软件体系结构 作业08

22920212204392 黄勖

1 什么是 IoC （Inversion of Control）、DIP （Dependency Inversion Principle）、Dependency Injection？请举例说明实现方式。

答：IoC - Inversion of Control，即“控制反转”，不是什么技术，而是一种设计思想。在Java开发中，Ioc意味着将你设计好的对象交给容器控制，而不是传统的在你的对象内部直接控制。如何理解好Ioc呢？理解好Ioc的关键是要明确“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”，那我们来深入分析一下：

●**谁控制谁，控制什么：**传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对象的创建；谁控制谁？当然是IoC 容器控制了对象；控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。

●**为何是反转，哪些方面反转了：**有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；哪些方面反转了？依赖对象的获取被反转了。

用图例说明一下，传统程序设计如图2-1，都是主动去创建相关对象然后再组合起来：

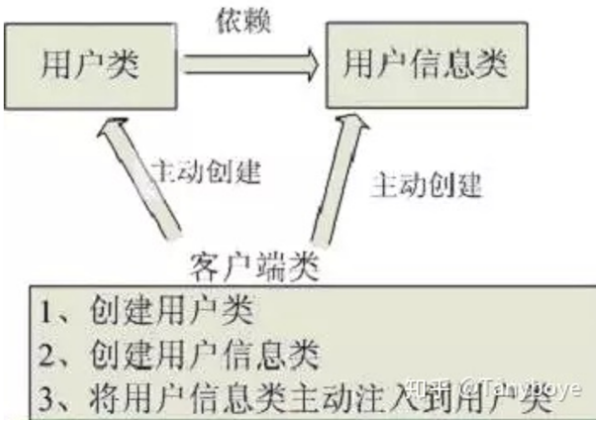

图2-1 传统应用程序示意图

当有了IoC/DI的容器后，在客户端类中不再主动去创建这些对象了，如图2-2所示:

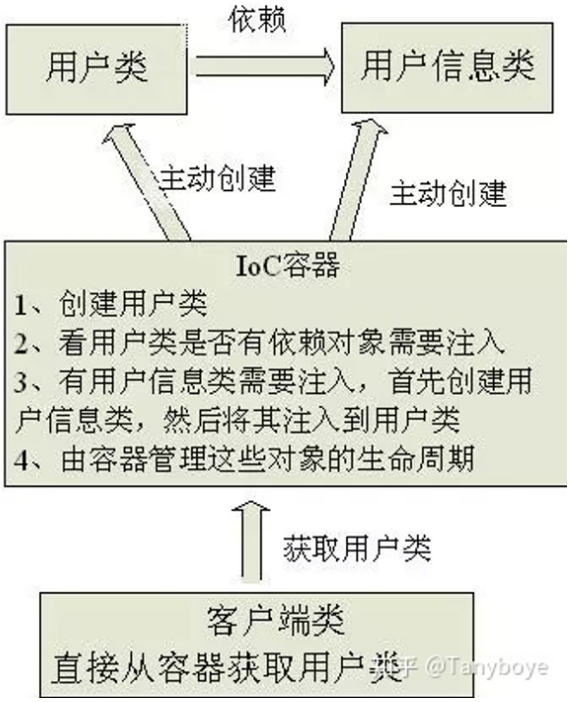

图2-2 有IoC/DI容器后程序结构示意图

`DI - Dependency Injection`，即“依赖注入”：是组件之间依赖关系由容器在运行期决定，形象的说，即由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台。通过依赖注入机制，我们只需要通过简单的配置，而无需任何代码就可指定目标需要的资源，完成自身的业务逻辑，而不需要关心具体的资源来自何处，由谁实现。

理解DI的关键是：“谁依赖谁，为什么需要依赖，谁注入谁，注入了什么”，那我们来深入分析一下：

●**谁依赖于谁：**当然是某个容器管理对象依赖于IoC容器；“被注入对象的对象”依赖于“依赖对象”；

●**为什么需要依赖：**容器管理对象需要IoC容器来提供对象需要的外部资源；

●**谁注入谁：**很明显是IoC容器注入某个对象，也就是注入“依赖对象”；

●**注入了什么：**就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）。

IoC和DI由什么关系呢？其实它们是同一个概念的不同角度描述，由于控制反转概念比较含糊（可能只是理解为容器控制对象这一个层面，很难让人想到谁来维护对象关系），所以2004年大师级人物Martin Fowler又给出了一个新的名字：“依赖注入”，相对IoC 而言，**“依赖注入”明确描述了“被注入对象依赖IoC容器配置依赖对象”。**

依赖倒置原则（Dependence Inversion Principle，DIP）是指设计代码结构时，高层模块不应该依赖低层模块，二者都应该依赖其抽象。

抽象不应该依赖细节，细节应该依赖抽象。通过依赖倒置，可以减少类与类之间的耦合性，提高系统的稳定性，提高代码的可读性和可维护性，并且能够降低修改程序所造成的风险。

DIP示例代码：

```
interface Course{
    public void study();
}
class JavaCourse implements Course{
    public void study(){
        System.out.println("正在学习Java");
    }
}
class CCourse implements Course{
    public void study(){
        System.out.println("正在学习C");
    }
}
class Maoli{
    public void study(Course course){
        course.study();
    }
}

public class Main {
    public static void main(String[] args) {
        Maoli maoli = new Maoli();
        maoli.study(new JavaCourse());
        maoli.study(new CCourse());
    }
}
```

运行结果：

IOC示例代码:

IAccountDao.java:

```
/**
 * 模拟保存账户
 */
public interface IAccountDao {
    void saveAccount();
}
```

AccountDaoImpl.java:

```
/**
 * 账户的持久层实现类
 */
public class AccountDaoImpl implements IAccountDao {
    public  void saveAccount(){
        System.out.println("保存了账户");
    }
}
```

编写bean.xml文件:

```
<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">
    <bean id="accountDao" class="com.chester.dao.impl.AccountDaoImpl"></bean>
</beans>
```

编写main函数:

```java
/**
 * 获取Spring的Ioc核心容器，并生成Bean对象
 * @param args
 */
public static void main(String[] args) {
    ApplicationContext ac = new ClassPathXmlApplicationContext("bean.xml");
    IAccountDao as = (IAccountDao)ac.getBean("accountDao");
    as.saveAccount();
}
```

运行结果: