IOC理论

1 原先的开发步骤

真正的开发不会这样子，这里只是举个例子，方便区分原先开发和IOC技术。

1.1 只有一个需求的时候

- UserDao——接口

```
public interface UserDao {
    void getUser();
}
```

- UserDaoImpl——接口实现类

```
public class UserDaoImpl implements UserDao {
    public void getUser() {
        System.out.println("获取到用户信息了");
    }
}
```

- UserService——业务接口

```
public interface UserService {
    void getUser();
}
```

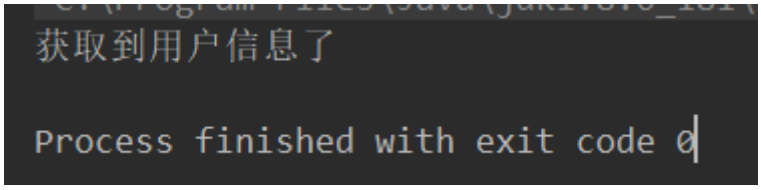
- UserServiceImpl——业务接口实现类

```
public class UserServiceImpl implements UserService {
    UserDao userDao = new UserDaoImpl();

    public void getUser() {
        userDao.getUser();
    }
}
```

- 测试运行

  用户实际调用的是Service层，dao层他们不需要接触。

```
@Test
public void getUser(){
    UserService userService = new UserServiceImpl();

    userService.getUser();
}
```



1.2 当有多个需求的时候

- 多个DaoImpl：

```java
public class UserDaoMySqlImpl implements UserDao {
    public void getUser() {
        System.out.println("获取到用户的Mysql数据信息");
    }
}
```

```java
public class UserDaoOracleImpl implements UserDao {
    public void getUser() {
        System.out.println("获取到用户的oracle数据");
    }
}
```

- 这时，如果用户需要获取mysql，那么需要在UserServiceImpl中修改。

```java
public class UserServiceImpl implements UserService {
    UserDao userDao = new UserDaoMySqlImpl();

    public void getUser() {
        userDao.getUser();
    }
}
```

如果，需要获取oracle的数据：

```java
public class UserServiceImpl implements UserService {
    UserDao userDao = new UserDaoOracleImpl();

    public void getUser() {
        userDao.getUser();
    }
}
```

- 显然这样子修改，十分的麻烦，而且只要用户需求变了，那么就需要更改我们的源代码，这是不可能存在的。
- 这样子，是通过**程序控制对象**（我们书写代码来决定使用哪个）

1.3 set注入进行改进

- 给UserDao一个set方法。

```java
public class UserServiceImpl implements UserService {
    UserDao userDao;

     //利用set进行动态实现值的注入
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    public void getUser() {
        userDao.getUser();
    }
}
```

- 当想要mysql的信息时：

```java
@Test
public void getUser() {
    UserService userService = new UserServiceImpl();
    ((UserServiceImpl) userService).setUserDao(new UserDaoMySqlImpl());

    userService.getUser();
}
```

- 当想要oracle的信息时：

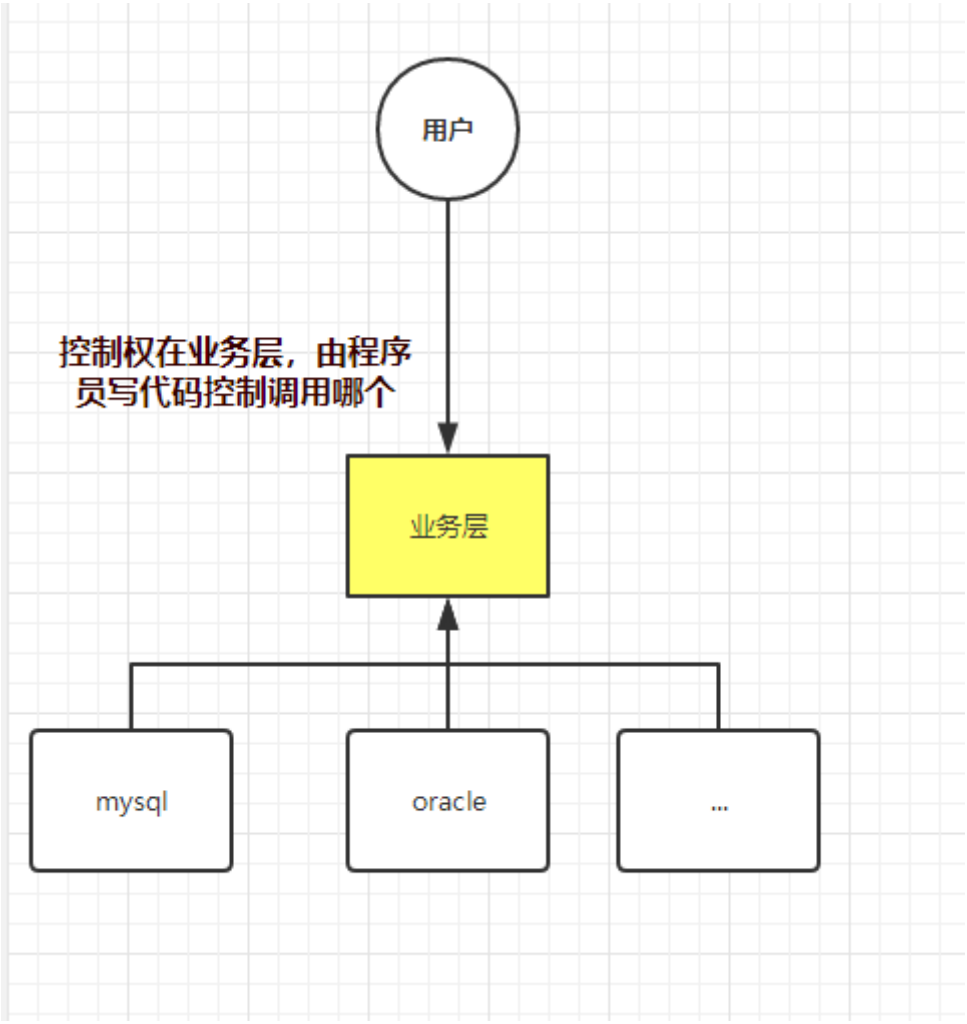
```java
@Test
public void getUser() {
    UserService userService = new UserServiceImpl();
    ((UserServiceImpl) userService).setUserDao(new UserDaoOracleImpl());

    userService.getUser();
}
```

- 当想要用户信息时：

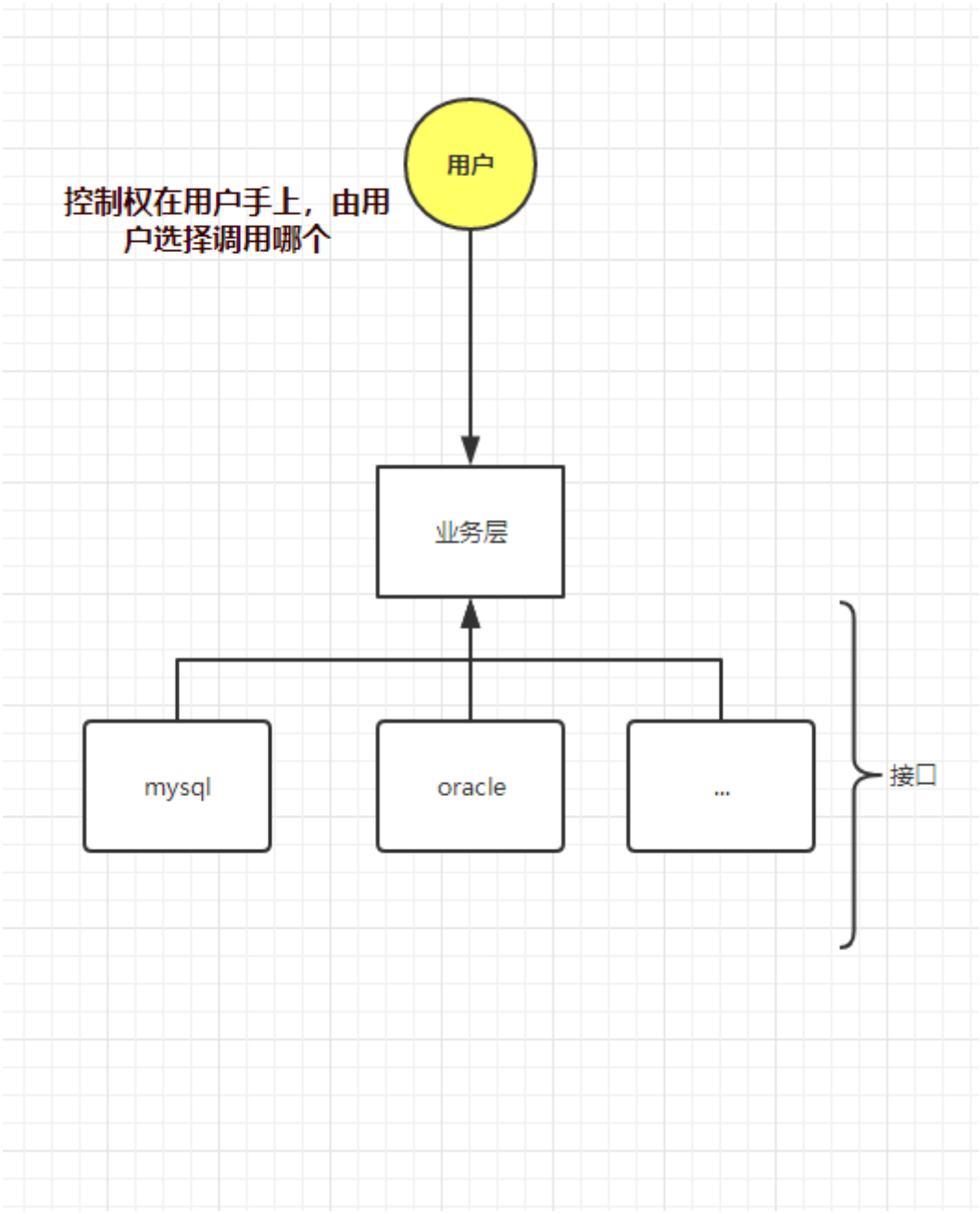
```java
public class test {
    @Test
    public void getUser() {
        UserService userService = new UserServiceImpl();
        ((UserServiceImpl) userService).setUserDao(new UserDaoImpl());

        userService.getUser();
    }

}
```

1.4 小结一下

- 在我们之前的业务中，用户的需求可能会影响我们原来的代码，我们需要根据用户的需求去修改源代码。如果程序代码量十分大，那么修改一次的成本会十分昂贵。
- 然而，我们用了set注入以后，解决了问题。
- 对比一下set和原先的模式：
    - 之前，是程序**主动创建对象**（控制权在程序员手上）一旦new了，那么就不可以改变了。



- 使用set注入，程序不再具有主动性，而是变成了**被动地接收对象。**



    - 这样子，就是控制反转（控制权变更了）
- 这种思想，从本质上解决了问题，程序员不用再去管理对象的创建，系统的耦合性大大降低，可以更加专注在业务的实现上。
- set的这种思想，只是IOC的原型。

多余的拓展参考下狂神的这篇博客，写的十分好，课也不错，后续的笔记都是来自这里（bilibili课程），加上自己的总结和看书。：

https://www.cnblogs.com/hellokuangshen/p/11249253.html

2 IOC本质

控制反转IoC(Inversion of Control)，是一种设计思想，DI(依赖注入)是实现IoC的一种方法，也有人认为DI只是IoC的另一种说法。没有IoC的程序中，我们使用面向对象编程，对象的创建与对象间的依赖关系完全硬编码在程序中，对象的创建由程序自己控制，控制反转后将对象的创建转移给第三方，个人认为所谓控制反转就是：**获得依赖对象的方式反转了。**

采用XML方式配置Bean的时候，Bean的定义信息是和实现分离的，而采用注解的方式可以把两者合为一体，Bean的定义信息直接以注解的形式定义在实现类中，从而达到了零配置的目的。

控制反转是一种通过描述（XML或注解）并通过第三方去生产或获取特定对象的方式。在Spring中实现控制反转的是IoC容器，其实现方法是依赖注入（Dependency Injection,DI）。