面向对象

作者：传智播客汤凤坡老师

参考：PHP官方文档、CSDN博客相关文章、博客园相关文章、脚本之家相关文章、开源中国相关文章、传智播客姚长江老师、罗弟华老师、蒲二飞老师授课笔记。对以上网站及个人表示感谢！

1、前言

在面向对象的程序设计（英语：Object-oriented programming，缩写：OOP）中，对象是一个由信息及对信息进行处理的描述所组成的整体，是对现实世界的抽象。

编程中的对象来源于现实生活，是现实生活中对象的抽象。

现实生活中，处处都是对象，比如人、汽车、动物、电脑、鼠标等等。而不同的对象又有不同的属性和行为，比如人有身高、体重、年龄、性别等属性，还有能说话、能够行走、能够谈恋爱等行为；电脑有品牌、型号等属性，还有能发出声音、能显示图像等行为。

编程中，只不过是用代码的方式表示这些对象，用代码的方式表示对象的特点和行为罢了。

> 要完全理解面向对象编程不是一件容易的事，对于初学者，建议先从语法角度入手，掌握一定的编程能力之后，自然而然会明白面向对象编程思想。

2、类与对象

比如说，我喜欢大眼睛、长头发、大长腿、会洗衣服、能做饭，能生孩子的女孩，比如有孙莉、冰冰。

这句话中的孙莉、冰冰就是实实在在的人，即对象，而大眼睛、长头发、大长腿就是对象的属性；会洗衣服、能做饭、能生孩子就是对象的行为。

在代码中，我们用 **类** 来描述对象的属性和行为；符合这个类的具体事物就是该类的对象。下面我们看一下，如何定义或声明一个类。

```php
//定义类
class Girl {
    //定义对象的属性
    public $eyes = '大'; //大眼睛
    public $hair = '长'; //头发长
    public $legs = '长'; //腿长

    //定义对象的行为
    public function wash(){
        echo '我洗得一手好衣服';
    }
    public function cook(){
        return '我做得一手好饭';
    }
    public function childbirth(){
        echo '我生得一手好孩子';
```

```
    }
}
```

从代码中，我们可以看出，使用 *class* 关键字定义类，后面的 *Girl* 是类的名字，后面直接写大括号，大括号里面的内容就是类的属性和行为。

> 属性可以用一个变量表示，但行为需要用一个函数来表示，在面向对象编程中，我们把类里面的函数称之为**方法**。因为属性和方法都属于类的成员，所以我们也可以把属性叫做成员属性，把方法叫做成员方法。

值得一提的是，类里面的成员属性和成员方法前面都写了一个 *public* ，它叫做成员的访问控制符，关于这一点，我们后续会学习。

学习了如何定义一个类，那么如何得到对象呢？我们看下面的代码：

```php
$sunli = new Girl;
$bingbing = new Girl();
```

代码中，使用 *new* 这个关键字来 *new* 前面定义的类名，这样就得到了 *Girl* 类的对象了，我们把得到的对象赋值给前面的变量，那么这个变量就是对象了。

> 在编程中，通常把用类创建对象的过程称为实例化 ，所以上面得到对象的过程也称之为实例化类得到对象。

> 在实例化对象的时候，**类名后面的小括号**可以写，也可以不写，但是**建议写上**。

3、成员属性

成员属性由访问控制符（如 *public* ）和普通的变量声明来组成，而且必须加访问控制符。属性中的变量可以初始化，但是初始化的值必须是常数，这里的常数是指 PHP 脚本在编译阶段时就可以得到其值，而不依赖于运行时的信息才能求值。

下面我们看一下正确的成员属性声明方式：

```php
define('ABC', 'hello');
class Test {
    public $name;
    public $age = 20;
    public $xx = ABC; //值前面定义的常量
    public $yy = array('apple', 'banana');
    public $var4 = 1+2; //PHP7中正确，PHP5中错误
    public $var5 = 'hello'.'world'; //PHP7中正确，PHP5中错误
    //在 PHP 5.3.0 及之后，下面的声明也正确
   public $zz = <<<'EOD'
hello world
EOD;
}
```

下面我们在来看一下错误的成员属性声明方式：

```php
class A {}
$x = 1;
class Test {
    $var0 = 123; //错误，因为前面没有 public 修饰
    public $var1 = new A(); //错误，值不是常数
    public $var2 = $x; //错误，值不是常数
    public $var3 = self::y(); //错误，值不是常数

    public static function y(){ }
}
```

4、成员方法

前面说，我们把类里面的函数称之为方法或成员方法。成员方法和函数的写法一致，可以有参数，也可以有返回值。和成员属性不同的是，成员方法前面不加 *public* 也是可以的，如果不加权限控制符 *public* ，那么该方法默认就是 *public* 修饰的，但是我们建议加上。

> 成员方法、成员属性的定义没有先后顺序，但是建议将所有的成员属性定义到成员方法之前。
>
> 成员方法中可以调用类中的其他成员， 关于如何调用，我们会在伪对象 *$this* 和静态成员讲解。

5、对象成员操作

首先明确一点，我们是通过类得到的对象，所以类里面的成员属性和成员方法，也可以叫做对象的成员属性和成员方法。本节，我们将讨论，在实例化对象之后，如何访问对象的成员。

> 到底是对象的成员还是类的成员，我们后续会讲，目前可以认为是对象的成员。

访问对象的成员，是通过操作符 `->` 来访问的，具体看代码：

```php
//定义类
class Girl {
    //定义对象的属性
    public $eyes = '大'; //大眼睛
    public $hair = '长'; //头发长
    public $legs = '长'; //腿长

    //定义对象的行为
    public function wash(){
        echo '我洗得一手好衣服';
    }
    public function cook($name){
        return $name . '做得一手好饭';
    }
    public function childbirth(){
        echo '我生得一手好孩子';
    }
}
```

```php
//实例化对象
$sunli = new Girl();
//输出对象的成员属性
echo $sunli->eyes;
//修改对象的成员属性
$sunli->hair = '短';
//删除对象的成员属性
unset($sunli->legs);
//当然，也可以为这个对象添加一个成员属性
$sunli->age = 20;
```

无论对 对象的成员属性进行何种操作，使用对象访问成员属性的语法是一样的，即 `对象->属性`

> 删除对象的属性 *legs* 并不是把 *Girl* 类里面的 *legs* 删除了，只是把 *sunli* 这个对象中的成员属性删除了。
>
> 同理，为 *sunli* 这个对象添加一个 *age* 属性，也只是给这个对象添加的，并不是给 *Girl* 类添加的
>
> 访问成员属性时，属性名前没有 *$* 符号

访问对象的成员方法，也是使用 `->` 这种语法，即 `对象->方法()` ，只不过像调用函数那样，后面必须加小括号。

```php
//调用对象的成员方法
$sunli->childbirth();
echo $sunli->cook('小明');
```

6、伪对象$this

前面我们讲到，成员方法中可以调用其他的成员（属性和方法），我们又说，类中的成员可以看做是对象的成员。那么在成员方法中如何访问其他的对象成员呢？答案就是使用 *$this*。

```php
class Person {
    public $name = '张三疯';
    public $age = 99;

    public function introduce(){
        echo '我的名字是' . $this->name; //调用成员属性
        echo '<br>' . $this->info(); //调用成员方法
    }

    public function info(){
        return '我的年龄是' . $this->age; //调用成员属性
    }
}
//实例化对象
$p = new Person();
$p->intro();
```

思考一下，在实例化对象后，是如何调用对象的成员的？答案是使用 `对象名->成员` 的方式。

上面代码中，在类里面调用对象的属性或方法，实际上也希望使用 `对象名->成员` 这种语法去调用，但是在调用这些属性或方法的时候，还没有实例化对象，所以只能用一个能够表示该类的对象的变量，这个变量就是 *$this*。反过来说， *$this* 能够表示该类的实例对象。

> *$this* 是在类中，表示该类的对象的一个变量，但是 *$this* 又不是一个具体的对象，所以叫做伪对象。
>
> 成员方法调用其他成员的时候，不分先后顺序，如 *introduce* 方法可以调用后面定义的 *info* 方法。

问题一：如果根据一个类实例化了多个对象， *$this* 表示哪个对象?

答：当前使用的是哪个对象，*$this* 就表示哪个对象。如下代码：

```php
class Person {
    public function test(){
        var_dump($this); //调用成员属性
    }
}
$p1 = new Person();
$p1->name = '111';
$p2 = new Person();
$p2->name = '222';

$p1->test(); //当前使用p1调用的test方法，所以test方法中的$this表示p1对象
$p2->test(); //当前使用p2调用的test方法，所以test方法中的$this表示p2对象
```

问题二：如何完成链式访问？例如类中有两个方法 *prev* 和 *next* ，实例化对象后，如何使用对象连续的调用这个两个方法呢?

答：前面的方法中返回 *$this* 即可。因为 *$this* 表示当前正在使用的对象，返回 *$this* 即返回该对象，这样我们就可以继续使用对象调用后面的方法了。代码如下：

```php
class Person {
    public function prev(){
        echo '我是前面的方法。';
        return $this; //把调用该方法的对象返回，以便继续调用其他方法
    }
    public function next(){
        echo '我是后面的方法。';
    }
}
$p = new Person();
$p->prev()->next();//调用prev方法后，得到返回值，返回值还是当前的$p对象，所以可以继续调用next方法
```

> 最后说明：*$this* 表示类的对象，它只能用于类的内部，或者更明确的说，只能用在类内部的方法中。

7、访问控制

对属性或方法的访问控制，是通过在前面添加关键字 *public*（公有），*protected*（受保护）或 *private*（私有）来实现的。

- 被定义为公有的类成员可以在任何地方（父类、子类、当前的类、类的外部）被访问。
- 被定义为受保护的类成员则可以被其自身以及其子类和父类访问。
- 被定义为私有的类成员则只能被其定义所在的类访问。

> 涉及到子类和父类的知识，我们将在继承中讲解。

```php
class Person {
    public $name = '张铁柱';
    protected $money = '1千万';
    private $lover = ['大美', '小美'];

    public function mySelf(){
        //类的内部，可以随意调用类中的成员，而不需要考虑这些成员的访问控制符是什么
        echo $this->name . '<br>';
        echo $this->money . '<br>';
        print_r($this->lover . '<br>');
        $this->myColor();
        $this->mySecret();
    }
    protected function myColor(){
        echo 'I am black';
    }
    private function mySecret(){
        echo "I'm afraid of mice";
    }
}
//实例化对象
$p = new Person();
echo $p->name; //没有问题，因为属性是公开的
echo $p->money; //报错，因为属性是受保护的
print_r($p->lover); //报错，因为属性是私有的
$p->mySelf(); //没有问题，因为方法是公开的
$p->myColor(); //报错，因为方法是受保护的
$p->mySecret(); //报错，因为方法是私有的
```

> 值得注意的是mySelf方法，根据这个方法中的内容，可以看出，类中的方法可以无限制的调用类的成员属性和成员方法，而不考虑被调用属性或被调用方法的访问控制符。

8、构造函数和析构函数

8.1 构造函数

构造函数也是类中的一个成员方法，只不过这个方法比较特别，特别之处在于下面两点：

- 构造函数的名字必须是 *__construct*
- 构造函数在实例化对象时自动调用，也就是说该方法会自动执行，而且是在实例化对象时最先执行的方法

下面用代码演示如何定义构造函数：

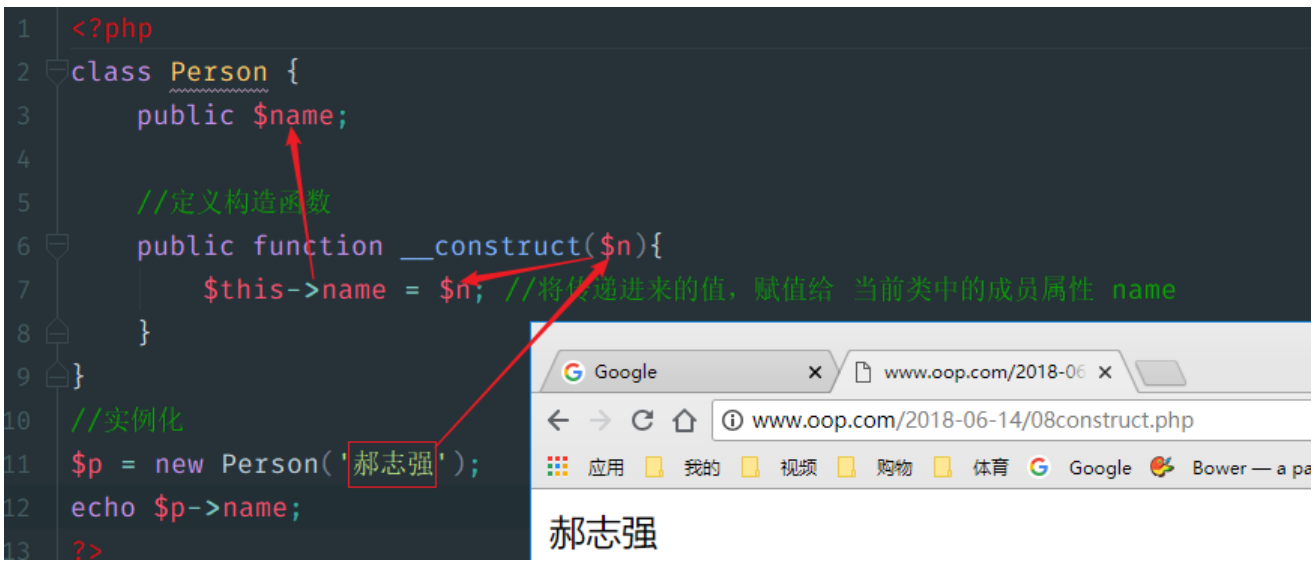
```php
class Person {
    //定义构造函数
    public function __construct(){
        echo 12345;
    }
}
$p = new Person(); //实例化的时候，构造函数自动调用，所以这行执行完，会输出12345
```

实例化的时候，自动调用构造函数，也就是说，构造函数必须是公开的，才能够在定义类之后实例化对象。代码如下：

```php
class Person {
    //定义构造函数，但是访问控制符为受保护的
    protected function __construct(){
        echo 12345;
    }
}
$p = new Person(); //实例化的时候，构造函数自动调用。因为构造函数为受保护的，所以这里报错。
```

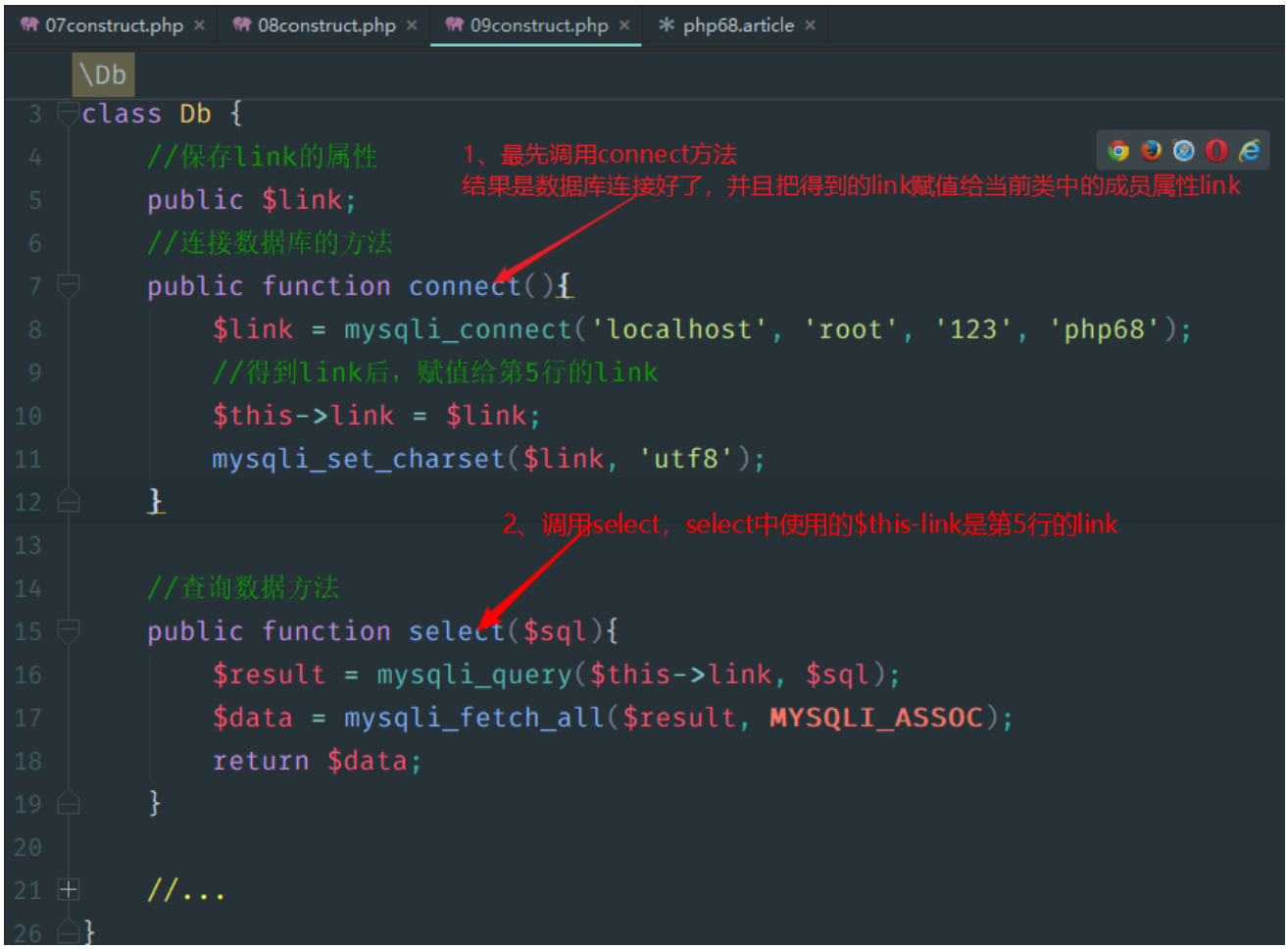
构造函数可以有参数，在实例化对象的时候，为构造函数传递实参。代码如下：

```php
class Person {
    public $name;
    //定义构造函数，但是访问控制符为受保护的
    public function __construct($n, $x, $y=5){
        $this->name = $n; //将张三疯赋值给第2行中的成员属性name
        echo $x + $y;
    }
}
$p = new Person('张三疯', 3); //输出 8
```



因为构造函数的特点，所以非常适合在使用对象之前做一些初始化工作。比如连接数据库：

初始代码：




```php
class Db {
    //保存link的属性
    public $link;
    //连接数据库的方法
    public function connect(){
        $link = mysqli_connect('localhost', 'root', '123', 'php68');
        //得到link后，赋值给第5行的link
        $this->link = $link;
        mysqli_set_charset($link, 'utf8');
    }

    //查询数据方法
    public function select($sql){
        $result = mysqli_query($this->link, $sql);
        $data = mysqli_fetch_all($result, MYSQLI_ASSOC);
        return $data;
    }

    //...
}
```

```php
//测试Db类 -- 查询article表的内容
$sql = "select id,title from article";
//实例化
$db = new Db();
//进行查询之前，必须先连接数据库
//$db->connect();
$data = $db->select($sql);
echo '<pre>';
print_r($data);
```

这时，需要手动调用connect方法，加入构造函数优化后的代码如下：

```php
class Mysql {
    private $link; //用于存放连接到数据库的资源标识符
    //构造函数
    public function __construct(){
        $this->connect();
```

```
    }
    public function connect() {
        $this->link = mysqli_connect('localhost', 'root', '123', 'itcast');
        mysqli_set_charset($this->link, 'utf8');
    }
    //查询方法
    public function select($sql){
        $result = mysqli_query($this->link, $sql);
        return mysqli_fetch_all($result, 1);
    }
}
//实例化对象
$mysql = new Mysql(); //实例化对象的时候，构造方法自动执行，所以数据库已经连接上了
$data = $mysql->select('select * from student'); //传递SQL语句，得到查询到的数据
```

执行过程（顺序）：

1、从18行开始，实例化了对象，实例化对象后，构造函数自动执行，构造函数中调用了connect方法，所以数据库连接好了，并且类中的成员属性 *$link* 也有值了。

2、第19行，直接调用select查询即可，因为数据库已经连接好了。

8.2 析构函数

析构函数：析构函数会在到某个对象的所有引用都被删除或者当对象被显式销毁时执行。

简单的说，析构函数会在对象即将消失的时候自动执行，析构函数一定是对象最后一个执行的方法，这个方法的名字必须是 *__destruct*。

那么对象什么时候消失呢？一是页面执行完毕，二是手动的使用unset销毁对象，三是设置对象的值为其他值。

```
class Person {
    public function __destruct()
    {
        echo '轻轻地我走了';
    }
}
$p = new Person();
$p = null; //设置变量p的值为null，则原来的对象消失，这里用unset($p)也可以。
echo '12345';
//执行结果：轻轻地我走了12345
```

和上面的代码不同，如果对变量 *$p* 没有进行任何操作，则在页面执行完毕后，对象才消失。代码如下：

```php
class Person {
    public function __destruct()
    {
        echo '轻轻地我走了';
    }
}
$p = new Person();
echo '12345';
//执行结果：12345轻轻地我走了
```

再来看一个例子：

```php
class Test {
    public $num;
    //构造函数
    public function __construct($n){
        $this->num = $n;
    }
    //析构函数
    public function __destruct(){
        echo $this->num;
    }
}
$t1 = new Test(1);
$t2 = new Test(2);
$t3 = new Test(3);
//页面执行完毕，释放对象
//输出结果 321
```

上面的例子体现了对象入栈和出栈的顺序，具体分析将在对象在内存中的表现形式一节讲解。

因为析构函数在对象释放时执行，所以经常用于释放一些额外的资源。比如数据库资源，代码如下：

```php
class Mysql {
    private $link; //用于存放连接到数据库的资源标识符
    //构造函数
    public function __construct() {
        $this->link = mysqli_connect('localhost', 'root', '123', 'itcast');
        mysqli_set_charset($this->link, 'utf8');
    }
    //查询方法
    public function select($sql){
        $result = mysqli_query($this->link, $sql);
        return mysqli_fetch_all($result, 1);
    }
    //析构方法
    public function __destruct(){
        mysqli_close($this->link); //对象销毁时，释放MySQL连接资源
```

```
    }
}
```

> 需要注意的是，析构函数不能有参数；试图用 *exit()* 阻止析构函数的执行是无效的，在析构函数中抛出异常将报错。

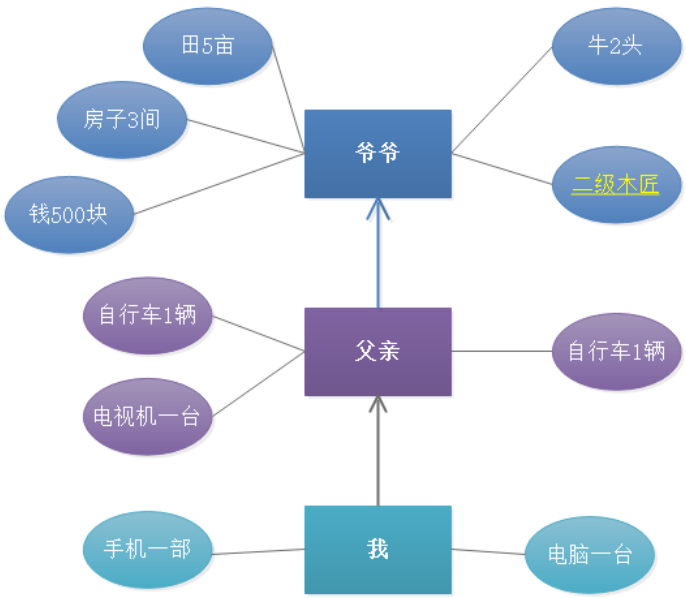
构造函数和析构函数有非常明显的特点，一是他们都是由两条下划线开头的，二是他们都是在特定情况下自动调用的。实际上符合这两个特点的方法还有很多，我们将这类方法统称为魔术方法。魔术方法都不能是静态方法。

> ps：为了使代码更加工整明了，我们自己定义的方法就不要使用两条下划线开头了。

9、继承

9.1 继承

继承是一个程序设计特性，犹如人类的继承，比如子承父业，其实和编程中的继承是一个意思。



上图中的”我“继承了”父亲“，”父亲“继承了”爷爷“。当继承关系存在时，”父亲“可以继承”爷爷“所有的财产和一门手艺。这样的话，“父亲”除了自己的财产以外，还拥有继承“爷爷”而来的所有的内容。同理，“我”除了拥有自己的东西以外，还会继承“父亲”和“爷爷”所有的内容。

PHP中的继承，至少会涉及到两个类，一个**父类**，一个**子类**。子类继承父类，语法如下：

```
//定义一个类，充当父类
class Animal {

}
//定义一个类，继承父类
class Birds extends Animal {

}
```

> 上述代码中的Birds类继承了Animal类。

这一节会涉及到几个和继承相关的概念：

- 继承：站在子类的角度，继承父类。
- 派生：站在父类的角度，派生子类。
- 扩展：站在父类的角度，扩展子类。
- 父类：被继承的类，也叫做上层类，或者基础类，或者基类。
- 子类：继承别的类的类，也叫做下层类，或者派生出来的类，或者后代类。

继承将会影响到类与类，对象与对象之间的关系。

比如，**当扩展一个类，子类就会继承父类所有的成员，但是由于屏蔽机制的原因，私有成员在子类中无法使用，所以也可以认为子类会继承父类所有公有的和受保护的成员。**

```php
class Dad {
    private $nickname = '老王';
    protected $money = 1000000;
    public $surname = '王';
    public function xx(){
        return '2级木匠';
    }
    protected function yy(){
        return '受保护的方法';
    }
    private function zz(){
        return '私有方法';
    }
}

class Child extends Dad {
    public function test(){
        echo $this->nickname; //Undefined property: Child::$nickname
        echo $this->money; //1000000
        echo $this->surname; //王
        var_dump(method_exists('Child', 'xx')); //true
        var_dump(method_exists('Child', 'yy')); //true
        var_dump(method_exists('Child', 'zz')); //true
        echo $this->xx(); //2级木匠
        echo $this->yy(); //受保护的方法
        echo $this->zz(); //Fatal error:  Uncaught Error: Call to private method Dad::zz() fro
    }
}
echo "<pre>";
$b1=new Child();
$b1->test();
```

> 上述代码中，子类Child继承了父类Dad，所以子类Child继承了Dad类所有的成员属性和成员方法。
>
> 但实际在子类中，能够调用的只有公开的和受保护的成员属性和成员方法。

继承对于功能的设计和抽象是非常有用的，而且对于类似的对象增加新功能就无须重新再写这些公用的功能。

9.2 方法重写

在继承链条上，如果父类和子类有同名的方法，而且同名方法的参数个数也一样，这种情况就叫做覆盖，**覆盖也叫做重写(override)**，即重新在子类中编写一个方法。下面我们就来讨论一下覆盖父类方法时，需要注意什么。

- 子类的方法名和父类的方法名一样，并且方法的参数个数要保持一致（构造方法除外）。
- 子类中重写的方法可见性不能小于父类的方法，包括构造方法。

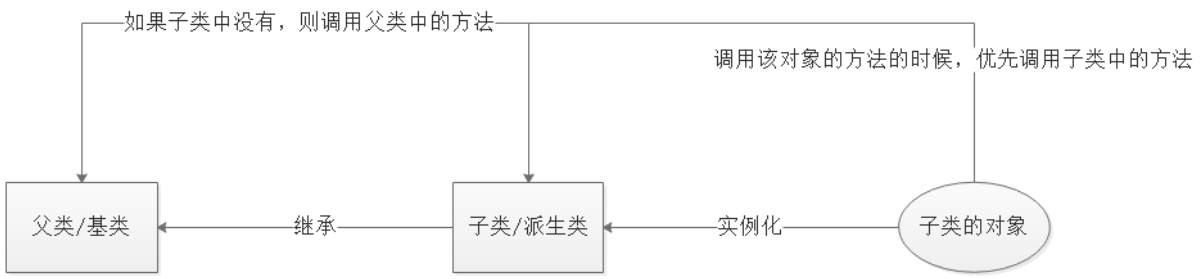
> 可见性指的的成员的修饰符，可见性： public > protected > private

下面两段代码演示了重写方法时的注意事项

```php
class Dad {
    public function t1($x, $y){}
    private function t2($x, $y){}
}
class Child extends Dad {
    //重写方法t1错误，参数个数必须和父类中t1方法一致，也就是必须是两个参数
    public function t1($x){}

    //重写方法t2，PHP5中报错，因为参数个数不一致；
    //重写方法t2，PHP7中不报错，因为父类的t2是私有的，子类不可见，所以谈不上重写，相当于父子两个t2
    public function t2($x){}
}
```

```php
class Dad {
    public function t1(){}
    protected function t2(){}
}
class Child extends Dad {
    protected function t1(){} //错误，父类方法可见性public，子类方法可见性只能是public
    private function t2(){} //错误，父类方法可见性protected，子类方法可见性应该为public或protected
}
```

在继承链条上，如何灵活调用父类或者子类的同名方法呢？看下面一张图：



具体看代码：

```php
class Dad {
    public function __construct(){
        echo 111;
    }
    public function t1(){
        echo 222;
    }
}
class Child extends Dad {
    public function __construct(){
        echo 333;
    }
    public function t1(){
        echo 444;
    }
}
$c = new Child();  //实例化的时候，Child类的构造方法自动执行，输出333
$c->t1(); //调用的是子类的t1，所以输出444
```

上面代码二中，父类的构造方法和 *t1* 方法就失去了它们的作用，直白的说就是白写了。因为实例化的类是子类 Child，调用的也是子类的 *t1* 方法。

被重写的情况下，如果想使用父类的构造方法或其他方法，往往会在子类中手工调用一次被重写的方法，这样的话，父类和子类的同名方法都会被应用到。**子类中调用父类中被覆盖的方法就不能用 *$this* 了，而用关键字 *parent*。**

```php
class Dad {
    public function __construct(){
        echo 111;
    }
    public function t1(){
        echo 222;
    }
}
class Child extends Dad {
    public function __construct(){
        parent::__construct(); //调用父类的构造方法
        parent::t1(); //调用父类的t1方法
        echo 333;
        $this->t1(); //调用Child类中的t1方法
    }
    public function t1(){
        echo 444;
    }
}
$c = new Child();  //实例化的时候，Child类的构造方法自动执行，输出111222333444
```

范围解析操作符（也可称作 Paamayim Nekudotayim）或者更简单地说是一对冒号，可以用于访问覆盖类中的方法，也可以访问静态成员和类常量。

> ps：*parent* 只能访问父类的方法，而不能访问父类的属性，除非这个属性是静态的。

我们再看一段代码，再次理解一下 *$this*：

```php
class Dad {
    public function t1(){
        echo 222;
    }
    public function test(){
        $this->t1(); //关键代码
    }
}
class Child extends Dad {
    public function t1(){
        echo 444;
    }
}
$c = new Child();  //实例化
$c->test(); //输出 444
```

> *$this* 是伪对象，上述代码第 *6* 行使用 *$this* 调用t1方法，实际上就是调用的 *$c* 对象的 *t1* 方法，所以优先调用的是子类的 *t1* 方法。

最后，关于重写这里只有一处需要 特别注意 的地方：

```php
class Dad {
    private $x = 1;
    private function t1(){
        echo 'Person';
    }
    public function test(){
        var_dump($this); //输出对象
        echo $this->x . '<br>'; //输出对象的x属性
        $this->t1(); //调用对象的t1方法
    }
}
class Child extends Dad {
    public $x = 2;
    public $y = 3;
    public function t1(){
        echo 'Child';
    }
}
$c = new Child();
$c->test();
//output
/*
object(Child)[1]
  public 'x' => int 2
  public 'y' => int 3
```

```
  private 'x' (Person) => int 1
1
Person
*/
```

> 这段代码有两个关键点，一是父类的成员属性 *$x* 和方法 *t1* 都是私有的，二是 *test* 方法在父类中，这种情况优先调用父类的私有成员。如果父类没有私有成员，则再考虑重写的情况。

9.3 属性重写

属性重写规则和方法重写相同。*parent* 关键字不能调用父类的属性，除非这个属性是静态的。

10、类常量

回顾一下之前学习过的常量的定义：

- define('常量名'，常量值); //define能够在任何位置使用；define定义的常量具有全局性
- const 常量名 = 常量值; //const只能在顶层使用；const在局部中定义常量

可以把在类中始终保持不变的值定义为常量。在定义和使用常量的时候不需要使用 $ 符号。

常量的值必须是一个定值，不能是变量、类属性、数学运算的结果或函数调用。

只能使用 *const* 关键字来定义类常量。在类的方法中，使用 *self* 或者类名调用类常量，在类外部，使用类名或对象名调用类常量。需要注意的是，无论继承与否， *self* 始终表示它所在的类。

```php
class Dad {
    const PHP_VERSION = 5;
    public function __construct(){
        echo self::PHP_VERSION; //类内部，可以使用self关键字调用类常量
        echo Dad::PHP_VERSION; //类内部，可以使用类名调用类常量
        echo static::ABC; //能够这样用，但是有特殊函数
    }
}
$d = new Dad();
echo Dad::PHP_VERSION; //类外部，可以使用类名来调用类常量
echo $d::PHP_VERSION; //类外部，可以使用对象名调用类常量
```

我们再来看下继承的情况下，如何灵活调用类常量：

```php
class Dad {
    const PHP_VERSION = 5;
    public function t1(){
        echo self::PHP_VERSION; //self只表示当前的类，即Dad类，这里调用的是Dad类的PHP_VERSION
        echo Dad::PHP_VERSION; //指定了类名，所以肯定是当前类的PHP_VERSION
    }
}

class Child extends Dad {
    const PHP_VERSION = 7; //重写（覆盖）了类常量
```

```
    public function __construct(){
        $this->t1();
        echo self::PHP_VERSION; //这里输出5.6 调用当前类的类常量
        echo parent::PHP_VERSION; //这里输出5.7 调用父类的类常量
    }
}
$c = new Child();  //输出 5--5--7--5
echo $c::PHP_VERSION; //7
echo Dad::PHP_VERSION; //5
echo Child::PHP_VERSION; //7
```

这段代码不需要过多的解释，只是发现了 ***self* 永远表示它所在的类，*parent* 永远表示父类。**

11、静态成员

使用关键字 *static* 声明类属性或方法为静态，就**可以不实例化类而直接访问。**

定义静态属性及访问静态属性的例子（注意 $ 符号必须有）：

```
class Dad {
    public static $name = '老王'; //定义静态属性
    public function t2(){
        echo self::$name; //类中方法，调用静态属性
        echo Dad::$name; //类中方法，调用静态属性
        echo static::$name; //类中方法，调用静态属性
    }
}
$d = new Dad();
$d->t2();
echo $d::$name; //类外，调用静态属性
echo Dad::$name; //类外，调用静态属性，此方法不需要实例化对象
```

> ps：至于修饰符 *public* 和 *static* 其实没有先后顺序，但习惯写成 *public static $name = '老王'* 。

定义及调用静态方法的例子：

```
class Dad {
    public static function t1(){
        echo 111;
    }
    public function t2(){
        //类内部，调用静态方法的4中形式。
        $this->t1();
        self::t1();
        static::t1();
        Dad::t1();
    }
}
```

```php
$d = new Dad();
$d->t2();
$d->t1(); //类外部，调用静态方法
$d::t1(); //类外部，调用静态方法
Dad::t1(); //类外部，调用静态方法，此方法不需要实例化对象
```

上面两段代码中，调用静态属性和静态方法的方式非常多，实际上在**类的内部，习惯用 *self* 来调用静态成员，在类外部习惯用类名调用静态成员**，这一点和调用类常量一样。非常不建议用 `对象名->静态成员` 的方式来调用（手册上说不可以，实测可以）。

另外，在静态方法中，不能使用 *$this*，基本上可以理解为，静态方法中不能调用非静态成员。

```php
class Dad {
    public static function t1(){
        $this->t2(); //报错，因为静态方法中使用了$this
        @self::t2(); //这个方法虽然可以调用到非静态的t2，但是如果不加错误抑制符@，会报错
    }
    public function t2(){
        echo 111;
    }
}
Dad::t1();
```

下面，我们讨论一下继承中静态成员的问题。

```php
class Dad {
    public static $name = '老王';
    public static function t1(){
        echo 111;
    }
    public static function t3(){
        echo 333;
    }
}
class Child extends Dad {
    public static $name = '小王';
    public static function t1(){
        echo 222;
    }
    public static function t2(){
        parent::t1(); //调用父类的t1方法
        self::t1(); //调用当前类的t1方法
        echo parent::$name; //调用父类的$name属性
        echo self::$name; //调用当前类的$name属性
    }
}
Child::t2(); //调用Child类的t2方法
Child::t3(); //静态方法也会被子类继承，所以这里调用继承而来的t3方法
```

上面的代码比较好理解，静态成员在继承过程中和非静态成员几乎一样。

需要注意的是，重写父类成员（属性和方法）时，要保证成员静态的一致性。比如父类方法是静态的，则重写方法时也要写成静态的，如果父类方法是非静态的，则子类重写方法也不能是静态的，即使父类方法是私有的。

> ps：如果父类属性是私有的，则重写时，可以不统一他们的静态性，不过并不建议这么做。

```php
class A {
    //静态属性
    public static $aa = 1;
    //非静态方法
    public function x(){
        echo 1;
    }
    //静态方法
    public static function y(){
        echo 2;
    }
}
class B extends A {
    public $aa = 2; //错误，重写时，应该也定义成静态的
    public static function x(){} //错误，重写时也必须是非静态的
    public function y(){} //错误，重写时，必须也定义成静态的
}
$b = new B();
```

下面我们讨论一下使用 *static* 调用静态成员时有什么特点：

```php
class Dad {
    public static $name = '老王';
    public static function t1(){
        //self表示当前的类，所以这里肯定输出“老王”
        echo self::$name;

        //static表示在执行到第15行的时候，才给static绑定值，值为使用的类，所以这里输出“小王”
        echo static::$name;
    }
}
class Child extends Dad {
    public static $name = '小王';

}
Child::t1();
```

上述代码体现了 *self* 和 *static* 的不同之处，*static* 表示在执行到第 *15* 行的时候，才给 *static* 绑定值，值为使用的类 *Child* ，这种情况叫做**后期静态绑定。**（后期静态绑定同样适用于类常量）

最后，我们总结一下 *$this* 、*self* 、*static* 、*parent* 都表示什么意思：

- *$this*： 表示类的实例，即对象。是当前使用的对象在类内部的化身。
- *self*： 表示 *self* 所在的类
- *static*： 表示代码执行时，使用的类，当前使用的是哪个类，static就表示这个类
- *parent*： 表示父类

再总结一下调用类常量及静态成员的方式（*self* 和类名都可以调用类常量和静态成员，但是建议使用 *self* ）：

	$this	self	parent	static	类名	对象名
类常量	×	√（类内）	√（类内）	√（类内）	√（类外）	不建议
静态属性	×	√（类内）	√（类内）	√（类内）	√（类外）	不建议
静态方法	不建议	√（类内）	√（类内）	√（类内）	√（类外）	不建议
普通成员属性	√（类内）	×	×	×	×	√（类外）
普通成员方法	√（类内）	不建议	√（类内）	×	不建议	√（类外）

能够调用，但不表示常用。我们只记住常用的即可。上面高亮的记住就可以了。

12、对象在内存中的表现形式

12.1 简介

计算机中的任何软件在工作的时候，都会将它的程序加载到内存中，然后在执行。

一段PHP代码在执行的时候，也会先将其加载到内存中，然后在执行的。我们先来分析一段简单的代码，看这段代码在执行的时候，内存中是如何体现的。

比如 *test.php* 文件中的代码如下：

```
1 $a = 'hello world';
2 echo $a;
```

上述代码非常简单，声明一个变量 *$a* ，然后输出 *$a* 。

当浏览器访问 *test.php* 文件的时候，对应的内存占用情况如下图所示：

首先会将 *test.php* 文件中的代码加载到内容中，比如代码执行到第1行，会占用一些内存，用于存放变量 *$a* ，值为1。

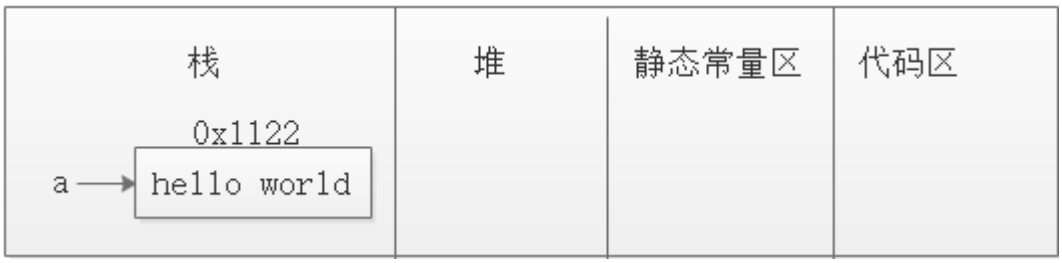
实际上，**内存中有不同的区域，分别存放不同形式的代码**，这如同我们的电脑有C盘、D盘等，分别存放不同的东西一样。那么内存中有哪些区域呢？**主要的区域有四个**（有些人或别的学科可能分的区更多，但我们不需要），**分别是代码区、静态常量区、栈区、堆区。**

- 代码区：存放函数、类等代码。
- 静态常量区：存放静态变量和常量。
- 栈区：在PHP中，可以直接存放字符串、整型、浮点、布尔、数组、资源型数据，也用于存放对象的引用。
- 堆区：存放对象。

到此，我们再来设计一张 *test.php* 执行时的内存图。

代码执行到第1行，会占用一些内存，存放的地方是内存中的栈区，所以叫做开辟新栈，用于存放变量 *$a* ，值为1。开辟的新栈会有地址，比如叫做0x1122。理解起来也比较简单，我们把一栋宿舍楼比作栈区，开辟的新栈就是楼里面的宿舍，而每个房间都会有门牌号，栈内存中的地址就如同宿舍的门牌号。

最后，我们再从新设计一张 *test.php* 执行时的内存图。

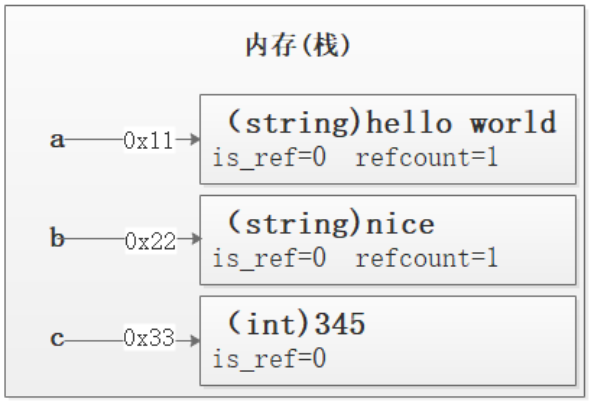


12.2 PHP变量存储机制zval

实际上，上图的变量表示还是比较简陋的。在PHP内部，变量是存储在一个叫做zval的容器中。它不仅仅包含变量的值，也包含变量的类型。变量容器中包含用来区分是否引用的字段 *is_ref*。同时它也包含这个值的引用计数 *refcount*。

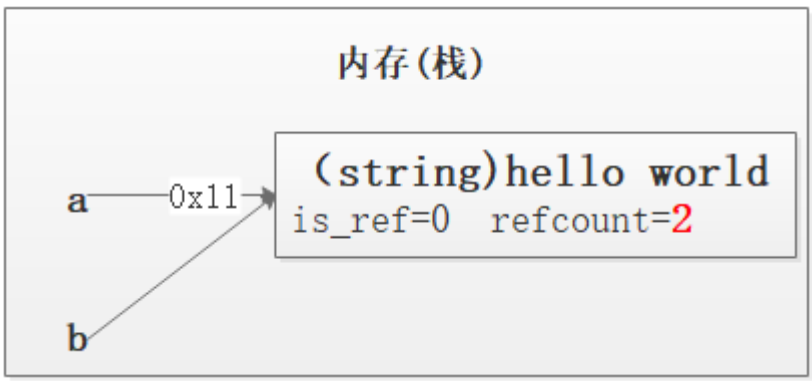
```
$a = 'hello world';
$b = 'nice';
$c = 345;
```

变量存储在一个相当于关联数组的符号表中。这个数组以变量名为key,并且指向包含了这些变量的容器。如下图所示：



PHP试着在变量拷贝(如 *$a = $b*)的时候变得聪明些。当进行赋值操作时，Zend引擎不会创建一个新的变量窗口，而是增大变量窗口的 *refcount* 字段，你可以想象一下，当这个变量是一个巨大的字符串或一个巨大 的数组时，这将节约多少的内存。如下图所示：

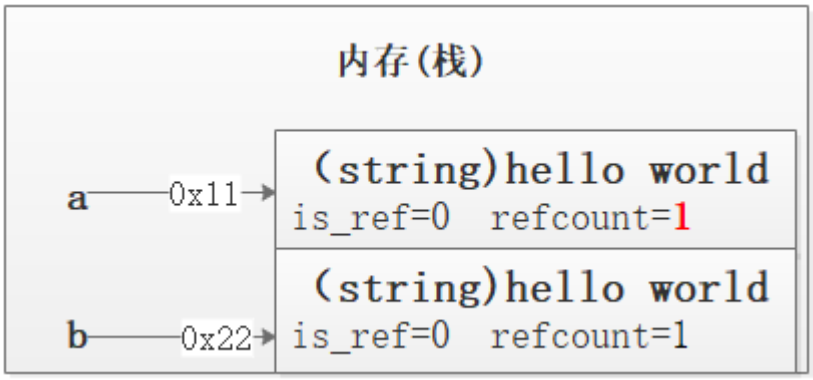
```
$a = 'hello world';
$b = $a;
```



那么如果修改变量 *$b* ， *$a* 还存在吗?

```
$a = 'hello world';
$b = $a;
$b = 'nice';
```

这个时候，首先看 *is_ref* 是否为 *0* ，如果为 *0* ，则执行分离操作，即原容器的 *refcount* 减去1，并为 *$b* 新增一个容器。此时 *$a* 和 *$b* 同时存在。如下图所示：



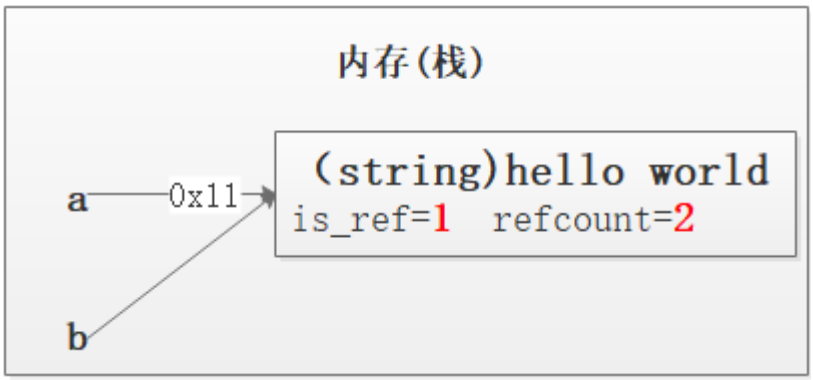
如果删除了 *$b*, 则也只是原容器的 *refcount* 值减去1。并不会影响到 *$a* 。这里就不再画图了。

如果删除了 *$b* ， 并且也将 *$a* 删除了呢？按照上面的思路，存放 *hello world* 的容器的 *refcount* 将减为 *0*。如果一个容器的 *refcount* 值小于 *1* ， Zend就会回收该zval，即释放内存，也就是垃圾回收。

另一个问题：

```
$a = 'hello world';
$b = &$a; //引用赋值
```

这时，内存图示如下：



这时， *$a* 和 *$b* 同样指向同一个容器，只不过容器的 *is_ref* 字段变为了 *1* 。

这种情况下，如果修改或删除了 *$b* ， 会影响到 *$a* 吗？答案是会的。因为当修改 *$b* 的时候，先判断 *is_ref* 的值，如果值为 *1* ， 则指向该容器的变量的任何变化都会影响到这个容器。

参考资料：https://www.jb51.net/article/38370.htm

参考资料：https://www.cnblogs.com/orlion/p/4980641.html

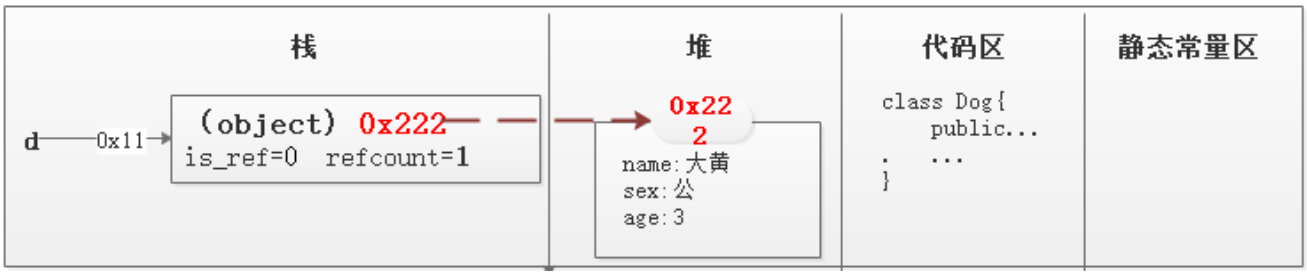
参考资料：http://www.laruence.com/2018/04/08/3170.html

12.3 对象在内存的表现形式

zval容器可以存储任何类型的PHP变量，但是也有所不同，简单来说，对于能存下值的变量则直接在zval中存值，不能存放值的，则存一个地址，对象就是这样的。

```php
//定义类
class Dog {
    public $name = '大黄';
    public $age = 3;
    public $sex = '公';
    public function say(){
        echo '汪汪';
    }
}
$d = new Dog(); //实例化
```

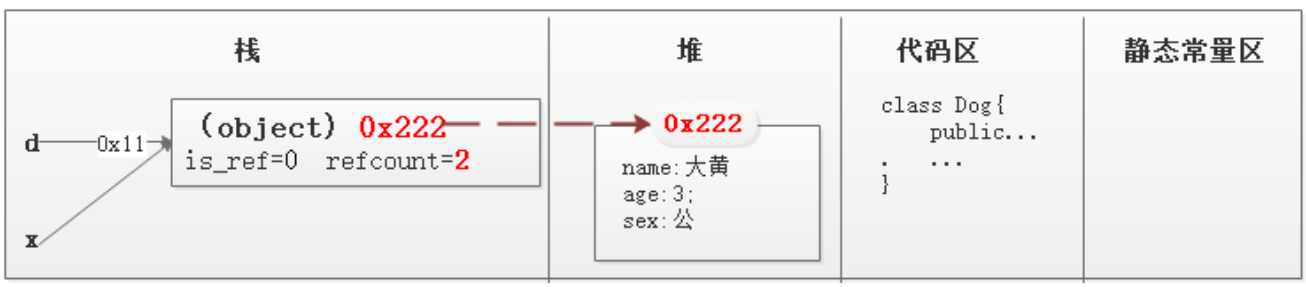
当实例化对象的时候，这段代码在内存中的形式如下图：



因为对象中的内容“五花八门”，用一个字符串不好表示，而且太大，所以干脆在zval容器中存放一个指向堆区的地址 *0x222* 。在操作对象的成员的时候，就可以通过这个地址找到对象的成员了。

那么，如果下面的代码执行结果是怎样的呢?

```php
//定义类
class Dog {
    public $name = '大黄';
    public $age = 3;
    public $sex = '公';
    public function say(){
        echo '汪汪';
    }
}
$d = new Dog(); //实例化
$x = $d;
$x->name = '小黄';
echo $d->name;
```



代码第 *11* 行的意思我们前面说过了，意思是 *$d* 和 *$x* 都指向同一个zval容器，该容器的 *refcount* 加 *1* 。

代码第 *12* 行，修改了 *$x* 的 *name* 属性，这时，就会根据zval保存的地址 *0x222* 找到堆区的内容，然后对 *name* 的值进行修改。

所以，*13* 行输出的内容是 *小黄* 。

> 把一个对象赋值给另一个变量的时候，不需要使用 *&* ，两个变量指向的都是同一个堆空间，所以无论改变哪个对象都会影响到另一个对象。这就是对象在内存中的形式以及和其他变量的不同之处。但是要知道加 *&* 和不加 *&* 还是有区别的。

那么如何得到两个完全独立的对象呢？下一节对象克隆（clone）将会讲解。

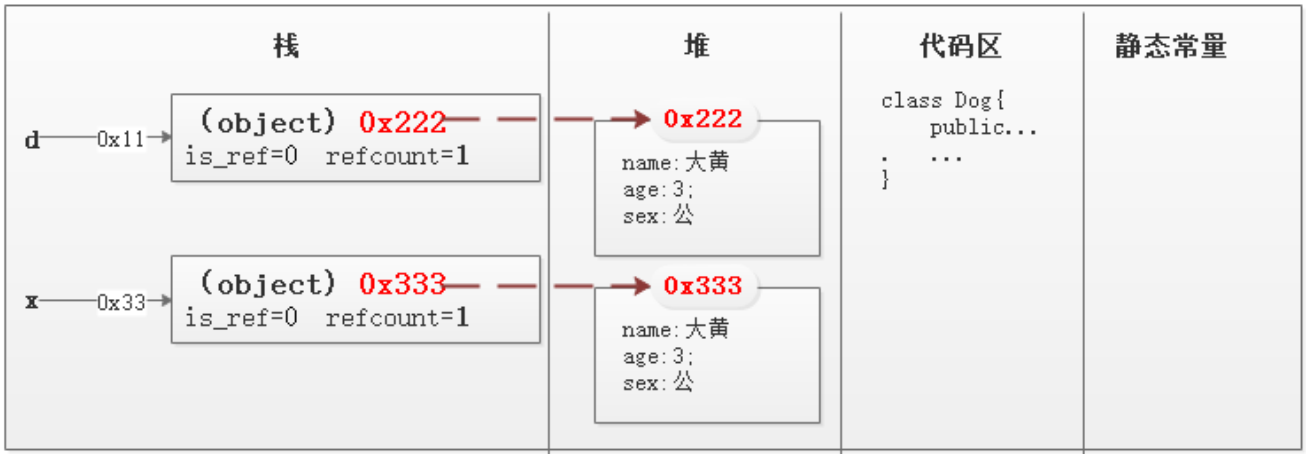
总结：除了对象以外，其他所有数据类型默认都是非引用传值。而对象默认情况下就相当于引用传值。

```php
$s = 'hello';
$a = &$s;
$a = 'nice';

class A {
    public $name = '老王';
}
$m = new A();
$n = $m;  //这里的赋值不需要 & ，最终的效果和加 & 的效果一样
$n->name = '小王';
```

12.4 对象克隆（复制）

使用关键字 *clone* ，克隆一个对象的时候，内存将发生如下变化：

```php
//定义类
class Dog {
    public $name = '大黄';
    public $age = 3;
    public $sex = '公';
    public function say(){
        echo '汪汪';
    }
}
$d = new Dog(); //实例化
$x = clone $d; //克隆对象
$x->name = '小黄';
echo $d->name;
```



克隆一个对象的时候，会在栈中新增一个zval容器，并且会在堆区新增一个保存对象 *x* 的空间。这样的话，变量 *d* 和 *x* 就是两个独立的对象了，修改其中一个对象就不会影响另一个对象了。

我们都知道，实例化一个对象的时候，能够通过构造函数完成一些初始化的工作，那么能不能在克隆一个对象的时候也完成一些初始化工作呢？答案是可以的，只需要在类里面声明一个魔术方法 *__clone* 即可，该方法会在对象被克隆时自动触发。

```php
class Dog{
    public $name;
    //构造方法
    public function __construct($n){
        $this->name = $n;
    }
    //克隆方法
    public function __clone(){
        $this->name = '小黄';
    }
}
$c = new Dog('大黄');
$x = clone $c;
echo $c->name; //输出 大黄
echo $x->name; //输出 小黄
```

> 克隆方法和析构方法一样，都不能有参数。也不能是静态方法。

12.5 对象比较

当使用比较运算符（==）比较两个对象变量时，比较的原则是：如果两个对象的属性和属性值 都相等，而且两个对象是同一个类的实例，那么这两个对象变量相等。

而如果使用全等运算符（===），这两个对象变量一定要指向某个类的同一个实例（即同一个对象）。

```php
//定义两个类
class A{
    public $flag = 1;
}
class B{
```

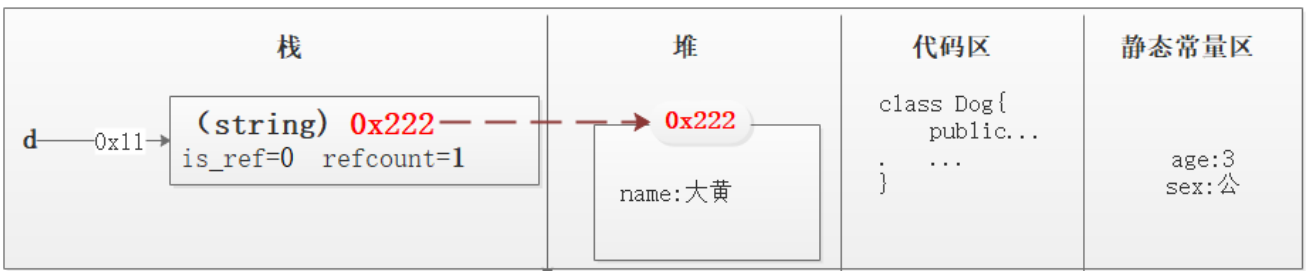
```php
    public $flag = 2;
}
//创建对象
$a1 = new A();
$a2 = new A();
$a3 = $a2;
$b1 = new B();
$b2 = clone $b1;
//比较各个对象
var_dump($a1 == $a2); //true
var_dump($a1 === $a2); //false
var_dump($a2 == $a3); //true
var_dump($a2 === $a3); //true
var_dump($a1 == $b1); //false
var_dump($b1 == $b2); //true
var_dump($b1 === $b2); //false
```

12.6 静态成员和类常量的存在形式

静态成员和类常量，存在于内存图示中的静态常量区。它们直接由类管理，不属于任何一个对象。

```php
class Dog {
    public $name = '大黄';
    public static $age = 3;
    const sex = '公';
}
$d = new Dog();
var_dump($d);
//输出
/*
object(Dog)[1]
  public 'name' => string '大黄' (length=6)
*/
```

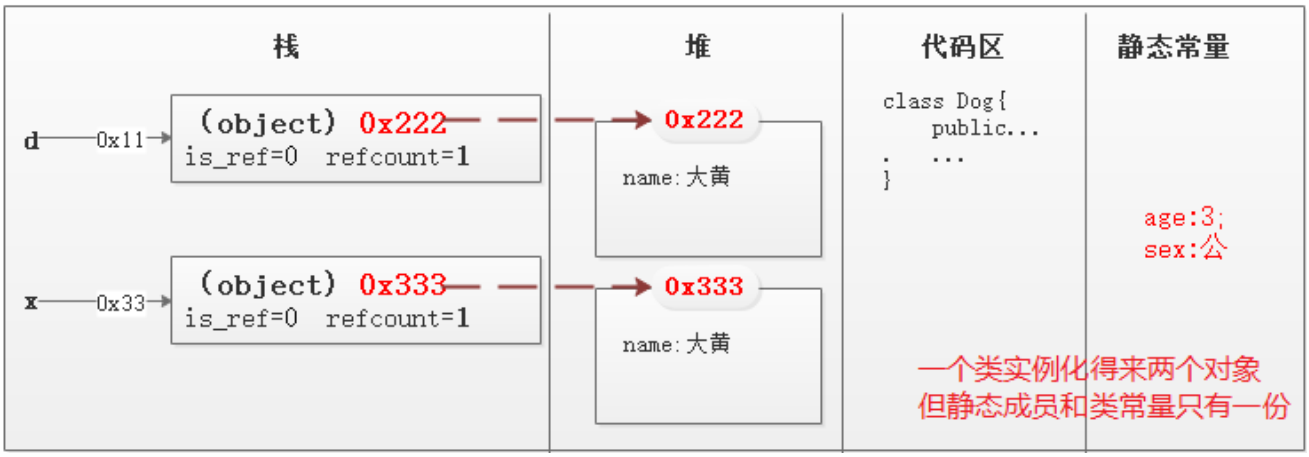
对应的图示：



从代码的输出结果中，我们可以看出，输出的对象中只有 *name* 属性，而静态属性和类常量并没有，说明静态成员和类常量只属于类，而不属于对象。只是对象和类有关系，所以对象才能使用静态成员和类常量的。

> 因为静态成员和类常量由类管理，不属于任何一个具体的对象，所以当一个类有多个实例的时候，静态成员和类常量在内存中只有一份（不会为每个对象都创建一份）。所以在实际开发中，还是比较建议多使用静态成员的。
>
> 静态方法和静态属性一样，也由类管理。
>
> **但是除了__callStatic方法外，所有的魔术方法都不能是静态的。**

```php
class Dog {
    public $name = '大黄';
    public static $age = 3;
    const sex = '公';
}
$d = new Dog();
$x = new Dog();
```



13、类文件自动加载

在一个项目中，一个文件一般保存一个类，所以最后就会有很多类文件。在使用这些类文件的时候，就会不断的写 *require* 或 *include* 语句。

类文件命名：

- 文件名：xxx.class.php 里面的类名：xxx。比如 Mysql.class.php
- 全部小写，文件名：xxx.php 里面的类名：xxx。比如 Mysql.php

我们后面开发的时候，全部使用第二种（全部小写）的方式定义类文件。

在 PHP 5 中，已经不再需要这样了。*__autoload* 和 *spl_autoload_register* 函数可以注册任意数量的自动加载器，当使用尚未被定义的类（class）和接口（interface）时自动去加载。通过注册自动加载器，脚本引擎在 PHP 出错失败前有了最后一个机会加载所需的类。

13.1 __autoload函数

注意，这个一个函数，并不是我们之前学过的魔术方法。

该函数需要一个参数，参数为待加载的类名。具体如下：

```php
// 文件:   a.php
class B {
    public $name = 'aaa';
}
```

```php
//文件:   b.php
class A {
    public $name = 'bbb';
}
```

```php
//文件:   11autoload.php  (a.php和b.php和11autoload.php在同一个目录下)
//声明自动加载函数
function __autoload($class_name){
    require_once strtolower($class_name) . '.php';
}
//测试
$a = new A();
$b = new A();
echo $a->name; //输出aaa
echo $b->name; //输出bbb
```

上述三个文件都在同一个目录中，其中 *a.php* 和 *b.php* 里面各声明了一个类，*11autoload.php* 定义自动加载函数，并进行测试。

11autoload.php 第 *7* 行和第 *8* 行代码，直接实例化一个类，但是发现前面并没有加载 a 类和 b 类，在报错之前，自动触发 *__autoload* 函数，并将使用的类名 *A* 和 *B* 当做参数传递给 *__autoload* 。从而实现了类文件的自动加载。

> 如果有自己写的 *require* 或 *include* 语句，则优先使用自己写的加载语句，没有这样的语句，才会触发 *__autoload* 。

```php
//文件:   11autoload.php  (a.php和b.php和11autoload.php在同一个目录下)
function __autoload($class_name) {
    echo '1'; //如果自动加载函数执行，则输出1
    require_once strtolower($class_name) . '.php';
}
require_once 'a.php';
$a = new A();
$b = new B();
// 结果输出一次 1
```

> 尽管 *autoload()* 函数也能自动加载类和接口，但更建议使用 *spl_autoload_register()* 函数。*spl_autoload_register()* 提供了一种更加灵活的方式来实现类的自动加载（同一个应用中，可以支持任意数量的加载器，比如第三方库中的）。因此，不再建议使用 *__autoload()* 函数，在以后的版本中它可能被弃用。

13.2 spl_autoload_register函数

spl_autoload_register() 是一个系统内置函数。用法有下面两种情况：

使用方法一：先定义一个函数或类中的方法，然后调用*spl_autoload_register()* 注册自定义的函数。

```php
//定义一个自动加载函数，函数名不是__autoload
function my_load($class_name) {
    echo '1'; //测试，如果自动加载函数执行，则输出1
    require_once $class_name . '.php';
}
spl_autoload_register('my_load'); //注册上面定义的函数为自动加载函数
$a = new a();
$b = new b();
```

有些人可能在想，把上面的自动加载函数名换成 *__autoload* 而去掉代码第 *6* 行不就可以了吗？为什么要多写一行代码呢？答案一是使用 *spl_autoload_register()* 比较灵活，可以实现多个函数为自动加载函数；二是可以把 *my_load* 函数定义到类里面。再看下面的代码：

```php
function my_load($class_name){
    echo 1;
    $file = strtolower($class_name) . '.php'; // a.php
    if(file_exists($file)){
        require_once $file;
    }
}

function my_load2($class_name){
    echo 2;
    $file = strtolower($class_name) . '.class.php'; // mysql.class.php
    if(file_exists($file)){
        require_once $file;
    }
}
//使用spl_autoload_register()注册my_load，使之有__autoload函数一样的功能
spl_autoload_register('my_load');
spl_autoload_register('my_load2');

$a = new A();
echo $a->name;

$b = new B();
echo $b->name;

$m = new Mysql(); //和当前文件同目录下有一个mysql.class.php文件
```

上述代码，就将 *my_load* 和 *my_load2* 都注册成了自动加载函数。在使用一个未加载的类的时候，会优先使用 *my_load*函数，如果这个函数能够加载成功，则不再调用 *my_load2*；如果 *my_load* 函数不能成功加载类文件，则才调用 *my_load2*。如果 *my_load2* 也不能成功加载，就会报错。

```php
//定义自动加载函数
function my_load($class_name) {
    $file = strtolower($class_name) . '.php';
    //判断文件是否存在，如果存在就 require
    if(file_exists($file)){
        require_once $file;
    }
}
//在类中定义一个方法为自动加载方法
class Test {
    public static function my_load2($class_name){
        $file = strtolower($class_name) . '.class.php';
        if(file_exists($file)){
            require_once $file;
        }
    }
}
//Test::my_load2()
spl_autoload_register('my_load'); //注册函数 my_load 为自动加载函数
spl_autoload_register('Test::my_load2'); //注册Test::my_load2 为自动加载函数
/***** 如果my_load2不是静态方法，则写成下面两行代码的形式 *****/
//$t = new Test();
//spl_autoload_register(array($t, 'my_load2'));
$a = new a();
$b = new b();
$m = new mysql();
```

上面的代码， *spl_autoload_register()* 就将两个函数（方法）都定义为了自动加载函数了。程序执行到第 *23* 行的时候，会先调用 *my_load* （因为先注册的这个函数），看能否加载使用的类。如果能，则不再调用另外的自动加载函数了；如果不能，则调用下一个自动加载函数，依次类推。如果所有的自动加载函数都不能加载使用的类文件，则报错。

使用方法二：直接调用 *spl_autoload_register()* ，将自动加载函数直接当做 *spl_autoload_register()* 的参数。

```php
//直接定义spl_autoload_register函数，它的参数就是自动加载函数
spl_autoload_register(
    function ($class_name){
        //...
        $file = strtolower($class_name) . '.php';
        if (file_exists($file)) require_once $file;

        $file = strtolower($class_name) . '.class.php';
        if (file_exists($file)) require_once $file;
    }
);

$a = new A();
$b = new B();
$m = new Mysql();
```

这种方式非常简单明了，不会占用过多的代码，而且在类方法中或类的外面都可以使用，比较推荐使用。

> 一定要注意的是，定义了自动加载函数后，只要使用了没有手动加载（即单独写一行require或include加载）的类，就会触发自动加载函数，比如实例化一个类、继承一个类等都会用到自动加载。

14、设计模式之单例模式

设计模式（Design pattern）代表了最佳的实践，通常被有经验的面向对象的软件开发人员所采用。设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。

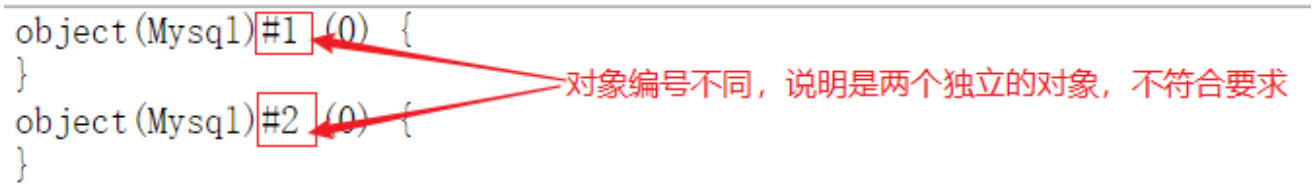
举例说明，有一个类，希望只能够得到该类的唯一一个对象，代码如何实现？如果解决了这个问题，那么你的代码就可以叫做设计模式了，设计模式的概念就是如此。

什么叫做得到该类的唯一一个对象呢？即无论通过什么方法，只要能够得到该类的对象而不报错，并且打印这些对象的时候必须是相同的结果（对象编号相同）。

下面我们一步一步实现这段代码（使用Mysql类为例）：

```php
class Mysql {

}
$m1 = new Mysql();
$m2 = new Mysql();
var_dump($m1, $m2);

```

输出结果如下：



我们要求得到的对象必须一样，这样的结果是不符合要求的。

改进代码，不能在类外部随意 *new* ，如果随意 *new* ，则肯定会得到很多不同的对象。前面学习过，实例化对象的时候，就会在类的外部自动执行构造函数，所以将构造函数定义为私有的，就可以阻止在类外部 *new* 一个类。

```php
class Mysql {
    //加入私有的构造函数
    private function __construct() {

    }
}
$m1 = new Mysql(); //实例化类的时候，报错
```

上面的代码也不符合要求，这样的写法，使得类无法实例化了。

但是也不绝对，私有方法虽然不能在类的外部被调用，但是可以在类的内部被调用，所以我们可以在类的内部定义一个公开的静态方法，在这个方法里面完成实例化的工作。

```
class Mysql {
    //加入私有的构造函数
    private function __construct() {

    }
    //定义公开的静态方法，用于创建对象
    public static function getInstance() {
        //$m = new Mysql(); //建议使用下面一行代码实例化
        $m = new self();
        return $m;
    }
}
$m1 = Mysql::getInstance();
$m2 = Mysql::getInstance();
```

上面的代码，虽然在类的内部完成了实例化的工作，但是当多次调用 *getInstance* 的时候，还是会得到多个对象。

解决方法是，进行判断。先设置一个保存 *Mysql* 类的对象的属性，起始值为 *null*，判断如果这个属性值为 *null* 就实例化 *Mysql*，并赋值给这个属性，否则直接返回这个属性。

```
class Mysql {
    //定义一个保存该类对象的成员属性
    private static $instance = null;
    //加入私有的构造函数
    private function __construct() {

    }
    //定义公开的静态方法，用于创建对象
    public static function getInstance() {
        if(self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }
}
$m1 = Mysql::getInstance();
$m2 = Mysql::getInstance();
```

这样的话，得到的 *$m1* 和 *$m2* 就是相同的结果了。但是还没有结束，如果根据 *$m1* 克隆一个 *$m3* 呢？得到的 *$m3* 和原来的 *$m1* 又不相同了，解决办法是，定义一个私有的 *__clone* 方法，阻止克隆对象。

```
class Mysql {
    //定义一个保存该类对象的成员属性
    private static $instance = null;
    //加入私有的构造函数
```

```
    private function __construct() {

    }
    //定义公开的静态方法，用于创建对象
    public static function getInstance() {
        if(self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }
    //定义私有的克隆方法
    private function __clone(){}
}
$m1 = Mysql::getInstance();
$m2 = Mysql::getInstance();
//$m3 = clone $m1; //报错，因为__clone是私有方法
```

到此，我们就得到了 *Mysql* 类的唯一对象了。我们通过代码的方式，解决了这个问题，所以这段代码就可以称之为设计模式了。实际上，这段代码早就被前人写好了，并给她起了一个名字，叫做**单例模式**。

> 总结起来，要完成单例模式，需要"**三私一公两静态**"，即三个私有成员（保存对象的私有属性、私有构造方法、私有克隆方法），一个公开成员（返回对象的方法），两静态（静态的得到对象的方法，静态的保存对象的属性）。

单例模式的意义是，很多时候一个类只需要一个对象就可以完成所有的工作了，这个时候最好就使用单例模式，典型的应用就是数据库操作类。

15、最终类（final）

定义一个类后，有两种选择使用这个类：

- 实例化这个类
- 被别的类继承

而最终类和抽象类分别体现了类的两个极端。

- 最终类：只能被实例化，不能被继承。
- 抽象类：只能被继承，不能被实例化。

15.1 最终类

定义一个最终类：

```
final class Mysql {

}
```

Mysql 类只能被实例化，而试图继承 *Mysql* 将会报错。

最终类的意义体现于，表明该类已经是继承链条上的最后一个类了，它不能有后代类（子类）了。另外的说法是，说明该类比较完善了，不需要被扩展了。*final* 类保证了对继承的控制。

最终类不能被继承，但是他可以继承别的类：

```
class Db {

}
//定义最终类
final class Mysql extends Db{

}
```

最终类的定义比较简单，类里面也可以定义任何形式的成员（类常量、静态成员、非静态成员、魔术方法）。

```
final class Test {
    const ABC = 'hello'; //类常量
    static $a = 1;  //静态属性
    private $name = 'zhangsan'; //私有属性
    protected $age = 20;
    public $sex = '男';
    public function __construct(){
        echo self::ABC;
        echo $this->name;
    }
    static function aa(){

    }
    public function __clone(){

    }
}
```

15.2 最终方法

使用 *final* 修饰的方法为最终方法，最终方法可以被定义到最终类里面和普通的类（非最终类）里面。**最终方法的特点是不能被重写**。根据最终方法的特点来看，将其定义到普通的类里面才有意义（因为最终类不能被继承，更谈不上重写）。

最终方法的意义在于，约束方法不能被重写。

```
class A {
    //定义最终方法
    final public function x(){
        echo 1;
    }
    //定义最终静态方法
    final public static function y(){
        echo 2;
    }
```

```php
    //定义最终构造方法
    final public function __construct(){
        echo 3;
    }
}
class B extends A {
    //下面三个方法，试图重写父类方法，全部报错
    //public function __construct(){ }
    //public function x(){}
    //public static function y(){}
}
$b = new B();
```

> 属性不能被定义为 final，只有类和方法才能被定义为 final。

16、抽象类（abstract）

16.1 抽象类

上一节提到，类的两个极端，一个是只能被实例化，不能被继承；而另一个极端是只能被继承而不能被实例化，这种类就是抽象类。

抽象类之所以叫做抽象类，是因为它确实很抽象。比如上级向下级传达精神：

1. 传智总裁：我们要争做行业内的老大，努力奋斗，不忘初心。 --- （非常抽象）
2. PHP学院院长：我们要充分领会总裁的精神，务必做到兢兢业业，不要误人子弟。 --- （一般抽象）
3. PHP教学总监：我们要充分领会院长的精神，好好备课，好好上课。 --- （有点具体）
4. PHP讲师：备课、上课、学习。 --- （太TM具体了）

抽象类就如同“总裁”，是继承链条中最顶层的类，它只规定一些方法，但并不去实现他们，而是把任务交给下一级去实现，如果下一级也也实现不了，则可以继续交给下一级，以此类推，到继承链条的最底层，则**必须**实现上层规定的方法。

用 *abstract* 声明抽象类。

```php
abstract class A {

}
```

试图实例化一个抽象类将会报错。

```php
abstract class A {

}
$a = new A(); //报错 抽象类不能被实例化，只能被继承
```

抽象类中的成员可以有之前学习过的任何内容（类常量、静态成员、非静态成员、魔术方法），除此之外，抽象类中还可以有抽象方法，有关抽象方法的知识，下一节讲解。

```php
abstract class A {
    const ABC = 'hello world'; //类常量
    public $x = 1;  //普通属性
    private $y = 2; //私有属性
    public static $z = 3; //静态属性
    //构造方法
    public function __construct(){
        echo $this->aa();
    }
    //私有方法
    private function aa(){
        return $this->y;
    }
    //静态方法
    protected static function bb(){

    }
    //.....
}
```

> 值得注意的是，抽象类只能被继承，如果抽象类中定义了私有成员，则子类是无法使用的，只能在抽象类中自己使用，否则就不要定义成员私有成员。

16.2 抽象方法

使用 *abstract* 定义抽象方法。抽象方法必须满足下面两个条件：

- 抽象方法不能有方法体，即不能有大括号。
- 抽象方法不能是私有的，因为继承它的子类必须重写它。

```php
abstract class A {
    abstract public function aa(); //正确
    abstract protected function bb($x, $y=1); //正确
    abstract public static function cc(); //正确
    abstract private static function dd(); //错误，因为方法是私有的
    abstract public static function ee(){ // 错误，因为方法有大括号

    }
}
```

> 抽象方法不能定义到普通的类里面。反过来说，含有抽象方法的类一定是抽象类。

当一个类继承了抽象类，就必须完成继承的抽象方法，即重写父类的抽象方法，而且要满足重写规则。

```php
abstract class A {
    abstract public function aa(); //正确
    abstract protected function bb($x, $y=1); //正确
    abstract public static function cc(); //正确
```

```
}
class B extends A {
    //实现父类aa方法
    public function aa(){

    }
    //实现父类bb方法，并且注意参数个数和默认值
    public function bb($x, $y=4){

    }
    //实现父类静态方法
    public static function cc(){

    }
}
```

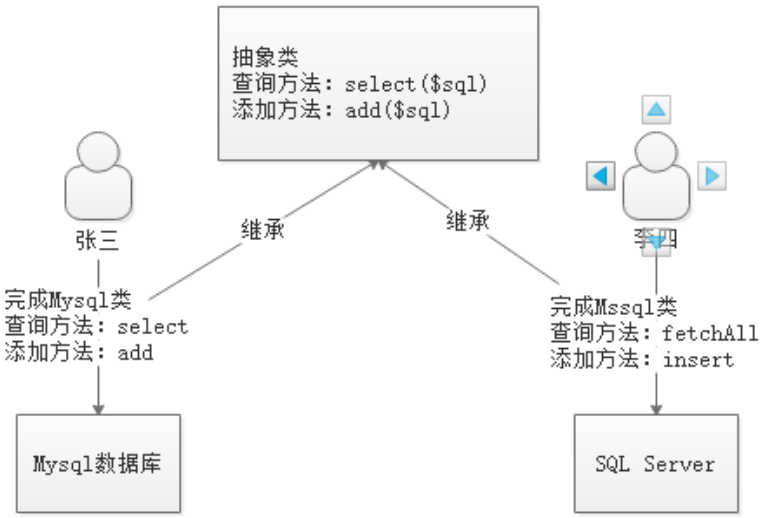
如果子类不能完成继承的抽象方法，则继续充当抽象方法，由它的后代类去完成这些抽象方法。

```
abstract class A {
    abstract public function aa(); //正确
    abstract protected function bb($x, $y=1); //正确
    abstract public static function cc(); //正确
}
abstract class B extends A {
    //实现父类抽象方法aa
    public function aa(){

    }
    //实现父类抽象方法bb
    public function bb($x, $y=4){

    }
    //抽象方法cc在B类中无法实现，则可以不写
    //public static function cc();
}
class C extends B {
    //实现上层未实现的方法
    public static function cc(){

    }
}
```

抽象类及抽象方法的意义在于，复杂的层级关系中，实现开发的规范化。

比如有一个项目，该项目需要对 *Mysql* 数据库进行操作，也需要对 *SQL Server* 数据库进行操作。则需要针对不同的数据库各自写一个类，假设项目经理让张三完成 Mysql 类，让李四完成 Mssql 类，张三在写查询方法的时候，定义的方法名为 *select* ，参数为一条SQL语句，而李四在写查询方法的时候，方法名叫 *fetchAll* ，参数为一条SQL语句和一个表示数据格式的 *type* 。这样的话，我们在做查询操作的时候，非常容易混淆两个查询方法。

好的做法是可以先定义一个抽象类，里面规定好查询方法叫什么名字，参数有哪些。张三和李四在写各自的类的时候，需要继承该抽象类，这样对两名程序员就要求一致了。我们在调用查询方法的时候，用法肯定是一样的了。



代码演示：

```php
abstract class Db {
    abstract public function select($sql);
    abstract public function add($sql);
}

//张三开发 Mysql类
final class Mysql extends Db {
    public function select($sql){

    }
    public function add($sql){

    }
}
//李四开发 Mssql类
final class Mssql extends Db {
    public function select($sql){

    }
    public function add($sql){

    }
}

$obj = new Mssql(); //如果更换类的话，select方法始终不变
$obj->select('select * from article');
```

17、对象接口（interface）

使用 *interface* 定义接口，接口中只能有接口常量和方法（静态和非静态都可以）。

- 接口中的接口常量，**后代**不能重写。

- 接口中的方法必须是 *public* 形式的
- 接口中的方法不能使用 *abstract* 定义，但都是抽象的，也就是不能有方法体（大括号）

```
interface A {
    const ABC = 'hello world';
    public function aa();
    public static function bb();
}
```

对象接口也能像抽象类一样被另外的对象接口继承，但不能被实例化。

```
interface A {
    const ABC = 'hello world';
    public function aa ($x, $y=1);
    public static function bb();
}
//定义接口B，继承A
interface B extends A {
    const ABC = 'nihao'; //错误，接口不能重写常量
    public function cc(); //可以，子接口中可以定义自己的抽象方法
}
```

可以使用关键字 *implements* 让一个类去实现接口，实现接口时，则必须完成接口中定义的所有方法，而且必须满足方法重写要求。

```
//定义接口A
interface A {
    const ABC = 'hello world';
    public function aa ($x, $y=1);
    public static function bb();
}
//用接口B继承接口A
interface B extends A {
    public function cc();
}
//定义类实现接口B，则必须实现A、B接口中所有的方法
class C implements B {
    public function cc(){

    }
    public function aa($x, $m=3){

    }
    public static function bb(){

    }
}
```

另一个问题，PHP中的继承是单继承，即一个类一次只能继承一个类。而类去实现接口，一次性可以实现多个接口，同样，类必须实现所有接口中的方法，并且这些被实现的接口中不能有重名方法。

```php
//定义接口A
interface A {
    const ABC = 'hello world';
    public function aa ($x, $y=1);
    public static function bb();
}
//定义接口B
interface B{
    public function cc();
     //11行错误，C同时实现了A、B接口，要求A、B接口中不能有同名方法，即使这里的bb方法是非静态的也不行
    public function bb();
}
//定义类C 实现接口A和B
class C implements A,B {
    public function cc()
    {
    }
    public function aa($x, $m=3){

    }
    public static function bb(){

    }
}
```

对象接口的意义：

对象接口比抽象类更加抽象，因为抽象类中除了有抽象方法以外还可以有非抽象方法，也就是说抽象类可能还会干点活，但是接口中除了常量外就只有抽象方法了，也就是说接口一点活都不干。那么对象接口的意义何在呢？

看一个例子，完成音乐播放的例子：

```php
//定义音乐播放接口
interface Player {
    function play();
    function stop();
    function prev();
    function next();
}
//定义USB设备接口
interface USB {
    const width = 12; //usb的宽度
    const height = 5;
    function in(); //输入方法
    function out(); //输出方法
}
```

```php
//mp3功放类，实现上面两个接口
class MP3Play implements Player,USB {
    function play(){}
    function stop(){}
    function prev(){}
    function next(){}
    function in(){} //输入方法
    function out(){} //输出方法
}
```

上面代码就是使用接口的一个典型的例子。

要完成一个工作，有些时候单一的继承（实现）无法完成，必须同时完成两个方面的工作也能够完成最终的工作，而 *Player* 和 *USB* 这两个东西又不具有父子关系，这个时候就需要用到接口了。

最后，何时使用抽象类？何时使用接口呢？

- 当为两个同类型的类设计一个上层的话，设计为抽象类。比如前面提到的两个数据库操作类。
- 当为一个类设计两个不同类型的上层的话，设计为接口。比如上面的音乐播放案例。

18、面向对象封装性

面向对象有三个特性，分别是继承性、封装性和多态性。

封装特性体现为：通过限制只有特定类的对象可以访问该类的成员（成员属性及成员方法），从而隐藏了某一方法的具体运行步骤的机制。

封装的好处是：将类的实现与使用分开，操作更方便灵活。使用类时，无需关心类的内部实现。可以使开发人员专注于开发，同时减少程序之间的相互依赖。

这犹如PHP内置函数，比如函数 *time()* 、*substr()* 、*imagecreatetruecolor()* 等，你只需要知道如何使用它们即可，而不需要知道它的底层是如何实现的，除非你是PHP开发组成员。

再比如，我们前面学习过的 *Mysqli* ，以及后面要学习的 *PDO* ，都是PHP内置的类，你只要会使用类中的方法完成工作即可，而不需要关系它们的内部实现。

封装性是通过访问控制符实现的。

实际开发中，我们提倡的原则是，最大限度的封装。即，对于属性/方法而言：

- 能设置为private的，不设置为protected
- 能设置为protected的，不设置为public

比如我们之前写的 *Mysql* 类：

```php
class Mysql {
    /************************ 定义成员属性 ***************************/
    private $host = 'localhost';
    private $user = 'root';
    private $pwd = '123';
    private $dbname = 'php68';
    private $charset = 'utf8';
    private $link = null; //用于存放连接数据库的资源
```

```php

    /************************ 定义成员方法 ************************/
    //构造方法
    public function __construct(){
        $this->connect(); //连接数据库
    }
    //连接数据库的方法
    private function connect(){
        $this->link = mysqli_connect($this->host, $this->user, $this->pwd, $this->dbname);
        mysqli_set_charset($this->link, $this->charset);
    }
    //查询所有行方法
    public function select($sql){
        $result = mysqli_query($this->link, $sql);
        return mysqli_fetch_all($result, MYSQLI_ASSOC);
    }
    //添加方法
    //....
}
```

类中的各项连接参数及连接数据库的方法都不希望在使用时被修改，所以就设置为私有的，而写这个类的目的就是为了能够使用里面的查询、添加等方法，所以 *select* 方法就是公开的，相当于 *Mysql* 类对外的一个接口。

> 只是我们是开发者，所以你可能觉得写成 *public* 或者 *private* 都无所谓，写成什么还不是取决于你的一念之间，但是当你把这个类提供给别人使用的时候，你希望什么？是不是希望他能够调用你的 *select* 方法就可以了，而不希望他把你的类该的乱七八糟的。

19、重载（overloading)

第一句话是，不要把重写(override)和重载(overload)弄混了。

前面学过的重写是子类继承父类时，重写了父类的成员，父子类中存在同名方法，而且参数一致。

而重载的本意是一个类中，有两个同名方法，但是他们的参数不同，Java语言中的重载就是这样的。

但是PHP的重载和Java的重载不太一样。PHP所提供的重载（overloading）是指动态地创建类属性和方法。我们是通过魔术方法（magic methods）来实现的。

当调用当前环境下未定义或不可见的类属性或方法时，重载方法会被调用。本节后面将使用不可访问属性（inaccessible properties）和不可访问方法（inaccessible methods）来称呼这些未定义或不可见的类属性或方法。

什么是不可见？有两种情况，一种是成员私有或受保护，在类的外面无法访问；另一种情况是根本就没有访问的成员。看下面的代码：

```php
class Test {
    private $a = 1;
    protected $b = 2;
    private function aa(){
        return 'aa';
    }
}
$t = new Test();
echo $t->a; // 访问不可见成员属性，因为属性私有
echo $t->aa(); // 访问不可见成员方法，因为方法私有
$t->c = 'abc'; // 访问对象不存在的成员，也叫做访问不可见成员
$t->bb(); // 访问对象不存在的方法，也叫做访问不可见成员
```

19.1 属性重载

可以先考虑一个问题，在类的外部无法直接访问到类的私有属性，那么有没有间接的办法呢？看下面的代码：

```php
class A {
    private $x = 1;
    private $y = 2;
    //定义公开的方法，返回私有属性的值
    public function get($name){
        return $this->$name;
    }
}
$a = new A();
echo $a->get('x'); // 1
```

间接的拿到类里面私有属性的办法就是定义一个公开的方法，让这个方法去拿私有属性。

上面的 *get* 方法是自己定义的，其实PHP内置了很多**魔术**方法，可以针对不同情况下的属性重载，这些魔术方法有 *__get()*、*__set()*、*__isset()* 、*__unset()* 。这些魔术方法在定义时，**必须是非静态的公开的方法。**

19.1.1__get()

读取不可访问属性的值时，*__get* 被调用。该方法有一个参数，参数为调用的属性名。

```php
class A {
    private $x = 1;
    //定义魔术方法__get，返回私有属性的值
    public function __get($name){
        return $this->$name;
    }
}
$a = new A();
echo $a->x; //此时，访问了不可见成员x，所以__get被触发。所以输出 1
```

19.1.2__set()

在给不可访问属性赋值时，*__set* 被调用。该方法有两个参数，第一个参数为属性名，第二个参数为属性

```
class A {
    private $x = 1;
    //定义魔术方法__get，返回私有属性的值
    public function __get($name){
        return $this->$name;
    }
    //定义魔术方法__set，设置私有属性的值
    public function __set($name, $value){
        $this->$name = $value;
    }
}
$a = new A();
$a->x = 2; //此时，设置了不可见属性的值，所以__set被触发。
echo $a->x; //此时，访问了不可见成员x，所以__get被触发。所以输出 2
```

19.1.3__isset()

当对不可访问属性使用 *isset()* 或 *empty()* 函数的时候，*__isset()* 方法被触发。该方法有一个参数，参数为属性名

```
class A {
    private $x = 1;
    //定义魔术方法__get，返回私有属性的值
    public function __get($name){
        return $this->$name;
    }
    //定义魔术方法__set，设置私有属性的值
    public function __set($name, $value){
        $this->$name = $value;
    }
    //定义魔术方法__isset
    public function __isset($name){
        var_dump(isset($this->$name));
    }
}
$a = new A();
if(isset($a->x)){ //当对不可见成员使用isset函数的时候，魔术方法__isset()被触发

}
```

19.1.4__unset()

当对不可访问属性使用 *unset()* 函数的时候，*__unset()* 方法被触发。该方法有一个参数，参数为属性名

```
class A {
```

```
    private $x = 1;
    //定义魔术方法__get，返回私有属性的值
    public function __get($name){
        if(isset($this->x)){
            return $this->$name;
        }
        return null;
    }
    //定义魔术方法__isset
    public function __unset($name){
        unset($this->$name);
    }
}
$a = new A();
unset($a->x);
var_dump($a->x);
```

解决一个疑问，为了体现封装性，将类中的成员都设置为私有了，但是又提供了诸如 *__set* 等方法，那么为什么还设置私有属性，直接设置成公开属性不就可以了吗？

答：设置为私有确实是为了体现面向对象的封装特性，尽量隐藏内部的实现，只提供对外的接口给用户。但是设置了 *__set()* 就一定要设置私有成员的值吗？不一定。我们在写 *__set* 方法时，可以加以判断。看下面的代码。

```
class A {
    private $x = 1;
    private $y = 2;
    //定义魔术方法__get，返回私有属性的值
    public function __get($name){
        //控制，只允许获取私有的x属性
        if($name == 'x'){
            return $this->$name;
        }
        return null;
    }
    //定义魔术方法__set，设置私有属性的值
    public function __set($name, $value){
        //控制，如果用户登录了就允许设置私有属性，反之记录错误信息
        session_start();
        if(isset($_SESSION['username'])){
            $this->$name = $value;
        }else{
            ini_set('error_log', './error.log');
            error_log(' --- 有人访问了'. $name. "\n");
        }
    }
}
$a = new A();
$a->x = 2; //此时，设置了不可见属性的值，并且没有登录，所以此次修改失败，并记录日志
echo $a->x; //此时，访问了不可见成员x，所以__get被触发。有因为前面没有修改成功，所以输出 1
```

19.2 方法重载

19.2.1__call

当访问对象不可见方法的时候，*__call* 被调用，该方法有两个参数，参数一是调用的方法名，参数二是数组，表示传递给被调用方法的参数。

```php
class A {
    public function __call($name, $arguments)
    {
        echo $name . '<br>';
        var_dump($arguments);
    }
}
$a = new A();
$a->aa(); //输出方法名aa，打印出来空数组
$a->aa(1,2,3); //输出方法名aa，打印出数组[1,2,3]
```

19.2.2__callStatic

当访问一个静态方法的时候，*__callStatic()* 被触发，该方法是所有魔术方法中唯一一个必须为静态的方法。该方法有两个参数，参数一是调用的方法名，参数二是数组，表示传递给被调用方法的参数。

```php
class A {
    //该魔术方法必须为静态
    public static function __callStatic($name, $arguments)
    {
        echo $name . '<br>';
        var_dump($arguments);
    }
}
A::aa(); //输出方法名aa，打印出空数组
A::aa(1,2,3); //输出方法名aa，打印出数组 [1,2,3]
```

20、面向对象多态性

多态也是面向对象的一大特性。

和重载一样，PHP的多态和Java中的多态也是不一样的，甚至可以说PHP中没有多态。如果非要说PHP中的多态，可以使用方法重载的知识来完成。

PHP中的多态可以理解为，**继承**一个类后，子类**重写**了父类的方法，我们通过**子类的对象来**灵活**调用**父子两个类中的同名方法的办法。

```php
class A {
    public function aa($argu){
        echo '父类方法';
        var_dump($argu);
    }
}
class B extends A {
    //子类重写了父类的aa方法
    public function aa($argu){
        echo '子类方法';
        var_dump($argu);
    }
}
$b = new B(); //实例化子类
//如何通过$b调用父类的aa和子类的aa呢?
```

要解决上面的问题，就要使用方法重载了。而实现了这种动态的调用，就是多态了。

```php
class A {
    public function aa($argu){
        echo '父类方法';
        var_dump($argu);
    }
}
class B extends A {
    //子类重写了父类的aa方法
    public function aa($argu){
        echo '子类方法';
        var_dump($argu);
    }
    //魔术方法，判断调用的是什么方法，然后由__call决定调用父类的aa还是子类的aa方法
    public function __call($name, $arguments)
    {
        if($name == 'a1'){
            $this->aa($arguments);
        }elseif ($name == 'a2'){
            parent::aa($arguments);
        }
    }
}
$b = new B(); //实例化子类
//如何通过$b调用父类的aa和子类的aa呢?
$b->a1(1,2); //实际上相当于调用了子类的aa方法
$b->a2(3,4); //实际上相当于调用了父类的aa方法
```

21、设计模式之工厂模式

工厂，即提供原材料，可以（批量）生产成品的地方。

工厂模式，即创建一个工厂类，具体的说是工厂类中有一个方法，为这个方法提供类名（原材料），就可以得到该类的对象（成品）。

工厂模式一般适用于对象有共同的属性或者方法的场合。举例说明：

```php
class A {
    //A类中有aa方法
    public function aa(){
        echo 'aa';
    }
}
class B {
    //B类中也有aa方法
    public function aa(){
        echo 'bb';
    }
}
//工厂类
class Factory {
    //静态方法，用于生产对象，参数为类名
    public static function getInstance($clas_name){
        return new $clas_name();
    }
}
//
$obj = Factory::getInstance('A');
//$obj = Factory::getInstance('B');
$obj->aa();
```

从上述代码中看出，有工厂类的好处是，简化调用。无论A类的对象，还是B类的对象，都叫做 *$obj* ，而且都有 *aa()* 方法，所以调用方法 *aa()* 这行代码永远不需要变化，需要调用A类的方法，就为工厂类提供类名A，需要调用B类的方法，就为工厂类提供类名B。

再看一个案例：四则运算案例。

```php
//定义运算接口，里面要求完成计算方法
interface Yunsuan {
    function jisuan($a, $b);
}
//加法类，实现接口，完成加法计算
class Jia implements Yunsuan {
    public function jisuan($a, $b){
        return $a + $b;
    }
}
//减法类，实现接口，完成减法运算
class Jian implements Yunsuan {
    public function jisuan($a, $b){
```

```
        return $a - $b;
    }
}
//工厂类
class Factory {
    public static function getInstance($clas_name){
        return new $clas_name();
    }
}
//
$obj = Factory::getInstance('Jia');
//$obj = Factory::getInstance('Jian');
echo $obj->jisuan(5, 3);
```

此例中，无论执行加法还是减法计算，都调用 *jisuan()* 方法，而到底执行什么运算取决于传递给工厂类的类名。

22、序列化和反序列化

22.1 序列化和反序列

序列化是将变量转化为特殊字符串的过程，反序列化是将特殊字符串转化为原来变量的过程。

为什么需要序列化或反序列化呢？不妨先看一个问题。如何将一个 *bool* 类型的 *true* 存放到记事本中或数据库中？

```
$a = true;
file_put_contents('a.txt', $a); //直接将变量存储到文件中
```

执行代码后，打开 *a.txt* ，发现 *a.txt* 中存储的是数字 *1* 。

问题来了，你如何知道 *a.txt* 存储的是数字 *1* 还是布尔型的 *true* 呢？要解决这个问题，只有将变量序列化，然后再将其存储。

使用 *serialize()* 将一个变量序列化。

```
$a = true;
file_put_contents('a.txt', serialize($a)); //将变量序列化之后，然后存储到文件中
```

这时，执行代码后，打开 *a.txt* ，发现 *a.txt* 中存储的是一个特别的字符串 *b:1;* 。字符串中的 *b* 表示布尔类型的意思，而 *1* 表示 *true* 的意思，这样我们就知道了文件中存储的是布尔型的 *true* 而不是整型的 *1* 。

> 我们都知道，当一段代码执行完毕后，会释放这个变量，而有了序列化，我们就可以将一个变量永久的存储（存文件或存数据库）了，这就是序列化的意义。

知道了什么是序列化，反序列化就变得容易理解了，它是将存储的特殊字符串转化为原来的变量的过程。

使用 *unserialize()* 完成反序列化。

```php
//$a = true;
//file_put_contents('a.txt', serialize($a)); //将变量序列化，然后存文件
$b = file_get_contents('a.txt'); //读取文件的内容
$b = unserialize($b); //反序列化
var_dump($b); //输出 bool(true)
```

> 序列化不只针对布尔类型的变量，我们可以将除资源以外的所有数据类型的变量执行序列化。不能序列化资源类型是因为资源属于外部得来的，而外部资源易便。比如 *mysql* 连接资源，假设进行序列化的时候密码是 *123*，过了一段时间将密码改为 *123456* 了，这个时候执行反序列化的结果已经不能够连接数据库了，因为密码已经改变了。

22.2 对象序列化和反序列化

我们拿之前学习过的 *Mysql* 类来举例。

Mysql.php 内容如下：

```php
//数据库操作类
class Mysql {
    private static $instance = null; //表示该类的对象
    private $host = 'localhost';
    private $user = 'root';
    private $pwd = '123';
    private $dbname = 'php68';
    private $charset = 'utf8';
    private $link = null; //存储连接数据库成功后的资源

    private function __construct(){
        $this->connect(); //调用连接数据库的方法
    }
    private function __clone(){}
    public static function getInstance(){
        if(self::$instance === null) {
            self::$instance = new self();
        }
        return self::$instance;
    }
    //连接数据库方法
    private function connect(){
        $this->link = mysqli_connect($this->host, $this->user, $this->pwd, $this->dbname);
        mysqli_set_charset($this->link, $this->charset);
    }
    //查询方法
    public function select($sql){
        $result = mysqli_query($this->link, $sql);
        return mysqli_fetch_all($result, MYSQLI_ASSOC);
    }
    //添加方法
    public function add($sql){
        return mysqli_query($this->link, $sql);
```

```
    }
    //...
}
```

我们新建一个文件，引入 *Mysql* 类，然后实例化对象，将对象序列化后存储到 *m.txt* 中。

```php
require_once 'mysql.php';
$m = Mysql::getInstance(); //调用getInstance方法，得到对象
file_put_contents('m.txt', serialize($m)); //将序列化后的结果存放到 m.txt 中
```

运行文件后，查看 *m.txt* 里面的内容（下面是m.txt里面的部分内容，其实这个内容是什么，我们无需关心，只是简单看一下即可）：

```
O:5:"Mysql":6:{
    s:11:" Mysql host";
    s:9:"localhost";
    s:11:" Mysql user";
    s:4:"root";
    s:10:" Mysql pwd";
    s:3:"123";
    s:13:" Mysql dbname";
    s:5:"php68";
    s:14:" Mysql charset";
    s:4:"utf8";
    s:11:" Mysql link";
    O:6:"mysqli":19:{
        s:13:"affected_rows";
        N;s:11:"client_info";
        N;s:14:"client_version";
        N;s:13:"connect_errno";
        N;s:13:"connect_error";
        ...
    }
}
```

下面另新建一个文件，获取 *m.txt* 里面的内容，并执行反序列化操作，检测一下反序列化后还是原来的对象吗？

```php
$a = file_get_contents('m.txt'); //获取存储的内容
$a = unserialize($a); //反序列化
echo "<pre>";
var_dump($a);
```

打印结果出现了错误，原因就是 *Mysql* 中的 *link* 是资源类型的值，只要在序列化对象的时候，明确存储哪些成员属性即可解决这一问题。

魔术方法 *__sleep()* ，在序列化对象时自动执行，*sleep()* 方法**需要返回一个一维索引数组**，数组中存在的单元就是序列化对象时要存储的成员属性。

所以，在 *Mysql* 中添加魔术方法 *sleep* .

```php
public function __sleep()
{
    return ['host','user', 'pwd', 'dbname', 'charset'];
}
```

加入该魔术方法后，我们再次执行一下序列化和反序列化，反序列化后发现没有错误了，打印反序列化后的对象如下：

```
object(__PHP_Incomplete_Class)#1 (6) {
  ["__PHP_Incomplete_Class_Name"]=>
  string(5) "Mysql"
  ["host":"Mysql":private]=>
  string(9) "localhost"
  ["user":"Mysql":private]=>
  string(4) "root"
  ["pwd":"Mysql":private]=>
  string(3) "123"
  ["dbname":"Mysql":private]=>
  string(5) "php68"
  ["charset":"Mysql":private]=>
  string(4) "utf8"
}
```

此时，没有了错误，但是发现该对象的类名是 *__PHP_Incomplete_Class* 了，而不是 *Mysql* 了，这是因为在反序列化的时候，还要引入 *Mysql* 类。修改代码：

```php
require_once 'mysql.php'; //引入Mysql类
$a = file_get_contents('m.txt'); //获取文件内容
$a = unserialize($a); //执行反序列化
echo "<pre>";
var_dump($a);
```

这时，打印的结果如下，就正确了。

```
object(Mysql)#1 (6) {
  ["host":"Mysql":private]=>
  string(9) "localhost"
  ["user":"Mysql":private]=>
  string(4) "root"
  ["pwd":"Mysql":private]=>
```

```
  string(3) "123"
  ["dbname":"Mysql":private]=>
  string(5) "php68"
  ["charset":"Mysql":private]=>
  string(4) "utf8"
  ["link":"Mysql":private]=>
  NULL
}
```

但是，打印不出错并不代表没有问题了，我们用反序列化后的对象调用 *select* 方法，完成数据查询：

```php
require_once 'mysql.php';
$a = file_get_contents('m.txt');
$a = unserialize($a);
echo "<pre>";
var_dump($a);
$data = $a->select('select * from article');
var_dump($data);
```

执行结果仍然是错误的，因为序列化的时候没有存储 *link* ，反序列化的时候 *link* 的值为 *null* ，表示没有成功连接数据库。解决此问题，需要使用魔术方法 *__wakeup()* 。

__wakeup() 方法在反序列的时候自动执行，我们只要在 *Mysql* 类中添加魔术方法 *wakeup()* ，并在该方法中完成数据库连接即可。

```php
public function __wakeup()
{
    $this->connect();
}
```

到此，我们就完成了对象的序列化和反序列化工作了。而且反序列化后的对象能够和以前一样调用 *select* 完成查询操作了。

23、trait

23.1 trait简介

自 PHP 5.4.0 起，PHP 实现了一种代码复用的方法，称为 trait（特点、特征）。

Trait 是为类似 PHP 的单继承语言而准备的一种代码复用机制。Trait 为了减少单继承语言的限制，使开发人员能够自由地在不同层次结构内独立的类中复用方法 method。Trait 和 Class 组合的语义定义了一种减少复杂性的方式，避免传统多继承和 Mixin 类相关典型问题。

trait 无法实例化，使用 trait 的类不需要有继承的关系。

trait 的声明和类一样，只不过 trait 中不可以定义常量，但可以定义抽象方法。

定义一个 trait：

```php
trait A {
    public $x = 1;
    private $y = 2;  //私有成员
    protected static $z = 3; //静态成员
    public function aa(){
        echo 123;
    }
    protected function bb(){
        echo $this->name;
    }
    private function cc(){
        echo 456;
    }
    public static function dd(){
        echo 789;
    }
    abstract public function ee();
}
```

23.2 trait由来和使用

trait 是怎么出现的呢？我们通过一段代码推导一下。假设有两个类，如下：

```php
class A {
    const ABC = 'hello';
    private $x = 1;
    public function __construct(){
        echo $this->x;
    }
    public static function aa(){
        echo 2;
    }
    private function cc(){
        echo 3;
    }
}

class B {
    const ABC = 'hello';
    private $x = 1;
    public function __construct(){
        echo $this->x;
    }
    public static function aa(){
        echo 2;
    }
    private function cc(){
        echo 4; //注意这里和A类的写法不一样
    }
```

```
}
```

我们发现，A 和 B 类里面的内容非常相似，这样写完两个类，代码非常多，我们可以将 A 类和 B 类中相同的内容抽离出来单独封装，封装的结果就是 trait， 这就是 trait 的由来。

修改上面的两个类：

```php
//将A类和B类中相同的内容，单独封装成trait
trait Same {
    private $x = 1;
    public function __construct(){
        echo $this->x;
    }
    public static function aa(){
        echo 2;
    }
}
class A {
    const ABC = 'hello';
    use Same; //相当于把剪切出去的内容，引入回来
    private function cc(){
        echo 3;
    }
}
class B {
    const ABC = 'hello';
    use Same; //相当于把剪切出去的内容，引入回来
    private function cc(){
        echo 4;
    }
}
```

在 A 类和 B 类中，使用关键字 *use* ，引入 trait。这样，类A和原来的A没有区别，用法也一样。类B也是如此。

```php
A::aa(); //输出2
$a = new A(); //输出 1，因为构造方法中输出 1
```

> 因为 trait 是抽离类中的部分内容而来的，所以 trait中的成员的可见性同样适用于使用 trait 的类。即trait 中的私有成员就相当于类中的私有成员，trait 中的受保护成员相当于类中的受保护成员，trait 中的公开成员相当于类中的公开成员。

```php
$a = new A();
echo $a->x; //报错，因为x是私有成员，类的外部不可见
```

注意事项：

- trait 定义了一个属性后，类就不能定义同样名称的属性，否则会产生 fatal error。

- Trait 定义的抽象方法，使用 trait 的类必须实现这些抽象方法

```php
trait Same {
    public $x = 1;
    abstract public function aa();
}
class A {
    use Same; //相当于把剪切出去的内容，引入回来
    public $x = 2; //错误，不能重写定义trait中的属性
    //必须实现trait中的抽象方法
    public function aa(){

    }
}
```

23.3 trait中可以使用另外的trait

正如 class 能够使用 trait 一样，其它 trait 也能够使用 trait。在 trait 定义时通过使用一个或多个 trait，能够组合其它 trait 中的部分或全部成员。

```php
trait A {
    public $x = 1;
    public function sayHello() {
        echo 'Hello ';
    }
}

trait B {
    //public $x = 2; // 因为在C中同时使用了A、B，有因为A中有 $x 属性了，所以这里不能重写定义
    public function sayWorld() {
        echo 'World!';
    }
}
//由上面两个trait组合成一个trait
trait C {
    use A, B;
}

class D {
    use C;
}

$o = new D();
$o->sayHello();
$o->sayWorld();
```

23.4 冲突的解决

上面的案例中，如果 trait A和B 中 存在同名方法，我们该如何解决呢？可以使用 *insteadof* 关键字来解决：

```php
trait A {
    public function sayHello() {
        echo 'Hello ';
    }
    public function aa(){
        echo 1;
    }
}

trait B {
    public function sayWorld() {
        echo 'World!';
    }
    public function aa(){
        echo 2;
    }
}

trait C {
    use A, B{
        A::aa insteadof B; //A中的aa方法代替B中的aa，所以使用的是A中的aa方法
        //B::aa insteadof A; //B中的aa方法代替A中的aa，所以使用的是B中的aa方法
    }
}
class D {
    use C;
}
$o = new D();
$o->sayHello();
$o->sayWorld();
```

这样，通过对象 *$o* 调用的 aa 方法就是 A 中的 aa 方法了。但是 B 中的 aa 方法呢？无法调用了，我们可以使用另一种方法解决上面的冲突，那就是使用 *as* 给冲突的方法设置别名。

```php
//修改上面代码中的 C
trait C {
    use A, B{
        A::aa insteadof B; //必须指定A中的aa还叫做aa
        B::aa as bb; //然后才可以给B中的aa起别名
    }
}
```

解决了冲突，通过 *$o* 就可以调用 aa 或 bb方法了。

> 在trait中可以使用另外的多个 trait，当然在类中也可以使用多个 trait，方法都是一样的。

23.5 修改访问控制

类使用 trait 后，可以使用 *as* 修改方法的访问控制。

```php
trait A {
    public function sayHello() { // trait 中是公有方法
        echo 'Hello World!';
    }
}

// 修改 sayHello 的访问控制
class B {
    use A {
        sayHello as protected; //将trait引入到B类中，修改方法sayHello为受保护的
    }
}

// 给方法一个改变了访问控制的别名
// 原版 sayHello 的访问控制则没有发生变化
class C {
    use A {
        //将trait引入到C类中，修改方法sayHello为私有的，并修改方法名为myPrivateHello
        sayHello as private myPrivateHello;
    }
}
$b = new B();
//$b->sayHello(); //报错，方法受保护
$c = new C();
//$c->myPrivateHello(); //报错，方法私有
```

23.6 继承的类中使用trait

一个类 B 继承了另一个类 A，同时类 B 使用了trait C ， 则 子类B 的成员覆盖了trait的方法，而 trait 则覆盖了父类 A 的方法。

通过子类对象调用方法的时候，优先级：子类中方法 > trait中方法 > 父类方法

```php
//定义trait C
trait C {
    public function cc(){
        echo 'C - cc';
    }
}
//定义A类
class A {
    public function cc(){
        echo 'A - cc';
    }
```

```php
}
//B类继承A类
class B extends A {
    public function cc(){
        echo 'B - cc';
    }
    use C;
}
$b = new B();
$b->cc(); //此时调用的是B类中的cc方法
```

24、类型约束

php是弱类型语言，声明的变量无需指定数据类型。

PHP5开始可以在声明函数的时候，约束参数是哪种数据类型。

PHP7和PHP5一样，可以对函数的参数的数据类型进行约束，但是较PHP5没有严格。

使用PHP7的强类型模式，则类型约束的结果和PHP5中一致。

推荐只对数组和对象类型的参数进行约束，其他数据类型最好不要约束（官方文档说类型约束不能用于标量类型，但实测可以）。

```php
//常规的函数声明
function test($a, $b){
    echo '<pre>';
    var_dump($a, $b);
}
```

上面的代码，调用函数 *test()* 的时候，可以为其传递任何数据类型的值。

下面声明函数，并对函数的参数进行类型约束：

```php
declare(strict_types=1); //开启强类型模式，这样PHP5和PHP7执行结果一致
//声明函数test，要求传递给test的第一个参数必须是整型，第二个参数必须是布尔型
function test(int $a, bool $b){
    echo '<pre>';
    var_dump($a, $b);
}
test(123, true); //正确
test(456, 'hello'); //错误，因为第二个参数必须是布尔型
test(123, []); //错误，因为第二个参数必须是布尔型
```

> 上面的代码只有开启强类型模式，即加入 *declare(strict_types=1);* 才会和PHP5版本执行结果一致。另外官方文档上说类型约束不能用于**标量类型**，实测可以，所以不建议约束参数为标量类型。

我们在看一个约束参数必须为数组或某个类的对象的例子：

```php
//声明test函数，要求第一个参数必须为数组，第二个参数必须为A类的实例（A类的对象）
function test(array $a, A $b){
    echo '<pre>';
    var_dump($a, $b);
}
//声明 A 类
class A{

}
//声明 B 类，继承A类
class B extends A {

}
//声明 C 类
class C{

}
//调用test函数
test(['x','y'],  new A()); //正确
test(['x','y'],  new B()); //正确
test(['x','y'],  new C()); //错误
```

> 上面的例子即使不开启强类型模式，PHP5和PHP7的执行结果都是一样的，也就是说PHP支持约束参数为数组或某个类的对象。

在函数中可以对参数进行类型约束，在类的方法中同样可以对象参数进行类型约束。

> 其实，除了可以对数组和对象进行类型约束，还可以对两种特殊类型进行约束，一种是递归类型，一种是回调类型，不过我们作为了解就可以了。

25、遍历对象属性

使用 *foreach* 结构可以对 对象进行遍历，在类的外部，遍历只能得到对象的可见属性；在类的内部遍历可以得到对象的所有属性。

```php
class A {
    public $x = 1; //共有属性
    protected $y = 2; //受保护的属性
    private $z = 3; //私有属性
    public static $a = 4;  //静态属性

    public function __construct(){
        //构造方法，在类的内部遍历对象的属性
        foreach ($this as $key=>$value){
            echo "{$key} -- {$value} <br>";
        }
    }
}

```

```php
$a = new A();
//实例化的时候输出:
/*
x -- 1
y -- 2
z -- 3
*/
echo '<hr />';
//类的外部遍历对象的属性
foreach ($a as $key=>$value){
    echo "{$key} -- {$value} <br>";
}
//输出
/*
x -- 1
*/
```

> 静态成员属于类，不属于对象，所以无论在类的内部还是外部都不能通过遍历对象得到。

26、对象相关魔术常量及魔术方法

26.1 其他魔术方法

前面已经把大部分魔术方法都学习了，还有 *__invoke()* 和 *__toString()*：

__invoke()：当把对象当做函数来调用的时候，会触发这个魔术方法。

__toString()：把对象当做字符串调用的时候，会自动触发。**该方法必须返回一个字符串类型的值。**

```php
class A {
    public function __invoke()
    {
        echo '你搞错了，我是对象，不是函数';
    }
    public function __toString()
    {
        return '你不能有echo输出我，因为我是对象'; //该方法必须返回一个字符串
    }
}

$a = new A();
$a(); //把对象当做函数调用了，所以输出“你搞错了，我是对象，不是函数”
echo $a; //把对象当做字符串调用了，所以输出“你不能有echo输出我，因为我是对象”
```

26.2 魔术常量

前面我们学习过几个魔术常量：

- **LINE**： 文件中的当前行号。
- **FILE** 文件的完整路径和文件名。
- **DIR** 文件所在的目录。它等价于 dirname(**FILE**)。除非是根目录，否则目录中名不包括末尾的斜杠。

- **FUNCTION** 函数名称。

有关面向对象的魔术常量如下

- **CLASS** 类的名称。如果类在命名空间中，则该常量包含命名空间。自 PHP 5.4 起 trait 中的**CLASS** 是调用 该 trait 的类的名字。
- **TRAIT** Trait 的名字。如果有命名空间，则该常量包含命名空间。
- **METHOD** 类的方法名。返回该方法被定义时的名字，形如 “A::a”。 如果有命名空间，则该常量包含命名空间。
- **NAMESPACE** 当前命名空间的名称。有关命名空间的知识，在后面讲。
- DIRECTORY_SEPARATOR表示两个目录直接的那条斜线，windows下是反斜线（\），linux下是正斜线（/）

27、类和对象相关函数

这部分知识具体可以查看官方文档：函数参考->变量与类型相关扩展->类/对象的信息->类/对象函数。

bool **class_exists** (string `$class_name` [, bool `$autoload` = true]) -- 检查类是否存在

bool **method_exists** (mixed `$object` , string `$method_name`) -- 检查方法是否存在

bool **property_exists** (mixed `$class` , string `$property`) -- 检查属性是否存在

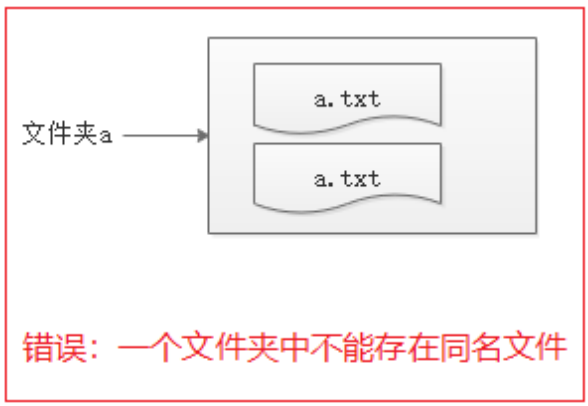
string **get_class** ([object `$object` = **NULL**]) -- 获取对象的类名

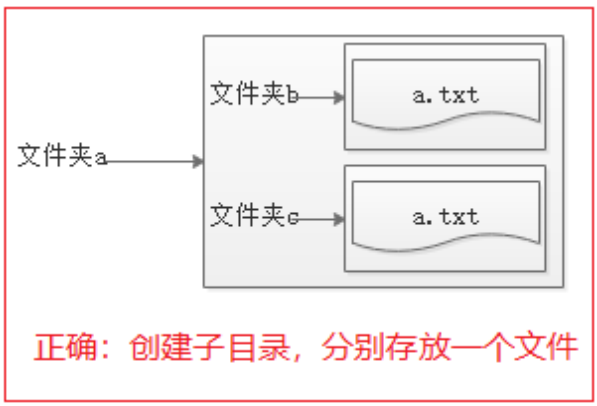
instanceof -- **判断对象是否是该类的对象**

命名空间

1、命名空间

这里提到的空间如同文件系统中的文件目录结构。试想，在同一个文件夹下能够存在两个同名文件吗？答案是不能的，但是可以用子目录的方式解决此问题。





同理，在同一个php文件中，能够定义同名的两个常量吗？能够定义同名的两个函数吗？能够定义同名的两个类吗？答案是不能的，可以使用命名空间解决此问题。

通过定义命名空间（namespace），给类、常量、函数提供不同的虚拟空间，来划分其作用的空间，避免相互冲突。同时可以用来避免自定义函数与系统函数的冲突。

> 这里的常量指的是由 *const* 定义的，而不能是 *define* 定义的。

使用 *namespace* 定义命名空间：

```
//定义命名空间 b
namespace b;
class Test{

}
const ABC = 'hello';
function aa(){

}
//定义命名空间 c
namespace c;
class Test{

}
const ABC = 'hello';
function aa(){

}
```

上面代码由于命名空间的存在，使得在同一个文件中同时存在同名类、函数、常量变的可行。

> 如同文件系统有根目录，命名空间也有根空间，也就是最大的空间。我们定义的所有空间都存在根空间中。开头的反斜线 “\” 表示根空间。

命名空间也可以像类那样使用大括号规定它的区域，不过这种写法比较少。

```
namespace b {
    class Test{

    }
    const ABC = 'hello';
    function aa(){

    }
}
```

```php
namespace c {
    class Test{

    }
    const ABC = 'hello';
    function aa(){

    }
}
```

受命名空间影响的只有类、函数和由 *const* 定义的常量。

实际开发中，一个文件只存放一个类，命名空间的定义要放到文件最开始的位置，即使*header*也不要在命名空间定义之前输出。

比如，两个文件

a.php 代码如下：

```php
namespace Home;
class A {

}
```

b.php 代码如下：

```php
namespace Admin;
include 'a.php';
class A {

}
```

前面说到，命名空间如同文件目录结构。既然文件夹可以有子文件夹，那么命名空间可不可以有子空间呢？答案是可以的。

```php
namespace A\B;
class Test {

}
```

上面的代码直接声明了 A 下面的 B 空间，则 A 空间自动创建。要注意的是，“A\B” 不能写成 “A/B” 。

2、访问空间中的内容

访问空间中的内容，意思为，如何使用命名空间中的内容？例如，如何实例化命名空间中的类？如何调用命名空间中的函数和常量？我们可以分为下面三种情况。

2.1 非限定名称访问

这种情况是在当前的命名空间中，直接使用类、函数或常量即可，和没有命名空间时的使用方式一样。

```php
namespace Blue;
class Top {
    public $name = '石头人';
}
$a = new Top(); //类和使用类的代码在同一个命名空间中，则和没有命名空间时的用法一样
echo $a->name;
```

2.2 限定名称访问

在有父子关系的两个空间中，**在父空间中调用子空间的类，使用这种方式。**

这种方式如同文件系统中的相对路径。

```php
//定义子空间
namespace A\B;
class T1 {
    public $name = '石头人';
}

//定义父空间
namespace A;
class T1 {
    public $name = '老司机';
}
//在父空间（A）中，调用用子空间（B）中的类
$o = new B\T1(); //直接指定子空间下面的类，如同文件系统中的相对路径。
echo $o->name; //输出 石头人
```

2.3 完全限定名称访问

毫无关系的两个空间，一个空间调用另一个空间中的内容，使用这种方式。这种方式用的比较多。

这种方式如同文件系统中的绝对路径。

```php
//定义B空间
namespace B;
class T1 {
    public $name = '石头人';
}
//定义A空间
namespace A;
class T1 {
    public $name = '老司机';
}
```

```php
//使用A空间的类，直接使用
$obj = new T1();
echo $obj->name; //输出 老司机

//在A空间中调用用B空间中的类
$o = new \B\T1(); // 最前面的反斜线表示根空间
echo $o->name; //输出 石头人
```

3、引入其他空间

还是两个毫无关系的空间，其中一个空间如何调用另外空间内容的问题。其他这个问题前面的完全限定名称访问已经说过了，这里在提供一种使用方式。这种方式就是使用 *use* 引入其他空间中的类到当前空间。

```php
//B空间中的类为T
namespace B;
class T {
    public $name = '石头人';
}
//A空间中的类为K
namespace A;
class K {
    public $name = '老司机';
}
//使用自己空间中的类，直接使用
$obj = new K();
echo $obj->name; //输出 老司机

//在A空间中引入 B空间中的类T
//use \B\T; //在A空间中引入 B空间中的类T
use B\T; //在A空间中引入 B空间中的类T，这样引入和上一行的引入是一个意思，建议使用这种方式
$o = new T();
echo $o->name; //输出石头人
```

> *use* 引入空间中的类的时候，前面的斜线（\）可以省略，因为use默认从根空间开始查找。而且建议不写最前面的斜线。

问题是，如果两个空间（B和A）中的类同名怎么办？答案是为避免两个同名类冲突，需要为引入的类起别名。

```php
//B空间中的类为T
namespace B;
class T {
    public $name = '石头人';
}
//A空间中的类也为T
namespace A;
class T {
```

```php
    public $name = '老司机';
}
//使用自己空间中的类，直接使用
$obj = new T();
echo $obj->name; //输出 老司机

//在A空间中引入 B空间中的类T
use \B\T as T1; //在A空间中引入B空间中的类T，定义别名为T1
$o = new T1();
echo $o->name; //输出石头人
```

> 上述所有案例，为了方便，都是在一个文件中演示的。其实，一般情况下，一个文件中只定义一个类。但无论情况是怎样的，使用命名空间的方式都是不变的，唯一要注意的是，引入文件（include_once）和引入命名空间（use）完全是两回事。

比如两个文件夹（a 文件夹和 b 文件夹）在同级目录中，两个文件夹中各有一个文件。

a 文件夹中的 *aa.php* 代码：

```php
namespace A;
class aa {
    public $name = 'A-test';
}
```

b 文件夹中的 *bb.php* 代码，这个文件中要使用 a 文件夹中的 *aa.php* 中定义的类，代码如下：

```php
namespace B;
require_once '../a/aa.php'; //如果不引入文件，只有下一行的use将报错。除非有自动加载。
use \A\aa;
class bb {
    public $name = 'B-test';
}
//使用bb类
$b = new bb();
//使用aa类
$a = new aa();
```

4、全局空间

我们还有一种情况没有讨论，如果一个类 A 没有定义在任何命名空间里，另一个类 B 定义在命名空间 Home 中，则在 Home 空间如何使用 A 类呢?

首先明确一个概念，**没有定义在命名空间中的类，则相当于在全局空间中**。

那么问题解决了，使用全局空间 （\），即可调用到类 A。

```
```

```php
//声明A类，它没有定义在任何命名空间中，则相当于是全局空间中的类
class A {
    public $name = 'a';
}
//定义命名空间 Home
namespace Home;
class B {
    public $name = 'b';
}
$b = new B(); //使用当前空间中的类，直接使用
echo $b->name; //输出 b

$a = new \A(); //调用全局空间中的类A
echo $a->name; //输出 a

```

另外一种情况，当前的文件没有定义命名空间，如何使用一个含有命名空间的类?

a.php 文件内容：

```php
namespace Home;
class A{
    public $name = 'aa';
}
```

b.php 文件内容：

```php
include 'a.php'; //包含另外的文件
class B {
    public $name = 'bb';
}
$b = new B();
echo $b->name; //bb

//使用Home空间中的A类
use Home\A; //use的时候，可以不加前面的斜线，建议使用这种方式
//use \Home\A; //use的时候，可以加前面的斜线
$a = new A();
echo $a->name; //aa
```

本书完!