# 目录

[目录 1](#_Toc477438931)

[1. 架构介绍 2](#_Toc477438932)

[1. 架构图 2](#_Toc477438933)

[2. 包结构 2](#_Toc477438934)

[2. 功能介绍 2](#_Toc477438935)

[3. 开发指南 3](#_Toc477438936)

[1. 引入依赖 3](#_Toc477438937)

[2. Spring配置 3](#_Toc477438938)

[3. 接口说明 5](#_Toc477438939)

[1. MqProducer<T> 5](#_Toc477438940)

[2. MqCallback<T> 5](#_Toc477438941)

[3. MqConsumer 5](#_Toc477438942)

[4. MqListener 6](#_Toc477438943)

[5. @Producer 6](#_Toc477438944)

[6. @Consumer 6](#_Toc477438945)

[7. @TxMessage 6](#_Toc477438946)

[4. Kafka 6](#_Toc477438947)

[5. 高优先级发送 7](#_Toc477438948)

[1. 中优先级发送 8](#_Toc477438949)

[2. 低优先级发送 8](#_Toc477438950)

[3. 高优先级接收 8](#_Toc477438951)

[5. 中优先级接收 8](#_Toc477438952)

[6. 低优先级接收 9](#_Toc477438953)

[6. Activemq 9](#_Toc477438954)

[4. 配置介绍 9](#_Toc477438955)

[5. 性能数据 9](#_Toc477438956)

# 架构介绍

## 架构图



## 包结构

pergesa-event：持久化事务框架

pergesa-msg-core：消息接口

pergesa-msg-kafka：kafka客户端实现

pergesa-msg-amq：activemq客户端实现

# 功能介绍

**统一消息框架：**基于消息中件间特点抽象出一套统一的消息发送、接收接口，简化使用。

**消息两阶段提交：**基于持久化事件框架的模拟消息两阶段提交，可以确保DB操作和消息发送的一致性。

**消息灰度：**基于发送端的灰度方案，可以随时切换生产环境和灰度环境的消息流量。

# 开发指南

## 引入依赖

<**dependency**>  
 <**groupId**>com.arto</**groupId**>  
 <**artifactId**>pergesa-event</**artifactId**>  
 <**version**>最新版本号</**version**>  
</**dependency**>

<**dependency**>  
 <**groupId**>com.arto</**groupId**>  
 <**artifactId**>pergesa-msg-core</**artifactId**>  
 <**version**>最新版本号</**version**>  
</**dependency**>

*<!— 使用Kafka时引入 -->*

<**dependency**>  
 <**groupId**>com.arto</**groupId**>  
 <**artifactId**>pergesa-msg-kafka</**artifactId**>  
 <**version**>最新版本号</**version**>  
</**dependency**>

*<!— 使用Amq时引入 -->*

<**dependency**>  
 <**groupId**>com.arto</**groupId**>  
 <**artifactId**>pergesa-msg-amq</**artifactId**>  
 <**version**>最新版本号</**version**>  
</**dependency**>

## Spring配置

必须配置如下：

*<!— 事件配置 -->*  
<**import resource="classpath:pergesa-event.xml"**/>

可选配置如下(如不需要使用事务消息功能，以下配置可以忽略)：

*<!-- 持久化事件恢复Job的ZK注册中心 -->*<**reg:zookeeper id="regCenter" serverLists="${serverLists}"  
 namespace="${namespace}"  
 baseSleepTimeMilliseconds="${baseSleepTimeMilliseconds}"  
 maxSleepTimeMilliseconds="${maxSleepTimeMilliseconds}"  
 maxRetries="${maxRetries}"** />  
  
*<!-- 持久化事件恢复Job -->*<**job:bean id="persistentEventRecoveryJob" class="com.arto.event.recovery.PersistentEventRecoveryJob" regCenter="regCenter"  
 shardingTotalCount="10"  
 cron="0 0/10 \* \* \* ? \*"  
 failover="true"  
 description="Persistent event recovery job"** />

*<!-- 持久化事件清理Job，默认清理修改时期30天以前的数据-->*<**job:bean id="persistentEventClearJob" class="com.arto.event.schedule.PersistentEventClearJob" regCenter="regCenter"  
 shardingTotalCount="1"  
 cron="0 10 3 \* \* ?"  
 shardingItemParameters="0=30"  
 failover="true"  
 description="Persistent event clear job"** />

*<!-- 持久化事件的jdbcTemplate，数据源必须与业务库一致 -->*

<**bean id="jdbcTemplate" class="org.springframework.jdbc.core.namedparam**

**.NamedParameterJdbcTemplate"**>

<**constructor-arg ref="dataSource"** />

</**bean**>  
  
<**bean id="transactionManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"** />  
</**bean**>  
  
<**tx:annotation-driven transaction-manager="transactionManager"**/>

如需使用事务消息，需要确认与事务消息同一个事务的业务DB中存在如下表：

**CREATE TABLE** event\_storage  
(  
 **id** bigserial,  
 **tag smallint**,  
 **system\_id character varying**(64),  
 **business\_id character varying**(64),  
 **business\_type character varying**(64),  
 **EVENT\_TYPE character varying**(128),  
 **status smallint**,  
 **payload** text,  
 **retried\_count\_d smallint**,  
 **retried\_count\_c smallint**,  
 **next\_retry\_time timestamp** without **time zone**,  
 **memo** text,  
 **gmt\_created timestamp** without **time zone**,  
 **gmt\_modified timestamp** without **time zone**,  
 **CONSTRAINT** event\_storage\_pkey **PRIMARY KEY** (**id**, **tag**, **system\_id**)  
);  
  
**create** index idx\_event\_storage\_ts **on** event\_storage(status);  
**create** index idx\_event\_storage\_gm **on** event\_storage(gmt\_modified);  
  
**commit**;

## 接口说明

#### MqProducer<T>

接口统一的消息发送接口

|  |  |
| --- | --- |
| **方法** | **void** send(T message) **throws** MqClientException |
|  | 发送消息(简化方法，非事务类消息可以使用直接使用此方法发送) |
| **方法** | **void** send(MessageRecord<T> record) **throws** MqClientException |
|  | 发送消息(简化方法，非事务类消息可以使用直接使用此方法发送) |
| **方法** | **void** sendNonTx(MessageRecord<T> record) **throws** MqClientException |
|  | 发送非事务消息(开启事务发送后，可使用此方法发送非事务消息) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | **参数名** | **类型** | **必录** | **说明** |
| MessageRecord | properties | Map<String, Object> | 否 | 自定义消息头，可接受Boolean、Int、Long、String类型的数据。 |
| businessId | String | 否 | 业务凭证流水号(事务消息时必填) |
| businessType | String | 否 | 业务类型(事务消息时必填) |
| messageId | String | 否 | 框架生成的消息ID，无需设定 |
| message | T | 是 | 具体消息 |

#### MqCallback<T>

异步发送时需实现此接口

|  |  |
| --- | --- |
| **方法** | 无 |

#### MqConsumer

统一的消息接收接口

|  |  |
| --- | --- |
| **方法** | 无 |

#### MqListener

统一的消息接收监听器

|  |  |
| --- | --- |
| **方法** | **void** onMessage(MessageRecord<T> record) |
|  | 收到消息时的实际处理逻辑 |
| **方法** | **boolean** checkRedeliver(MessageRecord<T> record) |
|  | 该消息是否已经处理，为True则跳过此消息 |

#### @Producer

消息发送注解

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **类型** | **必填** | **默认** | **说明** |
| type | MqTypeEnum | 否 | KAFKA | 消息中间件类型，分为KAFKA、ACTIVEMQ |
| destination | String | 是 | - | 消息目的地 |
| priority | MessagePriorityEnum | 否 | HIGH | 优先级，分为HIGH、MEDIUM、LOW |
| callback | Class | 否 | - | 异步回调 |

#### @Consumer

消息接收注解

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **参数** | **类型** | **必填** | **默认** | **说明** |
| type | MqTypeEnum | 否 | KAFKA | 消息中间件类型，分为KAFKA、ACTIVEMQ |
| destination | String | 是 | - | 消息目的地 |
| priority | MessagePriorityEnum | 否 | HIGH | 优先级，分为HIGH、MEDIUM、LOW |
| checkRedeliver | String | 否 | - | 去重检测方法名(方法返回true为重复消息) |
| numThreads | int | 否 | 2 | Kafka：并发线程数  Amq：1. 决定并行消费者数(同”concurrentConsumers”)  2.决定消费者线程数(coreSize=Math.ceil(numThreads() \* 1.3)， maxSize=Math.ceil(coreSize \* 1.5)) |
| batchSize | int | 否 | 5 | Kafka：批量接交消费标识  Amq：等同于”prefetchSize” |

#### @TxMessage

事务消息注解，无参数。

## Kafka

**Kafka发送时可定制的参数**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **分类** | **参数名** | **类型** | **必录** | **说明** |
| KafkaMessageRecord  (继承至MessageRecord) | key | String | 否 | Kafka根据key计算hash分配到不同的分片中 |
| partition | int | 否 | 发送到指定的分片中 |

实现原理：

**注意：没有在配置文件中设定” default.mq.type”参数或指定” default.mq.type=kafka”时，使用注解发送和接收消息时都可以省略” type = MqTypeEnum.KAFKA”，否则必须加上。**

#### 高优先级发送

对应Kafka的ack=all类型的消息发送，非不可抗力消息不会丢失，但效率较低，重要类消息可使用高优先级发送。此类型可开启事务发送，事务消息(使用两阶段提交的消息)，保证JDBC事务和消息发送的数据一致性。事务消息需设定在JDBC方法前加上”@TxMessage”，并保证开启事务。例：

@Service  
**public class** OrderServiceImpl **implements** OrderService{  
  
 @Producer(destination = **"test-high"**, priority = MessagePriorityEnum.***HIGH***)  
 **private** MqProducer<OrderDO> **producer**;  
@Autowired  
 **private** OrderDao **dao**;  
  
 @Override  
 @TxMessage  
 @Transactional  
 **public boolean** addOrder(OrderDO orderDO) {  
 **int** result = **dao**.addOrder(orderDO);  
 sendMessage(orderDO);  
 **return** 0 < result;  
 }  
  
 @Override  
 @Transactional  
 **public boolean** cancelOrder(OrderDO orderDO){  
 **return** 0 < **dao**.deleteOrder(orderDO);  
 }  
  
 **private void** sendMessage(OrderDO orderDO) {  
 **producer**.send(**new** MessageRecord<OrderDO>(**"oid"** + orderDO.getOrderId(),

**"order"**, orderDO));

*//如果已开启事务发送，但有消息不需要事务发送时可使用sendNonTx方法*

*//producer.sendNonTx(new MessageRecord<OrderDO>("oid" + orderDO.getOrderId(), "order", orderDO));*

}

}

#### 中优先级发送

对应Kafka的ack=1类型的消息发送，Kafka服务器宕机有几率丢失消息，一般类的消息可以使用此优先级发送。 例：

@Producer(destination = **"test-medium"**, priority = MessagePriorityEnum.***MEDIUM***)  
**private** MqProducer<OrderDO> **producer**;

#### 低优先级发送

对应Kafka的ack=0类型的消息发送，速度最快，客户端或Kafka服务器出现问题都可引起消息丢失。大批量、可容忍丢失的消息推荐此优先级。例：

@Producer(destination = **"test-low"**, priority = MessagePriorityEnum.***LOW***)  
**private** MqProducer<OrderDO> **producer**;

#### 高优先级接收

消息处理出错重试3次后持久化继续消费后面的消息，由调度任务重试此消息。例：

@Consumer(destination = **"test-high"**, priority = MessagePriorityEnum.***HIGH***, checkRedeliver = **"checkRedeliver"**)  
**public void** consumerHigh(MessageRecord<OrderDO> message){  
 System.***out***.println(**"ConsumerService.consumerHigh message:"** + message);  
}

**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

#### 中优先级接收

消息处理出错后重试三次后丢弃，适合于容忍丢失类型的消息。例：

@Consumer(destination = **"test-medium"**, priority = MessagePriorityEnum.***MEDIUM***

) , checkRedeliver = **"checkRedeliver"**)  
**public void** consumerLow(MessageRecord<OrderDO> message){  
 System.***out***.println(**"consumerMedium message:"** + message);  
}  
  
**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

#### 6. 低优先级接收

消息处理出错后自动丢弃，适合于容忍丢失类型的消息。例：

@Consumer(destination = **"test-low"**, priority = MessagePriorityEnum.***LOW***) , checkRedeliver = **"checkRedeliver"**)  
**public void** consumerLow(MessageRecord<OrderDO> message){  
 System.***out***.println(**"consumerLow message:"** + message);  
}  
  
**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

## Activemq

实现原理：

Activemq的发送使用的是JmsTemplate，接收使用的是DefaultMessageListenerContainer，将原来需要手动在XML中的配置全部改为通过Spring的BeanDefinitionRegistry动态注册。

**注意：在配置文件中指定” default.mq.type=activemq”时，使用注解发送和接收消息时都可以省略” type = MqTypeEnum.ACTIVEMQ”，否则必须加上。**

#### 高优先级发送

对应Activemq的普通发送模式，在没有强制开启异步发送模式时，所有持久化消息非不可抗力消息不会丢失。此类型可开启事务发送，事务消息(使用两阶段提交的消息)，保证JDBC事务和消息发送的数据一致性。事务消息需设定在JDBC方法前加上”@TxMessage”，并保证开启事务，但效率较低，重要类消息可使用高优先级发送。例：

@Service  
**public class** OrderServiceImpl **implements** OrderService{  
  
 @Producer(destination = **"test-high"**, priority = MessagePriorityEnum.***HIGH***)  
 **private** MqProducer<OrderDO> **producer**;  
@Autowired  
 **private** OrderDao **dao**;  
  
 @Override  
 @TxMessage  
 @Transactional  
 **public boolean** addOrder(OrderDO orderDO) {  
 **int** result = **dao**.addOrder(orderDO);  
 sendMessage(orderDO);  
 **return** 0 < result;  
 }  
  
 @Override  
 @Transactional  
 **public boolean** cancelOrder(OrderDO orderDO){  
 **return** 0 < **dao**.deleteOrder(orderDO);  
 }  
  
 **private void** sendMessage(OrderDO orderDO) {  
 **producer**.send(**new** MessageRecord<OrderDO>(**"oid"** + orderDO.getOrderId(),

**"order"**, orderDO));

*//如果已开启事务发送，但有消息不需要事务发送时可使用sendNonTx方法*

*//producer.sendNonTx(new MessageRecord<OrderDO>("oid" + orderDO.getOrderId(), "order", orderDO));*

}

}

#### 中优先级发送

对应Activemq的普通发送模式，在不开启消息二阶段提交时同高优先级发送，一般类的消息可以使用此优先级发送。 例：

@Producer(destination = **"test-medium"**, priority = MessagePriorityEnum.***MEDIUM***)  
**private** MqProducer<OrderDO> **producer**;

#### 低优先级发送

对应Activemq的普通发送模式，使用多线程异步发送，速度最快，客户端或服务器出现问题都可引起消息丢失。大批量、可容忍丢失的消息推荐此优先级。例：

@Producer(destination = **"test-low"**, priority = MessagePriorityEnum.***LOW***)  
**private** MqProducer<OrderDO> **producer**;

#### 高优先级接收

对应Activemq的”SessionAcknowledgeMode=Session.SESSION\_TRANSACTED”。

消息处理出错重试3次后持久化继续消费后面的消息，由调度任务重试此消息。例：

@Consumer(destination = **"test-high"**, priority = MessagePriorityEnum.***HIGH***, checkRedeliver = **"checkRedeliver"**)  
**public void** consumerHigh(MessageRecord<OrderDO> message){  
 System.***out***.println(**"ConsumerService.consumerHigh message:"** + message);  
}

**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

#### 中优先级接收

对应Activemq的”SessionAcknowledgeMode=Session.SESSION\_TRANSACTED”，并启用多线程消费。消息处理出错后重试三次后丢弃，适合于容忍丢失类型的消息。例：

@Consumer(destination = **"test-medium"**, priority = MessagePriorityEnum.***MEDIUM***

) , checkRedeliver = **"checkRedeliver"**)  
**public void** consumerLow(MessageRecord<OrderDO> message){  
 System.***out***.println(**"consumerMedium message:"** + message);  
}  
  
**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

#### 6. 低优先级接收

对应Activemq的”SessionAcknowledgeMode=Session.SESSION\_TRANSACTED”，并启用多线程消费。消息处理出错后自动丢弃，适合于容忍丢失类型的消息。例：

@Consumer(destination = **"test-low"**, priority = MessagePriorityEnum.***LOW***) , checkRedeliver = **"checkRedeliver"**)  
**public void** consumerLow(MessageRecord<OrderDO> message){  
 System.***out***.println(**"consumerLow message:"** + message);  
}  
  
**public boolean** checkRedeliver(MessageRecord<OrderDO> message){  
 **return** redeliver; *// true: 重复消息 false: 非重复消息*}

# 配置介绍

非必填项请谨慎修改。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **属性名** | **类型** | **必填** | **默认值** | **描述** |
| sar.name | String | 是 | - | 系统唯一标识，不能与其它系统重复(恢复Job使用) |
| default.mq.type | Int | 是 | kafka | 默认消息中间件类型，在创建生产者或消息者时如果没有选技消息中间件，将采用默认类型生产或消费 |
| bootstrap.servers | String | 是 | - | Kafka集群地址 |
| group.id | String | 是 | - | Kafka消费组标识，建议同sar.name |
| client.id | String | 是 | - | Kafka生产组标识，建议同sar.name |
| mq.pipeline.size | int | 否 | 30000 | 事务消息发送时的缓存队列长度 |
| kafka.producer.timeout | int | 否 | 10 | 秒，使用kafka发送消息时的超时时间 |
| kafka.message.maxsize | int | 否 | 1048576 | 字节，限制kafka最大发送消息Size |
| activemq.message.maxsize | int | 否 | 1048576 | 字节，限制AMQ最大发送消息Size |
| activemq.receiveTimeout | int | 否 | 1000 | 毫秒，放弃等待一条消息前应该等待的时间 |
| activemq.bootstrap.servers | String | 是 | - | amq服务器地址 |
| activemq.server.username | String | 是 | - | amq用户名 |
| activemq.server.password | String | 是 | - | amq密码 |
| activemq.maxConnections | int | 否 | 20 | 最大连接数 |
| activemq.maximumActiveSessionPerConnection | int | 否 | 50 | 每个连接最大session数 |
| activemq.idleTimeout | int | 否 | 120000 | 毫秒，连接最大空闲时间 |
| activemq.optimizeAcknowledge | boolean | 否 | false | 是否开启优化提交，只在优先级中低时有效 |
| activemq.optimizeAcknowledgeTimeOut | int | 否 | 10000 | 毫秒，当前消息与上条消息消费间隔大于XX自动发送确认 |
| activemq.optimizedAckScheduledAckInterval | int | 否 | 0 | 毫秒，每隔XX自动发送确认。慎用，会出现大量日志 |
|  |  |  |  |  |

# 性能数据

**KAFKA**：

服务器X2(虚拟机) CPU:3核E5-2660 RAM:8G

客户端X1 CPU:2核i5-4300M RAM:8G

单线程发送，双线程接收，消息体100byte以内

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式 | 消息量(条) | 生产(ms) | 消费(ms)ackSize=5 | 消费(ms)ackSize=10 |
| 1(事务) | 10000 | 60267 | 9441 | 7352 |
| 1 | 10000 | 40641 | 9316 | 7669 |
| 2 | 10000 | 19573 | - | - |
| 3 | 10000 | 413 | 8247 | 5908 |
| 1(事务) | 50000 | 128211 | 43282 | 31027 |
| 1 | 50000 | 203164 | 41364 | 30007 |
| 2 | 50000 | 97126 | - | - |
| 3 | 50000 | 787 | 40507 | 30663 |
| 1(事务) | 100000 | - | - | - |
| 1 | 100000 | - | - | - |
| 2 | 100000 | - | - | - |
| 3 | 100000 | 1071 | 89752 | 60083 |