1. Overview

Seata API is devided into 2 categories: High-Level API and Low-Level API

- High-Level API : Used for defining and controlling transaction boundary, and querying transaction status.
- Low-Level API : Used for controlling the propagation of transaction context.

2. High-Level API

2.1 GlobalTransaction

GlobalTransaction class contains methods about begin transaction, commit transaction, rollback transaction and get status of transaction and so on.

```
public interface GlobalTransaction {

    /**
     * Begin a global transaction(Use default transaction name and
timeout)
     */
    void begin() throws TransactionException;

    /**
     * Begin a global transaction, and point out the timeout(use default
transaction name)
     */
    void begin(int timeout) throws TransactionException;

    /**
     * Begin a global transaction, and point out the transaction name
and timeout.
     */
    void begin(int timeout, String name) throws TransactionException;

    /**
     * Commit globally
     */
    void commit() throws TransactionException;

    /**
     * Rollback globally
     */
    void rollback() throws TransactionException;

    /**
     * Get the status of transaction
     */
    GlobalStatus getStatus() throws TransactionException;

    /**
```

```
     * Get the XID of transaction
     */
    String getXid();

}
```

2.2 GlobalTransactionContext

GlobalTransaction instance can be retrieved from GlobalTransactionContext:

```

    /**
     * Retrieve current global transaction instance, if it doesn't
exist, create a new one.
     */
    public static GlobalTransaction getCurrentOrCreate() {
        GlobalTransaction tx = getCurrent();
        if (tx == null) {
            return createNew();
        }
        return tx;
    }

    /**
     * Reload the global transaction identified by XID, the instance
aren't allowed to begin transaction.
     * This API is usually used for centralized handling of failed
transaction later.
     * For example, if it's time out to commit globally, the subsequent
centralized processing steps are like this: reload the instance, from
which retrieve the status, then recommit the transaction globally or not
depends on the status value.
     */
    public static GlobalTransaction reload(String xid) throws
TransactionException {
        GlobalTransaction tx = new DefaultGlobalTransaction(xid,
GlobalStatus.UnKnown, GlobalTransactionRole.Launcher) {
            @Override
            public void begin(int timeout, String name) throws
TransactionException {
                throw new IllegalStateException("Never BEGIN on a
RELOADED GlobalTransaction. ");
            }
        };
        return tx;
    }
```

2.3 TransactionalTemplate

TransactionalTemplate: Wrap a business service invoke into a distributed transaction supported service with preceding GlobalTransaction and GlobalTransactionContext API.

```
public class TransactionalTemplate {

    public Object execute(TransactionalExecutor business) throws
TransactionalExecutor.ExecutionException {

        // 1. Get current global transaction instance or create a new
one
        GlobalTransaction tx =
GlobalTransactionContext.getCurrentOrCreate();

        // 2. Begin the global transaction
        try {
            tx.begin(business.timeout(), business.name());

        } catch (TransactionException txe) {
            // 2.1 Fail to begin
            throw new TransactionalExecutor.ExecutionException(tx, txe,
                TransactionalExecutor.Code.BeginFailure);

        }

        Object rs = null;
        try {
            // 3. invoke service
            rs = business.execute();

        } catch (Throwable ex) {

            // Exception from business service invoke
            try {
                // Rollback globally
                tx.rollback();

                // 3.1 Global rollback success, throw original business
exception
                throw new TransactionalExecutor.ExecutionException(tx,
TransactionalExecutor.Code.RollbackDone, ex);

            } catch (TransactionException txe) {
                // 3.2 Global rollback failed
                throw new TransactionalExecutor.ExecutionException(tx,
txe,
                    TransactionalExecutor.Code.RollbackFailure, ex);

            }
        }

        // 4. Commit globally
        try {
            tx.commit();

        } catch (TransactionException txe) {
            // 4.1 Global commit failed
            throw new TransactionalExecutor.ExecutionException(tx, txe,
                TransactionalExecutor.Code.CommitFailure);

        }
        return rs;
    }

}
```

The exception of template method: ExecutionException

```
    class ExecutionException extends Exception {

        // Transaction instance threw exception
        private GlobalTransaction transaction;

        // Exception code:
        // BeginFailure(Fail to begin transaction)
        // CommitFailure(Fail to commit globally)
        // RollbackFailure(Fail to rollback globally)
        // RollbackDone(Global rollback success)
        private Code code;

        // Original exception triggered by rollback
        private Throwable originalException;
```

Outer calling logic try-catch the exception, and do something based on the exception code:

- BeginFailure (Fail to begin transaction): getCause() gets the framework exception of begin transaction, getOriginalException() is null.
- CommitFailure(Fail to commit globally): getCause() gets the framework exception of commit transaction, getOriginalException() is null.
- RollbackFailure (Fail to rollback globally)： getCause() gets the framework exception of rollback transaction， getOriginalException() gets the original exception of business invoke.
- RollbackDone(Global rollback success): getCause() is null, getOriginalException() gets

the original exception of business invoke.

3. Low-Level API

3.1 RootContext

RootContext: It's responsible for maintaining XID during runtime of application.

```
    /**
     * Get the global XID of the current running application
     */
    public static String getXID() {
        return CONTEXT_HOLDER.get(KEY_XID);
    }

    /**
     * Bind the global XID to the current application runtime
     */
    public static void bind(String xid) {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("bind " + xid);
        }
        CONTEXT_HOLDER.put(KEY_XID, xid);
    }

    /**
     * Unbind the global XID from the current application runtime, and
return XID
     */
    public static String unbind() {
        String xid = CONTEXT_HOLDER.remove(KEY_XID);
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("unbind " + xid);
        }
        return xid;
    }

    /**
     * Check if the current application runtime is in the global
transaction context
     */
    public static boolean inGlobalTransaction() {
        return CONTEXT_HOLDER.get(KEY_XID) != null;
    }
```

The implementation of High-Level API is based on maintaining XID in the RootContext.

Whether or not the operation of the current running application is in a global transaction context, just check if there is an XID in the RootContext.

The default implementation of RootContext is based on ThreadLocal, which is the XID is in the context of current thread.

Two classic scenes of Low-Level API :

1. The propagation of transaction context by remote invoke

Retrieve current XID by remote invoke:

```
String xid = RootContext.getXID();
```

Propagating the XID to the provider of service by RPC, bind the XID to current RootContext before executing the business logic of provider.

```
RootContext.bind(rpcXid);
```

2. Pause and recover of transaction

In a global transaction, if some business logic shouldn't be in the scope of the global transaction, unbind XID before invoke it.

```
String unbindXid = RootContext.unbind();
```

Rebind the XID back after the execution of related business logic to achieve recovering the global transaction.

```
RootContext.bind(unbindXid);
```