# 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

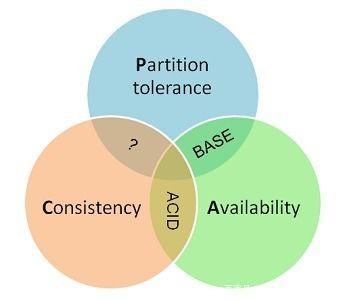
## 1. 概念

C-Consistency 一致性

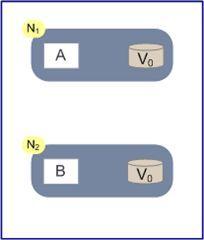
A-Availability 可用性

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。**

# 七、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：

其实都是将对应的数据（如配置、客户端ip信息）挂载在一个指定的Znode上，通过监听Znode子节点的变化（或数据、或节点）去动态实时响应。

* 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

如：服务上下线动态感知，节点运行状态

临时节点+监听机制

* 负载均衡
* 分布式消息同步和协调机制
* 对Dubbo的支持
* [分布式锁](#_3.zookeeper实现)

## 4. 内部原理

### 4.1三种角色

* Leader
* Follower
* Observer

可以理解为不参与投票的Follower，协助Follower处理读请求。当集群中读请求负载很高时，为什么不增加Follower节点，原因是写请求需要Follower节点超过半数同意，会增加Leader和Follower的通信压力，降低写效率。

### 4.2 两种模式

* 恢复模式

当服务启动或Leader崩溃后，zk进入恢复状态，选举leader，选举完成后将leader与其他节点数据同步，当大多数Follower与leader同步完成后，恢复模式完成。

* 广播模式

在恢复模式完成后，客户端发起写请求，leader采用ZAB协议广播该写请求，超过半数的follower同意后提交该事务，完成本次请求。

follower要么ack，要么放弃，leader无需等待所有的Follower应答。

### 4.3 Zxid

zxid Long(64) 纪元epoch（高32位）+ xid（低32位）

epoch：每个leader都有自己的纪元

xid：依次递增的事务id

使用方式：

在广播模式中，leader会生成一个zxid和写请求一并发送给Follower，follower本地也有自己的zxid，如果leader的zxid > follower，follower将leader.zxid写入本地日志中，返回ack应答，否则拒绝响应。

### 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（**[**ZAB**](#_六、ZAB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myid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。

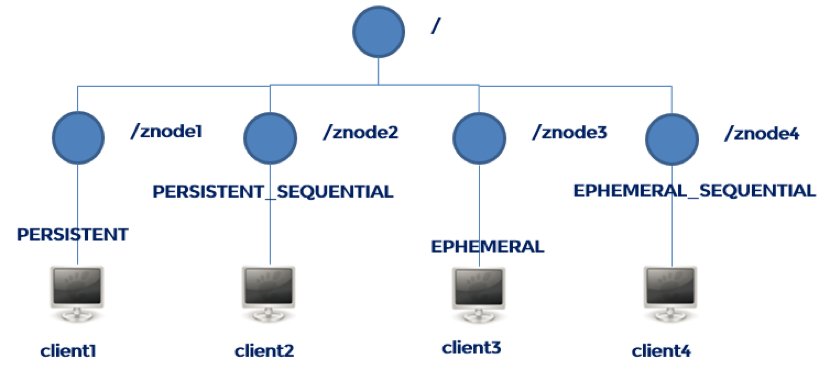
(3)服务器3启动，根据前面的理论分析，服务器3成为服务器1、2、3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的leader。

(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

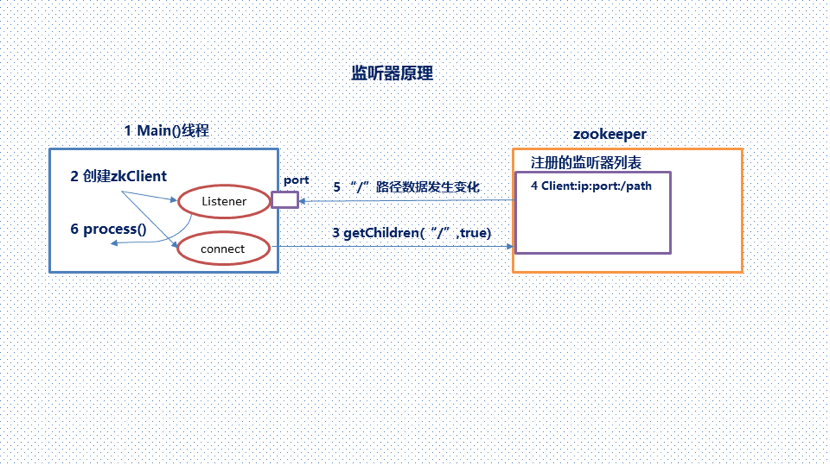
(2) 在main线程中创建ZK客户端，这是会创建两个线程，一个负责网络连接通信(connect),一个负责监听(listener)

(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



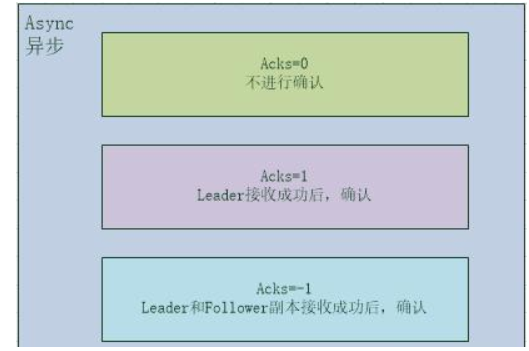
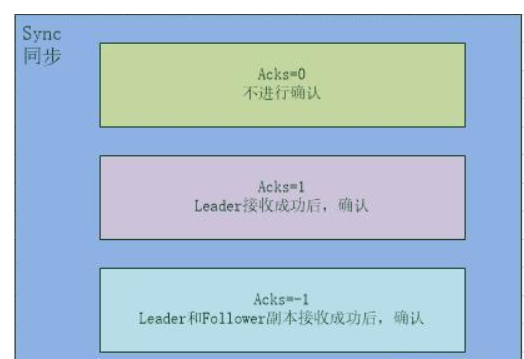
# 八、Kafka

## 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

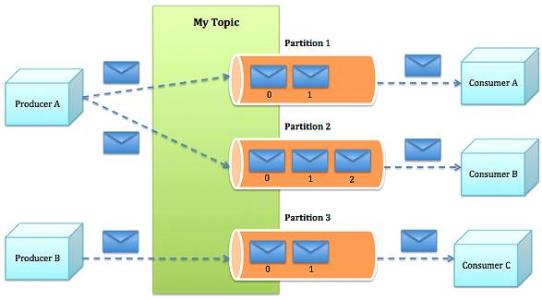
如：一次消费数据处理时间太长，kafka session timeout，partition断开连接，重新分区。

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：

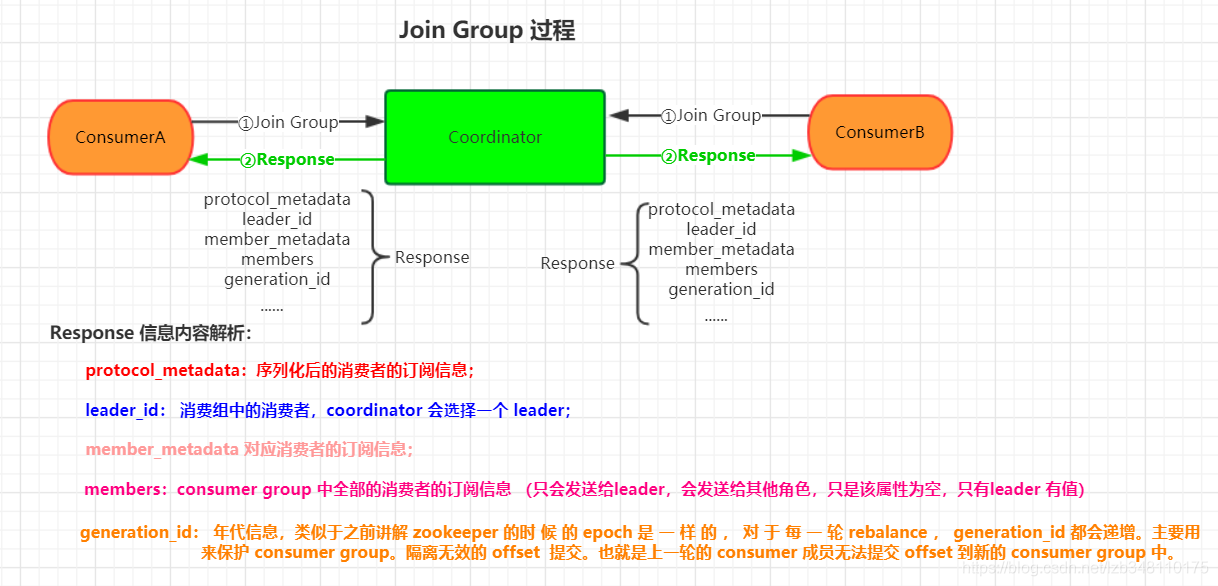
第一个启动的Consumer向kafka发送GroupCoordinateRequest请求，kafka在broker集群中选取一个负载最小的broke作为Coordinator，并发broker.id返回

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。

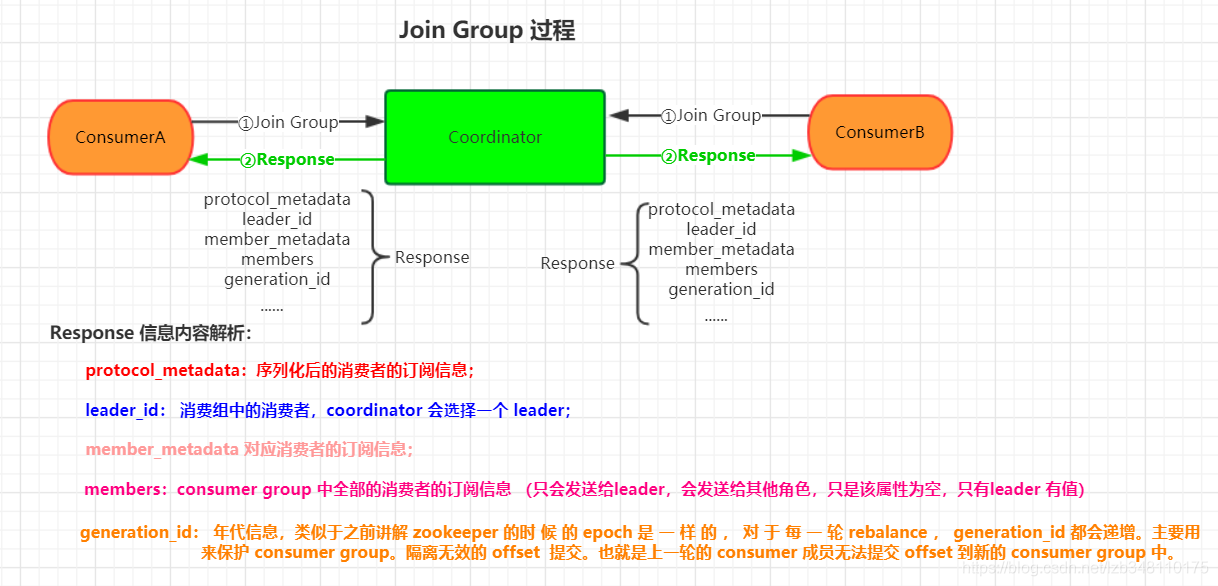
leader根据分区分配策略决定分区情况



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：

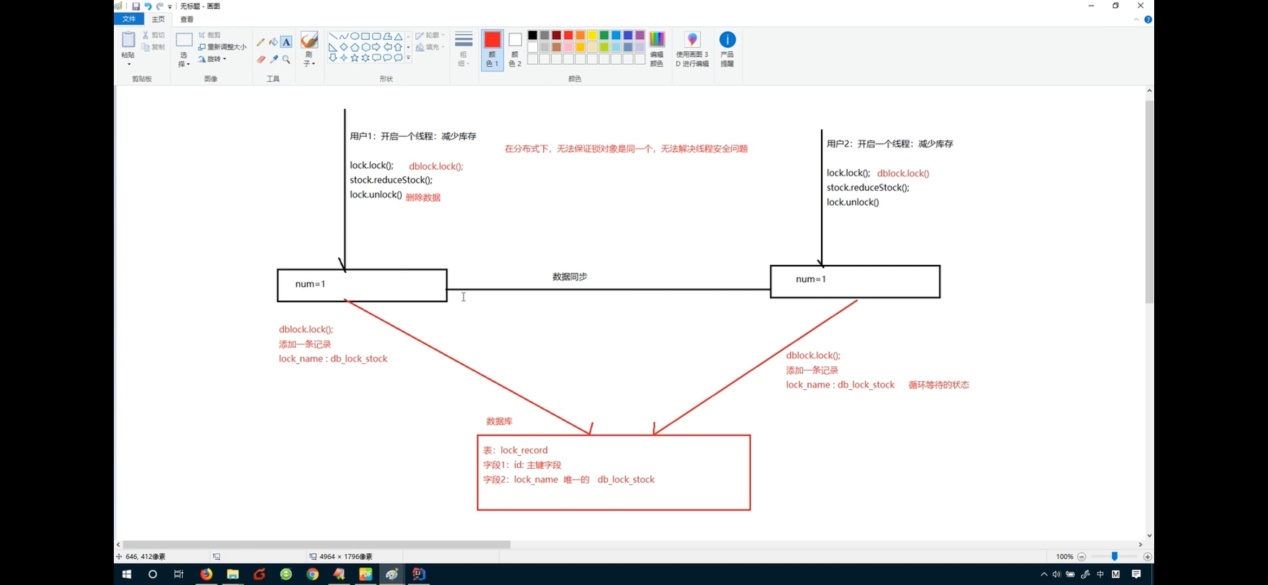
**kafka的分区分配方案是Consumer实现的，这样更具有灵活性。**

# 九、分布式锁

平常我们为了多线程安全问题，一般是在对共享变量加锁解决。但在分布式系统中，加锁并不能解决线程安全问题，因为分布式中，锁住的并不是同一个对象。

同一时刻请求所有节点上的同一数据应该是一致的。

## 1. 数据库实现



dblock是自身实现的一个锁，既能解决同一个节点的线程安全问题，同时在dblock.lock()时，往数据库插入一个数据。

用户1此时获取了该锁，往数据库插入了该操作的key，同时，数据库2也想获取该锁，发现数据已经存在了key，就循环查询等待，用户1完成操作后unlock()释放锁的同时将数据库中的key行删除，用户2就获取锁。

## 2. redis实现

redis的setnx

redission的lock

## 3. zookeeper实现

有序临时节点 + 监听机制

3个请求同时发送到3个服务器上操作同一共享资源，  
A请求 ----- server1 B请求-----server2 C请求----server3

每个请求都会在zookeeper下的lock节点下生成一个临时有序节点：

A：001 B:002 C:003

在操作资源之前，

1.先检查自身id是否是lock子节点中最小值，

2.如果是，则获取锁，进行操作，操作完成后释放锁（即删除自身节点）。

3.如果不是，添加001节点的监听事件，等待001删除事件。

# 十、RPC

## 1.RPC

RPC（remote process call）远程过程调用

一般基于TCP协议，也可使用HTTP2.0协议，请求体更小，效率更高

基于API层面实现，代码实现复杂，使用动态代理

可使用XX序列化，效率高

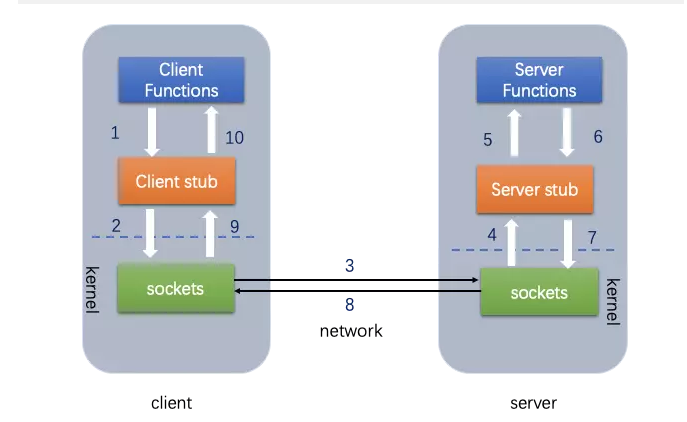
调用过程：

1.客户端发起一个请求，请求包括方法id+请求参数

2.客户端将1中的数据包序列化后传输给服务器

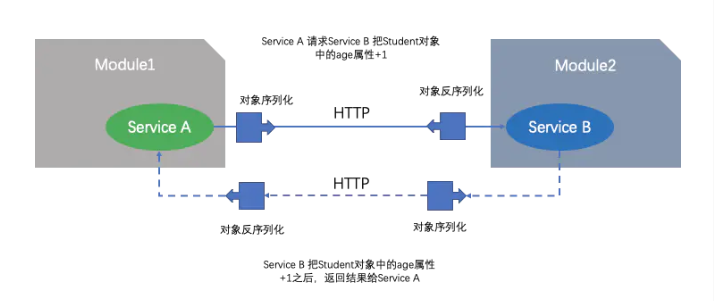
3.服务器管理着方法映射集合Map,服务端接收到数据后反序列得到id和参数，在map中根据id找到对应的方法，在本地调用该方法后返回给客户端。

4.客户端将接收的数据反序列化得到结果。



## 2.Http

http其实是一种网络传输协议，基于TCP，规定了网络传输的请求格式、响应格式、资源定位和操作的方式等。现在客户端浏览器与服务端通信基本都是采用Http协议。也可以用来进行远程服务调用。缺点是消息封装臃肿



## 3.异同

### 3.1 相同点

底层通讯都是基于socket，都可以实现远程调用，都可以实现服务调用服务

### 3.2 不同点

框架：

RPC：dubbo、cxf、（RMI远程方法调用）Hessian

http：框架有：httpClient

底层：

RPC： ①没有规定数据传输格式；②没有资源定位的路径；

③需要向本地调用服务一样调用远程服务，因此需要在api层面封装

HTTP：①规定了数据传输格式；②定义了资源定位的路径；

③请求和响应自己实现

关于③的理解：

服务器上有一个方法say(), 使用rpc，用户直接调用该方法即可，直接获取值，就和使用本地方法一样；但http就需要我们自己去封装请求头，请求方法，参数值等等信息，然后去处理返回的流信息。其实，rpc使用动态代理，在代理类内部使用socket去发送网络请求。

优缺点：

RPC:

缺点：

* 服务提供方和服务消费方必须使用统一的RPC框架
* 在API层面实现，限制了开发的语言环境

优势：

* 调用快、处理快
* 跨操作系统在同一编程语言内使用
* 更加透明方便

HTTP:

缺点：

* 消息臃肿，传输速度较慢
* 需要自己实现请求和响应的具体细节

优势：

* 跨系统跨编程语言的远程调用框架。http无需关注语言的实现，只需要遵循rest规范
* 通用性强，更灵活

### 3.3 如何选择

**速度**来看，RPC要比http更快，虽然底层都是TCP，但是http协议的信息往往比较臃肿

**难度**来看，RPC实现较为复杂，http相对比较简单

**灵活性**来看，http更胜一筹，因为它不关心实现细节，跨平台、跨语言。

如果对**效率**要求更高，并且开发过程使用统一的技术栈，那么用RPC还是不错的。

如果需要更加灵活，跨语言、跨平台，显然http更合适

## 4 与Feign的关系

Feign：伪RPC？

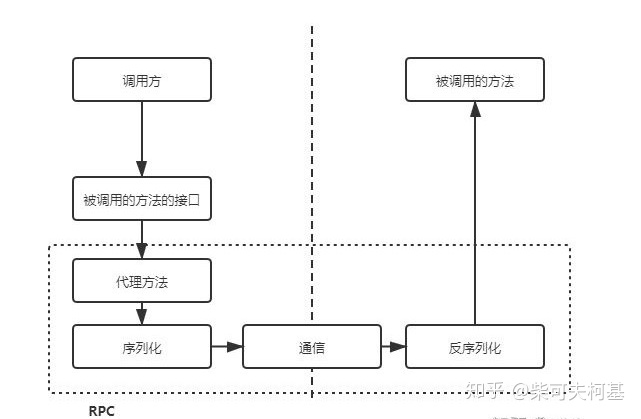
Feign是基于Http协议的远程过程调用。

实际上RPC也就是一种编程模型，初衷就是你可以不在乎底层的网络技术协议而实现远程调用。

判断是否为RPC：

**你像调本地接口一样调用远程接口的方式，就是RPC**

下图中虚线的框里是你不用去实现，RPC框架已经帮你做了，所以调用起来就像是调用本地接口一样。





# 十一、Dubbo

Dubbo具有调度、发现、监控、治理等功能，**支持相当丰富的服务治理能力**。

## 1、为什么需要dubbo

随着流量的增大，常规的垂直应用架构已无法应对。

架构演变过程：

* 单一应用架构
* 应用和数据库单独部署
* 应用和数据集群部署
* 数据库压力变大，读写分离
* 使用缓存技术加快速度
* 数据库分库分表
* 应用根据业务拆分成不同模块

问题也随之出现：

①应用之间的URL配置难以管理，单点负载均衡服务器压力变大

②进一步发展，服务依赖关系变得错综复杂，哪个服务要在哪个服务前启动，难以描述应用的架构关系

③接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？



## 2、技术架构





可选用Zookeeper和Redis作为注册中心

整个调用过程：

①服务器启动，注册服务

②消费端启动，订阅服务

考虑失败或变更：

注册中心返回服务列表给消费端，如果注册中心有改动，基于长连接将变更实时推送给消费端

消费端拿到服务列表，基于负载均衡算法访问其中某个节点，如果访问失败，则访问下一个节点

服务器和消费端在内存中统计调用次数和调用时间，定时每分钟推给监控中心

## 3、常用场景

### 3.1 启动时检查

dubbo缺省在启动时检查依赖的服务是否可用，不可用则抛出异常，阻止Spring的初始化

### 3.2 集群容错

|  |  |
| --- | --- |
| 集群模式 | 说明 |
| Failover | 失败自动切换（默认）。通常用于读操作，但重试会带来延迟，可设置retries=2来设置重试次数 |
| Failfast | 快速失败。失败即报错。通常用于非幂等性的写操作，如新增 |
| Failsafe | 失败安全。出现失败，直接忽略 |
| Failback | 失败自动重发。后台记录失败请求，定时重发，通常用于消息通知操作 |
| Forking | 并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数 |
| BroadCase | 广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息 |

### 3.3 负载均衡



### 3.4 其他

直连

只注册

只订阅

多协议机制

多注册中心

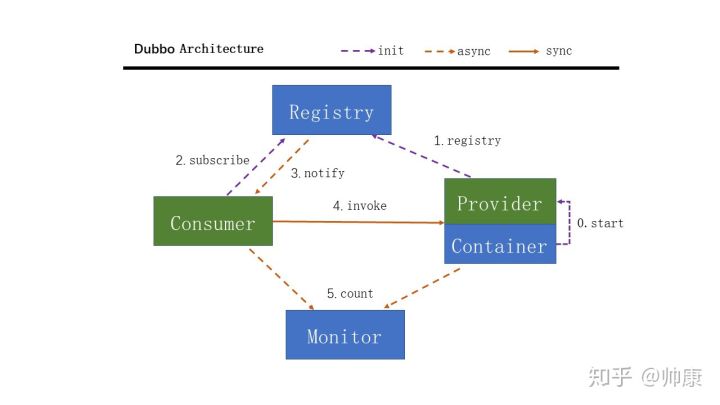
多版本

日志管理

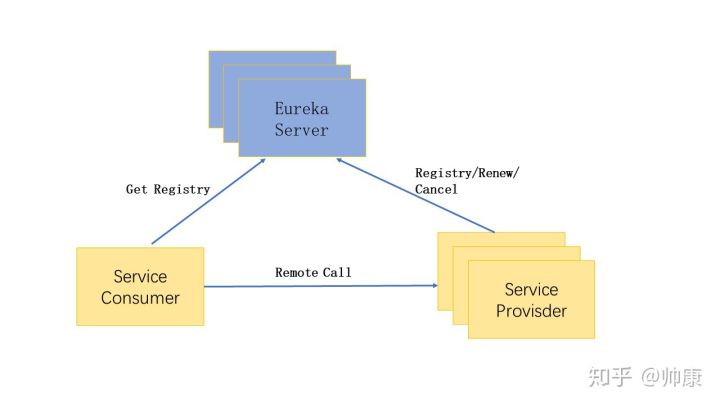
## 4、与SpringCloud比较

微服务的核心要素在于服务的发现、注册、路由、熔断、降级、分布式配置。基于这几点对Dubbo和SpringCloud比较。

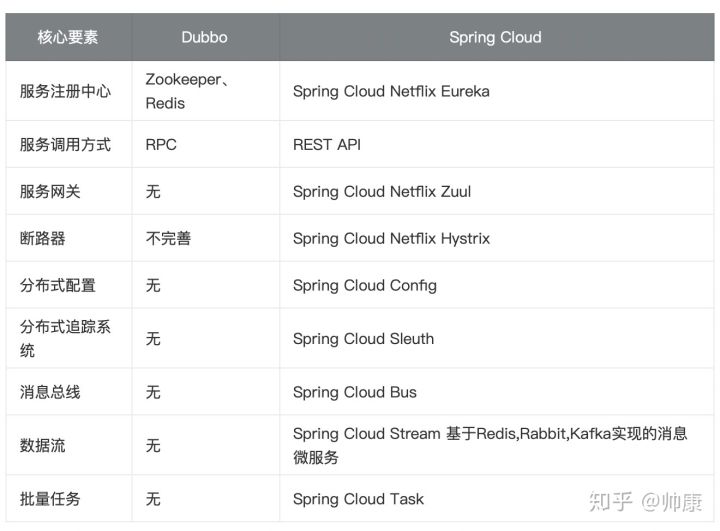
### 4.1 整体架构



SpringCloud:



### 4.2 核心要素



Dubbo 只是实现了服务治理，而 Spring Cloud 子项目分别覆盖了微服务架构下的众多部件，服务治理只是其中的一个方面。

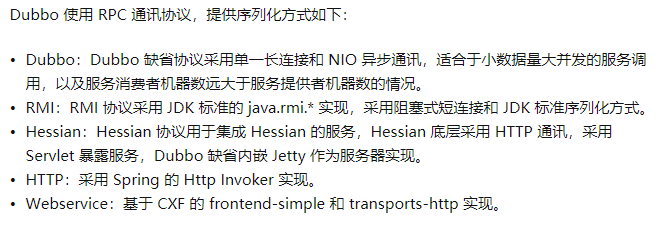
Dubbo 提供了各种 Filter，对于上述中“无”的要素，可以通过扩展 Filter 来完善。例如：

* 分布式配置：可以使用淘宝的 diamond、百度的 disconf 来实现分布式配置管理。
* 服务跟踪：可以使用京东开源的 Hydra，或者扩展 Filter 用 Zippin 来做服务跟踪。
* 批量任务：可以使用当当开源的 Elastic-Job、tbschedule。

点评：从核心要素来看，Spring Cloud 更胜一筹，在开发过程中只要整合 Spring Cloud 的子项目就可以顺利的完成各种组件的融合，而 Dubbo 却需要通过实现各种 Filter 来做定制，开发成本以及技术难度略高。

### 4.3 通讯协议与性能比较

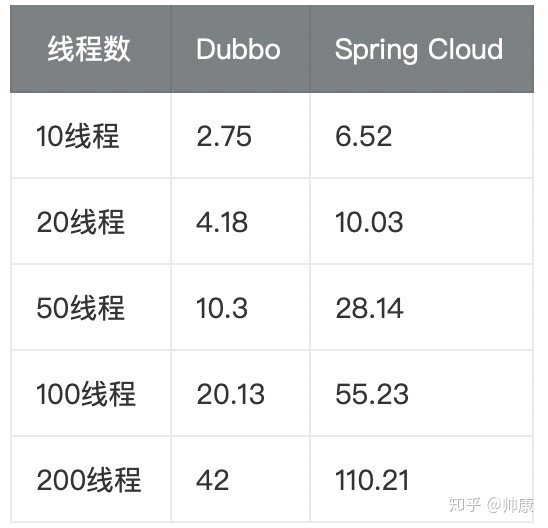
**【通讯协议】**



SpringCloud采用Http协议的REST API

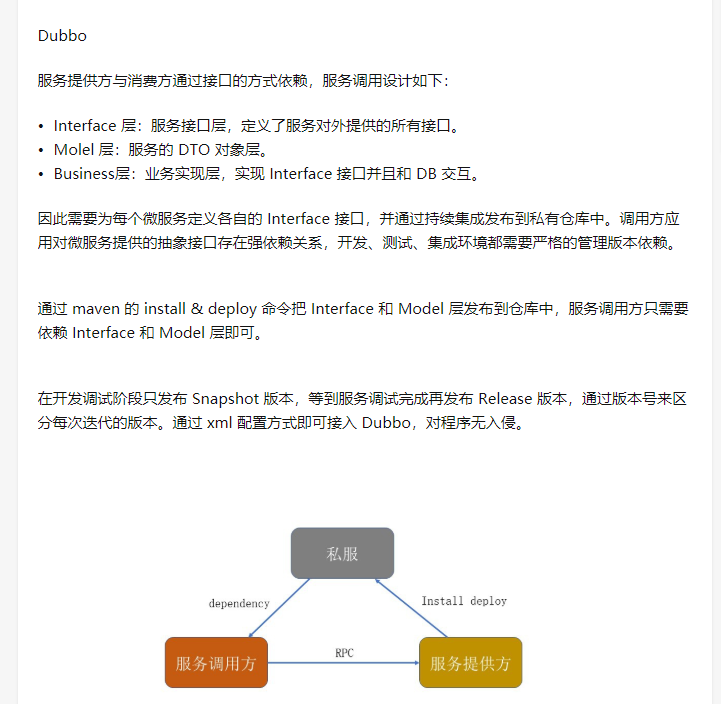
**【性能比较】**

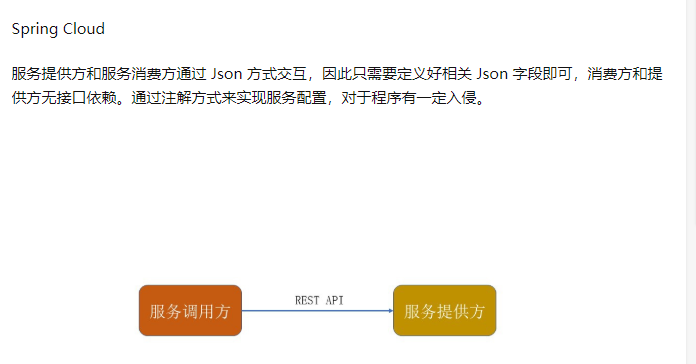
使用一个 Pojo 对象包含 10 个属性，请求 10 万次，Dubbo 和 Spring Cloud 在不同的线程数量下，每次请求耗时（ms）如下：



Dubbo 支持各种通信协议，而且消费方和服务方使用长链接方式交互，通信速度上略胜 Spring Cloud，如果对于系统的响应时间有严格要求，长链接更合适

### 4.4 服务依赖方式





Dubbo服务依赖略重，需要有完善的版本管理机制，但代码入侵少。

Spring Cloud 通过 Json 交互，省略了版本管理的问题，但是具体字段含义需要统一管理，自身 Rest API 方式交互，为跨平台调用奠定了基础。

### 4.4 总结

Dubbo 出生于阿里系，只需要通过 Spring 配置的方式即可完成服务化，对于应用无入侵，设计的目的还是**服务于自身的业务**为主。

如果我们使用配置中心、分布式跟踪这些内容都需要自己去集成，这样无形中增加了使用 Dubbo 的难度。

Spring Cloud 是大名鼎鼎的 Spring 家族的产品， 专注于企业级开源框架的研发。

Spring Cloud 自从发布到现在，仍然在不断的高速发展，几乎考虑了服务治理的方方面面，开发起来非常的便利和简单。

Dubbo 于 2017 年开始又重启维护，发布了更新后的 2.5.7 版本，而 Spring Cloud 更新的非常快。

因此，企业需要根据自身的研发水平和所处阶段选择合适的架构来解决业务问题，不管是 Dubbo 还是 Spring Cloud 都是实现微服务有效的工具。

**微服务架构是互联网很热门的话题，是互联网技术发展的必然结果。它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。**

# 统一

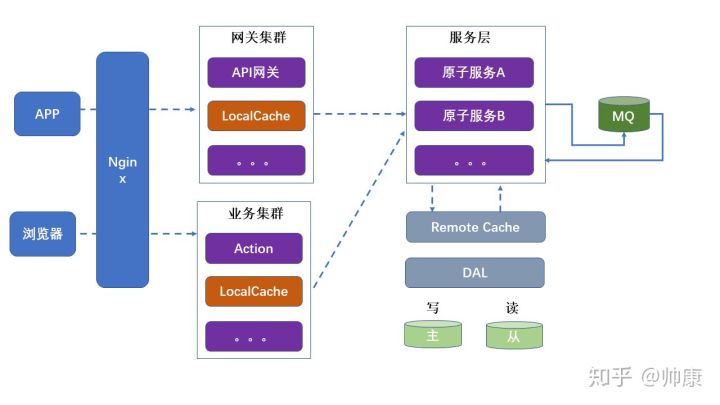
## 1.分布式和集群的联系与区别

分布式不一定就是不同的组件，同一个组件也可以，关键在于是否通过交换信息的方式进行协作。比如说Zookeeper的节点都是对等的，但它自己就构成一个分布式系统。

也就是说，分布式是指通过网络连接的多个组件，通过交换信息协作而形成的系统。而集群，是指同一种组件的多个实例，形成的逻辑上的整体。

可以看出这两个概念并不完全冲突，分布式系统也可以是一个集群，例子就是前面说的zookeeper等，它的特征是服务之间会互相通信协作。是分布式系统不是集群的情况，就是多个不同组件构成的系统；是集群不是分布式系统的情况，比如多个经过负载均衡的HTTP服务器，它们之间不会互相通信，如果不带上负载均衡的部分的话，一般不叫做分布式系统

## 2.微服务架构组成以及注意事项



**架构分解：**

* 网关集群： 数据的聚合、实现对接入客户端的身份认证、防报文重放与防数据篡改、功能调用的业务鉴权、响应数据的脱敏、流量与并发控制等。
* 业务集群：一般情况下移动端访问和浏览器访问的网关需要隔离，防止业务耦合。
* Local Cache：由于客户端访问业务可能需要调用多个服务聚合，所以本地缓存有效的降低了服务调用的频次，同时也提示了访问速度。本地缓存一般使用自动过期方式，业务场景中允许有一定的数据延时。
* 服务层：原子服务层，实现基础的增删改查功能，如果需要依赖其他服务需要在 Service 层主动调用。
* Remote Cache：访问 DB 前置一层分布式缓存，减少 DB 交互次数，提升系统的TPS。
* DAL：数据访问层，如果单表数据量过大则需要通过 DAL 层做数据的分库分表处理。
* MQ：消息队列用来解耦服务之间的依赖，异步调用可以通过 MQ 的方式来执行。
* 数据库主从：服务化过程中必经的阶段，用来提升系统的 TPS。

**注意事项：**

* 服务启动方式建议使用jar方式启动，启动速度快，更容易监控。
* 缓存、缓存、缓存，系统中能使用缓存的地方尽量使用缓存，通过合理的使用缓存可以有效的提高系统的TPS。
* 服务拆分要合理，尽量避免因服务拆分而导致的服务循环依赖。
* 合理的设置线程池，避免设置过大或者过小导致系统异常。

## 3.缓存穿透、缓存雪崩和缓存击穿

**【缓存穿透】**

场景：访问一个**不存在**的数据，每次都会访问DB。

解决方案：

通过一个key查询数据时，发现value不存在，我们可以将key对应的value设置一个默认值，并设置一个较短的缓存失效时间。

**【缓存雪崩】**

场景：**大量的缓存设置了相同的过期时间**，导致缓存在某一时刻同时失效，请求全部转发到了DB，DB由于压力过大瞬间崩溃。

解决方案：

随机设置缓存过期时间

**【缓存击穿】**

场景：访问热点数据，在缓存过期后，恰好这个时间点有大量的访问支持查询DB，导致DB崩溃

解决方案：

1.互斥锁（setnx）：

存在线程池阻塞的风险

Object get(String key){

value = redis.get(key);

if(value == null){

if(setnx(key1, “xx”)){

value = db.get(key);

redis.put(key, value, timeout);

delete(key1);

}

}

return value;

}

2.永远不过期：

只要访问就延长过期时间

3.Hystrix服务限流降级

## 4.声明式编程和编程式实现