课程目标

1、掌握享元模式和组合模式的应用场景。

2、了解享元模式的内部状态和外部状态。

3、掌握组合模式的透明写法和安全写法。

4、享元模式和组合模式的的优缺点。

内容定位

适合有项目开发经验的人群。

享元模式

面向对象技术可以很好地解决一些灵活性或可扩展性问题，但在很多情况下需要在系统中增加类和对象的个数。当对象数量太多时，将导致运行代价过高，带来性能下降等问题。享元模式正是为解决这一类问题而诞生的。

享元模式（Flyweight Pattern）又称为轻量级模式，是对象池的一种实现。类似于线程池，线程池可以避免不停的创建和销毁多个对象，消耗性能。提供了减少对象数量从而改善应用所需的对象结构的方式。其宗旨是共享细粒度对象，将多个对同一对象的访问集中起来，不必为每个访问者创建一个单独的对象，以此来降低内存的消耗，属于结构型模式。

原文：Use sharing to support large numbers of fine-grained objects efficiently.

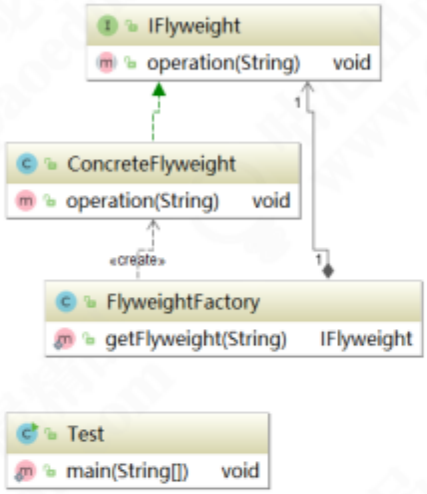
解释：使用共享对象可有效地支持大量的细粒度的对象。

享元模式把一个对象的状态分成内部状态和外部状态，内部状态即是不变的，外部状态是变化的；

然后通过共享不变的部分，达到减少对象数量并节约内存的目的。

享元模式模式的本质是缓存共享对象，降低内存消耗。

首先我们来看享元模式的通用 UML 类图：



从类图上看，享元模式有三个参与角色：

抽象享元角色（Flyweight）：享元对象抽象基类或者接口，同时定义出对象的外部状态和内部状态的接口或实现；

具体享元角色（ConcreteFlyweight）：实现抽象角色定义的业务。该角色的内部状态处理应该与环境无关，不能出现会有一个操作改变内部状态，同时修改了外部状态；

享元工厂（FlyweightFactory）：负责管理享元对象池和创建享元对象。

享元模式的应用场景

当系统中多处需要同一组信息时，可以把这些信息封装到一个对象中，然后对该对象进行缓存，这样，一个对象就可以提供给多处需要使用的地方，避免大量同一对象的多次创建，消耗大量内存空间。

享元模式其实就是工厂模式的一个改进机制，享元模式 同样要求创建一个或一组对象，并且就是通过工厂方法生成对象的，只不过享元模式中为工厂方法增加了缓存这一功能。主要总结为以下应用场景：

1、常常应用于系统底层的开发，以便解决系统的性能问题。

2、系统有大量相似对象、需要缓冲池的场景。

在生活中的享元模式也很常见，比如各中介机构的房源共享，再比如全国社保联网。

各类房源共享

全国社保联网

使用享元模式实现共享池业务

下面我们举个例子，我们每年春节为了抢到一张回家的火车票都要大费周折，进而出现了很多刷票软件，刷票软件会将我们填写的信息缓存起来，然后定时检查余票信息。抢票的时候，我们肯定是要查询下有没有我们需要的票信息，这里我们假设一张火车的信息包含：出发站，目的站，价格，座位类别。现在要求编写一个查询火车票查询伪代码，可以通过出发站，目的站查到相关票的信息。

比如要求通过出发站，目的站查询火车票的相关信息，那么我们只需构建出火车票类对象，然后提供一个查询出发站，目的站的接口给到客户进行查询即可，具体代码如下，创建 ITicket 接口：

```
public interface ITicket {
    void showInfo(String bunk);
}
```

然后，创建 TrainTicket 接口：

```
public class TrainTicket implements ITicket {
    private String from;
    private String to;
    private int price;

    public TrainTicket(String from, String to) {
        this.from = from;
        this.to = to;
    }


    public void showInfo(String bunk) {
        this.price = new Random().nextInt(500);
        System.out.println(String.format("%s->%s: %s 价格：%s 元", this.from, this.to, bunk, this.price));
    }
}
```

最后创建 TicketFactory 类：

```
public static class TicketFactory {
```

```
    public static ITicket queryTicket(String from, String to) {
        return new TrainTicket(from, to);
    }
}
```

编写客户端代码：

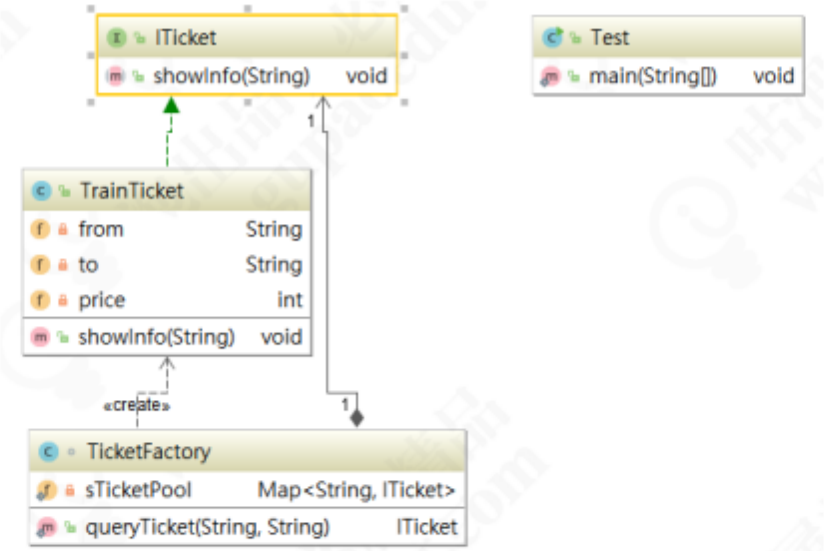
```
public static void main(String[] args) {
       ITicket ticket = TicketFactory.queryTicket("深圳北", "潮汕");
       ticket.showInfo("硬座");
}
```

分析上面的代码，我们发现客户端进行查询时，系统通过 TicketFactory 直接创建一个火车票对象，但是这样做的话，当某个瞬间如果有大量的用户请求同一张票的信息时，系统就会创建出大量该火车票对象，系统内存压力骤增。而其实更好的做法应该是缓存该票对象，然后复用提供给其他查询请求，这样一个对象就足以支撑数以千计的查询请求，对内存完全无压力，使用享元模式可以很好地解决这个问题。我们继续优化代码，只需在 TicketFactory 类中进行更改，增加缓存机制：

```
class TicketFactory {
   private static Map<String, ITicket> sTicketPool = new ConcurrentHashMap<String,ITicket>();

   public static ITicket queryTicket(String from, String to) {
       String key = from + "->" + to;
       if (TicketFactory.sTicketPool.containsKey(key)) {
           System.out.println("使用缓存：" + key);
           return TicketFactory.sTicketPool.get(key);
       }
       System.out.println("首次查询，创建对象：" + key);
       ITicket ticket = new TrainTicket(from, to);
       TicketFactory.sTicketPool.put(key, ticket);
       return ticket;
   }
}
```

运行结果如下：

可以看到，除了第一次查询创建对象后，后续查询相同车次票信息都是使用缓存对象，无需创建新对象了。来看一下类结构图：



其中 ITicket 就是抽象享元角色，TrainTicket 就是具体享元角色，TicketFactory 就是享元工厂。有些小伙伴一定会有疑惑了，这不就是注册式单例模式吗？对，这就是注册式单例模式。虽然，结构上很像，但是享元模式的重点在结构上，而不是在创建对象上。后面看看享元模式在 JDK 源码中的一个应用，大家应该就能彻底清除明白了。

再比如，我们经常使用的数据库连接池，因为我们使用 Connection 对象时主要性能消耗在建立连接和关闭连接的时候，为了提高 Connection 在调用时的性能，我们和将 Connection 对象在调用前创建好缓存起来，用的时候从缓存中取值，用完再放回去，达到资源重复利用的目的。来看下面的代码：

```
public class ConnectionPool {

    private Vector<Connection> pool;

    private String url = "jdbc:mysql://localhost:3306/test";
    private String username = "root";
    private String password = "root";
    private String driverClassName = "com.mysql.jdbc.Driver";
    private int poolSize = 100;

    public ConnectionPool() {
        pool = new Vector<Connection>(poolSize);

        try{
            Class.forName(driverClassName);
            for (int i = 0; i < poolSize; i++) {
                Connection conn = DriverManager.getConnection(url,username,password);
                pool.add(conn);
            }
        }catch (Exception e){
            e.printStackTrace();
        }
```

```
    }

    public synchronized Connection getConnection(){
        if(pool.size() > 0){
            Connection conn = pool.get(0);
            pool.remove(conn);
            return conn;
        }
        return null;
    }

    public synchronized void release(Connection conn){
        pool.add(conn);
    }
}
```

这样的连接池，普遍应用于开源框架，有效提升底层的运行性能。

享元模式在源码中的应用

1、String 中的享元模式

Java 中将 String 类定义为 final（不可改变的），JVM 中字符串一般保存在字符串常量池中，java 会确保一个字符串在常量池中只有一个拷贝，这个字符串常量池在 JDK6.0 以前是位于常量池中，位于永久代，而在 JDK7.0 中，JVM 将其从永久代拿出来放置于堆中。

我们做一个测试：

```
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        String s3 = "he" + "llo";
        String s4 = "hel" + new String("lo");
        String s5 = new String("hello");
        String s6 = s5.intern();
        String s7 = "h";
        String s8 = "ello";
        String s9 = s7 + s8;
        System.out.println(s1==s2);//true
        System.out.println(s1==s3);//true
        System.out.println(s1==s4);//false
        System.out.println(s1==s9);//false
        System.out.println(s4==s5);//false
        System.out.println(s1==s6);//true
    }
```

String 类的 final 修饰的，以字面量的形式创建 String 变量时，JVM 会在编译期间就把该字面量"hello"放到字符串常量池中，由 Java 程序启动的时候就已经加载到内存中了。这个字符串常量池的特点就是有且只有一份相同的字面量，如果有其它相同的字面量，JVM 则返回这个字面量的引用，如果没有相同的字面量，则在字符串常量池创建这个字面量并返回它的引用。

由于 s2 指向的字面量"hello"在常量池中已经存在了（s1 先于 s2），于是 JVM 就返回这个字面量绑定的引用，所以 s1==s2。

s3 中字面量的拼接其实就是"hello"，JVM 在编译期间就已经对它进行优化，所以 s1 和 s3 也是相等的。

s4 中的 new String("lo")生成了两个对象，lo，new String("lo")，lo 存在字符串常量池，new String("lo")存在堆中，String s4 = "hel" + new String("lo")实质上是两个对象的相加，编译器不会进行优化，相加的结果存在堆中，而 s1 存在字符串常量池中，当然不相等。s1==s9 的原理一样。

s4==s5 两个相加的结果都在堆中，不用说，肯定不相等。

s1==s6 中，s5.intern()方法能使一个位于堆中的字符串在运行期间动态地加入到字符串常量池中（字符串常量池的内容是程序启动的时候就已经加载好了），如果字符串常量池中有该对象对应的字面量，则返回该字面量在字符串常量池中的引用，否则，创建复制一份该字面量到字符串常量池并返回它的引用。因此 s1==s6 输出 true。

2、Integer 中的享元模式

再举例一个大家都非常熟悉的对象 Integer，也用到了享元模式，其中暗藏玄机，我们来看个例子：

```
    public static void main(String[] args) {
        Integer a = Integer.valueOf(100);
        Integer b = 100;

        Integer c = Integer.valueOf(1000);
        Integer d = 1000;

        System.out.println("a==b:" + (a==b));
        System.out.println("c==d:" + (c==d));
}
```

大家猜猜看它的运行结果是什么？我们跑完程序之后才发现总有些不对，得到了一个意向不到的结果，其运行结果如下：

之所以得到这样的结果，是因为 Integer 用到的享元模式，我们来看 Integer 的源码：

```
public final class Integer extends Number implements Comparable<Integer> {
...
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
}
...
}
```

我们发现 Integer 源码中的 valueOf()方法做了一个条件判断，如果目标值在-128 到 127 之间，则直接从缓存中取值，否则新建对象。那 JDK 为何要这样做呢？因为在-128 到 127 之间的数据在 int 范围内是使用最频繁的，为了节省频繁创建对象带来的内存消耗，这里就用到了享元模式，来提高性能。

3、Long 中的享元模式

```
public final class Long extends Number implements Comparable<Long> {
    public static Long valueOf(long var0) {
        return var0 >= -128L && var0 <= 127L ? Long.LongCache.cache[(int)var0 + 128] : new Long(var0);
    }
    private static class LongCache {
        private LongCache(){}

        static final Long cache[] = new Long[-(-128) + 127 + 1];

        static {
            for(int i = 0; i < cache.length; i++)
                cache[i] = new Long(i - 128);
        }
    }
    //...
}
```

同理，Long 中也有缓存，不过不能指定缓存最大值。

4、Apache Commons Pool2 中的享元模式

对象池化的基本思路是：将用过的对象保存起来，等下一次需要这种对象的时候，再拿出来重复使用，从而在一定程度上减少频繁创建对象所造成的开销。用于充当保存对象的“容器”的对象，被称为“对象池”（Object Pool，或简称 Pool）。

Apache Commons Pool 实现了对象池的功能。定义了对象的生成、销毁、激活、钝化等操作及其状态转换，并提供几个默认的对象池实现。有几个重要的对象：

PooledObject（池对象）：用于封装对象（如：线程、数据库连接、TCP 连接），将其包裹成可被池管理的对象。

PooledObjectFactory（池对象工厂）：定义了操作 PooledObject 实例生命周期的一些方法，PooledObjectFactory 必须实现线程安全。

ObjectPool （对象池）：ObjectPool 负责管理 PooledObject，如：借出对象，返回对象，校验对象，有多少激活对象，有多少空闲对象。

```
private final Map<IdentityWrapper<T>, PooledObject<T>> allObjects;
```

这里我们就不分析其具体源码了。

享元模式的内部状态和外部状态

享元模式的定义为我们提出了两个要求：细粒度和共享对象。因为要求细粒度对象，所以不可避免地会使对象数量多且性质相近，此时我们就将这些对象的信息分为两个部分：内部状态和外部状态。

内部状态指对象共享出来的信息，存储在享元对象内部并且不会随环境的改变而改变；外部状态指对象得以依赖的一个标记，是随环境改变而改变的、不可共享的状态。

比如，连接池中的连接对象，保存在连接对象中的用户名、密码、连接 url 等信息，在创建对象的时候就设置好了，不会随环境的改变而改变，这些为内部状态。而每个连接要回收利用时，我们需要给它标记为可用状态，这些为外部状态。

享元模式的优缺点

优点:

1、减少对象的创建，降低内存中对象的数量，降低系统的内存，提高效率;

2、减少内存之外的其他资源占用。

缺点：

1、关注内、外部状态、关注线程安全问题；

2、使系统、程序的逻辑复杂化。

组合模式

我们知道古代的皇帝想要管理国家，是不可能直接管理到具体每一个老百姓的，因此设置了很多机构，比如说三省六部，这些机构下面又有很多小的组织。他们共同管理着这个国家。再比如说，一个大公司，下面有很多小的部门，每一个部门下面又有很多个部门。说到底这就是组合模式。

组合模式（Composite Pattern）也称为整体-部分（Part-Whole）模式，它的宗旨是通过将单个对象（叶子节点）和组合对象（树枝节点）用相同的接口进行表示，使得客户对单个对象和组合对象的使用具有一致性，属于结构型模式。

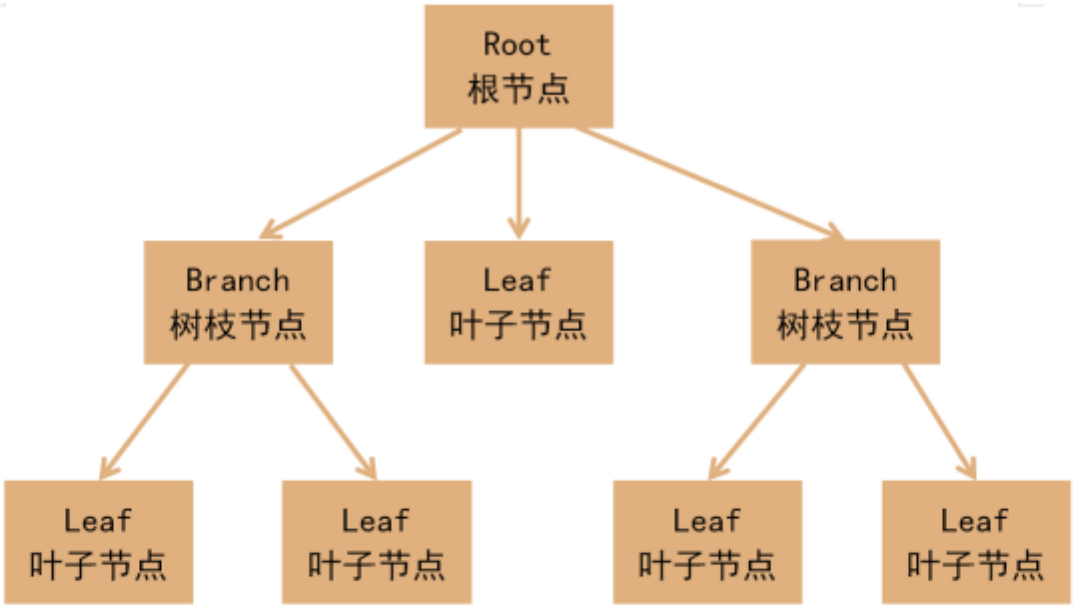
> **原文：** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
>
> **解释：** 将对象组合成树形结构以表示 “部分-整体” 的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

组合关系与聚合关系的区别：

1、组合关系：在古代皇帝三宫六院，贵妃很多，但是每一个贵妃只属于皇帝（具有相同的生命周期）。

2、聚合关系：一个老师有很多学生，但是每一个学生又属于多个老师（具有不同的生命周期）。

组合模式一般用来描述整体与部分的关系，它将对象组织到树形结构中，最顶层的节点称为根节点，根节点下面可以包含树枝节点和叶子节点，树枝节点下面又可以包含树枝节点和叶子节点。如下图所示：



由上图可以看出，其实根节点和树枝节点本质上是同一种数据类型，可以作为容器使用；而叶子节点与树枝节点在语义上不属于同一种类型，但是在组合模式中，会把树枝节点和叶子节点认为是同一种数据类型（用同一接口定义），让它们具备一致行为。这样，在组合模式中，整个树形结构中的对象都是同一种类型，带来的一个好处就是客户无需辨别树枝节点还是叶子节点，而是可以直接进行操作，给客户使用带来极大的便利。

组合模式包含 3 个角色：

1、抽象根节点（Component）：定义系统各层次对象的共有方法和属性，可以预先定义一些默认行为和属性；

2、树枝节点（Composite）：定义树枝节点的行为，存储子节点，组合树枝节点和叶子节点形成一个树形结构；

3、叶子节点（Leaf）：叶子节点对象，其下再无分支，是系统层次遍历的最小单位。

组合模式 在代码具体实现上，有两种不同的方式，分别是透明组合模式和安全组合模式。

组合模式的应用场景

当子系统与其内各个对象层次呈现树形结构时，可以使用组合模式让子系统内各个对象层次的行为操作具备一致性。客户端使用该子系统内任意一个层次对象时，无须进行区分，直接使用通用操作即可，

为客户端的使用带来了便捷。

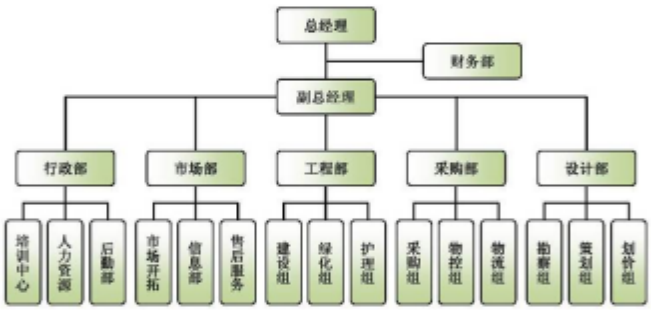
注：如果树形结构系统不使用组合模式进行架构，那么按照正常的思维逻辑，对该系统进行职责分析，按上文树形结构图所示，该系统具备两种对象层次类型：树枝节点和叶子节点。那么我们就需要构造两种对应的类型，然后由于树枝节点具备容器功能，因此树枝节点类内部需维护多个集合存储其他对象层次（如:List<Composite>,List<Leaf>），如果当前系统对象层次更复杂时，那么树枝节点内就又要增加对应的层次集合，这对树枝节点的构建带来了巨大的复杂性，臃肿性以及不可扩展性。同时客户端访问该系统层次时，还需进行层次区分，这样才能使用对应的行为，给客户端的使用也带来了巨大的复杂性。而如果使用组合模式构建该系统，由于组合模式抽取了系统各个层次的共性行为，具体层次只需按需实现所需行为即可，这样子系统各个层次就都属于同一种类型，所以树枝节点只需维护一个集合（List<Component>）即可存储系统所有层次内容，并且客户端也无需区分该系统各个层次对象，对内系统架构简洁优雅，对外接口精简易用。

先对组合模式主要总结为以下应用场景：

1、希望客户端可以忽略组合对象与单个对象的差异时；

2、对象层次具备整体和部分，呈树形结构。

在我们生活中的组合模式也非常常见，比如树形菜单，操作系统目录结构，公司组织架构等。

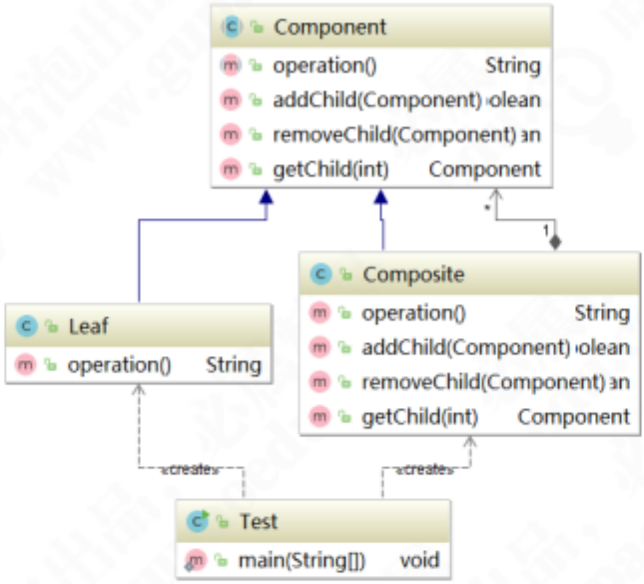


公司组织架构

操作系统的文件管理

透明组合模式的写法

透明组合模式是把所有公共方法都定义在 Component 中，这样做的好处是客户端无需分辨是叶子节点（Leaf）和树枝节点（Composite），它们具备完全一致的接口。其 UML 类图如下所示：



来看一个例子，还是以咕泡的课程为例。这次我们来设计一个课程的关系结构。比如我们有 Java 入门课程、人工智能课程、Java 设计模式、源码分析、软技能等，而 Java 设计模式、源码分析、软技能又属于 Java 架构师系列课程包，每个课程的定价都不一样。但是，这些课程不论怎么组合，都有一些共性，而且是整体和部分的关系，可以用组合模式来设计。先创建一个顶层的抽象组件 CourseComponent 类：

```
/**
 * Created by Tom.
 */
public abstract class CourseComponent {

    public void addChild(CourseComponent catalogComponent){
        throw new UnsupportedOperationException("不支持添加操作");
    }

    public void removeChild(CourseComponent catalogComponent){
        throw new UnsupportedOperationException("不支持删除操作");
    }


    public String getName(CourseComponent catalogComponent){
        throw new UnsupportedOperationException("不支持获取名称操作");
    }


    public double getPrice(CourseComponent catalogComponent){
        throw new UnsupportedOperationException("不支持获取价格操作");
    }


    public void print(){
        throw new UnsupportedOperationException("不支持打印操作");
    }
```

```
}
```

把所有可能用到的方法都定义到这个最顶层的抽象类中，但是不写任何逻辑处理的代码，而是直接抛异常。这里，有些小伙伴会有疑惑，为什么不用抽象方法？因为，用了抽象方法，其子类就必须实现，这样便体现不出各子类的细微差异。因此，子类继承此抽象类后，只需要重写有差异的方法覆盖父类的方法即可。下面我们分别创建课程类 Course 和课程包 CoursePackage 类。先创建 Course 类：

```
/**
 * Created by Tom.
 */
public class Course extends CourseComponent {
    private String name;
    private double price;

    public Course(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String getName(CourseComponent catalogComponent) {
        return this.name;
    }

    @Override
    public double getPrice(CourseComponent catalogComponent) {
        return this.price;
    }

    @Override
    public void print() {
        System.out.println(name + " (¥" + price + "元)");
    }

}
```

再创建 CoursePackage 类：

```
/**
 * Created by Tom.
 */
public class CoursePackage extends CourseComponent {
    private List<CourseComponent> items = new ArrayList<CourseComponent>();
    private String name;
    private Integer level;


    public CoursePackage(String name, Integer level) {
        this.name = name;
        this.level = level;
    }

    @Override
    public void addChild(CourseComponent catalogComponent) {
        items.add(catalogComponent);
    }
```

```
    @Override
    public String getName(CourseComponent catalogComponent) {
        return this.name;
    }

    @Override
    public void removeChild(CourseComponent catalogComponent) {
        items.remove(catalogComponent);
    }

    @Override
    public void print() {
        System.out.println(this.name);

        for(CourseComponent catalogComponent : items){
            //控制显示格式
            if(this.level != null){
                for(int  i = 0; i < this.level; i ++){
                    //打印空格控制格式
                    System.out.print("  ");
                }
                for(int  i = 0; i < this.level; i ++){
                    //每一行开始打印一个+号
                    if(i == 0){ System.out.print("+"); }
                    System.out.print("-");
                }
            }
            //打印标题
            catalogComponent.print();
        }
    }

}
```

来看测试代码：

```
public static void main(String[] args) {

        System.out.println("============透明组合模式===========");

        CourseComponent javaBase = new Course("Java 入门课程",8280);
        CourseComponent ai = new Course("人工智能",5000);

        CourseComponent packageCourse = new CoursePackage("Java 架构师课程",2);

        CourseComponent design = new Course("Java 设计模式",1500);
        CourseComponent source = new Course("源码分析",2000);
        CourseComponent softSkill = new Course("软技能",3000);

        packageCourse.addChild(design);
        packageCourse.addChild(source);
        packageCourse.addChild(softSkill);

        CourseComponent catalog = new CoursePackage("课程主目录",1);
        catalog.addChild(javaBase);
        catalog.addChild(ai);
        catalog.addChild(packageCourse);

        catalog.print();

}
```

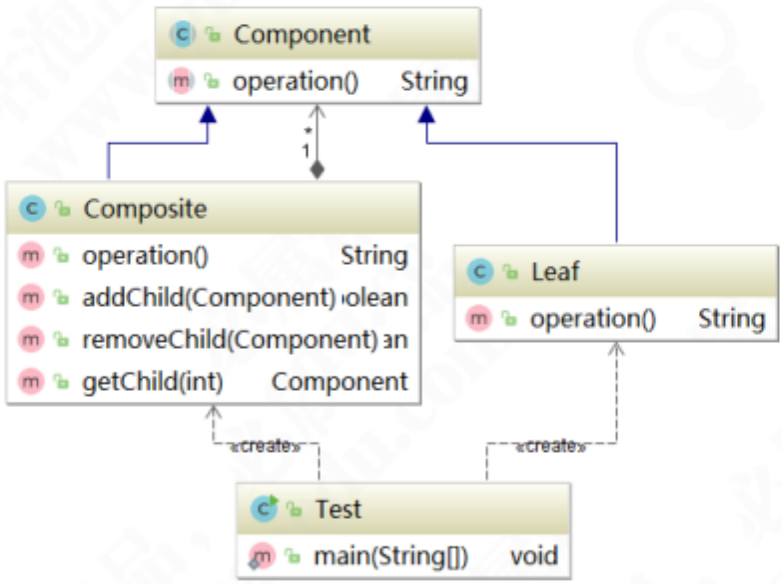
运行结果：

透明组合模式把所有公共方法都定义在 Component 中，这样做的好处是客户端无需分辨是叶子节点（Leaf）和树枝节点（Composite），它们具备完全一致的接口；缺点是叶子节点（Leaf）会继承得到一些它所不需要（管理子类操作的方法）的方法，这与设计模式 接口隔离原则相违背。

为了让大家更加透彻理解，下面我们来看安全组合模式的写法。

安全组合模式的写法

安全组合模式是只规定系统各个层次的最基础的一致行为，而把组合（树节点）本身的方法（管理子类对象的添加，删除等）放到自身当中。其 UML 类图如下所示：



再举一个程序员更熟悉的例子。对于程序员来说，电脑是每天都要接触的。电脑的文件系统其实就是一个典型的树形结构，目录包含文件夹和文件，文件夹里面又可以包含文件夹和文件…下面我们就用

代码来实现一个目录系统。

文件系统有两个大的层次：文件夹，文件。其中，文件夹能容纳其他层次，为树枝节点；文件为最小单位，为叶子节点。由于目录系统层次较少，且树枝节点（文件夹）结构相对稳定，而文件其实可以有很多类型，所以这里我们选择使用 安全组合模式 来实现目录系统，可以避免为叶子类型（文件）引入冗余方法。先创建最顶层的抽象组件 Directory 类：

```
public abstract class Directory {

    protected String name;

    public Directory(String name) {
        this.name = name;
    }

    public abstract void show();

}
```

然后分别创建 File 类和 Folder 类。先看 File 类：

```
public class File extends Directory {

    public File(String name) {
        super(name);
    }

    @Override
    public void show() {
        System.out.println(this.name);
    }

}
```

然后创建 Folder 类。

```
import java.util.ArrayList;
import java.util.List;

public class Folder extends Directory {
    private List<Directory> dirs;

    private Integer level;

    public Folder(String name,Integer level) {
        super(name);
        this.level = level;
        this.dirs = new ArrayList<Directory>();
    }

    @Override
    public void show() {
        System.out.println(this.name);
        for (Directory dir : this.dirs) {
            //控制显示格式
            if(this.level != null){
```

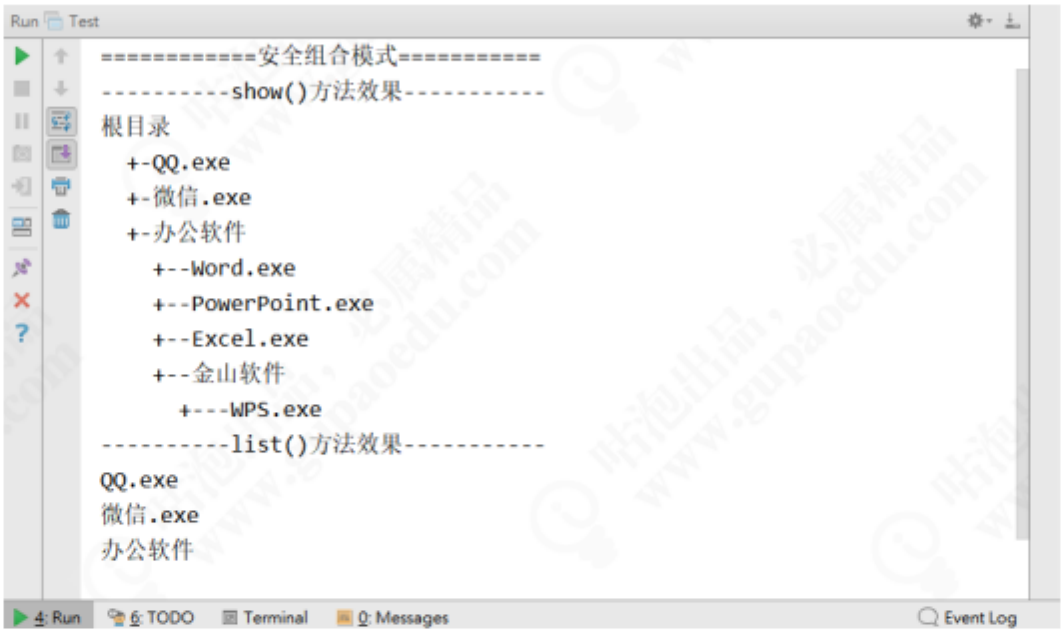
```
            for(int  i = 0; i < this.level; i ++){
                //打印空格控制格式
                System.out.print("  ");
            }
            for(int  i = 0; i < this.level; i ++){
                //每一行开始打印一个+号
                if(i == 0){ System.out.print("+"); }
                System.out.print("-");
            }
        }
        //打印名称
        dir.show();
    }
}

public boolean add(Directory dir) {
    return this.dirs.add(dir);
}

public boolean remove(Directory dir) {
    return this.dirs.remove(dir);
}

public Directory get(int index) {
    return this.dirs.get(index);
}

public void list(){
    for (Directory dir : this.dirs) {
        System.out.println(dir.name);
    }
}

}
```

注意 Folder 类不仅覆盖了顶层的 show()方法，而且还增加了 list()方法。看测试代码：

```
public static void main(String[] args) {

    System.out.println("============安全组合模式===========");

    File qq = new File("QQ.exe");
    File wx = new File("微信.exe");

    Folder office = new Folder("办公软件",2);

    File word = new File("Word.exe");
    File ppt = new File("PowerPoint.exe");
    File excel = new File("Excel.exe");

    office.add(word);
    office.add(ppt);
    office.add(excel);

    Folder wps = new Folder("金山软件",3);
    wps.add(new File("WPS.exe"));
    office.add(wps);

    Folder root = new Folder("根目录",1);
    root.add(qq);
    root.add(wx);
    root.add(office);
```

```
        System.out.println("----------show()方法效果-----------");
        root.show();

        System.out.println("----------list()方法效果-----------");
        root.list();

}
```

运行结果如下：



安全组合模式的好处是接口定义职责清晰，符合设计模式 单一职责原则 和 接口隔离原则；缺点是客户需要区分树枝节点（Composite）和叶子节点（Leaf），这样才能正确处理各个层次的操作，客户端无法依赖抽象（Component），违背了设计模式依赖倒置原则。

组合模式在源码中的应用

组合模式在源码中应用也是非常广泛的。首先我们来看一个非常熟悉的 HashMap，他里面有一个putAll()方法：

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable {
...
    public void putAll(Map<? extends K, ? extends V> m) {
        putMapEntries(m, true);
    }
...
final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
        int s = m.size();
        if (s > 0) {
            if (table == null) { // pre-size
                float ft = ((float)s / loadFactor) + 1.0F;
                int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                        (int)ft : MAXIMUM_CAPACITY);
                if (t > threshold)
```

```
                threshold = tableSizeFor(t);
            }
            else if (s > threshold)
                resize();
            for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
                K key = e.getKey();
                V value = e.getValue();
                putVal(hash(key), key, value, false, evict);
            }
        }
}
...
}
```

我们看到 putAll()方法传入的是 Map 对象，Map 就是一个抽象构件（同时这个构件中只支持键值对的存储格式），而 HashMap 是一个中间构件，HashMap 中的 Node 节点就是叶子节点。说到中间构件就会有规定的存储方式。HashMap 中的存储方式是一个静态内部类的数组 Node<K,V>[] tab，其源码如下：

```
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
        ...
      }

      ...

    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);
        else {
            Node<K,V> e; K k;
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
```

```
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
  ...
```

同理，我们常用的 ArrayList 对象也有 addAll()方法，其参数也是 ArrayList 的父类 Collection，来看源代码：

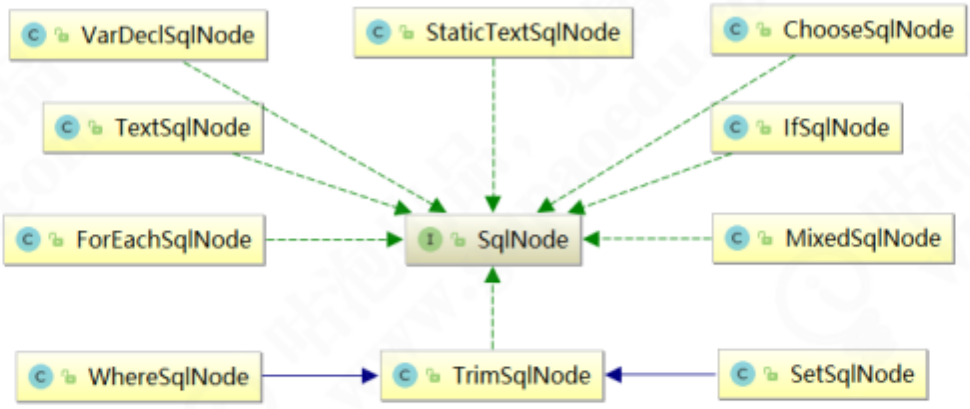
```
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    ...
    public boolean addAll(Collection<? extends E> c) {
        Object[] a = c.toArray();
        int numNew = a.length;
        ensureCapacityInternal(size + numNew);  // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
        return numNew != 0;
    }
      ...
}
```

组合对象和被组合对象都应该有统一的接口实现或者统一的抽象父类。在这里我再举一个开源框架中非常经典的案例，MyBatis 解析各种 Mapping 文件中的 SQL，设计了一个非常关键的类叫做 SqlNode，xml 中的每一个 Node 都会解析为一个 SqlNode 对象，最后把所有的 SqlNode 拼装到一起就成了一条完整的 SQL 语句，它的顶层设计非常简单。来看源代码：

```
public interface SqlNode {
  boolean apply(DynamicContext context);
}
```

Apply()方法会根据传入的参数 context，参数解析该 SqlNode 所记录的 SQL 片段，并调用 DynamicContext.appendSql()方法将解析后的 SQL 片段追加到 DynamicContext.的 sqlBuilder 中保

存。当 SQL 节点下的所有 SqlNode 完成解析后，可以通过 DynamicContext.getSql()获取一条完成的 SQL 语句。对具体源码实现感兴趣的小伙伴可以去研究一下，我们这里给大家展示一下类图：



组合模式的优缺点

很多小伙伴肯定还有个疑问，既然组合模式会被分为两种实现，那么肯定是不同的场合某一种会更加适合，也即具体情况具体分析。透明组合模式将公共接口封装到抽象根节点（Component）中，那么系统所有节点就具备一致行为，所以如果当系统绝大多数层次具备相同的公共行为时，采用透明组合模式也许会更好（代价：为剩下少数层次节点引入不需要的方法）；而如果当系统各个层次差异性行为较多或者树节点层次相对稳定（健壮）时，采用安全组合模式

> 注：设计模式的出现并不是说我们要写的代码一定要遵循设计模式所要求的方方面面，这是不现实同时也是不可能的。设计模式的出现，其实只是强调好的代码所具备的一些特征（六大设计原则），这些特征对于项目开发是具备积极效应的，但不是说我们每实现一个类就一定要全部满足设计模式的要求，如果真的存在完全满足设计模式的要求，反而可能存在过度设计的嫌疑。同时，23 种设计模式，其实都是严格依循设计模式六大原则进行设计，只是不同的模式在不同的场景中会更加适用。设计模式的理解应该重于意而不是形，真正编码时，经常使用的是某种设计模式的变形体，真正切合项目的模式才是正确的模式。

下面我们再来总结一下组合模式的优缺点。

优点：

1、清楚地定义分层次的复杂对象，表示对象的全部或部分层次

2、让客户端忽略了层次的差异，方便对整个层次结构进行控制

3、简化客户端代码

4、符合开闭原则

缺点：

1、限制类型时会较为复杂

2、使设计变得更加抽象