

Table of Contents

Introduction	1.1
0.概述	1.2
1.快速开始	1.3
1.1 常规部署	1.3.1
1.2 docker image部署	1.3.2
1.3 docker compose部署	1.3.3
2.适用场景	1.4
3.核心功能	1.5
3.1 服务端功能	1.5.1
3.1.1 定时任务	1.5.1.1
3.1.2 事务管理	1.5.1.2
3.1.3 事务协调	1.5.1.3
3.1.4 异常上报	1.5.1.4
3.1.5 消息上报	1.5.1.5
3.1.6 注册中心	1.5.1.6
3.2 客户端功能	1.5.2
3.2.1 业务拦截	1.5.2.1
3.2.2 事务关联	1.5.2.2
3.2.3 事件上报	1.5.2.3
3.2.4 负载均衡	1.5.2.4
3.2.5 补偿重试	1.5.2.5
3.3 事务支持功能	1.5.3
3.3.1 手动补偿	1.5.3.1
3.3.2 自动补偿	1.5.3.2
3.3.3 重试	1.5.3.3
3.3.4 超时	1.5.3.4
3.3.5 嵌套	1.5.3.5
3.4 服务降级	1.5.4
3.5 监控	1.5.5
4.配置说明	1.6
4.1 启动前配置	1.6.1
4.2 运行中配置	1.6.2
5.常见问题	1.7
6.产品优势	1.8

7.示例工程	1.9
7.1 Spring Boot示例工程	1.9.1
7.2 Spring Cloud示例工程	1.9.2
7.3 Dubbo示例工程	1.9.3
8.测试	1.10
8.1 性能测试	1.10.1

txle 中文技术参考手册

目录

参考 gitbook 左侧目录区 或 [SUMMARY.md](#)

PDF下载

[《txle中文技术参考手册.pdf》](#)

官方技术支持

- 代码库 github: github.com/actiontech/txle
- 文档库 github: github.com/actiontech/txle-docs-cn
- 文档库 github pages: actiontech.github.io/txle-docs-cn
- 网站: [TXLE官方网站](#)
- QQ group: 696990638
- 开源社区微信公众号



注意

本分支上的手册适用于txle 2.19.10.0版，其他版本的文档请参考对应tag分支或者release版文档。

联系我们

如果想获得txle 的商业支持, 您可以联系我们:

- 全国支持: 400-820-6580

- 华北地区: 86-13718877200, 王先生
- 华南地区: 86-18503063188, 曹先生
- 华东地区: 86-18930110869, 梁先生
- 西南地区: 86-18328335660, 雷先生

0.1 txle 简介与整体架构

0.1.1 txle简介

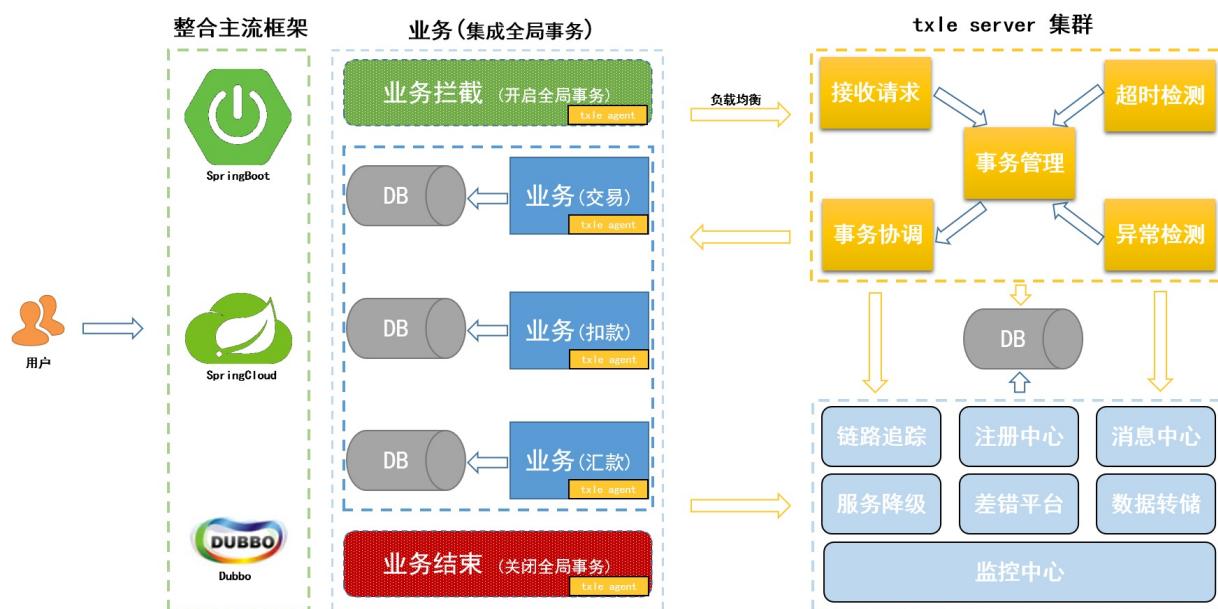
txle是[上海爱可生信息技术股份有限公司](#)基于开源项目ServiceComb Pack研发的分布式事务开源框架，核心功能是保证业务数据的最终一致性，存在以下几个优势特性：

- 数据最终一致性** 本框架不仅继承了ServiceComb Pack的手动补偿机制，还增设了自动补偿机制，方便业务通过多种方式实现数据最终一致性。
- 高性能** 各子业务均直接执行数据事务提交，而非像XA、TCC等需要锁定数据源。
- 低侵入** 最少情况两个注解即可。
- 支持重试和超时机制** 部分相关框架未能支持重试或超时。
- 支持异常快照处理** 当子事务发生异常时，且未能成功补偿情况下，系统会自动收集相应的异常信息，生成快照，供后续排查分析（如存储本地数据库或上报至差错平台）。
- 支持集成主流框架** 如Spring Boot、Spring Cloud和Dubbo。
- 完善的监控中心** 系统在许多层面进行了重要指标的监控，如服务端、客户端，甚至业务端。
- 最大化保证主营业务的正常运行** 系统提供全局事务的整体与部分降级功能，以及降级后的恢复功能。

0.1.2 txle由来

- txle是基于ServiceComb Pack进行研发的。首先，我们要感谢ServiceComb Pack项目的所有贡献者。
- 对于txle而言，我们更加专注于金融领域，可适应诸多的复杂业务场景。另外，在稳定性和高性能方面有显著改善，尤其在性能上，我们提升了几倍的QPS。

0.1.3 txle内部架构



1.快速开始

- [1.1 常规部署](#)
- [1.2 docker image部署](#)
- [1.3 docker compose部署](#)

1.1 快速开始

1.1.1 关于本节

- 本节内容为您介绍如何使用txle安装包快速部署并启动一个txle服务，并简单了解txle的使用和管理

1.1.2 安装准备

以下部分将被需要作为txle服务端启动的基础支撑

- 一个启动的MySQL实例

txle服务端是通过连接数据库实例来进行数据的存储，所以请至少准备一个正在运行的实例。在此仅以一个最简单的情况举例，通过Docker运行MySQL

```
$ docker run -itd -p 3306:3306 --name=backend-mysql -e MYSQL_ROOT_PASSWORD=123456 -e MYSQL_DATABASE=txle mysql:5.7 --character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
```

- 一个启动的consul集群

txle服务端在集群部署场景下，每个服务都会执行各自的定时器任务，由于定时器任务中带有更新操作，故多定时器同时执行时会带来一定的隐患。consul集群可自动选出一位TXLE的leader来执行特殊的定时器任务，若当前leader宕机，则再自动选出集群中另一个服务为leader。

```
$ docker run -d -p 8500:8500 --name=consul_1 --add-host ${txle_server_hostname}:${host_address} consul agent -server -ui -node=1 -client=0.0.0.0 -bootstrap
```

- JVM环境

txle是使用java开发的，所以需要启动txle您先需要在机器上安装java版本1.8或以上，并且确保JAVA_HOME参数被正确的设置

1.1.3 下载并安装txle服务端

- 通过此连接下载最新版本的安装包[https://github.com/actiontech/txle/releases/download/v\\${version}/actiontech-txle-\\${version}.tar.gz](https://github.com/actiontech/txle/releases/download/v${version}/actiontech-txle-${version}.tar.gz)
- 解压并安装txle服务端到指定文件夹中

```
$ mkdir -p ${working_dir}
$ wget -P ${working_dir} https://github.com/actiontech/txle/releases/download/v2.19.1.0.0/actiontech-txle-2.19.10.0.tar.gz
$ tar -zxvf ${working_dir}/actiontech-txle-${version}.tar.gz -C ${working_dir}
```

1.1.4 txle服务端的初始化配置

- 根据当前环境修改配置文件 \${working_dir}/actiontech-txle-\${version}/conf/application.yaml

```
```yaml ...
alpha: server: host: ${host_address} port: 8080
...
spring: profiles: mysql datasource: username: root password: 123456 url:
jdbc:mysql://${mysql_host_address}/txle?useSSL=false
...
cloud: consul: enabled: true # Consul must be enabled in a production environment. host:
${consul_host_address} port: 8500
```

```
1.1.5 启动txle服务端
+ 启动命令
```bash
$ ${working_dir}/actiontech-txle-${version}/bin/txle start
```

- 如果启动失败请使用此命令查看失败的详细原因

```
$ tail -f ${working_dir}/actiontech-txle-${version}/log/stdout.log
```

1.1.6 配置txle客户端

- 根据当前环境修改配置文件，这里以spring boot为例

```
```yaml
spring: application: name: ${application.name}
...
alpha: cluster: address: ${alpha.cluster.addresses}```
```

## 1.1.7 常见问题

- 启动txle服务端启动时报 Name or service not known 错误

解决方法： echo "127.0.0.1: \${txle\_server\_hostname} >> /etc/hosts"

## 1.2 快速开始(docker)

### 1.2.1 关于本节

- 本节内容为您介绍如何通过dockerhub上的txle镜像快速启动一个txle的demon

### 1.2.2 安装准备

- 安装docker

### 1.2.3 安装过程

按照顺序依次执行以下docker命令：

- 创建网络

```
$ docker network create --subnet=172.20.0.0/24 --gateway=172.20.0.254 txle_net
```

- 运行consul

```
$ docker volume create consul_data
$ docker run -d --name=consul -v consul_data:/consul/data -p 8500:8500 --network=txle_net --ip=172.20.0.1 consul agent -server -ui -node=1 -client=0.0.0.0 -bootstrap
```

- 运行MySQL

```
$ docker run -d --name=backend-mysql -p 3306:3306 --network=txle_net --ip=172.20.0.11 -e MYSQL_ROOT_PASSWORD=123456 -e MYSQL_DATABASE=txle mysql:5.7 --character-set-server=utf8mb4 --collation-server=utf8mb4_unicode_ci
```

- 运行txle server

```
$ docker run -itd --name=txle-server-1 -p 8090:8090 --network=txle_net --ip=172.20.0.21 actiontech/txle-server
```

- 运行spring boot demo

```
$ docker run -itd --name=global -p 8000:8000 --network=txle_net --ip=172.20.0.31 actiontech/txle-global
$ docker run -itd --name=transfer --network=txle_net --ip=172.20.0.32 actiontech/txle-transfer
$ docker run -itd --name=user --network=txle_net --ip=172.20.0.33 actiontech/txle-user
$ docker run -itd --name=merchant --network=txle_net --ip=172.20.0.34 actiontech/txle-merchant
```

### 1.2.4 使用工具发起全局事务

1. 发起转账1元，转账将会成功

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/1/1
$ ok
```

2. 访问数据库查看账户余额

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
| | | | | | | | |
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

`txle_sample_transfer` 记录了一条交易成功的记录，`txle_sample_user` 和 `txle_sample_merchant` 数据正常记录

3. 根据示例代码里的设定发起转账金额大于200元，第二个子服务merchant会报错导致转账将会失败

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/300/1
$ {"timestamp":1572429018848,"status":500,"error":"Internal Server Error","exception":"org.springframework.web.client.HttpServerErrorException","message":"500 null","path":"/testGlobalTransaction/1/300/1"}
```

4. 访问数据库查看账户余额

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
| 2 | 1 | 1 | 300.00000 | 1 | 2 | 0 | 2019-10-30 09:47:18 |
+----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```

txle\_sample\_transfer 记录了一条交易失败的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据回滚

## 1.3 快速开始(docker-compose)

### 1.3.1 关于本节

- 本节内容为您介绍如何通过txle的docker-compose文件来启动txle服务端和业务demo

### 1.3.2 安装准备

- 安装docker
- 安装docker-compose

### 1.3.3 安装过程

从txle项目中下载最新的docker-compose.yml文件

<https://github.com/actiontech/txle/blob/master/docker-images/docker-compose.yml>

使用 `docker-compose up -d` 命令直接启动txle，compose配置文件会从dockerhub拉取镜像并最终启动txle服务端和业务demo

### 1.3.4 使用工具发起全局事务

1. 发起转账1元，转账将会成功

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/1/1
$ ok
```

2. 访问数据库查看账户余额

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime |
|----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
|----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
|----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

txle\_sample\_transfer 记录了一条交易成功的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据正常记录

3. 根据示例代码里的设定发起转账金额大于200元，第二个子服务merchant会报错导致转账将会失败

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/300/1
$ {"timestamp":1572429018848,"status":500,"error":"Internal Server Error","exception":"org.springframework.web.client.HttpServerErrorException","message":"500 null","path":"/testGlobalTransaction/1/300/1"}
```

4. 访问数据库查看账户余额

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:4
7:54 |
| 2 | 1 | 1 | 300.00000 | 1 | 2 | 0 | 2019-10-30 09:5
0:18 |
+----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```

txle\_sample\_transfer 记录了一条交易失败的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据回滚

## 适用场景

### 不适用场景

此处先列举两个不适用场景，除此之外，适用其他绝大多数业务场景。

#### 数据实时性要求较高的业务场景

txle核心是解决分布式架构中的数据最终一致性，所以在数据实时一致性方面有所延迟。

#### 唯一补偿失败的业务场景：并发 + 热点数据 + 异常 + 中间笔业务 + 非计算型赋值

当然，这里仅分析业务场景，txle框架系统、网络或硬件等故障，这里暂不做讨论。

补偿失败的业务场景需要同时满足**五大因素**：并发 + 热点数据 + 异常 + 中间笔业务 + 非计算型赋值，该五大因素同时出现情况，会导致补偿失败，也是目前唯一导致补偿失败的业务场景。当然，补偿失败会有上报差错平台以及本地存储分析的等补救措施。

对于一般业务系统而言，同时满足以上五大因素，相信概率不是特别大，故在此业务场景概率不大的情况下，也是相对适合的。

## 适用复杂场景

#### 并发 + 热点数据 + 异常 + 最后一笔业务

当热点数据业务运行异常时，若发生异常的业务为该热点数据补偿时刻的最后一笔业务，则可以被正确地补偿成功。

示例：用户A在某店铺下单了某件商品，与此同时，用户B也在同店铺下单了同件商品

并发场景之一：

- 用户A下单成功
- 用户B下单成功
- 用户A扣减库存成功
- 用户B库存扣减失败

最终，txle对用户B的订单进行补偿操作，由于A在B之前下单，所以本次补偿完全不受A订单的影响，因此可以被成功地补偿，保证了数据的最终一致性。

#### 并发 + 热点数据 + 异常 + 中间笔业务 + 计算型赋值

当热点数据业务运行异常时，若发生异常的业务为该热点数据补偿时刻的中间笔业务，理论上讲会收到后续业务的影响，但若被补偿的热点数据为计算型赋值的数据，则也可以被正确地补偿成功。

示例：用户A和用户B同时订购相同火车票，该票目前余票总数假设为10张，且无他人购此票情况下

并发场景之一：

- 用户A订票成功，但未支付

- 余票总数扣减成功，即SQL(简化版):  $10-1=9$ 张余票
- 用户B订票成功，且已支付
- 余票总数扣减成功，即SQL:  $9-1=8$ 张余票
- 用户A在指定支付时间内未支付，故余票总数需被再加回一张，即SQL:  $8+1=9$ 张余票

余票总数在数据库中以数值类型存储，即可通过计算公式对其赋值，因此即使中间笔业务出现异常，也可以被正确地补偿成功，保证了数据的最终一致性。

## 其它适用场景

上面介绍了几种较为复杂的业务场景，可以看出，除对实时性要求高和需满足五大因素且高概率的业务场景不太适合，其它业务场景都非常适用。再次强调，如果业务中含有满足五大因素的可能性，但概率不高，也是值得推荐的。

## 总结

- 适用于数据实时性要求不高的绝大多数并发业务场景
- 适用于热点数据的大多数并发业务场景
- 适用于热点数据的绝大多数非并发业务场景
- 适用于非热点数据的绝大多数并发业务场景
- 适用于热点数据的绝大多数非并发业务场景

	并发业务	非并发业务
热点数据		
非热点数据		

## 3.核心功能

- 3.1 服务端
  - 3.1.1 定时任务
  - 3.1.2 全局事务管理
  - 3.1.3 全局事务协调
  - 3.1.4 异常上报
  - 3.1.5 消息上报
  - 3.1.6 注册中心
- 3.2 客户端
  - 3.2.1 业务拦截
  - 3.2.2 全局事务关联
  - 3.2.3 事件上报
  - 3.2.4 负载均衡
- 3.3 全局事务
  - 3.3.1 手动补偿
  - 3.3.2 自动补偿
  - 3.3.3 重试
  - 3.3.4 超时
  - 3.3.5 嵌套
- 3.4 服务降级
- 3.5 监控

## 3.1.服务端功能

- [3.1.1 定时任务](#)
- [3.1.2 全局事务管理](#)
- [3.1.3 全局事务协调](#)
- [3.1.4 异常上报](#)
- [3.1.5 消息上报](#)

## 服务端定时任务扫描器

- 更新超时状态

查询出【已完成但未标记DONE状态的】超时记录；

更新超时记录为完成状态；

- 查找超时事件

查询(未完成的)超时事件；

保存超时记录；

- 终止超时时间

查询【NEW状态】且【已超时的】超时记录，未超时的超时记录将不做处理；

更新【已超时的】超时记录为【PENDING】状态；

- 查找需补偿事件，保存未补偿命令

查询需补偿事件(即超时异常的和普通异常的)按照从左到右的顺序，优先排除过滤较多的条件；从未结束(即无【SagaEndedEvent】状态)的最小事件开始查询；含发生异常的；排除已补偿的；排除重试的；依据唯一标识升序；

保存未补偿命令；

- 执行补偿

查找状态为NEW的Command；

更新状态为NEW的Command的状态为PENDING；

调用客户端的补偿方法；

- 更新已补偿命令

查找已补偿但未结束的事件；

更新补偿状态为DONE；

## 事务管理

- 对全局事务及子事务进行新增、修改、删除和查询的相关基本操作。
- 支持子事务重试操作。
- 支持全局事务或子事务的超时。
- 全局事务或子事务在程序中以事件的形式存在。

## 事务协调

- 当全局事务执行失败时，事务协调器(txle server)会主动对该全局事务内的已完成子事务下达补偿命令，同时对该子事务标记为已补偿。
- 事务协调包含手动补偿和自动补偿。
- 若补偿或回滚失败，被协调客户端会收集异常信息，上报至差错平台并保存至数据库。

## 上报异常至差错处理平台

- 简介

分布式事务主要保证数据的一致性，如果业务出现异常情况，则会回滚掉所有执行过的子事务。然而，一些异常是系统无法抗拒与处理的(如数据被篡改)，会导致全局事务回滚失败，此时系统会将事务信息上报至差错平台，供相关人员查阅及制定处理方案。

- 配置

```
application.yml
```

```
txle:
 accident:
 platform:
 address:
 api: http://ip:port/receiveFailedGlobalTxInfo
```

```
application.properties
```

```
txle.accident.platform.address.api=http://ip:port/receiveFailedGlobalTxInfo
```

### 重试次数

```
application.yml
```

```
txle:
 accident:
 platform:
 retry:
 retries: 3
```

### 重试间隔(秒)

```
application.yml
```

```
txle:
 accident:
 platform:
 retry:
 interval: 3
```

- 上报时机

自动补偿失败时，系统将上报相关信息至差错平台。

手动补偿失败时，由于手动补偿接口由业务人员自行开发维护，故业务人员在编写手动补偿接口时需自行追加上报至差错平台相关代码，参考如下。

如图：

```
application.yml

@.Autowired
private ClientAccidentHandlingService clientAccidentHandlingService;

@Autowired(required = false)
private OmegaContext omegaContext;

public xx xxx_rollback(..) {
 try {
 // 补偿接口的业务代码....
 } catch (Exception e) {
 // 补偿接口失败时，上报差错平台
 JsonObject jsonParams = new JsonObject();
 jsonParams.addProperty("type", AccidentHandleType.ROLLBACK_ERROR.toInteger());
 jsonParams.addProperty("globaltxid", omegaContext.globalTxId());
 jsonParams.addProperty("localtxid", omegaContext.localTxId());

 // bizinfo由业务人员自行收集，此处为JSON格式的实体信息
 MerchantEntity merchantEntity = merchantRepository.findOne(merchantid);
 if (merchantEntity != null) {
 jsonParams.addProperty("bizinfo", merchantEntity.toJsonString());
 } else {
 jsonParams.addProperty("bizinfo", "{\"merchantid\": " + merchantid + "}");
 }

 LOG.error("Failed to execute method 'updateBalanceById_rollback', params [{}].", jsonParams.toString(), e);
 // 调用上报至差错平台的接口
 clientAccidentHandlingService.reportMsgToAccidentPlatform(jsonParams.toString());
 throw e;
 }
}
```

- 监控指标

- 上报成功数量指标 - txle\_report\_accident\_successful\_total

记录成功上报至差错平台的数量。

- 上报失败数量指标 - txle\_report\_accident\_failed\_total

记录上报至差错平台失败的数量。

- 告警

每次抓取到新的失败记录，将触发告警。

告警规则配置，在alert.rules文件中追加如下代码：

```
上报失败告警规则

Alert for txle instance that metric txle_report_accident_failed_total value > 0
- alert: ReportAccidentUnsuccessfully(init)
 expr: sum(count_over_time(txle_report_accident_failed_total[1m]) + txle_report_accident_failed_total) == 2
 for: 0s
 annotations:
 summary: "Failed to report accident on {{ $labels.instance }}.""

Alert for txle instance that metric txle_report_accident_failed_total value > 0
- alert: ReportAccidentUnsuccessfully
 expr: idelta(txle_report_accident_failed_total[1m]) > 0
 for: 0s
 annotations:
 summary: "Failed to report accident on {{ $labels.instance }}."
```

## 上报消息至Kafka平台

- 简介

因子业务无法确定全局事务是否成功完成，即数据是否最终达到一致性，故各子事务不能直接去刷新缓存数据。因此，需等待全局事务完成(无论成功失败)后，将全局事务内部所有子事务执行后影响的数据信息上报到消息管理平台(即Kafka)，由各子事务自行订阅刷新缓存。

- 消息数据表

字段	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		1
globaltxid	varchar	36	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
localtxid	varchar	36	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
status	int	1	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
version	int	2	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		
dbdrivername	varchar	100	0	<input type="checkbox"/>	<input type="checkbox"/>		
dburl	varchar	100	0	<input type="checkbox"/>	<input type="checkbox"/>		
dbusername	varchar	20	0	<input type="checkbox"/>	<input type="checkbox"/>		
tablename	varchar	255	0	<input type="checkbox"/>	<input type="checkbox"/>		
operation	varchar	20	0	<input type="checkbox"/>	<input type="checkbox"/>		
ids	blob	0	0	<input type="checkbox"/>	<input type="checkbox"/>		
createtime	datetime	0	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>		

- 集成逻辑

- 收集数据信息

即数据表Message中有关业务上的字段值信息，如数据库驱动、数据库连接地址、数据库用户名、数据表名称、编辑操作类型、影响数据标识集合。

- 自动补偿场景收集

生成自动补偿信息过程中，收集子事务所影响的业务数据。

- 手动补偿场景收集

因子事务的业务逻辑未知，可能存在数据库方面影响的数据与业务上影响数据不一致，所以应由各子事务自行实现收集所影响的业务数据。

手动补偿情况业务开发人员调用上报Kafka消息的示例代码：

```

@Autowired
private MessageSender messageSender;

public ... rollback... {
 // 业务代码.....

 // 手动补偿场景：由业务人员自行收集所影响的数据信息
 messageSender.reportMessageToServer(new KafkaMessage(...));
}

```

- 调用统一上报接口

```
org.apache.servicecomb.saga.omega.transaction.MessageSender#
reportMessageToServer(org.apache.servicecomb.saga.omega.transaction.KafkaMessage m
essage);
```

- 上报逻辑

采用gRPC方式进行上报

新增报文消息的proto文件GrpcMessage.proto

逻辑与客户端传输Event到服务端一致，并与其共用同一Endpoint

- TXLE服务端存储业务信息至数据库

即将客户端收集到的业务影响数据保存到数据表Message中。

- TXLE服务端发送业务信息至Kafka服务端

当全局事务结束后，读取数据表Message中相关消息列表，将整个全局事务的消息列表一次性发送至Kafka服务端。

- 发送成功

更新数据表Message中的消息状态status为发送成功。

记录日志。

- 发送失败

更新数据表Message中的消息状态status为发送失败。

记录日志。

上报差错平台。

- 配置文件与热加载

- 配置文件：..\\txle\\alpha\\alpha-server\\src\\main\\resources\\kafka.properties。

ps: 在测试与生产环境使用前，请先依据当时环境对此文件进行调整，如bootstrap.servers、topic、acks等。

- 默认不启用Kafka

可在启动服务前通过修改配置文件开启Kafka。

```
data-mysql.sql ×
1 /**
2 * Copyright (c) 2018-2019 ActionTech.
3 * License: http://www.apache.org/licenses/LICENSE-2.0 Apache License 2.0 or higher.
4 */
5
6 -- *** All words should be lower case. ***
7 -- Initialize configurations
8 INSERT INTO Config VALUES (1, null, null, null, 12, 0, 1, '1', '历史表间隔规则。值：0-日，1-月，2-季，3-年。注：不转储10天内的数据。', now());
9 INSERT INTO Config VALUES (2, null, null, null, 4, 0, 1, 'disabled', '默认关闭，避免因为未及时配置Kafka服务地址，导致每次请求时需花费60秒尝试发送Kafka消息。', now());
10 若想在启动前开启Kafka，则设置为enabled.
```

当然，也可以通过txle-ui服务进行配置。

- 热加载

可调用RESTful Api接口：<http://ip:port/reloadConfig/kafka>，对kafka.properties文件进行热加载。

# 注册中心

- 简介

为实现txle服务端的主从节点部署模式，系统引入注册中心对txle服务集群进行管理，包括服务注册、心跳检测以及领导选举等工作。

- 配置

```
spring:
 cloud:
 consul:
 enabled: true
 # host: 127.0.0.1
 # port: 8500
 servers: ip:port,ip:port,ip:port
 discovery:
 enabled: true
 register: true
 serviceName: ${spring.application.name}
 healthCheckPath: /health
 healthCheckInterval: 10s
 instanceId: ${spring.application.name}-${alpha.server.host}-${random.value}
 tags: txle-server-host=${alpha.server.host},txle-server-port=${alpha.server.p
 ort}
```

- 参数**spring.cloud.servers**

**servers**参数为自定义参数，主要用于配置注册中心服务集群下的多个节点地址。

- 其它参数

不建议修改。

- 参数**spring.cloud.enabled**

- 默认为**true**，代表启用注册中心，启用后，在发生异常时，支持补偿功能，以保证数据的准确性。

**提示：生产环境必须启用。**

- 支持设置为**false**，即不启用注册中心，但在发生异常时，不提供补偿功能，不保证数据的准确性。主要用于业务整合txle的整体流程调试场景使用，非正式环境使用。

- 设计思想：提供完美功能，或不提供，杜绝瑕疵功能。

## 3.2.客户端功能

- [3.2.1 业务拦截](#)
- [3.2.2 全局事务关联](#)
- [3.2.3 事件上报](#)
- [3.2.4 负载均衡](#)
- [3.2.5 补偿重试](#)

## 业务拦截

### 业务发起拦截

- 业务系统集成txle的关键一步是业务发起方法上添加注解@SagaStart，代表该方法使用全局事务。
- txle将会对带有@SagaStart注解的所有方法进行拦截。
- 拦截业务后，开启全局事务，并将其交由事务管理器进行管理
- 接着，执行业务发起函数中的相关子业务程序。

ps：开关全局事务/子事务均需交由事务管理器进行管理。

### 子业务拦截

- 业务系统集成txle关键的第二步是在子业务方法上添加注解@Compensable或@AutoCompensable，代表该子业务方法使用全局事务的手动补偿或自动补偿功能。
- txle将会对带有@Compensable或@AutoCompensable注解的所有方法进行拦截。
- 拦截子业务方法后，开启全局事务下的子事务，并与当前全局事务进行关联。
- 接着，执行子业务中的相关程序(如操作数据库等)。
- 每个子业务执行完成后，需关闭当前子业务对应的子事务。
- 待所有子业务均执行完成后，关闭全局事务，整个全局事务视为完成。

### SQL拦截

全局事务除了对业务方法进行拦截，还会对业务执行的SQL进行拦截，但不会影响业务SQL的正常执行，主要是为了实现自动补偿机制，以及统计SQL的耗时，便于系统性能排查。

## 全局事务与各子事务的关联

- 全局事务与各子事务是通过全局事务标识xid进行关联的。
- 全局事务发起时会创建当前全局事务的上下文信息，含xid。
- 调用子事务时，将xid隐藏传输至子事务中，实现关联。

可参考调用链路时序图：



## 上报事件

业务集成全局事务后，全局事务客户端的主要工作就是负责拦截与收集业务信息，随后形成对应的事件，通过RPC传输至服务端。

事件形成时机：

全局事务启动事件：业务发起时产生

子事务启动事件：执行被全局事务管理的子业务时产生

子事务结束事件：被全局事务管理的子业务执行完成后产生

异常中止事件：全局事务内出现异常或超时情况产生

补偿事件：全局事务内出现异常或超时，对子事务进行补偿时产生

全局事务结束事件：业务结束后产生

## 负载均衡

为防止大量并发集中到某一台服务器上而出现性能下降甚至服务器宕机等隐患，全局事务提供负载均衡功能。

服务端的集群节点支持无限水平扩展，客户端支持配置多个服务端地址，以将请求分散到多台服务。

目前采用的负载均衡算法是最快响应时间算法，即每一次请求完成后会记录响应时间，哪台服务器的响应时间短，则下次就请求到就使用哪台服务器。

PS：目前算法存在一定的缺陷，即某一时间段大量并发请求可能会都被请求到上次响应时间最快的服务器上，该问题正在处理中，待处理后会更新此文档。

## 补偿重试

### 简介

补偿重试即补偿接口支持失败重试。当补偿接口内出现异常时，可依据重试相关配置多次尝试补偿。

### 参数配置

补偿重试相关配置参数如下：

```
txle:
 compensable-retry:
 times: 3
 interval: 3
```

txle.compensable-retry.times - 重试次数，默认为3次。

txle.compensable-retry.interval - 重试间隔，单位秒，默认为3秒。

### 自动补偿重试支持

自动补偿目前已默认支持补偿重试机制。

### 手动补偿重试支持

因手动补偿接口由业务人员进行实现，故手动补偿的重试机制目前尚无法默认支持。但已提供相应解决方案，业务开发人员可按照如下步骤实现手动补偿的重试机制。

```
...
import com.github.rholder.retry.Retryer;
import org.springframework.beans.factory.annotation.Autowired;
...

/**
 * 业务接口类
 */
public class XXX {
 @Autowired
 private Retryer retryer;

 public XXX XXX_rollback(..) {
 try {
 retryer.call(() -> {
 // *****业务补偿核心代码 start*****
 // ...
 // *****业务补偿核心代码 end*****
 });
 } catch (java.lang.Throwable e) {
 // 补偿重试仍然失败后，需调用上报差错平台接口
 }
 }
}
```

## 注意事项

补偿代码只有抛出`java.lang.Exception`或其子类异常才会被重试。

若业务中含有自定义异常处理，则需满足继承`java.lang.Throwable`。

### 3.3. 事务支持功能

- [3.3.1 手动补偿](#)
- [3.3.2 自动补偿](#)
- [3.3.3 重试](#)
- [3.3.4 超时](#)
- [3.3.5 嵌套](#)

## 手动补偿

手动补偿功能是保证(微服务)分布式事务场景下的数据最终一致性的核心功能。

手动补偿功能需通过在业务方法上添加注解@Compensable和编写对应的补偿接口来启用，补偿接口需被编写在同类中，且接口返回类型、参数类型等均要与被补偿的接口保持统一。

若开启了手动补偿功能，则子业务执行前开启子事务的同时会记录对应的补偿方法到当前子事务事件中，供后续(全局事务出现异常时)补偿调用。

手动补偿失败时，会收集全局事务和业务的异常信息并形成异常快照[上报至差错平台](#)。

手动补偿接口需支持幂等性。

## 自动补偿

自动补偿机制同手动补偿实现思想几乎一致，主要区别是自动补偿通过对业务SQL的代理完成自动识别业务SQL功能，并生成反向补偿的对应SQL，在后续需要补偿时执行反向SQL。

客户端通过增设回滚日志数据表来记录相应事务的反向SQL。

## 重试机制

全局事务针对各子事务支持重试功能，以避免因网络抖动等因素影响到业务的正常完成。

### 开启/关闭方法

通过在注解@Compensable或@AutoCompensable中设置retries来开启或关闭重试功能。

设置该参数值设置为0或不设置该参数，代表不使用重试功能。

设置大于0的正整数参数值，代表启用重试功能。

### 参数值

- 默认未设置该参数，即值为0，代表不重试。
- 当值为-1时，无限重试。
- 超时情况，暂不支持重试。

### 间隔时间

重试间隔时间默认为3s，可通过参数[omega.connection.reconnectDelay]进行配置。

### 注意事项

不支持超时场景下的重试。

补偿接口不支持重试。

重试信息修改后需重启应用才能生效。

## 超时机制

全局事务与各子事务均支持超时功能。

### 开启/关闭方法

通过在注解@Compensable或@AutoCompensable中设置timeout来开启或关闭超时功能。

### 参数值

默认未设置该参数，即值为0，代表不使用超时功能。

设置大于0的正整数值，代表启用超时功能。

### ★★★超时检测★★★

#### 后台定时任务检测

在全局事务正常运行过程中，定时任务会每隔0.5s对全局事务进行超时检测。

#### 全局事务结束前检测

部分场景，定时任务执行时间大于全局事务的超时时间，这导致了还未等对全局事务进行超时检测，结果全局事务就已经结束了。为杜绝此现象，增设全局事务结束前检测超时。

### 超时处理

检测到超时的全局事务后，先后创建超时记录、创建中止事件、创建待补偿命令、下发补偿命令、创建补偿事件、更新超时记录状态、更新待补偿命令、场景全局事务完成事件。

超时场景会对超时的全局事务下的**所有子事务都进行补偿**，而非超时异常仅针对当前全局事务下的非当前异常的**其它子事务进行补偿**。

### 注意事项

超时信息修改后需重启应用才能生效。

## 事务嵌套

- 嵌套全局 事务

即有两个 定义为全局 事务(@SagaStart)接口 A、B，A 中调用 B，当 B 发生错误时，A、B 中的操作都应回滚。

结果：ok。

- 手动补偿子事务嵌套

如子事务接口 A1、A2，A 1被全局事务调用，但A1中调用 A2。程序执行后，仅记录A1一个子事务，A2不会被看做为一个单独的子事务。但是如果全局事务中直接调用A2，那么A2将被视为一个单独的子事务。

- 自动补偿子事务嵌套同手动补偿子事务嵌套

## 全局事务服务降级

- 简介

大部分框架和中间件在设计初期均应考虑后续对主营业务的影响最小化，txle严格遵守该原则，提供服务降级功能。在程序运行时的多个阶段都可对全局事务进行降级或半降级，以及降级后的恢复功能，这在很大程度上保证了业务的正常运行。

- 背景

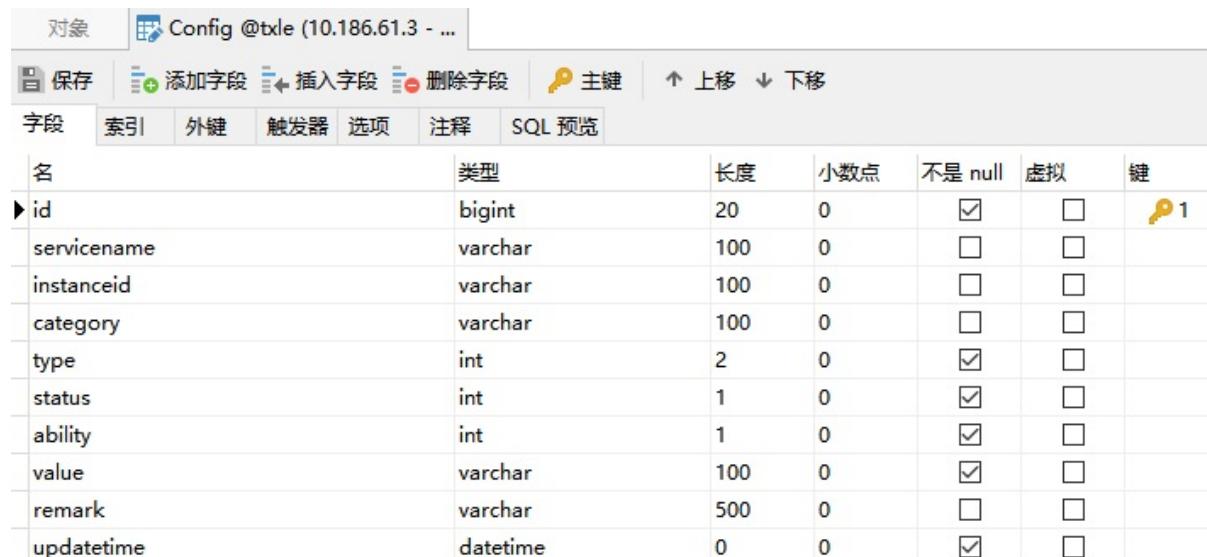
**性能影响：**当全局事务严重影响业务系统性能，且业务数据一致性要求不严苛时，可对全局事务采取降级处理。

**Bug：**当全局事务运行过程中出现bug时，未能保证业务数据不一致时，如未能及时定位问题与修复，可对全局事务采取降级处理。

**异常：**当全局事务运行过程中出现异常时，会回滚所涵盖的子业务功能，导致业务无法正常进行，可对全局事务采取降级处理。

**特殊情况：**除全局事务外，系统内其他主要功能，如监控、Kafka集成、差错平台等功能也提供了降级处理，主要作用都是在特殊情况不影响业务的正常运行。

- 配置中心数据表



The screenshot shows a database configuration interface with the following details:

**对象:** Config @txle (10.186.61.3 - ...)

**操作:** 保存, 添加字段, 插入字段, 删除字段, 主键, 上移, 下移

**字段:**

名	类型	长度	小数点	不是 null	虚拟	键
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1
servicename	varchar	100	0	<input type="checkbox"/>	<input type="checkbox"/>	
instanceid	varchar	100	0	<input type="checkbox"/>	<input type="checkbox"/>	
category	varchar	100	0	<input type="checkbox"/>	<input type="checkbox"/>	
type	int	2	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
status	int	1	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
ability	int	1	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
value	varchar	100	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
remark	varchar	500	0	<input type="checkbox"/>	<input type="checkbox"/>	
updatetime	datetime	0	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

```

CREATE TABLE IF NOT EXISTS Config (
 id bigint NOT NULL AUTO_INCREMENT,
 servicename varchar(100),
 instanceid varchar(100),
 type int(2) NOT NULL DEFAULT 0 COMMENT '1-globaltx, 2-compensation, 3-autocompensation, 4-bizinfotokafka, 5-txmonitor, 6-alert, 7-schedule, 8-globaltxfaulttolerant, 9-compensationfaulttolerant, 10-autocompensationfaulttolerant, 50-accidentreport, 51-sqlmonitor if values are less than 50, then configs for server, otherwise configs for client.',
 status int(1) NOT NULL DEFAULT 0 COMMENT '0-normal, 1-historical, 2-dumped',
 ability int(1) NOT NULL DEFAULT 1 COMMENT '0-do not provide ability, 1-provide ability ps: the client''s ability inherits the global ability.',
 value varchar(100) NOT NULL,
 remark varchar(500),
 updatetime datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
 PRIMARY KEY (id) USING BTREE,
 UNIQUE INDEX pk_id(id) USING BTREE,
 INDEX index_type(type) USING BTREE
) DEFAULT CHARSET=utf8;

```

- 配置管理

理应参考配置中心相关文档（暂未开发）。

提供展示UI，读取配置表数据并展示。

添加时，客户端名称和实例ID从全局事件数据表中读取。

由于数据量不大，故更新时先将原数据置为历史数据，再插入新数据。

删除全部为逻辑删除，即删除后状态变为历史数据。

- 降级类型

- 配置降级

即通过配置对部分功能进行降级处理与否。

系统提供部分核心功能的降级配置，当功能对应的配置值被设置为disabled时，则说明此功能被设置为降级处理，即程序暂停提供此功能直至解除降级。若程序运行一切正常，可通过设置配置值为enabled解除降级。

当然，降级配置值不仅仅包含disabled，后续会增设其它值，如范围值等。

- 容错降级

即全局事务系统运行过程中发生异常时，为不影响业务正常运行，将当前异常捕获掉，不回滚子业务已提交的事务。

容错降级也支持配置，即如果启用了相应功能的容错降级才会对异常进行捕获。

- 降级配置

**服务器端降级配置：**全局事务、手动补偿、自动补偿、Kafka集成、事务监控、告警、定时器。

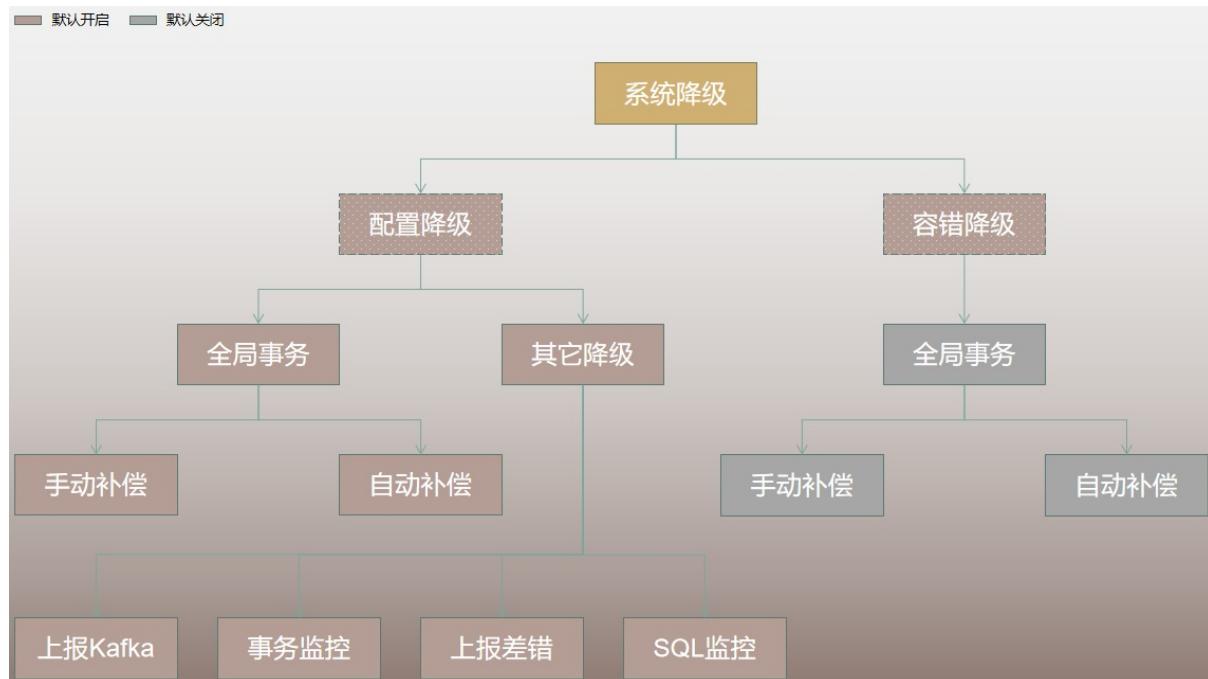
**容错降级配置：**全局事务容错、手动补偿容错、自动补偿容错。

**客户端降级配置：**上报差错、SQL监控。

ps：降级是概念上的，而真实的系统配置是针对实际的功能十分支持。

如针对降级配置，开启全局事务，意思是业务系统使用全局事务，关闭则是业务系统不使用全局事务；

如针对容错降级，开启全局事务是发生错误时，将进行容错，即不对子事务进行回滚，关闭时，如果发送错误，则会对子事务进行回滚操作；



### 全局事务：

在全局事务启动前会先检测是否启用全局事务配置，如配置值为disabled则不启动全局事务，直接执行业务。

每个全局事务启动前都会检查，以达到降级配置实时生效。已在运行的不受影响。

**手动补偿：** 同上。在启动子事务前检测。

**自动补偿：** 同上。

**Kafka集成：** 同上。在发送Kafka消息前检测。

**事务监控：** 在启用监控前检测。

**告警：** 暂未开发，可在Prometheus的配置文件中进行配置启用/禁用。

**定时器：** 暂未开发，目前已有定时器均针对所有客户端，故不便于降级。

**全局事务容错：** 在全局事务启动至结束代码处的异常捕获中增设容错降级功能，依据是否启用全局事务容错降级配置进行容错降级处理。

**手动补偿容错：** 同上。在手动补偿对应的子事务代码处。

**自动补偿容错：** 同上。在自动补偿对应的子事务代码处。

**上报差错：** 发生消息到差错平台前检测，依据该功能的降级配置值决定是否继续上报。

**SQL监控：**客户端业务执行过程中，如果开启全局事物则在手动补偿或自动补偿子事务启动前检测当前配置值，如果未开启全局事物则在业务执行前检测。

- 降级逻辑

系统针对部分核心功能提供降级配置。

默认除容错降级配置为disabled，其它降级配置值均为enabled。

每项配置都分为全局配置和客户端配置，全局配置提供是否支持某功能。若不支持，则采用全局默认值；若支持，优先客户端配置，客户端未配置则采用全局默认值；私有配置值不能超出全局配置值范围。

相关功能执行前，依据其对应的降级配置决定是否继续执行该功能。

由于降级需求往往都比较急迫，故所有配置均提供实时生效功能。

各配置检测时机不同，参考设计原则是“保证实时失效，节省性能”。

## 监控与告警

### 简介

监控主要对系统中的一些重要指标进行采集，采集后可友好地展示在UI中，便于后续对系统的性能、吞吐、稳定性等整体分析。

告警是依据采集的指标配合相关公式设计的告警规则，可及时地发现系统以及服务器存在的问题，避免时间长未处理而酿成重大后果。

除本系统重要指标外，也包括JVM、CPU、内存、磁盘和网络等。

### txle监控指标

#### TXLE项目

##### 全局事务数量

成功数量 - **txle\_transaction\_total**  
 失败数量 - **txle\_transaction\_failed\_total**  
 回滚数量 - **txle\_transaction\_rollbacked\_total**  
 超时数量 - **txle\_transaction\_time\_out\_total**  
 重试数量 - **txle\_transaction\_retried\_total**  
 暂停数量 - **txle\_transaction\_paused\_total**  
 继续数量 - **txle\_transaction\_continued\_total**  
 自动继续数量 - **txle\_transaction\_autocontinued\_total**

采用Prometheus的Counter类型统计，部分代码如下：

```
private final Counter txleTransactionTotal = buildCounter(name: "txle_transaction_total", help: "Total number of transactions.");
private final Counter txleTransactionSuccessfulTotal = buildCounter(name: "txle_transaction_successful_total", help: "Total number of successful transactions.");
private final Counter txleTransactionRollbackedTotal = buildCounter(name: "txle_transaction_rollbacked_total", help: "Total number of rollbacked transactions.");
private final Counter txleTransactionFailedTotal = buildCounter(name: "txle_transaction_failed_total", help: "Total number of transactions which had abnormality occurs.");
private final Counter txleTransactionRetriedTotal = buildCounter(name: "txle_transaction_retried_total", help: "Total number of retried transactions.");
private final Counter txleTransactionTimeoutTotal = buildCounter(name: "txle_transaction_time_out_total", help: "Total number of timeout transactions.");
private final Counter txleTransactionPausedTotal = buildCounter(name: "txle_transaction_paused_total", help: "Total number of paused transactions.");
private final Counter txleTransactionContinuedTotal = buildCounter(name: "txle_transaction_continued_total", help: "Total number of continued transactions.");
private final Counter txleTransactionAutocontinuedTotal = buildCounter(name: "txle_transaction_autocontinued_total", help: "Total number of auto-continued transactions.");

private Counter buildCounter(String name, String help) {
 return Counter.build(name, help).labelNames("business", "category").register();
}
```

```

public void countTxNumber(TxEvent event, boolean isTimeout, boolean isRetried) {
 if (!isEnableMonitor(event)) {
 return;
 }
 try {
 Set<String> eventTypesOfCurrentTx = globalTxIdAndTypes.get(event.globalTxId());
 if (eventTypesOfCurrentTx == null) {
 eventTypesOfCurrentTx = new HashSet<>();
 }
 if (globalTxIdAndTypes.isEmpty()) {
 // release memory
 globalTxIdAndTypes.clear();
 }
 String type = event.type(), serviceName = event.serviceName(), category = event.category();
 if (!eventTypesOfCurrentTx.contains(type)) {
 eventTypesOfCurrentTx.add(type);
 globalTxIdAndTypes.put(event.globalTxId(), eventTypesOfCurrentTx);

 if (SagaStartedEvent.name().equals(type)) {
 // ps: it would not appear in the metrics page if didn't set the labels' values.
 txleTransactionTotal.labels(serviceName, category).inc();
 } else if (SagaEndedEvent.name().equals(type)) {
 txleTransactionSuccessfulTotal.labels(serviceName, category).inc();
 globalTxIdAndTypes.remove(event.globalTxId());
 return;
 } else if (TxAbortedEvent.name().equals(type)) {
 txleTransactionFailedTotal.labels(serviceName, category).inc();
 } else if (TxCompensatedEvent.name().equals(type)) {
 txleTransactionRollbackedTotal.labels(serviceName, category).inc();
 } else if (AdditionalEventType.SagaPausedEvent.name().equals(type)) {
 txleTransactionPausedTotal.labels(serviceName, category).inc();
 }
 }
 }
}

```

## 子事务数量

**指标名称：txle\_transaction\_child\_total**

采用Prometheus的Counter类型统计，部分代码如下：

```

public void countChildTxNumber(TxEvent event) {
 if (!isEnableMonitor(event)) {
 return;
 }
 if (TxStartedEvent.name().equals(event.type()) && !localTxIdSet.contains(event.localTxId())) {
 txleTransactionChildTotal.labels(event.serviceName(), event.category()).inc();
 if (event.retries() > 0) {
 localTxIdSet.add(event.localTxId());
 }
 } else if (TxEndedEvent.name().equals(event.type()) || SagaEndedEvent.name().equals(event.type()) || TxAbortedEvent.name().equals(event.type())) {
 localTxIdSet.remove(event.localTxId());
 }

 if (localTxIdSet.isEmpty()) {
 log.info("The 'localTxIdSet' variable is empty, and it will be collected when JVM executes GC at the next time.");
 localTxIdSet.clear();
 }
}

```

## 全局事务时长

### 指标名称：txle\_transaction\_time\_seconds\_total

```
// Total seconds spent for someone transaction. Ps: It will show one row only if you search this metric in 'http://ip:9090/graph'.
// But, Prometheus will record every metric in different times, and you can search, like 'txle_transaction_time_seconds_total[5m]', then it will show many rows to you.
private final Gauge txleTransactionTimeSecondsTotal = buildGauge(name: "txle_transaction_time_seconds_total", help: "Total seconds spent executing the global transaction.");
private final Gauge txleTransactionChildTimeSecondsTotal = buildGauge(name: "txle_transaction_child_time_seconds_total", help: "Total seconds spent executing the child transaction.");

private Gauge buildGauge(String name, String help) {
 return Gauge.build(name, help).labelNames("business", "category").register();
}

public void startMarkTxDuration(TxEvent event) {
 if (!isEnabledMonitor(event)) {
 return;
 }
 // Start a timer to track a duration, for the gauge with no labels. So, we must set the value of the labelNames property.
 if (SagaStartedEvent.name().equals(event.type())) {
 txIdAndGaugeTimer.put(event.globalTxId(), txleTransactionTimeSecondsTotal.labels(event.serviceName(), event.category()).startTimer());
 } else if (TxStartedEvent.name().equals(event.type())) {
 txIdAndGaugeTimer.put(event.localTxId(), txleTransactionChildTimeSecondsTotal.labels(event.serviceName(), event.category()).startTimer());
 }
}
```

## 子事务时长

### 指标名称：txle\_transaction\_child\_time\_seconds\_total

同全局事务时长，将globalTxId改为localTxId。

## 业务SQL时长与数量

### 指标名称：txle\_sql\_time\_seconds\_total、txle\_sql\_total

即业务应用系统执行的SQL。

标准业务SQL统计时已依据业务和事务类别分组。

未分组的业务SQL主要包含：自动补偿SQL。

注：当某子业务被成功回滚时，对应的**txle\_sql\_total**会有7条，即针对当前子事务会有如下操作“开启事务、结束事务、查询回滚状态、新增待完成的回滚命令、新增回滚事件、更新回滚命令为完成、更新全局事务结束事件”。

## 维护SQL时长与数量

### 指标名称：txle\_sql\_time\_seconds\_total、txle\_sql\_total

即TXLE项目执行的SQL，含Client端自动补偿相关维护SQL和Server端的维护(或所有)SQL。

标准维护SQL统计时已依据业务和事务类别分组。

未分组的维护SQL主要包含：Client端自动补偿SQL、Server端部分定时器SQL(无全局事务标识参数的)、超时和补偿事件SQL。

## JVM

采用prometheus自带统计方式：

```
io.prometheus.client.hotspot.DefaultExports.initialize();
```

## CPU、内存、磁盘、网络

安装第三方工具**Node exporter**来统计指标。

## TXLE默认Prometheus客户端服务端口

### TXLE Server端默认Prometheus端口8099

默认端口逻辑：默认端口配置在**application.yaml**文件中，所以程序中获取到的任何值均可认为是维护人员主动编辑，所以在程序层面如果获取到无效值，则意味着不启动Prometheus客户端监控服务。

#### application.yml

```
txle:
 prometheus:
 metrics:
 port: 8099
```

## TXLE Client端默认Prometheus端口8098

默认端口逻辑：Client与Server不一样，Client端的默认端口只能写在**Java代码**中。因为Client端的Prometheus服务端口需由集成业务进行设置，所以无法在Client端进行默认配置。

#### application.yml

```
txle:
 prometheus:
 metrics:
 port: 8098
```

#### application.properties

```
txle.prometheus.metrics.port=8098
```

## 4.配置说明

- [4.1 启动前配置](#)
- [4.2运行中配置](#)

## 4.1 配置文件

### 4.1.1 txle服务端配置文件

#### 4.1.1.1 application.yaml主要参数说明

配置名称	配置内容	默认值
server.port	txle服务端提供UI服务的访问端口	8090
alpha.server.host	txle服务端提供全局事务一致性服务的绑定地址	0.0.0.0
alpha.server.port	txle服务端提供全局事务一致性服务的访问端口	8080
spring.datasource.url	txle服务端数据库访问地址	jdbc:mysql://127.0.0.1:3306/txle?characterEncoding=utf-8
spring.datasource.username	txle服务端数据库访问用户名	root
spring.datasource.password	txle服务端数据库访问密码	123456
spring.zipkin.base-url	txle服务端跟踪系统，配合zipkin server一起使用	http://127.0.0.1:9411/api/v2/spans
spring.cloud.consul.enabled	txle服务端启动consul选主	true
spring.cloud.consul.host	txle服务端启动consul选主时consul的访问地址	127.0.0.1
spring.cloud.consul.port	txle服务端启动consul选主时consul的访问端口	8500
txle.prometheus.metrics.port	txle服务端提供的Prometheus获取元数据端口	8099

#### 4.1.1.2 kafka.properties主要参数说明

配置名称	配置内容	默认值
bootstrap.servers	txle服务端使用的kafka集群的访问地址，多个服务端使用逗号分隔	kafka1:9092,kafka2:9092,kafka3:9092

### 4.1.2 txle业务端配置文件

配置名称	配置内容	默认值
alpha.cluster.address	所使用的的txle服务端提供的访问地址和端口，多个服务端使用逗号分隔	127.0.0.1:8080
txle.prometheus.metrics.port	txle业务端提供的Prometheus获取元数据端口	
spring.zipkin.base-url	txle业务端跟踪系统，配合zipkin server一起使用	http://127.0.0.1:9411/api/v2/spans
txle.compensable-retry.times	补偿重试次数	3
txle.compensable-retry.interval	补偿重试间隔时间(秒)	3

## 4.2 热加载配置

### 4.2.1 kafka配置

1. 修改 `/path/to/txle/conf/kafka.properties`
2. 调用REST API接口: `http://${txle_host_address}:${txle_rpc_port}/reloadConfig/kafka` 使配置生效

### 4.2.2 系统降级配置

1. 访问txle配置中心 `http://10.186.62.24:8090/#/txle/config_center`
2. 可配置项 参考[全局事务服务降级](#)

- 全局事务
- 手动补偿
- 自动补偿
- 业务信息上报
- 事务监控
- 告警
- 定时任务
- 全局事务容错
- 手动补偿容错
- 自动补偿容错
- 暂停全局事务
- 历史表间隔规则
- 差错上报
- SQL监控

## 常见问题

### 5.1 是否支持docker快速部署？

[支持常规部署、`docker`镜像部署、以及`docker compose`部署。](#)

### 5.2 QPS/TPS？

`txle` QPS 指标：

并发数	总耗时(s)	平均耗时 (ms)	吞吐率	事务量
100	38.031	36.27	2628.33	100000
500	38.271	181.01	2611.92	100000
1000	47.606	460.15	2099.69	100000
1250	47.615	575.74	2098.9	100000
1500	48.449	694.76	2073.28	100000
2000	49.019	914.19	2038.99	100000

`ServiceComb Pack` QPS 指标：

节点数	并发数	总耗时(s)	平均耗时(ms)	吞吐率	事务量
0.5.0 without Akka	100	102	1026	98/sec	10000
0.5.0 without Akka	500	99	4970	101/sec	10000
0.5.0 with Akka	100	14	142	714/sec	10000
0.5.0 with Akka	500	8	418	1250/sec	10000
0.5.0 with Akka	1000	8	858	1250/sec	10000
0.5.0 with Akka	2000	14	2888	714/sec	10000
0.5.0 with Akka	1000	39	786	1282/sec	50000
0.5.0 with Akka	2000	37	1519	1351/sec	50000
0.5.0 with Akka	3000	43	2687	1116/sec	50000

### 5.3 对业务系统的性能影响？

单事务分支平均响应时间在2ms左右。

## 5.4 保证ACID么？

保证ACD，不保证Isolation。

那支持并发场景么？或并发场景会存在什么隐患么？

当然支持并发场景。业务无任何异常情况并发场景完全ok；并发未更新同一数据，即使异常也可被补偿；并发更新同一数据时，若无异常完全ok，若最后的事务出现异常也可补偿；若非最后的事务出现异常但为计算型赋值也可补偿，若直接赋值将补偿失败，则将上报差错。

支持并发、支持补偿，支持部分并发补偿(并发时最后的事务异常可以被补偿，非最后事务会补偿失败，将上报差错)。

## 5.5 支持哪些数据库？

txle目前支持手动补偿机制和自动补偿机制。

手动补偿机制支持异构数据库，对数据库无要求。

自动补偿机制目前支持MySQL，其它数据库支持规划中。

## 5.6 对数据库连接池有特殊要求么？

暂无特殊要求。

## 5.7 对于补偿方法的要求？

- 幂等
- 与被补偿方法在同一类且参数完全一致
- 参数可被序列化

## 5.8 是否支持事务嵌套？

支持子事件嵌套。

## 5.9 txle server端支持水平扩展么？

支持。相关状态信息均存储于数据库。

## 5.10 txle server宕机重启后，是否可继续当前全局事务？

可以继续当前全局事务。

## 6.0 txle vs other softwares

### 6.0.1 txle vs XA/TCC

- **高性能** 无需锁定数据资源
- **多重补偿方式** 支持手动补偿与自动补偿机制

### 6.0.2 txle vs Seata

- **高性能**
- **手动补偿** 相比Seata自带的自动补偿机制更加灵活，支持复杂业务场景

### 6.0.3 txle vs ServiceComb Pack

- **高性能** 提升几倍的QPS
- **多重补偿方式** 支持自动补偿机制，相比手动补偿机制，节省了大量的开发成本

### 6.0.4 txle vs other softwares

- **高性能**
- **专注金融领域**
- **支持服务降级**
- **支持差错处理**

## 7.示例工程

- [7.1 Spring Boot示例工程](#)
- [7.2 Spring Cloud示例工程](#)
- [7.3 Dubbo示例工程](#)

## 7.1 spring boot示例

### 7.1.1 关于本节

- 本节内容为您介绍如何编译部署spring boot业务端代码
- 该示例包含4个服务
  - global
  - transfer
  - user
  - merchant

### 7.1.2 环境准备

- JDK 1.8
- Maven 3.x
- txle server

### 7.1.3 运行示例

1. 参照[1.2 docker image部署](#)的说明，安装部署txle server
2. 运行如下命令创建业务代码所需要的数据库

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> create database db1;
Query OK, 1 row affected (0.00 sec)

mysql> create database db2;
Query OK, 1 row affected (0.00 sec)

mysql> create database db3;
Query OK, 1 row affected (0.00 sec)
```

3. 运行如下命令编译打包业务端代码

```
$ git clone git@github.com:actiontech/txle.git
$ cd txle/examples/sample-txle-springboot && \
mvn clean package
```

4. 运行业务端代码

i. 运行global服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-springboot-global/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springboot-global-0.0.1-SNAPSHOT.jar
```

- ii. 运行transfer服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-springboot-transfer/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springboot-transfer-0.0.1-SNAPSHOT.jar
```



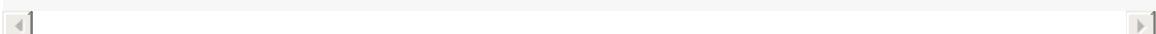
- iii. 运行user服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-springboot-user/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springboot-user-0.0.1-SNAPSHOT.jar
```



- iv. 运行merchant服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-springboot-merchant/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springboot-merchant-0.0.1-SNAPSHOT.jar
```



## 7.1.4 使用工具发起全局事务

1. 发起转账1元，转账将会成功

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/1/1
$ ok
```

2. 访问数据库查看账户余额

txle\_sample\_transfer 记录了一条交易成功的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据正常记录

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:5
4 |
+----+-----+-----+-----+-----+-----+
--+
1 row in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

3. 根据示例代码里的设定发起转账金额大于200元，第二个子服务merchant会报错导致转账将会失败

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/300/1
$ {"timestamp":1572429018848,"status":500,"error":"Internal Server Error","exception":
"org.springframework.web.client.HttpServerErrorException","message":"500 null","path":
"/testGlobalTransaction/1/300/1"}
```

4. 访问数据库查看账户余额

`txle_sample_transfer` 记录了一条交易失败的记录，`txle_sample_user` 和 `txle_sample_merchant` 数据回滚

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime |
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
| 2 | 1 | 1 | 300.00000 | 1 | 2 | 0 | 2019-10-30 09:50:18 |
+----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```

## 7.2 spring cloud示例

### 7.2.1 关于本节

- 本节内容为您介绍如何编译部署spring cloud业务端代码
- 该示例包含4个服务
  - global
  - transfer
  - user
  - merchant

### 7.2.2 环境准备

- JDK 1.8
- Maven 3.x
- txle server

### 7.2.3 运行示例

1. 参照[1.2 docker image部署](#)的说明，安装部署txle server
2. 运行如下命令创建业务代码所需要的数据库

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> create database db1;
Query OK, 1 row affected (0.00 sec)

mysql> create database db2;
Query OK, 1 row affected (0.00 sec)

mysql> create database db3;
Query OK, 1 row affected (0.00 sec)
```

3. 运行如下命令编译打包业务端代码

```
$ git clone git@github.com:actiontech/txle.git
$ cd txle/examples/sample-txle-springcloud && \
mvn clean package
```

4. 运行业务端代码

i. 运行eureka服务。打包好的jar在 txle/examples/sample-txle-springcloud/sample-txle-springcloud-eureka/target/

```
$ java -jar sample-txle-springcloud-eureka-0.0.1-SNAPSHOT.jar
```

ii. 运行global服务。打包好的jar在 txle/examples/sample-txle-springcloud/sample-txle-

```
springcloud-global/target/
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springcloud-global-0.0.1-SNAPSHOT.jar
```

- iii. 运行transfer服务。打包好的jar在 txle/examples/sample-txle-springcloud/sample-txle-springcloud-transfer/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springcloud-transfer-0.0.1-SNAPSHOT.jar
```

- iv. 运行user服务。打包好的jar在 txle/examples/sample-txle-springcloud/sample-txle-springcloud-user/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springcloud-user-0.0.1-SNAPSHOT.jar
```

- v. 运行merchant服务。打包好的jar在 txle/examples/sample-txle-springcloud/sample-txle-springcloud-merchant/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-springcloud-merchant-0.0.1-SNAPSHOT.jar
```

## 7.1.4 使用工具发起全局事务

1. 发起转账1元，转账将会成功

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/1/1
$ ok
```

2. 访问数据库查看账户余额

txle\_sample\_transfer 记录了一条交易成功的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据正常记录

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+-----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:5
4 |
+-----+-----+-----+-----+-----+-----+
--+
1 row in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+-----+-----+-----+-----+
| id | name | balance | version | createtime |
+-----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+-----+-----+-----+-----+
| id | name | balance | version | createtime |
+-----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

3. 根据示例代码里的设定发起转账金额大于200元，第二个子服务merchant会报错导致转账将会失败

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/300/1
$ {"timestamp":1572429018848,"status":500,"error":"Internal Server Error","exception":
"org.springframework.web.client.HttpServerErrorException","message":"500 null","path":
"/testGlobalTransaction/1/300/1"}
```

4. 访问数据库查看账户余额

`txle_sample_transfer` 记录了一条交易失败的记录，`txle_sample_user` 和 `txle_sample_merchant` 数据回滚

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime |
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
| 2 | 1 | 1 | 300.00000 | 1 | 2 | 0 | 2019-10-30 09:50:18 |
+----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```

## 7.3 dubbo示例

### 7.3.1 关于本节

- 本节内容为您介绍如何编译部署dubbo业务端代码
- 该示例包含4个服务
  - global
  - transfer
  - user
  - merchant

### 7.3.2 环境准备

- JDK 1.8
- Maven 3.x
- txle server

### 7.3.3 运行示例

1. 参照[1.2 docker image部署](#)的说明，安装部署txle server
2. dubbo架构需要zookeeper支持，可以使用如下命令创建

```
$ docker run --name=dubbo_zookeeper -p 2181:2181 --network=txle_net --ip=172.20.0.61
zookeeper
```

3. 运行如下命令创建业务代码所需要的数据库

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> create database db1;
Query OK, 1 row affected (0.00 sec)

mysql> create database db2;
Query OK, 1 row affected (0.00 sec)

mysql> create database db3;
Query OK, 1 row affected (0.00 sec)
```

4. 运行如下命令编译打包业务端代码

```
$ git clone git@github.com:actiontech/txle.git
$ cd txle/examples/sample-txle-dubbo && \
mvn clean package
```

5. 运行业务端代码

i. 运行transfer服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-dubbo-

```
provider-transfer/target/
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-dubbo-provider-transfer-0.0.1-SNAPSHOT.jar
```

- ii. 运行user服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-dubbo-provider-user/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-dubbo-provider-user-0.0.1-SNAPSHOT.jar
```

- iii. 运行merchant服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-dubbo-provider-merchant/target/

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-dubbo-provider-merchant-0.0.1-SNAPSHOT.jar
```

- iv. 运行consumer服务。打包好的jar在 txle/examples/sample-txle-springboot/sample-txle-dubbo-consumer/target/

**请务必等前面三个子服务启动完毕后，再启动consumer服务**

```
$ java -Dalpha.cluster.address=${txle_server_address}:8080 -jar sample-txle-dubbo-consumer-0.0.1-SNAPSHOT.jar
```

## 7.1.4 使用工具发起全局事务

1. 发起转账1元，转账将会成功

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/1/1
$ ok
```

2. 访问数据库查看账户余额

txle\_sample\_transfer 记录了一条交易成功的记录， txle\_sample\_user 和 txle\_sample\_merchant 数据正常记录

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
+----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.00 sec)
```

3. 根据示例代码里的设定发起转账金额大于200元，第二个子服务merchant会报错导致转账将会失败

```
$ curl http://${host_address}:8000/testGlobalTransaction/1/300/1
$ The 'Merchant' Service threw a runtime exception in case of balance was more than 200.
```

4. 访问数据库查看账户余额

`txle_sample_transfer` 记录了一条交易失败的记录，`txle_sample_user` 和 `txle_sample_merchant` 数据回滚

```
$ docker exec -it backend-mysql bash
mysql -uroot -p123456
mysql> select * from db1.txle_sample_transfer;
+----+-----+-----+-----+-----+-----+
| id | userid | merchantid | amount | payway | status | version | createtime |
+----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 1.00000 | 1 | 1 | 0 | 2019-10-30 09:47:54 |
| 2 | 1 | 1 | 300.00000 | 1 | 2 | 0 | 2019-10-30 09:50:18 |
+----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> select * from db2.txle_sample_user;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | user | 999.00000 | 1 | 2019-10-30 09:47:30 |
+----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select * from db3.txle_sample_merchant;
+----+-----+-----+-----+
| id | name | balance | version | createtime |
+----+-----+-----+-----+
| 1 | merchant | 1.00000 | 1 | 2019-10-30 09:47:31 |
+----+-----+-----+-----+
1 row in set (0.01 sec)
```



## 8.1 性能测试报告

### 8.1.1 关于本节

性能测试使用 jmeter 发送全局事务请求，txle服务端和业务端demo分别部署在物理服务器上的不同 docker容器中

### 8.1.2 配置

#### 8.1.2.1 物理服务器配置

名称	描述
服务器	Dell PowerEdge R740xd
CPU	Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz
核数	80
内存	256G
硬盘	SSD
网卡	Intel Corporation I350 Gigabit Network Connection (rev 01)

#### 8.1.2.2 Docker版本

```
$ docker version
Client: Docker Engine - Community
 Version: 19.03.4
 API version: 1.40
 Go version: go1.12.10
 Git commit: 9013bf583a
 Built: Fri Oct 18 15:54:09 2019
 OS/Arch: linux/amd64
 Experimental: false

Server: Docker Engine - Community
Engine:
 Version: 19.03.4
 API version: 1.40 (minimum version 1.12)
 Go version: go1.12.10
 Git commit: 9013bf583a
 Built: Fri Oct 18 15:52:40 2019
 OS/Arch: linux/amd64
 Experimental: false
containerd:
 Version: 1.2.10
 GitCommit: b34a5c8af56e510852c35414db4c1f4fa6172339
runc:
 Version: 1.0.0-rc8+dev
 GitCommit: 3e425f80a8c931f88e6d94a8c831b9d5aa481657
docker-init:
 Version: 0.18.0
 GitCommit: fec3683
```

### 8.1.2.3 jmeter版本

```
$ jmeter --version
5.1.1 r1855137
```

### 8.1.2.4 txle服务端配置

```

java \
-Djava.awt.headless=true \
-Djava.net.preferIPv4Stack=true \
-server -Xmx2g -Xms2g -Xmn256m \
-XX:PermSize=128m \
-Xloggc:/root/alpha-server/tule/1og/gc-stdout.log \
-XX:+DisableExplicitGC \
-XX:+UseConcMarkSweepGC \
-XX:+CMSParallelRemarkEnabled \
-XX:+UseCMSCompactAtFullCollection \
-XX:LargePageSizeInBytes=128m \
-XX:+UseFastAccessorMethods \
-XX:+UseCMSInitiatingOccupancyOnly \
-XX:CMSInitiatingOccupancyFraction=70
-jar tule-server-2.19.10.0.jar
--spring.datasource.username=root \
--spring.datasource.password=123456 \
--spring.datasource.url="jdbc:mysql://172.20.0.11:3306/tule?characterEncoding=utf-8"
\
--spring.profile.active=mysql

```

### 8.1.3 测试报告

总结：

- 吞吐率最高可达到每秒2628笔全局事务,每笔全局事务包含包含2个子事务
- 本单节点Alpha最优并发量为 2000, 平均耗<1s, 吞吐率 2000+/秒
- 本测试客户端和服务端分别部署在不同的docker容器中, 模拟真实网络通讯
- 本测试使用真实的springboot客户端, 模拟真实的使用场景
- 本测试使用jmeter分布式测试环境, 模拟真正的压力场景

并发数	总耗时(s)	平均耗时 (ms)	吞吐率	事务量
100	38.031	36.27	2628.33	100000
500	38.271	181.01	2611.92	100000
1000	47.606	460.15	2099.69	100000
1250	47.615	575.74	2098.9	100000
1500	48.449	694.76	2073.28	100000
2000	49.019	914.19	2038.99	100000