

分布式理论

分布式系统定义及面临的问题

分布式系统定义为：

分布式系统是一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。

上面这个简单的定义涵盖了几乎所有有效地部署了网络化计算机的系统。通俗的理解，所谓分布式系统，就是一个业务拆分成多个子业务，分布在不同的服务器节点，共同构成的系统称为分布式系统，同一个分布式系统中的服务器节点在空间部署上是可以随意分布的，这些服务器可能放在不同的机柜中，也可能在不同的机房中，甚至分布在不同的城市

类比一下，分布式系统就是一群一起干活。人多力量大，每个服务器的算力是有限的，但是通过分布式系统，由n个服务器组成起来的集群，算力是无限扩张的。

分布式环境的各种问题

通信异常

分布式系统需要在各个节点之间进行通信，通信必然要引入网络因素，而由于网络本身的不可靠性，因此每次网络通信都会伴随着网络不可用的风险（光纤、路由、DNS等硬件设备或系统的不可用），都会导致最终分布式系统无法顺利进行一次网络通信，另外，即使分布式系统各节点之间的网络通信能够正常执行，其延时也会大于单机操作，存在巨大的延时差别，也会影响消息的收发过程，因此消息丢失和消息延迟变的非常普遍。

网络分区

由于网络发生异常情况，导致分布式系统中部分节点之间的网络延迟不断增大，最终导致组成分布式系统中只有部分节点能够进行正常通信，而另外一些节点则不能，这种情况为网络分区，也就是俗称的“脑裂”。当出现网络分区时，网络之间出现了网络不连通，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干个孤立的区域，分布式系统就会出现局部小集群，在极端情况下，这些小集群会独立完成原本需要整个分布式系统才能完成的功能，包括数据的事务处理，这就对分布式一致性提出非常大的挑战。

三态

从上面的介绍，已经了解到了在分布式环境下，网络可能会出现各式各样的问题，因此分布式系统每一次请求与响应存在特有的“三态”概念，即成功、失败和超时。在传统的单机系统中，应用程序调用一个函数之后，能够得到非常明确的响应：成功或失败，而在分布式系统中，由于网络是不可靠的，虽然绝大部分情况下，网络通信能够接收到成功或失败的响应，但当网络出现异常的情况下，就会出现超时现象，通常有以下两种情况：

1. 由于网络原因，该请求并没有被成功的发送到接收方，而是在发送过程就发生了丢失现象。
2. 该请求成功的被接收方接收后，并进行了处理，但在响应反馈给发送方过程中，发生了消息丢失现象。

节点故障

节点故障是分布式系统下另一个比较常见的问题，指的是组成分布式系统的服务器节点出现的宕机或"僵死"现象，根据经验来说，每个节点都有可能出现故障，并且经常发生

分布式理论：一致性概念

分布式一致性是一个相当重要且被广泛探索与论证的问题，首先来看三种业务场景

1、火车站售票

假如说我们的终端用户是一位经常坐火车的旅行家，通常他是去车站的售票处购买车票，然后拿着车票去检票口，再坐上火车，开始一段美好的旅行。

想象一下，如果他选择的目的地是杭州，而某一趟开往杭州的火车只剩下最后一张车票，可能在同一时刻，不同售票窗口的另一位乘客也购买了同一张车票。假如说售票系统没有进行一致性的保障，两人都购票成功了。而在检票口检票的时候，其中一位乘客会被告知他的车票无效----当然，现代的中国铁路售票系统已经很少出现这样的问题了。但在这个例子中我们可以看出，终端用户对于系统的需求非常简单：

"请售票给我，如果没有余票了，请在售票的时候就告诉我票是无效的"

这就对购票系统提出了严格的一致性要求----系统的数据（本例中指的就是那趟开往杭州的火车的余票数）无论在哪个售票窗口，每时每刻都必须是准确无误的！

2、银行转账

假如我们的终端用户是一位刚毕业的大学生，通常在拿到第一个月工资的时候，都会选择向家里汇款。当他来到银行柜台，完成转账操作后，银行的柜台人员会友善地提醒他："您的转账将在N个工作日后到账！"。此时这名毕业生有一定的沮丧，会对那名柜台人员叮嘱："好吧，多久没关系，钱不要少就好了！"----这也成为了几乎所有用户对于现代银行系统最基本的需求

3、网上购物

假如说我们的终端用户是一位网购达人，当他看见一件库存量为5的心仪商品，会迅速地确认购买，写下收货地址，然后下单----然而，在下单的那个瞬间，系统可能会告知该用户："库存量不足！"。此时绝大部分消费者都会抱怨自己动作太慢，使得心爱的商品被其他人抢走了。

但其实在商品详情页上显示的那个库存量，通常不是该商品的真实库存量，只有在真正下单购买的时候，系统才会检查该商品的真实库存量。但是，谁在意呢？

问题的解读

对于上面三个例子，相信大家一定看出来了，我们的终端用户在使用不同的计算机产品时对于数据一致性的需求是不一样的：

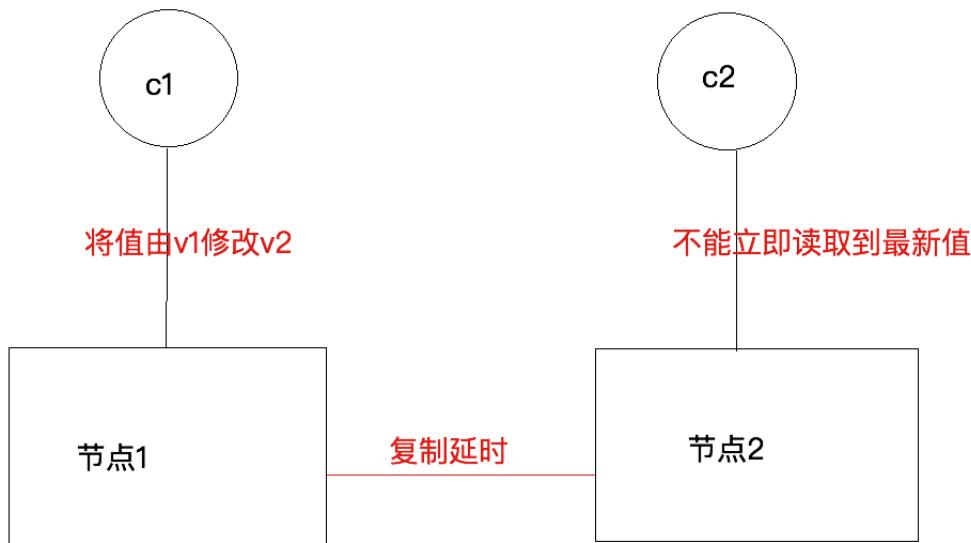
1、有些系统，既要快速地响应用户，同时还要保证系统的数据对于任意客户端都是真实可靠的，就像火车站售票系统

2、有些系统，需要为用户保证绝对可靠的数据安全，虽然在数据一致性上存在延时，但最终务必保证严格的一致性，就像银行的转账系统

3、有些系统，虽然向用户展示了一些可以说是"错误"的数据，但是在整个系统使用过程中，一定会在某一个流程上对系统数据进行准确无误的检查，从而避免用户发生不必要的损失，就像网购系统

分布式一致性的提出

在分布式系统中要解决的一个重要问题就是数据的复制。在我们的日常开发中，相信有很多同学都遇到过这样的问题：假设客户端C1将系统中的一个值K由V1更新为V2，但客户端C2无法立即读取到K的最新值，需要在一段时间之后才能读取到。这个例子就是常见的数据库之间复制的延时问题，因为数据库数据复制之间存在延时。



分布式系统对于数据的复制需求一般都来自于以下两个原因：

1、为了增加系统的可用性，以防止单点故障引起的系统不可用

2、提高系统的整体性能，通过负载均衡技术，能够让分布在不同地方的数据副本都能够为用户提供服务

数据复制在可用性和性能方面给分布式系统带来的巨大好处是不言而喻的，然而数据复制所带来的一致性挑战，也是每一个开发人员不得不面对的。

所谓分布式一致性问题，是指在分布式环境中引入数据复制机制之后，不同数据节点之间可能出现的，并无法依靠计算机应用程序自身解决的数据不一致的情况。简单讲，数据一致性就是指在对一个副本数据进行更新的时候，必须确保也能够更新其他的副本，否则不同副本之间的数据将不一致。

那么如何解决这个问题？一种思路是"既然是由于延时动作引起的问题，那我可以将写入的动作阻塞，直到数据复制完成后，才完成写入动作"。没错，这似乎能解决问题，而且有一些系统的架构也确实直接使用了这个思路。但这个思路在解决一致性问题的同时，又带来了新的问题：写入的性能。如果你的应用场景有非常多的写请求，那么使用这个思路之后，后续的写请求都将会阻塞在前一个请求的写操作上，导致系统整体性能急剧下降。

总的来说，我们无法找到一种能够满足分布式系统所有系统属性的分布式一致性解决方案。因此，如何既保证数据的一致性，同时又不影响系统运行的性能，是每一个分布式系统都需要重点考虑和权衡的。于是，一致性级别由此诞生：

1、强一致性

这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大

2、弱一致性

这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态

3、最终一致性

最终一致性是弱一致性的一个特例，系统会保证在一定时间内，能够达到一个数据一致的状态。这里之所以将最终一致性单独提出来，是因为它是弱一致性中非常推崇的一种一致性模型，也是业界在大型分布式系统的数据一致性上比较推崇的模型

分布式事务

在单机数据库中，我们很容易能够实现一套满足ACID特性的事务处理系统，但在分布式数据库中，数据分散在各台不同的机器上，如何对这些数据进行分布式的事务处理具有非常大的挑战。

分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于分布式系统的不同节点上，通常一个分布式事务中会涉及对多个数据源或业务系统的操作。

可以设想一个最典型的分布式事务场景：一个跨银行的转账操作涉及调用两个异地的银行服务，其中一个是本地银行提供的取款服务，另一个则是目标银行提供的存款服务，这两个服务本身是无状态并且相互独立的，共同构成了一个完整的分布式事务。如果从本地银行取款成功，但是因为某种原因存款服务失败了，那么就必须回滚到取款之前的状态，否则用户可能会发现自己的钱不翼而飞了。

从这个例子可以看到，一个分布式事务可以看做是多个分布式的操作序列组成的，例如上面例子的取款服务和存款服务，通常可以把这一系列分布式的操作序列称为子事务。因此，分布式事务也可以被定义为一种嵌套型的事务，同时也具有了ACID事务特性。但由于在分布式事务中，各个子事务的执行是分布式的，因此要实现一种能够保证ACID特性的分布式事务处理系统就显得格外复杂，尤其是对于一个高访问量，高并发的互联网分布式系统来说，如果我们期望实现一套严格满足ACID特性的分布式事务，很可能出现的情况就是在系统的可用性和严格一致性之间出现冲突 ----- 因为当我们要求分布式系统具有严格一致性时，很可能就需要牺牲掉系统的可用性。但毋庸置疑的一点是，可用性又是一个消费者不允许我们讨价还价的系统属性，比如像淘宝这样的在线购物网站，就要求7x24小时不间断地对外提供服务，而对于一致性，则更加是所有消费者对于一个软件的刚需。因此，在可用性和一致性之间永远无法存在一个两全其美的方案，于是如何构建一个兼顾可用性和一致性的分布式系统成为了无数开发人员探讨的难题，于是就出现了CAP和BASE这样的分布式系统经典理论

分布式理论：CAP定理

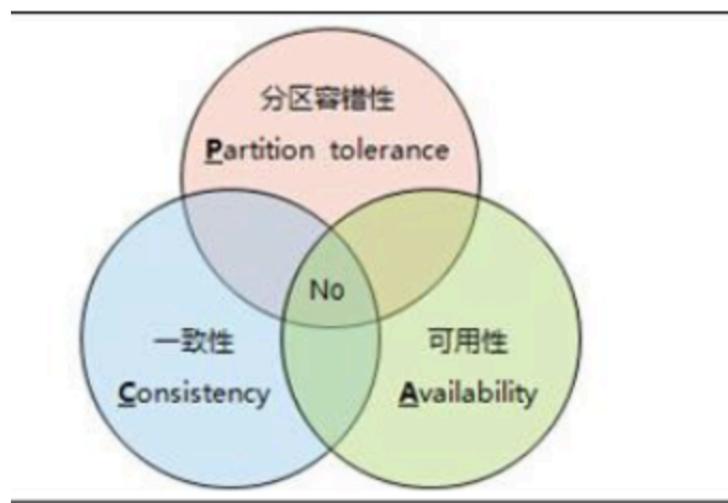
什么是 CAP 定理

2000 年 7 月的时候，加州大学伯克利分校的 Eric Brewer 教授提出了 CAP 猜想，2 年后，被来自麻省理工的 Seth Gilbert 和 Nancy Lynch 从理论上证明了猜想的可能性，从此，CAP 定理正式在学术上成为了分布式计算领域的公认定理。并深深的影响了分布式计算的发展。

CAP 理论告诉我们，一个分布式系统不可能同时满足一致性 (C: Consistency)，可用性 (A: Availability) 和分区容错性 (P: Partition tolerance) 这三个基本需求，最多只能同时满足其中的 2 个

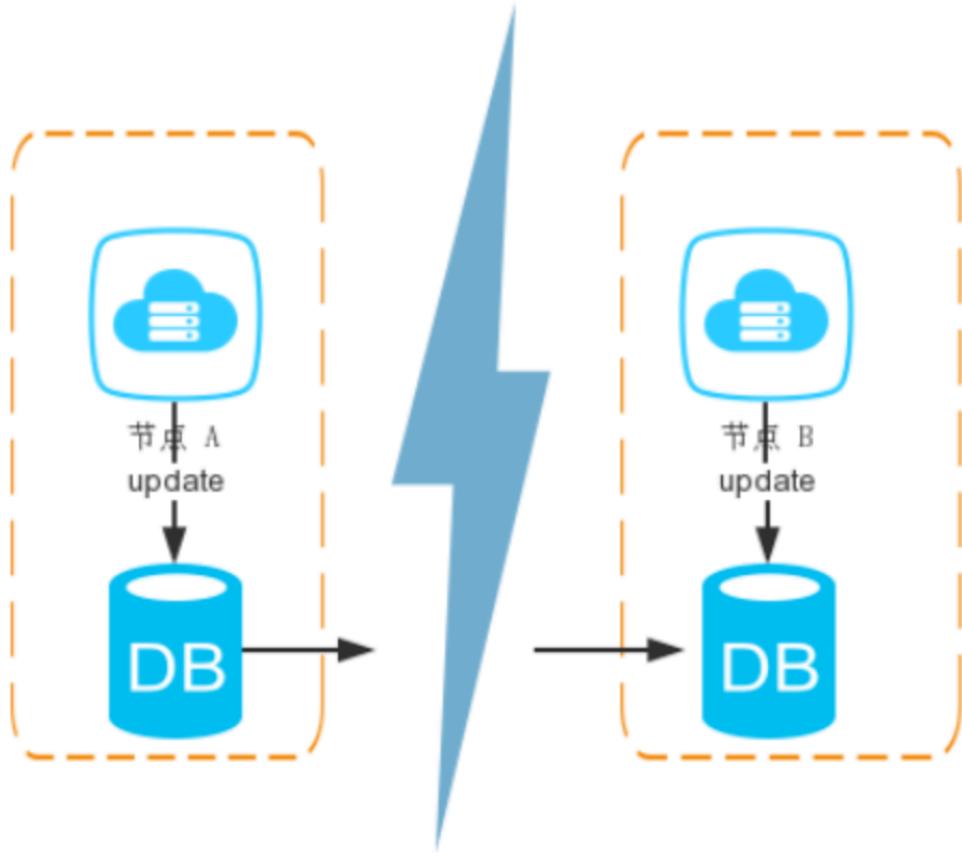
选项	描述
C (Consistence)	一致性，分布式环境中，数据在多个副本之间能够保持一致的特性（严格的一致性）。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处在一致的状态
A (Availability)	可用性，指系统提供的服务必须一直处于可用的状态，每次请求都能获取到非错的响应——但是不保证获取的数据为最新数据。
P (Partition tolerance)	分区容错性，分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。

为什么只能 3 选 2



首先，能不能同时满足这三个条件？

假设有一个系统如下：



整个系统由两个节点配合组成，之间通过网络通信，当节点 A 进行更新数据库操作的时候，需要同时更新节点 B 的数据库（这是一个原子的操作）。

上面这个系统怎么满足 CAP 呢？C：当节点A更新的时候，节点B也要更新，A：必须保证两个节点都是可用的，P：当节点 A,B 出现了网络分区，必须保证对外可用。

可见，根本完成不了同时满足CAP，因为只要出现了网络分区，C 就无法满足，因为节点 A 根本连接不上节点 B。如果强行满足 C 一致性，就必须停止服务运行，从而放弃可用性 A。

所以，最多满足两个条件：

组合	分析结果
CA	满足一致性和可用性，放弃分区容错。说白了，就是一个整体的应用，如果希望能够避免系统出现分区容错性问题，一种较为简单做法是将所有的数据（或者仅仅是那些与事务相关的数据）都放在一个分布式节点上。这样做虽然无法100%保证系统不会出错，但至少不会碰到由于网络分区带来的负面影响。但同时需要注意的是，放弃P的同时也就意味着放弃了系统的可扩展性
CP	满足一致性和分区容错性，一旦系统遇到网络分区或其他故障或为了保证一致性时，放弃可用性，那么受到影响的服务需要等待一定的时间，因此在等待期间系统无法对外提供正常的服务，即不可用
AP	满足可用性和分区容错性，当出现网络分区，同时为了保证可用性，必须让节点继续对外服务，这样必然导致失去一致性。这里所说的放弃一致性，并不是完全不需要数据一致性，如果真是这样的话，那系统数据就没有意义了，这里实际上指的是放弃数据的强一致性，而保留数据的最终一致性。这样的系统无法保证数据保持实时的一致性，但是能够承诺的是，数据最终会达到一个一致的状态。这就引入了一个时间窗口的概念，具体多久能够达到数据一致取决于系统的设计，主要包括数据副本在不同节点之间的复制时间长短

从CAP定理可以看出，一个分布式系统不可能同时满足一致性，可用性和分区容错性这三个基本需求，最多只能同时满足其中的2个，还需要明确的一点是：对于一个分布式系统而言，分区容错性可以说是一个最基本的要求。因为既然是一个分布式系统，那么分布式系统中的组件必然需要被部署到不同的节点，否则也就没有所谓的分布式系统了，因此必然出现子网络。而对于分布式系统而言，网络问题又是一个必定会出现的异常情况，因此分区容错性也就成了一个分布式系统必然需要面对和解决的问题。因此架构师往往需要把精力花在如何根据业务特点在C（一致性）和A（可用性）之间寻求平衡

能不能解决 3 选 2 的问题

想要解决3选2的问题，首先大家需要思考分区是百分之百出现的吗？如果不出现分区，那么就能够同时满足CAP。如果出现了分区，可以根据策略进行调整。比如C不必使用那么强的一致性，可以先将数据存起来，稍后再更新，实现所谓的“最终一致性”。

基于这个思路，也引出了第二个理论 Base 理论

分布式理论：BASE 理论

在CAP定理中，介绍了CAP不可能同时满足，而分区容错是对于分布式系统而言，是必须的。但如果系统能够同时实现CAP是再好不过的了，所以出现了BASE理论

什么是BASE理论

BASE：全称：Basically Available(基本可用), Soft state (软状态), 和 Eventually consistent (最终一致性) 三个短语的缩写，来自 ebay 的架构师提出。

Base 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大型互联网分布式实践的总结，是基于 CAP 定理逐步演化而来的。

其核心思想是：既然无法做到强一致性（Strong consistency），但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）

接下来，我们着重对BASE中的三要素进行讲解。

Basically Available(基本可用)

什么是基本可用呢？

基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性——但请注意，这绝不等价于系统不可用。以下就是两个“基本可用”的例子

- **响应时间上的损失：**正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。
- **功能上的损失：**正常情况下，在一个电子商务网站（比如淘宝）上购物，消费者几乎能够顺利地完成每一笔订单。但在一些节日大促购物高峰的时候（比如双十一、双十二），由于消费者的购物行为激增，为了保护系统的稳定性（或者保证一致性），部分消费者可能会被引导到一个降级页面，如下：



Soft state (软状态)

什么是软状态呢？相对于一致性，要求多个节点的数据副本都是一致的，这是一种“硬状态”。

软状态指的是：允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统在多个不同节点的数据副本之间进行数据同步的过程中存在延迟

Eventually consistent (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，最终能够达到一个一致的状态。因此最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性

在实际工程实践中，最终一致性存在以下五类主要变种：

1. 因果一致性 (Causal consistency)

指的是：如果节点 A 在更新完某个数据后通知了节点 B，那么节点 B 之后对该数据的访问和修改都是基于 A 更新后的值。于此同时，和节点 A 无因果关系的节点 C 的数据访问则没有这样的限制。

2. 读己之所写 (Read your writes)

这种就很简单了，节点 A 更新一个数据后，它自身总是能访问到自身更新过的最新值，而不会看到旧值。也就是说，对于单个数据获取者来说，其读取到的数据，一定不会比上次写入的值旧，因此，读己之所写也可以看成是一种特殊的因果一致性

3. 会话一致性 (Session consistency)

会话一致性将对系统数据的访问过程框定在了一个会话当中：系统能保证在同一个有效的会话中实现“读己之所写”的一致性，也就是说，执行更新操作之后，客户端能够在同一个会话中始终读取到该数据项的最新值。

4. 单调读一致性 (Monotonic read consistency)

单调读一致性是指如果一个节点从系统中读取出一个数据项的某个值后，那么系统对于该节点后续的任何数据访问都不应该返回更旧的值。

5. 单调写一致性 (Monotonic write consistency)

指一个系统要能够保证来自同一个节点的写操作被顺序的执行。

以上就是最终一致性的五类常见的变种，在实际系统实践中，可以将其中的若干个变种互相结合起来，以构建一个具有最终一致性特性的分布式系统

总的来说，BASE 理论面向的是大型高可用可扩展的分布式系统，和传统事务的 ACID 是相反的，它完全不同于 ACID 的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间是不一致的，但最终要保证数据一致。

分布式理论：一致性协议 2PC

为了使系统尽量能够达到 CAP，于是有了 BASE 协议，而 BASE 协议是在可用性和一致性之间做的取舍和妥协。

也就是说，我们在对分布式系统进行架构设计的过程中，往往需要我们在系统的可用性和数据一致性之间反复的权衡。于是，就涌现了很多经典的算法和协议，最著名的几种就是二阶段提交协议，三阶段提交协议，Paxos 算法等。接下来先重点介绍二阶段提交协议，简称 2PC

什么是 2PC

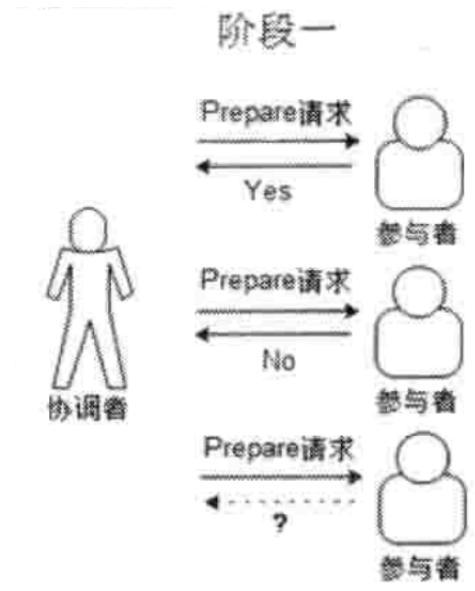
在分布式系统中，会有多个机器节点，每一个机器节点虽然能够明确地知道自己在进行事务操作过程中的结果是成功或失败，但无法直接获取到其他分布式节点的操作结果，因此，当一个事务操作需要跨越多个分布式节点的时候，为了保证事务处理的ACID特性，就需要引入一个“协调者”的组件来统一调度所有分布式节点的执行逻辑，这些被调度的节点则称为“参与者”，协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务真正进行提交。基于这个思想，就衍生了二阶段提交和三阶段提交两种协议。

2PC，是 Two-Phase Commit 缩写，即两阶段提交。是计算机网络，尤其是数据库领域中，为了使基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法，通常，2pc也被认为是一种一致性协议，用来保证分布式系统数据的一致性，目前，绝大部分的关系型数据库都是采用二阶段提交协议来完成分布式事务处理的，利用该协议能够非常方便地完成所有分布式事务参与者的协调，统一决定事务的提交和回滚，从而能够有效的保证分布式数据的一致性

协议说明：

顾名思义，二阶段提交就是将事务的提交过程分成了两个阶段来进行处理。流程如下：

2PC 阶段一：提交事务请求



1. 事务询问

协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应。

2. 执行事务

各参与者节点执行事务操作，并将 Undo 和 Redo 信息记入事务日志中

(Redo用来保证事务的原子性和持久性，Undo能保证事务的一致性，两者也是系统恢复的基础前提)

3. 各参与者向协调者反馈事务询问的响应

如果参与者成功执行了事务操作，那么就反馈给协调者 Yes 响应，表示事务可以执行；如果参与者没有成功执行事务，就返回 No 给协调者，表示事务不可以执行。

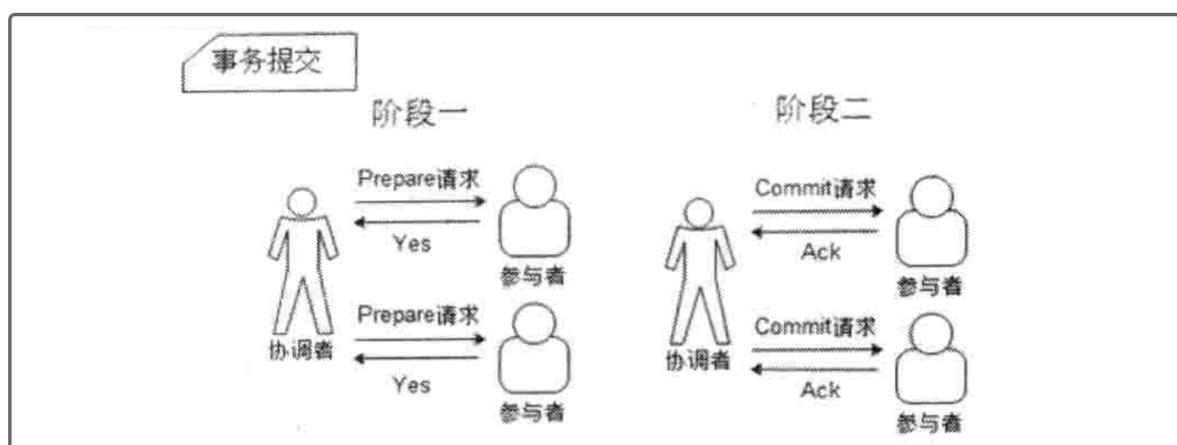
由于上面的内容在形式上近似是协调者组织各参与者对一次事务操作的投票表态过程，因此二阶段提交协议的阶段一也被称为“投票阶段”，即各参与者投票表明是否要继续执行接下去的事务提交操作

2PC 阶段二：执行事务提交

在阶段二中，就会根据阶段一的投票结果来决定最终是否可以进行事务提交操作，正常情况下，包含两种操作可能：提交事务，中断事务。

提交事务步骤如下：

假如协调者从所有的参与者获得的反馈都是yes响应，那么就会执行事务提交



1. 发送提交请求：

协调者向所有参与者发出 commit 请求。

2. 事务提交：

参与者收到 commit 请求后，会正式执行事务提交操作，并在完成提交之后释放整个事务执行期间占用的事务资源。

3. 反馈事务提交结果：

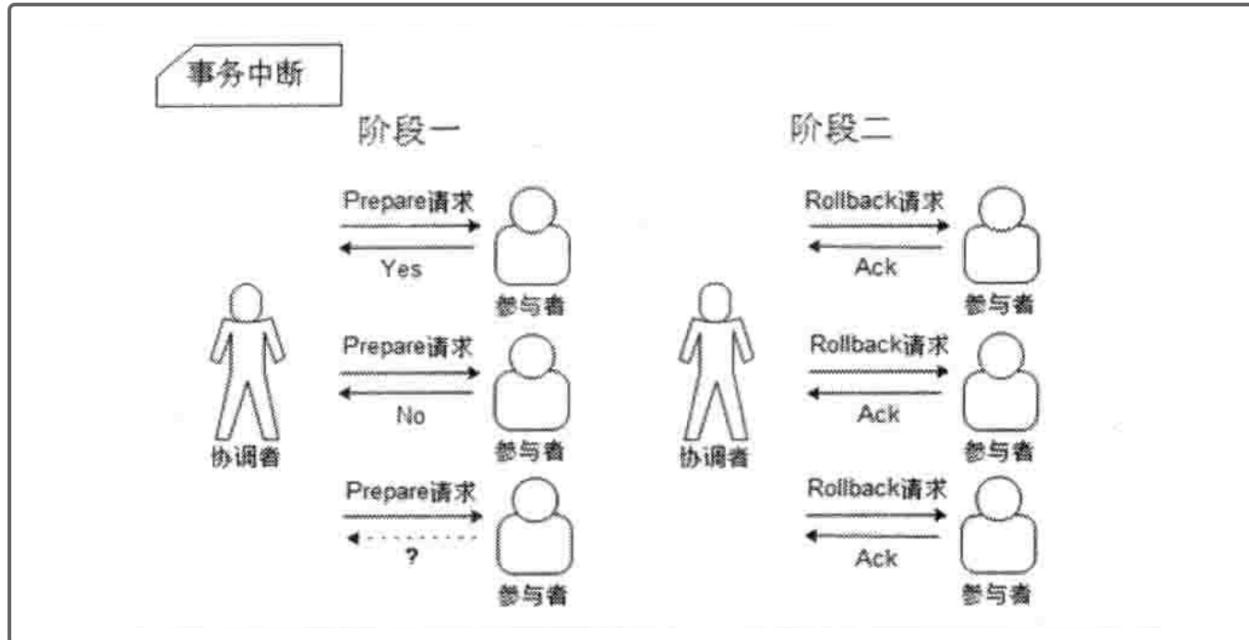
参与者在完成事务提交之后，向协调者发送 Ack 信息。

4. 完成事务：

协调者接收到所有参与者反馈的 Ack 信息后，完成事务。

中断事务步骤如下：

假如任何一个参与者向协调者反馈了No响应，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务



1. 发送回滚请求：

协调者向所有参与者发出 Rollback 请求。

2. 事务回滚：

参与者接收到 Rollback 请求后，会利用其在阶段一中记录的 Undo 信息来执行事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源。

3. 反馈事务回滚结果：

参与者在完成事务回滚之后，向协调者发送 Ack 信息。

4. 中断事务：

协调者接收到所有参与者反馈的 Ack 信息后，完成事务中断。

从上面的逻辑可以看出，二阶段提交就做了2个事情：投票，执行。

将一个事务的处理过程分为了投票和执行两个阶段，核心是对每个事务都采用先尝试后提交的处理方式，从而保证其原子性和一致性，因此也可以将二阶段提交看成一个强一致性的算法。I

2PC 优点缺点

优点

原理简单，实现方便

缺点

同步阻塞，单点问题，数据不一致，过于保守

- 同步阻塞：

二阶段提交协议存在最明显也是最大的一个问题就是同步阻塞，在二阶段提交的执行过程中，所有参与该事务操作的逻辑都处于阻塞状态，也就是说，各个参与者在等待其他参与者响应的过程中，无法进行其他操作。这种同步阻塞极大的限制了分布式系统的性能。

- 单点问题：

协调者在整个二阶段提交过程中很重要，如果协调者在提交阶段出现问题，那么整个流程将无法运转，更重要的是：其他参与者将会处于一直锁定事务资源的状态中，而无法继续完成事务操作。

- 数据不一致：

假设当协调者向所有的参与者发送 commit 请求之后，发生了局部网络异常或者是协调者在尚未发送完所有 commit 请求之前自身发生了崩溃，导致最终只有部分参与者收到了 commit 请求。这将导致严重的数据不一致问题。

- 过于保守：

如果在二阶段提交的提交询问阶段中，参与者出现故障而导致协调者始终无法获取到所有参与者的响应信息的话，这时协调者只能依靠其自身的超时机制来判断是否需要中断事务，显然，这种策略过于保守。换句话说，二阶段提交协议没有设计较为完善的容错机制，任意一个节点失败都会导致整个事务的失败。

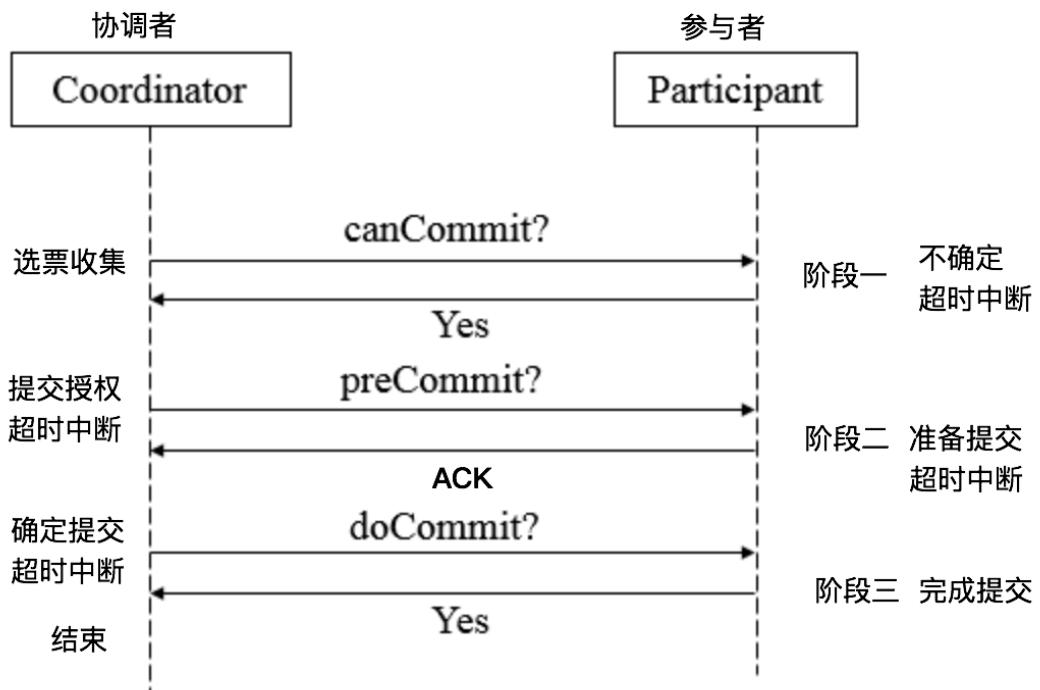
分布式理论：一致性协议 3PC

刚刚讲解了二阶段提交协议的设计和实现原理，并明确指出了其在实际运行过程中可能存在的诸如同步阻塞，单点问题，数据不一致，过于保守的容错机制等缺陷

而为了弥补二阶段提交的缺点，研究者们在2PC的基础上，提出了三阶段提交协议，简称3PC

什么是三阶段提交

3PC，全称“three phase commit”，是2PC的改进版，将2PC的“提交事务请求”过程一分为二，共形成了由CanCommit、PreCommit和doCommit三个阶段组成的事务处理协议。



阶段一：CanCommit

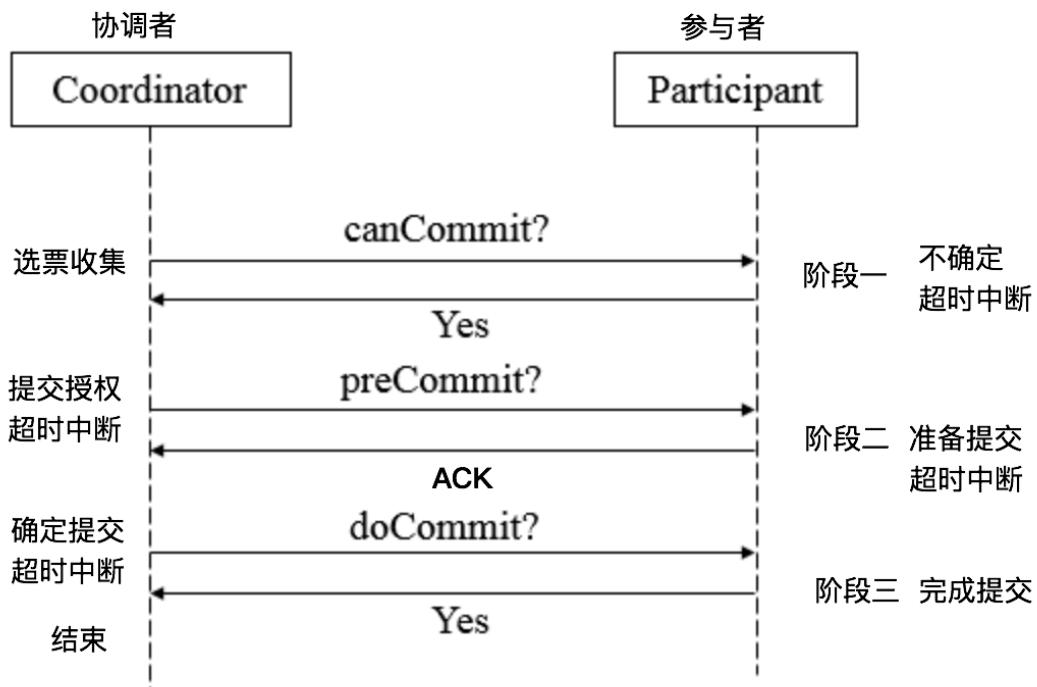
第一个阶段：CanCommit

① 事务询问

协调者向所有的参与者发送一个包含事务内容的canCommit请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应。

② 各参与者向协调者反馈事务询问的响应

参与者在接收到来自协调者的包含了事务内容的canCommit请求后，正常情况下，如果自身认为可以顺利执行事务，则反馈Yes响应，并进入预备状态，否则反馈No响应。



阶段二：PreCommit

协调者在得到所有参与者的响应之后，会根据结果有2种执行操作的情况：执行事务预提交，或者中断事务

假如所有参与反馈的都是Yes，那么就会执行事务预提交。

1. 执行事务预提交分为 3 个步骤

① 发送预提交请求：

协调者向所有参与者节点发出`preCommit`请求，并进入prepared阶段。

② 事务预提交：

参与者接收到`preCommit`请求后，会执行事务操作，并将Undo和Redo信息记录到事务日志中。

③ 各参与者向协调者反馈事务执行的结果：

若参与者成功执行了事务操作，那么反馈Ack，同时等待最终的指令：提交（commit）或终止（abort）。

若任一参与者反馈了No响应，或者在等待超时后，协调者尚无法接收到所有参与者反馈，则中断事务

2. 中断事务也分为2个步骤：

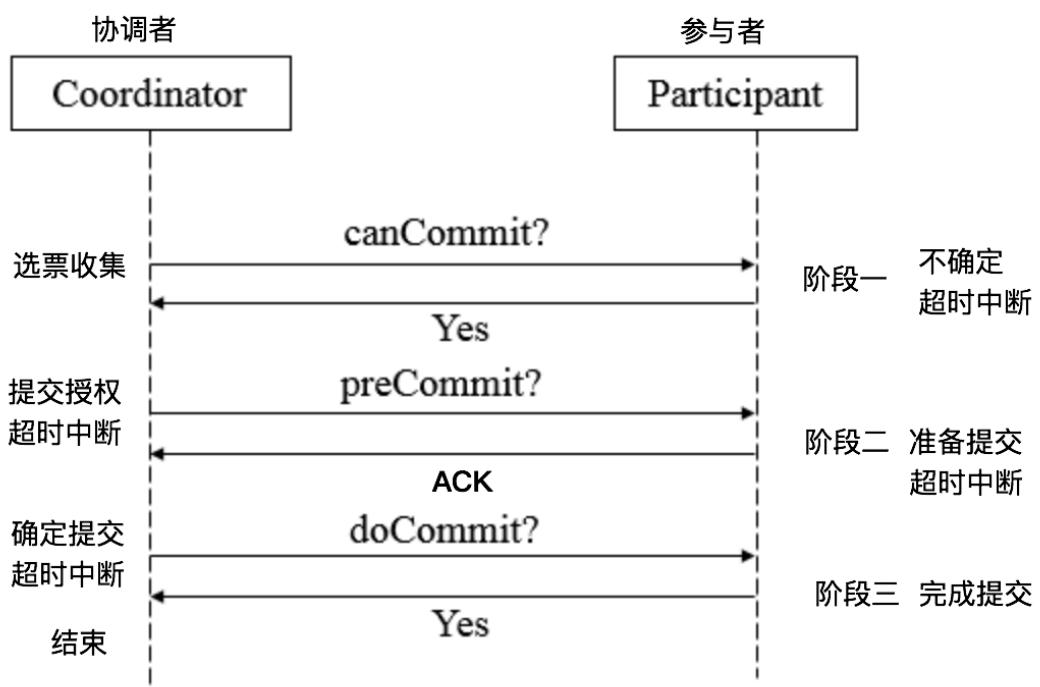
① 发送中断请求：

协调者向所有参与者发出`abort`请求。

② 中断事务：

无论是收到来自协调者的abort请求或者等待协调者请求过程中超时，参与者都会中断事务

阶段三：do Commit



该阶段做真正的事务提交或者完成事务回滚，所以就会出现两种情况：

1. 执行事务提交

① 发送提交请求：

进入这一阶段，假设协调者处于正常工作状态，并且它接收到了来自所有参与者的Ack响应，那么他将从预提交状态转化为提交状态，并向所有的参与者发送doCommit请求。

② 事务提交：

参与者接收到doCommit请求后，会正式执行事务提交操作，并在完成提交之后释放整个事务执行过程中占用的事务资源。

③ 反馈事务提交结果：

参与者在完成事务提交后，向协调者发送Ack响应。

④ 完成事务：

协调者接收到所有参与者反馈的Ack消息后，完成事务。

2. 中断事务

① 发送中断请求：协调者向所有的参与者节点发送abort请求。

② 事务回滚：参与者收到abort请求后，会根据记录的Undo信息来执行事务回滚，并在完成回滚之后释放整个事务执行期间占用的资源。

③ 反馈事务回滚结果：参与者在完成事务回滚后，向协调者发送Ack消息。

④ 中断事务：协调者接收到所有参与者反馈的Ack消息后，中断事务。

注意：一旦进入阶段三，可能会出现 2 种故障：

1. 协调者出现问题
2. 协调者和参与者之间的网络故障

如果出现了任一一种情况，最终都会导致参与者无法收到 doCommit 请求或者 abort 请求，针对这种情况，参与者都会在等待超时之后，继续进行事务提交

3PC 优点缺点

优点

相比较 2PC，最大的优点就是降低了参与者的阻塞范围（第一个阶段是不阻塞的），其次能够在单点故障后继续达成一致（2PC 在提交阶段会出现此问题，而 3PC 会根据协调者的状态进行回滚或者提交）。

缺点

如果参与者收到了 preCommit 消息后，出现了网络分区，此时协调者所在的节点和参与者所在的节点无法进行正常的网络通信，那么参与者等待超时后，会进行事务的提交，这必然会出现分布式数据不一致的问题。

2PC对比3PC

首先对于协调者和参与者都设置了超时机制（在2PC中，只有协调者拥有超时机制，即如果在一定时间内没有收到参与者的消息则默认失败）。其次在2PC的准备阶段和提交阶段之间，插入预提交阶段，使3PC拥有CanCommit、PreCommit、DoCommit三个阶段。PreCommit是一个缓冲，保证了在最后提交阶段之前各参与节点的状态是一致的。

分布式理论：一致性算法 Paxos

上面已经对二阶段和三阶段提交协议进行了学习，并了解了它们各自的特点以及解决的分布式问题，接下来，我们重点讲解另一种非常重要的分布式一致性算法：Paxos

什么是Paxos算法

Paxos 算法是 莱斯利* 兰伯特 (Leslie Lamport) 于1990年提出的一种基于消息传递且具有高度容错特性的一致性算法，是目前公认的解决分布式一致性问题最有效的算法之一，Paxos主要用来解决分布式系统中，如何就某个值达成一致的算法1

Google Chubby的作者Mike Burrows说过这个世界上只有一种一致性算法，那就是Paxos，其它的算法都是残次品，虽然说得有点夸张，但是至少说明了Paxos算法的地位，然而，它晦涩难懂的程度完全可以跟它的重要程度相匹敌

Paxos有点类似我们之前说的2PC, 3PC，但是解决了他们俩的各种硬伤。该算法在很多大厂都得到了工程实践，比如阿里的OceanBase的分布式数据库，底层就是使用的paxos算法。再比如Google的chubby分布式锁也是用的这个算法。可见该算法在分布式系统中的地位，甚至于，paxos就是分布式一致性的代名词

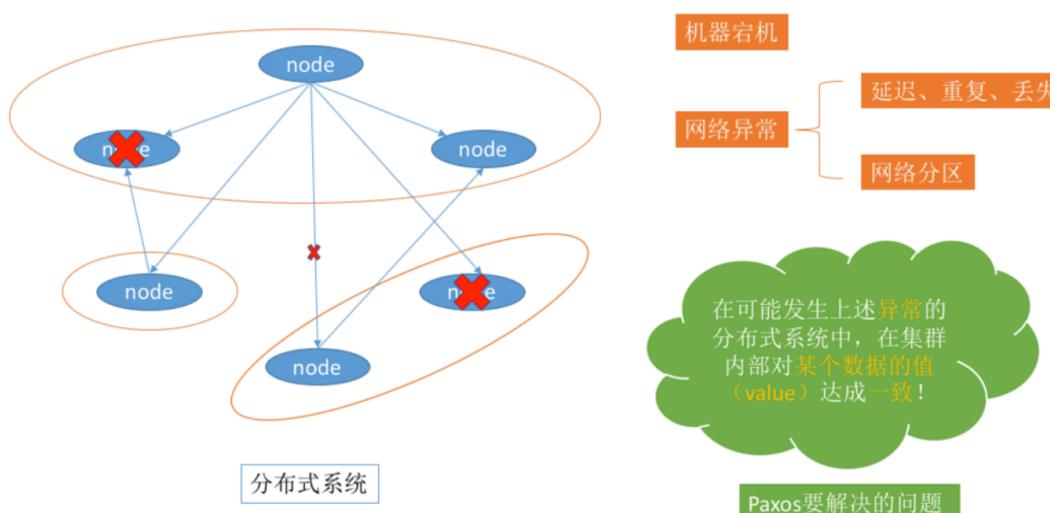
Paxos 解决了什么问题

答：解决了一致性问题。

在常见的分布式系统中，总会发生诸如机器宕机或网络异常（包括消息的延迟、丢失、重复、乱序，还有网络分区）等情况。Paxos算法需要解决的问题就是如何在一个可能发生上述异常的分布式系统中，快速且正确地在集群内部对某个数据的值达成一致，并且保证不论发生以上任何异常，都不会破坏整个系统的一致性。

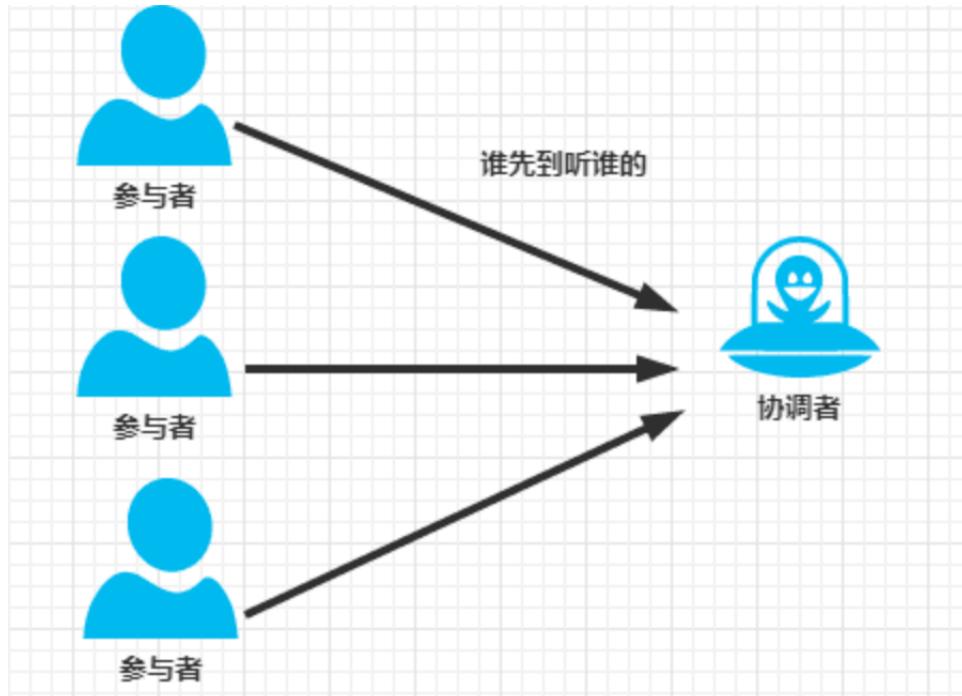
注：这里某个数据的值并不只是狭义上的某个数，它可以是一条日志，也可以是一条命令（command）。。。根据应用场景不同，某个数据的值有不同的含义

背景



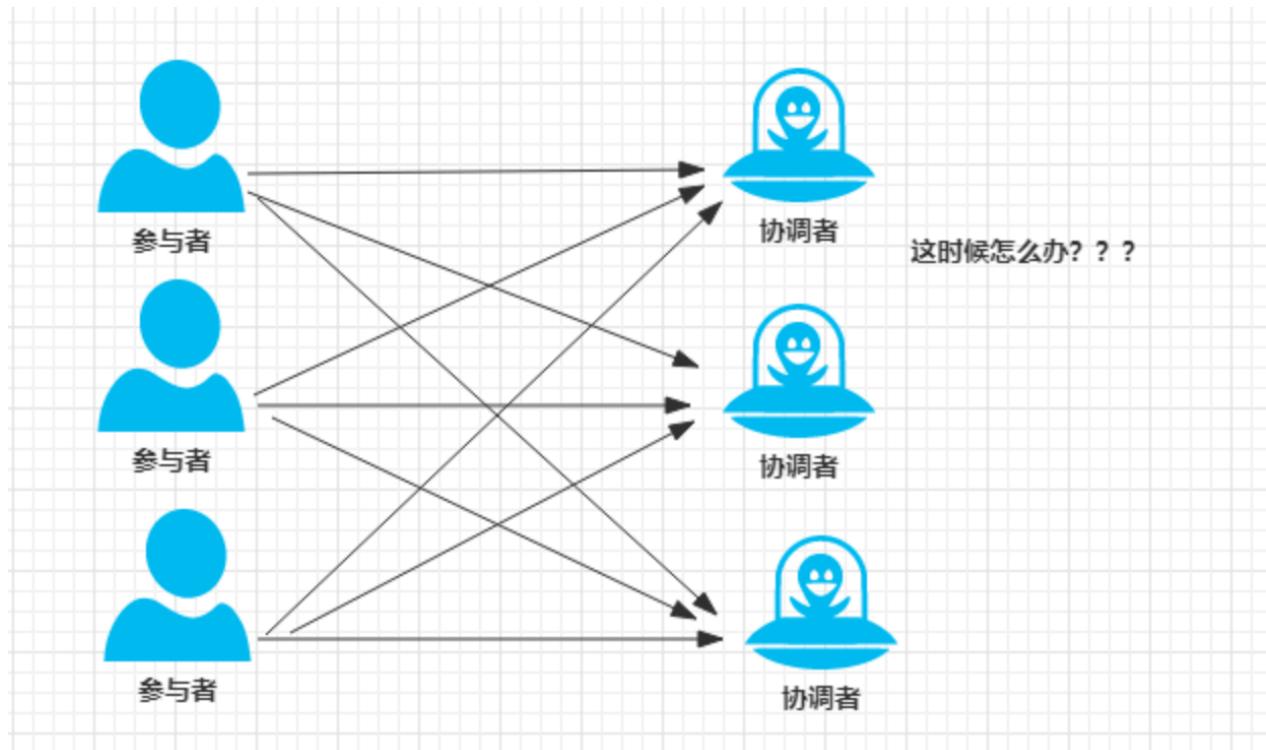
我们假设一种情况，在一个集群环境中，要求所有机器上的状态是一致的，其中有2台机器想修改某个状态，机器A想把状态改为A，机器B想把状态改为B，那么到底听谁的呢？

有的同学会想到，可以像2PC, 3PC一样引入一个协调者，谁先到，听谁的



但是如果，协调者宕机了呢？

所以需要对协调者也做备份，也要做集群。这时候，问题来了，这么多协调者，听谁的呢？



Paxos 算法就是为了解决这个问题而生的

Paxos相关概念

首先一个很重要的概念叫提案（Proposal）。最终要达成一致的value就在提案里。

提案 (Proposal): Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)

在Paxos算法中，有三种角色：

- **Proposer**: 提案发起者
- **Acceptor**: 决策者，可以批准提案
- **Learner**: 最终决策的学习者

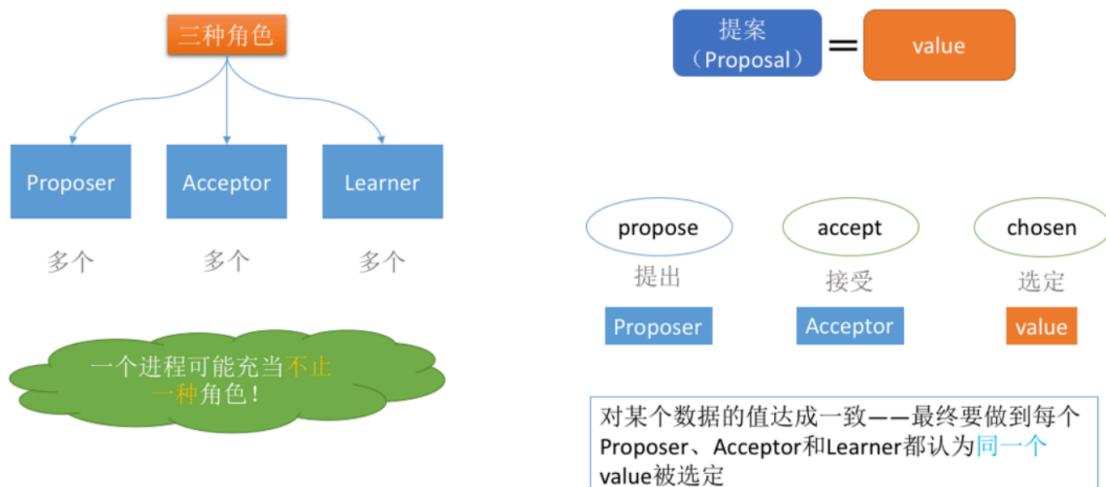
Proposer可以提出 (propose) 提案；Acceptor可以接受 (accept) 提案；如果某个提案被选定 (chosen)，那么该提案里的value就被选定了。

在具体的实现中，一个进程可能同时充当多种角色。比如一个进程可能既是**Proposer**又是**Acceptor**又是**Learner**。

回到刚刚说的『对某个数据的值达成一致』，指的是**Proposer**、**Acceptor**、**Learner**都认为同一个 value被选定 (chosen)。那么，**Proposer**、**Acceptor**、**Learner**分别在什么情况下才能认为某个 value被选定呢？

- Proposer: 只要Proposer发的提案被Acceptor接受（刚开始先认为只需要一个Acceptor接受即可，在推导过程中会发现需要半数以上的Acceptor同意才行），Proposer就认为该提案里的value被选定了。
- Acceptor: 只要Acceptor接受了某个提案，Acceptor就认为该提案里的value被选定了。
- Learner: Acceptor告诉Learner哪个value被选定，Learner就认为那个value被选定。

相关概念



问题描述

假设有一组可以提出提案的进程集合，那么对于一个一致性算法需要保证以下几点：

- 在这些被提出的提案中，只有一个会被选定
- 如果没有提案被提出，就不应该有被选定的提案。
- 当一个提案被选定后，那么所有进程都应该能学习 (**learn**) 到这个被选定的value

推导过程

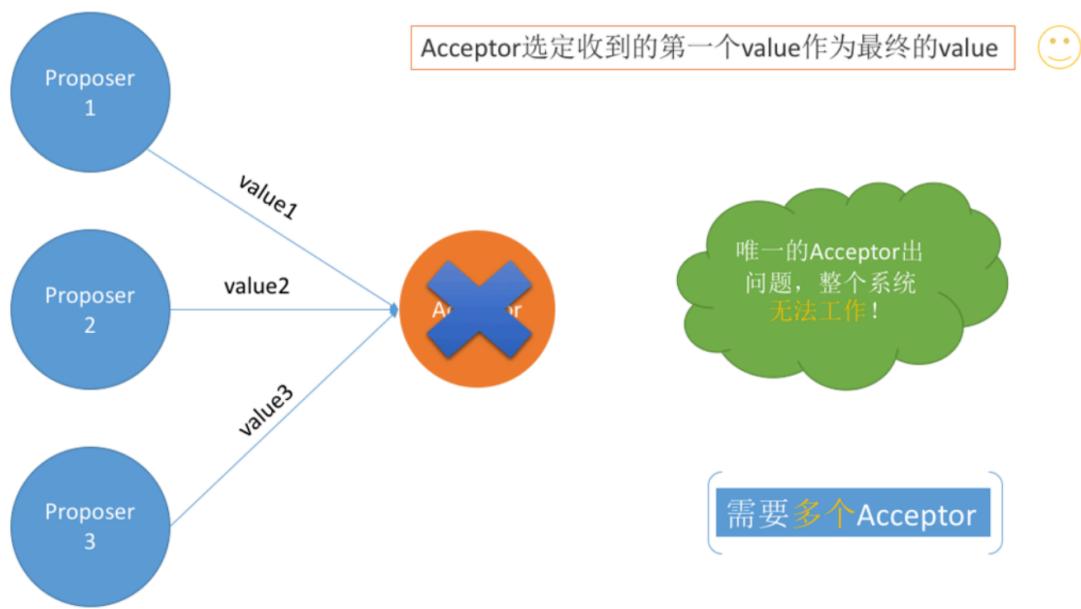
最简单的方案——只有一个Acceptor

假设只有一个Acceptor（可以有多个Proposer），只要Acceptor接受它收到的第一个提案，则该提案被选定，该提案里的value就是被选定的value。这样就保证只有一个value会被选定。

但是，如果这个唯一的Acceptor宕机了，那么整个系统就无法工作了！

因此，必须要有多个Acceptor！

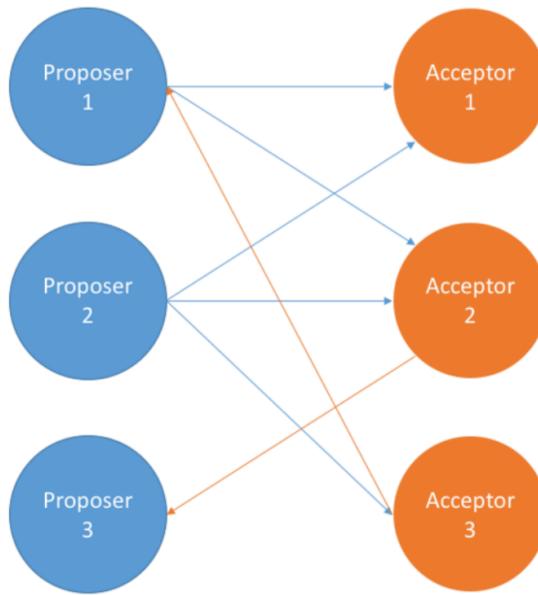
最简单方案——只有一个Acceptor



多个Acceptor

多个Acceptor的情况如下图。那么，如何保证在多个Proposer和多个Acceptor的情况下选定一个value呢？

多个Acceptors如何选定一个值



如何保证在多个
Proposer和
Acceptor的情况下
选定一个值？

下面开始寻找解决方案。

首先我们希望即使只有一个Proposer提出了一个value，该value也最终被选定。

那么，就得到下面的约束：

P1：一个Acceptor必须接受它收到的第一个提案。

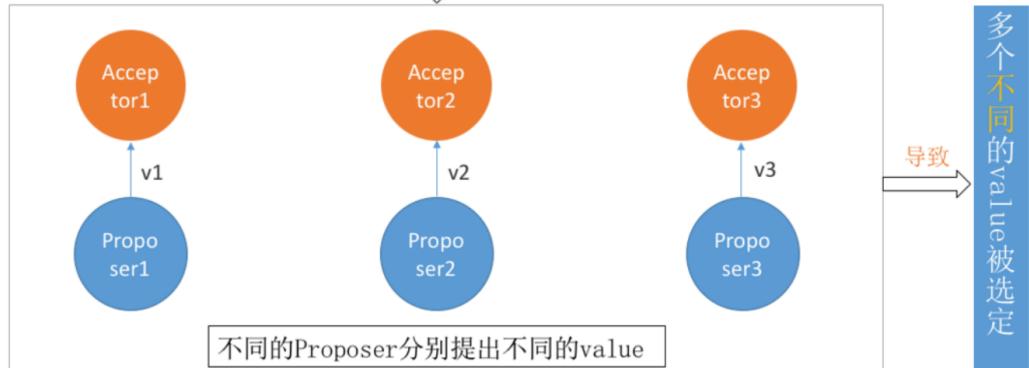
但是，这又会引出另一个问题：如果每个Proposer分别提出不同的value，发给不同的Acceptor。根据P1，Acceptor分别接受自己收到的第一个提案，就导致不同的value被选定。出现了不一致。如下图：

推导过程

希望：即使只有一个Proposer提出了一个value，该value也最终被选定。

P1：一个Acceptor必须接受它收到的第一个提案。

引出另一个问题



刚刚是因为『一个提案只要被一个Acceptor接受，则该提案的value就被选定了』才导致了出现上面不一致的问题。因此，我们需要加一个规定：

规定：一个提案被选定需要被半数以上的Acceptor接受

这个规定又暗示了：『一个Acceptor必须能够接受不止一个提案！』不然可能导致最终没有value被选定。比如上图的情况。v1、v2、v3都没有被选定，因为它们都只被一个Acceptor的接受。

所以在这种情况下，我们使用一个全局的编号来标识每一个Acceptor批准的提案，当一个具有某value值的提案被半数以上的Acceptor批准后，我们就认为该value被选定了，此时我们也认为该提案被选定了，强调下，提案和value不是同一个概念，『提案=提案编号+value』。

根据上面的内容，我们现在虽然允许多个提案被选定，但必须保证所有被选定的提案都具有相同的value值。否则又会出现不一致。

于是有了下面的约束：

P2：如果某个value为v的提案被选定了，那么每个编号更高的被选定提案的value必须也是v。

一个提案只有被Acceptor接受才可能被选定，因此我们可以把P2约束改写成对Acceptor接受的提案的约束P2a。

P2a：如果某个value为v的提案被选定了，那么每个编号更高的被Acceptor接受的提案的value必须也是v。

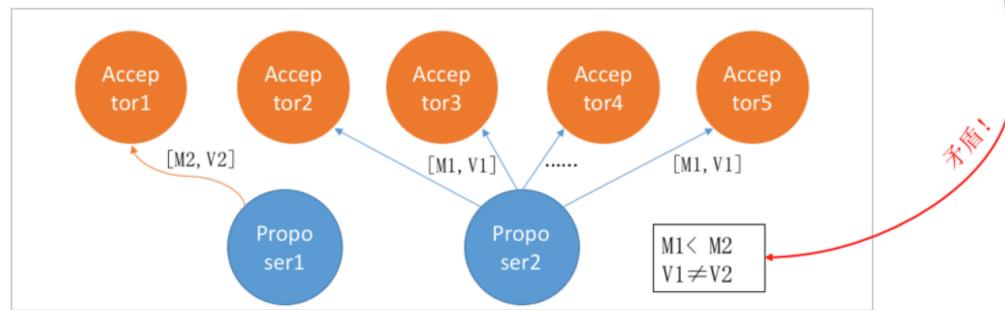
只要满足了P2a，就能满足P2。

推导过程

P2：如果某个value为v的提案被选定了，那么每个编号更高的被选定提案的value必须也是v。



P2a：如果某个value为v的提案被选定了，那么每个编号更高的被Acceptor接受的提案的value必须也是v。



但是，考虑如下的情况：假设总的有5个Acceptor。Proposer2提出[M1,V1]的提案，Acceptor2~5（半数以上）均接受了该提案，于是对于Acceptor2~5和Proposer2来讲，它们都认为V1被选定。

Acceptor1刚刚从宕机状态恢复过来（之前Acceptor1没有收到过任何提案），此时Proposer1向Acceptor1发送了[M2,V2]的提案（ $V2 \neq V1$ 且 $M2 > M1$ ），对于Acceptor1来讲，这是它收到的第一个提案。根据P1（一个Acceptor必须接受它收到的第一个提案。），Acceptor1必须接受该提案！同时Acceptor1认为V2被选定。这就出现了两个问题：

1. Acceptor1认为V2被选定，Acceptor2~5和Proposer2认为V1被选定。出现了不一致。
2. V1被选定了，但是编号更高的被Acceptor1接受的提案[M2,V2]的value为V2，且 $V2 \neq V1$ 。这就跟P2a（如果某个value为v的提案被选定了，那么每个编号更高的被Acceptor接受的提案的value必须也是v）矛盾了。

所以我们要对P2a约束进行强化！

P2a是对Acceptor接受的提案约束，但其实提案是Proposer提出来的，所有我们可以对Proposer提出的提案进行约束。得到P2b：

P2b：如果某个value为v的提案被选定了，那么之后任何Proposer提出的编号更高的提案的value必须也是v。

由P2b可以推出P2a进而推出P2。

那么，如何确保在某个value为v的提案被选定后，Proposer提出的编号更高的提案的value都是v呢？

只要满足P2c即可：

P2c：对于任意的 M_n 和 v_n ，如果提案 $[M_n, v_n]$ 被提出，那么肯定存在一个由半数以上的Acceptor组成的集合S，满足以下两个条件中的任意一个：

- * 要么S中每个Acceptor都没有接受过编号小于 M_n 的提案。
- * 要么S中所有Acceptor批准的所有编号小于 M_n 的提案中，编号最大的那个提案的value值为 v_n

从上面的内容，可以看出，从P1到P2c的过程其实是对一系列条件的逐步增强，如果需要证明这些条件可以保证一致性，那么就可以进行反向推导： $P2c \Rightarrow P2b \Rightarrow P2a \Rightarrow P2$ ，然后通过P2和P1来保证一致性

Proposer生成提案

接下来来学习，在P2c的基础上如何进行提案的生成

这里有个比较重要的思想：Proposer生成提案之前，应该先去『学习』已经被选定或者可能被选定的value，然后以该value作为自己提出的提案的value。如果没有value被选定，Proposer才可以自己决定value的值。这样才能达成一致。这个学习的阶段是通过一个『Prepare请求』实现的。

于是我们得到了如下的提案生成算法：

1. Proposer选择一个新的提案编号N，然后向某个Acceptor集合（半数以上）发送请求，要求该集合中的每个Acceptor做出如下响应（response）
 - (a) 向Proposer承诺保证不再接受任何编号小于N的提案。
 - (b) 如果Acceptor已经接受过提案，那么就向Proposer反馈已经接受过的编号小于N的，但为最大编号的提案的值。我们将该请求称为编号为N的Prepare请求。
2. 如果Proposer收到了半数以上的Acceptor的响应，那么它就可以生成编号为N，Value为V的提案[N,V]。这里的V是所有的响应中编号最大的提案的Value。如果所有的响应中都没有提案，那么此时V就可以由Proposer自己选择。生成提案后，Proposer将该提案发送给半数以上的Acceptor集合，并期望这些Acceptor能接受该提案。我们称该请求为Accept请求。

Acceptor接受提案

刚刚讲解了Paxos算法中Proposer的处理逻辑，怎么去生成的提案，下面来看看Acceptor是如何批准提案的

根据刚刚的介绍，一个Acceptor可能会受到来自Proposer的两种请求，分别是Prepare请求和Accept请求，对这两类请求作出响应的条件分别如下

- Prepare请求：Acceptor可以在任何时候响应一个Prepare请求
- Accept请求：在不违背Accept现有承诺的前提下，可以任意响应Accept请求

因此，对Acceptor接受提案给出如下约束：

P1a：一个Acceptor只要尚未响应过任何编号大于N的Prepare请求，那么他就可以接受这个编号为N的提案。

算法优化

上面的内容中，分别从Proposer和Acceptor对提案的生成和批准两方面来讲解了Paxos算法在提案选定过程中的算法细节，同时也在提案的编号全局唯一的前提下，获得了一个提案选定算法，接下来我们再对这个初步算法做一个小优化，尽可能的忽略Prepare请求

推导过程—算法优化（Acceptor）



如果Acceptor收到一个编号为N的Prepare请求，在此之前它已经响应过编号大于N的Prepare请求。根据P1a，该Acceptor不可能接受编号为N的提案。因此，该Acceptor可以忽略编号为N的Prepare请求。

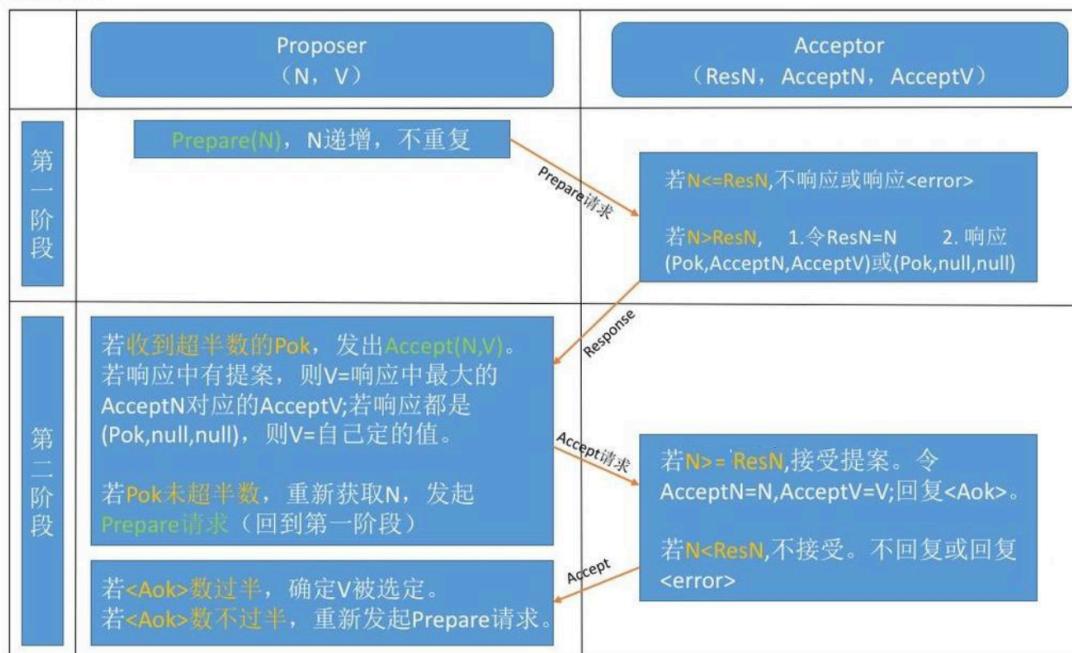
通过这个优化，每个Acceptor只需要记住它已经批准的提案的最大编号以及它已经做出Prepare请求响应的提案的最大编号，以便出现故障或节点重启的情况下，也能保证P2c的不变性，而对于Proposer来说，只要它可以保证不会产生具有相同编号的提案，那么就可以丢弃任意的提案以及它所有的运行时状态信息

Paxos算法描述

综合前面的讲解，我们来对Paxos算法的提案选定过程进行下总结，那结合Proposer和Acceptor对提案的处理逻辑，就可以得到类似于两阶段提交的算法执行过程

Paxos算法分为两个阶段。具体如下：

算法演示



- 阶段一：

- Proposer选择一个提案编号N，然后向半数以上的Acceptor发送编号为N的Prepare请求。
- 如果一个Acceptor收到一个编号为N的Prepare请求，且N大于该Acceptor已经响应过的所有Prepare请求的编号，那么它就会将它已经接受过的编号最大的提案（如果有的话）作为响应反馈给Proposer，同时该Acceptor承诺不再接受任何编号小于N的提案。

- 阶段二：

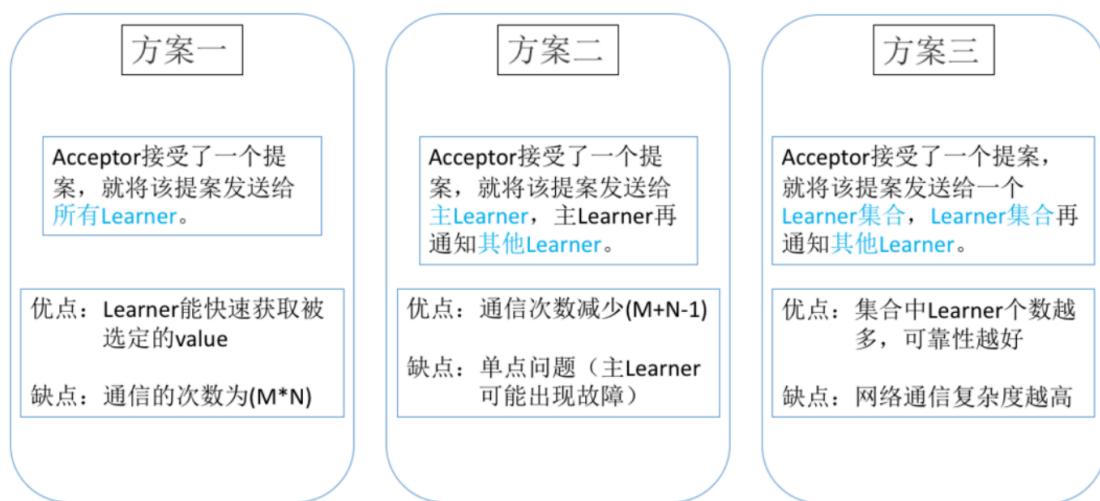
- 如果Proposer收到半数以上Acceptor对其发出的编号为N的Prepare请求的响应，那么它就会发送一个针对[N,V]提案的Accept请求给半数以上的Acceptor。注意：V就是收到的响应中编号最大的提案的value，如果响应中不包含任何提案，那么V就由Proposer自己决定。
- 如果Acceptor收到一个针对编号为N的提案的Accept请求，只要该Acceptor没有对编号大于N的Prepare请求做出过响应，它就接受该提案。

当然，实际运行过程中，每一个Proposer都有可能产生多个提案，但只要每个Proposer都遵循如上所述的算法运行，就一定能够保证算法执行的正确性

Learner学习被选定的value

刚刚我们介绍了如何来选定一个提案，下面我们再来看看如何让Learner获取提案，大体可以有以下三种方案：

Learner学习（获取）被选定的value



方案一：Learner获取一个已经被选定的提案的前提是，该提案已经被半数以上的Acceptor批准，因此，最简单的做法就是一旦Acceptor批准了一个提案，就将该提案发送给所有的Learner

很显然，这种做法虽然可以让Learner尽快地获取被选定的提案，但是却需要让每个Acceptor与所有的Learner逐个进行一次通信，通信的次数至少为二者个数的乘积

方案二：

另一种可行的方案是，我们可以让所有的Acceptor将它们对提案的批准情况，统一发送给一个特定的Learner（称为主Learner），各个Learner之间可以通过消息通信来互相感知提案的选定情况，基于这样的前提，当主Learner被通知一个提案已经被选定时，它会负责通知其他的learner

在这种方案中，Acceptor首先会将得到批准的提案发送给主Learner，再由其同步给其他Learner。因此较方案一而言，方案二虽然需要多一个步骤才能将提案通知到所有的learner，但其通信次数却大大减少了，通常只是Acceptor和Learner的个数总和，但同时，该方案引入了一个新的不稳定因素：主Learner随时可能出现故障

方案三：

在讲解方案二的时候，我们提到，方案二最大的问题在于主Learner存在单点问题，即主Learner随时可能出现故障，因此，对方案二进行改进，可以将主Learner的范围扩大，即Acceptor可以将批准的提案发送给一个特定的Learner集合，该集合中每个Learner都可以在一个提案被选定后通知其他的Learner。这个Learner集合中的Learner个数越多，可靠性就越好，但同时网络通信的复杂度也就越高

如何保证Paxos算法的活性

根据前面的内容讲解，我们已经基本上了解了Paxos算法的核心逻辑，那接下来再来看看Paxos算法在实际过程中的一些细节

活性：最终一定会发生的事情：最终一定要选定value

假设存在这样一种极端情况，有两个Proposer依次提出了一系列编号递增的提案，导致最终陷入死循环，没有value被选定，具体流程如下：

通过选取主Proposer保证算法的活性

假设有两个Proposer依次提出编号递增的提案，最终会陷入死循环，没有value被选定。（无法保证活性）

Proposer P1发出编号为M1的Prepare请求，收到过半响应。完成了阶段一的流程。

同时，Proposer P2发出编号为M2(M2>M1)的Prepare请求，也收到过半响应。也完成了阶段一的流程。于是Acceptor承诺不再接受编号小于M2的提案。

P1进入第二阶段的时候，发出的Accept请求被Acceptor忽略，于是P1再次进入阶段一并发出编号为M3(M3>M2)的Prepare请求。

这又导致P2在阶段二的Accept请求被忽略。
P2再次进入阶段一，发出编号为M4(M4>M3)的Prepare请求。。。

陷入死循环。。。都无法完成阶段二，没有value被选定。

选取一个主Proposer，只有主Proposer才能提出提案！

解决：通过选取主Proposer，并规定只有主Proposer才能提出议案。这样一来只要主Proposer和过半的Acceptor能够正常进行网络通信，那么但凡主Proposer提出一个编号更高的提案，该提案终将会被批准，这样通过选择一个主Proposer，整套Paxos算法就能够保持活性

总结：

前面的内容，主要就是从协议设计和原理实现角度讲解了二阶段提交协议，三阶段提交协议和Paxos这三种经典的一致性算法，可以说，这三种一致性协议都是非常优秀的分布式一致性协议，都从不同方面不同程度解决了分布式数据一致性的问题，使用范围都非常广泛，其中二阶段提交协议解决了分布式事务的原子性问题，保证了分布式事务的多个参与者要么都执行成功，要么都执行失败，但是，在二阶段解决部分分布式事务问题的同时，依然存在一些难以解决的诸如同步阻塞、无限期等待和“脑裂”等问题，三阶段提交协议则是在二阶段提交协议的基础上，添加了PreCommit过程，从而避免了二阶段提交协议中的无限期等待问题。

而paxos算法引入了“过半”的理论，通俗的讲就是少数服从多数的原则，同时Paxos算法支持分布式节点角色之间的轮换，这极大的避免了分布式单点问题的出现，因此Paxos算法即解决了无限期等待问题，也解决了“脑裂问题”，是目前来说最优秀的分布式一致性协议之一

分布式理论：一致性算法 Raft

什么是Raft 算法

首先说什么是 Raft 算法：Raft 是一种为了管理复制日志的一致性算法。

Raft提供了和Paxos算法相同的功能和性能，但是它的算法结构和Paxos不同。Raft算法更加容易理解并且更容易构建实际的系统。

为了提升可理解性，Raft将一致性算法分解成了几个关键模块，例如领导人选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量。

Paxos和Raft都是为了实现一致性（Consensus）这个目标，在Raft中这个过程如同选举一样，参选者需要说服大多数选民（服务器）投票给他，一旦选定后就跟随其操作。

Raft算法分为两个阶段，首先是选举过程，然后在选举出来的领导人带领进行正常操作，比如日志复制等。

领导人选举

Raft 通过选举一个领导人，然后给予他全部的管理复制日志的责任来实现一致性。

在Raft中，任何时候一个服务器都可以扮演下面的角色之一：

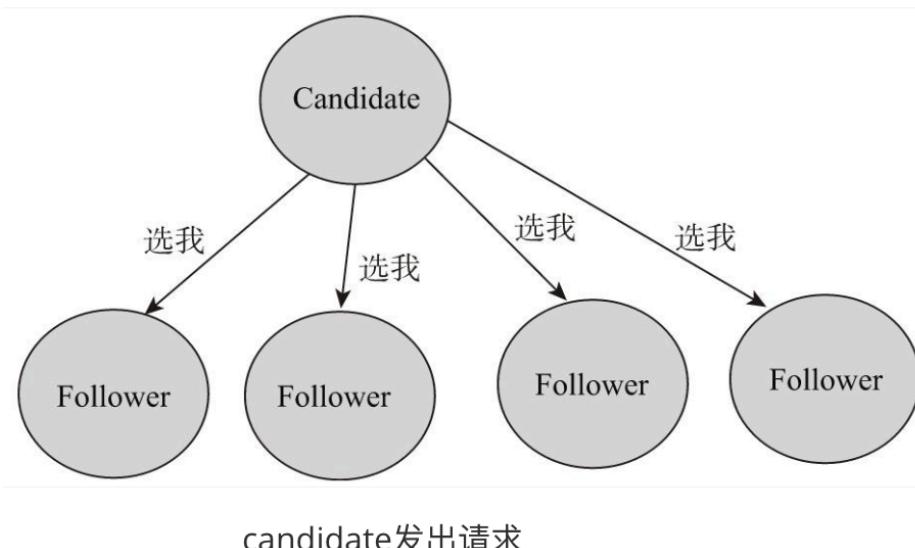
- 领导者：处理客户端交互，日志复制等动作，一般一次只有一个领导者
- 候选者：候选者就是在选举过程中提名自己的实体，一旦选举成功，则成为领导者
- 跟随者：类似选民，完全被动的角色，这样的服务器等待被通知投票

而影响他们身份变化的则是 **选举**。

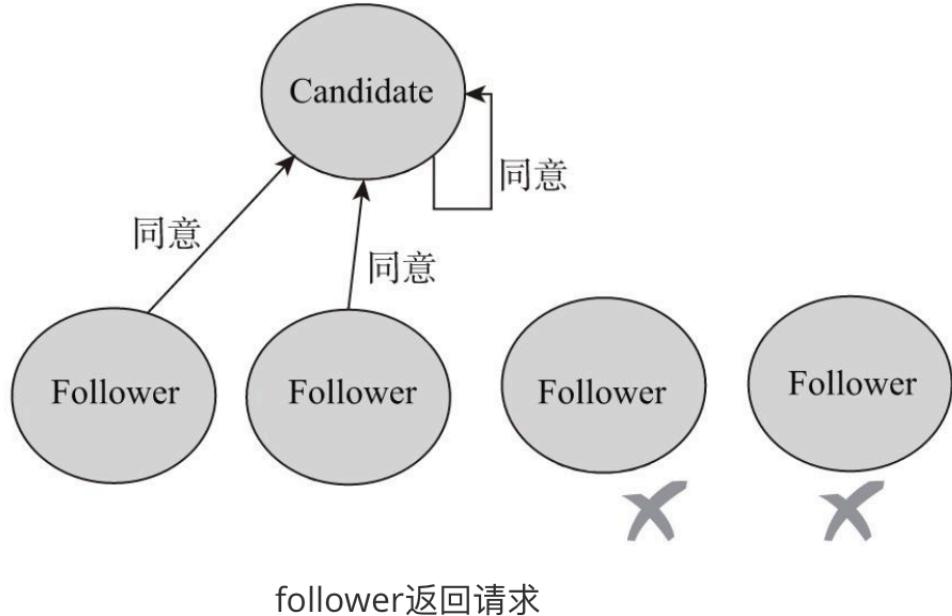
Raft使用心跳机制来触发选举。当server启动时，初始状态都是follower。每一个server都有一个定时器，超时时间为election timeout（一般为150-300ms），如果某server没有超时的情况下收到来自领导者或者候选者的任何消息，定时器重启，如果超时，它就开始一次选举。

下面用图示展示这个过程：

(1) 任何一个服务器都可以成为一个候选者，它向其他服务器（选民）发出选举自己的请求，如图

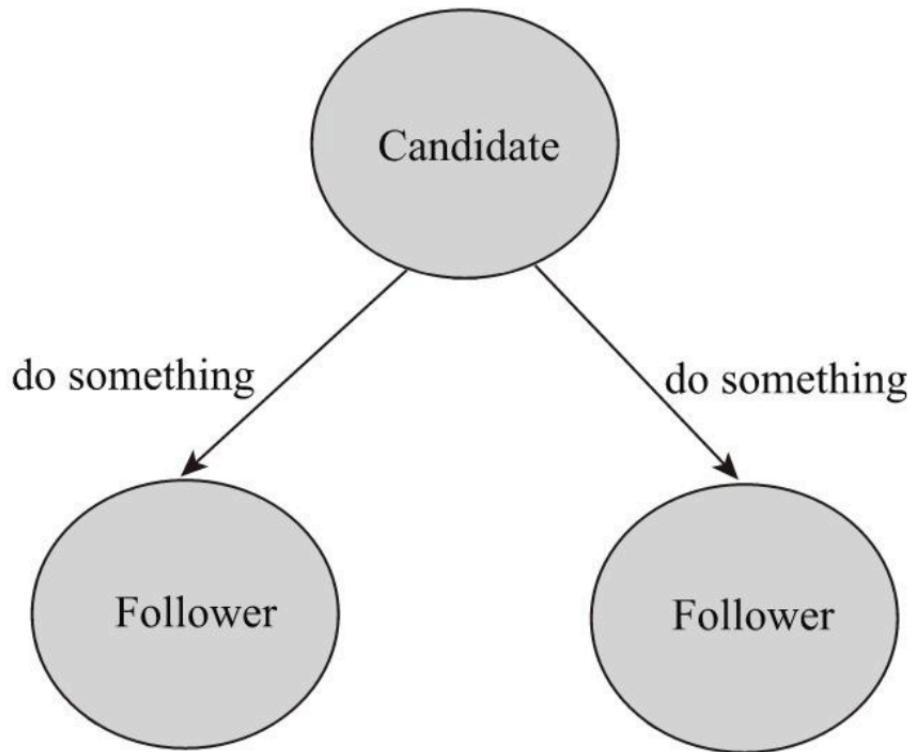


2) 其他服务器同意了，回复OK（同意）指令，如图所示。

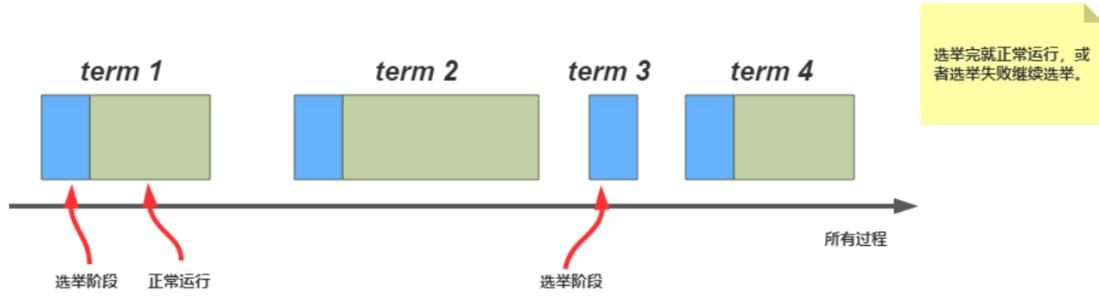


此时如果有一个Follower服务器宕机，没有收到请求选举的要求，则只要达到半数以上的票数，候选人还是可以成为领导者的。

3) 这样，这个候选者就成为领导者，它可以向选民们发出要执行具体操作动作的指令，比如进行日志复制



而领导者也有宕机的时候，宕机后引发新的 选举，所以，整个集群在选举和正常运行之间切换，具体如下图：



从上图可以看出，选举和正常运行之间切换，但请注意，上图中的 term 3 有一个地方，后面没有跟着正常运行阶段，为什么呢？

答：当一次选举失败（比如正巧每个人都投了自己），就执行一次 **加时赛**，每个 Server 会在一个随机的时间里重新投票，这样就能保证不冲突了。所以，当 term 3 选举失败，等了几十毫秒，执行 term 4 选举，并成功选举出领导人。

接着，领导者周期性的向所有跟随者发送心跳包来维持自己的权威。如果一个跟随者在一段时间里没有接收到任何消息，也就是超时，那么他就会认为系统中没有可用的领导者，并且发起选举以选出新的领导者。

要开始一次选举过程，跟随者先要增加自己的当前任期号并且**转换到候选人状态**。然后请求其他服务器为自己投票。那么会产生 3 种结果：

- a. 自己成功当选
- b. 其他的服务器成为领导者
- c. 僵住，没有任何一个人成为领导者

注意：

1. 每一个 server 最多在一个任期内投出一张选票（有任期号约束），先到先得。
2. 要求最多只能有一个人赢得选票。
3. 一旦成功，立即成为领导人，然后广播所有服务器停止投票阻止新得领导产生。

僵住怎么办？Raft 通过使用随机选举超时时间（例如 150 - 300 毫秒）的方法将服务器打散投票。每个候选人在僵住的时候会随机从一个时间开始重新选举。

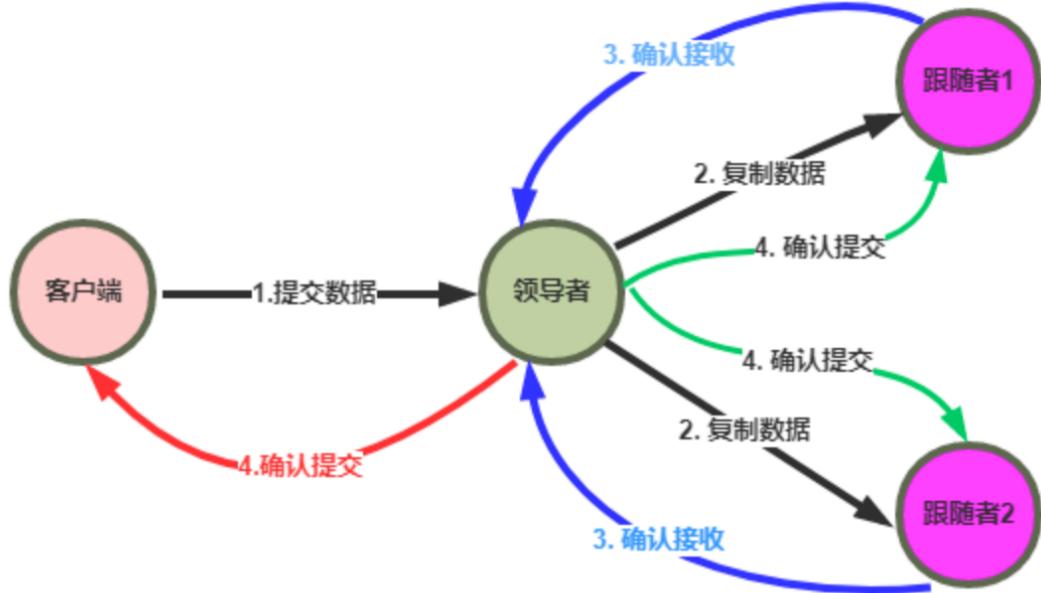
以上，就是 Raft 所有关于领导选举的策略。

日志复制（保证数据一致性）

日志复制的过程

Leader 选出后，就开始接收客户端的请求。Leader 把请求作为日志条目（Log entries）加入到它的日志中，然后并行的向其他服务器发起 AppendEntries RPC 复制日志条目。当这条日志被复制到大多数服务器上，Leader 将这条日志应用到它的状态机并向客户端返回执行结果。

下图表示了当一个客户端发送一个请求给领导，随后领导复制给跟随者的整个过程。



4 个步骤：

- 客户端的每一个请求都包含被复制状态机执行的指令。
- **leader**把这个指令作为一条新的日志条目添加到日志中，然后并行发起 RPC 给其他的服务器，让他们复制这条信息。
- 跟随者响应ACK,如果 follower 宕机或者运行缓慢或者丢包，**leader**会不断的重试，直到所有的 follower 最终都复制了所有的日志条目。
- 通知所有的Follower提交日志，同时领导人提交这条日志到自己的状态机中，并返回给客户端。

可以看到，直到第四步骤，整个事务才会达成。中间任何一个步骤发生故障，都不会影响日志一致性。

分布式系统设计策略

分布式系统本质是通过低廉的硬件攒在一起以获得更好的吞吐量、性能以及可用性等。

在分布式环境下，有几个问题是普遍关心的，我们称之为设计策略：

- 如何检测当前节点还活着？
- 如何保障高可用？
- 容错处理
- 负载均衡

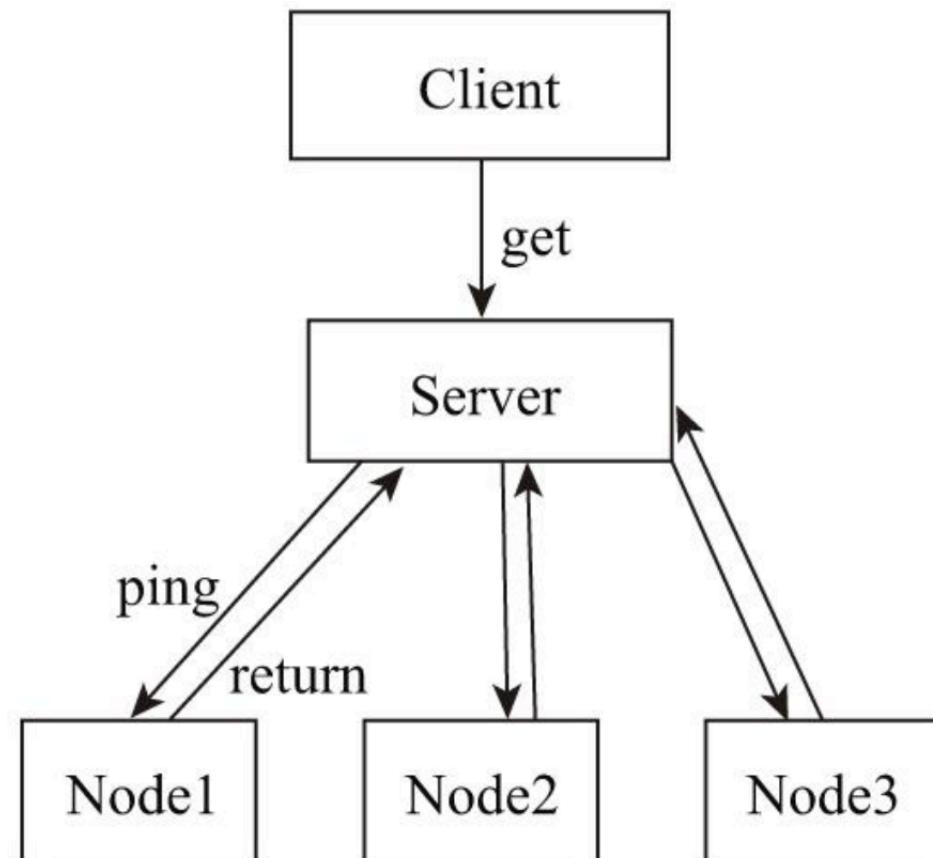
心跳检测

在分布式环境中，我们提及过存在非常多的节点（Node），其实质是这些节点分担任务的运行、计算或者程序逻辑处理。那么就有一个非常重要的问题，如何检测一个节点出现了故障乃至无法工作了？

通常解决这一问题是采用心跳检测的手段，如同通过仪器对病人进行一些检测诊断一样。

心跳顾名思义，就是以固定的频率向其他节点汇报当前节点状态的方式。收到心跳，一般可以认为一个节点和现在的网络拓扑是良好的。当然，心跳汇报时，一般也会携带一些附加的状态、元数据信息，以便管理

如图所示，Client请求Server，Server转发请求到具体的Node获取请求结果。Server需要与三个Node节点保持心跳连接，确保Node可以正常工作。



若Server没有收到Node3的心跳时，Server认为Node3失联。但是失联是失去联系，并不确定是否是Node3故障，有可能是Node3处于繁忙状态，导致调用检测超时；也有可能是Server与Node3之间链路出现故障或闪断。所以心跳不是万能的，收到心跳可以确认节点正常，但是收不到心跳也不能认为该节点就已经宣告“死亡”。此时，可以通过一些方法帮助Server做决定：周期检测心跳机制、累计失效检测机制。

周期检测心跳机制

Server端每间隔 t 秒向Node集群发起监测请求，设定超时时间，如果超过超时时间，则判断“死亡”。

累计失效检测机制

在周期检测心跳机制的基础上，统计一定周期内节点的返回情况（包括超时及正确返回），以此计算节点的“死亡”概率。另外，对于宣告“濒临死亡”的节点可以发起有限次数的重试，以作进一步判断。

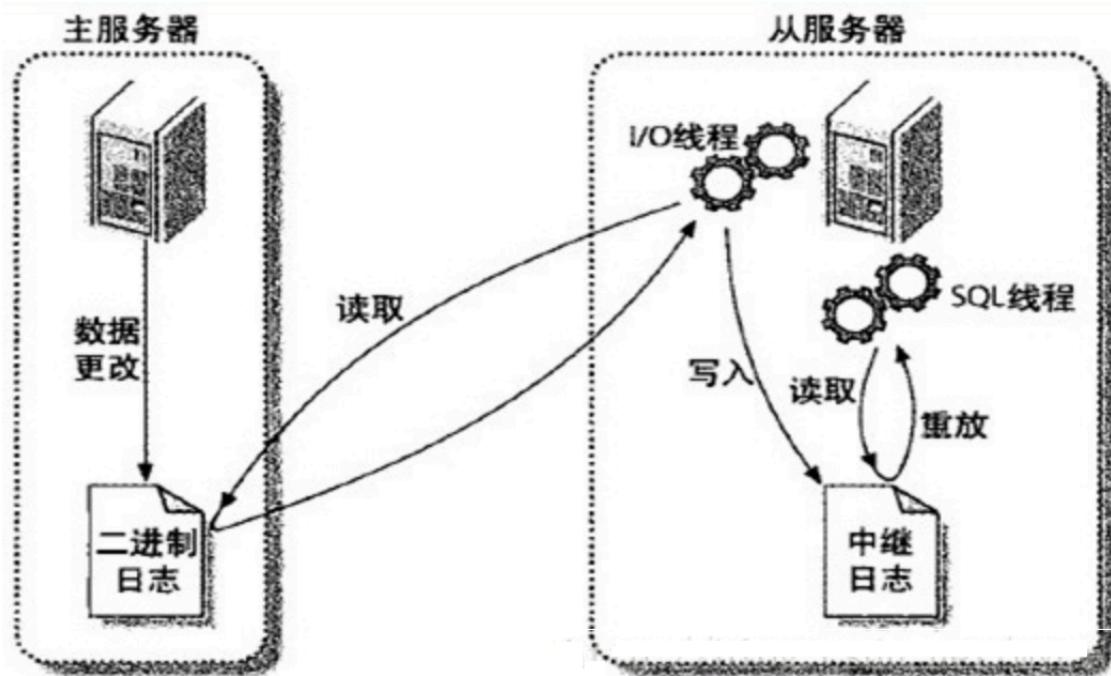
通过周期检测心跳机制、累计失效检测机制可以帮助判断节点是否“死亡”，如果判断“死亡”，可以把该节点踢出集群

高可用设计

高可用(High Availability)是系统架构设计中必须考虑的因素之一,通常是指,经过设计来减少系统不能提供服务的时间

系统高可用性的常用设计模式包括三种：主备（Master-SLave）、互备（Active-Active）和集群（Cluster）模式。

1. 主备模式 主备模式就是Active-Standby模式，当主机宕机时，备机接管主机的一切工作，待主机恢复正常后，按使用者的设定以自动（热备）或手动（冷备）方式将服务切换到主机上运行。在数据库部分，习惯称之为MS模式。MS模式即Master/Slave模式，这在数据库高可用性方案中比较常用，如MySQL、Redis等就采用MS模式实现主从复制。保证高可用，如图所示。



MySQL之间数据复制的基础是二进制日志文件（binary log file）。一台MySQL数据库一旦启用二进制日志后，作为master，它的数据库中所有操作都会以“事件”的方式记录在二进制日志中，其他数据库作为slave通过一个I/O线程与主服务器保持通信，并监控master的二进制日志文件的变化，如果发现master二进制日志文件发生变化，则会把变化复制到自己的中继日志中，然后slave的一个SQL线程会把相关的“事件”执行到自己的数据库中，以此实现从数据库和主数据库的一致性，也就实现了主从复制。

2.互备模式 互备模式指两台主机同时运行各自的服务工作且相互监测情况。在数据库高可用部分，常见的互备是MM模式。MM模式即Multi-Master模式，指一个系统存在多个master，每个master都具有read-write能力，会根据时间戳或业务逻辑合并版本。

我们使用过的、构建过的MySQL服务绝大多数都是Single-Master，整个拓扑中只有一个Master承担写请求。比如，基于Master-Slave架构的主从复制，但是也存在由于种种原因，我们可能需要MySQL服务具有Multi-Master的特性，希望整个拓扑中可以有不止一个Master承担写请求

3.集群模式

集群模式是指有多个节点在运行，同时可以通过主控节点分担服务请求。如Zookeeper。集群模式需要解决主控节点本身的高可用问题，一般采用主备模式。

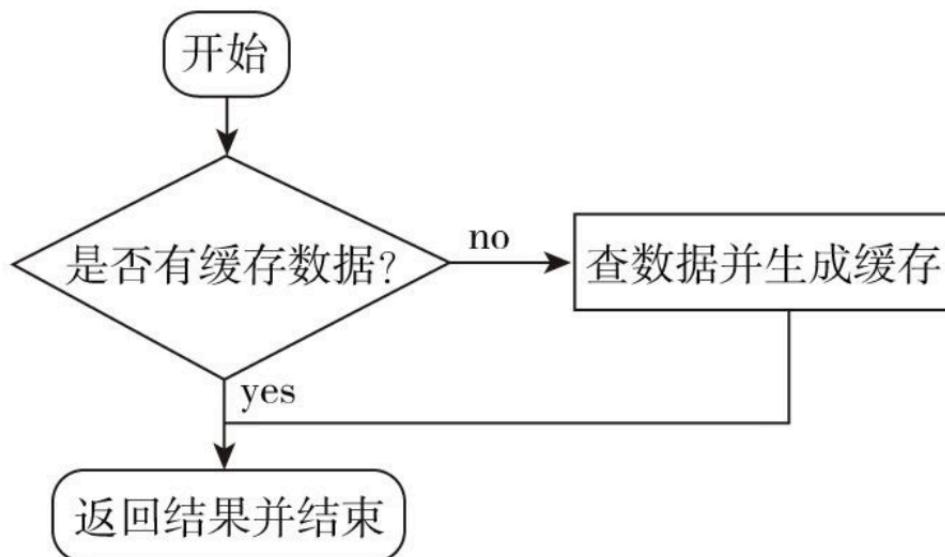
容错性

容错顾名思义就是IT系统对于错误包容的能力

容错的处理是保障分布式环境下相应系统的高可用或者健壮性，一个典型的案例就是对于缓存穿透 问题的解决方案。

我们来具体看一下这个例子，如图所示

有个业务逻辑



问题描述：

我们在项目中使用缓存通常都是先检查缓存中是否存在，如果存在直接返回缓存内容，如果不存在就直接查询数据库然后再缓存查询结果返回。这个时候如果我们查询的某一个数据在缓存中一直不存在，就会造成每一次请求都查询DB，这样缓存就失去了意义，在流量大时，或者有人恶意攻击

如频繁发起为id为“-1”的条件进行查询，可能DB就挂掉了。

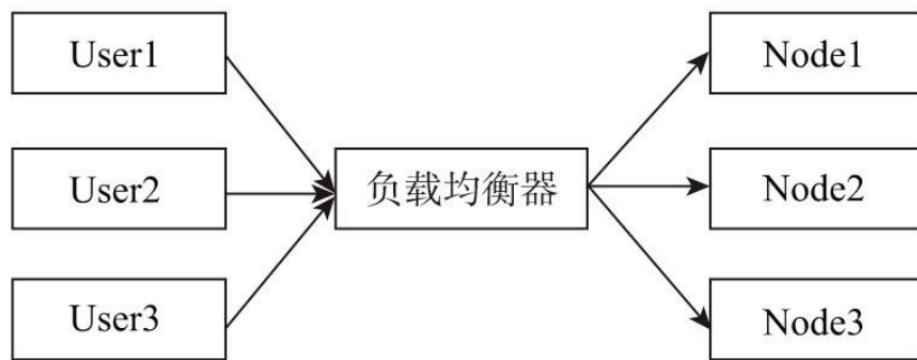
那这种问题有什么好办法解决呢？

一个比较巧妙的方法是，可以将这个不存在的key预先设定一个值。比如，key="null"。在返回这个null值的时候，我们的应用就可以认为这是不存在的key，那我们的应用就可以决定是否继续等待访问，还是放弃掉这次操作。如果继续等待访问，过一个时间轮询点后，再次请求这个key，如果取到的值不再是null，则可以认为这时候key有值了，从而避免了透传到数据库，把大量的类似请求挡在了缓存之中。

负载均衡

负载均衡：其关键在于使用多台集群服务器共同分担计算任务，把网络请求及计算分配到集群可用的不同服务器节点上，从而达到高可用性及较好的用户操作体验。

如图，不同的用户User1、User2、User3访问应用，通过负载均衡器分配到不同的节点。



负载均衡器有硬件解决方案，也有软件解决方案。硬件解决方案有著名的F5，软件有LVS、HAProxy、Nginx等。

以Nginx为例，负载均衡有以下几种策略：

- 轮询：即Round Robin，根据Nginx配置文件中的顺序，依次把客户端的Web请求分发到不同的后端服务器。
- 最少连接：当前谁连接最少，分发给谁。
- IP地址哈希：确定相同IP请求可以转发给同一个后端节点处理，以方便session保持。
- 基于权重的负载均衡：配置Nginx把请求更多地分发到高配置的后端服务器上，把相对较少的请求分发到低配服务器。

分布式架构网络通信

在分布式服务框架中，一个最基础的问题就是远程服务是怎么通讯的，在Java领域中有很多可实现远程通讯的技术，例如：RMI、ESB、Hessian、SOAP和JMS等，它们背后到底是基于什么原理实现的呢

基本原理

要实现网络机器间的通讯，首先得来看看计算机系统网络通信的基本原理，在底层层面去看，网络通信需要做的就是将流从一台计算机传输到另外一台计算机，基于传输协议和网络IO来实现，其中传输协议比较出名的有tcp、udp等等，tcp、udp都是在基于Socket概念上为某类应用场景而扩展出的传输协议，网络IO，主要有bio、nio、aio三种方式，所有的分布式应用通讯都基于这个原理而实现，只是为了应用的易用，各种语言通常都会提供一些更为贴近应用易用的应用层协议

什么是RPC

RPC全称为remote procedure call，即远程过程调用。

借助RPC可以做到像本地调用一样调用远程服务，是一种进程间的通信方式

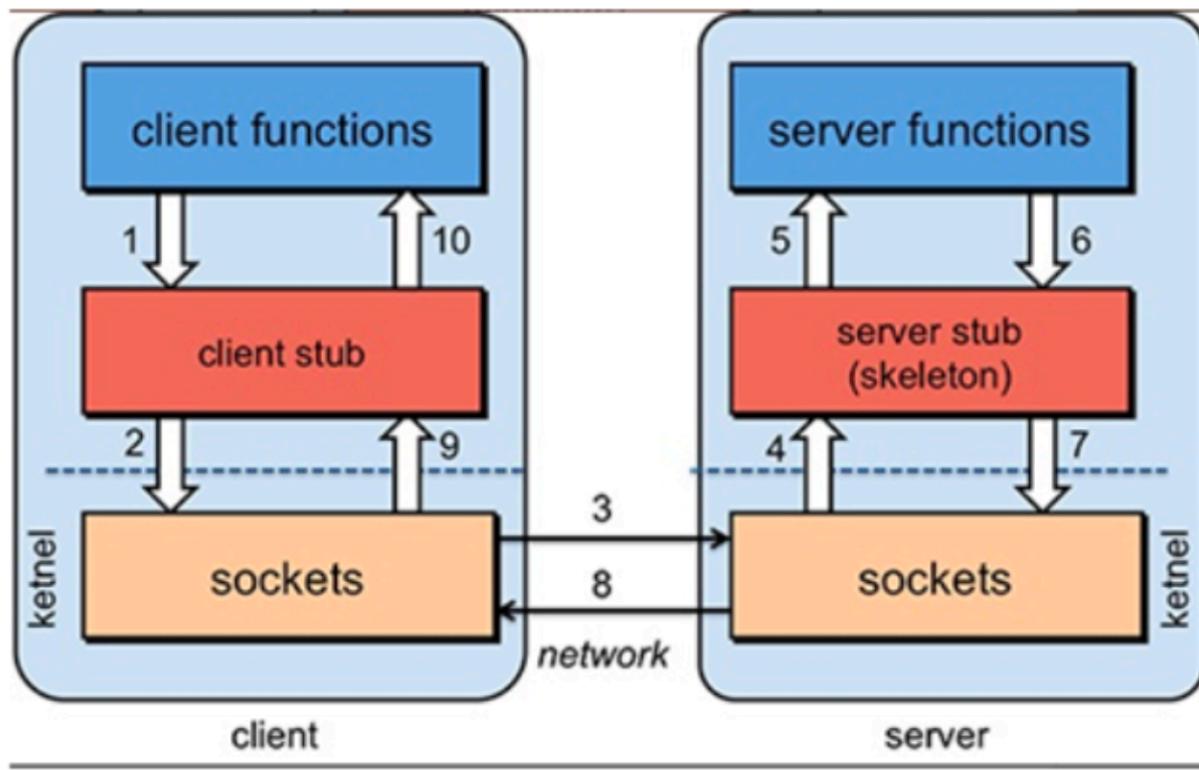
比如两台服务器A和B，A服务器上部署一个应用，B服务器上部署一个应用，A服务器上的应用想调用B服务器上的应用提供的方法，由于两个应用不在一个内存空间，不能直接调用，所以需要通过网络来表达调用的语义和传达调用的数据。

需要注意的是RPC并不是一个具体的技术，而是指整个网络远程调用过程。

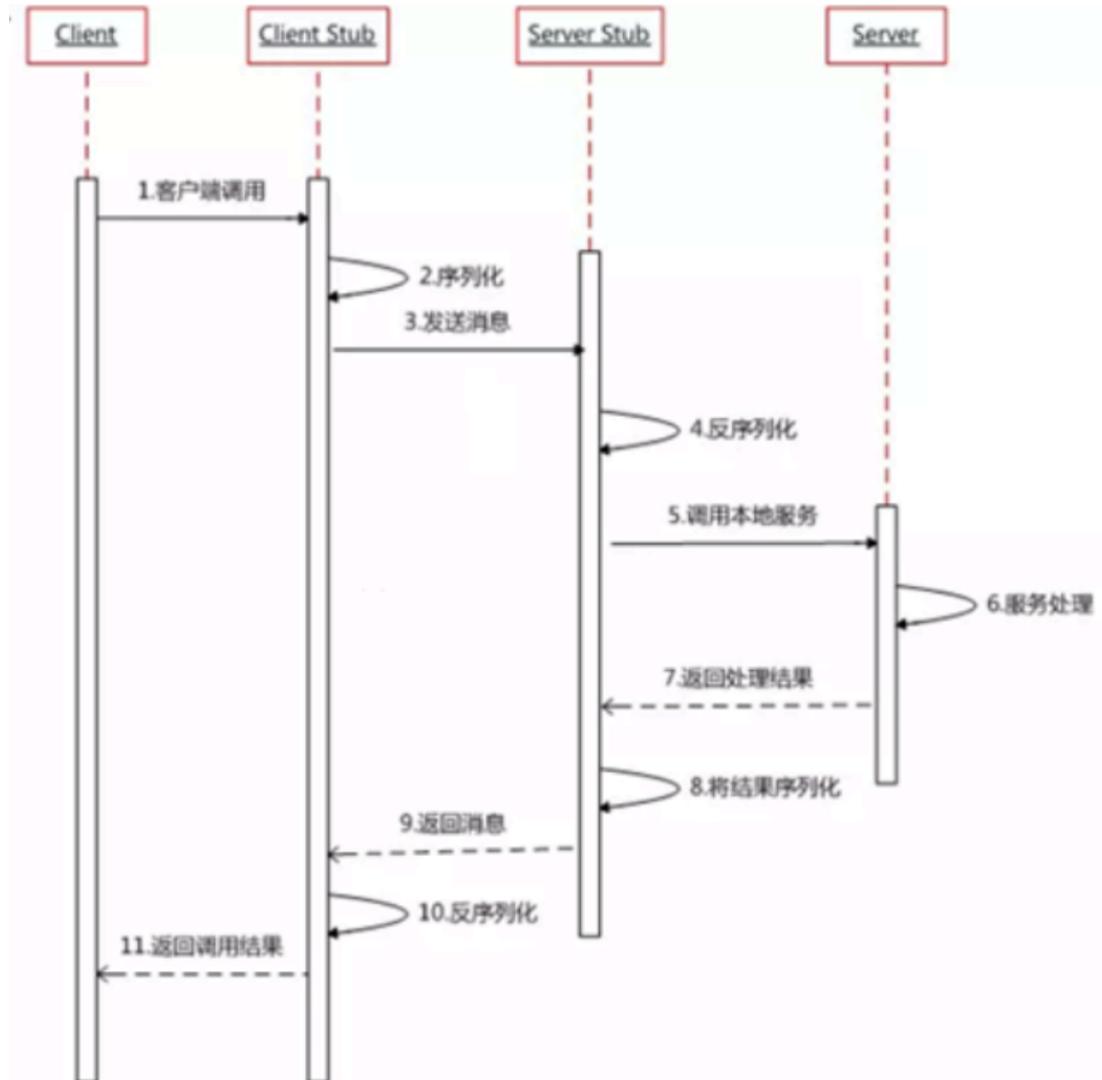
RPC架构

一个完整的RPC架构里面包含了四个核心的组件，分别是Client，Client Stub，Server以及Server Stub，这个Stub可以理解为存根。

- 客户端(Client)，服务的调用方。
- 客户端存根(Client Stub)，存放服务端的地址消息，再将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方。
- 服务端(Server)，真正的服务提供者。
- 服务端存根(Server Stub)，接收客户端发送过来的消息，将消息解包，并调用本地的方法。



RPC调用过程



- (1) 客户端 (client) 以本地调用方式 (即以接口的方式) 调用服务;
- (2) 客户端存根 (client stub) 接收到调用后, 负责将方法、参数等组装成能够进行网络传输的消息体 (将消息体对象序列化为二进制) ;
- (3) 客户端通过sockets将消息发送到服务端;
- (4) 服务端存根(server stub) 收到消息后进行解码 (将消息对象反序列化) ;
- (5) 服务端存根(server stub) 根据解码结果调用本地的服务;
- (6) 本地服务执行并将结果返回给服务端存根(server stub) ;
- (7) 服务端存根(server stub) 将返回结果打包成消息 (将结果消息对象序列化) ;
- (8) 服务端 (server) 通过sockets将消息发送到客户端;
- (9) 客户端存根 (client stub) 接收到结果消息, 并进行解码 (将结果消息发序列化) ;
- (10) 客户端 (client) 得到最终结果。

RPC的目标是要把2、3、4、7、8、9这些步骤都封装起来。

注意：无论是何种类型的数据，最终都需要转换成二进制流在网络上进行传输，数据的发送方需要将对象转换为二进制流，而数据的接收方则需要把二进制流再恢复为对象。

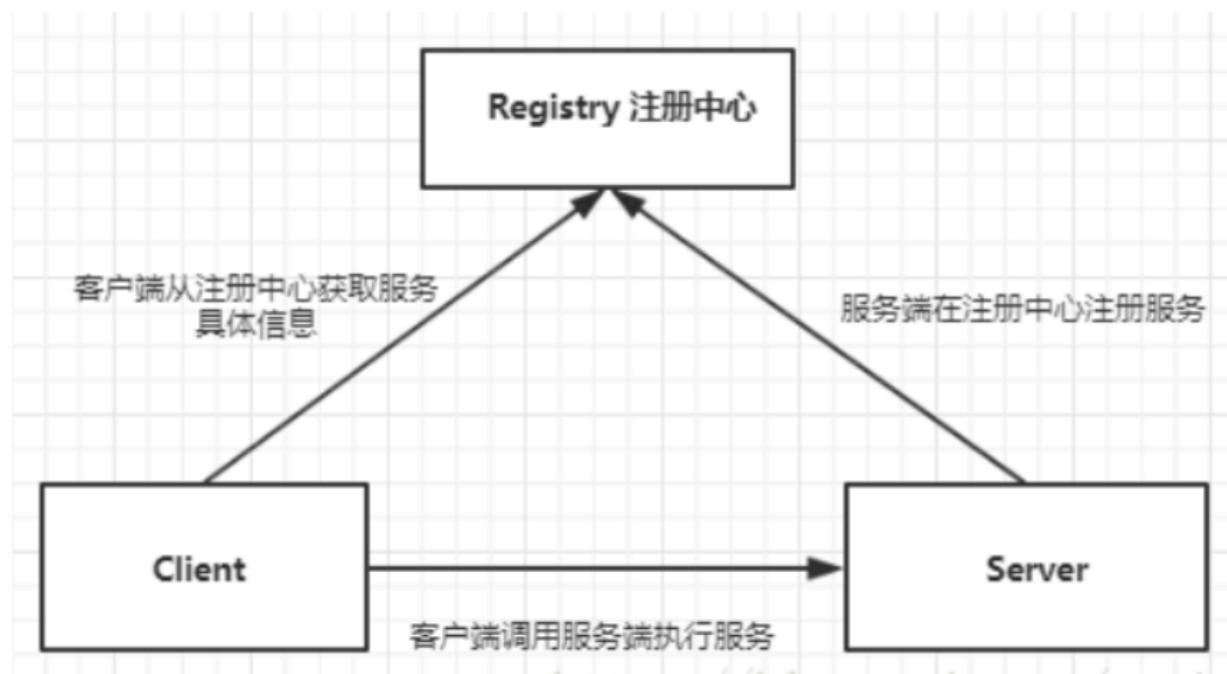
在java中RPC框架比较多，常见的有RMI、Hessian、gRPC、bRPC、Dubbo等，其实对于RPC框架而言，核心模块就是通讯和序列化

RMI

简介

Java RMI 指的是远程方法调用 (Remote Method Invocation),是java原生支持的远程调用 ,采用 JRMP (Java RemoteMessageing protocol) 作为通信协议，可以认为是纯java版本的分布式远程调用解决方案， RMI主要用于不同虚拟机之间的通信，这些虚拟机可以在不同的主机上、也可以在同一个主机上，这里的通信可以理解为一个虚拟机上的对象调用另一个虚拟机上对象的方法。

核心概念



以一个案例来进一步理解RMI的使用。

案例步骤

1. 创建远程接口，并且继承java.rmi.Remote接口
2. 实现远程接口，并且继承:UnicastRemoteObject
3. 创建服务器程序: createRegistry()方法注册远程对象
4. 创建客户端程序(获取注册信息，调用接口方法)

代码实现

1. 创建远程接口

```
import java.rmi.Remote;
import java.rmi.RemoteException;
/**
 * 远程服务对象接口必须继承Remote接口；同时方法必须抛出RemoteException异常
 */
public interface Hello extends Remote {
    public String sayHello(User user) throws RemoteException;
}
```

其中有一个引用对象作为参数

```
import java.io.Serializable;
/**
 * 引用对象应该是可序列化对象，这样才能在远程调用的时候：1. 序列化对象 2. 拷贝 3. 在网络中传输
 * 4. 服务端反序列化 5. 获取参数进行方法调用； 这种方式其实是将远程对象引用传递的方式转化为值传递的方式
 */
public class User implements Serializable {

    private String name;
    private int age;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

2. 实现远程服务对象

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
/**
 * 远程服务对象实现类写在服务端；必须继承UnicastRemoteObject或其子类
*/
```

```

/**/
public class HelloImpl extends UnicastRemoteObject implements Hello {

    /**
     * 因为UnicastRemoteObject的构造方法抛出了 RemoteException 异常，因此这里默认的构造
     * 方法必须写，必须声明抛出 RemoteException 异常
     *
     * @throws RemoteException
     */
    private static final long serialVersionUID = 3638546195897885959L;

    protected HelloImpl() throws RemoteException {
        super();
        // TODO Auto-generated constructor stub
    }

    @Override
    public String sayHello(User user) throws RemoteException {
        System.out.println("this is server, hello:" + user.getName());
        return "success";
    }
}

```

3. 服务端程序

```

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
/***
 * 服务端程序
 ***/
public class Server {

    public static void main(String[] args) {

        try {
            Hello hello = new HelloImpl(); // 创建一个远程对象，同时也会创建stub对象、
            skeleton对象
            //本地主机上的远程对象注册表Registry的实例，并指定端口为8888，这一步必不可少（Java
            默认端口是1099），必不可缺的一步，缺少注册表创建，则无法绑定对象到远程注册表上
            LocateRegistry.createRegistry(8080); //启动注册服务
            try {
                //绑定的URL标准格式为：rmi://host:port/name(其中协议名可以省略，下面两种写法
                //都是正确的)
                Naming.bind("//127.0.0.1:8080/zm", hello); //将stub引用绑定到服务地址上
            } catch (MalformedURLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
    System.out.println("service bind already!!");

} catch (RemoteException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

}
}

```

4. 客户端程序

```

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
/***
 * 客户端程序
 * @author zm
 *
 */
public class Client {
    public static void main(String[] args) {
        try {
            //在RMI服务注册表中查找名称为RHello的对象，并调用其上的方法
            Hello hello = (Hello) Naming.lookup("//127.0.0.1:8080/zm"); //获取远程对象
            User user = new User();
            user.setName("james");
            System.out.println(hello.sayHello(user));
        } catch (MalformedURLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch ( RemoteException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (NotBoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

5. 启动服务端程序

6. 客户端调用

服务端：

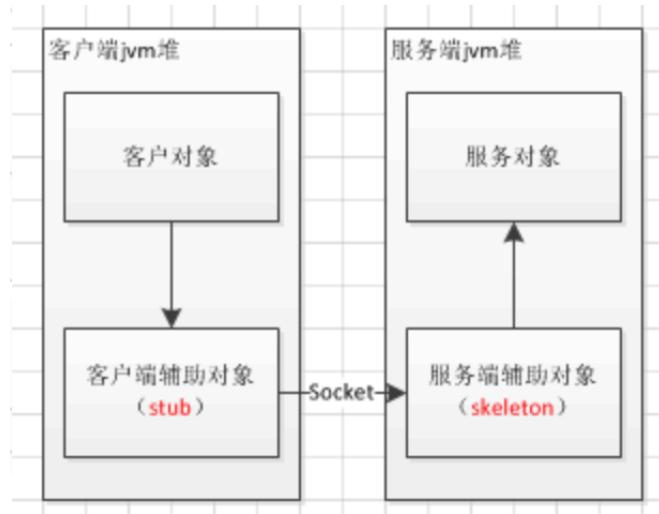
```
Server x client x
/Library/Java/JavaVirtualMachines/jdk-11.0.1
service bind already!!
this is server,hello:tom
```

客户端

```
Server x client x
/Library/Java/JavaVirtualMachines/jdk-11.0.1
success
```

实现原理

客户端只与代表远程主机中对象的Stub对象进行通信，丝毫不知道Server的存在。客户端只是调用Stub对象中的本地方法，Stub对象是一个本地对象，它实现了远程对象向外暴露的接口。客户端认为它是调用远程对象的方法，实际上是调用Stub对象中的方法。可以理解为Stub对象是远程对象在本地的一个代理，当客户端调用方法的时候，Stub对象会将调用通过网络传递给远程对象。Stub为远程对象在客户端的一个代理，当客户端调用远程对象的方法时，实际是委托Stub这个代理去调用远程的对象



BIO、NIO、AIO

IO: 即输入 (Input) 和输出(Output), 以内存为目标, 数据进内存叫 In, 出内存叫 Out

同步和异步

不能光从字面意思了解, 要从分布式的角度思考。

同步 (synchronize) 、异步 (asynchronous) 关注的是消息通信机制。

同步:

就是在发出一个调用时，在没有得到结果之前，该调用就不返回。但是当调用返回，就得到返回值了。简而言之，就是由调用者主动等待这个调用结果。通常所说的线程安全基本上就是指是同步的。
(亲力亲为且专注于一件事)

使用同步IO时，Java自己处理IO的读写

同步：

跟同步相反。调用在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，就是当一个异步进程调用发出之后，调用者不会立刻得到结果。而是在调用发出之后，被调用者通过状态、通知来通知调用者，或者通过回调函数来处理这个调用。

使用异步IO时，Java将IO读写委托给OS处理，需要将数据缓冲区地址和大小传给OS，OS需要支持异步IO操作

即：指示别人帮自己做事，完成后得到通知

举个例子：

你打电话问东门酸奶店老板有没有蓝莓味的炒酸奶，如果是同步通信机制，那么老板会说：“我先看一下冰箱啊，等我确认之后（可能是5分钟也可能半小时）告诉你结果（返回结果）不挂电话”

而异步通信机制，老板直接告诉你我先确认一下，确认好了再打电话给你，然后直接把电话给挂了（不返回结果）。等确认好了，他会主动打电话给你。这里老板通过“回电”这种方式来回调。挂电话

阻塞和非阻塞

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态。

阻塞调用是指 调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。线程挂起，就不会给这个线程分配CPU，节约了CPU资源。但是挂起线程的优先级高于其它，所以还是阻塞的。

举个例子：ATM机排队取款，你只能等待（使用阻塞IO的时候，Java调用会一直阻塞到读写完成才返回。）

非阻塞调用是指 在不能立刻得到结果之前，该调用不会阻塞当前线程。

举个例子：柜台取款，取个号，然后坐在椅子上做其他事，等广播通知，没到你的号你就不能去，但你可以不断的问大堂经理排到了没有。（使用非阻塞IO时，如果不能读写Java调用会马上返回，当IO事件分发器会通知可读写时再继续进行读写，不断循环直到读写完成）

例子

老张爱喝茶，煮开水。出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

1 老张把水壶放到火上，站立着等水开。（同步阻塞）

老张觉得自己有点傻

2 老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）

老张还是觉得自己有点傻，于是变高端了，买了把会响笛的那种水壶。水开之后，能大声发出嘀~~~~的噪音。

3 老张把响水壶放到火上，立等水开。（异步阻塞）

老张觉得这样傻等意义不大

4 老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）

老张觉得自己聪明了。

关于同步异步、阻塞非阻塞：

所谓同步异步，只是对于水壶而言。普通水壶，同步；响水壶，异步。虽然都能干活，但响水壶可以在自己完工之后，提示老张水开了。这是普通水壶所不能及的。同步只能让调用者去轮询自己（情况2中），造成老张效率的低下。

所谓阻塞非阻塞，仅仅对于老张而言。立等的老张，阻塞；看电视的老张，非阻塞。情况1和情况3中老张就是阻塞的，媳妇喊他都不知道。虽然3中响水壶是异步的，可对于立等的老张没有太大的意义。

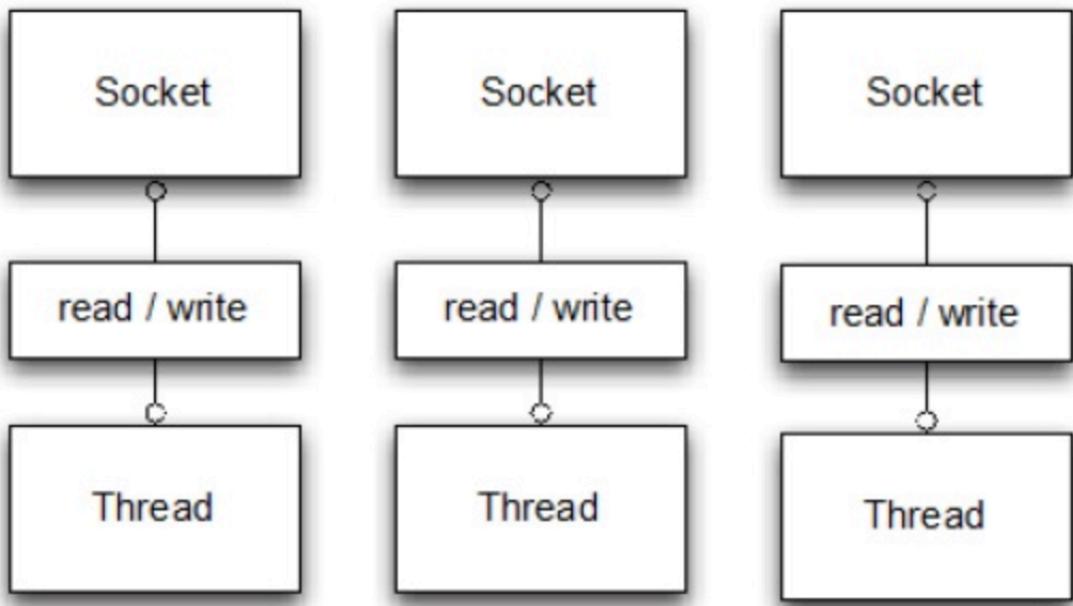
所以一般异步是配合非阻塞使用的，这样才能发挥异步的效用。

BIO

同步阻塞IO。B代表blocking

服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

适用场景：Java1.4之前唯一的选择，简单易用但资源开销太高



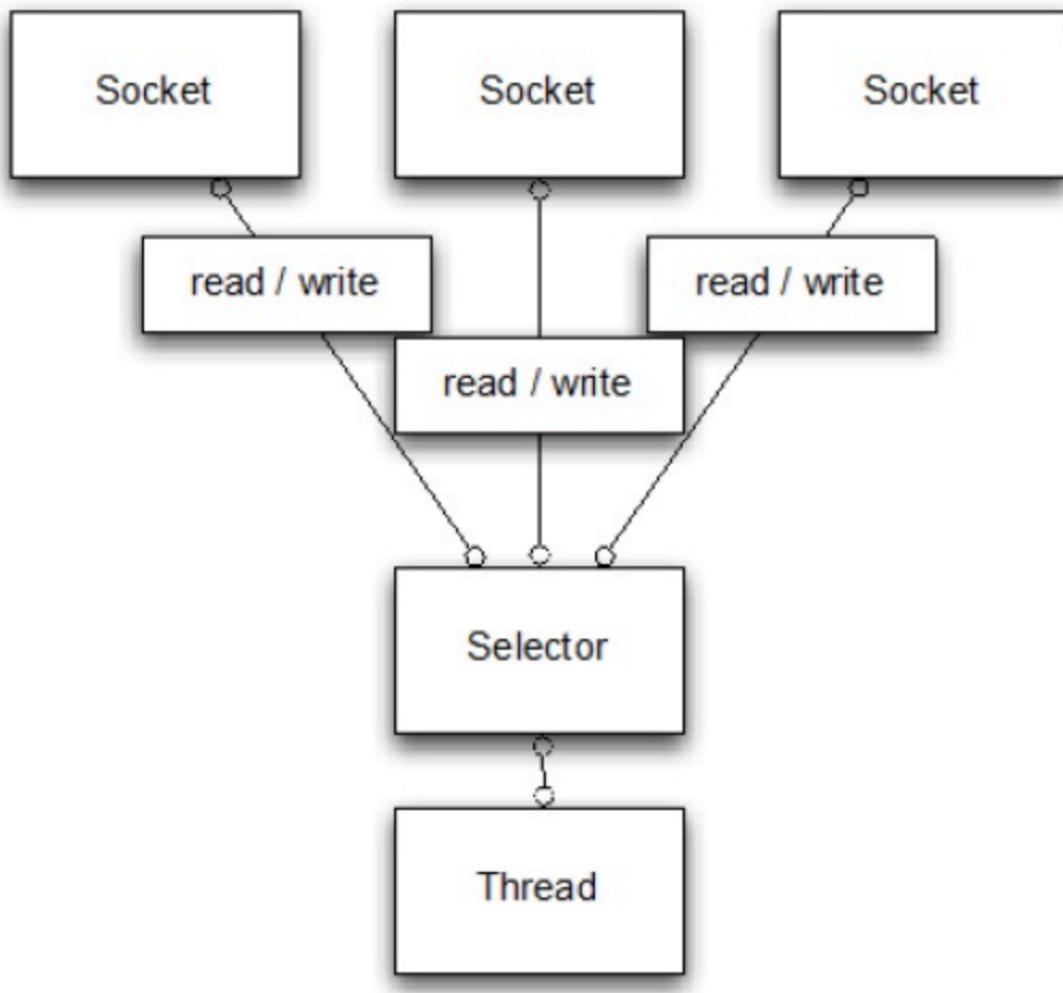
阻塞IO的通信方式

NIO

同步非阻塞IO。N代表non-blocking

服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有IO请求时才启动一个线程进行处理。

适用场景：连接数目多且连接比较短（轻操作）的架构，比如聊天服务器。编程较为复杂。**Java1.4**开始支持



非阻塞IO的通信方式

从这两图可以看出，NIO的单线程能处理连接的数量比BIO要高出很多，而为什么单线程能处理更多的连接呢？原因就是图二中出现的 Selector。当一个连接建立之后，他有两个步骤要做，第一步是接收完客户端发过来的全部数据，第二步是服务端处理完请求业务之后返回response给客户端。NIO和BIO的区别主要是在第一步。在BIO中，等待客户端发数据这个过程是阻塞的，这样就造成了一个线程只能处理一个请求的情况，而机器能支持的最大线程数是有限的，这就是为什么BIO不能支持高并发的原因。而NIO中，当一个Socket建立好之后，Thread并不会阻塞去接受这个Socket，而是将这个请求交给Selector，Selector会不断的去遍历所有的Socket，一旦有一个Socket建立完成，他会通知Thread，然后Thread处理完数据再返回给客户端——这个过程是不阻塞的，这样就能让一个Thread处理更多的请求了。

AIO

异步非阻塞IO。A代表asynchronize

服务器实现模式为一个有效请求一个线程，客户端的IO请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

使用场景：连接数目多且连接比较长（重操作）的架构，比如相册服务器。重点调用了OS参与并发操作，编程比较复杂。**Java7**开始支持

Netty

在开始了解Netty是什么之前，我们先来回顾一下，如果我们需要实现一个客户端与服务端通信的程序，使用传统的IO编程，应该如何来实现？

IO编程

我们简化下场景：客户端每隔两秒发送一个带有时间戳的"hello world"给服务端，服务端收到之后打印。

为了方便演示，下面例子中，服务端和客户端各一个类，把这两个类拷贝到你的IDE中，先后运行 `IOServer.java` 和 `IOClient.java` 可看到效果。

下面是传统的IO编程中服务端实现

`IOServer.java`

```
public class IOServer {
    public static void main(String[] args) throws Exception {

        //首先创建了一个`serverSocket`来监听8000端口
        ServerSocket serverSocket = new ServerSocket(8000);

        // (1) 接收新连接线程
        new Thread(() -> {
            while (true) {
                try {
                    // (1) 阻塞方法获取新的连接
                    Socket socket = serverSocket.accept();

                    // (2) 每一个新的连接都创建一个线程，负责读取数据
                    new Thread(() -> {
                        try {
                            byte[] data = new byte[1024];
                            InputStream inputStream = socket.getInputStream();
                            while (true) {
                                int len;
                                // (3) 按字节流方式读取数据
                                while ((len = inputStream.read(data)) != -1) {
                                    System.out.println(new String(data, 0,
len));
                                }
                            }
                        } catch (IOException e) {
                        }
                    }).start();
                } catch (IOException e) {
                }
            }
        }).start();
    }
}
```

```
        }

    }

}).start();
}

}
```

server端首先创建了一个 `serverSocket` 来监听8000端口，然后创建一个线程，线程里面不断调用阻塞方法 `serversocket.accept()`；获取新的连接，见(1)，当获取到新的连接之后，给每条连接创建一个新的线程，这个线程负责从该连接中读取数据，见(2)，然后读取数据是以字节流的方式，见(3)

下面是传统的IO编程中客户端实现

IOClient.java

```
public class IOclient {

    public static void main(String[] args) {
        new Thread(() -> {
            try {
                Socket socket = new Socket("127.0.0.1", 8000);
                while (true) {
                    try {
                        socket.getOutputStream().write((new Date() + ": hello
world").getBytes());
                        socket.getOutputStream().flush();
                        Thread.sleep(2000);
                    } catch (Exception e) {
                    }
                }
            } catch (IOException e) {
            }
        }).start();
    }
}
```

客户端的代码相对简单，连接上服务端8000端口之后，每隔2秒，我们向服务端写一个带有时间戳的 "hello world"。

IO编程模型在客户端较少的情况下运行良好，但是对于客户端比较多的业务来说，单机服务端可能需要支撑成千上万的连接，IO模型可能就不太合适了，我们来分析一下原因。

上面的demo，从服务端代码中我们可以看到，在传统的IO模型中，每个连接创建成功之后都需要一个线程来维护，每个线程包含一个while死循环，那么1w个连接对应1w个线程，继而1w个while死循环，这就带来如下几个问题：

1. 线程资源受限：线程是操作系统中非常宝贵的资源，同一时刻有大量的线程处于阻塞状态是非常严

- 重的资源浪费，操作系统耗不起
2. 线程切换效率低下：单机cpu核数固定，线程爆炸之后操作系统频繁进行线程切换，应用性能急剧下降。
 3. 除了以上两个问题，IO编程中，我们看到数据读写是以字节流为单位，效率不高。

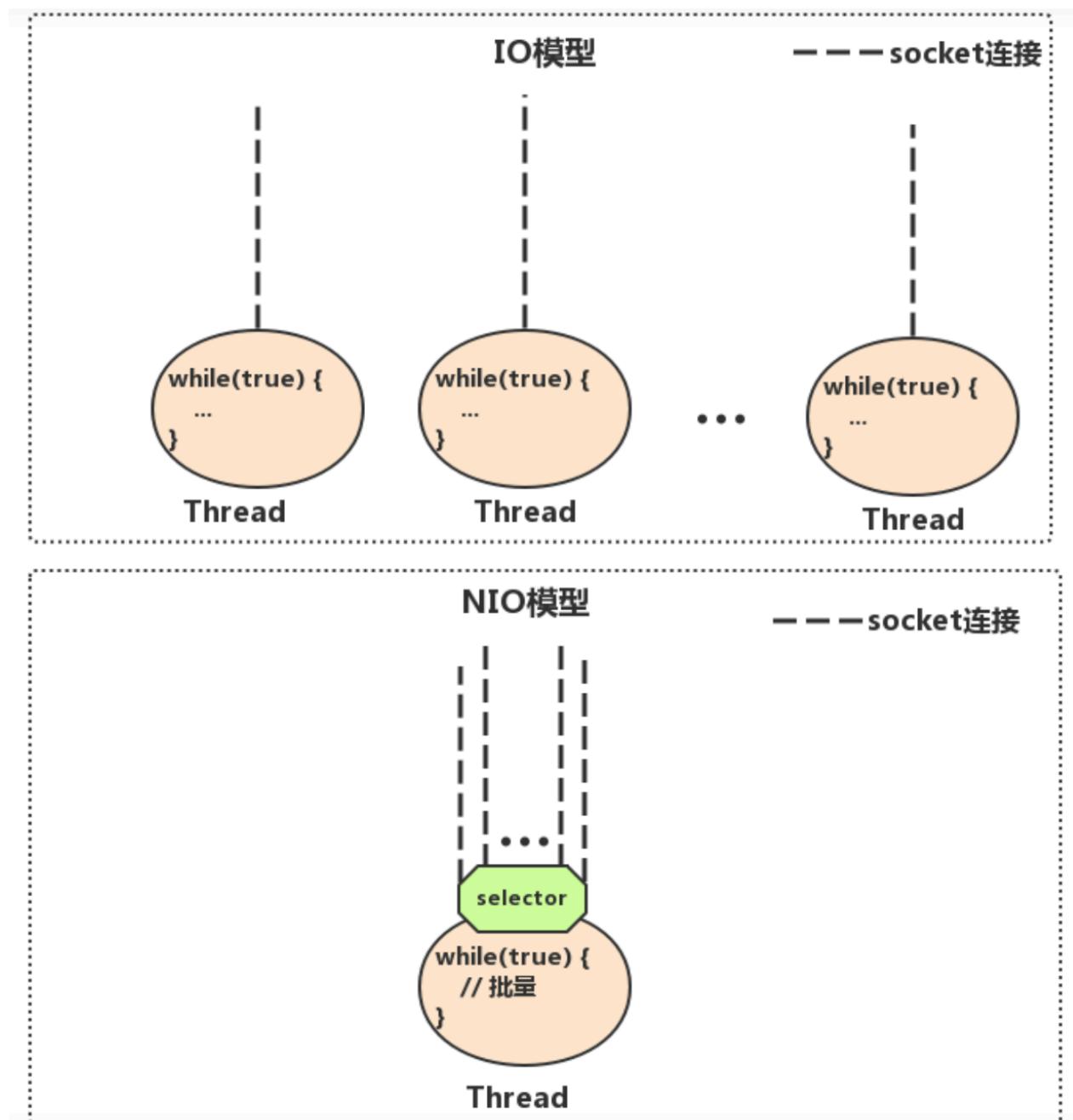
为了解决这三个问题，JDK在1.4之后提出了NIO

NIO编程

下面简单描述一下NIO是如何解决以上三个问题的。

线程资源受限

NIO编程模型中，新来一个连接不再创建一个新的线程，而是可以把这条连接直接绑定到某个固定的线程，然后这条连接所有的读写都由这个线程来负责，那么他是怎么做到的？我们用一幅图来对比一下IO与NIO



如上图所示，IO模型中，一个连接来了，会创建一个线程，对应一个while死循环，死循环的目的就是不断监测这条连接上是否有数据可以读，大多数情况下，1w个连接里面同一时刻只有少量的连接有数据可读，因此，很多个while死循环都白白浪费掉了，因为读不出啥数据。

而在NIO模型中，他把这么多while死循环变成一个死循环，这个死循环由一个线程控制，那么他又是如何做到一个线程，一个while死循环就能监测1w个连接是否有数据可读的呢？这就是NIO模型中selector的作用，一条连接来了之后，现在不创建一个while死循环去监听是否有数据可读了，而是直接把这条连接注册到selector上，然后，通过检查这个selector，就可以批量监测出有数据可读的连接，进而读取数据，下面我再举个非常简单的生活中的例子说明IO与NIO的区别

在一家幼儿园里，小朋友有上厕所的需求，小朋友都太小以至于你要问他要不要上厕所，他才会告诉你。幼儿园一共有100个小朋友，有两种方案可以解决小朋友上厕所的问题：

1. 每个小朋友配一个老师。每个老师隔段时间询问小朋友是否要上厕所，如果要上，就领他去厕所，100个小朋友就需要100个老师来询问，并且每个小朋友上厕所的时候都需要一个老师领着他去上，这就是IO模型，一个连接对应一个线程。
2. 所有的小朋友都配同一个老师。这个老师隔段时间询问所有的小朋友是否有人要上厕所，然后每一时刻把所有要上厕所的小朋友批量领到厕所，这就是NIO模型，所有小朋友都注册到同一个老师，对应的就是所有的连接都注册到一个线程，然后批量轮询。

这就是NIO模型解决线程资源受限的方案，实际开发过程中，我们会开多个线程，每个线程都管理着一批连接，相对于IO模型中一个线程管理一条连接，消耗的线程资源大幅减少

线程切换效率低下

由于NIO模型中线程数量大大降低，线程切换效率因此也大幅度提高

IO读写以字节为单位

NIO解决这个问题的方式是数据读写不再以字节为单位，而是以字节块为单位。IO模型中，每次都是从操作系统底层一个字节一个字节地读取数据，而NIO维护一个缓冲区，每次可以从这个缓冲区里面读取一块的数据，这就好比一盘美味的豆子放在你面前，你用筷子一个个夹（每次一个），肯定不如要勺子挖着吃（每次一批）效率来得高。

简单讲完了JDK NIO的解决方案之后，我们接下来使用NIO的方案替换掉IO的方案，我们先来看看，如果用JDK原生的NIO来实现服务端，该怎么做

NIOServer.java

```
public class NIOServer {  
    public static void main(String[] args) throws IOException {  
        Selector serverSelector = Selector.open();  
        Selector clientSelector = Selector.open();  
  
        new Thread(() -> {  
            try {  
                // 对应IO编程中服务端启动  
                ServerSocketChannel listenerChannel =  
                    ServerSocketChannel.open();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }).start();  
  
        clientSelector.register(listenerChannel,  
            SelectionKey.OP_ACCEPT);  
  
        while (true) {  
            clientSelector.select();  
            Iterator<SelectionKey> iterator = clientSelector.selectedKeys().  
                iterator();  
            while (iterator.hasNext()) {  
                SelectionKey key = iterator.next();  
                if (key.isAcceptable()) {  
                    ServerSocketChannel channel = (ServerSocketChannel) key.  
                        channel();  
                    channel.accept();  
                } else if (key.isReadable()) {  
                    ByteBuffer buffer = ByteBuffer.allocate(1024);  
                    channel.read(buffer);  
                    String message = new String(buffer.array());  
                    System.out.println("收到客户端消息：" + message);  
                }  
                iterator.remove();  
            }  
        }  
    }  
}
```

```
        listenerChannel.socket().bind(new InetSocketAddress(8000));
        listenerChannel.configureBlocking(false);
        listenerChannel.register(serverSelector,
SelectionKey.OP_ACCEPT);

        while (true) {
            // 监测是否有新的连接，这里的1指的是阻塞的时间为1ms
            if (serverSelector.select(1) > 0) {
                Set<SelectionKey> set = serverSelector.selectedKeys();
                Iterator<SelectionKey> keyIterator = set.iterator();

                while (keyIterator.hasNext()) {
                    SelectionKey key = keyIterator.next();

                    if (key.isAcceptable()) {
                        try {
                            // (1) 每来一个新连接，不需要创建一个线程，而是直接注册到clientSelector
                            SocketChannel clientChannel =
((ServerSocketChannel) key.channel()).accept();
                            clientChannel.configureBlocking(false);
                            clientChannel.register(clientSelector,
SelectionKey.OP_READ);

                        } finally {
                            keyIterator.remove();
                        }
                    }
                }
            }
        } catch (IOException ignored) {
    }

}).start();

new Thread(() -> {
    try {
        while (true) {
            // (2) 批量轮询是否有哪些连接有数据可读，这里的1指的是阻塞的时间为
1ms
            if (clientSelector.select(1) > 0) {
                Set<SelectionKey> set = clientSelector.selectedKeys();
                Iterator<SelectionKey> keyIterator = set.iterator();

                while (keyIterator.hasNext()) {
                    SelectionKey key = keyIterator.next();

```

```

        if (key.isReadable()) {
            try {
                SocketChannel clientChannel =
(SocketChannel) key.channel();
                ByteBuffer byteBuffer =
ByteBuffer.allocate(1024);
                // (3) 读取数据以块为单位批量读取
                clientChannel.read(byteBuffer);
                byteBuffer.flip();

                System.out.println(Charset.defaultCharset().newDecoder().decode(byteBuffer)
                        .toString());
            } finally {
                keyIterator.remove();
                key.interestOps(SelectionKey.OP_READ);
            }
        }
    }

}
} catch (IOException ignored) {
}
}).start();

}

}
}

```

相信大部分没有接触过NIO的同学应该会觉得，原来使用JDK原生NIO的API实现一个简单的服务端通信程序是如此复杂

强烈不建议直接基于JDK原生NIO来进行网络开发

1、JDK的NIO编程需要了解很多的概念，编程复杂，对NIO入门非常不友好，编程模型不友好，
 ByteBuffer的api非常繁琐复杂
 2、对NIO编程来说，一个比较合适的线程模型能充分发挥它的优势，而
 JDK没有给你实现，你需要自己实现，就连简单的自定义协议拆包都要你自己实现
 3、JDK的NIO底层由epoll实现，该实现饱受诟病的空轮训bug会导致cpu飙升100%
 4、项目庞大之后，自行实现的NIO很容易出现各类bug，维护成本较高

JDK的NIO虽然美好，但使用极其繁琐，正因为如此，就有了netty

Netty编程

什么是Netty

用一句简单的话来说就是：Netty封装了JDK的NIO，让你用得更爽，不用再写一大堆复杂的代码了。

用官方正式的话来说就是：Netty是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能服务器和客户端。

Netty 的内部实现是很复杂的，但是 Netty 提供了简单易用的API从网络处理代码中解耦业务逻辑。

Netty 是完全基于 NIO 实现的，所以整个 Netty 都是异步的

同时Netty也是最流行的 NIO 框架，它已经得到成百上千的商业、商用项目验证，许多框架和开源组件的底层 rpc 都是使用的 Netty，如 Dubbo、Elasticsearch 等等。

下面是官网给出的一些 Netty 的特性：

设计方面

- 对各种传输协议提供统一的 API（使用阻塞和非阻塞套接字时候使用的是同一个 API，只是需要设置的参数不一样）。
- 基于一个灵活、可扩展的事件模型来实现关注点清晰分离。
- 高度可定制的线程模型——单线程、一个或多个线程池。
- 真正的无数据报套接字（UDP）的支持（since 3.1）。

易用性

- 完善的 Javadoc 文档和示例代码。
- 不需要额外的依赖，JDK 5 (Netty 3.x) 或者 JDK 6 (Netty 4.x) 已经足够

性能

- 更好的吞吐量，更低的等待延迟。
- 更少的资源消耗。
- 最小化不必要的内存拷贝。

对于上面的特性我们在脑中有个简单了解和印象即可，下面开始我们的代码部分。

Netty版本的案例实现

首先，引入Maven依赖

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.6.Final</version>
</dependency>
```

然后，下面是服务端实现部分

NettyServer.java

```
package com.lagou.Netty;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
```

```
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

public class NettyServer {

    public static void main(String[] args) throws InterruptedException {

        //1.创建 NioEventLoopGroup的两个实例:bossGroup workerGroup
        // 当前这两个实例代表两个线程池，默认线程数为CPU核心数乘2
        // bossGroup接收客户端传过来的请求
        // workerGroup处理请求
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        //2、创建服务启动辅助类:组装一些必要的组件
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        serverBootstrap.group(bossGroup,workerGroup)
            //channel方法指定服务器监听的通道
            .channel(NioServerSocketChannel.class)
            //设置channel handler
            .childHandler(new ChannelInitializer<NioSocketChannel>() {
                @Override
                protected void initChannel(NioSocketChannel ch) throws
Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    pipeline.addLast(new StringEncoder());
                    pipeline.addLast(new StringDecoder());
                    pipeline.addLast(new
SimpleChannelInboundHandler<String>() {
                        @Override
                        protected void channelRead0(ChannelHandlerContext
channelHandlerContext, String s) throws Exception {
                            System.out.println(s);
                        }
                    });
                }
            });

        //bind监听端口
        ChannelFuture f = serverBootstrap.bind(8000).sync();
        System.out.println("tcp server start success..");
        //会阻塞等待直到服务器的channel关闭
        f.channel().closeFuture().sync();
    }
}
```

```
}
```

```
}
```

在编写 Netty 程序时，一开始都会生成 NioEventLoopGroup 的两个实例，分别是 bossGroup 和 workerGroup，也可以称为 parentGroup 和 childGroup，为什么创建这两个实例，作用是什么？可以这么理解，bossGroup 和 workerGroup 是两个线程池，它们默认线程数为 CPU 核心数乘以 2，bossGroup 用于接收客户端传过来的请求，接收到请求后将后续操作交由 workerGroup 处理。

接下来我们生成了一个服务启动辅助类的实例 bootstrap，bootstrap 用来为 Netty 程序的启动组装配置一些必须要组件，例如上面的创建的两个线程组。channel 方法用于指定服务器端监听套接字通道 NioServerSocketChannel，其内部管理了一个 Java NIO 中的 ServerSocketChannel 实例。

channelHandler 方法用于设置业务职责链，责任链是我们下面要编写的，责任链具体是什么，它其实就是由一个个的 ChannelHandler 串联而成，形成的链式结构。正是这一个个的 ChannelHandler 帮我们完成了要处理的事情。

ChannelInitializer 继承 ChannelInboundHandlerAdapter，用于初始化 Channel 的 ChannelPipeline。通过 initChannel 方法参数 sc 得到 ChannelPipeline 的一个实例。

当一个新的连接被接受时，一个新的 Channel 将被创建，同时它会被自动地分配到它专属的 ChannelPipeline

通过 addLast 方法将一个一个的 ChannelHandler 添加到责任链上并给它们取个名称（不取也可以，Netty 会给出默认名称），这样就形成了链式结构。在请求进来或者响应出去时都会经过链上这些 ChannelHandler 的处理。

最后再向链上加入我们自定义的 ChannelHandler 组件，处理自定义的业务逻辑

接着我们调用了 bootstrap 的 bind 方法将服务绑定到 8080 端口上，bind 方法内部会执行端口绑定等一系列操作，使得前面的配置都各就各位各司其职，sync 方法用于阻塞当前 Thread，一直到端口绑定操作完成。最后是应用程序将会阻塞等待直到服务器的 Channel 关闭。

客户端NIO的实现部分

NettyClient.java

```
public class NettyClient {
    public static void main(String[] args) throws InterruptedException {
        Bootstrap bootstrap = new Bootstrap();
        NioEventLoopGroup group = new NioEventLoopGroup();

        bootstrap.group(group)
            .channel(NioSocketChannel.class)
            .handler(new ChannelInitializer<Channel>() {
                @Override
                protected void initChannel(Channel ch) {
                    ch.pipeline().addLast(new StringEncoder());
                }
            })
    }
}
```

```

    });

    Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();

    while (true) {
        channel.writeAndFlush(new Date() + ": hello world!");
        Thread.sleep(2000);
    }
}
}

```

使用Netty之后，一方面Netty对NIO封装得如此完美，写出来的代码非常优雅，另外一方面，使用Netty之后，网络通信这块的性能问题几乎不用操心

基于Netty自定义RPC

RPC又称远程过程调用，我们所知的远程调用分为两种，现在在服务间通信的方式也基本以这两种为主

1.是基于HTTP的restful形式的广义远程调用，以springboot的feign和restTemplate为代表，采用的协议是HTTP的7层调用协议，并且协议的参数和响应序列化基本以JSON格式和XML格式为主。

2.是基于TCP的狭义的RPC远程调用，以阿里的Dubbo为代表，主要通过netty来实现4层网络协议，NIO来异步传输，序列化也可以是JSON或者hessian2以及java自带的序列化等，可以配置。

接下来我们主要以第二种的RPC远程调用来自行实现

模仿dubbo，消费者和提供者约定接口和协议，消费者远程调用提供者，提供者返回一个字符串，消费者打印提供者返回的数据。底层网络通信使用Netty

步骤

1. 创建一个公共的接口项目以及创建接口及方法，用于消费者和提供者之间的约定。
2. 创建一个提供者，该类需要监听消费者的请求，并按照约定返回数据。
3. 创建一个消费者，该类需要透明的调用自己不存在的方法，内部需要使用Netty请求提供者返回数据

公共模块

首先，在公共模块中添加netty的maven依赖

```

<dependencies>
    <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.16.Final</version>
    </dependency>

```

提供者及消费者工程都需依赖公共模块，这样提供者来实现接口并且提供网络调用，消费者直接通过接口来进行TCP通信及一定的协议定制获取提供者的实现返回值

接口的定义

```
public interface UserService {  
  
    String sayHello(String word);  
}
```

只是一个普通的接口，参数是支持序列化的String类型，返回值同理

提供者的实现

首先是接口的实现，这一点和普通接口实现是一样的

```
public class UserServiceImpl implements UserService {  
  
    @Override  
    public String sayHello(String word) {  
        System.out.println("调用成功--参数: " + word);  
        return "调用成功--参数: " + word;  
    }  
  
    public static void startServer(String hostName, int port) {  
        try {  
            NioEventLoopGroup eventLoopGroup = new NioEventLoopGroup();  
            ServerBootstrap bootstrap = new ServerBootstrap();  
  
            bootstrap.group(eventLoopGroup)  
                .channel(NioServerSocketChannel.class)  
                .childHandler(new ChannelInitializer<SocketChannel>() {  
                    @Override  
                    protected void initChannel(SocketChannel ch) throws  
Exception {  
                        ChannelPipeline p = ch.pipeline();  
                        p.addLast(new StringDecoder());  
                        p.addLast(new StringEncoder());  
                        p.addLast(new UserServerHandler());  
                    }  
                });  
            bootstrap.bind(hostName, port).sync();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

在实现中加入了netty的服务器启动程序，上面的代码中添加了String类型的编解码 handler，添加了一个自定义 handler

自定义 handler 逻辑如下：

```
public class UserServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        // 如何符合约定，则调用本地方法，返回数据
        if (msg.toString().startsWith("UserService")) {
            String result = new UserServiceImpl()

.sayHello(msg.toString().substring(msg.toString().lastIndexOf("#") + 1));
            ctx.writeAndFlush(result);
        }
    }

}
```

这里显示判断了是否符合约定（并没有使用复杂的协议，只是一个字符串判断），然后创建一个具体实现类，并调用方法写回客户端。

还需要一个启动类：

```
public class ServerBootstrap {

    public static void main(String[] args) {
        UserServiceImpl.startServer("localhost", 8990);
    }

}
```

关于提供者的代码就写完了，主要就是创建一个 netty 服务端，实现一个自定义的 handler，自定义 handler 判断是否符合之间的约定（协议），如果符合，就创建一个接口的实现类，并调用他的方法返回字符串。

消费者相关实现

消费者有一个需要注意的地方，就是调用需要透明，也就是说，框架使用者不用关心底层的网络实现。这里我们可以使用 JDK 的动态代理来实现这个目的。

思路：客户端调用代理方法，返回一个实现了 HelloService 接口的代理对象，调用代理对象的方法，返回结果。

我们需要在代理中做手脚，当调用代理方法的时候，我们需要初始化 Netty 客户端，还需要向服务端请求数据，并返回数据。

首先创建代理相关的类：

```
public class RpcConsumer {

    private static ExecutorService executor =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```

```
private static UserClientHandler client;

/**
 * 创建一个代理对象
 */
public Object createProxy(final Class<?> serviceClass, final String
providerName) {
    return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
        new Class<?>[]{serviceClass}, (proxy, method, args) -> {
        if (client == null) {
            initClient();
        }
        // 设置参数
        client.setPara(providerName + args[0]);
        return executor.submit(client).get();
    });
}

/**
 * 初始化客户端
 */
private static void initClient() {
    client = new UserClientHandler();
    EventLoopGroup group = new NioEventLoopGroup();
    Bootstrap b = new Bootstrap();
    b.group(group)
        .channel(NioSocketChannel.class)
        .option(ChannelOption.TCP_NODELAY, true)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws Exception
{
                ChannelPipeline p = ch.pipeline();
                p.addLast(new StringDecoder());
                p.addLast(new StringEncoder());
                p.addLast(client);
            }
        });
    try {
        b.connect("localhost", 8990).sync();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

该类有 2 个方法，创建代理和初始化客户端。

创建代理逻辑：使用 JDK 的动态代理技术，代理对象中的 invoke 方法实现如下：如果 client 没有初始化，则初始化 client，这个 client 既是 handler，也是一个 Callback。将参数设置进 client，使用线程池调用 client 的 call 方法并阻塞等待数据返回

初始化客户端逻辑：创建一个 Netty 的客户端，并连接提供者，并设置一个自定义 handler，和一些 String 类型的序列化方式。

UserClientHandler 的实现：

```
public class UserClientHandler extends ChannelInboundHandlerAdapter implements Callable {

    private ChannelHandlerContext context;
    private String result;
    private String para;

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        context = ctx;
    }

    /**
     * 收到服务端数据，唤醒等待线程
     */
    @Override
    public synchronized void channelRead(ChannelHandlerContext ctx, Object msg) {
        result = msg.toString();
        notify();
    }

    /**
     * 写出数据，开始等待唤醒
     */
    @Override
    public synchronized Object call() throws InterruptedException {
        context.writeAndFlush(para);
        wait();
        return result;
    }

    void setPara(String para) {
        this.para = para;
    }
}
```

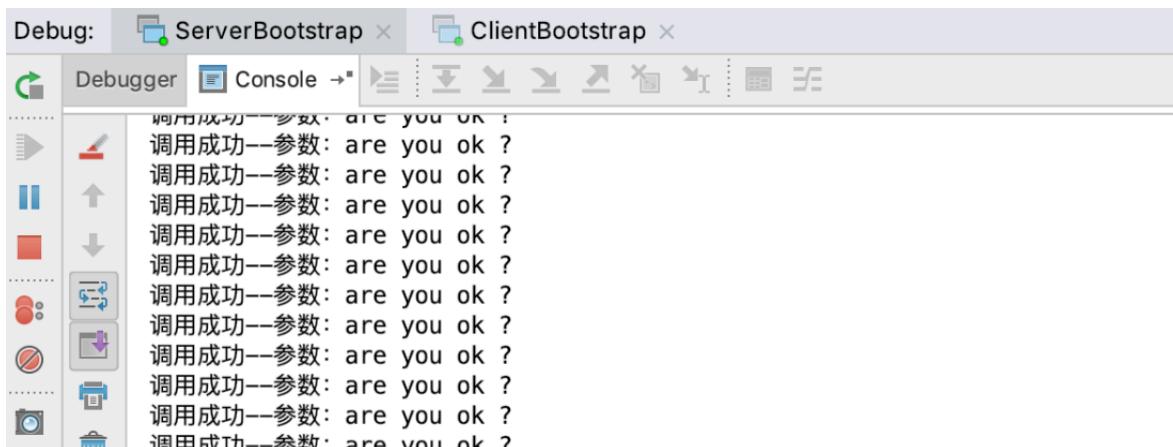
该类缓存了 ChannelHandlerContext，用于下次使用，有两个属性：返回结果和请求参数。

当成功连接后，缓存 ChannelHandlerContext，当调用 call 方法的时候，将请求参数发送到服务端，等待。当服务端收到并返回数据后，调用 channelRead 方法，将返回值赋值个 result，并唤醒等待在 call 方法上的线程。此时，代理对象返回数据。

再看看消费者调用方式，一般的TCP的RPC只需要这样调用即可，无需关心具体的协议和通信方式：

```
public class ClientBootstrap {  
  
    public static final String providerName = "UserService#sayHello";  
  
    public static void main(String[] args) throws InterruptedException {  
        RpcConsumer consumer = new RpcConsumer();  
        // 创建一个代理对象  
        UserService service = (UserService)  
            consumer.createProxy(UserService.class, providerName);  
        for (;;) {  
            Thread.sleep(1000);  
            System.out.println(service.sayHello("are you ok ?"));  
        }  
    }  
}
```

调用者首先创建了一个代理对象，然后每隔一秒钟调用代理的 sayHello 方法，并打印服务端返回的结果



可以看到，消费者无需通过jar包的形式引入具体的实现项目，而是通过远程TCP通信的形式，以一定的协议和代理通过接口直接调用了方法，实现远程service间的调用，是分布式服务的基础

