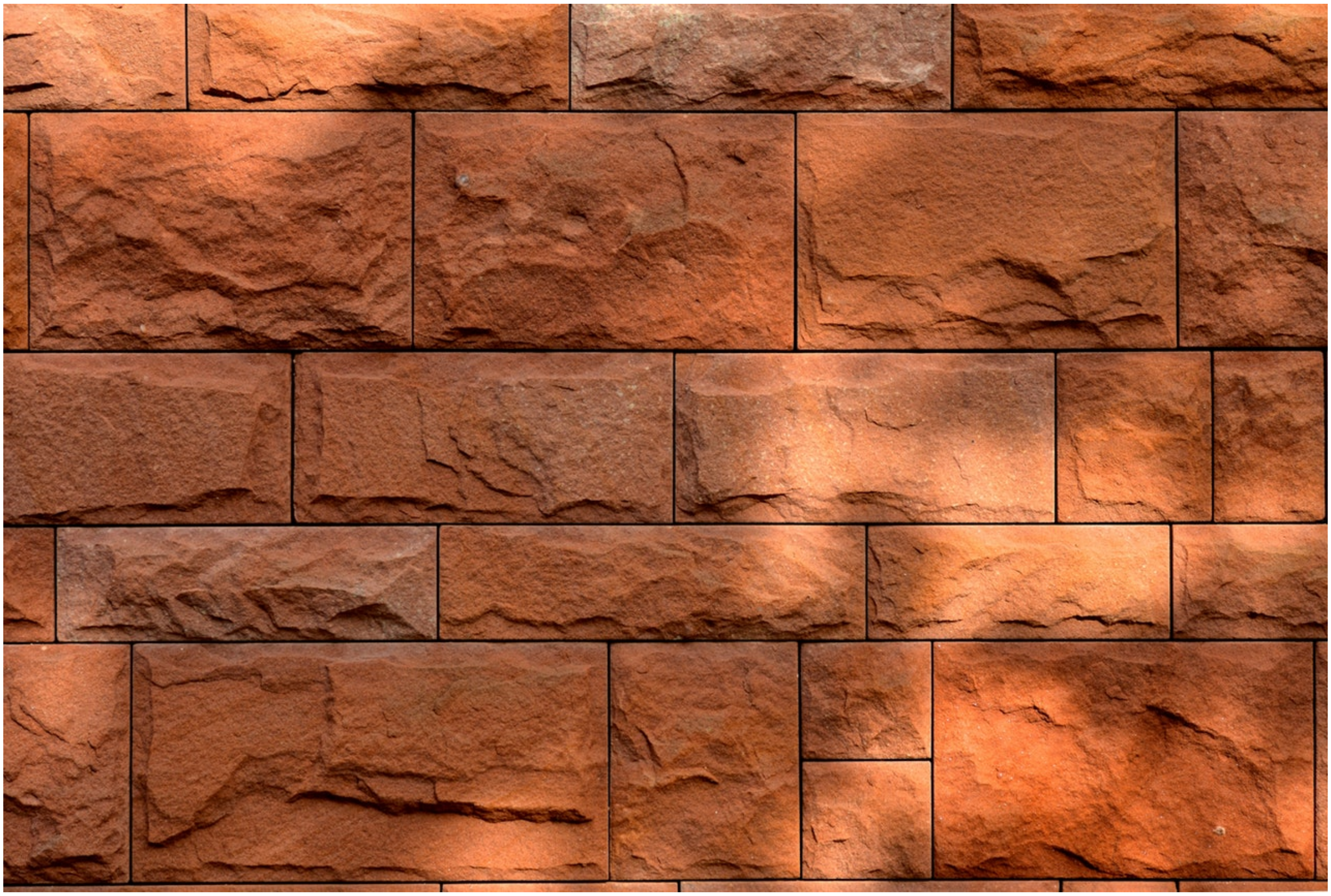
15讲理论一：对于单一职责原则，如何判定某个类的职责是否够“单一”



上几节课中，我们介绍了面向对象相关的知识。从今天起，我们开始学习一些经典的设计原则，其中包括，SOLID、KISS、YAGNI、DRY、LOD等。

这些设计原则，从字面上理解，都不难。你一看就感觉懂了，一看就感觉掌握了，但真的用到项目中的时候，你会发现，“看懂”和“会用”是两回事，而“用好”更是难上加难。从我之前的工作经历来看，很多同事因为对这些原则理解得不够透彻，导致在使用的时候过于教条主义，拿原则当真理，生搬硬套，适得其反。

所以，在接下来的讲解中，我不仅会讲解这些原则的定义，还会解释这些原则设计的初衷，能解决哪些问题，有哪些应用场景等，让你知其然知其所以然。在学习的时候，希望你能跟上我的思路，把握住重点，真正做到活学活用。

如何理解单一职责原则（SRP）？

文章的开头我们提到了SOLID原则，实际上，SOLID原则并非单纯的1个原则，而是由5个设计原则组成的，它们分别是：单一职责原则、开闭原则、里式替换原则、接口隔离原则和依赖反转原则，依次对应SOLID中的S、O、L、I、D这5个英文字母。我们今天要学习的是SOLID原则中的第一个原则：单一职责原则。

单一职责原则的英文是Single Responsibility Principle，缩写为SRP。这个原则的英文描述是这样的：A class or module should have a single reponsibility。如果我们把它翻译成中文，那就是：一个类或者模块只负责完成一个职责（或者功能）。

注意，这个原则描述的对象包含两个，一个是类（class），一个是模块（module）。关于这两个概念，在专栏中，有两种理解方式。一种理解是：把模块看作比类更加抽象的概念，类也可以看作模块。另一种理解是：把模块看作比类更加粗粒度的代码块，模块中包含多个类，多个类组成一个模块。

不管哪种理解方式，单一职责原则在应用到这两个描述对象的时候，道理都是相通的。为了方便你理解，接下来我只从“类”设计的角度，来讲解如何应用这个设计原则。对于“模块”来说，你可以自行引申。

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

我举一个例子来解释一下。比如，一个类里既包含订单的一些操作，又包含用户的一些操作。而订单和用户是两个独立的业务领域模型，我们将两个不相干的功能放到同一个类中，那就违反了单一职责原则。为了满足单一职责原则，我们需要将这个类拆分成两个粒度更细、功能更加单一的两个类：订单类和用户类。

如何判断类的职责是否足够单一？

从刚刚这个例子来看，单一职责原则看似不难应用。那是因为我举的这个例子比较极端，一眼就能看出订单和用户毫不相干。但大部分情况下，类里的方法是归为同一类功能，还是归为不相关的两类功能，并不是那么容易判定的。在真实的软件开发中，对于一个类是否职责单一的判定，是很难拿捏的。我举一个更加贴近实际的例子来给你解释一下。

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

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

对于这个问题，有两种不同的观点。一种观点是，UserInfo类包含的都是跟用户相关的信息，所有的属性和方法都隶属于用户这样一个业务模型，满足单一职责原则；另一种观点是，地址信息在UserInfo类中，所占的比重比较高，可以继续拆分成独立的UserAddress类，UserInfo只保留除Address之外的其他信息，拆分之后的两个类的职责更加单一。

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

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

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

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

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

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

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

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

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

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

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

类的职责是否设计得越单一越好？

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

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

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

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

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

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

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

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

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

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

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

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

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

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

重点回顾

今天的内容到此就讲完了。我们来一块总结回顾一下，你应该掌握的重点内容。

1.如何理解单一职责原则（SRP）？

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

2.如何判断类的职责是否足够单一？

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

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

3.类的职责是否设计得越单一越好？

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

课堂讨论

今天课堂讨论的话题有两个：

1. 对于如何判断一个类是否职责单一，如何判断代码行数过多，你还有哪些其他的方法吗？
2. 单一职责原则，除了应用到类的设计上，还能延伸到哪些其他设计方面吗？

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

blacknhole

在看文末的“3. 类的职责是否设计得越单一越好？”时，我惊喜地意识到：

1，内聚和耦合其实是对一个意思（即合在一块）从相反方向的两种阐述。

2，内聚是从功能相关来谈，主张高内聚。把功能高度相关的内容不必要地分离开，就降低了内聚性，成了低内聚。

3，耦合是从功能无关来谈，主张低耦合。把功能明显无关的内容随意地结合起来，就增加了耦合性，成了高耦合。

2019-12-06 12:19

编程界的小学生

1.方法就是全凭感觉。感觉不爽，就尝试着是否可以拆分多个类，感觉来了谁也挡不住。没有硬性要求吧，都是凭借经验。比如用户service可能包含用户的登录注册修改密码忘记密码等等，这些操作都需要验证邮箱，这时候你会发现这个类就很乱，就可以把他一分为二，弄个UserService再弄个UserEmailService专门处理用户相关邮件的操作逻辑，让UserService依赖Email的，等等这种，我觉得真的是全凭经验。换句话说，屎一样的代码写多了，写到自己看着都想吐的时候，经验就积累了。
2.方法设计上也用到了，比如自上而下的编程方式，先把核心方法定义好在去写具体细节，不要上来就把所有的细节都写到一个大而全的方法里。自上而下的编程方式他不香吗？

2019-12-06 00:18

辣么大

懂几个设计模式，只是花拳绣腿。掌握设计原则就才掌握了“道”。

设计你的系统，使得每个模块负责（响应）只满足一个业务功能需求。
Design your systems such that each module is responsible (responds to) the needs of just that one business function. (Robert C. Martin)

参考：https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

2019-12-06 09:27

Luciano李鑫

想请教一下争哥，关于代码代码持续重构的问题，所引出的额外测试、发布成本，和故障风险应该怎样平衡呢。

2019-12-06 11:06

作者回复

我推荐持续重构，不推荐等到代码烂到一定程度之后的大刀阔斧的重构。持续重构就像开发一样，是开发的一部分，所以也不存在额外的测试、发布成本之说，你就当成开发来看就行了。后面会讲到重构，你到时候再看下是否还有疑问。

2019-12-09 21:08

下雨天

回答问题

1. 类单一职责判断可以通过评估其对外提供接口是否满足不断变化的业务和需求来确定！问自己，该类是否对其他类是"黑盒"！

2. 类行数多=属性多+方法多
属性多: 要考虑这些属性是不是对类来说是必须的，需要移除么?
方法多: 方法间复用情况，方法间有没有写重复代码?
如上如果觉得没有可以改进的余地，就可以认为类行数恰当!

3. 单一职责还可以应用到方法，模块，功能点上!

2019-12-06 08:24

墨雨

我有个问题，就用户地址的设计来说，后续功能扩大再拆解是不是违反了开闭原则呢？而且后期拆分会比较影响现有业务逻辑吧，这个如何平衡呢?

2019-12-06 09:07

Chen

Android里面Activity过于臃肿会让感觉很头大，MVP,MVVM等框架都是为了让Activity变得职责单一。

2019-12-06 07:51

Dimple

因为学习设计模式，前几天刚和朋友在聊，说其实每个类的代码行数和函数的行数最好都需要控制下，能精简就精简，完成我们理解的重构。

刚好，今天就看到老师说的这个，赶紧分享给朋友，盛赞了这门课，哈哈

2019-12-06 11:06

啦啦啦

单一职责原则也可以用在服务上的拆分上

2019-12-06 07:46

荀麒睿

老师您好，有个问题想请教下，就是您举的UserInfo的例子，在抽取了地址相关的信息到新的类的时候，原来的userinfo类中需要再添加一个新的类的属性在里面么？感觉如果根据单一职责原则了话，新的类应该独立出来，UserInfo里应该不包含该类，那在这种情况下数据库的表一般会出现变化么？不然是否会造成一个新增的用户会在保存用户信息的时候对数据库进行两次操作？一次新增用户信息，然后获取了userId再进行一次操作修改地址相关信息？还是说在userinfo中存在新的地址相关类的属性，进行直接新增操作？因为没有这方面的实际经验，所以对这个比较疑惑。老师遇到这种情况一般作何处理

2020-01-05 22:26

作者回复

数据库跟业务代码的设计不是强耦合的。不然，对业务代码进行重构，那数据库还得跟着改，谁还敢重构啊。

不管userinfo是否有address的信息，我们都可以转化成数据库想要的数据格式，再一次性地写入到数据库中。

userinfo是否包含address的信息？理论上，既然已经拆出来了，职责单一了，就不必要包含了。

2020-01-08 21:24

小美

有个问题，比如有个OrderService中可能提供了各种订单查询、操作等，即一个OrderService有多个方法，是否符合SRP呢？

2019-12-09 20:50

作者回复

一个service当然可以有多个方法了，只要方法都是一个业务领域的，没有明显违背SRP，实际上都是合理的。

2019-12-09 21:13

李小四

设计模式_15

作业：

单一职责也可以用于方法的编写，在维护多年的项目中，我们会看到一些非常“庞大”的方法，这些方法的功能比它的命名丰富很多，它是一次次地改成了这个样子。本来是一个职责单一的简单方法，由于需求的变化，可能是方法被调用太多(不敢改)，也可能是框架设计(不能改)导致只能在方法内部添加特殊业务的判断条件，这样下来，这个方法就变得难以理解且难以维护。

感想

同事们也常讨论单一职责的边界, 始终没有一致的结论。今天的内容也坚定了观点，业务的发展一定程序决定了耦合的边界。我们学习前人总结的这些原则，目的是什么呢，我今天的感受是，系统性地降低工作量和出错概率。

- 降低工作量：我们要尽量保证，随着需求的增加，工作量的增加是线性的，而不是指数级的。据我了解，维护一些老代码的同学们，一直被产品同事质疑：就这么点功能，要做这么久吗？？？
- 降低出错概率：就像文中的序列化的例子，强行把序列化与反序列化方法拆开，会导致使用者需要花更多的时间来做一些同步的工作，如果文档不够清晰，或者阅读文档不够仔细，就会导致出错；有这样代码结构的系统，运行足够长的时间，一定会出更多的错误。

2019-12-07 16:43

黄林晴

打卡✔

安装ali规范插件，看到报警告的就按照规范修改，不过这个规范是死的，有时候和实际应用不同，不过大部分规范还是可以遵循的

2019-12-06 08:35

Jxin

回答问题：

1.不好说，职责单一这东西比较主观。得看自己对抽象出来的类的主观定义是什么。准的捏不住，但还是要把控一下范围的。

2.码出高效给出了方法行数不超过50行的一个基准标注。而我实践下来很难写出超过50行的方法，这50行还包括了大量注释。

3.方法的职责单一，业务领域的能力要单一（边界清晰）。

提问：

1.以前不代码规范不行，就逼着自己多思考，多写注释。现在养成了写注释的洁癖，不写就很难受。请问大佬，这怎么办，需要戒掉吗。我除了dao层的crud和数据类的setget外，其余方法都会带上注释。

2019-12-06 01:12

作者回复

哈哈，写注释不是挺好的吗？我后面讲到编程规范的时候会详细讲如何写注释的。

2019-12-09 21:22

刘学习来学习

前段时间需要对外提供sdk,最开始的设计就是根据职责定义了多个client对象供其他系统调用，后来角色不是很友好，最后还是提供了个聚合类，将所有的接口都集中到一起对外提供了,像这种情况,有的时候不知道该参考什么来设计

2020-01-07 23:33

作者回复

后面facade设计模式会讲到你的问题

2020-01-08 21:12

tuyu

小争哥, 数据库设计是不是不太适合设计那种抽象类的数据库表结构, 这样我写bo就会就会维护的很大很大

2020-01-01 23:23

作者回复

没太看懂你说的

2020-01-02 10:26

Smallfly

What

一个函数、类、模块只负责完成一个职责或者功能。

How

关于如何使用这个原则，不同的应用场景、需求背景、业务，对一个类职责是否单一，会有不同的判定结果。

这个说法其实很模糊，并没有多少实践上的指导意义。

定义中也存在令人迷惑的点，类由函数构成，模块由类构成（面向对象领域），如果每个函数职责单一，多个职责单一的功能函数组成一个类，那么这个类还算职责单一吗?

这里的区别在于看待职责单一的抽象维度不同，也称为职责颗粒度。

比如在函数的维度，在一个 getter 方法，只返回用户名，它是职责单一的；在用户类的维度，它还可以包含返回手机号的 getter 方法。

类也是有颗粒度的。以文中用户信息类为例，什么时候地址信息应该被作为单独类存在呢？我的结论是存在被复用的场景。

在初次设计时，可能并没有复用场景，那么都写在一个类里满足业务需求完全没问题。

但随着需求的发展，类中代码量膨胀，我们就需要根据复用性来对类进行拆分，此时单一职责原则可以作为一个指导思想。

过度使用单一职责

在设计时也不要过度使用单一职责原则，它会使得功能点不够内聚，增加维护成本。如果一个类，始终是作为整体被使用，即使它包含多种功能，放一起也可以。

当我们只想使用类的某一部分功能，又觉得其它功能鸡肋的时候，再考虑拆分也不迟。

Why

单一职责原则，是前辈在代码时间中的经验，将某一类经验抽象为这个名字。这个原则主要目的是为了，实现高内聚、低耦合、提高代码的复用性、可读性、可维护性。

2019-12-26 09:25

bboy孙晨杰

在数据库表设计的时候应该也有用到单一职责的思想吧，每个表只负责某一部分的业务数据，不要过多耦合，方便维护，阅读。

2019-12-10 13:51

赤城

项目初始阶段也是雄心勃勃，要把系统做出一个快速迭代、维护性高的系统，可是不断的需求变更导致开发任务过重，留给项

目整体的思考和重构时间被严重压缩，最终导致项目的技术管理失控，再加上人员变动等原因，项目死亡的概率急剧上升，都是惨痛的教训。

《三体》中常伟思的父亲经常说的是：要多想。

共勉！

2019-12-09 14:02

Cris

老师，我今天听到了一个概念叫面向切片编程(aop)，它和面向对象编程有什么联系呢？想听听老师的理解

2019-12-07 20:40

作者回复

这两个没有关系的，你可以看下spring的aop

2019-12-09 20:59