从PAXOS到ZOOKEEPER分布式一致性原理与实践

1.集中式特点：

所谓有的集中式系统就是指由一台或多台主计算机组成中心节点，数据集中存储于这个中心节点中，并且 整个系统的所有业务单元都集中部署在这个中心节点上，系统的所有功能均由其集中处理。每个终端或客户端负责数据的录入和输出，存储与控制处理完全交由主机处理。

2.分布

分布式系统是一个硬件与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。特征：

分布性：分布式系统中的多台计算机都会在空间上随意分布，同时，机器的分布情况也会随时变动。

对待性：分布式系统中的计算机没有主/从之分，即没有控制整个系统的主机，也没有被控制的从机，组成分布式系统的所有计算机节点都是对等的。副本是分布式系统最常见的概念之一，指的是分布式系统对数据和服务提供一种冗余方式。在常见的分布式系统中，为了对外提供高可用的服务，我们会对数据和服务进行副本处理。数据副本是指在不同的节点上持久化同一份数据，当某一个节点上存储的数据丢失时，可以从副本上读取到数据，这是解决分布式系统数据丢失的问题最为有效的手段。另一类是副本是服务副本，指多个节点提供同样的服务，每个节点都有能力接收来自外部的请求并进行相应的处理。

并发性：在一个计算机网络中，程序运行过程中的并发性操作是非常常见的行为，例如同一个分布式系统中的多个节点，可能会并发地操作一些共享的资源，诸如数据库或分布式存储等，如何准确并高效地协调分布式并发操作也成为了分布式系统架构与设计中最大的挑战之一。

缺乏全局时钟：

一个典型的分布式系统是由一系列在空间上随意健在的多个进程组成，具有明显的分布性，这些进程 之间通过交换消息来进行相互通信。因此，在分布式系统中，很难定义两个事件究竟谁先谁后，原因就是因为分布式系统缺少一个合局的时钟序列控制。

故障总是会发生：组成分布式系统的所有计算机，都有可能发生任何形式的故障。

分布式环境的各种问题：

通信异常：网络不可靠，断网，设备故障，通信会被延时。或消息丢失 。

网络分区：由于 网络异常只有部分节点能够正常通信 。

三态：成功，失败与超时。

节点故障：服务器节点宕机。

ACID：原子性(Atomicity)，一致性(consistency)，隔离性(isolation)，持久性(durability)

事务是一个对系统中数据进行操作的所能成的程序执行逻辑单元。当多个应用程序并发访问数据库时，事务可以对这些应用程序之间提供一个隔离方法，以防止彼此的操作互相干扰。能够让数据从失败中恢复正常，在异常情况下保持数据一致。

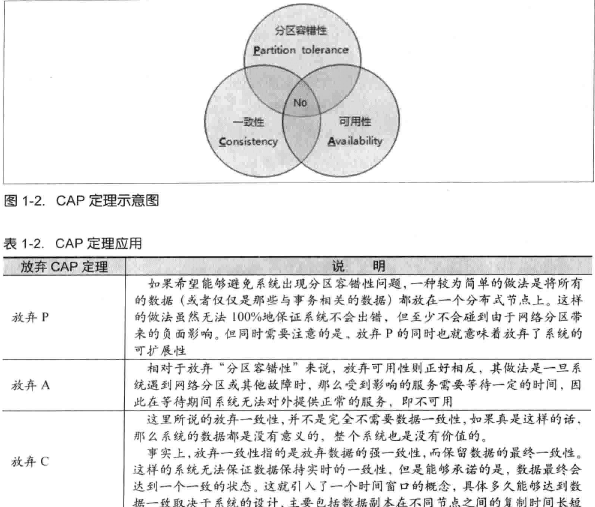
原子性：要不全部成功执行，要不全部不执行。任何一个操作失败都将导致整个事务失败。

一致性：事务的执行不能破坏数据库的完整性和一致性。

隔离性：并发环境中，并发的事务是相互隔离的，一个事务的执行不能被其他事务干扰。

串行化是最严格的事务隔离级别。它要求所有事务都被串行执行，即事务只能一个接一个地进行处理，不能并发执行。

在单机中，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之上。通常一个分布式事务中会涉及对多点数据源或业务系统的操作。



不能同时满足3项，只能满足2项。看看zookeeper能不能做到。

**现在分布式系统都用二阶段提交协议来完成分布式事务处理的：**

**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。**

**3.各参与者向协调者反馈询问的响应**

**如果参与者成功了事务操作，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，那么就反馈给协调者No响应，表示事务不可以执行。**

**阶段二：执行事务提交**

**协调者会根据各参与者的反馈情况来决定是否可以进行事务提交。**

**1.发送提交请求：协调者向所有参与者节点发出commit请求。**

**2.事务提交：参与者接收到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在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消息。**

**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。**

**中断事务：**

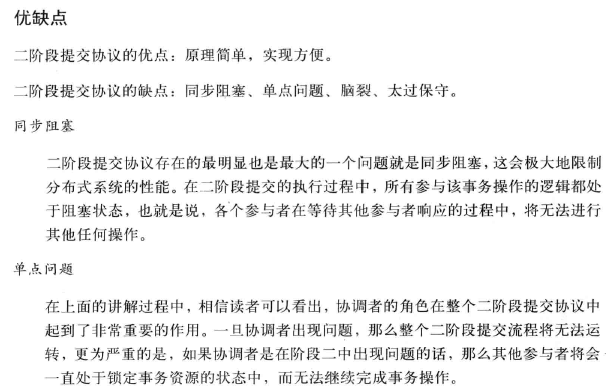
**假如任何一个参与者向协调者反馈了No响应，或者在等待超时之后，协调者尚玩法接收所有参与者的反馈响应，那么就会中断事务。**

**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，完成事务中断。**



理解2pc和3pc:

<https://coderknock.com/blog/2016/12/18/PC.html>

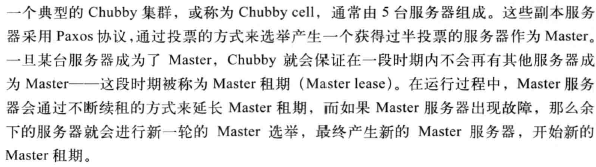
Paxos算法：

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。

<http://www.cnblogs.com/endsock/p/3480093.html> paxos算法 。

Chubby是Google的一个分布式锁服务。

应用场景：最典型的就是集群中服务器的Master选举。在Chubby中通常使用5台服务器来组成一个集群单元，根据Quorum机制 ，只要整个集群中有3台服务器是正常运行的，那么整个集群就可以对外提供正常的服务。相反，如果仅提供一个分布式一致性协议的客户端库，那么这些高可用性的系统部署都将交给开发人员自己来处理，这无疑提高了成本。

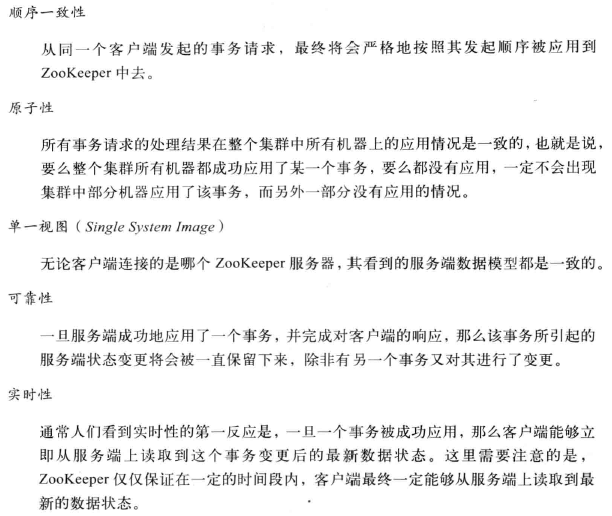


集群中的每个服务器都维护着一份服务端数据库的副本，但在实际运行过程中，只有Master服务器才能对数据库进行写操作，而其他服务器都是使用Paxos协议从Master服务器上同步数据库数据的更新。

Chubby上的每个数据节点都分为持久节点得临时节点。其中持久节点需要显示的调用接口api来进行删除，而临时节点则会在其对应的客户端会话失效后被自动删除。

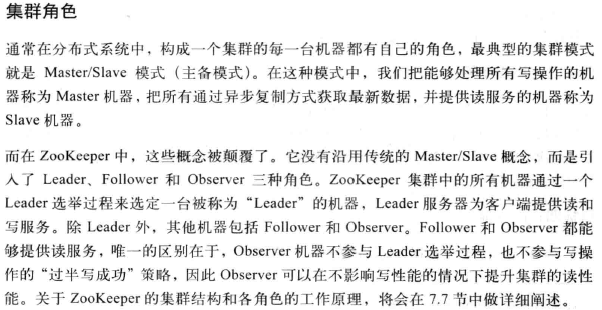
ZooKeeper为分布式应用提供了高效且可靠的分布式协调服务，提供了诸如统一命名服务，配置管理和分布式锁等分布式的基础服务。在解决分布式数据一致性方面，zookeeper没有使用Paxos算法，用的是ZAB(zookeeper atomic broadcast). 它是一个典型的分布式数据一致性的解决方案，分布式应用程序可以基于它实现诸如数据发布/订阅，负载均衡，命名服务，分布式协调/通知，集群管理，Master选举，分布式锁和分布式队列等功能。

ZooKeeper可以保证 ：



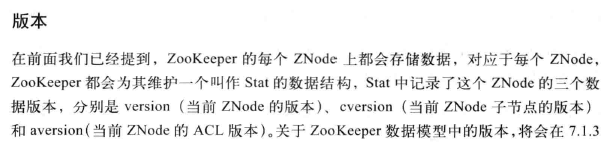
ZooKeeper致力于提供一个高性能，高可用，且具有严格的顺序访问控制能力（主要是写操作的严格顺序性）的分布式协调服务。高性能使得ZooKeeper能够应用于那些对系统吞吐有明确要求的大型分布式系统中，高可用使得分布式的单点问题得到了很好的解决，而严格的顺序访问控制使得客户端能够基于Zookeeper实现一些复杂的同步原语。

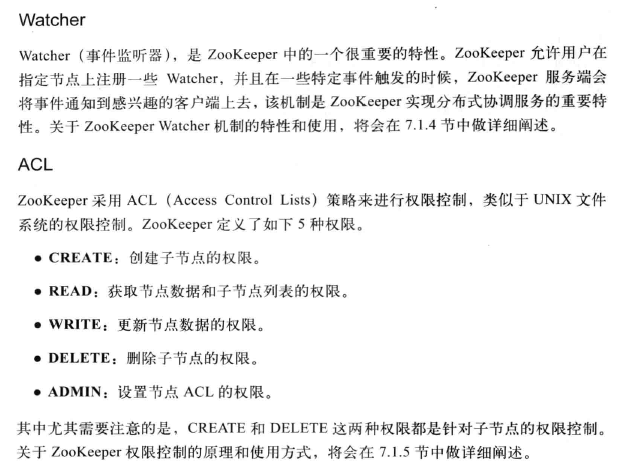
Zookeeper服务器内存中的一个数据模型，由一系列ZNode的数据节点组成，类似一个文件系统，而ZNode之间的层级关系，就像文件系统的目录结构一样。Zookeeper将全量数据存储在内存中，以此来实现提高服务器吞吐，减少延迟的目的。由3-5台机器组成一个可用的Zookeeper集群。组成Zookeeper集群的每台机器都会在内存中维护当前的服务器状态，并且每台机器之间都互相保持通信。只要存在超过一半的机器能够正常工作，那个整个集群就能够正常对外服务。



客户端与服务器连接产生session,在sesstionTimout之间如果session断开了，只要它能在超时时间内重新连上集群中的任何一台服务器，它们之前创建的会话都还是有效的。

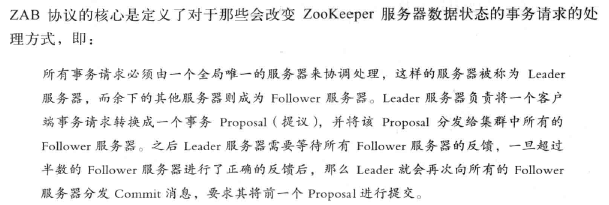
ZooKeeper将所有数据存储在内存中，数据模型是一棵树(ZNode tree)，由斜杠(/)进行分割的路径，就是一个Znode.ZNode可以分为持久节点和临时节点两类。持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在ZooKeeper上。临时节点它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。另外,Zookeeper还允许用户为每个节点添加一个特殊的属性:SEQUENTIAL.一旦节点被标记上这个属性，那么在这个节点被创建的时候，ZooKeeper会自动在其节点名后面追加上一个整型数字，这个整型数字是由一个由父节点维护的自增数字。





为什么选择ZooKeeper:

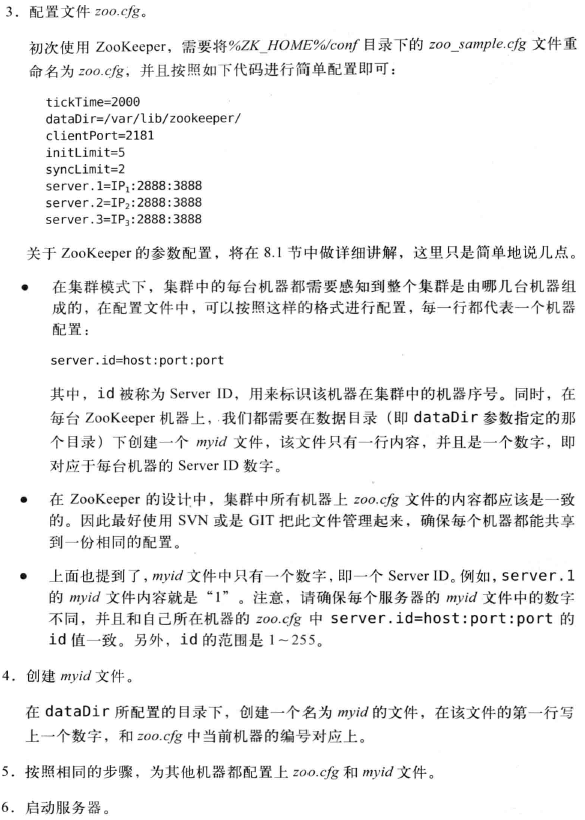
1.解决了分布式数据一致性。2.开源。3.免费

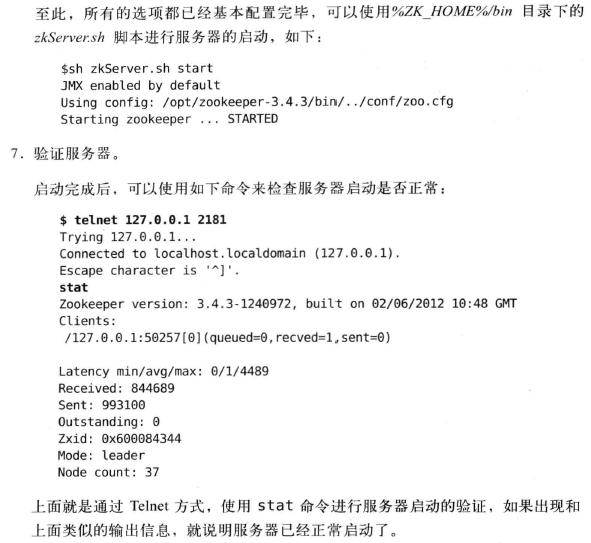


安装：

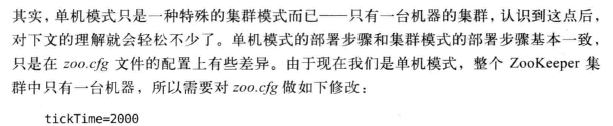
1.先安装java

2.下载ZooKeeper安装包:http://zookeeper.apache.org/releases.html ，解压到一个目录 ,设置%ZK\_HOME%。





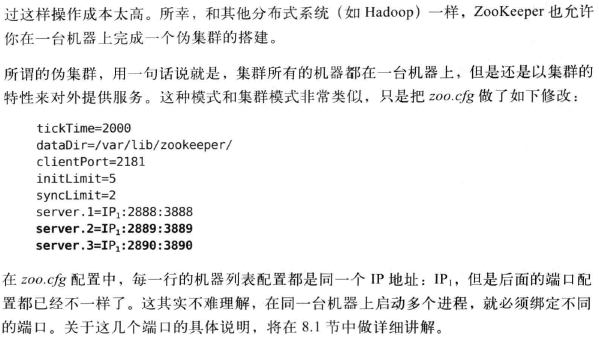
单机模式：

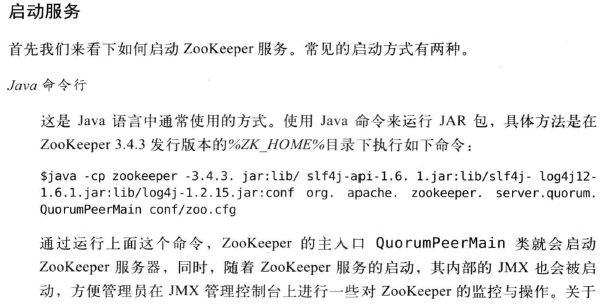




<http://apache.fayea.com/zookeeper/zookeeper-3.4.10/zookeeper-3.4.10.tar.gz>

伪集群：





或者用Zookeeper bin目录下文件执行：

zkServer : ZooKeeper 服务器的启动，停止 ，重启脚本

停止服务：$ sh zkServer.sh stop

用客户端看看是否连上：$sh zkCli.sh 连接远程机:$sh zkCli.sh -server ip:port

创建ZooKeeper节点：

create [-s] [-e] path data acl 比如(节点zk-book内容是123)：create /zk-book 123

其中-s或-e分别指定节点特性：顺序或临时节点。默认情况下，即不添加-s或-e参数，创建的就是持久节点。

读取:ls path [watch]

get: get path [watch] 比如: $get /zk-book

更新:set path data [version] 比如：$set /zk-book 456

删除: delete path [version]

不能批量创建节点。删除一个节点前必须要将它所有的子节点先删除掉。

ZooKeeper服务端在向客户端发送Watcher "NodeChildrenChanged"事件通知的时候，仅仅只会发出一个通知，而不会把节点的变化情况发送给客户端，需要客户端自己重新获取。另外由于 Watcher通知是一次性的，即一旦触发一次通知后，该Watcher就失效了，因此客户端需要反复注册Watcher.

数据内容或是数据版本变化，都会触发服务端的NodeDataChanged通知。

在zookeeper中数据版本都是从0开始计数的，所以严格来说"-1"并不是一个合法的数据版本，它仅仅是一个标识 符。

无论指定节点是否存在，通过调用exists接口都可以注册watcher.

exists接口中注册watcher,能够对节点的创建，更新，删除事件进行监听。

**if** (Event.EventType.***None*** == watchedEvent.getType() && **null** == watchedEvent.getPath()) {

*countDownLatch*.countDown();

} **else if** (Event.EventType.***NodeCreated*** == watchedEvent.getType()) {

System.***out***.println(**"创建 "** + watchedEvent.getPath());

*zk*.exists(watchedEvent.getPath(), **true**);

} **else if** (Event.EventType.***NodeDeleted*** == watchedEvent.getType()) {

System.***out***.println(**"删除 "** + watchedEvent.getPath());

*zk*.exists(watchedEvent.getPath(), **true**);

} **else if** (Event.EventType.***NodeDataChanged*** == watchedEvent.getType()) {

System.***out***.println(**"更新 "** + watchedEvent.getPath());

*zk*.exists(watchedEvent.getPath(), **true**);

}

必须要重新监听，不然就不会被监听到。

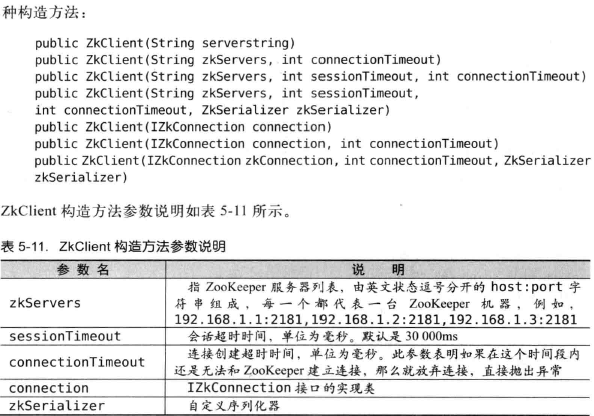
zookeeper提供了多种权限控制模式(scheme),分别是world,auth,digest,ip和super.ZooKeeper客户端提供了相应的Api接口来进行权限信息的设置，addAuthInfo(String scheme,byte[] auth);

zooKeeper.addAuthInfo(**"digest"**, **"foo:true"**.getBytes()); 类似于username:password的格式，完成权限信息的添加后，该示例还使用客户端会话在ZooKeeper上创建节点，这样此结点就受到了权限控制。

删除节点接口的权限控制比较特殊，当客户端对一个数据节点添加了权限信息后，对于删除操作而言，其作用范围是其子节点。也就是说，当我们对一个数据节点添加权限信息后，依然可以自由地删除这个节点，但坚于这个节点的子节点，就必须使用相应的权限信息才能删除掉它。

Zkclient, Curator是2个开源的ZooKeeper客户端产品。对Zookeeper原生api接口上进行了包装。





原生的Zookeeper会话的建立是一个异步的过程，开发人员需要自己来进行等待处理。而ZkClient将这个异步的会话创建过程同步化了。

Zkclient中:客户端可以对一个不存在的节点进行子节点变更的监听。

一旦客户端对一个节点注册了子节点列表变更监听之后，那么当该节点的子节点列表发生变更的时候，服务端都会通知客户端，并将最新的子节点列表发送给客户端。

该节点本身的创建或删除也会通知到客户端。

和ZooKeeper原生提供的Watcher不同的是，ZkClient的Listener不是一次性的，客户端只需要注册一次就会一直生效。

**Curator**是一套ZooKeeper客户端框架. 它还提供了ZooKeeper各种应用场景(Recipe,如共享锁服务，Master选举机制和分布式计数器等）的抽象封闭。

<dependency>

<groupId>org.apache.curator</groupId>

<artifactId>curator-framework</artifactId>

<version>2.4.2</version>

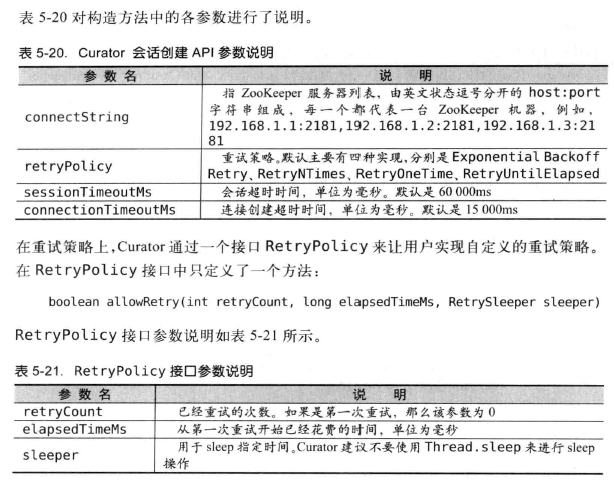
</dependency>

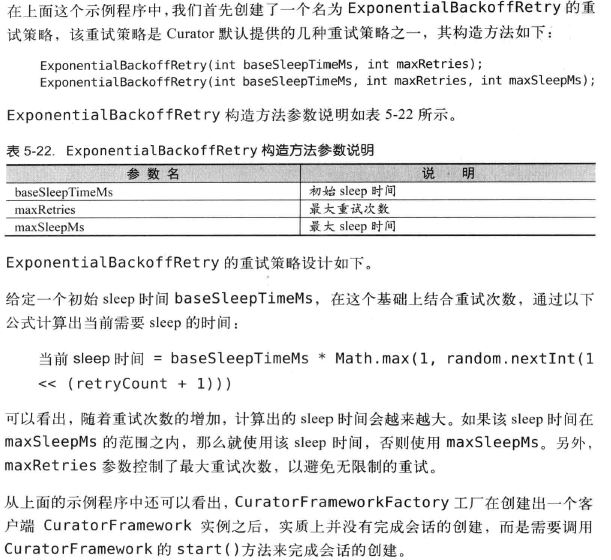
1.创建会话

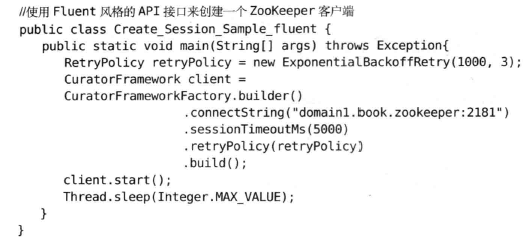
使用CuratorFrameworkFactory这个工厂类的两个静态方法来创建一个客户端：

static CuratorFramework newClient(String connectionString, RetryPolicy retryPolicy);

static CuratorFramework newClient(String connectionString, int sessionTimeoutMs, int connectionTimeoutMs, RetryPolicy retryPolicy);







Fluent风格的写法：

public class CuratorFrameworkFactory {

private static final int DEFAULT\_SESSION\_TIMEOUT\_MS = 2;

private static final int DEFAULT\_CONNECTION\_TIMEOUT\_MS = 3;

**public static CuratorFrameworkFactory.Builder builder() {**

**return new CuratorFrameworkFactory.Builder();**

**}**

public static CuratorFramework newClient(String connectString, RetryPolicy retryPolicy) {

return newClient(connectString, DEFAULT\_SESSION\_TIMEOUT\_MS, DEFAULT\_CONNECTION\_TIMEOUT\_MS, retryPolicy);

}

**private CuratorFrameworkFactory() {**

**}**

**public static class Builder {**

private int sessionTimeoutMs;

private int connectionTimeoutMs;

public CuratorFramework build() {

return new CuratorFrameworkImpl(this);

}

public CuratorFrameworkFactory.Builder connectString(String connectString) {

this.ensembleProvider = new FixedEnsembleProvider(connectString);

return this;

}

public CuratorFrameworkFactory.Builder sessionTimeoutMs(int sessionTimeoutMs) {

this.sessionTimeoutMs = sessionTimeoutMs;

return this;

}

public CuratorFrameworkFactory.Builder connectionTimeoutMs(int connectionTimeoutMs) {

this.connectionTimeoutMs = connectionTimeoutMs;

return this;

}

**private Builder() {**

this.sessionTimeoutMs = CuratorFrameworkFactory.DEFAULT\_SESSION\_TIMEOUT\_MS;

this.connectionTimeoutMs = CuratorFrameworkFactory.DEFAULT\_CONNECTION\_TIMEOUT\_MS;

this.threadFactory = null;

}

}

}

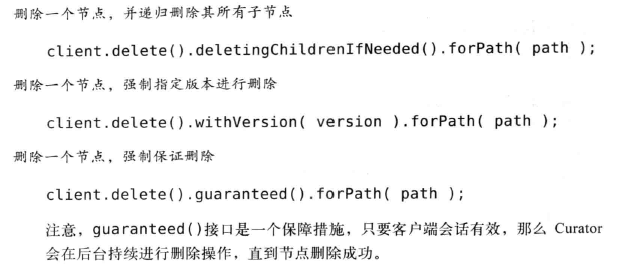
创建一个节点内容为空: client.create().forPath(path);

创建一个节点，附带初始内容: client.create().forPath(path, "init".getBytes());

创建一个临时节点，内容为空:clent.create().withMode(CreateMode.EPHEMERAL).forPath(path);

创建一个临时节点，并自动递归创建父节点: client.create().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL).forPath(path);

这样就不用判断父节点是否存在。ZooKeeper中规定了所有非叶子节点必须是持久节点，调用上面的这个api之后，只有path参数对应的数据节点是临时节点，其父节点均为持久节点。



curator中引入了一种重试机制：如果我们调用了guaranteed()方法，那么当客户端碰到上面这些网络异常的时候，会记录下这次失败的删除操作，只要客户端会话有效，那么其就会在后台反复重试，直到节点删除成功。通过这样的措施，就可以保证节点删除操作一定会生效。

读取一个节点的数据内容：client.getData().forPath(path);

读取一个节点的数据内容，同时获取此节点的stat: client.getData().storingStatIn(stat).forPath(path);

curator通过传入一个旧的stat变量的方式来存储服务端返回的最新的节点状态信息。

更新节点：client.setData().forPath(path);

更新一个节点的数据内容，强制指定版本进行更新：

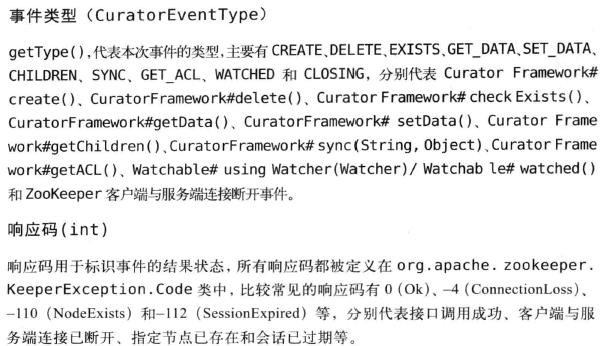
client.setData().withVersion(version).forPath(path);

注意：withVersion接口就是用业实现CAS(Compare and Swap)的，version（版本信息）通常是从一个旧的stat对象中获取得到的。

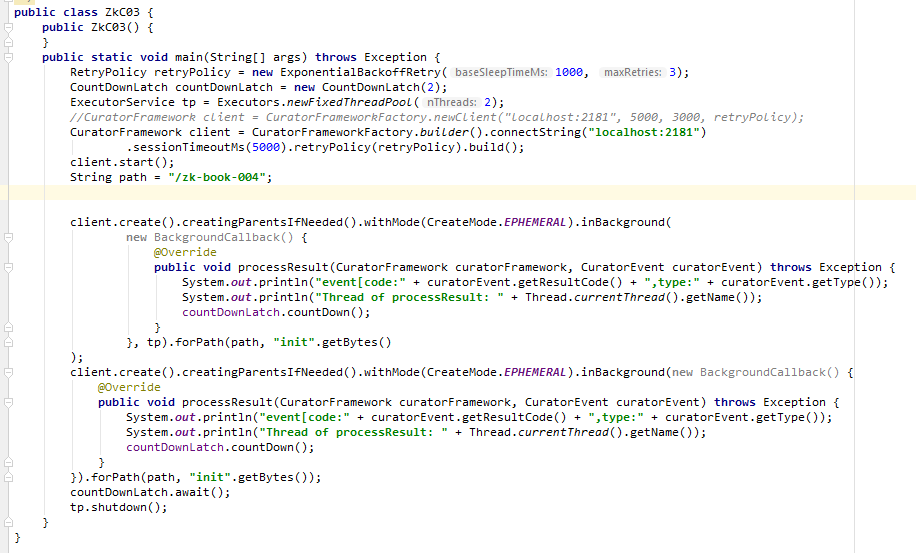
Curator中引入了BackgroundCallback接口，用来处理异步接口调用之后服务端返回的结果信息.

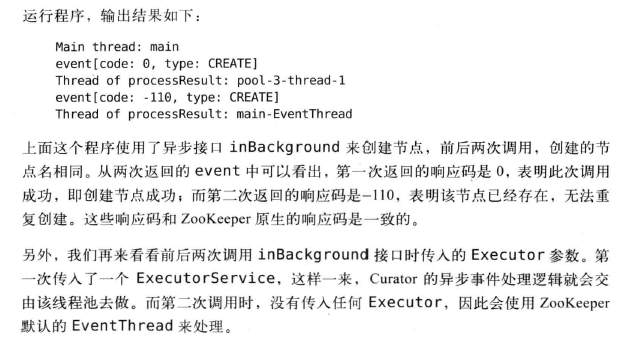
public void processResult(CuratorFramework client, CuratorEvent event) throws Exception ;

client:当前客户端实例；event :服务端事件;

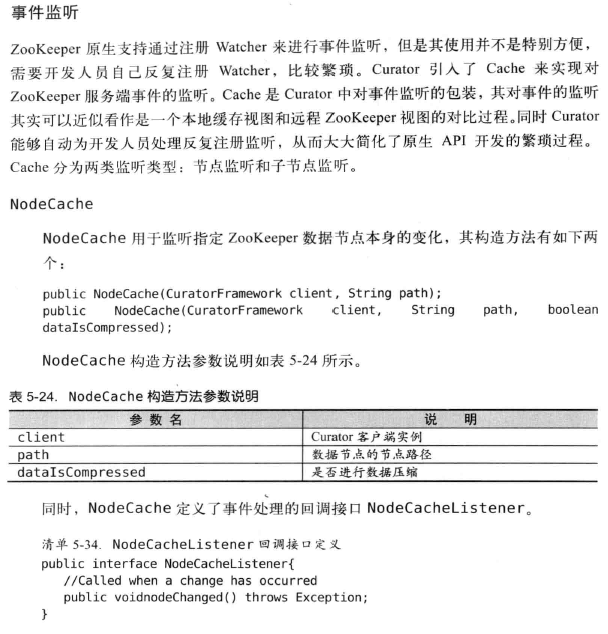


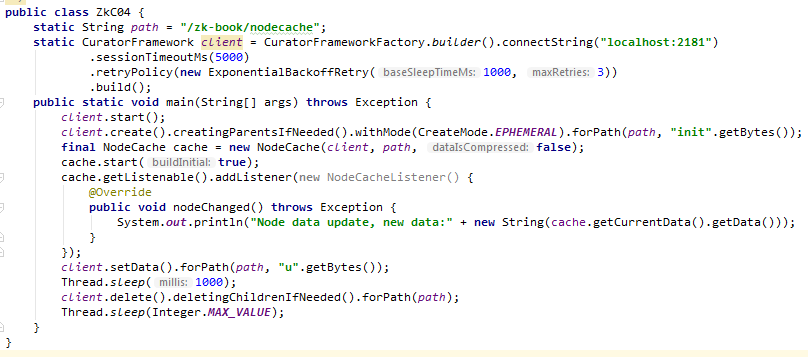






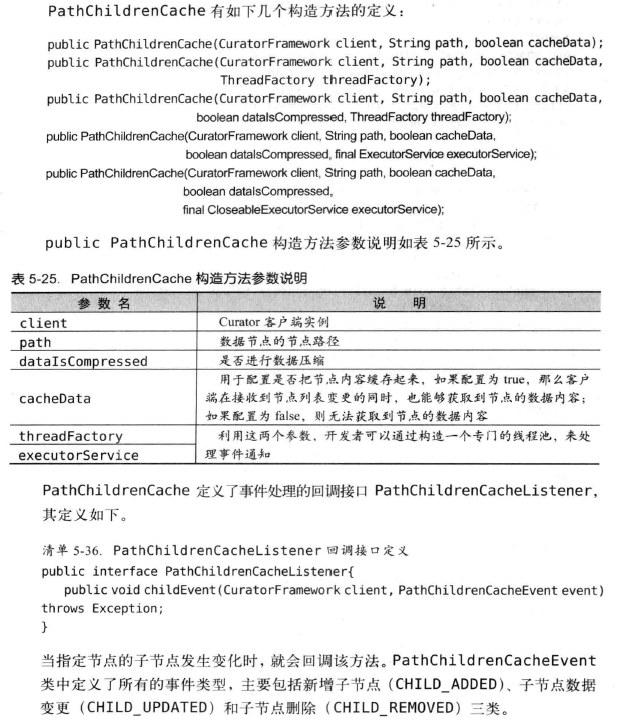
典型的使用场景：





构建一个NodeCache实例，然后调用start方法，该就去有个boolean类型的参数，默认是false,如果设置true,那么NodeCache在第一次启动的时候就会立刻从ZooKeeper上读取对应节点的数据内容，并保存在Cache中。NodeCache不仅可以用于监听数据节点的内容变更，也能监听指定节点是否存在。如果原来节点不存在，那么Cache就会在节点被创建后触发NodeCacheListener.但是如果此数据节点被删除，那么Curator就无法触发NodeCacheListener.

PathChildrenCache:用于监听指定ZooKeeper数据节点的子节点变化情况。



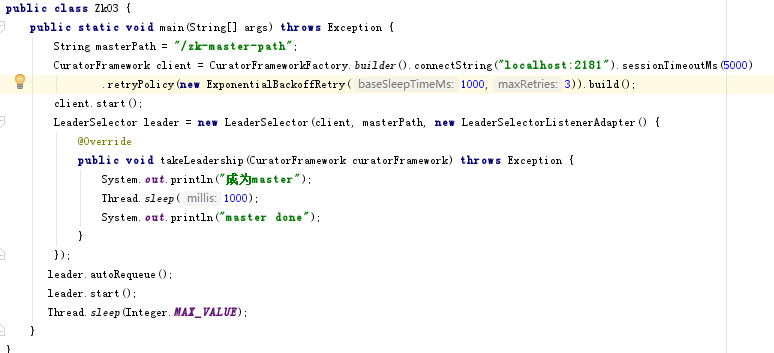
**Master选举**

在分布式系统中,经常会碰到这样的场景：对于一个复杂的任务，仅需要从集群中选举出一台进行处理即可。诸如此类的分布式问题，我们称为"Master选举".

ZooKeeper实现思路：

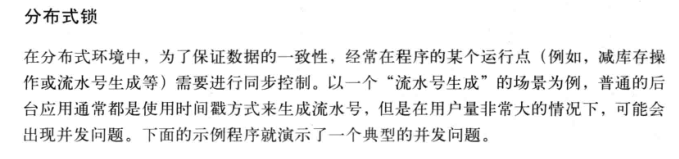
选择一个根节点，例如/master\_select,多台机器同时向该节点创建一个子节点/master\_select/lock,利用ZooKeeper的特性，最终只有一台机器能够创建成功，成功的那台机器就作为Master.

Curator也是基于此思路，但是它将节点创建，事件监听和自动选举过程进行了封装，开发人员只需要调用简单的API即可实现Master选举。





一旦执行完takeLeadership方法，Curator就会立即释放Master权利，然后重新开始新一轮的Master选举。当一个应用实例成为Master后，其他应用实例会进入等待，直到当前Master挂了或退出后才开始选举新的Master.



**public class** Zk04 {

**static** String *lock\_path* = **"/curator\_look\_path"**;

**static** CuratorFramework *client* = CuratorFrameworkFactory.*builder*()

.connectString(**"localhost:2181"**)

.retryPolicy(**new** ExponentialBackoffRetry(1000,3)).build();

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

*client*.start();

**final** InterProcessMutex lock = **new** InterProcessMutex(*client*, *lock\_path*);

**final** CountDownLatch down = **new** CountDownLatch(1);

**for** (**int** i = 0; i < 30; i++) {

**new** Thread(**new** Runnable(){

**public void** run(){

**try** {

down.await();

lock.acquire();

} **catch**(Exception e){

}

SimpleDateFormat sdf = **new** SimpleDateFormat(**"HH:mm:ss|SSS"**);

String orderNo = sdf.format(**new** Date());

System.***out***.println(**"生成的订单号是:"** + orderNo);

**try** {

lock.release();

}**catch**(Exception e){}

}

}).start();

}

down.countDown();

}

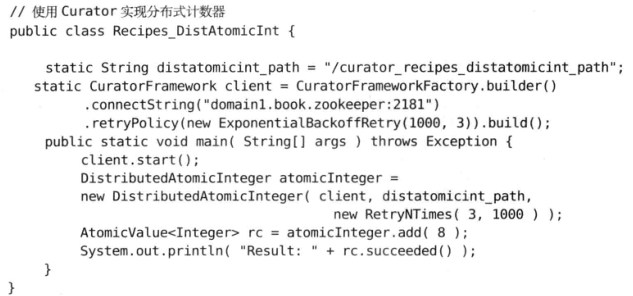
}

**分布式计数器**

有了上述分布式锁实现的基础之后，我们就很容易基于其实现一个分布式计数器。分布式计数器的一个典型场景是统计系统的在线人数。思路如下

指定一个ZooKeeper数据节点作为计数器，多个应用实例在分布式锁的控制下，通过更新该数据节点的内容来实现计数功能。

Curator同样将这一系列逻辑封装在了DistributedAtomicInteger类中，从其类名我们可以看出这是一个可以在分布式环境中使用的原子整型。



**分布式Barrier**

Barrier是一种用来控制多线程之间同步的经典方式，在JDK中也自带了CyclicBarrier实现。

**public class** Recipes\_Barrier {

**static** String *barrier\_path* = **"/curator\_recipes\_barrier\_path"**;

**static** DistributedBarrier *barrier*;

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

**for** (**int** i=0;i<5;i++){

**new** Thread(**new** Runnable(){

**public void** run(){

**try** {

CuratorFramework client = CuratorFrameworkFactory.*builder*().connectString(**"localhost:2181"**)

.retryPolicy(**new** ExponentialBackoffRetry(1000, 3)).build();

client.start();

*barrier* = **new** DistributedBarrier(client, *barrier\_path*);

System.***out***.println(Thread.*currentThread*().getName() + **" 号barrier设置"**);

*barrier*.setBarrier();

*barrier*.waitOnBarrier();

System.***err***.println(**"启动..."**);

} **catch**(Exception e){

}

}

}).start();

}

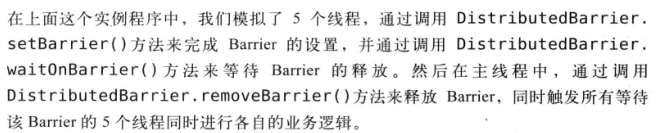
Thread.*sleep*(2000);

*barrier*.removeBarrier();

}

}

*如果不执行barrier*.removeBarrier(); 程序将一直阻塞。



**public class** Recipes\_Barrier {

**static** String *barrier\_path* = **"/curator\_recipes\_barrier\_path"**;

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

**for** (**int** i=0;i<5;i++){

**new** Thread(**new** Runnable(){

**public void** run(){

**try** {

CuratorFramework client = CuratorFrameworkFactory.*builder*().connectString(**"localhost:2181"**)

.retryPolicy(**new** ExponentialBackoffRetry(1000, 3)).build();

client.start();

DistributedDoubleBarrier barrier = **new** DistributedDoubleBarrier(client, *barrier\_path*, 5);

Thread.*sleep*(Math.*round*(Math.*random*()\*3000));

System.***out***.println(Thread.*currentThread*().getName() + **"号进入"**);

barrier.enter();

System.***out***.println(**"启动"**);

Thread.*sleep*(Math.*round*(Math.*random*()\*3000));

barrier.leave();

System.***out***.println(**"退出..."**);

} **catch**(Exception e){

}

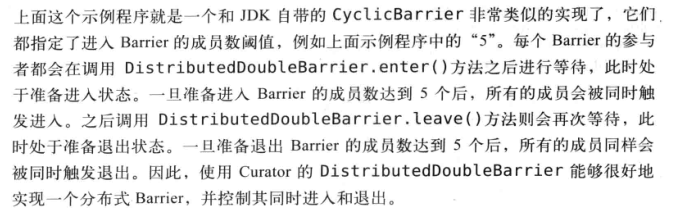
}

}).start();

}

}

}



Curator也提供了很多工具类，其中用得最多的就是ZKPaths和EnsurePath

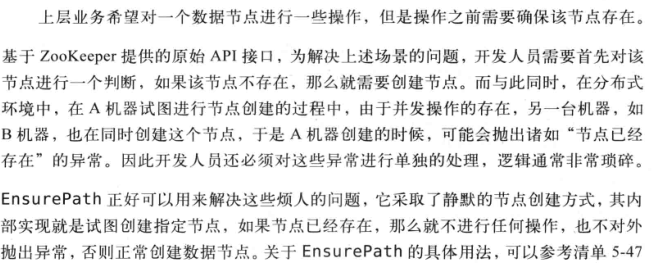
ZKPaths提供了一些简单的api来构建ZNode路径，递归创建和删除节点等，其使用方式非常简单。

ZKPaths.fixForNamespace(path,"sub");

ZKPaths.makePath(path, "sub");

ZKPaths.getNodeFromPath("/xxx/sub1");

EnsurePath 提供了一种能够确保数据节点存在的机制。



TestingServer

为了便于开发人员进行ZooKeeper的开发和测试，Curator提供了一种启动简易ZooKeeper服务的方法-TestingServer.TestingServer允许开发人员非常方便地启动一个标准的ZooKeeper服务器，并以此来进行一系列的单元测试。

<dependency>

<groupId>org.apache.curator</groupId>

<artifactId>curator-test</artifactId>

<version>2.4.2</version>

</dependency>

TestServer server = new TestServer(2181,new File("/home/admin/zk-book-data"));

TestingCluster

非常方便地在单元测试中启动一个ZooKeeper服务器，它是一个ZooKeeper集群的工具类。