1、消息中间件通用面试题

1.1 为什么使用消息队列？（初级）

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。

1.2 消息队列有什么优点和缺点?（中级）

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。

1.3 常见消息队列如何选择（中级）

如果一般的业务系统要引入 MQ，怎么选型：

用户访问量在 ActiveMQ 的可承受范围内，而且确实主要是基于解耦和异步来用的，可以考虑 ActiveMQ，也比较贴近 Java 工程师的使用习惯，但是 ActiveMQ 现在停止维护了，同时 ActiveMQ 并发不高，所以业务量一定的情况下可以考虑使用。

RabbitMQ 作为一个纯正血统的消息中间件，有着高级消息协议 AMQP 的完美结合，在消息中间件中地位无可取代，但是 erlang 语言阻止了我们去深入研究和掌控，对公司而言，底层技术无法控制，但是确实是开源的，有比较稳定的支持，活跃度也高。

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用 ActiveMQ、RabbitMQ 是不错的选择；大型公司，基础架构研发实力较强，用 RocketMQ 是很好的选择。

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，绝对没问题，社区活跃度很高，几乎是全世界这个领域的事实性规范。

1.4 使用了消息中间件后消息的重复及如何解决？（高级）

重复的原因

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。

2、RabbitMQ

2.1 描述下 RabbitMQ 概念里的 channel、exchange 和 queue 这些概念及作用？（初级）

Queue 就是消息队列，用于存储消息，具有自己的 erlang 进程。exchange 内部实现为保存 binding 关系的查找表；channel 是实际进行路由工作的实体，即负责按照 routing_key 将 message 投递给 queue 。在 RabbitMQ 中所有客户端与 RabbitMQ 之间的通讯都是在 channel 上，channel 是真实 TCP 连接之上的虚拟连接，所有 AMQP 命令都是通过 channel 发送的。

2.2 RabbitMQ 中的元数据有哪些？（中级）

元数据主要分为 Queue 元数据（queue 名字和属性等）、Exchange 元数据（exchange 名字、类型和属性等）、Binding 元数据（存放路由关系的查找表）、Vhost 元数据（vhost 范围内针对前三者的名字空间约束和安全属性设置），另外在集群中，元数据都是在一个 broker 中都是全局复制的。

2.3 RabbitMQ 中的 vhost 是什么？起什么作用？（初级）

vhost 可以理解为虚拟 broker ，即一个迷你版的 RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最最重要的是，其拥有独立的权限系统，可以做到

vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

2.4 RabbitMQ 上的一个 queue 中存放的 message 是否有数量限制？（中级）

默认情况下一般是无限制，因为限制取决于机器的内存，但是消息过多会导致处理效率的下降。同时可以通过参数来限制， x-max-length ：对队列中消息的条数进行限制 ，x-max-length-bytes ：对队列中消息的总量进行限制。

2.5 为什么对所有的 message 都使用持久化机制？（中级）

首先，必然导致性能的下降，因为写磁盘比写内存慢的多，Rabbit 的吞吐量有 10 倍的差距。

其次，message 的持久化机制用在 RabbitMQ 的集群时会出现“坑爹”问题。矛盾点在于，要实现持久化的话，必须消息、消息队列、交换器三者持久化，如果集群中不同机器中三者属性有差异，会发生不可预料的问题。所以一般处理原则是：仅对关键消息作持久化处理（根据业务重要程度），且应该保证关键消息的量不会导致性能瓶颈。

2.6 RAM node 和 disk node 的区别？（中级）

RAM node 就是内存节点，Rabbit 中的 queue、exchange 和 binding 等 RabbitMQ 基础构件中相关元数据保存到内存中，disk node 是磁盘节点，上述数据会在内存和磁盘中均进行存储。

一般在 RabbitMQ 集群中至少存在一个 disk node.

2.7 RabbitMQ 如何确保消息的可靠性传输（高级）

因为 MQ 中涉及到了 MQ 本身，生产者和消费，所以需要从三个角度来看。

（1）生产者

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 本身

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

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。

2.8 RabbitMQ 如何保证消息的顺序性（中级）

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。

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。

3、Kafka

3.1kafka 中的 zookeeper 起到什么作用，可以不用 zookeeper 么？（初级）

zookeeper 是一个分布式的协调组件，早期版本的 kafka 用 zk 做 meta 信息存储，consumer 的消费状态，group 的管理以及 offset 的值。考虑到 zk 本身的一些因素以及整个架构较大概率存在单点问题，新版本中逐渐弱化了 zookeeper 的作用。新的 consumer 使用了 kafka 内部的 group coordination 协议，也减少了对 zookeeper 的依赖，但是 broker 依然依赖于 ZK，zookeeper 在 kafka 中还用来选举和检测 broker 是否存活等等。

3.2 kafka 中 consumer group 是什么概念（初级）

同样是逻辑上的概念，是 Kafka 实现单播和广播两种消息模型的手段。同一个 topic 的数据，会广播给不同的 group；同一个 group 中的 worker，只有一个 worker 能拿到这个数据。换句话说，对于同一个 topic，每个 group 都可以拿到同样的所有数据，但是数据进入 group 后只能被其中的一个 worker 消费。group 内的 worker 可以使用多线程或多进程来实现，也可以将进程分散在多台机器上，worker 的数量通常不超过 partition 的数量，且二者最好保持整数倍关系，因为 Kafka 在设计时假定了一个 partition 只能被一个 worker 消费（同一 group 内）。

3.3kafka 为什么那么快？（中级）

系统缓存，页面缓存技术。

顺序写：由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。

Zero-copy 零拷技术减少拷贝次数。

批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。

Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。

3.4 Kafka 中是怎么体现消息顺序性的？（中级）

kafka 每个 partition 中的消息在写入时都是有序的，消费时，每个 partition 只能被每一个 group 中的一个消费者消费，保证了消费时也是有序的。

整个 topic 不保证有序。如果为了保证 topic 整个有序，那么将 partition 调整为 1.

3.5 kafka follower 如何与 leader 同步数据（高级）

Kafka 的复制机制既不是完全的同步复制，也不是单纯的异步复制。完全同步复制要求 All Alive Follower 都复制完，这条消息才会被认为 commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower 异步的从 Leader 复制数据，数据只要被 Leader 写入 log 就被认为已经 commit，这种情况下，如果 leader 挂掉，会丢失数据，kafka 使用 ISR 的方式很好的均衡了确保数据不丢失以及吞吐率。

kafka producer 如何优化生产速度

增加线程

提高 batch.size

增加更多 producer 实例

增加 partition 数

设置 acks=-1 时，如果延迟增大：可以增大 num.replica.fetchers（follower 同步数据的线程数）来调解；

跨数据中心的传输：增加 socket 缓冲区设置以及 OS tcp 缓冲区设置。

3.6 为什么 Kafka 不支持读写分离？（高级）

在 Kafka 中，生产者写入消息、消费者读取消息的操作都是与 leader 副本进行交互的，从而实现的是一种主写主读的生产消费模型。

Kafka 并不支持主写从读，因为主写从读有 2 个很明显的缺点:

(1)数据一致性问题。数据从主节点转到从节点必然会有一个延时的时间窗口，这个时间 窗口会导致主从节点之间的数据不一致。某一时刻，在主节点和从节点中 A 数据的值都为 X， 之后将主节点中 A 的值修改为 Y，那么在这个变更通知到从节点之前，应用读取从节点中的 A 数据的值并不为最新的 Y，由此便产生了数据不一致的问题。

(2)延时问题。类似 Redis 这种组件，数据从写入主节点到同步至从节点中的过程需要经 历网络→主节点内存→网络→从节点内存这几个阶段，整个过程会耗费一定的时间。而在 Kafka 中，主从同步会比 Redis 更加耗时，它需要经历网络→主节点内存→主节点磁盘→网络→从节 点内存→从节点磁盘这几个阶段。对延时敏感的应用而言，主写从读的功能并不太适用。

3.7 有几百万消息持续积压几小时怎么解决？（高级）

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 consumer 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。

3.8 Kafka 是如何实现高性能的？（高级）

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。

4、RocketMQ

4.1 为什么选择 RocketMQ？（中级）

性能：阿里支撑，经受住淘宝，天猫双 11 重重考验；性能高；可靠性好；可用性高；易扩展

功能：功能完善，我们需要的功能，基本都够满足，如：事务消息，消息重试，私信队列，定时消息等；

易用，跨平台：跨语言，多协议接入（支持 HTTP, MQTT, TCP 协议，支持 Restful 风格 HTTP 收发消息）

钱能解决的问题，一般都不是问题，所以免费服务不能满足的，适当的花钱购买所需服务是值得的，引进的就是 RocketMQ 的阿里云和 VIP 服务；

4.2 RocketMQ 特性（中级）

顺序性

消息过滤

消息持久化

消息回溯

大量消息堆积

定时消息

消息重试

4.3 你对 Namesrv 的了解？（初级）

Name Server 为 producer 和 consumer 提供路由信息，同时还有提供服务的注册和服务的剔除。

4.4 Rocket 中消费者消费模式有几种？（中级）

1. 集群消费

消费者的一种消费模式。一个 Consumer Group 中的各个 Consumer 实例分摊去消费消息，即一条消息只会投递到一个 Consumer Group 下面的一个实例。

2. 广播消费

消费者的一种消费模式。消息将对一 个 Consumer Group 下的各个 Consumer 实例都投递一遍。即即使这些 Consumer 属于同一个 Consumer Group ，消息也会被 Consumer Group 中的每个 Consumer 都消费一次。

4.5 消费者获取消息有几种模式？（中级）

消费者获取消息有两种模式：推送模式和拉取模式。

1. PushConsumer

推送模式（虽然 RocketMQ 使用的是长轮询）的消费者。消息的能及时被消费。使用非常简单，内部已处理如线程池消费、流控、负载均衡、异常处理等等的各种场景。

2. PullConsumer

拉取模式的消费者。应用主动控制拉取的时机，怎么拉取，怎么消费等。主动权更高。但要自己处理各种场景。

4.6 RocketMQ 与 kafka 的区别是啥？rocketMQ 与 kafka 的主要使用场景？rocketMQ 的部署架构是啥样的？对数据要求较高的场景，rocketMQ 主从复制和刷盘策略如何配置？（高级）

1、区别在于：

（1）rocketMQ 支持事务消息

（2）rocketmq 支持消息失败重试机制，kafka 不支持消息失败重试机制

（3）rocketMQ 是实时同步发送，kafka 是批量异步发送，当 kafka 的生产者出现宕机，消息会出现批量丢失，数据安全不够。

2、rocketMQ 主要使用在对数据的可靠性要求较高的场景，不允许数据丢失。如充值类应用。kafka 主要用于日志等流式数据的存储使用，因此对数据的绝对安全要求不高，丢失部分数据影响不大。

3、rocketMQ 由 nameserver 集群和 broker 主从集群组成。对数据要求较高的场景，建议的持久化策略是主 broker 和从 broker 采用同步复制方式，主从 broker 都采用异步刷盘方式。通过同步复制方式，保存数据热备份，通过异步刷盘方式，保证 rocketMQ 高吞吐量。