

## 二

服务器出问题，目前部分恢复

## 09 分布式事务：如何使用强一致性事务与柔性事务？

你好，欢迎进入第 09 课时的学习。今天，我们将介绍一个分布式环境下的重要主题，即分布式事务。在介绍 ShardingSphere 中的具体应用方式之前，我们有必要对分布式事务的基本概念做简要介绍。

### 如何理解分布式事务？

在传统的关系型数据库中，事务是一个标准组件，几乎所有成熟的关系型数据库都提供了对本地事务的原生支持。本地事务提供了 ACID 事务特性。基于本地事务，为了保证数据的一致性，我们先开启一个事务后，才可以执行数据操作，最后提交或回滚就可以了。更进一步，借助于 Spring 等集成化框架，开发人员只需关注引起数据改变的业务即可。

但在分布式环境下，事情就会变得比较复杂。假设系统中存在多个独立的数据库，为了确保数据在这些独立的数据库中保持一致，我们需要把这些数据库纳入同一个事务中。这时本地事务就无能为力了，我们需要使用分布式事务。

业界关于如何实现分布式事务也有一些通用的实现机制，例如支持两阶段提交的 XA 协议以及以 Saga 为代表的柔性事务。针对不同的实现机制，也存在一些供应商和开发工具。因为这些开发工具在使用方式上和实现原理上都有较大的差异性，所以开发人员的一大诉求在于，希望能有一套统一的解决方案能够屏蔽这些差异。同时，我们也希望这种解决方案能够提供友好的系统集成性。

ShardingSphere 作为一款分布式数据库中间件，势必要考虑分布式事务的实现方案。而在设计上，ShardingSphere 从一开始就充分考虑到了开发人员的这些诉求，接下来让我们一起来看看。

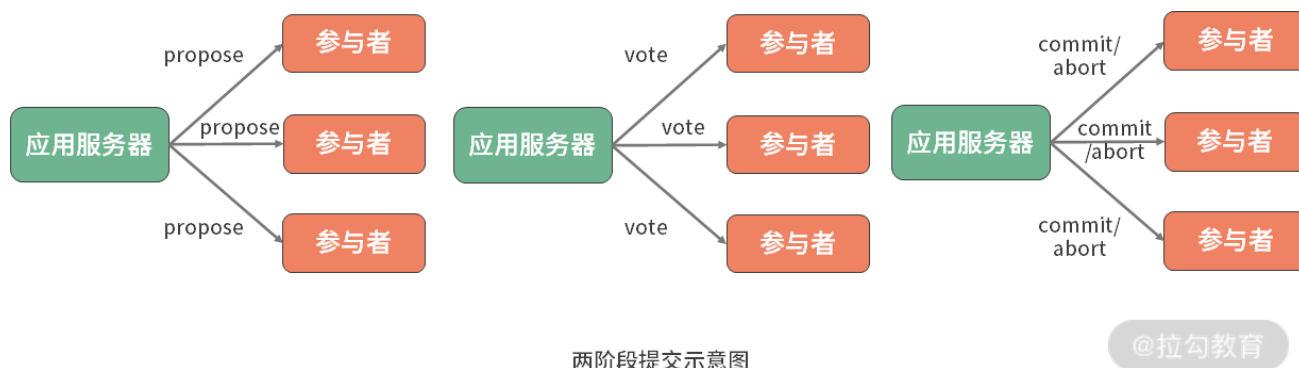
### ShardingSphere 中的分布式事务

在 ShardingSphere 中，除本地事务之外，还提供针对分布式事务的两种实现方案，分别是 XA 事务和柔性事务。这点可以从事务类型枚举值 `TransactionType` 中得到验证：

```
public enum TransactionType {  
    LOCAL, XA, BASE  
}
```

## XA 事务

XA 事务提供基于两阶段提交协议的实现机制。所谓两阶段提交，顾名思义分成两个阶段，一个是准备阶段，一个是执行阶段。在准备阶段中，协调者发起一个提议，分别询问各参与者是否接受。在执行阶段，协调者根据参与者的反馈，提交或终止事务。如果参与者全部同意则提交，只要有一个参与者不同意就终止。



两阶段提交示意图

### 两阶段提交示意图

目前，业界在实现 XA 事务时也存在一些主流工具库，包括 Atomikos、Narayana 和 Bitronix。ShardingSphere 对这三种工具库都进行了集成，并默认使用 Atomikos 来完成两阶段提交。

## BASE 事务

XA 事务是典型的强一致性事务，也就是完全遵循事务的 ACID 设计原则。与 XA 事务这种“刚性”不同，柔性事务则遵循 BASE 设计理论，追求的是最终一致性。这里的 BASE 来自基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventual Consistency）这三个概念。

关于如何实现基于 BASE 原则的柔性事务，业界也存在一些优秀的框架，例如阿里巴巴提供的 Seata。ShardingSphere 内部也集成了对 Seata 的支持。当然，我们也可以根据需要，集成其他分布式事务类开源框架，并基于微内核架构嵌入到 ShardingSphere 运行时环境中。

介绍完理论知识之后，接下来让我们分别使用 XA 事务和 BASE 事务来实现分布式环境下的数据一致性。

## 使用 XA 事务

在 Spring 应用程序中添加对 XA 事务的支持相对简单，无论是 Spring 框架，还是 ShardingSphere 自身，都为我们提供了低成本的开发机制。

### 开发环境准备

要想使用 XA 事务，我们首先要在 pom 文件中添加 sharding-jdbc-core 和 sharding-transaction-xa-core 这两个依赖：

```
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-jdbc-core</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-transaction-xa-core</artifactId>
</dependency>
```

在今天的案例中，我们将演示如何在分库环境下实现分布式事务，因此我们需要在 Spring Boot 中创建一个 .properties 文件，并包含分库需要的所有配置项信息：

```
spring.shardingsphere.datasource.names=ds0,ds1
spring.shardingsphere.datasource.ds0.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds0.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds0.jdbc-url=jdbc:mysql://localhost:3306/ds0
spring.shardingsphere.datasource.ds0.username=root
spring.shardingsphere.datasource.ds0.password=root
spring.shardingsphere.datasource.ds0.autoCommit: false
spring.shardingsphere.datasource.ds1.type=com.zaxxer.hikari.HikariDataSource
spring.shardingsphere.datasource.ds1.driver-class-name=com.mysql.jdbc.Driver
spring.shardingsphere.datasource.ds1.jdbc-url=jdbc:mysql://localhost:3306/ds1
spring.shardingsphere.datasource.ds1.username=root
spring.shardingsphere.datasource.ds1.password=root
spring.shardingsphere.datasource.ds0.autoCommit: false
spring.shardingsphere.sharding.default-database-strategy.inline.sharding-column=
spring.shardingsphere.sharding.default-database-strategy.inline.algorithm-expression=
spring.shardingsphere.sharding.binding-tables=health_record,health_task
spring.shardingsphere.sharding.broadcast-tables=health_level
spring.shardingsphere.sharding.tables.health_record.actual-data-nodes=ds$->{0
spring.shardingsphere.sharding.tables.health_record.key-generator.column=reco
spring.shardingsphere.sharding.tables.health_record.key-generator.type=SNOWFL
spring.shardingsphere.sharding.tables.health_record.key-generator.props.worker
spring.shardingsphere.sharding.tables.health_task.actual-data-nodes=ds$->{0..
spring.shardingsphere.sharding.tables.health_task.key-generator.column=task_i
spring.shardingsphere.sharding.tables.health_task.key-generator.type=SNOWFLAK
spring.shardingsphere.sharding.tables.health_task.key-generator.props.worker.
spring.shardingsphere.props.sql.show=true
```

## 实现 XA 事务

通过分库配置，我们将获取 SQL 执行的目标 DataSource。由于我们使用 Spring 框架而不是使用原生的 JDBC 进行事务管理，所以需要将 DataSource 与 Spring 中的事务管理器 PlatformTransactionManager 关联起来。

另一方面，为了更好地集成 ShardingSphere 中的分布式事务支持，我们可以通过 Spring 框架提供的 JdbcTemplate 模板类来简化 SQL 的执行过程。一种常见的做法是创建一个事务配置类来初始化所需的 PlatformTransactionManager 和 JdbcTemplate 对象：

```

@Configuration
@EnableTransactionManagement
public class TransactionConfiguration {

    @Bean
    public PlatformTransactionManager txManager(final DataSource dataSource)
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean
    public JdbcTemplate jdbcTemplate(final DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}

```

一旦初始化了 JdbcTemplate，就可以在业务代码中注入这个模板类来执行各种 SQL 操作，常见的做法是传入一个 PreparedStatementCallback，并在这个回调中执行各种具体的 SQL：

```

@Autowired
JdbcTemplate jdbcTemplate;
jdbcTemplate.execute(SQL, (PreparedStatementCallback<Object>) preparedStateme
...
return preparedStatement;
});

```

在上面的代码中，我们通过 PreparedStatementCallback 回调获取一个 PreparedStatement 对象。或者，我们可以使用 JdbcTemplate 另一种执行 SQL 的代码风格，通过使用更基础的 ConnectionCallback 回调接口：

```

jdbcTemplate.execute((ConnectionCallback<Object>) connection-> {
    ...
    return connection;
});

```

为了在业务代码中以最少的开发成本嵌入分布式事务机制，ShardingSphere 也专门提供了一个 @ShardingTransactionType 注解来配置所需要执行的事务类型：

```

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface ShardingTransactionType {

    TransactionType value() default TransactionType.LOCAL;
}

```

我们知道，ShardingSphere 提供的事务类型有三种，分别是 LOCAL、XA 和 BASE，默认使用的是 LOCAL。所以如果需要用到分布式事务，需要在业务方法上显式的添加这个注解：

```

@Transactional
@ShardingTransactionType(TransactionType.XA)
public void insert(){
    ...
}

```

另一种设置 TransactionType 的方式是使用 TransactionTypeHolder 工具类。  
TransactionTypeHolder 类中通过 ThreadLocal 来保存 TransactionType：

```

public final class TransactionTypeHolder {

    private static final ThreadLocal<TransactionType> CONTEXT = new ThreadLoc

    @Override
    protected TransactionType initialValue() {
        return TransactionType.LOCAL;
    }
};

public static TransactionType get() {
    return CONTEXT.get();
}

public static void set(final TransactionType transactionType) {
    CONTEXT.set(transactionType);
}

public static void clear() {
    CONTEXT.remove();
}
}

```

可以看到，TransactionTypeHolder 中默认采用的是本地事务，我们可以通过 set 方法来改变初始设置：

```
TransactionTypeHolder.set(TransactionType.XA);
```

现在，使用 XA 开发分布式事务的整体结构的方法已经梳理清楚了，我们可以通过创建一个 insertHealthRecords 方法，在其中添加对 HealthRecord 和 HealthTask 的数据插入代码：

```

private List<Long> insertHealthRecords() throws SQLException {
    List<Long> result = new ArrayList<>(10);
    jdbcTemplate.execute((ConnectionCallback<Object>) connection-> {
        connection.setAutoCommit(false);

        try {
            for (Long i = 1L; i <= 10; i++) {
                HealthRecord healthRecord = createHealthRecord(i);
                insertHealthRecord(healthRecord, connection);

                HealthTask healthTask = createHealthTask(i, healthRecord);
                insertHealthTask(healthTask, connection);
            }
        }
    });
    return result;
}

```

```

        result.add(healthRecord.getRecordId());
    }
    connection.commit();
} catch (final SQLException ex) {
    connection.rollback();
    throw ex;
}

return connection;
});

return result;
}

```

可以看到，在执行插入操作之前，我们关闭了 Connection 的自动提交功能。在 SQL 执行完毕之后，手动通过 Connection commit 方法执行事务提交。一旦在 SQL 的执行过程中出现任何异常时，就调用 Connection 的 rollback 方法回滚事务。

这里有必要介绍执行数据插入的具体实现过程，我们以 insertHealthRecord 方法为例进行展开：

```

private void insertHealthRecord(HealthRecord healthRecord, Connection connection) {
    try (PreparedStatement preparedStatement = connection.prepareStatement(
        preparedStatement.setLong(1, healthRecord.getUserId());
        preparedStatement.setLong(2, healthRecord.getLevelId() % 5 );
        preparedStatement.setString(3, "Remark" + healthRecord.getUserId());
        preparedStatement.executeUpdate();

        try (ResultSet resultSet = preparedStatement.getGeneratedKeys())
            if (resultSet.next()) {
                healthRecord.setRecordId(resultSet.getLong(1));
            }
        }
    }
}

```

首先通过 Connection 对象构建一个 PreparedStatement。请注意，由于我们需要通过 ShardingSphere 的主键自动生成机制，所以在创建 PreparedStatement 时需要特殊地设置：

```

connection.prepareStatement(sql_health_record_insert, Statement.RETURN_GENERATED_KEYS);

```

通过这种方式，在 PreparedStatement 完成 SQL 执行之后，我们就可以获取自动生成的主键值：

```

try (ResultSet resultSet = preparedStatement.getGeneratedKeys()) {
    if (resultSet.next()) {
        healthRecord.setRecordId(resultSet.getLong(1));
    }
}

```



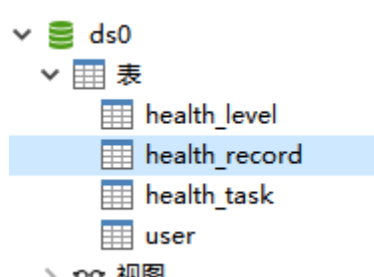
```
}  
}
```

当获取这个主键值之后，就将这个主键值设置回 HealthRecord，这是使用自动生成主键的常见做法。

最后，我们在事务方法的入口处，需要设置 TransactionType：

```
@Override  
public void processWithXA() throws SQLException {  
    TransactionTypeHolder.set(TransactionType.XA);  
  
    insertHealthRecords();  
}
```

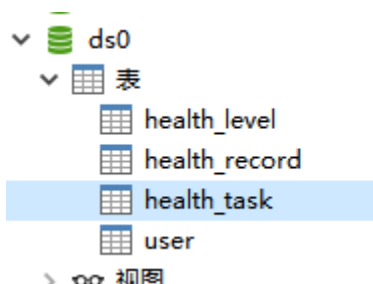
现在让我们执行这个 processWithXA 方法，看看数据是否已经按照分库的配置写入到目标数据库表中。下面是 ds0 中的 health\_record 表和 health\_task 表：



The screenshot shows a database management tool interface. On the left, a tree view displays the database structure for 'ds0', including tables 'health\_level', 'health\_record', 'health\_task', and 'user'. The 'health\_record' table is selected. On the right, the data of the 'health\_record' table is displayed in a table format.

| record_id          | user_id | level_id | remark     |
|--------------------|---------|----------|------------|
| 474308305003614209 |         | 2        | 2 Remark2  |
| 474308305339158529 |         | 4        | 4 Remark4  |
| 474308305561456641 |         | 6        | 1 Remark6  |
| 474308305771171841 |         | 8        | 3 Remark8  |
| 474308305997664257 | 10      |          | 0 Remark10 |

ds0 中的 health\_record 表

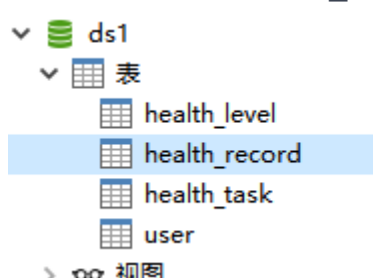


The screenshot shows a database management tool interface. On the left, a tree view displays the database structure for 'ds0', including tables 'health\_level', 'health\_record', 'health\_task', and 'user'. The 'health\_task' table is selected. On the right, the data of the 'health\_task' table is displayed in a table format.

| task_id            | user_id | record_id      | task_name  |
|--------------------|---------|----------------|------------|
| 474308305213329409 |         | 2 305003614209 | TaskName2  |
| 474308305381101569 | 4       | 305339158529   | TaskName4  |
| 474308305607593985 | 6       | 305561456641   | TaskName6  |
| 474308305850863617 | 8       | 305771171841   | TaskName8  |
| 474308306039607297 | 10      | 305997664257   | TaskName10 |

ds0 中的 health\_task 表

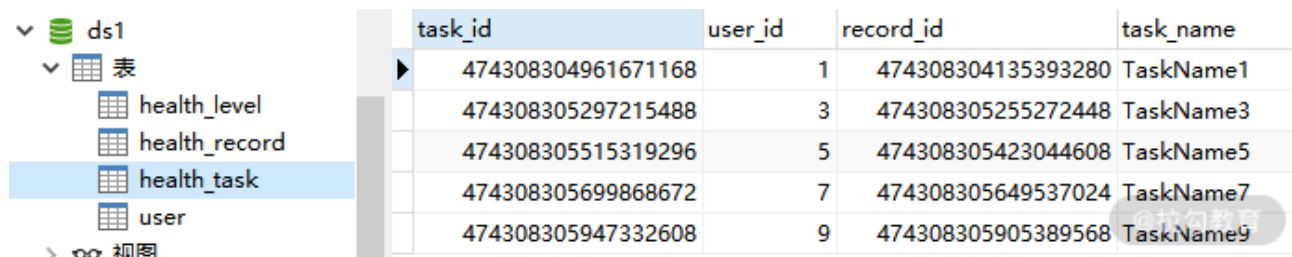
下面则是 ds1 中的 health\_record 表和 health\_task 表：



The screenshot shows a database management tool interface. On the left, a tree view displays the database structure for 'ds1', including tables 'health\_level', 'health\_record', 'health\_task', and 'user'. The 'health\_record' table is selected. On the right, the data of the 'health\_record' table is displayed in a table format.

| record_id          | user_id | level_id | remark    |
|--------------------|---------|----------|-----------|
| 474308304135393280 |         | 1        | 1 Remark1 |
| 474308305255272448 | 3       | 3        | 3 Remark3 |
| 474308305423044608 | 5       | 0        | 0 Remark5 |
| 474308305649537024 | 7       | 2        | 2 Remark7 |
| 474308305905389568 | 9       | 4        | 4 Remark9 |

ds1 中的 health\_record 表



| task_id            | user_id | record_id          | task_name |
|--------------------|---------|--------------------|-----------|
| 474308304961671168 | 1       | 474308304135393280 | TaskName1 |
| 474308305297215488 | 3       | 474308305255272448 | TaskName3 |
| 474308305515319296 | 5       | 474308305423044608 | TaskName5 |
| 474308305699868672 | 7       | 474308305649537024 | TaskName7 |
| 474308305947332608 | 9       | 474308305905389568 | TaskName9 |

ds1 中的 health\_task 表

我们也可以通过控制台日志来跟踪具体的 SQL 执行过程：

```

2020-06-01 20:11:52.043 INFO 10720 --- [main] ShardingSphere-SQL
2020-06-01 20:11:52.043 INFO 10720 --- [main] ShardingSphere-SQL
...
2020-06-01 20:11:52.043 INFO 10720 --- [main] ShardingSphere-SQL
...

```

现在，让我们模拟事务失败的场景，可以在代码执行过程中故意抛出一个异常来做到这一点：

```

try {
    for (Long i = 1L; i <= 10; i++) {
        HealthRecord healthRecord = createHealthRecord(i);
        insertHealthRecord(healthRecord, connection);

        HealthTask healthTask = createHealthTask(i, healthRecord);
        insertHealthTask(healthTask, connection);

        result.add(healthRecord.getRecordId());

        //手工抛出异常
        throw new SQLException("数据库执行异常!");
    }
    connection.commit();
} catch (final SQLException ex) {
    connection.rollback();
    throw ex;
}

```

再次执行 processWithXA 方法，基于 connection 提供的 rollback 方法，我们发现已经执行的部分 SQL 并没有提交到任何一个数据库中。

## 使用 BASE 事务

相较于 XA 事务，在业务代码中集成 BASE 事务的过程就显得相对复杂一点，因为我们需要借助外部框架来做到这一点。这里，我们将基于阿里巴巴提供的 Seata 框架来演示如何使用 BASE 事务。

### 开发环境准备



同样，要想使用基于 Seata 的 BASE 事务，我们首先需要在 pom 文件中添加对 sharding-jdbc-core 和 sharding-transaction-base-seata-at 这两个依赖：

```
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-jdbc-core</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.shardingsphere</groupId>
  <artifactId>sharding-transaction-base-seata-at</artifactId>
</dependency>
```

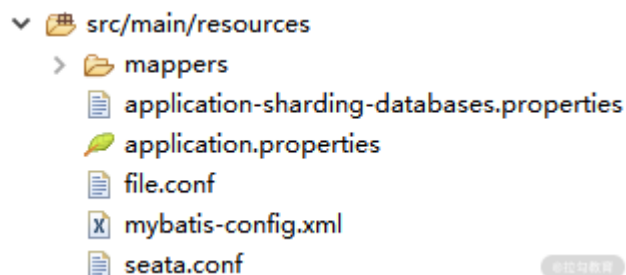
因为用到了 Seata 框架，所以也需要引入 Seate 框架的相关组件：

```
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-rm-datasource</artifactId>
</dependency>
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-tm</artifactId>
</dependency>
<dependency>
  <groupId>io.seata</groupId>
  <artifactId>seata-codec-all</artifactId>
</dependency>
```

然后，我们下载并启动 Seata 服务器，这个过程需要设置 Seata 服务器 config 目录下的 registry.conf，以便指定注册中心，这里使用 ZooKeeper 来充当注册中心。关于如何启动 Seata 服务器的过程可以参考 Seata 的官方文档。请注意，按照 Seata 的运行要求，我们需要在每一个分片数据库实例中创建一张 undo\_log 表。然后，我们还需要在代码工程中 classpath 中增加一个 seata.conf 配置文件：

```
client {
  application.id = health-base
  transaction.service.group = health-base-group
}
```

现在，在 src/main/resources 目录下的文件组织形式应该是这样：



当然，这里我们还是继续沿用前面介绍的分库配置。

## 实现 BASE 事务

基于 ShardingSphere 提供的分布式事务的抽象，我们从 XA 事务转到 BASE 事务唯一要做的事情就是重新设置 TransactionType，也就是修改一行代码：

```
@Override
public void processWithBASE() throws SQLException {
    TransactionTypeHolder.set(TransactionType.BASE);

    insertHealthRecords();
}
```

重新执行测试用例，我们发现在正常提交和异常回滚的场景下，基于 Seata 的分布式事务同样发挥了效果。

## 小结

分布式事务是 ShardingSphere 中提供的一大核心功能，也是分布式环境下数据处理所必须要考虑的话题。ShardingSphere 提供了两种处理分布式事务的实现方式，分别是基于强一致性的 XA 事务，以及基于最终一致性的 BASE 事务。今天，我们结合案例对这两种事务的使用方式做了详细的介绍。

这里给你留一道思考题：当使用 ShardingSphere 时，在业务代码中嵌入分布式事务有哪些开发方式？

本课时的内容就到这里。在下一课时中，我们将介绍 ShardingSphere 中提供了与数据访问安全性相关的一个话题，也就是通过数据脱敏完成对敏感数据的安全访问。

[上一页](#)

[下一页](#)