设计模式(一)

铨融医药科技有限公司　　张阳

为什么？

- 提高复杂代码的设计和开发能力
  - 如何分层、分模块?应该怎么划分类?每个类应该具有哪些属性、方法?怎么设计类之间的 交互?该用继承还是组合?该使用接口还是抽象类?怎样做到解耦、高内聚低耦合?该用单 例模式还是静态方法?用工厂模式创建对象还是直接 new 出来?如何避免引入设计模式提 高扩展性的同时带来的降低可读性问题?

- 让读源码、学框架事半功倍
  - 优秀的开源项目、框架、中间件，代码量、类的个数都会比较多，类结构、类之间的关系极其复杂，常常调用来调用去。代码中会使用到很多设计模式、设计原则或者设计思想。如果你不懂这些设计模式、原则、思想，在看代码的时候，你可能就会琢磨不透作者的设计思路，对于一些很明显的设计思路，你可能要花费很多时间才能参悟。

为什么?

- 应对面试中的设计模式相关问题
- 告别写被人吐槽的烂代码
- 为你的职场发展做铺垫，希望在职场有更高的成就、更好的发展，那就要 重视基本功的训练、基础知识的积累。

评判代码质量好坏的维度

- 可维护性
- 可读性
- 可扩展性
- 灵活性
- 简洁性
- 可复用性
- 可测试性

评判代码质量好坏的维度

- 可维护性
  - 在不破坏原有代码设计、不引入新的bug的情况下，能够快速修改或者添加代码
- 可读性
  - 代码是否符合编码规范、命名是否达意、注释是否详尽、函数是否长短合适、模块划分是否清晰、是否符合高内聚低耦合等等
- 可扩展性
  - 我们在不修改或少量修改原有代码的情况下，通过扩展的方式添加新的功能代码

评判代码质量好坏的维度

- 灵活性
  - 当我们添加一个新的功能代码的时候，原有的代码已经预留好了扩展点，我们不需要修改原有的代码，只要在扩展点上添加新的代码即可。这个时候，我们除了可以说代码易扩展，还可以说代码写得好灵活。
  - 当我们要实现一个功能的时候，发现原有代码中，已经抽象出了很多底层可以复用的模块、类等代码，我们可以拿来直接使用。这个时候，我们除了可以说代码易复用之外，还可以说代码写得好灵活。
  - 当我们使用某组接口的时候，如果这组接口可以应对各种使用场景，满足各种不同的需求，我们除了可以说接口易用之外，还可以说这个接口设计得好灵活或者代码写得好灵活。

评判代码质量好坏的维度

- 简洁性
  - 代码简单、逻辑清晰
- 可复用性
  - 尽量减少重复代码的编写，复用已有的代码
- 可测试性
  - 代码可测试性的好坏，能从侧面上非常准确地反应代码质量的好坏

代码设计原则-SOLID

- 单一职责 - Single Responsibility Principle
  - 一个类或者模块只负责完成一个职责（或功能）
- 开闭原则 - Open Closed Principle
  - 对修改关闭，对扩展开放
- 里式替换原则 - Liskov Substitution Principle
  - 子类对象可以替换父类对象出现的任何地方，并保证原来程序的行为逻辑和正确性不被破坏
- 接口隔离原则 - Interface Segregation Principle
  - 客户端不需要依赖它不需要的接口
- 依赖倒置原则 -Dependency Inversion Principle
  - 高层模块不依赖于底层模块，而要通过抽象来相互依赖。
  - 抽象不依赖具体实现细节，具体实现细节依赖抽象

代码设计原则

- DRY 原则、KISS 原则、YAGNI 原则、LOD 法则

设计模式分类

设计模式

创建型

常用：

- 单例模式
- 工厂模式（工厂方法和抽象工厂）
- 建造者模式

不常用：

- 原型模式

结构型

常用：

- 代理模式
- 桥接模式
- 装饰者模式
- 适配器模式

不常用：

- 门面模式
- 组合模式
- 享元模式

行为型

常用：

- 观察者模式
- 模板模式
- 策略模式
- 职责链模式
- 迭代器模式
- 状态模式

不常用：

- 访问者模式
- 备忘录模式
- 命令模式
- 解释器模式
- 中介模式

单例模式

- 定义
  - 一个类只允许创建一个对象
- 用处
  - 在系统中只应保存一份的数据，解决资源访问冲突

单例模式实现方式

- 饿汉式
- 懒汉式
- 双重检测
- 静态内部类
- 枚举

单例模式实现

```java
public class HungarySingleton {
    private static final HungarySingleton instance = new HungarySingleton();

    private HungarySingleton() {}

    public static HungarySingleton getInstance() {
        return instance;
    }

}
```

```java
public class LazySingleton {
    private static LazySingleton instance;

    private LazySingleton() {}

    public static synchronized LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

```java
public enum EnumSingleton {
    INSTANCE;

    private AtomicLong id = new AtomicLong( initialValue: 0);
    public long getId(){
        return id.incrementAndGet();
    }
}
```

```java
public class DoubleCheckSingleton {

    private static DoubleCheckSingleton instance;

    public static DoubleCheckSingleton getInstance() {
        if (instance == null) {
            synchronized (DoubleCheckSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckSingleton();
                }
            }
        }
        return instance;
    }
}
```

```java
public class StaticInnerIdGenerator {

    private StaticInnerIdGenerator() {
    }

    public static StaticInnerIdGenerator getInstance() { return SingletonHolder.INSTANCE; }

    private static class SingletonHolder {
        private static final StaticInnerIdGenerator INSTANCE = new StaticInnerIdGenerator();
    }
}
```

工厂模式

- 分类
  - 简单工厂
    - 将类的创建剥离到一个类中
  - 工厂模式
    - 用于定义一个创建产品实例的工厂接口，将实际创建工作推迟到子类中
  - 抽象工厂
    - 抽象工厂创建多个工厂类，当需要新增功能时，直接新增工厂实现类，不需要修改之前的代码

工厂模式

- 试题
  - 根据配置文件的后缀，选择不同的解析器

建造者模式

- 作用
  - 创建类型复杂的对象，通过设置不同的可选参数，“定制化”地创建不同的对象

建造者模式

- 试题
  - 定义一个资源配置类ResourcePoolConfig.有以下几个成员变量，也就是可配置项，编写这个ResourcePoolConfig类

成员变量	解释	是否必填?	默认值
name	资源名称	是	没有
maxTotal	最大总资源数量	否	8
maxIdle	最大空闲资源数量	否	8
minIdle	最小空闲资源数量	否	0

思考!

- 刚刚讲到，name是必填的，如果也通过set()方法设置，那校验逻辑无处安放
- 如果各配置项关系有一定的依赖关系，校验逻辑无处安放
- 如果ResourcePoolConfig类对象是不可变对象，在创建好之后，就不能修改内部的值，也就是不能再ResourcePoolConfig类中暴露Set()方法，应该如何设计

总结

- 单例模式用来创建全局唯一的对象。
- 工厂模式用来创建不同但是相关类型的对象(继承同一父类或者接口的一组子类)
- 由给定的参数来决定创建哪种类型的对象。建造者模式是用来创建复杂对象，可以通过设置不同的可选参数，"定制化"地创建不同的对象。

Thinks !