一、谈谈业务中使用分布式的场景

分布式主要是为了提供可扩展性以及高可用性，业务中使用分布式的场景主要有分布式存储以及分布式计算。

分布式存储中可以将数据分片到多个节点上，不仅可以提高性能（可扩展性），同时也可以使用多个节点对同一份数据进行备份（高可用性）。

至于分布式计算，就是将一个大的计算任务分解成小任务分配到多个节点上去执行，再汇总每个小任务的执行结果得到最终结果。MapReduce 是分布式计算最好的例子。

二、分布式事务

指事务的操作位于不同的节点上，需要保证事务的 AICD 特性。

产生原因

- 数据库分库分表；
- SOA 架构，比如一个电商网站将订单业务和库存业务分离出来放到不同的节点上。

应用场景

- 下单：减少库存、更新订单状态。库存和订单如果不在同一个数据库，就涉及分布式事务。
- 支付：买家账户扣款、卖家账户入账。买家和卖家账户信息如果不在同一个数据库，就涉及分布式事务。

解决方案

1. 两阶段提交协议

> 两阶段提交

两阶段提交协议可以很好地解决分布式事务问题。它可以使用 XA 来实现，XA 包含两个部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如 Oracle、DB2 这些商业数据库都实现了 XA 接口；而事务管理器作为全局的协调者，负责各个本地资源的提交和回滚。

2. 消息中间件

消息中间件也可称作消息系统 (MQ)，它本质上是一个暂存转发消息的一个中间件。在分布式应用当中，我们可以把一个业务操作转换成一个消息，比如支付宝的余额转入余额宝操作，支付宝系统执行减少余额操作之后向消息系统发送一个消息，余额宝系统订阅这条消息然后进行增加余额宝操作。

2.1 消息处理模型

（一）消息队列

（二）发布/订阅

2.2 消息的可靠性

（一）发送端的可靠性

发送端完成操作后一定能将消息成功发送到消息系统。

实现方法：在本地数据库建一张消息表，将消息数据与业务数据保存在同一数据库实例里，这样就可以利用本地数据库的事务机制。事务提交成功后，将消息表中的消息转移到消息中间件，若转移消息成功则删除消息表中的数据，否则继续重传。

（二）接收端的可靠性

接收端能够从消息中间件成功消费一次消息。

实现方法：

- 保证接收端处理消息的业务逻辑具有幂等性：只要具有幂等性，那么消费多少次消息，最后处理的结果都是一样的。
- 保证消息具有唯一编号，并使用一张日志表来记录已经消费的消息编号。

三、负载均衡的算法与实现

算法

1. 轮询（Round Robin）

轮询算法把每个请求轮流发送到每个服务器上。下图中，一共有 6 个客户端产生了 6 个请求，这 6 个请求按 (1, 2, 3, 4, 5, 6) 的顺序发送。最后，(1, 3, 5) 的请求会被发送到服务器 1，(2, 4, 6) 的请求会被发送到服务器 2。

该算法比较适合每个服务器的性能差不多的场景，如果有性能存在差异的情况下，那么性能较差的服务器可能无法承担过大的负载（下图的 Server 2）。

2. 加权轮询（Weighted Round Robbin）

加权轮询是在轮询的基础上，根据服务器的性能差异，为服务器赋予一定的权值。例如下图中，服务器 1 被赋予的权值为 5，服务器 2 被赋予的权值为 1，那么 (1, 2, 3, 4, 5) 请求会被发送到服务器 1，(6) 请求会被发送到服务器 2。

3. 最少连接（least Connections）

由于每个请求的连接时间不一样，使用轮询或者加权轮询算法的话，可能会让一台服务器当前连接数过大，而另一台服务器的连接过小，造成负载不均衡。例如下图中，(1, 3, 5) 请求会被发送到服务器 1，但是 (1, 3) 很快就断开连接，此时只有 (5) 请求连接服务器 1；(2, 4, 6) 请求被发送到服务器 2，只有 (2) 的连接断开。该系统继续运行时，服务器 2 会承担过大的负载。

最少连接算法就是将请求发送给当前最少连接数的服务器上。例如下图中，服务器 1 当前连接数最小，那么新到来的请求 6 就会被发送到服务器 1 上。

4. 加权最少连接（Weighted Least Connection）

在最少连接的基础上，根据服务器的性能为每台服务器分配权重，再根据权重计算出每台服务器能处理的连接数。

5. 随机算法（Random）

把请求随机发送到服务器上。和轮询算法类似，该算法比较适合服务器性能差不多的场景。

6. 源地址哈希法 (IP Hash)

源地址哈希通过对客户端 IP 哈希计算得到的一个数值，用该数值对服务器数量进行取模运算，取模结果便是目标服务器的序号。

- 优点：保证同一 IP 的客户端都会被 hash 到同一台服务器上。
- 缺点：不利于集群扩展，后台服务器数量变更都会影响 hash 结果。可以采用一致性 Hash 改进。

实现

1. HTTP 重定向

HTTP 重定向负载均衡服务器收到 HTTP 请求之后会返回服务器的地址，并将该地址写入 HTTP 重定向响应中返回给浏览器，浏览器收到后需要再次发送请求。

缺点：

- 用户访问的延迟会增加；
- 如果负载均衡器宕机，就无法访问该站点。

2. DNS 重定向

使用 DNS 作为负载均衡器，根据负载情况返回不同服务器的 IP 地址。大型网站基本使用了这种方式做为第一级负载均衡手段，然后在内部使用其它方式做第二级负载均衡。

缺点：

- DNS 查找表可能会被客户端缓存起来，那么之后的所有请求都会被重定向到同一个服务器。

3. 修改 MAC 地址

使用 LVS（Linux Virtual Server）这种链路层负载均衡器，根据负载情况修改请求的 MAC 地址。

4. 修改 IP 地址

在网络层修改请求的目的 IP 地址。

5. 代理自动配置

正向代理与反向代理的区别：

- 正向代理：发生在客户端，是由用户主动发起的。比如翻墙，客户端通过主动访问代理服务器，让代理服务器获得需要的外网数据，然后转发回客户端。
- 反向代理：发生在服务器端，用户不知道代理的存在。

PAC 服务器是用来判断一个请求是否要经过代理。

四、分布式锁

Java 提供了两种内置的锁的实现，一种是由 JVM 实现的 synchronized 和 JDK 提供的 Lock，对于单机单进程应用，可以使用它们来实现锁。当应用涉及到多机、多进程共同完成时，那么这时候就需要一个全局锁来实现多个进程之间的同步。

使用场景

在服务器端使用分布式部署的情况下，一个服务可能分布在不同的节点上，比如订单服务分布在节点 A 和节点 B 上。如果多个客户端同时对一个服务进行请求时，就需要使用分布式锁。例如一个服务可以使用 APP 端或者 Web 端进行访问，如果一个用户同时使用 APP 端和 Web 端访问该服务，并且 APP 端的请求路由到了节点 A，WEB 端的请求被路由到了节点 B，这时候就需要使用分布式锁来进行同步。

实现方式

1. 数据库分布式锁

（一）基于 MySQL 锁表

该实现完全依靠数据库的唯一索引。当想要获得锁时，就向数据库中插入一条记录，释放锁时就删除这条记录。如果记录具有唯一索引，就不会同时插入同一条记录。

这种方式存在以下几个问题：

- 锁没有失效时间，解锁失败会导致死锁，其他线程无法再获得锁。
- 只能是非阻塞锁，插入失败直接就报错了，无法重试。
- 不可重入，同一线程在没有释放锁之前无法再获得锁。

（二）采用乐观锁增加版本号

根据版本号来判断更新之前有没有其他线程更新过，如果被更新过，则获取锁失败。

2. Redis 分布式锁

（一）基于 SETNX、EXPIRE

使用 SETNX（set if not exist）命令插入一个键值对时，如果 Key 已经存在，那么会返回 False，否则插入成功并返回 True。因此客户端在尝试获得锁时，先使用 SETNX 向 Redis 中插入一个记录，如果返回 True 表示获得锁，返回 False 表示已经有客户端占用锁。

EXPIRE 可以为一个键值对设置一个过期时间，从而避免了死锁的发生。

（二）RedLock 算法

RedLock 算法使用了多个 Redis 实例来实现分布式锁，这是为了保证在发生单点故障时仍然可用。

- 尝试从 N 个相互独立 Redis 实例获取锁，如果一个实例不可用，应该尽快尝试下一个。
- 计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且从大多数（N/2+1）实例上获取了锁，那么就认为锁获取成功了。
- 如果锁获取失败，会到每个实例上释放锁。

3. Zookeeper 分布式锁

Zookeeper 是一个为分布式应用提供一致性服务的软件，例如配置管理、分布式协同以及命名的中心化等，这些都是分布式系统中非常底层而且是必不可少的基本功能，但是如果自己实现这些功能而且要达到高吞吐、低延迟同时还要保持一致性和可用性，实际上非常困难。

（一）抽象模型

Zookeeper 提供了一种树形结构级的命名空间，/app1/p_1 节点表示它的父节点为 /app1。

（二）节点类型

- 永久节点：不会因为会话结束或者超时而消失；
- 临时节点：如果会话结束或者超时就会消失；
- 有序节点：会在节点名的后面加一个数字后缀，并且是有序的，例如生成的有序节点为 /lock/node-0000000000，它的下一个有序节点则为 /lock/node-0000000001，依次类推。

（三）监听器

为一个节点注册监听器，在节点状态发生改变时，会给客户端发送消息。

（四）分布式锁实现

- 创建一个锁目录 /lock；
- 在 /lock 下创建临时的且有序的子节点，第一个客户端对应的子节点为 /lock/lock-0000000000，第二个为 /lock/lock-0000000001，以此类推；
- 客户端获取 /lock 下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听自己的前一个子节点，获得子节点的变更通知后重复此步骤直至获得锁；
- 执行业务代码，完成后，删除对应的子节点。

（五）会话超时

如果一个已经获得锁的会话超时了，因为创建的是临时节点，因此该会话对应的临时节点会被删除，其它会话就可以获得锁了。可以看到，Zookeeper 分布式锁不会出现数据库分布式锁的死锁问题。

（六）羊群效应

一个节点未获得锁，需要监听自己的前一个子节点，这是因为如果监听所有的子节点，那么任意一个子节点状态改变，其它所有子节点都会收到通知（羊群效应），而我们只希望它的后一个子节点收到通知。

五、分布式 Session

在分布式场景下，一个用户的 Session 如果只存储在一个服务器上，那么当负载均衡器把用户的下一个请求转发到另一个服务器上，该服务器没有用户的 Session，就可能导致用户需要重新进行登录等操作。

1. Sticky Sessions

需要配置负载均衡器，使得一个用户的所有请求都路由到一个服务器节点上，这样就可以把用户的 Session 存放在该服务器节点中。

缺点：当服务器节点宕机时，将丢失该服务器节点上的所有 Session。

2. Session Replication

在服务器节点之间进行 Session 同步操作，这样的话用户可以访问任何一个服务器节点。

缺点：需要更好的服务器硬件条件；需要对服务器进行配置。

3. Persistent DataStore

将 Session 信息持久化到一个数据库中。

缺点：有可能需要去实现存取 Session 的代码。

4. In-Memory DataStore

可以使用 Redis 和 Memcached 这种内存型数据库对 Session 进行存储，可以大大提高 Session 的读写效率。内存型数据库同样可以持久化数据到磁盘中来保证数据的安全性。

参考资料

- Comparing Load Balancing Algorithms
- 负载均衡算法及手段
- Redirection and Load Balancing
- Session Management using Spring Session with JDBC DataStore
- Apache Wicket User Guide - Reference Documentation
- 集群/分布式环境下 5 种 Session 处理策略
- 浅谈分布式锁
- 深入理解分布式事务
- 分布式系统的事务处理
- 关于分布式事务
- 基于 Zookeeper 的分布式锁