概述

> 代理模式为另一个对象提供一个替身或者占位符以控制对这个对象的访问

举一个生活中的例子：有时候我们想要买火车票，但是火车站太远，那么我们可以去附近的火车票代售点进行购买。此时，代售点就是代理，它拥有被代理对象的部分功能——售票功能。

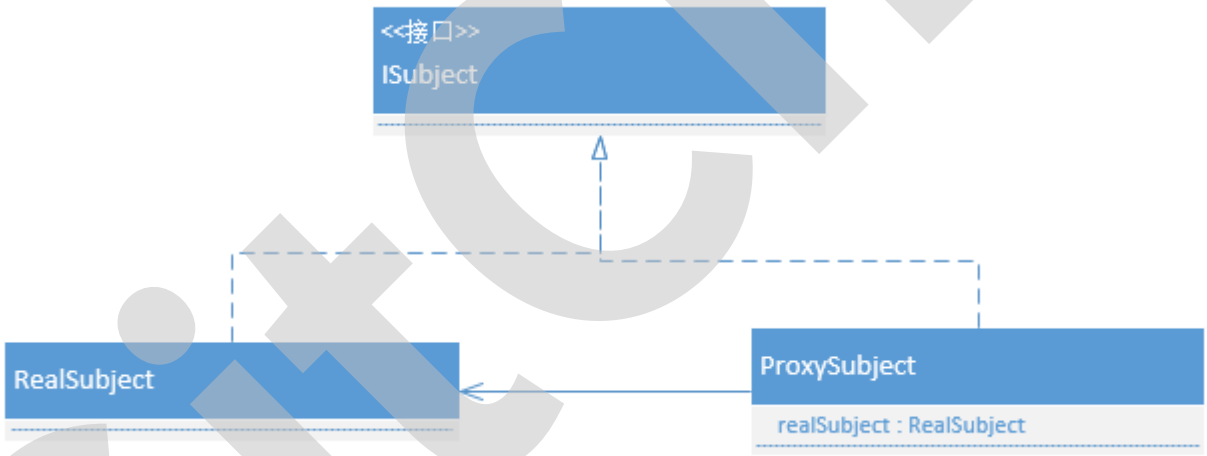
为什么需要代理？在生活中，我们去售票代理处买票是因为火车站太远了，但在代码的世界里可没有“太远”了这种说法。使用代理的基本目的是控制对真实对象的访问。

这里所说的“控制”并不是单纯地指权限控制，访问控制等，代理也可以为被代理的对象增加功能，例如使用Spring AOP给某方法执行前输出日志，aop也是一种代理。从这一点上来说代理模式和装饰模式有些类似，但是两者之间本质上还是有些区别的。

装饰模式主要是给对象增强功能。

代理模式是对访问对象加以控制。

代理模式类图：



代理模式中，代理类和被代理类都实现了同一个接口，并且在代理类中调用了被代理类，因此实际上我们最终调用的还是那个真实的对象，只不过由代理类帮我们调用而已。

下面我们就用代理买票的例子来写个简单的demo

代码示例

定义一个售票接口：

```
public interface TicketSell {
    void buyTicket();
}
```

车站售票类：

```
public class Station implements TicketSell {

    @Override
    public void buyTicket() {
        System.out.println("有人买了一张票");
    }
}
```

不从车站直接买票，而从代理点买票。我们这里模拟一下代理点有车票库存限制的场景：

```
public class ProxyStation implements TicketSell {

    private Station station;
    /**
     * 库存
     */
    private static Integer stock = 1;

    public ProxyStation(){
        station = new Station();
    }

    @Override
    public void buyTicket() {

        if(stock > 0){
            station.buyTicket();
            stock--;
        }else{
            throw new RuntimeException("库存不足");
        }

    }
}
```

在代理类中，我们既然可以做库存限制的功能，同理也能根据实际业务场景实现各种各样的功能。

调用的时候直接调用代理类：

```
TicketSell sell = new ProxyStation();
sell.buyTicket();
sell.buyTicket();
```

动态代理

在上面的例子中，其实可以称之为“静态代理”，静态代理有一个很明显的缺点：如果我们系统中有很多地方需要用到代理，那我们我们则需要写很多代理类。因此有时我们需要使用**动态代理**。

那么什么是动态代理？

其实很简单，在静态代理中，被代理的类是固定的，一个代理类对应一个被代理类。就像上面的例子中的 Station 类，其代理类为ProxyStation。而动态代理就是被代理的类是代码运行时指定的。

JDK为我们提供了一种动态代理的实现，通过实现 `InvocationHandler` 接口来实现动态代理。

控制库存的动态代理类：

```
public class StockHandler implements InvocationHandler {
    /**
     * 被代理类
     */
    private Object target;

    /**
     * 库存
     */
    private static Integer stock = 1;

    public StockHandler(Object target){
        super();
        this.target = target;
    }

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

        if(stock > 0){
            method.invoke(target,null);
            stock--;
        }else{
            throw new RuntimeException("库存不足");
        }

        return null;
    }
}
```

仔细观察上面的动态代理类，发现它的**被代理类** 使用了Object类型，而不是具体指定某一个类，在调用时再指定：

```
Station station = new Station();
InvocationHandler handler = new StockHandler(station);

Class<?> cls = station.getClass();

ClassLoader loader = cls.getClassLoader();

TicketSell sell = (TicketSell)Proxy.newProxyInstance(loader,cls.getInterfaces(),handler);

sell.buyTicket();
sell.buyTicket();
```

使用了动态代理类之后，如果还有其他地方需要用到同样逻辑的库存控制，就可以不用重复写代理类了。

JDK动态代理也有不足之处，它要求被代理类一定要实现某个接口，比如上面的 Station 类实现了TicketSell 接口。如果我们的类原本是没有实现接口的，总不能为了用代理而特意去给他加一个接口吧？

为了解决这个问题，我们可以使用cglib动态代理，它是基于类做的代理，而不是基于接口。这里不详细介绍，有兴趣的朋友可以自己度娘一下。

案例

延迟加载

在数据库查询操作中，经常要关联查询，例如查询User对象，一个User有多个Address，为了方便，我们希望在查User时直接关联查出Address。

下面简单地模拟一下：

Address类：

```
public class Address {

    private String addr;

    public Address(String addr){
        this.addr = addr;
    }

    public String getAddr() {
        return addr;
    }

    public void setAddr(String addr) {
        this.addr = addr;
    }

    @Override
    public String toString() {
        return "Address{" +
                "addr='" + addr + '\'' +
```

```
                '}';
    }
}
```

User类：

```
public class User {

    private Integer id;
    private String name;
    private List<Address> addressList;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Address> getAddressList() {
        return addressList;
    }

    public void setAddressList(List<Address> addressList) {
        this.addressList = addressList;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", addressList=" + addressList +
                '}';
    }
}
```

数据库查询类（此处直接给出数据模拟一下，并不真正查询数据库）：

```
public class Db {


    public User getUser(Integer id){
```

```
        User user = new User();

        user.setId(1);
        user.setName("tom");

        List<Address> addressList = new ArrayList<Address>();
        addressList.add(new Address("地址1"));
        addressList.add(new Address("地址2"));

        user.setAddressList(addressList);

        return user;
    }
}
```

调用：

```
Db db = new Db();
User user = db.getUser(1);
System.out.println(user);
```

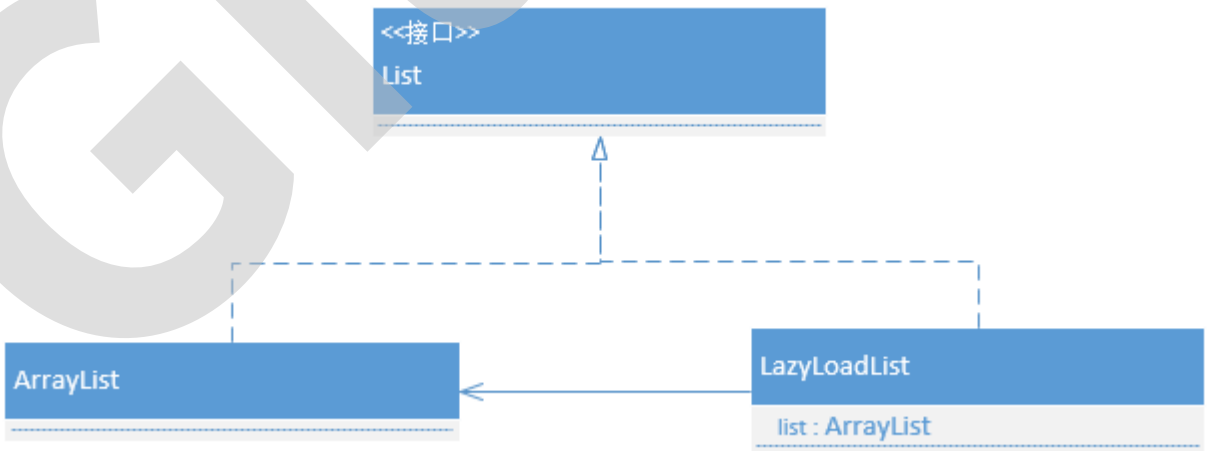
结果：

```
User{id=1, name='tom', addressList=[Address{addr='地址1'}, Address{addr='地址2'}]}
```

自动关联Address虽然给我们带来了很多便利，但是这种方式有一个缺陷：如果我们不需要Address，Db类也会去查询与User关联的Address。

我们希望的效果是"延迟加载"，当我们真正地去获取 address 时，才去查询关联的address。

要实现这个功能，我们需要写一个代理类，代理默认的ArrayList列表类：



代理类：（只给出一部分代码）

```
public class LazyLoadList<E> implements List {
```

```
    /**
     * 关联外键
     */
    private Integer key;

    private boolean hasSelect = false;

    public LazyLoadList(Integer key){
        this.key = key;
    }

    private List list = new ArrayList();

    /**
     * 真正查询数据库
     */
    private void doSelect(){
        //已查询过不用再查询
        if(hasSelect) return;

        System.out.println("开始查询");
        String sql = " select * from address where user_id = " + this.key;

        // sql查询

        //使用反射，注解等机制组装查询结果，此处模拟一下即可
        list.add(new Address("地址1"));
        list.add(new Address("地址2"));

        hasSelect = true;
    }

    @Override
    public int size() {
        this.doSelect();
        return list.size();
    }

    @Override
    public Object get(int index) {
        this.doSelect();
        return list.get(index);
    }
}
```

上面代理类中，`doSelect()` 方法才是真正查询数据库的操作。在客户端调用size()，get()等方法时，才是真正需要查询数据库的时候。

PS：实际开发中，这个代理类应该根据传入的泛型的实际类型来组装查询结果，而不能仅仅针对Address这个类。此处只是模拟一下。

修改Db类，将`new ArrayList`改成new我们自己定义的代理类`LazyLoadList`

```
public User getUser(Integer id){

    User user = new User();

    user.setId(1);
    user.setName("tom");

    List<Address> addressList = new LazyLoadList<Address>(id);

    user.setAddressList(addressList);

    return user;
}
```

测试：

```
Db db = new Db();
User user = db.getUser(1);

System.out.println(user);

List addressList = user.getAddressList();

System.out.println(addressList.get(0));
```

结果：

```
User{id=1, name='tom', addressList=com.itzhoujun.design.proxy.LazyLoadList@103dbd3}
com.itzhoujun.design.proxy.LazyLoadList@103dbd3
开始查询
Address{addr='地址1'}
```

从上面结果可以看到，addressList的类型是LazyLoadList代理类，当我们调用`addressList.get(0)`获取address时，才去数据库查询address，这样就实现了延迟加载。

总结

优点

扩展性强，对象更智能

缺点

代理类由于做了很多额外的操作，可能使请求速度变慢。