# [一：适配器模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id16)

**目录**

* [适配器模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#adapter)
  + [模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id3)
  + [模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id4)
  + [模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id5)
  + [时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id6)
  + [代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id7)
  + [模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id8)
  + [实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id9)
  + [优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id10)
  + [缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id11)
  + [适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id12)
  + [模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id13)
  + [模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id14)
  + [总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id15)

## [1.1. 模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id17)

* 在软件开发中采用类似于电源适配器的设计和编码技巧被称为适配器模式。
* 通常情况下，客户端可以通过目标类的接口访问它所提供的服务。有时，现有的类可以满足客户类的功能需要，但是它所提供的接口不一定是客户类所期望的，这可能是因为现有类中方法名与目标类中定义的方法名不一致等原因所导致的。
* 在这种情况下，现有的接口需要转化为客户类期望的接口，这样保证了对现有类的重用。如果不进行这样的转化，客户类就不能利用现有类所提供的功能，适配器模式可以完成这样的转化。
* 在适配器模式中可以定义一个包装类，包装不兼容接口的对象，这个包装类指的就是适配器(Adapter)，它所包装的对象就是适配者(Adaptee)，即被适配的类。
* 适配器提供客户类需要的接口，适配器的实现就是把客户类的请求转化为对适配者的相应接口的调用。也就是说：当客户类调用适配器的方法时，在适配器类的内部将调用适配者类的方法，而这个过程对客户类是透明的，客户类并不直接访问适配者类。因此，适配器可以使由于接口不兼容而不能交互的类可以一起工作。这就是适配器模式的模式动机。

## [1.2. 模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id18)

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

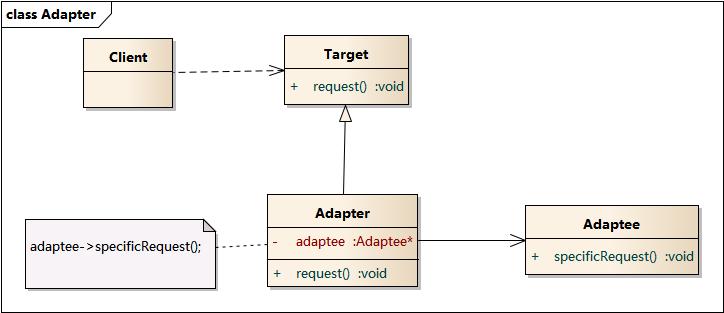
## [1.3. 模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id19)

适配器模式包含如下角色：

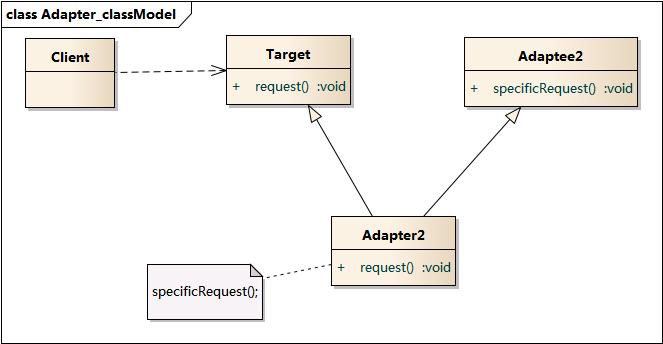
* Target：目标抽象类
* Adapter：适配器类
* Adaptee：适配者类
* Client：客户类

适配器模式有对象适配器和类适配器两种实现：

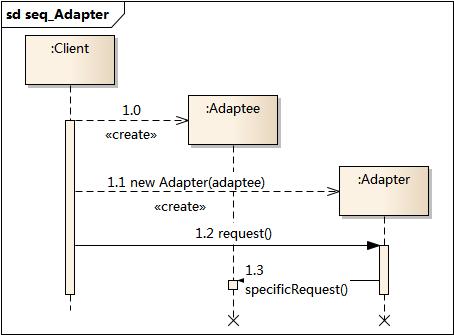
对象适配器：



类适配器：



## [1.4. 时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id20)

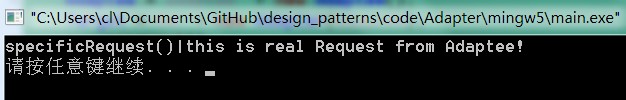


## [1.5. 代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id21)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | #include <iostream>  #include "Adapter.h"  #include "Adaptee.h"  #include "Target.h"  using namespace std;  int main(int argc, char \*argv[])  {  Adaptee \* adaptee = new Adaptee();  Target \* tar = new Adapter(adaptee);  tar->request();    return 0;  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | ///////////////////////////////////////////////////////////  // Adapter.h  // Implementation of the Class Adapter  // Created on: 03-十月-2014 17:32:00  // Original author: colin  ///////////////////////////////////////////////////////////  #if !defined(EA\_BD766D47\_0C69\_4131\_B7B9\_21DF78B1E80D\_\_INCLUDED\_)  #define EA\_BD766D47\_0C69\_4131\_B7B9\_21DF78B1E80D\_\_INCLUDED\_  #include "Target.h"  #include "Adaptee.h"  class Adapter : public Target  {  public:  Adapter(Adaptee \*adaptee);  virtual ~Adapter();  virtual void request();  private:  Adaptee\* m\_pAdaptee;  };  #endif // !defined(EA\_BD766D47\_0C69\_4131\_B7B9\_21DF78B1E80D\_\_INCLUDED\_) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | ///////////////////////////////////////////////////////////  // Adapter.cpp  // Implementation of the Class Adapter  // Created on: 03-十月-2014 17:32:00  // Original author: colin  ///////////////////////////////////////////////////////////  #include "Adapter.h"  Adapter::Adapter(Adaptee \* adaptee){  m\_pAdaptee = adaptee;  }  Adapter::~Adapter(){  }  void Adapter::request(){  m\_pAdaptee->specificRequest();  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | ///////////////////////////////////////////////////////////  // Adaptee.h  // Implementation of the Class Adaptee  // Created on: 03-十月-2014 17:32:00  // Original author: colin  ///////////////////////////////////////////////////////////  #if !defined(EA\_826E6B4F\_12BE\_4609\_A0A3\_95BD5E657D36\_\_INCLUDED\_)  #define EA\_826E6B4F\_12BE\_4609\_A0A3\_95BD5E657D36\_\_INCLUDED\_  class Adaptee  {  public:  Adaptee();  virtual ~Adaptee();  void specificRequest();  };  #endif // !defined(EA\_826E6B4F\_12BE\_4609\_A0A3\_95BD5E657D36\_\_INCLUDED\_) |

运行结果：



## [1.6. 模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id22)

## [1.7. 实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id23)

## [1.8. 优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id24)

* 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码。
* 增加了类的透明性和复用性，将具体的实现封装在适配者类中，对于客户端类来说是透明的，而且提高了适配者的复用性。
* 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”。

**类适配器模式还具有如下优点：**

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

**对象适配器模式还具有如下优点：**

一个对象适配器可以把多个不同的适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口。

## [1.9. 缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id25)

**类适配器模式的缺点如下：**

对于Java、C#等不支持多重继承的语言，一次最多只能适配一个适配者类，而且目标抽象类只能为抽象类，不能为具体类，其使用有一定的局限性，不能将一个适配者类和它的子类都适配到目标接口。

**对象适配器模式的缺点如下：**

与类适配器模式相比，要想置换适配者类的方法就不容易。如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

## [1.10. 适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id26)

在以下情况下可以使用适配器模式：

* 系统需要使用现有的类，而这些类的接口不符合系统的需要。
* 想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

## [1.11. 模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id27)

Sun公司在1996年公开了Java语言的数据库连接工具JDBC，JDBC使得Java语言程序能够与数据库连接，并使用SQL语言来查询和操作数据。JDBC给出一个客户端通用的抽象接口，每一个具体数据库引擎（如SQL Server、Oracle、MySQL等）的JDBC驱动软件都是一个介于JDBC接口和数据库引擎接口之间的适配器软件。抽象的JDBC接口和各个数据库引擎API之间都需要相应的适配器软件，这就是为各个不同数据库引擎准备的驱动程序。

## [1.12. 模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id28)

**认适配器模式(Default Adapter Pattern)或缺省适配器模式**

当不需要全部实现接口提供的方法时，可先设计一个抽象类实现接口，并为该接口中每个方法提供一个默认实现（空方法），那么该抽象类的子类可有选择地覆盖父类的某些方法来实现需求，它适用于一个接口不想使用其所有的方法的情况。因此也称为单接口适配器模式。

## [1.13. 总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/adapter.html#id29)

* 结构型模式描述如何将类或者对象结合在一起形成更大的结构。
* 适配器模式用于将一个接口转换成客户希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作，其别名为包装器。适配器模式既可以作为类结构型模式，也可以作为对象结构型模式。
* 适配器模式包含四个角色：目标抽象类定义客户要用的特定领域的接口；适配器类可以调用另一个接口，作为一个转换器，对适配者和抽象目标类进行适配，它是适配器模式的核心；适配者类是被适配的角色，它定义了一个已经存在的接口，这个接口需要适配；在客户类中针对目标抽象类进行编程，调用在目标抽象类中定义的业务方法。
* 在类适配器模式中，适配器类实现了目标抽象类接口并继承了适配者类，并在目标抽象类的实现方法中调用所继承的适配者类的方法；在对象适配器模式中，适配器类继承了目标抽象类并定义了一个适配者类的对象实例，在所继承的目标抽象类方法中调用适配者类的相应业务方法。
* 适配器模式的主要优点是将目标类和适配者类解耦，增加了类的透明性和复用性，同时系统的灵活性和扩展性都非常好，更换适配器或者增加新的适配器都非常方便，符合“开闭原则”；类适配器模式的缺点是适配器类在很多编程语言中不能同时适配多个适配者类，对象适配器模式的缺点是很难置换适配者类的方法。
* 适配器模式适用情况包括：系统需要使用现有的类，而这些类的接口不符合系统的需要；想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类一起工作。

# 二：[外观模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id16)

**目录**

* [外观模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#facade)
  + [模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id3)
  + [模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id4)
  + [模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id5)
  + [时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id6)
  + [代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id7)
  + [模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id8)
  + [实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id9)
  + [优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id10)
  + [缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id11)
  + [适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id12)
  + [模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id13)
  + [模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id14)
  + [总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id15)

## 1[模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id17)

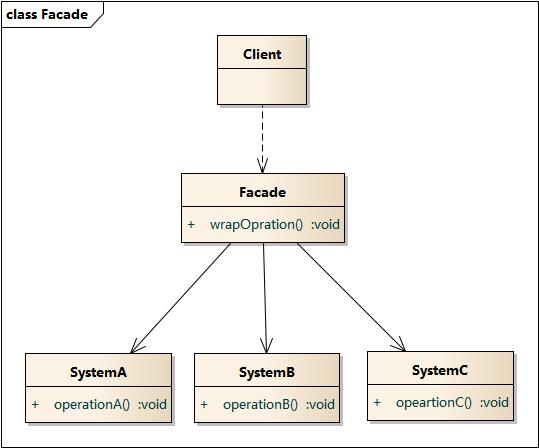
## [2.2模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id18)

外观模式(Facade Pattern)：外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式又称为门面模式，它是一种对象结构型模式。

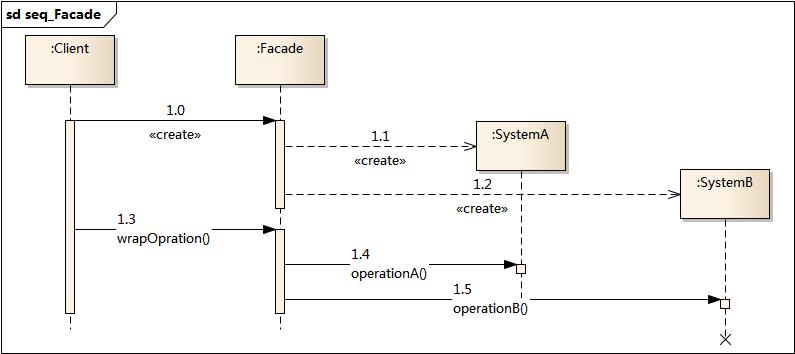
## [2.3模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id19)

外观模式包含如下角色：

* Facade: 外观角色
* SubSystem:子系统角色



## [2.4时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id20)



## [2.5代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id21)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #include <iostream>  #include "Facade.h"  using namespace std;  int main(int argc, char \*argv[])  {  Facade fa;  fa.wrapOpration();    return 0;  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | ///////////////////////////////////////////////////////////  // Facade.h  // Implementation of the Class Facade  // Created on: 06-十月-2014 19:10:44  // Original author: colin  ///////////////////////////////////////////////////////////  #if !defined(EA\_FD130A87\_92A9\_4168\_9B33\_7A925C47AFD5\_\_INCLUDED\_)  #define EA\_FD130A87\_92A9\_4168\_9B33\_7A925C47AFD5\_\_INCLUDED\_  #include "SystemC.h"  #include "SystemA.h"  #include "SystemB.h"  class Facade  {  public:  Facade();  virtual ~Facade();  void wrapOpration();  private:  SystemC \*m\_SystemC;  SystemA \*m\_SystemA;  SystemB \*m\_SystemB;  };  #endif // !defined(EA\_FD130A87\_92A9\_4168\_9B33\_7A925C47AFD5\_\_INCLUDED\_) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | ///////////////////////////////////////////////////////////  // Facade.cpp  // Implementation of the Class Facade  // Created on: 06-十月-2014 19:10:44  // Original author: colin  ///////////////////////////////////////////////////////////  #include "Facade.h"  Facade::Facade(){  m\_SystemA = new SystemA();  m\_SystemB = new SystemB();  m\_SystemC = new SystemC();  }  Facade::~Facade(){  delete m\_SystemA;  delete m\_SystemB;  delete m\_SystemC;  }  void Facade::wrapOpration(){  m\_SystemA->operationA();  m\_SystemB->operationB();  m\_SystemC->opeartionC();  } |

运行结果：



## [2.6模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id22)

根据“单一职责原则”，在软件中将一个系统划分为若干个子系统有利于降低整个系统的复杂性，一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小，而达到该目标的途径之一就是引入一个外观对象，它为子系统的访问提供了一个简单而单一的入口。 -外观模式也是“迪米特法则”的体现，通过引入一个新的外观类可以降低原有系统的复杂度，同时降低客户类与子系统类的耦合度。 - 外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，而不需要与子系统内部的很多对象打交道。 -外观模式的目的在于降低系统的复杂程度。 -外观模式从很大程度上提高了客户端使用的便捷性，使得客户端无须关心子系统的工作细节，通过外观角色即可调用相关功能。

## [2.7实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id23)

## [2.8 优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id24)

外观模式的优点

* 对客户屏蔽子系统组件，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。通过引入外观模式，客户代码将变得很简单，与之关联的对象也很少。
* 实现了子系统与客户之间的松耦合关系，这使得子系统的组件变化不会影响到调用它的客户类，只需要调整外观类即可。
* 降低了大型软件系统中的编译依赖性，并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一般不需要编译所有其他的子系统。一个子系统的修改对其他子系统没有任何影响，而且子系统内部变化也不会影响到外观对象。
* 只是提供了一个访问子系统的统一入口，并不影响用户直接使用子系统类。

## [2.9缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id25)

外观模式的缺点

* 不能很好地限制客户使用子系统类，如果对客户访问子系统类做太多的限制则减少了可变性和灵活性。
* 在不引入抽象外观类的情况下，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”。

## [2.10适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id26)

在以下情况下可以使用外观模式：

* 当要为一个复杂子系统提供一个简单接口时可以使用外观模式。该接口可以满足大多数用户的需求，而且用户也可以越过外观类直接访问子系统。
* 客户程序与多个子系统之间存在很大的依赖性。引入外观类将子系统与客户以及其他子系统解耦，可以提高子系统的独立性和可移植性。
* 在层次化结构中，可以使用外观模式定义系统中每一层的入口，层与层之间不直接产生联系，而通过外观类建立联系，降低层之间的耦合度。

## [2.11 模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id27)

## [2.12 模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id28)

**一个系统有多个外观类**

在外观模式中，通常只需要一个外观类，并且此外观类只有一个实例，换言之它是一个单例类。在很多情况下为了节约系统资源，一般将外观类设计为单例类。当然这并不意味着在整个系统里只能有一个外观类，在一个系统中可以设计多个外观类，每个外观类都负责和一些特定的子系统交互，向用户提供相应的业务功能。

**不要试图通过外观类为子系统增加新行为**

不要通过继承一个外观类在子系统中加入新的行为，这种做法是错误的。外观模式的用意是为子系统提供一个集中化和简化的沟通渠道，而不是向子系统加入新的行为，新的行为的增加应该通过修改原有子系统类或增加新的子系统类来实现，不能通过外观类来实现。

**外观模式与迪米特法则**

外观模式创造出一个外观对象，将客户端所涉及的属于一个子系统的协作伙伴的数量减到最少，使得客户端与子系统内部的对象的相互作用被外观对象所取代。外观类充当了客户类与子系统类之间的“第三者”，降低了客户类与子系统类之间的耦合度，外观模式就是实现代码重构以便达到“迪米特法则”要求的一个强有力的武器。

**抽象外观类的引入**

外观模式最大的缺点在于违背了“开闭原则”，当增加新的子系统或者移除子系统时需要修改外观类，可以通过引入抽象外观类在一定程度上解决该问题，客户端针对抽象外观类进行编程。对于新的业务需求，不修改原有外观类，而对应增加一个新的具体外观类，由新的具体外观类来关联新的子系统对象，同时通过修改配置文件来达到不修改源代码并更换外观类的目的。

## [2.13 总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/facade.html#id29)

* 在外观模式中，外部与一个子系统的通信必须通过一个统一的外观对象进行，为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。外观模式又称为门面模式，它是一种对象结构型模式。
* 外观模式包含两个角色：外观角色是在客户端直接调用的角色，在外观角色中可以知道相关的(一个或者多个)子系统的功能和责任，它将所有从客户端发来的请求委派到相应的子系统去，传递给相应的子系统对象处理；在软件系统中可以同时有一个或者多个子系统角色，每一个子系统可以不是一个单独的类，而是一个类的集合，它实现子系统的功能。
* 外观模式要求一个子系统的外部与其内部的通信通过一个统一的外观对象进行，外观类将客户端与子系统的内部复杂性分隔开，使得客户端只需要与外观对象打交道，而不需要与子系统内部的很多对象打交道。
* 外观模式主要优点在于对客户屏蔽子系统组件，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，它实现了子系统与客户之间的松耦合关系，并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；其缺点在于不能很好地限制客户使用子系统类，而且在不引入抽象外观类的情况下，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”。
* 外观模式适用情况包括：要为一个复杂子系统提供一个简单接口；客户程序与多个子系统之间存在很大的依赖性；在层次化结构中，需要定义系统中每一层的入口，使得层与层之间不直接产生联系。

# [三：装饰模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id16)

**目录**

* [装饰模式](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#decorator)
  + [模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id3)
  + [模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id4)
  + [模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id5)
  + [时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id6)
  + [代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id7)
  + [模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id8)
  + [实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id9)
  + [优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id10)
  + [缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id11)
  + [适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id12)
  + [模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id13)
  + [模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id14)
  + [总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id15)

## [3.1. 模式动机](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id17)

一般有两种方式可以实现给一个类或对象增加行为：

* 继承机制，使用继承机制是给现有类添加功能的一种有效途径，通过继承一个现有类可以使得子类在拥有自身方法的同时还拥有父类的方法。但是这种方法是静态的，用户不能控制增加行为的方式和时机。
* 关联机制，即将一个类的对象嵌入另一个对象中，由另一个对象来决定是否调用嵌入对象的行为以便扩展自己的行为，我们称这个嵌入的对象为装饰器(Decorator)

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任，换言之，客户端并不会觉得对象在装饰前和装饰后有什么不同。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。这就是装饰模式的模式动机。

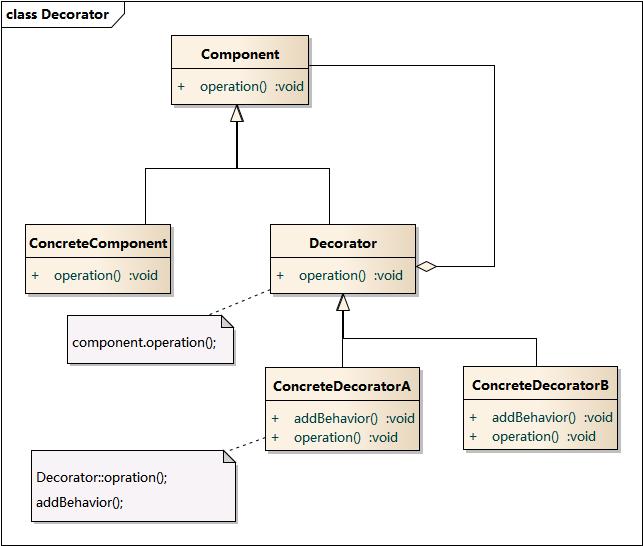
## [3.2. 模式定义](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id18)

装饰模式(Decorator Pattern) ：动态地给一个对象增加一些额外的职责(Responsibility)，就增加对象功能来说，装饰模式比生成子类实现更为灵活。其别名也可以称为包装器(Wrapper)，与适配器模式的别名相同，但它们适用于不同的场合。根据翻译的不同，装饰模式也有人称之为“油漆工模式”，它是一种对象结构型模式。

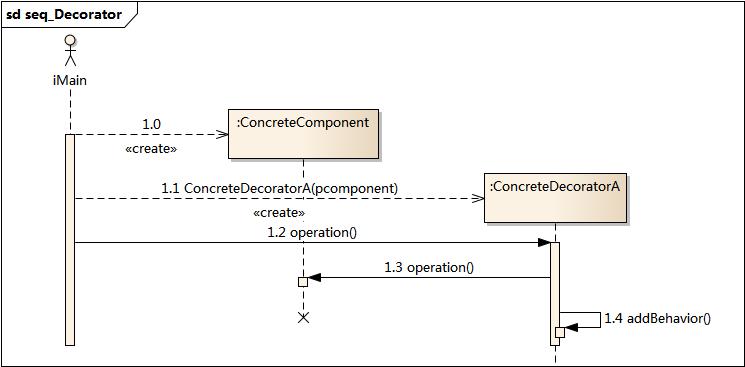
## [3.3. 模式结构](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id19)

装饰模式包含如下角色：

* Component: 抽象构件
* ConcreteComponent: 具体构件
* Decorator: 抽象装饰类
* ConcreteDecorator: 具体装饰类



## [3.4. 时序图](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id20)

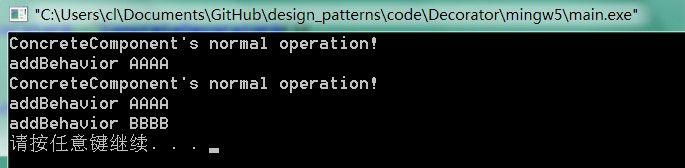


## [3.5. 代码分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id21)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | ///////////////////////////////////////////////////////////  // ConcreteComponent.cpp  // Implementation of the Class ConcreteComponent  // Created on: 03-十月-2014 18:53:00  // Original author: colin  ///////////////////////////////////////////////////////////  #include "ConcreteComponent.h"  #include <iostream>  using namespace std;  ConcreteComponent::ConcreteComponent(){  }  ConcreteComponent::~ConcreteComponent(){  }  void ConcreteComponent::operation(){  cout << "ConcreteComponent's normal operation!" << endl;  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | ///////////////////////////////////////////////////////////  // ConcreteDecoratorA.h  // Implementation of the Class ConcreteDecoratorA  // Created on: 03-十月-2014 18:53:00  // Original author: colin  ///////////////////////////////////////////////////////////  #if !defined(EA\_6786B68E\_DCE4\_44c4\_B26D\_812F0B3C0382\_\_INCLUDED\_)  #define EA\_6786B68E\_DCE4\_44c4\_B26D\_812F0B3C0382\_\_INCLUDED\_  #include "Decorator.h"  #include "Component.h"  class ConcreteDecoratorA : public Decorator  {  public:  ConcreteDecoratorA(Component\* pcmp);  virtual ~ConcreteDecoratorA();  void addBehavior();  virtual void operation();  };  #endif // !defined(EA\_6786B68E\_DCE4\_44c4\_B26D\_812F0B3C0382\_\_INCLUDED\_) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | ///////////////////////////////////////////////////////////  // ConcreteDecoratorA.cpp  // Implementation of the Class ConcreteDecoratorA  // Created on: 03-十月-2014 18:53:00  // Original author: colin  ///////////////////////////////////////////////////////////  #include "ConcreteDecoratorA.h"  #include <iostream>  using namespace std;  ConcreteDecoratorA::ConcreteDecoratorA(Component\* pcmp)  :Decorator(pcmp)  {  }  ConcreteDecoratorA::~ConcreteDecoratorA(){  }  void ConcreteDecoratorA::addBehavior(){  cout << "addBehavior AAAA" << endl;  }  void ConcreteDecoratorA::operation(){  Decorator::operation();  addBehavior();  } |

运行结果：



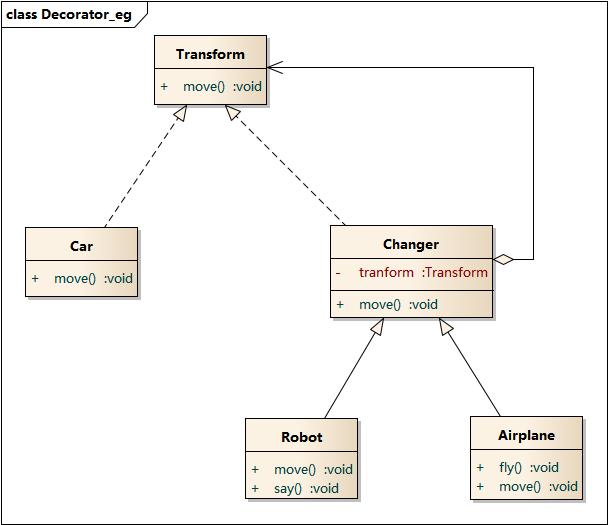
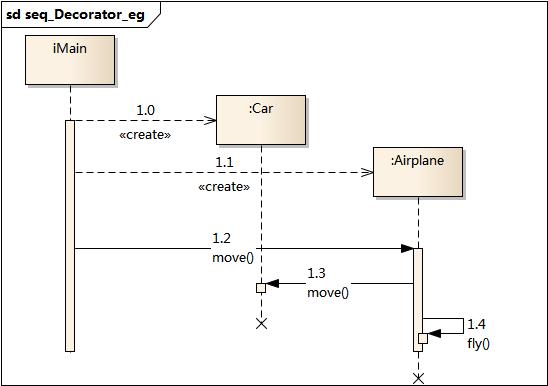
## [3.6. 模式分析](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id22)

* 与继承关系相比，关联关系的主要优势在于不会破坏类的封装性，而且继承是一种耦合度较大的静态关系，无法在程序运行时动态扩展。在软件开发阶段，关联关系虽然不会比继承关系减少编码量，但是到了软件维护阶段，由于关联关系使系统具有较好的松耦合性，因此使得系统更加容易维护。当然，关联关系的缺点是比继承关系要创建更多的对象。
* 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子类的情况下，将对象的功能加以扩展。

## [3.7. 实例](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id23)

实例：变形金刚

变形金刚在变形之前是一辆汽车，它可以在陆地上移动。当它变成机器人之后除了能够在陆地上移动之外，还可以说话；如果需要，它还可以变成飞机，除了在陆地上移动还可以在天空中飞翔。

## [3.8. 优点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id24)

装饰模式的优点:

* 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。
* 可以通过一种动态的方式来扩展一个对象的功能，通过配置文件可以在运行时选择不同的装饰器，从而实现不同的行为。
* 通过使用不同的具体装饰类以及这些装饰类的排列组合，可以创造出很多不同行为的组合。可以使用多个具体装饰类来装饰同一对象，得到功能更为强大的对象。
* 具体构件类与具体装饰类可以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类，在使用时再对其进行组合，原有代码无须改变，符合“开闭原则”

## [3.9. 缺点](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id25)

装饰模式的缺点:

* 使用装饰模式进行系统设计时将产生很多小对象，这些对象的区别在于它们之间相互连接的方式有所不同，而不是它们的类或者属性值有所不同，同时还将产生很多具体装饰类。这些装饰类和小对象的产生将增加系统的复杂度，加大学习与理解的难度。
* 这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需要逐级排查，较为烦琐。

## [3.10. 适用环境](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id26)

在以下情况下可以使用装饰模式：

* 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
* 需要动态地给一个对象增加功能，这些功能也可以动态地被撤销。
* 当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展和维护时。不能采用继承的情况主要有两类：第一类是系统中存在大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长；第二类是因为类定义不能继承（如final类）.

## [3.11. 模式应用](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id27)

## [3.12. 模式扩展](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id28)

装饰模式的简化-需要注意的问题:

* 一个装饰类的接口必须与被装饰类的接口保持相同，对于客户端来说无论是装饰之前的对象还是装饰之后的对象都可以一致对待。
* 尽量保持具体构件类Component作为一个“轻”类，也就是说不要把太多的逻辑和状态放在具体构件类中，可以通过装饰类

对其进行扩展。 - 如果只有一个具体构件类而没有抽象构件类，那么抽象装饰类可以作为具体构件类的直接子类。

## [3.13. 总结](https://design-patterns.readthedocs.io/zh_CN/latest/structural_patterns/decorator.html#id29)

* 装饰模式用于动态地给一个对象增加一些额外的职责，就增加对象功 能来说，装饰模式比生成子类实现更为灵活。它是一种对象结构型模 式。
* 装饰模式包含四个角色：抽象构件定义了对象的接口，可以给这些对 象动态增加职责（方法）；具体构件定义了具体的构件对象，实现了 在抽象构件中声明的方法，装饰器可以给它增加额外的职责（方法）； 抽象装饰类是抽象构件类的子类，用于给具体构件增加职责，但是具 体职责在其子类中实现；具体装饰类是抽象装饰类的子类，负责向构 件添加新的职责。
* 使用装饰模式来实现扩展比继承更加灵活，它以对客户透明的方式动 态地给一个对象附加更多的责任。装饰模式可以在不需要创造更多子 类的情况下，将对象的功能加以扩展。
* 装饰模式的主要优点在于可以提供比继承更多的灵活性，可以通过一种动态的 方式来扩展一个对象的功能，并通过使用不同的具体装饰类以及这些装饰类的 排列组合，可以创造出很多不同行为的组合，而且具体构件类与具体装饰类可 以独立变化，用户可以根据需要增加新的具体构件类和具体装饰类；其主要缺 点在于使用装饰模式进行系统设计时将产生很多小对象，而且装饰模式比继承 更加易于出错，排错也很困难，对于多次装饰的对象，调试时寻找错误可能需 要逐级排查，较为烦琐。
* 装饰模式适用情况包括：在不影响其他对象的情况下，以动态、透明的方式给 单个对象添加职责；需要动态地给一个对象增加功能，这些功能也可以动态地 被撤销；当不能采用继承的方式对系统进行扩充或者采用继承不利于系统扩展 和维护时。
* 装饰模式可分为透明装饰模式和半透明装饰模式：在透明装饰模式中，要求客 户端完全针对抽象编程，装饰模式的透明性要求客户端程序不应该声明具体构 件类型和具体装饰类型，而应该全部声明为抽象构件类型；半透明装饰模式允 许用户在客户端声明具体装饰者类型的对象，调用在具体装饰者中新增的方法。