15-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

www.ixuexi.cf 分享站 用户标记 zyceson

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) {
```

www.ixuexi.cf 分享站 用户标记 zyceson

```
    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：

- 编程界的小学生 2019-12-06 00:18:08
  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UserEmailServic

www.ixuexi.cf 分享站 用户标记 zyceson

e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？ [7赞]

- Chen 2019-12-06 07:51:18
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。 [4赞]

- 辣么大 2019-12-06 09:27:20
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 [2赞]

- 潇潇雨歇 2019-12-06 11:30:02
1、如何判断代码行数过多，这个我还是觉得要看感觉，如果自己看着觉得很不爽，那说明应该要拆分或者优化精简了；如果自己看不太出来可以找人review，别人或许能看出来。不过代码行数最好不要太多。
3、方法设计，接口设计等等，都可以使用单一职责，每个方法做一件事，如果里面的代码多了或者乱了，就应该试着拆分，接口设计也是同理，做某一个模块的事情，不能多个模块都写在一个接口里面。 [1赞]

- 墨雨 2019-12-06 09:07:40
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？ [1赞]

- 黄林晴 2019-12-06 08:35:09
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 [1赞]

- 下雨天 2019-12-06 08:24:14
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！ [1赞]

- shniu 2019-12-06 11:23:23
1. 仅通过代码行数来判断是否是SRP，太武断；好的做法还是应该把功能放在实际的需求场景、业务场景、成本、紧急程度等众多因素中，去考量引起变化的因素是不是有多个，如果存在多个，就尽量做拆分，

要做到：接口一定要SRP，实现类的设计尽量做到只有一个原因引起变化。这个度确实很难掌握，有可能会拆分过细，带来复杂性，增加心智负担
2. SPR 除了应用在类的设计上，还可以应用在方法的划分上，每个方法（函数）也尽量做到职责明确，这样有助于代码维护和可读性；此外，还可以应用在系统设计上，比如划分子系统，子系统划分模块，都尽量做到职责明确，最好一件事情，其实微服务也能算是一种 SRP 的实践。

- Dimple 2019-12-06 11:06:29
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

- Luciano李鑫 2019-12-06 11:06:09
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。

- qqq 2019-12-06 10:11:15
1. 想到一个: 这个类是不是能复用，如果可以，那方法和属性的覆盖率是不是很高？还是只用其中几个，就可以判断是否要继续拆分了
2. 函数的设计、组件划分、React 的 hook

- 阿西吧 2019-12-06 10:02:51
可以通过代码是否有“坏”味道来知道类是不是设计的不好，不满足单一职责。品味很重要:)

- moqifei 2019-12-06 09:58:48
单一职责原则，除了应用到类的设计上，还能延伸系统群设计上，多个子系统交互时，特别容易忽略这个原则。

- evolution 2019-12-06 09:54:42
记录一下个人粗浅的认知：
职责单一：其实就是对某个事物的增删改查。
例如上述用户信息例子，当有电商业务时，地址就需要分离出来，单独的对地址增删改查。
实战二中的接口鉴权，其实就是客户端token的产生（新增，AuthToken类），url的产生（新增，Url类），服务端token的产生（新增，CredentialStorage类），比对结果的产生（两个token比对，DefaultApiAuthencatorImpl类）

- Peter Cheng 2019-12-06 09:33:04
对于第一个问题，我认为只要定义合理的属性，就能看出这个类是否单一，因为类方法都是对属性的操作。第二个问题，作为一个前端，我们通过代码校验工具eslint强制所有的组件类最多300行代码，否则代码就无法提交，强制约束。

- liu_liu 2019-12-06 09:22:47
1. 通过大致看类中的方法与属性，是否和本身提供的功能相关。
2. 还可以用于接口，方法

- whistleman 2019-12-06 09:06:25
根据不同的场景对某个类或模块单一职责的判断是不同的，不能为了拆分而拆分，造成过度设计，难以维护。

www.ixuexi.cf 分享站 用户标记 zyceson

- 峰峰 2019-12-06 09:04:54
  除了在类上使用单一职责原则外，也可以应用在架构设计上。例如，数据库虽然支持像触发器或者存储过程用来处理业务数据，但是数据库最擅长的还是数据的存取；再者，队列的使用，redis虽然可以实现队列的效果，不过redis出色的功能还是缓存的使用，可以使用市面上比较流行的消息队列组件，其更有丰富的特性。因此，还是让专业的软件做专业的事比较好。

- DullBird 2019-12-06 08:58:38
  1.判断一个类是否职责单一,一般是根据方法定义，查看其实现的功能和该模块是否一致，如果疑惑，就是可能职责不单一情况。
  2. 代码行数的话，一般就是读到头疼就不合适了。。
  3. 单一职责还可以应用到微服务的拆分，和子模块的设计。。
  提一个问题:
  现在我有一个虚拟组织的抽象类，子类有班级组织和部门组织，虚拟组织提供了更新组织的功能，
  外部只需要和虚拟组织类打交道，告诉它更新一个组织
  (前端界面知道到底是传递部门组织信息还是班级组织信息，会封装更不同的对象)，虚拟组织内部通过传递的不同对象，判断是更新班级还是部门，调用对应的子类。
  但是现在增加了一个功能，是更新班级的某个状态的时候，在班级内的所有人都要一起更新人的状态。
  问题是：如果这个方法放到组织内，那么组织就同时操作了人的信息和组织的更新，不符合职责单一，如果放到虚拟组织外层，外层调用又需要理解组织内部的细节，去主动判断到底是更新组织还是班级，如果是班级就触发这一段逻辑。

- Rangers_Master 2019-12-06 08:56:22
  我的理解是：
  单一职责一般跟业务挂钩。比如设计一个查询个人信息的类，就不要引入查询订单的代码或者是跟个人信息不相关的代码等等。
  由此想到：这个跟设计通用的工具类还有一定的区别。设计通用的工具类也是可以满足单一职责：可以细分成屏幕相关的，设备相关的，权限相关的，等等。