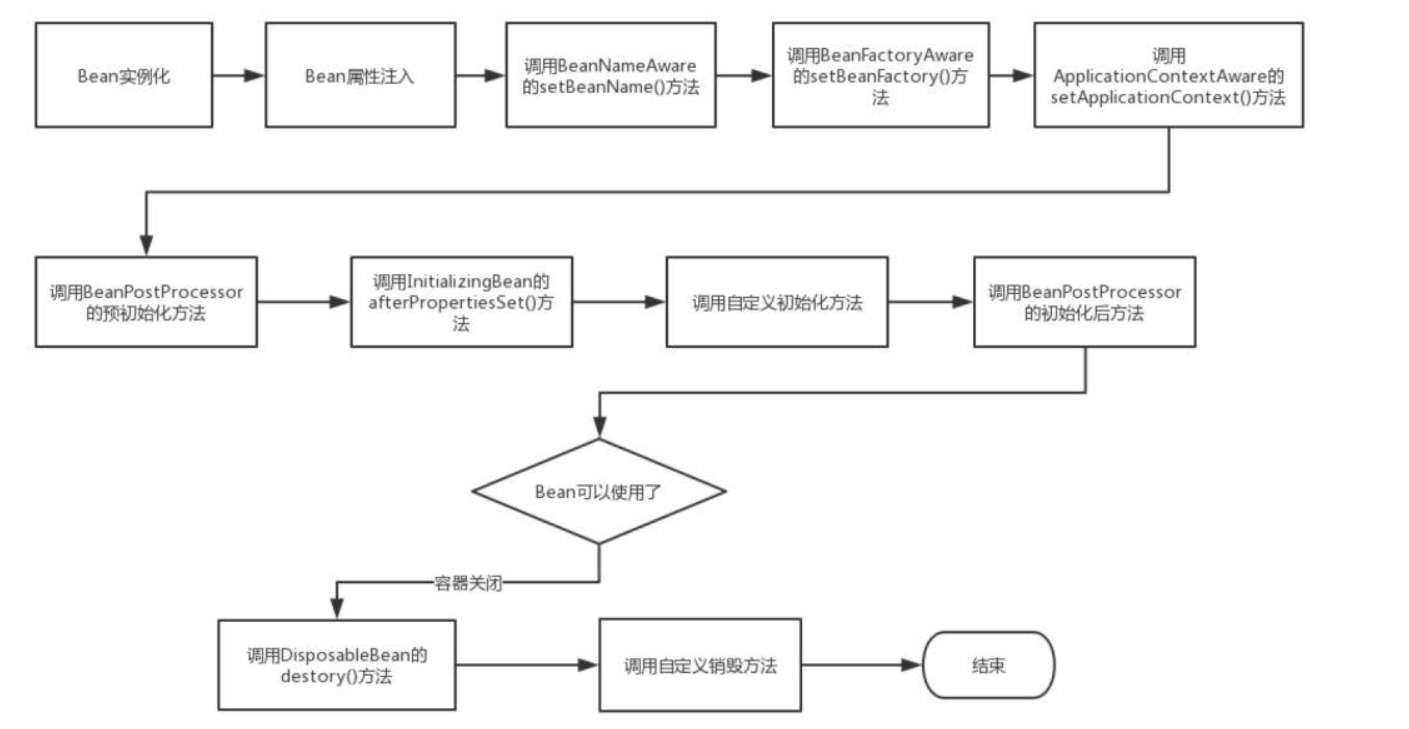
Bean的生命周期



Seata原理

1通过springboot自启动原理会加载所有jar包下的spring.factories文件读取seata的自启动类grobalTrasationalScanner的全限定类名

2反射加载grobalTrasationalScanner进行seata的初始化

3grobalTrasationalScanner继承了AbstractAutoProxyCreator类，这个类是aop的代理类并且他的实现了beanPostProcess接口，并且grobalTrasationalScanner实现了InitializingBean接口

4beanPostProcess的postProcessAfterInitialization方法里调用warpIfNessary方法校验所有经过的bean的方法上是否携带@grobalTranTional注解，如果是则采用方法拦截器MethodInterceptor对这个方法进行增强

5InitializingBean的afterPropertiesSet方法进行TM和RM的生成，并且注册到TC至此初始化过程结束

6当有人访问需要全局事务的方法MethodInterceptor就会生效会创建全局事务和分支事务，然后执行业务代码，然后通过feign携带全局事务id调用其他服务的方法

7其他服务的代理数据源接到调用后也会开启分支事务，然后执行，让后将执行结果告知tc，

8tm发起全局事务提交，或者全局事务回滚

TM如何得知需要提交还是回滚

通过观察者模式观察整个事务的变化。

观察者模式理解

1创建被观察类subject，subject类里面需要维护所有的观察者对象

public class Subject {  
//创建list里面维护着所有的观察者  
 private List<Observer> observers  
 = new ArrayList<Observer>();  
 private int state;  
  
 public int getState() {  
 return state;  
 }  
//改变被观察的字段status的时候要调用观察者的update方法，告知观察者被观察的对象改变了  
 public void setState(int state) {  
 this.state = state;  
 notifyAllObservers();  
 }  
  
 public void attach(Observer observer){  
 observers.add(observer);  
 }  
//调用观察者的update方法，让观察者做出改变  
 public void notifyAllObservers(){  
 for (Observer observer : observers) {  
 observer.update();  
 }  
 }  
}

2创建观察者，顶层抽象类

public abstract class Observer {  
 protected Subject subject;  
 public abstract void update();  
}

创建管着者1

public class BinaryObserver extends Observer{  
  
 public BinaryObserver(Subject subject){  
 this.subject = subject;  
 this.subject.attach(this);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println( "Binary String: "  
 + Integer.*toBinaryString*( subject.getState() ) );  
 }  
}

创建观察者2

public class HexaObserver extends Observer{  
  
 public HexaObserver(Subject subject){  
 this.subject = subject;  
 this.subject.attach(this);  
 }  
  
 @Override  
 public void update() {  
 System.*out*.println( "Hex String: "  
 + Integer.*toHexString*( subject.getState() ).toUpperCase() );  
 }  
}

3将观察者绑定到被观察者的队列上，那么被观察着改变的时候就可以通知观察者做出响应。

public class ObserverPatternDemo {  
 public static void main(String[] args) {  
 Subject subject = new Subject();  
  
 new HexaObserver(subject);  
 new OctalObserver(subject);  
 new BinaryObserver(subject);  
  
 System.*out*.println("First state change: 15");  
 subject.setState(15);  
 System.*out*.println("Second state change: 10");  
 subject.setState(10);  
 }  
}