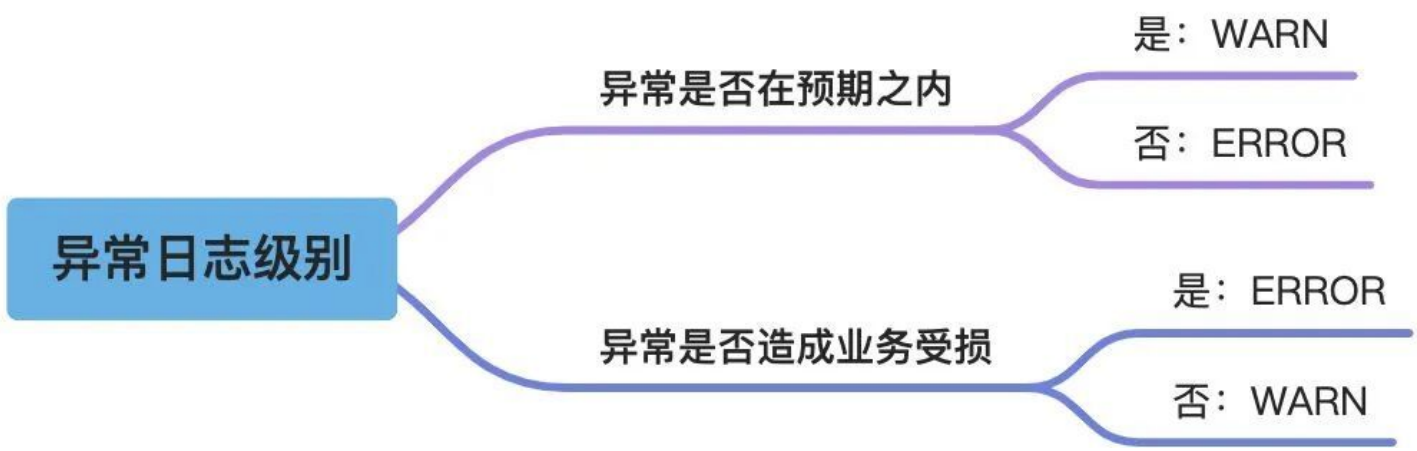
WARN 级别的主要输出警告性质的内容，这些内容是可以预知且是有规划的，比如，某个方法入参为空或者该参数的值不满足运行该方法的条件时。在 WARN 级别的时应输出较为详尽的信息，以便于事后对日志进行分析。

- **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。

如何选择 WARN/ERROR

当方法或者功能出现非正常逻辑执行情况时，需要打印 WARN 或者 ERROR 级别日志，那如何区分出现异常时打印 WARN 级别还是 ERROR 级别呢，我们可以从以下两个方面进行分析：



常见的 WARN 级别异常

- 用户输入参数错误
- 非核心组件初始化失败
- 后端任务处理最终失败（如果有重试且重试成功，就不需要 WARN）
- 数据插入幂等

常见的 ERROR 级别异常

- 程序启动失败
- 核心组件初始化失败
- 连不上数据库
- 核心业务访问依赖的外部系统持续失败

- OOM

不要滥用 ERROR 级别日志。一般来说在配置了告警的系统中，WARN 级别一般不会告警，ERROR 级别则会设置监控告警甚至电话报警，ERROR 级别日志的出现意味着系统中发生了非常严重的问题，必须有人立即处理。

错误的使用 ERROR 级别日志，不区分问题的重要程度，只要是问题就采用 ERROR 级别日志，这是极其不负责任的表现，因为大部分系统中的告警配置都是根据单位时间内 ERROR 级别日志出现的数量来定的，随意打 ERROR 日志将会造成极大的告警噪音，造成重要问题遗漏。

2.3 常见日志格式

摘要日志

摘要日志是格式化的标准日志文件，可用于监控系统进行监控配置和离线日志分析的日志，通常系统对外提供的服务以及集成的第三方服务都需要打印对应的服务摘要日志，摘要日志格式一般需包含以下几类关键信息：

- 调用时间
- 日志链路 id(traceId、rpcId)
- 线程名
- 接口名
- 方法名
- 调用耗时
- 调用是否成功(Y/N)
- 错误码
- 系统上下文信息(调用系统名、调用系统 ip、调用时间戳、是否压测(Y/N))

Code block

```
2022-12-12 06:05:05,129 [0b260533154071424510164O2xxxxx 0.3 - /// - ] INFO [SofaBizProcessor-4-thread-333] - [(interfaceName,methodName,1ms,Y,SUCCESS)(appName,ip地址,时间戳,Y)
```

详细日志

详细日志是用于补充摘要日志中的一些业务参数的日志文件，用于问题排查。详细日志一般包含以下几类信息：

- 调用时间
- 日志链路 id(traceId、rpcId)
- 线程名

- 接口名
- 方法名
- 调用耗时
- 调用是否成功(Y/N)
- 系统上下文信息(调用系统名、调用系统 ip、调用时间戳、是否压测(Y/N))
- 请求入参
- 请求出参

Code block

```
2022-12-12 06:05:05,129 [0b260533154071424510164O2xxxxx 0.3 - /// - ] INFO [SofaBizProcessor-4-thread-333] - [(interfaceName,methodName,1ms,Y,SUCCESS)(appName,ip地址,时间戳,Y)(参数1,参数2)(xxxx)
```

业务执行日志

业务执行日志就是系统执行过程中输出的日志，一般没有特定格式，是开发人员用于跟踪代码执行逻辑而打印的日志，个人看来在摘要日志、详细日志、错误日志齐全的情况下，需要打印系统执行日志的地方比较少。如果一定要打印业务执行日志，需要关注以下几个点：

这个日志是否一定要打印？如果不打印是否会影响后续问题排查，如果打印这个日志后续输出频率是否会太高，造成线上日志打印过多。

日志格式是否辨识度高？如果后续对该条日志进行监控或清洗，是否存在无法与其他日志区分或者每次打印的日志格式都不一致的问题？

输出当前执行的关键步骤和描述，明确的表述出打印该条日志的作用，方便后续维护人员阅读。

日志中需包含明确的打印意义，当前执行步骤的关键参数。

建议格式：[日志场景][日志含义]带业务参数的具体信息

Code block

```
[scene_bind_feature][feature_exists]功能已经存在[tagSource='MIF_TAG',tagValue='123']
```

三、日志最佳实践

3.1 强制

1. 打印日志的代码不允许失败，阻断流程！

一定要确保不会因为日志打印语句抛出异常造成业务流程中断，如下图所示，shop 为 null 的会导致抛出 NPE。

```
Code block
1   public void doSth(){
```

2. 禁止使用 System.out.println()输出日志

反例：

```
Code block
1   public void doSth(){
```

分析：

- 通过分析 System.out.println 源码可知，System.out.println 是一个同步方法，在高并发的情况下，大量执行 println 方法会严重影响性能。

```
Code block
1   public void println(String x) {
```

- 不能实现日志按等级输出。具体来说就是不能和日志框架一样，有 debug，info，error 等级别的控制。
- System.out、System.error 打印的日志都并没有打印在日志文件中，而是直接打印在终端，无法对日志进行收集。

正例：

在日常开发或者调试的过程中，尽量使用标准日志记录系统 log4j2 或者 logback(但不要直接使用其中的 API)，异步的进行日志统一收集。

```
Code block
1   public void doSth(){
```

3. 禁止直接使用日志系统(Log4j、Logback)中的 API

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、JCL--Jakarta Commons Logging)中的 API。

分析：

直接使用 Log4j 或者 Logback 中的 API 会导致系统代码实现强耦合日志系统，后续需要切换日志实现时会产生比较大的改造成本，统一使用 SLF4J 或者 JCL 等日志框架的 API，其是使用门面模式的日志框架，可以做到解耦具体日志实现的作用，有利于后续维护和保证各个类的日志处理方式统一。

正例：

Code block

```
// 使用 SLF4J:
```

4. 声明日志工具对象 Logger 应声明为 private static final

分析：

- 声明为 private 防止 logger 对象被其他类非法使用。
- 声明为 static 是为了防止重复 new 出 logger 对象；防止 logger 被序列化，导致出现安全风险；处于资源考虑，logger 的构造方法参数是 Class，决定了 logger 是根据类的结构来进行区分日志，所以一个类只要一个 logger，故 static。
- 声明为 final 是因为在类的生命周期无需变更 logger，避免程序运行期对 logger 进行修改。

正例：

Code block

```
private static final Logger LOGGER = LoggerFactory.getLogger(xxx.class);
```

5. 对于 trace/debug/info 级别的日志输出，必须进行日志级别的开关判断

反例：

Code block

```
public void doSth(){
```

分析：

如果配**置的日志级别是 warn** 的话，上述日志不会打印，但是会执行字符串拼接操作，如果 name 是对象， 还会执行 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，因此建议**加日志开关判断**。

正例：

在 trace、debug、info 级别日志打印前加上对应级别的日志开关判断，通常可以将开关判断逻辑包装在日志工具类中，统一实现。

```
Code block
1   public void doSth(){
```

6. 捕获异常后不要使用 e.printStackTrace()打印日志

反例：

```
Code block
1   public void doSth(){
```

分析：

- e.printStackTrace()打印出的堆栈日志跟业务代码日志是交错混合在一起的，通常排查异常日志不太方便。
- e.printStackTrace()语句产生的字符串记录的是堆栈信息，如果信息太长太多，字符串常量池所在的内存块没有空间了，即内存满了，系统请求将被阻塞。

正例：

```
Code block
1   public void doSth(){
```

7. 打印异常日志一定要输出全部错误信息

反例：

- 没有打印异常 e，无法定位出现什么类型的异常

```
Code block
1   public void doSth(){
```

- 没有记录详细的堆栈异常信息，只记录错误基本描述信息，不利于排查问题。

```
Code block
1   public void doSth(){
```

正例：

一般日志框架中的 warn、error 级别均有存在传递 Throwable 异常类型的 API，可以直接将抛出的异常传入日志 API 中。

```
Code block
1   void error(String var1, Throwable var2);
```

8. 日志打印时禁止直接用 JSON 工具将对象转换成 String

反例：

```
Code block
1   public void doSth(){
```

分析：

- fastjson 等序列化组件是通过调用对象的 get 方法将对象进行序列化，如果对象里某些 get 方法被覆写，存在抛出异常的情况，则可能会因为打印日志而影响正常业务流程的执行。
- 打日志过程中对一些对象的序列化过程也是比较耗性能的。首先序列化过程本身时一个计算密集型过程，费 cpu。其次这个过程会产生很多中间对象，对内存也不太友好。

正例：

可以使用对象的 toString()方法打印对象信息，如果代码中没有对 toString()有定制化逻辑的话，可以使用 apache 的 ToStringBulider 工具。

```
Code block
1   public void doSth(){
```

9. 不要打印无意义(无业务上下文、无关联日志链路 id)的日志

反例：

- 不带任何业务信息的日志，对排查故障毫无意义。

```
Code block
1   public void doSth(){
```

- 对于无异常分支的代码打印日志，一般流程下，异常分支都会打日志，如果没有出现异常，那就说明正常执行了。

```
Code block
1   public void doSth(){
```

正例：

- 日志一定要带相关的业务信息，有利于排查问题快速定位到原因。

```
Code block
1   public void doSth(){
```

- 对于打印过多的无意义日志，可以直接干掉或者以 debug 级别打印。

10. 不要在循环中打印 INFO 级别日志

反例：

```
Code block
1   public void doSth(){
```

11. 不要打印重复的日志

反例：

```
Code block
1   public void doSth(String s){
```

```
Code block
1   public void doSth(String s) {
```

- 在每一个嵌套环节都打印了重复的日志。
- 不要记录日志后又抛出异常。抛出去的异常，一般外层会处理。如果不处理，那为什么还要抛出去？原则是，无论是否发生异常，都不要在不同地方重复记录针对同一事件的日志消息。

正例：

直接干掉或者将日志降级成 debug 级别日志

12. 避免敏感信息输出

13. 日志单行大小必须不超过 200K

3.2 推荐

1. 日志语言尽量使用英文

建议：尽量在打印日志时输出英文，防止中文编码与终端不一致导致打印出现乱码的情况，对故障定位和排查存在一定的干扰。

2. 重要方法可以记录调用日志

建议在重要方法入口记录方法调用日志，出口打印出参，对于排查问题会有很大的帮助。

Code block

```
1   public String doSth(String id, String type){
```

3. 在核心业务逻辑中遇到 if...else 等条件，尽量每个分支首行都打印日志

在编写**核心业务逻辑代码**时，如遇到 **if...else...或者 switch** 这样的条件，可以在分支的首行就打印日志，这样排查问题时，就可以通过日志，确定进入了哪个分支，代码逻辑更清晰，也更方便排查问题了。

建议：

Code block

```
1   public void doSth(){
```

4. 建议只打印必要的参数，不要整个对象打印

反例：

Code block

```
1   public void doSth(){
```

分析：

首先分析下自己是否必须把所有对象里的字段打印出来？如果对象中有 50 个字段，但只需其中两个参数就可以定位具体的原因，那么全量打印字段将浪费内容空间且因为字段过多，影响根因排查。

正例：

Code block

```
1   public void doSth(){
```

使用这个种方法需及时防止 npe，并考虑是否核心场景，核心场景建议还是打全，避免漏打、少打影响线上问题定位 &排查。