# 4、常见面试题

## 为什么使用消息队列？

其实就是问问你消息队列都有哪些使用场景，然后你项目里具体是什么场景，说说你在这个场景里用消息队列是什么？

面试官问你这个问题，期望的一个回答是说，你们公司有个什么业务场景，这个业务场景有个什么技术挑战，如果不用MQ可能会很麻烦，但是你现在用了MQ之后带给了你很多的好处。消息队列的常见使用场景，其实场景有很多，但是比较核心的有3个：解耦、异步、削峰。

### 解耦：

A系统发送个数据到BCD三个系统，接口调用发送，那如果E系统也要这个数据呢？那如果C系统现在不需要了呢？现在A系统又要发送第二种数据了呢？而且A系统要时时刻刻考虑BCDE四个系统如果挂了咋办？要不要重发？我要不要把消息存起来？

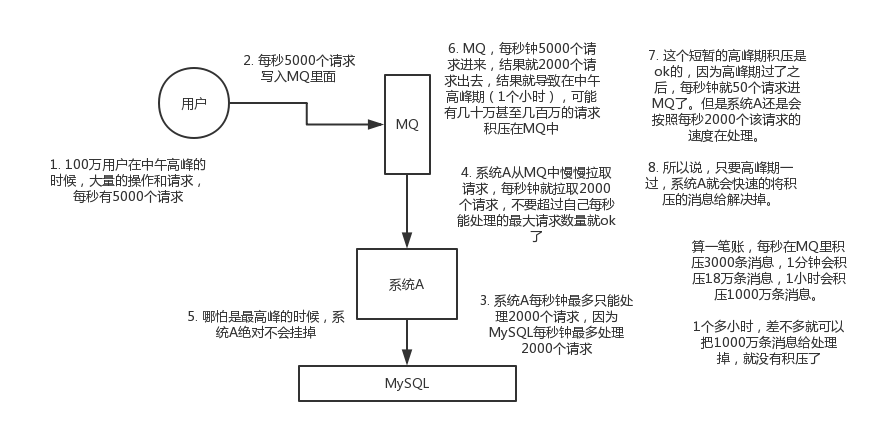
你需要去考虑一下你负责的系统中是否有类似的场景，就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用MQ给他异步化解耦，也是可以的，你就需要去考虑在你的项目里，是不是可以运用这个MQ去进行系统的解耦。

### 异步：

A系统接收一个请求，需要在自己本地写库，还需要在BCD三个系统写库，自己本地写库要30ms，BCD三个系统分别写库要300ms、450ms、200ms。最终请求总延时是30 + 300 + 450 + 200 = 980ms，接近1s，异步后，BCD三个系统分别写库的时间，A系统就不再考虑了。

### 削峰：

每天0点到16点，A系统风平浪静，每秒并发请求数量就100个。结果每次一到16点~23点，每秒并发请求数量突然会暴增到1万条。但是系统最大的处理能力就只能是每秒钟处理1000个请求啊。怎么办？需要我们进行流量的削峰，让系统可以平缓的处理突增的请求。



## 消息队列有什么优点和缺点?

优点上面已经说了，就是在特殊场景下有其对应的好处，解耦、异步、削峰。

缺点呢？

### 系统可用性降低

系统引入的外部依赖越多，越容易挂掉，本来你就是A系统调用BCD三个系统的接口就好了，ABCD四个系统好好的，没啥问题，你偏加个MQ进来，万一MQ挂了怎么办？MQ挂了，整套系统崩溃了，业务也就停顿了。

### 系统复杂性提高

硬生生加个MQ进来，怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？

### 一致性问题

A系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是BCD三个系统那里，BD两个系统写库成功了，结果C系统写库失败了，你这数据就不一致了。

所以消息队列实际是一种非常复杂的架构，你引入它有很多好处，但是也得针对它带来的坏处做各种额外的技术方案和架构来规避掉。

## 常见消息队列的比较

参见第一章中消息队列的比较章节。

## 消息的重复

### 原因

#### 第一类原因

消息发送端应用的消息重复发送,有以下几种情况。

* 消息发送端发送消息给消息中间件,消息中间件收到消息并成功存储,而这时消息中间件出现了问题,导致应用端没有收到消息发送成功的返回因而进行重试产生了重复。
* 消息中间件因为负载高响应变慢,成功把消息存储到消息存储中后,返回“成功”这个结果时超时。
* 消息中间件将消息成功写入消息存储,在返回结果时网络出现问题,导致应用发送端重试,而重试时网络恢复,由此导致重复。

可以看到,通过消息发送端产生消息重复的主要原因是消息成功进入消息存储后,因为各种原因使得消息发送端没有收到“成功”的返回结果,并且又有重试机制,因而导致重复。

#### 第二类原因

消息到达了消息存储,由消息中间件进行向外的投递时产生重复，有以下几种情况。

* 消息被投递到消息接收者应用进行处理,处理完毕后应用出问题了,消息中间件不知道消息处理结果,会再次投递。
* 消息被投递到消息接收者应用进行处理,处理完毕后网络出现问题了,消息中间件没有收到消息处理结果,会再次投递。
* 消息被投递到消息接收者应用进行处理,处理时间比较长,消息中间件因为消息超时会再次投递。
* 消息被投递到消息接收者应用进行处理,处理完毕后消息中间件出问题了,没能收到消息结果并处理,会再次投递
* 消息被投递到消息接收者应用进行处理,处理完毕后消息中间件收到结果但是遇到消息存储故障,没能更新投递状态,会再次投递。

可以看到,在投递过程中产生的消息重复接收主要是因为消息接收者成功处理完消息后,消息中间件不能及时更新投递状态造成的。

### 如何解决重复消费

那么有什么办法可以解决呢?主要是要求消息接收者来处理这种重复的情况,也就是要求消息接收者的消息处理是幂等操作。

#### 什么是幂等性？

对于消息接收端的情况,幂等的含义是采用同样的输入多次调用处理函数,得到同样的结果。例如，一个SQL操作

update stat\_table set count= 10 where id =1

这个操作多次执行,id等于1的记录中的 count字段的值都为10,这个操作就是幂等的,我们不用担心这个操作被重复。

再来看另外一个SQL操作

update stat\_table set count= count +1 where id= 1;

这样的SQL操作就不是幂等的,一旦重复,结果就会产生变化。

#### 常见办法

因此应对消息重复的办法是,使消息接收端的处理是一个幂等操作。这样的做法降低了消息中间件的整体复杂性,不过也给使用消息中间件的消息接收端应用带来了一定的限制和门槛。

##### 1. MVCC：

多版本并发控制，乐观锁的一种实现，在生产者发送消息时进行数据更新时需要带上数据的版本号，消费者去更新时需要去比较持有数据的版本号，版本号不一致的操作无法成功。例如博客点赞次数自动+1的接口：

public boolean addCount(Long id, Long version);

update blogTable set count= count+1,version=version+1 where id=321 and version=123

每一个version只有一次执行成功的机会，一旦失败了生产者必须重新获取数据的最新版本号再次发起更新。

##### 2. 去重表：

利用数据库表单的特性来实现幂等，常用的一个思路是在表上构建唯一性索引，保证某一类数据一旦执行完毕，后续同样的请求不再重复处理了（利用一张日志表来记录已经处理成功的消息的ID，如果新到的消息ID已经在日志表中，那么就不再处理这条消息。）

以电商平台为例子，电商平台上的订单id就是最适合的token。当用户下单时，会经历多个环节，比如生成订单，减库存，减优惠券等等。每一个环节执行时都先检测一下该订单id是否已经执行过这一步骤，对未执行的请求，执行操作并缓存结果，而对已经执行过的id，则直接返回之前的执行结果，不做任何操作。这样可以在最大程度上避免操作的重复执行问题，缓存起来的执行结果也能用于事务的控制等。

## 消息的可靠性传输

### ActiveMQ

要保证消息的可靠性，除了消息的持久化，还包括两个方面，一是生产者发送的消息可以被ActiveMQ收到，二是消费者收到了ActiveMQ发送的消息。

##### 生产者

非持久化又不在事务中的消息，可能会有消息的丢失。为保证消息可以被ActiveMQ收到，我们应该采用事务消息或持久化消息。

##### 消费者

对消息的确认有4种机制

* 1. AUTO\_ACKNOWLEDGE = 1    自动确认
  2. CLIENT\_ACKNOWLEDGE = 2    客户端手动确认
  3. DUPS\_OK\_ACKNOWLEDGE = 3    自动批量确认
  4. SESSION\_TRANSACTED = 0    事务提交并确认

ACK\_MODE描述了Consumer与broker确认消息的方式(时机),比如当消息被Consumer接收之后,Consumer将在何时确认消息。所以ack\_mode描述的不是producer于broker之间的关系，而是customer于broker之间的关系。

对于broker而言，只有接收到ACK指令,才会认为消息被正确的接收或者处理成功了,通过ACK，可以在consumer与Broker之间建立一种简单的“担保”机制.

###### AUTO\_ACKNOWLEDGE

自动确认

    “同步”(receive)方法返回message给消息时会立即确认。

     在"异步"(messageListener)方式中,将会首先调用listener.onMessage(message)，如果onMessage方法正常结束,消息将会正常确认。如果onMessage方法异常，将导致消费者要求ActiveMQ重发消息。

###### CLIENT\_ACKNOWLEDGE :

客户端手动确认，这就意味着AcitveMQ将不会“自作主张”的为你ACK任何消息，开发者需要自己择机确认。

我们可以在当前消息处理成功之后，立即调用message.acknowledge()方法来"逐个"确认消息，这样可以尽可能的减少因网络故障而导致消息重发的个数；当然也可以处理多条消息之后，间歇性的调用acknowledge方法来一次确认多条消息，减少ack的次数来提升consumer的效率，不过需要自行权衡。

###### DUPS\_OK\_ACKNOWLEDGE

类似于AUTO\_ACK确认机制，为自动批量确认而生，而且具有“延迟”确认的特点，ActiveMQ会根据内部算法，在收到一定数量的消息自动进行确认。在此模式下，可能会出现重复消息，什么时候？当consumer故障重启后，那些尚未ACK的消息会重新发送过来。

###### SESSION\_TRANSACTED

当session使用事务时，就是使用此模式。当决定事务中的消息可以确认时，必须调用session.commit()方法，commit方法将会导致当前session的事务中所有消息立即被确认。在事务开始之后的任何时机调用rollback()，意味着当前事务的结束，事务中所有的消息都将被重发。当然在commit之前抛出异常，也会导致事务的rollback。

### RabbitMQ

#### （1）生产者弄丢了数据

生产者将数据发送到RabbitMQ的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用RabbitMQ提供的事务功能，就是生产者发送数据之前开启RabbitMQ事务（channel.txSelect），然后发送消息，如果消息没有成功被RabbitMQ接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，RabbitMQ事务机制一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果要确保RabbitMQ的消息别丢，可以开启confirm模式，在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了RabbitMQ中，RabbitMQ会给你回传一个ack消息，告诉你说这个消息ok了。如果RabbitMQ没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和cnofirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息RabbitMQ接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用confirm机制的。

#### （2）RabbitMQ弄丢了数据

就是RabbitMQ自己弄丢了数据，这个你必须开启RabbitMQ的持久化，就是消息写入之后会持久化到磁盘，哪怕是RabbitMQ自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建queue和交换器的时候将其设置为持久化的，这样就可以保证RabbitMQ持久化相关的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时RabbitMQ就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。

而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，RabbitMQ挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。

哪怕是你给RabbitMQ开启了持久化机制，也有一种可能，就是这个消息写到了RabbitMQ中，但是还没来得及持久化到磁盘上，结果不巧，此时RabbitMQ挂了，就会导致内存里的一点点数据会丢失。

#### （3）消费端弄丢了数据

RabbitMQ如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ认为你都消费了，这数据就丢了。

这个时候得用RabbitMQ提供的ack机制，简单来说，就是你关闭RabbitMQ自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那RabbitMQ就认为你还没处理完，这个时候RabbitMQ会把这个消费分配给别的consumer去处理，消息是不会丢的。

### Kafka

#### （1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。

然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了

#### （2）kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partiton的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。

所以此时一般是要求起码设置如下4个参数：

给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本。

在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个是要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧。

在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了。

在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

#### （3）生产者会不会弄丢数据

如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

## 消息的顺序性

从根本上说，异步消息是不应该有顺序依赖的。在MQ上估计是没法解决。要实现严格的顺序消息，简单且可行的办法就是：保证生产者 - MQServer - 消费者是一对一对一的关系。

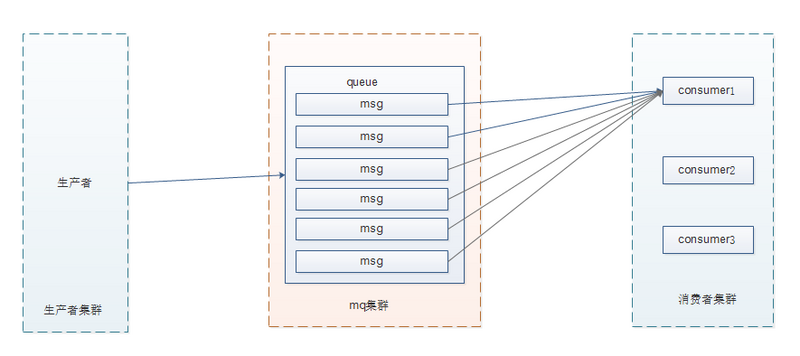
### ActiveMQ

1、通过高级特性consumer独有消费者（exclusive consumer）

queue = new ActiveMQQueue("TEST.QUEUE?consumer.exclusive=true");

consumer = session.createConsumer(queue);

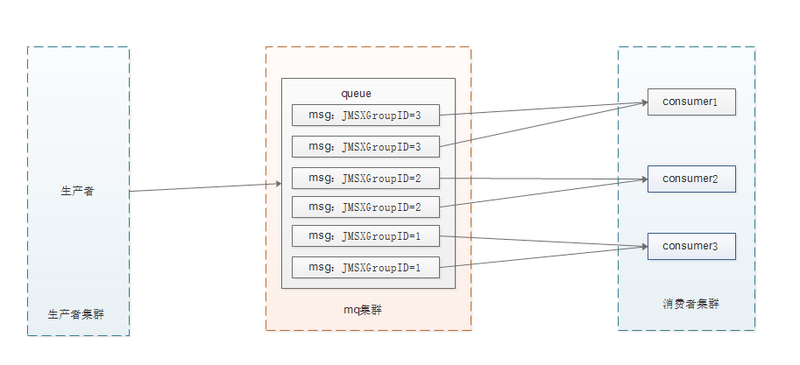
当在接收信息的时候，有多个独占消费者的时候，只有一个独占消费者可以接收到消息。



独占消息就是在有多个消费者同时消费一个queue时，可以保证只有一个消费者可以消费消息，这样虽然保证了消息的顺序问题，不过也带来了一个问题，就是这个queue的所有消息将只会在这一个主消费者上消费，其他消费者将闲置，达不到负载均衡分配，而实际业务我们可能更多的是这样的场景，比如一个订单会发出一组顺序消息，我们只要求这一组消息是顺序消费的，而订单与订单之间又是可以并行消费的，不需要顺序，因为顺序也没有任何意义，有没有办法做到呢？可以利用activemq的另一个高级特性之messageGroup

2、利用Activemq的高级特性：messageGroups

Message Groups特性是一种负载均衡的机制。在一个消息被分发到consumer之前，broker首先检查消息JMSXGroupID属性。如果存在，那么broker会检查是否有某个consumer拥有这个message group。如果没有，那么broker会选择一个consumer，并将它关联到这个message group。此后，这个consumer会接收这个message group的所有消息，直到：Consumer被关闭。Message group被关闭，通过发送一个消息，并设置这个消息的JMSXGroupSeq为-1



**bytesMessage**.setStringProperty("JMSXGroupID", "constact-20100000002");

**bytesMessage**.setIntProperty("JMSXGroupSeq", -1);

如上图所示，同一个queue中，拥有相同JMSXGroupID的消息将发往同一个消费者，解决顺序问题，不同分组的消息又能被其他消费者并行消费，解决负载均衡的问题。

### RabbitMQ

如果有顺序依赖的消息，要保证消息有一个hashKey，类似于数据库表分区的的分区key列。保证对同一个key的消息发送到相同的队列。A用户产生的消息（包括创建消息和删除消息）都按A的hashKey分发到同一个队列。只需要把强相关的两条消息基于相同的路由就行了，也就是说经过m1和m2的在路由表里的路由是一样的，那自然m1会优先于m2去投递。而且一个queue只对应一个consumer。

### Kafka

一个topic，一个partition，一个consumer，内部单线程消费

## 如何解决消息队列的延时以及过期失效问题？

rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间，而又没有设置死信队列机制，就会被rabbitmq给清理掉，这个数据就没了。

ActiveMQ则通过更改配置，支持消息的定时发送。

## 有几百万消息持续积压几小时怎么解决？

发生了线上故障，几千万条数据在MQ里积压很久。是修复consumer的问题，让他恢复消费速度，然后等待几个小时消费完毕？这是个解决方案。不过有时候我们还会进行临时紧急扩容。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条。1000多万条，所以如果积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来。

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉。

新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量。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。

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。

等快速消费完积压数据之后，再恢复原先部署架构，重新用原先的consumer机器来消费消息。

## Kafka是如何实现高性能的？

### 宏观架构层面利用Partition实现并行处理

Kafka中每个Topic都包含一个或多个Partition，不同Partition可位于不同节点。同时Partition在物理上对应一个本地文件夹，每个Partition包含一个或多个Segment，每个Segment包含一个数据文件和一个与之对应的索引文件。在逻辑上，可以把一个Partition当作一个非常长的数组，可通过这个“数组”的索引（offset）去访问其数据。

一方面，由于不同Partition可位于不同机器，因此可以充分利用集群优势，实现机器间的并行处理。另一方面，由于Partition在物理上对应一个文件夹，即使多个Partition位于同一个节点，也可通过配置让同一节点上的不同Partition置于不同的disk drive上，从而实现磁盘间的并行处理，充分发挥多磁盘的优势。

利用多磁盘的具体方法是，将不同磁盘mount到不同目录，然后在server.properties中，将log.dirs设置为多目录（用逗号分隔）。Kafka会自动将所有Partition尽可能均匀分配到不同目录也即不同目录（也即不同disk）上。

Partition是最小并发粒度，Partition个数决定了可能的最大并行度。。

### ISR实现可用性与数据一致性的动态平衡

#### 常用数据复制及一致性方案

##### Master-Slave

- RDBMS的读写分离即为典型的Master-Slave方案

- 同步复制可保证强一致性但会影响可用性

- 异步复制可提供高可用性但会降低一致性

##### WNR

- 主要用于去中心化的分布式系统中。

- N代表总副本数，W代表每次写操作要保证的最少写成功的副本数，R代表每次读至少要读取的副本数

- 当W+R>N时，可保证每次读取的数据至少有一个副本拥有最新的数据

- 多个写操作的顺序难以保证，可能导致多副本间的写操作顺序不一致。Dynamo通过向量时钟保证最终一致性

##### Paxos及其变种

- Google的Chubby，Zookeeper的原子广播协议（Zab），RAFT等

#### 基于ISR的数据复制方案

Kafka的数据复制是以Partition为单位的。而多个备份间的数据复制，通过Follower向Leader拉取数据完成。从一这点来讲，Kafka的数据复制方案接近于上文所讲的Master-Slave方案。不同的是，Kafka既不是完全的同步复制，也不是完全的异步复制，而是基于ISR的动态复制方案。

ISR，也即In-sync Replica。每个Partition的Leader都会维护这样一个列表，该列表中，包含了所有与之同步的Replica（包含Leader自己）。每次数据写入时，只有ISR中的所有Replica都复制完，Leader才会将其置为Commit，它才能被Consumer所消费。

这种方案，与同步复制非常接近。但不同的是，这个ISR是由Leader动态维护的。如果Follower不能紧“跟上”Leader，它将被Leader从ISR中移除，待它又重新“跟上”Leader后，会被Leader再次加加ISR中。每次改变ISR后，Leader都会将最新的ISR持久化到Zookeeper中。

由于Leader可移除不能及时与之同步的Follower，故与同步复制相比可避免最慢的Follower拖慢整体速度，也即ISR提高了系统可用性。

ISR中的所有Follower都包含了所有Commit过的消息，而只有Commit过的消息才会被Consumer消费，故从Consumer的角度而言，ISR中的所有Replica都始终处于同步状态，从而与异步复制方案相比提高了数据一致性。

ISR可动态调整，极限情况下，可以只包含Leader，极大提高了可容忍的宕机的Follower的数量。与Majority Quorum方案相比，容忍相同个数的节点失败，所要求的总节点数少了近一半。

### 具体实现层面高效使用磁盘特性和操作系统特性

#### 将写磁盘的过程变为顺序写

Kafka的整个设计中，Partition相当于一个非常长的数组，而Broker接收到的所有消息顺序写入这个大数组中。同时Consumer通过Offset顺序消费这些数据，并且不删除已经消费的数据，从而避免了随机写磁盘的过程。

由于磁盘有限，不可能保存所有数据，实际上作为消息系统Kafka也没必要保存所有数据，需要删除旧的数据。而这个删除过程，并非通过使用“读-写”模式去修改文件，而是将Partition分为多个Segment，每个Segment对应一个物理文件，通过删除整个文件的方式去删除Partition内的数据。这种方式清除旧数据的方式，也避免了对文件的随机写操作。

在存储机制上，使用了Log Structured Merge Trees(LSM) 。

注：Log Structured Merge Trees(LSM)，谷歌 “BigTable” 的论文，中提出，LSM是当前被用在许多产品的文件结构策略：HBase, Cassandra, LevelDB, SQLite,Kafka。LSM被设计来提供比传统的B+树或者ISAM更好的写操作吞吐量，通过消去随机的本地更新操作来达到这个目标。这个问题的本质还是磁盘随机操作慢，顺序读写快。这二种操作存在巨大的差距，无论是磁盘还是SSD，而且快至少三个数量级。

#### 充分利用Page Cache

使用Page Cache的好处如下

- I/O Scheduler会将连续的小块写组装成大块的物理写从而提高性能

- I/O Scheduler会尝试将一些写操作重新按顺序排好，从而减少磁盘头的移动时间

- 充分利用所有空闲内存（非JVM内存）。如果使用应用层Cache（即JVM堆内存），会增加GC负担

- 读操作可直接在Page Cache内进行。如果消费和生产速度相当，甚至不需要通过物理磁盘（直接通过Page Cache）交换数据

- 如果进程重启，JVM内的Cache会失效，但Page Cache仍然可用

Broker收到数据后，写磁盘时只是将数据写入Page Cache，并不保证数据一定完全写入磁盘。从这一点看，可能会造成机器宕机时，Page Cache内的数据未写入磁盘从而造成数据丢失。但是这种丢失只发生在机器断电等造成操作系统不工作的场景，而这种场景完全可以由Kafka层面的Replication机制去解决。如果为了保证这种情况下数据不丢失而强制将Page Cache中的数据Flush到磁盘，反而会降低性能。也正因如此，Kafka虽然提供了flush.messages和flush.ms两个参数将Page Cache中的数据强制Flush到磁盘，但是Kafka并不建议使用。

如果数据消费速度与生产速度相当，甚至不需要通过物理磁盘交换数据，而是直接通过Page Cache交换数据。同时，Follower从Leader Fetch数据时，也可通过Page Cache完成。

注：Page Cache，又称pcache，其中文名称为页高速缓冲存储器，简称页高缓。page cache的大小为一页，通常为4K。在linux读写文件时，它用于缓存文件的逻辑内容，从而加快对磁盘上映像和数据的访问。 是Linux操作系统的一个特色。

#### 支持多Disk Drive

Broker的log.dirs配置项，允许配置多个文件夹。如果机器上有多个Disk Drive，可将不同的Disk挂载到不同的目录，然后将这些目录都配置到log.dirs里。Kafka会尽可能将不同的Partition分配到不同的目录，也即不同的Disk上，从而充分利用了多Disk的优势。

#### 零拷贝

Kafka中存在大量的网络数据持久化到磁盘（Producer到Broker）和磁盘文件通过网络发送（Broker到Consumer）的过程。这一过程的性能直接影响Kafka的整体吞吐量。

传统模式下的四次拷贝与四次上下文切换

以将磁盘文件通过网络发送为例。传统模式下，一般使用如下伪代码所示的方法先将文件数据读入内存，然后通过Socket将内存中的数据发送出去。

buffer = File.readSocket.send(buffer)

这一过程实际上发生了四次数据拷贝。首先通过系统调用将文件数据读入到内核态Buffer（DMA拷贝），然后应用程序将内存态Buffer数据读入到用户态Buffer（CPU拷贝），接着用户程序通过Socket发送数据时将用户态Buffer数据拷贝到内核态Buffer（CPU拷贝），最后通过DMA拷贝将数据拷贝到NIC Buffer。同时，还伴随着四次上下文切换。

而Linux 2.4+内核通过sendfile系统调用，提供了零拷贝。数据通过DMA拷贝到内核态Buffer后，直接通过DMA拷贝到NIC Buffer，无需CPU拷贝。这也是零拷贝这一说法的来源。除了减少数据拷贝外，因为整个读文件-网络发送由一个sendfile调用完成，整个过程只有两次上下文切换，因此大大提高了性能。

从具体实现来看，Kafka的数据传输通过Java NIO的FileChannel的transferTo和transferFrom方法实现零拷贝。

注： transferTo和transferFrom并不保证一定能使用零拷贝。实际上是否能使用零拷贝与操作系统相关，如果操作系统提供sendfile这样的零拷贝系统调用，则这两个方法会通过这样的系统调用充分利用零拷贝的优势，否则并不能通过这两个方法本身实现零拷贝。

#### 减少网络开销批处理

批处理是一种常用的用于提高I/O性能的方式。对Kafka而言，批处理既减少了网络传输的Overhead，又提高了写磁盘的效率。

Kafka 的send方法并非立即将消息发送出去，而是通过batch.size和linger.ms控制实际发送频率，从而实现批量发送。

由于每次网络传输，除了传输消息本身以外，还要传输非常多的网络协议本身的一些内容（称为Overhead），所以将多条消息合并到一起传输，可有效减少网络传输的Overhead，进而提高了传输效率。

#### 数据压缩降低网络负载

Kafka从0.7开始，即支持将数据压缩后再传输给Broker。除了可以将每条消息单独压缩然后传输外，Kafka还支持在批量发送时，将整个Batch的消息一起压缩后传输。数据压缩的一个基本原理是，重复数据越多压缩效果越好。因此将整个Batch的数据一起压缩能更大幅度减小数据量，从而更大程度提高网络传输效率。

Broker接收消息后，并不直接解压缩，而是直接将消息以压缩后的形式持久化到磁盘。Consumer Fetch到数据后再解压缩。因此Kafka的压缩不仅减少了Producer到Broker的网络传输负载，同时也降低了Broker磁盘操作的负载，也降低了Consumer与Broker间的网络传输量，从而极大得提高了传输效率，提高了吞吐量。

#### 高效的序列化方式

Kafka消息的Key和Payload（或者说Value）的类型可自定义，只需同时提供相应的序列化器和反序列化器即可。因此用户可以通过使用快速且紧凑的序列化-反序列化方式（如Avro，Protocal Buffer）来减少实际网络传输和磁盘存储的数据规模，从而提高吞吐率。这里要注意，如果使用的序列化方法太慢，即使压缩比非常高，最终的效率也不一定高。