第一部分　架构设计

http://www.neubooks.com

第1章　Java Web应用架构设计

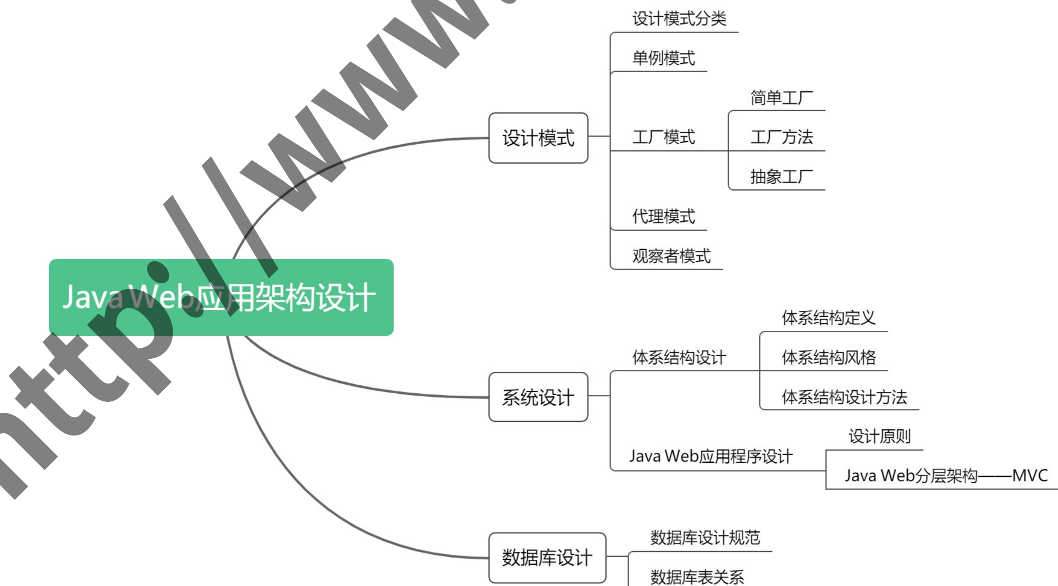
●本章知识点

设计模式
系统设计
数据库设计

●所支撑的职业技能

通过本章的学习，能够理解和掌握Java开发常用的设计模式(如单例模式、工厂模式、代理模式、观察者模式等)；能够应用Java开发常用设计模式进行Java Web应用程序的架构设计；能够应用数据库的编码规范进行数据库表结构与关系的设计。

●学习路线



Java Web应用基于B/S(Brower/Server)架构，摆脱了传统C/S结构应用在使用时需要安装客户端软件的麻烦，让用户随时随地通过网络服务浏览和访问服务器的资源，为用户提

http://www.neubooks.com

供图形化的、易于访问的直观界面,如图 1-1 所示。每个 Java Web 应用都是多个 Web 资源的集合,由多个静态 Web 资源和动态 Web 资源组成。

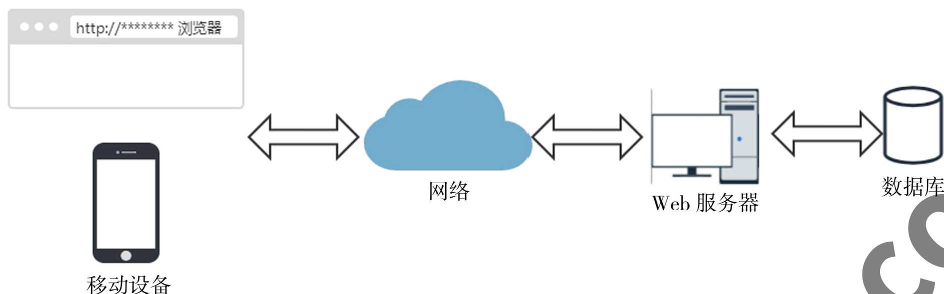


图 1-1 Java Web 应用

虽然每个 Java Web 应用都在为不同的用户服务,提供着截然不同的功能服务,但是在实现应用背后所使用的架构设计都是比较稳定的。一个合理、良好的架构,不仅让开发人员在开发时提高效率,更重要的是交付的应用能够健壮、稳定;而架构设计得不合理,不仅会让开发人员在开发时“受苦受难”,同时交付的应用本身的生命周期更是受到严重威胁。

本章将从设计模式、系统设计、数据库设计这三方面来介绍 Java Web 应用架构设计。

1.1 设计模式

本节知识点

知识点	了解	掌握	熟练	精通
设计模式分类	√			
单例模式				√
简单工厂				√
工厂方法			√	
抽象工厂			√	
代理模式		√		
观察者模式			√	

设计模式是优化的、可重用的、经过实践证明有效的解决方案,用于解决软件开发中反复出现的设计问题。设计模式并不是一段特定的代码实现可以直接在代码中使用,而是描述解决特定问题的一种思想,使用过程中你也很难直接在自己的程序中套用某个设计模式,都需要根据实际情况进行调整,来符合自己的场景解决实际问题。

设计模式和算法常常被混淆,它们都是对已知的、反复的问题提供解决方案,但是算法指出的是为了得到特定结果所要进行的一系列步骤,而设计模式则是对解决方案的更高层次描述。不同的人在使用同一模式时实现代码会不同,甚至同一个人在不同的程序中使用

http://www.neubooks.com

统一模式时实现代码也会不同。

算法更像是我们做菜时的菜谱，按照算法提供的步骤，我们就可以做出色香味俱全的菜。而模式更像是思想，告诉你使用这种方法就可以解决特定问题，但需要自己根据实际情况去确定实现步骤。

设计模式的描述通常会包括以下部分：

(1)模式名称

根据模式的问题、特点、解决方案、功能和效果，通常使用一个到两个词来描述模式，方便沟通和讨论。

(2)问题

描述该设计模式解决的问题和问题存在的前因后果，以及满足什么条件可使用该模式。

(3)解决方案

描述设计模式的每个部分和它们之间的关系。

1.1.1　设计模式分类

GoF(“四人组”，又称 Gang of Four，即 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 四人)提出了经典的 23 种设计模式，我们常常根据模式的目的(用来完成什么工作)来划分，分为结构型模式、创建型模式和行为型模式 3 种。

(1)结构型模式

结构型模式是组织不同的类和对象以形成更大的结构并提供新的功能，主要处理实体之间的关系，将对象和类组装成较大的结构，使这些实体更容易合作，并保持结构的灵活和高效。

(2)创建型模式

创建型模式是类实例化或对象创建，主要提供实例化机制，创建对象变得更容易，增加已有代码的灵活性和可复用性。

(3)行为型模式

行为型模式是识别对象之间常见的通信模式并进行实现，主要用于实体之间的通信，负责对象间的高效沟通和职责委派，使这些实体更容易、更灵活地进行通信。

23 种设计模式的具体划分如图 1-2 所示。这 23 种设计模式很多时候不是单独存在的，它们之间存在关联性，我们为了解决实际问题，可能需要同时使用好几种设计模式。下面对这 23 个经典设计模式进行简单介绍。

(1)单例(Singleton)模式

一个类只能生成一个实例，且只提供可供调用的单个实例和对该实例的全局访问。

(2)原型(Prototype)模式

把一个对象作为原型，在不影响性能和内存的情况下，通过对其进行复制而创建出新实例。

(3)工厂方法(Factory Method)模式

定义一个用于创建产品的接口，由子类决定生产什么产品。

http://www.neubooks.com

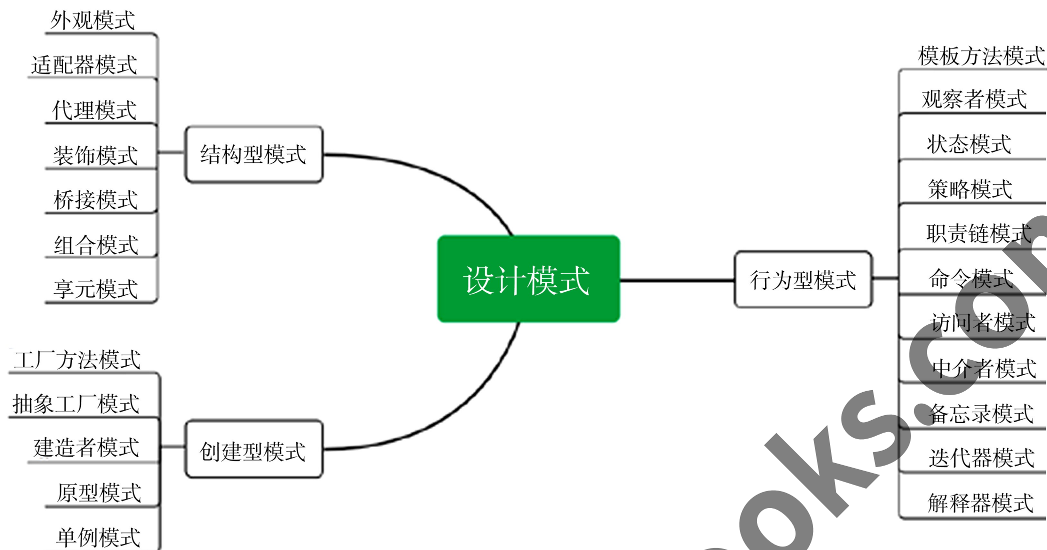


图 1-2 经典设计模式分类

(4)抽象工厂(Abstract Factory)模式

提供创建对象的接口,而不指定具体产品,由子类去实现;每个子类可以创建一类产品。

(5)建造者(Builder)模式

将构造与表示分离,把复杂的对象分解成多个简单的部分,逐步地构造,最后返回复杂对象的实例。

(6)代理(Proxy)模式

为某对象提供一种代理以控制对该对象的访问,即客户端通过代理间接地访问该对象,从而限制、增强或修改该对象的一些特性。

(7)适配器(Adapter)模式

将一个类的接口转换成客户希望的另外一个类的接口,使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

(8)桥接(Bridge)模式

将抽象与实现分离,使它们可以独立变化。它是用组合关系代替继承关系来实现的,从而降低抽象和实现这两个可变维度的耦合度。

(9)装饰(Decorator)模式

动态地给对象增加一些职责,即增加其额外的功能。

(10)外观(Facade)模式

为多个复杂的子系统提供一个一致的接口,使这些子系统更加容易被访问。

(11)享元(Flyweight)模式

运用共享技术来有效地支持大量细粒度对象的复用。

(12)组合(Composite)模式

将对象组合成树状层次结构,使用户对单个对象和组合对象具有一致的访问性。

http://www.neubooks.com

(13)模板方法(Template Method)模式

定义一个操作中的算法骨架,而将算法的一些步骤延迟到子类中,使得子类可以在不改变该算法结构的情况下重定义该算法的某些特定步骤。

(14)策略(Strategy)模式

定义了一系列算法,并将每个算法封装起来,使它们可以相互替换,且算法的改变不会影响使用算法的客户。

(15)命令(Command)模式

将一个请求封装为一个对象,使发出请求的责任和执行请求的责任分割开。

(16)职责链(Chain of Responsibility)模式

把请求从链中的一个对象传到下一个对象,直到请求被响应为止,通过这种方式去除对象之间的耦合。

(17)状态(State)模式

允许一个对象在其内部状态发生改变时改变其行为能力。

(18)观察者(Observer)模式

多个对象间存在一对多关系,当一个对象发生改变时,把这种改变通知给其他多个对象,从而影响其他对象的行为。

(19)中介者(Mediator)模式

定义一个中介对象来简化原有对象之间的交互关系,降低系统中对象间的耦合度,使原有对象之间不必相互了解。

(20)迭代器(Iterator)模式

提供一种方法来顺序访问聚合对象中的一系列数据,而不暴露聚合对象的内部表示。

(21)访问者(Visitor)模式

在不改变集合元素的前提下,为一个集合中的每个元素提供多种访问方式,即每个元素有多个访问者对象访问。

(22)备忘录(Memento)模式

在不破坏封装性的前提下,获取并保存一个对象的内部状态,以便以后恢复它。

(23)解释器(Interpreter)模式

提供如何定义语言的文法,以及对语言句子的解释方法。

1.1.2　单例模式

单例模式是一种创建型设计模式,它确保我们的代码在运行期间有且只有一个特定类的单个实例,并提供对单个实例的全局访问点。

1. 结构和说明

单例模式的结构如图1-3所示。

http://www.neubooks.com

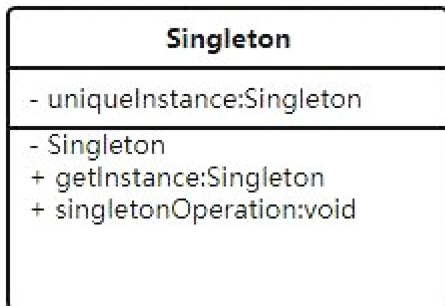


图 1-3 单例模式结构图

Singleton 是要实现单例的类，自己负责创建自己的唯一实例，私有化构造方法，提供静态的 getInstance 方法，让外部通过这个方法来获取唯一实例。

2. 实现示例

在 Java 中，单例模式的实现分为两种，一种是饿汉式，一种是懒汉式，它们之间的区别仅仅是在具体创建对象实例的时候使用了不同的实现方式。接下来我们分别看看这两种单例模式的实现。

(1)懒汉式

懒汉式在类加载时不初始化实例，延迟加载，仅当第一次被调用时进行创建。

【实例 1-1】

【任务要求】

实现懒汉式的单例模式。

【任务实现】

①私有化构造方法

如果想在运行期间控制某一个类的实例只有一个，首先就是要控制创建实例的方法，不能随随便便地去创建一个类的实例，否则就无法控制所创建类的实例个数。

那如何限制类的外部不能创建一个类的实例呢？从一个类的创建过程我们可以想到，将类的构造方法私有化以后就无法从外部来创建类的实例了。

```
private Singleton(){

}
```

②提供获取实例的方法

构造方法在私有化以后，当外部类使用这个类的时候无法创建类的实例，没有办法调用这个对象的方法，不能实现功能调用。私有化的方法只能在这个类内部访问，所以要通过这个单例类本身提供一个方法来返回该类的实例。

```
public Singleton getInstance(){


}
```

http://www.neubooks.com

③将获取实例的方法变成静态

有了获取实例的方法，但是外部类仍然无法访问，因为没有这个类的实例就无法访问类的实例方法，但是外部类还需要通过这个方法来获取实例。所以要考虑是否能不通过类的实例去访问方法。答案是通过 static 关键字，我们在返回实例类的方法前加上 static，让这个方法变成静态方法，就可以直接通过该类来调用这个方法了，而不需要先去实例化类。

```
public static Singleton getInstance(){


}
```

④定义静态属性

有了静态的方法，外部类可以直接调用这个方法了，但是如果直接在这个方法中创建并返回一个实例，那每次都是新创建了一个实例，与单例模式不符。为了解决这个问题，就需要用一个类的属性来记录已经创建过的实例，当第一次创建后，就把这个实例保存下来，以后每次请求的时候就返回这个实例，而不是重复创建。

```
private static Singleton instance  = null;
```

⑤完整地实现

有了静态属性和静态方法，当外部类请求获取实例时，我们需要在静态方法中进行判断，如果静态属性有了值，则直接返回；如果没有值（第一次被请求），则进行创建，并赋值给静态属性，再进行返回。

```
public class Singleton {
    //定义私有化的静态属性
    private static Singleton instance  = null;

    //私有化构造方法
    private Singleton() {

    }

    //获取实例的静态方法
    public static Singleton getInstance() {
        //判断实例变量是否有值，如果没有则进行创建
        if(instance == null){
            instance = new Singleton();
        }
        //如果已经有实例了，直接返回
        return instance;

    }

}
```

http://www.neubooks.com

(2)饿汉式

饿汉式与懒汉式的实现步骤相似,仅仅是在 getInstance 方法的实现上有所区别。在懒汉式中,静态属性在类装载时并没有被实例化,是 null,只有在外部类第一次调用 getInstance 时进行创建。在饿汉式中,使用了 static 的特性:

- static 变量在类装载的时候进行初始化。
- 多个实例的 static 变量会共享同一块内存区域。

【实例 1-2】

【任务要求】

实现饿汉式的单例模式。

【任务实现】

在属性定义时直接创建实例,getInstance 方法不进行判断,直接返回实例。

```
public class Singleton {
    //定义私有化的静态属性,并赋值实例
    private static Singleton instance  = new Singleton();

    //私有化构造方法
    private Singleton() {

    }

    //获取实例的静态方法
    public static Singleton getInstance() {

        //属性在定义时已创建实例,直接返回
        return instance;

    }
}
```

1.1.3 工厂模式

工厂模式也是一种创建型设计模式,在 Java 程序系统中随处可见。根据产品是具体产品还是具体工厂可分为简单工厂模式和工厂方法模式;根据工厂的抽象程度可分为工厂方法模式和抽象工厂模式。

1. 简单工厂模式

简单工厂模式提供一个创建对象实例的功能,而无须关心其具体实现。被创建实例的类型可以是接口、抽象类,也可以是具体的类。

http://www.neubooks.com

(1)结构和说明

简单工厂模式的结构如图1-4所示。

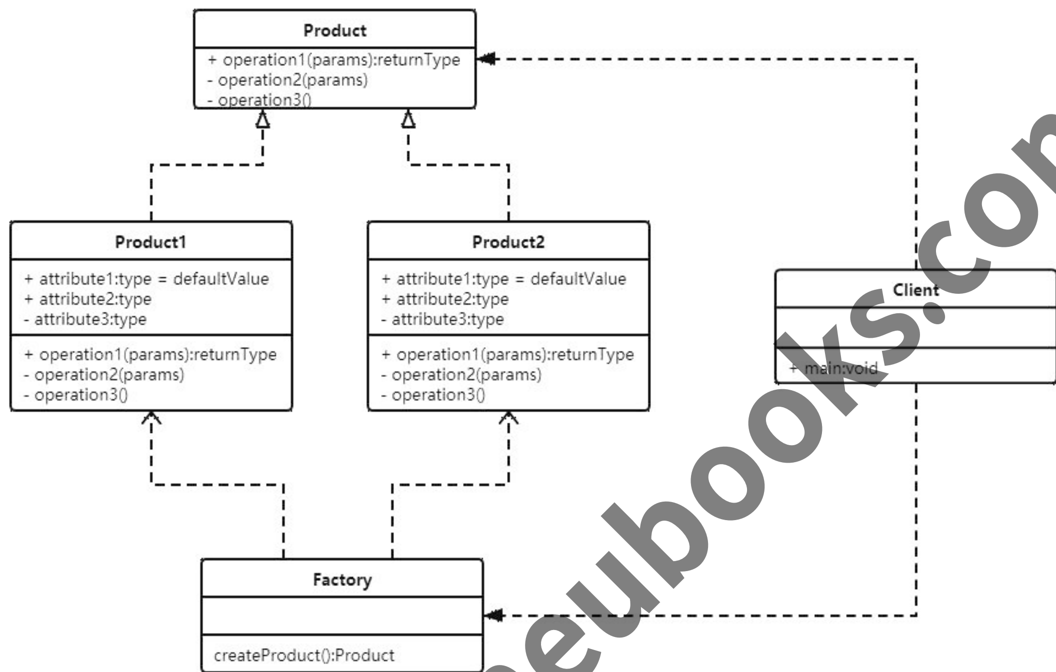


图1-4 简单工厂模式

- Factory:工厂类,选择合适的实现类来创建Product实例;
- Client:客户端,获取消费Product实例;
- Product:产品接口,定义客户所需要的功能接口;
- Product1和Product2:具体产品实现。

简单工厂模式的本质就是选择实现,其目的在于为客户端选择相应的实现,而不是由工厂类去实现,从而使得客户端和实现之间解耦。这样一来,具体实现发生了变化,就不会引起客户端发生变化。

(2)实现示例

简单工厂模式直接通过一个Factory类创建多个实体类的构造方式,在实际项目中用得比较多。

【实例1-3】

【任务要求】

实现简单工厂模式。

http://www.neubooks.com

【任务实现】

①产品 Product 定义

```
/**
 *产品接口,定义客户端要使用的功能接口
 */
public interface Product {

    //示例,具体功能方法的定义
    public void operation(String s);

}
```

②产品 Product 具体实现类

```
/**
 * Product 的具体实现类
 */
public class ProductImplA implements  Product{

    //实现具体的功能
    @Override
    public void operation(String s) {
        System.out.println("ProductImplA s:" + s);
    }
}
```

```
/**
 * Product 的具体实现类
 */
public class ProductImplB implements Product{

    //实现具体的功能
    @Override
    public void operation(String s) {
        System.out.println("ProductImplB s:" + s);
    }
}
```

http://www.neubooks.com

③简单工厂 Factory 的实现

```
/**
 * 工厂类,用来创建 Product 实例
 */
public class Factory {

    /**
     * 具体创建 Product 实例的方法
     * @param condition 从外部传入的选择条件
     * @return 返回创建好的 Product 实例
     */
    public static Product createProduct(int condition){
        Product product = null;
        if(condition == 1){
            product = new ProductImplA();
        }else if(condition == 2){
            product = new ProductImplB();
        }
        return product;
    }

}
```

④客户端消费 Product 实例的实现

```
/**
 * 客户端,使用 Product 接口
 */
public class Client {
    public static void main(String[] args){
        //通过简单工厂来获取 Product 实例
        Product product = Factory.createProduct(1);
        product.operation("使用简单工厂");
    }
}
```

2. 工厂方法模式

工厂方法模式定义一个用于创建对象的接口,但并不关心具体的实现,而由工厂子类负责生成具体的产品对象,通过 Factory Method 使一个类的实例化延迟到其子类。

http://www.neubooks.com

(1)结构和说明

工厂方法模式的结构如图 1-5 所示。

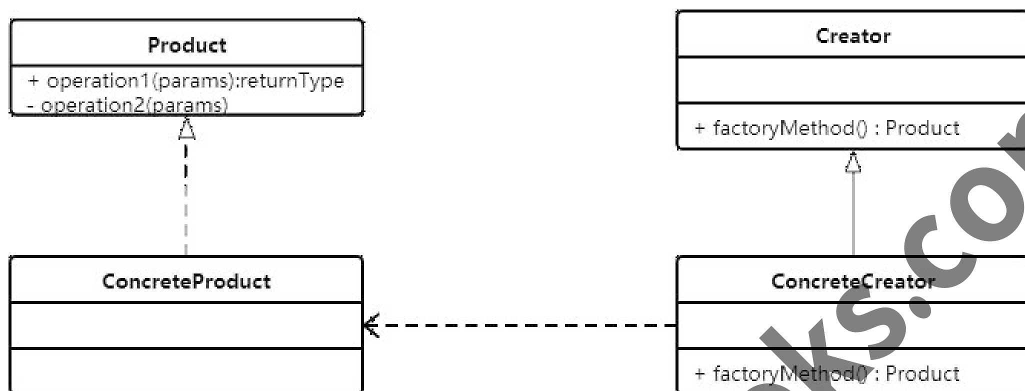


图 1-5 工厂方法模式

- Product:定义工厂方法所创建的对象的接口,也就是实际需要使用的对象的接口。
- ConcreteProduct:具体的 Product 接口的实现对象。
- Creator:创建器,声明工厂方法,一般是抽象方法。也可以在 Creator 里面提供工厂方法的默认实现,让工厂方法返回一个缺省的 Product 类型的实例对象。
- ConcreteCreator:具体的创建器对象,实现 Creator 定义的抽象工厂方法,返回具体的 Product 实例。

(2)实现示例

工厂方法模式是对简单工厂模式的进一步抽象化,让软件对象的生产和使用相分离,系统能够在不修改原来代码的情况下引进新的功能,满足开闭原则。

【实例 1-4】

【任务要求】

实现工厂方法模式。

【任务实现】

①产品 Product 的实现

```
/**
 * 工厂方法所创建的产品接口
 */
public interface Product {
    //具体业务属性和方法
}
```

http://www.neubooks.com

②产品 Product 具体实现类

```
/**
 * Product 的具体实现类
 */
public class ConcreteProduct implements Product {
    //实现 Product 要求的业务方法
}
```

③创建器 Creator 的实现

```
/**
 *创建器抽象类,用于声明工厂方法
 */
public abstract class Creator {
    /**
     *创建 Product 的抽象方法
     * @return Product 对象
     */
    protected abstract Product factoryMethod();

}
```

④具体创建器 ConcreteCreator 的实现

```
/**
 *具体的创建器对象
 */
public class ConcreteCreator extends Creator{
    /**
     *实现 Creator 的抽象工厂方法,创建具体的产品 Product 对象
     * @return Product 对象
     */
    @Override
    protected Product factoryMethod(){
        //返回一个具体的 Product 对象
        return new ConcreteProduct();
    }
}
```

3. 抽象工厂模式

简单工厂模式和抽象工厂模式解决的是单个产品对象的创建,每一个创建方法只关注自己的产品。如果需要创建一系列的产品对象,则需要用到抽象工厂模式。

http://www.neubooks.com

(1)结构和说明

抽象工厂模式的结构如图 1-6 所示。

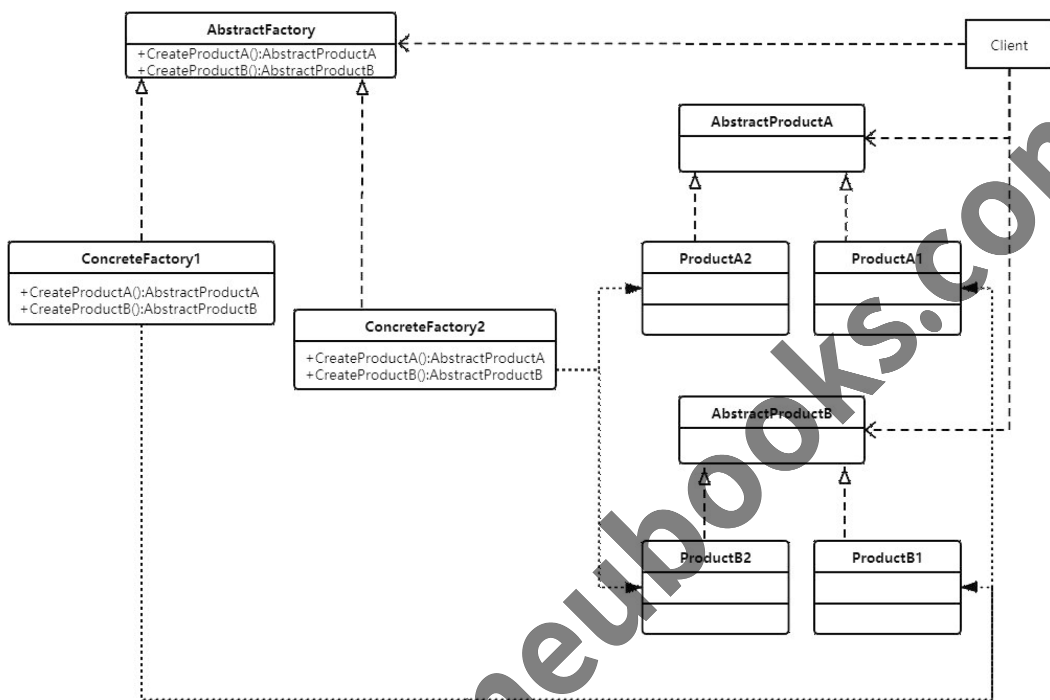


图 1-6　抽象工厂模式

- AbstractFactory:抽象工厂,定义创建一系列产品对象的操作接口。
- ConcreteFactory:具体工厂,实现抽象工厂所定义的方法,选择具体的产品实现类创建一系列产品对象。
- AbstractProduct:定义一类产品对象的接口。
- ProductA/ProductB:具体的产品实现对象。
- Client:客户端,主要使用抽象工厂来获取一系列所需要的产品对象,然后面向这些产品对象的接口编程,完成实际业务功能。

(2)实现示例

工厂模式只考虑每次生产一种产品,而抽象工厂模式将产品集中起来一次生产一个产品族。

【实例 1-5】

【任务要求】

实现抽象工厂模式。

http://www.neubooks.com

【任务实现】

①抽象产品 AbstractProductA 和 AbstractProductB 的实现

```
/**
 *抽象产品A的接口
 */
public interface AbstractProductA {
    //定义抽象产品方法

}
/**
 *抽象产品B的接口
 */
public interface AbstractProductB {
    //定义抽象产品方法
}
```

②抽象工厂 AbstractFactory 的实现

```
/**
 *抽象工厂的接口,声明创建抽象产品对象
 */
public interface AbstractFactory {
    /**
     *抽象方法,创建产品A的对象
     * @return
     */
    public AbstractProductA createProductA();
    /**
     *抽象方法,创建产品B的对象
     * @return
     */
    public AbstractProductB createProductB();
}
```

③具体产品A的实现

```
/**
 *产品A的具体实现
 */
public class ProductA1 implements AbstractProductA{
    //实现产品A的接口中定义的操作
}
/**
 *产品A的具体实现
 */
public class ProductA2 implements AbstractProductA{
    //实现产品A的接口中定义的操作
}
```

http://www.neubooks.com

④具体产品 B 的实现

```
/* *
 * 产品 B 的具体实现
 */
public class ProductB1 implements AbstractProductB{
    //实现产品 B 的接口中定义的操作
}
/* *
 * 产品 B 的具体实现
 */
public class ProductB2 implements AbstractProductB{
    //实现产品 B 的接口中定义的操作
}
```

⑤具体工厂的实现

```
/* *
 * 具体的工厂实现对象,实现创建具体产品对象的操作
 */
public class ConcreteFactory1 implements AbstractFactory{

    @Override
    public AbstractProductA createProductA() {
        return new ProductA1();
    }

    @Override
    public AbstractProductB createProductB() {
        return new ProductB1();
    }
}
/* *
 * 具体的工厂实现对象,实现创建具体产品对象的操作
 */
public class ConcreteFactory2 implements AbstractFactory{
    @Override
    public AbstractProductA createProductA() {
        return new ProductA2();
    }

    @Override
    public AbstractProductB createProductB() {
        return new ProductB2();
    }
}
```

http://www.neubooks.com

⑥客户端 Client 调用的实现

```
/**
 *客户端,调用具体的产品完成业务操作
 */
public class Client {
    public static void main(String[] args){
        //创建抽象工厂对象
        AbstractFactory af = new ConcreteFactory1();
        //通过抽象工厂来获取一系列产品对象
        af.createProductA();
        af.createProductB();

    }
}
```

1.1.4　代理模式

代理模式用来给某一个对象提供一个代理对象,其他类可以通过代理对象来访问目标对象。我们可以在目标对象实现原有对象功能的基础上,额外增加其他的功能操作,扩展目标对象的功能。

1. 结构和说明

代理模式的结构如图 1-7 所示。

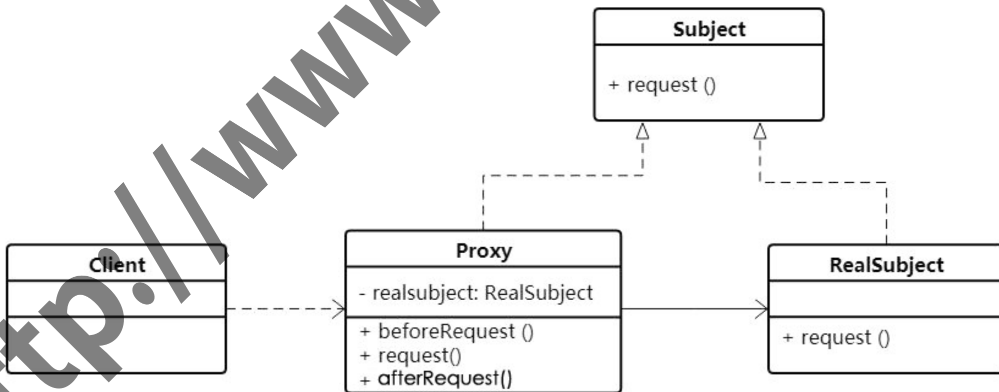


图 1-7　代理模式

● Proxy:代理对象,通常具有以下功能:

(1)实现与具体的目标对象一样的接口,可以使用代理来代理具体的目标对象;

(2)保存一个指向具体目标对象的引用,可以在需要的时候调用具体的目标对象;

(3)可以控制对具体目标对象的访问,并可以负责创建和删除它。

● Subject:目标接口,定义代理和具体目标对象的接口,可以在任何使用具体目标对象的地方使用代理对象。

http://www.neubooks.com

● RealSubject：具体的目标对象，真正实现目标接口要求的功能。

2. 实现示例

在实际项目中，我们可能会遇到要在别人的类上增加功能的情况，通过代理模式我们可以在不修改原有对象功能的情况下扩展目标对象的功能。

【实例 1-6】

【任务要求】

实现代理模式。

【任务实现】

(1)目标 Subjcet 的定义

```
/**
 * 目标接口，定义具体的目标对象和代理方法
 */
public interface Subject {

    //具体的请求方法
    public void request();
}
```

(2)具体目标 RealSubjcet 的定义

```
/**
 * 具体的目标实现，是真正被代理的对象
 */
public class RealSubject implements Subject{

    //实现具体业务方法
    @Override
    public void request(){
        //具体的业务方法
    }
}
```

(3)代理 Proxy 的定义

```
/**
 * 代理对象
 */
public class Proxy implements Subject{

    //被代理的具体目标对象
    private RealSubject realSubject = null;
```

http://www.neubooks.com

```
    /**
     * 构造方法,初始化代理目标对象
     * @param realSubject 被代理的具体目标对象
     */
    public Proxy(RealSubject realSubject){
        this.realSubject = realSubject;
    }

    @Override
    public void request() {
        //在调用具体目标对象的业务方法前,执行额外的功能,例如打印日志等

        //执行具体目标对象的业务方法
        realSubject.request();

        //在调用具体目标对象的业务方法后,执行额外的功能
    }
}
```

1.1.5　观察者模式

观察者模式用来定义对象之间的依赖关系。当一个目标对象的状态发生改变时,所有依赖于它的观察者对象都能够得到通知并且进行相应的更新。

1. 结构和说明

观察者模式的结构如图 1-8 所示。

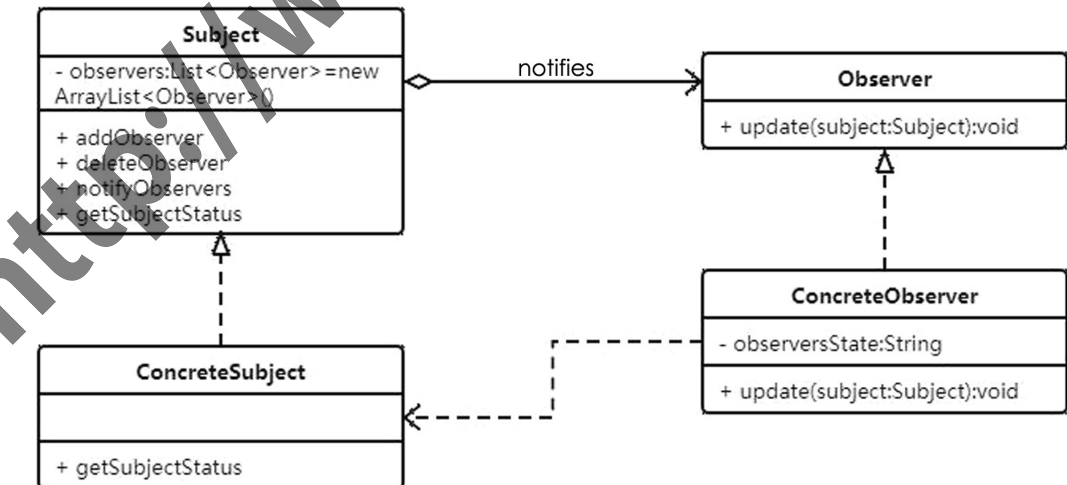


图 1-8　观察者模式

http://www.neubooks.com

● Subject：目标对象，至少需要具有如下功能：

(1)可以被多个观察者注册；

(2)提供对观察者的维护；

(3)当状态发生变化时，通知所有注册的、有效的观察者。

● Observer：定义观察者的接口，提供目标通知时对应的更新方法。我们可以在这个更新方法中进行相应的业务处理，在这个方法里面回调目标对象，以获取目标对象的数据。

● ConcreteSubject：具体的目标实现对象，用来维护目标状态。当目标对象的状态发生改变时，通知所有注册的、有效的观察者，让观察者执行相应的处理。

● ConcreteObserver：观察者的具体实现对象，用来接收目标的通知，并进行相应的后续处理。

2. 实现示例

观察者模式也称作发布-订阅模式，建立了目标与观察者之间的一套触发机制，满足依赖倒置原则。

【实例 1-7】

【任务要求】

实现观察者模式。

【任务实现】

(1)目标 Subjcet 的定义

```
/**
 * 目标对象接口，记录所有的观察者，提供注册和删除观察者的方法
 */
public class Subject {

    /**
     * 用来保存注册的观察者对象
     */
    private List<Observer> observers = new ArrayList<Observer>();

    /**
     * 注册观察者对象
     * @param observer 观察者对象
     */
    public void addobserver(Observer observer){
        observers.add(observer);
    }

    /**
     * 删除观察者对象
     * @param observer 观察者对象
```

http://www.neubooks.com

```
     */
    public void deleteobserver(Observer observer){
        observers.remove(observer);
    }

    /**
     * 发生变更时,通知所有的观察者
     */
    protected void notifyObservers(){
        for(Observer observer:observers){
            observer.update(this);
        }
    }
}
```

(2)具体目标 ConcreteSubject 的定义

```
public class ConcreteSubject extends Subject{
    private String subjectState;

    //获取目标当前状态
    public String getSubjectState(){
        return subjectState;
    }

    public void setSubjectState(String subjectState){
        this.subjectState = subjectState;
        //状态发生改变时,通知所有观察者
        this.notifyObservers();
    }
}
```

(3)观察者 Observer 的定义

```
public interface Observer {

    /**
     * 更新接口
     * @param subject 接收目标对象,用于获取目标对象的状态
     */
    public void update(Subject subject);
}
```

http://www.neubooks.com

(4)具体目标 ConcreteObserver 的定义

```
public class ConcreteObserver implements Observer{
    //观察者的状态
    private String observerState;

    @Override
    public void update(Subject subject) {
        //触发更新时观察者需要进行的业务操作

        //这里可能需要更新观察者的状态,使其与目标的状态保持一致
        observerState = ((ConcreteSubject)subject).getSubjectState();
    }
}
```

虽然通过使用设计模式,我们可以使代码更加灵活、可重用和可维护,但是不能过度实施。任何设计模式都可能是一把双刃剑,如果在错误的地方实施,它可能是灾难性的,并给我们造成许多问题;如果在正确的地方实施,它可以让项目中的复杂问题迎刃而解。

在项目中实现设计模式并不总是强制性的,我们在遇到问题的时候可以选择一个设计模式,也可以不选择设计模式。设计模式只是帮助我们解决常见的问题,我们需要在项目中根据实际需要,找出当前应该使用哪种模式,只有这样我们才能开发出健壮、灵活的代码。

1.2 系统设计

本节知识点

知识点	了解	掌握	熟练	精通
体系结构定义	√			
体系结构风格	√			
体系结构设计方法	√			
设计原则				√
MVC 工作流程				√
MVC 实现				√
MVC 与三层架构的区别		√		

软件系统的构建是为了满足组织的业务目标。架构是业务(通常是抽象的)目标和最终(具体的)结果系统之间的桥梁。虽然从抽象目标到连接系统的路径可能是复杂的,但是软件体系结构可以使用已知的技术来设计、分析、记录和实现,即通过这些技术来实现抽象的业务和具体任务目标。经过体系结构设计,我们将软件系统中的复杂性简化,使之易于处理。

http://www.neubooks.com

在本章中，我们将从软件工程师的角度来看待体系结构设计，探索软件体系结构设计的内容和方法。

1.2.1　体系结构设计

1. 体系结构定义

(1)体系结构是一组软件结构

体系结构可以而且确实包含了多个结构，单独拿出任何一个结构都不能称作该结构。这些结构与系统软件、应用软件、程序设计语言紧密结合，解决软件开发中的复杂问题。在软件开发中，体系结构位于软件需求和软件设计之间，它的定义不仅满足业务需求、设计和技术的要求，同时还考虑了像质量、安全、性能等方面的属性。

虽然软件包含无穷无尽的结构，但并不是所有的结构都是体系结构。如果一个结构支持系统和系统特性的实现，那它就是体系结构。系统和系统特性的实现包括系统所实现的功能、系统面对故障时的可用性、对系统进行具体更改的困难、系统对用户请求的响应能力以及许多其他方面。

(2)体系结构是一种抽象

体系结构定义了软件元素，体现了软件元素如何相互关联的信息。因此，体系结构首先是一个系统的抽象，它忽略那些不影响元素使用、被使用、关联或与其他元素交互的细节。系统将元素的细节划分为公共部分和私有部分，体系结构只与元素的公共部分有关，而元素的私有部分只与元素的内部实现有关，这部分不是体系结构。目前所有的现代系统中，各元素都通过接口相互交互。但是，除了接口之外，体系结构抽象还让我们从系统的元素、它们是如何排列的、它们是如何交互的、它们是如何组成的、它们支持系统实现的属性是什么等方面来看待系统。

(3)体系结构是一定存在的

每个系统都可以被分解成众多元素和它们之间的关系，证明每个系统都存在体系结构。在最极端的情况下，如果系统本身就是一个单一的元素，这样的系统还有体系结构存在吗？虽然这种系统实现不是一个很好的实现方式，但其仍然也是一个体系结构。

虽然每个系统都有一个体系结构，但并不意味着任何人都知道这个体系结构。可能是因为所有设计该系统的人都已不维护系统了，也可能是系统相关的文档已经丢失，亦或者源代码也丢失了，只剩下一堆正在执行的二进制代码，造成我们无从确定该系统的体系结构，但其体系结构仍然存在，只是不为我们所知。这揭示了系统的体系结构和该体系结构的实现之间的区别，说明了体系结构文档的重要性。

(4)体系结构包含元素行为

系统中的每个元素的行为都是体系结构的一部分，只要该行为可以用于实现系统的属性。元素的行为体现了元素如何相互作用，这也就是我们对体系结构定义的一部分。这并不意味着每个元素的确切行为和性能必须在所有情况下都被记录下来，如果行为的某些方面是细粒度的，低于我们的关注点，则不会被记录。但是，如果一个元素的行为影响另一个元素或影响整个系统的可接受性，则必须考虑这种行为，并应将其记录在文档中，作为软件体系结构的一部分。

http://www.neubooks.com

2. 体系结构风格

体系结构风格的形成是多年探索研究和工程实践的结果，一种体系结构风格以结构组织模式定义了一个系统家族。一个良好和通用的体系结构风格往往是工程技术领域成熟的标志。经过工程师多年的研究和发展，已经总结出许多成熟的软件体系结构风格。

（1）管道-过滤器风格

在管道-过滤器风格的软件体系结构中，主要有过滤器和管道两种元素，每个构件都有一组输入和输出，构件读取输入的数据流，经过内部处理，然后产生输出数据流，数据流在两个过滤器之间通过管道进行交互。

管道-过滤器风格的优点：

①易于扩展。软构件具有高内聚、低耦合的特点。

②简化系统实现。允许设计者将整个系统的输入、输出行为看成是多个过滤器的行为的简单合成。

③支持软件复用。只要提供适合在两个过滤器之间传送的数据，任何两个过滤器都可被连接起来。

④便于维护。新的过滤器可以添加到现有系统中，旧的过滤器可以被改进的过滤器替换掉。

⑤具有并发性。每个过滤器作为一个单独的任务完成，因此可与其他任务并行执行。

当然，这样的系统也存在相应的缺点：

①导致系统处理过程的成批操作。这是因为虽然过滤器可增量式地处理数据，但它们是独立的，所以设计者必须在每个过滤器中对输入、输出管道中的数据流进行解析或反解析，增加了过滤器具体实现的复杂性。

②不适合处理交互任务。该模型适于数据流的处理和变换，不适合为与用户交互频繁的系统建模。当用户要操作某一项数据时，要涉及多个过滤器对相应数据的操作，其实现较为复杂。

（2）事件驱动风格

在事件驱动的体系中，系统由若干个构件组成。这些构件不直接调用一个过程，而是声明或广播一个或多个事件，系统中其他构件中的过程在这些事件中进行注册。当一个事件被触发后，系统自动调用在这个事件中注册的所有构件过程，这样，广播一个事件就会触发另一个构件中的过程的调用，但是事件的广播者并不知道也不关心哪些构件会被这些事件影响。

事件驱动风格的优点：

①简化客户代码。

②提高了软件复用能力。在属于同一族的任何系统中，系统的高级管理子系统的描述是完全类似的，便于重用。

③良好的可扩展性。事件广播构件不需要知道哪些构件会响应事件，所以设计者只需为某个构件注册一个事件处理接口就可以将该构件引入整个系统，同时并不影响其他的系统构件。

事件驱动风格的缺点：

①构件削弱了自身对系统计算的控制能力，完全由系统来决定。

http://www.neubooks.com

②数据共享和传输困难，因为是通过广播事件来驱动，所以两个构件之间没有直接的交互方式，很难将一个构件的数据共享和传递给下一个构件。

(3) 面向对象风格

在面向对象体系结构中，数据抽象、抽象数据类型、类继承集成一体，使软件工程公认的模块化、信息隐藏、抽象、重用性等原则得到了充分实现。在这种体系结构中，数据表示和相关原语操作都被封装在抽象数据类型中，对象是构件，也称为抽象数据类型的实例，而对象与对象之间通过函数调用和过程调用来进行交互。

面向对象风格的优点：

①高度模块化。数据与其相关操作被组织为对象，成为模块组织的基本单位。

②实现封装。一组功能和其实现细节被封装在一个对象中，只将对外的功能通过接口暴露出来。

③代码共享。继承和封装方法为对象复用提供支持。

④易维护性。对象非常接近于我们对问题和解决方案模型的思维方式，易于理解和修改。

面向对象风格的缺点：

①显式调用。如果一个对象要调用另一个对象，则必须知道它的标识和名称；而一个对象的标识改变了，就必须修改所有其他明确调用它的对象。

②连锁反应。如果 A 使用了对象 B，C 也使用了对象 B，那么 C 对 B 的使用所造成的修改对 A 的影响可能无法预料，所以必须修改所有显式调用它的其他对象，并消除由此带来的一些副作用。

(4) 分层风格

在分层系统中，采用层次化的组织方式进行系统构建，每一层都具有高度的内聚性，包含抽象程度一致的各种构件，支持信息隐藏。

系统中有两个角色，而每一层都要承担这两个角色。一方面，它要为结构中的上层提供服务；另一方面，它还要作为结构中下面层次的客户，调用下层提供的功能函数。在分层体系结构中，每一层只对相邻层可见。

分层体系结构风格的优点：

①复杂系统简单化。支持基于抽象程度递增的系统设计，使设计者可以把一个复杂系统按递增的步骤进行分解。

②易于扩展。因为每一层至多和相邻的上下层交互，所以功能的改变最多影响相邻的上下层。

③便于复用。只要提供的服务接口定义不变，同一层的不同实现可以交换使用。这样，就可以定义一组标准的接口，而允许各种不同的实现方法。

分层体系结构风格的缺点：

①并非所有系统都能够按照层次来进行划分，甚至即使一个系统的逻辑结构是层次化的，但是出于对系统性能的综合考虑，我们也不得不把一些低级或高级的功能综合起来。

②很难找到一个合适的、正确的层次划分方法。

③系统比较难以调试，数据需要经过多层次。

http://www.neubooks.com

（5）客户端/服务器风格

客户端/服务器体系结构又叫 C/S 体系结构，有三个主要组成部分：数据库服务器、客户端应用程序和网络。三部分在系统中分别担任不同的角色，实现不同的功能，相互协作，完成系统需求。

①数据库服务器

a. 数据库安全性的要求；

b. 数据库访问并发性的控制；

c. 数据库前端客户应用程序的全局数据完整性规则；

d. 数据库的备份与恢复。

②客户端应用程序

a. 提供用户与数据库交互的界面；

b. 向数据库服务器提交用户请求并接受来自数据库服务器的信息；

c. 利用客户应用程序对存在客户端的数据执行应用逻辑要求。

③网络

完成数据库服务器和客户端应用程序之间的数据传输。

客户端/服务器体系结构的优点：

客户端应用程序和服务器构件分别运行在不同的计算机上，系统中每台服务器都可以符合各构件的要求，这样的系统有极大的适应性和灵活性，而且易于对系统进行扩展。

客户端/服务器体系结构的缺点：

①开发成本较高

客户端/服务器体系结构对客户端软硬件配置的要求较高，尤其是软件的不断升级，对硬件的要求不断提高，增加了整个系统的成本，且客户端变得越来越臃肿。

②客户端程序设计复杂

采用客户端/服务器体系结构进行软件开发，大部分工作量放在客户端的程序设计上，客户端显得十分庞大。

③信息内容和形式单一

传统应用一般为事务处理，界面基本遵循数据库的字段解释，开发之初就已确定，而且不能随时截取办公信息和档案等外部信息，用户获得的只是单纯的字符和数字，既枯燥又死板。

④软件移植困难

采用不同开发工具或平台开发的软件，一般互不兼容，不能或很难移植到其他平台上运行。

⑤软件维护和升级困难

C/S 体系结构的软件升级非常困难，对软件的一个小小改动需要每个客户机上的软件都进行升级维护。

（6）浏览器/服务器风格

浏览器/服务器风格就是三层 C/S 结构的一种实现方式，又叫 B/S 结构。Web 浏览器是客户端最主要的应用软件。这种风格统一了客户端，将系统功能实现的核心部分集中到

http://www.neubooks.com

服务器上，简化了系统的开发、维护和使用。客户机上只需要安装一个浏览器，如 Netscape Navigator 或 Internet Explorer，服务器安装 SQL Server、Oracle、MySQL 等数据库。客户机上的浏览器通过 Web Server 同数据库进行数据交互。

浏览器/服务器体系结构的优点：

维护和升级方式简单。系统管理员只需要管理服务器就行了，所有的客户端只是浏览器，根本不需要做任何维护。无论用户的规模有多大，都不会增加任何维护升级的工作量，所有的操作只需要针对服务器进行。

浏览器/服务器体系结构的缺点：

①安全性差。由于在互联网上进行数据传输，容易泄露数据。

②响应速度慢。相比较客户端/服务器系统，浏览器/服务器系统在数据查询等方面的响应速度比较慢。

（7）三层架构风格

传统的系统架构在系统中的各个组件之间合并了相对脆弱的耦合，使用数据结构（数据库记录、平面文件）以紧密耦合的方式构建事务、数据库、报表等。

传统架构的整体系统对变化很敏感，往往一个子系统的输出变化通常会导致整个系统崩溃，而切换到子系统的新实现方式通常还会导致旧的、静态绑定的系统一起变更。随着规模、需求、数量和业务变化率的增加，这种脆弱性会很快显现出来。这些方面中的任何一个方面的任何重大变化都将导致系统的脆弱性成为危机：网站不可用或响应不及时。由于耦合，IT 组织将无法应对变化；Web 的动态性又使这些脆弱的体系结构的管理变得不合理。

在 Web 应用实现中，经过设计人员多年的实践经验，遵循高内聚、低耦合的设计原则，分层结构在 Web 应用的架构设计中被广泛使用。

图 1-9 是一个简单的三层结构，在此基础上衍生出了很多其他分层结构，但是目的都是为了高内聚、低耦合，让整个系统对业务变化响应更及时、高效。

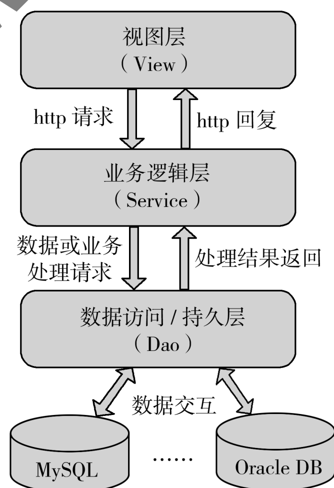


图 1-9　分层结构图

①视图层

视图层负责页面的显示和业务模块流程的控制；与用户交互，采集接收用户提交请求的

http://www.neubooks.com

参数，并调用业务逻辑层对数据进行相应的业务处理。

②业务逻辑层

业务逻辑层负责处理客户的业务逻辑，接收视图层传输的数据进行业务处理，处理后数据调用到已定义的数据层的接口。封装应用程序的业务逻辑有利于通用业务逻辑的独立性和重复利用性，程序显得非常简洁、易扩展。

③数据访问/持久层

数据访问/持久层负责数据持久层的工作，对非原始数据（数据库或者文本文件等存放数据的形式）的操作都封装在这一层，也就是说，数据访问层是对数据库的操作，而不是数据，具体为业务逻辑层或表示层提供数据服务。封装数据访问/持久层可以让上层业务逻辑在实现时不用关心数据存储的方式；如果后期应用程序从 MySQL 数据库转成了 Oracle 数据库，仅需要变更数据访问/持久层的实现即可，屏蔽对上层业务逻辑的影响。

三层并不一定是绝对的，在设计过程中，我们需要根据不同项目的复杂度来确定是否需要分层以及需要分几层。总而言之，分层是为了更好地解耦，提高代码复用性，降低需求变更的影响范围，更易于系统扩展。

3. 体系结构设计方法

(1)工件驱动的方法

工件驱动的体系结构设计方法从方法的工件描述中提取体系结构描述。工件驱动的体系结构设计方法的例子包括广为流行的面向对象分析和设计方法 OMT 和 OAD。

(2)用例驱动的方法

用例驱动的体系结构设计方法主要从用例导出体系结构抽象。一个用例，是指系统进行的一个活动系列，它为参与者提供一些结果值。参与者通过用例使用系统，参与者和用例共同构成了用例模型。用例模型的目的是作为系统预期功能及其环境的模型，并在客户和开发者之间起到合约的作用。

(3)模式驱动的方法

软件设计模式的目的在于编制一套可重用的基本原则，用于开发高质量的软件系统。体系结构设计模式是体系结构层次的一种抽象表示。

(4)领域驱动的方法

在领域驱动的体系结构设计方法中，体系结构抽象是从领域模型导出的。可以把领域驱动的软件体系结构(DSSA)看成是多系统范围内的体系结构，即它是从一组系统中导出的，而不是某一单独的系统。

(5)需求驱动的方法

需求驱动是指软件体系结构设计基于需求分析的结果，因此需求驱动下软件体系结构设计主要是描述所解决问题和解决方案之间的动态关系。需求分析首先在于问题的描述，可并行建立目标模型和场景模型。体系结构设计是根据组件或子系统之间的数据、控制及其他依赖关系描述的系统全局结构设计，该结构描述系统如何分解为组件，并且各组件如何相互交互。通过描述组件和连接的抽象属性，得到软件体系结构的抽象模型，再从需求分析模型确定的解决空间中发现体系结构设计方案，从而得到具体的体系结构实例。

http://www.neubooks.com

1.2.2　Java Web 应用程序设计

Java 语言已经越来越流行，常年在 TIOBE 编程语言排行榜上占据榜首。现在越来越多的企业和研发人员都在使用 Java 语言进行系统开发。

随着互联网的高速发展，传统 C/S 结构的应用已经不能满足企业业务发展的需要，Web 应用被广泛使用，越来越多的企业都投入 Web 应用的研发中。Java 语言因为其特性，与 Web 应用程序相结合，成为 Web 应用程序开发的主要语言，占据了 Web 应用开发的重要位置。

1. 设计原则

要进行 Java Web 应用程序的设计，首先要了解六大设计原则，只有在设计中时刻想着这六大原则，我们才能设计出结构良好的系统。

(1)单一职责原则(Single Responsibility Principle)

单一职责原则，简称 SRP，其定义是：应该有且仅有一个类引起类的变更。单一职责告诉我们，在设计时一个类只担负一个职责，如果出现了一个类有多个职责需要进行拆分。

遵守单一职责原则可以给设计带来的好处：

- 复杂性降低：每一个类都是单一职责，具体是做什么的都有清晰明确的定义；
- 可读性高：简化了类的复杂性，带给研发人员最直接的好处就是可读性高；
- 变更引起的风险降低：如果一个类只有单一职责，那么修改时的影响范围会最小，避免过多地影响其他功能模块。

(2)开闭原则(Open Closed Principle)

开闭原则，简称 OCP，其定义是：一个软件实体，例如类、模块和函数应该对扩展开放，对修改关闭。这是 Java 最基础的设计原则，要求一个软件实体应该通过扩展来实现变化，而不是通过修改已有的代码实现变化。

在系统的运行过程中，经常有新的需求提出，当我们修改的时候应该做到能不动原来的代码就不动，通过扩展的方式来满足需求，将对原系统的影响降到最低。

在 Java 中，遵循开闭原则的最好手段就是抽象，在设计之初就要考虑到未来所有可能发生变化的因素，将职责设计成接口方式，如果将来实现方式发生了变更，只需要更换相应的实现类即可。

(3)里氏替换原则(Liskov Substitution Principle)

里氏替换原则，简称 LSP，其定义是：如果对每一个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有对象 o1 都替换成 o2 的时候，程序 P 的行为都没有发生变化，那么类型 T2 是类型 T1 的子类型。简单来说就是：只要父类能出现的地方子类就可以出现，而且替换为子类也不会产生任何异常，但是反过来是不行的，因为子类可能扩展了父类的功能。

里氏替换原则为我们在设计时提供了一个继承的规范：

- 子类可以实现父类的抽象方法，但是不能覆盖父类的非抽象方法；
- 子类可以有自己的属性和方法；
- 子类覆盖或重载父类的方法时输入参数可以被放大。

http://www.neubooks.com