(http://baeldung.com)

I s frijoles p n una instancia Singleton en primavera

Última modificación: 11 de marzo de 2018

> por baeldung (http://www.baeldung.com/author/baeldung/) (http://www.baeldung.com/author/baeldung/)

Primavera (http://www.baeldung.com/category/spring/) +

Acabo de anunciar los nuevos módulos de *Spring 5* en REST With Spring:

>> COMPRUEBA EL CURSO (/rest-with-spring-course#new-modules)

1. Información general

En este artículo rápido, vamos a mostrar diferentes enfoques para **inyectar prototipos de beans en una instancia singleton** . Discutiremos los casos de uso y las ventajas / desventajas de cada escenario.

Por defecto, Spring beans son singletons. El problema surge cuando tratamos de conectar frijoles de diferentes ámbitos. Por ejemplo, un prototipo de frijol en un singleton. **Esto se conoce como el problema de inyección de frijol con alcance** .

Para obtener más información sobre los alcances de frijol, este artículo es un buen lugar para comenzar (http://www.baeldung.com/spring-bean-scopes) .

2 ma de inyección de frijoles

Pa nos los siguientes beans:

```
ory.SCOPE_PROTOTYPE)
    public PrototypeBean prototypeBean() {
        return new PrototypeBean();
    }

    @Bean
    public SingletonBean singletonBean() {
        return new SingletonBean();
    }
}
```

Tenga en cuenta que el primer bean tiene un ámbito de prototipo, el otro es un singleton.

Ahora, inyectemos el bean con ámbito de prototipo en el singleton, y luego *expongamos* si a través del método *getPrototypeBean ()* :

```
public class SingletonBean {

    // ..

    @Autowired
    private PrototypeBean prototypeBean;

    public SingletonBean() {
        logger.info("Singleton instance created");
    }

    public PrototypeBean getPrototypeBean() {
        logger.info(String.valueOf(LocalTime.now()));
        return prototypeBean;
    }
}
```

Luego, carguemos el *ApplicationContext* y obtengamos el bean singleton dos veces:

```
public static void main(String[] args) throws InterruptedException {
    AnnotationConfigApplicationContext context
      = new AnnotationConfigApplicationContext(AppConfig.class);

                                    n = context.getBean(SingletonBean.class);
                                    e = firstSingleton.getPrototypeBean();

                                    nce one more time
                                    on = context.getBean(SingletonBean.class);
                                    pe = secondSingleton.getPrototypeBean();

                                    s(secondPrototype), "The same instance should be r
```


Aquí está el resultado de la consola:

```
Singleton Bean created
Prototype Bean created
11:06:57.894
// should create another prototype bean instance here
11:06:58.895
```

Ambos beans se inicializaron solo una vez, al inicio del contexto de la aplicación.

3. Inyectar *ApplicationContext*

También podemos inyectar el *ApplicationContext* directamente en un bean.

Para lograr esto, utilice la anotación *@Autowire* o implemente la interfaz *ApplicationContextAware*:

```
public class SingletonAppContextBean implements ApplicationContextAware {

    private ApplicationContext applicationContext;

    public PrototypeBean getPrototypeBean() {
        return applicationContext.getBean(PrototypeBean.class);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
      throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

Cada vez que se llama al método *getPrototypeBean ()*, se *retorna* una nueva instancia de *PrototypeBean* desde *ApplicationContext*.

Sin embargo, este enfoque tiene serias desventajas. Contradice el principio de inversión de control, ya que solicitamos directamente las dependencias del contenedor.

A[illegible]an del *applicationContext* dentro de la clase *S[illegible]***fica acoplar el código al Spring Framework** .

4[illegible]n

O[illegible] inyectar el método con la **anotación *@Lookup*** :

```
[illegible] {
3
4       @Lookup
5       public PrototypeBean getPrototypeBean() {
6           return null;
7       }
8   }
```

Spring anulará el método *getPrototypeBean ()* anotado con *@Lookup.* Luego registra el bean en el contexto de la aplicación. Siempre que solicitemos el método *getPrototypeBean ()* , devuelve una nueva instancia de *PrototypeBean* .

Utilizará CGLIB para generar el bytecode responsable de obtener el *PrototypeBean* del contexto de la aplicación.

5. API *javax.inject*

La configuración junto con las dependencias requeridas se describen en este artículo de cableado de Spring (http://www.baeldung.com/spring-annotations-resource-inject-autowire) .

Aquí está el frijol singleton:

```
1   public class SingletonProviderBean {
2
3       @Autowired
4       private Provider<PrototypeBean> myPrototypeBeanProvider;
5
6       public PrototypeBean getPrototypeInstance() {
7           return myPrototypeBeanProvider.get();
8       }
9   }
```

Usamos la *interfaz del proveedor* para inyectar el prototipo de frijol. Para cada llamada al método *getPrototypeInstance ()* , *myPrototypeBeanProvider.* El método g *et ()* devuelve una nueva instancia de *PrototypeBean* .

6. Proxy con alcance

P encia al objeto real para realizar la inyección. **Aquí, cr tar el objeto real con el dependiente.**

C l objeto proxy, el proxy decide si crear una nueva in existente.

P lase *Appconfig* para agregar una nueva anotación @


```
y.SCOPE_PROTOTYPE,
RGET_CLASS)
```

De forma predeterminada, Spring utiliza la biblioteca CGLIB para subclase de objetos directamente. Para evitar el uso de CGLIB, podemos configurar el modo proxy con *ScopedProxyMode.* INTERFACES, para usar el proxy dinámico JDK en su lugar.

7. Interfaz de *ObjectFactory*

Spring proporciona la interfaz ObjectFactory <T> (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/ObjectFactoryCreatingFactoryBean.html) para producir objetos bajo demanda del tipo especificado (https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/config/ObjectFactoryCreatingFactoryBean.html) :

```
public class SingletonObjectFactoryBean {

    @Autowired
    private ObjectFactory<PrototypeBean> prototypeBeanObjectFactory;

    public PrototypeBean getPrototypeInstance() {
        return prototypeBeanObjectFactory.getObject();
    }
}
```

Echemos un vistazo al método *getPrototypeInstance ()* ; *getObject ()* devuelve una nueva instancia de *PrototypeBean* para cada solicitud. Aquí, tenemos más control sobre la inicialización del prototipo.

Además, *ObjectFactory* es una parte del marco; esto significa evitar una configuración adicional para usar esta opción.

8. Prueba

Va nple JUnit para ejercitar el caso con la interfaz
O

```
tion_WhenObjectFactory_ThenNewInstanceReturn() {

context
icationContext(AppConfig.class);

firstContext
nObjectFactoryBean.class);
secondContext
      = context.getBean(SingletonObjectFactoryBean.class);

    PrototypeBean firstInstance = firstContext.getPrototypeInstance();
    PrototypeBean secondInstance = secondContext.getPrototypeInstance();

    assertTrue("New instance expected", firstInstance != secondInstance);
}
```

Después de iniciar con éxito la prueba, podemos ver que cada vez que se *llama al* método *getPrototypeInstance ()* , se crea una nueva instancia de bean prototipo.

9. Conclusión

En este breve tutorial, aprendimos varias maneras de inyectar el prototipo de frijol en la instancia de singleton.

Como siempre, el código completo para este tutorial se puede encontrar en el proyecto GitHub (https://github.com/eugenp/tutorials/tree/master/spring-core) .

Deja una respuesta

1 comentario sobre "Inyección de frijoles prototipo en una instancia Singleton en primavera"

Join the discussion

▲ el más nuevo ▲ más antiguo ▲ el más votado

guración por mi parte - problema de generación de frijol malo.
como se esperaba!

hace 5 días

CATEGORÍAS

PRIMAVERA (HTTP://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
DESCANSO (HTTP://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTP://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SEGURIDAD (HTTP://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCIA (HTTP://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTP://WWW.BAELDUNG.COM/CATEGORY/JACKSON/)
HTTPCLIENT (HTTP://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

KOTLIN (HTTP://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIE

TUTORIAL "VOLVER AL BÁSICO" DE JAVA (HTTP://WWW.BAELDUNG.COM/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (HTTP://WWW.BAELDUNG.COM/JACKSON)
TUTORIAL DE HTTPCLIENT 4 (HTTP://WWW.BAELDUNG.COM/HTTPCLIENT-GUIDE)
REST CON SPRING TUTORIAL (HTTP://WWW.BAELDUNG.COM/REST-WITH-SPRING-SERIES/)
TUTORIAL DE SPRING PERSISTENCE (HTTP://WWW.BAELDUNG.COM/PERSISTENCE-WITH-SPRING-SERIES/)
SEGURIDAD CON SPRING (HTTP://WWW.BAELDUNG.COM/SECURITY-SPRING)

ACERCA D

ACERCA DE BAELDUNG (HTTP://WWW.BAELDUNG.COM/ABOUT/)
LOS CURSOS (HTTP://COURSES.BAELDUNG.COM)
TRABAJO DE CONSULTORÍA (HTTP://WWW.BAELDUNG.COM/CONSULTING)
META BAELDUNG (HTTP://META.BAELDUNG.COM/)
EL ARCHIVO COMPLETO (HTTP://WWW.BAELDUNG.COM/FULL_ARCHIVE)
ESCRIBIR PARA BAELDUNG (HTTP://WWW.BAELDUNG.COM/CONTRIBUTION-GUIDELINES)
CONTACTO (HTTP://WWW.BAELDUNG.COM/CONTACT)
INFORMACIÓN DE LA COMPAÑÍA (HTTP://WWW.BAELDUNG.COM/BAELDUNG-COMPANY-INFO)
TÉRMINOS DE SERVICIO (HTTP://WWW.BAELDUNG.COM/TERMS-OF-SERVICE)
POLÍTICA DE PRIVACIDAD (HTTP://WWW.BAELDUNG.COM/PRIVACY-POLICY)
EDITORES (HTTP://WWW.BAELDUNG.COM/EDITORS)
KIT DE MEDIOS (PDF) (HTTPS://S3.AMAZONAWS.COM/BAELDUNG.COM/BAELDUNG+-+MEDIA+KIT.PDF)