课程目标

1、了解什么是分布式、云原生

2、了解什么是消息中间件

3、了解主要的消息协议

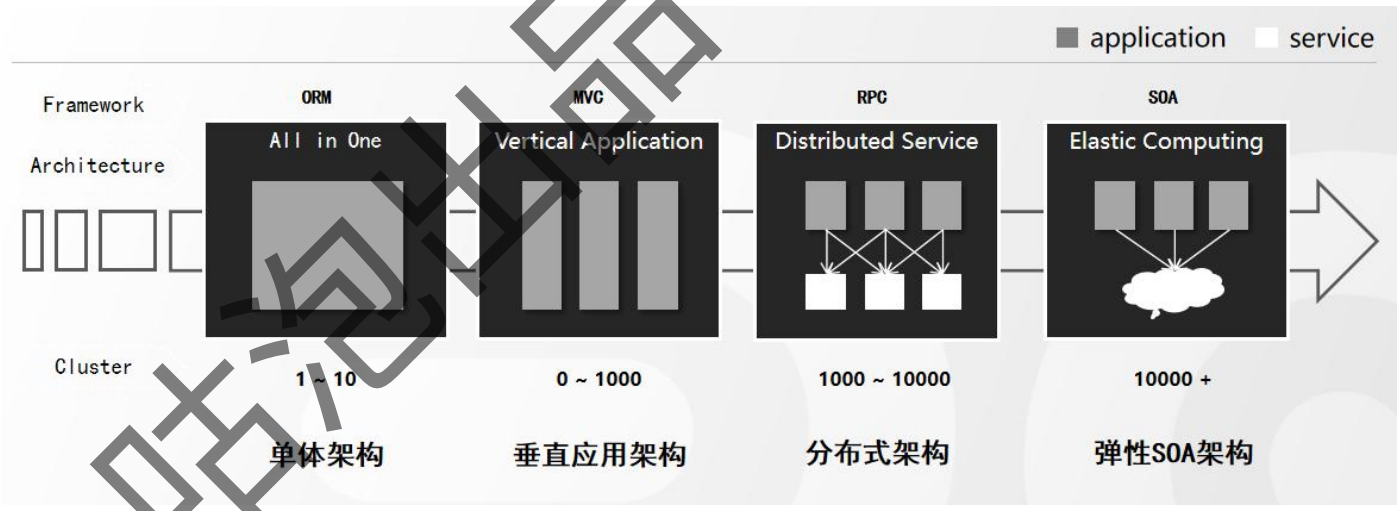
4、了解消息持久化

5、了解消息分发机制

6、了解高可用和高可靠的区别

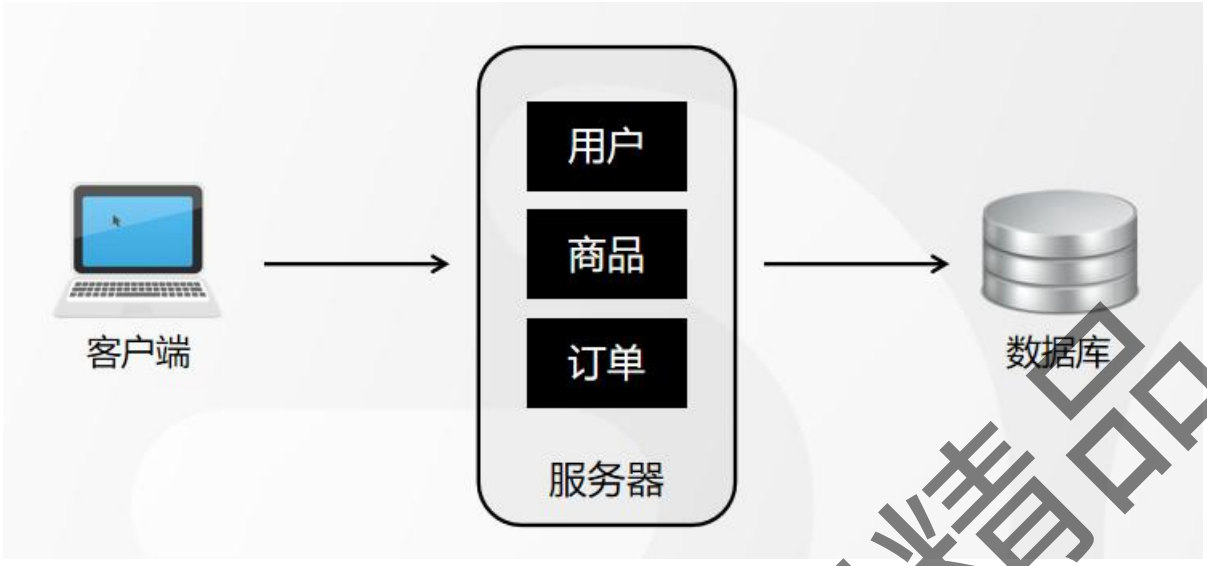
1、分布式架构的演进过程

分布式微服务架构的发展，主要经历了四个阶段：单一应用架构、垂直应用架构、分布式架构和弹性 SOA 架构。



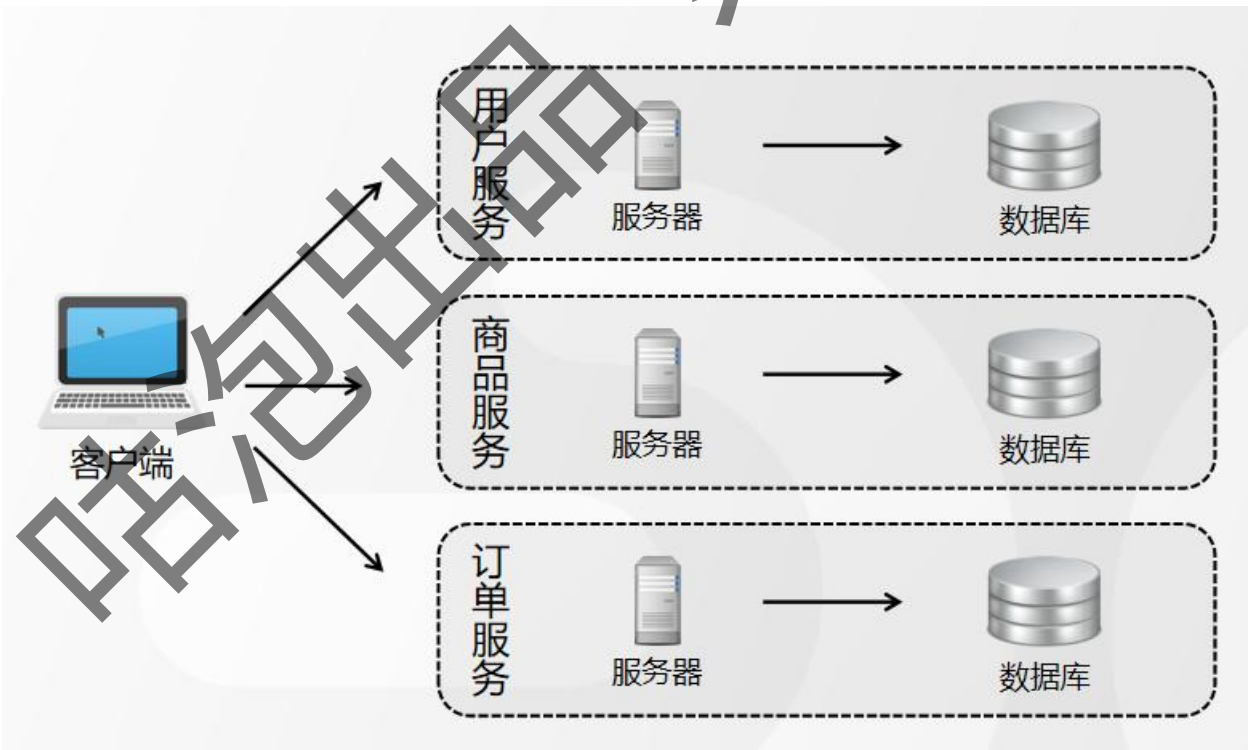
这张图是从 Dubbo 官网上下载的描述分布式架构演进过程的示意图，大家可以收藏一下。

1.1 单体架构（All in One）



从 2000 年开始，互联网在中国开始盛行起来，但是那时候网民数较少，网站流量也很小，只需要一台机器就可以运行所有的服务，只需要 All in One 单体架构就能满足需求。

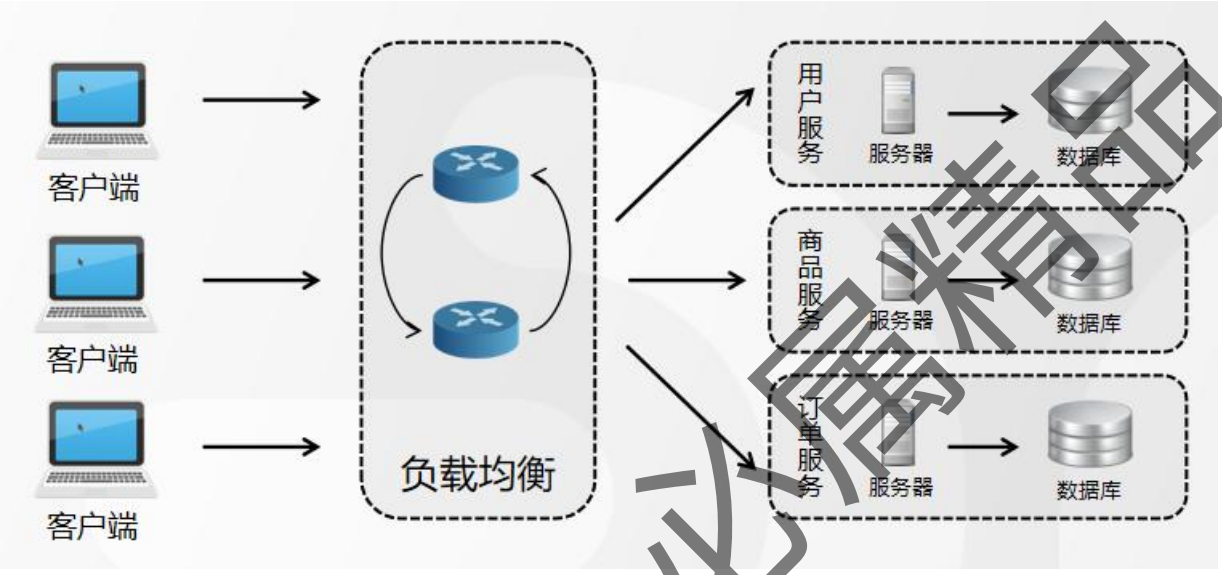
1.2 垂直应用架构(Vertical Application)



随着网站访问量的增加，一旦服务器或者数据库出现问题，将会导致整个系统故障，造成所有服务不可用，而且功能修改发布也不方便，所以，就把大系统拆分成了多个子系统，

比如将“用户”、“商品”、“订单“等按业务进行拆分，也就是”垂直拆分“，并且每个子系统都部署到不同的服务器上。

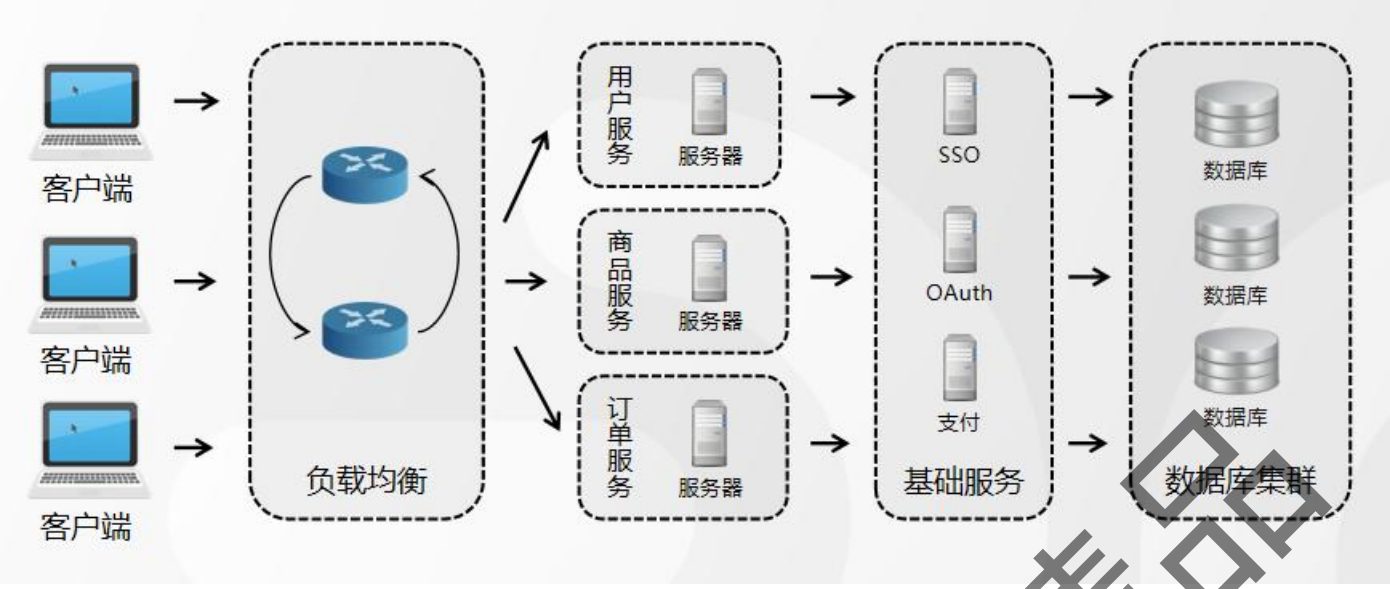
1.3 分布式架构(Distributed Service)



随着用户访问越来越大，意味着需要更多的服务器才能支撑服务的运行，而应用之间的交互不可能避免地变得越来越复杂。为了保证服务稳定，提高管理效率，需要对这些应用做负载均衡。因为用户不可能去了解后台服务器的数量和业务结构，有了负载均衡以后，用户只需要统一访问负载均衡服务器就可以，后端的应用就可以根据流量的大小进行动态的扩缩容，也就是“水平扩展”。

1.4 弹性 SOA 架构（Elastic Computing)

分布式架构运行一段时间之后，发现商品和订单服务中有很多重复的功能，比如 SSO、OAuth 权限和支付等。一旦项目大了，集群部署多了，这些重复的功能无疑就是浪费资源，就需要把这些重复的功能单独抽离出来单独部署，并且给它们起了个名字叫“Service(服务)“，并且对服务进行分层，比如 Base Service 基础服务、Business Service 业务服务等等。



有了服务的概念之后，我们还可以根据业务需要，将服务进一步细化并拆分。比如把对用户服务可以划分为用户积分服务、个人优惠券服务等等，这就是我们通常所说的微服务架构。

但是往往对于微服务架构实际上没有一个明确的定义，因为微服务的边界很难确定，所以微服务架构一直在不断地重构过程中发展，没有完美的微服务架构，只有适合公司业务场景的微服务架构。

当然，以上是我个人对微服务的理解，为了帮助大家更准确的理解微服务，这里引用Martin Folwer 前辈对于微服务描述的一段话，英文我就不读了，大家可以通过翻译软件来理解一下上述这段话。

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

1.5 服务网格

在 Java 中微服务架构的主流解决方案是 Spring Cloud，给程序员带来了非常便捷的开发体验。但是，有了 Spring Cloud 的微服务架构就完美了吗？现实并非如此，很快我们就发现微服务带来了很多的问题，我大致总结为以下三点：

第 1 点：非功能代码侵入严重。为了满足业务需要，我们可以使用框架提供给我们的各种组件，比如网络通信、消息处理等。于是，我们需要引入各种依赖， 加注解，写配置，最后将这些非业务功能的代码和业务代码打包一起部署，这就是“侵入式框架”；

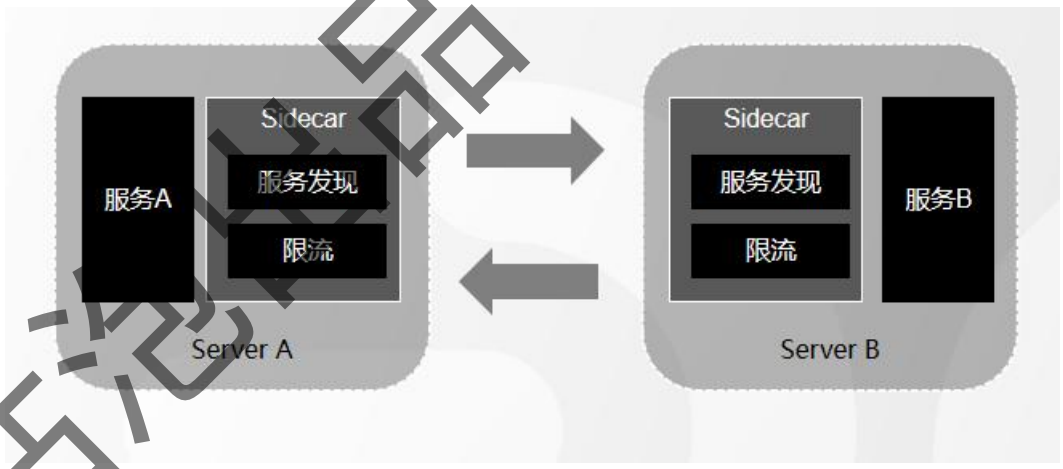
第 2 点：微服务框架学习成本还是较大。Spring Cloud 虽然能解决微服务领域的很多问题，但是需要程序员花费比较多的精力来学习它的使用。本来程序员应该把更多的精力投入到业务开发上，而不应该是非业务上。

第 3 点：维护成本变高。互联网公司产品的版本升级是非常频繁的，因为 Spring Cloud 是“代码侵入式的框架”，为了维护各个版本的兼容性、权限、流量等，这时候版本的升级就注定要让非业务代码一起， 再加上多语言之间的调用，一旦出现问题，程序员们会非常痛苦。

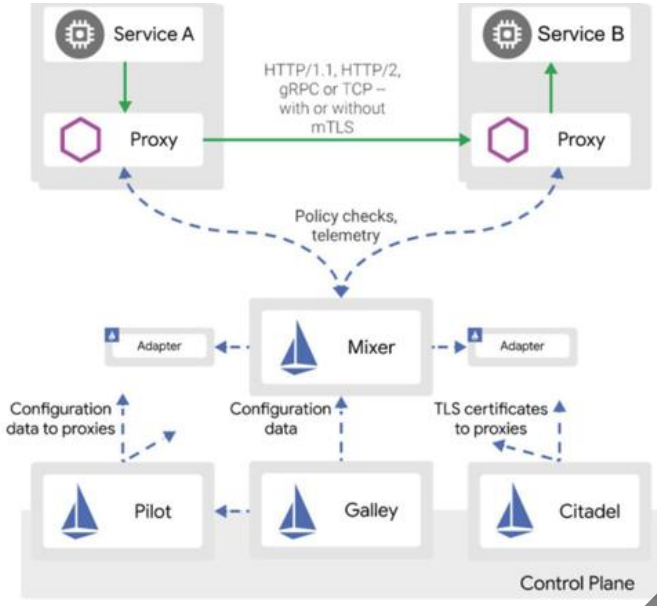
其实，大家有没有发现？服务拆分的越细，只是感觉上解耦了，但是维护成本却变高了，那怎么办呢？

网络的问题当然还是要交给网络本身来解决。所以，服务网格的技术就应运而生，也就是 Service Mesh。

我们可以为每个服务单独配置一个 Sidecar，所有服务通信的进出口流量都会通过 Sidecar 来进行操作，如图所示：



常见的服务网格产品有 Prana、Local Proxy、Linkerd、Istio 等，目前比较主流的是 Istio，我们来看看 Istio 的架构图：



可以发现，该架构图分为数据平面和控制平台，数据平面就是前面的 ServiceA 和 ServiceB 组成的部分，控制平面主要指的是 Mixer、Pilot、Galley、Cidatel 等组件，这些组件的详细功能这里就不详细展开了，大家感兴趣的话回复 666 ，我可以单独出一期视频。

回到主题，我们可以思考一下这些组件和服务部署起来是否会有难度？答案是肯定的，因为大家使用的基础设施和操作系统可能会有差异，所以，这时候我们需要使用容器化技术来部署。

1.6 容器化云原生架构

架构演进到现在这个阶段，我们可以基于容器化和容器编排的技术来实现公有云、私有云、混合云地无缝迁移，也就是可以使用云原生架构来考虑组件和服务的部署方案。通过这张图来理解一下：

不管底层的基础设施是什么，都可以用Kubernetes 作为适配，帮助开发运维人员屏蔽底层硬件的细节，同时采用DevOps 来打破传统开发和运维之间的界限。

2、分布式系统面临的挑战

2.1 分布式系统的普遍特征

高吞吐：更高的吞吐量，以每秒钟能处理的请求数来衡量。

高可用：也可以叫做容错性，以请求的成功率来衡量。可以用请求成功次数比例来算，也可以用正常服务时间比例来算。我们见到的多少个 9——99.9999%，就是对可用性的描述。

高并发：更高的并发量，系统同时能支持的最大请求数。

可伸缩：更侧重于系统的水平伸缩，通过廉价的服务器实现分布式计算；

可扩展：设计的可扩展性，代码的可扩展性.也可以说是系统设计的松耦合性，也可以说系统的松耦合性

低延时：更低的延迟，以交易平均完成时间来衡量。

2.2 分布式系统的数据一致性

怎么做到一致呢?肯定要有一个指导思想。所以有计算机大牛提出了 CAP 理论、BASE 理论，给大家实现一致性提供参考。后面又有了 ZAB 协议、Paxos 算法、Raft 算法等等来帮助大家实现一致性或者说达成共识。我们说数据库的本地事务是为了保证一致性，那么分布式事务就是为了保证分布式环境的一致性。当然，一致性并不一定只有事务一种手段去保证，因为分布式环境不一定需要实时一致。

2.3 分布式的开源技术

对于分布式环境存储、通信、计算的需求，出现了各种各样的开源技术，比如:1、服务协调。如果要管理分布式环境中的服务节点，或者协调节点之间的操作，必须要有一个中心化的节点，代表技术:Zookeeper，它可以用来实现分布式锁，注册中心等等。

2、消息中间件。如果要实现异步通信，需要用到存储和转发消息的中间件，代表技术: RabbitMQ、 Kafka、

RocketMQ，还可以用来实现服务解耦、削峰。

3、NoSQL 存储。如果要实现非关系型数据的存储，提供很高的读写速度，需要 NoSQL 存储的存储:代表技术: Redis，可以实现缓存、分布式锁、分布式 ID、分布式 Session 等。

4、任务调度。为了避免多个节点任务重复执行和漏掉执行，必须要有分布式环境下的任务调度工具，代表技术:Elastic-Job、xxl-job。

5、数据存储。为了实现分布式环境下的数据分片存储和一致性，关系型数据库本身是做不到的，必须提供动态数据源选择、数据源路由、SQL 改写、结果汇总等功能，代表技术: Mycat、 Sharding-JDBC。

6、负载均衡。为了解决分摊访问压力和提供统一入口的问题，需要有负载均衡的组件。代表技术:Nginx、HAProxy。

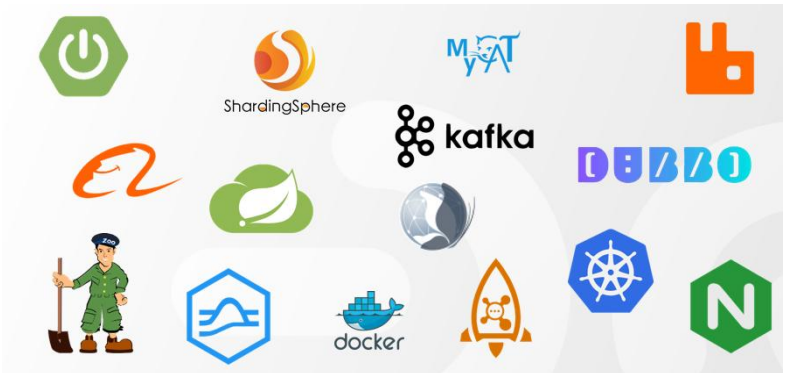
7、日志。当系统拆分后，根据日志追踪问题变得非常麻烦，需要一个个节点去查看，所以需要一个可以聚合日志并且提供搜索功能的系统，代表技术:ELK。

8、通信。分布式环境中的通信，代表技术:Netty。很多中间件的底层通信都是用 Netty 实现的。Dubbo 基于 Netty 实现了 RPC。

9、文件系统。为了解决文件的分片管理，比如淘宝的商品图片，一个节点肯定是搞不定的，需要分布式的文件管理系统:GFS、HDFS。

10、内容分发(CDN)。为了解决地域带来的网络延迟问题，需要把内容缓存在不同区域的节点上，用户访问时就近选择节点。这个需要网络提供商实现。

11、容器。为了提升硬件资源的利用效率，解决环境差异和组件依赖带来的部署问题，需要容器化技术，代表技术:Docker、K8s。



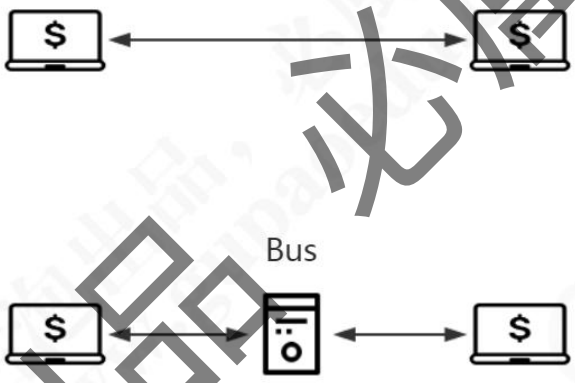
这些技术共同构成了分布式的生态体系。

3、消息中间件

3.1 MQ 的诞生历程

我们要去用 MQ，先来了解一下 MQ 是怎么诞生的，这样对于它解决了什么问题理解会更加深刻。

以前网络上的计算机 (或者说不同的进程) 传递数据，通信都是点对点的，而且要实现相同的协议 (HTTP、TCP、WebService）。1983 年的时候，有个在 MIT 工作的印度小伙突发奇想，能不能发明一种专门用来通信的中间件，就像主板（BUS）一样，把不同的软件集成起来呢？于是他搞了一家公司（Teknekron），开发了世界上第一个消息队列软件 The Information Bus(TIB)。最开始的时候，它被高盛这些公司用在金融交易里面。



3.2 什么是 MQ?

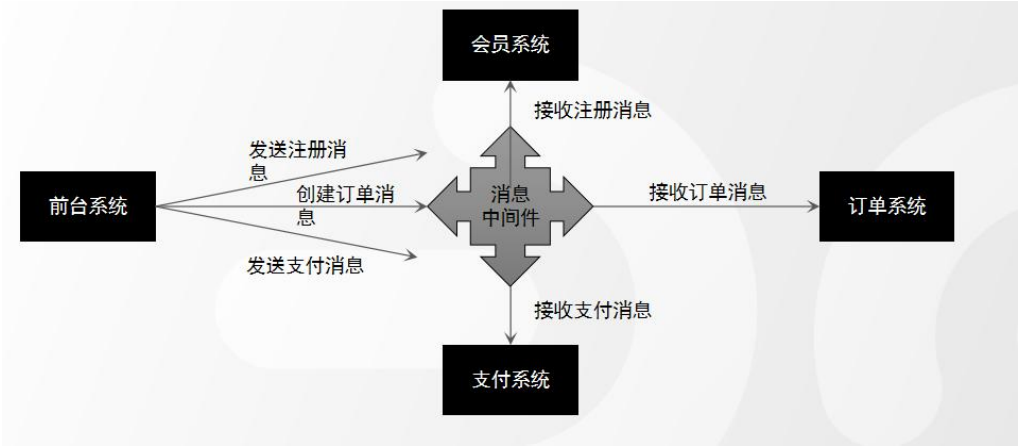
MQ 全称是 Message Queue，直译过来叫做消息队列，主要是作为分布式应用之间实现异步通信的方式。

主要由三个部分组成，分别是生产者、消息服务端和消费者。

生产者（Producer），是生产消息的一端，相当于消息的发起方，主要负责载业务信息的消息的创建。
然后是消息服务端（Server），是处理消息的单元，本质就是用来创建和保存消息队列，它主要负责消息的存储、投递以及跟消息队列相关的附加功能。消息服务端是整个消息队列最核心的组成部分。
第三个是消费者（Consumer），是消费消息的一端，主要是根据消息所承载的信息去处理各种业务逻辑。

3.3 为什么要使用 MQ?

这个问题换一个问法是这样说，如果是为了解决通信的问题，有那么不同不同协议的通信方式，比如 TCP、HTTP、RPC，为什么偏偏要选择 MQ?



因为 MQ 有一些特性是传统通信协议所不具备的，所以在一些特殊场景下更有优势。目前，主流的消息中间件有 ActiveMQ、RabbitMQ、Kafka、RocketMQ、Pulsar。在技术选型的时候，可以根据具体的业务需求更合适的中间件。

数据量大、吞吐量要求比较高的场景一般采用 Kafka；

对消息可靠性要求很高，甚至要求支持事务的场景，比如金融互联网，可以选择 RocketMQ；

对于中小型公司来说， 可以选择 RabbitMQ，它利用 erlang 语言本身的并发优势，性能好 在微秒级。

而 Pulsar 近两年开始流行起来了，它是下一代云原生分布式消息流平台，可以集消息、存储、轻量化函数式计算为一体。

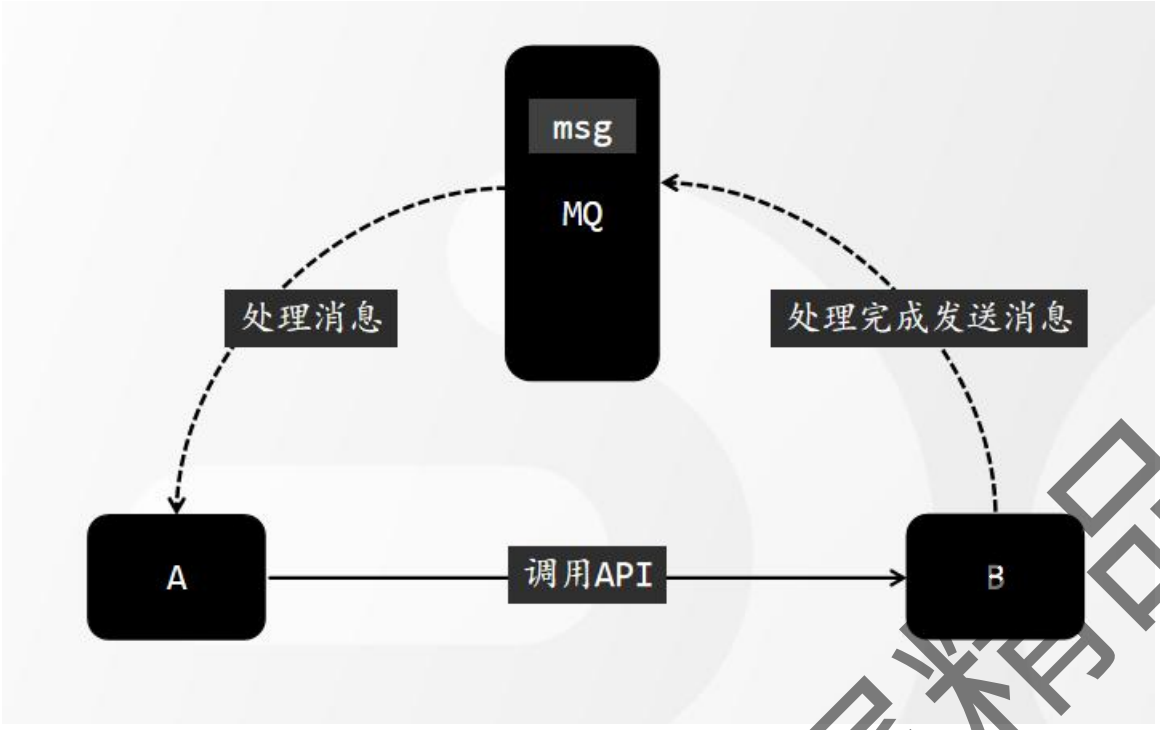
至于 ActiveMQ，目前的一些新项目很少有人用了。

3.3.1 实现异步通信

同步的通信是什么样的?

发出一个调用请求之后，在没有得到结果之前，就不返回。由调用者主动等待这个调用的结果。

而异步是相反的，调用在发出之后，这个调用就直接返回了，所以没有返回结果。也就是说，当一个异步过程调用发出后，调用者不会马上得到结果。而是在调用发出后，被调用者通过状态、通知来通知调用者，或通过回调函数处理这个调用。



举个例子：

大家都用过手机银行的跨行转账功能。大家用 APP 的转账功能的时候，有一个实时模式，有一个非实时模式。

实时转账实际上是异步通信，因为这个里面涉及的机构比较多，调用链路比较长，本行做了一些列的处理之后，转发给银联或者人民银行的支付系统，再转发给接收行，接受行处理以后再原路返回。

所以转账以后会有一行小字提示：具体到账时间以对方行处理为准，也就是说转出行只保证了这个转账的消息发出。那为什么到账时间又这么快呢？很多时候我们转账之后，不用几秒钟对方就收到了。是因为大部分的 MQ 都有一个低延迟的特性，能在短时间内处理非常多的消息。

很多理财软件提现也是一样，先提交请求，到账时间不定。这个是用 MQ 实现系统间异步通信的一个场景。

异步通信不需要客户端等待，可以减少客服端性能消耗，大大地提升用户体验。

3.3.2 实现系统解耦

第二个主要的功能，是用来实现系统解耦。既然说到解耦，那我们要先来了解一下耦合的概念。

耦合是系统内部或者系统之间存在相互作用，相互影响和相互依赖。

在分布式系统中，一个业务流程涉及多个系统的时候，他们之间就会形成一个依赖关系。

比如我们以 12306 网站退票为例，在传统的通信方式中，订单系统发生了退货的动作，那么要依次调用所有下游系统的 API，比如调用库存系统的 API 恢复库存，因为这张火车票还要释放出去给其他乘客购买；调用支付系统的 API，不论是支付宝微信还是银行卡，要把手续费扣掉以后，原路退回给消费者；调用通知系统 API 通知用户退货成功。

```
// 伪代码
public void returnGoods(){
    stockService.updateInventory ();
    payService.refund();
    expressService.notice();
}
```

这个过程是串行执行的，如果在恢复库存的时候发生了异常，那么后面的代码都不会执行。由于这一系列的动作，恢复库存，资金退还，发送通知，本质上没有一个严格的先后顺序，也没有直接的依赖关系，也就是说，只要用户提交了退货的请求，后面的这些动作都是要完成的。库存有没有恢复成功，不影响资金的退还和发送通知。

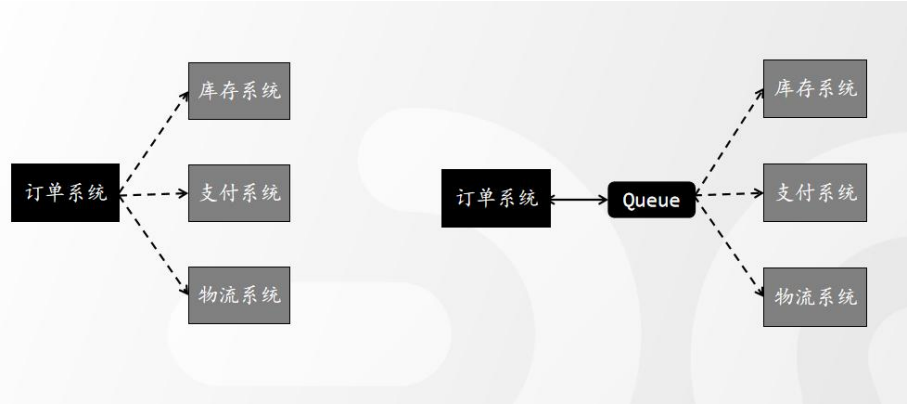
如果把串行改成并行，我们有什么思路?

（多线程）

多线程或者线程池是可以实现的，但是每一个需要并行执行的地方都引入线程，又会带来线程或者线程池的管理问题。

所以，这种情况下，我们可以引入 MQ 实现系统之间依赖关系的解耦合。

引入 MQ 以后:



订单系统只需要把退货的消息发送到消息队列上，由各个下游的业务系统自己创建队列，然后监听队列消费消息。

1、如果其他系统做了网络迁移，以前需要在订单系统配置和修改 IP、端口、接口地址，现在不需要，因为它不需要关心消费者在网络上的什么位置，只需要配置 MQ 服务器的地址。

2、如果某一个下游系统宕机了或者在停机升级，调用接口超时会导致订单系统业务逻辑失败。引入 MQ 以后没有任何影响，消息保存在 MQ 服务器，什么时候下游系统恢复了自己去消费就 OK 了。

3、假如下游业务系统运行正常，但是消费出了问题，比如改代码该出问题了或者数据库异常，生产者也不会受到影响。因为它只关心一件事情，就是消息有没有成功地发送到 MQ 服务器。

（如果要确保消费成功或者确保数据一致性肯定要靠其他的手段来解决，后面会说）

引入 MQ 以后，实现了系统之间依赖关系的解耦，系统的可扩展性和可维护性得到了提升。

3.3.3 实现流量削峰

第三个主要功能，是实现流量削峰。

在很多的电商系统里面，有一个瞬间流量达到峰值的情况，比如京东的 618，淘宝的双 11，还有小米抢购。普通的硬件服务器肯定支撑不了这种百万或者千万级别的并发量，就像 2012 年的小米一样，动不动服务器就崩溃。

如果通过堆硬件的方式去解决，那么在流量峰值过去以后就会出现巨大的资源浪费。那要怎么办呢？一个饭店，在国庆期间一天有几万个人想去吃饭，为什么它不马上扩张门店呢？它的实际做法是什么？首先肯定不

能拒绝顾客，说没位置了回家吧（排队啊）。

如果说要保护我们的应用服务器和数据库，限流也是可以的，但是这样又会导致订单的丢失，没有达到我们的目的。

为了解决这个问题，我们就可以引入 MQ。MQ 既然是 Queue，一定有队列的特性，我们知道队列的特性是什么?

（先进先出 FIFO）

先进先出，有一个排队的模型。这个饭店它就有排队的通道，不会因为想吃饭的人太多了就把饭店鸡煲或者把服务员累趴。而且它还可以线上排队，你都不用一直站着。

我们可以先把所有的流量承接下来，转换成 MQ 消息发送到消息队列服务器上，业务层就可以根据自己的消费速率去处理这些消息，处理之后再返回结果。

如果要处理快一点，大不了就是增加几个消费者。就像火车站在春运期间会多开几个窗口处理购票请求。

这个是我们利用 MQ 实现流量削峰的一个案例。

3.3.4 实现广播通信

实现一对多通信。以订单系统退货为例，如果新增了积分系统，需要获取订单状态变化信息，只需要增加队列监听就可以了，生产者没有任何代码修改。

比如你买了辆保时捷，如果要让村里所有人知道，只要跟大妈说一下就行了。

总结起来：

1） 对于数据量大或者处理耗时长的操作，我们可以引入 MQ 实现异步通信，减少客户端的等待，提

升响应速度，优化客户体验。

2） 对于改动影响大的系统之间，可以引入 MQ 实现解耦，减少系统之间的直接依赖，提升可维护性和可扩展性。

3） 对于会出现瞬间的流量峰值的系统，我们可以引入 MQ 实现流量削峰，达到保护应用和数据库的目的。

4） 一对多的广播通信。

如果大家的公司里面有用到 MQ 的话，也可以对号入座看看是起到了什么作用。

所以对于一些特定的业务场景，MQ 对于优化我们的系统性能还是有很大的帮助的。我们讲了这么多好处，以后只要是系统之间通信我全部用 MQ，不再考虑 HTTP 接口和 RPC 调用，可以吗?

肯定不行。下面我们分析一下使用 MQ 会带来的一些问题。

3.4 使用 MQ 带来的一些问题

打个简单的比方，以前你跟小王直接对话，现在改成了通过小黄传话给小王，肯定还是会有些问题出现的。

第一个就是运维成本的增加。既然要用 MQ，必须要分配资源部署 MQ，还要保证它时刻正常运行。

第二个是系统的可用性降低。原来是两个节点的通信，现在还需要独立运行一个服务。虽然一般的 MQ 都有很高的可靠性和低延迟的特性，但是一旦网络或者 MQ 服务器出现问题，就会导致请求失败，严重地影响业务。

第三个是系统复杂性提高。为什么说复杂？作为开发人员，要使用 MQ，首先必须要理解相关的模型和概念，才能正确地配置和使用 MQ。其次，使用 MQ 发送消息必须要考虑消息丢失和消息重复消费的问题。一旦消息没有被正确地消费，就会带来数据一致性的问题。

所以，我们在做系统架构，选择通信方式的时候一定要根据实际情况来分析，不要因为我们说了这么多的 MQ 能解决的问题，就盲目地引入 MQ。

4、消息协议

协议是计算机之间通信时共同遵从的一组约定，都遵守相同的约定，计算机之间才能相互交流。是对数据格式和计算机之间交换数据时必须遵守的规则的正式描述。

协议三要素：

语法：即数据与控制信息的结构或格式；

语义：即需要发出何种控制信息，完成何种动作以及做出何种响应；

时序（同步）：即事件实现顺序的详细说明

4.1 AMQP 协议

AMQP（Advanced Message Queuing Protocol）是高级消息队列协议 04 年 JPMorgan Chase(摩根大通集团)联合其他公司共同设计。

特性：事务支持、持久化支持，出生金融行业，在可靠性消息处理上具备天然的优势。

应用：RabbitMQ、ActiveMQ、OpenAMQ、Apache Qpid、Redhat Enterprise MRG、AMQP Infrastructure、ØMQ、Zyre。

4.2 MQTT 协议

MQTT（Message Queuing Telemetry Transport）消息队列遥测传输是 IBM 开发的一个即时通讯协议，物联网系统架构中的重要组成部分。

特性：轻量、结构简单、传输快、没有事务支持、没有持久化相关设计。

应用：RabbitMQ、ActiveMQ

4.3 Open Message 协议

Open Messaging 是近一两年由阿里发起，与雅虎、滴滴出行、Streamlio 等公司共同参与创立的分布式消息中间件、流处理领域的应用开发标准。

是国内首个在全球范围内发起的分布式消息领域国际标准

特性：结构简单、解析快、有事务设计、有持久化设计

应用：RocketMQ

4.4 Kafka 协议

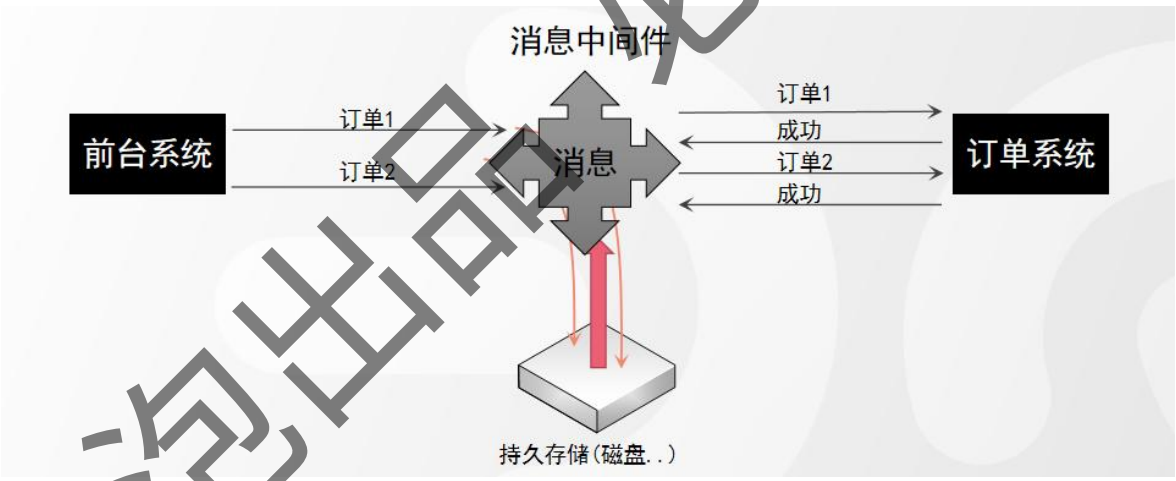
Kafka 协议是基于 TCP 的二进制协议。消息内部是通过长度来分隔，由一些基本数据类型组成。

特性：结构简单、解析快、无事务设计、有持久化设计

应用：Kafka

5、持久化

简单来说就是将数据存入磁盘，而不是存在内存中随服务重启而消失，使数据能够永久保存叫做持久化。



常用的消息中间件对持久化的支持

	ActiveMQ	RabbitMQ	Kafka	RocketMQ
文件系统	支持	支持	支持	支持
数据库	支持	/	/	/

6、消息分发策略

	ActiveMQ	RabbitMQ	Kafka	RocketMQ
发布订阅	支持	支持	支持	支持
轮询分发	支持	支持	支持	/
公平分发	/	支持	支持	/
重发	支持	支持	/	支持
消息拉取	/	支持	支持	支持