分布式计算-习题2

院(系):智能工程学院智能科学与技术　学号：22354189　姓名：张瑞程

1. 什么是网络分区？如何判断是否发生了网络分区？网络分区出现概率较高的场景是什么？有哪些常见的处理方法？

（1）什么是网络分区?

分布式集群中，一个大的网络被分隔成两个或更多的子网络。节点之间由于网络不通，导致集群中节点形成不同子集。子集内部各节点网络连通，而子集之间网络不连通。

（2）如何判断是否发生了网络分区？

判断网络分区是否发生于分布式系统架构密切相关，不同架构（集中式分布式集群与非集中式分布式集群）中各节点功能不同，因此分区有不同的判断方式。

• 集中式分布式集群

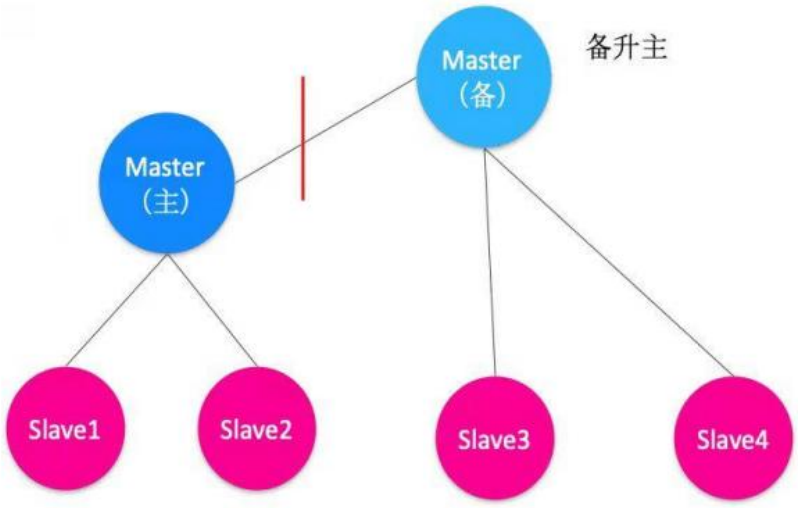

图 1 集中式分布式集群

在集中式架构中，通常有一个中心节点（主 master）或有几个备用中心节点负责协调和管理整个系统。所有的从节点（slave）与中心节点进行直接通信，而不直接与其他 slave 节点通信。如果发生了网络分区，则可能导致 slave 节点与中心节点的连接中断，使得 slave 节点无法协调和通信。

在这种情况下，可以采用**心跳机制**进行判断，即通过监测与中心节点的连接状态来判断网络分区：主节点判断来自从节点的心跳是否超时。如果某个从节点超时，系统可以将其归

类到一个子集，并判断发生了网络分区。

- **非集中式分布式集群**

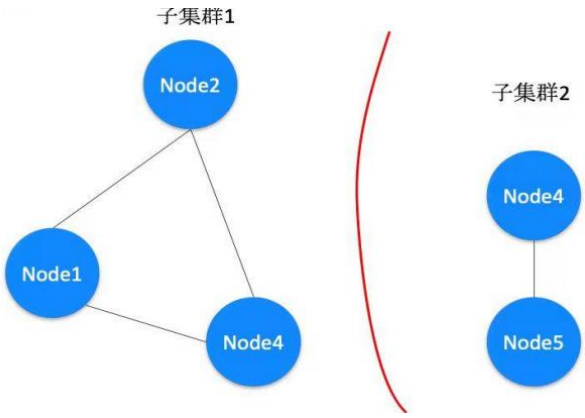

图 2 非分布式架构网络分区

在非集中式架构中，所有节点彼此平等，相互通信并共同协作。如果发生了网络分区，可能导致一部分节点无法与另一部分节点通信。判断网络分区的方式可能涉及到多个节点之间的通信状态。在这种情况下，同样可以采用心跳机制，但是所有节点之间进行相互检测。**当某个节点在一定时间内未收到来自其他节点的心跳响应，系统可以认为该节点可能发生了分区或故障。此外，消息通信的异常可以被视为网络分区的指标之一，当节点之间的消息丢失或延迟较大时，可能表明存在网络分区。**

（3）网络分区出现概率较高的场景是什么？

多数据中心部署： 分布在不同网络的数据中心之间，通常容易发生通信时延、网络不稳定性、故障容易发生或带宽受限等问题，导致容易出现网络分区。

高并发导致网络拥塞分区： 在高并发的情况下，集群中的通信量可能快速增加，当通信量超出系统设计的带宽限制时，可能导致网络拥塞，从而导致节点间的通信无法及时响应，进而引发网络分区。

（4）有哪些常见的网络分区处理方法?

基本方法有激进方式与保守方式两种。

- 激进方式指：一旦发现节点不可达，将该节点从集群剔除，并重新选主。

该方式存在问题：若分区是因为网络问题引起，那么会出现双主问题。

- 保守方式指：一旦发现节点不可达，则停止自己的服务。

存在问题：若所有分区都采取保守措施，则会导致整个系统停止服务。

因此，出现了一系列**均衡策略**：

• static quorum 策略：

该方法在系统启动前，设置固定票数。当分区后，分区中节点数不小于这个票数，则该分区为活动分区。为了避免出现多个活动分区导致双主或多主问题，应使固定票数满足以下要求：

$$\begin{cases} 固定票数 \leq 总票数 \\ 固定票数 > \dfrac{总票数}{2} \end{cases}$$

该策略同样存在问题：

①当出现多分区时，可能不存在满足票数的分区

②当集群中有节点加入时，固定票数可能不再适用。

• keep majority 策略：

该策略在出现网络分区时，保留集群中具有大多数节点的子集群。即

$$保留子集群\ i \mid \mathrm{num(i)} \geq \frac{n}{2}, n\ 为集群总节点数$$

为了避免出现双主问题，通常使得集群总节点 n 为奇数。该策略可以解决动态节点加入问题，但是不适用于产生多分区的场景：当出现多分区时，很难找到一个节点数满足要求的分区。

• 设置仲裁机制：

该机制引入第三方组件/节点作为仲裁者。仲裁者与集群中所有节点进行连接。所有节点向他上报心跳。出现分区时，仲裁者可以根据预先设定的规则或算法，决定保留具有最佳条件或最大节点数的子集群，从而保持集群的稳定性和一致性。

• 基于共享资源的方式：

设置共享资源锁并进行轮转。哪个子集群获得共享资源锁就保留子集群。

问题：获取锁的节点发生故障但未释放锁，则会导致其他子群不可用。

因此，获取锁的节点的可靠性需要被保证。

· 数据冗余：

在多个地理位置部署冗余数据，确保即使发生网络分区，仍能提供服务。

2. 面对大规模的请求，如何实现负载均衡？如果要考虑请求所需资源不同的情形，我们应该如何设计负载均衡策略呢？（提示：一致性哈希与相关优化方法）

负载均衡指在多个服务器之间分配网络流量，以确保没有任何一个服务器负载过重。负载均衡方法有**（加权）轮询策略、随机策略，**最常见的是**哈希**和**一致性哈希**

（1）轮询或加权轮询策略：按照节点顺序轮流分配请求。加权轮询下，每个节点被设置优先级，每次请求优先选取优先级最高的服务器处理。

（2）随机策略：用户请求到来时，利用随机函数随机发送到某个节点进行处理。

（3）哈希：确定一个哈希函数，根据请求的来源和/或目标的 IP 地址的哈希结果来分配请求。

优点：

①路由一致性：确保来自特定 IP 地址的所有请求都会被路由到同一台服务器上。

②数据均匀性：哈希函数设计得当时，可以保证很好的数据均匀性。

缺点：

①可扩展性差：哈希方法通常采用固定的哈希规则来确定数据分布到特定节点，这种静态的分配方式使得系统难以适应动态变化的负载情况；当系统需要扩展节点时，由于哈希算法的固定性，新增节点的引入可能导致大量数据迁移，影响系统性能。

②容错性差:在使用哈希方法进行负载均衡时，一旦某个节点发生故障或失效，其上的数据将需要重新分布。这种重新分布可能导致系统出现临时性的负载不均衡，影响整体稳定性；如果某个节点上的数据量较大，失效可能导致系统中的大量请求需要重新路由，增加了系统的复杂性和容错的难度。

③数据重新计算：当系统中的节点发生变化时，需要重新计算并移动数据至新的节点，计算开销大。

（2）一致性哈希：该方法改进了哈希的可扩展性、容错性。当机器增加或减少时，节点数据迁移仅限于两个集群节点之间，不会造成全局问题。实现步骤如下：

①构建环形哈希空间：根据哈希算法把散列码映射到一定范围$(0,2^{23}-1)$，成首尾相接的环。

②映射数据和集群节点到环上：使用哈希算法将集群节点和用户请求都映射到环的不同位置。

③数据储存与用户请求定位：当有用户请求时，通过哈希算法计算用户的哈希值，确定

其在环上的位置。然后，沿着环的顺时针方向找到第一个不小于用户哈希值的节点位置。将用户请求绑定到该节点上，实现了请求到节点的映射，从而实现了负载均衡。

·考虑请求所需资源不同的情形，我们应该如何设计负载均衡策略？

在上文的答案中，已经说明了哈希算法是基于哈希函数的，并未考虑资源权重。而加权轮询算法是考虑资源权重后的随机轮询算法；结合两种算法的思想，设计**基于请求资源权重的一致性哈希算法**，步骤如下：

①确定服务器与请求的资源权重：为每个服务器分配一个资源权重S_k，权重越高表示服务器具有**更强的处理能力**。确定请求的资源权重 N，权重越高表示该请求所需资源量越大。

②构建一致性哈希环：将所有服务器节点映射到一致性哈希环上。

③计算请求的哈希值：每个请求通过哈希函数计算出一个哈希值。

④查找节点：根据请求的哈希值沿着一致性哈希环顺时针或逆时针遍历，按顺序取出前 i 个节点。计算第 k 个节点的“加权得分”：

$$\text{Score} = \left|\frac{\text{N}}{\text{Source\_w}*\text{S}_\text{k}} - 1\right| + \frac{1}{\text{k}} * \text{distance\_w},$$

其中 distance_w 与 Source_w 为资源和路程在最后权重中占有的权重系数。公式第一项考虑了资源请求量与服务器处理能力的匹配程度，当请求量与能力匹配时，该项为 0；第二项考虑了服务器节点距离当前请求在哈希环上的距离。因此该公式可以综合“距离最近”与“服务器处理能力与请求资源相匹配”两种考虑。

⑤将请求分发到目标服务器：将请求发送到选定的目标服务器，完成负载均衡的过程。

通过结合资源权重和一致性哈希算法，可以实现更精细的负载均衡。资源权重决定了请求在选择目标服务器时的倾向性，而一致性哈希算法保持了负载均衡的稳定性和扩展性。同时，通过在哈希中引入虚拟节点、跟据服务器性能权重分配、跟据负载情况动态调整权重等优化方法，能够进一步提高系统的性能和容错性。

3. 简要介绍 Raft 和 Zookeeper 的选举机制（ZAB），比较他们在投票规则方面的区别，并分别阐述他们是如何保证日志同步的。在 Raft 选举过程中，如果 follower 与 leader 的日志长度差异很大，按照 raft 论文中的同步方式耗时较长，请给出改进策略。

①选举机制介绍：

• Raft 协议：

每一个节点设置一个定时器，当一个节点在超时时间没收到来自 leader 的心跳，则节点成为 candidate 并发起选举。当一个 Candidate 发起选举时，它会向其它节点发送 RequestVote RPCs。如果接收到的**有效票数**超过集群半数，那么该节点就会成为 Leader。如果期间收到有效 leader 的心跳，则节点返回到 follower 状态；如果超时得不到足够票数，则重新开始选举过程。

• Zookeeper：

Zookeeper 采用基于 TCP 的 FastleaderElection 机制。每个集群服务器都存在四种状态：looking，leading，following，observing。选举时，服务器转换到 Looking 状态，每个集群服务器节点生成包含递增事务号 zxid 的选票，然后将选票广播到所有服务器；每个服务器在每一轮中选择自己最新的选票，根据选票事务号进行比较，具有最新事务号的服务器成为领导者。若同时有两个服务器具有相同的选票事务号，则进入选票 PK 阶段。

②投票规则区别：

• Raft 协议：

在 Raft 协议中，候选者（Candidate）发起选举时，只有当接收者的任期号不比候选者的任期号更新时，接收者才会投票给候选者。**每个节点只能投一票**。此外，在同一**个任期内，最多只有**一个候选者能够获得**超过半数**的票数成为领导者。

• ZooKeeper：

在 ZooKeeper 中，每个节点在投票时会考虑两个关键因素：ZooKeeper 的逻辑时钟（ZXID）和服务器的 ID。首先，会选择具有最大 ZXID 的节点，如果 ZXID 相同，那么会选择服务器 ID 最大的节点。**节点通过两阶段提交投票来选举 Leader。在第一阶段，节点请求投票，第二阶段是收到其他节点的 ACK。**

区别： 在投票规则方面，Raft 主要关注的是任期号，而 ZooKeeper 考虑的是 ZXID 和服务器 ID。在 Raft 中，接收者仅在其日志不比候选者的日志更新时才投票给候选者。而在 ZooKeeper 中，首先选择具有最大 ZXID 的节点，如果 ZXID 相同，则选择服务器 ID 最大的节点。

③日志同步：

• Raft 协议：

A. Leader 负责接收客户端请求并将其转换为日志条目，并将该日志条目追加到自己的日志中。

B. Leader 将这个日志条目通过心跳消息广播给其他节点。当 Leader 发现大多数节点都已经复制了这个日志条目时，它就会将该日志条目标记为已提交 committed 状态，并向其他节点发送 Commit 指令。

C. 其他节点在收到 Commit 指令后，也会将该日志条目标记为已提交，并执行相应的保存更新操作。

通过以上机制，Leader 确保所有节点的日志保持一致，同时 Leader 的日志条目会被追加到大多数节点上，确保了日志的同步性。

• Zookeeper：

A. Leader 负责处理客户端请求并生成全局唯一的递增提案编号(zxid)。Leader 将生成的提案广播给其他节点。这些提案包含了客户端的请求以及 ZXID。

B. 当大多数节点确认接收到提案后，Leader 发送 Commit 指令，要求节点将这个提案应用到它们的状态机中。

C. 在应用过程中，提案的 ZXID 用于确保节点应用提案的顺序。节点会按照 ZXID 的递增顺序应用提案，从而保证所有节点的状态机按照相同的顺序执行相同的操作。

D. 当 Leader 被选出后，它会从最新的状态(ZXID)开始执行，并将这个状态通过 ZAB 协议发送给其它节点。其它节点在接收到这个状态后，会更新它们的状态。如果一个节点的状态落后于 Leader，那么它会从 Leader 那里获取最新的状态，从而保证了日志同步。

④改进策略：

Raft 协议中，日志复制是通过 Leader 将其日志条目一条一条地复制到 Follower 的方式来实现的。如果 Follower 与 Leader 的日志长度差异很大，会造成网络传输和复制的延迟和耗时。可考虑采取快照机制定时同步，以避免这种情况

- **快照机制：**

①Leader 定期生成集群的快照，包含当前的状态以及日志的快照。

②当 Follower 需要进行同步时，它可以向 Leader 请求最新的快照。

③Leader 将最新的快照发送给 Follower，Follower 可以直接应用这个快照，而不是逐条复制日志。

这种方式可以有效减少同步所需的传输数据量,特别是在 Follower 日志差异较大的情况下，通过传输整个快照而非逐条日志可以显著提高同步速度。

- **增量复制：**使用增量复制的方式，只传递 Follower 缺失的部分日志。Leader 可以将新增的日志条目发送给 Follower，而不是整个日志。这可以通过记录 Follower 已经拥有的最后一个共享的日志索引，然后只发送从这个索引之后的新增日志。

4. 简要介绍常见的流量控制的处理方法（提示：4 种）。什么是拥塞控制？它与流量控制的区别是什么？介绍服务熔断和降级的概念以及它们的使用场景。

（1）常见流量控制的处理方法：

①令牌桶策略：在令牌桶算法中，令牌以固定的速率被放入令牌桶中。当一个请求到达时，它需要从令牌桶中获取一个令牌才能被处理。如果令牌桶为空，请求可能需要等待，或者被丢弃。

②漏桶策略：漏桶算法是另一种常见的流量控制方法。在漏桶算法中，数据包被视为流入一个桶的水，而桶底的漏洞以固定的速率流出水（服务器以固定速率处理输入请求）。如果桶满了，那么新流入的水（数据包）会被丢弃。漏桶算法能够平滑网络流量，防止突发流量的发生。

③消息队列：在这种方法中，生产者维护一个待发送消息的队列，称为发送队列，而消费者维护一个接收消息的队列，称为接收队列。生产者发送消息到队列时，必须等待消费者的确认信息，一旦收到确认，发送队列才向后移动，允许生产者发送新的消息。

④滑动窗口：该方法中，发送方维持一个连续的，合理尺寸的队列，称为发送窗口；接收方也维持一个类似的接受窗口。发送方给接收方发数据后，必须收到接收方返回的确认信息，发送窗口才向后移动，发送新的数据。

（2）拥塞控制：

拥塞控制通过实时监测网络状况，灵活调整数据的传输速率，以防止网络中数据过度堆积，从而避免导致传输数据困难。

（3）与流量控制的区别：

流量控制是为了控制发送方与接收方之间的数据传输速率，以避免接收方因无法处理过多数据而溢出。网络传输中流量控制通过采用滑动窗口等机制调整发送方和接收方的数据处理速度；而**拥塞控制**则是为了防止网络中的拥塞，通过实时监测网络状态，及时疏导网络流量，以防止过多数据在网络中积聚，进而防止数据传输受阻。总体而言，流量控制与接收方和发送方业务相关，而拥塞控制更专注于维护网络的健康运行。

（4）服务熔断和降级的概念以及使用场景：

服务熔断和降级都应用于高并发场景下，用以保证系统达到负载极限时的可用性，在过载时确保核心功能的正常运行。但他们存在使用场景下的微小区别。

①服务熔断

• 概念

服务熔断是一种用于提高分布式系统稳定性的设计模式。它的核心思想是在系统中加入一种机制，监控对其他服务的调用，并在检测到连续失败的情况下迅速中断对该服务的调用，防止故障的传播，从而提高系统的可用性。熔断器处于打开状态时，会直接拒绝对服务的请求，而不会执行实际的调用，以免影响整个系统。当一段时间内服务的调用成功率达到阈值时，熔断器会进入半开放状态，允许一些请求进行尝试，以检测服务的恢复情况。

• 使用场景：熔断机制主要应用于一些依赖的外部服务故障场景，避免某服务的故障导致错误传播。

②服务降级

• 概念：

服务降级是一种应对系统负载激增或其他异常情况的策略，其目标是在系统压力过大时，通过有选择地**舍弃**某些功能，以确保核心服务的可用性和稳定性。

• 使用场景：服务降级机制主要应用在系统过载的高并发场景，在高并发时确保核心功能的可用性，**避免系统整体崩溃**。

5. 假设要设计一个秒杀系统，请结合所学知识谈谈如何有效保证高并发处理能力。（提示： 从硬件、架构设计、消息处理、流量控制等方面综合分析，结合业务层面和技术层面讨论。注意：高并发场景中，最后压力会积压在数据库读写层面，高并发处理需要考虑如何将压力在上游层层分散）

<1> 硬件和基础设施：

高性能硬件： 选择高性能的硬件设备，包括服务器、网络设备和存储设备，以确保系统能够处理大量的并发秒杀请求。

负载均衡： 在需求访问量大的地理区域多布局服务器，以避免出现局部地区大量网络拥塞导致服务崩溃。设计良好的负载均衡算法将请求均匀分发到不同的服务器上，防止单一节点成为瓶颈。

<2> 架构设计：

分布式架构：采用分布式架构，将系统拆分成多个独立的服务，包括用户认证、订单处理、库存管理等，以降低单一服务的压力。

缓存技术：使用缓存技术，将热点数据存储在内存中，有效减轻数据库的读取压力。

分布式数据库：使用分布式数据库，以应对不同地区的大量访问，确保数据存储的可扩展性和高可用性。

<3> 消息处理：

消息队列：引入消息队列等流量控制算法，通过异步处理，将瞬时的高峰流量分散到不同的时间段或后台任务中，以避免直接冲击数据库。

流量削峰：实现流量削峰机制，限制用户的访问频率，避免瞬时大量请求同时到达。

<4> 流量控制：

限流机制：通过令牌桶算法或漏桶算法，限制每个用户的请求频率，防止超过系统负荷。

分批处理：将秒杀请求分批处理，而不是一次性处理所有请求，以平滑地分散压力。

<5> 可用性保证：

服务降级： 在高峰期对一些非核心功能进行服务降级，确保核心秒杀功能的正常运行。

<6> 数据库层面：

读写分离：配置数据库的读写分离，将读请求分发到从库，写请求发送到主库，提高数据库的读写能力。

数据库分库分表：根据业务需求，进行数据库的分库分表操作，提高数据库的并发处理能力。

<7> 业务层面：

商品预热： 提前加载秒杀商品信息到缓存，减少请求落到数据库的频率。

用户预热： 提前加载用户信息，包括购物车、地址等，减少用户请求对数据库的冲击。

秒杀令牌机制： 引入秒杀令牌机制，对有资格参与秒杀的用户进行令牌分发，减少非法请求。

秒杀结果异步通知： 将秒杀结果异步通知用户，减少用户在等待结果时的长时间占用。

<8> 安全性与一致性：

防止刷单和抢购：实施合理的用户身份验证和防刷机制，防止用户通过非法手段抢购。

分布式锁：在关键操作中使用分布式锁，确保数据的一致性。

通过以上设计，综合考虑了多个层面的优化措施。**从业务层面**，首先通过减小**并发**的数据请求规模，有效地降低了系统的负担。**在硬件配置上**，采用了缓存技术，通过分布式服务器的部署，从空间上有效地分散了每个服务器的请求规模，提高了整个系统的横向扩展性。在**技术层面**，通过逐步分散每个服务器收到的请求量，引入了消息队列等流量控制算法，以异步处理高峰流量，进一步保护数据库免受直接冲击。同时，引入高可用技术，确保系统在面临单点故障时能够保持可用性，提高系统的鲁棒性。