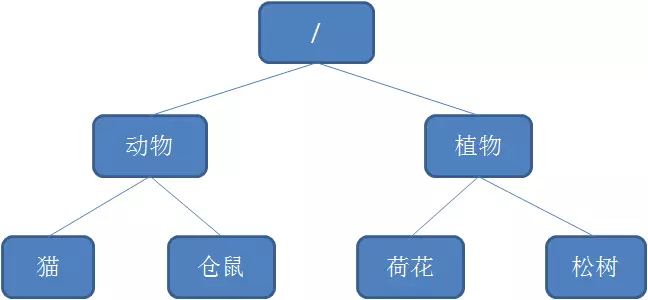
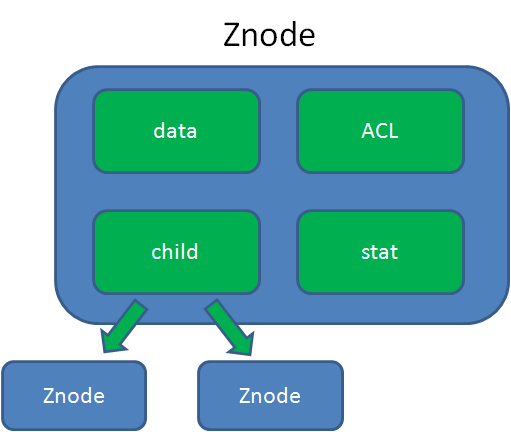
ZooKeeper

1）、Zookeeper是一种分布式协调服务，可以在分布式系统中共享配置，协调锁资源，提供命名服务。

ZooKeeper的数据模型像数据结构中的树，也很像文件系统的目录。



节点Znode,引用方式是路径引用，如：/动物/仓鼠，这样每个节点拥有唯一的路径



ACL（AccessControlLists）：记录Znode 的访问权限，即哪些人或哪些IP 可以访问本节点

Stat：包含Znode的各种元数据，比如事务ID、版本号、时间戳、大小等

Child：当前节点的子节点引用，类似于二叉树的左孩子和右孩子

ZooKeeper是为读多写少的场景所设计，Znode用于存储少量的状态和配置信息，每个节点的数据最大不超过1MB。

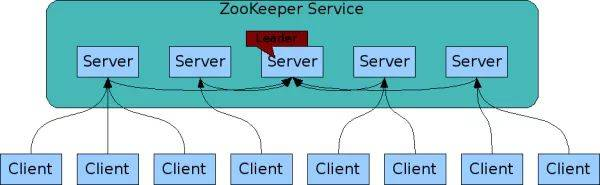
2）、操作Znode节点：提供了API和触发器机制

Create、delete、setData 、exits、getData、getChildren（三者属于读操作）。

设置watch：可以理解成注册在特定Znode上的触发器，当这个Znode发生改变的时候，即调用了create、delete、setData方法的时候，将会触发Znode上注册的对应事件，请求watch的客户端会接收到异步通知。

ZooKeeper的一致性：ZooKeeper身为分布式系统的协调服务，如果自身挂掉了，该？

为了防止单机挂掉的情况，ZooKeeper维护了一个集群。



ZooKeeper Service是一主多从结构，更新数据时，首先更新到主节点（服务器），在同步到从节点。在读取数据时，直接读取任意从节点。为了保证主从节点的数据一致性，采用ZAB协议。ZooKeeper Atomic Broadcast

协议定义的三种状态：looking（选举状态），Following(follower节点所处的状态)。Leading（Leader节点所处的状态）

最大ZXID:节点本地的最新事务编号，包含epoch（纪元）和计数两部分，epoch相当于Raft算法（共识算法）选主时候的term。

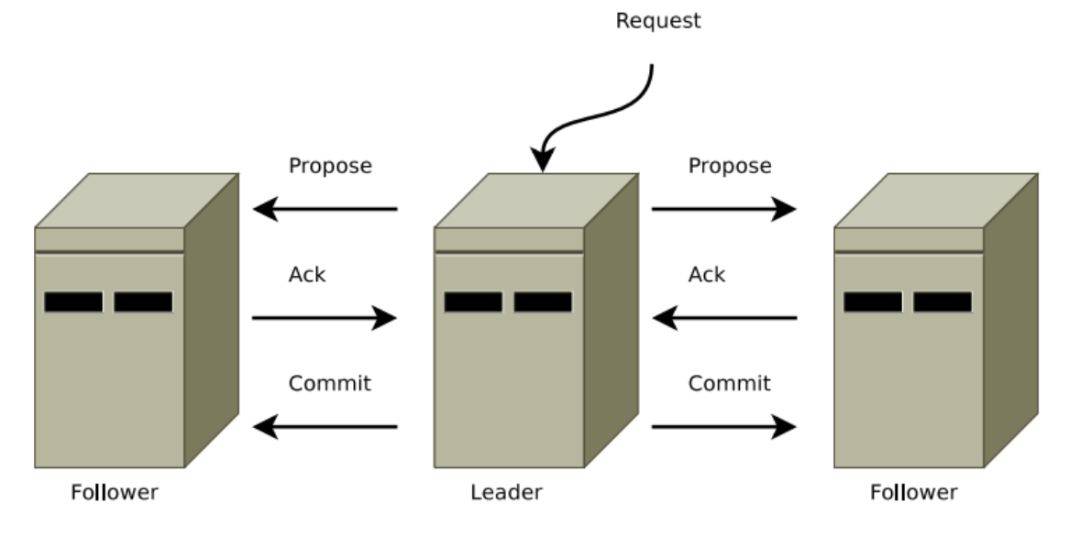
假如Zookeeper当前的主节点挂掉了，集群会进行崩溃恢复。

崩溃恢复三阶段：选举阶段，自身服务器ID和ZXID，投票给ZXID最新的节点。

发现阶段,

同步阶段

ZooKeeper更新数据：客户端发出写入数据请求给任意Follower，Follower把写入数据请求转发给Leader,leader采用二阶段提交方式，先发propose广播给Follower,follower接收到propose消息，写入日志成功后，返回ACK消息给leader，leader接收到半数以上ACK消息后，返回成功给客户端，并且广播Commit请求给Follower。



ZooKeeper的应用：

分布式锁：利用ZooKeeper的临时顺序节点，可以轻松实现分布式锁。

服务注册和发现：阿里的分布式RPC框架和Dubbo

共享配置和状态信息：R的分布式解决方案Codis，就利用了Zookeeper来存放数据路由表和 codis-proxy 节点的元信息。同时 codis-config 发起的命令都会通过 ZooKeeper 同步到各个存活的 codis-proxy。此外，Kafka、HBase、Hadoop，也都依靠Zookeeper同步节点信息，实现高可用。

线程锁和分布式锁：

1. Java的synchronize和Lock都是属于线程锁，保证同一个进程内的多线程对共享变量修改访问同步。原理是设置一个可以让所有线程访问到的标记，如synchronize是获取对象的monitor，而lock是基于AQS的volatile修饰的state。
2. 分布式锁是属于进程范畴的，而且进程可以在不同的机器上。它要保证的是多个进程对共享变量修改访问同步。分布式锁也同样需要一个可以让所有进程访问到的标记（如数据库的字段，redis的数据，zookeeper的节点）来控制同步。

分布式锁：“分布式锁是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。”

分布式锁的实现：

1. Memcached分布式锁：利用Memcached的add命令，此命令是原子性操作，只有在key不存在的情况下，才能add成功，即线程得到了锁。
2. Redis分布式锁：利用Redis的setnx命令，此命令同样是原子性操作，只有在key不存在的情况下，才能set成功但是setnx命令并不完善。
3. Zookeeper分布式锁：利用Zookeeper的顺序临时节点，来实现分布式锁和等待队列。Zookeeper设计的初衷就是实现分布式锁
4. Chubby:google公司实现的粗粒度分布式锁服务，底层利用了Paxos一致性算法

* 基于数据库的分布式锁：

1. 基于表主键唯一：加锁时向数据库插入以锁名为主键的数据，解锁时删除该数据。

优化：  
1.没有过期时间，如果获得锁的服务解锁失败，将导致其他服务无法获得锁。网上也有解决方法：增加过期时间字段，定期清除过期锁数据。  
2.重入性，增加获得锁服务的标记字段。

1. 基于版本号字段：为表结构增加版本号字段，每次获取数据将版本号获取到，下次修改数据时判断当前版本是否是该版本号。
2. 排他锁：

* 基于Redis实现分布式锁：

1. 加锁：key是锁的唯一标识，setnx（key，1）当一个线程setnx返回1时，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx’返回0，说明key已经存在，该线程抢锁失败
2. 解锁：得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行del指令，
3. 锁超时：如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放，这块资源将会永远被锁住，别的线程无法访问。

expire（key， 30）

分布式锁实现的第一版伪代码如下：

if（setnx（key，1） == 1）{

expire（key，30）

try {

do something ......

} finally {

del（key）

}

}

致命问题：

1. setnx和expire的非原子性：线程1得到锁，还没来得及expire，挂了，锁一直在，没法访问。、

解决：新版本Redis，使用set增加了可选参数，set（key，1，30，NX）。

1. del误删：线程2得到了锁，设置的超时时间是30秒，某些原因线程过了30秒没执行完，锁过期自动释放，线程3得到了锁，随后，线程2结束。执行del命令来释放锁，，但这时候锁在3上，线程3还没执行完，所以线程2释放的其实是线程3的锁。

解决：只能在释放锁之前判断当前的锁是不是自己加的锁。加锁的时候以线程ID作为value，删除时验证key对应的value是不是自己的线程ID。这样做隐含一个问题就是：判断和释放锁是两个独立操作，不是原子性。

所以这一块可以使用Lua脚本实现，这样就是原子操作了：

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

redisClient.eval(luaScript，Collections.singletonList(key), Collections.singletonList(threadId));

1. 出现并发的可能性：基于第2描述的情况，线程2没干完，锁没了。我们可以让获得锁的线程开启一个守护线程，给快要过期的锁续航。

过了29秒没执行完守护线程就会执行expire指令，为这把锁续命20秒，此后每20秒执行一次。当线程2执行完任务，就会显式关掉守护线程。

另外，如果节点一忽然断电，由于线程2和守护线程在同一个进程，守护线程也会停下。这把锁到了超时的时候，不续航就会自动释放。

* 基于ZooKeeper的分布式锁：

有序性是ZooKeeper中一个非常重要的特性，所有的更新都是全局有序的，每个更新都有一个唯一的时间戳，这个时间戳称为zxid，而读请求只会相对于更新有序，也就是读请求返回结果中会带有这个ZooKeeper最新的zxid。

ZOOkeeper节点的性质：

1. 有序节点：父节点创建子节点，指明有序，生成子节点时会根据当前子节点的数量自动添加整数序号。
2. 临时节点：客户端可以建立一个临时节点，在会话结束后或者会话超时后，ZooKeeper会自动删除该节点。
3. 事件监听：在读取数据时，我们可以同时对节点设置事件监听，当节点数据或结构变化时，ZoOKeeper会通知客户端。

分布式锁的流程：

1. 客户端连接ZooKeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-00000000,……
2. 客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听/lock的子节点变更消息，获得子节点变更通知后重复此步骤直至获得锁。
3. 执行业务代码
4. 完成业务流程后，删除对应的子节点释放锁。

首先，由于创建的是临时节点，能够保证在故障的情况下锁也能被释放。

其次，获取子节点列表与设置监听这两步操作的原子性问题。客户端a对应子节点编号为0，客户端b对应子节点编号为1，客户端b获取子节点列表时发现自己不是序号最小的，但是在设置监听器前客户端a结束了，并删除了子节点0，那客户端b是不是就陷入了永远等待？解决：zookeeper的API 中设置监听器的操作与读操作是原子执行的，即在读子节点列表时同时设置监听器，保证了不会丢失事件。

最后，算法的极大优化点：加入当前有1000个节点在等待锁，且都有事件监听，有锁的客户端释放锁时，1000个客户端都会被唤醒，这种情况被称为“羊群效应”，这种情况最好是只唤醒新的最小节点对应的客户端。解决：在设置监听事件时，每个客户端应该对刚好在它之前的子节点设置事件监听，比如序号为1的客户端监听序号为0的子节点删除消息，序号为2的监听序号为1的。。。。。。、

所以调整后的分布式锁算法流程如下：

* 客户端连接zookeeper，并在/lock下创建临时的且有序的子节点，第一个客户端对应的子节点为/lock/lock-0000000000，第二个为/lock/lock-0000000001，以此类推；
* 客户端获取/lock下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听刚好在自己之前一位的子节点删除消息，获得子节点变更通知后重复此步骤直至获得锁；
* 执行业务代码；
* 完成业务流程后，删除对应的子节点释放锁。

CAP定理：任何一个分布式系统有三个特性：一致性，可用性和分布容错性。任何分布式系统最多可以有两个属性。