一、ZooKeeper实现分布式锁

在分布式应用, 往往存在多个进程提供同一服务. 这些进程有可能在相同的机器上, 也有可能分布在不同的机器上. 如果这些进程共享了一些资源, 可能就需要分布式锁来锁定对这些资源的访问.

1.1、实现分布式锁的几种方式

1）数据库方式：利用表索引的唯一性来实现

创建一个表， 通过索引唯一的方式

create table (id , methodname ...) methodname增加唯一索引

insert 一条数据XXX delete 语句删除这条记录

mysql for update

首先性能不是特别高。

通过数据库的锁来实现多进程之间的互斥，但是这貌似也有一个问题：就是sql超时异常的问题

jdbc超时具体有3种超时，具体见深入理解JDBC的超时设置

框架层的事务超时

jdbc的查询超时

Socket的读超时

这里只涉及到后2种的超时，jdbc的查询超时还好（mysql的jdbc驱动会向服务器发送kill query命令来取消查询），如果一旦出现Socket的读超时，对于如果是同步通信的Socket连接来说(底层实现Connection的可能是同步通信也可能是异步通信)，该连接基本上不能使用了，需要关闭该连接，从新换用新的连接，因为会出现请求和响应错乱的情况，比如jedis出现的类型转换异常，详见Jedis的类型转换异常深究

2）redis实现方式：setNX 、存在则会返回0， 不存在返回1

setnx来创建一个key，如果key不存在则创建成功返回1，如果key已经存在则返回0。依照上述来判定是否获取到了锁

获取到锁的执行业务逻辑，完毕后删除lock_key，来实现释放锁

其他未获取到锁的则进行不断重试，直到自己获取到了锁

改进方案一：

上述逻辑在正常情况下是OK的，但是一旦获取到锁的客户端挂了，没有执行上述释放锁的操作，则其他客户端就无法获取到锁了，所以在这种情况下有2种方式来解决：

为lock_key设置一个过期时间

对lock_key的value进行判断是否过期

改进方案二：

问题1： lock timeout的存在也使得失去了锁的意义，即存在并发的现象。一旦出现锁的租约时间，就意味着获取到锁的客户端必须在租约之内执行完毕业务逻辑，一旦业务逻辑执行时间过长，租约到期，就会引发并发问题。所以有lock timeout的可靠性并不是那么的高。

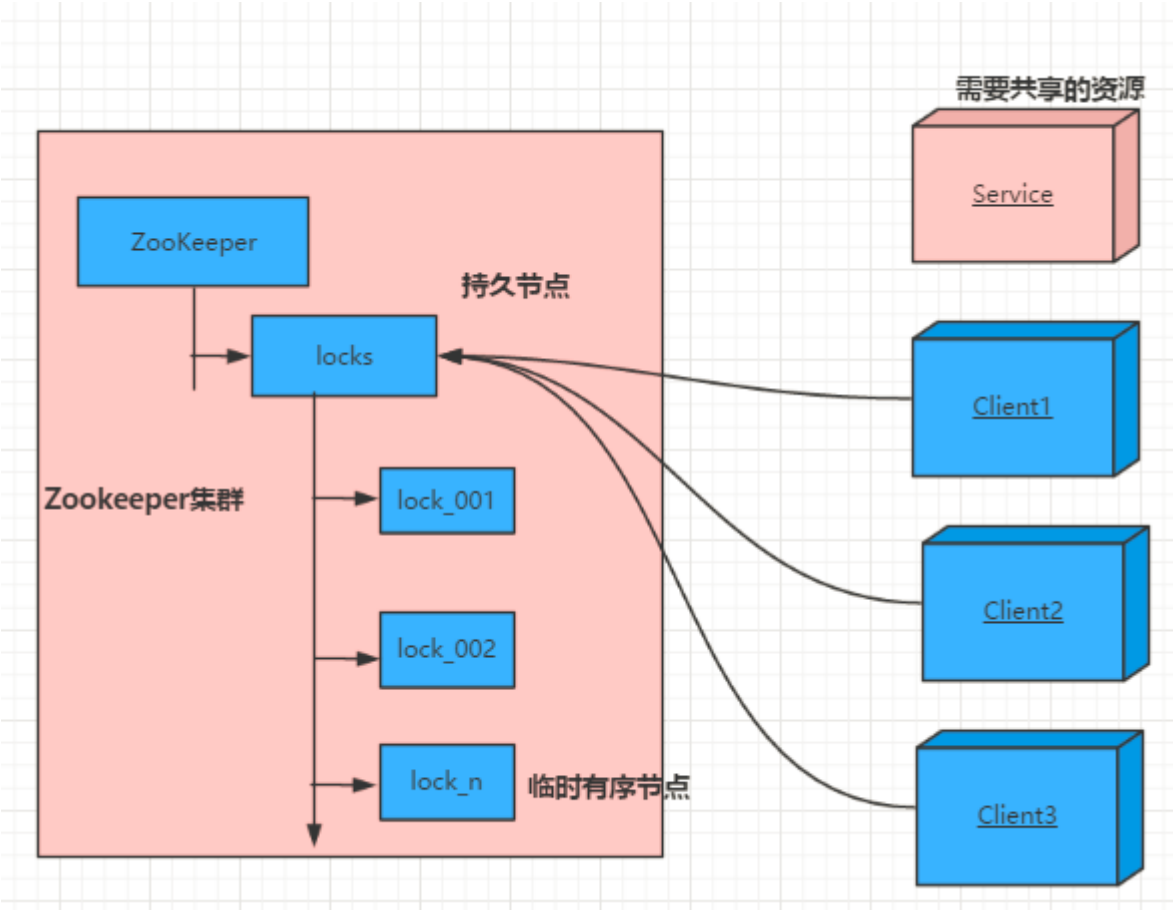
问题2： 上述方式仅仅是redis单机情况下，还存在redis单点故障的问题。如果为了解决单点故障而使用redis的sentinel或者cluster方案，则更加复杂，引入的问题更多。

3）ZooKeeper实现方式

进程需要访问共享数据时, 就在"/lock"节点下创建一个sequence类型的子节点, 称为thisPath. 当thisPath在所有子节点中最小时, 说明该进程获得了锁. 进程获得锁之后, 就可以访问共享资源了. 访问完成后, 需要将thisPath删除. 锁由新的最小的子节点获得.

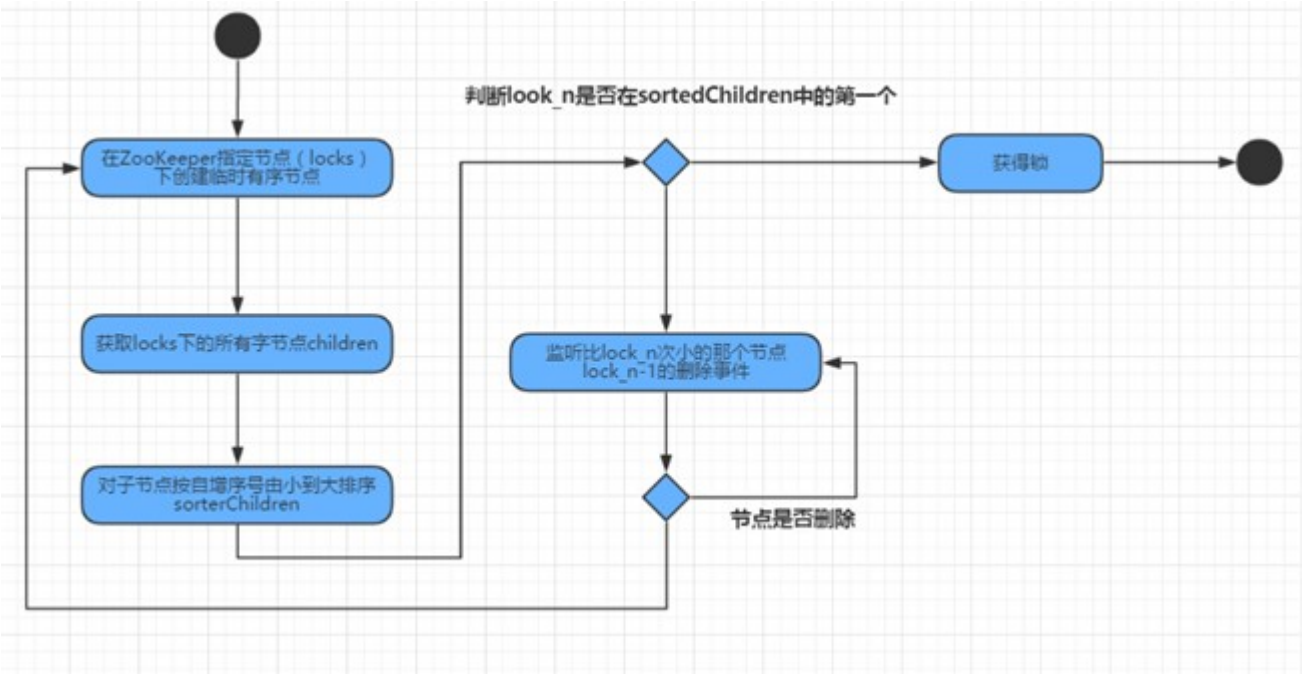
有了清晰的思路之后, 还需要补充一些细节. 进程如何知道thisPath是所有子节点中最小的呢? 可以在创建的时候, 通过getChildren方法获取子节点列表, 然后在列表中找到排名比thisPath前1位的节点, 称为waitPath, 然后在waitPath上注册监听, 当waitPath被删除后, 进程获得通知, 此时说明该进程获得了锁.

原理图如下：



ZooKeeper分布式锁原理图

算法流程图如下：



算法流程图

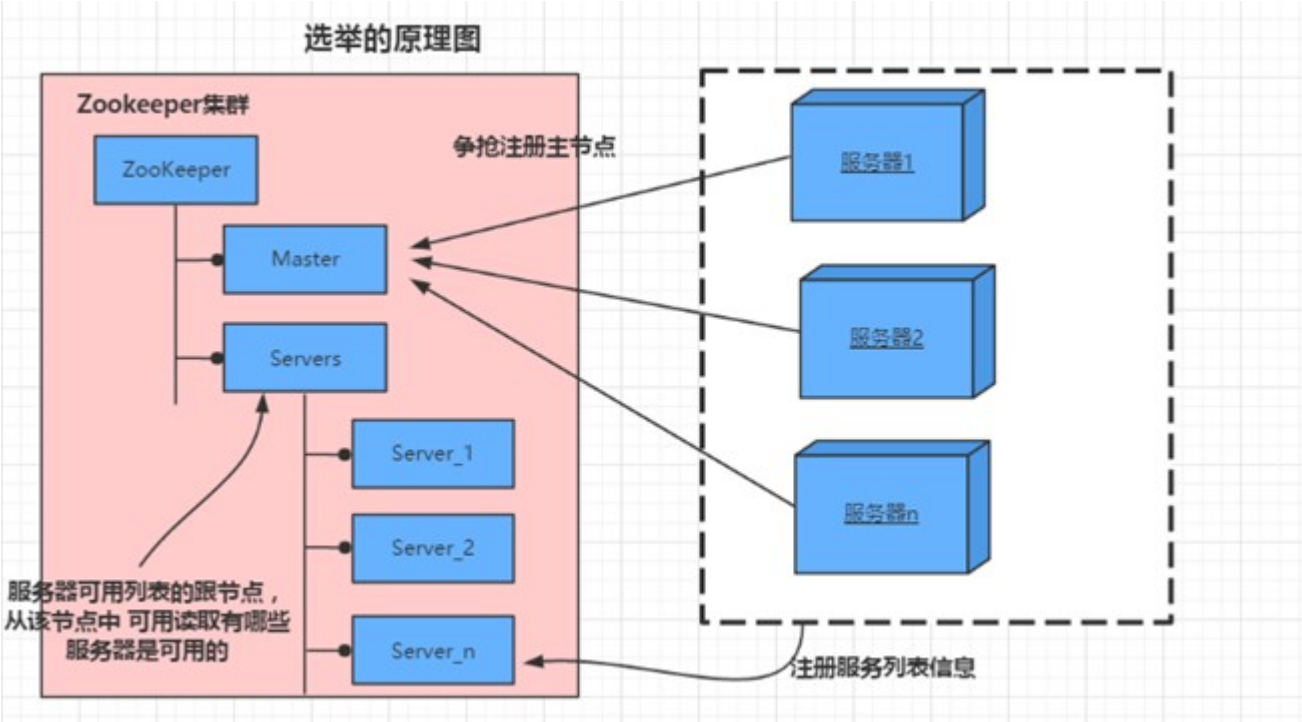
代码实现见github地址： https://git.oschina.net/BJZGD/zookeeper-demo

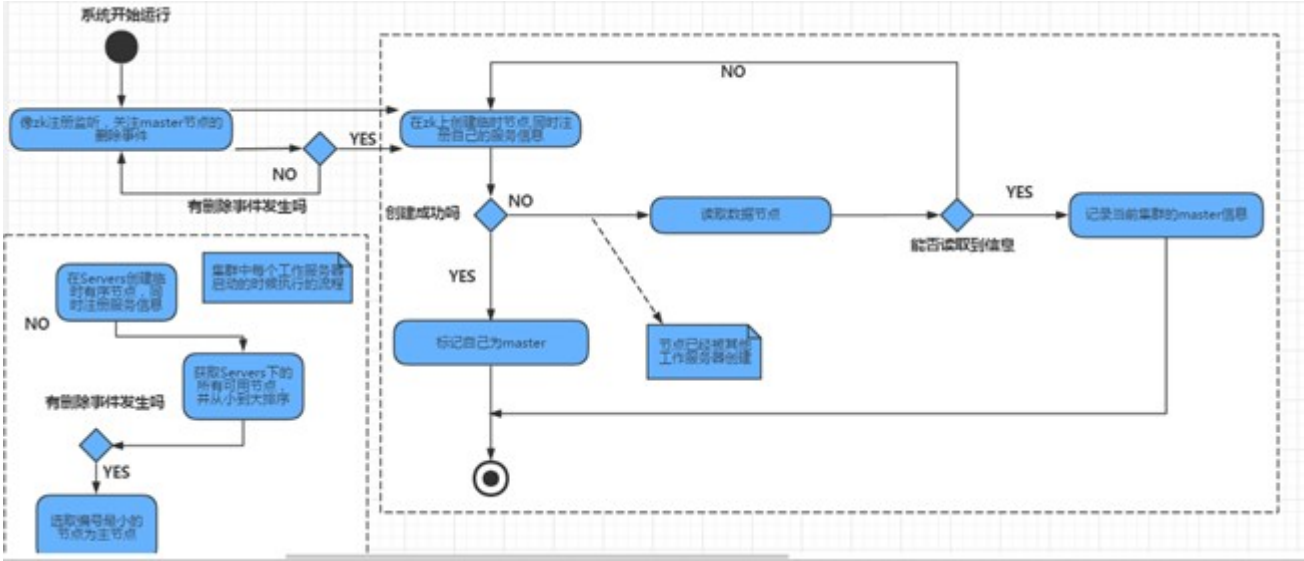
二、ZooKeeper实现Master选举算法

现在很多时候我们的服务需要7*24小时工作，假如一台机器挂了，我们希望能有其它机器顶替它继续工作。此类问题现在多采用master-salve模式，也就是常说的主从模式，正常情况下主机提供服务，备机负责监听主机状态，当主机异常时，可以自动切换到备机继续提供服务(这里有点儿类似于数据库主库跟备库，备机正常情况下只监听，不工作)，这个切换过程中选出下一个主机的过程就是master选举。

1.1实现原理：

选主原理介绍：zookeeper的节点有两种类型，持久节点跟临时节点。临时节点有个特性，就是如果注册这个节点的机器失去连接(通常是宕机)，那么这个节点会被zookeeper删除。选主过程就是利用这个特性，在服务器启动的时候，去zookeeper特定的一个目录下注册一个临时节点(这个节点作为master，谁注册了这个节点谁就是master)，注册的时候，如果发现该节点已经存在，则说明已经有别的服务器注册了(也就是有别的服务器已经抢主成功)，那么当前服务器只能放弃抢主，作为从机存在。同时，抢主失败的当前服务器需要订阅该临时节点的删除事件，以便该节点删除时(也就是注册该节点的服务器宕机了或者网络断了之类的)进行再次抢主操作。从机具体需要去哪里注册服务器列表的临时节点，节点保存什么信息，根据具体的业务不同自行约定。选主的过程，其实就是简单的争抢在zookeeper注册临时节点的操作，谁注册了约定的临时节点，谁就是master。





实现代码：

详见GitHub地址：https://git.oschina.net/BJZGD/zookeeper-demo

三、ZooKeeper实现负载均衡

1、介绍

负载均衡设备作为纵跨网络2-7层协议的设备，往往放置在网络设备和应用设备的连接处，对工程师在网络和应用基本知识方面的要求远高于其他设备，所以我们要在基本功能的理解上下更多的功夫。负载均衡设备还有另外一个称呼：4/7层交换机，但它首先是个2-3层交换机，这要求我们首先掌握2-3层的基本知识，然后才是本文介绍的内容。

服务器负载均衡有三大基本Feature：负载均衡算法，健康检查和会话保持，这三个Feature是保证负载均衡正常工作的基本要素。其他一些功能都是在这三个功能之上的一些深化。下面我们具体介绍一下各个功能的作用和原理。负载均衡设备作为纵跨网络2-7层协议的设备，往往放置在网络设备和应用设备的连接处，对工程师在网络和应用基本知识方面的要求远高于其他设备，所以我们要在基本功能的理解上下更多的功夫。负载均衡设备还有另外一个称呼：4/7层交换机，但它首先是个2-3层交换机，这要求我们首先掌握2-3层的基本知识，然后才是本文介绍的内容。

2、负载均衡算法

一般来说负载均衡设备都会默认支持多种负载均衡分发策略，例如：

Ø 轮询（RoundRobin）将请求顺序循环地发到每个服务器。当其中某个服务器发生故障，AX就把其从顺序循环队列中拿出，不参加下一次的轮询，直到其恢复正常。

Ø 比率（Ratio）：给每个服务器分配一个加权值为比例，根据这个比例，把用户的请求分配到每个服务器。当其中某个服务器发生故障，AX就把其从服务器队列中拿出，不参加下一次的用户请求的分配，直到其恢复正常。

Ø 优先权（Priority）：给所有服务器分组，给每个组定义优先权，将用户的请求分配给优先级最高的服务器组（在同一组内，采用预先设定的轮询或比率算法，分配用户的请求）；当最高优先级中所有服务器或者指定数量的服务器出现故障，AX将把请求送给次优先级的服务器组。这种方式，实际为用户提供一种热备份的方式。

Ø 最少连接数（LeastConnection）：AX会记录当前每台服务器或者服务端口上的连接数，新的连接将传递给连接数最少的服务器。当其中某个服务器发生故障，AX就把其从服务器队列中拿出，不参加下一次的用户请求的分配，直到其恢复正常。

Ø 最快响应时间（Fast Reponse time）：新的连接传递给那些响应最快的服务器。当其中某个服务器发生故障，AX就把其从服务器队列中拿出，不参加下一次的用户请求的分配，直到其恢复正常。

以上为通用的负载均衡算法，还有一些算法根据不同的需求也可能会用到，例如：

Ø 哈希算法(hash): 将客户端的源地址，端口进行哈希运算，根据运算的结果转发给一台服务器进行处理，当其中某个服务器发生故障，就把其从服务器队列中拿出，不参加下一次的用户请求的分配，直到其恢复正常。

Ø 基于策略的负载均衡：针对不同的数据流设置导向规则，用户可自行编辑流量分配策略，利用这些策略对通过的数据流实施导向控制。

Ø 基于数据包的内容分发：例如判断HTTP的URL，如果URL中带有.jpg的扩展名，就把数据包转发到指定的服务器。

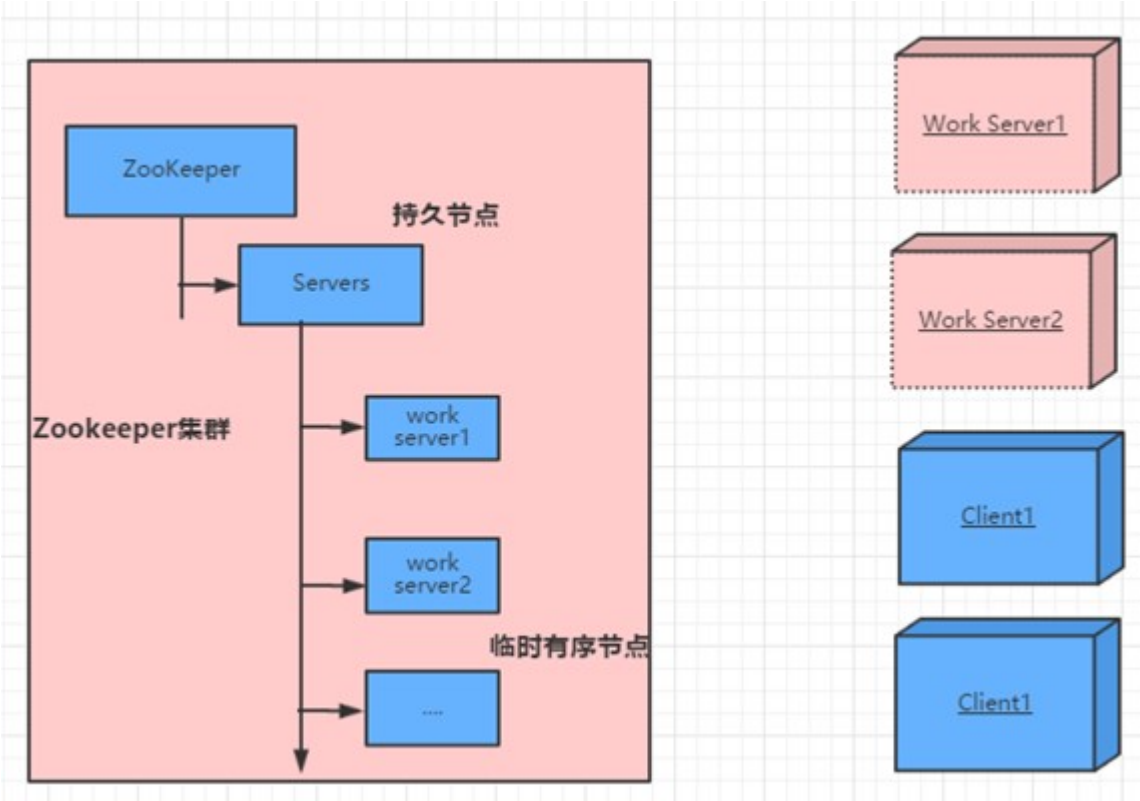
继续看图分析，第二个用户207.17.117.21也访问www.a10networks.com，负载均衡设备根据负载均衡算法将第二个用户的请求转发到第二台服务器来处理。

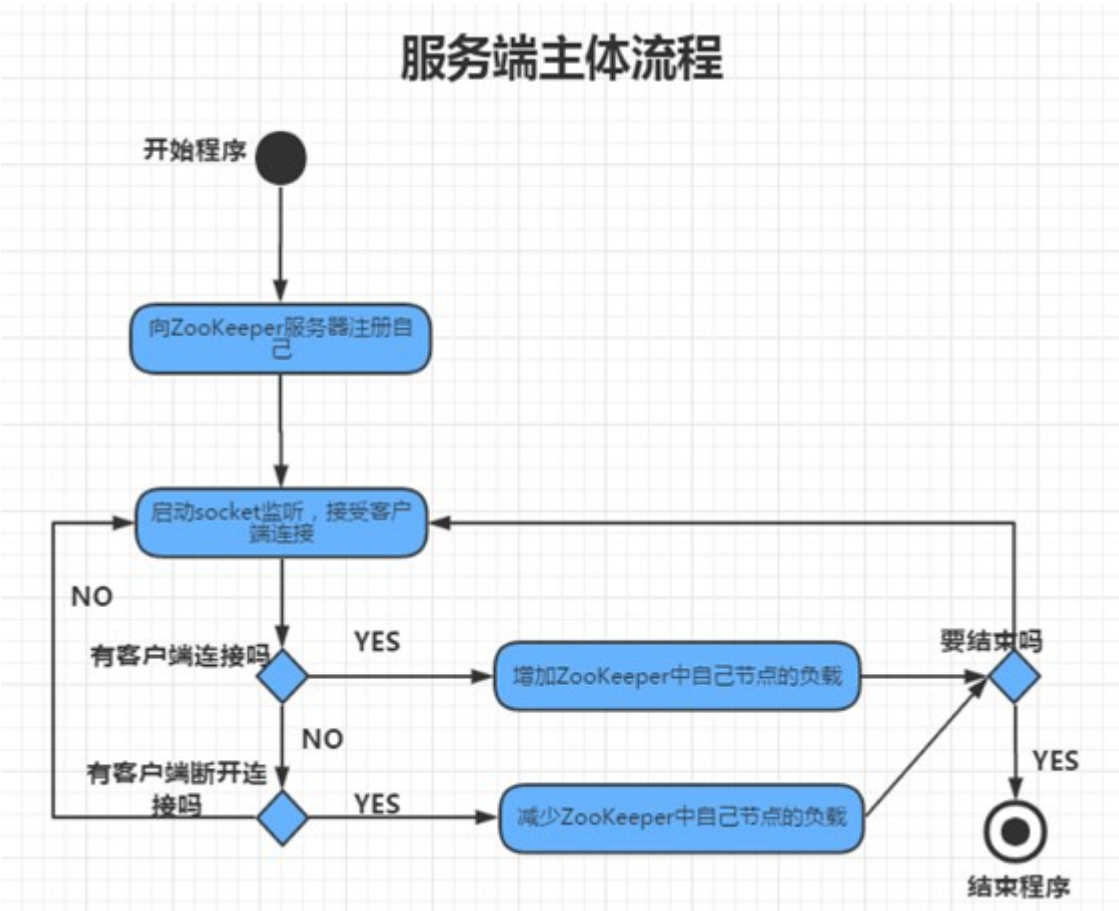
3、实现思路

每台WorkServer启动的时候都会到Server创建临时节点。

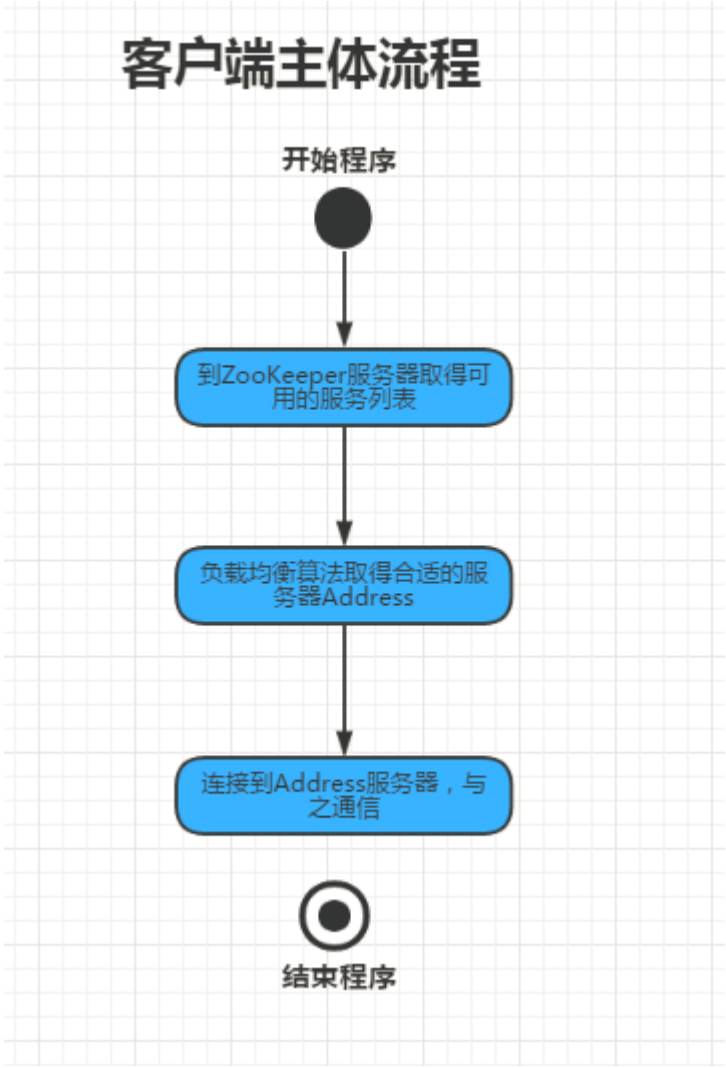
每台ClientServer启动的时候，都会到Server节点下面取得所有WorksServer节点，并通过一定算法取得一台并与之连接。

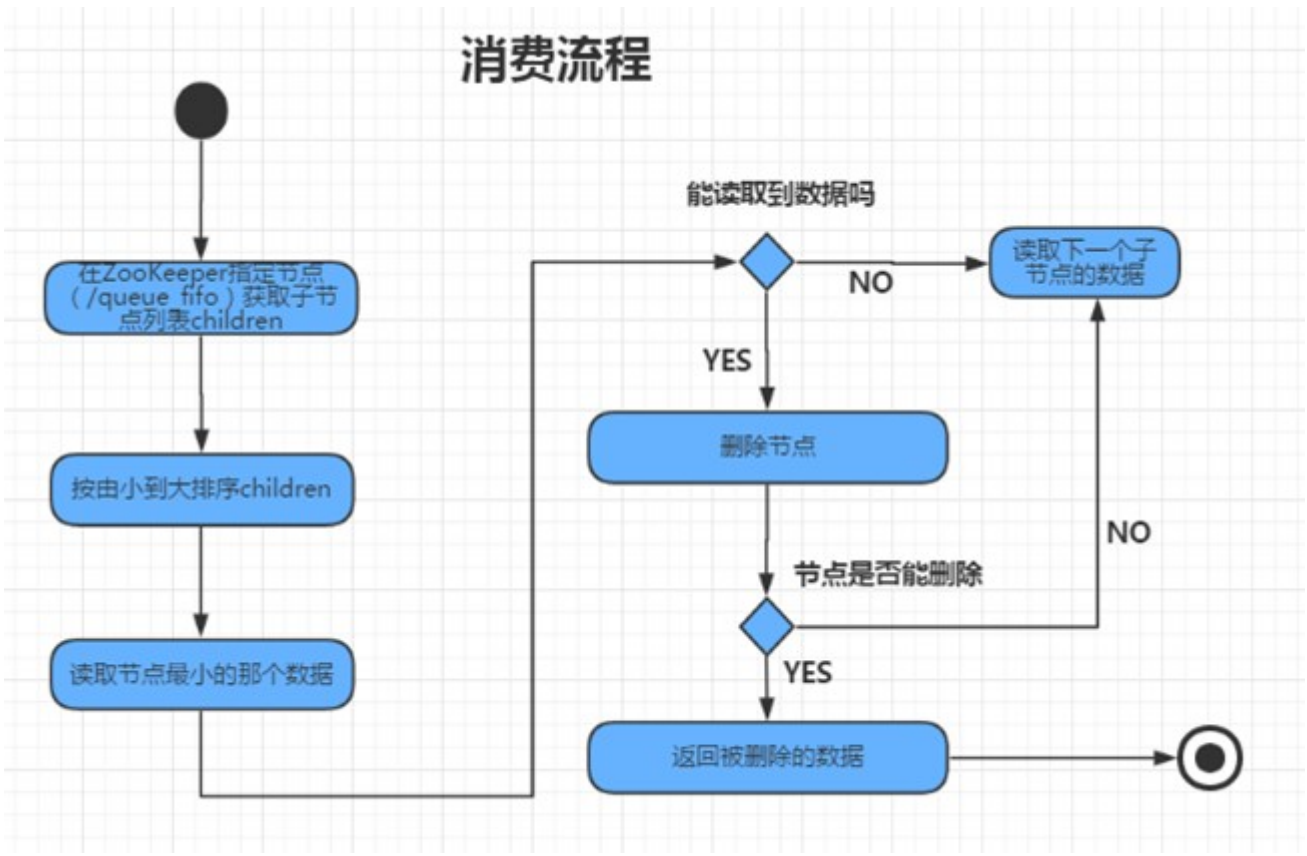
zookeeper实现原理图：



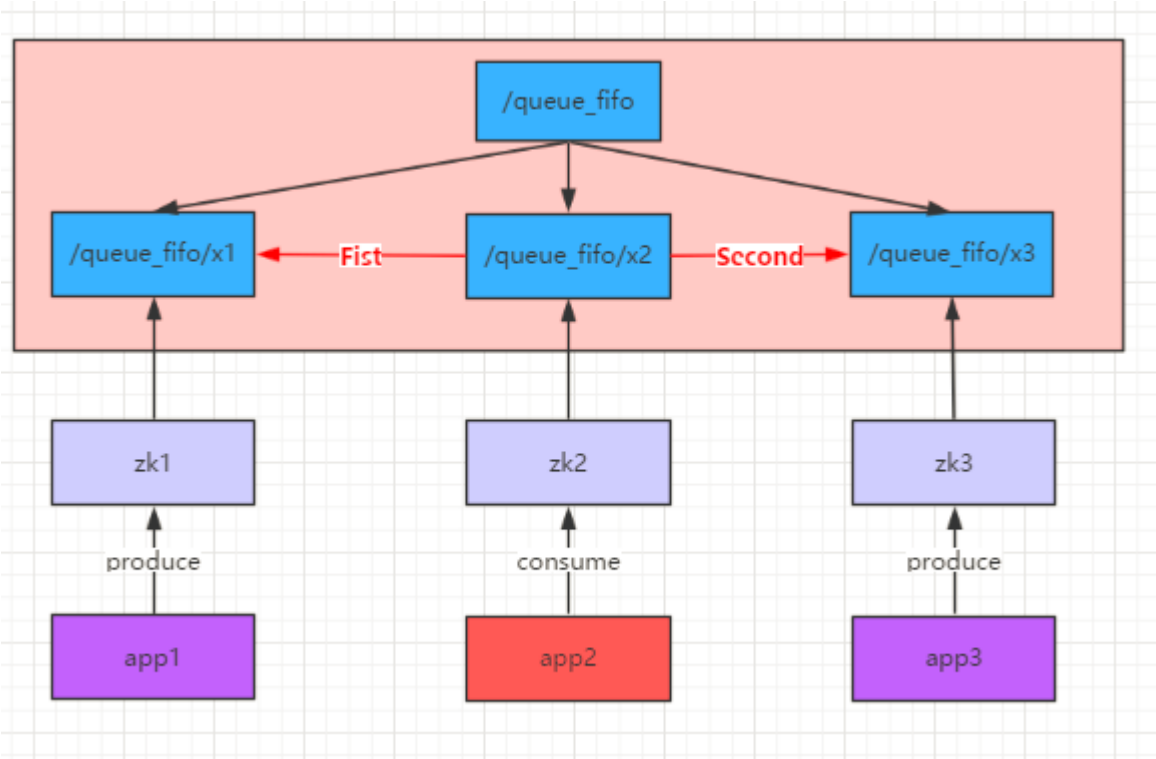


有ClientServer与之建立连接，这台WorksServer的负载计数器加一，断开连接负载计数器减一。负载计数器作为客户端负载均衡算法的依据，客户端会选择负载最轻的WorksServer建立连接。


客户端主体流程
开始程序
到ZooKeeper服务器取得可用的服务列表
负载均衡算法取得合适的服务器Address
连接到Address服务器，与之通信
结束程序

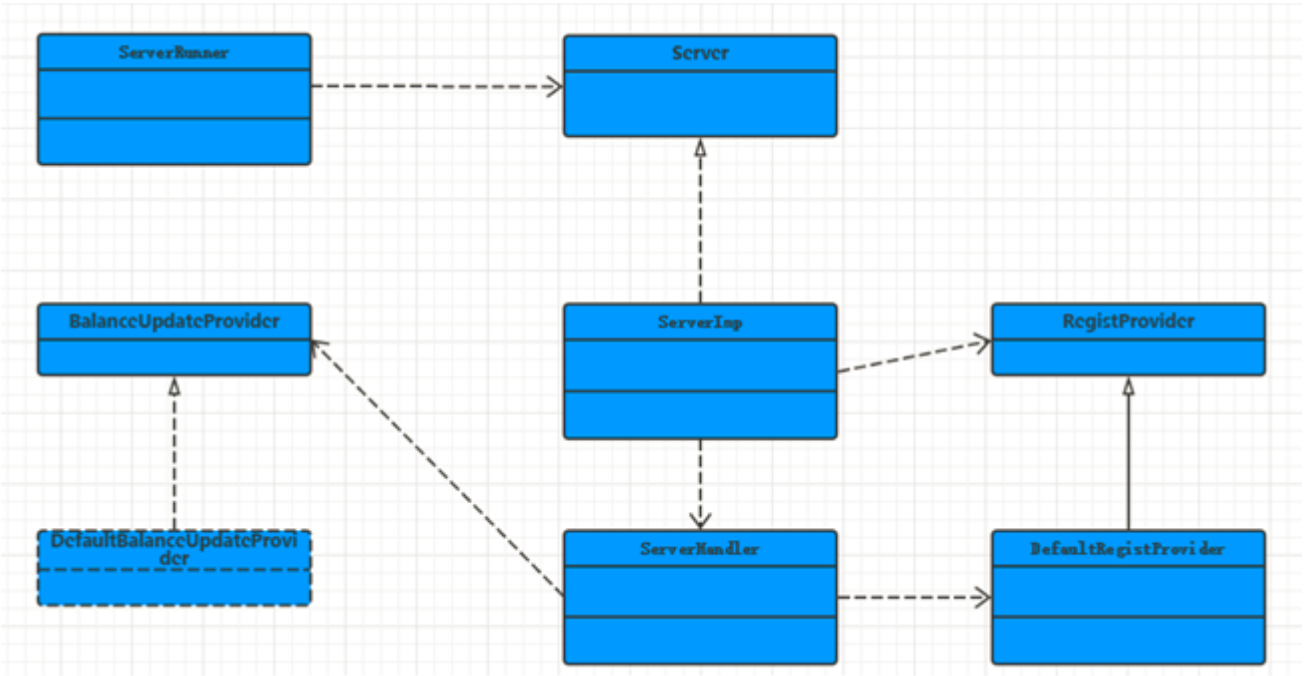

消费流程
在ZooKeeper指定节点（/queue_fifo）获取子节点列表children
按由小到大排序children
读取节点最小的那个数据
能读取到数据吗
NO
读取下一个子节点的数据
YES
删除节点
节点是否能删除
NO
YES
返回被删除的数据

实现实例：



4、代码设计

服务端代码设计：

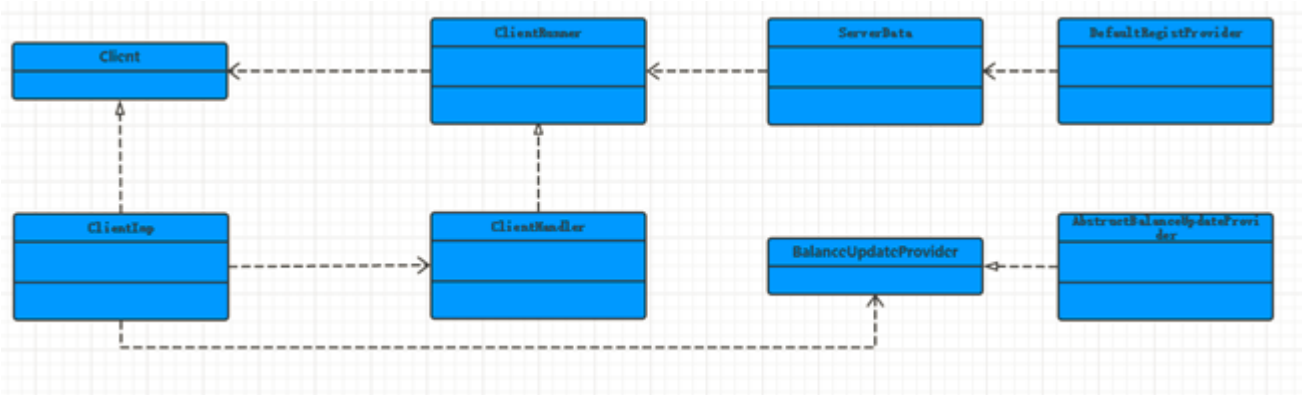


ServerRunner 调度类

RegistProvider 服务端启动时的注册过程

ServerHander 处理与客户端之间的连接

DefaultBalanceUpdateProvider 连接建立与断开，修改负载信息



ClientRunner 调度类

ClientHander 处理与服务器之间的通信

BanceProvider 负载的算法

ServerData 服务器和客户端公用的类，计算负载等使用

见GitHub地址: https://git.oschina.net/BJZGD/zookeeper-demo

四、ZooKeeper实现发布订阅（pub/sub）

ZooKeeper 是一个高可用的分布式数据管理与系统协调框架。基于对 Paxos 算法的实现，使该框架保证了分布式环境中数据的强一致性，也正是基于这样的特性，使得 ZooKeeper 可以解决很多分布式问题。

随着互联网系统规模的不断扩大，大数据时代飞速到来，越来越多的分布式系统将 ZooKeeper 作为核心组件使用，如 Hadoop、Hbase、Kafka、Storm等，因此，正确理解 ZooKeeper 的应用场景，对于 ZooKeeper 的使用者来说显得尤为重要。本节主要将重点围绕数据发布/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方面来讲解 ZooKeeper 的典型应用场景及实现。

1、数据发布/订阅

发布/订阅模式是一对多的关系，多个订阅者对象同时监听某一主题对象，这个主题对象在自身状态发生变化时会通知所有的订阅者对象。使它们能自动的更新自己的状态。发布/订阅可以使得发布方和订阅方独立封装、独立改变。当一个对象的改变需要同时改变其他对象，而且它不知道具体有多少对象需要改变时可以使用发布/订阅模式。发布/订阅模式在分布式系统中的典型应用有配置管理和服务发现、注册。

配置管理是指如果集群中的机器拥有某些相同的配置并且这些配置信息需要动态的改变，我们可以使用发布/订阅模式把配置做统一集中管理，让这些机器格子各自订阅配置信息的改变，当配置发生改变时，这些机器就可以得到通知并更新为最新的配置。

服务发现、注册是指对集群中的服务上下线做统一管理。每个工作服务器都可以作为数据的发布方向集群注册自己的基本信息，而让某些监控服务器作为订阅方，订阅工作服务器的基本信息，当工作服务器的基本信息发生改变如上下线、服务器角色或服务范围变更，监控服务器可以得到通知并响应这些变化。

1.1、配置管理

- 所谓的配置中心，顾名思义就是发布者将数据发布到 ZooKeeper 的一个或一系列节点上，供订阅者进行数据订阅，进而达到动态获取数据的目的，实现配置信息的集中式管理和数据的动态更新。

发布/订阅系统一般有两种设计模式，分别是推(Push)模式和拉(Pull)模式。

- 推模式

服务端主动将数据更新发送给所有订阅的客户端。

- 拉模式

客户端通过采用定时轮询拉取。

ZooKeeper采用的是推拉相结合的方式：客户端向服务端注册自己需要关注的节点，一旦该节点的数据发生变更，那么服务端就会向相应的客户端发送Watcher事件通知，客户端接收到这个消息通知之后，需要主动到服务端获取最新的数据。

如果将配置信息存放到ZK上进行集中管理，那么通常情况下，应用在启动的时候会主动到ZK服务器上进行一次配置信息的获取，同时，在指定上注册一个Watcher监听，这样一来，但凡配置信息发生变更，服务器都会实时通知所有订阅的客户端，从而达到实时获取最新配置信息的目的。

下面我们通过一个“配置管理”的实际案例来展示ZK在“数据发布/订阅”场景下的使用方式。

在我们平常的应用系统开发中，经常会碰到这样的需求：系统中需要使用一些通用的配置信息，例如机器列表信息、运行时的开关配置、数据库的配置信息等。这些全局配置信息通常具备以下特性：

1)、数据量通常比较小

2)、数据内容在运行时会发生变化

3)、集群中各机器共享、配置一致

对于这类配置信息，一般的做法通常可以选择将其存储的本地配置文件或是内存变量中。无论采取哪种配置都可以实现相应的操作。但是一旦遇到集群规模比较大的情况的话，两种方式就不再可取。而我们还需要能够快速的做到全部配置信息的变更，同时希望变更成本足够小，因此我们需要一种更为分布式的解决方案。

接下来我们以“数据库切换”的应用场景展开，看看如何使用ZK来实现配置管理。

- 配置存储

在进行配置管理之前，首先我们需要将初始化配置存储到ZK上去，一般情况下，我们可以在ZK上选取一个数据节点用于配置的存储，我们将需要集中管理的配置信息写入到该数据节点中去。

- 配置获取

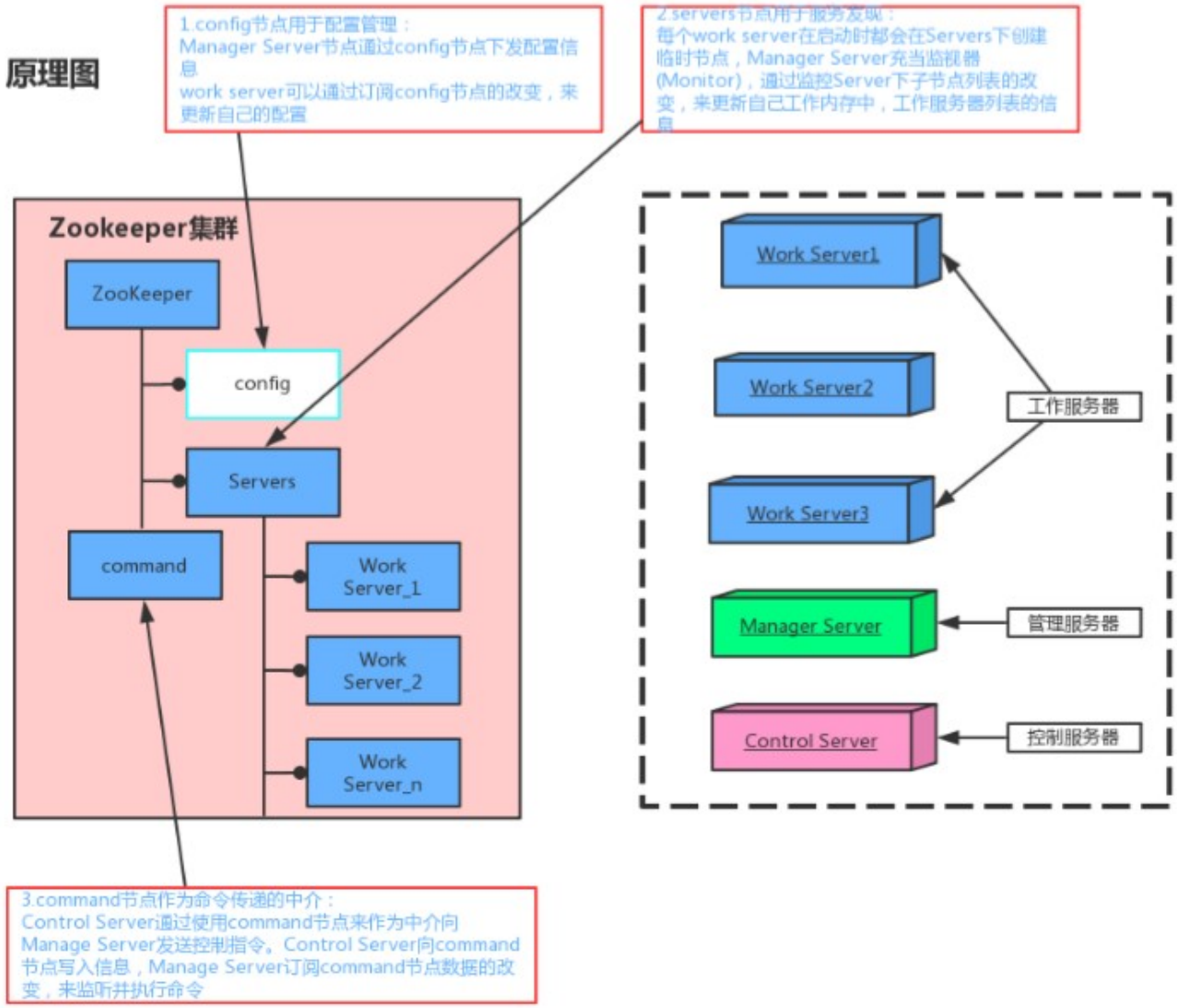
集群中每台机器在启动初始化阶段，首先会从上面提到的ZK的配置节点上读取数据库信息，同时，客户端还需要在该配置节点上注册一个数据变更的Watcher监听，一旦发生节点数据变更，所有订阅的客户端都能够获取数据变更通知。

- 配置变更

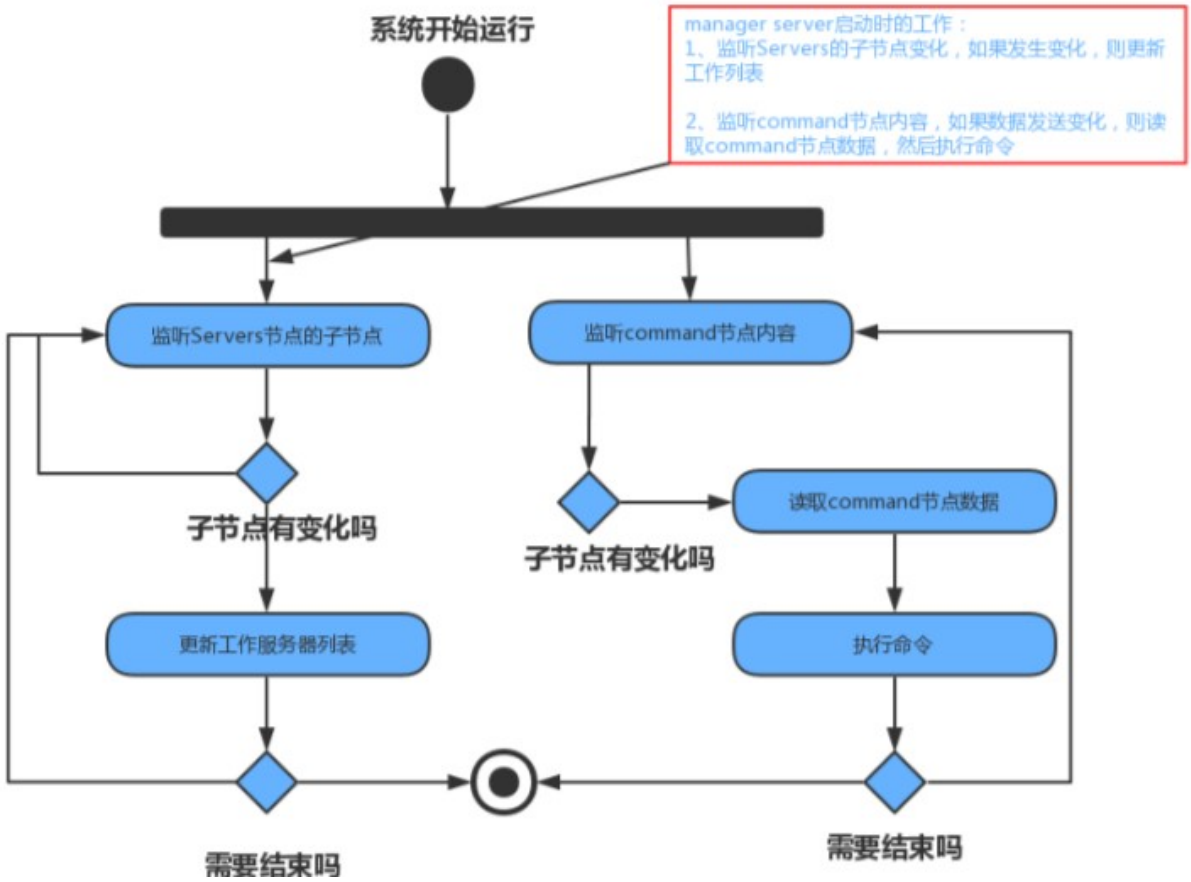
在系统运行过程中，可能会出现需要进行书局切换的情况，这个时候就需要进行配置变更。借助ZK，我们只需要对ZK上配置节点的内容进行更新，ZK就能够帮我们将数据变更的通知发送到各个客户端，每个客户端在接收到这个变更通知后，就可以重新进行最新数据的获取。

2、设计原理

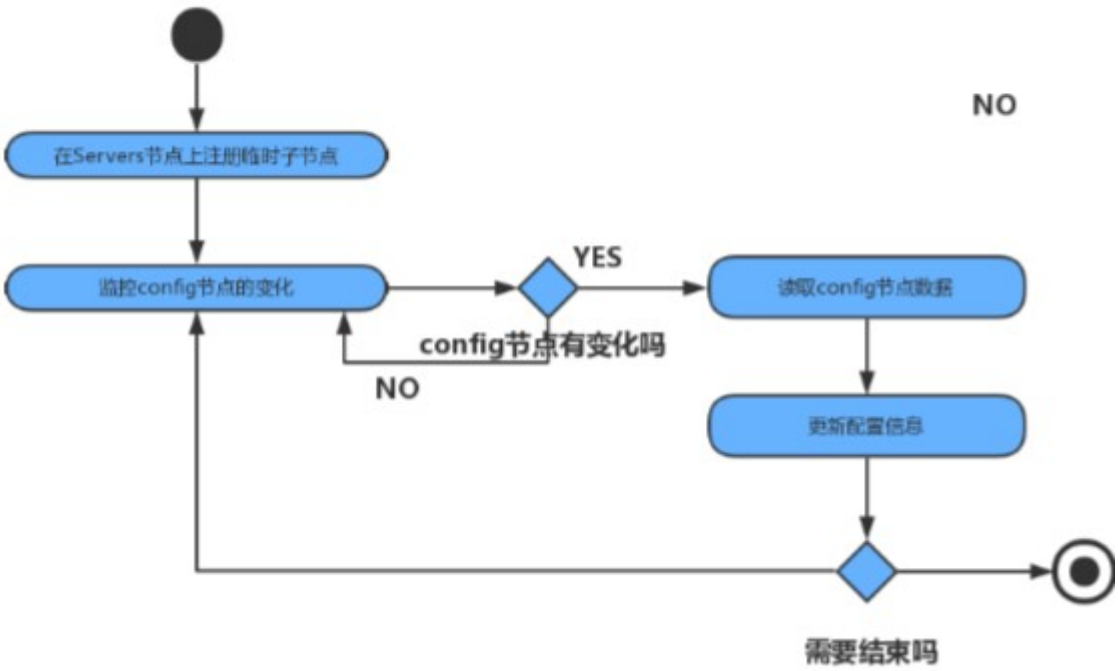
原理图

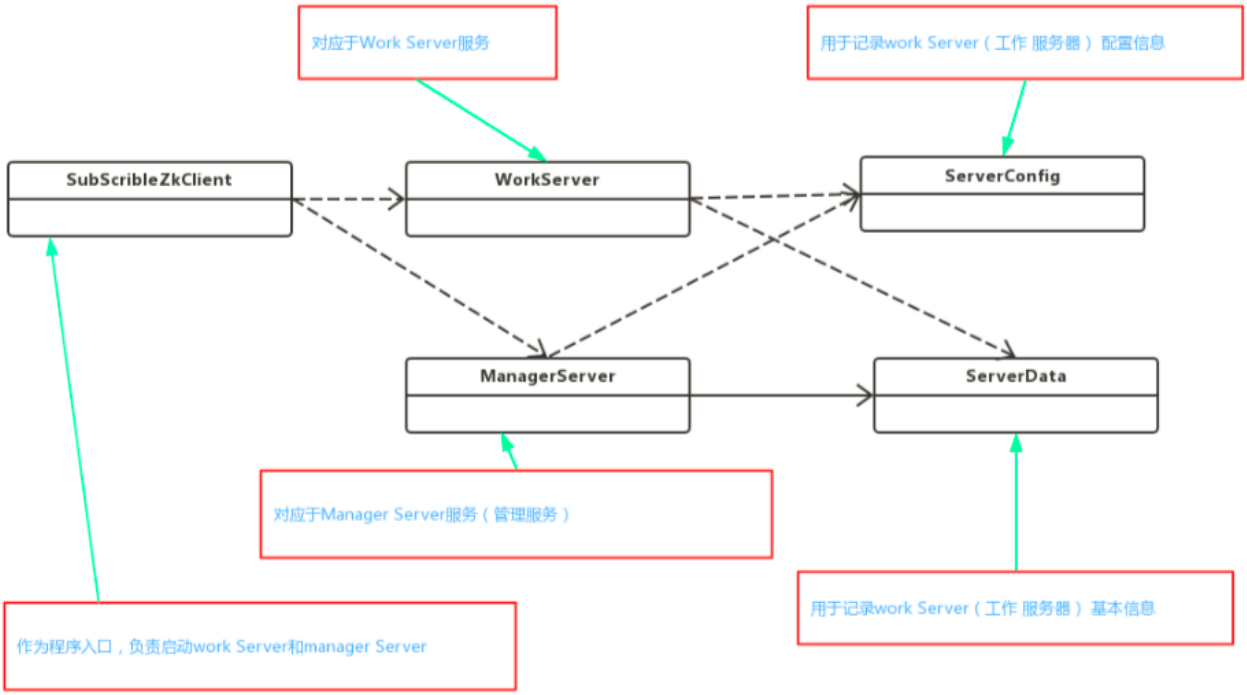

1.config节点用于配置管理：
Manager Server节点通过config节点下发配置信息
work server可以通过订阅config节点的改变，来更新自己的配置
2.servers节点用于服务发现：
每个work server在启动时都会在Servers下创建临时节点，Manager Server充当监视器(Monitor)，通过监控Server下子节点列表的改变，来更新自己工作内存中，工作服务器列表的信息
Zookeeper集群
ZooKeeper
config
Servers
command
Work Server_1
Work Server_2
Work Server_n
Work Server1
Work Server2
Work Server3
工作服务器
Manager Server
管理服务器
Control Server
控制服务器
3.command节点作为命令传递的中介：
Control Server通过使用command节点来作为中介向Manage Server发送控制指令，Control Server向command节点写入信息，Manage Server订阅command节点数据的改变，来监听并执行命令

manager server流程图



work server流程图





3、代码实现

见GitHub地址：

演示效果：

```
work server start...``work server list changed, new list is ``[192.168.1.0]``work server start...``work server list changed, new list is ``[192.168.1.1, 192.168.1.0]``work server start...``work server list changed, new list is ``[192.168.1.1, 192.168.1.0, 192.168.1.2]``work server start...``work server list changed, new list is ``[192.168.1.1, 192.168.1.0, 192.168.1.3, 192.168.1.2]``work server start...``work server list changed, new list is ``[192.168.1.1, 192.168.1.0, 192.168.1.3, 192.168.1.2, 192.168.1.4]``敲回车键退出！
```

```
[zk: localhost:2181(CONNECTED) 1] create /command list
Created /command
[zk: localhost:2181(CONNECTED) 2] set /command modify
cZxid = 0x500000010
ctime = Fri Feb 02 11:16:40 CST 2018
mZxid = 0x500000011
mtime = Fri Feb 02 11:17:11 CST 2018
pZxid = 0x500000010
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
[zk: localhost:2181(CONNECTED) 3]
```

`cmd:list``[192.168.1.1, 192.168.1.0, 192.168.1.3, 192.168.1.2, 192.168.1.4]``cmd:modify``new Work server config is:ServerConfig [dbUrl=jdbc:mysql://localhost:3306/mydb, dbPwd=123456, dbUser=root_modify]``new Work server config is:ServerConfig [dbUrl=jdbc:mysql://localhost:3306/mydb, dbPwd=123456, dbUser=root_modify]``new Work server config is:ServerConfig [dbUrl=jdbc:mysql://localhost:3306/mydb, dbPwd=123456, dbUser=root_modify]``new Work server config is:ServerConfig [dbUrl=jdbc:mysql://localhost:3306/mydb, dbPwd=123456, dbUser=root_modify]``new Work server config is:ServerConfig [dbUrl=jdbc:mysql://localhost:3306/mydb, dbPwd=123456, dbUser=root_modify]`