# Zookeeper概述

Zookeeper是源代码开放的分布式协调服务，是一个高性能的分布式数据一致性的解决方案，它将那些复杂的，容易出错的分布式一致性服务封装起来。用户可以通过调用Zookeeper提供的接口来解决一些分布式应用中的实际问题。

分布式是指：把一个计算任务分解成若干个计算单元，并分派到不同的计算机中去执行，最终汇总计算结果的过程。

典型应用场景

（1）数据发布/订阅

数据的发布与订阅，顾名思义就是一方把数据发布出来，另一方通过某种手段获取。

通常数据发布与订阅有两种模式：推模式和拉模式，推模式一般是服务器主动往客户端推送信息，拉模式是客户端主动去服务端请求目标数据（通常采用定时轮询的方式）

Zookeeper采用两种方式互相结合：发布者将数据发布到Zookeeper集群节点上，订阅者通过一定的方法告诉Zookeeper服务器，自己对哪个节点的数据感兴趣，那么在服务端数据发生变化时，就会通知客户端去获取这些信息。

（2）负载均衡

　首先在服务端启动的时候，把自己在zookeeper服务器上注册成临时节点。在服务端出问题时，节点会自动的从zookeeper上删除，使zookeeper服务器上的列表时刻可用

　客户端在需要访问服务器的时候首先会去Zookeeper获得所有可用的服务端的连接信息,通过一定的策略选择一个与之建立连接。

　当客户端发现连接不可用时，会再次从zookeeper上获取可用的服务端连接，并同时删除之前获取的连接列表。

　 （3）命名服务

自增长id仅适合在单表单库中使用，uuid适合在分布式系统中使用但由于id没有规律难以理解。而ZK提供接口可以用来获取一个顺序增长的，可以在集群环境下使用的id。

　 （4）分布式协调，通知，心跳服务

　在分布式服务系统中，我们常常需要知道哪个服务是可用的，哪个服务是不可用的，传统的方式是通过ping主机来实现的，ping得200的结果说明说明该服务是OK的。

　而在使用 zookeeper时，可以将所有的服务都注册成一个临时节点，我们判断一个服务是否可用，只需要判断这个节点是否在zookeeper集群中存在就可以了，不需要直接去连接和ping服务所在主机，减少系统的复杂度和对服务主机的压力。

## 安装

配置java环境变量

解压并修改位置

tar -zxvf zookeeper-3.4.14.tar.gz -C /usr/local

cd /usr/local/

mv zookeeper-3.4.14/ zookeeper

5，修改配置文件

#打开zk的配置文件目录

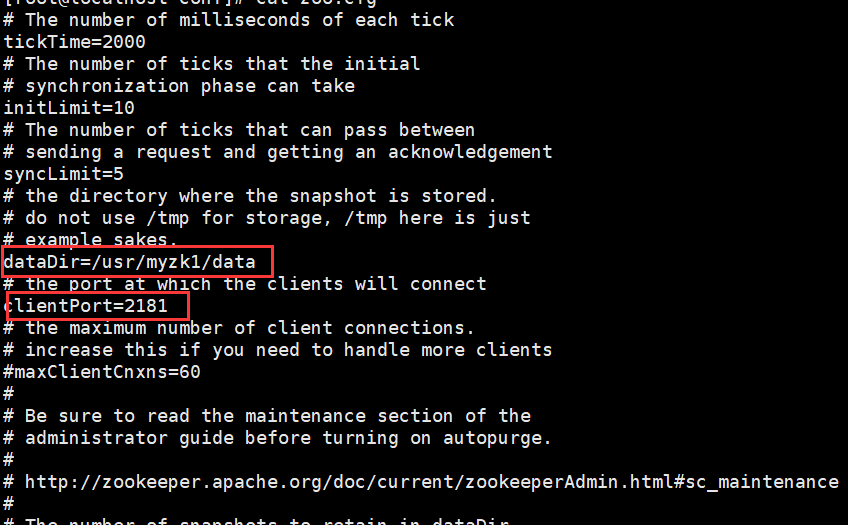
cd /usr/local/zookeeper/conf

#修改zoo\_sample.cfg 为zoo.cfg [一定要改]

mv zoo\_sample.cfg zoo.cfg

#编辑zoo.cfg

vi zoo.cfg



## 启动关闭连接



#启动

/usr/local/zookeeper/bin/zkServer.sh start

#停止

/usr/local/zookeeper/bin/zkServer.sh stop

查看zk的运行状态

/usr/local/zookeeper/bin/zkServer.sh status

连接

/usr/local/zookeeper/bin/zkCli.sh

# 文件系统

每个子目录项都被称作为znode(目录节点)，和文件系统一样，我们能够自由的增加、删除znode，唯一的不同在于znode本身是可以存储数据的(文件夹本身也能存数据)

## 四种类型的znode：

**PERSISTENT-持久化目录节点**     客户端与zookeeper断开连接后，该节点依旧存在

**PERSISTENT\_SEQUENTIAL-持久化顺序编号目录节点** 客户端与zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

**EPHEMERAL-临时目录节点**  客户端与zookeeper断开连接后，该节点被删除

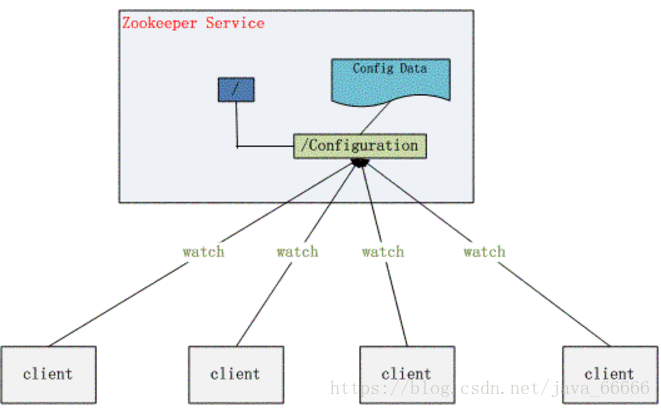
**EPHEMERAL\_SEQUENTIAL-临时顺序编号目录节点**  客户端与zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号

# 监听机制

客户端注册监听它关心的目录节点，当目录节点发生变化（数据改变、被删除、子目录节点增加删除）时，zookeeper会通知客户端。

zookeeper可以实现诸如分布式应用配置管理、统一命名服务、状态同步服务、集群管理等功能，我们这里拿比较简单的分布式应用配置管理为例来说明。

假设我们的程序是分布式部署在多台机器上，如果我们要改变程序的配置文件，需要逐台机器去修改，非常麻烦，现在把这些配置全部放到zookeeper上去，保存在 zookeeper 的某个目录节点中，然后所有相关应用程序对这个目录节点进行监听，一旦配置信息发生变化，每个应用程序就会收到 zookeeper 的通知，然后从 zookeeper 获取新的配置信息应用到系统中。



# Zookeeper单机

create [-s] [-e] path data acl 创建节点

-s表示顺序节点 会在节点后面额外加上数字序号

-e表示临时节点。默认创建持久节点

path是节点路径，data是节点数据，acl是用来进行权限控制的。

get /节点 获取指定节点的内容和属性信息

get /节点 watch **节点监控**

set path data 更新节点内容

delete /节点 **删除节点**

目录节点存在子节点时，无法删除,要用rmr path 删除当前节点及子节点

# Zookeeper集群搭建及使用

## 配置参数说明

1．tickTime =2000：通信心跳数，Zookeeper服务器与客户端心跳时间，单位毫秒

Zookeeper使用的基本时间，服务器之间或客户端与服务器之间维持心跳的时间间隔，也就是每个tickTime时间就会发送一个心跳，时间单位为毫秒。

它用于心跳机制，并且设置最小的session超时时间为两倍心跳时间。(session的最小超时时间是2\*tickTime)

2．initLimit =10：LF初始通信时限

Follower跟随者服务器与Leader领导者服务器之间初始连接时能容忍的最多心跳数，用它来限定集群中的Zookeeper服务器连接到Leader的时限。

3．syncLimit =5：LF同步通信时限

Leader与Follower最大响应时间单位，假如响应超过syncLimit \* tickTime，Leader认为Follwer死掉，从服务器列表中删除Follwer。

4．dataDir：数据文件目录+数据持久化路径

主要用于保存Zookeeper中的数据。

5．clientPort =2181：客户端连接端口

监听客户端连接的端口。

## 选举机制（Paxos算法）

Zookeeper虽然在配置文件中并没有指定Master和Slave。但是Zookeeper工作时，通过内部的选举机制临时产生Leader。

两个机制:

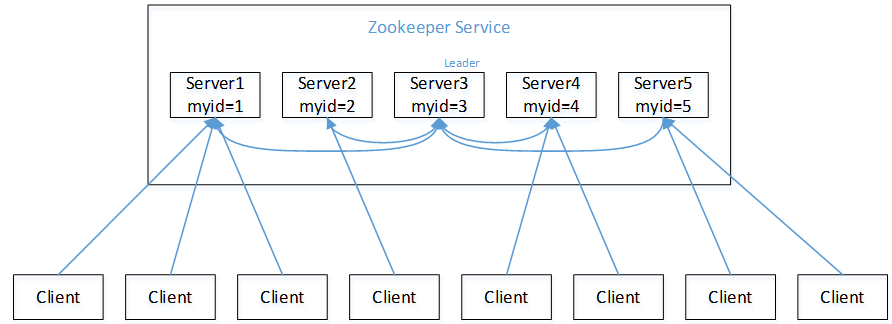
初始阶段,所有server都投给自己

半数机制：半数以上机器存活，集群可用。需要奇数台服务器。

id机制:未决定leader的情况下,优先投票大的id

以一个简单的例子来说明整个选举的过程

假设有五台服务器组成的Zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。



（1）服务器1启动，此时只有一台服务器，它发出去的报文没有响应，所以它的选举状态一直是LOOKING状态

（2）服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3)，所以服务器1、2还是继续保持LOOKING状态。

（3）服务器3启动

第一轮: 1->1 2->2 3->3

第二轮: 1->3 2->3 3->3

3获得超过半数的票,成为leader

（4）服务器4启动，虽然id最大，但是已经有半数以上选举了服务器3，所以它只能following

（5）服务器5启动，同4一样following

但是反过来从5-1开始启动

（1/2）服务器5以及4启动，不满足半数,选举状态一直是LOOKING状态

（3）服务器3启动

第一轮: 5->5 4->5 3->5

5获得超过半数的票,成为leader

从id较大的server开始启动,一轮就完成了投票

而投票是极其消耗时间的,可以通过改变server的启动顺序来加速投票

app\_id=2021001182652144&method=alipay.user.info.share&charset=utf-8&sign\_type=RSA2&timestamp=2020-07-24 09:34:11&version=1.0&biz\_content={"name":["aa"]}&sign=RDeRoP11O9uuy1xBBOZeHUJhvZQQDiZtAZyPpRA+AFAS5MfS30UKWqHekEvJ4TzHqCklj5t+jcVGp65KN+eEpX+WYYOni5XtTa5FjrWvc8tzhtWK32D7UTQ5pvBYRkF/UV4o1t7jmGplvk6EL6ouP3ZOSxrSwLMPAdlFhpZ6zYPbTQ3FHH3nM44XZECI662NNiGlQRO5qnBMURvAClLFAACL+SgZgfESaUkx0y9lMKDt9sOsQGpywUXZckcvCj2ZyPQ0Gj+APB927w09qKKKy4IpfeCkO/ydRXXt1YE0fLRkMddxyrfR9nC2VW4WBWAhqTtQwj71VwcoQoiENA5gbw==

## 集群搭建

|  |  |  |
| --- | --- | --- |
| 机器编号 | Ip 地址 | 端口 |
| Zk-1 | 192.168.186.128 | 2181 |
| Zk-2 | 192.168.186.128 | 2182 |
| Zk-3 | 192.168.186.128 | 2183 |

复制3份zookeeper,每个创建2个目录,1个文件

创建data数据目录

mkdir /usr/local/zookeeper-cluster/zk1/data

创建myid文件

vim /usr/local/zookeeper-cluster/zk1/data/myid

1 (id号)

创建log日志目录

mkdir /usr/local/zookeeper-cluster/zk1/log

修改配置文件

# zookeeper 的数据目录，三个zk 的数据目录需要不同才行

dataDir=/usr/local/zookeeper-cluster/zk1/data

# zookeeper的日志目录

dataLogDir=/usr/local/zookeeper-cluster/zk1/log

# zookeeper 的client的端口号

clientPort=2181

# 集群额外的配置文件

#第几个服务器（1，2，3来自数据目录的一个myid文件，该文件里面保存着当前集群的标识（1，2，3））

# 第一个端口：集群内部数据复制的端口 第二个端口代表：选举端口

端口可以重复,但复制和选举不能相同端口

server.1=192.168.186.128:2888:3888

server.2=192.168.186.128:2889:3889

server.3=192.168.186.128:2887:3887

# 新角色Observer

之前的ZooKeeper版本只有leader和follower。后来引入种新角色Observer，除了没有投票功能外，其它和follower一样

# ZooKeeper如何处理请求？

ZooKeeper集群中的每个server都能为客户端提供读、写服务。

对于客户端的读请求，server会直接从它本地的内存数据库中取出数据返回给客户端，这个过程不涉及其它任何操作，也不会联系leader。

对于客户端的写请求，因为写操作会修改znode的数据、状态，所以必须要在ZooKeeper集群中进行协调。处理过程如下：

收到写请求的那个server，首先将写请求发送给leader。

leader收到来自follower(或observer)的写请求后，首先计算这次写操作之后的状态，然后将这个写请求转换成带有各种状态的事务(如版本号、zxid等等)。

leader将这个事务以提议的方式广播出去(即发送proposal)。

所有follower收到proposal后，对这个提议进行投票，投票完成后返回ack给leader。follower的投票只有两种方式：(1)确认这次提议表示同意；(2)丢弃这次提议表示不同意。

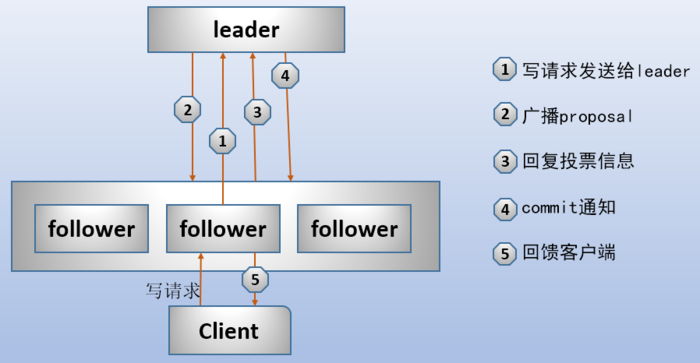
leader收集投票结果，只要投票数量达到了大多数的要求(例如，5个节点的集群，3个或3个以上的节点才算大多数)，这次提议就通过。

提议通过后，leader向所有server发送一个提交通知。

所有节点将这次事务写入事务日志，并进行提交。

提交后，收到写请求的那个server向客户端返回成功信息。

下面是ZooKeeper集群处理写请求过程的一个简图：



当ZooKeeper集群中follower的数量很多时，投票过程会成为一个性能瓶颈，为了解决投票造成的压力，于是出现了observer角色。

observer角色不参与投票，它只是投票结果的"听众"，除此之外，它和follower完全一样，例如能接受读、写请求。就这一个特点，让整个ZooKeeper集群性能大大改善。

和follower一样，当observer收到客户端的读请求时，会直接从内存数据库中取出数据返回给客户端。

对于写请求，当写请求发送到某server上后，无论这个节点是follower还是observer，都会将它发送给leader。然后leader组织投票过程，所有server都收到这个proposal(包括observer，因为proposal是广播出去的)，但是leader和follower以及observer通过配置文件，都知道自己是不是observer以及谁是observer。自己是observer的server不参与投票。当leader收集完投票后，将那些observer的server去掉，在剩下的server中计算大多数，如果投票结果达到了大多数，这次写事务就成功，于是leader通知所有的节点(包括observer)，让它们将事务写入事务日志，并提交。

3.Observer的优点

observer角色除了减轻了投票的压力，还带来了几个额外的优点。

1.提高了伸缩性。

伸缩性指的是通过添加服务器来负载请求，从而提高整个集群处理请求的能力。也就是"一头牛拉不动了，找更多牛来拉"。

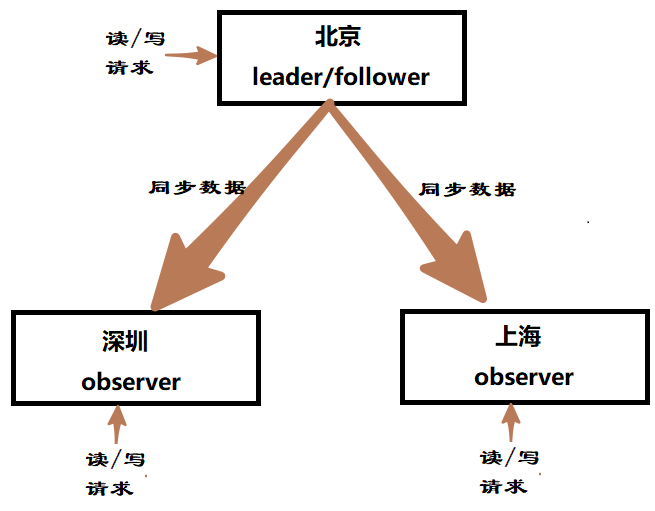
在出现Observer之前，ZooKeeper集群的伸缩性由follower来实现。虽然对于读写操作来说，follower是"无状态"的，这使得添加新的follower到集群(或者从集群中减少follower)很方便，能提高ZooKeeper集群负载能力。但是，对于投票来说，follower是有状态的，增、减follower的数量，都直接影响投票结果，特别是follower的数量越多，投票过程的性能就越差。

而observer无论是读写请求还是投票，都是无状态的，增、减observer的数量不会影响投票结果。这样就可以让一部分server作为follower参与投票，另一部分作为observer单纯地提供读写服务。这使得ZooKeeper的伸缩性大大提高。

2.部署跨地区的ZooKeeper数据中心更方便。

observer能直接从本地内存数据库中取出数据来响应读请求，所以提高了读的吞吐量。对于写请求，虽然它要发送给leader并接受leader的通知，但相比于投票过程中传递的信息，它的数据量很小，所以即使在广域网也能有很好的性能。

实际上，很多跨机房、跨地区的数据中心就是通过observer来实现的。



4.如何配置Observer？

要配置observer，只需稍微修改一下配置文件即可。

首先，在想要成为observer的配置文件中，加上下面一行：

peerType=observer

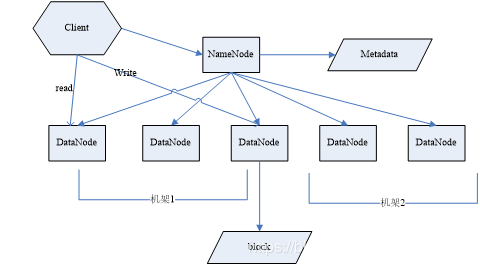
这表示这个server以observer角色运行，即不参与投票。

再在所有 server的配置文件中，修改server.X配置项，在那些observer的节点上加上:observer后缀。

例如，server.1对应的server要作为observer：

server.1=IP:2181:3181:observer

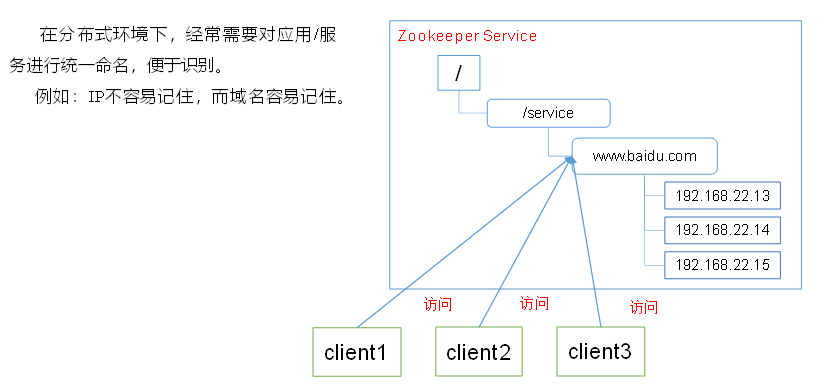
这样配置后，ZooKeeper集群中的所有服务器节点都知道哪些节点扮演的是observer角色。



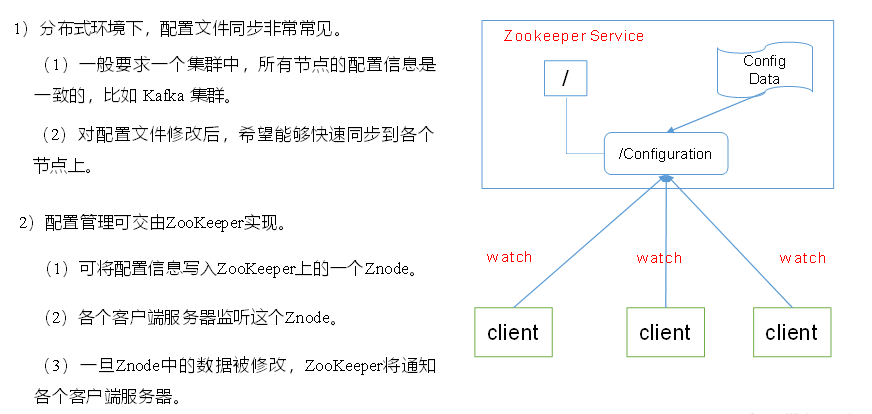
# 应用场景

统一命名服务、统一配置管理、统一集群管理、服务器节点动态上下线、软负载均衡等。

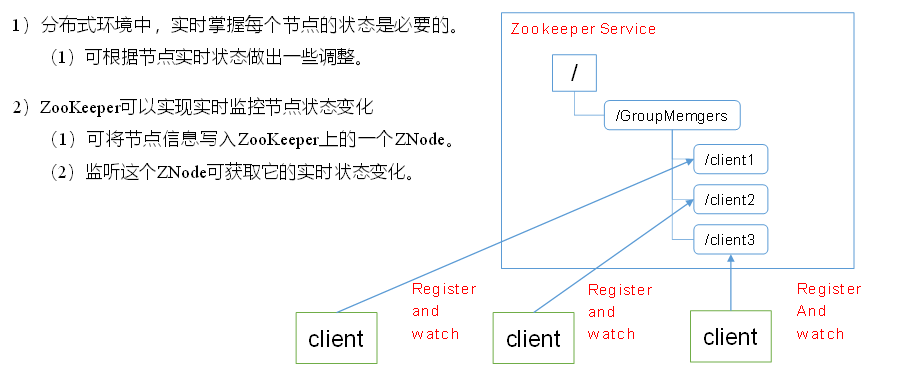
## 统一命名服务



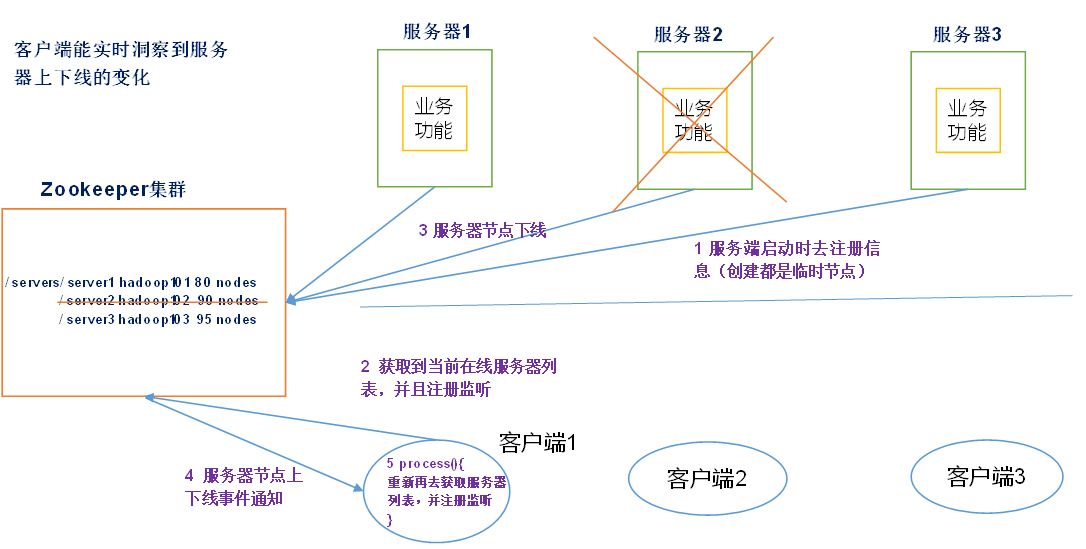
## 统一配置管理



**4，统一集群管理**



## 服务动态上线



# java操作zookeeper

<!-- zkclient -->

<dependency>

<groupId>com.101tec</groupId>

<artifactId>zkclient</artifactId>

<version>0.11</version>

</dependency>

创建create

修改writeData

查询readData

删除delete

|  |
| --- |
| **public** **class** TestCRUD {  **private** **static** **final** String ***SERVERSTRING*** = "192.168.33.115:2181,192.168.33.115:2182,192.168.33.115:2183";  **private** **static** ZkClient *zkClient*=**null**;  **static** {*zkClient*=**new** ZkClient(***SERVERSTRING***,10000,10000);}  **public** **static** **void** main(String[] args) {}  } |

## 节点监听(需创建序列化的类)

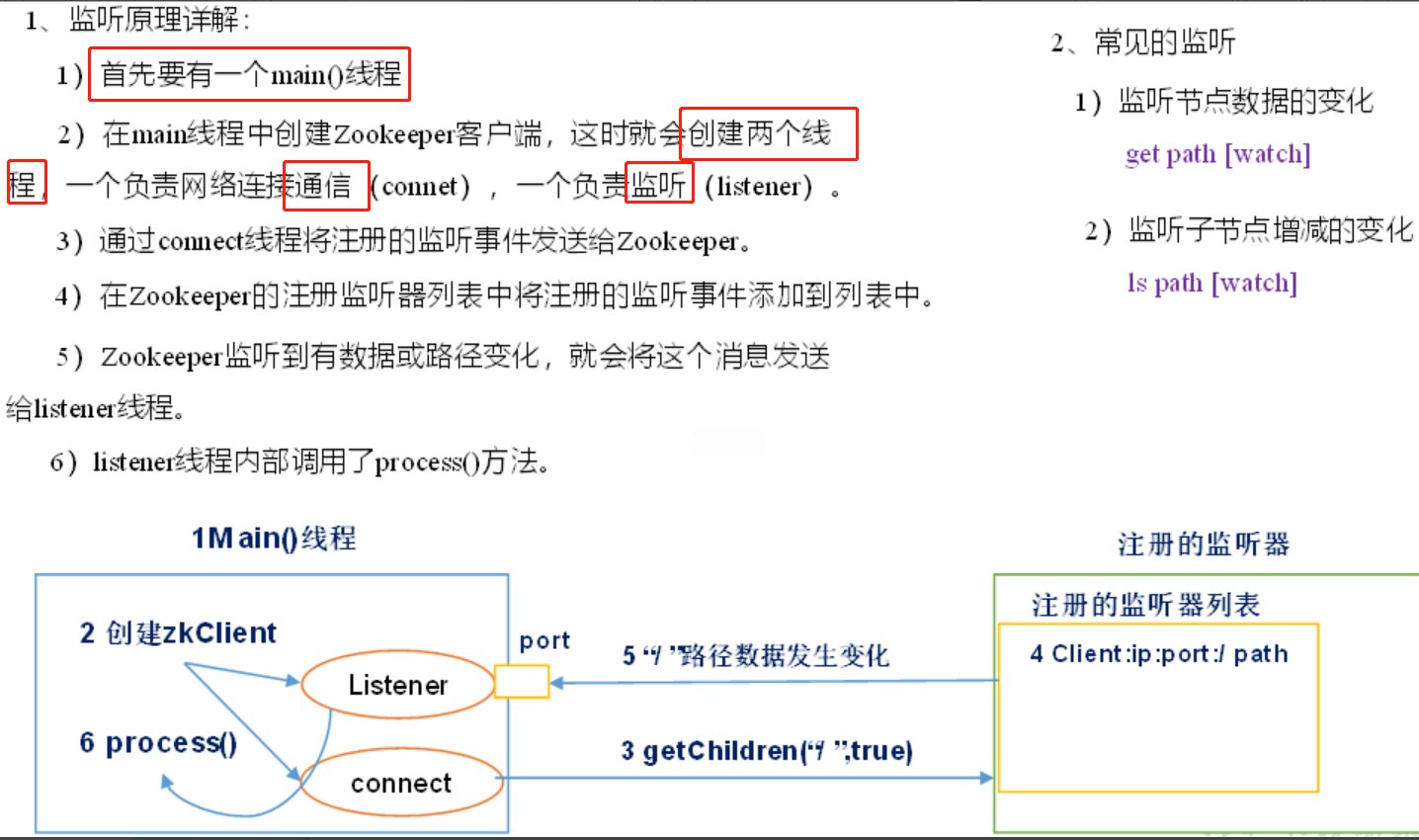
### 序列化

|  |
| --- |
| **public** **class** CustomerSerializer **implements** ZkSerializer {  /\*\* default utf 8 \*/  **private** String charset = "UTF-8";    **public** CustomerSerializer() {  }  **public** CustomerSerializer(String charset) {  **this**.charset = charset;  }  /\*\*  \* 序列化  \*/  @Override  **public** **byte**[] serialize(Object data) **throws** ZkMarshallingError {  **try** {  **byte**[] bytes = String.*valueOf*(data).getBytes(charset);  **return** bytes;  } **catch** (UnsupportedEncodingException e) {  **throw** **new** ZkMarshallingError("Wrong Charset:" + charset);  }  }  /\*\*  \* 反序列化  \*/  @Override  **public** Object deserialize(**byte**[] bytes) **throws** ZkMarshallingError {  String result = **null**;  **try** {  result = **new** String(bytes, charset);  } **catch** (UnsupportedEncodingException e) {  **throw** **new** ZkMarshallingError("Wrong Charset:" + charset);  }  **return** result;  }  } |

### 监听

|  |
| --- |
| **public** **class** TestWatch {  **private** **static** **final** String ***SERVERSTRING*** = "192.168.33.115:2181,192.168.33.115:2182,192.168.33.115:2183";  **private** **static** ZkClient *zkClient*=**null**;    **static** {  *zkClient*=**new** ZkClient(***SERVERSTRING***,10000,10000,**new** CustomerSerializer());  }  **public** **static** **void** main(String[] args) **throws** IOException {    //监听当前节点的改变和删除事件  *zkClient*.subscribeDataChanges("/sanguo", **new** IZkDataListener() {    /\*\*  \* 当dataPath被删除时回调用的方法  \*/  @Override  **public** **void** handleDataDeleted(String dataPath) **throws** Exception {  System.***out***.println("handleDataDeleted:"+dataPath);  }    /\*\*  \* 当dataPath这个节点的数据发生改变时回调用  \* data:改变这之后的值  \*/  @Override  **public** **void** handleDataChange(String dataPath, Object data) **throws** Exception {  System.***out***.println("handleDataChange:"+dataPath+" "+data);  }  });    //监听子节的变化  *zkClient*.subscribeChildChanges("/sanguo", **new** IZkChildListener() {    /\*\*  \* parentPath父节点路径  \* currentChilds 当前父节点下面的所有子节点  \*/  @Override  **public** **void** handleChildChange(String parentPath, List<String> currentChilds) **throws** Exception {    }  });  System.***out***.println("监听启动成功");  System.***in***.read();//不让程序结束  }  } |

# 监听器原理



# ZooKeeper和CAP理论及一致性原则

## CAP理论

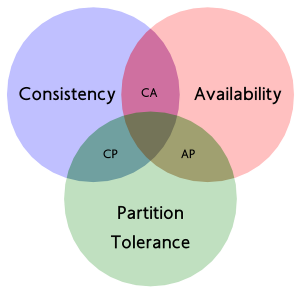
CAP理论告诉我们，一个分布式系统不可能同时满足以下三种

一致性（C:Consistency）

可用性（A:Available）

分区容错性（P:Partition Tolerance）

这三个基本需求，最多只能同时满足其中的两项，因为P是必须的,因此往往选择就在CP或者AP中。



### 一致性（C:Consistency）

分布式中一致性是指数据在多个副本之间是否能够保持数据一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一致的状态。

### 可用性（A:Available）横向扩容

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。“有效的时间内”是指，对于用户的一个操作请求，系统必须能够在指定的时间（即响应时间）内返回对应的处理结果，如果超过了这个时间范围，那么系统就被认为是不可用的。

“返回结果”是可用性的另一个非常重要的指标，它要求系统在完成对用户请求的处理后，返回一个正常的响应结果。正常的响应结果通常能够明确的反映出对请求的处理结果，即成功或失败，而不是一个让用户感到困惑的返回结果。

### 分区容错性（P:Partition Tolerance）

分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

网络分区是指在分布式系统中，不同的节点分布在不同的子网络中，由于一些特殊的原因导致这些子网络之间出现网络不连通的状况，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干个孤立的区域。需要注意的是，组成一个分布式系统的每个节点的加入与退出都可以看作是一个特殊的网络分区。

## 一致性的分类

一致性是指从系统外部读取系统内部的数据时，在一定约束条件下相同，即数据变动在系统内部各节点应该是同步的。根据一致性的强弱程度不同，可以将一致性的分类为如下几种：

**强一致性：（strong consistency）**。任何时刻，任何用户都能读取到最近一次成功更新的数据。

**单调一致性：（monotonic consistency）**。任何时刻，任何用户一旦读到某个数据在某次更新后的值，那么就不会再读到比这个值更旧的值。也就是说，可获取的数据顺序必是单调递增的。

**会话一致性：（session consistency）。**任何用户在某次会话中，一旦读到某个数据在某次更新后的值，那么在本次会话中就不会再读到比这值更旧的值，会话一致性是在单调一致性的基础上进一步放松约束，只保证单个用户单个会话内的单调性，在不同用户或同一用户不同会话间则没有保障。

**最终一致性：（eventual consistency）**。用户只能读到某次更新后的值，但系统保证数据将最终达到完全一致的状态(例如在提交表单时,向用户展示最新数据)，只是所需时间不能保障。

**弱一致性：（weak consistency）**。用户无法在确定时间内读到最新更新的值。

## ZooKeeper提供的一致性服务

**顺序一致性：**来自任意特定客户端的更新都会按其发送顺序被提交保持一致。也就是说，如果一个客户端将Znode z的值更新为a，在之后的操作中，它又将z的值更新为b，则没有客户端能够在看到z的值是b之后再看到值a

**原子性：**每个更新要么成功，要么失败。这意味着如果一个更新失败，则不会有客户端会看到这个更新的结果

**单一系统映像：**一个客户端无论连接到哪一台服务器，它看到的都是同样的系统视图。这意味着，如果一个客户端在同一个会话中连接到一台新的服务器，它所看到的系统状态不会比 在之前服务器上所看到的更老。当一台服务器出现故障，导致它的一个客户端需要尝试连接集合体中其他的服务器时，所有滞后于故障服务器的服务器都不会接受该连接请求，除非这些服务器赶上故障服务器。

**持久性：**一个更新一旦成功，其结果就会持久存在并且不会被撤销。这表明更新不会受到服务器故障的影响。

**实时性：**在特定的一段时间内，客户端看到的系统需要被保证是实时的（在十几秒的时间里）。在此时间段内，任何系统的改变将被客户端看到，或者被客户端侦测到。

## ZooKeeper保证CP,不保证A(可用)

不能保证每次服务请求的可用性。任何时刻对ZooKeeper的访问请求能得到一致的数据结果，同时系统对网络分割具备容错性；但是它不能保证每次服务请求的可用性（ZooKeeper可能会丢弃一些请求，消费者程序需要重新请求才能获得结果）

进行leader选举时集群不可用。在使用ZooKeeper获取服务列表时，当leader因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的