zookeeper

- zookeeper 事件监听机制
- zookeeper 集群搭建
- 一致性协议:zab协议
- zookeeper的leader选举
- observer角色及其配置
- zookeeperAPI连接集群

1.zookeeper 事件监听机制

1.1 watcher概念

zookeeper提供了数据的发布/订阅功能，多个订阅者可同时监听某一特定主题对象，当该主题对象的自身状态发生变化时(例如节点内容改变、节点下的子节点列表改变等)，会实时、主动通知所有订阅者

zookeeper采用了Watcher机制实现数据的发布/订阅功能。该机制在被订阅对象发生变化时会异步通知客户端，因此客户端不必在Watcher注册后轮询阻塞，从而减轻了客户端压力。

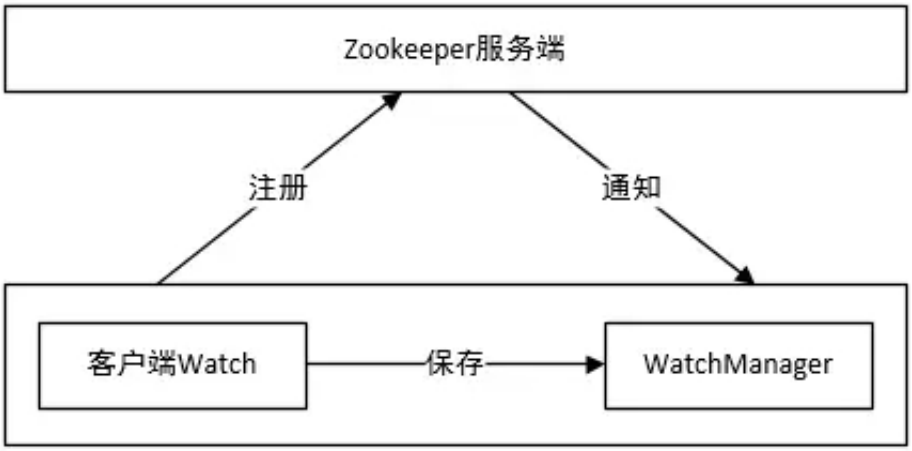
watcher机制实际上与观察者模式类似，也可看作是一种观察者模式在分布式场景下的实现方式。

1.2 watcher架构

Watcher实现由三个部分组成：

- Zookeeper服务端
- Zookeeper客户端
- 客户端的ZKWatchManager对象

客户端首先将Watcher注册到服务端，同时将Watcher对象保存到客户端的Watch管理器中。当ZooKeeper服务端监听的数据状态发生变化时，服务端会主动通知客户端，接着客户端的Watch管理器会触发相关Watcher来回调相应处理逻辑，从而完成整体的数据发布/订阅流程。

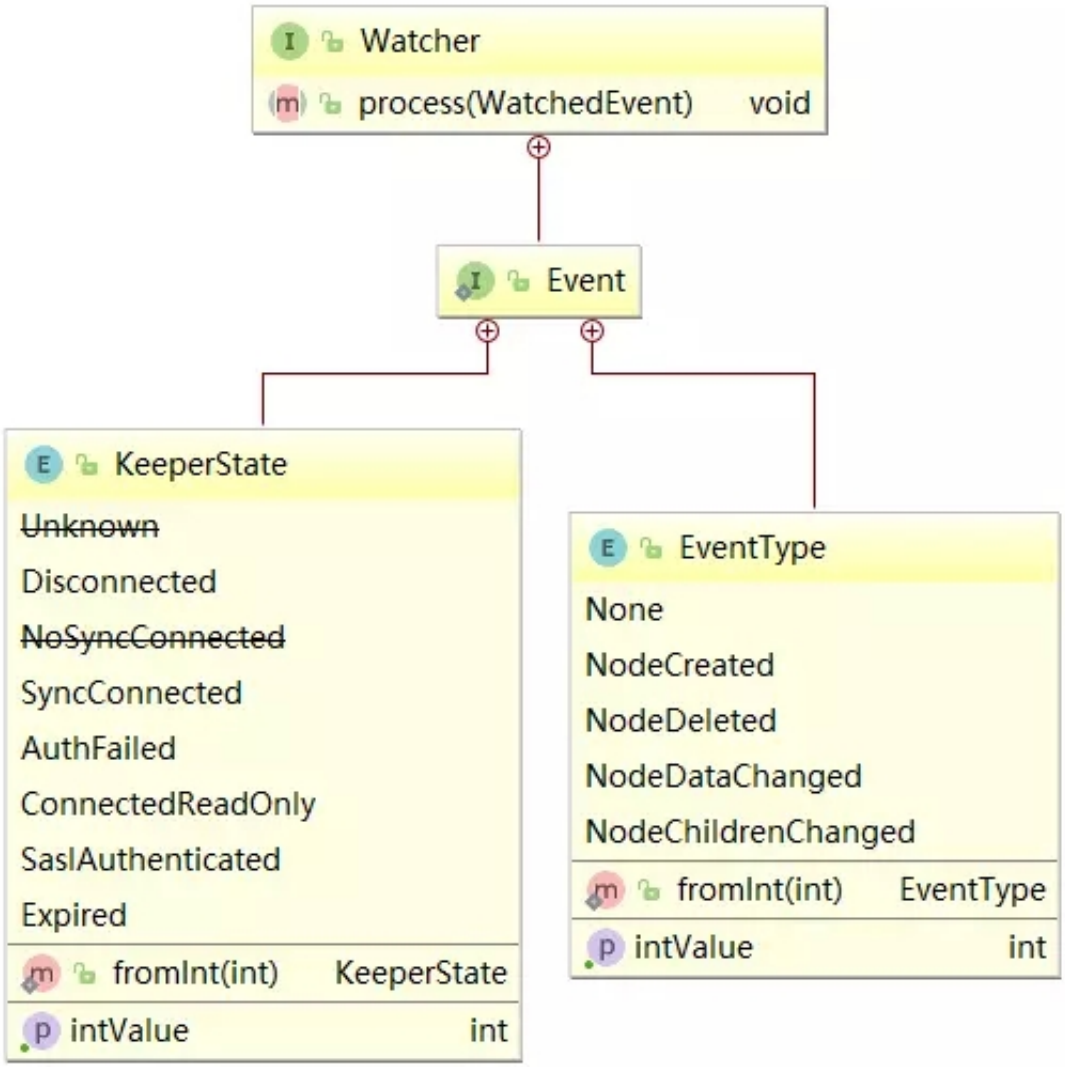


1.3 watcher特性

特性	说明
一次性	watcher是一次性的，一旦被触发就会移除，再次使用时需要重新注册
客户端顺序回调	watcher回调是顺序串行化执行的，只有回调后客户端才能看到最新的数据状态。一个watcher回调逻辑不应该太多，以免影响别的watcher执行
轻量级	WatchEvent是最小的通信单元，结构上只包含通知状态、事件类型和节点路径，并不会告诉数据节点变化前后的具体内容；
时效性	watcher只有在当前session彻底失效时才会无效，若在session有效期内快速重连成功，则watcher依然存在，仍可接收到通知；

1.4 watcher接口设计

Watcher是一个接口，任何实现了Watcher接口的类就是一个新的Watcher。Watcher内部包含了两个枚举类：KeeperState、EventType



- **Watcher**通知状态**(KeeperState)**

  KeeperState是客户端与服务端连接状态发生变化时对应的通知类型。路径为org.apache.zookeeper.Watcher.Event.KeeperState，是一个枚举类，其枚举属性如下：

枚举属性	说明
SyncConnected	客户端与服务器正常连接时
Disconnected	客户端与服务器断开连接时
Expired	会话session失效时
AuthFailed	身份认证失败时

- **Watcher**事件类型**(EventType)**

EventType是数据节点(znode)发生变化时对应的通知类型。EventType变化时KeeperState永远处于SyncConnected通知状态下；当KeeperState发生变化时，EventType永远为None。其路径为org.apache.zookeeper.Watcher.Event.EventType，是一个枚举类，枚举属性如下：

枚举属性	说明
None	无
NodeCreated	Watcher监听的数据节点被创建时
NodeDeleted	Watcher监听的数据节点被删除时
NodeDataChanged	Watcher监听的数据节点内容发生变更时(无论内容数据是否变化)
NodeChildrenChanged	Watcher监听的数据节点的子节点列表发生变更时

注：客户端接收到的相关事件通知中只包含状态及类型等信息，不包括节点变化前后的具体内容，变化前的数据需业务自身存储，变化后的数据需调用get等方法重新获取；

1.5 捕获相应的事件

上面讲到zookeeper客户端连接的状态和zookeeper对znode节点监听的事件类型，下面我们来讲解如何建立zookeeper的watcher监听。在zookeeper中采用zk.getChildren(path, watch)、zk.exists(path, watch)、zk.getData(path, watcher, stat)这样的方式为某个znode注册监听。

下表以node-x节点为例，说明调用的注册方法和可监听事件间的关系：

注册方式	**Created**	**ChildrenChanged**	**Changed**	**Deleted**
zk.exists(“/node-x”,watcher)	可监控		可监控	可监控
zk.getData(“/node-x”,watcher)			可监控	可监控
zk.getChildren(“/node-x”,watcher)		可监控		可监控

1.6 注册**watcher**的方法

1.6.1 客服端与服务器的连接状态

```
KeeperState 通知状态
SyncConnected:客户端与服务器正常连接时
Disconnected:客户端与服务器断开连接时
Expired:会话session失效时
AuthFailed:身份认证失败时

事件类型为:None
```

案例:

```
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import java.util.concurrent.CountDownLatch;

public class ZKConnectionWatcher implements Watcher {
    // 计数器对象
    static CountDownLatch countDownLatch = new CountDownLatch(1);
    // 连接对象
    static ZooKeeper zooKeeper;

    @Override
    public void process(WatchedEvent event) {
        try {
            // 事件类型
            if (event.getType() == Event.EventType.None) {
                if (event.getState() == Event.KeeperState.SyncConnected)
{
                    System.out.println("连接创建成功!");
                    countDownLatch.countDown();
                } else if (event.getState() ==
Event.KeeperState.Disconnected) {
                    System.out.println("断开连接! ");
                } else if (event.getState() == Event.KeeperState.Expired)
{
                    System.out.println("会话超时!");
                    zooKeeper = new ZooKeeper("192.168.60.130:2181",
5000, new ZKConnectionWatcher());
                } else if (event.getState() ==
Event.KeeperState.AuthFailed) {
                    System.out.println("认证失败! ");
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }


    public static void main(String[] args) {
```

```
        try {
            zooKeeper = new ZooKeeper("192.168.60.130:2181", 5000, new
ZKConnectionWatcher());
            // 阻塞线程等待连接的创建
            countDownLatch.await();
            // 会话id
            System.out.println(zooKeeper.getSessionId());
            // 添加授权用户

zooKeeper.addAuthInfo("digest1","itcast1:1234561".getBytes());
            byte [] bs=zooKeeper.getData("/node1",false,null);
            System.out.println(new String(bs));
            Thread.sleep(50000);
            zooKeeper.close();
            System.out.println("结束");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

1.6.1 检查节点是否存在

```
// 使用连接对象的监视器
exists(String path, boolean b)
// 自定义监视器
exists(String path, Watcher w)

// NodeCreated:节点创建
// NodeDeleted:节点删除
// NodeDataChanged:节点内容发生变化
```

- **path**- znode路径。
- **b**- 是否使用连接对象中注册的监视器。
- **w**-监视器对象。

案例:

```java
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.io.IOException;
import java.util.concurrent.CountDownLatch;

public class ZKWatcherExists {

    String IP = "192.168.60.130:2181";
    ZooKeeper zooKeeper = null;

    @Before
    public void before() throws IOException, InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        // 连接zookeeper客户端
        zooKeeper = new ZooKeeper(IP, 6000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("连接对象的参数!");
                // 连接成功
                if (event.getState() == Event.KeeperState.SyncConnected)
{
                    countDownLatch.countDown();
                }
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        countDownLatch.await();
    }

    @After
    public void after() throws InterruptedException {
        zooKeeper.close();
    }
```

```
    @Test
    public void watcherExists1() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:使用连接对象中的watcher
        zooKeeper.exists("/watcher1", true);
        Thread.sleep(50000);
        System.out.println("结束");
    }


    @Test
    public void watcherExists2() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:自定义watcher对象
        zooKeeper.exists("/watcher1", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("自定义watcher");
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherExists3() throws KeeperException,
InterruptedException {
        // watcher一次性
        Watcher watcher = new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("自定义watcher");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());
                    zooKeeper.exists("/watcher1", this);

                } catch (Exception ex) {
```

```
                    ex.printStackTrace();
                }
            }
        };
        zooKeeper.exists("/watcher1", watcher);
        Thread.sleep(80000);
        System.out.println("结束");
    }


    @Test
    public void watcherExists4() throws KeeperException,
InterruptedException {
        // 注册多个监听器对象
        zooKeeper.exists("/watcher1", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("1");
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        zooKeeper.exists("/watcher1", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("2");
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        Thread.sleep(80000);
        System.out.println("结束");
    }
}
```

1.6.2 查看节点

```
// 使用连接对象的监视器
getData(String path, boolean b, Stat stat)
// 自定义监视器
getData(String path, Watcher w, Stat stat)

// NodeDeleted:节点删除
// NodeDataChanged:节点内容发生变化
```

- **path**- znode路径。
- **b**- 是否使用连接对象中注册的监视器。
- **w**-监视器对象。
- **stat**- 返回znode的元数据。

案例：

```
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.apache.zookeeper.data.Stat;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.io.IOException;
import java.util.concurrent.CountDownLatch;

public class ZKWatcherGetData {

    String IP = "192.168.60.130:2181";
    ZooKeeper zooKeeper = null;

    @Before
    public void before() throws IOException, InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(1);
        // 连接zookeeper客户端
        zooKeeper = new ZooKeeper(IP, 6000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("连接对象的参数!");
                // 连接成功
                if (event.getState() == Event.KeeperState.SyncConnected)
{
                    countDownLatch.countDown();
                }
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        countDownLatch.await();
    }

    @After
    public void after() throws InterruptedException {
        zooKeeper.close();

    }
```

```
    @Test
    public void watcherGetData1() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:使用连接对象中的watcher
        zooKeeper.getData("/watcher2", true, null);
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherGetData2() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:自定义watcher对象
        zooKeeper.getData("/watcher2", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("自定义watcher");
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        }, null);
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherGetData3() throws KeeperException,
InterruptedException {
        // 一次性
        Watcher watcher = new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("自定义watcher");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());

                    if(event.getType()==Event.EventType.NodeDataChanged)
```

```
{
                        zooKeeper.getData("/watcher2", this, null);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        };
        zooKeeper.getData("/watcher2", watcher, null);
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherGetData4() throws KeeperException,
InterruptedException {
        // 注册多个监听器对象
        zooKeeper.getData("/watcher2", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("1");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());
                    if(event.getType()==Event.EventType.NodeDataChanged)
{
                        zooKeeper.getData("/watcher2", this, null);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        },null);
        zooKeeper.getData("/watcher2", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("2");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());
                    if(event.getType()==Event.EventType.NodeDataChanged)
```

```
{
                        zooKeeper.getData("/watcher2", this, null);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        },null);
        Thread.sleep(50000);
        System.out.println("结束");
    }
}
```

1.6.3 查看子节点

```
// 使用连接对象的监视器
getChildren(String path, boolean b)
// 自定义监视器
getChildren(String path, Watcher w)

// NodeChildrenChanged:子节点发生变化
// NodeDeleted:节点删除
```

- **path**- znode路径。
- **b**- 是否使用连接对象中注册的监视器。
- **w**-监视器对象。

案例:

```
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import java.io.IOException;
import java.util.List;
import java.util.concurrent.CountDownLatch;

public class ZKWatcherGetChild {
    String IP = "192.168.60.130:2181";
    ZooKeeper zooKeeper = null;

    @Before
    public void before() throws IOException, InterruptedException {
        CountDownLatch connectedSemaphore = new CountDownLatch(1);
        // 连接zookeeper客户端
        zooKeeper = new ZooKeeper(IP, 6000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("连接对象的参数!");
                // 连接成功
                if (event.getState() == Event.KeeperState.SyncConnected)
{
                    connectedSemaphore.countDown();
                }
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        connectedSemaphore.await();
    }

    @After
    public void after() throws InterruptedException {
        zooKeeper.close();
    }
```

```
    @Test
    public void watcherGetChild1() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:使用连接对象中的watcher
        zooKeeper.getChildren("/watcher3", true);
        Thread.sleep(50000);
        System.out.println("结束");
    }


    @Test
    public void watcherGetChild2() throws KeeperException,
InterruptedException {
        // arg1:节点的路径
        // arg2:自定义watcher
        zooKeeper.getChildren("/watcher3", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                System.out.println("自定义watcher");
                System.out.println("path=" + event.getPath());
                System.out.println("eventType=" + event.getType());
            }
        });
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherGetChild3() throws KeeperException,
InterruptedException {
        // 一次性
        Watcher watcher = new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("自定义watcher");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());

                    if (event.getType() ==
```

```
Event.EventType.NodeChildrenChanged) {
                        zooKeeper.getChildren("/watcher3", this);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        };
        zooKeeper.getChildren("/watcher3", watcher);
        Thread.sleep(50000);
        System.out.println("结束");
    }

    @Test
    public void watcherGetChild4() throws KeeperException,
InterruptedException {
        // 多个监视器对象
        zooKeeper.getChildren("/watcher3", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("1");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());
                    if (event.getType() ==
Event.EventType.NodeChildrenChanged) {
                        zooKeeper.getChildren("/watcher3", this);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        });
        zooKeeper.getChildren("/watcher3", new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                try {
                    System.out.println("2");
                    System.out.println("path=" + event.getPath());
                    System.out.println("eventType=" + event.getType());
                    if (event.getType() ==
```

```
Event.EventType.NodeChildrenChanged) {
                        zooKeeper.getChildren("/watcher3", this);
                    }
                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        });
        Thread.sleep(50000);
        System.out.println("结束");
    }
}
```

1.7 配置中心案例

工作中有这样的一个场景: 数据库用户名和密码信息放在一个配置文件中，应用读取该配置文件，配置文件信息放入缓存。

若数据库的用户名和密码改变时候，还需要重新加载缓存，比较麻烦，通过ZooKeeper可以轻松完成，当数据库发生变化时自动完成缓存同步。

设计思路：

1. 连接zookeeper服务器
2. 读取zookeeper中的配置信息，注册watcher监听器，存入本地变量
3. 当zookeeper中的配置信息发生变化时，通过watcher的回调方法捕获数据变化事件
4. 重新获取配置信息

案例:

```
import java.util.concurrent.CountDownLatch;
import com.itcast.watcher.ZKConnectionWatcher;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.EventType;
import org.apache.zookeeper.ZooKeeper;

public class MyConfigCenter implements Watcher {

    //  zk的连接串
    String IP = "192.168.60.130:2181";
    //  计数器对象
    CountDownLatch countDownLatch = new CountDownLatch(1);
    // 连接对象
    static ZooKeeper zooKeeper;

    // 用于本地化存储配置信息
    private String url;
    private String username;
    private String password;

    @Override
    public void process(WatchedEvent event) {
        try {
            // 捕获事件状态
            if (event.getType() == Event.EventType.None) {
                if (event.getState() == Event.KeeperState.SyncConnected)
{
                    System.out.println("连接成功");
                    countDownLatch.countDown();
                } else if (event.getState() ==
Event.KeeperState.Disconnected) {
                    System.out.println("连接断开!");
                } else if (event.getState() == Event.KeeperState.Expired)
{
                    System.out.println("连接超时!");
                    // 超时后服务器端已经将连接释放，需要重新连接服务器端
                    zooKeeper = new ZooKeeper("192.168.60.130:2181",
6000,

                            new ZKConnectionWatcher());
```

```
                } else if (event.getState() ==
Event.KeeperState.AuthFailed) {
                    System.out.println("验证失败!");
                }
                // 当配置信息发生变化时
            } else if (event.getType() == EventType.NodeDataChanged) {
                initValue();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 构造方法
    public MyConfigCenter() {
        initValue();
    }


    // 连接zookeeper服务器，读取配置信息
    public void initValue() {
        try {
            // 创建连接对象
            zooKeeper = new ZooKeeper(IP, 5000, this);
            // 阻塞线程，等待连接的创建成功
            countDownLatch.await();
            // 读取配置信息
            this.url = new String(zooKeeper.getData("/config/url", true,
null));
            this.username = new
String(zooKeeper.getData("/config/username", true, null));
            this.password = new
String(zooKeeper.getData("/config/password", true, null));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    public static void main(String[] args) {
        try {
```

```
            MyConfigCenter myConfigCenter = new MyConfigCenter();
            for (int i = 1; i <= 20; i++) {
                Thread.sleep(5000);
                System.out.println("url:"+myConfigCenter.getUrl());

System.out.println("username:"+myConfigCenter.getUsername());

System.out.println("password:"+myConfigCenter.getPassword());

System.out.println("########################################");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

1.8 生成分布式唯一ID

在过去的单库单表型系统中，通常可以使用数据库字段自带的auto_increment属性来自动为每条记录生成一个唯一的ID。但是分库分表后，就无法在依靠数据库的auto_increment属性来唯一标识一条记录了。此时我们就可以用zookeeper在分布式环境下生成全局唯一ID。

设计思路：

1.连接zookeeper服务器

2.指定路径生成临时有序节点

3.取序列号及为分布式环境下的唯一ID

案例:

```
import java.util.concurrent.CountDownLatch;
import com.itcast.watcher.ZKConnectionWatcher;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.ZooKeeper;

public class GloballyUniqueId implements Watcher {
    //  zk的连接串
    String IP = "192.168.60.130:2181";
    //  计数器对象
    CountDownLatch countDownLatch = new CountDownLatch(1);
    //  用户生成序号的节点
    String defaultPath = "/uniqueId";
    //  连接对象
    ZooKeeper zooKeeper;

    @Override
    public void process(WatchedEvent event) {
        try {
            // 捕获事件状态
            if (event.getType() == Watcher.Event.EventType.None) {
                if (event.getState() ==
Watcher.Event.KeeperState.SyncConnected) {
                    System.out.println("连接成功");
                    countDownLatch.countDown();
                } else if (event.getState() ==
Watcher.Event.KeeperState.Disconnected) {
                    System.out.println("连接断开!");
                } else if (event.getState() ==
Watcher.Event.KeeperState.Expired) {
                    System.out.println("连接超时!");
                    // 超时后服务器端已经将连接释放，需要重新连接服务器端
                    zooKeeper = new ZooKeeper(IP, 6000,
                            new ZKConnectionWatcher());
                } else if (event.getState() ==
Watcher.Event.KeeperState.AuthFailed) {

                    System.out.println("验证失败!");
```

```
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 构造方法
    public GloballyUniqueId() {
        try {
            //打开连接
            zooKeeper = new ZooKeeper(IP, 5000, this);
            // 阻塞线程，等待连接的创建成功
            countDownLatch.await();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    // 生成id的方法
    public String getUniqueId() {
        String path = "";
        try {
            //创建临时有序节点
            path = zooKeeper.create(defaultPath, new byte[0],
Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        // /uniqueId0000000001
        return path.substring(9);
    }

    public static void main(String[] args) {
        GloballyUniqueId globallyUniqueId = new GloballyUniqueId();
        for (int i = 1; i <= 5; i++) {
            String id = globallyUniqueId.getUniqueId();
            System.out.println(id);
        }
    }
}
```

1.9 分布式锁

分布式锁有多种实现方式，比如通过数据库、redis都可实现。作为分布式协同工具ZooKeeper，当然也有着标准的实现方式。下面介绍在zookeeper中如何实现排他锁。

设计思路：

1.每个客户端往/Locks下创建临时有序节点/Locks/Lock
000000001

2.客户端取得/Locks下子节点，并进行排序，判断排在最前面的是否为自己，如果自己的锁节点在第一位，代表获取锁成功

3.如果自己的锁节点不在第一位，则监听自己前一位的锁节点。例如，自己锁节点
Lock 000000001

4.当前一位锁节点（Lock
000000002）的逻辑

5.监听客户端重新执行第2步逻辑，判断自己是否获得了锁

案例:

```
import org.apache.zookeeper.*;
import org.apache.zookeeper.data.Stat;
import java.io.IOException;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;

public class MyLock {
    //  zk的连接串
    String IP = "192.168.60.130:2181";
    //  计数器对象
    CountDownLatch countDownLatch = new CountDownLatch(1);
    //ZooKeeper配置信息
    ZooKeeper zooKeeper;
    private static final String LOCK_ROOT_PATH = "/Locks";
    private static final String LOCK_NODE_NAME = "Lock_";
    private String lockPath;

    // 打开zookeeper连接
    public MyLock() {
        try {
            zooKeeper = new ZooKeeper(IP, 5000, new Watcher() {
                @Override
                public void process(WatchedEvent event) {
                    if (event.getType() == Event.EventType.None) {
                        if (event.getState() ==
Event.KeeperState.SyncConnected) {
                            System.out.println("连接成功!");
                            countDownLatch.countDown();
                        }
                    }
                }
            });
            countDownLatch.await();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }


    //获取锁
```

```
    public void acquireLock() throws Exception {
        //创建锁节点
        createLock();
        //尝试获取锁
        attemptLock();
    }

    //创建锁节点
    private void createLock() throws Exception {
        //判断Locks是否存在，不存在创建
        Stat stat = zooKeeper.exists(LOCK_ROOT_PATH, false);
        if (stat == null) {
            zooKeeper.create(LOCK_ROOT_PATH, new byte[0],
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
        }
        // 创建临时有序节点
        lockPath = zooKeeper.create(LOCK_ROOT_PATH + "/" +
LOCK_NODE_NAME, new byte[0], ZooDefs.Ids.OPEN_ACL_UNSAFE,
CreateMode.EPHEMERAL_SEQUENTIAL);
        System.out.println("节点创建成功:" + lockPath);
    }

    //监视器对象，监视上一个节点是否被删除
    Watcher watcher = new Watcher() {
        @Override
        public void process(WatchedEvent event) {
            if (event.getType() == Event.EventType.NodeDeleted) {
                synchronized (this) {
                    notifyAll();
                }
            }
        }
    };

    //尝试获取锁
    private void attemptLock() throws Exception {
        // 获取Locks节点下的所有子节点
        List<String> list = zooKeeper.getChildren(LOCK_ROOT_PATH, false);
        // 对子节点进行排序
        Collections.sort(list);

        // /Locks/Lock_000000001
```

```
            int index =
list.indexOf(lockPath.substring(LOCK_ROOT_PATH.length() + 1));
            if (index == 0) {
                System.out.println("获取锁成功!");
                return;
            } else {
                // 上一个节点的路径
                String path = list.get(index - 1);
                Stat stat = zooKeeper.exists(LOCK_ROOT_PATH + "/" + path,
watcher);
                if (stat == null) {
                    attemptLock();
                } else {
                    synchronized (watcher) {
                        watcher.wait();
                    }
                    attemptLock();
                }
            }

        }

        //释放锁
        public void releaseLock() throws Exception {
                //删除临时有序节点
                zooKeeper.delete(this.lockPath,-1);
                zooKeeper.close();
                System.out.println("锁已经释放:"+this.lockPath);
        }

        public static void main(String[] args) {
            try {
                MyLock myLock = new MyLock();
                myLock.createLock();
            } catch (Exception ex) {
                ex.printStackTrace();
            }

        }
}
```

```
public class TicketSeller {
    private void sell(){
        System.out.println("售票开始");
        // 线程随机休眠数毫秒，模拟现实中的费时操作
        int sleepMillis = 5000;
        try {
            //代表复杂逻辑执行了一段时间
            Thread.sleep(sleepMillis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("售票结束");
    }
    public void sellTicketWithLock() throws Exception {
        MyLock lock = new MyLock();
        // 获取锁
        lock.acquireLock();
        sell();
        //释放锁
        lock.releaseLock();
    }
    public static void main(String[] args) throws Exception {
        TicketSeller ticketSeller = new TicketSeller();
        for(int i=0;i<10;i++){
            ticketSeller.sellTicketWithLock();
        }
    }
}
```

2.zookeeper 集群搭建

单机环境下，jdk、zookeeper 安装完毕，基于一台虚拟机，进行zookeeper伪集群搭建，zookeeper集群中包含3个节点，节点对外提供服务端口号分别为2181、2182、2183

1. 基于zookeeper-3.4.10复制三份zookeeper安装好的服务器文件，目录名称分别为zookeeper2181、zookeeper2182、zookeeper2183

```
cp -r zookeeper-3.4.10 zookeeper2181
cp -r zookeeper-3.4.10 zookeeper2182
cp -r zookeeper-3.4.10 zookeeper2183
```

2. 修改zookeeper2181服务器对应配置文件。

```
#服务器对应端口号
clientPort=2181
#数据快照文件所在路径
dataDir=/home/zookeeper/zookeeper2181/data
#集群配置信息
    #server.A=B:C:D
    #A：是一个数字，表示这个是服务器的编号
    #B：是这个服务器的ip地址
    #C：Zookeeper服务器之间的通信端口
    #D：Leader选举的端口
server.1=192.168.60.130:2287:3387
server.2=192.168.60.130:2288:3388
server.3=192.168.60.130:2289:3389
```

3. 在 上一步 dataDir 指定的目录下，创建 myid 文件，然后在该文件添加上一步 server 配置的对应 A 数字。

```
#zookeeper2181对应的数字为1
#/home/zookeeper/zookeeper2181/data目录下执行命令
echo "1" > myid
```

4. zookeeper2182、zookeeper2183参照步骤2/3进行相应配置
5. 分别启动三台服务器，检验集群状态

   登录命令：

```
./zkCli.sh -server 192.168.60.130:2181
./zkCli.sh -server 192.168.60.130:2182
./zkCli.sh -server 192.168.60.130:2183
```

3.一致性协议:zab协议

zab协议 的全称是 **Zookeeper Atomic Broadcast** （zookeeper原子广播）。zookeeper 是通过 zab协议来保证分布式事务的最终一致性

基于zab协议，zookeeper集群中的角色主要有以下三类，如下表所示：

zab广播模式工作原理，通过类似两阶段提交协议的方式解决数据一致性：

1. leader从客户端收到一个写请求
2. leader生成一个新的事务并为这个事务生成一个唯一的ZXID
3. leader将这个事务提议(propose)发送给所有的follows节点
4. follower节点将收到的事务请求加入到历史队列(history queue)中,并发送ack给leader
5. 当leader收到大多数follower（半数以上节点）的ack消息，leader会发送commit请求
6. 当follower收到commit请求时，从历史队列中将事务请求commit

4.zookeeper的leader选举

4.1 服务器状态

looking：寻找leader状态。当服务器处于该状态时，它会认为当前集群中没有leader，因此需要进入leader选举状态。

leading： 领导者状态。表明当前服务器角色是leader。

following： 跟随者状态。表明当前服务器角色是follower。

observing：观察者状态。表明当前服务器角色是observer。

4.2 服务器启动时期的leader选举

在集群初始化阶段，当有一台服务器server1启动时，其单独无法进行和完成leader选举，当第二台服务器server2启动时，此时两台机器可以相互通信，每台机器都试图找到leader，于是进入leader选举过程。选举过程如下:

1. 每个server发出一个投票。由于是初始情况，server1和server2都会将自己作为leader服务器来进行投票，每次投票会包含所推举的服务器的myid和zxid，使用(myid, zxid)来表示，此时server1的投票为(1, 0)，server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。
2. 集群中的每台服务器接收来自集群中各个服务器的投票。
3. 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投票进行pk，pk规则如下
    - 优先检查zxid。zxid比较大的服务器优先作为leader。
    - 如果zxid相同，那么就比较myid。myid较大的服务器作为leader服务器。

    对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的zxid，均为0，再比较myid，此时server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。
4. 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于server1、server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了leader
5. 改变服务器状态。一旦确定了leader，每个服务器就会更新自己的状态，如果是follower，那么就变更为following，如果是leader，就变更为leading。

4.3 服务器运行时期的**Leader**选举

在zookeeper运行期间，leader与非leader服务器各司其职，即便当有非leader服务器宕机或新加入，此时也不会影响leader，但是一旦leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮leader选举，其过程和启动时期的Leader选举过程基本一致。

假设正在运行的有server1、server2、server3三台服务器，当前leader是server2，若某一时刻leader挂了，此时便开始Leader选举。选举过程如下:

1. 变更状态。leader挂后，余下的服务器都会将自己的服务器状态变更为looking，然后开始进入leader选举过程。
2. 每个server会发出一个投票。在运行期间，每个服务器上的zxid可能不同，此时假定server1的zxid为122，server3的zxid为122，在第一轮投票中，server1和server3都会投自己，产生投票(1, 122)，(3, 122)，然后各自将投票发送给集群中所有机器。
3. 接收来自各个服务器的投票。与启动时过程相同
4. 处理投票。与启动时过程相同，此时，server3将会成为leader。

5. 统计投票。与启动时过程相同。
6. 改变服务器的状态。与启动时过程相同。

5.observer角色及其配置

observer角色特点：

1. 不参与集群的leader选举
    1. 不参与集群中写数据时的ack反馈

    为了使用observer角色，在任何想变成observer角色的配置文件中加入如下配置：

```
peerType=observer
```

并在所有server的配置文件中，配置成observer模式的server的那行配置追加:observer，例如：

```
server.3=192.168.60.130:2289:3389:observer
```

6.zookeeperAPI连接集群

```
ZooKeeper(String connectionString, int sessionTimeout, Watcher watcher)
```

- **connectionString** - zooKeeper集合主机。
- **sessionTimeout** - 会话超时（以毫秒为单位）。
- **watcher** - 实现“监视器”界面的对象。ZooKeeper集合通过监视器对象返回连接状态。

案例:

```
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import java.util.concurrent.CountDownLatch;

public class ZookeeperConnection {
    public static void main(String[] args) {
        try {
            // 计数器对象
            CountDownLatch countDownLatch=new CountDownLatch(1);
            // arg1:服务器的ip和端口
            // arg2:客户端与服务器之间的会话超时时间  以毫秒为单位的
            // arg3:监视器对象
            ZooKeeper zooKeeper=new
ZooKeeper("192.168.60.130:2181,192.168.60.130:2182,192.168.60.130:2183",
5000, new Watcher() {
                @Override
                public void process(WatchedEvent event) {
                    if(event.getState()==Event.KeeperState.SyncConnected)
{
                        System.out.println("连接创建成功!");
                        countDownLatch.countDown();
                    }
                }
            });
            // 主线程阻塞等待连接对象的创建成功
            countDownLatch.await();
            // 会话编号
            System.out.println(zooKeeper.getSessionId());
            zooKeeper.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```