

大家好，从本小节开始我们一起交流常用的设计模式。设计模式是一种思想，并不是一门具体的技术，没有很多的工作积累是不可能真正理解设计模式的。本专刊中的设计模式章节，我们重点阐述面试中常见的设计模式的原理与使用。

设计模式就是在软件开发过程中所总结形成的一系列准则。当我们遇到一些场景的时候，使用恰当的设计模式可以使软件设计更加健壮，增强程序的扩展性。在讲解面试中常考的设计模式之前，我们先来看看设计模式的六大原则吧。

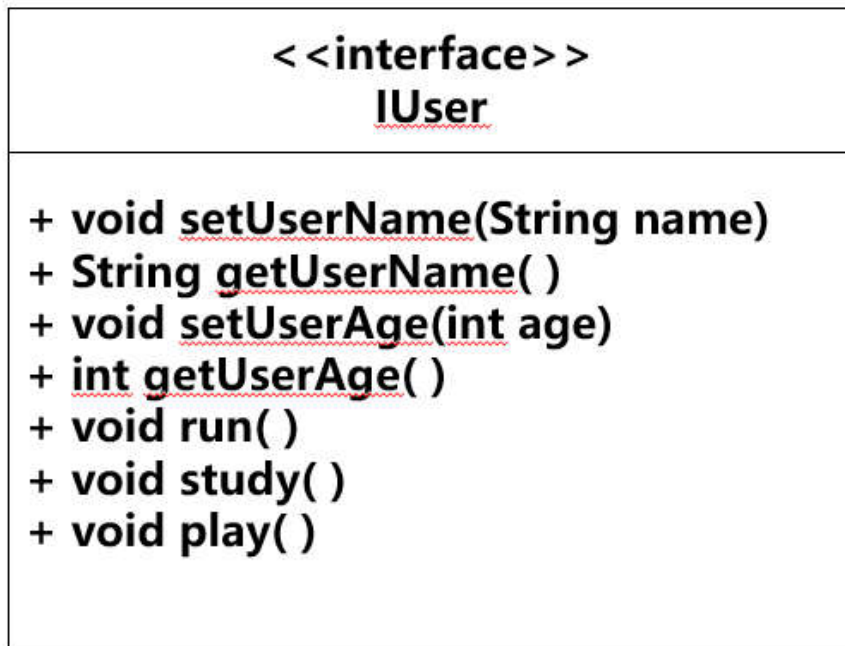
设计模式的六大原则如下：

- 单一职责原则
- 里氏替换原则
- 依赖倒置原则
- 接口隔离原则
- 迪米特法则
- 开闭原则

单一职责原则（Single responsibility principle，SRP）：

单一职责规定了一个类应该只有一个发生变化的原因。如果一个类承担了多个职责，则会导致多个职责耦合在一起。但部分职责发生变化的时候，可能会导致其余职责跟着受到影响，也就是说我们的程序耦合性太强，不利于变化。

举个例子，我们来看下边的一个接口IUser的类图：

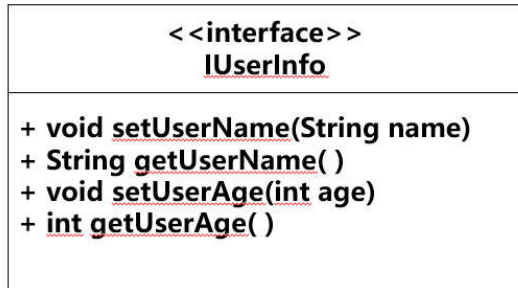


IUser的类图

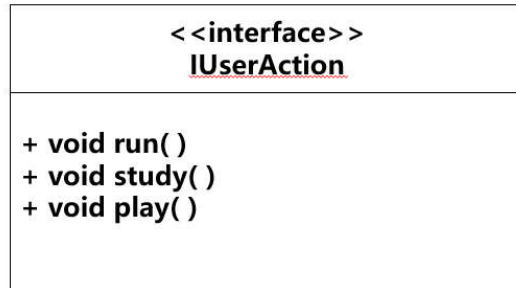
IUser类中拥有设置和获取用户名字和年龄的方法，还有指挥该User去跑步，学习以及玩耍的功能。那么，IUser类其实就是违反了单一职责原则。我们可以将其拥有到职责进行划分，一类是用户User的固有属性，另一类是用户User拥有的实际能力。固有属性包括设置和获取名字和年龄，实际能力则包括User具有的跑步，学习以及玩耍功能。

所以，我们需要将该IUser类进行划分，分为多个接口，划分之后的类图如下所示：

5



IUserInfo的类图



IUserAction的类图

牛客@我是祖国的花朵

那么，单一职责有哪些优点呢？

- 降低了类的复杂度，每一个类都有清晰明确的职责。
- 程序的可读性和可维护性都得到了提高。
- 降低业务逻辑变化导致的风险，一个接口的修改只对相应的实现类有影响，对其他接口无影响。

里氏替换原则（Liskov Substitution Principle, LSP）：

里氏替换是指所有父类可以出现的地方，子类就都可以出现，使用子类来替换父类，调用方不需要关心目前传递的父类还是子类。

在前面Java基础章节中，我们介绍了继承。通过继承，实现了代码的共享和复用。但是继承是强侵入性的，子类必须拥有父类所有的非私有属性和方法，在一定程度上降低了子类的灵活性。

通过里氏替换原则，我们可以将子类对象做为父类对象来使用，屏蔽了不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。里氏替换之后，父类的对象就可以根据当前赋值给它的子类对象的特性以不同的方式运作。

接下来，我们一起看一个Demo：

```
1 package niuke;
2
3 public class LiSiTest {
4
5     public static void main(String[] args) {
6         // 通过传入子类对象来替换父类
7         eat(new Dog());
8
9         eat(new Cat());
10    }
11
12    private static void eat(Animals animal){
13        animal.eat();
14    }
15 }
16
17 abstract class Animals{
18     abstract void eat();
19 }
```

```

20  class Dog extends Animals{
21
22      @Override
23      void eat() {
24          System.out.println("我是小狗，喜欢吃大肉");
25      }
26  }
27
28  class Cat extends Animals{
29
30      @Override
31      void eat() {
32          System.out.println("我是小猫，喜欢吃小鱼干");
33      }
34  }

```

5

那么，里氏替换原则有哪些优点呢？

里氏替换原则可以增强程序的健壮性，子类可以任意增加和缩减，我们都不需要修改接口参数。在实际开发中，实现了传递不同的子类来完成不同的业务逻辑。

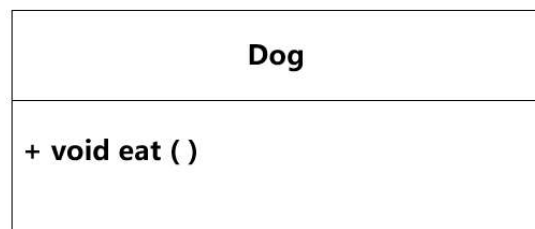
依赖倒置原则（Dependence Inversion Principle, DIP）：

依赖倒置原则是指高层模块不应该依赖于底层模块，抽象不应该依赖细节，细节应该依赖抽象。在Java中，接口和抽象类都是抽象，而其实现类就是细节。也就是说，我们应该做到面向接口编程，而非面向实现编程。

我们来看一个案例，讲述饲养员喂养的故事。初始类图如下：



Feeder的类图



Dog的类图

👉 牛客@我是祖国的花朵

开始的时候，饲养员只需要喂养小狗即可。代码实现如下：

```

1  package niuke;
2
3  public class DIPTest {
4      public static void main(String[] args) {
5          Dog dog = new Dog();
6          Feeder feeder = new Feeder();
7          // 饲养员喂食小狗
8          feeder.feed(dog);
9      }
10 }
11
12 // 定义小狗类
13 class Dog{
14     public void eat(){
15         System.out.println("小狗在吃东西,,,,,");
16     }
17 }
18 // 定义饲养员类
19 class Feeder{
20     public void feed(Dog dog){

```

```

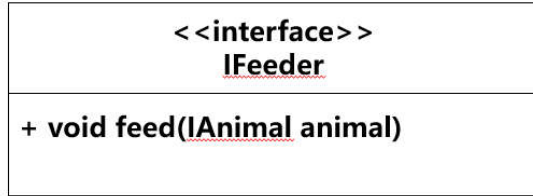
21         dog.eat();
22     }
23 }

```

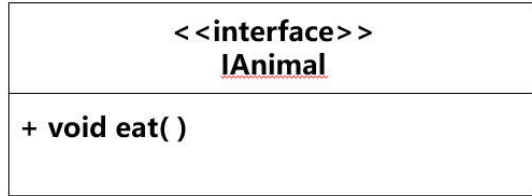
这是一个面向实现编程的案例，可以看到，我们的上层模块依赖了下层模块，并没有做到面向接口编程。当饲养员想要喂食其余动物的时候，发现其并不具备相应的能力，因为feed方法中的参数是一个具体的实现，是一个细节。

5

然后，我们再来看下，如何做到依赖倒置，实现面向接口编程。



IFeeder的类图



IAnimal的类图

牛客@我是祖国的花朵

代码实现如下：

```

1  package niuke.mode;
2
3  public class DIPTest {
4      public static void main(String[] args) {
5          Dog dog = new Dog();
6          Cat cat = new Cat();
7          Feeder1 zhangsan = new Feeder1();
8          // 饲养员zhangsan喂食小狗，小狗
9          zhangsan.feed(dog);
10         zhangsan.feed(cat);
11         // -----
12         // 接下来是另一位饲养员lisi，它们的工作方式不同
13         Feeder2 lisi = new Feeder2();
14         // 饲养员zhangsan喂食小狗，小狗
15         lisi.feed(dog);
16         lisi.feed(cat);
17     }
18 }
19
20 // 定义一个抽象
21 interface IAnimal{
22     void eat();
23 }
24
25 // 定义实现，小狗类
26 class Dog implements IAnimal{
27     public void eat(){
28         System.out.println("小狗在吃东西,,,,,"");
29     }
30 }
31
32 // 定义实现，小猫类
33 class Cat implements IAnimal{
34     public void eat(){
35         System.out.println("小猫咪在吃东西,,,,,"");
36     }
37 }
38
39 // 定义饲养员接口
40 interface IFeeder{
41     public void feed(IAnimal animal);
42 }
43
44 // 定义1号饲养员

```

```
45 class Feeder1 implements IFeeder{
46
47     @Override
48     public void feed(IAnimal animal) {
49         animal.eat();
50     }
51 }
52
53 // 定义2号饲养员
54 class Feeder2 implements IFeeder{
55
56     @Override
57     public void feed(IAnimal animal) {
58         this.eat();
59         animal.eat();
60     }
61     public void eat(){
62         System.out.println("工作之前，我的先填饱肚子");
63     }
64 }
```

5

面向接口编程之后，我们可以定义多个拥有不同工作方式的饲养员，并且每个饲养员都可以喂养多个动物，只要该动物实现IAnimal接口即可。

那么，依赖导致原则的好处有哪些呢？

- 依赖倒置通过抽象(接口或抽象类)使各个类或模块的独立，实现模块间的松耦合。
- 面向接口编程可以使得当需求变化的时候，程序改动的工作量不至于太大。

接口隔离原则(Interface Segregation Principle, ISP):

接口隔离原则是指客户端不应该依赖它不需要的接口，一个类对另一个类的依赖应该建立在最小的接口上。

接口隔离原则的使用原则：

- 根据接口隔离原则拆分接口时，首先必须满足单一职责原则。
- 接口需要高内聚，提高接口，类和模块的处理能力，减少对外的交互。
- 定制服务，单独为一个个体提供优良服务(只提供访问者需要的方法)。
- 接口设计要有限度，接口设计的太小，容易造成开发难度增加或者可维护性降低。

迪米特法则(Law of Demeter, LoD):

迪米特法则也叫最少知识原则，是指一个对象应该对其依赖的对象有最少的了解。该类不需要知道其依赖类的具体实现，只需要依赖类给其提供一个公开对外的public方法即可，其余一概不需要了解。

迪米特法则的核心就是解耦合，减弱类间的各个耦合，提高类的复用率。

开闭原则 (Open Close Principle, OCP) :

开闭原则是指一个软件实体如类，模块和函数应该对扩展开放，对修改关闭。也就是说，通过开闭原则，我们可以通过扩展行为来实现新的功能，而不是通过修改已有的代码。开闭原则可以帮助我们构建一个稳定，灵活的软件系统。

六大设计原则是我们在进行程序设计和软件开发过程中应该去重点参考和遵循的准则。但是，鉴于我们复杂的业务逻辑场景以及多变的业务需求，往往不可能做到遵循全部的准则。所以，六大设计原则只是一个参考，具体设计中需要灵活使用各个原则，争取设计出优雅的软件架构。

接下来，我们对六大设计原则做一个简单的总结。

- **单一职责原则**：类或者接口要实现职责单一
- **里氏替换原则**：使用子类来替换父类，做出通用的编程
- **依赖倒置原则**：面向接口编程
- **接口隔离原则**：接口的设计需要精简单一
- **迪米特法则**：降低依赖之间耦合
- **开闭原则**：对扩展开放，对修改关闭

5

介绍了六大设计原则之后，我们来一起看看常见的设计模式吧。在面试中，常见考察的设计模式包括单例模式，工厂方法模式，模式和模板方法模式等。本小节中，我们先来较为详细的阐述面试中**最高频考察的设计模式-单例模式**吧。

单例模式：

单例模式是指在一个系统中，一个类有且只有一个对象实例。

单例模式的实现：单例模式从创建方式上又分为饿汉式和懒汉式两种。

饿汉式的单例模式实现如下：

```
1  /**
2   * 饿汉式的单例模式
3   * @author ywq
4   */
5  class Single{
6      private static final Single s = new Single();
7      private Single(){}
8      public static Single getInstance(){
9          return s;
10     }
11 }
```

饿汉式的单例模式在程序初始化的时候即创建了对象，在需要的时候可以直接返回该对象实例。

懒汉式的单例模式实现如下：

```
1  /**
2   * 懒汉式的单例模式
3   * @author ywq
4   */
5  class Single{
6      private static Single s = null;
7      private Single(){}
8      public static Single getInstance(){
9          if(null==s)
10             s = new Single();
11         return s;
12     }
13 }
```

懒汉式的单例模式是在真正需要使用到的时候，才会去创建实例对象。

那么，聪明的你看到这里肯定会问：“如上实现的单例模式不是线程安全的吧？”

答：没错，上边实现的是最简单的单例模式，只适合于单线程环境下使用。因为多个线程并发环境下，还是会创建多个实例对象。

我们可以通过加入synchronized内部锁来解决多线程环境下的线程安全问题，代码实现如下所示：

5

```
1  /**
2   * 多线程环境下的懒汉式单例模式(DCL，双检锁实现)
3   * @author ywq
4   */
5  class Single{
6      private static Single s = null;
7      private Single(){}
8
9      public static Single getInstance(){
10         if(null==s){
11             synchronized(Single.class){
12                 if(null==s)
13                     s = new Single();
14             }
15         }
16         return s;
17     }
18 }
```

一般情况，我们上边加入synchronized的单例模式实现已经是比较准确的了。但是，在前面章节的学习中，我们学习了指令的重排序相关的知识点，忘记的同学可以返回去，看看第9小节的内容。这里我们再次简单阐述：

Instance instance = new Instance()都发生了啥？

具体步骤如下三步所示：

- 在堆内存上分配对象的内存空间
- 在堆内存上初始化对象
- 设置instance指向刚分配的内存地址

第二步和第三步可能会发生重排序，导致引用型变量指向了一个不为null但是也不完整的对象。所以，在多线程下上述的代码会返回一个不完整的对象。根据前面章节所学的内容，我们需要加入一个**volatile**关键字来禁止指令重排序。

完整的多线程环境下的单例模式的实现代码如下（面试必会）：

```
1  /**
2   * 多线程环境下的懒汉式单例模式(DCL，双检锁+volatile实现)
3   * 加入了volatile变量来禁止指令重排序
4   * @author ywq
5   */
6  class Single{
7      private static volatile Single s = null;
8      private Single(){}
9
10     public static Single getInstance(){
11         if(null==s){
12             synchronized(Single.class){
13                 if(null==s)
14                     s = new Single();
15             }
16         }
17         return s;
18     }
19 }
```

以上介绍的是最常见的单例模式的实现方式，那么除了上边所述，大家还知道哪些单例模式的实现方式呢？欢迎大家在评论区交流~

单例模式的优点：单例模式保证了一个类在一个系统中有且只有一个对象实例，减少了系统内存和性能的开销。

单例模式的使用场景：创建一个对象需要消耗太多的资源或者在一个系统中不适合创建多个对象实例的情况下，我们可以采用单例模式设计实现。

总结：

在本小节中，我们主要是对设计模式的六大原则进行了简单的阐述，六大原则的使用可以使我们设计出高效通用并且易于维护的程序架构。接着，我们阐述了单例模式相关实现方式。单例模式是为数不多的在面试阶段需要手写的设计模式。

当面试官要求我们**手写单例模式的时候，我们需要直接给出使用volatile+DCL实现的完整版本的单例模式实现，并且可以对其中的含义做出准确的解释。**在下一小节中，我们会接着交流学习面试中常见的设计模式知识点。

限于作者水平，文章中难免会有不妥之处。大家在学习过程中遇到我没有表达清楚或者表述有误的地方，欢迎随时在文章下边指出，我会及时关注，随时改正。另外，大家有任何话题都可以在下边留言，我们一起交流探讨。

讨论

评论

柳杰201905011049420

1#

叮🔔

发表于 2020-01-14 16:47:51

赞(0) 回复(0)


柳杰201905011049420

2#

Volatile 现在感觉不常用了吧

发表于 2020-01-29 13:38:28

赞(1) 回复(4)

我是祖国的花朵  作者： 必须用呀，经典单例模式实现中必须使用volatile关键字的

2020-02-20 17:59:24

赞(0) 回复(0)

梁凉不凉： 了解一下JVM，就知道Volatile在单例时候必须使用了。

2020-06-16 11:35:26

赞(0) 回复(0)

梁凉不凉： 多线程重排序问题

2020-06-16 11:51:54

赞(0) 回复(0)

LurnaGao： 请问synchronized不是已经禁止指令重排序了吗？

今天 09:55:18

赞(0) 回复(0)

请输入你的观点

回复



双重检测,静态内部类,枚举,实现单例模式

3#

发表于 2020-02-17 12:21:47

赞(0) 回复(1)

我是祖国的花朵 (作者) : 赞, 其实我觉得静态内部类实现的单例模式很优雅。

2020-02-20 17:59:51

赞(0) 回复(0)

5

回复



牧水s

4#

打卡, 通过枚举和cas可以实现无锁化创建单例

发表于 2020-03-08 20:24:52

赞(0) 回复(1)

我是祖国的花朵 (作者) : 可以关注下静态内部类实现方式, 很优雅

2020-03-08 20:26:33

赞(0) 回复(0)

回复



现在的菜鸡, 未来的顶尖技术总监

5#

楼主你好, 我想问一下单例模式的中为啥要判断两次 `null == s` ?

发表于 2020-05-29 12:20:13

赞(0) 回复(2)

oneCODEOR : 我的理解是这样的, 两个线程都走到了synchronized的时候, 一个线程拿到锁进去实例对象后释放锁, 这个时候另外一个线程就可以拿到锁了, 第二个线程不判空的话就会又new一个对象出来了

2020-06-04 10:00:44

赞(0) 回复(0)

现在的菜鸡, 未来的顶尖技... 回复 oneCODEOR : 我觉得是如果只有一个判断的话, 当一个线程被锁住后, 另一个线程执行到synchronized块的时候会一直等待那个线程释放锁, 从而不继续进行, 而添加了外层的判断后, 就直接判断不会等待了

2020-06-04 10:12:14

赞(0) 回复(0)

回复