Zookeeper

版本：

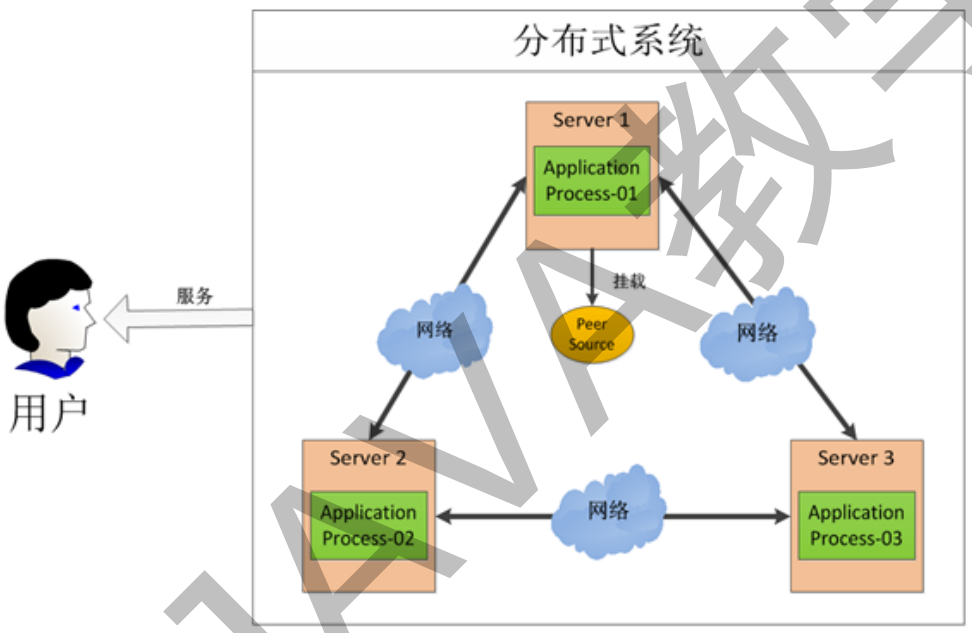
Zookeeper 3.4.6

官网下载地址

第一节 简介

1.1 Zookeeper简介

1.1.1 分布式系统



上述这张图，在这图中有三台机器，每台机器各跑一个应用程序。然后我们将这三台机器通过网络将其连接起来，构成一个系统来为用户提供服务，对用户来说这个系统的架构是透明的，他感觉不到我这个系统是一个什么样的架构。那么我们就可以把这种系统称作一个分布式系统

目前，在分布式协调技术方面做得比较好的就是Google的Chubby还有Apache的ZooKeeper他们都是分布式锁的实现者。既然有了Chubby为什么还要弄一个ZooKeeper，难道Chubby做得不够好吗？主要是Chbby是非开源的，Google自家 用。后来雅虎模仿Chubby开发出了ZooKeeper，也实现了类似的分布式锁的功能，并且将ZooKeeper作为一种开源的程序捐献给了Apache，那么这样就可以使用ZooKeeper所提供锁服务。而且在分布式领域久经考验，它的可靠性，可用性都是经过理论和实践的验证的。

1.1.2 分布式的问题

分布式应用是非常困难的，主要原因就是局部故障。一个消息通过网络在两个节点之间传递时，网络如果发生故障，发送方并不知道接收方是否接收到了这个消息。可能在网络故障前就收到了此消息，也可能没有收到，又或者可能接收方的进程死了。发送方了解情况的唯一方法就是再次连接发送方，并向他进行询问。

这就是局部故障：根本不知道操作是否失败。因此，大部分分布式应用需要一个主控、协调控制器来管理物理分布的子进程。目前，大部分应用需要开发私有的协调程序，缺乏一个通用的机制。协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器。协调服务非常容易出错，并很难从故障中恢复。例如：协调服务很容易处于竞态甚至死锁。Zookeeper的设计目的，是为了减轻分布式应用程序所承担的协调任务。

Zookeeper并不能阻止局部故障的发生，因为它们的本质是分布式系统。他当然也不会隐藏局部故障。ZooKeeper的目的就是提供一些工具集，用来建立安全处理局部故障的分布式应用。

1.1.3 Zookeeper是什么

ZooKeeper译名为“动物园管理员”。动物园里当然有好多的动物，游客可以根据动物园提供的向导图到不同的场馆观赏各种类型的动物，而不是像走在原始丛林里，心惊胆颤的被动 物所观赏。为了让各种不同的动物呆在它们应该呆的地方，而不是相互串门，或是相互厮杀，就需要动物园管理员按照动物的各种习性加以分类和管理，这样我们才能更加放心安全的观赏动物。

回到企业级应用系统中，随着信息化水平的不断提高，企业级系统变得越来越庞大臃肿，性能急剧下降，客户抱怨频频。拆分系统是目前我们可选择的解决系统可伸缩性和性能问题的唯一行之有效的方法。但是拆分系统同时也带来了系统的复杂性—各子系统不是孤立存在的，它们彼此之间需要协作和交互，这就是我们常说的分布式系统0。各个子系统就好比动物园里的动物，为了使各个子系统能正常为用户提供统一的服务，必须需要一种机制来进行协调—这就是ZooKeeper（动物园管理员）。

Zoopkeeper 提供了一套很好的分布式集群管理的机制，就是它这种基于层次型的目录树的数据结构，并对树中的节点进行有效管理，从而可以设计出多种多样的分布式的数据管理模型。

ZooKeeper是一个分布式小文件系统，并且被设计为高可用性。通过选举算法和集群复制可以避免单点故障，由于是文件系统，所以即使所有的ZooKeeper节点全部挂掉，数据也不会丢失，重启服务器之后，数据即可恢复。另外ZooKeeper的节点更新是原子的，也就是说更新不是成功就是失败。通过版本号，ZooKeeper实现了更新的乐观锁，当版本号不相符时，则表示待更新的节点已经被其他客户端提前更新了，而当前的整个更新操作将全部失败。当然所有的一切ZooKeeper已经为开发者提供了保障，我们需要做的只是调用API。与此同时，随着分布式应用的的不断深入，需要对集群管理逐步透明化监控集群和作业状态，可以充分利ZK的独有特性。

ZooKeeper性能上的特点决定了它能够用在大型的、分布式的系统当中。从可靠性方面来说，它并不会因为一个节点的错误而崩溃。除此之外，它严格的序列访问控制意味着复杂的控制原语可以应用在客户端上。ZooKeeper在一致性、可用性、容错性的保证，也是ZooKeeper的成功之处，它获得的一切成功都与它采用的协议—Zab协议是密不可分的

1.1.4 Zookeeper的应用

ZooKeeper本质上是一个分布式的小文件存储系统。原本是Apache Hadoop的一个组件，现在被拆分为一个Hadoop的独立子项目，在HBase（Hadoop的另外一个被拆分出来的子项目，用于分布式环境下的超大数据量的（DBMS）中也用到了ZooKeeper集群。
Hadoop，使用Zookeeper的事件处理确保整个集群只有一个NameNode，存储配置信息等.
HBase，使用Zookeeper的事件处理确保整个集群只有一个HMaster，察觉HRegionServer联机和宕(dàng)机，存储访问控制列表等。

在ZooKeeper诞生的地方—Yahoo!他被用作雅虎消息代理的协调和故障恢复服务。雅虎消息代理是一个高度可扩展的发布-订阅系统，他管理着成千上万台联及程序和信息控制系统。它的吞吐量标准已经达到大约每秒10000基于写操作的工作量。对于读操作的工作量来说，它的吞吐量标准还要高几倍。

1.1.5 Zookeeper的特点

分布式环境下的程序和活动为了达到协调一致目的，通常具有某些共同的特点，例如，简单性、有序性等。ZooKeeper不但在这些目标的实现上有自身特点，并且具有独特优势

简单化

ZooKeeper允许各分布式进程通过一个共享的命名空间相互联系，该命名空间类似于一个标准的层次型的文件系统：由若干注册了的数据节点构成(用Zookeeper的术语叫znode)，这些节点类似于文件和目录。典型的文件系统是基于存储设备的，文传统的文件系统主要用于存储功能，然而ZooKepper的数据是保存在内存中的。也就是说，可以获得高吞吐和低延迟。

ZooKeeper的实现非常重视高性能、高可靠，以及严格的有序访问。
高性能保证了ZooKeeper可以用于大型的分布式系统，高可靠保证了ZooKeeper不会发生单点故障，严格的顺序访问保证了客户端可以获得复杂的同步操作原语。

健壮性

就像ZooKeeper需要协调的分布式系统一样，它本身就是具有冗余结构，它构建在一系列主机之上，叫做一个”ensemble”。
构成ZooKeeper服务的各服务器之间必须相互知道，它们维护着一个状态信息的内存映像，以及在持久化存储中维护着事务日志和快照。只要大部分服务器正常工作，ZooKeeper服务就能正常工作。
客户端连接到一台ZooKeeper服务器。客户端维护这个TCP连接，通过这个连接，客户端可以发送请求、得到应答，得到监视事件以及发送心跳。如果这个连接断了，客户端可以连接到另一个ZooKeeper服务器。

有序性

ZooKeeper给每次更新附加一个数字标签，表明ZooKeeper中的事务顺序，后续操作可以利用这个顺序来完成更高层次的抽象功能，例如同步原语。

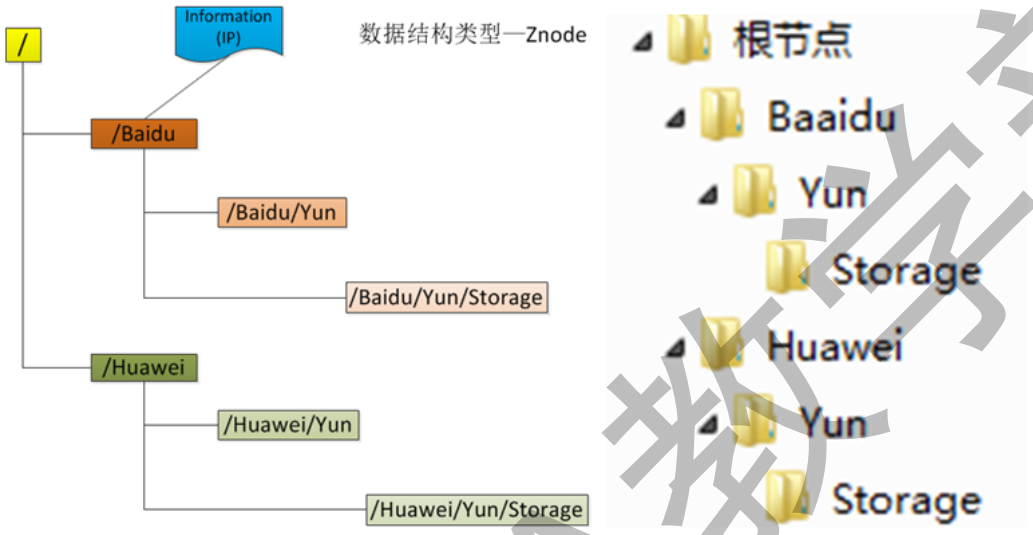
速度优势

ZooKeeper特别适合于以读为主要负荷的场合。ZooKeeper可以运行在数千台机器上，如果大部分操作为读，例如读写比例为10:1，ZooKeeper的效率会很高。

1.2 ZooKeeper数据模型

ZooKeeper所提供的服务主要是通过：数据结构+原语+watcher机制，三个部分来实现的

1.2.1 ZooKeeper数据模型Znode



从图中可以看出ZooKeeper的数据模型，在结构上和标准文件系统的非常相似，都是采用这种树形层次结构，ZooKeeper树中的每个节点被称为—Znode。和文件系统的目录树一样，ZooKeeper树中的每个节点可以拥有子节点。但也有不同之处：

引用方式

Zonde通过路径引用，如同Unix中的文件路径。路径必须是绝对的，因此他们必须由斜杠字符来开头。除此以外，他们必须是唯一的，也就是说每一个路径只有一个表示，因此这些路径不能改变。在ZooKeeper中，路径由Unicode字符串组成，并且有一些限制。字符串"/zookeeper"用以保存管理信息，比如关键配额信息。

Znode结构

ZooKeeper命名空间中的Znode，兼具文件和目录两种特点。既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分。图中的每个节点称为一个Znode。 每个Znode由3部分组成：

- stat：此为状态信息，描述该Znode的版本，权限等信息
- data：与该Znode关联的数据
- children：该Znode下的子节点

ZooKeeper虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以KB为大小单位。ZooKeeper的服务器和客户端都被设计为严格检查并限制每个Znode的数据大小至多1M，但常规使用中应该远小于此值。

数据访问

ZooKeeper中的每个节点存储的数据要被原子性的操作。也就是说读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的ACL(访问控制列表)，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。

节点类型

ZooKeeper中的节点有两种，分别为临时节点和永久节点。节点的类型在创建时即被确定，并且不能改变。
临时节点：该节点的生命周期依赖于创建它们的会话。一旦会话(Session)结束，临时节点将被自动删除，当然可以也可以手动删除。虽然每个临时的Znode都会绑定到一个客户端会话，但他们对所有的客户端还是可见的。另外，ZooKeeper的临时节点不允许拥有子节点。
永久节点：该节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

顺序节点

当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的，它的格式为"%10d"(10位数字，没有数值的数位用0补充，例如"0000000001")。当计数值大于232-1时，计数器将溢出。

观察

客户端可以在节点上设置watch，我们称之为监视器。当节点状态发生改变时(Znode的增、删、改)将会触发watch所对应的操作。当watch被触发时，ZooKeeper将会向客户端发送且仅发送一条通知，因为watch只能被触发一次，这样可以减少网络流量。

1.2.2 ZooKeeper中的时间

ZooKeeper有多种记录时间的形式，其中包含以下几个主要属性：

Zxid

致使ZooKeeper节点状态改变的每一个操作都将使节点接收到一个Zxid格式的时间戳，并且这个时间戳全局有序。也就是说，也就是说，每个对 节点的改变都将产生一个唯一的Zxid。如果Zxid1的值小于Zxid2的值，那么Zxid1所对应的事件发生在Zxid2所对应的事件之前。实际 上，ZooKeeper的每个节点维护者三个Zxid值，为别为：cZxid、mZxid、pZxid。
cZxid： 是节点的创建时间所对应的Zxid格式时间戳。
cZxid： 是节点的创建时间所对应的Zxid格式时间戳。
mZxid：是节点的修改时间所对应的Zxid格式时间戳。

实现Zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch。低32位是个递增计数。

版本号

对节点的每一个操作都将致使这个节点的版本号增加。每个节点维护着三个版本号，他们分别为：
version：节点数据版本号
cversion：子节点版本号
aversion：节点所拥有的ACL版本号

1.2.3 ZooKeeper节点属性

属性	描述
czxid	节点被创建的 zxid
mzxid	节点被修改的 zxid
ctime	节点被创建的时间
mtime	节点被修改的 zxid
version	节点被修改的版本号
cversion	节点所拥有的子节点被修改的版本号
aversion	节点的 ACL 被修改的版本号
ephemeralOwner	如果此节点为临时节点，那么他的值为这个节点拥有者的会话 ID；否则，他的值为 0
dataLength	节点数长度
numChildren	节点用的子节点长度
pzxid	最新修改的 zxid，貌似与 mzxid 重合了

1.3 ZooKeeper服务中操作

操作	描述
create	创建 Znode（父 Zonde 必须存在）
delete	删除 Znode（Znode 没有子节点）
exists	测试 Znode 是否存在，并获取他的元数据
getACL/ setACL	为 Znode 获取/设置 ACL
getChildren	获取 Znode 所有子节点的列表
getData/setData	获取/设置 Znode 的相关数据
sync	使客户端的 Znode 视图与 ZooKeeper 同步

更新ZooKeeper操作是有限制的。delete或setData必须明确要更新的Znode的版本号，我们可以调用exists找到。如果版本号不匹配，更新将会失败。
更新ZooKeeper操作是非阻塞式的。因此客户端如果失去了一个更新(由于另一个进程在同时更新这个Znode)，他可以在不阻塞其他进程执行的情况下，选择重新尝试或进行其他操作。
尽管ZooKeeper可以被看做是一个文件系统，但是处于便利，摒弃了一些文件系统地操作原语。因为文件非常的小并且使整体读写的，所以不需要打开、关闭或是寻地的操作。

1.4 Watch触发器

1.4.1 watch概述

ZooKeeper可以为所有的读操作设置watch，这些读操作包括：exists()、getChildren()及getData()。watch事件是一次性的触发器，当watch的对象状态发生改变时，将会触发此对象上watch所对应的事件。watch事件将被异步地发送给客户端，并且ZooKeeper为watch机制提供了有序的一致性保证。理论上，客户端接收watch事件的时间要快于其看到watch对象状态变化的时间。

1.4.2 watch类型

ZooKeeper所管理的watch可以分为两类：

```
数据watch(data  watches)：getData和exists负责设置数据watch
孩子watch(child watches)：getChildren负责设置孩子watch
```

可以通过操作**返回的数据**来设置不同的watch：

```
getData和exists：返回关于节点的数据信息
getChildren：返回孩子列表
```

返回

```
一个成功的setData操作将触发Znode的数据watch
一个成功的create操作将触发Znode的数据watch以及孩子watch
一个成功的delete操作将触发Znode的数据watch以及孩子watch
```

1.4.3 watch注册与触发

设置 watch	watch 触发器				
	create		delete		setData
	Znode	child	Znode	child	Znode
exists	NodeCreated		NodeDeleted		NodeDataChanged
getdata			NodeDeleted		NodeDataChanged
getChildren		NodeChildrenChanged	NodeDeleted	NodeDeletedChanged	

```
exists操作上的watch，在被监视的Znode创建、删除或数据更新时被触发。
getData操作上的watch，在被监视的Znode删除或数据更新时被触发。在被创建时不能被触发，因为只有Znode一定存在，getData操作才会成功。
getChildren操作上的watch，在被监视的Znode的子节点创建或删除，或是这个Znode自身被删除时被触发。可以通过查看watch事件类型来区分是Znode，还是他的子节点被删除：NodeDelete表示Znode被删除NodeDeletedChanged表示子节点被删除。
```

```
Watch由客户端所连接的ZooKeeper服务器在本地维护，因此watch可以非常容易地设置、管理和分派。当客户端连接到一个新的服务器 时，任何的会话事件都将可能触发watch。另外，当从服务器断开连接的时候，watch将不会被接收。但是，当一个客户端重新建立连接的时候，任何先前 注册过的watch都会被重新注册。
```

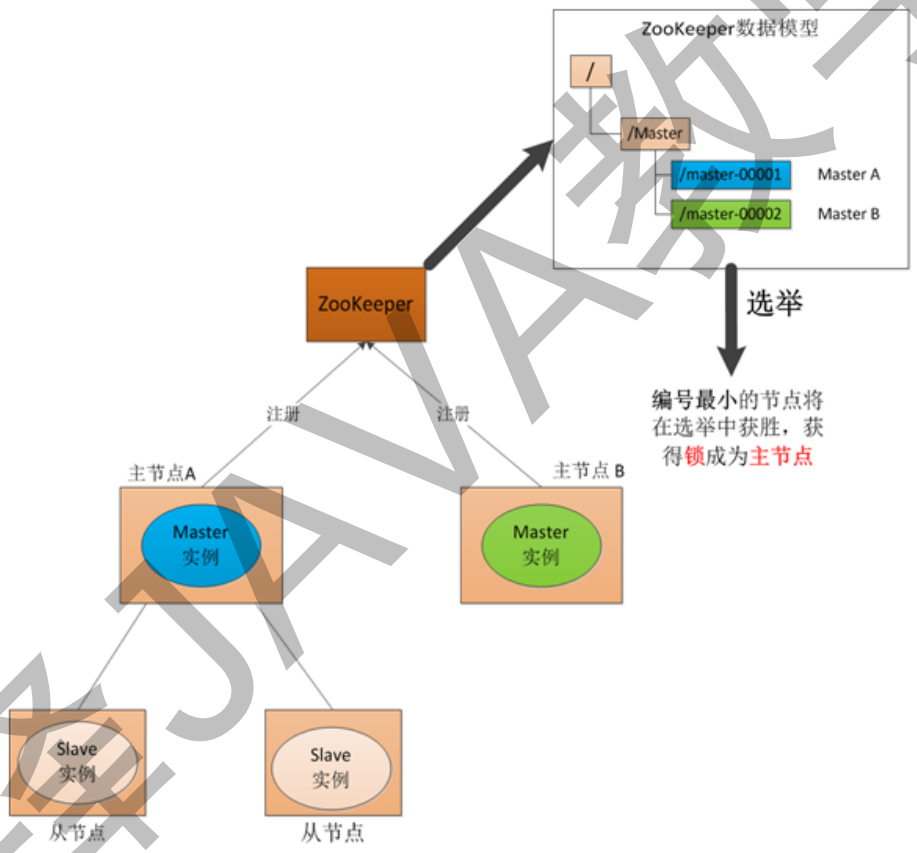
```
Zookeeper的watch实际上要处理两类事件：
连接状态事件(type=None，path=null)
        这类事件不需要注册，也不需要我们连续触发，我们只要处理就行了。
节点事件
        节点的建立，删除，数据的修改。它是one time trigger，我们需要不停的注册触发，还可能发生事件丢失的情况。
```

节点事件的触发，通过函数exists，getData或getChildren来处理这类函数，有双重作用：
注册触发事件
函数本身的功能
函数的本身的功能又可以用异步的回调函数来实现,重载processResult()过程中处理函数本身的的功能

1.5 Zookeeper分布式应用

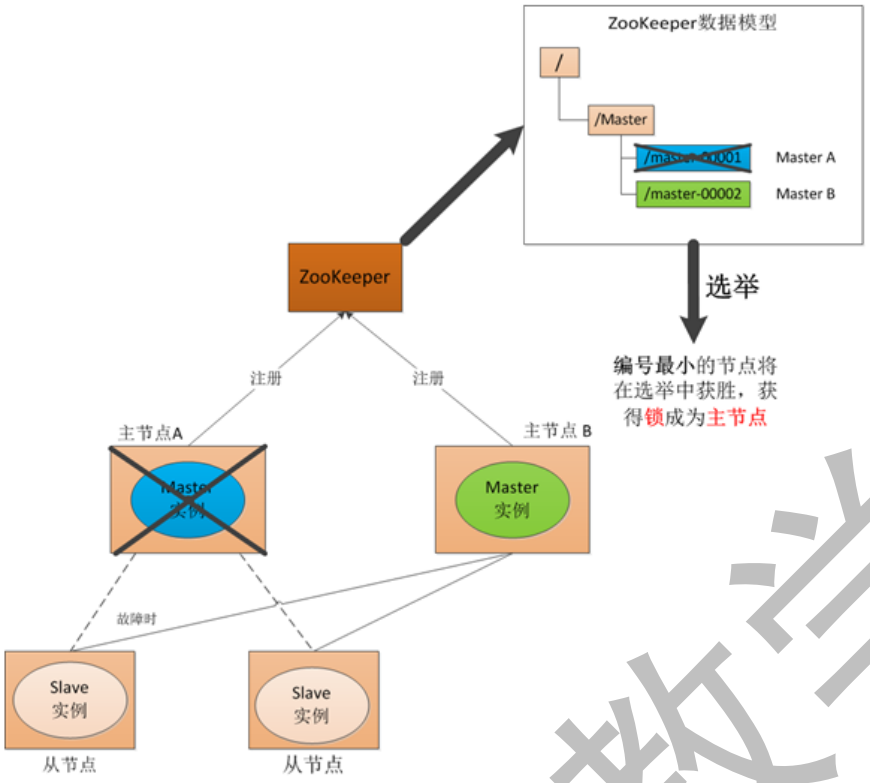
1.5.1 Master启动

使用Zookeeper启动了两个主节点，"主节点-A"和"主节点-B"他们启动以后，都向ZooKeeper去注册一个节点。我们 假设"主节点-A"锁注册地节点是"master-00001"，"主节点-B"注册的节点是"master-00002"，注册完以后进行选举，编号最 小的节点将在选举中获胜获得锁成为主节点，也就是我们的"主节点-A"将会获得锁成为主节点，然后"主节点-B"将被阻塞成为一个备用节点。那么，通过这 种方式就完成了对两个Master进程的调度。

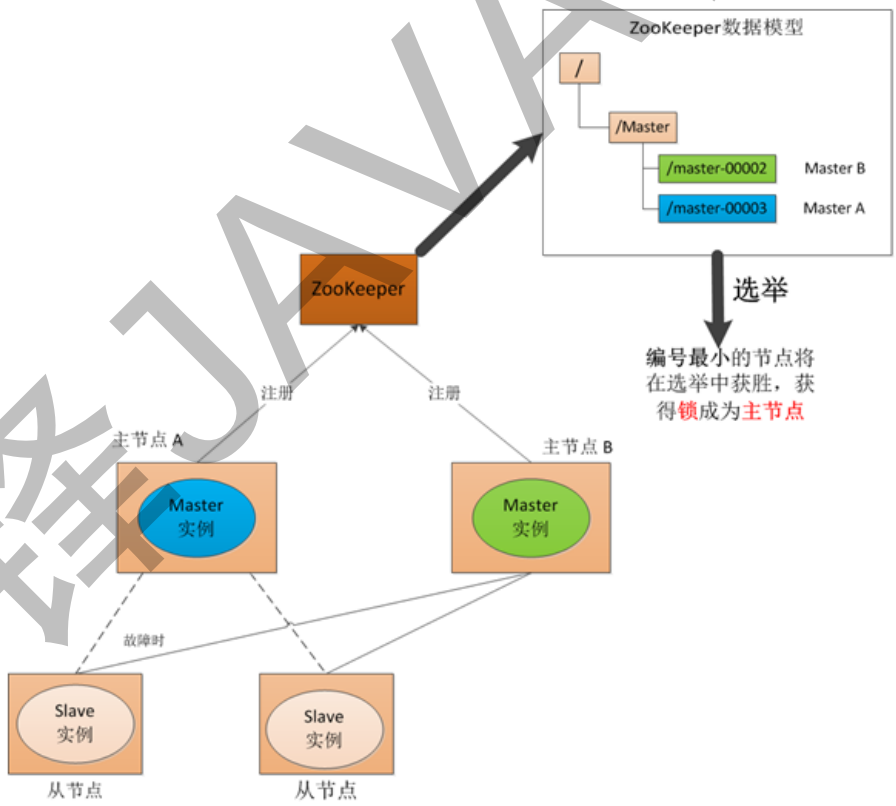


1.5.2 Master故障

如果"主节点-A"挂了，这时候他所注册的节点将被自动删除，ZooKeeper会自动感知节点的变化，然后再次发出选举，这时候"主节点-B"将在选举中获胜，替代"主节点-A"成为主节点。



1.5.3 Master 恢复



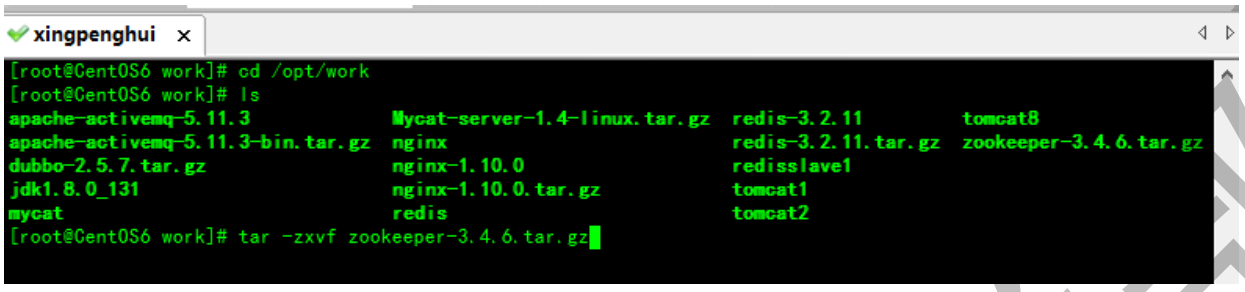
如果主节点恢复了，他会再次向ZooKeeper注册一个节点，这时候他注册的节点将会是"master-00003"，ZooKeeper会感知节点的变化再次发动选举，这时候"主节点-B"在选举中会再次获胜继续担任"主节点"，"主节点-A"会担任备用节点

第二节 安装与配置

2.1 上传和解压

点击官网下载地址下载指定版本的Zookeeper，然后上传到服务器

存储目录还是在/opt/work下

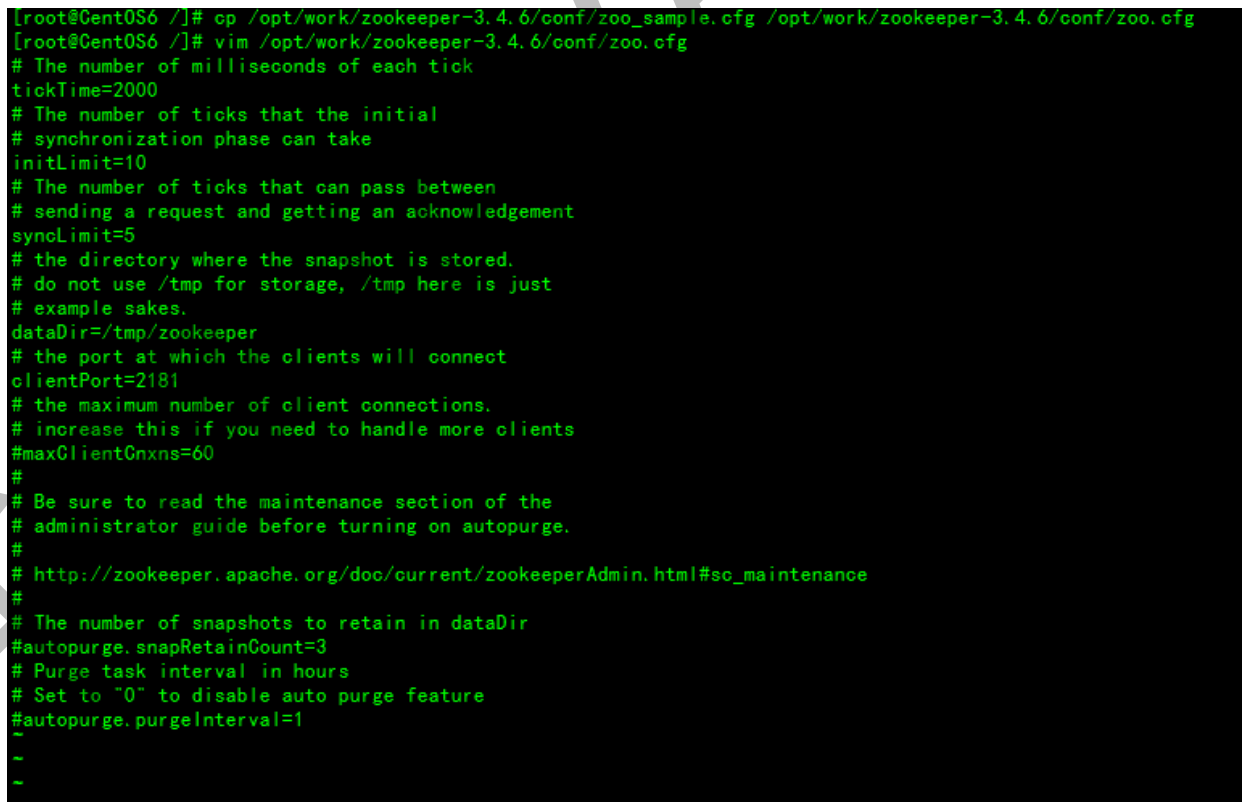


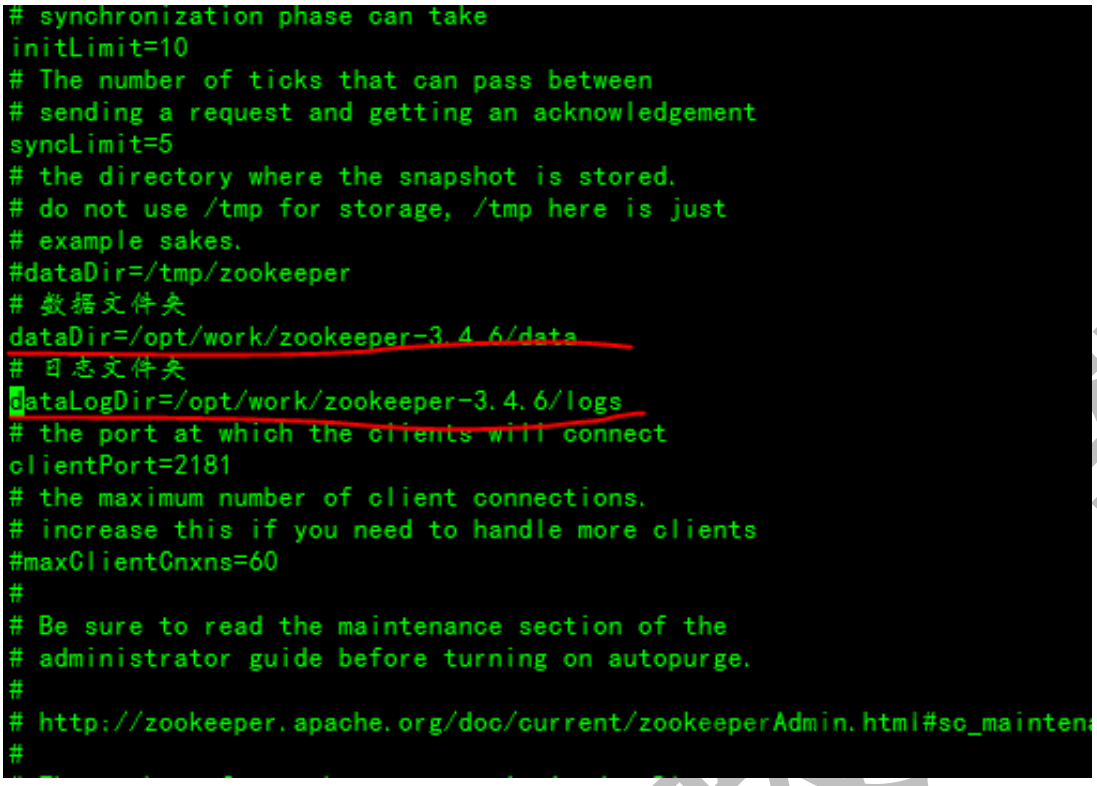
2.2 配置和启动

进入到配置文件目录进行配置

命令

```
cp /opt/work/zookeeper-3.4.6/conf/zoo_sample.cfg /opt/work/zookeeper-
3.4.6/conf/zoo.cfg
vim /opt/work/zookeeper-3.4.6/conf/zoo.cfg   指定数据目录和日志目录
mkdir /opt/work/zookeeper-3.4.6/logs    创建日志目录
```





让Zookeeper添加到环境变量

命令:

```
vim /etc/profile  编辑
在末尾新增:
 export ZOOKEEPER_HOME=/opt/work/zookeeper-3.4.6/
 export PATH=$ZOOKEEPER_HOME/bin:$PATH
 export PATH

source /etc/profile  环境变量生效

zkServer.sh start      启动
zkServer.sh status     查看状态
zkServer.sh restart 重启
zkServer.sh stop    停止
```

```
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done

unset i
unset -f pathmunge
export JAVA_HOME=/opt/work/jdk1.8.0_131
export JRE_HOME=/opt/work/jdk1.8.0_131/jre
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar:$JRE_HOME/lib
export PATH=$PATH:$JAVA_HOME/bin

 # idea - zookeeper-3.4.6 config start

    export ZOOKEEPER_HOME=/opt/work/zookeeper-3.4.6/
    export PATH=$ZOOKEEPER_HOME/bin:$PATH
    export PATH

    # idea - zookeeper-3.4.6 config start
-- INSERT --
```

xingpenghui ×

```
[root@CentOS6 /]# zkServer.sh start
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@CentOS6 /]# zkServer.sh status
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: standalone
[root@CentOS6 /]# zkServer.sh restart
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@CentOS6 /]# zkServer.sh stop
JMX enabled by default
Using config: /opt/work/zookeeper-3.4.6/bin/../conf/zoo.cfg
Stopping zookeeper ... STOPPED
[root@CentOS6 /]#
```