设计原则

第一章、为什么学习设计模式

一、由着性子写代码

之前，初入职场，很简单的一个功能，花了好久才写完，但是令我崩溃的是，经过领导的code-review后直接将我的代码推翻，来来回回，不停修改，知道“领导他”满意才行。很多时候，我甚至开始骂领导全家，代码能用不就行了，跑起来不就行了？

但是后来，公司派我带几个外包程序员干活，由于缺乏管理经验，项目进度有卡的很严，很多程序员都是“由着性子写代码”，属于管理，降低了对代码的审核要求。有时候，这并不是坏事，有些“交差性”的项目，这样做确实很快，很舒服。但是一旦一个项目是一个认真的、需要迭代的项目，这样做简直就是给自己挖坑，所谓的【屎山】项目就逐渐出来了。

我痛定思痛，决定只要我负责的项目，一定要严格规范，合理设计，然而完成这些，不仅仅需要我们扎实了解基础的计算机编程知识，设计模式也是不可或缺的一部分。

二、什么样的代码才是好的代码

好的代码应该具有以下特点：可读性、可扩展性、可复用性、可维护性、低耦合、高内聚和遵循设计原则。现在我们来详细说明这些特点，并通过举例进行解释。

好的代码具有以下特点：

1、可读性强

好的代码应该易于理解。变量和函数名称应清晰表达它们的作用，避免使用含糊不清或过于简短的命名。注释也是提高代码可读性的重要手段，应适时地为关键部分和复杂逻辑添加注释。

例如：

```
// 计算两点之间的距离
public double calculateDistance(Point point1, Point point2) {
    double deltaX = point2.getX() - point1.getX();
    double deltaY = point2.getY() - point1.getY();
    return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
}
```

2、高性能

好的代码应该具有高性能，避免不必要的计算和资源浪费。在编写代码时，应考虑算法的时间和空间复杂度，并在适当时候优化，这个需要我们对数据结构和算法有所了解。

例如，使用HashSet而非List来查找某个元素，以提高性能：

```
Set<String> uniqueNames = new HashSet<>();
uniqueNames.add("Alice");
uniqueNames.add("Bob");

if (uniqueNames.contains("Alice")) {
    System.out.println("Alice is in the set.");
}
```

3、遵循编程规范

好的代码应遵循相应的**编程规范**，包括命名规范、代码格式规范等。这有助于保持代码的一致性，使其他开发者更容易阅读和理解代码。

例如，在Java中，类名应使用大驼峰命名法，方法和变量名应使用小驼峰命名法：

```
class UserDaository {
    public User findUserByName(String userName) {
        // ...
    }
}
```

4、易于测试

好的代码应该易于测试，以确保代码的**正确性和稳定性**。编写可测试的代码，并编写单元测试来覆盖关键功能。

易于测试是指代码易于编写单元测试以验证其功能。为了使代码易于测试，我们需要关注以下几点：

1. 保持代码简洁，避免过长的方法和类。
2. 遵循单一职责原则，使每个类和方法只负责一个功能。

3. 使用依赖注入和接口，降低组件间的耦合，以便于替换和模拟依赖。

这次我们以一个计算器程序为例，对比易于测试和不易于测试的代码实现。

首先，我们创建一个 `Calculator` 类，它包含一个 `add` 方法用于求和。在不易测试的实现中，我们将输入验证逻辑直接放在 `add` 方法内，如下所示：

```
public class Calculator {
    public int add(int a, int b) {
        if (a < 0 || b < 0) {
            throw new IllegalArgumentException("Negative numbers are not allowed");
        }
        return a + b;
    }
}
```

这个实现的问题在于，输入**验证逻辑与实际计算逻辑耦合在一起，使得我们无法分别测试它们。**为了使代码易于测试，我们需要**将这两部分逻辑解耦。可以通过将输入验证逻辑移到一个单独的方法 `validateInput` 中来实现。**

```
public class Calculator {
    public int add(int a, int b) {
        validateInput(a, b);
        return a + b;
    }

    private void validateInput(int a, int b) {
        if (a < 0 || b < 0) {
            throw new IllegalArgumentException("Negative numbers are not allowed");
        }
    }
}
```

现在我们来编写测试用例。对于不易测试的实现，我们需要同时测试输入验证和计算逻辑：

```
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();

        // 测试正常输入
        assertEquals(5, calculator.add(2, 3));

        // 测试非法输入
        assertThrows(IllegalArgumentException.class, () -> calculator.add(-1, 3));
    }
}
```

对于易于测试的实现，我们可以分别测试输入验证和计算逻辑：

```
public class CalculatorTest {
    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();

        // 测试正常输入
        assertEquals(5, calculator.add(2, 3));
    }

    @Test
    public void testValidateInput() {
        Calculator calculator = new Calculator();

        // 测试非法输入
        assertThrows(IllegalArgumentException.class, () -> calculator.add(-1, 3));
    }
}
```

在易于测试的实现中，我们将输入验证和计算逻辑分离，使得测试更加简单和清晰。此外，这种解耦还有助于提高代码的可维护性和可读性。以上的代码十分简单，可能大家觉得有点矫枉过正，但是一旦代码变得复杂，很多的逻辑耦合在一起，对于测试而言会变得极难。

5、异常处理

好的代码应具备良好的异常处理机制。在可能出现异常的地方，使用try-catch语句捕获异常，并在适当的情况下记录错误信息或抛出新的异常，当然还要建立合适的异常全局捕获机制。

例如：

```
public void readFile(String filePath) {
    try {
        // 尝试打开文件
    } catch (FileNotFoundException e) {
        // 记录错误信息，并抛出新的异常
        logger.error("File not found: " + filePath, e);
        throw new CustomException("File not found", e);
    }
}
```

6、文档和注释

为关键部分和复杂逻辑编写详细的文档和注释，以帮助其他开发者更容易地理解和维护代码。同时，保持文档和注释的更新，以反映代码的最新状态。

例如：

```
/**
 * 这是一个用于处理字符串的工具类。
 */
class StringUtils {
    /**
     * 判断一个字符串是否为空或空白。
     *
     * @param str 需要判断的字符串
     * @return 如果字符串为空或空白，返回true，否则返回false
     */
    public static boolean isBlank(String str) {
        // ...
    }
}
```

通过遵循这些原则和实践，我们可以编写出具有高质量的代码，从而提高软件的可维护性、可扩展性和可靠性。同时，我们也应不断学习和积累经验，以便更好地掌握编程技巧，编写出更优秀的代码。

7、可复用性

好的代码应该具有可复用性，避免重复编写相同的代码。将通用功能抽取成独立的模块或库，以便在多个项目或模块中使用。

例如，创建一个日期工具类来处理日期相关的操作：

```
class DateUtils {
    // 将日期字符串转换为Date对象
    public static Date parseDate(String dateString, String format) {
        // ...
    }

    // 将Date对象格式化为字符串
    public static String formatDate(Date date, String format) {
        // ...
    }
}
```

8、易于维护

好的代码应该具有良好的**模块化和低耦合度**，使其易于修改和扩展。遵循单一职责原则，确保每个类和方法只负责一项任务。

例如：

```
// 文件操作类
class FileOperations {
    // 读取文件内容
    public String readFile(String filePath) {
        // ...
    }

    // 写入文件内容
    public void writeFile(String filePath, String content) {
        // ...
    }
}
```

9、可扩展性

好的代码应该具有良好的可扩展性，即当需求变更或新增功能时，能够方便地进行扩展，而不需要大规模地改动现有代码。。

我们要记住，**永远不变的就是变**。鬼才知道甲方爸爸或者产品什么时候要修改需求，好的设计一定是在面对变化的需求可以少量修改代码，甚至不修改代码。这个内容是我们学习设计模式的重点之一，这里就不举例子了，在课程中有很多案例。

三、为什么要学习设计模式

1、为了不写烂代码

学习设计模式有助于提高我们编写高质量代码的能力。设计模式是一套经过验证的、面向对象的编程解决方案，可以帮助我们处理软件开发过程中的常见问题。通过学习和应用设计模式，我们可以编写出更加清晰、易于理解和维护的代码，从而避免编写糟糕的代码。

2、为了面试

在软件工程师的面试过程中，设计模式通常是一个重要的考察点。面试官可能会要求你解释和应用某个特定的设计模式，或者询问你如何解决某个特定的编程问题。掌握设计模式不仅可以帮助你在面试中展示出扎实的编程基础，还能让你在解决问题时更加游刃有余。

3、为了提高设计和开发能力

学习设计模式可以帮助我们提高软件设计和开发能力。通过学习设计模式，我们可以更好地理解面向对象的设计原则，掌握如何将这些原则应用于实际问题。此外，设计模式还有助于提高我们解决问题的效率，因为我们可以直接应用经过验证的最佳实践，而无需从头开始尝试和错误。

4、为了更好的学习框架，阅读源码

许多流行的软件框架和库都是基于设计模式构建的，因此掌握设计模式有助于我们更好地理解和学习这些框架。当我们阅读框架或库的源代码时，了解设计模式可以帮助我们更快地理解代码的结构和逻辑，从而提高我们学习新技术的速度。同时，了解设计模式也有助于我们在自己的项目中更好地应用这些框架和库，从而提高我们的开发效率。

第二章 理解设计原则

一、单一原则

在之前的几节课程中，我们学习了面向对象编程的相关知识。从今天开始，我们将进入经典设计原则的学习阶段，这些原则包括 SOLID、KISS、DRY 和 迪米特法则等。

尽管这些设计原则在表面上看起来很容易理解，你可能觉得一看就能理解和掌握它们。然而，当真正将这些原则应用到实际项目中时，你会发现“理解”和“会应用”是两个不同层次的概念，而“有效地应用”则更具挑战性。根据我过去的工作经验，许多同事对这些原则的理解并不够深入，以至于在实际应用时过于教条主义。他们将这些原则视为绝对真理，僵化地套用，而非灵活地运用，从而产生了反作用。

1、如何理解单一职责原则（SRP）？

单一职责原则（Single Responsibility Principle，简称SRP），它**要求一个类或模块应该只负责一个特定的功能。**这有助于**降低类之间的耦合度**，**提高代码的可读性和可维护性。**

上边的概念中提到了类（class）和模块（module），关于这两个概念，在后边的学习中可以视作一样。

我们可以把模块看作比类更加抽象的概念，类也可以看作模块。或者把模块看作比类更加粗粒度的代码块，模块中包含多个类，多个类组成一个模块。

单一职责原则的定义描述非常简单，也不难理解。一个类只负责完成一个职责或者功能。也就是说，**不要设计大而全的类，要设计粒度小、功能单一的类。**换个角度来讲就是，一个类包含了两个或者两个以上业务不相干的功能，那我们就说它职责不够单一，应该将它拆分成多个功能更加单一、粒度更细的类。

让我们通过一个简单的例子来理解单一职责原则：

假设我们需要实现一个员工管理系统，**处理员工的信息和工资计算。**不遵循单一职责原则的实现可能如下：

```
class Employee {
    private String name;
    private String position;
    private double baseSalary;

    public Employee(String name, String position, double baseSalary) {
        this.name = name;
        this.position = position;
        this.baseSalary = baseSalary;
    }

    // Getter 和 Setter 方法
    public double calculateSalary() {
        // 计算员工工资的逻辑
        return baseSalary * 1.2;
    }

    public void saveEmployee() {
        // 保存员工信息到数据库的逻辑
    }
}
```

上面的代码中，`Employee` 类负责了员工信息的管理、工资计算以及员工信息的持久化。这违反了单一职责原则。为了遵循该原则，我们可以将这些功能拆分到不同的类中：

```
class Employee {
    private String name;
    private String position;
    private double baseSalary;

    public Employee(String name, String position, double baseSalary) {
        this.name = name;
```

```
        this.position = position;
        this.baseSalary = baseSalary;
    }

    // Getter 和 Setter 方法
    public double calculateSalary() {
        // 计算员工工资的逻辑
        return baseSalary * 1.2;
    }
}

class EmployeeRepository {
    public void saveEmployee(Employee employee) {
        // 保存员工信息到数据库的逻辑
    }
}
```

在遵循单一职责原则的代码中，我们将员工信息的持久化操作从 `Employee` 类中抽离出来，放到了一个新的 `EmployeeRepository` 类中。现在，`Employee` 类只负责员工信息的管理和工资计算，而 `EmployeeRepository` 类负责员工信息的持久化操作。这样，**每个类都只关注一个特定的职责，更易于理解、维护和扩展。**

遵循单一职责原则有助于提高代码的可读性、可维护性和可扩展性。请注意，这个原则并不是绝对的，需要根据具体情况来判断是否需要拆分类和模块。过度拆分可能导致过多的类和模块，反而增加系统的复杂度。

2、如何判断类的职责是否足够单一

从刚刚这个例子来看，单一职责原则看似不难应用。但大部分情况下，类里的方法是归为同一类功能，还是归为不相关的两类功能，并不是那么容易判定的。在真实的软件开发中，对于一个类是否职责单一的判定，是很难拿捏的。我举一个更加贴近实际的例子来给你解释一下。

在一个社交产品中，我们用下面的 UserInfo 类来记录用户的信息。你觉得，UserInfo 类的设计是否满足单一职责原则呢？

```
public class UserInfo {
  private long userId;
  private String username;
  private String email;
  private String telephone;
  private String avatarUrl;
  private String province; // 省
  private String cityOf; // 市
  private String region; // 区
  private String detailedAddress; // 详细地址
  // ... 省略其他属性和方法...
}
```

对于这个问题，有两种不同的观点。

一种观点是，UserInfo 类包含的都是跟用户相关的信息，所有的属性和方法都隶属于用户这样一个业务模型，满足单一职责原则；

另一种观点是，地址信息在 UserInfo 类中，所占的比重比较高，可以继续拆分成独立的 Address 类，UserInfo 只保留除 Address 之外的其他信息，拆分之后的两个类的职责更加单一。

哪种观点更对呢？实际上，要从中做出选择，我们不能脱离具体的应用场景。如果在这个社交产品中，用户的地址信息跟其他信息一样，只是单纯地用来展示，那 UserInfo 现在的设计就是合理的。但是，如果这个社交产品发展得比较好，之后又在产品中添加了电商的模块，用户的地址信息还会用在电商物流中，那我们最好将地址信息从 UserInfo 中拆分出来，独立成用户物流信息（或者叫地址信息、收货信息等）。

所以，我么还有记住一句话，**脱离了业务谈设计是耍流氓**，事实上脱离了业务谈什么都是耍流氓，**技术服务于业务这是亘古不变的道理。**

我们再进一步延伸一下。如果做这个社交产品的公司发展得越来越好，公司内部又开发出了跟多其他产品（可以理解为其他 App）。公司希望支持统一账号系统，也就是用户一个账号可以在公司内部的所有产品中登录。这个时候，我们就需要继续对 UserInfo 进行拆分，将跟身份认证相关的信息（比如，email、telephone 等）抽取成独立的类。

从刚刚这个例子，我们可以总结出，不同的应用场景、不同阶段的需求背景下，对同一个类的职责是否单一的判定，可能都是不一样的。在某种应用场景或者当下的需求背景下，一个类的设计可能已经满足单一职责原则了，但如果换个应用场景或着在未来的某个需求背景下，可能就不满足了，需要继续拆分成粒度更细的类。

除此之外，从不同的业务层面去看待同一个类的设计，对类是否职责单一，也会有不同的认识。比如，例子中的 UserInfo 类。如果我们从“用户”这个业务层面来看，UserInfo 包含的信息都属于用户，满足职责单一原则。如果我们从更加细分的“用户展示信息”“地址信息”“登录认证信息”等等这些更细粒度的业务层面来看，那 UserInfo 就应该继续拆分。

综上所述，评价一个类的职责是否足够单一，我们并没有一个非常明确的、可以量化的标准，可以说，这是件非常主观、仁者见仁智者见智的事情。实际上，在真正的软件开发中，**我们也没必要过于未雨绸缪，过度设计。**所以，**我们可以先写一个粗粒度的类，满足业务需求。随着业务的发展，如果粗粒度的类越来越庞大，代码越来越多，这个时候，我们就可以将这个粗粒度的类，拆分成几个更细粒度的类。这就是所谓的持续重构**（后面的章节中我们会讲到）。

听到这里，你可能会说，这个原则如此含糊不清、模棱两可，到底该如何拿捏才好啊？我这里还有一些小技巧，能够很好地帮你，从侧面上判定一个类的职责是否够单一。而且，我个人觉得，下面这几条判断原则，比起很主观地去思考类是否职责单一，要更有指导意义、更具有可执行性：

- 类中的代码行数、函数或属性过多，会影响代码的可读性和可维护性，我们就需要考虑对类进行拆分；
- 类依赖的其他类过多，或者依赖类的其他类过多，不符合**高内聚、低耦合**的设计思想，我们就需要考虑对类进行拆分；
- 私有方法过多，我们就要考虑能否将私有方法独立到新的类中，设置为 public 方法，供更多的类使用，从而提高代码的复用性；
- 比较难给类起一个合适名字，很难用一个业务名词概括，或者只能用一些笼统的 Manager、Context 之类的词语来命名，这就说明类的职责定义得可能不够清晰；
- 类中大量的方法都是集中操作类中的某几个属性，比如，在 UserInfo 例子中，如果一半的方法都是在操作 address 信息，那就可以考虑将这几个属性和对应的方法拆分出来。

不过，你可能还会有这样的疑问：在上面的判定原则中，我提到类中的代码行数、函数或者属性过多，就有可能不满足单一职责原则。那多少行代码才算是行数过多呢？多少个函数、属性才称得上过多呢？

比较初级的工程师经常会问这类问题。实际上，这个问题并不好定量地回答，就像你问大厨“放盐少许”中的“少许”是多少，大厨也很难告诉你一个特别具体的量值。

如果继续深究一下的话，你可能还会说，一些菜谱确实给出了，做某某菜需要放多少克盐，放多少克油的具体量值啊。我想说的是，那是给家庭主妇用的，那不是给专业的大厨看的。类比一下做饭，如果你是没有太多项目经验的编程初学者，实际上，我也可以给你一个凑活能用、比较宽泛的、可量化的标准，那就是一个类的代码行数最好不能超过 200 行，函数个数及属性个数都最好不要超过 10 个。

实际上， 从另一个角度来看，当一个类的代码，读起来让你头大了，实现某个功能时不知道该用哪个函数了，想用哪个函数翻半天都找不到了，只用到一个小功能要引入整个类（类中包含很多无关此功能实现的函数）的时候，这就说明类的行数、函数、属性过多了。实际上，等你做多项目了，代码写多了，在开发中慢慢“品尝”，自然就知道什么是“放盐少许”了，这就是所谓的“专业第六感”。

3、类的职责是否设计得越单一越好

为了满足单一职责原则，是不是把类拆得越细就越好呢？答案是否定的。我们还是通过一个例子来解释一下。Serialization 类实现了一个简单协议的序列化和反序列功能，具体代码如下：

```
/**
 * Protocol format: identifier-string;{gson string}
 * For example: UEUEUE;{"a":"A","b":"B"}
 */
public class Serialization {
  private static final String IDENTIFIER_STRING = "UEUEUE;";
  private Gson gson;

  public Serialization() {
    this.gson = new Gson();
  }

  public String serialize(Map<String, String> object) {
    StringBuilder textBuilder = new StringBuilder();
    textBuilder.append(IDENTIFIER_STRING);
    textBuilder.append(gson.toJson(object));
    return textBuilder.toString();
  }

  public Map<String, String> deserialize(String text) {
```

```
    if (!text.startsWith(IDENTIFIER_STRING)) {
        return Collections.emptyMap();
    }
    String gsonStr = text.substring(IDENTIFIER_STRING.length());
    return gson.fromJson(gsonStr, Map.class);
  }
}
```

如果我们想让类的职责更加单一，我们对 Serialization 类进一步拆分，拆分成一个只负责序列化工作的 Serializer 类和另一个只负责反序列化工作的 Deserializer 类。拆分后的具体代码如下所示：

```
public class Serializer {
  private static final String IDENTIFIER_STRING = "UEUEUE;";
  private Gson gson;

  public Serializer() {
    this.gson = new Gson();
  }

  public String serialize(Map<String, String> object) {
    StringBuilder textBuilder = new StringBuilder();
    textBuilder.append(IDENTIFIER_STRING);
    textBuilder.append(gson.toJson(object));
    return textBuilder.toString();
  }
}
public class Deserializer {
  private static final String IDENTIFIER_STRING = "UEUEUE;";
  private Gson gson;

  public Deserializer() {
    this.gson = new Gson();
  }

  public Map<String, String> deserialize(String text) {
    if (!text.startsWith(IDENTIFIER_STRING)) {
        return Collections.emptyMap();
    }
    String gsonStr = text.substring(IDENTIFIER_STRING.length());
    return gson.fromJson(gsonStr, Map.class);
  }
```

```
}
```

虽然经过拆分之后，Serializer 类和 Deserializer 类的职责更加单一了，但也随之带来了新的问题。如果我们修改了协议的格式，数据标识从“UEUEUE”改为“DFDFDF”，或者序列化方式从 JSON 改为了 XML，那 Serializer 类和 Deserializer 类都需要做相应的修改，代码的内聚性显然没有原来 Serialization 高了。而且，如果我们仅仅对 Serializer 类做了协议修改，而忘记了修改 Deserializer 类的代码，那就会导致序列化、反序列化不匹配，程序运行出错，也就是说，拆分之后，代码的可维护性变差了。

实际上，不管是应用设计原则还是设计模式，最终的目的还是提高代码的可读性、可扩展性、复用性、可维护性等。我们在考虑应用某一个设计原则是否合理的时候，也可以以此作为最终的考量标准。

4、重点回顾

今天的内容到此就讲完了。我们来一块总结回顾一下，你应该掌握的重点内容。

1. 如何理解单一职责原则（SRP）？

一个类只负责完成一个职责或者功能。不要设计大而全的类，要设计粒度小、功能单一的类。单一职责原则是为了实现代码高内聚、低耦合，提高代码的复用性、可读性、可维护性。

2. 如何判断类的职责是否足够单一？

不同的应用场景、不同阶段的需求背景、不同的业务层面，对同一个类的职责是否单一，可能会有不同的判定结果。实际上，一些侧面的判断指标更具有指导意义和可执行性，比如，出现下面这些情况就有可能说明这类的设计不满足单一职责原则：

- 类中的代码行数、函数或者属性过多；
- 类依赖的其他类过多，或者依赖类的其他类过多；
- 私有方法过多；
- 比较难给类起一个合适的名字；
- 类中大量的方法都是集中操作类中的某几个属性。

3. 类的职责是否设计得越单一越好？

单一职责原则通过避免设计大而全的类，避免将不相关的功能耦合在一起，来提高类的内聚性。同时，类职责单一，类依赖的和被依赖的其他类也会变少，减少了代码的耦合性，以此来实现代码的高内聚、低耦合。但是，如果拆分得过细，实际上会适得其反，反倒会降低内聚性，也会影响代码的可维护性。

5、课堂讨论

今天课堂讨论的话题有两个：

对于如何判断一个类是否职责单一，如何判断代码行数过多，你还有哪些其他的方法吗？

单一职责原则，除了应用到类的设计上，还能延伸到哪些其他设计方面吗？

二、开闭原则

今天，我们来学习 SOLID 中的第二个原则：开闭原则。开闭原则是 SOLID 中最难理解、最难掌握，同时也是最有用的一条原则。

1、原理概述

开闭原则的英文全称是 Open Closed Principle，简写为 OCP。它的英文描述是：software entities (modules, classes, functions, etc.) should be open for extension , but closed for modification。我们把它翻译成中文就是：软件实体（模块、类、方法等）应该“对扩展开放、对修改关闭”。

说人话就是，当我们需要添加一个新的功能时，应该在已有代码基础上**扩展代码**（新增模块、类、方法等），而**非修改已有代码**（修改模块、类、方法等）。

以下是一个常见的生产环境中的例子，我们将展示一个简化的**电商平台的订单折扣策略。**

你觉得下边的代码有问题吗？

```
class Order {
    private double totalAmount;

    public Order(double totalAmount) {
        this.totalAmount = totalAmount;
    }

    // 计算折扣后的金额
    public double getDiscountedAmount(String discountType) {
        double discountedAmount = totalAmount;

        if (discountType.equals("FESTIVAL")) {
            discountedAmount = totalAmount * 0.9; // 节日折扣，9折
```

```
        } else if (discountType.equals("SEASONAL")) {
            discountedAmount = totalAmount * 0.8; // 季节折扣，8折
        }

        return discountedAmount;
    }
}
```

上述代码中，`Order` 类包含一个计算折扣金额的方法，它根据不同的折扣类型应用折扣。当我们需要添加新的折扣类型时，就不得不需要修改 `getDiscountedAmount` 方法的代码，这显然是不合理的，这就**违反了开闭原则。**

遵循开闭原则的代码：

```
// 抽象折扣策略接口
interface DiscountStrategy {
    double getDiscountedAmount(double totalAmount);
}

// 节日折扣策略
class FestivalDiscountStrategy implements DiscountStrategy {
    @Override
    public double getDiscountedAmount(double totalAmount) {
        return totalAmount * 0.9; // 9折
    }
}

// 季节折扣策略
class SeasonalDiscountStrategy implements DiscountStrategy {
    @Override
    public double getDiscountedAmount(double totalAmount) {
        return totalAmount * 0.8; // 8折
    }
}

class Order {
    private double totalAmount;
    private DiscountStrategy discountStrategy;

    public Order(double totalAmount, DiscountStrategy discountStrategy) {
        this.totalAmount = totalAmount;
        this.discountStrategy = discountStrategy;
    }
```

```
    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    // 计算折扣后的金额
    public double getDiscountedAmount() {
        return discountStrategy.getDiscountedAmount(totalAmount);
    }
}
```

在遵循开闭原则的代码中，我们定义了一个**抽象的折扣策略接口** `DiscountStrategy`，然后为每种折扣类型创建了一个实现该接口的策略类。`Order` 类使用**组合的方式**，包含一个 `DiscountStrategy` 类型的成员变量，以便**在运行时设置或更改折扣策略**，（可以通过编码，配置、依赖注入等形式）。这样，当我们需要添加新的折扣类型时，只需实现 `DiscountStrategy` 接口即可，而无需修改现有的 `Order` 代码。这个例子遵循了开闭原则。

2、修改代码就意味着违背开闭原则吗

开闭原则的核心思想是要**尽量减少对现有代码的修改**，以**降低修改带来的风险和影响**。在实际开发过程中，**完全不修改代码是不现实的**。当需求变更或者发现代码中的错误时，修改代码是正常的。然而，开闭原则鼓励我们通过**设计更好的代码结构**，使得在添加新功能或者扩展系统时，尽量减少对现有代码的修改。

以下是一个简化的日志记录器的示例，展示了在适当情况下修改代码，也不违背开闭原则。在这个例子中，我们的应用程序支持将日志输出到控制台和文件。假设我们需要添加一个新功能，以便在输出日志时同时添加一个时间戳。

原始代码：

```
interface Logger {
    void log(String message);
}

class ConsoleLogger implements Logger {
    @Override
    public void log(String message) {
        System.out.println("Console: " + message);
```

```
    }
}

class FileLogger implements Logger {
    @Override
    public void log(String message) {
        System.out.println("File: " + message);
        // 将日志写入文件的实现省略
    }
}
```

为了添加时间戳功能，我们需要修改现有的 `ConsoleLogger` 和 `FileLogger` 类。虽然我们需要修改代码，但由于这是对现有功能的改进，而不是添加新的功能，所以这种修改是可以接受的，不违背开闭原则。

修改后的代码：

```
interface Logger {
    void log(String message);
}

class ConsoleLogger implements Logger {
    @Override
    public void log(String message) {
        String timestamp =
LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
        System.out.println("Console [" + timestamp + "]: " + message);
    }
}

class FileLogger implements Logger {
    @Override
    public void log(String message) {
        String timestamp =
LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
        String logMessage = "File [" + timestamp + "]: " + message;
        System.out.println(logMessage);
        // 将日志写入文件的实现省略
    }
}
```

在这个例子中，我们只是对现有的日志记录器类进行了适当的修改，以添加时间戳功能。这种修改不会影响到其他部分的代码，因此不违背开闭原则。总之，适当的修改代码并不一定违背开闭原则，关键在于我们如何权衡修改的影响和代码设计。

当我们遵循开闭原则时，其目的是为了让我们的代码**更容易维护、更具可复用性**，同时降低了引入新缺陷的风险。但是，在某些情况下，遵循开闭原则可能**会导致过度设计，增加代码的复杂性**。因此，在实际开发中，我们应该根据**实际需求和预期的变化来平衡遵循开闭原则的程度**。写代码不是为了设计而设计，**脱离需求谈设计都是耍流氓**，有些场景，比如项目的使用频率不高，修改的可能性很低，或者代码本来就很简单使用了设计模式可能会增加开发难度，提升开发成本，反而得不偿失。

3、如何做到“对扩展开放、修改关闭”

开闭原则讲的就是代码的**扩展性问题**，是判断一段代码是否易扩展的“黄金标准”。如果某段代码在应对未来需求变化的时候，能够做到“对扩展开放、对修改关闭”，那就说明这段代码的扩展性比较好。事实上，我学习设计模式的很重要的一个目的就是写出扩展性好的代码。

在讲具体的方法论之前，我们先来看一些更加偏向顶层的指导思想。为了尽量写出扩展性好的代码，我们要时刻具备扩展意识、抽象意识、封装意识。这些“潜意识”可能比任何开发技巧都重要。

如果我们给自己的定位是“工程师”，而非码农，那我们在写任何一段代码时都应该思考一些问题：

- 我要写的这段代码未来**可能有哪些需求变更、如何设计代码结构，事先留好扩展点，以便在未来需求变更的时候，**不需要改动代码整体结构、做到最小代码改动的情况下，新的代码能够很灵活地插入到扩展点上，做到“对扩展开放、对修改关闭”。
- 我们还要识别出代码**可变部分和不可变部分**，要将可变部分封装起来，隔离变化，提供**抽象化的不可变接口**，给上层系统使用。当具体的实现发生变化的时候，我们只需要基于相同的抽象接口，扩展一个新的实现，替换掉老的实现即可，上游系统的代码几乎不需要修改。

聊完了如何思考，我们可以采用以下策略和设计模式：

1. **抽象与封装**：通过定义**接口或抽象类来封装变化的部分，将共性行为抽象出来。**当需要添加新功能时，只需要实现接口或继承抽象类，而不需要修改现有代码。
2. **组合/聚合**：使用组合或聚合的方式，将多个不同功能模块组合在一起，形成一个更大的系统。当需要扩展功能时，只需要添加新的组件，而不需要修改现有的

组件。
3. **使用依赖注入：**
4. **使用设计模式**：设计模式是针对某些特定问题的通用解决方案。很多设计模式都是为了支持“对扩展开放、修改关闭”的原则。例如，策略模式、工厂模式、装饰器模式等，都是为了实现这个原则。
5. **使用事件和回调**：通过事件驱动和回调函数，可以让系统在运行时根据需要动态地添加或修改功能，而无需修改现有代码。
6. **使用插件机制**：通过插件机制，可以允许第三方开发者为系统添加新功能，而无需修改系统的核心代码。这种机制常用于框架和大型软件系统中。

需要注意的是，遵循开闭原则并不意味着永远不能修改代码。在实际开发过程中，完全不修改代码是不现实的。开闭原则的目标是要尽量降低修改代码带来的风险和影响，提高代码的可维护性和可复用性。在实际开发中，我们应该根据项目需求和预期的变化来平衡遵循开闭原则的程度。

三、里氏替换原则

1、原理概述

里式替换原则的英文翻译是：Liskov Substitution Principle，缩写为 LSP（老色胚）。这个原则最早是在 1986 年由 Barbara Liskov 提出，他是这么描述这条原则的：

> Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it。（使用基类引用指针的函数必须能够在不知情的情况下使用派生类的对象。）

我们综合两者的描述，将这条原则用中文描述出来，是这样的：**子类对象（object of subtype/derived class）能够替换程序（program）中父类对象（object of base/parent class）出现的任何地方，并且保证原来程序的逻辑行为（behavior）不变及正确性不被破坏。**

以下是一个简单的示例：

```
// 基类：鸟类
public class Bird {
    public void fly() {
        System.out.println("I can fly");
    }
```

```
}

// 子类：企鹅类
public class Penguin extends Bird {
    // 企鹅不能飞，所以覆盖了基类的fly方法，但这违反了里氏替换原则
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

为了遵循LSP，我们可以重新设计类结构，将**能飞的行为抽象到一个接口**中，让需要飞行能力的鸟类实现这个接口：

```
// 飞行行为接口
public interface Flyable {
    void fly();
}

// 基类：鸟类
public class Bird {
}

// 子类：能飞的鸟类
public class FlyingBird extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("I can fly");
    }
}

// 子类：企鹅类，不实现Flyable接口
public class Penguin extends Bird {
}
```

通过这样的设计，我们遵循了里氏替换原则，同时也保证了代码的可维护性和复用性。

我们再来看一个基于数据库操作的案例。假设我们正在开发一个支持多种数据库的程序，包括MySQL、PostgreSQL和SQLite。我们可以使用里氏替换原则来设计合适的类结构，确保代码的可维护性和扩展性。

首先，我们定义一个抽象的`Database`基类，它包含一些通用的数据库操作方法，如`connect()`、`disconnect()`和`executeQuery()`。这些方法的具体实现将在子类中完成。

```
public abstract class Database {
    public abstract void connect();
    public abstract void disconnect();
    public abstract void executeQuery(String query);
}
```

然后，为每种数据库类型创建一个子类，继承自`Database`基类。这些子类需要实现基类中定义的抽象方法，并可以添加特定于各自数据库的方法。

```
public class MySQLDatabase extends Database {
    @Override
    public void connect() {
        // 实现MySQL的连接逻辑
    }

    @Override
    public void disconnect() {
        // 实现MySQL的断开连接逻辑
    }

    @Override
    public void executeQuery(String query) {
        // 实现MySQL的查询逻辑
    }

    // 其他针对MySQL的特定方法
}

public class PostgreSQLDatabase extends Database {
    // 类似地，为PostgreSQL实现相应的方法
}

public class SQLiteDatabase extends Database {
    // 类似地，为SQLite实现相应的方法
}
```

这样设计的好处是，我们可以在不同的数据库类型之间灵活切换，而不需要修改大量代码。只要这些子类遵循里氏替换原则，我们就可以放心地使用基类的引用来操作不同类型的数据库。例如：

```
public class DatabaseClient {
    private Database database;

    public DatabaseClient(Database database) {
        this.database = database;
    }

    public void performDatabaseOperations() {
        database.connect();
        database.executeQuery("SELECT * FROM users");
        database.disconnect();
    }
}

public class Main {
    public static void main(String[] args) {
        // 使用MySQL数据库
        DatabaseClient client1 = new DatabaseClient(new MySQLDatabase());
        client1.performDatabaseOperations();

        // 切换到PostgreSQL数据库
        DatabaseClient client2 = new DatabaseClient(new PostgreSQLDatabase());
        client2.performDatabaseOperations();

        // 切换到SQLite数据库
        DatabaseClient client3 = new DatabaseClient(new SQLiteDatabase());
        client3.performDatabaseOperations();
    }
}
```

通过遵循里氏替换原则，我们确保了代码的可维护性和扩展性。如果需要支持新的数据库类型，只需创建一个新的子类，实现`Database`基类中定义的抽象方法即可。

好了，我们稍微总结一下。虽然从定义描述和代码实现上来看，多态和里式替换有点类似，但它们关注的角度是不一样的。多态是面向对象编程的一大特性，也是面向对象编程语言的一种语法。它是一种代码实现的思路。而里式替换是一种**设计原则，是用来指导继承关系中子类该如何设计的，子类的设计要保证在替换父类的时候，不改变原有程序的逻辑以及不破坏原有程序的正确性。**

2、哪些代码明显违背了 LSP?

违背里氏替换原则（LSP）的代码通常具有以下特征：

（1）子类覆盖或修改了基类的方法

当子类覆盖或修改基类的方法时，可能导致子类无法替换基类的实例而不引起问题。这违反了LSP，会导致代码变得脆弱和不易维护。

```
public class Bird {
    public void fly() {
        System.out.println("I can fly");
    }
}

public class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

在这个例子中，`Penguin`类覆盖了`Bird`类的`fly()`方法，抛出了一个异常。这违反了LSP，因为现在`Penguin`实例无法替换`Bird`实例而不引发问题。

（2）子类违反了基类的约束条件

当子类违反了基类中定义的约束条件（如输入、输出或异常等），也会违反LSP。

```
public class Stack {
    private int top;
    private int[] elements;

    public Stack(int size) {
        elements = new int[size];
        top = -1;
    }

    public void push(int value) {
        if (top >= elements.length - 1) {
            throw new IllegalStateException("Stack is full");
        }
```

```
        elements[++top] = value;
    }

    public int pop() {
        if (top < 0) {
            throw new IllegalStateException("Stack is empty");
        }
        return elements[top--];
    }
}

// 正数的栈
public class NonNegativeStack extends Stack {
    public NonNegativeStack(int size) {
        super(size);
    }

    @Override
    public void push(int value) {
        if (value < 0) {
            throw new IllegalArgumentException("Only non-negative values are
allowed");
        }
        super.push(value);
    }
}

// 正确的写法
public class NonNegativeStack extends Stack {
    public NonNegativeStack(int size) {
        super(size);
    }

    public void pushNonNegative(int value) {
        if (value < 0) {
            throw new IllegalArgumentException("Only non-negative values are
allowed");
        }
        super.push(value);
    }
}
```

在这个例子中，`NonNegativeStack`子类违反了`Stack`基类的约束条件，因为它在`push()`方法中添加了一个新的约束，即只允许非负数入栈。这使得`NonNegativeStack`实例无法替换`Stack`实例而不引发问题，违反了LSP。

（3）子类与基类之间缺乏"is-a"关系

当子类与基类之间缺乏真正的"is-a"关系时，也可能导致违反LSP。例如，如果一个类继承自另一个类，仅仅因为它们具有部分相似性，而不是完全的"is-a"关系，那么这种继承关系可能不满足LSP。

为了避免违反LSP，我们需要在设计和实现过程中注意以下几点：

1. 确保子类和基类之间存在真正的"is-a"关系。
2. 遵循其他设计原则，如单一职责原则、开闭原则和依赖倒置原则。

四、接口隔离原则

1、原理概述

接口隔离原则的英文翻译是“ Interface Segregation Principle”，缩写为 ISP。Robert Martin 在 SOLID 原则中是这样定义它的：“Clients should not be forced to depend upon interfaces that they do not use。”直译成中文的话就是：客户端不应该强迫依赖它不需要的接口。其中的“客户端”，可以理解为接口的**调用者或者使用者**。

实际上，“接口”这个名词可以用在很多场合中。生活中我们可以用它来指插座接口等。在软件开发中，我们既可以把它看作一组抽象的约定，也可以具体指系统与系统之间的 API 接口，还可以特指面向对象编程语言中的接口等。

前面我提到，理解接口隔离原则的关键，就是理解其中的“接口”二字。在这条原则中，我们可以把“接口”理解为下面三种东西：

- 一组 API 接口集合
- 单个 API 接口或函数
- OOP 中的接口概念

接下来，我就按照这三种理解方式来详细讲一下，在不同的场景下，这条原则具体是如何解读和应用的。

接口隔离原则（Interface Segregation Principle，ISP）是一种面向对象编程的设计原则，它要求我们将大的、臃肿的接口拆分成更小、更专注的接口，以确保类之间的解耦。这样，客户端只需要依赖它实际使用的接口，而不需要依赖那些无关的接口。

接口隔离原则有以下几个要点：

1. 将一个**大的、通用的接口拆分成多个专用的接口**。这样可以降低类之间的耦合度，提高代码的可维护性和可读性。
2. 为每个接口定义一个独立的职责。这样可以确保接口的粒度适当，同时也有助于遵循单一职责原则。
3. 在定义接口时，要考虑到客户端的实际需求。客户端不应该被迫实现无关的接口方法。

下面我们来看一个示例，说明如何应用接口隔离原则。

假设我们正在开发一个机器人程序，机器人具有多种功能，如行走、飞行和工作。我们可以为这些功能创建一个统一的接口：

```
public interface Robot {
    void walk();
    void fly();
    void work();
}
```

然而，这个接口并不符合接口隔离原则，因为它将多个功能聚合在了一个接口中。对于那些只需要实现部分功能的客户端来说，这个接口会导致不必要的依赖。为了遵循接口隔离原则，我们应该将这个接口拆分成多个更小、更专注的接口：

```
public interface Walkable {
    void walk();
}

public interface Flyable {
    void fly();
}

public interface Workable {
    void work();
}
```

现在，我们可以根据需要为不同类型的机器人实现不同的接口。例如，对于一个只能行走和工作的机器人，我们只需要实现`walkable`和`workable`接口：

```
public class WalkingWorkerRobot implements Walkable, Workable {
    @Override
    public void walk() {
        // 实现行走功能
    }

    @Override
    public void work() {
        // 实现工作功能
    }
}
```

通过遵循接口隔离原则，我们确保了代码的可维护性和可读性，同时也降低了类之间的耦合度。在实际项目中，要根据需求和场景来判断何时应用接口隔离原则。

有的同学可以会发出疑问，如果按照上边的例子来说，是不是每个接口只能定义一个方法了？

一个接口只定义一个方法确实可以满足接口隔离原则，但这并不是一个绝对的标准。在设计接口时，我们需要权衡**接口的粒度和实际需求。过度拆分接口可能导致过多的单方法接口，这会增加代码的复杂性，降低可读性和可维护性。**

关键在于确保接口的**职责清晰且单一**，以便客户端只需依赖它们真正需要的接口。在某些情况下，一个接口包含多个方法是合理的，只要这些方法服务于一个单一的职责。例如，一个数据库操作接口可能包含`connect()`、`disconnect()`、`executeQuery()`等方法，这些方法都是数据库操作的一部分，因此可以放在同一个接口中。

总之，在遵循接口隔离原则时，我们需要根据实际情况进行权衡，以确保代码既简洁又易于维护。过度拆分接口并不总是一个好主意，关键是确保接口的职责单一且清晰。

2、案例解析

（1）案例一

我们还是结合一个例子来讲解。微服务用户系统提供了一组跟用户相关的 API 给其他系统使用，比如：注册、登录、获取用户信息等。具体代码如下所示：

```
public interface UserService {
  boolean register(String cellphone, String password);
  boolean login(String cellphone, String password);
  UserInfo getUserInfoById(long id);
  UserInfo getUserInfoByCellphone(String cellphone);
}
public class UserServiceImpl implements UserService {
  //...
}
```

现在，我们的后台管理系统要实现删除用户的功能，希望用户系统提供一个删除用户的接口。这个时候我们该如何来做呢？你可能会说，这不是很简单吗，我只需要在 UserService 中新添加一个 deleteUserByCellphone() 或 deleteUserById() 接口就可以了。这个方法可以解决问题，但是也隐藏了一些安全隐患。

删除用户是一个非常慎重的操作，我们只希望通过后台管理系统来执行，所以这个接口只限于给后台管理系统使用。如果我们把它放到 UserService 中，那所有使用到 UserService 的系统，都可以调用这个接口。不加限制地被其他业务系统调用，就有可能导致误删用户。

当然，最好的解决方案是从架构设计的层面，**通过接口鉴权的方式来限制接口的调用**。不过，如果暂时没有鉴权框架来支持，我们还可以从代码设计的层面，尽量避免接口被误用。我们参照接口隔离原则，调用者不应该强迫依赖它不需要的接口，将删除接口单独放到另外一个接口 RestrictedUserService 中，然后将 RestrictedUserService 只打包提供给后台管理系统来使用。具体的代码实现如下所示：

```
public interface UserService {
  boolean register(String cellphone, String password);
  boolean login(String cellphone, String password);
  UserInfo getUserInfoById(long id);
  UserInfo getUserInfoByCellphone(String cellphone);
}
public interface RestrictedUserService {
  boolean deleteUserByCellphone(String cellphone);
  boolean deleteUserById(long id);
}
public class UserServiceImpl implements UserService, RestrictedUserService {
  // ... 省略实现代码...
}
```

在刚刚的这个例子中，我们把接口隔离原则中的接口，理解为一组接口集合，它可以是某个微服务的接口，也可以是某个类库的接口等等。在设计微服务或者类库接口的时候，如果部分接口只被部分调用者使用，那我们就需要将这部分接口隔离出来，单独给对应的调用者使用，而不是强迫其他调用者也依赖这部分不会被用到的接口。

（2）案例二

现在我们再换一种理解方式，把接口理解为**单个接口或函数**（以下为了方便讲解，我都简称为“函数”）。那接口隔离原则就可以理解为：**函数的设计要功能单一，不要将多个不同的功能逻辑在一个函数中实现。**接下来，我们还是通过一个例子来解释一下。

```
public class Statistics {
  private Long max;
  private Long min;
  private Long average;
  private Long sum;
  private Long percentile99;
  private Long percentile999;
  //... 省略 constructor/getter/setter 等方法...
}
public Statistics count(Collection<Long> dataSet) {
  Statistics statistics = new Statistics();
  //... 省略计算逻辑...
  return statistics;
}
```

在上面的代码中，count() 函数的功能不够单一，包含很多不同的统计功能，比如，求最大值、最小值、平均值等等。按照接口隔离原则，我们应该把 count() 函数拆成几个更小粒度的函数，每个函数负责一个独立的统计功能。拆分之后的代码如下所示：

```
public Long max(Collection<Long> dataSet) { //... }
public Long min(Collection<Long> dataSet) { //... }
public Long average(Colletion<Long> dataSet) { //... }
// ... 省略其他统计函数...
```

不过，你可能会说，在某种意义上讲，count() 函数也不能算是职责不够单一，毕竟它做的事情只跟统计相关。我们在讲单一职责原则的时候，也提到过类似的问题。实际上，判定功能是否单一，除了**很强的主观性，还需要结合具体的场景。**

如果在项目中，对每个统计需求，Statistics 定义的那几个统计信息都有涉及，那 count() 函数的设计就是合理的。相反，如果每个统计需求只涉及 Statistics 罗列的统计信息中一部分，比如，有的只需要用到 max、min、average 这三类统计信息，有的只需要用到 average、sum。而 count() 函数每次都会把所有的统计信息计算一遍，就会做很多无用功，势必影响代码的性能，特别是在需要统计的数据量很大的时候。所以，在这个应用场景下，count() 函数的设计就有点不合理了，我们应该按照第二种设计思路，将其拆分成粒度更细的多个统计函数。

不过，你应该已经发现，接口隔离原则跟单一职责原则有点类似，不过稍微还是有点区别。**单一职责原则针对的是模块、类、接口的设计。而接口隔离原则相对于单一职责原则，一方面它更侧重于接口的设计，另一方面它的思考的角度不同。它提供了一种判断接口是否职责单一的标准：通过调用者如何使用接口来间接地判定。如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。**

（3）案例三

除了刚讲过的两种理解方式，我们还可以把“接口”理解为 OOP 中的接口概念，比如 Java 中的 interface。我还是通过一个例子来给你解释。

假设我们的项目中用到了三个外部系统：Redis、MySQL、Kafka。每个系统都对应一系列配置信息，比如地址、端口、访问超时时间等。为了在内存中存储这些配置信息，供项目中的其他模块来使用，我们分别设计实现了三个 Configuration 类：RedisConfig、MysqlConfig、KafkaConfig。具体的代码实现如下所示。注意，这里我只给出了 RedisConfig 的代码实现，另外两个都是类似的，我这里就不贴了。

```
public class RedisConfig {
  private ConfigSource configSource; // 配置中心（比如 zookeeper）
  private String address;
  private int timeout;
  private int maxTotal;
  // 省略其他配置: maxWaitMillis,maxIdle,minIdle...
  public RedisConfig(ConfigSource configSource) {
    this.configSource = configSource;
  }
  public String getAddress() {
    return this.address;
  }
  //... 省略其他 get()、init() 方法...
  public void update() {
   // 从 configSource 加载配置到 address/timeout/maxTotal...
  }
}
```

```
public class KafkaConfig { //... 省略... }
public class MysqlConfig { //... 省略... }
```

现在，我们有一个新的功能需求，希望支持 Redis 和 Kafka 配置信息的热更新。所谓“热更新（hot update）”就是，如果在配置中心中更改了配置信息，我们希望在不用重启系统的情况下，能将最新的配置信息加载到内存中（也就是 RedisConfig、KafkaConfig 类中）。但是，因为某些原因，我们并不希望对 MySQL 的配置信息进行热更新。

为了实现这样一个功能需求，我们设计实现了一个 ScheduledUpdater 类，以固定时间频率（periodInSeconds）来调用 RedisConfig、KafkaConfig 的 update() 方法更新配置信息。具体的代码实现如下所示：

```
public interface Updater {
  void update();
}
public class RedisConfig implemets Updater {
  //... 省略其他属性和方法...
  @Override
  public void update() { //...
  }
}
public class KafkaConfig implements Updater {
  //... 省略其他属性和方法...
  @Override
  public void update() { //...
  }
}
public class MysqlConfig { //... 省略其他属性和方法...
}
public class ScheduledUpdater {
  private final ScheduledExecutorService executor =
Executors.newSingleThreadScheduledExecutor();;
  private long initialDelayInSeconds;
  private long periodInSeconds;
  private Updater updater;
  public ScheduleUpdater(Updater updater, long initialDelayInSeconds, long
periodInSeconds) {
      this.updater = updater;
      this.initialDelayInSeconds = initialDelayInSeconds;
      this.periodInSeconds = periodInSeconds;
  }
  public void run() {
```

```
        executor.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                updater.update();
            }
        }, this.initialDelayInSeconds, this.periodInSeconds, TimeUnit.SECONDS);
    }
}
public class Application {
    ConfigSource configSource = new ZookeeperConfigSource(/* 省略参数 */);
    public static final RedisConfig redisConfig = new RedisConfig(configSource);
    public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
    public static final MySqlConfig mysqlConfig = new MysqlConfig(configSource);
    public static void main(String[] args) {
        ScheduledUpdater redisConfigUpdater = new
ScheduledUpdater(redisConfig, 300, 300);
        redisConfigUpdater.run();

        ScheduledUpdater kafkaConfigUpdater = new
ScheduledUpdater(kafkaConfig, 60, 60);
        redisConfigUpdater.run();
    }
}
```

刚刚的热更新的需求我们已经搞定了。现在，我们又有了一个新的监控功能需求。通过命令行来查看 Zookeeper 中的配置信息是比较麻烦的。所以，我们希望能有一种更加方便的配置信息查看方式。

我们可以在项目中开发一个内嵌的 SimpleHttpServer，输出项目的配置信息到一个固定的 HTTP 地址，比如：http://127.0.0.1:2389/config 。我们只需要在浏览器中输入这个地址，就可以显示出系统的配置信息。不过，出于某些原因，我们只想暴露 MySQL 和 Redis 的配置信息，不想暴露 Kafka 的配置信息。

为了实现这样一个功能，我们还需要对上面的代码做进一步改造。改造之后的代码如下所示：

```
public interface Updater {
    void update();
}
public interface Viewer {
    String outputInPlainText();
    Map<String, String> output();
}
```

```
public class RedisConfig implemets Updater, Viewer {
  //... 省略其他属性和方法...
  @Override
  public void update() { //...
  }
  @Override
  public String outputInPlainText() { //...
  }
  @Override
  public Map<String, String> output() { //...
  }
}
public class KafkaConfig implements Updater {
  //... 省略其他属性和方法...
  @Override
  public void update() { //...
  }
}
public class MysqlConfig implements Viewer {
  //... 省略其他属性和方法...
  @Override
  public String outputInPlainText() { //...
  }
  @Override
  public Map<String, String> output() { //...
  }
}
public class SimpleHttpServer {
  private String host;
  private int port;
  private Map<String, List<Viewer>> viewers = new HashMap<>();

  public SimpleHttpServer(String host, int port) {//...
  }

  public void addViewers(String urlDirectory, Viewer viewer) {
    if (!viewers.containsKey(urlDirectory)) {
      viewers.put(urlDirectory, new ArrayList<Viewer>());
    }
    this.viewers.get(urlDirectory).add(viewer);
  }

  public void run() { //...
```

```
  }
}
public class Application {
  ConfigSource configSource = new ZookeeperConfigSource();
  public static final RedisConfig redisConfig = new RedisConfig(configSource);
  public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
  public static final MySqlConfig mysqlConfig = new MySqlConfig(configSource);

  public static void main(String[] args) {
    ScheduledUpdater redisConfigUpdater =
      new ScheduledUpdater(redisConfig, 300, 300);
    redisConfigUpdater.run();

    ScheduledUpdater kafkaConfigUpdater =
      new ScheduledUpdater(kafkaConfig, 60, 60);
    redisConfigUpdater.run();

    SimpleHttpServer simpleHttpServer = new SimpleHttpServer(“127.0.0.1”,
2389);
    simpleHttpServer.addViewer("/config", redisConfig);
    simpleHttpServer.addViewer("/config", mysqlConfig);
    simpleHttpServer.run();
  }
}
```

至此，热更新和监控的需求我们就都实现了。我们来回顾一下这个例子的设计思想。

我们设计了两个功能非常单一的接口：Updater 和 Viewer。ScheduledUpdater 只依赖 Updater 这个跟热更新相关的接口，不需要被强迫去依赖不需要的 Viewer 接口，满足接口隔离原则。同理，SimpleHttpServer 只依赖跟查看信息相关的 Viewer 接口，不依赖不需要的 Updater 接口，也满足接口隔离原则。

你可能会说，如果我们不遵守接口隔离原则，不设计 Updater 和 Viewer 两个小接口，而是设计一个大而全的 Config 接口，让 RedisConfig、KafkaConfig、MysqlConfig 都实现这个 Config 接口，并且将原来传递给 ScheduledUpdater 的 Updater 和传递给 SimpleHttpServer 的 Viewer，都替换为 Config，那会有什么问题呢？我们先来看一下，按照这个思路来实现的代码是什么样的。

```
public interface Config {
  void update();
  String outputInPlainText();
  Map<String, String> output();
}
```

```
public class RedisConfig implements Config {
  //... 需要实现 Config 的三个接口 update/outputIn.../output
}
public class KafkaConfig implements Config {
  //... 需要实现 Config 的三个接口 update/outputIn.../output
}
public class MysqlConfig implements Config {
  //... 需要实现 Config 的三个接口 update/outputIn.../output
}
public class ScheduledUpdater {
  //... 省略其他属性和方法..
  private Config config;
  public ScheduleUpdater(Config config, long initialDelayInSeconds, long
periodInSeconds) {
      this.config = config;
      //...
  }
  //...
}
public class SimpleHttpServer {
  private String host;
  private int port;
  private Map<String, List<Config>> viewers = new HashMap<>();

  public SimpleHttpServer(String host, int port) {//...}

  public void addViewer(String urlDirectory, Config config) {
    if (!viewers.containsKey(urlDirectory)) {
      viewers.put(urlDirectory, new ArrayList<Config>());
    }
    viewers.get(urlDirectory).add(config);
  }

  public void run() { //... }
}
```

这样的设计思路也是能工作的，但是对比前后两个设计思路，在同样的代码量、实现复杂度、同等可读性的情况下，第一种设计思路显然要比第二种好很多。为什么这么说呢？主要有两点原因。

首先，第一种设计思路更加灵活、易扩展、易复用。因为 Updater、Viewer 职责更加单一，单一就意味了通用、复用性好。比如，我们现在又有一个新的需求，开发一个 Metrics 性能统计模块，并且希望将 Metrics 也通过 SimpleHttpServer 显示在网页上，以方便查看。这个时候，尽管 Metrics 跟 RedisConfig 等没有任何关系，但我们仍然可以让 Metrics 类实现非常通用的 Viewer 接口，复用 SimpleHttpServer 的代码实现。具体的代码如下所示：

```
public class ApiMetrics implements Viewer {//...}
public class DbMetrics implements Viewer {//...}
public class Application {
  ConfigSource configSource = new ZookeeperConfigSource();
  public static final RedisConfig redisConfig = new RedisConfig(configSource);
  public static final KafkaConfig kafkaConfig = new KakfaConfig(configSource);
  public static final MySqlConfig mySqlConfig = new MySqlConfig(configSource);
  public static final ApiMetrics apiMetrics = new ApiMetrics();
  public static final DbMetrics dbMetrics = new DbMetrics();

  public static void main(String[] args) {
    SimpleHttpServer simpleHttpServer = new SimpleHttpServer(“127.0.0.1”,
2389);
    simpleHttpServer.addViewer("/config", redisConfig);
    simpleHttpServer.addViewer("/config", mySqlConfig);
    simpleHttpServer.addViewer("/metrics", apiMetrics);
    simpleHttpServer.addViewer("/metrics", dbMetrics);
    simpleHttpServer.run();
  }
}
```

其次，第二种设计思路在代码实现上做了一些无用功。因为 Config 接口中包含两类不相关的接口，一类是 update()，一类是 output() 和 outputInPlainText()。理论上，KafkaConfig 只需要实现 update() 接口，并不需要实现 output() 相关的接口。同理，MysqlConfig 只需要实现 output() 相关接口，并需要实现 update() 接口。但第二种设计思路要求 RedisConfig、KafkaConfig、MySqlConfig 必须同时实现 Config 的所有接口函数（update、output、outputInPlainText）。除此之外，如果我们要往 Config 中继续添加一个新的接口，那所有的实现类都要改动。相反，如果我们的接口粒度比较小，那涉及改动的类就比较少。

> 问：

java.util.concurrent 并发包提供了 AtomicInteger 这样一个原子类，其中有一个函数 getAndIncrement() 是这样定义的：给整数增加一，并且返回未增之前的值。我的问题是，这个函数的设计是否符合单一职责原则和接口隔离原则？为什么？

```
/**
 * Atomically increments by one the current value.
 * @return the previous value
 */
public final int getAndIncrement() {//...}
```

五、依赖倒置原则

关于 SOLID 原则，我们已经学过单一职责、开闭、里式替换、接口隔离这四个原则。今天，我们再来学习最后一个原则：**依赖倒置原则。**

1、原理

依赖倒置原则（Dependency Inversion Principle，简称 DIP）是面向对象设计的五大原则（SOLID）之一。这个原则强调**要依赖于抽象而不是具体实现。**遵循这个原则可以**使系统的设计更加灵活、可扩展和可维护。**

依赖倒置原则有两个关键点：

1. **高层模块不应该依赖于低层模块，它们都应该依赖于抽象。**
2. **抽象不应该依赖于具体实现，具体实现应该依赖于抽象。**

倒置（Inversion）在这里的确是指“反过来”的意思。在依赖倒置原则（Dependency Inversion Principle, DIP）中，我们需要**改变依赖关系的方向**，使得**高层模块和低层模块都依赖于抽象**，而**不是高层模块直接依赖于低层模块。**这样一来，依赖关系就从**直接依赖具体实现“反过来”依赖抽象了。**

在没有应用依赖倒置原则的传统软件设计中，高层模块通常**直接依赖于低层模块。**这会导致系统的**耦合度较高**，**低层模块的变化很容易影响到高层模块。**当我们应用依赖倒置原则时，高层模块和低层模块的依赖关系发生了改变，**它们都依赖于抽象（例如接口或抽象类）**，而不再是高层模块直接依赖于低层模块。这样，我们就实现了依赖关系的“倒置”。

这种“倒置”的依赖关系使得系统的耦合度降低，提高了系统的可维护性和可扩展性。因为当低层模块的具体实现发生变化时，只要不改变抽象，高层模块就不需要进行调整。所以这个原则叫做依赖倒置原则。

2、如何理解抽象

当我们在讨论依赖倒置原则中的抽象时，**绝对不能仅仅把他理解为一个接口**。抽象的目的是将**关注点从具体实现转移到概念和行为**，使得我们在设计和编写代码时能够更加关注问题的本质。通过使用抽象，我们可以创建更加灵活、可扩展和可维护的系统。

事实上抽象是一个很广泛的概念，它可以包括**接口、抽象类**以及**由大量接口，抽象类和实现组成的更高层次的模块**。通过将系统分解为更小的、可复用的组件，我们可以实现更高层次的抽象。这些组件可以独立地进行替换和扩展，从而使整个系统更加灵活。

在依赖倒置原则的背景下，我们可以从以下几个方面理解抽象：

（1）接口

接口是 Java 中实现抽象的一种常见方式。接口定义了一组方法签名，表示实现该接口的类应具备哪些行为。接口本身并不包含具体实现，所以它强调了行为的抽象。

假设我们正在开发一个在线购物系统，其中有一个订单处理模块。订单处理模块需要与**不同的支付服务提供商（如 PayPal、Stripe 等）进行交互**。如果我们直接依赖于支付服务提供商的具体实现，那么在更换支付服务提供商或添加新的支付服务提供商时，我们可能需要对订单处理模块进行大量修改。为了避免这种情况，我们应该依赖于接口而不是具体实现。

首先，我们定义一个支付服务接口：

```
public interface PaymentService {
    boolean processPayment(Order order);
}
```

然后，为每个支付服务提供商实现该接口：

```
public class PayPalPaymentService implements PaymentService {
    @Override
    public boolean processPayment(Order order) {
        // 实现 PayPal 支付逻辑
    }
}

public class StripePaymentService implements PaymentService {
    @Override
    public boolean processPayment(Order order) {
        // 实现 Stripe 支付逻辑
    }
}
```

现在，我们可以在订单处理模块中依赖 `PaymentService` 接口，而不是具体的实现：

```
public class OrderProcessor {
    private PaymentService paymentService;

    public OrderProcessor(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder(Order order) {
        // 其他订单处理逻辑...

        boolean paymentResult = paymentService.processPayment(order);

        // 根据 paymentResult 处理支付结果
    }
}
```

通过这种方式，当我们需要更换支付服务提供商或添加新的支付服务提供商时，只需要提供一个新的实现类，而不需要修改 `OrderProcessor` 类。我们可以在运行时通过构造函数注入不同的支付服务实现，使得系统更加灵活和可扩展。

这个例子展示了如何依赖接口而不是实现来编写代码，从而提高系统的灵活性和可维护性。

（2）抽象类

抽象类是另一种实现抽象的方式。与接口类似，抽象类也可以定义抽象方法，表示子类应该具备哪些行为。不过抽象类还可以包含部分具体实现，这使得它们比接口更加灵活。

```
abstract class Shape {
    abstract double getArea();

    void displayArea() {
        System.out.println("面积为: " + getArea());
    }
}

class Circle extends Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double getArea() {
        return Math.PI * Math.pow(radius, 2);
    }
}

class Square extends Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    @Override
    double getArea() {
        return Math.pow(side, 2);
    }
}
```

在这个示例中，我们定义了一个抽象类 `Shape`，它具有一个抽象方法 `getArea`，用于计算形状的面积。同时，它还包含了一个具体方法 `displayArea`，用于打印面积。`Circle` 和 `Square` 类继承了 `Shape`，分别实现了 `getArea` 方法。在其他类中我们可以依赖抽象Shape而非 Square和Circle。

（3）高层模块

在某些情况下，我们可以通过将系统分解为更小的、可复用的组件来实现抽象。**这些组件可以独立地进行替换和扩展，从而使整个系统更加灵活。**这种抽象方法往往在软件架构和模块化设计中有所体现。

让我们来看另一个关于高层策略抽象的例子：插件化架构，案例中的插件十分简单，仅仅只有一个接口，事实上我们日常实现插件功能时往往比这个复杂的多。

假设我们正在构建一个文本编辑器，它允许用户通过插件来扩展功能。我们可以将插件系统作为一个高层策略抽象，以便于在不修改核心编辑器代码的情况下，添加新功能。

首先，我们可以定义一个插件接口：

```
public interface Plugin {
    String getName();
    void execute();
}
```

然后，我们在文本编辑器的核心代码中，引入插件管理器来负责加载和执行插件：

```
public class TextEditor {
    private List<Plugin> plugins = new ArrayList<>();

    public void loadPlugin(Plugin plugin) {
        plugins.add(plugin);
    }

    public void executePlugins(String name) {
        for (Plugin plugin : plugins) {
            plugin.execute();
        }
    }
}
```

现在，如果我们想要添加一个新功能，例如支持 Markdown 格式的预览，我们可以创建一个实现 `Plugin` 接口的新类：

```
public class MarkdownPreviewPlugin implements Plugin {
    @Override
    public String getName() {
        return "Markdown Preview";
    }

    @Override
    public void execute() {
        // 实现 Markdown 预览功能
    }
}
```

最后，我们可以将新的插件加载到文本编辑器中：

```
public static void main(String[] args) {
    TextEditor editor = new TextEditor();
    editor.loadPlugin(new MarkdownPreviewPlugin());
    editor.executePlugin("Markdown Preview");
}
```

通过采用插件化架构，我们将特定功能的实现与核心编辑器代码解耦，使得整个系统更加灵活和可扩展。这是一个典型的高层策略抽象的应用示例。

有的朋友就说了，这还不是一个接口吗？你能用一个接口写出一个markdown的插件吗？

总之，抽象是一个广泛的概念，它有助于我们将注意力从具体实现转移到行为和概念。在依赖倒置原则的背景下，抽象可以帮助我们实现更加灵活、可扩展和可维护的系统。

3、如何理解高层模块和底层模块

所谓高层模块和低层模块的划分，简单来说就是，在调用链上，调用者属于高层，被调用者属于低层。在平时的业务代码开发中，高层模块依赖底层模块是没有任何问题的。实际上，这条原则主要还是用来指导框架层面的设计，跟前面讲到的控制反转类似。我们拿 Tomcat 这个 Servlet 容器作为例子来解释一下。

从业务代码上讲，举一个简单的例子就是controller要依赖service的接口而不是实现，service实现要依赖dao层的接口而不是实现，调用者要依赖被调用者的接口而不是实现。

以一个简单的音频播放器为例，高层模块 `AudioPlayer` 负责播放音频，而音频文件的解码由低层模块 `Decoder` 实现。为了遵循依赖倒置原则，我们可以引入一个抽象的解码器接口：

```
interface AudioDecoder {
    AudioData decode(String filePath);
}

class AudioPlayer {
    private final AudioDecoder decoder;

    public AudioPlayer(AudioDecoder decoder) {
        this.decoder = decoder;
    }

    public void play(String filePath) {
        AudioData audioData = decoder.decode(filePath);
        // 使用解码后的音频数据进行播放
    }
}

class MP3Decoder implements AudioDecoder {
    @Override
    public AudioData decode(String filePath) {
        // 实现 MP3 文件解码
    }
}
```

在这个例子中，我们将高层模块 `AudioPlayer` 和低层模块 `MP3Decoder` 解耦，使它们都依赖于抽象接口 `AudioDecoder`。这样，我们可以根据需要轻松地更换音频解码器（例如，支持不同的音频格式），而不影响音频播放器的逻辑。为了支持新的音频格式，我们只需要实现新的解码器类，并将其传递给 `AudioPlayer`。

假设我们现在要支持 WAV 格式的音频文件，我们可以创建一个实现 `AudioDecoder` 接口的新类：

```
class WAVDecoder implements AudioDecoder {
    @Override
    public AudioData decode(String filePath) {
        // 实现 WAV 文件解码
    }
}
```

然后，在创建 `AudioPlayer` 对象时，我们可以根据需要选择使用 `MP3Decoder` 或 `WAVDecoder`：

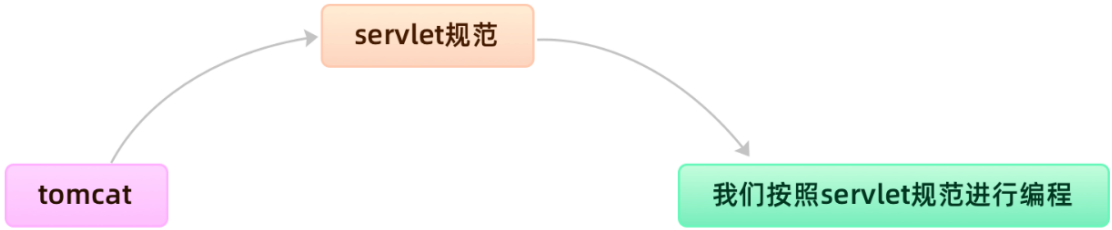
```
public static void main(String[] args) {
    AudioDecoder mp3Decoder = new MP3Decoder();
    AudioPlayer mp3Player = new AudioPlayer(mp3Decoder);
    mp3Player.play("example.mp3");

    AudioDecoder wavDecoder = new WAVDecoder();
    AudioPlayer wavPlayer = new AudioPlayer(wavDecoder);
    wavPlayer.play("example.wav");
}
```

通过遵循依赖倒置原则，我们将高层模块 `AudioPlayer` 与低层模块 `MP3Decoder` 和 `WAVDecoder` 解耦，使它们都依赖于抽象接口 `AudioDecoder`。这样的设计使得我们可以轻松地为音频播放器添加新的音频格式支持，同时保持整个系统的灵活性和可维护性。

再举一个例子，我们从更加高层的一个角度来理解：

Tomcat 是运行 Java Web 应用程序的容器。我们编写的 Web 应用程序代码只需要部署在 Tomcat 容器下，便可以被 Tomcat 容器调用执行。按照之前的划分原则，**Tomcat 就是高层模块**，我们编写的 **Web 应用程序代码就是低层模块。Tomcat 和应用程序代码之间并没有直接的依赖关系，两者都依赖同一个“抽象”，也就是 Sevlet 规范。**Servlet 规范不依赖具体的 Tomcat 容器和应用程序的实现细节，而 Tomcat 容器和应用程序依赖 Servlet 规范。这样做的好处就是tomcat中可以运行任何实现了servlet规范的应用程序，同时我们编写的servlet实现（web）工程也可以运行在不同的web服务器中。



4、IOC容器

我们现在思考：依赖倒置的目的是，**低层模块可以随时替换**，以提高代码的可扩展性。

其实我们学过spring的同学应该都清楚，在spring中实现这个很简单的，我们只需要向容器中注入特定的bean就能切换具体实现。同时我们在编写日常代码时，有意无意的都会**遵循设计原则**，我相信此时此刻不会再有同学问**service层为什么要设计接口了。**

控制反转是一种软件设计原则，它将传统的控制流程颠倒过来，将控制权交给一个中心化的容器或框架。

依赖注入是指不通过 new() 的方式在类内部创建依赖类对象，而是将依赖的类对象在外部创建好之后，通过构造函数、函数参数等方式传递（或注入）给类使用。

通过**控制翻转**和**依赖注入**结合，我们只要保证依赖抽象而不是实现，就能很轻松的替换实现。如给容器注入一个myslq的数据，则所有依赖数据源的部分会自动使用mysql，如果想替换数据源则仅仅需要给容器注入一个新的数据源就好了，不需要修改一行代码。

六、KISS 原则

之前的课程中，我们学习了经典的 **SOLID 原则。**现在，我们开始学习KISS 原则和 YAGNI 原则。其中，KISS 原则比较经典，耳熟能详，我们就从kiss原则入手。

1、理解“KISS 原则”

KISS 原则（Keep It Simple, Stupid）：**KISS 原则强调保持代码简单，易于理解和维护。在编写代码时，应避免使用复杂的逻辑、算法和技术，尽量保持代码简洁明了。**这样可以提高代码的**可读性、可维护性和可测试性。**当然，这并不意味着要牺牲代码的性能和功能，而是要在保证性能和功能的前提下，尽量简化代码实现。

KISS 原则算是一个万金油类型的设计原则，可以应用在很多场景中。它不仅经常用来指导软件开发，还经常用来指导更加广泛的系统设计、产品设计等，比如，冰箱、建筑、iPhone 手机的设计等等。

我们知道，代码的**可读性和可维护性是衡量代码质量非常重要的两个标准。**而 KISS 原则就是保持代码可读和可维护的重要手段。代码足够简单，也就意味着很容易读懂，bug 比较难隐藏。即便出现 bug，修复起来也比较简单。

2、代码简单和代码行数少

冒泡排序：

```
public class BubbleSort {
    public static void main(String[] args) {
        int[] arr = {4, 2, 7, 1, 8, 5};
        bubbleSort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }

    // 冒泡排序算法实现
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        // 外层循环控制遍历次数
        for (int i = 0; i < n - 1; i++) {
            // 内层循环用于比较相邻的元素
            for (int j = 0; j < n - 1 - i; j++) {
                // 如果前一个元素大于后一个元素，则交换位置
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

快速排序：

```
public class QuickSort {
    public static void main(String[] args) {
        int[] arr = {4, 2, 7, 1, 8, 5};
        quickSort(arr, 0, arr.length - 1);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
```

```
    // 快速排序算法实现
    public static void quickSort(int[] arr, int left, int right) {
        if (left < right) {
            // 划分数组
            int pivotIndex = partition(arr, left, right);
            // 分别对左右子数组进行快速排序
            quickSort(arr, left, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, right);
        }
    }

    // 将数组划分为两部分，并返回基准元素的索引
    public static int partition(int[] arr, int left, int right) {
        int pivot = arr[left]; // 选择基准元素
        int i = left, j = right;
        while (i < j) {
            // 从右向左找到一个小于基准的元素
            while (i < j && arr[j] >= pivot) {
                j--;
            }
            // 从左向右找到一个大于基准的元素
            while (i < j && arr[i] <= pivot) {
                i++;
            }
            // 交换两个元素的位置
            if (i < j) {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // 将基准元素和指针相遇位置的元素交换
        arr[left] = arr[i];
        arr[i] = pivot;
        return i;
    }
}
```

我们可以看到，快速排序明显比冒泡排序复杂，写多出了很多行，那么我们能说下边的代码不符合kiss原则吗？

我们将以一个简单的电影票系统为例子，进行两次迭代，以满足KISS原则。在该场景中，我们需要为不同类型的用户（如普通用户、学生、会员等）提供不同的票价折扣。

版本1（不满足KISS原则）：

```
public class MovieTicketSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入用户类型（1.普通用户 2.学生 3.会员）：");
        int userType = scanner.nextInt();
        System.out.println("请输入原票价：");
        double originalPrice = scanner.nextDouble();

        double discountedPrice;
        if (userType == 1) {
            discountedPrice = originalPrice;
        } else if (userType == 2) {
            discountedPrice = originalPrice * 0.8;
        } else if (userType == 3) {
            discountedPrice = originalPrice * 0.5;
        } else {
            System.out.println("输入错误！");
            return;
        }

        System.out.println("折扣后的票价为：" + discountedPrice);
    }
}
```

版本2（经过第一次迭代）：

```
public class MovieTicketSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入用户类型（1.普通用户 2.学生 3.会员）：");
        int userType = scanner.nextInt();
        System.out.println("请输入原票价：");
        double originalPrice = scanner.nextDouble();

        double discountedPrice = getDiscountedPrice(userType, originalPrice);

        if (discountedPrice < 0) {
```

```
            System.out.println("输入错误！");
        } else {
            System.out.println("折扣后的票价为：" + discountedPrice);
        }
    }

    public static double getDiscountedPrice(int userType, double originalPrice) {
        switch (userType) {
            case 1:
                return originalPrice;
            case 2:
                return originalPrice * 0.8;
            case 3:
                return originalPrice * 0.5;
            default:
                return -1;
        }
    }
}
```

在第一次迭代中，我们将计算折扣价格的逻辑提取到一个新的方法`getDiscountedPrice`中，使`main`方法更加简洁。同时，我们使用`switch`语句代替`if-else`语句来处理不同用户类型的折扣计算，使得代码更加清晰易读。

版本3（经过第二次迭代，满足KISS原则）：

```
public class MovieTicketSystem {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int userType = getUserType(scanner);
        double originalPrice = getOriginalPrice(scanner);

        double discountedPrice = getDiscountedPrice(userType, originalPrice);

        if (discountedPrice < 0) {
            System.out.println("输入错误！");
        } else {
            System.out.println("折扣后的票价为：" + discountedPrice);
        }
    }

    public static int getUserType(Scanner scanner) {
        System.out.println("请输入用户类型（1.普通用户 2.学生 3.会员）：");
```

```
        return scanner.nextInt();
    }

    public static double getOriginalPrice(Scanner scanner) {
        System.out.println("请输入原票价：");
        return scanner.nextDouble();
    }

    public static double getDiscountedPrice(int userType, double originalPrice) {
        switch (userType) {
            case 1:
                return originalPrice;
            case 2:
                return originalPrice * 0.8;
            case 3:
                return originalPrice * 0.5;
            default:
                return -1;
        }
    }
}
```

经过两次迭代，我们将原始代码的不同功能部分进行了拆分和重构，使得整个程序的结构更加清晰和简洁，易于理解和维护。这就是遵循KISS原则的一个典型例子。

3、代码逻辑复杂就违背 KISS 原则吗?

并非所有的复杂代码逻辑都违背了KISS原则。KISS原则强调的是保持代码简洁、易于理解和维护。有时候，为了实现某个功能，确实需要一定程度的复杂逻辑。关键在于如何尽可能地让代码简单和清晰。

遵循KISS原则的复杂代码应当具备以下特点：

1. 模块化：将复杂的代码逻辑拆分成多个简单、独立的模块，每个模块负责一个特定的功能。这有助于降低代码的复杂度，提高代码的可读性和可维护性。
2. 清晰的命名：为变量、方法、类等使用清晰、有意义的命名，以便于其他人（或未来的你）阅读和理解代码。
3. 注释和文档：为复杂的代码逻辑编写清晰、详细的注释和文档，解释代码的作用和实现原理。这有助于其他人（或未来的你）更容易地理解和维护代码。
4. 避免不必要的复杂度：尽量避免引入不必要的复杂性，如使用过于复杂的算法或数据结构。在实现功能的同时，要考虑代码的简洁性和可读性。

总之，遵循KISS原则并不意味着代码必须简单到极致。相反，它鼓励我们在实现功能的同时，尽量保持代码的简洁、易于理解和维护。一个良好的开发者应当在实际项目中找到适当的平衡点，既满足功能需求，又不过分增加代码的复杂度。

4、如何写出满足 KISS 原则的代码?

实际上，我们前面已经讲到了一些方法。这里我稍微总结一下。

- 不要使用同事可能不懂的技术来实现代码。比如前面例子中的正则表达式，还有一些编程语言中过于高级的语法等。
- 不要重复造轮子，要善于使用已经有的工具类库。经验证明，自己去实现这些类库，出 bug 的概率会更高，维护的成本也比较高。
- 不要过度优化。不要过度使用一些奇技淫巧（比如，位运算代替算术运算、复杂的条件语句代替 if-else、使用一些过于底层的函数等）来优化代码，牺牲代码的可读性。

实际上，代码是否足够简单是一**个挺主观的评判**。同样的代码，有的人觉得简单，有的人觉得不够简单。而往往自己编写的代码，**自己都会觉得够简单**。所以，评判代码是否简单，还有一个很有效的间接方法，那就是 code review。如果在 code review 的时候，同事对你的代码有很多疑问，那就说明你的代码有可能不够“简单”，需要优化啦。

小问题：你怎么看待在开发中重复造轮子这件事情？什么时候要重复造轮子？什么时候应该使用现成的工具类库、开源框架？

七、 DRY 原则

DRY 原则（Don't Repeat Yourself）：**DRY 原则强调避免代码重复**，尽量将相似的代码和逻辑提取到共享的方法、类或模块中。遵循 DRY 原则可以**减少代码的冗余和重复**，提高代码的**复用性和可维护性**。当需要修改某个功能时，只需修改对应的共享代码，而**无需在多处进行相同的修改**。这有助于降低维护成本，提高开发效率。

这条看似简单的原则，实则很难把控，设计原则不是1+1，有些代码在有些场景下就是符合某些原则的，在其他场景下就是不符合的，我们学习了dry原则，不能狭隘的理解。不是说只要两段代码长得一样，那就是违反 DRY 原则，同时有些看似不重复的代码也有可能会违反DRY原则。

1、实现方法

在 Java 编程中，我们可以通过以下方法遵循 DRY 原则：

（1）**使用方法（functions）**：当你发现自己在多处重复相同的代码时，可以将其抽取为一个方法，并在需要的地方调用该方法。

例如：

```
public class DryExample {
    public static void main(String[] args) {
        printHello("张三");
        printHello("李四");
    }

    public static void printHello(String name) {
        System.out.println("你好，" + name + "!");
    }
}
```

在这个例子中，我们使用 `printHello` 方法避免了重复的 `System.out.println` 语句。

（2）**使用继承和接口**：当多个类具有相似的行为时，可以使用继承和接口来抽象共享的功能，从而减少重复代码。

例如：

```
public abstract class Animal {
    public abstract void makeSound();

    public void eat() {
        System.out.println("动物在吃东西");
    }
}

public class Dog extends Animal {
    public void makeSound() {
        System.out.println("汪汪");
    }
}

public class Cat extends Animal {
    public void makeSound() {
```

```
        System.out.println("喵喵");
    }
}
```

在这个例子中，我们使用抽象类 `Animal` 和继承来避免在 `Dog` 和 `Cat` 类中重复 `eat` 方法的代码。

（3）**重用代码库和框架：**使用成熟的代码库和框架可以避免从零开始编写一些通用功能。例如，使用 Java 标准库、Apache Commons 或 Google Guava 等库。

遵循 DRY 原则可以帮助我们编写更高质量的代码，并更容易进行维护和扩展。同时，要注意不要过度优化，以免影响代码的可读性和理解性。

2、只要两段代码长得一样，那就是违反 DRY 原则吗

不一定。DRY 原则的核心思想是**减少重复代码**，以提高代码的**可维护性、可读性和可重用性。**然而，并不是所有看起来相似的代码都违反了 DRY 原则。在某些情况下，重复的代码片段可能**具有完全不同的逻辑含义，因此将它们合并可能会导致误解。**

在评估是否违反了 DRY 原则时，你需要考虑以下几点：

1. 逻辑一致性：如果两段代码的逻辑和功能是一致的，那么将它们合并为一个方法或类是有意义的。如果它们实际上是在**执行不同的任务**，那么合并它们可能导致难以理解的代码。
2. 可维护性：如果将两段看似相同的代码合并可能导致难以维护的代码（例如，增加了过多的条件判断），那么保留一些重复可能是更好的选择。
3. 变更影响：考虑未来的需求变更。如果两段看似相同的代码很可能在未来分别发生变化，那么将它们合并可能导致更多的维护负担。在这种情况下，保留一些重复代码可能是更实际的选择。

总之，判断是否违反了 DRY 原则需要权衡多个因素。关键在于**寻找适当的平衡点，以提高代码质量，同时确保可维护性和可读性。**

以下是一些从不同角度说明两段看似相似或相同的代码并不违反 DRY 原则的示例：

（1）不同的业务逻辑：

```
public class Example1 {
    public static void main(String[] args) {
        double priceAfterDiscount1 = applyClothingDiscount(100);
        double priceAfterDiscount2 = applyElectronicsDiscount(100);

        System.out.println("服装打折后的价格: " + priceAfterDiscount1);
```

```
        System.out.println("电子产品打折后的价格: " + priceAfterDiscount2);
    }

    // 服装折扣逻辑
    public static double applyClothingDiscount(double originalPrice) {
        return originalPrice * 0.9; // 10% 折扣
    }

    // 电子产品折扣逻辑
    public static double applyElectronicsDiscount(double originalPrice) {
        return originalPrice * 0.8; // 20% 折扣
    }
}
```

尽管 `applyClothingDiscount` 和 `applyElectronicsDiscount` 方法的代码看起来相似，我们甚至可以将他们合并成一个方法，当其实并不建议这样去做，因为它们代表了不同的业务逻辑。因此，将它们合并并不符合 DRY 原则，结果这可能导致**代码难以理解和维护**，比如以后想针对服装或者电器做特殊的折扣逻辑如何合并了就会比较麻烦。

（2）专门用于不同目的的方法：

```
public class Example2 {
    public static void main(String[] args) {
        double baseFare = 50;
        double fareWithTax1 = addSalesTax(baseFare);
        double fareWithTax2 = addVatTax(baseFare);

        System.out.println("含销售税的费用: " + fareWithTax1);
        System.out.println("含增值税的费用: " + fareWithTax2);
    }

    // 添加消费税
    public static double addSalesTax(double baseFare) {
        return baseFare + (baseFare * 0.05); // 5% 销售税
    }

    // 添加增值税
    public static double addVatTax(double baseFare) {
        return baseFare + (baseFare * 0.1); // 10% 增值税
    }
```

```
}
```

虽然 `addSalesTax` 和 `addVatTax` 方法的代码看起来相似，代码结构几乎雷同，但是他们两个都有不同目的，一个是计算增值税，一个计算消费税。在这种情况下，将它们合并可能导致代码混乱，因此保留这些看似相似的代码段是合理的。

（3）针对不同数据类型的操作：

```
public class Example3 {
    public static void main(String[] args) {
        int[] intArray = {1, 2, 3, 4, 5};
        double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5};

        int intSum = sum(intArray);
        double doubleSum = sum(doubleArray);

        System.out.println("整数数组的和: " + intSum);
        System.out.println("浮点数数组的和: " + doubleSum);
    }

    // 求整数数组之和
    public static int sum(int[] array) {
        int sum = 0;
        for (int i : array) {
            sum += i;
        }
        return sum;
    }

    // 求浮点数数组之和
    public static double sum(double[] array) {
        double sum = 0;
        for (double d : array) {
            sum += d;
        }
        return sum;
    }
}
```

在这个示例中，我们有两个求和方法：`sum(int[] array)` 和 `sum(double[] array)`。虽然这两个方法的代码看起来相似，但它们处理的数据类型不同（一个处理整数数组，另一个处理浮点数数组）。由于 Java 不支持泛型数组，我们不能简单地使用一个泛型方法来替代这两个方法。因此，在这种情况下，保留这些看似相似的代码段是合理的。
在这些示例中，我们展示了一些不同情况下看似相似或相同的代码，并没有违反 DRY 原则。这些例子说明了我们在实际编程中需要根据具体情况权衡各种因素，以找到合适的平衡点。

三、什么样的代码违反了DRY原则呢?

1、功能重复

我们现在看一个例子。在同一个项目代码中有下面两个函数：validatePhone() 和 validatePhoneNum()。尽管两个函数的命名不同，实现逻辑不同，但功能是一样的，都是用来校验手机号是否满足条件。

之所以在同一个项目中会有两个功能相同的函数，那是因为这两个函数是由两个不同的小伙伴开发的，其中一个小伙伴在不知道已经有了 validatePhone() 的情况下，自己又定义并实现了同样的函数。

一个小伙伴使用正则表达式来校验手机号：

```
/**
 * 使用正则表达式验证手机号是否合法
 *
 * @param phoneNum 待验证的手机号
 * @return boolean 验证结果，true表示合法，false表示不合法
 */
public static boolean validatePhone(String phoneNum) {
    // 定义手机号的正则表达式
    String regex = "^1[3456789]\\d{9}$";
    // 利用String类的matches方法进行正则匹配
    return phoneNum.matches(regex);
}
```

另一个小伙伴不使用正则表达式来校验手机号：

```
/**
 * 不使用正则表达式验证手机号是否合法
 *
 * @param phoneNum 待验证的手机号
```

```
     * @return boolean 验证结果，true表示合法，false表示不合法
     */
    public static boolean validatePhoneNum(String phoneNum) {
        // 首先判断手机号长度是否为11位
        if (phoneNum.length() != 11) {
            return false;
        }
        // 然后逐个字符判断是否都是数字
        for (int i = 0; i < phoneNum.length(); i++) {
            char c = phoneNum.charAt(i);
            if (c < '0' || c > '9') {
                return false;
            }
        }
        // 最后判断手机号是否以1开头
        return phoneNum.startsWith("1");
    }
```

在这个例子中，我们的代码书写并不相同，但是他表达的含义，实现的功能却是重复的，我们认为它违反了 DRY 原则。我们应该在项目中，应该将相同功能的方法统一处理，如果不统一，将来手机校验规则一旦发生改变，而同学只知道validatePhoneNum函数的存在，并对其做了修改，而使用validatePhone的地方就可能**留下祸患**。

2、代码执行重复

我们再来看一个例子。其中，UserService 中 login() 函数用来校验用户登录是否成功。如果失败，就返回异常；如果成功，就返回用户信息。具体代码如下所示：

```
public class UserService {
    private UserDao UserDao;// 通过依赖注入或者 IOC 框架注入
    public User login(String phone, String password) {
        // 检查数据库是否存在该用户
        boolean existed = UserDao.checkIfUserExisted(phone, password);
        if (!existed) {
            // ... throw AuthenticationFailureException...
        }
        // 通过手机查找用户
        User user = UserDao.getUserByPhone(phone);
        return user;
    }
```

```
}
public class UserDao {
   // 检查用户是否存在
   public boolean checkIfUserExisted(String phone, String password) {
      if (!PhoneValidation.validate(phone)) {
         // ... throw InvalidPhoneException...
      }
      if (!PasswordValidation.validate(password)) {
         // ... throw InvalidPasswordException...
      }
      //...query db to check if phone&password exists...
   }

   // 去数据库查询用户
   public User getUserByPhone(String phone) {
      if (!PhoneValidation.validate(phone)) {
         // ... throw InvalidPhoneException...
      }
      //...query db to get user by phone...
   }
}
```

现在大家开始思考，以上的代码存在什么问题呢?

1、相同的代码有没有被执行多次?

2、相似的逻辑有没有被执行多次?

细心的同学一定发现了:

问题一: 在checkIfUserExisted和getUserByPhone中都调用了PhoneValidation.validate(phone)方法，这显然不满足DRY原则

问题二: 查询用户的操作执行了两次，显然，这是一个重复操作，IO操作时很浪费资源的，这一点一定要谨记。

修改如下，这其实也是我们日常写的最多的代码，只是如果我们不注意可能很多地方就会写的出现这样或那样的问题:

```
public class UserService {
   private UserDao UserDao;// 通过依赖注入或者 IOC 框架注入
   public User login(String phone, String password) {
      if (!PhoneValidation.validate(phone)) {
```

```
            // ... throw InvalidPhoneException...
        }
        if (!PasswordValidation.validate(password)) {
            // ... throw InvalidPasswordException...
        }
        User user = UserDao.getUserByEmail(phone);
        if (user == null || !password.equals(user.getPassword()) {
            // ... throw AuthenticationFailureException...
        }
        return user;
    }
}


public class UserDao {
    public boolean checkIfUserExisted(String phone, String password) {
        //...query db to check if phone&password exists
    }
    public User getUserByPhone(String phone) {
        //...query db to get user by phone...
    }
}
```

小节：DRY原则（Don't Repeat Yourself），即不要重复自己的原则，是软件工程中非常重要的一条设计原则。它强调在软件开发中避免重复代码，减少代码的冗余性，提高代码的可维护性、可读性和可扩展性。DRY原则的核心思想是避免重复的代码，尽可能将重复的代码封装成可重用的模块、函数、类等，提高代码的复用性，降低代码的耦合性。

DRY原则有以下几个要点：

1. 避免重复的代码：尽可能减少代码的冗余和重复，将重复的代码封装成可重用的函数、类、模块等。
2. 提高代码的可维护性：避免重复代码可以降低代码的冗余性和复杂度，提高代码的可维护性和可读性。
3. 降低代码的耦合性：通过避免重复代码，可以减少代码之间的耦合度，提高代码的灵活性和可扩展性。
4. 提高代码的复用性：通过将重复的代码封装成可重用的函数、类、模块等，提高代码的复用性，减少代码的编写和维护成本。

总之，DRY原则是一个非常实用的设计原则，可以在软件开发过程中帮助开发者减少代码的冗余和重复，提高代码的可维护性和可读性，降低代码的耦合度和维护成本，从而实现高效、可靠、可维护的软件系统。

八、迪米特法则

今天，我们讲最后一个设计原则：迪米特法则。

1、理论原理

迪米特法则（Law of Demeter, LoD），又称**最少知识原则**（Least Knowledge Principle, LKP），是一种面向对象编程设计原则。它的核心思想是：一个对象应该**尽量少地了解其他对象，降低对象之间的耦合度**，从而提高代码的可维护性和可扩展性。

关于这个设计原则，我们先来看一下它最原汁原味的英文定义：

> Each unit should have only limited knowledge about other units: only units “closely” related to the current unit. Or: Each unit should only talk to its friends; Don’t talk to strangers.

我们把它直译成中文，就是下面这个样子：

> 每个模块（unit）只应该了解那些与它关系密切的模块（units: only units “closely” related to the current unit）的有限知识（knowledge）。或者说，每个模块只和自己的朋友“说话”（talk），不和陌生人“说话”（talk）。

我们之前讲过，大部分设计原则和思想都非常抽象，有各种各样的解读，要想灵活地应用到实际的开发中，需要有实战经验的积累。迪米特法则也不例外。所以，我结合我自己的理解和经验，我们可以说的更加直白一点：**两个类之间尽量不要直接依赖，如果必须依赖，最好只依赖必要的接口。**

迪米特法则的主要指导原则如下：

- 类和类之间**尽量不直接依赖。**
- 有依赖关系的类之间，尽量只依赖**必要的接口。**

从上面的描述中，我们可以看出，迪米特法则包含前后两部分，这两部分讲的是两件事情，我用两个实战案例分别来解读一下。

2、类之间不直接依赖

"不该有直接依赖关系的类之间，不要有依赖"这个原则强调的是降低类与类之间的耦合度，避免不必要的依赖。这通常意味着我们应该使用抽象（如接口或抽象类）来解耦具体实现。让我们通过一个例子来理解这个原则。

假设我们有一个简单的报告生成系统，它需要从不同类型的数据源（如数据库、文件、API等）获取数据，并输出不同格式的报告（如CSV、JSON、XML等）。以下是一个不遵循这一原则的实现：

```
// 具体的数据库类
class Database {
    public String fetchData() {
        // 从数据库中获取数据
        return "data from database";
    }
}

// 具体的报告生成类
class ReportGenerator {
    private Database database;

    public ReportGenerator(Database database) {
        this.database = database;
    }

    public String generateCSVReport() {
        String data = database.fetchData();
        // 将数据转换为CSV格式
        return "CSV report: " + data;
    }
}
```

在上述实现中，`ReportGenerator` 类直接依赖于具体的 `Database` 类。这意味着如果我们想从其他类型的数据源（如文件）获取数据，或者使用不同的数据库实现，我们需要修改 `ReportGenerator` 类。这违反了开闭原则（对扩展开放，对修改封闭），并增加了类与类之间的耦合。

为了遵循 "不该有直接依赖关系的类之间，不要有依赖"原则，我们可以引入抽象来解耦具体实现。下面是一个修改后的实现：

```
// 数据源接口
interface DataSource {
```

```
    String fetchData();
}

// 具体的数据库类
class Database implements DataSource {
    @Override
    public String fetchData() {
        // 从数据库中获取数据
        return "data from database";
    }
}

// 具体的文件类
class FileDataSource implements DataSource {
    @Override
    public String fetchData() {
        // 从文件中获取数据
        return "data from file";
    }
}

// 报告生成类
class ReportGenerator {
    private DataSource dataSource;

    public ReportGenerator(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public String generateCSVReport() {
        String data = dataSource.fetchData();
        // 将数据转换为CSV格式
        return "CSV report: " + data;
    }
}
```

在修改后的实现中，我们引入了 `DataSource` 接口，并使 `ReportGenerator` 类依赖于该接口，而不是具体的实现。这样，我们可以轻松地为报告生成器添加新的数据源类型，而无需修改现有代码。

通过引入抽象，我们遵循了 "不该有直接依赖关系的类之间，不要有依赖"原则，降低了类与类之间的耦合，

3、只依赖必要的接口

"有依赖关系的类之间，尽量只依赖必要的接口"这个原则强调的是，当一个类需要依赖另一个类时，应该尽可能地**依赖于最小化的接口**。这可以降低类与类之间的耦合，提高系统的可扩展性和灵活性。

我们还是以上边的用户信息管理案例给大家讲解：

```
public interface UserService {
    boolean register(String cellphone, String password);
    boolean login(String cellphone, String password);
    UserInfo getUserInfoById(long id);
    UserInfo getUserInfoByCellphone(String cellphone);
}
public interface RestrictedUserService {
    boolean deleteUserByCellphone(String cellphone);
    boolean deleteUserById(long id);
}
public class UserServiceImpl implements UserService, RestrictedUserService {
    // ... 省略实现代码...
}
```

对于绝大部分场景，我们可能只关心和删除无关的方法，如UserController，所以他只需要依赖他所需要的接口UserService即可：

```
public class UserController{
    UserService userService;
    // ... 省略实现代码...
}
```

然而用户管理员需要更多的权限，我们则可以通过组合的形式来实现，让其依赖两个必要的接口：

```
public class UserManagerController{
    UserService userService;
    RestrictedUserService restrictedUserService;
    // ... 省略实现代码...
}
```

再举一个例子，假如我们要开一个飞行比赛，我们可以写出如下的案例来满足迪米特法则：

```
// 飞行行为接口
public interface Flyable {
    void fly();
}

// 基类：鸟类
public class Bird {
}

// 子类：能飞的鸟类
public class Sparrow extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("sparrow can fly");
    }
}

// 子类：飞机
public class Plane implements Flyable {
    @Override
    public void fly() {
        System.out.println("plane can fly");
    }
}

// 子类：企鹅类，不实现Flyable接口
public class Penguin extends Bird {
}

//
public class AirRace {
    List<Flyable> list;

    public void addFlyable(Flyable flyable){
        list.add(flyable);
    }

    // ...
}
```

这种实现符合 "有依赖关系的类之间，尽量只依赖必要的接口" 原则，降低了类与类之间的耦合，提高了系统的可扩展性和灵活性。

4、辩证思考与灵活应用

在实际工作中，确实需要在不同的设计原则之间进行权衡。迪米特法则（Law of Demeter，LoD）是一种有助于降低类之间耦合度的原则，但过度地应用迪米特法则可能导致代码变得复杂和难以维护。因此，在实际项目中，我们应该根据具体的场景和需求灵活地应用迪米特法则。以下是一些建议：

1. 避免过度封装：尽管迪米特法则强调类之间的低耦合，但是过度封装可能导致系统变得难以理解和维护。当一个类需要访问另一个类的属性或方法时，我们应该权衡封装的成本和收益，而不是盲目地遵循迪米特法则。
2. 拒绝过度解耦：在实际项目中，过度解耦可能导致大量的中间层和传递性调用。当一个类需要访问另一个类的方法时，如果引入大量的中间层会导致系统变得复杂和低效，那么我们应该考虑放宽迪米特法则的约束。
3. 与其他设计原则和模式相结合：在实际项目中，我们应该灵活地将迪米特法则与其他设计原则（如单一职责原则、开闭原则等）和设计模式（如外观模式、代理模式等）相结合。这样可以使我们在降低耦合度的同时，保持代码的可读性、可维护性和可扩展性。
4. 考虑实际需求和场景：在应用迪米特法则时，我们应该关注实际的需求和场景。如果一个项目的需求和场景较为简单，那么过度地应用迪米特法则可能导致不必要的开发成本。相反，如果一个项目的需求和场景较为复杂，那么遵循迪米特法则可能有助于提高系统的稳定性和可维护性。

总之，在实际项目中，我们应该根据需求和场景灵活地应用迪米特法则，既要降低类之间的耦合度，又要保持代码的可读性、可维护性和可扩展性。这样，我们才能够更好地满足项目的需求，提高软件的质量和效率。

小问题：在今天的讲解中，我们提到了“高内聚、松耦合”“单一职责原则”“接口隔离原则”“基于接口而非实现编程”“迪米特法则”，你能总结一下它们之间的区别和联系吗？

设计原则是你在绝大部分场景中应该这样做，不是必须这么做。（怎么样写出优秀的代码？）

1、职责要单一，类和接口的粒度要适中，类要有内聚性 （单一职责，接口隔离） 高内聚

2、面向抽象编程，要依赖抽象，而不依赖具体 要解耦合、（开闭原则，依赖倒置，迪米特法则） 低耦合 可扩展，可维护

3、类和类之间的关联要紧凑，只依赖必要的接口（迪米特法则） 低耦合

4、组合优于继承，继承后最好不要重写（最好保证继承的作用是代码复用），如果重写不要采用一些手段禁用方法（里式替换原则）

5、不要写重复的代码，代码实现（在保证功能和性能的前提下）最好简单一点，多用大家常用的库，最好不用大家不熟悉的语法，最好不要自己造轮子，写代码禁止花里胡哨，编写好的注释和文档，使用好的命名规范。（kiss，dry） 可读性 安全性