# 日志追踪

本章描述如何借助于开源框架[TLog](https://tlog.yomahub.com/)，在全链路日志中打印RequestId。进而可以根据RequestId，串联起单个模块中同一请求的全部日志，或者在不同模块之间对同一请求进行日志追踪。项目集成Tlog后日志打印效果如下：

2023-11-17 10:21:47,249 INFO [http-nio-8080-exec-2] (TestController.java:21) <0><XTQg877497500041162752> main thread

2023-11-17 10:21:47,250 INFO [Thread-10] (TestController.java:23) <0><XTQg877497500041162752> sub thread

2023-11-17 10:21:47,250 INFO [common-task-worker-1] (TestController.java:26) <0><XTQg877497500041162752> pool thread

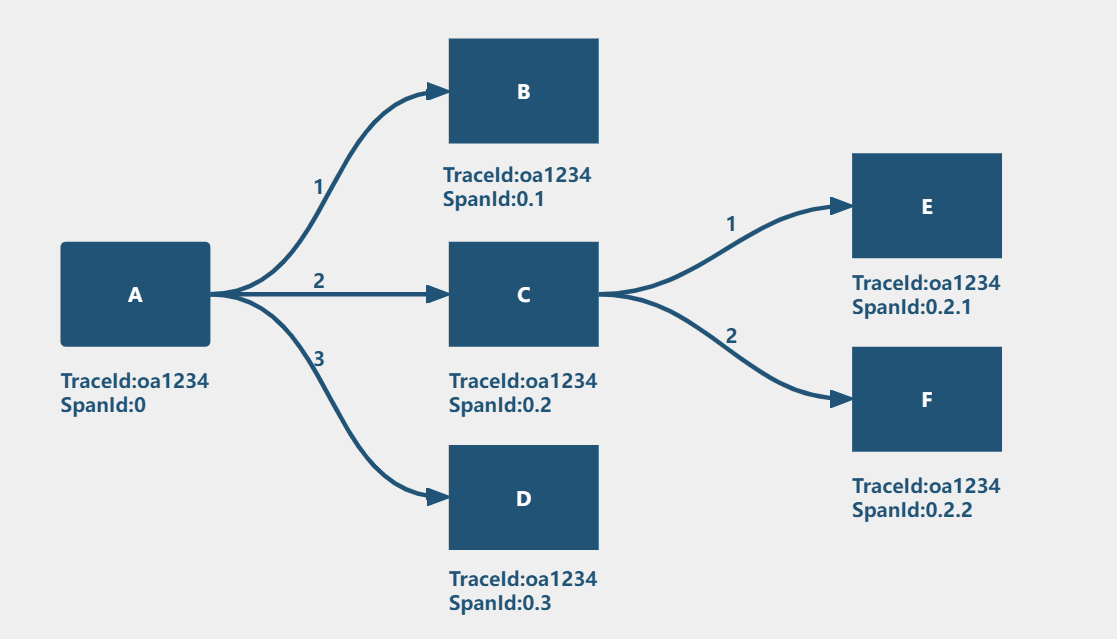
## 1.1. TLog概述

[TLog](https://tlog.yomahub.com/)是一个轻量级的分布式日志标记追踪框架。它不收集日志，也不需要额外的存储空间，它只是自动的对日志进行打标签，自动生成RequestId贯穿微服务的一整条链路。并且提供上下游节点信息。

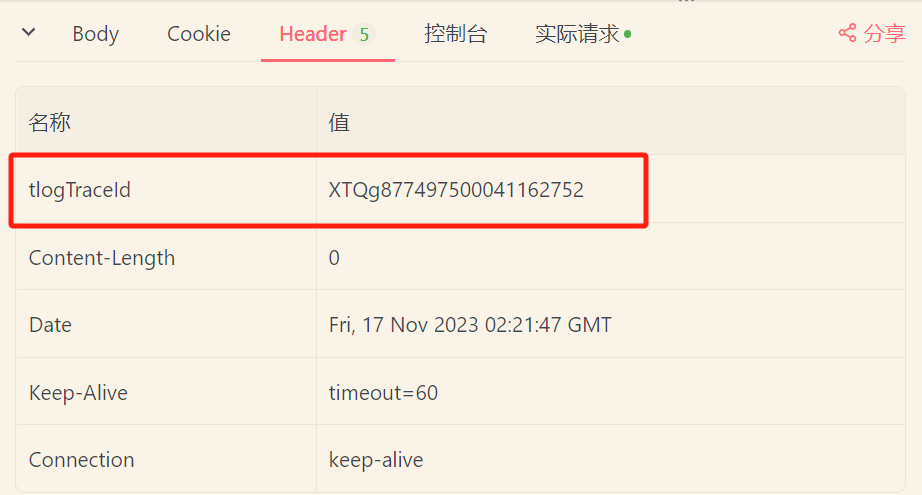
Tlog可实现对单模块接口调用、模块间Feign调用、XXL-JOB任务框架（包括异步任务和定时任务）、以及异步线程（包括线程池，多级异步线程等场景）的日志追踪。基于以上四条，该框架已自动覆盖项目内链路中主要节点，完成全链路日志追踪。

此外，它还具有如下特性：

* 对业务代码无侵入，引入依赖、增加配置即可。
* 几乎无性能损耗，快速稳定，经过压测，损耗在0.01%。
* 支持SpanId，可以打印出链路调用层次，SpanId生成逻辑如下图所示：



* 接口请求结果的header里，携带requestId，如下图所示。



* 可以在日志中打印上游节点信息，包括上游微服务的节点名称、Host Name以及IP地址。
* 通过在方法上增加特定注解，可以在日志中打印部分入参，如资源id、名称等。同时该标签也可以在整条链路中传递。
* 支持通过设置，在日志中打印出参数信息和调用时间信息，效果如下：

2020-12-01 19:20:07.768 [DubboServerHandler-127.0.0.1:30900-thread-2] INFO c.y.tlog.dubbo.filter.TLogDubboInvokeTimeFilter - <0.1><7592057736843136> [TLOG]开始调用接口[DemoService]的方法[sayHello],参数为:["jack"]

2020-12-01 19:20:07.795 [DubboServerHandler-127.0.0.1:30900-thread-2] INFO c.y.tlog.dubbo.filter.TLogDubboInvokeTimeFilter - <0.1><7592057736843136> [TLOG]结束接口[DemoService]中方法[sayHello]的调用,耗时为:90毫秒

关于Tlog的更详细的介绍，请移步官网：https://tlog.yomahub.com

## TLog集成

### 1.2.1. 添加依赖

这里可选择全量依赖，示例如下：

<dependency>  
   <groupId>com.yomahub</groupId>  
   <artifactId>tlog-all-spring-boot-starter</artifactId>  
   <version>1.5.1</version>  
</dependency>

如果不想依赖不必要的包，也可选择按需依赖，示例如下：

<dependency>  
   <groupId>com.yomahub</groupId>  
   <artifactId> tlog-XXX-spring-boot-starter</artifactId>  
   <version>1.5.1</version>  
</dependency>

具体模块和描述如下表:

|  |  |
| --- | --- |
| 模块名 | 描述 |
| tlog-feign-spring-boot-starter | 适用于spring cloud中open feign的项目 |
| tlog-gateway-spring-boot-starter | 适用于spring cloud中的gateway网关服务 |
| tlog-web-spring-boot-starter | 适用于有spring web的项目 |

### 1.2.2. 更新Logback配置文件

将encoder的class设置为AspectLogbackEncoder

<?xml version="1.0" encoding="UTF-8"?>  
<configuration debug="false">  
   <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">  
      **<!--这里替换成AspectLogbackEncoder-->  
 <encoder class="com.yomahub.tlog.core.enhance.logback.AspectLogbackEncoder">  
  <pattern>%d [%t] %-5level - %msg%n</pattern>  
 </encoder>**   </appender>​  
   <root level="INFO">  
       <appender-ref ref="STDOUT" />  
   </root>  
</configuration>

### 1.2.3. 添加AOP，处理XXL-JOB定时任务

如果不需要在定时任务中打印RequestId，可跳过此步骤，代码如下：

@Aspect  
@Component  
public class TLogXxlJobAspect {  
  
 private final TLogRPCHandler tLogRPCHandler = new TLogRPCHandler();  
  
 @Pointcut("execution(\* com.xxl.job.core.handler.IJobHandler+.\*(..))")  
 public void cut() { }  
  
 @Around("cut()")  
 public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {  
 try {  
 tLogRPCHandler.processProviderSide(new TLogLabelBean());  
 return joinPoint.proceed();  
 } finally {  
 tLogRPCHandler.cleanThreadLocal();  
 }  
 }  
   
}

### 1.2.4. 集成后日志打印效果

集成后日志打印效果如下：

2023-11-17 10:21:47,249 INFO [http-nio-8080-exec-2] (TestController.java:21)

<0><XTQg877497500041162752> main thread

2023-11-17 10:21:47,250 INFO [Thread-10] (TestController.java:23) <0><XTQg877497500041162752> sub thread

2023-11-17 10:21:47,250 INFO [common-task-worker-1] (TestController.java:26) <0><XTQg877497500041162752> pool thread

## TLog原理

### 1.3.1. TLog是怎样将RequestId输出到日志中的？

CMP中当前使用的日志框架是Logback，该框架支持使用方实现其接口，对Logback进行定制化扩展。TLog利用了Logback这个能力，其核心代码如下：

public class AspectLogbackConverter extends ClassicConverter {  
   @Override  
   public String convert(ILoggingEvent event) {  
       //只有在MDC没有设置的情况下才加到message里  
       if (!TLogContext.hasTLogMDC()) {  
           **String logValue = AspectLogContext.getLogValue();**           if (StringUtils.isBlank(logValue)) {  
               return event.getFormattedMessage();  
          } else {  
               **return logValue + " " + event.getFormattedMessage();**  
          }  
      } else {  
           return event.getFormattedMessage();  
      }  
  }  
}

当项目集成TLog后，每次调用logger.info()等方法打印日志前，都会先执行上述代码中的convert方法。该方法会从TLog定义的上下文AspectLogContext中取出包含RequestId等信息的logValue，然后与原本的日志信息拼接返回。

这样就实现了日志中打印RequestId等信息的功能，所以接下来的问题是，针对不同的场景，TLog是如何正确的设置RequestId到AspectLogContext上下文中的。

### 1.3.2. 接口调用时，TLog如何设置RequestId？

TLog实现了Spring的Web拦截器HandlerInterceptor接口，在该实现类中，如果发现上游请求Header中没有携带RequestId，则重新生成一个，并设置到AspectLogContext上下文中。实现该功能的核心代码如下：

public class TLogRPCHandler {  
​  
   protected static final Logger log = LoggerFactory.getLogger(TLogRPCHandler.class);  
​  
   public void processProviderSide(TLogLabelBean labelBean) {

……  
       //如果没有获取到，则重新生成一个traceId  
       if (StringUtils.isBlank(labelBean.getTraceId())) {  
           **labelBean.setTraceId(TLogIdGeneratorLoader.getIdGenerator().withExtData(labelBean.getExtData()).generateTraceId());**  
           log.debug("[TLOG]可能上一个节点[{}]没有正确传递traceId,重新生成traceId[{}]", labelBean.getPreIvkApp(), labelBean.getTraceId());  
      }  
         
       //生成日志标签  
       String tlogLabel = TLogLabelGenerator.generateTLogLabel(labelBean.getPreIvkApp(),  
               labelBean.getPreIvkHost(),  
               labelBean.getPreIp(),  
               TLogContext.getCurrIp(),  
               labelBean.getTraceId(),  
               TLogContext.getSpanId());  
​  
       //往日志切面器里放一个日志前缀  
      **AspectLogContext.putLogValue(tlogLabel);**

……  
  }  
     
}

### 1.3.3. Feign调用时，TLog如何设置RequestId？

TLog实现了Feign的RequestInterceptor接口，在该实现类中TLog将RequestId等信息塞到请求Header里，发送给下游服务，代码如下：

public class TLogFeignFilter implements RequestInterceptor {  
​  
   private static final Logger log = LoggerFactory.getLogger(TLogFeignFilter.class);  
​  
   @Value("${spring.application.name}")  
   private String appName;  
​  
   @Override  
   public void apply(RequestTemplate requestTemplate) {  
**String traceId = TLogContext.getTraceId();**  
​  
       if(StringUtils.isNotBlank(traceId)){  
**requestTemplate.header(TLogConstants.TLOG\_TRACE\_KEY, traceId);**  
           requestTemplate.header(TLogConstants.TLOG\_SPANID\_KEY, SpanIdGenerator.generateNextSpanId());  
           requestTemplate.header(TLogConstants.PRE\_IVK\_APP\_KEY, appName);  
           requestTemplate.header(TLogConstants.PRE\_IVK\_APP\_HOST, LocalhostUtil.getHostName());  
           requestTemplate.header(TLogConstants.PRE\_IP\_KEY, LocalhostUtil.getHostIp());  
      }else{  
           log.debug("[TLOG]本地threadLocal变量没有正确传递traceId,本次调用不传递traceId");  
      }  
  }  
}

Feign下游服务，从Header中取出RequestId等信息，设置到TLog的AspectLogContext上下文中，这一步骤和接口调用时设置方式相同。

### 1.3.4. 在XXL-JOB定时任务中，TLog如何设置RequestId？

TLog使用AOP，拦截所有的XXL-JOB定时任务，并在该AOP执行定时任务代码前，将RequestId设置到AspectLogContext上下文。AOP代码如下：

@Aspect  
public class TLogXxlJobAop {  
​  
   private final TLogRPCHandler tLogRPCHandler = new TLogRPCHandler();  
​  
  **@Pointcut("execution(\* com.xxl.job.core.handler.IJobHandler+.\*(..))")**  
   public void cut() {  
  }  
​  
   @Around("cut()")  
   public Object around(ProceedingJoinPoint jp) throws Throwable {  
       try{  
          **tLogRPCHandler.processProviderSide(new TLogLabelBean());**  
           return jp.proceed();  
      }finally {  
           tLogRPCHandler.cleanThreadLocal();  
      }  
  }  
}

因为该版本TLog适配的XXL-JOB，与项目当前集成的版本不一致，导致TLog在XXL-JOB定时任务中无法正常工作。所以在章节1.2中集成TLog时，才需要额外配置该AOP。

### 1.3.5. 在异步线程中，TLog如何设置RequestId？

JDK提供的ThreadLocal可以在线程内传递数据，InheritableThreadLocal可以在线程之间传递数据，但是JDK并未提供可以在线程池复用线程之间正确传递数据的能力。

因此，TLog选择借助于阿里开源的TransmittableThreadLocal来实现多级异步线程、特别是线程池内复用的线程之间的RequestId等信息的传递，它在前面一直提到的AspectLogContext上下文中发挥作用，代码如下：

public class AspectLogContext {

**private static TransmittableThreadLocal<String> logValueTL = new TransmittableThreadLocal<>();**

public static void putLogValue(String logValue) {

logValueTL.set(logValue);

}

public static string getLogValue() {

return logValueTL.get();

}

public static void remove() {

logValueTL.remove ();

}

}

所以，针对多级异步线程、线程池复用线程场景，TLog不需要再额外做其他处理，就已经可以实现在日志中正确打印RequestId了。

# 日志治理

当前项目中，各模块会将日志未经分类，全部输出到控制台。比如接口调用日志、icos接口调用参数及异常信息日志、定时任务日志、依赖的其他模块日志。部分模块存在控制台日志快速的不停的打印现象，导致在需要定位问题时，因日志刷新太快看不到相关日志，只能通过关键字、线程ID等信息在日志文件中检索。

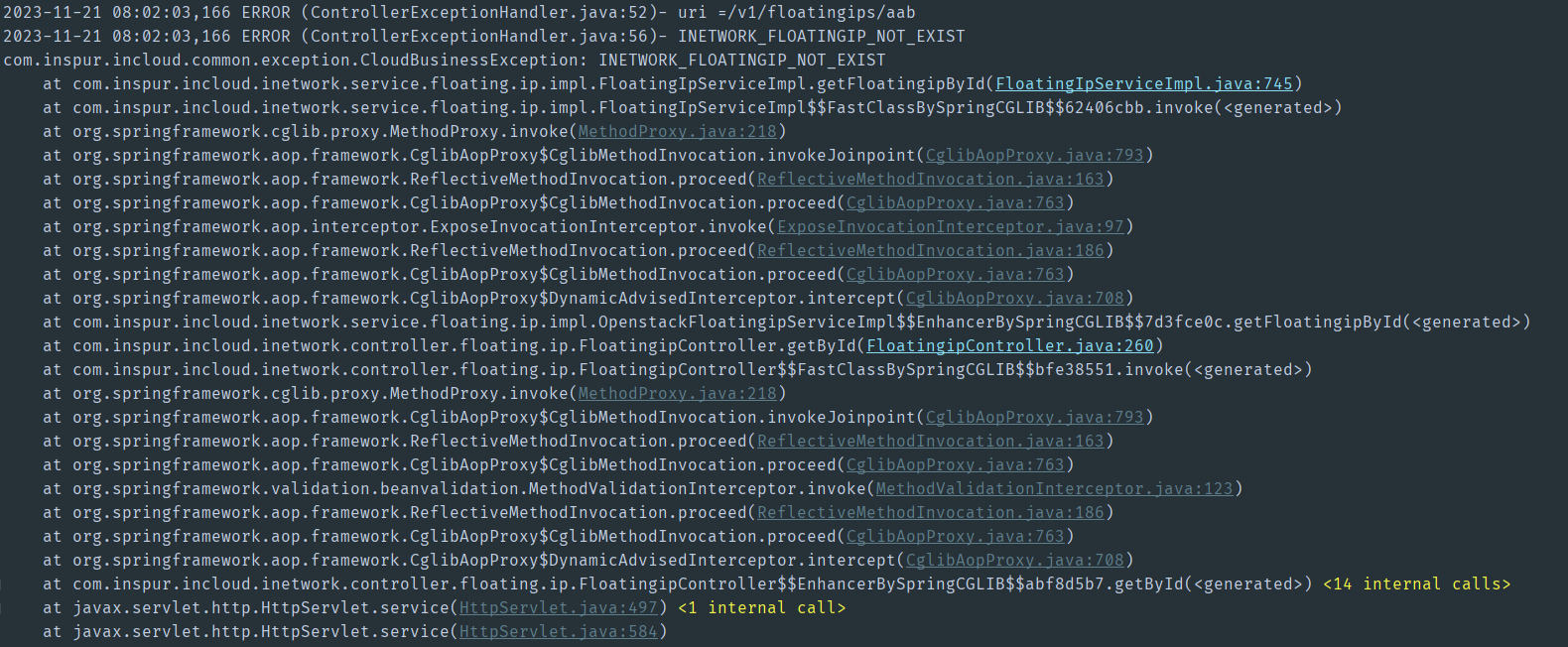
本章以ICompute模块为切入点，对该问题产生的主要原因进行分析，并提出针对性的解决方案。

此外，日志治理是一个长期的、持续的过程，随着业务的不断扩展，需要针对新出现的问题进行持续优化。

## 2.1. 问题原因

### 2.1.1. 部分异常无需打印异常堆栈

例如，不存在此资源异常，如浮动IP不存在、安全组不存在。通过错误码即可获取足够的信息，不需要打印完整的异常堆栈。



以inetwork为例，该问题可在统一异常处理器中集中处理，示例代码如下：

@Slf4j  
@ControllerAdvice  
@ResponseBody  
public class ControllerExceptionHandler {  
 ***// 这些错误不打印详细日志*** **final static List<String> exceptErrList = Arrays.asList("INETWORK\_FLOATINGIP\_NOT\_EXIST"  
 , "INETWORK\_SECURITY\_GROUP\_NOT\_EXIST", "INETWORK\_VPC\_NOT\_EXIST", "INETWORK\_NET\_NOT\_EXIST");**  
  
 @ExceptionHandler(CloudBusinessException.class)  
 public OperationResult<Object> handlerCloudBusinessException(CloudBusinessException e, HttpServletRequest request) {  
 // 1.打印错误日志  
 log.error("uri ={}", request.getRequestURI());  
 if **(exceptErrList.contains(e.getMsgCode())) {  
 log.error(e.getMessage());  
 }** else {  
 log.error(e.getMessage(), e);  
 }  
  
 // 2.记录操作日志  
 operLogUtils.writeOperLog(false, e);  
  
 // 3.return  
 return OperationResult.fail(e.getMsgCode(), e.getParamList());  
 }  
   
}

### 2.1.2. 错误堆栈频繁多次重复打印

例如，连接openstack资源池时，如果连接出错，会在不同位置重复打印四次日志。去除其中的三个，只保留一个在业务调用处的日志打印即可。

并且结合2.1.1章节，这些日志有些只需要打印e.getMessage()，比如401问题、超时问题，这样可以进一步减少日志在控制台上的输出，以及在全量日志中的存储。

public static OSClientObj onnectCenter(OpenstackCenterConnectionInfo center, String projectId) throws Exception {  
 log.info("------ start connect openstack ceneter: " + center.getIp() + ", projectMor: " + projectId + " ------");  
 OSClientV3 clientV3 = (OSClientV3) OpenstackConnPool.execute(new OSClientThreadExtended(null, parameters) {  
 @Override  
 public OSClientV3 innerCall(OSClientV3 os, Object[] parameters) throws Exception {  
 OSClientV3 osclientv3 = null;  
 try {   
 ...  
 } catch (Exception e) {  
 **log.error(e.toString(), e);**

throw **e;**  
 }  
 return osclientv3;  
 }  
  
 }, OpenstackConstant.THREAD\_DEFAULT\_WAIT\_TIME);  
 ...  
 } catch (Exception e) {  
 **log.error(e.toString(), e);**

throw **e;**  
 }  
 return osclientobj;  
 }

public class OpenstackConnPool {  
 public static Object execute(Callable task, Long outTime, boolean needResponse) throws RuntimeException {  
 try {  
 ...  
 } catch (Exception e) {  
 **log.error(e.getMessage(), e);** thrownew **RuntimeException(e);**  
 }  
 return null;  
 }  
}

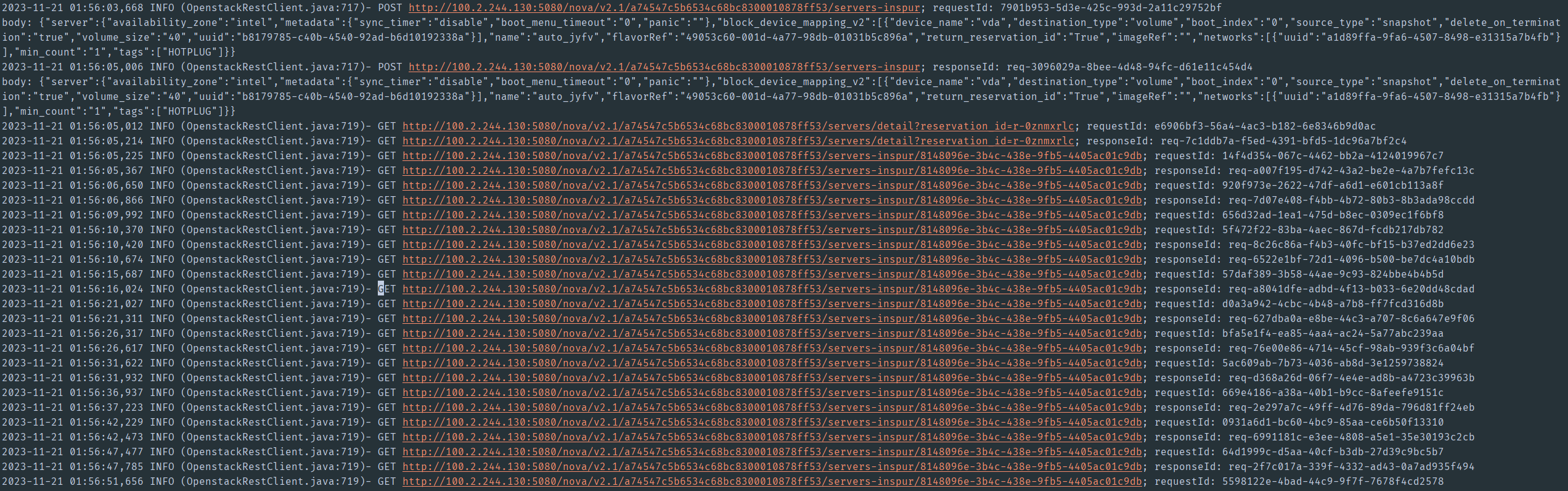
### 2.1.3. 接口调用和定时任务日志未分离

目前定时任务日志和接口调用日志，都输出在控制台。而平时排查问题时，查看定时任务日志的场景较少，所以可以将定时任务日志单独输出到一个文件，不在控制台打印。

### 2.1.4. icos request id频繁打印

icos request id是在调用icos接口时，传给icos的一个用来追踪icos接口调用的参数。以icompute模块为例，该模块频繁输出大量该类型日志。但是只有在需要icos协助定位问题时，该日志才有用，使用场景较少。所以可以将该类型日志单独输出到一个文件，不在控制台打印。

同时我们也对该类型日志进行了优化，增加了参数打印，可以不通过日志上下文，仅通过icos request id这一行日志，定位到需要查找的接口。



## 2.2. 解决方案

### 2.2.1. 接口调用和定时任务日志分离

将定时任务单独输出到一个文件，不在控制台打印。实现该方案需要借助于章节一的TLog框架。下面介绍下该方案的详细实施步骤：

1. 在1.2.3章节，我们已经添加了AOP，用来给XXL-JOB定时任务设置RequestId。现在这里要稍微做些调整，给RequestId拼接些特殊字符，给后面过滤用。

@Aspect  
@Component  
public class TLogXxlJobAspect {  
  
 private final TLogRPCHandler tLogRPCHandler = new TLogRPCHandler();  
  
 @Pointcut("execution(\* com.xxl.job.core.handler.IJobHandler+.\*(..))")  
 public void cut() { }  
  
 @Around("cut()")  
 public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {  
 try {  
 final TLogLabelBean labelBean = new TLogLabelBean();  
 **labelBean.setTraceId("xxl-job-itask-" + TLogIdGeneratorLoader.getIdGenerator().generateTraceId());**  
 tLogRPCHandler.processProviderSide(labelBean);  
 return joinPoint.proceed();  
 } finally {  
 tLogRPCHandler.cleanThreadLocal();  
 }  
 }  
   
}

1. 创建过滤器类，给logback的控制台appender使用，过滤掉定时任务日志，代码如下：

public class LogBackFilter extends Filter<ILoggingEvent> {  
 @Override  
 public FilterReply decide(ILoggingEvent iLoggingEvent) {  
 ......

if**(TLogContext.getTraceId() !=** null **&& TLogContext.getTraceId().contains("xxl-job-itask")){** return **FilterReply.DENY;  
 }**  
  
 return FilterReply.ACCEPT;  
 }  
}

上述代码中的TLogContext是TLog框架用来存储RequestId的上下文。

1. 更新logback配置文件，为控制台appender设置步骤2过滤器。

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
 <!-- filter中定义的logger，日志不输出到控制台 -->  
 **<**filter **class="com.inspur.incloud.inetwork.config.LogBackFilter"/>**  
 <encoder class="com.yomahub.tlog.core.enhance.logback.AspectLogbackEncoder">  
 <pattern>%d %p (%file:%line\)- %m%n</pattern>  
 <!-- 控制台也要使用UTF-8，不要使用GBK，否则会中文乱码 -->  
 <charset>UTF-8</charset>  
 </encoder>  
</appender>

1. 创建过滤器类，过滤掉**非定时任务**日志，代码如下：

public class LogBackXxlJobFilter extends Filter<ILoggingEvent> {  
 @Override  
 public FilterReply decide(ILoggingEvent iLoggingEvent) {  
 if**(TLogContext.getTraceId() !=** null **&& TLogContext.getTraceId().contains("xxl-job-itask")){** return **FilterReply.ACCEPT;  
 }** return FilterReply.DENY;  
 }  
}

1. 更新logback配置文件，新增appender只接收定时任务日志。

<!-- 定时任务日志 -->  
<appender name="xxl-job-itask" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 **<**File**>${LOG\_PATH}/xxl-job-itask.log</**File**>  
 <**filter **class="com.inspur.incloud.inetwork.config.LogBackXxlJobFilter"/>**  
 <encoder>  
 <pattern>%d %p (%file:%line\)- %m%n</pattern>  
 <charset>UTF-8</charset>  
 </encoder>  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <fileNamePattern>${LOG\_PATH}/daily/xxl-job-itask.log.%d{yyyy-MM-dd}.gz</fileNamePattern>  
 <maxHistory>30</maxHistory>  
 </rollingPolicy>  
</appender>  
  
<root level="INFO">  
 <appender-ref ref="console" />  
 <appender-ref ref="full" />  
 <appender-ref ref="SYSLOG" />  
 **<**appender-ref **ref="xxl-job-itask" />**  
</root>

### 2.2.2. icos request id日志单独存储

本章节介绍将icos request id日志在控制台屏蔽，单独输出到一个文件的方法。同时，该方法可复用到其他相同诉求场景。下面介绍下该方法实施的具体 步骤：

1. 创建过滤器类，给logback的控制台appender使用，过滤掉icos request id日志，代码如下：

public class LogBackFilter extends Filter<ILoggingEvent> {  
  
 // icos requestId相关日志  
 private static final String [] REQUEST\_LOGGER\_NAMES = new String[] {  
 "com.inspur.incloud.openstacksdk.connection.OpenstackRestClient"  
 };  
  
 @Override  
 public FilterReply decide(ILoggingEvent iLoggingEvent) {  
 final ArrayList<String> loggerNameList = new ArrayList<>();  
 loggerNameList.addAll(Arrays.asList(REQUEST\_LOGGER\_NAMES));  
  
 for (String loggerName : loggerNameList) {  
 if **(iLoggingEvent.getLoggerName().contains(loggerName)) {** return **FilterReply.DENY;  
 }** }  
  
 return FilterReply.ACCEPT;  
 }  
}

1. 更新logback配置文件，为控制台appender设置步骤1过滤器。

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">  
 <!-- filter中定义的logger，日志不输出到控制台 -->  
 **<**filter **class="com.inspur.incloud.inetwork.config.LogBackFilter"/>**  
 <encoder class="com.yomahub.tlog.core.enhance.logback.AspectLogbackEncoder">  
 <pattern>%d %p (%file:%line\)- %m%n</pattern>  
 <!-- 控制台也要使用UTF-8，不要使用GBK，否则会中文乱码 -->  
 <charset>UTF-8</charset>  
 </encoder>  
</appender>

1. 更新logback配置文件，单独定义一个logger将icos request id相关日志输出到新的appender。

<appender **name="icos-request-id"** class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <File>${LOG\_PATH}/icos-request-id.log</File>  
 <encoder>  
 <pattern>%d %p (%file:%line\)- %m%n</pattern>  
 <charset>UTF-8</charset>  
 </encoder>  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <fileNamePattern>${LOG\_PATH}/daily/icos-request-id.log.%d{yyyy-MM-dd}.gz</fileNamePattern>  
 <maxHistory>30</maxHistory>  
 </rollingPolicy>  
</appender>  
<logger name="com.inspur.incloud.openstacksdk.connection.OpenstackRestClient" level="INFO">  
 **<**appender-ref **ref="icos-request-id" />**  
</logger>

### 2.2.3. 遵守日志规范

以下主要参考于阿里代码规约，对于日志部分的规定

1. 【强制】可以只用e.getMessage()表达清楚的异常，不要打印完整的异常堆栈。例如根据ID查询不到资源、401无权限、404找不到资源。
2. 【强制】同一个请求的完全相同的异常堆栈，不要在不同位置重复打印。
3. 【强制】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 中的 API，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。

import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
private static final Logger logger = LoggerFactory.getLogger(Abc.class);

1. 【强制】日志文件推荐至少保存 15 天，因为有些异常具备以“周”为频次发生的特点。
2. 【强制】应用中的扩展日志(如打点、临时监控、访问日志等)命名方式: appName\_logType\_logName.log。logType:日志类型，推荐分类有 stats/desc/monitor/visit 等；logName:日志描述。这种命名的好处: 通过文件名就可知道日志文件属于什么应用，什么类型，什么目的，也有利于归类查找。 **正例** : mppserver 应用中单独监控时区转换异常，如: mppserver\_monitor\_timeZoneConvert.log **说明** : 推荐对日志进行分类，如将错误日志和业务日志分开存放，便于开发人员查看，也便于 通过日志对系统进行及时监控。
3. 【强制】对 trace/debug/info 级别的日志输出，必须使用条件输出形式或者使用占位符的方 式。 **说明** : logger.debug("Processing trade with id: " + id + " symbol: " + symbol);如果日志级别是 warn，上述日志不会打印，但是会执行字符串拼接操作，如果 symbol 是对象，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。 **正例** : (条件)

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

**正例** : (占位符)

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

1. 【强制】避免重复打印日志，浪费磁盘空间，务必在 log4j.xml 中设置 additivity=false。 **正例** : <logger name="com.taobao.dubbo.config" additivity="false">
2. 【强制】异常信息应该包括两类信息: 案发现场信息和异常堆栈信息。如果不处理，那么通过关键字 throws 往上抛出。 **正例** : logger.error(各类参数或者对象 toString + "\_" + e.getMessage(), e);
3. 【推荐】谨慎地记录日志。生产环境禁止输出 debug 日志；有选择地输出 info 日志；如果使用 warn 来记录刚上线时的业务行为信息，一定要注意日志输出量的问题，避免把服务器磁盘 撑爆，并记得及时删除这些观察日志。 **说明** : 大量地输出无效日志，不利于系统性能提升，也不利于快速定位错误点。记录日志时请思考: 这些日志真的有人看吗? 看到这条日志你能做什么? 能不能给问题排查带来好处?
4. 【参考】可以使用 warn 日志级别来记录用户输入参数错误的情况，避免用户投诉时，无所适从。注意日志输出的级别，error 级别只记录系统逻辑出错、异常等重要的错误信息。如非必要，请不要在此场景打出 error 级别。