Web高级功能知识

1. **分布式锁**

分布式锁主要用于在分布式环境中保护跨进程、跨主机、跨网络的共享资源实现互斥访问，以达到保证数据的一致性。

一般是三种方式：数据库乐观锁、基于Redis的分布式锁、基于zookeeper的分布式锁。

**独占锁:**

又称排它锁，从字面意思上很容易理解他们的用途。即如果某个操作 O1 对访问资源 R1 的过程加锁，在操作 O1 结束对资源 R1 访问前，其他操作不允许访问资源 R1。

**Redis分布式锁：**

首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：

1. 互斥性。在任意时刻，只有一个客户端能持有锁。
2. 不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
3. 具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
4. 解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

pom.xml文件配置：

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.9.0</version>

</dependency>

RedisTool类：

import java.util.Collections;

import redis.clients.jedis.Jedis;

public class RedisTool {

    private static final String LOCK\_SUCCESS = "OK";

    private static final String SET\_IF\_NOT\_EXIST = "NX";

    private static final String SET\_WITH\_EXPIRE\_TIME = "PX";

    private static final Long RELEASE\_SUCCESS = 1L;

    /\*\*

     \* 尝试获取分布式锁

     \* @param jedis Redis客户端

     \* @param lockKey 锁

     \* @param requestId 请求标识

     \* @param expireTime 超期时间

     \* @return 是否获取成功

     \*/

    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {

        String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);

        if (LOCK\_SUCCESS.equals(result)) {

            return true;

        }

        return false;

    }

    /\*\*

     \* 释放分布式锁

     \* @param jedis Redis客户端

     \* @param lockKey 锁

     \* @param requestId 请求标识

     \* @return 是否释放成功

     \*/

    public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

        Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

        if (RELEASE\_SUCCESS.equals(result)) {

            return true;

        }

        return false;

    }

}

加锁：

可以看到，我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。

第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；

第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。

心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。





解锁：

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。

那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。那么为什么执行eval()方法可以确保原子性，源于Redis的特性。

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。

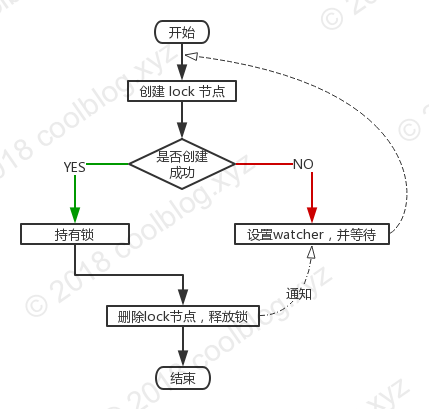




**zookeeper的分布式锁：**

对于独占锁，我们可以将资源 R1 看做是 lock 节点，操作 O1 访问资源 R1 看做创建 lock 节点，释放资源 R1 看做删除 lock 节点。这样我们就将独占锁的定义对应于具体的 Zookeeper 节点结构，通过创建 lock 节点获取锁，删除节点释放锁。详细的过程如下：

1. 多个客户端竞争创建 lock 临时节点
2. 其中某个客户端成功创建 lock 节点，其他客户端对 lock 节点设置 watcher
3. 持有锁的客户端删除 lock 节点或该客户端崩溃，由 Zookeeper 删除 lock 节点
4. 其他客户端获得 lock 节点被删除的通知
5. 重复上述4个步骤，直至无客户端在等待获取锁了



1. **防止订单重复提交**

造成订单重复提交的原因有哪些？

(1)转发引起的。

我们知道页面跳转有两种方式：转发和重定向。所谓转发是在服务器端进行跳转，对用户时透明的，这时，浏览器中的地址是不会改变的。所以，如果此时刷新页面，会再次对服务器请求一次，造成多次下单；

而重定向，是在客户端发生跳转，跳转时浏览器中的地址会发生改变。也就是说，我下单成功后，会马上重定向到一个新页面，浏览器地址栏改变，这是刷新，只会请求显示的地址，而不会请求上一次的地址。

(2)服务器响应慢引起的。

当用户点击下订单后，由于服务器反应时间过长没能及时看到响应信息，或者出于其它目的，多次点击“提 交”按钮，从而导致在服务器端接收到两条或多条相同的信息。如果信息需要存储到后台数据库中，如此以来就会产生数据库操作异常提示信息，以至于给用户带来 错误信息提示，从而给用户的使用带来不便。

**解决方式**：

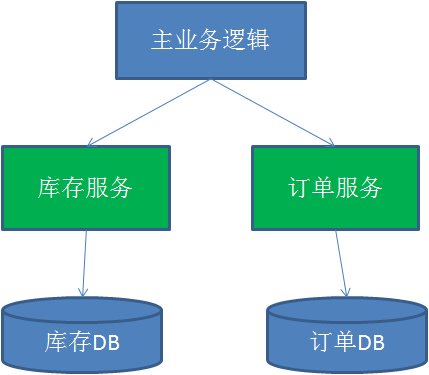
1. 下单成功后，重定向到其他页面，这种方式，只能解决刷新时，生成订单的可能，但是不能防止多次点击的情况。
2. 利用struts2的token拦截器。

**原理**：Struts 2已经内置了能够防止用户重复提交同一个HTML表单的功能。它的工作原理：让服务器生成一个唯一标记，并在服务器和表单里各保存一份这个标记的副本。此后，在用户提交表单的时候，表单里的标记将随着其他请求参数一起发送到服务器，服务器将对他收到的标记和它留存的标记进行比较。如果两者匹配，这次提交的表单被认为是有效的，服务器将对之做出必要的处理并重新设置一个新标记。随后，提交相同的表单就会失败，因为服务器上的标记已经重置。

恶意用户可利用这一性质，不断重复访问页面，以致Session中保存的标识号不断增多，最终严重消耗服务器内存。可以采用在Session中记录用户发帖的时间，然后通过一个时间间隔来限制用户连续发帖的数量来解决这一问题。

1. 客户端，也就是在前端控制唯一流水只能提交一次，服务端未返回不能重复提交。此种方法有个弊端就是万一绕过前端，订单仍然可以重复提交。
2. **分布式事务**

**概念**：分布式事务就是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同节点之上。简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。



上图中包含了库存和订单两个独立的微服务，每个微服务维护了自己的数据库。在交易系统的业务逻辑中，一个商品在下单之前需要先调用库存服务，进行扣除库存，再调用订单服务，创建订单记录。正常情况下，两个数据库各自更新成功，两边数据维持着一致性。但是，在非正常情况下，有可能库存的扣减完成了，随后的订单记录却因为某些原因插入失败。这个时候，两边数据就失去了应有的一致性。

如何实现分布式事务？

XA分布式事务协议、MQ事务、TCC事务

**XA协议：**

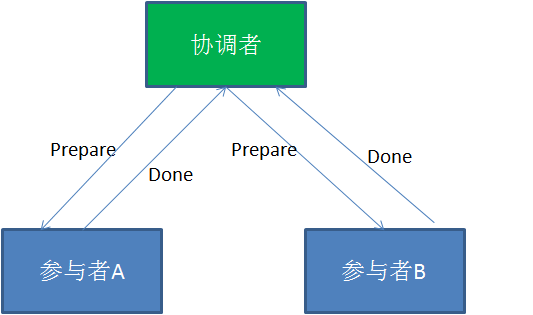
一般常用的数据库，比如Oracle、MySql、Microsoft SQL Server、DB2等都支持XA标准，XA(eXtended Architecture)标准是开发组织提出的分布式事务处理方案，定义了事务管理器与资源管理器之间通信的接口协议；XA协议由数据库实现。

XA的两阶段提交（2PC）和三阶段提交（3PC）

**两阶段提交：**

在XA协议中包含着两个角色：**事务协调者**和**事务参与者**。让我们来看一看他们之间的交互流程：

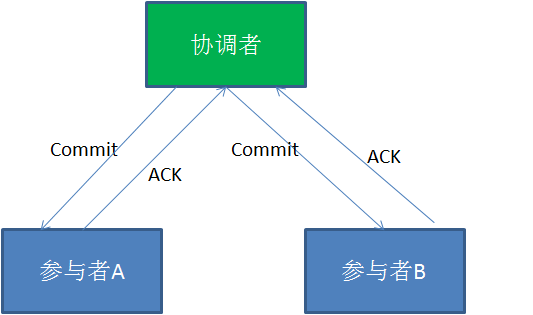
第一阶段：



在XA分布式事务的第一阶段，作为事务协调者的节点会首先向所有的参与者节点发送Prepare请求。在接到Prepare请求之后，每一个参与者节点会各自执行与事务有关的数据更新，写入Undo Log和Redo Log。如果参与者执行成功，暂时不提交事务，而是向事务协调节点返回“完成”消息。

当事务协调者接到了所有参与者的返回消息，整个分布式事务将会进入第二阶段。

第二阶段：



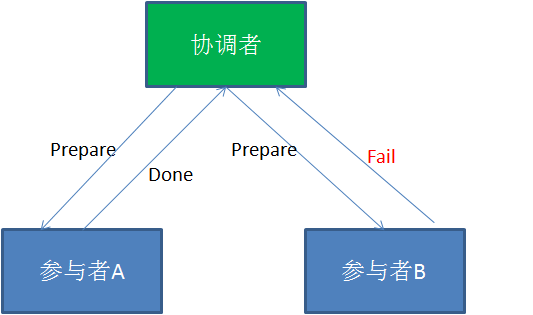
在XA分布式事务的第二阶段，如果事务协调节点在之前所收到都是正向返回，那么它将会向所有事务参与者发出Commit请求。

接到Commit请求之后，事务参与者节点会各自进行本地的事务提交，并释放锁资源。当本地事务完成提交后，将会向事务协调者返回“完成”消息。

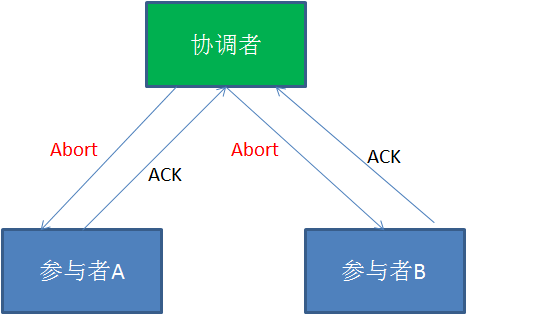
当事务协调者接收到所有事务参与者的“完成”反馈，整个分布式事务完成。

以上所描述的是XA两阶段提交的正向流程，接下来我们看一看失败情况的处理流程：

第一阶段：



第二阶段：



在XA的第一阶段，如果某个事务参与者反馈失败消息，说明该节点的本地事务执行不成功，必须回滚。

于是在第二阶段，事务协调节点向所有的事务参与者发送Abort请求。接收到Abort请求之后，各个事务参与者节点需要在本地进行事务的回滚操作，回滚操作依照Undo Log来进行。

**缺点**：

1.性能问题

XA协议遵循强一致性。在事务执行过程中，各个节点占用着数据库资源，只有当所有节点准备完毕，事务协调者才会通知提交，参与者提交后释放资源。这样的过程有着非常明显的性能问题。

2.协调者单点故障问题

事务协调者是整个XA模型的核心，一旦事务协调者节点挂掉，参与者收不到提交或是回滚通知，参与者会一直处于中间状态无法完成事务。

3.丢失消息导致的不一致问题。

在XA协议的第二个阶段，如果发生局部网络问题，一部分事务参与者收到了提交消息，另一部分事务参与者没收到提交消息，那么就导致了节点之间数据的不一致。

如果避免XA两阶段提交的种种问题呢？有许多其他的分布式事务方案可供选择：

1.XA三阶段提交

XA三阶段提交在两阶段提交的基础上增加了CanCommit阶段，并且引入了超时机制。一旦事物参与者迟迟没有接到协调者的commit请求，会自动进行本地commit。这样有效解决了协调者单点故障的问题。但是性能问题和不一致的问题仍然没有根本解决。

2.MQ事务

利用消息中间件来异步完成事务的后一半更新，实现系统的最终一致性。这个方式避免了像XA协议那样的性能问题。

3.TCC事务

TCC事务是Try、Commit、Cancel三种指令的缩写，其逻辑模式类似于XA两阶段提交，但是实现方式是在代码层面来人为实现。

**三阶段提交：**

针对两阶段提交存在的问题，三阶段提交协议通过引入一个“预询盘”阶段，以及超时策略来减少整个集群的阻塞时间，提升系统性能。三阶段提交的三个阶段分别为：can\_commit，pre\_commit，do\_commit。

**第一阶段：can\_commit**

该阶段协调者会去询问各个参与者是否能够正常执行事务，参与者根据自身情况回复一个预估值，相对于真正的执行事务，这个过程是轻量的，具体步骤如下：

1. 协调者向各个参与者发送事务询问通知，询问是否可以执行事务操作，并等待回复。

2. 各个参与者依据自身状况回复一个预估值，如果预估自己能够正常执行事务就返回确定信息，并进入预备状态，否则返回否定信息。

**第二阶段：pre\_commit**

本阶段协调者会根据第一阶段的询盘结果采取相应操作，询盘结果主要有三种：

1. 所有的参与者都返回确定信息。

2. 一个或多个参与者返回否定信息。

3. 协调者等待超时。

针对第一种情况，协调者会向所有参与者发送事务执行请求，具体步骤如下：

1. 协调者向所有的事务参与者发送事务执行通知。

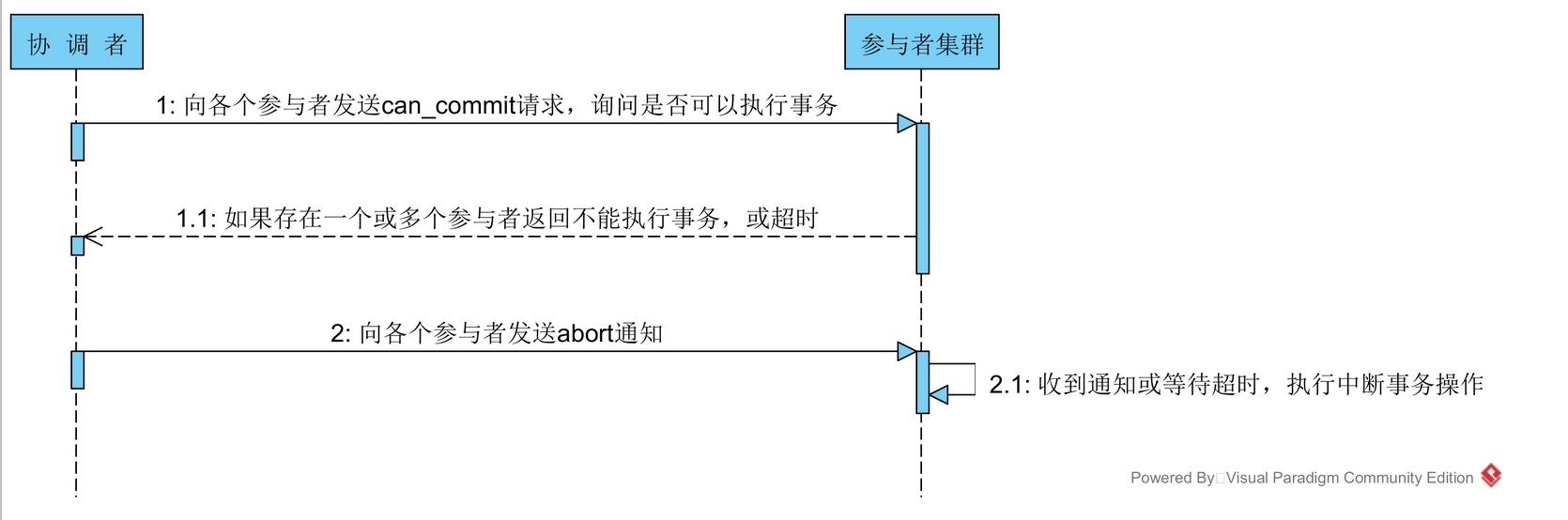
2. 参与者收到通知后，执行事务，但不提交。

3. 参与者将事务执行情况返回给客户端。

在上面的步骤中，如果参与者等待超时，则会中断事务。 针对第二、三种情况，协调者认为事务无法正常执行，于是向各个参与者发出abort通知，请求退出预备状态，具体步骤如下：

1. 协调者向所有事务参与者发送abort通知

2. 参与者收到通知后，中断事务



**第三阶段：do\_commit**

如果第二阶段事务未中断，那么本阶段协调者将会依据事务执行返回的结果来决定提交或回滚事务，分为三种情况：

1. 所有的参与者都能正常执行事务。

2. 一个或多个参与者执行事务失败。

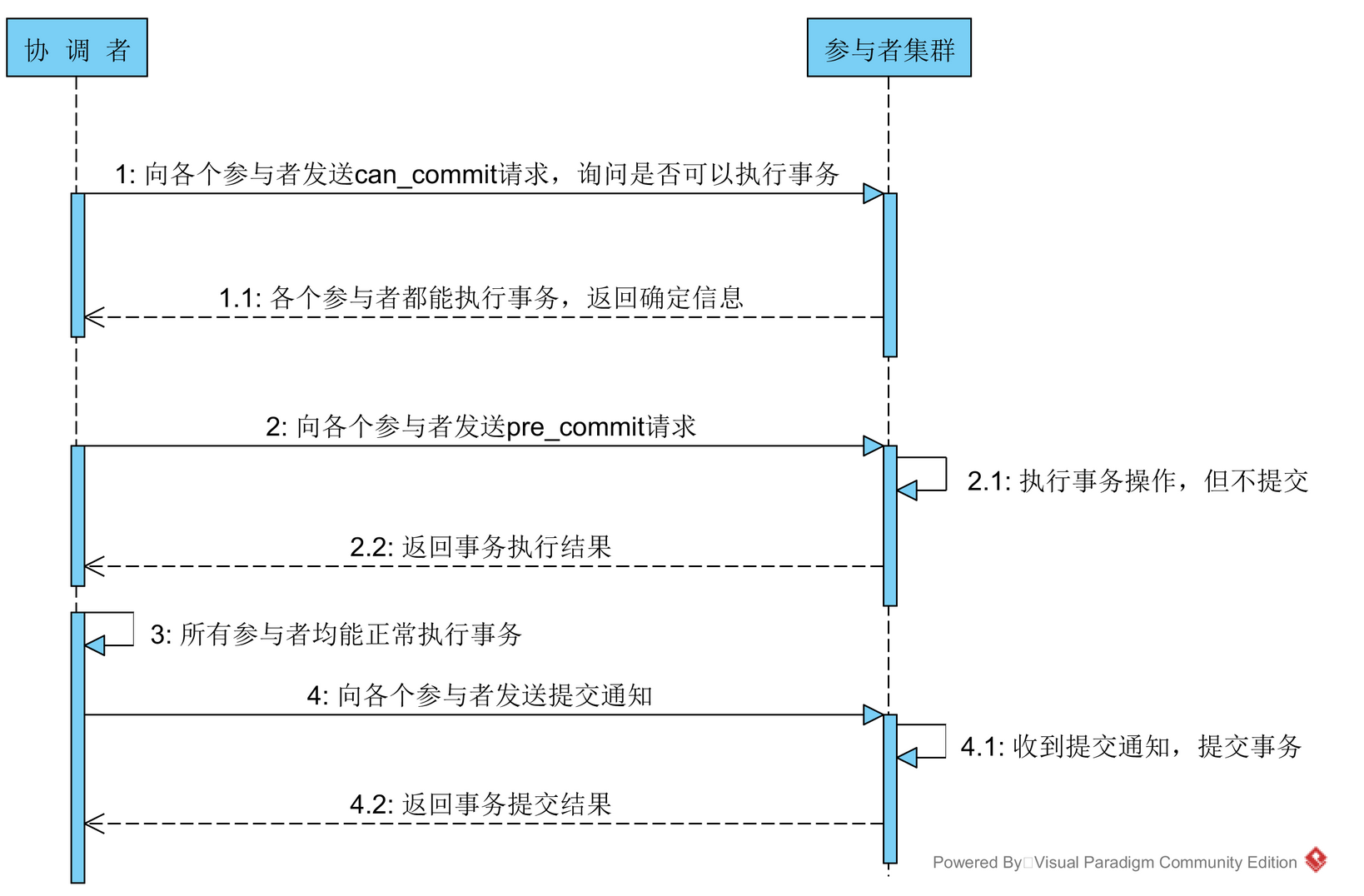
3. 协调者等待超时。

针对第一种情况，协调者向各个参与者发起事务提交请求，具体步骤如下：

1. 协调者向所有参与者发送事务commit通知。

2. 所有参与者在收到通知之后执行commit操作，并释放占有的资源。

3. 参与者向协调者反馈事务提交结果。

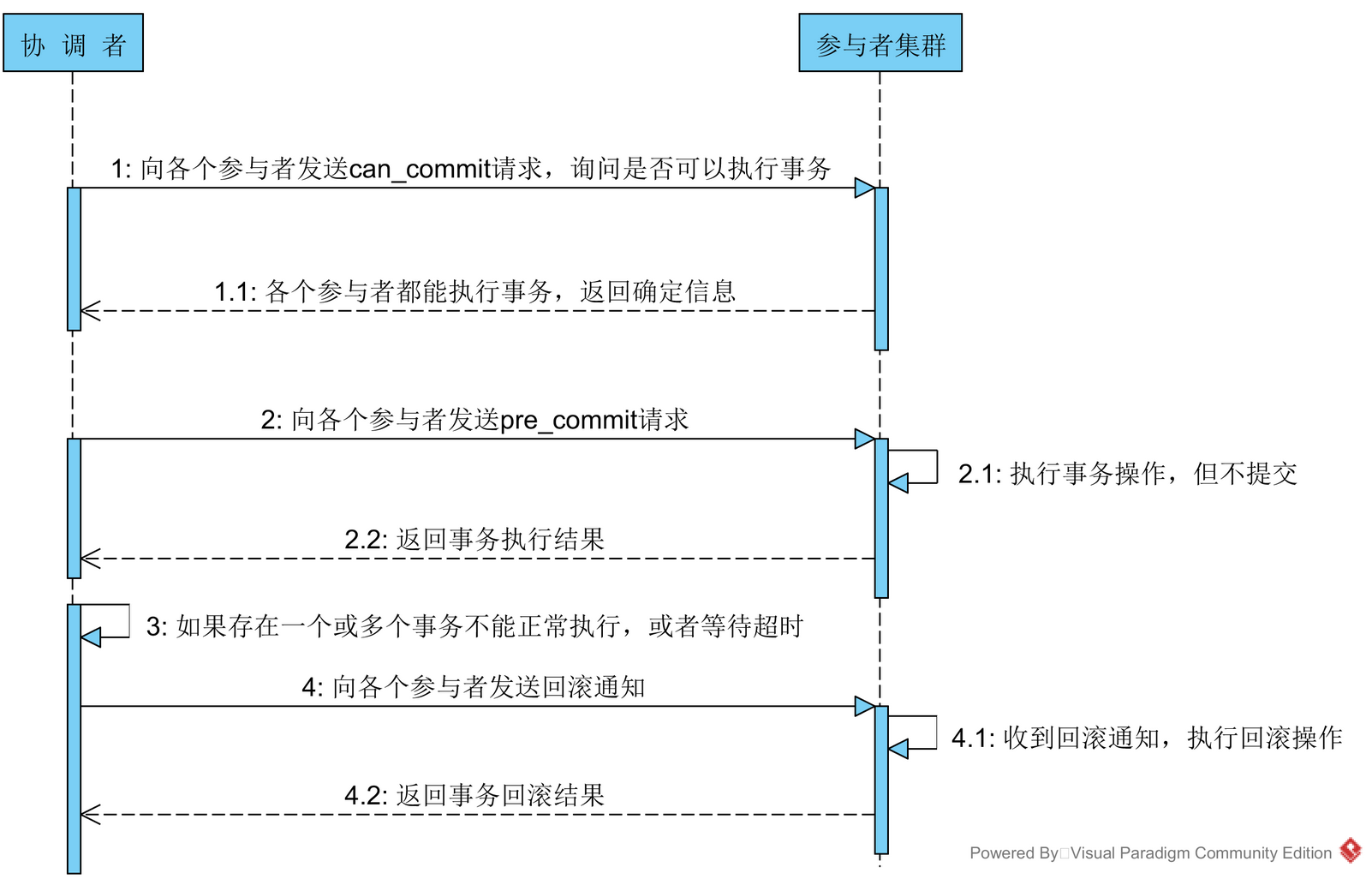


针对第二、三种情况，协调者认为事务无法正常执行，于是向各个参与者发送事务回滚请求，具体步骤如下：

1. 协调者向所有参与者发送事务rollback通知。

2. 所有参与者在收到通知之后执行rollback操作，并释放占有的资源。

3. 参与者向协调者反馈事务提交结果。



在本阶段如果因为协调者或网络问题，导致参与者迟迟不能收到来自协调者的commit或rollback请求，那么参与者将不会如两阶段提交中那样陷入阻塞，而是等待超时后继续commit。相对于两阶段提交虽然降低了同步阻塞，但仍然无法避免数据的不一致性。

**MQ事务：**

如果仔细观察生活的话，生活的很多场景已经给了我们提示。

比如在北京很有名的姚记炒肝点了炒肝并付了钱后，他们并不会直接把你点的炒肝给你，而是给你一张小票，然后让你拿着小票到出货区排队去取。为什么他们要将付钱和取货两个动作分开呢？原因很多，其中一个很重要的原因是为了使他们接待能力增强（并发量更高）。

还是回到我们的问题，只要这张小票在，你最终是能拿到炒肝的。同理转账服务也是如此，当支付宝账户扣除1万后，我们只要生成一个凭证（消息）即可，这个凭证（消息）上写着“让余额宝账户增加 1万”，只要这个凭证（消息）能可靠保存，我们最终是可以拿着这个凭证（消息）让余额宝账户增加1万的，即我们能依靠这个凭证（消息）完成最终一致性。

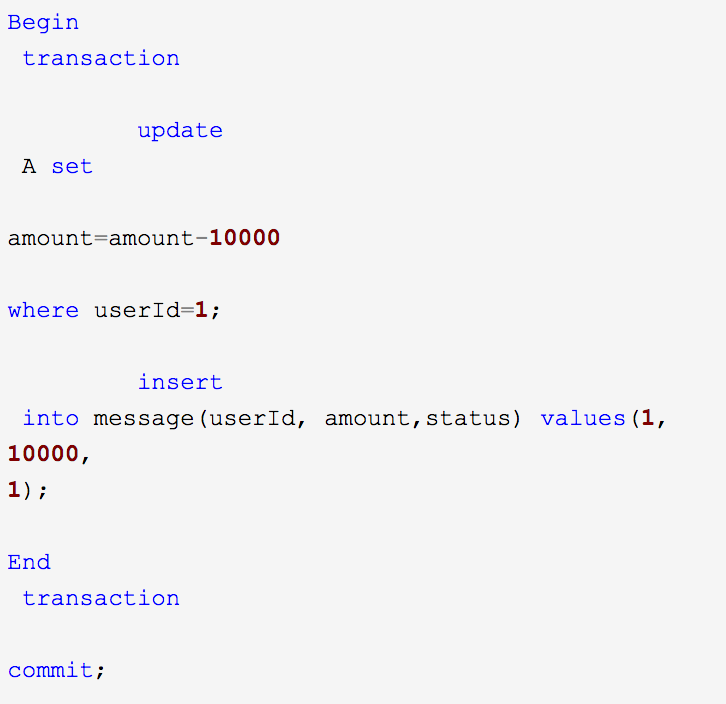
如何可靠保存凭证（消息）？

有两种方法：

**消息表的方式：**

这种方式业务与消息耦合

支付宝在完成扣款的同时，同时记录消息数据，这个消息数据与业务数据保存在同一数据库实例里（消息记录表表名为message）。



上述事务能保证只要支付宝账户里被扣了钱，消息一定能保存下来。

当上述事务提交成功后，我们通过实时消息服务将此消息通知余额宝，余额宝处理成功后发送回复成功消息，支付宝收到回复后删除该条消息数据。

**基于消息方式：**

业务与消息解耦方式。

上述保存消息的方式使得消息数据和业务数据紧耦合在一起，从架构上看不够优雅，而且容易诱发其他问题。为了解耦，可以采用以下方式。

1）支付宝在扣款事务提交之前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送，只有提交事务成功后才会发送消息；

2）当支付宝扣款事务被提交成功后，向实时消息服务确认发送。只有在得到确认发送指令后，实时消息服务才真正发送该消息；

3）当支付宝扣款事务提交失败回滚后，向实时消息服务取消发送。在得到取消发送指令后，该消息将不会被发送；

4）对于那些未确认的消息或者取消的消息，需要有一个消息状态确认系统定时去支付宝系统查询这个消息的状态并进行更新。为什么需要这一步骤，举个例子：假设在第2步支付宝扣款事务被成功提交后，系统挂了，此时消息状态并未被更新为“确认发送”，从而导致消息不能被发送。

优点：消息数据独立存储，降低业务系统与消息系统间的耦合；

缺点：一次消息发送需要两次请求；业务处理服务需要实现消息状态回查接口。

**如何解决消息重复投递的问题？**

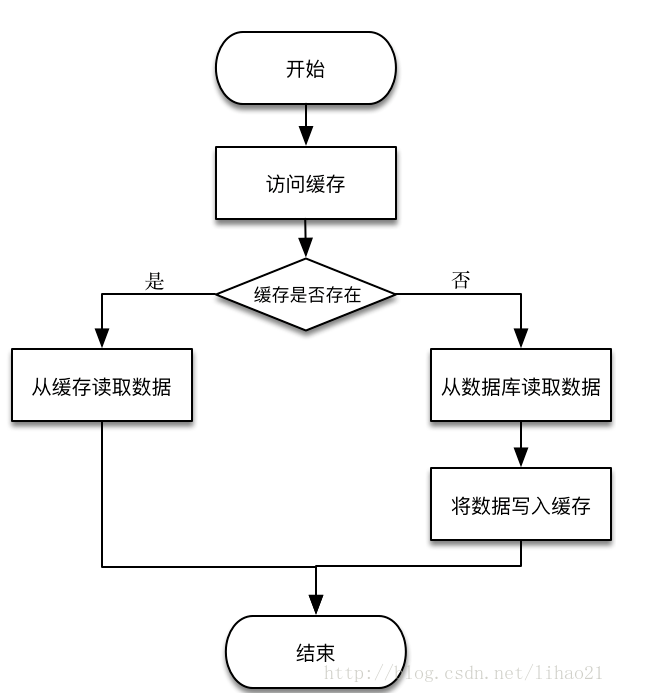
还有一个很严重的问题就是消息重复投递，以我们支付宝转账到余额宝为例，如果相同的消息被重复投递两次，那么我们余额宝账户将会增加2万而不是1万了。

为什么相同的消息会被重复投递？比如余额宝处理完消息msg后，发送了处理成功的消息给支付宝，正常情况下支付宝应该要删除消息msg，但如果支付宝这时候悲剧的挂了，重启后一看消息msg还在，就会继续发送消息msg。

解决方法很简单，在余额宝这边增加消息应用状态表（message\_apply），通俗来说就是个账本，用于记录消息的消费情况，每次来一个消息，在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表（同一事务）。

1. **一致性Hash（Consistent Hashing）**

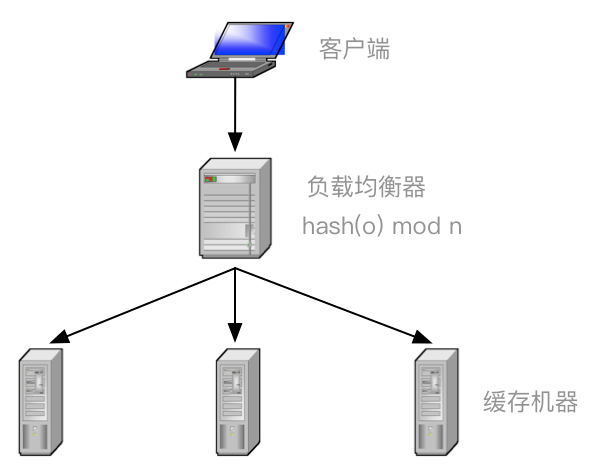
在业务开发中，我们常把数据持久化到数据库中。如果需要读取这些数据，除了直接从数据库中读取外，为了减轻数据库的访问压力以及提高访问速度，我们更多地引入缓存来对数据进行存取。读取数据的过程一般为：



对于分布式缓存，不同机器上存储不同对象的数据。为了实现这些缓存机器的负载均衡，可以使用式子1来定位对象缓存的存储机器：

m = hash(o) mod n ——式子1

其中，o为对象的名称，n为机器的数量，m为机器的编号，hash为一hash函数。图2中的负载均衡器（load balancer）正是使用式子1来将客户端对不同对象的请求分派到不同的机器上执行，例如，对于对象o，经过式子1的计算，得到m的值为3，那么所有对对象o的读取和存储的请求都被发往机器3执行。



式子1在大部分时候都可以工作得很好，然而，当机器需要扩容或者机器出现宕机的情况下，事情就比较棘手了。当机器扩容，需要增加一台缓存机器时，负载均衡器使用的式子变成：

m = hash(o) mod (n + 1) ——式子2

当机器宕机，机器数量减少一台时，负载均衡器使用的式子变成：

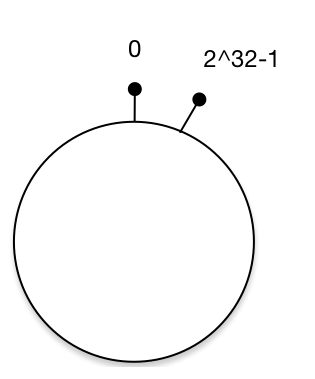
m = hash(o) mod (n - 1) ——式子3

我们以机器扩容的情况为例，说明简单的取模方法会导致什么问题。假设机器由3台变成4台，对象o1由式子1计算得到的m值为2，由式子2计算得到的m值却可能为0，1，2，3（一个 3t + 2的整数对4取模，其值可能为0，1，2，3，读者可以自行验证），大约有75%（3/4)的可能性出现缓存访问不命中的现象。随着机器集群规模的扩大，这个比例线性上升。当99台机器再加入1台机器时，不命中的概率是99%（99/100）。这样的结果显然是不能接受的，因为这会导致数据库访问的压力陡增，严重情况，还可能导致数据库宕机。

一致性hash算法正是为了解决此类问题的方法，它可以保证当机器增加或者减少时，对缓存访问命中的概率影响减至很小。下面我们来详细说一下一致性hash算法的具体过程。

**一致性Hash环:**

一致性hash算法通过一个叫作一致性hash环的数据结构实现。这个环的起点是0，终点是2^32 - 1，并且起点与终点连接，环的中间的整数按逆时针分布，故这个环的整数分布范围是[0, 2^32-1]，如下图3所示：



将对象放置到Hash环

假设现在我们有4个对象，分别为o1，o2，o3，o4，使用hash函数计算这4个对象的hash值（范围为0 ~ 2^32-1）:

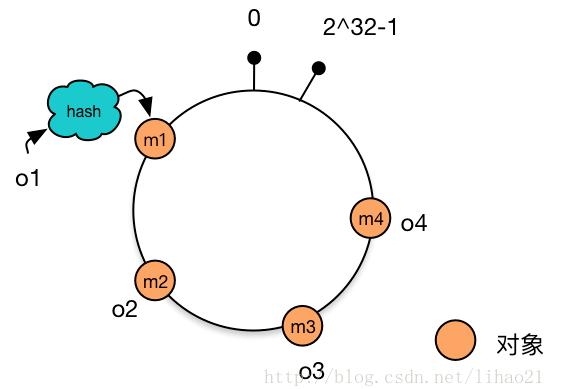
hash(o1) = m1

hash(o2) = m2

hash(o3) = m3

hash(o4) = m4

把m1，m2，m3，m4这4个值放置到hash环上，得到如下图4：



将机器放置到Hash环

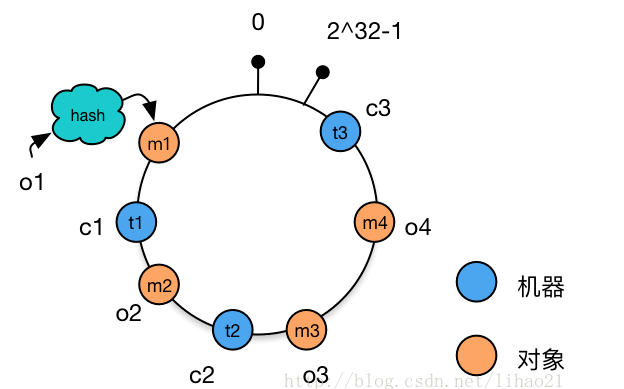
使用同样的hash函数，我们将机器也放置到hash环上。假设我们有三台缓存机器，分别为 c1，c2，c3，使用hash函数计算这3台机器的hash值：

hash(c1) = t1

hash(c2) = t2

hash(c3) = t3

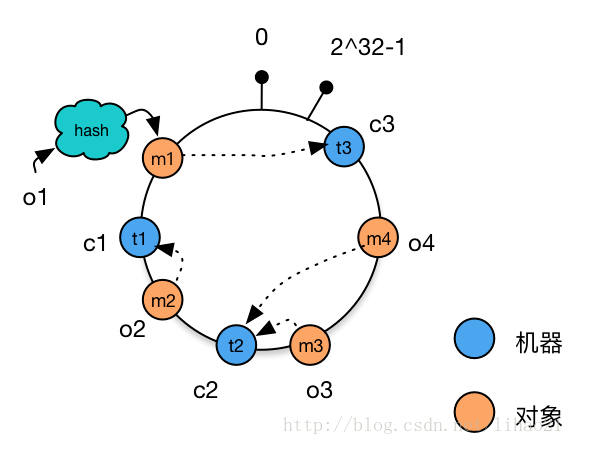
把t1，t2，t3 这3个值放置到hash环上，得到如下图5：



为对象选择机器

将对象和机器都放置到同一个hash环后，在hash环上顺时针查找距离这个对象的hash值最近的机器，即是这个对象所属的机器。

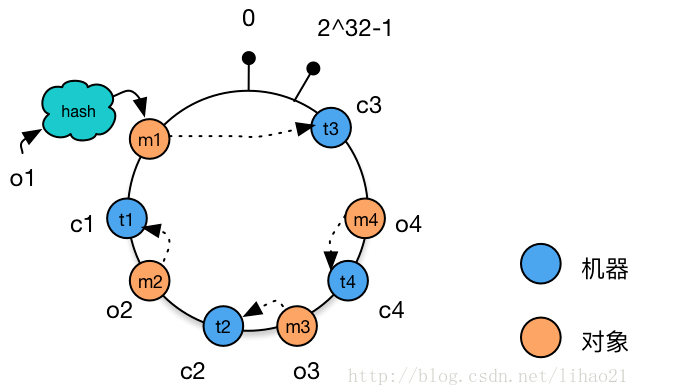
例如，对于对象o2，顺序针找到最近的机器是c1，故机器c1会缓存对象o2。而机器c2则缓存o3，o4，机器c3则缓存对象o1。



处理机器增减的情况

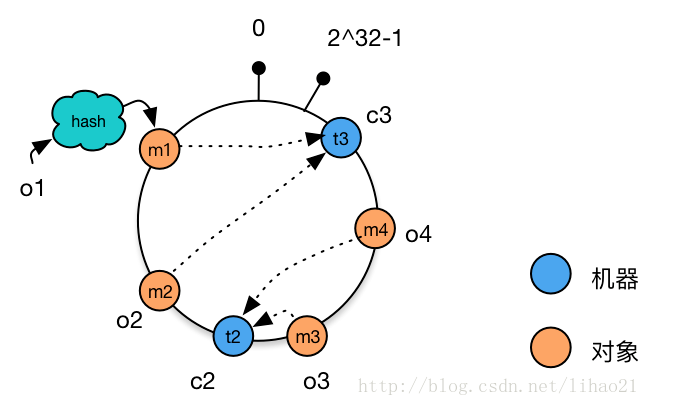
对于线上的业务，增加或者减少一台机器的部署是常有的事情。

例如，增加机器c4的部署并将机器c4加入到hash环的机器c3与c2之间。这时，只有机器c3与c4之间的对象需要重新分配新的机器。对于我们的例子，只有对象o4被重新分配到了c4，其他对象仍在原有机器上。如图7所示：



如上文前面所述，使用简单的求模方法，当新添加机器后会导致大部分缓存失效的情况，使用一致性hash算法后这种情况则会得到大大的改善。前面提到3台机器变成4台机器后，缓存命中率只有25%（不命中率75%）。而使用一致性hash算法，理想情况下缓存命中率则有75%，而且，随着机器规模的增加，命中率会进一步提高，99台机器增加一台后，命中率达到99%，这大大减轻了增加缓存机器带来的数据库访问的压力。

再例如，将机器c1下线（当然，也有可能是机器c1宕机），这时，只有原有被分配到机器c1对象需要被重新分配到新的机器。对于我们的例子，只有对象o2被重新分配到机器c3，其他对象仍在原有机器上。如图8所示：



**虚拟节点:**

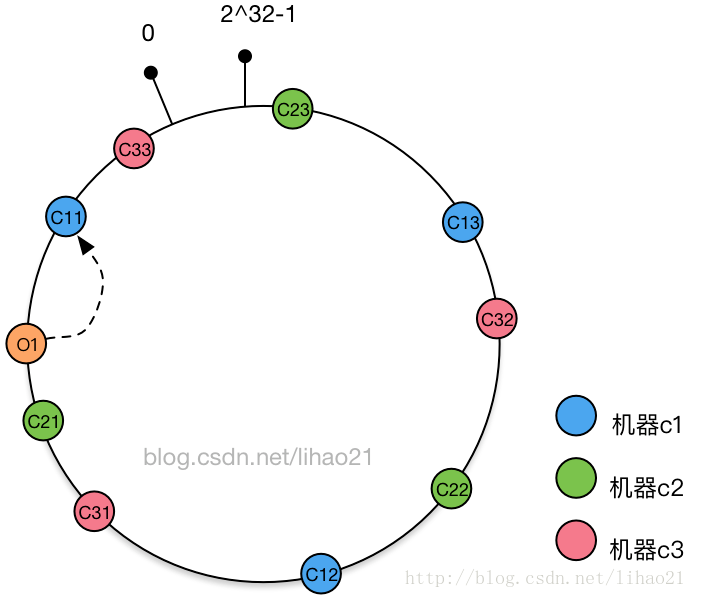
上面提到的过程基本上就是一致性hash的基本原理了，不过还有一个小小的问题。新加入的机器c4只分担了机器c2的负载，机器c1与c3的负载并没有因为机器c4的加入而减少负载压力。如果4台机器的性能是一样的，那么这种结果并不是我们想要的。

为此，我们引入虚拟节点来解决负载不均衡的问题。

将每台物理机器虚拟为一组虚拟机器，将虚拟机器放置到hash环上，如果需要确定对象的机器，先确定对象的虚拟机器，再由虚拟机器确定物理机器。

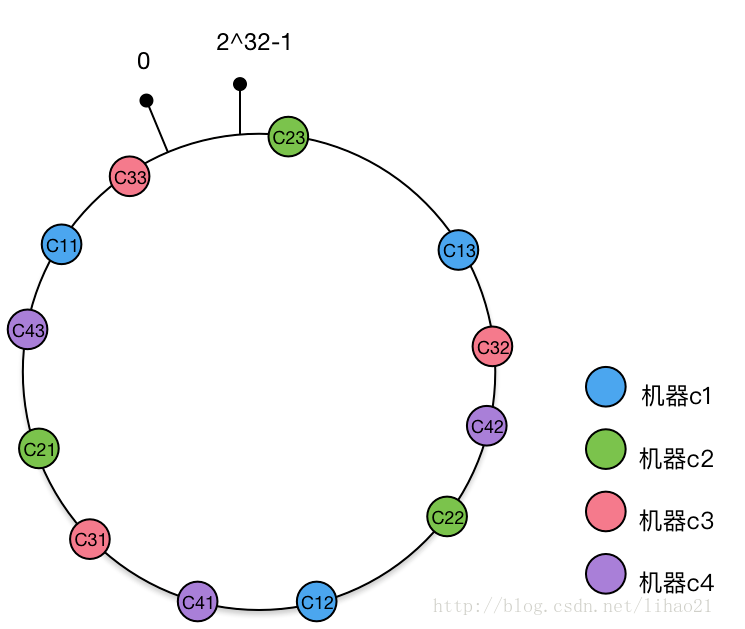
说得有点复杂，其实过程也很简单。

还是使用上面的例子，假如开始时存在缓存机器c1，c2，c3，对于每个缓存机器，都有3个虚拟节点对应，其一致性hash环结构如图9所示：



假设对于对象o1，其对应的虚拟节点为c11，而虚拟节点c11对象缓存机器c1，故对象o1被分配到机器c1中。

新加入缓存机器c4，其对应的虚拟节点为c41，c42，c43，将这三个虚拟节点添加到hash环中，得到的hash环结构如图10所示：



新加入的缓存机器c4对应一组虚拟节点c41，c42，c43，加入到hash环后，影响的虚拟节点包括c31，c22，c11（顺时针查找到第一个节点），而这3个虚拟节点分别对应机器c3，c2，c1。即新加入的一台机器，同时影响到原有的3台机器。理想情况下，新加入的机器平等地分担了原有机器的负载，这正是虚拟节点带来的好处。而且新加入机器c4后，只影响25%（1/4）对象分配，也就是说，命中率仍然有75%，这跟没有使用虚拟节点的一致性hash算法得到的结果是相同的。

**总结：**

一致性hash算法解决了分布式环境下机器增加或者减少时，简单的取模运算无法获取较高命中率的问题。通过虚拟节点的使用，一致性hash算法可以均匀分担机器的负载，使得这一算法更具现实的意义。正因如此，一致性hash算法被广泛应用于分布式系统中。

1. **Aa**