iuap 红皮书
分布式事务

2022 年 5 月

版权

本文摘要

YTS 全称为 Yonyou Transaction System，是 YonBIP 微服务间服务调用时解决各微服务数据一致性的分布式事务管理框架。框架支持 YonBIP 系统的 MDD、Iris 以及基于 HTTP 协议 RESTful 方式的远程接口调用。

本文主要介绍了 YTS 框架的基本原理，功能特性，支持的事务模式以及在各种框架下的开发，集成以及测试等内容。可概括为以下核心要点：

1、YTS 的基础架构和功能架构；

2、YTS 运行机制以及事务模式以及使用业务场景；

3、各框架接入的 YTS 的对接说明；

4、分布式事务测试中模拟异常机制的使用；

5、事务模式以及使用应用场景介绍。

开发者通过对本文的学习，可了解 YTS 在微服务数据一致性场景的开发，测试，部署中的相关知识。掌握如何在微服务场景中使用 YTS 解决 RPC 调用引发的各微服务间数据一致性问题的方法。

熟悉本文内容，可了解系统中数据一致性产生原因、解决方法以及如何使用 YTS 提供的运维工具发现产生数据不一致的异常，以及如何根据 YTS 提供的异常堆栈排查问题，使业务数据达到最终一致。

文档修订摘要

日期	修订号	描述	著者	审阅者
2020-10-19	序列号 1	对文档的简单概述	GH	
2022-05-18	序列号 2	增加 httpclient 对接以及自动重试	LYS	
2022-05-20	序列号 3	增加使用场景	YMY	
2022-05-26	序列号 4	调整目录结构，增加新的打桩以及场景介绍	LYS	
2022-05-28	序列号 5	调整开发示例	YMY	
2022-06-07	序列号 6	增加框架运行原理，以及 Mock 机制说明，并且完善了开放接口说明	LYS	
2022-06-20	序列号 7	增加内容摘要	YMY	

目 录

第一章 基本概念

1.1 背景

在业务系统中，业务对象会有多个阶段的处理，需要保证业务的处理的事务是一致。常见的比如取款过程中，减掉账户和付给现金的操作，都会引起账户数据的变化，所以需要在业务发生过程，保持一致性。为了满足此场景，需要提供分布式事务的处理的解决方案。

本框架目前针对用友 YonBIP 以及其他各领域云如何保障各个微服务之间的数据一致性考虑，主要针对同步调用场景下的 MDD,iris 微服务以及基于 SpingMVC 架构的 RESTful 接口的相互调用。

框架和 iuap 平台下的 MDD、RPC 以及 HTTP 结合，结合本地数据库事务提交机制，目前支持 Mysql、oscar,PostgreSQL 数据库，并结合用友的数据库中间件的 SQL 翻译支持更多的数据库。

技术中台的云端，提供事务监控中心，监控和查看未及时完成的以及未正确结束的长事务，支持云端查看错误堆栈、链路拓扑图、下发重试命令等功能。

1.2 名词解释

MDD:iuap 平台下的元数据驱动设计框架，前后端的统一基于元数据的架构。

YTS-SDK： iuap 微服务架构下，数据一致性框架提供的 java SDK，包括底层的事务管理器和事务协调器、上层的 rpc 框架适配及容错相关的功能。

定时任务：处于 YTS-SDK 中的定时处理指定功能的线程。

规则（rule）： MDD 概念中的编码规则，一个规则对应一系列的代码执行操作。

事务模式：YTS 框架所支持的分布式事务解决方案，不同的方案解决不通的问题场景，目前所支持的事务模式有同步 Sagas 模式、TCC 模式以在同一长事务中 Sagas，TCC 模式混用。

回滚：在全局事务失败之后，由 YTS 事务协调器发起的针对所有参与者的反向操作。

重试：本文特指在调用回滚(cancel)或提交(confirm)操作报错后在云端下发命令后业务重新执行回滚(cancel)或提交(confirm)的操作。

1.3 产品亮点

1.3.1 并发幂等性

YTS 框架分布式事务的 LOG 与业务数据同库模式，确保事务状态的变化与业务数据操作的一致性，从而保证了全局事务状态的最终一致性

业务幂等性：

YTS 框架严格保证正向业务幂等，业务不用再考虑幂等性，即使业务本身无法进行幂等控制，框架也能确保业务只能成功执行一次

confirm/cancel 幂等性：

YTS 框架严格保证 confirm/cancel 幂等，即便多次发起 confirm/cancel 等操作，框架也能确保其只能正确执行一次

并发性：

YTS 框架严格保证因网络等原因造成的业务并发执行或 confirm/cancel 并发执行、空回滚、正向滞后、正反向悬挂等场景造成的数据不一致现象

1.3.2 隔离性

隔离性是指在全局事务参与者在其业务方法的本地事务成功提交到其收到事务协调器的提交或回滚操作期间，为了避免其它全局事务对该事务提交的数据进行操作而导致脏读、幻读或无法回滚、无法提交等现象的解决方案，通常基于最终一致性的分布式事务的隔离是软隔离，需要业务系统遵循相应的规则。

1.3.3 容错性

YTS 框架提供完善的容错与问题排查机制，方便用户定位问题并解决问题。事务提交或回滚失败后，YTS 框架会将该全局事务上报到云端控制台，并根据链路信息能够定位到导致错误的原因，并提供努力向前重试机制，确保最终能提交或回滚成功。

第二章 技术架构

2.1 总体架构设计

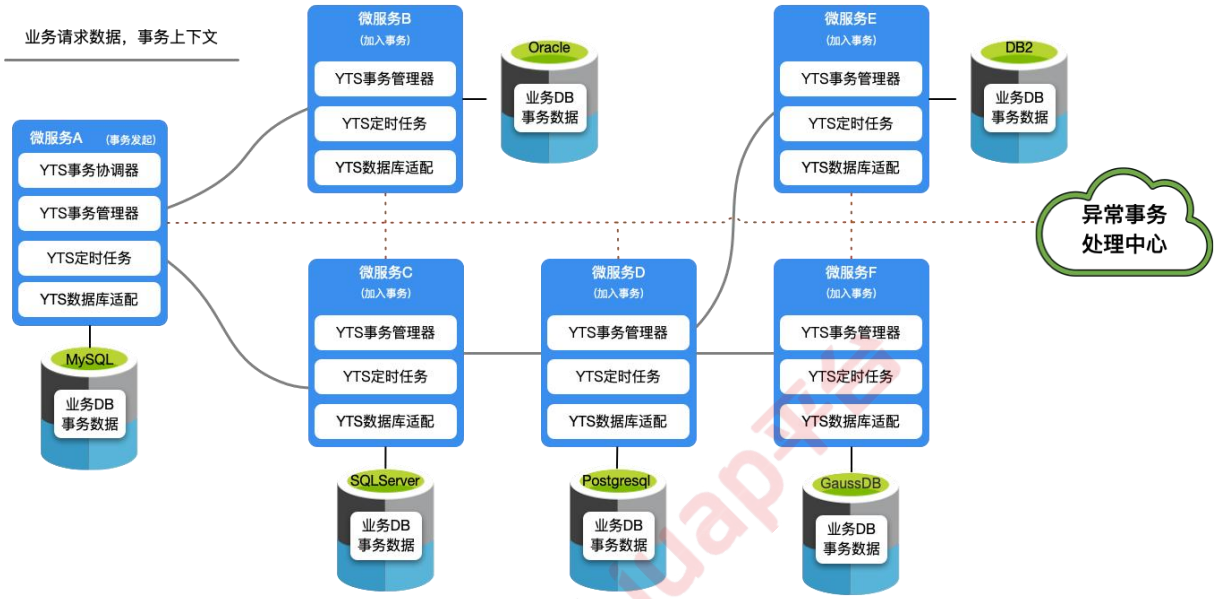


图 1：架构设计

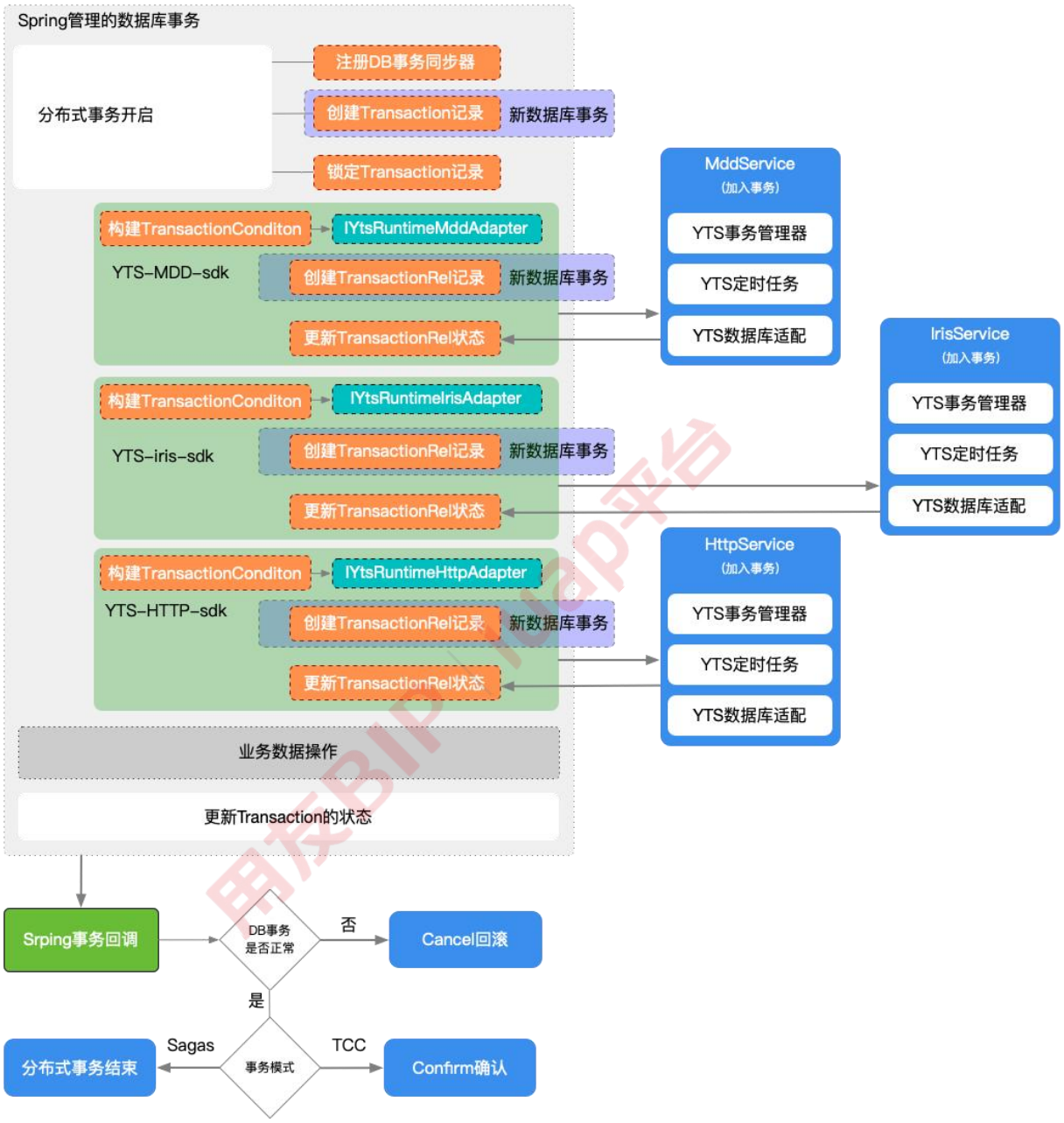
（1） 分布式事务框架 YTS 依托于微服务治理平合的 RPC 框架，和 iuap 平台的技术平台紧密结合，统一对外提供数据最终一致性解决方案；

（2） YTS 框架支持多种运行态，包括 MDD、纯 IRIS、 HITIP 等；支持多种一致性控制模式，如同步 Sagas、TCC 等。不同的运行时和控制模式的组合满足了绝大多数的业务场景需求。

（3） 框架依赖很多技术底层技术：数据库本地事务回调、分段的数据库事务日志，定时任务、Spting 框架的事务管理器等等。框架支持多种数据库类型，支持 Mysql、SQIServer、Oracle 等主流数据库及达梦等国产数据库。

（4） YTS 框架包含可视化监控及问题排查能力，通过对错误数据的上报，错误堆栈的抓取，全局链路数据的跟踪，可以方便的进行错误事务的拓扑图展示。

2.2 YTS 工作机制



YTS 的抽象了 RPC 框架的调用接口，协调器根据 RPC 框架的接口实现来保存相应框架的 YTS 事务上下文，并使用 Spring 数据库事务的扩展机制在 Spring 的数据库事务提交前（beforeCommit）以及事务结束后注册回调来实现在事务提交前修改 YTS 的事务记录状态，并在事务结束后根据分布式事务模式以及 Spring 事务的提交的状态来调用相应 RPC 框架的 Confirm 或者 Cancel 逻辑。

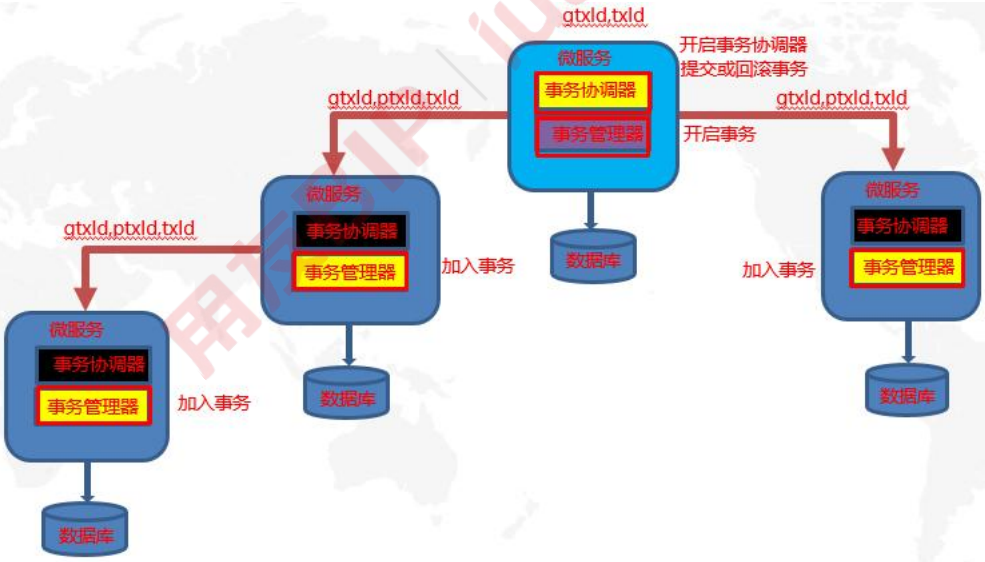
2.3 YTS 事务

2.3.1 YTS 事务协调器

YTS 事务协调器是基于分布式模式，以模块的形式供应 用使用，事务协调器的作用在于对根据全局事务的状态及事务模式，决定是否发起回滚或者是否发起提交。全局事务起点开启事务时，YTS 框架会启动 YTS 事务协调器，并根据事务模式，注册相应的 Spting 事务回调钩子。

2.3.2 YTS 事务管理器

YTS 事务管理器负 责事务边界管理，负责开启或加入事务，并记录相应的事务日志及调用关系。事务管理器与事务协调器一样，以模块方式提供，当通过 AT 开启开启事务或基于注解或 MDD 规则配置开启 事务或加入事务时接入事务管理功能，如果是事务起点，开启事务协调器。



第三章 功能介绍

3.1 功能概述

3.1.1 功能架构

YTS 功能架构

3.1.2 功能特性

3.1.2.1 支持多种事务模式

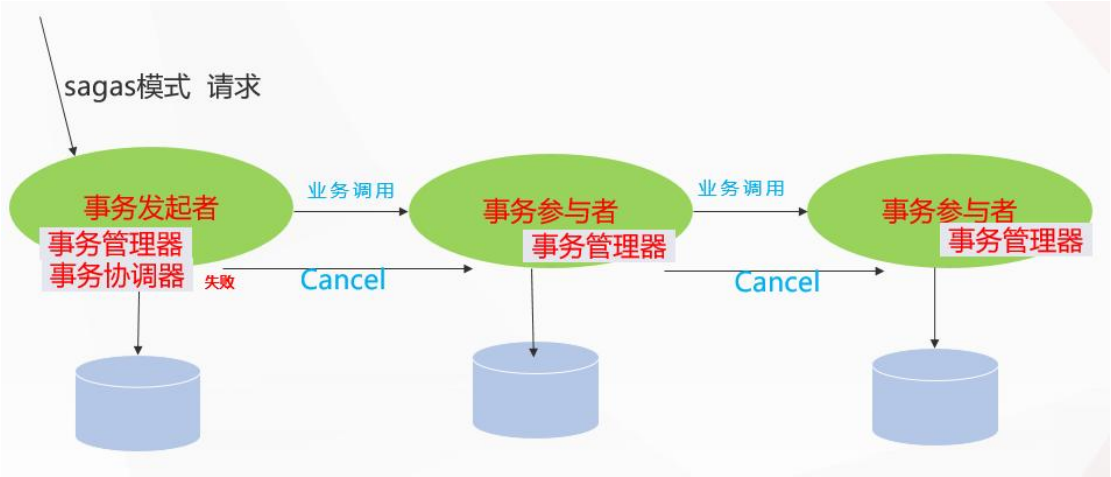
YTS 事务框架作为统一的分布式事务框架，提供了主流的分布式事务解决方案，如同步 Sagas 模式、TCC 模式、以及同一长事务内 Sagas，TCC 混用，基本涵盖了绝大部分业务场景。

3.1.2.1.1 sagas 模式

Saga 是一种补偿模式，分布式事务内有多个参与者，每一个参与者的业务都存在一个反向操作，用于取消全局事务异常时参与者需要反向执行的动作，取消正向操作成功后的影响

分布式事务执行过程中，依次执行各参与者的正向操作，如果所有正向业务均执行成功，那么分布式事务提交。如果任何一个正向业务执行失败，那么分布式事务触发去

执行前面各参与者的反向回滚操作，回滚已提交的参与者，使分布式事务回到初始状态



sagas 模式

Sagas 模式适用场景：

每个参与者均提供反向操作，当正向业务有失败时，由事务协调器调用参与者提供的反向操作

Sagas 模式缺点：

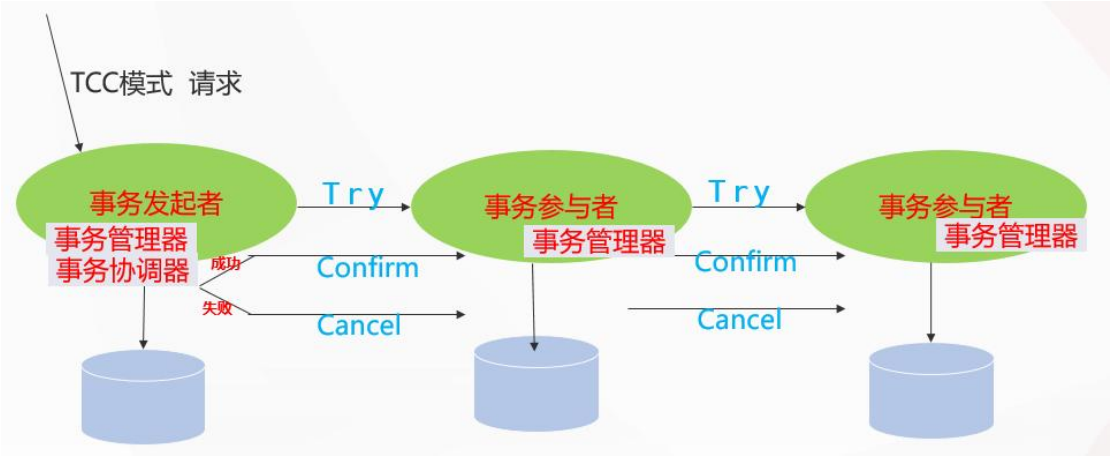
不具备隔离性，可能会导致无法回滚的情况

Sagas 模式条件：

反向操作一定成功，但允许全局事务在一定时间内的不一致，即在所有参与者成功回滚前，各参与者的数据可能是不一致的

3.1.2.1.2 TCC 模式

TCC 模式把业务的操作从逻辑上划分为两个阶段即 Try 阶段和 Confirm/Cancel 阶段，首先通过 RPC 框架调用各参与者的业务方法（Try），如果事务发起者本地事务成功提交，事务管理器则认为全局事务成功，调用各参与者的 Confirm 方法，否则事务发起者调用各参与者的 Cancel 方法



TCC 模式

Try：尝试执行业务阶段

完成所有业务检查（一致性）

预留必须业务资源（准隔离性）

基于中间状态执行资源处理

Confirm：提交阶段

确认执行业务，只使用 Try 阶段预留的业务资源真正执行业务

将中间状态的资源置为最终状态

Cancel：回滚阶段

释放 Try 阶段预留的业务资源

将中间状态的资源恢复

TCC 模式特点：

1)由于从业务层拆分成两个阶段，可以通过两阶段模式实现业务资源的隔离

2)每个阶段执行完成后提交本地事务，性能好

TCC 模式缺点：

业务侵入性高，对于一些老系统代码改动逻辑大

TCC 模式条件：

Confirm/Cancel 操作一定成功，允许全局事务在一定时间内的不一致

3.1.2.2 异常事务自动重试

当事务由于服务异常获取网络故障导致 cancel 或者 confirm 异常，云端控制台可根据配置事务异常时自动重试，解决由于服务不稳定以及网络故障导致 cancel 和 confirm 不正常执行导致的数据不一致的问题。云端控制台可为单独的微服务配置，每次自动重试的间隔时间以及总计自动重试的次数。

第四章 开发示例

4.1 YTS 事务开发

4.1.1 MDD 接入模式

基于 MDD 规则配置 YTS 事务及事务模式，在执行远程规则时，如果需要将远程规则纳入 YTS 分布式事务管理，则通过改规则的 config 字段配置事务信息为如下内容，同时跨域规则需要实现 ISagaRule 或 ITccRule 接口

```
{
   "needDTC": "true",
   "transactionType": "sagas",
   "params":"param,testcacel"
}
```

needDTC：true，标识要开启 YTS 事务

transactionType：支持 tcc、sagas 模式

params：正向的 map 中需要向 cancel 传递值的 key 列表，多个以“,”分隔

ISagaRule 接口定义

```
public interface ISagaRule {
RuleExecuteResult cancel(BillContext billContext, Map<String,Object> paramMap) throws Exception;
}
```

ITccRule 接口定义

```
public interface ITccRule {
RuleExecuteResult confirm(BillContext billContext, Map<String,Object> paramMap) throws Exception;
RuleExecuteResult cancel(BillContext billContext, Map<String,Object> paramMap) throws Exception;
}
```

本地事务配置

如果远程规则配置了 YTS 分布式事务，调用该规则前必须已经开启了本地数据库事务，远程规则服务执行规则前，也必须开启了本地数据库事务，否则框架将会报错

4.1.2 IRIS 微服务接入模式

在业务接口定义上添加 YTS 事务注解

```
@YtsTransactional(
    cancel = "cancelPlane",
    confirm = "confirmPlane",
    mode = TransactionMode.SAGAS
    )
```

配置 YTS 分布式事务

cancel 属性：指定事务回滚时的回滚方法，与业务方法在同一个类

confirm 属性：TCC 模式时第二阶段提交方法，与业务方法同一个类

mode 属性：事务模式，支持 tcc、sagas 模式

本地事务配置

配置了 YTS 事务的接口实现方法执行时必须开启了本地数据库事务，否则会报错

4.1.3 HTTP 服务端接入 YTS

在 controller 上加入事务注解并提供 confirm 或 cancel 方法即可

```
    @GetMapping("/save")
@YtsTransactional(mode = TransactionMode.SAGAS, cancel="cancelSagas")
public String sagas(HttpServletRequest req, @RequestParam("uid") Long uid, @RequestParam("amount") Long amount) throws SQLException {

        // 正向逻辑
    return "hello";
}
    public String CancelSagas(HttpServletRequest req, @RequestParam("uid") Long uid, @RequestParam("amount") Long amount) throws SQLException {
            // 回滚逻辑
    return "hello";
}
```

调用支持 YTS 分布式事务 HTTP 的代码：

注入 YtsHttpClient 的服务

使用如下方式调用：

```
    String url = "http://demo.app.yyuap.com/yts/save?uid=" + uid;
String result = ytsHttpClient.get(url);
```

4.1.4 HTTP 客户端 httpclient 的接入

4.1.4.1 依赖 yts-httpclient-support 的包

```
<dependency>
    <groupId>com.yonyou.cloud</groupId>
    <artifactId>yts-httpclient-support</artifactId>
</dependency>
```

4.1.4.2 修改基于 httpClient 的 sdk

为 sdk 增加 HttpClient 自动加载 HttpRequestInterceptor 的插件机制。

```
HttpClientBuilder httpClientBuilder = HttpClients.custom().setConnectionManager(this.httpPool).setDefaultRequestConfig(this.requestConfig);
try {
    ServiceLoader<HttpRequestInterceptor> requestInterceptors = ServiceLoader.load(HttpRequestInterceptor.class);
    requestInterceptors.forEach((x) -> {
        httpClientBuilder.addInterceptorFirst(x);
    });
    ServiceLoader<HttpResponseInterceptor> responseInterceptors = ServiceLoader.load(HttpResponseInterceptor.class);
    responseInterceptors.forEach((x) -> {
        httpClientBuilder.addInterceptorFirst(x);
    });
} catch (Throwable e) {
    logger.error(e.getMessage(), e);
}

this.httpClient = httpClientBuilder.build();
```

4.1.4.3 项目中注入使用业务数据库连接池的 JdbcTemplate 的 bean

4.1.4.4 为 http 的调用开启 yts 的支持

header("ytsEnable", "true");

Sagas 模式

header("ytsMode", "sagas");

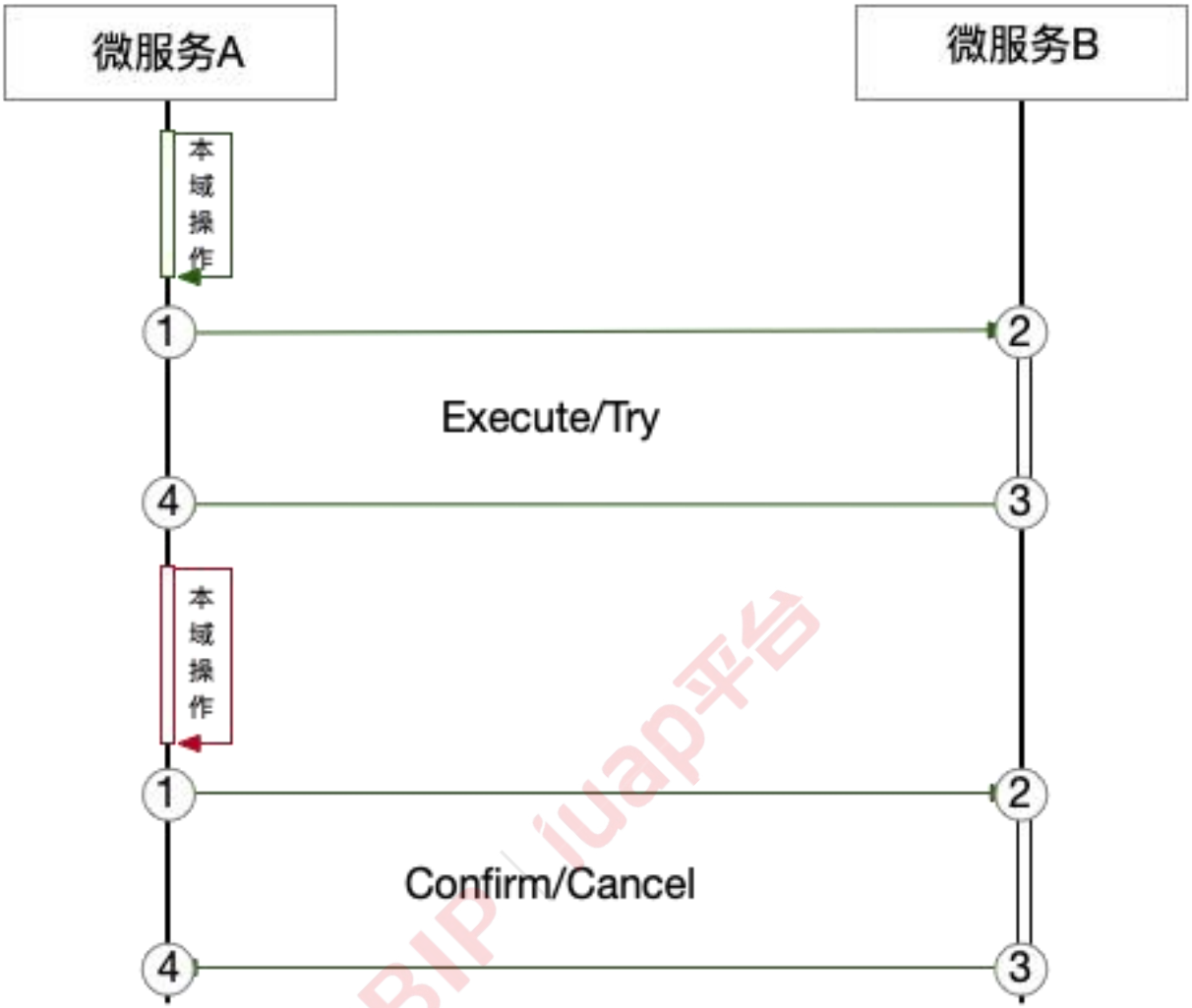
TCC 模式

header("ytsMode", "tcc");

4.2 打桩 Mock 机制

为了开发人员在开发和测试过程中方便对 YTS 分布式事务各个阶段功能进行测试，YTS 框架在业务调用的调用方，被调用方的各个关键节点进行埋点，不在关键节点可以进行常见类型的异常模拟。

对各个节点打桩(模拟异常)进行了界面话操作，方便开发，测试人员在开发和测试阶段方便的进行打桩，Mock 机制的埋点如下图：

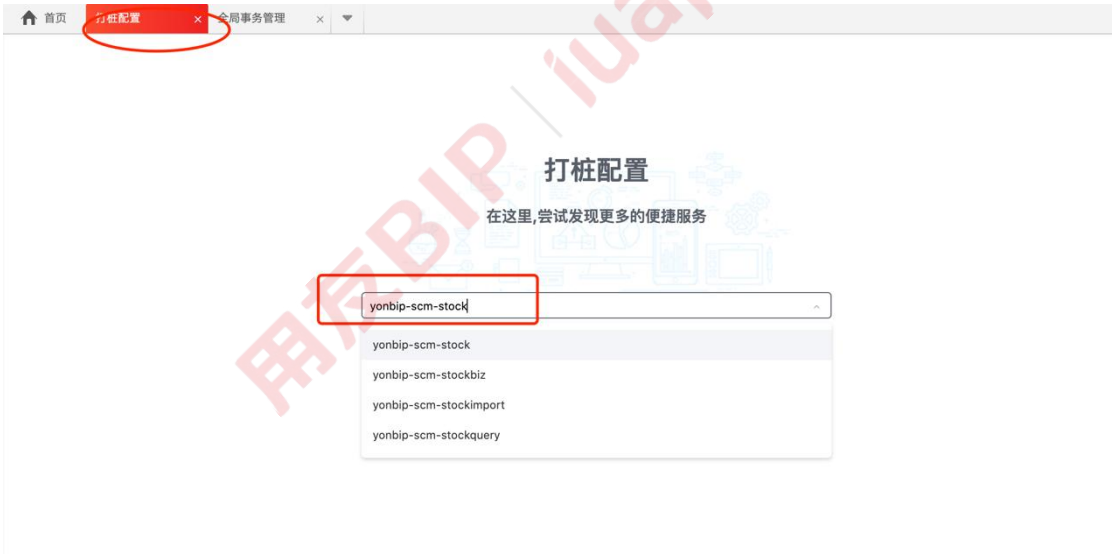


YTS 在 Sagas 模式时有 Execute，正常业务调用阶段以及发生异常后 Cancel（回滚）阶段；TCC 模式时有业务资源尝试锁定(Try)以及所有下游 Try 成功后 Confirm 阶段，以及如果有异常而 Cancel(回滚)阶段。在任何阶段的上游调用前（1）后（4）以及被调用方的业务逻辑执行前(3)后(4)位置有 YTS 的异常 Mock 的埋点。可以通过动态配置的方式指定在指定阶段，以及位置抛出指定异常。

技术中台配置文件方式模拟异常

进入开发者中心，打开测试服务下的打桩配置标签页。

不同租户只会显示各自权限下的服务名称，选择想要打桩的服务名。



进入打桩页面之后首先需要选定打桩的环境以及打桩类型，点击“新增”可以添加对应类型的打桩信息，不同的打桩类型对应不同的填写字段，添加后的打桩信息可以修改和删除。

注意：需要明确对应的操作场景下，模拟报错的单据类型、动作、具体的规则 code、租户 id；

下面说明三种不同类型框架的打桩字段：

4.2.1 Mdd 框架

租户 id：租户标志如 cwginzia，可以通过在 cookie 中尝试获取。

单据类型：单据类型如 cb539bf6，通常在表单中设定。

动作：模拟报错的具体动作，常见的填写包括 save、add、refer、audit 等。

RULE ID：模拟报错的打桩规则如 beforeSaveHandleIsUpdateCost，通常可以查询应用构建中的规则配置。

阶段：模拟报错的阶段，区分为执行（execute）、确认(confirm)、回滚(cancel)

异常类型：模拟报错的类型，区分为服务宕机（crash）、数据库保存失败（sql）、超

时网络抖动（timeout）。

超时时间：如果报错类型为超时网络抖动（timeout），则需要在超时时间中填写超时时间，否则不需要。

位置：模拟报错的位置，区分为前（front）和后（rear）,可以理解为调用前/后报错。

异常信息：对该次打桩的信息说明。

4.2.2 Iris 框架

包名：如 com.yonyou.iuap.rpc.service。

类名：如 IFastJsonTestServicefastJsonTest。

方法名：如 fastJsonTest。

参数类型列表：如 java.lang.String,java.lang.String(注意要用英文逗号隔开各个参数，没有参数则不填)。

阶段：模拟报错的阶段，区分为执行（execute）、确认(confirm)、回滚(cancel)。

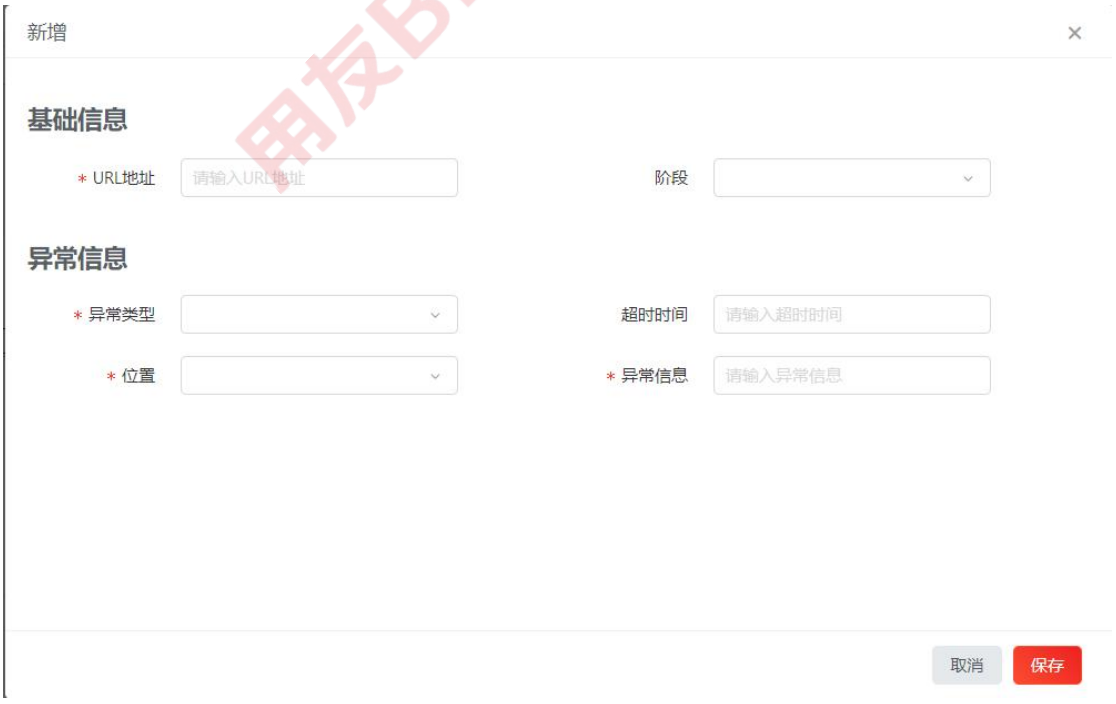
其余字段同 mdd。

4.2.3 Http 框架

URL 地址：如 http://yh-mwclient-provider-yh.online.app.yyuap.com/yts/save。

阶段：模拟报错的阶段，区分为执行（execute）、确认(confirm)、回滚(cancel)。

其余字段同 mdd。



第五章 典型业务场景介绍

5.1 MDD 规则引擎框架下使用

MDD 的框架内置 YTS 相关的逻辑，标准 MDD 项目无需引入其他的依赖包。

5.1.1 Sagas 模式

（1）开启分布式事务的 Rule 如果事务模式是 Sagas 模式则需要实现

com.yonyou.ucf.mdd.ext.trans.itf.ISagaRule 接口实现 Sagas 模式 Rule 的核心实现逻辑样例：

```
/**
 * 处理用户信用变动的逻辑，正向操作时先记录信用变动明细记录，然后改变客户的信用值。信用变动记
* 录中保存全局事务 ID(gtxId)得值记录信用变动记录和分布式事务的关联关系。cancel 时先查看变动明细的
* 状态，如果没有回滚过则修改状态为已回滚并改变客户的信用。如果状态已 cancel 则直接返回回滚成功
 */
@Component("creditChangeSagasSaveRule")
public class CreditChangeSagasSaveRule extends AbstractCommonRule implements ISagaRule {
    @Override
public RuleExecuteResult execute(BillContext billContext, Map<String, Object> map) throws Exception {
    List<BizObject> bills = this.getBills(billContext, map);
    for (BizObject bizObject : bills) {
        String saleOrg = bizObject.get(B_SALE_ORG);
        if (StringUtils.isEmpty(saleOrg)) {
            throw new BusinessException("销售组织不能为空");
        }
        String billCode = bizObject.get(CODE);


        BizObject credit = getCredit(saleOrg, billCode);
        if (credit == null) {
            throw new BusinessException("客户信用记录不存在");
        }
        BigDecimal billValue = bizObject.get(B_REBATE_MONEY);
        BigDecimal curValue = credit.get(C_CREDIT_AMOUNT);
        BigDecimal curavailableValue = credit.get(C_CREDIT_AVAILABLE_AMOUNT);

        BillDataDto creditLogDto = buildCreditLog(credit, bizObject);
```

```
            BillContext creditLogContext = new BillContext();
            creditLogContext.setBillnum(CREDIT_LOG_BILL_NUM);
            creditLogContext.setFullname(CREDIT_LOG_URI);
            creditLogContext.setAction(OperationTypeEnum.SAVE.getValue());
            creditLogContext.setEntityCode(CREDIT_LOG_BILL_NUM);

            Map<String, Object> creditLogMap = Maps.newHashMap();
            creditLogMap.put("param", creditLogDto);

            try {
                BillBiz.executeRule(OperationTypeEnum.SAVE.getValue(), creditLogContext, creditLogMap);
            } catch (Exception e) {
                throw new BusinessException("信用明细保存失败", e);
            }

            credit.setEntityStatus(EntityStatus.Update);
            credit.set(C_CREDIT_AMOUNT, curValue.add(billValue));
            credit.set(C_CREDIT_AVAILABLE_AMOUNT, curavailableValue.add(billValue));
            MetaDaoHelper.update(CREDIT_URI, credit);
        }
        return null;
    }
    @Override
    public RuleExecuteResult cancel(BillContext billContext, Map<String, Object> paramMap) throws Exception {
        String gtxId = currentGtxId();
        QueryConditionGroup typegroup = QueryConditionGroup.and(
                QueryCondition.name("gtxId").eq(gtxId));
        QuerySchema query = QuerySchema.create()
                .addSelect("*")
                .addCondition(typegroup);
        List<Map<String, Object>> datas;
        try {
            datas = MetaDaoHelper.query(CREDIT_LOG_URI, query);
        } catch (Exception e) {
            throw new BusinessException("没有查询到信用明细", e);
        }
        if (CollectionUtils.isEmpty(datas)) {
            throw new BusinessException("没有查询到信用明细");
        }
        long creditId = 0;
        BigDecimal money = null;
```

```
        List<BizObject> logs = new ArrayList<>();
        for (Map<String, Object> data : datas) {
            BizObject log = toBizObject(data);
            log.setEntityStatus(EntityStatus.Update);
            log.set(STATUS, "已回滚");
            money = log.get(CREDIT_AMOUNT);
            logs.add(log);
            creditId = Long.parseLong(log.get("creditId"));
        }
        try {
            MetaDaoHelper.update(CREDIT_LOG_URI, logs);
        } catch (Exception e) {
            throw new BusinessException("回滚信用数据明细失败", e);
        }
        BizObject credit = MetaDaoHelper.getById(CREDIT_URI, creditId);
        if (credit == null) {
            throw new BusinessException("没有查询到信用数据");
        }
        BigDecimal curValue = credit.get("creditAmount");
        BigDecimal curavailableValue = credit.get("availableCreditAmount");
        credit.set(C_CREDIT_AMOUNT, curValue.subtract(money));
        credit.set(C_CREDIT_AVAILABLE_AMOUNT, curavailableValue.subtract(money));
        credit.setEntityStatus(EntityStatus.Update);

        try {
            MetaDaoHelper.update(CREDIT_URI, credit);
        } catch (Exception e) {
            throw new BusinessException("回滚信用数据失败", e);
        }
        return new RuleExecuteResult();
    }

    private BizObject toBizObject(Map<String, Object> data) {
        BizObject bizObject = new BizObject();
        bizObject.putAll(data);
        return bizObject;
    }
}
```

调用方注册 rule 时 config 的配置如下：

{"needDTC":true}

5.1.2 TCC 模式

（1）开启分布式事务的 Rule 如果事务模式是 TCC 模式则需要实现 com.yonyou.ucf.mdd.ext.trans.itf.ITccRule 接口 TCC 模式 Rule 的核心实现逻辑样例：

```
/**
 * 销售返利单的收款单处理逻辑，该 rule 的模式为 TCC 模式，会调用信用服务的 Sagas 的 Rule 来修改信用信息
 * @author : luysh@yonyou.com
 * @date : 2021/3/12
 */
@Component("financeRateBillSaveRule")
public class FinanceRateBillSaveRule extends AbstractCommonRule implements ITccRule {

    private static final String URI = "GT52131AT25.GT52131AT25.yts_rebate_bill";
    private static final String BILL_NUM = "d7456fdf";

    protected final static Logger logger = LoggerFactory.getLogger(FinanceRateBillSaveRule.class);

    /**
     * 执行 TCC 模式的 try 阶段的业务逻辑来锁定相关资源
     * @param billContext
     * @param map
     * @return
     * @throws Exception
     */
    @Override
    public RuleExecuteResult execute(BillContext billContext, Map<String, Object> map) throws Exception {

        logger.info("FinanceRateBillSaveRule start...");

        RuleExecuteResult result = new RuleExecuteResult();
        List<BizObject> bills = this.getBills(billContext, map);
        for (BizObject bizObject : bills) {
            dealRebateOrder(bizObject, result);
        }
        // 使用 YTS 的事务上下文记录需要回滚的数据
            YtsContext.setYtsContent("cancelGtxId", currentGtxId());
        return result;
    }
    private void dealRebateOrder(BizObject rebate, RuleExecuteResult result) {
```

```
        String gtxId = currentGtxId();
        BizObject bill = new BizObject();
        bill.put(F_AGENT, rebate.get(S_AGENT)); // 客户
        bill.put(F_BILL_DATE, rebate.get(S_VOCHE_DATE)); // 单据日期
        bill.put(F_EXCHANGE_RAGE, rebate.get(S_EXCHANGE_RAGE)); // 汇率
        bill.put(F_REBATE_MONEY, rebate.get(S_REBATE_MONEY)); // 返利金额
        bill.put(F_SAGE_ORG, rebate.get(S_SAGE_ORG)); // 销售组织
        bill.put(F_REBATE_NO, rebate.get(S_REBATE_NO)); // 返利单号
        bill.put(F_ORIGINAL_NAME, rebate.get(S_ORIGINAL_NAME)); // 原币种
        bill.put(F_DOMESTICK_NAME, rebate.get(S_DOMESTICK_NAME)); //本币币种
        bill.put(F_STATUS, "待确认");
        bill.put(F_IS_DELETED, 0);
        bill.put(F_GTX_ID, gtxId);
        bill.put(F_tenant_id, rebate.get(F_tenant_id));

        BillContext billContext = new BillContext();
        billContext.setBillnum(BILL_NUM);
        billContext.setFullname(URI);
        billContext.setAction(OperationTypeEnum.SAVE.getValue());
        billContext.setEntityCode(BILL_NUM);

        Map<String, Object> billMap = Maps.newHashMap();
        bill.setEntityStatus(EntityStatus.Insert);
        billMap.put("param", toBillDataDto(bill));

        try {
            BillBiz.executeRule(OperationTypeEnum.SAVE.getValue(), billContext, billMap);
        } catch (Exception e) {
            throw new BusinessException("创建收款单错误", e);
        }
        }

private BillDataDto toBillDataDto(BizObject bizObject) {
        BillDataDto dto = new BillDataDto();
        dto.setData(bizObject);
        return dto;
}

@Override
public RuleExecuteResult confirm(BillContext billContext, Map<String, Object> paramMap) throws Exception {
            // 从 yts 事务上下文获取需要回滚的全局事务 ID
```

```
        String gtxId = YtsContext.getYtsContext("cancelGtxId");
        logger.info("confirm start gtxId={},paramMap: {}", gtxId, toJSON(paramMap));
        RuleExecuteResult result = new RuleExecuteResult();
        QueryConditionGroup typegroup = QueryConditionGroup.and(
                QueryCondition.name("gtxId").eq(gtxId));
        QuerySchema query = QuerySchema.create()
                .addSelect(
                        "id, gtxId, billStatus,isDeleted")
                .addCondition(typegroup);
        List<BizObject> bills;
        try {
            bills = MetaDaoHelper.query(URI, query);
        } catch (Exception e) {
            throw new BusinessException("没有查询到需要 confirm 的单据", e);
        }
        if (CollectionUtils.isEmpty(bills)) {
            throw new BusinessException("没有查询到需要 confirm 的单据");
        }
        Map<String, Object> bill = bills.get(0);
        logger.info("rebatBill: {}", toJSON(bill));
        String status = (String)bill.get(F_STATUS);
        if ("已确认".equals(status)) {
            return result;
        } else {
            bill.put(F_STATUS, "已确认");
            bill.put("_status", EntityStatus.Update);
            update(URI, toJSON(bill));
        }
        return result;
    }
    /**
     * 处理回滚的逻辑
     */
    @Override
    public RuleExecuteResult cancel(BillContext billContext, Map<String, Object> paramMap) throws Exception {
        // 从 yts 事务上下文获取需要回滚的全局事务 ID
        String gtxId = YtsContext.getYtsContext("cancelGtxId");
        logger.info("cancel start gtxId={},paramMap: {}", gtxId, toJSON(paramMap));
        RuleExecuteResult result = new RuleExecuteResult();

        QueryConditionGroup typegroup = QueryConditionGroup.and(
```

```
                QueryCondition.name("gtxId").eq(gtxId));
        QuerySchema query = QuerySchema.create()
                .addSelect(
                        "id, gtxId, billStatus, isDeleted")
                .addCondition(typegroup);
        List<Map<String, Object>> bills = MetaDaoHelper.query(URI, query);
        if (CollectionUtils.isEmpty(bills)) {
            return result;
        }
        BizObject bill = toBizObject(bills.get(0));
        String status = bill.get(F_STATUS);
        if ("已确认".equals(status)) {
            throw new BusinessException("已经确认的单据不能回滚");
        }
        logger.info("rebatBill: {}", toJSON(bill));
        int isDeleted = bill.get(F_IS_DELETED);
        if (isDeleted == 1) {
            return result;
        } else {
            bill.put(F_IS_DELETED, 1);
            bill.put("_status", EntityStatus.Update);
            update(URI, toJSON(bill));
        }
        return result;
}

private BizObject toBizObject(Map<String, Object> data) {
        BizObject bizObject = new BizObject();
        bizObject.putAll(data);
        return bizObject;
}

public static String toJSON(Object o) {
        try {
            return JSON.toJSONString(o);
        } catch (Exception e) {
            return o.toString();
        }
}
```

（2）调用方注册 rule 时 config 的配置如下：

```
{"needDTC":true, "transactionMode":"tcc"}
```

5.2 RPC IRIS 框架下使用

5.2.1 开发服务接口

将对外提供的服务接口统一定义在一个项目中，单独打包为一个 XX-api.jar 包，方便修改和定义，如 yts-iris-pubapi，其中定义了对外服务接口及公共对象通过在对外服务接口方法上加上@YtsTransactional 注解即可加入或开启 YTS 事务，

该注解有三个参数

cancel：指定事务失败是的回滚方法，必须与执行业务方法签名一致，且在同一个接口上定义

confirm：在 tcc 或异步 SAGA 模式下提交业务方法，必须与执行业务方法签名一致，且在同一个接口上定义

mode：支持的事务模式，默认 sagas，可以指定 tcc 模式、asynsagas 模式

这里举个例子来说明，例子中由三个服务组成，其中 yts-webapi 模拟网关层，接入前端请求，在本地 service 层通过远程服务接口 IOrderService 调用 yts-iris-demo-a 的订单服务，yts-iris-demo-a 继续调用 yts-iris-demo-b 的远程服务接口 IMsPlaneService

5.2.2 sagas 模式接口定义举例如下：

1） yts-iris-demo-a 服务接口定义

```
@RemoteCall("yts-iris-demo-a@c87e2267-1001-4c70-bb2a-ab41f3b81aa3")
public interface IOrderService {
 //正向逻辑
    @ApiOperation(value = "下旅游订单", response = TourOrder.class)
    @YtsTransactional(cancel = "cancelOrder")
    public abstract TourOrder order(TourOrder paramTourOrder);

 //反向逻辑
 @ApiOperation(value = "取消旅游订单", response = TourOrder.class)
 public abstract TourOrder cancelOrder(TourOrder paramTourOrder);
}
```

2）yts-iris-demo-b 服务接口定义

```
@RemoteCall("yts-iris-demo-b@c87e2267-1001-4c70-bb2a-ab41f3b81aa3")
public interface IMsPlaneService {
    //正向逻辑
@ApiOperation(value = "预订机票", response = Void.class)
@YtsTransactional(cancel = "cancelPlane")
public abstract void orderPlane(PlaneOrder paramPlaneOrder);

    //反向逻辑
@ApiOperation(value = "取消机票预定", response = Void.class)
public abstract void cancelPlane(PlaneOrder paramPlaneOrder);
}
```

(5).tcc 模式接口定义如下

1） yts-iris-demo-a 服务接口定义

```
@RemoteCall("yts-iris-demo-a@c87e2267-1001-4c70-bb2a-ab41f3b81aa3")
public interface IOrderService {
//第一阶段逻辑
@YtsTransactional(cancel="cancelOrder",confirm="confirmOrder",mode = TransactionMode.RPCTCC)
public abstract TourOrder tccOrder(TourOrder paramTourOrder);

//第二阶段 confirm 逻辑
@ApiOperation(value = "提交旅游订单", response = TourOrder.class)
public abstract TourOrder confirmOrder(TourOrder paramTourOrder);

//第二阶段 cancel 逻辑
@ApiOperation(value = "取消旅游订单", response = TourOrder.class)
public abstract TourOrder cancelOrder(TourOrder paramTourOrder);
}
```

2）yts-iris-demo-b 服务接口定义

```
@RemoteCall("yts-iris-demo-b@c87e2267-1001-4c70-bb2a-ab41f3b81aa3")
public interface IMsPlaneService {
//第一阶段逻辑
```

```
@ApiOperation(value = "预订机票", response = Void.class)
@YtsTransactional(cancel="cancelPlane",confirm="confirmPlane", mode = TransactionMode.RPCTCC)
public abstract void tccOrderPlane(PlaneOrder paramPlaneOrder);

//第二阶段 confirm 逻辑
@ApiOperation(value = "提交机票预定", response = Void.class)
public abstract void confirmPlane(PlaneOrder paramPlaneOrder);

//第二阶段 cancel 逻辑
@ApiOperation(value = "取消机票预定", response = Void.class)
public abstract void cancelPlane(PlaneOrder paramPlaneOrder);
}
```

5.3 使用 RestFul 接口调用场景

5.3.1 服务调用方(httpclient)

5.3.1.1 依赖 yts-httpclient-support 的包(版本使用统一二方包的版本)

```
<dependency>
    <groupId>com.yonyou.cloud</groupId>
    <artifactId>yts-httpclient-support</artifactId>
</dependency>
```

5.3.1.2 修改基于 httpClient 的 sdk

为 sdk 增加 HttpClient 自动加载 HttpRequestInterceptor 的插件机制。

```
HttpClientBuilder httpClientBuilder = HttpClients.custom().setConnectionManager(this.httpPool).setDefaultRequestConfig(this.requestConfig);
try {
    ServiceLoader<HttpRequestInterceptor> requestInterceptors = ServiceLoader.load(HttpRequestInterceptor.class);
    requestInterceptors.forEach((x) -> {
        httpClientBuilder.addInterceptorFirst(x);
    });
    ServiceLoader<HttpResponseInterceptor> responseInterceptors = ServiceLoader.load(HttpResponseInterceptor.class);
    responseInterceptors.forEach((x) -> {
        httpClientBuilder.addInterceptorFirst(x);
```

```
        });
    } catch (Throwable e) {
        logger.error(e.getMessage(), e);
    }

    this.httpClient = httpClientBuilder.build();
```

5.3.1.3 项目中注入使用业务数据库连接池的 JdbcTemplate 的 bean

为 http 的调用开启 yts 的支持

```
header("ytsEnable", "true");
```

Sagas 模式

```
header("ytsMode", "sagas");
```

TCC 模式

```
header("ytsMode", "tcc");
```

5.3.1.4 调用 RESTful 接入 YTS 的代码示例

```
@Service("httpService")
@Transactional
public class HttpService {
    @Autowired
    YtsHttpClient ytsHttpClient;
    /**
     * 开启分布式事务接口 ytsEnable 有值即开启
     * @param uid
     * @param amount
     * @param orderId
     * @return
     */
    public Long tccSave(Long uid, Long amount, String orderId) {
        String insertSql = "insert into user_save_order (uid, amount,order_id,create_time,status,update_time) " +
                "values (?,?,?,?,?,?)";
        GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcTemplate.update(new PreparedStatementCreator() {
            @Override
            public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
```

```
                PreparedStatement ps = con.prepareStatement(insertSql, Statement.RETURN_GENERATED_KEYS);
                long nowTs = new Date().getTime();
                ps.setLong(1, uid);
                ps.setLong(2, amount);
                ps.setString(3, orderId);
                ps.setLong(4, nowTs);
                ps.setString(5, "save");
                ps.setLong(6, nowTs);
                return ps;
            }
        }, keyHolder);
        String url = ytsHttpUrl+ "/yts/tccSave?uid=" + uid + "&amount=" + amount;
        Map<String, List<String>> params = new HashMap<>();
        try {
            Map<String, List<String>> headerMap = new HashMap<>();
            headerMap.put("ytsEnable", Arrays.asList("true"));
            headerMap.put("ytsMode", Arrays.asList("tcc"));
            HttpResult httpRet = YtsSpringContextHolder.getBean(AuthSDKClient.class)
                    .execute(url, params, headerMap, EnumRequestType.GET);
            LOG.info("request url:{}, response:{}", url, httpRet.getResponseString());
            processHttpResult(httpRet);
        } catch (Exception e) {
            LOG.error("http request failed, url:{}", url, e);
            throw new RuntimeException(e);
        }
        return keyHolder.getKey().longValue();
    }
}
```

5.3.2 服务提供方

5.3.2.1 添加 YTS HTTP 框架的依赖包

```
<dependency>
    <groupId>com.yonyou.cloud</groupId>
    <artifactId>yts-http-springboot-support</artifactId>
</dependency>
```

5.3.3.2.2 为支持 yts 分布式事务的 controller 增加拦截器

```
@Configuration
public class YtsHttpConfig extends WebMvcConfigurationSupport {
```

```
    @Autowired
    private YtsHttpInterceptor ytsHttpInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        //注册 TestInterceptor 拦截器
        InterceptorRegistration registration = registry.addInterceptor(ytsHttpInterceptor);
        registration.addPathPatterns("/**");                    //所有路径都被拦截
        registration.excludePathPatterns(                       //添加不拦截路径
                "/**/*.html",           //html 静态资源
                "/**/*.js",             //js 静态资源
                "/**/*.css",            //css 静态资源
                "/**/*.woff",
                "/**/*.ttf"
        );
    }
}
```

* 如果项目以后拦截器配置则直接添加 YtsHttpInterceptor 即可，领域也可根据自己需要开启 YTS 的地址上添加拦截器。

5.3.2.2 注入 YTS 使用的持久化组件（JdbcTemplate）

项目中注入使用业务数据库连接池的 JdbcTemplate 的 bean

5.3.2.3 添加 YTS 分布式事务支持的注解并实现 confirm 以及 cancel 逻辑

```
@RestController
@RequestMapping("/yts")
public class YtsHttpController {


    @GetMapping("/save")
    // cancel 方法指定回滚调用的方法名，方法的参数需要和正向参数个数和类型一致
    @YtsTransactional(mode = TransactionMode.SAGAS, cancel="saveCancel")
    public String save(HttpServletRequest req, @RequestParam("uid") Long uid, @RequestParam("amount") Long amount) throws SQLException {
        String txId = getTxId(req);
        // 正向逻辑，为了方便进行回滚，将事务 ID 记录到业务的数据记录中
```

```
        ytsTransactionService.saveMoney(uid, amount, txId, false);
        return "充值成功";
    }

    public String saveCancel(HttpServletRequest req, @RequestParam("uid") Long uid, @RequestParam("amount")
Long amount) throws SQLException {
        String txId = getTxId(req);
        // 处理回滚逻辑
        try {
            ytsTransactionService.cancelSave(uid, amount, txId);
        } catch (Exception e) {
            throw new SQLException(e);
        }
        return "cancel success";
    }
}
```

* 正向和反向的业务操作如果有数据回滚则不能捕获异常，否则会导致数据不一致

5.3.2.4 业务逻辑实现示例

```
@Service
public class YtsTransactionService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional
    public boolean saveMoney(Long uid, Long amount, String txId, boolean isTcc) throws SQLException {
        String insertSql = "insert into user_amount_log (uid,create_time,amount,tx_id,action,status,update_time) " +
                "values (?,?,?,?,?,?,?)";
        long nowTs = new Date().getTime();
        Object[] params = new Object[] {uid, nowTs, amount, txId, "save", "正常", nowTs};
        int rows = jdbcTemplate.update(insertSql, params);
        if (rows <= 0) {
            throw new SQLException("日志插入失败");
        }
        long availAmount = amount;
        if (isTcc) {
            availAmount = 0;
```

```
        }
        rows = jdbcTemplate.update("update user_info set amount=amount+?,avail_amount=avail_amount+? where uid=?",
                amount, availAmount, uid);
        if (rows <= 0) {
            throw new SQLException("没有查询到充值的用户");
        }
        return true;
    }

    @Transactional
    public boolean cancelSave(Long uid, Long amount, String txId) throws SQLException {
        long nowTs = System.currentTimeMillis();
        int rows = jdbcTemplate.update("update user_amount_log set status='cancel',update_time=? " +
                        "where uid=? and amount=? and status='正常' and tx_id=?",
                nowTs, uid, amount, txId);
        if (rows > 0) {
            rows = jdbcTemplate.update("update user_info set amount=amount-?,avail_amount=avail_amount-? where uid=? and amount>=?",
                    amount, amount, uid, amount);
            if (rows <= 0) {
                throw new SQLException("用户余额不足不能回滚");
            }
        }
        return true;
    }

    @Transactional
    public boolean confirmMoney(Long uid, Long amount, String txId) throws SQLException {
        long nowTs = System.currentTimeMillis();
        int rows = jdbcTemplate.update("update user_amount_log set status='confirm',update_time=? " +
                        "where uid=? and amount=? and status='正常' and tx_id=?",
                nowTs, uid, amount, txId);
        if (rows > 0) {
            rows = jdbcTemplate.update("update user_info set avail_amount=avail_amount+? where uid=?",
                    amount, uid);
        }
        return true;
    }

    @Transactional
```

```
    public boolean tccCancelSave(Long uid, Long amount, String txId) throws SQLException {
        long nowTs = System.currentTimeMillis();
        int rows = jdbcTemplate.update("update user_amount_log set status='cancel',update_time=? " +
                                "where uid=? and amount=? and status='正常' and tx_id=?",
                nowTs, uid, amount, txId);
        if (rows > 0) {
            rows = jdbcTemplate.update("update user_info set amount=amount-? where uid=?",
                    amount, uid);
            if (rows <= 0) {
                throw new SQLException("用户余额不足不能回滚");
            }
        }
        return true;
}
```

第六章 开放接口说明

YTS 提供了再 YTS 开发时一些方便使用的 API，具体的功能如下：

6.1 冻结

接口名称：YtsContext.frozen(FrozenVo vo)

参数说明：vo，需要冻结的表单信息

用途：用于支持冻结该事务执行期间要修改的数据，确保其它事务无法同时修改

示例：

```
    FrozenVo vo= new FrozenVo();
    // 需要冻结的表名
vo.setTableName("biz_tourorder");
    // 需要冻结的表的主键字段
vo.setPkField("tourOrderId");
        // 主键值，可多个
Set<String> pks = new HashSet<String>();
pks.add(id);
vo.setPks(pks);
        // 冻结 API，在事务会成功提交或回滚前，单据会一直冻结中
YtsContext.frozen(vo);
```

特殊说明：在一次事务上下文里可以多次调用

6.2 绑定业务信息到事务

接口名称：YtsContext.bindBizIdToTransaction(String bizTable, String bizPk, Set<String> values)

参数说明：bizTable，当前事务业务操作的表

bizPk，当前事务业务操作表的主键字段

values，当前事务操作的表的主键字段值

用途：业务调用此接口将业务修改的业务数据传给 YTS 框架，便于问题排查处理，根据 YTS 的事务 ID 能够定位到该事务所修改的业务数据

示例：

```
Set<String> pkValues = new HashSet<String>();
pkValues.add(bizId);
pkValues.add(bizId + "_copy");
YtsContext.bindBizIdToTransaction("biz_tourorder", "tourOrderId", pkValues);
```

特殊说明：在一次事务上下文里可以多次调用

6.3 绑定参数到事务上下文

接口名称：YtsContext.setYtsContext(String key, Object param)

参数说明：key，参数对应的 key

param，参数对应的值

用途：在业务处理阶段将业务数据绑定到事务上下文，便于在第二阶段提交或回滚时使用参数进行回滚或提交

示例：

```
// 绑定业务数据到事务上下文，便于在第二阶段获取
String id = UUID.randomUUID().toString();
String id2 = UUID.randomUUID().toString();
YtsContext.setYtsContext("id", id);
YtsContext.setYtsContext("id2", id2);
```

特殊说明：在一次事务上下文里可以多次调用

6.4 将业务单据信息与 YTS 的事务关联

接口名称：YtsContext.setYtsBillInfo(YtsBillInfo ytsBillInfo)

参数说明：ytsBillInfo 业务单据信息

用途：在非标准 MDD 原数据，以及 iris，HTTP 框架中将业务单据的信息与 YTS 的事务信息进行关联，方便事务异常时可以从 YTS 的异常事务中定位业务单据，方便问题排查。

示例：

```
YtsBillInfo billInfo = new YtsBillInfo();
```

```
List<String> billIds = new ArrayList<>();
String orderId = "TourOrder-" + UUID.randomUUID().toString();
billIds.add(orderId);
billInfo.setBillIds(billIds);
List<String> billCodes = new ArrayList<>();
billCodes.add("code-" + orderId);
billInfo.setBillCodes(billCodes);
billInfo.setBillType(TourOrder.class.getSimpleName());
billInfo.setCreator("testUser");
billInfo.setFullName(TourOrder.class.getName());
billInfo.setTenantId("0000KPC165PABLPTS60000");

YtsContext.setYtsBillInfo(billInfo);
```

特殊说明：单据信息可以根据业务需求自己对各个属性赋值。