

期待的接口

张逸

www.agiledon.com

定义接口时需要注意什么？是实现，还是消费？窃以为，接口是抽象了的服务，服务的消费者只会关心服务能够提供什么，而不会考虑服务如何实现。例如在 ATM 机上取款，取款人只需要考虑怎样插入储蓄卡，怎么选择功能项，然后输入正确的密码和取款金额，再等待正确数额的钞票从机器中吐出，最后取走。至于内部的实现机制，则不在取款人的思考范畴。因此，接口必须符合调用者的期待，不然就会给设计带来障碍。接口的定义是为调用者准备的，接口具备的方法以及方法具备的签名，都必须站在调用者的角度来考虑。当调用者是测试用例时，这样的设计就变成了测试驱动设计。

例如编写一个银行账务管理系统，存取款服务的接口定义应该是这样：

```
public interface IBankService {
    bool Withdraw(Money money);
    bool Deposit(Money money);
}
```

在 IBankService 的实现类中，会通过调用一个 Account 类，实现存款和取款的功能：

```
public class Account {
    public bool Add(Money money) {
        //实现
    }
    public bool Substract(Money money) {
        //实现
    }
}

public class BankServiceImpl:IBankService {
    private Account m_account;
    public BankServiceImpl(Account account) {
        m_account = account;
    }
    public bool Withdraw(Money money) {
        try {
```

```
        m_account.Subtract(money);
        return true;
    } catch {
        return false;
    }
}

public bool Deposit(Money money) {
    try {
        m_account.Add(money);
        return true;
    } catch {
        return false;
    }
}
```

注意 `IBankService` 和 `Account` 的方法名，两者均实现了存取款功能，为何名称大相径庭？原因就在于对象的调用者并不相同。`IBankService` 暴露给 UI，实际上就代表了它是与存取款业务直接相关的。`Withdraw` 和 `Deposit` 的命名正好符合这样的逻辑。对于 `Account` 而言，表现出来的是帐户上的余额是增加或减少，它并不知道存取款的业务逻辑。

虽然设计者才是接口定义的主宰，然而调用者作为顾客，他才是真正的上帝。调用者说：“我希望使用这样的接口。”潜在的含义是，当我创建这样的实现类时，当我传递需要的输入实参时，你已经帮我考虑好了。调用者就像是守候在无人售货机前的顾客，选一罐可口可乐，然后按价塞入相应的钱币，就听到叮里咣当，最后滚出的一定是一罐可口可乐，而不是百事可乐。接口的设计者需要考虑调用者的感受，同时却不能对调用者做出任何假设。无人售货机如果只能接收五元和十元的纸币，就必须能够防止顾客放入错误的钱币。

一旦服务提供的接口并不符合调用者的期待，就存在一个“适配”的工作，这正是 `Adapter` 模式的意图。`Adapter` 对象是一个高明的调解人，负责将两个不协调的接口统一，既有效地保证了第三方接口对象的重用，又能够很好的支持服务的扩展。

虽然服务的定义者必须要符合调用者的期待，但反过来，定义者也给予了调用者一定的限制。此时，接口代表一种规约，它是对调用者进行了合理的限制。以 Java 的线程处理为例，就要求执行多线程逻辑的对象必须要实现 `Runnable` 接口，否则 `Thread` 的 `start()` 方法

就不能执行。

```
class MyThreadStart implements Runnable {
    public void run()
    { //执行相关操作 }
}
Thread controller = new Thread(new MyThreadStart());
controller.start();
```

如果方法要求传入的参数类型为抽象类型，则表明该方法的实现可能是变化的。这同样属于对接口的期待。好的设计不应该与具体类型耦合在一处，而是应该将创建具体对象的职责交由调用者去决定。“依赖注入”的方式正是基于这一点，例如 Order 实体对象的定义：

```
public interface IOrderRepository { }

public class Order {
    public Order(IOrderRepository repository) { }
}
```

Order 类的定义对 IOrderRepository 接口存在一个期待，即我们应该传入实现 IOrderRepository 接口的具体类对象，而不是其他类型。这种对接口的期待是开放的。例如，我们可以在单元测试的时候，考虑定义 MockOrderRepository 类去实现 IOrderRepository 接口，从而完成对真正的资源库对象进行模仿。

如果实现接口的类包含的公开方法比接口宽，就需要思考这样的设计是否合理。因为，这意味着这些公开方法对扩展是封闭的，违背了开放封闭原则。调用者在调用这样的类时，如果仍然采用多态的方式去调用，则需要对接口类型进行强制转换。例如：

```
public interface IConfigReader {
    string Read(string section);
}

public class XmlConfigHandler : IConfigReader {
    public string Read(string section) { }
    public void Write(string section,string value) { }
}

public class ConfigSettingManager {
    private string m_section;
    public ConfigSettingManager(string section) {
```

```
        m_section = section;
    }
    public void Config(IConfigurationReader reader) {
        string value = reader.Read(m_section);
        if (value != expectedValue) {
            ((XmlConfigHandler)reader).
                Write(m_section, expectedValue);
        }
    }
}
```

如上的设计是不协调的。Config()方法的实现破坏了程序结构的平衡与和谐。它带来两个问题。其一，方法的实现与期待不符。既然参数类型为 IConfigurationReader，则表明 Config方法期待对配置文件的读功能。那么，对写功能的调用就是不合理的。其二，方法引入了与 XmlConfigHandler 的具体依赖关系，从而让 IConfigurationReader 接口对可扩展性做出的努力付诸东流。

对上述设计的修改基于两种不同的策略，有两种不同的结果。如果对接口方法的期待更加细粒度，即希望分别对读操作和写操作进行区别对待，可以再定义一个 IConfigurationWriter 接口。如果 XmlConfigHandler 类需要实现 Write()方法，就可以实现 IConfigurationWriter 接口。这就对 Write()方法实现了抽象，使得它与 Read()方法能够处于相同的抽象层面。如果不需要做这样的区别对待，且读操作和写操作的变化方向与变化粒度是一致的，就可以将其定义为一个统一的接口，例如 IConfigurationHandler：

```
public interface IConfigurationHandler {
    string Read(string section) ;
    void Write(string section, string value) ;
}
```

所以，在通常情况下，我们不应该对传入的接口对象进行强制类型转换。接口是设计者对调用者的一种约束和控制，如果进行强制类型转换，说明设计者自己违背了这样的约束，丢失了对调用者的控制力。只有一种例外，即标记接口的使用。Uncle Bob 在《敏捷软件开发》中提出的 Acyclic Visitor 模式即使用了这样的标记接口，如下所示：

```
public interface ModemVisitor { }
public interface HayesVisitor {
    void visit(Hayes modem);
}
```

```
}  
public class Hayes {  
    public void accept(ModemVisitor v) {  
        try {  
            HayesVisitor hv = (HayesVisitor)v;  
            hv.visit(this);  
        } catch (){}  
    }  
}
```

标记接口通常被定义为空。它不是为了调用者的期待而定义，其意图是抽象，将那些不能抽象在一起的类，利用一个标记绑定起来，为其提供统一的接口。标记接口保证了调用方法的一致性。虽然强制类型转换会引入具体依赖，却不会有任何副作用，因为在方法实现中，设计者的期待本身就是要转换的类型。这里不存在扩展，如 Hayes 类中 accept() 方法的实现，它期待的只能是 HayesVisitor 类型。