# 单例（概述）

参考：

1. 主：<https://blog.csdn.net/qq_25333681/article/details/93662660>
2. 辅：

<https://blog.csdn.net/mnb65482/article/details/80458571>

<https://blog.csdn.net/jq_ak47/article/details/54894793>

## 1.1 概述

按照惯有的分类方式，设计模式共分3大类：创建型，结构型，行为型

单例是创建型设计模式的一种，它确保某一个类在系统中只有一个实例，并自行实例化，同时向外部提供获取这个唯一实例的接口。从而，单例有三大特性：

1. 单例类只有一个实例
2. 单例类必须自己实例化自己
3. 单例类需要向外提供实例

下面分析5种经典写法，逐步分析成因和背后原理：

## 1.2 饿汉式

### （1） 写法

|  |
| --- |
| public class EagerSingleton {   // 静态变量，类在创建之初就会执行实例化动作  private static EagerSingleton *sInstance* = new EagerSingleton();  // 私有化构造函数，使外界无法创建实例  private EagerSingleton() {  }  // 为外界提供获取实例的接口  public static EagerSingleton getInstance() {  return *sInstance*;  }   public void show() {  System.*out*.println("饿汉式");  } } |

上面是饿汉式单例模式的标准代码，所谓的“饿汉式”只是形象的比喻：

EagerSingleton类的实例因为变量sInstance声明为static的关系，在类加载过程中便会执行（类加载时会对静态变量分配内存，设置初始值）。由此带来的好处是Java的类加载机制本身为我们保证了实例化过程的线程安全性，缺点是这种空间换时间的方式，即使类实例本身还未用到，实例也会被创建。

### （2） 缺陷

饿汉式的缺点有：

1. 空间使用率不高：在需要实例之前类加载时就完成了初始化，在系统中单例场景较多的情况下，会造成内存占用，加载速度慢问题。
2. 类加载时实例化，意味着该类无法在程序运行过程中通过运行参数实例化，代码失去灵活性。

饿汉式在如今硬件设备条件下，缺点的存在其实关系不大，对空间不是特别严苛的，且用不到初始化参数的应用来说，非常建议使用这种方式。

## 1.3 懒汉式

### （1） 是什么

“懒汉式”是针对饿汉式单例模式缺点而生的懒加载模式，所谓懒加载的意思是，只有当需要使用实例的时候才去实例化。

### （2） 写法

#### 写法1：线程不安全

示例代码

|  |
| --- |
| public class LazySingleton {   private static LazySingleton *sInstance*;   private LazySingleton() {   }   public static LazySingleton getInstance() {  if (*sInstance* == null) {  *sInstance* = new LazySingleton(); //懒加载  }  return *sInstance*;  }   public void show() {  System.*out*.println("懒汉式");  } } |

饿汉式和懒汉式的区别在于，饿汉式在类加载时便被实例化，而懒汉式是在getInstance()函数调用时，当instance==null时，才去实例化，否则直接返回实例。

##### 缺陷：

但这里有个问题，单例模式的核心是系统中只存在一个单例类的实例，这其实隐含了实例只创建一次的意思。但上述LazySingleton类只能保证在单线程中只创建一次，在多线程中却不能保证。

如果有两个线程，Thread1、Thread2，两个线程先后调用getInstance()函数。若Thread1的调用，执行到if(instance==null)的语句块被中断，此时instance的值还未改变，Thread2也执行到了这里被中断，可以预见，两个线程都将分别创建一个LazySingleton实例【即多个线程可能同时检测到实例没有创建而分别创建】，从而破坏单例约束。

这个缺点的原因，涉及到并发编程的原子性【即一个操作或者多个操作，要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行】。示例中，创建实例的代码逻辑失去了原子性从而导致可能存在多个实例创建的情况。

#### 写法2：线程安全但效率低下

综上，给示例代码加上原子性即可：

|  |
| --- |
| public class LazySingleton {   private static LazySingleton *sInstance*;   private LazySingleton() {   }   public static synchronized LazySingleton getInstance() {  if (*sInstance* == null) {  *sInstance* = new LazySingleton(); //懒加载  }  return *sInstance*;  }   public void show() {  System.*out*.println("懒汉式");  } } |

synchronized是Java中实现代码块原子性的关键字之一。

##### 缺陷：

getInstance()函数加上了原子性后，确实解决了问题。但这又引入了新的问题：在getInstance()函数声明上加synchronized，意味着每次函数调用都会进行同步检查，这是低效的；由于对象只需要在初次初始化时需要同步，多数情况下不需要互斥地获得对象，加锁会造成巨大无意义的资源消耗。实际上，我们只需要保证如下创建实例代码的原子性即可

|  |
| --- |
| if (*sInstance* == null) {  *sInstance* = new LazySingleton(); //懒加载 } |

也就是说，这种实现方案的同步范围扩大了，这个问题由**双重检查锁**来解决。

## 1.4 双重检查锁【对懒汉式的改进】

### （1） 写法

在懒汉式中，在getInstance()加了synchronized，扩大了同步范围，导致了低效。现在来用双重检查锁方式减小一下同步范围：

|  |
| --- |
| public class DoubleCheckLockSingleton {   private volatile static DoubleCheckLockSingleton *sInstance*;   private DoubleCheckLockSingleton() {   }   public static DoubleCheckLockSingleton getsInstance() {  //先检查实例是否存在，如果不存在才进入下面的同步块  if (*sInstance* == null) {  //同步块，线程安全地创建实例  synchronized (DoubleCheckLockSingleton.class) {  //再次检查实例是否存在，如果不存在才真正地创建实例  if (*sInstance* == null) {  *sInstance* = new DoubleCheckLockSingleton();  }  }  }  return *sInstance*;  }   public void show() {  System.*out*.println("双重检查锁式");  } } |

所谓的**双重检查**，是在同步前后的两次if(sInstance==null)判断是否已经存在实例；**锁**自然指的就是synchronized关键字。

#### 说明：

##### 线程安全：

对着代码，再来考虑两个线程同时通过了第一道if(sInstance==null)检查，但因为同步锁是互斥的，只能第一个线程释放后，第二个线程才能持有。这保证了同步代码块的原子性，在同步代码块中，如果sInstance还未创建，这时才会创建。

##### volatile的使用：

此外，还需要注意的是**volatile**关键字的使用。

在sInstance=new DoubleCheckLockSingleton()；这行代码执行时，虚拟机大概可以分为三个指令步骤：

1. 在内存中给DoubleCheckLockSingleton实例分配空间。
2. 调用DoubleCheckLockSingleton构造函数，初始化成员
3. 为DoubleCheckLockSingleton实例指向第一步分配的内存空间（执行完后instance不为空）。

代码在编译时，存在指令优化的现象。指令优化只保证单线程条件下执行结果一致，而不保证执行的顺序。所以前面三个指令的执行顺序是不确定的，可能是1-2-3，也可以是1-3-2。如果顺序是1-3-2，当第三步执行完后，instance已经不为空了，但成员并未初始化，第二个线程使用该instance自然会报错。怎么解决呢？

volatile可以解决这个问题，该关键字可以确保相关变量涉及的代码指令不被优化顺序。

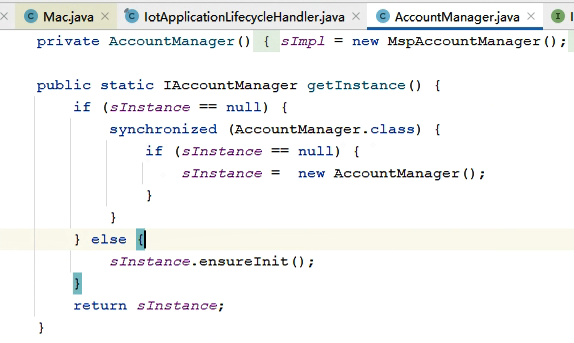
#### 总结：

回过来看双重检查锁的代码，既实现了线程安全，也实现了懒加载。已经很完美了，唯一缺点是：有点复杂。

### （2） 实例

#### 例1：

扩展写法：



## 1.5 静态内部类

由以上应该明白，最好的单例实现，需要满足两个条件：1.线程安全2.懒加载。前面的双重检查锁是手动实现，下面展示的是静态内部类方式的非手动实现：

|  |
| --- |
| public class StaticInnerClassSingleton {   private StaticInnerClassSingleton() {   }   private static class SingletonHolder {  private static StaticInnerClassSingleton *sInstance* = new StaticInnerClassSingleton();  }   public static StaticInnerClassSingleton getInstance() {  return SingletonHolder.*sInstance*;  }   public void show() {  System.*out*.println("静态内部类式");  } } |

1. 懒加载：使用静态内部类的方式，让类SingletonHolder只有在调getInstance()的时候才会被加载，单例才会创建，借机实现了懒加载。
2. 线程安全：由静态内部类中的静态成员初始化时创建实例，通过JVM类加载机制来保证线程的安全性。

## 1.6 枚举

### （1） 是什么

比静态内部类更简单的单例实现方法。

### （2） 写法

声明：

|  |
| --- |
| public enum EnumSingleton {   *INSTANCE*;   public void show() {  System.*out*.println("枚举式");  } } |

调用：

|  |
| --- |
| EnumSingleton.*INSTANCE*.show(); |

### （3） 说明

#### 参1：<https://blog.csdn.net/qq_25333681/article/details/93662660>

虽然在《高效Java 第二版》中说，单元素的枚举类型是实现单例的最佳方式。

该方式确实简洁方便，又不怕出错，但有一个缺点：无法在程序运行过程中通过运行参数实例化，代码失去灵活性。

#### 参2：<https://blog.csdn.net/mnb65482/article/details/80458571>

枚举在java中与普通类一样，都能拥有字段与方法，而且枚举实例创建是线程安全的，在任何情况下，它都是一个单例。

# 1. 单例（优缺点+适用场景）

## （1） 场景

### 参考1：《Head First Design Patterns电子版》P236

There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we’d run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

### 参考2：

<https://www.cnblogs.com/damsoft/p/6105122.html>

<https://blog.csdn.net/BlackPlus28/article/details/82794376>

1. 概述

单例模式只允许创建一个对象，因此节省内存，加快对象访问速度，因此对象需要被公用的场合适合使用，如多个模块使用同一个数据源连接对象等等。如：

  1. 需要频繁实例化然后销毁的对象。

2. 创建对象时耗时过多或者耗资源过多，但又经常用到的对象。

3. 有状态的工具类对象。   
 4. 频繁访问数据库或文件的对象。

5. 需要生成唯一序列的环境；

6. 方便资源相互通信的环境；

【2】 经典场景

1. 资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，应用配置。
2. 控制资源的情况下，方便资源之间的互相通信。如线程池等。

【3】 实例

1. 外部资源：每台计算机有若干个打印机，但只能有一个PrinterSpooler（后台打印程序），以避免两个打印作业同时输出到打印机。

内部资源：大多数软件都有一个（或多个）属性文件存放系统配置，这样的系统应该有一个对象管理这些属性文件；

1. Windows的Task Manager（任务管理器）就是很典型的单例模式（这个很熟悉吧），想想看，是不是呢，你能打开两个windows task manager吗？ 不信你自己试试看哦~；
2. windows的Recycle Bin（回收站）也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一个实例。

继续说回收站，我们在实际使用中并不存在需要同时打开两个回收站窗口的必要性。假如我每次创建回收站时都需要消耗大量的资源，而每个回收站之间资源是共享的，那么在没有必要多次重复创建该实例的情况下，创建了多个实例，这样做就会给系统造成不必要的负担，造成资源浪费。

1. 网站的计数器，一般也是采用单例模式实现，否则难以同步。 如果你存在多个计数器，每一个用户的访问都刷新计数器的值，这样的话你计数的值是难以同步的。但是如果采用单例模式实现就不会存在这样的问题，而且还可以避免线程安全问题。
2. 应用程序的日志应用，一般都用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
3. Web应用的配置对象的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。
4. 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因此  
   单例模式来维护，就可以大大降低这种损耗。
5. 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。
6. 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。

### 参考3：<https://hefeijoe.github.io/post/singeleton/>

开发工具类库中的很多工具类都应用了单例模式，比例线程池、缓存、日志对象等，它们都只需要创建一个对象。

如果创建多份实例，可能会带来不可预知的问题，比如资源的浪费、结果处理不一致等问题。单例模式的解决的痛点就是节约资源，节省时间从两个方面看。

1. 由于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些重量级的对象而言，是很重要的.
2. 因为不需要频繁创建对象，我们的GC压力也减轻了，而在GC中会有STW(stop the world)，从这一方面也节约了GC的时间。

### 参考4：<https://zhuanlan.zhihu.com/p/37382515>

#### 概述

什么是单例模式呢，单例模式(Singleton)又叫单态模式，它出现目的是为了保证一个类在系统中只有一个实例，并提供一个访问它的全局访问点。从这点可以看出，单例模式的出现是为了可以保证系统中一个类只有一个实例而且该实例又易于外界访问，从而方便对实例个数的控制并节约系统资源而出现的解决方案。

在下面几个场景中适合使用单例模式：

1、有频繁实例化然后销毁的情况，也就是频繁的 new 对象，可以考虑单例模式；

2、创建对象时耗时过多或者耗资源过多，但又经常用到的对象；

3、频繁访问 IO 资源的对象，例如数据库连接池或访问本地文件；

#### 【2】 实例

##### 《1》 网站在线人数统计

其实就是全局计数器，也就是说所有用户在相同的时刻获取到的在线人数数量都是一致的。要实现这个需求，计数器就要全局唯一，也就正好可以用单例模式来实现。当然这里不包括分布式场景，因为计数是存在内存中的，并且还要保证线程安全。

下面代码是一个简单的计数器实现。

|  |
| --- |
| public class Counter {  private static class CounterHolder {  private static final Counter *sCounter* = new Counter();  }   private Counter() {  System.*out*.println("init...");  }   public static Counter getInstance() {  return CounterHolder.*sCounter*;  }   private AtomicLong mOnline = new AtomicLong();   public long getOnline() {  return mOnline.get();  }   public long add() {  return mOnline.incrementAndGet();  } } |

##### 《2》 配置文件访问类

项目中经常需要一些环境相关的配置文件，比如短信通知相关的、邮件相关的。比如 properties 文件，这里就以读取一个properties 文件配置为例，如果你使用的 Spring ，可以用 @PropertySource 注解实现，默认就是单例模式。如果不用单例的话，每次都要 new 对象，每次都要重新读一遍配置文件，很影响性能；如果用单例模式，则只需要读取一遍就好了。

以下是文件访问单例类简单实现：

|  |
| --- |
| public class SingleProperty {   private static Properties *prop*;   private static class SinglePropertyHolder {  private static final SingleProperty *singleProperty* = new SingleProperty();  }   */\*\*  \* config.properties 内容是 test.name=kite  \*/* private SingleProperty() {  System.*out*.println("构造函数执行");  *prop* = new Properties();  InputStream stream = SingleProperty.class.getClassLoader()  .getResourceAsStream("config.properties");  try {  *prop*.load(new InputStreamReader(stream, "utf-8"));  } catch (IOException e) {  e.printStackTrace();  }  }   public static SingleProperty getInstance() {  return SinglePropertyHolder.*singleProperty*;  }   public String getName() {  return *prop*.get("test.name").toString();  }   public static void main(String[] args) {  SingleProperty singleProperty = SingleProperty.*getInstance*();  System.*out*.println(singleProperty.getName());  } } |

##### 《3》 数据库连接池的实现，也包括线程池

为什么要做池化，是因为新建连接很耗时，如果每次新任务来了，都新建连接，那对性能的影响实在太大。所以一般的做法是在一个应用内维护一个连接池，这样当任务进来时，如果有空闲连接，可以直接拿来用，省去了初始化的开销。所以用单例模式，正好可以实现一个应用内只有一个线程池的存在，所有需要连接的任务，都要从这个连接池来获取连接。如果不使用单例，那么应用内就会出现多个连接池，那也就没什么意义了。如果你使用 Spring 的话，并集成了例如 druid 或者 c3p0 ，这些成熟开源的数据库连接池，一般也都是默认以单例模式实现的。

# 1. 单例（实例）

## 例1：

<https://time.geekbang.org/column/article/176075>



## 例2：

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

懒汉式单例

# 1. 单例（Android多进程相关）

见

（1） “Android进程.docx”-3（3）【2】《3》；

（2） <https://time.geekbang.org/column/article/196790>

# 2. 工厂模式（概述）

参考：

1】 主：<https://www.cnblogs.com/yssjun/p/11102162.html>

2】 《Design Patterns: Elements of Reusable Object-Oriented Software》

3】 《Head First Design Patterns》

4】 carson：

简单工厂：<https://blog.csdn.net/carson_ho/article/details/52223153>

工厂方法：<https://www.jianshu.com/p/d0c444275827>

抽象工厂：<https://carsonho.blog.csdn.net/article/details/54910287>

5】 王争：《设计模式之美》

## （1） 是什么

工厂顾名思义就是创建产品，该模式用于封装和管理对象的创建，是一种创建型模式。

## （2） 分类

根据参考2】：只有工厂方法和抽象工厂。

根据参考3】：简单工厂事实上不是设计模式，更多的是一种编程习惯。所以通常将简单工厂看成工厂方法的特例，两者归为一类。

根据参考1】：根据产品是具体产品还是具体工厂可分为简单工厂模式和工厂方法模式，根据工厂的抽象程度可分为工厂方法模式和抽象工厂模式。

这里按递进的顺序从易到难对三种进行说明。

### 简单工厂模式

#### 是什么

该模式对对象创建管理方式最为简单，因为其仅仅简单的对不同类对象的创建进行了一层薄薄的封装。该模式通过向工厂传递类型来指定要创建的对象，其UML类图如下：

图示

描述已自动生成

注：参考3】：in design patterns, the phrase “implement an interface” does NOT always mean “write a class that implements a Java interface, by using the ‘implements’ keyword in the class declaration.” In the general use of the phrase, a concrete class implementing a method from a supertype (which could be a class OR interface) is still considered to be “implementing the interface” of that supertype.

所以这里用实现符号表示Product和AbstractProduct之间的关系。

#### 示例

##### 非静态工厂

**Phone接口**：对应AbstractProduct

|  |
| --- |
| public interface Phone {  void make(); } |

**OppoPhone类**：对应Product1

|  |
| --- |
| public class OppoPhone implements Phone {   public OppoPhone() {  this.make();  }   @Override  public void make() {  System.*out*.println("make oppo phone!");  } } |

**IPhone类**：对应Product2

|  |
| --- |
| public class IPhone implements Phone {   public IPhone() {  this.make();  }   @Override  public void make() {  System.*out*.println("make iphone!");  } } |

**PhoneFactory类**：对应Factory

|  |
| --- |
| public class PhoneFactory {  public Phone makePhone(String phoneType) {  if (phoneType.equals("OppoPhone")) {  return new OppoPhone();  } else if (phoneType.equals("IPhone")) {  return new IPhone();  }  return null;  } } |

Client：

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  PhoneFactory factory = new PhoneFactory();  Phone oppoPhone = factory.makePhone("OppoPhone"); //根据type参数创建指定对象  IPhone iPhone = (IPhone) factory.makePhone("IPhone"); //根据type参数创建指定对象  } } |

运行结果：

手机屏幕截图

描述已自动生成

##### 静态工厂

把**PhoneFactory类**改成：

|  |
| --- |
| public class PhoneFactory {  public static Phone makePhone(String phoneType) {  if (phoneType.equals("OppoPhone")) {  return new OppoPhone();  } else if (phoneType.equals("IPhone")) {  return new IPhone();  }  return null;  } } |

把Client改成：

|  |
| --- |
| public class Demo {  public static void main(String[] args) {  Phone oppoPhone = PhoneFactory.*makePhone*("OppoPhone"); //根据type参数创建指定对象  IPhone iPhone = (IPhone) PhoneFactory.*makePhone*("IPhone"); //根据type参数创建指定对象  } } |

##### 注意：

1. 静态工厂的优点是不需要实例化工厂，节省代码，但缺点是静态方法无法被继承和重写，造成工厂角色无法形成基于继承的等级结构。

### 工厂方法模式

#### 是什么

1. 和简单工厂模式中工厂负责生产所有产品相比，工厂方法模式将生成具体产品的任务分发给具体的产品工厂，其UML类图如下：

图示

描述已自动生成

【2】 工厂方法模式通过让子类决定该创建的对象是什么，来达到将对象创建的过程封装的目的。

**The Factory Method Pattern** defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

#### 示例

##### 非静态工厂

Phone接口，IPhone类，OppoPhone类同上面简单工厂模式。

**PhoneFactory接口**：对应AbstractFactory

|  |
| --- |
| public interface PhoneFactory {  Phone makePhone(); } |

**OppoFactory类**：对应Factory1

|  |
| --- |
| public class OppoFactory implements PhoneFactory{  @Override  public Phone makePhone() {  return new OppoPhone();  } } |

**AppleFactory类**：对应Factory2

|  |
| --- |
| public class AppleFactory implements PhoneFactory{  @Override  public Phone makePhone() {  return new IPhone();  } } |

Client：

|  |
| --- |
| public class Client {   public static void main(String[] args) {  //生产oppo手机  OppoFactory oppoFactory = new OppoFactory();  oppoFactory.makePhone();  //生产Apple手机  AppleFactory appleFactory = new AppleFactory();  appleFactory.makePhone();  } } |

运行结果：

文本

描述已自动生成

##### 静态工厂

工厂方法模式无静态工厂写法。

### 抽象工厂模式

#### 是什么

上面两种模式不管工厂怎么拆分抽象，都只是针对一类产品**Phone**（AbstractProduct），如果要生成另一种产品**PC**，应该怎么表示呢？

最简单的方式是把【2】中介绍的工厂方法模式完全复制一份，不过这次生产的是PC。但同时也就意味着我们要完全复制和修改Phone生产管理的所有代码，显然这是一个笨办法，并不利于扩展和维护。

抽象工厂模式通过在AbstarctFactory中增加创建产品的接口方法，并在具体子工厂中实现新加产品的创建，当然前提是子工厂支持生产该产品，否则继承的这个接口方法可以什么也不干。

其UML类图如下：

图示

描述已自动生成

从上面类图结构中可以清楚的看到如何在工厂方法模式中通过增加新产品的接口方法来实现产品的增加的【即工厂方法模式和抽象工厂模式的关联】。

#### 示例

##### 非静态工厂

**Phone接口**【对应AbstractProductA】，**IPhone类**【对应ProductA1】，**OppoPhone类**【对应ProductA2】同上面简单工厂模式。

**PC接口**：对应AbstractProductB

|  |
| --- |
| public interface PC {  void make(); } |

**OppoPC类**：对应ProductB1

|  |
| --- |
| public class OppoPC implements PC {   public OppoPC() {  this.make();  }   @Override  public void make() {  System.*out*.println("make oppo PC!");  } } |

**MAC类**：对应ProductB2

|  |
| --- |
| public class MAC implements PC {   public MAC() {  this.make();  }   @Override  public void make() {  System.*out*.println("make MAC!");  } } |

**ElectronicFactory接口**：对应AbstractFactory，增加制造PC的接口方法

|  |
| --- |
| public interface ElectronicFactory {  Phone makePhone();  PC makePC(); } |

**OppoFactory类**：对应Factory1，增加oppo PC的制造方法

|  |
| --- |
| public class OppoFactory implements ElectronicFactory {   @Override  public Phone makePhone() {  return new OppoPhone();  }   @Override  public PC makePC() {  return new OppoPC();  } } |

**AppleFactory类**：对应Factory2，增加苹果PC的制造方法

|  |
| --- |
| public class AppleFactory implements ElectronicFactory {   @Override  public Phone makePhone() {  return new IPhone();  }   @Override  public PC makePC() {  return new MAC();  } } |

Client：

|  |
| --- |
| public class Client {   public static void main(String[] args) {  OppoFactory oppoFactory = new OppoFactory();  AppleFactory appleFactory = new AppleFactory();  oppoFactory.makePhone();  oppoFactory.makePC();  appleFactory.makePhone();  appleFactory.makePC();  } } |

运行结果：

文本

描述已自动生成

##### 静态工厂

抽象工厂模式无静态工厂写法。

# 工厂模式（优缺点+场景）

## 场景

### 参考1：carson\_ho

1. 简单工厂模式：<https://www.jianshu.com/p/e55fbddc071c>

* 客户如果只知道传入工厂类的参数，对于如何创建对象的逻辑不关心时；
* 当工厂类负责创建的对象（具体产品）比较少时。

【2】 工厂方法模式：<https://www.jianshu.com/p/d0c444275827>

 当一个类不知道它所需要的对象的类时  
在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可；

 当一个类希望通过其子类来指定创建对象时  
在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

【3】 抽象工厂模式：<https://www.jianshu.com/p/7deb64f902db>

 一个系统不要求依赖产品类实例如何被创建、组合和表达的表达，这点也是所有工厂模式应用的前提。

 这个系统有多个系列产品，而系统中只消费其中某一系列产品

 系统要求提供一个产品类的库，所有产品以同样的接口出现，客户端不需要依赖具体实现。

### 参考2：<https://xie.infoq.cn/article/88c926822394aa1c80847dd2a>

# 2. 工厂模式（实例）

## （1） 简单工厂模式

【1】 <https://time.geekbang.org/column/article/165114> 课堂讨论

## （2） 工厂方法模式

【1】 <https://time.geekbang.org/column/article/179615> 理论解读与代码实

战一的最后

1. RxView-rxjava中

文本

描述已自动生成

## （3） 抽象工厂模式

## 暂未区分：

【2】 glide第二节课的1：00：32

【3】 Lottiecompositionfactory；

【4】<https://www.liaoxuefeng.com/wiki/1252599548343744/1281319170474017>

Integer.valueOf

# 建造者模式

## 前言

<https://time.geekbang.org/column/article/199674>

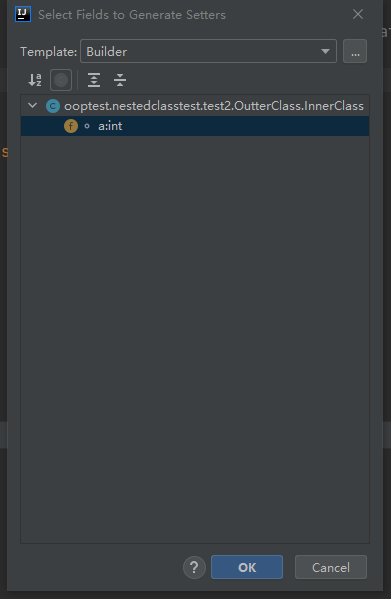
今天，我们再来学习另外一个比较常用的创建型设计模式，**Builder 模式**，中文翻译为**建造者模式**或者**构建者模式**，也有人叫它**生成器模式**。

实际上，建造者模式的原理和代码实现非常简单，掌握起来并不难，难点/重点是掌握应用场景，避免过度使用。比如，你有没有考虑过这样几个问题：直接使用构造函数或者配合 set 方法就能创建对象，为什么还需要建造者模式来创建呢？建造者模式和工厂模式都可以创建对象，那它们两个的区别在哪里呢？

话不多说，带着上面两个问题，让我们开始今天的学习吧！

## 写法

注：Builder模式在Builder类中的setter方法通常有return this，即链式调用，在Intellij IDEA中也有模板

，但return this不是必须的。

### 写法1【常见】

<https://time.geekbang.org/column/article/199674>

#### 【x】 为什么需要建造者模式/应用场景

##### 原因1：

###### 《1》 概述

如果一个类中有很多属性，为了避免构造函数的参数列表过长，影响代码的可读性和易用性，我们可以通过构造函数配合 set() 方法来解决。但是，如果存在下面情况中的任意一种，我们就要考虑使用建造者模式了：

* 我们把类的必填属性放到构造函数中，强制创建对象的时候就设置。如果必填的属性有很多，把这些必填属性都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填属性通过 set() 方法设置，那校验这些必填属性是否已经填写的逻辑就无处安放了。
* 如果类的属性之间有一定的依赖关系或者约束条件，我们继续使用构造函数配合 set() 方法的设计思路，那这些依赖关系或约束条件的校验逻辑就无处安放了。
* 如果我们希望创建不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值，要实现这个功能，我们就不能在类中暴露 set() 方法。构造函数配合 set() 方法来设置属性值的方式就不适用了。

###### 《2》 示例

见下面

##### 原因2：

实际上，使用建造者模式创建对象，还能避免对象存在无效状态。我再举个例子解释一下。比如我们定义了一个长方形类，如果不使用建造者模式，采用先创建后 set 的方式，那就会导致在第一个 set 之后，对象处于无效状态。具体代码如下所示：

|  |
| --- |
| Rectangle r = new Rectange(); // r is invalid  r.setWidth(2); // r is invalid  r.setHeight(3); // r is valid |

为了避免这种无效状态的存在，我们就需要使用构造函数一次性初始化好所有的成员变量。如果构造函数参数过多，我们就需要考虑使用建造者模式，先设置建造者的变量，然后再一次性地创建对象，让对象一直处于有效状态。

##### 注意

实际上，如果我们并不是很关心对象是否有短暂的无效状态，也不是太在意对象是否是可变的。比如，对象只是用来映射数据库读出来的数据，那我们直接暴露 set() 方法来设置类的成员变量值是完全没问题的。

#### 【x】 示例【原因1】

下面按序引出建造者模式：

##### 《1》 需求1

假设有这样一道设计面试题：我们需要定义一个资源池配置类 ResourcePoolConfig。这里的资源池，你可以简单理解为线程池、连接池、对象池等。在这个资源池配置类中，有以下几个成员变量，也就是可配置项。现在，请你编写代码实现这个 ResourcePoolConfig 类。



##### 《2》 用“构造器初始化所有成员变量”来实现需求1

###### 1） 思路

只要你稍微有点开发经验，那实现这样一个类对你来说并不是件难事。最常见、最容易想到的实现思路如下代码所示。因为 maxTotal、maxIdle、minIdle 不是必填变量，所以在创建 ResourcePoolConfig 对象的时候，我们通过往构造函数中，给这几个参数传递 null 值，来表示使用默认值。

###### 2） 代码

|  |
| --- |
| public class ResourcePoolConfig {  private static final int DEFAULT\_MAX\_TOTAL = 8;  private static final int DEFAULT\_MAX\_IDLE = 8;  private static final int DEFAULT\_MIN\_IDLE = 0;  private String name;  private int maxTotal = DEFAULT\_MAX\_TOTAL;  private int maxIdle = DEFAULT\_MAX\_IDLE;  private int minIdle = DEFAULT\_MIN\_IDLE;  public ResourcePoolConfig(String name, Integer maxTotal, Integer maxIdle, Integer minIdle) {  if (StringUtils.isBlank(name)) {  throw new IllegalArgumentException("name should not be empty.");  }  this.name = name;  if (maxTotal != null) {  if (maxTotal <= 0) {  throw new IllegalArgumentException("maxTotal should be positive.");  }  this.maxTotal = maxTotal;  }  if (maxIdle != null) {  if (maxIdle < 0) {  throw new IllegalArgumentException("maxIdle should not be negative.");  }  this.maxIdle = maxIdle;  }  if (minIdle != null) {  if (minIdle < 0) {  throw new IllegalArgumentException("minIdle should not be negative.");  }  this.minIdle = minIdle;  }  }  //...省略getter方法...  } |

###### 3） 缺陷

现在，ResourcePoolConfig 只有 4 个可配置项，对应到构造函数中，也只有 4 个参数，参数的个数不多。但是，如果可配置项逐渐增多，变成了 8 个、10 个，甚至更多，那继续沿用现在的设计思路，构造函数的参数列表会变得很长，代码在可读性和易用性上都会变差。在使用构造函数的时候，我们就容易搞错各参数的顺序，传递进错误的参数值，导致非常隐蔽的 bug。

|  |
| --- |
| // 参数太多，导致可读性差、参数可能传递错误  ResourcePoolConfig config = new ResourcePoolConfig("dbconnectionpool", 16, null, 8, null, false , true, 10, 20，false，true); |

##### 《3》 用“set函数给成员变量赋值”来实现需求1

###### 1） 思路

解决这个问题的办法你应该也已经想到了，那就是用 set() 函数来给成员变量赋值，以替代冗长的构造函数。

###### 2） 代码

我们直接看代码，具体如下所示。其中，配置项 name 是必填的，所以我们把它放到构造函数中设置，强制创建类对象的时候就要填写。其他配置项 maxTotal、maxIdle、minIdle 都不是必填的，所以我们通过 set() 函数来设置，让使用者自主选择填写或者不填写。

|  |
| --- |
| public class ResourcePoolConfig {  private static final int DEFAULT\_MAX\_TOTAL = 8;  private static final int DEFAULT\_MAX\_IDLE = 8;  private static final int DEFAULT\_MIN\_IDLE = 0;  private String name;  private int maxTotal = DEFAULT\_MAX\_TOTAL;  private int maxIdle = DEFAULT\_MAX\_IDLE;  private int minIdle = DEFAULT\_MIN\_IDLE;    public ResourcePoolConfig(String name) {  if (StringUtils.isBlank(name)) {  throw new IllegalArgumentException("name should not be empty.");  }  this.name = name;  }  public void setMaxTotal(int maxTotal) {  if (maxTotal <= 0) {  throw new IllegalArgumentException("maxTotal should be positive.");  }  this.maxTotal = maxTotal;  }  public void setMaxIdle(int maxIdle) {  if (maxIdle < 0) {  throw new IllegalArgumentException("maxIdle should not be negative.");  }  this.maxIdle = maxIdle;  }  public void setMinIdle(int minIdle) {  if (minIdle < 0) {  throw new IllegalArgumentException("minIdle should not be negative.");  }  this.minIdle = minIdle;  }  //...省略getter方法...  } |

接下来，我们来看新的 ResourcePoolConfig 类该如何使用。我写了一个示例代码，如下所示。没有了冗长的函数调用和参数列表，代码在可读性和易用性上提高了很多。

|  |
| --- |
| // ResourcePoolConfig使用举例  ResourcePoolConfig config = new ResourcePoolConfig("dbconnectionpool");  config.setMaxTotal(16);  config.setMaxIdle(8); |

##### 《4》 需求2【引出“建造者模式”】

至此，我们仍然没有用到建造者模式，通过构造函数设置必填项，通过 set() 方法设置可选配置项，就能实现我们的设计需求。

如果我们把问题的难度再加大点，比如，还需要解决下面这三个问题，那现在的设计思路就不能满足了：

* 我们刚刚讲到，name 是必填的，所以，我们把它放到构造函数中，强制创建对象的时候就设置。如果必填的配置项有很多，把这些必填配置项都放到构造函数中设置，那构造函数就又会出现参数列表很长的问题。如果我们把必填项也通过 set() 方法设置，那校验这些必填项是否已经填写的逻辑就无处安放了。
* 除此之外，假设配置项之间有一定的依赖关系，比如，如果用户设置了 maxTotal、maxIdle、minIdle 其中一个，就必须显式地设置另外两个；或者配置项之间有一定的约束条件，比如，maxIdle 和 minIdle 要小于等于 maxTotal。如果我们继续使用现在的设计思路，那这些配置项之间的依赖关系或者约束条件的校验逻辑就无处安放了。
* 如果我们希望 ResourcePoolConfig 类对象是不可变对象，也就是说，对象在创建好之后，就不能再修改内部的属性值。要实现这个功能，我们就不能在 ResourcePoolConfig 类中暴露 set() 方法。

为了解决这些问题，建造者模式就派上用场了。

##### 《5》 用“建造者模式”来实现需求2

###### 1） 思路

我们可以把校验逻辑放置到 Builder 类中，先创建建造者，并且通过 set() 方法设置建造者的变量值，然后在使用 build() 方法真正创建对象之前，做集中的校验，校验通过之后才会创建对象。除此之外，我们把 ResourcePoolConfig 的构造函数改为 private 私有权限。这样我们就只能通过建造者来创建 ResourcePoolConfig 类对象。并且，ResourcePoolConfig 没有提供任何 set() 方法，这样我们创建出来的对象就是不可变对象了。

###### 2） 代码

我们用建造者模式重新实现了上面的需求，具体的代码如下所示：

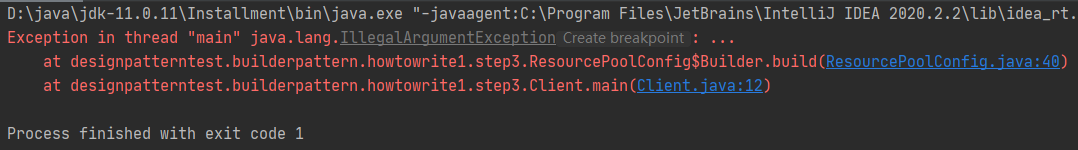
ResourcePoolConfig：

|  |
| --- |
| public class ResourcePoolConfig {  private String name;  private int maxTotal;  private int maxIdle;  private int minIdle;   private ResourcePoolConfig(Builder builder) {  this.name = builder.name;  this.maxTotal = builder.maxTotal;  this.maxIdle = builder.maxIdle;  this.minIdle = builder.minIdle;  }  //...省略getter方法...   //我们将Builder类设计成了ResourcePoolConfig的内部类。  //我们也可以将Builder类设计成独立的非内部类ResourcePoolConfigBuilder。  public static class Builder {  private static final int *DEFAULT\_MAX\_TOTAL* = 8;  private static final int *DEFAULT\_MAX\_IDLE* = 8;  private static final int *DEFAULT\_MIN\_IDLE* = 0;   private String name;  private int maxTotal = *DEFAULT\_MAX\_TOTAL*;  private int maxIdle = *DEFAULT\_MAX\_IDLE*;  private int minIdle = *DEFAULT\_MIN\_IDLE*;   public ResourcePoolConfig build() {  // 校验逻辑放到这里来做，包括必填项校验、依赖关系校验、约束条件校验等  if (StringUtils.*isBlank*(name)) {  throw new IllegalArgumentException("...");  }  if (maxIdle > maxTotal) {  throw new IllegalArgumentException("...");  }  if (minIdle > maxTotal || minIdle > maxIdle) {  throw new IllegalArgumentException("...");  }   return new ResourcePoolConfig(this);  }   public Builder setName(String name) {  if (StringUtils.*isBlank*(name)) {  throw new IllegalArgumentException("...");  }  this.name = name;  return this;  }   public Builder setMaxTotal(int maxTotal) {  if (maxTotal <= 0) {  throw new IllegalArgumentException("...");  }  this.maxTotal = maxTotal;  return this;  }   public Builder setMaxIdle(int maxIdle) {  if (maxIdle < 0) {  throw new IllegalArgumentException("...");  }  this.maxIdle = maxIdle;  return this;  }   public Builder setMinIdle(int minIdle) {  if (minIdle < 0) {  throw new IllegalArgumentException("...");  }  this.minIdle = minIdle;  return this;  }  } } |

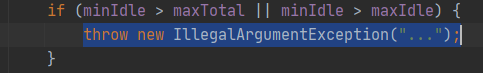
Client：

|  |
| --- |
| public class Client {   public static void main(String[] args) {  // 这段代码会抛出IllegalArgumentException，因为minIdle>maxIdle  ResourcePoolConfig config = new ResourcePoolConfig.Builder()  .setName("dbconnectionpool")  .setMaxTotal(16)  .setMaxIdle(10)  .setMinIdle(12)  .build();  } } |

运行结果：



定位到最终报错位置



#### 【x】 建造者模式的缺陷

使用建造者模式来构建对象，代码实际上是有点重复的，ResourcePoolConfig 类中的成员变量，要在 Builder 类中重新再定义一遍。

#### 【x】 与工厂模式有何区别

##### 《1》 区别概述

从上面的讲解中，我们可以看出，**建造者模式**是让建造者类来负责对象的创建工作。上一节课中讲到的**工厂模式**，是由工厂类来负责对象创建的工作。那它们之间有什么区别呢？

实际上，**工厂模式**是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**是用来创建一种类型的复杂对象，通过设置不同的可选参数，“定制化”地创建不同的对象。

##### 《2》 区别示例

网上有一个经典的例子很好地解释了两者的区别：

顾客走进一家餐馆点餐，我们利用**工厂模式**，根据用户不同的选择，来制作不同的食物，比如披萨、汉堡、沙拉。对于披萨来说，用户又有各种配料可以定制，比如奶酪、西红柿、起司，我们通过**建造者模式**根据用户选择的不同配料来制作披萨。

##### 《3》 注意点

实际上，我们也不要太学院派，非得把工厂模式、建造者模式分得那么清楚，我们需要知道的是，每个模式为什么这么设计，能解决什么问题。只有了解了这些最本质的东西，我们才能不生搬硬套，才能灵活应用，甚至可以混用各种模式创造出新的模式，来解决特定场景的问题。

##### 自归纳：也问了启昌

1） 相同

两者都属于创建型模式，即用来创建对象的。

2） 不同

a》 工厂模式：类似现实中工厂的概念，就是给一个方法参数【类似

工厂订单】来建造给定对象，没细化到内部实例变量的自定义设

置；

b》 builder模式：顾名思义建造，细化到了对象内部实例变量的自定

义设置。

#### 【x】 练习题

##### 题目：

在下面的 ConstructorArg 类中，当 isRef 为 true 的时候，arg 表示 String 类型的 refBeanId，type 不需要设置；当 isRef 为 false 的时候，arg、type 都需要设置。请根据这个需求，完善 ConstructorArg 类。

|  |
| --- |
| public class ConstructorArg {  private boolean isRef;  private Class type;  private Object arg;  // TODO: 待完善...  } |

##### 参考答案：

###### 答1



|  |
| --- |
| public class ConstructorArg {  private boolean isRef;  private Class type;  private Object arg;   public boolean isRef() {  return isRef;  }   public Class getType() {  return type;  }   public Object getArg() {  return arg;  }   private ConstructorArg(Builder builder) {  this.isRef = builder.isRef;  this.type = builder.type;  this.arg = builder.arg;  }   public static class Builder {  private boolean isRef;  private Class type;  private Object arg;   public ConstructorArg build() {  if(isRef && type != null) {  throw new IllegalArgumentException("...");  }   if (!isRef && type == null) {  throw new IllegalArgumentException("...");  }   if (this.isRef && (arg != null && arg.getClass() != String.class)) {  throw new IllegalArgumentException("...");  }   if (!this.isRef && arg == null) {  throw new IllegalArgumentException("...");  }   return new ConstructorArg(this);  }   public Builder setRef(boolean ref) {  if(ref && this.type != null) {  throw new IllegalArgumentException("...");  }  this.isRef = ref;  return this;  }   public Builder setType(Class type) {  if (this.isRef || type == null) {  throw new IllegalArgumentException("...");  }  this.type = type;  return this;  }   public Builder setArg(Object arg) {  if (this.isRef && (arg != null && arg.getClass() != String.class)) {  throw new IllegalArgumentException("...");  }   if (!this.isRef && arg == null) {  throw new IllegalArgumentException("...");  }  this.arg = arg;  return this;  }  } } |

一些评论：



###### 答2



|  |
| --- |
| public class ConstructorArg {  private boolean isRef;  private Class type;  private Object arg;   private ConstructorArg(Builder builder) {  this.isRef = builder.isRef;  this.type = builder.type;  this.arg = builder.arg;  }   public static class Builder {  private boolean isRef;  private Class type;  private Object arg;   public ConstructorArg build() {  if (arg == null) {  throw new IllegalArgumentException("arg必须设置");  }  if (isRef) {  if (!(arg instanceof String)) {  throw new IllegalArgumentException("arg必须为String类型的对象");  }  } else {  if (type == null) {  throw new IllegalArgumentException("arg必须设置");  }  }   return new ConstructorArg(this)  }   public Builder setRef(boolean ref) {  isRef = ref;  return this;  }   public Builder setArg(Object arg) {  this.arg = arg;  return this;  }   public Builder setType(Class type) {  this.type = type;  return this;  }  } } |

### 写法2

参考：

1】 <https://blog.csdn.net/jason0539/article/details/44992733>

2】 carson的2篇：内容相同，评论不同

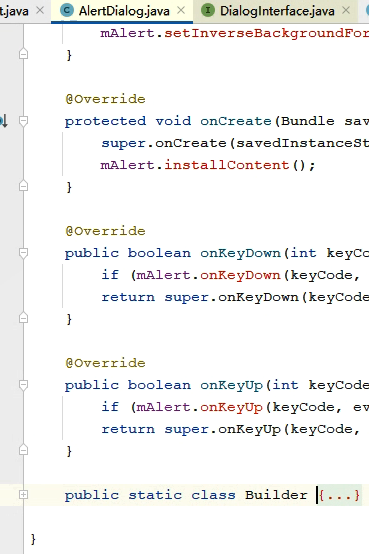
<https://blog.csdn.net/carson_ho/article/details/54910597>

<https://www.jianshu.com/p/be290ccea05a>

## （x） 实例

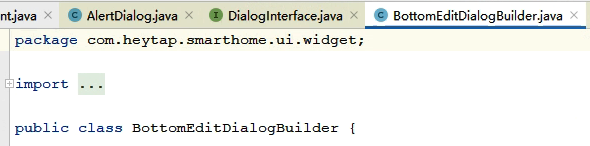
### 例1

android sdk的AlertDialog。



### 例2

oppo智能家居app



其中的show()方法：，内部代码

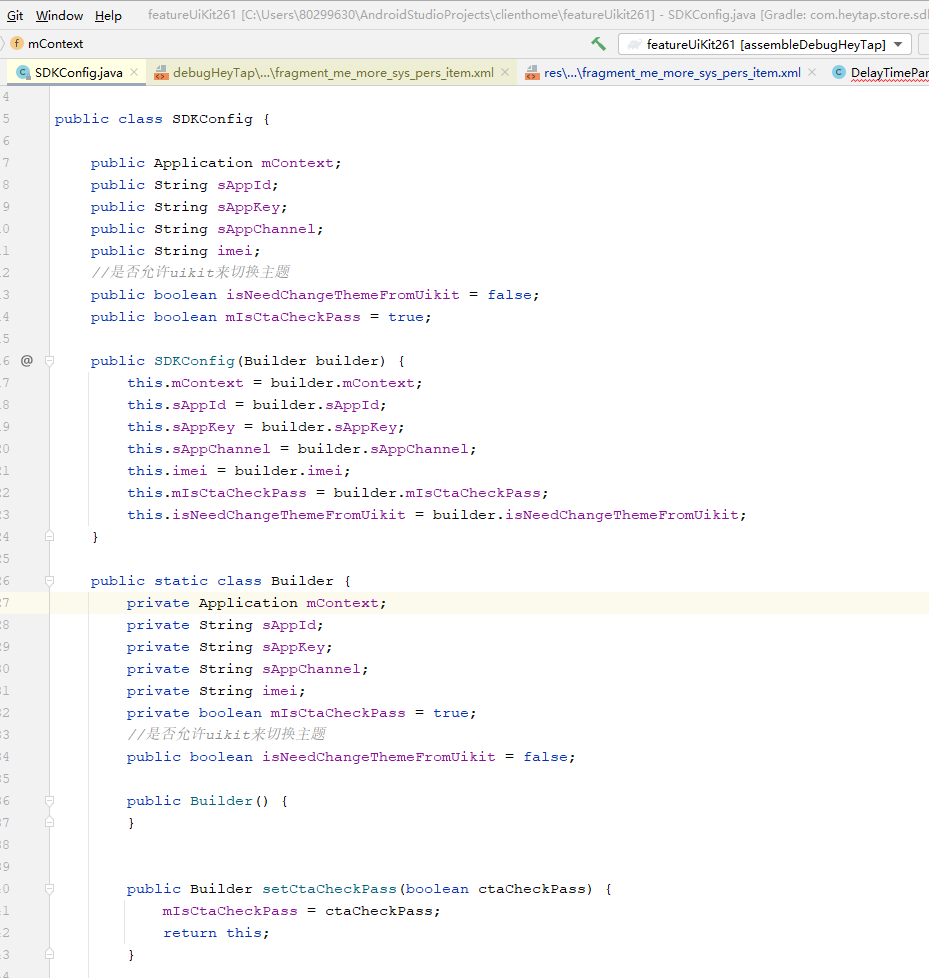


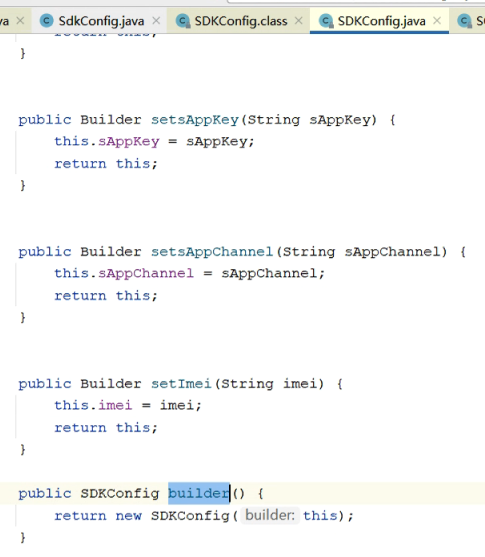


mBottomSheetDialog为：

### 例3

oppo智能家居app





### 例4：Retrofit

Retrofit.Builder

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

### 例5：OkHttp

OkHttpClient.Builder：

文本

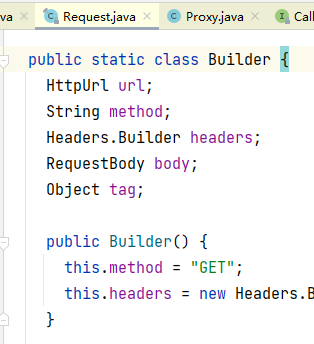
描述已自动生成

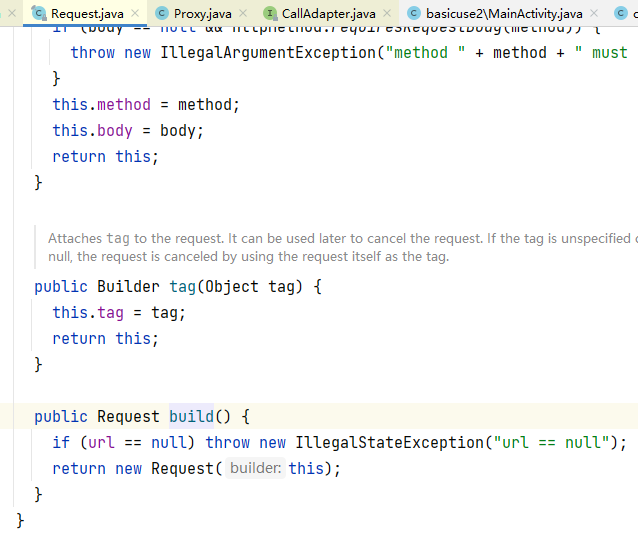


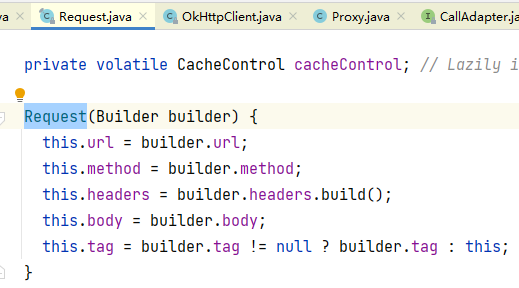
图形用户界面, 文本, 应用程序

描述已自动生成

Request.Builder：







# 4. 适配器模式

参考：

1. 主：<https://blog.csdn.net/wwwdc1012/article/details/82780560>【不少差评】
2. 辅：
3. 王争<https://time.geekbang.org/column/article/205912>
4. 《Head First Design Patterns电子版》CPT7
5. <https://www.jianshu.com/p/9d0575311214>：示例有问题
6. <https://blog.csdn.net/zxt0601/article/details/52848004>

## （1） 概述

### 参1：<https://blog.csdn.net/wwwdc1012/article/details/82780560>

将一个接口转换成客户希望的另一个接口，使接口不兼容的那些类可以一起工作，其别名为包装器(Wrapper)。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。

在适配器模式中，我们通过增加一个新的适配器类来解决接口不兼容的问题，使得原本没有任何关系的类可以协同工作。

### 参2：<https://time.geekbang.org/column/article/205912>

适配器模式的英文翻译是 Adapter Design Pattern。顾名思义，这个模式就是用来做适配的，它将不兼容的接口转换为可兼容的接口，让原本由于接口不兼容而不能一起工作的类可以一起工作。对于这个模式，有一个经常被拿来解释它的例子，就是 USB 转接头充当适配器，把两种不兼容的接口，通过转接变得可以一起工作。

注：上述usb转接头见<https://item.jd.com/100009644462.html>

图形用户界面

描述已自动生成

### 参3：《Head First Design Patterns》中文版 CPT7



上图就是电源/压适配器：

<https://baike.baidu.com/item/%E7%94%B5%E6%BA%90%E9%80%82%E9%85%8D%E5%99%A8/3032433>

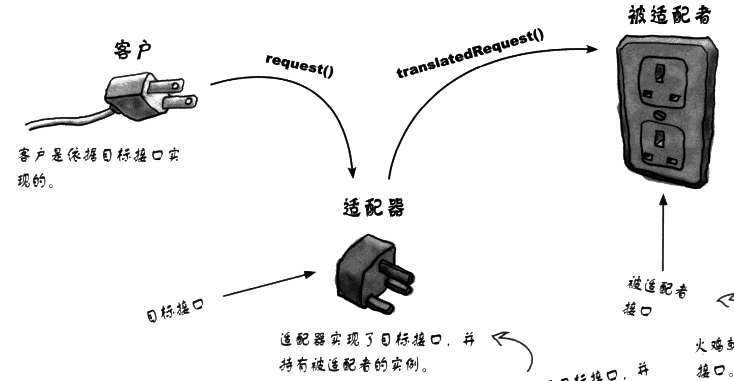
图片包含 信件

描述已自动生成

### 归纳：

用

结合



进行记忆。包括3个角色，角色之间的关系。

## （2） 角色

### 【1】 3大角色

#### 《1》 **Target：目标接口**

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

目标抽象类定义客户所需接口，可以是一个抽象类或接口，也可以是具体类。

<https://time.geekbang.org/column/article/205912>

ITarget 表示要转化成的接口定义。

<https://blog.csdn.net/zxt0601/article/details/52848004>

最终需要的输出（我们想要的），简称 **dst** (destination，即Target)

#### 《2》 **Adaptee：（被）适配者类**

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

适配者即被适配的角色，它定义了一个已经存在的接口，这个接口需要适配，适配者类一般是一个具体类，包含了客户希望使用的业务方法，在某些情况下可能没有适配者类的源代码。

<https://time.geekbang.org/column/article/205912>

Adaptee 是一组不兼容 ITarget 接口定义的接口

<https://blog.csdn.net/zxt0601/article/details/52848004>

需要被适配的类、接口、对象（我们有的），简称 **src**（source）

#### 《3》 **Adapter：适配器类**

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

适配器可以调用另一个接口，作为一个转换器，对Adaptee和Target进行适配，适配器类是适配器模式的核心，在对象适配器中，它通过继承Target并关联一个Adaptee对象使二者产生联系。

<https://time.geekbang.org/column/article/205912>

Adaptor 将 Adaptee 转化成一组符合 ITarget 接口定义的接口。

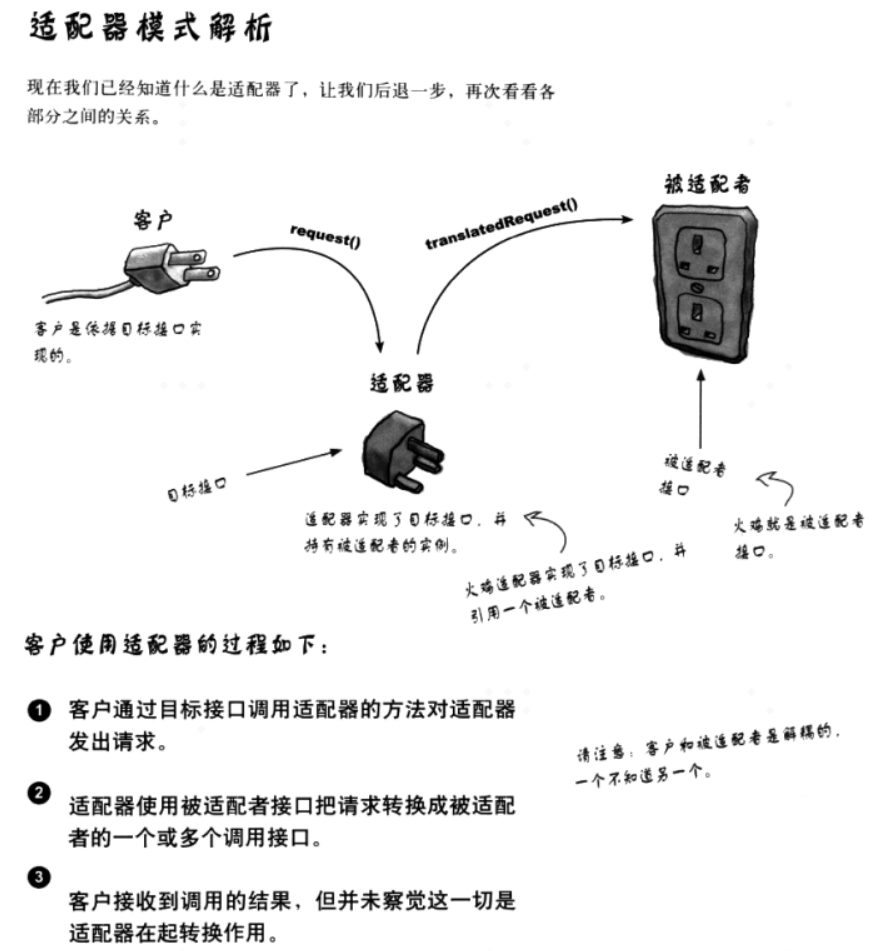
<https://blog.csdn.net/zxt0601/article/details/52848004>

适配器称之为 **Adapter** 。

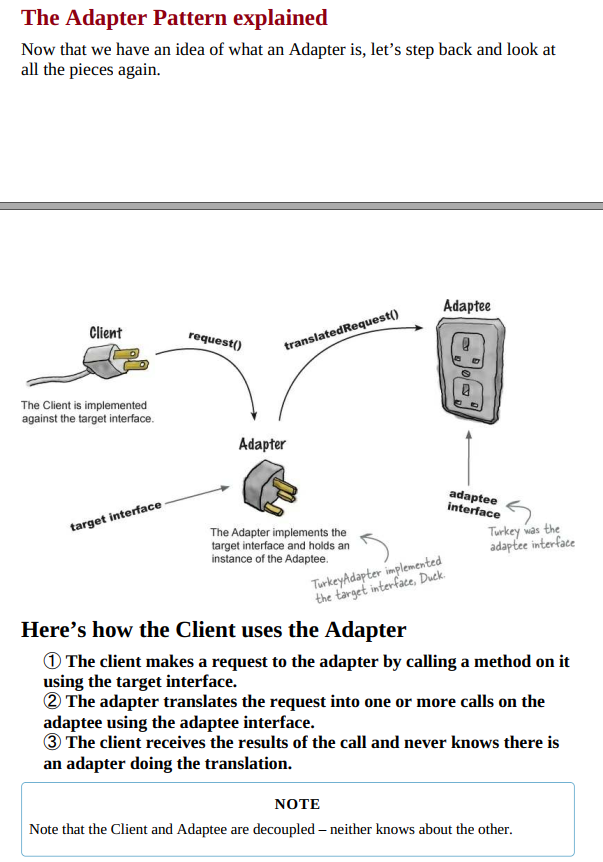
### 【2】 3者关系

#### 参1：《head first设计模式》

* 中文电子版 P241



* 对应的英文电子版



#### 参2：<https://blog.csdn.net/zxt0601/article/details/52848004>

一句话描述适配器模式的感觉： src->Adapter->dst,即src以某种形式（三种形式分别对应三种适配器模式）给到Adapter里，最终转化成了dst。

拿我们Android开发最熟悉的展示列表数据的三大控件：ListView，GridView，RecyclerView的Adapter来说，它们三个控件需要的是View(dst),而我们有的一般是datas(src),所以适配器Adapter就是完成了数据源datas 转化成 ItemView的工作。

带入src->Adapter->dst中，即datas->Adapter->View.

## （3） 写法

### 【1】 概述

#### 参1：<https://blog.csdn.net/wwwdc1012/article/details/82780560>

根据适配器类与适配者类的关系不同，适配器模式可分为对象适配器和类适配器两种，在**对象适配器模式**中，适配器与适配者之间是**关联**关系；在**类适配器模式**中，适配器与适配者之间是**继承**（或实现）关系。

对象适配器与类适配器不同之处在于，类适配器通过继承来完成适配，对象适配器则是通过关联来完成。

#### 参2：<https://time.geekbang.org/column/article/205912>

适配器模式有两种实现方式：类适配器和对象适配器。其中，类适配器使用继承关系来实现，对象适配器使用组合关系来实现。

#### 参3：<https://blog.csdn.net/zxt0601/article/details/52848004>

我个人理解，三种命名方式，是根据 src是以怎样的形式给到Adapter（在Adapter里的形式）来命名的。

类适配器，以类给到，在Adapter里，就是将src当做类，继承，

对象适配器，以对象给到，在Adapter里，将src作为一个对象，持有。

接口适配器，以接口给到，在Adapter里，将src作为一个接口，实现。

Adapter模式最大的作用还是将原本不兼容的接口融合在一起工作。

但是在实际开发中，实现起来不拘泥于本文介绍的三种经典形式，

例如Android中ListView、GridView的适配器Adapter，就不是以上三种经典形式之一，

我个人理解其属于对象适配器模式，一般日常使用中，我们都是在Adapter里持有datas，然后通过getView()/onCreateViewHolder()方法向ListView/RecyclerView提供View/ViewHolder。

Client是Lv Gv Rv ，它们是显示View的类。

所以dst(Target)是View。

一般来说我们有的src是数据datas，

即，我们希望：datas（src）->Adapter->View(dst)->Rv(Client)。

### 【2】 2种基本写法

注：下面的类适配器和对象适配器中的单接口方法和多接口方法，单接口方法其实是多接口方法中的一种情况，之后考虑合并。

#### 《1》 如何写

##### 1） 类适配器

###### 1》 单接口方法

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

a） 代码

Adaptee类：已存在的将被适配的类，即（被）适配者

|  |
| --- |
| public class Adaptee {   public void adapteeRequest() {  System.*out*.println("被适配者的方法");  } } |

Target接口：目标接口

|  |
| --- |
| public interface Target {   void request(); } |

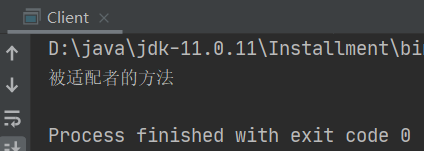
Adapter类：通过一个适配器类，实现 Target 接口，同时继承了 Adaptee 类，然后在实现的 request() 方法中调用父类的 adapteeRequest()

|  |
| --- |
| public class Adapter extends Adaptee implements Target {   @Override  public void request() {  //...一些操作...  super.adapteeRequest();  //...一些操作...  } } |

Client：

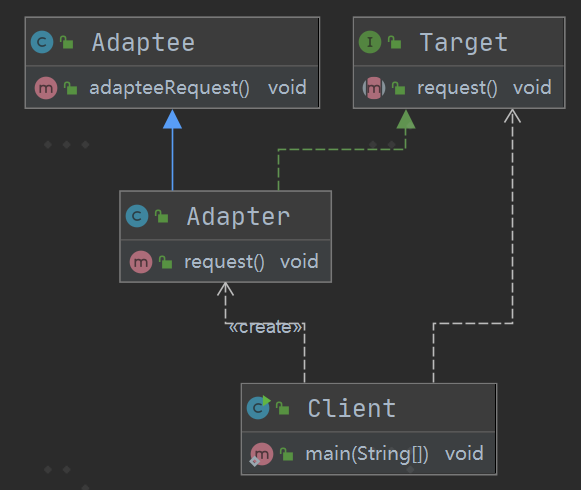
|  |
| --- |
| public class Client {   public static void main(String[] args) {  Target adapterTarget = new Adapter();  adapterTarget.request();  } } |

结果：



b） UML

idea ultimate的show diagram



###### 2》 多接口方法

<https://time.geekbang.org/column/article/205912>

|  |
| --- |
| // 类适配器: 基于继承  public interface ITarget {  void f1();  void f2();  void fc();  }  public class Adaptee {  public void fa() { //... }  public void fb() { //... }  public void fc() { //... }  }  public class Adaptor extends Adaptee implements ITarget {  public void f1() {  super.fa();  }    public void f2() {  //...重新实现f2()...  }    // 这里fc()不需要实现，直接继承自Adaptee，这是跟对象适配器最大的不同点  } |

##### 2） 对象适配器

###### 1》 单接口方法

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

a） 代码

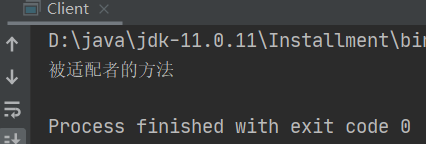
Adaptee类：同“1）1》a）”

Target接口：同“1）1》a）”

Adapter类：这里稍微修改一下 Adapter 类即可转变为对象适配器

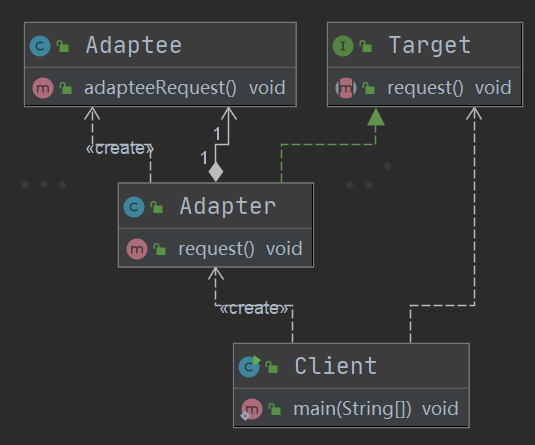
|  |
| --- |
| public class Adapter implements Target {   // 适配者是对象适配器的一个属性  private Adaptee adaptee = new Adaptee();   @Override  public void request() {  //...一些操作...  adaptee.adapteeRequest();  //...一些操作...  } } |

Client：同“1）1》a）”

结果：

b） UML

idea ultimate的show diagram



###### 2》 多接口方法

<https://time.geekbang.org/column/article/205912>

|  |
| --- |
| // 对象适配器：基于组合  public interface ITarget {  void f1();  void f2();  void fc();  }  public class Adaptee {  public void fa() { //... }  public void fb() { //... }  public void fc() { //... }  }  public class Adaptor implements ITarget {  private Adaptee adaptee;    public Adaptor(Adaptee adaptee) {  this.adaptee = adaptee;  }    public void f1() {  adaptee.fa(); //委托给Adaptee  }    public void f2() {  //...重新实现f2()...  }    public void fc() {  adaptee.fc();  }  } |

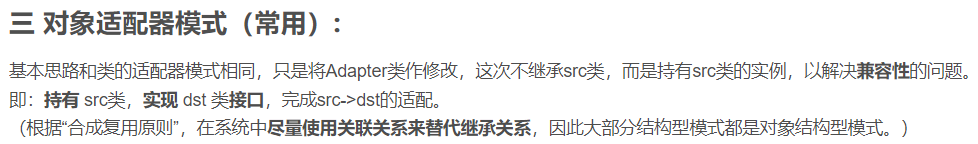
#### 《2》 如何选择

##### 参1：<https://time.geekbang.org/column/article/205912>

针对这两种实现方式，在实际的开发中，到底该如何选择使用哪一种呢？判断的标准主要有两个，一个是 Adaptee 接口的个数，另一个是 Adaptee 和 ITarget 的契合程度。

* 如果 Adaptee 接口并不多，那两种实现方式都可以。
* 如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都相同，那我们推荐使用类适配器，因为 Adaptor 复用父类 Adaptee 的接口，比起对象适配器的实现方式，Adaptor 的代码量要少一些。
* 如果 Adaptee 接口很多，而且 Adaptee 和 ITarget 接口定义大部分都不相同，那我们推荐使用对象适配器，因为组合结构相对于继承更加灵活。

##### 参2：<https://blog.csdn.net/zxt0601/article/details/52848004>



#### 《3》 示例：电压/源适配器

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

<https://blog.csdn.net/zxt0601/article/details/52848004>

<https://www.jianshu.com/p/31686bf8f9a2>

<https://blog.csdn.net/bboyfeiyu/article/details/43950185>

<https://blog.csdn.net/u010618194/article/details/65951320>

##### 1） 需求

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

再来一个好理解的例子，我们国家的民用电都是 220V，日本是 110V，而我们的手机充电一般需要 5V，这时候要充电，就需要一个电压适配器，将 220V 或者 110V 的输入电压变换为 5V 输出。

##### 2） 实现

###### 1》 类适配器

<https://blog.csdn.net/zxt0601/article/details/52848004>

充电器本身相当于Adapter，220V交流电相当于src【即适配者类】，我们的目标dst【即目标接口】是5V直流电。

我们现有的src类：

|  |
| --- |
| */\*\*  \* 介绍：src类: 我们有的220V电压  \* 作者：zhangxutong  \* 邮箱：zhangxutong@imcoming.com  \* 时间： 2016/10/18.  \*/* public class Voltage220 {  public int output220V() {  int src = 220;  System.*out*.println("我是" + src + "V");  return src;  } } |

我们想要的dst接口：

|  |
| --- |
| */\*\*  \* 介绍：dst接口：客户需要的5V电压  \* 作者：zhangxutong  \* 邮箱：zhangxutong@imcoming.com  \* 时间： 2016/10/18.  \*/* public interface Voltage5 {  int output5V(); } |

适配器类：

|  |
| --- |
| */\*\*  \* 介绍：Adapter类：完成220V-5V的转变  \* 通过继承src类，实现 dst 类接口，完成src->dst的适配。  \* 作者：zhangxutong  \* 邮箱：zhangxutong@imcoming.com  \* 时间： 2016/10/18.  \*/* public class VoltageAdapter extends Voltage220 implements Voltage5 {  @Override  public int output5V() {  int src = output220V();  System.*out*.println("适配器工作开始适配电压");  int dst = src / 44;  System.*out*.println("适配完成后输出电压：" + dst);  return dst;  } } |

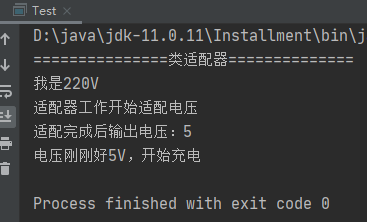
Client类：

|  |
| --- |
| */\*\*  \* 介绍：Client类：手机 .需要5V电压  \* 作者：zhangxutong  \* 邮箱：zhangxutong@imcoming.com  \* 时间： 2016/10/18.  \*/* public class Mobile {  */\*\*  \* 充电方法  \*  \** ***@param*** *voltage5  \*/* public void charging(Voltage5 voltage5) {  if (voltage5.output5V() == 5) {  System.*out*.println("电压刚刚好5V，开始充电");  } else if (voltage5.output5V() > 5) {  System.*out*.println("电压超过5V，都闪开 我要变成note7了");  }  } } |

测试代码：

|  |
| --- |
| public class Test {   public static void main(String[] args) {  System.*out*.println("===============类适配器==============");  Mobile mobile = new Mobile();  mobile.charging(new VoltageAdapter());  } } |

结果：



###### 2》 对象适配器

A） 写法1：标准写法

<https://blog.csdn.net/zxt0601/article/details/52848004>

同“1》类适配器”，其他部分不变，仅修改以下部分：

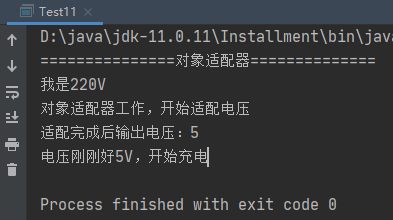
Adapter类如下：

|  |
| --- |
| */\*\*  \* 介绍：对象适配器模式：  \* 持有 src类，实现 dst 类接口，完成src->dst的适配。 。以达到解决\*\*兼容性\*\*的问题。  \* 作者：zhangxutong  \* 邮箱：zhangxutong@imcoming.com  \* 时间： 2016/10/18.  \*/* public class VoltageAdapter implements Voltage5 {  private Voltage220 mVoltage220;   public VoltageAdapter(Voltage220 voltage220) {  mVoltage220 = voltage220;  }   @Override  public int output5V() {  int dst = 0;  if (null != mVoltage220) {  int src = mVoltage220.output220V();  System.*out*.println("对象适配器工作，开始适配电压");  dst = src / 44;  System.*out*.println("适配完成后输出电压：" + dst);  }  return dst;  } } |

测试代码：

|  |
| --- |
| public class Test {   public static void main(String[] args) {  System.*out*.println("===============对象适配器==============");  VoltageAdapter voltageAdapter2 = new VoltageAdapter(new Voltage220());  Mobile mobile2 = new Mobile();  mobile2.charging(voltageAdapter2);  } } |

输出：



B） 写法2：变种

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

定义输出交流电接口，

|  |
| --- |
| public interface AC {  int outputAC(); } |

和两个电源类：输出220V交流电类和输出110V交流电类

|  |
| --- |
| public class AC110 implements AC {  public final int output = 110;   @Override  public int outputAC() {  return output;  } } |

|  |
| --- |
| public class AC220 implements AC {  public final int output = 220;   @Override  public int outputAC() {  return output;  } } |

接着定义接口，其中 support() 方法用于检查输入的电压是否与适配器匹配，outputDC5V() 方法则用于将输入的电压变换为 5V 后输出

|  |
| --- |
| public interface DC5Adapter {  boolean support(AC ac);   int outputDC5V(AC ac); } |

然后实现中国变压适配器和日本变压适配器（对象适配器）

|  |
| --- |
| public class ChinaPowerAdapter implements DC5 {  public static final int *voltage* = 220;   @Override  public boolean support(AC ac) {  return (*voltage* == ac.outputAC());  }   @Override  public int outputDC5V(AC ac) {  int adapterInput = ac.outputAC(); //和AC关联  //变压器...  int adapterOutput = adapterInput / 44;  System.*out*.println("使用ChinaPowerAdapter变压适配器，输入AC:" + adapterInput + "V" + "，输出DC:" + adapterOutput + "V");  return adapterOutput;  } } |

|  |
| --- |
| public class JapanPowerAdapter implements DC5 {  public static final int *voltage* = 110;   @Override  public boolean support(AC ac) {  return (*voltage* == ac.outputAC());  }   @Override  public int outputDC5V(AC ac) {  int adapterInput = ac.outputAC();  //变压器...  int adapterOutput = adapterInput / 22;  System.*out*.println("使用JapanPowerAdapter变压适配器，输入AC:" + adapterInput + "V" + "，输出DC:" + adapterOutput + "V");  return adapterOutput;  } } |

测试，准备中国变压适配器和日本变压适配器各一个，定义一个方法可以根据电压找到合适的变压器，然后进行测试

|  |
| --- |
| public class Test {  private List<DC5> adapters = new LinkedList<DC5>();   public Test() {  this.adapters.add(new ChinaPowerAdapter());  this.adapters.add(new JapanPowerAdapter());  }   // 根据电压找合适的变压器  public DC5 getPowerAdapter(AC ac) {  DC5 adapter = null;  for (DC5 ad : this.adapters) {  if (ad.support(ac)) {  adapter = ad;  break;  }  }  if (adapter == null) {  throw new IllegalArgumentException("没有找到合适的变压适配器");  }  return adapter;  }   public static void main(String[] args) {  Test test = new Test();  DC5 adapter = null;  // 中国电压  AC chinaAC = new AC220();  adapter = test.getPowerAdapter(chinaAC);  adapter.outputDC5V(chinaAC);   // 去日本旅游，电压是 110V  AC japanAC = new AC110();  adapter = test.getPowerAdapter(japanAC);  adapter.outputDC5V(japanAC);  } } |

结果



### 【3】 变体写法

#### 《1》 缺省/接口适配器模式

##### 是什么

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

缺省适配器模式(Default Adapter Pattern)：当不需要实现一个接口所提供的所有方法时，可先设计一个抽象类实现该接口，并为接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可以选择性地覆盖父类的某些方法来实现需求，它适用于不想使用一个接口中的所有方法的情况，又称为单接口适配器模式。缺省适配器模式是适配器模式的一种变体，其应用也较为广泛。在JDK类库的事件处理包java.awt.event中广泛使用了缺省适配器模式，如WindowAdapter、KeyAdapter、MouseAdapter等。

##### 示例

<https://blog.csdn.net/zxt0601/article/details/52848004>

Android属性动画ValueAnimator的AnimatorListenerAdapter类。

## （4） 优缺点

### 优点

#### 《1》 适配器模式

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

1） 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，无须修改原有结构。

2） 增加了类的透明性和复用性，将具体的业务实现过程封装在适配者类中，对于客户端类而言是透明的，而且提高了适配者的复用性，同一个适配者类可以在多个不同的系统中复用。

3） 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。

#### 《2》 类适配器

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

具体来说，类适配器模式还有如下优点：

* 由于适配器类是适配者类的子类，因此可以在适配器类中置换一些适配者的方法，使得适配器的灵活性更强。

#### 《3》 对象适配器

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

对象适配器模式还有如下优点：

* 一个对象适配器可以把多个不同的适配者适配到同一个目标；
* 可以适配一个适配者的子类，由于适配器和适配者之间是关联关系，根据“里氏代换原则”，适配者的子类也可通过该适配器进行适配。

### 缺点

#### 适配器模式

<https://www.jianshu.com/p/9d0575311214>

过多的使用适配器，会让系统非常零乱，不易整体进行把握

#### 《2》 类适配器

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

类适配器模式的缺点如下：

* 对于Java、C#等不支持多重类继承的语言，一次最多只能适配一个适配者类，不能同时适配多个适配者；
* 适配者类不能为最终类，如在Java中不能为final类，C#中不能为sealed类；
* 在Java、C#等语言中，类适配器模式中的目标抽象类只能为接口，不能为类，其使用有一定的局限性。

#### 《3》 对象适配器

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

对象适配器模式的缺点如下：

* 与类适配器模式相比，要在适配器中置换适配者类的某些方法比较麻烦。如果一定要置换掉适配者类的一个或多个方法，可以先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

## （5） 应用场景

### 参1：

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

* 系统需要使用一些现有的类，而这些类的接口（如方法名）不符合系统的需要，甚至没有这些类的源代码。
* 想创建一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

### 参2：<https://time.geekbang.org/column/article/205912>

一般来说，适配器模式可以看作一种“补偿模式”，用来补救设计上的缺陷。应用这种模式算是“无奈之举”。如果在设计初期，我们就能协调规避接口不兼容的问题，那这种模式就没有应用的机会了。

前面我们反复提到，适配器模式的应用场景是“接口不兼容”。那在实际的开发中，什么情况下才会出现接口不兼容呢？我建议你先自己思考一下这个问题，然后再来看我下面的总结 。

#### 封装有缺陷的接口设计

#### 统一多个类的接口设计

#### 替换依赖的外部系统

#### 兼容老版本接口

#### 适配不同格式的数据

### 其他已看过资料：

<https://juejin.cn/post/6850418114694709256>

<https://blog.csdn.net/zxt0601/article/details/52848004>

<https://m.yisu.com/zixun/308699.html>

<https://www.cnblogs.com/v1haoge/p/6479118.html>

<https://juejin.cn/post/6844903893441183752>

## （6） 实例

### spring AOP中的适配器模式

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

### spring JPA中的适配器模式

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

### spring MVC中的适配器模式

<https://blog.csdn.net/wwwdc1012/article/details/82780560>

### Java日志

<https://time.geekbang.org/column/article/205912>

### 【5】 RecyclerView

<https://www.jianshu.com/p/31686bf8f9a2>

### 【6】 ListView

<https://blog.csdn.net/bboyfeiyu/article/details/43950185>

### 【x】 GridView

<https://blog.csdn.net/zxt0601/article/details/52848004>

### 【x】 POS机app

<https://blog.csdn.net/u010618194/article/details/65951320>

### 【x】 Jetpack-Room

### 【x】 ViewPager

## （7） 和代理、桥接、装饰器 的区别

<https://time.geekbang.org/column/article/205912>

代理、桥接、装饰器、适配器，这 4 种模式是比较常用的结构型设计模式。它们的代码结构非常相似。笼统来说，它们都可以称为 Wrapper 模式，也就是通过 Wrapper 类二次封装原始类。

尽管代码结构相似，但这 4 种设计模式的用意完全不同，也就是说要解决的问题、应用场景不同，这也是它们的主要区别。

这里我就简单说一下它们之间的区别：

* **代理模式**：代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同。
* **桥接模式**：桥接模式的目的是将接口部分和实现部分分离，从而让它们可以较为容易、也相对独立地加以改变。
* **装饰器模式**：装饰者模式在不改变原始类接口的情况下，对原始类功能进行增强，并且支持多个装饰器的嵌套使用。
* **适配器模式**：适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。

# 5. 代理模式（总述）

参考：

1】 主：

2】 辅：

1》 carson：<https://www.jianshu.com/p/a8aa6851e09e>

## （1） 是什么

<https://www.jianshu.com/p/a8aa6851e09e>

给目标对象提供一个代理对象，并由代理对象控制对目标对象的引用

|  |
| --- |
| 1. 代理对象：起到中介作用，连接客户端和目标对象 2. 例子：电脑桌面的快捷方式。电脑对某个程序提供一个快捷方式（代理对象），快捷方式连接客户端和程序，客户端通过操作快捷方式就可以操作那个程序 |

## （2） 为什么

<https://www.jianshu.com/p/a8aa6851e09e>

通过引入**代理对象**的方式来间接访问**目标对象，**防止**直接**访问目标对象给系统带来的不必要复杂性。

## （3） 角色组成

<https://www.mianshigee.com/tutorial/android_interview/java-basis-reflection2.md>

https://www.jianshu.com/p/a8aa6851e09e

<https://wangkuiwu.github.io/2012/10/21/design_pattern/>

代理模式的角色分四种：

图示

描述已自动生成

### ****抽象对象/主题：****

Subject是委托对象和代理对象都共同实现的接口，即代理类所实现的行为接口。Request()是委托对象和代理对象共同拥有的方法。

### ****原始/被代理/真实/目标对象/主题：****

RealSubject是原对象，也就是被代理的对象。

### ****代理对象/主题：****

Proxy是代理对象，用来封装真实主题类的代理类。

### ****客户端：****

使用代理类和主题接口完成一些工作。

注：委托的定义：

<https://baike.baidu.com/item/%E5%A7%94%E6%89%98/34711>

同参考：<https://juejin.cn/post/7000350241262305293>

<https://www.jianshu.com/p/4ffd3326222d>

## （4） 分类

<https://www.mianshigee.com/tutorial/android_interview/java-basis-reflection2.md>

代理模式的实现分为：

* **静态代理：**见下面的“代理模式（静态代理）”
* **动态代理：**见下面的“代理模式（动态代理）”

## （5） 应用场景

### 参考1：<https://www.jianshu.com/p/a8aa6851e09e>

表格

描述已自动生成

### 参考2：<https://time.geekbang.org/column/article/201823>

代理模式常用在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类统一处理，让程序员只需要关注业务方面的开发。

除此之外，代理模式还可以用在 RPC、缓存等应用场景中。

### 参考3：

<https://blog.csdn.net/u011151135/article/details/50864869>

 代理模式是常用的结构型设计模式之一，它为对象的间接访问提供了一个解决方案，可以对对象的访问进行控制。代理模式类型较多，其中远程代理、虚拟代理、保护代理等在软件开发中应用非常广泛。

**(1) 当客户端对象需要访问远程主机中的对象时可以使用远程代理。**

**(2) 当需要用一个消耗资源较少的对象来代表一个消耗资源较多的对象，从而降低系统开销、缩短运行时间时可以使用虚拟代理，例如一个对象需要很长时间才能完成加载时。**

**(3) 当需要为某一个被频繁访问的操作结果提供一个临时存储空间，以供多个客户端共享访问这些结果时可以使用缓冲代理。通过使用缓冲代理，系统无须在客户端每一次访问时都重新执行操作，只需直接从临时缓冲区获取操作结果即可。**

**(4) 当需要控制对一个对象的访问，为不同用户提供不同级别的访问权限时可以使用保护代理。**

**(5) 当需要为一个对象的访问（引用）提供一些额外的操作时可以使用智能引用代理。**

### 参考4：<https://www.cnblogs.com/Eason-S/p/5856141.html>

**场景1：**当我们需要使用的对象很复杂或者需要很长时间去构造，这时就可以使用代理模式(Proxy)。例如：如果构建一个对象很耗费时间和计算机资源，代理模式(Proxy)允许我们控制这种情况，直到我们需要使用实际的对象。

一个代理(Proxy)通常包含和将要使用的对象同样的方法，一旦开始使用这个对象，这些方法将通过代理(Proxy)传递给实际的对象。

 一些可以使用代理模式(Proxy)的情况：

* 一个对象，比如一幅很大的图像，需要载入的时间很长，加载前可用一个占位的图像替代。
* 一个需要很长时间才可以完成的计算结果，并且需要在它计算过程中显示中间结果
* 一个存在于远程计算机上的对象，需要通过网络载入这个远程对象则需要很长时间，特别是在网络传输高峰期。
* 一个对象只有有限的访问权限，代理模式(Proxy)可以验证用户的权限

**场景2：**代理模式(Proxy)也可以被用来区别一个对象实例的请求和实际的访问，例如：在程序初始化过程中可能建立多个对象，但并不都是马上使用，代理模式(Proxy)可以载入需要的真正的对象。比如有一个需要载入和显示一幅很大的图像的程序，当程序启动时，就必须确定要显示的图像，但是实际的图像只能在完全载入后才可以显示！这时我们就可以使用代理模式(Proxy)。

**场景3：**使用代理模式来将由一系列无关逻辑组合在一起的代码进行解耦合，比如业务代码中的日志代码就可以在代理中进行。spring的AOP就是典型的动态代理应用。

不同代理类型：

1. 远程代理(Remote Proxy) -可以隐藏一个对象存在于不同地址空间的事实。也使得客户端可以访问在远程机器上的对象，远程机器可能具有更好的计算性能与处理速度，可以快速响应并处理客户端请求。
2. 虚拟代理(Virtual Proxy) – 允许内存开销较大的对象在需要的时候创建。只有我们真正需要这个对象的时候才创建。
3. 写入时复制代理(Copy-On-Write Proxy) – 用来控制对象的复制，方法是延迟对象的复制，直到客户真的需要为止。是虚拟代理的一个变体。
4. 保护代理(Protection (Access)Proxy) – 为不同的客户提供不同级别的目标对象访问权限
5. 缓存代理(Cache Proxy) – 为开销大的运算结果提供暂时存储，它允许多个客户共享结果，以减少计算或网络延迟。
6. 防火墙代理(Firewall Proxy) – 控制网络资源的访问，保护主题免于恶意客户的侵害。
7. 同步代理(SynchronizationProxy) – 在多线程的情况下为主题提供安全的访问。
8. 智能引用代理(Smart ReferenceProxy) - 当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来等。
9. 复杂隐藏代理(Complexity HidingProxy) – 用来隐藏一个类的复杂集合的复杂度，并进行访问控制。有时候也称为外观代理(Façade Proxy)，这不难理解。复杂隐藏代理和外观模式是不一样的，因为代理控制访问，而外观模式是不一样的，因为代理控制访问，而外观模式只提供另一组接口。

### 参考5：https://zhuanlan.zhihu.com/p/92051694

**代理模式的适用场景** 一个设计模式，只有当它有了应用场景，它才具备存在的价值。代理模式有很多可以使用的场景，主要分为如下几类：

**远程代理**：为一个对象的地址空间提供局部代表。  
**虚拟代理**：根据需要来创建开销较大的对象。  
**保护代理**：用于对象应该具有不同访问权限的场景，控制对原始对象的访问。  
**智能指引**：取代简单的指针，它在访问对象时执行一些附加操作。  
-- 引自《设计模式：可复用面向对象软件的基础》

### 参考6：

<https://blog.csdn.net/u011784767/article/details/78281384>

(1)设计模式中有一个设计原则是开闭原则，是说对修改关闭对扩展开放，我们在工作中有时会接手很多前人的代码，里面代码逻辑让人摸不着头脑(sometimes the code is really like shit)，这时就很难去下手修改代码，那么这时我们就可以通过代理对类进行增强。

(2)我们在使用RPC框架的时候，框架本身并不能提前知道各个业务方要调用哪些接口的哪些方法 。那么这个时候，就可通过动态代理的方式来建立一个中间人给客户端使用，也方便框架进行搭建逻辑，某种程度上也是客户端代码和框架松耦合的一种表现。

(3)Spring的AOP机制就是采用动态代理的机制来实现切面编程。

### 参考7：

<https://my.oschina.net/jack90john/blog/1786686>

   在足球比赛中一个比较形象的反映就是：主教练把队长（或其他队员，这里就指定为队长吧）叫到场边告诉他：叫XXX参与防守或叫XXX压上进攻。这时：队长就是代理类，而XXX就是主教练想要访问的目标对象。下面就基于以上场景来理解代理模式。

在一个对象不适合或者不能直接引用另一个对象时，我们可以使用代理模式来创建代理类在客户端和目标对象之间起到中介的作用。

### 参考8：

<https://www.huaweicloud.com/articles/0ba0e5ecac093b6d056f7bea7f04312b.htm>

在生活中，我们经常见到这样的场景，如：租房中介、售票黄牛、婚介、经纪人、快递、  
事务代理、非侵入式日志监听等，这些都是代理模式的实际体现。代理模式（Proxy  
Pattern）的定义也非常简单，是指为其他对象提供一种代理，以控制对这个对象的访问。  
代理对象在客服端和目标对象之间起到中介作用，代理模式属于结构型设计模式。使用  
代理模式主要有两个目的：一保护目标对象，二增强目标对象。

### 参考9：<http://c.biancheng.net/view/1359.html>

当无法或不想直接引用某个对象或访问某个对象存在困难时，可以通过代理对象来间接访问。使用代理模式主要有两个目的：一是保护目标对象，二是增强目标对象。

前面分析了代理模式的结构与特点，现在来分析以下的应用场景。

* 远程代理，这种方式通常是为了隐藏目标对象存在于不同地址空间的事实，方便客户端访问。例如，用户申请某些网盘空间时，会在用户的文件系统中建立一个虚拟的硬盘，用户访问虚拟硬盘时实际访问的是网盘空间。
* 虚拟代理，这种方式通常用于要创建的目标对象开销很大时。例如，下载一幅很大的图像需要很长时间，因某种计算比较复杂而短时间无法完成，这时可以先用小比例的虚拟代理替换真实的对象，消除用户对服务器慢的感觉。
* 安全代理，这种方式通常用于控制不同种类客户对真实对象的访问权限。
* 智能指引，主要用于调用目标对象时，代理附加一些额外的处理功能。例如，增加计算真实对象的引用次数的功能，这样当该对象没有被引用时，就可以自动释放它。
* 延迟加载，指为了提高系统的性能，延迟对目标的加载。例如，[Hibernate](http://c.biancheng.net/hibernate/" \t "_blank) 中就存在属性的延迟加载和关联表的延时加载。

### 参考10：

[**https://www.runoob.com/design-pattern/proxy-pattern.html**](https://www.runoob.com/design-pattern/proxy-pattern.html)

按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access）代理。 5、Cache代理。 6、防火墙（Firewall）代理。 7、同步化（Synchronization）代理。 8、智能引用（Smart Reference）代理。

### 参考11：

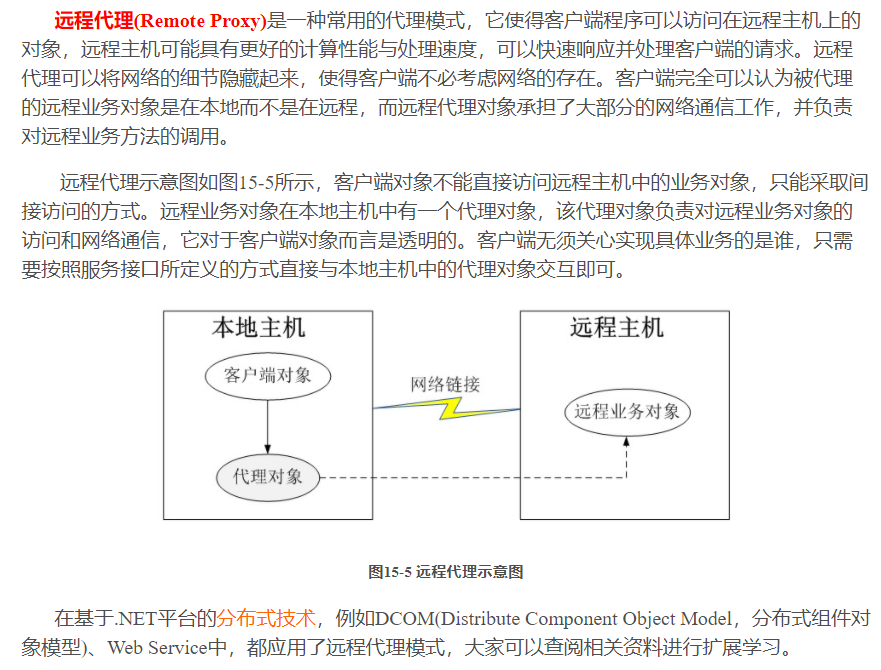
<https://zhuanlan.zhihu.com/p/37577256>

<https://blog.csdn.net/Mark_LQ/article/details/48158187>

1. 延迟加载：延迟加载的思想：如果当前没有使用这个组件时，则不需要真正地去初始化它，而是用一个代理对象去替代它的原有位置。当真正需要使用的时候，才对它进行加载。使用代理模式实现延迟加载是很有意义的，首先从时间轴上分散系统的压力，尤其在系统启动时，不必完成所有的初始化工作，从而加速启动时间；其次，对于很多真实主题而言，在软件启动到系统关闭的整个过程，可能都不会被调用，初始化这些数据无疑是一种资源的浪费。

例：假设某客户端软件，根据用户请求，去数据库查询数据的功能。在数据库查询前需要获得数据库连接。在系统启动时，初始化系统所有的类，此时尝试获得数据库连接。当系统存在大量类似的操作时（XML解析等），所有这些初始化操作都必须叠加，使得系统很慢。为此，使用代理模式，使用代理类，封装对数据库的查询操作。当系统启动时，初始化这个代理类，而非真实的数据库查询类，而代理类什么都不做。当真正开始查询的时候，才初始化查询对象。

2. 远程代理：为一个对象在不同地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。例如



3. 虚拟代理：是根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象。例如：老阮(MrRuan)在地点A，要到餐厅柜台办事（地点B），因为距离远却是很费劲，而老李刚好在这里（地点B）上班，所以让老李去办是很可行的办法。

4. 安全代理：用来控制真实对象访问时的权限。例如：老阮跟餐厅的柜台MM刚分手不方便去办理，所以需要借助老李去完成事项的办理。

5. 智能代理：是指当调用真实的对象时，代理去处理另外一些事情。例如：老李帮助老阮办理卡片激活时，顺便说说老阮的好话，让她俩能够和好。

## （6） 优缺点

### 优点

<https://www.jianshu.com/p/a8aa6851e09e>

* 协调调用者和被调用者，降低了系统的耦合度
* 代理对象作为客户端和目标对象之间的中介，起到了保护目标对象的作用

### 缺点

<https://www.jianshu.com/p/a8aa6851e09e>

* 由于在客户端和真实主题之间增加了代理对象，因此会造成请求的处理速度变慢；
* 实现代理模式需要额外的工作（有些代理模式的实现非常复杂），从而增加了系统实现的复杂度。

# 5. 代理模式（静态代理）

参考：

1】 主：carson <https://www.jianshu.com/p/a8aa6851e09e>

2】 辅：

1》 王争<https://time.geekbang.org/column/article/201823>

2》 

## 是什么

<https://www.mianshigee.com/tutorial/android_interview/java-basis-reflection2.md>

**静态代理：**代理类是在编译时就实现好的。也就是说 Java 编译完成后代理类是一个实际的 class 文件。

## 写法

### 【1】 写法1：真实类和代理类实现相同接口

<https://www.jianshu.com/p/a8aa6851e09e>

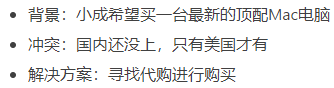
#### 《1》 UML类图 & 组成



#### 《2》 示例1

接下来我用一个实例来对代理模式进行更深一步的介绍。

##### 1） 实例概况





##### 2） 使用步骤

###### **步骤1：**创建抽象对象接口（Subject）：声明你（真实对象）需要让代购（代理对象）帮忙做的事（买Mac）

|  |
| --- |
| //抽象对象接口 public interface Subject {   public void buyMac(); } |

###### **步骤2：**创建真实对象类（RealSubject），即“我”

|  |
| --- |
| //真实对象类 public class RealSubject implements Subject {   @Override  public void buyMac() {  System.*out*.println("买一台Mac");  } } |

###### **步骤3：**创建代理对象类（Proxy），即”代购“，并通过代理类创建真实对象实例并访问其方法

|  |
| --- |
| //代理对象类 public class Proxy implements Subject {   @Override  public void buyMac() {  //创建真实对象实例，即“我”  RealSubject realSubject = new RealSubject();   //调用真实对象的方法，进行代理购买Mac  realSubject.buyMac();  //代理对象额外做的操作  this.wrapMac();  }   private void wrapMac() {  System.*out*.println("用盒子包装好Mac");  } } |

###### **步骤4：**客户端调用

|  |
| --- |
| //客户端 public class Client {   public static void main(String[] args) {  Subject proxy = new Proxy();  proxy.buyMac();  } } |

###### 结果：

图形用户界面, 文本, 网站

中度可信度描述已自动生成

#### 《3》 示例2

<https://time.geekbang.org/column/article/201823>

### 【2】 写法2：代理类继承真实类

<https://time.geekbang.org/column/article/201823>

#### 《1》 前提

参照基于接口而非实现编程的设计思想，将原始类对象替换为代理类对象的时候，为了让代码改动尽量少，在刚刚的代理模式的代码实现中，代理类和原始类需要实现相同的接口。但是，如果原始类并没有实现接口，并且原始类代码并不是我们开发维护的（比如它来自一个第三方的类库），我们也没办法直接修改原始类，让它重新实现一个接口。在这种情况下，我们该如何实现代理模式呢？

对于这种外部类的扩展，我们一般都是采用继承的方式。这里也不例外。我们让代理类继承原始类，然后扩展附加功能。

#### 《2》 示例

|  |
| --- |
| // UserControllerProxy为**代理类**，UserController为**真实/原始类**  public class UserControllerProxy extends UserController {  private MetricsCollector metricsCollector;  public UserControllerProxy() {  this.metricsCollector = new MetricsCollector();  }  public UserVo login(String telephone, String password) {  long startTimestamp = System.currentTimeMillis();  UserVo userVo = super.login(telephone, password); //主要功能，其上下为扩展功能  long endTimeStamp = System.currentTimeMillis();  long responseTime = endTimeStamp - startTimestamp;  RequestInfo requestInfo = new RequestInfo("login", responseTime, startTimestamp);  metricsCollector.recordRequest(requestInfo);  return userVo;  }  public UserVo register(String telephone, String password) {  long startTimestamp = System.currentTimeMillis();  UserVo userVo = super.register(telephone, password); //主要功能，其上下为扩展功能  long endTimeStamp = System.currentTimeMillis();  long responseTime = endTimeStamp - startTimestamp;  RequestInfo requestInfo = new RequestInfo("register", responseTime, startTimestamp);  metricsCollector.recordRequest(requestInfo);  return userVo;  }  }  //**客户端**：UserControllerProxy使用举例  UserController userController = new UserControllerProxy(); |

# 5. 代理模式（动态代理）

参考：

1】 主：carson <https://www.jianshu.com/p/5dc416ea58a2>

2】 辅：

1》 王争<https://time.geekbang.org/column/article/201823>

2》 

## （1） 目录

图形用户界面, 应用程序

描述已自动生成

## （2） 为什么要用动态代理

### 【1】 背景

代理模式中的**静态代理模式**存在一些特点：

* 1个静态代理 只服务1种类型的目标对象
* 若要服务多类型的目标对象，则需要为每种目标对象都实现一个静态代理对象

### 【2】 冲突

在目标对象较多的情况下，若采用静态代理，则会出现 **静态代理对象量多、代码量大，从而导致代码复杂的问题**

### 【3】 解决方案

采用 **动态代理模式**。

## （3） 动态代理模式介绍

### 【1】 实现原理（jdk）

#### 参1：<https://time.geekbang.org/column/article/201823>

如果你熟悉的是 Java 语言，实现动态代理就是件很简单的事情。因为 Java 语言本身就已经提供了动态代理的语法（实际上，动态代理底层依赖的就是 Java 的反射语法）。

#### 参2：<https://www.jianshu.com/p/5dc416ea58a2>

* 设计动态代理类（DynamicProxy）时，**不需要显式实现与目标对象类（RealSubject）相同的接口**，而是将这种实现推迟到程序运行时由 JVM来实现

|  |
| --- |
| 1. 即：在使用时再创建动态代理类 & 实例 2. 静态代理则是在代理类实现时就指定与目标对象类（RealSubject）相同的接口 |

* 通过Java 反射机制的method.invoke（），通过调用动态代理类对象方法，从而自动调用目标对象的方法

#### 参3：<https://www.jianshu.com/p/ad902069e5c3>

利用Java的反射技术(Java Reflection)，在运行时创建一个实现某些给定接口的新类（也称“动态代理类”）及其实例（对象）。

### 【2】 优点

* 只需要1个动态代理类就可以解决创建多个静态代理的问题，避免重复、多余代码
* 更强的灵活性

|  |
| --- |
| 1. 设计动态代理类（DynamicProxy）时，**不需要显式实现与目标对象类（RealSubject）相同的接口**，而是将这种实现推迟到程序运行时由 JVM来实现  2. 在使用时（调用目标对象方法时）才会动态创建动态代理类 & 实例，不需要事先实例化 |

### 【3】 缺点

 效率低  
相比静态代理中 **直接调用目标对象方法**，动态代理则需要先通过Java反射机制 从而 **间接调用目标对象方法**

 应用场景局限  
因为 Java 的单继承特性（每个代理类都继承了 Proxy 类），即只能**针对接口** 创建 代理类，**不能针对类** 创建代理类

|  |
| --- |
| 即只能动态代理 **实现了接口**的类 |

### 【4】 应用场景

* 基于静态代理应用场景下，需要代理对象数量较多的情况下使用动态代理
* AOP 领域

|  |
| --- |
| 1. 定义：即 Aspect Oriented Programming = 面向切面编程，是OOP的延续、函数式编程的一种衍生范型  2. 作用：通过预编译方式和运行期动态代理实现程序功能的统一维护。  3. 优点：降低业务逻辑各部分之间的耦合度 、 提高程序的可重用性 & 提高了开发的效率  4. 具体应用场景：日志记录、性能统计、安全控制、异常处理等 |

### 【5】 与静态代理模式的区别

表格

描述已自动生成

## （4） 示例

接下来，我将用1个具体实例来对 动态代理模式 进行更深一步的介绍。

### 【1】 实例概况

* 背景：小成 希望买一台最新的顶配 Mac 电脑；小何希望买一台 iPhone
* 冲突：国内还没上，只有美国才有
* 解决方案：寻找一个代购一起进行购买

|  |
| --- |
| 1. 即1个代购（动态代理对象）同时 代替 小成 & 小何（目标对象） 去买Mac（间接访问的操作） 2. 该代购是代购任何商品 = 什么人有什么需求就会去代购任何东西（动态代理） |

### 【2】 使用步骤

#### 步骤1：声明 目标对象类实现的抽象接口

|  |
| --- |
| //目标对象类实现的抽象接口 public interface Subject {   public void buy(); } |

#### 步骤2：声明 目标对象类

*Buyer1.java*

|  |
| --- |
| //目标对象类 public class Buyer1 implements Subject {   @Override  public void buy() {  System.*out*.println("小成要买Mac");  } } |

*Buyer2.java*

|  |
| --- |
| //目标对象类 public class Buyer2 implements Subject {   @Override  public void buy() {  System.*out*.println("小何要买iPhone");  } } |

#### 步骤3：声明 调用处理器类

|  |
| --- |
| //调用处理器类 public class DynamicProxyHandler implements InvocationHandler {   private Object mProxiedObject;   public DynamicProxyHandler(Object proxiedObject) {  mProxiedObject = proxiedObject;  }   @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  //调目标对象方法之前  System.*out*.println("代购出门了");  Object result = null;  // 通过Java反射机制调用目标对象方法  result = method.invoke(mProxiedObject, args);  //调目标对象方法之前  System.*out*.println("代购买完了");  return result;  } } |

#### 步骤4：通过动态代理对象，调用目标对象的方法

|  |
| --- |
| //客户端 public class Client {   public static void main(String[] args) {  // 1. 创建目标类对象  Buyer1 buyer1 = new Buyer1();   // 2. 创建调用处理器类对象  DynamicProxyHandler dynamicProxyHandler1 = new DynamicProxyHandler(buyer1);   // 3. 利用反射创建动态代理类 & 对象  Subject buyer1\_DynamicProxy = (Subject) Proxy.*newProxyInstance*(buyer1.getClass().getClassLoader(), buyer1.getClass().getInterfaces(), dynamicProxyHandler1);   // 4. 通过调用动态代理对象的方法从而调用目标对象的方法  // 实际上是调用了”调用处理器类对象“的invoke()，再通过该invoke()里的反射机制调用目标对象的方法  buyer1\_DynamicProxy.buy();  // 以上是代购为小成代购Mac   System.*out*.println("------------");  // 以下是代购为小何代购iPhone  Buyer2 buyer2 = new Buyer2();  DynamicProxyHandler dynamicProxyHandler2 = new DynamicProxyHandler(buyer2);  Subject buyer2\_DynamicProxy = (Subject) Proxy.*newProxyInstance*(buyer2.getClass().getClassLoader(), buyer2.getClass().getInterfaces(), dynamicProxyHandler2);  buyer2\_DynamicProxy.buy();  } } |

#### 结果

文本

描述已自动生成

## （5） 源码分析（版本一：carson）

<https://www.jianshu.com/p/5dc416ea58a2>

### 【1】 概述

### 【2】 （关注1）创建动态代理类&对象：通过调用处理器类对象newProxyInstance（）

### 【3】 （关注2）通过调用动态代理对象方法从而调用目标对象方法

### 【4】 原理总结

## （5） 源码分析（版本二：自己总结）

基于jdk中提供的编程接口实现动态代理的核心是3部分：InvocationHandler接口，Proxy.newProxyInstance方法，Method.invoke方法。

### InvocationHandler接口

<https://www.jianshu.com/p/ad902069e5c3>

<https://www.jianshu.com/p/5dc416ea58a2>

<https://www.mianshigee.com/tutorial/android_interview/java-basis-reflection2.md>

源码：

文本

描述已自动生成

该接口只有一个抽象方法invoke。

invoke的作用就是在动态代理对象中的指定方法m1被调时会先调被实现的该invoke方法，然后在该invoke方法中调Method.invoke()完成对目标对象指定方法m1的调用。

invoke的3个参数：

* proxy：动态代理对象【即哪个动态代理对象调了method】【即Proxy.newProxyInstance方法返回的对象】
* method：动态代理对象调用的方法，即目标对象需被调用的方法
* args：被调用方法method的参数

### Proxy.newProxyInstance方法

* 源码：

图形用户界面, 文本

描述已自动生成

* 作用：根据3个参数生成动态代理对象Object
* 参数：

<https://www.jianshu.com/p/5dc416ea58a2>

<https://www.jianshu.com/p/ad902069e5c3>

* loader：产生代理对象的类加载器。需要将其指定为和目标对象同一个类加载器。通过Class.getClassLoader()获取。
* interfaces：动态代理类需要实现的接口（们）。通过Class.getInterfaces()获取，见图形用户界面, 应用程序, 电子邮件

  描述已自动生成。
* h：和动态代理对象关联的InvocationHandler对象。调用动态代理对象中的方法时，会自动调用h里面的invoke方法，然后在实现的invoke方法中反射调用目标对象的方法。
* 原理：

<https://blog.jrwang.me/2015/java-dynamic-proxy-and-java-reflection/>

图形用户界面, 文本, 应用程序

描述已自动生成

### Method.invoke方法

参考：

1】 

2】 

源码：

文本

描述已自动生成

作用：调用obj对象中的该方法【即this】，并传入实参args【若该方法无参数，则不传】，得到返回结果Object。

## （6） 总结

我用两张图总结整篇文章的内容

图形用户界面

描述已自动生成

表格

描述已自动生成

## （7） 补充

【1】 java中除了用jdk实现动态代理，还能用cglib【了解即可】

<https://blog.csdn.net/u010066934/article/details/104087121>

<https://time.geekbang.org/column/article/201823> 图片包含 图示

描述已自动生成

# 5. 代理模式（实例）

（1） Android Binder

见“oppo二级认证课程证书-设计模式 试题”

（2） Retrofit【动态代理】

# 6. 观察者模式（概述）

参考：

1】 主：<https://blog.csdn.net/itachi85/article/details/50773358>

2】 辅：王争课<https://time.geekbang.org/column/article/210170>

## （1） 定义

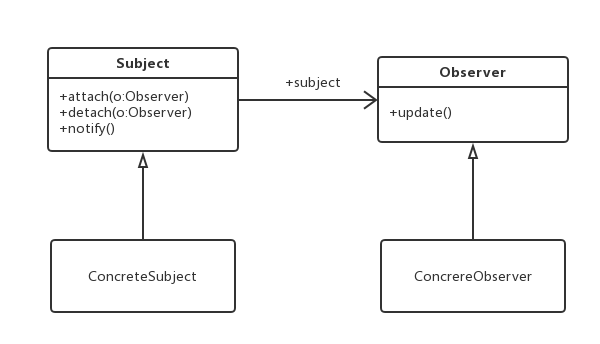
观察者模式（又被称为发布-订阅：Publish-Subscribe模式【被观察者对应发布者，观察者对应订阅者】），属于行为型模式的一种，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

## （2） 结构图

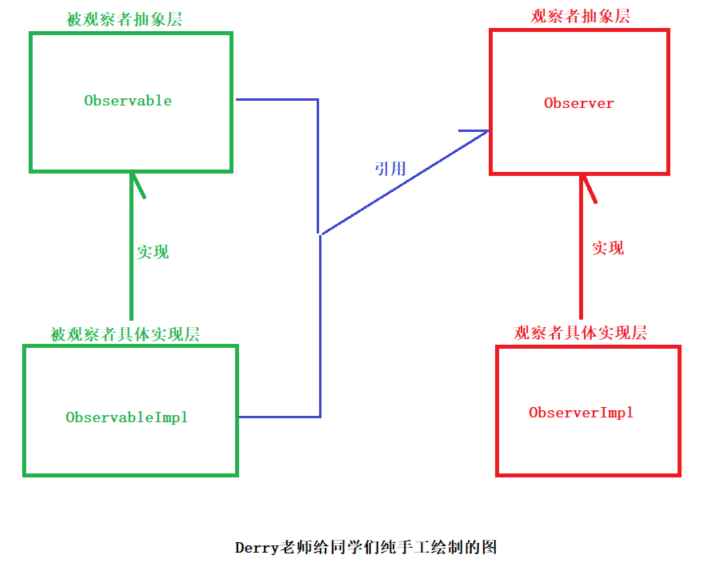
### 第一种：

参考：

1】 <https://blog.csdn.net/itachi85/article/details/50773358>



或



在观察者模式中有如下角色：

* Subject：抽象主题（抽象被观察者），抽象主题角色把所有观察者对象保存在一个集合里，每个主题都可以有任意数量的观察者，抽象主题提供一个接口，可以增加和删除观察者对象。
* ConcreteSubject：具体主题（具体被观察者），该角色将有关状态存入具体观察者对象，在具体主题的内部状态发生改变时，给所有注册过的观察者发送通知。
* Observer：抽象观察者，是具体观察者的抽象类，它定义了一个更新接口，使得在得到主题更改通知时更新自己。
* ConcrereObserver：具体观察者，实现抽象观察者定义的更新接口，以便在得到主题更改通知时更新自身的状态。

### 第二种：

<https://www.runoob.com/design-pattern/observer-pattern.html>

<https://time.geekbang.org/column/article/210170>

观察者模式使用三个类 Subject、Observer 和 Client。Subject 对象带有绑定观察者到 Client 对象和从 Client 对象解绑观察者的方法。我们创建 *Subject* 类、*Observer* 抽象类和扩展了抽象类 *Observer* 的实体类。

图示

描述已自动生成

## （3） 示例

### 例1：

观察者模式这种发布-订阅的形式我们可以拿微信公众号来举例，假设微信用户就是观察者，微信公众号是被观察者，有多个微信用户关注了程序猿这个公众号，当这个公众号更新时就会通知这些订阅的微信用户。好了我们来看看用代码如何实现：

#### 【1】 抽象观察者（Observer）

里面定义了一个更新的方法：

|  |
| --- |
| public interface Observer {   public void update(String message); } |

#### 【2】 具体观察者（ConcreteObserver）

微信用户是观察者，里面实现了更新的方法：

|  |
| --- |
| public class WeixinUser implements Observer {   private String name; //微信用户名   public WeixinUser(String name) {  this.name = name;  }   @Override  public void update(String message) {  System.*out*.println(name + "-" + message);  } } |

#### 【3】 抽象被观察者（Subject）

抽象主题，提供了attach、detach、notify三个方法：

|  |
| --- |
| public interface Subject {   */\*\*  \* 增加订阅者  \** ***@param*** *observer  \*/* public void attach(Observer observer);   */\*\*  \* 删除订阅者  \** ***@param*** *observer  \*/* public void detach(Observer observer);   */\*\*  \* 通知订阅者更新消息  \** ***@param*** *message  \*/* public void notify(String message); } |

#### 【4】 具体被观察者（ConcreteSubject）

微信公众号是具体主题（具体被观察者），里面存储了订阅该公众号的微信用户 并 实现了抽象主题中的方法：

|  |
| --- |
| public class SubscriptionSubject implements Subject {   //储存订阅公众号的微信用户  private List<Observer> weixinUserList = new ArrayList<>();   @Override  public void attach(Observer observer) {  weixinUserList.add(observer);  }   @Override  public void detach(Observer observer) {  weixinUserList.remove(observer);  }   @Override  public void notify(String message) {  for (Observer observer : weixinUserList) {  observer.update(message);  }  } } |

#### 【5】 client调用

|  |
| --- |
| public class Client {   public static void main(String[] args) {  SubscriptionSubject subscriptionSubject = new SubscriptionSubject();  //创建微信用户  WeixinUser user1 = new WeixinUser("杨影枫");  WeixinUser user2 = new WeixinUser("月眉儿");  WeixinUser user3 = new WeixinUser("紫轩");  //订阅公众号  subscriptionSubject.attach(user1);  subscriptionSubject.attach(user2);  subscriptionSubject.attach(user3);  //公众号更新，发出消息给订阅的微信用户  subscriptionSubject.notify("刘望舒的专栏更新了");  } } |

结果：

文本

描述已自动生成

#### 注：

该示例中抽象被观察者Subject未被用到，所以可以不用定义，即结构图中的第二种。

### 其他示例：

<https://www.runoob.com/design-pattern/observer-pattern.html>

<https://juejin.cn/post/6844904100459446285>

# 6. 观察者模式（优缺点+场景）

## （1） 场景

参考1：<https://blog.csdn.net/itachi85/article/details/50773358>

* 关联行为场景，需要注意的是，关联行为是可拆分的，而不是“组合”关系。
* 事件多级触发场景。
* 跨系统的消息交换场景，如消息队列、事件总线的处理机制。

参考2：<https://juejin.cn/post/6844903924718108686>

公众号推送消息（消息的订阅和推送）：用户是观察者，公众号是被观察者，用户可以离开加入公众 号的订阅列表。

参考3：https://refactoringguru.cn/design-patterns/observer

**当一个对象状态的改变需要改变其他对象， 或实际对象是事先未知的或动态变化的时， 可使用观察者模式。**

当你使用图形用户界面类时通常会遇到一个问题。 比如， 你创建了自定义按钮类并允许客户端在按钮中注入自定义代码， 这样当用户按下按钮时就会触发这些代码。

观察者模式允许任何实现了订阅者接口的对象订阅发布者对象的事件通知。 你可在按钮中添加订阅机制， 允许客户端通过自定义订阅类注入自定义代码。

参考4：<https://blog.csdn.net/daidaineteasy/article/details/103672436>

对一个对象状态的更新，需要其他对象同步更新，而且其他对象的数量动态可变。

如，用户支付成功一个订单时，会触发这些操作：

* 记录文本日志
* 发送短信消息
* 赠送活动优惠券

并且以后还会扩展一些其它的操作，这时比较适合用观察者模式。

参考5：<https://zhuanlan.zhihu.com/p/263657828>

* 一个对象的数据或状态更新需要其它对象同步更新时；
* 系统存在事件多级触发时；
* 一个对象仅需要将自己的更新通知给其它对象而不需要知道其它对象细节时，如消息推送；
* 跨系统的消息交换场景，如通信过程中的消息队列处理机制。

参考6：<http://c.biancheng.net/view/1390.html>

在软件系统中，当系统一方行为依赖另一方行为的变动时，可使用观察者模式松耦合联动双方，使得一方的变动可以通知到感兴趣的另一方对象，从而让另一方对象对此做出响应。

通过前面的分析与应用实例可知观察者模式适合以下几种情形。

1. 对象间存在一对多关系，一个对象的状态发生改变会影响其他对象。
2. 当一个抽象模型有两个方面，其中一个方面依赖于另一方面时，可将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。
3. 实现类似广播机制的功能，不需要知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播。
4. 多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知。

## 优点

参考1：<https://blog.csdn.net/itachi85/article/details/50773358>

解除耦合，让耦合的双方都依赖于抽象，从而使得各自的变换都不会影响另一边的变换。

## 缺点

参考1：<https://blog.csdn.net/itachi85/article/details/50773358>

在应用观察者模式时需要考虑一下开发效率和运行效率的问题，程序中包括一个被观察者、多个观察者，开发、调试等内容会比较复杂，而且在Java中消息的通知一般是顺序执行，那么一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般会采用异步实现。

# 6. 观察者模式（实例）

## 参考1：<https://blog.csdn.net/itachi85/article/details/50773358>

### 【1】 概述

android源码中也有很多使用了观察者模式，比如OnClickListener、ContentObserver、

android.database.Observable等；还有组件通讯库RxJava、RxAndroid、EventBus；

### 【2】 Adapter

在这里将拿我们最常用的Adapter的notifyDataSetChanged()方法来举例：

当我们用ListView的时候，数据发生变化的时候我们都会调用Adapter的notifyDataSetChanged()方法，这个方法定义在BaseAdaper中，我们来看看BaseAdapter的部分源码：

|  |
| --- |
| public abstract class BaseAdapter implements ListAdapter, SpinnerAdapter {  //数据集观察者  private final DataSetObservable mDataSetObservable = new DataSetObservable();  public boolean hasStableIds() {  return false;  }  public void registerDataSetObserver(DataSetObserver observer) {  mDataSetObservable.registerObserver(observer);  }  public void unregisterDataSetObserver(DataSetObserver observer) {  mDataSetObservable.unregisterObserver(observer);  }  /\*\*  \* 当数据集变化时，通知所有观察者  \*/  public void notifyDataSetChanged() {  mDataSetObservable.notifyChanged();  } |

很明显BaseAdapter用的是观察者模式，BaseAdapter是具体被观察者，接下来看看mDataSetObservable.notifyChanged()：

|  |
| --- |
| public class DataSetObservable extends Observable<DataSetObserver> {  public void notifyChanged() {  synchronized(mObservers) {  for (int i = mObservers.size() - 1; i >= 0; i--) {  mObservers.get(i).onChanged();  }  }  } |

我们看到了mObservers，这就是观察者的集合，这些观察者是在ListView通过setAdaper()设置Adaper时产生的：

|  |
| --- |
| @Override  public void setAdapter(ListAdapter adapter) {  if (mAdapter != null && mDataSetObserver != null) {  mAdapter.unregisterDataSetObserver(mDataSetObserver);  }  ...  super.setAdapter(adapter);  if (mAdapter != null) {  mAreAllItemsSelectable = mAdapter.areAllItemsEnabled();  mOldItemCount = mItemCount;  mItemCount = mAdapter.getCount();  checkFocus();  //创建数据观察者  mDataSetObserver = new AdapterDataSetObserver();  //注册观察者  mAdapter.registerDataSetObserver(mDataSetObserver);  ...  } |

接下来看看观察者AdapterDataSetObserver中处理了什么：

|  |
| --- |
| class AdapterDataSetObserver extends AdapterView<ListAdapter>.AdapterDataSetObserver {  @Override  public void onChanged() {  super.onChanged();  if (mFastScroller != null) {  mFastScroller.onSectionsChanged();  }  }  @Override  public void onInvalidated() {  super.onInvalidated();  if (mFastScroller != null) {  mFastScroller.onSectionsChanged();  }  }  } |

从上面的代码看不出什么，再看看AdapterDataSetObserver的父类AdapterView的AdapterDataSetObserver：

|  |
| --- |
| class AdapterDataSetObserver extends DataSetObserver {  private Parcelable mInstanceState = null;  @Override  public void onChanged() {  mDataChanged = true;  mOldItemCount = mItemCount;  mItemCount = getAdapter().getCount();  if (AdapterView.this.getAdapter().hasStableIds() && mInstanceState != null  && mOldItemCount == 0 && mItemCount > 0) {  AdapterView.this.onRestoreInstanceState(mInstanceState);  mInstanceState = null;  } else {  rememberSyncState();  }  checkFocus();  //重新布局  requestLayout();  }  ...  public void clearSavedState() {  mInstanceState = null;  }  } |

我们看到在onChanged()方法中调用了requestLayout()方法来重新进行布局。

好了，看到这里我们都明白了，当ListView的数据发生变化时，我们调用Adapter的notifyDataSetChanged()方法，这个方法又会调用观察者们（AdapterDataSetObserver）的onChanged()方法，onChanged()方法又会调用requestLayout()方法来重新进行布局。

### 【3】 Observer+Observable

参考：

1】 <https://www.jianshu.com/p/13931cb40de2>

2】 <https://developer.android.com/reference/android/database/DataSetObserver#inherited-methods>

3】 <https://developer.android.com/reference/android/database/DataSetObservable>

## 参考2：

<https://blog.csdn.net/swengineer/article/details/6268244>

## 参考3：<https://zhuanlan.zhihu.com/p/20540213>

## 参考4：<https://www.52pojie.cn/thread-1415435-1-1.html>

## 参考5：<https://time.geekbang.org/column/article/210170>

邮件订阅、RSS Feeds

## 参考6：<http://imhuchao.com/1500.html>

## 其他实例：

（1） Jetpack databinding

（2） android 广播

<https://blog.csdn.net/carson_ho/article/details/52973504>

（3） Rxjava【非标准观察者】

（4） EventBus