日志

网上有个段子：某大厂招了一个程序员，入职不到两天被辞退，原因竟然是写代码不记录日志

> *在应用当中，我们少不了要记录日志，记录日志的目的有两个：*
>
> 1. *用于线上错误排查*
> 2. *分析日志（埋点日志）*

日志发展史

1. 阶段一

> *2001年以前，Java是没有日志库的，打印日志全凭`System.out`和`System.err`*

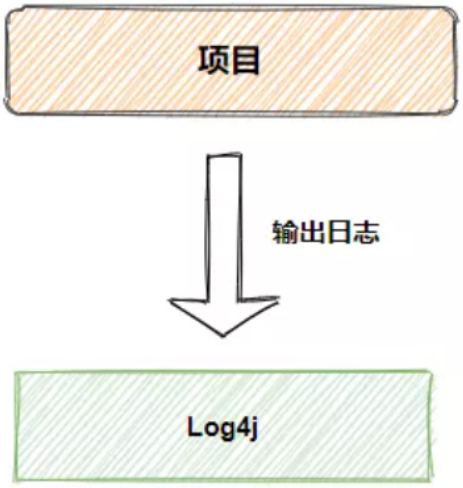
缺点：

1. **产生大量的IO操作 同时在生产环境中 无法合理的控制是否需要输出**
2. **输出的内容不能保存到文件**
3. **只打印在控制台，打印完就过去了，也就是说除非你一直盯着程序跑**
4. **无法定制化，且日志粒度不够细**

2. 阶段二

> *2001年，一个叫`Ceki Gülcü`的大佬搞了一个日志库`log4j`，后来`Log4j`成为`Apache`项目，`Ceki`也加入`Apache`组织*

Apache还曾经建议Sun引入Log4j到Java的标准库中，但Sun拒绝了



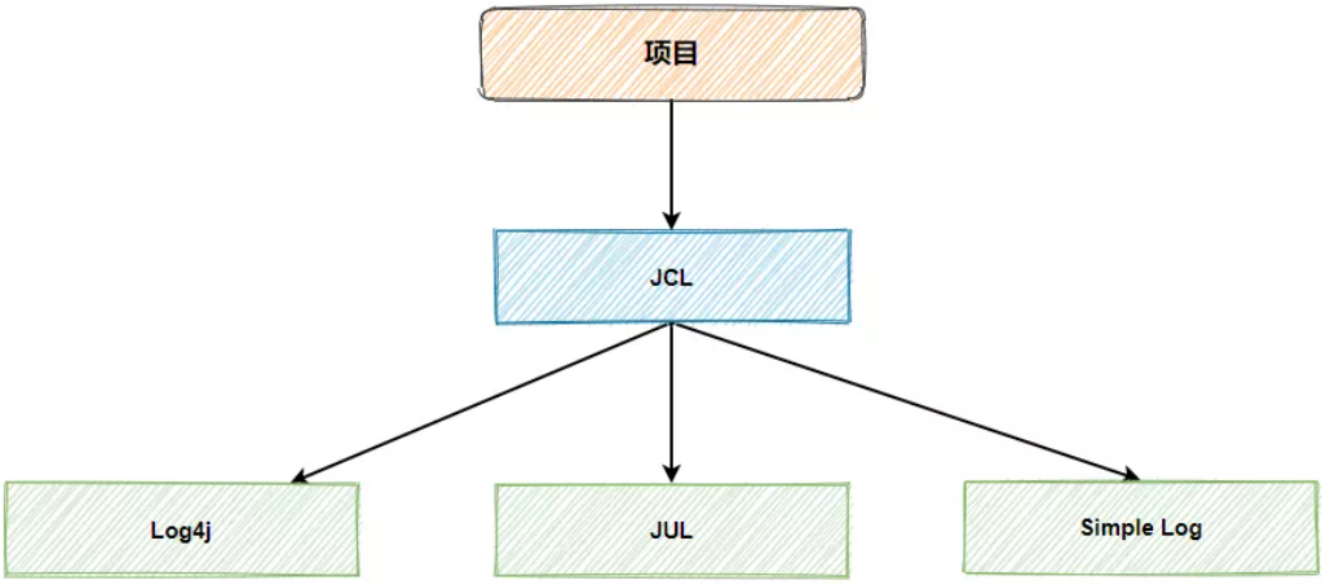
3. 阶段三

> *Sun有自己的小心思，2002年2月JDK1.4发布，Sun推出了自己的日志标准库JUL（Java Util Logging），其实是照着Log4j抄的，而且还没抄好，还是在JDK1.5以后性能和可用性才有所提升。*

由于Log4j比JUL好用，并且成熟，所以Log4j在选择上占据了一定的优势。

4. 阶段四

> *2002年8月Apache推出了JCL（Jakarta Commons Logging），也就是日志抽象层，支持运行时动态加载日志组件的实现，当然也提供一个默认实现Simple Log（**在ClassLoader中进行查找，如果能找到Log4j则默认使用log4j实现，如果没有则使用JUL 实现，再没有则使用JCL内部提供的Simple Log实现**）。*



但是JCL有三个缺点：

1. **效率较低**
2. **容易引发混乱**
3. **使用了自定义ClassLoader的程序中，使用JCL会引发内存泄露**

5. 阶段五

> *2006年巨佬Ceki（Log4j的作者）因为一些原因离开了Apache组织，之后Ceki觉得JCL不好用，自己撸了一套新的日志标准接口规范Slf4j（Simple Logging Facade for Java），也可以称为日志门面，很明显Slf4j是对标JCL，后面也证明了Slf4j比JCL更优秀。*

巨佬Ceki提供了一系列的桥接包来帮助Slf4j接口与其他日志库建立关系，这种方式称为桥接设计模式。

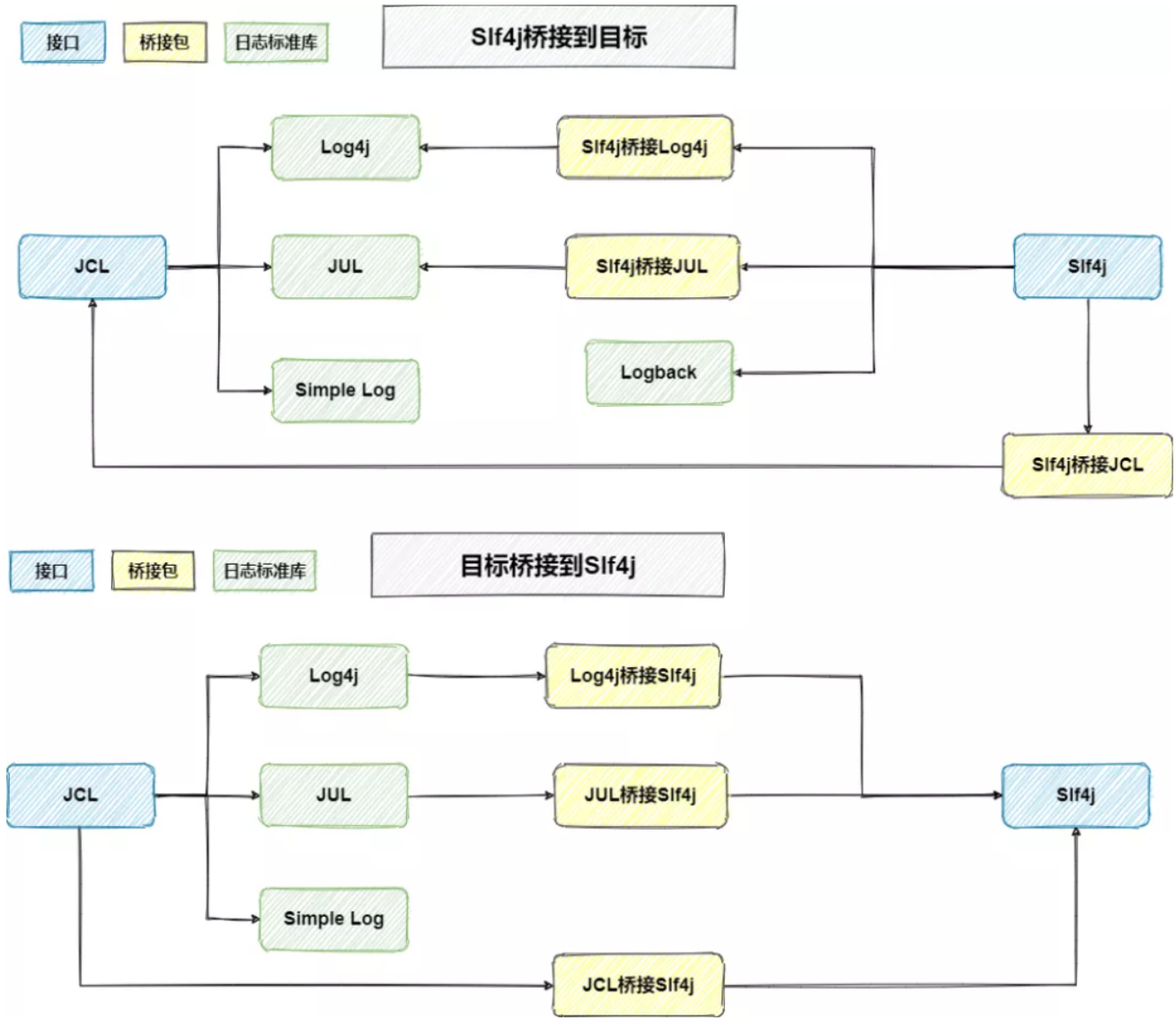
代码使用Slf4j接口，就可以实现日志的统一标准化，后续如果想要更换日志实现，只需要引入Slf4j与相关的桥接包，再引入具体的日志标准库即可。

6. 阶段六

> *Ceki巨佬觉得市场上的日志标准库都是间接实现Slf4j接口，也就是说每次都需要配合桥接包，因此在2006年，Ceki巨佬基于Slf4j接口撸出了Logback日志标准库，做为Slf4j接口的默认实现，Logback也十分给力，在功能完整度和性能上超越了所有已有的日志标准库。*

根本原因还在于，随着用户体量的提升，Log4j无法满足高性能的要求，成为应用的性能瓶颈

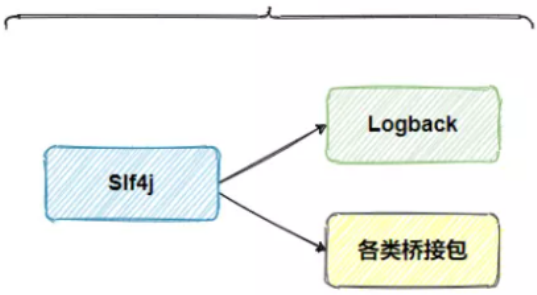
目前Java日志体系关系图如下

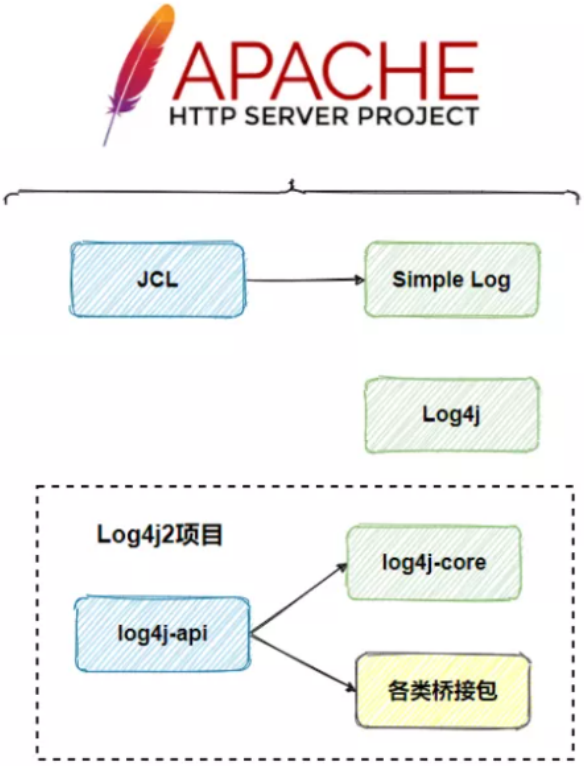


7. 阶段七

> *2012年，Apache直接推出新项目Log4j2（**不兼容Log4j**），Log4j2全面借鉴Slf4j+Logback。*

Log4j2不仅仅具有Logback的所有特性，还做了分离设计，分为log4j-api和log4j-core，log4j-api是日志接口，log4j-core是日志标准库，并且Apache也为Log4j2提供了各种桥接包。





> *虽然Log4j2有明显的抄袭嫌疑，但是毕竟是最新的日志库，汲取了logback优秀设计的同时，还解决了一些问题，性能有了极大的提升，官方测试是18倍，所以我们选择使用Log4j2来作为日志库*

1. Log4j2

> *SpringBoot使用Log4j2：*
>
> 1. *SpringBoot默认集成的是logback，所以使用Log4j2需要先排除spring-boot-starter-logging*
> 2. *Log4j2需要使用异步记录日志（18倍性能的由来）*

目前市面上最主流的日志门面就是SLF4J，虽然Log4j2 也是日志门面，因为它的日志实现功能非常强大，性能优越。所以大家一般还是将 Log4j2 看作是日志的实现，**Slf4j + Log4j2 应该是未来的大势所趋**。

1.1 集成Log4j2

在mszlu-xt-parent中，添加对应的依赖管理：

注意：这里必须要加版本号

```
 <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
                <version>2.5.0</version>
                <exclusions>
                    <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-
logging</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter</artifactId>
                <version>2.5.0</version>
                <exclusions>
                    <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-
logging</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
 <!--异步日志依赖 -->
            <dependency>
                <groupId>com.lmax</groupId>
                <artifactId>disruptor</artifactId>
                <version>3.3.4</version>
            </dependency>
```

由于common每个模块都需要导入，在common中添加log4j2的依赖：

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-log4j2</artifactId>
        </dependency>
   <!--异步日志依赖 -->
        <dependency>
            <groupId>com.lmax</groupId>
            <artifactId>disruptor</artifactId>
        </dependency>
```

1.1.1 添加 log4j2.xml 配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
  日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE
> ALL
    status="warn" 日志框架本身的输出日志级别，可以修改为debug
    monitorInterval="5" 自动加载配置文件的间隔时间，不低于 5秒；生产环境中修改配置
文件，是热更新，无需重启应用
 -->
<!--
  自定义命名格式:
  %d: 发生时间，%d{yyyy-MM-dd HH:mm:ss,SSS}，输出类似：2020-02-20
22:10:28,921
  %F: 输出所在的类文件名
  %t: 线程名称
  %p: 日志级别
  %c: 日志消息所在类名
  %m: 消息内容
  %M: 输出所在函数名
  %x: 输出和当前线程相关联的NDC(嵌套诊断环境),尤其用到像java servlets这样的多客户
多线程的应用中。
  %l: 执行的函数名(类名称:行号)
com.core.LogHelper.aroundService(LogHelper.java:32)
  %n: 换行
  %i: 从1开始自增数字
  %-5level: 输出日志级别，-5表示左对齐并且固定输出5个字符，如果不足在右边补0
-->
<configuration status="WARN" monitorInterval="5">
    <!--
        集中配置属性进行管理
        使用时通过:${name}
    -->
    <properties>
```

```
        <property name="LOG_HOME">D:/logs/xt</property>
        <property name="PATTERN">%d %highlight{%-5level}{ERROR=Bright
RED, WARN=Bright Yellow, INFO=Bright Green, DEBUG=Bright Cyan,
TRACE=Bright White} %style{[%t]}{bright,magenta} %style{%c{1.}.%M(%L)}
{cyan}: %msg%n</property>
        <property name="FILE_PATTERN">%d %-5level [%t] %c{1.}.%M(%L):
%m%n</property>
    </properties>

    <!-- 日志处理 -->
    <Appenders>
        <!-- 控制台输出 appender，SYSTEM_OUT输出黑色，SYSTEM_ERR输出红色 在vm
配置-Dlog4j.skipJansi=false可开启彩色日志-->
        <Console name="Console" target="SYSTEM_OUT">
            <!-- 设置控制台只输出INFO及以上级别的信息(onMatch),其他的直接拒绝
(onMismatch) -->
            <ThresholdFilter level="INFO" onMatch="ACCEPT"
onMismatch="DENY"/>
            <PatternLayout pattern="${PATTERN}" />
        </Console>

        <!-- 日志文件输出 appender 所有的日志信息会打印到此文件中，append=false
每次启动程序会自动清空 -->
        <File name="file" fileName="${LOG_HOME}/xt.log" append="true">
            <!-- 设置控制台只输出INFO及以上级别的信息(onMatch),其他的直接拒绝
(onMismatch) -->
            <ThresholdFilter level="INFO" onMatch="ACCEPT"
onMismatch="DENY"/>
            <PatternLayout pattern="${FILE_PATTERN}" />
        </File>

        <Async name="Async">
            <AppenderRef ref="file" />
        </Async>

        <!-- 日志文件输出 appender 业务日志 一般打印domain中的日志 -->c
        <!--RollingFile 滚动文件，达到指定条件 生成新的文件进行记录-->
        <RollingFile name="business"
fileName="${LOG_HOME}/business/business.log"
filePattern="${LOG_HOME}/business/business_%d{yyyy-MM-dd HH}.log">
            <PatternLayout pattern="${FILE_PATTERN}" />

            <Policies>
                <TimeBasedTriggeringPolicy/>
                <!-- 按大小划分 -->
```

```
                <SizeBasedTriggeringPolicy size="100MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7
个文件，超过该数量，会滚动删除前面的记录 -->
            <DefaultRolloverStrategy max="24"/>
        </RollingFile>

        <Async name="Async-Business">
            <AppenderRef ref="business" />
        </Async>

        <RollingFile name="service"
fileName="${LOG_HOME}/service/service.log"
filePattern="${LOG_HOME}/service/service_%d{yyyy-MM-dd HH}.log">
            <PatternLayout pattern="${FILE_PATTERN}" />

            <Policies>
                <TimeBasedTriggeringPolicy/>
                <!-- 按大小划分 -->
                <SizeBasedTriggeringPolicy size="100MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7
个文件，超过该数量，会滚动删除前面的记录 -->
            <DefaultRolloverStrategy max="24"/>
        </RollingFile>

        <Async name="Async-Service">
            <AppenderRef ref="service" />
        </Async>

        <RollingFile name="controller"
fileName="${LOG_HOME}/controller/controller.log"
filePattern="${LOG_HOME}/controller/controller_%d{yyyy-MM-dd HH}.log">
            <PatternLayout pattern="${FILE_PATTERN}" />

            <Policies>
                <TimeBasedTriggeringPolicy/>
                <!-- 按大小划分 -->
                <SizeBasedTriggeringPolicy size="100MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7
个文件，超过该数量，会滚动删除前面的记录 -->
            <DefaultRolloverStrategy max="24"/>
        </RollingFile>
```

```
        <Async name="Async-Controller">
            <AppenderRef ref="controller" />
        </Async>


        <RollingFile name="common"
fileName="${LOG_HOME}/common/common.log"
filePattern="${LOG_HOME}/common/common_%d{yyyy-MM-dd HH}.log">
            <PatternLayout pattern="${FILE_PATTERN}" />

            <Policies>
                <TimeBasedTriggeringPolicy/>
                <!-- 按大小划分 -->
                <SizeBasedTriggeringPolicy size="100MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7
个文件，超过该数量，会滚动删除前面的记录 -->
            <DefaultRolloverStrategy max="24"/>
        </RollingFile>

        <Async name="Async-Common">
            <AppenderRef ref="common" />
        </Async>

        <RollingFile name="error" fileName="${LOG_HOME}/error/error.log"
filePattern="${LOG_HOME}/error/error_%d{yyyy-MM-dd}.log">
            <PatternLayout pattern="${FILE_PATTERN}" />
            <Filters>
                <ThresholdFilter level="ERROR" onMatch="ACCEPT"
onMismatch="DENY"/>
            </Filters>
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <!-- 按大小划分 -->
                <SizeBasedTriggeringPolicy size="100MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7
个文件，超过该数量，会滚动删除前面的记录 -->
            <DefaultRolloverStrategy max="24"/>
        </RollingFile>

        <Async name="Async-Error">
            <AppenderRef ref="error" />
        </Async>
    </Appenders>
```

```
    <!-- logger 定义 -->
    <Loggers>
        <!-- 自定义 logger 对象
            includeLocation="false" 关闭日志记录的行号信息，开启的话会严重影响异步输出的性能
            additivity="false" 不再继承 rootlogger对象
         -->
        <AsyncLogger name="com.mszlu" level="INFO" includeLocation="false" additivity="false">
            <!-- 指定日志使用的处理器 -->
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <AsyncLogger name="com.mszlu.xt.web.api" level="INFO" includeLocation="false" additivity="false">
            <AppenderRef ref="Async-Controller" />
            <AppenderRef ref="Async" />
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <AsyncLogger name="com.mszlu.xt.web.handler" level="INFO" includeLocation="false" additivity="false">
            <AppenderRef ref="Async-Controller" />
            <AppenderRef ref="Async" />
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <AsyncLogger name="com.mszlu.xt.web.service" level="INFO" includeLocation="false" additivity="false">
            <AppenderRef ref="Async-Service" />
            <AppenderRef ref="Async" />
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <AsyncLogger name="com.mszlu.xt.web.domain" level="INFO" includeLocation="false" additivity="false">
            <AppenderRef ref="Async-Business" />
            <AppenderRef ref="Async" />
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <AsyncLogger name="com.mszlu.common" level="INFO" includeLocation="false" additivity="false">
            <AppenderRef ref="Async-Common" />
```

```
            <AppenderRef ref="Async" />
            <AppenderRef ref="Console" />
            <AppenderRef ref="Async-Error" />
        </AsyncLogger>
        <!-- 使用 rootLogger 配置 日志级别 level="trace" -->
        <Root level="trace">
            <!-- 指定日志使用的处理器 -->
            <AppenderRef ref="Console" />
            <!-- 使用异步 appender -->
            <AppenderRef ref="Async" />

            <AppenderRef ref="Async-Error" />
        </Root>
    </Loggers>
</configuration>
```

1.1.2 测试

> *在代码中，输入info，error等错误日志，进行测试*

2. 埋点日志

> *埋点日志通常用于进行数据分析，比如记录登录日志，关键业务的操作日志，用于分析用户的活跃度，留存率，停留时间等用于支撑运营的数据*

2.1 技术选型

很明显，埋点数据非常多，并且需要事后进行统计分析，所以需要选择一款数据来存储，这里我们选择mongodb，一是好查询（类关系型数据库），二是存储的数据量级大。

再分析，由于埋点数据和业务完全无关，不能因为埋点日志的问题，影响核心的业务逻辑，通俗点就是说，当埋点日志业务挂掉了，我的应用还能用不能受影响，所以我们在记录埋点日志的时候，使用队列MQ，这里我们选用RocketMQ

2.2 Mongo介绍

在线教程：https://www.runoob.com/mongodb/mongodb-tutorial.html

应用特征	YES / NO
应用不需要事务及复杂 join 支持	必须 Yes
新应用，需求会变，数据模型无法确定，想快速迭代开发	?
应用需要2000-3000以上的读写QPS（更高也可以）	?
应用需要TB甚至 PB 级别数据存储	?
应用发展迅速，需要能快速水平扩展	?
应用要求存储的数据不丢失	?
应用需要99.999%高可用	?
应用需要大量的地理位置查询、文本查询	?

如果上述有1个 Yes，可以考虑 MongoDB，2个及以上的 Yes，选择 MongoDB 绝不会后悔。

2.3 安装

2.3.1 Docker介绍

Docker 属于 Linux 容器的一种封装，提供简单易用的容器使用接口。它是目前最流行的 Linux 容器解决方案。

Docker 将应用程序与该程序的依赖，打包在一个文件里面。运行这个文件，就会生成一个虚拟容器。程序在这个虚拟容器里运行，就好像在真实的物理机上运行一样。有了 Docker，就不用担心环境问题。

总体来说，Docker 的接口相当简单，用户可以方便地创建和使用容器，把自己的应用放入容器。容器还可以进行版本管理、复制、分享、修改，就像管理普通的代码一样。

2.3.2 Docker 的用途

Docker 的主要用途，目前有三大类。

（1）提供一次性的环境。比如，本地测试他人的软件、持续集成的时候提供单元测试和构建的环境。

（2）提供弹性的云服务。因为 Docker 容器可以随开随关，很适合动态扩容和缩容。

（3）组建微服务架构。通过多个容器，一台机器可以跑多个服务，因此在本机就可以模拟出微服务架构。

2.3.3 Docker 的安装

Docker 是一个开源的商业产品，有两个版本：社区版（Community Edition，缩写为 CE）和企业版（Enterprise Edition，缩写为 EE）。企业版包含了一些收费服务，个人开发者一般用不到。

我们安装的Docker CE。

按照以下的步骤 安装docker

```
# 1、yum 包更新到最新
yum update
# 2、安装需要的软件包， yum-util 提供yum-config-manager功能，另外两个是
devicemapper驱动依赖的
yum install -y yum-utils device-mapper-persistent-data lvm2
# 3、 设置yum源
yum-config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
# 4、 安装docker，出现输入的界面都按 y
yum install -y docker-ce
# 5、 查看docker版本，验证是否验证成功
docker -v
#启动docker
/bin/systemctl start docker.service
```

2.3.4 安装Mongo

docker中有重要的三个概念：

1. 仓库

   使用docker安装软件，需要有下载软件的地址，这个地址就是仓库，类似maven的仓库概念,https://hub.docker.com/

2. 镜像

   docker安装的软件包，比如redis:5.0.5镜像，就是redis的5.0.5的安装包

3. 容器

   拉取镜像后，需要安装后才能使用，安装完成后，才能使用

> *安装完容器后，需要运行才能使用，容器运行起来后，一个容器就相当于一个操作系统，有独立的ip，容器和容器之间需要通过ip才能互相访问，同时必须在一个网段内。*
>
> *docker所在的操作系统或者机器我们称为宿主机，宿主机和docker容器之间无法直接互通，需要通过数据卷，数据卷就是宿主机的一个文件目录和容器内一个目录的映射，操作宿主机的一个文件目录就相当于操作容器内的目录。*
>
> *外部如果需要访问容器内安装的软件，需要通过宿主机，所以容器和宿主机之间需要做端口映射，比如 -p 80:8080 代表如果访问某个容器内部的8080端口，需要访问宿主机的80端口才行*

安装mongo的命令：

```
# 拉取镜像
docker pull mongo
# 创建数据卷目录，用于映射容器内的数据存储目录
mkdir -p /docker/mongo/db
# 运行容器 run是创建并运行 create只创建 运行需要docker start 容器名称
docker run -id -p 27017:27017 -v /docker/mongo/db:/data/db --name mongo
mongo
# 查看容器是否允许
docker ps
# 进入容器
docker exec -it mongo /bin/bash
# 进入mongo
root@b606ac4f949a:/# mongo
```

2.4 sso集成mongo

在sso-dao中导入mongo的相关依赖：

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
```

在sso-api中加入mongo的配置

```
# mongo的配置
spring.data.mongodb.uri=mongodb://192.168.200.100:27017/xt
```

2.4.1 测试mongo

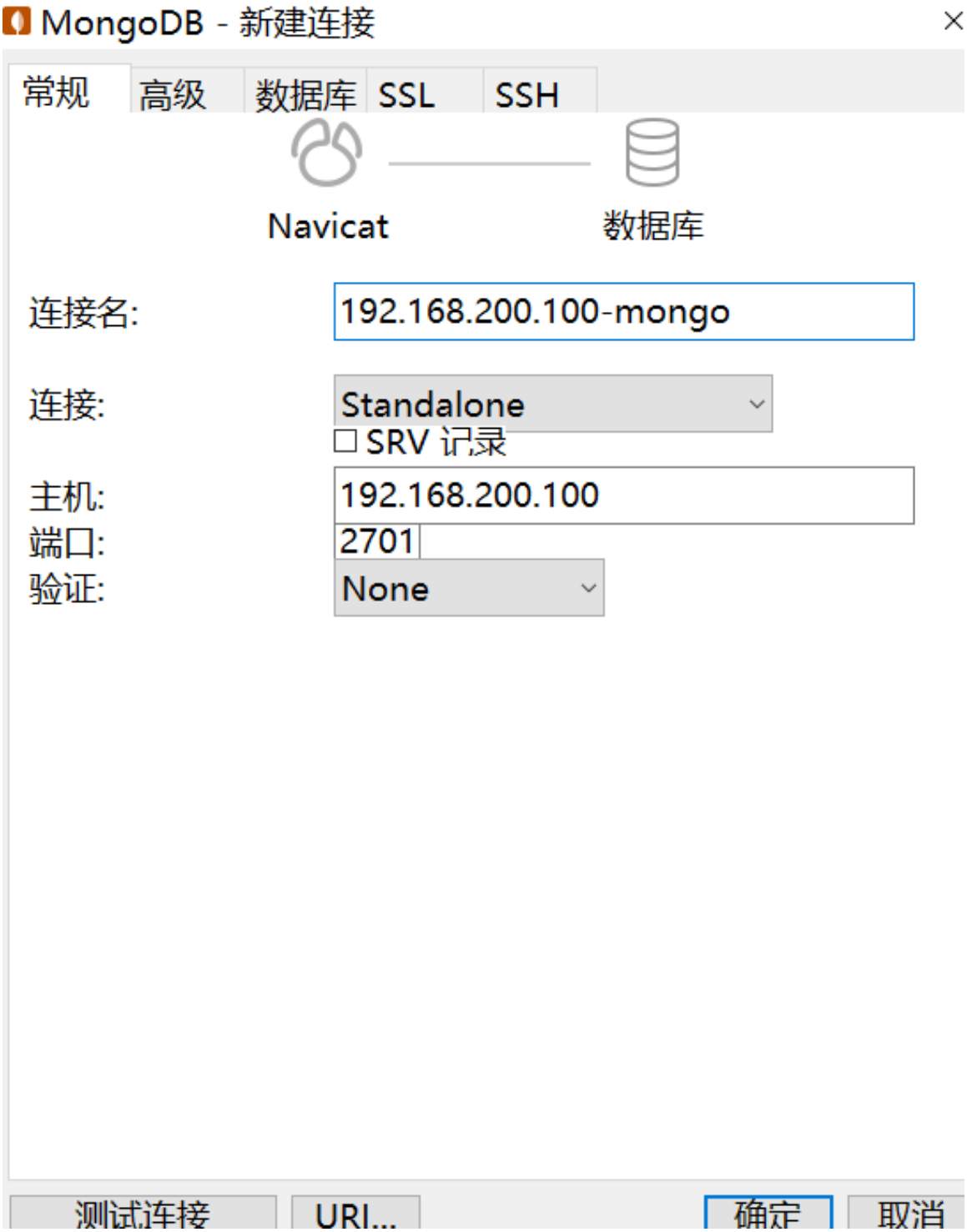
在api的test中新建mongo的测试类，测试mongo是否能正常使用

```
package com.mszlu.xt.sso.dao.mongo.data;

import lombok.Data;
import org.bson.types.ObjectId;
import org.springframework.data.mongodb.core.mapping.Document;

@Data
@Document(collection = "user_log")
public class UserLog {

    private ObjectId id;

    private Long userId;

    private boolean newer;

    private Long registerTime;

    private Long lastLoginTime;

    private Integer sex;
}
```

```
package com.mszlu.xt.sso.mongo;

import com.mszlu.xt.sso.dao.data.User;
import com.mszlu.xt.sso.dao.mongo.data.UserLog;
import org.junit.jupiter.api.Test;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;

import java.util.List;

@SpringBootTest
public class MongoTest {


    @Autowired
    private MongoTemplate mongoTemplate;

    @Test
    public void testSave(){
        UserLog userLog = new UserLog();
        userLog.setNewer(true);
        userLog.setSex(1);
        userLog.setUserId(1000L);
        userLog.setLastLoginTime(System.currentTimeMillis());
        userLog.setRegisterTime(System.currentTimeMillis());
        mongoTemplate.save(userLog);

        Query query = Query.query(Criteria.where("userId").is(1000));
        query.limit(1);
        UserLog log = mongoTemplate.findOne(query, UserLog.class);
        System.out.println(log);
    }
}
```

2.4.2 使用navicat连接mongo


MongoDB - 新建连接
常规
高级
数据库
SSL
SSH
Navicat
数据库
连接名:
192.168.200.100-mongo
连接:
Standalone
SRV 记录
主机:
192.168.200.100
端口:
2701
验证:
None
测试连接
URI...
确定
取消


192.168.200.100-mongo
xt
集合
user_log
视图
函数
索引
MapReduce
GridFS 存储桶
查询
开始事务
文本
筛选
排序
全部折叠
类型颜色
导入
导出
分析
_id
userId
newer
registerTime
lastLoginTime
sex
_class
616997d9c04d4
1000
true
1634310105512
1634310105512
1
com.mszlu.xt.ss

2.5 RocketMQ

> *RocketMQ 是由阿里用java语言开发的一款高性能、高吞吐量的分布式消息中间件，于2017年正式捐赠 Apache 基金会并成为顶级开源项目。*

好处：

1. **解耦**

   比如货款抵扣业务场景，用户生成订单发送MQ后立即返回，结算系统去消费该MQ进行用户账户金额的扣款。这样订单系统只需要关注把订单创建成功，最大可能的提高订单量，并且生成订单后立即返回用户。而结算系统重点关心的是账户金额的扣减，保证账户金额最终一致。

2. **冗余**

   有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。MQ把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。

3. **扩展性**

   因为MQ解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。

4. **灵活性和峰值处理能力**

   在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用MQ能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

   通过MQ的方式，消费端控制拉取速度，可以使流量趋于平稳，达到了削峰填谷的目的。

5. **可恢复性**

   系统的一部分组件失效时，不会影响到整个系统。MQ降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

6. **顺序保证**

   在大多使用场景下，数据处理的顺序都很重要。大部分MQ本来就是排序的，并且能保证数据会按照特定的顺序来处理

7. **缓冲**

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行——写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度

8. **异步通信**

   很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。

> *对系统而言，MQ消息队列机制能承受更大访问压力；对架构而言，松耦合，系统维护性方便；对用户而言，系统访问更快、系统体验更好*

2.5.1 安装

```
#docker 拉取
docker pull foxiswho/rocketmq:4.8.0
#启动nameserver
docker run -d -v /usr/local/rocketmq/logs:/opt/docker/rocketmq/logs \
      --name rmqnamesrv \
      -e "JAVA_OPT_EXT=-Xms512M -Xmx512M -Xmn128m" \
      -p 9876:9876 \
      foxiswho/rocketmq:4.8.0 \
      sh mqnamesrv
#broker.conf
brokerIP1=192.168.200.100
namesrvAddr=192.168.200.100:9876
brokerName=broker_all
#启动broker
docker run -d  -v /opt/docker/rocketmq/logs:/usr/local/rocketmq/logs -v
/opt/docker/rocketmq/store:/usr/local/rocketmq/store \
      -v /opt/docker/rocketmq/conf:/usr/local/rocketmq/conf \
      --name rmqbroker \
      -e "NAMESRV_ADDR=192.168.200.100:9876" \
      -e "JAVA_OPT_EXT=-Xms512M -Xmx512M -Xmn128m" \
      -p 10911:10911 -p 10912:10912 -p 10909:10909 \
      foxiswho/rocketmq:4.8.0 \
      sh mqbroker -c /usr/local/rocketmq/conf/broker.conf
#rocketmq-console-ng
docker pull styletang/rocketmq-console-ng

docker run --name rmqconsole --link rmqnamesrv:rmqnamesrv \
-e "JAVA_OPTS=-Drocketmq.namesrv.addr=192.168.200.100:9876 -
Dcom.rocketmq.sendMessageWithVIPChannel=false" \
```

```
-p 8180:8080 -t styletang/rocketmq-console-ng

#启动访问 http://192.168.200.100:8180/
```

2.6 sso集成rocketmq

在sso-domain中导入rocketmq的依赖:

```
  <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.8.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-spring-boot-starter</artifactId>
            <version>2.2.0</version>
        </dependency>
```

添加对应的配置:

```
#rocketmq配置
rocketmq.name-server=192.168.200.100:9876
rocketmq.producer.group=xt_log_group
```

消息生产者:

```
package com.mszlu.xt.sso.mongo;

import com.mszlu.xt.sso.dao.mongo.data.UserLog;
import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;

@SpringBootTest
public class RocketMQTest {
```

```
    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    @Test
    public void testSend(){
        UserLog userLog = new UserLog();
        userLog.setNewer(true);
        userLog.setSex(1);
        userLog.setUserId(1000L);
        userLog.setLastLoginTime(System.currentTimeMillis());
        userLog.setRegisterTime(System.currentTimeMillis());
        rocketMQTemplate.convertAndSend("xt_log_sso_login",userLog);
    }
}
```

消息消费者:

需要写在代码中，并重启sso，消费者为监听程序

```
package com.mszlu.xt.sso.handler;

import com.mszlu.xt.sso.dao.mongo.data.UserLog;
import lombok.extern.slf4j.Slf4j;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.stereotype.Component;

@RocketMQMessageListener(topic = "xt_log_sso_login",consumerGroup =
"login_group")
@Slf4j
@Component
public class Consumer implements RocketMQListener<UserLog> {
    @Override
    public void onMessage(UserLog message) {
        log.info("消息消费:{}",message);
    }
}
```

2.7 登录日志实现

2.7.1 记录日志

```
    @Override
     public CallResult wxLoginCallBack(LoginParam loginParam) {
         LoginDomain loginDomain =
this.loginDomainRepository.createDomain(loginParam);
         return this.serviceTemplate.execute(new AbstractTemplateAction()
{
             @Override
             public CallResult doAction() {
                 return loginDomain.wxLoginCallBack();
             }

             @Override
             public void finishUp(CallResult callResult) {
                 loginDomain.wxLoginCallBackFinishUp(callResult);
             }
         });
     }
```

```
@Autowired
    private RocketMQTemplate rocketMQTemplate;
    private User user;
    private boolean newer;

    public void wxLoginCallBackFinishUp(CallResult callResult) {
        //记录日志
        if (callResult.isSuccess()){
            UserLog userLog = new UserLog();
            userLog.setRegisterTime(user.getRegisterTime());
            userLog.setLastLoginTime(user.getLastLoginTime());
            userLog.setUserId(user.getId());
            userLog.setSex(user.getSex());
            userLog.setNewer(newer);

 this.rocketMQTemplate.convertAndSend("xt_log_sso_login",userLog);
        }
    }
```

2.7.2 问题

> *当rocketMQ挂掉的时候，我们发现登录功能不能用了?*
>
> *记录日志的功能还能影响登录功能?*

有两种解决方案:

1. 使用异步
2. 使用线程池

> *这里，我们使用线程池解决*

线程池配置:

```
package com.mszlu.xt.sso.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.task.AsyncTaskExecutor;
import org.springframework.scheduling.annotation.EnableAsync;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;

import java.util.concurrent.*;

@Configuration
@EnableAsync
public class SpringAsyncConfig {

    @Bean("taskExecutor")
    public Executor asyncServiceExecutor() {
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        // 设置核心线程数
        executor.setCorePoolSize(5);
        // 设置最大线程数
        executor.setMaxPoolSize(5);
        //配置队列大小
        executor.setQueueCapacity(Integer.MAX_VALUE);
        // 设置线程活跃时间（秒）
        executor.setKeepAliveSeconds(60);
        // 设置默认线程名称
        executor.setThreadNamePrefix("sso-thread");
        // 等待所有任务结束后再关闭线程池
```

```
        executor.setWaitForTasksToCompleteOnShutdown(true);
        //执行初始化
        executor.initialize();
        return executor;
    }
}
```

使用线程池，将队列发消息的行为放入线程池执行：

```
package com.mszlu.xt.sso.domain.thread;

import com.mszlu.xt.sso.dao.mongo.data.UserLog;
import org.apache.rocketmq.spring.core.RocketMQTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.scheduling.annotation.Async;
import org.springframework.stereotype.Component;

@Component
public class LogThread {

    @Autowired
    private RocketMQTemplate rocketMQTemplate;

    @Async("taskExecutor")
    public void send(String topic, UserLog userLog) {
        rocketMQTemplate.convertAndSend(topic,userLog);
    }
}
```

```
 @Autowired
    private LogThread logThread;

    public void recordUserLog(UserLog userLog) {
        logThread.send("xt_log_sso_login",userLog);
    }
```

```
 private User user;
    private boolean newer;


    public void wxLoginCallBackFinishUp(CallResult callResult) {
        //记录日志
```

```
        if (callResult.isSuccess()){
            UserLog userLog = new UserLog();
            userLog.setRegisterTime(user.getRegisterTime());
            userLog.setLastLoginTime(user.getLastLoginTime());
            userLog.setUserId(user.getId());
            userLog.setSex(user.getSex());
            userLog.setNewer(newer);
            this.loginDomainRepository.recordUserLog(userLog);
        }
    }
```

2.7.3 消费

> *日志经由sso发送后，需要写一个程序专门处理日志，将其记录到mongo表中，为后续的业务(比如 数据分析)做好持久化的操作*

新建模块mszlu-xt-log

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>mszlu-xt-parent</artifactId>
        <groupId>com.mszlu</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>mszlu-xt-log</artifactId>

    <dependencies>
        <dependency>
            <groupId>com.mszlu</groupId>
            <artifactId>mszlu-xt-common</artifactId>
            <version>1.0-SNAPSHOT</version>
            <exclusions>
                <exclusion>
                    <groupId>org.springframework.boot</groupId>
                    <artifactId>spring-boot-starter-jdbc</artifactId>
                </exclusion>
            </exclusions>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-client</artifactId>
            <version>4.8.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.rocketmq</groupId>
            <artifactId>rocketmq-spring-boot-starter</artifactId>
            <version>2.2.0</version>
        </dependency>
    </dependencies>
</project>
```

配置:

```
##启动端口号
server.port=8318
spring.application.name=xt-log

# mongo的配置
spring.data.mongodb.uri=mongodb://192.168.200.100:27017/xt

#rocketmq配置
rocketmq.name-server=192.168.200.100:9876
rocketmq.producer.group=xt_log_group
```

日志消费，记录mongo中:

```
package com.mszlu.xt.log.consumer;
import lombok.extern.slf4j.Slf4j;
import org.apache.rocketmq.spring.annotation.RocketMQMessageListener;
```

```java
import org.apache.rocketmq.spring.core.RocketMQListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Component;

@RocketMQMessageListener(topic = "xt_log_sso_login",consumerGroup =
"login_group")
@Slf4j
@Component
public class SSOConsumer implements RocketMQListener<UserLog> {

    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public void onMessage(UserLog message) {
        log.info("消息消费:{}",message);
        mongoTemplate.save(message);
    }
}
```

```java
package com.mszlu.xt.log;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LogApp {

    public static void main(String[] args) {
        SpringApplication.run(LogApp.class,args);
    }
}
```