码农求职小助手：分布式事务和分布式锁

笔记本： 40-微服务 和 分布式
创建时间： 2019/9/16 11:04 **更新时间：** 2019/9/16 11:05
作者： pc941206@163.com

更多资料请关注微信公众号：**码农求职小助手**

一、分布式理论

1、CAP 原理

C：Consistency 即一致性，访问所有的节点得到的数据应该是一样的。注意，这里的一致性指的是强一致性，也就是数据更新完，访问任何节点看到的数据完全一致，要和弱一致性，最终一致性区分开来。

A：Availability 即可用性，所有的节点都保持高可用性。注意，这里的高可用还包括不能出现延迟，比如：如果节点B由于等待数据同步而阻塞请求，那么节点 B 就不满足高可用性。也就是说，任何没有发生故障的服务必须在有限的时间内返回合理的结果集。

P：Partiton tolerence 即分区容忍性，这里的分区是指网络意义上的分区。由于网络是不可靠的，所有节点之间很可能出现无法通讯的情况，在节点不能通信时，要保证系统可以继续正常服务。

CAP 原理说：一个数据分布式系统不可能同时满足 C 和 A 和 P 这3个条件。所以系统架构师在设计系统时，不要将精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。由于网络的不可靠性质，大多数开源的分布式系统都会实现 P，也就是分区容忍性，之后在 C 和 A 中做抉择。

2、Base 理论

CAP 理论告诉我们一个悲惨但不得不接受的事实——我们只能在 C、A、P 中选择两个条件。而对于业务系统而言，我们往往选择牺牲一致性来换取系统的可用性和分区容错性。不过这里要指出的是，所谓的“牺牲一致性”并不是完全放弃数据一致性，而是牺牲强一致性换取弱一致性。下面来介绍下 BASE 理论：

BA：Basic Available 基本可用：整个系统在某些不可抗力的情况下，仍然能够保证“可用性”，即一定时间内仍然能够返回一个明确的结果。只不过“基本可用”和“高可用”的区别是：

1、“一定时间”可以适当延长。当举行大促时，响应时间可以适当延长；

2、给部分用户返回一个降级页面。给部分用户直接返回一个降级页面，从而缓解服务器压力。但要注意，返回降级页面仍然是返回明确结果。

S：Soft State：柔性状态： 同一数据的不同副本的状态，可以不需要实时一致。

E：Eventual Consisstency：最终一致性： 同一数据的不同副本的状态，可以不需要实时一致，但一定要保证经过一定时间后仍然是一致的。

3、酸碱平衡

ACID 能够保证事务的强一致性，即数据是实时一致的。这在本地事务中是没有问题的，**在分布式事务中，强一致性会极大影响分布式系统的性能，因此分布式系统中遵循 BASE 理论即可。**但分布式系统的不同业务场景对一致性的要求也不同。如交易场景下，就要求强一致性，此时就需要遵循 ACID 理论，而在注册成功后发送短信验证码等场景下，并不需要实时一致，因此遵循 BASE 理论即可。因此要根据具体业务场景，在 ACID 和 BASE 之间寻求平衡。

二、分布式事务的基本概念

1、什么是分布式事务?

以下单过程举例：

当我们的系统采用了微服务架构后，一个电商系统往往被拆分成如下几个子系统：商品系统、订单系统、支付系统、积分系统等。整个下单的过程如下：

1、用户通过商品系统浏览商品，他看中了某一项商品，便点击下单；

2、此时订单系统会生成一条订单；

3、订单创建成功后，支付系统提供支付功能；

4、当支付完成后，由积分系统为该用户增加积分。

上述步骤 2、3、4 需要在一个事务中完成。对于传统单体应用而言，实现事务非常简单，只需将这三个步骤放在一个方法 A 中，再用 Spring 的 @Transactional 注解标识该方法即可。Spring 通过数据库的事务支持，保证这些步骤要么全都执行完成，要么全都不执行。但在这个微服务架构中，这三个步骤涉及三个系统，涉及三个数据库，此时我们必须在数据库和应用系统之间，通过某项黑科技，实现分布式事务的支持。

2、两阶段提交协议 2PC

分布式系统的一个难点是如何保证架构下多个节点在进行事务性操作的时候保持一致性。为实现这个目的，二阶段提交算法的成立基于以下假设：

1、该分布式系统中，存在一个节点作为协调者(Coordinator)，其他节点作为参与者(Cohorts)。且节点之间可以进行网络通信。

2、所有节点都采用预写式日志，且日志被写入后即被保持在可靠的存储设备上，即使节点损坏不会导致日志数据的消失。所有节点不会永久性损坏，即使损坏后仍然可以恢复。

1、第一阶段（投票阶段）：

1、协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应；

2、参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）；

3、各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个”同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个”中止”消息。

2、第二阶段（提交执行阶段）：

当协调者节点从所有参与者节点获得的相应消息都为”同意”时：

1、协调者节点向所有参与者节点发出”正式提交(commit)”的请求；

2、参与者节点正式完成操作，并释放在整个事务期间内占用的资源；

3、参与者节点向协调者节点发送”完成”消息；

4、协调者节点受到所有参与者节点反馈的”完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为”中止”，或者协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：

1、协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求；

2、参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源；

3、参与者节点向协调者节点发送”回滚完成”消息；

4、协调者节点受到所有参与者节点反馈的”回滚完成”消息后，取消事务。

不管最后结果如何，第二阶段都会结束当前事务。

二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个缺点的：

1、执行过程中，**所有参与节点都是事务阻塞型的。**当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态；

2、参与者发生故障：协调者需要给每个参与者额外指定超时机制，超时后整个事务失败。（没有多少容错机制）；

3、协调者发生故障：参与者会一直阻塞下去。需要额外的备机进行容错。（这个可以依赖后面要讲的 Paxos 协议实现 HA）；

4、二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

3、三阶段提交协议 3PC

与两阶段提交不同的是，三阶段提交有两个改动点：

1、引入超时机制。同时在协调者和参与者中都引入超时机制；

2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 **CanCommit、PreCommit、DoCommit** 三个阶段。

1、CanCommit 阶段

3PC 的 CanCommit 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 commit 请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

1、事务询问：协调者向参与者发送 CanCommit 请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

2、响应反馈：参与者接到 CanCommit 请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回 Yes 响应，并进入预备状态。否则反馈 No。

2、PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以继续事务的 PreCommit 操作。根据响应情况，有以下两种可能：

假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行。

1、发送预提交请求：协调者向参与者发送 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、响应反馈：如果参与者成功的执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

1、发送中断请求：协调者向所有参与者发送 abort 请求。

2、中断事务：参与者收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

3、doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

3.1 执行提交

1、发送提交请求：协调接收到参与者发送的 ACK 响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。

2、事务提交：参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

3、响应反馈：事务提交完之后，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的 ACK 响应之后，完成事务。

3.2 中断事务

协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

1、发送中断请求：协调者向所有参与者发送 abort 请求。

2、事务回滚：参与者接收到 abort 请求之后，利用其在阶段二记录的 undo 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

3、反馈结果：参与者完成事务回滚之后，向协调者发送 ACK 消息。

4、中断事务：协调者接收到参与者反馈的 ACK 消息之后，执行事务的中断。

三阶段提交的问题：

网络分区可能会带来问题。需要四阶段解决：四阶段直接调用远程服务的数据状态，确定当前数据一致性的情况。

4、TCC 模式

TCC 模式本质也是 2PC，只是 TCC 在应用层控制，数据库只是负责第一个阶段。XA 在数据库层控制两阶段提交。

TCC 是对二阶段的一个改进，try 阶段通过预留资源的方式避免了同步阻塞资源的情况，但是 TCC 编程需要业务自己实现try、confirm、cancle 方法，对业务入侵太大，实现起来也比较复杂。

分布式事务的基本原理本质上都是两阶段提交协议（2PC），TCC （try-confirm-cancel）其实也是一种 2PC，只不过 TCC 规定了在服务层面实现的具体细节，即参与分布式事务的服务方和调用方至少要实现三个方法：try 方法、confirm 方法、cancel 方法。

5、Paxos

所有的分布式一致性算法都是 paxos 算法的部分，只有 paxos 算法才能完美实现分布式一致性。

6、ZAB 协议

- ZAB 协议 (Zookeeper Atomic Broadcast) 原子广播协议

三、分布式事务的解决方案

分布式事务的解决方案有如下几种：

1、全局消息；

2、基于可靠消息服务的分布式事务；

3、最大努力通知；

4、TCC。

方案 1：全局事务（DTP 模型）

全局事务基于 DTP 模型实现。它规定了要实现分布式事务，需要三种角色：

AP：Application 应用系统

它就是我们开发的业务系统，在我们开发的过程中，可以使用资源管理器提供的事务接口来实现分布式事务。

TM：Transaction Manager 事务管理器

1、分布式事务的实现由事务管理器来完成，它会提供分布式事务的操作接口供我们的业务系统调用。这些接口称为 TX 接口；

2、事务管理器还管理着所有的资源管理器，通过它们提供的 XA 接口来统一调度这些资源管理器，以实现分布式事务；

3、DTP 只是一套实现分布式事务的规范，并没有定义具体如何实现分布式事务，TM 可以采用 2PC、3PC、Paxos 等协议实现分布式事务。

RM：Resource Manager 资源管理器

1、能够提供数据服务的对象都可以是资源管理器，比如：数据库、消息中间件、缓存等。大部分场景下，数据库即为分布式事务中的资源管理器。

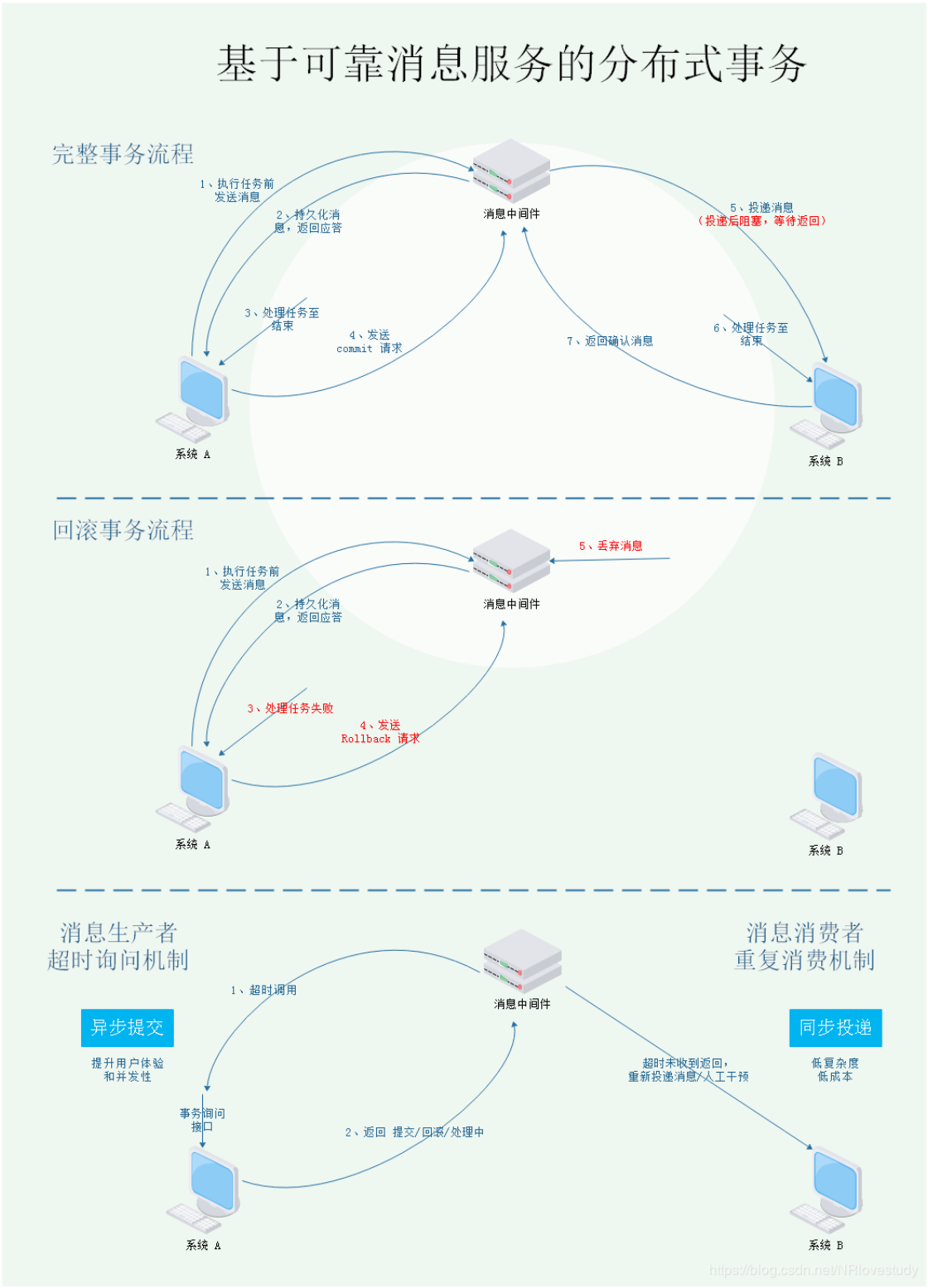
2、资源管理器能够提供单数据库的事务能力，它们通过 XA 接口，将本数据库的提交、回滚等能力提供给事务管理器调用，以帮助事务管理器实现分布式的事务管理。

3、XA 是 DTP 模型定义的接口，用于向事务管理器提供该资源管理器(该数据库)的提交、回滚等能力。

4、DTP 只是一套实现分布式事务的规范，RM 具体的实现是由数据库厂商来完成的。

方案 2：基于可靠消息服务【处理任务前-发消息】

这种实现分布式事务的方式需要通过消息中间件来实现。假设有 A 和 B 两个系统，分别可以处理任务 A 和任务 B。此时系统 A 中存在一个业务流程，需要将任务 A 和任务 B 在同一个事务中处理。下面来介绍基于消息中间件来实现这种分布式事务。



1、完整事务流程

1、在系统 A 处理任务 A 前，首先向消息中间件发送一条消息；

2、消息中间件收到后将该条消息持久化，但并不投递。此时下游系统 B 仍然不知道该条消息的存在。消息中间件持久化成功后，便向系统 A 返回一个确认应答；

3、系统 A 收到确认应答后，则可以开始处理任务 A；

4、任务 A 处理完成后，向消息中间件发送 Commit 请求。该请求发送完成后，对系统 A 而言，该事务的处理过程就结束了，此时它可以处理别的任务了。（）commit 消息可能会在传输途中丢失，从而消息中间件并不会向系统 B 投递这条消息，从而系统就会出现不一致性。这个问题由消息中间件的事务回查机制完成，下文会介绍）；

5、消息中间件收到 Commit 指令后，便向系统 B 投递该消息；

6、系统 B 执行任务；

7、当任务 B 执行完成后，向消息中间件返回一个确认应答，告诉消息中间件该消息已经成功消费，此时，这个分布式事务完成。

上述过程可以得出如下几个结论：

1、消息中间件扮演者分布式事务协调者的角色；

2、系统 A 完成任务 A 后，到任务 B 执行完成之间，会存在一定的时间差。在这个时间差内，整个系统处于数据不一致的状态，但这短暂的不一致性是可以接受的，因为经过短暂的时间后，系统又可以保持数据一致性，满足 BASE 理论。

2、回滚流程

上述过程中，如果任务 A 处理失败，那么需要进入回滚流程：

1、系统 A 在处理任务 A 时失败；

2、系统 A 向消息中间件发送 Rollback 请求。和发送 Commit 请求一样，系统 A 发完之后便可以认为回滚已经完成，它便可以去做其他的事情；

3、消息中间件收到回滚请求后，直接将该消息丢弃，而不投递给系统 B，从而不会触发系统 B 的任务 B。

此时系统又处于一致性状态，因为任务 A 和任务 B 都没有执行。

3、消息生产者：超时询问机制

上面所介绍的 Commit 和 Rollback 都属于理想情况，但在实际系统中，Commit 和 Rollback 指令都有可能在传输途中丢失。那么当出现这种情况的时候，消息中间件是如何保证数据一致性呢？——答案就是：**超时询问机制。**

系统 A 除了实现正常的业务流程外，还需提供一个事务询问的接口，供消息中间件调用。当消息中间件收到一条事务型消息后便开始计时，如果到了超时时间也没收到系统 A 发来的 Commit 或 Rollback 指令的话，就会主动调用系统 A 提供的事务询问接口询问该系统目前的状态。该接口会返回三种结果：

1、提交： 若获得的状态是“提交”，则将该消息投递给系统 B。

2、回滚： 若获得的状态是“回滚”，则直接将条消息丢弃。

3、处理中： 若获得的状态是“处理中”，则继续等待。

消息中间件的超时询问机制能够防止上游系统因在传输过程中丢失 Commit/Rollback 指令而导致的系统不一致情况，而且能降低上游系统的阻塞时间，上游系统只要发出 Commit/Rollback 指令后便可以处理其他任务，无需等待确认应答。而 Commit/Rollback 指令丢失的情况通过超时询问机制来弥补，这样大大降低上游系统的阻塞时间，提升系统的并发度。

4、消息消费者：重复消费机制

下面来说一说消息投递过程的可靠性保证。

当上游系统执行完任务并向消息中间件提交了 Commit 指令后，便可以处理其他任务了，此时它可以认为事务已经完成，接下来消息中间件一定会保证消息被下游系统成功消费掉！那么这是怎么做到的呢？这由消息中间件的投递流程来保证。

消息中间件向下游系统投递完消息后便进入阻塞等待状态，下游系统便立即进行任务的处理，任务处理完成后便向消息中间件返回应答。消息中间件收到确认应答后便认为该事务处理完毕！

如果消息在投递过程中丢失，或消息的确认应答在返回途中丢失，那么消息中间件在等待确认应答超时之后就会重新投递，直到下游消费者返回消费成功响应为止。当然，一般消息中间件可以设置消息重试的次数和时间间隔，比如：当第一次投递失败后，每隔五分钟重试一次，一共重试 3 次。如果重试 3 次之后仍然投递失败，那么这条消息就需要人工干预。

5、消息投递失败后为什么不回滚消息，而是不断尝试重新投递？

这就涉及到整套分布式事务系统的实现成本问题。

我们知道，当系统 A 将向消息中间件发送 Commit 指令后，它便去做别的事情了。如果此时消息投递失败，需要回滚的话，就需要让系统 A 事先提供回滚接口，这无疑增加了额外的开发成本，业务系统的复杂度也将提高。对于一个业务系统的设计目标是，在保证性能的前提下，最大限度地降低系统复杂度，从而能够降低系统的运维成本。

不知大家是否发现，上游系统 A 向消息中间件提交 Commit / Rollback 消息采用的是异步方式，也就是当上游系统提交完消息后便可以去做别的事情，接下来提交、回滚就完全交给消息中间件来完成，并且完全信任消息中间件，认为它一定能正确地完成事务的提交或回滚。然而，消息中间件向下游系统投递消息的过程是同步的。也就是消息中间件将消息投递给下游系统后，它会阻塞等待，等下游系统成功处理完任务返回确认应答后才取消阻塞等待。为什么这两者在设计上是不一致的呢？

首先，**上游系统和消息中间件之间采用异步通信是为了提高系统并发度。业务系统直接和用户打交道，用户体验尤为重要，因此这种异步通信方式能够极大程度地降低用户等待时间。**此外，异步通信相对于同步通信而言，没有了长时间的阻塞等待，因此系统的并发性也大大增加。**但异步通信可能会引起 Commit/Rollback 指令丢失的问题，这就由消息中间件的超时询问机制来弥补。**

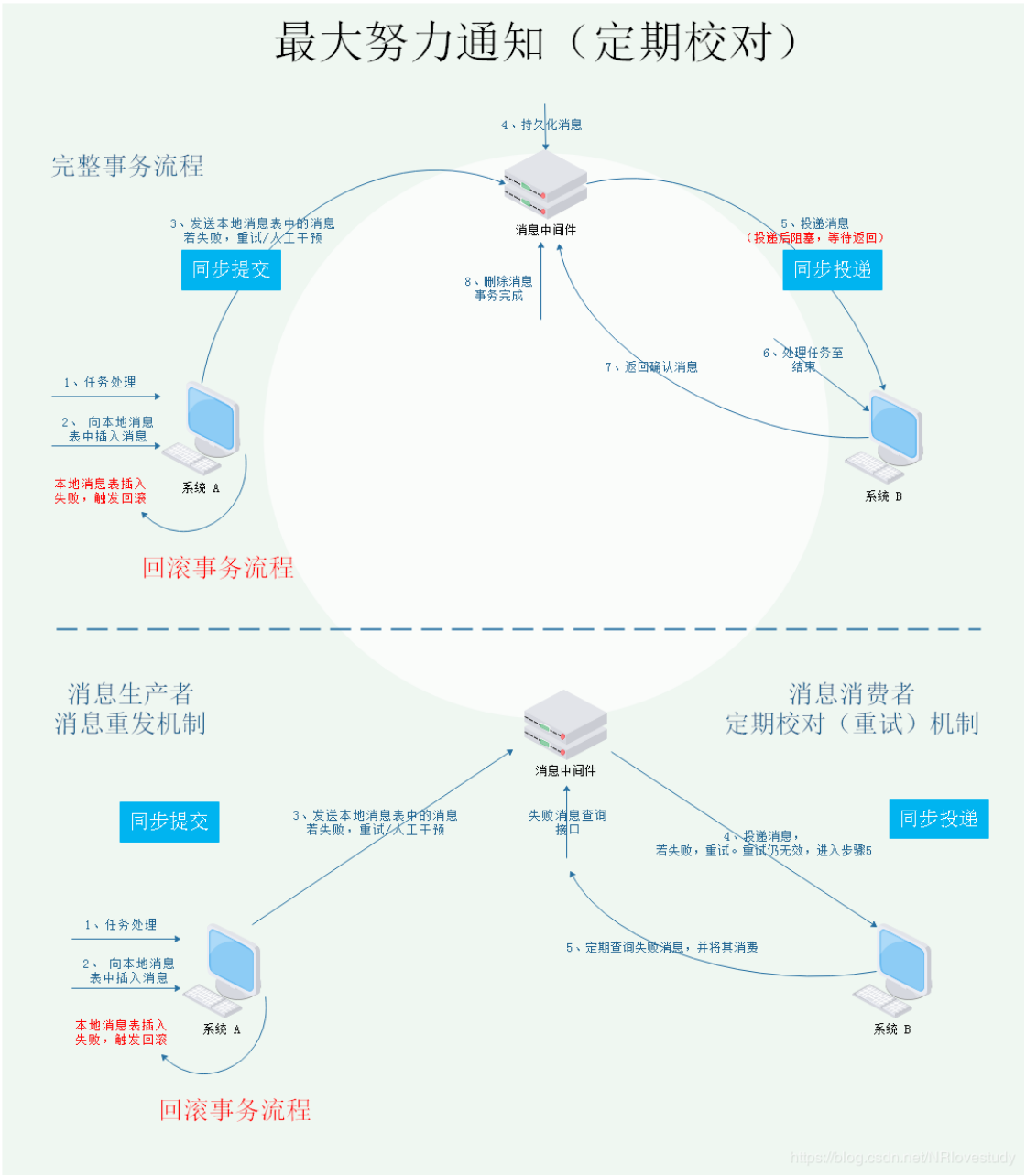
6、消息中间件和下游系统之间为什么要采用同步通信呢?

异步能提升系统性能，但随之会增加系统复杂度；而同步虽然降低系统并发度，但实现成本较低。因此，在对并发度要求不是很高的情况下，或者服务器资源较为充裕的情况下，我们可以选择同步来降低系统的复杂度。

我们知道，消息中间件是一个独立于业务系统的第三方中间件，它不和任何业务系统产生直接的耦合，它也不和用户产生直接的关联，它一般部署在独立的服务器集群上，具有良好的可扩展性，所以不必太过于担心它的性能，如果处理速度无法满足我们的要求，可以增加机器来解决。而且，即使消息中间件处理速度有一定的延迟那也是可以接受的，因为前面所介绍的 BASE 理论就告诉我们了，我们追求的是最终一致性，而非实时一致性，因此消息中间件产生的时延导致事务短暂的不一致是可以接受的。

方案 3：基于可靠服务的最大努力通知（定期校对）【任务完成后-发消息】

最大努力通知也被称为定期校对，其实在方案二中已经包含，这里再单独介绍，主要是为了知识体系的完整性。这种方案也需要消息中间件的参与，其过程如下：



1、上游系统在完成任务后，向消息中间件同步地发送一条消息，确保消息中间件成功持久化这条消息，然后上游系统可以去做别的事情了；

2、消息中间件收到消息后负责将该消息同步投递给相应的下游系统，并触发下游系统的任务执行；

3、当下游系统处理成功后，向消息中间件反馈确认应答，消息中间件便可以将该条消息删除，从而该事务完成。

上面是一个理想化的过程，但在实际场景中，往往会出现如下几种意外情况：

（1）上游系统向消息中间件发送消息失败；

（2）消息中间件向下游系统投递消息失败。

对于第 1 种情况：**需要在上游系统中建立消息重发机制。**可以在上游系统建立一张本地消息表，并将 任务处理过程 和 向本地消息表中插入消息 这两个步骤放在一个本地事务中完成。如果向本地消息表插入消息失败，那么就会触发回滚，之前的任务处理结果就会

被取消。如果这两步都执行成功，那么该本地事务就完成了。接下来会有一个专门的消息发送者不断地发送本地消息表中的消息，如果发送失败它会返回重试。当然，也要给消息发送者设置重试的上限，一般而言，达到重试上限仍然发送失败，那就意味着消息中间件出现严重的问题，此时也只有人工干预才能解决问题。

对于第 2 种情况：消息中间件具有重试机制，我们可以在消息中间件中设置消息的重试次数和重试时间间隔，对于网络不稳定导致的消息投递失败的情况，往往重试几次后消息便可以成功投递，如果超过了重试的上限仍然投递失败，那么消息中间件不再投递该消息，而是记录在失败消息表中，消息中间件需要提供失败消息的查询接口，下游系统会定期查询失败消息，并将其消费，这就是所谓的“**定期校对**”。如果重复投递和定期校对都不能解决问题，往往是因为下游系统出现了严重的错误，此时就需要人工干预。

对于不支持事务型消息的消息中间件，如果要实现分布式事务的话，就可以采用这种方式。它能够通过 **重试机制+定期校对** 实现分布式事务，但相比于第二种方案，它达到数据一致性的周期较长，而且还需要在上游系统中实现消息重试发布机制，以确保消息成功发布给消息中间件，这无疑增加了业务系统的开发成本，使得业务系统不够纯粹，并且这些额外的业务逻辑无疑会占用业务系统的硬件资源，从而影响性能。

因此，尽量选择支持事务型消息的消息中间件来实现分布式事务，如 RocketMQ。【优先选择 方案 2】

方案 4：TCC（两阶段型、补偿型）

从分布式事务的需求来源来看：

1、跨数据库

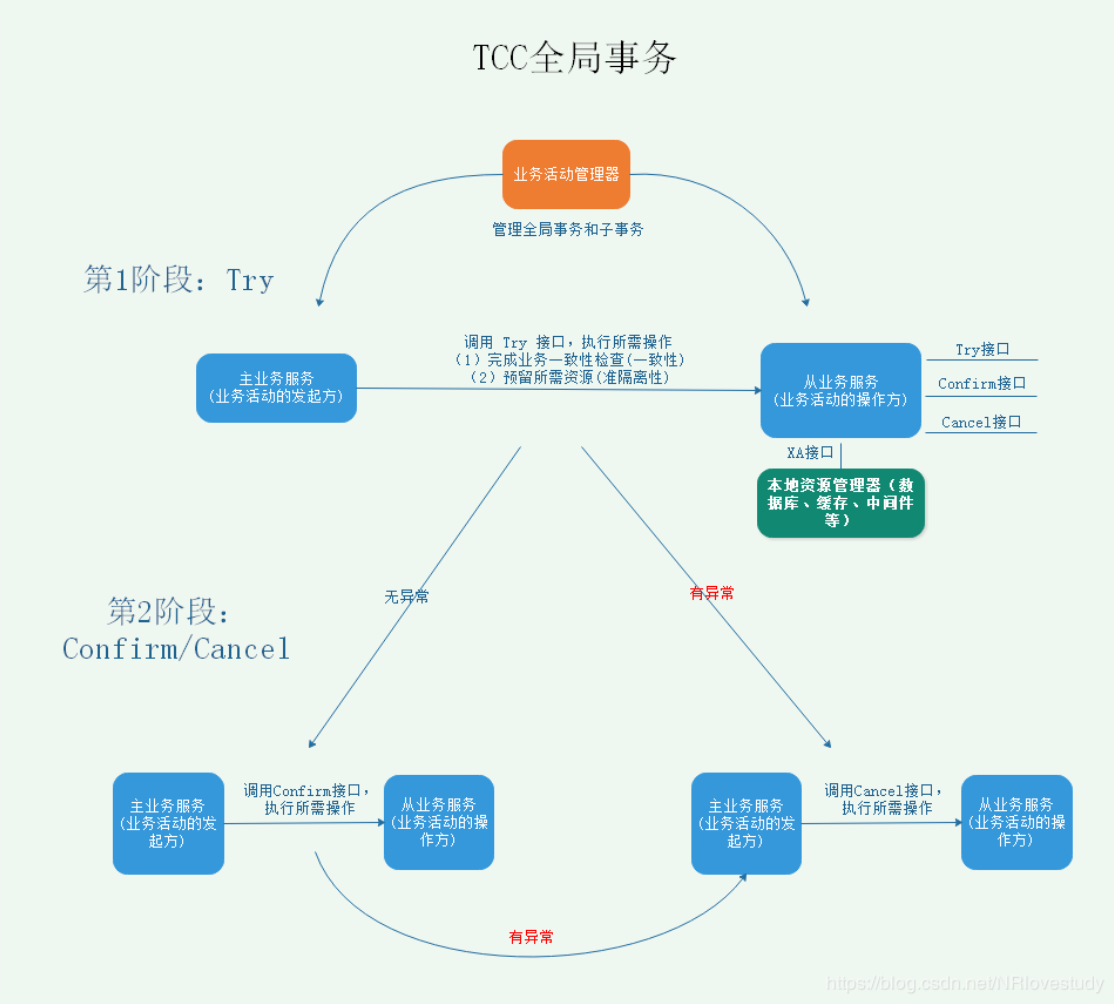
- 数据库拆分(水平、垂直)带来的分布式事务->保证跨库操作的原子性
- 基于单个 JVM

2、跨应用

- 应用拆分带来的分布式事务->保证跨应用业务操作的原子性
- 跨 JVM

跨应用的业务操作原子性要求，其实是比较常见的。比如在第三方支付场景中的组合支付，用户在电商网站购物后，要同时使用余额和红包支付该笔订单，而余额系统和红包系统分别是不同的应用系统，支付系统在调用这两个系统进行支付时，就需要保证余额扣减和红包使用要么同时成功，要么同时失败。

TCC 事务的出现正是为了解决应用拆分带来的跨应用业务操作原子性的问题。



TCC 即为Try Confirm Cancel，它属于补偿型分布式事务，是应用层的 2PC(2 Phase Commit, 两阶段提交)。

4.1 三个步骤：

TCC 实现分布式事务一共有三个步骤：

1、Try：尝试待执行的业务：这个过程并未执行业务，只是完成所有业务的一致性检查，并预留好执行所需的全部资源；

2、Confirm：执行业务：这个过程真正开始执行业务，由于 Try 阶段已经完成了一致性检查，因此本过程直接执行，而不做任何检查。并且在执行的过程中，会使用到 Try 阶段预留的业务资源；

3、Cancel：取消执行的业务：若业务执行失败，则进入 Cancel 阶段，它会释放所有占用的业务资源，并回滚 Confirm 阶段执行的操作。

4.2 三个参与方：

一个完整的 TCC 事务参与方包括三部分：

1、**主业务服务**：主业务服务为整个业务活动的发起方，如前面提到的组合支付场景，支付系统即是主业务服务；

2、**从业务服务**：从业务服务负责提供 TCC 业务操作，是整个业务活动的操作方。从业务服务必须实现 Try、Confirm 和Cancel 三个接口，供主业务服务调用。由于 Confirm 和 Cancel 操作可能被重复调用，故**要求 Confirm 和 Cancel 两个接口必须是幂等的。**前面的组合支付场景中的余额系统和红包系统即为从业务服务。

3、**业务活动管理器**：业务活动管理器管理控制整个业务活动，包括记录维护 TCC 全局事务的事务状态和每个从业务服务的子事务状态，并在业务活动提交时确认所有的 TCC 型操作的 confirm 操作，在业务活动取消时调用所有 TCC 型操作的 cancel 操作。

可见整个 TCC 事务对于主业务服务来说是透明的，其中业务活动管理器和从业务服务各自干了一部分工作。

4.3 RM 本地事务支持

TCC 必须基于 RM 本地事务来实现全局事务。TCC 服务是由 Try/Confirm/Cancel 业务构成的，其 Try/Confirm/Cancel 业务在执行时，会访问资源管理器（Resource Manager，下文简称 RM）来存取数据。这些存取操作，必须要参与 RM 本地事务，以使其更改的数据要么都commit，要么都 rollback。

4.4 TCC 的优点和限制

- **优点**：

  解决了跨应用业务操作的原子性问题，在诸如组合支付、账务拆分场景非常实用。

  TCC 实际上把数据库层的二阶段提交上提到了应用层来实现，对于数据库来说是一阶段提交，规避了数据库层的 2PC 性能低下问题。

- **缺点**

  主要就一个：TCC 的 Try、Confirm 和 Cancel 操作功能需业务提供，开发成本高。

四、分布式锁

https://www.cnblogs.com/seesun2012/p/9214653.html
https://blog.csdn.net/wuzhiwei549/article/details/80692278

1、什么是分布式锁?

1.1 什么是锁?

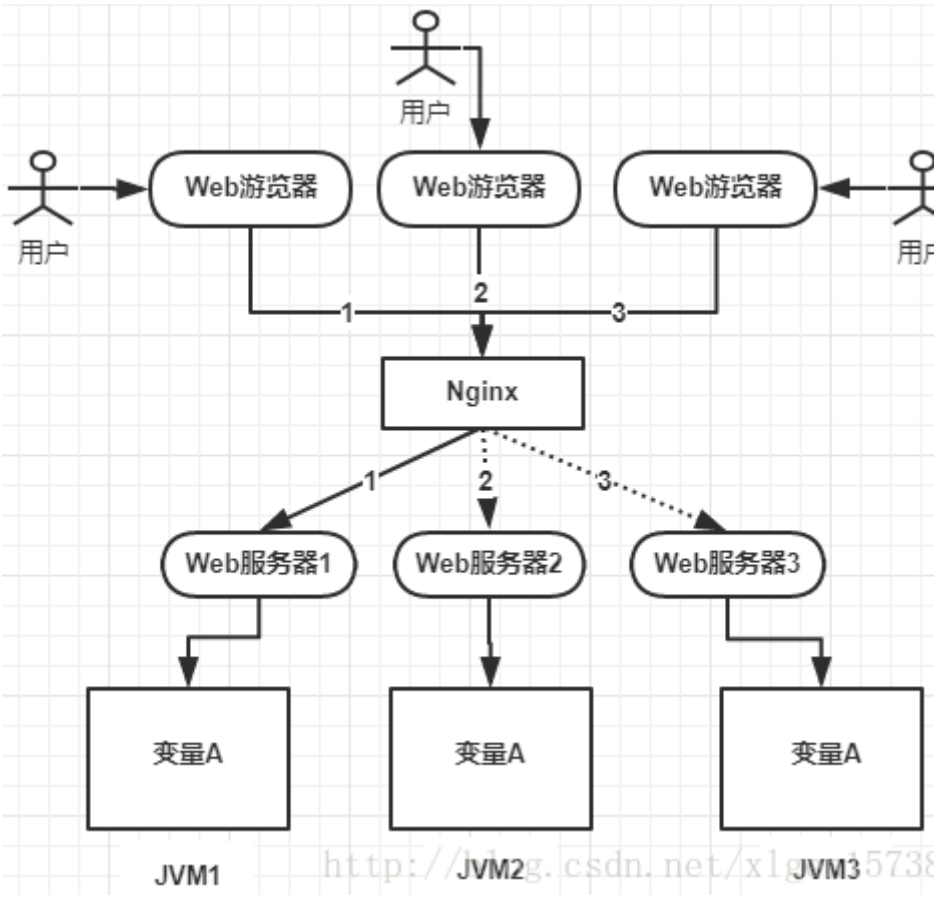
在单进程的系统中，当存在多个线程可以同时改变某个变量（可变共享变量）时，就需要对变量或代码块做同步，使其在修改这种变量时能够线性执行消除并发修改变量。

而**同步的本质是通过锁来实现的。**为了实现多个线程在一个时刻同一个代码块只能有一个线程可执行，那么需要在某个地方做个标记，这个标记必须每个线程都能看到，当标记不存在时可以设置该标记，其余后续线程发现已经有标记了则等待拥有标记的线程结束同步代码块取消标记后再去尝试设置标记。这个标记可以理解为锁。

不同地方实现锁的方式也不一样，只要能满足所有线程都能看得到标记即可。如 Java 中 synchronized 是在对象头设置标记，Lock 接口的实现类基本上都只是某一个 volitile 修饰的 int 型变量其保证每个线程都能拥有对该 int 的可见性和原子修改，linux 内核中也是利用互斥量或信号量等内存数据做标记。

1.2 什么是分布式锁?

> **分布式与单机情况下最大的不同在于其不是多线程而是多进程。** 多线程由于可以共享堆内存，因此可以简单的采取内存作为标记存储位置。而进程之间甚至可能都不在同一台物理机上，因此需要将标记存储在一个所有进程都能看到的地方。



如上图所示：一个应用需要部署到几台机器上然后做负载均衡。

变量 A 存在 JVM1、JVM2、JVM3 三个 JVM 内存中（这个变量 A 主要体现是在一个类中的一个成员变量，是一个有状态的对象，例如：UserController 控制器中的一个整形类型的成员变量），如果不加任何控制的话，变量 A 同时都会在 JVM 分配一块内存，三个请求发过来同时对这个变量操作，显然结果是不对的。即使不是同时发过来，三个请

求分别操作三个不同 JVM 内存区域的数据，变量 A 之间不存在共享，也不具有可见性，处理的结果也是不对的。

上面这种情况再依靠传统的锁机制是无法解决的。为了解决这个问题就需要一种跨 JVM 的互斥机制来控制共享资源的访问，这就是分布式锁要解决的问题。

1.3 分布式锁应该具备哪些条件?

1、在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；

2、高可用的获取锁与释放锁；

3、高性能的获取锁与释放锁；

4、具备可重入特性；

5、具备锁失效机制，防止死锁；

6、具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

2、分布式锁的三种实现方式

为了保证一个方法在同一时间内只能被同一个线程执行。可以有以下三种实现方式：

1、基于数据库实现分布式锁；

2、基于缓存（Redis 等）实现分布式锁；

3、基于 Zookeeper 实现分布式锁。

2.1 基于数据库做分布式锁

基于数据库做分布式锁的场景不太多，下面只是简单列下几种实现方式，具体的实现可以参照：
https://www.cnblogs.com/seesun2012/p/9214653.html

- **1、基于表主键唯一性做分布式锁**

利用主键唯一的特性，如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，当方法执行完毕之后，想要释放锁的话，删除这条数据库记录即可。

- **2、基于表字段版本号做分布式锁**

这个策略源于 mysql 的 mvcc 机制，使用这个策略其实本身没有什么问题，唯一的问题就是对数据表侵入较大，我们要为每个表设计一个版本号字段，然后写一条判断 sql 每次进行判断，增加了数据库操作的次数，在高并发的要求下，对数据库连接的开销也是无法忍受的。

- **3、基于数据库排它锁做分布式锁**

在查询语句后面增加 for update，数据库会在查询过程中给数据库表增加排他锁 (注意： InnoDB 引擎在加锁的时候，只有通过索引进行检索的时候才会使用行级锁，否则会使用表级锁。这里我们希望使用行级锁，就要给要执行的方法字段名添加索引，值得注意的是，这个索引一定要创建成唯一索引，否则会出现多个重载方法之间无法同时被访问的问题。重载方法的话建议把参数类型也加上。)。当某条记录被加上排他锁之后，其他线程无法再在该行记录上增加排他锁。

2.2 基于 Redis 做分布式锁

- **1、基于 Redis 的 SETNX()、EXPIRE() 方法做分布式锁**

setnx()： 含义就是 SET if Not Exists，其主要有两个参数 setnx(key, value)。该方法是原子的，如果 key 不存在，则设置当前 key 成功，返回 1；如果当前 key 已经存在，则设置当前 key 失败，返回 0。

expire()： 设置过期时间，要注意的是 setnx 命令不能设置 key 的超时时间，只能通过 expire() 来对 key 设置。

使用步骤：

1、setnx(lockkey, 1) 如果返回 0，则说明占位失败；如果返回 1，则说明占位成功；

2、expire() 命令对 lockkey 设置超时时间，为的是避免死锁问题；

3、执行完业务代码后，可以通过 delete 命令删除 key。

问题：

这个方案其实是可以解决日常工作中的需求的，但从技术方案的探讨上来说，可能还有一些可以完善的地方。比如，如果在第一步 setnx 执行成功后，在 expire() 命令执行成功前，发生了宕机的现象，那么就依然会出现死锁的问题，所以如果要对其进行完善的话，可以使用 redis 的 setnx()、get() 和 getset() 方法来实现分布式锁。

- **2、基于 REDIS 的 SETNX()、GET()、GETSET()方法做分布式锁**

这个方案的背景主要是在 setnx() 和 expire() 的方案上针对可能存在的死锁问题，做了一些优化。

getset()： 这个命令主要有两个参数 getset(key，newValue)。该方法是原子的，对 key 设置 newValue 这个值，并且返回 key 原来的旧值。假设 key 原来是不存在的，那么多次执行这个命令，会出现下边的效果：

- getset(key, “value1”) 返回 null 此时 key 的值会被设置为 value1
- getset(key, “value2”) 返回 value1 此时 key 的值会被设置为 value2
- 依次类推！

使用步骤：

1、setnx(lockkey, 当前时间+过期超时时间)，如果返回 1，则获取锁成功；如果返回 0 则没有获取到锁，转向 2；

2、get(lockkey) 获取值 oldExpireTime ，并将这个 value 值与当前的系统时间进行比较，如果小于当前系统时间，则认为这个锁已经超时，可以允许别的请求重新获取，转向3；

3、计算 newExpireTime = 当前时间+过期超时时间，然后 getset(lockkey, newExpireTime) 会返回当前 lockkey 的值currentExpireTime。

4、判断 currentExpireTime 与 oldExpireTime 是否相等，如果相等，说明当前 getset 设置成功，获取到了锁。如果不相等，说明这个锁又被别的请求获取走了，那么当前请求可以直接返回失败，或者继续重试。

5、在获取到锁之后，当前线程可以开始自己的业务处理，当处理完毕后，比较自己的处理时间和对于锁设置的超时时间，如果小于锁设置的超时时间，则直接执行 delete 释放锁；如果大于锁设置的超时时间，则不需要再锁进行处理。

- **3、基于 REDLOCK 做分布式锁**

Redlock 是 Redis 的作者 antirez 给出的集群模式的 Redis 分布式锁，它基于 N 个完全独立的 Redis 节点（通常情况下 N 可以设置成 5）。

- 算法的步骤如下：

1、客户端获取当前时间，以毫秒为单位；

2、客户端尝试获取 N 个节点的锁，（每个节点获取锁的方式和前面说的缓存锁一样），N 个节点以相同的 key 和 value 获取锁。客户端需要设置接口访问超时，接口超时时间需要远远小于锁超时时间，比如锁自动释放的时间是 10s，那么接口超时大概设置 5-50ms。这样可以在有 redis 节点宕机后，访问该节点时能尽快超时，而减小锁的正常使用。

3、客户端计算在获得锁的时候花费了多少时间，方法是用当前时间减去在步骤一获取的时间，只有客户端获得了超过 3 个节点的锁，而且获取锁的时间小于锁的超时时间，客户端才获得了分布式锁；

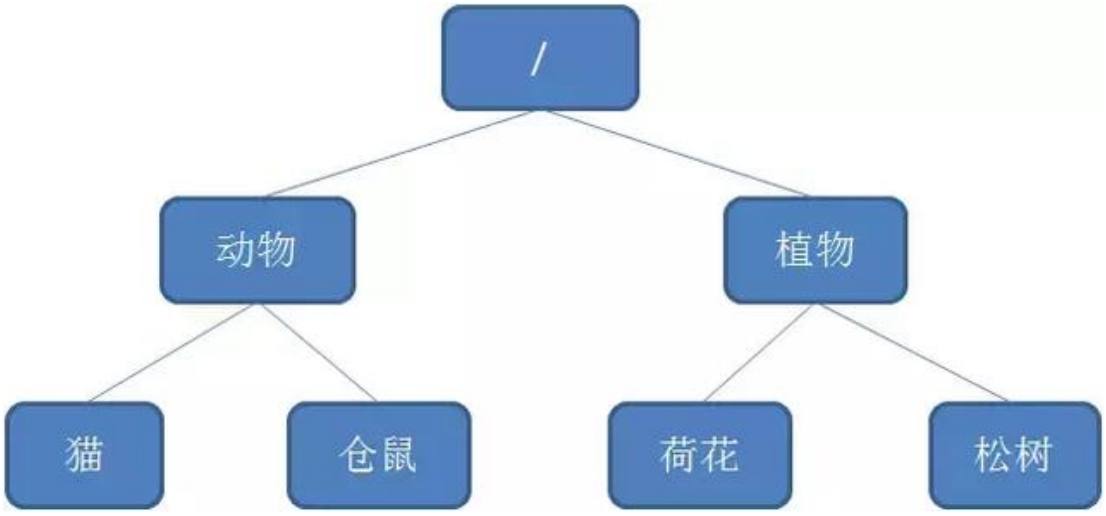
4、客户端获取的锁的时间为设置的锁超时时间减去步骤三计算出的获取锁花费时间；

5、如果客户端获取锁失败了，客户端会依次删除所有的锁。

使用 Redlock 算法，可以保证在挂掉最多 2 个节点的时候，分布式锁服务仍然能工作，这相比之前的数据库锁和缓存锁大大提高了可用性，由于 redis 的高效性能，分布式缓存锁性能并不比数据库锁差。

2.3 基于 Zookeeper 做分布式锁

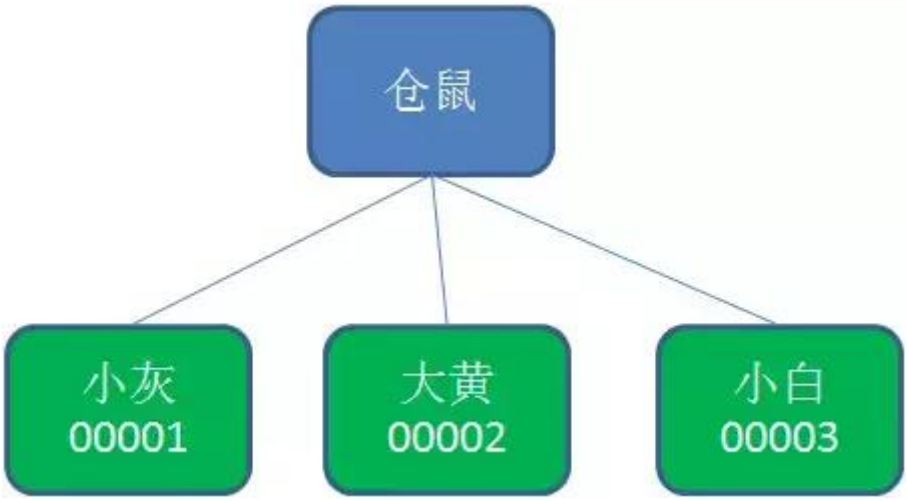
让我们来回顾一下 Zookeeper 节点的概念：



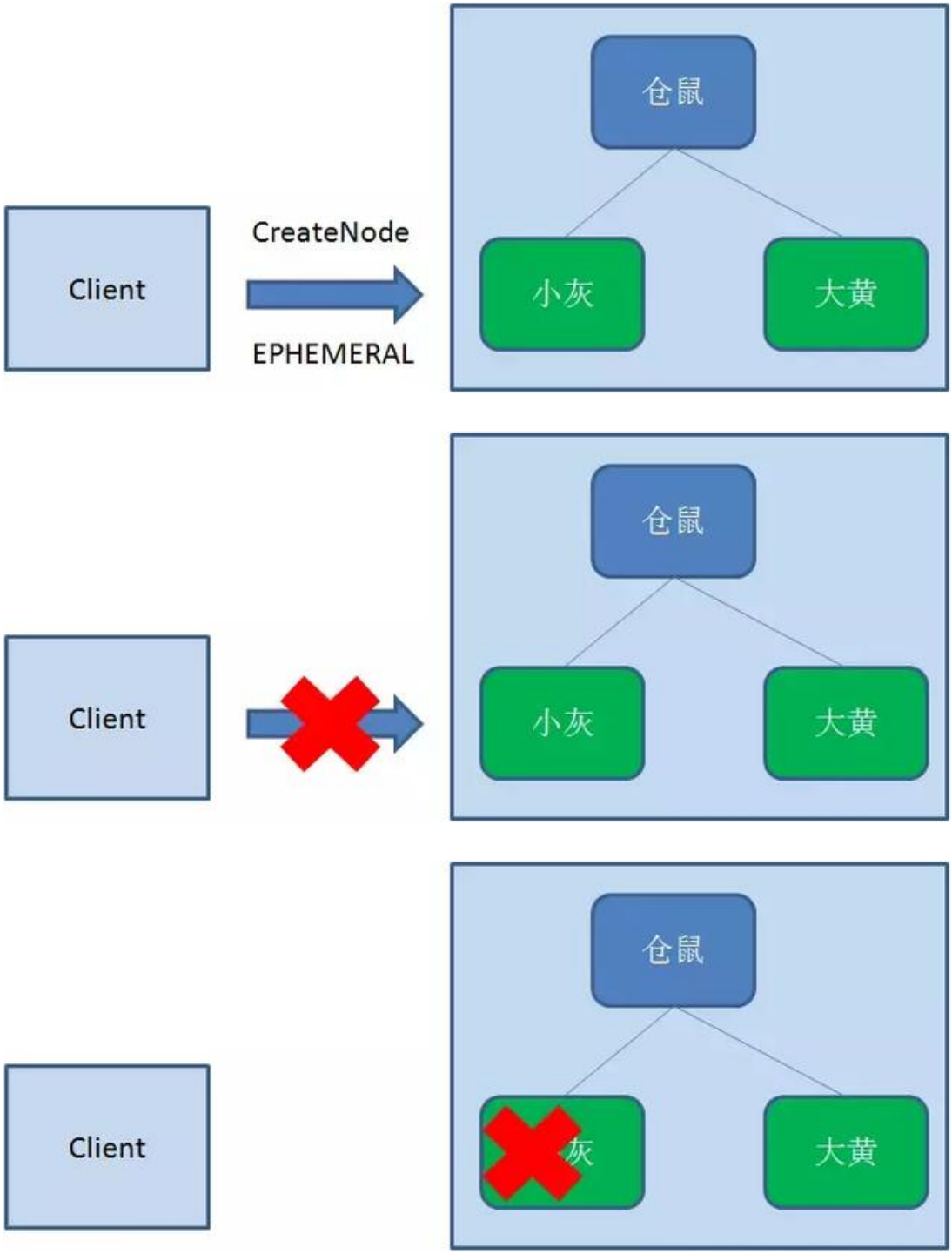
Zookeeper 的数据存储结构就像一棵树，这棵树由节点组成，这种节点叫做 Znode。Znode 分为以下四种类型：

1、持久节点 （PERSISTENT）：默认的节点类型。创建节点的客户端与 Zookeeper 断开连接后，该节点依旧存在 。

2、持久节点顺序节点（PERSISTENT_SEQUENTIAL）：所谓顺序节点，就是在创建节点时，Zookeeper 根据创建的时间顺序给该节点名称进行编号：



3、临时节点（EPHEMERAL）：和持久节点相反，当创建节点的客户端与 zookeeper 断开连接后，临时节点会被删除：



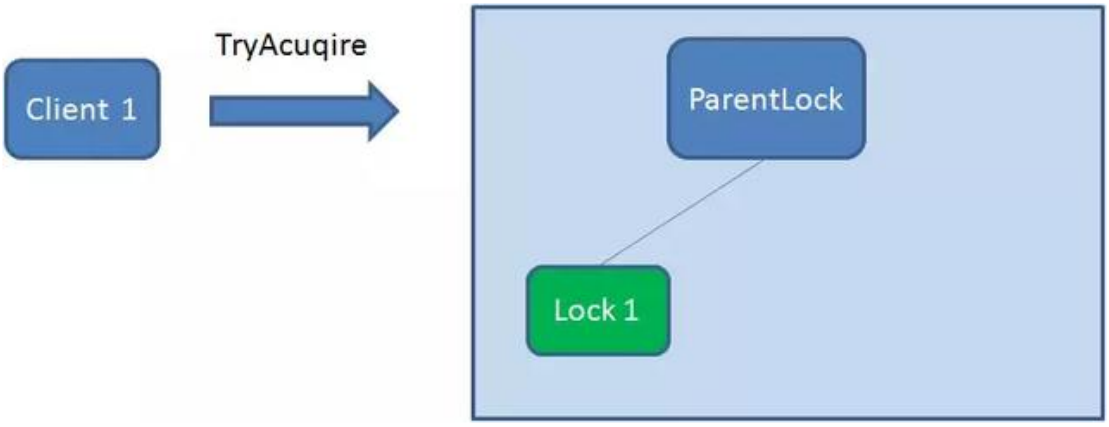
4、临时顺序节点（EPHEMERAL_SEQUENTIAL）：顾名思义，临时顺序节点结合和临时节点和顺序节点的特点：在创建节点时，Zookeeper 根据创建的时间顺序给该节点名称进行编号；当创建节点的客户端与 Zookeeper 断开连接后，临时节点会被删除。

Zookeeper 分布式锁的原理：

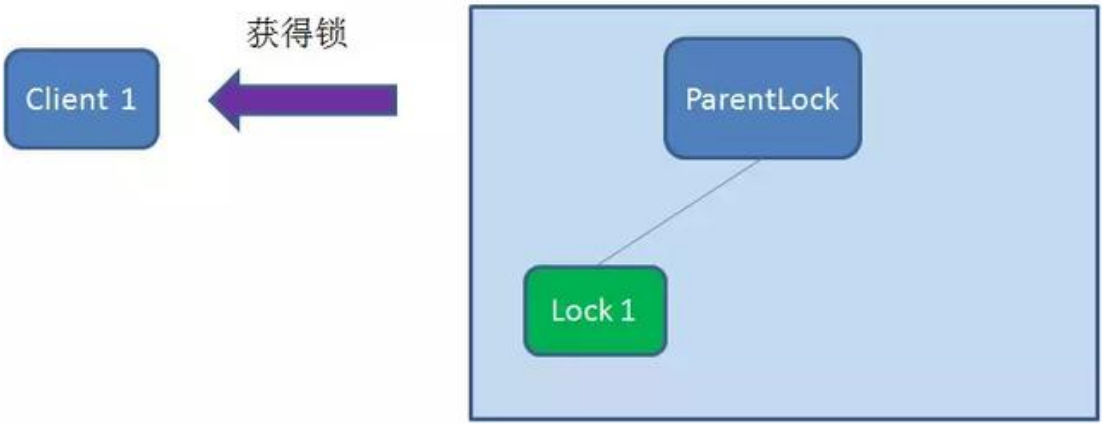
Zookeeper 分布式锁恰恰应用了临时顺序节点。具体如何实现呢？让我们来看一看详细步骤：

- **1、获取锁**

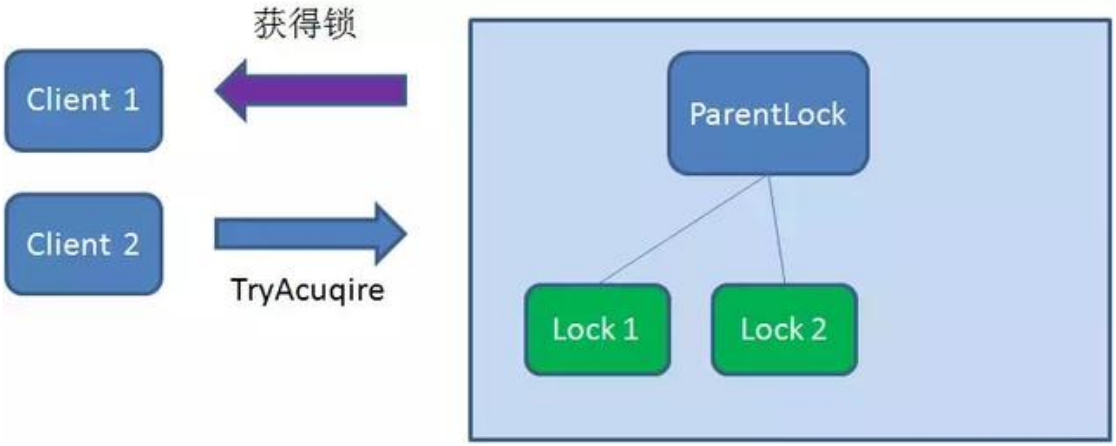
首先，在 Zookeeper 当中创建一个持久节点 ParentLock。当第一个客户端想要获得锁时，需要在 ParentLock 这个节点下面创建一个临时顺序节点 Lock1。



之后，Client1 查找 ParentLock 下面所有的临时顺序节点并排序，判断自己所创建的节点 Lock1 是不是顺序最靠前的一个。如果是第一个节点，则成功获得锁。

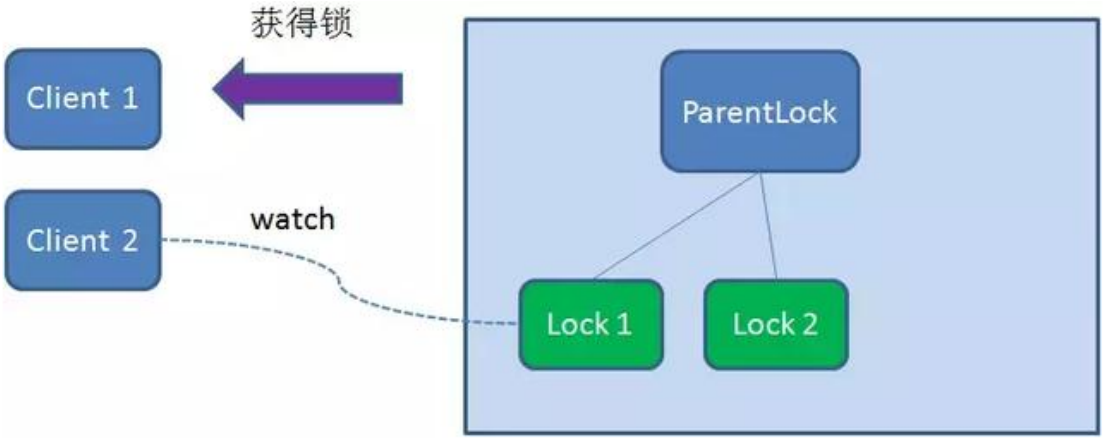


这时候，如果再有一个客户端 Client2 前来获取锁，则在 ParentLock 下载再创建一个临时顺序节点 Lock2。

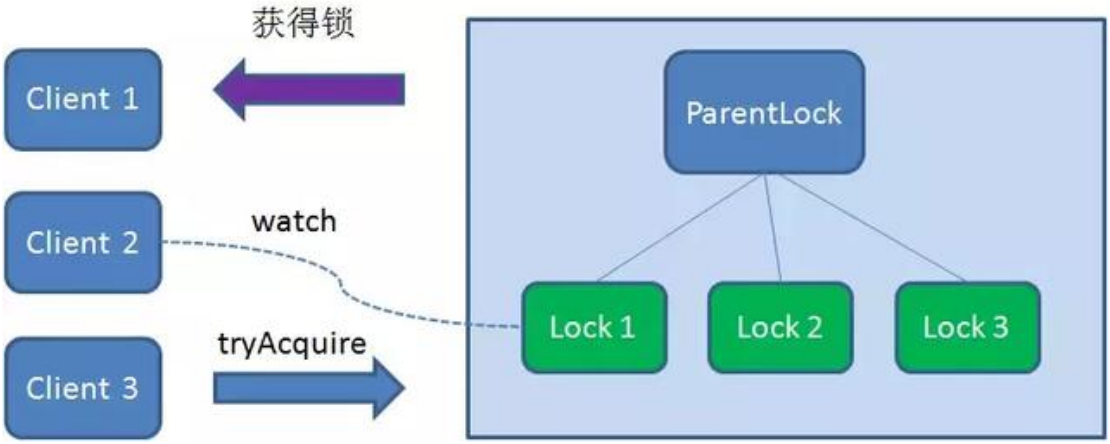


Client2 查找 ParentLock 下面所有的临时顺序节点并排序，判断自己所创建的节点 Lock2 是不是顺序最靠前的一个，结果发现节点 Lock2 并不是最小的。

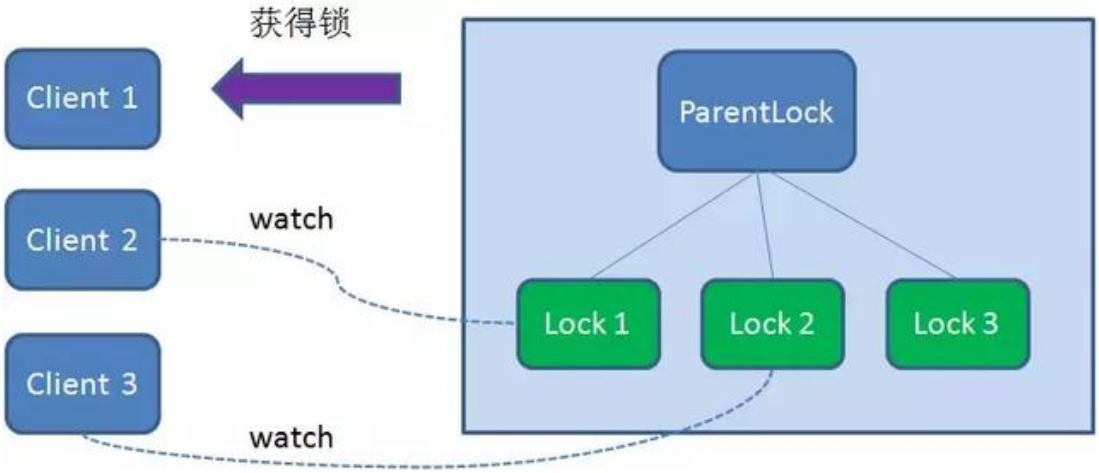
于是，**Client2 向排序仅比它靠前的节点 Lock1 注册 Watcher，用于监听 Lock1 节点是否存在。**这意味着 Client2 抢锁失败，进入了等待状态。



这时候，如果又有一个客户端Client3 前来获取锁，则在 ParentLock 下载再创建一个临时顺序节点 Lock3。



Client3 查找 ParentLock 下面所有的临时顺序节点并排序，判断自己所创建的节点 Lock3 是不是顺序最靠前的一个，结果同样发现节点 Lock3 并不是最小的。于是，Client3 向排序仅比它靠前的节点 Lock2 注册Watcher，用于监听 Lock2 节点是否存在。这意味着 Client3 同样抢锁失败，进入了等待状态。



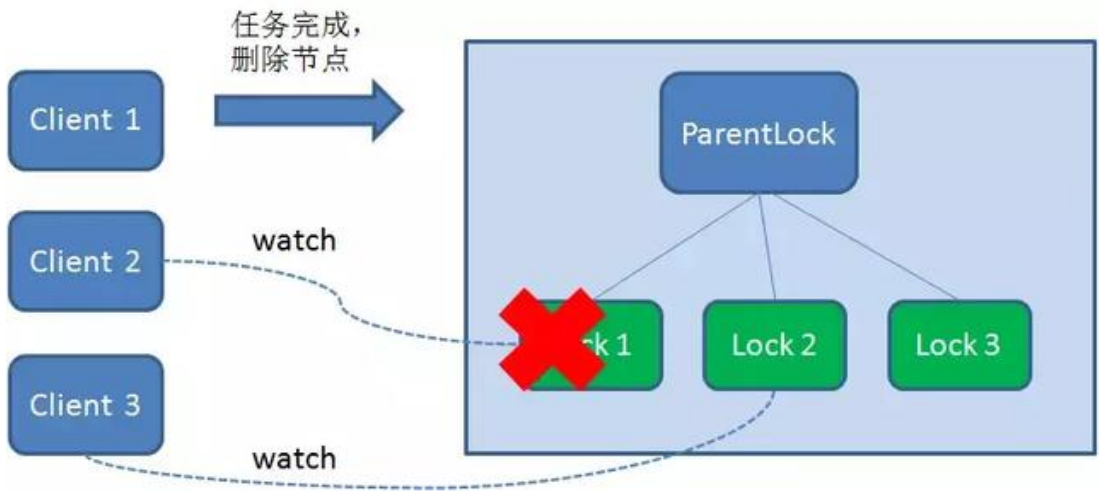
这样一来，Client1得到了锁，Client2 监听了 Lock1，Client3 监听了 Lock2。这恰恰形成了一个等待队列。

- **2、释放锁：**

释放锁分为两种情况：

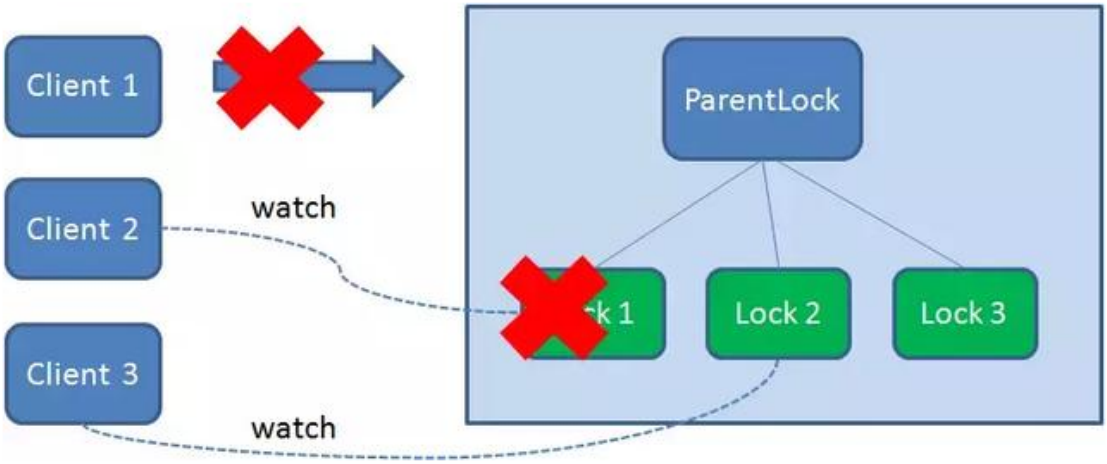
1、任务完成，客户端显示释放

当任务完成时，Client1 会显示调用删除节点 Lock1 的指令。

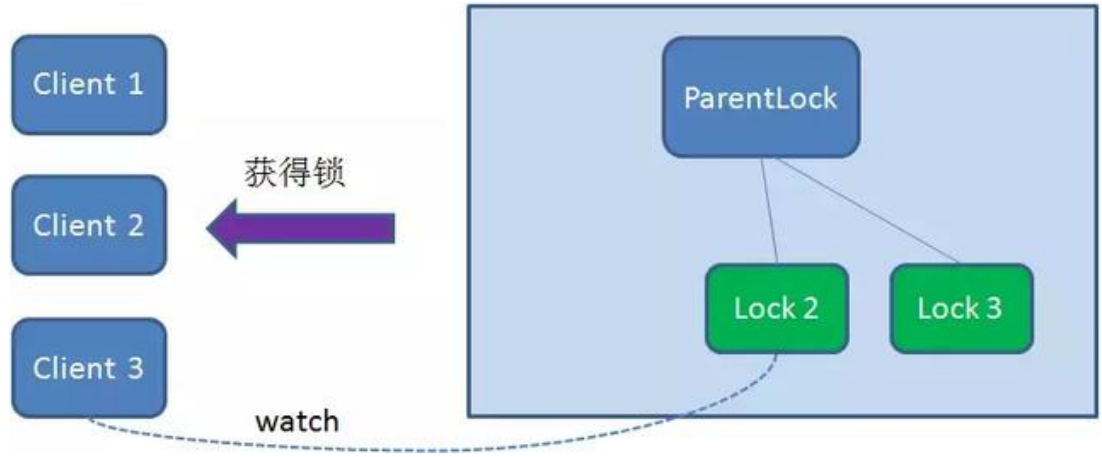


2、任务执行过程中，客户端崩溃

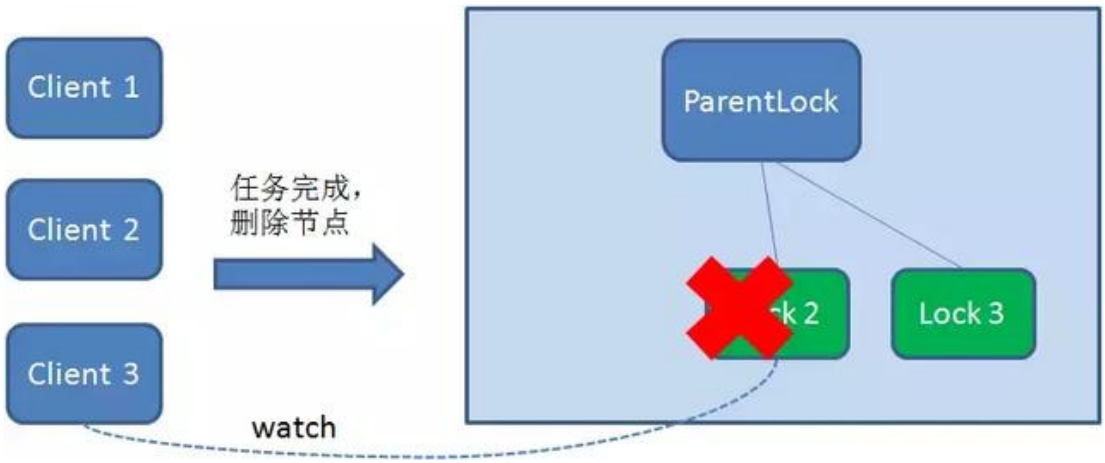
获得锁的 Client1 在任务执行过程中，如果崩溃了，则会断开与 Zookeeper 服务端的链接。**根据临时节点的特性，相关联的节点 Lock1 会随之自动删除。**



由于 Client2 一直监听着 Lock1 的存在状态，当 Lock1 节点被删除，Client2 会立刻收到通知。这时候 Client2 会再次查询 ParentLock 下面的所有节点，确认自己创建的节点 Lock2 是不是目前最小的节点。如果是最小，则 Client2 顺理成章获得了锁。



同理，如果 Client2 也因为任务完成或者节点崩溃而删除了节点 Lock2，那么 Client3 就会接到通知。



最终，Client3 成功得到了锁。

- **补充：**

可以直接使用 zookeeper 第三方库 Curator 客户端，这个客户端中封装了一个可重入的锁服务。

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException {
    try {
        return interProcessMutex.acquire(timeout, unit);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return true;
}
public boolean unlock() {
    try {
        interProcessMutex.release();
    } catch (Throwable e) {
        log.error(e.getMessage(), e);
    } finally {
        executorService.schedule(new Cleaner(client, path), delayTimeForClean, TimeUnit.MILLISECONDS);
    }
    return true;
}
```

Curator 提供的InterProcessMutex 是分布式锁的实现。acquire 方法用户获取锁，release 方法用于释放锁。

2.4 总结

下面的表格总结了 Zookeeper 和 Redis 分布式锁的优缺点：

分布式锁	优点	缺点
Zookeeper	1.有封装好的框架，容易实现 2.有等待锁的队列，大大提升抢锁效率。	添加和删除节点性能较低。
Redis	Set和Del指令的性能较高。	1.实现复杂，需要考虑超时、原子性、误删等情形。 2.没有等待锁的队列，只能在客户端自旋来等锁，效率低下。

三种方案的比较：

上面几种方式，哪种方式都无法做到完美。就像 CAP 一样，在复杂性、可靠性、性能等方面无法同时满足，所以，根据不同的应用场景选择最适合自己的才是王道。

从理解的难易程度角度（从低到高）：数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）：Zookeeper >= 缓存 > 数据库

从性能角度（从高到低）：缓存 > Zookeeper >= 数据库

从可靠性角度（从高到低）Zookeeper > 缓存 > 数据库