## ​​​ 六、分布式

### 1、关于分布式的一些基本概念

分布式系统、分布式技术

**zookeeper、分布式事务、分布式锁、负载均衡**

#### 1、分布式

小明的公司有3个系统：系统A，系统B和系统C，这三个系统所做的业务不同，被部署在3个独立的机器上运行，他们之间互相调用（当然是跨域网络的），通力合作完成公司的业务流程。

将不同的业务分部在不同的地方，就构成了一个分布式的系统，现在问题来了，系统A是整个分布式系统的脸面，用户直接访问，用户访问量大的时候要么是速度巨慢，要么直接挂掉，怎么办？

由于系统A只有一份，所以会引起单点失败。。。

#### 2、集群（Cluster）

小明的公司不差钱，就多买几台机器吧，小明把系统A一下子部署了好几份（例如下图的3个服务器），每一份都是系统A的一个实例，对外提供同样的服务，这样，就不怕其中一个坏掉了，还有另外两个呢。

这三个服务器的系统就组成了一个集群。

可是对用户来说，一下子出现这么多系统A，每个系统的IP地址都不一样，到底访问哪一个呢？

如果所有人都访问服务器1.1，那服务器1.1会被累死，剩下两个闲死，成了浪费钱的摆设

#### 3、负载均衡（Load Balancer）

小明要尽可能的让3个机器上的系统A工作均衡一些，比如有3万个请求，那就让3个服务器各处理1万个（理想情况），这叫负载均衡

很明显，这个负载均衡的工作最好独立出来，放到独立的服务器上（例如nginx）：

#### 4、更具体的分布式问题

##### 一、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。例如在下单场景下，库存和订单如果不在同一个节点上，就需要涉及分布式事务。

本地消息

原理

本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性。

在分布式事务操作的一方，它完成写业务数据的操作之后向本地消息表发送一个消息，本地事务能保证这个消息一定会被写入本地消息表中。之后将本地消息表中的消息转发到 Kafka 等消息队列（MQ）中，如果转发成功则将消息从本地消息表中删除，否则继续重新转发。在分布式事务操作的另一方从消息队列中读取一个消息，并执行消息中的操作。

分析

本地消息表利用了本地事务来实现分布式事务，并且使用了消息队列来保证最终一致性。

两阶段提交协议2PC

##### 二、分布式锁

可以使用 Java 提供的内置锁来实现进程同步：由 JVM 实现的 synchronized 和 JDK 提供的 Lock。但是在分布式场景下，需要同步的进程可能位于不同的节点上，那么就需要使用分布式锁来同步。

原理锁可以有阻塞锁和乐观锁两种实现方式，这里主要探讨阻塞锁实现。阻塞锁通常使用互斥量来实现，互斥量为 1 表示有其它进程在使用锁，此时处于锁定状态，互斥量为 0 表示未锁定状态。1 和 0 可以用一个整型值来存储，也可以用某个数据存在或者不存在来存储，某个数据存在表示互斥量为 1，也就是锁定状态。

实现

数据库的唯一索引

当想要获得锁时，就向表中插入一条记录，释放锁时就删除这条记录。唯一索引可以保证该记录只被插入一次，那么就可以用这个记录是否存在来判断是否存于锁定状态。

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。

Redis 的 SETNX 指令

使用 SETNX（set if not exist）指令插入一个键值对，如果 Key 已经存在，那么会返回 False，否则插入成功并返回 True。

SETNX 指令和数据库的唯一索引类似，可以保证只存在一个 Key 的键值对，可以用一个 Key 的键值对是否存在来判断是否存于锁定状态。

EXPIRE 指令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。

Redis 的 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。

Zookeeper 的有序节点

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，所以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的唯一索引实现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。

##### 三、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。

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。

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。

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。

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。

##### 四、负载均衡

算法

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）。

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。

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。

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。

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。

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

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。

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。

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。

修改 IP 地址

在网络层修改请求的目的 IP 地址。

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。

后来小明发现，这个负载均衡的服务器虽然工作内容简单，就是拿到请求，分发请求，但是它还是有可能挂掉，单点失败还是会出现。

没办法，只好把负载均衡也搞成一个集群，bug和系统A的集群有两点不同：

1.这个新的集群中虽然有两个机器，但是我们可以用某种办法，让这个机器对外只提供一个IP地址，也就是用户看到的好像只有一个机器。

2.同一时刻，我们只让一个负载均衡的机器工作，另外一个原地待命，如果工作的那个拐到了，待命的那个就顶上去。

4、弹性

如果3个系统A的实例还是满足不了大量请求，例如双十一，可以申请增加服务器，双十一过后，新增的服务器闲置，成了摆设，于是小明决定尝试云计算，在云端可以轻松的创建，删除虚拟的服务器，那样就可以轻松的随着用户的请求动图的增减服务器了。

## 5、失效转移

上面的系统看起来很美好，但是做了一个不切实际的假设：

所有的服务都是无状态的，换句话说，假设用户的两次请求直接是没有关联的。

但是现实是，大部分服务都是有状态的，例如购物车。

用户访问系统，在服务器上创建了一个购物车，并向其中加了几个商品，然后服务器1.1挂掉了，用户后续访问就找不到服务器1.1了，这时候就要做失效转移，让另外几个服务器去接管，去处理用户的请求。

可是问题来了，在服务器1.2,1.3上有用户的购物车吗？如果没有，用户就会抱怨，我刚创建的购物车哪里去了？

还有更严重的，假设用户登录过得信息保存到了该服务器1.1上登录的，用户登录过的信息保存到了该服务器的session中，现在这个服务器挂了，用的session就不见了，会把用户踢到了登录界面，让用户再次登录！

处理不好状态的问题，集群的威力就大打折扣，无法完成真正的失效转移，甚至无法使用。

### 2、springboot与spring相比优势是什么？

一、SSM优缺点应该分开来说的，比如

1）spring 不说了，核心ioc、aop技术，ioc解耦，使得代码复用，可维护性大幅度提升，aop提供切面编程，同样的增强了生产力。

2）spring mvc嘛，是对比struts2等mvc框架来说的，不说struts2爆出的那么多安全漏洞，而且是类拦截，所有Action变量共享，同时是filter入口的，而spring mvc是方法拦截，controller独享request response数据，采用的serlvet入口，与spring无缝对接。开发而言，spring mvc更加轻量和低入门。

3）mybatis嘛，看业务场景，主要是mybatis的sql可以由开发者去掌控和调优，相对hibernate等orm框架来说，更加直观。在业务场景比较复杂，sql好多联合关联的情况下，mybatis谁用谁知道。当然缺点就是对sql不熟悉的开发者就不太友好了。

二、 SSM框架和spring boot全家桶相比有哪些优缺点？ 这两者对比起来有点奇怪。因为SSM是WEB应用框架，涵盖整个应用层，而spring boot你可以看做一个启动、配置、快速开发的辅助框架，本身针对的是微服务。

springboot 只是为了提高开发效率，是为了提升生产力的：

1、springboot一个应用是一个可执行jar（启动类main方法启动web应用），而不像传统的war，内嵌tomcat容器，可以jar形式启动一个服务，可以快速部署发布web服务，微服务最好不过了。

2、将原有的xml配置，简化为java配置

3、当然结构可能跟一般的ssm有一定区别，但其实主要是在资源文件。

Spring Boot 默认“约定”从资源目录的这些子目录读取静态资源：

src/main/resources/META-INF/resources

src/main/resources/static （推荐）

src/main/resources/public

### 3、springboot自动配置的原理是什么？

SpringBoot的核心就是自动配置，自动配置又是基于条件判断来配置Bean。

### 4、分布式框架有哪些？

### 5、dubbo和spring cloud的比较；

### 6、Restful风格的接口的理解；

### 7、rpc框架有哪些？如何使用？

### 8、rpc和http的对比；

### 9、负载均衡的理解；

### 10、常用的mq的种类；（补充）

**11、RabbitMQ的作用？**