Cykl życia

Cykle życia Bean w Spring framework

Każdy obiekt ma swój cykl życia w wirtualnej maszynie Java.
Gdy obiekt jest tworzony, przechodzi on szereg etapów inicjalizacji na poziomie klasy, następnie na poziomie instancji, po czym wywoływany jest konstruktor klasy.

Kiedy (i jeśli) obiekt staje się nieosiągalny dla kodu programu, obiekt może zostać oczyszczony przez garbage collector, który sam może wywoływać specjalnie nazwane metody, aby obiekt mógł wyczyścić wszelkie przydzielone zasoby, co jest procesem znanym jako finalizacja.

Scope

Spring zapewnia dwa podstawowe scope dla obiektów w standardowym ApplicationContext – singleton i prototype.

singleton - domyślny – jest tworzony i niszczony raz w danym kontekście aplikacji. Jeśli pobierasz obiekt wiele razy, za każdym razem otrzymasz to samo odwołanie do obiektu.

prototype - konstruowany podczas pobierania, w każdym kolejnym miejscu wstrzyknięcia będzie tworzona nowa instancja obiektu. Takie rozwiązanie staje się bardzo przydatne w przypadku klas przechowujących stan, gdy wymagane jest kolekcjonowanie danych niezależnie, w kolejnych miejscach wstrzyknięcia.

W module webowym Springa dostępne są jeszcze inne zakresy, omówione są w rozdziale 6 książki “Beginning Spring 5”

Tworzenie przykładowego projektu

```
mkdir -p chapter4/src/main/java/com/bsg5/chapter4
mkdir -p chapter4/src/test/java/com/bsg5/chapter4
mkdir -p chapter4/src/test/resources
```

Stworzenie struktury katalogów, zaczynając od ogólnego katalogu projektu

```
dependencies {
    compile "org.springframework:spring-core:$springFrameworkVersion"
    compile "org.springframework:spring-context:$springFrameworkVersion"
    compile "org.springframework:spring-test:$springFrameworkVersion"
    compile "javax.annotation:javax.annotation-api:1.3.2"
}
```

Utworzenie build.gradle i settings.gradle

```
package com.bsg5.chapter4;

import java.util.Objects;

abstract class  HasData {
    String datum = "default";

  public String getDatum() {
      return datum;
  }

  public void setDatum(String datum) {
      this.datum = datum;
  }

  @Override
  public boolean equals(Object o) {
      if (this == o) return true;
      if (!(o instanceof HasData)) return false;
      HasData hasData = (HasData) o;
      return Objects.equals(getDatum(), hasData.getDatum());
  }

  @Override
  public int hashCode() {
      return Objects.hash(getDatum());
  }
}
```

W celu zademonstrowania cyklu życia, tworzymy klasę abstrakcyjną do przechowywania pojedynczego fragmentu danych
— zwaną HasData — i rozszerzamy ją o wiele klas osadzonych w naszym kodzie testowym.

Każda metoda została wygenerowana automatycznie przez IDEA.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<!-- chapter4/src/test/resources/config-01.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!--
    note that "singleton" scope is the default, so this declaration is
    unnecessary.
    -->
    <bean name="foo"
          class="com.bsg5.chapter4.FirstObject"
          scope="singleton"/>
    <bean name="bar"
          class="com.bsg5.chapter4.FirstObject"
          scope="prototype"/>
</beans>
```

Aby ustawić obiekt jako prototype, po prostu dodajemy atrybut scope do konfiguracji. Singleton jest wartością domyślną, dodawaną automatycznie.

Definiujemy dwa komponenty bean – foo i bar – oba tego samego typu, ponieważ możemy pobierać obiekty według nazwy, z różnymi zakresami.

```
class FirstObject extends HasData {
}

@ContextConfiguration(locations = "/config-01.xml")
public class TestLifecycle01 extends AbstractTestNGSpringContextTests {
    @Autowired
    ApplicationContext context;

    @DataProvider

   Object[][] getReferences() {
       return new Object[][]{
               {"foo", true},
               {"bar", false}
       };
   }

   @Test(dataProvider = "getReferences")
   public void testReferenceTypes(String name, boolean singleton) {
       HasData o1 = context.getBean(name, HasData.class);

       String defaultValue = o1.getDatum();
       o1.setDatum(UUID.randomUUID().toString());

       HasData o2 = context.getBean(name, HasData.class);
       if (singleton) {
           assertSame(o1, o2);
           assertEquals(o1, o2);
           assertNotEquals(defaultValue, o2.getDatum());
       } else {
           assertNotSame(o1, o2);
           assertNotEquals(o1, o2);
           assertEquals(defaultValue, o2.getDatum());
       }
   }
}
```

Jeśli typ jest określony jako singleton, obiekty powinny być tą samą instancją (tj. o1 == o2), a ustawienie danych w jednym powinno być również odzwierciedlone w drugim (innymi słowy, istnieje zarówno równość instancji, jak i równość danych).

Jeśli zamiast tego typ jest określony jako prototyp, wówczas obiekty nie powinny być tą samą instancją, a ustawienie danych w jednym powinno pozostawić drugi w stanie domyślnym.

Konstruktor

Możemy zdefiniować konstruktor dla Spring beans poprzez nazwanie argumentu bądź użycie indeksu.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- chapter4/src/test/resources/config-02.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="foo" class="com.bsg5.chapter4.SecondObject">
        <constructor-arg name="initialValue"
                         value="Initial Value"/>
    </bean>
</beans>
```

Wywoływanie metod

Istnieje możliwość wywołania metody po konstruktorze ale zanim kontekst opuści wszystkie referencje, którymi zarządza.

W tym celu używamy init-method i destroy-method.

Metody te nie mogą zwracać żadnych wartości oraz nie mogą dziedziczyć po innych metodach.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- chapter4/src/test/resources/config-03.xml -->

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean name="foo"
          class="com.bsg5.chapter4.ThirdObject"
          init-method="init"
          destroy-method="dispose"
      />
</beans>
```

```
ThirdObject foo=new ThirdObject();
foo.init();
```

Wywoływanie destroy-method

Destroy-method to nie to samo co Java finalizer. Kolejność wywoływania poszczególnych metod będzie różna w ich przypadku.

Metoda destroy może zostać zastosowana tylko jeśli dany bean jest singletonem. Dlatego, że spring konstruuje prototypy, przekazuje do kodu i potem już nimi nie zarządza. Po utworzeniu stają się one niezarządzalne.

Zwróć uwagę!

Semafory

Test pobiera bean z aplikacji, a następnie weryfikuje:

- czy instancja komponentu wygląda poprawnie
- czy semafor jest wypełniony

Następnie zamykamy kontekst, co powoduje ustawienie semafora na wartość null.

```
@ContextConfiguration(locations = "/config-03.xml")
public class TestLifecycle03 extends AbstractTestNGSpringContextTests {
    @Autowired
    ConfigurableApplicationContext context;

    @Test
    public void testInitDestroyMethods() {
        ThirdObject o1 = context.getBean(ThirdObject.class);
        assertNotNull(ThirdObject.semaphore);
        assertEquals(o1.getDatum(), "default");
        context.close();
        assertNull(ThirdObject.semaphore);
    }
}
```

Semafory

Nasz test zdefiniuje ThirdObject i utworzy odwołanie, które potocznie możemy nazwać semaforem. Ta referencja jest zawarta w metodzie init.

Definiujemy też metodę destroy, która ustawi semafor na wartość null.

```
package com.bsg5.chapter4;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.
AbstractTestNGSpringContextTests;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

class ThirdObject extends HasData {
    static Object semaphore = null;

    public void init() {
        semaphore = new Object();
    }
    public void dispose() {
        semaphore = null;
    }
}
```

Cykle życia przy użyciu adnotacji JSR-250

Adnotacje wykorzystywane do definicji cyklów życia:

- umożliwiają konfigurację większości funkcji w porównaniu z konfiguracją XML
- większość funkcji cyklu życia jest identyczna (z interfejsami **InitializingBean** i **DisposableBean**), albo ma swoje bezpośrednie odpowiedniki (np. z adnotacjami zawierającymi nazwy metod dla init-method).

Plik konfiguracyjny do testów

Dla następnych testów cykla życia z adnotacjami, został wykorzystany następujący plik konfiguracyjny:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans

        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.bsg5.chapter4"/>
</beans>
```

Adnotacje w konfiguracji zakresów

Zdefiniowanie zakresu dla danych komponentów Bean jest możliwe poprzez dodanie adnotacji **@Scope** z odpowiednią nazwą zakresu, jak np. ConfigurableBeanFactory.SCOPE_SINGLETON, czy ConfigurableBeanFactory.SCOPE_PROTOTYPE.

Przykład:

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
class FifthObject extends HasData {
}

@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
class SixthObject extends HasData {
}
```

Konstruktor z adnotacją @Autowired

```
abstract class Foo {
    protected String initialValue;

    void printValue() {
        this.initialValue = "Initial Value";
    }
}
```

Note

As of Spring Framework 4.3, an `@Autowired` annotation
constructor to begin with. However, if several constructor
constructors must be annotated with `@Autowired` in orc
resolution for details.

```
@SpringBootApplication
public class AutowireConstructorExample implements CommandLineRunner {
    public static void main(String[] args) {
        SpringApplication.run(AutowireConstructorExample.class, args);
    }

    private final Foo fooBar;

    @Autowired
    public AutowireConstructorExample(Foo fooBar) {
        this.fooBar = fooBar;
    }

    @Override
    public void run(String... args) {
        fooBar.printValue();
    }
}
```

Adnotacje @PostConstruct i @PreDestroy

```
@Component
class SeventhObject extends HasData {
    static Object semaphore = null;

    @PostConstruct
    public void initialize() throws Exception {
        semaphore = new Object();
    }

    @PreDestroy
    public void dispose() throws Exception {
        semaphore = null;
    }
}
```

```
@ContextConfiguration("/annotated-06.xml")
public class TestLifecycle06 extends AbstractTestNGSpringContextTests {
    @Autowired
    ConfigurableApplicationContext context;

    @Test
    public void testInitDestroyMethods() {
        EighthObject o1 = context.getBean(EighthObject.class);
        assertNotNull(EighthObject.semaphore);
        assertEquals(o1.getDatum(), "default");
        context.close();
        assertNull(EighthObject.semaphore);
    }
}
```

```java
package com.bsg5.chapter4;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.
AbstractTestNGSpringContextTests;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

class FourthObject extends HasData
        implements InitializingBean, DisposableBean {
    static Object semaphore = null;

    @Override
    public void afterPropertiesSet() throws Exception {
        semaphore = new Object();
    }
    @Override
    public void destroy() throws Exception {
        semaphore = null;
    }
}

@ContextConfiguration(locations = "/config-04.xml")
public class TestLifecycle04 extends AbstractTestNGSpringContextTests {
    @Autowired
    ConfigurableApplicationContext context;

    @Test
    public void testLifecycleMethods() {
        FourthObject o1 = context.getBean(FourthObject.class);
        assertNotNull(FourthObject.semaphore);
        assertEquals(o1.getDatum(), "default");
        context.close();
        assertNull(FourthObject.semaphore);
    }
}
```

Listenery
(nasłuchiwanie zdarzeń) cyklu życia

Zaimplementowane metody
afterPropertiesSet() oraz destroy()
zastępują metody init i dispose z konfiguracji XML.

Cykl życia z konfiguracją Java

- Wszystkie komponenty bean z adnotacjami są dobre, natomiast adnotacje powodują, że ich konfiguracja jest globalna. Ustawienie nazwy komponentu bean z adnotacjami spowoduje ustawienie tej samej nazwy dla każdej instancji tej samej klasy w bieżącej ścieżce klasy.

- Komponenty bean można ustawić na różne zakresy za pomocą XML i adnotacji. Istnieje również możliwość użycia klasy konfiguracji.

- Zaletą klasy konfiguracyjnej jest to, że w ścieżce klasy może być obecnych wiele klas konfiguracyjnych w dowolnym momencie, co zapewnia moc i przejrzystość podejścia opartego na adnotacjach, z elastycznością podejścia XML.

- Test odpowiada za tworzenie obiektu rozszerzającego HasData o nazwie NinthObject. Konfiguracja przechowywana jest w lokalnej klasie o nazwie Config08.

- Zadeklarowane zostały dwie metody foo() i bar(). Obydwie metody oznaczone są @Bean.

- Zamieniono również @ContextConfiguration dla klasy testowej, aby odnosiła się do Config08.class zamiast nazwanego pliku konfiguracyjnego.

```
@Component
class EighthObject extends HasData
        implements InitializingBean, DisposableBean {
    static Object semaphore = null;

    @Override
    public void afterPropertiesSet() throws Exception {
        semaphore = new Object();
    }

    @Override
    public void destroy() throws Exception {
        semaphore = null;
    }
}

@ContextConfiguration(locations = "/annotated-07.xml")
public class TestLifecycle07 extends AbstractTestNGSpringContextTests {
    @Autowired
    ConfigurableApplicationContext context;

    @Test
    public void testLifecycleMethods() {
        EighthObject o1 = context.getBean(EighthObject.class);
        assertNotNull(EighthObject.semaphore);
        assertEquals(o1.getDatum(), "default");
        context.close();
        assertNull(EighthObject.semaphore);
    }
}
```

Dodatkowe zakresy

Spring w naturalny sposób dodaje więcej do zakresów prototypowych i pojedynczych, niż zostało to omówione w powyższym rozdziale.

Spowodowane jest to głównie tym, że dodatkowe zakresy istnieją w kontekście Spring Web. Zasady określania zakresu nie zmieniają się, ale nazwy i osiągalność tak; komponent może być oznaczony w taki sposób, że istnieje na przykład jako singleton w kontekście pojedynczego żądania HTTP lub w kontekście sesji użytkownika.

Dziękujemy za uwagę

Autorzy:

Anna Pietruszka
Edyta Mróz
Karina Wcisło
Mateusz Jamrozik
Tomasz Malinowski
Małgorzata Zieleń