

第三篇 结构型设计模式

第一章 代理模式

代理设计模式（Proxy Design Pattern）是一种结构型设计模式，它为其他对象**提供一个代理，以控制对这个对象的访问**。代理模式可以用于实现懒加载、安全访问控制、日志记录等功能。

在设计模式中，代理模式可以分为**静态代理和动态代理**。静态代理是指代理类在编译时就已经确定，而动态代理是指代理类在运行时动态生成。

一、静态代理的原理解析

代理模式（Proxy Design Pattern）的原理和代码实现都不难掌握。它在不改变原始类（或叫被代理类）代码的情况下，通过引入代理类来给原始类附加功能。我们通过一个简单的例子来解释一下这段话。

二、静态代理使用场景

我们通过询问chatgpt，让他帮我们列举几个代理设计模式的使用场景，以下都是通过静态代理实现的，我们不妨先看看他的实现方式，最后再去详细的做出总结：

1、缓存代理

ON

那您具体说说缓存代理的具体应用

< 2 / 2 >



缓存代理（Caching Proxy）是一种特殊类型的代理模式，它可以为耗时的操作或者重复的请求提供缓存功能，从而提高程序的执行效率。缓存代理通常会在内部维护一个缓存数据结构，如 HashMap 或者 LinkedHashMap，用来存储已经处理过的请求及其结果。

以下是一个缓存代理的应用示例：

假设有一个数据查询接口，它从数据库或其他数据源中检索数据。在没有缓存代理的情况下，每次查询都需要访问数据库，这可能会导致较高的资源消耗和延迟。通过引入缓存代理，我们可以将查询结果存储在内存中，从而避免重复查询数据库。

缓存代理（Caching Proxy）是一种特殊类型的代理模式，它可以为耗时的操作或者重复的请求提供缓存功能，从而提高程序的执行效率。缓存代理通常会在**内部维护一个缓存数据结构**，如 HashMap 或者 LinkedHashMap，用来存储已经处理过的请求及其结果。

以下是一个缓存代理的应用示例：

假设有一个数据查询接口，它从数据库或其他数据源中检索数据。在没有缓存代理的情况下，每次查询都需要访问数据库，这可能会导致较高的资源消耗和延迟。通过引入缓存代理，我们可以将查询结果存储在内存中，从而避免重复查询数据库。

首先，我们定义一个数据查询接口：

```
public interface DataQuery {  
    String query(String queryKey);  
}
```

然后，实现一个真实的数据查询类，它从数据库中检索数据：

```
public class DatabaseDataQuery implements DataQuery {  
    @Override  
    public String query(String queryKey) {  
        // 查询数据库并返回结果  
        return "Result from database: " + queryKey;  
    }  
}
```

接下来，我们创建一个缓存代理类，它实现了 DataQuery 接口，并在内部使用 HashMap 作为缓存：

```
public class CachingDataQueryProxy implements DataQuery {  
    private final DataQuery realDataQuery;  
    private final Map<String, String> cache;  
  
    public CachingDataQueryProxy(DataQuery realDataQuery) {  
        this.realDataQuery = new DatabaseDataQuery();  
        cache = new HashMap<>();  
    }  
  
    @Override  
    public String query(String queryKey) {  
        String result = cache.get(queryKey);  
        if (result == null) {  
            result = realDataQuery.query(queryKey);  
        }  
    }  
}
```

```

        cache.put(queryKey, result);
        System.out.println("Result retrieved from database and added to cache.");
    } else {
        System.out.println("Result retrieved from cache.");
    }
    return result;
}
}

```

最后，我们可以在客户端代码中使用缓存代理：

```

public class Client {
    public static void main(String[] args) {
        DataQuery realDataQuery = new DatabaseDataQuery();
        DataQuery cachingDataQueryProxy = new
        CachingDataQueryProxy(realDataQuery);

        String queryKey = "example_key";

        // 第一次查询，从数据库中获取数据并将其缓存
        System.out.println(cachingDataQueryProxy.query(queryKey));

        // 第二次查询相同的数据，从缓存中获取
        System.out.println(cachingDataQueryProxy.query(queryKey));
    }
}

```

通过这个示例，你可以看到缓存代理如何提供缓存功能，以提高程序的执行效率。

2、安全代理

(Security Proxy) 是一种**代理模式的应用**，它用于控制对真实主题对象的访问。通过安全代理，可以实现访问控制、权限验证等安全相关功能。

以下是一个简单的安全代理示例：

假设我们有一个敏感数据查询接口，只有具有特定权限的用户才能访问：

首先，我们定义一个数据查询接口：

```
public interface SensitiveDataQuery {  
    String queryData(String userId);  
}
```

接着，实现一个真实的敏感数据查询类：

```
public class SensitiveDataQueryImpl implements SensitiveDataQuery {  
    @Override  
    public String queryData(String userId) {  
        // 查询敏感数据并返回结果  
        return "Sensitive data for user: " + userId;  
    }  
}
```

然后，我们创建一个安全代理类，它实现了 SensitiveDataQuery 接口，并在内部进行权限验证：

```
public class SecurityProxy implements SensitiveDataQuery {  
    private final SensitiveDataQuery sensitiveDataQuery;  
    private final UserAuthenticator userAuthenticator;  
  
    public SecurityProxy(SensitiveDataQuery sensitiveDataQuery,  
        UserAuthenticator userAuthenticator) {  
        this.sensitiveDataQuery = sensitiveDataQuery;  
        this.userAuthenticator = userAuthenticator;  
    }  
  
    @Override  
    public String queryData(String userId) {  
        if (userAuthenticator.hasPermission(userId)) {  
            return sensitiveDataQuery.queryData(userId);  
        } else {  
            return "Access Denied: Insufficient permission for user" + userId;  
        }  
    }  
}
```

在此示例中，我们使用一个 UserAuthenticator 类来模拟用户权限验证：

```

public class UserAuthenticator {
    private final List<String> authorizedUserIds;

    public UserAuthenticator() {
        // 模拟从数据库或配置文件中获取已授权的用户列表
        authorizedUserIds = Arrays.asList("user1", "user2", "user3");
    }

    public boolean hasPermission(String userId) {
        return authorizedUserIds.contains(userId);
    }
}

```

最后，我们可以在客户端代码中使用安全代理：

```

public class Client {
    public static void main(String[] args) {
        SensitiveDataQuery sensitiveDataQuery = new SensitiveDataQueryImpl();
        UserAuthenticator userAuthenticator = new UserAuthenticator();
        SensitiveDataQuery securityProxy = new SecurityProxy(sensitiveDataQuery,
            userAuthenticator);

        String userId1 = "user1";
        String userId2 = "user4";

        // 用户1具有访问权限
        System.out.println(securityProxy.queryData(userId1));

        // 用户4没有访问权限
        System.out.println(securityProxy.queryData(userId2));
    }
}

```

通过这个示例，你可以看到安全代理如何在访问真实主题对象之前进行权限验证

写完了，这两个例子后，我觉得他所实现的方式和后边我们要学习的装饰者设计模式很像，于是我追问：



你的观察很到位。实际上，代理模式和装饰者模式之间有一定的相似性。两者都属于结构型设计模式，它们都使用组合而不是继承来扩展对象的行为。这两种模式都涉及到一个接口和

你的观察很到位。实际上，代理模式和装饰者模式之间有一定的相似性。两者都属于结构型设计模式，它们都使用组合而不是继承来扩展对象的行为。这两种模式都涉及到一个接口和一些实现该接口的类，其中一个类充当代理或装饰者，将接口方法的调用委托给另一个类。不过，它们的主要区别在于它们的目的是应用场景：

1. 目的：

- 代理模式的主要目的是控制对实际对象的访问，例如访问控制、懒加载、远程访问等。
- 装饰者模式的主要目的是在不修改现有类的基础上，动态地扩展对象的功能。装饰者模式可以用来给一个对象添加新功能，而不影响其他对象。

2. 应用场景：

- 代理模式通常用于创建一个具有相同接口的替代对象，来控制对实际对象的访问。
- 装饰者模式通常用于在运行时动态地为对象添加功能，同时不改变对象的结构。

在我给出的缓存代理示例中，确实可以看出与装饰者模式的相似之处。这是因为缓存代理在这里也起到了扩展功能的作用。然而，代理模式的主要目的仍然是控制对实际对象的访问。在这个例子中，我们通过缓存来减少对数据库的访问次数。当然，在实际应用中，代理模式和装饰者模式可能会共同使用以满足不同的需求。

3、虚拟代理

(Virtual Proxy) 是一种代理模式，用于在需要时**延迟创建耗时或资源密集型**对象。虚拟代理在初始访问时才创建实际对象，之后将直接使用该对象。这可以避免在实际对象尚未使用的情况下就创建它，从而节省资源。

以下是一个虚拟代理的应用示例：

假设我们有一个大型图片类，它从网络加载图像。由于图像可能非常大，我们希望在需要显示时才加载它。为了实现这一点，我们可以创建一个虚拟代理来代表大型图片类。

首先，我们定义一个图片接口：

```
public interface Image {  
    void display();  
}
```

然后，实现一个大型图片类，它从网络加载图像并实现 `display()` 方法：

```
public class LargeImage implements Image {  
    private final String imageUrl;  
  
    public LargeImage(String imageUrl) {  
        this.imageUrl = imageUrl;  
        loadImageFromNetwork();  
    }  
  
    private void loadImageFromNetwork() {  
        System.out.println("Loading image from network: " + imageUrl);  
        // 真实的图像加载逻辑...  
    }  
  
    @Override  
    public void display() {  
        System.out.println("Displaying image: " + imageUrl);  
    }  
}
```

接下来，我们创建一个虚拟代理类，它实现了 `Image` 接口，并在内部使用 `LargeImage`：

```
public class VirtualImageProxy implements Image {  
    private final String imageUrl;  
    private LargeImage largeImage;  
  
    public VirtualImageProxy(String imageUrl) {  
        this.imageUrl = imageUrl;  
    }  
  
    @Override  
    public void display() {  
        if (largeImage == null) {  
            largeImage = new LargeImage(imageUrl);  
        }  
        largeImage.display();  
    }  
}
```

```
}  
}
```

最后，我们可以在客户端代码中使用虚拟代理：

```
public class Client {  
    public static void main(String[] args) {  
        Image virtualImageProxy = new  
        VirtualImageProxy("https://example.com/large-image.jpg");  
  
        System.out.println("Image will not be loaded until it is displayed.");  
  
        // 调用 display() 方法时，才会创建并加载大型图片  
        virtualImageProxy.display();  
    }  
}
```

通过这个示例，你可以看到虚拟代理如何实现懒加载，以减少资源消耗和提高程序性能。当实际对象的创建和初始化非常耗时或占用大量资源时，虚拟代理是一个很好的选择。

4、远程代理

(Remote Proxy) 是一种代理模式，用于访问位于不同地址空间的对象。远程代理可以为本地对象提供与远程对象相同的接口，使得客户端可以透明地访问远程对象。通常，远程代理需要处理网络通信、序列化和反序列化等细节。我们以后做rpc时也会使用。

以下是一个简化的远程代理示例：

首先，我们定义一个服务接口：

```
public interface RemoteService {  
    String fetchData(String dataId);  
}
```

然后，实现一个远程服务类，它在服务器端运行并实现 `fetchData()` 方法：


```

public class RemoteServiceImpl implements RemoteService {
    @Override
    public String fetchData(String dataId) {
        // 实际操作，例如从数据库获取数据
        return "Data from remote service: " + dataId;
    }
}

```

接下来，我们创建一个远程代理类，它实现了 `RemoteService` 接口，并在内部处理网络通信等细节：

```

public class RemoteServiceProxy implements RemoteService {
    private final String remoteServiceUrl;
    private RemoteService remoteService;

    public RemoteServiceProxy(String remoteServiceUrl) {
        this.remoteServiceUrl = remoteServiceUrl;
        this.remoteService = new RemoteService();
    }

    @Override
    public String fetchData(String dataId) {
        // 网络通信、序列化和反序列化等逻辑
        System.out.println("Connecting to remote service at: " + remoteServiceUrl);
        // 假设我们已经获取到远程服务的数据
        String result = remoteService.fetchData(dataId);
        System.out.println("Received data from remote service.");
        return result;
    }
}

```

最后，我们可以在客户端代码中使用远程代理：

```
public class Client {  
    public static void main(String[] args) {  
        RemoteService remoteServiceProxy = new  
RemoteServiceProxy("https://example.com/remote-service");  
  
        // 使用远程代理访问远程服务  
        String dataId = "example_data_id";  
        String result = remoteServiceProxy.fetchData(dataId);  
  
        System.out.println("Result: " + result);  
    }  
}
```

这个示例展示了远程代理的基本概念。请注意，为了简化代码，这里省略了实际的网络通信、序列化和反序列化逻辑。在实际应用中，你可以使用 Java RMI、gRPC 或其他远程通信库来实现远程代理。

5、静态代理步骤总结

通过前四个案例，我们也大致了解了静态代理的使用方式，其大致流程如下：

1. 创建一个接口，定义代理类和被代理类共同实现的方法。
2. 创建被代理类，实现这个接口，并且在其中定义实现方法。
3. 创建代理类，也要实现这个接口，同时在其中定义一个被代理类的对象作为成员变量。
4. 在代理类中实现接口中的方法，方法中调用被代理类中的对应方法。
5. 通过创建代理对象，并调用其方法，方法增强。

这样，被代理类的方法就会被代理类所覆盖，实现了对被代理类的**增强或修改**。

当然在静态代理中，**也可以使用继承来实现代理**。具体步骤如下：

1. 创建被代理类，定义需要被代理的方法。
2. 创建代理类，继承被代理类，重写被代理类中的方法，对方法进行增强。
3. 在重写的方法中添加代理逻辑，例如在调用被代理类中的方法前后添加日志记录、安全检查等功能。
4. 在使用代理类时，创建代理类的对象，调用重写的方法。

这样，被代理类的方法就会被代理类所覆盖，实现了对被代理类的增强或修改。使用继承来实现代理的好处是简单易懂，不需要创建接口，同时继承可以继承被代理类的属性和方法，可以更方便地访问被代理类中的成员。但是，这种方式也有一些缺点，**例如代理类与被代理类的耦合度较高，不够灵活。**

四、动态代理

Java 中动态代理的实现方式主要有两种：基于 JDK 的动态代理和基于 CGLIB 的动态代理。

静态代理**需要手动编写代理类，代理类与被代理类实现相同的接口或继承相同的父类**，对被代理对象进行包装。**在程序运行前，代理类的代码就已经生成**，并在程序运行时调用。静态代理的优点是简单易懂，缺点是**需要手动编写代理类，代码复杂度较高，且不易扩展。**

动态代理是**在程序运行时动态生成代理类**，无需手动编写代理类，**大大降低了代码的复杂度**。动态代理一般使用 Java 提供的反射机制实现，可以对任意实现了接口的类进行代理。动态代理的优点是灵活性高，可以根据需要动态生成代理类，缺点是性能相对较低，由于使用反射机制，在运行时会产生额外的开销。

总之，静态代理和动态代理都是代理模式的实现方式，其主要区别在于代理类的生成时机和方式。静态代理需要手动编写代理类，适用于代理类数量较少、不需要频繁修改的场景。而动态代理不需要手动编写代理类，可以动态生成代理类，适用于代理类数量较多、需要频繁修改的场景。

1、基于 JDK 的动态代理实现步骤

基于 JDK 的动态代理需要使用 `java.lang.reflect.Proxy` 类和 `java.lang.reflect.InvocationHandler` 接口。我们依旧使用上述的缓存代理的案例来实现，具体步骤如下：

(1) 定义一个接口，声明需要代理的方法：

```
public interface DataQuery {  
    String query(String queryKey);  
    String queryAll(String queryKey);  
}
```

(2) 创建一个被代理类，实现这个接口，并在其中定义实现方法：

```

public class DatabaseDataQuery implements DataQuery {
    @Override
    public String query(String queryKey) {
        // 他会使用数据源从数据库查询数据很慢
        System.out.println("正在从数据库查询数据");
        return "result";
    }

    @Override
    public String queryAll(String queryKey) {
        // 他会使用数据源从数据库查询数据很慢
        System.out.println("正在从数据库查询数据");
        return "all result";
    }
}

```

(3) 创建一个代理类，实现 `InvocationHandler` 接口，并在其中定义一个被代理类的对象作为属性。

```

public class CacheInvocationHandler implements InvocationHandler {

    private HashMap<String,String> cache = new LinkedHashMap<>(256);

    private DataQuery databaseDataQuery;

    public CacheInvocationHandler(DatabaseDataQuery databaseDataQuery) {
        this.databaseDataQuery = databaseDataQuery;
    }

    public CacheInvocationHandler() {
        this.databaseDataQuery = new DatabaseDataQuery();
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws
    Throwable {
        // 1、判断是哪一个方法
        String result = null;
        if("query".equals(method.getName())){
            // 2、查询缓存，命中直接返回
            result = cache.get(args[0].toString());
            if(result != null){

```

```

        System.out.println("数据从缓存重获取。");
        return result;
    }

    // 3、未命中，查数据库（需要代理实例）
    result = (String) method.invoke(databaseDataQuery, args);

    // 4、如果查询到了,进行呢缓存
    cache.put(args[0].toString(),result);
    return result;
}

// 当其他的方法被调用，不希望被干预，直接调用原生的方法
return method.invoke(databaseDataQuery,args);
}
}

```

在代理类中，我们实现了 `InvocationHandler` 接口，并在其中定义了一个被代理类的对象作为属性。在 `invoke` 方法中，我们可以对被代理对象的方法进行增强，并在方法调用前后输出日志。

(4) 在使用代理类时，创建被代理类的对象和代理类的对象，并使用 `Proxy.newProxyInstance` 方法生成代理对象。

```

public class Main {

    public static void main(String[] args) {
        // jdk提供的代理实现，主要是使用Proxy类来完成
        // 1、classLoader：被代理类的类加载器
        ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
        // 2、代理类需要实现的接口数组
        Class[] interfaces = new Class[]{DataQuery.class};
        // 3、InvocationHandler
        InvocationHandler invocationHandler = new CacheInvocationHandler();

        DataQuery dataQuery = (DataQuery)Proxy.newProxyInstance(
            classLoader, interfaces, invocationHandler
        );

        // 事实上调用了query方法的使用，他是调用了invoke
        String result = dataQuery.query("key1");
        System.out.println(result);
        System.out.println("-----");
    }
}

```

```

        result = dataQuery.query("key1");
        System.out.println(result);
        System.out.println("-----");
        result = dataQuery.query("key2");
        System.out.println(result);
        System.out.println("+++++++");

        // 事实上调用queryAll方法的使用，他是调用了invoke
        result = dataQuery.queryAll("key1");
        System.out.println(result);
        System.out.println("-----");
        result = dataQuery.queryAll("key1");
        System.out.println(result);
        System.out.println("-----");
        result = dataQuery.queryAll("key2");
        System.out.println(result);
        System.out.println("-----");
    }
}

```

在这个示例中，我们使用 `Proxy.newProxyInstance` 方法生成代理对象，并将代理对象转换成 `DataQuery` 接口类型，以便调用其代理的方法。在代理对象调用方法时，会调用 `CacheInvocationHandler` 类中的 `invoke` 方法，实现对被代理对象的方法的增强。

2、基于 CGLIB 的动态代理实现步骤

基于 CGLIB 的动态代理需要使用 `net.sf.cglib.proxy.Enhancer` 类和 `net.sf.cglib.proxy.MethodInterceptor` 接口。具体步骤如下：

- (1) 创建一个被代理类，定义需要被代理的方法。

```

public class DatabaseDataQuery {

    public String query(String queryKey) {
        // 他会使用数据源从数据库查询数据很慢
        System.out.println("正在从数据库查询数据");
        return "result";
    }

    public String queryAll(String queryKey) {
        // 他会使用数据源从数据库查询数据很慢
    }
}

```

```

        System.out.println("正在从数据库查询数据");
        return "all result";
    }
}

```

(2) 创建一个方法拦截器类，实现 `MethodInterceptor` 接口，并在其中定义一个被代理类的对象作为属性。

```

public class CacheMethodInterceptor implements MethodInterceptor {

    private HashMap<String,String> cache = new LinkedHashMap<>(256);

    private DatabaseDataQuery databaseDataQuery;

    public CacheMethodInterceptor() {
        this.databaseDataQuery = new DatabaseDataQuery();
    }

    @Override
    public Object intercept(Object obj, Method method, Object[] args,
        MethodProxy proxy) throws Throwable {

        // 1、判断是哪一个方法
        String result = null;
        if("query".equals(method.getName())){
            // 2、查询缓存，命中直接返回
            result = cache.get(args[0].toString());
            if(result != null){
                System.out.println("数据从缓存重获取。");
                return result;
            }

            // 3、未命中，查数据库（需要代理实例）
            result = (String) method.invoke(databaseDataQuery, args);

            // 4、如果查询到了,进行呢缓存
            cache.put(args[0].toString(),result);
            return result;
        }

        return method.invoke(databaseDataQuery,args);
    }
}

```

在这个代理类中，我们实现了 `MethodInterceptor` 接口，并在其中定义了一个被代理类的对象作为属性。在 `intercept` 方法中，我们可以对被代理对象的方法进行增强，并在方法调用前后输出日志。

(3) 在使用代理类时，创建被代理类的对象和代理类的对象，并使用 `Enhancer.create` 方法生成代理对象。

```
public class Main {
    public static void main(String[] args) {

        // cglib通过Enhancer
        Enhancer enhancer = new Enhancer();
        // 设置他的父类
        enhancer.setSuperclass(DatabaseDataQuery.class);
        // 设置一个方法拦截器，用来拦截方法
        enhancer.setCallback(new CacheMethodInterceptor());
        // 创建代理类
        DatabaseDataQuery databaseDataQuery =
            (DatabaseDataQuery)enhancer.create();

        databaseDataQuery.query("key1");
        databaseDataQuery.query("key1");
        databaseDataQuery.query("key2");

    }
}
```

在这个示例中，我们使用 `Enhancer.create` 方法生成代理对象，并将代理对象转换成 `RealSubject` 类型，以便调用 `request` 方法。在代理对象调用 `request` 方法时，会调用 `DynamicProxy` 类中的 `intercept` 方法，实现对被代理对象的增强。

在实际应用中，基于 CGLIB 的动态代理可以代理任意类，但是**生成的代理类比较重量级**。如果被代理类是一个接口，建议使用基于 JDK 的动态代理来实现，这也是 spring 的做法；如果被代理类没有实现接口或者需要代理的方法是 `final` 方法，建议使用基于 CGLIB 的动态代理来实现。

3、spring中aop的使用步骤

在 Spring 中，AOP（面向切面编程）提供了一种有效的方式来对程序中的多个模块进行横切关注点的处理，例如日志、事务、缓存、安全等。使用 Spring AOP，可以在程序运行时动态地将代码织入到目标对象中，从而实现对目标对象的增强。

Spring AOP 的使用步骤如下：

(1) 引入 AOP 相关依赖。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

(2) 开启自动代理@EnableAspectJAutoProxy

```
@SpringBootApplication
@EnableAspectJAutoProxy
public class Main {
    public static void main(String[] args) {
        SpringApplication.run(Main.class, args);
    }
}
```

(3) 定义接口和实现类，并将具体实现注入容器：

```
public interface DataQuery {
    String query(String queryKey);
}

@Component
public class DatabaseDataQuery implements DataQuery {
    @Override
    public String query(String queryKey) {
        // 他会使用数据源从数据库查询数据很慢
        System.out.println("正在从数据库查询数据");
        return "result";
    }
}
```

(4) 定义切面类，对方法做增强

```
@Component
@Aspect
public class CacheAspectj {

    @Pointcut("execution(*
com.ydlclass.proxy.dynamicProxy.aop.DatabaseDataQuery.query(..)")
    public void pointcut() {}

    @Around("pointcut()")
    public String around(ProceedingJoinPoint joinPoint){
        Object[] args = joinPoint.getArgs();
        String key = args[0].toString();

        // 1、查询缓存，命中则返回
        String result = Cache.get(key);
        if(result != null){
            System.out.println("数据从缓存中获取");
            return result;
        }

        // 未命中则去数据库查询，实际上是调用被代理bean的方法
        try {
            result = joinPoint.proceed().toString();
            // 如果查询有结果，进行缓存
            Cache.put(key,result);
        } catch (Throwable e) {
            throw new RuntimeException(e);
        }
        return result;
    }
}
```

定义缓存

```

public class Cache {
    private static Map<String,String> map = new ConcurrentHashMap<>(256);

    public static String get(String key){
        return map.get(key);
    }

    public static void put(String key,String value){
        map.put(key, value);
    }
}

```

(5) 测试:

在使用增强后的 Bean 时，Spring AOP 会自动代理这些 Bean，并在方法调用前后调用相应的通知方法。例如：

```

@SpringBootTest
public class AopTest {

    @Resource
    private DataQuery dataQuery;

    @Test
    public void testAop(){
        dataQuery.query("key1");
        dataQuery.query("key1");
        dataQuery.query("key2");
    }
}

```

在这个示例中，我们定义了一个名为 `MyService` 的服务类，并使用 `@Service` 注解将其添加到 Spring 容器中。由于我们已经添加了名为 `loggingAspect` 的切面，因此在调用 `doSomething` 方法时，Spring AOP 会自动增强这个方法，调用前置通知和后置通知。

这就是使用 Spring AOP 的基本步骤。需要注意的是，在配置切点时，我们可以使用多种方式来定义切点，例如基于表达式、基于注解、基于 XML 配置等。具体选择哪种方式，取决于项目的实际需求和开发者的个人习惯。

在使用 Spring AOP 时，还需要注意以下几点：

- 如果目标对象实现了接口，则默认使用 JDK 动态代理进行代理；如果目标对象没有实现接口，则使用 CGLIB 进行代理。如果想要强制使用 CGLIB 进行代理，可以在配置类中使用 `@EnableAspectJAutoProxy(proxyTargetClass = true)` 注解。
- 前置通知、后置通知、环绕通知等通知类型都可以使用 `@Before`、`@After`、`@Around` 等注解进行定义。在通知方法中，可以通过 `JoinPoint` 参数获取目标方法的参数等信息。
- 如果需要将通知方法定义在不同的切面中，可以使用 `@Order` 注解指定切面的执行顺序。
- 如果需要在同一个通知方法中定义多个切点，可以使用 `&&` 和 `||` 等逻辑运算符来组合切点表达式。

总的来说，Spring AOP 是一种非常方便的 AOP 实现方式，可以大大简化程序中的横切关注点处理，提高代码的可重用性和可维护性。

面向切面编程（AOP）和代理是两个相关但不同的概念。

代理是一种结构型设计模式，它为**其他对象提供一种代理以控制对这个对象的访问**。代理模式可以分为静态代理和动态代理两种。

在静态代理中，代理类和被代理类都必须实现同一个接口，代理类通过实现接口的方式来实现被代理类的功能，并在调用被代理类的方法前后执行一些额外的逻辑。在动态代理中，代理类是在运行时动态生成的，不需要实现接口，通过 Java 反射机制来动态地生成代理对象，从而实现对被代理类的增强。

AOP 是一种编程范式，它通过对程序中的**多个模块进行横切关注点的处理，实现对程序行为的增强**。AOP 的实现方式可以有多种，例如**基于代理、基于字节码增强、基于注解等**。其中基于代理的 AOP 实现方式，也就是 Spring AOP，是一种常见的 AOP 实现方式。

在 Spring AOP 中，切面是 AOP 的核心概念，它由切点和通知组成。切点定义了哪些方法或类需要被增强，通知则定义了增强的具体逻辑。在基于代理的 AOP 实现方式中，代理类负责在调用被代理对象的方法前后调用相应的通知方法，从而实现对被代理对象的增强。

因此，**代理和 AOP 是两个相关的概念，代理是 AOP 的一种实现方式**，它们都可以用于在程序中实现对目标对象的增强。区别在于，代理主要是针对单个对象的方法调用进行增强，而 AOP 则是针对程序中多个模块的横切关注点进行增强。

五、动态代理的应用场景

动态代理是一种代理模式，它在运行时动态生成代理对象，而无需提前创建具体的代理类。动态代理在 Java 中通常使用 `java.lang.reflect.Proxy` 类和 `java.lang.reflect.InvocationHandler` 接口来实现。以下是一些动态代理的使用场景：

1. 日志记录：使用动态代理可以在方法调用前后自动添加日志记录，从而跟踪方法的执行过程。这样可以方便地监控系统运行情况，诊断问题，而无需修改实际的代码。
2. 性能监控：动态代理可用于测量方法执行的时间，以评估性能。在方法调用前后记录时间戳，然后计算时间差，就可以得到方法执行所需的时间。
3. 事务管理：在数据库操作中，动态代理可用于自动管理事务。在方法调用前开始一个事务，在方法成功执行后提交事务，如果发生异常，则回滚事务。这样可以确保数据的一致性和完整性。
4. 权限验证：使用动态代理可以在方法调用前进行权限验证，确保只有具有适当权限的用户才能访问受保护的资源。这可以提高系统的安全性。
5. 缓存：动态代理可用于实现方法结果的缓存。在方法调用前检查缓存，如果缓存中有结果，则直接返回，否则执行方法并将结果存入缓存。这可以提高程序的执行效率。
6. 负载均衡与故障转移：在分布式系统中，动态代理可以用于实现负载均衡和故障转移。代理对象根据某种策略（如轮询、随机等）选择一个可用的服务实例，并将请求转发给它。如果服务实例发生故障，代理对象可以自动选择另一个可用的实例。
7. API 速率限制：使用动态代理，可以在方法调用前检查 API 请求速率是否超过预设的限制。如果超过限制，可以拒绝请求或将请求延迟一段时间后再执行。
8. 数据验证：在方法调用前，动态代理可以用于验证传入的参数是否符合预期的规则和约束。这有助于确保数据的有效性和一致性。
9. 重试机制：当方法调用失败时（例如，因为网络问题、服务不可用等原因），动态代理可以实现自动重试的机制。代理对象可以在一定的时间间隔内尝试重新执行方法，直到成功或达到最大重试次数。
10. 懒加载与资源管理：动态代理可以用于实现资源的懒加载和管理。例如，代理对象可以在第一次访问资源时才创建和初始化它。此外，代理对象还可以在资源不再需要时自动释放它，以减少内存占用和提高性能。
11. 跨语言和跨平台调用：动态代理可以实现跨语言和跨平台的对象调用。例如，一个 Java 客户端可以使用动态代理调用一个基于 Python 的服务。在这种情况下，代理对象会负责处理跨语言通信的细节，如序列化、反序列化和网络传输。
12. AOP（面向切面编程）：动态代理是实现 AOP 的一种方式。AOP 允许在程序运行时动态地插入和修改横切关注点（如日志记录、性能监控等），而无需修改实际代码。动态代理可以轻松地实现 AOP，以提高代码的可维护性和可重用性。

这些仅仅是动态代理的一些应用场景。动态代理是一种非常灵活的技术，可以用于解决许多不同类型的问题。在实际项目中，根据具体需求，可以将动态代理与其他设计模式结合使用。

事实上，我们如果不是编写中间件产品，绝大部分的场景都是在spring环境下实现的，我们理解了代理的本是就是增强，他是aop的一种实现方式，工作中绝大部分的场景都是使用aop来实现的。接下来，我们列举几个例子来详细介绍aop的使用场景。

1、API 速率限制

这是一个使用 Spring AOP 实现 API 速率限制的简单示例。我们将使用 Spring Boot 和 Spring AOP。首先，确保你的项目中包含以下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

接下来，我们创建一个自定义注解 `RateLimiter`，用于标记需要进行速率限制的方法：

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface RateLimiter {
    int value() default 1;
    int durationInSeconds() default 1;
}
```

现在，我们需要创建一个切面类（Aspect），实现速率限制的逻辑。这里我们使用 Google Guava 提供的 `RateLimiter` 类

```
@Aspect
@Component
public class RateLimiterAspect {

    private final ConcurrentHashMap<String, RateLimiter> rateLimiters = new
    ConcurrentHashMap<>();

    @Pointcut("@annotation(rateLimiterAnnotation)")
    public void rateLimiterPointcut(RateLimiter rateLimiterAnnotation) {
```

```

}

@Around("rateLimiterPointcut(rateLimiterAnnotation)")
public Object around(ProceedingJoinPoint joinPoint, RateLimiter
rateLimiterAnnotation) throws Throwable {
    int permits = rateLimiterAnnotation.value();
    int durationInSeconds = rateLimiterAnnotation.durationInSeconds();

    // 使用方法签名作为 RateLimiter 的 key
    String key = joinPoint.getSignature().toLongString();
    com.google.common.util.concurrent.RateLimiter rateLimiter =
rateLimiters.computeIfAbsent(key, k ->
com.google.common.util.concurrent.RateLimiter.create((double) permits /
durationInSeconds));

    // 尝试获取令牌，如果获取到则执行方法，否则抛出异常
    if (rateLimiter.tryAcquire()) {
        return joinPoint.proceed();
    } else {
        throw new RuntimeException("Rate limit exceeded.");
    }
}
}
}

```

在这个切面类中，我们定义了一个名为 `rateLimiterPointcut` 的切入点，用于匹配带有 `@RateLimiter` 注解的方法。`around` 方法用 `@Around` 注解标注，实现了速率限制的逻辑。我们使用 `ConcurrentHashMap` 存储 `RateLimiter` 实例，以便在多线程环境下安全地访问它们。

最后，我们可以在需要进行速率限制的方法上添加 `@RateLimiter` 注解。例如，以下是一个简单的 REST 控制器：

```

@RestController
public class ApiController {

    @GetMapping("/api/limited")
    @RateLimiter(value = 10, durationInSeconds = 60) //限制为每分钟 10 次请求
    public String limitedEndpoint() {
        return "This API has a rate limit of 10 requests per minute.";
    }
}

```

```

@GetMapping("/api/unlimited")
public String unlimitedEndpoint() {
    return "This API has no rate limit.";
}
}

```

2、重试机制

下面是使用 Spring AOP 实现重试机制的一个示例。首先，我们创建一个自定义注解 `Retry`，用于标记需要进行重试的方法：

```

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Retry {
    int maxAttempts() default 3;
    long delayInMillis() default 1000;
    Class<? extends Throwable>[] retryOn() default {Throwable.class};
}

```

然后，我们需要创建一个切面类（Aspect），实现重试的逻辑：

```

@Aspect
@Component
public class RetryAspect {

    @Pointcut("@annotation(retryAnnotation)")
    public void retryPointcut(Retry retryAnnotation) {
    }

    @Around("retryPointcut(retryAnnotation)")
    public Object around(ProceedingJoinPoint joinPoint, Retry retryAnnotation)
    throws Throwable {
        int maxAttempts = retryAnnotation.maxAttempts();
        long delayInMillis = retryAnnotation.delayInMillis();
        Set<Class<? extends Throwable>> retryOn = new HashSet<>
(Array.asList(retryAnnotation.retryOn()));

        int attempts = 0;
        while (true) {
            try {
                attempts++;
            }

```



```

        return joinPoint.proceed();
    } catch (Throwable throwable) {
        if (attempts >= maxAttempts || !retryOn.contains(throwable.getClass()))
        {
            // 此处可以记录重试日志
            throw throwable;
        }
        // 等待一段时间后重试
        try {
            Thread.sleep(delayInMillis);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw throwable;
        }
    }
}
}
}
}

```

在这个切面类中，我们定义了一个名为 `retryPointcut` 的切入点，用于匹配带有 `@Retry` 注解的方法。 `around` 方法用 `@Around` 注解标注，实现了重试的逻辑。当方法抛出异常时，我们检查异常类型是否需要重试，以及是否达到最大重试次数。如果满足重试条件，则等待一段时间后再次尝试执行方法。

最后，我们可以在需要进行重试的方法上添加 `@Retry` 注解。例如，以下是一个简单的服务类：

```

@Service
public class DemoService {

    @Retry(maxAttempts = 5, delayInMillis = 500, retryOn =
    {RuntimeException.class})
    public String retryableMethod() {
        System.out.println("Executing retryableMethod...");
        throw new RuntimeException("An error occurred");
    }

    public String nonRetryableMethod() {
        System.out.println("Executing nonRetryableMethod...");
        return "This method does not have a retry mechanism.";
    }
}

```

```
}
```

在这个示例中，我们为 `retryableMethod` 方法设置了重试机制（最多重试 5 次，每次间隔 500 毫秒，仅在发生 `RuntimeException` 的时候进行重试。当然重试的时间可以动态调整，如第一次 5 秒，第二次 10 秒...

3、日志记录

使用 Spring AOP 实现日志记录的示例如下。首先，确保你的项目中包含以下依赖：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

然后，我们创建一个切面类（Aspect）来实现日志记录的逻辑：

```
@Aspect
@Component
public class LoggingAspect {
    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Pointcut("within(@org.springframework.web.bind.annotation.RestController *)")
    public void restControllerMethods() {
    }

    @Before("restControllerMethods()")
    public void logMethodCall(JoinPoint joinPoint) {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String methodName = joinPoint.getSignature().getName();
        logger.info("Entering method [{}.{}]", className, methodName);
    }

    @AfterReturning(pointcut = "restControllerMethods()", returning = "result")
    public void logMethodReturn(JoinPoint joinPoint, Object result) {
        String className = joinPoint.getSignature().getDeclaringTypeName();
        String methodName = joinPoint.getSignature().getName();
        logger.info("Exiting method [{}.{}], return value: {}", className,
            methodName, result);
    }
}
```

```
}
```

在这个切面类中，我们定义了一个名为 `restControllerMethods` 的切入点，用于匹配所有带有 `@RestController` 注解的类中的方法。我们使用 `@Before` 注解标注 `logMethodCall` 方法，以便在方法调用前记录日志。类似地，我们使用 `@AfterReturning` 注解标注 `logMethodReturn` 方法，以便在方法成功返回后记录日志。

这个示例使用了 SLF4J 作为日志记录库，你可以根据自己的需求更换其他日志记录库。请确保项目中包含 SLF4J 及其所需的依赖。

以下是一个简单的 REST 控制器示例，它将自动记录方法调用和返回的日志：

```
@RestController
public class SampleController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello, World!";
    }
}
```

当你访问 `/hello` 端点时，控制台将输出如下日志：

```
Entering method [SampleController.hello]
Exiting method [SampleController.hello], return value: Hello, World!
```

这个示例展示了如何使用 Spring AOP 实现简单的日志记录。你可以根据实际需求调整日志记录级别、格式和内容，以及扩展切入点表达式以覆盖更多的方法。

4、实现数据校验

要使用 Spring AOP 实现数据校验，你可以创建一个切面类，对方法的输入参数进行校验。这里，我们将使用 JSR 380 (Java Bean Validation 2.0) 规范实现数据校验。首先，确保你的项目中包含以下依赖：

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>

```

然后，创建一个自定义注解 `validate`，用于标记需要进行数据校验的方法：

```

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Validate {
}

```

接下来，我们创建一个切面类（Aspect），实现数据校验逻辑：

```

@Aspect
@Component
public class ValidationAspect {
    @Autowired
    private Validator validator;

    @Pointcut("@annotation(Validate)")
    public void validationPointcut() {
    }

    @Before("validationPointcut()")
    public void validateMethodArguments(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        for (Object arg : args) {
            Errors errors = new BeanPropertyBindingResult(arg,
                arg.getClass().getName());
            validator.validate(arg, errors);

            if (errors.hasErrors()) {
                throw new ValidationException("Validation failed: " + errors.toString());
            }
        }
    }
}

```

```
}
```

在这个切面类中，我们定义了一个名为 `validationPointcut` 的切入点，用于匹配带有 `@Validate` 注解的方法。我们使用 `@Before` 注解标注 `validateMethodArguments` 方法，在方法调用前进行数据校验。如果校验失败，我们将抛出一个 `validationException` 异常。

最后，我们需要在需要进行数据校验的方法上添加 `@validate` 注解。例如，以下是一个简单的服务类：

```
@Service
public class DemoService {
    @Validate
    public void processData(@Valid DataModel data) {
        // Process data
    }
}

public class DataModel {
    @NotNull
    private String field1;

    @Size(min = 5, max = 10)
    private String field2;

    // Getters and setters
}
```

在这个示例中，我们为 `processData` 方法添加了 `@valid` 注解，以便在方法调用时自动进行数据校验。如果校验失败，切面类将抛出一个异常，从而阻止方法的执行。

这个示例展示了如何使用 Spring AOP 实现数据校验。你可以根据实际需求调整校验规则，以及自定义异常处理逻辑。

第二章 装饰器模式

一、实现原理

装饰器设计模式 (Decorator) 是一种**结构型设计模式**，它允许**动态地为对象添加新的行为**。它通过创建一个包装器来实现，即**将对象放入一个装饰器类中**，再将装饰器类放入另一个装饰器类中，以此类推，形成一条**包装链**。这样，我们可以在**不改变原有对象的情况下**，**动态地添加新的行为或修改原有行为**。

在 Java 中，实现装饰器设计模式的步骤如下：

- 1、定义一个接口或抽象类，作为被装饰对象的基类。

```
public interface Component {  
    void operation();  
}
```

在这个示例中，我们定义了一个名为 `Component` 的接口，它包含一个名为 `operation` 的抽象方法，用于定义被装饰对象的基本行为。

- 2、定义一个具体的被装饰对象，实现基类中的方法。

```
public class ConcreteComponent implements Component {  
    @Override  
    public void operation() {  
        System.out.println("ConcreteComponent is doing something...");  
    }  
}
```

在这个示例中，我们定义了一个名为 `ConcreteComponent` 的具体实现类，实现了 `Component` 接口中的 `operation` 方法。

- 3、定义一个抽象装饰器类，继承基类，并将被装饰对象作为属性。

```
public abstract class Decorator implements Component {

    protected Component component;

    public Decorator(Component component) {
        this.component = component;
    }

    @Override
    public void operation() {
        component.operation();
    }
}
```

在这个示例中，我们定义了一个名为 `Decorator` 的抽象类，继承了 `Component` 接口，并将被装饰对象作为属性。在 `operation` 方法中，我们调用被装饰对象的同名方法。

4、定义具体的装饰器类，继承抽象装饰器类，并实现增强逻辑。

```
public class ConcreteDecoratorA extends Decorator {

    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    @Override
    public void operation() {
        super.operation();
        System.out.println("ConcreteDecoratorA is adding new behavior...");
    }
}
```

在这个示例中，我们定义了一个名为 `ConcreteDecoratorA` 的具体装饰器类，继承了 `Decorator` 抽象类，并实现了 `operation` 方法的增强逻辑。在 `operation` 方法中，我们先调用被装饰对象的同名方法，然后添加新的行为。

5、使用装饰器增强被装饰对象。

```
public class Main {  
    public static void main(String[] args) {  
        Component component = new ConcreteComponent();  
        component = new ConcreteDecoratorA(component);  
        component.operation();  
    }  
}
```

在这个示例中，我们先创建了一个被装饰对象 `ConcreteComponent`，然后通过 `ConcreteDecoratorA` 类创建了一个装饰器，并将被装饰对象作为参数传入。最后，调用装饰器的 `operation` 方法，这样就可以实现对被装饰对象的增强。

有同学可能已经发现了，如上的写法和预期实现的功能和我们上边讲的静态代理设计模式非常像，这里就要解释一下了，虽然装饰器模式和静态代理模式有一些相似之处，但它们之间还是有区别的：

代理模式的目的是为了**控制对对象的访问**，它在对象的外部提供一个代理对象来控制对原始对象的访问。代理对象和原始对象通常实现同一个接口或继承同一个类，以保证二者可以互相替代。

装饰器模式的目的是为了**动态地增强对象的功能**，它在对象的内部通过一种包装器的方式来实现。装饰器模式中，**装饰器类和被装饰对象通常实现同一个接口或继承同一个类，以保证二者可以互相替代**。装饰器模式也被称为包装器模式。

因此，虽然装饰器模式和代理模式的代码实现上有些相似，但它们的目的和实现方式是不同的。代理模式的主要目的是**控制对对象的访问**，所以我们可以在代理对象中构建被代理者，装饰器模式的主要目的是**动态地增强对象的功能**。

需要注意的是，装饰器模式虽然可以实现动态地为对象增加行为，但是会增加系统的复杂性，因此在使用时需要仔细权衡利弊。

二、使用场景

在 Java 中，装饰器模式的应用非常广泛，特别是在 I/O 操作中。Java 中的 I/O 类库就是使用装饰器模式来实现不同的数据流之间的转换和增强的。

1、从IO库的设计理解装饰器

在初学 Java 的时候，曾经对 Java IO 的一些用法产生过很大疑惑，比如下面这样一段代码。我们打开文件 test.txt，从中读取数据。其中，InputStream 是一个抽象类，FileInputStream 是专门用来读取文件流的子类。BufferedInputStream 是一个支持带缓存功能的数据读取类，可以提高数据读取的效率，具体的代码如下：

```
InputStream in = new FileInputStream("D:/test.txt");
InputStream bin = new BufferedInputStream(in);
byte[] data = new byte[128];
while (bin.read(data) != -1) {
    //...
}
```

初看上面的代码，我们会觉得 Java IO 的用法比较麻烦，需要先创建一个 FileInputStream 对象，然后再传递给 BufferedInputStream 对象来使用。我在想，Java IO 为什么不设计一个继承 FileInputStream 并且支持缓存的 BufferedFileInputStream 类呢？这样我们就可以像下面的代码中这样，直接创建一个 BufferedFileInputStream 类对象，打开文件读取数据，用起来岂不是更加简单？

```
InputStream bin = new BufferedFileInputStream("/user/wangzheng/test.txt");
byte[] data = new byte[128];
while (bin.read(data) != -1) {
    //...
}
```

(1) 基于继承的设计方案

如果 InputStream 只有一个子类 FileInputStream 的话，那我们在 FileInputStream 基础之上，再设计一个孙子类 BufferedFileInputStream，也算是可以接受的，毕竟继承结构还算简单。但实际上，继承 InputStream 的子类有很多。我们需要给每一个 InputStream 的子类，再继续派生支持缓存读取的子类。

除了支持缓存读取之外，如果我们还需要对功能进行其他方面的增强，比如下面的 DataInputStream 类，支持按照基本数据类型（int、boolean、long 等）来读取数据。

```
FileInputStream in = new FileInputStream("/user/wangzheng/test.txt");
DataInputStream din = new DataInputStream(in);
int data = din.readInt();
```

在这种情况下，如果我们继续按照继承的方式来实现的话，就需要再继续派生出 `DataFileInputStream`、`DataPipedInputStream` 等类。如果我们还需要既支持缓存、又支持按照基本类型读取数据的类，那就要再继续派生出 `BufferedDataFileInputStream`、`BufferedDataPipedInputStream` 等 n 多类。这还只是附加了两个增强功能，如果我们需要附加更多的增强功能，那就会导致组合爆炸，类继承结构变得无比复杂，代码既不好扩展，也不好维护。这也是我们在第 10 节中讲的不推荐使用继承的原因。

(2) 基于装饰器模式的设计方案

在第 10 节中，我们还讲到“组合优于继承”，可以“使用组合来替代继承”。针对刚刚的继承结构过于复杂的问题，我们可以通过将继承关系改为组合关系来解决。下面的代码展示了 Java IO 的这种设计思路。不过，我对代码做了简化，只抽象出了必要的代码结构，如果你感兴趣的话，可以直接去查看 JDK 源码。

```
public abstract class InputStream {
    //...
    public int read(byte b[]) throws IOException {
        return read(b, 0, b.length);
    }

    public int read(byte b[], int off, int len) throws IOException {
        //...
    }

    public long skip(long n) throws IOException {
        //...
    }

    public int available() throws IOException {
        return 0;
    }

    public void close() throws IOException {}
    public synchronized void mark(int readlimit) {}

    public synchronized void reset() throws IOException {
        throw new IOException("mark/reset not supported");
    }

    public boolean markSupported() {
        return false;
    }
}
```

```

}
public class BufferedInputStream extends InputStream {
    protected volatile InputStream in;
    protected BufferedInputStream(InputStream in) {
        this.in = in;
    }

    //...实现基于缓存的读数据接口...
}
public class DataInputStream extends InputStream {
    protected volatile InputStream in;
    protected DataInputStream(InputStream in) {
        this.in = in;
    }

    //...实现读取基本类型数据的接口
}

```

看了上面的代码，你可能会问，那装饰器模式就是简单的“用组合替代继承”吗？当然不是。从 Java IO 的设计来看，装饰器模式相对于简单的组合关系，还有两个比较特殊的地方。

第一个比较特殊的地方是：装饰器类和原始类继承同样的父类，这样我们可以对原始类“嵌套”多个装饰器类。比如，下面这样一段代码，我们对 `FileInputStream` 嵌套了两个装饰器类：`BufferedInputStream` 和 `DataInputStream`，让它既支持缓存读取，又支持按照基本数据类型来读取数据。

```

InputStream in = new FileInputStream("/user/wangzheng/test.txt");
InputStream bin = new BufferedInputStream(in);
DataInputStream din = new DataInputStream(bin);
int data = din.readInt();

```

第二个比较特殊的地方是：装饰器类是对功能的增强，这也是装饰器模式应用场景的一个重要特点。实际上，符合“组合关系”这种代码结构的设计模式有很多，比如之前讲过的代理模式，还有现在的装饰器模式。尽管它们的代码结构很相似，但是每种设计模式的意图是不同的。就拿比较相似的代理模式和装饰器模式来说吧，代理模式中，代理类附加的是跟原始类无关的功能，而在装饰器模式中，装饰器类附加的是跟原始类相关的增强功能。

```

// 代理模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
    void f();
}

```

```

}
public class A implements IA {
    public void f() { //... }
}
public class AProxy implements IA {
    private IA a;
    public AProxy(IA a) {
        this.a = a;
    }

    public void f() {
        // 新添加的代理逻辑
        a.f();
        // 新添加的代理逻辑
    }
}
// 装饰器模式的代码结构(下面的接口也可以替换成抽象类)
public interface IA {
    void f();
}
public class A implements IA {
    public void f() { //... }
}
public class ADecorator implements IA {
    private IA a;
    public ADecorator(IA a) {
        this.a = a;
    }

    public void f() {
        // 功能增强代码
        a.f();
        // 功能增强代码
    }
}

```

实际上，如果去查看 JDK 的源码，你会发现，BufferedInputStream、DataInputStream 并非继承自 InputStream，而是另外一个叫 FilterInputStream 的类。那这又是出于什么样的设计意图，才引入这样一个类呢？

我们再重新来看一下 `BufferedInputStream` 类的代码。`InputStream` 是一个抽象类而非接口，而且它的大部分函数（比如 `read()`、`available()`）都有默认实现，按理来说，我们只需要在 `BufferedInputStream` 类中重新实现那些需要增加缓存功能的函数就可以了，其他函数继承 `InputStream` 的默认实现。但实际上，这样做是行不通的。

对于即便是不需要增加缓存功能的函数来说，`BufferedInputStream` 还是必须把它重新实现一遍，简单包裹对 `InputStream` 对象的函数调用。具体的代码示例如下所示。如果不重新实现，那 `BufferedInputStream` 类就无法将最终读取数据的任务，委托给传递进来的 `InputStream` 对象来完成。这一部分稍微有点不好理解，你自己多思考一下。

```
public class BufferedInputStream extends InputStream {
    protected volatile InputStream in;
    protected BufferedInputStream(InputStream in) {
        this.in = in;
    }

    // f()函数不需要增强，只是重新调用一下InputStream in对象的f()
    public void f() {
        in.f();
    }
}
```

实际上，`DataInputStream` 也存在跟 `BufferedInputStream` 同样的问题。为了避免代码重复，Java IO 抽象出了一个装饰器父类 `FilterInputStream`，代码实现如下所示。`InputStream` 的所有的装饰器类（`BufferedInputStream`、`DataInputStream`）都继承自这个装饰器父类。这样，装饰器类只需要实现它需要增强的方法就可以了，其他方法继承装饰器父类的默认实现。

```
public class FilterInputStream extends InputStream {
    protected volatile InputStream in;
    protected FilterInputStream(InputStream in) {
        this.in = in;
    }

    public int read() throws IOException {
        return in.read();
    }

    public int read(byte b[] throws IOException {
        return read(b, 0, b.length);
    }
}
```

```

public int read(byte b[], int off, int len) throws IOException {
    return in.read(b, off, len);
}
public long skip(long n) throws IOException {
    return in.skip(n);
}
public int available() throws IOException {
    return in.available();
}
public void close() throws IOException {
    in.close();
}
public synchronized void mark(int readlimit) {
    in.mark(readlimit);
}
public synchronized void reset() throws IOException {
    in.reset();
}
public boolean markSupported() {
    return in.markSupported();
}
}

```

2、mybatis的缓存设计

创建缓存的过程：

```

public Cache useNewCache(Class<? extends Cache> typeClass,
    Class<? extends Cache> evictionClass,
    Long flushInterval,
    Integer size,
    boolean readWrite,
    boolean blocking,
    Properties props) {
    Cache cache = new CacheBuilder(currentNamespace)
        // 根据类型生成实例，并进行配置
        .implementation(valueOrDefault(typeClass, PerpetualCache.class))
        .addDecorator(valueOrDefault(evictionClass, LruCache.class))
        .clearInterval(flushInterval)

```

```

        .size(size)
        .readWrite(readWrite)
        .blocking(blocking)
        .properties(props)
        .build();
configuration.addCache(cache);
currentCache = cache;
return cache;
}

```

默认的缓存如下，本质就是维护了一个简单的HashMap：

```

public class PerpetualCache implements Cache {

    private final String id;

    private final Map<Object, Object> cache = new HashMap<>();

    public PerpetualCache(String id) {
        this.id = id;
    }

    @Override
    public void putObject(Object key, Object value) {
        cache.put(key, value);
    }

    @Override
    public Object getObject(Object key) {
        return cache.get(key);
    }
    // ...省略其他的简单的方法

}

```

缓存的实现当然不可能这么简单，事实上他的构造过程如下：

```

public Cache build() {
    // 设置默认的cache实现，并绑定默认的淘汰策略
    setDefaultImplementations();
    // 利用反射创建实例
}

```

```

Cache cache = newBaseCacheInstance(implementation, id);
// 设置properties属性
setCacheProperties(cache);
// 不应用装饰自定义缓存，自定义缓存需要自己实现对应的特性，如淘汰策略等
// 通常情况自定义缓存有自己的独立配置，如redis、ehcache
if (PerpetualCache.class.equals(cache.getClass())) {
    for (Class<? extends Cache> decorator : decorators) {
        cache = newCacheDecoratorInstance(decorator, cache);
        setCacheProperties(cache);
    }
    // 这是标准的装饰器，这里使用了装饰器设计模式
    cache = setStandardDecorators(cache);
} else if (!LoggingCache.class.isAssignableFrom(cache.getClass())) {
    cache = new LoggingCache(cache);
}
return cache;
}

```

mybatis会使用装饰者设计模式，对默认cache进行装饰，使其具有LRU的能力，如下：

```

private void setDefaultImplementations() {
    if (implementation == null) {
        implementation = PerpetualCache.class;
        if (decorators.isEmpty()) {
            decorators.add(LruCache.class);
        }
    }
}
}

```

LruCache实现如下，默认情况下的LRU算法实现是基于LinkedHashMap实现的：

```

public class LruCache implements Cache {

    // 代理目标缓存
    private final Cache delegate;
    private Map<Object, Object> keyMap;
    private Object eldestKey;
}

```


// LruCache用来装饰默认的缓存，这里实现了缓存的高级特性

```
public LruCache(Cache delegate) {  
    this.delegate = delegate;  
    setSize(1024);  
}
```

@Override

```
public String getId() {  
    return delegate.getId();  
}
```

@Override

```
public int getSize() {  
    return delegate.getSize();  
}
```

// 设置长度，构建一个LinkedHashMap，重写removeEldestEntry

```
public void setSize(final int size) {
```

// 第三个参数accessOrder为true，可以使LinkedHashMap维护一个【访问顺序】

// 最近被访问的数据会被放在链表的尾部，天然实现lru

```
keyMap = new LinkedHashMap<Object, Object>(size, .75F, true) {  
    private static final long serialVersionUID = 4267176411845948333L;
```

// 重写该方法，父类直接返回false

// 只要实际容量size() 大于 初始化容量 size 认定当前的缓存已经满了

// 该方法会在LinkedHashMap的afterNodeInsertion方法中被主动调用

// 会将头节点当作eldest删除

@Override

```
protected boolean removeEldestEntry(Map.Entry<Object, Object> eldest)  
{  
  
    boolean tooBig = size() > size;  
    if (tooBig) {  
        // 同时将这个key复制给成员变量eldestKey  
        eldestKey = eldest.getKey();  
    }  
    return tooBig;  
}  
};  
}
```

// put一个缓存的过程

// 放入当前的缓存值，淘汰eldestKey

@Override

```
public void putObject(Object key, Object value) {  
    delegate.putObject(key, value);  
    cycleKeyList(key);  
}
```

// get一个缓存的过程

// 获得该值，同时提升key热度，主动访问一下keyMap.get(key)

@Override

```
public Object getObject(Object key) {  
    keyMap.get(key);  
    return delegate.getObject(key);  
}
```

@Override

```
public Object removeObject(Object key) {  
    return delegate.removeObject(key);  
}
```

@Override

```
public void clear() {  
    delegate.clear();  
    keyMap.clear();  
}
```

// 循环key的集合

```
private void cycleKeyList(Object key) {  
    keyMap.put(key, key);  
    if (eldestKey != null) {  
        delegate.removeObject(eldestKey);  
        eldestKey = null;  
    }  
}
```

```
}
```

最后使用其他的装饰器对cache进行装饰，使其就有更多的能力：

```
private Cache setStandardDecorators(Cache cache) {  
    try {
```

```

    MetaObject metaCache = SystemMetaObject.forObject(cache);
    // 设置大小, 默认1024
    if (size != null && metaCache.hasSetter("size")) {
        metaCache.setValue("size", size);
    }
    if (clearInterval != null) {
        cache = new ScheduledCache(cache);
        ((ScheduledCache) cache).setClearInterval(clearInterval);
    }
    if (readWrite) {
        cache = new SerializedCache(cache);
    }
    cache = new LoggingCache(cache);
    cache = new SynchronizedCache(cache);
    if (blocking) {
        cache = new BlockingCache(cache);
    }
    return cache;
} catch (Exception e) {
    throw new CacheException("Error building standard cache decorators.
Cause: " + e, e);
}
}

```

我们选取几个核心的装饰器的核心方法：

1、ScheduledCache

```

private boolean clearWhenStale() {
    if (System.currentTimeMillis() - lastClear > clearInterval) {
        clear();
        return true;
    }
    return false;
}

@Override
public void clear() {
    lastClear = System.currentTimeMillis();
    delegate.clear();
}

```

2、SerializedCache

readWrite配置为true时，缓存的实例可以进行读写操作，核心就是序列化和反序列化：

```
@Override
public void putObject(Object key, Object object) {
    if (object == null || object instanceof Serializable) {
        // 存储时将对象序列化为byte数组
        delegate.putObject(key, serialize((Serializable) object));
    } else {
        throw new CacheException("SharedCache failed to make a copy of a non-serializable object: " + object);
    }
}

@Override
public Object getObject(Object key) {
    Object object = delegate.getObject(key);
    // 获取时将对象反序列化为对象
    return object == null ? null : deserialize((byte[]) object);
}
```

3、LoggingCache

该日志缓存可以使用日志输出缓存的命中率：

```
public class LoggingCache implements Cache {

    private final Log log;
    private final Cache delegate;
    protected int requests = 0;
    protected int hits = 0;

    public LoggingCache(Cache delegate) {
        this.delegate = delegate;
        this.log = LogFactory.getLog(getId());
    }

    // 获取值的时候会统计请求次数，命中率
}
```

```

@Override
public Object getObject(Object key) {
    requests++;
    final Object value = delegate.getObject(key);
    // 命中了, 命中的次数+1
    if (value != null) {
        hits++;
    }
    // 使用日志输出 某一个 缓存的命中率
    if (log.isDebugEnabled()) {
        log.debug("Cache Hit Ratio [" + getId() + "]: " + getHitRatio());
    }
    return value;
}

private double getHitRatio() {
    return (double) hits / (double) requests;
}

// ...省略其他

}

```

4、SynchronizedCache

使用此包装器可以将所有方法加synchronized，保证线程安全：

```

public class SynchronizedCache implements Cache {

    @Override
    public synchronized void putObject(Object key, Object object) {
        delegate.putObject(key, object);
    }

    @Override
    public synchronized Object getObject(Object key) {
        return delegate.getObject(key);
    }

    @Override
    public synchronized Object removeObject(Object key) {
        return delegate.removeObject(key);
    }
}

```

```
}
```

5、阻塞式缓存的实现如下

阻塞式的缓存和阻塞队列不同，阻塞队列的特点是：

- 1、调用enqueue入队方法时，如果队列已满，则阻塞等待。
- 2、调用dequeue出队方法时，如果队列为空，则阻塞等待。

而我们阻塞式的缓存特点是：

- 1、调用get方法获取同一个key时，需要进行阻塞等待，特别是第一个线程需要去db查询数据的时候

阻塞式缓存装饰器使用了CountDownLatch实现了get方法的阻塞，CountDownLatch的使用方法简单回顾一下：

```
CountDownLatch latch = new CountDownLatch(10);
for(int i = 0; i<10; i++){
    new Thread(() -> latch.countDown()).start();
}
latch.await();
```

BlockingCache实现如下：

```
public class BlockingCache implements Cache {

    // 超时时间
    private long timeout;
    private final Cache delegate;
    private final ConcurrentHashMap<Object, CountDownLatch> locks;

    public BlockingCache(Cache delegate) {
        this.delegate = delegate;
        this.locks = new ConcurrentHashMap<>();
    }

    @Override
    public Object getObject(Object key) {
        // 获取锁，让其等待
        acquireLock(key);
```

```

Object value = delegate.getObject(key);
if (value != null) {
    // 如果获取了对应的value，解锁
    releaseLock(key);
}
return value;
}

private void acquireLock(Object key) {
    // 创建一个CountDownLatch，步长为1
    CountDownLatch newLatch = new CountDownLatch(1);
    while (true) {
        // 如果所指定的 key 已经在 Map 中存在，返回和这个 key 值对应的 value，
        // 如果所指定的 key 不在 Map 中存在，则返回 null。
        CountDownLatch latch = locks.putIfAbsent(key, newLatch);
        // 如果之前不存在，说明是第一次访问
        if (latch == null) {
            break;
        }
        // 说明该缓存不是第一次命中，前边有人正在访问
        try {
            if (timeout > 0) {
                // 等待正在访问的线程调用countdown方法
                boolean acquired = latch.await(timeout, TimeUnit.MILLISECONDS);
                if (!acquired) {
                    throw new CacheException("Couldn't get a lock in " + timeout + "
for the key " + key + " at the cache " + delegate.getId());
                }
            } else {
                latch.await();
            }
        } catch (InterruptedException e) {
            throw new CacheException("Got interrupted while trying to acquire lock
for key " + key, e);
        }
    }
}

private void releaseLock(Object key) {
    // 移除这个CountDownLatch
    CountDownLatch latch = locks.remove(key);
    if (latch == null) {
        // 检测到试图释放未获得的锁。这绝不应该发生。
    }
}

```

```

        throw new IllegalStateException("Detected an attempt at releasing
unacquired lock. This should never happen.");
    }
    latch.countDown();
}

// 添加缓存，此方法是缓存中查询不到，去DB查询结构后主动调用
@Override
public void putObject(Object key, Object value) {
    try {
        delegate.putObject(key, value);
    } finally {
        releaseLock(key);
    }
}

// ...
}

```

装饰器模式主要解决继承关系过于复杂的问题，通常是通过组合来替代继承。它主要的作用是给原始类添加增强功能。这也是判断是否该用装饰器模式的一个重要的依据。除此之外，装饰器模式还有一个特点，那就是可以对原始类嵌套使用多个装饰器。为了满足这个应用场景，在设计的时候，装饰器类需要跟原始类继承相同的抽象类或者接口。

Q: 在学习代理设计模式的时候，我们通过装饰者模式给 `InputStream` 添加缓存读取数据功能。那对于“添加缓存”这个应用场景来说，我们到底是该用代理模式还是装饰器模式呢？你怎么看待这个问题？

事实上对于绝大多数的“添加缓存”的业务场景，核心目的主要就是想增强对象的功能（即增加缓存功能），而并不是控制对对象的访问，所以装饰器模式可能会更合适。但是大家要记住设计模式不是教条主义，比如我们想强制对持久层增加一个本地缓存，代理设计模式也是很好的选择。

第三章 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，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！

一、抛砖引玉

首先，我们可以通过 JDBC 驱动的例子来看一下这个设计模式，**JDBC 驱动是桥接模式的经典应用**。

当然我们需要复习一下利用 JDBC 驱动来查询数据库的方法。获取一个连接的具体的代码如下所示，我们：

// 1.数据库连接的4个基本要素：

```
String url = "jdbc:mysql://127.0.0.1:3306/ydlclass?
useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia
/Shanghai";
```

```
String user = "root";
```

```
String password = "root";
```

// 8.0之后名字改了 com.mysql.cj.jdbc.Driver

```
String driverName = "com.mysql.cj.jdbc.Driver";
```

// 2.实例化Driver,可省略

// 3.注册驱动,可省略

// 4.获取连接

```
conn = DriverManager.getConnection(url, user, password);
```

如果我们想要把 MySQL 数据库换成 Oracle 数据库，只需要把驱动 `com.mysql.cj.jdbc.Driver` 换成 `oracle.jdbc.driver.OracleDriver`，同时将url进行替换就可以了。

当然事实上我们在工作中会将**上边的内容配置在配置文件**当中，当发生改变的时候，我们仅仅需要修改配置文件即可，完全不需要修改一行代码。

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.cj.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。

```

public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    //
    // Register ourselves with the DriverManager
    //
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }

    /**
     * Construct a new driver and register it with DriverManager
     * @throws SQLException if a database error occurs.
     */
    public Driver() throws SQLException {
        // Required for Class.forName().newInstance()
    }
}

```

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.cj.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。

```

public class DriverManager {

    // List of registered JDBC drivers
    private static final CopyOnWriteArrayList<DriverInfo> registeredDrivers = new
CopyOnWriteArrayList<>();

```

```

public static void registerDriver(java.sql.Driver driver,
                                DriverAction da)
    throws SQLException {

    /* Register the driver if it has not already been added to our list */
    if (driver != null) {
        registeredDrivers.addIfAbsent(new DriverInfo(driver, da));
    } else {
        // This is for compatibility with the original DriverManager
        throw new NullPointerException();
    }

    println("registerDriver: " + driver);

}

private static Connection getConnection(
    String url, java.util.Properties info, Class<?> caller) throws SQLException {
    /*
    * When callerCl is null, we should check the application's
    * (which is invoking this class indirectly)
    * classloader, so that the JDBC driver class outside rt.jar
    * can be loaded from here.
    */
    ClassLoader callerCL = caller != null ? caller.getClassLoader() : null;
    if (callerCL == null || callerCL == ClassLoader.getPlatformClassLoader()) {
        callerCL = Thread.currentThread().getContextClassLoader();
    }

    if (url == null) {
        throw new SQLException("The url cannot be null", "08001");
    }

    println("DriverManager.getConnection(\"" + url + "\");");

    ensureDriversInitialized();

    // Walk through the loaded registeredDrivers attempting to make a
connection.
    // Remember the first exception that gets raised so we can reraise it.
    SQLException reason = null;

    for (DriverInfo aDriver : registeredDrivers) {

```

```

// If the caller does not have permission to load the driver then
// skip it.
if (isDriverAllowed(aDriver.driver, callerCL)) {
    try {
        println("  trying " + aDriver.driver.getClass().getName());
        Connection con = aDriver.driver.connect(url, info);
        if (con != null) {
            // Success!
            println("getConnection returning " +
aDriver.driver.getClass().getName());
            return (con);
        }
    } catch (SQLException ex) {
        if (reason == null) {
            reason = ex;
        }
    }

} else {
    println("  skipping: " + aDriver.driver.getClass().getName());
}

}

// if we got here nobody could connect.
if (reason != null) {
    println("getConnection failed: " + reason);
    throw reason;
}

println("getConnection: no suitable driver found for "+ url);
throw new SQLException("No suitable driver found for "+ url, "08001");
}

}

```

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.cj.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。

这就是JDBC中使用桥接模式的一个例子。通过将**抽象的JDBC接口与具体数据库厂商的实现相分离**，我们可以实现对抽象和实现的独立扩展。这种设计模式使得在**不修改客户端代码的情况下更换数据库驱动程序成为可能，从而提高了代码的可维护性和可扩展性**。

二、桥接模式的原理解析

桥接模式（Bridge Pattern）是一种结构型设计模式，用于将**抽象与其实现相分离**，以便两者可以独立地进行变化。它通过将抽象的类与实现的类分离，实现了抽象和实现的解耦。

桥接模式的核心概念：

1. **抽象化**：在抽象类中定义抽象业务的接口和一个对实现层次结构的引用。抽象化的主要目的是**隐藏实现的细节，以便可以在不影响客户端的情况下更改实现**。
2. **实现化**：这是一个接口，定义了实现抽象化的方法。不同的具体实现类可以有不同的实现方式。
3. **扩展抽象化**：这是抽象化的一个具体实现，它定义了抽象业务的具体操作。
4. **具体实现化**：实现化接口的具体实现类。这些类为抽象业务提供具体的实现。

优点：

- 抽象与实现分离，可以独立地进行变化。
- 提高了可扩展性。
- 符合单一职责原则，抽象部分专注于抽象化，实现部分专注于具体实现。

缺点：

- 增加了系统的复杂性。

首先我们定义具体实现化类的相关api，这里我们可以理解为具体的mysql的实现：

```
// 可以当做具体的jdbc实现
```

```

interface Implementor {
    void operationImpl();
}

// 具体实现化类A, 具体实现可以多样化, 独立变化
class ConcretImplementorA implements Implementor {
    @Override
    public void operationImpl() {
        System.out.println("具体实现A");
    }
}

// 具体实现化类B
class ConcretImplementorB implements Implementor {
    @Override
    public void operationImpl() {
        System.out.println("具体实现B");
    }
}

```

接下来编写抽象化类的相关内容，可以理解为jdbc提供的api接口：

```

abstract class Abstraction {
    protected Implementor implementor;

    // 抽象依赖实现的接口, 而不依赖实现本身
    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    abstract void operation();
}

// 抽象化类
class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }

    // 请记住这里的方法名字不同, 他们不需要实现相同的接口, 他可以独立变化
    @Override
    void operation() {

```

```
        implementor.operationImpl();
    }
}
```

这里展示桥接的过程：

```
public class BridgePatternDemo {
    public static void main(String[] args) {
        Implementor implementorA = new ConcreteImplementorA();
        Abstraction abstractionA = new RefinedAbstraction(implementorA);
        abstractionA.operation();

        Implementor implementorB = new ConcreteImplementorB();
        Abstraction abstractionB = new RefinedAbstraction(implementorB);
        abstractionB.operation();
    }
}
```

在这个例子中，我们有一个**抽象化类** `Abstraction` 和一个**实现化接口** `Implementor`。通过将抽象化类与实现化接口分离，我们可以在**不影响客户端的情况下更改实现**。具体来说，**我们可以轻松地地为 `Implementor` 添加新的具体实现类，同时在 `Abstraction` 类的子类中使用这些新的实现**。

在上述示例中，我们创建了两个具体实现化类 `ConcreteImplementorA` 和 `ConcreteImplementorB`，它们分别实现了 `Implementor` 接口。然后，我们创建了一个扩展抽象化类 `RefinedAbstraction`，它扩展了抽象化类 `Abstraction` 并实现了 `operation` 方法。在 `BridgePatternDemo` 的 `main` 方法中，我们分别使用了这两个具体实现类来创建 `Abstraction` 类的实例，并调用了它们的 `operation` 方法。

这个例子展示了桥接模式的核心概念，即**将抽象与实现分离**。这种分离使得我们可以**独立地改变抽象类和实现类**，提高了代码的**可扩展性和可维护性**。同时，这种设计也符合了单一职责原则，使得抽象部分和实现部分**各自关注自己的核心职责**。

我们在看了这个设计模式之后，发现和装饰器、代理设计模式的代码长得非常相似，于是我只能求教 chatgpt，和他一起探讨一下这个问题；



确实，桥接模式和装饰者模式在某些方面有一定的相似性，但它们解决的问题和目的是不同的。让我们来对比一下这两种设计模式：

确实，桥接模式和装饰者模式在某些方面有一定的相似性，但它们解决的问题和目的是不同的。让我们来对比一下这两种设计模式：

相似性：

- 两者都是结构型设计模式。
- 两者都使用了组合来解决问题。

区别：

1. 目的：装饰者模式主要用于**在不修改原始类代码的情况下向对象添加新功能**，而桥接模式主要用于**将抽象与实现分离，以便两者可以独立地进行变化**。
2. 结构：装饰者模式中，装饰者类和原始类通常实现相同的接口，而桥接模式中，抽象类和实现类通常是分属不同的接口。
3. 扩展性：装饰者模式可以动态地为对象添加新功能，而桥接模式则关注于抽象和实现的独立扩展。

尽管桥接模式和装饰者模式在某些方面有相似之处，但它们解决的问题和应用场景是不同的。在实际开发中，根据具体问题选择合适的设计模式非常重要。

三、桥接模式的应用举例

1、消息通知

接下来，我们来看一个消息通知系统的例子，这个系统需要支持多种通知方式（例如邮件、短信、即时消息等）以及多种通知紧急程度（普通、紧急、非常紧急等）。

这里，我们可以使用桥接模式将**通知方式**和**通知紧急程度**分离，使得它们可以独立地进行变化和扩展。

// 通知方式接口（实现化角色）

```
interface MessageSender {  
    void send(String message);  
}
```


// 邮件通知实现类

```
class EmailSender implements MessageSender {  
    @Override  
    public void send(String message) {  
        System.out.println("发送邮件通知: " + message);  
    }  
}
```

// 短信通知实现类

```
class SmsSender implements MessageSender {  
    @Override  
    public void send(String message) {  
        System.out.println("发送短信通知: " + message);  
    }  
}
```

// 即时消息通知实现类

```
class InstantMessageSender implements MessageSender {  
    @Override  
    public void send(String message) {  
        System.out.println("发送即时消息通知: " + message);  
    }  
}
```

// 抽象通知类（抽象化角色）

```
abstract class Notification {  
    protected MessageSender messageSender;  
  
    public Notification(MessageSender messageSender) {  
        this.messageSender = messageSender;  
    }  
  
    abstract void notify(String message);  
}
```

// 普通通知子类（扩展抽象化角色）

```
class NormalNotification extends Notification {  
    public NormalNotification(MessageSender messageSender) {  
        super(messageSender);  
    }  
  
    @Override
```

```
void notify(String message) {  
    messageSender.send("普通: " + message);  
}  
}
```

// 紧急通知子类 (扩展抽象化角色)

```
class UrgentNotification extends Notification {  
    public UrgentNotification(MessageSender messageSender) {  
        super(messageSender);  
    }  
  
    @Override  
    void notify(String message) {  
        messageSender.send("紧急: " + message);  
    }  
}
```

// 非常紧急通知子类 (扩展抽象化角色)

```
class CriticalNotification extends Notification {  
    public CriticalNotification(MessageSender messageSender) {  
        super(messageSender);  
    }  
  
    @Override  
    void notify(String message) {  
        messageSender.send("非常紧急: " + message);  
    }  
}
```

```
public class BridgePatternExample {  
    public static void main(String[] args) {  
        MessageSender emailSender = new EmailSender();  
        MessageSender smsSender = new SmsSender();  
        MessageSender instantMessageSender = new InstantMessageSender();  
  
        Notification normalEmailNotification = new  
NormalNotification(emailSender);  
        normalEmailNotification.notify("有一个新的任务待处理。");  
  
        Notification urgentSmsNotification = new UrgentNotification(smsSender);  
        urgentSmsNotification.notify("系统出现故障, 请尽快处理!");  
    }  
}
```

```
Notification criticalInstantMessageNotification = new
CriticalNotification(instantMessageSender);
    criticalInstantMessageNotification.notify("系统崩溃，请立即处理！");
}
}
```

在这个示例中，我们使用桥接模式将通知方式（`MessageSender` 接口及其实现类）和通知紧急程度（`Notification` 类及其子类）分离。这使得我们可以独立地添加更多的通知方式和通知紧急程度，而不会导致类的数量爆炸性增长。

以下是如何运行这个示例的步骤：

1. 我们定义了一个 `MessageSender` 接口，用于表示通知方式。然后，我们创建了几个实现了 `MessageSender` 接口的具体类：`EmailSender`、`SmsSender` 和 `InstantMessageSender`，它们分别表示通过邮件、短信和即时消息发送通知。
2. 接下来，我们定义了一个抽象类 `Notification`，它持有一个 `MessageSender` 对象。`Notification` 类有一个抽象方法 `notify`，用于发送通知。然后，我们创建了几个扩展 `Notification` 类的子类：`NormalNotification`、`UrgentNotification` 和 `CriticalNotification`，它们分别表示普通、紧急和非常紧急的通知。
3. 在 `BridgePatternExample` 的 `main` 方法中，我们创建了 `EmailSender`、`SmsSender` 和 `InstantMessageSender` 对象，并将它们与不同紧急程度的通知对象组合。然后，我们调用这些通知对象的 `notify` 方法来发送通知。

通过使用桥接模式，我们可以轻松地**为消息通知系统添加新的通知方式和通知紧急程度**，而无需修改现有的类结构。这种设计增强了代码的可扩展性和可维护性。

2、支付方式

当然，让我们来看一个电商项目中的例子。在这个例子中，我们需要处理**多种支付方式**（如信用卡、PayPal、支付宝等）和**多种折扣策略**（如VIP折扣、新用户折扣、优惠券等）。

我们可以**使用桥接模式将支付方式和折扣策略分离**，使得**它们可以独立地进行变化和扩展**。

```
interface PaymentMethod {
    void pay(double amount);
}
```

// 信用卡支付实现类

```
class CreditCardPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("使用信用卡支付: " + amount);
    }
}
```

// PayPal支付实现类

```
class PayPalPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("使用PayPal支付: " + amount);
    }
}
```

// 支付宝支付实现类

```
class AlipayPayment implements PaymentMethod {
    @Override
    public void pay(double amount) {
        System.out.println("使用支付宝支付: " + amount);
    }
}
```

// 折扣策略接口（抽象化角色）

```
abstract class DiscountStrategy {
    protected PaymentMethod paymentMethod;

    public DiscountStrategy(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    abstract double getDiscountedAmount(double originalAmount);

    public void payWithDiscount(double originalAmount) {
        double discountedAmount = getDiscountedAmount(originalAmount);
        paymentMethod.pay(discountedAmount);
    }
}
```

// VIP折扣策略子类（扩展抽象化角色）

```
class VipDiscountStrategy extends DiscountStrategy {
    public VipDiscountStrategy(PaymentMethod paymentMethod) {
        super(paymentMethod);
    }
}
```

```

    }

    @Override
    double getDiscountedAmount(double originalAmount) {
        return originalAmount * 0.9; // VIP用户享有9折优惠
    }
}

// 新用户折扣策略子类（扩展抽象化角色）
class NewUserDiscountStrategy extends DiscountStrategy {
    public NewUserDiscountStrategy(PaymentMethod paymentMethod) {
        super(paymentMethod);
    }

    @Override
    double getDiscountedAmount(double originalAmount) {
        return originalAmount * 0.95; // 新用户享有95折优惠
    }
}

public class ECommerceExample {
    public static void main(String[] args) {
        PaymentMethod creditCardPayment = new CreditCardPayment();
        PaymentMethod payPalPayment = new PayPalPayment();
        PaymentMethod alipayPayment = new AlipayPayment();

        DiscountStrategy vipCreditCardStrategy = new
        VipDiscountStrategy(creditCardPayment);
        vipCreditCardStrategy.payWithDiscount(100);

        DiscountStrategy newUserPayPalStrategy = new
        NewUserDiscountStrategy(payPalPayment);
        newUserPayPalStrategy.payWithDiscount(100);
    }
}

```

在这个示例中，我们使用桥接模式将支付方式（`PaymentMethod` 接口及其实现类）和折扣策略（`DiscountStrategy` 类及其子类）分离。这样，我们可以独立地添加更多的支付方式和折扣策略，而不会导致类的数量爆炸性增长。

以下是如何运行这个示例的步骤：

1. 我们定义了一个 `PaymentMethod` 接口，用于表示支付方式。然后，我们创建了几个实现了 `PaymentMethod` 接口的具体类：`CreditCardPayment`、`PayPalPayment` 和 `AlipayPayment`，它们分别表示通过信用卡、PayPal和支付宝进行支付。
2. 接下来，我们定义了一个抽象类 `DiscountStrategy`，它持有一个 `PaymentMethod` 对象。`DiscountStrategy` 类有一个抽象方法 `getDiscountedAmount`，用于计算折扣后的金额。然后，我们创建了几个扩展 `DiscountStrategy` 类的子类：`VipDiscountStrategy` 和 `NewUserDiscountStrategy`，它们分别表示VIP折扣和新用户折扣。
3. 在 `ECommerceExample` 的 `main` 方法中，我们创建了 `CreditCardPayment`、`PayPalPayment` 和 `AlipayPayment` 对象，并将它们与不同的折扣策略对象组合。然后，我们调用这些折扣策略对象的 `paywithDiscount` 方法来进行支付。

通过使用桥接模式，我们可以轻松地地为电商项目添加新的支付方式和折扣策略，而无需修改现有的类结构。这种设计增强了代码的可扩展性和可维护性。

3、crm中的使用场景

在CRM（客户关系管理）系统中，桥接模式可以应用于多个场景，帮助我们解决不同维度的问题，使得它们可以独立地进行变化和扩展。以下是一些使用桥接模式的场景示例：

1. **报表生成**：CRM系统通常需要生成各种报表，如销售报表、客户统计报表等。这些报表可能需要支持多种文件格式（如PDF、Excel、CSV等）和多种展示样式（如图表、表格等）。我们可以使用桥接模式将文件格式和展示样式分离，使得它们可以独立地进行变化和扩展。
2. **通知系统**：CRM系统可能需要发送多种类型的通知，如邮件、短信、即时消息等。同时，这些通知可能针对不同级别的客户群体有不同的发送策略。我们可以使用桥接模式将通知方式和发送策略分离，使得它们可以独立地进行变化和扩展。
3. **客户数据同步**：CRM系统可能需要与其他系统（如ERP、E-commerce等）进行客户数据同步。这些系统可能有多种接口类型（如REST API、SOAP、消息队列等）和多种数据格式（如XML、JSON、CSV等）。我们可以使用桥接模式将接口类型和数据格式分离，使得它们可以独立地进行变化和扩展。

这些只是桥接模式在CRM系统中的一些应用场景。实际上，在实际开发过程中，我们可以根据需要灵活地应用桥接模式来解决不同维度的问题。

第四章 适配器模式

一、原理与实现

适配器设计模式 (Adapter Design Pattern) 是一种结构型设计模式，用于**解决两个不兼容接口之间的问题**。适配器允许**将一个类的接口转换为客户端期望的另一个接口**，使得原本由于接口不兼容而不能一起工作的类可以一起工作。

在适配器设计模式中，主要包含以下四个角色：

1. **目标接口 (Target)**：这是客户端期望使用的接口，它定义了特定领域的操作和方法。
2. **需要适配的类 (Adaptee)**：这是一个已存在的类，它具有客户端需要的功能，但其接口与目标接口不兼容。适配器的目标是使这个类的功能能够通过目标接口使用。
3. **适配器 (Adapter)**：这是适配器模式的核心角色，它实现了目标接口并持有需要适配的类的一个实例。适配器通过封装Adaptee的功能，使其能够满足Target接口的要求。
4. **客户端 (Client)**：这是使用目标接口的类。客户端与目标接口进行交互，不直接与需要适配的类交互。通过使用适配器，客户端可以间接地使用需要适配的类的功能。

适配器模式的主要目的是在不修改现有代码的情况下，使不兼容的接口能够协同工作。通过引入适配器角色，客户端可以使用目标接口与需要适配的类进行通信，从而实现解耦和扩展性。

适配器模式有两种实现方式：类适配器和对象适配器。

类适配器

类适配器使用继承来实现适配器功能。适配器类继承了原有的类 (Adaptee) 并实现了目标接口 (Target)。

```
// 目标接口
interface Target {
    void request();
}

// 需要适配的类 (Adaptee)
class Adaptee {
    void specificRequest() {
        System.out.println("Adaptee's specific request");
    }
}
```

```

    }
}

// 类适配器
class ClassAdapter extends Adaptee implements Target {
    @Override
    public void request() {
        specificRequest();
    }
}

public class ClassAdapterExample {
    public static void main(String[] args) {
        Target target = new ClassAdapter();
        target.request();
    }
}

```

对象适配器

对象适配器使用组合来实现适配器功能。适配器类包含一个原有类的实例 (Adaptee) 并实现了目标接口 (Target) 。

```

// 目标接口
interface Target {
    void request();
}

// 需要适配的类 (Adaptee)
class Adaptee {
    void specificRequest() {
        System.out.println("Adaptee's specific request");
    }
}

// 对象适配器
class ObjectAdapter implements Target {
    private Adaptee adaptee;

    public ObjectAdapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }
}

```



```

    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}

public class ObjectAdapterExample {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Target target = new ObjectAdapter(adaptee);
        target.request();
    }
}

```

适配器模式可以用于解决不同系统、库或API之间的接口不兼容问题，使得它们可以协同工作。在实际开发中，应根据具体需求选择使用类适配器还是对象适配器。

二、应用场景总结

原理和实现讲完了，都不复杂。我们再来看，到底什么时候会用到适配器模式呢？

前面我们反复提到，适配器模式的应用场景是“**接口不兼容**”。那在实际的开发中，什么情况下才会出现接口不兼容呢？我建议你先自己思考一下这个问题，然后再来看我下面的总结。

1、封装有缺陷的接口设计

假设我们依赖的外部系统在接口设计方面有缺陷（比如包含大量静态方法），引入之后会影响到我们自身代码的可测试性。为了隔离设计上的缺陷，我们希望对外部系统提供的接口进行二次封装，抽象出更好的接口设计，这个时候就可以使用适配器模式了。

具体我还是举个例子来解释一下，你直接看代码应该会更清晰。具体代码如下所示：

```

public class Outer { //这个类来自外部sdk，我们无权修改它的代码
    //...
    public static void staticFunction1() { //...
    }
}

```

```

    public void uglyNamingFunction2() { //...
    }

    public void tooManyParamsFunction3(int paramA, int paramB, ...) { //...
    }

    public void lowPerformanceFunction4() { //...
    }

}

```

// 使用适配器模式进行重构

```

public class ITarget {
    void function1();
    void function2();
    void fucntion3(ParamsWrapperDefinition paramsWrapper);
    void function4();
    //...
}

```

// 注意：适配器类的命名不一定非得末尾带Adaptor

```

public class OuterAdaptor extends CD implements ITarget {
    //...

    public void function1() {
        super.staticFunction1();
    }

    public void function2() {
        super.uglyNamingFucntion2();
    }

    public void function3(ParamsWrapperDefinition paramsWrapper) {
        super.tooManyParamsFunction3(paramsWrapper.getParamA(), ...);
    }

    public void function4() {
        //...reimplement it...
    }

}

```

2、统一多个类的接口设计

某个功能的实现依赖多个外部系统（或者说类）。通过适配器模式，将它们的接口适配为统一的接口定义，然后我们就可以使用多态的特性来复用代码逻辑。具体我还是举个例子来解释一下。

假设我们的系统要对用户输入的文本内容做敏感词过滤，为了提高过滤的召回率，我们引入了多款第三方敏感词过滤系统，依次对用户输入的内容进行过滤，过滤掉尽可能多的敏感词。但是，每个系统提供的过滤接口都是不同的。这就意味着我们没法复用一套逻辑来调用各个系统。这个时候，我们就可以使用适配器模式，将所有系统的接口适配为统一的接口定义，这样我们可以复用调用敏感词过滤的代码。

你可以配合着下面的代码示例，来理解我刚才举的这个例子。

```
public class ASensitiveWordsFilter { // A敏感词过滤系统提供的接口
    //text是原始文本，函数输出用***替换敏感词之后的文本
    public String filterSexyWords(String text) {
        // ...
    }

    public String filterPoliticalWords(String text) {
        // ...
    }
}

public class BSensitiveWordsFilter { // B敏感词过滤系统提供的接口
    public String filter(String text) {
        //...
    }
}

public class CSensitiveWordsFilter { // C敏感词过滤系统提供的接口
    public String filter(String text, String mask) {
        //...
    }
}

// 未使用适配器模式之前的代码：代码的可测试性、扩展性不好
public class RiskManagement {
    private ASensitiveWordsFilter aFilter = new ASensitiveWordsFilter();
    private BSensitiveWordsFilter bFilter = new BSensitiveWordsFilter();
    private CSensitiveWordsFilter cFilter = new CSensitiveWordsFilter();

    public String filterSensitiveWords(String text) {
        String maskedText = aFilter.filterSexyWords(text);
```

```

        maskedText = aFilter.filterPoliticalWords(maskedText);
        maskedText = bFilter.filter(maskedText);
        maskedText = cFilter.filter(maskedText, "***");
        return maskedText;
    }
}

```

// 使用适配器模式进行改造

```

public interface ISensitiveWordsFilter { // 统一接口定义
    String filter(String text);
}

public class ASensitiveWordsFilterAdaptor implements ISensitiveWordsFilter {
    private ASensitiveWordsFilter aFilter;
    public String filter(String text) {
        String maskedText = aFilter.filterSexyWords(text);
        maskedText = aFilter.filterPoliticalWords(maskedText);
        return maskedText;
    }
}

```

//...省略BSensitiveWordsFilterAdaptor、CSensitiveWordsFilterAdaptor...

// 扩展性更好，更加符合开闭原则，如果添加一个新的敏感词过滤系统，

// 这个类完全不需要改动；而且基于接口而非实现编程，代码的可测试性更好。

```

public class RiskManagement {
    private List<ISensitiveWordsFilter> filters = new ArrayList<>();

    public void addSensitiveWordsFilter(ISensitiveWordsFilter filter) {
        filters.add(filter);
    }

    public String filterSensitiveWords(String text) {
        String maskedText = text;
        for (ISensitiveWordsFilter filter : filters) {
            maskedText = filter.filter(maskedText);
        }
        return maskedText;
    }
}

```

3、替换依赖的外部系统

当我们把项目中依赖的一个外部系统替换为另一个外部系统的时候，利用适配器模式，可以减少对代码的改动。具体的代码示例如下所示：

// 外部系统A

```
public interface IA {  
    //...  
    void fa();  
}  
  
public class A implements IA {  
    //...  
    public void fa() { //...  
    }  
}
```

// 在我们的项目中，外部系统A的使用示例

```
public class Demo {  
    private IA a;  
    public Demo(IA a) {  
        this.a = a;  
    }  
    //...  
}  
  
Demo d = new Demo(new A());
```

// 将外部系统A替换成外部系统B

```
public class BAdaptor implements IA {  
    private B b;  
    public BAdaptor(B b) {  
        this.b = b;  
    }  
    public void fa() {  
        //...  
        b.fb();  
    }  
}
```

// 借助BAdaptor，Demo的代码中，调用IA接口的地方都无需改动，

// 只需要将BAdaptor如下注入到Demo即可。

```
Demo d = new Demo(new BAdaptor(new B()));
```

4、兼容老版本接口

在兼容老版本的角度上思考，我们一般有两种场景：

- 1、兼容老版本接口，新版本接口要在老版本接口做扩展，两个版本均可用。
- 2、老版本接口计划废弃，标注deprecated，但是不想改动已有代码，让两个版本兼容并行，但新功能不使用老版本。

首先我们思考第一种场景，这是老版本的支付接口：

```
// 老版本支付接口
public interface OldPayment {
    void pay(double amount);
}
```

然后，这是新版本的支付接口：

```
// 新版本支付接口
public interface NewPayment {
    void makePayment(double amount, String currency);
}
```

接下来，我们创建一个适配器类，实现新版本的支付接口，并在内部使用老版本的支付接口：

```
// 适配器类，实现新版本支付接口
public class PaymentAdapter implements NewPayment {
    private OldPayment oldPayment;

    public PaymentAdapter(OldPayment oldPayment) {
        this.oldPayment = oldPayment;
    }

    @Override
    public void makePayment(double amount, String currency) {
// 假设老版本支付接口只接受人民币，我们需要将其他货币转换为人民币
        if ("CNY".equals(currency)) {
            oldPayment.pay(amount);
        } else {
            double convertedAmount = convertToCNY(amount, currency);

```

```

        oldPayment.pay(convertedAmount);
    }
}

private double convertToCNY(double amount, String currency) {
    // 在这里进行货币转换的逻辑
    // 为了简化示例，我们假设所有其他货币都是1:1兑换人民币
    return amount;
}
}

```

最后，我们在客户端代码中使用适配器类，使其可以兼容新旧两种支付接口：

```

public class Client {
    public static void main(String[] args) {
        // 创建一个老版本支付实例
        OldPayment oldPaymentInstance = new OldPaymentImpl();
        // 创建适配器实例
        NewPayment paymentAdapter = new
        PaymentAdapter(oldPaymentInstance);

        // 通过适配器使用新版本支付接口
        paymentAdapter.makePayment(100, "CNY");
        paymentAdapter.makePayment(200, "USD");
    }
}

```

这样，我们就可以在不修改原有 `OldPayment` 接口的情况下，实现新旧接口的兼容。

接下来看第二种场景，老版本的接口**要废弃不使用**，但是很多地方使用了老版本的接口，我们想在不影响新老接口的使用的情况下，完成升级。为了完成该需求，我们可以将**适配器类修改为实现老版本接口**，然后**在内部使用新版本接口**。这样，原有的代码可以继续使用适配器类，而不需要进行任何修改。

首先，这是老版本的支付接口：

```

// 老版本支付接口
public interface OldPayment {
    void pay(double amount);
}

```

然后，这是新版本的支付接口：

```
// 新版本支付接口
public interface NewPayment {
    void makePayment(double amount, String currency);
}
```

接下来，我们创建一个适配器类，实现老版本的支付接口，并在内部使用新版本的支付接口：

```
// 适配器类，实现老版本支付接口
public class PaymentAdapter implements OldPayment {
    private NewPayment newPayment;

    public PaymentAdapter(NewPayment newPayment) {
        this.newPayment = newPayment;
    }

    @Override
    public void pay(double amount) {
// 假设新版本支付接口使用人民币，我们直接调用新接口
        newPayment.makePayment(amount, "CNY");
    }
}
```

最后，在客户端代码中，我们将原来使用老版本接口的实例替换为适配器实例：

```
public class Client {
    public static void main(String[] args) {
// 创建一个新版本支付实例
        NewPayment newPaymentInstance = new NewPaymentImpl();
// 创建适配器实例（我们只需要将这个新的适配器实例注入容器即可）
        OldPayment paymentAdapter = new
        PaymentAdapter(newPaymentInstance);

// 通过适配器使用老版本支付接口
        paymentAdapter.pay(100);
    }
}
```

这样，我们就可以在废弃老版本接口的情况下，实现新旧接口的兼容。原有的代码可以继续使用适配器类，而不需要进行任何修改。

三、源码中的使用

1、日志中的应用

Java 中有很多日志框架，比较常用的有 log4j、logback，以及 JDK 提供的 JUL(java.util.logging) 和 Apache 的 JCL (Jakarta Commons Logging)) 等。

大部分日志框架都提供了相似的功能，比如按照不同级别（debug、info、warn、erro.....）打印日志等，但它们却并没有实现统一的接口。这主要是历史的原因，它不像 JDBC 那样，**一开始就制定了数据库操作的接口规范。**

如果我们只是开发一个自己用的项目，那用什么日志框架都可以，log4j、logback 随便选一个就好。但是，如果我们开发的是一个集成到其他系统的**组件、框架、类库**等，那日志框架的选择就没那么随意了。

我们举一个例子，**项目中用到的某个组件使用 log4j 来打印日志**，而我们项目本身**使用的是 logback**。将组件引入到项目之后，我们的项目就相当于**有了两套日志打印框架**。每种日志框架都有自己特有的配置方式。所以，我们要针对每种日志框架编写不同的配置文件（比如，日志存储的文件地址、打印日志的格式）。如果引入多个组件，每个组件使用的日志框架都不一样，那**日志本身的管理工作就变得非常复杂**。所以，为了解决这个问题，我们需要**统一日志打印框架**。

Slf4j 这个日志框架你肯定不陌生，它相当于 JDBC 规范，是一套门面日志，提供了一套打印日志的统一接口规范。不过，它只定义了接口，并没有提供具体的实现，需要配合其他日志框架（log4j、logback.....）来使用。

不仅如此，Slf4j 的出现晚于 JUL、JCL、log4j 等日志框架，所以，这些日志框架也不可能牺牲掉版本兼容性，将接口改造成符合 Slf4j 接口规范。Slf4j 也事先考虑到了这个问题，所以，它不仅仅提供了统一的接口定义，还提供了**针对不同日志框架的适配器**。对不同日志框架的接口进行二次封装，适配成统一的 Slf4j 接口定义。具体的代码示例如下所示：

我们接下来就以 slf4j 为例，看看其中的绑定和桥接功能是如何巧妙实现兼容不同形式的日志的。

日志绑定

使用 SLF4J (Simple Logging Facade for Java) 绑定 Log4j 后，就可以无脑使用 SLF4J 的 api 进行日志记录，而实现还是原来的 log4j 实现，为了完成此功能我们需要执行以下步骤：

(1) 添加SLF4J和Log4j依赖

首先，需要在项目中添加SLF4J和Log4j的依赖。如果使用Maven，可以在 `pom.xml` 文件中添加以下依赖：

```
<dependencies>
  <!-- 添加SLF4J API依赖 -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.32</version>
  </dependency>
  <!-- 添加SLF4J绑定Log4j依赖，这个依赖是关键，使用了适配器模式进行了适配 -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.32</version>
  </dependency>
  <!-- 添加Log4j依赖 -->
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
  </dependency>
</dependencies>
```

(2) 创建一个Log4j配置文件

在项目的 `resources` 目录下，创建一个名为 `log4j.properties` 的文件。这是Log4j的配置文件，我们在这里定义日志记录的级别、格式和输出位置。

例如，我们可以创建如下配置：

```
propertiesCopy code
# 设置Log4j的根日志级别为INFO
log4j.rootLogger=INFO, stdout

# 配置输出到控制台
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss}
%-5p %c{1}:%L - %m%n
```

这个配置表示根日志级别为INFO，日志将输出到控制台，日志格式为 日期 时间 日志级别 类名:行号 - 消息内容。

(3) 编写Java代码使用SLF4J API进行日志记录

接下来，我们编写一个简单的Java类，使用SLF4J API进行日志记录：

```
public class SLF4JExample {  
    // 获取Logger实例  
    private static final Logger logger =  
        LoggerFactory.getLogger(SLF4JExample.class);  
  
    public static void main(String[] args) {  
        // 使用SLF4J API记录不同级别的日志  
        logger.debug("这是一条DEBUG级别的日志");  
        logger.info("这是一条INFO级别的日志");  
        logger.warn("这是一条WARN级别的日志");  
        logger.error("这是一条ERROR级别的日志");  
    }  
}
```

(4) 运行代码

现在我们已经完成了SLF4J绑定Log4j的配置和代码编写，接下来可以编译和运行这个示例。

运行结果应该显示如下：

```
2023-04-24 12:34:56 INFO SLF4JExample:10 - 这是一条INFO级别的日志  
2023-04-24 12:34:56 WARN SLF4JExample:11 - 这是一条WARN级别的日志  
2023-04-24 12:34:56 ERROR SLF4JExample:12 - 这是一条ERROR级别的日志
```

请注意，由于我们的Log4j配置中将根日志级别设置为INFO，所以DEBUG级别的日志不会被输出。

至此，我们已经成功地使用SLF4J绑定了Log4j，并通过SLF4J API进行了日志记录。

小节一下，看看原理：

slf4j-log4j12 是一个SLF4J的实现库，它将SLF4J API的**日志记录请求转发给Log4j 1.2作为底层日志框架。它实际上是一个适配器**，将SLF4J API与Log4j 1.2 API进行了适配。我们来看一下源码中的关键部分，以理解其实现原理。

(1) **Log4jLoggerFactory**：slf4j-log4j12 实现了SLF4J的 **ILoggerFactory** 接口，创建Log4j 1.2的 **Logger** 实例。这个工厂类负责将SLF4J的请求转换为Log4j 1.2的请求。

```
public class Log4jLoggerFactory implements ILoggerFactory {

    public Logger getLogger(String name) {
        // 获取Log4j 1.2的Logger实例
        org.apache.log4j.Logger log4jLogger = LogManager.getLogger(name);
        // 将Log4j 1.2的Logger实例包装成SLF4J的Logger实例并返回
        return new Log4jLoggerAdapter(log4jLogger);
    }
}
```

(2) **Log4jLoggerAdapter**：这个类实现了SLF4J的 **Logger** 接口，将SLF4J API转换为Log4j 1.2的API。它包装了一个Log4j 1.2的 **Logger** 实例，用于实际的日志记录。

```
public final class Log4jLoggerAdapter extends MarkerIgnoringBase {
    final Logger logger; // Log4j 1.2的Logger实例

    public Log4jLoggerAdapter(Logger logger) {
        this.logger = logger;
        this.name = logger.getName();
    }

    public boolean isDebugEnabled() {
        return logger.isDebugEnabled();
    }

    public void debug(String msg) {
        logger.log(FQCN, Level.DEBUG, msg, null);
    }

    // 其他方法，例如info(), error()等，也类似地转发给Log4j 1.2的Logger实例
}
```

当我们在项目中调用SLF4J的 `LoggerFactory` 获取一个 `Logger` 实例时，SLF4J会自动发现并使用 `slf4j-log4j12` 提供的 `Log4jLoggerFactory`。

`Log4jLoggerFactory` 会创建一个 `Log4jLoggerAdapter` 实例，这个实例内部包装了一个Log4j 1.2的 `Logger`。当我们使用SLF4J API进行日志记录时，`Log4jLoggerAdapter` 会将这些请求转换为Log4j 1.2可以处理的请求，从而实现了日志绑定。

通过这种适配器模式，`slf4j-log4j12` 实现了SLF4J API与Log4j 1.2的无缝集成，使得我们可以在项目中使用SLF4J API进行日志记录，同时底层使用Log4j 1.2作为实际的日志框架。这使得客户端代码只需关注SLF4J API，而无需关心底层日志框架的实现细节。此外，这种设计还为我们提供了灵活性，可以轻松地在不同的日志框架之间进行切换，只需更改项目依赖即可。

通过使用 `slf4j-log4j12`，我们可以在不修改客户端代码的情况下，将项目中的日志记录从其他日志框架迁移到Log4j 1.2，或者从Log4j 1.2迁移到其他框架。这大大简化了项目中日志框架迁移和升级的工作。

2、SpringMVC框架

在SpringMVC中，为了适配各种类型的处理器（Handler），使用了适配器设计模式。例如，`org.springframework.web.servlet.HandlerAdapter` 接口为各种处理器提供了统一的适配。具体实现类有

`org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter` 等。

// HandlerAdapter接口

```
public interface HandlerAdapter {  
    boolean supports(Object handler);  
    ModelAndView handle(HttpServletRequest request, HttpServletResponse  
response, Object handler) throws Exception;  
    long getLastModified(HttpServletRequest request, Object handler);  
}
```

// RequestMappingHandlerAdapter类实现了HandlerAdapter接口

```
public class RequestMappingHandlerAdapter extends WebContentGenerator  
implements HandlerAdapter {  
    // ...  
    // 判断是否支持此处理器  
    public boolean supports(Object handler) {  
        return handler instanceof HandlerMethod;  
    }  
}
```

```

    }

    // 处理请求
    public ModelAndView handle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        // ...
    }

    // 获取最后修改时间
    public long getLastModified(HttpServletRequest request, Object handler) {
        // ...
    }
}

```

四、代理、桥接、装饰器、适配器 4 种设计模式的区别

代理、桥接、装饰器、适配器，这 4 种模式是比较常用的结构型设计模式。它们的代码结构非常相似。笼统来说，它们都可以称为 Wrapper 模式，也就是通过 Wrapper 类二次封装原始类。

尽管代码结构相似，但这 4 种设计模式的用意完全不同，也就是说要解决的问题、应用场景不同，这也是它们的主要区别。这里我就简单说一下它们之间的区别。

代理模式：代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同。

桥接模式：桥接模式的目的是将接口部分和实现部分分离，从而让它们可以较为容易、也相对独立地加以改变。

装饰器模式：装饰者模式在不改变原始类接口的情况下，对原始类功能进行增强，并且支持多个装饰器的嵌套使用。

适配器模式：适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。

第五章 门面模式

一、门面模式的原理与实现

门面模式，也叫外观模式，英文全称是 Facade Design Pattern。在 GoF 的《设计模式》一书中，门面模式是这样定义的：

Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

翻译成中文就是：门面模式**为子系统提供一组统一的接口**，定义一组高层接口让子系统更易用。

门面模式（Facade Pattern）是一种结构型设计模式，它为一组**复杂的子系统提供了一个简单的接口**，使得**子系统更容易使用和理解**。在 Java 中，门面模式通常使用一个门面类（Facade Class）来包装一个或多个复杂的子系统，使得客户端只需要和门面类交互，而不需要直接与子系统交互。

门面模式通常在以下情况下使用：

1. **将复杂的子系统进行抽象和封装**：当一个系统变得复杂时，往往会由**多个子系统组成**。门面模式可以将这些子系统进行抽象和封装，提供一个简单的接口供客户端使用。
2. **隐藏子系统的复杂性**：门面模式可以将子系统的**复杂性隐藏起来**，使得客户端**不需要了解子系统的内部实现细节**。
3. **提供一个简单的接口**：门面模式可以为客户端提供一个简单的接口，使得客户端可以更容易地使用子系统。
4. **降低客户端与子系统的耦合**：门面模式可以将客户端和子系统解耦，使得客户端**不需要了解子系统的内部实现细节**，也不需要直接和子系统交互。

在 Java 中，门面模式通常用于以下场景：

1. **对外提供 API**：当一个系统需要对外提供 API 时，可以使用门面模式将 API 进行封装，使得客户端只需要使用一个简单的接口就可以调用系统的功能。
2. **与其他系统进行交互**：当一个系统需要与其他系统进行交互时，可以使用门面模式将与其他系统的交互进行封装，使得系统可以更容易地与其他系统进行集成。
3. **简化复杂的操作**：当一个系统需要进行复杂的操作时，可以使用门面模式将操作进行封装，使得操作可以更容易地使用和理解。
4. **对系统进行分层**：当一个系统需要进行分层时，可以使用门面模式将不同层之间的接口进行封装，使得各层之间的依赖关系更加清晰，同时也更容易进行维护和修改。

例子一：

当一个系统需要与多个第三方服务进行交互时，可以使用门面模式来对这些服务进行封装，使得**客户端只需要与一个门面类交互就可以完成对多个服务的调用**。下面是一个简单的示例代码：

// 外部服务接口

```
interface ExternalService {  
    void doSomething();  
}
```

// 外部服务实现类1

```
class ExternalServiceImpl1 implements ExternalService {  
    @Override  
    public void doSomething() {  
        System.out.println("ExternalServiceImpl1.doSomething");  
    }  
}
```

// 外部服务实现类2

```
class ExternalServiceImpl2 implements ExternalService {  
    @Override  
    public void doSomething() {  
        System.out.println("ExternalServiceImpl2.doSomething");  
    }  
}
```

// 门面类

```
class Facade {  
    private ExternalService service1;  
    private ExternalService service2;  
  
    public Facade() {  
        service1 = new ExternalServiceImpl1();  
        service2 = new ExternalServiceImpl2();  
    }  
  
    public void doSomething1() {  
        service1.doSomething();  
    }  
  
    public void doSomething2() {  
        service2.doSomething();  
    }  
}
```



```

    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.doSomething1();
        facade.doSomething2();
    }
}

```

在这个示例中，ExternalService 是一个外部服务的接口，ExternalServiceImpl1 和 ExternalServiceImpl2 是这个接口的两个具体实现。Facade 是一个门面类，它将这两个外部服务进行封装，并提供了两个简单的方法 doSomething1 和 doSomething2。客户端只需要与 Facade 类交互，就可以完成对这两个服务的调用。

我们不妨举再一个例子：

当一个系统需要访问多个不同的数据库时，可以使用门面模式来对这些数据库进行封装，使得客户端只需要与一个门面类交互就可以完成对多个数据库的访问。下面是一个简单的示例代码：

```

// 数据库接口
interface Database {
    void execute(String sql);
}

// MySQL数据库实现类
class MySQLDatabase implements Database {
    @Override
    public void execute(String sql) {
        System.out.println("Executing " + sql + " in MySQL database");
    }
}

// Oracle数据库实现类
class OracleDatabase implements Database {
    @Override
    public void execute(String sql) {
        System.out.println("Executing " + sql + " in Oracle database");
    }
}

```

```

    }
}

// 门面类
class DatabaseFacade {
    private Database mysqlDatabase;
    private Database oracleDatabase;

    public DatabaseFacade() {
        mysqlDatabase = new MySQLDatabase();
        oracleDatabase = new OracleDatabase();
    }

    public void executeSQL(String sql, String databaseType) {
        if (databaseType.equals("MySQL")) {
            mysqlDatabase.execute(sql);
        } else if (databaseType.equals("Oracle")) {
            oracleDatabase.execute(sql);
        } else {
            throw new IllegalArgumentException("Unknown database type: " +
databaseType);
        }
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        DatabaseFacade facade = new DatabaseFacade();
        facade.executeSQL("SELECT * FROM users", "MySQL");
        facade.executeSQL("SELECT * FROM customers", "Oracle");
    }
}

```

在这个示例中，Database 是一个数据库接口，MySQLDatabase 和 OracleDatabase 是这个接口的两个具体实现。DatabaseFacade 是一个门面类，它将这两个数据库进行封装，并提供了一个 executeSQL 方法，用于执行 SQL 语句。客户端只需要与 DatabaseFacade 类交互，并指定要访问的数据库类型，就可以完成对这两个数据库的访问。

事实上，我们学习了设计模式，更应该思考的问题是，面对不合理的接口，我们应该如何设计合理的接口粒度以兼顾接口的易用性和通用性？

以下是一些策略和指导方针：

1. **单一职责原则 (SRP)**：每个接口或方法应该只有一个单一的职责。这可以保持接口的简单性，并使其更易于理解和使用。
2. **抽象程度**：接口的设计应该足够抽象，能够容纳多种具体的实现，这可以提高接口的通用性。但是，过度的抽象可能会使接口难以理解和使用，因此需要找到一个合适的平衡点。
3. **尽量减少接口的依赖性**：接口的方法不应该有过多的参数，尽量减少对其他对象或方法的依赖。这样可以使接口更简单，更易于使用。
4. **易于使用的API**：设计接口时，应考虑使用者的需求和使用场景。应提供易于使用的API，例如，提供默认参数，使用有意义的方法和变量名，以及充分的文档和示例。
5. **灵活性和扩展性**：接口设计应考虑未来可能的改变和扩展。接口的方法和参数应该有足够的灵活性，可以适应新的需求和变化。
6. **封装**：接口应该尽可能地隐藏其实现的细节。这可以使接口更易于使用，同时也可以防止使用者直接访问和修改内部状态。

二、门面模式的应用场景举例

还要强调一下，门面模式定义中的“子系统 (subsystem)”也可以有多种理解方式。它既可以是一个完整的系统，也可以是更细粒度的类或者模块。关于这一点，在下面的讲解中也会有体现。

门面模式可以让子系统更加易用，实际上，它除了解决易用性问题之外，还能解决其他很多方面的问题，如分布式事物等。

1、源码使用

门面模式在 JDK 源码中有很多应用。以下是一些常见的使用场景：

(1) JDBC：在 Java 中使用 JDBC 连接数据库时，通常会使用 DriverManager 来获取连接。DriverManager 就是一个门面类，它将多个数据库驱动进行封装，使得客户端只需要使用一个简单的接口就可以访问不同的数据库。

(2) Spring 框架：在 Spring 框架中，ApplicationContext 就是一个门面类，它将 Spring 中的各个组件进行封装，使得客户端可以更容易地使用 Spring 中的功能。

(3) Servlet API：在 Servlet API 中，HttpServletRequest 和 HttpServletResponse 接口就是门面类，它们将底层的网络通信进行封装，使得开发者可以更容易地编写 Web 应用程序。

2、使用场景

在生产环境中，门面模式常常用于封装复杂的第三方 API 或系统，以提供简单、易用的接口给客户端使用。一个具体的例子是，假设我们的系统需要与多个支付系统进行交互，而每个支付系统的接口和参数都不一样，这时候就可以使用门面模式来对这些支付系统进行封装，使得客户端只需要使用一个简单的接口就可以完成对多个支付系统的调用。

以下是一个示例代码：

// 支付系统接口

```
interface PaymentSystem {  
    void pay(double amount);  
}
```

// 支付宝接口实现类

```
class AliPay implements PaymentSystem {  
    @Override  
    public void pay(double amount) {  
        System.out.println("支付宝支付: " + amount + "元");  
    }  
}
```

// 微信支付接口实现类

```
class WeChatPay implements PaymentSystem {  
    @Override  
    public void pay(double amount) {  
        System.out.println("微信支付: " + amount + "元");  
    }  
}
```

// 门面类

```
class PaymentFacade {  
    private PaymentSystem aliPay;  
    private PaymentSystem weChatPay;  
  
    public PaymentFacade() {  
        aliPay = new AliPay();  
        weChatPay = new WeChatPay();  
    }  
  
    public void payByAliPay(double amount) {  
        aliPay.pay(amount);  
    }  
}
```

```

    }

    public void payByWeChatPay(double amount) {
        weChatPay.pay(amount);
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        PaymentFacade paymentFacade = new PaymentFacade();
        paymentFacade.payByAliPay(100.0);
        paymentFacade.payByWeChatPay(200.0);
    }
}

```

在这个示例中，PaymentSystem 是支付系统的接口，AliPay 和 WeChatPay 是这个接口的两个具体实现。PaymentFacade 是一个门面类，它将这两个支付系统进行封装，并提供了两个简单的方法 payByAliPay 和 payByWeChatPay。客户端只需要与 PaymentFacade 类交互，就可以完成对这两个支付系统的调用。这种方式可以方便地支持新的支付系统的加入，同时也可以提高客户端的调用效率和代码可读性。

最后我想说几句：

我们知道，类、模块、系统之间的“通信”，一般都是通过接口调用来完成的。接口设计的好坏，直接影响到类、模块、系统是否好用。所以，我们要多花点心思在接口设计上。我经常说，**完成接口设计，就相当于完成了一半的开发任务。只要接口设计得好，那代码就差不到哪里去。**

接口粒度设计得太大，太小都不好。太大会导致接口不可复用，太小会导致接口不易用。在实际的开发中，接口的可复用性和易用性需要“微妙”的权衡。针对这个问题，我的一个基本的处理原则是，**尽量保持接口的可复用性，但针对特殊情况，允许提供冗余的门面接口，来提供更易用的接口。**

门面模式除了解决接口易用性问题之外，我们今天还讲到了其他 2 个应用场景，用它来解决性能问题和分布式事务问题。

q: 事实上，适配器模式和门面模式都可以将不好用的接口适配成好用的接口，那他们之间又有什么区别呢？

1. 目标和目的：

- **适配器模式**：适配器模式的目标是为了让现有接口能适配不同的类，让不兼容的接口能够一起工作。这是为了解决接口不兼容性问题，可以在没有修改源码的情况下使用现有的类。
- **门面模式**：门面模式的目标是为了提供一个统一的高级接口，隐藏子系统的复杂性。这是为了简化客户端对子系统的访问和使用。

2. 应用场景：

- **适配器模式**：当我们希望将一个已经存在的类的接口转换成另一个接口以供客户程序使用，或者当我们希望创建一个可重用的类，这个类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作时，可以使用适配器模式。
- **门面模式**：当我们希望为一个复杂的子系统提供一个简单的接口，或者客户程序与多个子系统之间存在大量的依赖关系时，可以使用门面模式。

3. 封装级别和复杂性：

- **适配器模式**：适配器模式通常只包装一个类或对象，将一个接口转换为另一个接口。
- **门面模式**：门面模式包装了一整个子系统或者一组接口，将一系列复杂操作封装成一个简单的接口。

所以，尽管适配器模式和门面模式都在某种程度上提供了接口转换，但它们的应用上下文，目的和封装的粒度却是不同的。

第六章 组合模式

一、组合模式的原理与实现

在 GoF 的《设计模式》一书中，组合模式是这样定义的：

Compose objects into tree structure to represent part-whole hierarchies. Composite lets client treat individual objects and compositions of objects uniformly.

翻译过来就是：将一组对象组织（Compose）成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（在很多设计模式书籍中，“客户端”代指代码的使用者）可以统一单个对象和组合对象的处理逻辑。

组合模式（Composite Pattern）是一种**结构型设计模式**。在组合模式中，**每个对象都有相同的接口**，这使得客户端**不需要知道对象的具体类型**，而只需要调用对象的**通用接口**即可。

组合模式包含以下几个角色：

1. Component（抽象构件）：定义组合对象的通用接口，可以包含其他组合对象或叶子对象。
2. Leaf（叶子节点）：表示组合对象中的叶子节点，它没有子节点。
3. Composite（组合节点）：表示组合对象中的组合节点，它可以包含其他组合对象或叶子对象。

下面是一个简单的组合模式示例代码，用于表示文件系统中的文件和文件夹：

```
// Component（抽象构件）
interface FileSystem {
    void display();
}

// Leaf（叶子节点）
class File implements FileSystem {
    private String name;

    public File(String name) {
        this.name = name;
    }

    @Override
    public void display() {
        System.out.println("File: " + name);
    }
}

// Composite（组合节点）
class Folder implements FileSystem {
    private String name;
    private List<FileSystem> children;

    public Folder(String name) {
        this.name = name;
        children = new ArrayList<>();
    }

    public void add(FileSystem fileSystem) {
        children.add(fileSystem);
    }
}
```

```

public void remove(FileSystem fileSystem) {
    children.remove(fileSystem);
}

@Override
public void display() {
    System.out.println("Folder: " + name);
    for (FileSystem fileSystem : children) {
        fileSystem.display();
    }
}
}

```

// 客户端代码

```

public class Client {
    public static void main(String[] args) {
        FileSystem file1 = new File("file1.txt");
        FileSystem file2 = new File("file2.txt");

        Folder folder1 = new Folder("folder1");
        folder1.add(file1);
        folder1.add(file2);

        FileSystem file3 = new File("file3.txt");
        FileSystem file4 = new File("file4.txt");

        Folder folder2 = new Folder("folder2");
        folder2.add(file3);
        folder2.add(file4);
        folder2.add(folder1);

        folder2.display();
    }
}

```

在这个示例中，FileSystem 是**抽象构件**，它定义了组合对象的通用接口 display。File 是叶子节点，表示文件，它实现了 FileSystem 接口，并在 display 方法中输出文件名。Folder 是组合节点，表示文件夹，它实现了 FileSystem 接口，并维护了一个子节点列表 children，可以添加和删除子节点。在 display 方法中，它首先输出文件夹名，然后依次调用子节点的 display 方法输出子节点信息。

在客户端代码中，我们创建了一些文件和文件夹，然后将它们组合成了一个树形结构，最后调用根最后调用根节点（即 folder2）的 display 方法，输出了整个文件系统的信息。这样，我们就可以通过组合模式，使用相同的方式来处理单个文件和整个文件系统。

组合模式的优点包括：

1. 可以使用相同的方式来处理单个对象和组合对象，客户端无需知道对象的具体类型。
2. 可以方便地增加新的组合对象或叶子对象，同时也可以方便地对组合对象进行遍历和操作。
3. 可以使代码更加简洁和易于维护，因为使用组合模式可以避免大量的 if-else 或 switch-case 语句。

组合模式的缺点包括：

1. 在组合对象中，可能会包含大量的叶子对象，这可能会导致系统的性能下降。
2. 可能会使设计过于抽象化，使得代码难以理解和维护。

总之，组合模式在处理树形结构等层次结构时非常有用，可以方便地处理单个对象和组合对象，使得代码更加简洁和易于维护。

那接下来我们将文件目录的案例做一个升级，如何设计实现支持递归遍历的文件系统目录树结构？

假设我们有这样一个需求：设计一个类来表示文件系统中的目录，能方便地实现下面这些功能：

- 动态地添加、删除某个目录下的子目录或文件；
- 统计指定目录下的文件个数；
- 统计指定目录下的文件总大小。

我这里给出了这个类的骨架代码，如下所示。其中的核心逻辑并未实现，你可以试着自己去补充完整，再来看我的讲解。在下面的代码实现中，我们把文件和目录统一用 FileSystemNode 类来表示，并且通过 isFile 属性来区分。

```
public class FileSystemNode {
    private String path;
    private boolean isFile;
    private List<FileSystemNode> subNodes = new ArrayList<>();
    public FileSystemNode(String path, boolean isFile) {
        this.path = path;
        this.isFile = isFile;
    }
}
```

```

public int countNumOfFiles() {
    // TODO:...
}
public long countSizeOfFiles() {
    // TODO:...
}
public String getPath() {
    return path;
}
public void addSubNode(FileSystemNode fileOrDir) {
    subNodes.add(fileOrDir);
}
public void removeSubNode(FileSystemNode fileOrDir) {
    int size = subNodes.size();
    for (int i = 0; i < size; ++i) {
        if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
            subNodes.remove(i);
            i--;
        }
    }
}
}
}

```

我把两个函数的代码实现贴在下面了，你可以对照着看一下。

```

public int countNumOfFiles() {
    if (isFile) {
        return 1;
    }
    int numOfFiles = 0;
    for (FileSystemNode fileOrDir : subNodes) {
        numOfFiles += fileOrDir.countNumOfFiles();
    }
    return numOfFiles;
}

public long countSizeOfFiles() {
    if (isFile) {
        File file = new File(path);
        if (!file.exists()) return 0;
        return file.length();
    }
}

```

```

    }
    long sizeofFiles = 0;
    for (FileSystemNode fileOrDir : subNodes) {
        sizeofFiles += fileOrDir.countSizeOfFiles();
    }
    return sizeofFiles;
}

```

单纯从功能实现角度来说，上面的代码没有问题，已经实现了我们想要的功能。但是，如果我们开发的是一个大型系统，**从扩展性**（文件或目录可能会对应不同的操作）、**业务建模**（文件和目录从业务上是两个概念）、**代码的可读性**（文件和目录区分对待更加符合人们对业务的认知）的角度来说，我们最好对文件和目录进行区分设计，定义为 File 和 Directory 两个类。

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：

```

public abstract class FileSystemNode {
    protected String path;
    public FileSystemNode(String path) {
        this.path = path;
    }
    public abstract int countNumOfFiles();
    public abstract long countSizeOfFiles();
    public String getPath() {
        return path;
    }
}

public class File extends FileSystemNode {
    public File(String path) {
        super(path);
    }
    @Override
    public int countNumOfFiles() {
        return 1;
    }
    @Override
    public long countSizeOfFiles() {
        java.io.File file = new java.io.File(path);
        if (!file.exists()) return 0;
        return `

```

```

    }
}

public class Directory extends FileSystemNode {
    private List<FileSystemNode> subNodes = new ArrayList<>();
    public Directory(String path) {
        super(path);
    }
    @Override
    public int countNumOfFiles() {
        int numOfFiles = 0;
        for (FileSystemNode fileOrDir : subNodes) {
            numOfFiles += fileOrDir.countNumOfFiles();
        }
        return numOfFiles;
    }
    @Override
    public long countSizeOfFiles() {
        long sizeofFiles = 0;
        for (FileSystemNode fileOrDir : subNodes) {
            sizeofFiles += fileOrDir.countSizeOfFiles();
        }
        return sizeofFiles;
    }
    public void addSubNode(FileSystemNode fileOrDir) {
        subNodes.add(fileOrDir);
    }
    public void removeSubNode(FileSystemNode fileOrDir) {
        int size = subNodes.size();
        int i = 0;
        for (; i < size; ++i) {
            if (subNodes.get(i).getPath().equalsIgnoreCase(fileOrDir.getPath())) {
                break;
            }
        }
        if (i < size) {
            subNodes.remove(i);
        }
    }
}

```

文件和目录类都设计好了，我们来看，如何用它们来表示一个文件系统中的目录树结构。具体的代码示例如下所示：

```

public class Demo {
    public static void main(String[] args) {
        Directory fileSystemTree = new Directory("/");
        Directory nodeYdlclass = new Directory("/ydlclass/");
        Directory nodeYdl = new Directory("/ydl/");
        fileSystemTree.addSubNode(nodeYdlclass);
        fileSystemTree.addSubNode(nodeYdl);

        File nodeYdlclassA = new File("/ydlclass/a.txt");
        File nodeYdlclassB = new File("/ydlclass/b.txt");
        Directory nodeYdlclassMovies = new Directory("/ydlclass/movies/");
        nodeYdlclass.addSubNode(nodeYdlclassA);
        nodeYdlclass.addSubNode(nodeYdlclassB);
        nodeYdlclass.addSubNode(nodeYdlclassMovies);
        File nodeYdlclassMoviesC = new File("/ydlclass/movies/c.avi");
        nodeYdlclassMovies.addSubNode(nodeYdlclassMoviesC);

        System.out.println("/ files num:" + fileSystemTree.countNumOfFiles());
        System.out.println("/wz/ files num:" + node_wz.countNumOfFiles());
    }
}

```

我们对照着这个例子，再重新看一下组合模式的定义：“将一组对象（文件和目录）组织成树形结构，以表示一种‘部分 - 整体’的层次结构（目录与子目录的嵌套结构）。组合模式让客户端可以统一单个对象（文件）和组合对象（目录）的处理逻辑（递归遍历）。”

实际上，刚才讲的这种组合模式的设计思路，与其说是一种设计模式，倒不如说是对业务场景的一种数据结构和算法的抽象。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。

二、组合模式的应用场景举例

刚刚我们讲了文件系统的例子，对于组合模式，我这里再举一个例子。搞懂了这两个例子，你基本上就算掌握了组合模式。在实际的项目中，遇到类似的可以表示成树形结构的业务场景，你只要“照葫芦画瓢”去设计就可以了。

假设我们在开发一个 OA 系统（办公自动化系统）。公司的组织结构包含部门和员工两种数据类型。其中，部门又可以包含子部门和员工。在数据库中的表结构如下所示：

我们希望在内存中构建整个公司的人员架构图（部门、子部门、员工的隶属关系），并且提供接口计算出部门的薪资成本（隶属于这个部门的所有员工的薪资和）。

部门包含子部门和员工，这是一种嵌套结构，可以表示成树这种数据结构。计算每个部门的薪资开支这样一个需求，也可以通过在树上的遍历算法来实现。所以，从这个角度来看，这个应用场景可以使用组合模式来设计和实现。

这个例子的代码结构跟上一个例子的很相似，代码实现我直接贴在了下面，你可以对比着看一下。其中，HumanResource 是部门类（Department）和员工类

（Employee）抽象出来的父类，为的是能统一薪资的处理逻辑。Demo 中的代码负责从数据库中读取数据并在内存中构建组织架构图。

```
public abstract class HumanResource {
    protected long id;
    protected double salary;
    public HumanResource(long id) {
        this.id = id;
    }
    public long getId() {
        return id;
    }
    public abstract double calculateSalary();
}

public class Employee extends HumanResource {
    public Employee(long id, double salary) {
        super(id);
        this.salary = salary;
    }
    @Override
    public double calculateSalary() {
        return salary;
    }
}

public class Department extends HumanResource {
    private List<HumanResource> subNodes = new ArrayList<>();
    public Department(long id) {
        super(id);
    }
    @Override
    public double calculateSalary() {
        double totalSalary = 0;
```

```

    for (HumanResource hr : subNodes) {
        totalSalary += hr.calculateSalary();
    }
    this.salary = totalSalary;
    return totalSalary;
}

public void addSubNode(HumanResource hr) {
    subNodes.add(hr);
}
}

// 构建组织架构的代码
public class Demo {
    private static final long ORGANIZATION_ROOT_ID = 1001;
    private DepartmentRepo departmentRepo; // 依赖注入
    private EmployeeRepo employeeRepo; // 依赖注入
    public void buildOrganization() {
        Department rootDepartment = new Department(ORGANIZATION_ROOT_ID);
        buildOrganization(rootDepartment);
    }

    private void buildOrganization(Department department) {
        List<Long> subDepartmentIds =
departmentRepo.getSubDepartmentIds(department.getId());
        for (Long subDepartmentId : subDepartmentIds) {
            Department subDepartment = new Department(subDepartmentId);
            department.addSubNode(subDepartment);
            buildOrganization(subDepartment);
        }
        List<Long> employeeIds =
employeeRepo.getDepartmentEmployeeIds(department.getId());
        for (Long employeeId : employeeIds) {
            double salary = employeeRepo.getEmployeeSalary(employeeId);
            department.addSubNode(new Employee(employeeId, salary));
        }
    }
}

```

我们再拿组合模式的定义跟这个例子对照一下：“将一组对象（员工和部门）组织成树形结构，以表示一种部分 - 整体的层次结构（部门与子部门的嵌套结构）。组合模式让客户端可以统一单个对象（员工）和组合对象（部门）的处理逻辑（递归遍历）。”

三、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。

组合模式的设计思路，与其说是一种设计模式，倒不如说是对业务场景的一种数据结构和算法的抽象。其中，数据可以表示成树这种数据结构，业务需求可以通过在树上的递归遍历算法来实现。

组合模式，将一组对象组织成树形结构，将单个对象和组合对象都看做树中的节点，以统一处理逻辑，并且它利用树形结构的特点，递归地处理每个子树，依次简化代码实现。使用组合模式的前提在于，你的业务场景必须能够表示成树形结构。所以，组合模式的应用场景也比较局限，它并不是一种很常用的设计模式。

四、源码应用

1、jdk源码

组合模式在 JDK 源码中也有很多应用。以下是一些常见的使用场景：

1. Java Collection 框架：在 Java Collection 框架中，Collection 接口就是一个抽象构件，它定义了集合对象的通用接口。List、Set 和 Map 等具体集合类就是组合节点或叶子节点，用于存储和操作集合中的元素。
2. Servlet API：在 Servlet API 中，ServletRequest 和 ServletResponse 接口就是一个抽象构件，它定义了 Servlet 的通用接口。HttpServletRequest 和 HttpServletResponse 等具体类就是组合节点或叶子节点，用于处理 Web 请求和响应。

总之，组合模式在 JDK 源码中也有着广泛的应用，可以帮助开发者更加方便地操作各种层次结构。

2、ssm源码

在 SSM (Spring + Spring MVC + MyBatis) 框架中，组合模式也有一些应用场景，以下是一些常见的使用场景：

1. Spring MVC：在 Spring MVC 中，Controller 就是一个组合节点，它可以包含其他组合对象或叶子对象，用于处理 Web 请求和响应。对于复杂的请求处理逻辑，可以将一个 Controller 分解成多个子 Controller，然后通过组合的方式将它们组合起来，使得请求处理逻辑更加清晰和易于维护。
2. MyBatis：在 MyBatis 中，SqlNode 就是一个抽象构件，它定义了 SQL 节点的通用接口。WhereSqlNode、ChooseSqlNode、IfSqlNode 等具体类就是组合节点或叶子节点，用于构建 SQL 语句，解析动态sql。

总之，组合模式可以帮助开发者更加方便地管理和组织各种组件和模块。

第七章 享元模式

一、享元模式原理与实现

所谓“享元”，顾名思义就是被共享的单元。他也是一个不怎么常用的设计模式，享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。

具体来讲，当一个系统中存在**大量重复对象的时候，如果这些重复的对象是不可变对象，我们就可以利用享元模式将对象设计成享元**，在内存中只保留一份实例，供多处代码引用。这样可以减少内存中对象的数量，起到节省内存的目的。实际上，**不仅仅相同对象可以设计成享元，对于相似对象，我们也可以将这些对象中相同的部分（字段）提取出来，设计成享元，让这些大量相似对象引用这些享元。**

这里定义中的“不可变对象”指的是，一旦通过构造函数初始化完成之后，它的状态（对象的成员变量或者属性）就不会再被修改了。所以，不可变对象不能暴露任何 set() 等修改内部状态的方法。之所以要求享元是不可变对象，那是因为它会被多处代码共享使用，避免一处代码对享元进行了修改，影响到其他使用它的代码。

接下来，我们通过一个简单的例子解释一下享元模式。

假设我们在开发一个棋牌游戏（比如象棋）。一个游戏厅中有成千上万个“房间”，每个房间对应一个棋局。棋局要保存每个棋子的数据，比如：棋子类型（将、相、士、炮等）、棋子颜色（红方、黑方）、棋子在棋局中的位置。利用这些数据，我们就能显示一个完整的棋盘给玩家。具体的代码如下所示。其中，ChessPiece 类表示棋子，ChessBoard 类表示一个棋局，里面保存了象棋中 30 个棋子的信息。

```
public class ChessPiece {//棋子
    private int id;
    private String text;
    private Color color;

    public ChessPiece(int id, String text, Color color, int positionX, int positionY) {
        this.id = id;
        this.text = text;
        this.color = color;
        this.positionX = positionX;
        this.positionY = positionX;
    }
    public static enum Color {
        RED, BLACK
    }
}
```

```

    }
    // ...省略其他属性和getter/setter方法...
}

public class ChessBoard { //棋局
    private Map<Integer, ChessPiece> chessPieces = new HashMap<>();
    public ChessBoard() {
        init();
    }
    private void init() {
        chessPieces.put(1, new ChessPiece(1, "車", ChessPiece.Color.BLACK, 0, 0));
        chessPieces.put(2, new ChessPiece(2, "馬", ChessPiece.Color.BLACK, 0, 1));
        //...省略摆放其他棋子的代码...
    }
    public void move(int chessPiecelId, int toPositionX, int toPositionY) {
        //...省略...
    }
}

```

为了记录每个房间当前的棋局情况，我们需要给每个房间都创建一个 ChessBoard 棋局对象。因为游戏大厅中有成千上万的房间（实际上，百万人同时在线的游戏大厅也有很多），那保存这么多棋局对象就会消耗大量的内存。有没有什么办法来节省内存呢？

这个时候，享元模式就可以派上用场了。像刚刚的实现方式，在内存中会有大量的相似对象。这些相似对象的 id、text、color 都是相同的，唯独 positionX、positionY 不同。实际上，我们可以将棋子的 id、text、color 属性拆分出来，设计成独立的类，并且作为享元供多个棋盘复用。这样，棋盘只需要记录每个棋子的位置信息就可以了。具体的代码实现如下所示：

```

// 享元类
@Override
public class ChessUnit {

    private Long id;
    private String text;
    private Color color;

    public ChessUnit(Long id, String text, ChessUnit.Color color) {
        this.id = id;
        this.text = text;
        this.color = color;
    }
}

```

```

    public enum Color{
        RED,BLACK
    }
}

```

// 享元工厂

```

public class ChessUnitFactory {

    private static Map<Long,ChessUnit> chessUnitMap = new HashMap<>(64);

    static {
        chessUnitMap.put(1L,new ChessUnit(1L,"兵",ChessUnit.Color.RED));
        chessUnitMap.put(2L,new ChessUnit(2L,"马",ChessUnit.Color.RED));
        chessUnitMap.put(3L,new ChessUnit(3L,"炮",ChessUnit.Color.RED));
        chessUnitMap.put(4L,new ChessUnit(4L,"将",ChessUnit.Color.RED));
        chessUnitMap.put(5L,new ChessUnit(5L,"将",ChessUnit.Color.BLACK));
    }

    /**
     * 暴露一个工厂方法，用来获取棋子
     * @param id 棋子的id
     * @return 棋子
     */
    public static ChessUnit getChessUnit(Long id){
        return chessUnitMap.get(id);
    }

}

@Data
@AllArgsConstructor
@NoArgsConstructor
@EqualsAndHashCode
public class Position {
    private int positionX;
    private int positionY;
}

@Data
@AllArgsConstructor
public class ChessPiece {

```

```

private ChessUnit chessUnit;
private Position position;

}

public class ChessBoard {

    // 应该持有一个套棋子（有具体的坐标）
    private Map<Position,ChessPiece> chessPieceMap ;

    public ChessBoard() {
        // 构造棋牌
        this.chessPieceMap = new HashMap<>(64);
        // 初始化棋牌
        Position position1 = new Position(1, 2);
        chessPieceMap.put(position1,new
ChessPiece(ChessUnitFactory.getChessUnit(1L),position1));
        Position position2 = new Position(1, 4);
        chessPieceMap.put(position2,new
ChessPiece(ChessUnitFactory.getChessUnit(1L),position2));
        Position position3 = new Position(1, 5);
        chessPieceMap.put(position3,new
ChessPiece(ChessUnitFactory.getChessUnit(3L),position3));

    }
    public void display(){
        for (Map.Entry<Position,ChessPiece> entry : chessPieceMap.entrySet()){
            System.out.println(entry.getKey() + "-->" + entry.getValue());

        }

    }

    public static void main(String[] args) {
        ChessBoard chessBoard = new ChessBoard();
        chessBoard.display();
    }
}

```

在上面的代码实现中，我们利用工厂类来缓存 ChessPieceUnit 信息（也就是 id、text、color）。通过工厂类获取到的 ChessPieceUnit 就是享元。所有的 ChessBoard 对象共享这 30 个 ChessPieceUnit 对象（因为象棋中只有 30 个棋子）。在使用享元模式之前，记录 1 万个棋局，我们要创建 30 万（30*1 万）个棋

子的 ChessPieceUnit 对象。利用享元模式，我们只需要创建 30 个享元对象供所有棋局共享使用即可，大大节省了内存。

那享元模式的原理讲完了。实际上，它的代码实现非常简单，主要是通过工厂模式，在工厂类中，通过一个 Map 来缓存已经创建过的享元对象，来达到复用的目的。

二、源码应用

1、享元模式在 Java Integer 中的应用

我们先来看下面这样一段代码。你可以先思考下，这段代码会输出什么样的结果。

```
Integer i1 = 56;  
Integer i2 = 56;  
Integer i3 = 129;  
Integer i4 = 129;  
System.out.println(i1 == i2);  
System.out.println(i3 == i4);
```

如果不熟悉 Java 语言，你可能会觉得，i1 和 i2 值都是 56，i3 和 i4 值都是 129，i1 跟 i2 值相等，i3 跟 i4 值相等，所以输出结果应该是两个 true。这样的分析是不对的，主要还是因为你对 Java 语法不熟悉。要正确地分析上面的代码，我们需要弄清楚下面两个问题：

如何判定两个 Java 对象是否相等（也就代码中的“==”操作符的含义）？

什么是自动装箱（Autoboxing）和自动拆箱（Unboxing）？

在加餐一中，我们讲到，Java 为基本数据类型提供了对应的包装器类型。具体如下所示：

所谓的自动装箱，就是自动将基本数据类型转换为包装器类型。所谓的自动拆箱，也就是自动将包装器类型转化为基本数据类型。具体的代码示例如下所示：

```
Integer i = 56; //自动装箱  
int j = i; //自动拆箱
```

数值 56 是基本数据类型 int，当赋值给包装器类型（Integer）变量的时候，触发自动装箱操作，创建一个 Integer 类型的对象，并且赋值给变量 i。其底层相当于执行了下面这条语句：

```
Integer i = 59; 底层执行了: Integer i = Integer.valueOf(59);
```

反过来, 当把包装器类型的变量 `i`, 赋值给基本数据类型变量 `j` 的时候, 触发自动拆箱操作, 将 `i` 中的数据取出, 赋值给 `j`。其底层相当于执行了下面这条语句:

```
int j = i; 底层执行了: int j = i.intValue();
```

弄清楚了自动装箱和自动拆箱, 我们再来看, 如何判定两个对象是否相等? 不过, 在此之前, 我们先要搞清楚, Java 对象在内存中是如何存储的。我们通过下面这个例子来说明一下。

```
User a = new User(123, 23); // id=123, age=23
```

针对这条语句, 我画了一张内存存储结构图, 如下所示。a 存储的值是 User 对象的内存地址, 在图中就表现为 a 指向 User 对象。

当我们通过“==”来判定两个对象是否相等的时候, 实际上是在判断两个局部变量存储的地址是否相同, 换句话说, 是在判断两个局部变量是否指向相同的对象。

了解了 Java 的这几个语法之后, 我们重新看一下开头的那段代码。

```
Integer i1 = 56;  
Integer i2 = 56;  
Integer i3 = 129;  
Integer i4 = 129;  
System.out.println(i1 == i2);  
System.out.println(i3 == i4);
```

前 4 行赋值语句都会触发自动装箱操作, 也就是会创建 Integer 对象并且赋值给 `i1`、`i2`、`i3`、`i4` 这四个变量。根据刚刚的讲解, `i1`、`i2` 尽管存储的数值相同, 都是 56, 但是指向不同的 Integer 对象, 所以通过“==”来判定是否相同的时候, 会返回 false。同理, `i3==i4` 判定语句也会返回 false。

不过, 上面的分析还是不对, 答案并非是两个 false, 而是一个 true, 一个 false。看到这里, 你可能会比较纳闷了。实际上, 这正是因为 Integer 用到了享元模式来复用对象, 才导致了这样的运行结果。当我们通过自动装箱, 也就是调用 `valueOf()` 来创建 Integer 对象的时候, 如果要创建的 Integer 对象的值在 -128 到 127 之间, 会从 IntegerCache 类中直接返回, 否则才调用 `new` 方法创建。看代码更加清晰一些, Integer 类的 `valueOf()` 函数的具体代码如下所示:

```

public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}

```

实际上，这里的 IntegerCache 相当于，我们上一节课中讲的生成享元对象的工厂类，只不过名字不叫 xxxFactory 而已。我们来看它的具体代码实现。这个类是 Integer 的内部类，你也可以自行查看 JDK 源码。

```

/**
 * Cache to support the object identity semantics of autoboxing for values
 * between
 * -128 and 127 (inclusive) as required by JLS.
 *
 * The cache is initialized on first usage. The size of the cache
 * may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.
 * During VM initialization, java.lang.Integer.IntegerCache.high property
 * may be set and saved in the private system properties in the
 * sun.misc.VM class.
 */
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) - 1);
            } catch (NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;
        cache = new Integer[(high - low) + 1];
    }
}

```

```

    int j = low;
    for(int k = 0; k < cache.length; k++)
        cache[k] = new Integer(j++);
    // range [-128, 127] must be interned (JLS7 5.1.7)
    assert IntegerCache.high >= 127;
}
private IntegerCache() {}
}

```

为什么 IntegerCache 只缓存 -128 到 127 之间的整型值呢？

在 IntegerCache 的代码实现中，当这个类被加载的时候，缓存的享元对象会被集中一次性创建好。毕竟整型值太多了，我们不可能在 IntegerCache 类中预先创建好所有的整型值，这样既占用太多内存，也使得加载 IntegerCache 类的时间过长。所以，我们只能选择缓存对于大部分应用来说最常用的整型值，也就是一个字节的大小（-128 到 127 之间的数据）。

实际上，JDK 也提供了方法来让我们可以自定义缓存的最大值，有下面两种方式。如果你通过分析应用的 JVM 内存占用情况，发现 -128 到 255 之间的数据占用的内存比较多，你就可以用如下方式，将缓存的最大值从 127 调整到 255。不过，这里注意一下，JDK 并没有提供设置最小值的方法。

```

//方法一：
-Djava.lang.Integer.IntegerCache.high=255
//方法二：
-XX:AutoBoxCacheMax=255

```

现在，让我们再回到最开始的问题，因为 56 处于 -128 和 127 之间，i1 和 i2 会指向相同的享元对象，所以 i1==i2 返回 true。而 129 大于 127，并不会被缓存，每次都会创建一个全新的对象，也就是说，i3 和 i4 指向不同的 Integer 对象，所以 i3==i4 返回 false。

实际上，除了 Integer 类型之外，其他包装器类型，比如 Long、Short、Byte 等，也都利用了享元模式来缓存 -128 到 127 之间的数据。比如，Long 类型对应的 LongCache 享元工厂类及 valueOf() 函数代码如下所示：

```

private static class LongCache {
    private LongCache(){}
    static final Long cache[] = new Long[-(-128) + 127 + 1];
    static {
        for(int i = 0; i < cache.length; i++)
            cache[i] = new Long(i - 128);
    }
}

```



```

}
public static Long valueOf(long l) {
    final int offset = 128;
    if (l >= -128 && l <= 127) { // will cache
        return LongCache.cache[(int)l + offset];
    }
    return new Long(l);
}

```

在我们平时的开发中，对于下面这样三种创建整型对象的方式，我们优先使用后两种。

```

Integer a = new Integer(123);
Integer a = 123;
Integer a = Integer.valueOf(123);

```

第一种创建方式并不会使用到 IntegerCache，而后面两种创建方法可以利用 IntegerCache 缓存，返回共享的对象，以达到节省内存的目的。举一个极端一点的例子，假设程序需要创建 1 万个 -128 到 127 之间的 Integer 对象。使用第一种创建方式，我们需要分配 1 万个 Integer 对象的内存空间；使用后两种创建方式，我们最多只需要分配 256 个 Integer 对象的内存空间。

2、享元模式在 Java String 中的应用

刚刚我们讲了享元模式在 Java Integer 类中的应用，现在，我们再来看下，享元模式在 Java String 类中的应用。同样，我们还是先来看一段代码，你觉得这段代码输出的结果是什么呢？

```

String s1 = "楠老师";
String s2 = "楠老师";
String s3 = new String("楠老师");

System.out.println(s1 == s2);
System.out.println(s1 == s3);

```

上面代码的运行结果是：一个 true，一个 false。跟 Integer 类的设计思路相似，String 类利用享元模式来复用相同的字符串常量（也就是代码中的“小争哥”）。JVM 会专门开辟一块存储区来存储字符串常量，这块存储区叫作“字符串常量池”。上面代码对应的内存存储结构如下所示：

不过，String 类的享元模式的设计，跟 Integer 类稍微有些不同。Integer 类中要共享的对象，是在类加载的时候，就集中一次性创建好的。但是，对于字符串来说，我们没法事先知道要共享哪些字符串常量，所以没办法事先创建好，只能在某个字符串常量第一次被用到的时候，存储到常量池中，当之后再用到的时候，直接引用常量池中已经存在的即可，就不需要再重新创建了。

三、享元模式和单例、缓存、池化的区别

在上面的讲解中，我们多次提到“共享”“缓存”“复用”这些字眼，那它跟单例、缓存、对象池这些概念有什么区别呢？我们来简单对比一下。

我们先来看享元模式跟单例的区别。

在单例模式中，一个类只能创建一个对象，而在享元模式中，一个类可以创建多个对象，每个对象被多处代码引用共享。实际上，享元模式有点类似于之前讲到的单例的变体：多例。

我们前面也多次提到，区别两种设计模式，不能光看代码实现，而是要看设计意图，也就是要解决的问题。尽管从代码实现上来看，享元模式和多例有很多相似之处，但从设计意图上来看，它们是完全不同的。应用享元模式是为了对象复用，节省内存，而应用多例模式是为了限制对象的个数。

我们再来看享元模式跟缓存的区别。

在享元模式的实现中，我们通过工厂类来“缓存”已经创建好的对象。这里的“缓存”实际上是“存储”的意思，跟我们平时所说的“数据库缓存”“CPU 缓存”“MemCache 缓存”是两回事。我们平时所讲的缓存，主要是为了提高访问效率，而非复用。

最后我们来看享元模式跟对象池的区别。

对象池、连接池（比如数据库连接池）、线程池等也是为了复用，那它们跟享元模式有什么区别呢？

虽然对象池、连接池、线程池、享元模式都是为了复用，但是，如果我们再细致地抠一抠“复用”这个字眼的话，对象池、连接池、线程池等池化技术中的“复用”和享元模式中的“复用”实际上是不同的概念。

池化技术中的“复用”可以理解为“重复使用”，主要目的是节省时间（比如从数据库池中取一个连接，不需要重新创建）。在任意时刻，每一个对象、连接、线程，并不会被多处使用，而是被一个使用者独占，当使用完成之后，放回到池中，再由其他使用者重复利用。享元模式中的“复用”可以理解为“共享使用”，在整个生命周期中，都是被所有使用者共享的，主要目的是节省空间。