Comportamientos del IoC en Spring Framework

Daniel Alonso Chavarro Chipatecua[1]
Andres Felipe Murillo Torres[2]
[1] dachavarroc@udistrital.edu.co
[2] afmurillot@udistrital.edu.co

Punto 1 El laboratorio de los Beans

Tema

Creación, selección y posponer Beans.

Descripción del problema

Se necesita crear varios Beans de distintos tipos: @Component y @Bean, los cuales tienen que ser diferenciados según el nombre que se le asigne a cada uno, usando @Qualifier y ademas debe ser posible posponer la creación de alguno de estos Beans con @lazy.

Codigo base

Se mostrara este proceso al crear la clase ExperimentService y la clase Config. La primera sera un @Component, mientras que la segunda contendra 2 @Beans. Para poder diferenciarlos se crea una clase llamada SelectionService en la cual se elegira uno de los Beans dependiendo del nombre que tengan:

```
@Component("ExperimentService")
public class ExperimentService {
    public ExperimentService() {
         System.out.println("Se ha creado ExperimentService!");
    }
}
@Configuration
public class Config {
    @Bean("ConfigurationService1")
    public Service1 service1() {
        return new Service1();
    }
    @Bean("ConfigurationService2")
    public Service2 service2() {
        return new Service2();
    }
}
@Component
public class SelectionService {
```

```
    @Autowired
    SelectionService(@Qualifier("ConfigurationService1") Service1 Service){
        System.out.println("Inyeccion:"+ Service);
    }
}
```

Implementación de @lazy

Se implementa @lazy en ExperimentService:

```
@Component("ExperimentService")
@Lazy
public class ExperimentService {
    public ExperimentService() {
        System.out.println("Running ExperimentService...");
    }
}
```

Se implementa @lazy en los Beans de Config:

```
@Configuration
public class Config {
    @Bean("ConfigurationService1")
    @Lazy
    public Service1 service1() {
        return new Service1();
    }
    @Bean("ConfigurationService2")
    @Lazy
    public Service2 service2() {
        return new Service2();
    }
}
```

Resultado

El resultado que se obtiene cuando ExperimentService tiene @Lazy es:

```
Se ha creado Service 1!
Inyeccion:taller1.Service1@6a84bc2a
Se ha creado Service 2!
```

Se obtiene la creacion de los beans para service 1 y 2. Pero como ExperimentService tiene @Lazy y este no es solicitado, no se crea el bean correspondiente.

El resultado que se obtiene cuando los @Bean tienen @Lazy es:

```
Se ha creado Service 1!
Inyeccion:taller1.Service1@1c12f3ee
```

Esto es debido a que en SelectionService se esta usando el Bean de service 1, mostrando que @Lazy esta evitando que se generen los beans que no fueron solicitados.

Conclusiones

Se observa que para la creación de beans usando @Bean es necesario el uso de una clase para la configuración de este, como tambien el tener un metodo que permita generar

el bean. Mientras que en @Component solo es necesario añadir la etiqueta para una clase. En cuanto a la ejecución se observa que es igual en ambos casos, incluso con la implementación de @Lazy.

Punto 3 — La Conspiración de los Qualifiers

Tema

Resolución de dependencias y ambigüedades en el contenedor de Spring.

Descripción del problema

En este ejercicio se analizó el comportamiento del contenedor de Spring al inyectar dependencias cuando existen múltiples implementaciones de una misma interfaz. Se definió la interfaz `DiscountService` y dos clases que la implementan: `BasicDiscountService` y `PremiumDiscountService`, ambas anotadas con `@Service`.

Posteriormente, en la clase `OrderService` se intentó inyectar un objeto de tipo `DiscountService` sin especificar cuál implementación debía usarse. El contexto de Spring lanzó una excepción durante la inicialización.

Código base

```java
public interface DiscountService {
    double applyDiscount(double amount);
}

@Service
public class BasicDiscountService implements DiscountService {
    public double applyDiscount(double amount) {
        return amount * 95 / 100;
    }
}

@Service
public class PremiumDiscountService implements DiscountService {
    public double applyDiscount(double amount) {
        return amount * 90 / 100;
    }
}

@Service
public class OrderService {
    private final DiscountService discountService;

    @Autowired
    public OrderService(DiscountService discountService) {
        this.discountService = discountService;
    }

    public void processOrder(double amount) {
        double finalPrice = discountService.applyDiscount(amount);
        System.out.println("Final amount after discount: " + finalPrice);
    }
```

```
}
```

Error obtenido

Al ejecutar la clase `Runner` que inicializa el contexto con `AnnotationConfigApplicationContext`, se produjo el siguiente error:

```
org.springframework.beans.factory.UnsatisfiedDependencyException:
No qualifying bean of type 'DiscountService' available:
expected single matching bean but found 2:
basicDiscountService, premiumDiscountService
```

Este error ocurre porque Spring encuentra dos beans del mismo tipo (`DiscountService`) y no puede determinar automáticamente cuál debe inyectar.

Solución implementada

Existen dos formas comunes de resolver esta ambigüedad:

1. Usar la anotación `@Primary` en una de las implementaciones:

```
@Primary
@Service
public class BasicDiscountService implements DiscountService { ... }
```

De esta forma, Spring utilizará siempre la implementación marcada como primaria cuando no se especifique un `@Qualifier`.

Resultado en consola:

```
Enter the order amount:
1000000
Final amount after discount: 950000.0
```

2. Especificar explícitamente el bean con `@Qualifier`:

```
@Service
public class OrderService {
    private final DiscountService discountService;

    @Autowired
    public OrderService(@Qualifier("premiumDiscountService")
        DiscountService ds) {
        this.discountService = ds;
    }
}
```

El nombre del bean (`"premiumDiscountService"`) coincide con el nombre de la clase con la primera letra en minúscula.

Resultado en consola:

```
Enter the order amount:
1000000
Final amount after discount: 900000.0
```

Ahora, y si cambiamos los nombres de los beans y dejar un nombre incorrecto en el qualifier o el mismo nombre para ambos?

- **Caso Nombres Incorrectos**: Cambiamos el bean a la clase `PremiumDiscountService` de la siguiente manera:

```
@Service("premiumDiscountApplyService")
class PremiumDiscountService implements DiscountService {
```

Se mantiene la misma estructura del `OrderService`, el resultado en consola es:

```
Caused by: org.springframework.beans.factory.
    NoSuchBeanDefinitionException:
No qualifying bean of type 'org.qualifiers.service.discount.
    DiscountService' available:
expected at least 1 bean which qualifies as autowire candidate.
Dependency annotations: {@org.springframework.beans.factory.
    annotation.Qualifier("premiumDiscountService")}
```

Lo cual claramente referencia a la inexistencia del bean marcado en `OrderService`.

- **Caso Nombres Iguales**: Se cambia ambos beans de los descuentos a ”discountService”:

```
@Service("discountService")
class BasicDiscountService implements DiscountService { ... }

@Service("discountService")
class PremiumDiscountService implements DiscountService { ... }

@Autowired
public OrderService(@Qualifier("discountService") DiscountService
    discountService) {
    this.discountService = discountService;
}
```

Sucede algo similar como en el caso de una inyección de dependencia sin especificar la clase o el bean, y no importa si tiene la etiqueta @Primary, dando como resultado:

```
Annotation-specified bean name 'discountService' for bean class
[org.qualifiers.service.discount.PremiumDiscountService]
    conflicts with existing,
non-compatible bean definition of same name and class
[org.qualifiers.service.discount.BasicDiscountService]
```

Uso de @Autowired(required = false)

De forma predeterminada, `@Autowired` requiere que el bean exista en el contexto; de lo contrario, el contenedor lanza una excepción `NoSuchBeanDefinitionException`. Spring permite modificar este comportamiento marcando la dependencia como opcional mediante el atributo `required = false`:

```java
@Service
public class OrderService {
    private final DiscountService discountService;

    @Autowired(required = false)
    public OrderService(@Qualifier("experimentalDiscountService")
        DiscountService discountService) {
        this.discountService = discountService;
    }

    public OrderService() {
        this.discountService = null;
    }

    public void placeOrder(double amount) {
        if (discountService != null) {
            double discountedAmount = discountService.applyDiscount(amount)
                ;
            System.out.println("Order placed. Original amount: " + amount +
                               ", Discounted amount: " + discountedAmount);
        } else {
            System.out.println("Order placed. Amount: " + amount +
                               " (No discount applied)");
        }
    }
}
```

En este caso, si no existe un bean de tipo `experimentalDiscountService` llamado `experimentalDiscount`, el contenedor no genera error y la variable se mantiene en `null`. Esto permite manejar dependencias condicionales o opcionalmente disponibles.

Resultado en consola:

```
Enter the order amount:
1000000
Order placed. Amount: 1000000.0 (No discount applied)
```

Cabe aclarar que cuando se utiliza un `@Autowired(required = false)` sobre un constructor, en el caso de que el bean no exista, se requerirá un constructor alternativo que no utilice esa dependencia. De igual manera para inyección por setters.

Conclusión

El uso de `@Primary` simplifica la configuración cuando se desea una implementación por defecto, mientras que `@Qualifier` brinda control preciso cuando se necesita seleccionar entre varias implementaciones. Entender esta diferencia es fundamental para evitar errores de inyección en aplicaciones con múltiples estrategias o servicios similares.

Punto 4 — El Bucle Infinito

Tema

Dependencias circulares, orden de inicialización y contexto de carga en Spring.

Descripción del problema

En este ejercicio se explora el comportamiento del contenedor de Spring cuando dos beans se necesitan mutuamente durante la inicialización. El caso presentado ocurre entre las clases `InventoryService` y `OrderService`, las cuales se inyectan entre sí mediante sus constructores. Al iniciar la aplicación, el contenedor intenta crear ambos beans simultáneamente, provocando una dependencia circular imposible de resolver.

Código base

El siguiente código corresponde a los archivos originales utilizados para el ejercicio:

```java
package org.loop.withLoops.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class InventoryService {

    private final OrderService orderService;

    @Autowired
    public InventoryService(OrderService orderService) {
        System.out.println("Starting Inventory Service");
        this.orderService = orderService;
    }

    public boolean checkInventory(String product) {
        System.out.println("Checking if Inventory exists for product " +
            product);
        return true;
    }

    public void verifyIfOrderExits(String product) {
        if (orderService.verifyOrder(product)) {
            System.out.println("Order has been executed for product " +
                product);
        } else  {
            System.out.println("Order has been cancelled for product " +
                product);
        }
    }
}
```

```java
package org.loop.withLoops.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class OrderService {

    private final InventoryService inventoryService;

    @Autowired
    public OrderService(InventoryService inventoryService) {
```

```
        System.out.println("Starting Order Service");
        this.inventoryService = inventoryService;
    }

    public void createOrder(String product) {
        if (inventoryService.checkInventory(product)) {
            System.out.println("Order has been created");
        } else  {
            System.out.println("Order has not been created, out of stock");
        }
    }

    public boolean verifyOrder(String product) {
        System.out.println("Checking if order exists for product " +
            product);
        return true;
    }
}
```

```
package org.loop.withLoops;

import org.loop.withLoops.service.InventoryService;
import org.loop.withLoops.service.OrderService;
import org.springframework.context.annotation.
    AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import java.util.Scanner;

@Configuration
@ComponentScan
public class Runner {

    static void main() {
        AnnotationConfigApplicationContext context = new
            AnnotationConfigApplicationContext(Runner.class);

        Scanner scanner = context.getBean(Scanner.class);
        OrderService orderService = context.getBean(OrderService.class);
        InventoryService inventoryService = context.getBean(
            InventoryService.class);

        System.out.println("Enter product name:");
        String productName = scanner.next();

        orderService.createOrder(productName);
        inventoryService.verifyIfOrderExits(productName);
    }
}
```

Error obtenido

Durante la ejecución del programa, Spring no logra crear el contexto debido a la referencia circular entre los servicios. El error mostrado en consola es el siguiente:

```
Error creating bean with name 'orderService': Unsatisfied
```

```
dependency expressed through constructor parameter 0: Error
creating bean with name 'inventoryService': Requested bean
is currently in creation: Is there an unresolvable circular
reference or an asynchronous initialization dependency?
```

Este error indica que el contenedor intenta crear el bean `OrderService`, pero para hacerlo necesita un `InventoryService` ya inicializado, el cual a su vez depende nuevamente de `OrderService`, generando así un bucle infinito de inicialización.

Análisis

La excepción `BeanCurrentlyInCreationException` aparece cuando dos o más beans dependen entre sí por constructor. En este escenario, el contenedor no puede completar la creación del primero porque necesita el segundo, que también está en proceso de creación.

El flujo interno del error puede describirse así:

1. Spring detecta `OrderService` y crea una instancia, pero antes de completarla intenta inyectar `InventoryService`.
2. El contenedor procede a crear `InventoryService`, pero detecta que este requiere un `OrderService`.
3. Como `OrderService` aún no está completamente inicializado, el ciclo no puede romperse y Spring lanza la excepción.

Solución aplicada

Para resolver este comportamiento se utilizó una de las estrategias recomendadas en Spring: romper la dependencia directa. En el ejercicio, la corrección se realizó modificando una de las inyecciones (ya sea aplicando `@Lazy` o cambiando a inyección por setter), lo que permite que el contenedor inicialice primero uno de los beans y posponga la creación del otro hasta que realmente se necesite.

```
1     @Lazy
2     @Autowired
3     public InventoryService(OrderService orderService) {...}
```

Evidencia de ejecución

Una vez corregido el problema, la aplicación inicia sin excepción y produce la siguiente salida en consola:

```
Starting Inventory Service
Starting Order Service
Enter product name:
PC
Checking if Inventory exists for product PC
Order has been created
Checking if order exists for product PC
Order has been executed for product PC
```

Se logra ver que se inicializa el servicio con la etiqueta Lazy primero antes que el Order, se supone que al lazy usa un proxy temporal con el fin de no inicializar directamente la otra dependencia, se inicializa el `InventoryService` y con eso se inyecta para instanciar el `OrderService`.

Solución de diseño

Normalmente cuando se encuentra dependencias circulares en una aplicación significa un mal diseño de la aplicación, la recomendación principal es analizar nuevamente el diseño y modificar toda la lógica de negocio de ser necesario. Sin embargp si ese no es el caso y esas 2 clases si estan fuertemente acopladas, se puede utilizar el patrón de diseño de comportamiento **Mediator**, donde las clases se desacopla su lógica fuertemente acoplada haciendo que no sepan la implementación de la otra clase, y el mediador (en este caso será `SalesManagerService`) cordina ambos servicios.

Codigo modificado:

```
    package org.loop.solution.service;

import org.springframework.stereotype.Service;

@Service
public class InventoryService {

    public boolean checkInventory(String product) {
        System.out.println("Checking if Inventory exists for product " +
            product);
        return true;
    }

    public void verifyIfOrderExits(String product) {
        System.out.println("Order has been executed for product " + product
            );
    }
}
```

```
package org.loop.solution.service;

import org.springframework.stereotype.Service;

@Service
public class OrderService {

    public void createOrder(String product) {
        System.out.println("Order has been created");
    }

    public boolean verifyOrder(String product) {
        System.out.println("Checking if order exists for product " +
            product);
        return true;
    }
}
```

```
package org.loop.solution.service;

```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class SalesManagerService {
    private final InventoryService inventoryService;
    private final OrderService orderService;

    @Autowired
    public SalesManagerService(InventoryService inventoryService,
        OrderService orderService) {
        this.inventoryService = inventoryService;
        this.orderService = orderService;
    }

    public void createOrder(String product) {
        if (inventoryService.checkInventory(product)) {
            orderService.createOrder(product);
        } else {
            System.out.println("Order has not been created, out of stock");
        }
    }

    public void verifyIfOrderExits(String product) {
        if (orderService.verifyOrder(product)) {
            inventoryService.verifyIfOrderExits(product);
        } else {
            System.out.println("Failed to verify if order exits");
        }
    }
}
```

```java
package org.loop.solution;

import org.loop.solution.service.InventoryService;
import org.loop.solution.service.OrderService;
import org.loop.solution.service.SalesManagerService;
import org.springframework.context.annotation.
    AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import java.util.Scanner;

@Configuration
@ComponentScan
public class Runner {

    static void main() {
        AnnotationConfigApplicationContext context = new
            AnnotationConfigApplicationContext(Runner.class);

        Scanner scanner = new Scanner(System.in);
        SalesManagerService salesManagerService = context.getBean(
            SalesManagerService.class);

        System.out.println("Enter product name:");
        String productName = scanner.next();
```

```
24
25          salesManagerService.createOrder(productName);
26          salesManagerService.verifyIfOrderExits(productName);
27      }
28  }
```

Resultado de la consola:

```
Enter product name:
laptop
Checking if Inventory exists for product laptop
Order has been created
Checking if order exists for product laptop
Order has been executed for product laptop
```

Es una idea vaga de la solución, pero es mas efectivo que usar un @Lazy, con el único riesgo de romper el principio de responsabilidad única

Conclusión

El error `BeanCurrentlyInCreationException` es uno de los más comunes al trabajar con servicios interdependientes. Spring no permite inyecciones cíclicas por constructor porque ninguno de los beans puede existir antes del otro. Para evitarlo, se recomienda:

- Analizar las relaciones de dependencia antes del diseño final.
- Sustituir la inyección por constructor por setter en uno de los beans.
- Usar `@Lazy` para retrasar la carga del bean dependiente.
- Considerar refactorizar la lógica dividiendo responsabilidades entre servicios o usando mediadores entre las clases.

Con esta modificación, el contexto de Spring se inicializa correctamente y se evita el bucle de creación infinita.