分布式与高可用

1.分布式事务

1）什么是分布式事务

事务是一个抽象概念，它将某个事件进行的所有操作纳入到一个不可分割的执行单元，组成事务的所有操作必须同时执行以保证数据的一致性。

本地事务（比如 MySQL 的 InnoDB 引擎下的事务）遵循 ACID 的原则，是通过多种日志和锁来保证的。其中：

- I，隔离性通过数据库锁实现；
- D，持久性通过 redo log 和 binlog 实现；
- A/C，原子性和一致性通过 undo log 实现。

分布式事务就是指事务的参与者（支持事务的服务器、资源服务器及事务管理器）分别位于不同的分布式节点上，简单来说就是事务的各个操作由不同的应用节点去执行，分布式事务就是为了保证不同节点的数据一致性。

一致性分为三种情况：强一致性、弱一致性和最终一致性。

2）分布式事务的基础

CAP 和 BASE 理论，见下文。

3）分布式事务常见的解决分案

2PC

2PC，二阶段提交协议（Two-phase commit protocol）。二阶段提交引入一个事务协调者的角色来管理各参与者的提交和回滚，二阶段是指事务同步的两个阶段：

- 准备（预提交）阶段：事务协调者要求每个涉及到事务操作的参与者预提交此操作，并反映是否可以提交；
- 提交（回滚）阶段：事务协调者要求每个数据库提交（回滚）数据，只要有一个参与者预提交失败，事务协调者就会向所有的参与者发送回滚命令。

从上看出，如果第一阶段有参与者提交失败，那么协调者就让所有的参与者回滚。那如果第二阶段提交失败呢？两种情况分析：

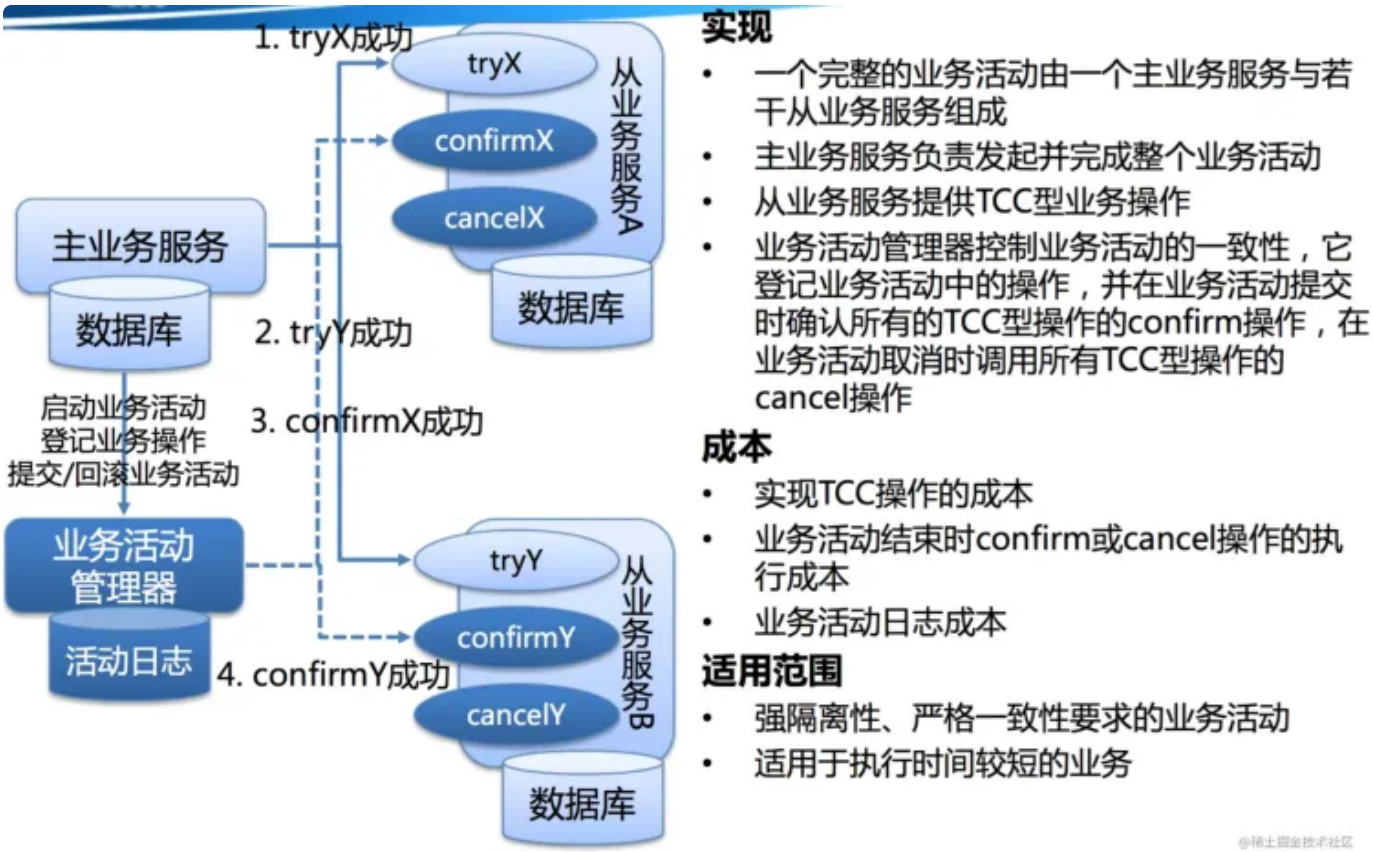
1. 如果第二阶段执行的是回滚操作，那么就不断重试，直到所有参与者都回滚，不然第一阶段准备提交的参与者会一直阻塞；
2. 如果第二阶段是提交事务操作，那么也不断重试，直到所有参与者都提交成功（头铁往前冲），最后如果出问题了，只能人工介入处理。

优点：2PC 尽量保持了数据强一致性，实现成本低，各大主流数据库都有实现，比如 MySQL 从 5.5 版本开始支持。

缺点：

1. 单点故障问题：各参与者都依赖协调者来决定是否执行事务，若协调者节点发生故障或者宕机，就会导致参与者处于阻塞状态；
2. 数据不一致：当协调者在判断事务是否提交后，像各参与者发送处理命令，若此时节点挂掉，会导致只有一部分参与者收到了提交或者回滚的命令，会导致数据不一致；
3. 不支持高并发：二阶段提交过程是同步阻塞的，效率低下。

TCC



TCC（Try-Confirm-Cancel）引入业务活动管理器，管理各节点事务的一致性，它将事务处理分为三个阶段：

1. Try 阶段：尝试执行，完成所有业务检查，预留必须的业务资源；
2. Confirm 阶段：确认执行业务（满足幂等性），不做任务业务检查，只使用 Try 阶段预留的业务资源，失败后会进行重试；
3. Cancel 阶段：取消执行，释放 Try 阶段预留的业务资源，也满足幂等性。

举例说明：比如你用 100 元买一瓶水：

- Try：检查钱包的钱是否大于等于 100，并锁住资源（100 元和这瓶水）；

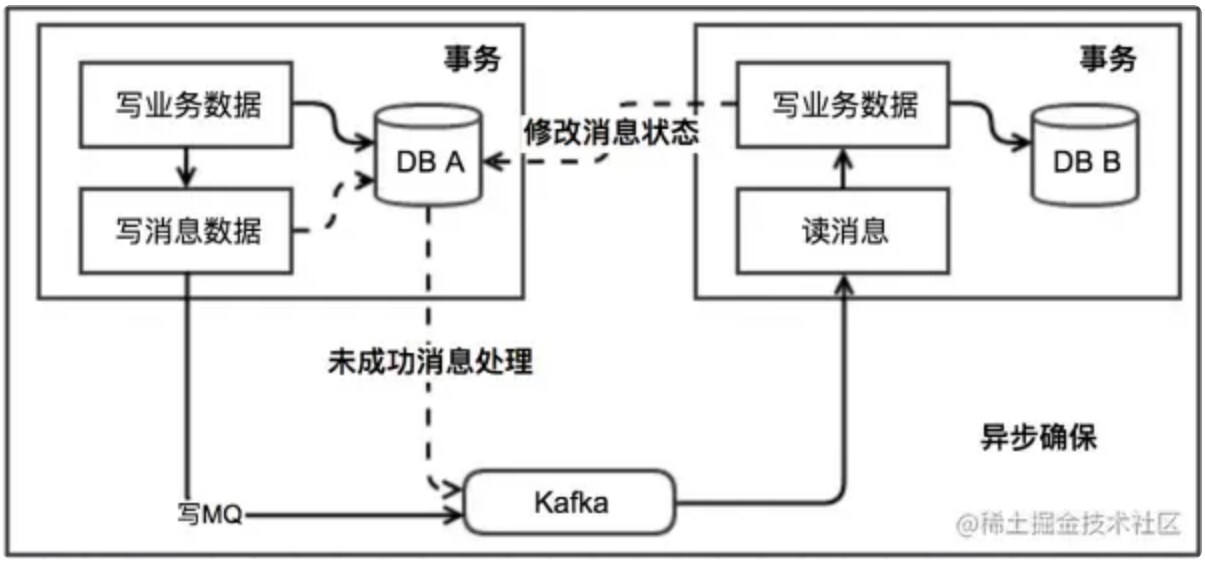
- Cancel：如果有一个资源锁定失败，则进行 cancel 释放资源，这个过程中无论 cancel 还是其它操作失败都进行重试 cancel，所以需要保证幂等性；
- Confirm：如果资源锁定都成功，则进行 confirm，资源交换，这个过程中无论 confirm 还是其它操作失败都进行重试 confirm，需保证幂等性。

TCC 的出现解决二阶段提交的几个缺点：

1. 单点故障问题：引入了多个业务活动管理器，集群下高可用；
2. 数据不一致问题：引入超时补偿机制，由业务活动管理器来控制一致性；
3. 同步阻塞问题：引入超时补偿机制，不会锁定同步，将资源转换为业务逻辑形式，粒度更小。

本地消息表

本地消息表是 ebay 公司提出的事务解决方案（完整：https://queue.acm.org/detail.cfm?id=1394128），它的核心原理是将需要分布式处理的任务通过消息日志的方式来异步执行。消息日志可以存储到本地文件、数据库或消息队列，再通过业务规则或人工发起重试。



本地消息表基于 BASE 理论，实现数据的最终一致性，实现过程中需要注意幂等性原则。

最大努力通知

本地消息表，或者通过 MQ 对事务进行通知都可以算作最大努力。本地消息表通过后台定时任务去异步保证数据的一致性，就是一种最大努力通知的思想：代表系统各模块之间已经最大程度地保证事务的最终一致性了。

4）分布式算法

1）Paxos 算法【读音：pksos】

Paxos 算法将系统中的节点分为三类：

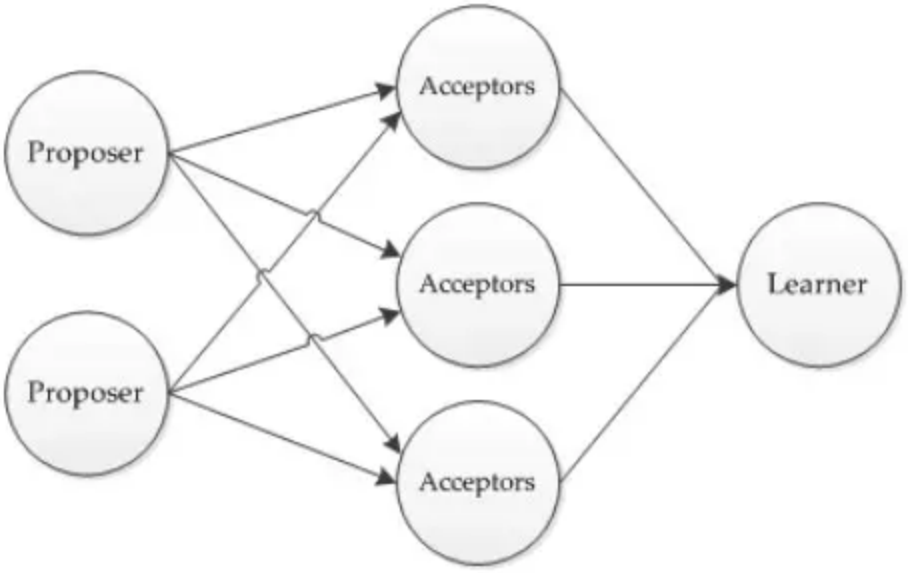


Figure 1: Basic Paxos architecture. A number of proposers make proposals to acceptors. When an acceptor accepts a value it sends the result to learner nodes.

- 提议者（Proposer）：提议一个值
- 接受者（Aceeptor）：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。

提议的时候，包含俩字段：[n, v]，其中 n 为序号，v 为提议值。每个 Aceeptor 在接收提议请求的时候，会比对其中的序号 n：

- 当前序号小于已存在的 n 时，则不予理会；
- 当前序号大于 n 时，会返回响应，表示接受了这个序号为 n 的提议。

当一个 Proposer 接收到超过半数的 Aceeptor 响应时，说明该提议值被 Paxos 选择了出来，就可以发送通知给所有的 Learner。

2）Raft 算法

引入主节点，通过竞选来获取主节点。节点分为三类：

- 领头结点 Leader
- 从节点 Follower
- 候选节点 Candidate

1. Leader 变为 Candidate

每个 Follower 都会接收 Leader 周期性的心跳，一般为 150~300ms，如果一段时间之后还未收到心跳包，Follower 就变为 Candidate。

2. Candidate 竞选 Leader

Follower 变为 Candidate 后，开始发送投票消息给其它所有存活节点，其它节点会对其请求进行回复，如果超过半数的节点回复了竞选请求，那么该 Candidate 就会变成 Leader 节点。如果平票，则每个节点设置一个随机时间后开始竞选，所有节点重新进行投票。

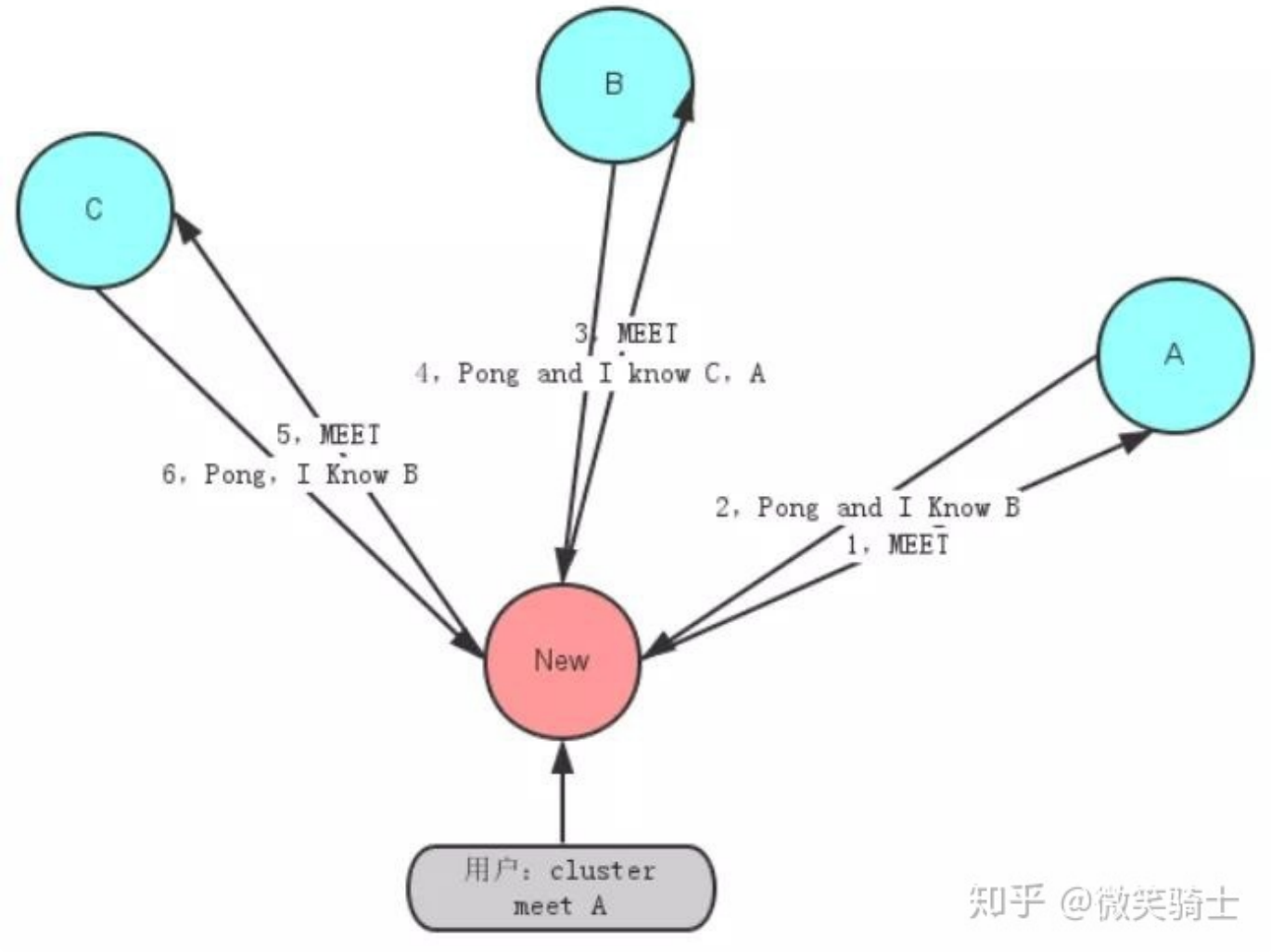
3. 新 Leader 开始工作

新 Leader 周期性发送心跳包给 Follower，Follower 收到心跳包以后重新计时。这时，Leader 如果接收到了客户端请求，会将数据变更写入日志中，并把数据复制到所有 Follower。当大多数 Follower 进行修改后，将数据变更操作提交。然后，Leader 会通知所有的 Follower 让它们提交修改，此时所有节点的数据达成一致。

3）Gossip算法

Gossip 又被称为流行病算法，它与流行病毒在人群中传播的性质类似，由初始的几个节点向周围互相传播，到后期的大规模互相传播，最终达到一致性。

Gossip 协议被广泛应用于 P2P 网络，同时一些分布式的数据库，如 Redis 集群的消息同步使用的也是 Gossip 协议，另一个重大应用是被用于比特币的交易信息和区块信息的传播。



Gossip传输示意图

Gossip 协议的整体流程非常简单，传输示意图见上图.初始由几个节点发起消息，这几个节点会将消息的更新内容告诉自己周围的节点，收到消息的节点再将这些信息告诉周围的节点。依照这种方式，获得消息的节点会越来越多，总体消息的传输规模会越来越大，消息的传偶速度也越来越快。

虽然不是每个节点都能在相同的时间达成一致，但是最终集群中所有节点的信息必然是一致的，Gossip 协议确保的是分布式集群的最终一致性。

预先设定好消息更新的周期时间 T，以及每个节点每个周期能够传播的周围节点数 2，我们可以得到大致的消息更新流程如下：

1. 节点 A 收到新消息并更新
2. 节点 A 将收到的消息传递给与之直接相连的 B,C
3. B,C 各自将新更新的消息传给与之直接相连的两个节点，这些节点不包含 A
4. 最终集群达成一致

4）一致性hash算法

解决hash算法的迁移成本，向集群中添加节点时，有一致hash算法，可以极大地降低迁移数据量。

1）不带虚拟节点的一致性hash算法，和普通的hash算法一样，都是将节点取模进行路由寻址。不同的是，一致性hash算法通过对 2^32进行取模，形成一个硕大的hash环，节点取模后将位置映射到hash环上。读取key值时，通过hash函数，将key对应的位置在环上找出来，并沿一个方向进行查找，遇到的第一个节点就是key对应的节点。这时，如果其中某个节点宕机了，只需要在环上的相近位置扩容一个节点就可，此时的数据迁移仅仅是新节点和宕机节点之间的数据。它的问题是，可能会存在很多访问请求只集中在少量的几个节点上，数据分布不均。

带虚拟节点的一致性 hash 算法，对每个服务器计算多个 hash 值，在每个计算结果对应的位置上，都放置一个虚拟的节点。而当新的访问到虚拟节点时，会映射到真实的节点上。

2）项目场景，某个region区的缓存达到上限？新增缓存节点，为了保证缓存数据的均匀，一般会采用key值hash，然后取模的方式。最后根据结果，确认数据落到哪个节点上。　　这时有个问题，我们在删减服务器的时候，大部分缓存都会失效。我们希望增删缓存服务器时，大部分的key依旧在原来的缓存服务器上不变，用“一致性hash算法”。

2.高可用设计

2.1 限流

当系统的处理能力不能应对外部突增的流量访问时，为了让系统保持稳定，必须采取限流措施。

1）限流指标

- TPS，Transactions Per Second，每秒完成的事务数。用这个值来做限流是最合理的，但是不太现实，因为在分布式业务系统中，事务往往需要多个模块配合完成。按照 TPS 来限流，时间粒度可能会很大，很难准确评估系统的响应性能。
- HPS，Hits Per Second，每秒请求数。如果每笔事务完成一个请求，那 TPS 和 HPS 是等同的。但分布式场景下，完成一笔事务可能需要多次请求，所以 TPS 和 HPS 不能等同看待。
- QPS，Query Per Second，每秒能响应客户端查询的数量。一般指数据库系统的查询次数，也是衡量服务器整体性能的一个重要标准。

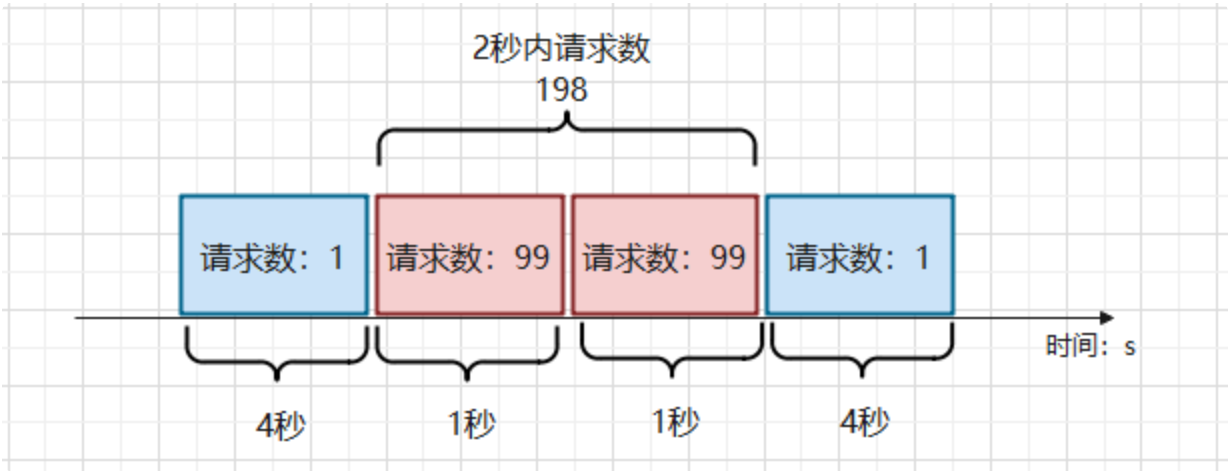
目前，主流的限流方法多采用 HPS 作为限流指标。

2）限流方法

1. 流量计数器

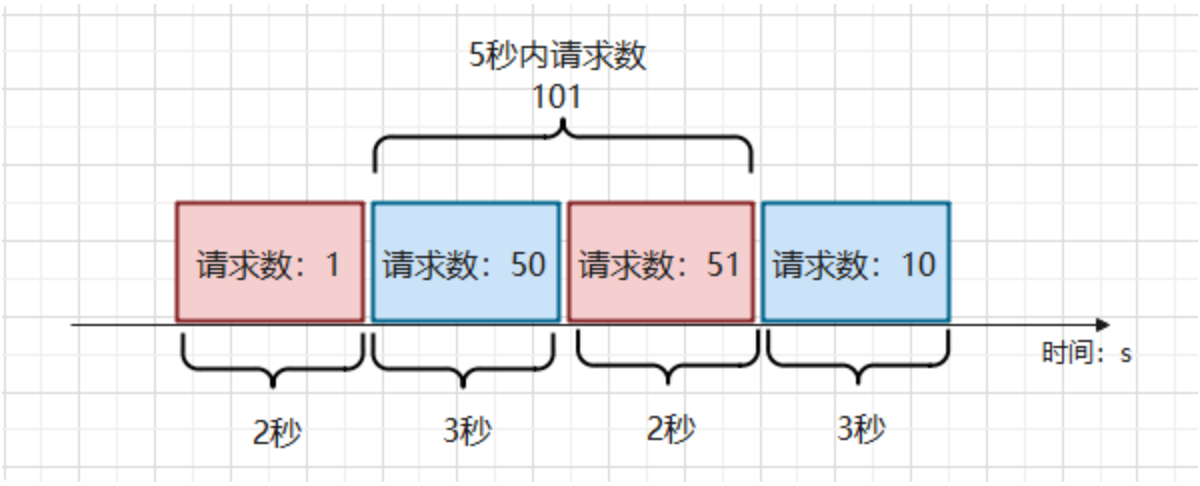
最简单直接的方法，比如限制 5 秒内请求数量最大为 100，超过这个数量就拒绝访问。但这个方法存在两个明显的问题：

1. 单位时间（比如：为什么是 5 秒内）很难把控，容易出现集中时间访问。比如出现以下场景：



前 4 秒只有一个访问量，第 5 秒有 99 个访问量；第 6 秒有 99 个访问量，接下来的 4 秒又只有 1 个访问量。从全局看，10 秒内 200 个请求流量没有超出，但从图中来看，这种流量情况肯定是异常的。

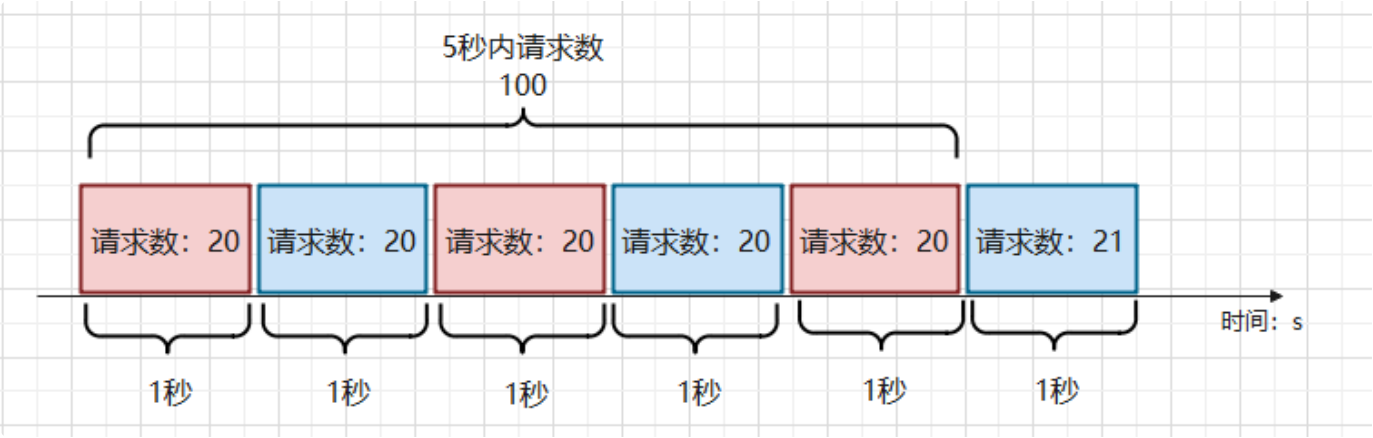
2. 有一段时间超了，但是并不一定真的需要限流。比如以下场景：



如果中间两块访问正好在一个 5 秒的周期内，那流量就超出限制了。这种情况下，后续的 10 个请求可能就会被丢弃，不太合理。

2. 滑动时间窗口

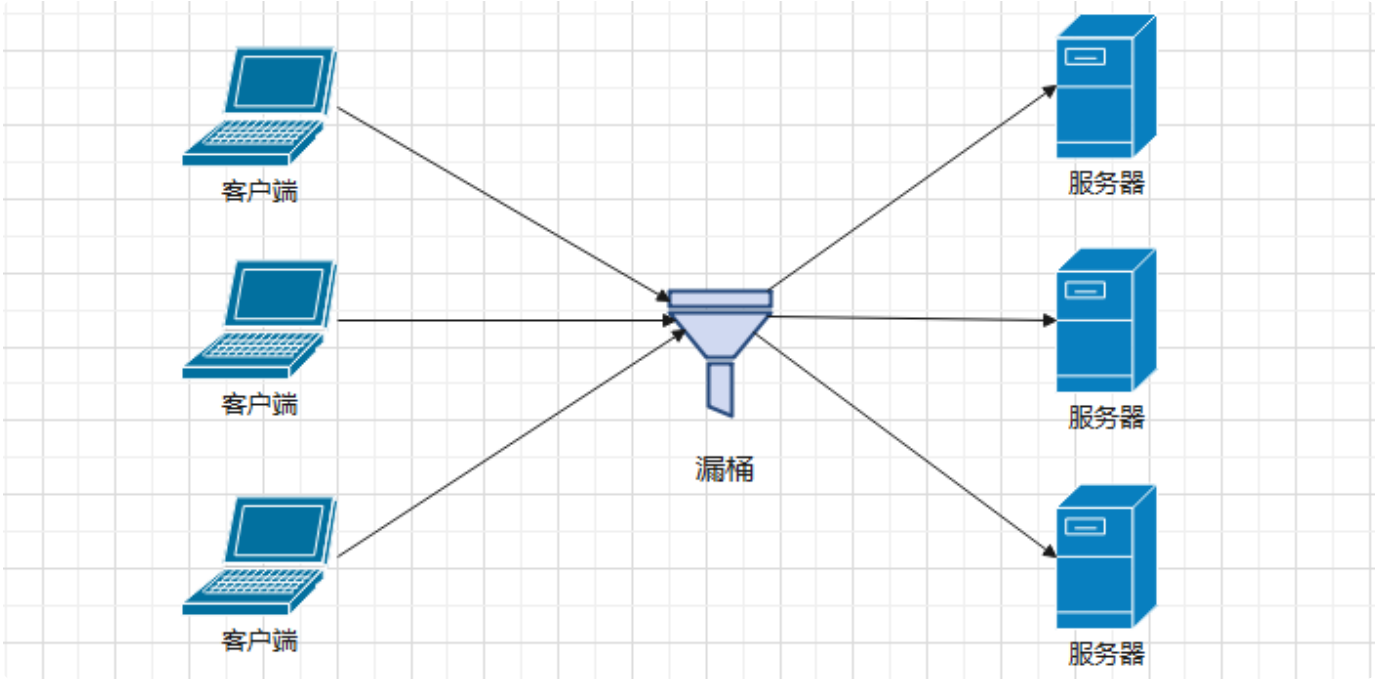
滑动时间窗口是目前比较流行的限流算法，主要思想是把时间看作是一个向前滚动的窗口，如下图：



它的特点是，将时间分片处理，滑动窗口每次统计一个总时间周期内的请求数。下一个时间段时，就把前面的时间片抛弃，加入后面时间片的请求数，解决了流量计数器可能出现的问题。它的缺点在于**对流量控制不够精细，不能限制集中在短时间内的流量**。

3. 漏桶算法

漏桶算法的思想如下图：



漏桶是一个大小固定的队列，会把客户端发送的请求缓存起来，然后再均匀地发送到服务器上。

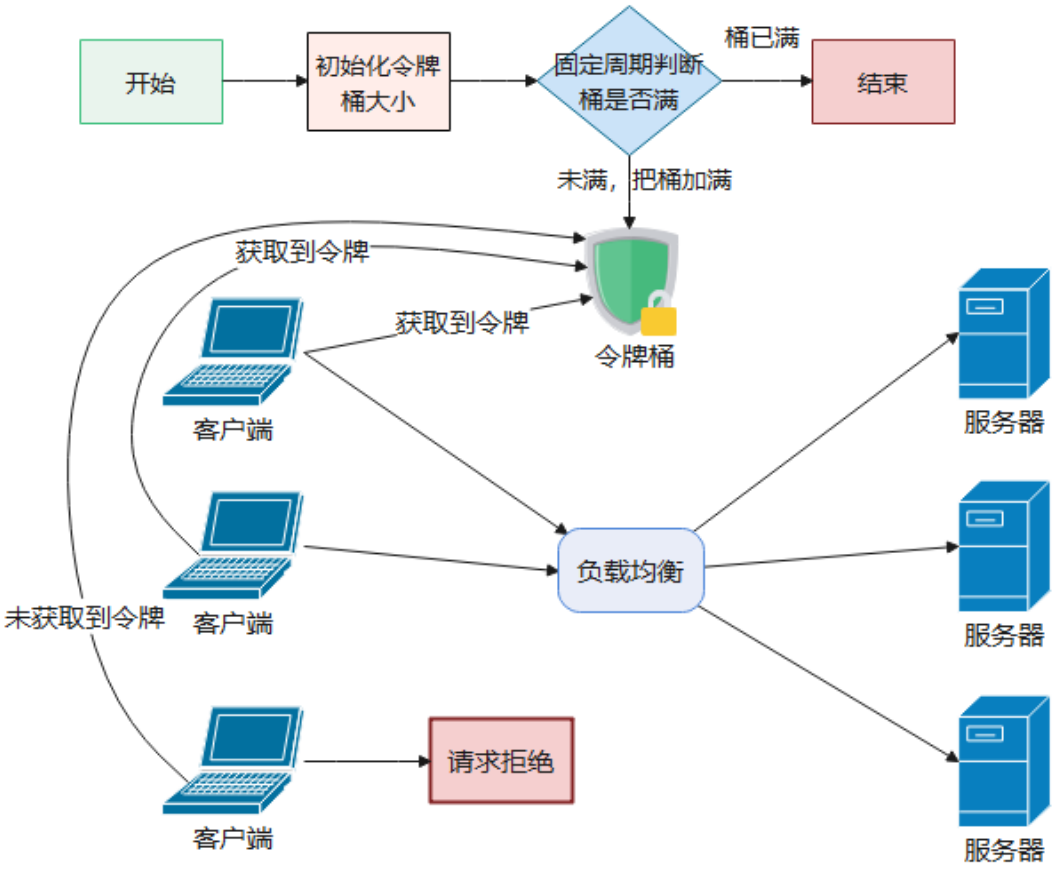
如果客户端请求速率太快，漏桶的队列满了，就会直接拒绝掉，或者走降级处理逻辑，不会冲击到服务器端。

漏桶算法的优点是实现简单，可以使用**消息队列**来削峰填谷。但是它也有几个问题：

- 漏桶大小不容易控制，太大会给服务器带来较大压力，太小可能会导致大量请求被丢弃；
- 漏桶给服务端的请求速率难以控制；
- 使用缓存请求的方式，会让请求的响应时间变长。

4. 令牌桶算法

令牌桶算法和去医院挂号是差不多的逻辑，看医生之前需要先挂号，而医院每天放的号是有限的：

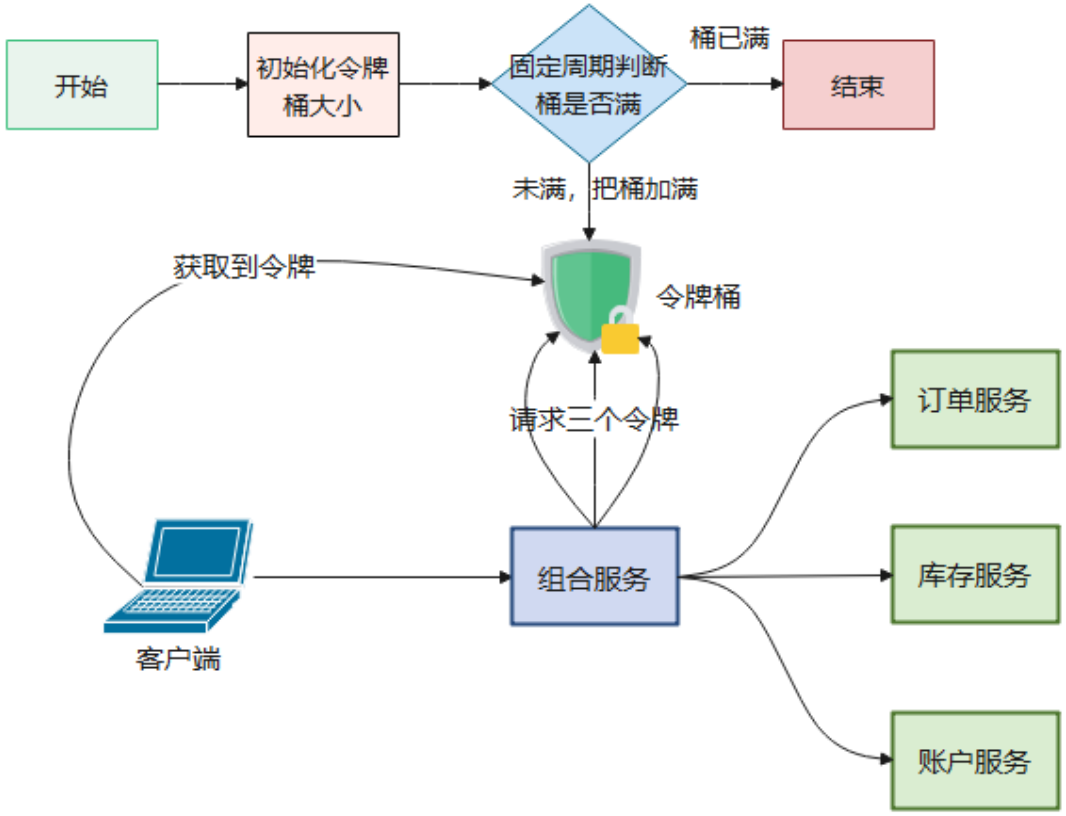


令牌桶算法中的令牌也是一样，客户端在发送请求之前，都需要先从令牌桶中获取令牌。如果取到了，就发送请求；如果取不到令牌，就只能被拒绝需要等待。

令牌桶算法解决了漏桶算法的三个问题（速率难控制，漏桶大小难控制和请求周期变长的问题），而且实现并不复杂，使用信号量就可以实现。在实际的限流场景中使用最多，比如 Google 和 Guava 中就使用了令牌桶限流。

5. 分布式场景下如何限流

在分布式场景中，上述限流方案还能否适用呢？举个栗子：



如果我们把令牌放到一个单独的中间件（比如 Redis）中供整个分布式系统用，那客户端在调用组合服务，组合服务调用订单、库存和账户服务时都需要和令牌桶交互，交互次数明显增加了很多。

有一种改进是，客户端在调用服务之前首先获取 4 个令牌，调用组合服务时减去一个令牌并传递给组合服务三个令牌，调用子服务时分别消耗一个令牌。

6. hystrix 限流

在 Go 服务中，可以采用 hystrix-go 开源包来限流，只需配置关键字信息：

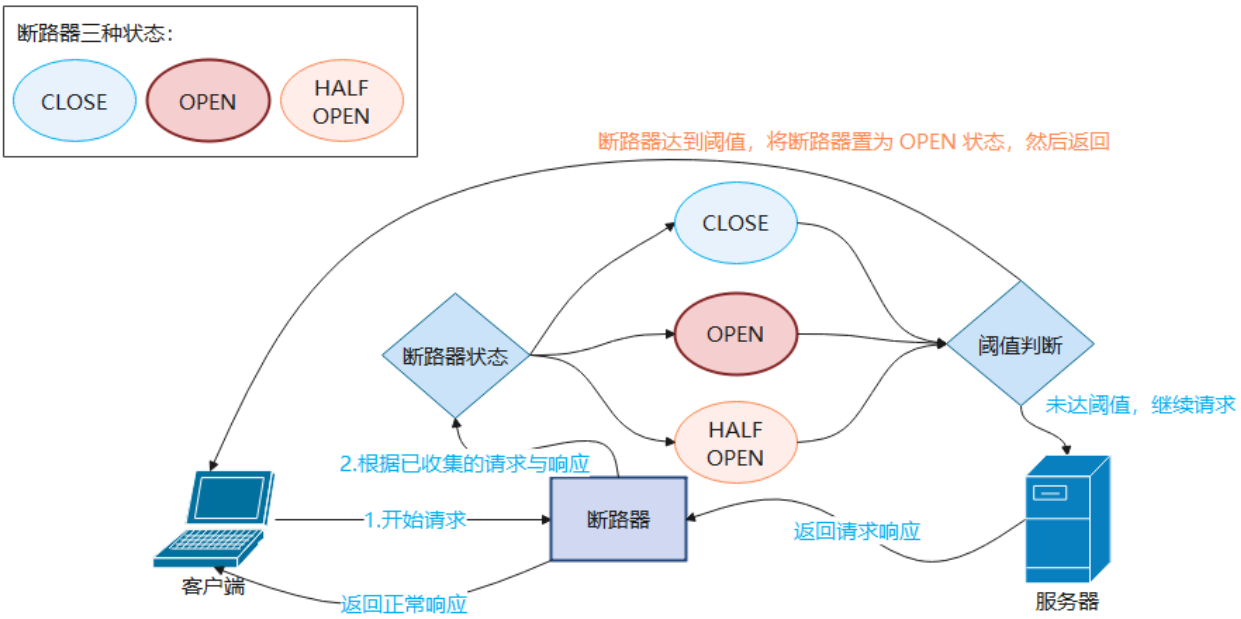
```
1   Timeout: int(3 * time.Second),// 执行command的超时时间为3s
2   MaxConcurrentRequests: 500,// command的最大并发量，限流个数
```

2.2 熔断

1. 熔断相关的概念

熔断，相当于在请求和服务之间加了保险丝。当服务扛不住持续的访问压力时，保险丝就断开，防止服务器扛不住压力而崩溃。

服务熔断是指在访问服务时先通过断路器做代理进行访问，断路器会持续观察服务返回的状态是正常还是异常，当失败次数超过设置的阈值（比如：100次请求有50次都是异常的）时断路器打开，接下来一段时间的请求就不能真正访问服务了。



断路器有三种状态：

- CLOSE：默认关闭状态。断路器观察到请求失败的比例没有达到阈值，断路器继续处于关闭状态；
- OPEN：开启状态。请求失败比例已达阈值，断路器打开，请求不再到达服务器，而是直接返回；
- HALF OPEN：半开启状态。断路器打开一段时间过后，切换为半打开状态，这时断路器会尝试去请求服务器以查看请求是否已正常。若成功，则断路器转为 CLOSE 状态；否则转为 OPEN 状态。

2. hystrix 熔断

熔断机制有很多，Go 语言里的 hystrix-go 开源包可以完美支持，它提供如下字段：

```
Timeout: int(3 * time.Second),// 执行command的超时时间为3s
MaxConcurrentRequests: 10,// command的最大并发量
RequestVolumeThreshold: 5000,// 统计窗口10s内的请求数量，达到这个请求数量后才去判断是否要开启熔断
SleepWindow: 20,// 当熔断被打开后，多久以后去尝试服务是否可用了
ErrorPercentThreshold: 30,// 错误百分比，请求数量>=RequestVolumeThreshold，并且错误率到达这个百分比后启动熔断
```

使用 hystrix-go 需要考虑的问题：

- 针对不同异常，定义不同的熔断处理逻辑（比如：订单服务需要给用户良好的反馈信息，服务器正忙请稍后再试）；
- 服务器故障或者升级时，让客户端知晓目前正在升级，不影响其它模块使用。

2.3 服务降级

1. 常见场景

相比限流和熔断，服务降级是站在系统全局的视角来考虑的。在服务发生熔断以后，一般会让请求走实现配置的处理方法，这个处理方法就是一个降级逻辑。

服务降级是对非核心、非关键的服务进行降级。有如下使用场景：

- 服务处理异常，把请求缓存下来，给客户返回一个中间态，事后再重试缓存里的请求；
- 系统监控检测到突增流量，为了避免非核心功能消费系统资源，临时关闭这些非核心业务功能；
- 对于耗时的同步任务，可以改为异步处理；
- 暂时关闭批处理任务，以节省系统资源。

2. hystrix 简介

Hystrix 是 Netflix 开源的一款服务治理框架，包含常用的容错方法：

- 线程池隔离
- 信号量隔离
- 支持限流、熔断、降级回退

2.4 总结

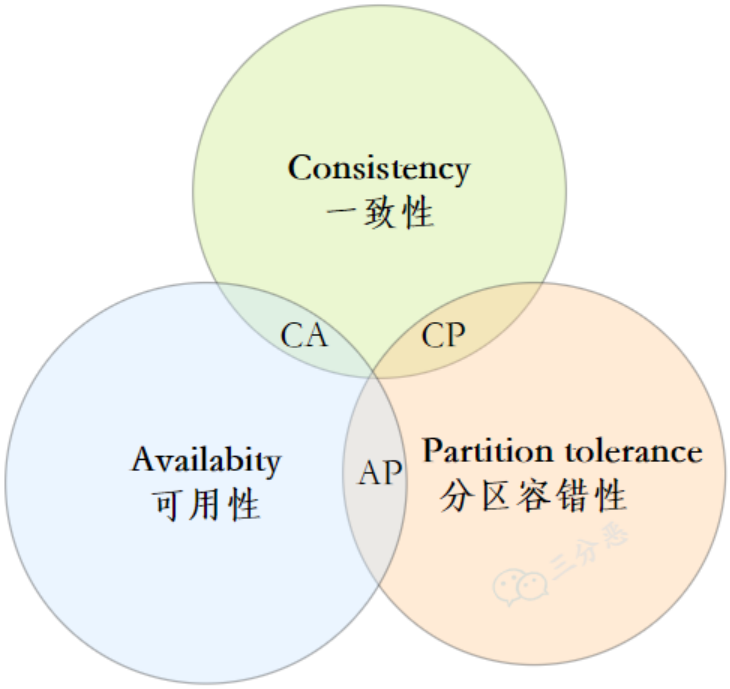
限流、熔断和服务降级是系统容错的重要设计模式，从一定意义上讲限流和熔断也是服务降级的手段。

熔断和服务降级主要是针对非核心业务，而核心业务如果流量超过预估峰值，最好提前通过性能测试找到这个峰值，对其进行限流。对于限流，选择合适的限流算法很重要，令牌桶算法的优势较为明显，也是很多大型项目中采用的限流算法。

在系统设计的时候，这些模式需要配合业务量的评估、性能测试的数据进行阈值的修改，而这些阈值最好保存在可热更新的配置中心，方便实时修改。

3. CAP 理论

CAP 是指在分布式系统中，Consistency（一致性）、Availability（可用性）和 Partition tolerance（分区容错性）这三个基本原则，最多只能同时满足其中的 2 个。



1）CAP 的概念

一致性C

一致性分为三种，分别是强一致性、弱一致性和最终一致性：

- 强一致性：对于关系型数据库，要求更新过的数据后续访问都能看到。比如在订单系统中球鞋库存还剩 10 双，张三刚买了一双球鞋，数据更新完成后，接下来李四看到的球鞋数量就只有 9 双，否则就可能会出现超卖的情况；
- 弱一致性：系统中的数据被更新后，可以容忍后续的访问拿到更新之前的数据，也就是后续的部分访问或者全部访问可能会读到未同步的数据；

- 最终一致性：是弱一致性的特殊形式，要求系统的数据更新完成，在一段时间以后，后续的所有访问都能拿到最新的数据。比如订单系统中抢商品，可以给抢购留一个缓冲期，用户买完商品付了钱，提醒用户等待一段时间。一段时间以后，数据同步完成，告诉用户商品是否抢购成功：若成功，则订单完成，准备发货；若失败，则给用户操作退款。

一般的业务系统基于性价比的考量，绝大多数都是采用“最终一致性”作为分布式系统的设计思想。

而 CAP 理论里的“一致性”，则要求是强一致性。正如官方文档中描述的那样："All nodes see the same data at the same time"，所有节点在同一时间内数据完全一致。

可用性A

可用性描述的是系统能够很好地为用户服务，不会出现用户操作失败或者访问超时的情况，影响用户体验。

即 "Reads and writes always succeed"，服务在正常响应时间内一直可用。

分区容错性P

分区从广义上来讲是指某个业务系统的多个节点，为了解决系统的单点故障问题，分布式系统中一定需要多个分区来保证系统的高可用。

分区容错性是指多个分区，也就是节点之间的网络通信出现故障了，或者系统中的某一个节点出问题了，我们仍然需要保证业务系统可用。

即 "The system continues to operate despite arbitrary message loss or failure of part of the system"，分布式系统在遇到某个节点或者网络分区故障时，仍然能够对外提供满足一致性或可用性的服务。

2）CAP 的证明

为什么 C，A，P 三者不可兼得？首先，我们得知道，在分布式系统中，**分区容错性是一定要保证的**。

试想如果只有一个分区，谈分布式就没有意义了；而多个分区，一定会有分区的故障问题，分布式系统中保证分区容错就变成最基本的诉求了。所以现在我们只需考虑在分区容错的基础上，能否同时满足一致性和可用性。

假设现在有两个分区 P1 和 P2，分区上都有同一份数据 D1 和 D2， 现在它们是完全相同的。接下来，有一个请求 A 访问了 P1，更改了 D1 上的数据；然后又有一个请求 B 访问了 P2，去访问 D2 的同一份数据：

- 满足一致性，那要求 D2 和 D1 数据完全一致，那必须在更新 D1 数据时给 P2 上的 D2 数据上锁，等待 D1 更新完成后再同步更新 D2。但这个过程中，锁住的 D2 肯定就没法给请求 B 实时响应，也就是违背了 P2 上的可用性；
- 满足可用性，就要求 P1 和 P2 都可以实时响应，因此 D2 和 D1 最新的数据肯定是不一致的，也就违背了 P1 和 P2 上的数据一致性。

可以看出，在保证分区容错的前提下，一致性和可用性不能同时满足。

3）CAP 如何权衡

CAP 三者不可兼得，该怎么选择呢？一般根据我们的业务可以有以下选择：

- CP without A：保证分区的强一致性（C），不要求可用（A）。相当于请求到达某个系统之前，需要等待数据完全同步以后，才会得到系统的数据响应，一般在数据需严格保持一致的金融系统中会使用这种模式；
- AP without C：保证分区的可用性（A），不要求强一致性（C）。当请求访问某个分区的数据时，可能拿到未同步的老数据，这种模式一般只要求数据满足最终一致性，进而保证系统响应速度和高可用，在业界使用范围较广，比如著名的 BASE 理论（下节细讲）。
- CA without P：同时保证系统的强一致性（C）和可用性（A），在分布式系统中不成立，因为分区是客观存在而无法避免的，而单体系统中的数据库可以通过事务保证数据的一致性和可用性。

4. BASE 理论

BASE 理论是当今互联网分布式系统的实践总结，它的核心思想在于，既然在分布式系统中实现强一致性的代价太大，那不如退而求其次：只需要各应用分区在提供高可用服务的基础上，尽最大能力保证数据一致性，也就是保证数据的**最终一致性**。

BASE 理论是 CAP 中保证分区容错（P）的前提下，对可用性（A）和一致性（C）的权衡，它由 Basically Available（基本可用），Soft State（软状态），Eventually-Consistent（最终一致性）三个词组构成。

1）基本可用

一个系统的可能出现非**核心**功能需求或者**非功能需求**的异常，其中：

- 非核心功能需求：比如一个银行系统，它的提款、转账等交易模块就是核心功能，是用户的基本需求，不能出问题；而非核心功能可以出现异常，但需要保证在一段时间内修复。
- 非功能需求：比如银行转账需要在 0.5 秒内完成，但是由于网络延迟等原因，可以延迟响应至1~2 秒。

由于系统出现此类异常，从而影响了系统的高可用性，但核心流程依然可用，即基本可用性。

2）软状态

软状态是指系统服务可能处于中间状态，数据在保证一致性的过程中可能延迟同步，但不会影响系统的可用性。比如我们在购买火车票付款结束之后，就可能处在一个既没有完全成功，也没有失败的中间等待状态。用户需要等待系统的数据完全同步以后，才会得到是否购票成功的最终状态。

3）最终一致性

最终一致性强调的是系统所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。它不像强一致性那样，需要分区数据保证实时一致，导致系统数据的同步代价过高；也不像弱一致性那样，数据更新后不保证数据一致，导致后续的请求只能访问到老数据。

在实际的业务系统中，最终一致性包含以下 5 种场景：

- 因果一致性（Causal consistency），如果节点 A 更新完某个数据后通知了节点 B，那么 A 和 B 接下来的访问数据都可以拿到最新的值，而其它和 A 没有因果关系的节点则可能会访问到老数据；
- 读自己所写（Read your writes），某个节点更新后的数据自身总是可以访问到，其它节点可能需要一段时间后再同步；
- 会话一致性（Session consistency），在一个会话当中，系统中的所有节点都可以读到最新的数据，实现会话中的 "读自己所写"；
- 单调读一致性（Monotonic read consistency），当从某个节点中读出了最新值，系统中的其它节点就必须保证读到的值是最新的；
- 单调写一致性（Monotonic write consistency），系统需要保证同一个节点上的写操作顺序执行。

当前业界的分布式系统，甚至关系数据库系统的数据，大都是用最终一致性实现的。比如 MySQL 的主从备份，就是在一段时间内通过 binlog 日志让从库和主库的数据保持最终一致。

总的来说，BASE 理论其实就是牺牲了各节点数据的强一致性，允许不同节点的数据在一段时间内不一致，来获得更高的性能和高可用性。

在单体系统中，数据库还能通过 ACID 来实现事务的强一致性，但分布式事务需要考虑节点通信的延迟和网络故障。所以，BASE 理论常常是我们在实际业务系统中考量的方案。

5. 分布式性能优化

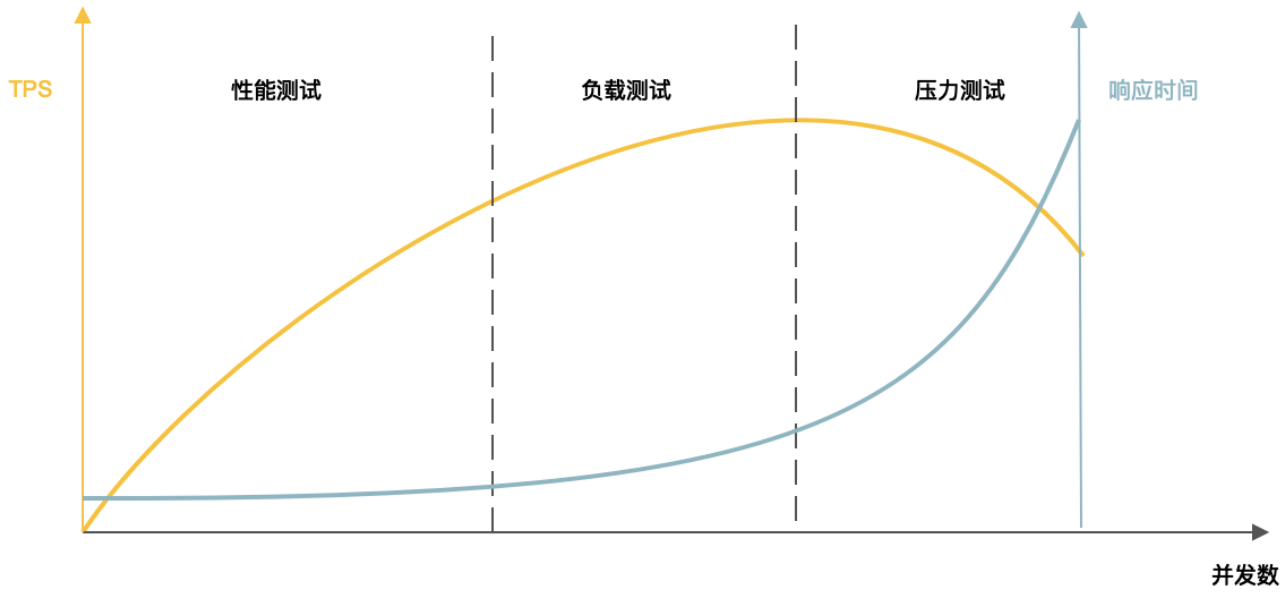
5.1 性能量化的三个指标

首先，系统性能的量化主要分为三个指标：响应时间、并发数和吞吐量

指标名称	定义	作用
响应时间	从发出请求到收到最后响应数据所需要的时间	是系统最重要的性能指标，最直接地反映了系统的快慢。
并发数	系统同时处理的请求数	反映系统的负载压力情况。性能测试的时候，通常在性能压测工具中，用多线程模拟并发用户请求，每个线程模拟一个用户请求，这个线程数就是性能指标中的并发数。
吞吐量	单位时间内系统处理请求的数量	体现系统的处理能力。我们一般用每秒HTTP请求数HPS、每秒事务数TPS、每秒查询数QPS这样的一些指标来衡量。

一般来说，吞吐量 = 并发数/响应时间。这几个非常重要的性能指标可以在系统运行期间通过监控获取，也可以上线前通过测试获取。

5.2 系统测试的三个阶段



1. 性能测试：以系统设计初期规划的性能指标为预期目标，对系统不断施加压力，验证系统在资源可接受的范围内是否达到了性能预期目标。
2. 负载测试：对系统不断施加并发请求，增加系统的压力，直到系统的某项或多项指标达到安全临界值。
3. 压力测试：超过安全负载的情况下，增加并发请求数，对系统继续施加压力，直到系统崩溃或不再处理任何请求，此时的并发数就是系统的最大压力承受能力。

5.3 架构优化的三板斧

1. 负载均衡

通过分布式集群扩展服务器节点，降低单一节点的负载压力。

2. 分布式缓存

负载均衡降低了服务器节点的访问压力，但是没法降低数据库的负载压力。所以引入分布式缓存来降低系统的读负载压力。

高并发架构中常见的分布式缓存有三种：CDN、反向代理和分布式对象缓存。

3. 消息队列

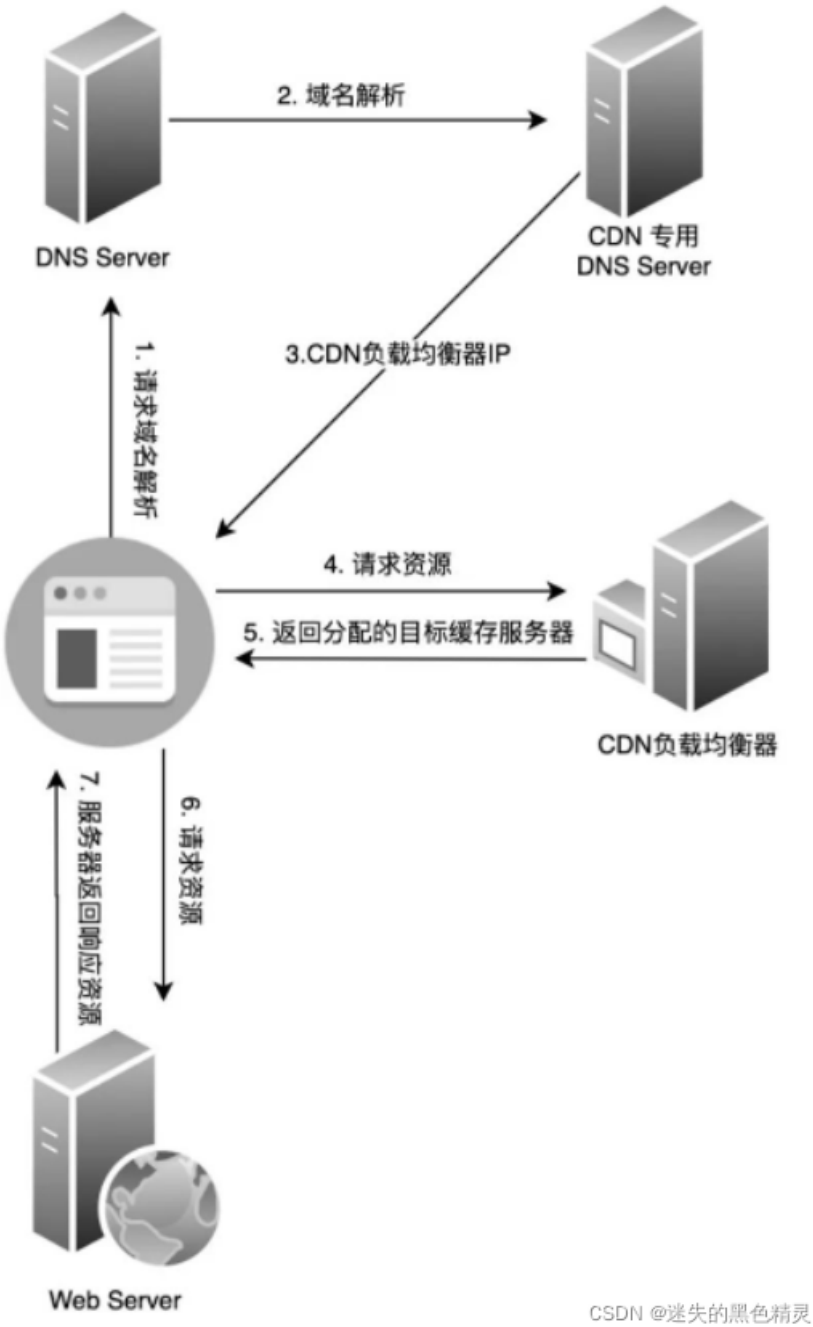
降低系统的写负载压力。

6. CDN

CDN，内容分发网络（content delivery network），部署在网络运营商机房的一种缓存服务器。因为离用户距离近，所以 CDN 可以更快速的响应用户请求，加快访问速度。同时，CDN 还能极大降低数据中心的访问压力。

工作流程：将服务器部署到用户广泛使用的地区节点，当用户访问时，通过全局负载技术将请求分发到最近的正常的服务器上，由它们直接给用户提供服务。

采用 CDN 的优点是能够极大地避免网络拥堵的情况，使得内容传输更快、更稳定，访问 CDN 的流程如下：



1. 由于 DNS 服务器将对 CDN 的域名解析权交给了 CNAME 指向的专用 DNS 服务器，所以对用户输入域名的解析最终是在 CDN 专用的 DNS 服务器上完成的。
2. 解析出的结果 IP 地址并非确定的 CDN 缓存服务器地址，而是 CDN 的负载均衡器的地址。
3. 浏览器会重新向该负载均衡器发起请求，经过对用户 IP 地址的距离、所请求资源内容的位置及各个服务器复杂状况的综合计算，返回给用户确定的缓存服务器IP地址。
4. 对目标缓存服务器请求所需资源的过程。

这个过程也可能会发生所需资源未找到的情况，那么此时便会依次向其上一级缓存服务器继续请求查询，直至追溯到网站的根服务器并将资源拉取到本地。

CDN 网络的核心功能包括两点：

1. 缓存：缓存指的是将所需的静态资源文件复制一份到 CDN 缓存服务器上；
2. 回源，回源指的是如果未在 CDN 缓存服务器上查找到目标资源，或 CDN 缓存服务器上的缓存资源已经过期，则重新追溯到网站根服务器获取相关资源的过程。

7. Nginx

反向代理服务器，主要提供缓存功能。

用户访问某词条时，Nginx 先查找自己服务器上是否有缓存该词条内容，如果有就直接返回；如果没有，Nginx 就会访问应用服务器获取请求内容。将新数据缓存到自己服务器，并返回给用户。

8. 主从架构

9. 集群架构

10. 分层架构

11.一致性选举算法

12. 异地多活

高可用架构中的各种策略，基本上都是针对一个数据中心内的系统架构，针对服务器级别的软硬件故障进行设计的。但如果我们整个数据中心都不可用，比如数据中心所在的称号四遭遇了地震，机房停电或者遇到火灾等情况，不管我们的架构设计的多么高可用，应用依然是不可用的。

为了解决这个问题，很多大型互联网应用都采用了异地多活的多机房架构策略，也就是将数据中心分布在多个不同地点的机房里，这些机房都可以对外提供服务。当某个机房断电了，用户可以连接任意一个其它机房进行访问，保证系统的高可用性。