22-理论八：如何用迪米特法则（LOD）实现“高内聚、松耦合”？

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?

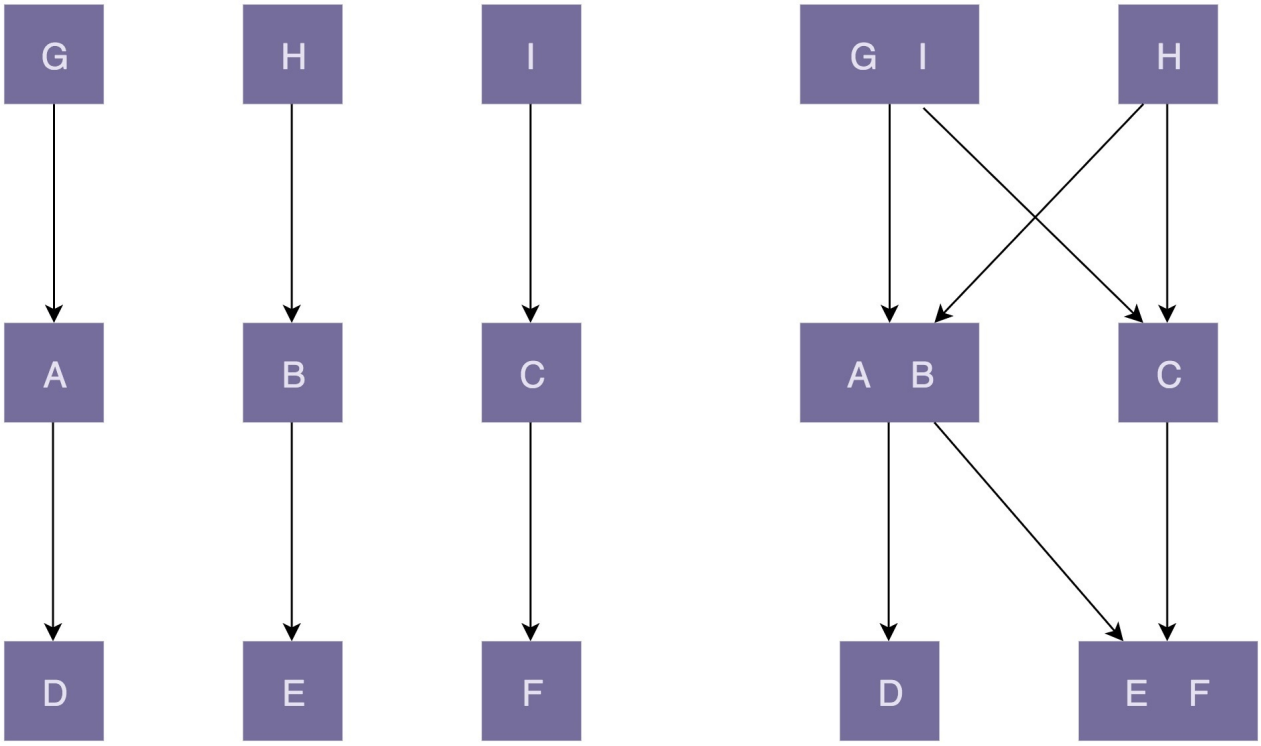
所谓高内聚，就是指相近的功能应该放到同一个类中，不相近的功能不要放到同一个类中。相近的功能往往会被同时修改，放到同一个类中，修改会比较集中，代码容易维护。实际上，我们前面讲过的单一职责原则是实现代码高内聚非常有效的设计原则。对于这一点，你可以回过头再去看下专栏的第15讲。

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：

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

public class HtmlDownloader {
  private NetworkTransporter transporter;//通过构造函数或IOC注入

  public Html downloadHtml(String url) {
    Byte[] rawHtml = transporter.send(new HtmlRequest(url));
    return new Html(rawHtml);
  }
}

public class Document {
  private Html html;
  private String url;

  public Document(String url) {
    this.url = url;
    HtmlDownloader downloader = new HtmlDownloader();
    this.html = downloader.downloadHtml(url);
  }
  //...
}
```

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HtmlRequest。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HtmlRequest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HtmlRequest对象就相当于钱包，HtmlRequest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HtmlRequest交给NetworkTransporter。根据这个思路，NetworkTransporter重构之后的代码如下所示：

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

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：

```
public class HtmlDownloader {
  private NetworkTransporter transporter;//通过构造函数或IOC注入

  // HtmlDownloader这里也要有相应的修改
  public Html downloadHtml(String url) {
    HtmlRequest htmlRequest = new HtmlRequest(url);
    Byte[] rawHtml = transporter.send(
      htmlRequest.getAddress(), htmlRequest.getContent().getBytes());
    return new Html(rawHtml);
  }
}
```

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：

```
public class Document {
  private Html html;
  private String url;

  public Document(String url, Html html) {
    this.html = html;
    this.url = url;
  }
  //...
}

// 通过一个工厂方法来创建Document
public class DocumentFactory {
  private HtmlDownloader downloader;

  public DocumentFactory(HtmlDownloader downloader) {
    this.downloader = downloader;
  }

  public Document createDocument(String url) {
    Html html = downloader.downloadHtml(url);
    return new Document(url, html);
  }
}
```

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。

```
public class Serialization {
  public String serialize(Object object) {
    String serializedResult = ...;
    //...
    return serializedResult;
  }

  public Object deserialize(String str) {
    Object deserializedResult = ...;
    //...
    return deserializedResult;
  }
}
```

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：

```
public class Serializer {
  public String serialize(Object object) {
    String serializedResult = ...;
    ...
    return serializedResult;
  }
}

public class Deserializer {
  public Object deserialize(String str) {
    Object deserializedResult = ...;
    ...
    return deserializedResult;
  }
}
```

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。

```
public interface Serializable {
  String serialize(Object object);
}

public interface Deserializable {
  Object deserialize(String text);
}

public class Serialization implements Serializable, Deserializable {
  @Override
  public String serialize(Object object) {
    String serializedResult = ...;
    ...
    return serializedResult;
  }

  @Override
  public Object deserialize(String str) {
    Object deserializedResult = ...;
    ...
    return deserializedResult;
  }
}

public class DemoClass_1 {
  private Serializable serializer;
```

```
  public Demo(Serializable serializer) {
    this.serializer = serializer;
  }
  //...
}

public class DemoClass_2 {
  private Deserializable deserializer;

  public Demo(Deserializable deserializer) {
    this.deserializer = deserializer;
  }
  //...
}
```

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，拆分出两个接口，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：

```
public class Serializer { // 参看JSON的接口定义
  public String serialize(Object object) { //... }
  public String serializeMap(Map map) { //... }
  public String serializeList(List list) { //... }

  public Object deserialize(String objectString) { //... }
  public Map deserializeMap(String mapString) { //... }
  public List deserializeList(String listString) { //... }
}
```

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。

精选留言：

- 知行合一 2019-12-23 08:18:15
  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。 [25赞]

- Ken张云忠 2019-12-23 09:12:07
  “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 reponsibility.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. [8赞]

- 辣么大 2019-12-23 09:37:14
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()
lastOders = orders.last()
totals = lastOders.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
[6赞]

- 黄林晴 2019-12-23 07:43:14
  打卡
  所有的设计原则都相辅相成 [2赞]

- 落叶飞逝的恋 2019-12-23 22:11:03
  课后讨论：代码的最终目的是高内聚、松耦合的。而为了达到这个目的，就需要利用到迪米特法则。而迪米特法则的实现，又需要利用单一职责将单个类定义职责单一化，并且为了解决多个类之间的关系，又需要用到基于接口编程而非实现编程。这样类与类之间就相当于契约化，也就是不关心类的具体实现。 [1赞]

- 逆风星痕 2019-12-23 09:49:56
  感觉所有设计原则和模式，都是为了代码的可读性，复用和扩展而总结出来的。好多原则可能是针对某个场景下提高代码的复用和扩展，这样有时也会辅助其他原则。迪米特原则描述类之间的关系，尽量减少依赖，但也需要类遵循单一职责原则。设计代码的时候，可以根据自己的目的，从参考相应原则的设计 [1赞]

- 再见孙悟空 2019-12-23 08:39:26
  “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”都是为了实现 “高内聚、低耦合”的手段。做到了接口隔离，一般情况下职责也比较单一，基于接口而非实现编程，往往也会降低耦合性。有的时候使用了迪米特法则或者单一职责原则，可能会破坏高内聚原则，这种情况就要具体分析场景，以及使用接口来实现。 [1赞]

- 失火的夏天 2019-12-23 07:56:38
  接口隔离感觉就是为了迪米特法则的应用，接口隔离开不需要依赖的类，只引入需要的接口和方法。

  高内聚低耦合，是针对具体的实现类的，实现类实现多个接口，相似的功能都在同一个实现类中完成。

  接口的隔离又保证对外只暴露了调用方需要的方法，外部也不能直接看到不需要的方法。代码结构也更加整洁，逻辑更清晰 [1赞]

- 刘浒 2019-12-23 22:50:12
  “高内聚、松耦合” 是衡量好代码的标准之一，为了实现这样的目标，我们需要遵循如下原则：
  “基于接口而非实现编程”，接口本身就是一层抽象，接口是稳定的，实现是易变的，强调的是基于契约编程，这样能够隔离变化。实现细节代码的变化，不影依赖该接口的对象，到从而达到松耦合的目的。
  “迪米特法则”，定义的是发布的接口（类、模块等）能不能依赖，如何依赖的问题。使用者去除不必要的依赖，只依赖必要的接口。这样即使接口一旦发生变化，需要了解这一变化的类就会比较少，达到松耦合的目的。
  “接口隔离原则”，从使用者的角度考虑如何设计接口，让使用者只依赖必要的接口，不会被迫依赖不用的接口。这样即使接口一旦发生变化，需要了解这一变化的类就会比较少，这样就能符合 “迪米特法则”。
  “单一职责原则”，针对模块、类、接口的设计，将功能相关性很强的代码抽取到一起，达到高内聚的目标。

- Frank 2019-12-23 21:47:15
  个人觉得，单一职责原则、接口隔离原则、基于接口而非实现编程、迪米特法则都是实现高内聚和低耦合在不同角度的更加落地的思考。高内聚、低耦合是一个顶层目标，而这些原则是这个目标的不同角度的思

考，比如接口隔离原则，基于接口而非实现编程是从接口的角度来思考。单一职责原则是从模块、类的角度来思考。而迪米特法则强调units之间不该有的关系就不要有，最好关系是清清楚楚，明明白白的。

- DullBird 2019-12-23 20:32:02
同: 都是为了保证代码的稳定性，在变化的情况下，影响最小，不过各自纬度可能不同。
单一职责原则: 基于功能点，要结合业务来考虑是否职责是否单一
基于接口而非实现编程: 是基于变化的点，为了应对变化
“接口隔离原则”和“迪米特法则”: 没太区分开。我目前理解是一样的，在于对调用方的关系上面，做到无关的不用理解。
“高内聚、松耦合”: 是总的原则，类似于纲领，这个没太理解，总是挂嘴上，但是感觉有点模糊?

提问:
类似与序列化类的问题，代码的重构是渐进的。比如一开始，
class A{
//序列化
a();
//反序列化
b();
}
当有一天，有3个序列化和3个反序列化方法的时候，准备拆分了，就像文中一样，实现两个接口，代码重构一下即可以让每个调用方知道最少，符合迪米特法则,应用内部修改确实没什么问题。但是比如dubbo服务，如果一开始没拆开，后面想拆开。又要考虑向前兼容的问题，目前的做法就是class A 实现2个接口，新增6个方法，有2个和原来的序列化反序列化一样，标记旧的过期。通常还有什么更好的办法么?

- blacknhole 2019-12-23 20:04:35
对耦合的含义或理解方式有些不同看法：

1，与其从依赖关系来谈松耦合，说它是依赖关系简单清晰，不如从功能的相关性来谈，说它是无关功能不放在一个类中。这时，高内聚的含义就只是相关或相近功能放在一个类中了。

类之间的依赖关系，在类的设计完成之后，是无法选择和改变的，除非重新设计类。比如，类之间应该依赖时，在不改变类的设计的前提下，无法刻意不依赖或减少依赖。依赖关系简单清晰，只是在对高内聚和松耦合的适度追求之下，设计并实现类之后，自然产生的一个结果。

2，高内聚和松耦合，都是用来指导类本身的设计的。并非用松耦合来指导类之间依赖关系的设计。换句话说，类之间依赖关系的设计应该包含在类本身的设计之中（事实上，只有类的设计，而不存在什么类之间依赖关系的设计，见下文）。

在设计一个类时，应该考虑它与其他类的关系，以达到这个类的适度内聚和耦合。在这里，内聚是从功能相关的角度来观察类——有关的功能是不是放在一起了，耦合是从功能无关的相反角度来观察类——无关的功能是不是分散开来了。

依赖关系是类的设计完成之后，对类之间相关性的描述，有关就叫有依赖关系，无关就叫无依赖关系。所以，并不存在什么对依赖关系的设计，依赖关系是类的设计完成之后的一个自然结果。

3，这样的理解，对类的设计来说才是更有指导意义的，也才能使概念的边界足够清晰，从而使内聚与耦合的本质更易被准确理解。

- Kang 2019-12-23 18:01:46
打卡

- 岁月 2019-12-23 15:07:38
  如果说高内聚、松耦合等价于"中国特色社会主义", 那么“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”这几个原则就像是在说如何才能做到做到这样的社会? 答案就是我们要"倡导富强、民主、文明、和谐, 自由、平等、公正、法治，爱国、敬业、诚信、友善"

- Yes 2019-12-23 14:50:02
  我认为所有的设计原则和设计模式都是为了达到高内聚、松耦合的目的，而设计原则是指导思想，设计模式是前人根据设计原则再结合常见痛点得出的具体落地方案。

  按照指导思想设计出来的代码就是会符合高内聚、松耦合。

  指导思想之间也是相辅相成的。

  单一职责思考的角度是功能方面，一个类一个模块职责单一，而迪米特法则考虑的是最小化依赖原则，也就是在职责单一的情况下再深挖，尽可能的减少类之间的依赖关系，以达到松耦合的目的。

  并且在这个过程中我们肯定选择的是基于接口而非实现编程，依赖抽象而不是依赖实现。在这种情况下再考虑最小化依赖原则，我们肯定是想剥除接口里面一些此场景下不需要的方法，因此接口隔离原则就运用上了。

  也就是说平日我们开发中需要时刻的想着这几种原则，它们分别从不同的角度来指导我们。但是盲目的运用是没必要的，一个再也简单两三个方法纷纷运用上这些原则，就过犹不及了，要想着KISS。

  代码是持续改进、不断重构的。

- 守拙 2019-12-23 14:37:35
  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？

  Answer:

  1.高内聚与单一职责原则：

  高内聚形容类聚焦于本身的职责，是单一职责原则的具体体现。换句话说，遵守了单一职责原则而设计出的类，具有高内聚的特性。

  2.松耦合与接口隔离原则：

  低耦合形容类尽可能少的依赖其他类（接口），是接口隔离原则的具体体现。如果遵守接口隔离原则，就能设计出松耦合的类。

3.基于接口而非实现编程与迪米特法则

基于接口而非实现编程是一种面向对象设计思想，通过接口依赖而非具体类依赖的方式达到类之间松耦合的目的。

此思想一定意义上沉淀为迪米特法则：解耦类之间的依赖关系，能不依赖的，就不要依赖；必须依赖的，要依赖接口，而不依赖实现。总而言之：基于接口而非实现编程是形而上的飘渺思想，迪米特法则是萃取思想精华的戒律清规。在每日的修行中：遵守戒律，参悟思想，才能有所精进。

- whistleman 2019-12-23 13:45:26
  打卡～
  1.不该有直接依赖关系的类之间，不要有依赖；
  2.有依赖关系的类之间，尽量只依赖必要的接口。

- 睡觉zzz 2019-12-23 11:21:05
  高内聚:指将将相近的功能封装在一个类或者模块中，不要包含语义不相干的功能，是类或者模块本身的设计原则
  松耦合:是模块与模块、类与类之间的依赖关系。依赖的模块只包含我依赖的功能，没有附件功能。尽量让依赖关系成线性

  单一职责:是实现高内聚与松耦合的编程手段。每一个api，每一个类，每一个模块的职责要尽可能的单一
  基于接口编程:多而小的接口相互组合，让松耦合变得更加容易实现

- Askerlve 2019-12-23 10:58:47
  打卡，把自己知道的有层次的讲出来，让人易懂，真是一门学问呢～

- evolution 2019-12-23 10:26:21
  理论解读与代码实战二的解决办法，让我一下子明白了框架源码为何那么写