分布式系统

1. CAP理论

CAP理论由加州大学的计算机科学家 Eric Brewer 提出衡量分布式系统的三个指标：

1. **Consistency(一致性)**：所有节点在同一时刻看到的数据内容是一样的
2. **Availability(可用性)**：每一个request都能得到成功或者失败的response
3. **Partition tolerance(分割容忍性)**：在任意节点故障的情况下，系统能对外提供服务

分布式系统不能够同时满足上述三种特性，但是可以去做trade-off

Partition tolerance是一个既定的事实，因为总是需要假定网络是不可靠的，因此**CAP实际上是在C和A之间做trade-off**

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。

2. 数据一致性(C侧)

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。

一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器，由于维护数据副本的一致性代价高，因此许多系统**采用弱一致性来提高性能**，一些不同的一致性模型也相继被提出。

- **强一致性**： 系统中的某个值被更新之后，后续所有的读操作都要能读到更新后的数据
- **弱一致性**：除了强一致性以外所有的情况都叫弱一致性，即不保证所有的读操作都能读到更新后的数据

2.1. 一致性的基本要求

规范的说，理想的分布式系统一致性应该满足：

1. 可终止性（Termination）：一致的结果在有限时间内能完成；
2. 共识性（Consensus）：不同节点最终完成决策的结果应该相同；

3. 合法性（Validity）：决策的结果必须是其它进程提出的提案。

第一点很容易理解，这是计算机系统可以被使用的前提。需要注意，在现实生活中这点并不是总能得到保障的，例如取款机有时候会是 服务中断 状态，电话有时候是 无法连通 的。

第二点看似容易，但是隐藏了一些潜在信息。算法考虑的是任意的情形，凡事一旦推广到任意情形，就往往有一些惊人的结果。例如现在就剩一张票了，中关村和西单的电影院也分别刚确认过这张票的存在，然后两个电影院同时来了一个顾客要买票，从各自观察看来，自己的顾客都是第一个到的……怎么能达成结果的共识呢？记住我们的唯一秘诀：**核心在于需要把两件事情进行排序，而且这个顺序还得是合理的、大家都认可的。**

第三点看似绕口，但是其实比较容易理解，即达成的结果必须是节点执行操作的结果。仍以卖票为例，如果两个影院各自卖出去一千张，那么达成的结果就是还剩八千张，决不能认为票售光了。

2.2. 强一致性的分类

- **线性一致性(Linearizable Consistency)**

线性一致性也叫严格一致性（Strict Consistency）或者原子一致性（Atomic Consistency），它的条件是：

1. **任何一次读都能读取到某个数据最近的一次写的数据。**
2. **所有进程看到的操作顺序都跟全局时钟下的顺序一致。**

线性一致性是对一致性要求最高的一致性模型，就现有技术是不可能实现的。因为它要求所有操作都实时同步，在分布式系统中要做到全局完全一致时钟现有技术是做不到的。首先通信是必然有延迟的，一旦有延迟，时钟的同步就没法做到一致。当然不排除以后新的技术能够做到，但目前而言线性一致性是无法实现的。

- **顺序一致性(Sequential Consistency)**

顺序一致性是 Lamport（1979）在解决多处理器系统共享存储器时首次提出来的。它的条件是：

1. **任何一个读操作都是按照某种顺序的**
2. **所有进程看到的读写顺序都一样**

那么线性一致性和顺序一致性的区别在哪里呢？通过上面的分析可以发现，**顺序一致性虽然通过逻辑时钟保证所有进程保持一致的读写操作顺序，但这些读写操作的顺序跟实际上发生的顺序并不一定一致。**而线性一致性是严格保证跟实际发生的顺序一致的。

2.3. 弱一致性的分类

- **因果一致性(Casual Consistency)**

因果一致性是一种弱化的顺序一致性模型，因为它将具有潜在因果关系的事件和没有因果关系的事件区分开了。那么什么是因果关系？如果事件 B 是由事件 A 引起的或者受事件 A 的影响，那么这两个事件就具有因果关系。

> 因果关系X -> Y包含：
>
> 1. 在同一节点上，X发生在Y之前（single thread）
> 2. 在不同的节点上，X causes Y（e.g. X是添加照片，Y是删除照片）
> 3. X -> Z -> Y（具有传递性）

因果一致性的条件包括：

1. **所有进程必须以相同的顺序看到具有因果关系的读写操作。**
2. **不同进程可以以不同的顺序看到并发的读写操作。**

- **最终一致性(Eventual Consistency)**

是弱一致性的一种特例，保证用户最终（即不一致时间窗口窗口尽量长）能够读取到某操作对系统特定数据的更新。

> 弱一致性不保证经过“不一致时间窗口“后所有节点读到的数据都是一样的
> 最终一致性保证经过“不一致时间窗口”后所有节点读到的数据都是一样的

- **以客户端为中心的一致性(Client-Centric Consistency)**

前面我们讨论的一致性模型都是针对数据存储的多副本之间如何做到一致性，考虑这么一种场景：在最终一致性的模型中，如果客户端在数据不同步的时间窗口内访问不同的副本的同一个数据，会出现读取同一个数据却得到不同的值的情况。为了解决这个问题，有人提出了以客户端为中心的一致性模型。**以客户端为中心的一致性为单一客户端提供一致性保证，保证该客户端对数据存储的访问的一致性，但是它不为不同客户端的并发访问提供任何一致性保证。**
举个例子：客户端 A 在副本 M 上读取 x 的最新值为 1，假设副本 M 挂了，客户端 A 连接到副本 N 上，此时副本 N 上面的 x 值为旧版本的 0，那么一致性模型会保证客户端 A 读取到的 x 的值为 1，而不是旧版本的 0。一种可行的方案就是给数据 x 加版本标记，同时客户端 A 会缓存 x 的值，通过比较版本来识别数据的新旧，保证客户端不会读取到旧的值。

以客户端为中心的一致性包含了四种子模型：

1. **单调读一致性（Monotonic-read Consistency）**：如果一个进程读取数据项 x 的值，那么该进程对于 x 后续的所有读操作要么读取到第一次读取的值要么读取到更新的值。即保证客户端不会读取到旧值。
2. **单调写一致性（Monotonic-write Consistency）**：一个进程对数据项 x 的写操作必须在该进程对 x 执行任何后续写操作之前完成。即保证客户端的写操作是串行的。
3. **读写一致性（Read-your-writes Consistency）**：一个进程对数据项 x 执行一次写操作的结果总是会被该进程对 x 执行的后续读操作看见。即保证客户端能读到自己最新写入的值。
4. **写读一致性（Writes-follow-reads Consistency）**：同一个进程对数据项 x 执行的读操作之后的写操作，保证发生在与 x 读取值相同或比之更新的值上。即保证客户端对一个数据项的写操作是基于该客户端最新读取的值。

2.4. 一致性解决方案

1. 分布式事务：两段提交
2. 分布式锁
3. 消息队列、消息持久化、重试、幂等操作
4. Raft / Paxos 等一致性算法

3. 服务可用性(A侧)

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。

3.1. 可用性解决方案

1. **负载均衡**：尽力将网络流量平均分发到多个服务器上，以提高系统整体的响应速度和可用性。
2. **降级**：当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。
3. **熔断**：对于目标服务的请求和调用大量超时或失败，这时应该熔断该服务的所有调用，并且对于后续调用应直接返回，从而快速释放资源。确保在目标服务不可用的这段时间内，所有对它的调用都是立即返回的、不会阻塞的，等到目标服务好转后进行接口恢复。
4. **流量控制**：流量控制可以有效的防止由于网络中瞬间的大量数据对网络带来的冲击，保证用户

网络高效而稳定的运行，类似于TCP拥塞控制方法。

5. **异地多活**：在不同地区维护不同子系统，并保证子系统的可用性

熔断是减少由于下游服务故障对自己的影响；而降级则是在整个系统的角度上，考虑业务整体流量，保护核心业务稳定。

4. 分区容错性(P侧)

网络分区(Network Partition)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(Network Partition)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。

一般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。

5. 高并发系统的设计

5.1. 系统拆分

将一个系统拆分为多个子系统，用 RPC 来搞。然后每个系统连一个数据库，这样本来就一个库，现在多个数据库，不也可以扛高并发么。

5.2. 缓存

大部分的高并发场景，都是读多写少，那你完全可以在数据库和缓存里都写一份，然后读的时候大量走缓存不就得了。毕竟 Redis 轻轻松松单机几万的并发。所以你可以考虑考虑你的项目里，那些承载主要请求的读场景，怎么用缓存来抗高并发。

5.3. 消息队列

可能你还是会出现高并发写的场景，比如说一个业务操作里要频繁搞数据库几十次，增删改增删改。那高并发绝对搞挂你的系统，你要是用 Redis 来承载写那肯定不行，人家是缓存，数据随时就被 LRU 了，数据格式还无比简单，没有事务支持。所以该用 MySQL 还得用 MySQL 啊。那你咋办？用 MQ 吧，大量的写请求灌入 MQ 里，后边系统消费后慢慢写，控制在 MySQL 承载范围

之内。所以你得考虑考虑你的项目里，那些承载复杂写业务逻辑的场景里，如何用 MQ 来异步写，提升并发性。

5.4. 分库分表

分库分表，可能到了最后数据库层面还是免不了抗高并发的要求，好吧，那么就将一个数据库拆分为多个库，多个库来扛更高的并发；然后将一个表拆分为多个表，每个表的数据量保持少一点，提高 SQL 跑的性能。

5.5. 读写分离

读写分离，这个就是说大部分时候数据库可能也是读多写少，没必要所有请求都集中在一个库上吧，可以搞个主从架构，主库写入，从库读取，搞一个读写分离。读流量太多的时候，还可以加更多的从库。

6. 分布式缓存

6.1. 应用场景

1. **页面缓存**：用来缓存Web 页面的内容片段,包括HTML、CSS 和图片等;
2. **应用对象缓存**：缓存系统作为ORM 框架的二级缓存对外提供服务,目的是减轻数据库的负载压力,加速应用访问;解决分布式Web部署的 session 同步问题，状态缓存.缓存包括Session 会话状态及应用横向扩展时的状态数据等,这类数据一般是难以恢复的,对可用性要求较高,多应用于高可用集群。
3. **并行处理**：通常涉及大量中间计算结果需要共享;
4. **云计算领域提供分布式缓存服务**

6.2. 缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效、新缓存未到之间(**例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期**)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

6.3. 缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。**这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空**（*相当于进行了两次无用的查询*）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

6.4. 缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

6.5. 缓存更新

除了缓存服务器自带的缓存失效策略之外，我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

1. 定时去清理过期的缓存；
2. 当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

6.6. 缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是 **保证核心服务可用，即使是有损的**。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

1. **一般**：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；
2. **警告**：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；
3. **错误**：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承

受的最大阈值，此时可以根据情况自动降级或者人工降级；

4. **严重错误**：比如因为特殊原因数据错误了，此时需要紧急人工降级。