关于Spring的log

Spring框架默认使用Logback用于日志。除了Spring自己框架使用了日志以外，我们在业务代码里也有使用log的需求，因此时常需要获得一个log实例，获得一个log实例的方式很多，根据使用的不同的包，有很多种方式，举例如下：

```
@CommonsLog
private static final org.apache.commons.logging.Log log =
org.apache.commons.logging.LogFactory.getLog(LogExample.class);
@Flogger
private static final com.google.common.flogger.FluentLogger log =
com.google.common.flogger.FluentLogger.forEnclosingClass();
@JBossLog
private static final org.jboss.logging.Logger log =
org.jboss.logging.Logger.getLogger(LogExample.class);
@Log
private static final java.util.logging.Logger log =
java.util.logging.Logger.getLogger(LogExample.class.getName());
@Log4j
private static final org.apache.log4j.Logger log =
org.apache.log4j.Logger.getLogger(LogExample.class);
@Log4j2
private static final org.apache.logging.log4j.Logger log =
org.apache.logging.log4j.LogManager.getLogger(LogExample.class);
@Slf4j
private static final org.slf4j.Logger log =
org.slf4j.LoggerFactory.getLogger(LogExample.class);
@XSlf4j
private static final org.slf4j.ext.XLogger log =
org.slf4j.ext.XLoggerFactory.getXLogger(LogExample.class);
```

这些不同的log提供的功能都大同小异，对我们来说，只是格式化有点差异而已。

值得一提的是，lombok这个插件为我们提供了直接生成log的便利，当我们在类上标注上面代码框中的注解时，会自动为我们生成一个log实例供我们使用，不同的注解生成对应库的log，如：

```
@CommonsLog
public class LogExample {

}
//就相当于：
public class LogExample {
                private static final org.apache.commons.logging.Log log
      = org.apache.commons.logging.LogFactory.getLog(LogExample.class);
}
```

一般系统出了bug，我们极大依赖于日志，但我在log的使用过程中，发现有这么几个问题：

1. log过少

   即该打log的地方没有打log，导致完全不知道系统中发生了什么，只能根据异常去倒查，异常实际上丢失了一些信息，比如调用参数信息等，debug很困难。

2. log过多

   像我做的一些模块，之前没有经验，喜欢把参数信息和返回值都输出，有时候就会日志洪泛，像我们这块儿，返回值有时候就是一些文章内容什么的，一下刷十几屏，淹没了有效信息。

3. log在平时看的时候过多，在debug时过少

   这是上两个的结合体了，有时候debug需要的信息，日常没出错时不需要，但我们也没法断定说这什么时候出错，什么时候不出错。所以有些地方的日志就变成了不出错时太多，出错时太少。

4. 有些框架日志明显不需要，比如我们连接eureka的时候，每5分钟发一次心跳包，info级别日志，几天不看，这个心跳包日志就完全刷屏了，看着就很烦，如下图：

```
2019-04-29 16:24:07.337  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
2019-04-29 16:29:07.337  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
2019-04-29 16:34:07.338  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
2019-04-29 16:39:07.339  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
2019-04-29 16:44:07.339  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
2019-04-29 16:49:07.340  INFO 1 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver      : Resolving eureka endpoints
via configuration
```

日志输出的过多或者不够，会造成出现问题就要改代码、重新打包上传、再部署、再运行，这样的一个步骤，debug之后又要原路改回去。这样的事情我干过不少次。

所以引出了最近的几点思考，试简单总结如下：

日志分级

一般来讲，日志分为5级，trace、debug、info、warn、error，从下到上依次严重，我们在代码里对应的一般就是`log.trace()`,`log.debug()`这样方法。

我们把debug需要的那部分信息通过`log.debug()`输出，正常日志通过info输出，这样就把不同级别的日志区分开了，当我们需要debug时，代码不需要改动，只需要在参数中改变日志的level，就能看到debug日志，修改完之后把debug日志关闭即可。这样就避免了频繁改变文件打包。

Spring的log日志级别设置

spring默认使用logback记录日志，日志级别通过`logging.level.xxx`参数设置。其中`logging.level.`是通用前缀，`xxx`是具体包名。如：

```
logging.level.com.mycompany.tahiti.attachment.aop.ServiceLogger: debug
```

是指`com.mycompany.tahiti.attachment.aop.ServiceLogger`这个包里生成的log使用`debug`级别。而其他的log仍然使用默认的级别。如果要更改默认级别，可以使用以下命令修改：

```
logging.level.root=debug
```

但在Spring中特别不建议这么做，因为我们Spring框架中也会同样输出debug日志，于是日志被各种debug信息占满，完全没法看了。一般还是针对需要debug的那几个包设置debug级别就足够了。

这个参数可以直接加在`application.yaml`文件中，也可以加在`application.apprities`文件里，当我们使用k8s时，还可以直接在k8s中增加args，如下图：

```
spec:
        template:
                spec:
                        containers:
        args: ["--logging.level.root=info","--arg2=value2"]
```

所以我们上面提出的第四个问题也解决了，我们先找到框架中输出这个心跳包的类，然后直接将其日志级别设置为warn。

```
logging.level.com.netflix.discovery.shared.resolver.aws.ConfigClusterResolv
er: warn
```

这样这个包的info信息就不生成了。

如果我们自己的log也想使用这个机制，建议使用

```
@lombok.extern.apachecommons.CommonsLog
public class MyClass{ }
```

或者直接手写一个log：

```
private static final org.apache.commons.logging.Log log =
org.apache.commons.logging.LogFactory.getLog(LogExample.class);
```

其他的log包可能有的不能和spring的配置协作，未验证，因此最好不用。如果再观察一下配置文件的解析，发现配置文件是由:

```
org.springframework.boot.logging.LoggingApplicationListener
```

这个包来解析的。先是解析配置文件，建立了一个class的fullname到level的map，将这个map注册到LogFactory中，我们上面的log是通过LogFactory静态方法取得的，这样就可以确保获得的log有正确的level。这可以说是factory的经典应用了。

log exception

另外，当我们记录exception时，有时候会写这样的代码：

```
try{
        dosth();
} catch (Exception e){
        log.error("sth bad happened");
        e.printStack();
}
```

这样造成了一个隐含问题，e.printStack()是将数据输出到stdout中，当log的输出源也是stdout时，这样没有问题，当log的输出源被改为文件后，这两个输出就不在一起了。因此，建议使用

```
log.error("sth bad happened", e);
```

log与AOP

有时候，我们需要记录系统进行了哪些操作，比如想记录每一次对controller的调用，那么，我们要在每个方法里加一个log.info("xxx方法被调用")，这样显然很麻烦，也影响代码的整洁美观。利用SpringAOP，我们可以建立一个这样的类：

```
@Aspect
@Component
@CommonsLog
public class ServiceLogger {

    @Pointcut("execution(*
com.mycompany.tahiti.attachment.service.impl.AttachmentServiceImpl2.*
(..))")
    public void action() {
    }

    @Around("action()")
    public Object logRequest(ProceedingJoinPoint joinPoint) throws
Throwable {
```

```
        log.info(String.format("--------%s方法被调用",
joinPoint.getSignature()));
        if (log.isDebugEnabled()) {
            log.debug(String.format("共收到%d个参数%s",
joinPoint.getArgs().length,
                    joinPoint.getArgs().length == 0 ? "" : "，以下是详细参数信
息："));
            for (int i = 0; i != joinPoint.getArgs().length; ++i) {
                Object arg = joinPoint.getArgs()[i];
                if (Collection.class.isAssignableFrom(arg.getClass())) {
                    log.debug(String.format("参数%d是容器类，共有%d个元素", i,
((Collection) arg).size()));
                    log.debug(String.format("        %s", ((Collection<?>)
arg).stream().map(Object::toString).collect(Collectors.joining(","))));
                } else if (Map.class.isAssignableFrom(arg.getClass())) {
                    log.debug(String.format("参数%d是Map类，共有%d个元素", i,
((Map) arg).size()));
                    StringBuilder sb = new StringBuilder();
                    ((Map<?, ?>) arg).forEach((k, v) ->
sb.append("key=").append(k).append("value=").append(v).append("; "));
                    log.debug(String.format("        %s", sb.toString()));
                } else {
                    log.debug(String.format("参数%d为：%s", i,
arg.toString()));
                }
            }
        }
        Object retValue = joinPoint.proceed(joinPoint.getArgs());
        log.info(String.format("------%s方法返回",
joinPoint.getSignature()));

        if (log.isDebugEnabled()) {
            if (Collection.class.isAssignableFrom(retValue.getClass())) {
                log.debug(String.format("返回值是容器类，共有%d个元素",
((Collection) retValue).size()));
                log.debug(String.format("        %s", ((Collection<?>)
retValue).stream().map(Object::toString).collect(Collectors.joining(","))))
;
            } else if (Map.class.isAssignableFrom(retValue.getClass())) {
                log.debug(String.format("返回值是Map类，共有%d个元素", ((Map)
retValue).size()));
                StringBuilder sb = new StringBuilder();
                ((Map<?, ?>) retValue).forEach((k, v) ->
sb.append("key=").append(k).append("value=").append(v).append("; "));
                log.debug(String.format("        %s", sb.toString()));
            } else {
                log.debug(String.format("返回值为：%s",
retValue.toString()));
            }
        }

        return retValue;
    }
```

```
}
```

这样可以保证每一个方法调用时都能够产生日志，不用每次手写日志。

Log与时区问题

我们经常发现我们的日志中的时间和北京时间相差八小时，大多数时候这一点我们自己看的时候在脑子里做个时间转换就可以了，但这样还是容易让人疑惑和烦躁，因此我们需要某种方式来统一时间。

在linux中，当地时间由`/etc/localtime`确定的，而这个文件实际上是一个软链，链接到`/var/db/timezone/zoneinfo/Asia/Shanghai`。因此，如果要改变linux的标准时间，只需要改变这个软链接的指向即可。

我们现在主要用Docker进行部署，一般来讲，Python语言的项目打的包会有一个完整的linux环境，因为它依赖的东西太多了，`pip install`甚至还要求docker镜像里有完整的开发环境，所以一个包经常有1.2G以上。但java的docker就精简很多，甚至最常用的ls命令都没有打包到`java:8`这个基础镜像里，因此一个包经常只有100M左右。我们分别以这两种语言的docker包为例，实例如何正确显示当地时间。

1. Java

   只需要在Dockerfile中添加一个环境变量，因为java的极简包里什么都没有，也不通过`/etc/localtime`来确定时间。

   ```
   FROM openjdk:8-alpine
   WORKDIR /app
   ENV TZ Asia/Shanghai                    # 就是这一句
   COPY target/ROOT.jar .
   ENTRYPOINT ["java", "-XX:+UnlockExperimentalVMOptions", "-
   XX:+UseCGroupMemoryLimitForHeap", "-jar", "ROOT.jar"]
   ```

2. python项目

   在Dockerfile中直接更改`/etc/localtime`的软链。

   ```
   FROM rappdw/docker-java-python:latest

   RUN mkdir -p ~/.pip
   RUN echo "[global]\n\
   trusted-host =  mirrors.aliyun.com\n\
   index-url = http://mirrors.aliyun.com/pypi/simple" > ~/.pip/pip.conf

   COPY requirements.txt ./
   RUN pip3 install --no-cache-dir -r requirements.txt

   COPY ./ ./

   RUN ln -fs /usr/share/zoneinfo/Asia/Shanghai /etc/localtime      # 主要
   ```

```
是这一句

RUN chmod a+x run.sh

ENTRYPOINT ["/run.sh"]
```