

22 | 理论八：如何用迪米特法则（LOD）实现“高内聚、松耦合”？

王争 · 设计模式之美



今天，我们讲最后一个设计原则：迪米特法则。尽管它不像 SOLID、KISS、DRY 原则那样，人尽皆知，但它却非常实用。利用这个原则，能够帮我们实现代码的“高内聚、松耦合”。今天，我们就围绕下面几个问题，并结合两个代码实战案例，来深入地学习这个法则。

什么是“高内聚、松耦合”？

如何利用迪米特法则来实现“高内聚、松耦合”？

有哪些代码设计是明显违背迪米特法则的？对此又该如何重构？

话不多说，让我们开始今天的学习吧！

何为“高内聚、松耦合”？

“高内聚、松耦合”是一个非常重要的设计思想，能够有效地提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。实际上，在前面的章节中，我们已经多次提到过这个设计思想。很多设计原则都以实现代码的“高内聚、松耦合”为目的，比如单一职责原则、基于接口而非实现编程等。

实际上，“高内聚、松耦合”是一个比较通用的设计思想，可以用来指导不同粒度代码的设计与开发，比如系统、模块、类，甚至是函数，也可以应用到不同的开发场景中，比如微服务、框架、组件、类库等。为了方便我讲解，接下来我以“类”作为这个设计思想的应用对象来展开讲解，其他应用场景你可以自行类比。

在这个设计思想中，“高内聚”用来指导类本身的设计，“松耦合”用来指导类与类之间依赖关系的设计。不过，这两者并非完全独立不相干。高内聚有助于松耦合，松耦合又需要高内聚的支持。

那到底什么是“高内聚”呢？

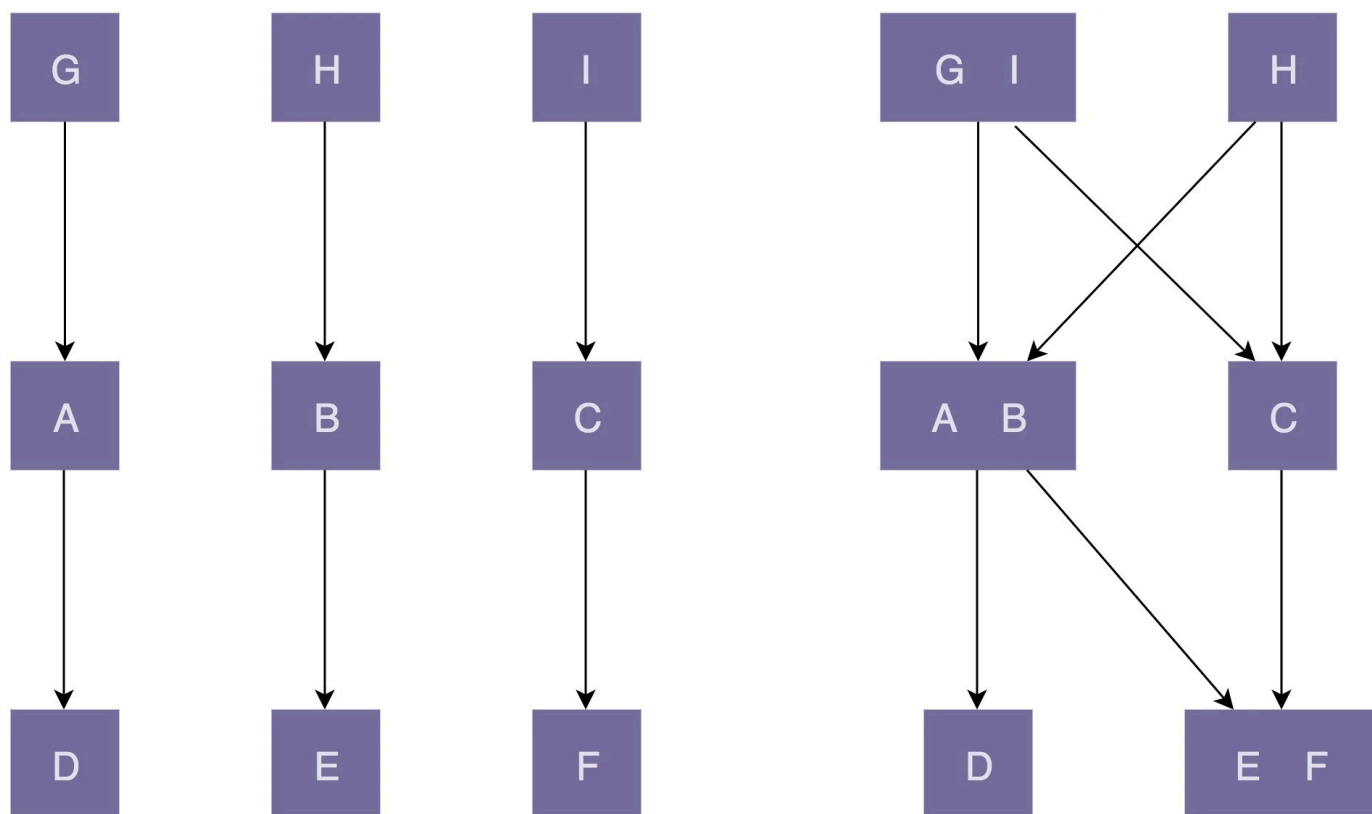
所谓高内聚，就是指相近的功能应该放到同一个类中，不相近的功能不要放到同一个类中。相近的功能往往会被同时修改，放到同一个类中，修改会比较集中，代码容易维护。实际上，我们前面讲过的单一职责原则是实现代码高内聚非常有效的设计原则。对于这一点，你可以回过头再去看下专栏的 [第 15 讲](#)。

我们再来看一下，什么是“松耦合”？

所谓松耦合是说，在代码中，类与类之间的依赖关系简单清晰。即使两个类有依赖关系，一个类的代码改动不会或者很少导致依赖类的代码改动。实际上，我们前面讲的依赖注入、接口隔离、基于接口而非实现编程，以及今天讲的迪米特法则，都是为了实现代码的松耦合。

最后，我们来看一下，“内聚”和“耦合”之间的关系。

前面也提到，“高内聚”有助于“松耦合”，同理，“低内聚”也会导致“紧耦合”。关于这一点，我画了一张对比图来解释。图中左边部分的代码结构是“高内聚、松耦合”；右边部分正好相反，是“低内聚、紧耦合”。



图中左边部分的代码设计中，类的粒度比较小，每个类的职责都比较单一。相近的功能都放到了一个类中，不相近的功能被分割到了多个类中。这样类更加独立，代码的内聚性更好。因为职责单一，所以每个类被依赖的类就会比较少，代码低耦合。一个类的修改，只会影响到一个依赖类的代码改动。我们只需要测试这一个依赖类是否还能正常工作就行了。

图中右边部分的代码设计中，类粒度比较大，低内聚，功能大而全，不相近的功能放到了一个类中。这就导致很多其他类都依赖这个类。当我们修改这个类的某一个功能代码的时候，会影响依赖它的多个类。我们需要测试这三个依赖类，是否还能正常工作。这也就是所谓的“牵一发而动全身”。

除此之外，从图中我们也可以看出，高内聚、低耦合的代码结构更加简单、清晰，相应地，在可维护性和可读性上确实要好很多。

“迪米特法则”理论描述

迪米特法则的英文翻译是：Law of Demeter，缩写是 LOD。单从这个名字上来看，我们完全猜不出这个原则讲的是什么。不过，它还有另外一个更加达意的名字，叫作最小知识原则，英文翻译为：The Least Knowledge Principle。

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，对刚刚的定义重新描述一下。注意，为了统一讲解，我把定义描述中的“模块”替换成了“类”。


不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口（也就是定义中的“有限知识”）。

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。

理论解读与代码实战一

我们先来看这条原则中的前半部分，“**不该有直接依赖关系的类之间，不要有依赖**”。我举个例子解释一下。

这个例子实现了简化版的搜索引擎爬取网页的功能。代码中包含三个主要的类。其中，NetworkTransporter 类负责底层网络通信，根据请求获取数据；HtmlDownloader 类用来通过 URL 获取网页；Document 表示网页文档，后续的网页内容抽取、分词、索引都是以此为处理对象。具体的代码实现如下所示：

 复制代码


```
1 public class NetworkTransporter {
2     // 省略属性和其他方法...
3     public Byte[] send(HtmlRequest htmlRequest) {
4         //...
5     }
6 }
7
8 public class HtmlDownloader {
9     private NetworkTransporter transporter;//通过构造函数或IOC注入
10
11     public Html downloadHtml(String url) {
12         Byte[] rawHtml = transporter.send(new HtmlRequest(url));
13         return new Html(rawHtml);
14     }
15 }
16
17 public class Document {
18     private Html html;
19     private String url;
20
21     public Document(String url) {
22         this.url = url;
23         HtmlDownloader downloader = new HtmlDownloader();
24         this.html = downloader.downloadHtml(url);
25     }
26     //...
27 }
```

这段代码虽然“能用”，能实现我们想要的功能，但是它不够“好用”，有比较多的设计缺陷。你可以先试着思考一下，看看都有哪些缺陷，然后再来看我下面的讲解。

首先，我们来看 NetworkTransporter 类。作为一个底层网络通信类，我们希望它的功能尽可能通用，而不只是服务于下载 HTML，所以，我们不应该直接依赖太具体的发送对象


HttpRequest。从这一点上讲，NetworkTransporter 类的设计违背迪米特法则，依赖了不该有直接依赖关系的 HttpRequest 类。

我们应该如何进行重构，让 NetworkTransporter 类满足迪米特法则呢？我这里有个形象的比喻。假如你现在要去商店买东西，你肯定不会直接把钱包给收银员，让收银员自己从里面拿钱，而是你从钱包里把钱拿出来交给收银员。这里的 HttpRequest 对象就相当于钱包，HttpRequest 里的 address 和 content 对象就相当于钱。我们应该把 address 和 content 交给 NetworkTransporter，而非是直接把 HttpRequest 交给 NetworkTransporter。根据这个思路，NetworkTransporter 重构之后的代码如下所示：

 复制代码

```
1 public class NetworkTransporter {
2     // 省略属性和其他方法...
3     public Byte[] send(String address, Byte[] data) {
4         //...
5     }
6 }
```


我们再来看 HtmlDownloader 类。这个类的设计没有问题。不过，我们修改了 NetworkTransporter 的 send() 函数的定义，而这个类用到了 send() 函数，所以我们需要对它做相应的修改，修改后的代码如下所示：

 复制代码

```
1 public class HtmlDownloader {
2     private NetworkTransporter transporter;//通过构造函数或IOC注入
3
4     // HtmlDownloader这里也要有相应的修改
5     public Html downloadHtml(String url) {
6         HttpRequest htmlRequest = new HttpRequest(url);
7         Byte[] rawHtml = transporter.send(
8             htmlRequest.getAddress(), htmlRequest.getContent().getBytes());
9         return new Html(rawHtml);
10    }
11 }
```

最后，我们来看下 **Document** 类。这个类的问题比较多，主要有三点。第一，构造函数中的 `downloader.downloadHtml()` 逻辑复杂，耗时长，不应该放到构造函数中，会影响代码的可测试性。代码的可测试性我们后面会讲到，这里你先知道有这回事就可以了。第二，`HtmlDownloader` 对象在构造函数中通过 `new` 来创建，违反了基于接口而非实现编程的设计思想，也会影响到代码的可测试性。第三，从业务含义上来讲，`Document` 网页文档没必要依赖 `HtmlDownloader` 类，违背了迪米特法则。

虽然 `Document` 类的问题很多，但修改起来比较简单，只要一处改动就可以解决所有问题。修改之后的代码如下所示：

 复制代码

```
1 public class Document {
2     private Html html;
3     private String url;
4
5     public Document(String url, Html html) {
6         this.html = html;
7         this.url = url;
8     }
9     //...
10 }
11
12 // 通过一个工厂方法来创建Document
13 public class DocumentFactory {
14     private HtmlDownloader downloader;
15
16     public DocumentFactory(HtmlDownloader downloader) {
17         this.downloader = downloader;
18     }
19
20     public Document createDocument(String url) {
21         Html html = downloader.downloadHtml(url);
22         return new Document(url, html);
23     }
24 }
```

理论解读与代码实战二


现在，我们再来看一下这条原则中的后半部分：“有依赖关系的类之间，尽量只依赖必要的接口”。我们还是结合一个例子来讲解。下面这段代码非常简单，Serialization 类负责对象的序列化和反序列化。提醒你一下，有个类似的例子在之前的第 15 节课中讲过，你可以结合着一块儿看一下。

 复制代码

```
1 public class Serialization {
2     public String serialize(Object object) {
3         String serializedResult = ...;
4         //...
5         return serializedResult;
6     }
7
8     public Object deserialize(String str) {
9         Object deserializedResult = ...;
10        //...
11        return deserializedResult;
12    }
13 }
```

单看这个类的设计，没有一点问题。不过，如果我们把它放到一定的应用场景里，那就还有继续优化的空间。假设在我们的项目中，有些类只用到了序列化操作，而另一些类只用到反序列化操作。那基于迪米特法则后半部分“有依赖关系的类之间，尽量只依赖必要的接口”，只用到序列化操作的那部分类不应该依赖反序列化接口。同理，只用到反序列化操作的那部分类不应该依赖序列化接口。

根据这个思路，我们应该将 Serialization 类拆分为两个更小粒度的类，一个只负责序列化（Serializer 类），一个只负责反序列化（Deserializer 类）。拆分之后，使用序列化操作的类只需要依赖 Serializer 类，使用反序列化操作的类只需要依赖 Deserializer 类。拆分之后的代码如下所示：

 复制代码

```
1 public class Serializer {
2     public String serialize(Object object) {
3         String serializedResult = ...;
4         ...
5         return serializedResult;
6     }
7 }
```




```

6    }
7  }
8
9  public class Deserializer {
10     public Object deserialize(String str) {
11         Object deserializedResult = ...;
12         ...
13         return deserializedResult;
14     }
15 }

```

不知道你有没有看出来，尽管拆分之后的代码更能满足迪米特法则，但却违背了高内聚的设计思想。高内聚要求相近的功能要放到同一个类中，这样可以方便功能修改的时候，修改的地方不至于过于分散。对于刚刚这个例子来说，如果我们修改了序列化的实现方式，比如从 JSON 换成了 XML，那反序列化的实现逻辑也需要一并修改。在未拆分的情况下，我们只需要修改一个类即可。在拆分之后，我们需要修改两个类。显然，这种设计思路的代码改动范围变大了。

如果我们既不想违背高内聚的设计思想，也不想违背迪米特法则，那我们该如何解决这个问题呢？实际上，通过引入两个接口就能轻松解决这个问题，具体的代码如下所示。实际上，我们在 [第 18 节课](#) 中讲到“接口隔离原则”的时候，第三个例子就使用了类似的实现思路，你可以结合着一块儿来看。

 复制代码

```

1  public interface Serializable {
2      String serialize(Object object);
3  }
4
5  public interface Deserializable {
6      Object deserialize(String text);
7  }
8
9  public class Serialization implements Serializable, Deserializable {
10     @Override
11     public String serialize(Object object) {
12         String serializedResult = ...;
13         ...
14         return serializedResult;
15     }
16 }

```

```

17  @Override
18  public Object deserialize(String str) {
19      Object deserializedResult = ...;
20      ...
21      return deserializedResult;
22  }
23 }
24
25 public class DemoClass_1 {
26     private Serializable serializer;
27
28     public Demo(Serializable serializer) {
29         this.serializer = serializer;
30     }
31     //...
32 }
33
34 public class DemoClass_2 {
35     private Deserializable deserializer;
36
37     public Demo(Deserializable deserializer) {
38         this.deserializer = deserializer;
39     }
40     //...
41 }

```

尽管我们还是要往 DemoClass_1 的构造函数中，传入包含序列化和反序列化的 Serialization 实现类，但是，我们依赖的 Serializable 接口只包含序列化操作，DemoClass_1 无法使用 Serialization 类中的反序列化接口，对反序列化操作无感知，这也就符合了迪米特法则后半部分所说的“依赖有限接口”的要求。

实际上，上面的代码实现思路，也体现了“基于接口而非实现编程”的设计原则，结合迪米特法则，我们可以总结出一条新的设计原则，那就是“基于最小接口而非最大实现编程”。有些同学之前问，新的设计模式和设计原则是怎么创造出来的，实际上，就是在大量的实践中，针对开发痛点总结归纳出来的套路。

辩证思考与灵活应用

对于实战二最终的设计思路，你有没有什么不同的观点呢？

整个类只包含序列化和反序列化两个操作，只用到序列化操作的使用者，即便能够感知到仅有的一个反序列化函数，问题也不大。那为了满足迪米特法则，我们将一个非常简单的类，拆分成两个接口，是否有点过度设计的意思呢？

设计原则本身没有对错，只有能否用对之说。不要为了应用设计原则而应用设计原则，我们在应用设计原则的时候，一定要具体问题具体分析。

对于刚刚这个 `Serialization` 类来说，只包含两个操作，确实没有太大必要拆分成两个接口。但是，如果我们对 `Serialization` 类添加更多的功能，实现更多更好用的序列化、反序列化函数，我们来重新考虑一下这个问题。修改之后的具体的代码如下：

 复制代码

```
1 public class Serializer { // 参看JSON的接口定义
2     public String serialize(Object object) { //... }
3     public String serializeMap(Map map) { //... }
4     public String serializeList(List list) { //... }
5
6     public Object deserialize(String objectString) { //... }
7     public Map deserializeMap(String mapString) { //... }
8     public List deserializeList(String listString) { //... }
9 }
```

在这种场景下，第二种设计思路要更好些。因为基于之前的应用场景来说，大部分代码只需要用到序列化的功能。对于这部分使用者，没必要了解反序列化的“知识”，而修改之后的 `Serialization` 类，反序列化的“知识”，从一个函数变成了三个。一旦任一反序列化操作有代码改动，我们都需要检查、测试所有依赖 `Serialization` 类的代码是否还能正常工作。为了减少耦合和测试工作量，我们应该按照迪米特法则，将反序列化和序列化的功能隔离开来。

重点回顾

好了，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“高内聚、松耦合”？

“高内聚、松耦合”是一个非常重要的设计思想，能够有效提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。“高内聚”用来指导类本身的设计，“松耦合”用来指导类与类之间依赖关系的设计。

所谓高内聚，就是指相近的功能应该放到同一个类中，不相近的功能不要放到同一类中。相近的功能往往会被同时修改，放到同一个类中，修改会比较集中。所谓松耦合指的是，在代码中，类与类之间的依赖关系简单清晰。即使两个类有依赖关系，一个类的代码改动也不会或者很少导致依赖类的代码改动。

2. 如何理解“迪米特法则”？

不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口。迪米特法则是希望减少类之间的耦合，让类越独立越好。每个类都应该少了解系统的其他部分。一旦发生变化，需要了解这一变化的类就会比较少。

课堂讨论

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？

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。

AI智能总结

迪米特法则是一个重要的设计原则，旨在实现代码的“高内聚、松耦合”。文章首先介绍了“高内聚、松耦合”设计思想的重要性和实际应用，强调了高内聚有助于松耦合，而低内聚则会导致紧耦合。接着，文章详细解释了迪米特法则的理论描述，即每个模块（类）只应该了解与之关系密切的模块的有限知识，或者说每个模块只和自己的朋友“说话”，不和陌生人“说话”。迪米特法则的核心是避免直接依赖关系的类之间存在依赖，尽量只依赖必要的接口，以实现代码的松耦合。文章还提供了两个代码实战案例，以帮助读者更好地理解迪米特法则的应用。

第一个案例展示了如何根据迪米特法则重构代码，避免直接依赖关系的类之间存在依赖，以实现松耦合。通过对NetworkTransporter、HtmlDownloader和Document类的重构，读者可以清晰地了解如何应用迪米特法则来改善代码设计，提高代码的可维护性和可测试性。

第二个案例则展示了如何根据迪米特法则的后半部分原则，即“有依赖关系的类之间，尽量只依赖必要的接口”，来优化代码设计。文章通过拆分Serialization类为Serializer和Deserializer类，并引入接口的方式，解决

了迪米特法则和高内聚原则之间的矛盾，使得代码更加灵活和可维护。

通过本文的学习，读者可以深入了解迪米特法则的原理和实际应用，从而在实际开发中实现“高内聚、松耦合”的代码设计目标。文章还提到了“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”，这些设计原则之间的区别和联系，帮助读者更好地理解和应用这些原则。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (126)

最新 精选



王大喵

2020-11-16

联系：

“接口隔离原则”是客户端不应该被强迫依赖不需要的接口，和“迪米特法则”中的有限知识异曲同工，接口簇会更加“单一职责”实现方式“基于接口而非实现编程”，达到的目的是高内聚，松耦合。

区别：

1. 各种原则最终的目的是为了实现“高内聚、松耦合”。
2. 单一职责原则 主要是指导类和模块，避免大而全，提高内聚性。
3. 接口隔离和迪米特(最小知识)主要指导“松耦合”，解耦使用方的依赖。
4. 基于接口而非实现编程：主要是解耦接口和实现，是指导思想，提高扩展性。

作者回复: 🤔🤔🤔🤔🤔🤔

共 2 条评论 >

👍 14



戒惜舍得

2019-12-27

相近的功能。 怎么算相近啊。学晕了。

作者回复: 咋这么能抬杠呢 你说我问你怎么相近 你能回答上来吗 这个也没法量化 设计模式不是算法 什么都能定量 更多的是教你思想 你自己去感受

共 2 条评论 >

👍 4



提姆

2020-07-07

老师您好，请问一下Document的修改为什么会使用到Factory的方式去产生？我对这一步的修改没有很大的感受，我这里的认知是Document只是单纯对文件的操作，那是不是可以透过HtmlDownloader实现相关取得文件的接口，像是IDownloader之类的名称，并直接回传Document这个类(亦或者对此类做其它的延伸)，未来也可以实现其它像是json等其他格式的downloader，不知道我这个想法是不是可行？

作者回复: 你的设计也可以的。设计本身没有最优解，合理、能自圆其说就可以了。总体上来讲，这个例子是为了展示尽量减少类之间的耦合。

共 2 条评论 >

👍 3



大方方

2020-06-29

我想知道假如

```
Public class NetworkTransporter {  
    // 省略属性和其他方法...  
    public Byte[] send(HtmlRequest htmlRequest) {  
  
        //...    }  
}
```

中的send 方法，必须需要HtmlRequest 才能实现功能呢？老师修改后，参数上看起来是不依赖外部对象了，但是在很多其他实际操作时，很有可能还是需要用到外部对象来解决问题。这种情况是不是需要做类扩展，在扩展中再具体引用HtmlRequest？

作者回复: “很有可能还是需要用到外部对象来解决问题”能举个例子吗？

“这种情况是不是需要做类扩展，在扩展中再具体引用HtmlRequest”也可以直接改这个类~，不过还是没太懂你的意思~

共 2 条评论 >

👍



斐波那契

2019-12-29

老师 我有一个问题：在项目开发中，我写了一个类A 里面定义了一个方法，后来又写了一个类B 发现在B里面要用到A里面定义的那个方法(基本上一模一样)，但是A跟B本身是两个不相

关的类 这个时候要怎么解决? PS:这个方法不能算是工具类, 只是一段数据集的处理逻辑
那是否是搞一个抽象类 然后让A和B继承这个抽象类 把那个方法写进抽象类里?

作者回复: 使用组合能不能解决复用问题呢

共 2 条评论 >



prowu

2019-12-28

一直有一个消息结构与程序内部数据结构取舍的问题: 程序内部是否直接复用消息协议的结构? 比如: 通讯消息使用的是protobuf协议, 那程序内部的逻辑是直接使用protobuf的数据结构, 还是自己在定义一套结构体? 如果直接使用protobuf协议, 那程序就紧耦合于协议了(这边就是与protobuf绑在一起了), 如果自己在定义一套结构体, 那就要多一层协议与内部结构的转换。

作者回复: 这个也要具体问题具体分析 跟我们讲的vo bo很像

共 2 条评论 >



知行合一

2019-12-23

目的都是实现高内聚低耦合, 但是出发的角度不一样, 单一职责是从自身提供的功能出发, 迪米特法则则是从关系出发, 针对接口而非实现编程是使用者的角度, 殊途同归。

共 9 条评论 >

383



下雨天

2019-12-25

1.单一职责原则

适用对象:模块, 类, 接口

侧重点:高内聚, 低耦合

思考角度:自身

2.接口隔离原则

适用对象:接口, 函数

侧重点:低耦合

思考角度:调用者

3.基于接口而非实现编程

适用对象:接口, 抽象类

侧重点:低耦合

思考角度:调用者

4.迪米特法则

适用对象:模块, 类

侧重点:低耦合

思考角度:类关系

共 12 条评论 >

👍 236



Ken张云忠

2019-12-23

“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”, 它们之间的区别和联系吗?

区别:

高内聚、松耦合:是一个重要的设计思想,能够有效地提高代码的可读性和可维护性,缩小功能改动导致的代码改动范围.

单一职责原则:A class or module should have a single responsibility.提供的功能上要单一.

接口隔离原则:Clients should not be forced to depend upon interfaces that they do not use.与外部关系上只依赖需要的抽象.

基于接口而非实现编程:Program to an interface, not an implementation.是一条比较抽象、泛化的设计思想,为了提高代码的灵活性/扩展性/可维护性.

迪米特法则: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.每个单元只该依赖与它关系密切的单元,最少知道,只与关系密切的单一交互.

联系:

职责越单一越容易做到接口隔离,也越容易做到最少知道的迪米特法则.

基于抽象编程抽象的知识越顶层越脱离具体实现,相对知道的内容就越少,也容易实现迪米特法则.

接口隔离原则与迪米特法则都强调只依赖需要的部分,接口隔离原则是相对偏上层来说的,迪米特法则是相对偏具体实现来说的.

单一职责原则/接口隔离原则/基于接口而非实现编程/迪米特法则都以实现代码的“高内聚、松耦合”为目的,提高代码的可读性和可维护性,缩小功能改动导致的代码改动范围,降低风险.



👍 58



辣么大

2019-12-23

关于LoD，请记住一条：方法中不要使用ChainMethods。

坏的实践：

```
Amount = customer.orders().last().totals().amount()
```

和

```
orders = customer.orders()
```

```
lastOrders = orders.last()
```

```
totals = lastOrders.totals()
```

```
amount = totals.amount()
```

上面的例子中，chain中的方法改变会影响很多地方。这里注意区别建造者模式和pipeline管道，这两种的chain中的方法不易改变。

出现这样的代码，需要考虑可能是设计或实现出了问题。

LoD如何使用：

一个类C中的方法只能调用：

- 1、C中其他实例方法
- 2、它自己的参数方法
- 3、它创建对象的方法
- 4、不要调用全局变量（包括可变对象、可变单例）

例如：

```
class HtmlDownloader{
    Html html;
    public void downloadHtml(Transporter trans, String url){
        if(checkUrl(url)){// ok 自己的实例方法
            // return
        }
        rawData = trans.send(uri);// ok 参数对象的方法
        Html html = createHtml(rawData); // ok 它创建的对象
        html.save();// ok 它创建对象的方法
    }
    private boolean checkUrl(String url){
        // check
    }
}
```

参考：

The Pragmatic Programmer 1st edition and 2nd edition

共 24 条评论 >

 41