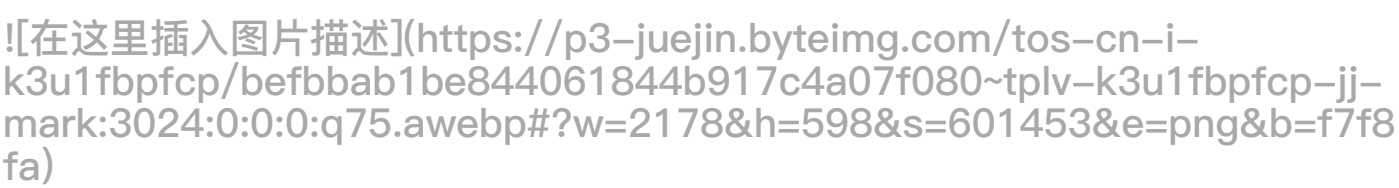
记一次线上日志堆栈不打印问题排查(附:高并发系统日志打印方案可收藏)

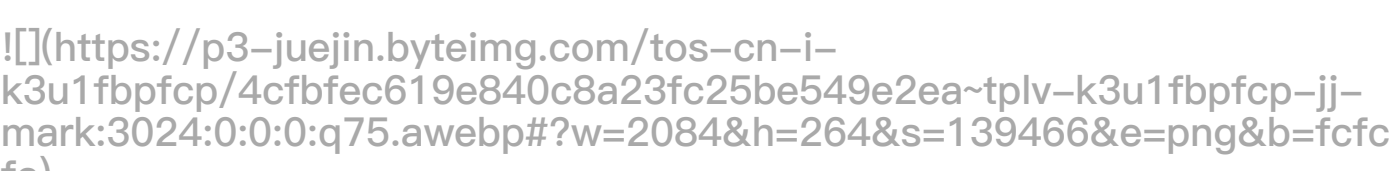
一.线上的日志堆栈不打印了

线上的报错 **error** 日志不打印详细的堆栈信息了.本着追根到底的精神.仔细排查了下.目前的日志打印过程.系统和代码虽然是公司的,解决问题都是自己的呀.

正常打印.拥有详细的堆栈信息.



不正常打印.仅打印**Exception** **className**

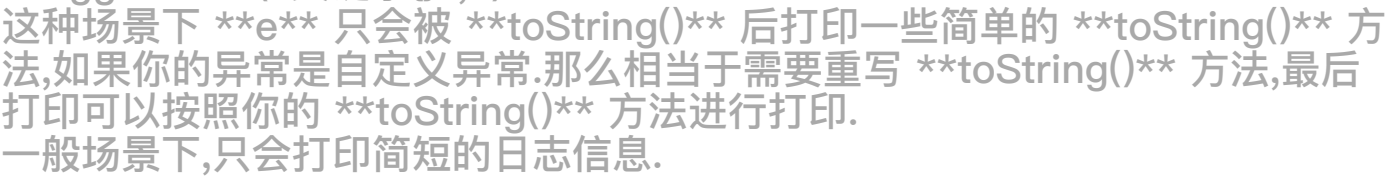


二.一步一步仔细排查

第一反应是哪个神仙写法导致的.一般场景下不打印线上错误日志有这么几种场景.

* logger.error(”关键字{}”,e)

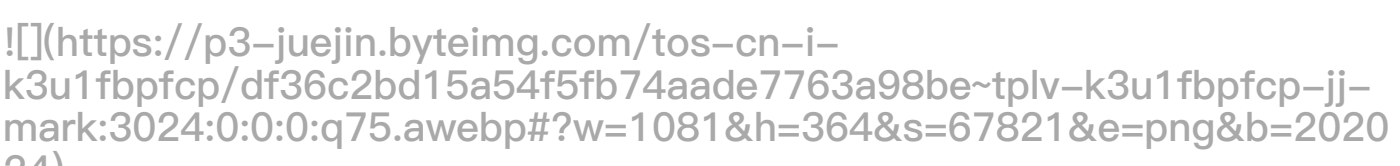
这种场景下 **e** 只会被 **toString()** 后打印一些简单的 **toString()** 方法,如果你的异常是自定义异常.那么相当于需要重写 **toString()** 方法,最后打印可以按照你的 **toString()** 方法进行打印.
一般场景下,只会打印简短的日志信息.



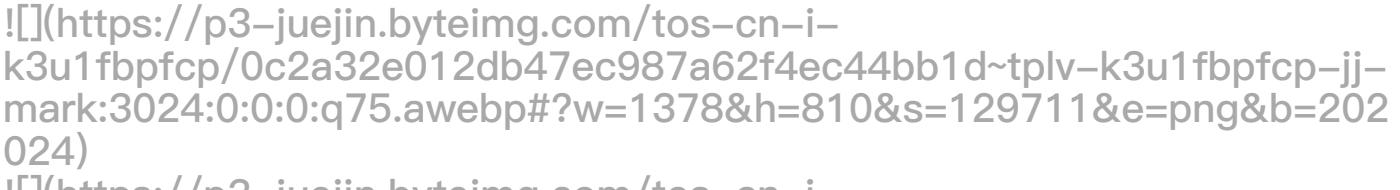
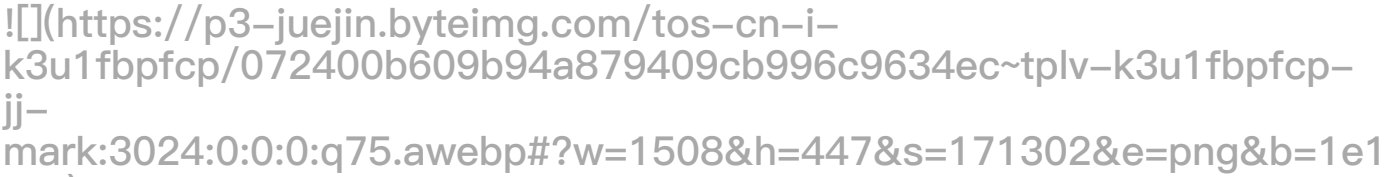
mark:3024:0:0:0:q75.awebp#?w=616&h=282&s=70365&e=png&b=202124)
* logger.error(”关键字”+e.toString())

这种写法其实和上面第一种写法是一样的逻辑.仅仅只是通过手动拼接的逻辑来处理的.

一步两步跟踪代码.


mark:3024:0:0:0:q75.awebp#?w=1081&h=364&s=67821&e=png&b=202024)
第二步下面的日志打印.

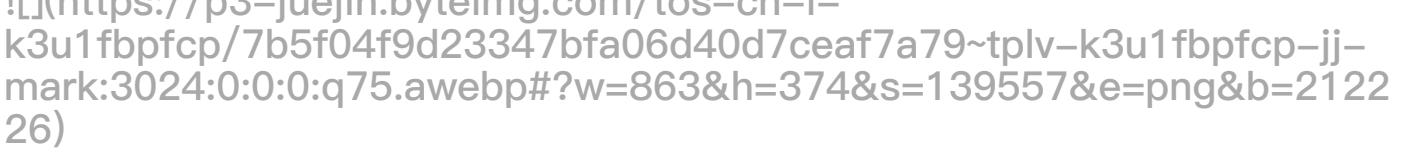

mark:3024:0:0:0:q75.awebp#?w=1049&h=330&s=62799&e=png&b=1f2023)
如上第二步真实的日志打印如上.可以发现第二步其实没有什么问题.真实就是直接调用的 **logger** 的打印方法.

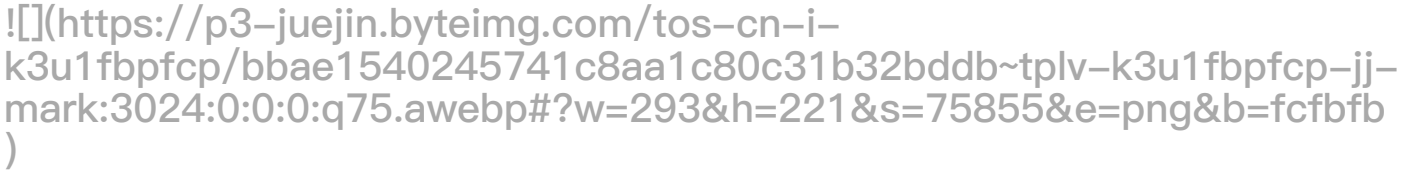

mark:3024:0:0:0:q75.awebp#?w=1378&h=810&s=129711&e=png&b=202024)

mark:3024:0:0:0:q75.awebp#?w=1508&h=447&s=171302&e=png&b=1e1f22)
那其实可以发现具体就是在这个方法循环过程中出了问题,没有进入到的循环体中导致无法正确进行拼接,程序员的思维.通过配置发现整体的堆栈深度配置为 **1** .先不论 **1** 是否合适.那么这里有一个判断.

```
int stackTraceLineNum = stackTraceLineNum >
stackTraceElements.length ?
stackTraceElements.length : stackTraceLineNum;
```

配置的值大于堆栈的深度 **length** 则使用**stack**具体的深度.否则则使用配置的**stackTraceLineNum**.那么这种情况下就是 **stackTraceElements.length=0**.

那么问题来了什么情况会导致这个**length**为空呢?.点进去.


巴拉巴拉说了很长.发现有这么一段话.***In the extreme case. a virtual machine that has no stack trace information concerning this throwable is permitted to return a zero-length array from this method***.平时喜欢学外语的朋友都知道.这段话:在极端场景下.虚拟机考虑到堆栈信息可能被允许返回一个空数组从这个方法.我草,这是什么鬼.一时间发现这还是我写的那个 **Java** 吗 **???**


线索到这里基本断了.

理了理思路重新梳理了下发现问题的过程,重新将代码部署在预发环境.整体跑了一个遍,并通过 **debug** 也没哟复现.都是可以正常获取到 **stack** **length**.就这样上午一天过去了.并没有什么进展.第二天,脑子稍微清醒了点,确实想的不一样了.再来回想一下线上环境和预发环境的特点,特征.回归原点思考问题.

线上环境:

* 流量特征:流量场景丰富.流量压力并发高.
* 机器环境:线上机器一般配置较高.

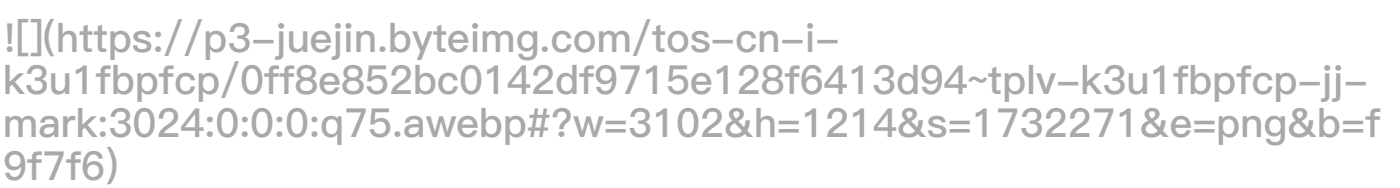
预发开发等环境:

* 流量特征:场景贫乏.流量压力小.
* 机器环境:线上机器可能配置较差[可以通过配置升级来抹平]

那么其实还是要磨平线上环境之间的差异.并尝试了批量的压测,最终他出现了.老小子你还是来了.

通过发现每每执行到一段时间堆栈就不打印了,这个是不是就是注释里说的极端情况.联系注释提到的 **virtual machine**. 次数+**vm**. 这块你想到了什么.不知道你想到了什么 .在老早之前总结过 **jit** 的一些特性.一时间马上检索一下(关键字: **jit** 堆栈不打印优化).确实有了.

oracle官网: [www.oracle.com/java/techno...](http://cxyroad.com/”https://www.oracle.com/java/technologies/javase/release-notes-introduction.html”)


三.最后搞定
======

马上开搞.

```
-XX:+OmitStackTraceInFastThrow
```

配置上预发压测走起来,搞定.

幸福之余也再次查阅了一些资料.

发现这是 **java** 的 **fast** **throw** 优化,感叹确实 **jit** 牛哈.通过次数来检测是否进行详细的堆栈打印.返回来也能理解,虚拟机默认在一些次数打印之后就不再打印详细的堆栈.默认开发者已经到相关的堆栈信息了.以此来减少性能损耗.主要应对常见的一些 **Exception** .

```
NullPointerException
```

ArithmeticException
ArrayIndexOutOfBoundsException
ArrayStoreException
ClassCastException

```

**fast throw** 本身从 **jvm** 的 **jit** 层面来说是一个正向的优化措施,在线上整体的流量还是比较高的场景下,如果一旦将屏蔽优化内容.将徒增很多日志的打印,导致磁盘的 IO 升高以及磁盘利用率升高.所以本次的屏蔽也仅仅是开启了部分分组的配置,即能够保证在指定机器排查问题的场景下完成日志的追踪,也能够包装 **fast throw** 自带优化能够大部分被利用到.如果你的应用也是这种场景我也是建议使用这种方案.

## 四.聊一聊线上日志到底应该怎么打印

不止于此通过这次我们正好来聊一聊日志打印的这样一个流程.以及分享目前在搞的一个高并发系统是怎样去打印日志的.

### 4.1 日志打印的诉求

一般场景下打印日志主要基于以下几种诉求.

目标

* 技术诉求:排查问题;基于关键字主动告警监测系统异常情况.
* 业务运营诉求:基于关键字做流量统计,数据分析获取用户的具体产品功能反馈.以期进一步改进产品方案

关键路径

* 正常业务处理的日志告警关键字打印,需要能将堆栈打印;
* 支持一些巨量的存储以及结构化复杂查询能力
* 具备大促高峰流量的降级可配置

### 4.2 常见的系统日志上报方案
```

第一步:应用系统所在机器.一般是 **Linux** 完成系统日志采集.当发生所关心的日志文件变化时.由监听的进程将日志发送这里有很多种选择.常见的如下:

rsyslog:相对性能比较好,久经考验的老战士.配置较为复杂.
filebeat:性能相比没有**rsyslog**好.但使用相对简单.使用也比较广泛.

第二步:完成服务端日志接收.这里一般是 **Logstash**.也就是我们常说的**ELK**里的**L**.

第三步:完成日志入库.这里的库可以是
message.**es**.**redis**.**mysql** 集成的三方还是比较多的.

第四步:选择合适的工具来完成日志的展示.比如说:**kibana**

4.2.1 ELK 方案

![在这里插入图片描述](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/03eee61f57b94cee9196210d53dbac40~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1924&h=866&s=81343&e=png&b=2a2a2a)

一般比较常见的大厂方案基本如此.

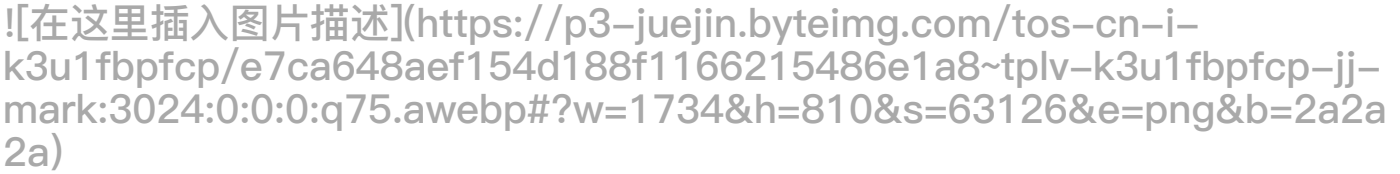
* step1:通过 **slf4j** 门面完成应用日志文件输出.
* step2:通过 **rsyslog** 监听文件变更完成日志文件发送.
* step3:通过 **logstash** 完成日志服务端解析发送.支持的插件也比较多.具体可参考: [www.elastic.co/guide/en/lo...](http://cxyroad.com/"https://www.elastic.co/guide/en/logstash/current/output-plugins.html")
-step4:完成 **es** 数据存储.

一键开箱即用的**es**存储满足巨量的存储且支持水平扩展.并提供比较友好的查询 **kibana** 能力.

适用场景:业务系统的日志打印抽取.
优点:完全独立在系统应用之外不影响任何系统性能;日志侧可以做到资源的友好

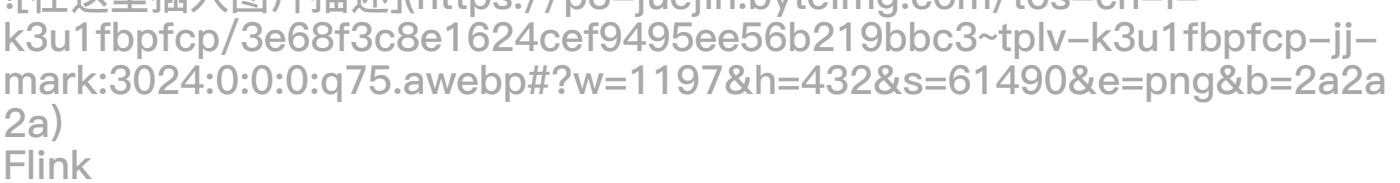
伸缩.
缺点:巨量的日志打印后容易出现资源的浪费

4.2.2 自定义log appender 完成应用日志采集.



这种自定义**appender**方案比较适用于一些常见的后台操作.如**ERP**内部的用户日志记录.如:用户在2024年04月26日17:42:29 完成对租户 101 ，库存 201 的商品
库存调整。

4.2.3 拓展:日志分析


Flink

> 数据采集：

> 首先，确定您要分析的日志类型和来源。这可能包括服务器日志、应用程序日志、网络流量日志等。
> 然后，选择合适的日志采集工具。一些常用的选择包括 Logstash、Fluentd 和 Filebeat。这些工具可以将日志数据从源头发送到后续处理阶段。

> 数据存储：

> 在选择数据存储方案时，需要考虑数据量、数据类型和查询需求。一种常见的选择是使用 Elasticsearch 作为主要的数据存储引擎，它可以提供强大的全文搜索和聚合功能。

> 另外，您还可以考虑使用 Apache Kafka 作为数据缓冲区，以平衡数据流量和后续处理的速度。

> 数据处理：

> Flink 是一个强大的流处理引擎，可以用于实时的数据处理和分析。您可以使用 Flink 的流处理功能来处理实时生成的日志数据。
> 首先，将日志数据发送到 Flink 中进行流处理。您可以使用 Flink 的连接器来将数据从 Kafka 或其他数据源读取到 Flink 中。
> 然后，使用 Flink 的算子和转换功能来执行您所需的数据处理操作。这可能包括过滤、聚合、计数、窗口操作等。
> 最后，将处理后的数据发送到适当的目标，例如数据库、可视化工具或其他存储系统。

> 可视化展示：

> 最后一步是将处理后的数据以易于理解和可视化的方式展示出来。您可以使用一些数据可视化工具，如 Kibana、Grafana 或自定义的前端界面。
> 这些工具可以帮助您创建仪表板、图表和报告，以直观地展示日志数据的关键指标和趋势。

常见的分析工具比如还有 Doris, ClickHouse. [Doris 与 ClickHouse 的深度对比及选型建议](http://cxyroad.com/
”https://blog.csdn.net/u011250186/article/details/135963133”)

4.3 日志常见框架傻傻分不清

一套日志编写标准.

> SLF4J：是一套简易Java日志门面，本身并无日志的实现。（Simple Logging Facade for Java，缩写Slf4j）

以下为一些具体的实现.

> 1. Jul (Java Util Logging)：JDK中的日志记录工具，也常称为JDKLog、jdk-logging，自Java1.4以来的官方日志实现。
> 2. Log4j：Apache Log4j是一个基于Java的日志记录工具。它是由Ceki Gülcü首创的，现在则是Apache软件基金会的一个项目。 Log4j是几种Java日志框架之一。
> 3. Log4j2：一个具体的日志实现框架，是Log4j 1的下一个版本，与Log4j 1发生了很大的变化，Log4j 2不兼容Log4j 1。
> 4. Logback：一个具体的日志实现框架，和Slf4j是同一个作者，但其性能更好。

4.4 日志在高并发系统中需要注意的 tips

4.4.1 配置合理的日志级别

> 常见的日志级别.
> 分别是 **all**, **error**、**warning**、**info**、**debug**、**trace**。

日常开发中，我们需要选择恰当的日志级别

* all: 所有日志级别打印.
* fatal :无法修复的程序异常
* error： 错误日志，指比较严重的错误，对正常业务有影响，需要运维配置监控的；通常我们会在throw异常时进行打印。
* warn： 警告日志，一般的错误，对业务影响不大，但是需要开发。
* info： 信息日志，记录排查问题的关键信息，通常我们会在方法入口和出口打印出入参信息等。
* debug： 用于开发DEBUG的，关键逻辑里面的运行时数据。通常我们用来调试关键程序，例如我们可以将SQL的日志级别调为debug，在本地或者测试环境调试时，可以看到具体SQL。
* trace： 最详细的信息，一般这些信息只记录到日志文件中。

对于一些流量比较高的场景线上尽量不打印非**error**级别日志.无差别打印所有级别日志将会对磁盘的造成性能损耗，很可能就会将磁盘打爆，进而影响正常的系统的业务**IO**。

4.4.2 记录合理的链路

基于上一点我们总会有排查一些线上问题的时候，这无可避免。这时候如果要打印 用以排查问题的日志（非error级别），对于这种场景还是秉着尽可能少打印的原则。
比如说：保证能够做到随机抽量打印或者业务id过滤后打印，这个就视具体的业务类型而定。
如：库存，商品支持商品id来进行过滤；订单，售后支持订单id来进行过滤；

具备日志白名单(用户 **id or pin**)后查阅用户链路

4.4.3 配置异步打印

* 添加依赖

```
<dependency>
        <groupId>com.lmax</groupId>
        <artifactId>disruptor</artifactId>
        <version>3.3.4</version>
</dependency>
```

* 配置AsyncLogger

AsyncLogger是log4j2 的重头戏，也是官方推荐的异步方式。它可以使得调用Logger.log返回的更快，可以有两种选择：全局异步和混合异步。

* + 全局异步：所有的日志都异步的记录，在配置文件上不用做任何改动，只需要添加一个
log4j2.component.properties 配置
Log4jContextSelector=org.apache.logging.log4j.core.async.AsyncLoggerContextSelector
* + 混合异步：可以在应用中同时使用同步日志和异步日志，这使得日志的配置方式更加灵活

```
```

```
<Loggers>
        <AsyncLogger name="com.baixiu.aynclogger" level="trace"
includeLocation="false" additivity="false">
                <AppenderRef ref="file"/>
        </AsyncLogger>
        <Root level="info" includeLocation="true">
                <AppenderRef ref="file"/>
        </Root>
</Loggers>
```

附完成的日志文件配置.

```
<?xml version='1.0' encoding="UTF-8" ?>
<Configuration status="DEBUG" monitorInterval="60">

    <Properties>
        <!-- 在当前目录下创建名为log目录做日志存放的目录 -->
        <Property name="LOG_FILE_PREFIX"
value="/xxx/applicationName" />
        <!-- 日志文件大小，超过这个大小将被压缩 -->
        <Property name="LOG_FILE_MAX_SIZE" value="512 MB" />
        <!-- 触发rollover时最大计数 -->
        <Property name="REQUEST_MAX_HISTORY" value="2" />
        <Property name="MAIN_MAX_HISTORY" value="3" />
        <Property name="RPC_MAX_HISTORY" value="2" />
        <Property name="ERROR_MAX_HISTORY" value="3" />
        <Property name="FILE_LOG_PATTERN" value="%d{yyyy-MM-dd
HH:mm:ss}  %5level [%t] %logger{50} - [%X{PFTID}] - %msg%n"/>
        <Property name="LOG_ERROR_PATTERN" value="%d{yyyy-MM-
dd HH:mm:ss} %5level [%t] %logger{50} - [%X{PFTID}] -
%m%throwable{10}%n"    />
    </Properties>

    <Appenders>

        <Console name="CONSOLE" target="SYSTEM_OUT">
            <PatternLayout pattern="[%7r] %6p - %30.30c - %m \n"/>
        </Console>
        <RollingRandomAccessFile name="info"
fileName="${LOG_FILE_PREFIX}/info.log"
                                 filePattern="${LOG_FILE_PREFIX}/info.%d{yyyy-
MM-dd}_%i.log"
                                 immediateFlush="false">
```

```
            <Filters>
                <!-- 此Filter意思是，只输出ERROR级别的数据
                    DENY，日志将立即被抛弃不再经过其他过滤器；
                    NEUTRAL，有序列表里的下个过滤器过接着处理日志；
                    ACCEPT，日志会被立即处理，不再经过剩余过滤器。 -->
                <ThresholdFilter level="ERROR" onMatch="ACCEPT"
onMismatch="DENY"/>
            </Filters>

            <PatternLayout pattern="${FILE_LOG_PATTERN}"/>
            <Policies>
                <!-- 如果启用此配置，则日志会按文件名生成新文件，
                    即如果filePattern配置的日期格式为 %d{yyyy-MM-dd HH}
                    ，则每小时生成一个压缩文件， 如果filePattern配置的日期格式
为 %d{yyyy-MM-dd} ，则天生成一个压缩文件,默认为1 -->
                <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
                <SizeBasedTriggeringPolicy size="${LOG_FILE_MAX_SIZE}"/>
            </Policies>
            <!--文件夹下最多的文件个数-->
            <DefaultRolloverStrategy max="${MAIN_MAX_HISTORY}"/>
        </RollingRandomAccessFile>

        <RollingFile name="warning"
fileName="${LOG_FILE_PREFIX}/warning.log"
                     filePattern="${LOG_FILE_PREFIX}/warning.%d{yyyy-MM-
dd}_%i.log"
                     immediateFlush="false">
            <Filters>
                <ThresholdFilter level="ERROR" onMatch="ACCEPT"
onMismatch="DENY"/>
            </Filters>
            <PatternLayout pattern="${LOG_ERROR_PATTERN}"/>
            <Policies>
                <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
                <SizeBasedTriggeringPolicy size="${LOG_FILE_MAX_SIZE}"/>
            </Policies>
            <DefaultRolloverStrategy max="${ERROR_MAX_HISTORY}"/>
        </RollingFile>




        <RollingRandomAccessFile name="debug"
                                 fileName="${LOG_FILE_PREFIX}/debug.log"
filePattern="${LOG_FILE_PREFIX}/flowLimit.%d{yyyy-MM-dd}.log">
    <PatternLayout pattern="${FILE_LOG_PATTERN}"/>
            <Policies>
```

```
            <TimeBasedTriggeringPolicy/>
            <SizeBasedTriggeringPolicy size="${LOG_FILE_MAX_SIZE}"/>
        </Policies>
        <DefaultRolloverStrategy>
            <Delete basePath="${LOG_FILE_PREFIX}" maxDepth="1">
                <IfFileName glob="flowLimit.*.log" />
                <IfAccumulatedFileCount exceeds="5" />
            </Delete>
        </DefaultRolloverStrategy>
    </RollingRandomAccessFile>

     <RollingFile name="error"
fileName="${LOG_FILE_PREFIX}/error.log"
                 filePattern="${LOG_FILE_PREFIX}/error.%d{yyyy-MM-
dd}_%i.log"
                 immediateFlush="false">
        <Filters>
            <ThresholdFilter level="INFO" onMatch="ACCEPT"
onMismatch="DENY"/>
        </Filters>
        <PatternLayout pattern="${FILE_LOG_PATTERN}"/>
        <Policies>
            <TimeBasedTriggeringPolicy modulate="true" interval="1"/>
            <SizeBasedTriggeringPolicy size="${LOG_FILE_MAX_SIZE}"/>
        </Policies>
        <DefaultRolloverStrategy max="${REQUEST_MAX_HISTORY}"/>
    </RollingFile>

    <RollingRandomAccessFile name="customLog"
                             fileName="${LOG_FILE_PREFIX}/customLog.log"
filePattern="${LOG_FILE_PREFIX}/customLog.%d{yyyy-MM-dd}.log">
    <PatternLayout pattern="${FILE_LOG_PATTERN}"/>
        <Policies>
            <TimeBasedTriggeringPolicy/>
            <SizeBasedTriggeringPolicy size="${LOG_FILE_MAX_SIZE}"/>
        </Policies>
        <DefaultRolloverStrategy>
            <Delete basePath="${LOG_FILE_PREFIX}" maxDepth="1">
                <IfFileName glob="timeoutException.*.log" />
                <IfAccumulatedFileCount exceeds="5" />
            </Delete>
        </DefaultRolloverStrategy>
    </RollingRandomAccessFile>

</Appenders>

<Loggers>
```

```
        <AsyncLogger name="INFO" level="INFO" additivity="false">
            <AppenderRef ref="info"/>
        </AsyncLogger>

        <AsyncLogger name="WARN" level="WARN" additivity="false">
            <AppenderRef ref="warning"/>
        </AsyncLogger>

        <AsyncLogger name="debug" level="DEBUG" additivity="false">
            <AppenderRef ref="debug"/>
        </AsyncLogger>

        <AsyncLogger name="error" level="ERROR" additivity="false">
            <AppenderRef ref="error"/>
        </AsyncLogger>

        <AsyncRoot level="INFO" includeLocation="false">
        </AsyncRoot>

    </Loggers>
</Configuration>
```

如上配置： **com.baixiu.aynclogger** 日志是异步的，**root** 日志是同步的

> 赠人玫瑰 手有余香 我是柏修 一名持续更新的晚熟程序员
> 期待您的点赞,加收藏，加个不迷路，感谢
> 您的鼓励是我更新的最大动力
> ↓↓↓↓↓↓

原文链接: https://juejin.cn/post/7362062562479521804