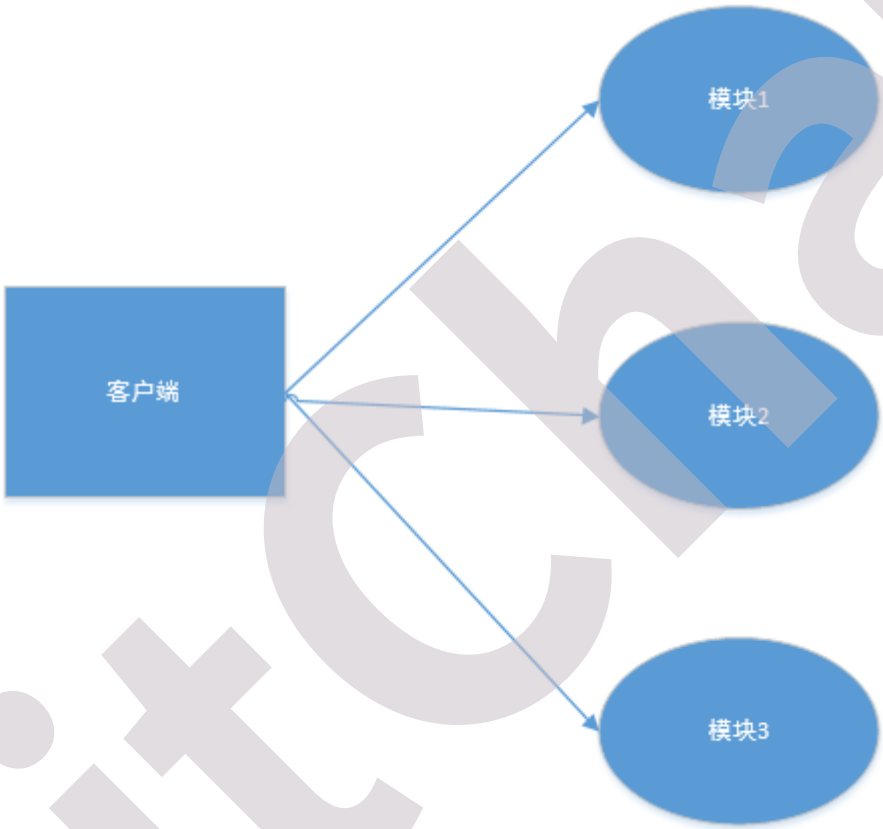
概述

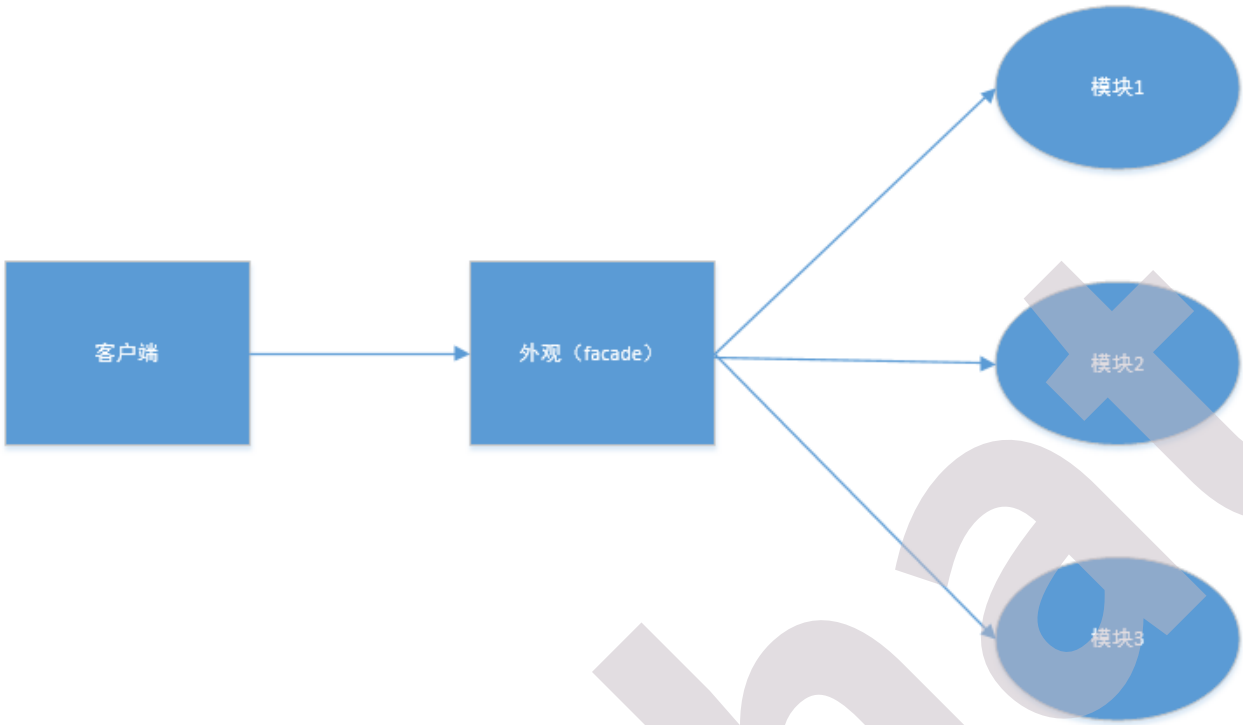
> 外观模式提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

飞机驾驶舱不少人都见过，当看到那些密密麻麻的按钮时，心想要是能一键启动就好了。在代码的世界里，我们也常常遇到一个业务功能需要调用很多接口甚至很多系统的情况，就像下图：



有时候被我们调用模块之间还需要互相调用，模块之间的关系都可以画出一张蜘蛛网。在这种情况下，要求开发者需要对每一个模块都有一定的了解，还需要了解他们之间的关系，开发一个功能的成本简直太高了，令人崩溃。

飞机驾驶舱的按钮由于某些原因不可能做成一键启动，但是我们的代码可以：



外观就是这个一键启动的按钮，它将多个模块或系统的代码进行了整合，而我们只要简单地调用外观暴露出来的一个接口。

这就是外观模式（也叫门面模式），其作用显而易见，就是提供一个简单接口来调用后方一群复杂的接口。

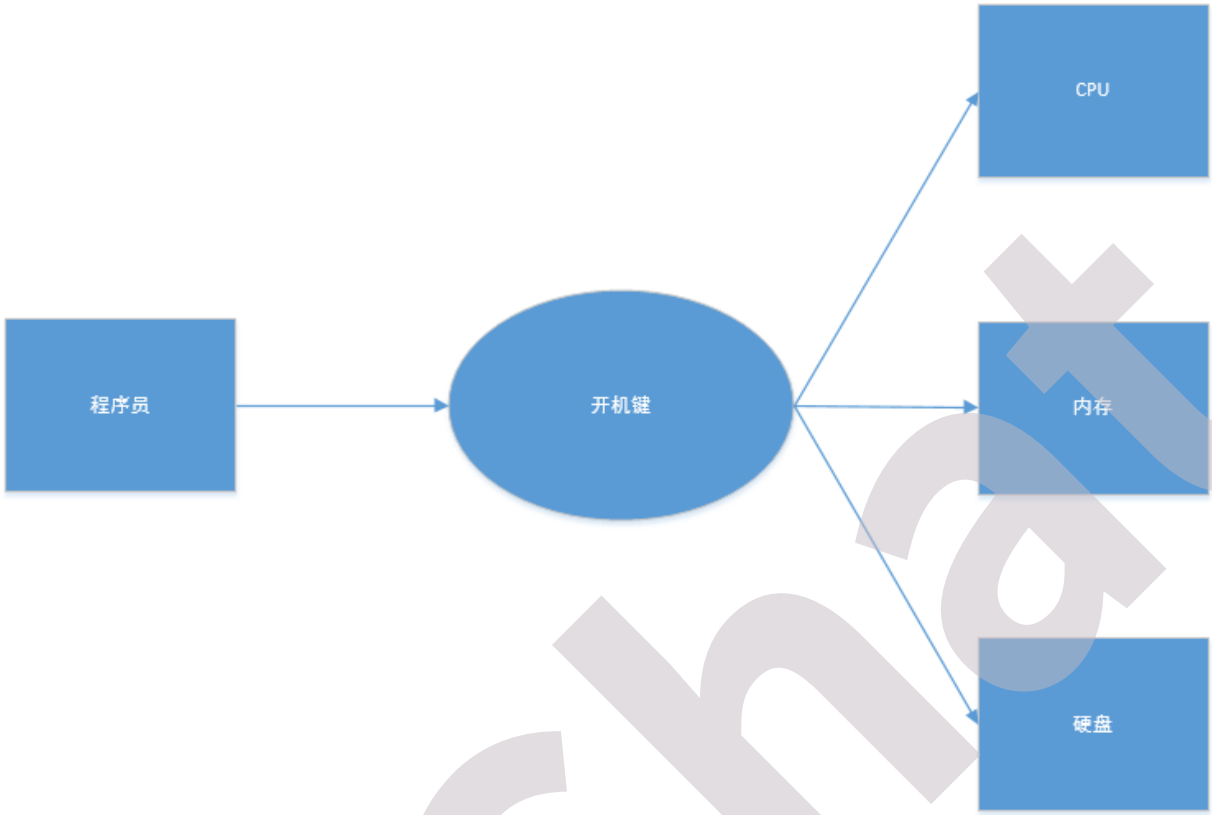
在外观模式中主要有三个角色：

1. 子系统：已有模块或子系统，提供了一系列复杂的接口或功能
2. **外观（门面）**：它了解子系统，并对外暴露一个简单的接口
3. 客户：调用外观提供的接口来实现功能，无需了解复杂的子系统

代码示例

下面我们写一个简单的电脑启动的例子。（这个例子在许多设计模式教程中都曾出现，笔者认为这是最好的例子之一，直接借用了）

启动电脑我们通常只需要按下开机键就可以了，但电脑内部实际上启动了多个模块，如CPU，硬盘，内存等



开机键就是一个很好的外观，让程序员们无需了解CPU，内存和硬盘如何启动。

```
//CPU
public class CPU {
    public void start(){
        System.out.println("启动CPU");
    }
}
//硬盘
public class Disk {
    public void start(){
        System.out.println("启动硬盘");
    }
}
//内存
public class Memory {
    public void start(){
        System.out.println("启动内存");
    }
}
```

如果没有开机键，我们需要这么做：

```
new CPU().start();
new Disk().start();
new Memory().start();
```

有了开机键，这些操作都交给开机键去做：

```
//开机键
public class StartBtn {

    public void start(){
        new CPU().start();
        new Disk().start();
        new Memory().start();
    }
}
```

而我们只需要：

```
new StartBtn().start();
```

外观模式不仅为我们提供了一个简单方便的接口，也让我们的系统和子系统解耦。

迪米特法则（最少知道原则）

迪米特法则是说每一个类都应该尽量的少知道别的类，外观模式就是迪米特法则的应用。原本我们需要知道许多的子系统或接口，用了外观类之后，我们仅仅需要知道外观类即可。

换句话说就是：知道的太多对你没好处。

迪米特法则是希望类之间减少耦合，类越独立越好。有句话叫牵一发而动全身，如果类之间关系太紧密，与之关联的类太多，一旦你修改该类，也许会动到无数与之关联的类。

实际案例

JAVA三层结构

用JAVA开发我们经常使用三层结构：

- controller 控制器层
- service 服务层
- dao 数据访问层

有时候业务很简单，例如根据用户ID或者用户信息，service层这样写：

```
User getUserById(Integer id){
    return userDao.getUserById(id);
}
```

dao：

```
User getUserById(Integer id){
    //查询数据库
    return user;
}
```

service层直接调用了`userDao`的`getUserById()`，service本身并没有执行什么额外的代码，那么为什么不省去service层呢？

其实三层结构也蕴含了外观模式的思想在内。假如service有一个转账方法：

```
public boolean transMoney(Integer user1,Integer user2,Float money){
    //用户1加钱
    userDao.addMoney(user1,money);
    //用户2扣钱
    userDao.decMoney(user2,money);
    //转账日志
    logDao.addLog(user1,user2,money);
}
```

作为调用方来说，并不想知道转账操作具体要调用哪些Dao，一行代码`transMoney()`就能搞定岂不是皆大欢喜。

因此service是很有必要的，一般在业务系统中，service层的类不仅仅是简单的调用dao，而是作为外观，给controller提供了更方便好用的接口。

不过无论多复杂的系统，总会有service直接调用dao的`getUserById()`的情况，我们是否可以偷懒直接在Controller调用Dao呢？

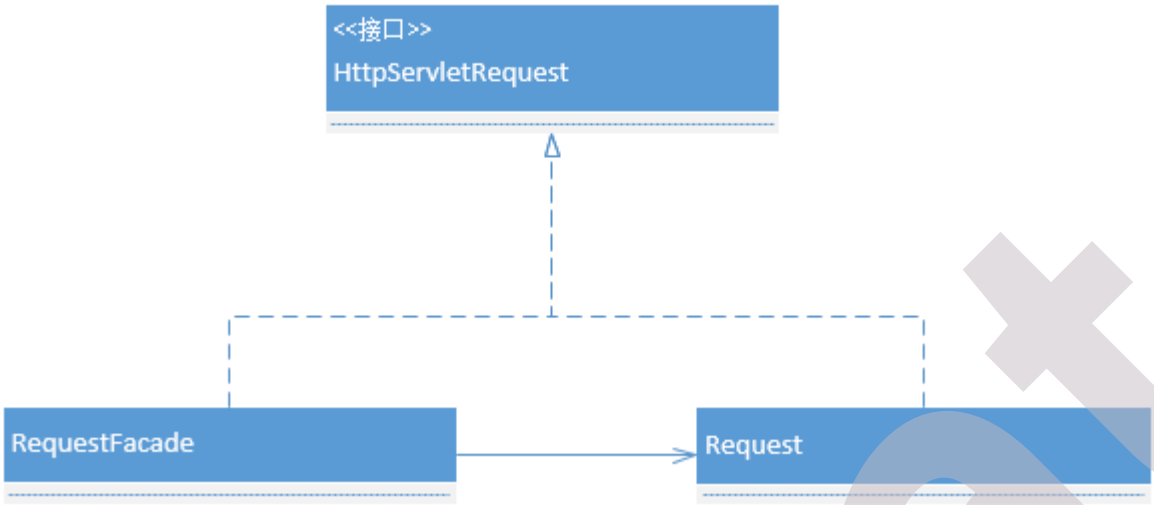
理论上是没问题的，但是强烈建议不要这么干，因为这样会导致层侵入，三层结构的层级混乱。

除非你的业务真的简单到极致，那么干脆直接舍弃service层。只要你有service层，就请不要跨层调用。

Tomcat中的外观模式

在做Servlet开发时，我们经常用的两个对象就是`HttpServletRequest`和`HttpServletResponse`，但我们拿到的这两个对象其实是被Tomcat经过了外观包裹的对象，那么Tomcat为什么要这么做呢？

首先我们先来了解一下`HttpServletRequest`，通过源码可以发现，`HttpServletRequest`是一个接口，有两个类`RequestFacade`和`Request`实现了`HttpServletRequest`：



以`Facade`命名的，毫无疑问是用了外观模式，下面给出一部分源码：

Request类，继承 HttpServletRequest ：

```
public class Request implements HttpServletRequest {

}
```

当我们需要使用Request对象时，Tomcat传给我们的其实并不是Request对象，而是RequestFacade

RequestFacade 类：

```
package org.apache.catalina.connector;

public class RequestFacade implements HttpServletRequest {

    protected Request request = null;

    public RequestFacade(Request request) {
        this.request = request;
    }

    public Object getAttribute(String name) {
        if (this.request == null) {
            throw new IllegalStateException(sm.getString("requestFacade.nullRequest"));
        } else {
            return this.request.getAttribute(name);
        }
    }


    public String getProtocol() {
        if (this.request == null) {
            throw new IllegalStateException(sm.getString("requestFacade.nullRequest"));

        } else {
```

```
            return this.request.getProtocol();
        }
    }

}
```

从上面 `RequestFacade` 源码中可以看到，当调用 `getAttribute()` ， `getProtocol()` 等方法时，其实还是调用了 `Request` 对象的 `getAttribute()` 方法。

既然如此，为什么要多次一举弄个 `RequestFacade` 呢 ，其实是为了安全，Tomcat不想把过多的方法暴露给别人。

Tomcat内部有很多组件，组件之间经常需要通讯，有些方法不得不定义为Public，这样才能被其他组件所调用。

但是有些方法只希望内部通讯用，并不想暴露给Web开发者，否则会有安全问题。所以定义一个外观类，只实现想要暴露给外部的方法。

所以Tomcat要传Request给我们的时候，其实是这么做的：

```
return new RequestFacade(request);
```

从这个案例中可以看出外观模式不仅仅用于将复杂的接口包装为一个简单的接口，也可以用于隐藏一些不想暴露给别人的方法或接口。

总结

外观模式主要使用场景：

- 包装多个复杂的子系统，提供一个简单的接口
- 重新包装系统，隐藏不想暴露的接口

优点

将复杂的接口简单化，减少了客户端与接口之间的耦合，提高了安全性。

缺点

可能产生大量的中间类（外观类），一定程度上增加了系统的复杂度。