**UML工具：**

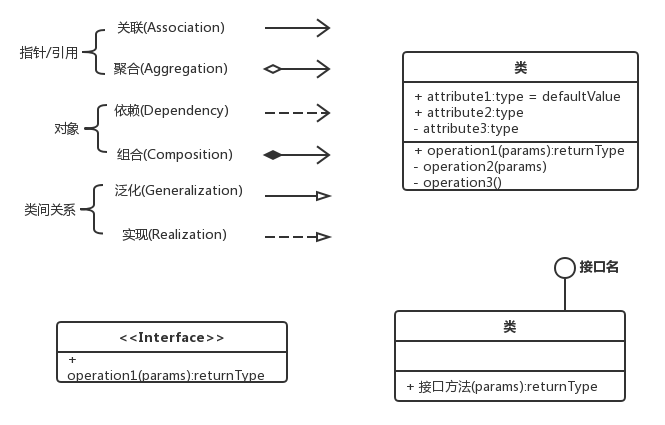
1、Process On 在线

2、Doxygen + Graphviz

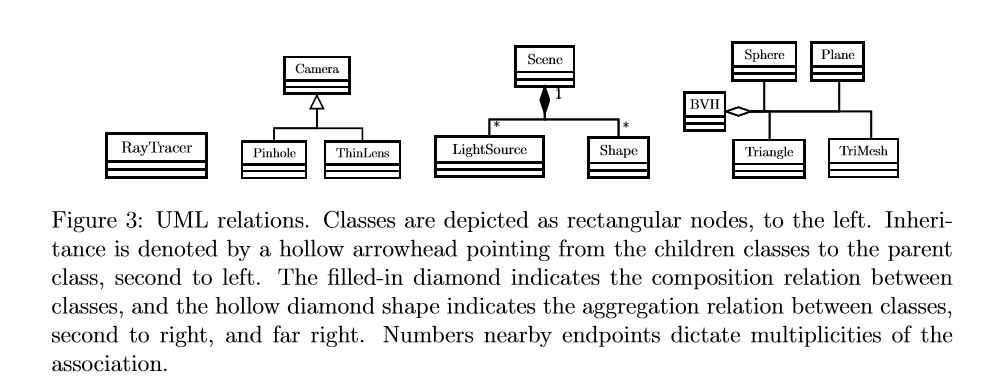
<http://www.360doc.com/content/12/0812/14/7851074_229747305.shtml>

3、Design Pattern 在线文档

<https://www.oodesign.com/>



<https://blog.csdn.net/guo13313/article/details/79485821>



原

**C++基础——用C++实例理解UML类图**

2015年11月01日 11:54:04 [Inside\_Zhang](https://me.csdn.net/lanchunhui" \t "_blank) 阅读数：10208

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/lanchunhui/article/details/49557083

[类展示](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%B1%BB%E5%B1%95%E7%A4%BA" \t "_self)

[基本概念及术语](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E5%9F%BA%E6%9C%AC%E6%A6%82%E5%BF%B5%E5%8F%8A%E6%9C%AF%E8%AF%AD" \t "_self)

[可见性visibility](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E5%8F%AF%E8%A7%81%E6%80%A7visibility" \t "_self)

[参数的方向parameter direction](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E5%8F%82%E6%95%B0%E7%9A%84%E6%96%B9%E5%90%91parameter-direction" \t "_self)

[类成员变量或者函数的类型](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%B1%BB%E6%88%90%E5%91%98%E5%8F%98%E9%87%8F%E6%88%96%E8%80%85%E5%87%BD%E6%95%B0%E7%9A%84%E7%B1%BB%E5%9E%8B" \t "_self)

[类关系](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%B1%BB%E5%85%B3%E7%B3%BB" \t "_self)

[关系的多重性multiplicity](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E5%85%B3%E7%B3%BB%E7%9A%84%E5%A4%9A%E9%87%8D%E6%80%A7multiplicity" \t "_self)

[类关系](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%B1%BB%E5%85%B3%E7%B3%BB-1" \t "_self)

[assocationkonws a](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "assocationkonws-a" \t "_self)

[dependency](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "dependency" \t "_self)

[Aggregation](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "aggregation" \t "_self)

[Composition](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "composition" \t "_self)

[继承泛化](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%BB%A7%E6%89%BF%E6%B3%9B%E5%8C%96" \t "_self)

[类模板](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "%E7%B1%BB%E6%A8%A1%E6%9D%BF" \t "_self)

[References](https://blog.csdn.net/lanchunhui/article/details/49557083" \l "references" \t "_self)

本文包括以下内容：

类间存在哪几种常见关系？

它们之间的区别和联系是什么？

如何在代码中反映类间的关系？

如何理解 **IN/OUT** mode型的参数？

**类展示**

class Circle

{

private:

double radius\_;

Point center\_;

public:

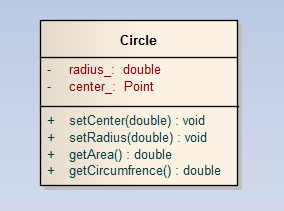
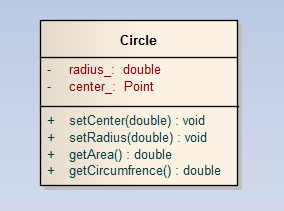
void setRadius(double \_radius);

void setCenter(Point \_center);

double getArea() const;

double getCircumfrence() const;

}



**基本概念及术语**

**可见性（visibility）**

‘+’ public

‘-’ private

‘#’ protected

**参数的方向（parameter direction）**

‘in’：用于输入的参数，get the value

‘out’：用于输出的参数， set the value

‘inout’：既可作为输入又可作为输出， get the value and set the value

这种参数形式常见于，这样一种情况，传递的是一个指针变量（如果传递的是一个既非指针，也非引用的变量，就纯做输入（in）了），既然是指针变量，便可提领指针，修改其指向的内容，既具备提供输入，又具有修改原值 的双重属性。

举例如下：

对一个接口作如下documentation：

Prototype: ULONG GetActivationState( ULONG \* pActivationState );

Parameters

Type: ULONG\*

Variable: pActivationState

Mode: IN/OUT

客户端代码

ULONG activationState = 1;

ULONG result = GetActivationState(&activationState);

既然提供的是变量的地址，便可轻易地get the value 和 set the value。 所以根据GetActivationState(ULONG\*) 参数的IN/OUT状态，推测其中的代码如下：

ULONG GetActivationState()

{

if (\*activationState == 1) *// 可读属性，in*

{

*// TODO*

*// 对变量值进行修改，例如，改为0*

\*activationState = 0; *// 可写属性，out*

}

return \*activationState;

}

**类成员（变量或者函数）的类型**

静态成员，下划线表示

纯虚函数，斜体

**类关系**

Assocation (knows a)

Dependency (uses a)

Composition (has a)

Aggregation (has a)

Inheritance (is a)

Class template

**关系的多重性（multiplicity）**

‘0..1’， 无实例，或者1个实例

‘1’，只有一个实例

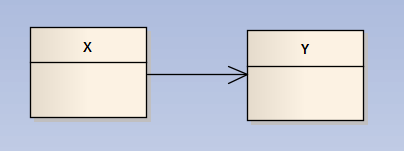
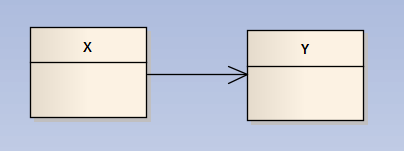
‘0..\*’，

‘1..\*’

**类关系**

**关联assocation（konws a）**

一个对象知道另一个对象的存在，该对象持有另一个对象的指针或者引用。



class X

{

public:

X(Y\* \_ptrY):ptrY\_(\_ptrY) {}

void setY(Y \*y) { ptrY\_ = y;}

void f() { ptrY\_->foo();}

private/public:

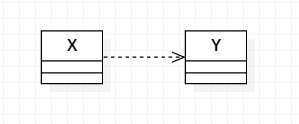
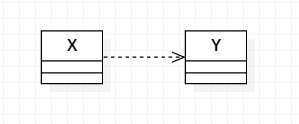
Y\* ptrY\_; *// X类持有Y的一个指针，可据此调用Y中的成员方法*

}

**依赖dependency**

当类Y与类X彼此依赖，而类Y是类X成员函数的一个参数，或者X中成员函数的一个局部变量。

本质上，这是因为类Y用到了类X的对象，或者类X的成员（即使是通过指针或引用来调用，也必须先有类X的定义）。



class X

{

...

void f1(Y y) { ...; y.foo(); }

void f2(Y& y) { ...; y.foo(); }

void f3(Y\* y) { ...; y->foo(); }

void f4() { ...; Y y; y.foo(); }

void f5() { Y::staticFoo();}

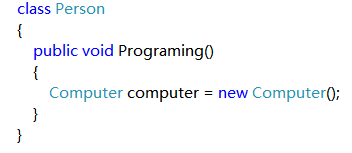
...

}

**依赖关系的三种表现形式：**

1.Computer类是public的，Person类可以调用它。

2.Computer类是Person类中某个方法的局部变量，则Person类可以调用它。代码如下：

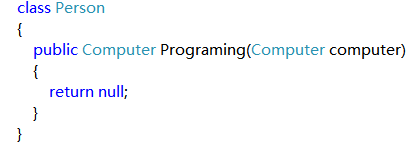


Person有一个Programing方法，Computer类作为该方法的变量来使用。

注意Computer类的生命周期，当Programing方法被调用的时候，才被实例化。

持有Computer类的是Person类的一个方法，而不是Person类，这点是最重要的。

3.Computer类作为Person类中某个方法的参数或返回值。



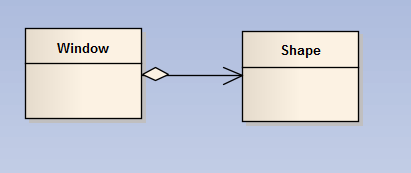
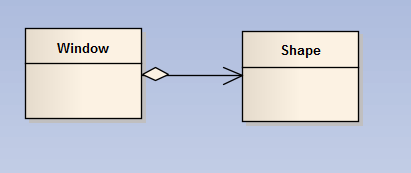
Computer类被Person类的一个方法所持有，生命周期随着方法执行结束而结束。

**在依赖关系中，必须使用这三种方法之一。**

**聚合Aggregation**

聚类发生的场景是：一个类是另一个类的集合或者容器，但被包含的类与容器本身并不具备相同的生命期，也就是容器销毁时，其包含的内容未必。

聚合与关联从形式上看相同，只是聚合有逻辑上的整体与部分的关系。关联（association）与聚合（aggregation）的不同之处在于，关联关系不存在一个明确的容器包含另外一个类。比如汽车和轮胎的关系，就是一种聚合关系，汽车包含着轮胎的集合，汽车的销毁，轮胎仍然可以存在。



class Window

{

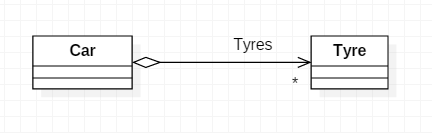
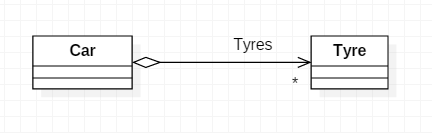
public:

//...

private:

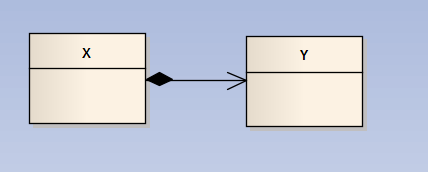
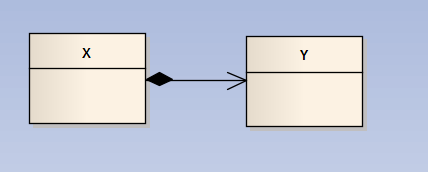
vector<Shape\*> ptrShapes;

}



**组合Composition**

组合（composition）是聚合（Aggregation）的增强版。组合和聚合的根本不同在于类与其所持有元素的是否具有相同的生命期，要求必须相同是一种组合关系。



class Circle

{

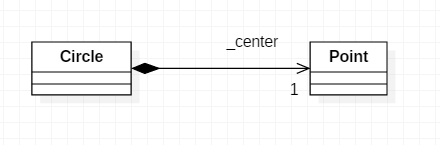
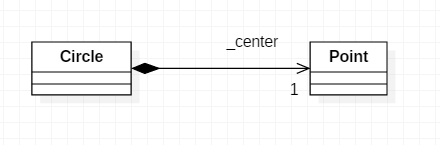
private:

...

Point center\_;

...

}



class X

{

...

Y y; // composition, 相同的生命期，

// 如果是引用或者指针类型，则是关联类型，是一种弱联系

Y b[10]; // 0...10, composition

}

class X

{

X() { a = new Y[10]; }

~X() { delete[] a;}

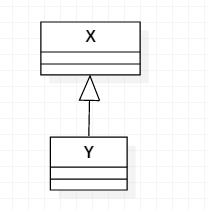
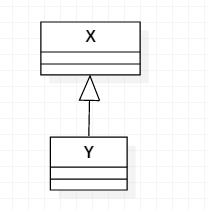
...

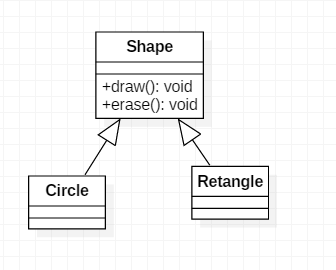
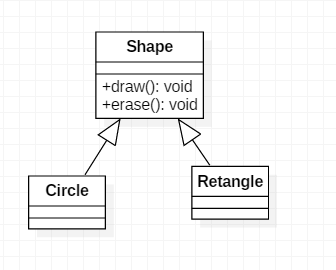
Y \*a; // 0...10, composition，X对象析构时，必须对a进行释放

}

**继承（泛化、实现）**

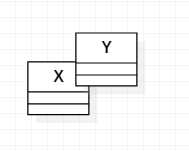
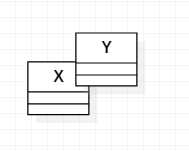
类间的继承关系表达的是一种**derived class B is a base class A**。





**类模板**

类模板意味着泛型类。



template<class T>

class X

{...};

X<Y> a;

**除组合关系外，其他类关系，都是类A持有类B的指针或引用，而不是持有类B的对象。**

### 聚合与组合[#](https://www.cnblogs.com/jiqing9006/p/5915023.html" \l "1624281931)

（1）聚合与组合都是一种结合关系，只是额外具有整体-部分的意涵。

（2）部件的生命周期不同

聚合关系中，整件不会拥有部件的生命周期，所以整件删除时，部件不会被删除。再者，多个整件可以共享同一个部件。  
组合关系中，整件拥有部件的生命周期，所以整件删除时，部件一定会跟着删除。而且，多个整件不可以同时间共享同一个部件。

（3）聚合关系是“has-a”关系，组合关系是“contains-a”关系。

### 关联和聚合[#](https://www.cnblogs.com/jiqing9006/p/5915023.html" \l "2043209151)

（1）表现在代码层面，和关联关系是一致的，只能从语义级别来区分。

（2）关联和聚合的区别主要在语义上，关联的两个对象之间一般是平等的，例如你是我的朋友，聚合则一般不是平等的。

（3）关联是一种结构化的关系，指一种对象和另一种对象有联系。

（4）关联和聚合是视问题域而定的，例如在关心汽车的领域里，轮胎是一定要组合在汽车类中的，因为它离开了汽车就没有意义了。但是在卖轮胎的店铺业务里，就算轮胎离开了汽车，它也是有意义的，这就可以用聚合了。

### 关联和依赖[#](https://www.cnblogs.com/jiqing9006/p/5915023.html" \l "1024436363)

（1）关联关系中，体现的是两个类、或者类与接口之间语义级别的一种强依赖关系，比如我和我的朋友；这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。

（2）依赖关系中，可以简单的理解，就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是B类的变化会影响到A。比如某人要过河，需要借用一条船，此时人与船之间的关系就是依赖；表现在代码层面，为类B作为参数被类A在某个method方法中使用；

**References**

[1] [UML Class Diagram Explained With C++ samples](https://cppcodetips.wordpress.com/2013/12/23/uml-class-diagram-explained-with-c-samples/" \t "_blank)

[2] [IN/OUT Parameters and how to work with them in C++](http://stackoverflow.com/questions/6900035/in-out-parameters-and-how-to-work-with-them-in-c" \t "_blank)

[3] <https://www.cnblogs.com/jiqing9006/p/5915023.html>

**[Java依赖、关联、聚合和组合之间区别](https://www.cnblogs.com/twuxian/p/11325178.html)**

**依赖(Dependency)关系**是类与类之间的联接。依赖关系表示一个类依赖于另一个类的定义。例如，一个人(Person)可以买车(car)和房子(House)，Person类依赖于Car类和House类的定义，因为Person类引用了Car和House。与关联不同的是，Person类里并没有Car和House类型的属性，Car和House的实例是以参量的方式传入到buy()方法中去的。一般而言，依赖关系在Java语言中体现为局域变量、方法的形参，或者对静态方法的调用。   
  
**关联(Association）关系**是类与类之间的联接，它使一个类知道另一个类的属性和方法。关联可以是双向的，也可以是单向的。在Java语言中，关联关系一般使用成员变量来实现。   
  
**聚合(Aggregation)关系**是关联关系的一种，是强的关联关系。聚合是整体和个体之间的关系。例如，汽车类与引擎类、轮胎类，以及其它的零件类之间的关系便整体和个体的关系。与关联关系一样，聚合关系也是通过实例变量实现的。但是关联关系所涉及的两个类是处在同一层次上的，而在聚合关系中，两个类是处在不平等层次上的，一个代表整体，另一个代表部分。   
  
**组合(Composition)关系**是关联关系的一种，是比聚合关系强的关系。它要求普通的聚合关系中代表整体的对象负责代表部分对象的生命周期，组合关系是不能共享的。代表整体的对象需要负责保持部分对象和存活，在一些情况下将负责代表部分的对象湮灭掉。代表整体的对象可以将代表部分的对象传递给另一个对象，由后者负责此对象的生命周期。换言之，代表部分的对象在每一个时刻只能与一个对象发生组合关系，由后者排他地负责生命周期。部分和整体的生命周期一样。  
  
  
  
以上关系的耦合度依次增强（关于耦合度的概念将在以后具体讨论，这里可以暂时理解为当一个类发生变更时，对其他类造成的影响程度，影响越小则耦合度越弱，影响越大耦合度越强）。由定义我们已经知道，依赖关系实际上是一种比较弱的关联，聚合是一种比较强的关联，而组合则是一种更强的关联，所以笼统的来区分的话，实际上这四种关系、都是关联关系。   
  
依赖关系比较好区分，它是耦合度最弱的一种，在java中表现为局域变量、方法的形参，或者对静态方法的调用，如下面的例子：Driver类依赖于Car类，Driver的三个方法分别演示了依赖关系的三种不同形式。

1 import java.util.\*;

2

3 public class Test {

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

5 Driver dr = new Driver();

6 Car car = new Car();

7 dr.driver(car);

8

9 }

10

11 }

12 class Car{

13 public static void run() {

14 System.out.println("汽车在跑");

15 }

16 }

17 class Driver{

18 public void driver(Car car) {

19 //使用形参方式发生依赖关系

20 car.run();

21 }

22 public void driver() {

23 //使用局部变量发生依赖关系

24 Car car = new Car();

25 car.run();

26 }

27 public void driver1() {

28 //使用静态方法发生依赖关系

29 Car.run();

30 }

31 }

关联关系在java中一般使用成员变量来实现，有时也用方法形参的形式实现。依然使用Driver和Car的例子，使用方法参数形式可以表示依赖关系，也可以表示关联关系，毕竟我们无法在程序中太准确的表达语义。在本例中，使用成员变量表达这个意思：车是我自己的车，我“拥有”这个车。使用方法参数表达：车不是我的，我只是个司机，别人给我什么车我就开什么车，我使用这个车。

1 class Driver {

2 //使用成员变量形式实现关联

3 Car mycar;

4 public void drive(){

5 mycar.run();

6 }

7 ...

8 //使用方法参数形式实现关联

9 public void drive(Car car){

10 car.run();

11 }

12 }

假如给上面代码赋予如下语义：车是一辆私家车，是司机财产的一部分。则相同的代码即表示聚合关系了。聚合关系一般使用setter方法给成员变量赋值。   
  
假如赋予如下语义：车是司机的必须有的财产，要想成为一个司机必须要先有辆车，车要是没了，司机也不想活了。而且司机要是不干司机了，这个车就砸了，别人谁也别想用。那就表示组合关系了。一般来说，为了表示组合关系，常常会使用构造方法来达到初始化的目的，例如上例中，加上一个以Car为参数的构造方法。

1 public Driver(Car car){

2 mycar = car;

3 }

所以，关联、聚合、组合只能配合语义，结合上下文才能够判断出来，而只给出一段代码让我们判断是关联，聚合，还是组合关系，则是无法判断的。