# 功能

* 分布式事务的tcc开源方案
* 支持嵌套事务(nested transaction support)
* 采用disruptor框架进行事务日志的异步读写，与rpc框架的性能毫无差别
* 支持springboot-starter项目启动，使用简单
* rpc框架支持:dubbo,springcloud
* 本地事务存储支持:redis,mongodb,zookeeper,file,mysql
* 事务日志序列化支持:java，hessian，kryo，protostuff
* 采用aspect aop切面思想与spring无缝集成，支持集群
* 分布式事务场景demo工程，并有swagger-ui可视化界面可以快速体验

# 项目结构

* li-tcc

li-tcc-common提供分布式事务的@Tcc注解,tcc框架的一个公共项目，里面主要是一些配置，枚举，异常定义等

li-tcc-core该项目是tcc框架的核心实现，包括服务的启动，调用流程，aop切面，重试等实现

li-tcc-dubbo该项目是对dubbo框架的支持，里面主要针对dubbo的特性的实现

li-tcc-springcloud该项目是对springcloud框架的支持，里面主要针对springcloud的特性的实现

* li-tcc-admin

li-tcc-admin该项目是分布式事务的跟踪管理后台（调用链跟踪，控制补偿事务等功能）

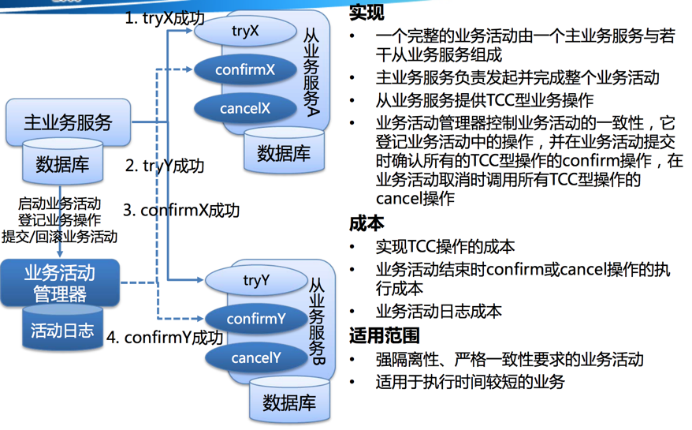
li-tcc-platform该项目是分布式事务管理后台的前端源码，采用vue.js+element UI实现

* li-tcc-demo

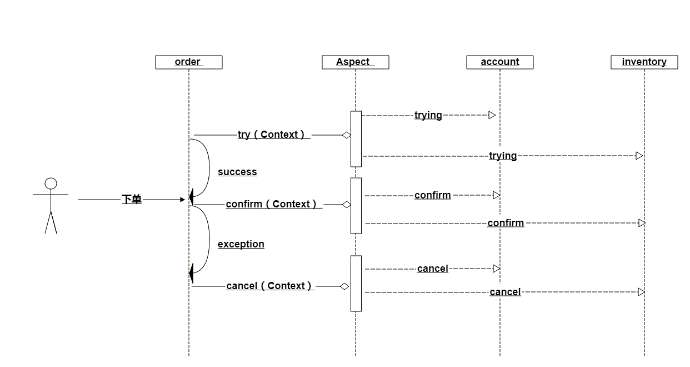
li-tcc-demo这是体验的demo项目，里面有针对dubbo和springcloud的案列，具体的配置，用户可以参考

# 原理图

* 原理图



* 流程图



# 源码解析

* @Tcc

AbstractTccTransactionAspect:

@Aspect

public abstract class AbstractTccTransactionAspect {

private TccTransactionInterceptor tccTransactionInterceptor;

protected void setTccTransactionInterceptor(

final TccTransactionInterceptor tccTransactionInterceptor) {

this.tccTransactionInterceptor = tccTransactionInterceptor;

}

/\*\*

\* this is point cut with Tcc

\*/

@Pointcut("@annotation(com.li.tcc.common.annotation.Tcc)")

public void liTccInterceptor() {

}

/\*\*

\* this is around in Tcc

\*

\* @param proceedingJoinPoint

\* @return Object

\* @throws Throwable

\*/

@Around("liTccInterceptor()")

public Object interceptTccMethod(

final ProceedingJoinPoint proceedingJoinPoint) throws Throwable {

return tccTransactionInterceptor.interceptor(proceedingJoinPoint);

}

/\*\*

\* spring Order

\*

\* @return int

\*/

public abstract int getOrder();

}

说明：

1. 这里定义@Tcc的切点
2. 加了@Tcc注解的，在调用的时候，都会进行tccTransactionInterceptor.interceptor调用
3. 该类是一个抽象类，肯定会有其他类继承它，对应

dubbo用户，他的继承类为: DubboLiTransactionAspect;

springcloud的用户,它的继承类为: SpringCloudLiTransactionAspect

他们都实现了Spring的Ordered接口，并重写了getOrder方法，都返回了Ordered.HIGHEST\_PRECEDENCE，他是优先级最高的切面

* 启动详解

LiTransactionBootstrap

1. 它继承TccConfig能获取在xml配置的属性信息，实现ApplicationContextAware spring容器初始化的时候，会自动的将ApplicationContext注入进来
2. 进入**initialization**方法
3. 进入loadSpiSupport(tccConfig)，loadSpiSupport采用jdk自带的spi加载
4. 进入liTransactionEventPublisher.start(tccConfig.getBufferSize())，disruptor，executor
5. 进入**coordinatorService.start(tccConfig)，coordinatorRepository.init(appName,tccConfig)**就是根据spi思想来具体初始化，**scheduledRollBack()**执行定时补偿

* 事务处理

1. **ScheduledService.scheduledRollBack()**执行定时补偿
2. tccTransactionAspectService.invoke(final TccTransactionContext tccTransactionContext,final ProceedingJoinPoint point)
3. 进入txTransactionHandler.handler(point, tccTransactionContext),已经进入了分布式事务处理的入口了

@Override

public Object handler(final ProceedingJoinPoint point,

TccTransactionContext context) throws Throwable {

Object returnValue = null;

try {

TccTransaction tccTransaction;

context = TransactionContextLocal.getInstance().get();

if (context == null) {

tccTransaction = liTransactionExecutor.begin(point);

try {

// execute try

returnValue = point.proceed();

tccTransaction.setStatus(TccActionEnum.TRYING.getCode());

liTransactionExecutor.updateStatus(tccTransaction);

} catch (Throwable throwable) {

// if exception ,execute cancel

liTransactionExecutor.cancel(liTransactionExecutor

.getCurrentTransaction());

throw throwable;

}

// execute confirm

liTransactionExecutor.confirm(liTransactionExecutor

.getCurrentTransaction());

} else if (context.getAction() == TccActionEnum.CONFIRMING

.getCode()) {

// execute confirm

liTransactionExecutor.confirm(liTransactionExecutor

.getCurrentTransaction());

}

} finally {

liTransactionExecutor.remove();

}

return returnValue;

}

1. 进入liTransactionExecutor.begin(point)，这里保存了事务信息，并且开启了事务上下文，并把它保存在了ThreadLoacl里面
2. 进入liTransactionExecutor.\*\*

@SuppressWarnings("rawtypes")

private void executeParticipantMethod(final TccInvocation tccInvocation)

throws Exception {

if (Objects.nonNull(tccInvocation)) {

final Class clazz = tccInvocation.getTargetClass();

final String method = tccInvocation.getMethodName();

final Object[] args = tccInvocation.getArgs();

final Class[] parameterTypes = tccInvocation.getParameterTypes();

final Object bean = SpringBeanUtils.getInstance().getBean(clazz);

MethodUtils.invokeMethod(bean, method, args, parameterTypes);

}

}

# 接入说明

* @Tcc详解

@Tcc该注解为分布式事务的切面（AOP point）

1. 注解中confirmMethod="xxx"为在tcc分布式事务中confirm角色的方法名称
2. 注解中cancelMethod="xxx"为在tcc分布式事务中cancel角色的方法名称
3. 注解中TccPatternEnum为在tcc分布式事务中的模式，现在有tcc，和cc2种

特别注意：try，confirm，cancel 3个方法的参数类型必须一致

cc模式含义为confrim，cancel，即在try中没有任何数据的操作，只有对数据的校验，在try阶段发生异常，不会进行cancel方法的调用

使用配置：

1. 在接口上添加@Tcc注解（dubbo则需要填加在api接口上，springcloud则需要加在feignClient上），具体参考demo工程
2. 在接口实现上添加@Tcc(confirmMethod="方法名称",cancelMethod="方法名称"),并提供confrim，cancel方法名称，具体参考demo工程

* applicationContext.xml详解

<!--扫描框架的包-->

<context:component-scan base-package="com.li.tcc.\*" />

<!-- Aspect 切面配置，是否开启AOP切面-->

<aop:aspectj-autoproxy expose-proxy="true" />

<!--启动类属性配置-->

<bean id="liTransactionBootstrap" class="com.li.tcc.core.bootstrap.LiTransactionBootstrap">

<property name="serializer" value="kryo"/>

<property name="recoverDelayTime" value="120"/>

<property name="retryMax" value="3"/>

<property name="scheduledDelay" value="120"/>

<property name="scheduledThreadMax" value="4"/>

<property name="repositorySupport" value="db"/>

<property name="tccDbConfig">

<bean class="com.li.tcc.common.config.TccDbConfig">

<property name="url" value="jdbc:mysql://mysql-dev:3306/tcc?useUnicode=true&amp;characterEncoding=utf8"/>

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

</property>

</bean>

说明：

1. LiTransactionBootstrap属性见com.li.tcc.common.config.TccConfig
2. 本地数据保存配置与详解

spi扩展支持db，redis，zookeeper，mongodb，file,详情配置请参照demo工程

<property name="repositorySupport" value="db" />

<property name="tccDbConfig">

<bean class="TccDbConfig">

<property name="url" value="jdbc:mysql://mysql-dev:3306/tcc?useUnicode=true&amp;characterEncoding=utf8" />

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="password" value="root" />

<property name="username" value="root" />

</bean>

</property>

<property name="repositorySupport" value="redis" />

<property name="tccRedisConfig">

<bean class="TccRedisConfig">

<property name="hostName" value="redis-dev" />

<property name="port" value="6379" />

</bean>

</property>

<property name="repositorySupport" value="zookeeper" />

<property name="tccZookeeperConfig">

<bean class="TccZookeeperConfig">

<property name="host" value="zookeeper-dev:2181" />

<property name="sessionTimeOut" value="100000" />

<property name="rootPath" value="/tcc" />

</bean>

</property>

<property name="repositorySupport" value="mongodb" />

<property name="tccMongoConfig">

<bean class="TccMongoConfig">

<property name="mongoDbUrl" value="mongodb-dev:27017" />

<property name="mongoDbName" value="tcc" />

<property name="mongoUserName" value="root" />

<property name="mongoUserPwd" value="root" />

</bean>

</property>

<property name="repositorySupport" value="file" />

<property name="tccFileConfig">

<bean class="TccFileConfig">

<property name="path" value="/inventory" />

<property name="prefix" value="inventory" />

</bean>

</property>

# li-tcc-admin部署

* 分离部署

li-tcc-platform：

npm install（首次运行时）

npm run build

npm run start

浏览器运行http://localhost:9001/index.html#/

li-tcc-admin：

运行AdminApplication的main方法

* 统一部署

li-tcc-platform：

npm install（首次运行时）

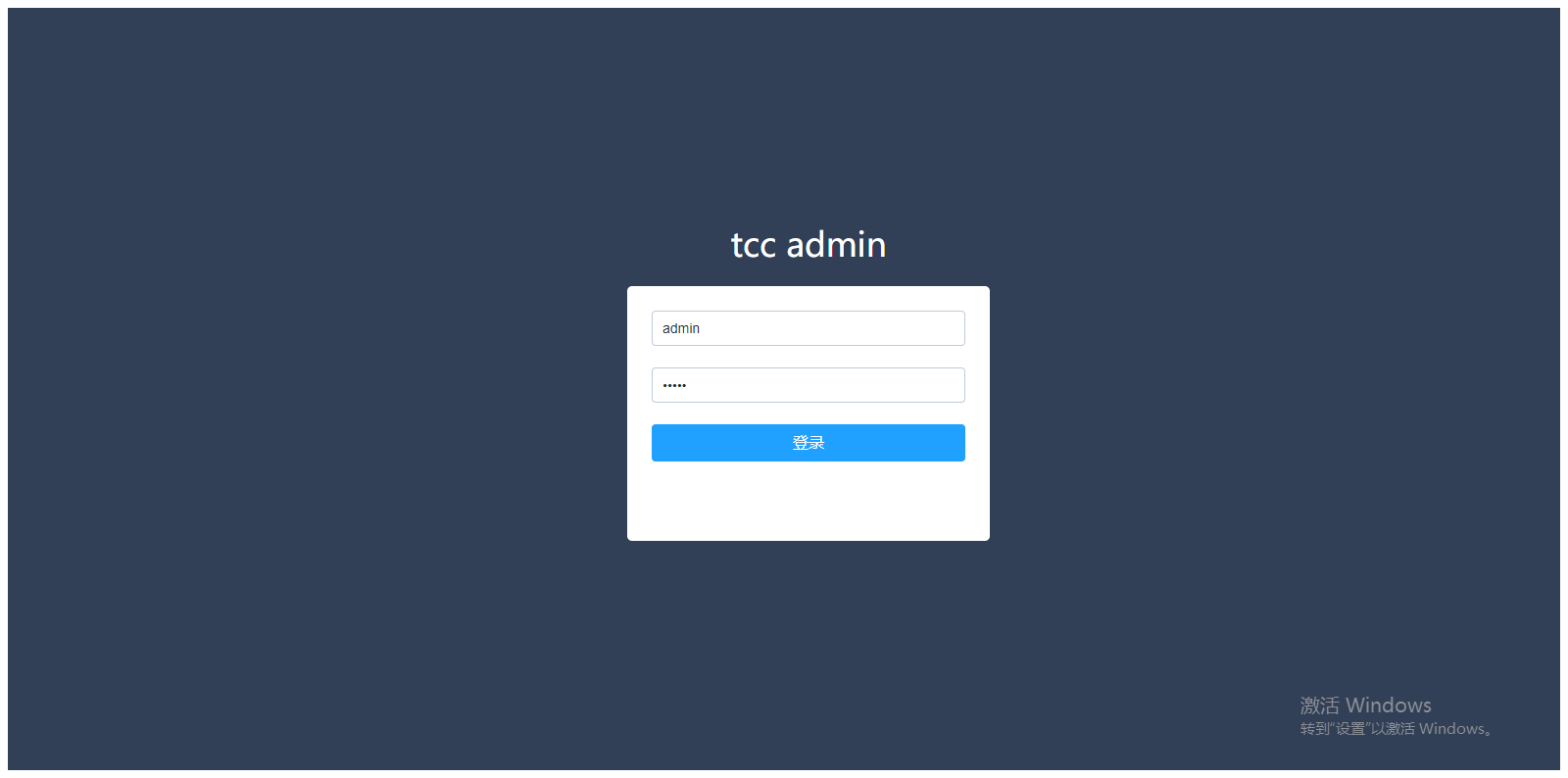
npm run build

li-tcc-admin：

替换li-tcc-platform/dist/到li-tcc-admin的resources\static

运行AdminApplication的main方法

浏览器运行<http://localhost:8888/tcc-admin/index.html>





# li-tcc-demo示例

* 执行步骤

1. 执行tcc-demo.sql
2. li-tcc-admin部署
3. li-tcc-demo示例

* dubbo

1. 进入li-tcc-demo-dubbo-account项目，修改application.yml中的数据库配置
2. 修改applicationContext.xml中的配置
3. 修改spring-dubbo.xml 中的zookeeper配置
4. Inventory,order项目的配置修改和上面的一样，注意dubbo的端口不要重复。
5. 依次执行AccountApplication,InventoryApplication,OrderApplication中的main方法
6. 访问<http://localhost:8083/swagger-ui.html>进入体验体验dubbo的分布式事务

* springcloud

1. 修改各项目中的application.yml的数据库配置
2. 修改各项目中applicationContext.xml的配置
3. 执行li-tcc-demo-springcloud-eureka项目中的EurekaServerApplication类的main方法
4. 依次执行AccountApplication,InventoryApplication,OrderApplication中的main方法
5. 访问<http://localhost:8884/swagger-ui.html> 进入体验springcloud分布式事务

* 部署
* 服务器：使用springboot框架内嵌tomcat中间件，无需单独安装
* 编译命令：mvn clean install -Pdev -Dmaven.test.skip=true
* 启动命令：java -jar \*\*\*\*.jar -Dspring.profiles.active=dev
* 停止命令：kill \*\*\*

echo "停止进程"

java\_process=`ps -ef |grep java |grep -v grep | awk '{print $2}'`

if [ -n "$java\_process" ]

then

ps -ef |grep java |grep -v grep| awk '{print $2}' | xargs kill

echo "kill java success"

fi

# 笔记

tcc try confirm cancel

aop：理解了aop 就理解tcc 就理解了分布式事务的实现

invoker去调用的 路由，集群，负载 filter

dubbo filter接口 在服务发起调用前走的，那么就可以做很多事情，就可以保存方法的所有的信息，方法签名，接口

要点：

1，aop根据添加的注解进行aop处理

2，rpc框架的特性，在调用前，会执行filter接口，会获取所调用的方法签名

3，threadlocal保存事务上下文，rpc框架的参数传递

4，事务日志，到底存储了什么东西？是怎么存储的？是怎么序列化的。

5，高效并发 jmeter压测

6，学一个框架开源，关键要学习里面代码的风格，设计模式，新的知识点

tcc的优缺点：

缺点：代码量多，使用中必须要知道confirm和cancel方法的正确书写。使用场景有限。需要记录事务日志：推荐使用高效的存储，推荐使用mongo集群 kroy序列化

优点：天然支持集群，不依赖事务，依赖事务日志，最终一致性