# 1、日志链路

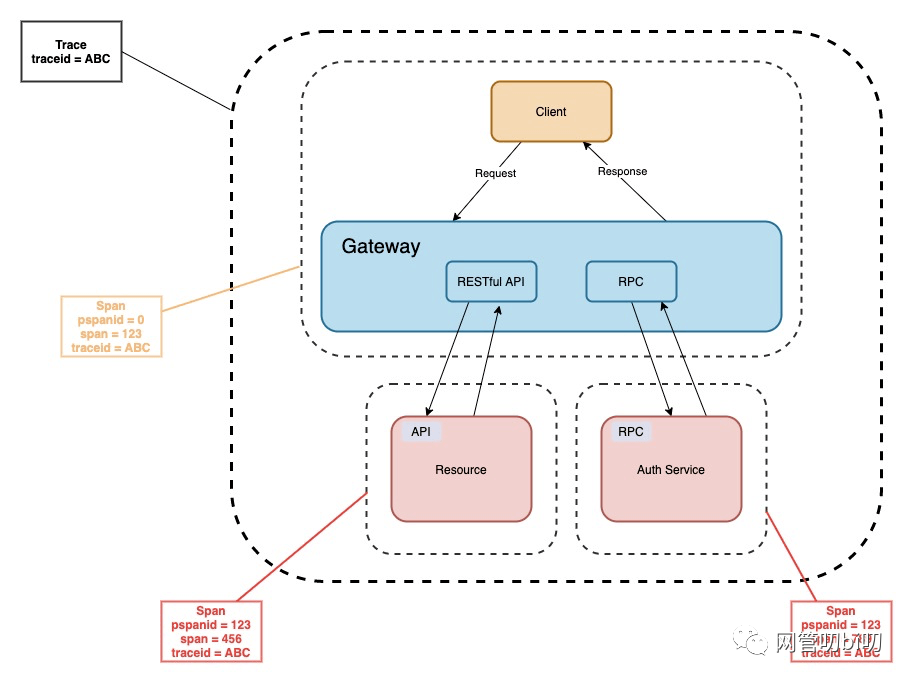
前端生成上报数据，调用接口上报kafka、大数据。

后端采用MDC+logback日志跟踪。

采用重要的概念：跟踪（trace）和 跨度（span）。trace 是请求在分布式系统中的整个链路视图，span 则代表整个链路中不同服务内部的视图，span 组合在一起就是整个 trace 的视图。

在整个请求的调用链中，请求会一直携带 traceid 往下游服务传递，每个服务内部也会生成自己的 spanid 用于生成自己的内部调用视图，并和traceid一起传递给下游服务。

traceid 在请求的整个调用链中始终保持不变，所以在日志中可以通过 traceid 查询到整个请求期间系统记录下来的所有日志。请求到达每个服务后，服务都会为请求生成spanid，而随请求一起从上游传过来的上游服务的 spanid 会被记录成parent-spanid或者叫 pspanid。当前服务生成的 spanid 随着请求一起再传到下游服务时，这个spanid 又会被下游服务当做 pspanid 记录。

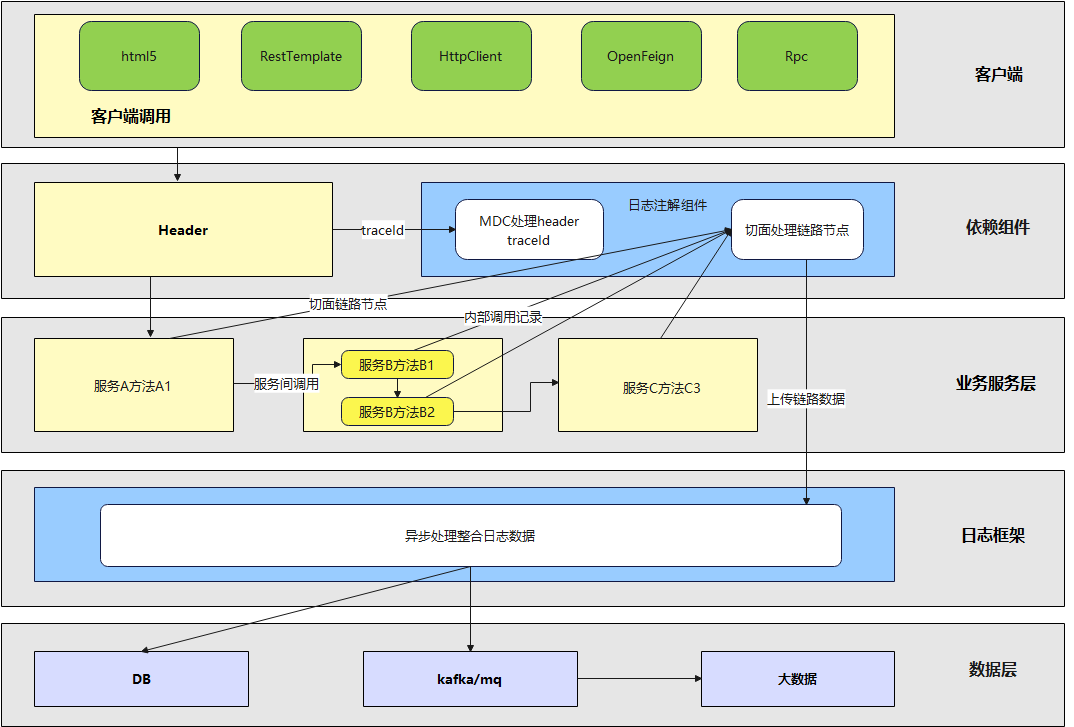


分布式链路跟踪中的trace和span通过在访问日志和业务日志里记录的traceid、spanid 和 pspanid 能完整的还原出整个请求的调用链路视图，对错误排查能起到很大的帮助。

上面就是分布式链路跟踪的原理，我们可以自己实现，也可以依赖 opentracing 这种开源的解决方案。一般是在请求到达网关的开始，生成本次请求的traceid 和 在网关服务内的spanid ，将他们放在HTTP 请求头或者RPC调用的元数据里，在调用下游服务时继续向下传递。下游的RESTful API服务的全局路由中间件和RPC服务的拦截器里会接收请求携带的traceid 和生成当次请求在服务内部的spanid，从上游接收到的 spanid 在这里会被转换成 pspanid。除此之外我们甚至可以把 traceid 和 spanid 注入到一些数据库连接池应用里，让记录的慢SQL日志里同样能打上 traceid 和 spanid 信息，为请求的响应过慢提供有效的分析数据。

后端上报数据：traceId、spanId/pspanId（工程名、类名、方法名）、时间、参数数据（json格式字符串）、IP、链路节点顺序。

整体框架图



## 1）后端生成链路ID

后端定义规则生成traceId，例如：雪花算法 + 时间戳 + 终端设备码

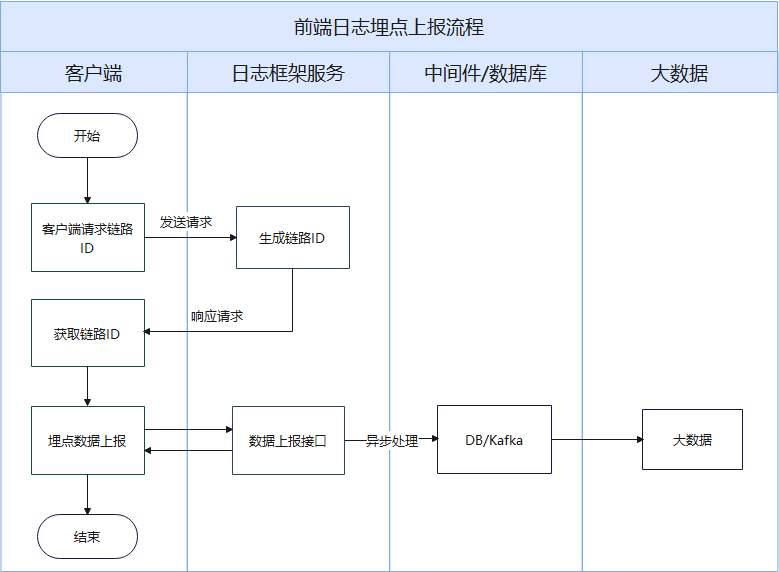
前端调用后台接口获取traceId。

## 2）前/后端埋点

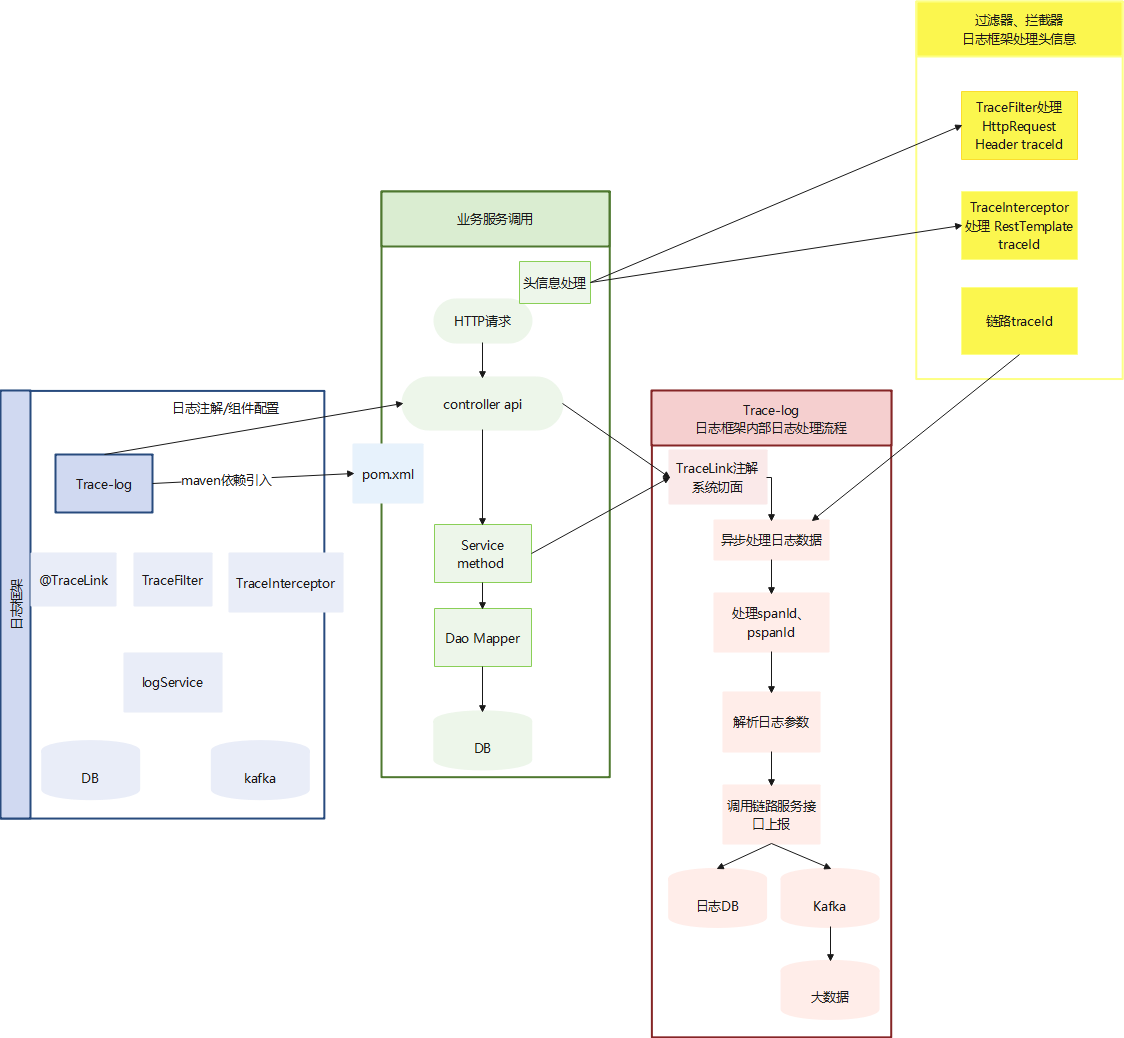
前端埋点数据，从后端接口获取traceId，放置头信息header，调用上报接口，传输数据至kafka、大数据。

上报参数数据后期需求定义。

前端流程



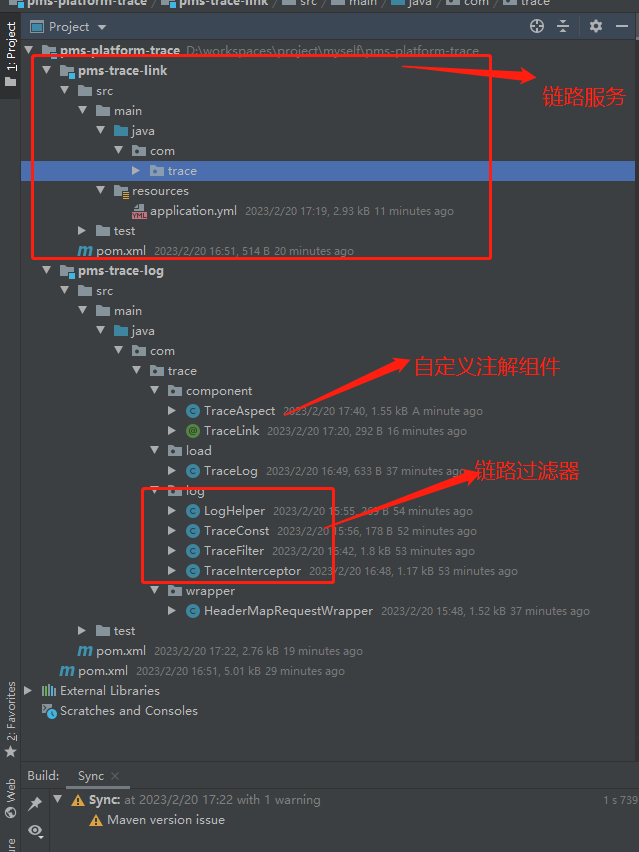
后端流程



## 3）工程划分

后端工程分为trace-log、trace-link两部分，trace-log用于客户端依赖sdk，trace-link用于实现链路数据上报。

、

r

## 4）过滤器处理

公共组件过滤器判断traceId，否则生成traceId，生成规则与前端保持一致。traceId放置header。

package com.yozo.common.log.trace;  
  
import com.yozo.common.core.wrapper.HeaderMapRequestWrapper;  
import lombok.extern.slf4j.Slf4j;  
import org.apache.commons.lang3.StringUtils;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.slf4j.MDC;  
import org.springframework.context.annotation.Configuration;  
  
import javax.servlet.\*;  
import javax.servlet.http.HttpServletRequest;  
import java.io.IOException;  
  
*/\*\*  
 \* 公共过滤器，处理traceId  
 \*/*@Slf4j  
@Configuration  
public class TraceFilter implements Filter {  
  
 private Logger logger = LoggerFactory.*getLogger*(TraceFilter.class);  
  
 @Override  
 public void init(FilterConfig filterConfig) throws ServletException {  
  
 }  
  
 @Override  
 public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) throws IOException, ServletException {  
 try {  
 HttpServletRequest request = (HttpServletRequest) servletRequest;  
 //从header获取traceId  
 String traceId = request.getHeader(TraceConst.*TRACE\_ID*);  
 traceId = StringUtils.*isEmpty*(traceId) ? LogHelper.*getTraceId*() : traceId;  
 //MDC put traceId  
 MDC.*put*(TraceConst.*TRACE\_ID*, traceId);  
  
 HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;  
 HeaderMapRequestWrapper requestWrapper = new HeaderMapRequestWrapper(httpRequest);  
 //如果请求中带有这个参数，则进行过滤加一个header头  
 requestWrapper.addHeader(TraceConst.*TRACE\_ID*, traceId);  
 filterChain.doFilter(requestWrapper, servletResponse);  
 } catch (Exception e) {  
 logger.error("trace过滤处理链路id异常", e);  
 } finally {  
 }  
 }  
  
 @Override  
 public void destroy() {  
  
 }  
}

## 5）RestTemplate调用

服务间调用采用feign方式或者restful restTemplate调用，服务公共组件拦截器或者过滤器判断header里设置traceId，向下游传输。

例如：

package com.yozo.common.log.trace;  
  
import feign.RequestInterceptor;  
import feign.RequestTemplate;  
import org.apache.commons.lang3.StringUtils;  
import org.slf4j.MDC;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.stereotype.Component;  
import org.springframework.web.context.request.RequestContextHolder;  
import org.springframework.web.context.request.ServletRequestAttributes;  
import org.springframework.web.servlet.HandlerInterceptor;  
  
*/\*\*  
 \* RestTemplate拦截器处理traceId  
 \*/*@Configuration  
public class TraceInterceptor implements RequestInterceptor, HandlerInterceptor {  
  
 @Override  
 public void apply(RequestTemplate requestTemplate) {  
 ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*();  
 if (null != attributes && null != attributes.getRequest()) {  
 //从header获取traceId  
 String traceId = attributes.getRequest().getHeader(TraceConst.*TRACE\_ID*);  
 traceId = StringUtils.*isEmpty*(traceId) ? LogHelper.*getTraceId*() : traceId;  
 //处理resttemplate header, 向下游传输  
 requestTemplate.header(TraceConst.*TRACE\_ID*, traceId);  
 }  
 }  
  
}

## 6）自定义注解

自定义注解标注需要埋点的接口方法。

例如：

方法名注解@TraceLink

package com.trace.component;  
  
import java.lang.annotation.\*;  
  
*/\*\*  
 \* 链路节点自定义注解  
 \*/*@Target(ElementType.*METHOD*)  
@Retention(RetentionPolicy.*RUNTIME*)  
@Documented  
public @interface TraceLink {  
  
 //服务-类-方法路径  
 String traceTarget() default "";  
}

package com.trace.component;  
  
import lombok.extern.slf4j.Slf4j;  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.Around;  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.stereotype.Component;  
  
*/\*\*  
 \* trace链路数据上报接口  
 \*/*@Aspect  
@Component  
@Slf4j  
public class TraceAspect {  
  
 */\*\*  
 \* 上报日志接口  
 \*/* @Value("${elb.trace.url}")  
 public String traceLinkUrl;  
  
 */\*\*  
 \* 定义切点 @Pointcut  
 \* 在注解的位置切入代码  
 \*/* @Pointcut("@annotation(com.trace.component.TraceLink)")  
 public void logPointCut() {  
 }  
  
 */\*\*  
 \* 环绕通知，获取接口调用前后入参及返回值数据  
 \** ***@param*** *proceedingJoinPoint  
 \** ***@return*** *\*/* @Around("logPointCut()")  
 public Object syncSysLog(ProceedingJoinPoint proceedingJoinPoint) {  
 Object proceed = null;  
 try {  
 Object[] args = proceedingJoinPoint.getArgs();  
 //前置通知 异步线程池处理日志上报接口 *TODO* proceed = proceedingJoinPoint.proceed(args);  
  
 //后置通知 异步线程池处理日志上报接口 *TODO* } catch (Throwable e) {  
 *log*.error(e.getMessage());  
 //异常通知 异步线程池处理日志上报接口 *TODO* } finally {  
 //*TODO* }  
  
 //返回 *TODO* return proceed;  
 }  
  
}

## 7）集中公共组件

合并feign拦截器、restful过滤器

package com.yozo.common.log.load;  
  
import com.yozo.common.log.trace.TraceFeignInterceptor;  
import com.yozo.common.log.trace.TraceFilter;  
import org.springframework.context.annotation.ImportSelector;  
import org.springframework.core.type.AnnotationMetadata;  
  
public class TraceLog implements ImportSelector {  
 @Override  
 public String[] selectImports(AnnotationMetadata importingClassMetadata) {  
 String[] traceLogImport = new String[] {  
 //trace 过滤器  
 TraceFilter.class.getName(),  
 //restTemplate拦截器  
 TraceInterceptor.class.getName()  
 };  
 return traceLogImport;  
 }  
}

## 8）数据存储

上报埋点数据推入DB或者kafka消费数据至大数据。

前期不具备kafka 大数据环境，可以入库DB，需要设计表。

## 9）统计计算

统计计算维度由大数据做实时计算或者离线计算。

前期DB只能通过定时任务、存储过程做准实时统计计算。

# 客户端import

客户端使用链路import组件、定义日志文件及格式

## pom依赖

<!--链路依赖-->

<dependency>

<groupId>com.pms</groupId>

<artifactId>pms-trace-log</artifactId>

</dependency>

## 公共组件Import

在启动类@Import

@Import({TraceLog.class})  
@SpringBootApplication(scanBasePackages = {"com.trace"})

配置文件：

## 3）日志文件规范定义

logback日志文件，整合MDC traceId打印日志输出。

采用logback控制日志文件的切分，技术人员根据各个服务具体情况进行调整修改，默认

每天生成一个文件夹，每小时生成一个日志文件，保存15天日志文件

日志文件格式：*/目录路径/服务名/年-月-日/服务名.年-月-日-时.log*



error级别日志，每天生成一个日志文件，保存30天

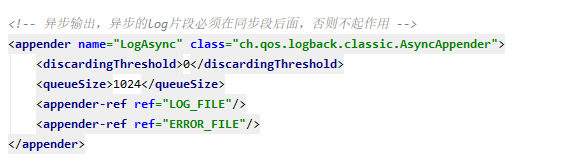


例如：

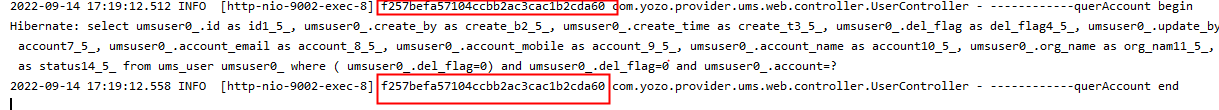
/logs/pms-framework-portal/2023-02-16/pms-framework-portal.2023-02-16\_20.0.log

/logs/pms-framework-portal/2023-02-16/pms-framework-portal.error.2023-02-16.log

日志异步配置



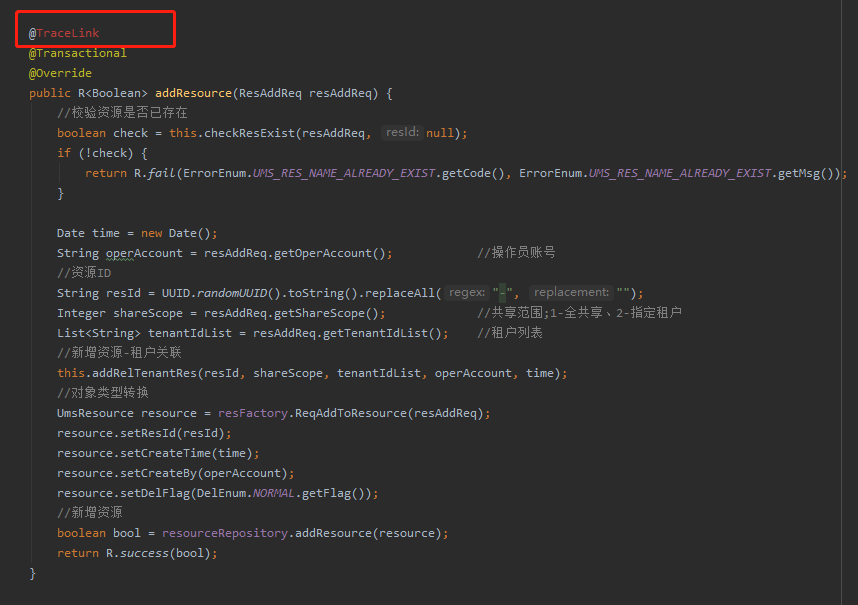
日志输出结果：



## 4）使用注解

在方法名上使用注解，以标识此方法在链路节点上。

例如：@TraceLink



注入：针对三方应用调入，可以在网关层生成链路ID，向下游传输。

# 3、日志规范

## 日志级别

现在分为5个日志级别，info、debug、warn、error、trace。

info：信息日志，包括业务日志、入参、出参。

error：错误日志，包括错误日志、异常日志、验证不通过等场景。以便问题定位查找分析。

debug：主要用于开发、运行日志。生产环境根据情况尽可能不打印。

warn：告警日志，用于不影响系统服务业务及功能作用，尽量少用。

trace：详尽日志，考虑日志量，不使用。

## 2）日志打印

A) 打印接口/方法的入参、出参日志。

B) 重要的业务逻辑、步骤、条件必须打印日志。

C) 日志的粗细颗粒度，根据数据报文的大小打印必要的日志数据。

D) 列表返参根据情况是否需要全部打印。

## 3）日志格式

日志格式：时间戳 + 日志级别 + 线程栈 + trace日志链路Id + 全路径类

%d{yyyy-MM-dd HH:mm:ss.SSS} %-5level [%thread] %X{traceId} %logger - %msg%n

## 4）条件日志

If else、switch等条件判断，条件复杂、每个分支首行都尽量打印日志。

## 5）日志框架SLF4J

建议使用日志框架SLF4J中的API

例如：

采用lombok注入方式，@Slf4j

手动创建Logger方式：

private static final Logger logger = LoggerFactory.getLogger(XXX.class);

## 6）占位打印。

建议使用参数占位{}，而不是用+拼接。

例如：

***log***.info(**"这里是网关,客户端请求路径:{},上下文traceId:{}"** ,exchange.getRequest().getURI(), finalTraceId);

## 7）异步打印

使用异步的方式来输出日志。

在logback.xml配置

例如：

<**appender name="LogAsync" class="ch.qos.logback.classic.AsyncAppender"**>  
 <**discardingThreshold**>0</**discardingThreshold**>  
 <**queueSize**>1024</**queueSize**>  
 <**appender-ref ref="LOG\_FILE"**/>  
 <**appender-ref ref="ERROR\_FILE"**/>  
</**appender**>

## 8）日志数据

核心功能模块，建议打印较完整的日志

## 9）对象反射

不要用反射工具类解析打印对象，反射机制对性能消耗过大。

例如：打印对象采用JSON.*toJSONString*(httpEntity)

## 10）低级别日志

尽量不要打开debug级别日志，debug会把底层运行日志全部打印，这类对业务系统无效的日志暂不需要打印，会大大占用日志文件空间。

业务日志用info，异常错误用error。

## 11）异常日志

错误异常日志处理。

异常不建议使用e.printStackTrace()，手动打印异常错误日志。

例如：

try{ // 业务代码处理}catch(Exception e){ log.error("业务处理异常",e);

//不建议用e.printStackTrace();}

## 12）异常抛出

尽可能把原始异常抛出来。

异常日志抛出execption，返回可以用自定义异常错误码

反例：

try{ // 业务代码处理}catch(Exception e){

e.printStackTrace();}

if (XXX == null) {  
 log.info("XXX”);  
 throw new ServiceException(ErrorEnum.XXX);  
}

正例：

try{ // 业务代码处理}catch(Exception e){ log.error("业务处理异常",e);

}

if (XXX == null) {  
 log.info("XXX”);  
 return ErrorEnum.XXXX;}

业务逻辑判断，不满足条件不建议手动自定义抛异常给上层，因为这不属于程序代码执行错误，会在日志中打印出异常日志，并且通过自定义异常全局捕获会消耗一定的性能。

不满足条件可以返回结果码或者布尔型。