**GMQ消息队列开发指南**

针对版本V1.0.0

**©基础平台架构组**

**2016/07/01**

GMQ修订记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本号 | 修订内容 | 作者 | 审核 | 修订日期 |
| V1.0.0 | 初始版本 | 谭特贤 | 基础平台架构组 | 2016/07/01 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目 录

[1 概述 6](#_Toc458781415)

[2 接入申请流程 6](#_Toc458781416)

[3 专业术语 7](#_Toc458781417)

[4 GMQ网络部署图 9](#_Toc458781418)

[5 GMQ关键特性及应用场景 9](#_Toc458781419)

[5.1 普通send带确认消息发送 9](#_Toc458781420)

[5.2 高性能单向SendOneWay消息发送 10](#_Toc458781421)

[5.3 生产者集群、消费者集群模式 10](#_Toc458781422)

[5.4 广播、集群消费模式 10](#_Toc458781423)

[5.5 延时消费模式 11](#_Toc458781424)

[5.6 顺序消费模式 11](#_Toc458781425)

[5.7 指定Tag过滤消费模式 11](#_Toc458781426)

[5.8 支持1W+持久化队列及亿级消息堆积 12](#_Toc458781427)

[6 GMQ使用注意事项 12](#_Toc458781428)

[6.1 消息堆积问题解决办法 12](#_Toc458781429)

[6.2 消息size最大支持128K 12](#_Toc458781430)

[6.3 消息发送失败如何处理 12](#_Toc458781431)

[6.4 消费过程要做到幂等 13](#_Toc458781432)

[6.5 消费速度慢处理方式 13](#_Toc458781433)

[6.6 消费打印日志 14](#_Toc458781434)

[6.7 客户端支持 14](#_Toc458781435)

[6.8 多套业务环境共用消息队列 14](#_Toc458781436)

[6.9 复杂场景生产者组与消费者组 15](#_Toc458781437)

[7 Maven集成 16](#_Toc458781438)

[7.1 配置公司Maven私服 16](#_Toc458781439)

[7.2 Maven 工程pom.xml 16](#_Toc458781440)

[7.3 Maven工程pom.xml(集成Spring) 16](#_Toc458781441)

[8 GMQ非Spring示例代码 17](#_Toc458781442)

[8.1 更新NAMESRV\_ADDR配置项 17](#_Toc458781443)

[8.2 普通生产者、消费者模式 17](#_Toc458781444)

[8.2.1 普通Send生产者 17](#_Toc458781445)

[8.2.2 SendOneWay生产者 18](#_Toc458781446)

[8.2.3 普通消费者(默认集群模式) 19](#_Toc458781447)

[8.2.4 广播模式消费者 21](#_Toc458781448)

[8.3 顺序生产者、消费者模式 22](#_Toc458781449)

[8.3.1 顺序模式生产者 22](#_Toc458781450)

[8.3.2 顺序模式消费者 23](#_Toc458781451)

[8.4 延时生产者、消费者模式 24](#_Toc458781452)

[8.4.1 延时模式生产者 25](#_Toc458781453)

[8.4.2 延时模式消费者 26](#_Toc458781454)

[8.5 设置消息Tag 27](#_Toc458781455)

[8.5.1 生产者设置消息Tag 28](#_Toc458781456)

[8.5.2 消费者设置消息Tag 28](#_Toc458781457)

[8.6 记录消息MsgId、设置消息key 29](#_Toc458781458)

[9 GMQ 集成Spring示例代码 30](#_Toc458781459)

[9.1 普通生产者、消费者模式(Spring方式) 30](#_Toc458781460)

[9.1.1 生产者(Spring方式) 30](#_Toc458781461)

[9.1.2 生产者配置文件(producer.xml) 32](#_Toc458781462)

[9.1.3 消费者(Spring方式) 32](#_Toc458781463)

[9.1.4 消费者配置文件(consumer.xml) 33](#_Toc458781464)

[9.2 顺序生产者、消费者模式(Spring方式) 34](#_Toc458781465)

[9.2.1 生产者(Spring方式) 34](#_Toc458781466)

[9.2.2 生产者配置文件(producer.xml) 36](#_Toc458781467)

[9.2.3 消费者(Spring方式) 36](#_Toc458781468)

[9.3 延时生产者、消费者模式(Spring方式） 36](#_Toc458781469)

[9.3.1 生产者(Spring方式) 36](#_Toc458781470)

[9.3.2 生产者配置文件(producer.xml) 38](#_Toc458781471)

[9.3.3 消费者(Spring方式) 38](#_Toc458781472)

[9.4 设置消息Tag(Spring方式) 38](#_Toc458781473)

[9.4.1 生产者设置Tag(Spring方式) 38](#_Toc458781474)

[9.4.2 消费者设置Tag(Spring方式) 38](#_Toc458781475)

[附件一 申请业务场景参数清单 39](#_Toc458781476)

[附件二 GMQ开发者联系方式 41](#_Toc458781477)

# 概述

本文档旨在描述GMQ业务接入申请流程，及在使用中支持的应用场景、使用规范、注意事项等。其中集成方式包括maven集成，普通调用方式及Spring集成调用方式。

# 接入申请流程

◼**阅读最新版《GMQ消息队列使用指南》**

首先申请接入消息队列的用户，请仔细阅读本文档。了解GMQ能够支持的业务场景，及使用注意事项。确认是否满足实际使用的业务场景。

◼**明确申请业务场景参数**

用户需要给出具体的使用场景描述、并发量、TPS、峰值流量、每小时及每天预估流量等及需要申请的消息类型（[见附件一：申请业务场景参数清单](#_附件一:_申请业务场景参数清单)）。若无法提供准确值，则需要提供对应的预估值。

◼**明确申请业务的Topic名称**

所有Topic都需要先申请，再接入使用。用户需要根据自身业务申请对应Topic名称。为了性能及后续维护方便，建议一个业务或者一个组使用一个Topic（同一个Topic可以根据该Topic下不同的Tag来分类过滤）、不同业务原则上不建议共用一个Topic。

注意：申请Topic需要注明类型（普通send、sendOneWay、有序、延时等），不同类型Topic不能共用。Topic名称需要全公司唯一。为了方便后续维护，建议命名为与业务相关，例如o2m-SkuByDelay（表示o2m延时Topic用于处理Sku相关，注意Topic必须以组名标识开头，例如o2m-\*） 。

◼**统一申请方式：邮件**

将上述要求的相关业务场景参数、Topic名称等配置信息（格式[参考附件一：申请业务场景参数清单](#_附件一:_申请业务场景参数清单)），以邮件形式发送基础平台架构组。具体联系邮箱（详见[附件二：GMQ开发者联系方式](#_附件二：GMQ开发者联系方式)）。

◼**邮件反馈请求结果**

对于接入消息队列申请审核结果，会以邮件形式反馈申请人，若审核通过，则邮件回复对应申请的Topic名称、及回复当前集群NAMESRV\_ADDR地址（在集群环境变化，相应NAMESRV\_ADDR地址也会发生改变，后续也会以邮件通知各组及时更新以保证业务的可靠运行）。

注意：为了保证整个公司消息队列的稳定性，对于那些不适合接入消息队列的业务场景，或者对消息队列造成很大性能压力的接入请求，可能会被拒绝接入。因此务必在邮件中写明业务使用场景及业务场景参数。其次，在后续过程中，若有业务系统消息队列异常（例如只发送，不消费，大量堆积消息等），为了保证生产环境消息队列整体可用性，可能会对异常业务队列进行降级处理。

# 专业术语

◼**Producer**

消息生产者，负责产生消息，一般由业务系统负责产生消息。

◼**Consumer**

消息消费者，负责消费消息，一般是后台系统负责异步消费。

◼**Producer Group**

一类 Producer 的集合名称，该类 Producer 通常发送一类消息，且发送逻辑一致。

◼**Consumer Group**

一类 Consumer 的集合名称，该类Consumer 通常消费一类消息，且消费逻辑一致。

◼**Broker**

消息中转角色，负责存储消息，转发消息，一般也称为 Server。在 JMS 规范中称为 Provider。

◼**Topic**

消息主题，生产者和消费者可以约定一个主题作为唯一标识，来消息生产与接收。

**注：**在线上环境，Topic不允许自己创建，需要提交申请创建对应的Topic。

◼**Tag**

同一个消息主题下面，生产者和消费者可以约定Tag作为该主题下唯一标识，用来消息过滤消费。

◼**广播消费**

一条消息被多个 Consumer 消费，即使返些 Consumer 属于同一个 Consumer Group，消息也会Consumer Group 中的每个 Consumer 都消费一次，广播消费中的 Consumer Group 概念可以认为在消息划分方面无意。

◼**集群消费**

一个 Consumer Group 中的 Consumer 实例平均分摊消费消息。例如某个 Topic 有 9 条消息，其中一个Consumer Group 有3个实例（可能是3个进程，或者3台机器），那么每个实例只消费其中的 3 条消息。

**注**：在 CORBA Notification 规范中，无此消费方式。

在 JMS 规范中，JMS point-to-point model 与之类似，但是 GMQ的集群消费功能比P2P模型功能更丰富。GMQ下单个 Consumer Group 内的消费者类类似P2P，但是一个 Topic/Queue 可以被多个 Consumer Group 分摊消费。

◼**顺序消息**

消费消息的顺序要同发送消息的顺序一致，在GMQ中，主要指的是局部顺序，即一类消息为满足顺序性，必须 Producer 单线程顺序发送，且发送到同一个队列，这样 Consumer 就可以按照 Producer 发送的顺序去消费消息。

# GMQ网络部署图



# GMQ关键特性及应用场景

## 普通send带确认消息发送

**send**消息方法，只要不抛异常就代表发送成功。适用于大部分业务场景。其中消息发送完毕有四种返回值：

◼**SEND\_OK**

消息发送成功。

◼**FLUSH\_SLAVE\_TIMEOUT**

消息发送成功，但是服务器同步到 Slave 时超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失。

◼**SLAVE\_NOT\_AVAILABLE**

消息发送成功，但是此时 slave 不可用，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失。

◼**FLUSH\_DISK\_TIMEOUT**

消息发送成功，但是服务器刷盘超时，消息已经进入服务器队列，只有此时服务器宕机，消息才会丢失。

注：对于严格顺序消息的应用，由于顺序消息的局限性，可能会涉及到主备切换问题，所以如果sendResult 中的status字段不等于SEND\_OK，就应该尝试重试。对于其他应用，则没有必要这样。对于消息不可丢失应用，务必要有消息重发机制，例如，消息如果发送失败，存储到数据库，能有定时程序尝试重发，或者人工触发重新发送。

## 高性能单向SendOneWay消息发送

适用于性能要求高，耗时非常短，但是对可靠性要求不是很高的业务场景。例如日志收集类应用，此类应用可以采用oneWay形式调用，oneWay形式只发送请求不等待应答，而发送请求在客户端实现层面只是一个 OS系统调用的开销，即将数据写入客户端的socket缓冲区，此过程耗时通常在微秒级。

## 生产者集群、消费者集群模式

生产者可作为集群来生产消息，消费者可作为集群消费消息，集群可以是多个线程也可以是多台机器。同一个生产者集群通常发送同一类消息，并且发送逻辑一致。同一个消费者集群通常消费同一类消息，并且消费逻辑一致。若多个生产者将ProducerGroupID设置为相同，则多个生产者就组成一个生产者集群。若多个消费者将ConsumerGroupID设置为相同则为同一个消费者集群。一个Group下可以包含多个实例，可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个对象。

## 广播、集群消费模式

在广播消费模式下，所有订阅了对应Topic的消费者都能接受到消息。在集群消费模式下，不同Consumer Group ID的各个消费者集群都能接收到对应Topic下的所有消息，但是同个消费者集群（具有相同Consumer Group ID）下面的所有消费者分摊消费所有消息。

**注：**介于上述性质，所有生产者及消费者集群如果处理的不是同一类逻辑消息，则应该设置为不一样。

## 延时消费模式

GMQ支持延时消费模式，但为了保证消费队列的高性能，因此目前只支持指定等级的延时消费。目前支持的延时等级为："1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h"。

**注：**由于延时消费会一定程度上影响性能，因此非特定场景下不建议使用延时消费模式。

## 顺序消费模式

正常情况下可以保证完全的顺序消息，但是一旦发生通信异常，Broker 重启，由于队列总数发生发化，哈希取模后定位的队列会发化，产生短暂的消息顺序不一致。如果业务能容忍在集群异常情况（如某个Broker 宕机或者重启）下，消息短暂的乱序，使用普通顺序方式比较合适。若要使用顺序消费模式，则必须Producer 单线程顺序发送，且Consumer单线程接收。发送顺序消息无法利用集群 FailOver 特性，消费顺序消息的并行度依赖于队列数量，队列热点问题，个别队列因哈希不均导致消息过多，消费速度跟不上，产生消息堆积问题，遇到消息失败的消息，无法跳过，导致当前队列消费暂停。

**注意：**顺序消费对可靠性、稳定性或者性能都将有所下降。建议非特殊情况请不要采用顺序消费模式。

## 指定Tag过滤消费模式

Tag使得同一个Topic下的消息能够进行过滤处理(同一类逻辑消息正常应该发送到同一个Topic下)，可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在GMQ服务器过滤。

## 支持1W+持久化队列及亿级消息堆积

在GMQ的原理设计中，充分考虑了队列数量及消息堆积的情况。因此对大量队列持久化及消息堆积有很好的支持能力。

**注：**消息堆积过多，对消息队列性能有一定影响，因此禁止将无用的消息往GMQ中发送而不做消费。

# GMQ使用注意事项

## 消息堆积问题解决办法

目前Broker只保存 3 天的消息，3 天还未被消费的数据将会自动从队尾删除。发生消息堆积时，如果消费速度一直追不上发送速度，可以选择丢弃不重要的消息。其中消息堆积超过磁盘90%则消息会自动从消息队列中删除。

## 消息size最大支持128K

虽然消息size最大支持128k，但是随着消息size的增大消息队列TPS严重下降。因此建议消息size越小越好。

## 消息发送失败如何处理

Producer 的 send 方法本身支持内部重试，重试逻辑如下：

◼重试次数

至多重试 3 次发送。

◼轮转Broker

如果发送失败，则轮转到下一个Broker。

◼总耗时不超过阈值

这个方法总耗时时间不超过sendMsgTimeout设置的值，默认10s。所以，如果本身向broker发送消息产生超时异常，就不会再做重试。以上策略仍然不能保证消息一定发送成功，为保证消息一定成功，建议应用这样做：如果调用 send 同步方法发送失败，则尝试将消息存储到 db，由后台线程定时重试，保证消息一定到达 Broker。

## 消费过程要做到幂等

GMQ 无法避免消息重复，如果业务对消费重复非常敏感，务必要在业务层面去重，有以下几种去重方式：

◼消息唯一键去重

将消息的唯一键，可以是msgId，也可以是消息内容中的唯一标识字段，例如订单 Id 等，消费之前判断是否在Db 或 Tair(全局 KV 存储)中存在，如果不存在则插入，并消费，否则跳过。（实际过程要考虑原子性问题，判断是否存在可以尝试插入，如果报主键冲突，则插入失败，直接跳过）msgId 一定是全局唯一标识符，但是可能会存在同样的消息有两个不同 msgId 的情况（有多种原因），返种情况可能会使业务上重复消费，建议最好使用消息内容中的唯一标识字段去重。

◼业务层面状态机去重

A状态无论接收多少次相同更新请求，都只能改变到B状态。

◼Restful风格幂等API去重

Restful风格幂等指在若干次相同请求对统一资源操作的情况下，资源的状态都是一致的。

## 消费速度慢处理方式

可以通过启动多个消费者线程或者多台机器，通过提高消费并行度加快消费速度。

## 消费打印日志

如果消息量较少，建议在消费入口方法打印消息，方便后续排查问题。如果能打印每条消息消费耗时，那举在排查消费慢等线上问题时，会更方便。

## 客户端支持

当前版本仅支持java客户端调用。

## 多套业务环境共用消息队列

使用GMQ的业务系统存在200、300、500等多套运行环境，但GMQ只部署一套测试环境，在此种客观条件下为保证消息在多套环境统一管理，建议在topic和groupId方面加以区分。

◼**申请不同环境的Topic**

每套环境申请topic加上运行环境的数字后缀做区分，如微店组申请200环境的topic，就是wd-xxxx-200，这样业务系统就可对不同环境的topic发送和消费消息而不会有额外代码调整。

◼**设置唯一的ConsumerGroupID**

发送消息设置ProducerGroupID是为了保持发送消息逻辑一致；消费消息设置ConsumerGroupID是为了保持消费消息逻辑一致。

集群消费模式下，同一个ConsumerGroupID组的不同Consumer对象均可消费消息。若多套运行环境的ConsumerGroupID相同，就存在消息被交叉消费的异常情况。建议业务系统在不同的运行环境下设置唯一的ConsumerGroupID。如200环境普通消息设置的ConsumerGroupID为SimpleConsumerGroupId-test-200，而对于300环境则设置为SimpleConsumerGroupId-test-300。

## 复杂场景生产者组与消费者组



◼生产者组和消费者组

具有相同ProducerGroupID的Producer实例组成一个生产者组；具有相同ConsumerGroupID的Consumer实例组成一个消费者组。

◼不同的消费者组，均可收到同一条消息

例如 ConsumerGroupID-A组、 ConsumerGroupID-B组均可以收到消息A。

◼对于确定的一条消息，每个消费者组内部只能有一个Consumer实例能收到此消息

例如“ConsumerGroup-A”消费者组的Consumer实例X与实例Y，只能有一个实例能收到消息A；同理“ConsumerGroup-B”消费者组的Consumer实例Z与实例W，只能有一个实例能收到消息A。

◼构成Group组的实例

一个Group组可以包含多个实例，多个实例可以是多台机器，也可以是一台机器的多个进程，或者一个进程的多个对象。相同Group组的生产者实例，发送消息逻辑一致。相同Group组的消费者实例，消费消息逻辑一致。

◼若更改消费者GroupID，新的消费者组会收到历史消息，即使消息已被其他Group消费过

例如topicA已经存在200条历史消息（即使这些消息已经被其他消费者组消费过），若将本例中的“ConsumerGroup-A” 消费者组，修改为“ConsumerGroup-C”消费者组并订阅topicA，那么该消费者组“ConsumerGroup-C”也会收到topicA的200条历史消息。

# Maven集成

## 配置公司Maven私服

所有关于GMQ的包都上传至公司Maven私服，后续维护更新也将在Maven私服中操作。具体关于如何配置公司Maven私服，超出本文档范围、因此本文档不做讲解。

## Maven 工程pom.xml

<dependency>  
 <groupId>com.gome.gmq</groupId>  
 <artifactId>gmq-client</artifactId>  
 <version>1.0.0-SNAPSHOT</version>  
</dependency>

注意:

◼gmq-client会持续更新，当前版本使用SNAPSHOT做版本号

◼同一快照版本对外接口保持不变，升级过程中，若接口发生变化，版本号也会同步升级

◼每次编译请使用 clean install –U 参数，实时更新为最新jar包

## Maven工程pom.xml(集成Spring)

假若需要使用gmq-client的Spring模式，且自身项目中没有集成spring，则需要集成spring依赖包至项目。

<dependency>  
 <groupId>com.gome.gmq</groupId>  
 <artifactId>gmq-client</artifactId>  
 <version>1.0.0-SNAPSHOT</version>  
</dependency>

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-context-support</artifactId>  
 <version>3.2.2.RELEASE</version>  
</dependency>

# GMQ非Spring示例代码

## 更新NAMESRV\_ADDR配置项

以下所有代码的namesrv地址(127.0.0.1:9876)，请全部更新为申请topic邮件所反馈的namesrv地址。

## 普通生产者、消费者模式

### **普通Send生产者**

package com.gome.demo.simple;  
  
import *java.util.Properties*;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.Producer*;  
import *com.gome.api.open.base.SendResult*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.common.PropertiesConst*;  
  
/\*\*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ProducerTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的生产者组ID（ProducerGroupId）  
 properties.put(*PropertiesConst.Keys*.***ProducerGroupId***, "SimpleProducerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 *Producer* producer = *MQFactory*.createProducer(properties);  
 // 在发送消息前，必须调用start方法来启动Producer，只需调用一次即可。  
 producer.start();  
 // 循环发送消息  
 for (int i = 0; i < 10; i++) {  
 *Msg* msg = new Msg(  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 ("Hello MQ " + i).getBytes());  
  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_" + i);  
  
 // 发送消息，只要不抛异常就是成功  
 // 建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 *SendResult* sendResult = producer.send(msg);  
 *System*.***out***.println(sendResult);  
 }  
  
 *System*.***out***.println("simple producer send end.");  
 // 在应用退出前，销毁Producer对象  
 // 注意：如果不销毁也没有问题  
 producer.shutdown();  
 }  
}

### **SendOneWay生产者**

package com.gome.demo.simple;  
  
import *java.util.Properties*;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.Producer*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.common.PropertiesConst*;  
  
/\*\*  
 \* sendOneway模式：只管单边发送。由于没有返回消息状态结果逻辑，因此吞吐量及性能相对于send有较大提高。  
 \* 因此适用于可靠性要求不高，但是吞吐量性能要求很高的业务场景（例如：日志收集处理场景）  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ProducerSendOneWayTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的生产者组ID（ProducerGroupId）  
 properties.put(*PropertiesConst.Keys*.***ProducerGroupId***, "SendOneWayProducerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 *Producer* producer = *MQFactory*.createProducer(properties);  
 // 在发送消息前，必须调用start方法来启动Producer，只需调用一次即可。  
 producer.start();  
 // 循环发送消息  
 for (int i = 0; i < 10; i++) {  
 *Msg* msg = new Msg( //  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 ("Hello MQ " + i).getBytes());  
  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_" + i);  
  
 // 发送消息，只要不抛异常就是成功  
 // 建议业务程序自行记录生产及消费log日志，以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 // sendOneway模式：适用于可靠性要求不高，但是吞吐量性能要求很高的业务场景（例如：日志收集处理场景）  
 producer.sendOneway(msg);  
  
 }  
 *System*.***out***.println("send one way message end.");  
 // 在应用退出前，销毁Producer对象  
 // 注意：如果不销毁也没有问题  
 producer.shutdown();  
 }  
}

### **普通消费者(默认集群模式)**

package com.gome.demo.simple;  
  
import *java.util.Properties*;  
  
import *com.gome.api.open.base*.\*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.common.PropertiesConst*;  
  
/\*\*  
 \* 集群方式订阅消息(所有消费订阅者共同消费消息(分摊)，消息队列默认为集群消费)  
 \* 注意：集群模式消费则ConsumerGroupId必须相同。  
 \* 其次为了保证消息队列性能，消息队列自身并不保证消息不会重复消费(在某些异常情况下偶尔会出现极少数重复消息)，  
 \* 若业务系统使用在非常严格的不允许消息重复的业务场景，则需要业务系统自身处理重复消息幂等  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ConsumerTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的消费者组ID（ConsumerGroupId）  
 // 集群模式下消费，该ConsumerGroupId必须相同  
 properties.put(*PropertiesConst.Keys*.***ConsumerGroupId***, "SimpleConsumerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 // 创建普通类型消费者  
 *Consumer* consumer = *MQFactory*.createConsumer(properties);  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 consumer.subscribe("TopicTestMQ", "\*", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("Receive Msg : " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage;***  
 }  
 catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
 });  
 // 启动消费者，开始消费  
 consumer.start();  
 *System*.***out***.println("Simple Push consumer Started");  
 }  
}

### **广播模式消费者**

package com.gome.demo.simple;  
  
import java.util.Properties;  
  
import com.gome.api.open.base.\*;  
import com.gome.api.open.factory.MQFactory;  
import com.gome.common.PropertiesConst;  
  
  
/\*\*  
 \* 广播方式订阅消息(所有消费订阅者都能收到消息)  
 \* 注意：为了保证消息队列性能，消息队列自身并不保证消息不会重复消费(在某些异常情况下偶尔会出现极少数重复消息)，  
 \* 若业务系统使用在非常严格的不允许消息重复的业务场景，则需要业务系统自身处理重复消息幂等  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *BroadcastingConsumerTest* {  
 public static void main(*String*[] args) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的消费者组ID（ConsumerGroupId）  
 properties.put(PropertiesConst.Keys.ConsumerGroupId, "BroadcastingConsumerGroupId-test");  
 // 设置为广播消费模式（不设置则默认为集群消费模式）  
 properties.put(PropertiesConst.Keys.MessageModel, PropertiesConst.Values.BROADCASTING);  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(PropertiesConst.Keys.NAMESRV\_ADDR, "127.0.0.1:9876");  
  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 Consumer consumer = MQFactory.createConsumer(properties);  
 consumer.subscribe("TopicTestMQ", "\*", new MsgListener() {  
 public Action consume(Msg msg, ConsumeContext context) {  
 System.out.println("Receive: " + new String(msg.getBody()));  
 try {  
 // do something...  
 return Action.CommitMessage;  
 }  
 catch (Exception e) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return Action.ReconsumeLater;  
 }  
 }  
 });  
 // 消费者启动，开始消费消息  
 consumer.start();  
 System.out.println("Simple Broadcasting Consumer Started");  
 }  
}

## 顺序生产者、消费者模式

### **顺序模式生产者**

package com.gome.demo.order;  
  
import *java.util.Properties*;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.SendResult*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.api.open.order.OrderProducer*;  
import *com.gome.common.PropertiesConst*;  
  
  
/\*\*  
 \* 顺序消息的生产者（顺序消息的消费者与普通消费者一致）  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *OrderProducerTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的生产者组ID（ProducerGroupId）  
 properties.put(*PropertiesConst.Keys*.***ProducerGroupId***, "OrderProducerGroupId-test");  
 // 设置nameserver地址，不设置则默认为 127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 // 创建顺序类型生产者（建议尽量使用常规模式，顺序类型会降低性能及可靠性）  
 *OrderProducer* orderProducer = *MQFactory*.createOrderProducer(properties);  
 // 在发送消息前，必须调用start方法来启动Producer，只需调用一次即可。  
 orderProducer.start();  
 *System*.***out***.println("order producer started");  
 // 循环发送消息  
 for (int i = 0; i < 10; i++) {  
 *Msg* msg = new Msg( //  
 // Msg Topic  
 "TopicOrderTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 ("Hello MQ, I'm message " + i).getBytes());  
 // 设置代表消息的业务关键属性，请尽可能全局唯一（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_" + i);  
  
 // 由于是顺序消息，因此只能选择一个queue生产和消费消息  
 // shardingKey用来随机获取集群中的一个queue（可以自由设置该值，建议此处尽可能唯一，便于消息队列分散到不同的queue上）  
 *String* shardingKey = "OrderProducerTestShardingKey";  
  
 // 发送消息，只要不抛异常就是成功  
 // 建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 *SendResult* sendResult = orderProducer.send(msg, shardingKey);  
 *System*.***out***.println(sendResult);  
 }  
  
 *System*.***out***.println("order producer send message end.");  
 // 在应用退出前，销毁Producer对象  
 // 注意：如果不销毁也没有问题  
 orderProducer.shutdown();  
 }  
}

### **顺序模式消费者**

package com.gome.demo.order;  
  
import java.util.Properties;  
  
import com.gome.api.open.base.Msg;  
import com.gome.api.open.factory.MQFactory;  
import com.gome.api.open.order.ConsumeOrderContext;  
import com.gome.api.open.order.MsgOrderListener;  
import com.gome.api.open.order.OrderAction;  
import com.gome.api.open.order.OrderConsumer;  
import com.gome.common.PropertiesConst;  
  
  
/\*\*  
 \* 顺序消息的消费者者（顺序消息的消费者与普通消费者使用方法一致）  
 \* 注意：为了保证消息队列性能，消息队列自身并不保证消息不会重复消费(在某些异常情况下偶尔会出现极少数重复消息)，  
 \* 若业务系统使用在非常严格的不允许消息重复的业务场景，则需要业务系统自身处理重复消息幂等  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *OrderConsumerTest* {  
 public static void main(String[] args) {  
 Properties properties = new Properties();  
 // 您在控制台创建的消费者组ID（consumerGroupId）  
 properties.put(PropertiesConst.Keys.ConsumerGroupId, "OrderProducerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(PropertiesConst.Keys.NAMESRV\_ADDR, "127.0.0.1:9876");  
  
 // 创建顺序类型消费者（建议尽量使用常规模式，顺序类型会降低性能及可靠性）  
 OrderConsumer orderedConsumer = MQFactory.createOrderedConsumer(properties);  
  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 orderedConsumer.subscribe("TopicOrderTestMQ", "\*", new MsgOrderListener() {  
 @Override  
 public OrderAction consume(Msg msg, ConsumeOrderContext consumeOrderContext) {  
 System.out.println(new String(msg.getBody()));  
 try {  
 // do something..  
 return OrderAction.Success;  
 } catch (Exception e) {  
 // 消费失败，Suspend，消息被放置到重试队列，延时后下次重新消费  
 return OrderAction.Suspend;  
 }  
 }  
 });  
 // 启动消费者，开始消费  
 orderedConsumer.start();  
 System.out.println("order consumer started");  
 }  
}

## 延时生产者、消费者模式

### **延时模式生产者**

package com.gome.demo.delay;  
  
import *java.text.SimpleDateFormat*;  
import *java.util.Date*;  
import *java.util.Properties*;  
import *java.util.Random*;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.Producer*;  
import *com.gome.api.open.base.SendResult*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.common.DelayLevelConst*;  
import *com.gome.common.PropertiesConst*;  
  
  
/\*\*  
 \* 延时消费类型生产者(延时消费的消费者与普通消费者一致，延时消费生产者与普通生产者一致，只是在发送消息时，增加延时等级即可)  
 \*  
 \* @author tantexian  
 \* @since 2016/6/28  
 \*/  
public class *ProducerDelayTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的生产者组ID（ProducerGroupId）  
 properties.put(*PropertiesConst.Keys*.***ProducerGroupId***, "DelayProducerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 *Producer* producer = *MQFactory*.createProducer(properties);  
 // 在发送消息前，必须调用start方法来启动Producer，只需调用一次即可。  
 producer.start();  
 // 循环发送消息  
 *SimpleDateFormat* sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");  
 for (int i = 0; i < 10; i++) {  
 // 随机0-10s中延时设置  
 try {  
 *Thread*.sleep(new Random().nextInt(10) \* 1000);  
 } catch (*InterruptedException e*) {  
 *e*.printStackTrace();  
 }  
 *Msg* msg = new Msg( //  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 (" {MsgBornTime: " + sdf.format(new Date()) + "} {MsgBody: Hello MQ " + i + "}").getBytes());  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_" + i);  
  
 // 延时模式（建议尽量使用常规模式，延时模式会降低性能及可靠性）  
 // deliver time level 为延时等级（当前版本只支持固定的延时等级）,具体值参考DelayLevelConst枚举类，  
 // 指定一个延时等级，在这个等级延时时刻之后才能被消费，这个例子表示 10s 后才能被消费  
 msg.setDelayTimeLevel(*DelayLevelConst*.***TenSecond***.val());  
  
 // 发送消息，只要不抛异常就是成功  
 // 建议业务程序自行记录生产及消费log日志，以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 *SendResult* sendResult = producer.send(msg);  
 *System*.***out***.println(sendResult);  
 }  
 *System*.***out***.println("ProducerDelayTest send message end.");  
  
 // 在应用退出前，销毁Producer对象  
 // 注意：如果不销毁也没有问题  
 producer.shutdown();  
 }  
}

### **延时模式消费者**

package com.gome.demo.delay;  
  
import *java.text.SimpleDateFormat*;  
import *java.util.Date*;  
import *java.util.Properties*;  
  
import *com.gome.api.open.base*.\*;  
import *com.gome.api.open.factory.MQFactory*;  
import *com.gome.common.PropertiesConst*;  
  
/\*\*  
 \* 延时消费类型消费者(延时消费的消费者与普通消费者一致)  
 \* 注意：为了保证消息队列性能，消息队列自身并不保证消息不会重复消费(在某些异常情况下偶尔会出现极少数重复消息)，  
 \* 若业务系统使用在非常严格的不允许消息重复的业务场景，则需要业务系统自身处理重复消息幂等  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ConsumerDelayTest* {  
 public static void main(*String*[] *args*) {  
 *Properties* properties = new Properties();  
 // 您在控制台创建的消费者组ID（ConsumerGroupId）  
 properties.put(*PropertiesConst.Keys*.***ConsumerGroupId***, "DelayConsumerGroupId-test");  
 // 设置nameserver地址，不设置则默认为127.0.0.1:9876  
 properties.put(*PropertiesConst.Keys*.***NAMESRV\_ADDR***, "127.0.0.1:9876");  
  
 // 创建普通类型消费者  
 *Consumer* consumer = *MQFactory*.createConsumer(properties);  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 // 延时模式（建议尽量使用常规模式，延时模式会降低性能及可靠性）  
 final *SimpleDateFormat* sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");  
 consumer.subscribe("TopicTestMQ", "\*", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("[NowTime:" + *sdf*.format(new Date()) + "] Receive Msg : " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 } catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
 });  
 // 启动消费者，开始消费  
 consumer.start();  
 *System*.***out***.println("ConsumerDelayTest consumer Started");  
 }  
}

## 设置消息Tag

消息Tag 可理解为Gmail中的标签，可对同一个topic下的消息进行再归类，方便Consumer指定过滤条件在MQ服务端过滤。

### **生产者设置消息Tag**

◼通过构造方法设置消息tag

*Msg* msgOne = new Msg(topic, "tagA", "data1".getBytes());  
*System*.***out***.println(msgOne.getTags());

◼通过属性设置消息tag

*Msg* msgTwo = new Msg(topic, null, "data3".getBytes());  
msgTwo.setTags("tagB");  
*System*.***out***.println(msgTwo.getTags());

### **消费者设置消息Tag**

Consumer端有三种方式可以设置消息Tag，过滤单个Tag、多个Tag、全部Tag。

◼订阅topic单个Tag消息

// 订阅topic名称为“TopicTestMQ”主题下“TagA”标签的所有消息  
consumer.subscribe("TopicTestMQ", "TagA", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("Receive message: " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 } catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
});

◼订阅topic多个Tag消息

// 订阅topic名称为“TopicTestMQ”主题下，tags分别等于TagA或TagB或TagC下所有消息  
consumer.subscribe("TopicTestMQ", "TagA || TagB || TagC", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("Receive message: " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 } catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
});

◼订阅topic的全部tag消息

// 订阅topic名称为“TopicTestMQ”主题全部标签的所有消息  
consumer.subscribe("TopicTestMQ", "\*", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("Receive message: " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 } catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
});

## 记录消息MsgId、设置消息key

每条消息均有一个全局唯一的消息ID，字段名称为msgId。Producer端发送消息成功则自动创建msgId，Consumer端则根据msgId消息消息。建议使用GMQ的各业务子系统在发送消息成功之后、消费消息之前均采用日志记录msgId。

◼Producer端记录msgId、设置key

// 发送消息，只要不抛异常就是成功  
// 消费者订阅消费，建议业务程序自行记录生产及消费log日志，以方便您在无法正常收到消息情况下，

// 可通过MQ控制台或者MQ日志查询消息并补发。  
try {  
 msg.setKey("ORDERID\_100");  
 *SendResult* sendResult = producer.send(msg);  
 assert sendResult != null;  
 *System*.***out***.println("send success. msgId=" + sendResult.getMsgId());  
} catch (*GomeClientException e*) {  
 *System*.***out***.println("send error: " + *e*.getMessage());  
 *e*.printStackTrace();  
}

◼Consumer端记录msgId、获取key

// 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
// 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
consumer.subscribe("TopicTestMQ", "", new *MsgListener*() {  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("msgId=" + *msg*.getMsgId() + ",key=" + *msg*.getKey() + ",body=" + new String(*msg*.getBody()));

try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 } catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
});

◼消息msgId与消息key区别

消息msgId主要用于消息中间件维护数据，是GMQ全局唯一的标识字段。

消息key从另一个纬度标识一条消息，其存在的价值偏向于业务层面，即消息key能在业务层面唯一标识一条消息，例如订单组可采用订单orderId字段来当做消息key值。建议使用GMQ的各个业务系统尽可能保持消息key值唯一。

# GMQ 集成Spring示例代码

## 普通生产者、消费者模式(Spring方式)

### **生产者(Spring方式)**

package com.gome.demo.springwithbean;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.Producer*;  
import *com.gome.api.open.base.SendResult*;  
import *com.gome.api.open.exception.GomeClientException*;  
import *org.springframework.context.ApplicationContext*;  
import *org.springframework.context.support.ClassPathXmlApplicationContext*;  
  
/\*\*  
 \* @author tantexian  
 \* @date 2016/6/27  
 \*/  
public class *ProducerWithSpring* {  
 public static void main(*String*[] *args*) {  
 /\*\*  
 \* 生产者Bean配置在producer.xml中,可通过ApplicationContext获取或者直接注入到其他类(比如具体的Controller)中.  
 \* 如果项目本身已经集成spring、则直接使用项目已有的spring配置bean、获取bean方式，不需要使用下述ClassPathXmlApplicationContext类方法来获取bean  
 \*/  
 *ApplicationContext* context = new ClassPathXmlApplicationContext("producer.xml");  
 // 获取普通生产者Bean  
 *Producer* producer = (*Producer*) context.getBean("producer");  
 assert producer != null;  
 //循环发送消息  
 for (int i = 0; i < 10; i++) {  
 *Msg* msg = new Msg( //  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 ("(ProducerWithSpring) Hello MQ " + i).getBytes());  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ 控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_100");  
 // 发送消息，只要不抛异常就是成功  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 try {  
 *SendResult* sendResult = producer.send(msg);  
 assert sendResult != null;  
 *System*.***out***.println("send success. msgId=" + sendResult.getMsgId());  
 } catch (*GomeClientException e*) {  
 *System*.***out***.println("send error: " + *e*.getMessage());  
 *e*.printStackTrace();  
 }  
 }  
 *System*.***out***.println("ProducerWithSpring send message end.");  
 *System*.exit(0);  
 }  
}

### **生产者配置文件(producer.xml)**

此配置文件有如下注意事项

◼**更新NAMESRV\_ADDR配置项**

生产者配置文件的NAMESRV\_ADDR配置项，请更新为申请topic邮件所反馈的namesrv地址。

*<?xml version=*"1.0" *encoding=*"UTF-8"*?>*<beans *xmlns=*"http://www.springframework.org/schema/beans"  
 *xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"  
 *xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <bean *id=*"producer" *class=*"com.gome.api.open.bean.ProducerBean" *init-method=*"start" *destroy-method=*"shutdown">  
 <property *name=*"properties" > <!--生产者配置信息-->  
 <props>  
 <prop *key=*"ProducerGroupId">ProducerGroupId-test</prop>  
 <!--设置NAMESRV\_ADDR，不设置默认为127.0.0.1:9876-->  
 <prop *key=*"NAMESRV\_ADDR">127.0.0.1:9876</prop>  
 </props>  
 </property>  
 </bean>  
</beans>

### **消费者(Spring方式)**

◼****实现消息监听器（com.gome.api.open.base.MsgListener）接口

package com.gome.demo.springwithbean;  
  
import *com.gome.api.open.base.Action*;  
import *com.gome.api.open.base.ConsumeContext*;  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.MsgListener*;  
  
/\*\*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ConsumerMessageListener* implements *MsgListener* {  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 public *Action* consume(*Msg msg*, *ConsumeContext context*) {  
 *System*.***out***.println("Receive: " + new String(*msg*.getBody()));  
 try {  
 // do something..  
 return *Action*.***CommitMessage***;  
 }  
 catch (*Exception e*) {  
 // 消费失败，返回ReconsumeLater，消息被放置到重试队列，延时后下次重新消费  
 return *Action*.***ReconsumeLater***;  
 }  
 }  
}

◼Consumer端集成Spring

package com.gome.demo.springwithbean;  
  
import *org.springframework.context.ApplicationContext*;  
import *org.springframework.context.support.ClassPathXmlApplicationContext*;  
  
/\*\*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *ConsumerWithSpring* {  
 public static void main(*String*[] *args*) {  
 /\*\*  
 \* 消费者Bean配置在consumer.xml中,可通过ApplicationContext获取或者直接注入到其他类(比如具体的Controller)中.  
 \* 此处为启动消费者，具体消息消费，在consumer.xml中配置的对应Listener  
 \*/  
 *ApplicationContext* context = new ClassPathXmlApplicationContext("consumer.xml");  
 assert context != null;  
 *System*.***out***.println("ConsumerWithSpring consumer Started");  
 }  
}

### **消费者配置文件(consumer.xml)**

此配置文件有如下注意事项

◼**更新NAMESRV\_ADDR配置项**

消费者配置文件的NAMESRV\_ADDR配置项，请更新为申请topic邮件所反馈的namesrv地址。

◼**更新Listener监听器接口地址**

消费消息的监听器接口（com.gome.api.open.base.MsgListener）的实现，需由业务系统自行编写，并将实现此接口的java文件的完整路径，更新到如下配置文件的msgListener配置项。

*<?xml version=*"1.0" *encoding=*"UTF-8"*?>*<beans *xmlns=*"http://www.springframework.org/schema/beans"  
 *xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"  
 *xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
 <!-- Listener配置-->  
 <bean *id=*"msgListener" *class=*"com.gome.demo.springwithbean.ConsumerMessageListener"></bean>  
  
 <!-- Consumer配置 -->  
 <bean *id=*"consumer" *class=*"com.gome.api.open.bean.ConsumerBean" *init-method=*"start" *destroy-method=*"shutdown">  
 <property *name=*"properties">  
 <props>  
 <prop *key=*"ConsumerGroupId">ConsumerGroupId-test</prop>  
 <!--设置NAMESRV\_ADDR，不设置默认为127.0.0.1:9876-->  
 <prop *key=*"NAMESRV\_ADDR">127.0.0.1:9876</prop>  
 </props>  
 </property>  
  
 <property *name=*"subscriptionTable">  
 <map>  
 <entry *value-ref=*"msgListener">  
 <key>  
 <bean *class=*"com.gome.api.open.bean.Subscription">  
 <property *name=*"topic" *value=*"TopicTestMQ"/>  
 <property *name=*"expression" *value=*"\*"/>  
 </bean>  
 </key>  
 </entry>  
 <!--更多的订阅添加entry节点即可-->  
 </map>  
 </property>  
 </bean>  
</beans>

## 顺序生产者、消费者模式(Spring方式)

### **生产者(Spring方式)**

package com.gome.demo.springwithbean;  
  
import *com.gome.api.open.order.OrderProducer*;  
import *org.springframework.context.ApplicationContext*;  
import *org.springframework.context.support.ClassPathXmlApplicationContext*;  
  
import *com.gome.api.open.base.Msg*;  
import *com.gome.api.open.base.SendResult*;  
import *com.gome.api.open.exception.GomeClientException*;  
  
  
/\*\*  
 \* 顺序消息的生产者（顺序消息的消费者与普通消费者使用方法一致）  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class *OrderProducerWithSpring* {  
  
 public static void main(*String*[] *args*) {  
 /\*\*  
 \* 生产者Bean配置在producer.xml中,  
 \* 可通过ApplicationContext获取或者直接注入到其他类(比如具体的Controller)中.  
 \*/  
 *ApplicationContext* context = new ClassPathXmlApplicationContext("orderProducer.xml");  
 // 获取顺序消费者Bean  
 *OrderProducer* orderProducer = (*OrderProducer*) context.getBean("orderProducer");  
 assert orderProducer != null;  
 // 循环发送消息  
 for (int i = 0; i < 10; i++) {  
 *Msg* msg = new Msg( //  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 ("(OrderProducerWithSpring) Hello MQ " + i).getBytes());  
  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ 控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_97");  
  
 // 发送消息，只要不抛异常就是成功,  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下,可通过MQ控制台或者MQ日志查询消息并补发。  
 try {  
 */\*  
 \* 顺序发送的消息的shardingKey值必须相同。由于是顺序消息，因此只能选择一个queue生产和消费消息  
 \* shardingKey用来随机获取集群中的一个queue  
 \* （可以自由设置shardingKey值，建议此处尽可能唯一，便于消息队列分散到不同的queue上）  
 \*/  
 String* shardingKey = "orderProducerShardingKey125";  
 *SendResult* sendResult = orderProducer.send(msg, shardingKey);  
 assert sendResult != null;  
 *System*.***out***.println("send success. msgId=" + sendResult.getMsgId());  
 } catch (*GomeClientException e*) {  
 *System*.***out***.println("send error: " + *e*.getMessage());  
 *e*.printStackTrace();  
 }  
 }  
 *System*.***out***.println("OrderWithSpring send message end.");  
 *System*.exit(0);  
 }  
}

### **生产者配置文件(producer.xml)**

顺序生产者spring配置文件与普通生产者spring配置文件一致，具体请参考8.1.2章节

### **消费者(Spring方式)**

顺序消费者spring方式与普通消费者一致，具体请参考8.1.3、8.1.4章节。

## 延时生产者、消费者模式(Spring方式）

### **生产者(Spring方式)**

package com.gome.demo.springwithbean;  
  
import com.gome.api.open.base.Msg;  
import com.gome.common.DelayLevelConst;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.support.ClassPathXmlApplicationContext;  
  
import com.gome.api.open.base.Producer;  
import com.gome.api.open.base.SendResult;  
import com.gome.api.open.exception.GomeClientException;  
  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
  
/\*\*  
 \* 延时消费类型生产者(延时消费的消费者与普通消费者一致，延时消费生产者与普通生产者一致，只是在发送消息时，增加延时等级即可)  
 \*  
 \* @author tantexian  
 \* @since 2016/6/27  
 \*/  
public class DelayProducerWithSpring {  
 public static void main(String[] args) {  
 /\*\*  
 \* 生产者Bean配置在producer.xml中,  
 \* 可通过ApplicationContext获取或者直接注入到其他类(比如具体的Controller)中  
 \*/  
 ApplicationContext context = new ClassPathXmlApplicationContext("producer.xml");  
 // 获取普通生产者Bean  
 Producer producer = (Producer) context.getBean("producer");  
 assert producer != null;  
 SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");  
 // 循环发送消息  
 for (int i = 0; i < 10; i++) {  
 Msg msg = new Msg(  
 // Msg Topic  
 "TopicTestMQ",  
 // Msg Tag 可理解为Gmail中的标签，对消息进行再归类，方便Consumer指定过滤条件在MQ服务器过滤  
 "TagA",  
 // Msg Body 可以是任何二进制形式的数据， MQ不做任何干预，  
 // 需要Producer与Consumer协商好一致的序列化和反序列化方式  
 (" {MsgBornTime: " + sdf.format(new Date()) + "} {MsgBody: Hello MQ " + i + "}").getBytes());  
 // 设置代表消息的业务关键属性，请尽可能全局唯一。（例如订单ID）。  
 // 以方便您在无法正常收到消息情况下，可通过MQ 控制台查询消息并补发。  
 // 注意：不设置也不会影响消息正常收发  
 msg.setKey("ORDERID\_180");  
  
 // 延时模式（建议尽量使用常规模式，延时模式会降低性能及可靠性）  
 // deliver time level 为延时等级（当前版本只支持固定的延时等级）,具体值参考DelayLevelConst枚举类，  
 // 指定一个延时等级，在这个等级延时时刻之后才能被消费，这个例子表示 10s 后才能被消费  
 msg.setDelayTimeLevel(DelayLevelConst.TenSecond.val());  
  
 // 发送消息，只要不抛异常就是成功  
 // 消费者订阅消费，建议业务程序自行记录生产及消费log日志，  
 // 以方便您在无法正常收到消息情况下，可通过MQ控制台或者MQ日志查询消息并补发。  
 try {  
 SendResult sendResult = producer.send(msg);  
 assert sendResult != null;  
 System.out.println("send success. msgId=" + sendResult.getMsgId());  
 } catch (GomeClientException e) {  
 System.out.println("send error: " + e.getMessage());  
 e.printStackTrace();  
 }  
 }  
 System.out.println("DelayWithSpring send message end.");  
 System.exit(0);  
 }  
}

### **生产者配置文件(producer.xml)**

延时生产者spring配置文件与普通生产者spring配置文件一致，具体请参考8.1.2。

### **消费者(Spring方式)**

顺序消费者spring方式与普通消费者一致，具体请参考8.1.3、8.1.4章节。

## 设置消息Tag(Spring方式)

### **生产者设置Tag(Spring方式)**

Proudcer端集成Spring设置消息Tag与普通Producer设置Tag一致，具体请参考8.4.1章节。

### **消费者设置Tag(Spring方式)**

集成Spring组件的Consumer端过滤消息Tag，可在相应Consumer端的配置文件做过滤。示例如下

**◼订阅topic单个Tag消息**

<property *name=*"subscriptionTable">  
 <map>  
 <entry *value-ref=*"msgListener">  
 <key>  
 <bean *class=*"com.gome.api.open.bean.Subscription">  
 <property *name=*"topic" *value=*"TopicTestMQ"/>  
 <!-- 订阅topic名称为“TopicTestMQ”主题下的“TagA”标签下所有消息-->  
 <property *name=*"expression" *value=*"TagA"/>  
 </bean>  
 </key>  
 </entry>  
 </map>  
</property>

**◼订阅topic多个Tag消息**

<property *name=*"subscriptionTable">  
 <map>  
 <entry *value-ref=*"msgListener">  
 <key>  
 <bean *class=*"com.gome.api.open.bean.Subscription">  
 <property *name=*"topic" *value=*"TopicTestMQ"/>  
 <!-- 订阅topic名称为“TopicTestMQ”主题下，tags分别等于TagA或TagB或TagC下的消息-->  
 <property *name=*"expression" *value=*"TagA || TagB || TagC"/>  
 </bean>  
 </key>  
 </entry>  
 </map>  
</property>

**◼订阅topic的全部tag消息**

<property *name=*"subscriptionTable">  
 <map>  
 <entry *value-ref=*"msgListener">  
 <key>  
 <bean *class=*"com.gome.api.open.bean.Subscription">  
 <property *name=*"topic" *value=*"TopicTestMQ"/>  
 <!-- 订阅topic名称为“TopicTestMQ”主题下所有tag消息-->  
 <property *name=*"expression" *value=*"\*"/>  
 </bean>  
 </key>  
 </entry>  
 </map>  
</property>

# 附件一 申请业务场景参数清单

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Topic名称** | **消息类型**  **(普通、顺序、延时、sendOneWay)** | **使用场景描述** | **TPS** | **并发量** | **峰值流量** | **流量**  **每小时/每天** |
| o2m-SkuByDelay | 延时 | 用于o2m处理Sku相关消息，且每个消息需要延时10秒 | 1k/s | 1w | 5k | 10w/h  200w/d |
| o2m-LogBySendOneWay | sendOneWay | 用于处理日志相关、日志发送量非常大，但是对日志发送的可靠性要求不高 | 1w/s | 1k | 1w | 20w/h  500w/d |
| wd-DeliverByOrder | 顺序 | 用于处理微店的发货消息、且消息需要顺序发送 | 2k/s | 1k | 5k | 1k/h  1w/d |
| wd-OrderBySend | 普通send | 用于处理微店订单消息 | 100/s | 1k | 1w | 1k/h  1w/d |
| … |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

◼ **Topic命名规范**

为了方便后续运行维护，规范所有topic命名以唯一的组标识开头（例o2m-\*）

◼ **Topic环境区别**

目前GMQ只部署了一套测试环境，为了便于区分开发过程的各种环境（开发、测试、预发布），申请的topic名称将以各条线的数字后缀做区别（如微店组申请200环境的topic， wd-xxxx-200）

# 附件二 GMQ开发者联系方式

◼ 成都信息中心基础平台邮件组

[cdxxjcpt@gome.com.cn](mailto:cdxxjcpt@gome.com.cn)

◼ 成都基础平台组成员

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名 | 联系方式 | 部门 | 更新日期 |
| 谭特贤 | tantexian@gome.com.cn | 成都研发中心 | 2016/07/01 |
| 郜焱磊 | gaoyanlei@gome.com.cn | 成都研发中心 | 2016/07/01 |
| 田玉粮 | tianyuliang@gome.com.cn | 成都研发中心 | 2016/07/01 |
| 尹同强 | yintongjiang@gome.com.cn | 成都研发中心 | 2016/07/01 |
| 罗继 | luoji@gome.com.cn | 成都研发中心 | 2016/07/01 |
|  |  |  |  |
|  |  |  |  |