# 其他相关技术

## 类的自动加载

含义：

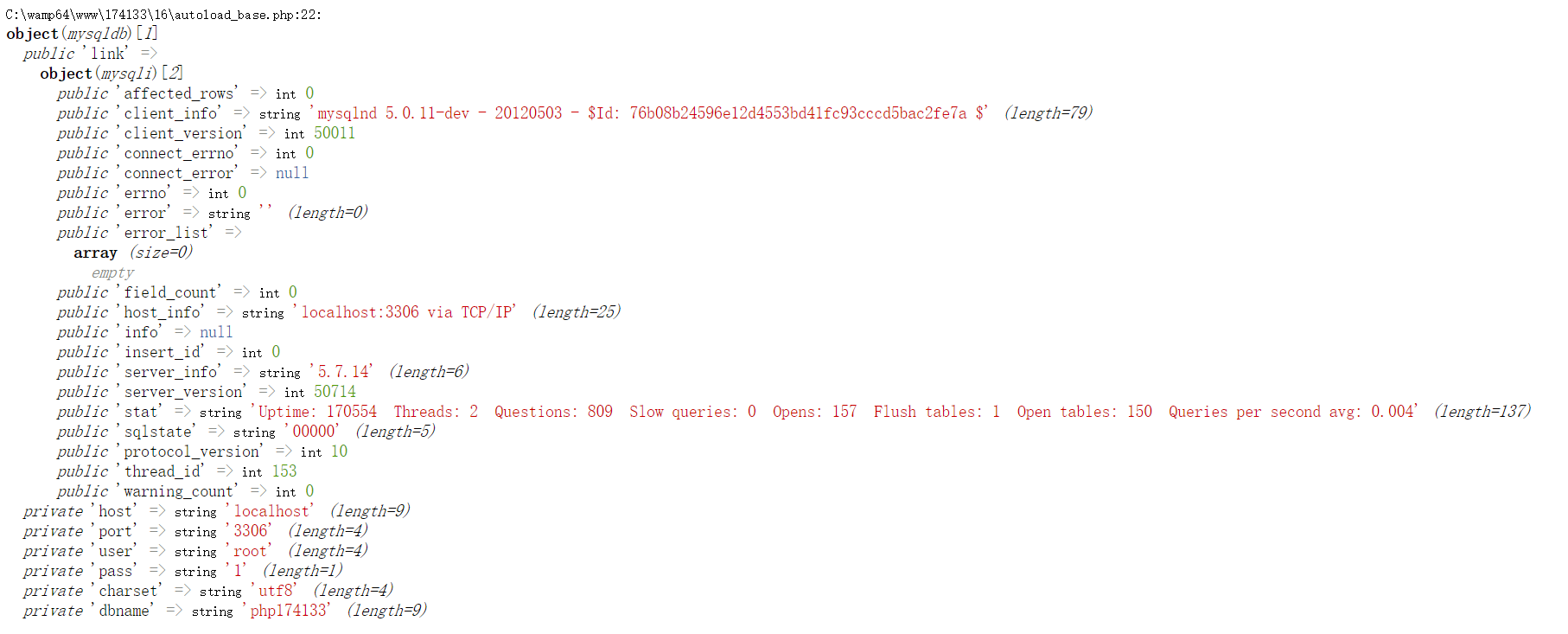
当某行代码需要一个类的时候，PHP的内部机制可以做到“自动加载该类文件”，以满足该行需要一个类的这种需要

什么时候需要一个类

* 1. New一个对象的时候
  2. 使用一个类的静态方法的时候
  3. 定义一个类（B）并以另一个类（A）作为父类的时候



结果



## 条件和要求

1. 当需要一个类的时候，就会自动调用某个函数（默认是\_\_autoload），并传入所需要类的名字
2. 一个类应该保存到一个独立的“类文件中”：即其中只有该类的定义，没有别的代码
3. 习惯上，类文件的类名要有一定的“规则”，通常是：类名.class.php
4. 通常，我们需要将各种类，存储在一些特定的目录中，以便确定其路径
5. 在该自动加载的函数中，“充分”使用传过来的类名，以构建一个合适的文件路径并载入

# 自定义自动加载函数

刚才，\_\_autoload函数，是系统内部的自动加载函数，我们只是定义其函数体

但：

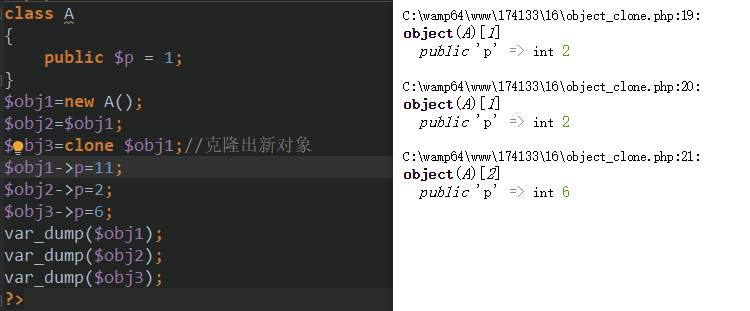
我们可以使用更多（自定义），来实现更灵活的自动加载



# 对象的赋值 克隆

对象的克隆语法就是用于将对象“制作”双份的语法；类似之前普通数据的值传递

$obj2=clone $obj1 //这样，就有了一个跟obj1一样的对象



# 对象的遍历

## 对象的遍历，跟数组的遍历一样

## 只能遍历出实例属性和值

Foreach($对象名as $ket=>$value){

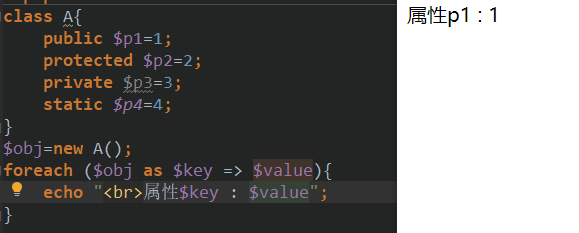
//处理key和value

//key表示对象的属性

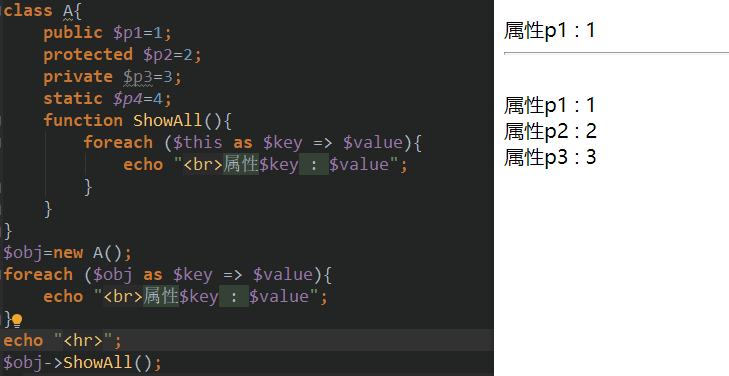
//value表示其对应值

//能够遍历处理的属性，只能是在该范围中的“可访问属性”

}



## 遍历里所有属性



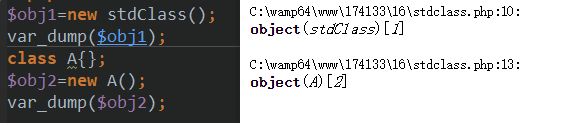
在类的内部可以访问所有级别的实例属性

# PHP内置标准类

PHP语言中，有很多现成的类 ，其中有一个，被称为“内置标准类”

这个类“内部”可以认为什么都没有，类似

Class A{ }



其作用，可以用于存储一些临时的简单数据:

$obj1->pp1=1;

$obj1->name=174133;

也可以于类型转换时，用于存储数据，如下所示

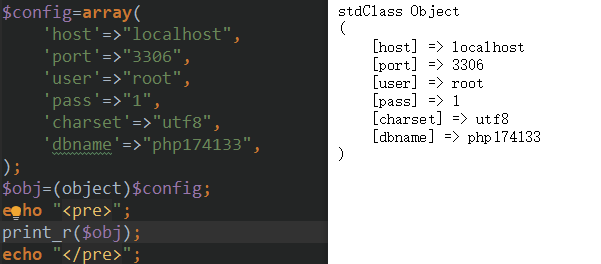
# 其他数据类型转换为对象类型

得到的结果是内置标准类stdclass的一个对象

语法：

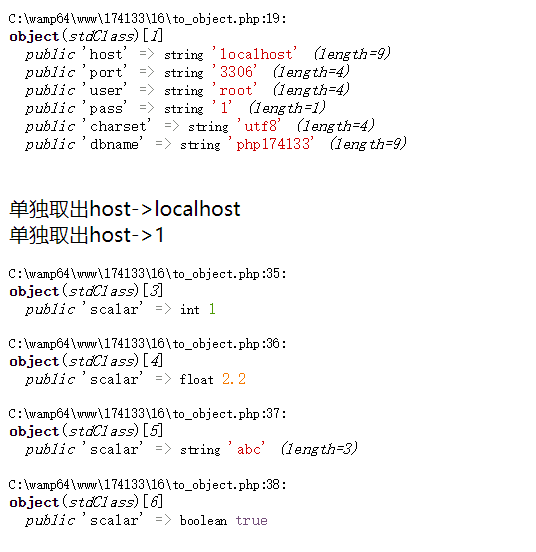
$obj1 = (objcet)其他类型数据

* 对象转换为对象：没有变化；
* 数组转换为对象：数组的键名当作属性名，值为对应值；
* null转换为对象：空对象；
* 其他标量数据转换为对象：属性名为固定的“scalar”，值为该变量的值





结果为



# 类型约束

要求某个变量只能使用（接收，存储）某种指定的数据类型

Php属于“弱类型语言”，通常不支持类型约束

相应的，强类型语言，类型约束却是其“基本特征”

## 只支持局部的部分类型约束

只支持在函数（或方法）的形参上，设定类型的约束目标，形式如下

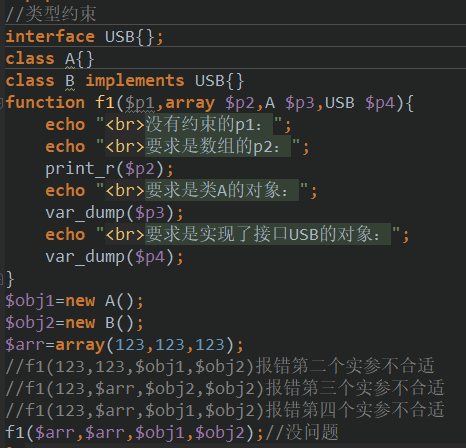
Function 方法名（【要求使用的数据类型】$形参）{

//……

}

说明：

1. 定义一个函数或方法时，一个形参，可以使用类型约束，也可以不使用
2. 如果使用的类型约束，则对应的实参数据，就必须是要求的类型
3. 能够使用的类型约束：
   1. 数组：Array
   2. Object：使用类的名称，表示，传递过来的实参，必须是该类的实例
   3. 接口：使用接口的名称，表示，传递过来的实参，必须是实现了该接口的类的实例



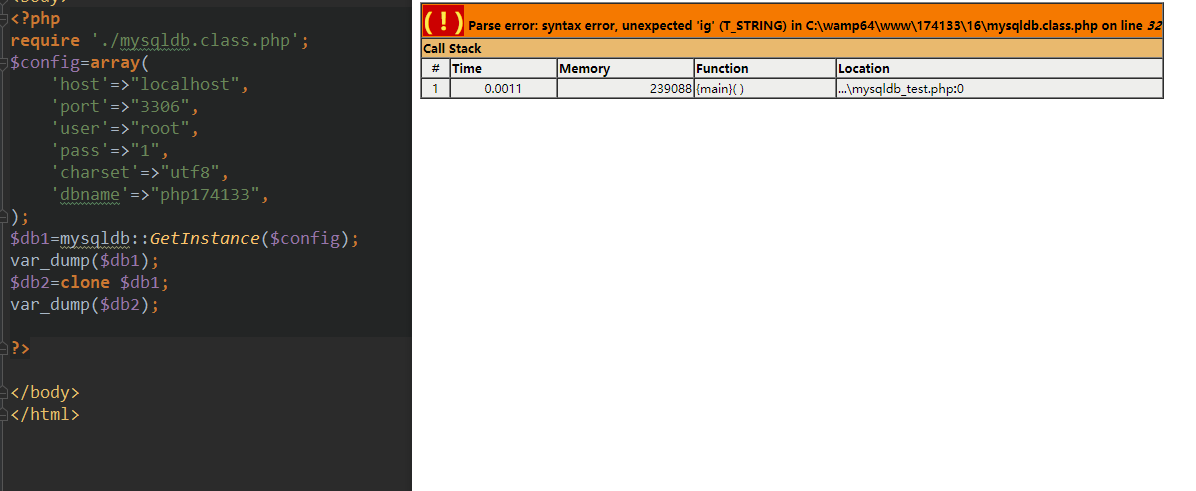
# 单例类的加强：禁止克隆

对于一个类的对象，如果使用clone运算符，就会克隆出一个跟当前对象完全一样的新对象，此时还会自动调用该类中的魔术方法\_\_clone（）只要其中有该方法；

则，要想实现单例类，就应该对这个单例类的对象“禁止克隆”

做法：

私有化这个魔术方法



# 与类有关的其他魔术方法

## 序列化与反序列化技术

含义：  
 序列化

就是将一个变量所代表的“内存”数据，转换为字符串形式并持久保存在硬盘上的一种做法  
 反序列化

就是将序列化之后保存在硬盘上的“字符串”数据，恢复为其原来的内存形式的变量数据的一种做法。

序列化做法：  
 $v1=123; //代表任意内存数据

$s1=serialize($v1) //将任何类型的变量数据转换为“字符串”

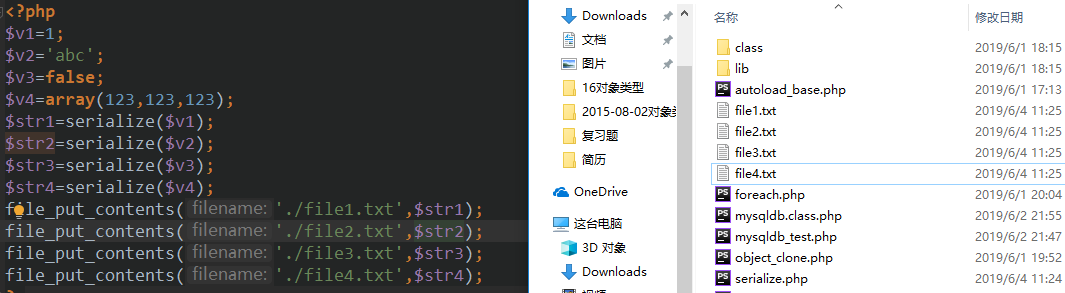
File\_put\_contents(‘要保存的目标文本文件’,，$s1)

反序列化做法：

$s1=File\_get\_contents(‘要保存的目标文本文件’) //从一个文件里读出其中的所有字符串

$v1=unserialize($v1) //将该字符串数据，反序列化转换为变量数据

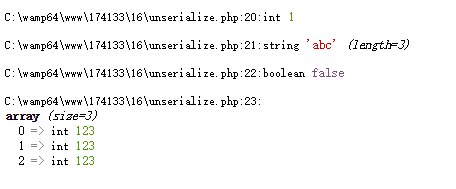
举例：基本数据的序列化



生成的这4个文件只能看把你改，因为里边内容改了就读取不出来

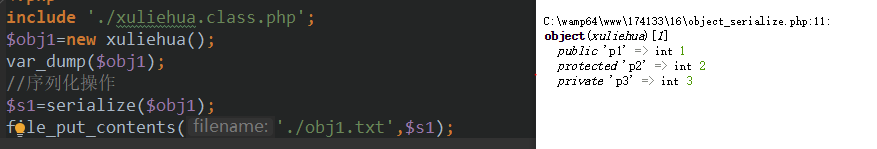
再反序列化：





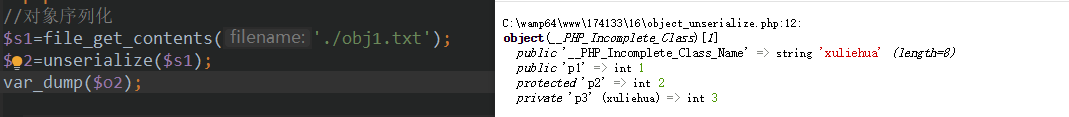
## 对象的序列化

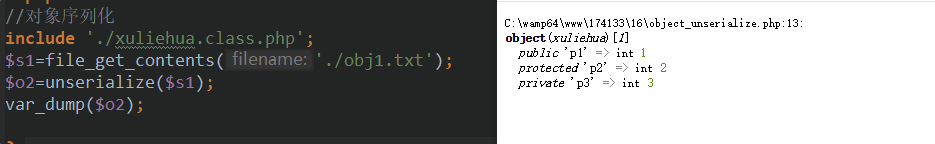
1. 对一个对象进行序列化，只能将其属性数据“保存起来”，而方法被忽略
2. 对象的序列化的时候，会自动调用该对象所属类的魔术方法：\_\_sleep（），此时，该方法必须返回一个数组，数组中是“计划”要进行序列化的属性名



## 对象的反序列化

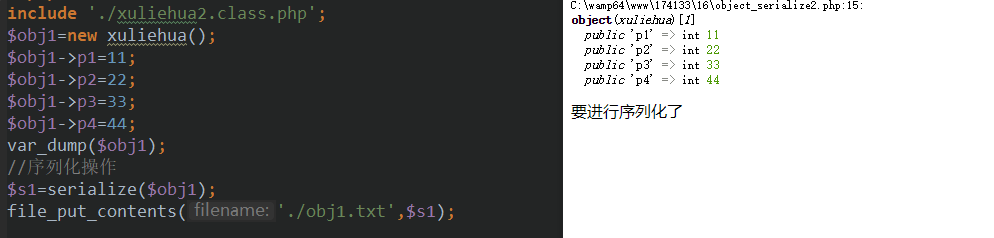
1. 对一个对象进行反序列化，其实是恢复其原来保存起来的属性数据，此时，必然需要依赖该对象原本的所属类
2. 对象的反序列化的时候，会自动调用该对象所属类的魔术方法：\_\_wakeup（），





下面演示可选属性的序列化（通过\_\_sleep）



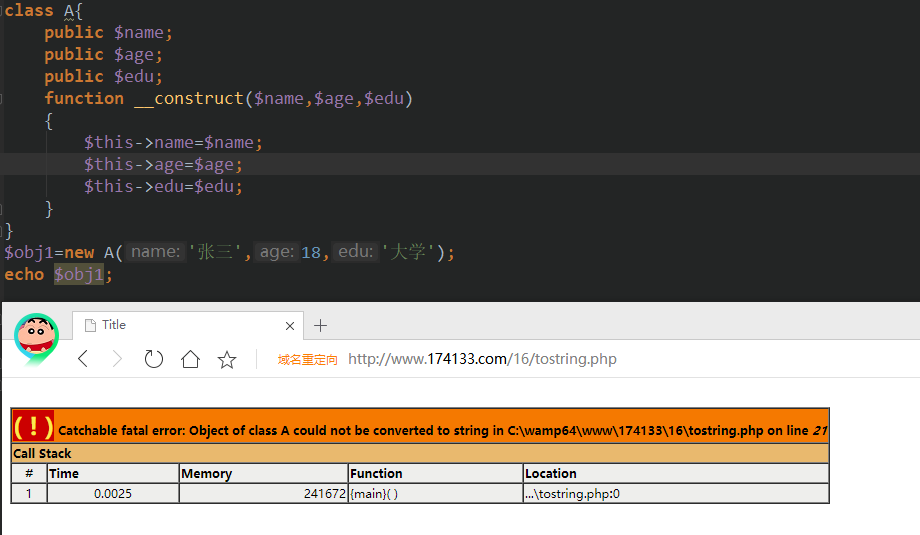


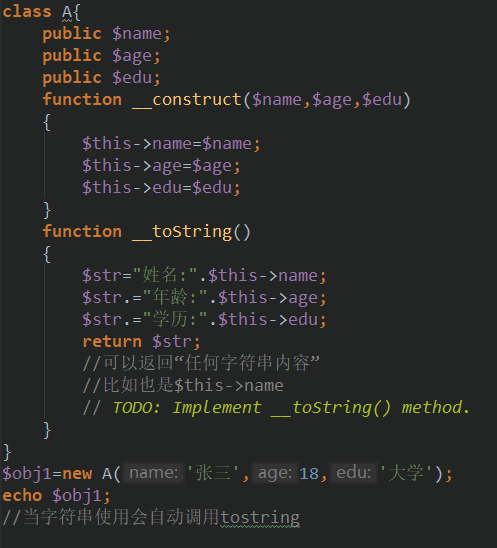


# \_\_tostring魔术方法 比较常用

含义：  
 将一个对象“当做”一个字符串来使用的时候，会调用该方法，并且在该方法中，可以返回一定的字符串，以表明该对象转换为字符串之后的结果

注意：  
 如果没有定义该方法，则对象无法当做字符串来使用！！







# Invoke（）魔术方法

将对象当作函数来使用到的时候，会自动调用该方法



# 其他

与类有关的魔术常量：

## 以前学过的

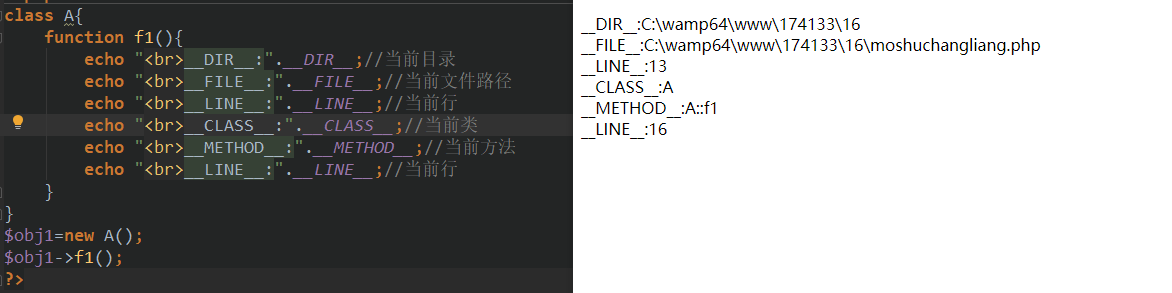
\_\_file\_\_

\_\_dir\_\_

\_\_line\_\_

现在：  
\_\_class\_\_ 当前所在的类

\_\_METHOD\_\_ 当前方法



## 与类有关的系统函数

Class\_\_exists（”类名”） 判断一个类是否存在（是否定义过）

Interface\_exists（”接口名”）判断一个接口是否存在（是否定义过）

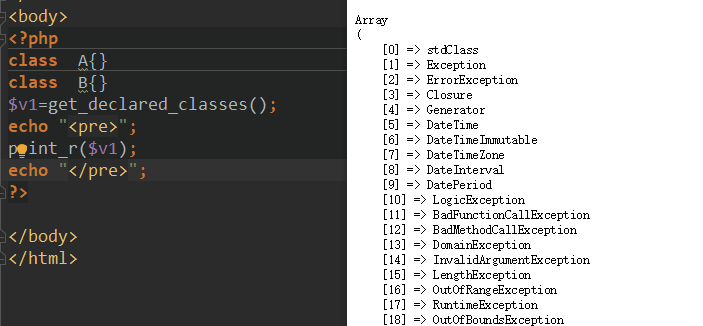
Get\_class（$obj） 获取某个对象$obj的所属类

Get\_parent\_class（$obj）获取某个对象$obj的所属类的父类

Get\_class\_methods（）获得一个类的所有方法名，结果是一个数组，存储方法名称

Get\_class\_vars（）获得一个类的所有属性名，结果是一个数组，存储属性名称

Get\_declared\_classes（）获得整个系统所定义的所有类名



## 与对象有关的系统函数：

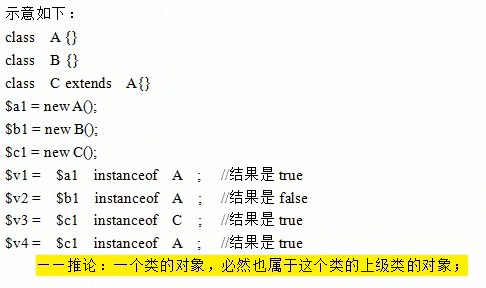
Is\_object（）//判断某个变量是否为一个对象

Get\_object\_vars（）//获得一个对象的所有属性，结果是一个数组

## 与类有关的运算符

New

Instanceof 判断一个“变量”（对象，数据），是否是某个类的“实例”



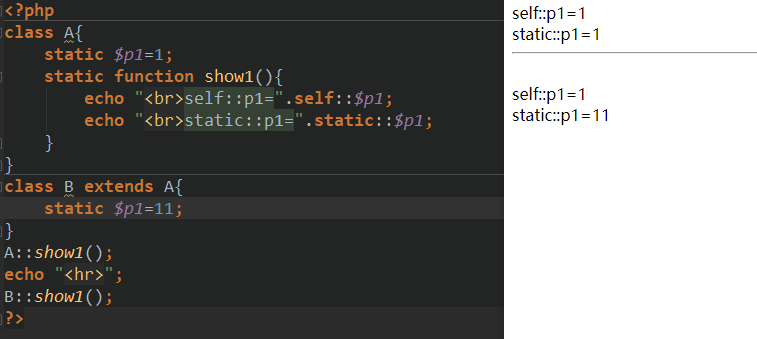
# Static关键字的新用法和总结

Static这个关键字，也可以像“self”一样，代表“当前类”，用于访问一个类的“静态属性或静态方法”

但，static，在应用中更灵活，因此更常见！

因为static，它代表的是“调用“当前方法的，而不是“其代码所在的类”

Self它就比较死板，只代表这个单词本身所在位置的所在类



这里Static的

A::调用show1时，代表A

B::调用show1时，代表B

## Static关键字的总结：

|  |  |
| --- | --- |
| 含义 | 位置示例 |
| 代表函数或方法中的静态变量 | Function f1(){  Static $v1=1  } |
| 代表类中的静态成员 | Class A{  Static $p1=1  Static function f1(){}  } |
| 代表调用当前方法的“当前类” | Class A{  Static $p1=1  Static function f1(){  Echo static ::属性或方法  }  } |

# 面向对象编程思想的三个特征

## 封装

无非是一个大的指导思想，目的是为了将一个类设计得更加健壮！

其基本做法是：  
 尽可能地将一个类的成员私有化，只开放那些必不可少的对外的属性或方法，能private的就不要protected，就protected的就不要public

## 继承

是面向对象的基本思想和基本做法。

继承是代码重用的一种重要机制

## 多态

多态就是多种形态，其实指的是，现实世界的“丰富多彩的表现形式”，比如

人在吃饭

猪在吃食

鱼在进食

小鸡啄米

…………..

他们都是“吃”这个行为！

在实际代码（应用）中。多态常常有两种表现形式

1. 不同对象，使用相同的方法，会表现不同的结果
2. 同一个对象，使用相同的方法，也可能会表现为不同——方法重载现象