您好，我是来自华南理工大学软件学院的连嘉伟，我的专业绩点排名年级前25%,曾获得两次国家励志奖学金。在校期间学习了计算机网络，数据库，操作系统，数据结构等相关知识，我使用的语言是java，现在正在阿里健康实习，主要负责页面可配置化项目的后端研发，该项目主要解决了B端页面重复开发的问题，对于一些功能和格式相对简单的页面，只需要添加相应页面配置，就可以完成相关页面的前后端开发。希望能够通过面试获得贵部门的后端岗位。

三个词形容自己

沉稳、乐于助人、有责任感

别人对我的评价

热心、有集体荣誉感和责任感

项目

项目中遇到的困难以及怎么解决：

股票量化交易：

计网

TCP与UDP

TCP三次握手与四次挥手

握手时为什么TCP客户端最后还要发送一次确认呢?

3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

> 现在把三次握手改成仅需要两次握手，死锁是可能发生的。作为例子，考虑计算机S和C之间的通信，假定C给S发送一个连接请求分组，S收到了这个分组，并发 送了确认应答分组。按照两次握手的协定，S认为连接已经成功地建立了，可以开始发送数据分组。可是，C在S的应答分组在传输中被丢失的情况下，将不知道S 是否已准备好，不知道S建立什么样的序列号，C甚至怀疑S是否收到自己的连接请求分组。在这种情况下，CS认为连接还未建立成功，将忽略S发来的任何数据分 组，只等待连接确认应答分组。而S在发出的分组超时后，重复发送同样的分组。这样就形成了死锁。

挥手时为什么客户端最后还要等待2MSL?

虽然按道理，四个报文都发送完毕，我们可以直接进入CLOSE状态了，但是我们必须假象网络是不可靠的，有可以最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。在Client发送出最后的ACK回复，但该ACK可能丢失。Server如果没有收到ACK，将不断重复发送FIN片段。所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最

大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

为什么建立连接是三次握手，关闭连接确是四次挥手呢?

因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

如果已经建立了连接，但是客户端突然出现故障了怎么办?

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

输入url到浏览器显示html经历了哪些步骤

1. **DNS解析**
2. **TCP连接**
3. **发送HTTP请求（如果是https请求的话还要进行https握手）**
4. **服务器处理请求并返回HTTP报文**
5. **浏览器解析渲染页面**
6. **连接结束**

DNS解析

第一步：浏览器将会检查缓存中有没有这个域名对应的解析过的IP地址，如果有该解析过程将会结束。浏览器缓存域名也是有限制的，包括缓存的时间、大小，可以通过TTL属性来设置。

第二步：如果用户的浏览器中缓存中没有，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

第三步：如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

第四步：如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/ip参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

第五步：如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

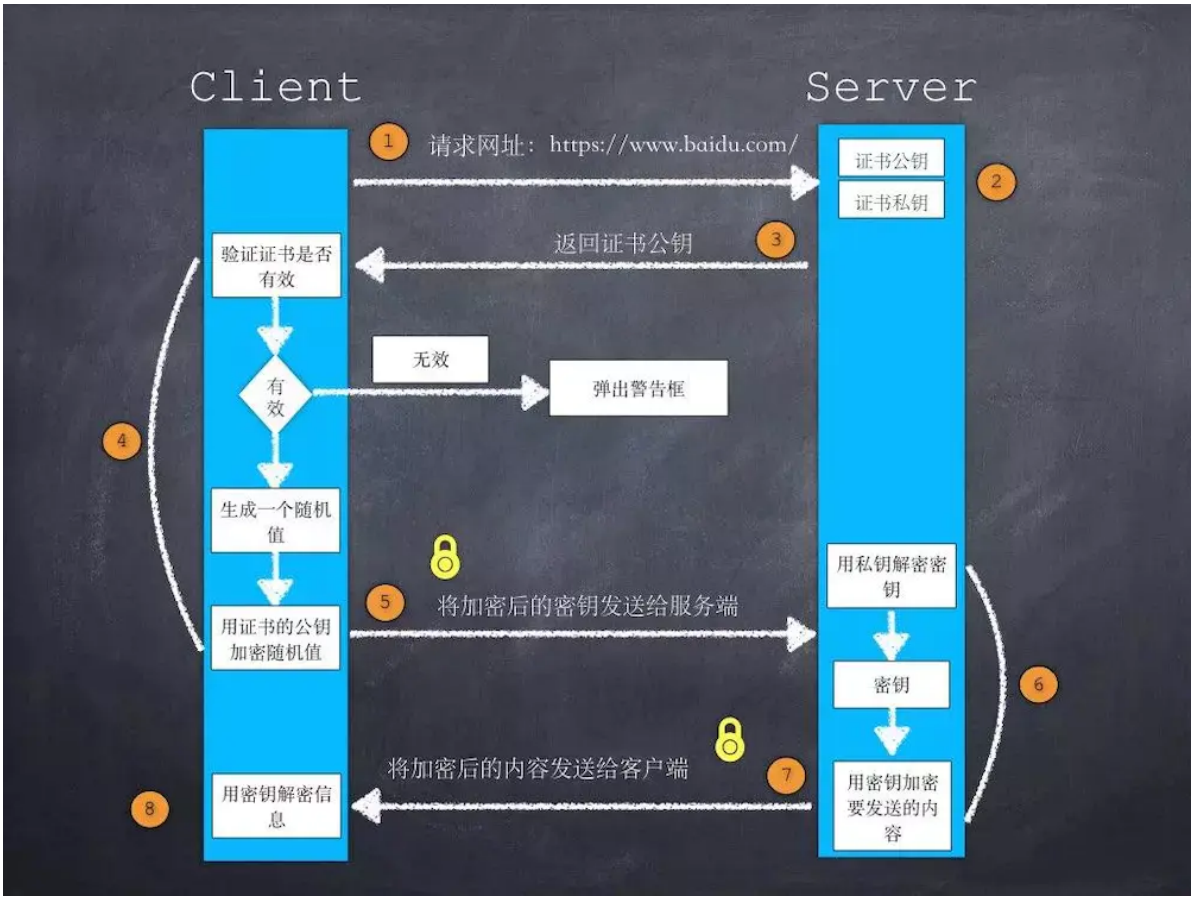
第六步：如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地DNS就把请求发至13台根DNS，根DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找域名域服务器，重复上面的动作，进行查询，直至找到域名对应的主机。

第七步：如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

总得来说就是

浏览器缓存 ----> 本地hosts文件 ----> 本地DNS服务器 ----> 根DNS服务器 ----> 顶级域名服务器 ----> 管理.com的DNS服务器 ----> 本地DNS服务器

https握手（SSL/TLS握手）



首先，客户端发起握手请求，以明文传输请求信息，包含版本信息，加密-套件候选列表，压缩算法候选列表，随机数，扩展字段等信息(这个没什么好说的，就是用户在浏览器里输入一个HTTPS网址，然后连接到服务端的443端口。)

服务端的配置，采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面。这套证书其实就是一对公钥和私钥。如果对公钥不太理解，可以想象成一把钥匙和一个锁头，只是世界上只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

服务端返回协商的信息结果，包括选择使用的协议版本 version，选择的加密套件 cipher suite，选择的压缩算法 compression method、随机数 random_S 以及证书。(这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。)

客户端验证证书的合法性，包括可信性，是否吊销，过期时间和域名。(这部分工作是由客户端的SSL/TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警示框，提示证书存在的问题。如果证书没有问题，那么就生成一个随机值。然后用证书（也就是公钥）对这个随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。)

客户端使用公匙对对称密匙加密，发送给服务端。(这部分传送的是用证书加密后的随机值，目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。)

服务器用私钥解密，拿到对称加密的密匙。(服务端用私钥解密后，得到了客户端传过来的随机值，然后把内容通过该随机值进行对称加密，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。)

传输加密后的信息，这部分信息就是服务端用私钥加密后的信息，可以在客户端用随机值解密还原。

客户端解密信息，客户端用之前生产的私钥解密服务端传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

数据库

索引用B+树的好处

检索效率稳定，每次都要到叶子节点

中间节点没有存放数据，能放更多的指针，树结构更加扁平化，IO次数减少

范围查找更有效率，因为每块数据都有一根指针连接起来

事务的传播机制

多个事务方法相互调用时,事务如何在这些方法间传播。

REQUIRED (required)：如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
SUPPORTS(supports)：如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
MANDATORY(mandatory)：如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。
REQUIRES_NEW(requires_new)：创建一个新的事务，如果当前存在事务，则把当前事务挂起。
NOT_SUPPORTED(not_supported)：以非事务方式运行，如果当前存在事务，则把当前事务挂起。
NEVER(never)：以非事务方式运行，如果当前存在事务，则抛出异常。
NESTED(nested)：如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 REQUIRED 。
指定方法：通过使用 propagation 属性设置，例如：@Transactional(propagation = Propagation.REQUIRED)

redis

三种缓存读写策略

1. ### Cache Aside Pattern（旁路缓存模式）比较常见

   写：

   - 先更新 DB
   - 然后直接删除 cache 。

   读：

   - 从 cache 中读取数据，读取到就直接返回
   - cache中读取不到的话，就从 DB 中读取数据返回

- 再把数据放到 cache 中。

2. **Read/Write Through Pattern（读写穿透）非常少见**

写（Write Through）：

- 先查 cache，cache 中不存在，直接更新 DB。
- cache 中存在，则先更新 cache，然后 cache 服务自己更新 DB（**同步更新 cache 和 DB**）。

读(Read Through)：

- 从 cache 中读取数据，读取到就直接返回 。
- 读取不到的话，先从 DB 加载，写入到 cache 后返回响应。

3. **Write Behind Pattern（异步缓存写入）非常少见**

Write Behind Pattern 和 Read/Write Through Pattern 很相似，两者都是由 cache 服务来负责 cache 和 DB 的读写。但是，两个又有很大的不同：**Read/Write Through 是同步更新 cache 和 DB，而 Write Behind Caching 则是只更新缓存，不直接更新 DB，而是改为异步批量的方式来更新 DB。**

操作系统

java

为什么重写 `equals` 时必须重写 `hashCode` 方法?

equals（）相等的的两个等价对象因为hashCode不同，所以在hashmap中的table数组的下标不同，从而这两个对象就会同时存在于集合中，在调用hashmap集合中的方法时就会出现逻辑的错误，也就是，你的equals（）方法也“白白”重写了。

因此，对于“为什么重写equals()就一定要重写hashCode()方法？”这个问题应该是有个前提，就是你需要用到HashMap,HashSet等Java集合。用不到哈希表的话，其实仅仅重写equals()方法也可以吧。而工作中的场景是常常用到Java集合，所以Java官方建议重写equals()就一定要重写hashCode()方法。

线程、程序、进程的基本概念

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。 线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

线程有哪些基本状态

BIO,NIO,AIO 有什么区别

- **BIO (Blocking I/O):** 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (Non-blocking/New I/O):** NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel , Selector，Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。 NIO 提供了与传统 BIO 模型中的 `Socket` 和 `ServerSocket` 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O):** AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的 IO 模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

java反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

java绑定

绑定是指一个方法的调用和该方法所属的类（所在的类）相关联，意思就是在执行方法调用的时候，jvm所知道调用了哪个类的方法，类和调用方法相关联。

静态绑定就是程序在执行前就知道了该方法所属的类，即在编译前该方法已经绑定，在java中只有private， static，final修饰的方法和构造方法是静态绑定

运行时绑定是指在程序运行时根据对象类型进行绑定，也叫动态绑定或者后期绑定

spring

IOC/DI

当某个角色需要另外一个角色协助的时候，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在spring中创建被调用者的工作不再由调用者来完成，因此称为控制反转。创建被调用者的工作由spring来完成，然后注入调用者。因此也称为依赖注入。

AOP

AOP能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任，例如事务处理、日志管理、权限控制等，封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

好处：降低模块的耦合度、使系统容易扩展、提高代码复用性

springboot

自动装配

通过注解或者一些简单的配置就能在 Spring Boot 的帮助下实现某块功能。Spring Boot 通过@EnableAutoConfiguration开启自动装配，通过 SpringFactoriesLoader 最终加载META-INF/spring.factories中的自动配置类实现自动装配，自动配置类其实就是通过@Conditional按需加载的配置类，想要其生效必须引入spring-boot-starter-xxx包实现起步依赖。

springcloud

CAD

- C：数据一致性(consistency)
    - **所有**节点拥有数据的最新版本
- A：可用性(availability)
    - 数据具备高可用性
- P：分区容错性(partition-tolerance)
    - **容忍网络出现分区**，分区之间网络不可达。

消息队列

优势

削峰填谷（主要解决瞬时写压力大于应用服务能力导致消息丢失、系统奔溃等问题）

系统解耦（解决不同重要程度、不同能力级别系统之间依赖导致一死全死）

异步提速，提升性能（当存在一对多调用时，可以发一条消息给消息系统，让消息系统通知相关系统）

蓄流压测（线上有些链路不好压测，可以通过堆积一定量消息再放开来压测）

缺点

降低系统可用性

提高系统复杂度

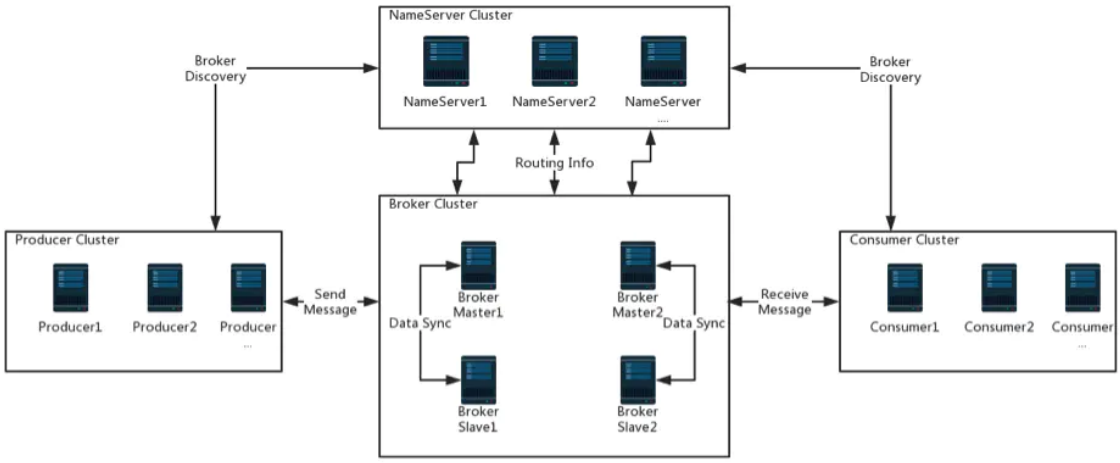
带来一系列的重复消费，顺序消费，分布式事务，消息堆积的问题

Rocketmq相比于Rabbitmq、kafka具有主要优势特性有：

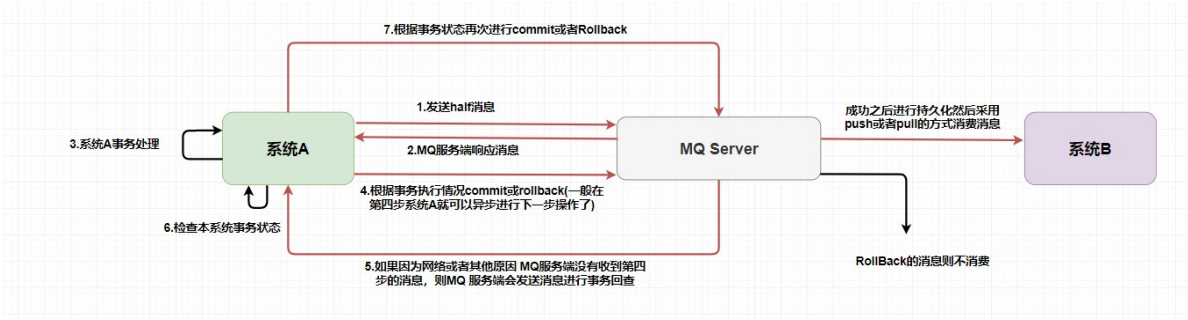
- 支持事务型消息（消息发送和DB操作保持两方的最终一致性，rabbitmq和kafka不支持）
- 支持结合rocketmq的多个系统之间数据最终一致性（多方事务，二方事务是前提）
- 支持18个级别的延迟消息（rabbitmq和kafka不支持）
- 支持指定次数和时间间隔的失败消息重发（kafka不支持，rabbitmq需要手动确认）

- 支持consumer端tag过滤，减少不必要的网络传输（rabbitmq和kafka不支持）
- 支持重复消费（rabbitmq不支持，kafka支持）

Rocketmq



分布式事务



Name Server

Name Server是一个几乎无状态节点，可集群部署，节点之间无任何信息同步的注册中心，类似于Eureka、Zookeeper。

Broker

主要负责消息的存储、投递和查询以及服务高可用保证。说白了就是消息队列服务器嘛，生产者生产消息到 Broker ，消费者从 Broker 拉取消息并消费。

幂等性

一个操作执行多次与执行一次的结果相同。生产者给mq的消息是不会重复的，因为有两阶段提交。而mq给消费者的消息是可能重复的，因为可能mq没收到消费者的ack所以重新发了一份，但是概率小，加锁不划算。可以通过写入redis来保证，因为 `Redis` 的 `key` 和 `value` 就是天然支持幂等的。还有使用 **数据库插入法** ，基于数据库的唯一键来保证重复数据不会被插入多条。

redis：可以通过加锁来实现，但是因为重复消息出现的可能性较小，加锁有点过了，可以使用redis的lua脚本实现原子性操作进而保证原子性。

Rocketmq整个流程

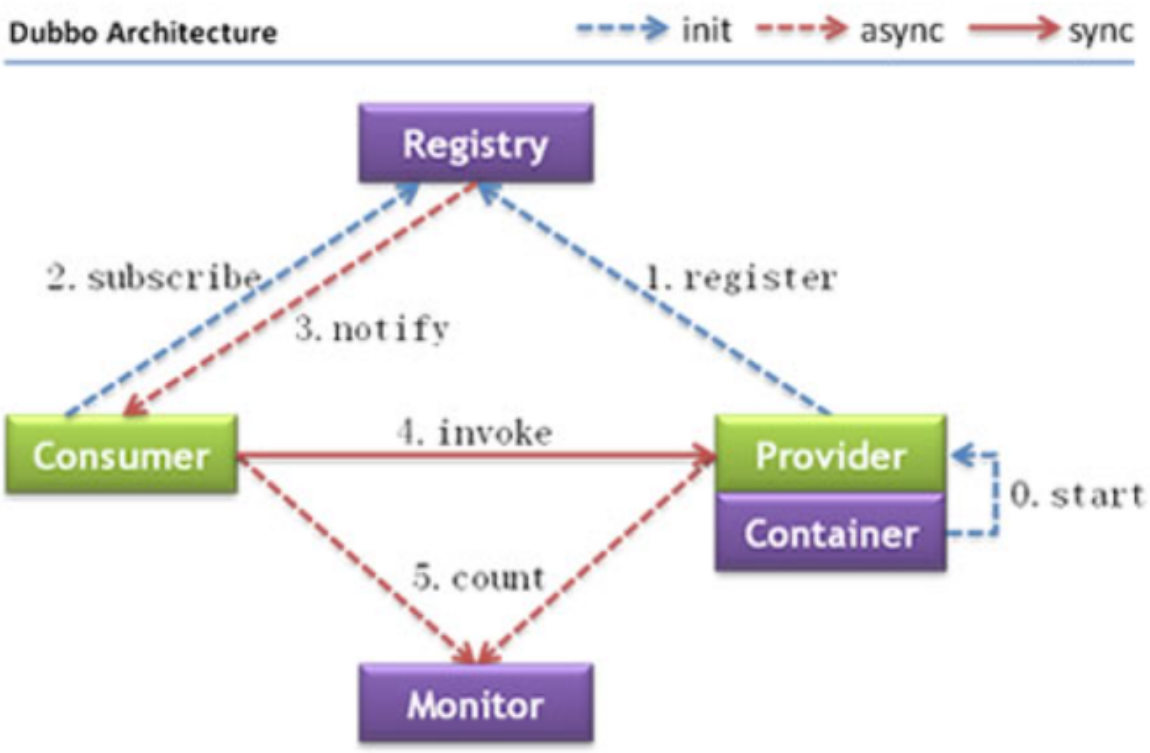
1.发送消息，producer会在本地缓存的broker列表里面获取一个broker，如果没有，那就会去namesrv获取broker的地址，然后发送消息到broker。
2.broker接受到消息后，会进行一系列检查，然后把收到的消息，存入到commitLog当中，然后进行刷盘，如果是同步的刷盘策略，那么就会在写入到pageCache中时，再真正落地到磁盘的时候才会进行response的返回，如果是异步的话，那么会在写入pageCache中时即可返回response，然后累计一定量的消息的时候才会持久化，相应的，异步的策略的效率更高了，可是带来的就是风险性的评估，那么这个时候选择就需要根据实际情况来考虑问题了。
3.后台有一个线程ReputMessageService一直在查看commitLog的偏移量，如果commitLog的写入偏移了，那么这个时候就会把消息的长度，id，起始位置等传送给consumeQueue，和index（index就是索引，根据topic和id得到相应msg的位置）。
4.consumer端消费，有两种模式，push和pull模式，其实都是基于pull来做的，PullMessageProcessor里面是相应的逻辑处理，大致的处理流程：系统能够知道每个consumeQueue现在读取到的位置，如果没有消息未读取的话，就会阻塞response，默认20s。当有消息的时候，先会去读取consumeQueue，然后读取出消息的起始位置等，然后去commitLog读取消息，并且修改commitLog中的消息的读取次数等，消费者接着处理业务逻辑，然后返回给broker相应的结果，如果消费成功，那么修改consumeQueue的消费的位置（不一定刷盘，所以需要保证业务端幂等）。如果消费失败，那么普通消息就会进入到重试队列，（类似于延时队列）进行重试，达到一定重试次数之后仍然失败，那么就会放入到死信队列当中，这个时候需要人工介入（例如这个时候，可以在这个地方进行源码的二次开发，或者注册监听器，当有消息放入死信队列的时候就发送提醒）。

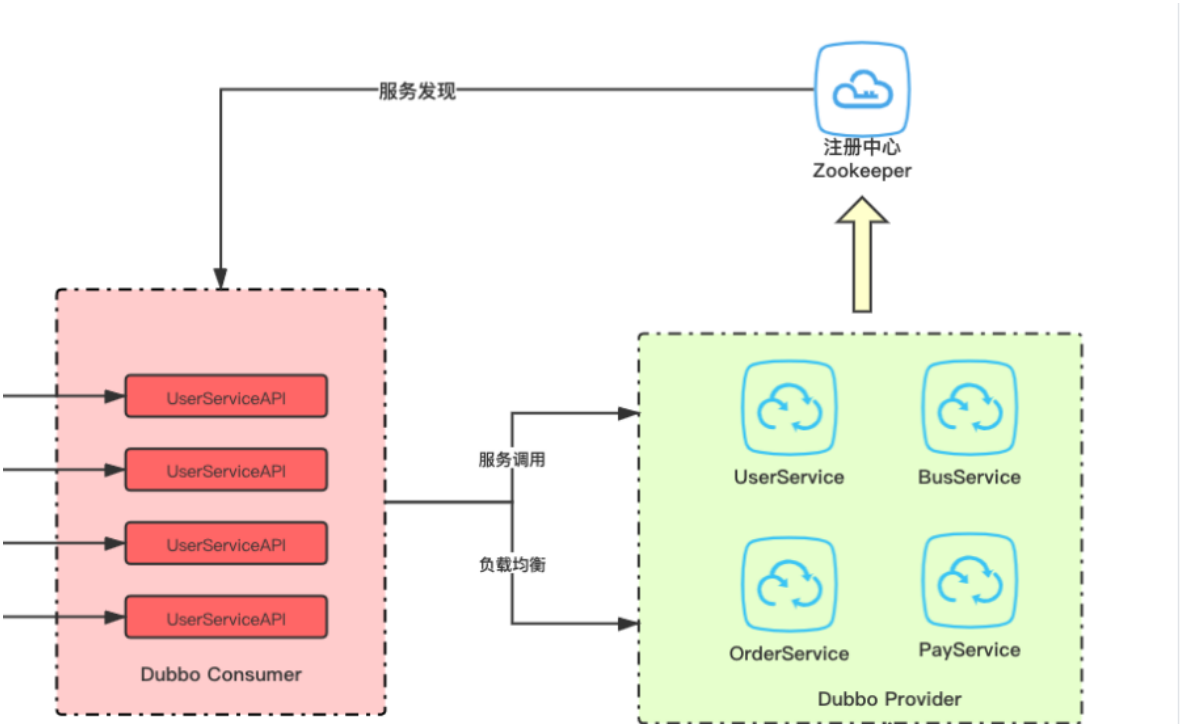
Zookeeper

分布式服务治理框架，用于服务注册与发现，实现远程调用功能等。

Dubbo

架构图





节点角色说明

节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

调用关系说明

0.服务容器负责启动，加载，运行服务提供者。

1.服务提供者在启动时，向注册中心注册自己提供的服务。

2.服务消费者在启动时，向注册中心订阅自己所需的服务。

3.注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

4.服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

5.服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。