目录

简介

KDTX简介

KDTX目前支持注解使用方式。用户在需要引入分布式事务的方法上，增加一行注解界定事务边界，就可以轻松实现分布式事务。

注解使用方式

使用场景

假设有一个商品购买的业务场景，包含了 **下单** 和 **支付** 2个步骤，需要关联3个服务，每个服务有自己的DB： 1）**product服务**：库存管理，负责记录产品库存，当支付后，执行减库存操作； 2）**account服务**：账户管理，负责维护用户账户，当下单时，锁定账户金额；支付后，扣除账户金额； 3）**order服务**：订单管理，下单时，创建订单，关联产品和对应的账户信息；支付后，根据支付结果更新订单状态；

现在考虑 下单 和 支付 的服务调用： **下单**：查询产品信息（product） 锁定账户金额（account） 生成订单（order） **支付**：减库存（product） 扣除账户金额（account） 更新订单状态（order）

很显然，2个步骤都涉及多个模块的服务调用，需要保证调用过程中的事务性，因而需要使用全局事务。

此处通过springcloud调用示例说明kdtx全局事务接入的示例

创建事务分组

步骤一：在KDTX控制台页面，点击“事务分组”->“新建事务分组”,如图所示： 步骤二：根据实际需要进行事务分组的相关配置

引入依赖

使用maven构建项目时，需要引入kdtx客户端依赖： 首先，需要将kdtx的maven私服配置到项目的仓库列表中：

```
<repositories>
    <repository>
        <id>kdtx-nexus-public</id>
        <name>kdtx-nexus-releases</name>
        <url>http://120.92.116.210:8081/repository/maven-public/</url>
    </repository>
</repositories>
```

然后，在模块中引入kdtx的客户端starter：

```
<dependency>
    <groupId>com.ksyun.kdtx</groupId>
    <artifactId>kdtx-spring-starter</artifactId>
    <version>0.5.0</version>
</dependency>
```

创建kdtx客户端配置文件

kdtx客户端需要通过名为registry.conf的配置文件指定连接server服务的配置，registry.conf文件放在classpath根路径下；文件格式如下：

```
registry {
  # 指定服务的注册方式
  type = "file"

  # 指定对应的配置文件名称
  file {
    name = "kdtx-server.conf"
  }
}

config {
  # 指定服务的配置方式
  type = "file"

  # 指定对应的配置文件名称
  file {
    name = "kdtx-server.conf"
  }
}
```

对应kdtx-server.conf配置文件格式如下：

```
transport {
  # tcp udt unix-domain-socket
  type = "TCP"
  #NIO NATIVE
  server = "NIO"
  #enable heartbeat
  heartbeat = true
  #thread factory for netty
  thread-factory {
    boss-thread-prefix = "NettyBoss"
    worker-thread-prefix = "NettyServerNIOWorker"
    server-executor-thread-prefix = "NettyServerBizHandler"
    share-boss-worker = false
    client-selector-thread-prefix = "NettyClientSelector"
    client-selector-thread-size = 10
    client-worker-thread-prefix = "NettyClientWorkerThread"
    # netty boss thread size,will not be used for UDT
    boss-thread-size = 1
    #auto default pin or 8
    worker-thread-size = 32
  }
}

service {
  # 配置事务分组对应的服务器地址映射，vgroup_mapping后面接具体的事务组名称;
  vgroup_mapping.test_group_name = "kdtx-server"
  # 配置kdtx服务地址
  kdtx-server.grouplist = "ip:port"
  enableDegrade = false
  disable = false
}

client {
  async.commit.buffer.limit = 10000
  lock {
    retry.internal = 10
    retry.times = 30
  }
  report.retry.count = 5
}
```

在业务数据库中创建undo_log表

undo_log表用于在全局事务运行过程中，记录业务数据的变更记录，以便于在全局事务发生回滚时，恢复旧数据。需要在每个业务模块对应的数据库中创建undo_log表；建表语句如下：

```
CREATE TABLE `undo_log` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `branch_id` char(64) DEFAULT NULL,
  `xid` char(64) DEFAULT NULL,
  `rollback_info` longblob NOT NULL,
  `log_status` int(11) NOT NULL,
  `log_created` datetime NOT NULL,
  `log_modified` datetime NOT NULL,
  `ext` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_unionkey` (`xid`,`branch_id`)
);
```

配置数据源和事务分组

运行事务分组前，需要为服务配置对应的事务分组，以及对应的token，用于标记当前服务使用的是哪个事务分组，并进行权限检查。 在springcloud项目的yml配置文件中添加事务分组配置：

```
spring:
  # 数据源
  datasource:
    url: xxxx
username: xxxx
    password: xxxx

kdtx.spring.starter:
  # 事务分组名称
  txServiceGroup: "test_group_name"
  # 事务分组对应的token
  token: "xxxxxxxxxxxxxxxxxxxx"
```

注： 1）如果使用的是properties配置文件，则将上面配置修改为相应格式； 2）事务分组的名称和token可从金山云 分布式事务 事务分组 页面找到；

添加全局事务注解

实现具体的业务逻辑，并在需要开启全局事务的方法上添加@GlobalTransactional注解。比如：下单方法实现示意：

```
@GlobalTransactional
public PocOrder orderCreate(OrderReq req) throws BusinessException {

   // 查询商品，计算订单总价
   BaseResp<List<PocProductVo>> productResp =  productClient.getProducts( req.getProductId());
   if (productResp.getStatus() == 200) {
      if (productResp.getData() == null) {
         throw new BusinessException("不存在此商品");
      }
      if (productResp.getData().size() == 0) {
         throw new BusinessException("不存在此商品");
      }
      PocProductVo pocProductVo = productResp.getData().get(0);
      Integer orderAmount = pocProductVo.getProductPrice() * req.getProductNum();
      if (pocProductVo.getStoreNum() < req.getProductNum()) {
         throw new BusinessException("库存不足无法进行下单");
      }

      // 查询用户账户，查看余额是否够用，如果够用将订单额冻结
      OrderLockAmountReq orderLockAmountReq = new OrderLockAmountReq();
      orderLockAmountReq.setTenantId(req.getTenantId());
      orderLockAmountReq.setUserId(req.getUserId());
      orderLockAmountReq.setAmount(orderAmount);
      BaseResp<OrderLockAmountResp> accResp = accountClient.lockAmount(orderLockAmountReq);
      if (accResp.getStatus().intValue() == 666) {
         throw new BusinessException(accResp.getMsg());
      } else if (accResp.getStatus().intValue() == 200){
         PocOrder pocOrder = new PocOrder();
         pocOrder.setOrderId(Integer.valueOf(idGeneratorClient.getSegmentId(ORDER_ID)));
         pocOrder.setTenantId(req.getTenantId());
         pocOrder.setUserId(req.getUserId());
         pocOrder.setProductId(req.getProductId());
         pocOrder.setProductNum(req.getProductNum());
         pocOrder.setAmount(orderAmount);
         pocOrder.setStatus("0");
         pocOrder.setCreateTime(TimeUtils.dateToTime());
         pocOrder.setModifyTime(TimeUtils.dateToTime());
         orderDaoService.insertOrder(pocOrder);

         stringRedisTemplate.opsForValue().set(ORDER_ID_REDIS_HASH + pocOrder.getOrderId(),
               gson.toJson(pocOrder));
         stringRedisTemplate.expire(ORDER_ID_REDIS_HASH + pocOrder.getOrderId(), 30000, TimeUnit.SECONDS);
         return pocOrder;
      } else {
         throw new BusinessException("账户服务未知报错");
      }
   } else {
      throw new BusinessException(productResp.getMsg());
   }
}
```

支付方法实现示意：

```
@GlobalTransactional
public void payTheBill(PayReq req) throws BusinessException {
   PocOrder pocOrder = orderDaoService.selectOrderByPrimaryKey(req.getOrderId());
   if (pocOrder.getStatus().equals("1")) {
      throw new BusinessException("此订单已经完成结算");
   }
   if (pocOrder.getStatus().equals("3")) {
      throw new BusinessException("此订单已经取消");
   }

   // 产品减库存
   ReduceProductStoreNumReq reduceReq= new ReduceProductStoreNumReq();
   reduceReq.setProductNum(pocOrder.getProductNum());
   // 产品减少库存，如果分布式事务进行回滚的话这里库存会回滚
   BaseResp<String> prodResp = productClient.reduceProductStoreNum(reduceReq,pocOrder.getProductId());
   if (prodResp.getStatus().intValue() == 200) {
      PayAmountReq payAmountReq = new PayAmountReq();
      payAmountReq.setTenantId(pocOrder.getTenantId());
      payAmountReq.setUserId(pocOrder.getUserId());
      payAmountReq.setAmount(pocOrder.getAmount());
      // 设置回滚位
      payAmountReq.setB(req.getB());
      // 账户减冻结金额
      BaseResp<String> payResp = accountClient.payAmount(payAmountReq);
      if (payResp.getStatus().intValue() == 200) {
         pocOrder.setModifyTime(TimeUtils.dateToTime());
         pocOrder.setStatus("1");
         orderDaoService.updateByPk(pocOrder);
         stringRedisTemplate.opsForValue().set(ORDER_ID_REDIS_HASH + pocOrder.getOrderId(), gson.toJson(po
cOrder));
         stringRedisTemplate.expire(ORDER_ID_REDIS_HASH + pocOrder.getOrderId(), 30000, TimeUnit.SECONDS);
      } else {
         throw new BusinessException(payResp.getMsg());
      }
   } else {
      throw new BusinessException(prodResp.getMsg());
   }

}
```

配置完成后，运行服务，当用户通过HTTP接口调用到标记注解的方法时，即可开启全局事务功能。