author：黄继升

**一、软件设计模式的关键基本要素**

（1）模式名称

（2）问题

（3）解决方案

（4）效果

**二、GoF的23种设计模式**

设计模式有两种分类方法：

（1）根据模式的目的来分

创建型模式：用于描述“怎样创建对象”，特点是“将对象的创建与使用分离”。有5种创建型模式：单例、原型、工厂方法、抽象工厂、建造者。

结构型模式：用于描述“如何将类或对象按某种布局组成更大的结构”。有7种结构型模式：代理、适配器、桥接、装饰、外观、享元、组合

行为型模式：用于描述“类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责”。有11种行为型模式：模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器。

（2）根据模式的作用范围来分

类模式：用于处理类与子类之间的关系，这些关系通过继承来建立，是静态的，在编译时刻便确定下来了。有4种类模式：工厂方法、（类）适配器、模板方法、解释器

对象模式：用于处理对象之间的关系，这些关系可以通过组合或聚合来实现，在运行时刻是可以变化的，更具动态性。

目的

|  |  |  |  |
| --- | --- | --- | --- |
| 范围 | 创建型模式 | 结构型模式 | 行为型模式 |
| 类模式 | **工厂方法** | **（类）适配器** | **模板方法**  解释器 |
| 对象模式 | **单例**  **原型**  **抽象工厂**  **建造者** | **代理**  **（对象）适配器**  **桥接**  **装饰**  **外观**  享元  **组合** | **策略**  **命令**  **职责链**  **状态**  **观察者**  中介者  **迭代器**  访问者  备忘录 |

**三、统一建模语言UML**

（1）类、接口和类图

类的UML表示：

类名 +

属性集合 {[可见性]属性名:类型[=默认值]，…} +

操作集合（成员方法集合）{[可见性]名称(参数列表)[:返回类型]}

可见性：public + 、 private - 、 protected # 、 friendly ~

eg：学生类的UML表示：

|  |
| --- |
| 学生  Student |
| -no:long  -name:String  -school:String  -totalScore:float |
| +display():void |

接口的UML表示：

eg：Graph接口的UML表示：

Graph

+getArea():double

+getPerimeter():double

类图是用来表示系统中的类、接口、协作以及它们之间的静态结构和关系的一种静态模型，用于描述软件系统的结构化设计。

|  |
| --- |
| 圆形  Circular |
| -radius:double |
| +getArea():double  +getPerimeter():double |

|  |
| --- |
| 访问类  Client |
|  |
| +calculate(Graph tx):void |

Graph

+getArea():double

+getPerimeter():double

|  |
| --- |
| 长方形  Rectangle |
| -length:double  -width:double |
| +getArea():double  +getPerimeter():double |

（2）类之间的关系 （按照类与类之间的耦合度从弱到强排列）

1、依赖关系

一种使用关系，耦合度最弱。在代码中，某个类的方法通过局部变量、方法参数或者对静态方法的调用来访问被依赖类中的某些方法来完成功能。

|  |
| --- |
| 人  Person |
| -name: String |
| +call (MobilePhone mp):void |

|  |
| --- |
| 手机  MobilePhone |
|  |
| +transfer() : void |

打电话

2、关联关系

表示对象之间的一种引用关系，用于表示一类对象与另一类对象之间的联系，如老师和学生，丈夫和妻子等。分为一般关联关系、聚合关联关系、组合关联关系。关联可以是双向的，也可以是单向的。双向关联用两个箭头或不带箭头的实现表示，单向关联用一个箭头实线表示，箭头从使用类指向被关联的类。

在代码中通常将一个类的对象作为另一个类的成员变量来实现关联关系。

|  |
| --- |
| 老师  Teacher |
| -name: String  教学者  -stus：List<Student> |
| +teaching ():void |

|  |
| --- |
| 学生  Student |
| -name: String  -teas: List<Teacher> |
| +study() : void |

teaches to

学习者

3、聚合关系

聚合关系属强关联关系，是整体与部分之间的关系，是has - a的关系。聚合关系通过成员对象来实现，成员对象是整体对象的一部分，但是成员对象可以脱离整体对象而独立存在。

UML类图中用带空心菱形实现表示，菱形指向整体。

|  |
| --- |
| 教师  Teacher |
| -name:String |
| +teaching():void |

|  |
| --- |
| 大学  University |
| -teas:List<Teacher> |
|  |

4、组合关系

组合关系属更强的关联关系，是一种更强烈的聚合关系，也表示类的整体与部分的关系，是cxmtains – a关系。在组合关系中，整体对象可以控制部分对象的生命周期，一旦整体对象不存在，部分对象也将不存在，部分对象不能脱离整体对象而存在。

在UML类图中，组合关系用带实心菱形的实线表示，菱形指向整体。

|  |
| --- |
| 头  Head |
| -mouth:Mouth |
|  |

|  |
| --- |
| 嘴  Mouth |
|  |
| +eat():void |

5、泛化关系

泛化关系是对象之间耦合度最大的两种关系之一。表示一般与特殊，**父类与子类**，是一种继承关系，是is - a

|  |
| --- |
| 人  Person |
| -name: String  -age:int |
| +speak():void |

|  |
| --- |
| 老师  Teacher |
| -teacherNo:long |
| +teaching():void |

|  |
| --- |
| 学生  Student |
| -studentNo:long |
| +study():void |

6、实现关系

实现关系是对象之间耦合度最大的两种关系之一。实现关系是接口与实现类之间的关系。

车辆

Vehicle

+move():void

|  |
| --- |
| 船  Ship |
|  |
| +move() : void |

|  |
| --- |
| 汽车  Car |
|  |
| +move() : void |

**四、软件设计原则**

开闭原则（OCP）：<http://c.biancheng.net/view/1322.html>

里式替换原则（LSP）：<http://c.biancheng.net/view/1324.html>

依赖倒置原则（DIP）：<http://c.biancheng.net/view/1326.html>

单一职责原则（SRP）：<http://c.biancheng.net/view/1327.html>

接口隔离原则（ISP）：<http://c.biancheng.net/view/1330.html>

迪米特法则（LoD/LKP）：<http://c.biancheng.net/view/1331.html>

合成复用原则（CRP/CARP）：<http://c.biancheng.net/view/1333.html>

**五、设计模式详解**

1、**单例（Singleton）模式** <http://c.biancheng.net/view/1338.html>

定义：一个类只有一个实例，且该类能自行创建这个实例的一种模式。

特点：（1）单例类只有一个实例对象

（2）该单例对象必须由单例类自行创建（静态私有实例），即将类的构

造函数设置为私有，外部类无法通过new调用该构造函数生成多

个实例。

（3）单例类对外提供一个访问该单例的全局访问点（静态公有函数，

用于创建或获取该静态私有实例）。

结构：单例类：包含一个实例且能自行创建这个实例的类。

访问类：使用单例的类

实现：（1）懒汉式单例：类加载时没有生成单例，只有当第一次调用getIns

tance()方法时才去创建这个单例。

（2）饿汉式单例：类一旦加载就创建一个单例，保证在调用 getInsta

nce 方法之前单例已经存在了。

2、原型（Prototype）模式 <http://c.biancheng.net/view/1343.html>

3、**工厂方法模式**  <http://c.biancheng.net/view/1348.html>

定义：定义一个创建产品对象的工厂接口，将产品对象的实际创建工作推迟

到具体子工厂类当中。如果要创建的产品不多，只要一个工厂类就可

以完成，则称之为“简单工厂模式”

特点：（1）满足创建型模式“创建与使用相分离”的特点，创建的对象称为

“产品”，创建产品的对象称为“工厂”；

（2）用户只需要知道具体工厂的名称就可得到所要的产品，无须知道

产品的具体创建过程；

（3）在系统增加新的产品时只需要添加具体产品类和对应的具体工厂

类，无须对原工厂进行任何修改，满足开闭原则；

（4）每增加一个产品就要增加一个具体产品类和一个对应的具体工厂

类，这增加了系统的复杂度

结构：（1）抽象工厂（Abstract Factory）：提供了创建产品的接口，调用者通

过它访问具体工厂的工厂方法 newProduct() 来创建产品。

（2）具体工厂（Concrete Factory）：主要是实现抽象工厂中的抽象方

法，完成具体产品的创建。

（3）抽象产品（Product）：定义了产品的规范，描述了产品的主要特

性和功能

（4）具体产品（ConcreteProducet）：实现了抽象产品角色所定义的接

口，由具体工厂来创建，它同具体工厂之间一一对应。

实现：

4、**抽象工厂模式** <http://c.biancheng.net/view/1351.html>

定义：是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访

问类无须指定所要产品的具体类就能得到同族的不同等级的产品的

模式结构。

特点：（1）抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产

一个等级的产品，而抽象工厂模式可生产多个等级的产品。

（2）系统中有多个产品族，每个具体工厂创建同一族但属于不同等级

结构的产品。

（3）系统一次只可能消费其中某一族产品，即同族的产品一起使用。

（4）可以在类的内部对产品族中相关联的多等级产品共同管理，而不

必专门引入多个新的类来进行管理；当增加一个新的产品族时不

需要修改原代码，满足开闭原则。

（5）当产品族中需要增加一个新的产品时，所有的工厂类都需要进行

修改。

结构：（1）抽象工厂（Abstract Factory）：提供了创建产品的接口，它包含

多个创建产品的方法 newProduct()，可以创建多个不同等级的产

品。

（2）具体工厂（Concrete Factory）：主要是实现抽象工厂中的多个抽

象方法，完成具体产品的创建。

（3）抽象产品（Product）：定义了产品的规范，描述了产品的主要特

性和功能，抽象工厂模式有多个抽象产品；

（4）具体产品（ConcreteProduct）：实现了抽象产品角色所定义的接

口，由具体工厂来创建，它同具体工厂之间是多对一的关系

实现：

5、**建造者模式** <http://c.biancheng.net/view/1354.html>

定义：指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创

建不同的表示。它是将一个复杂的对象分解为多个简单的对象，然后

一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变

的，但每一部分是可以灵活选择的

优点：（1）各个具体的建造者相互独立，有利于系统的扩展

（2）客户端不必知道产品内部组成的细节，便于控制细节风险

缺点：（1）产品的组成部分必须相同，这限制了其使用范围

（2）如果产品的内部变化复杂，该模式会增加很多的建造者类

结构：（1）产品角色（Product）：它是包含多个组成部件的复杂对象，由具

体建造者来创建其各个部件；

（2）抽象建造者（Builder）：它是一个包含创建产品各个子部件的抽

象方法的接口，通常还包含一个返回复杂产品的方法 getResult()；

（3）具体建造者(Concrete Builder）：实现 Builder 接口，完成复杂产

品的各个部件的具体创建方法

（4）指挥者（Director）：它调用建造者对象中的部件构造与装配方法

完成复杂对象的创建，在指挥者中不涉及具体产品的信息

实现：

扩展：建造者（Builder）模式在应用过程中可以根据需要改变，如果创建的

产品种类只有一种，只需要一个具体建造者，这时可以省略掉抽象建

造者，甚至可以省略掉指挥者角色

6、**代理模式** <http://c.biancheng.net/view/1359.html>

定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这

时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问

对象和目标对象之间的中介。

优点：（1）代理模式在客户端与目标对象之间起到一个中介作用和保护目标

对象的作用；

（2）代理对象可以扩展目标对象的功能；

（3）代理模式能将客户端与目标对象分离，在一定程度上降低了系统

的耦合度；

缺点：（1）在客户端和目标对象之间增加一个代理对象，会造成请求处理速

度变慢；

（2）增加了系统的复杂度；

结构：（1）抽象主题（Subject）类：通过接口或抽象类声明真实主题和代理

对象实现的业务方法；

（2）真实主题（Real Subject）类：实现了抽象主题中的具体业务，是

代理对象所代表的真实对象，是最终要引用的对象；

（3）代理（Proxy）类：提供了与真实主题相同的接口，其内部含有

对真实主题的引用，它可以访问、控制或扩展真实主题的功能

实现：

扩展：动态代理，如SpringAOP

7、**适配器模式**  <http://c.biancheng.net/view/1361.html>

定义：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口

不兼容而不能一起工作的那些类能一起工作。分为类适配器模式和对

象适配器模式，前者耦合度高于后者，相对应用较少。

优点：（1）客户端通过适配器可以透明地调用目标接口

（2）复用了现存的类，程序员不需要修改原有代码而重用现有的适配

者类

（3）将目标类和适配者类解耦，解决了目标类和适配者类接口不一致

的问题

缺点：（1）对类适配器来说，更换适配器的实现过程比较复杂

结构：（1）目标（Target）接口：当前系统业务所期待的接口，它可以是抽

象类或接口

（2）适配者（Adaptee）类：它是被访问和适配的现存组件库中的组

件接口

（3）适配器（Adapter）类：它是一个转换器，通过继承或引用适配

者的对象，把适配者接口转换成目标接口，让客户按目标接口的

格式访问适配者

实现：

扩展：双向适配器，既可以把适配器接口转换成目标接口，也可以把目标接

口转换成适配者接口

8、桥接模式 <http://c.biancheng.net/view/1364.html>

9、**装饰模式**  <http://c.biancheng.net/view/1366.html>

定义：指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即

增加其额外功能）的模式

优点：（1）采用装饰模式扩展对象的功能比采用继承方式更加灵活

（2）可以设计出多个不同的具体装饰类，创造出多个不同行为的组合；

缺点：（1）装饰模式增加了许多子类，如果过度使用会使程序变得很复杂；

结构：（1）抽象构件（Component）：定义一个抽象接口以规范准备接收附

加责任的对象

（2）具体构件（Concrete    Component）：实现抽象构件，通过

装饰角色为其添加一些职责；

（3）抽象装饰（Decorator）：实现抽象构件，并包含具体构件的实例，

可以通过其子类扩展具体构件的功能；

（4）具体装饰（ConcreteDecorator）：继承抽象装饰的相关方法，并

给具体构件对象添加附加的责任

实现：

10、**外观模式**  <http://c.biancheng.net/view/1369.html>

定义：是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系

统更加容易被访问的模式。该模式对外有一个统一接口，外部应用程

序不用关心内部子系统的具体的细节，这样会大大降低应用程序的复

杂度，提高了程序的可维护性。

优点：（1）降低了子系统与客户端之间的耦合度，使得子系统的变化不会影

响调用它的客户类；

（2）对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得

子系统使用起来更加容易

（3）降低了大型软件系统中的编译依赖性，简化了系统在不同平台之

间的移植过程，因为编译一个子系统不会影响其他的子系统，也

不会影响外观对象

缺点：（1）不能很好地限制客户使用子系统类；

（2）增加新的子系统可能需要修改外观类或客户端的源代码，违背了

“开闭原则”

结构：（1）外观（Facade）角色：为多个子系统对外提供一个共同的接口；

（2）子系统（Sub System）角色：实现系统的部分功能，客户可以通

过外观角色访问它

（3）客户（Client）角色：通过一个外观角色访问各个子系统的功能

实现：

扩展：引入抽象外观类角色，可解决“违背开闭原则”的问题。

11、享元模式 <http://c.biancheng.net/view/1371.html>

12、**组合模式** <http://c.biancheng.net/view/1373.html>

定义：它是一种将对象组合成树状的层次结构的模式，用来表示“部分-整

体”的关系，使用户对单个对象和组合对象具有一致的访问性

优点：（1）组合模式使得客户端代码可以一致地处理单个对象和组合对象，

无须关心自己处理的是单个对象，还是组合对象，这简化了客户

端代码；

（2）更容易在组合体内加入新的对象，客户端不会因为加入了新的对

象而更改源代码，满足“开闭原则”；

缺点：（1）设计较复杂，客户端需要花更多时间理清类之间的层次关系；

（2）不容易限制容器中的构件；

（3）不容易用继承的方法来增加构件的新功能

结构：（1）抽象构件（Component）角色：它的主要作用是为树叶构件和树

枝构件声明公共接口，并实现它们的默认行为。在透明式的组合

模式中抽象构件还声明访问和管理子类的接口；在安全式的组合

模式中不声明访问和管理子类的接口，管理工作由树枝构件完成；

（2）树叶构件（Leaf）角色：是组合中的叶节点对象，它没有子节点，

用于实现抽象构件角色中声明的公共接口；

（3）树枝构件（Branch）角色：是组合中的分支节点对象，它有子节

点。它实现了抽象构件角色中声明的接口，它的主要作用是存储

和管理子部件，通常包含 Add()、Remove()、GetChild()等方法；

实现：（1）透明方式：在该方式中，由于抽象构件声明了所有子类中的全部

方法，所以客户端无须区别树叶对象和树枝对象，对客户端来说

是透明的。但其缺点是：树叶构件本来没有 Add()、Remove() 及

GetChild() 方法，却要实现它们（空实现或抛异常），这样会带来

一些安全性问题；

（2）安全方式：在该方式中，将管理子构件的方法移到树枝构件中，

抽象构件和树叶构件没有对子对象的管理方法，这样就避免了上

一种方式的安全性问题，但由于叶子和分支有不同的接口，客户

端在调用时要知道树叶对象和树枝对象的存在，所以失去了透明

性。

扩展：

13、**模板方法模式** <http://c.biancheng.net/view/1376.html>

14、**策略模式** <http://c.biancheng.net/view/1378.html>

定义：该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互

替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行

为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分

割开来，并委派给不同的对象对这些算法进行管理

优点：（1）多重条件语句不易维护，而使用策略模式可以避免使用多重条件

语句；

（2）策略模式提供了一系列的可供重用的算法族，恰当使用继承可以

把算法族的公共代码转移到父类里面，从而避免重复的代码；

（3）策略模式可以提供相同行为的不同实现，客户可以根据不同时间

或空间要求选择不同的；

（4）策略模式提供了对开闭原则的完美支持，可以在不修改原代码的

情况下，灵活增加新算法

（5）策略模式把算法的使用放到环境类中，而算法的实现移到具体策

略类中，实现了二者的分离

缺点：（1）客户端必须理解所有策略算法的区别，以便适时选择恰当的算法

类；

（2）策略模式造成很多的策略类；

结构：（1）抽象策略（Strategy）类：定义了一个公共接口，各种不同的算

法以不同的方式实现这个接口，环境角色使用这个接口调用不同

的算法，一般使用接口或抽象类实现；

（2）具体策略（Concrete Strategy）类：实现了抽象策略定义的接口，

提供具体的算法实现；

（3）环境（Context）类：持有一个策略类的引用，最终给客户端调

用；

实现：

扩展：当策略很多时，可在环境类中采用策略工厂模式来管理这些策略，减

少客户端的工作复杂度

15、**命令模式**  <http://c.biancheng.net/view/1380.html>

16、**责任链模式** <http://c.biancheng.net/view/1383.html>

定义：为了避免请求发送者与多个请求处理者耦合在一起，将所有请求的处

理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求

发生时，可将请求沿着这条链传递，直到有对象处理它为止。

优点：（1）降低了对象之间的耦合度。该模式使得一个对象无须知道到底是

哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥

有对方的明确信息；

（2）增强了系统的可扩展性。可以根据需要增加新的请求处理类，满

足开闭原则；

（3）增强了给对象指派职责的灵活性。当工作流程发生变化，可以动

态地改变链内的成员或者调动它们的次序，也可动态地新增或者

删除责任；

（4）责任链简化了对象之间的连接。每个对象只需保持一个指向其后

继者的引用，不需保持其他所有处理者的引用，这避免了使用众

多的 if 或者 if···else 语句；

（5）责任分担。每个类只需要处理自己该处理的工作，不该处理的传

递给下一个对象完成，明确各类的责任范围，符合类的单一职责

原则；

缺点：（1）不能保证每个请求一定被处理。由于一个请求没有明确的接收者

，所以不能保证它一定会被处理，该请求可能一直传到链的末端

都得不到处理；

（2）对比较长的职责链，请求的处理可能涉及多个处理对象，系统性

能将受到一定影响；

（3）职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性

，可能会由于职责链的错误设置而导致系统出错，如可能会造成

循环调用；

结构：（1）抽象处理者（Handler）角色：定义一个处理请求的接口，包含

抽象处理方法和一个后继连接

（2）具体处理者（Concrete Handler）角色：实现抽象处理者的处理

方法，判断能否处理本次请求，如果可以处理请求则处理，否则

将该请求转给它的后继者

（3）客户类（Client）角色：创建处理链，并向链头的具体处理者对

象提交请求，它不关心处理细节和请求的传递过程；

实现：

扩展：（1）纯的职责链模式：一个请求必须被某一个处理者对象所接收，且

一个具体处理者对某个请求的处理只能采用以下两种行为之一：

自己处理（承担责任）；把责任推给下家处理；

（2）不纯的职责链模式：允许出现某一个具体处理者对象在承担了请

求的一部分责任后又将剩余的责任传给下家的情况，且一个请求

可以最终不被任何接收端对象所接收

17、状态模式 <http://c.biancheng.net/view/1388.html>

18、**观察者模式** <http://c.biancheng.net/view/1390.html>

定义：指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，

所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作

发布-订阅模式、模型-视图（MVC中的M和V）模式，它是对象行为

型模式

优点：（1）降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系

（2）目标与观察者之间建立了一套触发机制

缺点：（1）目标与观察者之间的依赖关系并没有完全解除，而且有可能出现

循环引用

（2）当观察者对象很多时，通知的发布会花费很多时间，影响程序的

效率

结构：（1）抽象目标类：它提供了一个用于保存观察者对象的聚集类和增加

、删除观察者对象的方法，以及通知所有观察者的抽象方法；

（2）具体目标类：它实现抽象目标中的通知方法，当具体目标的内部

状态发生改变时，通知所有注册过的观察者对象。

（3）抽象观察者：它是一个抽象类或接口，它包含了一个更新自己的

抽象方法，当接到具体目标类的更改通知时被调用。

（4）具体观察者：实现抽象观察者中定义的抽象方法，以便在得到目

标的更改通知时更新自身的状态。

实现：

扩展：在Java中，通过java.util.Observable类和java.util.Observer接口定义

了观察者模式，只要实现它们的子类就可以编写观察者模式实例。

19、中介者模式 <http://c.biancheng.net/view/1393.html>

20、**迭代器模式** <http://c.biancheng.net/view/1395.html>

定义：提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对

象的内部表示。迭代器模式是一种对象行为型模式。

优点：（1）访问一个聚合对象的内容而无须暴露它的内部表示；

（2）遍历任务交由迭代器完成，这简化了聚合类；

（3）它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类

以支持新的遍历；

（4）增加新的聚合类和迭代器类都很方便，无须修改原有代码；

（5）封装性良好，为遍历不同的聚合结构提供一个统一的接口

缺点：（1）增加了类的个数，这在一定程度上增加了系统的复杂性；

结构：（1）抽象聚合（Aggregate）角色：定义存储、添加、删除聚合对象

以及创建迭代器对象的接口；

（2）具体聚合（ConcreteAggregate）角色：实现抽象聚合类，返回一

个具体迭代器的实例；

（3）抽象迭代器（Iterator）角色：定义访问和遍历聚合元素的接口，

通常包含 hasNext()、first()、next() 等方法；

（4）具体迭代器（Concretelterator）角色：实现抽象迭代器接口中所

定义的方法，完成对聚合对象的遍历，记录遍历的当前位置

实现：

扩展：迭代器模式常常与组合模式结合起来使用，在对组合模式中的容器构

件进行访问时，经常将迭代器潜藏在组合模式的容器构成类中。当然，

也可以构造一个外部迭代器来对容器构件进行访问。

21、访问者模式 <http://c.biancheng.net/view/1397.html>

22、备忘录模式 <http://c.biancheng.net/view/1400.html>

23、解释器模式 <http://c.biancheng.net/view/1402.html>