Published in 程式爱好者

YC Following
Sep 21, 2020 · 5 min read

Save

使人疯狂的SOLID 原则：依赖反向原则 (Dependency Inversion Principle)

> 今天我们要说的是最后一个原则：依赖反向原则 (DIP)。

> 定义：高层模块不应依赖低层模块，它们都应依赖于抽象接口。抽象接口不应该依赖于具体实作，具体实作应依赖抽象接口。

什么是高层模块，什么是低层模块?

低层模块：该模块的实现都是不可分割的原子逻辑层，如 MVC 中的 Model 层。

高层模块：该模块的业务逻辑多是由低层模块组合而成，如 MVC 中的 Controller 层与 Client 端。

「正常的」程式，可能会有很多高层模块依赖于底层模块。我们常常看到的可能是像以下：

```
class Controller {

    Mysql mysqlDB = new Mysql();
    string dbHost = mysqlDB.host;

}
```


那如果写成以下的方式：

```
class Controller {
    private Mysql mysqlDB;

    public Controller(Mysql mysqlDB)
    {
        this.mysqlDB = mysqlDB;
    }

    string dbHost = this.mysqlDB.host;

}
```

这样写是可以算是进了一步，我们使用了**依赖注入来把 Mysql 放到 Controller 中，而不是直接在 Controller 中 new 出一个新的 object 出来，**可以算是解耦了一点点。

那到底怎么样写才是乎合 **DIP** 原则的写法**？*多用抽象层！***

```
class Controller {
    private Database database;

    public Controller(Database database) //只注入 Database 接口
    {
        this.database = database;
    }

    string dbHost = this.database.gethost();
}

interface Database {
    public string getHost();
    public string getPort();
    public string getUsername();
    public string getPassword();
}

class Mysql implement Database {
    public string getHost() {
        ...
        return host;
    }
    ...
}
```


以上的例子，我们可以看到在 Controller 当中只依赖于抽象层 (interface Database)，而 Mysql 类别也同样赖于相同的抽象层。今天就算是要换成 MonogoDB，那只要一样用 MonogoDB 类别赖于 Database 接口，Controller不用做出改变就可以加以扩充。

这样的做法其实是一种将**低层模块的控制权从原来的高层模块中抽离，将两者的耦合只放在抽象层上**。

物件的创建则应以抽象工厂模式进行，同样的原因，就是避免高度耦合的发生。我们来看看以下例子：

```
class Application {
    ...
    private DatabaseFactory db = new DatabaseFactoryImpl();
    private Database = db.makeDb();
}

interface DatabaseFactory {
    public void makeDb();
}

class DatabaseFactoryImpl implment DatabaseFactory {
    public void makeDb() {
        return new Mysql();
    }
}

interface Database {
    public string getHost();
    public string getPort();
    public string getUsername();
    public string getPassword();
}

class Mysql implement Database {
    public string getHost() {
        ...
        return host;
    }
    ...
}
```

以上的工厂能让高层模块最低限度地依赖其他的模块 (只依赖于 DatabaseFactory 接口)，而实现逻辑的会是于 DatabaseFactoryImpl 中发生。
同时，DatabaseFactory 产生的物件又能被 Database 接口所使用，让 Application 这个类别完全不必接触到非抽象层，将耦合性大大降低。

有些朋友会想到那在更大的范围来看软件时，那程式不是一样依赖于 Framework 跟 Database 吗?

对的，没错，程式的确是依赖于这样不够抽象的层次，但是

> *我们知道可以信任他们不会改变，所以我们可以容忍那些具体的依赖关系。*

不是所有时候都需要 100% 按照 SOLID 原则来设计，这样只会变得没完没了，总会得到一个无法再相依赖于抽象层的模块。

> *不要过度设计。*

以上， 我理解的 SOLID原则终于告一段落，谢谢各位收看。

如果你觉得我的文章帮助到你，希望你也可以为文章拍手，分别 Follow 我的个人页与程式爱好者出版，按赞我们的粉丝页喔，支持我们推出更多更好的内容创作！

Object Oriented Solid Dip Software Architecture