

一:什么是事务?

1.1)什么是事物?

事务是逻辑上的一组执行单元, 要么都执行, 要么都不执行.

eg: 张三给李四转1000元钱, 涉及到二个操作

张三的账户-1000 李四的账户+1000

上述二个步骤要么一起成功, 要么一起失败. 不存在张三的钱扣了, 李四没收到转账.

1.2)事物的特性(ACID)



什么是ACID

ACID是指数据库管理系统DBMS中事物所具有四个特性

eg:在数据库系统中, 一个事务由一系列的数据库操作组成一个完整的逻辑过程, 比如银行转账, 从原账户扣除金额, 目标账户增加金额

①:atomicity【原子性】

原子性表现为**操作不能被分割**, 那么这二个操作 要么**同时完成**, 要么就**全部不完成**, 若事务出错了, 那么事务就会回滚, 好像什么 都 没有发生过

②:Consistency【一致性】

一致性也比较好理解, 也就是说数据库要一直处于一致的状态, 事务开始前是一个一致状态, 事务结束后是另一个一致状态, **事务将数据库从一个一致状态转移到另一个一致状态**

③:Isolation【隔离性】

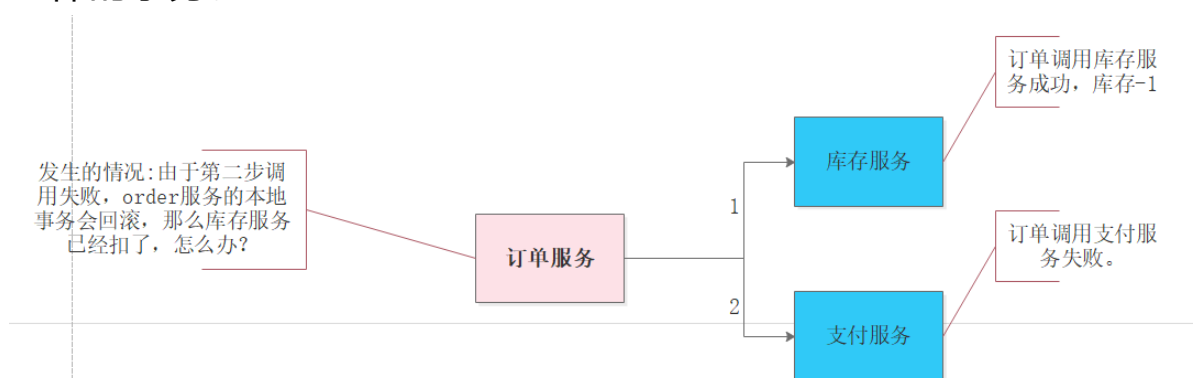
所谓的独立性就是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务还未提交，它所访问的数据就不受未提交事务的影响。换句话说，一个事务的影响在该事务提交前对其它事务是不可见的

④:Durability【持久性】

若事务已经提交了，那么就回在数据库中永久的保存下来

二:什么是分布式事务?

分布式事务：就是把各个节点上的微服务的本地事务 结合起来，看成一个整体的事务。



分布式事务组成部分:

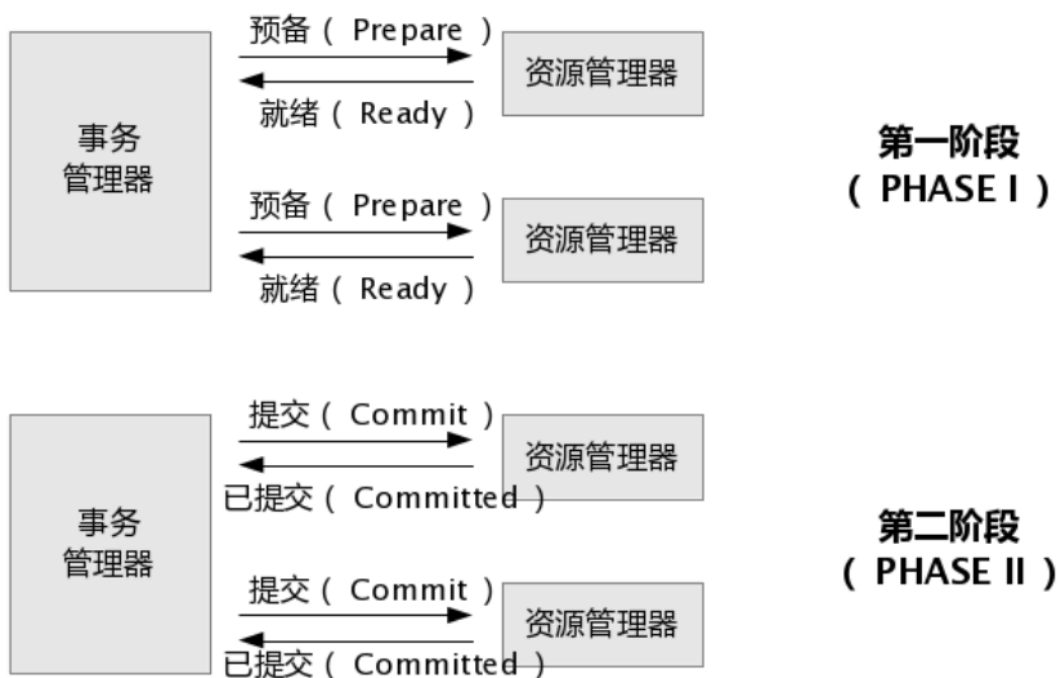
- ①:事务参与者(就是我们的一个一个的微服务, 比如订单服务,库存服务)
- ②:资源服务器(说白了就是我们的对应微服务连接的数据库 比如订单库, 库存库)
- ③:事务管理器 (用于决策各个事务参数者的提交和回滚)

二阶段提交:(2PC的视频地址, 这里不对二阶段深入讲解)

链接 :

<https://pan.baidu.com/s/1bKF9LIVqDiFAfGo1yl6rrQ>

提取码 : 862x



1.准备阶段：事务协调者(事务管理器)给每个参与者(资源管理器)发送 Prepare消息，每个参与者要么直接返回失败(如权限验证失败)，要么在本地执行事务，写本地的redo和undo日志，但不提交，到达一种“万事俱备，只欠东风”的状态。(关于每一个参与者在准备阶段具体做了什么目前我还没有参考到确切的资料，但是有一点非常确定：参与者在准备阶段完成了几乎所有正式提交的动作，有的材料上说是进行了“试探性的提交”，只保留了最后一步耗时非常短暂的正式提交操作给第二阶段执行。)

******第一阶段会占用数据库连接**

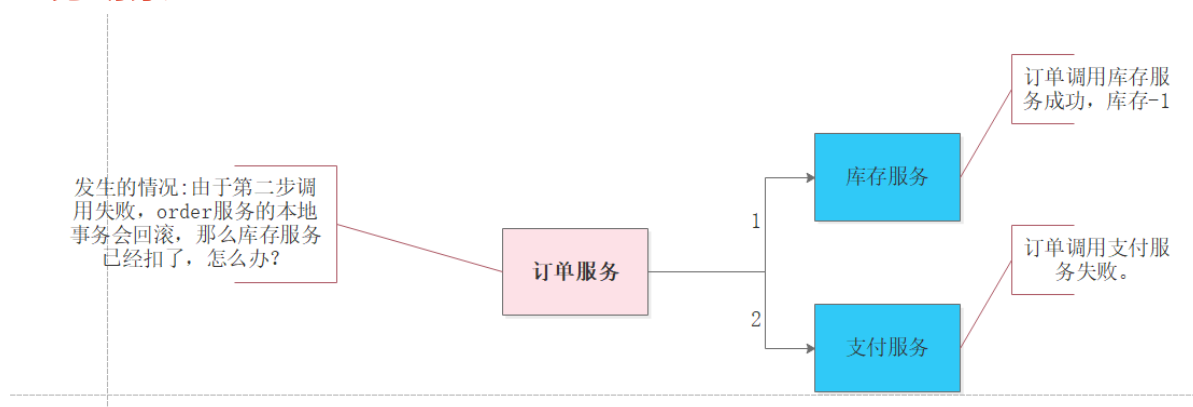
2.提交阶段：如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意:必须在最后阶段释放锁资源)

三:什么是Seata?

Seata 是一款开源的分布式事务解决方案，致力于提供高性能和简单易用的分布式事务服务。Seata 将为用户提供了 **AT**、TCC、SAGA 和 XA 事务模式，为用户打造一站式的分布式解决方案(**AT模式是阿里首推的模式，阿里云上有商用版本的GTS[Global Transaction service 全局事务服务]**)。

提示:

业务场景:



3.1)角色划分:

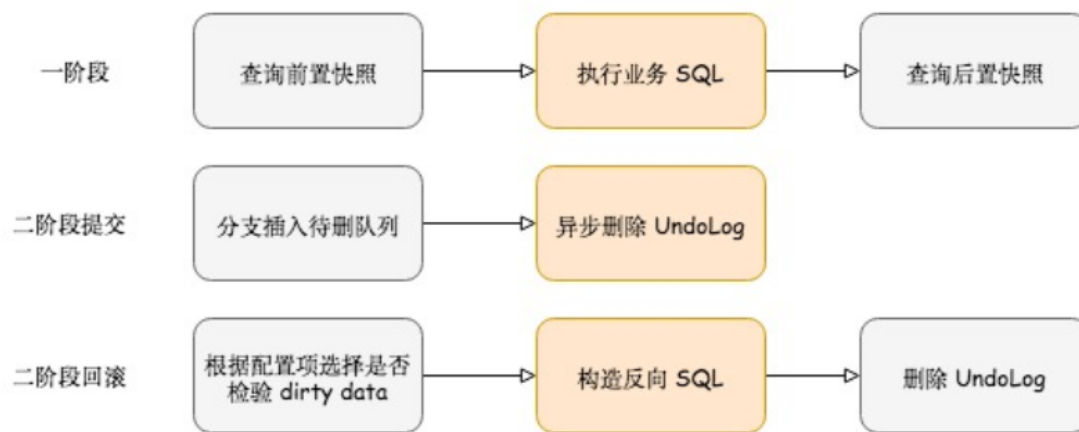
RM(ResourceManager 资源管理者)理解为 我们的一个一个的微服务 也叫做**事务的参与者**.

TM(TranactionManager 事务管理者) 也是我们的一个微服务，但是该微服务是一个**带头大哥，充当全局事务的发起者(决定了全局事务的开启，回滚，提交等)**

*****凡是我们的微服务中标注了@GlobalTransactional，那么该微服务就会被看出一个TM。我们业务场景中订单微服务就是一个事务发起者,同时也是一个RM**

TC(全局事务的协调者):这里就是我们的Seata-server，用来保存全局事务，分支事务，全局锁等记录，然后会通知各个RM进行回滚或者提交。

二:整体机制(**两阶段提交协议的演变**)



核心逻辑:

```
1 set autocommit = 0;
2 -- CTRL SHIFT + R
3 select * from product where product_id=1 for update;
4
5 UPDATE Product SET count = count - 1 WHERE product_id=1;
6
7 select * from product where product_id=1 for update;
8
9 commit;
10
11 对应的核心代码逻辑
12 connectionProxy.setAutoCommit(false);
13 return new LockRetryPolicy(connectionProxy.getTargetConnection()).execute(()
-> {
14     T result = executeAutoCommitFalse(args);
15     connectionProxy.commit();
16     return result;
17 });
```

2.1)工作原理

业务表: `product`

Field	Type	Key
id	bigint(20)	PRI
name	varchar(100)	
since	varchar(100)	

执行业务SQL `update product set name = 'GTS' where name = 'TXC';`

第一阶段:

1:解析 SQL: 得到 SQL 的类型 (UPDATE) , 表 (product) , 条件 (where id = '1') 等相关的信息。

2:查询前镜像: 根据解析得到的条件信息, 生成查询语句, 定位数据。

```
select id, name, since from product where name = 'TXC';
```

得到前镜像:

id	name	since
1	TXC	2014

3:执行业务 SQL: 更新这条记录的 name 为 'GTS'。

update product **set** name = 'GTS' **where** name = 'TXC';

4:查询后镜像: 根据前镜像的结果, 通过 主键 定位数据

```
select id, name, since from product where id = 1;
```

得到后镜像:

id	name	since
1	GTS	2014

5:插入回滚日志: 把前后镜像数据以及业务 SQL 相关的信息组成一条回滚日志记录, 插入到 UNDO_LOG 表中。

```
1 {
2   "branchId": 641789253,
3   "undoItems": [{
4     "afterImage": {
5       "rows": [{
6         "fields": [{
7           "name": "id",
8           "type": 4,
9           "value": 1
10        }, {
11          "name": "name",
12          "type": 12,
13          "value": "GTS"
14        }, {
15          "name": "since",
16          "type": 12,
17          "value": "2014"
18        }]
19      }],
20      "tableName": "product"
21    },
22    "beforeImage": {
23      "rows": [{
24        "fields": [{
25          "name": "id",
26          "type": 4,
27          "value": 1
28        }, {
29          "name": "name",
30          "type": 12,
31          "value": "TXC"
32        }, {
33          "name": "since",
34          "type": 12,
35          "value": "2014"
36        }]
37      }],
38      "tableName": "product"
39    },
40    "sqlType": "UPDATE"
41  ]],
42  "xid": "xid:xxx"
```

6:提交前，向 TC 注册分支：申请 product 表中，主键值等于 1 的记录的 全局锁。

7:本地事务提交：业务数据的更新和前面步骤中生成的 UNDO LOG 一并提交。

8:将本地事务提交的结果上报给 TC。

二阶段-回滚

1:收到 TC 的分支回滚请求，开启一个本地事务，执行如下操作

2:通过 XID 和 Branch ID 查找到相应的 UNDO LOG 记录。

3:数据校验：拿 UNDO LOG 中的后镜与当前数据进行比较，如果有不同，说明数据被当前全局事务之外的动作做了修改

4:根据 UNDO LOG 中的前镜像和业务 SQL 的相关信息生成并执行回滚的语句：

5:提交本地事务。并把本地事务的执行结果（即分支事务回滚的结果）上报给 TC。

二阶段-提交

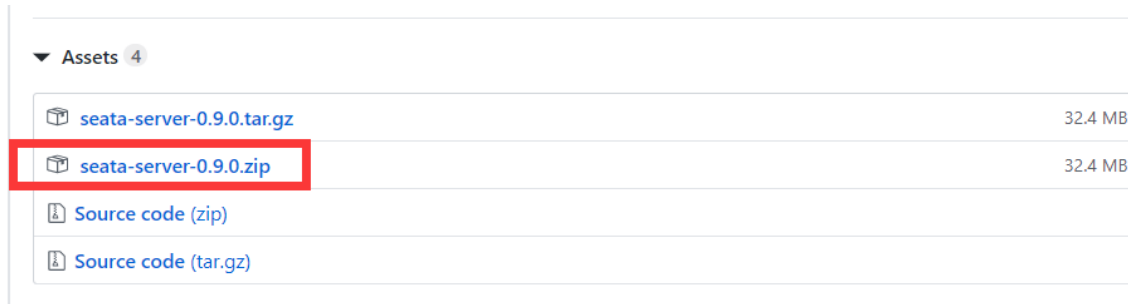
1:收到 TC 的分支提交请求，把请求放入一个异步任务的队列中，马上返回提交成功的结果给 TC。

2:异步任务阶段的分支提交请求将异步和批量地删除相应 UNDO LOG 记录。

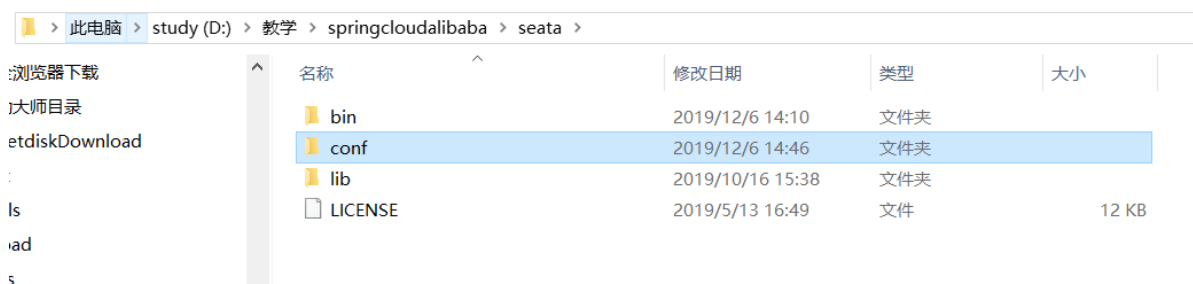
三:快速开始搭建Seata环境

3.1)Seata-server环境搭建

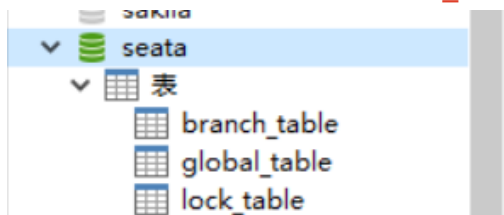
第一步:<https://github.com/seata/seata/releases> 下载seata-server包(目前最新的是0.9.0)



第二步: 解压我们的下载包seata-server包，解压的路径结构



第三步: 进入conf目录下 拿到db_store.sql脚本 , 然后再本地数据库创建一个seata的数据库,执行脚本db_store.sql。



第四步: 修改conf目录下的file.conf文件

此电脑 > study (D:) > 教学 > springcloudalibaba > seata > conf				
名称	修改日期	类型	大小	
META-INF	2019/10/16 15:38	文件夹		
db_store.sql	2019/10/16 15:38	SQL Text File	2 KB	
db_undo_log.sql	2019/10/16 15:38	SQL Text File	1 KB	
file.conf	2019/12/6 14:41	CONF 文件	4 KB	
file.conf.bak	2019/12/5 15:01	BAK 文件	4 KB	
logback.xml	2019/10/16 15:38	XML 文档	3 KB	
nacos-config.py	2019/10/16 15:38	PY 文件	1 KB	
nacos-config.sh	2019/10/16 15:38	Shell Script	1 KB	
nacos-config.txt	2019/12/6 14:53	文本文档	3 KB	
nacos-config.txt.bak	2019/12/6 14:45	BAK 文件	3 KB	
registry.conf	2019/12/5 15:04	CONF 文件	2 KB	
registry.conf.bak	2019/12/5 15:02	BAK 文件	2 KB	
seata.log	2019/12/6 14:46	文本文档	1 KB	

修改的节点: service节点

```
1 service {
2   #vgroup->rgroup
3   //修改全局事务分组
4   vgroup_mapping.prex_tx_group = "default"
5   #only support single node
6   #seata-server的连接地址
7   default.grouplist = "127.0.0.1:8091"
8   #degrade current not support
9   enableDegrade = false
10  #disable
11  disable = false
12  #unit ms,s,m,h,d represents milliseconds, seconds, minutes, hours, days, de
13  fault permanent
14  max.commit.retry.timeout = "-1"
15  max.rollback.retry.timeout = "-1"
16 }
```

修改store节点:

```
1 store {
2   ## store mode: file、db
3   //存储模式 使用db
4   mode = "db"
```

```

5  file{//file的不要改
6
7  }
8  db {
9  ## the implement of javax.sql.DataSource, such as DruidDataSource(druid)/BasicDataSource(dbcp) etc.
10 //数据源的类型
11 datasource = "druid"
12 ## mysql/oracle/h2/oceanbase etc.
13 db-type = "mysql"
14 driver-class-name = "com.mysql.jdbc.Driver"
15 //你seata库的地址
16 url = "jdbc:mysql://localhost:3306/seata"
17 user = "root"
18 password = "Zw726515"
19 min-conn = 1
20 max-conn = 3
21 global.table = "global_table"
22 branch.table = "branch_table"
23 lock-table = "lock_table"
24 query-limit = 100
25 }

```

第五步：修改conf目录下的register.conf文件

此电脑 > study (D:) > 教学 > springcloudalibaba > seata > conf

名称	修改日期	类型	大小
META-INF	2019/10/16 15:38	文件夹	
db_store.sql	2019/10/16 15:38	SQL Text File	2 KB
db_undo_log.sql	2019/10/16 15:38	SQL Text File	1 KB
file.conf	2019/12/6 14:41	CONF 文件	4 KB
file.conf.bak	2019/12/5 15:01	BAK 文件	4 KB
logback.xml	2019/10/16 15:38	XML 文档	3 KB
nacos-config.py	2019/10/16 15:38	PY 文件	1 KB
nacos-config.sh	2019/10/16 15:38	Shell Script	1 KB
nacos-config.txt	2019/12/6 14:53	文本文档	3 KB
nacos-config.txt.bak	2019/12/6 14:45	BAK 文件	3 KB
registry.conf	2019/12/5 15:04	CONF 文件	2 KB
registry.conf.bak	2019/12/5 15:02	BAK 文件	2 KB
seata.log	2019/12/6 14:46	文本文档	1 KB

修改registry节点的类型为nacos

```

1 registry {
2 # file 、 nacos 、 eureka、 redis、 zk、 consul、 etcd3、 sofa
3 type = "nacos"
4
5 nacos {
6 serverAddr = "localhost:8848"

```

```
7 namespace = ""
8 cluster = "default"
9 }
10   . . . .
11   . . . .
12   . . . .
13 }
```

修改config节点的type改为nacos

```
1 config {
2   # file、nacos 、apollo、zk、consul、etcd3
3   type = "nacos"
4
5   nacos {
6     serverAddr = "localhost:8848"
7     namespace = ""
8     cluster = "default"
9   }
10   . . . . .
11   . . . . .
12 }
```

第六步:修改conf文件下的nacos-config.txt文件

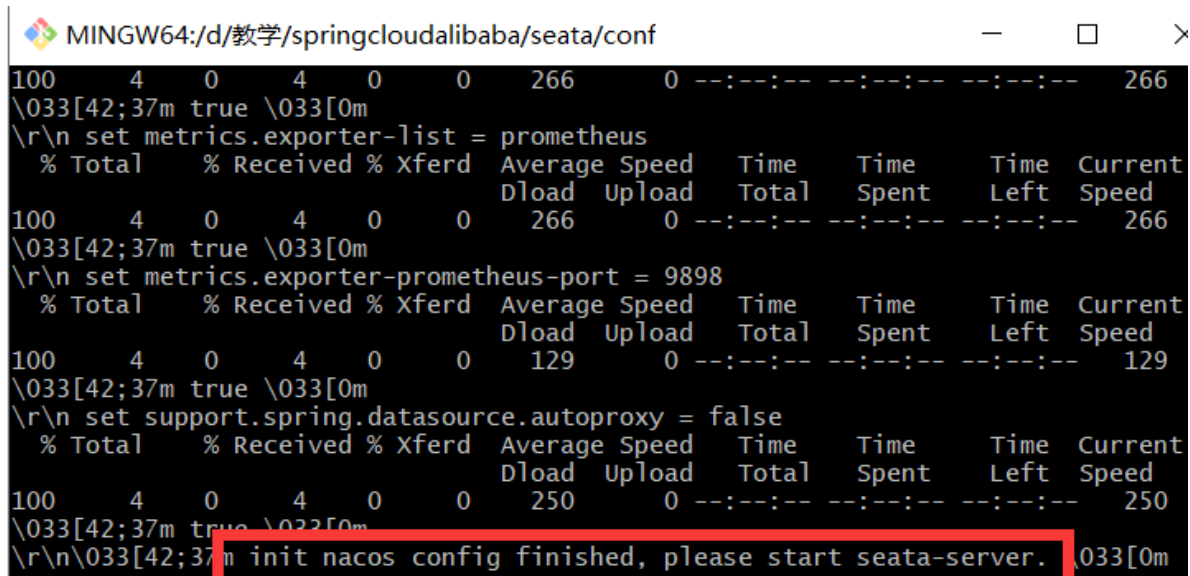
```

4 service.vgroup_mapping.prex_tx_group=default
5 service.enableDegrade=false
6 service.disable=false
7 service.max.commit.retry.timeout=-1
8 service.max.rollback.retry.timeout=-1
9 client.async.commit.buffer.limit=10000
0 client.lock.retry.internal=10
1 client.lock.retry.times=30
2 client.lock.retry.policy.branch-rollback-on-conflict=true
3 client.table.meta.check.enable=true
4 client.report.retry.count=5
5 client.tm.commit.retry.count=1
6 client.tm.rollback.retry.count=1
7 store.mode=db
8 store.file.dir=file_store/data
9 store.file.max-branch-session-size=16384
0 store.file.max-global-session-size=512
1 store.file.file-write-buffer-cache-size=16384
2 store.file.flush-disk-mode=async
3 store.file.session.reload.read_size=100
4 store.db.datasource=druid
5 store.db.db-type=mysql
6 store.db.driver-class-name=com.mysql.jdbc.Driver
7 store.db.url=jdbc:mysql://localhost:3306/seata?useUnicode=true
8 store.db.user=root
9 store.db.password=Zw726515

```

第六步:老师这里式wind环境, 我使用git的控制台 执行sh脚本

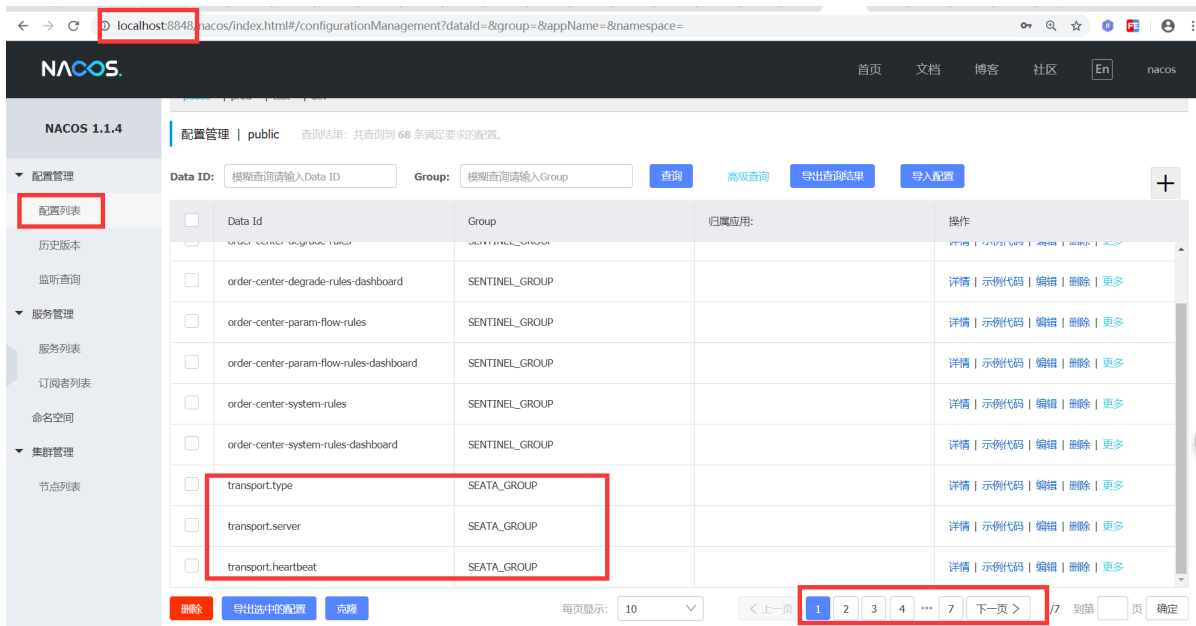
sh nacos-config.sh localhost 把seata的配置导入到nacos的配置中心上去



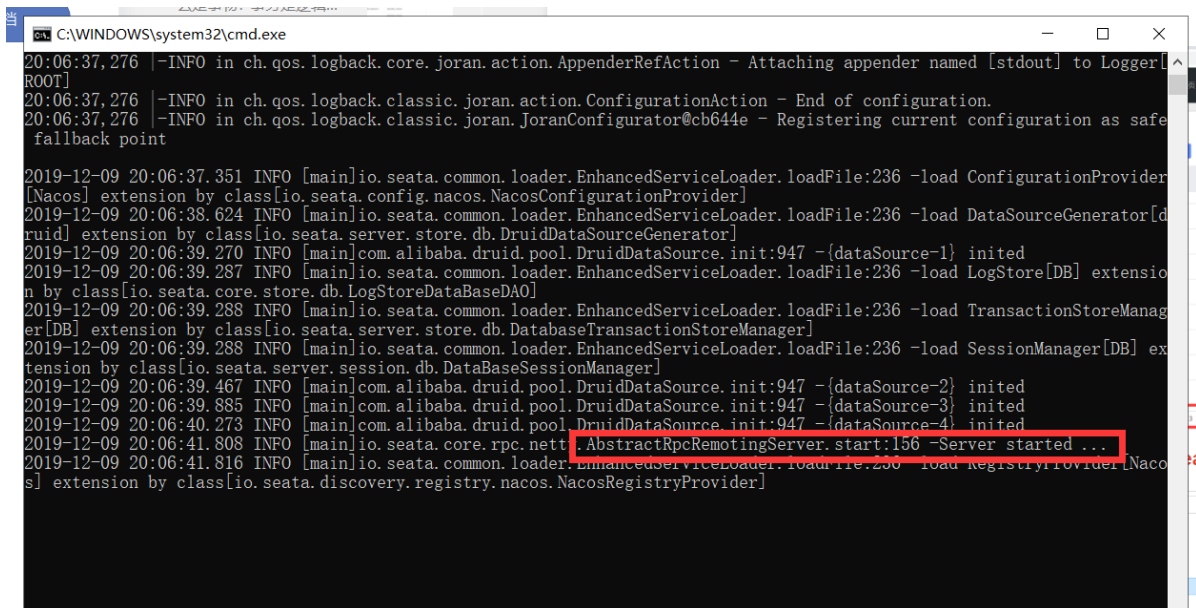
```

MINGW64:/d/教学/springcloudalibaba/seata/conf
100   4   0   4   0   0   266   0  --:--:--  --:--:--  --:--:--   266
\r\n[42;37m true \033[0m
\r\n set metrics.exporter-list = prometheus
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                               Dload  Upload    Total   Spent    Left   Speed
100   4   0   4   0   0   266   0  --:--:--  --:--:--  --:--:--   266
\r\n[42;37m true \033[0m
\r\n set metrics.exporter-prometheus-port = 9898
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                               Dload  Upload    Total   Spent    Left   Speed
100   4   0   4   0   0   129   0  --:--:--  --:--:--  --:--:--   129
\r\n[42;37m true \033[0m
\r\n set support.spring.datasource.autoproxy = false
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                               Dload  Upload    Total   Spent    Left   Speed
100   4   0   4   0   0   250   0  --:--:--  --:--:--  --:--:--   250
\r\n[42;37m true \033[0m
\r\n\033[42;37m init nacos config finished, please start seata-server. \033[0m

```



第七步:启动 seata-server服务 进入seata的bin目录点击执行seata-server.bat



3.2)微服务搭建步骤

第一步:添加pom依赖

```

1 <!--seata-->
2 <dependency>
3   <groupId>com.alibaba.cloud</groupId>
4   <artifactId>spring-cloud-starter-alibaba-seata</artifactId>

```

```

5 <exclusions>
6 <exclusion>
7 <artifactId>seata-all</artifactId>
8 <groupId>io.seata</groupId>
9 </exclusion>
10 </exclusions>
11 </dependency>
12 <dependency>
13 <groupId>io.seata</groupId>
14 <artifactId>seata-all</artifactId>
15 <version>${seata.version}</version>
16 </dependency>

```

第二步:写注解

@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)

```

1 @EnableFeignClients
2 @EnableDiscoveryClient
3 @SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
4 public class Tulingvip06MsAlibabaOrderApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(Tulingvip06MsAlibabaOrderApplication.class, args);
8     }
9
10 }
11

```

微服务发起者需要写全局事务注解（这里是order服务为发起者）

@GlobalTransactional(name = "prex-create-order",rollbackFor = Exception.class)

```

1 @GlobalTransactional(name = "prex-create-order",rollbackFor =
Exception.class)
2 @Override
3 public void createOrder(Order order) {
4     log.info("当前 XID: {}", RootContext.getXID());
5     log.info("下单开始,用户:{},商品:{},数量:{},金额:{},", order.getUserId(),
order.getProductId(), order.getCount(), order.getPayMoney());
6     //创建订单
7     order.setStatus(0);
8     orderMapper.saveOrder(order);
9     log.info("保存订单{}", order);
10
11     //远程调用库存服务扣减库存

```

```

12  log.info("扣减库存开始");
13  remoteStorageService.reduceCount(order.getProductId(), order.getCount());
14  log.info("扣减库存结束");
15
16  //远程调用账户服务扣减余额
17  log.info("扣减余额开始");
18  remoteAccountService.reduceBalance(order.getUserId(), order.getPayMoney());
19  log.info("扣减余额结束");
20
21  //修改订单状态为已完成
22  log.info("修改订单状态开始");
23  orderMapper.updateOrderStatusById(order.getId(),1);
24  log.info("修改订单状态结束");
25
26  log.info("下单结束");
27  }

```

第三步:写配置添加代理数据源配置

```

1  @Configuration
2  @MapperScan(basePackages = {"com.tuling.seata.mapper"})
3  public class MyBatisConfig {
4
5
6  /**
7   * 从配置文件获取属性构造datasource，注意前缀，这里用的是druid，根据自己情况配置，
8   * 原生datasource前缀取"spring.datasource"
9   *
10  * @return
11  */
12  @Bean
13  @ConfigurationProperties(prefix = "spring.datasource.hikari")
14  public DataSource hikariDataSource() {
15  return new HikariDataSource();
16  }
17
18  /**
19   * 构造datasource代理对象，替换原来的datasource
20   *
21   * @param hikariDataSource
22   * @return

```

```

23  */
24  @Primary
25  @Bean("dataSource")
26  public DataSourceProxy dataSourceProxy(DataSource hikariDataSource) {
27      return new DataSourceProxy(hikariDataSource);
28  }
29
30  @Bean
31  public SqlSessionFactoryBean sqlSessionFactory(DataSourceProxy dataSourceProxy) throws Exception {
32      SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
33      sqlSessionFactoryBean.setMapperLocations(new PathMatchingResourcePatternResolver().
34          getResources("classpath:/mybatis/mapper/**/*.xml"));
35      sqlSessionFactoryBean.setConfigLocation(new PathMatchingResourcePatternResolver().getResources("classpath:/mybatis/mybatis-config.xml"));
36      sqlSessionFactoryBean.setTypeAliasesPackage("com.tuling.seata.domin");
37      sqlSessionFactoryBean.setDataSource(dataSourceProxy);
38      return sqlSessionFactoryBean;
39  }
40
41  }

```

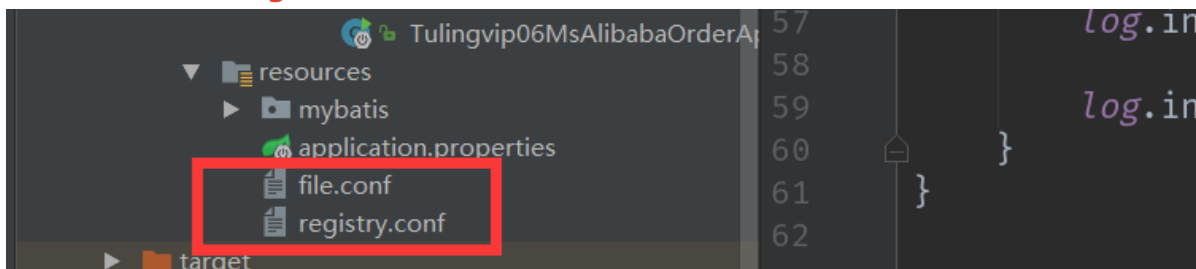
第四步:修改配置文件 yml中添加配置文件

```

1  #seata配置 (配置事务组 需要和seata-server的配置一样)
2  spring.cloud.alibaba.seata.tx-service-group=prex_tx_group

```

修改file.conf 和register.conf文件 (跟seata-server的改动一样)



4:微服务测试

<http://localhost:8081/order/create?userId=1&productId=1&count=1&payMoney=50>

4.1) 正常情况

← → ↻ ⓘ localhost:8081/order/create?userId=1&productId=1&count=1&payMoney=50

```
▼ {  
  "success": true,  
  "msg": "创建订单成功",  
  "data": "订单ID:119"  
}
```

订单库:

- > 视图
- > 函数
- > 事件
- > 查询
- > 报表
- > 备份
- ▼ seata-order
 - ▼ 表
 - order
 - undo_log

id	user_id	pay_money	product_id	status	count
(Null)	(Null)	(Null)	(Null)	(Null)	(Null)

- undo_log
 - > 视图
 - > 函数
 - > 事件
 - > 查询
 - > 报表
 - > 备份
 - ▼ seata-order
 - ▼ 表
 - order
 - undo_log
- > 视图

id	user_id	pay_money	product_id	status	count
119	1	50	1	1	1

微服务成功

库存库

微服务成功调用

id	product_id	price	count
1	1	50	71

1	1	50	70
---	---	----	----

支付库:

微服务成功

id	user_id	balance
1	1	98750

1	1	98700
---	---	-------

4.2) 异常情况,我们把支付服务人工模拟抛出异常。

```
29
30
31 @Override
32 public boolean reduceBalance(Integer userId, BigDecimal balance) throws Ex
33
34     log.info("当前 XID: {}", RootContext.getXID());
35     checkBalance(userId, balance);
36
37     log.info("开始扣减用户 {} 余额", userId);
38     //模拟异常
39     System.out.println(1/0);
40     Integer record = accountMapper.reduceBalance(userId, balance);
41     log.info("结束扣减用户 {} 余额结果:{}", userId, record > 0 ? "操作成功" :
42     return record > 0;
43 }
```

localhost:8081/order/create?userId=1&productId=1&count=1&payMoney=50

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Tue Dec 10 13:37:00 CST 2019

There was an unexpected error (type=Internal Server Error, status=500).

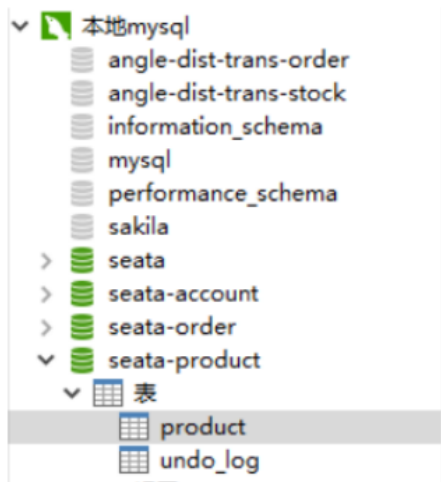
status 500 reading RemoteAccountService#reduceBalance(Integer,BigDecimal)

订单库:

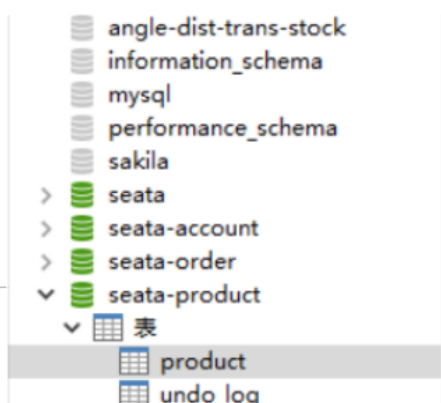
id	user_id	pay_money	product_id	status	count
119	1	50	1	1	1

抛异常订单没有生成, 119是上一个请求生成的

库存库:



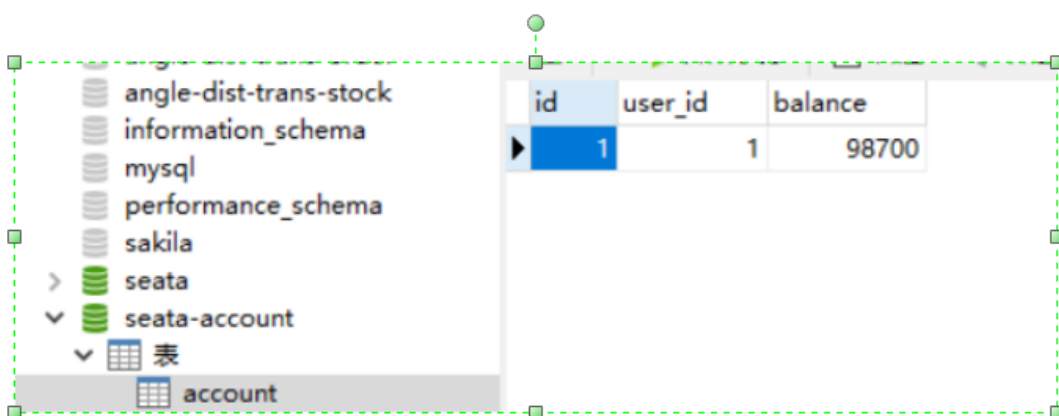
id	product_id	price	count
1	1	50	70



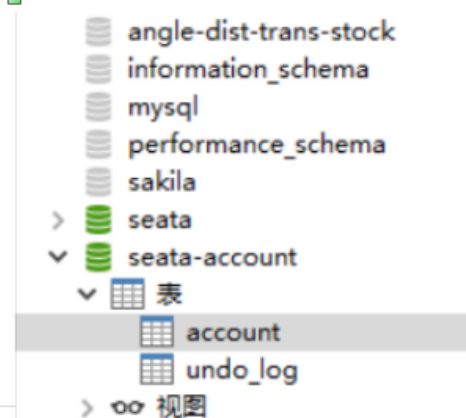
id	product_id	price	count
1	1	50	70

库存没有改变

支付库



id	user_id	balance
1	1	98700



id	user_id	balance
1	1	98700

金额也没有扣除

