参考：

1. 主：王争课

课url：<https://time.geekbang.org/column/intro/250>

Github：<https://github.com/wangzheng0822?tab=repositories>

1. 辅

Wiki：<https://en.wikipedia.org/wiki/Object-oriented_programming>

注1：这里按王争课的目录整理

注2：这里的设计原则+设计模式都是指面向对象的

# 设计模式学习导读

## 评价代码质量好坏的维度

<https://time.geekbang.org/column/article/160985>

### 【1】 如何评价代码质量高低

### 【2】 最常用的评价标准

#### 可读性：readability

<https://time.geekbang.org/column/article/160985>

##### 1） 可读性的重要性

软件设计大师 Martin Fowler 曾经说过：“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”翻译成中文就是：“任何傻瓜都会编写计算机能理解的代码。好的程序员能够编写人能够理解的代码。”Google 内部甚至专门有个认证就叫作 Readability。只有拿到这个认证的工程师，才有资格在 code review 的时候，批准别人提交代码。可见代码的可读性有多重要，毕竟，代码被阅读的次数远远超过被编写和执行的次数。

我个人认为，代码的可读性应该是评价代码质量最重要的指标之一。我们在编写代码的时候，时刻要考虑到代码是否易读、易理解。除此之外，代码的可读性在非常大程度上会影响代码的可维护性。毕竟，不管是修改 bug，还是修改添加功能代码，我们首先要做的事情就是读懂代码。代码读不大懂，就很有可能因为考虑不周全，而引入新的 bug。

##### 2） 可读性的评价指标

既然可读性如此重要，那我们又该如何评价一段代码的可读性呢？

我们需要看代码是否符合编码规范、命名是否达意、注释是否详尽、函数是否长短合适、模块划分是否清晰、是否符合高内聚低耦合等等。你应该也能感觉到，从正面上，我们很难给出一个覆盖所有评价指标的列表。这也是我们无法量化可读性的原因。

##### 3） 使用code review

实际上，code review 是一个很好的测验代码可读性的手段。如果你的同事可以轻松地读懂你写的代码，那说明你的代码可读性很好；如果同事在读你的代码时，有很多疑问，那就说明你的代码可读性有待提高了。

##### 4） 实例

###### 1》 例1

<https://hio.oppo.com/app/module/detail?tkh_id=2390625&itm_id=11079&mod_id=283710> 



#### 可扩展性：extensibility

<https://time.geekbang.org/column/article/160985>

##### 1） 可扩展性的重要性

可扩展性也是一个评价代码质量非常重要的标准。它表示我们的代码应对未来需求变化的能力。跟可读性一样，代码是否易扩展也很大程度上决定代码是否易维护。

##### 2） 可扩展性的定义

那到底什么是代码的可扩展性呢？

**代码的可扩展性**表示，我们在不修改或少量修改原有代码的情况下，通过扩展的方式添加新的功能代码。说直白点就是，代码预留了一些功能扩展点，你可以把新功能代码，直接插到扩展点上，而不需要因为要添加一个功能而大动干戈，改动大量的原始代码。

##### 3） 扩展性和开闭原则

关于代码的扩展性，在后面讲到“对修改关闭，对扩展开放”这条设计原则的时候，我会来详细讲解，今天我们只需要知道，代码的可扩展性是评价代码质量非常重要的标准就可以了。

#### 可复用性：reusability

##### 什么是代码复用

https://time.geekbang.org/column/article/179607

代码复用表示一种行为：我们在开发新功能的时候，尽量复用已经存在的代码。

##### 什么是代码可复用性

https://time.geekbang.org/column/article/179607

代码的可复用性表示一段代码可被复用的特性或能力：我们在编写代码的时候，让代码尽量可复用。

##### 例子

https://time.geekbang.org/column/article/160985

比如，当讲到面向对象特性的时候，我们会讲到继承、多态存在的目的之一，就是为了提高代码的可复用性；当讲到设计原则的时候，我们会讲到单一职责原则也跟代码的可复用性相关；当讲到重构技巧的时候，我们会讲到解耦、高内聚、模块化等都能提高代码的可复用性。可见，可复用性也是一个非常重要的代码评价标准，是很多设计原则、思想、模式等所要达到的最终效果。

##### 与DRY原则关系

https://time.geekbang.org/column/article/160985

实际上，代码可复用性跟 DRY（Don’t Repeat Yourself）这条设计原则的关系挺紧密的，所以，在后面的章节中，当我们讲到 DRY 设计原则的时候，我还会讲更多代码复用相关的知识，比如，“有哪些编程方法可以提高代码的复用性”等。

#### 健壮性/鲁棒性：robustness

##### 1） 健壮性的定义

###### 参1：<https://blog.csdn.net/bigpudding24/article/details/49069805>

健壮性是指软件对于规范要求以外的输入情况的处理能力。

所谓健壮的系统是指对于规范要求以外的输入能够判断出这个输入不符合规范要求，并能有合理的处理方式。

另外健壮性有时也和容错性【关于容错见“”】，可移植性，正确性有交叉的地方：

（1） 比如，一个软件可以从错误的输入推断出正确合理的输入，这属于容错性量度标准，但是也可以认为这个软件是健壮的

（2） 一个软件可以正确地运行在不同环境下，则认为软件可移植性高，也可以叫，软件在不同平台下是健壮的

（3） 一个软件能够检测自己内部的设计或者编码错误，并得到正确的执行结果，这是软件的正确性标准，但是也可以说，软件有内部的保护机制，是模块级健壮的

###### 参2：

<https://en.wikipedia.org/wiki/Robustness_(computer_science)#Robust_programming>

Robust programming is a style of programming that focuses on handling unexpected termination and unexpected actions.[[7]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-robust_programming-7) It requires code to handle these terminations and actions gracefully by displaying accurate and unambiguous [error messages](https://en.wikipedia.org/wiki/Error_message). These error messages allow the user to more easily debug the program.：健壮编程是一种专注于处理意外终止和意外操作的编程风格。它需要代码通过显示准确和明确的错误消息来优雅地处理这些终止和操作。这些错误消息允许用户更容易地调试程序。

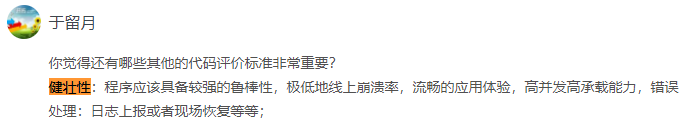
###### 参3：<https://en.wikipedia.org/wiki/Defensive_programming>

【源自<https://en.wikipedia.org/wiki/Robustness_(computer_science)>的

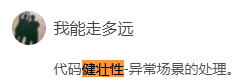
】

###### 参4：<https://time.geekbang.org/column/article/160985>

1》



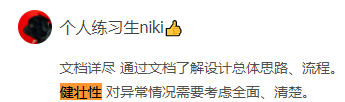
2》



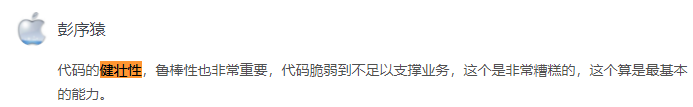
3》



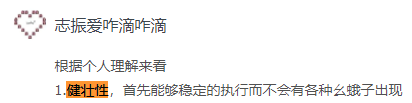
4》



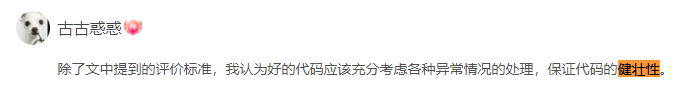
5》



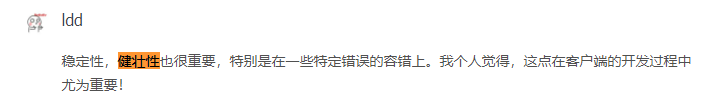
6》



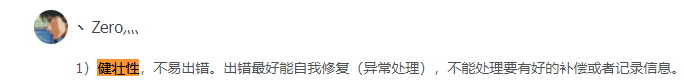
7》



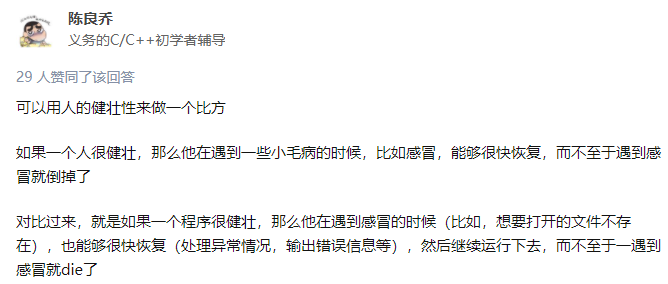
8》



9》



###### 参5：<https://www.zhihu.com/question/20589437>



###### 参6：<https://blog.qingtian16265.com/2021/02/26/program-fault-tolerance-and-code-robustness/>

从程序容错角度也会引申出一个更大的话题，那就是代码的健壮性。

代码的健壮性很多时候是依赖于程序容错的处理。换句话说就是：

程序容错处理的得当与否直接决定了代码的健壮性。

它对代码健壮性的影响是极其深远的。

###### 参7：计算机系统角度

<https://en.wikipedia.org/wiki/Robustness_(computer_science)>

<https://zh.wikipedia.org/wiki/%E5%81%A5%E5%A3%AE%E6%80%A7_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)>

In [computer science](https://en.wikipedia.org/wiki/Computer_science), **robustness** is the ability of a computer system to cope with errors during execution[[1]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-1)[[2]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-IEEE-2) and cope with erroneous input.

Robustness can encompass many areas of computer science, such as [robust programming](https://en.wikipedia.org/wiki/Defensive_programming), [robust machine learning](https://en.wikipedia.org/wiki/Overfitting), and [Robust Security Network](https://en.wikipedia.org/wiki/Robust_Security_Network).

##### 2） 健壮性的重要性

<https://blog.csdn.net/bigpudding24/article/details/49069805>

软件健壮性是一个比较模糊的概念，但是却是非常重要的软件外部量度标准。软件设计的健壮与否直接反应了分析设计和编码人员的水平。即所谓的高手写的程序不容易死。

<https://time.geekbang.org/column/article/160985>



##### 3） 如何保证健壮性

###### 参1：[https://en.wikipedia.org/wiki/Robustness\_(computer\_science)#Principles](https://en.wikipedia.org/wiki/Robustness_(computer_science)" \l "Principles)

**Paranoia：偏执狂**

When building software, the programmer assumes users are out to break their code.[[7]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-robust_programming-7) The programmer also assumes that their own written code may fail or work incorrectly.：当构建软件时，程序员会假设用户会破坏他们的代码。程序员还假定他们自己编写的代码可能会失败或无法正常工作。

**Stupidity：愚蠢**

The programmer assumes users will try incorrect, bogus and malformed inputs.[[7]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-robust_programming-7) As a consequence, the programmer returns to the user an unambiguous, intuitive error message that does not require looking up error codes. The error message should try to be as accurate as possible without being misleading to the user, so that the problem can be fixed with ease.：程序员假设用户将尝试不正确的、伪造的和格式不正确的输入。因此，程序员返回给用户一个明确的、直观的错误消息，不需要查找错误代码。错误消息应该尽可能准确，不会误导用户，这样问题就可以很容易地解决。

**Dangerous implements：危险的实现**

Users should not gain access to [libraries](https://en.wikipedia.org/wiki/Library_(computing)), [data structures](https://en.wikipedia.org/wiki/Data_structure), or [pointers](https://en.wikipedia.org/wiki/Pointer_(computer_programming)) to data structures.[[7]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-robust_programming-7) This information should be hidden from the user so that the user doesn't accidentally modify them and introduce a bug in the code. When such [interfaces](https://en.wikipedia.org/wiki/Interface_(object-oriented_programming)) are correctly built, users use them without finding loopholes to modify the interface. The interface should already be correctly implemented, so the user does not need to make modifications. The user therefore focuses solely on their own code.：用户不应该访问库、数据结构或指向数据结构的指针。这些信息应该对用户隐藏起来，这样用户就不会无意中修改它们，从而在代码中引入错误。当这些接口被正确构建时，用户使用它们，而不会发现漏洞来修改接口。接口应该已经正确实现，因此用户不需要进行修改。因此，用户只关注他们自己的代码。

[**Can't happen**](https://en.wikipedia.org/wiki/Assertion_(software_development))

Very often, code is modified and may introduce a possibility that an "impossible" case occurs. Impossible cases are therefore assumed to be highly unlikely instead.[[7]](https://en.wikipedia.org/wiki/Robustness_(computer_science)#cite_note-robust_programming-7) The developer thinks about how to handle the case that is highly unlikely, and implements the handling accordingly.：通常情况下，代码被修改，可能会引入“不可能”情况发生的可能性。因此，不可能的情况被认为是非常不可能的。开发人员考虑如何处理极不可能发生的情况，并相应地实现处理。

###### 参2：<https://www.1024sou.com/article/621612.html>

要写出健壮的前端代码，就要处理规范以外的输入，错误和异常。具体来说，有4点：

* 异常处理。
* 输入检查。
* 写法优化。
* 第三方库的选择。

###### 参3：<https://cloud.tencent.com/developer/article/1459341>



###### 参4：

<https://www.shymean.com/article/%E4%B8%80%E4%BA%9B%E6%8F%90%E9%AB%98%E4%BB%A3%E7%A0%81%E5%81%A5%E5%A3%AE%E6%80%A7%E7%9A%84%E6%96%B9%E6%B3%95>

（1） 单元测试

（2） 打日志

（3） 更安全地访问对象

（4） 记得异常处理

（5） 更稳定的第三方模块

（6） 本地配置文件

（7） code review

###### 参5：<https://blog.csdn.net/zzh920625/article/details/50761058>

1、在做前台页面过程中，对于ASP.NET验证控件的使用、正则表达式的使用，要融入到我们的日常编程习惯当中。传到后台的错误（类似于到B层逻辑判断进行不下去）才发现的错误，在前台一定要保证根本不让这些参数传入到后台，扼杀在摇篮中。

2、合理布局函数返回值，保证函数返回值一致

之前很多时候写函数往往很随性，返回值类型可以能代表函数执行成功或者失败的Bool型，也会有代表实际结果的Str或者Int等类型。这样的函数在外部调用时痛苦非常，因为在函数调用后处理时，处理不当就会出现typeError，所以在函数编写前，要思考后本函数的作用，同时确定返回值类型，在函数的所有涉及到返回结果时，给予一致类型的返回值，方便外部调用。

3、必要情况下的Try…Catch…处理

Try…Catch…出来处理异常是各种语言都有的模式。但到底在何处使用却有讲究。在没有抛异常的语句使用try语句，会降低性能，带来代码冗余，而在需要处理的语句未加异常处理，则会带来运行崩溃的可能。所以，要深刻的了解代码的语句，是否存在抛异常的可能，对可能抛异常的语言要加以处理。（这方面的介绍在下一篇实战博客中会进行讲解）

4、清理代码，去掉冗余代码

很多时候，我们的代码都是迭代开发的。往往会罗列一些无用的函数，引入一些无用的类库。这些内容貌似无意义，但却是代码中的隐患。可能在后续的类库更新或者函数变更中爆炸。所以，代码要保持清理，对于无用的引用和定义，要加以清除。

###### 参6：<https://blog.csdn.net/zzh920625/article/details/50761058>

**程序容错的几个原则**

如果从编码实践来看，我认为可以有以下几个原则可以作为指导。

具体内容如下：

* 对自我可控的程序输入可作选择性的容错（明确的不可能场景可以不处理容错）
* 对任何第三方的响应数据作为输入的，必须完成输入数据的容错处理（具体做法可以是两种：校验不通过的数据直接报错/直接过滤掉其中的错误数据）
* 对于容错的处理需要有一定程度的抽象，为后续逻辑变更降低影响面和成本做好基础设计

##### 4） 实例

<https://blog.csdn.net/bigpudding24/article/details/49069805>

写一段功能性的代码，可能需要一百行代码，但是写一段健壮的程序，至少需要300行代码。

例如：房贷计算器的代码，算法异常简单，十多行就完成了，但是，这段程序完全不具备健壮性，很简单，我的输入是不受限制的，这个程序要求从用户界面读取利率，年限，贷款额三个数据，一般同学的写法很简单，一句

|  |
| --- |
| doubleNum = Double.parseDouble(JOptionPane.showInputDialog(null,"请输入"+StrChars)) ; |

就万事OK了。

但是，真的有这么简单么，开玩笑，这么简单就好了，列举以下事例  
　　1，我输入了负数  
　　2，我的输入超出了double类型所能涵盖的范围  
　　3，我输入了标点符号  
　　4，我输入了中文  
　　5，我没输入  
　　6，我选择了取消或者点了右上角的关闭

这一切都是有可能发生的事件，而且超出了你程序的处理范围，这种事情本不该发生，但是程序使用时，一切输入都是有可能的，怎么办，你只能在程序中限制输入。

作为一个程序员，你如何让你的代码在执行的时候响应这些事件呢，我用了四十行代码编写了一个方法，用来限定我的输入只能为正实数，否则就报错；用户点击取消或者关闭按钮，则返回一个特殊数值，然后在主方法增加一个循环，在调用输入方法的时候检查返回值，如果为特殊值，就返回上层菜单或者关闭程序。

#### 《5》 逻辑严谨性

##### 1） 来源

Oppo



##### 2） 逻辑严谨性的定义

Oppo



##### 3） 已浏览资料

<https://www.h5w3.com/175352.html>

<https://www.cnblogs.com/ITyueguangyang/p/4301459.html>

<https://blog.csdn.net/miapecloud/article/details/99865141>

<https://blog.csdn.net/qq_38157171/article/details/108795535>

<https://www.zhihu.com/question/280098528>

<https://blog.csdn.net/maosidiaoxian/article/details/86291675>

<https://blog.csdn.net/james_1_2_1_1/article/details/73277441>

<https://time.geekbang.org/column/article/160985>

### 【3】 如何写出高质量代码

# 设计原则与思想

## 面向对象基础

### 面向对象（编程）4大特性

<https://time.geekbang.org/column/article/161114>

#### 《1》 封装

##### 实例

1. 迭代器模式

<https://time.geekbang.org/column/article/219290?cid=100039001>

#### 《2》 抽象【通常不算在面向对象特性中】

##### 是什么

<https://time.geekbang.org/column/article/161114>

封装主要讲的是如何隐藏信息、保护数据，而抽象讲的是如何隐藏方法的具体实现，让调用者只需要关心方法提供了哪些功能，并不需要知道这些功能是如何实现的。

##### 如何实现

https://time.geekbang.org/column/article/161114

###### 使用编程语言的语法机制

在面向对象编程中，我们常借助编程语言提供的接口类（比如 Java 中的 interface 关键字语法）或者抽象类（比如 Java 中的 abstract 关键字语法）这两种语法机制，来实现抽象这一特性。

注：这里我稍微说明一下，在专栏中，我们把编程语言提供的接口语法叫作“接口类”而不是“接口”。之所以这么做，是因为“接口”这个词太泛化，可以指好多概念，比如 API 接口等，所以，我们用“接口类”特指编程语言提供的接口语法。

###### 不使用编程语言的语法机制

实际上，抽象这个特性是非常容易实现的，并不需要非得依靠接口类或者抽象类这些特殊语法机制来支持。换句话说，并不是说一定要为实现类（PictureStorage）【见实例1】抽象出接口类（IPictureStorage），才叫作抽象。即便不编写 IPictureStorage 接口类，单纯的 PictureStorage 类本身就满足抽象特性。

之所以这么说，那是因为，类的方法是通过编程语言中的“函数”这一语法机制来实现的。通过函数包裹具体的实现逻辑，这本身就是一种抽象。调用者在使用函数的时候，并不需要去研究函数内部的实现逻辑，只需要通过函数的命名、注释或者文档，了解其提供了什么功能，就可以直接使用了。比如，我们在使用 C 语言的 malloc() 函数的时候，并不需要了解它的底层代码是怎么实现的。

##### Why通常不含在面向对象特性中

抽象这个概念是一个非常通用的设计思想，并不单单用在面向对象编程中，也可以用来指导架构设计等。而且这个特性也并不需要编程语言提供特殊的语法机制来支持，只需要提供“函数”这一非常基础的语法机制，就可以实现抽象特性、所以，它没有很强的“特异性”，有时候并不被看作面向对象编程的特性之一。

##### 意义

<https://time.geekbang.org/column/article/161114>

###### 1》 思考层面

实际上，如果上升一个思考层面的话，抽象及其前面讲到的封装都是人类处理复杂性的有效手段。在面对复杂系统的时候，人脑能承受的信息复杂程度是有限的，所以我们必须忽略掉一些非关键性的实现细节。而抽象作为一种只关注功能点不关注实现的设计思路，正好帮我们的大脑过滤掉许多非必要的信息。

###### 2》 设计思想层面

除此之外，抽象作为一个非常宽泛的设计思想，在代码设计中，起到非常重要的指导作用。很多设计原则都体现了抽象这种设计思想，比如基于接口而非实现编程、开闭原则（对扩展开放、对修改关闭）、代码解耦（降低代码的耦合性）等。我们在讲到后面的内容的时候，会具体来解释。

##### 使用抽象思维

<https://time.geekbang.org/column/article/161114>

换一个角度来考虑，我们在定义（或者叫命名）类的方法的时候，也要有抽象思维，不要在方法定义中，暴露太多的实现细节，以保证在某个时间点需要改变方法的实现逻辑的时候，不用去修改其定义。举个简单例子，比如 getAliyunPictureUrl() 就不是一个具有抽象思维的命名，因为某一天如果我们不再把图片存储在阿里云上，而是存储在私有云上，那这个命名也要随之被修改。相反，如果我们定义一个比较抽象的函数，比如叫作 getPictureUrl()，那即便内部存储方式修改了，我们也不需要修改命名。

##### 实例

1. 例1

<https://time.geekbang.org/column/article/161114>

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

1. 例2

<https://time.geekbang.org/column/article/165103>

评论区第3条

#### 《3》 继承

#### 《4》 多态

## 面向对象设计原则

### 单一职责原则：Single Responsibility Principle《SRP》

<https://time.geekbang.org/column/article/171771>

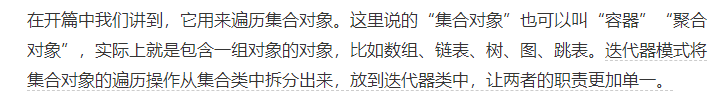
<https://time.geekbang.org/column/article/188882>

#### 是什么

#### 实例

##### 例1

https://time.geekbang.org/column/article/219290



##### 例2

迭代器模式

<https://time.geekbang.org/column/article/219290?cid=100039001>

##### 例3

<https://time.geekbang.org/column/article/176075>



### 开闭原则：Open(-)Closed Principle《OCP》

<https://time.geekbang.org/column/article/176075>

#### 前言

在上一节课中，我们学习了单一职责原则。今天，我们来学习 SOLID 中的第二个原则：开闭原则。我个人觉得，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. 之所以说这条原则难理解，那是因为，“怎样的代码改动才被定义为‘扩展’？怎样的代码改动才被定义为‘修改’？怎么才算满足或违反‘开闭原则’？修改代码就一定意味着违反‘开闭原则’吗？”等等这些问题，都比较难理解。
2. 之所以说这条原则难掌握，那是因为，“如何做到‘对扩展开放、修改关闭’？如何在项目中灵活地应用‘开闭原则’，以避免在追求扩展性的同时影响到代码的可读性？”等等这些问题，都比较难掌握。
3. 之所以说这条原则最有用，那是因为，扩展性是代码质量最重要的衡量标准之一。在 23 种经典设计模式中，大部分设计模式都是为了解决代码的扩展性问题而存在的，主要遵从的设计原则就是开闭原则。

所以说，今天的内容非常重要，希望你能集中精力，跟上我的思路，将开闭原则理解透彻，这样才能更好地理解后面章节的内容。话不多说，让我们正式开始今天的学习吧！

#### 《2》 是什么

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”。

这个描述比较简略，如果我们详细表述一下，那就是，添加一个新的功能应该是，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）。

#### 《3》 示例

为了让你更好地理解这个原则，我举一个例子来进一步解释一下。

##### 1） 原始需求

###### 1》 需求

这是一段 API 接口监控告警的代码。

其中，AlertRule 存储告警规则，可以自由设置。Notification 是告警通知类，支持邮件、短信、微信、手机等多种通知渠道。NotificationEmergencyLevel 表示通知的紧急程度，包括 SEVERE（严重）、URGENCY（紧急）、NORMAL（普通）、TRIVIAL（无关紧要），不同的紧急程度对应不同的发送渠道。

关于 API 接口监控告警这部分，更加详细的业务需求分析和设计，我们会在后面的设计模式模块再拿出来进一步讲解，这里你只要简单知道这些，就够我们今天用了。

###### 2》 code

|  |
| --- |
| public class Alert {  private AlertRule rule;  private Notification notification;   public Alert(AlertRule rule, Notification notification) {  this.rule = rule;  this.notification = notification;  }   public void check(String api, long requestCount, long errorCount, long durationOfSeconds) {  long tps = requestCount / durationOfSeconds;  if (tps > rule.getMatchedRule(api).getMaxTps()) {  notification.notify(NotificationEmergencyLevel.URGENCY, "...");  }  if (errorCount > rule.getMatchedRule(api).getMaxErrorCount()) {  notification.notify(NotificationEmergencyLevel.SEVERE, "...");  }  } } |

上面这段代码非常简单，业务逻辑主要集中在 check() 函数中。当接口的 TPS 超过某个预先设置的最大值时，以及当接口请求出错数大于某个最大允许值时，就会触发告警，通知接口的相关负责人或者团队。

##### 2） 新增需求

###### 1》 需求

现在，如果我们需要添加一个功能，当每秒钟接口超时请求个数，超过某个预先设置的最大阈值时，我们也要触发告警发送通知。这个时候，我们该如何改动代码呢？

###### 2》 思路1：基于“修改”

A） 思路

主要的改动有两处：第一处是修改 check() 函数的入参，添加一个新的统计数据 timeoutCount，表示超时接口请求数；第二处是在 check() 函数中添加新的告警逻辑。

B） code

具体的代码改动如下所示：

|  |
| --- |
| public class Alert {  // ...省略AlertRule/Notification属性和构造函数...   // 改动一：添加参数timeoutCount  public void check(String api, long requestCount, long errorCount, long timeoutCount, long durationOfSeconds) {  long tps = requestCount / durationOfSeconds;  if (tps > rule.getMatchedRule(api).getMaxTps()) {  notification.notify(NotificationEmergencyLevel.URGENCY, "...");  }  if (errorCount > rule.getMatchedRule(api).getMaxErrorCount()) {  notification.notify(NotificationEmergencyLevel.SEVERE, "...");  }  // 改动二：添加接口超时处理逻辑  long timeoutTps = timeoutCount / durationOfSeconds;  if (timeoutTps > rule.getMatchedRule(api).getMaxTimeoutTps()) {  notification.notify(NotificationEmergencyLevel.URGENCY, "...");  }  } } |

C） 问题

这样的代码修改实际上存在挺多问题的。一方面，我们对接口进行了修改，这就意味着调用这个接口的代码都要做相应的修改。另一方面，修改了 check() 函数，相应的单元测试都需要修改（关于单元测试的内容我们在重构那部分会详细介绍）。

###### 3》 思路2：基于“扩展”

上面的代码改动是基于“修改”的方式来实现新功能的。如果我们遵循开闭原则，也就是“对扩展开放、对修改关闭”。那如何通过“扩展”的方式，来实现同样的功能呢？

A） 重构Alert类提升扩展性

A》 重构Alert类

我们先重构一下之前的 Alert 代码，让它的扩展性更好一些。重构的内容主要包含两部分：

* 第一部分是将 check() 函数的多个入参封装成 ApiStatInfo 类；
* 第二部分是引入 handler 的概念，将 if 判断逻辑分散在各个 handler 中。

具体的代码实现如下所示：

|  |
| --- |
| public class Alert {  private List<AlertHandler> alertHandlers = new ArrayList<>();   public void addAlertHandler(AlertHandler alertHandler) {  this.alertHandlers.add(alertHandler);  }   public void check(ApiStatInfo apiStatInfo) {  for (AlertHandler handler : alertHandlers) {  handler.check(apiStatInfo);  }  } }  public class ApiStatInfo {//省略constructor/getter/setter方法  private String api;  private long requestCount;  private long errorCount;  private long durationOfSeconds; }  public abstract class AlertHandler {  protected AlertRule rule;  protected Notification notification;  public AlertHandler(AlertRule rule, Notification notification) {  this.rule = rule;  this.notification = notification;  }  public abstract void check(ApiStatInfo apiStatInfo); }  public class TpsAlertHandler extends AlertHandler {  public TpsAlertHandler(AlertRule rule, Notification notification) {  super(rule, notification);  }   @Override  public void check(ApiStatInfo apiStatInfo) {  long tps = apiStatInfo.getRequestCount()/ apiStatInfo.getDurationOfSeconds();  if (tps > rule.getMatchedRule(apiStatInfo.getApi()).getMaxTps()) {  notification.notify(NotificationEmergencyLevel.URGENCY, "...");  }  } }  public class ErrorAlertHandler extends AlertHandler {  public ErrorAlertHandler(AlertRule rule, Notification notification){  super(rule, notification);  }   @Override  public void check(ApiStatInfo apiStatInfo) {  if (apiStatInfo.getErrorCount() > rule.getMatchedRule(apiStatInfo.getApi()).getMaxErrorCount()) {  notification.notify(NotificationEmergencyLevel.SEVERE, "...");  }  } } |

B》 使用Alert类

上面的代码是对 Alert 的重构，我们再来看下，重构之后的 Alert 该如何使用呢？具体的使用代码我也写在这里了。

其中，ApplicationContext 是一个单例类，负责 Alert 的创建、组装（alertRule 和 notification 的依赖注入）、初始化（添加 handlers）工作。

|  |
| --- |
| public class ApplicationContext {  private AlertRule alertRule;  private Notification notification;  private Alert alert;   public void initializeBeans() {  alertRule = new AlertRule(/\*.省略参数.\*/); //省略一些初始化代码  notification = new Notification(/\*.省略参数.\*/); //省略一些初始化代码  alert = new Alert();  alert.addAlertHandler(new TpsAlertHandler(alertRule, notification));  alert.addAlertHandler(new ErrorAlertHandler(alertRule, notification));  }  public Alert getAlert() { return alert; }   // 饿汉式单例  private static final ApplicationContext *instance* = new ApplicationContext();  private ApplicationContext() {  initializeBeans();  }  public static ApplicationContext getInstance() {  return *instance*;  } }  public class Demo {  public static void main(String[] args) {  ApiStatInfo apiStatInfo = new ApiStatInfo();  // ...省略设置apiStatInfo数据值的代码  ApplicationContext.*getInstance*().getAlert().check(apiStatInfo);  } } |

B） 基于重构后的Alert类实现新增需求

A》 思路

现在，我们再来看下，基于重构之后的代码，如果再添加上面讲到的那个新功能，每秒钟接口超时请求个数超过某个最大阈值就告警，我们又该如何改动代码呢？主要的改动有下面四处：

* 第一处改动是：在 ApiStatInfo 类中添加新的属性 timeoutCount。
* 第二处改动是：添加新的 TimeoutAlertHander 类。
* 第三处改动是：在 ApplicationContext 类的 initializeBeans() 方法中，往 alert 对象中注册新的 timeoutAlertHandler。
* 第四处改动是：在使用 Alert 类的时候，需要给 check() 函数的入参 apiStatInfo 对象设置 timeoutCount 的值。

B》 code

改动之后的代码如下所示：

|  |
| --- |
| public class Alert { //代码未改动... }  public class ApiStatInfo { //省略constructor/getter/setter方法  private String api;  private long requestCount;  private long errorCount;  private long durationOfSeconds;  private long timeoutCount; // 改动一：添加新字段 }  public abstract class AlertHandler { //代码未改动... }  public class TpsAlertHandler extends AlertHandler {//代码未改动... }  public class ErrorAlertHandler extends AlertHandler {//代码未改动... }  // 改动二：添加新的handler public class TimeoutAlertHandler extends AlertHandler {//省略代码... }  public class ApplicationContext {  private AlertRule alertRule;  private Notification notification;  private Alert alert;   public void initializeBeans() {  alertRule = new AlertRule(/\*.省略参数.\*/); //省略一些初始化代码  notification = new Notification(/\*.省略参数.\*/); //省略一些初始化代码  alert = new Alert();  alert.addAlertHandler(new TpsAlertHandler(alertRule, notification));  alert.addAlertHandler(new ErrorAlertHandler(alertRule, notification));  // 改动三：注册handler  alert.addAlertHandler(new TimeoutAlertHandler(alertRule, notification));  }  //...省略其他未改动代码... }  public class Demo {  public static void main(String[] args) {  ApiStatInfo apiStatInfo = new ApiStatInfo();  // ...省略apiStatInfo的set字段代码  apiStatInfo.setTimeoutCount(289); // 改动四：设置tiemoutCount值  ApplicationContext.getInstance().getAlert().check(apiStatInfo);  } } |

C》 好处

重构之后的代码更加灵活和易扩展。如果我们要想添加新的告警逻辑，只需要基于扩展的方式创建新的 handler 类即可，不需要改动原来的 check() 函数的逻辑。而且，我们只需要为新的 handler 类添加单元测试，老的单元测试都不会失败，也不用修改。

#### 《4》 修改代码就意味着违背开闭原则吗？

看了上面重构之后的代码，你可能还会有疑问：在添加新的告警逻辑的时候，尽管改动二（添加新的 handler 类）是基于扩展而非修改的方式来完成的，但改动一、三、四貌似不是基于扩展而是基于修改的方式来完成的，那改动一、三、四不就违背了开闭原则吗？

##### 1） 改动一

**我们先来分析一下改动一：往 ApiStatInfo 类中添加新的属性 timeoutCount。**

实际上，我们不仅往 ApiStatInfo 类中添加了属性，还添加了对应的 getter/setter 方法。那这个问题就转化为：给类中添加新的属性和方法，算作“修改”还是“扩展”？

我们再一块回忆一下开闭原则的定义：软件实体（模块、类、方法等）应该“对扩展开放、对修改关闭”。从定义中，我们可以看出，开闭原则可以应用在不同粒度的代码中，可以是模块，也可以类，还可以是方法（及属性）。同样一个代码改动，在粗代码粒度下，被认定为“修改”，在细代码粒度下，又可以被认定为“扩展”。比如，改动一，添加属性和方法相当于修改类，在类这个层面，这个代码改动可以被认定为“修改”；但这个代码改动并没有修改已有的属性和方法，在方法（及属性）这一层面，它又可以被认定为“扩展”。

实际上，我们也没必要纠结某个代码改动是“修改”还是“扩展”，更没必要太纠结它是否违反“开闭原则”。我们回到这条原则的设计初衷：只要它没有破坏原有的代码的正常运行，没有破坏原有的单元测试，我们就可以说，这是一个合格的代码改动。

##### 2） 改动三和改动四

**我们再来分析一下改动三和改动四：在 ApplicationContext 类的 initializeBeans() 方法中，往 alert 对象中注册新的 timeoutAlertHandler；在使用 Alert 类的时候，需要给 check() 函数的入参 apiStatInfo 对象设置 timeoutCount 的值。**

这两处改动都是在方法内部进行的，不管从哪个层面（模块、类、方法）来讲，都不能算是“扩展”，而是地地道道的“修改”。不过，有些修改是在所难免的，是可以被接受的。为什么这么说呢？我来解释一下。

在重构之后的 Alert 代码中，我们的核心逻辑集中在 Alert 类及其各个 handler 中，当我们在添加新的告警逻辑的时候，Alert 类完全不需要修改，而只需要扩展一个新 handler 类。如果我们把 Alert 类及各个 handler 类合起来看作一个“模块”，那模块本身在添加新的功能的时候，完全满足开闭原则。而且，我们要认识到，添加一个新功能，不可能任何模块、类、方法的代码都不“修改”，这个是做不到的。类需要创建、组装、并且做一些初始化操作，才能构建成可运行的的程序，这部分代码的修改是在所难免的。我们要做的是尽量让修改操作更集中、更少、更上层，尽量让最核心、最复杂的那部分逻辑代码满足开闭原则。

#### 《x》 如何做到“对扩展开放、修改关闭”？

##### 1） 关于知识和经验的积累

在刚刚的例子中，我们通过引入一组 handler 的方式来实现支持开闭原则。如果你没有太多复杂代码的设计和开发经验，你可能会有这样的疑问：这样的代码设计思路我怎么想不到呢？你是怎么想到的呢？

先给你个结论，之所以我能想到，靠的就是理论知识和实战经验，这些需要你慢慢学习和积累。

##### 2） 关于“开闭原则”的指导思想和方法论

对于如何做到“对扩展开放、修改关闭”，我们也有一些指导思想和具体的方法论，我们一块来看一下。

###### 1》 开闭原则和代码扩展性

实际上，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“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。所以，问如何才能做到“对扩展开放、对修改关闭”，也就粗略地等同于在问，如何才能写出扩展性好的代码。

###### 2》 指导思想

A） 概述

**在讲具体的方法论之前，我们先来看一些更加偏向顶层【个人理解是类似计算机系统的分层思路，顶层是应用调用方，底层是操作系统服务提供方】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。**

B） 详解

* 在写代码的时候后，我们要多花点时间往前多思考一下，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”。
* 还有，在识别出代码可变部分和不可变部分之后，我们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。

###### 3》 方法论

A） 概述

**刚刚我们讲了实现开闭原则的一些偏向顶层的指导思想，现在我们再来看下，支持开闭原则的一些更加具体的方法论。**

B） 详解

我们前面讲到，代码的扩展性是代码质量评判的最重要的标准之一。实际上，我们整个专栏的大部分知识点都是围绕扩展性问题来讲解的。专栏中讲到的很多设计原则、设计思想、设计模式，都是以提高代码的扩展性为最终目的的。特别是 23 种经典设计模式，大部分都是为了解决代码的扩展性问题而总结出来的，都是以开闭原则为指导原则的。

在众多的设计原则、思想、模式中，最常用来提高代码扩展性的方法有：多态、依赖注入、基于接口而非实现编程，以及大部分的设计模式（比如，装饰、策略、模板、职责链、状态等）。设计模式这一部分内容比较多，后面课程中我们能会详细讲到，这里就不展开了。今天我重点讲一下，如何利用多态、依赖注入、基于接口而非实现编程，来实现“对扩展开放、对修改关闭”。

实际上，多态、依赖注入、基于接口而非实现编程，以及前面提到的抽象意识，说的都是同一种设计思路，只是从不同的角度、不同的层面来阐述而已。这也体现了“很多设计原则、思想、模式都是相通的”这一思想。

C） 示例

接下来，我就通过一个例子来解释一下，如何利用这几个设计思想或原则来实现“对扩展开放、对修改关闭”。注意，依赖注入后面会讲到，如果你对这块不了解，可以暂时先忽略这个概念，只关注多态、基于接口而非实现编程以及抽象意识。

比如，我们代码中通过 Kafka 来发送异步消息。对于这样一个功能的开发，我们要学会将其抽象成一组跟具体消息队列（Kafka）无关的异步消息接口。所有上层系统都依赖这组抽象的接口编程，并且通过依赖注入的方式来调用。当我们要替换新的消息队列的时候，比如将 Kafka 替换成 RocketMQ，可以很方便地拔掉老的消息队列实现，插入新的消息队列实现。具体代码如下所示：

|  |
| --- |
| // 这一部分体现了抽象意识  public interface MessageQueue { //... }  public class KafkaMessageQueue implements MessageQueue { //... }  public class RocketMQMessageQueue implements MessageQueue {//...}  public interface MessageFromatter { //... }  public class JsonMessageFromatter implements MessageFromatter {//...}  public class ProtoBufMessageFromatter implements MessageFromatter {//...}  public class Demo {  private MessageQueue msgQueue; // 基于接口而非实现编程  public Demo(MessageQueue msgQueue) { // 依赖注入  this.msgQueue = msgQueue;  }    // msgFormatter：多态、依赖注入  public void sendNotification(Notification notification, MessageFormatter msgFormatter) {  //...  }  } |

###### 4》 补充说明

对于如何写出扩展性好的代码、如何实现“对扩展开放、对修改关闭”这个问题，我今天只是比较笼统地总结了一下，详细的知识我们在后面的章节中慢慢学习。

#### 《x》 如何在项目中灵活应用开闭原则

##### 1） 扩展点的判断

###### 1》 识别扩展点的方法

前面我们提到，写出支持“对扩展开放、对修改关闭”的代码的关键是预留扩展点。那问题是如何才能识别出所有可能的扩展点呢？

* 如果你开发的是一个业务导向的系统，比如金融系统、电商系统、物流系统等，要想识别出尽可能多的扩展点，就要对业务有足够的了解，能够知道当下以及未来可能要支持的业务需求。
* 如果你开发的是跟业务无关的、通用的、偏底层的系统，比如，框架、组件、类库，你需要了解“它们会被如何使用？今后你打算添加哪些功能？使用者未来会有哪些更多的功能需求？”等问题。

###### 2》 识别扩展点的注意项

不过，有一句话说得好，“唯一不变的只有变化本身”。即便我们对业务、对系统有足够的了解，那也不可能识别出所有的扩展点，即便你能识别出所有的扩展点，为这些地方都预留扩展点，这样做的成本也是不可接受的。我们没必要为一些遥远的、不一定发生的需求去提前买单，做过度设计。

最合理的做法是，对于一些比较确定的、短期内可能就会扩展，或者需求改动对代码结构影响比较大的情况，或者实现成本不高的扩展点，在编写代码的时候，我们就可以事先做些扩展性设计。但对于一些不确定未来是否要支持的需求，或者实现起来比较复杂的扩展点，我们可以等到有需求驱动的时候，再通过重构代码的方式来支持扩展的需求。

##### 2） 扩展性和可读性的冲突问题

而且，开闭原则也并不是免费的。有些情况下，代码的扩展性会跟可读性相冲突。比如，我们之前举的 Alert 告警的例子。为了更好地支持扩展性，我们对代码进行了重构，重构之后的代码要比之前的代码复杂很多，理解起来也更加有难度。很多时候，我们都需要在扩展性和可读性之间做权衡。在某些场景下，代码的扩展性很重要，我们就可以适当地牺牲一些代码的可读性；在另一些场景下，代码的可读性更加重要，那我们就适当地牺牲一些代码的可扩展性。

在我们之前举的 Alert 告警的例子中，如果告警规则并不是很多、也不复杂，那 check() 函数中的 if 语句就不会很多，代码逻辑也不复杂，代码行数也不多，那最初的第一种代码实现思路简单易读，就是比较合理的选择。相反，如果告警规则很多、很复杂，check() 函数的 if 语句、代码逻辑就会很多、很复杂，相应的代码行数也会很多，可读性、可维护性就会变差，那重构之后的第二种代码实现思路就是更加合理的选择了。总之，这里没有一个放之四海而皆准的参考标准，全凭实际的应用场景来决定。

#### 《x》 总结

今天的内容到此就讲完了。我们一块来总结回顾一下，你需要掌握的重点内容。

##### 1） 如何理解“对扩展开放、对修改关闭”？

添加一个新的功能，应该是通过在已有代码基础上扩展代码（新增模块、类、方法、属性等），而非修改已有代码（修改模块、类、方法、属性等）的方式来完成。关于定义，我们有两点要注意。第一点是，开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发。第二点是，同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”。

##### 2） 如何做到“对扩展开放、修改关闭”？

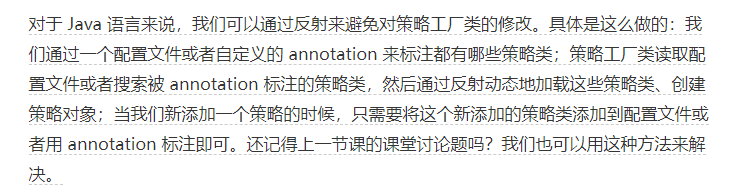
我们要时刻具备扩展意识、抽象意识、封装意识。在写代码的时候，我们要多花点时间思考一下，这段代码未来可能有哪些需求变更，如何设计代码结构，事先留好扩展点，以便在未来需求变更的时候，在不改动代码整体结构、做到最小代码改动的情况下，将新的代码灵活地插入到扩展点上。

很多设计原则、设计思想、设计模式，都是以提高代码的扩展性为最终目的的。特别是 23 种经典设计模式，大部分都是为了解决代码的扩展性问题而总结出来的，都是以开闭原则为指导原则的。最常用来提高代码扩展性的方法有：多态、依赖注入、基于接口而非实现编程，以及大部分的设计模式（比如，装饰、策略、模板、职责链、状态）。

#### 《x》 实例

##### 1） 例1

<https://time.geekbang.org/column/article/215132>



##### 2） 例2

https://time.geekbang.org/column/article/217395



##### 3） 例3

迭代器模式

<https://time.geekbang.org/column/article/219290?cid=100039001>

#### 《x》 课堂讨论

学习设计原则，要多问个为什么。不能把设计原则当真理，而是要理解设计原则背后的思想。搞清楚这个，比单纯理解原则讲的是啥，更能让你灵活应用原则。所以，今天课堂讨论的话题是，为什么我们要“对扩展开放、对修改关闭”？

### 里氏替换原则：Liskov Substitution Principle《LSP》

### 接口隔离原则：Interface Segregation Principle《ISP》

### 依赖反转原则: Dependency Inversion Principle《DIP》

<https://time.geekbang.org/column/article/177444>

#### 《1》 控制反转：Inversion Of Control<IOC>

实例：dagger（刘恒8.9ojt）

#### 《2》 依赖注入：Dependency Injection<DI>

#### 《3》 依赖反转/倒置原则：Dependency Inversion Principle<DIP>

### KISS原则和YAGNI原则

### DRY原则：Don’t Repeat Yourself

<https://time.geekbang.org/column/article/179607>

#### DRY是什么

#### 三种典型代码重复情况

#### 代码复用性

##### 代码复用性，代码复用和DRY原则

##### 如何提高代码复用性

### 迪米特法则：Law of Demeter《LOD》

### “基于接口而非实现编程”原则

<https://time.geekbang.org/column/article/165114>

#### 课堂讨论

原本

|  |
| --- |
| interface ImageStore {  String upload(Image image, String bucketName);  Image download(String url); }  class AliyunImageStore implements ImageStore {  //...省略属性、构造函数等...   public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  String accessToken = generateAccessToken();  //...上传图片到阿里云...  //...返回图片在阿里云上的地址(url)...  }   public Image download(String url) {  String accessToken = generateAccessToken();  //...从阿里云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  }   private String generateAccessToken() {  // ...根据accesskey/secrectkey等生成access token  } }  // 上传下载流程改变：私有云不需要支持access token class PrivateImageStore implements ImageStore {  public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  //...上传图片到私有云...  //...返回图片的url...  }   public Image download(String url) {  //...从私有云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  } }  // ImageStore的使用举例 public class ImageProcessingJob {  private static final String *BUCKET\_NAME* = "ai\_images\_bucket";  //...省略其他无关代码...   public void process() {  Image image = ...;//处理图片，并封装为Image对象  ImageStore imageStore = new PrivateImageStore(...);  imagestore.upload(image, *BUCKET\_NAME*);  } } |

##### 工厂模式

###### 简单工厂

* 1. 代码：写法1

*SimpleImageStoreFactory*

|  |
| --- |
| public class SimpleImageStoreFactory {   public ImageStore createImageStore(String type) {  ImageStore imageStore = null;   if (type.equals("Private")) {  imageStore = new PrivateImageStore();  }...   return imageStore;  } } |

*ImageProcessingJob1*

|  |
| --- |
| interface ImageStore {  String upload(Image image, String bucketName);  Image download(String url); }  class AliyunImageStore implements ImageStore {  //...省略属性、构造函数等...   public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  String accessToken = generateAccessToken();  //...上传图片到阿里云...  //...返回图片在阿里云上的地址(url)...  }   public Image download(String url) {  String accessToken = generateAccessToken();  //...从阿里云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  }   private String generateAccessToken() {  // ...根据accesskey/secrectkey等生成access token  } }  // 上传下载流程改变：私有云不需要支持access token class PrivateImageStore implements ImageStore {  public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  //...上传图片到私有云...  //...返回图片的url...  }   public Image download(String url) {  //...从私有云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  } }  // ImageStore的使用举例 class ImageProcessingJob1 {  private static final String *BUCKET\_NAME* = "ai\_images\_bucket";  //...省略其他无关代码...   SimpleImageStoreFactory factory;   public ImageProcessingJob1(SimpleImageStoreFactory factory) {  this.factory = factory;  }   public void process(String type) {  Image image = ...;//处理图片，并封装为Image对象  ImageStore imageStore;  imageStore = factory.createImageStore(type);  imageStore.upload(image, *BUCKET\_NAME*);  } } |

* 1. 代码：写法2

图形用户界面, 文本, 应用程序

描述已自动生成

* 1. 缺陷

但也没解决需要修改创建ImageStore对象代码问题（对修

改关闭）。

###### 简单工厂+配置文件

参考：

<https://time.geekbang.org/column/article/165114> 评论（辣么大）

+<https://github.com/gdhucoder/Algorithms4/tree/master/geekbang/designpattern/u009> v2

* 1. 代码

*SimpleImageStoreFactory*

|  |
| --- |
| public class SimpleImageStoreFactory2 {   private static final String *PRIVATE* = "private";  private static final String *ALIYUN* = "aliyun";   private ImageStoreFactory() {}   public static ImageStore createImageStore(String type) {  switch (type) {  case *PRIVATE*:  return new PrivateImageStore();  case *ALIYUN*:  return new AliyunImageStore();  default:  throw new IllegalArgumentException("not implemented yet");  }  } } |

*ImageProcessingJob2*

|  |
| --- |
| interface ImageStore {  String upload(Image image, String bucketName);  Image download(String url); }  class AliyunImageStore implements ImageStore {  //...省略属性、构造函数等...   public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  String accessToken = generateAccessToken();  //...上传图片到阿里云...  //...返回图片在阿里云上的地址(url)...  }   public Image download(String url) {  String accessToken = generateAccessToken();  //...从阿里云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  }   private String generateAccessToken() {  // ...根据accesskey/secrectkey等生成access token  } }  // 上传下载流程改变：私有云不需要支持access token class PrivateImageStore implements ImageStore {  public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  //...上传图片到私有云...  //...返回图片的url...  }   public Image download(String url) {  //...从私有云下载图片...  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  } }  // ImageStore的使用举例 class ImageProcessingJob2 {   private static final String *BUCKET\_NAME* = "ai\_images\_bucket";  private static final String *STORE\_TYPE* = "STORE\_TYPE";  private static final String *PROP\_PATH* = "geekbang/designpattern/u009/v2/config.properties";   //...省略其他无关代码...   public void process() {  Image image = ...;//处理图片，并封装为Image对象  String storeType = FileUtils.*load*(*PROP\_PATH*)  .getProperty(*STORE\_TYPE*);  ImageStore imageStore = SimpleImageStoreFactory2.*createImageStore*(storeType);  imageStore.upload(image, *BUCKET\_NAME*);  } }  */\*\*  \* 调用process方法  \*/* class ImageProcessingTest {   public static void main(String[] args) {  ImageProcessingJob2 job = new ImageProcessingJob2();  job.process();  } } |

*FileUtils*

|  |
| --- |
| public class FileUtils {   private FileUtils() {}   public static Properties load(String path) {   Properties properties = null;  try (InputStream in = new FileInputStream(Paths.*get*(path).toFile());) {  properties = new Properties();  properties.load(in);  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  return properties;  } } |

*config.properties*

|  |
| --- |
| STORE\_TYPE=private |

* 1. 缺陷

同1》，再新增另一种存储手段时，需要修改工厂类（添加新的类）。修改工厂类，违反了开放-封闭原则。

###### 简单工厂+配置文件+反射

参考：

<https://time.geekbang.org/column/article/165114> 评论（辣么大）

https://github.com/gdhucoder/Algorithms4/tree/master/geekbang/designpattern/u009/v3

* 1. 代码

图形用户界面, 文本, 应用程序

描述已自动生成

*ImageProcessingJob*

|  |
| --- |
| public class ImageProcessingJob {   private static final String *BUCKET\_NAME* = "ai\_images\_bucket";  private static final String *STORE\_CLASS* = "STORE\_CLASS";  private static final String *PROP\_PATH* = "src/reflecttest/config.properties";   public void process() {  Image image = null;  try {  String storeClass = FileUtils.*load*(*PROP\_PATH*)  .getProperty(*STORE\_CLASS*);  ImageStore imageStore = (ImageStore) Class.*forName*(storeClass).newInstance();  System.*out*.println(imageStore.upload(image, *BUCKET\_NAME*));  } catch (Exception e) {  e.printStackTrace();  }  }   public static void main(String[] args) {  ImageProcessingJob job = new ImageProcessingJob();  // PrivateImageStore upload  job.process();  } } |

AliyunImageStore+PrivateImageStore

|  |
| --- |
| class AliyunImageStore implements ImageStore {  //...省略属性、构造函数等...   public String upload(Image image, String bucketName) { // createBucketIfNotExisting(bucketName); // String accessToken = generateAccessToken();  //...上传图片到阿里云...  //...返回图片在阿里云上的地址(url)...  return "PrivateImageStore";  }   public Image download(String url) { // String accessToken = generateAccessToken();  //...从阿里云下载图片...  return null;  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  }  // private String generateAccessToken() { // // ...根据accesskey/secrectkey等生成access token // } }  // 上传下载流程改变：私有云不需要支持access token class PrivateImageStore implements ImageStore {    public String upload(Image image, String bucketName) {  createBucketIfNotExisting(bucketName);  //...上传图片到私有云...  //...返回图片的url...  return "PrivateImageStore";  }   public Image download(String url) {  //...从私有云下载图片...  return null;  }   private void createBucketIfNotExisting(String bucketName) {  // ...创建bucket...  // ...失败会抛出异常..  } } |

*ImageStore*

|  |
| --- |
| public interface ImageStore {   String upload(Image image, String bucketName);   Image download(String url); } |

*config.properties*

|  |
| --- |
| STORE\_CLASS=reflecttest.PrivateImageStore |

*FileUtils*

|  |
| --- |
| public class FileUtils {   private FileUtils() {}   public static Properties load(String path) {   Properties properties = null;  try (InputStream in = new FileInputStream(Paths.*get*(path).toFile())) {  properties = new Properties();  properties.load(in);  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  return properties;  } } |

结果：

文本

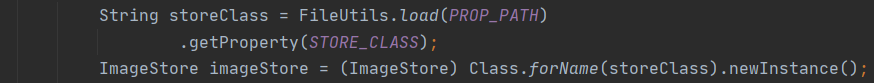
描述已自动生成

* 1. 缺陷

使用反射，在大量创建对象时会有性能损失。

* 1. 待补知识点

1. 反射



1. try-with-resources statement

文本

描述已自动生成

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

##### 依赖注入

###### 思路

文本

描述已自动生成

或

文本, 信件

描述已自动生成

或



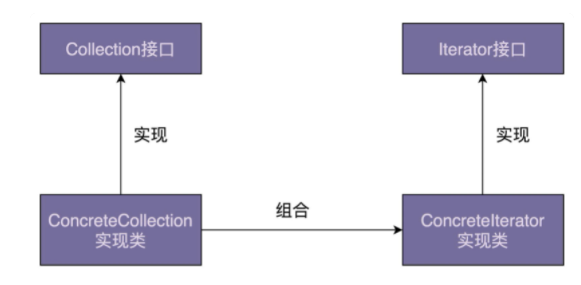
##### 策略模式

#### 实例

##### 例1

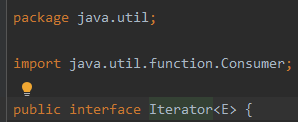
<https://time.geekbang.org/column/article/219290>

迭代器是用来遍历容器的，所以，一个完整的迭代器模式一般会涉及容器和容器迭代器两部分内容。为了达到基于接口而非实现编程的目的，容器又包含容器接口、容器实现类，迭代器又包含迭代器接口、迭代器实现类。对于迭代器模式，我画了一张简单的类图，你可以看一看，先有个大致的印象。

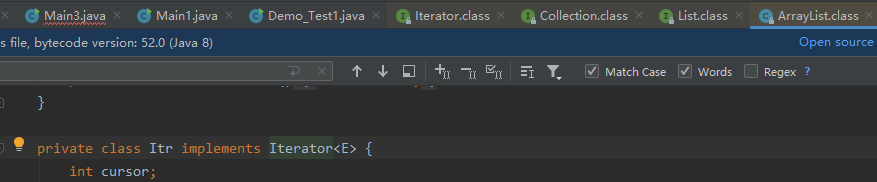


Java代码：

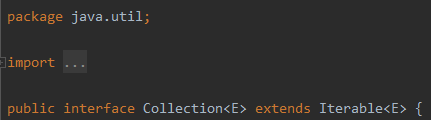
*Iterator接口：*



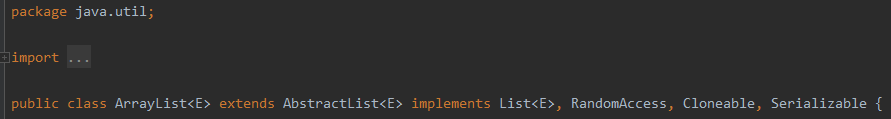
*Iterator实现：*



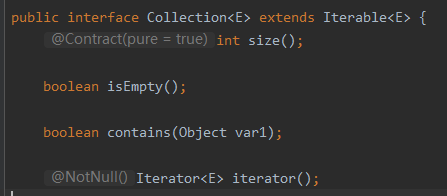
*Collection接口：*

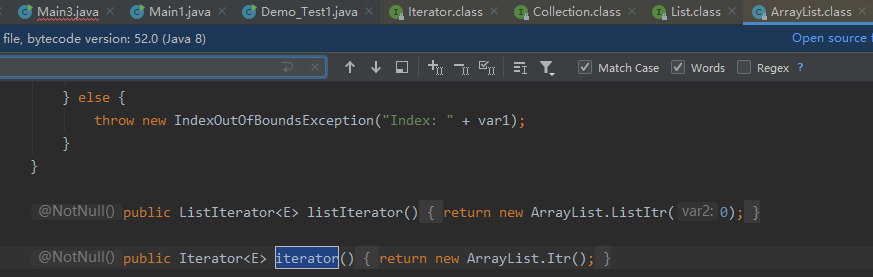


*Collection实现：*



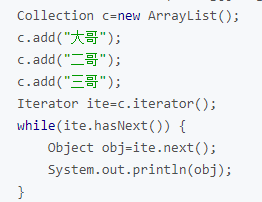
ConcreteCollection组合ConcreteIterator





使用：

<https://blog.csdn.net/zsp151296/article/details/76714137>



### “多用组合少用继承”原则

<https://time.geekbang.org/column/article/169593>

## 规范与重构

## 总结课

# 设计模式与范式

## 面向对象设计模式分类（按类型和是否常用）

<https://time.geekbang.org/column/article/160991>

### 创建型

#### 是什么

<https://time.geekbang.org/column/article/201823>

创建型模式主要解决“对象的创建”问题，封装复杂的创建过程，解耦对象的创建代码和使用代码。

#### 《2》 有哪些

* 常用：单例模式、工厂模式、建造者模式。
* 不常用：原型模式。

#### 《3》 详解

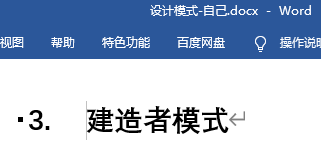
##### 1） 单例模式

<https://time.geekbang.org/column/article/194035>

###### 1》 实例

<https://time.geekbang.org/column/article/218375> 实现三

##### 2） Builder/建造者/构建者/生成器 模式

见

### 结构型

#### 是什么

<https://time.geekbang.org/column/article/201823>

<https://time.geekbang.org/column/article/210170>

结构型模式主要总结了一些类或对象组合在一起的经典结构，这些经典的结构可以解决特定应用场景的问题。

结构型设计模式主要解决“类或对象的组合或组装”问题。

#### 《2》 有哪些

* 常用：代理模式、桥接模式、装饰者/装饰器模式、适配器模式。
* 不常用：门面/外观模式、组合模式、享元模式。

#### 《3》 详解

##### x） 代理模式

见 “设计模式-自己.docx”

##### x） 装饰器模式

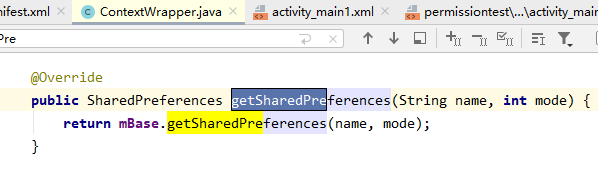
###### 1》 实例

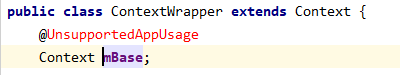
1） java io

<https://time.geekbang.org/column/article/204845>

inputstream，outputstream

2） android context:上下文 体系





Context创建

3） Resource创建

##### x） 适配器模式

见 “设计模式-自己.docx”

##### x） 门面模式

###### 1》 是什么

参1：

<https://ke.qq.com/webcourse/347420/103755197#taid=12286557109439772&vid=3701925924185390006> 1:37:46

你去买东西，你只需要跟购物商店，那个门面打交道即可，至于这个商店在哪进货，是否改变货源什么，我们都不用管。黑盒作用。

###### x》 实例

例1：

Arouter

例2：

DecorView的内部结构：“AndroidView-待分类.docx”-搜“门面”

###### x》 门面和代理的区别

<https://ke.qq.com/webcourse/347420/103755197#taid=12286557109439772&vid=3701925924185390006> 1:47:25

##### x） 享元模式

###### 1》 实例

A） 例1

Android的Message.obtain()

### 行为型

#### 是什么

<https://time.geekbang.org/column/article/210170>

行为型设计模式主要解决的就是“类或对象之间的交互”问题。

#### 《2》 有哪些

* 常用：观察者模式、模板模式、策略模式、职责链/责任链模式、迭代器模式、状态模式。
* 不常用：访问者模式、备忘录模式、命令模式、解释器模式、中介模式。

#### 《3》 详解

##### x） 模板模式

###### 1》 实例

Jetpack-Room

##### x） 职责链模式

<https://time.geekbang.org/column/article/216278>

###### 1》 是什么

参1：<https://time.geekbang.org/column/article/216278>

**定义：**

职责链模式的英文翻译是 Chain Of Responsibility Design Pattern。

在 GoF 的《设计模式》中，它是这么定义的：

|  |
| --- |
| Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. |

翻译成中文就是：

|  |
| --- |
| 将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，直到链上的某个接收对象能够处理它为止。 |

这么说比较抽象，我用更加容易理解的话来进一步解读一下：

|  |
| --- |
| 在职责链模式中，多个处理器依次处理同一个请求。一个请求先经过 A 处理器处理，然后再把请求传递给 B 处理器，B 处理器处理完后再传递给 C 处理器，以此类推，形成一个链条。链条上的每个处理器各自承担各自的处理职责，所以叫作职责链模式。 |

在 GoF 的定义中，一旦某个处理器能处理这个请求，就不会继续将请求传递给后续的处理器了。当然，在实际的开发中，也存在对这个模式的变体，那就是请求不会中途终止传递，而是会被所有的处理器都处理一遍。

**实现方式：**

职责链模式有多种实现方式。其中有两种常用的实现：一种是使用链表来存储处理器，另一种是使用数组来存储处理器，后面一种实现方式更加简单。

###### x》 实例

1） Okhttp的拦截器；

2） <https://time.geekbang.org/column/article/176075> 

3） android事件分发机制

##### x） 迭代器模式

###### 1》 实例

* + 1. List（ArrayList）+Iterator（ArrayList.Itr） <https://blog.csdn.net/zhujiangtaotaise/article/details/50515939>

<https://www.cnblogs.com/amboyna/archive/2007/09/25/904804.html>

可借鉴，但细节不对（看源码）

##### x） 策略模式

###### 1》 实例

Java带Comparator 参数的sort（）

# 一些作者归纳的

## 设计原则和设计模式 的关系

<https://time.geekbang.org/column/article/215132>

设计原则和思想其实比设计模式更加普适和重要，掌握了代码的设计原则和思想，我们甚至可以自己创造出来新的设计模式。

## 考验代码设计好坏的标准

1. 标准1

<https://time.geekbang.org/column/article/165114>

在软件开发中，**最大的挑战之一就是需求的不断变化（软件开发中唯一不变的就是变化）**，这也是考验代码设计好坏的一个标准。

越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计，不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能够在不破坏原有代码设计的情况下灵活应对。

而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。

## 如何写好接口

参考：<https://time.geekbang.org/column/article/165114>

我们在做软件开发的时候，一定要有抽象意识/思维、封装意识、接口意识。在定义接口的时候，不要暴露任何实现细节。接口的定义只表明做什么，而不是怎么做。而且，在设计接口的时候，我们要多思考一下，这样的接口设计是否足够通用，是否能够做到在替换具体的接口实现的时候，不需要任何接口定义的改动。

## 设计模式存在变体

<https://time.geekbang.org/column/article/217395>

从源码中，我们还可以发现，尽管上一节课中我们有给出职责链模式的经典代码实现，但在实际的开发中，我们还是要具体问题具体对待，代码实现会根据不同的需求有所变化。实际上，这一点对于所有的设计模式都适用。

## 提高代码扩展性的重要性

<https://time.geekbang.org/column/article/176075>

代码的扩展性是代码质量评判的最重要的标准之一。实际上，我们整个专栏的大部分知识点都是围绕扩展性问题来讲解的。专栏中讲到的很多设计原则、设计思想、设计模式，都是以提高代码的扩展性为最终目的的。特别是 23 种经典设计模式，大部分都是为了解决代码的扩展性问题而总结出来的，都是以开闭原则为指导原则的。

## 最常用来提高代码扩展性的方法

在众多的设计原则、思想、模式中，最常用来提高代码扩展性的方法有：多态、依赖注入、基于接口而非实现编程，以及大部分的设计模式（比如，装饰、策略、模板、职责链、状态等）。

## 设计思路的相通性

实际上，多态、依赖注入、基于接口而非实现编程，以及前面提到的抽象意识，说的都是同一种设计思路，只是从不同的角度、不同的层面来阐述而已。这也体现了“很多设计原则、思想、模式都是相通的”这一思想。

## 如何学习设计原则

学习设计原则，要多问个为什么。不能把设计原则当真理，而是要理解设计原则背后的思想。搞清楚这个，比单纯理解原则讲的是啥，更能让你灵活应用原则。

## 设计原则的主要目的

迭代器模式主要作用是解耦容器代码和遍历代码，这也印证了我们前面多次讲过的应用设计模式的主要目的是解耦。

## 编程中的抽象意识/思维

### 是什么

<https://mp.weixin.qq.com/s/x4mXHHM8p9cJlc8pXfe_Ug>

<https://time.geekbang.org/column/article/161114>

### 实例

<https://time.geekbang.org/column/article/176075>