

## 49 | 桥接模式：如何实现支持不同类型和渠道的消息推送系统？

王争 · 设计模式之美



上一节课我们学习了第一种结构型模式：代理模式。它在不改变原始类（或者叫被代理类）代码的情况下，通过引入代理类来给原始类附加功能。代理模式在平时的开发经常被用到，常用在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。

今天，我们再学习另外一种结构型模式：桥接模式。桥接模式的代码实现非常简单，但是理解起来稍微有点难度，并且应用场景也比较局限，所以，相对于代理模式来说，桥接模式在实际的项目中并没有那么常用，你只需要简单了解，见到能认识就可以，并不是我们学习的重点。

话不多说，让我们正式开始今天的学习吧！

### 桥接模式的原理解析

**桥接模式**，也叫作**桥梁模式**，英文是 **Bridge Design Pattern**。这个模式可以说是 23 种设计模式中最难理解的模式之一了。我查阅了比较多的书籍和资料之后发现，对于这个模式有两种不同的理解方式。

当然，这其中“最纯正”的理解方式，当属 GoF 的《设计模式》一书中对桥接模式的定义。毕竟，这 23 种经典的设计模式，最初就是由这本书总结出来的。在 GoF 的《设计模式》一书中，桥接模式是这么定义的：“Decouple an abstraction from its implementation so that the two can vary independently。”翻译成中文就是：“将抽象和实现解耦，让它们可以独立变化。”

关于桥接模式，很多书籍、资料中，还有另外一种理解方式：“一个类存在两个（或多个）独立变化的维度，我们通过组合的方式，让这两个（或多个）维度可以独立进行扩展。”通过组合关系来替代继承关系，避免继承层次的指数级爆炸。这种理解方式非常类似于，我们之前讲过的“组合优于继承”设计原则，所以，这里我就不多解释了。我们重点看下 GoF 的理解方式。

GoF 给出的定义非常的简短，单凭这一句话，估计没几个人能看懂是什么意思。所以，我们通过 JDBC 驱动的例子来解释一下。JDBC 驱动是桥接模式的经典应用。我们先来看一下，如何利用 JDBC 驱动来查询数据库。具体的代码如下所示：

 复制代码

```
1 Class.forName("com.mysql.jdbc.Driver");//加载及注册JDBC驱动程序
2 String url = "jdbc:mysql://localhost:3306/sample_db?user=root&password=your_passw
3 Connection con = DriverManager.getConnection(url);
4 Statement stmt = con.createStatement();
5 String query = "select * from test";
6 ResultSet rs=stmt.executeQuery(query);
7 while(rs.next()) {
8     rs.getString(1);
9     rs.getInt(2);
10 }
```

如果我们想要把 MySQL 数据库换成 Oracle 数据库，只需要把第一行代码中的 `com.mysql.jdbc.Driver` 换成 `oracle.jdbc.driver.OracleDriver` 就可以了。当然，也有更灵活的实现方式，我们可以把需要加载的 Driver 类写到配置文件中，当程序启动的时候，自动从配置文件中加载，这样在切换数据库的时候，我们都不需要修改代码，只需要修改配置文件就可以了。

不管是改代码还是改配置，在项目中，从一个数据库切换到另一种数据库，都只需要改动很少的代码，或者完全不需要改动代码，那如此优雅的数据库切换是如何实现的呢？

源码之下无秘密。要弄清楚这个问题，我们先从 `com.mysql.jdbc.Driver` 这个类的代码看起。我摘抄了部分相关代码，放到了这里，你可以看一下。

 复制代码

```
1 package com.mysql.jdbc;
2 import java.sql.SQLException;
3
4 public class Driver extends NonRegisteringDriver implements java.sql.Driver {
5     static {
6         try {
7             java.sql.DriverManager.registerDriver(new Driver());
8         } catch (SQLException E) {
9             throw new RuntimeException("Can't register driver!");
10        }
11    }
12
13    /**
14     * Construct a new driver and register it with DriverManager
15     * @throws SQLException if a database error occurs.
16     */
17    public Driver() throws SQLException {
18        // Required for Class.forName().newInstance()
19    }
20 }
```

结合 `com.mysql.jdbc.Driver` 的代码实现，我们可以发现，当执行 `Class.forName("com.mysql.jdbc.Driver")` 这条语句的时候，实际上是做了两件事情。第一件事情是要求 JVM 查找并加载指定的 `Driver` 类，第二件事情是执行该类的静态代码，也就是将 MySQL Driver 注册到 `DriverManager` 类中。

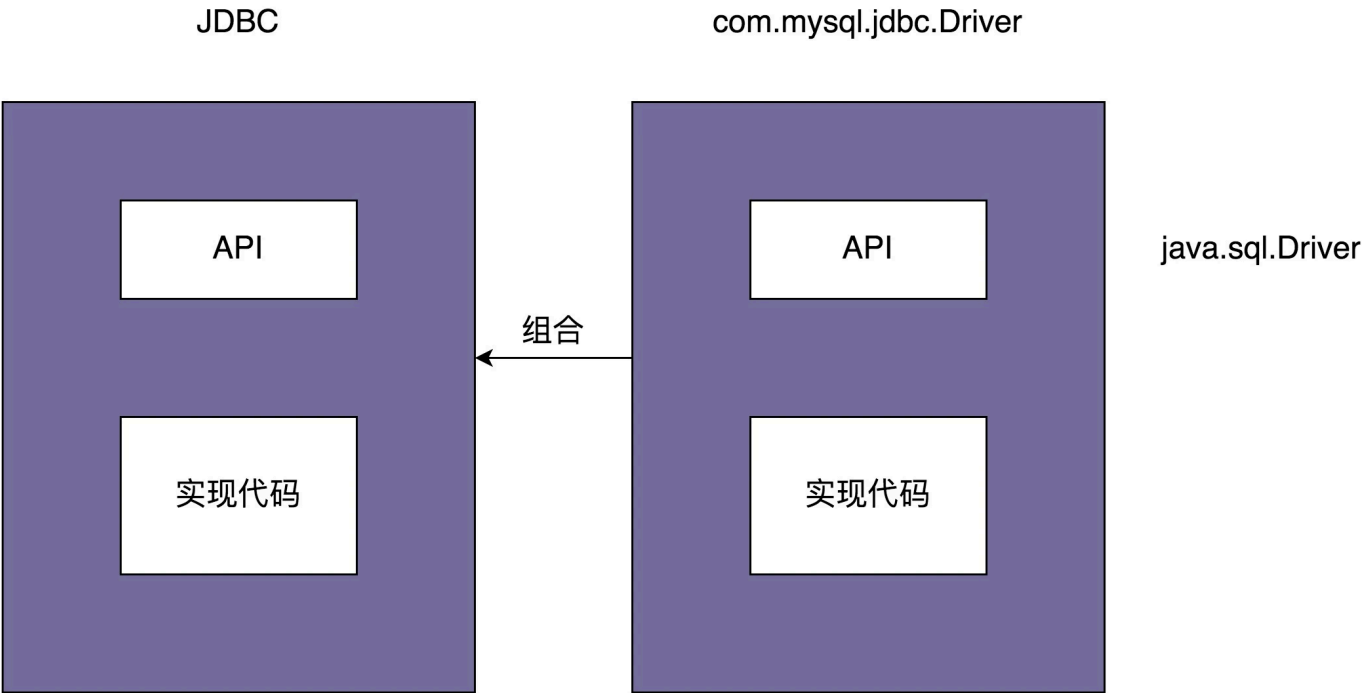
现在，我们再来看一下，`DriverManager` 类是干什么用的。具体的代码如下所示。当我们把具体的 `Driver` 实现类（比如，`com.mysql.jdbc.Driver`）注册到 `DriverManager` 之后，后续所有对 JDBC 接口的调用，都会委派到对具体的 `Driver` 实现类来执行。而 `Driver` 实现类都实现了相同的接口（`java.sql.Driver`），这也是可以灵活切换 `Driver` 的原因。

```
1 public class DriverManager {
2     private final static CopyOnWriteArrayList<DriverInfo> registeredDrivers = new C
3
4     //...
5     static {
6         loadInitialDrivers();
7         println("JDBC DriverManager initialized");
8     }
9     //...
10
11     public static synchronized void registerDriver(java.sql.Driver driver) throws S
12         if (driver != null) {
13             registeredDrivers.addIfAbsent(new DriverInfo(driver));
14         } else {
15             throw new NullPointerException();
16         }
17     }
18
19     public static Connection getConnection(String url, String user, String password
20         java.util.Properties info = new java.util.Properties();
21         if (user != null) {
22             info.put("user", user);
23         }
24         if (password != null) {
25             info.put("password", password);
26         }
27         return (getConnection(url, info, Reflection.getCallerClass()));
28     }
29     //...
30 }
```

桥接模式的定义是“将抽象和实现解耦，让它们可以独立变化”。那弄懂定义中“抽象”和“实现”两个概念，就是理解桥接模式的关键。那在 JDBC 这个例子中，什么是“抽象”？什么是“实现”呢？

实际上，JDBC 本身就相当于“抽象”。注意，这里所说的“抽象”，指的并非“抽象类”或“接口”，而是跟具体的数据库无关的、被抽象出来的一套“类库”。具体的 Driver（比如，com.mysql.jdbc.Driver）就相当于“实现”。注意，这里所说的“实现”，也并非指“接口的实现类”，而是跟具体数据库相关的一套“类库”。JDBC 和 Driver 独立开发，通过对象之间的组合关系，组装在一起。JDBC 的所有逻辑操作，最终都委托给 Driver 来执行。

我画了一张图帮助你理解，你可以结合着我刚才的讲解一块看。



## 桥接模式的应用举例

在 [第 16 节](#) 中，我们讲过一个 API 接口监控告警的例子：根据不同的告警规则，触发不同类型的告警。告警支持多种通知渠道，包括：邮件、短信、微信、自动语音电话。通知的紧急程度有多种类型，包括：SEVERE（严重）、URGENCY（紧急）、NORMAL（普通）、TRIVIAL（无关紧要）。不同的紧急程度对应不同的通知渠道。比如，SERVE（严重）级别的消息会通过“自动语音电话”告知相关人员。

在当时的代码实现中，关于发送告警信息那部分代码，我们只给出了粗略的设计，现在我们来一块实现一下。我们先来看最简单、最直接的一种实现方式。代码如下所示：

复制代码

```
1 public enum NotificationEmergencyLevel {
2     SEVERE, URGENCY, NORMAL, TRIVIAL
3 }
4
5 public class Notification {
6     private List<String> emailAddresses;
7     private List<String> telephones;
```

```
8     private List<String> wechatIds;
9
10    public Notification() {}
11
12    public void setEmailAddress(List<String> emailAddress) {
13        this.emailAddresses = emailAddress;
14    }
15
16    public void setTelephones(List<String> telephones) {
17        this.telephones = telephones;
18    }
19
20    public void setWechatIds(List<String> wechatIds) {
21        this.wechatIds = wechatIds;
22    }
23
24    public void notify(NotificationEmergencyLevel level, String message) {
25        if (level.equals(NotificationEmergencyLevel.SEVERE)) {
26            //...自动语音电话
27        } else if (level.equals(NotificationEmergencyLevel.URGENCY)) {
28            //...发微信
29        } else if (level.equals(NotificationEmergencyLevel.NORMAL)) {
30            //...发邮件
31        } else if (level.equals(NotificationEmergencyLevel.TRIVIAL)) {
32            //...发邮件
33        }
34    }
35 }
36
37 //在API监控告警的例子中，我们如下方式来使用Notification类：
38 public class ErrorAlertHandler extends AlertHandler {
39     public ErrorAlertHandler(AlertRule rule, Notification notification){
40         super(rule, notification);
41     }
42
43
44     @Override
45     public void check(ApiStatInfo apiStatInfo) {
46         if (apiStatInfo.getErrorCount() > rule.getMatchedRule(apiStatInfo.getApi()).g
47             notification.notify(NotificationEmergencyLevel.SEVERE, "...");
48         }
49     }
50 }
```




Notification 类的代码实现有一个最明显的问题，那就是有很多 if-else 分支逻辑。实际上，如果每个分支中的代码都不复杂，后期也没有无限膨胀的可能（增加更多 if-else 分支判断），那这样的设计问题并不大，没必要非得一定要摒弃 if-else 分支逻辑。

不过，Notification 的代码显然不符合这个条件。因为每个 if-else 分支中的代码逻辑都比较复杂，发送通知的所有逻辑都扎堆在 Notification 类中。我们知道，类的代码越多，就越难读懂，越难修改，维护的成本也就越高。很多设计模式都是试图将庞大的类拆分成更细小的类，然后再通过某种更合理的结构组装在一起。

针对 Notification 的代码，我们将不同渠道的发送逻辑剥离出来，形成独立的消息发送类（MsgSender 相关类）。其中，Notification 类相当于抽象，MsgSender 类相当于实现，两者可以独立开发，通过组合关系（也就是桥梁）任意组合在一起。所谓任意组合的意思就是，不同紧急程度的消息和发送渠道之间的对应关系，不是在代码中固定写死的，我们可以动态地去指定（比如，通过读取配置来获取对应关系）。

按照这个设计思路，我们对代码进行重构。重构之后的代码如下所示：

 复制代码

```
1 public interface MsgSender {
2     void send(String message);
3 }
4
5 public class TelephoneMsgSender implements MsgSender {
6     private List<String> telephones;
7
8     public TelephoneMsgSender(List<String> telephones) {
9         this.telephones = telephones;
10    }
11
12    @Override
13    public void send(String message) {
14        //...
15    }
16
17 }
18
19 public class EmailMsgSender implements MsgSender {
20     // 与TelephoneMsgSender代码结构类似，所以省略...
21 }
22
```

```

23 public class WechatMsgSender implements MsgSender {
24     // 与TelephoneMsgSender代码结构类似，所以省略...
25 }
26
27 public abstract class Notification {
28     protected MsgSender msgSender;
29
30     public Notification(MsgSender msgSender) {
31         this.msgSender = msgSender;
32     }
33
34     public abstract void notify(String message);
35 }
36
37 public class SevereNotification extends Notification {
38     public SevereNotification(MsgSender msgSender) {
39         super(msgSender);
40     }
41
42     @Override
43     public void notify(String message) {
44         msgSender.send(message);
45     }
46 }
47
48 public class UrgencyNotification extends Notification {
49     // 与SevereNotification代码结构类似，所以省略...
50 }
51 public class NormalNotification extends Notification {
52     // 与SevereNotification代码结构类似，所以省略...
53 }
54 public class TrivialNotification extends Notification {
55     // 与SevereNotification代码结构类似，所以省略...
56 }

```

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

总体上来讲，桥接模式的原理比较难理解，但代码实现相对简单。

对于这个模式有两种不同的理解方式。在 GoF 的《设计模式》一书中，桥接模式被定义为：“将抽象和实现解耦，让它们可以独立变化。”在其他资料和书籍中，还有另外一种更加简



单的理解方式：“一个类存在两个（或多个）独立变化的维度，我们通过组合的方式，让这两个（或多个）维度可以独立进行扩展。”

对于第一种 GoF 的理解方式，弄懂定义中“抽象”和“实现”两个概念，是理解它的关键。定义中的“抽象”，指的并非“抽象类”或“接口”，而是被抽象出来的一套“类库”，它只包含骨架代码，真正的业务逻辑需要委派给定义中的“实现”来完成。而定义中的“实现”，也并非“接口的实现类”，而是一套独立的“类库”。“抽象”和“实现”独立开发，通过对象之间的组合关系，组装在一起。

对于第二种理解方式，它非常类似我们之前讲过的“组合优于继承”设计原则，通过组合关系来替代继承关系，避免继承层次的指数级爆炸。

## 课堂讨论

在桥接模式的第二种理解方式的第一段代码实现中，Notification 类中的三个成员变量通过 set 方法来设置，但是这样的代码实现存在一个明显的问题，那就是 emailAddresses、telephones、wechatIds 中的数据有可能在 Notification 类外部被修改，那如何重构代码才能避免这种情况的发生呢？

 复制代码

```
1 public class Notification {
2     private List<String> emailAddresses;
3     private List<String> telephones;
4     private List<String> wechatIds;
5
6     public Notification() {}
7
8     public void setEmailAddress(List<String> emailAddress) {
9         this.emailAddresses = emailAddress;
10    }
11
12    public void setTelephones(List<String> telephones) {
13        this.telephones = telephones;
14    }
15
16    public void setWechatIds(List<String> wechatIds) {
17        this.wechatIds = wechatIds;
18    }
19    //...
```

欢迎留言和我分享你的思考和疑惑。如果有收获，也欢迎你把这篇文章分享给你的朋友。

## AI智能总结

桥接模式是一种结构型设计模式，旨在将抽象和实现解耦，让它们可以独立变化。本文通过JDBC驱动的例子解释了桥接模式的应用，以及在API接口监控告警的例子中的实际应用。文章重点讲解了桥接模式的原理和实际应用，以及对桥接模式的两种不同理解方式。在第二种理解方式的代码实现中，通过组合关系将不同渠道的发送逻辑剥离出来，形成独立的消息发送类，从而避免了庞大类的问题。读者需要重点掌握桥接模式的原理和实际应用，以及对桥接模式的两种不同理解方式。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 全部留言 (173)

最新 精选



zhengyu.nie

2020-04-29

举个很简单的例子，现在有两个纬度

Car 车 （奔驰、宝马、奥迪等）

Transmission 档位类型 （自动挡、手动挡、手自一体等）

按照继承的设计模式，Car是一个Abstract基类，假设有M个车品牌，N个档位一共要写M\*N个类去描述所有车和档位的结合。

而当我们使用桥接模式的话，我首先new一个具体的Car（如奔驰），再new一个具体的Transmission（比如自动挡）。然后奔驰.set(手动挡)就可以了。

那么这种模式只有M+N个类就可以描述所有类型，这就是M\*N的继承类爆炸简化成了M+N组合。

```
public abstract class AbstractCar {  
  
    protected Transmission gear;  
  
    public abstract void run();  
  
    public void setTransmission(Transmission gear) {
```

```
    this.gear = gear;
  }

}
```

所以桥接模式解决的应该是继承爆炸问题。

可以看作是两个abstract组合在一起，独立去拓展，在运行之前将两个具体实现组合到一起。遵循以下原则

- 依赖倒置原则
- 迪米特法则
- 里氏替换原则
- 接口隔离原则
- 单一职责原则
- 开闭原则

作者回复: 是我说的第二种理解方式

共 29 条评论 >

👍 248



蹦哒

2020-06-13

老师请问是否可以这样理解：代理模式是一个类与另一个类的组合，桥接模式是一组类和另外一组类的组合

作者回复: 有点那个意思~👍

共 3 条评论 >

👍 95



乾坤瞬间

2020-11-23

回过头再看，总结一下

桥接在网络osi模型中的链路层上代表一种设备，这种设备通过学习连接到此设备上的计算机macid来识别并原封不动的转发数据包。在软件上抽象层类似于桥接设备，具体实现类似于连接到抽象层的设备一样，如果用户通过抽象的桥接层发送消息，那么就要通过macid寻找具体设备，而这个macid在桥接模式中类似于抽象层定义的需要由具体层实现的方法的集合。同时也要注意一下物理链路寻找macid的过程是在第一次的时候就动态绑定指定计算机macid到

桥接器中，换句话说，可以通过在创建类或者初始化类的时候直接就绑定了对象。这个过程可以通过static方法块和初始化函数中创建

作者回复: 嗯嗯 桥接不好理解



风不会停息。

2020-04-30

个人理解，这个模式，跟组合模式很像，两者的区别是什么呢？

作者回复: 组合模式跟他完全是两回事，你可以先看下组合模式再说。它有点类似“组合”关系。

共 2 条评论 >



业余爱好者

2020-03-04

桥接看着就像是面向接口编程这一原则的原旨---将实现与抽象分离。让我迷惑的是，让两者独立变化的说法，接口不是应该稳定吗，为什么要变化？

多个纬度独立变化那个解释倒是比较容易理解。文中举的警报的例子很贴切。紧急程度和警报的方式可以是两个不同的纬度。可以有不同的组合方式。这与slf4j这一日志门面的设计有异曲同工之妙。slf4j其中有三个核心概念，logger,appender和encoder。分别指这个日志记录器负责哪个类的日志，日志打印到哪里以及日志打印的格式。三个纬度上可以有不同的实现，使用者可以在每一纬度上定义多个实现，配置文件中将各个纬度的某一个实现组合在一起就ok了。

行文至此，开头的那个问题也有了答案。一句话就是，桥接就是面向接口编程的集大成者。面向接口编程只是说在系统的某一个功能上将接口和实现解藕，而桥接是详细的分析系统功能，将各个独立的纬度都抽象出来，使用时按需组合。

共 12 条评论 >

211



下雨天

2020-02-24

课后题：可以考虑使用建造者模式来重构！参见46讲中

建造者使用场景：

1.构造方法必填属性很多，需要检验

- 2.类属性之间有依赖关系或者约束条件
- 3.创建不可变对象(此题刚好符合这种场景)

共 12 条评论>

👍 84



松花皮蛋me

2020-02-24

这个模式和策略模式的区别是？

共 14 条评论>

👍 33



李朝辉

2020-03-06

一点思考：如果notification类针对一次告警，需要同时在微信、电话、邮件上发送通知，当前的Notification类定义就没办法满足条件了，可以将组合的MsgSender变成一个list或者set，将不同渠道的sender注册进去，这样，就可以在调用notify的时候，将list或set内的sender，都调用一遍send

共 3 条评论>

👍 21



忆水寒

2020-02-24

参数不多的情况可以在构造函数初始化，如果参数较多 就可以使用建造者模式初始化。



👍 21



攻城拔寨

2020-02-28

我觉得桥接模式解释成： 一个类存在不同纬度的变化，可以通过组合的方式，让它们独自扩展。

栗子：白色圆形，白色正方形，黑色圆形，黑色正方形。抽象成 颜色 跟 形状 两个纬度去搞，就是桥接模式啦。

至于 jdbc 的，我水平有限啊，还是理解不了～

共 8 条评论>

👍 17