第 1 章
设计模式的原则与分类

1.1 本章要点

本章主要对设计模式的六大原则（合成复用原则额外在 1.4 节说明）和设计模式的分类进行阐述，使读者能够从宏观角度对设计模式有一个全面的了解。对设计模式原则，以热门源码中的示例为引（如 JDK8 源码、Spring5 源码），再结合开发场景进行原则的讲解，未接触过源码的读者也无须担心，本章所引用的源码示例及开发场景都非常具有代表性、常规性，相信读者能够完全理解。本章内容要点如下。

- 单一职责原则。
- 接口隔离原则。
- 依赖倒置原则。
- 里氏替换原则。
- 迪米特法则（迪米特原则）。
- 开闭原则。
- 创建型设计模式。
- 结构型设计模式。
- 行为型设计模式。

可能此时部分读者会有所疑问，为何本章要点中没有提到“合成复用原则”？请读者不要着急，“合成复用原则”相关内容会在 1.4 节为大家进行说明，未纳入章节重点的原因也会在 1.4 节中说明，满足大家对“合成复用原则”相关内容的学习要求。

1.2 设计模式的原则

1.2.1 单一职责原则

引用百度词条对单一职责原则的定义：“单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，面向对象五个基本原则（SOLID）之一。它规定一个类应该只有一个发生变化的原因。该原则是 Robert C. Martin 于《敏

捷软件开发：原则、模式与实践》一书中给出的。Martin 表示此原则是基于 Tom DeMarco 和 Meilir Page-Jones 的著作中的内聚性原则发展出的。”

单一职责原则，强调的是职责的分离，一个类，只需要负责一种职责即可，一个类发生变化的原因，必然是所负责的职责发生变化。这听起来很抽象，干涩的文字描述可能会让经验稍浅的读者产生一定的困惑，可实际上，从我们接触代码的那一刻起，已经时时刻刻地在遵守单一职责原则。

- main 函数所在的类，作为程序的启动入口，职责单一。
- SpringBoot 框架的启动类是单一职责原则最完美的写照。
- 我们创建的 utils 工具类，对日期的处理一般会封装到一个 DateUtils.java 中，职责单一。即便我们将所有的工具类方法封装到一个统一的 CommonUtils.java 中，也依然遵循了单一职责原则，因为它的职责就是工具。
- 大家众所周知的 MVC 框架，提倡将接入层 Controller、服务层 Service、持久层 DAO 分别进行实现，划分不同的职责，也遵循了单一职责原则。

……

单一职责原则，是一个备受争议的原则，也是设计模式最为基础的原则。备受争议的原因是，每个人对职责划分的看法不一样，不同项目需求所面临的挑战和划分方式不一样。我们以一个简单的“用户注册和用户登录”需求为例，展开一场讨论，相信这个需求对读者来说是非常简单的，很快就能书写以下代码。

```
@Service
public class UserService {
    /**
     * 用户注册功能
     * @param parameters 此处为示例代码，实际参数根据业务需求而定
     * @return 此处为示例代码，实际返回根据业务需求而定
     */
    public Object register(String ... parameters) {
        return null;
    }

    /**
     * 用户登录功能
     * @param parameters 此处为示例代码，实际参数根据业务需求而定
     * @return 此处为示例代码，实际返回根据业务需求而定
     */
    public Object login(String ... parameters) {
        return null;
    }
}
```

这是一个非常完美的设计，UserService 类完全遵循了单一职责原则，不管是 register 注册功能还是 login 登录功能，都属于“用户操作”的相关职责，简单的注册和登录功能，用粗粒度的“用户操作”职责进行了正确的划分。

然而，如果登录功能需要满足多种第三方账号授权登录，注册功能需要进行短信、邮箱等动态码验证，金融相关软件还需要进行身份证验证和人脸识别等一系列附加功能，我们可能就要考虑将登录功能和注册功能以更加细粒度的职责进行划分。

（1）登录功能单独划分，提供默认登录方式（拥有本站账号的用户）和第三方账号的验证及登录功能。

```
@Service
public class LoginService {
    /**
     * 默认登录功能，使用本站用户名和密码登录
     * @param account
     * @param password
     * @return
     */
    public Object doDefaultLogin(String account, String password) {
        return null;
    }

    /**
     * 第三方账号授权登录
     * @param parameters 此处为示例代码，实际参数根据业务需求而定
     * @return 此处为示例代码，实际返回根据业务需求而定
     */
    public Object do3rdPartLogin(String ... parameters) {
        return null;
    }

    /**
     * 第三方账号授权验证
     * @param parameters 此处为示例代码，实际参数根据业务需求而定
     * @return 此处为示例代码，实际返回根据业务需求而定
     */
    private Object vaild3rdAccount(String ... parameters) {
        return null;
    }
}
```

（2）注册功能单独划分，分为注册功能以及注册过程中所需要的相关验证功能如下。

```
@Service
public class RegisterService {
    /**
```

```
     * 注册功能
     * @param parameters
     * @return
     */
    public Object register(String ... parameters) {
        return null;
    }

    /**
     * 短信验证
     * @param phoneNum
     * @return
     */
    public Object phoneCodeSend(String phoneNum) {
        return null;
    }

    /**
     * 邮箱验证
     * @param emailAddress
     * @return
     */
    public Object mailCodeSend(String emailAddress) {
        return null;
    }
}
```

这样的设计也是非常正确的，随着业务的复杂程度越来越高，每一个细粒度的职责都可能拥有非常复杂的逻辑，那么我们就要考虑是否可以按照项目需求进行更加细粒度的职责拆分，从而保证单一职责原则下的低耦合度。当然，部分读者会认为，是否可以将“手机验证功能”和“邮箱验证功能”再次进行单一职责的划分？当然可以，但是要避免过度的职责拆分，要根据项目需求的复杂程度进行合理的规划。

最后，我想对广大读者说的是，单一职责原则的划分没有正确与错误之分，每个开发者都有自己的考量角度和划分方式。一切都应该以项目实际情况为出发点进行设计，因地制宜才是我们真正需要遵守的原则。

1.2.2 接口隔离原则

对于接口隔离原则（Interface Segregation Principle，IPS），单从“隔离”这个字眼来说，我们很容易错误地理解接口隔离原则。因为接口“隔离”意味着接口“划分”，那么“划分”是不是就是指职责划分？职责划分是不是就是指上一小节所叙述的“单一职责原则”？

其实，单一职责原则是接口隔离原则的基础，单一职责原则注重职责的划

分，从职责角度进行类和接口的划分；在此基础上，接口隔离原则登场，注重接口使用的“精确性”和“最小化”。如果读者依然没有理解两者的区别和关系，请不要着急，可以继续往下阅读，我会以 JDK 源码为例，为读者进行更明确的解释。接口隔离原则，主要体现在以下两个方面。

（1）不要使用没有任何依赖关系的接口。

肯定有读者会认为这是一句空话，而且这句话还泛着一点傻气。正常开发过程中，如果不依赖某个接口，肯定是没有人使用的，但是随着业务代码的累加，也难免会出现这样的问题，即使是 JDK 中数据结构源码的作者，也依然在不经意间使用了无须使用的接口。接下来我们对 JDK 源码的阐述，纯粹从技术角度进行展开。

我们先来看下以下代码执行逻辑。

```
public static void main(String[] args) {
      List<Object> list = Collections.emptyList(); //java.util.
Collections;
    list.add(new Object()); // 此处会报错
}
```

当我们执行这个 main 函数的时候，会收到以下报错信息。

```
Exception in thread “main” java.lang.UnsupportedOperationException
       at java.util.AbstractList.add(AbstractList.java:148)
       at java.util.AbstractList.add(AbstractList.java:108)
```

从以上的报错信息我们能够清楚看到，是 add 方法发生了异常，因为 Collections 通过 emptyList() 方法创建的空集合，是不能够进行元素添加的。可是，当我们跳转到源码看 EmptyList 的类结构时，却发现了 RandomAccess 接口的存在，而 RandomAccess 接口真的不应该出现，它是一个无用的接口实现。

```
private static class EmptyList<E>
    extends AbstractList<E>
    implements RandomAccess, Serializable {
...}
```

为什么说上边源码中的 RandomAccess 是无用的接口实现呢？大部分读者可能都知道，RandomAccess 接口是一个标志接口（Marker），只要类 List 集合实现这个接口，就能支持快速随机访问。可是当前的 EmptyList，从对象的创建到灭亡，都不会有任何元素的添加，没有元素又何谈元素的快速随机访问呢？

如果单从接口隔离原则上来讲，这个 RandomAccess 接口，完全没有存在的必要，这是源码作者的一个疏忽。当然，此处的 RandomAccess 的使用，也可

能是为了后续对 EmptyList 对象的扩展，支持元素添加功能，只需要实现父类 AbstractList 中的 add 方法即可，那么 RandomAccess 也就自然有了它的用武之地。

我们继续分析，这部分源码首先遵循了单一职责原则，RandomAccess 作为快速随机访问的标志类，就是一个明确的职责划分，符合单一职责原则；而接口隔离原则在此处的作用是告诉大家：RandomAccess 当用时则用，不当用时勿用。

（2）一个类对另外一个类的依赖性应当是建立在最小的接口上的。

一个接口代表一个角色，不应当将不同的角色都交给一个接口。如果将没有关系的接口合并在一起，形成一个臃肿的大接口，那并不是一个好的设计，违背了接口隔离原则。我们以 JDK 的 ArrayList 源码设计进行说明，体会接口隔离原则的意义。我们来看以下 ArrayList 类结构：

```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.
io.Serializable
{
...
}
```

从上边的类结构代码中能够很明显地看到，ArrayList 有四种特质：

①集合特质：因为实现了 List 接口。

②快速随机访问：因为实现了 RandomAccess 接口。

③支持克隆：因为实现了 Cloneable 接口。

④支持序列化：因为实现了 Serializable 接口。

这样的类结构，首先是遵循了单一职责原则，按职责划分为不同的接口；然后在此基础上，接口隔离原则登场，细化接口的方法，保持接口的纯洁性，在满足需求的前提下，尽量减少接口的方法，做到专业、精确、最小化接口。

当然，我们平时工作过程中，已经不知不觉中遵循了接口隔离原则，不使用没有依赖关系的接口、接口细化、剔除无用方法等，都是接口隔离原则很好的实践。还是那句话，一切都应该以项目实际情况为出发点进行设计，因地制宜才是我们真正需要遵守的原则。

1.2.3 里氏替换原则

里氏替换原则（Liskov Substitution Principle，LSP）最初由 Barbara Liskov 在 1987 年的一次学术会议中提出，里氏替换原则是一种针对子类和父类关系的设计原则。在我们工作和学习过程中，会经常接触到子类与父类，里氏替换原则早已被我们潜移默化地使用了，并不是什么新鲜事，有基础的读者可以快速浏览或

跳过本小节内容。下面我们对该原则进行更加细致的说明。

（1）子类需要实现父类中所有的抽象方法（为实现“替换”做好准备）。

看到这里的读者可能会微微一笑，子类不实现父类的抽象方法，连开发工具（如 IDEA、Eclipse 等）都不同意，开发工具会自动提示我们进行抽象方法的覆写。那么，为实现“替换”做好准备如何理解呢？

我们先来分别看看 ArrayList、LinkedList 和 AbstractSequentialList 的类结构。

```
 /**
   ArrayList 类结构
   */
public class ArrayList<E> extends AbstractList<E>
           implements List<E>, RandomAccess, Cloneable, java.
io.Serializable{
...
}
 /**
   LinkedList 类结构
   */
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable {
...
}
/**
   AbstractSequentialList 类结构
   */
public abstract class AbstractSequentialList<E> extends AbstractList<E> {
...
}
```

我们可以看到，ArrayList 和 LinkedList 都属于 List 接口的子类，都属于 AbstractList 抽象类的子类（虽然 LinkedList 中间还有一个 AbstractSequentialList 的父类，但在整个继承链上依然是 AbstractList 的子类）。现在我们基于 ArrayList 和 LinkedList 书写以下代码来展示“替换”的精妙之处。

```
public class Demo {
    public static void main(String[] args) {
        ArrayList arrayList = new ArrayList();
        LinkedList linkedList = new LinkedList();
        addElement(arrayList, 1);
        addElement(linkedList, 1);
    }

     //方法的第一个参数为 List 类型，父类类型，依然可以支持传入 ArrayList 和
LinkedList 子类类型，此处体现了“替换”思想
    public static void addElement(List list, Object element) {
```

```
        list.add(element);
    }
}
```

通过以上代码，我们看到了，addElement 方法的第一个参数虽然为父类 List 类型，但可以支持传入任何 List 类型的子类型。这并不是什么新的伎俩，我们刚刚接触 JavaSE 对象多态的学习中，就可能已经这样做了，只不过没有那么高级的修饰（里氏替换原则）词罢了。

（2）子类可以加入自己的特有方法及属性。

龙生九子，九子各不同，但都是龙族的血脉。如果子类没有自己的特色，与父类完全一样，那我们何必多此一举进行继承呢？这部分的知识是非常容易理解的，LinkedList 中有 addFirst 方法而 ArrayList 却没有，这就是 LinkedList 作为子类的一个独特属性。

总以 JDK 源码进行说明多少会有些乏味，这次，我们以 Spring 源码中 BeanFactory 为例进行说明。

```
public interface BeanFactory {
// 获取 Bean
    Object getBean(String name) throws BeansException;
// 获取 Bean Provider
    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);
// 判断 Bean 是否存在
    boolean containsBean(String name);
// 是否为单例
    boolean isSingleton(String name) throws NoSuchBeanDefinitionException;
// 是否为多例
    boolean isPrototype(String name) throws NoSuchBeanDefinitionException;
//Bean 类型是否匹配
    boolean isTypeMatch(String name, ResolvableType typeToMatch) throws
NoSuchBeanDefinitionException;
    // 获取 Bean 类型
    Class<?> getType(String name) throws NoSuchBeanDefinitionException;
// 获取别名
    String[] getAliases(String name);
}
```

BeanFactory 作为最顶端（最基础）的父类接口，仅仅包含了对 Bean 的获取、类型获取及判断等相关方法，随着我们对 Spring 的使用越来越深入，这些方法肯定是不能够满足我们日常使用的。那么我们来看看作为 BeanFactory 子类，Spring 中 Bean 加载的核心类——DefaultListableBeanFactory 中的方法，如图 1-1 所示。

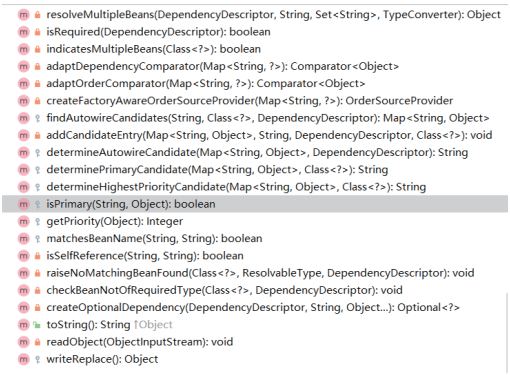

图 1-1

我们可以看到，DefaultListableBeanFactory 中有了更加细节的方法，比如方法 isPrimary（String，Object），它关注 Bean 上的 @Primary 注解；再比如 getPriority（String，Object）方法，它关注 Bean 上的 @Priority 注解。作为子类的 DefaultListableBeanFactory 有了自己独有的方法，为使用者提供了更加广阔的平台（很遗憾这不是一本解读 Spring 源码的书籍，所以不能过多展开 Spring 源码的讲解，我相信未来会有机会为大家进行 Spring 源码的讲解）。

（3）关于子类覆盖父类已实现方法（父类非抽象方法）的讨论。

相信部分读者在一些博客论坛上看到过这样一句话：“子类覆盖父类已实现方法，可以放大方法入参的类型”。笔者认为，这句话失之偏颇，并不是对里氏替换原则的正确解读。为了避免读者被此观点误导，请允许我首先以非源码的示例对此观点进行描述，因为笔者翻阅了大量源码，没有找到能够印证此观点的源码示例。

一些资料上进行了以下代码示例。

```
public class Demo {
    public static void main(String[] args) {
// 创建父类对象，执行方法，入参为 ArrayList
        BaseClass baseClass = new BaseClass();
        baseClass.process(new ArrayList());
// 创建子类对象，执行方法，入参为 ArrayList
        SubClass subClass= new SubClass();
        subClass.process(new ArrayList());
    }
}
// 父类
class BaseClass {
    public void process(ArrayList list) {
        System.out.println("BaseClass take process !");
    }
```

```
}
// 子类
class SubClass extends BaseClass {
    public void process(List list) {// 子类方法放大了参数类型
        System.out.println("SubClass take process !");
    }
}
```

我们可以看到，子类 SubClass 的 process 方法放大了参数类型，采用 List 接口为入参类型；父类 BaseClass 的 process 方法入参为 ArrayList 类型。当我们运行 main 函数时，发现无论是父类执行方法还是子类执行方法，得到的结果都是打印了“BaseClass take process !”。按照这个观点来说，子类对象可以完美地“替换”父类对象，而不会导致结果的改变。乍一看，好像很有道理，没什么问题，可是大家有没有想过以下问题。

- 我们 new SubClass() 对象是为了什么？不就是希望使用 SubClass 中的方法吗？然而因为参数的放大，即便是使用 SubClass 对象调用方法，还是依然会执行父类的逻辑，这叫“替换”吗？
- 我们在 SubClass 对象中创建同样的方法是为了什么？不就是为了能够与父类 BaseClass 的方法执行逻辑有所不同吗？然而因为参数的放大，SubClass 中的方法无法执行，总是执行父类 BaseClass 的方法，这叫“替换”吗？
- 我们为什么要在子类里放大参数类型？正确的父子关系，不都是应该父类采用范围更大的类型（泛型 T），然后子类定义确切的类型吗？难道仅仅是为了展现所谓的“替换”思想刻意为之吗？既然使用子类对象调用方法，就是为了使用子类的方法，而不是为了达到所谓的“替换”，进而埋没了子类的方法。而现实开发之中，我们真正创建对象的时候，也都是创建的子类对象，创建 List 对象，大部分会选用子类 new ArrayList()，而不是创建父类对象。

以上的代码示例，仅仅是一个方法“重载”（方法名称一致，返回值一致，方法入参不一致）的障眼法，为什么两次打印结果一致的根本原因是方法入参都是 new ArrayList() 呢？在 JVM 中，“重载”是“静态分派”的经典实现，方法参数的“静态类型”在编译期已经确定了，由于“静态类型”在编译期可知，所以在编译阶段，Javac 编译器就会根据参数的“静态类型”决定了使用哪个重载版本，所谓的子类参数类型放大而演示出来的效果，无非就是借助了 JVM 的“静态分派—重载”。

那我们应该如何覆写父类中已经实现的方法呢？很简单，使用 @Override，保持方法名称、返回值和方法参数一致即可。就好比我们在覆写 Object 类中的 hashCode 方法，代码如下：

```
public class Demo {
    @Override
    public int hashCode() {
        // 此处为示例代码，根据需求计算 hashCode
        return 0;
    }
}
```

此外，我依然想为读者引入 Spring 源码的示例，一是为了再次印证如何正确地覆写父类方法，二是为了尽最大可能为读者提供更多的扩展内容。我们来看看 Spring 容器对 BeanDefinition 注册的设计。

首先定义了父类 interface BeanDefinitionRegistry，并添加抽象方法 void registerBeanDefinition 供子类实现，代码如下：

```
public interface BeanDefinitionRegistry extends AliasRegistry {
        // 定义抽象方法
void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
      throws BeanDefinitionStoreException;
...
}
```

再来看看 Spring 容器相关子类 GenericApplicationContext 对这个方法的实现，直接采用 @Override 注解进行，对方法整体结构没有任何修改。除此之外，GenericApplicationContext 作为 Spring 的 ApplicationContext 相关类，也完美地完成了 DefaultListableBeanFactory 的初始化工作，通过无参构造进行初始化，成为了连接 Bean 工厂 DefaultListableBeanFactory 和 BeanDefinition 注册的容器（容器也称为上下文），代码如下：

```
public class GenericApplicationContext extends AbstractApplicationContext
implements BeanDefinitionRegistry {
     // 初始化核心 Bean 工厂：DefaultListableBeanFactory，为 beanDefinition
注册到 Bean 工厂之中做好了准备
private final DefaultListableBeanFactory beanFactory;
public GenericApplicationContext() {
        this.beanFactory = new DefaultListableBeanFactory();
    }

// 直接使用 @Override 实现父类的抽象方法，不改变父类方法的结构
@Override
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
   throws BeanDefinitionStoreException {
   this.beanFactory.registerBeanDefinition(beanName, beanDefinition);
   }
...
}
```

到这里依然没有结束，上边的源码中展现了 GenericApplicationContext 容器类对 Bean 工厂的初始化，所有的 BeanDefinition 最终都会注册到 Bean 工厂，那么我们来看看 Bean 工厂 DefaultListableBeanFactory 的源码，依然是使用 @Override 注解，不改变方法结构，覆写 registerBeanDefinition 方法，完成最终 BeanDefinition 的 register，代码如下：

```
public class DefaultListableBeanFactory extends AbstractAutowireCapableB
eanFactory
        implements ConfigurableListableBeanFactory,
BeanDefinitionRegistry, Serializable {
// 将所有的 BeanDefinition 注册到这个 map 数据结构之中
private final Map<String, BeanDefinition> beanDefinitionMap = new
ConcurrentHashMap<>(256);

// 直接使用 @Override 实现父类的抽象方法，不改变方法的结构
@Override
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
      throws BeanDefinitionStoreException {
...
this.beanDefinitionMap.put(beanName, beanDefinition);
...
}
...
}
```

无论从实际开发角度还是从源码角度进行印证，正确的子类覆盖父类非抽象方法的途径就是在不改变整体方法结构的前提下直接进行 @Override。所谓的子类方法扩大方法入参类型，根本没有任何落地的实际意义。

1.2.4 依赖倒置原则

依赖倒置原则（Dependence Inversion Principle，DIP），指的是程序要依赖于抽象接口，不要依赖于具体实现。简单地说就是要求对抽象进行编程，不要对实现进行编程。更简单的解释，就是“面向接口编程”。

面向接口编程，相信大家都比较熟悉这个概念，也理解其中的意义。继续以 1.2.3 小节的 Spring 源码示例进行说明，我们可以看到容器类 GenericApplicationContext 和 Bean 工厂核心类 DefaultListableBeanFactory 都实现了父类 BeanDefinitionRegistry 中的 registerBeanDefinition 方法。父类 BeanDefinitionRegistry 就是接口，我们要面向接口编程，就是要面向 BeanDefinitionRegistry 编程。那么 Spring 源码中哪里能体现出面向 BeanDefinitionRegistry 编程呢？带着这个疑惑，我们看看，是谁调用了 registerBeanDefinition 方法，以及是如何调用的，代码如下：

```
public abstract class BeanDefinitionReaderUtils {
public static void registerBeanDefinition(
      BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
      throws BeanDefinitionStoreException {
       String beanName = definitionHolder.getBeanName();
// 请看下 registry 的类型是什么？没错，是接口 BeanDefinitionRegistry，而不是具体子类
       registry.registerBeanDefinition(beanName,
       definitionHolder.getBeanDefinition());

}
}
```

请看以上代码为大家提供的注释，我们可以看到 registry 的类型是 BeanDefinitionRegistry 接口，而不是具体的容器子类 GenericApplicationContext 和 Bean 工厂子类 DefaultListableBeanFactory。面向接口 BeanDefinitionRegistry 进行方法的调用，就是面向 BeanDefinitionRegistry 编程，即面向接口编程。

关于依赖倒置原则，也有一些其他观点的补充，如高层模块不应该依赖低层模块，都应该依赖它们的抽象、细节依赖抽象等，其实都是对“面向接口编程”的另外一种表达方式，这里就不再过多地说明。

笔者总结了这样一句话供大家玩味，或许有助于大家对以上四个原则的认识和理解：“单一职责原则以职责为基准划分类和接口；划分出来的接口需要最小化，剔除无用接口方法，在接口隔离原则下进行精确的使用；子类对父类的实现需要依据里氏替换原则，在实现所有抽象方法的前提下，可以增加个性化功能，至此子类和父类就创建完成了；当我们使用该类时，要面向接口编程，遵循依赖倒置原则。”让我们继续对剩余的原则进行说明，然后在 1.2.6 小节对这句话进行最后的补充。

1.2.5 迪米特法则

迪米特法则（Law of Demeter，LOD），又叫作最少知识原则（The Least Knowledge Principle），一个类对于其他类知道得越少越好。简单来说就是只暴露方法入口，而实现细节不需要暴露给调用者。

干涩的文字描述总是不能给读者带来清晰的认知，在本书后续项目落地设计模式实战的“多种类第三方支付”章节，会为大家带来“策略模式 + 门面模式 + 工厂模式 + 享元模式”的落地实战。调用者只需要关心支付类型（如支付宝支付、微信支付等），无须关心具体的支付交互细节，“门面模式”就是一种遵循迪米特法则的、极具代表性的实现。

关于项目实战落地，我们后续章节细细道来。在这里，我更想和大家聊聊

Spring 源码中哪部分的设计是最具代表性的、最能体现遵循迪米特法则的设计。我们先来回忆一下 Spring 容器使用的简单示例，代码如下：

```
public class BeanTest {
   public static void main(String[] args) {
// 可以通过 XML 配置文件，创建 Spring 容器
      ApplicationContext contextA =
            new ClassPathXmlApplicationContext("spring-config.xml");
// 也可以通过配置类 @ComponentScan 配置扫描路径，创建 Spring 容器
      AnnotationConfigApplicationContext contextB =
            new AnnotationConfigApplicationContext(AppConfig.class);

// 两种容器的创建方式，都可以成功地加载我们注入的 Bean
      System.out.println(contextA.getBean(TestBean.class).getName());
      System.out.println(contextB.getBean(TestBean.class).getName());
   }
}
```

以上代码是 Spring 容器使用最为基础的示例，相信接触过 Spring 框架的读者对此都了然于心。从代码的书写过程中可以看到，想要使用 Spring 框架，我们仅仅关心以下三点即可。

① XML 配置文件的创建或者标有 @ComponentScan 配置类的创建及注解使用。

② ApplicationContext 的构造函数及入参。

③ getBean 方法的使用。

作为 Spring 框架的使用者，我们仅仅需要知道以上三点就可以了，可是 Spring 源码为我们做了很多。以注解容器 AnnotationConfigApplicationContext 的创建方式为例（注解方式是趋势，简单方便，避免对复杂 XML 配置文件的维护管理），我们来看一下 Spring 源码为我们做了什么，先来看容器创建的入口——AnnotationConfigApplicationContext 的构造函数，代码如下：

```
public class AnnotationConfigApplicationContext extends
GenericApplicationContext implements AnnotationConfigRegistry {
public AnnotationConfigApplicationContext(Class<?>...
componentClasses) {
       this();
       register(componentClasses);
       refresh();
    }
}
```

构造函数很清晰，包含三行代码，接下来我们分别对这三行函数所包含的内容进行说明。

（1）this() 方法。

this() 方法调用了无参构造函数，我们来看看无参构造函数做了些什么，代

码如下：

```
public AnnotationConfigApplicationContext() {
   this.reader = new AnnotatedBeanDefinitionReader(this);
   this.scanner = new ClassPathBeanDefinitionScanner(this);
}
```

依然是很清晰的逻辑，表面看无非是创建了两个对象，但是这两行代码实际上为我们做了很多重要的底层逻辑，很抱歉不能够更深层次地进行源码的展示，但笔者会尽最大可能为大家解释这两行代码所完成的功能，如果你阅读过 Spring 源码必然会产生共鸣，如果你即将要阅读 Spring 源码必然会有所帮助，当然，你最终也会深刻地体会到 Spring 的这部分源码设计是如何遵循迪米特法则的。接下来我们来介绍以下这两行代码的功能。

①创建读取 Annotation 注解下的 BeanDefinition 阅读器，为 Bean 工厂添加比较器依赖，如 @Order 注解、@Priority 注解，为 Bean 工厂添加延时加载依赖，如 @Lazy 注解。

②注册 Spring 依赖的 BeanDefinition，如内部注解配置处理器 internalConfigurationAnnotationProcessor、内部依赖注入处理器 internalAutowiredAnnotationProcessor 等。

③初始化 Spring 运行的 Environment，如 SystemProperties、systemEnvironment。

④基于“初始化子类之前需提前初始化父类”的规则，隐性地在父类 GenericApplicationContext 的无参构造中初始化 Bean 工厂核心类 —— DefaultListableBeanFactory。

⑤创建 classPath 的包扫描器，为后续自定义的 Bean 提供扫描功能。

这些都是我们无须关心的，各种 Bean 对象的读取、注册、扫描，甚至运行环境的初始化我们都无须关心。最小知识原则就体现在这里。

（2）register(componentClasses) 方法。

我们先来看看该方法的源码，代码如下：

```
public void register(Class<?>... componentClasses) {
    Assert.notEmpty(componentClasses, "At least one component class must
be specified");
   this.reader.register(componentClasses);
}
```

我们可以看到，该方法依靠我们在 this() 方法中创建的 reader 进行配置类的注册，最终会通过我们在“依赖倒置原则”所介绍的 registerBeanDefinition 方法进行 BeanDefinition 的注册，此处就不再过多地展开描述。

（3）**refresh() 方法**。

这个方法是 Spring 容器初始化的核心，这个方法涉及的内容十分丰富，我们来看看它都做了些什么，代码如下（方法功能介绍请参见注释）：

```
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // 容器刷新准备
        prepareRefresh();
        // 获取并配置 Bean 工厂
            ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();
        prepareBeanFactory(beanFactory);
        try {
            // 后置处理器的扩展，执行及注册
            postProcessBeanFactory(beanFactory);
            invokeBeanFactoryPostProcessors(beanFactory);
            registerBeanPostProcessors(beanFactory);
            // 初始化消息组件，如国际化和消息解析
            initMessageSource();
            // 初始化事件派发器
            initApplicationEventMulticaster();
            // 子类扩展接口，SpringBoot 使用了该扩展创建了内嵌的 Web 容器
            onRefresh();
            // 注册监听器 .
            registerListeners();
            // 加载无须延迟加载的单例 Bean
            finishBeanFactoryInitialization(beanFactory);
            // 结束容器刷新
            finishRefresh();
        } catch (BeansException ex) {
            ...
        } finally {
            ...
        }
    }
}
```

试想一下，如果 Spring 没有为我们做以上的功能，那么监听器要自己实现，消息处理器要自己实现，事件分发要自己考虑，这将是非常可怕的事情。Spring 仅为使用者暴露了有限的、简单的方法调用入口，而将其他所有的功能隐藏起来，做到了最少知识原则，符合迪米特法则。

1.2.6 开闭原则

开闭原则中，“开”是指对扩展开放，“闭”是指对修改关闭。简单来说，开闭原则就是指：如果你想要修改一个功能，请不要直接进行内部的代码修改，而是使用扩展的方式进行。如果读者对此原则感觉模糊，也不必着急，在我们后续

项目实战章节“日志的解析处理”部分，会通过“模板方法模式”进行项目的实战落地，而模板方法模式就是开闭原则的典型设计模式之一。

按照笔者的习惯，我们依然要以源码为示例，对开闭原则的应用示例进行说明。开闭原则的源码应用非常广泛，我们可以基于上节内容涉及的 Spring 容器核心 refresh() 方法中的后置处理器扩展方法 postProcessBeanFactory(beanFactory) 进行说明，想要实现不同功能的后置处理器，不需要修改 Spring 本身的代码（对内修改关闭），只需要子类覆写 postProcessBeanFactory(beanFactory) 方法进行后置处理器的定义即可（对外扩展开放）；也可以基于 refresh() 方法中的 onRefresh() 方法进行说明，引入 SpringBoot 内嵌 Web 容器的源码实现，对 SpringBoot 源码中的 ServletWebServerApplicationContext 类进行分析，体会 Spring 对 SpringBoot 的扩展开放；还可以基于 JDK 的 HashMap 源码为其子类 LinkedHashMap 提供的 afterNodeAccess、afterNodeInsertion、afterNodeRemoval 三大扩展方法进行说明，体会 HashMap 的对内修改关闭，以及对外（LinkedHashMap）开放扩展的设计等。

但是，经过笔者的再三考虑，我想为大家提供更加热门的、有关并发编程的 AQS 源码示例来阐述开闭原则的“对外扩展开放”与“对内修改关闭”。

相信大家对 AQS 并不陌生，它的全称是 AbstractQueuedSynchronizer，它是位于 JDK 源码 Java 并发包中的一个抽象类，此类基于 FIFO（First In - First Out，先进先出）队列，提供了实现锁和线程同步的框架，我们开发过程中经常使用到的 ReentrantLock 就是基于 AQS 抽象类进行的锁的实现。

前文提到过模板方法模式是开闭原则的代表性实现，AQS 也是基于模板方法模式的设计，我们先来看看 AQS 中“对外扩展开放”的源码片段，代码如下：

```
// 子类可以根据需要进行加锁（排它锁）逻辑的设计
protected boolean tryAcquire(int arg) {
    throw new UnsupportedOperationException();
}
// 子类可以根据需要进行释放锁（排它锁）逻辑的设计
protected boolean tryRelease(int arg) {
    throw new UnsupportedOperationException();
}
// 子类可以根据需要进行加锁（共享锁）逻辑的设计
protected int tryAcquireShared(int arg) {
    throw new UnsupportedOperationException();
}
// 子类可以根据需要进行释放锁（共享锁）逻辑的设计
protected boolean tryReleaseShared(int arg) {
    throw new UnsupportedOperationException();
}
```

```
// 判断当前线程是否获取到了锁
protected boolean isHeldExclusively() {
    throw new UnsupportedOperationException();
}
```

我们可以看到，AQS 源码中提供了五个需要扩展的方法，这些方法都是同步锁最核心的逻辑：加锁、释放锁、是否持有锁判断。如此核心的逻辑可以任由子类进行个性化扩展，这就是所谓的“对外开放扩展”。

那么，“对内修改关闭”从何处体现呢？ AQS 将以上五个方法，嵌入到了统一的代码模板中，虽然支持子类个性化的扩展核心逻辑，但是整体执行流程是不可以进行修改的，为了说明“对内修改关闭”，我们先来看以下源码片段。

```
public final void acquire(int arg) {
//tryAcquire 方法调用点
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}

public final boolean release(int arg) {
//tryRelease 方法调用点
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

从以上代码我们可以清晰地看到，tryAcquire 方法的调用时机和 tryRelease 方法的调用时机，都已经被 AQS 实现了，AQS 对整体的调用逻辑和时机进行了全面的把控，子类无法进行修改，这就是“对内修改关闭”的写照。

笔者经常把 AQS 开闭原则的设计比喻成生老病死的自然规律，芸芸众生，从呱呱坠地到归于尘土，这一自然规律无法修改，这就是“对内修改关闭”；然而，众生万象，都有各自的轨迹，不同的选择造就了不同的命运轨迹，这就是“对外扩展开放”。无论子类如何扩展，终究难免归于尘土。

我们继续补充笔者总结的那句话：“单一职责原则以职责为基准划分类和接口；划分出来的接口需要最小化，剔除无用接口方法，在接口隔离原则下进行精确的使用；子类对父类的实现需要依据里氏替换原则，在实现所有抽象方法的前提下，可以增加个性化功能，至此子类和父类就创建完成了；当我们使用该类时，要面向接口编程，遵循依赖倒置原则；不同的类或接口之间，要遵循最小知

识原则，减少互相的耦合，遵循迪米特法则；如果你的设计需要修改，请尽量提升代码的扩展性，追求对内修改关闭，对外开放扩展的开闭原则。”

设计模式的原则，固然是无数先辈程序员留给我们最好的财富，我们会认真品味、吸收，融会贯通到实际的开发场景之中。但规则是死的，人是活的，我们依然要以具体的业务流程和需求场景为出发点进行代码的设计，因地制宜，没有最优的，只有最合适的。

1.3 设计模式的分类

设计模式的分类是相对简单的概念知识。根据设计模式不同的特点和目的进行划分，设计模式可分为“创建型模式”“结构型模式”和“行为型模式”。本节为概念性知识点，建议读者快速浏览。

1.3.1 创建型模式

创建型模式以“是否创建对象”为依据进行划分，创建型设计模式在项目开发中的使用非常广泛，主要应用于对象创建的场景。包含以下五种设计模式。

- 工厂方法模式。
- 抽象工厂模式。
- 单例模式。
- 建造者模式。
- 原型模式。

1.3.2 结构型模式

结构型模式，更加注重类或对象的结合方式，将类或对象进行结合，形成一个更大的结构，在该结构下，不同的组件扮演不同的角色。例如，代理模式体现了代理类与被代理类的结构关系；适配器模式体现了适配器与被适配对象的结构关系。结构型模式包含以下七种设计模式。

- 适配器模式。
- 桥接模式。
- 装饰模式。
- 组合模式。
- 外观模式。
- 享元模式。

- 代理模式。

1.3.3 行为型模式

行为型模式更加注重设计模式所体现出的行为动作。例如，命令模式体现了下发命令的行为；状态模式体现了状态变化的行为；观察者模式体现了观察、监听这一行为。行为型模式共包含以下 11 种设计模式。

- 策略模式。
- 模板方法模式。
- 观察者模式。
- 迭代子模式。
- 责任链模式。
- 命令模式。
- 备忘录模式。
- 状态模式。
- 访问者模式。
- 中介者模式。
- 解释器模式。

设计模式的分类，无非是对设计模式的特点进行一个粗粒度的划分，读者不必对此太过于纠结和执着。比如，部分设计模式的学习者对装饰器模式的划分方式有一定的意见分歧，装饰器模式被划分到了结构型模式，因为它体现了装饰者与被装饰者的结构关系。但是从行为的角度出发，装饰器体现了装饰行为，从这个角度来看，装饰器模式是否可以被划分到行为型模式呢？再比如，对访问者模式的分歧，如果从结构型角度分析访问者模式，体现了访问者与被访问者的类结构，那是否可以将访问者模式划分到结构型模式呢？所以，我们建议读者不必过分纠结于此，因为它仅仅是一个划分角度而已。

1.4 合成复用原则

合成复用原则，又称为组合 / 聚合复用原则（Composition/Aggregate Reuse Principle，CARP）。该原则要求在代码复用时，要尽量先使用组合或者聚合等关联关系（也就是包含、使用属性成员的方式）来实现，其次才考虑使用继承关系来实现。

合成复用原则，在部分资料中将其称之为设计模式的第七个原则，所以部分

资料中会有“设计模式七大原则”的字眼。对于设计模式原则而言，六大原则也好，七大原则也罢，两种说法并无对错之分。随着设计模式的应用越来越多，将来可能还会有其他原则的出现。正因为如此，由于各种资料对设计模式原则介绍的不统一性，所以笔者没有将该原则纳入章节重点。虽然没有纳入章节重点，但笔者依然会在此小节对合成复用原则进行详细的说明。

为了更加清晰地说明合成复用原则，笔者提出了一个常见的小问题：“我们在 SpringMVC 或 SpringBoot 项目的接口开发过程中，Controller 层如何调用 Service 层的方法呢？”

相信大部分读者都能够迅速地给出答案，使用 @Autowired 注解，将 Service 注入到 Controller 就可以了，正确的代码示例如下：

```
@RestController
public class UserController {
    @Autowired
    private UserService userService;

    public Object login(String ... parameters) {
        return userService.login(parameters);
    }
}
```

从以上代码示例中可以看出，我们通过 @Autowired 注解将 UserService 注入到了 UserController，我们可以随时对 UserService 中的方法进行调用，这就是合成复用原则之中提到的“要尽量先使用组合或者聚合等关联关系（也就是包含，使用属性成员的方式）来实现”。依赖注入，就是合成复用原则最具代表性、最经典的案例体现。

大家试想一下，如果我们采用继承的方式进行 UserService 方法调用，就会出现如下“毁三观”的代码。

```
// 以下为错误代码示例！！！
@RestController
public class UserController extends UserService {
    public Object login(String ... parameters) {
        return super.login(parameters);
    }
}
```

虽然依然能够达成我们调用 UserService 中方法的目的，但是这样的代码书写方式无疑增大了 UserController 和 UserService 的耦合性，而且违背了我们长久以来的开发方式。部分读者可能会有疑问：“难道合成复用原则下，所有的代码都不使用继承了吗？各种源码中，很多代码的类继承、接口实现都是有

问题的吗？”解答这些疑问，首先要理解类继承和接口实现的核心思想，两个能够成为父子关系的类或接口必然有相似的特性，或者有扩展的需要，如子类 LinkedHashMap 与父类 HashMap，再如 DefaultListableBeanFactory 子类与 BeanFactory 接口的关系，都是同宗同族，且子类都进行了扩展操作。然而以上代码示例中，仅仅为了调用 UserService 的方法而进行的继承，则是一种错误的方式。

1.5 章节回顾

本章通过 JDK8 源码、Spring5 源码中的常规示例以及开发过程中的常用案例对设计模式的七种原则进行了介绍，包括单一职责原则、接口隔离原则、里氏替换原则、依赖倒置原则、迪米特法则、开闭原则以及合成复用原则。此外，本章依然对设计模式的分类（创建型、结构型、行为型）进行了说明，使读者对设计模式有了宏观的了解。

接下来，我们要进入本书的重点内容，即以互联网核心需求为基准，对设计模式进行落地实战，让设计模式真正为我们的项目服务。