链滴

Spring 嵌入式轻量消息队列

作者：crick77

原文链接：https://ld246.com/article/1588682208829

来源网站：链滴

为Spring-boot提供消息队列能力的starter, 并提供了VM线程的轻量级实现。

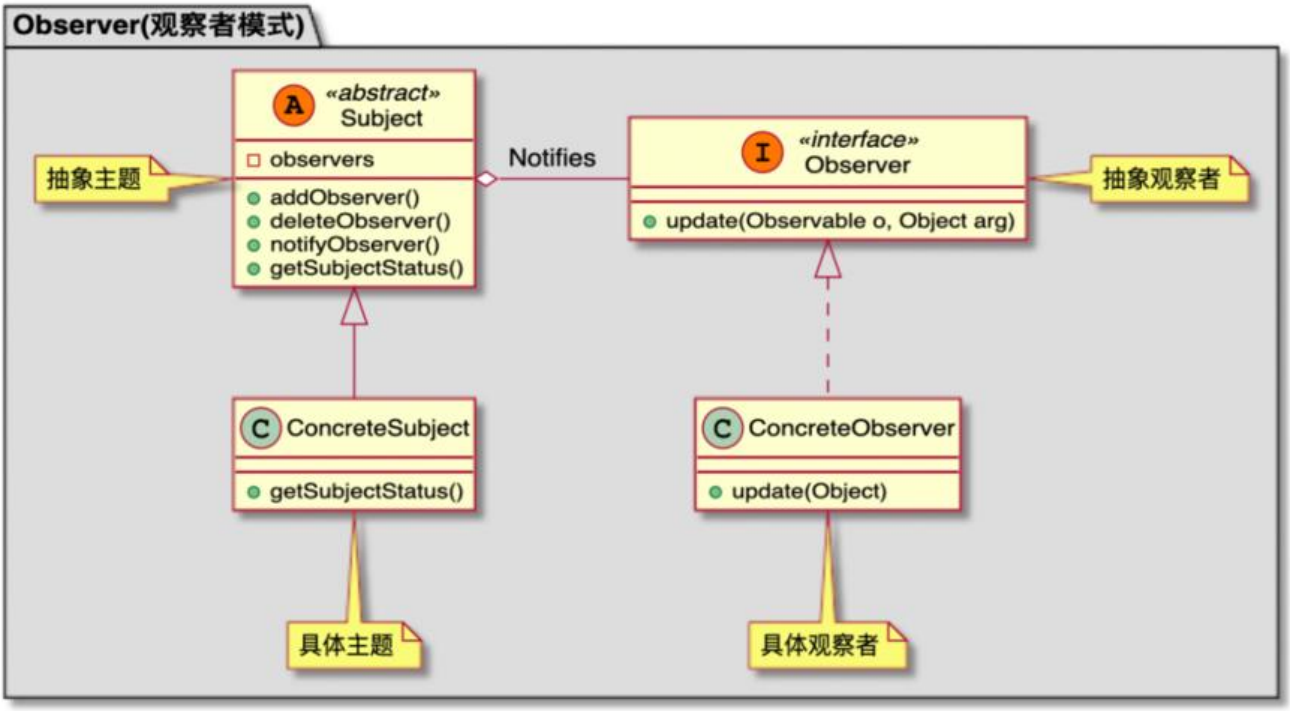
项目地址: https://github.com/wangyuheng/embedded-mq-spring-boot-starter

什么是消息队列

消息队列是用于存放消息的容器，可供消费者取出消息进行消费。

观察者模式

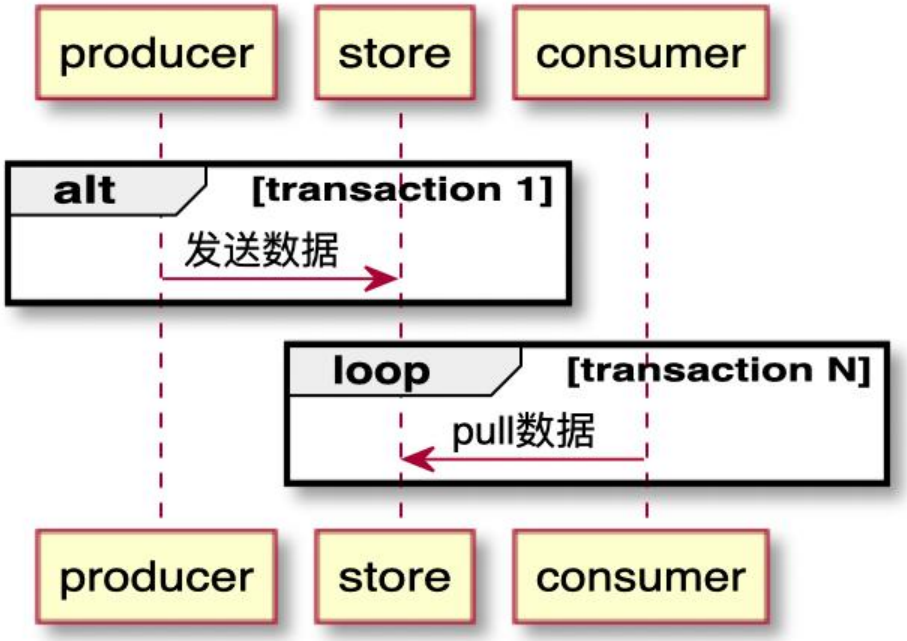
观察者（Observer）模式的定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模，它是对象行为型模式。



Observer本来的意思是**观察者**，但具体的实现中并不是主动去观察，而是被动的接收来自 Subject的知，所以更合适的名字应该是"消息投递"。

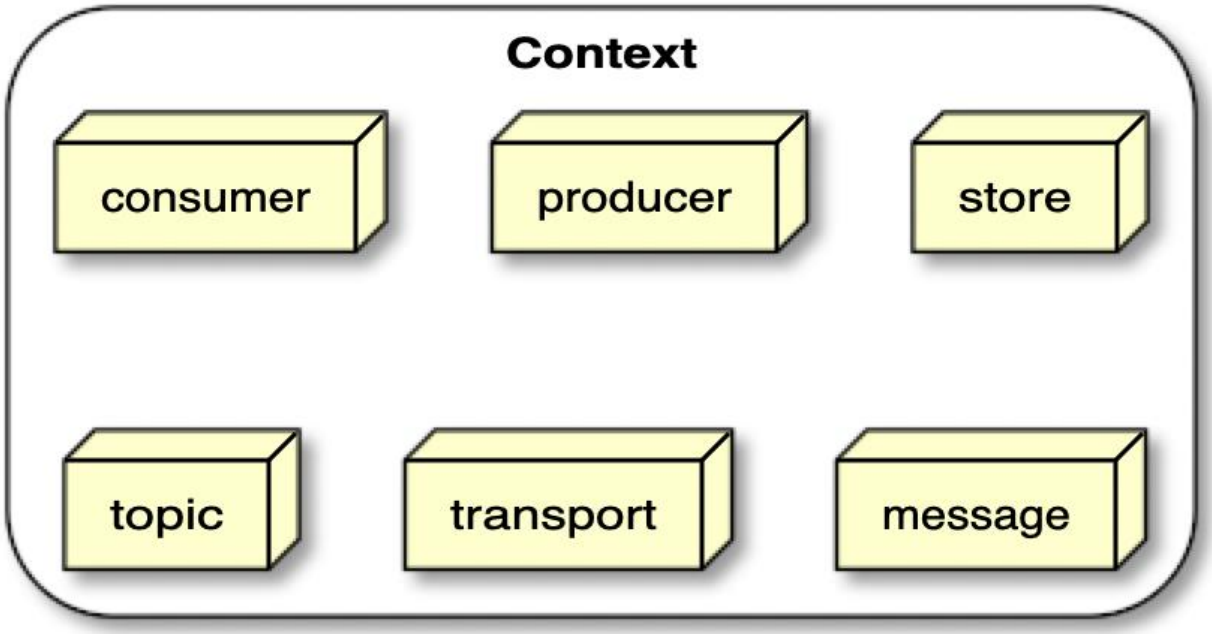
而且通知的模式还存在一个**弊端**: 通知及多个 ConcreteObserver的消费程序仍在一个同步线程内，以只是代码结构层面的解耦，底层还是一个事务内。

为了解决这个**弊端**，将消息的发送及N个消费程序拆分为N+1个事务，所以引入消息队列用于存储 Subject。

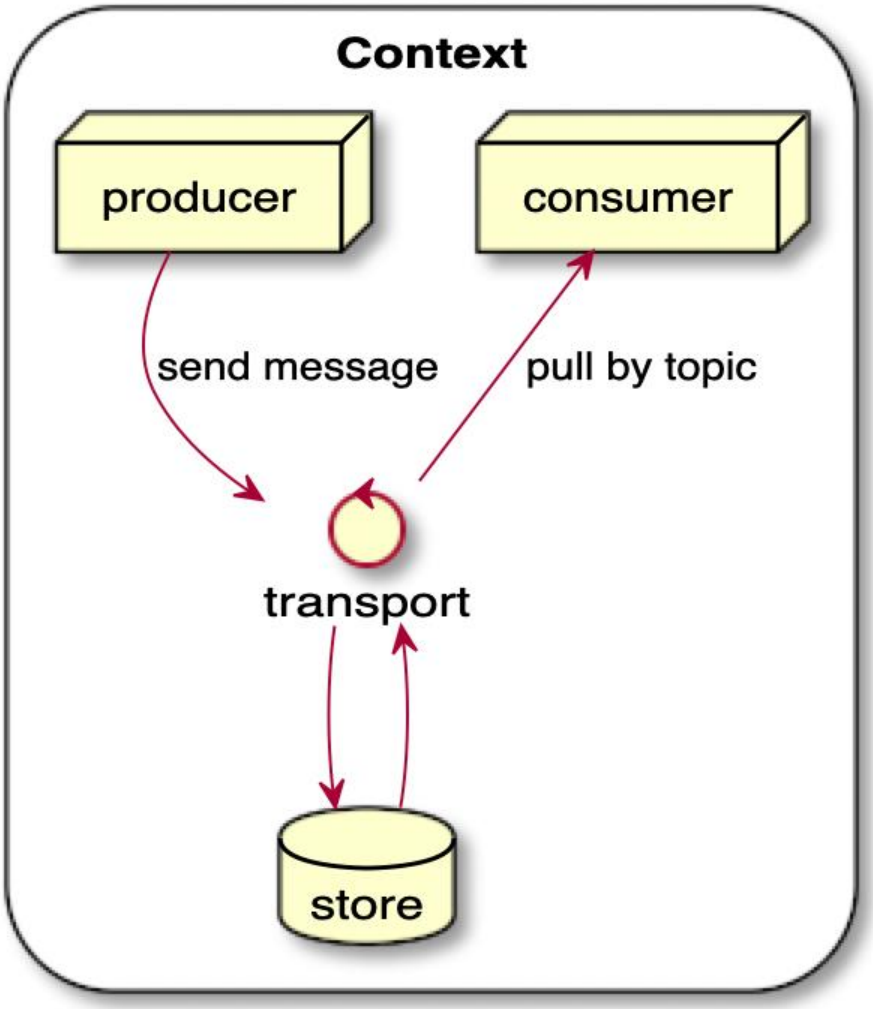


领域模型设计

- 罗列领域概念



- 梳理交互关系



代码实现

1. LinkedBlockingQueue 作为存储Message的容器。

2. Store用于存储消息。为了兼容多个 Consumer，每个 Consumer指定一个唯一标识作为 Partition Key，对应唯一的一个 LinkedBlockingQueue。 e.g. Map<Partition, LinkedBlockingQueue<Message>> messageQueueMap = new ConcurrentHashMap<>();

3. Producer 通过 Transport 将消息发送只多个 Partition Key的 LinkedBlockingQueue队列中

4. 每个 Consumer开启一个线程，通过轮询方式从 LinkedBlockingQueue队列中消费消息。

代码片段

- VmStore

```
    private Map<Partition, LinkedBlockingQueue<Message>> messageQueueMap = new Con
urrentHashMap<>();

    @Override
    public void append(Message message, Partition partition) {
        initQueue(partition);
        messageQueueMap.get(partition).add(message);
```

原文链接：Spring 嵌入式轻量消息队列

```
    }

    @Override
    public LinkedBlockingQueue<Message> findByPartition(Partition partition) {
        initQueue(partition);
        return messageQueueMap.get(partition);
    }

    private void initQueue(Partition partition) {
        if (!messageQueueMap.containsKey(partition)) {
            synchronized (this) {
                if (!messageQueueMap.containsKey(partition)) {
                    messageQueueMap.put(partition, new LinkedBlockingQueue<>());
                }
            }
        }
    }
```

- Transport

```
public void transfer(Message message) {
    final String topic = message.getTopic();
    topicClientIdMap.get(topic).forEach(clientId -> {
        Partition partition = new Partition(clientId, topic);
        store.append(message, partition);
    });
}
```

- ConsumerCluster

```
/**
 * 开启消费线程
 *  只能开启一次
 */
public synchronized void start(Store store) {
    if (!initialized.get()) {
        synchronized (this) {
            SimpleAsyncTaskExecutor taskExecutor = new SimpleAsyncTaskExecutor();
            taskExecutor.setDaemon(true);
            taskExecutor.execute(new ConsumerListener(this.getMessageHandler(), store.findByPa
tition(this.generatePartition())));
            initialized.set(true);
        }
    }
}

/**
 * 关闭消费线程
 */
public void shutdown() {
    liveToggle.set(false);
}

/**
```

```
     * 暂停消费
     */
    public void pause() {
        runToggle.set(false);
    }

    /**
     * 重启暂停的消费线程
     */
    public void restart() {
        runToggle.set(true);
    }

    class ConsumerListener implements Runnable {

        private MessageHandler handler;
        private LinkedBlockingQueue<Message> queue;

        ConsumerListener(MessageHandler handler, LinkedBlockingQueue<Message> queue) {
            this.handler = handler;
            this.queue = queue;
        }

        @Override
        public void run() {
            while (true) {
                try {
                    if (!liveToggle.get()) {
                        break;
                    }
                    if (runToggle.get()) {
                        Message message = queue.poll();
                        if (null == message) {
                            Thread.sleep(100);
                        } else {
                            handler.handle(message);
                        }
                    } else {
                        Thread.sleep(100);
                    }
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

使用 LinkedBlockingQueue却未使用 take方法的原因是为了灵活控制消费线程的启停。

spring集成

为了方便使用，通过 annotation的形式与spring框架进行集成。

示例

- Consumer

```
@Consumer(topic = CONSUMER_TOPIC, id = CUSTOM_CONSUMER_ID)
public void consumerMessage(Message message) {
    consumerRecordMap.get(CUSTOM_CONSUMER_ID).add(message);
}
```

- Producer

```
@Autowired
private DefaultProducer<String> producer;

public void sendMessage(){
    producer.send(new Message<>(CUSTOM_TOPIC, "This is a message!"));
}
```

代码实现

```
/**
 * 注册消费者bean
 *
 * @see Consumer
 * @see MessageHandler
 * @see Store
 */
public class ConsumerBeanDefinitionRegistryPostProcessor implements BeanPostProcessor,
pplicationContextAware {

    private ConfigurableApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansExc
ption {
        this.applicationContext = (ConfigurableApplicationContext) applicationContext;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansE
ception {
        Class<?> targetClass = AopProxyUtils.ultimateTargetClass(bean);
        Method[] methods = ReflectionUtils.getAllDeclaredMethods(targetClass);
        for (Method method : methods) {
            if (AnnotatedElementUtils.hasAnnotation(method, Consumer.class)) {
                final String topic = method.getAnnotation(Consumer.class).topic();
                final String id = StringUtils.isEmpty(method.getAnnotation(Consumer.class).id()) ? b
anName + method.getName() : method.getAnnotation(Consumer.class).id();
                final BeanFactory beanFactory = applicationContext.getBeanFactory();
                final Store store = beanFactory.getBean(Store.class);

                final MessageHandler messageHandler = message -> ReflectionUtils.invokeMethod
method, bean, message);
```

```
                final BeanDefinitionBuilder beanDefinitionBuilder = BeanDefinitionBuilder.genericB
anDefinition(ConsumerCluster.class, () -> {
                    ConsumerCluster consumerCluster = new ConsumerCluster();
                    consumerCluster.setId(id);
                    consumerCluster.setTopic(topic);
                    consumerCluster.setMessageHandler(messageHandler);
                    consumerCluster.start(store);
                    return consumerCluster;
                });
                BeanDefinition beanDefinition = beanDefinitionBuilder.getRawBeanDefinition();
                ((DefaultListableBeanFactory) beanFactory).registerBeanDefinition(beanName + me
thod.getName() + "Listener", beanDefinition);
            }
        }
        return bean;
    }
}
```

其他

1. 如何跨应用消费？通过Mysql、Redis等公共存储替换Store及Transport实现。Mysql需要考虑行
。