**Overview**

Generally speaking, there are two types of logs: business logs and exception logs. Using logs, we hope to achieve the following goals:

* Record and monitor the operation of the program;
* If necessary, you can learn more about the internal operating status of the program;
* Minimal impact on system performance;

Java logging framework

* Log4j or Log4j2 - Apache's open source project. By using Log4j, we can control the delivery of log information to the console, files, GUI components, even socket servers, NT event recorders, UNIX Syslog daemons, etc .; The user can also control the output format of each log; by defining the level of each log information, the user can control the log generation process in more detail. These can be flexibly configured through a configuration file (XML or Properties file) without modifying the program code. Log4j 2 is an upgrade of its predecessor, referring to many features of Logback;
* Logback - Logback is another open source diary component designed by the founder of log4j. Logback is currently divided into three modules: logback-core, logback-classic and logback-access. Logback-core is the basic module of the other two modules. logback-classic is an improved version of log4j. In addition, logback-classic fully implements the SLF4J API so that you can easily replace it with other journaling systems such as log4j or JDK14 Logging;
* Slf4j - SLF4J is to provide a simple and unified interface for various Logging APIs, so that users can configure the Logging API implementation they want during deployment;

Logging level

The Java logging framework generally provides the following log levels. By default, the info level is turned on. The debug and trace level logs are not output in the production environment. In the development and test environments, the debug level can be opened through different log configuration files.

* Fatal - serious, causing service interruption errors;
* Error - other errors runtime errors;
* Warn - warning information, such as a program calling an interface that is about to be invalidated, improper use of the interface, the running state is not expected, but it can still be processed, etc.
* Info - meaningful event information, such as program startup, shutdown event, received request event, etc.
* Debug - debug information, which can record the step of detailed business processing, and the current variable status;
* Trace - more detailed trace information;

Meaningful log

* Usually, some meaningful status data is recorded in the program log:

1. When the program starts and exits;
2. The time it takes to run the program;
3. The execution progress of a time-consuming program;
4. Changes in the status of important variables;

**Example:**

**long startTime = System.currentTime();**

**logger.info(“execution cost : {} ms”, (System.currentTIme() - startTime));**

Log Exception properly

* Always logging the customized message together with the whole exception object in catch statement(**IMPORTANT**)

**Example:**

**Integer a = null;**  
**try {  
 if (a.toString().equals("1")) {  
 a = 0;  
 }  
} catch (Exception ex) {  
 logger.error("Error happened", ex);  
}**

* Do not throw the exception after logging error message in catch statement

**Example:**

**catch(Exception ex){  
 logger.error("Error happened", ex);**

**throw new Exception("Error happened", ex)**

**}**

* Do not return from catch statement without logging any useful message

**Example:**

**catch(Exception ex){  
 return false;**

**}**

* Do not only log customized error message

**Example:**

**catch(Exception ex){  
 logger.error("Error happened");**

**}**

* Do not only log message of the exception, Most of the time the message is not helpful

**Example:**

**logger.error(ex.getMessage());** //When ex is NullPointerException, the return value of ex.getMessage() is just null.

Logs easy to read, easy to parse

There are two groups of receivers particularly interested in your application logs: human beings (programmers belong to this group) and computers (typically shell scripts written by system administrators). Logs should be suitable for both of these groups

* **logger.debug("Request TTL set to: {} ({})", new Date(ttl), ttl);** // Request TTL set to: Wed Apr 28 20:14:12 CEST 2010 (1272478452437)

* **final String duration = DurationFormatUtils.formatDurationWords(durationMillis, true, true);**

**logger.info("Importing took: {}ms ({})", durationMillis, duration);** //Importing took: 123456789ms (1 day 10 hours 17 minutes 36 seconds)

Log security

* Don’t log any sensitive information in the log. For sensitive information such as user ID numbers, passwords can be encrypted and stored; to prevent the log file from inadvertently leaking and keep the user's data safe; the log usually does not allow modification, and if necessary, it can be checked through Check to verify that the log is correct.
* Sanitize all user inputs before writing them in the logs. This includes checking its size, content, encoding, syntax, etc... As for any user input, validate using whitelists whenever possible. Enabling users to write what they want in your logs can have many impacts. It could for example use all your storage space or compromise your log indexing service.
* Log enough information to monitor suspicious activities and evaluate the impact an attacker might have on your systems. Register events such as failed logins, successful logins, server side input validation failures, access denials and any important transaction.
* Monitor the logs for any suspicious activity.

Logging basic rules

1. The API in the log system (Log4j2, Logback) cannot be used directly in the application but should rely on the use of the logging framework the API in SLF4J uses the logging framework of the facade mode, which is conducive to the maintenance and unification of various types of log processing methods.
2. Only one Logger object is usually used in an object. The Logger should be static final. Only use private final in a few cases where the logger needs to be passed in the constructor.

**Example:　private static final Logger logger = LoggerFactory.getLogger(Test.class);**

The naming method of extended logs (temporary monitoring, access logs, etc.) in the application. appName\_logType\_logName.log.

logType: such as stats/monitor/access, etc .; logName: log description. The benefits of this naming: You can know what application, what type, and what purpose the log file belongs to through the file name, which is also conducive to the classification search.

It is recommended to categorize the logs, such as storing the error log and the business log separately so that developers can view it, and it is convenient to monitor the system in time through the log.

Example: In the force-web application, the time zone conversion exception is monitored separately, such as: force\_web\_timeZoneConvert.log

1. In the log output, the stitching between string variables uses a placeholder. Because string stitching will use the StringBuilder append () method, there is a certain performance cost. Using placeholders is just a replacement action, which can effectively improve performance.

**Example: logger.debug("Processing trade with id: {} and symbol: {}", id, symbol);**

1. For the log output of the trace / debug / info level, the switch of log level must be added.

**Example:**

**if (logger.isDebugEnabled()) {**

**logger.debug("Current ID is: {} and name is: {}", id, getName());**

**}**

1. To avoid repeatedly printing the log and wasting disk space, be sure to set **additivity = false** in log4j.xml

**Example: <logger name="com.hal.prdm.service " additivity="false">**

1. Exception information should include two types of information: scene information and exception stack information. If not processed, then throw up through the keyword throws.

**Example: logger.error(“Error executing workflow welltest - {}”, e.getMessage(), e);**

1. It is forbidden to output debug logs in the production environment; selectively output info logs; if you use warn to record business behavior information just after going online, you must pay attention to the problem of log output, avoid bursting the server disk, and remember to delete these logs in time.
2. You can use the warn log level to record user input parameter errors. If it is not necessary, please do not set an error level in this scene to avoid frequent alarms.
3. Log information must be described in English

Bad practice

* Using System.out.println() statements in code

1. Cannot easily change log levels, turn it off, customize it

* Using printStackTrace method to print error message

1. Error message will not be written into logging file
2. Waste memory resource

Logger framework POM dependency

<properties>

<log4jVersion>2.13.3</log4jVersion >

<deploy.env>prod</deploy.env> <!-- This property is default value of deployment environment, the value should will assigned via –Ddeploy.env={prod|dev} -->

</properties>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-slf4j-impl</artifactId>

<version>${log4jVersion}</version>

</dependency>

<!-- Specify resource directory dynamically -->

<build>

<resources>

<resource>

<directory>src/main/resources/${deploy.env}</directory>

</resource>

</resources>

</build>

Logger configuration

Development

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN" monitorInterval="30">

<Properties>

<Property name="LOG\_PATTERN">

%d{yyyy-MM-dd HH:mm:ss.SSS} %5p ${hostName} [%15.15t - %tid - %r] %-30.30c{1.} : %m%n

</Property>

</Properties>

<Appenders>

<Console name="ConsoleAppender" target="SYSTEM\_OUT" follow="true">

<PatternLayout pattern="${LOG\_PATTERN}"/>

</Console>

<!-- Rolling File Appender -->

<RollingFile name="FileAppender" fileName=" ${env:LOGS\_DIR:-logs}/dfs12-workflow-manager.log"

filePattern="${env:LOGS\_DIR:-logs}/dfs12-workflow-manager -%d{yyyy-MM-dd}-%i.log">

<PatternLayout>

<Pattern>${LOG\_PATTERN}</Pattern>

</PatternLayout>

<Policies>

<TimeBasedTriggeringPolicy interval="1" />

<SizeBasedTriggeringPolicy size="100MB" />

</Policies>

<DefaultRolloverStrategy max="10"/>

</RollingFile>

</Appenders>

<Loggers>

<AsyncLogger name="com.hal.dfs" level="debug"

additivity="false">

<AppenderRef ref="ConsoleAppender" />

<AppenderRef ref="FileAppender" />

</AsyncLogger>

<Root level="info">

<AppenderRef ref="ConsoleAppender" />

<AppenderRef ref="FileAppender" />

</Root>

</Loggers>

</Configuration>

Production

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN" monitorInterval="30">

<Properties>

<Property name="LOG\_PATTERN">

%d{yyyy-MM-dd HH:mm:ss.SSS} %5p ${hostName} [%15.15t - %tid - %r] %-30.30c{1.} : %m%n

</Property>

</Properties>

<Appenders>

<!-- Rolling File Appender -->

<RollingFile name="FileAppender" fileName="${env:LOGS\_DIR:-logs} /dfs12-workflow-manager.log"

filePattern="${env:LOGS\_DIR:-logs}/dfs12-workflow-manager -%d{yyyy-MM-dd}-%i.log">

<PatternLayout>

<Pattern>${LOG\_PATTERN}</Pattern>

</PatternLayout>

<Policies>

<TimeBasedTriggeringPolicy interval="1" />

<SizeBasedTriggeringPolicy size="100MB" />

</Policies>

<DefaultRolloverStrategy max="10"/>

</RollingFile>

</Appenders>

<Loggers>

<AsyncLogger name="com.hal.dfs " level="info"

additivity="false">

<AppenderRef ref="FileAppender" />

</AsyncLogger>

<Root level="info">

<AppenderRef ref="FileAppender" />

</Root>

</Loggers>

</Configuration>

**${env:LOGS\_DIR:-logs} – Get specific directory from environment variable LOGS\_DIR, default value ‘logs’ will be used when LOGS\_DIR is not exist.**