一:Seata 源码剖析之 TM全局事务是如何发起的

1.1)经过我们上一节课知道了，我们的Order服务是一个TM（全局事务发起者）,因为我们在order服务的service中标注了@GlobalTransactional注解.

我们发现createOrder的业务逻辑中主要做了四个事情

①: 保存订单(订单状态还没有完结)

②:通过feign远程调用扣减库存

③:通过远程feign调用扣减金额

④:更新订单状态

```
@GlobalTransactional(name = "prex-create-order",rollbackFor = Exception.class)
@Override
public void createOrder(Order order) {

 order.setStatus(0);
 orderMapper.saveOrder(order);

 remoteStorageService.reduceCount(order.getProductId(), order.getCount());

  remoteAccountService.reduceBalance(order.getUserId(), order.getPayMoney());

  orderMapper.updateOrderStatusById(order.getId(),1);
}
```

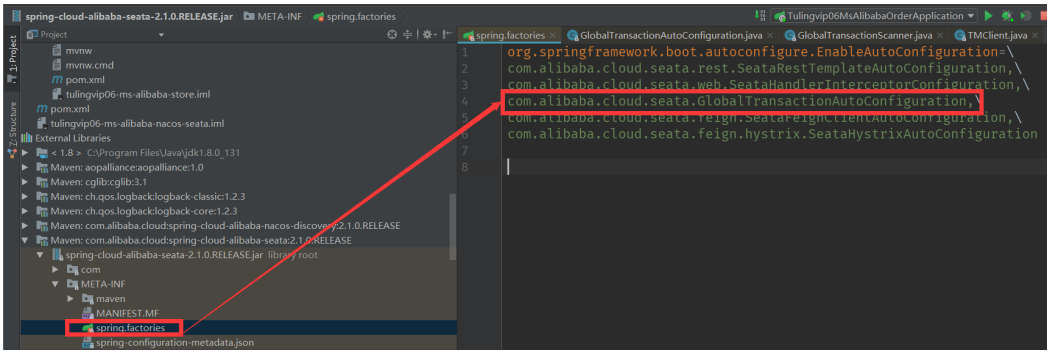
引发思考1）为啥我们在业务方法中添加一个**@GlobalTransactional** 就可以实现了我们的分布式事务的功能，底层是如何做到的?

答案: SpringAop功能,我们结合Spring事务注解的考虑，我们知道spring的原生的事务是通过aop功能完成的。

引发思考2)我们使用Spring 事务的时候，需要自己配置事务管理器，但是我们使用分布式事务注解，却没有事务管理器？那么这个组件配置在哪里?

答案:我们用的是SpringBoot工程，自然而然的想到了SpringBoot的自动装配功能，然而我们又导入了一个seata的启动依赖包。果然我们就发现了

com.alibaba.cloud.seata.GlobalTransactionAutoConfiguration



1.2）现在我们的突破口已经找到了 就是我们的

GlobalTransactionAutoConfiguration

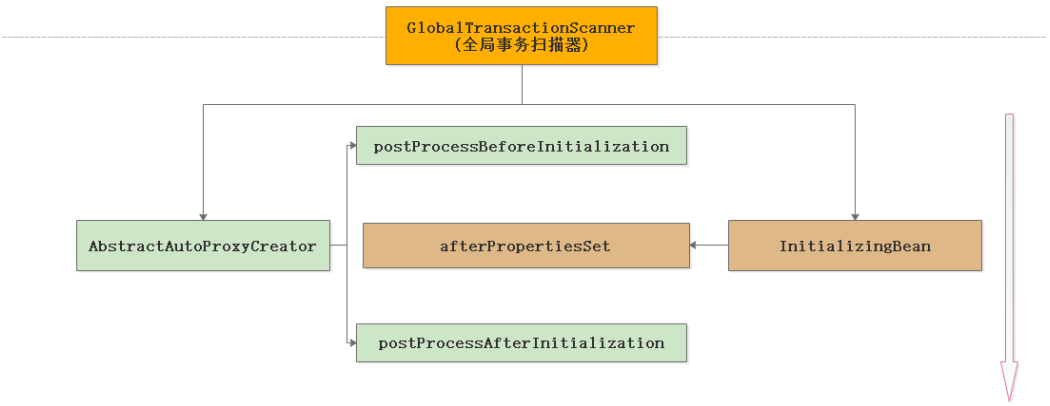
我们发现该自动配置类导入了一个GlobalTransactionScanner 注解扫描器 从名字上我们可以看出来 是用来处理我们的@GlobalTransactionl注解的

```
@Configuration
@EnableConfigurationProperties(SeataProperties.class)
public class GlobalTransactionAutoConfiguration {
 ........其他代码省略
 @Bean
 public GlobalTransactionScanner globalTransactionScanner() {
 //获取我们的微服务应用的名称
 String applicationName = applicationContext.getEnvironment()
 .getProperty("spring.application.name");
 //读取我们的事务分租
 String txServiceGroup = seataProperties.getTxServiceGroup();
 //如没有事务分支，自己生成事务分组
 if (StringUtils.isEmpty(txServiceGroup)) {
 txServiceGroup = applicationName + "-fescar-service-group";
 seataProperties.setTxServiceGroup(txServiceGroup);
 }
 //创建全局事务注解的扫描器
 return new GlobalTransactionScanner(applicationName, txServiceGroup);
 }
}
```

1.3）GlobalTransactionScanner 功能分析。

从1.2)步骤，我们知道最终SpringBoot自动装配功能给我们导入了一个GlobalTransactionScanner 组件,那么我们就分析该组件的功能.

public class GlobalTransactionScanner extends **AbstractAutoProxyCreator** implements **InitializingBean**, ...,



1.3.1)所以我们先研究GlobalTransactionScanner 的afterPropertiesSet方法

①:我们发现afterPropertiesSet--->调用了initClient()方法

```
public void afterPropertiesSet() {
 if (disableGlobalTransaction) {
 if (LOGGER.isInfoEnabled()) {
```

```
LOGGER.info("Global transaction is disabled.");
}
return;
}
//初始化我们的netty的客户端
initClient();
}
```

第二步:所以我们就看下initClient()的方法的作用

①:初始化一个TMClient (事务管理者的客户端，专门用于和seata-server交互)

作用:用于发起一个全局事务，回滚全局事务等.

②:初始化一个RMClient(事务参与者的客户端，用于和seata-server交互)

作用: 注册分支事务，上报本地事务提交的情况。

③:registerSpringShutdownHook （） Spring容器关闭，用户销毁netty的连接的.

```
private void initClient() {

.....省略代码.....
//init TM
TMClient.init(applicationId, txServiceGroup);

//init RM
RMClient.init(applicationId, txServiceGroup);

 registerSpringShutdownHook();

}
```

1.3.2)postProcessAfterInitialization方法该方法最终会调用 wrapIfNecessary(由子类实现GlobalTransactionScanner)

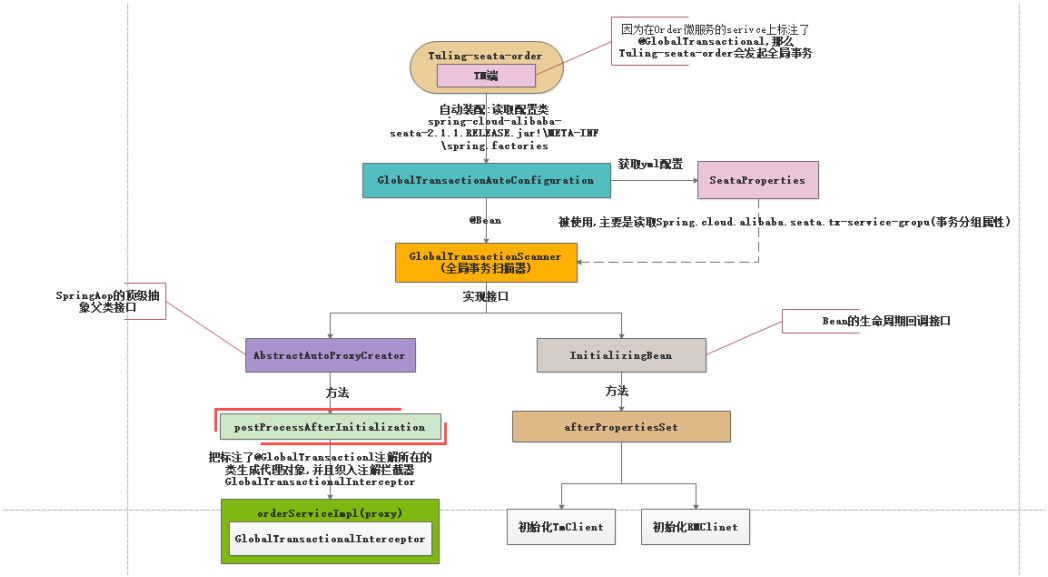
①:创建一个GlobalTransactionalInterceptor组件，用于拦截标注了@GlobalTransactionl方法的类

②:把标注了@GlobalTransactionl方法所在的类 变成一个代理对象。

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
 。。。省略代码
try {
synchronized (PROXYED_SET) {
if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName, applicationContext)) {
//拦截tcc模式的
interceptor = new TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
} else {
//拦截at模式的
 Class<?> serviceInterface = SpringProxyUtils.findTargetClass(bean);
 Class<?>[] interfacesIfJdk = SpringProxyUtils.findInterfaces(bean);
 //会创建一个GlobalTransactionalInterceptor对象，
 该拦截器会拦截标注了@GlobalTransactional执行的方法
 if (interceptor == null) {
 interceptor = new GlobalTransactionalInterceptor(failureHandlerHook);
```

```
16  }
17  }
18
19  if (!AopUtils.isAopProxy(bean)) {
20  //把标注了@GlobalTransactionl注解的类生成代理对象.
21  bean = super.wrapIfNecessary(bean, beanName, cacheKey);
22  } else {
23
24  }
25  PROXYED_SET.add(beanName);
26  return bean;
27  }
28  } catch (Exception exx) {
29  throw new RuntimeException(exx);
30  }
31  }
```

原理图:



2.1)如何开启全局事务?

第一步:由前端发起下单请求

http://localhost:8081/order/createuserId=1&productId=1&count=1&payMoney=50

请求到我们的OrderController，我们的orderController会调用我们的orderService.

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

@Override

public void createOrder(Order order) {}

2.2.1)而我们的orderSerivceImpl是一个代理对象，会被我们的GlobalTransactionalInterceptor给拦截住.

会调用GlobalTransactionalInterceptor的invoke方法

1)MethodInvocation 中解析出调用类的所属Class对象

2)从class中解析出所执行的方法

3)判断我们方法上是否标注了@GlobalTransactional

4)判断方法上是否标注了@GlobalLock注解(后面讲作用)

5)根据3，4步的注解解析情况实现不同的逻辑

```
@Override
public Object invoke(final MethodInvocation methodInvocation) throws Throwable {
 //获取我们的Class对象
 Class<?> targetClass = (methodInvocation.getThis() != null ? AopUtils.getTargetClass(meth
odInvocation.getThis()) : null);
 //根据class找到对应的方法
 Method specificMethod = ClassUtils.getMostSpecificMethod(methodInvocation.getMethod(), ta
rgetClass);
 final Method method = BridgeMethodResolver.findBridgedMethod(specificMethod);
 //解析我们的方法上的GlobalTransactional
 final GlobalTransactional globalTransactionalAnnotation = getAnnotation(method, GlobalTra
nsactional.class);
  //解析我们方法上的GlobalLock
  final GlobalLock globalLockAnnotation = getAnnotation(method, GlobalLock.class);
  if (globalTransactionalAnnotation != null) {
  return handleGlobalTransaction(methodInvocation, globalTransactionalAnnotation);
  } else if (globalLockAnnotation != null) {
  return handleGlobalLock(methodInvocation);
  } else {
  return methodInvocation.proceed();
  }

```

2.2.2）执行 io.seata.spring.annotation.GlobalTransactionalInterceptor#handleGlobalTransaction 用来处理全局事务注解的逻辑

1)该方法中，通过事务transactionalTemplate.execute()执行业务逻辑，传入一个匿名内部了TransactionalExecutor对象进去。TransactionalExecutor三个方法

1)执行真正业务逻辑的方法

2)获取全局事务名称的方法

3)用户封装全局事务注解属性对象的方法

```
private Object handleGlobalTransaction(final MethodInvocation methodInvocation,
 final GlobalTransactional globalTrxAnno) throws Throwable {
 try {
 return transactionalTemplate.execute(new TransactionalExecutor() {
 //执行真正的业务逻辑
 @Override
 public Object execute() throws Throwable {
 return methodInvocation.proceed();
 }
  //全局事务的名称
  public String name() {
  String name = globalTrxAnno.name();
  if (!StringUtils.isNullOrEmpty(name)) {
```

```
14  return name;
15  }
16  return formatMethod(methodInvocation.getMethod());
17  }
18
19  //全局事务注解解析封装的对象
20  @Override
21  public TransactionInfo getTransactionInfo() {
22  TransactionInfo transactionInfo = new TransactionInfo();
23  transactionInfo.setTimeOut(globalTrxAnno.timeoutMills());
24  transactionInfo.setName(name());
25  Set<RollbackRule> rollbackRules = new LinkedHashSet<>();
26  for (Class<?> rbRule : globalTrxAnno.rollbackFor()) {
27  rollbackRules.add(new RollbackRule(rbRule));
28  }
29  for (String rbRule : globalTrxAnno.rollbackForClassName()) {
30  rollbackRules.add(new RollbackRule(rbRule));
31  }
32  for (Class<?> rbRule : globalTrxAnno.noRollbackFor()) {
33  rollbackRules.add(new NoRollbackRule(rbRule));
34  }
35  for (String rbRule : globalTrxAnno.noRollbackForClassName()) {
36  rollbackRules.add(new NoRollbackRule(rbRule));
37  }
38  transactionInfo.setRollbackRules(rollbackRules);
39  return transactionInfo;
40  }
41  });
42 }
```

2)io.seata.tm.api.TransactionalTemplate#execute

该方法干了五个事情,

第一步:创建或者生成一个新的全局事务

第二步:获取@GlobalTransacationl注解解析出来的对象的信息

第三步:开启全局事务

第四步:执行业务逻辑

第五步:

执行业务逻辑没有抛出异常，全局提交

执行业务逻辑抛出异常，全局回滚。

```
1
2 public Object execute(TransactionalExecutor business) throws Throwable {
3  //创建或者获取一个全局事务
4  GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();
5
6  // 调用TransactionalExecutor 上一步传递进来的TransactionalExecutor
7  //并且调用该匿名对象的getTransactionInfo获取全局事务注解信息
```

```
TransactionInfo txInfo = business.getTransactionInfo();
if (txInfo == null) {
 throw new ShouldNeverHappenException("transactionInfo does not exist");
 }
 try {

 //开启全局事务
 beginTransaction(txInfo, tx);

 Object rs = null;
 try {

 // 执行业务方法，也就是我们的orderService的createOrder方法
 rs = business.execute();

 } catch (Throwable ex) {

 //若执行业务方法抛出异常，触发全局回滚
 completeTransactionAfterThrowing(txInfo,tx,ex);
 throw ex;
 }

 // 4.触发全局事务提交
 commitTransaction(tx);

 return rs;
 } finally {
 //5. clear
 triggerAfterCompletion();
 cleanUp();
 }
}
```

第一步:创建或者获取一个全局事务?

io.seata.tm.api.GlobalTransactionContext#getCurrentOrCreate

>io.seata.tm.api.GlobalTransactionContext#getCurrent（获取全局事务）

>io.seata.tm.api.GlobalTransactionContext#createNew(创建全局事务)

```
/**
 * 判断当前线程是否绑定了一个全局事务，若没有直接创建要给
 *
 * @return new context if no existing there.
 */
public static GlobalTransaction getCurrentOrCreate() {
 //获取当前的一个全局事务
 GlobalTransaction tx = getCurrent();
 //没有获取到全局事务,就新创建一个全局事务
 if (tx == null) {
 return createNew();
```

```
 }
 return tx;
}

//获取一个全局事务的业务逻辑.........
/**
 * Get GlobalTransaction instance bind on current thread.
 *
 * @return null if no transaction context there.
 */
private static GlobalTransaction getCurrent() {
 //去当前上下文中获取全局事务ID
 String xid = RootContext.getXID();
 //没有获取到就直接返回
 if (xid == null) {
 return null;
 }
 //获取到了，那么当前就是事务的参与者而不是发起者
 return new DefaultGlobalTransaction(xid, GlobalStatus.Begin, GlobalTransactionRole.Parti
cipant);
}

/**创建一个全局事务，并且是发起者
 * Try to create a new GlobalTransaction.
 *
 * @return
 */
private static GlobalTransaction createNew() {
 GlobalTransaction tx = new DefaultGlobalTransaction();
 return tx;
}
DefaultGlobalTransaction() {
 this(null, GlobalStatus.UnKnown, GlobalTransactionRole.Launcher);
}
```

第二步:获取全局事务注解的信息

获取@GlobalTransacationl注解解析出来的对象的信息

```
public TransactionInfo getTransactionInfo() {
 TransactionInfo transactionInfo = new TransactionInfo();
 //超时
 transactionInfo.setTimeOut(globalTrxAnno.timeoutMills());
 //全局事务名称
 transactionInfo.setName(name());
 /全局事务 回滚 和不回滚的规则设置.
 Set<RollbackRule> rollbackRules = new LinkedHashSet<>();
 for (Class<?> rbRule : globalTrxAnno.rollbackFor()) {
 rollbackRules.add(new RollbackRule(rbRule));
 }
```

```
 for (String rbRule : globalTrxAnno.rollbackForClassName()) {
 rollbackRules.add(new RollbackRule(rbRule));
 }
 for (Class<?> rbRule : globalTrxAnno.noRollbackFor()) {
 rollbackRules.add(new NoRollbackRule(rbRule));
 }
 for (String rbRule : globalTrxAnno.noRollbackForClassName()) {
 rollbackRules.add(new NoRollbackRule(rbRule));
 }
 transactionInfo.setRollbackRules(rollbackRules);
 return transactionInfo;
}
```

第三步:开启全局事务，说白了就是把刚刚创建的全局事务信息注册到seata-server上保存到了global_table上

io.seata.tm.api.TransactionalTemplate#beginTransaction

>io.seata.tm.api.DefaultGlobalTransaction#begin(int, java.lang.String)

>io.seata.tm.DefaultTransactionManager#begin

```
 private void beginTransaction(TransactionInfo txInfo, GlobalTransaction tx) throws Transac
tionalExecutor.ExecutionException {
 try {
 //空方法
 triggerBeforeBegin();
 //真正提交的方法GlobalTransaction 的默认类DefaultGlobalTransaction处理
 tx.begin(txInfo.getTimeOut(), txInfo.getName());
 //空方法
 triggerAfterBegin();
 } catch (TransactionException txe) {
 throw new TransactionalExecutor.ExecutionException(tx, txe,
 TransactionalExecutor.Code.BeginFailure);

 }
}

//真正的开启全局事务的逻辑
public void begin(int timeout, String name) throws TransactionException {
 //判断若不是事务发起者 不执行开始逻辑
 if (role != GlobalTransactionRole.Launcher) {
 check();
 if (LOGGER.isDebugEnabled()) {
 LOGGER.debug("Ignore Begin(): just involved in global transaction [" + xid + "]");
 }
 return;
 }
 //若有了xid了 ,那么就是事务的参与者，上有服务已经传递过来的xid
 if (xid != null) {
 throw new IllegalStateException();
```

```
29   }
30   if (RootContext.getXID() != null) {
31   throw new IllegalStateException();
32   }
33   //真正的开启一个事务,由DefaultTransactionManager发起全局事务提交开启
34
35   xid = transactionManager.begin(null, null, name, timeout);
36   status = GlobalStatus.Begin;
37   RootContext.bind(xid);
38   if (LOGGER.isInfoEnabled()) {
39   LOGGER.info("Begin new global transaction [" + xid + "]");
40   }
41
42
43  //开启全局事务
44  public String begin(String applicationId, String transactionServiceGroup, String name, ir
t timeout)
45   throws TransactionException {
46   //封装全局事务的请求
47   GlobalBeginRequest request = new GlobalBeginRequest();
48   //事务的名称
49   request.setTransactionName(name);
50   全局事务超时
51   request.setTimeout(timeout);
52   //同步调用seata-server的begin接口，返回全局事务id
53   GlobalBeginResponse response = (GlobalBeginResponse)syncCall(request);
54   if (response.getResultCode() == ResultCode.Failed) {
55   throw new TmTransactionException(TransactionExceptionCode.BeginFailed,
response.getMsg());
56   }
57   return response.getXid();
58  }
59
60  private AbstractTransactionResponse syncCall(AbstractTransactionRequest request) throws T
ransactionException {
61   try {
62   return (AbstractTransactionResponse)TmRpcClient.getInstance().sendMsgWithResponse(reques
t);
63   } catch (TimeoutException toe) {
64   throw new TmTransactionException(TransactionExceptionCode.IO, "RPC timeout", toe);
65   }
66  }
```

第四步: Seata-server服务端 开启全局事务源码分析

4.1)我们由客户端发起的全局事务的开启最终调用到seata-server上，会被DefaultCoordinator.doGlobalBegin()来处理全局事务的开启业务逻辑

1）我们发现这里会调用DefaultCore.begin()进行真的业务出来

```
protected void doGlobalBegin(GlobalBeginRequest request, GlobalBeginResponse response, RpcContext rpcContext)
  throws TransactionException {
  response.setXid(core.begin(rpcContext.getApplicationId(), rpcContext.getTransactionServiceGroup(),
  request.getTransactionName(), request.getTimeout()));
}
```

2)所以我们需要研究一下DefaultCore.begin代码业务逻辑

```
/**
 * 方法实现说明:该方法是被我们的TM(事务管理者调用,用于开启全局事务)
 * @author:smlz
 * @param applicationId:开启全局事务的应用名称
 * @param transactionServiceGroup:事务分组
 * @param name:分布式事务名称
 * @param timeout 分布式事务超时时间
 * @return: String 全局事务Id
 * @exception:
 * @date:2019/12/11 13:50
 */
@Override
public String begin(String applicationId, String transactionServiceGroup, String name, int timeout)
  throws TransactionException {
  //第一步:创建要给全局事务session
  GlobalSession session = GlobalSession.createGlobalSession(
  applicationId, transactionServiceGroup, name, timeout);
  /**
  * 1:为session中添加回调监听
  * 2:SessionHolder.getRootSessionManager() 去获取一个全局session管理器(DataBaseSessionManager)
  * SessionLifecycleListener[DataBaseSessionManager]
  * --
  */
  session.addSessionLifecycleListener(SessionHolder.getRootSessionManager());

  //开启全局事务
  session.begin();

  //transaction start event
  eventBus.post(new GlobalTransactionEvent(session.getTransactionId(), GlobalTransactionEvent.ROLE_TC,
  session.getTransactionName(), session.getBeginTime(), null, session.getStatus()));

  LOGGER.info("Successfully begin global transaction xid = {}", session.getXid());
  return session.getXid();
}
```

3)开启全局事务session.begin 那么他最终调用的我们的 SessionLifecycleListener 的onBegin方法,然后我们的debug的时候,发现调用的是SessionLifecycleListener 的抽象类

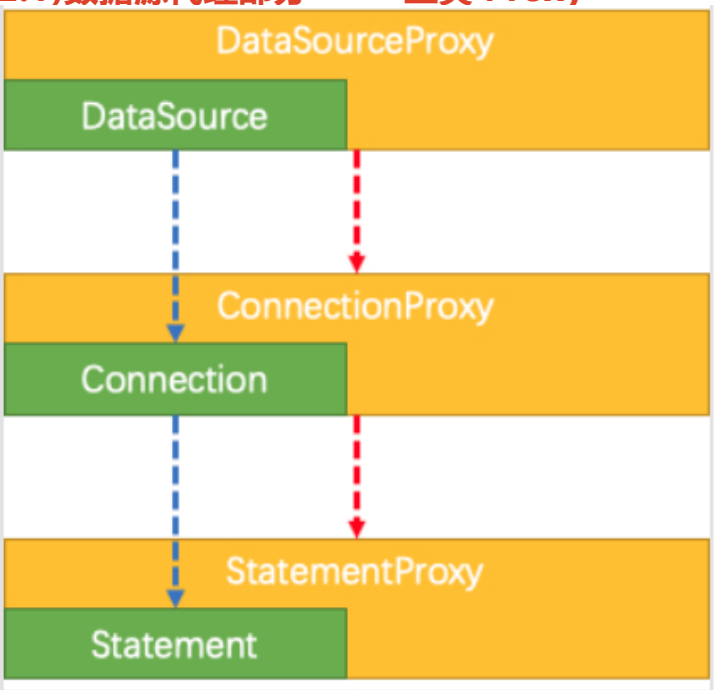
AbstractSessionManager的onBegin方法，最后我们定位到addGlbalSession调用的是我们通过Spi读取出来的DataBaseSessionManager的addGlobalSession方法

```
public void begin() throws TransactionException {

 this.status = GlobalStatus.Begin;
 this.beginTime = System.currentTimeMillis();
 this.active = true;
 //通过SesionLifecycleListner 去开启一个全局事务
 for (SessionLifecycleListener lifecycleListener : lifecycleListeners) {
 //调用监听的onBegin的方法去开启全局事务
 lifecycleListener.onBegin(this);
  }
 }

 ====================AbstractSessionManager==================
@Override
public void onBegin(GlobalSession globalSession) throws TransactionException {
  addGlobalSession(globalSession);
 }

public void addGlobalSession(GlobalSession session) throws TransactionException {
  if (StringUtils.isBlank(taskName)) {
  //这里的transactionStoreManager也是通过我们的SPI去读取实例化的
  boolean ret = transactionStoreManager.writeSession(LogOperation.GLOBAL_ADD, session);
  if (!ret) {
  throw new StoreException("addGlobalSession failed.");
  }
  } else {
  boolean ret = transactionStoreManager.writeSession(LogOperation.GLOBAL_UPDATE, session);
  if (!ret) {
  throw new StoreException("addGlobalSession failed.");
  }
  }
 }

 原生的jdbc代码进行插入globalTbale中
public boolean insertGlobalTransactionDO(GlobalTransactionDO globalTransactionDO) {
  String sql = LogStoreSqls.getInsertGlobalTransactionSQL(globalTable, dbType);
  Connection conn = null;
  PreparedStatement ps = null;
  try {
  conn = logStoreDataSource.getConnection();
  conn.setAutoCommit(true);
  ps = conn.prepareStatement(sql);
  ps.setString(1, globalTransactionDO.getXid());
  ps.setLong(2, globalTransactionDO.getTransactionId());
  ps.setInt(3, globalTransactionDO.getStatus());
  ps.setString(4, globalTransactionDO.getApplicationId());
```

```
47  ps.setString(5, globalTransactionDO.getTransactionServiceGroup());
48  String transactionName = globalTransactionDO.getTransactionName();
49  transactionName = transactionName.length() > transactionNameColumnSize ?
50  transactionName.substring(0, transactionNameColumnSize) : transactionName;
51  ps.setString(6, transactionName);
52  ps.setInt(7, globalTransactionDO.getTimeout());
53  ps.setLong(8, globalTransactionDO.getBeginTime());
54  ps.setString(9, globalTransactionDO.getApplicationData());
55  return ps.executeUpdate() > 0;
56  } catch (SQLException e) {
57  throw new StoreException(e);
58  } finally {
59  if (ps != null) {
60  try {
61  ps.close();
62  } catch (SQLException e) {
63  }
64  }
65  if (conn != null) {
66  try {
67  conn.close();
68  } catch (SQLException e) {
69  }
70  }
71  }
72 }
73
```

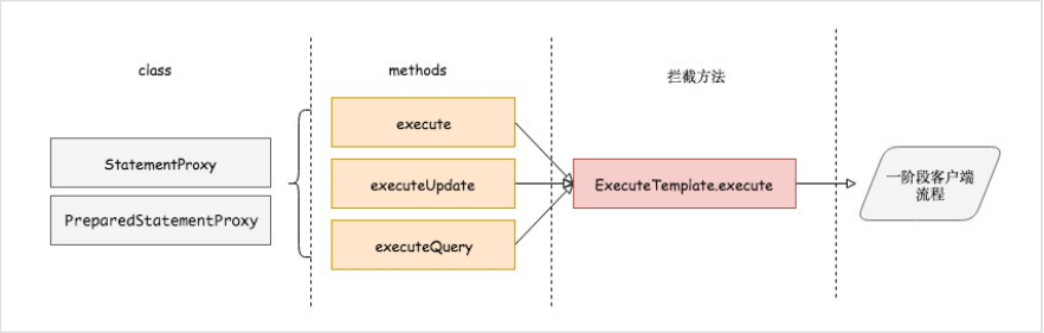
二:RM（事务参与者注册分支事务）

2.1)数据源代理部分 —— 三类 Proxy



Seata 中主要针对 java.sql 包下的 DataSource、Connection、Statement、PreparedStatement 四个接口进行了再包装，包装类分别为 DataSourceProxy、ConnectionProxy、StatementProxy、PreparedStatementProxy，很好一一对印，其功能是在 SQL 语句执行前后、事务 commit 或者 rollbakc 前后进行一些与 Seata 分布式事务相关的操作，例如**分支注册、状态回报、全局锁查询、快照存储、反向 SQL 生成**等。

2.2)ExecuteTemplate.execute



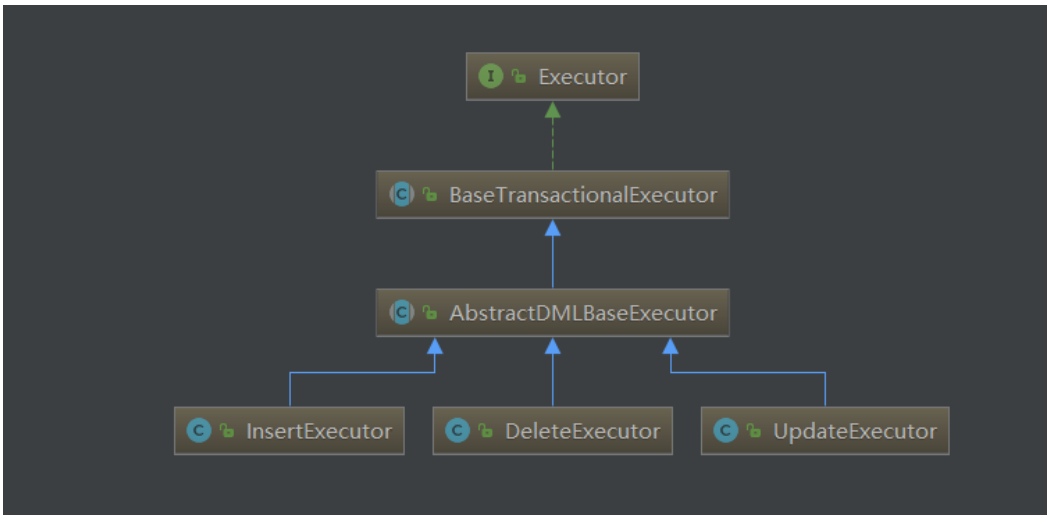
AT 模式下，真正分支事务开始是在 StatementProxy 和 PreparedStatementProxy 的 execute、executeQuery、executeUpdate 等具体执行方法中，这些方法均实现
自 Statement 和 PreparedStatement 的标准接口，而方法体内调用了 ExecuteTemplate.execute 做方法拦截，下面我们来看看这个方法的实现：

①:我们的入口就是ExecuteTemplate.execute的方法

```
/**
 * Execute t.
 *
 * @param <T> the type parameter
 * @param <S> the type parameter
 * @param sqlRecognizer the sql recognizer
 * @param statementProxy the statement proxy
 * @param statementCallback the statement callback
 * @param args the args
 * @return the t
 * @throws SQLException the sql exception
 */
public static <T, S extends Statement> T execute(SQLRecognizer sqlRecognizer,
 StatementProxy<S> statementProxy,
 StatementCallback<T, S> statementCallback,
 Object... args) throws SQLException {
 //若不处于全局分布式事务下，或者是执行的方法上没有被@GlobalTable修饰，那么
 //执行的Sql 是不被纳入到被seata框架管理范围
 if (!RootContext.inGlobalTransaction() && !RootContext.requireGlobalLock()) {
 // Just work as original statement
 return statementCallback.execute(statementProxy.getTargetStatement(), args);
 }

 //生成sql语句的识别器，用于识别原始sql所涉及到的执行的类型
 //比如是数据库类型,CRUD语句类型,操作的表
 if (sqlRecognizer == null) {
```

```
27  sqlRecognizer = SQLVisitorFactory.get(
28  statementProxy.getTargetSQL(),
29  statementProxy.getConnectionProxy().getDbType());
30  }
31  Executor<T> executor = null;
32  if (sqlRecognizer == null) {
33  executor = new PlainExecutor<T, S>(statementProxy, statementCallback);
34  } else {
35  //根据我们原始sql执行的脚本类 生成不同sql的执行器
36  switch (sqlRecognizer.getSQLType()) {
37  case INSERT:
38  //insert 语句执行器
39  executor = new InsertExecutor<T, S>(statementProxy, statementCallback, sqlRecognizer);
40  break;
41  case UPDATE:
42  //update语句执行器
43  executor = new UpdateExecutor<T, S>(statementProxy, statementCallback, sqlRecognizer);
44  break;
45  case DELETE:
46  //delete语句执行器
47  executor = new DeleteExecutor<T, S>(statementProxy, statementCallback, sqlRecognizer);
48  break;
49  case SELECT_FOR_UPDATE:
50  //select_for_update语句执行器
51  executor = new SelectForUpdateExecutor<T, S>(statementProxy, statementCallback, sqlRecog
nizer);
52  break;
53  default:
54  //其他类型的
55  executor = new PlainExecutor<T, S>(statementProxy, statementCallback);
56  break;
57  }
58  }
59  T rs = null;
60  try {
61  //根据具体的执行器执行我们的sql语句
62  rs = executor.execute(args);
63  } catch (Throwable ex) {
64  if (!(ex instanceof SQLException)) {
65  // Turn other exception into SQLException
66  ex = new SQLException(ex);
67  }
68  throw (SQLException)ex;
69  }
70  return rs;
71 }
```

②:各个语句执行器的结构继承图



最最关键的AbstarctDMLBaseExecutor 类暴露了二个模板方法给子类调用

```
前置镜像查询语句
protected abstract TableRecords beforeImage() throws SQLException;
后置镜像查询语句
protected abstract TableRecords afterImage(TableRecords beforeImage) throws SQLException;
```

③：执行器的具体的执行的方法.executor.execute(args);

该方法是一个公共方法被BaseTransactionExecutor提供

```
public Object execute(Object... args) throws Throwable {
 //判断是不是在全局分布式事务中
 if (RootContext.inGlobalTransaction()) {
 //全局全局分布式事务的id
 String xid = RootContext.getXID();
 //把xid全局事务id绑定到我们的ConnectionProxy上
 statementProxy.getConnectionProxy().bind(xid);
 }
 //判断是不是执行了@GlobalLock注释修饰的方法
  if (RootContext.requireGlobalLock()) {
  statementProxy.getConnectionProxy().setGlobalLockRequire(true);
  } else {
  statementProxy.getConnectionProxy().setGlobalLockRequire(false);
  }
  //执行我么你业务sql
  return doExecute(args);
}
```

④:doExecute方法是我们的抽象类AbstractDMLBaseExecutor.doExecute方法

```
public T doExecute(Object... args) throws Throwable {
 //获取数据库连接代理对象(包装了我们原生的连接)
 AbstractConnectionProxy connectionProxy = statementProxy.getConnectionProxy();
 //根据原生的数据库连接的自动提交模式 来执行，我们原生的提交模式是没有该表的
 //所以会执行我们的executeAutoCommitTrue
 if (connectionProxy.getAutoCommit()) {
 return executeAutoCommitTrue(args);
 } else {
```

```
 return executeAutoCommitFalse(args);
 }
}
```

⑤:io.seata.rm.datasource.exec.AbstractDMLBaseExecutor#executeAutoCommitTrue

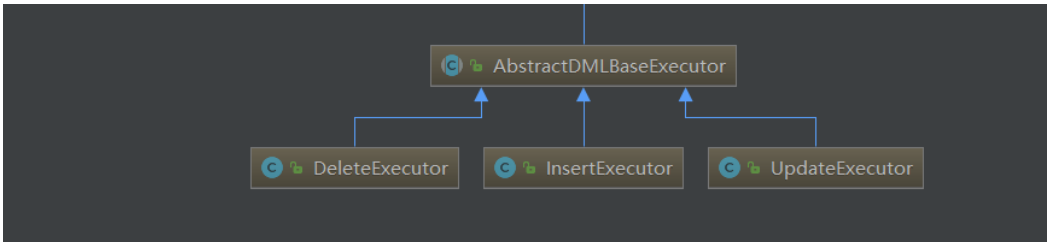
```
protected T executeAutoCommitTrue(Object[] args) throws Throwable {
 //获取数据库连接
 AbstractConnectionProxy connectionProxy = statementProxy.getConnectionProxy();
 try {
 //把连接自动提交关闭，进行手动提交//
 //然后回执行execute方法.
 connectionProxy.setAutoCommit(false);
 return new LockRetryPolicy(connectionProxy.getTargetConnection()).execute(() -> {
 T result = executeAutoCommitFalse(args);
  connectionProxy.commit();
  return result;
  });
  } catch (Exception e) {
  // when exception occur in finally,this exception will lost, so just print it here
  LOGGER.error("execute executeAutoCommitTrue error:{}", e.getMessage(), e);
  if (!LockRetryPolicy.isLockRetryPolicyBranchRollbackOnConflict()) {
  connectionProxy.getTargetConnection().rollback();
  }
  throw e;
  } finally {
  ((ConnectionProxy) connectionProxy).getContext().reset();
  connectionProxy.setAutoCommit(true);
  }
}

protected <T> T doRetryOnLockConflict(Callable<T> callable) throws Exception {
  LockRetryController lockRetryController = new LockRetryController();
  //写一个死循环
  while (true) {
  try {
  //真正的执行我们的业务逻辑
  return callable.call();
  } catch (LockConflictException lockConflict) {
  //全局锁异常,回滚
  onException(lockConflict);
  //sleep 重试10次，每次30ms，重试去获取全局锁
  lockRetryController.sleep(lockConflict);
  } catch (Exception e) {
  onException(e);
  throw e;
  }
  }
}
```

```
44
45
46 ======================这段逻辑不好理解 我们把代码变动一下========让大家看的更加直观
47 protected T executeAutoCommitTrue(Object[] args) throws Throwable{
48  //获取代理连接
49  AbstractConnectionProxy connectionProxy = statementProxy.getConnectionProxy();
50
51  try{
52  //关闭自动提交
53  connectionProxy.setAutoCommit(false);
54
55  while(true) {
56  try{
57  //执行目标sql的前置快照
58  TableRecords beforeImage = beforeImage();
59  //执行目标方法
60  T result = statementCallback.execute(statementProxy.getTargetStatement(), args);
61  //执行后置快照
62  TableRecords afterImage = afterImage(beforeImage);
63  //把前后快照构建一个undoLog对象
64  prepareUndoLog(beforeImage, afterImage);
65
66
67  //分支事务注册=======================================可能抛出LockConflictException异常
68  register();
69
70  插入 undoLog表
71  UndoLogManagerFactory.getUndoLogManager(this.getDbType()).flushUndoLogs(this);
72
73  targetConnection.commit();
74
75  break;
76
77  }catch (LockConflictException lockConflict){
78  //回滚
79  onException(lockConflict);
80  //睡30ms 10次
81  lockRetryController.sleep(lockConflict);
82  }catch(Exception e){
83  onException(e);
84  throw e;
85  }
86
87
88  }
89
90  } catch (Exception e) {
91
92  if (!LockRetryPolicy.isLockRetryPolicyBranchRollbackOnConflict()) {
```

```
93   connectionProxy.getTargetConnection().rollback();
94   }
95   throw e;
96   } finally {
97   connectionProxy.setAutoCommit(true);
98   }
99  }
```

⑥:io.seata.rm.datasource.exec.AbstractDMLBaseExecutor#beforeImage

该方法是一个模板方法，具体的实现是由具体的语句执行器实现。



a)我们先分析**InsertExecutor.beforeImage**的前置方法

就是我们的原生的insert语句是不需要做前置镜像的。因为回滚的时候，我们只需要根据后置镜像数据把对应的业务sql插入的数据给删除掉

```
1  protected TableRecords beforeImage() throws SQLException {
2   return TableRecords.empty(getTableMeta());
3  }
```

InsertExecutor.afterImage方法 的后置镜像。

```
1  protected TableRecords afterImage(TableRecords beforeImage) throws SQLException {
2   //判断insert插入数据的注解 是不是自增的，若是 获取刚刚自增的注解id
3   List<Object> pkValues = containsPK() ? getPkValuesByColumn() :
4   (containsColumns() ? getPkValuesByAuto() : getPkValuesByColumn());
5   //传入主键，通过拼接select * from where id= 主键值 for update 并且执行该语句
6   //封装到tableRecode
7   TableRecords afterImage = buildTableRecords(pkValues);
8
9   if (afterImage == null) {
10   throw new SQLException("Failed to build after-image for insert");
11   }
12
13   return afterImage;
14  }
```

b)现在我们分析UpdateExecutor.beforeImage的前置方法

```
1  @Override
2  protected TableRecords beforeImage() throws SQLException {
3   //预编译参数列表集合
4   ArrayList<List<Object>> paramAppenderList = new ArrayList<>();
5   //获取我们的表结构
6   TableMeta tmeta = getTableMeta();
7   //构建前置镜像sql
8   String selectSQL = buildBeforeImageSQL(tmeta, paramAppenderList);
```

```
 //执行select id,count from product where product_id=? for update
 //返回前置镜像
 return buildTableRecords(tmeta, selectSQL, paramAppenderList);
}


private String buildBeforeImageSQL(TableMeta tableMeta, ArrayList<List<Object>> paramAppenderList) {
 //sql识别器
 SQLUpdateRecognizer recognizer = (SQLUpdateRecognizer)sqlRecognizer;
 //获取更新的列数
 List<String> updateColumns = recognizer.getUpdateColumns();
 //拼接select 查询关键字
 StringBuilder prefix = new StringBuilder("SELECT ");
 //更新的列数中是否包含了注解
 if (!tableMeta.containsPK(updateColumns)) {
 //拼接sql select id,
 prefix.append(getColumnNameInSQL(tableMeta.getPkName()) + ", ");
 }

 //查询语句的 from product
 StringBuilder suffix = new StringBuilder(" FROM " + getFromTableInSQL());
 //构建查找的条件 from product where product_id=?
 String whereCondition = buildWhereCondition(recognizer, paramAppenderList);
 if (StringUtils.isNotBlank(whereCondition)) {
 suffix.append(" WHERE " + whereCondition);
 }
 //拼接 from product where product_id=? for update
 suffix.append(" FOR UPDATE");
 //拼接select id,count from product where product_id=? for update
 StringJoiner selectSQLJoin = new StringJoiner(", ", prefix.toString(), suffix.toString());
 for (String updateColumn : updateColumns) {
 selectSQLJoin.add(updateColumn);
 }
 return selectSQLJoin.toString();
}

```

后置镜像 afterImage

```
protected TableRecords afterImage(TableRecords beforeImage) throws SQLException {
 //获取到表的结构
 TableMeta tmeta = getTableMeta();
 //前置镜像没有的花，那么后置镜像也返回空
 if (beforeImage == null || beforeImage.size() == 0) {
 return TableRecords.empty(getTableMeta());
 }
 //构建后置sql的语句
 String selectSQL = buildAfterImageSQL(tmeta, beforeImage);
```

```
10  TableRecords afterImage = null;
11  PreparedStatement pst = null;
12  ResultSet rs = null;
13  try {
14  pst = statementProxy.getConnection().prepareStatement(selectSQL);
15  int index = 0;
16  for (Field pkField : beforeImage.pkRows()) {
17  index++;
18  pst.setObject(index, pkField.getValue(), pkField.getType());
19  }
20  rs = pst.executeQuery();
21  afterImage = TableRecords.buildRecords(tmeta, rs);
22
23  } finally {
24  if (rs != null) {
25  rs.close();
26  }
27  if (pst != null) {
28  pst.close();
29  }
30  }
31  return afterImage;
32 }
33
34
35 private String buildAfterImageSQL(TableMeta tableMeta, TableRecords beforeImage) throws S
QLException {
36
37  SQLUpdateRecognizer recognizer = (SQLUpdateRecognizer)sqlRecognizer;
38  //通过sql识别器去识别update更新 列数
39  List<String> updateColumns = recognizer.getUpdateColumns();
40  // sql 语句 select
41  StringBuilder prefix = new StringBuilder("SELECT ");
42  //判断更新的列数是否包含注解
43  if (!tableMeta.containsPK(updateColumns)) {
44  // PK should be included. select id,
45  prefix.append(getColumnNameInSQL(tableMeta.getPkName()) + ", ");
46  }
47  // from product where product_id=?
48  String suffix = " FROM " + getFromTableInSQL() + " WHERE " + buildWhereConditionByPKs(be
foreImage.pkRows());
49  //select id,count from product where product_id=?
50  StringJoiner selectSQLJoiner = new StringJoiner(", ", prefix.toString(), suffix);
51  for (String column : updateColumns) {
52  selectSQLJoiner.add(column);
53  }
54  return selectSQLJoiner.toString();
55 }
```

⑦前置后置镜像保存完毕，准备注册分支事务，提交本地事务

```
private void processGlobalTransactionCommit() throws SQLException {
 try {
 //调用netty 和seata-server 的注册接口,注意 在这里注册分支事务
 //注册回带入全局锁(表名:修改的字段值),若全局锁在服务端由冲突，回抛出异常
 register();
 } catch (TransactionException e) {
 recognizeLockKeyConflictException(e, context.buildLockKeys());
 }

 try {
 //保存我们的前置后置镜像到业务数据库中的undoLog表
 if (context.hasUndoLog()) {
 UndoLogManagerFactory.getUndoLogManager(this.getDbType()).flushUndoLogs(this);
 }
 //提交本地事务
 targetConnection.commit();
 } catch (Throwable ex) {
 LOGGER.error("process connectionProxy commit error: {}", ex.getMessage(), ex);
 //上报分支事务提交失败
 report(false);
 throw new SQLException(ex);
 }
 //上报分支事务本地提交失败
 report(true);
 context.reset();
}
```

A:注册分支事务

注册分支事务的seata-server的业务逻辑方法为

io.seata.server.coordinator.DefaultCoordinator#doBranchRegister

>io.seata.server.coordinator.DefaultCore#branchRegister

```
/**
 * 方法实现说明:该方法用于分支事务注册逻辑
 * @author:smlz
 * @return:
 * @exception:
 * @date:2019/12/11 23:35
 */
@Override
protected void doBranchRegister(BranchRegisterRequest request, BranchRegisterResponse resp
onse,
 RpcContext rpcContext) throws TransactionException {
 /**
 * 执行的业务逻辑是DefaultCore.branchRegister()来注册分支事务
```

```
13   */
14   response.setBranchId(
15   core.branchRegister(request.getBranchType(), request.getResourceId(), rpcContext.getClie
ntId(),
16   request.getXid(), request.getApplicationData(), request.getLockKey()));
17
18  }
```

真正的业务逻辑:

```
1  @Override
2  public Long branchRegister(BranchType branchType, String resourceId, String clientId, Stri
ng xid,
3   String applicationData, String lockKeys) throws TransactionException {
4   //获取一个全局事务
5   GlobalSession globalSession = assertGlobalSessionNotNull(xid);
6   //加锁执行
7   return globalSession.lockAndExcute(() -> {
8   //判断全局事务是否开启
9   if (!globalSession.isActive()) {
10   throw new GlobalTransactionException(GlobalTransactionNotActive,
11   String.format("Could not register branch into global session xid = %s status = %s", glob
alSession.getXid(), globalSession.getStatus()));
12   }
13   //SAGA type accept forward(retry) operation, forward operation will register remaining b
ranches
14   if (globalSession.getStatus() != GlobalStatus.Begin && !BranchType.SAGA.equals(branchTyp
e)) {
15   throw new GlobalTransactionException(GlobalTransactionStatusInvalid,
16   String.format("Could not register branch into global session xid = %s status = %s while
expecting %s", globalSession.getXid(), globalSession.getStatus(), GlobalStatus.Begin));
17   }
18   //添加监听器，用于监听分支事务的状态保存到数据库中
19   globalSession.addSessionLifecycleListener(SessionHolder.getRootSessionManager());
20   //获取创建一个分支事务对象
21   BranchSession branchSession = SessionHelper.newBranchByGlobal(globalSession, branchType,
resourceId,
22   applicationData, lockKeys, clientId);
23   /**
24   *真正的核心逻辑是这一块，申请全局锁的逻辑
25   */
26   if (!branchSession.lock()) {
27   throw new BranchTransactionException(LockKeyConflict,
28   String.format("Global lock acquire failed xid = %s branchId = %s",
globalSession.getXid(), branchSession.getBranchId()));
29   }
30   try {
31   //保存分支事务到数据库中
32   globalSession.addBranch(branchSession);
33   } catch (RuntimeException ex) {
34   branchSession.unlock();
35   throw new BranchTransactionException(FailedToAddBranch,
```

```
36   String.format("Failed to store branch xid = %s branchId = %s", globalSession.getXid(), b
ranchSession.getBranchId()));
37   }
38   LOGGER.info("Successfully register branch xid = {}, branchId = {}",
globalSession.getXid(), branchSession.getBranchId());
39   return branchSession.getBranchId();
40   });
41  }
42
43
```

全局锁申请(十分重要的业务逻辑)

```
1  public boolean lock() throws TransactionException {
2   //调用加锁管理器默认是:DefaultLockManager 来加锁
3   return LockerFactory.getLockManager().acquireLock(this);
4  }
5
6  //io.seata.server.lock.DefaultLockManager#acquireLock
7  public boolean acquireLock(BranchSession branchSession) throws TransactionException {
8   if (branchSession == null) {
9   throw new IllegalArgumentException("branchSession can't be null for memory/file
locker.");
10   }
11   //从分支事务对象中拿到 全局锁的key
12   //lockKey的格式为 表名:主键id product:1
13   //若是多个资源上锁 product:1,2,3.....
14   String lockKey = branchSession.getLockKey();
15   if (StringUtils.isNullOrEmpty(lockKey)) {
16   //no lock
17   return true;
18   }
19   //get locks of branch
20   //收集行锁
21   List<RowLock> locks = collectRowLocks(branchSession);
22   if (CollectionUtils.isEmpty(locks)) {
23   //no lock
24   return true;
25   }
26   return getLocker(branchSession).acquireLock(locks);
27  }
28
29  //io.seata.server.lock.AbstractLockManager#collectRowLocks(io.seata.server.session.Branch
Session)
30  protected List<RowLock> collectRowLocks(BranchSession branchSession) {
31   List<RowLock> locks = new ArrayList<>();
32   if (branchSession == null || StringUtils.isBlank(branchSession.getLockKey())) {
33   return locks;
34   }
35   //获取全局ID 192.168.159.1:8091:2029808902
```

```
36  String xid = branchSession.getXid();
37  //jdbc:mysql://localhost:3306/seata-product
38  String resourceId = branchSession.getResourceId();
39  //2029808902
40  long transactionId = branchSession.getTransactionId();
41  //拿到全局锁
42  String lockKey = branchSession.getLockKey();
43
44  return collectRowLocks(lockKey, resourceId, xid, transactionId, branchSession.getBranchI
d());
45 }
46
47 //封装行锁对象,就是把product:1,2封装成二个行锁对象
48 protected List<RowLock> collectRowLocks(String lockKey, String resourceId, String xid, Lo
ng transactionId,
49  Long branchID) {
50  List<RowLock> locks = new ArrayList<RowLock>();
51  product:1,2
52  String[] tableGroupedLockKeys = lockKey.split(";");
53  for (String tableGroupedLockKey : tableGroupedLockKeys) {
54  int idx = tableGroupedLockKey.indexOf(":");
55  if (idx < 0) {
56  return locks;
57  }
58  //获取表名 product
59  String tableName = tableGroupedLockKey.substring(0, idx);
60  //1,2
61  String mergedPKs = tableGroupedLockKey.substring(idx + 1);
62  if (StringUtils.isBlank(mergedPKs)) {
63  return locks;
64  }
65  [1,2]
66  String[] pks = mergedPKs.split(",");
67  if (pks == null || pks.length == 0) {
68  return locks;
69  }
70  for (String pk : pks) {
71  if (StringUtils.isNotBlank(pk)) {
72  RowLock rowLock = new RowLock();
73  rowLock.setXid(xid);
74  rowLock.setTransactionId(transactionId);
75  rowLock.setBranchId(branchID);
76  rowLock.setTableName(tableName);
77  rowLock.setPk(pk);
78  rowLock.setResourceId(resourceId);
79  locks.add(rowLock);
80  }
81  }
```

```
 }
 return locks;
}

//通过SPI的机制去读取meta-inf/service/io.seata.core.lock.Locker
读取我们的io.seata.server.lock.db.DataBaseLocker 数据库加锁器
通过io.seata.server.lock.db.DataBaseLocker 进行加锁
@Override
public boolean acquireLock(List<RowLock> locks) {
 if (CollectionUtils.isEmpty(locks)) {
 //no lock
 return true;
 }
 try {
 //真正的加锁逻辑
 return lockStore.acquireLock(convertToLockDO(locks));
 } catch (StoreException e) {
 throw e;
  } catch (Exception t) {
  LOGGER.error("AcquireLock error, locks:" + CollectionUtils.toString(locks), t);
  return false;
  }
}


==========================================真正加锁的逻辑
@Override
public boolean acquireLock(List<LockDO> lockDOs) {
 //准备数据库连接等
 Connection conn = null;
 PreparedStatement ps = null;
 ResultSet rs = null;
 List<LockDO> unrepeatedLockDOs = null;
 //数据库存在函数的keys
 Set<String> dbExistedRowKeys = new HashSet<>();
 boolean originalAutoCommit = true;
 try {
 conn = logStoreDataSource.getConnection();
 if (originalAutoCommit = conn.getAutoCommit()) {
 conn.setAutoCommit(false);
 }
 //check lock
 /**
 假如我们的分支事务 需要更新的资源是productId 为1 和2
 那么他的全局锁的key是product:1,2 而List<LockDO>.size为2
 所以下面代码拼接为(?,?) 若只有productId为1的那么他的拼接(?)
 **/
```

```
130
131  StringBuilder sb = new StringBuilder();
132  for (int i = 0; i < lockDOs.size(); i++) {
133  sb.append("?");
134  if (i != (lockDOs.size() - 1)) {
135  sb.append(", ");
136  }
137  }
138  boolean canLock = true;
139  //执行查询我们的lock_table
140  String checkLockSQL = LockStoreSqls.getCheckLockableSql(lockTable, sb.toString(), dbTyp
e);
141  ps = conn.prepareStatement(checkLockSQL);
142  //预编译参数
143  for (int i = 0; i < lockDOs.size(); i++) {
144  ps.setString(i + 1, lockDOs.get(i).getRowKey());
145  }
146  rs = ps.executeQuery();
147  //获取全局事务参与者所属的分布式事务全局ID
148  String currentXID = lockDOs.get(0).getXid();
149  //循环数据库
150  while (rs.next()) {
151  //通过查询到数据库中的
152  String dbXID = rs.getString(ServerTableColumnsName.LOCK_TABLE_XID);
153  //锁当前的和数据库中的全局事务ID不相等，说明其他的分布式事务真正对该记录进行操作
154  // 线程1对应的分布式事务 对product:1 进行加锁操作
155  //线程2对应的分布式事务对product:1进行操作,那么去数据库查询出来的全局
156  //xid和当前线程2对应的xid是不相等的,那么就加锁失败。
157  if (!StringUtils.equals(dbXID, currentXID)) {
158  if (LOGGER.isInfoEnabled()) {
159  String dbPk = rs.getString(ServerTableColumnsName.LOCK_TABLE_PK);
160  String dbTableName = rs.getString(ServerTableColumnsName.LOCK_TABLE_TABLE_NAME);
161  Long dbBranchId = rs.getLong(ServerTableColumnsName.LOCK_TABLE_BRANCH_ID);
162  LOGGER.info("Global lock on [{}:{}] is holding by xid {} branchId {}", dbTableName, dbP
k, dbXID,
163  dbBranchId);
164  }
165  //取反操作，直接加锁失败
166  canLock &= false;
167  //跳出循环
168  break;
169  }
170
171  //收集数据库中操作的行锁的key
172  dbExistedRowKeys.add(rs.getString(ServerTableColumnsName.LOCK_TABLE_ROW_KEY));
173  }
174  //加锁失败直接返回
175  if (!canLock) {
```

```
176   conn.rollback();
177   return false;
178   }
179   //线程1：分布式事务product:1,那么数据库加锁成功
180   //后面线程1:有执行一次语句 product:1,2 那么dbExistedRowKeys[1]
181   if (CollectionUtils.isNotEmpty(dbExistedRowKeys)) {
182   //lockDOs[1,2] 过滤后unrepeatedLockDOs [2]
183   unrepeatedLockDOs = lockDOs.stream().filter(lockDO ->
!dbExistedRowKeys.contains(lockDO.getRowKey()))
184   .collect(Collectors.toList());
185   } else {
186   unrepeatedLockDOs = lockDOs;
187   }
188   if (CollectionUtils.isEmpty(unrepeatedLockDOs)) {
189   conn.rollback();
190   return true;
191   }
192
193   //lock
194   for (LockDO lockDO : unrepeatedLockDOs) {
195   //进行加锁,说白了就是保存数据库
196   if (!doAcquireLock(conn, lockDO)) {
197   if (LOGGER.isInfoEnabled()) {
198   LOGGER.info("Global lock acquire failed, xid {} branchId {} pk {}", lockDO.getXid(),
199   lockDO.getBranchId(), lockDO.getPk());
200   }
201   conn.rollback();
202   return false;
203   }
204   }
205   conn.commit();
206   return true;
207   } catch (SQLException e) {
208   throw new StoreException(e);
209   } finally {
210   if (rs != null) {
211   try {
212   rs.close();
213   } catch (SQLException e) {
214   }
215   }
216   if (ps != null) {
217   try {
218   ps.close();
219   } catch (SQLException e) {
220   }
221   }
222   if (conn != null) {
```

```
223   try {
224   if (originalAutoCommit) {
225   conn.setAutoCommit(true);
226   }
227   conn.close();
228   } catch (SQLException e) {
229   }
230   }
231   }
232  }
233
```

全局事务提交(由我们的TM执行完毕业务逻辑没有抛出异常发起全局提交)

seata-server端代码

io.seata.server.coordinator.DefaultCoordinator#doGlobalCommit

>io.seata.server.coordinator.DefaultCoordinator#core

```
1  protected void doGlobalCommit(GlobalCommitRequest request, GlobalCommitResponse response,
RpcContext rpcContext)
2   throws TransactionException {
3   //真正的核心业务逻辑 调用DefaultCore的commit
4   response.setGlobalStatus(core.commit(request.getXid()));
5  }
6
```

io.seata.server.coordinator.DefaultCoordinator#core

```
1  public GlobalStatus commit(String xid) throws TransactionException {
2   //数据库查询GlobalTable
3   GlobalSession globalSession = SessionHolder.findGlobalSession(xid);
4   if (globalSession == null) {
5   return GlobalStatus.Finished;
6   }
7   //添加一个session监听 用于操作数据库
8   globalSession.addSessionLifecycleListener(SessionHolder.getRootSessionManager());
9   // just lock changeStatus
10   boolean shouldCommit = globalSession.lockAndExcute(() -> {
11   //close and clean
12   //close操作 就是修改GlobalSession的标志为 active为false，防止后续的分支事务注册上来
13   //clean 释放全局锁，
14   globalSession.closeAndClean(); // Highlight: Firstly, close the session, then no more br
anch can be registered.
15   if (globalSession.getStatus() == GlobalStatus.Begin) {
16   //修改全局事务的状态
17   globalSession.changeStatus(GlobalStatus.Committing);
18   return true;
19   }
20   return false;
21   });
```

```
22  if (!shouldCommit) {
23  return globalSession.getStatus();
24  }
25  if (globalSession.canBeCommittedAsync()) {
26  asyncCommit(globalSession);
27  return GlobalStatus.Committed;
28  } else {
29  doGlobalCommit(globalSession, false);
30  }
31  return globalSession.getStatus();
32 }
```