分布式事务（DTF）

产品文档

腾讯云TCE

文档目录

产品简介
产品概述

最近更新时间: 2025-02-18 16:02:00

什么是分布式事务

分布式事务（Distributed Transaction Framework，下文简称 DTF）是腾讯云金融专区自主研发的高性能、高可用的分布式事务中间件，用于提供分布式的场景中，特别是微服务架构下的事务一致性服务。

分布式事务 DTF 拥抱 Spring Cloud、Spring Boot 开发框架，支持包括 Dubbo 的任意 Java 开发框架，可以与 MySQL、CynosDB、MariaDB 等数据库配合使用，帮助企业用户轻松管理跨数据库、跨服务事务的部署与可视化管理；DTF 还与微服务平台 TSF 打通，配合使用即可实现对应用的全生命周期管理，轻松构建大型分布式系统。

产品功能

DTF 提供对跨数据库事务、跨服务事务、混合型事务的支持，提供 TCC 和 FMT 两种接入模式。DTF 还支持对事务分组运行数据的全方位监控，支持控制台重试异常事务。

跨数据库分布式事务

随着业务的发展，单数据库或数据表难以满足海量存储的要求，而数据库的分库分表又带来了无法保障一致性的问题。分布式事务可保障分库分表后事务的一致性，让您从容处理跨库事务，应对海量数据。

跨服务分布式事务

微服务架构在用户业务不断发展的今天越来越流行，而随着系统功能解耦成多个微服务，一个完整的业务通常会涉及到多个微服务，在一个事务中调用多个微服务对数据一致性带来了挑战。分布式事务同样可解决跨应用、跨服务的数据一致性问题，同时，DTF 支持原生 Spring Cloud 框架，并可与 TSF 打通，助您轻松管理微服务下的分布式事务。

混和（跨数据库&服务）分布式事务

DTF 同样支持同时引入跨数据库、跨服务以及消息队列等资源，保障不同资源间的数据一致性。您可自定义的接入不同类型事务参与者，实现复杂场景下的业务诉求。

TCC 接入模式

TCC（Try - Confirm/Cancel）模式中，用户可以根据自己的业务场景实现 Try、Confirm 和 Cancel 操作，目前 TCC 是现今被广泛应用的一种分布式事务模式。 DTF 针对 TCC 接入模式进行了对协调器和数据库操作的优化，使

其拥有高达传统两阶段提交十倍的性能。TCC 接入模式还拥有极高灵活度，在该模式下您可自定义补偿型、资源预留型、消息队列型事务，同时，TCC 模式支持事务多重嵌套，也支持嵌套 FMT 子事务。 在 TCC 事务中，您可接入包括数据库、微服务、Redis、MQ 等多种资源，灵活处理各种业务场景。

FMT 接入模式

FMT（Framework - managed Transaction）模式下，DTF 通过框架解析您的 SQL 语句，免去了编写 Confirm/Cancel 方法的烦恼，接入使用便捷，对代码无侵入，助您高效完成业务分布式事务的开发。 您可在一条主事务中同时接入 TCC 模式与 FMT 模式的子事务。支持在 TCC 子事务下嵌套 FMT 子事务，支持 FMT 子事务下继续嵌套 FMT 子事务。

全方位可视化监控

DTF 控制台提供了对事务分组的全方位监控能力，包括查看任意时间段主事务、异常事务、总事务的数量，以及对运行事务分组的健康度监控。您可针对监控内容自定义设置告警，在第一时间处理异常事务。

异常事务重试

您可在控制台查看异常事务的具体信息，DTF 支持在控制台对异常事务进行重试。您可一键重试全部异常事务，或自定义选择任意异常事务进行重试。

异步可嵌套事务

使用 DTF，您可在子事务下继续嵌套子事务，实现更长链路的服务调用。DTF 基于异步 TCC 实现，让您在开发分布式事务逻辑时，无需改变原有服务间调用逻辑，减少您的代码改造成本。

产品优势

最近更新时间: 2025-02-18 16:02:00

高性能

对事务协调器进行了大量调优，减少了每个主事务对数据库操作的次数，使得 DTF 拥有约十倍于传统分布式事务的性能。

高可用

每个协调器集群均由三个无状态的虚拟机构成，即使出现节点故障或应用宕机也能保有高性能和数据一致性。

接入简单

通过注解即可接入，提供详细的部署步骤，开发新人也可快速上手。提供不同开发框架部署demo，助您快速熟悉分布式事务开发部署流程。

多维度监控

提供多样化的事务信息监控，随时了解业务运行状况，异常事务处理便捷。用户还可监控底层硬件的健康状况，掌控全局。

支持多种开发框架

完全支持主流 Spring Cloud 开发框架。同时支持 Spring Free，任何 Java 开发框架都可接入。

故障隔离

在控制台、监控出现问题时不影响业务的正常运行；在事务协调器、数据库出现问题时可在控制台监控故障信息。

应用场景

最近更新时间: 2025-02-18 16:02:00

DTF 在涉及到分布式架构的领域（如跨数据库、微服务）中，有着广泛的应用。在金融、政务等涉及较大交易金额的场景下，DTF 所保障的数据一致性是不可或缺的。

典型行业应用场景

金融行业多场景覆盖

- **高频交易** 证券、基金公司的高频交易，对吞吐量要求极高。DTF 性能强劲，可保障高频交易不受性能约束。通过高性能带来的数据高效同步，可助力金融机构减少每笔交易时长，用时间赢得财富。
- **转账** 转账业务往往涉及多数据库与高并发量，高效且正确的转账是金融服务的基础。支付和转账作为分布式事务 Hello World 型场景，在应用 DTF 后，可轻松应对高并发，满足业务需求。
- **账务管理** 金融行业在应对审计和监管时，需保证不同数据库中账务的一致性。运用 DTF 可从容应对复杂业务场景带来的数据不一致问题，降低数据同步的难度与成本。

政务领域支付更便捷

- **生活缴费** 作为支付、转账场景的延伸，生活缴费在政务云中不可或缺。DTF 可保障缴费过程更加安全可靠，关联信息同步修改，跨系统信息及时同步。
- **跨地域信息即刻同步** 人员流动、跨地域信息变更会引入信息不一致的问题，若不同地域的数据存放在不同数据库中，仅通过一般同步手段在信息变动频繁的今天会带来数据脏读脏写问题。采用 DTF 可保障政务信息高效同步。

泛互联网多领域应用

订单、优惠券、积分 以游戏为例，使用钻石（游戏内硬通货）购买游戏金币，会涉及到游戏商城买豆服务、扣减账户钻石数量（数据库）、增加账户金币数（数据库）、VIP 积分增加等服务。目前使用对账的方式来应对此类场景的性能较低，涉及业务扩展或改变时改造成本高。而简单改造分布式事务基础 demo 即可应对该场景的问题，后续开发也更简单。

典型技术应用场景

跨数据库分布式事务

随着业务规模的扩张，业务模块的增多，使用单库单表难以满足业务的需求，这时通过对数据库、数据表进行水平拆分可解决业务数据按模块拆分的问题。但在分库分表后，原先对本地数据库进行的由本地数据库保障的事务操作，将变为对多库多表的分布式事务操作。利用 DTF 可保障分布式场景下的数据一致性。

跨服务分布式事务

在基于 SOA（Service-Oriented Architecture，面向服务架构）的越来越流行的今天，跨服务的一致性问题难以避免。DTF 可与微服务平台结合使用，解决微服务框架下服务间调用的数据一致性。

混和场景分布式事务

例如王者荣耀商城用点券购买皮肤，业务链路上涉及的核心服务有：交易、扣款、发货、账务。用户购买一款皮肤时：

1. 执行皮肤交易服务。
2. 交易服务调用点券数据库扣款，调用发货服务修改用户数据。
3. 扣款后后台账务服务记录账户交易记录，并发送游戏内通知给用户购买记录。

DTF 可保障多个服务及涉及的数据库操作同时成功或失败，第三步交易记录的通知发送可通过消息队列解耦账务服务和通知服务，利用消息队列的特性保证通知的送达。

操作指南

概览

最近更新时间: 2025-02-18 16:02:00

分布式事务的 控制台概览 主要展示事务统计和分组数据等信息。

事务统计

在事务统计中，展示：当前租户下的事务分组总数、当日事务总数、当日主事务数、当日异常事务数。

分组数据

在分组数据模块中，选择一个事务分组后，分组数据内的三个模块（事务统计、事务类型占比、当前 TPS 峰值）都会显示该事务分组的数据。

在事务统计视图中，您可以通过鼠标框选的形式，选定需要查看的时间段，即可方便的查看该时间段的监控数据。同时，事务类型占比和当前 TPS 峰值中的数据也会同步变更。

您还可通过视图右上方的【查看详情】图标，直接查询该事务分组中当前时间段的全部事务。通常，您可通过该视图粗略的查看事务的统计数据，通过鼠标框选进入您关注的时间段，查看更细粒度的统计信息。再通过点击【查看详情】图标，查看当前时间段的具体事务，对异常事务发起重试等。

事务分组

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您通过分布式事务 DTF 控制台，进行分布式事务的创建和管理。

操作步骤

新建事务分组

1. 登录 分布式事务 DTF 控制台，在左侧导航栏单击【事务分组】。
2. 在事务分组页面，单击【新建】，在新建事务分组的弹框中，输入分组名称及备注（选填）。
   - 分组名称：支持中、英文（大小写敏感）、数字、下划线、连接号（-），长度不超过40个字符
   - 备注（选填）：最长200个字符
3. 单击【提交】，完成创建。

获取事务分组基本信息

前提：已新建过事务分组或有存量事务分组。

1. 在 事务分组 页面，单击事务分组 ID。
2. 单击【基本信息】，可获取事务分组 ID，以及事务分组对应的集群 TC 端口号（Broker List）。在配置pom文件时需要用到。

事务的三种检索方式

前提：事务分组中已有运行中的事务。

查看近期事务

默认为您查询该事务分组下近24小时运行的全部主事务。
您可通过修改时间范围来检索自定义时间段内的全部事务。
Tips：通过概览页的监控界面检索自定义时间段事务，更加直观便捷。

按照异常事务搜索

通过按照异常事务搜索，您可以快速找到异常事务，并处理异常事务。
处理异常事务：

异常事务可以分为以下两类：

1. 业务逻辑错误。
2. 某个服务、数据库、Redis、ES、MQ等在Try阶段和Confirm/Cancel阶段的故障状态不一致（如在Try阶段正常，在Confirm/Cancel阶段宕机、故障）。
3. 网络故障。
   在异常事务发生前，DTF框架已进行了三次重试，逻辑为：执行Confirm/Cancel逻辑后，hold 15分钟，若失败，则发起3次重试，每次都hold 15分钟。因此异常事务重试后的状态可能需要一小时刷新。
   DTF不会无限重试异常事务，在以上重试尝试失败后，需要您手动处理异常事务。处理时，您需要逐一排查业务逻辑是否有问题，事务涉及到的服务、数据库等是否出现故障，网络故障是否恢复。排查完成之后再重试异常事务。

   您可直接点击右侧的重试，来重试单条异常事务。
   也可选择多条异常事务，点击上方【重试】来发起小批量重试。
   【重试全部】可对全部异常事务发起重试，但需要花费较多时间，同时会影响重试这段时间内的性能。

按照主事务ID搜索

当您需要精准搜索某一条主事务信息时，可以通过此功能进行搜索。主事务ID 可以通过stdout 来获取。一般情况下，您无需使用此搜索功能。

监控

在 事务分组 页面中，单击监控列的图表，即可查看该组事务的监控信息。

您可自定义需要查询的监控时间窗以及数据显示的时间粒度。

监控指标的解释如下：

- 主事务：在当前时间粒度中主事务的次数。
- 异常事务：在当前时间粒度中异常事务的次数。
- TPS（分钟级均值）：在当前时间粒度下，平均到每秒的事务总数。举例：在1分钟时间内总事务数共150条，这里数据会显示为150 ÷ 60 = 2.5次。
- 成功提交主事务数：在当前时间粒度中成功完成提交的主事务数。
- 回滚主事务数：在当前时间粒度中成功完成回滚的主事务数。
- 事务分组健康度：范围为0% - 100%。事务分组健康度反映了事务分组中异常事务的比例，当健康度不为100%时，建议尽快手动处理异常事务。

告警策略

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您通过云监控控制台，配置分布式事务 DTF 的告警策略。 关于告警策略的详细说明，请参考云监控的告警服务 文档。

操作步骤

1. 在云监控控制台的 告警策略 中，单击【新增】。
2. 在新增策略页面，填写以下信息：
   - 策略名称：1-20个中英文字符或下划线。
   - 备注（选填）：1-20个中英文字符或下划线。
   - 策略类型：选择“分布式事务-分组健康度”。
   - 所属项目：选择已有的项目。
   - 告警对象：选择需要配置告警的事务分组 ID。
   - 触发条件：选中配置触发条件。条件建议配置为当健康度不满100%（即出现异常事务，则发起告警）。您可根据需要自定义配置告警触发条件。
   - 告警渠道：选取合适的接收组/接收人、合适的接收渠道。
   - 高级渠道：启用弹性伸缩策略后，达到告警条件可触发弹性伸缩策略。
   - 接口回调（选填）：填写公网可访问到的 url 作为回调接口地址（域名或IP[:端口][/path]），云监控将及时把告警信息推送到该地址。
3. 单击【完成】。

子用户与协作组使用 DTF

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您通过访问管理控制台，使用主账号给子用户/协作者授权使用分布式事务 DTF。

操作步骤

1. 登录 访问管理控制台 ，进入用户列表。
2. 在用户列表中，找到需要授权的子用户，单击操作列的【授权】。

3. 在关联策略搜索框中，输入“dtf”，按 Enter（或单击搜索图标）后，显示 DTF 预设策略。
4. 根据子用户/协作者需求，勾选全读写权限/只读权限，单击【确认】完成授权。

最佳实践
本地调试

最近更新时间: 2025-02-18 16:02:00

通过本篇文档，您将会掌握进行本地调试分布式事务 DTF（以下简称 DTF）的基本操作。整体流程预计耗时10分钟左右。

操作原理

通过一台跳板机，打通本地与腾讯云金融专区的分布式事务 TC 集群，最终实现在本地 IDE 中启动应用，在 DTF 控制台上查看事务运行状况。

> 注意：该方法主要用于开发测试阶段。在生产环境中，为保证运行的稳定性，建议您通过 CVM/TSF 部署分布式事务应用，以保证网络的稳定性。

准备工作

- 下载 Xshell。本文以 Xshell6版本为例。
- 购买1台有公网 IP 的 CVM，CVM 的规格选择最基础的配置。详细操作请参考 创建实例 文档。

操作步骤

步骤1：打通地址

以 Xshell 为例，打通需要使用的事务分组 TC 集群的3个地址，具体操作如下：

1. 使用 Xshell 登录 CVM。打开 Xshell ，在菜单栏中，选择【文件】>【新建】，新建一个远程会话，配置项说明如下：

- 名称：自定义，无限制
- 协议：SSH
- 主机：准备工作 中的主 IPv4 公网 IP
- 端口号：22

2. 单击【确定】，登录 CVM。
3. 在 DTF 控制台的 事务分组 中，单击事务分组 ID，在基本信息中查看需要打通的地址，即为下图的集群 TC 端口号（Broker List）。以现在云上共享集群为例，显示如下：
4. 在 Xshell【连接】>【SSH】>【隧道】中，添加并填写目标主机与端口（3个）。
5. 在 Xshell 下方的转移规则中，查看状态。如状态为“打开”，则打通成功。

步骤2：修改 IP 地址和端口号

本地启动调试前，需要在工程的 application.yml 中，将事务分组对应的 IP 地址和端口号改为本地 IP（127.0.0.1）以及端口号（15110）。

步骤3：修改数据库地址

本地启动调试前，还需要本地运行 MySQL 数据库，并在 application.yml 中更改数据库 IP 为本地 IP（127.0.0.1）。

步骤4：启动轻量级服务注册中心

如果您需要服务注册发现功能，则需要在本地启动轻量级服务注册中心，详细操作请参考 轻量级服务注册中心 文档。

步骤5：发送请求，触发事务

通过 postman 给入口应用发送请求，此后即可在 DTF 控制台的 事务分组 中查看事务运行情况。

> 说明：端口号在 demo 中默认为19000。

开发手册
准备工作

最近更新时间: 2025-02-18 16:02:00

环境要求

- JDK：建议1.8.161或以上版本
- Maven：建议3.5.2或以上版本
- 网络：能够连接到公网即可

安装 Java

检查 Java 安装

打开终端，执行如下命令：

```
java -version
```

如果输出 Java 版本号，说明 Java 安装成功；如果没有安装 Java，请 下载安装 Java 软件开发套件（JDK）。

设置 Java 环境

设置 JAVA_HOME 环境变量，并指向您机器上的 Java 安装目录。 以 Java 1.6.0_21 版本为例，操作系统的输出如下：

操作系统	输出
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.6.0_21
Linux	export JAVA_HOME=/usr/local/java-current
Mac OSX	export JAVA_HOME=/Library/Java/Home

将 Java 编译器地址添加到系统路径中。

操作系统	输出
Windows	将字符串";C:\Program Files\Java\jdk1.6.0_21\bin"添加到系统变量"Path"的末尾

操作系统	输出
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac OSX	not required

使用上面提到的 **java -version** 命令验证 Java 安装。

安装 Maven

下载安装 Maven

参考 Maven 下载。

设置 MAVEN_HOME 和 PATH 环境变量

- Windows 系统下

```
新建系统变量 MAVEN_HOME 变量值：E:\Maven\apache-maven-3.3.9
编辑系统变量 Path 添加变量值： ;%MAVEN_HOME%\bin
```

- Linux、macOS 系统下

```
export MAVEN_HOME=/usr/local/maven/apache-maven-3.3.9
export PATH=$MAVEN_HOME/bin:$PATH
```

验证 Maven 安装

当 Maven 安装完成后， 通过执行如下命令验证 Maven 是否安装成功。

```
mvn --version
```

若出现正常的版本号信息后，说明 Maven 安装成功。

配置 Maven 仓库

下载 示例工程 后，可以通过以下配置引入Maven 仓库，以便于自动获取 DTF 相关依赖。 在示例工程 中，pom.xml 所在目录执行 mvn clean package 即可下载 DTF SDK。

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://imgcache.finance.cloud.tencent.com:80maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://imgcache.finance.cloud.tencent.com:80www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://imgcache.finance.cloud.tencent.com:80maven.apache.org/SETTINGS/1.0.
0 http://imgcache.finance.cloud.tencent.com:80maven.apache.org/xsd/settings-1.0.0.xsd">
<!-- localRepository
| The path to the local repository maven will use to store artifacts.
|
| Default: ${user.home}/.m2/repository
<localRepository>/path/to/local/repo</localRepository>
-->

<pluginGroups></pluginGroups>
<proxies></proxies>
<servers></servers>
<mirrors></mirrors>

<profiles>
<profile>
<id>qcloud-repo</id>
<repositories>
<repository>
<id>qcloud-central</id>
<name>qcloud mirror central</name>
<url>http://imgcache.finance.cloud.tencent.com:80mirrors.finance.cloud.tencent.com/nexus/reposito
ry/maven-public/</url>
<snapshots>
<enabled>true</enabled>
</snapshots>
<releases>
<enabled>true</enabled>
</releases>
</repository>
</repositories>
<pluginRepositories>
<pluginRepository>
<id>qcloud-plugin-central</id>
<url>http://imgcache.finance.cloud.tencent.com:80mirrors.finance.cloud.tencent.com/nexus/reposito
ry/maven-public/</url>
<snapshots>
<enabled>true</enabled>
</snapshots>
<releases>
<enabled>true</enabled>
</releases>
</pluginRepository>
</pluginRepositories>
```

```
</profile>
</profiles>

<activeProfiles>
<activeProfile>qcloud-repo</activeProfile>
</activeProfiles>
</settings>
```

获取API 密钥

使用 DTF 时，需要用户自行获取 访问密钥 信息（即 SecretId 和 SecretKey）。

获取事务分组 ID、TC 集群 IP 与端口

使用 DTF 时，需要用户自行 创建事务分组，并获取其 GroupId（事务分组 ID）、BorkerList（TC 集群 IP 与端口信息）。

快速部署

最近更新时间: 2025-02-18 16:02:00

准备工作

参考 准备工作 文档。

下载 DTF 示例工程

此处可以下载 DTF 的 示例工程，以便于快速上手分布式事务研发工作。

初始化数据库

1. 准备一台数据库（MySQL 即可），请确保数据库与后续部署程序包的虚拟机在同一 VPC 下。建议使用 云数据库 MySQL，在线初始化数据库，检查数据库与虚拟机是否成功连接。

   > **说明：**
   >
   > 执行 示例工程初始化脚本: 00_init.sql。
   >
   > 如果需要使用 FMT，在每个业务库中，执行 FMT初始化脚本: 01_init_fmt_tables.sql。

2. 获取 MySQLHost、MySQLPort（若使用云数据库MySQL，您可直接使用数据库实例的内网地址）。

工程依赖及应用配置

DTF 可以在**非 Spring**、**Spring Boot**、**Spring Cloud** 三种环境中运行，分别需要使用不同的 Maven POM 配置和应用配置进行启动。

> **说明：**
>
> - 列示例中的 `${dtf.version}` 可以根据 Release Note 选择最新（推荐）或指定的版本。
> - 下列示例中，每个**场景**都可以**单独**部署使用。

非 Spring 应用

示例工程： `single-transfer`

修改示例工程中 com.tencent.cloud.dtf.demo.TransferApplication 类的以下内容。

> **说明：**
>
> 设置的值请在 准备工作 步骤中获取。

```
/**
* 设置启动参数
*/
private static void initEnv() {
// 应用唯一标识，具有相同标识的应用节点被视作对等节点
DtfEnv.setServer("single-transfer");
// API密钥：SecretId
DtfEnv.setSecretId("${SecretId}");
// API密钥：SecretKey
DtfEnv.setSecretKey("${SecretKey}");
// 事务分组ID，事务协调器BrokerList。
DtfEnv.addTxmBroker("${GroupId}", "${BorkerList}");
}
```

修改示例工程中 com.tencent.cloud.dtf.demo.transfer.util.DBUtil 类的以下内容。

> **说明：**
>
> 设置的值请在 初始化数据库（MySQL 即可） 步骤中获取。

```
private static final String URL1 = "jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_account_1?use
SSL=false&characterEncoding=utf8&serverTimezone=GMT";

private static final String URL2 = "jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_account_2?use
SSL=false&characterEncoding=utf8&serverTimezone=GMT";
```

示例工程测试方法：启动后自动执行，不需要外部触发。

Spring Boot应用 - 单应用多数据源场景

示例工程：

TCC模式：

`single-transfer-spring-boot`

FMT模式：

`single-transfer-fmt-spring-boot`

修改示例工程中的application.yml文件。

> **说明：**
>
> 设置的值请在 准备工作、初始化数据库（MySQL 即可） 两个步骤中获取。

```
spring:
application:
name: single-transfer
datasource:
primary:
driver-class-name: com.mysql.cj.jdbc.Driver
jdbcUrl: jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_account_1?useSSL=false&characterEnco
ding=utf8&serverTimezone=GMT
username: dtf_demo_account_1
password: 1q2w3e4r@TX
secondary:
driver-class-name: com.mysql.cj.jdbc.Driver
jdbcUrl: jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_account_2?useSSL=false&characterEnco
ding=utf8&serverTimezone=GMT
username: dtf_demo_account_2
password: 1q2w3e4r@TX
dtf:
env:
groups:
${GroupId}: ${BorkerList}
secretId: ${SecretId}
secretKey: ${SecretKey}
```

示例工程测试方法：启动后自动执行，不需要外部触发。

Spring Boot 应用 - 多应用场景

> **说明：**
>
> 该组示例中 TCC 与 FMT 节点可以混布。

示例工程：

TCC 模式：

`spring-boot-order` ， `spring-boot-inventory` ， `spring-boot-payment` ， `spring-boot-point`

FMT 模式

`spring-boot-fmt-order` ， `spring-boot-fmt-inventory` ， `spring-boot-fmt-payment` ， `spring-boot-fmt-point`

远程调用场景：

```
Order ..... spring-boot-order/spring-boot-fmt-order
├ Inventory ..... spring-boot-inventory/spring-boot-fmt-inventory
└ Payment ..... spring-boot-payment/spring-boot-fmt-payment
└ Point ..... spring-boot-point/spring-boot-fmt-point
```

修改示例工程中的 application.yml 文件的以下部分。

> **说明：**
>
> 设置的值请在 准备工作、初始化数据库（MySQL即可） 两个步骤中获取。

```
spring:
datasource:
url: jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_inventory?useSSL=false&characterEncoding
=utf8&serverTimezone=GMT
dtf:
env:
groups:
${GroupId}: ${BorkerList}
secretId: ${SecretId}
secretKey: ${SecretKey}
```

Spring Boot 应用没有服务注册发现，所以需要手动配置远程调用地址。 修改 spring-boot-order ， spring-boot-fmt-order 工程的以下内容： com.tencent.cloud.dtf.demo.order.proxy.InventoryRestTemplate 类

> **说明：**
>
> ${inventory-host} 为部署 spring-boot-inventory 或 spring-boot-fmt-inventory 的主机地址。

```
public Boolean deduct(Order order) {
Inventory inventory = new Inventory();
inventory.setProductId(order.getProductId());
inventory.setQty(order.getQty());
return restTemplate.postForObject("http://imgcache.finance.cloud.tencent.com:80${inventory-host}:1
9001/deduct", inventory, Boolean.class);
}
```

com.tencent.cloud.dtf.demo.order.proxy.PaymentRestTemplate 类

> **说明：**
>
> ${payment-host} 为部署 spring-boot-payment 或 spring-boot-fmt-payment 的主机地址。

```
public Boolean pay(Order order) {
Payment payment = new Payment();
```

```
payment.setAccountId(order.getAccountId());
payment.setBalance(order.getQty());
return restTemplate.postForObject("http://imgcache.finance.cloud.tencent.com:80${payment-host}:19
002/pay", payment, Boolean.class);
}
```

修改 spring-boot-payment ， spring-boot-fmt-payment 工程的以下内容： com.tencent.cloud.dtf.demo.payment.proxy.PointRestTemplate 类

> **说明：**
>
> ${point-host} 为部署 spring-boot-point 或 spring-boot-fmt-point 的主机地址。

```
public boolean point(Payment payment) {
Point point = new Point();
point.setAccountId(payment.getAccountId());
point.setPoint(payment.getBalance());
return restTemplate.postForObject("http://imgcache.finance.cloud.tencent.com:80${point-host}:1900
3/point", point, Boolean.class);
}
```

示例工程测试方法：

需要手工调用 Order 工程的下单接口：

> **说明：**
>
> ${order-host} 为部署 spring-boot-order 或 spring-boot-fmt-order 的主机地址。

```
curl --location --request POST 'http://imgcache.finance.cloud.tencent.com:80${order-host}:19000/ord
er' \
--header 'Content-Type: application/json' \
-d '{
"productId": 4,
"qty": 1,
"accountId": 1
}'
```

Spring Cloud应用 - 多应用场景

> **说明：**
>
> 该组示例中 TCC 与 FMT 节点可以混布。

示例工程：

TCC 模式： `spring-cloud-order` ， `spring-cloud-inventory` ， `spring-cloud-payment` ， `spring-cloud-point`

FMT 模式： `spring-cloud-fmt-order` ， `spring-cloud-fmt-inventory` ， `spring-cloud-fmt-payment` ， `spring-cloud-fmt-point`

远程调用场景：

```
Order ..... spring-cloud-order/spring-cloud-fmt-order
├ Inventory ..... spring-cloud-inventory/spring-cloud-fmt-inventory
└ Payment ..... spring-cloud-payment/spring-cloud-fmt-payment
└ Point ..... spring-cloud-point/spring-cloud-fmt-point
```

修改示例工程中的 application.yml 文件的以下部分。

> **说明：**
>
> 设置的值请在 准备工作、初始化数据库（MySQL即可） 两个步骤中获取。

```
spring:
datasource:
url: jdbc:mysql://${MySQLHost}:${MySQLPort}/dtf_demo_inventory?useSSL=false&characterEncoding=utf8&serverTimezone=GMT
dtf:
env:
groups:
${GroupId}: ${BorkerList}
secretId: ${SecretId}
secretKey: ${SecretKey}
```

示例工程测试方法：

需要手工调用 Order 工程的下单接口：

> **说明：**
>
> `${order-host}` 为部署 `spring-boot-order` 或 `spring-boot-fmt-order` 的主机地址。

```
curl --location --request POST 'http://imgcache.finance.cloud.tencent.com:80${order-host}:19000/order' \
--header 'Content-Type: application/json' \
-d '{
"productId": 4,
"qty": 1,
"accountId": 1
}'
```

打包示例工程

在示例工程**根目录**下使用以下脚本打包示例工程。

```
mvn clean package
```

> **说明：**
>
> 非 Spring 应用在打包后会出现 single-transfer-x.x.x-RELEASE.jar 与 single-transfer-x.x.x-RELEASE-jar-with-dependenices.jar 两个应用包，请使用后者上传和部署。

部署应用

在 CVM 或 TSF 中部署刚刚打包好的示例工程 jar 包。请确保 CVM 或 TSF 中用于部署的机器，与数据库在同一 VPC 下。

TSF 中部署（推荐）

在 TSF 控制台 部署，可参考 应用基本操作。

CVM 中部署

在 CVM 控制台 中部署，步骤如下：

1. 上传 jar 包到指定服务器（服务器需要安装 JDK 1.8或以上版本）。

2. 使用以下命令启动程序包。

```
nohup java -jar ${jar_file} > root.log &
```

> **说明：**
>
> ${jar_file} 为 打包示例工程 时生成的 jar 包。

检查运行结果

在 DTF控制台 上检查事务分组的对应数据。

在业务日志中检查执行过程产生的日志。

开发详解
TCC 模式 Spring Boot 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 TCC 模式下进行 Spring Boot 开发。 TCC 事务，也可以理解为手动事务。需要用户提供 Try、Confirm、Cancel 接口并进行实现，同时需要保证三个接口的**幂等性**。

准备工作

参考 准备工作 文档。

Maven 配置

通过配置业务代码的 pom.xml 文件，可以引入 DTF 的 SDK 到您的工程中。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>spring-boot-dtf</artifactId>
<version>${dtf.version}</version>
</dependency>
```

> 说明：如果需要同时使用 tsf-sleuth 和 druid，需要切换到 spring-boot-dtf-druid 客户端，配置如下：

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>spring-boot-dtf-druid</artifactId>
</dependency>
```

客户端配置

在客户端中，支持以下配置自定义：

```
dtf:
env:
groups:
${GroupId}: ${BorkerList}
secretId: ${SecretId}
secretKey: ${SecretKey}
server: ${Server}
```

配置项	数据类型	必填	默认值	描述
dtf.env.groups.${GroupId}	String	是	共享集群 TC 列表，如果是独占集群则需要填写	用户的事务分组ID，单客户端使用多个事务分组时可以配置多项。
dtf.env.groups.secretId	String	是	无	用户的腾讯云金融专区SecretID。
dtf.env.groups.secretKey	String	是	无	用户的腾讯云金融专区SecretKey。
dtf.env.groups.server	String	否	${spring.application.name}	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识。
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用。

通常情况下，仅需要在 dtf.env.groups 下配置一个事务分组。例如： 用户A，创建了一个事务分组 `group-x3k9s0ns` ，在 分布式事务控制台 获取该分组的 TC 集群地址为 `127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082` 。该用户访问密钥的 SecretId 为 `SID` ，SecretKey 为 `SKEY` 。需要在业务应用 `app-test` 上使用该事物时，配置样例为：

```
spring:
application:
name: app-test
dtf:
env:
groups:
group-x3k9s0ns: 127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082
secretId: SID
secretKey: SKEY
```

> 说明：此时 `dtf.env.groups.server` 的值为 `app-test` 。

启用分布式事务服务

在 @SpringBootApplication 注解处增加 `@EnableDtf` 注解来启用分布式事务服务。

```
@SpringBootApplication
@EnableDtf
@EnableTransactionManagement
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

> 说明：通常建议同时启用本地事务管理 `@EnableTransactionManagement` 。

主事务管理

主事务的生命周期可以分为：开启、提交/回滚。

您可以根据实际业务的需要，选择**通过注解管理主事务**或**通过 API 管理主事务**。

通过注解管理主事务

主事务通常建议在入口 Controller 方法处开启。一般注释在**实现类**方法上，并且该类需要注入为 Bean。

以下面注解了 `@DtfTransactional` 的 order 方法为例：

```
@DtfTransactional
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
// 执行业务逻辑或分支事务
}
```

1. 进入 order 方法前 DTF 框架**开启**主事务。
2. 执行业务逻辑或分支事务。

- 如果该方法正常执行完毕，返回业务数据（或者 void 方法无返回值），DTF 框架**提交**主事务。
- 如果该方法执行出现问题，抛出异常时，DTF 框架**回滚**主事务。

3. DTF 框架自动关闭当前线程**主事务上下文**。

主事务注解支持的能力包括

参数	数据类型	必填	默认值	描述
timeout	Integer	否	60 × 1000	事务超时时间（主事务**开启**到**提交/回滚**的时长），单位：毫秒
groupId	String	否	-	在此事务分组下开启主事务

DTF 目前支持通过 @DtfTransactional 传染主事务。当您的主事务有多个入口时，使用多个@DtfTransactional 不会报错。全局事务的开始与结束，将由第一个开始执行的标有 @DtfTransactional 的主事务纳管。

> 说明：如果 dtf.env.groups 下只配置了**1个**事务分组 ID，则 @DtfTransactional 注解中**不需要**填写 groupId，DTF 框架会自动从配置中获取。

通过 API 管理主事务

如果业务存在异步操作或者有特殊诉求（例如：一个主事务不能在单一方法闭环），也可以使用 API 来进行主事务管理。

还是以上面的 order 方法为例，此时需要等待一个 orderCallback 回调来确认提交或回滚主事务：

```
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
try {
Boolean result;
// 开启主事务
DtfTransaction.begin(DTF.DEFAULT_TX_TIMEOUT);
// 执行业务逻辑或分支事务 > result
return result;
} catch(Throwable t) {
// 回滚主事务
DtfTransaction.rollback();
} finally {
// 关闭当前线程主事务上下文
DtfTransaction.end();
}
}

@RequestMapping("/order/callback")
public Boolean orderCallback(@RequestBody OrderCallback orderCallback) {
try {
// 绑定 DTF 上下文
// 如果全局使用 DTF 框架，可以忽略该步骤，框架会自动完成上下文传递。详见[远程请求时传递分布式事务上下文]章节
DtfTransaction.bind(orderCallback.getGroupId(), orderCallback.getTxId(), orderCallback.getLastBranchId());
// 处理业务回调逻辑
```

```
if(orderCallback.getResult()) {
// 回调成功时，提交主事务
DtfTransaction.commit();
} else {
// 回调失败时，回滚主事务
DtfTransaction.rollback();
}
return orderCallback.getResult();
} catch(Throwable t) {
// 回滚主事务
DtfTransaction.rollback();
} finally {
// 关闭当前线程主事务上下文
DtfTransaction.end();
}
}
```

分支事务管理

分支事务的生命周期可以分为：开启、提交/回滚。

可以根据实际业务的需要选择**通过注解管理分支事务**或**通过 API 管理分支事务**。

一个 TCC 分支事务中，需要包含 Try、Confirm、Cancel 三个部分。

- 分支事务的 Try、Confirm、Cancel 方法所在的类需要被**注入为 Bean**。
- 分支事务的 Try、Confirm、Cancel 方法建议使用本地事务管理（例如注解 Spring 的@Transactional）。
- 分支事务的 Try、Confirm、Cancel 方法的参数**保持一致**。
- 分支事务的 Try、Confirm、Cancel 方法的前两个参数固定为 `Long txId` 和 `Long branchId` 。

Try 方法：

- 本地调用 Try 方法时 `txId` 和 `branchId` 参数传 `null` ，其他参数正常传递。
- 返回值为**业务逻辑**需要的返回值。

Confirm 方法：

- 返回值固定为 Boolean 类型。
- 仅在返回 **true** 时视为分支事务 **Confirm 成功**。
- 返回 **false** 或**抛出异常**时，视为分支事务 **Confirm 失败**。

Cancel 方法：

- 返回值固定为 Boolean 类型。

- 仅在返回 **true** 时视为分支事务 **Cancel 成功**。
- 返回 **false** 或**抛出异常**时，视为分支事务 **Cancel 失败**。

通过注解管理分支事务

分支事务通常建议注解在业务的 Service上。可以注解在**接口**或**实现类**上，并且该类需要注入为 Bean。

以下面注解了 `@DtfTcc` 的 order 方法为例：

```
public interface IOrderService {
@DtfTcc
public boolean order(Long txId, Long branchId, Order order);

public boolean confirmOrder(Long txId, Long branchId, Order order);

public boolean cancelOrder(Long txId, Long branchId, Order order);
}
```

分支事务注解支持的参数包括：

参数	数据类型	必填	默认值	描述
name	String	否	@DtfTcc 方法名 + 方法签名 Hash	分支事务名称，请在同一事务分组
confirmClass	String	否	@DtfTcc 注解所在 Class	Confirm 操作类名
confirmMethod	String	否	confirm 前缀 + @DtfTcc 注解方法名首字母大写	Confirm 操作方法名
cancelClass	String	否	@DtfTcc 注解所在 Class	Cancel 操作类名
cancelMethod	String	否	Cancel 前缀 + @DtfTcc 注解方法名首字母大写	Cancel 操作方法名
rollbackFor	Class[]	否	{}	分支事务在识别到以下异常时回滚主事务，未配置时不回滚

在上面的例子中：

- `try` ：IOrderService.order(Long txId, Long branchId, Order order)
- `confirmClass` ：IOrderService
- `confirmMethod` ：confirmOrder(Long txId, Long branchId, Order order)
- `cancelClass` ：IOrderService
- `cancelMethod` ：cancelOrder(Long txId, Long branchId, Order order)

- rollbackFor：默认为空。若想要在发生异常时回滚，可设置为 Exception。

通过 API 管理分支事务（不推荐）

可以参考 Spring Free 开发指导 中的分支事务管理章节。

远程请求时传递分布式事务上下文

使用 RestTemplate 或 FeginClient 时，DTF 框架支持自动化的分布式事务上下文传递。

如果使用了其他的通信框架，也可以**手动处理分布式事务上下文**。

主调 - RestTemplate

使用 RestTemplate 访问下游服务时，DTF 框架自动注入了 TxRestTemplateInterceptor，向请求头中装载分布式事务上下文信息。

DTF 框架注入的请求头信息为：

```
# 事务分组 ID
DTF-Group-ID: ${GroupId}
# 主事务 ID
DTF-Tx-ID: ${TxId}
# 父级分支事务 ID
DTF-Last-Branch-ID: ${LastBranchId}
```

主调 - FeginClient

使用 FeginClient 访问下游服务时，DTF 框架自动注入了 TxFeignInterceptor，向请求头中装载分布式事务上下文信息。

需要引入 feign 依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

DTF 框架注入的请求头信息为：

```
# 事务分组 ID
DTF-Group-ID: ${GroupId}
# 主事务 ID
DTF-Tx-ID: ${TxId}
```

```
# 父级分支事务 ID
DTF-Last-Branch-ID: ${LastBranchId}
```

主调 - 手动处理

可以参考 Spring Free开发指导 中的**远程请求时传递分布式事务上下文**章节。

被调 - Spring MVC - Controller

使用 Spring MVC 的应用，在进入 Controller 前，DTF 框架会自行从请求头中检索下列 Header key。

```
# Header key的常量ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key的常量ClientConstant.HTTP_HEADER.TX_ID
DTF-Tx-ID: ${TxId}
# Header key的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

检索后通过 TxContextRestore 切点还原分布式事务上下文。

被调 - 手动处理

可以参考 Spring Free 开发指导 中的**远程请求时传递分布式事务上下文**章节。

与 TSF 结合使用

DTF 框架完全兼容 TSF 应用，请按照下面的指引使用。

> 说明：目前仅支持 Greenwich 版本的 TSF SDK。

Maven POM

```
<!-- TSF 启动器 -->
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>spring-cloud-tsf-starter</artifactId>
</dependency>
```

启用 TSF

```
@SpringBootApplication
@EnableDtf
@EnableTsf
@EnableTransactionManagement
```

```
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

TCC 模式 Spring Free 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 TCC 模式下进行 Spring Free 开发。

TCC 事务，也可以理解为手动事务。需要用户提供 Try、Confirm、Cancel 接口并进行实现，同时需要保证三个接口的**幂等性**。

准备工作

参考 准备工作 文档。

开发步骤

Maven 配置

通过配置业务代码的 pom.xml 文件，可以引入 DTF 的 SDK 到您的工程中。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>dtf-core</artifactId>
<version>${dtf.version}</version>
</dependency>
```

客户端配置

在客户端中，使用以下 API 进行 DTF 的配置设置。

```
DtfEnv.setServer(String server);
DtfEnv.setSecretId(String secretId);
DtfEnv.setSecretKey(String secretKey);
DtfEnv.addTxmBroker(String groupId, String txmBrokerList);
```

配置项说明：

配置项	数据类型	必填	默认值	描述

配置项	数据类型	必填	默认值	描述
groupId	String	是	无	用户的事务分组 ID。
txmBrokerList	String	是	无	TC 集群节点列表。
secretId	String	是	无	用户的腾讯云金融专区 SecretID。
secretKey	String	是	无	用户的腾讯云金融专区 SecretKey。
server	String	是	无	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识。
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用。

通常情况下，仅需要在使用 DtfEnv.addTxmBroker() 配置一个事务分组。例如： 用户 A，创建了一个事务分组 group-x3k9s0ns ，在 分布式事务控制台 获取该分组的 TC 集群地址为 127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082 。该用户访问密钥的 SecretId 为 SID ，SecretKey 为 SKEY 。需要在业务应用 app-test 上使用该事物时，配置样例为：

```
DtfEnv.setServer("app-test");
DtfEnv.setSecretId("SID");
DtfEnv.setSecretKey("SKEY");
DtfEnv.addTxmBroker("group-x3k9s0ns", "127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082");
```

注册 TCC

对应用内的 TCC 信息进行手动注册。

```
// 获取 TCC 实例
TCC.getInstance(Object confirmClassInstance, String confirmMethodName, Object cancelClassInstanc
e,
String cancelMethodName);
// 或
TCC.getInstance(Object confirmClassInstance, Method confirmMethod, Object cancelClassInstance,
Method cancelMethod);
// 注册 TCC 信息
TccRegistry.register(String tccName, TCC tccInstance);
```

> 注意：TCC 的名称（tccName）需要保持稳定，并且在一个应用中不可重复。

例如：

```
/**
* 注册 TCC 信息
*/
private static void registerTcc() {
ITransferService transferService = new TransferService();
// debit 的 Confirm 和 Cancel
TccRegistry.register("debit",
TCC.getInstance(transferService, "confirmDebit", transferService, "cancelDebit"));
// credit 的 Confirm 和 Cancel
TccRegistry.register("credit",
TCC.getInstance(transferService, "confirmCredit", transferService, "cancelCredit"));
}
```

示例中注册了两个 TCC： `debit` （扣款）和 `credit` （入账）。

`debit` 中参数如下：

参数	说明
`confirmClassInstance`	TransferService 的实例 transferService
`confirmMethodName`	TransferService中 名称为 confirmDebit 的方法
`cancelClassInstance`	TransferService 的实例 transferService
`cancelMethodName`	TransferService 中名称为 cancelDebit

`credit` 中参数如下：

参数	说明
`confirmClassInstance`	TransferService 的实例 transferService
`confirmMethodName`	TransferService 中名称为 confirmCredit 的方法
`cancelClassInstance`	TransferService 的实例 transferService
`cancelMethodName`	TransferService 中名称为 cancelCredit 的方法

启用分布式事务服务

在用户应用处理逻辑完成，并且完成**客户端配置**和**注册 TCC**步骤后，可以使用 API 来开启分布式事务服务。

```
// 启动分布式事务客户端
DtfClient.start();
```

主事务管理

主事务的生命周期可以分为：开启、提交/回滚。

开启主事务

在**客户端配置**步骤中只添加了一个事务分组时，可以使用**开启主事务：使用默认事务分组。**添加了多个事务分组时，必须使用**开启主事务：使用指定事务分组。**

```
// 开启主事务：使用默认事务分组
DtfTransaction.begin(Integer timeout);
// 开启主事务：使用指定事务分组
DtfTransaction.begin(Integer timeout, String groupId)
```

提交/回滚主事务

```
// 提交主事务
DtfTransaction.commit();
// 回滚主事务
DtfTransaction.rollback();
```

关闭主事务上下文

> 注意：该操作仅关闭**当前线程**的主事务上下文，不会对主事务状态产生影响。

```
DtfTransaction.end();
```

使用示例：

```
public boolean execute() {
// 业务检查
Long txId = DtfTransaction.begin(10000);
try {
// 执行各个分支事务或业务逻辑
// 提交主事务
DtfTransaction.commit();
return true;
} catch (Throwable e) {
// 回滚主事务
DtfTransaction.rollback();
LOG.error("Bank transfer failed.", e);
return false;
} finally {
// 关闭主事务上下文
DtfTransaction.end();
```

```
}
}
```

分支事务管理

分支事务的生命周期可以分为：开启、提交/回滚。

一个 TCC 分支事务中，需要包含 Try、Confirm、Cancel 三个部分。

- 分支事务的 Try、Confirm、Cancel 方法的参数**保持一致**。
- 分支事务的 Try、Confirm、Cancel 方法的前两个参数固定为 `Long txId` 和 `Long branchId` 。

Try 方法：

- 本地调用 Try 方法时 `txId` 和 `branchId` 参数传 `null` ，其他参数正常传递。
- 返回值为 `业务逻辑` 需要的返回值。

Confirm 方法：

- 返回值固定为 Boolean 类型。
- 仅在返回 true 时视为分支事务 Confirm 成功。
- 返回 false 或抛出异常时，视为分支事务 Confirm 失败。

Cancel 方法：

- 返回值固定为 Boolean 类型。
- 仅在返回 true 时视为分支事务 Cancel 成功。
- 返回 false 或抛出异常时，视为分支事务 Cancel 失败。

开启分支事务

```
DtfTccBranch.begin(String name, Object[] params);
```

参数	说明
`name`	TCC 名称
`params`	Try 方法的业务参数，前两个参数（即 `Long txId` 和 `Long branchId` ）填null

检查主事务状态是否为 Trying

仅在 Trying 状态时允许提交分支事务，该接口主要用于防止分支事务 Try 阶段延迟提交本地事务。

```
DtfTccBranch.checkTxTrying();
```

结束分支事务

> 注意：该操作仅关闭**当前线程**的分支事务上下文，不会对分支事务状态产生影响。

```
DtfTccBranch.end();
```

示例：

```
// === 开启主事务 ===

// 开启分支事务1：扣款
Long branchId1 = DtfTccBranch.begin("debit", new Object[] { null, null, this.to, this.amount });
// 执行Try方法1
transferService.debit(txId, branchId1, this.to, this.amount);
// 关闭分支事务1上下文
DtfTccBranch.end();

// 开启分支事务2：入账
Long branchId2 = DtfTccBranch.begin("credit", new Object[] { null, null, this.from, this.amount });
// 执行Try方法2
transferService.credit(txId, branchId2, this.from, this.amount);
// 关闭分支事务2上下文
DtfTccBranch.end();

// === 提交 / 回滚主事务 ===
```

> 说明：在执行 Try 方法的本地事务末尾，需要使用 `DtfTccBranch.checkTxTrying();` 接口防止 Try 阶段延迟提交本地事务。

远程请求时传递分布式事务上下文

主调

需要从上下文中提取 `groupId` ， `txId` ， `lastBranchId` 三个数据传递到下游。

使用以下 API 提取：

```
String groupId = DtfContextHolder.get().getGroupId();
Long txId = DtfContextHolder.get().getTxId();
Long lastBranchId = DtfContextHolder.get().getBranchIdStack().peek();
```

建议放到下列 Header 的 key 中，下游可以通过 DTF SDK 自行注入。

```
# Header key的常量ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key的常量ClientConstant.HTTP_HEADER.TX_ID
```

```
DTF-Tx-ID: ${TxId}
# Header key的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

被调

根据上游业务的特性手动获取 `groupId` ， `txId` ， `lastBranchId` 三个数据。

如果上游使用的是 DTF 封装的 RestTemplate 或 Fegin，请从以下请求头中获取：

```
# Header key的常量ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key的常量ClientConstant.HTTP_HEADER.TX_ID
DTF-Tx-ID: ${TxId}
# Header key的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

如果是其他方式传递，则需要用户按照接口协议自行获取。

获取了三个数据后，通过 API 绑定分布式事务上下文。

```
DtfTransaction.bind(${GroupId}, ${TxId}, ${LastBranchId});
```

FMT 模式 Spring Boot 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 FMT 模式下进行 Spring Boot 开发。

FMT 事务，也可以理解为框架托管事务。需要用户仅需要在 **FMT 规范**下正常实现业务逻辑即可实现分布式事务。相对于 TCC事务，省去了编写 Confirm、Cancel 的代码工作。 FMT 事务的实现原理：代理用户执行 PrepareStatement 和 CreateStatement 操作，对执行的 DML 语句进行解析，记录前后象，并生成UNDO信息。鉴于此，FMT 对用户使用的数据库和 SQL 语句会有一定的要求，详见 FMT规范。

准备工作

- 参考 准备工作 文档，完成环境配置和开发前准备。
- 参考 快速部署 文档，执行 FMT 初始化脚本

Maven 配置

通过配置业务代码的 pom.xml 文件，可以引入 DTF 的 SDK 到您的工程中。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>spring-boot-dtf</artifactId>
<version>${dtf.version}</version>
</dependency>
```

> 说明：如果需要同时使用 tsf-sleuth 和 druid，需要切换到 spring-boot-dtf-druid 客户端，配置如下：

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>spring-boot-dtf-druid</artifactId>
</dependency>
```

客户端配置

在客户端中，支持以下配置自定义：

```
dtf:
env:
groups:
${GroupId}: ${BorkerList}
secretId: ${SecretId}
secretKey: ${SecretKey}
server: ${Server}
```

配置项	数据类型	必填	默认值	描述
dtf.env.groups.${GroupId}	String	是	共享集群 TC 列表，如果是独占集群则需要填写	用户的事务分组 ID，单客户端使用多个事务分组时可以配置多项
dtf.env.groups.secretId	String	是	无	用户的腾讯云金融专区 SecretID
dtf.env.groups.secretKey	String	是	无	用户的腾讯云金融专区 SecretKey
dtf.env.groups.server	String	否	${spring.application.name}	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识。
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用。

通常情况下，仅需要在dtf.env.groups下配置一个事务分组。例如： 用户 A，创建了一个事务分组 `group-x3k9s0ns` ，在分布式事务控制台获取该分组的 TC 集群地址为 `127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082` 。该用户访问密钥的 SecretId 为 `SID` ，SecretKey为 `SKEY` 。需要在业务应用 `app-test` 上使用该事物时，配置样例为：

```
spring:
application:
name: app-test
dtf:
env:
groups:
group-x3k9s0ns: 127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082
secretId: SID
secretKey: SKEY
```

> 说明：此时 dtf.env.groups.server 的值为 app-test 。

启用分布式事务服务

在@SpringBootApplication注解处增加 @EnableDtf 注解来启用分布式事务服务。

```
@SpringBootApplication
@EnableDtf
@EnableTransactionManagement
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

> 说明：通常建议同时启用本地事务管理 @EnableTransactionManagement 。

主事务管理

主事务的生命周期可以分为：开启，提交/回滚。

可以根据实际业务的需要选择**通过注解管理主事务**或**通过 API 管理主事务**。

通过注解管理主事务

主事务通常建议在入口 Controller 方法处开启。一般注释在**实现类**方法上，并且该类需要注入为 Bean。

以下面注解了 @DtfTransactional 的 order 方法为例：

```
@DtfTransactional
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
// 执行业务逻辑或分支事务
}
```

1. 进入 order 方法前 DTF 框架开启主事务。

2. 执行业务逻辑或分支事务。
   - 如果该方法正常执行完毕，返回业务数据（或者 void 方法无返回值），DTF 框架**提交**主事务。
   - 如果该方法执行出现问题，抛出异常时，DTF框架**回滚**主事务。

3. DTF 框架自动关闭当前线程**主事务上下文**。

主事务注解支持的能力包括

参数	数据类型	必填	默认值	描述
timeout	Integer	否	60 × 1000	事务超时时间（主事务**开启**到**提交/回滚**的时长），单位：毫秒
groupId	String	否	-	在此事务分组下开启主事务

如果 dtf.env.groups 下只配置了**1个**事务分组 ID，则 @DtfTransactional 注解中**不需要**填写groupId，DTF 框架会自动从配置中获取。

DTF 现在支持通过 @DtfTransactional 传染主事务。当您的主事务有多个入口时，使用多个@DtfTransactional 不会报错。全局事务的开始与结束，将由第一个开始执行的标有 @DtfTransactional 的主事务纳管。

通过 API 管理主事务

如果业务存在异步操作或者有特殊诉求（例如：一个主事务不能在单一方法闭环），也可以使用 API 来进行主事务管理。

还是以上面的 order 方法为例，此时需要等待一个 orderCallback 回调来确认提交或回滚主事务：

```
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
try {
Boolean result;
// 开启主事务
DtfTransaction.begin(DTF.DEFAULT_TX_TIMEOUT);
// 执行业务逻辑或分支事务 > result
return result;
} catch(Throwable t) {
// 回滚主事务
DtfTransaction.rollback();
} finally {
// 关闭当前线程主事务上下文
DtfTransaction.end();
}
}

@RequestMapping("/order/callback")
public Boolean orderCallback(@RequestBody OrderCallback orderCallback) {
try {
// 绑定DTF上下文。
// 如果全局使用 DTF 框架，可以忽略该步骤，框架会自动完成上下文传递。详见[远程请求时传递分布式事务
```

```
上下文]章节
DtfTransaction.bind(orderCallback.getGroupId(), orderCallback.getTxId(), orderCallback.getLastBranch
Id());
// 处理业务回调逻辑
if(orderCallback.getResult()) {
// 回调成功时，提交主事务
DtfTransaction.commit();
} else {
// 回调失败时，回滚主事务
DtfTransaction.rollback();
}
return orderCallback.getResult();
} catch(Throwable t) {
// 回滚主事务
DtfTransaction.rollback();
} finally {
// 关闭当前线程主事务上下文
DtfTransaction.end();
}
}
```

分支事务管理

FMT 的分支事务管理是自动化的，并且与 TCC 分支事务略有不同。

FMT 分支事务的注解并不会直接触发分支事务生成，而是将注解方法的执行上下文切换为FMT模式。该方法内的所有本地调用所执行的**每1行 DML 语句**都会自行生成**1个分支事务**。

通过注解管理分支事务

分支事务通常建议注解在业务的 Service 上。可以注解在**接口**或**实现类**上，并且该类需要注入为 Bean。

以下面注解了 `@DtfFmt` 的 order 方法为例：

```
public interface IOrderService {
@DtfFmt
public boolean order(Order order);
}
```

该方法的实现为：

```
// public boolean order(Order order);实现
@Transactional
@Override
public boolean order(Order order) {
```

```
orderDao.createOrder(order);
return true;
}

// orderDao.createOrder(order);实现
public int createOrder(Order order) {
String sql = "INSERT INTO dtf_demo_order (product_id, qty, account_id) VALUES (?, ?, ?)";
try {
GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(new PreparedStatementCreator() {
@Override
public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
PreparedStatement ps = con.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
ps.setInt(1, order.getProductId());
ps.setInt(2, order.getQty());
ps.setInt(3, order.getAccountId());
return ps;
}
}, keyHolder);
return keyHolder.getKey().intValue();
} catch (Exception e) {
LOG.error("Create order failed.", e);
throw new RuntimeException("Create order failed.");
}
}
```

在 `orderDao.createOrder(order);` 方法中执行了一句Insert语句，此时框架会注册一个分支事务对这个Insert进行全局的事务管理。

分支事务注解支持的参数包括：

参数	数据类型	必填	默认值	描述
rollbackFor	Class<? extends Throwable>[]	否	{}	分支事务在识别到以下异常时回滚主事务，未配置时不回滚

rollbackFor：默认为空。若想要在发生异常时回滚，可设置为 Exception。

远程请求时传递分布式事务上下文

使用 `RestTemplate` 或 `FeginClient` 时，DTF框架支持自动化的分布式事务上下文传递。

如果使用了其他的通信框架，也可以**手动处理分布式事务上下文。**

主调 - RestTemplate

使用 RestTemplate 访问下游服务时，DTF 框架自动注入了 TxRestTemplateInterceptor，向请求头中装载分布式事务上下文信息。

DTF 框架注入的请求头信息为：

```
# 事务分组 ID
DTF-Group-ID: ${GroupId}
# 主事务 ID
DTF-Tx-ID: ${TxId}
# 父级分支事务 ID
DTF-Last-Branch-ID: ${LastBranchId}
```

主调 - FeginClient

使用 FeginClient 访问下游服务时，DTF 框架自动注入了 TxFeignInterceptor，向请求头中装载分布式事务上下文信息。

需要引入 feign 依赖：

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

DTF 框架注入的请求头信息为：

```
# 事务分组 ID
DTF-Group-ID: ${GroupId}
# 主事务 ID
DTF-Tx-ID: ${TxId}
# 父级分支事务 ID
DTF-Last-Branch-ID: ${LastBranchId}
```

主调 - 手动处理

可以参考 Spring Free 开发指导 中的**远程请求时传递分布式事务上下文**章节。

被调 - Spring MVC - Controller

使用 Spring MVC 的应用，在进入 Controller 前，DTF 框架会自行从请求头中检索下列 Header key。

```
# Header key的常量ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key的常量ClientConstant.HTTP_HEADER.TX_ID
DTF-Tx-ID: ${TxId}
```

```
# Header key的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

并通过 TxContextRestore 切点还原分布式事务上下文。

被调 - 手动处理

可以参考 Spring Free 开发指导 中的 远程请求时传递分布式事务上下文 章节。

与 TSF 结合使用

DTF 框架完全兼容 TSF 应用，请按照下面的指引使用。

> 说明：目前仅支持 Greenwich 版本的 TSF SDK。

Maven POM

```
<!-- TSF 启动器 -->
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>spring-cloud-tsf-starter</artifactId>
</dependency>
```

启用 TSF

```
@SpringBootApplication
@EnableDtf
@EnableTsf
@EnableTransactionManagement
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

FMT 模式 Spring Free 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 FMT 模式下进行 Spring Free 开发。

FMT 事务，也可以理解为框架托管事务。需要用户仅需要在 **FMT 规范**下正常实现业务逻辑即可实现分布式事务。相对于 TCC 事务，省去了编写 Confirm、Cancel 的代码工作。 FMT 事务的实现原理：代理用户执行 PrepareStatement 和 CreateStatement 操作，对执行的 DML 语句进行解析，记录前后象，并生成 UNDO 信息。鉴于此，FMT 对用户使用的数据库和 SQL 语句会有一定的要求，详见 FMT 规范。

准备工作

- 参考 准备工作 文档，完成环境配置和开发前准备。
- 参考 快速部署 文档，执行 FMT 初始化脚本.

开发步骤

Maven 配置

通过配置业务代码的 pom.xml 文件，可以引入 DTF 的 SDK 到您的工程中。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>dtf-core</artifactId>
<version>${dtf.version}</version>
</dependency>
```

客户端配置

在客户端中，使用以下 API 进行 DTF 的配置设置。

```
DtfEnv.setServer(String server);
DtfEnv.setSecretId(String secretId);
DtfEnv.setSecretKey(String secretKey);
DtfEnv.addTxmBroker(String groupId, String txmBrokerList);
```

配置项说明：

配置项	数据类型	必填	默认值	描述
groupId	String	是	无	用户的事务分组 ID。
txmBrokerList	String	是	无	TC 集群节点列表。
secretId	String	是	无	用户的 SecretID。
secretKey	String	是	无	用户的 SecretKey。
server	String	是	无	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识。
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用。

通常情况下，仅需要在使用 `DtfEnv.addTxmBroker()` 配置一个事务分组。例如： 用户 A，创建了一个事务分组 `group-x3k9s0ns` ，在 分布式事务控制台 获取该分组的 TC 集群地址为 `127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082` 。该用户访问密钥的 SecretId 为 `SID` ，SecretKey 为 `SKEY` 。需要在业务应用 `app-test` 上使用该事物时，配置样例为：

```
DtfEnv.setServer("app-test");
DtfEnv.setSecretId("SID");
DtfEnv.setSecretKey("SKEY");
DtfEnv.addTxmBroker("group-x3k9s0ns", "127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082");
```

代理数据源

对应用内的数据源（DataSource）信息进行手动代理。

```
DtfDataSourceProxyUtil.dataSourceProxy(String dataSourceName, DataSource dataSource);
```

> **说明：**
>
> 数据源的名称（dataSourceName）需要保持稳定，并且在一个应用中不可重复。

例如：

```
/**
* 代理数据源
*/
private static MysqlDataSource proxyDataSource() {
MysqlDataSource ds = new MysqlDataSource();
```

```
ds.setUrl(URL1);
ds.setUser(USER1);
ds.setPassword(PASSWORD);
return DtfDataSourceProxyUtil.dataSourceProxy("DATA_SOURCE1", ds);
}
```

启用分布式事务服务

在用户应用处理逻辑完成，并且完成**客户端配置**和**代理数据源**步骤后，可以使用 API 来开启分布式事务服务。

```
// 启动分布式事务客户端
DtfClient.start();
```

主事务管理

主事务的生命周期可以分为：开启、提交/回滚。

开启主事务

在**客户端配置**步骤中只添加了一个事务分组时，可以使用**开启主事务：使用默认事务分组。**添加了多个事务分组时，必须使用**开启主事务：使用指定事务分组。**

```
// 开启主事务：使用默认事务分组
DtfTransaction.begin(Integer timeout);
// 开启主事务：使用指定事务分组
DtfTransaction.begin(Integer timeout, String groupId)
```

标识一次FMT事务

每一次数据库操作都需要进行标识。

```
DtfContextHolder.get().pushBranchType(DTF.BranchType.FMT);
```

提交/回滚主事务

```
// 提交主事务
DtfTransaction.commit();
// 回滚主事务
DtfTransaction.rollback();
```

关闭主事务上下文

> **说明：**
>
> 该操作仅关闭**当前线程**的主事务上下文，不会对主事务状态产生影响。

```
DtfTransaction.end();
```

使用示例：

```
public boolean execute() {
// 业务检查
Long txId = DtfTransaction.begin(10000);
try {
// 执行各个分支事务或业务逻辑
// 标识一次FMT事务
DtfContextHolder.get().pushBranchType(DTF.BranchType.FMT);
// ... excute SQL
// 提交主事务
DtfTransaction.commit();
return true;
} catch (Throwable e) {
// 回滚主事务
DtfTransaction.rollback();
LOG.error("Bank transfer failed.", e);
return false;
} finally {
// 关闭主事务上下文
DtfTransaction.end();
}
}
```

远程请求时传递分布式事务上下文

主调

需要从上下文中提取 `groupId` 、 `txId` 、 `lastBranchId` 三个数据传递到下游。

使用以下 API 提取：

```
String groupId = DtfContextHolder.get().getGroupId();
Long txId = DtfContextHolder.get().getTxId();
Long lastBranchId = DtfContextHolder.get().getBranchIdStack().peek();
```

建议放到下列 Header 的 key 中，下游可以通过 DTF SDK 自行注入。

```
# Header key 的常量 ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key 的常量 ClientConstant.HTTP_HEADER.TX_ID
DTF-Tx-ID: ${TxId}
# Header key 的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

被调

根据上游业务的特性手动获取 `groupId`、`txId`、`lastBranchId` 三个数据。

如果上游使用的是 DTF 封装的 RestTemplate 或 Fegin，请从以下请求头中获取：

```
# Header key 的常量 ClientConstant.HTTP_HEADER.GROUP_ID
DTF-Group-ID: ${GroupId}
# Header key 的常量 ClientConstant.HTTP_HEADER.TX_ID
DTF-Tx-ID: ${TxId}
# Header key 的常量 ClientConstant.HTTP_HEADER.LAST_BRANCH_ID
DTF-Last-Branch-ID: ${LastBranchId}
```

如果是其他方式传递，则需要用户按照接口协议自行获取。

获取了三个数据后，通过 API 绑定分布式事务上下文。

```
DtfTransaction.bind(${GroupId}, ${TxId}, ${LastBranchId});
```

SAGA 模式 Spring Boot 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 Saga 模式下进行 Spring Boot 开发。 手动启动 Saga 事务，需要用户自行编写 Execute和 Compensate 接口的实现，并保证这两个方法的**幂等性**。

准备工作

- 参考 准备工作 文档，完成环境配置和开发前准备。
- 参考 快速部署 文档，执行 FMT 初始化脚本.

引入 DTF SDK

通过以下方式引入 Spring Cloud 版本的 DTF SDK。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>dtf-core</artifactId>
<version>{dtf.version}</version>
</dependency>
```

> **说明：**
>
> version 填写 Release Note 中最新版本的即可。

客户端配置

在客户端中，支持以下配置自定义：

```
dtf:
env:
groups:
{事务分组ID 1}: [{独占集群列表}]
{事务分组ID 2}: [{独占集群列表}]
secretId: {SecretID}
```

```
secretKey: {SecretKey}
server: {客户端服务标识}
```

配置项	数据类型	必填	默认值	描述
dtf.env.groups.${GroupId}	String	是	共享集群 TC 列表，如果是独占集群则需要填写	用户的事务分组 ID，单客户端使用多个事务分组时可以配置多项
dtf.env.secretId	String	是	无	用户的SecretID
dtf.env.secretKey	String	是	无	用户的SecretKey
dtf.env.server	String	否	${spring.application.name}	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用

> **说明：**
>
> 通常情况下，仅需要在 dtf.env.groups 下配置一个事务分组。

启用分布式事务服务

在 @SpringBootApplication 注解处增加 `@EnableDtf` 注解来启用分布式事务服务。

```
@SpringBootApplication
@EnableDtf
@EnableTransactionManagement
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

> **说明：**
>
> 建议同时启用本地事务管理：@EnableTransactionManagement。

开启主事务

通过以下注解开启主事务。 主事务通常在入口 Controller 处开启。一般建议标记在**实现类**上。注解所在的方法所在的类需要注入为 Bean。

主事务注解方法正常返回时提交主事务，在抛出异常时进行回滚。

```
@DtfTransactional
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
// 业务逻辑
}
```

主事务注解支持的能力包括

参数	数据类型	必填	默认值	描述
timeout	Integer	否	60 * 1000	事务超时时间（所有 Try 阶段），单位：毫秒
groupId	String	否	dtf.env.groups仅配置了一个事务分组时，使用该值	主事务的事务分组 ID
autoCommit	Boolean	否	true	为false时需要手动提交事务，即在能获取到事务上下文的地方显示调用 `DtfTransaction.commit()`

开启 Saga 分支事务

通过以下注解开启分支事务。

> **说明：**
>
> - 分支事务通常标记在 Service 上。可以标记在接口或实现类上。
> - 分支事务最好被 Spring 的 @Transactional 注解管理。

```
@DtfSaga
public boolean order(Long txId, Long branchId, Order order);
```

分支事务注解支持的参数包括：

参数	数据类型	必填	默认值	描述
name	String	否	@DtfSaga 方法名+方法签名 Hash	分支事务名称，请在同一事务分组

参数	数据类型	必填	默认值	描述
compensateClass	String	否	@DtfSaga 注解所在 Class	compensate 操作类名，建议填写 beanname
compensateMethod	String	否	execute 前缀 + @DtfSaga 注解方法名首字母大写	compensate 操作方法名

Compensate 操作

一个分支事务中，需要包含 Execute 和 Compensate 两个部分。可以使用 步骤5 中的默认值简化配置。

- 分支事务的 Execute 和 Compensate 方法所在的类需要被**注入为 Bean**。
- 分支事务的 Execute 和 Compensate 方法最好被 Spring 的 @Transactional 注解管理
- 分支事务的 Execute 和 Compensate 方法的参数**保持一致**。
- 分支事务的 Execute 和 Compensate 方法的前两个参数固定为 `Long txId` 和 `Long branchId` 。

Execute 方法：

- 本地调用 Execute 方法时 `txId` 和 `branchId` 参数传 `null` ，其他参数正常传递。
- 返回值为**业务逻辑**需要的返回值。

Compensate 方法：

- 返回值固定为 Boolean 类型。
- 仅在返回 `true` 时视为分支事务 `Compensate 成功` 。
- 返回 `false` 或**抛出异常**时，视为分支事务 `Compensate 失败` 。

```
public interface IOrderService {

@DtfSaga
boolean order(Long txId, Long branchId, Order order);

boolean compensateOrder(Long txId, Long branchId, Order order);

}
```

远程请求

方式一：使用 RestTemplate + Spring MVC 切点

使用 RestTemplate 访问下游服务（也使用了 DTF SDK）的 Controller。DTF SDK 框架托管了全局事务传递的处理。

使用以下方式进行常规注入即可。

```
@Bean
public RestTemplate restTemplate() {
return new RestTemplate();
}
```

方式二：使用 Feign

需要引入 openfeign 依赖：

```
<dependency>;
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>;
```

然后按照正常方式使用 Feign 即可：

```
@FeignClient("spring-cloud-payment")
public interface PaymentProxy {
@RequestMapping("/pay")
public Boolean pay(@RequestBody Payment payment);
}
```

方式三：自行处理

- **上游处理**：需要从上下文中提取 `txId`、`groupId`、`lastBranchId` 三个内容传递到下游。

  ```
  txId: DtfContextHolder.get().getTxId();
  groupId: DtfContextHolder.get().getGroupId();
  lastBranchId: DtfContextHolder.get().getBranchIdStack().peek();
  ```

> **说明：**
>
> 建议放到下列 Header 的 key 中，下游可以通过 DTF SDK 自行注入。

```
ClientConstant.HTTP_HEADER.TX_ID: txId
ClientConstant.HTTP_HEADER.GROUP_ID: groupId
```

```
ClientConstant.HTTP_HEADER.LAST_BRANCH_ID: lastBranchId
```

- **下游处理**： 下游可以使用以下方法将从上游传递的三个变量绑定到本地，重新开启全局事务。

```
DtfContextHolder.set(new DtfContext(txId, lastBranchId, groupId));
```

与 TSF 结合使用

引入依赖后（注意 SDK 版本），直接正常使用 TSF 即可。

使用方式

目前支持 Greenwich（G）和 Finchley（F）版本的 TSF SDK。您可以单击以下页签，查看对应的使用方式。

```
<!-- TSF 启动器 -->
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>spring-cloud-tsf-starter</artifactId>
<version>1.23.0-Greenwich-RELEASE</version>
</dependency>
```

> **说明：**
>
> 需要再排除 DTF 中的一些依赖。

```
<!-- TSF 启动器 -->
<dependency>
<groupId>com.tencent.tsf</groupId>
<artifactId>spring-cloud-tsf-starter</artifactId>
<version>1.23.5-Finchley-RELEASE</version>
</dependency>
<!-- Spring Boot DTF -->
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>spring-boot-dtf</artifactId>
<version>${dtf.version}</version>
<exclusions>
<exclusion>
<groupId>org.springframework</groupId>
<artifactId>spring-context</artifactId>
</exclusion>
<exclusion>
<groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter</artifactId>
</exclusion>
<exclusion>
<groupId>org.springframework</groupId>
<artifactId>spring-aspects</artifactId>
</exclusion>
<exclusion>
<groupId>org.springframework</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</exclusion>
<exclusion>
<groupId>io.github.openfeign</groupId>
<artifactId>feign-core</artifactId>
</exclusion>
</exclusions>
</dependency>
```

启用 TSF

```
@SpringBootApplication
@EnableDtf
@EnableTsf
@EnableTransactionManagement
public class OrderApplication {
public static void main(String[] args) {
SpringApplication.run(OrderApplication.class, args);
}
}
```

SAGA 模式 Spring Free 开发

最近更新时间: 2025-02-18 16:02:00

操作场景

该任务指导您在 Saga 模式下进行 Spring Free 开发。 手动启动 Saga 事务，需要用户自行编写 Execute，Compensate 接口的实现，并保证这两个方法的**幂等性**。

准备工作

- 参考 准备工作 文档，完成环境配置和开发前准备。
- 参考 快速部署 文档，执行 FMT 初始化脚本.

接入步骤

1. 引入 DTF SDK

通过以下方式引入 Spring Free 版本的 DTF SDK。

```
<dependency>
<groupId>com.tencent.cloud</groupId>
<artifactId>dtf-core</artifactId>
<version>{dtf.version}</version>
</dependency>
```

> **说明：**
>
> version 填写 Release Note 中最新版本的即可。

2. 客户端配置

在客户端中，使用以下方式添加基本配置。

```
DtfEnv.setServer(String server);
DtfEnv.setSecretId(String secretId);
DtfEnv.setSecretKey(String secretKey);
DtfEnv.addTxmBroker(String groupId, String txmBrokerList);
```

配置项	数据类型	必填	默认值	描述

配置项	数据类型	必填	默认值	描述
groupId	String	是	无	用户的事务分组 ID
txmBrokerList	String	是	无	TC 集群节点列表
secretId	String	是	无	用户的SecretID
secretKey	String	是	无	用户的SecretKey
server	String	是	无	客户端服务标识，一个事务分组下，同一服务需要使用相同的标识
dtf.env.fmt	Boolean	否	true	启动时会对 DB 进行大量初始化工作，若不需使用 fmt 建议禁用

> **说明：**
>
> 通常情况下，仅需要在 dtf.env.groups 下配置一个事务分组。

3. 注册 Saga

使用以下接口，手动注册 Saga 接口。

```
private static void registerSaga() {
ITransferService transferService = new TransferService();
// debit 的 compensate
SagaRegistry.register("com.tencent.cloud.dtf.demo.transfer.TransferService.debit",
SagaRegistry.SAGA.getInstance(transferService, "compensateDebit"));
// credit 的 compensate
SagaRegistry.register("com.tencent.cloud.dtf.demo.transfer.TransferService.credit",
SagaRegistry.SAGA.getInstance(transferService, "compensateCredit"));
}
```

注册方式可以参考 SagaRegistry.register() 和 SAGA.getInstance() 的 javadoc。

4. 启动分布式事务服务

通过以下方式启动分布式事务服务。

```
public static void main(String[] args) {
// 设置参数
initEnv();
// 注册 Saga
registerSaga();
// 启动分布式事务客户端
```

```
DtfClient.start();
}
```

5. 开启主事务

通过以下接口开启主事务。

主事务注解方法正常返回时提交主事务，在抛出异常时进行回滚。

```
Long txId = DtfTransaction.begin(Integer timeout);
// 或
Long txId = DtfTransaction.begin(String groupId, Integer timeout);
```

参数	数据类型	必填	默认值	描述
timeout	Integer	否	60 * 1000	事务超时时间（所有 Try 阶段），单位：毫秒
groupId	String	否	DtfEnv.addTxmBroker 仅配置了一个事务分组时，使用该值	主事务的事务分组 ID

6. 开启分支事务

通过以下接口开启分支事务。

```
Long branchId1 = DtfTccBranch.begin("name", new Object[] {null, null, this.to, this.amount});
```

分支事务注解支持的参数包括：

参数	数据类型	必填	默认值	描述
name	String	否	无	分支事务名称，请在同一事务分组
参数列表	Array <String>	否	无	参数列表，前两个参数固定为 null

7. Compensate 操作

一个分支事务中，需要包含 Execute 和 Compensate 两个部分。可以使用 步骤5 中的默认值简化配置。

- 分支事务的 Execute 和 Compensate 方法的参数**保持一致**。
- 分支事务的 Execute 和 Compensate 方法的前两个参数固定为 `Long txId` 和 `Long branchId`。

Execute 方法：

- 本地调用 Execute 方法时 `txId` 和 `branchId` 参数传 `null`，其他参数正常传递。

- 返回值为**业务逻辑**需要的返回值。

Compensate 方法：

- 返回值固定为 Boolean 类型。
- 仅在返回 `true` 时视为分支事务 Compensate 成功。
- 返回 `false` 或**抛出异常**时，视为分支事务 Compensate 失败。

```
void debit(Long txId, Long branchId, Account account, int amount) throws Exception;

boolean compensateDebit(Long txId, Long branchId, Account account, int amount);
```

8. 远程请求（自行处理）

- **上游处理**：需要从上下文中提取 `txId`、`groupId`、`lastBranchId` 三个内容传递到下游。

```
txId: DtfContextHolder.get().getTxId();
groupId: DtfContextHolder.get().getGroupId();
lastBranchId: DtfContextHolder.get().getBranchIdStack().peek();
```

> **说明：**
>
> 建议放到下列 Header 的 key 中，下游可以通过 DTF SDK 自行注入。

```
ClientConstant.HTTP_HEADER.TX_ID: txId
ClientConstant.HTTP_HEADER.GROUP_ID: groupId
ClientConstant.HTTP_HEADER.LAST_BRANCH_ID: lastBranchId
```

- **下游处理**：下游可以使用以下方法将从上游传递的三个变量绑定到本地，重新开启全局事务。

```
DtfContextHolder.bind(txId, lastBranchId, groupId);
```

SDK 进阶用法

最近更新时间: 2025-02-18 16:02:00

使用场景

该任务指导您通过配置sdk的高级参数来更加灵活地控制dtf框架的行为。

本地多事务分组配置

DTF框架支持本地微服务里存在多个不同事务分组的主事务，可以通过@DtfTransactional注解里的groupId字段来设置分组id参数。这个方法适用于分组id不频繁更换的场景，分组id就和注解一起编译在代码里了。DTF SDK还提供了一种通过配置文件参数动态调整groupId的方法。

这里引入了一个bizId的概念，用于标记不同的业务函数，然后就可以在配置文件里通过设置bizId与groupId的映射来达到动态调整的目的了。

配置项	数据类型	必填	默认值	描述
dtf.env.biz.${bizId}	String	否	无	bizId对应的groupId，单客户端使用多个事务分组时可以配置多项

下面以order方法为例，演示该如何使用bizId配置：

```
@DtfTransactional(bizId="mybiz1")
@RequestMapping("/order")
public Boolean order(@RequestBody Order order) {
// 执行业务逻辑或分支事务
}
```

首先我们将该事务标记为 `mybiz1` ，然后在配置文件里为他设置groupId为group-x3k9s0ns。

```
spring:
application:
name: app-test
dtf:
env:
groups:
group-x3k9s0ns: 127.0.0.1:8080;127.0.0.1:8081;127.0.0.1:8082
biz:
mybiz1: group-x3k9s0ns
```

```
secretId: SID
secretKey: SKEY
```

如果@DtfTransactional注解里没有标记bizId，那将和原先的逻辑一致，通过dtf.env.groups查找配置。如果同时填写了bizId和groupId，将直接使用注解的groupId。

全局锁配置

全局锁是在FMT模式下用于事务隔离的锁，SDK提供了配置，用于调整锁的超时时间。

配置项	数据类型	必填	默认值	描述
dtf.env.globalLockWaitTimeout	Integer	否	5000	全局锁等待超时时间，单位：ms
dtf.env.globalLockReleaseTimeout	Integer	否	60000	全局锁释放超时时间，单位：ms

服务端地址发现

前面客户端通过dtf.env.groups配置来指定服务端tm的地址列表，但不能很好地支持服务端扩容等地址变更的场景。客户端需要修改并且重启配置才能连接上新的服务端实例。 为了解决该问题，DTF支持了服务端地址发现的能力。客户端填写一个固定的地址发现配置即可。

配置方式如下

```
dtf:
env:
discovery:
type: builtin
builtin:
host: dtf-manager.tce.svc
port: 9810
groups:
- group-qy9484ln
secretId: sid
secretKey: skey
```

- 通过dtf.env.discovery.type指定服务发现类型，目前只支持builtin类型
- 通过dtf.env.discovery.builtin.host port指定地址端口。在控制台的事务分组->基本信息中可以看到该地址
- 通过dtf.env.discovery.groups填写事务分组列表

FMT 规范

最近更新时间: 2025-02-18 16:02:00

本文介绍 DTF 在 FMT 模式下支持的 DML 语句类型、SQL 实例，请在使用 FMT 模式时遵循相关的规范。

DML 类型	SQL 实例	语句支持
INSERT	INSERT INTO tb1_name (col_name,...)VALUES ({expr | FAULT},...),(...),... INSERT INTO tb1_nameSET col_name={expr | DEFAULT}, ...	是
UPDATE	UPDATE tb1_nameSET col_name1=expr1 [, col_name2=expr2 ...][WHERE where_definition]	是
DELETE	DELETE FROM tb1_name [WHERE where_definition]	是
SELECT	SELECT [ALL | DISTINCT | DISTINCTROW]select_expr, ... FROM tb1_name[WHERE where_definition]	是
REPLACE	REPLACE [LOW_PRIORITY | DELAYED][INTO] tb1_name [(col_name,...)]VALUES ({expr | DEFAULT},...),(...),... REPLACE [LOW_PRIORITY | DELAYED][INTO] tb1_nameSET col_name={expr | DEFAULT}, ...	否
TRUNCATE	TRUNCATE [TABLE] tb1_name	否

常见问题
使用问题

最近更新时间: 2025-02-18 16:02:00

在事务协调器故障的情况下，事务的提交和回滚还能正常进行吗?

事务协调器采用无状态节点，只要没有全部发生故障，就能够正常运行业务（性能会受到影响）。若节点全部故障，事务的提交和回滚也不受影响，重启服务后，事务仍能够正常提交或回滚。

DTF 如何与 TSF 协同使用？

1. 在 Maven pom 文件中修改配置，引入 TSF SDK。
2. 添加一行 @EnableTsf 注解，即可在分布式事务中引入微服务平台相关能力。

使用 TSF 相关能力，请参考 TSF 产品文档。

在 TCC 模式中，Confirm 和 Cancel 可能和 Try 并行甚至在 Try 之前执行吗？

可能出现。例如空回滚问题和悬挂问题。

- **空回滚问题描述**：Try 未执行时，Cancel 执行。
- *空回滚问题解决方案**：Cancel 操作判断一阶段是否已执行，识别空回滚后，直接返回成功；具体来讲：需要一张额外的事务控制表，其中有分布式事务 ID 和分支事务 ID，第一阶段 Try 方法里会插入一条记录，表示一阶段执行了。Cancel 接口里读取该记录，如果该记录存在，则正常回滚；如果该记录不存在，则是空回滚。
- **悬挂问题描述**：Try 超时，执行 Cancel，Try 接口恢复。出现的原因是 Try 由于网络拥堵而超时，事务管理器生成回滚，触发 Cancel 接口，而最终又收到了 Try 接口调用，但是 Cancel 比 Try 先到。
- *悬挂问题解决方案**：按照前面允许空回滚的逻辑，回滚会返回成功，事务管理器认为事务已回滚成功，则此时的 Try 接口不应该执行，否则会产生数据不一致，所以我们在 Cancel 空回滚返回成功之前先记录该条事务 xid 或业务主键，标识这条记录已经回滚过，Try 接口先检查这条事务xid或业务主键如果已经标记为回滚成功过，则不执行 Try 的业务操作。

在 TCC 模式中，是否需要考虑幂等？

需要考虑幂等问题。

幂等问题的解决方案为：DTF 框架会在 Confirm 和 Cancel 执行后将事务状态改为已提交或已回滚状态，因此，重复调用在 Confirm 和 Cancel 接口时，先获取状态，如果已执行，直接返回成功。

如果调用链为 A->B->C，B.try 报错，A.cancel，B.cancel，C.cancel 哪些会执行？

B 的 Try 报错，因此 C 的 Try 还未操作，因此事务协调器会发起对 A 与 B 的 Cancel。A 与 B 的 Cancel 无先后顺序。

总结：事务协调器发起 Confirm 与 Cancel 操作均无先后顺序，在 Try 成功/失败后，后续阶段的顺序本身无意义。无需执行能够带给您更高性能。

如果 Confirm 或者 Cancel 报错，重试机制是怎样的？

会有三次重试。分为两种情况：

- 情况一：Confirm/Cancel 逻辑有误，TC 下发 Confirm/Cancel 请求5秒内客户端无响应，此时会立刻开始下一次重试。若三次重试请求下发5秒内客户端都无响应，则本条主事务状态变为异常事务。
- 情况二：闪断和节点故障导致的。即 TC 下发 Confirm/Cancel 请求后客户端有响应，则会等待15分钟。以下为每一次客户端都成功响应的兜底重试策略。 第一次 Confirm 或 Cancel（非重试），持续15分钟。若超时，在1秒后开始第一次重试，持续15分钟；若依然超时，等待2秒开始第二次重试，持续15分钟；若还是超时，等待3秒开始第三次重试15分钟。若依然失败，则转为异常事务。

如果 Try 超时了，框架是否会回滚 Try 的本地事务并阻止后续调用链？

会回滚，需要注意实现幂等以及防悬挂可空回滚。会阻止后续调用。

FMT 接入模式下是否支持操作没有主键的数据库？

不支持操作没有主键的数据库。为了开发及使用的方便，请选择具有主键的数据库。

TCC 与 FMT 能否嵌套使用？

在 TCC 事务下，可嵌套 TCC 与 FMT 子事务。

在 FMT 事务下，可嵌套 FMT 子事务，不可嵌套 TCC 子事务（暂时不支持，在后续支持其他类型子事务，如非 DB 操作的 FMT 子事务，也可能产生 FMT 下 TCC 事务的主子关系）。

在一个主事务中，TCC 与 FMT 子事务支持混用。

使用 Apach 的 Sharding jdbc 报错如何处理？

DTF 的 FMT 模式暂时不兼容 Sharding jdbc，在 FMT 模式下会出现兼容性问题。如果您只需要使用 TCC 模式，可参考开发文档 - 客户端配置，配置 dtf.env.fmt 为 false 以解决该问题。

通用参考
TCC 模式

最近更新时间: 2025-02-18 16:02:00

概述

TCC（Try - Confirm/Cancel）指户根据自己的业务场景实现 Try（初步操作）、Confirm（确认操作）和 Cancel（取消操作）三种操作，TCC 是目前被广泛应用的一种分布式事务模式。

TCC 模式中，资源管理器（RM）可对跨数据库、跨服务的资源进行管理，使得对多个数据库的访问和对不同业务的操作转变为一个原子操作，从而解决了复杂场景下事务的一致性问题。且 TCC 模式中无全局行锁，每次操作对于数据库而言都属于本地操作，结束操作则事务结束，数据库资源释放，相比传统的 2PC 方案性能显著提升。

TCC 执行流程

TCC 事务中的三个重要角色：

- TM（TransactionManager）：事务管理器
- RM（Resource Manager）：资源管理器
- TC（Transaction Coordinator）：事务协调器

TCC 事务执行流程如下：

1. 事务发起者的 TM 向 TC 申请开启一个全局事务，得到主事务 ID。
2. TM 将主事务 ID 传给需要调用的各微服务中。
3. 各微服务中的 RM 利用得到的主事务 ID 向 TC 注册分支事务 ID，将其纳入主事务 ID 的管辖。
4. RM 完成自身业务逻辑（即完成 TCC 中的 Try 阶段）。
5. 发起者的 TM 向 TC 确认执行 Confirm/Cancel，取决于全局事务 Try 阶段的结果。
6. TC 通知各 TM 执行 Confirm/Cancel。
7. 各服务的 RM 执行 Confirm/Cancel。
8. 各服务的 TM 上报给 TC 执行 Confirm/Cancel 的结果。

DTF 的 TCC 优势

相比于传统的 TCC 模式，DTF 的 TCC有以下优势：

性能优化

- DTF 对 TC 的处理逻辑进行优化，通过优化主事务、分支事务的状态机制，减少了超过80%每次主事务、分支事务对数据库的操作次数。
- DTF 对 TC 集群的数据库操作进行了优化，移除了 TC 中全部本地事务操作，用代码逻辑保证一致性，大幅提升性能。

TC 集群高可用

- TC 部分宕机时，TC 会自动尝试另一协调器节点，保障业务正常运行。
- TC 采用无状态节点，可以快速故障恢复。在 TC 冷启动时，可以从数据库恢复上下文。
- 当 TC 集群的数据库出现异常时，可自动切换备用数据库。
- 即使 TC 完全宕机，也不会导致丢失事务或导致事务不一致。TC 此时会阻止新事务的注册，并在恢复后保障进行中事务的一致性。

FMT 模式

最近更新时间: 2025-02-18 16:02:00

概述

FMT（Framework-managed Transaction）模式下，DTF 通过框架解析您的 SQL 语句，免去了编写 Confirm/Cancel 方法的烦恼，接入使用便捷，对代码无侵入，助您高效完成业务分布式事务的开发。 FMT 模式与 TCC 模式的事务协调器 TC 并无差异，仅在 RM 操作层面与 TCC 产生差异。

FMT 执行流程

FMT 框架会代理数据驱动层，执行流程如下：

- 执行正常逻辑时，可以理解为 Try 阶段。框架解析 SQL 语句，生成 SELELCT 语句和 UNDO 语句模板，等待全局锁，然后在同一个本地事务中：注册分支事务、查询前象、执行 SQL、查询后象、记录 UNDO LOG。如果过程中出现异常，则会释放全局锁，否则会等待分支事务提交/回滚时再释放全局锁。
- Confirm 过程相对简单，删除 UNDO LOG 并释放全局锁即可。
- Cancel 过程稍微复杂，先要检查当前数据是否符合 Try 中查询的后象，然后执行 UNDO 语句，再检查数据是否符合 Try 中查询的前象，最后删除 UNDO LOG 并释放全局锁。 如果出现前象或者后象不一致时，回滚本地事务，不释放全局锁，等待人工介入（异常事务）。

DTF 的 FMT 优势

支持可重入锁

相比目前友商的 FMT 模式或类似模式，DTF 能够处理更复杂的应用场景：可重入锁。

- 举例如下：在一条主事务中，多个分支事务对数据表中某一行数据进行了多次操作：增（create）、改（update 可能多次）、删（delete），此时会涉及到可重入锁的问题。
- 实际场景：在一次交易中，新增了一条订单数据，而本次交易又涉及到其他操作（其他分支事务），需要修改本条订单数据。这种场景下，如果需要回主事务，会涉及到按倒序回滚改操作、增操作，即反向执行可重入锁。

为了保证事务一致性，FMT 会在业务操作某一行数据时进行锁行，在重入锁的场景下，会涉及到生成多次行锁。目前友商的同类型模式，不支持处理重入锁场景的回滚操作。DTF 的 FMT 模式支持可重入锁，助您在配置简单的 FMT 接入模式下实现更广阔的业务需求。

词汇表

最近更新时间: 2025-02-18 16:02:00

ACID

ACID 是数据库事务正确执行的四个特性的缩写：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务的数据库，必需要具有这四种特性，否则在事务过程当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。

BrokerList/集群 TC 端口号

BrokerList/TC 端口号是 TC 集群的端口号。在分布式事务 DTF 中，用户需要配置工程文件中的 BrokerList 使得 TC 集群能够操作主事务/分支事务。

DTF/分布式事务

分布式事务（Distributed Transaction Framework，DTF）是自主研发的高性能、高可用的分布式事务中间件，用于提供分布式场景（特别是微服务架构）中事务一致性服务。在分布式事务中，事务的发起者、资源、资源管理器和事务协调者分别位于不同的分布式系统的不同节点之上。

分支事务

一个分布式事务可能包含多个数据库本地事务，在分布式事务框架下，分支事务（Branch Transaction）可能是一个分库上执行的 SQL 语句，或是一个自定义模式服务的调用。

FMT

FMT（Framework-managed Transaction）是框架管理事务，通过框架托管的分布式事务实现模式，无需编写 Confirm/Cancel 逻辑。

两阶段提交协议

两阶段提交协议（Two-Phase Commit protocol，2PC）是分布式事务的处理协议。该协议将一个分布式的事务过程拆分成两个阶段：投票和事务提交 ，以保证分布式系统中数据的一致性。

SecretKey

SecretId 和 SecretKey 合称为云 API 密钥，是用户访问 API 进行身份验证时需要用到的安全凭证。SecretKey 是用于加密签名字符串和服务器端验证签名字符串的密钥。一个 APPID 可以创建多个云 API 密钥。

SecretId

SecretId 和 SecretKey 合称为云 API 密钥，是用户访问 API 进行身份验证时需要用到的安全凭证。SecretId 用于标识 API 调用者身份。一个 APPID 可以创建多个云 API 密钥。

事务

事务（Transaction）是作为单个逻辑工作单元执行的一系列操作。 事务的执行有一致性，同一个事务只能同时被操作或不被操作。

事务发起者

事务发起者（Transaction Manager）负责发起分布式事务，将参与者纳入到分布式事务当中，决定最终分布式事务是提交还是回滚。一个主事务只能有一个事务发起者。

事务分组

事务分组是分布式事务 DTF 的逻辑资源，类似于一个服务实例。

TC/事务协调器

事务协调器（Transaction Coordinator，TC）是负责调度事务运行的支撑环境集群，由虚拟机和数据库组成。

TCC

TCC（Try-Confirm/Cancel）指用户根据自己的业务场景实现 Try（初步操作）、Confirm（确认操作） 和 Cancel（取消操作）三种操作，是类似两阶段提交协议的经典分布式事务实现。

API文档