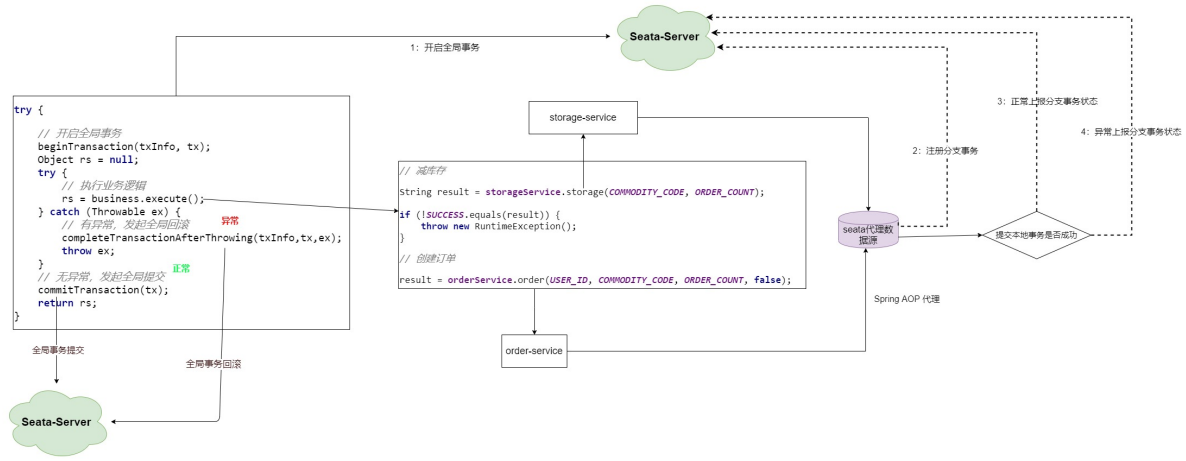
Seata AT事务底层原理分析

Seata 主线流程图

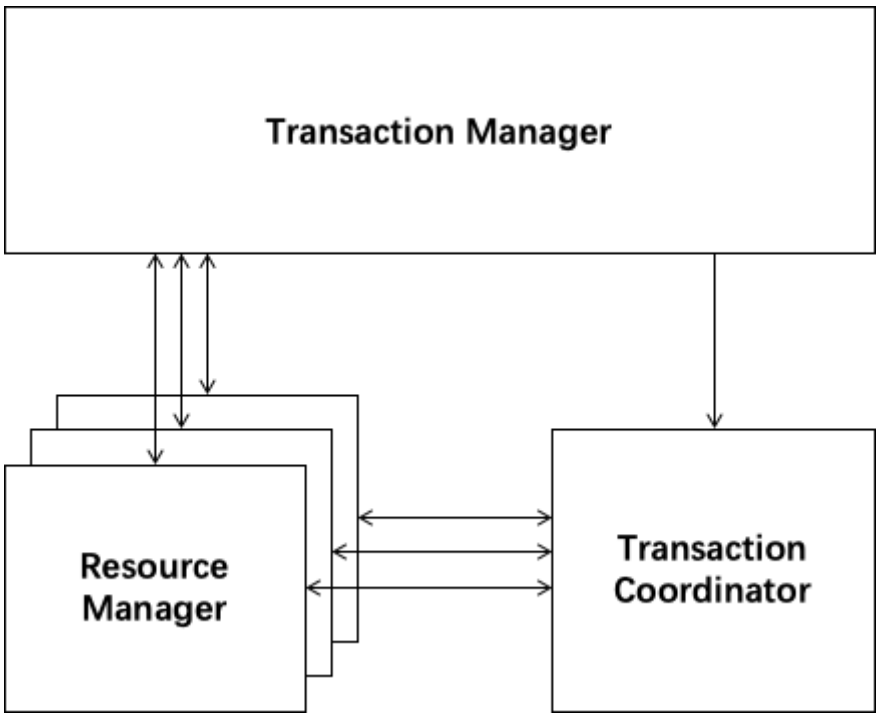


Seata AT事务详解

> AT 模式下，把每个数据库被当做是一个 Resource，Seata 里称为 DataSource Resource。业务通过 JDBC 标准接口访问数据库资源时，Seata 框架会对所有请求进行拦截，做一些操作。每个本地事务提交时，Seata RM（Resource Manager，资源管理器） 都会向 TC（Transaction Coordinator，事务协调器） 注册一个分支事务。当请求链路调用完成后，发起方通知 TC 提交或回滚分布式事务，进入二阶段调用流程。此时，TC 会根据之前注册的分支事务回调到对应参与者去执行对应资源的第二阶段。TC 是怎么找到分支事务与资源的对应关系呢？每个资源都有一个全局唯一的资源 ID，并且在初始化时用该 ID 向 TC 注册资源。在运行时，每个分支事务的注册都会带上其资源 ID。这样 TC 就能在二阶段调用时正确找到对应的资源。

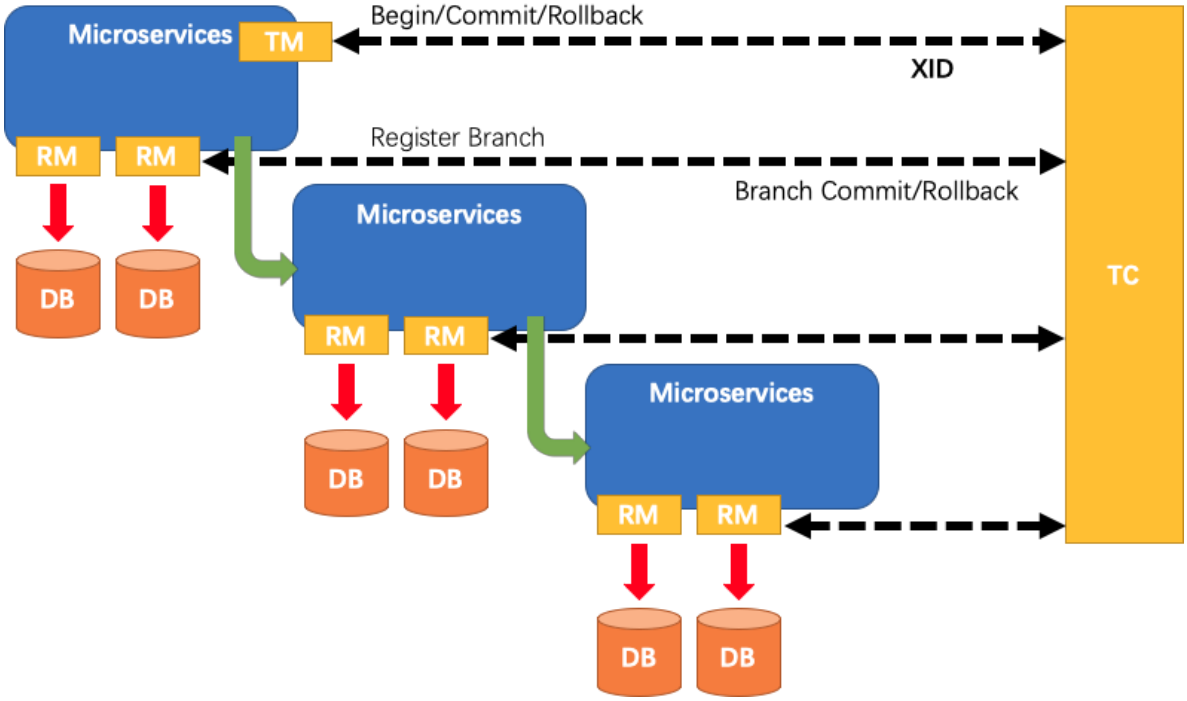
Seata重要组件

- 事务协调器（TC）： 维护全局事务和分支事务的状态，驱动全局提交或回滚。
- 事务管理器（TM）：定义全局事务的范围，开始全局事务，提交或回滚全局事务。
- 资源管理器（RM）：管理正在处理的分支事务的资源，与TC对话以注册分支事务并报告分支事务的状态，并驱动分支事务的提交或回滚。



Seata管理的分布式事务的典型生命周期：

1. TM要求TC开始一项新的全局事务。TC生成代表全局事务的XID，TC 也作为 Seata 的服务端独立部署。
2. XID通过微服务的调用链传播。
3. RM将本地事务注册为XID到TC的相应全局事务的分支。
4. TM要求TC提交或回退相应的XID全局事务。
5. TC驱动XID的相应全局事务下的所有分支事务以完成分支提交或回滚



在 Seata 中，分布式事务的执行流程：

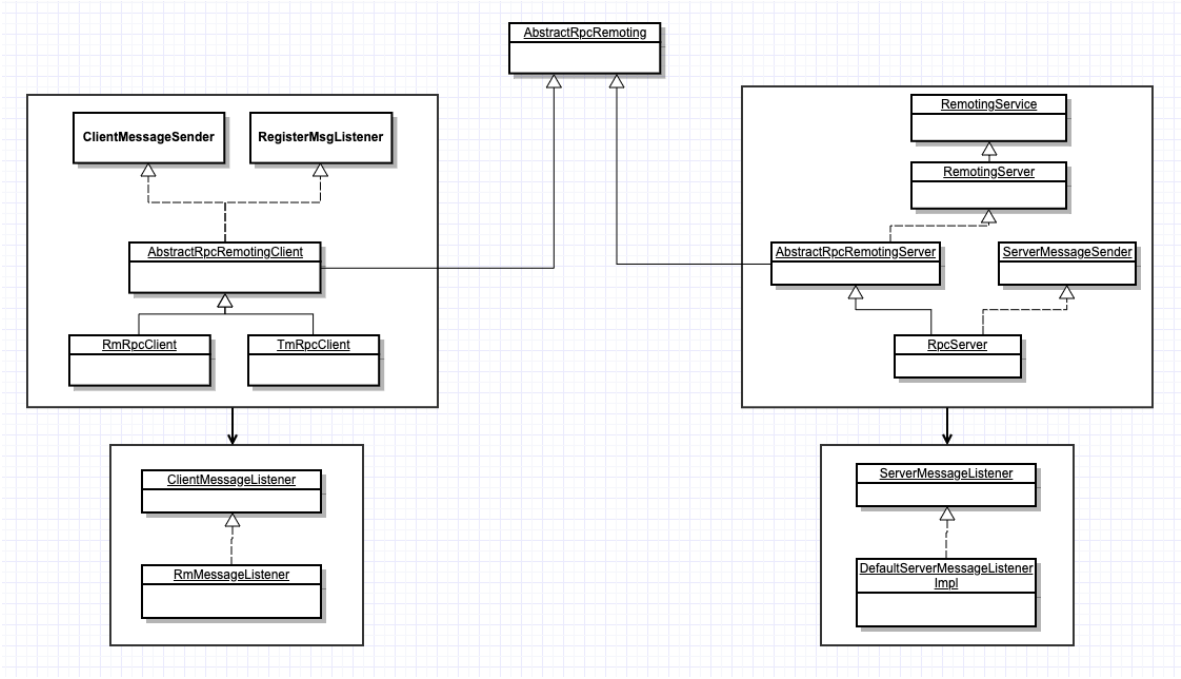
- TM 开启分布式事务（TM 向 TC 注册全局事务记录）；
- 按业务场景，编排数据库、服务等事务内资源（RM 向 TC 汇报资源准备状态 ）；
- TM 结束分布式事务，事务一阶段结束（TM 通知 TC 提交/回滚分布式事务）；
- TC 汇总事务信息，决定分布式事务是提交还是回滚；

- TC 通知所有 RM 提交/回滚 资源，事务二阶段结束。

Seata类图分析

TC类图

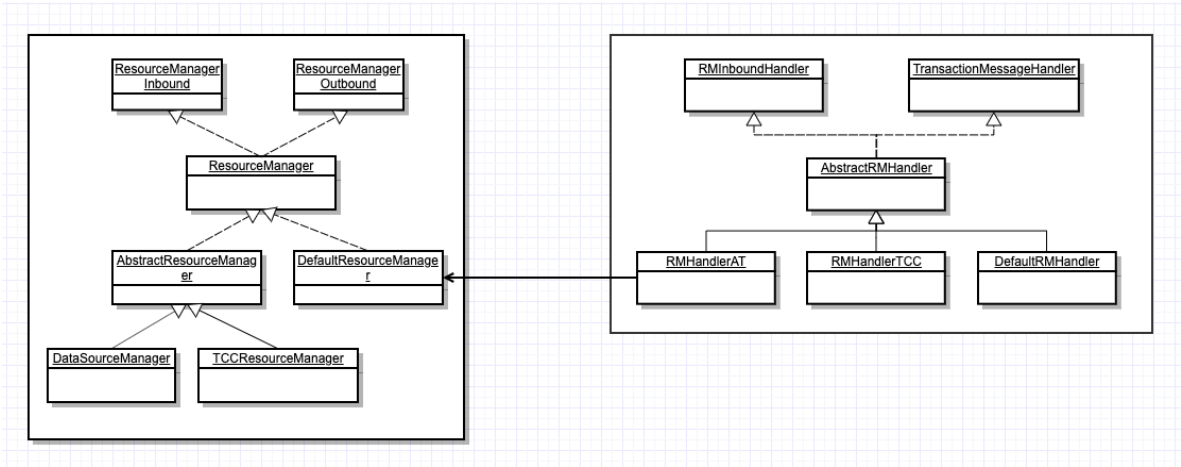
TC在Seata中其实就是Server端，Client端其实属于RM和TM，但是为了看起来完整，把Client端的RpcCient和Listener也放在了这里，类图如下图所示：



CLient listenner监听服务端发过来的消息，从而进行相应的操作。

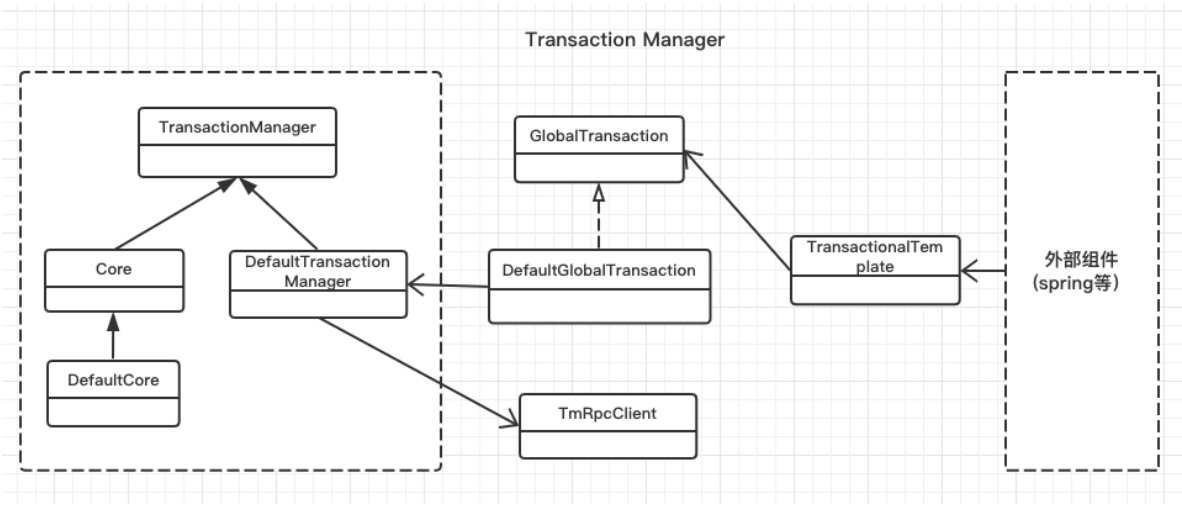
RM类图

RM相关类图如下图所示：



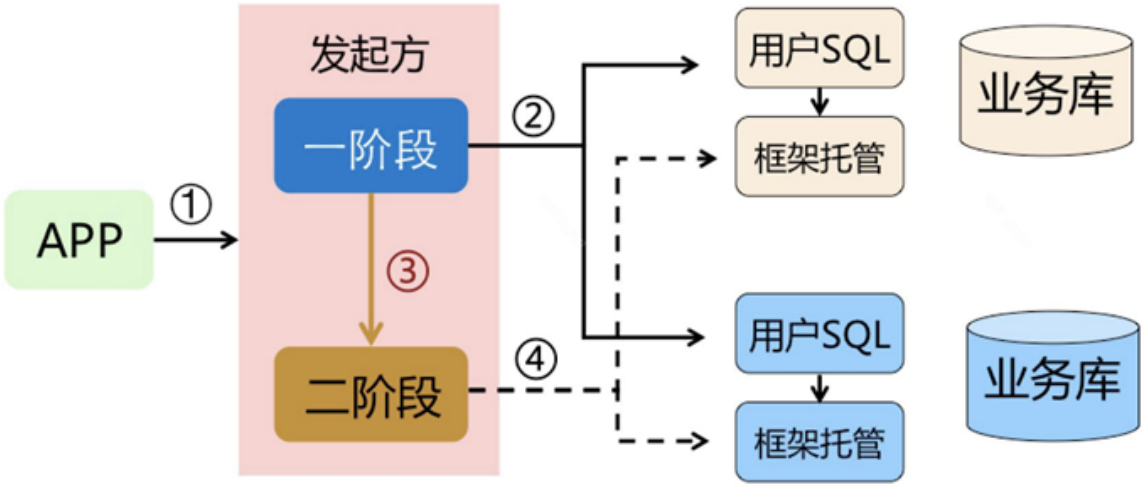
TM类图

TM相关类图如下图所示：



AT模式

AT 模式是一种无侵入的分布式事务解决方案。在 AT 模式下，用户只需关注自己的“业务 SQL”，用户的“业务 SQL”作为一阶段，Seata 框架会自动生成事务的二阶段提交和回滚操作。



在分析AT模式前，我们先提问几个问题：

1、分布式Seata，怎么在系统中集成使用？ 2、开始事务前，做了什么初始化工作？ 3、AT模式具体怎么实现的？

初始化工作

Seata在GlobalTransactionScanner中进行了TM和RM初始化：

```
private void initClient() {
    //init TM
    TMClient.init(applicationId, txServiceGroup);

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

    //Register Spring ShutdownHook
    registerSpringShutdownHook();
}
```

其中TMClient的初始化如下：

```
public static void init(String applicationId, String transactionServiceGroup) {
    TmRpcClient tmRpcClient = TmRpcClient.getInstance(applicationId,
transactionServiceGroup);
    tmRpcClient.init();
}
```

其中初始化了一个TmRpcClient对象，它的init方法中，会启动与Server的重连接线程（每5秒），以及消息发送线程和超时线程。重连接是按照配置方式，寻找Server信息，比如采用file形式，那么首先从file.conf中根据分组名称(service_group)找到集群名称(cluster_name)，再根据集群名称找到seata-server集群ip端口列表，然后从ip列表中选择一个用netty进行连接。

RmClient的初始化如下：

```
public static void init(String applicationId, String transactionServiceGroup) {
    RmRpcClient rmRpcClient = RmRpcClient.getInstance(applicationId,
transactionServiceGroup);
    rmRpcClient.setResourceManager(DefaultResourceManager.get());
    rmRpcClient.setClientMessageListener(new
RmMessageListener(DefaultRMHandler.get()));
    rmRpcClient.init();
}
```

其中，主要工作为：

- 创建一个RmRpcClient对象
- RmRpcClient对象设置ResourceManager;
- RmRpcClient对象设置消息Listener;
- RmRpcClient对象初始化

初始化阶段总结： 1、Spring启动时，初始化了2个客户端TmClient、RmClient 2、TmClient与Server通过Netty建立连接并发送消息 3、RmClient与Server通过Netty建立连接，负责接收二阶段提交、回滚消息并在回调器(RmHandler)中做处理

因此，使用Seata，需要初始化GlobalTransactionScanner进行初始化工作!

TM全局事务控制

哪些方法需要进行全局事务控制？Seata定义了注解GlobalTransactional,只要添加了注解的方法，就代表需要分布式事务。

注解对应着拦截器，Seata定义的拦截器为GlobalTransactionalInterceptor。其核心处理逻辑如下：

```
@Override
public Object invoke(final MethodInvocation methodInvocation) throws Throwable {
    Class<?> targetClass = (methodInvocation.getThis() != null ?
AopUtils.getTargetClass(methodInvocation.getThis()) : null);
    Method specificMethod =
ClassUtils.getMostSpecificMethod(methodInvocation.getMethod(), targetClass);
    final Method method =
BridgeMethodResolver.findBridgedMethod(specificMethod);

    final GlobalTransactional globalTransactionalAnnotation =
getAnnotation(method, GlobalTransactional.class);
    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();
    }
}
```

上面方法的逻辑为：如果业务方法上有全局事务GlobalTransactional注解时，就执行全局事务处理handleGlobalTransaction；如果业务方法上有全局锁GlobalLock注解时，就执行全局锁处理handleGlobalLock；否则，就按照普通方法执行。

handleGlobalTransaction中，会调用TransactionalTemplate的execute方法，execute方法如下：

```
public Object execute(TransactionalExecutor business) throws Throwable {
    // 1. get or create a transaction
    GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();

    // 1.1 get transactionInfo
    TransactionInfo txInfo = business.getTransactionInfo();
    if (txInfo == null) {
        throw new ShouldNeverHappenException("transactionInfo does not exist");
    }
    try {
        // 2. begin transaction
        beginTransaction(txInfo, tx);

        Object rs = null;
        try {
            // 3. Do Your Business
            rs = business.execute();
        } catch (Throwable ex) {
            // 4.the needed business exception to rollback.
            completeTransactionAfterThrowing(txInfo,tx,ex);
            throw ex;
        }

        // 5. everything is fine, commit.
        commitTransaction(tx);

        return rs;
    } finally {
        //6. clear
        triggerAfterCompletion();
        cleanUp();
    }
}
```

1、创建或创建一个GlobalTransaction，并根据业务TransactionalExecutor获取到事务需要的相关信息；

2、开始全局事务

开始全局事务的方法为beginTransaction方法，方法内部调用了DefaultGlobalTransaction.begin方法，而DefaultGlobalTransaction.begin方法的内部又调用了TransactionManager的begin方法，

```
xid = transactionManager.begin(null, null, name, timeout);
```

TransactionManager接口类的默认实现为DefaultTransactionManager，其begin方法为：

```
public String begin(String applicationId, String transactionServiceGroup, String
name, int timeout)
    throws TransactionException {
    GlobalBeginRequest request = new GlobalBeginRequest();
    request.setTransactionName(name);
    request.setTimeout(timeout);
    GlobalBeginResponse response = (GlobalBeginResponse)syncCall(request);
    return response.getXid();
}
```

方法中，向服务器发起开始全局事务请求，服务会返回一个全局事务标示XID。OK，此时，由TM发起的一个全局事务就开始了！

3、执行业务逻辑

4、回滚全局事务

如果业务都执行没有全部成功，那么就执行全局事务回滚，方法为completeTransactionAfterThrowing。方法执行逻辑与开始事务方法内部逻辑基本一致：内部调用了DefaultGlobalTransaction.rollback方法，而DefaultGlobalTransaction.rollback方法的内部又调用了TransactionManager的rollback方法，向服务器发起rollback，服务会返回GlobalStatus.

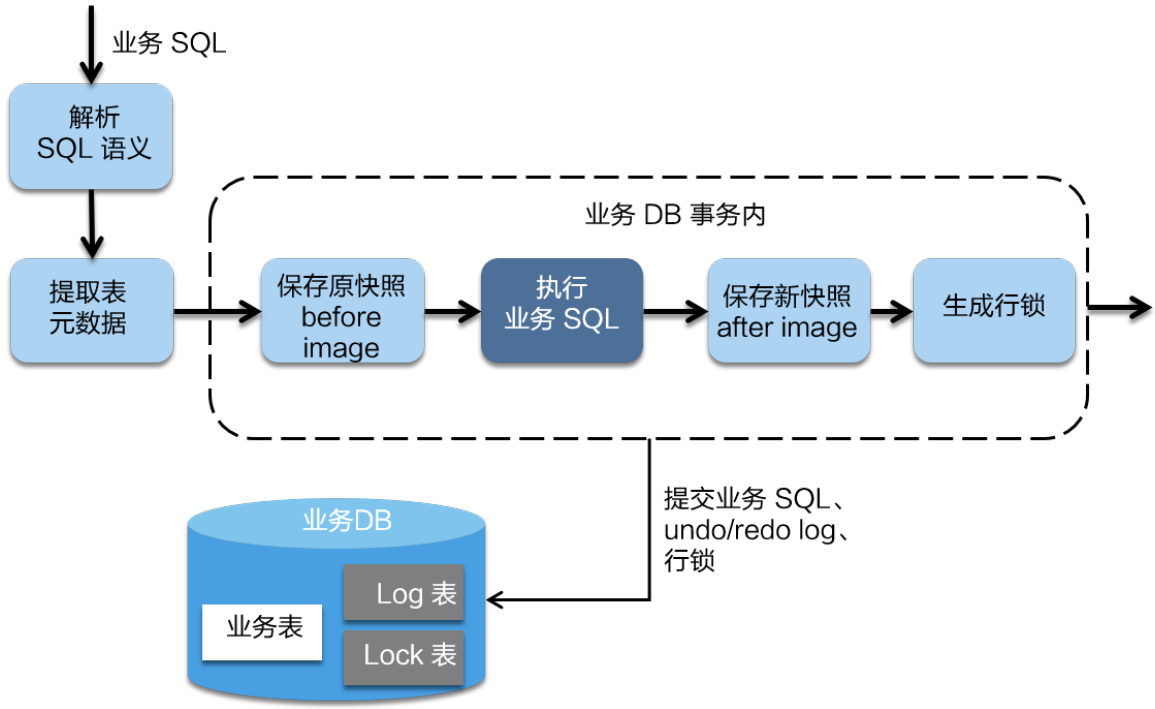
5、提交全局事务

如果业务都执行成功，那么就全局commit，方法为commitTransaction。方法执行逻辑与开始事务方法和回滚事务内部逻辑基本一致：内部调用了DefaultGlobalTransaction.commit方法，而DefaultGlobalTransaction.commit方法的内部又调用了TransactionManager的commit方法，向服务器发起Commit，服务会返回GlobalStatus.

6、清理 完成整个事务后，进行相关资源变量清理。

AT模式一阶段

在一阶段，Seata 会拦截“业务 SQL”，首先解析 SQL 语义，找到“业务 SQL”要更新的业务数据，在业务数据被更新前，将其保存成“before image”，然后执行“业务 SQL”更新业务数据，在业务数据更新之后，再将其保存成“after image”，最后生成行锁。以上操作全部在一个数据库事务内完成，这样保证了一阶段操作的原子性。



实现上，Seata对数据源做了封装代理，然后对于数据源的操作处理，就由Seata内部逻辑完成了。看一个Demo中的数据源加载配置：

```
@Bean(name = "order")
public DataSourceProxy masterDataSourceProxy(@Qualifier("originOrder")
DataSource dataSource) {
    return new DataSourceProxy(dataSource);
}
```

从Demo中可以看到，我们使用的是Seata封装的代理数据源DataSourceProxy。DataSourceProxy初始化时，会进行Resouce注册：

```
private void init(DataSource dataSource, String resourceGroupId) {
    this.resourceGroupId = resourceGroupId;
    Connection connection = dataSource.getConnection()
    jdbcUrl = connection.getMetaData().getURL();
    dbType = JdbcUtils.getDbType(jdbcUrl, null);
    DefaultResourceManager.get().registerResource(this);
}
```

Seata除了对数据库的DataSource进行了封装，同样也对Connection，Statement进行了封装代理，分别为ConnectionProxy和StatementProxy。

DataSourceProxy中重写了DataSource的getConnection方法，以此来获得ConnectionProxy，方法如下：

```
@Override
public ConnectionProxy getConnection() throws SQLException {
    Connection targetConnection = targetDataSource.getConnection();
    return new ConnectionProxy(this, targetConnection);
}
```

ConnectionProxy代理了Sql connection。我们重点关于一下其createStatement方法，prepareStatement方法，commit方法，rollback方法以及setAutoCommit方法。

ConnectionProxy的createStatement会返回一个代理的StatementProxy，如下：

```
@Override
public Statement createStatement() throws SQLException {
    Statement targetStatement = getTargetConnection().createStatement();
    return new StatementProxy(this, targetStatement);
}
```

ConnectionProxy的prepareStatement同样会返回一个代理的PreparedStatementProxy，如下：

```
@Override
public PreparedStatement prepareStatement(String sql) throws SQLException {
    PreparedStatement targetPreparedStatement =
getTargetConnection().prepareStatement(sql);
    return new PreparedStatementProxy(this, targetPreparedStatement, sql);
}
```

StatementProxy和PreparedStatementProxy是对执行Sql的Statement的一个封装，两者基本一样，我们看看StatementProxy的execute方法：

```
@Override
public boolean execute(String sql) throws SQLException {
    this.targetSQL = sql;
    return ExecuteTemplate.execute(this, new StatementCallback<Boolean, T>() {
        @Override
        public Boolean execute(T statement, Object... args) throws SQLException
{
            return statement.execute((String) args[0]);
        }
    }, sql);
}
```

可以看到，内部交给了ExecuteTemplate执行，ExecuteTemplate的execute方法如下：

```
public static <T, S extends Statement> T execute(SQLRecognizer sqlRecognizer,
                                                 StatementProxy<S> statementProxy,
                                                     StatementCallback<T, S>
statementCallback,
                                                     Object... args) throws
SQLException {

    if (!RootContext.inGlobalTransaction() && !RootContext.requireGlobalLock())
{
        // Just work as original statement
        return statementCallback.execute(statementProxy.getTargetStatement(),
args);
    }

    if (sqlRecognizer == null) {
        sqlRecognizer = SQLVisitorFactory.get(
                statementProxy.getTargetSQL(),
                statementProxy.getConnectionProxy().getDbType());
    }
    Executor<T> executor = null;
    if (sqlRecognizer == null) {
```

```
            executor = new PlainExecutor<T, S>(statementProxy, statementCallback);
        } else {
            switch (sqlRecognizer.getSQLType()) {
                case INSERT:
                    executor = new InsertExecutor<T, S>(statementProxy,
statementCallback, sqlRecognizer);
                    break;
                case UPDATE:
                    executor = new UpdateExecutor<T, S>(statementProxy,
statementCallback, sqlRecognizer);
                    break;
                case DELETE:
                    executor = new DeleteExecutor<T, S>(statementProxy,
statementCallback, sqlRecognizer);
                    break;
                case SELECT_FOR_UPDATE:
                    executor = new SelectForUpdateExecutor<T, S>(statementProxy,
statementCallback, sqlRecognizer);
                    break;
                default:
                    executor = new PlainExecutor<T, S>(statementProxy,
statementCallback);
                    break;
            }
        }
        T rs = null;
        try {
            rs = executor.execute(args);
        } catch (Throwable ex) {
            if (!(ex instanceof SQLException)) {
                // Turn other exception into SQLException
                ex = new SQLException(ex);
            }
            throw (SQLException)ex;
        }
        return rs;
    }
```

方法内部流程如下： 1）先判断是否开启了全局事务，如果没有，不走代理； 2）调用SQLVisitorFactory对目标sql进行解析 3）针对特定类型sql操作(INSERT,UPDATE,DELETE,SELECT_FOR_UPDATE)等进行特殊解析 4）执行sql并返回结果

InsertExecutor,UpdateExecutor,DeleteExecutor均继承自AbstractDMLBaseExecutor，AbstractDMLBaseExecutor又继承自BaseTransactionalExecutor，BaseTransactionalExecutor又继承自Executor。其中execute方法实现为：

```
@Override
public Object execute(Object... args) throws Throwable {
    if (RootContext.inGlobalTransaction()) {
        String xid = RootContext.getXID();
        statementProxy.getConnectionProxy().bind(xid);
    }

    if (RootContext.requireGlobalLock()) {
        statementProxy.getConnectionProxy().setGlobalLockRequire(true);
    } else {
        statementProxy.getConnectionProxy().setGlobalLockRequire(false);
```

```
    }
    return doExecute(args);
}
```

其中doExecute方法如下：

```
@Override
public T doExecute(Object... args) throws Throwable {
    AbstractConnectionProxy connectionProxy =
statementProxy.getConnectionProxy();
    if (connectionProxy.getAutoCommit()) {
        return executeAutoCommitTrue(args);
    } else {
        return executeAutoCommitFalse(args);
    }
}
```

其中executeAutoCommitFalse方法为：

```
protected T executeAutoCommitFalse(Object[] args) throws Throwable {
    TableRecords beforeImage = beforeImage();
    T result = statementCallback.execute(statementProxy.getTargetStatement(),
args);
    TableRecords afterImage = afterImage(beforeImage);
    prepareUndoLog(beforeImage, afterImage);
    return result;
}
```

其中首先获取执行Sql前的beforeImage，然后执行Sql,执行晚，获取执行后的afterImage，并将执行前后Image赋给ConnectionProxy。注意插入，删除，更新SQl语句，获取beforeImage和afterImage的方法不同，因此定义了InsertExecutor,UpdateExecutor,DeleteExecutor。

执行完Sql，此时需要提交，ConnectionProxy的commit方法如下：

```
@Override
public void commit() throws SQLException {
    if (context.inGlobalTransaction()) {
        processGlobalTransactionCommit();
    } else if (context.isGlobalLockRequire()) {
        processLocalCommitWithGlobalLocks();
    } else {
        targetConnection.commit();
    }
}
```

方法逻辑为： 1）如果处于全局事务中，则调用processGlobalTransactionCommit()处理全局事务提交 2）如果加了全局锁注解，加全局锁并提交 3）如果没有对应注释，按直接进行事务提交

重点关注一下processGlobalTransactionCommit代码：

```
private void processGlobalTransactionCommit() throws SQLException {
    try {
        register();
    } catch (TransactionException e) {
        recognizeLockKeyConflictException(e);
```

```
    }

    try {
        if (context.hasUndoLog()) {
            UndoLogManager.flushUndoLogs(this);
        }
        targetConnection.commit();
    } catch (Throwable ex) {
        report(false);
        if (ex instanceof SQLException) {
            throw new SQLException(ex);
        }
    }
    report(true);
    context.reset();
}
```

流程分为如下几步：

1、注册分支事务register()，并将branchId分支id绑定到上下文中：

```
private void register() throws TransactionException {
    Long branchId = DefaultResourceManager.get().branchRegister(BranchType.AT,
getDataSourceProxy().getResourceId(),
            null, context.getXid(), null, context.buildLockKeys());
    context.setBranchId(branchId);
}
```

2、如果包含undolog，则将之前绑定到上下文中的undolog进行入库：

```
UndoLogManager.flushUndoLogs(this);
```

3、提交本地事务；

4、如果操作失败，report()中通过RM提交Branch失败消息，如果成功，report()提交Branch成功消息，其中Report方法如下：

```
private void report(boolean commitDone) throws SQLException {
    int retry = REPORT_RETRY_COUNT;
    while (retry > 0) {
        try {
          DefaultResourceManager.get().branchReport(BranchType.AT,
context.getXid(), context.getBranchId(),
                    (commitDone ? BranchStatus.PhaseOne_Done :
BranchStatus.PhaseOne_Failed), null);
            return;
        } catch (Throwable ex) {
            retry--;
            if (retry == 0) {
                throw new SQLException("Failed to report branch status " +
commitDone, ex);
            }
        }
    }
}
```

AT模式二阶段

AT时分为两个阶段的，第一阶段，就是各个阶段本地提交操作；第二阶段会根据第一阶段的情况决定是进行全局提交还是全局回滚操作。

全局提交回滚操作由TM发起，具体为，如果Branch执行没有出现异常，那么就表明各个Branch均执行成功，即进行全局提交，如果某个Branch执行时出现异常，那么就需要进行全局回滚。代码即为TransactionalTemplate的execute的第4步和5步，如下：

```
try {
    // 3. Do Your Business
    rs = business.execute();
} catch (Throwable ex) {
    // 4.the needed business exception to rollback.
    completeTransactionAfterThrowing(txInfo,tx,ex);
    throw ex;
}

// 5. everything is fine, commit.
commitTransaction(tx);

return rs;
```

Seata Serer 收到TM发送的提交或者回滚请求后，就会向各个RM 发送消息，让他们执行相应的提交或者回滚操作。 RM接收消息是通过RMHandlerAT，在初始化RmListrenner时，指定了RMHandlerAT，其中处理方法为：

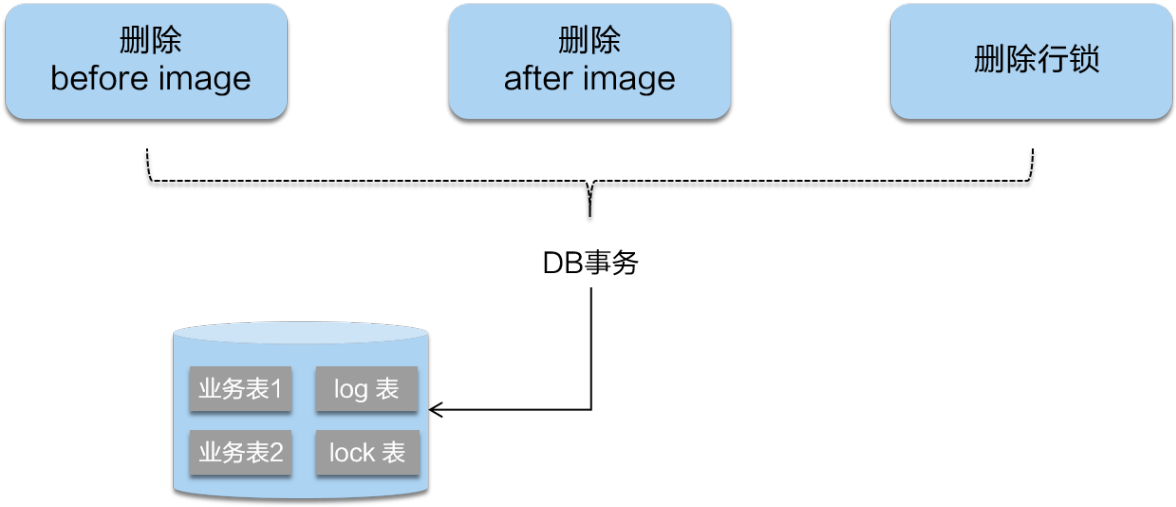
```
@Override
public BranchCommitResponse handle(BranchCommitRequest request) {
    BranchCommitResponse response = new BranchCommitResponse();
    exceptionHandleTemplate(new AbstractCallback<BranchCommitRequest,
BranchCommitResponse>() {
        @Override
        public void execute(BranchCommitRequest request, BranchCommitResponse
response)
            throws TransactionException {
            doBranchCommit(request, response);
        }
    }, request, response);
    return response;
}

@Override
public BranchRollbackResponse handle(BranchRollbackRequest request) {
    BranchRollbackResponse response = new BranchRollbackResponse();
    exceptionHandleTemplate(new AbstractCallback<BranchRollbackRequest,
BranchRollbackResponse>() {
        @Override
        public void execute(BranchRollbackRequest request,
BranchRollbackResponse response)
            throws TransactionException {
            doBranchRollback(request, response);
        }
    }, request, response);
    return response;
```

```
}
```

二阶段提交

二阶段如果是提交的话，因为“业务 SQL”在一阶段已经提交至数据库， 所以 Seata 框架只需将一阶段保存的快照数据和行锁删掉，完成数据清理即可。



如果所有Branch RM都执行成功了，那么就进行全局Commit。因为此时我们不用回滚，而每个Branch本地数据库操作已经完成了，那么我们其实主要做的事情就是把本地的Undolog删了即可，看看Seata内部实现逻辑。

doBranchCommit方法内部会调用ResourceManager的branchCommit方法：

```
BranchStatus status = getResourceManager().branchCommit(request.getBranchType(),
xid, branchId, resourceId,applicationData);
```

根据Seata SPI，其实就是执行的DataSourceManager类的branchCommit，如下：

```
@Override
public BranchStatus branchCommit(BranchType branchType, String xid, long
branchId, String resourceId, String applicationData) throws TransactionException
{
    return asyncWorker.branchCommit(branchType, xid, branchId, resourceId,
applicationData);
}
```

DataSourceManager中调用了asyncWorker来异步提交，看下AsyncWorker中branchCommit方法:

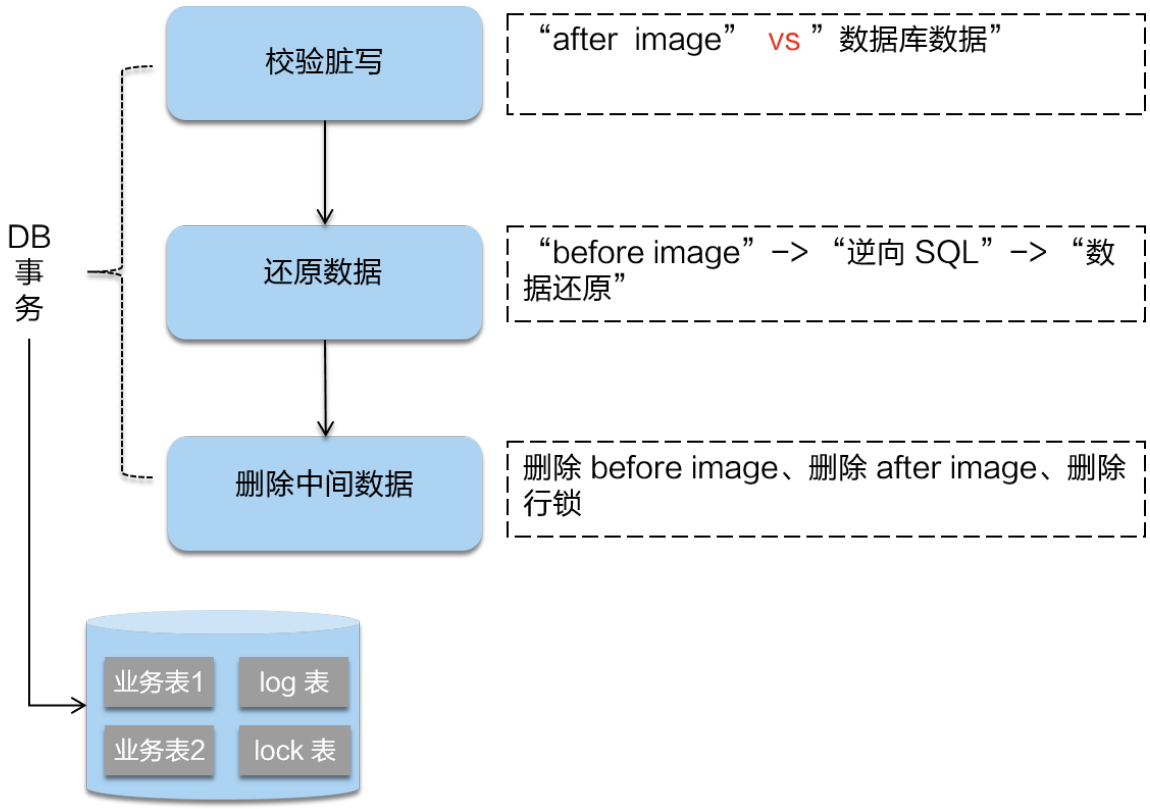
```
@Override
public BranchStatus branchCommit(BranchType branchType, String xid, long
branchId, String resourceId,
                                 String applicationData) throws
TransactionException {
    if (!ASYNC_COMMIT_BUFFER.offer(new Phase2Context(branchType, xid, branchId,
resourceId, applicationData))) {
        LOGGER.warn("Async commit buffer is FULL. Rejected branch [" + branchId
+ "/" + xid
            + "] will be handled by housekeeping later.");
    }
    return BranchStatus.PhaseTwo_Committed;
}
```

方法中往ASYNC_COMMIT_BUFFER缓冲List中新增了一个二阶段提交的context，那么真正branchCommit在哪里了?

其实是在AsyncWorker的init()方法中，Init中会起一个定时线程，每一秒执行一次，线程中的主要逻辑为： 1）先按resourceId(也就是数据连接)对提交操作进行分组，一个数据库的可以一起操作，提升效率； 2） 根据resourceId找到对应DataSourceProxy，并获取一个普通的数据库连接getPlainConnection()，估计这本身不需要做代理操作，故用了普通的数据库连接； 3）调用UndoLogManager.deleteUndoLog(commitContext.xid, commitContext.branchId, conn)删除undolog

二阶段回滚

二阶段如果是回滚的话，Seata 就需要回滚一阶段已经执行的“业务 SQL”，还原业务数据。回滚方式便是用“before image”还原业务数据；但在还原前要首先要校验脏写，对比“数据库当前业务数据”和“after image”，如果两份数据完全一致就说明没有脏写，可以还原业务数据，如果不一致就说明有脏写，出现脏写就需要转人工处理。



AT 模式的一阶段、二阶段提交和回滚均由 Seata 框架自动生成，用户只需编写“业务 SQL”，便能轻松接入分布式事务，AT 模式是一种对业务无任何侵入的分布式事务解决方案。

如果某个RM Branch执行失败了，那么就要进行全局回滚。RMHandlerAT收到回滚消息，进行回滚处理，内部会调用DataSourceManager的branchRollback方法：

```
@Override
public BranchStatus branchRollback(BranchType branchType, String xid, long
branchId, String resourceId, String applicationData) throws TransactionException
{
    DataSourceProxy dataSourceProxy = get(resourceId);
    if (dataSourceProxy == null) {
        throw new ShouldNeverHappenException();
    }
    try {
        UndoLogManager.undo(dataSourceProxy, xid, branchId);
    } catch (TransactionException te) {
        if (te.getCode() ==
TransactionExceptionCode.BranchRollbackFailed_Unretriable) {
            return BranchStatus.PhaseTwo_RollbackFailed_Unretryable;
        } else {
            return BranchStatus.PhaseTwo_RollbackFailed_Retryable;
        }
    }
    return BranchStatus.PhaseTwo_Rollbacked;

}
```

交由UndoLogManager执行undo操作,undo中主要分为以下几步： 1、首先去数据库查询，是否有undolog，sql语句为：

```
private static String SELECT_UNDO_LOG_SQL = "SELECT * FROM " +
UNDO_LOG_TABLE_NAME WHERE branch_id = ? AND xid = ? FOR UPDATE";
```

2、数据库中有如有undolog，那么查出来构造BranchUndoLog对象，一条log对应一个对象； 3、对于每一个BranchUndoLog对象，内部可能包含多个SQLUndoLog,遍历BranchUndoLog中的SQLUndoLog List,分别进行处理；

```
for (SQLUndoLog sqlUndoLog : branchUndoLog.getSqlUndoLogs()) {
    TableMeta tableMeta = TableMetaCache.getTableMeta(dataSourceProxy,
sqlUndoLog.getTableName());
    sqlUndoLog.setTableMeta(tableMeta);
    AbstractUndoExecutor undoExecutor = UndoExecutorFactory.getUndoExecutor(
            dataSourceProxy.getDbType(),
            sqlUndoLog);
    undoExecutor.executeOn(conn);
}
```

处理过程就是，根实际操作反着来，比如我们插入一条数据，那么就删除这条数据；如果修改了一条数据，那么就给他修改回去。相应的undoExecutor分别为：MySQLUndoDeleteExecutor，MySQLUndoInsertExecutor，MySQLUndoUpdateExecutor。因为我们第一阶段执行本地操作时，保存了beforeImage和afterImage，所以反着构建SQl时，可以拿到原来的数据。

@GlobalTransactional源码分析

seata中的三个重要部分：

1. TC：事务协调器，维护全局事务和分支事务的状态，驱动全局提交或回滚，就是seata的服务端。
2. TM：事务管理器，开始全局事务，提交或回滚全局事务。
3. RM：资源管理器，管理正在处理的分支事务的资源，向TC注册并报告分支事务的状态，并驱动分支事务的提交或回滚。

自动配置类SeataAutoConfiguration

seata的自动配置类命名非常的直接，就叫做：SeataAutoConfiguration，我们打开这个类

```
@ComponentScan(basePackages = "io.seata.spring.boot.autoconfigure.properties")
@ConditionalOnProperty(prefix = StarterConstants.SEATA_PREFIX, name = "enabled",
havingValue = "true", matchIfMissing = true)
@Configuration
@EnableConfigurationProperties({SeataProperties.class})
public class SeataAutoConfiguration {

}
```

首先，@Configuration表明，SeataAutoConfiguration被定义为了spring的配置类。

@ConditionalOnProperty将配置类生效条件设置为seata.enabled=true，默认值是true，所以可以开关分布式事务功能。

@EnableConfigurationProperties将配置包转成了一个SeataProperties的Bean对象来使用。

@ComponentScan扫描了一下properties包，加载了一大堆类似SeataProperties的Bean对象。

接下来阅读SeataAutoConfiguration的内部代码

```
@Autowired
private SeataProperties seataProperties;

@Bean
public SpringUtils springUtils() {
    return new SpringUtils();
}

@Bean
@DependsOn({"springUtils"})
@ConditionalOnMissingBean(GlobalTransactionScanner.class)
public GlobalTransactionScanner globalTransactionScanner() {return new
GlobalTransactionScanner(seataProperties.getApplicationId(),
seataProperties.getTxServiceGroup());
}
```

SpringUtils是一个实现了ApplicationContextAware的工具包，可以便捷地从容器当中getBean操作。

自动配置的核心点落在了下面的一个Bean，GlobalTransactionScanner。

我们看到构造这个Bean非常的简单，构造方法只需要一个applicationId和txServiceGroup。

applicationId: 就是spring.application.name=你定义的当前应用的名字，例如：userService

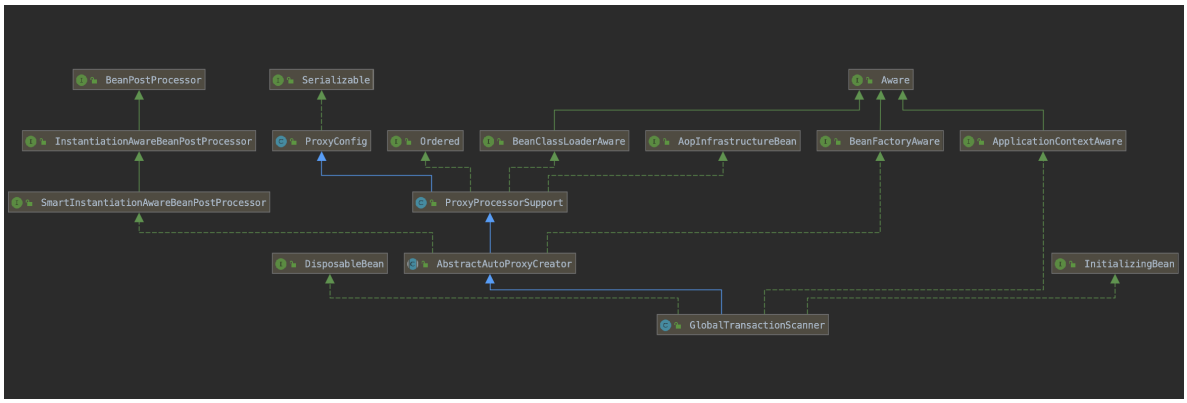
txServiceGroup: 就是以applicationId 加上 -seata-service-group命名的，例如：userService-seata-service-group。如果版本较低的话，那时候可能还不叫seata而是fescar，因此默认命名就是以fescar为后缀。

new了一个GlobalTransactionScanner对象，SeataAutoConfiguration这个自动配置类的作用就结束了。有点草率？是的，不过不影响。毕竟SeataAutoConfiguration只是做了一个启动引导的作用。

GlobalTransactionScanner主体逻辑

既然核心点落在GlobalTransactionScanner这个类，我们继续关注它。看这个名字其实就可以猜测到一点它的作用，扫描@GlobalTransactional这个注解，并对代理方法进行拦截增强事务的功能。

要了解这个类，不得不先阅读一下它的UML图



可以看到，GlobalTransactionScanner主要有4个点值得关注：

1）Disposable接口，表达了spring容器销毁的时候会进行一些操作

2）InitializingBean接口，表达了初始化的时候会进行一些操作

3）AbstractAutoProxyCreator表示它会对spring容器中的Bean进行切面增强，也就是我们上面的拦截事务增强的猜测。

4）ApplicationContextAware表示可以拿到spring容器

这里我们稍微关注一下这4个的执行顺序：

ApplicationContextAware -> InitializingBean -> AbstractAutoProxyCreator -> DisposableBean

我们重点关注一下InitializingBean和AbstractAutoProxyCreator的内容

InitializingBean

```
@Override
public void afterPropertiesSet() {
    if (disableGlobalTransaction) {
        return;
    }
    initClient();
}
```

初始化Seata的Client端的东西，Client端主要包括TransactionManager和ResourceManager。或许是为了简化吧，并没有把initClient这件事从GlobalTransactionScanner里面独立出来一个类。

跟进initClient方法

```
private void initClient() {
    if (StringUtils.isNullOrEmpty(applicationId) ||
StringUtils.isNullOrEmpty(txServiceGroup)) {
        throw new IllegalArgumentException(
            "applicationId: " + applicationId + ", txServiceGroup: " +
txServiceGroup);
    }

    //init TM
    TMClient.init(applicationId, txServiceGroup);

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

    registerSpringShutdownHook();
}
```

initClient逻辑并不复杂，单纯调用TMClient.init初始化TransactionManager的RPC客户端，RMClient.init初始化ResourceManager的RPC客户端。seata的RPC采用netty来实现，seata封装简化了一下使用。

TMClient比较简单，当初初始化RPC组件

```
public static void init(String applicationId, String transactionServiceGroup) {
    TmRpcClient tmRpcClient = TmRpcClient.getInstance(applicationId,
transactionServiceGroup);
    tmRpcClient.init();
}
```

我们关注一下RMClient的init方法

```
public static void init(String applicationId, String transactionServiceGroup) {
    // 获取单例对象
    RmRpcClient rmRpcClient = RmRpcClient.getInstance(applicationId,
transactionServiceGroup);
    // 设置ResourceManager的单例对象
    rmRpcClient.setResourceManager(DefaultResourceManager.get());
    // 添加监听器，监听Server端的消息推送
    rmRpcClient.setClientMessageListener(new
RmMessageListener(DefaultRMHandler.get()));
    // 初始化RPC
    rmRpcClient.init();
}
```

和TMClient想比，RMClient多出了一个监听Server端消息并处理的机制。也就是说TM的职责更多的是主动与Server端通信，比如：全局事务的begin、commit、rollback等。

而RM除了主动操作本地资源外，还会因为全局事务的commit、rollback等的消息推送，从而对本地资源进行相关操作。

AbstractAutoProxyCreator

GlobalTransactionScanner初始化完了TM和RM以后，我们再关注一下AbstractAutoProxyCreator，自动代理。

自动代理，它代理啥东西呢？或者说它给spring中的Bean增强了什么功能？

GlobalTransactionScanner主要扩展了AbstractAutoProxyCreator的两个方法

1）wrapIfNecessary：代理增强的前置判断处理，表示是否该Bean需要增强，如果增强的话创建代理类

2）postProcessAfterInitialization：当一个spring的Bean已经初始化完毕的时候，后置处理一些东西

wrapIfNecessary前置处理

```
@Override
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey)
{
    try {
        synchronized (PROXYED_SET) {
            // 相同Bean排重
            if (PROXYED_SET.contains(beanName)) {
                return bean;
            }

            interceptor = null;
            // 判断是否开启TCC模式
            if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName,
applicationContext)) {
                // TCC实现的拦截器
                interceptor = new
TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
            } else {
                Class<?> serviceInterface =
SpringProxyUtils.findTargetClass(bean);
                Class<?>[] interfacesIfJdk =
SpringProxyUtils.findInterfaces(bean);

                // 判断是否存在@GlobalTransactional或者@GlobalLock注解
                if (!existsAnnotation(new Class[]{serviceInterface})
                    && !existsAnnotation(interfacesIfJdk)) {
                    return bean;
                }

                if (interceptor == null) {
                    // 非TCC的拦截器
                    interceptor = new
GlobalTransactionalInterceptor(failureHandlerHook);

 ConfigurationFactory.getInstance().addConfigListener(ConfigurationKeys.DISABLE_
GLOBAL_TRANSACTION,(ConfigurationChangeListener)interceptor);
                }
            }
            // 判断当前Bean是否已经是spring的代理类了
```

```
            if (!AopUtils.isAopProxy(bean)) {
                // 如果还不是，那么走一轮spring的代理过程即可
                bean = super.wrapIfNecessary(bean, beanName, cacheKey);
            } else {
                // 如果是一个spring的代理类，那么反射获取代理类中已经存在的拦截器集合，然后
添加到该集合当中
                AdvisedSupport advised =
SpringProxyUtils.getAdvisedSupport(bean);
                Advisor[] advisor = buildAdvisors(beanName,
getAdvicesAndAdvisorsForBean(null, null, null));
                for (Advisor avr : advisor) {
                    advised.addAdvisor(0, avr);
                }
            }

            PROXYED_SET.add(beanName);
            return bean;
        }
    } catch (Exception exx) {}
}
```

wrapIfNecessary方法较长我们分步骤看看

1）isTccAutoProxy判断是否开启tcc模式，开启的话选择了TccActionInterceptor拦截器，非tcc模式选择GlobalTransactionalInterceptor拦截器，默认不开启

2）existAnnotation判断当前Bean是否有类或者接口的方法存在@GlobalTransactional或者@GlobalLock注解，如果没有则直接返回

3）isAopProxy方法是判断当前的Bean是否已经是spring的代理类了，无论是JDK动态代理还是Cglib类代理。如果是普通的Bean，走原有的生成代理逻辑即可，如果已经是代理类，那么要通过反射获取代理对象内的拦截器集合也叫做Advisor，直接添加到该集合当中。

wrapIfNecessary的方法并不复杂，但是如果对代理不是很熟悉或许对细节点会有些困惑。

postProcessAfterInitialization数据源代理

wrapIfNecessary创建了代理类，最后看看后置处理又做了啥。跟进该方法

```
@Override
public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
    // 判断是否数据源，且不是数据源代理
    if (bean instanceof DataSource && !(bean instanceof DataSourceProxy) &&
ConfigurationFactory.getInstance().getBoolean(DATASOURCE_AUTOPROXY, false)) {
        // 创建静态代理
        DataSourceProxy dataSourceProxy =
DataSourceProxyHolder.get().putDataSource((DataSource) bean);
        Class<?>[] interfaces = SpringProxyUtils.getAllInterfaces(bean);
        // 创建动态代理类
        return
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
interfaces, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
```

```
                Method m = BeanUtils.findDeclaredMethod(DataSourceProxy.class,
method.getName(), method.getParameterTypes());
                if (null != m) {
                    // 如果静态代理存在目标对象的代理方法，那么调用该代理方法，从而调用目标
方法
                    return m.invoke(dataSourceProxy, args);
                } else {
                    // 如果静态代理不存在目标对象的代理方法，那么直接调用目标对象的代理方法
                    boolean oldAccessible = method.isAccessible();
                    try {
                        method.setAccessible(true);
                        return method.invoke(bean, args);
                    } finally {
                        //recover the original accessible for security reason
                        method.setAccessible(oldAccessible);
                    }
                }
            }
        });
    }
    // 不需要处理数据源代理的，按照原有逻辑处理
    return super.postProcessAfterInitialization(bean, beanName);
}
```

这里有两大块逻辑：

1）是数据源，且需要进行数据源代理的，那么特立独行地走if内的逻辑

2）不需要数据源代理的，那么走原有逻辑

我们关注一下数据源自代理的逻辑，数据源代理或许是seata非常重要的实现之一

首先，DataSource是一个接口，DataSourceProxy对DataSource是一种实现的关系。但是，请注意！！！spring中的数据源的Bean和DataSource是实现关系，可是该Bean和DataSourceProxy并不是继承或者实现的关系，而是组合关系。

因此，DataSourceProxy作为数据源Bean的静态代理而存在，而它实现了DataSource的接口，但是很有可能并未实现Bean的一些接口。

这样一来，就很有必要创建一个动态代理类，来关联一下DataSourceProxy和Bean之间的关系了。如果DataSourceProxy代理过的方法，那么调用代理方法，如果没有就直接调用Bean中的方法。

后置处理的大体逻辑就是这样，基本上就是对数据源进行自动代理处理。

总结

自动配置围绕着GlobalTransactionScanner这个Bean展开。核心逻辑主要是三块：

1) 初始化TransactionManager和ResourceManager的RPC客户端

2) 对@GlobalTransactional和@GlobalLock注解的方法进行增强

3) 数据源代理