Seata AT 模式处理逻辑

客户端启动逻辑

全局事务由TM管理，TM由io.seata.tm.api.GlobalTransaction生成

**public** **interface** **GlobalTransaction** {

**void** **begin**() **throws** TransactionException;

**void** **begin**(**int** timeout) **throws** TransactionException;

**void** **begin**(**int** timeout, String name) **throws** TransactionException;

**void** **commit**() **throws** TransactionException;

**void** **rollback**() **throws** TransactionException;

GlobalStatus **getStatus**() **throws** TransactionException;

*// ...*

}

手动调用API时可以通过 GlobalTransactionContext 创建一个 GlobalTransaction

*//#initClient;*

TMClient.init(“applicationId”, “txServiceGroup”);

RMClient.init(”applicationId“, ”txServiceGroup“);

*//trx*

GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();

**try** {

tx.begin(60000, "testBiz");

*//* 事务处理

*// ...*

tx.commit();

} **catch** (Exception exx) {

tx.rollback();

**throw** exx;

}

在Spring工程中一般使用@GlobalTransactional注解代替上面的逻辑。SEATA则提供了GlobalTransactionScanner利用 Spring AOP 的特性，用模板模式把这些冗余代码封装模版里。

**public** **class** **GlobalTransactionScanner** **extends** **AbstractAutoProxyCreator** **implements** **InitializingBean**, **ApplicationContextAware**, **DisposableBean** {

*// ...*

}

GlobalTransactionScanner的初始化流程

【5:InitializingBean】阶段TMClient.init及RMClient.init的逻辑

* TMClient的初始化操作

io.seata.tm.TMClient#init

**public** **static** **void** **init**(String applicationId, String transactionServiceGroup) {

*//* 获取 *TmRpcClient* 实例

TmRpcClient tmRpcClient = TmRpcClient.getInstance(applicationId, transactionServiceGroup);

*//* 初始化 *TM Client*

tmRpcClient.init();

}

TmRpcClient#getInstance：

/\*\*

\* Gets instance.

\*

\* @return the instance

\*/

public static TmRpcClient getInstance() {

if (null == instance) {

synchronized (TmRpcClient.class) {

if (null == instance) {

// 创建 Netty 客户端配置文件对象

NettyClientConfig nettyClientConfig = new NettyClientConfig();

// 创建 messageExecutor 线程池，该线程池用于在处理各种与服务端的消息交互

final ThreadPoolExecutor messageExecutor = new ThreadPoolExecutor(

nettyClientConfig.getClientWorkerThreads(), nettyClientConfig.getClientWorkerThreads(),

KEEP\_ALIVE\_TIME, TimeUnit.SECONDS,

new LinkedBlockingQueue<>(MAX\_QUEUE\_SIZE),

new NamedThreadFactory(nettyClientConfig.getTmDispatchThreadPrefix(),

nettyClientConfig.getClientWorkerThreads()),

RejectedPolicies.runsOldestTaskPolicy());

instance = new TmRpcClient(nettyClientConfig, null, messageExecutor);

}

}

}

return instance;

}

\*TmRpcClient extends AbstractRpcRemotingClient

public AbstractRpcRemotingClient(NettyClientConfig nettyClientConfig, EventExecutorGroup eventExecutorGroup,

ThreadPoolExecutor messageExecutor, NettyPoolKey.TransactionRole transactionRole) {

super(messageExecutor);

this.transactionRole = transactionRole;

// 创建 ClientBootstrap，用于管理 Netty 服务的启停

clientBootstrap = new RpcClientBootstrap(nettyClientConfig, eventExecutorGroup, this, transactionRole);

// 专门用于管理 Netty 客户端对象池

clientChannelManager = new NettyClientChannelManager(

new NettyPoolableFactory(this, clientBootstrap), getPoolKeyFunction(), nettyClientConfig);

}

AbstractRpcRemotingClient#init

@Override

public void init() {

clientBootstrap.start();

// 定时重新发送 RegisterTMRequest（RM 客户端会发送 RegisterRMRequest）请求尝试连接服务端，具体逻辑是在 NettyClientChannelManager 中的 channels 中缓存了客户端 channel，如果此时 channels 不存在获取已过期，那么就会尝试连接服务端以重新获取 channel 并将其缓存到 channels 中

timerExecutor.scheduleAtFixedRate(new Runnable() {

@Override

public void run() {

clientChannelManager.reconnect(getTransactionServiceGroup());

}

}, SCHEDULE\_INTERVAL\_MILLS, SCHEDULE\_INTERVAL\_MILLS, TimeUnit.SECONDS);

if (NettyClientConfig.isEnableClientBatchSendRequest()) {

// 开启一条单独线程（MAX\_MERGE\_SEND\_THREAD=1），用于处理异步请求发送。\*会将同一个fescar-server的消息合并发送(减少netty通信次数)

mergeSendExecutorService = new ThreadPoolExecutor(MAX\_MERGE\_SEND\_THREAD,

MAX\_MERGE\_SEND\_THREAD,

KEEP\_ALIVE\_TIME, TimeUnit.MILLISECONDS,

new LinkedBlockingQueue<>(),

new NamedThreadFactory(getThreadPrefix(), MAX\_MERGE\_SEND\_THREAD));

mergeSendExecutorService.submit(new MergedSendRunnable());

}

super.init();

}

\*AbstractRpcRemotingClient extends AbstractRpcRemoting

io.seata.core.rpc.netty.AbstractRpcRemoting#init

**public** **void** **init**() {

// 用于定时清除已过期的 futrue，futures 是保存发送请求需要返回结果的 future 对象，该对象有个超时时间，过了超时时间就会自动抛异常，因此需要定时清除已过期的 future 对象

timerExecutor.scheduleAtFixedRate(**new** Runnable() {

**@Override**

**public** **void** **run**() {

**for** (Map.Entry<Integer, MessageFuture> entry : futures.entrySet()) {

**if** (entry.getValue().isTimeout()) {

futures.remove(entry.getKey());

entry.getValue().setResultMessage(**null**);

**if** (LOGGER.isDebugEnabled()) {

LOGGER.debug("timeout clear future: {}", entry.getValue().getRequestMessage().getBody());

}

}

}

nowMills = System.currentTimeMillis();

}

}, TIMEOUT\_CHECK\_INTERNAL, TIMEOUT\_CHECK\_INTERNAL, TimeUnit.MILLISECONDS);

}

* RMClient的初始化操作

io.seata.rm.RMClient#init

**public** **static** **void** **init**(String applicationId, String transactionServiceGroup) {

RmRpcClient rmRpcClient = RmRpcClient.getInstance(applicationId, transactionServiceGroup);

// ResourceManager 是 RM 资源管理器，负责分支事务的注册、提交、上报、以及回滚操作，以及全局锁的查询操作，DefaultResourceManager 会持有当前所有的 RM 资源管理器，进行统一调用处理。

// Seata中对ResourceManager，AbstractRMHandler用到了Java Spi拓展机制，会扫描 META-INF/services/ 目录下的配置类并进行动态加载。

// Java SPI 机制: 首先自定义接口。然后在含有实现类的Jar里的META-INF/services/下加入文件名和文件内容为实现类的全限定名的文件。使用具体实现的实例时则使用ServiceLoader进行初始化。

rmRpcClient.setResourceManager(DefaultResourceManager.get());

// RmMessageListener 是 RM 消息处理监听器，用于负责处理从 TC 发送过来的指令，并对分支进行分支提交、分支回滚，以及 undo log 删除操作

rmRpcClient.setClientMessageListener(**new** RmMessageListener(DefaultRMHandler.get()));

rmRpcClient.init();

}

【8:wrapIfNecessary】阶段生成Spring Bean的代理时的逻辑

※Spring在依赖注入完成之前的postProcessBeforeInitialization内和在依赖注入完成之后的postProcessAfterInitialization内会调用AbstractAutoProxyCreator的wrapIfNecessary来真正产生Bean的代理

io.seata.spring.annotation.GlobalTransactionScanner#wrapIfNecessary

**protected** Object **wrapIfNecessary**(Object bean, String beanName, Object cacheKey) {

*//* 判断是否有开启全局事务

**if** (disableGlobalTransaction) {

**return** bean;

}

**try** {

**synchronized** (PROXYED\_SET) {

**if** (PROXYED\_SET.contains(beanName)) {

**return** bean;

}

interceptor = **null**;

*//check TCC proxy*

**if** (TCCBeanParserUtils.isTccAutoProxy(bean, beanName, applicationContext)) {

*//TCC interceptor, proxy bean of sofa:reference/dubbo:reference, and LocalTCC*

interceptor = **new** TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));

} **else** {

Class<?> serviceInterface = SpringProxyUtils.findTargetClass(bean);

Class<?>[] interfacesIfJdk = SpringProxyUtils.findInterfaces(bean);

*//* 判断 *bean* 中是否有 *GlobalTransactional* 和 *GlobalLock* 注解

**if** (!existsAnnotation(**new** Class[]{serviceInterface})

&& !existsAnnotation(interfacesIfJdk)) {

**return** bean;

}

**if** (interceptor == **null**) {

*//* 创建*GlobalTransactionalInterceptor*代理类

interceptor = **new** GlobalTransactionalInterceptor(failureHandlerHook);

}

}

LOGGER.info("Bean[{}] with name [{}] would use interceptor [{}]",

bean.getClass().getName(), beanName, interceptor.getClass().getName());

**if** (!AopUtils.isAopProxy(bean)) {

bean = **super**.wrapIfNecessary(bean, beanName, cacheKey);

} **else** {

AdvisedSupport advised = SpringProxyUtils.getAdvisedSupport(bean);

*//* 执行包装目标对象到代理对象

*//* 用*GlobalTransactionalInterceptor* 代替被 *GlobalTransactional* 和 *GlobalLock* 注解的方法

Advisor[] advisor = **super**.buildAdvisors(beanName, getAdvicesAndAdvisorsForBean(**null**, **null**, **null**));

**for** (Advisor avr : advisor) {

advised.addAdvisor(0, avr);

}

}

PROXYED\_SET.add(beanName);

**return** bean;

}

} **catch** (Exception exx) {

**throw** **new** RuntimeException(exx);

}

}

io.seata.spring.annotation.GlobalTransactionalInterceptor#invoke

**public** Object **invoke**(**final** MethodInvocation methodInvocation) **throws** Throwable {

Class<?> targetClass = methodInvocation.getThis() != **null** ? AopUtils.getTargetClass(methodInvocation.getThis()) : **null**;

Method specificMethod = ClassUtils.getMostSpecificMethod(methodInvocation.getMethod(), targetClass);

**final** Method method = BridgeMethodResolver.findBridgedMethod(specificMethod);

**final** GlobalTransactional globalTransactionalAnnotation = getAnnotation(method, GlobalTransactional.class);

**final** GlobalLock globalLockAnnotation = getAnnotation(method, GlobalLock.class);

**if** (globalTransactionalAnnotation != **null**) {

*//* 全局事务注解

*//* 此处执行了*TransactionalTemplate* 模版类的 *execute* 方法

**return** handleGlobalTransaction(methodInvocation, globalTransactionalAnnotation);

} **else** **if** (globalLockAnnotation != **null**) {

*//* 全局锁注解

**return** handleGlobalLock(methodInvocation);

} **else** {

**return** methodInvocation.proceed();

}

}

io.seata.tm.api.TransactionalTemplate#execute

**public** Object **execute**(TransactionalExecutor business) **throws** Throwable {

*// 1. get or create a transaction*

GlobalTransaction tx = GlobalTransactionContext.getCurrentOrCreate();

*// 1.1 get transactionInfo*

TransactionInfo txInfo = business.getTransactionInfo();

**if** (txInfo == **null**) {

**throw** **new** ShouldNeverHappenException("transactionInfo does not exist");

}

**try** {

*// 2. begin transaction*

beginTransaction(txInfo, tx);

Object rs = **null**;

**try** {

*// Do Your Business*

rs = business.execute();

} **catch** (Throwable ex) {

*// 3.the needed business exception to rollback.*

completeTransactionAfterThrowing(txInfo,tx,ex);

**throw** ex;

}

*// 4. everything is fine, commit.*

commitTransaction(tx);

**return** rs;

} **finally** {

*//5. clear*

triggerAfterCompletion();

cleanUp();

}

}

心得

阅读源码加深了对框架机制的理解

次回：服务端处理逻辑分析

参考

<http://seata.io/zh-cn/blog/seata-at-mode-start.html>

<https://gitlab.wxbpdapp.com/root/seata/blob/master/core/src/main/java/io/seata/core/rpc/netty/AbstractRpcRemotingClient.java>

<https://blog.csdn.net/hosaos/article/details/89403552>

<https://gitlab.wxbpdapp.com/root/seata/blob/master/core/src/main/java/io/seata/core/rpc/netty/AbstractRpcRemoting.java>

<https://crossoverjie.top/2020/02/24/wheel/cicada8-spi/>

<https://gitlab.wxbpdapp.com/root/seata/blob/master/spring/src/main/java/io/seata/spring/annotation/GlobalTransactionScanner.java>

<https://www.cnblogs.com/micrari/p/7552571.html>

<https://juejin.im/post/6844903788881559566>