**分布式系统原理要点 - 雨润的技术博客 - CSDN博客**

| | |
|---|---|
| **笔记本:** | My Notebook |
| **创建时间:** | 2019/9/4 22:58 |
| **URL:** | https://blog.csdn.net/sunyurun/article/details/8297419 |

# 分布式系统原理要点

# 0. 前言

本文是在百度公司刘杰先生创作的《分布式系统原理介绍》一书基础上，整理的要点笔记。一则在整理过程中加深记忆，二来作为一个简洁的
本人删除。

# 1. 分布式系统相关概念

## 1.1 模型

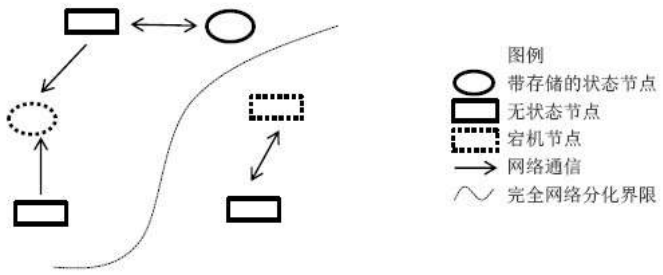



图 1-1 本文的分布式系统模型

### 1.1.1 节点

节点是一个可以独立按照分布式协议完成一组逻辑的程序个体，工程中往往指进程。

### 1.1.2 通信

节点之间完全独立互相隔离，通信唯一方式是通过不可靠的网络。

### 1.1.3 存储

节点可以通过将数据写入与节点在同一台机器的本地存储设备保存数据

### 1.1.4 异常

(1) **机器down机**

大型集群每日down机发生概率0.1%，后果是该机器节点不能工作、重启后失去所有内存信息。

(2) **网络异常**

- 消息丢失：网络拥塞、路由变动、设备异常、network partition（部分不正常）
- 消息乱序：IP存储转发、路由不确定性、网络报文乱序
- 数据错误：比特错误
- 不可靠TCP：到达协议栈之后与到达进程之间、突然down机、分布式多个节点的tcp乱序

(3) **分布式系统的三态**

任何请求都要考虑三种情况：成功、失败、超时。对于超时的请求，无法获知该请求是否被成功执行。

(4) **存储数据丢失**

(5) **其他异常**

IO操作缓慢、网络抖动、拥塞

## 1.2 副本

### 1.2.1 副本的概念

replica/copy 指在分布式系统中为数据或服务提供的冗余：

- 数据副本：在不同的节点上持久化同一份数据。例如GFS同一个chunk的数个副本
- 服务副本：数个节点提供相同的服务，服务不依赖本地存储，数据来自其他节点。例如Map Reduce的Job Worker

### 1.2.2 副本的一致性

副本的consistency是针对分布式系统而言的，不是针对某一个副本而言。根据强弱程度分为：

- 强一致性：任何时刻任何用户/节点都可以读到最近一次更新成功的副本数据
- 单调一致性：任何时刻任何用户一旦读到某个数据某次更新后的值，就不会再读到更旧的值
- 会话一致性：任何时刻任何用户在某次会话内一旦读到某个数据某次更新后的值，就不会在这次会话再读到更旧的值
- 最终一致性：各个副本的数据最终将达到一致状态，但时间不保证
- 弱一致性：没有实用价值，略。

## 1.3 衡量分布式系统的指标

### 1.3.1 性能

- 吞吐量：某一时间可以处理的数据总量
- 响应延迟：完成某一功能需要使用的时间
- 并发能力：QPS(query per second)

### 1.3.2 可用性

系统停服务的时间与正常服务的时间的比例

### 1.3.3 可扩展性

通过扩展集群机器提高系统性能、存储容量、计算能力的特性，是分布式系统特有的性质

### 1.3.4 一致性

副本带来的一致性问题

# 2. 分布式系统原理

## 2.1 数据分布方式

无论计算还是存储，问题输入对象都是数据，如何拆分分布式系统的输入数据称为分布式系统的基本问题。

### 2.1.1 哈希方式

一种常见的哈希方式是按照数据属于的用户id计算哈希。

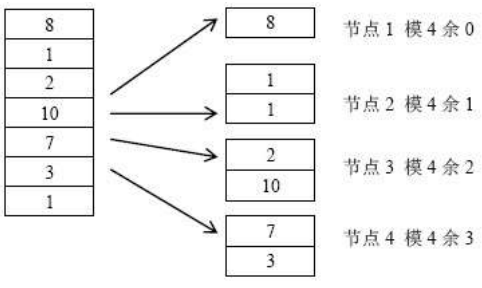



图 2-1 哈希方式分数据

优点：

- 散列性：好
- 元信息：只需要函数+服务器总量

缺点：

- 可扩展性：差。一旦集群规模扩展，大多数数据都需要被迁移并重新分布
- 数据倾斜：当某个用户id的数据量异常庞大时，容易达到单台服务器处理能力的上限

## 2.1.2 按数据范围分布

将数据按特征值的值域范围划分数据。例如将用户id的值域分为[1, 33), [33, 90), [90, 100)，由三台服务器处理。注意区间大小与区间内的数

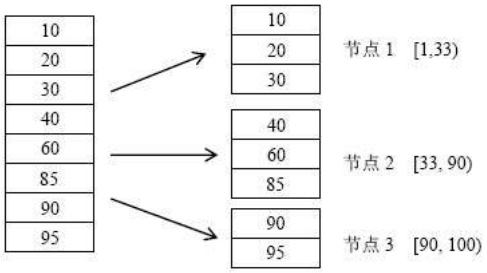



图 2-3 按数据范围分布

优点：

- 可扩展性：好。灵活根据数据量拆分原有数据区间

缺点：

- 元信息：大。容易成为瓶颈。

## 2.1.3 按数据量分布

与按范围分布数据方式类似，元信息容易成为瓶颈

## 2.1.4 一致性哈希

(1) **以机器为节点**

用一个hash函数计算数据（特征）的hash值，令该hash函数的值域成为一个封闭的环，将节点随机分布在环上。每个节点负责处理从自己开的数据。

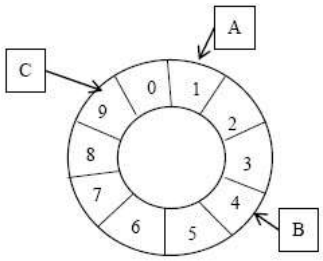



图 2-4 一致性哈希

优点：

- 可扩展性：极好。任意动态添加、删除节点，只影响相邻节点

缺点：

- 元信息：大而且复杂
- 随机分布节点容易造成不均匀
- 动态增加节点后只能缓解相邻节点
- 一个接点异常时压力全转移到相邻节点

(2) **虚节点**

虚节点，虚节点个数远大于机器个数，将虚节点均匀分布到值域环上，通过元数据找到真实机器节点。

优点：

- 某一个节点不可用导致多个虚节点不可用，均衡了压力
- 加入新节点导致增加多个虚节点，缓解了全局压力

### 2.1.5 副本与数据分布

前边4中数据分布方式的讨论中没有考虑数据副本的问题。

(1) **以机器为单位的副本**

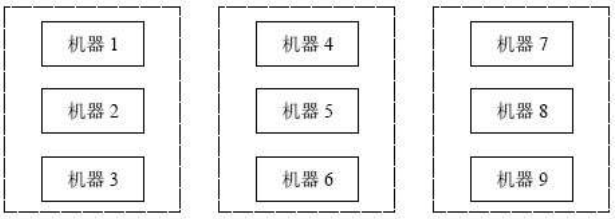



图 2-5 以机器为单位的副本

缺点：

- 恢复效率：低。假如1出问题，从2 3 中全盘拷贝数据较消耗资源，为避免影响服务一般会将2下线专门做拷贝，导致正常工作的副本只有
- 可扩展性：差。假如系统3台机器互为副本，只增加两台机器的情况下无法组成新的副本组。
- 可用性：差。一台donw机，剩下两台压力增加50%。理想情况会均摊到整个集群，而不是单个副本组

(2) **以数据段为单位的副本**

例如3台服务器，10G数据，按100hash取模得到100M每个的数据段，每台服务器负责333个数据段。一旦将数据分成数据段，就可以以数据
机器不再硬相关。

例如系统中3个数据o p q, 每个数据段有三个副本，系统中有4台机器：

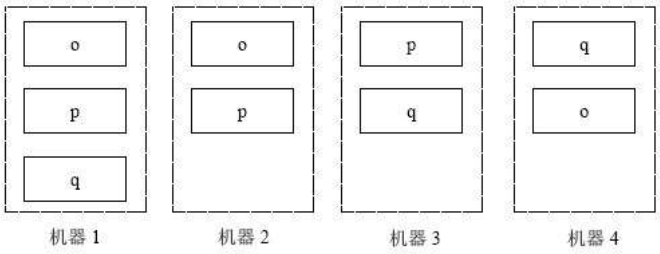



图 2-6 以数据段为单位的副本

优点：

- 恢复效率：高。可以整个集群同时拷贝
- 可用性：好。机器donw机的压力分散到整个集群

工程中完全按照数据段建立副本会引起元数据开销增大，这种做法是将数据段组成一个大数据段。

### 2.1.6 本地化计算

如果计算节点和存储节点位于不同的物理机器，开销很大，网络带宽会成为系统的瓶颈；将计算调度到与存储节点在同一台物理机器上的节点

## 2.2 基本副本协议

### 2.2.1 中心化副本控制协议

- 中心化副本控制协议的基本思路：由一个中心节点协调副本数据的更新、维护副本之间一致性，并发控制
- 并发控制：多个节点同时需要修改副本数据时，需要解决的“写写”、“读写”等并发冲突

单机系统使用加锁等方式进行并发控制，中心化协议也可以使用。缺点是过分依赖中心节点。

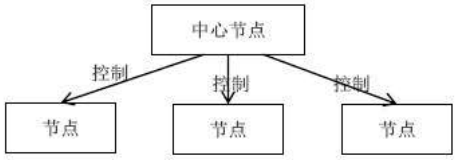



图 2-7 中心化副本控制

## 2.2.2 primary-secondary协议

这是一种常用的中心化副本控制协议，有且仅有一个节点作为primary副本。有4个问题需要解决：

(1) **数据更新基本流程**

- 由primary协调完成更新
- 外部节点将更新操作发给primary节点
- primary节点进行并发控制并确定并发更新操作的先后顺序
- primary节点将更新操作发送给secondary节点
- primary根据secondary节点的完成情况决定更新是否成功，然后将结果返回外部节点

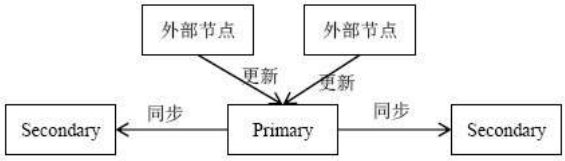



图 2-8 primary-secondary 基本更新流程

其中第4步，有些系统如GFS使用接力的方式同步数据，primary -> secondary1 ->secondary2，避免primary的带宽称为瓶颈。

(2) **数据读取方式**

- 方法一：由于数据更新流程都是由primary控制的，primary副本上数据一定最新。所以永远只读primary副本的数据能够实现强一致性，用之前讨论的方法，将数据分段，将数据段作为副本单位。达到每台机器都有primary副本的目的。
- 方法二：由primary控制secondary的可用性。当primary更新某个secondary不成功时，将其标记为不可用。不可用的secondary副本续直到成功同步后转为可用状态。
- 方法三：Quorum机制。后续讨论。

(3) **primary副本的确定与切换**

- 如何确定primary副本？primary副本所在机器异常时，如何切换副本？
- 通常在primary-secondary分布式系统中，哪个副本为primary这一信息属于元信息，由专门元数据服务器维护。执行更新操作时，首先primary信息，进一步执行数据更新流程。
- 切换副本难点有两个方面：如何确定节点状态以发现原primary节点出现异常(Lease机制)。切换primary后不能影响一致性(Quorum机制

(4) **数据同步**

当发生secondary与primary不一致的情况，需要secondary向primary进行同步(reconcile)。

不一致的原因有3种：

- 网络分化等异常导致secondary落后于primary

  常用的方式是回放primary上的操作日志(redo日志)，追上primary的更新进度

- 某些协议下secondary是脏数据

  丢弃转到第三种情况；或者设计基于undo日志的方式

- secondary是一个新增副本完全没有数据

  直接copy primary的数据，但要求同时primary能继续提供更新服务，这就要求primary副本支持快照(snapshot)功能。即对某一刻的副照，再使用回放日志的方式追赶快照后的更新操作。

### 2.2.3 去中心化副本控制协议

去中心化副本控制协议没有中心节点，节点之间通过平等协商达到一致。工程中唯一能应用在强一致性要求下的是paxos协议。后续介绍。

## 2.3 lease机制

### 2.3.1 基于lease的分布式cache系统

(1) **需求**：分布式系统中有一个节点A存储维护系统的元数据，系统中其他节点通过访问A读取修改元数据，导致A的性能成为系统瓶颈。为此各个节点上cache元数据信息，使得：

- 减少对A的访问，提高性能
- 各个节点cache数据始终与A一致
- 最大可能处理节点down机、网络中断等异常

(2) **解决方案原理**：

- A向各个节点发送数据的同时向节点颁发一个lease，每个lease具有一个有效期，通常是一个明确的时间点。一旦真实时间超过次时间点
- 在lease有效期内，A保证不会修改对应数据的值。
- A在修改数据时，首先阻塞所有新的读请求，等待之前为该数据发出的所有lease过期，然后修改数据的值

(3) **客户端节点读取元数据的流程**

```
if (元数据处于本地cache && lease处于有效期内) {
    直接返回cache中的元数据;
} else {
    Result = 向A请求读取元数据信息;
    if (Result.Status == SUCCESS) {
        WriteToCache(Result.data, Result.lease);
    } else if (Result.Status == FAIL || Result.Status == TIMEOUT) {
        retry() or exit();
    }
}
```

(4) **客户端节点修改元数据的流程**

- 节点向A发起修改元数据的请求
- A收到修改请求后阻塞所有新的读数据请求，即接受读请求但不返回数据
- A等待所有与该元数据相关的lease超时
- A修改元数据并向节点返回修改成功

(5) **优化点**

- A修改元数据时要阻塞所有新的读请求

  这是为了防止发出新的lease而引起不断有新的cache节点持有lease，形成活锁。优化的手段是：一旦A进入修改流程，不是盲目屏蔽读据不返回lease。造成cache节点只能读取数据，不能cache数据。进一步的优化是返回当前已发出的lease的最大值。这样同样能避免活

- A在修改元数据时需要等待所有lease过期

优化手段是：A主动通知各个持有lease的节点，放弃lease并清除cache中相关数据。如果收到cache节点的reply，确认协商通过，则可间失败或超时，则进入正常等待lease过期的流程，不会影响协议正确性。

### 2.3.2 lease机制的深入分析

(1) **lease定义**

lease是由颁发者授予的再某一有效期内的承诺。颁发者一旦发出lease，无论接收方是否收到，无论后续接收方处于何种状态，只要lease不另一方面，接受者在lease的有效期内可以使用颁发者的承诺，一旦lease过期则一定不能继续使用。

(2) **lease的解读**

由于lease仅仅是一种承诺，具体的承诺内容可以非常宽泛，可以是上节中数据的正确性，也可以是某种权限。例如并发控制中同一时刻只给有lease才能修改数据；例如primary-secondary中持有lease的节点具有primary身份等等

(3) **lease的高可用性**

- 有效期的引入，非常好的解决了网络异常问题。由于lease是确定的时间点，所以可以简单重发；一旦受到lease之后，就不再依赖网络通

(4) **lease的时钟同步**

工程上将颁发者有效期设置的比接受者打，大过时钟误差，来避免对lease有效性产生影响。

### 2.3.3 基于lease机制确定节点状态。

在分 布式系统中确定一个节点是否处于正常工作状态困难的问题。由可能存在网络分化，节点的状态无法通过网络通信来确定的。

例如： a b c 互为副本 a为主节点，q为监控节点。q通过ping来判断abc的状态，认为a不能工作。就将主节点切换成b。但是事实上仅仅是有问题，就出现了 a b都觉得自己是主节点的“双主”问题。

解决方法是q在发送heartbeat时加上lease，表示确认了abc的状态，并允许节点在lease有效期内正常工作。q给primary节点一个特殊的lea工作。一旦q希望切换primary，只需要等待之前primary的lease过期，避免出现双主问题。

### 2.3.4 lease的有效期时间选择

工程上一般选择10秒钟

## 2.4 Quorum机制

### 2.4.1 Write-all-read-one

对于某次更新操作Wi，只有在所有N个副本上都更新成功，才认为是一次“成功提交的更新操作”，对应的数据为“成功提交的数据”，数

缺点：

- 频繁读写版本号容易成为瓶颈
- N-1个副本成功的情况下，仍然被认为更新失败

### 2.4.2 Quorum定义

WARO最大程度增强读服务可用性，最大程度牺牲写服务的可用性。将读写可用性折中，就会得到Quorum机制：

Quorum机制下，某次更新Wi一旦在所有N个副本中的W个副本上成功，就称为“成功提交的更新操作”，对应的数据为“成功提交的数据取R个副本则一定能读到Wi更新后的数据Vi。

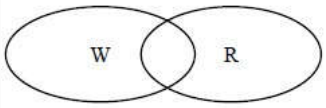



图 2-10 Quorum 机制

由此可见WARO是Quorum中W = N时的一个特例。

### 2.4.3 读取最新成功提交的数据

Quorum的定义基于两个假设：

- 读取者知道当前已提交的版本号

- 每次都是一次成功的提交

现在取消这两个假设，分析下面这个实际问题：

N = 5, W = 3, R = 3，某一次的副本状态为(V2 V2 V2 V1 V1)。理论上读取任何3个副本一定能读到最新的数据V2，但是仅读3个副本却无法

例如读到的是(V2 V1 V1)，因为当副本状态为(V2 V1 V1 V1 V1)时也会读到(V2 V1 V1)，而此时V2版本的数据是一次失败的提交。因此只读
是V2还是V1。


图 2-11 仅读 R 个副本无法判断最新已成功提交的数据

解决方案：

- 限制提交的更新操作必须严格递增，只有前一个更新操作成功后才可以提交下一个。保证成功的版本号连续
- 读取R个副本，对其中版本号最高的数据：若已存在W个，则该数据为最新结果；否则假设为X个，则继续读取其他副本直到成功读取W
到W个，则第二大的版本号为最新成功提交的版本。

因此单纯使用Quorum机制时，最多需要读取R + (W - R - 1) = N个副本才能确定最新成功提交的版本。实际工程中一般使用quorum与pri
避免需要读取N个副本。

### 2.4.4 基于Quorum机制选择primary

- 在primary-secondary协议中引入quorum机制：即primary成功更新W个副本后向用户返回成功
- 当primary异常时需要选择一个新的primary，之后secondary副本与primary同步数据

通常情况下切换primary由监控节点q完成，引入quorum之后，选择新的primary的方式与读取数据相似，即q读取R个副本，选择R个副本中
primary。新primary与除去q选举出的副本外，其余的至少W个副本完成数据同步后，再作为新的primary。

蕴含的道理是：

- R个副本中版本号最高的副本一定蕴含了最新的成功提交的数据。
- 虽然不能确定版本号最高的数据 == 这个最新成功提交的数据，但新的primary立即完成了对W个副本的更新，从而使其变成了成功提交

例如：N = 5 W = 3 R = 3的系统，某时刻副本状态(V2 V2 V1 V1 V1)，此时V1是最新成功提交的数据，V2是处于中间状态未成功提交的数
是作为新数据被同步，完全取决于能否参与q主持的新primary的选举大会。如果q选择(V1 V1 V1)，则在接下来的更新过程中 V2会被当成脏
V1)则V2会将V1更新掉，成为最新成功的数据。

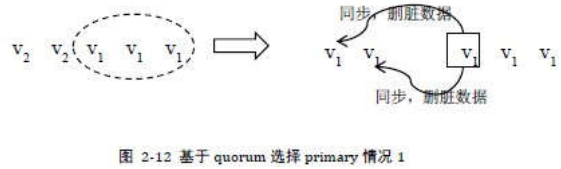
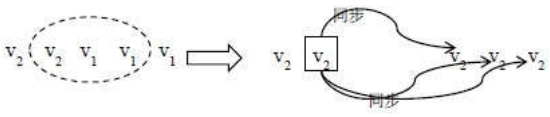



图 2-12 基于 quorum 选择 primary 情况 1




## 2.5 日志技术

日志技术不是一种分布式系统技术，但分布式系统广泛使用日志技术做down机恢复。

### 2.5.1 数据库系统日志回顾

数据库的日志分为 undo redo redo/undo no-redo/no-undo四种，这里不做过多介绍。

### 2.5.2 redo与check point

通过redo log恢复down机的缺点是需要扫描整个redolog，回放所有redo日志。解决这个问题的办法是引入checkpoint技术，简化模型下
中间，将内存以某种数据组织方式dump到磁盘上。这样down机恢复时只需要从最后一个end向前找到最近一个begin，恢复中间的内存状

### 2.5.3 no-undo/no-redo

这种技术也叫做01目录，即有两个目录，活动目录和非活动目录，另外还有一个主记录，要么“记录目录0”，妖魔记录“使用目录1”，目文件中的位置。

## 2.6 两阶段提交协议

## 2.7 基于MVCC的分布式事务

由于这两个都与数据库事务有关，且两阶段提交协议在工程中使用价值不高，均略去不谈。

## 2.8 Paxos协议

唯一在工程中有使用价值的去中心化副本节点控制协议。过于复杂，没看懂。

## 2.9 CAP理论

- Consitency
- Availiablity
- Tolerance to the Partition of network

无法设计一种分布式协议，使得完全具备CAP三个属性。永远只能介于三者之间折中。理论的意义是：不要妄想设计完美的分布式系统。