zookeeper学习总结

作者：guobing
原文链接：https://ld246.com/article/1470808874184
来源网站：链滴

<h3 id="1-zookeeper的作用" data-anchor-id="vmfs">1. zookeeper的作用</h3>
<blockquote class="white-blockquote" data-anchor-id="oozb">

配置中心
分布式锁（和我遇到的分布式锁的区别）
统一命名服务。和JNDI类似
分布式系统的故障修复。由master监控集群中服务器状态。当有服务器挂掉时通知其他服务器重分配不同节点的计算任务。master挂掉会新一轮重新选举master。

</blockquote>

<h3 id="2-特点" data-anchor-id="aoqg">2. 特点</h3>
<blockquote class="white-blockquote" data-anchor-id="92uw">

精简的文件系统。管理小文件。hadoop是大型文件系统。
采用观察者设计模式

</blockquote>

<h3 id="3-分布式一致性协议" data-anchor-id="q54t">3. 分布式一致性协议</h3>
<blockquote class="white-blockquote" data-anchor-id="5535">

二阶段提交
第一阶段:
协调者会问所有的参与者点，是否可以执行提交操作。
各个参与者开始事务执行的准备工作：如：为资源上锁预留资源，写undo/redo log……
参与者响应协调者，如果事务的准备工成功，则回应“可以提交”，否则回应“拒绝提交”。
<sp n>第二阶段:
如果所有的参与者都回应“可以提交”，那么，调者向所有的参与者发送“正式提交”的命令。参与者完成正式提交，并释放所有资源然后回应“完成”，协调者收集各结点的“完成”回应后结束这个Global Tansaction。
如果有一个参与者回应“拒绝提交”，那么，协调者向所有参与者发送“回滚操作”，并释放所有资源，然后回应“回滚完成”，协者收集各结点的“回滚”回应后，取消这个Global Transaction。

</blockquote>
1）如果第一阶段中，参与者没有收到询问请求，或是参与者的回应没到达协调者。那么，需要协调者做超时处理，一旦超时，可以当作失败，也可以重试。
2）如果第二阶段中，正式提交发出后，如果有的参与者没有收到，是参与者提交/回滚后的确认信息没有返回，一旦参与者的回应超时，要么重试，要么把那个参与者记为问题结点剔除整个集群，这样可以保证服务结点都是数据一致性的。
3）糟糕的情况是，第二阶段中，如果参与者收不到协调者的commit/fllback指令，参与者将处于“状态未知”阶段，参与者完全不知道要怎么办，比如：如所有的参与者完成第一阶段的回复后（可能全部yes，可能全部no，可能部分yes部分no），如果协者在这个时候挂掉了。那么所有的结点完全不知道怎么办（问别的参与者都不行）。为了一致性，要死等协调者，要么重发第一阶段的yes/no命令。
两段提交最大的问题就是第3）项，如果第一阶段完成后，参与者在第阶没有收到决策，那么数据结点会进入“不知所措”的状态，这个状态会block住整个务。也就是说，协调者Coordinator对于事务的完成非常重要，Coordinator的可用性是个关键。 因，我们引入三段提交，三段提交在Wikipedia上的描述如下，他把二段提交的第一个段break成了两：询问，然后再锁资源。最后真正提交。
三段提交的核心理念是：在询问的时候并不锁定资源，除非所人都同意了，才开始锁资源。
分布式一致性协议是由两个将军问题基础上提出来的
两将军问题的具体描述为：
1）第一位将军先发送一段消息“让我们在上午9点开进攻”。然而，一旦信使被派遣，他是否通过了山谷，第一位将军就不得而知了。任何一点的

确定性都会使得第一位将军攻击犹豫，因为如果第二位将军不能在同一时刻发动攻击，那座城市的驻就会击退他的军队的进攻，导致他的军对被摧毁。
2）知道了这一点，第二位将军就需要发送一个确认回条：“我到您的邮件，并会在9点的攻击。”但是，如果带着确认消息的信使被抓怎么办？所以第二位军会犹豫自己的确认消息是否能到达。
3）于是，似乎我们还要让第一位将军再发送一条确认消息—‐“我收到了你的确认”。然而，如果这位信使被抓怎么办呢？
4）这样一来，是不是我们还要第二位将军发送一个“确认收到的确认”的信息。
靠，于是你会发现，这事情很快就发展成为不管发送多少个确认消息，没有办法来保证两位将军有足够的自信自己的信使没有被敌军捕获。
两个将军问题和的无解证明首先由E.A.Akkoyunlu,K.Ekanadham和R.V.Huber于1975年在《一些限制与折衷的网络信设计》一文中发表，就在这篇文章的第73页中一段描述两个黑帮之间的通信中被阐明。 1978年，在im Gray的《数据库操作系统注意事项》一书中（从第465页开始）被命名为两个将军悖论。作为两将军问题的定义和无解性的证明的来源，这一参考被广泛提及。
这个实验意在阐明：试图通过建立在一个不可靠的连接上的交流来协一项行动的隐患和设计上的巨大挑战。
从工程上来说，一个解决两个将军问题的实际方法是使用一个能够承通信信道不可靠性的方案，并不试图去消除这个不可靠性，但要将不可靠性削减到一个可以接受的程。比如，第一位将军排出了100位信使并预计他们都被捕的可能性很小。在这种情况下，不管第二位军是否会攻击或者受到任何消息，第一位将军都会进行攻击。另外，第一位将军可以发送一个消息流而第二位将军可以对其中的每一条消息发送一个确认消息，这样如果每条消息都被接收到，两位将军感觉更好。然而我们可以从证明中看出，他们俩都不能肯定这个攻击是可以协调的。他们没有算法可（比如，收到4条以上的消息就攻击）能够确保防止仅有一方攻击。再者，第一位将军还可以为每条息编号，说这是1号，2号……直到n号。这种方法能让第二位将军知道通信信道到底有多靠，并且返回合适的数量的消息来确保最后一条消息被接收到。如果信道是可靠的话，只要一条消息行了，其余的就帮不上什么忙了。最后一条和第一条消息丢失的概率是相等的。
两将军问题可以扩展成更变态的<code>拜占庭将军问题</code> (Byzantine Generals Problem)，其故事背景是这样的：拜占庭位于现在土耳其的伊斯坦布尔，是罗马帝国的首都。由于当时拜占庭罗马帝国国土辽阔，为了防御目的，因此每个军队都分隔很远，将与将军之间只能靠信差传消息。 在战争的时候，拜占庭军队内所有将军必需达成一致的共识，决定是有赢的机会才去攻打敌人的阵营。但是，军队可能有叛徒和敌军间谍，这些叛徒将军们会扰乱或左右策的过程。这时候，在已知有成员谋反的情况下，其余忠诚的将军在不受叛徒的影响下如何达成一致协议，这就是拜占庭将军问题。

<h3 id="paxos算法" data-anchor-id="wmeb">Paxos算法</h3>
第一阶段：Prepare阶段
A把申请修的请求Prepare Request发给所有的结点A，B，C。注意，Paxos算法会有一个Sequence Number你可以认为是一个提案号，这个数不断递增，而且是唯一的，也就是说A和B不可能有相同的提案号）这个提案号会和修改请求一同发出，任何结点在“Prepare阶段”时都会拒绝其值小于前提案号的请求。所以，结点A在向所有结点申请修改请求的时候，需要带一个提案号，越新的提案这个提案号就越是是最大的。
如果接收结点收到的提案号n大于其它结点发过来的提案号，这个结点回应Yes（本结点上最新的被批准提案号），并保证不接收其它
优化：在上述 prepare 过程中，如果任何一个结点发现存在一个更高号的提案，则需要通知 提案人，提醒其中断这次提案。
第二阶段：Accept阶段
如果提案者A到了超过半数的结点返回的Yes，然后他就会向所有的结点发布Accept Request（同样，需要带上提号n），如果没有超过半数的话，那就返回失败。
当结点们收到了Accept Request后，如果对于接收的结点来说，n是大的了，那么，它就会修改这个值，如果发现自己有一个更大的提案号，那么，结点就会拒绝修改。
我们可以看以，这似乎就是一个“两段提交”的优化。其，2PC/3PC都是分布式一致性算法的残次版本，Google Chubby的作者Mike Burrows说过这个世界

只有一种一致性算法，那就是Paxos，其它的算法都是残次品。
<hr />
重新梳理分布式一致性协议

<h3 id="二阶段提交协议" data-anchor-id="xqxg">二阶段提交协议</h3>
阶段一 : 提交事务请求
<blockquote class="white-blockquote" data-anchor-id="dy92">

事务询问
向参与者发送事务内容,询问是否可以进行事务提交操作.并开始等待参与的响应
参与者执行事务
参与者收到消息之后,进行上锁, 记录undo/redo log
参与者向协调者反馈事务事务询问的响应
这个过程相当于投票的过程.称为投票阶

</blockquote>
阶段二 : 执行事务提交
当第一阶段都返回yes的时候,协者会发出事务提交的指令.
<blockquote class="white-blockquote" data-anchor-id="s44z">

发出提交请求
协调者向所有参与者发出事务提交请求
事务提交
参与者收到提交请求之后,会正式执行事务提交操作.并在提交完成之后释占用的事务资源
参与者反馈事务提交结果
当所有反馈都是yes的时候,说明事务提交成功.

</blockquote>

<h4 id="缺点" data-anchor-id="h5vd">缺点:</h4>
<blockquote class="white-blockquote" data-anchor-id="rtbx">

同步阻塞
在执行事务提交的过程中,一直处于阻塞状态
单点问题
在阶段二中,如果协调者出现故障,其他参与者将会一直处于锁定事务资源状态.
数据不一致
保守
协调者只能通过超时机制来判断,没有完善的容错机制

</blockquote>

<h3 id="三阶段提交协议" data-anchor-id="sou9">三阶段提交协议</h3>
阶段一 : canCommit
<blockquote class="white-blockquote" data-anchor-id="xu11">

事务询问
各参与者向协调者反馈询问

</blockquote>
阶段二 : PreCommit
<blockquote class="white-blockquote" data-anchor-id="0ejf">

协调者发送预提交请求
参与者执行事务操作,并记录undo/redo log
各参与者向协调者反馈执行结果
<code>可能有事务中断的情况</code>

</blockquote>
阶段三: doCommit

<blockquote class="white-blockquote" data-anchor-id="ayn5">

协调者向参与者发送提交请求
事务提交
参与者反馈事务结果
<code>可能事务中断</code>

</blockquote>
优点
<blockquote class="white-blockquote" data-anchor-id="ndgk">

最大化的降低了参与者的阻塞范围

</blockquote>
lamport在论文中提出了一种计算机容错理论,理论的描述就是<code>占庭将军问题</code>,在分布式计算领域,试图在异构网络和不可靠信道上实现一致性状态是不可能的然后实际上,大多数系统都是部署在一个局域网里,消息被篡改的几率非常小.所以在工程实践中,可以假不存在<code>拜占庭将军问题</code>

<h3 id="paxos算法详解" data-anchor-id="ksqv">paxos算法详解</h3>

<h3 id="zookeeper的设计目标" data-anchor-id="qsly">zookeeper的设计目标</h3>
致力于提供一个高性能高可用的分布式协调服务.对大型分布式系统的点问题能很好的解决,起源于雅虎

<h1 id="1-服务器角色介绍" data-anchor-id="uoc0">1. 服务器角色介绍</h1>
<ol data-anchor-id="open">

leader
是整个zookeeper集群工作机制的核心，主要工作有两个
<blockquote class="white-blockquote">

事务请求的唯一调度者和处理者，保证集群中事务处理顺序
集群内部各服务器的调度者

</blockquote>

Follower
zookeeper集群状态的跟随者，主要工作有
<blockquote class="white-blockquote">

处理客户端非事务请求，转发事务请求给leader服务器。
参与事务请求 proposal的投票
参与leader选举投票

</blockquote>

Observer
充当一个观察者的角色，和Follower类似，对非事务请求，都可以进行立处理。而对于事务请求，则交给leader来处理。和follower的区别是，Observer不参与任何形式的票。通常用于不影响集群事务处理能力的情况下增加集群非事务处理能力。

<h1 id="2-leader选举" data-anchor-id="wvfc">2. Leader选举</h1>
SID：用来唯一标识zookeeper集群中的一台机器，不能重复，和myid致
myid：机器编号的id
ZXID：事务Id，根据事务的执行情况增加。数越大表示状态最新。数据恢复最快。
至少两台服务器才可以，这里用三台来说明问题。

erver1（1,0）、server2（2,0）、server3（3,0）

<h4 id="1-服务器启动期间的leader选举" data-anchor-id="0un3">1. 服务器启动期间的leader选
</h4>
<ol data-anchor-id="ydtk">
每个server发出一个投票
由于是初始阶段，每台服务器都会把自己当做Leader服
器来投票。server1（1,0）、server2（2,0）,然后发给集群中其他机器。
接受来自各个服务器的投票结果。
每台服务器都会接收来自其他服务器的投票，
收到后先判断有效性，包括检查是否是本轮投票，是否来自Looking状态的服务。

处理投票
对接收到的每一个投票，都需要和自己的投票进行pk，pk规则如下：</
>
<blockquote class="white-blockquote">

先检查ZXID，ZXID大的优先作为Leader
ZXID相同的话，比较myid，myid大的作为Leader服务器。

</blockquote>
对于server1（1，0）来说，收到的投票（2,0）大于自己的投票，于是更新自己的投票为（2,0
，再向集群中发一次。而server2（2,0）不用变，直接向集群中再发一次。

统计投票
每次投票。服务器都会统计是否有过半的服务器接收了相同的投票。当
的时候，认为已经选出了Leader
改变服务器状态。
一旦确定了Leader，每个服务器就更新自己的状态。如果是Foll
wer，改变为Following，如果是Leader，变为Leading。

<h4 id="2-服务器运行期间的leader选举" data-anchor-id="ynnv">2. 服务器运行期间的leader选
</h4>
zookeeper集群在正常运行过程中，一旦选出了Leader，所有服务器
集群角色不再发生变化，除非Leader节点挂了。进入新一轮的Leader选举。运行期间的Leader选举
启动时候的Leader选举大致类似。
server1（1,123）、server2（2,123）、server3（3
122）共三台机器。假设leader是server2，现在server2挂了。
<ol data-anchor-id="bonh">
状态变更
Leader挂了之后，剩余的Observer都会将自己的服务器状态变更为LOOK
NG，进入Leader选举流程。
每个server发出一个投票
生成投票信息（myid，ZXID）。第一轮投票都会投自己
生成（1,123）、（3,122）的投票信息，并发送给其他机器。
接收其他机器的投票信息
处理投票
（1,123）会成为Leader
统计投票
改变服务器状态。

<h1 id="3-zookeeper应用实践" data-anchor-id="0sqz">3. zookeeper应用实践</h1>
<ol data-anchor-id="qxf3">
Canal（阿里）
解决数据库主从同步问题。伪装成一个slave，不断向master发送d
mp请求，master收到请求后，不断推送binlog给slave，Canal收到binlog解析后就可以进行二次消
了。
Hbase
全称Hadoop database。分布式的文件存储系统。
kafka
分布式消息系统，由LinkedIn于2010年开源。scala语言编写。低延迟的发
和收集大量的事件和日志数据。
hadoop
分布式计算框架。核心是HDFS和Mapreduce（最新一代 是YARN），
别提供了对海量数据的存储和计算能力

Dubbo（阿里）
RPC服务框架。服务发现，服务治理等方案
metamorphosis
阿里的分布式消息中间件
storm
Twitter开源的高容错分布式实时计算系统。
jstorm是阿里在storm的基础上改进的实时计算系统，稳定性更好，功能更多。用于一些无状态据的实时计算上。

<h1 id="4-zk分布式锁具体怎么实现" data-anchor-id="7vk3">4. zk分布式锁具体怎么实现</h1>

<h1 id="分布式一致性哈希" data-anchor-id="qhsu">分布式一致性哈希</h1>
<p data-anchor-id="hjia">Hash 算法的一个衡量指标是单调性（ Monotonicity ），定义如下：<p>
<pre data-anchor-id="chsr"><code>单调性是指如果已经有一些内容通过哈希分派到了相应的缓中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到新的缓冲中，而不会被映射到旧的缓冲集合中的其他缓冲区。</code></pre>