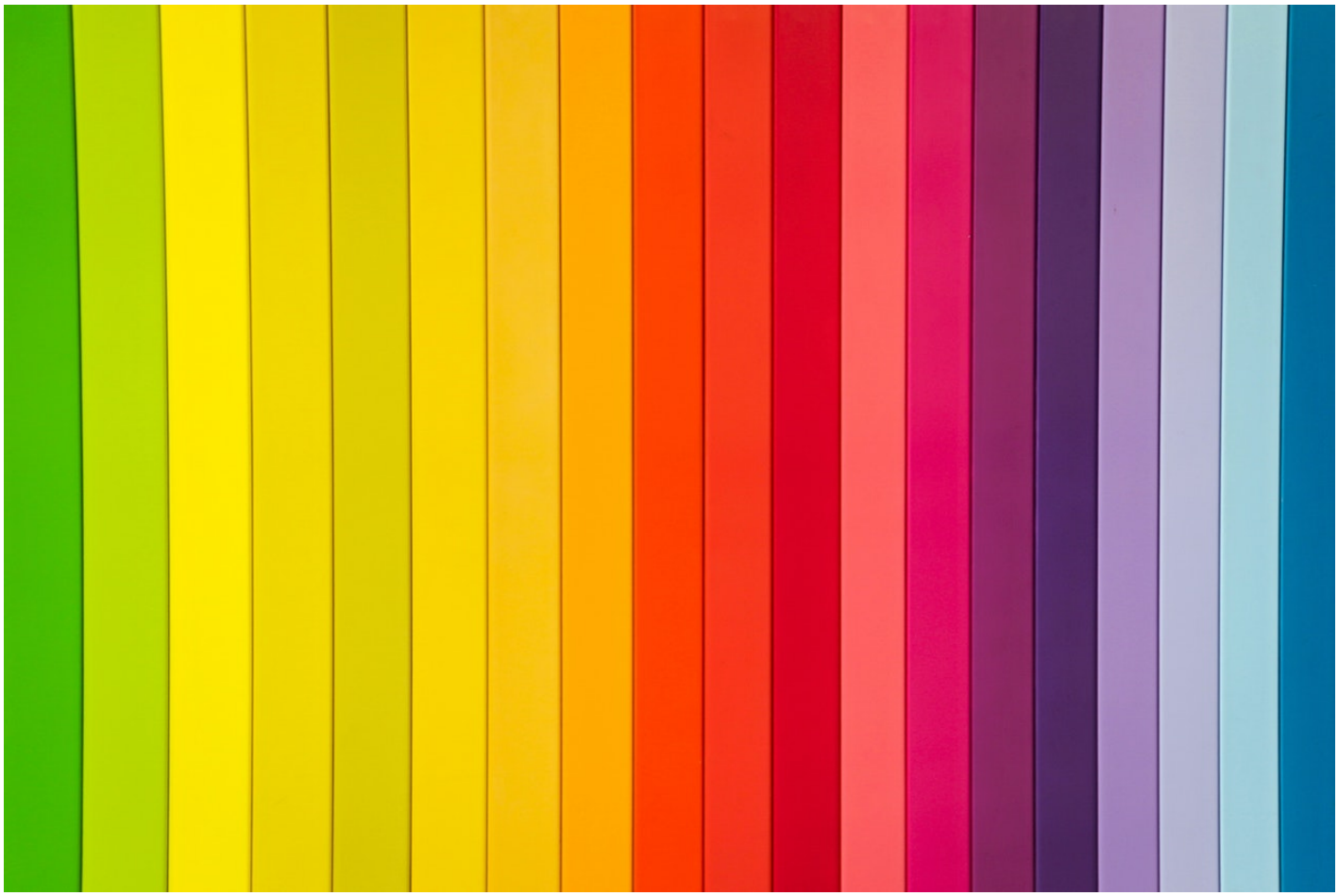
16讲理论二：如何做到“对扩展开放、修改关闭”扩展和修改各指什么



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。

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。

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。

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！

如何理解“对扩展开放、修改关闭”？

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）。

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。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。

```
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。

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？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。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, "...");
    }
  }
}
```

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）。

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？

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, "...");
    }
```

```
  }
}
```

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() {
    instance.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);
  }
}
```

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。

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);
}
```

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。

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?

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（及其属性）。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（及其属性）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。

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。

如何做到“对扩展开放、修改关闭”?

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。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。

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。

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。

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。

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。

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。

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 MessageFormatter { //... }
public class JsonMessageFormatter implements MessageFormatter {//...}
public class MessageFormatter implements MessageFormatter {//...}

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) {
    //...
  }
}
```

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。

如何在项目中灵活应用开闭原则?

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。

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，在编写代码的时候之后，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。

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。

重点回顾

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）。

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”?

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言

下雨天

对拓展开放是为了应对变化(需求)，对修改关闭是为了保证已有代码的稳定性；最终结果是为了让系统更有弹性！

2019-12-09 05:59

辣么大

开闭原则：基于接口或抽象实现“封闭”，基于实现接口或继承实现“开放”（拓展）。

争哥的第一个例子，AlertHandler为抽象，一般是固定不变的。子类TpsAlertHandler为继承；再看第二个例子，MessageQueue，MessageFormater为接口，具体实现为KafkaMessageQueue和JsonMessageFromatter等。以后替换或者增加其他的AlertHandler和message queue很容易。

两个例子中的抽象类和接口是固定的（封闭），继承或实现是可扩展的。通过“抽象-具体”体现了开闭原则，增加了软件的可维护性。

开闭原则具体应用，需要慢慢积累经验。争哥也说了，首先需要有对业务深刻的理解。其次就是学习一些设计原则和模式了。

补充：
1、Bertrand Meyer 1988 年提出open-closed principle。
2、再推荐一篇经典文章 Robert C. Martin 2006年写的The Open-Closed Principle。不方便下载的话，我放到github上了：https://github.com/gdhucoder/Algorithms4/tree/master/designpattern/pdf
2019-12-09 08:01

Paul Shan
基于一定的粒度（例如模块，类，属性等），扩展是平行地增加，修改是变更更细粒度的子集。扩展和修改和具体的粒度有关。不同的粒度下，扩展和修改定义不同。
我个人以为，扩展的结果是引入了更多的平行结构（例如相似的派生类handler），以及支持这些平行结构的代码（利用多态，在关键的地方使用接口）。这些引入会让代码结构变的扁平一些，但是也更晦涩一些。修改，往往会增加代码的深度（这里指更低粒度的复杂度），例如，文中log例子，修改后，check函数有五个参数，内部的if else逻辑更多。但是，如果从参数以及if作用域的角度，这也可算作扩展。所以，扩展还是修改更本质的区别在于修改发生的粒度和层次。
通常偏好修改发生在更高的层次上，这要求我们能够用接口和组合把系统合理的切分，做到高内聚和低耦合。高内聚可以让修改发生在更高层次上，替换掉整个低层次实现细节。低耦合，可以让模块之间的调用最小化，可以让高层次的修改最小化。
支持高层次的平行结构不是免费的，除非有明确的收益（例如文中隔离Kafka实现细节的例子），不然还是让重构等待到需要的那一刻，预测未来的大部分平行结构其实不会被真正用到。
2019-12-09 05:10


墨雨
听前一部分的时候觉得，哇原来代码还可以这样重构，我以后写代码一定要这么写！看到最后，恩……还是要结合具体业务需求，考虑实现开闭的代价，取舍哪些需要适应开闭原则哪些可以忽略，避免过度设计。整体来说在写代码的时候要多思考一下如何应对短期内可能面临的变化。知识+经验+多思考，看起来编程没有银弹，多思考，多总结。
2019-12-09 09:01


知行合一
对原有代码改动越少，引入问题的概率越小。在熟悉业务的情况下，多思考一步，为了未来需求预留扩展点，就是有扩展性的代码。但凡事都有个度，扩展性和可读性需要不断权衡，需求在不断变化，系统也在不断演化，通过不断重构来解决过度设计的问题。
2019-12-09 05:30


(´田ω田`)
修改老功能，可能需要重新进行各种功能验证、测试，并且如果是接收的遗留代码，更是费时费力；
但是扩展的话，只需要对自己新增加的功能进行测试，工作量会小很多。
2019-12-09 01:47

古杨
我所在的公司，现在写代码入参全用map，写了两年我都不知道什么叫对象了。感觉自己废了
2019-12-26 07:28


木木
文章写的是真的好，很容易读懂。主要的还是要知道为什么要这么做。感谢老师。
2019-12-11 10:17


李小四
设计模式_16
作业：
开闭原则核心好处是：
- 减少因为新增功能而增加的工作量
- 减少因为新增功能而增加的出错数

感想：
之前一直有一些执念，想要找到某一原则非黑即白的分割线。比如开闭原则，有两个极端：
- 任何的“修改”都不能接受
- 任何不能“扩展”的代码都不能接受
然后就进入了“走火入魔”的状态，最终陷入对原则的怀疑。

需求变更对于代码结构影响很大时，要提高对其扩展的权重；读到这里时，我拍了一下大腿，我想，我更加理解开闭原则了。
2019-12-09 09:02

简单来说，就是尽量减少调用方为了应对而导致的变更。
就例如本文的例子，为了应对变化需要增加函数的参数的时候。所有调用方都需要改代码。
而如果依照开闭原则，则增加handler 以及相应修改即可。并不会影响调用方。

其实个人认为，也是通过了 类似于“中间件”的形式。例如，小明，作为公司代表需要跟各个国外公司的人谈业务。他去跟美国人谈业务，需要学英语；跟日本谈业务，要学日语；跟毛子谈业务，又要学毛子语。
这时候，的解决方案：
1，跟各个国家说好，大家都说英语。或者都说汉语。就算再有其他的国家，也让他强制用英语。
2，小明自己只用汉语。然后谈业务时，带个多语种翻译，去谈业务时把翻译带上。这时候，如果有新的国家需要新的语种，那么就让翻译去掌握更多的语种。
应对今天的例子，翻译掌握的语种，其实就是handler。小明和各国代表谈业务时，各自都不需要变更自己的接口。只需要对【翻译】进行扩展即可。

2019-12-10 10:43

土豆哪里挖

什么时候出其他语言的demo呢，不懂java，理解起来太痛苦了

2019-12-09 20:11

作者回复

关注我的github：https://github.com/wangzheng0822

2019-12-09 20:47

梦倚栏杆

关于修改后的报警规则代码实现有两个疑问：
1. ApiStateInfo class 是充血模型还是贫血模型。
2.其实各个handler侧重的是不同的方方面面，比如错误次数，超时次数。统一接收ApiStateInfo 和 某一个handler接收具体的类比如：ErrorRequestApiStateInfo, TimeOutStateInfo， 哪种方式好呢？比较依据是什么

2019-12-09 07:07

作者回复

1. 是贫血模型
2. 不好讲，拆分之后，类增加，维护成本高一些，但职责更单一，更加高内聚、低耦合，扩展性更好些。

2019-12-09 20:58

feifei

文中的alter 一步一步的改造，看的眼花缭乱的 ，我就问下，为什么不能直接在原始的Alter 类中，重载一个只有新增业务参数的check 放到的，这样不就最简单，原先开发好也不用动，这样对于Alter 类来说不是对扩展开放，对修改关爱了吗？请教下大神，我这种用重载的思路有啥，不好的地方

2019-12-18 11:02

作者回复

重载可以。但如果报警规则很多的花 类会无限膨胀 可读性比较茶

2019-12-20 07:26

wenxueliu

spring 是如何应用开闭原则的，可以参考本文https://blog.csdn.net/wenxueliu/article/details/103467359

2019-12-10 12:14

辉仔lovers

老师 您好，请教几个问题
AlertHandler 使用的是抽象类，而不是接口。就是为了让子类去继承构造方法吗？
这个扩展跟spring中handlerMapping的写法一样，类似于策略模式吧？
单例模式的时候 使用静态代码块来初始化添加handler 随着类加载一次是不是就不用搞成单例的了？

```
static{
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(alertRule,notification);
alert.addAlertHandler(new TpsAlertHandler(alertRule, notification));
alert.addAlertHandler(new ErrorAlertHandler(alertRule, notification));
}
```

我们是不是可以把实现类（不同的handler）放到配置文件中。使用jdk的spi扩展机制。更加灵活一些?

2019-12-09 11:17

小晏子

对于课后题，想到2点:

1，减少出错概率，修改出错的概率比扩展要大

2，边界的问题，比如用户边界，尽量减少用户侧代码的改动，比如文中alert的事例，check函数本身的修改意味着所有使用的地方都要修改，而使用了开闭原则的代码对于老用户是无须修改的，降低了用户修改的成本。

2019-12-09 08:26

薯片

if分支很多，用handler导致类爆炸怎么处理?

2019-12-29 20:00

作者回复

没看懂你说的 为什么handler类会爆炸呢

2019-12-30 08:33

deepz

老师您好，我把代码实践了后发现， 单例初始化那块可能有点问题。private static final ApplicationContext instance = new ApplicationContext();

private ApplicationContext() {
instance.initializeBeans();
}
这个“instance”报了空指针。

2019-12-10 14:23

哈喽沃德

我想这篇对扩展开放，多修改关闭的文章应该会成为争哥这个设计模式系列最好的文章。很难想象，一个杰出的程序员的语言思维逻辑也是如此清晰。

2019-12-10 10:01

L.

学到了，谢谢老师；

2019-12-09 10:35