**[设计模式七大原则(C++描述)](https://my.oschina.net/u/4414584/blog/3470426" \t "_blank)**

**设计模式的七大原则**

**1.单一职责原则**

**2.开放-封闭原则**

**3.依赖倒置原则**

**4.里氏替换原则(LSP)**

**5.接口隔离原则**

**6.迪米特原则(最少知道原则)**

**7.合成复用原则**

**1.单一职责原则**

**准确解析:就一个类而言,应该仅有一个引起它变化的原因**

当一个类职责变化时不会导致另一个类职责的变化.

优点:可以降低类的复杂度,提高可读性

**2.开放-封闭原则**

**准确解析:软件实体(类,模板,函数等等)应该可以扩展,但不可修改**

开闭原则是面对对象设计的核心所在;开放人员应该仅对程序中呈现出频繁**变化** 的那些部分做出抽象.

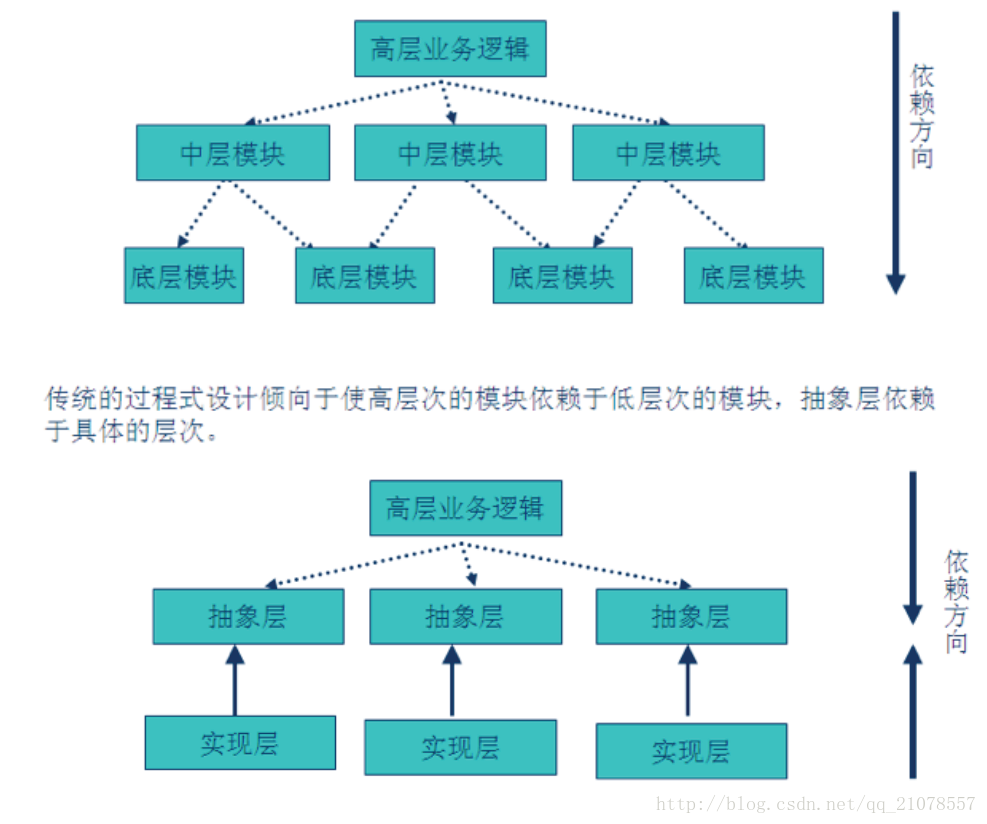
**3.依赖倒置原则**

**准确解析:A.高层模板(稳定)不应该依赖底层模板(变化).两个都应该依赖抽象(稳定)**

**B.抽象(稳定)不应该依赖实现细节(变化).细节(变化)应该依赖抽象(稳定).**

不论变化还是稳定都应该依赖于稳定

**说白了:要面对接口编程,不要对实现编程.**



#include<iostream>

class Book

{

public:void look()

{

....

}

.....

}

class Man

{

puclic:void Action(Book book)

{

book.look();

}

....

}

int main()

{

Man man=new Man();

Book book=new book();

Man->Action(book);

....

}

上面显示的是人看书的行为

那么假设现有我想要人进行看视频行为,视频类的代码如下:

class Video

{

public:void Video()

{

....

}

.....

}

那么我不仅要对人这个类中修改,还有对主函数的代码进行修改;如果有大量的需要的话,这个修改过程将会变得非常痛苦,因为书和人的耦合度太高.

接下来使用依赖倒置原则来会解决当前的痛苦,能够降低书和人的耦合度

书和视频我们当作一个可以看的东西ILOOK作为接口类,然后书和视频继承这个类

class ILOOK

{

public:virtual void look()=0;

}

class Bookpublic ILOOk

{

public:void look()

{

....

}

.....

}

class Video:public ILOOk

{

public:void look()

{

....

}

.....

}

class Man

{

puclic:void Action(ILOOK ilook)

{

ilook.look();

}

....

}

int main()

{

Man man=new Man();

ILOOK ilook=new book();

Man->Action(ilook);

ILOOK ilook2=new video();

Man->Action(ilook2);

....

}

这样就实现了简单的依赖倒置,人依赖于ILOOK这个类,并且书和视频也都依赖于ILook(即高层和底层都应该依赖抽象

这便是一个简单的面对接口编程.

这个依赖倒置原则将会贯串于所有设计模式,所以对于这个原则一定要有清晰的认识

**4.里氏替换原则(LSP)**

**准确解析:子类型必须能够替换掉它们的父类型**

**说白了就是一种IS-A的另一种表达**

比如说:鸟是一个父类,有 fly()这个虚函数,燕子是一个鸟,因为它能够飞,所以它可以继承鸟类;

企鹅不能飞,所以它不能继承鸟类,即使他在生物学上是鸟类,但它在编程世界中不能够继承鸟类

**这里说出LSP的一个特点:只有当子类可以替换掉父类,软件单位的功能不受影响时,父类才能够被复用,而子类也能够在父类的基础上增加新的行为**

通俗来说:子类可以扩展父类的功能,但不能改变父类原来的功能。

**包括4层含义:1.子类可以实现父类的抽象方法,但不能覆盖父类的非抽象方法。**

**2.子类中可以增加自己特有的方法。**

**3.当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。**

**4.当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。**

4种含义不展开讲,但用下面的一个例子来简单说明

#include<iostream>

class A

{

public:

int fun1(int a, int b) {

return a - b;

}

};

class B :public A

{

public:

int fun1(int a, int b) {

return a + b;

}

int fun2(int a, int b)

{

return fun1(a, b)-100; //想要a-b-100,但现实是a+b-100

}

};

int main()

{

int a = 100, b = 20;

B\* m\_b=new B();

std::cout << m\_b->fun2(a, b) << std::endl;

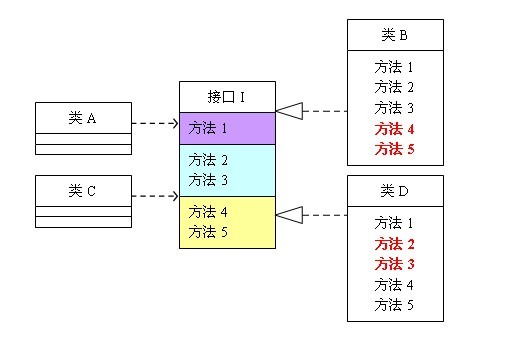
}

上面显示的结果会是20,因为B类中的fun1()覆盖到了A类中的fun1();所以fun2()中调用的是B类的fun1(),这便违反了里氏替换原则

**不遵循里氏替换原则的后果是:出问题的概率会大大提高**

**5.接口隔离原则**

**准确解释:不应该强迫客户程序依赖他们不用的方法;接口应该小而完备**



class I

{

public:

void method1()=0;

void method2()=0;

void method3()=0;

void method4()=0;

void method5()=0;

}

class A

{

public:

void depend1(I i)

{

i.method1();

}

void depend2(I i)

{

i.method2();

}

void depend3(I i)

{

i.method3();

}

}

class B:public I

{

public:

void method1()

{

std::cout<<"B实现方法1"<<std::endl;

}

void method2()

{

std::cout<<"B实现方法2"<<std::endl;

}

void method3()

{

std::cout<<"B实现方法3"<<std::endl;

}

//B类种方法4和5不是必须的

//但方法4和5因为继承的原因仍让需要空实现

void method4(){}

void method5(){}

}

class C

{

public:

void depend1(I i)

{

i.method1();

}

void depend2(I i)

{

i.method4();

}

void depend3(I i)

{

i.method5();

}

}

class D:public I

{

public:

void method1()

{

std::cout<<"B实现方法1"<<std::endl;

}

void method4()

{

std::cout<<"B实现方法4"<<std::endl;

}

void method5()

{

std::cout<<"B实现方法4"<<std::endl;

}

//B类种方法2和3不是必须的

//但方法2和3因为继承的原因仍让需要空实现

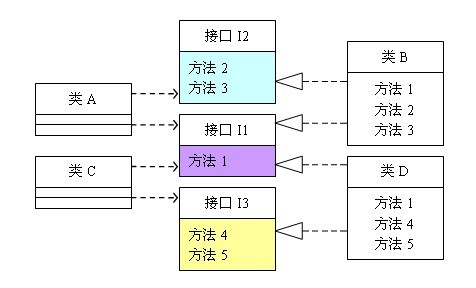
void method2(){}

void method3(){}

}

**上面便没有使用接口隔离原则**

**下面便使用了接口隔离,所以一些无关的方法就可以不用去实现**



class I1

{

public:

void method1()=0;

}

class I2

{

public:

void method2()=0;

void method3()=0;

}

class I3

{

public:

void method4()=0;

void method5()=0;

}

class A

{

public:

void depend1(I1 i)

{

i1.method1();

}

void depend2(I2 i)

{

i2.method2();

}

void depend3(I2 i)

{

i2.method3();

}

}

class B:public I1,public I2

{

public:

void method1()

{

std::cout<<"B实现I1方法1"<<std::endl;

}

void method2()

{

std::cout<<"B实现I2方法2"<<std::endl;

}

void method3()

{

std::cout<<"B实现I2方法3"<<std::endl;

}

}

class C

{

public:

void depend1(I1 i)

{

i1.method1();

}

void depend2(I2 i)

{

i3.method4();

}

void depend3(I2 i)

{

i3.method5();

}

}

class D:public I1,public I3

{

public:

void method1()

{

std::cout<<"B实现I1方法1"<<std::endl;

}

void method4()

{

std::cout<<"B实现I3方法4"<<std::endl;

}

void method5()

{

std::cout<<"B实现I3方法4"<<std::endl;

}

}

**使用接口隔离原则时应注意:**

**1.接口尽量小，但是要有限度。如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度。**

**2.为依赖接口的类定制服务，只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。**

**3.提高内聚，减少对外交互。使接口用最少的方法去完成最多的事情。**

**这个原则可以在实践多花时间思考,才可以准确地使用它**

**6.迪米特原则(最少知道原则)**

**准确解释:一个对象应该对其他对象保持最少的了解**

因为类之间的关系最紧密,耦合度越高,一个类变化时对另一个类的影响也大

我们使用迪米特原则就是要降低类之间的耦合度

**C++中一个重要的特性:高内聚,低耦合.高内聚,低耦合.高内聚,低耦合.(重要的事情说三遍)**

#include<iostream>

#include<list>

#include<string>

class Employee

{

private:

std::string m\_id;

public:

Employee(){}

Employee(std::string id) :m\_id(id) {}

std::string get\_id()

{

return m\_id;

}

};

class SubEmployee

{

private:

std::string m\_id;

public:

SubEmployee() {

}

SubEmployee(std::string id) :m\_id(id) {}

std::string get\_id()

{

return m\_id;

}

};

class SubCompanyManager

{

public:

std::list<SubEmployee> getAllEmployee()

{

std::list<SubEmployee> list(100);

for (int i = 0; i < 100; i++)

{

SubEmployee emp("分公司" + std::to\_string(i));

list.push\_back(emp);

}

return list;

}

};

class CompanyManager

{

public:

std::list<Employee> getAllEmployee()

{

std::list<Employee> list(30);

for (int i = 0; i < 30; i++)

{

Employee emp("总公司"+std::to\_string(i));

list.push\_back(emp);

}

return list;

}

void printALLEmployee(SubCompanyManager sub)

{

std::list<SubEmployee> list1(100);

list1 = sub.getAllEmployee();

std::list<SubEmployee>::iterator itor= list1.begin();

for (; itor != list1.end(); itor++)

{

std::cout << itor->get\_id();

}

std::list<Employee> list2(30);

list2= getAllEmployee();

std::list<Employee>::iterator itor2 = list2.begin();

for (; itor2 != list2.end(); itor2++)

{

std::cout << itor2->get\_id();

}

}

};

int main()

{

CompanyManager\* e = new CompanyManager();

SubCompanyManager s;

e->printALLEmployee(s);

system("pause");

return 0;

}

**上面的代码违反了迪米特原则**

根据迪米特法则，只与直接的朋友发生通信，而SubEmployee类并不是CompanyManager类的直接朋友（以局部变量出现的耦合不属于直接朋友），从逻辑上讲总公司只与他的分公司耦合就行了，与分公司的员工并没有任何联系，这样设计显然是增加了不必要的耦合。

|  |
| --- |
| class SubCompanyManager  {  public:  std::list<SubEmployee> getAllEmployee()  {  std::list<SubEmployee> list(100);  for (int i = 0; i < 100; i++)  {  SubEmployee emp("分公司" + std::to\_string(i));  list.push\_back(emp);  }  return list;  }  void printALLEmployee()  {  std::list<SubEmployee> list = getAllEmployee();  std::list<SubEmployee>::iterator itor = list.begin();  for (; itor != list.end(); itor++)  {  std::cout << itor->get\_id();  }  }  };  class CompanyManager  {  public:  std::list<Employee> getAllEmployee()  {  std::list<Employee> list(30);  for (int i = 0; i < 30; i++)  {  Employee emp("总公司" + std::to\_string(i));  list.push\_back(emp);  }  return list;  }  void printALLEmployee(SubCompanyManager sub)  {  sub.printALLEmployee();  std::list<Employee> list2(30);  list2 = getAllEmployee();  std::list<Employee>::iterator itor2 = list2.begin();  for (; itor2 != list2.end(); itor2++)  {  std::cout << itor2->get\_id();  }  }  }; |

为分公司增加了打印人员ID的方法，总公司直接调用来打印，从而避免了与分公司的员工发生耦合。

##### 另外切记不要过分使用迪米特原则,否则会产生大量的这样的中介和传递类,

#### 7.合成复用原则

##### ****准确解析:尽量先使用组合后聚合等关联关系来实现,其次才考虑使用继承关系来实现****

继承复用:又称"白箱""复用,耦合度搞,不利于类的扩展和维护

组合或聚合复用:又称"黑箱"复用,耦合度低,灵活度高



上面的图使用继承复合产生了大量的子类,如何需要增加新的"动力源"或者"颜色" 都要修改源代码,因为耦合度高,这违背了开闭原则

如果改为组合或聚合复用就可以很好的解决上述问题,如下图所示



#### 七点原则总结

##### 单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。****而开闭原则是总纲，他告诉我们要对扩展开放，对修改关闭****。合成复用原则告诉我们要优先使用组合或者聚合关系复用，少用继承关系复用。

##### 我们在实践时应该根据实际情况灵活使运用,才能达到良好的设计

参考博客:<html> [http://www.uml.org.cn/sjms/201211023.asp#2](https://www.oschina.net/action/GoToLink?url=http%3A%2F%2Fwww.uml.org.cn%2Fsjms%2F201211023.asp%232)