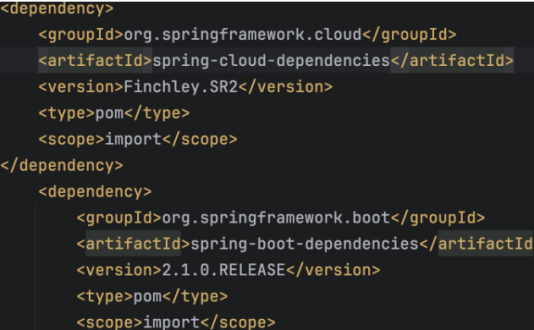
# 启动必须启动

Mqtt zookeeper

# 初化项目

首先我们的项目是一个springboot的项目，你可以选择sprinbgoot或者自己建立都是可以的，这里比较重要的是pom引入，

通过这个pom，可以规范我们和springboot和cloud的版本是一致的，不会有版本错误，同时这个项目的pom引入十分的干净，不会有很乱的pom存在，大家也可以下载以后自己看看。子模块需要什么就自己去继承，不需要版本号，会继承统一的版本管理。子模块之间互相引用会用到子的所有包，所以越上面的越什么都不用引入。

这里就可以启动项目了，不过你需要注释所有配置和功能。

# 然后我们引入mysql

这里推荐是下载绿色版，也就是不需要安装只是解压使用的这种

下载地址

<https://dev.mysql.com/downloads/mysql/>

请下载8.4.5版本

下载以后解压，然后初始化你的mysql，在bin目录下

mysqld --initialize --console

然后就会生成root的默认密码

记下来，然后安装mysql，同目录下执行

Mysqld --install MYSQL8

然后启动net start mysql8

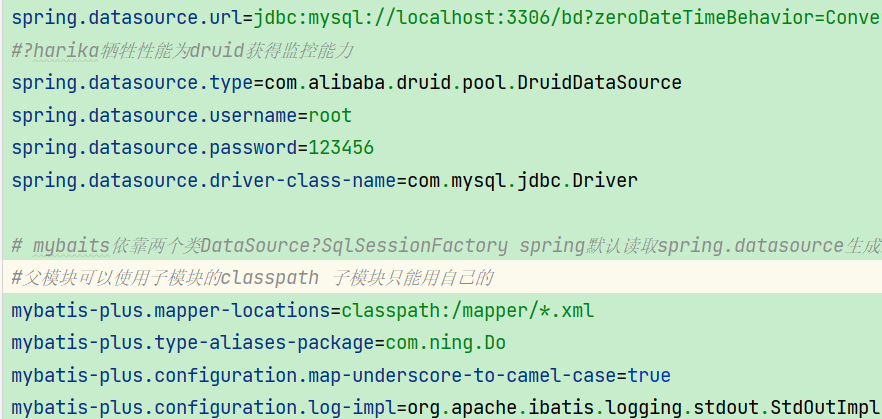
然后登录

Mysql -uroot -p 复制上你刚才的那个密码

登录以后修改密码

set password='123456';

有了mysql以后对项目进行修改，配置中增加数据源配置，启动类增加xml文件扫描即可。



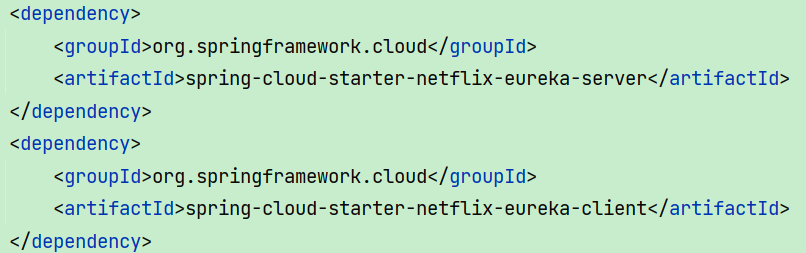


实际使用时会集群部署

# 引入eureka

Eureka没有外部启动，所以我们需要作为server同时也作为client注册上去。

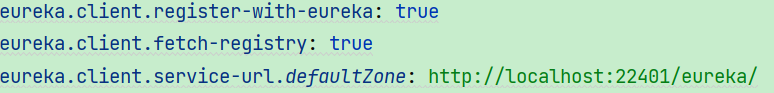
首先引入pom，在api包引入



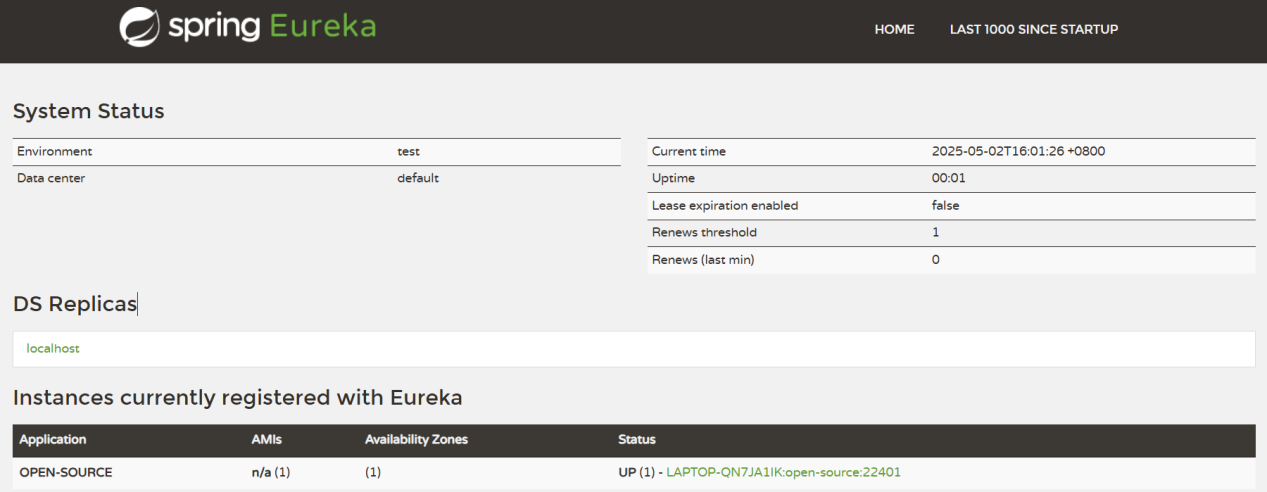
服务端只要在启动类上增加注解即可



客户端则是需要在配置中写明，是否要注册上去，以及注册的地址



成功以后的界面如下，登录UI为我们自己的端口，唯一的服务也是我们注册上去的



和eureka交互，可以使用client



eureka去除实例是通过心跳续约和我们之前一样，定时任务删除

不同的是有保护机制，即大量失效的时候，会停止删除，判断此时网络并不好

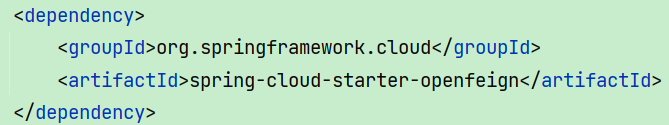
有主动把自己剔除的接口，也可以自己直接调取api如果获取某个服务的地址端口号那些

配置中需要配置你的注册地址在哪里，如果你是服务端，那么端口号就是你的项目启动号

没有密码配置，也就是说如果需要安全验证，需要自己拦截去做

# 引入feign

一个pom即可



然后就是直接使用了



正常我们应该是把第三方的api引入进来，但是这里是我们一个项目，所以只能自己引用自己了

子模块互相依赖确实是不可以的，解决方式是提取出去，然后共同依赖那一个模块

和mq一样可能会有循环调用 需要自己业务避免

唯一需要注意的是 你的服务名字应该和eureka上面的对应上即可

# 引入canal

MySQL 的权限是基于用户名和主机名的组合进行管理的。例如，'root'@'localhost' 和 'root'@'%' 是两个不同的用户。

账号需要权限，同时mysql需要启用binlog并且格式是row的。

默认端口3306

下载deployer，

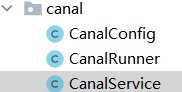
<https://github.com/alibaba/canal/releases?spm=5176.28103460.0.0.2f2e1db83S5glh>

然后启动，默认端口1111，配置需要监听的数据库地址和账号，启动参数修改的小一些

然后是项目中

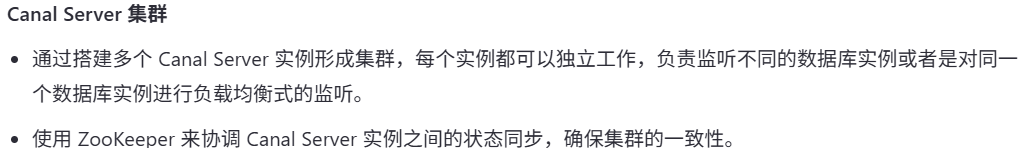


通过这里



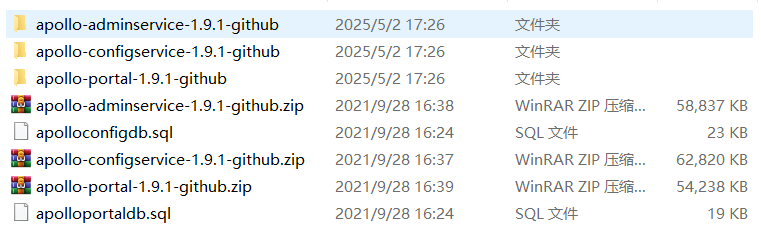
完成canal客户端的初始化，启动监听器，监听数据并处理

实际使用时是集群化部署，一个负责几张表，不会多个负责一个，有顺序问题



# 引入apollo

Apollo是依赖mysql的所以需要执行初始化mysql，同时需要启动三个组件



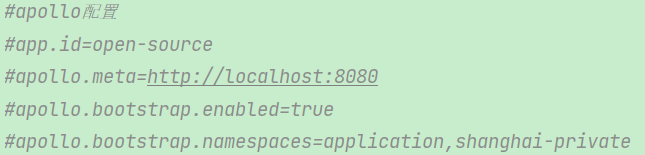
apollo的配置分级如下 环境--集群-namespace

环境就是dev这些，集群应该是按照位置（上海）或者逻辑划分来决定，namespace可以说是最小的单位，和appid绑定，不和集群是绑定。

对于本应用来说创建一个namespace，每个集群都能看到，但是内容是各自独立的。一个应用的公共namespace是可以被其他应用使用的。如果想要用其他应用的配置则是通过去关联那个应用的公共namespace来完成。

加载哪个环境、哪个集群通过jvm启动参数控制,配置的优先级控制是在配置文件中谁在前谁优先，我们也可以通过指定namespace去获取特定的配置

页面端口号8070，springboot连接config组件，默认端口为8080



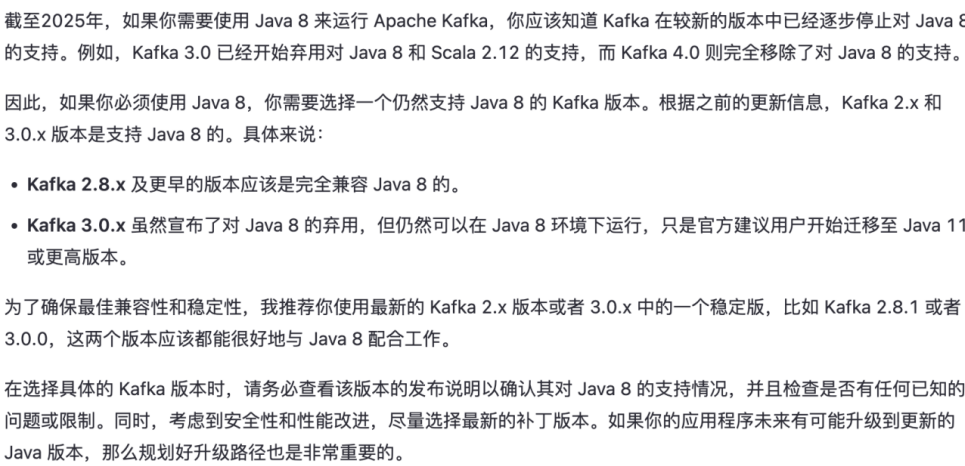




# 引入kafka

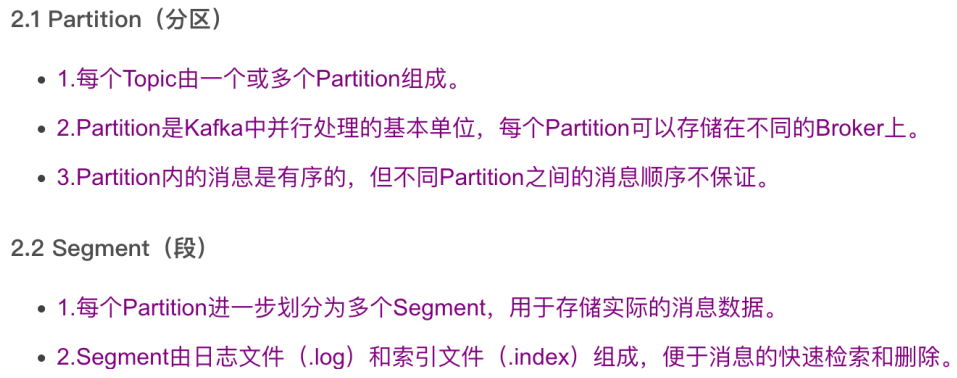
<https://kafka.apache.org/documentation/#quickstart>

Kafka依赖zookeeper，所以在启动时候需要先启动zookeeper



关于消息队列的一些问题：

kafka的分区是保证顺序的，同时kafka的一个分区只能有一个消费者



kafka消费消息，失败了就失败了，如果需要重试那些都需要自己写。

kafka中消息就是消息，没有什么消息类型，没有mq的延迟、顺序、定时消息等等。

kafka也支持挺多能力，但是基本上都需要自己去显式的设置参数，或者去封装逻辑。

--------------------------------------kafka和mq的分界线--------------------------------------

mq的队列和分区意思是差不多的，

mq的消费者都需要属于一个消费者组

mq的消费者可以只消费某个topic下的某个队列，通过queueId过滤即可

mq的topic的默认队列数是4

生产者发送消息时，队列选择默认时轮训，不过可以自己去指定，通过某个key进行计算，这样可以把需要保证顺序的消息在一个队列中，只有这样才能保证消息消费的顺序性

mq的消费逻辑如下，topic下的多个消费者，如果属于同一个消费者组，那么就会负载均衡，消费者会被分配队列，确保不会有同一个消费者组中消费者消费同一个队列，避免了重复消费问题。如果有消费者退出或者加入，会重新负载均衡。如果是属于多个消费者组，那么就会

mq不论是推、拉都会遇到当前线程未处理完，下一次如何触发的问题：

先说pull模式，完全由我们自己控制，可以决定这次没处理完是下一次并发处理，还是等这次结束再发起下一次。

如果是push，首先mq的push也是pull，push中有参数可以控制批次大小，和处理线程min和max，同时mq自己也有流量控制，选择合适的即可。

rocketMq的持久化也是log+同异步刷盘，唯一不同的是每个队列有一个索引文件，索引自己的消息是属于那个log文件的，我认为是便与快速恢复的。

mq的重试是这样的，消费者消费不成功，mq会把消息给到重试队列，这个队列是一个消费者组一个，然后重试，重试超过最大次数比如16次，就会到死信队列，死信队列需要自己定期去手动维护了就。

如何消费过去的数据我想重试消费，如何做呢？

可以使用Message ID再查询出这条

mq中的消息只会根据你的时间和存储比例来删除，不会根据其他，这和百度一样，死心队列则是不会删除

broker有几个如何决定？其中有一个topic和队列呢？topic是不是在namespace里面存着？

broker通过你的吞吐来决定 topic是全部的 只是下面的队列会分布在不同的broker上面

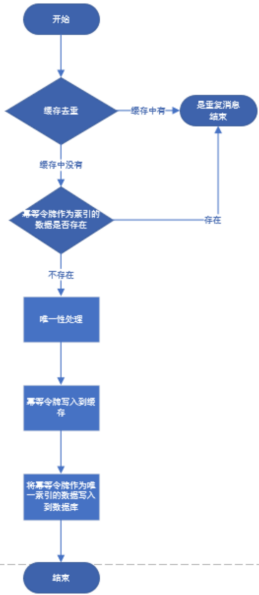
消费者的消费位点存储在broker中，而broker通过主从增加了HA能力

mq的master-slave是broker层级的，类似于kafka在创建topic中的副本概念

mq的主从模式处理和问题解决可以参照mysql都一样的

mq的消费者应该实现幂等性支持，因为网络原因等不可估算，比如produce和broker之间、broker和consumer之间、又或者consumer rebalance了都会造成重复消费。

解决方式如下



面对消息堆积：首先就会引起消费延迟，有些实时性业务是接受不了的，一定是增加消费者，解决，然后优化消费逻辑，增加队列和消费者数量、提高消费者并发度、减少IO能本地做的就本地做。

选择pull还是push：取决于你的需求，pull控制力更大可以自定义很多东西，push延迟更小而且开发简单，push就算设置多个线程也是一个消费者，也就是说会影响顺序性。

mq produce发送消息 同步的话失败会换broker重试 如何异步则是一个broker重试 无法保证消息不丢失。

mq的顺序消息在被消费时，如果失败就会不断重试，所以要重点监控。无序消息的重试可以通过消费者去返回失败状态来重试，需要注意的是只对集群消费有效对广播消费无效。重试时间是不断拉长的。

mq的消费者是有类型的，就是我们继承的那个类，有并发和顺序型，当消费失败，mq自己重试时，顺序性会阻塞后面的消息，而并发的不会。

最大的区别是 kafka依赖zookeeper。mq有更多样的队列种类重试、死信队列，更灵活的路由比如Tag，kafka需要自己实现。kafka可以增加分区不能减少、mq不能修改。kafka更适合大数据量的存储，mq更适合业务的处理。

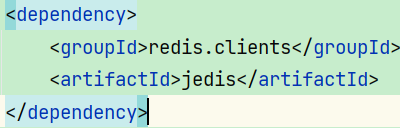
# 引入redis

下载安装redis 5.0.14.1

启动命令

redis-server.exe redis.windows.conf

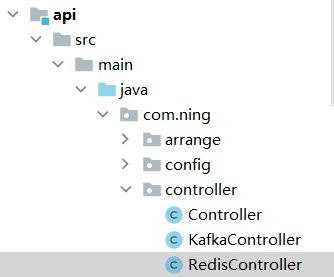
选择使用jedis进行通讯



使用lua 示例在这里



主要解决原子性问题，性能问题，复杂操作问题如获取多值、数据聚合等



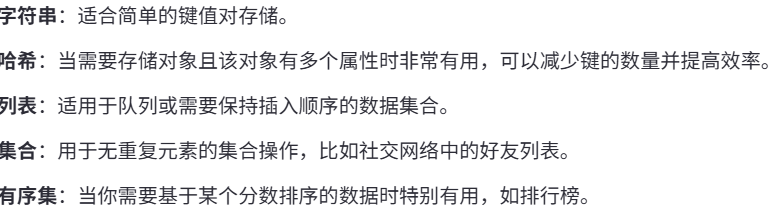
Lua示例

Lua脚本在集群中运行时，需要所操作数据都在一个节点上面，因为要保证原子性，如果在不同节点上就保证不了原子性了，当然可以通过分布式事务来做，但是redis认为其代价太高不好维护，所以是没有的。

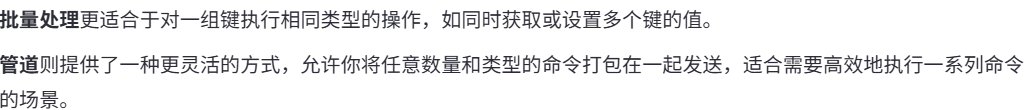
Redis的故障转移是指一个从节点down了，会选举出一个从节点升主，再更新集群配置，会给客户端发送move或者ack来更新其指向。

你的数据放在哪个slot是根据你的key名字通过CRC16算法再对16384取模来决定的。你也可以通过hashTag来控制相同的在一起，如：{user:1000}.followers

根据你的需要选择类型



尽量使用批处理和管道来增加效率



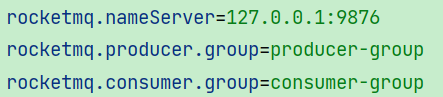
# 引入rocketMQ

下载rocketMQ 4.9.3

引入pom



写入配置



启动mqnamesrv，启动broker

示例位置



事务消息需要实例化事务生产者，同时需要有执行本地事务和回查事务的类实现

MessageListenerOrderly作为消费者时，消费失败，会不断重试阻塞后续消息，为了保证顺序性。如果你是并发消费，不会阻塞。

实际部署时一般是两个namesrv，多个broker，每个broker都和所有的namesrv进行连接

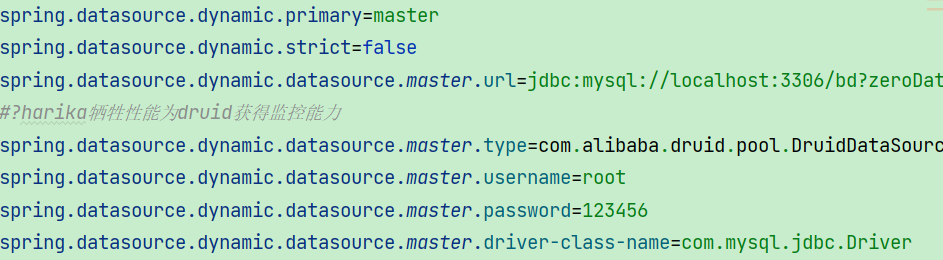
# 多数据源整合

整合多个数据源

一般是生成多个datasource、然后设置sqlSession、共享事务等等



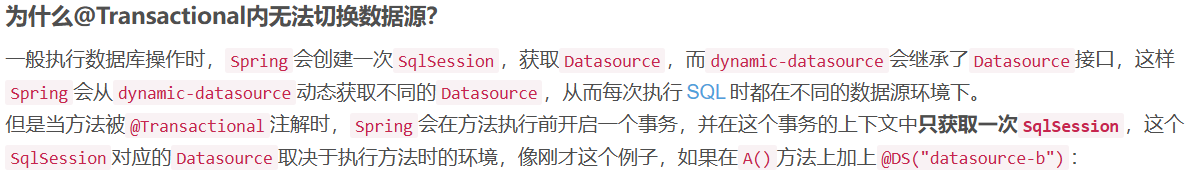
我们这里使用了dynamic-datasource组件



配合注解@DS可以直接开始使用

如果还需要集成es、ck等数据源，则可以通过client的方式来整合

多数据源下的事务通过注解DSTransactional来实现，原来的Transactional切换数据源会失效事务效果还在，原因如下



也就是说你在A方法中操作了a库和b库，但是都会在b库执行。

DSTransactional是如何实现的呢？

通过把每个数据源操作都记录下来，最后循环一遍，依次提交或者回滚。

如果你还使用了mongo，则需要把mongo的事务管理器初始化出来，在注解中使用。

在@Transactional实现原理中，有一个事务管理器的概念，Spring框架通过PlatformTransactionManager接口来定义事务管理器的统一接口，这个接口规定了事务的基本操作，如开启事务、提交事务、回滚事务等。对于不同的数据访问技术，例如JDBC、Hibernate、JPA等，都有对应的事务管理器实现。

这样使用可以让多数据源和mongo都会滚

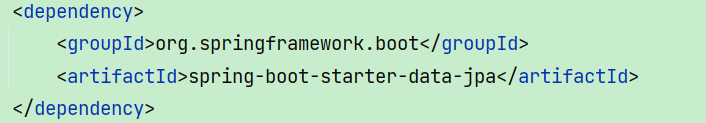
@DSTransactional

@Transactional(transactionManager = "mongoTransactionManager")

# 整合jpa

经常jpa和mybatis是会一起使用的，jpa完成简单操作，mybatis完成复杂sql，又或者是历史原因等等

引入pom



配置类

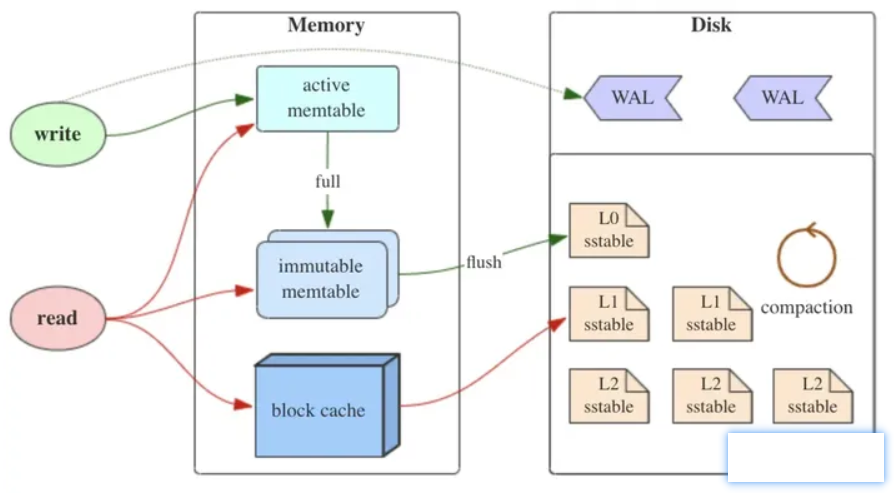




配置中的内容请根据自己的实际路径进行修改，然后就可以进行使用了

# 整合es

你来补充

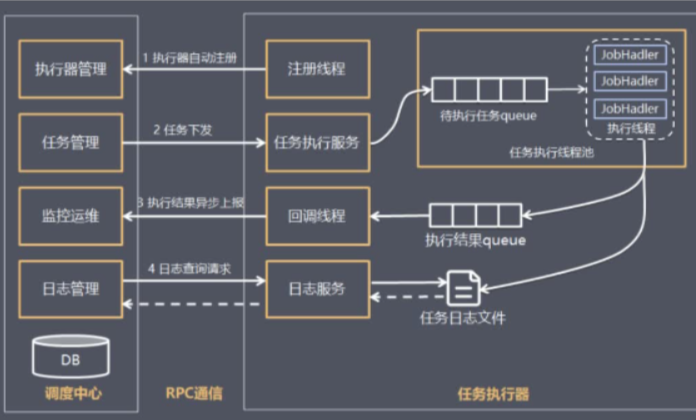


# 整合k8s

你来补充

# 引入Xxljob

一个定时任务平台适合分布式，单体的springtask不够用



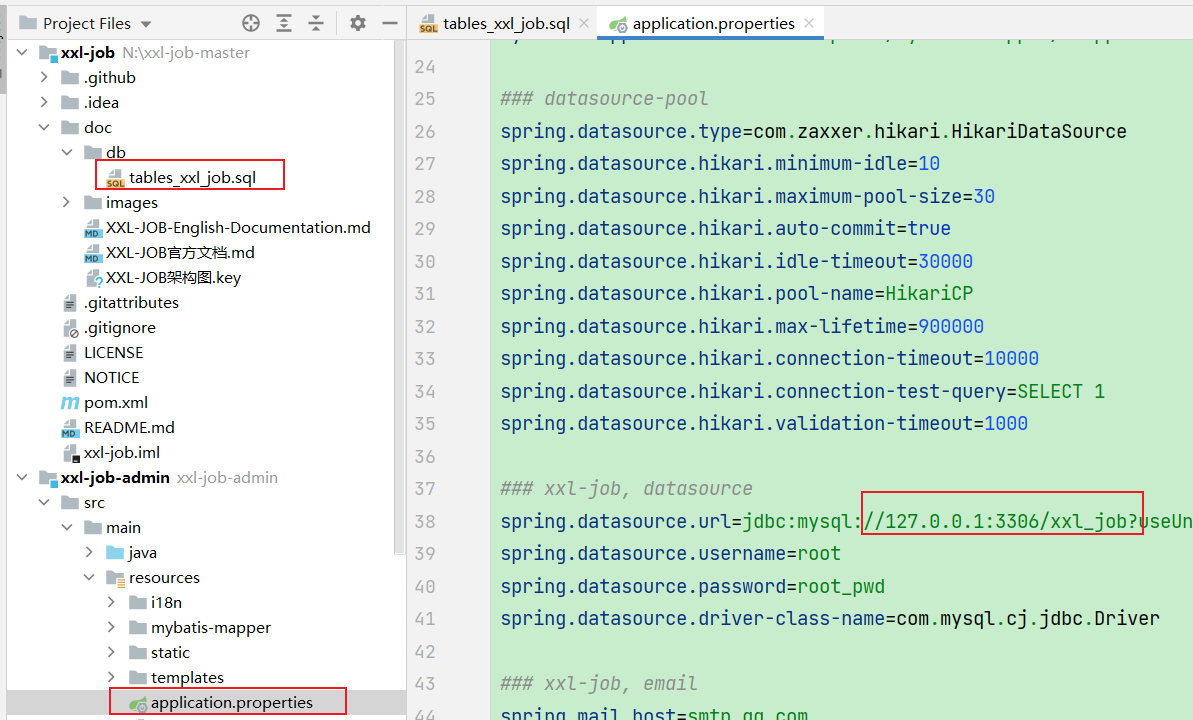
从这张图就能大致看出来运行原理了

下载地址

<https://github.com/xuxueli/xxl-job>

请下载2.3版本

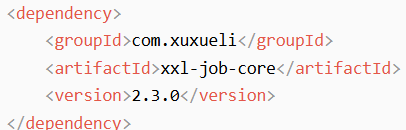
打开项目，执行sql，修改配置

l

启动admin



自己项目中引入pom和配置即可



# 引入Zookeeper

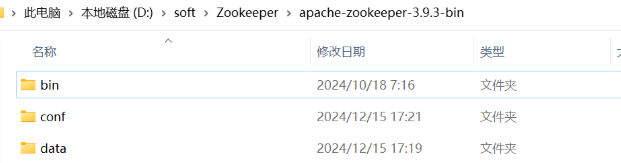
先安装一个zoo，为后面的dubbo做准备

下载地址 下载版本3.9.3

<https://archive.apache.org/dist/zookeeper/zookeeper-3.9.3/>

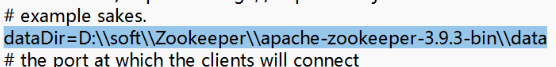
下载以后

目录夹下创建data目录





并在这个配置文件中写上我们刚才的路径



然后是启动校验

服务端是



客户端是



默认端口是2181

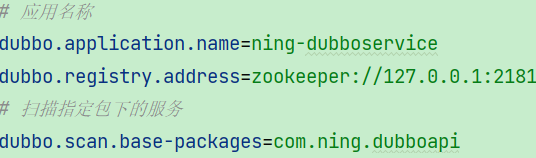
# 引入Dubbo

有了zoo以后就可以搭建dubbo了，还是三步

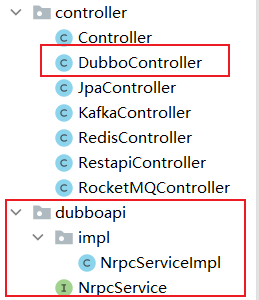
引pom



写配置



示例



在实践过后，大家就会明白为什么springcloud是微服务解决方案，而dubbo只是远程调用方式了吧，所有这些服务注册都是可以跨语言的，只要有client就可以，如果必须引pom，只要有rpc转那个语言的插件，那也是没问题的，毕竟最后的最后都是网络传输和序列化，有什么做不了的呢？而且这些注册配置中心本身都是可以被替代的，很多注册中心如nacos就会提供springcloud可以用的pom，而且就算本身没有包支持，你也是可以用第三方包自己写逻辑来支持的。

Dubbo支持多种协议去暴露你的服务，一般根据你的需要来选择，多语言那就rest，网络第几层？或者你是需要一些性能要求的等等。

Dubbo是可以一个接口共存多个版本的，

Nacos的出现是为了有更多微服务治理的功能和生态，但是在一致性上不如zoo，只能说在选择的时候需要根据自己的场景做出选择。

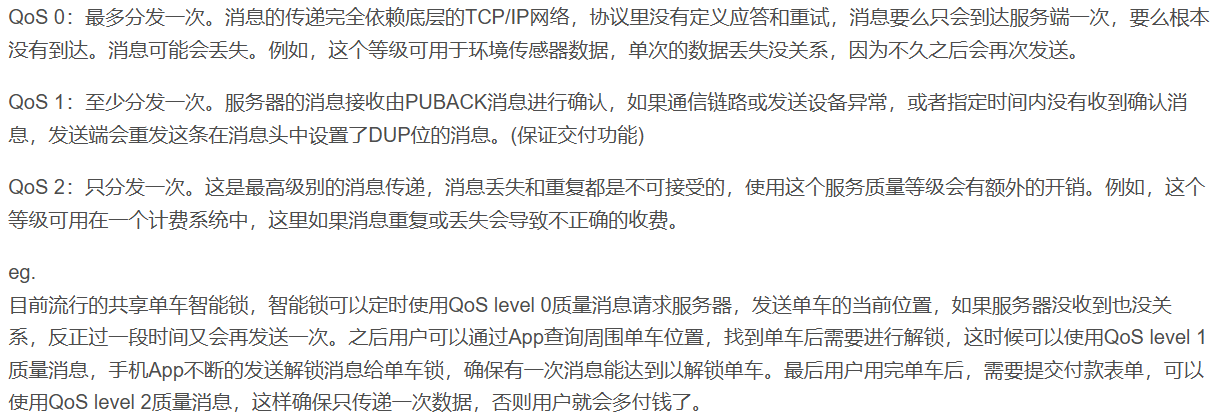
Service注解比较重要的参数有version group timeout retryTime loadblance connections executes protocol

同样的参数在消费者和生产者都设置时，一般都是以消费者的设置为主。

# 引入Mqtt

物联网之所以选择mqtt是因为，mqtt天生就是为了解决网络不好的情况的，同时轻量级对于设备来说负担小，最小报文两个字节。还有恢复会话的功能。还有就是大家都支持mqtt这是生态了就。

Mqtt中有一个消息等级，在我看来就是可能丢失、保证不丢失但是会重复、不丢失也不重试



和rocketmq对比：

Mqtt有遗嘱消息，可以设置消息等级，同时持久化不同是根据你的会话状态（Clean Session）和broker具体实现（如 Mosquitto、EMQ X 等），一些 Broker 可能支持将消息存储到磁盘上，而另一些可能仅在内存中维持这些信息）来决定的。Mqtt也是可以过滤的通过通配符有特定语法支持。支持websocket可以直接从浏览器、手机端去使用。只有推的方式比较符合低功耗要求。

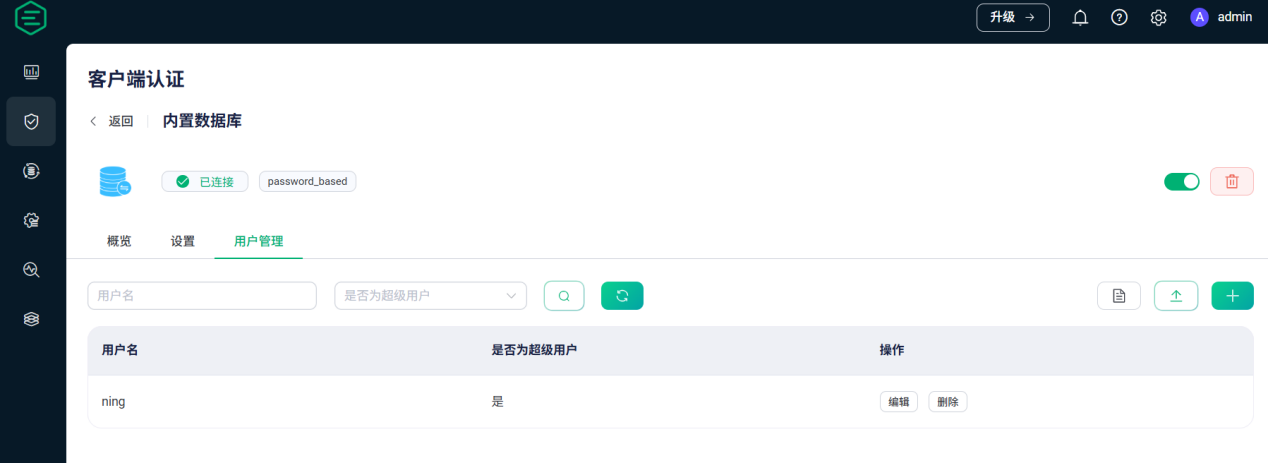
<https://www.emqx.com/zh/downloads/broker/v5.3.2>

启动命令在bin目录下的 emqx start

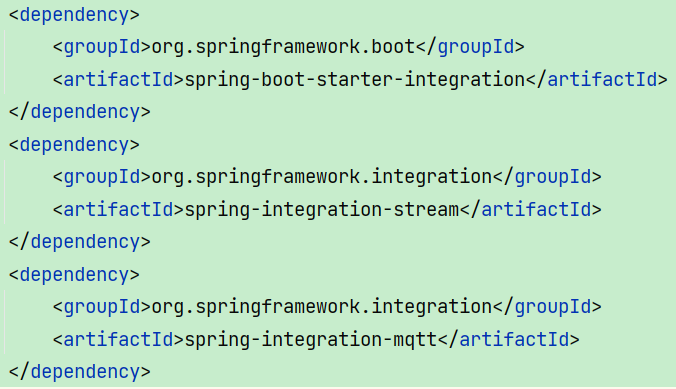
Web登录页地址<http://localhost:18083/#/login>

登录默认账号 admin 密码public

登录进去以后是一个dasboard，在里面增加我们的用户，后面集成springboot需要



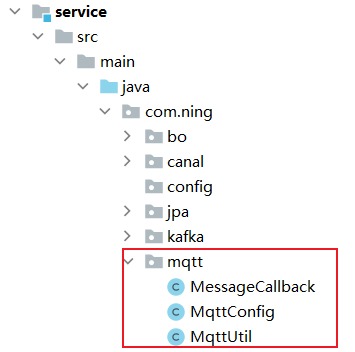
引入pom



写入配置



启动示例



示例中callback继承的是mqttCallbackExtend比mqttCallback接口多了一些方法

有四个方法

connectComplete

客户端连接上以后触发，一般是写一些topic订阅，初始化资源等代码

connectionLost

客户端连接丢失以后触发，一般是启动定时任务去重新连接，同时记录日志，还会通知相关组件自己现在不能提供功能了

messageArrived

处理到来的消息

deliveryComplete

消息成功发送以后调用，这里会把本地的消息缓存清空，如果你的消息持久化了的话，会修改状态为已发送等等

发送时message对象的retained作用是，这一条消息会被客户端立刻收到，意思就是一个客户端可能重连上来了，这时候不会有数据空白期即等待你的下一个发送，而是会把这一条数据立刻发送过去，相当于时是最新的状态信息，也就是最后一次的状态。

消息发送到broker上面保存多久，看你的配置是否开启了会话持久化、broker实现、消息的等级等等都会影响。

发送消息会返回MqttDeliveryToken，每个这个就是一个消息的动作绑定对象，可以为这个对象设置回调函数，去确定后续操作，还可以调用waitForCompletion变成同步操作，适合你必须发送完消息以后才能做某件事情的时候使用。

# Jsqlparser填坑

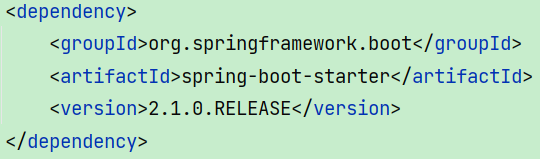
一个可以把sql语法分析成AST的工具，up之前开源的算法较为简陋，建议大家用这个。

Springboot2.1适配版本

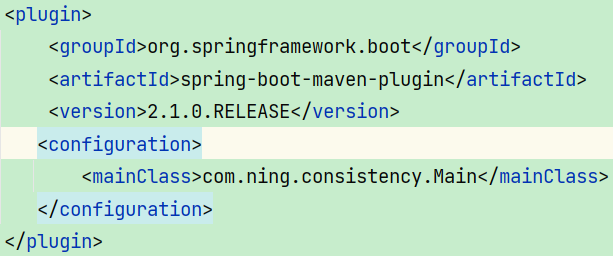
# 自建starter

我们创建一个自己的start项目，并引入，去使用其中的功能。

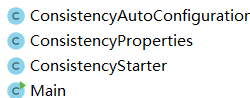
创建一个maven项目，引入pom，这一个就足够不需要其他



打包插件，不要添加其他，会导致jar包结构不同，这里我们需要打包成一个被引用的，而不是可以独立运行的jar包，不要加repackage



创建4个类，一个处理主项目的配置做初始化，一个是我们真正的服务类（根据配置去生成），一个配置类就是利用starter机制的钩子类，去把我们的服务类注入进spring中，供主项目使用。一个主类打包时使用。



创建spring.factories写入实现类的路径

到此就OK了，然后打包，放到本地maven中，命令如下

mvn install:install-file -Dfile=consistency-springboot-starter-1.0-SNAPSHOT.jar -DgroupId=com.ning -DartifactId=consistency-springboot-starter -Dversion=1.0-SNAPSHOT -Dpackaging=jar

然后去主项目中引用，使用即可，注意主项目中增加对应的配置。

# 使用Sentinel

下载地址

https://github.com/alibaba/Sentinel/releases

启动命令 不要用8080端口 容易与其他组件冲突

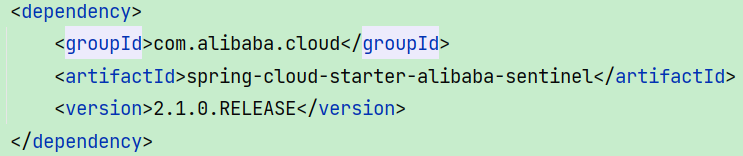
java -Dserver.port=8081 -Dcsp.sentinel.dashboard.server=localhost:8081 -Dproject.name=sentinel-dashboard -jar sentinel-dashboard-1.8.1.jar

进入登录页面

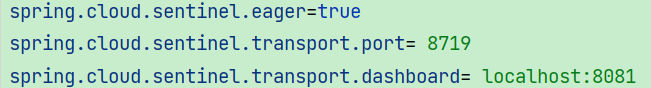
[http://localhost:8081/#/login](http://localhost:8080/#/login)

账号密码都是sentinel

引入pom



写入配置



示例



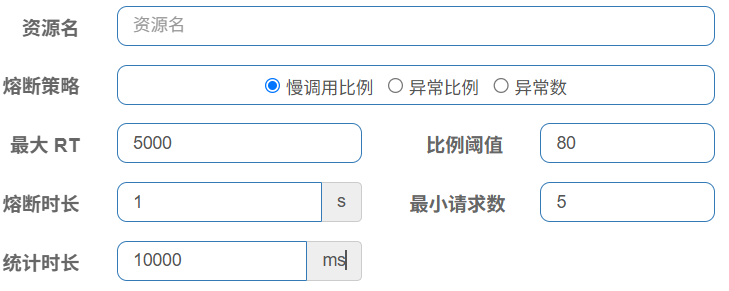
Sentinel可以做什么



可以看到你的资源有哪些，也就是你的接口，可以为每个接口根据来源，（即链路流控这个来源匹配是sentinel自己内置的网络头参数需要对应上才能识别到，这个需要看文档）去设置流控根据qps或者线程来限制，还可以关联流控当A限流B也限流，如何限制可以选择快速失败或者排队等方式。还有warmup也就是水位逐渐升高，举例就是qps到3了之后就限制住，然后逐渐升高到10，举一个场景：有一个接口长时间qps很低，那么对应的缓存大多数都会失效，这时候如果大流量来了，可以在qps逐渐增长的情况下，完成缓存的重新建立，又有了缓存又抬高了qps应付了流量激增。



还可以建立降级规则，通过异常请求比例、或者慢请求比例、异常个数来触发。



这个设置的含义是：在10秒内至少5个请求，有80%是慢调用超过了5s，就会熔断1s，1s以后尝试是否服务已经恢复。

热点规则是根据请求中的参数来进行限制，相当是更加细粒度的控制了。

系统规则就是内置的规则了，比如你们部门的、同组的一些规则，大家应该使用一样的。

集群流控，可以对集群内的机器上的某个端口进行具体的qps设置。

机器列表和实时监控、簇点链路一样，都是监视集群内部的各种信息的。

Sentinel默认是不保存规则的，规则的持久化，通过自己修改sentinel自身，或者你的应用在初始化时就把规则传上来，相当于时自己做了持久化。

# 使用Skywalking跟踪

下载地址

<https://skywalking.apache.org/downloads/>

旧版本下载地址8.9 最后一个支持jdk8的 9.0之后的就是jdk11了

老版本地址

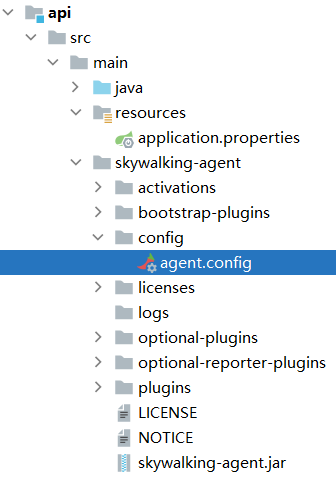
<https://archive.apache.org/dist/skywalking/>

<https://archive.apache.org/dist/skywalking/java-agent/>

启动apm在bin目录下的startup.bat

Web页面端口修改位置在webapp下的yml文件中 我改为了8082

然后拷贝agent项目到你的根目录下



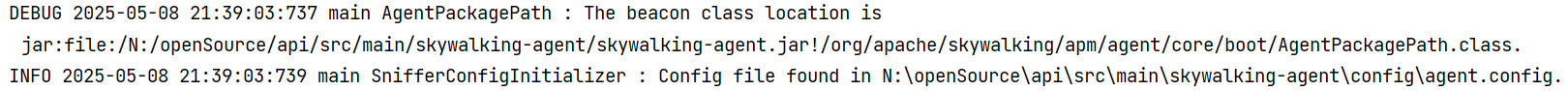
修改启动参数增加jvm参数

-javaagent:N:/openSource/api/src/main/skywalking-agent/skywalking-agent.jar

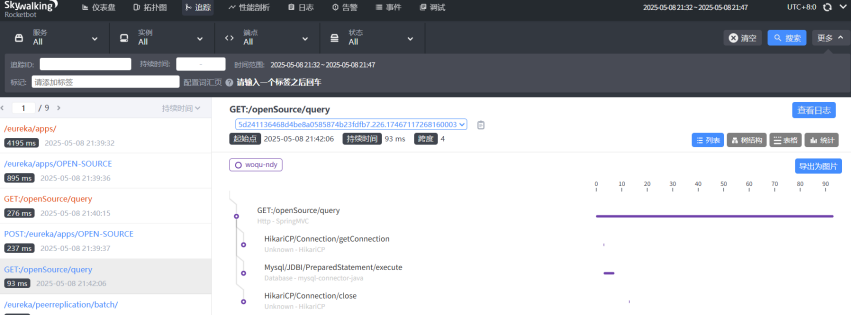
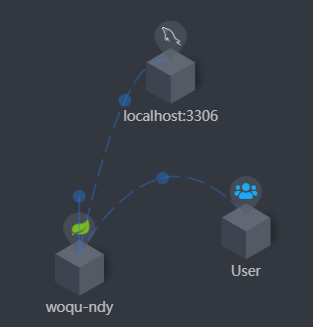
-Dskywalking.agent.service\_name=woqu-ndy

-Dskywalking.collector.backend\_service=127.0.0.1:11800

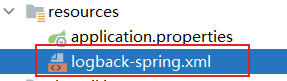
出现这个说明你已经连接上了

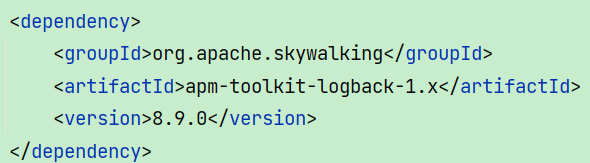


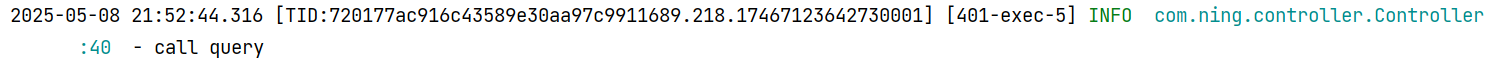
可以看到界面上有了信息



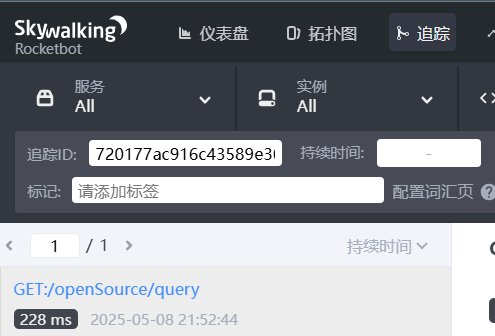
增加日志打印，打印出traceId

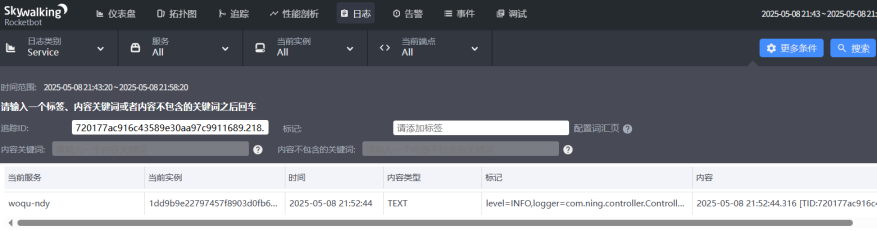




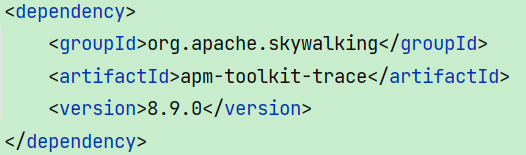


然后在这里搜索就可以搜索到对应的请求了





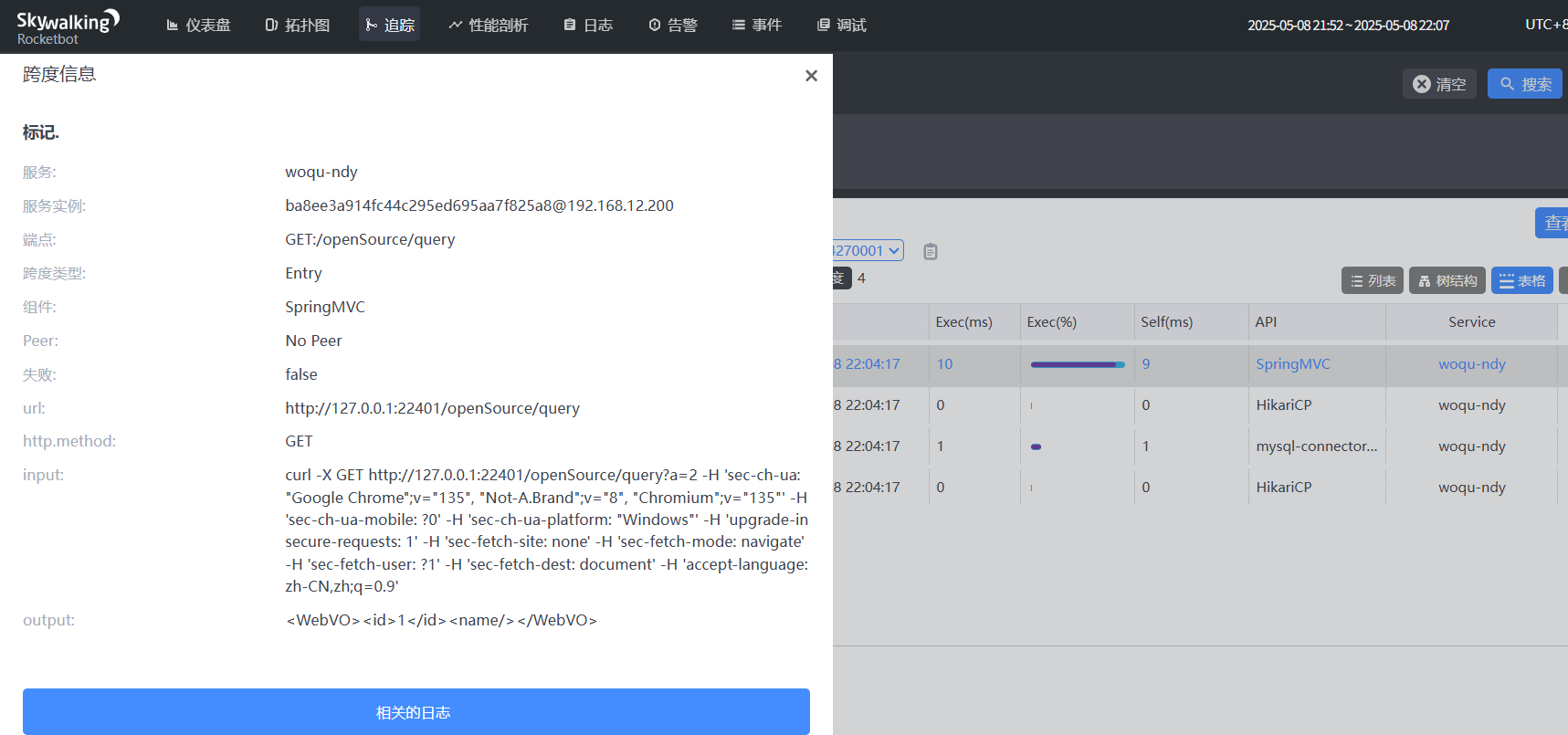
我们还想看到入参出参



增加拦截类



查看跨度信息



就可以看到出参入参了

# 更换为Nacos

下载地址

<https://nacos.io/download/release-history/?spm=5238cd80.2ef5001f.0.0.3f613b7cCJFvMM>

默认端口是8848

我们需要修改一下bin下面的startup。Cmd文件set MODE="cluster",替换为set MODE="standalone"，然后启动即可

Web登录url [http://127.0.0.1:8848/nacos](http://127.0.0.1:8848/nacos" \o "http://127.0.0.1:8848/nacos)

账号密码都是nacos

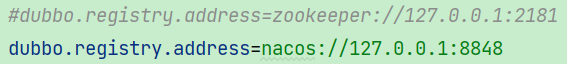


然后我们把dubbo的注册中心从zookeeper变成nacos

注释掉zoo的pom，引入nacos的，nacos版本和dubbo版本保持一致

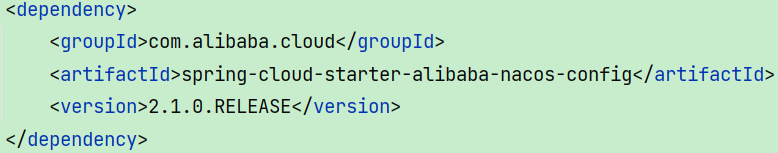


修改配置文件



Nacos就具有服务发现和配置中心的能力，完全可以替代eureka+apollo的这一套。

来实践一下配置



新增配置文件



在nacos中新增配置



示例中读取即可



说明一下nacos文件读取规则

${spring.application.name}-${spring.profiles.active}.${spring.cloud.nacos.config.file-extension}作为文件id，来读取配置。组和命名空间那些也是都支持的。

配置共享的话通过命名来控制，名称中只有${spring.application.name}.${spring.cloud.nacos.config.file-extension}就会被所有的环境匹配上。优先级的话如下



总的来说nacos比起apollo，运维是要明显轻松很多的，apollo还需要数据库支持，同时在性能上nacos也更好一些。同时nacos还有服务发现功能，更加全面。

# 介绍Shardingsphere

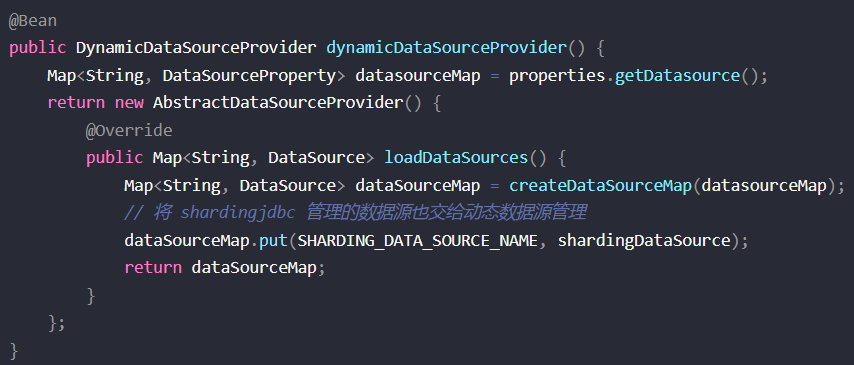
分库分表可以使用这个组件

官网位置

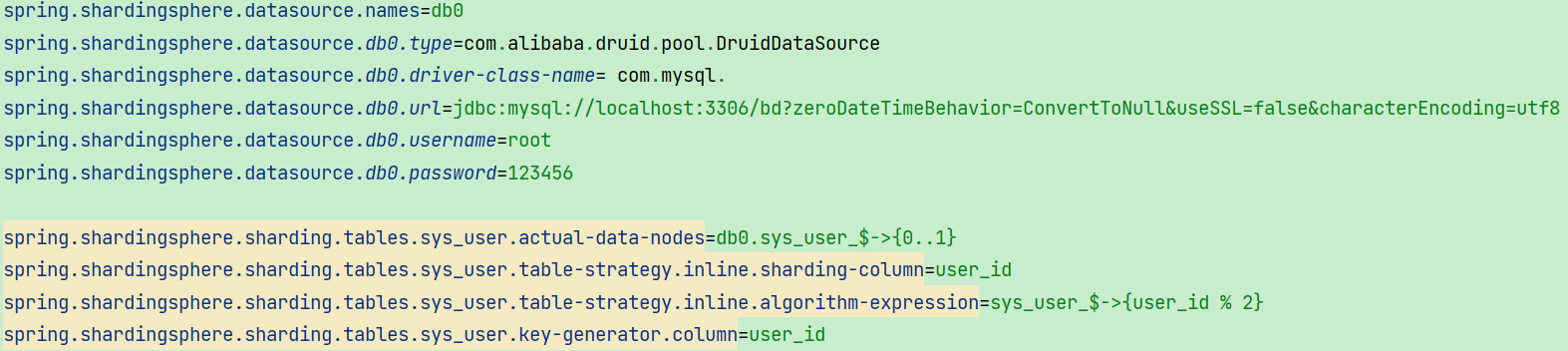
<https://shardingsphere.apache.org/document/current/cn/user-manual/shardingsphere-jdbc/yaml-config/rules/readwrite-splitting/>

我们前面集成了dynamic，shardingsphere也可以作为数据源放在dunamic中进行使用

在配置文件中将shardingsphere解析为数据源加入到dynamic中即可。



配置可以参考

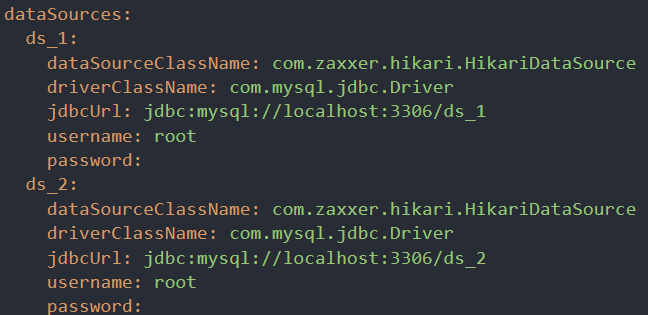


说一下他们之间的关系dynamic是mybatis的插件类似于，引入就可以直接用，用注解去标识操作哪个数据库，shardingsphere是开源分库分表项目，可以单独使用，也可以集成进去

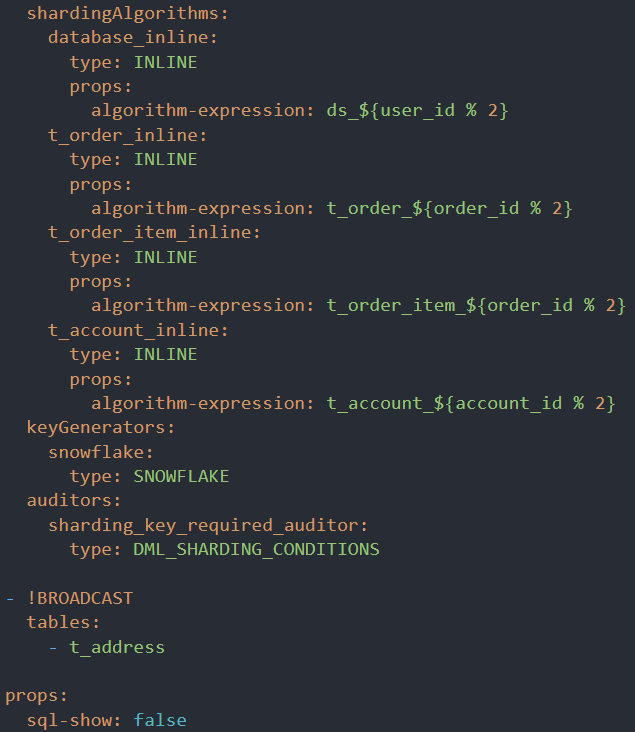
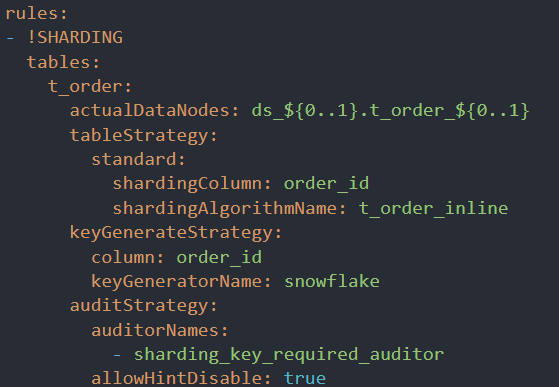
dynamic作为一个特殊的数据源。

shardingsphere有两个部分，jdbc和proxy，前者是轻量java框架，强化版的驱动，集成就可以使用通过yml来完成复杂的分库分表逻辑。proxy以代理的形式部署在应用程序与数据库之间，是一个单独的服务，实现了对 SQL 的解析和改写以及请求的转发，主要在异构环境中使用。Jdbc是嵌入式的，proxy则是完全的在拦截。

Yml中需要写明有哪些数据库



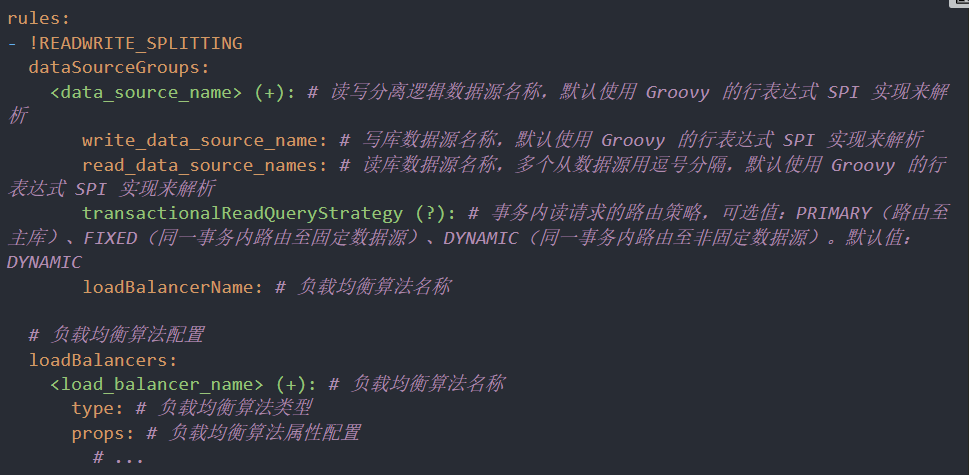
分片规则是什么



示例上是分片表，分片表sql可以不带分区键变成全分片查询。还有广播表，即每个分片都有并且完全相同的表数据。还有关联表也就是绑定表，是把一些会做join操作的表放在一组，因为他们完全可以本地join，因为在一个节点上的都是一个分区键，所以分区键就是join建的话可以极大的加快sql速度。

读写分离配置。

原理就是根据你的配置和所执行sql的类型，去做sql的改写，最后合并，我没看都能猜出来。



还可以数据加密、脱敏。一般是数据库中存加密数据，数据返回前执行脱敏流程。

Shardingsphere支持同库join，不支持异库join。如果你想做异库join可能需要自己在应用层去join了。或者把经常join的表变成广播表。还可以使用hit机制，自己决定sql路由到哪里自己写逻辑。

Shardingsphere中执行sql也是有限制的，毕竟是分布式的，肯定有些sql、或者函数、还有DDL等执行都会比较麻烦，这种开源的东西只能说根本就不好用。除非你去把人家那一套数据库维护都买回来，不然应该用着很痛苦。大家还是去拥抱天生的分布式新数据库比较好一些真的。

Shardingsphere也支持事务，local、XA、BASE

Local就是本地事务，直接用spring的都可以

XA则需要引入，前提是数据库自己实现了XA接口



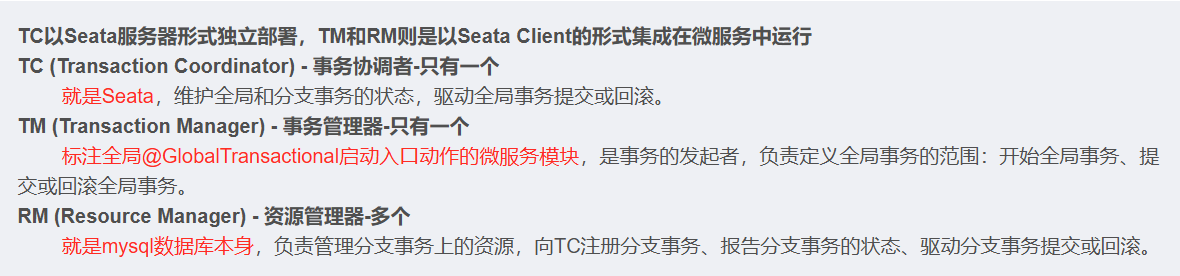
配置事务管理器

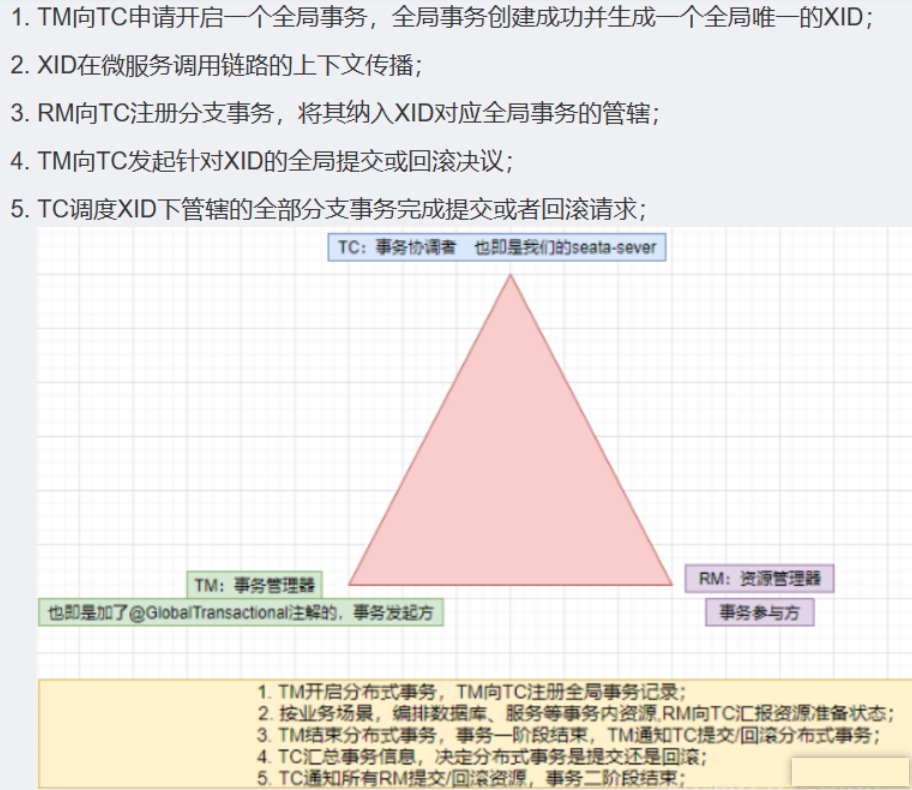
使用则是通过注解

@ShardingTransactionType(TransactionType.XA) // 支持TransactionType.LOCAL, TransactionType.XA, TransactionType.BASE

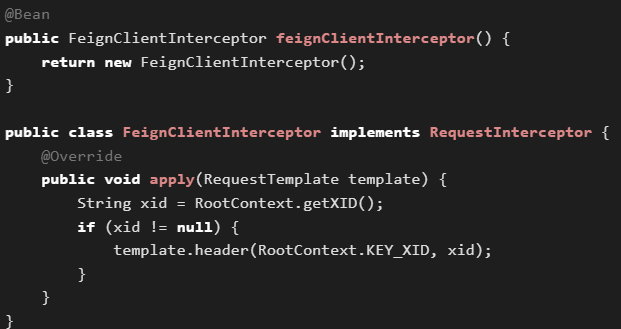
Base则是弱事务，即最终一致性，实现依托seata服务，这里我们就提一下，后面也会集成。

# 调研Seata





回答一个关键问题，如何把多个微服务调用的db也纳入到一个事务中的，是这样做的直接在feign拦截，把XID作为请求头传给了之后的微服务



一旦 XID 到达被调用的服务，Seata 的 RootContext 类会负责将这个 XID 绑定到当前线程的上下文中。

提供四种事务方式

AT默认：

通过代理数据源控制undo log来完成提交和回滚

核心逻辑：

一阶段：在业务SQL对应数据源的`undo\_log`表里保存业务数据更新前(before image)和更新后(after image)的数据镜像

二阶段：提交，一阶段实际上已经提交了，这里仅是把`undo\_log`记录删除

三阶段：回滚，将`undo\_log`里更新后的数据镜像(after image)和实际数据对比，如果一致则回滚（用before image更新数据），不一致说明有脏写，需要人工删除记录并优化业务代码.

TCC：

实现try、confirm、cancel方法来保证事务，比较复杂，还需要幂等性

SAGA：

通过长事务分割为小事务来实现，补偿逻辑复杂，需业务开发者实现

XA：

基于XA协议来实现

其中AT和XA都需要依赖你的数据库自己实现了标准接口

整合seata之后我们的DSTransactional就会失效，应该用GlobalTransactional去代替它。

同时enable-auto-data-source-proxy要设置为false，否则数据源切换会失败，原因是Seata和DS都用了DataSourceProxy，Seata的Bean覆盖了DS的Bean

总结一下之前多数据整合和这一章中对事务的使用我们可以知道：

1.@Transactional注解方法内，无法切换数据源

2.管理MongoDB事务需要使用配置的事务管理器:

配置后，管理MongoDB事务需要在事务方法上加上@Transactional(transactionManager = "mongoTransactionManager")，注意，通过@Transactional无法同时实现JDBC事务管理和MongoDB事务管理。

3. @DSTransactional注解用于管理多数据源事务，但是整合Seata后，@DSTransactional会失效，需要使用Seata的注解@GlobalTransactional来管理事务

4. Seata AT模式，整个事务会被分配一个XID，@GlobalTransactional注解的方法下，一般来讲，每个SQL操作都会被分配一个branchId（执行方法时打断点，在undo\_log表里可以查看所有的branchId，执行完后记录一般会被删除），但是如果调用的子方法含有@Transactional注解，则该子方法内的所有SQL操作只会被分配一个branchId。

5. Seata事务回滚时，会回滚undo\_log表里的每个branch，每个branch会保存一份更改前和更改后的数据镜像，回滚时会检查更改后镜像和实际数据，一致则会回滚，不一致则不会回滚并抛出异常，所以编写业务代码时应该保证一次Seata事务内的对同一行表数据修改的操作被分配在同一个branch内（用一条SQL语句解决，或者包含在@Transactional注解的方法内）(注意：调用@Transactional方法的父方法如果在同一个类里，由于同类方法内互相调用不会用到代理对象，所以@Transactional等于无效)