

1项目简介

2 错误日志

3 循环依赖错误原因

3.1 错误调试发现步骤

3.1.1 创建 sqlSessionFactory

3.1.2 创建 fescarDataSource

3.1.3 创建 druidDataSource

3.1.4 关键：调用 DataSourceInitializerPostProcessor#postProcessAfterInitialization

3.1.5 这一步出错！：调用 DataSourceInitializerInvoker#afterPropertiesSet

4 三种解决方案

4.1 移除数据源自动配置类

4.2 调整 beanDefinition 初始化的顺序就可以避免循环依赖

4.3 初始化主数据源的时候不要依赖其它数据源

5 总结

参考

1项目简介

- 集成 fescar 做分布式事务一致性实验
- 首先 zookeeper 和 fescar-server 已经开启了
- 该模块是 fescar-client，需要自定义数据源
- fescar.version: 0.1.3

```
//fescar数据源配置
@Configuration
public class DruidConfig {
    //配置一个DruidDataSource：必须是DruidDataSource
    @Bean("druidDataSource")
    @ConfigurationProperties(prefix = "spring.datasource")
    public DruidDataSource druidDataSource() {
        DruidDataSource druidDataSource = new DruidDataSource();
        return druidDataSource;
    }
    //这是fescar需要的dataSource：其中DataSourceProxy的参数必须是DruidDataSource
    @Primary
    @Bean("fescarDataSource")
    public DataSource dataSource(DruidDataSource druidDataSource) {
        DataSourceProxy dataSourceProxy = new DataSourceProxy(druidDataSource);
        return dataSourceProxy;
    }
}
```

2 错误日志

```

accountController (field com.wangkang.mapper.AccountMapper com.wangkang.controller.AccountController.accountMapper)
    ↓
accountMapper defined in file [D:\Download\github\fescar-test\account-service\target\classes\com\wangkang\mapper\AccountMapper.class]
    ↓
sqlSessionFactory defined in class path resource [org/mybatis/spring/boot/autoconfigure/MybatisAutoConfiguration.class]
-----
fescarDataSource defined in class path resource [com/wangkang/config/DruidConfig.class]
    ↓
druidDataSource defined in class path resource [com/wangkang/config/DruidConfig.class]
    ↓
org.springframework.boot.autoconfigure.jdbc.DataSourceInitializerInvoker

```

<https://blog.csdn.net/kangsa998>

3 循环依赖错误原因

- `DataSourceAutoConfiguration` 中注册了 `dataSourceInitializerPostProcessor` 后置处理器
- 它在第一次初始化数据源时会被触发，会去初始化**主数据源**，如果第一次初始化的是主数据源，就会导致数据源初始化循环依赖

3.1 错误调试发现步骤

3.1.1 创建 sqlSessionFacotry

```
@Bean
@ConditionalOnMissingBean
//关键:构造器注入dataSource , 这里注入的是fescarDataSource(因为被@Primay注解了)
public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {}
```

3.1.2 创建 fescarDataSource

```
@Primary
@Bean("fescarDataSource")
//需要递归创建 druidDataSource
public DataSource dataSource(DruidDataSource druidDataSource) {
    DataSourceProxy dataSourceProxy = new DataSourceProxy(druidDataSource);
    return dataSourceProxy;
}
```

3.1.3 创建 druidDataSource

```
@Bean("druidDataSource")
@ConfigurationProperties(prefix = "spring.datasource")
public DruidDataSource druidDataSource() {
    DruidDataSource druidDataSource = new DruidDataSource();
    return druidDataSource;
}
```

3.1.4 关键：调用

DataSourceInitializerPostProcessor#postProcessAfterInitialization

```
//调用的原因：DataSourceAutoConfiguration中注入的DataSourceInitializerPostProcessor
@Override
public Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException {
    if (bean instanceof DataSource) {
        //第一次调用的时候会去初始化DataSourceInitializerInvoker
        this.beanFactory.getBean(DataSourceInitializerInvoker.class);
    }
    return bean;
}
```

3.1.5 这一步出错! : 调用 DataSourceInitializerInvoker#afterPropertiesSet

```
@Override
//初始化DataSourceInitializerInvoker, 要调用这个方法
public void afterPropertiesSet() {
    DataSourceInitializer initializer = getDataSourceInitializer();
    if (initializer != null) {
        boolean schemaCreated = this.dataSourceInitializer.createSchema();
        if (schemaCreated) {
            initialize(initializer);
        }
    }
}

private DataSourceInitializer getDataSourceInitializer() {
    if (this.dataSourceInitializer == null) {
        //最终这里报错, 这里又调用fescarDataSource!!! 产生循环依赖
        DataSource ds = this.dataSource.getIfUnique();
        if (ds != null) {
            this.dataSourceInitializer = new DataSourceInitializer(ds,
                                                                    this.properties,
                                                                    this.applicationContext);
        }
    }
    return this.dataSourceInitializer;
}
```

4 三种解决方案

4.1 移除数据源自动配置类

- 移除之后没什么影响, 只是不能自动执行 `sql` 配置文件, 完全可以把执行 `sql` 配置文件的代码抽取出来, 做个定时任务

```
//关键是: exclude = DataSourceAutoConfiguration.class
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
public class AccountServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(AccountServiceApplication.class, args);
    }
}
```

4.2 调整 beanDefinition 初始化的顺序就可以避免循环依赖

解决方式： 在任意一个配置类中，添加一个不是主数据源的依赖

```
@Configuration
public class DruidConfig {
    //加了一下依赖，就不会产生循环依赖
    @Resource(name = "druidDataSource")
    DataSource druidDataSource;
}
```

思路解析：

- 打断点找到所有beanDefinition列表

```
public void refresh() {
    finishBeanFactoryInitialization(beanFactory);
}
protected void finishBeanFactoryInitialization(beanFactory){
    beanFactory.preInstantiateSingletons();
}
public void preInstantiateSingletons() {
    //在这里打断点，查看beanDefinition列表
    List<String> beanNames = new ArrayList<>(this.getBeanDefinitionNames);
    //这里就开始遍历初始化bean了
    for (String beanName : beanNames) {}
}
```

- [有循环依赖的代码](#)

- 下图是有循环依赖的所有beanDefinition列表

```

▼ this.beanDefinitionNames = {ArrayList@6224} size = 181
  ▶ 0 = "org.springframework.context.annotation.internalConfigurationAnnotationProcessor"
  ▶ 1 = "org.springframework.context.annotation.internalAutowiredAnnotationProcessor"
  ▶ 2 = "org.springframework.context.annotation.internalCommonAnnotationProcessor"
  ▶ 3 = "org.springframework.context.event.internalEventListenerProcessor"
  ▶ 4 = "org.springframework.context.event.internalEventListenerFactory"
  ▶ 5 = "accountServiceApplication"
  ▶ 6 = "org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory"
  ▶ 7 = "druidConfig"
  ▶ 8 = "dubboConfig"
  ▶ 9 = "fescarConfig"
  ▶ 10 = "accountController"
  ▶ 11 = "apiServiceApplication"
  ▶ 12 = "druidDataSource"
  ▶ 13 = "fescarDataSource"

```

<https://blog.csdn.net/kangsa998>

- 有循环依赖的原因：

1. 按照 `beanDefinition` 初始化的顺序，第一个初始化和数据源有关的类是 `accountController`
2. 这个类里依赖了 `accountMapper` 类，所以要递归初始化 `accountMapper`
3. 而 `accountMapper` 需要 `SqlSessionFactory`，`SqlSessionFactory` 又在构造器中依赖了 `Primary_dataSource`
4. 而 `Primary_dataSource` 中有依赖了 `druidDataSource`，初始化 `druidDataSource` 完后，就会进入 3.1.4
5. 关键是3.1.5那 `if` 为 `true`（因为是第一次初始化数据源），进入又调用了 `Primary_dataSource`。产生循环依赖！

- [相同配置，调整顺序后的产生没有循环依赖的代码](#)

- 下图是没有循环依赖的所有beanDefinition列表

```

▼ beanNames = {ArrayList@5479} size = 176
  ▶ 0 = "org.springframework.context.annotation.internalConfigurationAnnotationProcessor"
  ▶ 1 = "org.springframework.context.annotation.internalAutowiredAnnotationProcessor"
  ▶ 2 = "org.springframework.context.annotation.internalCommonAnnotationProcessor"
  ▶ 3 = "org.springframework.context.event.internalEventListenerProcessor"
  ▶ 4 = "org.springframework.context.event.internalEventListenerFactory"
  ▶ 5 = "springBootDatasourceApplication"
  ▶ 6 = "org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory"
  ▶ 7 = "datasourceConfig"
  ▶ 8 = "JdbcTemplateConfig"
  ▶ 9 = "personController"
  ▶ 10 = "c3p0Datasource"
  ▶ 11 = "hikarDatasource"
  ▶ 12 = "dbcpDatasource"

```

<https://blog.csdn.net/kangsa998>

- 没有循环依赖的原因：

1. 按照 `beanDefinition` 初始化的顺序，第一个初始化和 `dataSource` 有关的类是 `JdbcTemplateConfig`
2. 因为这个类中依赖了 `c3p0DataSource` 类，会递归初始化它
3. `c3p0DataSource` 初始化完成后，会调用所有的后置处理器，这样就会进入3.1.4，关键是，这时就会初始化 `DataSourceInitializerInvoker` 了
4. 关键：等初始化主数据源(`primaryDataSource`)时，就不会初始化 `DataSourceInitializerInvoker`，进而进入3.1.5了，从而不会产生循环依赖了

思路总结：只要保证首先初始化的数据源不是主数据源即可

4.3 初始化主数据源的时候不要依赖其它数据源

```
@Primary
@Bean("fescarDataSource")
public DataSource dataSource() {
    DataSourceProxy dataSourceProxy = new DataSourceProxy(null);
    return dataSourceProxy;
}
//在初始化完了之后才调用的，这样就不会去递归初始化`druidDataSource`了
@PostConstruct
public void init() {
    DataSource dataSource = dataSource();
    if (dataSource instanceof DataSourceProxy) {
        Class cz = dataSource.getClass().getSuperclass();
        try {
            Field f1 = cz.getDeclaredField("targetDataSource");
            f1.setAccessible(true);
            f1.set(dataSource, druidDataSource());
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

5 总结

- 解决这个问题的关键在于熟悉 `springboot` 初始化类的过程，以下是简化的过程

```
prepareEnvironment();//获取设置配置文件(.yml等)
invokeBeanFactoryPostProcessors();//解析配置文件，生成所有需要的beanDefinition;
//初始化非懒加载beanDefinition
finishBeanFactoryInitialization(){
    Iterator var2 = beanNames.iterator();//遍历所有的beanDefinition
    getBean(var2.next());
}
```

```
//关键！：这里是初始化一个bean的过程
getBean(beanName) {
    doGetBean(); //这里递归处理依赖类
    createBeanInstance(); //反射创建实例
    populateBean(); //填充bean属性，包含依赖类
    invokeAwareMethods(); //如果bean实现了某个Aware接口，就会调用它的方法
    applyBeanPostProcessorsBeforeInitialization(); //递归调用所有的后置处理器的 before 方法，
    其中通过CommonAnnotationPostProcessor调用了@PostConstruct指定的初始化方法
    invokeInitMethods(); //调用实现InitializingBean的afterPropertySet()方法，然后调用
    @Bean(initMethod="...")指定的方法
    applyBeanPostProcessorsAfterInitialization(); //递归调用所有的后置处理器的 after 方法
}
```

- 通过上面的过程，我们可以知道Init方法的调用已经和循环依赖没有关系了
- 从两个 `beanDefinition` 列表，也可以推出 `beanDefinition` 列表中的大致顺序
 1. 一些框架类和 `main` 对应的类在最前面
 2. 然后按包的顺序，排序其中的配置类(`@Configuration`、`@Component`等修饰)
 3. 按顺序配置类中的 `@bean` 修饰的方法返回的类
 4. 第三方包中的类
- 较为springboot启动过程和初始化bean的流程文献
 1. [Springboot 源码分析——总纲](#)
 2. [Springboot 源码分析——bean初始化流程、beanPostProcessor用法、循环依赖](#)

参考

[没有循环依赖的代码](#) [有循环依赖的代码](#) [Springboot 源码分析——总纲](#) [Springboot 源码分析——bean初始化流程、beanPostProcessor用法、循环依赖](#) [Springboot 源码分析——自动运行schema.sql解析](#) [github 源码地址](#)