# 第三课：zookeeper 典型使用场景实践

**课程概要：**

1. 分布式集群管理
2. 分布式注册中心
3. 分布式锁
4. 分布式JOB

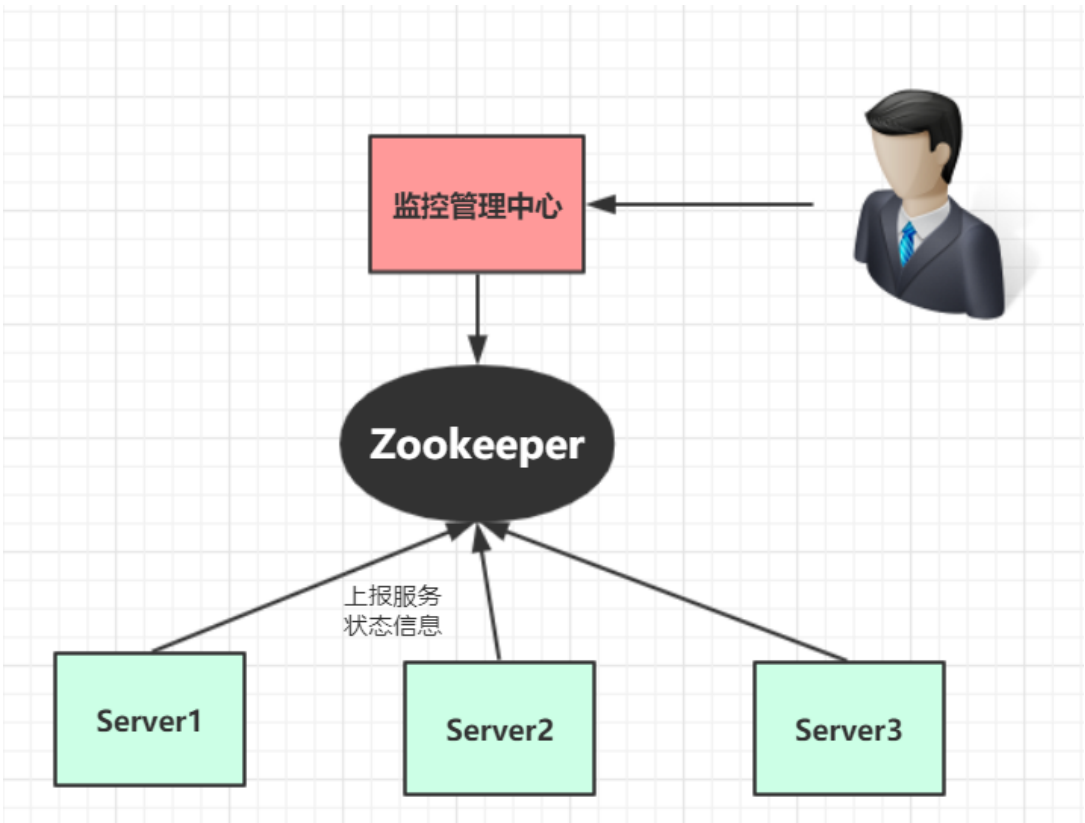
## 一、 分布式集群管理

---

**分布式集群管理的需求：**

1. 主动查看线上服务节点
2. 查看服务节点资源使用情况
3. 服务离线通知
4. 服务资源（CPU、内存、硬盘）超出阀值通知

**架构设计：**




**节点结构：**

1. tuling-manger // 根节点
    a. server00001 :<json> //服务节点 1
    b. server00002 :<json>//服务节点 2
    c. server.....n :<json>//服务节点 n

服务状态信息:

a. ip
b. cpu
c. memory
d. disk

**功能实现:**

**数据生成与上报:**

1. 创建临时节点:
2. 定时变更节点状态信息:

**主动查询:**

1、实时查询 zookeeper 获取集群节点的状态信息。

**被动通知:**

1. 监听根节点下子节点的变化情况,如果CPU 等硬件资源低于警告位则发出警报。

**关键示例代码:**

```
package com.tuling;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.tuling.os.CPUMonitorCalc;
import com.tuling.os.OsBean;
import org.I0Itec.zkclient.IZkChildListener;
import org.I0Itec.zkclient.ZkClient;

import java.io.IOException;
import java.lang.instrument.Instrumentation;
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryUsage;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
 * @author Tommy
 * Created by Tommy on 2019/9/22
 **/
public class Agent {

    private String server = "192.168.0.149:2181";
    ZkClient zkClient;
```

```
    private static Agent instance;
    private static final String rootPath = "/tuling-manger";
    private static final String servicePath = rootPath + "/service";
    private String nodePath;
    private Thread stateThread;
    List<OsBean> list = new ArrayList<>();

    public static void premain(String args, Instrumentation
instrumentation) {
        instance = new Agent();
        if (args != null) {
            instance.server = args;
        }
        instance.init();

    }

    // 初始化连接
    public void init() {
        zkClient = new ZkClient(server, 5000, 10000);
        System.out.println("zk连接成功" + server);
        buildRoot();
        createServerNode();
        stateThread = new Thread(() -> {
            while (true) {
                updateServerNode();
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, "zk_stateThread");
        stateThread.setDaemon(true);
        stateThread.start();

    }

    // 构建根节点
    public void buildRoot() {
        if (!zkClient.exists(rootPath)) {
            zkClient.createPersistent(rootPath);
        }
    }

    // 生成服务节点
    public void createServerNode() {
        nodePath = zkClient.createEphemeralSequential(servicePath,
getOsInfo());
        System.out.println("创建节点:" + nodePath);
    }
```

```
77
78      // 监听服务节点状态改变
79
80
81      public void updateServerNode() {
82          zkClient.writeData(nodePath, getOsInfo());
83      }
84
85      // 更新服务节点状态
86      public String getOsInfo() {
87          OsBean bean = new OsBean();
88          bean.lastUpdateTime = System.currentTimeMillis();
89          bean.ip = getLocalIp();
90          bean.cpu = CPUMonitorCalc.getInstance().getProcessCpu();
91          MemoryUsage memoryUsag =
ManagementFactory.getMemoryMXBean().getHeapMemoryUsage();
92          bean.usableMemorySize = memoryUsag.getUsed() / 1024 / 1024;
93          bean.usableMemorySize = memoryUsag.getMax() / 1024 / 1024;
94          ObjectMapper mapper = new ObjectMapper();
95          try {
96              return mapper.writeValueAsString(bean);
97          } catch (JsonProcessingException e) {
98              throw new RuntimeException(e);
99          }
100     }
101
102     public void updateNode(String path, Object data) {
103         if (zkClient.exists(path)) {
104             zkClient.writeData(path, data);
105         } else {
106             zkClient.createEphemeral(path, data);
107         }
108     }
109
110
111     public static String getLocalIp() {
112         InetAddress addr = null;
113         try {
114             addr = InetAddress.getLocalHost();
115         } catch (UnknownHostException e) {
116             throw new RuntimeException(e);
117         }
118         return addr.getHostAddress();
119     }
120
121 }
```

实现效果图：

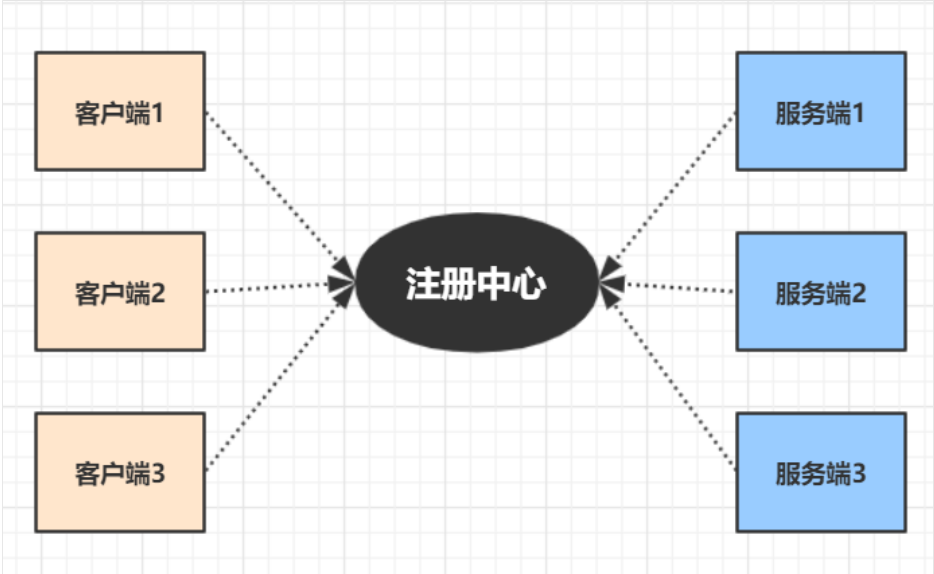
**图灵分布式集群管理系统**

| pid | ip | CPU负载 | 占用内存 | 剩余内存 |
| --- | --- | --- | --- | --- |
| 8368@CAF0EMACHYH5C79 | 192.168.0.132 | 0.078 | 98MB | 3,604MB |

## 二 、分布式注册中心

---

在单体式服务中，通常是由多个客户端去调用一个服务，只要在客户端中配置唯一服务节点地址即可，当升级到分布式后，服务节点变多，像阿里一线大厂服务节点更是上万之多，这么多节点不可能手动配置在客户端，这里就需要一个中间服务，专门用于帮助客户端发现服务节点，即许多技术书籍经常提到的**服务发现**。

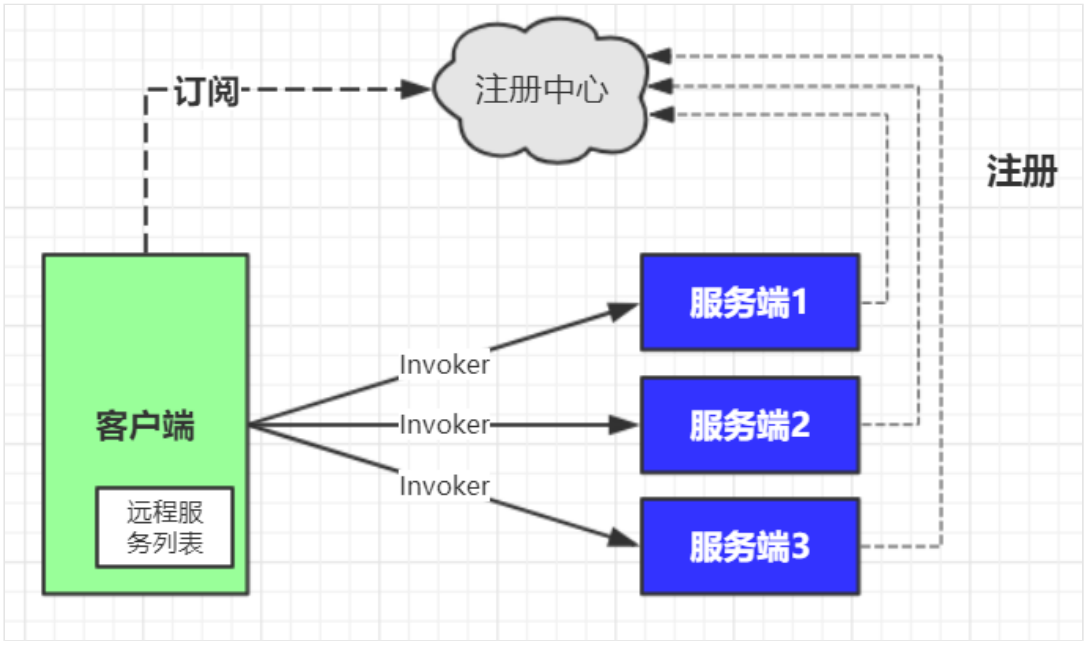



一个完整的注册中心涵盖以下功能特性：

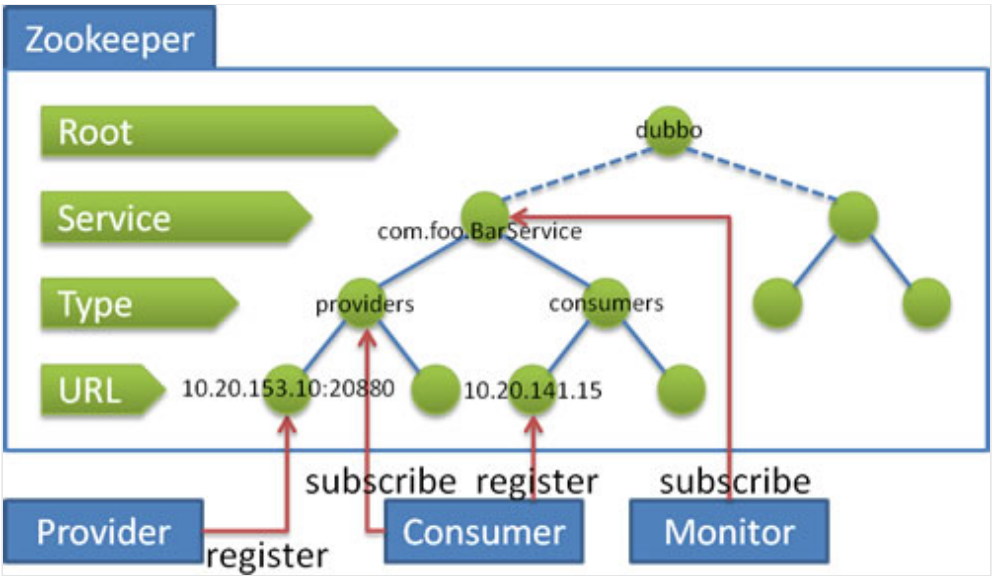
- **服务注册**：提供者上线时将自提供的服务提交给注册中心。
- **服务注销**：通知注册心提供者下线。
- **服务订阅**：动态实时接收服务变更消息。
- **可靠**：注册服务本身是集群的，数据冗余存储。避免单点故障，及数据丢失。
- **容错**：当服务提供者出现宕机，断电等极情况时，注册中心能够动态感知并通知客户端服务提供者的状态。

### Dubbo 对zookeeper的使用

阿里著名的开源项目Dubbo 是一个基于JAVA的RCP框架，其中必不可少的注册中心可基于多种第三方组件实现，但其官方推荐的还是Zookeeper做为注册中心服务。




**Dubbo Zookeeper注册中心存储结构：**




**节点说明：**

| 类别 | 属性 | 说明 |
| --- | --- | --- |
| Root | 持久节点 | 根节点名称，默认是 ”dubbo” |
| Service | 持久节点 | 服务名称，完整的服务类名 |
| type | 持久节点 | 可选值：providers(提供者)、consumers（消费者）、configurators(动态配置)、routers |
| URL | 临时节点 | url名称 包含服务提供者的 IP 端口 及配置等信息。 |

**流程说明：**

1. 服务提供者启动时: 向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址
2. 服务消费者启动时: 订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并向 /dubbo/com.foo.BarService/consumers 目录下写入自己的

URL 地址

3. 监控中心启动时: 订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址。

**示例演示：**

服务端代码：

```
package com.tuling.zk.dubbo;

import com.alibaba.dubbo.config.ApplicationConfig;
import com.alibaba.dubbo.config.ProtocolConfig;
import com.alibaba.dubbo.config.RegistryConfig;
import com.alibaba.dubbo.config.ServiceConfig;

import java.io.IOException;

/**
 * @author Tommy
 * Created by Tommy on 2019/10/8
 **/
public class Server {
    public void openServer(int port) {
        // 构建应用
        ApplicationConfig config = new ApplicationConfig();
        config.setName("simple-app");

        // 通信协议
        ProtocolConfig protocolConfig = new ProtocolConfig("dubbo", port);
        protocolConfig.setThreads(200);

        ServiceConfig<UserService> serviceConfig = new ServiceConfig();
        serviceConfig.setApplication(config);
        serviceConfig.setProtocol(protocolConfig);
        serviceConfig.setRegistry(new
RegistryConfig("zookeeper://192.168.0.149:2181"));
        serviceConfig.setInterface(UserService.class);
        UserServiceImpl ref = new UserServiceImpl();
        serviceConfig.setRef(ref);
        //开始提供服务  开张做生意
        serviceConfig.export();
        System.out.println("服务已开启!端
口:"+serviceConfig.getExportedUrls().get(0).getPort());
        ref.setPort(serviceConfig.getExportedUrls().get(0).getPort());
    }

    public static void main(String[] args) throws IOException {
        new Server().openServer(-1);
        System.in.read();
    }
```

```
}
```

客户端代码：

```
package com.tuling.zk.dubbo;

import com.alibaba.dubbo.config.ApplicationConfig;
import com.alibaba.dubbo.config.ReferenceConfig;
import com.alibaba.dubbo.config.RegistryConfig;

import java.io.IOException;

/**
 * @author Tommy
 * Created by Tommy on 2018/11/20
 **/
public class Client {
    UserService service;

    // URL 远程服务的调用地址
    public UserService buildService(String url) {
        ApplicationConfig config = new ApplicationConfig("young-app");
        // 构建一个引用对象
        ReferenceConfig<UserService> referenceConfig = new
ReferenceConfig<>();
        referenceConfig.setApplication(config);
        referenceConfig.setInterface(UserService.class);
//        referenceConfig.setUrl(url);
        referenceConfig.setRegistry(new
RegistryConfig("zookeeper://192.168.0.149:2181"));
        referenceConfig.setTimeout(5000);
        // 透明化
        this.service = referenceConfig.get();
        return service;
    }

    static int i = 0;

    public static void main(String[] args) throws IOException {
        Client client1 = new Client();
        client1.buildService("");
        String cmd;
        while (!(cmd = read()).equals("exit")) {
            UserVo u = client1.service.getUser(Integer.parseInt(cmd));
            System.out.println(u);
        }
    }
    private static String read() throws IOException {
        byte[] b = new byte[1024];
        int size = System.in.read(b);
        return new String(b, 0, size).trim();
    }
```

```
}
```

查询zk 实际存储内容：

```
/dubbo
/dubbo/com.tuling.zk.dubbo.UserService
/dubbo/com.tuling.zk.dubbo.UserService/configurators
/dubbo/com.tuling.zk.dubbo.UserService/routers

/dubbo/com.tuling.zk.dubbo.UserService/providers
/dubbo/com.tuling.zk.dubbo.UserService/providers/dubbo://192.168.0.132:20880/com.tuling.zk.dubbo.UserService?anyhost=true&application=simple-app&dubbo=2.6.2&generic=false&interface=com.tuling.zk.dubbo.UserService&methods=getUser&pid=11128&side=provider&threads=200&timestamp=1570518302772
/dubbo/com.tuling.zk.dubbo.UserService/providers/dubbo://192.168.0.132:20881/com.tuling.zk.dubbo.UserService?anyhost=true&application=simple-app&dubbo=2.6.2&generic=false&interface=com.tuling.zk.dubbo.UserService&methods=getUser&pid=12956&side=provider&threads=200&timestamp=1570518532382
/dubbo/com.tuling.zk.dubbo.UserService/providers/dubbo://192.168.0.132:20882/com.tuling.zk.dubbo.UserService?anyhost=true&application=simple-app&dubbo=2.6.2&generic=false&interface=com.tuling.zk.dubbo.UserService&methods=getUser&pid=2116&side=provider&threads=200&timestamp=1570518537021

/dubbo/com.tuling.zk.dubbo.UserService/consumers
/dubbo/com.tuling.zk.dubbo.UserService/consumers/consumer://192.168.0.132/com.tuling.zk.dubbo.UserService?application=young-app&category=consumers&check=false&dubbo=2.6.2&interface=com.tuling.zk.dubbo.UserService&methods=getUser&pid=9200&side=consumer&timeout=5000&timestamp=1570518819628
```

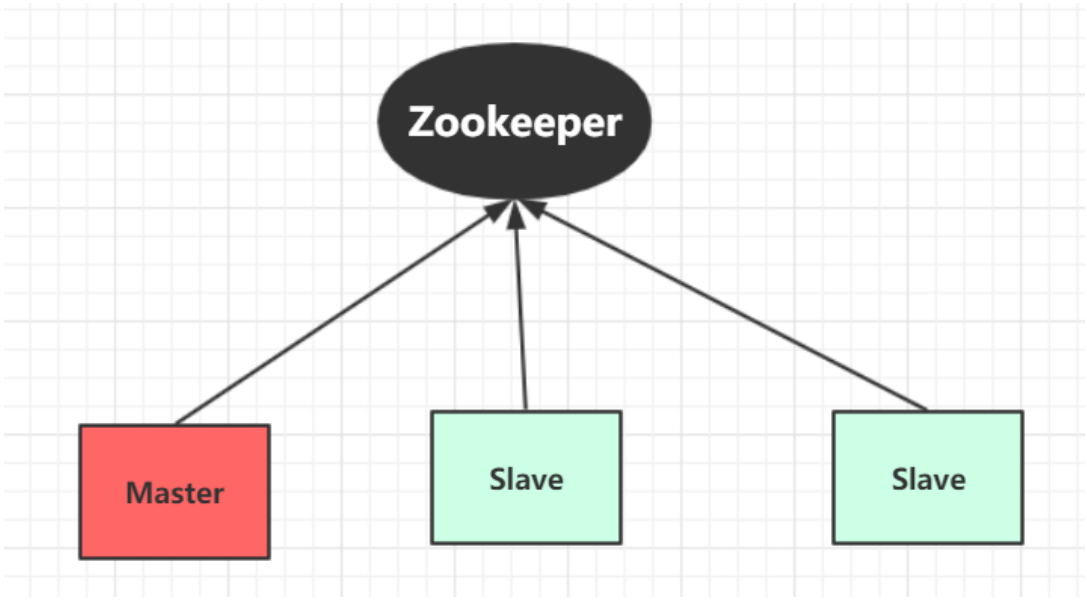
## 三、分布式JOB

---

**分布式JOB需求：**

1. 多个服务节点只允许其中一个主节点运行JOB任务。
2. 当主节点挂掉后能自动切换主节点，继续执行JOB任务。

**架构设计：**




**node结构：**

1. tuling-master
   a. server0001:master
   b. server0002:slave
   c. server000n:slave

**选举流程：**

服务启动：

1. 在tuling-maste下创建server子节点，值为slave
2. 获取所有tuling-master 下所有子节点
3. 判断是否存在master 节点
4. 如果没有设置自己为master节点

子节点删除事件触发：

1. 获取所有tuling-master 下所有子节点
2. 判断是否存在master 节点
3. 如果没有设置最小值序号为master 节点

## 四、分布式锁

---

**锁的的基本概念：**

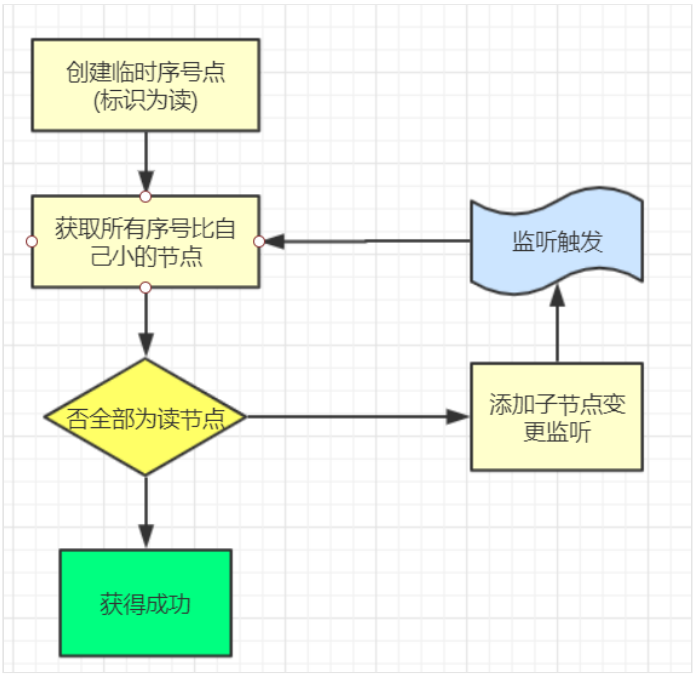
开发中锁的概念并不陌生，通过锁可以实现在多个线程或多个进程间在争抢资源时，能够合理的分配置资源的所有权。在单体应用中我们可以通过 synchronized 或ReentrantLock 来实现锁。但在分布式系统中，仅仅是加synchronized 是不够的，需要借助第三组件来实现。比如一些简单的做法是使用 关系型数据行级锁来实现不同进程之间的互斥，但大型分布式系统的性能瓶颈往往集中在数据库操作上。为了提高性能得采用如Redis、Zookeeper之内的组件实现分布式锁。

**共享锁：**也称作只读锁，当一方获得共享锁之后，其它方也可以获得共享锁。但其只允许读取。在共享锁全部释放之前，其它方不能获得写锁。
**排它锁：**也称作读写锁，获得排它锁后，可以进行数据的读写。在其释放之前，其它方不能获得任何锁。

**锁的获取：**

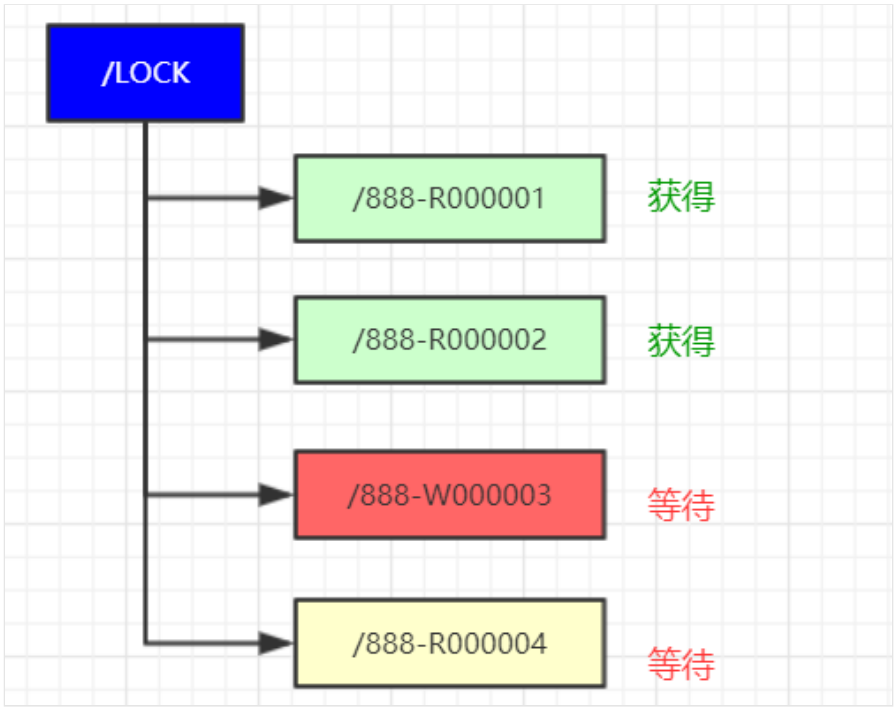
某银行帐户，可以同时进行帐户信息的读取，但读取其间不能修改帐户数据。其帐户ID为:888

- 获得读锁流程：




1、基于资源ID创建临时序号读锁节点
/lock/888.R0000000002 Read
2、获取 /lock 下所有子节点，判断其最小的节点是否为读锁，如果是则获锁成功
3、最小节点不是读锁，则阻塞等待。添加lock/ 子节点变更监听。
4、当节点变更监听触发，执行第2步

**数据结构：**




- 获得写锁：

1、基于资源ID创建临时序号写锁节点

/lock/888.R0000000002 Write

2、获取 /lock 下所有子节点，判断其最小的节点是否为自己，如果是则获锁成功

3、最小节点不是自己，则阻塞等待。添加lock/ 子节点变更监听。

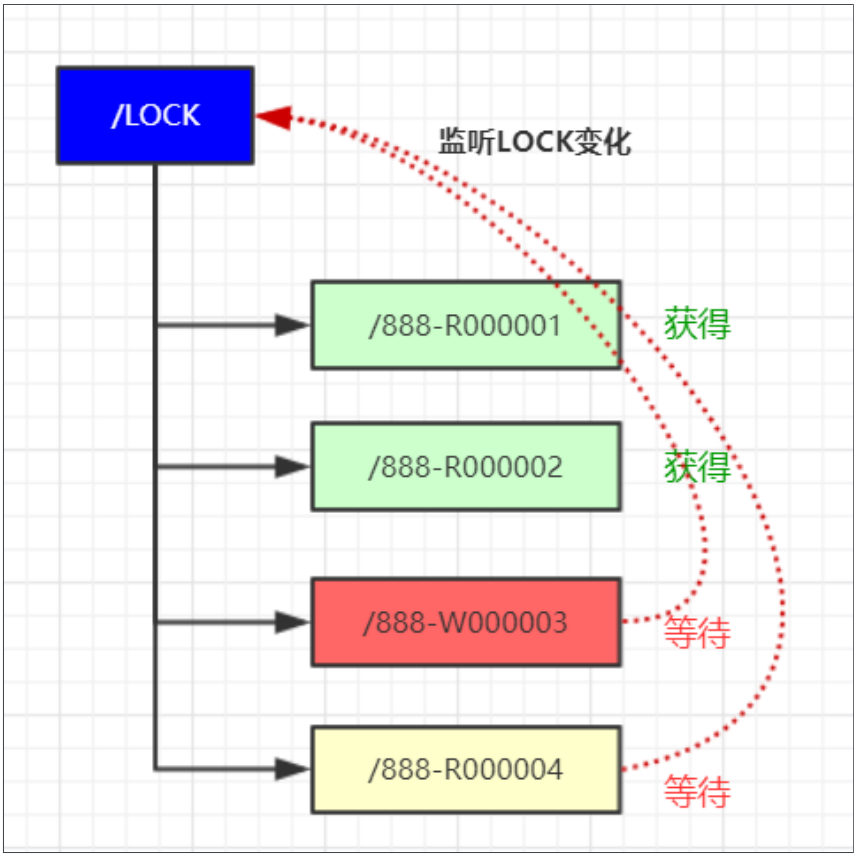
4、当节点变更监听触发，执行第2步

- 释放锁：

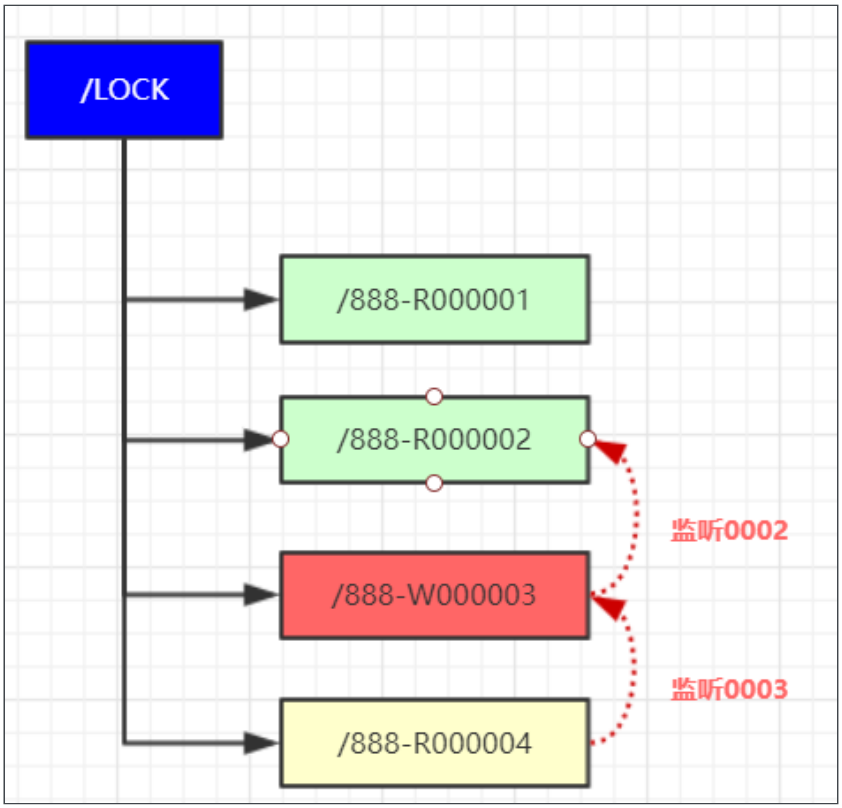
读取完毕后，手动删除临时节点，如果获锁期间宕机，则会在会话失效后自动删除。

**关于羊群效应：**

在等待锁获得期间，所有等待节点都在监听 Lock节点，一但lock 节点变更所有等待节点都会被触发，然后在同时反查Lock 子节点。如果等待对例过大会使用Zookeeper承受非常大的流量压力。




为了改善这种情况，可以采用监听链表的方式，每个等待对列只监听前一个节点，如果前一个节点释放锁的时候，才会被触发通知。这样就形成了一个监听链表。




**示例演示：**

```
package com.tuling.zookeeper.lock;

import org.I0Itec.zkclient.IZkDataListener;
import org.I0Itec.zkclient.ZkClient;

import java.util.List;
import java.util.stream.Collectors;

/**
 * @author Tommy
 * Created by Tommy on 2019/9/23
 **/
public class ZookeeperLock {
    private String server = "192.168.0.149:2181";
    private ZkClient zkClient;
    private static final String rootPath = "/tuling-lock";

    public ZookeeperLock() {
        zkClient = new ZkClient(server, 5000, 20000);
        buildRoot();
    }

    // 构建根节点
    public void buildRoot() {
        if (!zkClient.exists(rootPath)) {
            zkClient.createPersistent(rootPath);
        }
```

```
    }

    public Lock lock(String lockId, long timeout) {
        Lock lockNode = createLockNode(lockId);
        lockNode = tryActiveLock(lockNode);// 尝试激活锁
        if (!lockNode.isActive()) {
            try {
                synchronized (lockNode) {
                    lockNode.wait(timeout);
                }
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
        if (!lockNode.isActive()) {
            throw new RuntimeException(" lock  timeout");
        }
        return lockNode;
    }

    public void unlock(Lock lock) {
        if (lock.isActive()) {
            zkClient.delete(lock.getPath());
        }
    }

    // 尝试激活锁
    private Lock tryActiveLock(Lock lockNode) {
        // 判断当前是否为最小节点
        List<String> list = zkClient.getChildren(rootPath)
                .stream()
                .sorted()
                .map(p -> rootPath + "/" + p)
                .collect(Collectors.toList());
        String firstNodePath = list.get(0);
        if (firstNodePath.equals(lockNode.getPath())) {
            lockNode.setActive(true);
        } else {
            String upNodePath = list.get(list.indexOf(lockNode.getPath())
- 1);
            zkClient.subscribeDataChanges(upNodePath, new
IZkDataListener() {
                @Override
                public void handleDataChange(String dataPath, Object data)
throws Exception {

                }

                @Override
                public void handleDataDeleted(String dataPath) throws
Exception {
```

```
                    // 事件处理 与心跳 在同一个线程，如果Debug时占用太多时间，将导致本节点被删除，从而影响锁逻辑。
                    System.out.println("节点删除:" + dataPath);
                     Lock lock = tryActiveLock(lockNode);
                    synchronized (lockNode) {
                        if (lock.isActive()) {
                            lockNode.notify();
                        }
                    }
                    zkClient.unsubscribeDataChanges(upNodePath, this);
                }
            });
        }
        return lockNode;
    }


    public Lock createLockNode(String lockId) {
        String nodePath = zkClient.createEphemeralSequential(rootPath + "/" + lockId, "lock");
        return new Lock(lockId, nodePath);
    }
}
```