

# 分布式事务框架Seata深入剖析与应用实践 集成剖析篇



主讲人：陈东

2020.7.16

- Seata框架与Spring集成应用实践
- Seata框架与Dubbo集成应用实践



## 01. Seata框架与Spring集成剖析

---

## Spring模块剖析

### ➤ 作用

- 借助 spring扩展点对代理的bean进行操作
- 生成数据源代理类，并自动代理；
- 对被全局事务注解的bean 织入不同事务模式对应的 advisor 实现类。

### ➤ 核心类

- AutoDataSourceProxyRegistrar
- GlobalTransactionScanner
- GlobalTransactionalInterceptor
- TccActionInterceptor

```
> seata-spring (in spring) [seata 1.2.0]
  > src/main/java
    > io.seata.spring
      > annotation
        > datasource
          > AutoDataSourceProxyRegistrar.java
          > DataSourceProxyHolder.java
          > EnableAutoDataSourceProxy.java
          > SeataAutoDataSourceProxyAdvice.java
          > SeataAutoDataSourceProxyCreator.java
          > SeataProxy.java
          > GlobalLock.java
          > GlobalTransactional.java
          > GlobalTransactionalInterceptor.java
          > GlobalTransactionScanner.java
          > MethodDesc.java
        > tcc
      > util
        > SpringProxyUtils.java
        > TCCBeanParserUtils.java
```

## Spring模块剖析

- **AutoDataSourceProxyRegistrar**
  - 数据源自动代理注册器
- **SeataAutoDataSourceProxyCreator**
  - 用于实例初始化时织入拦截器
- **SeataAutoDataSourceProxyAdvice**
  - 数据源代理拦截器

```
public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
    BeanDefinitionRegistry registry) {
    if (!registry.containsBeanDefinition(BEAN_NAME_SEATA_AUTO_DATA_SOURCE_PROXY_CREATOR))
    {
        boolean useJdkProxy =
            Boolean.parseBoolean(importingClassMetadata.getAnnotationAttributes(EnableAutoDataSourceProxy.class.getName()).get(ATTRIBUTE_KEY_USE_JDK_PROXY).toString());
        String[] excludes = (String[])
            importingClassMetadata.getAnnotationAttributes(EnableAutoDataSourceProxy.class.getName()).get(ATTRIBUTE_KEY_EXCLUDES);
        AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder
            .genericBeanDefinition(SeataAutoDataSourceProxyCreator.class)
            .addConstructorArgValue(useJdkProxy)
            .addConstructorArgValue(excludes)
            .getBeanDefinition();
        registry.registerBeanDefinition(BEAN_NAME_SEATA_AUTO_DATA_SOURCE_PROXY_CREATOR,
            beanDefinition);
    }
}
```

**BEAN\_NAME\_SEATA\_AUTO\_DATA\_SOURCE\_PROXY\_CREATOR = "seataAutoDataSourceProxyCreator"**

## Spring模块剖析

- AutoDataSourceProxyRegistrar
  - 数据源自动代理注册器
- SeataAutoDataSourceProxyCreator
  - 用于实例初始化时织入拦截器
- SeataAutoDataSourceProxyAdvice
  - 数据源代理拦截器

```
public class SeataAutoDataSourceProxyCreator extends AbstractAutoProxyCreator {  
    private static final Logger LOGGER =  
        LoggerFactory.getLogger(SeataAutoDataSourceProxyCreator.class);  
    private final String[] excludes;  
    private final Advisor advisor = new DefaultIntroductionAdvisor(new  
        SeataAutoDataSourceProxyAdvice());  
  
    //.....  
  
    @Override  
    protected Object[] getAdvisesAndAdvisorsForBean(Class<?> beanClass, String beanName,  
        TargetSource customTargetSource) throws BeansException {  
        if (LOGGER.isInfoEnabled()) {  
            LOGGER.info("Auto proxy of [{}]", beanName);  
        }  
        return new Object[]{advisor};  
    }  
    //.....  
}
```

## Spring模块剖析

- AutoDataSourceProxyRegistrar
  - 数据源自动代理注册器
- SeataAutoDataSourceProxyCreator
  - 用于实例初始化时织入拦截器
- SeataAutoDataSourceProxyAdvice
  - 数据源代理拦截器

```
public class SeataAutoDataSourceProxyAdvice implements MethodInterceptor,
IntroductionInfo {
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        DataSourceProxy dataSourceProxy =
DataSourceProxyHolder.get().putDataSource((DataSource) invocation.getThis());
        Method method = invocation.getMethod();
        Object[] args = invocation.getArguments();
        Method m = BeanUtils.findDeclaredMethod(DataSourceProxy.class, method.getName(),
method.getParameterTypes());
        if (m != null) {
            return m.invoke(dataSourceProxy, args);
        } else {
            return invocation.proceed();
        }
    }
    @Override
    public Class<?>[] getInterfaces() {
        return new Class[]{SeataProxy.class};
    }
}
```



## Spring模块剖析

### ➤ GlobalTransactionScanner--全局事务扫描器

- **TM、RM的初始化**
- 注册销毁时调用的钩子
- 实例初始化时织入拦截器

```
private void initClient() {
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Initializing Global Transaction Clients ... ");
    }
    if (StringUtils.isNullOrEmpty(applicationId) ||
        StringUtils.isNullOrEmpty(txServiceGroup)) {
        throw new IllegalArgumentException(String.format("applicationId: %s, txServiceGroup: %s", applicationId, txServiceGroup));
    }
    //init TM
    TMClient.init(applicationId, txServiceGroup);
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Transaction Manager Client is initialized. applicationId[{}] txServiceGroup[{}]", applicationId, txServiceGroup);
    }
    //init RM
    RMClient.init(applicationId, txServiceGroup);
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Resource Manager is initialized. applicationId[{}] txServiceGroup[{}]", applicationId, txServiceGroup);
    }

    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Global Transaction Clients are initialized. ");
    }
    registerSpringShutdownHook();
}
```



## Spring模块剖析

### ➤ GlobalTransactionScanner--全局事务扫描器

- TM、RM的初始化
- **注册销毁时调用的钩子**
- 实例初始化时织入拦截器

```
private void registerSpringShutdownHook() {  
    if (applicationContext instanceof ConfigurableApplicationContext) {  
        ((ConfigurableApplicationContext) applicationContext).registerShutdownHook();  
        ShutdownHook.removeRuntimeShutdownHook();  
    }  
    ShutdownHook.getInstance().addDisposable(TmRpcClient.getInstance(applicationId,  
txServiceGroup));  
    ShutdownHook.getInstance().addDisposable(RmRpcClient.getInstance(applicationId,  
txServiceGroup));  
}
```

## Spring模块剖析

### ➤ GlobalTransactionScanner--全局事务扫描器

- TM、RM的初始化
- 注册销毁时调用的钩子
- **实例初始化时织入拦截器**

**@TwoPhaseBusinessAction**  
**@GlobalTransactional**  
**@GlobalLock**

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
    if (disableGlobalTransaction) {
        return bean;
    }
    try {
        synchronized (PROXYED_SET) {
            if (PROXYED_SET.contains(beanName)) {
                return bean;
            }
            interceptor = null;
            //检查是否TCC代理类
            if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName, applicationContext)) {
                interceptor = new
TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
            } else {
                Class<?> serviceInterface = SpringProxyUtils.findTargetClass(bean);
                Class<?>[] interfacesIfJdk = SpringProxyUtils.findInterfaces(bean);
                //检查是否存在@GlobalTransactional或者@GlobalLock注解
                if (!existsAnnotation(new Class[]{serviceInterface})
                    && !existsAnnotation(interfacesIfJdk)) {
                    return bean;
                }
            }
        }
    }
}
```

## Spring模块剖析

### ➤ GlobalTransactionScanner--全局事务扫描器

- TM、RM的初始化
- 注册销毁时调用的钩子
- **实例初始化时织入拦截器**

```
if (interceptor == null) {
    if (globalTransactionalInterceptor == null) {
        //创建全局事务拦截器
        globalTransactionalInterceptor = new
        GlobalTransactionalInterceptor(failureHandlerHook);
        ConfigurationCache.addConfigListener(
            ConfigurationKeys.DISABLE_GLOBAL_TRANSACTION,
            (ConfigurationChangeListener)globalTransactionalInterceptor);
    }
    interceptor = globalTransactionalInterceptor;
}
}
if (!AopUtils.isAopProxy(bean)) {
    bean = super.wrapIfNecessary(bean, beanName, cacheKey);
} else {
    AdvisedSupport advised = SpringProxyUtils.getAdvisedSupport(bean);
    Advisor[] advisor = buildAdvisors(beanName,
    getAdvicesAndAdvisorsForBean(null, null, null));
    //织入拦截类
    for (Advisor avr : advisor) {
        advised.addAdvisor(0, avr);
    }
}
PROXYED_SET.add(beanName);
return bean;
}
} catch (Exception exx) {
    throw new RuntimeException(exx);
}
}
```

## Spring模块剖析

- 全局事务
- TCC



## Spring模块剖析

- GlobalTransactionalInterceptor -- 事务拦截器
  - 拦截处理事务注解
- GlobalTransactional
  - 开启全局事务，可以自定义超时时间、全局事务的名字、回滚时调用的类
- GlobalLock
  - 开启数据资源全局锁，申明操作数据资源也需要事务统一管理，保证资源隔离性。



## Spring模块剖析

```
public Object invoke(final MethodInvocation methodInvocation) throws Throwable {
    // 获取代理类
    Class<?> targetClass = methodInvocation.getThis() != null ? AopUtils.getTargetClass(methodInvocation.getThis()) : null;
    Method specificMethod = ClassUtils.getMostSpecificMethod(methodInvocation.getMethod(), targetClass);
    final Method method = BridgeMethodResolver.findBridgedMethod(specificMethod);
    // 获取拦截方法注解
    final GlobalTransactional globalTransactionalAnnotation = getAnnotation(method, GlobalTransactional.class);
    final GlobalLock globalLockAnnotation = getAnnotation(method, GlobalLock.class);
    if (!disable && globalTransactionalAnnotation != null) {
        // 执行全局事务处理
        return handleGlobalTransaction(methodInvocation, globalTransactionalAnnotation);
    } else if (!disable && globalLockAnnotation != null) {
        // 执行本地事务，但操作资源需竞争全局锁，保证数据隔离性
        return handleGlobalLock(methodInvocation);
    } else {
        return methodInvocation.proceed();
    }
}
```



## Spring模块剖析

➤ GlobalTransactional

➤ GlobalLock

```
private Object handleGlobalTransaction(final MethodInvocation methodInvocation, final GlobalTransactional globalTrxAnno)
throws Throwable {
    try {
        return transactionalTemplate.execute(new TransactionalExecutor() {
            public Object execute()
            throws Throwable {
                return methodInvocation.proceed();
            }
            public String name() {
                String name = globalTrxAnno.name();
                if (!StringUtils.isEmpty(name)) {
                    return name;
                }
                return formatMethod(methodInvocation.getMethod());
            }
            public TransactionInfo getTransactionInfo() {
                TransactionInfo transactionInfo = new TransactionInfo();
                transactionInfo.setTimeout(globalTrxAnno.timeoutMills());
                transactionInfo.setName(name());
                transactionInfo.setPropagation(globalTrxAnno.propagation());
                Set<RollbackRule> rollbackRules = new LinkedHashSet<>();
                for (Class<?> rbRule : globalTrxAnno.rollbackFor()) {
                    rollbackRules.add(new RollbackRule(rbRule));
                }
                for (String rbRule : globalTrxAnno.rollbackForClassName()) {
                    rollbackRules.add(new RollbackRule(rbRule));
                }
                for (Class<?> rbRule : globalTrxAnno.noRollbackFor()) {
                    rollbackRules.add(new NoRollbackRule(rbRule));
                }
                for (String rbRule : globalTrxAnno.noRollbackForClassName()) {
                    rollbackRules.add(new NoRollbackRule(rbRule));
                }
                transactionInfo.setRollbackRules(rollbackRules);
                return transactionInfo;
            }
        });
    }
}
```

## Spring模块剖析

➤ GlobalTransactional

➤ GlobalLock

```
private Object handleGlobalLock(final MethodInvocation methodInvocation) throws Exception
{
    return globalLockTemplate.execute(() -> {
        try {
            return methodInvocation.proceed();
        } catch (Exception e) {
            throw e;
        } catch (Throwable e) {
            throw new RuntimeException(e);
        }
    });
}
```

## Spring模块剖析

- TccActionInterceptor
  - TCC事务模型拦截器
  - 一个环绕通知，调用TCC处理器；
- ActionInterceptorHandler
  - 生成TCC运行时上下文、透传业务参数、注册分支事务记录

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
    if (disableGlobalTransaction) {
        return bean;
    }
    try {
        synchronized (PROXYED_SET) {
            if (PROXYED_SET.contains(beanName)) {
                return bean;
            }
            interceptor = null;
            //检查是否TCC代理类
            if (TCCBeanParserUtils.isTccAutoProxy(bean, beanName, applicationContext)) {
                interceptor = new
TccActionInterceptor(TCCBeanParserUtils.getRemotingDesc(beanName));
            } else {
                Class<?> serviceInterface = SpringProxyUtils.findTargetClass(bean);
                Class<?>[] interfacesIfJdk = SpringProxyUtils.findInterfaces(bean);
                //检查是否存在@GlobalTransactional或者@GlobalLock注解
                if (!existsAnnotation(new Class[]{serviceInterface})
                    && !existsAnnotation(interfacesIfJdk)) {
                    return bean;
                }
            }
        }
    }
}
```

## Spring模块剖析

- 全局事务
- **TCC**



## Spring模块剖析

- TccActionInterceptor
  - TCC事务模型拦截器
  - 一个环绕通知，调用TCC处理器；
- ActionInterceptorHandler
  - 生成TCC运行时上下文、透传业务参数、注册分支事务记录

```
public static boolean isTccAutoProxy(Object bean, String beanName, ApplicationContext
applicationContext) {
    boolean isRemotingBean = parserRemotingServiceInfo(bean, beanName);
    //get RemotingBean description
    RemotingDesc remotingDesc =
DefaultRemotingParser.get().getRemotingBeanDesc(beanName);
    //is remoting bean
    if (isRemotingBean) {
        if (remotingDesc != null && remotingDesc.getProtocol() == Protocols.IN_JVM) {
            //LocalTCC
            return isTccProxyTargetBean(remotingDesc);
        } else {
            // sofa:reference / dubbo:reference, factory bean
            return false;
        }
    } else {
        if (remotingDesc == null) {
            //check FactoryBean
            if (isRemotingFactoryBean(bean, beanName, applicationContext)) {
                remotingDesc = DefaultRemotingParser.get().getRemotingBeanDesc(beanName);
                return isTccProxyTargetBean(remotingDesc);
            } else {
                return false;
            }
        } else {
            return isTccProxyTargetBean(remotingDesc);
        }
    }
}
```



## Spring模块剖析

- TccActionInterceptor
  - TCC事务模型拦截器
  - 一个环绕通知，调用TCC处理器；
- ActionInterceptorHandler
  - 生成TCC运行时上下文、透传业务参数、注册分支事务记录

```
public RemotingDesc parserRemotingServiceInfo(Object bean, String beanName,
RemotingParser remotingParser) {
    //.....
    if (remotingParser.isService(bean, beanName)) {
        try {
            Object targetBean = remotingBeanDesc.getTargetBean();
            for (Method m : methods) {
                TwoPhaseBusinessAction twoPhaseBusinessAction =
m.getAnnotation(TwoPhaseBusinessAction.class);
                if (twoPhaseBusinessAction != null) {
                    TCCResource tccResource = new TCCResource();
                    tccResource.setActionName(twoPhaseBusinessAction.name());
                    tccResource.setTargetBean(targetBean);
                    tccResource.setPrepareMethod(m);
                    tccResource.setCommitMethodName(...);
                    //.....
                    //registry tcc resource
                    DefaultResourceManager.get().registerResource(tccResource);
                }
            }
        } catch (Throwable t) {
            throw new FrameworkException(t, "parser remoting service error");
        }
    }
    //.....
    return remotingBeanDesc;
}
```



## Spring模块剖析

```
public class TCCResourceManager extends AbstractResourceManager {  
  
    private Map<String, Resource> tccResourceCache = new ConcurrentHashMap<>();  
    public TCCResourceManager() {  
    }  
  
    @Override  
    public void registerResource(Resource resource) {  
        TCCResource tccResource = (TCCResource)resource;  
        tccResourceCache.put(tccResource.getResourceId(), tccResource);  
        super.registerResource(tccResource);  
    }  
}
```



## 02. Seata框架与Dubbo集成剖析

---

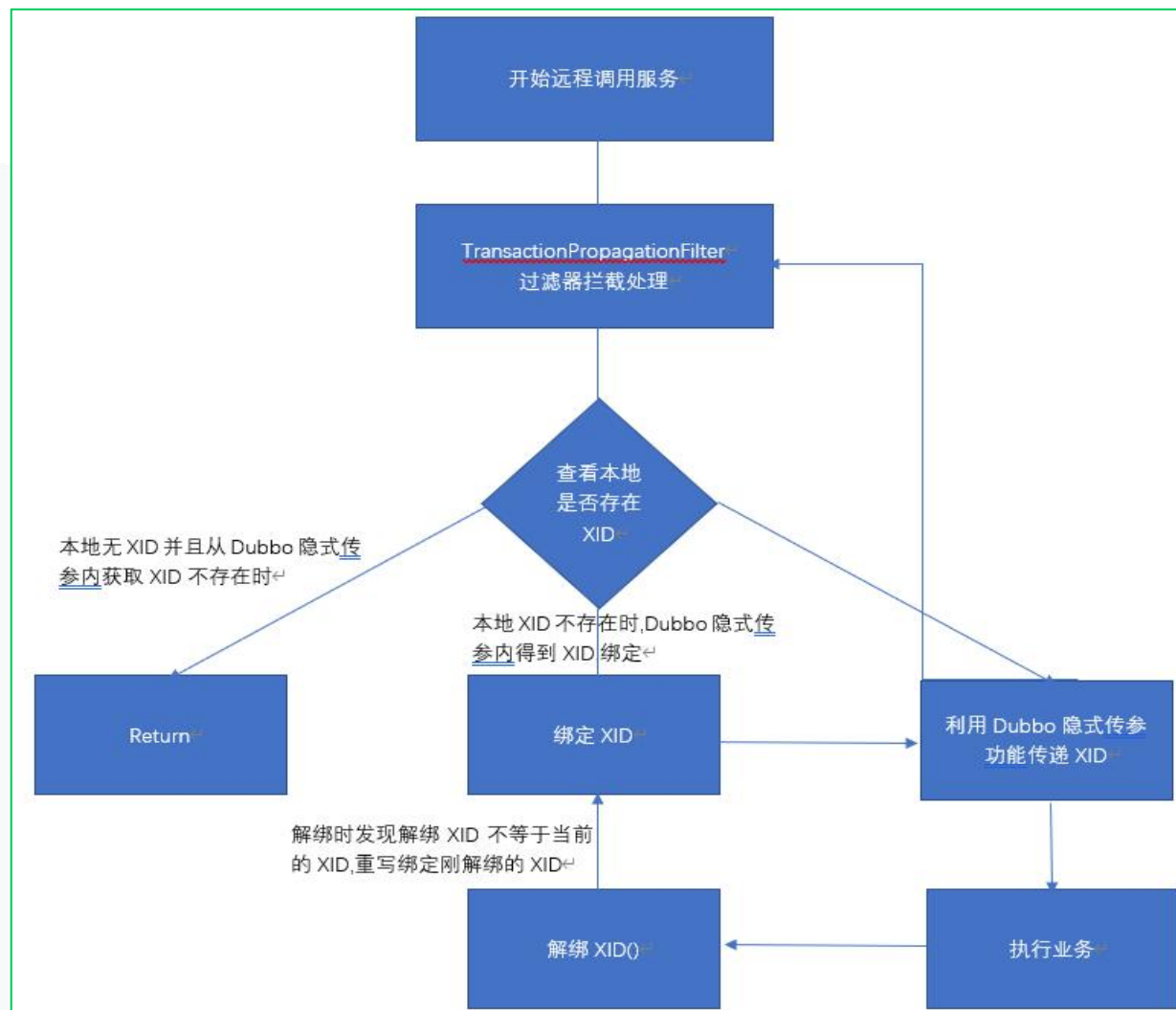
## Integration模块剖析

- ApacheDubboTransactionPropagationFilter
  - Dubbo过滤器，实现了Dubbo 应用的 Seata 全局事务的传播。
- 服务消费者
  - 发起 Dubbo 远程调用时，将 Seata 全局事务 XID 通过隐式参数传递
- 服务提供者
  - 收到 Dubbo 远程调用时，从隐式参数中解析出 Seata 全局事务 XID。

## Integration模块剖析

```
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获取本地XID
    String xid = RootContext.getXID();
    String xidInterceptorType = RootContext.getXIDInterceptorType();
    // 获取Dubbo隐式传参中的XID
    String rpcXid = getRpcXid();
    String rpcXidInterceptorType = RpcContext.getContext().getAttachment(RootContext.KEY_XID_INTERCEPTOR_TYPE);
    boolean bind = false;
    if (xid != null) { // 传递XID
        RpcContext.getContext().setAttachment(RootContext.KEY_XID, xid);
        RpcContext.getContext().setAttachment(RootContext.KEY_XID_INTERCEPTOR_TYPE, xidInterceptorType);
    } else {
        if (rpcXid != null) {
            // 绑定XID
            RootContext.bind(rpcXid);
            RootContext.bindInterceptorType(rpcXidInterceptorType);
            bind = true;
        }
    }
    try {
        return invoker.invoke(invocation);
    } finally {
        if (bind) {
            // 进行剔除已完成事务的XID
            String unbindInterceptorType = RootContext.unbindInterceptorType();
            String unbindXid = RootContext.unbind();
            // 如果发现解绑的XID并不是当前接收到的XID
            if (!rpcXid.equalsIgnoreCase(unbindXid)) {
                LOGGER.warn("xid in change during RPC from {} to {}, xidInterceptorType from {} to {} ", rpcXid, unbindXid, rpcXidInterceptorType,
                    unbindInterceptorType);
                if (unbindXid != null) {
                    // 重新绑定XID
                    RootContext.bind(unbindXid);
                    RootContext.bindInterceptorType(unbindInterceptorType);
                    LOGGER.warn("bind [{}] interceptorType[{}] back to RootContext", unbindXid, unbindInterceptorType);
                }
            }
        }
    }
}
```

## Integration模块剖析



## TCC相关剖析

- DubboRemotingParser
  - 将Dubbo 代理类 解析成RemotingDesc。
- RemotingDesc
  - 事务流程需要的远程 bean 的一些具体信息



## TCC相关剖析

```
@Override
public RemotingDesc getServiceDesc(Object bean, String beanName) throws FrameworkException {
    if (!this.isRemoting(bean, beanName)) {
        return null;
    }
    try {
        RemotingDesc serviceBeanDesc = new RemotingDesc();
        Class<?> interfaceClass = (Class<?>)ReflectionUtil.invokeMethod(bean, "getInterfaceClass");
        String interfaceClassName = (String)ReflectionUtil.getFieldValue(bean, "interfaceName");
        String version = (String)ReflectionUtil.invokeMethod(bean, "getVersion");
        String group = (String)ReflectionUtil.invokeMethod(bean, "getGroup");
        serviceBeanDesc.setInterfaceClass(interfaceClass);
        serviceBeanDesc.setInterfaceClassName(interfaceClassName);
        serviceBeanDesc.setUniqueId(version);
        serviceBeanDesc.setGroup(group);
        serviceBeanDesc.setProtocol(Protocols.DUBBO);
        if (isService(bean, beanName)) {
            Object targetBean = ReflectionUtil.getFieldValue(bean, "ref");
            serviceBeanDesc.setTargetBean(targetBean);
        }
        return serviceBeanDesc;
    } catch (Throwable t) {
        throw new FrameworkException(t);
    }
}
```



## 03. 测试题讲解

---

### 一、以下哪些场景会出现分布式事务需求

- A、跨数据库分布式事务
- B、跨服务分布式事务
- C、混合式分布式事务：跨数据库分布式事务 + 跨服务分布式事务。
- D、上述场景都可以分布式事务需求。

答案：A、B、C、D

### 二、柔性事务对事务的 ACID 特性的支持情况如下

- A、完全支持原子性。
- B、提供最终一致性支持，因为执行操作和补偿操作都是独立的事务。
- C、不完全保证隔离性，存在数据脏读现象。
- D、完全支持持久性。

答案：A、B、C、D

### 三、关于Seata AT 模型的资源隔离说法错误的有哪些

- A、写隔离：Commit时，利用全局锁实现并发线程顺序串行
- B、写隔离：Rollback时，利用全局锁实现并发线程逆序串行
- C、读隔离：利用for update语法打标识，检测是否存在全局锁进行等待阻塞
- D、上述说法都是错误的

答案：D

### 四、补偿型事务因为补偿行为的独立化，因此需要遵循以下哪些原则：

- A、补偿方法允许空补偿
- B、补偿事务所有步骤都需要保持幂等性
- C、补偿方法防止资源悬挂
- D、补偿事务需要自处理数据隔离。

答案：A、B、C、D

### 五、Seata支持哪些事务模型

- A、AT事务模型
- B、XA事务模型
- C、TCC事务模型
- D、Saga 拦截器事务模型

答案：A、B、C



# NiX 奈学教育

---



欢迎关注本人公众号  
“架构之美”