# 设计模式

1、创建型模式

软件设计的过程是循序渐进的，一步一步来的。在软件设计中对象的创建和对象的使用是分开的，因为对象的创建会消耗掉系统的很多资源，所以单独对对象的创建进行研究，从而能够高效地创建对象就是创建型模式要探讨的问题。这里就提供了多种创建型模式进行选择使用。

2、结构型模式

在解决了对象的创建问题之后，对象的组成以及对象之间的依赖关系就成了开发人员关注的焦点，因为如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性、代码的健壮性、耦合性等。所以也有多种结构型模式可供开发人员选择使用。

3、行为型模式

在对象的结构和对象的创建问题都解决了之后，就剩下对象的行为问题了，如果对象的行为设计的好，那么对象的行为就会更清晰，它们之间的协作效率就会提高。

# 面向对象设计六大原则

## 单一职责原则

### 核心思想

[单一职责原则（SRP, Single Responsibility Principle）](https://www.cnblogs.com/libingql/p/3804327.html)规定一个类应该有且只有一个变化的原因。

### 引入单一职责原则的原因

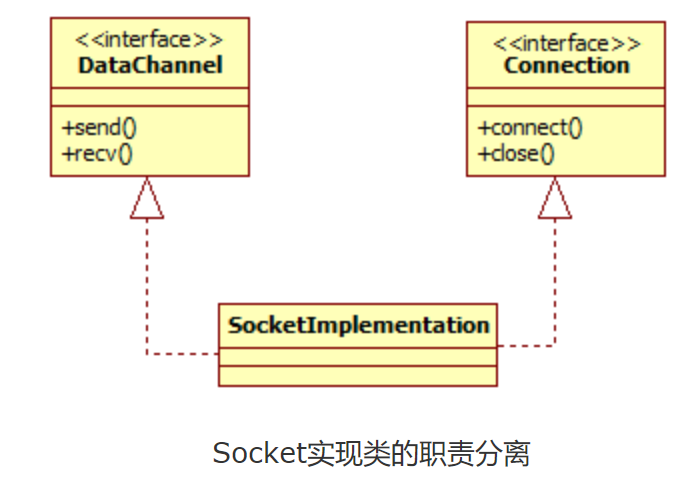
当需求变化时，将通过更改职责相关的类来体现。如果一个类拥有多于一个的职责，则多个职责耦合在一起，会有多于一个原因来导致这个类发生变化。一个职责的变化可能会影响到其他的职责，另外，把多个职责耦合在一起，影响复用性。

### 单一职责原则的优点

1. 降低类的复杂度；  
    2. 提高类的可读性，提高系统的可维护性；  
    3. 降低变更引起的风险（降低对其他功能的影响）。

### 单一职责原则实现

单一职责原则关键点：要求接口的职责单一，从而实现该接口的类的职责单一。



　　IDataChannel职责：数据通信

　　IConnection职责：连接管理

　　SocketImplementation：两个职责耦合，这不是所希望的，但或许是必要的。

### 单一职责原则的重构

业务规则和持久化两个职责应该分开：业务规则往往会频繁变化，而持久化的方式却不会如此频繁的变化，并且变化的原因完全不同。

违反SRP原则的重构可采取设计模式：[外观模式（Facade）](http://www.cnblogs.com/libingql/p/3636939.html" \t "https://www.cnblogs.com/libingql/p/_blank)、[代理模式（Proxy）](http://www.cnblogs.com/libingql/p/3637660.html" \t "https://www.cnblogs.com/libingql/p/_blank)或数据访问对象（DAO）。

### 使用单一职责原则的注意点

1. 单一职责最难划分的是职责。  
    2. 单一职责原则提出标准：用职责和变化原因来衡量接口或类设计的是否优良，但是职责和变化原因都是不可度量的，因项目、环境而异。  
    3. 接口一定要做到单一职责，类的设计尽量做到只有一个原因引起变化。

## 开闭原则

### 开闭原则概述

开闭原则（OCP，The Open-Closed Principle）：一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。

开闭原则是面向对象设计中可复用设计的基石。

### 开闭原则的实现

开闭原则实现关键：抽象。

抽象基类：把系统的所有可能的行为抽象成一个抽象底层，这个抽象底层规定出所有的具体实现必须提供的方法的特征。作为系统设计的抽象层，要预见所有可能的扩展，从而使得在任何扩展情况下，系统的抽象底层不需修改。

派生类：从抽象基类派生一个或多个新的具体实现，可以扩展基类的行为，系统设计对扩展开放。

### 如何使用开闭原则

抽象约束：

1. 通过接口或者抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法；
2. 参数类型、引用对象尽量使用接口或者抽象类，而不是实现类；
3. 抽象层尽量保持稳定，一旦确定即不允许修改。

让设计对于最有可能发生的变化遵循OCP原则。遵循OCP原则的代价是很昂贵的，创建适当的对象需要开发时间和精力，抽象增加软件复杂度。把OCP应用限定在最有可能发生的变化上。

### 开闭原则的优点

1. 可复用性
2. 可维护性

### 开闭原则重构

违反开闭原则（OCP）原则的重构可采取设计模式：[策略模式（Strategy）](http://www.cnblogs.com/libingql/p/3498451.html" \t "https://www.cnblogs.com/libingql/p/_blank)、[模板方法模式（Template Method）](http://www.cnblogs.com/libingql/p/3638002.html" \t "https://www.cnblogs.com/libingql/p/_blank)。

## 里氏代换原则

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

例如：void func(ParentClass param);可以传父类型的参数，那么就一定也可以传子类型的参数。

## 依赖倒置原则

依赖倒转原则(Dependency Inversion  Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对抽象（接口）编程，而不是针对实现细节编程。

开闭原则（OCP）是面向对象设计原则的基础也是整个设计的一个终极目标，而依赖倒置原则（DIP )则是实现OCP原则的一个基础。

### 使用依赖倒置原则注意点

在使用DIP是需要注意一下几点

1. 继承自高层接口的类要实现所有接口中的方法。

2.子类中除了接口的方法，在用接口声明的对象调用的地方是无法被调用到的。除非直接调用子类，但是直接调用子类是违背DIP的。

3. DIP是实现OCP的重要原则保障，一般违背了DIP很难不违背OCP

4.LSP 是实现DIP的基础，多态给实现DIP提供了可能。

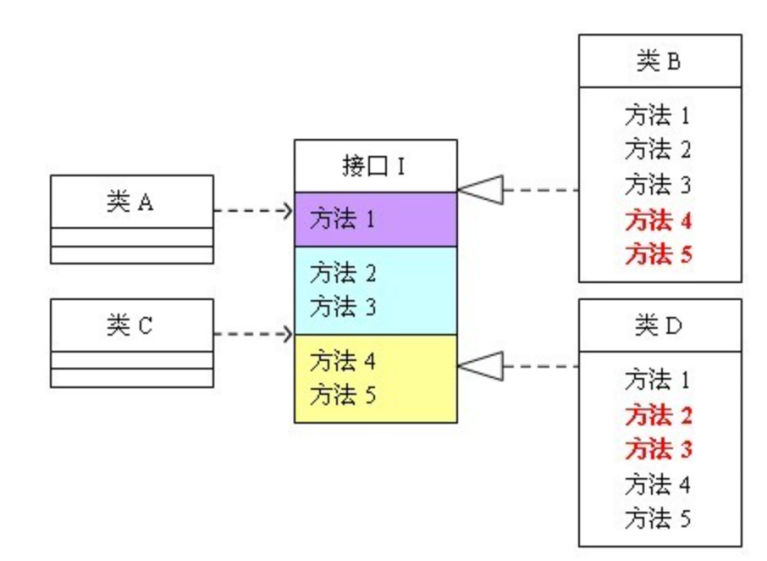
## 接口隔离原则

第一种定义: Clients should not be forced to depend upon interfaces that they don't use. 客户端不应该依赖它不需用的接口。

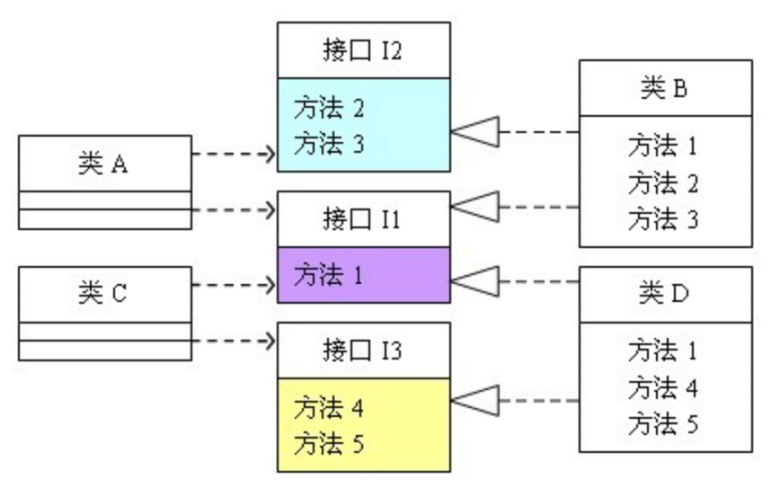
第二种定义：The dependency of one class to another one should depend on the smallest possible interface。类间的依赖关系应该建立在最小的接口上。

理解：用不到或用不上的方法就不要在接口上定义，如果一个接口有5个方法，一个实现类只用到3个方法，另外一个实现类用到另外3种方法，那么可以考虑把这个接口划分成2个或多个接口，让这2个实现类去实现，，保证每个实现的方法都有用。

未遵循接口隔离原则的设计:



遵循接口隔离原则的设计:



### 接口隔离原则与单一职责原则的区别

单一职责要求的是类和接口职责单一，注重的是职责，没有要求接口的方法减少，例如一个职责可能包含 10个方法，这 10个方法都放在一个接口中，并且提供给多个模块访问，各个模块按照规定的权限来访问，在系统外通过文档约束不使用的方法不要访问，按照单一职责原则是允许的，按照接口隔离原则是不允许的，因为它要求“尽量使用多个专门的接口”,专门的接口指什么？就是指提供给多个模块的接口，提供给几个模块就应该有几个接口，而不是建立一个庞大的臃肿的接口，所有的模块可以来访问。

## 迪米特法则

迪米特法则（Law of Demeter）又叫作最少知识原则（Least Knowledge Principle 简写LKP）：一个软件实体应当尽可能少的与其他实体发生相互作用。每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

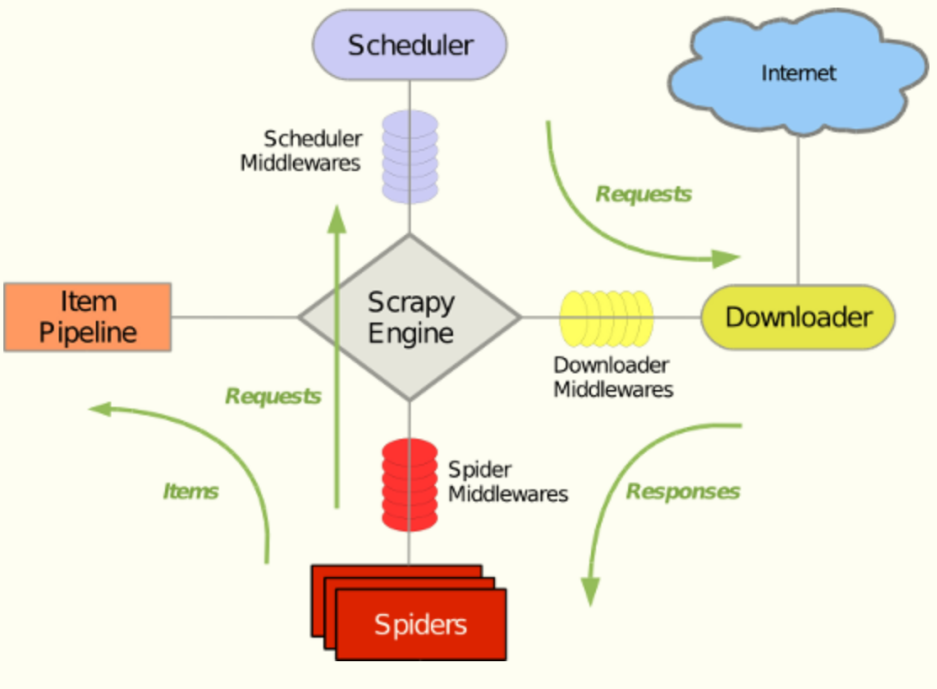
迪米特法则不希望类之间建立直接的联系。如果真的有需要建立联系，也希望能通过它的[友元类](https://baike.baidu.com/item/%E5%8F%8B%E5%85%83%E7%B1%BB/518734" \t "https://my.oschina.net/u/3568600/blog/_blank)来转达。因此，应用迪米特法则有可能造成的一个后果就是：系统中存在大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系——这在一定程度上增加了系统的复杂度。

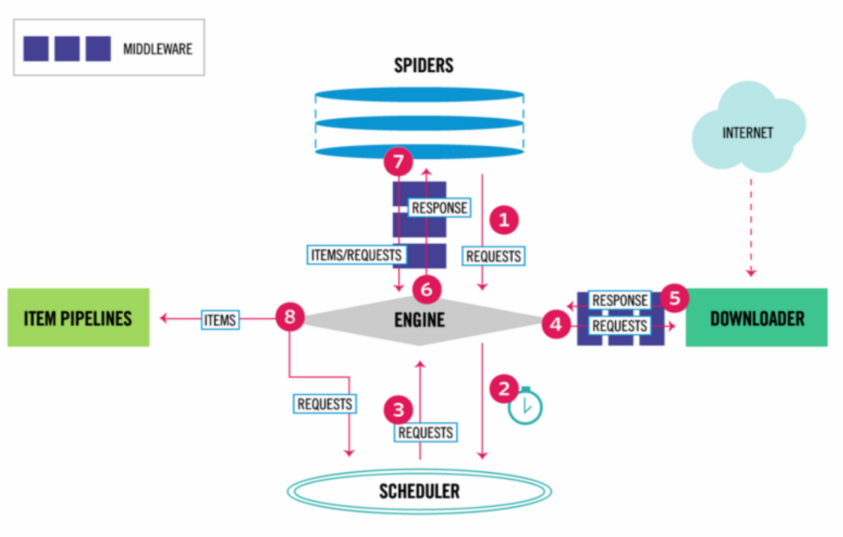
迪米特法则，如果两个类不必彼此直接通向，那么这两个类就不应当发生直接的相互作用。如果其中一个类需要调用另一个类的某一个方法的话，可以通过第三者转发这个调用。

迪米特法则首先强调的前提是在类的结构设计上，每一个类都应当降低成员的访问权限，也就是说，一个类包装好自己的private状态，不需要让别的类知道的字段或行为就不要公开。

应用例子：[门面模式](https://baike.baidu.com/item/%E9%97%A8%E9%9D%A2%E6%A8%A1%E5%BC%8F/764642" \t "https://my.oschina.net/u/3568600/blog/_blank)（[Facade](https://baike.baidu.com/item/Facade/2954918" \t "https://my.oschina.net/u/3568600/blog/_blank)）和中介模式（Mediator）。

Python的scrapy架构图：





Scrapy通过中间的引擎把其他模块都分开，所有模块都和引擎直接关联，再又引擎调用其他模块。这样就避免了多个模块的直接交流。例如爬虫模块需要和调度器模块，管道模块直接关联；下载器需要和爬虫关联，管道需要和爬虫关联... 现在这些模块都只和引擎关联。

# 创建型

## 单例模式（Singleton Pattern）

**定义：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。**

**目的：一个程序只有一个实例，防止有多个实例共同调用方法，导致错乱。**

单例类特点：

1. 单例类只能有一个实例。
2. 单例类必须自己创建自己的唯一实例。
3. 单例类必须给所有其他对象提供这一实例。

使用场景：在Spring中创建的Bean实例默认都是单例模式存在的。

### 饿汉式

代码：designpattrn.creation.singleton.EHSSingleton

饿汉式单例类在类被加载的时候，静态变量instance就会初始化，调用私有构造函数，创建实例。

饿汉式是典型的空间换时间，当类装载的时候就会创建类的实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断，节省了运行时间。

### 懒汉式

代码：designpattrn.creation.singleton.LHSSingleton

懒汉式在被调用创建方法时才会创建实例。

懒汉式是典型的时间换空间,就是每次获取实例都会进行判断，看是否需要创建实例，浪费判断的时间。当然，如果一直没有人使用的话，那就不会创建实例，则节约内存空间。由于懒汉式的实现是线程安全的，这样会降低整个访问的速度，而且每次都要判断。

### 双重锁形式

代码：designpattrn.creation.singleton.DoubleLockSingleton

所谓“双重检查加锁”机制，指的是：并不是每次进入getInstance方法都需要同步，而是先不同步，进入方法后，先检查实例是否存在，如果不存在才进行下面的同步块，这是第一重检查，进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

“双重检查加锁”机制的实现会使用关键字volatile，它的意思是：被volatile修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。

注意：在java1.4及以前版本中，很多JVM对于volatile关键字的实现的问题，会导致“双重检查加锁”的失败，因此“双重检查加锁”机制只只能用在java5及以上的版本。

### 嵌套类形式

代码：designpattrn.creation.singleton.NestedClassSingleton

当getInstance方法第一次被调用的时候，它第一次读取NestedClassSingletonHolder.instance，导致NestedClassSingletonHolder类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建Singleton的实例，由于是静态的域，因此只会在虚拟机装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

1.相应的基础知识

1.1 什么是类级内部类？

简单点说，类级内部类指的是，有static修饰的成员式内部类。如果没有static修饰的成员式内部类被称为对象级内部类。

类级内部类相当于其外部类的static成分，它的对象与外部类对象间不存在依赖关系，因此可直接创建。而对象级内部类的实例，是绑定在外部对象实例中的。

类级内部类中，可以定义静态的方法。在静态方法中只能够引用外部类中的静态成员方法或者成员变量。

类级内部类相当于其外部类的成员，只有在第一次被使用的时候才被会装载。

1.2 多线程缺省同步锁的知识

大家都知道，在多线程开发中，为了解决并发问题，主要是通过使用synchronized来加互斥锁进行同步控制。但是在某些情况中，JVM已经隐含地为您执行了同步，这些情况下就不用自己再来进行同步控制了。这些情况包括：

1.由静态初始化器（在静态字段上或static{}块中的初始化器）初始化数据时

2.访问final字段时

3.在创建线程之前创建对象时

4.线程可以看见它将要处理的对象时

2.解决方案的思路

要想很简单地实现线程安全，可以采用静态初始化器的方式，它可以由JVM来保证线程的安全性。比如前面的饿汉式实现方式。但是这样一来，不是会浪费一定的空间吗？因为这种实现方式，会在类装载的时候就初始化对象，不管你需不需要。

如果现在有一种方法能够让类装载的时候不去初始化对象，那不就解决问题了？一种可行的方式就是采用类级内部类，在这个类级内部类里面去创建对象实例。这样一来，只要不使用到这个类级内部类，那就不会创建对象实例，从而同时实现延迟加载和线程安全。

### 枚举

代码：designpattrn.creation.singleton.EnumSingleton

使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

## 原型模式（Prototype Pattern）

**定义：通过给出一个原型对象来指明所有创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象**。简单地说，就是创建一个新的对象，这个对象和原型对象的属性有相同的值。

**目的：复制一个有当前状态的新对象。不用重新赋值多写代码，不用new节省资源。**

特点：

1. 克隆的对象与原对象并不是同一个对象，分别占用不同的堆空间 x.clone()!=x。
2. 克隆的对象与原对象的类型一样 x.clone().getClass()==x.clone().getClass()。
3. 如果对象x的equals()方法定义其恰当的话，那么x.clone().equals(x)应当成立的。

优点：

1. 使用原型模式创建对象比直接new一个对象在性能上要好的多，因为Object类的clone方法是一个本地方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。
2. 可以使用深克隆方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起来，以便在需要的时候使用(例如恢复到历史某一状态)，可辅助实现撤销操作。

缺点：

1. 需要为每一个类配置一个克隆方法，而且该克隆方法位于类的内部，当对已有类进行改造的时候，需要修改代码，违反了开闭原则。
2. 在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重签到引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来会比较麻烦。

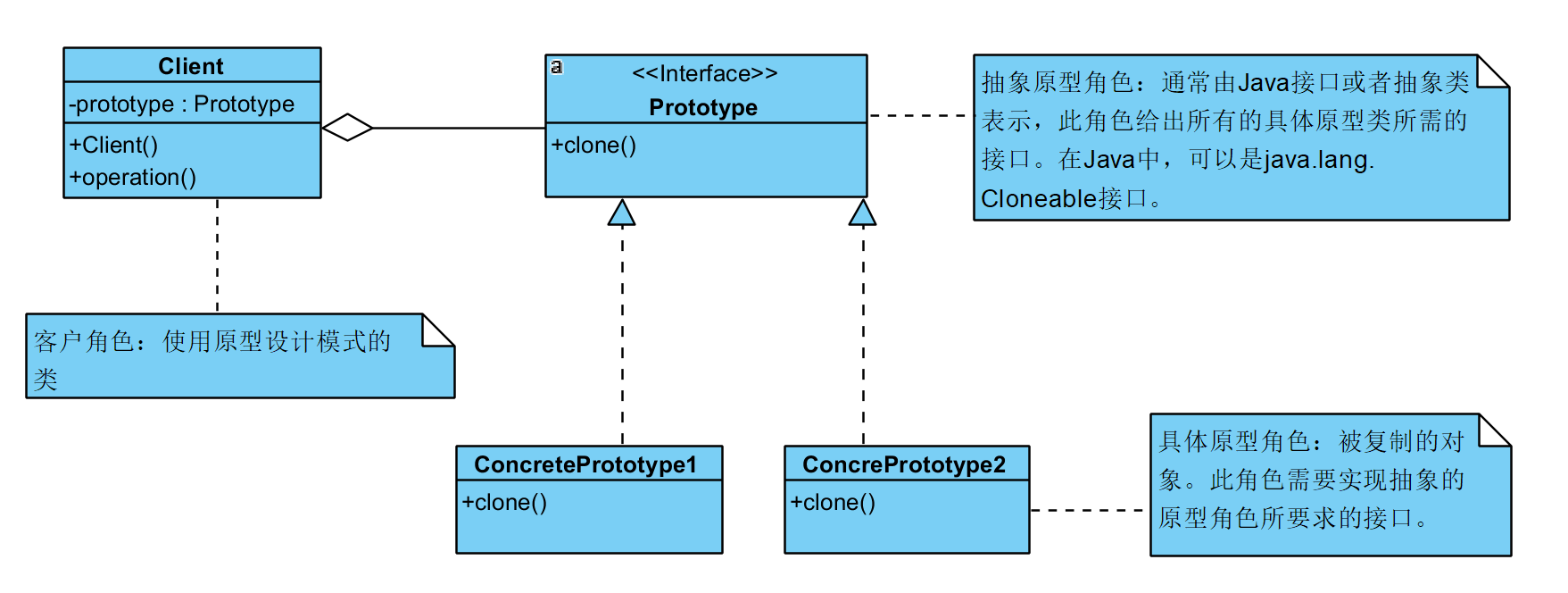
使用场景：

1. Android的Intent：不用在每个Activity中使用intent携带参数，而跳转到下一个Activity，直接调用clone()方法返回intent对象；
2. ArrayList：把后台返回的ArrayList集合数据，调用clone方法直接转为copyList，不用重新定义新的集合来添加后台返回的list集合；注意这里是ArrayList，而不是List；
3. OkHttp；

### 浅拷贝

案例：每周填的周报内容的差不多，用new创建对象消耗资源，不如用Object的clone，通过本地方法拷贝对象。

结构：



代码：

designpattrn.creation.prototype.Main

designpattrn.creation.prototype.WeeklyLog

其他浅拷贝方式：

1. 循环复制
2. 使用List的构造函数
3. List.addAll()
4. System.arraycopy()

### 深拷贝

案例：要求周报添加多附件

分析：因为组存储的是地址，而直接用clone复制，他们会指向同一个地址，当A通过拷贝生成B，A的数组的值改变，B的数组的值也会同时改变。这时可以实现序列化接口，通过，通过ObjectInputStream，ObjectOutputStream流来实现深拷贝。注意transient变量的值将无法序列化。

代码：

designpattrn.creation.prototype.Main

designpattrn.creation.prototype.DeepClone<T>

designpattrn.creation.prototype.AttachmentWeeklyLog

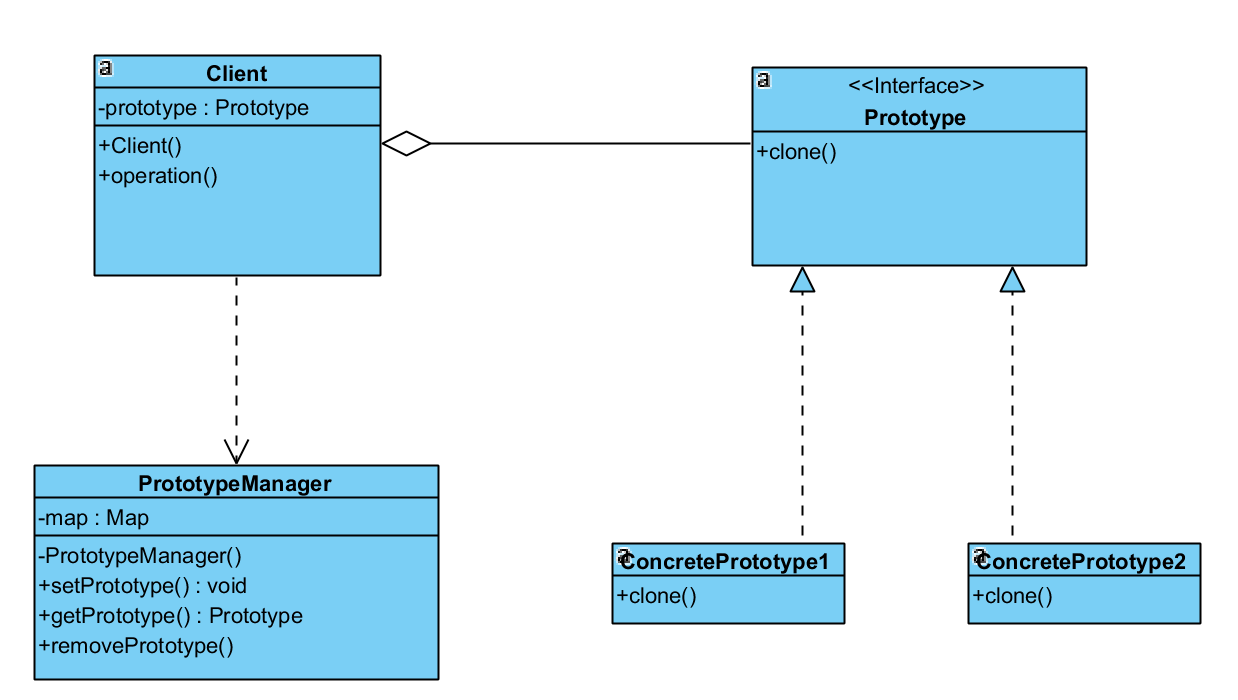
designpattrn.creation.prototype.Attachment

designpattrn.creation.prototype.WeeklyLog

### 登记形式的原型模式

登记形式的原型模式是通过一个管理器Map，将多个原型管理起来。例如Map存了所有员工的周报，获取的时候就可以通过员工id获取对应员工的周报。管理器如果获取不到原型，就需要客户端自己创建原型，然后再放到管理器，交由管理器管理。

结构：



代码：

designpattrn.creation.prototype.Main

designpattrn.creation.prototype.WeeklyLog

designpattrn.creation.prototype.PrototypeManager

如果需要创建的原型对象数目较少而且比较固定的话，可以采取第一种形式。在这种情况下，原型对象的引用可以由客户端自己保存。

如果要创建的原型对象数目不固定的话，可以采取第二种形式。在这种情况下，客户端不保存对原型对象的引用，这个任务被交给管理员对象。在复制一个原型对象之前，客户端可以查看管理员对象是否已经有一个满足要求的原型对象。如果有，可以直接从管理员类取得这个对象引用；如果没有，客户端就需要自行复制此原型对象。

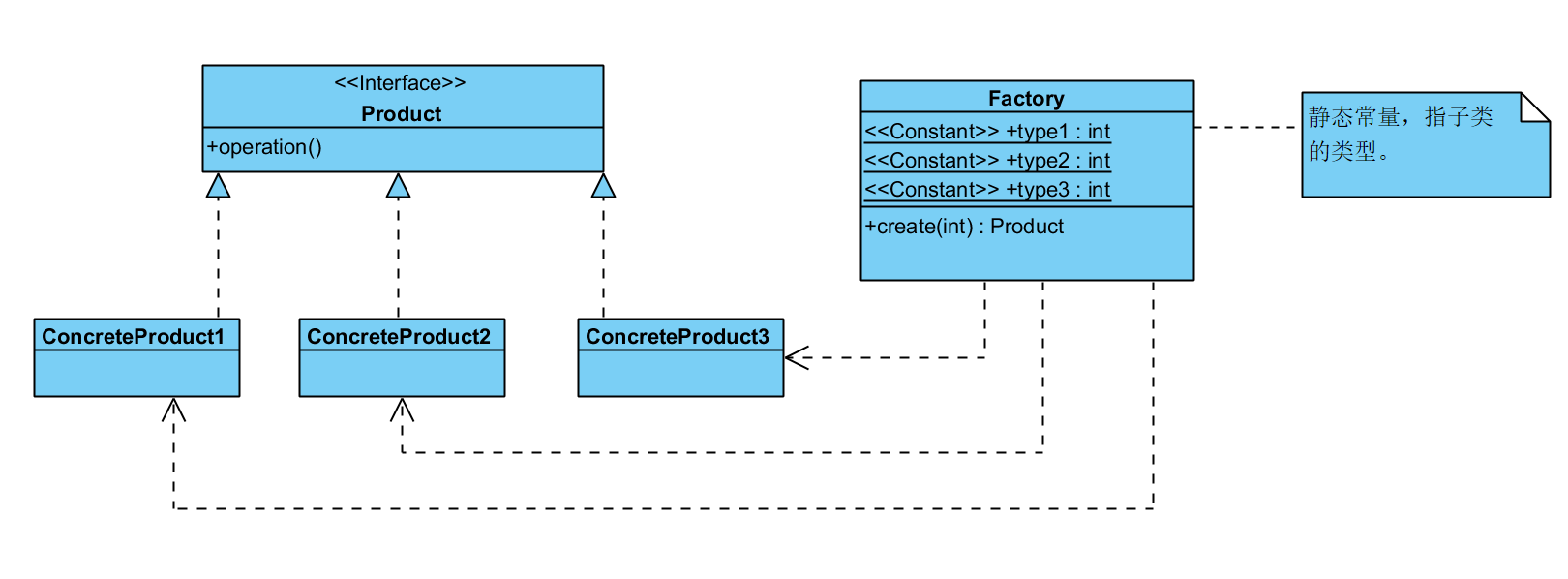
## 工厂方法模式（Factory Method Pattern）

### 简单工厂模式

简单工厂模式是属于创建型模式，又叫做静态工厂方法（Static Factory Method）模式，但不属于23种GOF设计模式之一。简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式，可以理解为是不同工厂模式的一个特殊实现。

**目的：把创建对象的功能抽取出来，统一创建。**

结构：



代码：

designpattrn.creation.factorymethod.Main的simpleFactory();

缺点：

每新增一个子类，都需要修改Factory的create方法。方法一复杂，改代码就容易出错，现在希望只是新增代码，而不修改代码。如果工厂的create方法有2个参数，分别有m，和n中情况。就会有m \* n个if，如果新增一种类型，就会有m \* (n+1)个if，只是加了一种类型，就多了m个if。

简单工厂模式适用于create的方法只有1个参数。

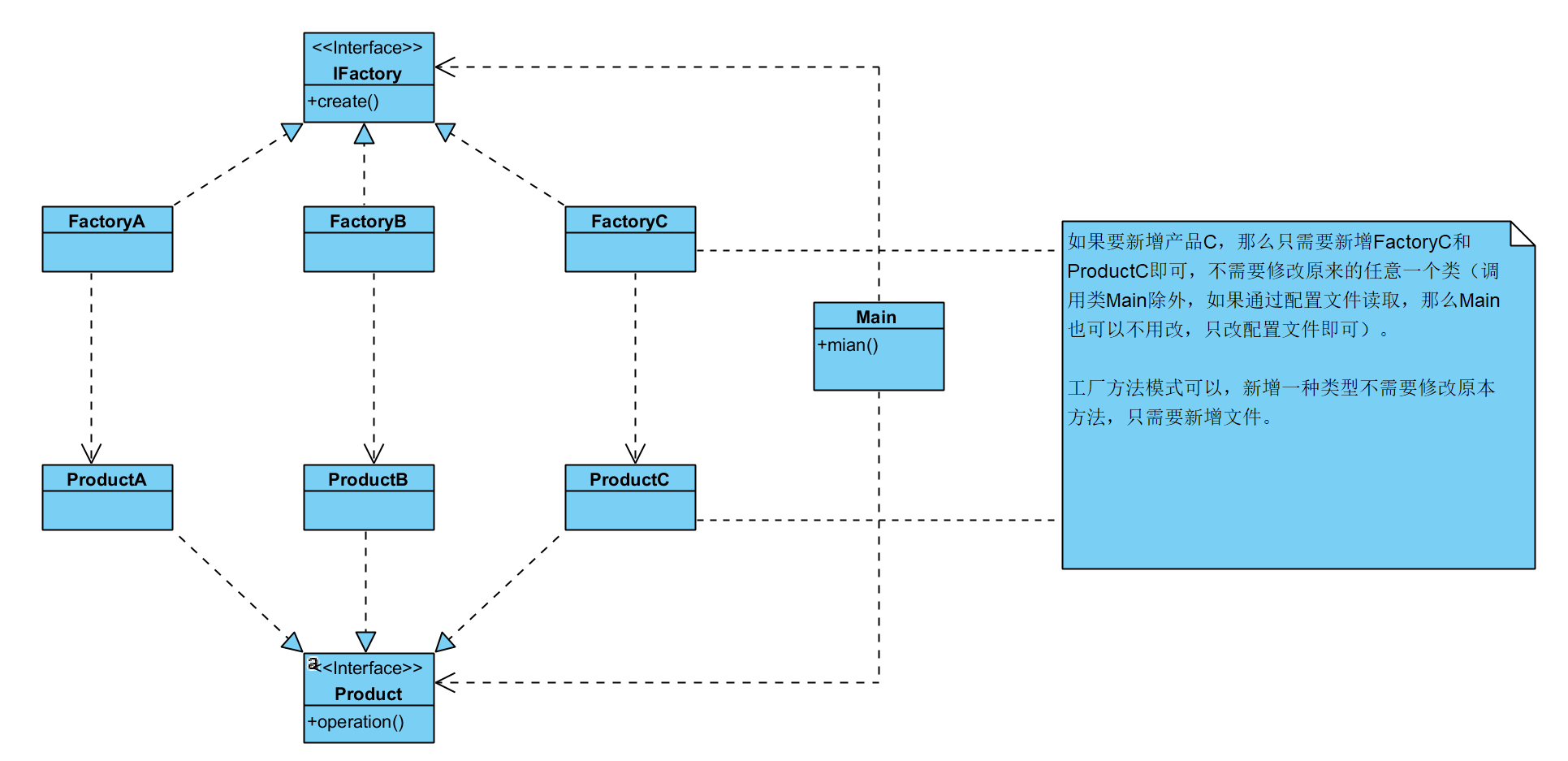
### 工厂方法模式

工厂方法模式是类的创建模式，又叫做虚拟构造子(Virtual Constructor)模式或者多态性工厂（Polymorphic Factory）模式。工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

如果系统需要加入一个新的类型，那么所需要的就是向系统中加入一个这个类以及所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的类型而言，这个系统完全支持“开-闭原则”。

**目的：新增一种类型，不需要修改原来的类和方法，只需要新增类和方法。**

结构：



角色：

抽象工厂角色：负责定义创建对象的方法，这个角色也常常使用抽象类实现。

具体工厂角色：与业务逻辑相关，可以创建出具体产品

抽象产品角色：具体产品的抽象，实际开发通常使用抽象类实现

具体产品角色：具体产品

代码：

designpattrn.creation.factorymethod.Main的factoryMethod();

使用场景：hibernate里通过sessionFactory创建session

问题：

如果生成一个对象不只是又一个参数决定，是由多个参数决定，而且这些参数分成两批，是不能共用的，那怎么办？

## 抽象工厂模式（Abstract Factory Pattern）

**定义：为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。**

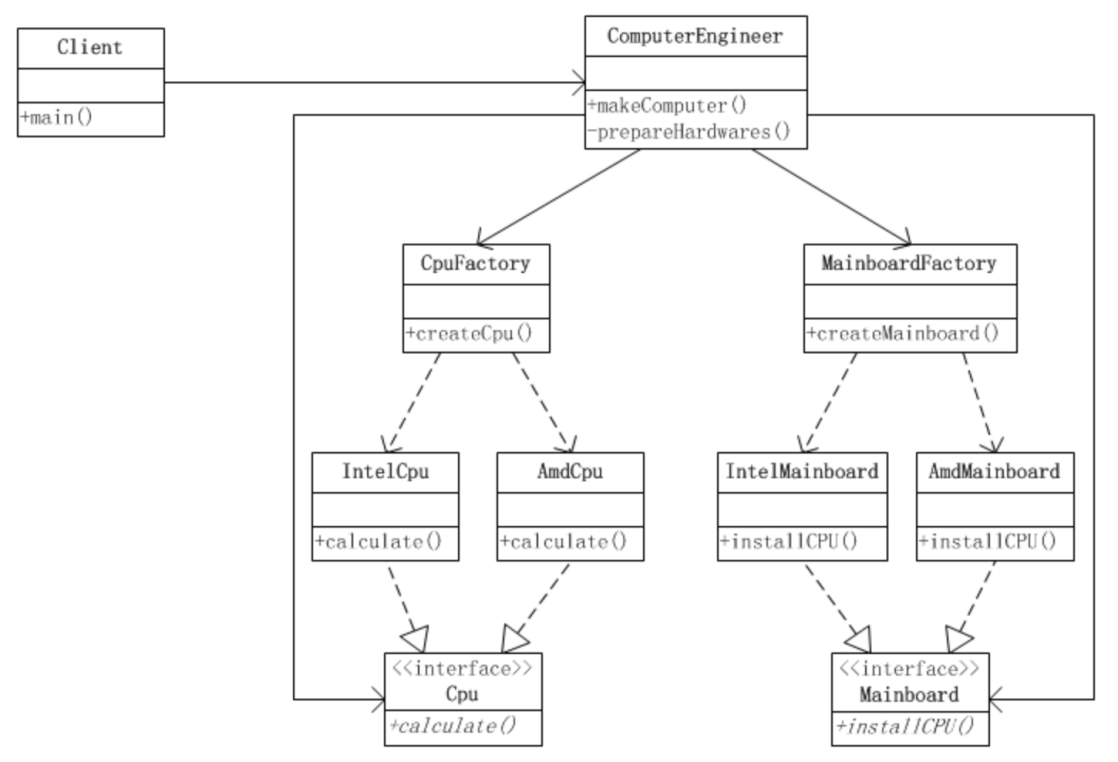
**目的：创建一个对象，这个对象需要多个参数，而这些参数又是一系列的，相关联的。抽象工厂模式的目的是为了让这一系列参数相互约束，是在一起使用的。**

场景问题：

举个生活中常见的例子——组装电脑，我们在组装电脑的时候，通常需要选择一系列的配件，比如CPU、硬盘、内存、主板、电源、机箱等。为讨论使用简单点，只考虑选择CPU和主板的问题。

在最终确定这个装机方案之前，还需要整体考虑各个配件之间的兼容性。比如：CPU和主板，如果使用Intel的CPU和AMD的主板是根本无法组装的。因为Intel的CPU针脚数与AMD主板提供的CPU插口不兼容，就是说如果使用Intel的CPU根本就插不到AMD的主板中，所以装机方案是整体性的，里面选择的各个配件之间是有关联的。

使用简单工厂模式：

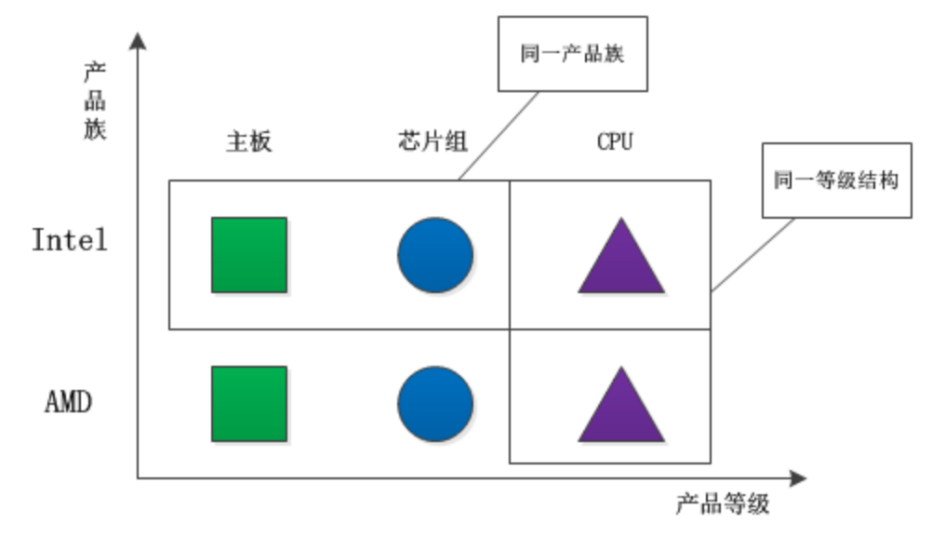


但是上面的类图，有一个问题没解决：那就是这些CPU对象和主板对象其实是有关系的，需要相互匹配的。而上面的实现中，并没有维护这种关联关系。

### 产品族和产品等级

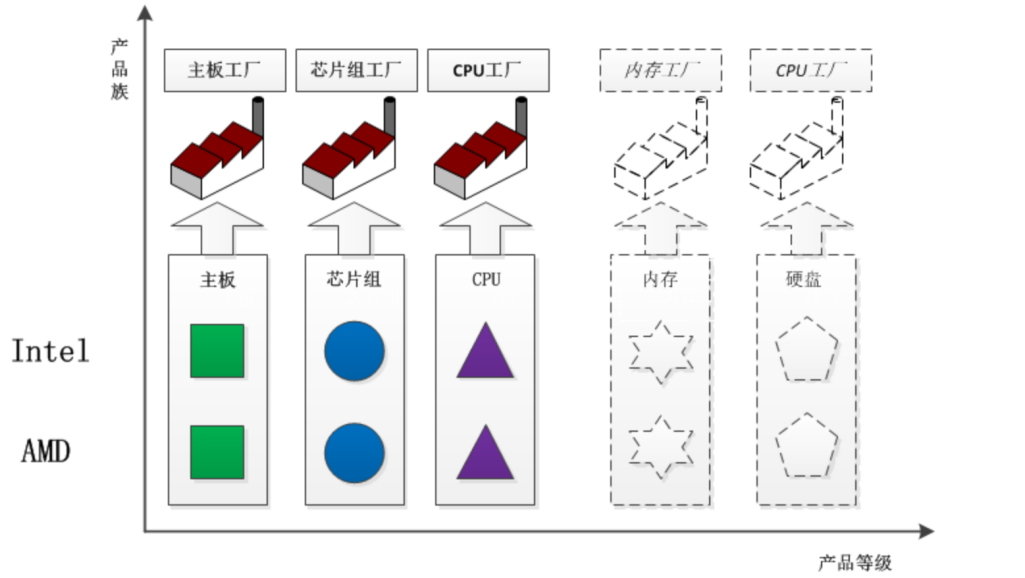
抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构。

所谓产品族，是指位于不同产品等级结构中，功能相关联的产品组成的家族。比如AMD的主板、芯片组、CPU组成一个家族，Intel的主板、芯片组、CPU组成一个家族。而这两个家族都来自于三个产品等级：主板、芯片组、CPU。一个等级结构是由相同的结构的产品组成。如下图：

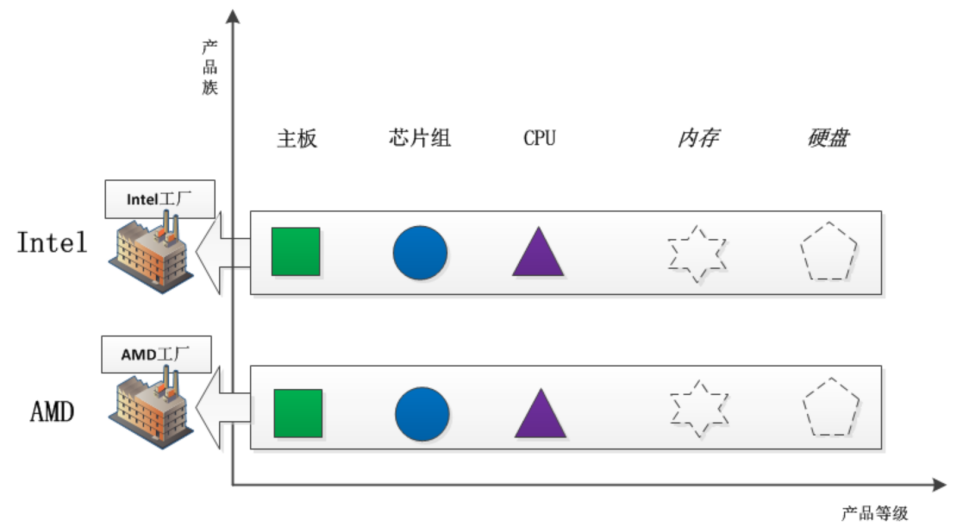


显然，每一个产品族中含有产品的数目，与产品等级结构的数目是相等的。产品的等级结构与产品族将产品按照不同方向划分，形成一个二维的坐标系。横轴表示产品的等级结构，纵轴表示产品族，上图共有两个产品族，分布于三个不同的产品等级结构中。只要指明一个产品所处的产品族以及它所属的等级结构，就可以唯一的确定这个产品。

上面所给出的三个不同的等级结构具有平行的结构。因此，如果采用工厂方法模式，就势必要使用三个独立的工厂等级结构来对付这三个产品等级结构。由于这三个产品等级结构的相似性，会导致三个平行的工厂等级结构。随着产品等级结构的数目的增加，工厂方法模式所给出的工厂等级结构的数目也会随之增加。如下图：

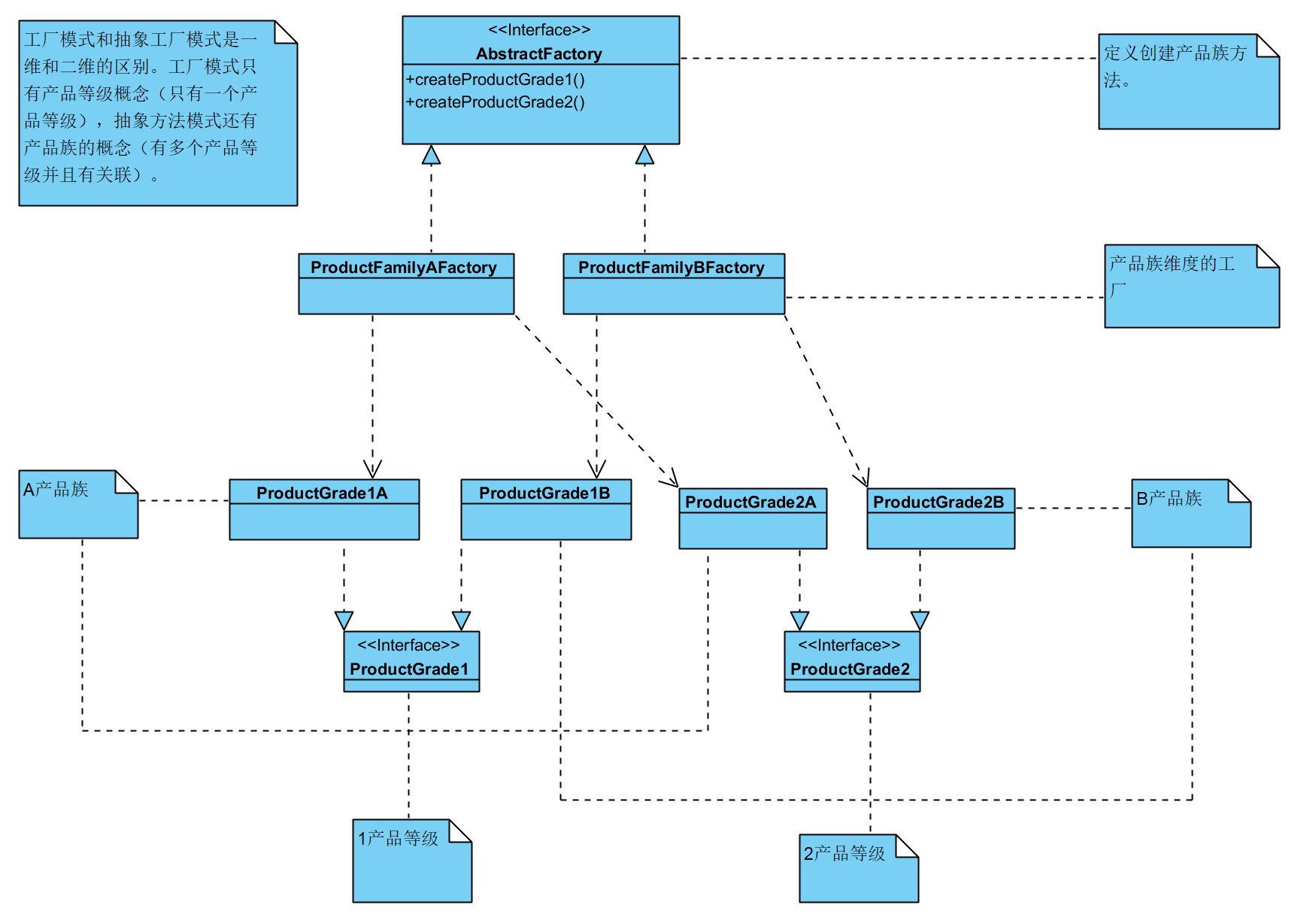


那么，是否可以使用同一个工厂等级结构来对付这些相同或者极为相似的产品等级结构呢？当然可以的，而且这就是抽象工厂模式的好处。同一个工厂等级结构负责三个不同产品等级结构中的产品对象的创建。



可以看出，一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象。显然，这时候抽象工厂模式比简单工厂模式、工厂方法模式更有效率。对应于每一个产品族都有一个具体工厂。而每一个具体工厂负责创建属于同一个产品族，但是分属于不同等级结构的产品。

结构：



代码：

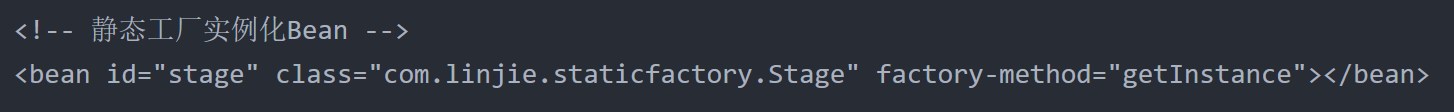
designpattrn.creation.abstractfactory.Main

### 抽象工厂模式与工厂方法模式的不同

抽象工厂模式创建的对象是通过一系列相互制约的参数，生成多个对应产品等级的对象，再组装成一个产品；而工厂方法模式只有产品等级没有产品族。代码的区别是抽象工厂模式的抽象工厂有多个创建产品等级的方法：creatCPU(), creteMainboard(),createXXX()，而工厂方法模式只有一个create()方法。

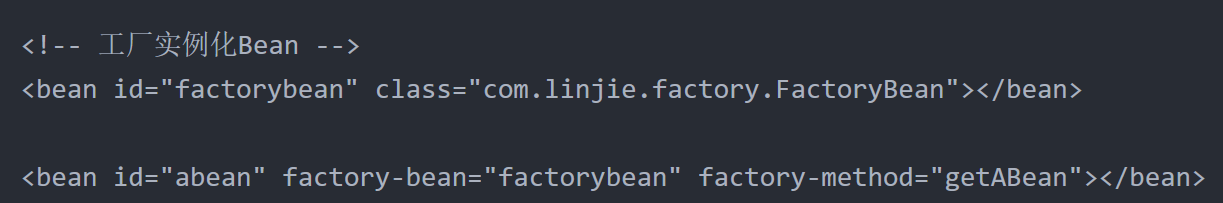
### Spring集成工厂模式

1. 静态工厂实例化Bean



Class写是工厂类，factory-method指的是返回实例的方法。Spring将调用Stage.getInstace();方法返回新实例。

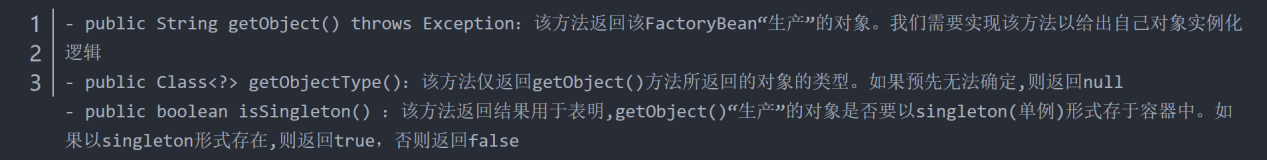
1. 工厂方法实例化Bean

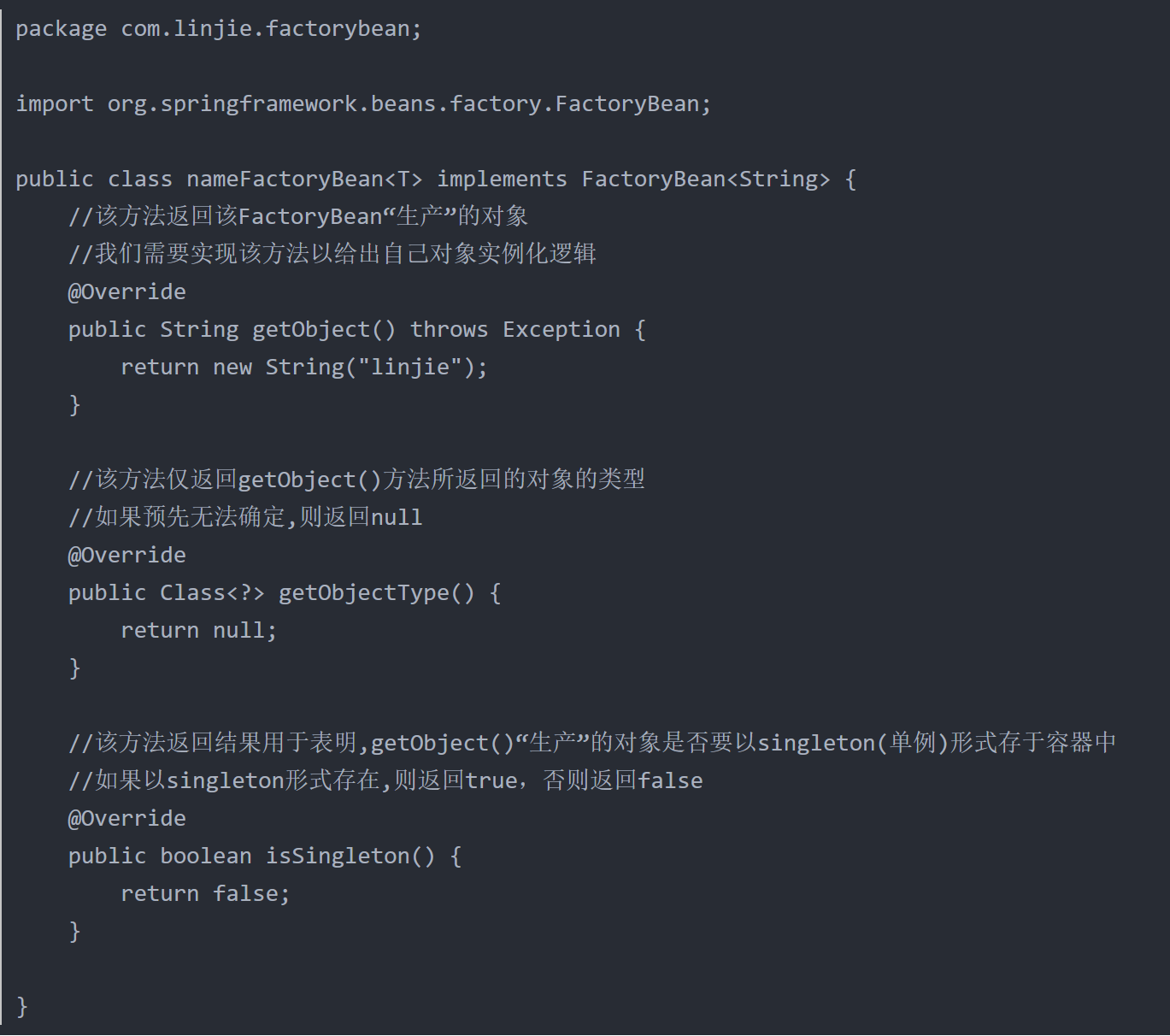


Factorybean是实例化工厂，abean是工厂返回的实例，factory-bean指的是工厂类的id（类似于ref），factory-method指的是工厂类创建Bean的方法。

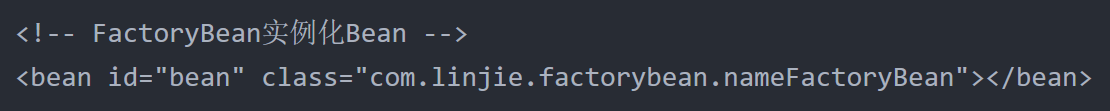
1. 实现Spring的FactoryBean接口

3.1 需要返回的实例实现org.springframework.beans.factory.FactoryBean接口实现3个方法：

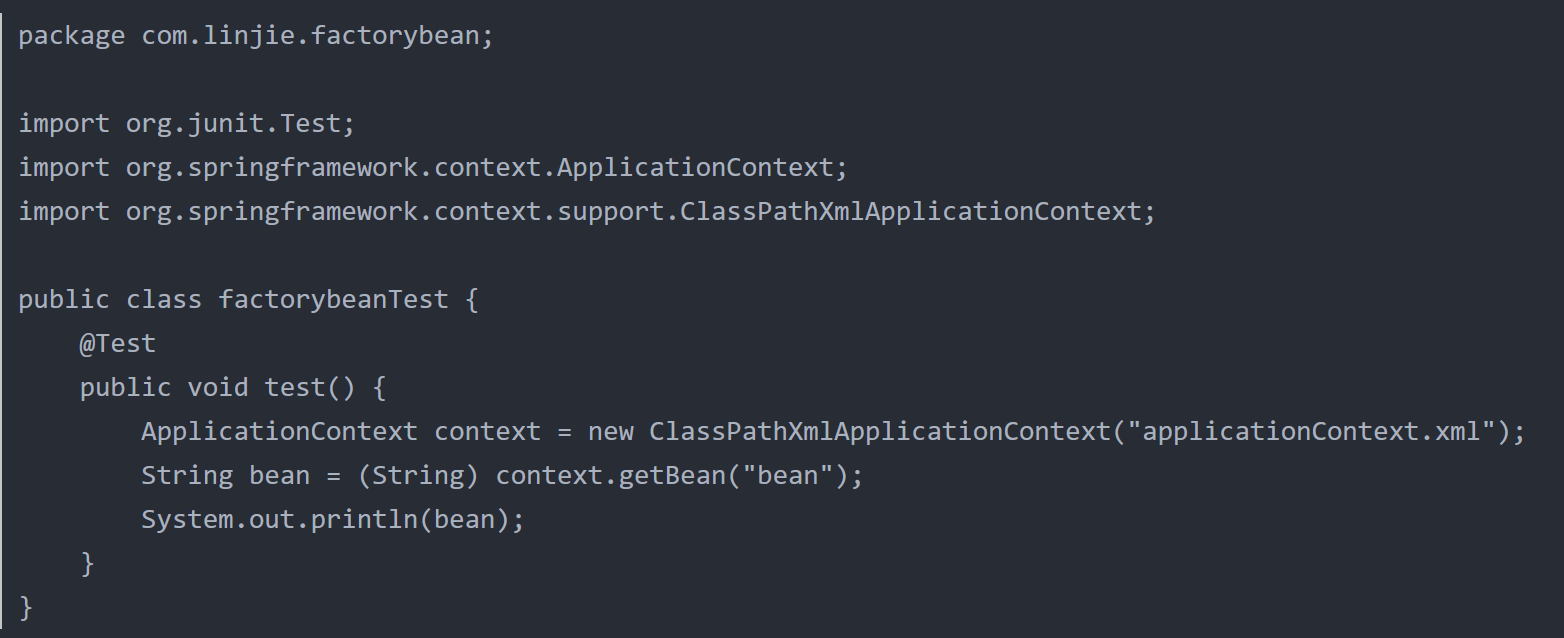




3.2 配置xml



3.3测试



## 建造者模式（Builder Pattern）

**定义：建造模式是对象的创建模式。建造模式可以将一个产品的内部表象（internal representation）与产品的生产过程分割开来，从而可以使一个建造过程生成具有不同的内部表象的产品对象。**

**产品的内部表象**

一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象，它们通常又叫做产品的内部表象（internal representation）。不同的产品可以有不同的内部表象，也就是不同的零件。使用建造模式可以使客户端不需要知道所生成的产品有哪些零件，每个产品的对应零件彼此有何不同，是怎么建造出来的，以及怎么组成产品。

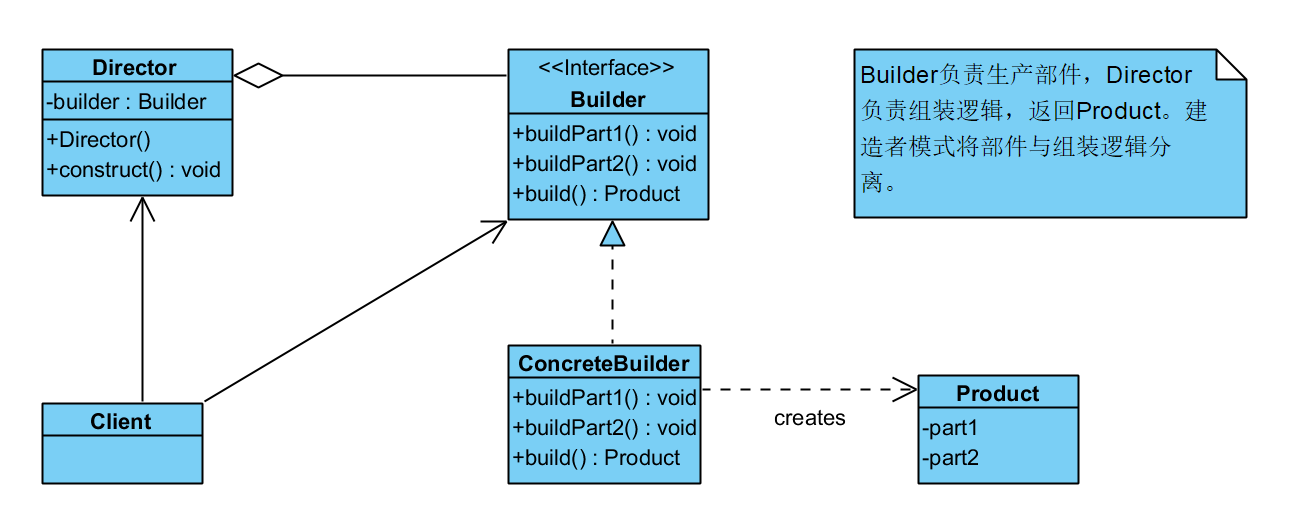
**对象性质的建造**

有些情况下，一个对象会有一些重要的性质，在它们没有恰当的值之前，对象不能作为一个完整的产品使用。比如，一个电子邮件有发件人地址、收件人地址、主题、内容、附录等部分，而在最起码的收件人地址得到赋值之前，这个电子邮件不能发送。

有些情况下，一个对象的一些性质必须按照某个顺序赋值才有意义。在某个性质没有赋值之前，另一个性质则无法赋值。这些情况使得性质本身的建造涉及到复杂的商业逻辑。这时候，此对象相当于一个有待建造的产品，而对象的这些性质相当于产品的零件，建造产品的过程是建造零件的过程。由于建造零件的过程很复杂，因此，这些零件的建造过程往往被“外部化”到另一个称做建造者的对象里，建造者对象返还给客户端的是一个全部零件都建造完毕的产品对象。

建造模式利用一个导演者对象和具体建造者对象一个个地建造出所有的零件，从而建造出完整的产品对象。建造者模式将产品的结构和产品的零件的建造过程对客户端隐藏起来，把对建造过程进行指挥的责任和具体建造者零件的责任分割开来，达到责任划分和封装的目的。

结构：



**抽象建造者（Builder）角色：**给 出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，此接口独立于应用程序的商业逻辑。模式中直接创建产品对象的是具体建造者 (ConcreteBuilder)角色。具体建造者类必须实现这个接口所要求的两种方法：一种是建造方法(buildPart1和 buildPart2)，另一种是返还结构方法(retrieveResult)。一般来说，产品所包含的零件数目与建造方法的数目相符。换言之，有多少 零件，就有多少相应的建造方法。

具体建造者（ConcreteBuilder）角色：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序调用下创建产品的实例。这个角色要完成的任务包括：1.实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。2.在建造过程完成后，提供产品的实例。

**导演者（Director）角色：**担任这个角色的类调用具体建造者角色以创建产品对象。应当指出的是，导演者角色并没有产品类的具体知识，真正拥有产品类的具体知识的是具体建造者角色。

**产品（Product）角色：**产品便是建造中的复杂对象。一般来说，一个系统中会有多于一个的产品类，而且这些产品类并不一定有共同的接口，而完全可以是不相关联的。

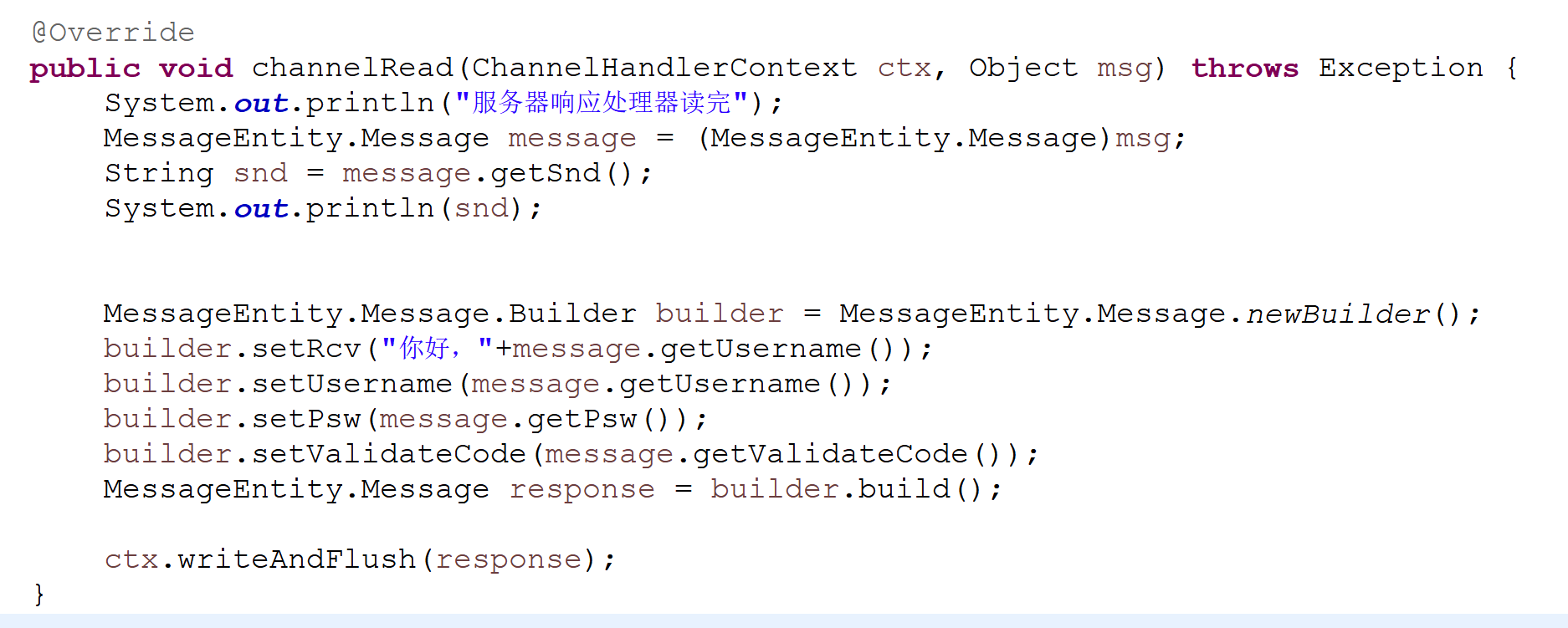
导演者角色是与客户端打交道的角色。导演者将客户端创建产品的请求划分为对各个零件的建造请求，再将这些请求委派给具体建造者角色。具体建造者角色是做具体建造工作的，但是却不为客户端所知。

一般来说，每有一个产品类，就有一个相应的具体建造者类。这些产品应当有一样数目的零件，而每有一个零件就相应地在所有的建造者角色里有一个建造方法。

代码：

designpattrn.creation.builder.Main#builder()

应用：protobuf



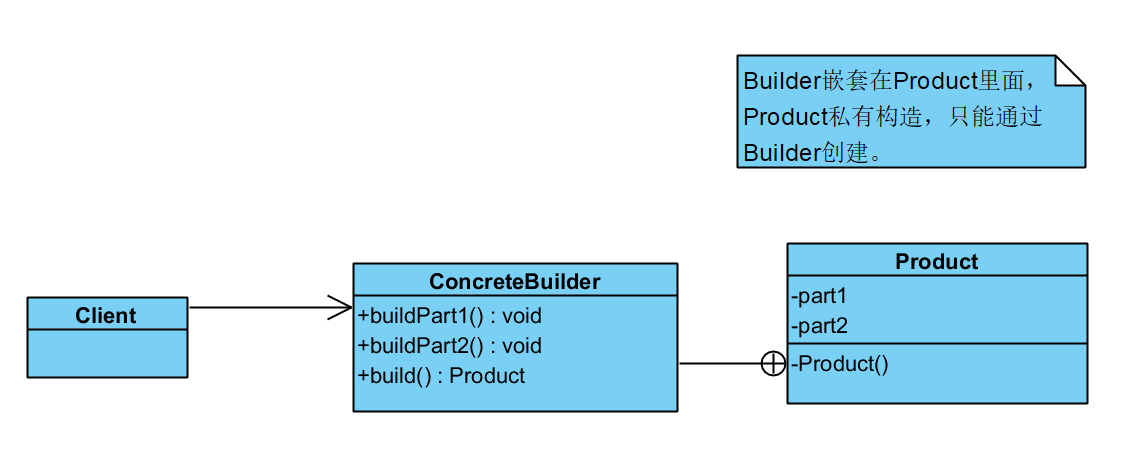
### 建造者模式简化

考虑这样一个实际应用，要创建一个保险合同的对象，里面很多属性的值都有约束，要求创建出来的对象是满足这些约束规则的。约束规则比如：保险合同通常情况下可以和个人签订，也可以和某个公司签订，但是一份保险合同不能同时与个人和公司签订。这个对象里有很多类似这样的约束，采用建造模式来构建复杂的对象，通常会对建造模式进行一定的简化，因为目标明确，就是创建某个复杂对象，因此做适当简化会使程序更简洁。大致简化如下：

1. 由于是用Builder模式来创建某个对象，因此就没有必要再定义一个Builder接口，直接提供一个具体的建造者类就可以了。
2. 对于创建一个复杂的对象，可能会有很多种不同的选择和步骤，干脆去掉“导演者”，把导演者的功能和Client的功能合并起来，也就是说,Client这个时候就相当于导演者，它来指导构建器类去构建需要的复杂对象。

简化的建造者和没有简化的建造者模式的区别是：简化的建造者模式固定了导演的剧本（construct()），而简化的制造者模式通过更换不同的导演类，construct()建造过程就会不同，更灵活。

结构：



代码：designpattrn.creation.builder.Main#simplyBuilder()

### 建造者模式和抽象工厂模式的区别

1. 建造者模式更注重的一个对象的内部，例如部件和组装逻辑，而抽象工厂模式更注重的是一个产品族，各个产品等级是否是一个产品族，不是就不能对应起来，不能产生一个有用的对象，而且抽象工厂模式能自己设置的属性会比较少，通常是一个产品族配套的。
2. 建造者模式更自由，例如一辆车用建造者模式可以使用奔驰的发动机，宝马的变速箱，而抽象工厂模式不能。

# 结构型

## 适配器模式（Adapter Pattern）

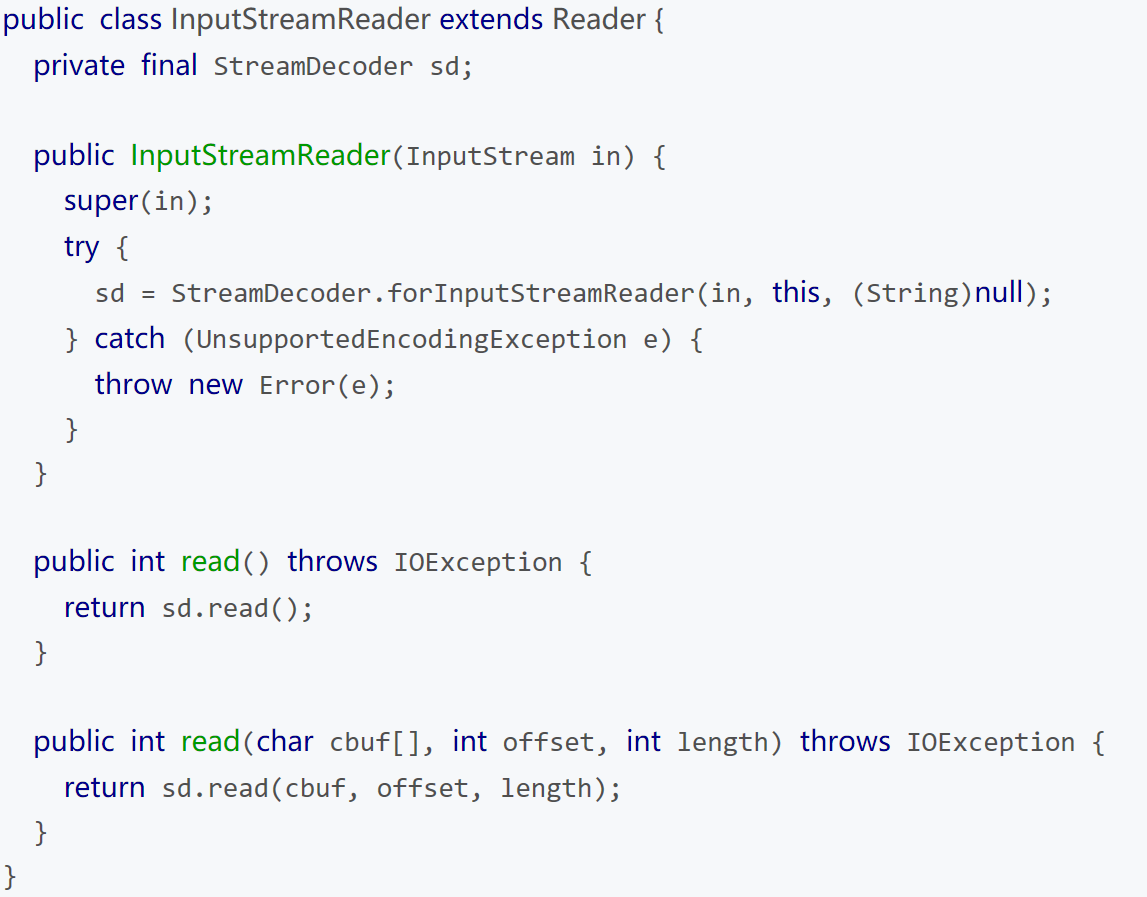
**定义：适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。**

适配器的用途：用电器做例子，笔记本电脑的插头一般都是三相的，即除了阳极、阴极外，还有一个地极。而有些地方的电源插座却只有两极，没有地极。电源插座与笔记本电脑的电源插头不匹配使得笔记本电脑无法使用。这时候一个三相到两相的转换器（适配器）就能解决此问题，而这正像是本模式所做的事情。

适配器模式有类的适配器模式和对象的适配器模式两种不同的形式。

应用：

java.io.InputStreamReader(InputStream)



InputStreamReader是字节流转换为字符流的桥梁(适配器)。其中，Reader对应Target(接口或抽象类)，InputStreamReader对应Adapter，InputStream对应Adaptee。

java.io.OutputStreamWriter(OutputStream)

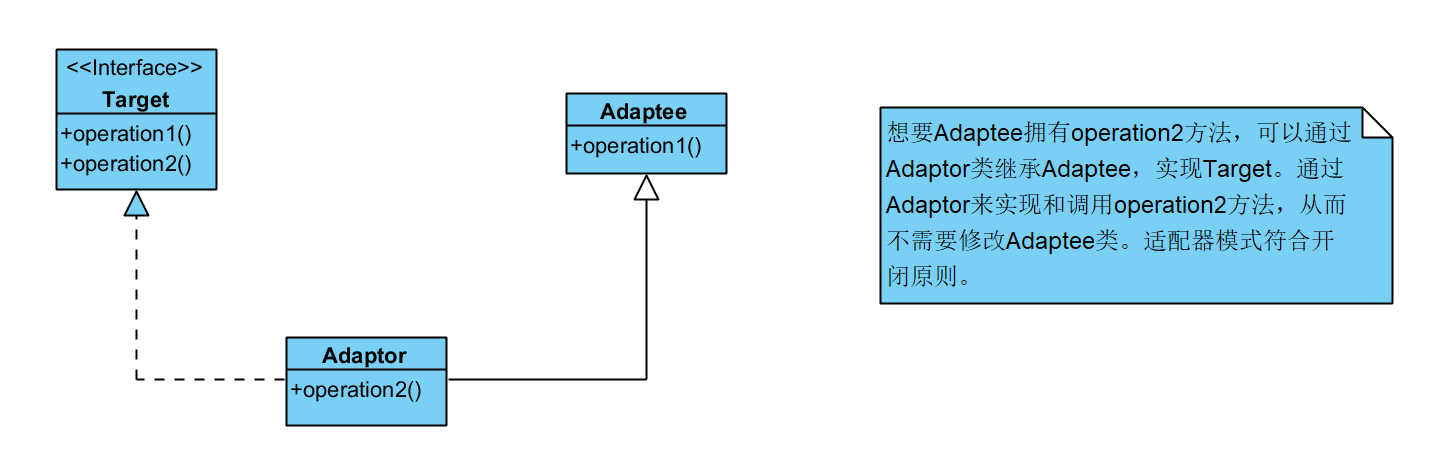
java.util.Collections#synchronizedCollection()

java.util.Arrays#asList()

### 类的适配器模式

类的适配器模式把适配的类的API转换成为目标类的API。

结构：



　模式所涉及的角色有：

　目标(Target)角色：这就是所期待得到的接口。注意：由于这里讨论的是类适配器模式，因此目标不可以是类。

　源(Adapee)角色：现在需要适配的接口。

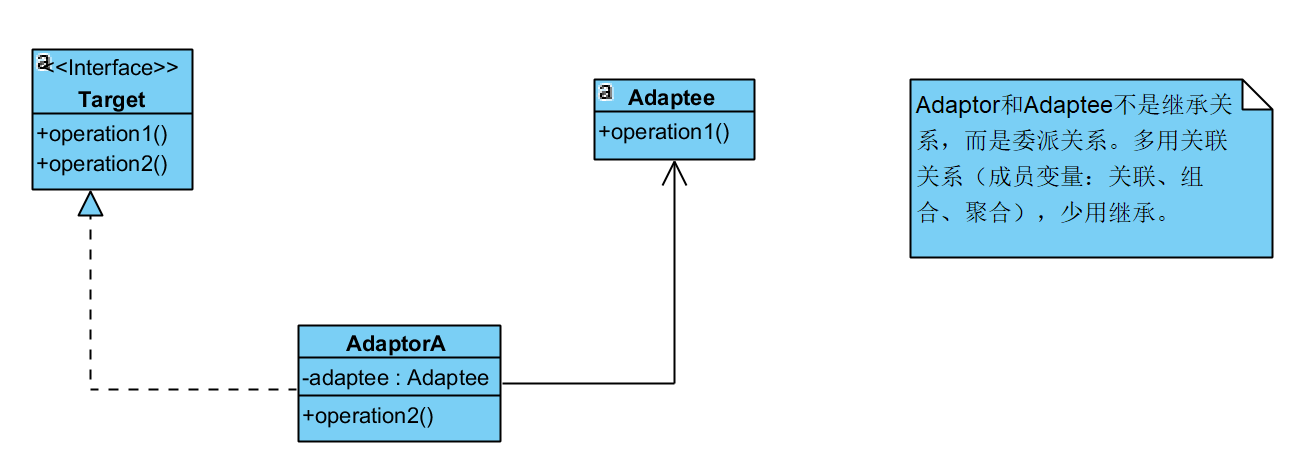
　适配器(Adaper)角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，而必须是具体类。

代码：designpattrn.structure.adapter.Main.classAdapter()

### 对象的适配器模式

与类的适配器模式一样，对象的适配器模式把被适配的类的API转换成为目标类的API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到Adaptee类，而是使用委派关系连接到Adaptee类。

结构：



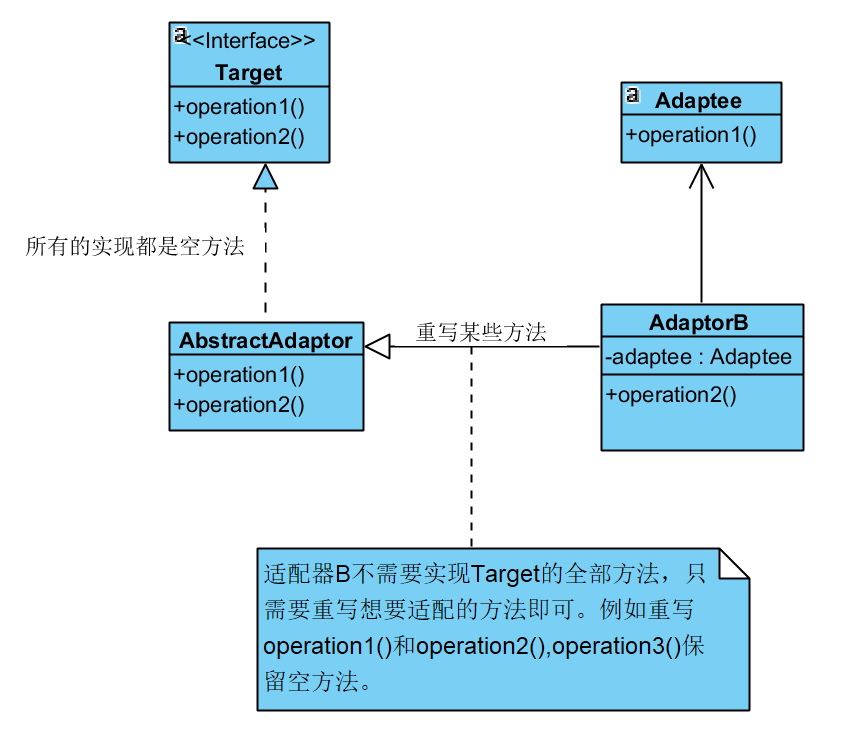
代码：designpattrn.structure.adapter.Main.objectAdapter()

### 缺省适配器模式

缺省适配(Default Adapter)模式为一个接口提供缺省实现，这样子类型可以从这个缺省实现进行扩展，而不必从原有接口进行扩展。作为适配器模式的一个特例，缺省是适配模式在JAVA语言中有着特殊的应用。

缺省适配器模式中，适配器不在实现接口，而是继承一个抽象适配器类，抽象适配器类实现Target接口，实现的内容为全部空方法，这样适配器就不必实现Target接口的所有方法。

结构：



代码：designpattrn.structure.adapter.Main.defaultAdapter()

### 类适配器与对象适配器的权衡

1. 类适配器使用对象继承的方式，是静态的定义方式；而对象适配器使用对象组合的方式，是动态组合的方式。
2. 对于类适配器，由于适配器直接继承了Adaptee，使得适配器不能和Adaptee的子类一起工作，因为继承是静态的关系，当适配器继承了Adaptee后，就不可能再去处理  Adaptee的子类了。

对于对象适配器，一个适配器可以把多种不同的源适配到同一个目标。换言之，同一个适配器可以把源类和它的子类都适配到目标接口。因为对象适配器采用的是对象组合的关系，只要对象类型正确，是不是子类都无所谓。

1. 对于类适配器，适配器可以重定义Adaptee的部分行为，相当于子类覆盖父类的部分实现方法。

对于对象适配器，要重定义Adaptee的行为比较困难，这种情况下，需要定义Adaptee的子类来实现重定义，然后让适配器组合子类。虽然重定义Adaptee的行为比较困难，但是想要增加一些新的行为则方便的很，而且新增加的行为可同时适用于所有的源。

1. 对于类适配器，仅仅引入了一个对象，并不需要额外的引用来间接得到Adaptee。对于对象适配器，需要额外的引用来间接得到Adaptee。
2. 建议尽量使用对象适配器的实现方式，多用合成/聚合、少用继承。当然，具体问题具体分析，根据需要来选用实现方式，最适合的才是最好的。

### 多用关联少用继承

继承和关联都可以达到复用代码的目的，继承复用又叫“白盒复用”，父类的内部细节对于子类来说是可见的。组合复用又叫“黑盒复用”，因为当前对象只能委托所包含的对象调用其方法，这样一来，当前对象所包含的对象方法的细节对当前对象是不可见的。

**继承的优点：**

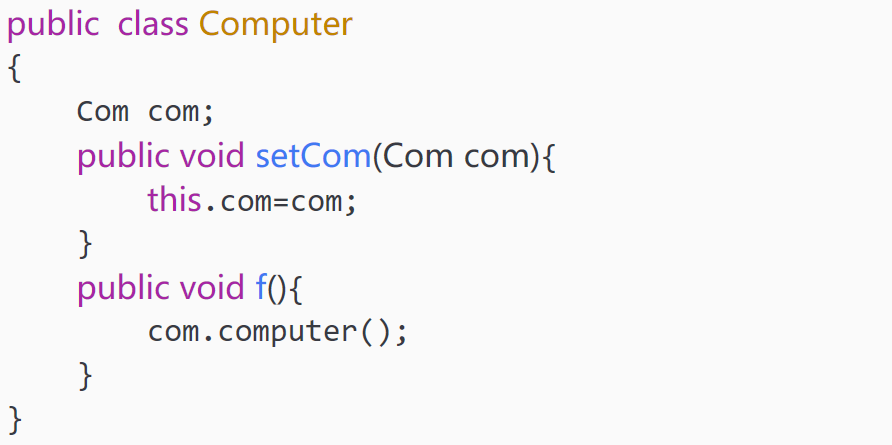
1. 方便扩展。

**继承的缺点：**

1. 继承来的方法在编译期就已经确定，无法在运行期间改变从父类继承来的方法的行为。
2. 子类和父类是强耦合关系，父类改变，子类必须改变。

**组合的优点：**

1. 对象与所包含的对象是弱耦合关系，因为，如果修改当前对象所包含对象类的代码，不必修改当前对象的代码。
2. 当前对象可以在运行时动态指定所包含的对象，例如，假设Com是一个接口，该接口中有一个computer()方法，那么下列Computer类的对象可以在运行是动态指定所包含的对象，即运行期间，Computer类的实例可以调用setCom(Com com)方法将其中的com变量存放任何实现Com接口对象的引用。



**组合的缺点：**

1. 容易导致系统中的对象过多。
2. 为了组合多个对象，对于接口的定义必须仔细。

### 适配器模式的优缺点

优点：

1. 更好的复用性

系统需要使用现有的类，而此类的接口不符合系统的需要。那么通过适配器模式就可以让这些功能得到更好的复用。

1. 更好的扩展性

在实现适配器功能的时候，可以调用自己开发的功能，从而自然地扩展系统的功能。

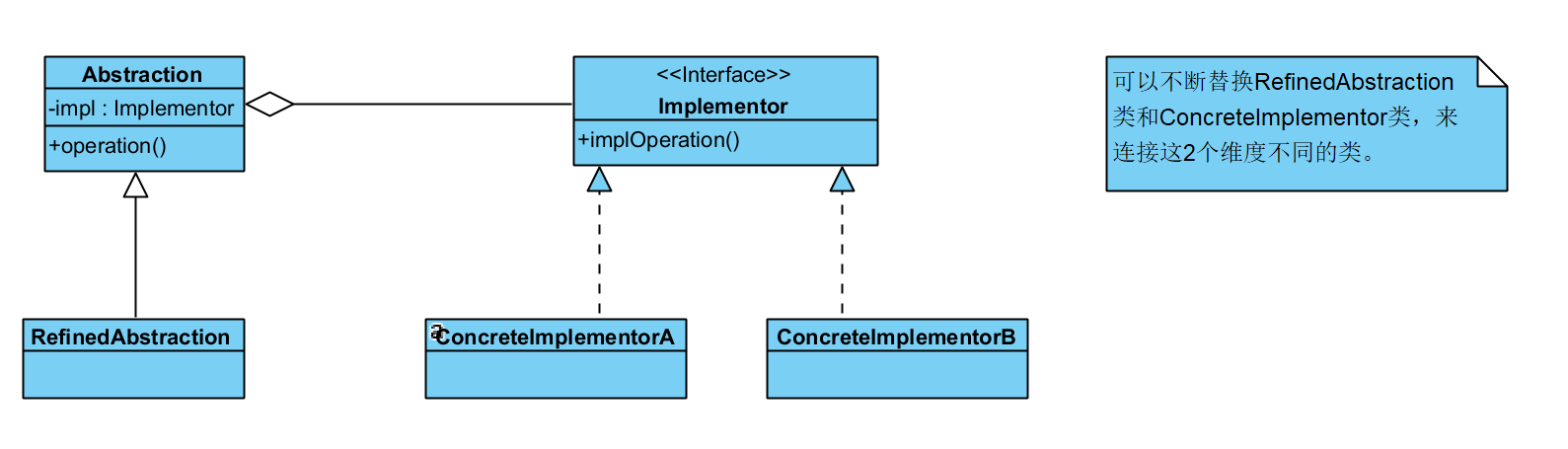
缺点：

过多的使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。

## 桥接模式（Bridge Pattern）

**定义：桥梁模式是对象的结构模式。又称为柄体(Handle and Body)模式或接口(Interface)模式。桥梁模式的用意是“将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化”。**

结构：抽象类聚合接口，从而连接抽象类的实现类和接口的实现类。



可以看出，这个系统含有两个等级结构：

1. 由抽象化角色和修正抽象化角色组成的抽象化等级结构。
2. 由实现化角色和两个具体实现化角色所组成的实现化等级结构。

桥梁模式所涉及的角色有：

1. 抽象化(Abstraction)角色：抽象化给出的定义，并保存一个对实现化对象的引用。
2. 修正抽象化(RefinedAbstraction)角色：扩展抽象化角色，改变和修正父类对抽象化的定义。
3. 实现化(Implementor)角色：这个角色给出实现化角色的接口，但不给出具体的实现。必须指出的是，这个接口不一定和抽象化角色的接口定义相同，实际上，这两个接口可以非常不一样。实现化角色应当只给出底层操作，而抽象化角色应当只给出基于底层操作的更高一层的操作。
4. 具体实现化(ConcreteImplementor)角色：这个角色给出实现化角色接口的具体实现。

抽象化角色就像是一个水杯的手柄，而实现化角色和具体实现化角色就像是水杯的杯身。手柄控制杯身，这就是此模式别名“柄体”的来源。

对象是对行为的封装，而行为是由方法实现的。在这个示意性系统里，抽象化等级结构中的类封装了operation()方法；而实现化等级结构中的类封装的是operationImpl()方法。当然，在实际的系统中往往会有多于一个的方法。

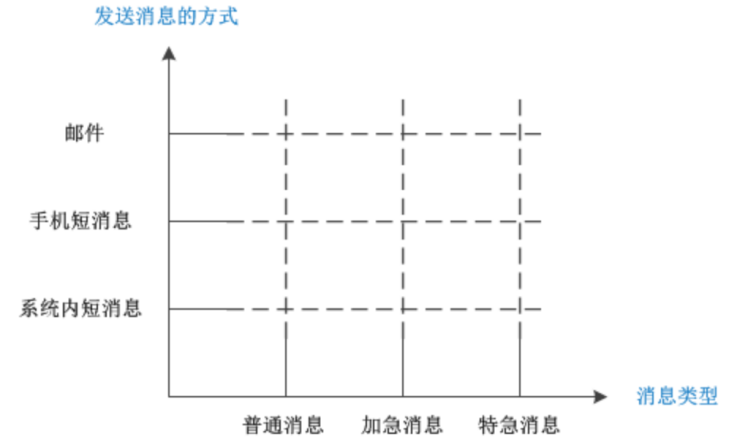
抽象化等级结构中的方法通过向对应的实现化对象的委派实现自己的功能，这意味着抽象化角色可以通过向不同的实现化对象委派，来达到动态地转换自己的功能的目的。

代码：designpattrn.structure.bridge.Main.main(String[])

应用场景：

两个类要关联，但是这两个类又有相似的类可以排列组合关联，可以把这些类抽象成一个抽象类和一个接口，这两个类分别继承抽象类和实现接口，在抽象层关联，在实现层可以自由切换。

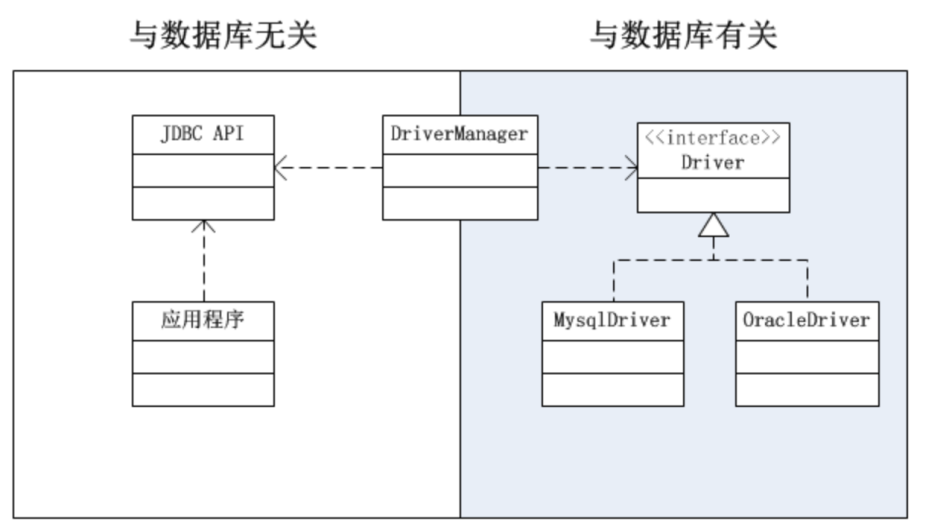
根据业务的功能要求，业务的变化具有两个维度，一个维度是抽象的消息，包括普通消息、加急消息和特急消息，这几个抽象的消息本身就具有一定的关系，加急消息和特急消息会扩展普通消息；另一个维度是在具体的消息发送方式上，包括系统内短消息、邮件和手机短消息，这几个方式是平等的，可被切换的方式。



符合面向对象原则：

1. 开闭原则，对新增代码（新增实现）开放，对修改代码关闭。

### 桥接模式在JDBC的使用



应用系统作为一个等级结构，与JDBC驱动器这个等级结构是相对独立的，它们之间没有静态的强关联。应用系统通过委派与JDBC驱动器相互作用，这是一个桥梁模式的例子。

JDBC的这种架构，把抽象部分和具体部分分离开来，从而使得抽象部分和具体部分都可以独立地扩展。对于应用程序而言，只要选用不同的驱动，就可以让程序操作不同的数据库，而无需更改应用程序，从而实现在不同的数据库上移植；对于驱动程序而言，为数据库实现不同的驱动程序，并不会影响应用程序。

## 组合模式（Composite Pattern）

**定义：将对象组合成树形结构以表示‘部分-整体’的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。**

理解：在一个树形结构的组合对象里，无论这个对象是是否是叶子节点，客户端对这个数据结构的方法的调用是一样的，只是叶子节点和非叶子节点的方法实现不一样。如果是叶子节点，则调用叶子节点对象的方法，如果不是叶子节点，则递归遍历这个节点的子节点。

使用场景：

1. 当想表达对象的部分-整体的层次结构时。例如文件夹，公司部门这些结构。
2. 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象时。

优缺点：

1. 优点：

（1）、简化客户端操作。客户端只需要面对一致的对象而不用考虑整体部分或者节点叶子的问题。

（2）、具有较强的扩展性。当我们要更改组合对象时，我们只需要调整内部的层次关系，客户端不用做出任何改动。

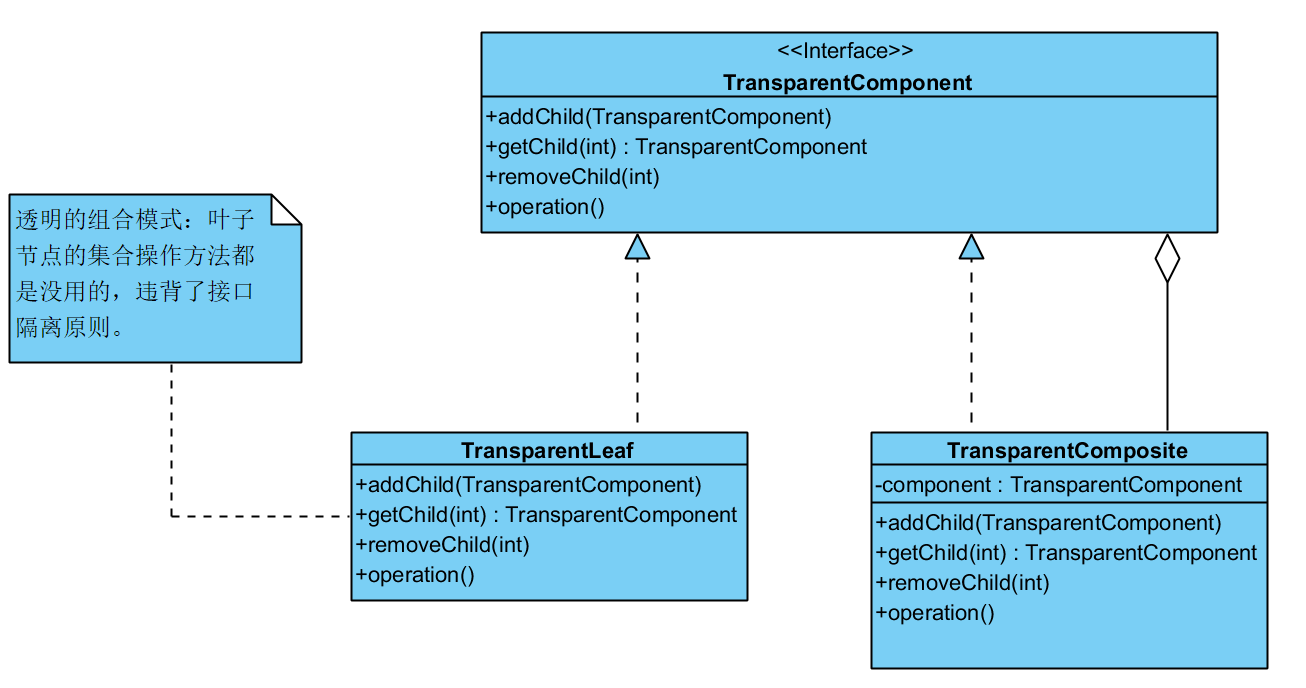
（3）、方便创建出复杂的层次结构。客户端不用理会组合里面的组成细节，容易添加节点或者叶子从而创建出复杂的树形结构。

1. 缺点：
2. 、要求较高的抽象性，如果节点和叶子有很多差异性的话，比如很多方法和属性都不一样，难以实现组合模式。

### 透明的组合模式

透明的组合模式，客户端声明树枝节点和叶子节点都用抽象类型的引用。对于客户端，树枝节点和叶子结点没有区别，但是由于叶子节点没有add，remove，getChild(int i)等方法，又实现了这个接口，叶子节点实现这3个方法只能抛异常或者使用空方法，对于叶子节点这3个方法没有用，违背了接口隔离原则。

结构：



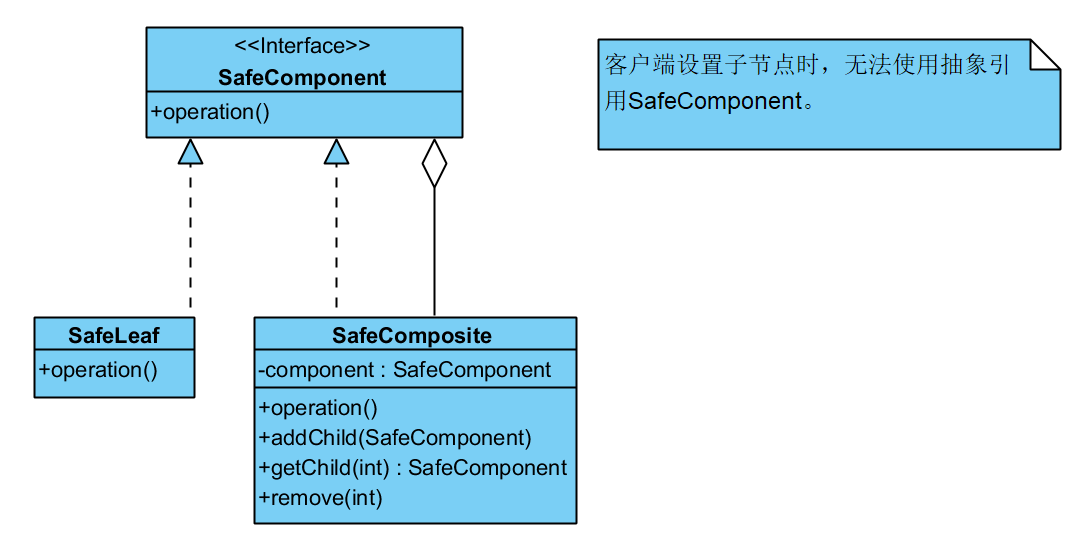
代码：designpattrn.structure.composite.Main.transparentComposite()

### 安全的组合模式

使用安全的组合模式，符合了接口隔离原则，但是客户端在声明树枝节点和叶子节点时就不能统一用抽象引用，只能用具体的引用。这样树枝节点的引用类型和叶子节点的引用类型不同，客户端还需要了解整个树形结构内部的不同，这是不透明的。

建议使用安全的组合模式，不使用透明的组合模式。

结构：



代码：designpattrn.structure.composite.Main.safeComposite()

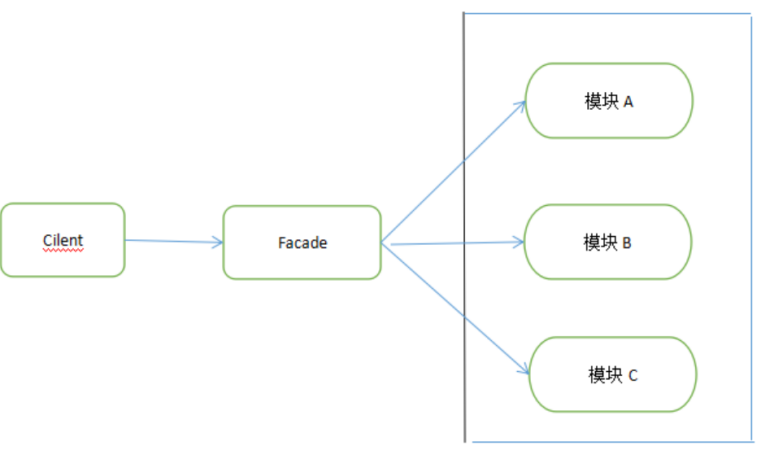
## 外观模式**（Facade Pattern）**

**定义：外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。**

这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

外观模式主要解决的问题是：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。

结构：没有专门的类图表示。



1. 门面角色：外观模式的核心。它被客户角色调用，它熟悉子系统的功能。内部根据客户角色的需求预定了几种功能的组合。
2. 子系统角色:实现了子系统的功能。它对客户角色和Facade时未知的。它内部可以有系统内的相互交互，也可以由供外界调用的接口。
3. 客户角色:通过调用Facede来完成要实现的功能。

代码：designpattrn.structure.facade.Main.main(String[])

优点：

1. 客户端可以通过简单的方法调用复杂的流程。
2. 使用外观类可以分割对外暴露的方法和内部使用的方法。
3. 使用外观类可以选择性的暴露方法。

使用场景：

1. 为复杂的模块或子系统提供外界访问的模块；
2. 子系统相互独立；
3. 在层析结构中，可以使用外观模式定义系统的每一层的入口。

实际例子：

Jquery简化原生js比较长函数名和封装多个方法的调用，还兼容了不同浏览器。

Jquery的绑定事件：

$(target).on(eventType,fn)

原生js绑定事件并兼容浏览器：

if(dom.addEventListener){

dom.addEventListener(type,fn,false)

} else if(dom.attachEvent) {

dom.attachEvent('on'+type,fn)

} else {

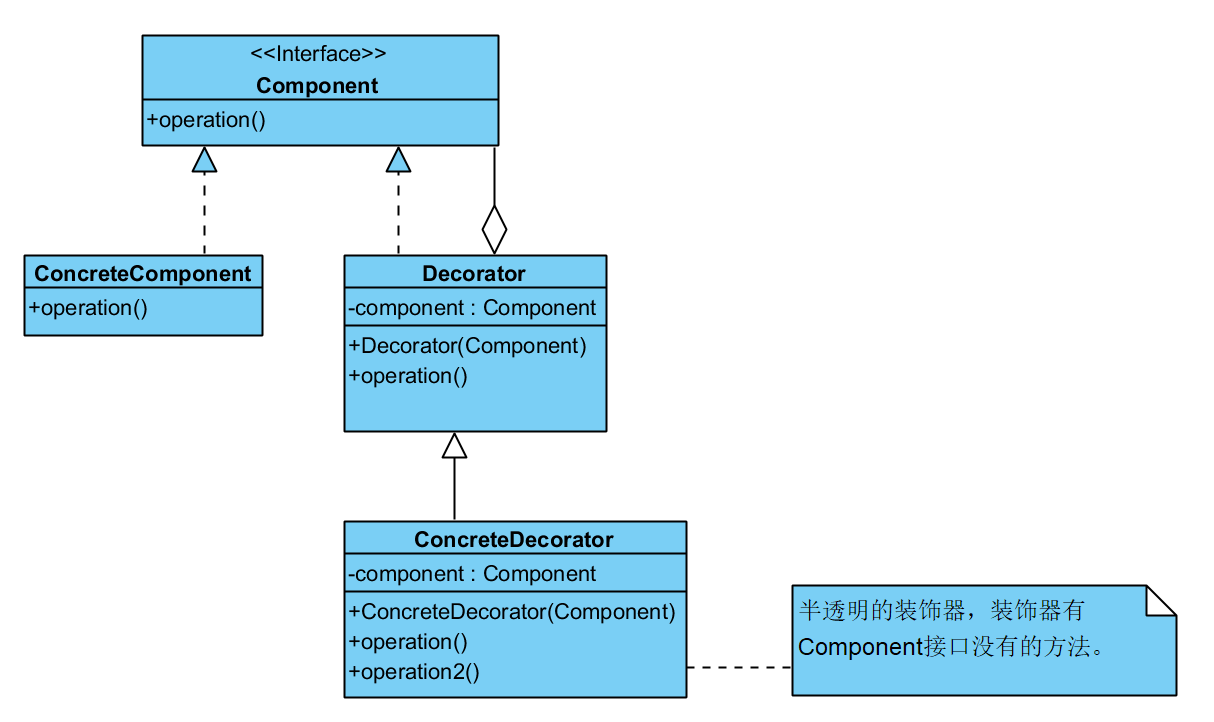
dom['on'+type] = fn

}

## 装饰模式（Decorator Pattern）

装饰模式指的是在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。

结构：



在装饰模式中的角色有：

1. 抽象构件(Component)角色：给出一个抽象接口，以规范准备接收附加责任的对象。
2. 具体构件(ConcreteComponent)角色：定义一个将要接收附加责任的类。
3. 装饰(Decorator)角色：持有一个构件(Component)对象的实例，并定义一个与抽象构件接口一致的接口。
4. 具体装饰(ConcreteDecorator)角色：负责给构件对象“贴上”附加的责任。

代码：designpattrn.structure.decorator.Main.main(String[])，同《Head First设计模式》的星巴克案例。

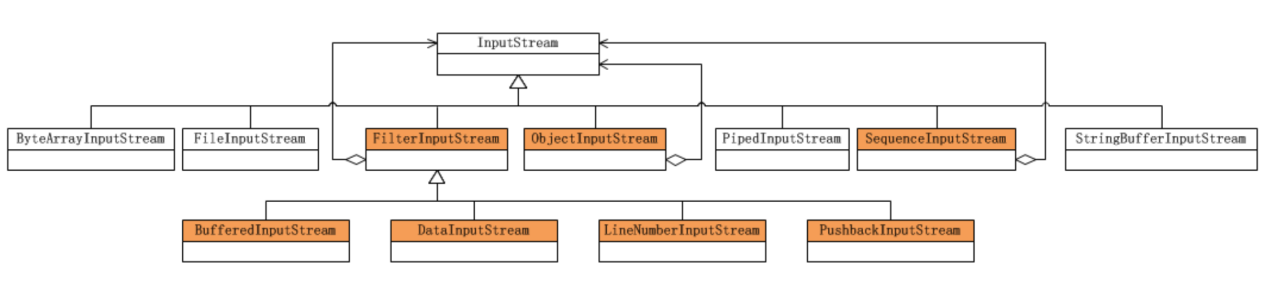
优点：

1. 装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。装饰模式允许系统动态决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。继承关系则不同，继承关系是静态的，它在系统运行前就决定了。
2. 通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

缺点：

1. 由于使用装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是，在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象看上去都很相像。

### Java I/O库中的应用



根据上图可以看出：

1. 抽象构件(Component)角色：由InputStream扮演。这是一个抽象类，为各种子类型提供统一的接口。
2. 具体构件(ConcreteComponent)角色：由ByteArrayInputStream、FileInputStream、PipedInputStream、StringBufferInputStream等类扮演。它们实现了抽象构件角色所规定的接口。
3. 抽象装饰(Decorator)角色：由FilterInputStream扮演。它实现了InputStream所规定的接口。
4. 具体装饰(ConcreteDecorator)角色：由几个类扮演，分别是BufferedInputStream、DataInputStream以及两个不常用到的类LineNumberInputStream、PushbackInputStream。

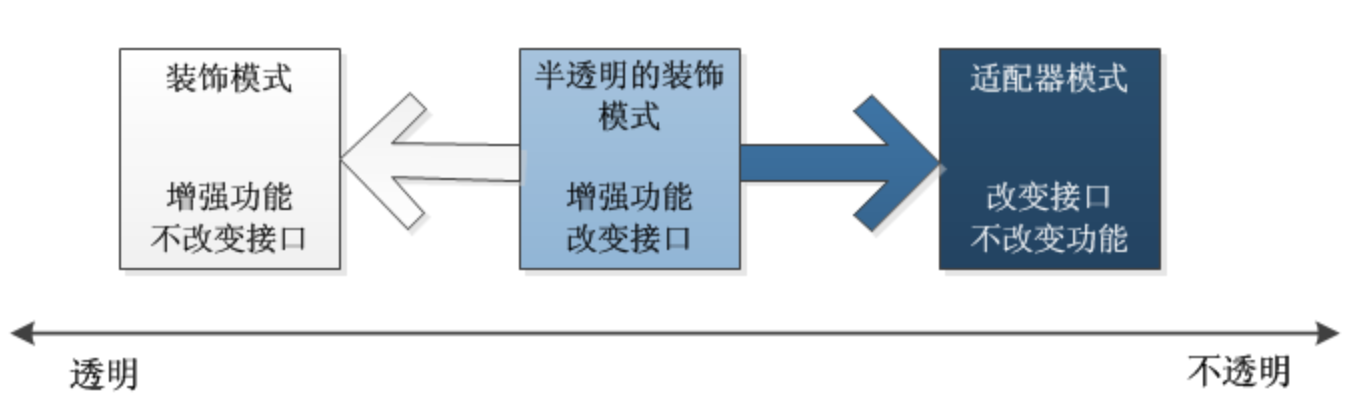
I/O流的装饰者模式都是半透明的装饰者模式，装饰类由一些方法是源没有的。

### 半透明的装饰者模式

装饰模式和适配器模式都是“包装模式(Wrapper Pattern)”，它们都是通过封装其他对象达到设计的目的的，但是它们的形态有很大区别。

理想的装饰模式在对被装饰对象进行功能增强的同时，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。而适配器模式则不然，一般而言，适配器模式并不要求对源对象的功能进行增强，但是会改变源对象的接口，以便和目标接口相符合。

装饰模式有透明和半透明两种，这两种的区别就在于装饰角色的接口与抽象构件角色的接口是否完全一致。透明的装饰模式也就是理想的装饰模式，要求具体构件角色、装饰角色的接口与抽象构件角色的接口完全一致。相反，如果装饰角色的接口与抽象构件角色接口不一致，也就是说装饰角色的接口比抽象构件角色的接口宽的话，装饰角色实际上已经成了一个适配器角色，这种装饰模式也是可以接受的，称为“半透明”的装饰模式，如下图所示。



装饰模式和适配器模式的异同

相同：

1. 对象的适配器模式和装饰模式都适配器/装饰者聚合源对象。

不同：

1. 适配器模式是让源角色具有源角色所不具备的接口，而适装饰模式的目的在于适配器与源角色具有相同的接口，增强或修改接口的实现。

## 享元模式（Flyweight Pattern）

享元模式（Flyweight Pattern）主要用于减少创建对象的数量，以减少内存占用和提高性能。这种类型的设计模式属于结构型模式，它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。

享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。

享元模式的优点在于它大幅度地降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

1. 享元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化。
2. 享元模式将享元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

作用：减少频繁创建相同或相似的对象，用于优化性能。

应用例子：

1. String的常量池
2. 其他XX池

### 外部状态和内部状态

　享元模式采用一个共享来避免大量拥有相同内容对象的开销。这种开销最常见、最直观的就是内存的损耗。享元对象能做到共享的关键是区分内部状态(Internal State)和外部状态(External State)。

一个内部状态是存储在享元对象内部的，并且是不会随环境的改变而有所不同。因此，一个享元可以具有内部状态并可以共享。

一个外部状态是随环境的改变而改变的、不可以共享的。享元对象的外部状态必须由客户端保存，并在享元对象被创建之后，在需要使用的时候再传入到享元对象内部。外部状态不可以影响享元对象的内部状态，它们是相互独立的。

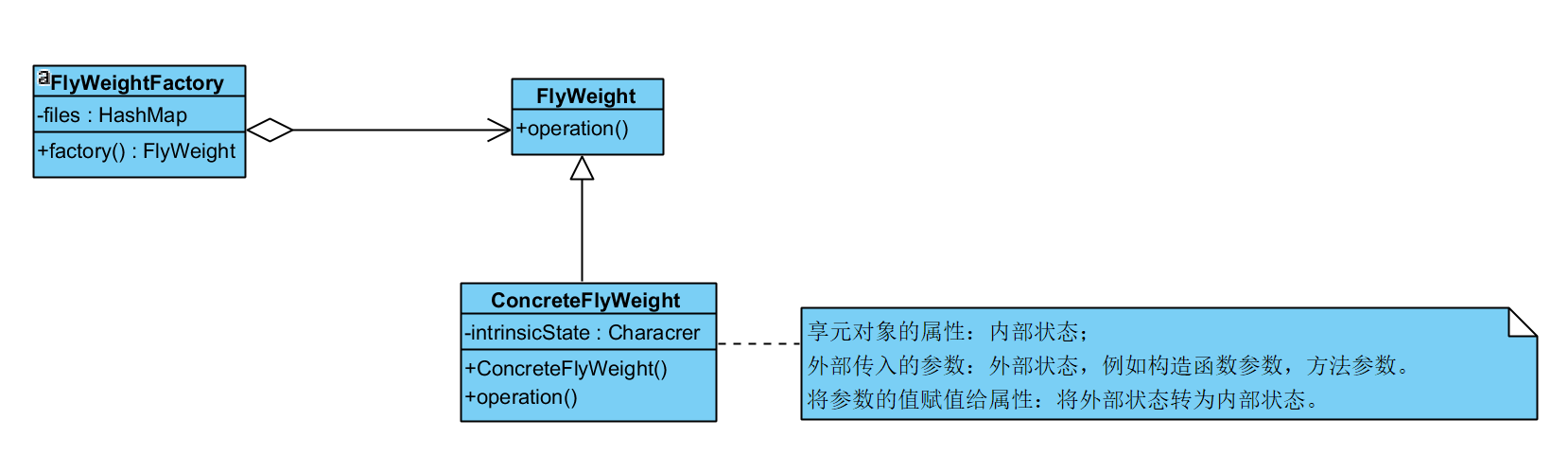
理解：

1. 享元对象的属性：内部状态；
2. 外部传入的参数：外部状态，例如构造函数参数，方法参数。
3. 将参数的值赋值给属性：将外部状态转为内部状态。

### 单纯的享元模式

共享的对象是一个简单对象。

结构：



单纯享元模式所涉及到的角色如下：

抽象享元(Flyweight)角色 ：给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定出的接口。如果有内部状态的话，必须负责为内部状态提供存储空间。

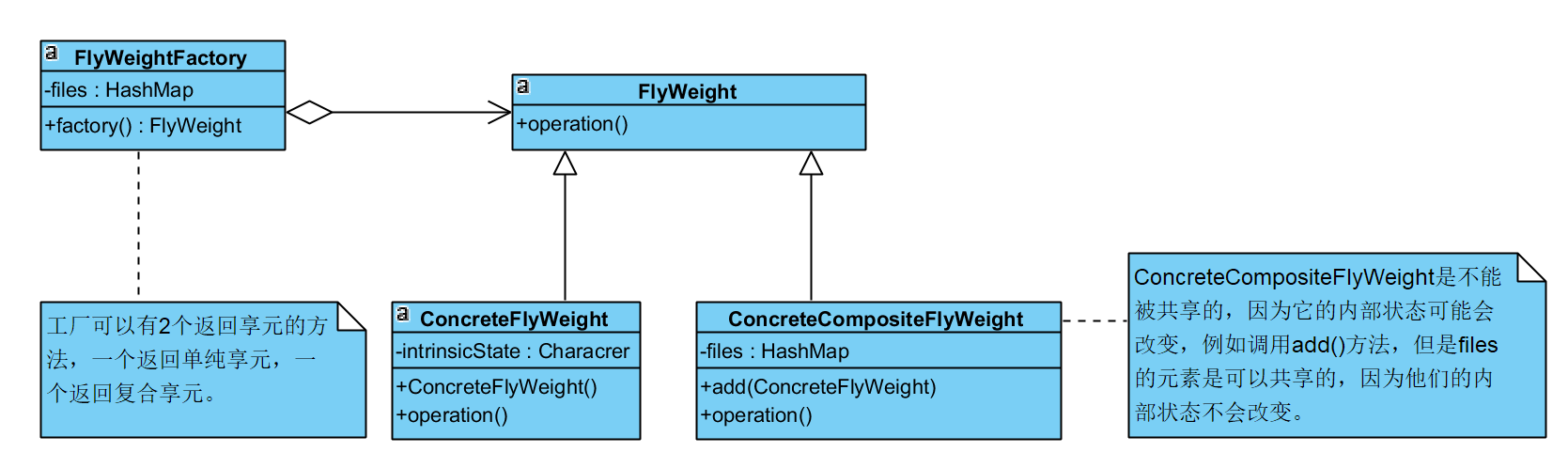
享元工厂(FlyweightFactory)角色 ：本角色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个合适的享元对象。

代码：designpattrn.structure.flyweight.Main.flyWeight()

### 复合享元模式

共享的对象是一个集合。集合这个对象不能共享，但是集合里面的元素是可以共享的。

结构：



复合享元角色所涉及到的角色如下：

抽象享元(Flyweight)角色 ：给出一个抽象接口，以规定出所有具体享元角色需要实现的方法。

具体享元(ConcreteFlyweight)角色：实现抽象享元角色所规定出的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。

复合享元(ConcreteCompositeFlyweight)角色 ：复合享元角色所代表的对象是不可以共享的，但是一个复合享元对象可以分解成为多个本身是单纯享元对象的组合。复合享元角色又称作不可共享的享元对象。

复合享元对象是由单纯享元对象通过复合而成的，因此它提供了add()这样的聚集管理方法。由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象被创建之后加入，这本身就意味着复合享元对象的状态是会改变的，因此复合享元对象是不能共享的。

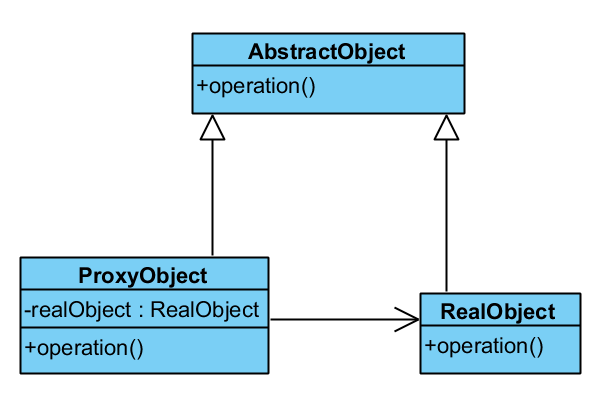
享元工厂(FlyweightFactory)角色 ：本角 色负责创建和管理享元角色。本角色必须保证享元对象可以被系统适当地共享。当一个客户端对象调用一个享元对象的时候，享元工厂角色会检查系统中是否已经有 一个符合要求的享元对象。如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个 合适的享元对象。

代码：designpattrn.structure.flyweight.Main.compositeFlyWeight()

## 代理模式（Proxy Pattern）

代理模式是对象的结构模式。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

结构：



在代理模式中的角色：

抽象对象角色：声明了目标对象和代理对象的共同接口，这样一来在任何可以使用目标对象的地方都可以使用代理对象。

目标对象角色：定义了代理对象所代表的目标对象。

代理对象角色：代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象。

代码：designpattrn.structure.proxy.Main.main(String[])

### 保护代理

保护（Protect or Access）代理：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。

### 远程代理

远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同地址空间可以是在本机器中，也可以是另一台机器中。远程代理又叫大使（Ambassador）。

远程代理可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。客户端完全可以认为被代理的远程业务对象是在本地而不是在远程，而远程代理对象承担了大部分的网络通信工作，并负责对远程业务方法的调用。

在基于.NET平台的分布式技术，例如DCOM(Distribute Component Object Model，分布式组件对象模型)、Web Service中，都应用了远程代理模式，大家可以查阅相关资料进行扩展学习。

### 虚拟代理

虚拟代理(Virtual Proxy)也是一种常用的代理模式，对于一些占用系统资源较多或者加载时间较长的对象，可以给这些对象提供一个虚拟代理。在真实对象创建成功之前虚拟代理扮演真实对象的替身，而当真实对象创建之后，虚拟代理将用户的请求转发给真实对象。

通常，在以下两种情况下可以考虑使用虚拟代理：

1. 由于对象本身的复杂性或者网络等原因导致一个对象需要较长的加载时间，此时可以用一个加载时间相对较短的代理对象来代表真实对象。通常在实现时可以结合多线程技术，一个线程用于显示代理对象，其他线程用于加载真实对象。这种虚拟代理模式可以应用在程序启动的时候，由于创建代理对象在时间和处理复杂度上要少于创建真实对象，因此，在程序启动时，可以用代理对象代替真实对象初始化，大大加速了系统的启动时间。当需要使用真实对象时，再通过代理对象来引用，而此时真实对象可能已经成功加载完毕，可以缩短用户的等待时间。例如Java的Future模式。
2. 当一个对象的加载十分耗费系统资源的时候，也非常适合使用虚拟代理。虚拟代理可以让那些占用大量内存或处理起来非常复杂的对象推迟到使用它们的时候才创建，而在此之前用一个相对来说占用资源较少的代理对象来代表真实对象，再通过代理对象来引用真实对象。为了节省内存，在第一次引用真实对象时再创建对象，并且该对象可被多次重用，在以后每次访问时需要检测所需对象是否已经被创建，因此在访问该对象时需要进行存在性检测，这需要消耗一定的系统时间，但是可以节省内存空间，这是一种用时间换取空间的做法。例如Hibernate的懒加载。

无论是以上哪种情况，虚拟代理都是用一个“虚假”的代理对象来代表真实对象，通过代理对象来间接引用真实对象，可以在一定程度上提高系统的性能。

### 缓冲代理

缓冲代理(Cache Proxy)也是一种较为常用的代理模式，它为某一个操作的结果提供临时的缓存存储空间，以便在后续使用中能够共享这些结果，从而可以避免某些方法的重复执行，优化系统性能。例如Hibernate的缓存管理，直接操作代理对象（session），而不是操作被代理对象（Connection查询或者map获取缓存）。

### 防火墙代理

防火墙（Firewall）代理：保护目标，不然恶意用户接近。

### 同步化代理

同步化（Synchronization）代理：使多个用户同时使用一个对象而没有冲突。

### 智能引用代理

智能引用（SmartReference）代理：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。

在所有种类的代理模式中，虚拟（Virtual）代理、远程（Remote）代理、智能引用代理（SmartReference Proxy）和保护（Protector Access）代理是最为常见的代理模式。

### 代理模式、装饰模式、适配器模式的异同

相同：

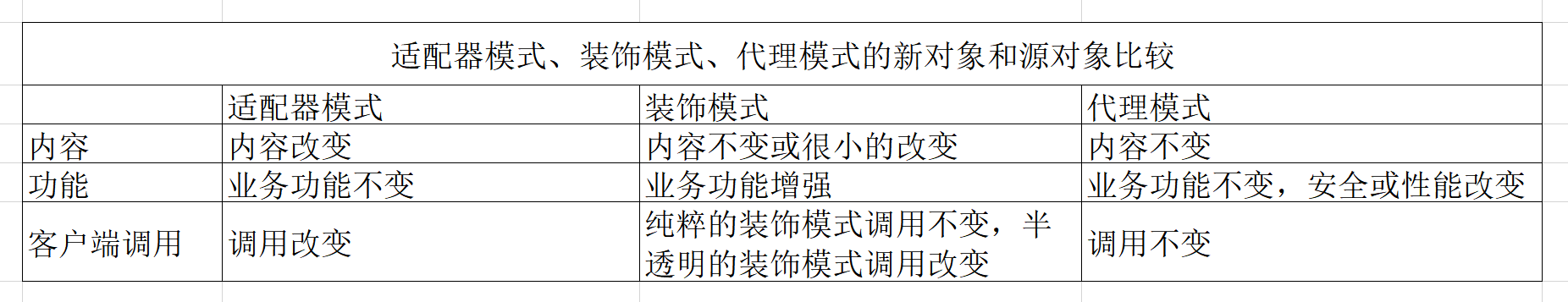
1. 都是代理对象/装饰器对象/适配器对象关联（关联、聚合、组合，最为成员变量）源对象。
2. 新对象方法的实现都依赖于源对象对应的方法。

不同：

1. 目的不同

适配器模式的目的是为了转换接口（同一件事不同做法，例如中文书适配成英语书，法语数；字节流适配成字符流，）。装饰者模式的目的是为了增强功能。代理模式的目的可能是为了保护源对象，不让客户端访问源对象，可能是为了解决性能或资源问题使用懒加载或者使用缓存。

1. 新对象和源对象改变的不同



适配器模式会改变内容，但功能不变。例如《Head Fiset设计模式》，中文版和英文版的内容不同，一个写中文，一个写英文，但是他们的业务功能都相同，读完之后都能学到一样的知识。

装饰模式内容会改变，业务功能也会改变。例如《Head Fiset设计模式》，普通版和限量珍藏版，内容一样，但是限量珍藏版多了一张设计模式和设计原则总结的海报。

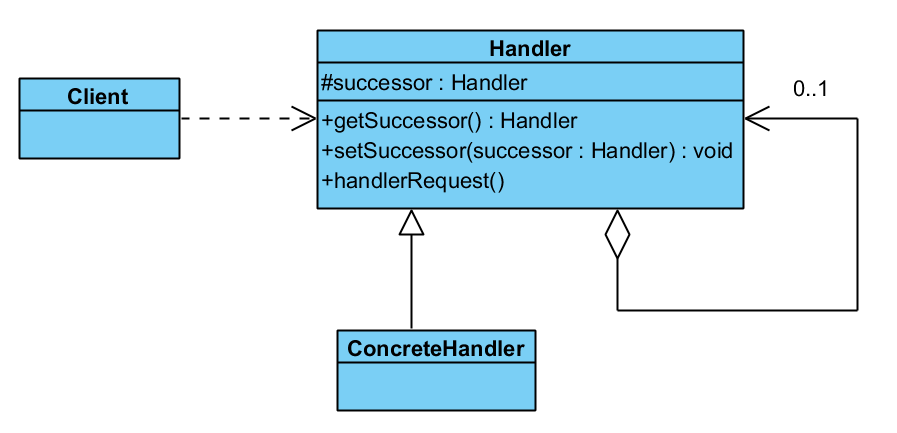
代理模式的内容不变，业务功能也不变。例如《Head Fiset设计模式》没带，在车上想看，可以在手机上的电子书看，既不会把书弄破（保护代理），又可以想看的时候才拿出来看，不需要一直把半斤重的数带在身上（虚拟代理）。

# 行为型

## 责任链模式（Chian-of-responsibility Pattern）

责任链模式是一种对象的行为模式。在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

结构：



责任链模式涉及到的角色如下所示：

**抽象处理者(Handler)角色**：定义出一个处理请求的接口。如果需要，接口可以定义 出一个方法以设定和返回对下家的引用。这个角色通常由一个Java抽象类或者Java接口实现。上图中Handler类的聚合关系给出了具体子类对下家的引用，抽象方法handleRequest()规范了子类处理请求的操作。

**具体处理者(ConcreteHandler)角色**：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家。由于具体处理者持有对下家的引用，因此，如果需要，具体处理者可以访问下家。

优点：

**请求者和接收者松散耦合**：在职责链模式里面，请求者并不知道接收者是谁，也不知道具体如何处理，请求者只是负责向职责链发出请求就可以了。而每个职责对象也不用管请求者或者是其它的职责对象，只负责处理自己的部分，其它的就交由其它的职责对象去处理。也就是说，请求者和接收者是完全解耦的。

**动态组合职责**：职责链模式会把功能处理分散到单独的职责对象里面，然后在使用的时候，可以动态组合职责形成职责链，从而可以灵活的给对象分配职责，也可以灵活的实现和改变对象的职责。

**简化了对象**：请求处理对象仅需维持一个指向其后继者的引用，而不需要维持它对所有的候选处理者的引用，可简化对象的相互连接。

**增加新的请求处理类很方便**：在系统中增加一个新的具体请求处理者时无须修改原有系统的代码，只需要在客户端重新建链即可，从这一点来看是符合“开闭原则”的。

缺点：

**产生很多细粒度对象**：职责链模式会把功能处理分散到单独的职责对象里面，也就是每个职责对象只是处理一个方面的功能，要把整个业务处理完，需要大量的职责对象的组合，这会产生大量的细粒度职责对象。

**不一定能被处理**：职责链模式的每个职责对象只负责自己处理的那一部分，因此可能会出现某个请求，把整个链传递完了，都没有职责对象处理它。这就需要在使用职责链模式的时候注意，需要提供默认的处理，并且注意构建的链的有效性。

**可能造成循环调用**：如果建链不当，可能会造成循环调用，将导致系统陷入死循环。

**调试不方便**：对于比较长的职责链，请求的处理可能涉及到多个处理对象，在进行代码调试时不太方便，有碍于除错。

应用场景：

1. 有多个对象可以处理同一个请求，但是具体由哪个对象来处理该请求，是运行时刻动态确定的。这种情况可以使用职责链模式，把处理请求的对象实现成为职责对象，然后把它们构成一个职责链，当请求在这个链中传递的时候，具体由哪个职责对象来处理，会在运行时动态判断。
2. 你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求的话，可以使用职责链模式，职责链模式实现了请求者和接收者之间的解耦，请求者不需要知道究竟是哪一个接收者对象来处理了请求。
3. 想要动态指定处理一个请求的对象集合，可以使用职责链模式，职责链模式能动态的构建职责链，也就是动态的来决定到底哪些职责对象来参与到处理请求中来，相当于是动态指定了处理一个请求的职责对象集合。

实际例子：

1. Tomcat的Filter
2. Netty的ChannelHandler
3. Java的异常捕获机制
4. JavaScript中事件的冒泡和捕获机制
5. Servlet开发中，过滤器的链式处理

### 两种形成链的方法

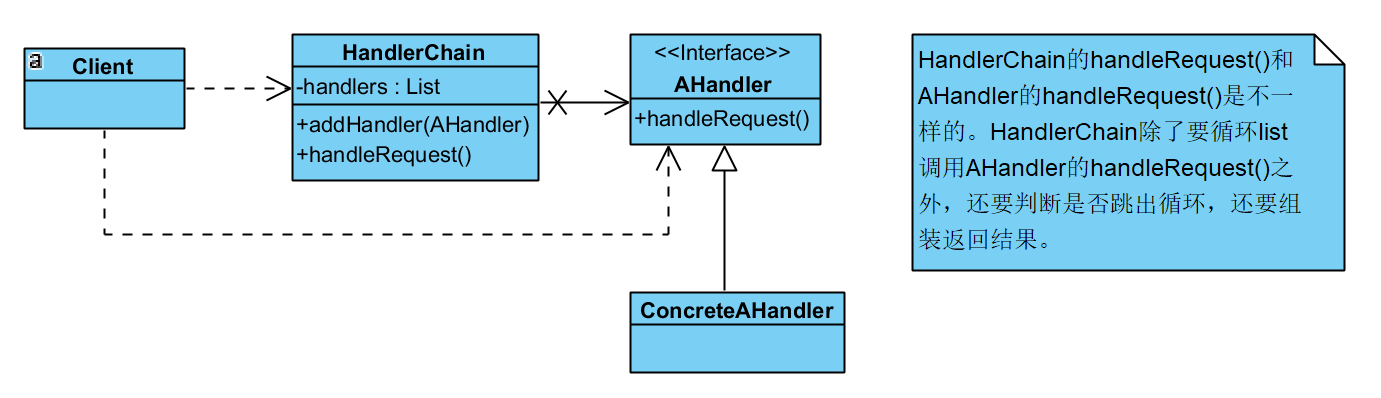
1. 通过设置Handler里面的成员变量successor，successor设置它自己的successor...从而形成链，类似于链表的Node和Node里面的next。通过这样设置可以形成指定执行顺序的责任链。

结构：同上面的类图

代码：designpattrn.behavior.chainofresponsibility.Main.chainOfResponsibility1()

1. 通过数组来设置，通过数组的索引来设置每个Handler的执行顺序。

结构：



Handler之间通过HandlerChain来关联，Handler之间不再有关联关系，无法在当前Handler获取下一个Handler的信息。

代码：designpattrn.behavior.chainofresponsibility.Main.chainOfResponsibility2()

### 纯的与不纯的责任链模式

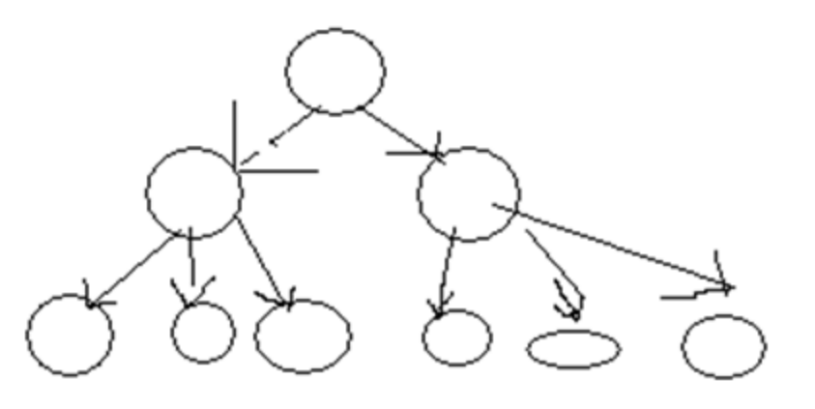
一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，二是把责任推给下家。不允许出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。

在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接收；在一个不纯的责任链模式里面，一个请求可以最终不被任何接收端对象所接收。

纯的责任链模式的实际例子很难找到，一般看到的例子均是不纯的责任链模式的实现。有些人认为不纯的责任链根本不是责任链模式，这也许是有道理的。但是在实际的系统里，纯的责任链很难找到。如果坚持责任链不纯便不是责任链模式，那么责任链模式便不会有太大意义了。

### 责任链的扩展——树状链结构

职责链中，我们之前看到的都是一些单链结构，但是其实在很多情况下，每一个节点都对应着很多其他的部分。



那么这样，我们的每一个节点都可以使用一个List来维护他节点的下一节点，甚至可以用组合模式来分别设计每一节点。

## 命令模式（Command Pattern）

命令模式属于对象的行为模式。命令模式又称为行动(Action)模式或交易(Transaction)模式。

命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

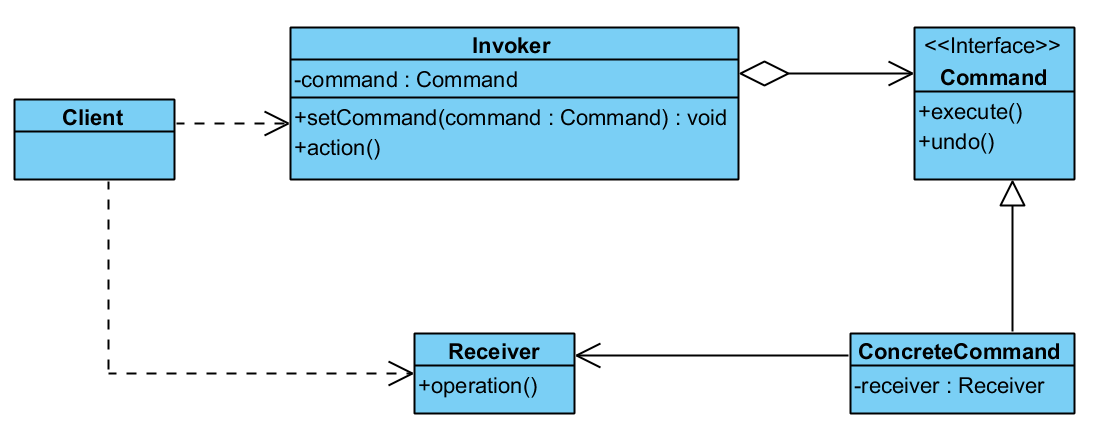
结构：

命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。

每一个命令都是一个操作：请求的一方发出请求要求执行一个操作；接收的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令允许请求的一方和接收请求的一方能够独立演化，从而具有以下的优点：

1. 命令模式使新的命令很容易地被加入到系统里。
2. 允许接收请求的一方决定是否要否决请求。
3. 能较容易地设计一个命令队列。
4. 可以容易地实现对请求的撤销和恢复。
5. 在需要的情况下，可以较容易地将命令记入日志。



命令模式涉及到五个角色，它们分别是：

**客户端(Client)角色**：创建一个具体命令(ConcreteCommand)对象并确定其接收者。

**命令(Command)角色**：声明了一个给所有具体命令类的抽象接口。

**具体命令(ConcreteCommand)角色**：定义一个接收者和行为之间的弱耦合；实现execute()方法，负责调用接收者的相应操作。execute()方法通常叫做执行方法。

**请求者(Invoker)角色**：负责调用命令对象执行请求，相关的方法叫做行动方法。

**接收者(Receiver)角色**：负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

代码：designpattrn.behavior.command.Main.main(String[])

命令模式的优点：

1. 更松散的耦合

命令模式使得发起命令的对象——客户端，和具体实现命令的对象——接收者对象完全解耦，也就是说发起命令的对象完全不知道具体实现对象是谁，也不知道如何实现。

1. 更动态的控制

命令模式把请求封装起来，可以动态地对它进行参数化、队列化和日志化等操作，从而使得系统更灵活。

1. 很自然的复合命令

命令模式中的命令对象能够很容易地组合成复合命令，也就是宏命令，从而使系统操作更简单，功能更强大。

1. 更好的扩展性

由于发起命令的对象和具体的实现完全解耦，因此扩展新的命令就很容易，只需要实现新的命令对象，然后在装配的时候，把具体的实现对象设置到命令对象中，然后就可以使用这个命令对象，已有的实现完全不用变化。

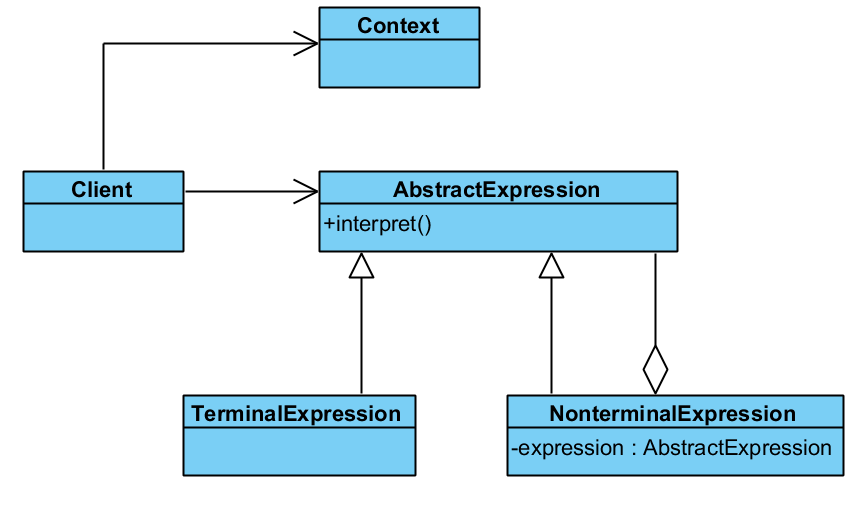
### 宏命令

所谓宏命令简单点说就是包含多个命令的命令，是一个命令的组合。一个命令通过list.add()方法添加子命令，再遍历递归调用子命令，而外部暴露只一个简单的接口，简单地调用批量命令，相当于外观模式和组合模式的结合。

## 解释器模式（Interpreter Pattern）

解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。

结构：



模式所涉及的角色如下所示：

1. **抽象表达式(Expression)角色**：声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个interpret()方法，称做解释操作。
2. **终结符表达式(Terminal Expression)角色**：实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式R=R1+R2，在里面R1和R2就是终结符，对应的解析R1和R2的解释器就是终结符表达式。
3. **非终结符表达式(Nonterminal Expression)角色**：文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式R=R1+R2中，“+"就是非终结符，解析“+”的解释器就是一个非终结符表达式。
4. **环境(Context)角色**：这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如R=R1+R2，我们给R1赋值100，给R2赋值200。这些信息需要存放到环境角色中，很多情况下我们使用Map来充当环境角色就足够了。

代码：designpattrn.behavior.interpreter.Main.main(String[])

优点：

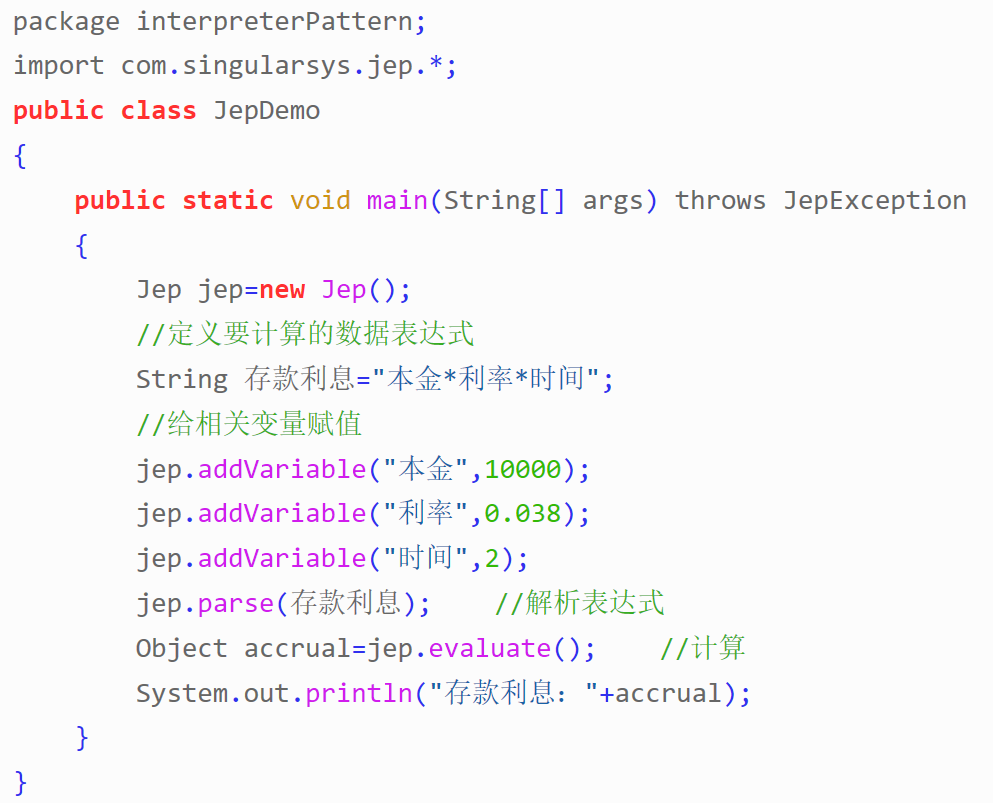
1. 可扩展性比较好，灵活。
2. 增加了新的解释表达式的方式。
3. 易于实现文法。

缺点：

1. 执行效率比较低，可利用场景比较少。
2. 对于复杂的文法比较难维护。

注意：解释器模式在实际的软件开发中使用比较少，因为它会引起效率、性能以及维护等问题。如果碰到对表达式的解释，在 [Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank) 中可以用 Expression4J 或 Jep 等来设计，也可以在项目中可以使用shell,JRuby,Groovy等脚本语言代替解释器模式。

代码：



运行结果：



应用例子：

1. 正则表达式

## 迭代模式（Iterator Pattern）

**迭代子模式又叫游标(Cursor)模式，是对象的行为模式。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象（internal representation）。**

### 聚集和Java聚集

多个对象聚在一起形成的总体称之为聚集(Aggregate)，聚集对象是能够包容一组对象的容器对象。聚集依赖于聚集结构的抽象化，具有复杂化和多样性。数组就是最基本的聚集，也是其他的JAVA聚集对象的设计基础。

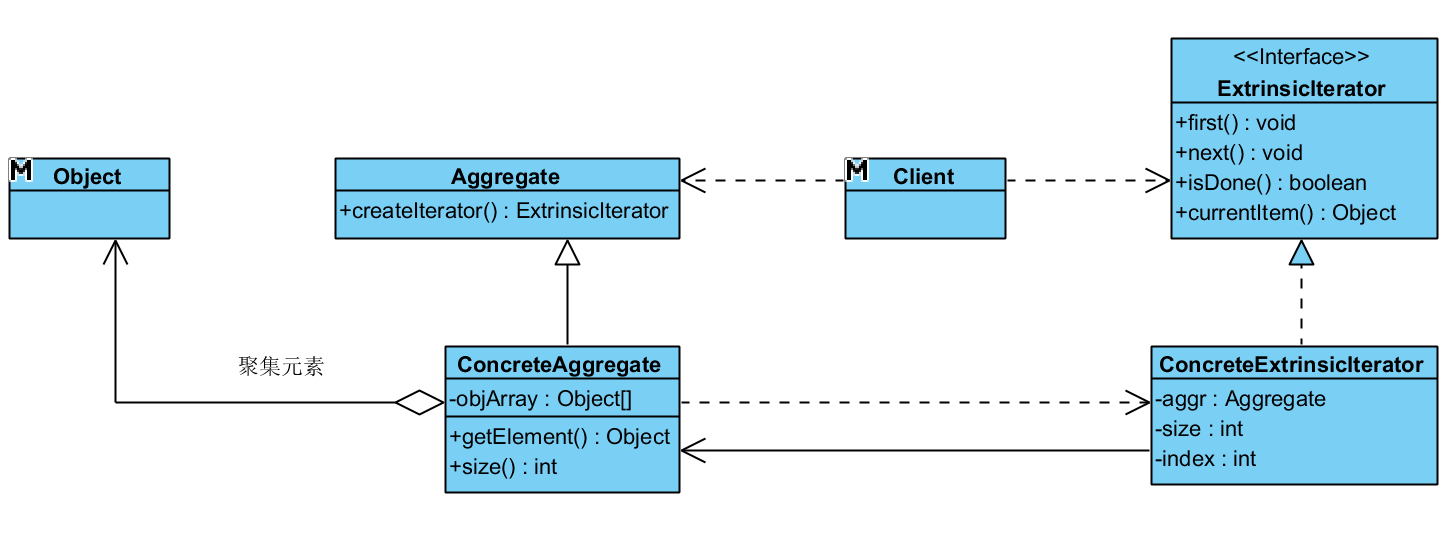
JAVA聚集对象是实现了共同的java.util.Collection接口的对象，是JAVA语言对聚集概念的直接支持。从1.2版开始，JAVA语言提供了很多种聚集，包括Vector、ArrayList、HashSet、HashMap、Hashtable等，这些都是JAVA聚集的例子。

### 白箱聚集与外禀迭代子

迭代子模式有两种实现方式，分别是**白箱聚集与外禀迭代子**和**黑箱聚集与内禀迭代子**。

如果一个聚集的接口提供了可以用来修改聚集元素的方法，这个接口就是所谓的**宽接口**。

如果聚集对象为所有对象提供同一个接口，也就是宽接口的话，当然会满足迭代子模式对迭代子对象的要求。但是，这样会破坏对聚集对象的封装。这种提供宽接口的聚集叫做**白箱聚集**。聚集对象向外界提供同样的宽接口，如下图所示：



由于聚集自己实现迭代逻辑，并向外部提供适当的接口，使得迭代子可以从外部控制聚集元素的迭代过程。这样一来迭代子所控制的仅仅是一个游标而已，这种迭代子叫做**游标迭代子（Cursor Iterator）**。由于迭代子是在聚集结构之外的，因此这样的迭代子又叫做**外禀迭代子（Extrinsic Iterator）**。

现在看一看白箱聚集与外禀迭代子的实现。一个白箱聚集向外界提供访问自己内部元素的接口（称作遍历方法或者Traversing Method），从而使外禀迭代子可以通过聚集的遍历方法实现迭代功能。

因为迭代的逻辑是由聚集对象本身提供的，所以这样的外禀迭代子角色往往仅仅保持迭代的游标位置。

迭代子模式涉及到以下几个角色：

**抽象迭代子(Iterator)角色**：此抽象角色定义出遍历元素所需的接口。

**具体迭代子(ConcreteIterator)角色**：此角色实现了Iterator接口，并保持迭代过程中的游标位置。

**聚集(Aggregate)角色**：此抽象角色给出创建迭代子(Iterator)对象的接口。

**具体聚集(ConcreteAggregate)角色**：实现了创建迭代子(Iterator)对象的接口，返回一个合适的具体迭代子实例。

**客户端(Client)角色**：持有对聚集及其迭代子对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

代码：designpattrn.behavior.iterator.Main.extrinsicIterator()

理解：

白箱指的是ConcreteAggregate有getElement(index)方法，对外暴露了获取元素的接口，使得本身不一定要通过迭代器来遍历，也可以通过不断调用getElement(index)来实现遍历。

黑箱指的是ConcreteAggregate没有getElement(index)方法，要获取聚集的元素，只能通过迭代器来获取。

外禀指的是迭代器不是作为聚集的内部类存在，这使得任何时候都可以new一个迭代器来对这个聚集或其他聚集进行迭代。

內禀指的是迭代器作为聚集的私有内部类，这使得只有聚集元素才能创建这个迭代器，无法从外界创建这个迭代器。

**外禀迭代子的意义**

一个常常会问的问题是：既然白箱聚集已经向外界提供了遍历方法，客户端已经可以自行进行迭代了，为什么还要应用迭代子模式，并创建一个迭代子对象进行迭代呢？

客户端当然可以自行进行迭代，不一定非得需要一个迭代子对象。但是，迭代子对象和迭代模式会将迭代过程抽象化，将作为迭代消费者的客户端与迭代负责人的迭代子责任分隔开，使得两者可以独立的演化。在聚集对象的种类发生变化，或者迭代的方法发生改变时，迭代子作为一个中介层可以吸收变化的因素，而避免修改客户端或者聚集本身。

此外，如果系统需要同时针对几个不同的聚集对象进行迭代，而这些聚集对象所提供的遍历方法有所不同时，使用迭代子模式和一个外界的迭代子对象是有意义的。具有同一迭代接口的不同迭代子对象处理具有不同遍历接口的聚集对象，使得系统可以使用一个统一的迭代接口进行所有的迭代。

### 黑箱聚集与內禀迭代子

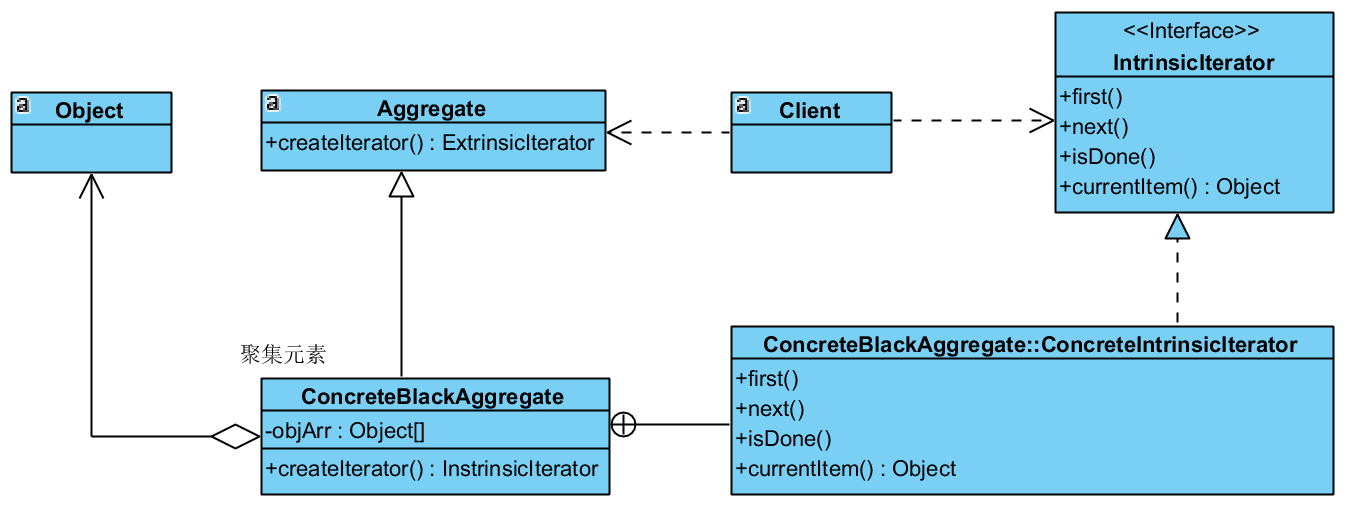
如果一个聚集的接口没有提供修改聚集元素的方法，这样的接口就是所谓的**窄接口**。

聚集对象为迭代子对象提供一个宽接口，而为其他对象提供一个窄接口。换言之，聚集对象的内部结构应当对迭代子对象适当公开，以便迭代子对象能够对聚集对象有足够的了解，从而可以进行迭代操作。但是，聚集对象应当避免向其他的对象提供这些方法，因为其他对象应当经过迭代子对象进行这些工作，而不是直接操控聚集对象。

　在JAVA语言中，实现双重接口的办法就是将迭代子类设计成聚集类的内部成员类。这样迭代子对象将可以像聚集对象的内部成员一样访问聚集对象的内部结构。下面给出一个示意性的实现，说明这种双重接口的结构时怎么样产生的，以及使用了双重接口结构之后迭代子模式的实现方案。这种同时保证聚集对象的封装和迭代子功能的实现的方案叫做**黑箱实现方案**。

由于迭代子是聚集的内部类，迭代子可以自由访问聚集的元素，所以迭代子可以自行实现迭代功能并控制对聚集元素的迭代逻辑。由于迭代子是在聚集的结构之内定义的，因此这样的迭代子又叫做**内禀迭代子（Intrinsic Iterator）**。

结构：



代码：designpattrn.behavior.iterator.Main.intrinsicIterator()

### 主动迭代子与被动迭代子

主动迭代子和被动迭代子又称作外部迭代子和内部迭代子。

所谓主动（外部）迭代子，指的是由客户端来控制迭代下一个元素的步骤，客户端会明显调用迭代子的next()等迭代方法，在遍历过程中向前进行。

所谓被动（内部）迭代子，指的是由迭代子自己来控制迭代下一个元素的步骤。因此，如果想要在迭代的过程中完成工作的话，客户端就需要把操作传递给迭代子，迭代子在迭代的时候会在每个元素上执行这个操作，类似于JAVA的回调机制。

总体来说外部迭代器比内部迭代器要灵活一些，因此我们常见的实现多属于主动迭代子。

### 静态迭代子与动态迭代子

静态迭代子由聚集对象创建，并持有聚集对象的一份快照(snapshot)，在产生后这个快照的内容就不再变化。客户端可以继续修改原聚集的内容，但是迭代子对象不会反映出聚集的新变化。

静态迭代子的好处是它的安全性和简易性，换言之，静态迭代子易于实现，不容易出现错误。但是由于静态迭代子将原聚集复制了一份，因此它的短处是对时间和内存资源的消耗。

动态迭代子则与静态迭代子完全相反，在迭代子被产生之后，迭代子保持着对聚集元素的引用，因此，任何对原聚集内容的修改都会在迭代子对象上反映出来。

完整的动态迭代子不容易实现，但是简化的动态迭代子并不难实现。大多数JAVA设计师遇到的迭代子都是这种简化的动态迭代子。为了说明什么是简化的动态迭代子，首先需要介绍一个新的概念：Fail Fast。

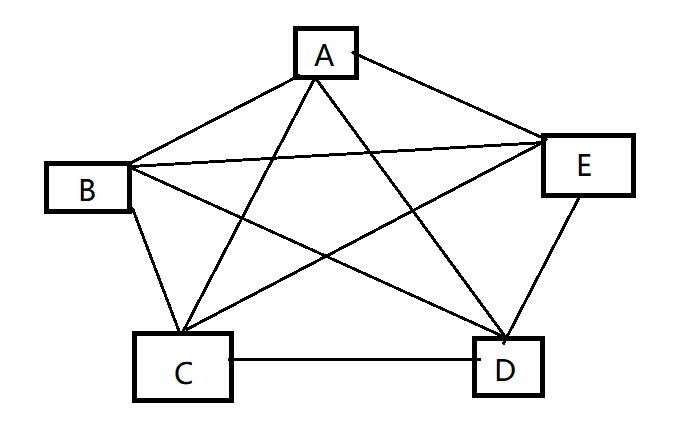
在Java中，如果遍历List在迭代开始后，聚集的内容被外界绕过迭代子对象而直接修改的话，这个方法会立即抛出ConcurrentModificationException()异常。这就是说，AbstractList.Itr迭代子是一个Fail Fast的迭代子。如果需要边增删元素，边遍历元素的话可以使用CopyOnWriterArrayList。

## 中介者模式（Mediator Pattern）

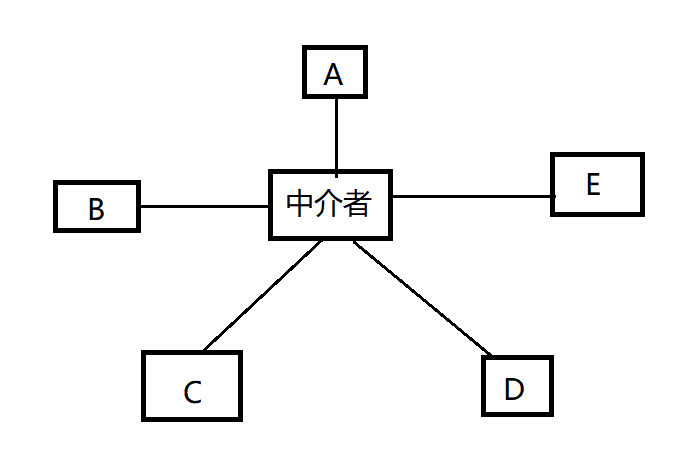
**定义**：用一个中介者对象封装一系列的对象交互，中介者使各对象不需要显示地相互作用，从而使耦合松散，而且可以独立地改变它们之间的交互。

中介者模式的作用是把各个类之间的关系统一放在中介者类中管理，把原本类之间的复杂的关联关系改为类与中介者类的关联，从而把类中对应多个类的关联关系改为类中对应一个中介者类的对应关系。

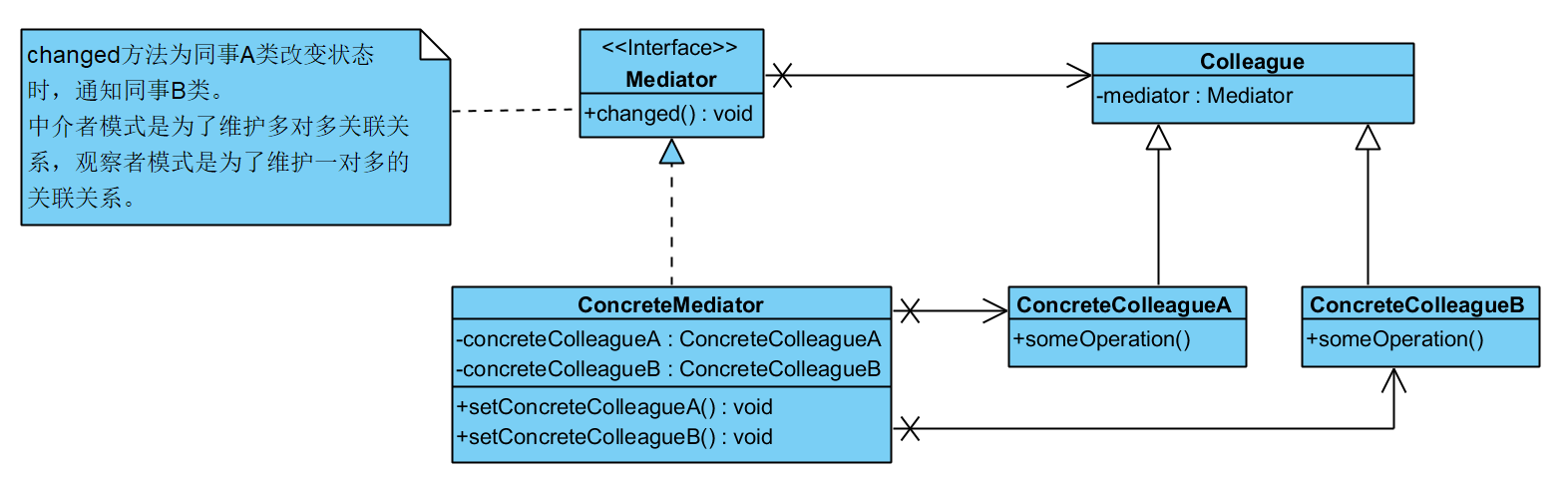
不使用中介者模式：每个类都有其他多个类的成员，修改关联关系时可能需要在每个类都修改代码，从而维护关联关系。



使用中介者模式：每个类都只有一个成员变量——中介者类，修改关联关系时只需要修改中介者类的代码，而且成员变量从多个变成一个，减少实现功能所必须知道的知识（变量），符合迪米特法则。



结构：



中介者模式涉及的角色及其职责如下：

1. **抽象中介者(Mediator)角色**：一般定义为接口，用来定义各个同事之间交互需要的方法，可以是公共的通信方法，比如 changed()方法，大家都用，也可以是小范围的交互方法。
2. **具体中介者(ConcreteMediator)角色**：实现Mediator中定义的接口，它需要了解并维护各个同事对象，并负责具体的协调各同事对象的交互关系。
3. **抽象同事(Colleague)角色**：通常实现成为抽象类，主要负责约束同事对象的类型，并实现一些具体同事类之间的公共功能，比如，每个具体同事类都应该知道中介者对象，也就是具体同事类都会持有中介者对象，都可以定义到这个类里面。
4. **具体同事(ConcreteColleague)角色**：继承自Colleague，实现自己的业务，在需要与其他同事通信的时候，就与持有的中介者通信，中介者负责与他的同事交互。

中介者模式的核心在于中介者类的引入，在中介者模式中，中介者类承担了两方面的职责：

**中转作用（结构性）**：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，可通过中介者来实现间接调用。该中转作用属于中介者在结构上的支持。

**协调作用（行为性）**：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致的和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。

代码：designpattrn.behavior.mediator.Main.mediator()

优点：

1. 中介者模式通过把多个同事对象之间的交互封装到中介者对象里面，从而使得同事对象之间松散耦合，基本上可以做到互补依赖。这样一来，同事对象就可以独立地变化和复用，而不再像以前那样“牵一处而动全身”了。
2. 集中控制交互　多个同事对象的交互，被封装在中介者对象里面集中管理，使得这些交互行为发生变化的时候，只需要修改中介者对象就可以了，当然如果是已经做好的系统，那么就扩展中介者对象，而各个同事类不需要做修改。
3. 多对多变成一对多没有使用中介者模式的时候，同事对象之间的关系通常是多对多的，引入中介者对象以后，中介者对象和同事对象的关系通常变成双向的一对多，这会让对象的关系更容易理解和实现。

缺点：

中介者模式的一个潜在缺点是，过度集中化。如果同事对象的交互非常多，而且比较复杂，当这些复杂性全部集中到中介者的时候，会导致中介者对象变得十分复杂，而且难于管理和维护。

应用场景：

1. 如果一组对象之间的通信方式比较复杂，导致相互依赖、结构混乱，可以采用中介者模式，把这些对象相互的交互管理起来，各个对象都只需要和中介者交互，从而使得各个对象松散耦合，结构也更清晰易懂。
2. 如果一个对象引用很多的对象，并直接跟这些对象交互，导致难以复用该对象，可以采用中介者模式，把这个对象跟其他对象的交互封装到中介者对象里面，这个对象只需要和中介者对象交互就可以了。

实际例子：

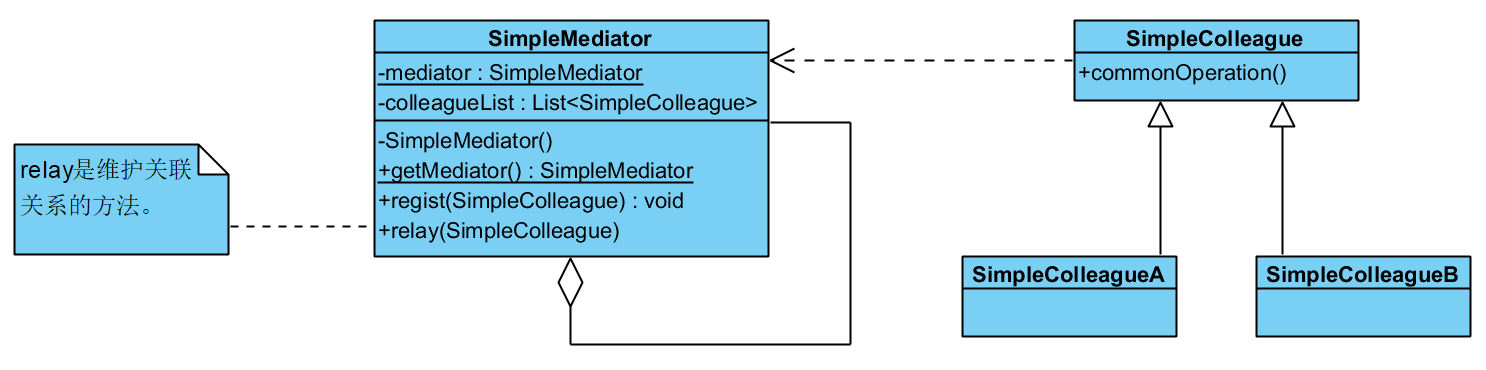
1. Scrapy的引擎，负责维护爬虫、调度器、下载器、管道模块之间的关联关系。
2. Spring MVC的DispatcherServlet，负责维护处理器映射器，处理器适配器，视图解析器之间的关联关系。
3. Java的java.util.Timer，定时调度我们写的各种任务，将任务添加到 TaskQueue 任务队列中，给 TimerThread 执行，让任务与执行线程解耦。
4. MVC模式中的Controller，负责维护页面与后台业务之间的关联关系。

### 中介者模式的简化

在实际应用开发中，经常会使用简化版的中介者模式进行开发，被称为广义中介者，常有如下简化：

1. 通常会去掉同事对象的父类，这样可以让任意的对象，只要需要相互交互，就可以成为同事。
2. 通常不定义Mediator接口，把具体的中介者对象实现成为单例。
3. 同事对象不再持有中介者，而是在需要的时候直接获取中介者对象并调用；中介者也不再持有同事对象，而是在具体处理方法里面去创建，或者获取，或者从参数传入需要的同事对象。

结构：



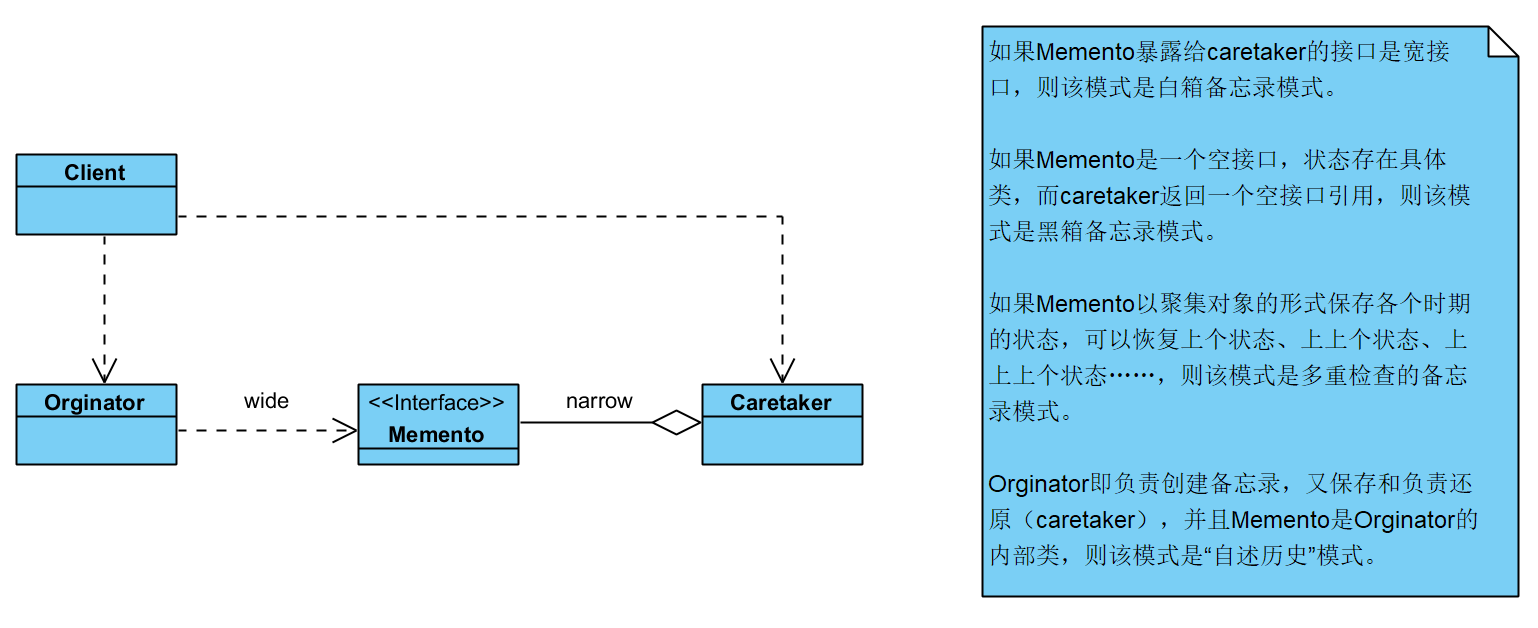
代码：designpattrn.behavior.mediator.Main.simpleMediator()

## 备忘录模式（Memento Pattern）

备忘录模式又叫做快照模式(Snapshot Pattern)或Token模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉(Capture)住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式常常与命令模式和迭代子模式一同使用。

总体结构：



备忘录模式所涉及的角色有三个：**备忘录(Memento)角色**、**发起人(Originator)角色**、**负责人(Caretaker)角色**。

**备忘录(Memento)角色**

备忘录角色又如下责任：

1. 将发起人（Originator）对象的内战状态存储起来。备忘录可以根据发起人对象的判断来决定存储多少发起人（Originator）对象的内部状态。
2. 备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。

备忘录有两个等效的接口：

（1）窄接口：负责人（Caretaker）对象（和其他除发起人对象之外的任何对象）看到的是备忘录的窄接口(narrow interface)，这个窄接口只允许它把备忘录对象传给其他的对象。

（2）宽接口：与负责人对象看到的窄接口相反的是，发起人对象可以看到一个宽接口(wide interface)，这个宽接口允许它读取所有的数据，以便根据这些数据恢复这个发起人对象的内部状态。

**发起人（Originator）角色**

发起人角色有如下责任：

1. 创建一个含有当前的内部状态的备忘录对象。
2. 使用备忘录对象存储其内部状态。

**负责人（Caretaker）角色**

负责人角色有如下责任：

1. 负责保存备忘录对象。
2. 不检查、修改备忘录对象的内容。

优点：

1. **提供状态恢复。**对象恢复到特定的历史步骤。
2. **封装信息。**备忘录保存了原发器（发起人，Originator）的状态，并且只有原发器可以改动。

缺点：

1. **资源消耗大。**特别是保存的原发器类的对象较多时，需要占据内存，保存操作也需要开销。

适用范围：

1. 需要实现撤销操作。
2. 需要封装对象的历史状态，并且避免暴露给其他对象。

实际应用：

1. Jdk中java.util.Date类中有private transient long fastTime;保存历史时间

### 白箱的备忘录模式

备忘录角色对任何对象都提供一个接口，即宽接口，备忘录角色的内部所存储的状态就对所有对象公开。因此这个实现又叫做“白箱实现”。

“白箱”实现将发起人角色的状态存储在一个大家都看得到的地方，因此是破坏封装性的。但是通过程序员自律，同样可以在一定程度上实现模式的大部分用意。因此白箱实现仍然是有意义的。

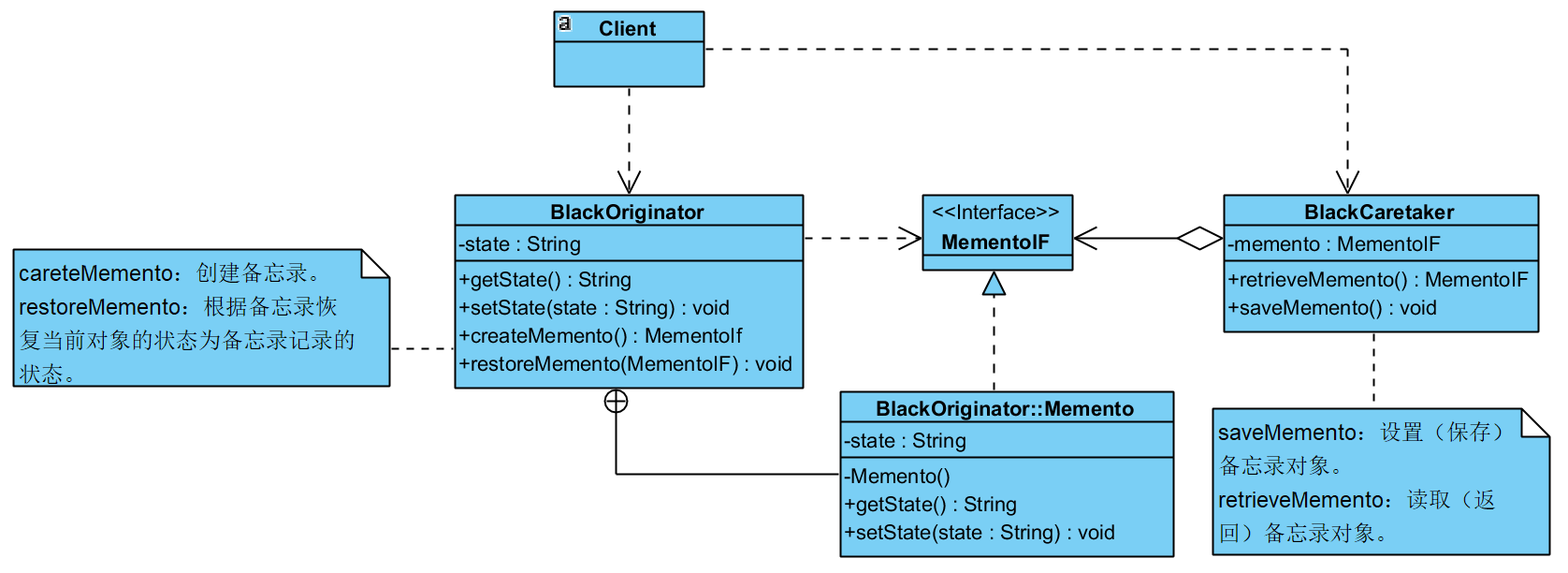
### 黑箱的备忘录模式

备忘录角色对发起人（Originator）角色对象提供一个宽接口，而为其他对象提供一个窄接口。这样的实现叫做“黑箱实现”。

在JAVA语言中，实现双重接口的办法就是将备忘录角色类设计成发起人角色类的内部成员类。

将Memento设成Originator类的内部类，从而将Memento对象封装在Originator里面；在外部提供一个标识接口MementoIF给Caretaker以及其他对象。这样，Originator类看到的是Menmento的所有接口，而Caretaker以及其他对象看到的仅仅是标识接口MementoIF所暴露出来的接口。

结构：



MementoIF是一个空接口，即使从BlackCaretaker获取到备忘录对象也无法直接读取备忘录的状态，需要强转成BlackOriginator.Memento类（接口）才能通过getter获取其状态，但是因为是私有内部类，所以只有在BlackOriginator里面才能强转。例如BlackOrginator的restoreMemento就把参数的MementoIF强转成BlackOriginator.Memento来恢复状态。

代码：designpattrn.behavior.memento.Main.blackMemento()

### 多重检查点

如果我今天发现系统异常，想要将系统数据恢复到昨天甚至前天或上个星期五的状态，这个时候我们就需要有多个备份数据提供选择。可以选择恢复多重备份的点叫做检查点（check point）, 检查点就是一个标记，例如它可以是一个系统的时间戳等，每个检查点会对应一个备份数据以备后期选择性恢复。

Caretaker通过list或者map等聚集对象存储多个备忘录，获取的时候再通过索引或者key来获取指定的备忘录，例如：

备份：

caretaker.saveMemento("2018-06-21", originator.createMemento());

caretaker.saveMemento("2018-06-22", originator.createMemento());

caretaker.saveMemento("2018-06-23", originator.createMemento());

caretaker.saveMemento("2018-06-24", originator.createMemento());

还原：

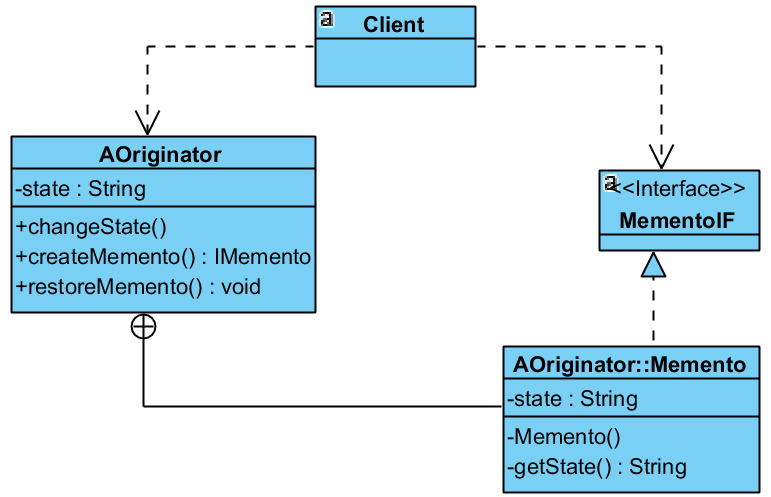
originator.restoreMemento(caretaker.getMemento("2018-06-22"));

### “自述历史”模式

所谓“自述历史”模式(History-On-Self Pattern)实际上就是备忘录模式的一个变种。在备忘录模式中，发起人(Originator)角色、负责人(Caretaker)角色和备忘录(Memento)角色都是独立的角色。虽然在实现上备忘录类可以成为发起人类的内部成员类，但是备忘录类仍然保持作为一个角色的独立意义。在“自述历史”模式里面，发起人角色自己兼任负责人角色。

由于“自述历史”作为一个备忘录模式的特殊实现形式非常简单易懂，它可能是备忘录模式最为流行的实现形式。

结构：



**备忘录角色有如下责任：**

1. 将发起人（Originator）对象的内部状态存储起来。
2. 备忘录可以保护其内容不被发起人（Originator）对象之外的任何对象所读取。

**发起人角色有如下责任：**

1. 创建一个含有它当前的内部状态的备忘录对象。
2. 使用备忘录对象存储其内部状态。

客户端角色有负责保存备忘录对象的责任。

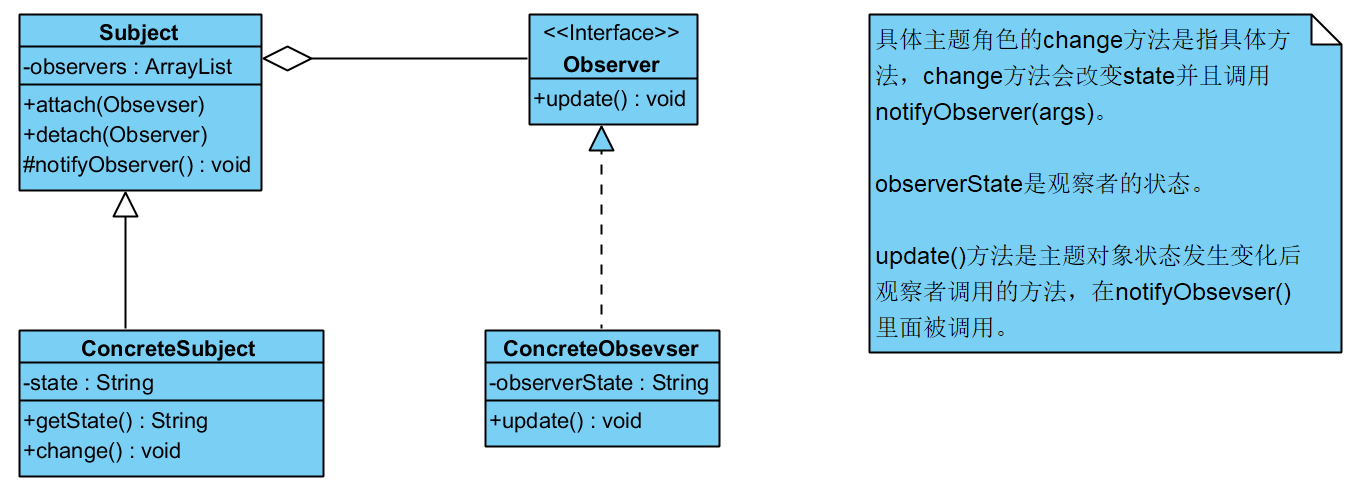
代码：designpattrn.behavior.memento.Main.blackMemento()

## 观察者模式（Observer Pattern）

**观察者模式是对象的行为模式，又叫发布-订阅(Publish/Subscribe)模式、模型-视图(Model/View)模式、源-监听器(Source/Listener)模式或从属者(Dependents)模式。**

**观察者模式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上发生变化时，会通知所有观察者对象，使它们能够自动更新自己。**

结构：



观察者模式所涉及的角色有：

**抽象主题(Subject)角色**：抽象主题角色把所有对观察者对象的引用保存在一个聚集（比如ArrayList对象）里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，抽象主题角色又叫做抽象被观察者(Observable)角色。

**具体主题(ConcreteSubject)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改变时，给所有登记过的观察者发出通知。具体主题角色又叫做具体被观察者(Concrete Observable)角色。

**抽象观察者(Observer)角色**：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己，这个接口叫做更新接口。

**具体观察者(ConcreteObserver)角色**：存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态 像协调。如果需要，具体观察者角色可以保持一个指向具体主题对象的引用。

代码：designpattrn.behavior.observer.Main.observer()

实际应用：

1. 界面元素状态的变化
2. 消息队列

### 推模型和拉模型

在观察者模式中，又分为推模型和拉模型两种方式。

**推模型**

主题对象向观察者推送主题的详细信息，不管观察者是否需要，推送的信息通常是主题对象的全部或部分数据。

**拉模型**

主题对象在通知观察者的时候，只传递少量信息。如果观察者需要更具体的信息，由观察者主动到主题对象中获取，相当于是观察者从主题对象中拉数据。一般这种模型的实现中，会把主题对象自身通过update()方法传递给观察者，这样在观察者需要获取数据的时候，就可以通过这个引用来获取了。

代码上的区别：

推模型的notifyObverser需要传参数args，并把参数args通过观察者的update()方法传到观察者；拉模型的notifyObverser不需要传参数，把自身通过观察者的update()方法传递给观察者，由观察者获取主题对象的成员变量。

### Java提供对观察者模式的支持

被观察者继承java.util.Observable类，setChange()方法表示标记状态改变，notifyObservers()方法调用观察者，调用完后会调用clearChange()方法重置标记。

观察者需要实现java.util.Observer接口，实现update(Observable o , Object arg)方法。

代码：designpattrn.behavior.observer.Main.jdkObserver()

### 观察者模式和桥接模式的异同

相同：

1. 都是抽象类聚合接口。

不同：

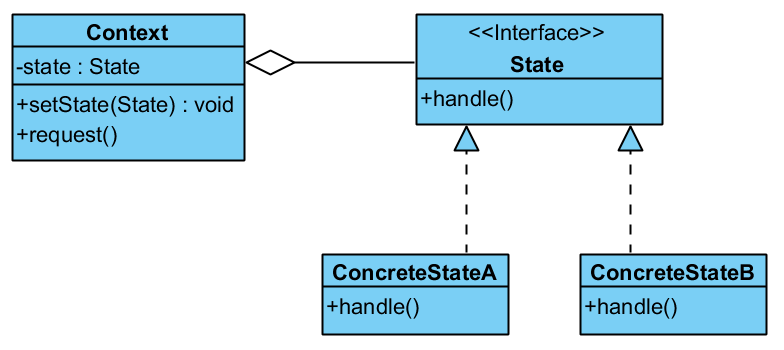
1. 观察者模式是行为型模式，桥接模式是构造型模式。观察者模式有固定的方法，例如改变状态的方法，notifyObservers()方法；桥接模式只是规定类的结构，并没有规定类的行为。
2. 使用场景不同。观察者模式的目的在于监听，语言上可以用“一……就……”来表示；桥接模式的目的在于解耦。

## 状态模式（State Pattern）

**状态模式，又称状态对象模式（Pattern of Objects for States），状态模式是对象的行为模式。**

**状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。**

结构：



状态模式所涉及到的角色有：

**环境(Context)角色**，也成上下文：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。

**抽象状态(State)角色**：定义一个接口，用以封装环境（Context）对象的一个特定的状态所对应的行为。

**具体状态(ConcreteState)角色**：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。

代码：designpattrn.behavior.state.Main.main(String[])

要点：

1. 客户不会和状态进行交互,全盘了解状态是 context的工作。Context内部自己实现状态转移，也不需要客户端设置状态。
2. 使用状态模式总是会增加设计中类的数目，这是为了要获得程序可扩展性，弹性的代价，如果你的代码不是一次性的，后期可能会不断加入不同的状态，那么状态模式的设计是绝对值得的。

优点：

1. Context类封装了状态转移的规则。
2. 枚举状态的可能。即在当前状态下，可以知道下一个状态是转移到有哪几种状态。
3. 封装了相同条件的if...else...,避免了大量的if...else...，从每个方法判断状态，变成判断一次状态，每个状态这一系列的方法。新增一种状态就修改一次代码（在原本的if...else...的代码加一个else if），其它都是新增代码，部分符合开闭原则（还是有修改代码，也算缺点）。
4. 多个环境（Context）可以共享一个状态对象，从而减少系统中对象的个数。
5. Context的状态没有暴露给客户端，符合迪米特法则。

缺点：

1. 新增大量类。
2. 并不是每个状态逻辑上都有这一系列方法，所以每个状态都有很多空方法或者直接抛异常的方法。

应用场景：

1. java.util.Iterator
2. javax.faces.lifecycle.LifeCycle#execute()

### 状态和行为

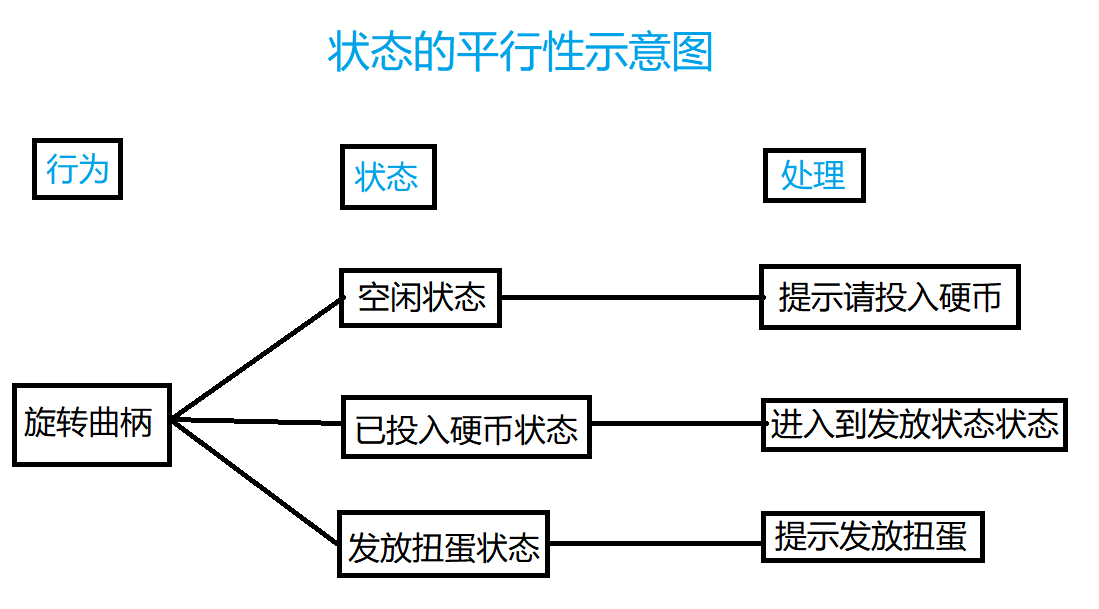
所谓对象的状态，通常指的就是对象实例的属性的值；而行为指的就是对象的功能，再具体点说，行为大多可以对应到方法上。

状态模式的功能就是分离状态的行为，通过维护状态的变化，来调用不同状态对应的不同功能。也就是说，状态和行为是相关联的，它们的关系可以描述为：**状态决定行为**。

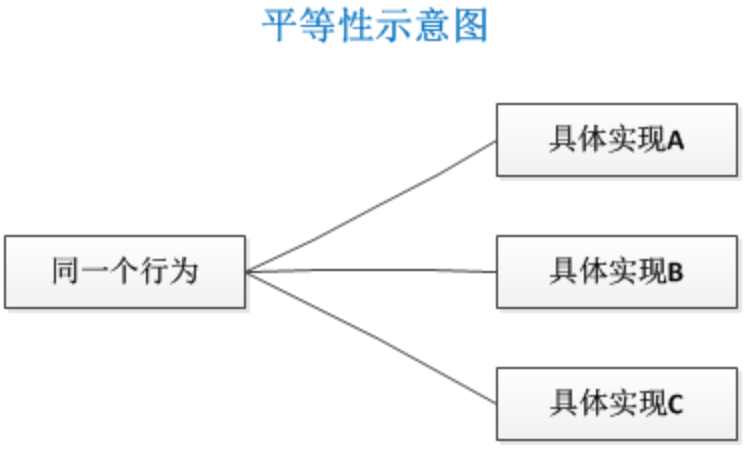
由于状态是在运行期被改变的，因此行为也会在运行期根据状态的改变而改变。

### 行为的平行性和平等性

所谓平行性指的是各个状态的行为所处的层次是一样的，相互独立的、没有关联的，是根据不同的状态来决定到底走平行线的哪一条。行为是不同的，当然对应的实现也是不同的，相互之间是不可替换的。例如扭蛋机的在不同状态（空闲、已投入硬币，发放）下的旋转曲柄行为的处理是不同的，空闲状态下旋转曲柄是提示投入硬币，已投入硬币状态下旋转曲柄是跳转到发放扭蛋状态，发放扭蛋状态下旋转曲柄是提示请等待发放扭蛋。



平等性强调的是可替换性，大家是同一行为的不同描述或实现，因此在同一个行为发生的时候，可以根据条件挑选任意一个实现来进行相应的处理。例如扭蛋机的运行状态的空间状态、已投入硬币状态和发放扭蛋状态可以互相替换。



大家可能会发现状态模式的结构和策略模式的结构完全一样，但是，它们的目的、实现、本质却是完全不一样的。还有行为之间的特性也是状态模式和策略模式一个很重要的区别，状态模式的行为是平行性的，不可相互替换的；而策略模式的行为是平等性的，是可以相互替换的。

### 环境和状态处理对象

在状态模式中，环境(Context)是持有状态的对象，但是环境(Context)自身并不处理跟状态相关的行为，而是把处理状态的功能委托给了状态对应的状态处理类来处理。

在具体的状态处理类中经常需要获取环境(Context)自身的数据，甚至在必要的时候会回调环境(Context)的方法，因此，通常将环境(Context)自身当作一个参数传递给具体的状态处理类。

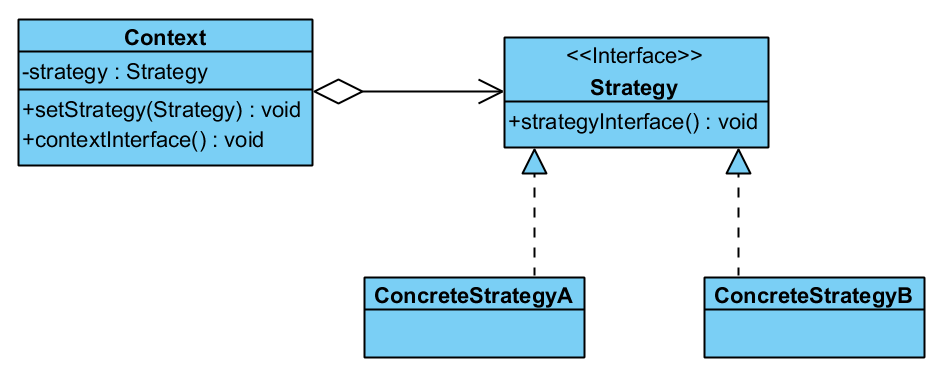
客户端一般只和环境(Context)交互。客户端可以用状态对象来配置一个环境(Context)，一旦配置完毕，就不再需要和状态对象打交道了。客户端通常不负责运行期间状态的维护，也不负责决定后续到底使用哪一个具体的状态处理对象。

## 策略模式（Strategy Pattern）

**策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。**

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。用一句话来说，就是：“准备一组算法，并将每一个算法封装起来，使得它们可以互换”。

结构：



这个模式涉及到三个角色：

**环境(Context)角色**：持有一个Strategy的引用。

**抽象策略(Strategy)角色**：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

**具体策略(ConcreteStrategy)角色**：包装了相关的算法或行为。

代码：designpattrn.behavior.strategy.Main.main(String[])

**策略模式的优点：**  
 1. 算法可以自由切换；  
 2. 避免使用多重条件判断；  
 3. 扩展性良好。

**策略模式的缺点：** 1. 策略类会增多  
 2. 所有策略类都需要对外暴露

**解决策略模式的缺陷：**

 1. 增加一个映射配置文件，实现策略类的隐藏。使用枚举对策略类进行映射处理，避免高层模块直接访问策略类，同时由工厂方法模式根据映射产生策略对象。

**策略模式的适用场景：** 1. 当一个系统中有许多类，它们之间的区别仅在于它们的行为，希望动态地让一个对象在许多行为中选择一种行为时；  
 2. 当一个系统需要动态地在几种算法中选择一种时；  
 3. 当一个对象有很多的行为，不想使用多重的条件选择语句来选择使用哪个行为时。

**注意事项：**

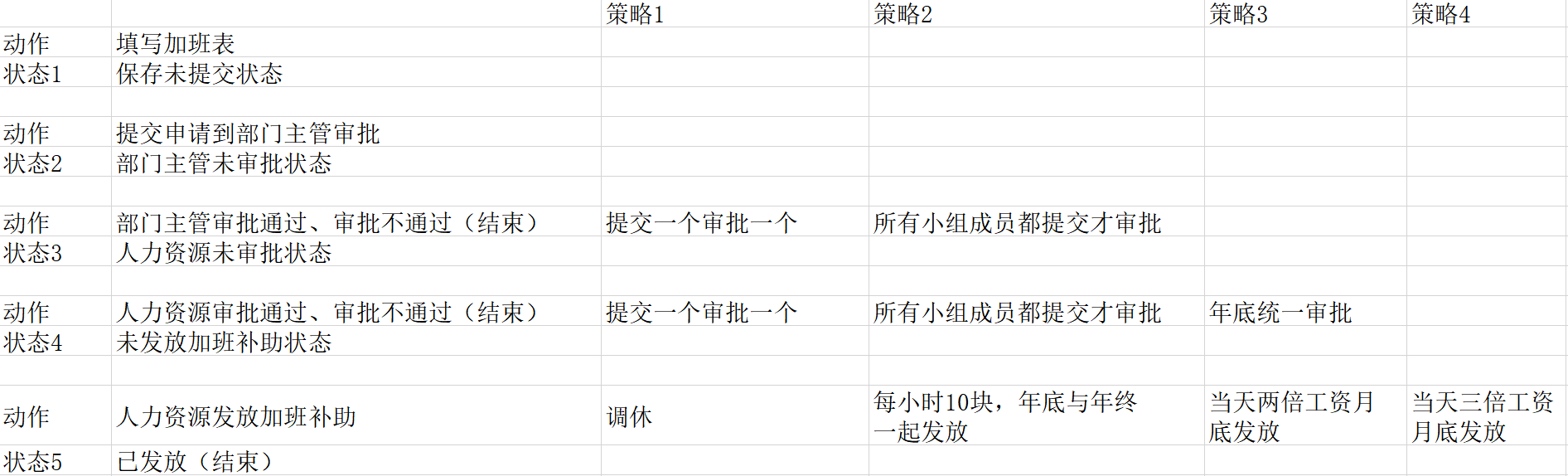
1. 一个策略家族的具体策略数量超过4个就需要考虑使用混合模式避免类膨胀。

### 算法的平等性

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正因为这个平等性，才能实现算法之间可以相互替换。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

### 策略模式和状态模式的区别

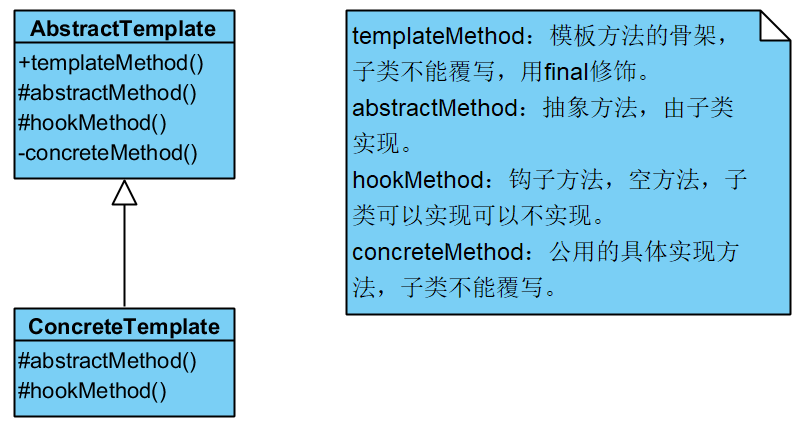


1. 策略模式封装了一组相关算法，它允许Client在运行时使用可互换的行为；状态模式帮助一个类在不同的状态显示不同的行为。
2. 状态模式封装了对象的状态，而策略模式封装算法或策略。因为状态是跟对象密切相关的，它不能被重用；而通过从Context中分离出策略或算法，我们可以重用它们。
3. 在状态模式中，每个状态通过持有Context的引用，来实现状态转移；但是每个策略都不持有Context的引用，它们只是被Context使用。
4. 策略实现可以作为参数传递给使用它的对象，例如Collections.sort()，它的参数包含一个Comparator策略。另一方面，状态是Context对象自己的一部分，随着时间的推移，Context对象从一个状态转移到另一个状态。
5. 虽然它们都符合OCP原则，策略模式也符合SRP原则（单一职责原则），因为每个策略都封装自己的算法，且不依赖其他策略。一个策略的改变，并不会导致其他策略的变化。
6. 另一个理论上的不同：策略模式定义了对象“怎么做”的部分。例如，排序对象怎么对数据排序。状态模式定义了对象“是什么”和“什么时候做”的部分。例如，对象处于什么状态，什么时候处在某个特定的状态。
7. 状态模式中很好的定义了状态转移的次序；而策略模式并无此需要：Client可以自由的选择任何策略。
8. 一些常见的策略模式的例子是封装算法，例如排序算法，加密算法或者压缩算法。如果你看到你的代码需要使用不同类型的相关算法，那么考虑使用策略模式吧。而识别何时使用状态模式是很简单的：如果你需要管理状态和状态转移，但不想使用大量嵌套的条件语句，那么就是它了。
9. 状态模式是内部状态决定行为、策略模式是外部client自有选择行为。

## 模板方法模式（Template Method Pattern）

**模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意。**

结构：



**抽象模板(Abstract Template)角色有如下责任：**

1. 定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。
2. 定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

**具体模板(Concrete Template)角色有如下责任：**

1. 实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。
2. 每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

代码：designpattrn.behavior.template.Main.main(String[])

模板模式的关键是：**子类可以置换掉父类的可变部分，但是子类却不可以改变模板方法所代表的顶级逻辑。**

每当定义一个新的子类时，不要按照控制流程的思路去想，而应当按照“责任”的思路去想。换言之，应当考虑哪些操作是必须置换掉的，哪些操作是可以置换掉的，以及哪些操作是不可以置换掉的。使用模板模式可以使这些责任变得清晰。

### 模板方法模式中的方法

模板方法中的方法可以分为两大类：模板方法和基本方法。

**模板方法**

一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。

一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。

**基本方法**

基本方法又可以分为三种：抽象方法(Abstract Method)、具体方法(Concrete Method)和钩子方法(Hook Method)。

1. 抽象方法：一个抽象方法由抽象类声明，由具体子类实现。在Java语言里抽象方法以abstract关键字标示。
2. 具体方法：一个具体方法由抽象类声明并实现，而子类并不实现或置换。
3. 钩子方法：一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。

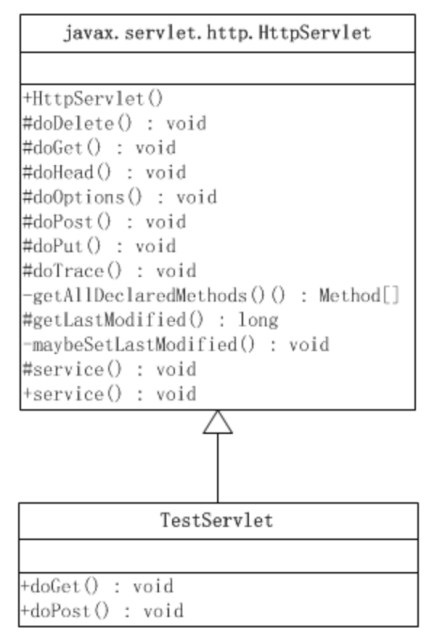
### 默认钩子方法

这种空的钩子方法叫做“Do Nothing Hook”。显然，这种默认钩子方法在缺省适配模式里面已经见过了，一个缺省适配模式讲的是一个类为一个接口提供一个默认的空实现，从而使得缺省适配类的子类不必像实现接口那样必须给出所有方法的实现，因为通常一个具体类并不需要所有的方法。

### 命名规则

钩子方法的名字应当以do开始，这是熟悉设计模式的Java开发人员的标准做法。在上面的例子中，钩子方法hookMethod()应当以do开头；在HttpServlet类中，也遵从这一命名规则，如doGet()、doPost()等方法。

### 模板方法模式在Servlet中的应用



**HttpServlet担任抽象模板角色**

模板方法：由service()方法担任。Service()方法骨架是获取请求的方式（get、post、put、delete、head...），然后分别调用对应的钩子方法（doGet,doPost....）

基本方法：由doPost()、doGet()等方法担任。

**TestServlet担任具体模板角色**

TestServlet置换掉了父类HttpServlet中七个基本方法中的其中两个，分别是doGet()和doPost()。

## 访问者模式（Visitor Pattern）

**访问者模式是对象的行为模式。访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保持不变。**

变量被声明时的类型叫做变量的静态类型(Static Type)，有些人又把静态类型叫做明显类型(Apparent Type)；而变量所引用的对象的真实类型又叫做变量的实际类型(Actual Type)。比如：

List list = null;

list = new ArrayList();

声明了一个变量list，它的静态类型（明显类型）是List，而它的实际类型是ArrayList。

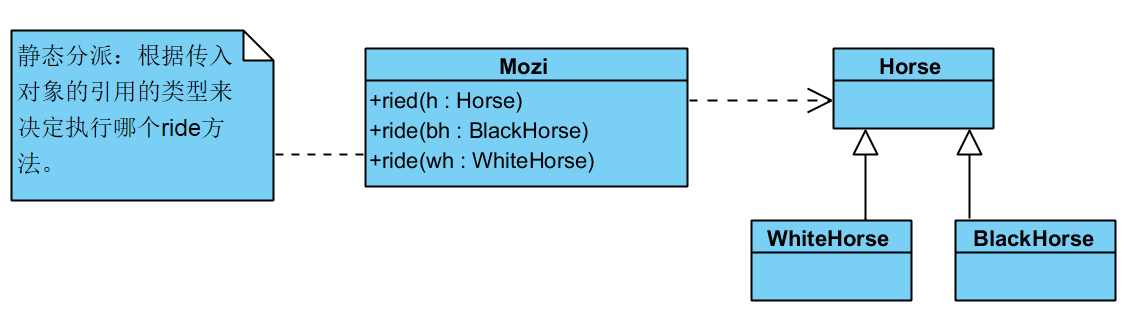
根据对象的类型而对方法进行的选择，就是分派(Dispatch)，分派(Dispatch)又分为两种，即**静态分派**和**动态分派**。

### **静态分派**

**静态分派(Static Dispatch)**发生在编译时期，分派根据静态类型信息发生。静态分派对于我们来说并不陌生，方法重载就是静态分派。

Java通过方法重载支持静态分派。Java中执行重载函数中的哪个函数取决于传入参数的对象的引用类型，而不是取决于传入对象的实际类型。

结构：



代码：designpattrn.behavior.visitor.Main.staticDispatch()

### **动态分派**

**动态分派(Dynamic Dispatch)**发生在运行时期，动态分派动态地置换掉某个方法。

Java通过方法的重写支持动态分派。Java中执行重写函数函数中的哪个函数取决于传入参数的对象的实际类型，而不是取决于传入参数的对象的引用类型。

代码：designpattrn.behavior.visitor.Main.dynamicDispatch()

### **分类的类型**

一个方法所属的对象叫做方法的接收者。方法的接收者与方法的参数统称做方法的宗量。

根据分派可以基于多少种宗量，可以将面向对象的语言划分为**单分派语言(Uni-Dispatch)**和**多分派语言(Multi-Dispatch)**。单分派语言根据一个宗量的类型进行对方法的选择，多分派语言根据多于一个的宗量的类型对方法进行选择。

理解：

1. 静态分派中，编译器要判断接受者的静态类型，确定是哪个类上的方法，这个时候已经判定了接受者这个宗量了；
2. 编译器还需要确定参数的静态类型，已确定重载方法中的版本，这个时候又判定了参数这个宗量。编译判断了哪个对象（方法接收者，父子对象）调用哪个方法（方法参数[对象的引用的类型]）

所以静态分派是多分派。

1. 到了运行期，虚拟机首先就是判断接收者的实际类型，去其上搜索匹配的方法，只判定了接收者的实际类型，方法参数的类型已经不重要了。运行期只判断哪个对象（方法接收者），不判断调用哪个方法（方法参数[实际对象的类型]））

所以动态分派是单分派。

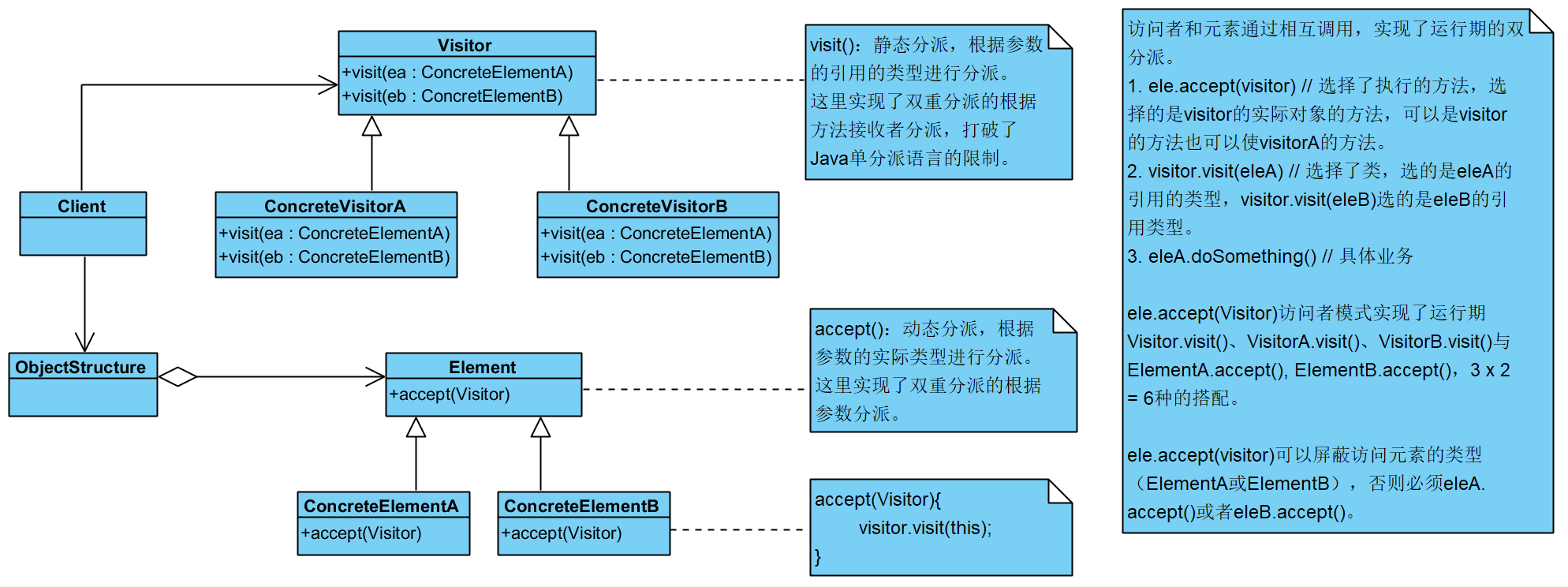
C++和Java均是单分派语言，多分派语言的例子包括CLOS和Cecil。按照这样的区分，Java就是**动态的单分派语言**，因为这种语言的动态分派仅仅会考虑到方法的接收者的类型，同时又是**静态的多分派语言**，因为这种语言对重载方法的分派会考虑到方法的接收者的类型以及方法的所有参数的类型。

在一个支持动态单分派的语言里面，有两个条件决定了一个请求会调用哪一个操作：一是请求的名字，而是接收者的真实类型。单分派限制了方法的选择过程，使得只有一个宗量可以被考虑到，这个宗量通常就是方法的接收者。在Java语言里面，如果一个操作是作用于某个类型不明的对象上面，那么对这个对象的真实类型测试仅会发生一次，这就是动态的单分派的特征。

### **双重分派**

一个方法根据两个宗量的类型来决定执行不同的代码，这就是“双重分派”。Java语言不支持动态的多分派，也就意味着Java不支持动态的双分派。但是通过使用设计模式（访问者模式），也可以在Java语言里实现动态的双重分派。

结构：



访问者模式结构图中包含以下5个角色：

1. **Visitor（抽象访问者）**：抽象访问者为对象结构中每一个具体元素类ConcreteElement声明一个访问操作，从这个操作的名称或参数类型可以清楚知道需要访问的具体元素的类型，具体访问者则需要实现这些操作方法，定义对这些元素的访问操作。
2. **ConcreteVisitor（具体访问者）**：具体访问者实现了抽象访问者声明的方法，每一个操作作用于访问对象结构中一种类型的元素。
3. **Element（抽象元素）**：一般是一个抽象类或接口，定义一个Accept方法，该方法通常以一个抽象访问者作为参数。
4. **ConcreteElement（具体元素）**：具体元素实现了Accept方法，在Accept方法中调用访问者的访问方法以便完成一个元素的操作。
5. **ObjectStructure（对象结构）**：对象结构是一个元素的集合，用于存放元素对象，且提供便利其内部元素的方法。

代码：designpattrn.behavior.visitor.Main.visitor()

优点：

1. 增加新的访问操作十分方便，不痛不痒，**符合开闭原则**。
2. 将有关元素对象的访问行为集中到一个访问者对象中，而不是分散在一个个的元素类中，类的职责更加清晰， **符合单一职责原则**。

缺点：

1. 增加新的元素类很困难，需要在每一个访问者类中增加相应访问操作代码，违背了**开闭原则**。
2. 元素对象有时候必须暴露一些自己的内部操作和状态，否则无法供访问者访问，**破坏了元素的封装性**。

应用场景：

1. 一个对象结构包含多个类型的对象，希望对这些对象实施一些依赖其具体类型的操作。**不同的类型可以有不同的访问操作**。
2. 对象结构中对象对应的类**很少改变**，但经常需要在此对象结构上定义新的操作。

# 设计模式在jdk中的应用

## 单例模式

• java.lang.Runtime#getRuntime()

• java.awt.Desktop#getDesktop()

## 原型模式

• java.lang.Object#clone() (支持浅克隆的类必须实现java.lang.Cloneable接口)

## 工厂方法模式

• java.lang.Object#toString() (在其子类中可以覆盖该方法)

• java.lang.Class#newInstance()

• java.lang.Integer#valueOf(String) (Boolean, Byte, Character,Short, Long, Float 和 Double与之类似)

• java.lang.Class#forName()

• java.lang.reflect.Array#newInstance()

• java.lang.reflect.Constructor#newInstance()

## 抽象工厂模式

• java.util.Calendar#getInstance()

• java.util.Arrays#asList()

• java.util.ResourceBundle#getBundle()

• java.net.URL#openConnection()

• java.sql.DriverManager#getConnection()

• java.sql.Connection#createStatement()

• java.sql.Statement#executeQuery()

• java.text.NumberFormat#getInstance()

• java.lang.management.ManagementFactory (所有getXXX()方法)

• java.nio.charset.Charset#forName()

• javax.xml.parsers.DocumentBuilderFactory#newInstance()

• javax.xml.transform.TransformerFactory#newInstance()

• javax.xml.xpath.XPathFactory#newInstance()

## 建造者模式

• java.lang.StringBuilder#append()

• java.lang.StringBuffer#append()

• java.nio.ByteBuffer#put() (CharBuffer, ShortBuffer, IntBuffer,LongBuffer, FloatBuffer 和DoubleBuffer与之类似)

• javax.swing.GroupLayout.Group#addComponent()

• java.sql.PreparedStatement

• java.lang.Appendable的所有实现类

## 适配器模式

•java.util.Arrays#asList()

•javax.swing.JTable(TableModel)

•java.io.InputStreamReader(InputStream)

•java.io.OutputStreamWriter(OutputStream)

•javax.xml.bind.annotation.adapters.XmlAdapter#marshal()

•javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal()

## 桥接模式

• AWT (提供了抽象层映射于实际的操作系统)

•JDBC

## 组合模式

•javax.swing.JComponent#add(Component)

•java.awt.[Container](http://lib.csdn.net/base/4" \o "Docker知识库" \t "https://blog.csdn.net/u013782203/article/details/_blank)#add(Component)

•java.util.Map#putAll(Map)

•java.util.List#addAll(Collection)

•java.util.Set#addAll(Collection)

## 外观模式

•java.lang.Class

•javax.faces.webapp.FacesServlet

## 装饰模式

•java.io.BufferedInputStream(InputStream)

•java.io.DataInputStream(InputStream)

•java.io.BufferedOutputStream(OutputStream)

•java.util.zip.ZipOutputStream(OutputStream)

•java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

## 享元模式

•java.lang.Integer#valueOf(int)

•java.lang.Boolean#valueOf(boolean)

• java.lang.Byte#valueOf(byte)

•java.lang.Character#valueOf(char)

## 代理模式

• java.lang.reflect.Proxy

•java.rmi.\*

## 责任链模式

•java.util.logging.Logger#log()

•javax.servlet.Filter#doFilter()

## 命令模式

• java.lang.Runnable

• javax.swing.Action

## 解释器模式

• java.util.Pattern

• java.text.Normalizer

• java.text.Format

• javax.el.ELResolver

## 迭代器模式

• java.util.Iterator

• java.util.Enumeration

## 中介者模式

• java.util.Timer (所有scheduleXXX()方法)

• java.util.concurrent.Executor#execute()

• java.util.concurrent.ExecutorService (invokeXXX()和submit()方法)

• java.util.concurrent.ScheduledExecutorService (所有scheduleXXX()方法)

•java.lang.reflect.Method#invoke()

## 备忘录模式

•java.util.Date

•java.io.Serializable

•javax.faces.component.StateHolder

## 观察者模式

•java.util.Observer/java.util.Observable

•java.util.EventListener (所有子类)

•javax.servlet.http.HttpSessionBindingListener

•javax.servlet.http.HttpSessionAttributeListener

•javax.faces.event.PhaseListener

## 状态模式

•java.util.Iterator

•javax.faces.lifecycle.LifeCycle#execute()

## 策略模式

• java.util.Comparator#compare()

• javax.servlet.http.HttpServlet

• javax.servlet.Filter#doFilter()

## 模板方法模式

•java.io.InputStream, java.io.OutputStream, java.io.Reader和java.io.Writer的所有非抽象方法

•java.util.AbstractList, java.util.AbstractSet和java.util.AbstractMap的所有非抽象方法

•javax.servlet.http.HttpServlet#doXXX()

## 访问者模式

•javax.lang.model.element.AnnotationValue和AnnotationValueVisitor

•javax.lang.model.element.Element和ElementVisitor

•javax.lang.model.type.TypeMirror和TypeVisitor

# 引用

1. [java 23种设计模式 深入理解](https://www.cnblogs.com/foryang/p/5849402.html)https://www.cnblogs.com/foryang/p/5849402.html
2. [设计模式：原型模式https://www.cnblogs.com/songyaqi/p/PrototypePattern.html](https://www.cnblogs.com/songyaqi/p/PrototypePattern.html)
3. 设计模式之原型模式https://blog.csdn.net/chenliguan/article/details/69855738
4. Spring工厂方法与FactoryBean <https://blog.csdn.net/w_linux/article/details/80063062>
5. 程序设计-多用组合少用继承原则https://blog.csdn.net/zw3413/article/details/80077772
6. [《JAVA设计模式》之桥接模式(Bridge)](https://www.cnblogs.com/betterboyz/p/9361784.html)https://www.cnblogs.com/betterboyz/p/9361784.html
7. [结构型—桥接（Bridge）模式](https://www.cnblogs.com/-crazysnail/p/3977815.html) <https://www.cnblogs.com/-crazysnail/p/3977815.html>
8. [设计模式笔记：单一职责原则（SRP, Single Responsibility Principle）](https://www.cnblogs.com/libingql/p/3804327.html)https://www.cnblogs.com/libingql/p/3804327.html
9. [设计模式笔记：开闭原则（OCP，The Open-Closed Principle）](https://www.cnblogs.com/libingql/p/3804655.html)https://www.cnblogs.com/libingql/p/3804655.html
10. [【面向对象设计原则】之依赖倒置原则（DIP）](https://www.cnblogs.com/vaiyanzi/p/6904449.html)https://www.cnblogs.com/vaiyanzi/p/6904449.html
11. [设计模式原则之接口隔离原则](https://www.cnblogs.com/oumyye/p/4392854.html) [www.cnblogs.com/oumyye/p/4392854.html](http://www.cnblogs.com/oumyye/p/4392854.html)
12. [java设计模式之组合模式](https://www.cnblogs.com/lfxiao/p/6816026.html) <https://www.cnblogs.com/lfxiao/p/6816026.html>
13. [java设计模式之外观模式（门面模式）](https://www.cnblogs.com/lthIU/p/5860607.html) https://www.cnblogs.com/lthIU/p/5860607.html
14. 以jq为案例查看外观模式https://cloud.tencent.com/developer/article/1374509
15. 代理模式https://wenku.baidu.com/view/4fffc00e7cd184254b3535ed.html
16. 代理模式（三）：远程代理，虚拟代理，缓冲代理https://blog.csdn.net/will130/article/details/50729535
17. 详解设计模式中的proxy代理模式及在Java程序中的实现https://www.jb51.net/article/84659.htm
18. [《JAVA与模式》之责任链模式](https://www.cnblogs.com/java-my-life/archive/2012/05/28/2516865.html) <http://www.cnblogs.com/java-my-life/archive/2012/05/28/2516865.html>
19. [设计模式解密（20）- 职责链模式](https://www.cnblogs.com/JsonShare/p/7357673.html)<https://www.cnblogs.com/JsonShare/p/7357673.html>
20. [重温设计模式（三）——职责链模式(chain of responsibility)](https://www.cnblogs.com/langtianya/p/4060941.html) <https://www.cnblogs.com/langtianya/p/4060941.html>
21. 解释器模式（详解版）http://c.biancheng.net/view/1402.html
22. 抽象工厂模式 <http://www.cnblogs.com/java-my-life/archive/2012/03/28/2418836.html>
23. 工厂方法 [http://www.cnblogs.com/java-my-life/archive/2012/03/25/2416227.html](http://www.cnblogs.com/java-my-life/archive/2012/03/28/2418836.html)
24. 建造者模式  [http://www.cnblogs.com/java-my-life/archive/2012/04/07/2433939.html](http://www.cnblogs.com/java-my-life/archive/2012/04/07/2433939.html" \t "https://www.cnblogs.com/foryang/p/_blank)
25. 原型模式 [http://www.cnblogs.com/java-my-life/archive/2012/04/11/2439387.html](http://www.cnblogs.com/java-my-life/archive/2012/04/11/2439387.html" \t "https://www.cnblogs.com/foryang/p/_blank)
26. 单态模式 [http://www.cnblogs.com/java-my-life/archive/2012/03/31/2425631.html](http://www.cnblogs.com/java-my-life/archive/2012/03/31/2425631.html" \t "https://www.cnblogs.com/foryang/p/_blank)
27. 适配器模式 [http://www.cnblogs.com/java-my-life/archive/2012/04/13/2442795.html](http://www.cnblogs.com/java-my-life/archive/2012/04/13/2442795.html" \t "https://www.cnblogs.com/foryang/p/_blank)
28. 桥接模式 [http://blog.csdn.net/jason0539/article/details/22568865](http://blog.csdn.net/jason0539/article/details/22568865" \t "https://www.cnblogs.com/foryang/p/_blank)
29. 组合模式 [http://blog.csdn.net/jason0539/article/details/22642281](http://blog.csdn.net/jason0539/article/details/22642281" \t "https://www.cnblogs.com/foryang/p/_blank)
30. 外观模式 [http://blog.csdn.net/jason0539/article/details/22775311](http://blog.csdn.net/jason0539/article/details/22775311" \t "https://www.cnblogs.com/foryang/p/_blank)
31. 装饰者模式 [http://www.cnblogs.com/java-my-life/archive/2012/04/20/2455726.html](http://www.cnblogs.com/java-my-life/archive/2012/04/20/2455726.html" \t "https://www.cnblogs.com/foryang/p/_blank)
32. 享元模式 [http://www.cnblogs.com/java-my-life/archive/2012/04/26/2468499.html](http://www.cnblogs.com/java-my-life/archive/2012/04/26/2468499.html" \t "https://www.cnblogs.com/foryang/p/_blank)
33. 代理模式 [http://www.cnblogs.com/java-my-life/archive/2012/04/23/2466712.html](http://www.cnblogs.com/java-my-life/archive/2012/04/23/2466712.html" \t "https://www.cnblogs.com/foryang/p/_blank)
34. 责任链模式 [http://blog.csdn.net/zhouyong0/article/details/7909456](http://blog.csdn.net/zhouyong0/article/details/7909456" \t "https://www.cnblogs.com/foryang/p/_blank)
35. 命令模式 [http://www.cnblogs.com/java-my-life/archive/2012/06/01/2526972.html](http://www.cnblogs.com/java-my-life/archive/2012/06/01/2526972.html" \t "https://www.cnblogs.com/foryang/p/_blank)
36. 解释器模式 [http://www.cnblogs.com/java-my-life/archive/2012/06/19/2552617.html](http://www.cnblogs.com/java-my-life/archive/2012/06/19/2552617.html" \t "https://www.cnblogs.com/foryang/p/_blank)
37. 迭代模式 [http://www.cnblogs.com/java-my-life/archive/2012/05/22/2511506.html](http://www.cnblogs.com/java-my-life/archive/2012/05/22/2511506.html" \t "https://www.cnblogs.com/foryang/p/_blank)
38. 中介者模式 [http://blog.csdn.net/chenhuade85/article/details/8141831](http://blog.csdn.net/chenhuade85/article/details/8141831" \t "https://www.cnblogs.com/foryang/p/_blank)
39. 备忘录模式 [http://www.cnblogs.com/java-my-life/archive/2012/06/06/2534942.html](http://www.cnblogs.com/java-my-life/archive/2012/06/06/2534942.html" \t "https://www.cnblogs.com/foryang/p/_blank)
40. 观察者模式 [http://www.cnblogs.com/java-my-life/archive/2012/05/16/2502279.html](http://www.cnblogs.com/java-my-life/archive/2012/05/16/2502279.html" \t "https://www.cnblogs.com/foryang/p/_blank)
41. 状态模式 [http://www.cnblogs.com/java-my-life/archive/2012/06/08/2538146.html](http://www.cnblogs.com/java-my-life/archive/2012/06/08/2538146.html" \t "https://www.cnblogs.com/foryang/p/_blank)
42. 策略模式 [http://www.cnblogs.com/java-my-life/archive/2012/05/10/2491891.html](http://www.cnblogs.com/java-my-life/archive/2012/05/10/2491891.html" \t "https://www.cnblogs.com/foryang/p/_blank)
43. 模板方法模式 [http://www.cnblogs.com/java-my-life/archive/2012/05/14/2495235.html](http://www.cnblogs.com/java-my-life/archive/2012/05/14/2495235.html" \t "https://www.cnblogs.com/foryang/p/_blank)
44. 访问者模式 [http://www.cnblogs.com/java-my-life/archive/2012/06/14/2545381.html](http://www.cnblogs.com/java-my-life/archive/2012/06/14/2545381.html" \t "https://www.cnblogs.com/foryang/p/_blank)
45. 设计模式 | 中介者模式及典型应用https://blog.csdn.net/wwwdc1012/article/details/83389158
46. JAVA设计模式--中介者模式https://blog.csdn.net/pengjunlee/article/details/54984322
47. 备忘录模式——java设计模式（二十）https://blog.csdn.net/qq\_40369829/article/details/80370606
48. [备忘录模式-Memento Pattern(Java实现)](https://www.cnblogs.com/noKing/p/9069079.html) [www.cnblogs.com/noKing/p/9069079.html](http://www.cnblogs.com/noKing/p/9069079.html)
49. 备忘录模式https://blog.csdn.net/lyabc123456/article/details/80740819
50. [JAVA设计模式：状态模式](https://www.cnblogs.com/pony1223/p/7518226.html) <https://www.cnblogs.com/pony1223/p/7518226.html>
51. 设计模式在jdk中的应用https://blog.csdn.net/u013782203/article/details/52214393
52. [Examples of GoF Design Patterns in Java's core libraries](https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries) <https://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries>
53. 策略模式https://blog.51cto.com/turnsole/2299661
54. Java中，状态模式和策略模式的区别https://blog.csdn.net/songwenbinasdf/article/details/51831926
55. [设计模式的征途—16.访问者（Visitor）模式](https://www.cnblogs.com/edisonchou/p/7247990.html) <https://www.cnblogs.com/edisonchou/p/7247990.html>
56. Java的方法调用机制https://blog.csdn.net/yuhongye111/article/details/30348331