REliable, INtelligent & Scalable Systems

互联网应用开发技术

Web Application Development

第11课
WEB后端-依赖注入

Episode Eleven

Spring IoC

陈昊鹏

chen-hp@sjtu.edu.cn

Web Application Development

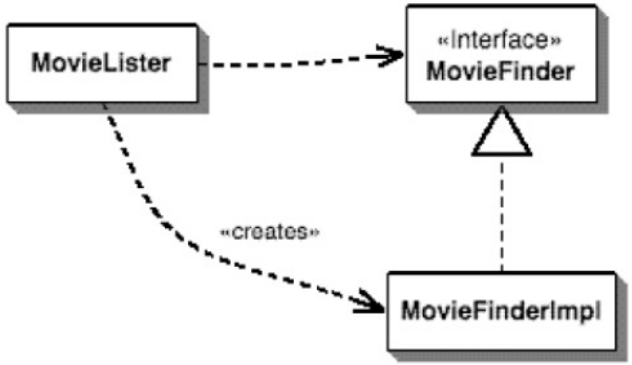
A Naive Example


```
class MovieLister...
  public Movie[] moviesDirectedBy(String arg) {
    List allMovies = finder.findAll();
    for (Iterator it = allMovies.iterator(); it.hasNext();) {
      Movie movie = (Movie) it.next();
      if (!movie.getDirector().equals(arg)) it.remove();
    }
    return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
}
```

- How we connect the `lister` object with a particular finder object?

A Naive Example


```
public interface MovieFinder { List findAll(); }
class MovieLister...
  private MovieFinder finder;
  public MovieLister() {
    finder = new ColonDelimitedMovieFinder("movies1.txt");
  }
```



- The `MovieLister` class is dependent on both the `MovieFinder` interface and upon the implementation!

Inversion of Control

- Problem
  - Implemented class of `MovieFinder` needn't connect to program during compiling the program.
    - Hope to plug-in concrete implemented class during run-time.
  - How to make `MovieLister` class to cooperate with other instances while they don't know the details of the implemented class?

- Solution
  - Inversion of Control

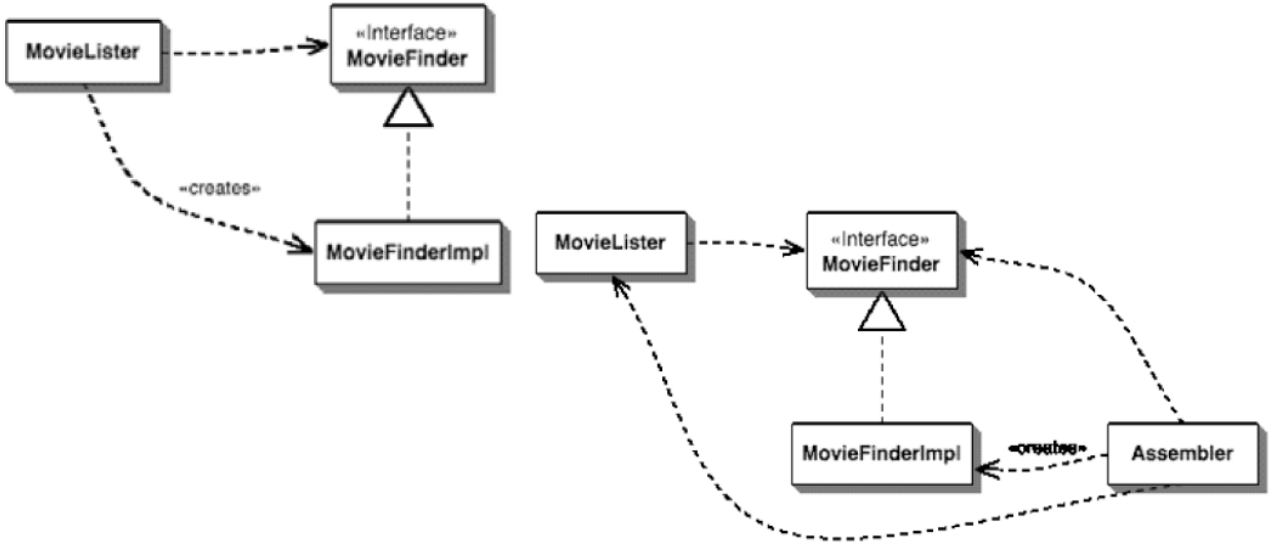
Inversion of Control

- What aspect of control are they inverting?
  - In naive example the `lister` looked up the finder implementation by directly instantiating it
    - This stops the finder from being a plugin.
  - The inversion is about how they lookup a plugin implementation.
    - Any user of a plugin follows some convention that allows a separate assembler module to inject the implementation into the `lister`.

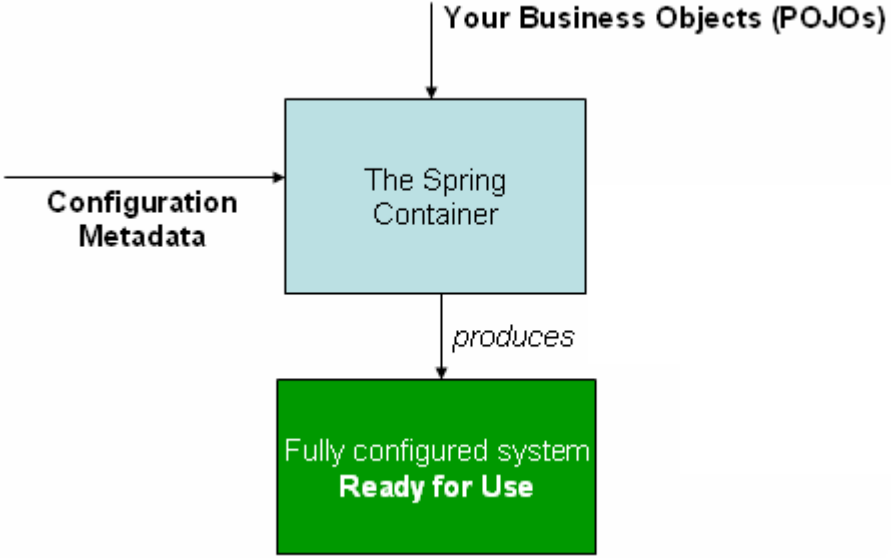
- Dependency Injection

Dependency Injection

- The basic idea of the Dependency Injection
  - Have a separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface



Spring Core Technologies - IoC container



An Example

- hello/MessageService.java

```
package hello;

public interface MessageService {
   String getMessage();
}
```

An Example

- hello/MessagePrinter.java

```
package hello;

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

@Component
public class MessagePrinter {

   @Autowired
   private MessageService service;

   public void printMessage() {
      System.out.println(this.service.getMessage());
   }
}
```

An Example

- hello/Application.java

```
@Configuration
@ComponentScan
public class Application {

  @Bean
  public MessageService mockMessageService() {
    return new MockMessageService();
    return new AnotherMessageService();
    return new MessageService() {
        public String getMessage() {
          return "Hello World!";
        }
    };
  }
```

An Example

- hello/Application.java

```
  public static void main(String[] args) {
    ApplicationContext context =
      new AnnotationConfigApplicationContext(Application.class);
    MessagePrinter printer = context.getBean(MessagePrinter.class);
    printer.printMessage();
  }
}
```

An Example

- hello/MockMessageService

```
package hello;
public class MockMessageService implements MessageService {
   public String getMessage() {
      return "Hello World! Mock Message Service!";
   }
}
```

- hello/AnotherMessageService

```
package hello;
public class AnotherMessageService implements MessageService {
   public String getMessage() {
      return "Hello World! Another Message Service!";
   }
}
```

Bean overview

- A Spring IoC container manages one or more *beans*.
- These beans are created with the configuration metadata that you supply to the container,
  - for example, in the form of XML <bean/> definitions.
- Within the container itself, these bean definitions are represented as BeanDefinition objects, which contain (among other information) the following metadata:
  - *A package-qualified class name:* typically the actual implementation class of the bean being defined.
  - Bean behavioral configuration elements, which state how the bean should behave in the container(scope, lifecycle callbacks, and so forth).
  - References to other beans that are needed for the bean to do its work; these references are also called *collaborators* or *dependencies*.
  - Other configuration settings to set in the newly created object,
    - for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool.
- This metadata translates to a set of properties that make up each bean definition.

Dependency injection

- Constructor-based dependency injection

```
public class ExampleBean {
  // No. of years to the calculate the Ultimate Answer
  private int years;
  // The Answer to Life, the Universe, and Everything
  private String ultimateAnswer;

  public ExampleBean(int years, String ultimateAnswer) {
    this.years = years;
    this.ultimateAnswer = ultimateAnswer;
  }

  public String answer() {
    return ultimateAnswer + " " + years;
  }
}
```

Dependency injection

- Constructor-based dependency injection

```
<?xml version="1.0" encoding="UTF-8"?>
<beans >
  <bean id="exampleBean" class="hello.ExampleBean">
    <constructor-arg type="int" value="7500000" />
    <constructor-arg type="java.lang.String" value="50" />
or
    <constructor-arg index="0" value="7500000" />
    <constructor-arg index="1" value="40" />
or
    <constructor-arg name="years" value="7500000" />
    <constructor-arg name="ultimateAnswer" value="42" />
   </bean>
</beans>
```

Dependency injection

- hello/ExampleBeanClient.java

```
package hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ExampleBeanClient {

  public static void main(String[] args) {
    ApplicationContext context =
        new ClassPathXmlApplicationContext(new String[] {"ExampleContext.xml"});
     ExampleBean bean = context.getBean(ExampleBean.class);
     System.out.println(bean.answer());
  }
}
```

Dependency injection

- Setter-based dependency injection

```
public class ExampleBean {
  // No. of years to the calculate the Ultimate Answer
  private int years;
  // The Answer to Life, the Universe, and Everything
  private String ultimateAnswer;

  public void setYears(int years) {
    this.years = years;
  }

  public void setUltimateAnswer(String ultimateAnswer) {
    this.ultimateAnswer = ultimateAnswer;
  }

  public String answer() {
    return ultimateAnswer + " " + years;
  }
}
```

Dependency injection

- Setter-based dependency injection

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="exampleBean" class="hello.ExampleBean">
    <property name="years">
        <value>7500000</value>
    </property>
    <property name="ultimateAnswer">
        <value>45</value>
    </property>
  </bean>
</beans>
```

Dependency injection

- **Constructor-based or setter-based DI?**
  - Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies.
  - The Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy, especially when properties are optional.
    - Setter methods also make objects of that class amenable to reconfiguration or re-injection later.
    - Management through JMX MBeans is a compelling use case.
  - Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state.
    - The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

Architecture

- Layered Architecture
  - Separation of Interface and Implementation
  - Entity – Auto mapped from database schema
  - Repository – Extended from existing lib class
  - Dao – Your own access control logic
  - Service – Business logic
  - Controller – Dispatch requests
  - IoC/DI – Independent of implementation

Entity Class

- Person.java

```
@Entity
@Table(name = "persons", schema = "test", catalog = "")
@JsonIgnoreProperties(value = {"handler","hibernateLazyInitializer","fieldHandler"})
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "personId")
public class Person {
  private int personId;
  private Integer age;
  private String firstname;
  private String lastname;

  @Id
  @Column(name = "PERSON_ID")
  public int getPersonId() {
    return personId;
  }
  .......
}
```

Repository Interface

- PersonRepository.java

```
package com.example.demo.repository;

import com.example.demo.entity.Person;
import org.springframework.data.jpa.repository.JpaRepository;

public interface PersonRepository extends JpaRepository<Person, Integer>{

}
```

DAO Interface

- PersonDAO.java

```
package com.example.demo.dao;

import com.example.demo.entity.Person;

public interface PersonDao {
   Person findOne(Integer id);
}
```

DAO Implementation

- PersonDAOImpl.java

```
@Repository
public class PersonDaoImpl implements PersonDao {
    @Autowired
    private PersonRepository personRepository;


    @Override
    public Person findOne(Integer id) {
      return personRepository.getOne(id);
    }
}
```

Service Interface

- PersonService.java

```
package com.example.demo.service;

import com.example.demo.entity.Person;

public interface PersonService {
   Person findEventById(Integer id);
}
```

Service Implementation

- PersonServiceImpl.java

```
@Service
public class PersonServiceImpl implements PersonService {

   @Autowired
   private PersonDao personDao;

   @Override
   public Person findEventById(Integer id){
      return personDao.findOne(id);
   }
}
```

Controller

- PersonController.java

```
@RestController
public class PersonController {

    @Autowired
    private PersonService personService;

    @GetMapping(value = "/findPerson/{id}")
    public Person findPerson(@PathVariable("id") Integer id) {
        System.out.println("Searching Person: " + id);
        return personService.findEventById(id);
    }
}
```

References

- Spring Document - 31. Working with SQL Databases,
  - https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-sql.html

- *Web开发技术*
- *Web Application Development*

Thank You!