# Redis、MQ、Kafka作为消息队列对比

消息队列

优点：

1. 解耦，将消息写入到消息队列，需要的系统来订阅
2. 异步，加快速度
3. 消峰， 防止一段时间大量并发访问数据库，造成数据库连接异常；通过消息队列，消费者可以根据数据库的并发量去拉取消息

缺点：

系统可用性降低、复杂性增加

Rocket等mq中间件都属于很成熟的产品，性能一般但可靠性较强，而kafka原本设计的初衷是日志统计分析，现在基于大数据的背景下也可以做运营数据的分析统计，而redis的主要场景是内存数据库，作为消息队列来说可靠性太差，而且速度太依赖网络IO，在服务器本机上的速度较快，且容易出现数据堆积的问题，在比较轻量的场合下能够适用。

MQ：

性能一般但比较可靠

Kafka:

设计初衷是日志统计分析

Redis:

内存数据库

多用于实时性较高的消息推送，并不保证可靠

不支持分组（kafka的Group功能）

作为消息队列可靠性差，速度过于依赖网络IO，容器出现数据堆积

适合轻量场合

|  |  |  |  |
| --- | --- | --- | --- |
|  | RabbitMQ | Kafka | Redis |
| 开发语言 | Java | Scala | ANSI C |
| 支持协议 | AMQP | TCP |  |
| 消息存储 | 支持大量数据堆积 | 支持少量堆积 |  |
| 消息事务 | 支持 | 支持 |  |
| 负载均衡 | 基于zookeepr，做的非常好 | 支持不好 | 不支持，需要自定义 |
| 可用性 | 非常高（分布式） | 高（主从） |  |
| 吞吐量TPS | 极大 | 较大 |  |
| 消息重试 | 不支持 | 不支持 |  |

# 二、ACK机制

# 三、CAP

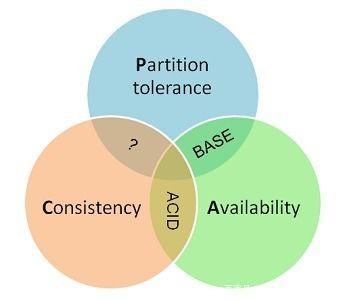
## 1. 概念

C-Consistency 一致性

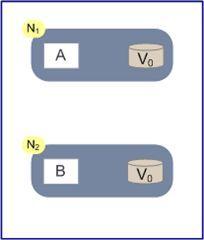
A-Availability 可用性

P-Patition Tolerance 分区容忍性

**CAP理论是指一个分布式系统中最多满足C、A、P中的两个**。



## 2. 举例说明



网络中有两个节点N1、N2，A和B是分布式系统的两个部分，V是子数据库。

现在user1和user2分别同时访问N1和N2节点。正常步骤：

* user1访问N1，修改V将V0改成V1
* N1此时发现自己的数据有变化，向N2发起同步数据的操作
* user2访问N2，等待N2同步数据成功，此时访问到的数据就是V1了，也就是正确的数据。

可以看出，理想情况下满足了CAP三项

## 3. CAP理解

### 3.1 Consistency

一致性指的是所有节点在同一时间的数据完全一致。就如user1和user2一样读取的数据都是一样的。

从客户端看：多路并发读取数据一致

从服务器看：一节点改变数据，在同步数据期间，所有对该数据的访问全部阻塞。

### 3.2 Avaibility

可用性是指服务一直可用，并且响应时间正常。不会出现用户访问不了，后者响应超时的情况

### 3.3 Partition Torelance

一个分布式系统里面，节点组成的网络本来应该是连通的。然而可能因为一些故障，使得有些节点之间不连通了，整个网络就分成了几块区域。数据就散布在了这些不连通的区域中。这就叫分区。

当数据只在一个节点保存时，该节点一出现故障，其他节点将无法访问此数据，这时候分区就是无法容忍的。

提高分区容忍性的方法就是把数据复制到多个节点，分区出现后，数据也分布到各个区了，这样就可以访问数据。

**数据存在的节点越多，分区容错性就越高，但节点数据之间的一致性就很难保证。为了一致性，同步节点之间的数据的时间就越长，可用性就会降低。**

# 四、Zookeeper

## 1. docker安装Zookeeper

docker logs -f my\_zookeeper #验证是否成功安装

## 2. 常用命令

2.1、首先使用命令进入服务器: docker exec -it zk3 bash

2.2、使用命令 ./bin/zkServer.sh status 来查看节点的状态

2.3、使用zkCli.sh开启客户端

2.4、使用create -e /node1 node1.1 创建临时节点，当客户端关闭时候，该节点会随之删除。不加参数－e创建永久节点。

 get /node:获取节点值

 ls /node：列出节点

 delete /node 删除节点

 stat：查看节点信息

setAct path acl：用于设置节点访问权限

getAcl path ：查看节点的权限信息

## 3. 简介

致力于开发和维护开源服务器，是一个高度可靠的分布式服务协调组件。

**CP（一致性+分区容忍性）**

基于观察者模式设计的分布式服务管理框架。负责存储和管理数据，然后接受观察者的注册，一旦数据状态发生变化，zookeeper将通知注册的观察者做出相应的反应，从而实现集群中类似的Maset/Slaver管理模式

**文件系统（基于层次型的目录树的数据结构）+通知机制**

应用场景：

* 集群统一配置管理
* 集群统一命名服务：命名服务是通过对资源命名，然后通过命名去定位资源
* 集群统一管理
* 服务上下线动态感知
* 负载均衡
* 分布式消息同步和协调机制
* 对Dubbo的支持

## 4. 内部原理

### 4.1 选举机制

**Server ID： myid(权重越大)**  
**Zxid：数据ID(先一数据低进行选择)**

**半数机制（Paxos 协议）：**

集群中半数以上机器存活，集群可用。所以zookeeper适合装在奇数台机器上。

Leader是通过内部的选举机制临时产生的。

以一个简单的例子来说明整个选举的过程。

假设有五台服务器组成的zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么。

(1)服务器1启动，此时只有它一台服务器启动了，它发出去的报没有任何响应，所以它的选举状态一直是LOOKING状态。

(2)服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3)，所以服务器1、2还是继续保持LOOKING状态。

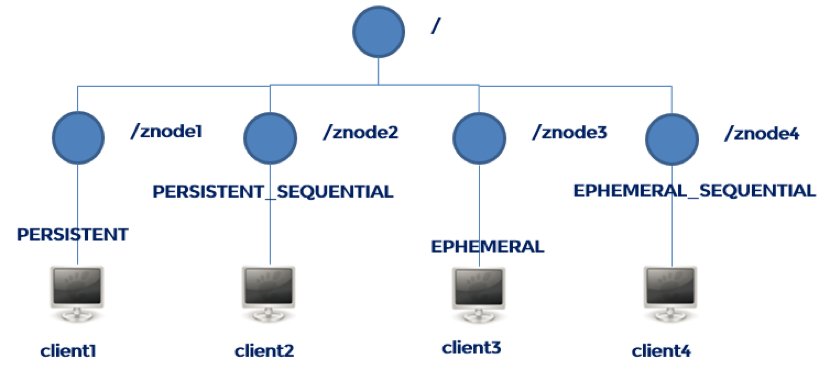
(3)服务器3启动，根据前面的理论分析，服务器3成为服务器1、2、3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的leader。

(4)服务器4启动，根据前面的分析，理论上服务器4应该是服务器1、2、3、4中最大的，但是由于前面已经有半数以上的服务器选举了服务器3，所以它只能接收当小弟的命了。

(5)服务器5启动，同4一样当小弟。

### 4.2 Znode节点类型

Znode = path + nodeValue + Stat



### 4.3 zoo.cfg

(1) tickTime 通信心跳时间 session超时时间是两倍tickTime

(2) initLimit LF初始通信时限

Leader与Follower建立连接的时间

(3) syncLimit LF同步通信时限

Leader与Follower之间的最大响应时间

(4) dataDir 数据存储目录

(5) clientPort 客户端连接zookeeper端口，默认是2181

### 4.4 Stat结构体



### 4.5 监听器原理

一次触发：数据发生改变后，watch只会监听一次，当数据再次改变时，需要创建新的watch去监听此次改变事件

数据观察和子节点观察

(1) 首先要有一个main()线程

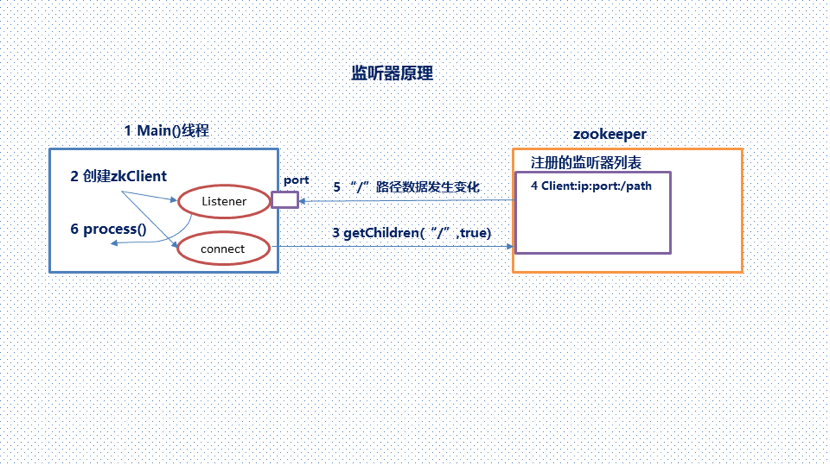
(2) 在main线程中创建ZK客户端，这是会创建两个线程，一个负责网络连接通信(connect),一个负责监听(listener)

(3) 通过connect线程将注册的监听事件发送给ZK

(4) 在ZK的注册监听器列表中将注册的监听事件添加到列表中

(5) ZK监听到有数据或路径发生变化时，就会将这个消息发送给listener线程

(6) Listener线程内部调用process()方法



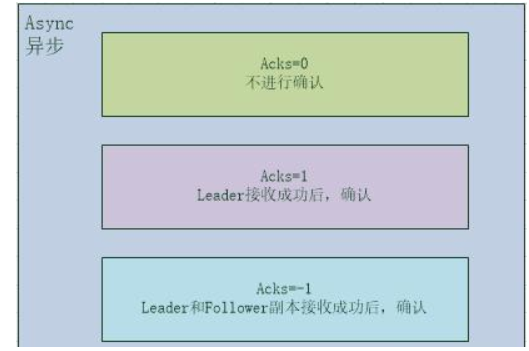
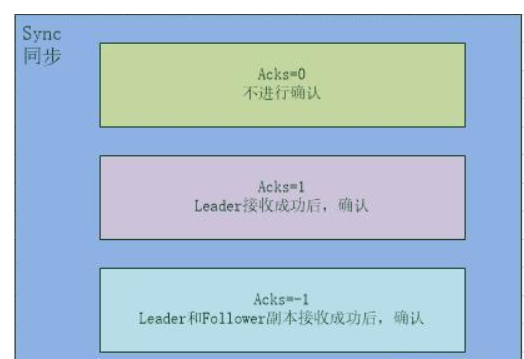
# 五、Kafka

## 1. kafka消息丢失

**消息发送方式**：

product.type sync（默认同步）、async（异步）

request.requires.acks 消息确认



**丢失场景：**

1. 网络异常

当网络异常时，ack=0，客户端无法感知消息发送成功与否

1. 客户端异常

Async异步发送模式下，客户端发送消息并没有直接发给kafka，而是在Client端按一定规则缓存并批量发送，在此期间，如果客户端死机，消息丢失

1. 缓冲区满

异步模式下，Client端缓存大小超出缓冲池的大小，也可能发生消息丢失

1. Leader副本异常

ack = 1，leader副本接收成功，返回确认信息给客户端，此时Follower副本还在同步，leader副本异常，同步失败，主从数据不一致就会出现消息丢失的情况。

## 2. kakfa重复消费数据

终极原因就是消费端消费了数据，但offset没有提交成功。

原因：

1. 消费端宕机，offset没有提交
2. 设置offset为自动提交，关闭kafka时，如果在close之前，调用 consumer.unsubscribe() 则有可能部分offset没提交，下次重启会重复消费
3. 重新分配分区，触发rebalance，可能出现从头开始消费，导致重发问题

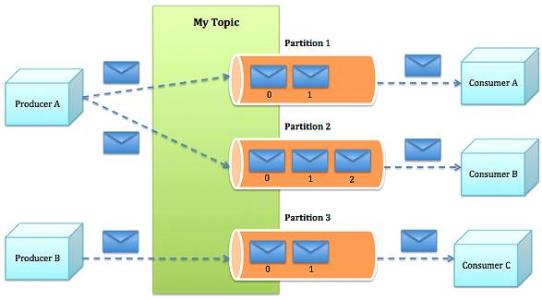
如：一次消费数据处理时间太长，kafka session timeout，partition断开连接，重新分区。

org.apache.kafka.clients.consumer.CommitFailedException: Commit cannot be completed since the group has already rebalanced and assigned the partitions to another member. This means that the time between subsequent calls to poll() was longer than the configured max.poll.interval.ms, which typically implies that the poll loop is spending too much time message processing. You can address this either by increasing the session timeout or by reducing the maximum size of batches returned in poll() with max.poll.records

poll间隔时间：max.poll.interval.ms (默认300s)

消费者在处理完一批poll数据时，提交offset报的错：由于处理该轮数据耗时太长，超过了300s，与session一样，kafka认为消费端挂了，断开连接，提交offset异常。

## 3. kafka分区分配策略



### 3.1 前言

kafka一个topic有多个分区，一个分区对应着同一组下的一个消费者，所以对于同一组下的不同消费者只有一个能接收到同一个主题的数据。

举例：

P0、P1、P2 P0-C0、P1-C1、P2-C2

C0、C1、C2

C0、C1 P0/P1-C0、P2-C2

C0、C1、C2、C3 P0-C0、P1-C1、P2-C2

由此可见：**同一个消费组下的消费者数量不能大于分区数**，否则将有消费者接收不到数据，造成资源浪费。

### 3.2 多个partition的优点

* 对broke数据分片，通过减少消息容量来提升IO性能
* 提高消费端的消费能力，多个消费者消费同一个topic下的不同partition，实现消费端的负载均衡

### 3.3 策略

RangeAssignor范围分区、RoundRobinAssignor轮询分区

自定义分区策略

## 4. Rebalance

Rebalance: 即对kafka中分区进行重新分配的过程

触发场景：

1. 同一个消费组下新增消费者
2. 消费者宕机、主动取消订阅（kafka与消费者断开连接）
3. 新增分区

操作过程：

Kafka提供了一个Coordinator来管理消费端group

1. 确定Coordinator：

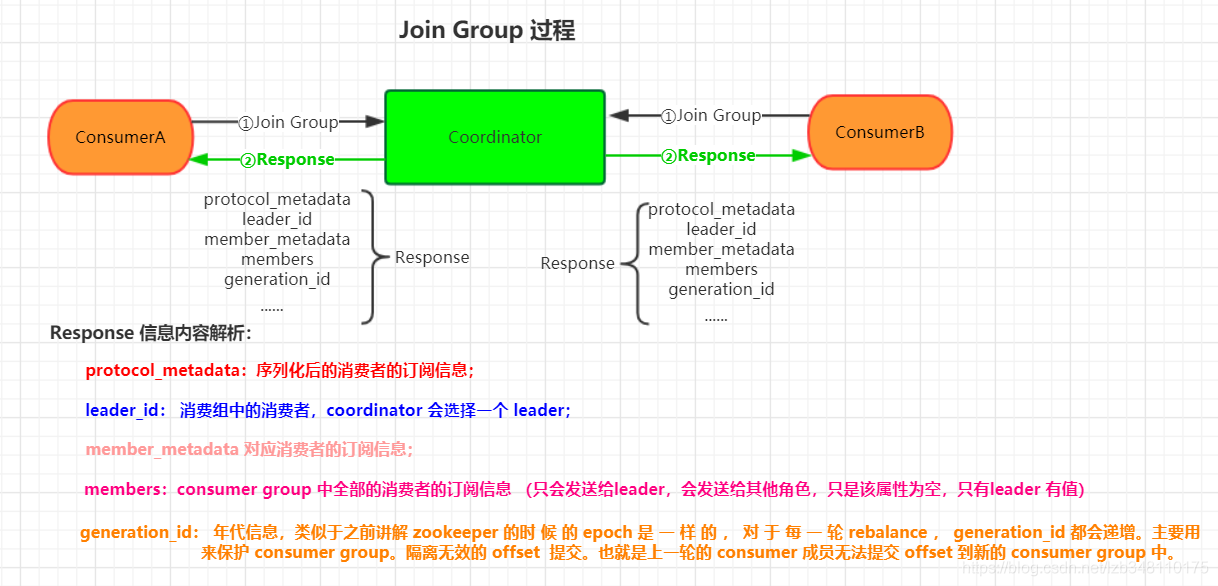
第一个启动的Consumer向kafka发送GroupCoordinateRequest请求，kafka在broker集群中选取一个负载最小的broke作为Coordinator，并发broker.id返回

1. JoinGroup

确定了Coordinator，所有的消费者都会向Coordinator发送一个JoinGroup请求（只要启动就会发送），Coordinator在Group中选取一个leader，并返回组员信息和订阅信息。

根据下图，可以看出，只有leader才有消费者的订阅信息。

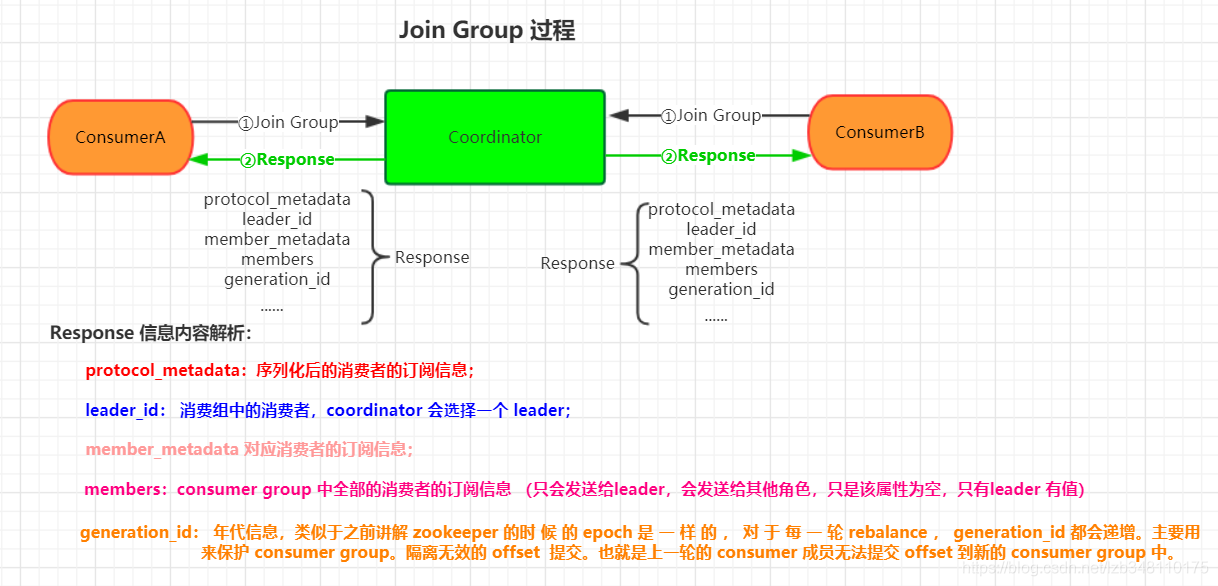
leader根据分区分配策略决定分区情况



1. Synchronizing Group State 阶段

该阶段主要完成将leader的分配方案同步给Group下的所有Consumer

完成分区分配后，所有Consumer都会发送SyncGroup请求，只有leader会发送分配方案，其他Consumer也会发送只不过是空值，Coordinator向所有Consumer返回分配方案。这样所有的成员都会知道自己消费的分区。



根据上面的流程，可以发现：

**kafka的分区分配方案是Consumer实现的，这样更具有灵活性。**