# 日志

目前Java应用常用的日志框架log4j, apache commons logging,

目前Java应用常用的日志系统log4j2, logBack ,slf4j, JUL等。

分布式环境针对日志实时分析统计的日志，一般采用apache flume、facebook scribe等

1. 日志框架：提供日志调用的接口，实际的日志输出托付给日志系统实现。

JCL(Jakarta Commons Logging)：比較流行的日志框架，非常多框架都依赖JCL，比如Spring等。

SLF4j：提供新的API，初衷是配合Logback使用，但同一时候兼容Log4j，比如实现类slf4j-log4jl2

2.日志系统：负责输出日志

Log4j：经典的一种日志解决方式。内部把日志系统抽象封装成Logger 、appender 、pattern 等实现。我们能够通过配置文件轻松的实现日志系统的管理和多样化配置。

Log4j2：Log4j的2.0版本号。对Log4j进行了优化。比方支持參数API、支持异步appender、插件式架构等

Logback：Log4j的替代产品。须要配合日志框架SLF4j使用

JUL(java.util.logging)：JDK提供的日志系统。较混乱，不经常使用

3.maven依赖

以下是： slf4j + Log4j2 日志组件的引入

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.13</version>

</dependency>

<dependency>

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

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

<version>2.4.1</version>

</dependency>

<!--兼容log4j-->

<dependency>

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

<artifactId>log4j-1.2-api</artifactId>

<version>2.0</version>

</dependency>

<dependency>

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

<artifactId>log4j-api</artifactId>

<version>2.4.1</version>

</dependency>

<dependency>

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

<artifactId>log4j-core</artifactId>

<version>2.4.1</version>

</dependency>

<!--log4j2 异步appender须要-->

<dependency>

<groupId>com.lmax</groupId>

<artifactId>disruptor</artifactId>

<version>3.2.0</version>

</dependency>

# log4j

## 是什么

用来做日志的工具！市场上95%以上的软件都在使用它来作为日志工具！

给你的程序做日记！

## 让log4j与System.out.println()比较

\* 前者专业日志工具

> 可以随意的指定输出目的地（可以是控制台，文件）

> 可以灵活控制级别（只输出错误级别的日志，而信息级别的不输出）

> 可以控制输出的格式！

\* 后者只能输出到控制台！无法控制级别（信息、错误）！

## 示例代码

public class Demo1{

@Test

public void fun1() {

/\*

\* 1. 得到日志器

\*/

Logger logger = Logger.getLogger(Demo1.class);

/\*

\* 2. 创建格式（简单格式）

\*/

Layout layout = new SimpleLayout();

/\*

\* 3. 使用格式来创建目的地（控制台）

\*/

Appender appender = new ConsoleAppender(layout);

/\*

\* 4. 把目的地给日志器

\*/

logger.addAppender(appender);

// 设置日志器级别

logger.setLevel(Level.ALL);//只输出错误级别以上的日志

/\*

\* 5. 输出日志

\*/

logger.debug("hello");//调试级别

logger.info("hello");//信息级别

logger.warn("hello");//警告级别

logger.error("hello");//错误级别

logger.fatal("hello");//大错误级别

}

}

## ~~日志器相关信息~~

~~1. 目的地(发送器)：Appender~~

~~2. 格式：Layout~~

~~3. 级别：Level~~

~~通过配置来设置以上相关信息~~

~~log4j.properties --> 还是在使用properties配置文件！~~

~~1. 把log4j.properties文件放到src下，一般项目习惯于创建一个与src同级的resource的包，将这些properties文件放进去！~~

## 不同级别的使用

### ERROR

基本概念

影响到程序正常运行、当前请求正常运行的异常情况:

* 打开配置文件失败
* 所有第三方对接的异常(包括第三方返回错误码)
* 所有影响功能使用的异常，包括:SQLException和除了业务异常之外的所有异常(RuntimeException和Exception)

不应该出现的情况:

比如要使用Azure传图片，但是Azure未响应

如果有Throwable信息，需要记录完成的堆栈信息:

log.error("获取用户[{}]的用户信息时出错",userName,e);

说明

如果进行了抛出异常操作，请不要记录error日志，由最终处理方进行处理：

反例(不要这么做):

try{  
    ....  
}catch(Exception ex){  
  String errorMessage=String.format("Error while reading information of user [%s]",userName);  
  logger.error(errorMessage,ex);  
  throw new UserServiceException(errorMessage,ex);  
}

### WARN

基本概念

不应该出现但是不影响程序、当前请求正常运行的异常情况:

* 有容错机制的时候出现的错误情况
* 找不到配置文件，但是系统能自动创建配置文件

即将接近临界值的时候，例如：

缓存池占用达到警告线

业务异常的记录,比如:

当接口抛出业务异常时，应该记录此异常

### INFO

基本概念

系统运行信息

* Service方法中对于系统/业务状态的变更
* 主要逻辑中的分步骤

外部接口部分

* 客户端请求参数(REST/WS)
* 调用第三方时的调用参数和调用结果

说明

* 并不是所有的service都进行出入口打点记录,单一、简单service是没有意义的(job除外,job需要记录开始和结束,)。反例(不要这么做):

public List listByBaseType(Integer baseTypeId) {  
 log.info("开始查询基地");  
 BaseExample ex=new BaseExample();  
 BaseExample.Criteria ctr = ex.createCriteria();  
 ctr.andIsDeleteEqualTo(IsDelete.USE.getValue());  
 Optionals.doIfPresent(baseTypeId, ctr::andBaseTypeIdEqualTo);  
    log.info("查询基地结束");  
 return baseRepository.selectByExample(ex);  
}

* 对于复杂的业务逻辑，需要进行日志打点，以及埋点记录，比如电商系统中的下订单逻辑，以及OrderAction操作(业务状态变更)。
* 对于整个系统的提供出的接口(REST/WS)，使用info记录入参
* 如果所有的service为SOA架构，那么可以看成是一个外部接口提供方，那么必须记录入参。
* 调用其他第三方服务时，所有的出参和入参是必须要记录的(因为你很难追溯第三方模块发生的问题)

### DEBUG

基本概念

* 可以填写所有的想知道的相关信息(但不代表可以随便写，debug信息要有意义,最好有相关参数)
* 生产环境需要关闭DEBUG信息
* 如果在生产情况下需要开启DEBUG,需要使用开关进行管理，不能一直开启。

说明

如果代码中出现以下代码，可以进行优化:

//1. 获取用户基本薪资  
//2. 获取用户休假情况  
//3. 计算用户应得薪资

优化后的代码:

logger.debug("开始获取员工[{}] [{}]年基本薪资",employee,year);

logger.debug("获取员工[{}] [{}]年的基本薪资为[{}]",employee,year,basicSalary);  
logger.debug("开始获取员工[{}] [{}]年[{}]月休假情况",employee,year,month);  
logger.debug("员工[{}][{}]年[{}]月年假/病假/事假为[{}]/[{}]/[{}]",employee,year,month,annualLeaveDays,sickLeaveDays,noPayLeaveDays);  
logger.debug("开始计算员工[{}][{}]年[{}]月应得薪资",employee,year,month);  
logger.debug("员工[{}] [{}]年[{}]月应得薪资为[{}]",employee,year,month,actualSalary);

### TRACE

基本概念

特别详细的系统运行完成信息，业务代码中，不要使用.(除非有特殊用意，否则请使用DEBUG级别替代)

规范示例说明

@Override  
@Transactional  
public void createUserAndBindMobile(@NotBlank String mobile, @NotNull User user) throws CreateConflictException{  
    boolean debug = log.isDebugEnabled();  
    if(debug){  
        log.debug("开始创建用户并绑定手机号. args[mobile=[{}],user=[{}]]", mobile, LogObjects.toString(user));  
    }  
    try {  
        user.setCreateTime(new Date());  
        user.setUpdateTime(new Date());  
        userRepository.insertSelective(user);  
        if(debug){  
            log.debug("创建用户信息成功. insertedUser=[{}]",LogObjects.toString(user));  
        }  
        UserMobileRelationship relationship = new UserMobileRelationship();  
        relationship.setMobile(mobile);  
        relationship.setOpenId(user.getOpenId());  
        relationship.setCreateTime(new Date());  
        relationship.setUpdateTime(new Date());  
        userMobileRelationshipRepository.insertOnDuplicateKey(relationship);  
        if(debug){  
            log.debug("绑定手机成功. relationship=[{}]",LogObjects.toString(relationship));  
        }  
        log.info("创建用户并绑定手机号. userId=[{}],openId=[{}],mobile=[{}]",user.getId(),user.getOpenId(),mobile); // 如果考虑安全，手机号记得脱敏  
    }catch(DuplicateKeyException e){  
        log.info("创建用户并绑定手机号失败,已存在相同的用户. openId=[{}],mobile=[{}]",user.getOpenId(),mobile);  
        throw new CreateConflictException("创建用户发生冲突, openid=[%s]",user.getOpenId());  
    }  
}

# 使用slf4j

* 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。
* 实现方式统一使用： Logback框架

# 打日志的正确方式

## 什么时候应该打日志

* 当你遇到问题的时候，只能通过debug功能来确定问题，你应该考虑打日志，良好的系统，是可以通过日志进行问题定为的。
* 当你碰到if…else 或者 switch这样的分支时，要在分支的首行打印日志，用来确定进入了哪个分支
* 经常以功能为核心进行开发，你应该在提交代码前，可以确定通过日志可以看到整个流程

## 基本格式

必须使用参数化信息的方式:

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

对于debug日志，必须判断是否为debug级别后，才进行使用:

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

不要进行字符串拼接,那样会产生很多String对象，占用空间，影响性能。  
反例(不要这么做):

logger.debug("Processing trade with id: " + id + " symbol: " + symbol);

## 使用[]进行参数变量隔离

如有参数变量，应该写成如下写法:

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

这样的格式写法，可读性更好，对于排查问题更有帮助。

# Logback

SpringBoot工程自带logback和slf4j的依赖，所以重点放在编写配置文件上，需要引入什么依赖，日志依赖冲突统统都不需要我们管了。

全面讲解logback的使用和源码： <https://mp.weixin.qq.com/s/3HVbJRr2SVDf4jrGUgaWpg>

## logback

logback框架会默认加载classpath下命名为logback-spring或logback的配置文件。将所有日志都存储在一个文件中文件大小也随着应用的运行越来越大并且不好排查问题，正确的做法应该是将error日志和其他日志分开，并且不同级别的日志根据时间段进行记录存储

案例如下

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

<configuration>

<property resource="logback.properties"/>

<appender name="CONSOLE-LOG" class="ch.qos.logback.core.ConsoleAppender">

<layout class="ch.qos.logback.classic.PatternLayout">

<pattern>[%d{yyyy-MM-dd' 'HH:mm:ss.sss}] [%C] [%t] [%L] [%-5p] %m%n</pattern>

</layout>

</appender>

<!--获取比info级别高(包括info级别)但除error级别的日志-->

<appender name="INFO-LOG" class="ch.qos.logback.core.rolling.RollingFileAppender">

<filter class="ch.qos.logback.classic.filter.LevelFilter">

<level>ERROR</level>

<onMatch>DENY</onMatch>

<onMismatch>ACCEPT</onMismatch>

</filter>

<encoder>

<pattern>[%d{yyyy-MM-dd' 'HH:mm:ss.sss}] [%C] [%t] [%L] [%-5p] %m%n</pattern>

</encoder>

<!--滚动策略-->

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!--路径-->

<fileNamePattern>${LOG\_INFO\_HOME}//%d.log</fileNamePattern>

<maxHistory>30</maxHistory>

</rollingPolicy>

</appender>

<appender name="ERROR-LOG" class="ch.qos.logback.core.rolling.RollingFileAppender">

<filter class="ch.qos.logback.classic.filter.ThresholdFilter">

<level>ERROR</level>

</filter>

<encoder>

<pattern>[%d{yyyy-MM-dd' 'HH:mm:ss.sss}] [%C] [%t] [%L] [%-5p] %m%n</pattern>

</encoder>

<!--滚动策略-->

<rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">

<!--路径-->

<fileNamePattern>${LOG\_ERROR\_HOME}//%d.log</fileNamePattern>

<maxHistory>30</maxHistory>

</rollingPolicy>

</appender>

<root level="info">

<appender-ref ref="CONSOLE-LOG" />

<appender-ref ref="INFO-LOG" />

<appender-ref ref="ERROR-LOG" />

</root>

</configuration>

## 部分标签说明

<root>标签，必填标签，用来指定最基础的日志输出级别

<appender-ref>标签，添加append

<append>标签，通过使用该标签指定日志的收集策略

name属性指定appender命名

class属性指定输出策略，通常有两种，控制台输出和文件输出，文件输出就是将日志进行一个持久化。ConsoleAppender将日志输出到控制台

<filter>标签，通过使用该标签指定过滤策略

<level>标签指定过滤的类型

<encoder>标签，使用该标签下的标签指定日志输出格式

<rollingPolicy>标签指定收集策略，比如基于时间进行收集

<fileNamePattern>标签指定生成日志保存地址，通过这样配置已经实现了分类分天手机日志的目标了

## logback 高级特性异步输出日志

之前的日志配置方式是基于同步的，每次日志输出到文件都会进行一次磁盘IO。采用异步写日志的方式而不让此次写日志发生磁盘IO，阻塞线程从而造成不必要的性能损耗。异步输出日志的方式很简单，添加一个基于异步写日志的appender，并指向原先配置的appender即可

<!-- 异步输出 -->

<appender name="ASYNC-INFO" class="ch.qos.logback.classic.AsyncAppender">

<!-- 不丢失日志.默认的,如果队列的80%已满,则会丢弃TRACT、DEBUG、INFO级别的日志 -->

<discardingThreshold>0</discardingThreshold>

<!-- 更改默认的队列的深度,该值会影响性能.默认值为256 -->

<queueSize>256</queueSize>

<!-- 添加附加的appender,最多只能添加一个 -->

<appender-ref ref="INFO-LOG"/>

</appender>

<appender name="ASYNC-ERROR" class="ch.qos.logback.classic.AsyncAppender">

<!-- 不丢失日志.默认的,如果队列的80%已满,则会丢弃TRACT、DEBUG、INFO级别的日志 -->

<discardingThreshold>0</discardingThreshold>

<!-- 更改默认的队列的深度,该值会影响性能.默认值为256 -->

<queueSize>256</queueSize>

<!-- 添加附加的appender,最多只能添加一个 -->

<appender-ref ref="ERROR-LOG"/>

</appender>

# 实践案例

## 如何记录日志

<https://mp.weixin.qq.com/s/JC51S_bI02npm4CE5NEEow>

## 设置log文件的生成路径.txt

注意：使用log4j的时候，必须使用Commons-logging.jar.

一、通过配置listener，向这个应用中添加路径信息的属性。

1.创建listener

public class log4jlistener implements ServletContextListener {

/\*\*

\* 系统属性的key

\*/

public static final String log4jdirkey = "log4jdir";

/\*\*

\* 关闭应用的时候，移除公共属性

\*/

public void contextDestroyed(ServletContextEvent servletcontextevent) {

System.getProperties().remove(log4jdirkey);

}

/\*\*

\* 初始化的时候，向系统里添加公共属性。

\*/

public void contextInitialized(ServletContextEvent servletcontextevent) {

String log4jdir = servletcontextevent.getServletContext().getRealPath("/");

//这个目录是D:\Program Files\apache-tomcat-6.0.35\webapps\JqueryDemo\

System.setProperty(log4jdirkey, log4jdir);

}

}

2.配置web.xml,尽可能往前配置。

<listener>

<listener-class>com.zx.web.listener.log4jlistener</listener-class>

</listener>

3.编辑src下的log4j.properties

log4j.rootLogger=info,c1,f1

log4j.appender.c1=org.apache.log4j.ConsoleAppender

log4j.appender.c1.layout=org.apache.log4j.PatternLayout

log4j.appender.c1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%p] %m [%t] %c [%l]%n

log4j.appender.f1.File=${log4jdir}WEB-INF/logs/app1.log <----这就是在该应用的具体目录下创建日志文件。

log4j.appender.f1=org.apache.log4j.DailyRollingFileAppender

log4j.appender.f1.layout=org.apache.log4j.PatternLayout

log4j.appender.f1.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%p] %m [%t] %c [%l]%n