通过Zookeeper在分布式应用程序中处理互斥

【摘要】Zookeeper是一种功能强大，可行的方法来构建实施开放API的分布式应用程序，使开发人员能够应用自己强大的协调原语。 这项研究的目的是双重的）研究zookeeper的解剖和生命周期，并利用作为分布式应用程序的高性能协调服务的作用）在支付过程中介绍了Zookeeper实现的案例研究，其中同步 并且由于双服务器实现，通过estore应用程序放置订单，协调不能满足。 最后，Zookeeper复制的潜力被认为是解决可靠性和性能问题。

【关键词】zookeeper，Z序节点，相互排斥，分布式应用程序。

1. 介绍

分布式系统由多台计算机组成，通过计算机网络进行通信，并相互交互以实现共同目标。 zookeeper是一个开源服务器，可实现高可靠性的分布式应用。它有助于在高度分散的互联网规模系统中进行等待协调，互斥是主要关注的问题。 这是实现共享系统互斥的一种方式。 它允许分布式应用程序通过与标准文件系统非常相似的共享层级命名空间相互协调。 名称空间由类似于文件和目录的znode组成。 zookeeper的每个节点都允许与其孩子一起拥有与之相关联的数据。

* 1. Zookeeper支持以下API：
* Create：在树中的某个位置创建一个节点
* Delete：删除节点x存在：检查节点是否存在于某个位置
* GetData：从节点读取数据
* SetData：将数据写入节点
* Get Children：检索节点的子节点
* Sync：等待传播数据
  1. zookeeper的解析

Zookeeper涉及3个基础实体的相互作用，因为它为分布式应用程序提供了等待协调服务：

* 客户端是使用Zookeeper服务的机器。
* 服务器表示使用Zookeeper服务的服务。
* Znode是命名空间层次结构中的内存数据节点。
  1. Zookeeper的生命周期
* 启动：它创建一批待处理作业。 然后一次处理一个作业并检查其状态。如果任务的状态未完成，则会为挂起的任务创建一个父节点和子节点。
* 执行：创建子节点意味着任务no。 属于它经历处理。 创建子节点后，Zookeeper会向其应用锁定，直到任务被完全处理为止。
* 结论：完成任务处理后，与完成任务相关的子节点被删除。多个节点的创建和锁定可以互相排斥的方式来对各个任务进行无偿处理。

Zookeeper从一套可靠的分布式系统技术和协议中获得其鲁棒性，并在一组机器上运行，以提供无缝服务。

1. 方法
   1. 事态

客户可以从软件开发公司开发的商店购买产品。在购买产品时，支付流程的结算在另一个称为结算点的节点处理。应用程序和结算节点在两个不同的服务器上运行，用于负载分配。在这两台服务器之间保持数据同步/流。在结算节点中，调度程序从数据库中选择正在等待结算的订单，这意味着未处理付款的订单。然后，调度程序将创建一批未处理的订单并处理它们。在处理订单后，其状态更改为CLOSED在数据库中。下次运行调度程序时，排除处于CLOSED状态的那些订单。

2.2问题范围

如上一节所述，两台服务器上的结算节点的存在，强制数据流和调度器都在两个服务器上运行，都会导致一个主要问题。在某些情况下，运行在不同服务器上的调度程序会收到相同的订单，从而客户被收费两次。以下是上述问题领域的图示：

Server1

Server1

DB

DB

两个一致的订单

订单处理两次（客户开帐单）

图1 问题范围

2.3 过程采用：Zookeeper环境通过确保分布式锁的同步和应用程序来防止这种情况的发生，以确保在处理过程中相互排斥。

在等待订单的结算期间，调度程序检查数据库以查找待处理的订单。它创建了一批待处理订单。

如果状态未关闭，它一次需要一个订单，检查订单的状态（是否关闭）：Zookeeper首次创建一个父节点，在该节点下将创建所有其他子节点。它将每个订单视为子节点。在创建父节点之后，它检查该订单的子节点是否已经存在。如果节点已经存在，那么它限制为该顺序创建任何另外的节点，否则它将在父节点下创建一个新的子节点。在创建特定订单号码进行进一步处理和结算（支付相关）的子节点时）。在结算订单时，订单的状态更新为“关闭”。Zookeeper将删除该订单创建的子节点。

上述步骤有助于排除每个订单，从而防止客户收费两次。例如，结算作业正在两个不同的服务器中执行。当在不同的服务器上运行的调度程序都可以获得同样的订单待处理以进行进一步处理时，可能会出现这种情况。这将导致客户收费两次。 Zookeeper服务器将通过其互斥功能来防止这种情况。一旦收到订单以进行结算，Zookeeper将为其创建一个节点，并将为其提供锁定，直到被处理为止。一旦处理，它将释放锁并且节点将被删除。如果在不同的服务器上运行的其他结算作业拾取相同的顺序，则它将无法为其创建节点并进一步处理它，如该节点已经存在于Zookeeper中。

以下是上述情况的图示表示和流程图：

是否存在子节点？

是否存在父节点？

是

是

线程

下个线程

否

否

创建父节点

创建子节点

锁定节点

进程节点/命令

删除节点

释放锁

JVM1

JVM2

DB

DB

ZOOKEEPER（Zookeeper锁定顺序，并且不允许其他线程处理订单，直到锁定被释放）

处理一次订单（客户只收一次）

1. 编码与实现

3.1 步骤

* 稳定版本的Zookeeper可以从以下链接下载：http://zookeeper.apache.org/releases.html
* 对conf文件夹中的zoo.cfg文件进行更改将dataDir更改为要在其中创建快照的webServer的某个位置。
* 在'zkServer.sh'文件中增加export JAVA\_HOME=<Insert app’s JAVA\_HOME> 在文件的顶部。
* 转到/ bin＆run'./zkServer.sh start'
* 检查服务器状态，运行'./zkServer.sh status'
* 添加zookeeper maven插件到您的应用程序。
* 将zookeeper jar粘贴到所有库存在的文件夹中

3.2在所述问题领域实施的Zookeeper的伪代码

* 创建一批待处理订单。
* 一次一个订单检查订单的状态。
* 我们检查了该订单的childNode是否已经存在。

if(isChildCreated=createZookeeperNodeForOrderId (orderUid))

* 在createZookeeperNodeForOrderId中实现：首先创建一个新的Zookeeper对象，然后创建子节点，如下所示：

String childNode = PARENT\_NODE + "/"+orderUid;

Boolean isChildExists = Boolean.FALSE;

* 然后我们通过调用Zookeeper的exists() API来检查父节点是否存在：

if (null == zooKeeper.exists(PARENT\_NODE, false))

那么我们创建一个父节点，否则父节点已经存在。

* 接下来我们通过调用Zookeeper的exists() API来检查子节点是否存在：

if (null == zooKeeper.exists(childNode, false))

* 通过调用Zookeeper的create() API来创建一个子节点：

String childPath = zooKeeper.create();

isChildExists = Boolean.TRUE;

否则isChildExists = Boolean.FALSE; return isChildExists;

* 执行待处理订单。
* 最后，当订单状态更改为关闭时，我们删除创建的子节点。

deleteChildNode(orderUid);

* 在deleteChildNode中实现：

if (null != zooKeeper.exists(childNode, false))

* 调用Zookeeper的删除API：

zooKeeper.delete (childNode, -1)

1. 结果

通过在我们的应用程序上实施Zookeeper，解决了客户被收费两次的主要问题。 待处理订单只处理一次。 一旦一个线程拿起一个订单进行结算zookeeper为该任务创建了一个名为znode的节点，并为其分配一个锁，然后处理该订单。处理订单后，znode已删除，订单状态在数据库中已更改。 即使某个其他线程选择了相同的顺序，因为该任务的znode已经存在，所以没有输入到处理部分。 在分布式应用程序中实现相互排斥。由于只有一旦客户只收取一次订单才被处理。 通过的过程解决了我们的问题。

1. 总结

Zookeeper环境确保分布式锁的同步和应用，以保证对进程的互斥。这是一个很好的协调服务。锁定的znode，分配，释放的创建类似于信号量，这有助于许多互相排斥问题。关键部分一次只能由一个进程访问。独立运行的Zookeeper方便评估，开发和测试。基于我们的问题陈述，我们创建了一个Zookeeper服务器，虽然同一进程的复制可以创建仲裁。将来，Zookeeper服务器的复制将被视为解决可靠性和性能问题。即使一个人的失败也不会中断这个过程，因为它会被其他人接管。这里的感觉是使用Zookeeper作为“快速失败”，这意味着如果进程在运行阶段存在异常，那么它将自动重新启动，并重新加入仲裁以提供不间断的服务。

【参考文献】

[1] Hadoop: <http://zookeeper.apache.org>

[2] Ranganath Atreya “A Quorum-Based Group Mutual Exclusion Algorithm for a Distributed System with Dynamic Group Set”, Parallel and Distributed Systems, IEEE,18 , Issue: 10, Oct. 2007.pp 1345 – 1360.

[3] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC), Aug.2007.

[4] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In SOSP ’07: Proceedings of the 21st ACM symposium on Operating systems principles, New York, NY, 2007.

[5] Yoshida, E. ”A learning system for the problem of mutual exclusion in multithreaded programming”. Advanced Learning Technologies, 2004. Proceedings. IEEE International Conference, 30 Aug.-1 Sept. 2004,pp 2 - 6

[6] A. Hastings. Distributed lock management in a transaction processing environment. In Proceedings of IEEE 9th Symposium on Reliable Distributed Systems, Oct. 1990.