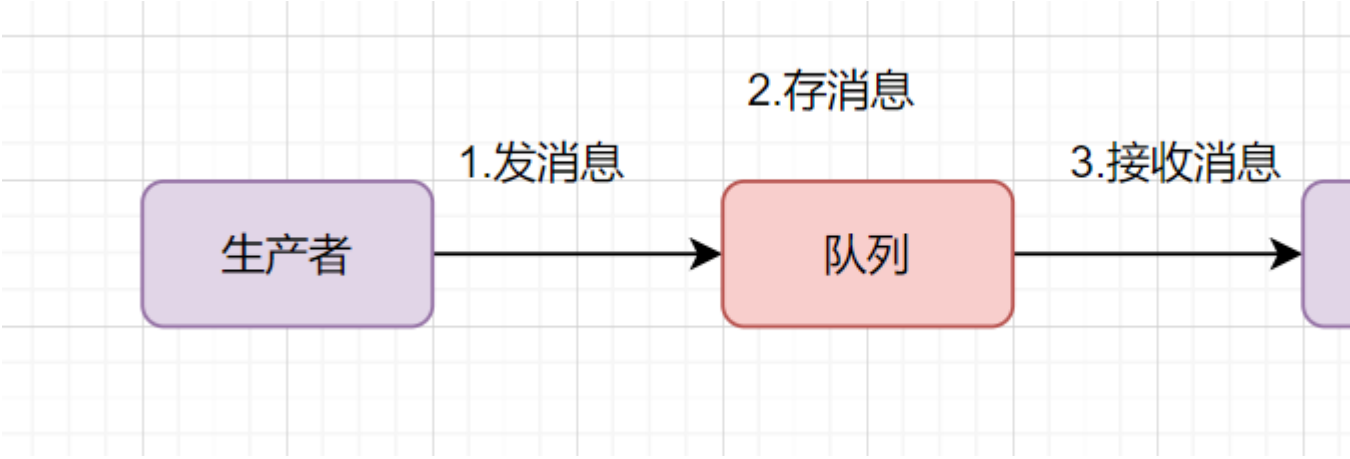
1 前沿

1 什么是消息队列
2 消息队列的应用场景
3 消息队列如何解决消息丢失问题
4 消息队列如何保证消息的顺序性。
5 消息有可能发生重复消费吗？如何幂等处理？
6 如何处理消息队列的消息积压问题
7 消息队列技术选型，Kafka 还是 RocketMQ，还是 RabbitMQ
8 消息中间件如何做到高可用？
9 如何保证数据一致性，事务消息如何实现
10 如果让你写一个消息队列，该如何进行架构设计？

11 前言消息队列常见问题

11.1 什么是消息队列

你可以把消息队列理解为一个**使用队列来通信**的组件。它的本质，就是个**转发器**，包含**发消息、存消息、消费消息**的过程。最简单的消息队列模型如下：



我们通常说的消息队列，简称 **MQ（Message Queue）**，它其实就指**消息中间件**，当前业界比较流行的开源消息中间件包括：RabbitMQ、RocketMQ、Kafka。

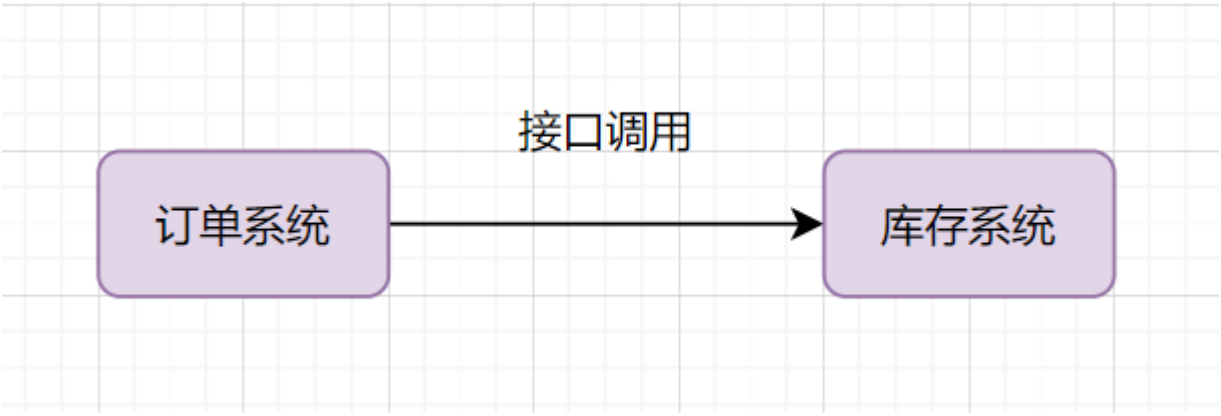
11.2 消息队列有哪些使用场景。

有时候面试官会换个角度问你，**为什么使用消息队列**。你可以回答以下这几点：

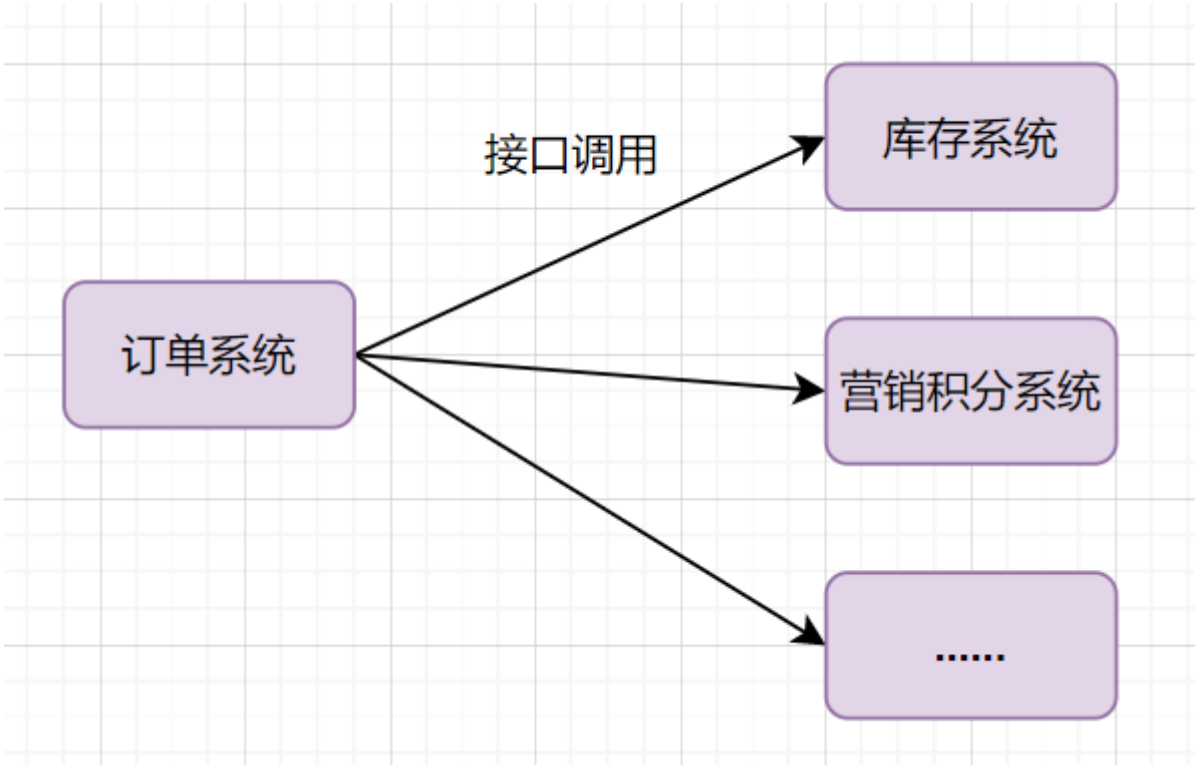
1. 应用解耦
2. 流量削峰
3. 异步处理
4. 消息通讯
5. 远程调用

11.2.1 应用解耦

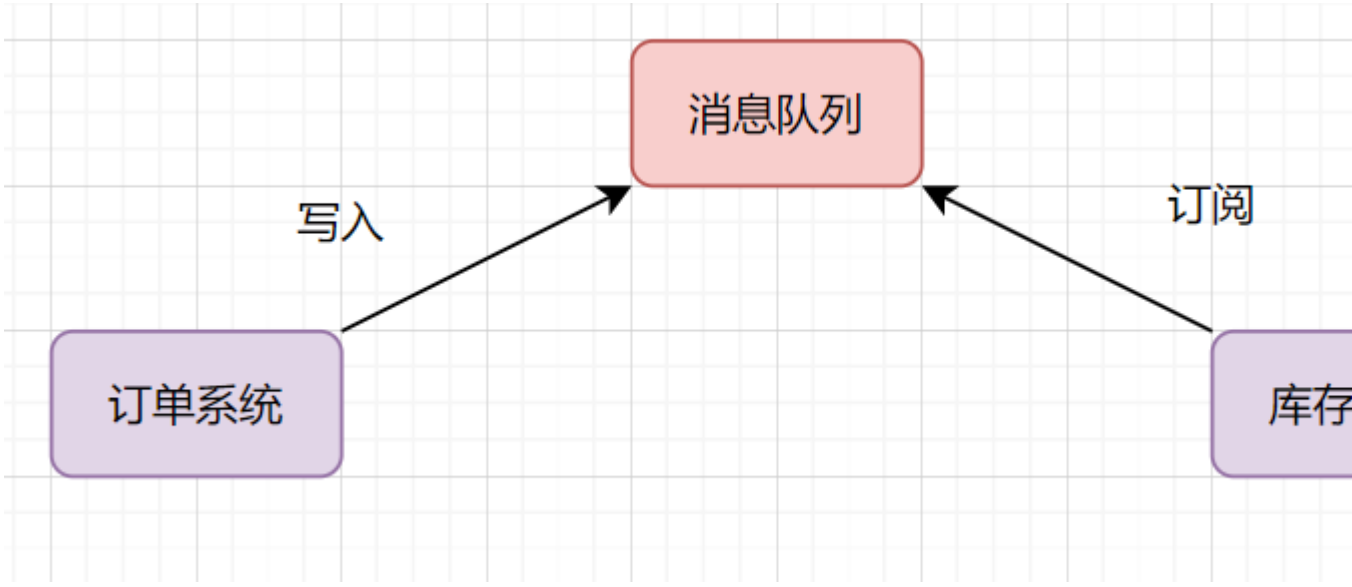
举个常见业务场景：下单扣库存，用户下单后，订单系统去通知库存系统扣减。传统的做法就是订单系统直接调用库存系统：



- 如果库存系统**无法访问**，下单就会失败，订单和库存系统存在耦合关系
- 如果业务又接入一个营销积分服务，那订单下游系统要扩充，如果未来接入越来越多的下游系统，那订单系统代码需要经常**修改**



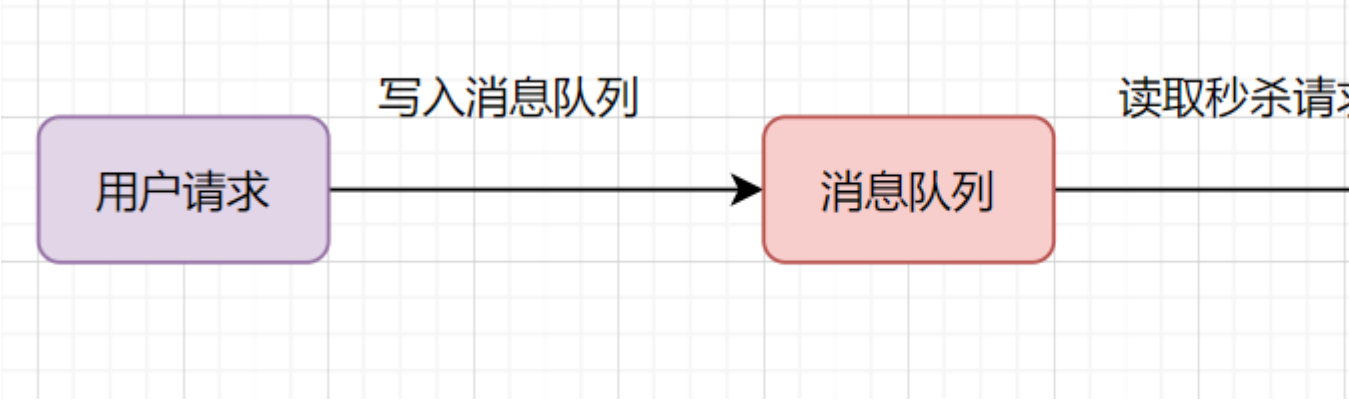
如何解决这个问题呢？可以引入消息队列



1. 订单系统：用户下单后，消息写入到消息队列，返回下单成功
2. 库存系统：订阅下单消息，获取下单信息，进行库存扣减操作。

11.2.2 流量削峰

流量削峰也是**消息队列的常用场景**。我们做秒杀实现的时候，需要避免流量暴涨，打垮应用系统的风险。可以在应用前面加入消息队列。



假设秒杀系统每秒最多可以处理 2k 个请求，每秒却有 5k 的请求过来，可以引入消息队列，秒杀系统每秒从消息队列拉 2k 请求处理得了。

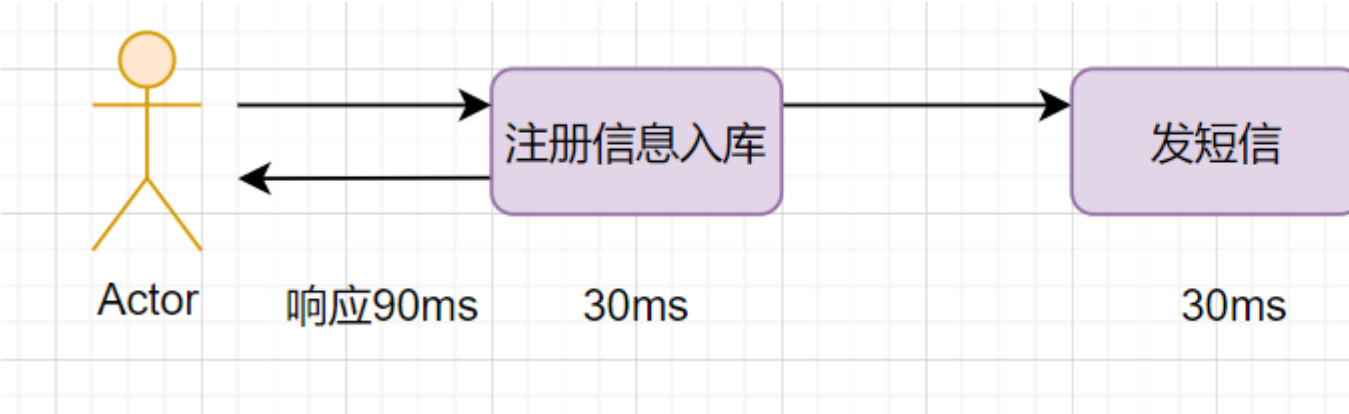
有些伙伴担心这样会出现**消息积压**的问题，

- 首先秒杀活动不会每时每刻都那么多请求过来，**高峰期过去后**，积压的请求可以慢慢处理；
- 其次，如果消息队列长度超过最大数量，可以直接抛弃用户请求或跳转到错误页面；

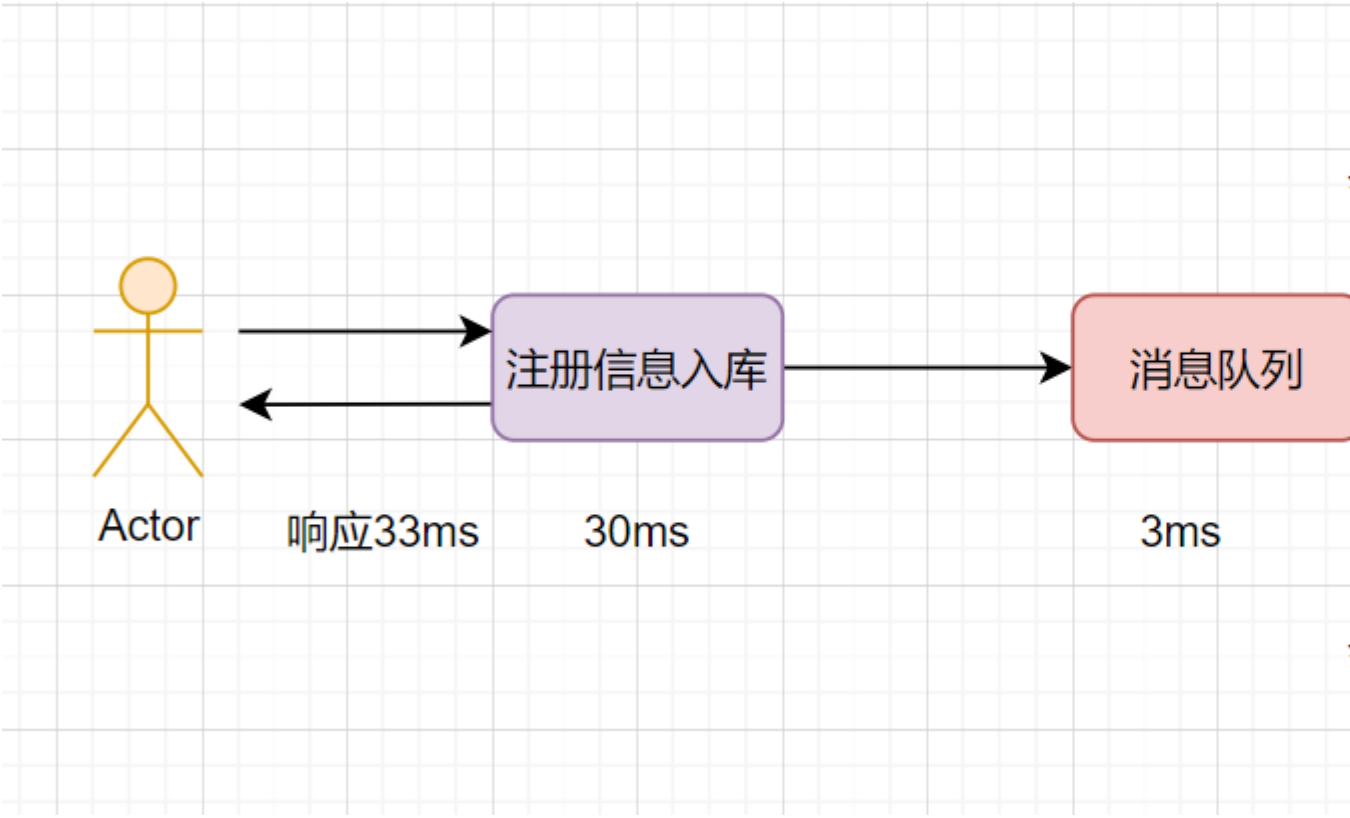
11.2.3 异步处理

我们经常会遇到这样的业务场景：用户注册成功后，给它发个短信和发个邮件。

如果注册信息入库是 30ms，发短信、邮件也是 30ms，三个动作**串行执行**的话，会比较耗时，响应 90ms：



如果采用并行执行的方式，可以减少响应时间。注册信息入库后，同时异步发短信和邮件。如何实现异步呢，用消息队列即可，就是说，注册信息入库成功后，写入到消息队列（这个一般比较快，如只需要 3ms），然后异步读取发邮件和短信。



11.2.4 消息通讯

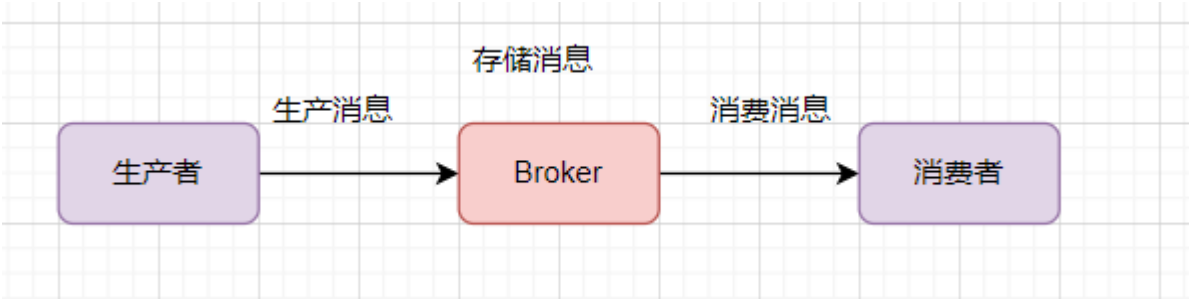
消息队列内置了高效的通信机制，可用于消息通讯。如实现点对点消息队列、聊天室等。

11.2.5 远程调用

我们公司基于 MQ，自研了远程调用框架。

11.3 消息队列如何解决消息丢失问题？

一个消息从生产者产生，到被消费者消费，主要经过这 3 个过程：



因此如何保证 MQ 不丢失消息，可以从这三个阶段阐述：

- 生产者保证不丢消息
- 存储端不丢消息
- 消费者不丢消息

11.3.1 生产者保证不丢消息

生产端如何保证不丢消息呢？**确保生产的消息能到达存储端**。

如果是 **RocketMQ** 消息中间件，Producer 生产者提供了三种发送消息的方式，分别是：

- 同步发送
- 异步发送
- 单向发送

生产者要想发消息时保证消息不丢失，可以：

- 采用**同步方式**发送，send 消息方法返回**成功**状态，就表示消息正常到达了存储端 Broker。
- 如果 send 消息**异常**或者返回**非成功**状态，可以**重试**。
- 可以使用事务消息，RocketMQ 的事务消息机制就是为了保证零丢失来设计的

11.3.2 存储端不丢消息

如何保证存储端的消息不丢失呢？确保消息**持久化**到磁盘。大家很容易想到就是**刷盘机制**。

刷盘机制分**同步刷盘和异步刷盘**：

- 生产者消息发过来时，只有持久化到磁盘，RocketMQ 的存储端 Broker 才返回一个成功的 ACK 响应，这就是**同步刷盘**。它保证消息不丢失，但是影响了性能。
- 异步刷盘的话，只要消息写入 PageCache 缓存，就返回一个成功的 ACK 响应。这样提高了 MQ 的性能，但是如果这时候机器断电了，就会丢失消息。

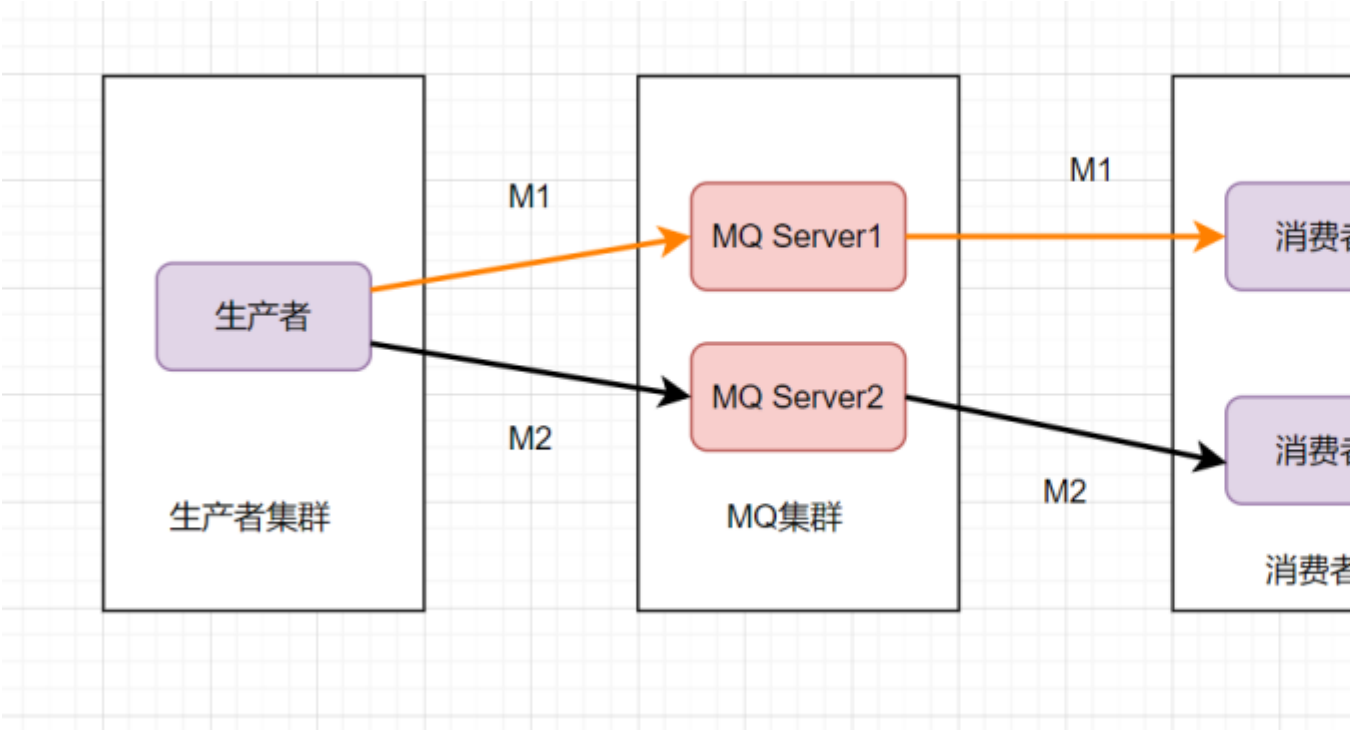
Broker 一般是**集群部署**的，有 master 主节点和 slave 从节点。消息到 Broker 存储端，只有主节点和从节点都写入成功，才反馈成功的 ack 给生产者。这就是**同步复制**，它保证了消息不丢失，但是降低了系统的吞吐量。与之对应的就是**异步复制**，只要消息写入主节点成功，就返回成功的 ack，它速度快，但是会有性能问题。

11.3.3 消费阶段不丢消息

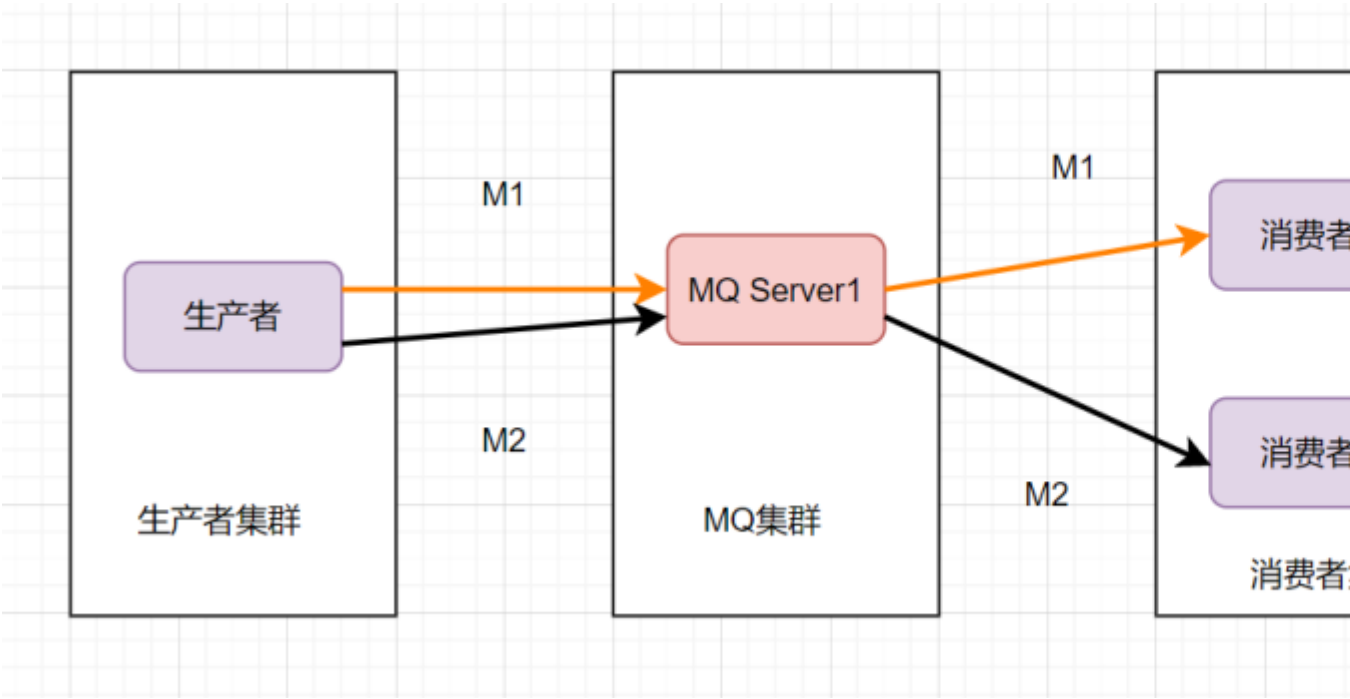
消费者执行完业务逻辑，再反馈会 Broker 说消费成功，这样才可以保证消费阶段不丢消息。

11.4 消息队列如何保证消息的顺序性。

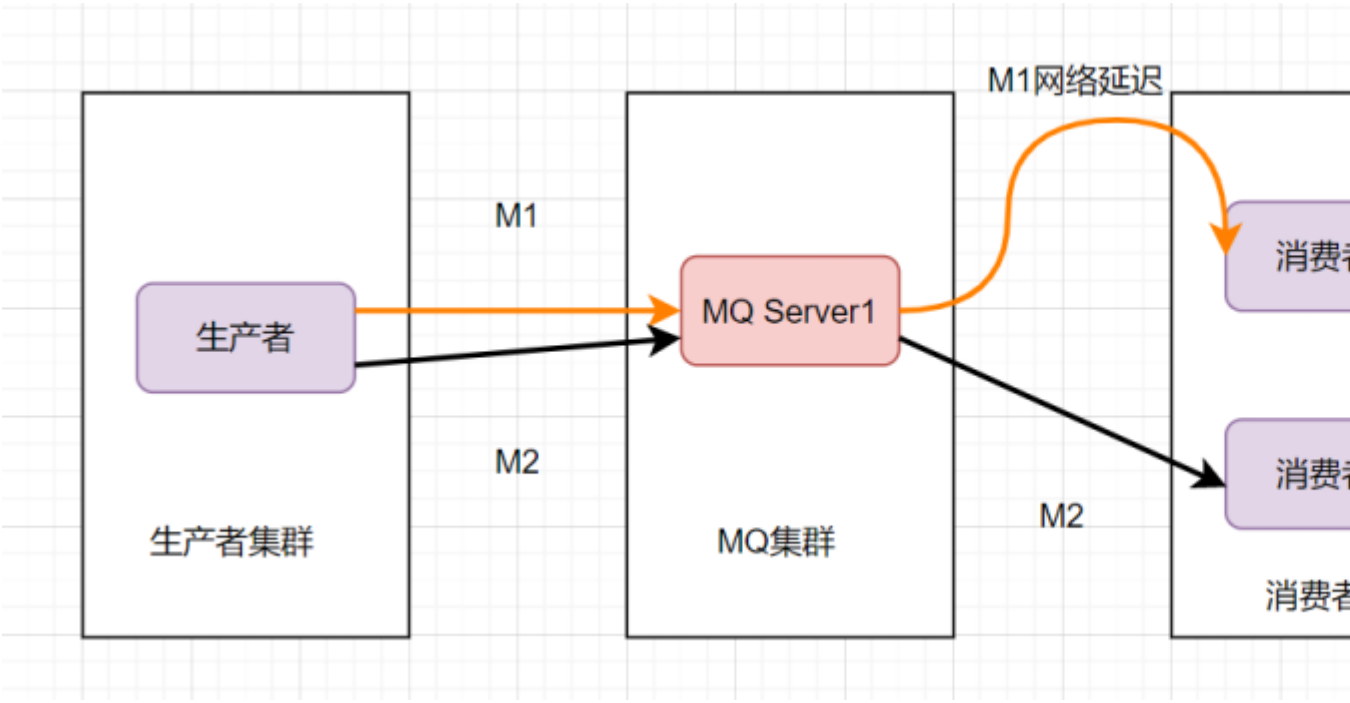
消息的有序性，就是指可以按照消息的发送顺序来消费。有些业务对消息的顺序是有要求的，比如**先下单再付款，最后再完成订单**，这样等。假设生产者先后产生了两条消息，分别是下单消息（M1），付款消息（M2），M1 比 M2 先产生，**如何保证 M1 比 M2 先被消费**呢。



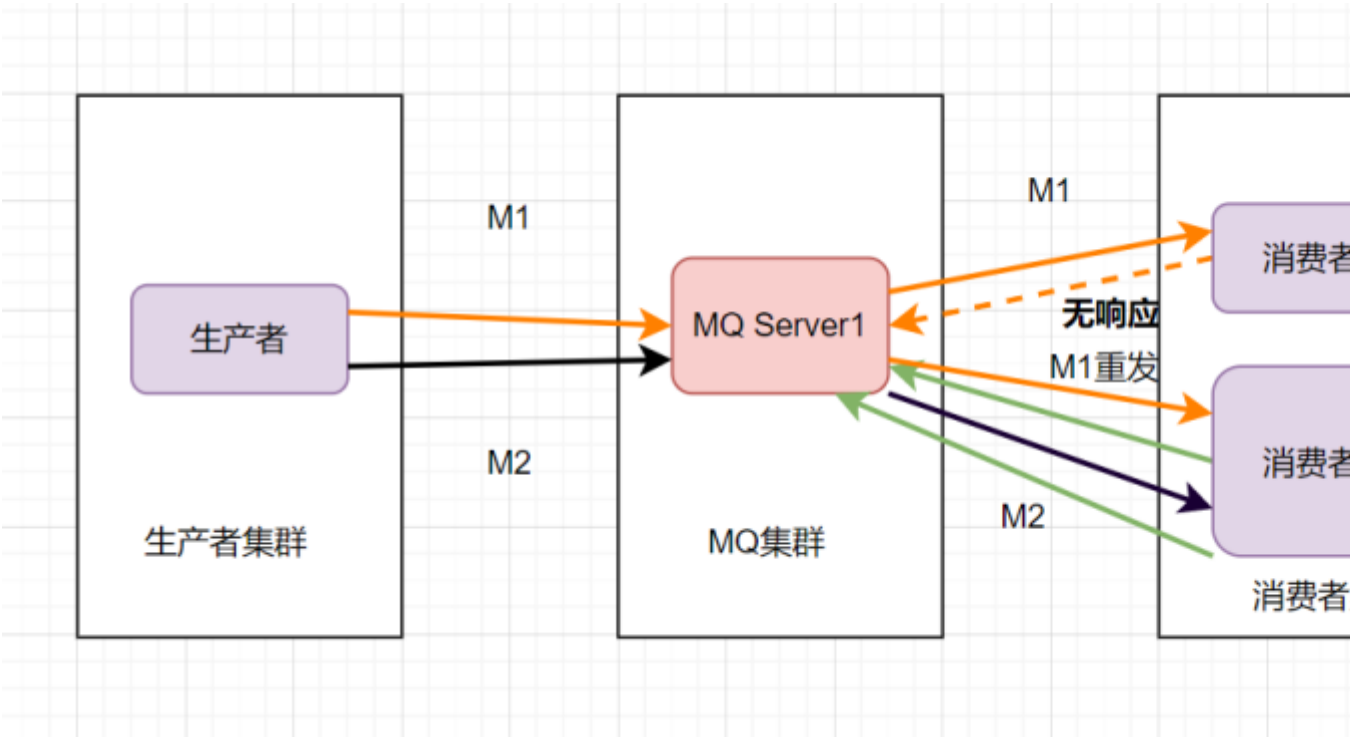
为了保证消息的顺序性，可以**将 M1、M2 发送到同一个 Server** 上，当 M1 发送完收到 ack 后，M2 再发送。如图：



这样还是可能会有问题，因为从 MQ 服务器到消费端，可能**存在网络延迟**，虽然 M1 先发送，但是它**比 M2 晚到**。



那还能怎么办才能保证消息的顺序性呢？将 M1 和 M2 发往**同一个消费者**，且发送 M1 后，等到消费端 ACK 成功后，才发送 M2 就得了。



消息队列保证顺序性**整体思路**就是这样啦。比如 Kafka 的全局有序消息，就是**这种思想**的体现：就是生产者发消息时，1 个 Topic 只能对应 1 个 Partition，一个 Consumer，内部单线程消费。

但是这样**吞吐量太低**，一般保证消息**局部有序**即可。在发消息的时候指定 Partition Key，Kafka 对其进行 Hash 计算，根据计算结果决定放入哪个 Partition。这样 Partition Key 相同的消息会放在同一个 Partition。然后多消费者单线程消费指定的 Partition。

11.5 消息队列有可能发生重复消费，如何避免，如何做到幂等？

消息队列是可能发生重复消费的。

- 生产端为了保证消息的可靠性，它可能往 MQ 服务器重复发送消息，直到拿到成功的 ACK。
- 再然后就是消费端，消费端消费消息一般是这个流程：**拉取消息、业务逻辑处理、提交消费位移**。假设业务逻辑处理完，事务提交了，但是需要更新消费位移时，消费者却挂了，这时候另一个消费者就会拉到重复消息了。

如何幂等处理重复消息呢？

我之前写过一篇幂等设计的文章，大家有兴趣可以看下哈：聊聊幂等设计

幂等处理重复消息，简单来说，就是搞个本地表，带**唯一业务标记的**，利用主键或者唯一性索引，每次处理业务，先校验一下就好啦。又或者用 redis 缓存下业务标记，每次看下是否处理过了。

11.6 如何处理消息队列的消息积压问题

消息积压是因为生产者的生产速度，大于消费者的消费速度。遇到消息积压问题时，我们需要先排查，是不是有 bug 产生了。

如果不是 bug，我们可以**优化一下消费的逻辑**，比如之前是一条一条消息消费处理的话，我们可以确认是不是可以优为**批量处理消息**。如果还是慢，我们可以考虑水平扩容，增加 Topic 的队列数，和消费组机器的数量，提升整体消费能力。

如果是 bug 导致几百万消息持续积压几小时。有如何处理呢？需要解决 bug，**临时紧急扩容**，大概思路如下：

1. 先修复 consumer 消费者的问题，以确保其恢复消费速度，然后将现有 consumer 都停掉。
2. 新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。
3. 然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。
4. 接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。
5. 等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

11.7 消息队列技术选型，Kafka 还是 RocketMQ，还是 RabbitMQ

先可以对比下它们优缺点：

	Kafka	RocketMQ	RabbitMQ
单机吞吐量	17.3w/s	11.6w/s	2.6w/s（消息做持久化）
开发语言	Scala/Java	Java	Erlang

	Kafka	RocketMQ	RabbitMQ
主要维护者	Apache	Alibaba	Mozilla/Spring
订阅形式	基于 topic，按照 topic 进行正则匹配的发布订阅模式	基于 topic/messageTag，按照消息类型、属性进行正则匹配的发布订阅模式	提供了 4 种：direct，topic ，Headers 和 fanout。fanout 就是广播模式
持久化	支持大量堆积	支持大量堆积	支持少量堆积
顺序消息	支持	支持	不支持
集群方式	天然的 Leader-Slave，无状态集群，每台服务器既是 Master 也是 Slave	常用 多对’Master-Slave’ 模式，开源版本需手动切换 Slave 变成 Master	支持简单集群，’复制’模式，对高级集群模式支持不好。
性能稳定性	较差	一般	好

- RabbitMQ 是开源的，比较稳定的支持，活跃度也高，但是不是 Java 语言开发的。
- 很多公司用 RocketMQ，比较成熟，是阿里出品的。
- 如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的。

11.8 消息中间件如何做到高可用

消息中间件如何保证高可用呢？单机是没有高可用可言的，高可用都是对集群来说的，一起看下 kafka 的高可用吧。

Kafka 的基础集群架构，由多个 broker 组成，每个 broker 都是一个节点。当你创建一个 topic 时，它可以划分为多个 partition，而每个 partition 放一部分数据，分别存在于不同的 broker 上。也就是说，一个 topic 的数据，是分散放在多个机器上的，每个机器就放一部分数据。

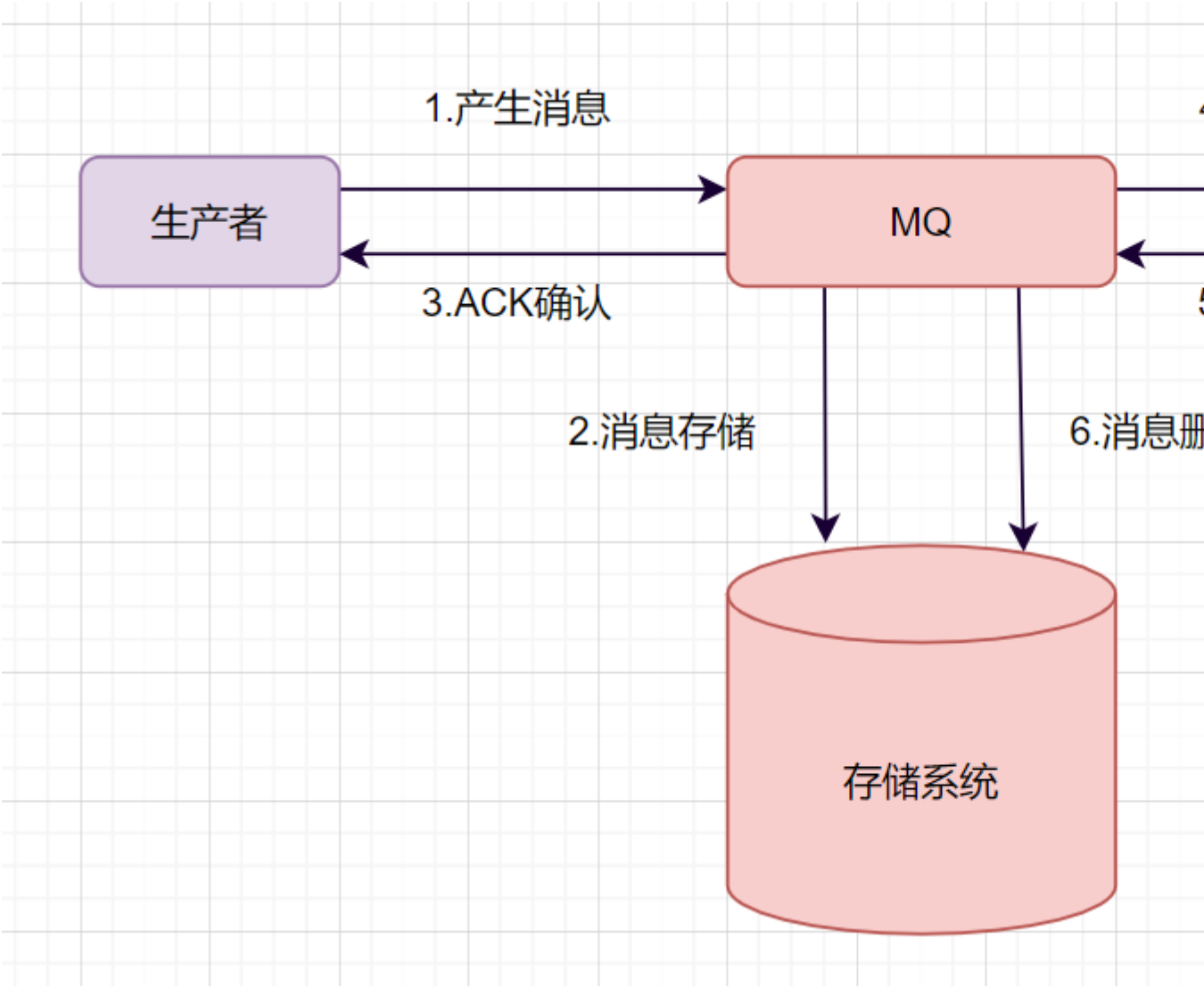
有些伙伴可能有疑问，每个 partition 放一部分数据，如果对应的 broker 挂了，那这部分数据是不是就丢失了？那还谈什么高可用呢？

Kafka 0.8 之后，提供了复制品副本机制来保证高可用，即每个 partition 的数据都会同步到其它机器上，形成多个副本。然后所有的副本会选举一个 leader

出来，让 leader 去跟生产和消费者打交道，其他副本都是 follower。写数据时，leader 负责把数据同步给所有的 follower，读消息时， 直接读 leader 上的数据即可。如何保证高可用的？就是假设某个 broker 宕机，这个 broker 上的 partition 在其他机器上都有副本的。如果挂的是 leader 的 broker 呢？其他 follower 会重新选一个 leader 出来。

11.9 如何保证数据一致性，事务消息如何实现

一条普通的 MQ 消息，从产生到被消费，大概流程如下：

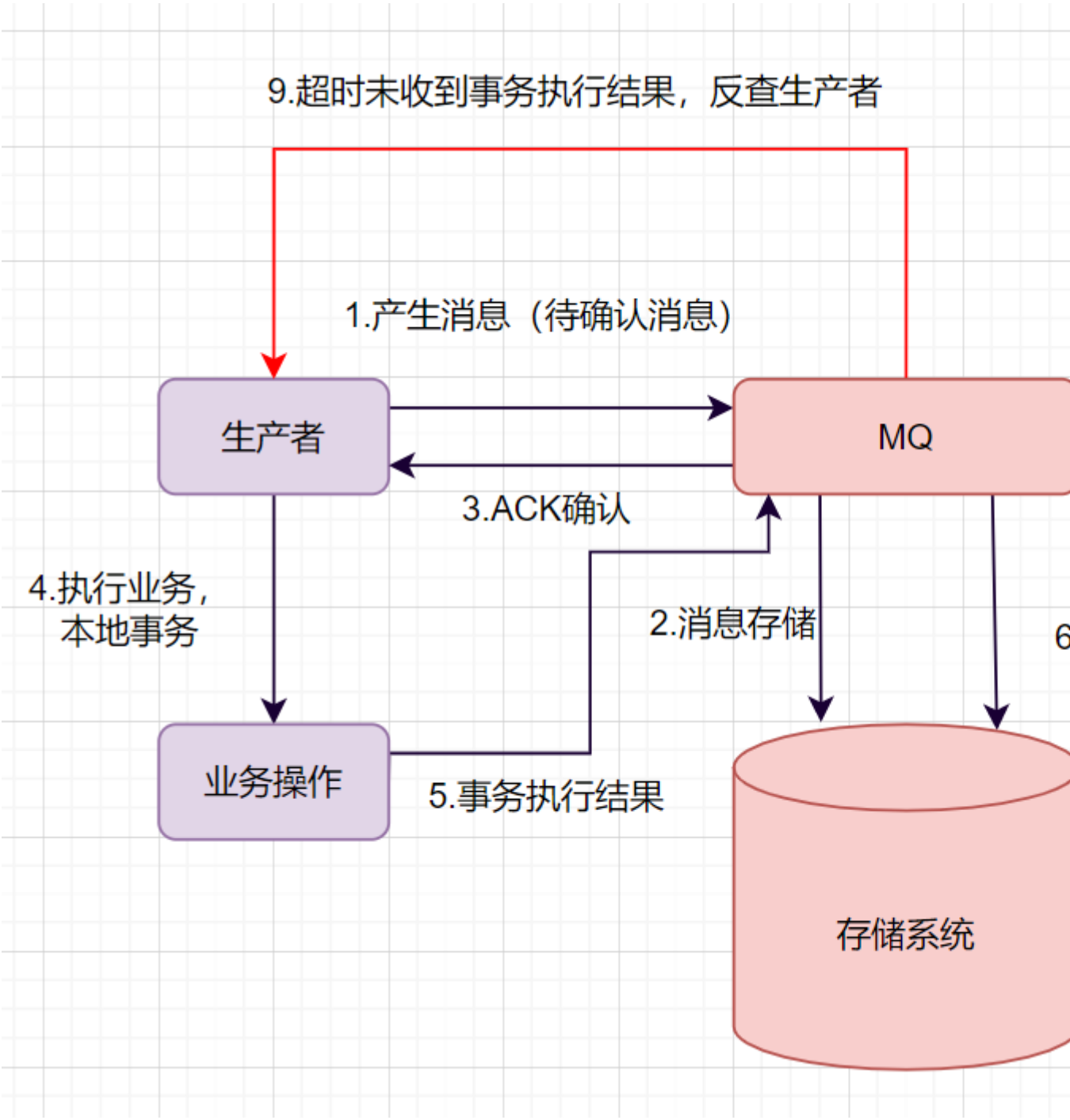


1. 生产者产生消息，发送带 MQ 服务器
2. MQ 收到消息后，将消息持久化到存储系统。
3. MQ 服务器返回 ACk 到生产者。

4. MQ 服务器把消息 push 给消费者
5. 消费者消费完消息，响应 ACK
6. MQ 服务器收到 ACK，认为消息消费成功，即在存储中删除消息。

我们举个**下订单**的例子吧。订单系统创建完订单后，再发送消息给下游系统。如果订单创建成功，然后消息没有成功发送出去，下游系统就无法感知这个事情，出导致数据不一致。

如何保证数据一致性呢？可以使用**事务消息**。一起来看下事务消息是如何实现的吧。



1. 生产者产生消息，发送一条**半事务消息**到 MQ 服务器
2. MQ 收到消息后，将消息持久化到存储系统，这条消息的状态是**待发送**状态。
3. MQ 服务器返回 ACK 确认到生产者，此时 MQ 不会触发消息推送事件
4. 生产者执行本地事务
5. 如果本地事务执行成功，即 commit 执行结果到 MQ 服务器；如果执行失败，发送 rollback。

6. 如果是正常的 commit，MQ 服务器更新消息状态为**可发送**；如果是 rollback，即删除消息。
7. 如果消息状态更新为可发送，则 MQ 服务器会 push 消息给消费者。消费者消费完就回 ACK。
8. 如果 MQ 服务器长时间没有收到生产者的 commit 或者 rollback，它会反查生产者，然后根据查询到的结果执行最终状态。

11.10 让你写一个消息队列，该如何进行架构设计？

这个问题面试官主要考察三个方面的知识点：

- 你有没有对消息队列的架构原理比较了解
- 考察你的个人设计能力
- 考察编程思想，如什么高可用、可扩展性、幂等等等。

遇到这种设计题，大部分人会很蒙圈，因为平时没有思考过类似的问题。大多数人平时埋头增删改啥，不去思考框架背后的一些原理。有很多类似的问题，比如让你来设计一个 Dubbo 框架，或者让你来设计一个 MyBatis 框架，你会怎么思考呢？

回答这类问题，并不要求你研究过那技术的源码，你知道那个技术框架的基本结构、工作原理即可。设计一个消息队列，我们可以从这几个角度去思考：

- 整体架构
- RPC 如何设计
- 如何持久化
- 消息关系设计
- 消息可靠性
- 消息队列高可用
- 消息事务控制
- 可扩展性

1. 首先是消息队列的整体流程，producer 发送消息给 broker，broker 存储好，broker 再发送给 consumer 消费，consumer 回复消费确认等。
2. producer 发送消息给 broker，broker 发消息给 consumer 消费，那就需要两次 RPC 了，RPC 如何设计呢？可以参考开源框架 Dubbo，你可以说说服务发现、序列化协议等等
3. broker 考虑如何持久化呢，是放文件系统还是数据库呢，会不会消息堆积呢，消息堆积如何处理呢。
4. 消费关系如何保存呢？点对点还是广播方式呢？广播关系又是如何维护呢？zk 还是 config server
5. 消息可靠性如何保证呢？如果消息重复了，如何幂等处理呢？

6. 消息队列的高可用如何设计呢？可以参考 Kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。
7. 消息事务特性，与本地业务同个事务，本地消息落库;消息投递到服务端，本地才删除；定时任务扫描本地消息库，补偿发送。
8. MQ 得伸缩性和可扩展性，如果消息积压或者资源不够时，如何支持快速扩容，提高吞吐？可以参照一下 Kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

12 总结(Summary)