# 前言

Kafka是一款由Scala和Java编写的高吞吐量的分布式发布-订阅消息系统(消息队列)，是Apache开源的一款顶级项目

# 概述

## 1.1消息队列的几种模式

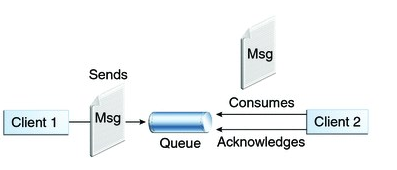
消息队列是分布式系统开发中的一种重要的中间件，在消息队列中，存在两种场景：

·生产者-消费者模式：一个消息只能有一个消费者

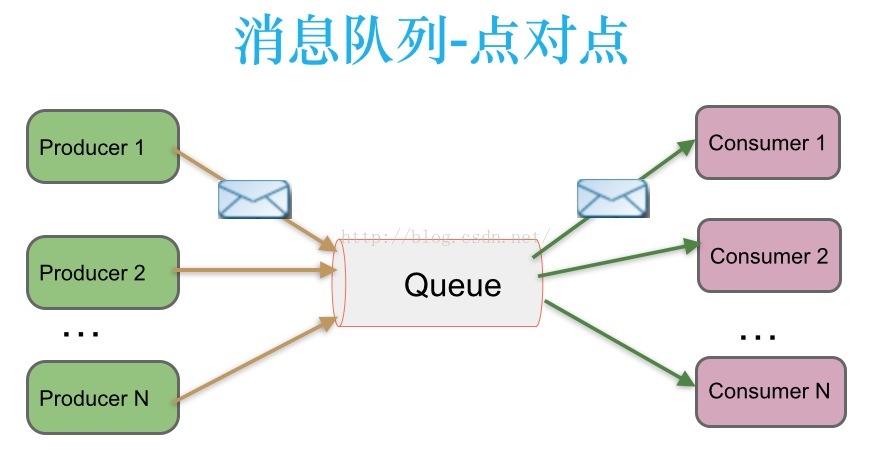
·发布者-订阅者模式：一个消息可以被多个消费者收到

### 1.1.1点对点模式

点对点模式是最原始的生产者-消费者模型，消息的生产者向一个特定的消息队列(queue)发送消息，消息的消费者从该队列中接收消息并消费(consuumes)，每一个成功处理的消息(msg)都由消息消费者签收确认（Acknowledge）



点对点的消息队列有一个特点：一个消息只能被消费一次，也就是说，当存在多个消费者时，对于一个消息，有且只有一个消费者会接受到。即一旦消费者读取队列中的消息，消息就从该队列中消失，这和发布-订阅模式有着极大的不同



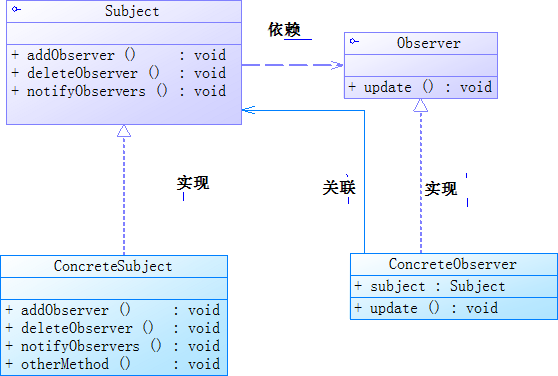
### 1.1.2发布-订阅模式

在讨论发布-订阅模式之前，我们需要先了解一种设计模式——观察者模式

#### 1.1.2.1观察者模式

什么是观察者模式？观察者模式不同于上面所说的模式，他是23中软件设计模式中的最常用的一种。它在对象之间定义了一对多的依赖，这样一来，当父对象改变状态，它的依赖对象会收到通知并自动更新。

比如一个公众号(被观察者)，公众号持有一个用户列表，存储所有关注该公众号的用户信息(观察者)，当公众号有文章推送时，就会去遍历所有持有用户，并通知他们，用户接到通知后，自行进行相应操作



从上图中可以看出，该模式包含四种角色：

* 抽象被观察者角色(Subject)：也就是一个抽象Subject，它把所有观察者对象的引用保存在一个集合中，每个Subject都可以有任意数量的观察者。抽象Subject提供可以增加和删除观察者角色的接口 为，还有一个通知观察者对象的接口。一般用一个抽象类和接口来实现。

public interface Subject {

public void registerObserver(Observer o);

public void removeObserver(Observer o);

public void notifyObserver();

}

* 抽象观察者角色(Observer)：为所有的具体观察者定义一个接口，在得到主题通知时更新自己，执行相关操作。

public interface Observer {

public void update(String message);

}

* 具体被观察者角色(ConcreateSubject)：也就是一个具体的主题，在集体主题的内部状态改变时，所有登记过的观察者发出通知。

public class ConcreateSubject implements Subject {

//注意到这个List集合的泛型参数为Observer接口，设计原则：面向接口编程而不是面向实现编程

private List<Observer> list;

private String message;

public ConcreateSubject () {

list = new ArrayList<Observer>();

}

@Override

public void registerObserver(Observer o) {

list.add(o);

}

@Override

public void removeObserver(Observer o) {

if(!list.isEmpty())

list.remove(o);

}

//遍历

@Override

public void notifyObserver() {

for(int i = 0; i < list.size(); i++) {

Observer observer = list.get(i);

observer.update(message);

}

}

public void setInfomation(String s) {

this.message = s;

System.out.println("微信服务更新消息： " + s);

//消息更新，通知所有观察者

notifyObserver();

}

}

* 具体观察者角色(ConcreateSubject)：实现抽象观察者角色所需要的更新接口，一边使本身的状态与制图的状态相协调。

public class ConcreateSubject implements Observer {

private String name;

private String message;

public User(String name) {

this.name = name;

}

@Override

public void update(String message) {

this.message = message;

read();

}

public void read() {

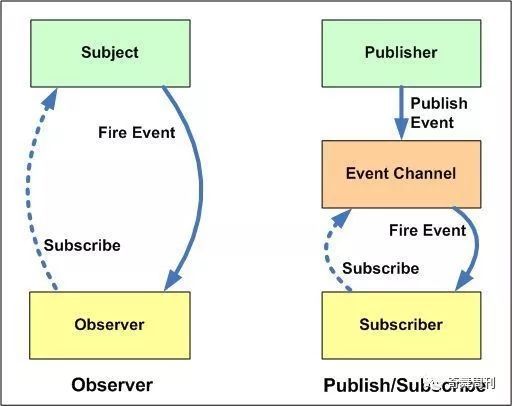
System.out.println(name + " 收到推送消息： " + message);

}

}

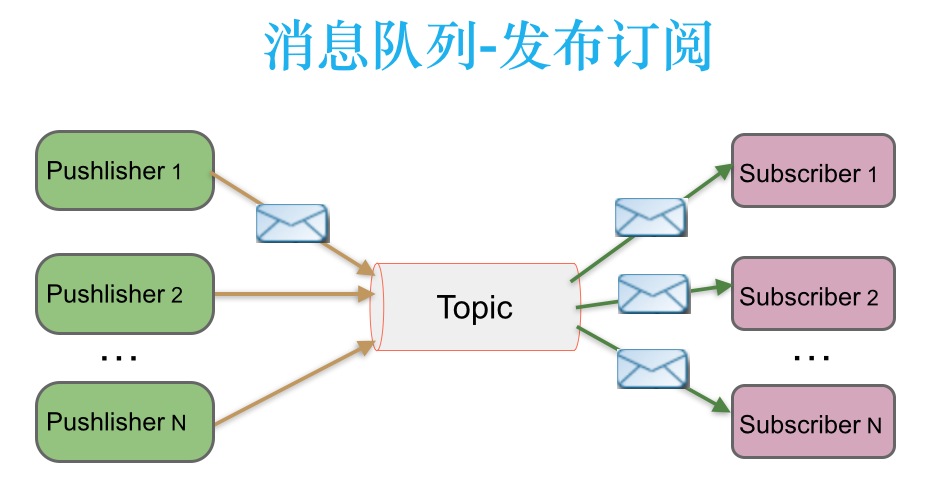
#### 1.1.2.2发布-订阅模式

那么发布-订阅模式和观察者模式有什么关联吗？发布-订阅模式是消息队列的一种消息模式，不属于23种设计模式，其类似于观察者模式，但也有不同之处



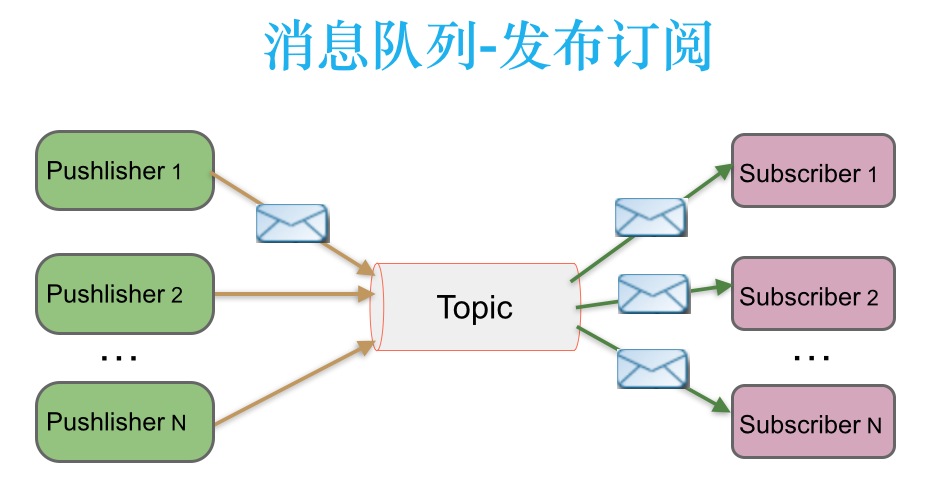
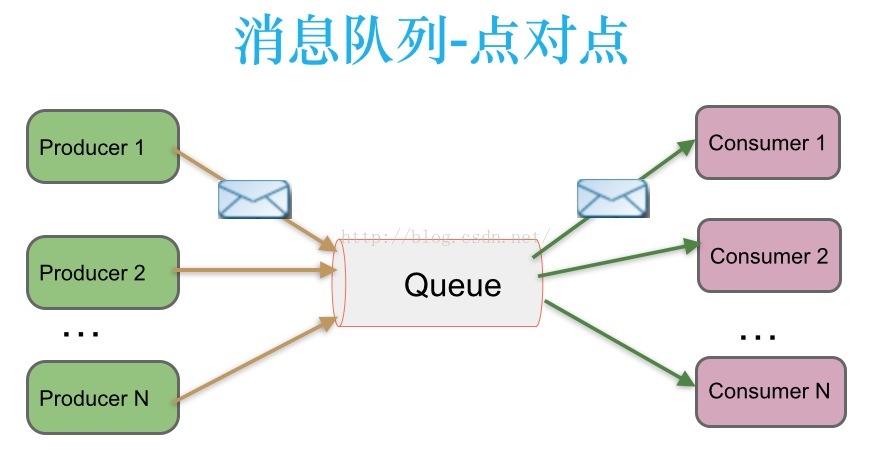
如上图所示，左侧是观察者模式，右侧是发布-订阅模式，那么他们最大的不同在哪呢？

1. 观察者模式中，发布者是知道订阅者有哪些的，然而，在发布-订阅模式中，发布者并不知道订阅者有哪些，它们只有通过消息代理进行通信。
2. 观察者模式大多数时候是同步执行的，比如当事件触发，Subject就会去遍历调用观察者的update方法(通知观察者)。而发布-订阅模式大多数时候是异步的（使用消息队列）。



传统的观察者模式并没有加入消息队列的机制，当有发布消息的需要时，被观察者会立即遍历通知所有观察者，并依次调用观察者的update方法，当这一切都完成后，才算发布完毕，这是极为耗时的操作，试想一下，如果观察者达到数万甚至数十万的量级，或者观察者的update方法耗时极长，那么每次的消息发布，我们都需要付出极大的代价。而发布-订阅模式加入了消息队列，发布者发送到topic的消息，只有订阅了topic的订阅者才会收到消息，并且所有订阅了topic的订阅者都会收到，这样一来，发布者只需要把要发布的消息发送到对应的topic队列中，发布就算完成了，订阅者会自行去topic中取出消息进行” 消费”。

### 1.1.3两种模式的比较



生产者发送一条消息到queue，一个queue可以有很多消费者，但是一个消息只能被一个消费者接受

当发布者发布一个消息，所有订阅这个topic的服务都能得到这个消息，所以从1到N个订阅者都能得到这个消息的拷贝。

### 1.1.4消费模式之推模式和拉模式

推模式(Push)： 即当生产者(Producer)发出的消息到达后，服务端马上将这条消息投递(push)给消费者(Consumer)；

拉模式(Pull)： 即服务端收到这条消息后什么也不做，只是等着消费者(Consumer)主动到自己这里来读，即消费者(Consumer)这里有一个“拉取”(Pull)的动作。

这两种模式都是消费端获取消息的方式，那么这两种方式有什么区别呢？这里根据几种场景进行分析：

**场景1：Producer 的速率大于 Consumer 的速率(生产>消费)**

对于 Producer 速率大于 Consumer 速率的情况，有两种可能性需要讨论，第一种是Producer 本身的效率就要比 Consumer 高(比如说：Consumer 端处理消息的业务逻辑可能很复杂，或者涉及到磁盘、网络等I/O操作)；另一种则是Consumer出现故障，导致短时间内无法消费或消费不畅。

Push模式：

在这种情况下，Push 方式由于无法得知当前 Consumer 的状态，所以只要有数据产生，便会不断地进行推送，在以上两种情况下时，可能会导致 Consumer 的负载进一步加重，甚至是崩溃。需要Consumer 有合适的反馈机制能够让服务端知道自己的状况。

Pull模式：

而采取 Pull 的方式问题就简单了许多，由于 Consumer 是主动到服务端拉取数据，当消费者跟不上生产者的效率时，只需要降低消费者的访问(pull)频率就好了。

**场景2：强调消息的实时性**

在一些强调实时性的场景中

Push模式：

采用 Push 的方式时，一旦消息到达，服务端即可马上将其推送给服务端，这种方式的实时性显然是非常好的

Pull模式：

采用 Pull 方式时，为了不给服务端造成压力（尤其是当数据量不足时，不停的轮询显得毫无意义），需要控制好自己轮询的间隔时间，但这必然会给实时性带来一定的影响。

**场景3：Pull 的长轮询**

Pull 模式有什么问题呢？由于主动权在消费方，消费方无法准确地决定何时去拉取最新的消息。如果一次 pull 取到消息了还可以继续去 pull，如果没有 pull 取到消息则需要等待一段时间再重新 pull。

但等待时间就很难判定了。你可能会说，我可以有xx动态拉取时间调整算法，但问题的本质在于，有没有消息到来这件事情决定权不在消费方。也许1分钟内连续来了1000条消息，然后半个小时没有新消息产生，可能你的算法算出下次最有可能到来的时间点是31分钟之后，或者60分钟之后，结果下条消息10分钟后到了，是不是很让人沮丧？

当然也不是说延迟就没有解决方案了，业界较成熟的做法是从短时间开始（不会对 CMQ broker 有太大负担），然后指数级增长等待。比如开始等5ms，然后10ms，然后20ms，然后40ms……直到有消息到来，然后再回到5ms。即使这样，依然存在延迟问题：假设40ms 到80ms 之间的50ms 消息到来，消息就延迟了30ms，而且对于半个小时来一次的消息，这些开销就是白白浪费的。

有一种优化的做法-长轮询，来平衡 pull/push 模型各自的缺点。基本方式是：消费者如果尝试拉取失败，不是直接 return，而是把连接挂在那里 wait，服务端如果有新的消息到来，把连接拉起，返回最新消息。

**场景4：部分或全部Consumer不在线**

在消息系统中，Producer 和 Consumer 是完全解耦的，Producer 发送消息时，并不要求Consumer 一定要在线，对于 Consumer 也是同样的道理，这也是消息通信区别于 RPC 通信的主要特点；但是对于 Consumer不在线的情况，却有很多值得讨论的场景。

首先，在 Consumer 偶然宕机或下线的情况下，Producer 的生产是可以不受影响的，Consumer 上线后，可以继续之前的消费，此时消息数据不会丢失；但是如果 Consumer 长期宕机或是由于机器故障无法再次启动，就会出现问题，即服务端需不需要为 Consumer 保留数据，以及保留多久的数据等等。

Push模式：

采用 Push 方式时，因为无法预知 Consumer 的宕机或下线是短暂的还是持久的，如果一直为该 Consumer 保留自宕机开始的所有历史消息，那么即便其他所有的 Consumer 都已经消费完成，数据也无法清理掉，随着时间的积累，队列的长度会越来越大，此时无论消息是暂存于内存还是持久化到磁盘上（采用 Push 模型的系统，一般都是将消息队列维护于内存中，以保证推送的性能和实时性），都将对服务端造成巨大压力，甚至可能影响到其他 Consumer 的正常消费，尤其当消息的生产速率非常快时更是如此；但是如果不保留数据，那么等该 Consumer 再次起来时，则要面对丢失数据的问题。

折中的方案是：给数据设定一个超时时间，当 Consumer 宕机时间超过这个阈值时，则清理数据；但这个时间阈值也并太容易确定。

Pull模式：

在采用 Pull 模型时，情况会有所改善；服务端不再关心 Consumer 的状态，而是采取“你来了我才服务”的方式，Consumer 是否能够及时消费数据，服务端不会做任何保证（也有超时清理时间）。

## 1.2消息队列的应用场景

### 1.2.1异步处理

用户注册后，需要发注册邮件(也可以是其他操作比如给用户发注册后的优惠券等)和注册短信。传统的做法有两种 1.串行的方式；2.并行方式：

1.串行方式：将注册信息写入[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "_blank)成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端



2.并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。与串行的差别是，并行的方式可以提高处理的时间



上述对比，串行处理需耗时150ms，并行需耗时100ms，但是这样的执行方式，效率仍然低下，下面引入消息队列



由图可知，响应时间降低到55ms，大幅提升了系统的吞吐量

### 1.2.2流量削峰

秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用加入消息队列。

用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面，秒杀业务根据消息队列中的请求信息，再做后续处理

### 1.2.3应用解耦

用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口



假如库存系统无法访问，则订单减库存将失败，从而导致订单失败，传统方式还会导致订单系统与库存系统耦合

加入消息队列后



 订单系统：用户下单后，订单系统完成持久化处理后，将消息写入消息队列，返回用户订单下单成功

 库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作

这样一来即使库存系统宕机，用户仍然可以进行下单操作，同时也实现订单系统与库存系统的应用解耦

当然这里只是举个不恰当的例子，上述处理方式很容易造成商品的超买超卖

### 1.2.4日志处理

在一些应用场景中，需要对日志进行大量采集和处理操作，比如能源管理系统，底层设备将本设备能源使用情况记录成日志，并将日志传输给上层处理，kafka更适用当前场景



日志采集客户端，负责日志数据采集，定时写入Kafka队列

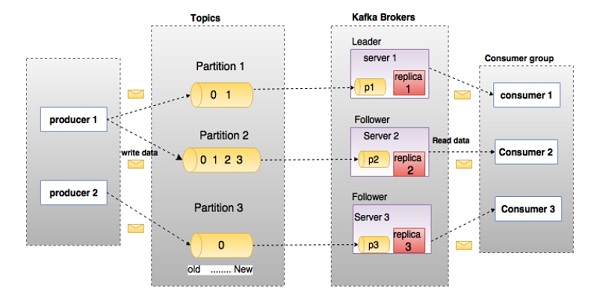
Kafka消息队列，负责日志数据的接收，存储和转发

日志处理应用：订阅并消费kafka队列中的日志数据

上述所列只是消息队列的部分应用场景，仅供参考

## 1.3kafka相关概念

Kafka专为分布式高吞吐量的系统设计，相比于其他消息队列，Kafka非常快，并保证零停机和零数据丢失。



**主题(Topics)：**同一类消息被归纳进一个topic中。生产者发送消息到指定的Topic下，消费者从这个Topic下消费消息。

**消息分区(Partition)：**一个Topic下面会分为很多分区，它是topic在物理上的分组。 每每个 partition 是一个有序的队列。partition 中的每条消息都会被分配一个有序的 id(offset)。

**复制因子(replication-factor)：**为了提高消息的可靠性，Kafka每个topic的partition有N个副本（replicas），其中N(大于等于1)是topic的复制因子的个数，实际上就是为partition中的消息指定副本个数。

Partition（分区）

主题可能有许多分区，因此它可以处理任意数量的数据。

3

Partition offset（分区偏移）

每个分区消息具有称为" offset "的唯一序列标识。

4

Replicas of partition（分区备份）

副本只是一个分区的"备份"。 副本从不读取或写入数据。 它们用于防止数据丢失。

5

Brokers（经纪人）

代理是负责维护发布数据的简单系统。 每个代理中的每个主题可以具有零个或多个分区。 假设，如果在一个主题和N个代理中有N个分区，每个代理将有一个分区。

假设在一个主题中有N个分区并且多于N个代理(n + m)，则第一个N代理将具有一个分区，并且下一个M代理将不具有用于该特定主题的任何分区。

假设在一个主题中有N个分区并且小于N个代理(n-m)，每个代理将在它们之间具有一个或多个分区共享。 由于代理之间的负载分布不相等，不推荐使用此方案。

6

Kafka Cluster（Kafka集群）

Kafka有多个代理被称为Kafka集群。 可以扩展Kafka集群，无需停机。 这些集群用于管理消息数据的持久性和复制。

7

Producers（生产者）

生产者是发送给一个或多个Kafka主题的消息的发布者。 生产者向Kafka经纪人发送数据。 每当生产者将消息发布给代理时，代理只需将消息附加到最后一个段文件。 实际上，该消息将被附加到分区。 生产者还可以向他们选择的分区发送消息。

8

Consumers（消费者）

Consumers从经纪人处读取数据。 消费者订阅一个或多个主题，并通过从代理中提取数据来使用已发布的消息。

Leader（领导者）

" Leader "是负责给定分区的所有读取和写入的节点。 每个分区都有一个服务器充当Leader

。

10

Follower（追随者）

跟随领导者指令的节点被称为Follower。 如果领导失败，一个追随者将自动成为新的领导者。 跟随者作为正常消费者，拉取消息并更新其自己的数据存储。