昨日回顾 2

有关类和对象的相关知识 3

类的自动加载 3

做法1：使用\_\_autoload魔术函数。 3

做法2：使用spl\_autoload\_register函数 4

对象的复制（克隆）ｃｌｏｎｅ 5

浅克隆： 5

深克隆： 6

对象遍历 6

PHP内置标准类 8

将数据转换为对象 8

类型约束： 9

与类有关的魔术常量： 10

数据（变量）序列化／反序列化 11

其他魔术方法 12

\_\_sleep(), 和 \_\_wakeup() 12

\_\_tostring()方法 13

\_\_invoke()方法 14

一些有关类和对象的系统函数和运算符 14

两个特定语法场景的辨析： 15

对象向下传递特性； 15

static后期静态绑定特性： 17

面向对象3大思想特征介绍： 18

昨日回顾

[设计模式](#Toc10421)

[单例模式](#Toc7720)

目标：是设计一个只能创建出一个实例的类。

class A{

private static $instance ;

private function \_\_construct(){}

private function \_\_clone(){}

static function getInstance(){

if( !isset(self::$instance) ){

self::$instance = new self();

}

return self::$instance

}

}

[实现MysqlDB类的单例模式及完整功能：](#Toc24349)

再写一下该类的设计要点

class MySQLDB{

private $host;

private $port;

private $username;

private $password;

private $charset;

private $dbname;

private $source; //存储连接后的资源

private static $instance ;

private function \_\_construct($config){

//初始化6项数据.....

........................

$this->connect(); //连接数据库

$this->setcharset($this->charset);//设定编码；

$this->selectdb($this->dbname); //选择数据库

}

private function \_\_clone(){}

static function getInstance( $config ){

if( !isset(self::$instance) ){

self::$instance = new self($config);

}

return self::$instance

}

private function connect(){};

function setcharset($charset){设定编码}

function selectdbt($db){选择数据库}

function query($sql){执行所有sql}

function fetchAll($sql){取得多行数据返回二维数组}

function fetchAll($sql){取得一行数据返回一维数组}

function fetchAll($sql){取得1行1列数据返回单个数据}

}

[抽象类，抽象方法](#Toc13061)

[抽象类](#Toc23586)：类的前面加abstract

[抽象方法：](#Toc20654)方法的前面加abstract，并且完全省略方法体；

[抽象类抽象方法细节关系描述](#Toc30486)

一个类可以设计成抽象类，但里面可以没有抽象方法。

一个抽象方法，一定要放在抽象类中。

抽象类必须需要去继承出子类

抽象方法必须需要在子类中去重写（覆盖）

[PHP中的重载技术](#Toc15529)

[通常面向对象语言的重载技术](#Toc12991)

function f1($x);

function f1($x, $y);

[属性重载](#Toc26747)：

用于应对使用一个对象的不存在的属性的情形，只有4个固定的魔术方法：

\_\_set(), \_\_get(), \_\_isset(), \_\_unset();

[方法重载](#Toc31226):

\_\_call() \_\_callstatic()

[接口interface](#Toc32085)

接口的定义：就是一个其中只有常量和抽象方法的“容器”，interface

接口的实现（多实现）：implements，可以多实现。

一个类“继承”接口的特性，被称为“实现”

接口的继承：

接口之间也可以相互继承，extends

有关类和对象的相关知识

类的自动加载

类的自动加载是指，在外面的页面中，并不需要去“引入”（包含）类文件，但是程序会在需要一个类的时候就自动去“动态加载”该类。

什么叫做“需要一个类”？通常是这样的情况：

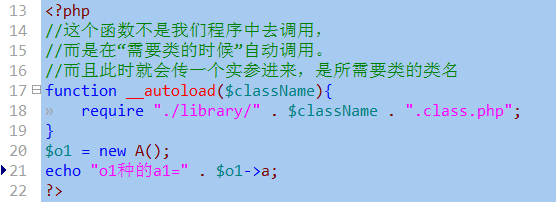
１，创建一个对象的时候（new）

2，直接使用一个类名（操作静态或静态方法）

做法非常简单：

做法1：使用\_\_autoload魔术函数。

即，“当程序需要一个类”的时候，就会去调用该函数：该函数我们需要自己去定义并在其中写好加载类文件的通用语句：



做法2：使用spl\_autoload\_register函数

该函数的作用是：用它“注册”（声明）多个可以用来代替\_\_autoload函数作用的函数，语法如下：

spl\_autoload\_regist(“函数名1”);

spl\_autoload\_regist(“函数名2”);

.........

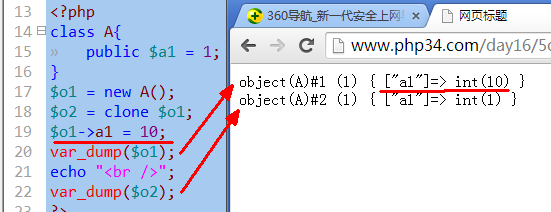
自然，其后也得去定义这些函数，并且这些函数的作用跟\_\_autoload一样，不过此时就可以应对“更多的情形”——比如类文件分布在不同的目录中。



对象的复制（克隆）ｃｌｏｎｅ

浅克隆：

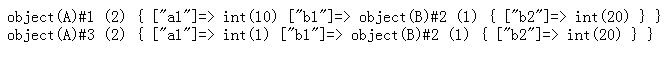
只能克隆对象中的“非对象非资源”数据：



但，如果对象中的属性存储的是“对象”类型，则就可以看到克隆不完全的情形，如下：



结果显示为：



可见，修改了ｏ１中的ｂ１数据，发现ｏ２种的ｂ１数据也跟着改变了——克隆不完全。

我们可以使用“深克隆”来解决：

深克隆：

php中，默认克隆是浅克隆

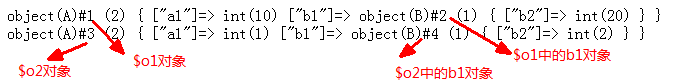
要想实现深克隆（一个对象的所有属性数据都彻底实现了“复制”），就需要对该对象类使用魔术方法：

\_\_clone（），并在里面来实现深度克隆——人为去复制浅克隆复制不了数据。

则刚才的代码，改进如下：



结果为：



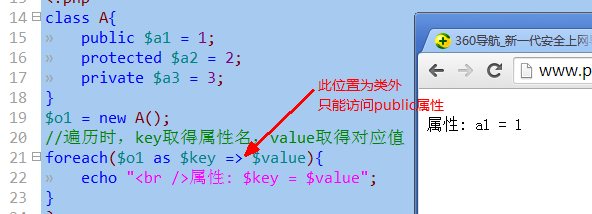
对象遍历

对象也可以可以使用foreach语句进行便利，有两点注意：

1，只能便利属性。

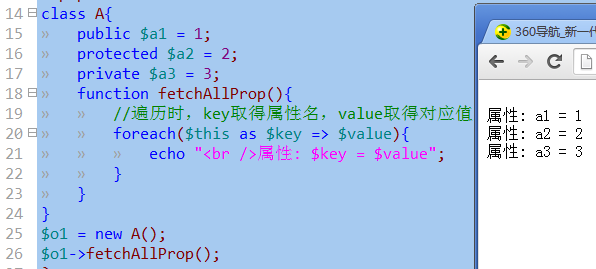
2，只能便利“看得到”的属性——代码所在范围可访问。

在外面遍历：

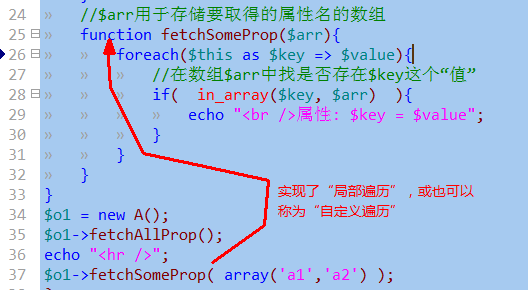


可见，此时protected和private属性都没有遍历出来。

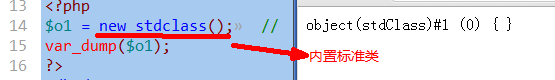
如果需要全部遍历，则改造为：



一个思考题：如果只需要某对象中的部分数据？比如：只要a1,a2这个两个属性的数据？

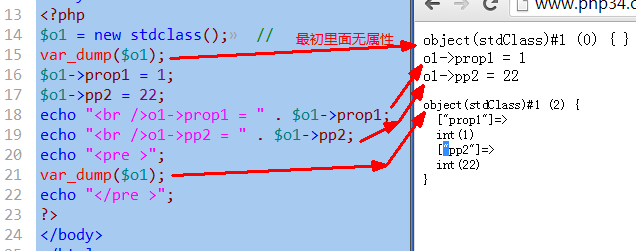


PHP内置标准类



可见该类内部没有定义任何属性。

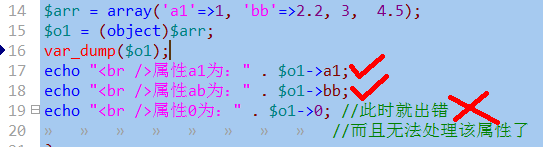
但我们还是可以使用该类的属性——不存在的属性。



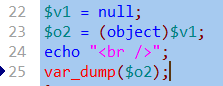
将数据转换为对象

转换基本语法：（目标类型）数据；

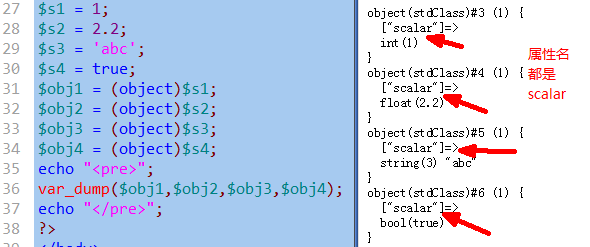
* 对象转换为对象：没有变化；
* 数组转换为对象：数组的键名当作属性名，值为对应值；
  + 但通常，如果数数字下标，则其对应属性 并不方便操作（->）
  + 所以通常就适合于纯字符下标的数组。



* null转换为对象：空对象；

，结果为：

* 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



类型约束：

什么事类约束？

其他很多语言中，类型约束是语法上的要求，即定义一个变量的时候，必须指定类型，并以后也只能存储该类型数据——这种是强类型语言的一个特点。比如：

int n1 = 1; //int类型变量

float f1 ; //float类型变量

bool function getConcat( string s1, string s2){。。。。} //两个形参是string，返回值也是bool型

php是弱类型语言，其特点是无需为变量指定类型，而且在其后也可以存储任何类型。

但：

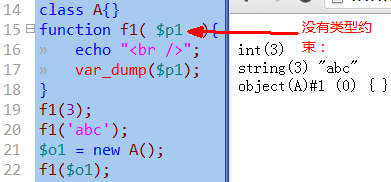
在php的较新的语法中，在某些特定场合，针对某些特定类型，也可以进行语法约束。

特定场合：函数（或方法）的形参变量

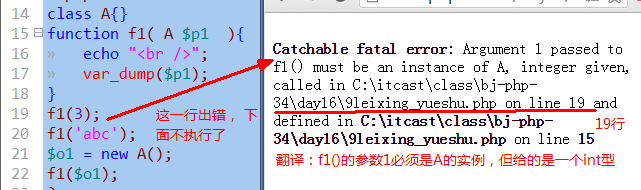
特定类型：对象类型（类名）， 接口类型（接口名）， 数组类型（array）， 函数类型（callable）

可以对函数（或方法）的参数设定必须使用的类型。只能对对象，接口，数组和函数进行约束，如下所示：

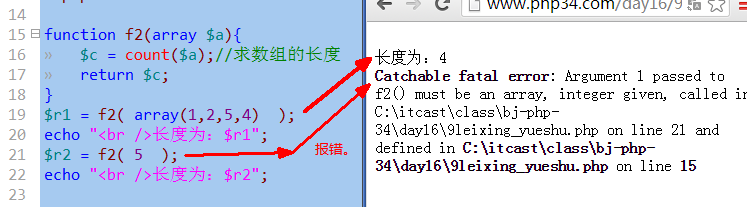
* function f1(类名 $p1){....}：要求该参数只能使用该类的对象；
* function f1(接口名 $p1){....}：要求该参数只能使用实现该接口的对象；
* function f1(array $p1){....}：要求该参数只能使用数组；
* function f1(callable $p1){....}：要求该参数只能是一个函数或方法，此时称之为回调函数（方法）



同样的代码，对f1加上类型约束



再来举个数组例子：



与类有关的魔术常量：

\_\_CLASS\_\_,：获取其所在的类的类名。

\_\_METHOD\_\_：获取其所在的方法的方法名。

class A{

function f1(){

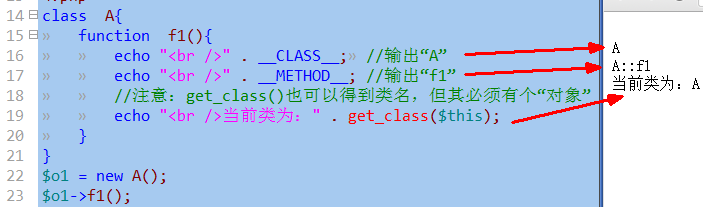
echo \_\_CLASS\_\_； //输出“A”

echo \_\_METHOD\_\_； //输出“f1”

//注意：get\_class()也可以得到类名，但其必须有个“对象”

}

}



数据（变量）序列化／反序列化

什么叫序列化：

就是将一个变量的数据“转换为”字符串——但并不是类型转换。

其目的（意图）是将该字符串“存储起来”（比如存为文本文件）

然后在其他时间（或其他运行环境/电脑）上，将其又恢复原样。

比如：很多有些设计中，都有一个功能：保留场景及当前所有装备，武器，服装。。。。

则也就是对应了有“反序列化”工作

序列化的做法非常简单：

1，$str = serialize($变量); //此时该变量的数据就变成了字符串，并存入变量$str1;

但通常的应用出，此时就应该将其保存为文件（或数据库），如下：

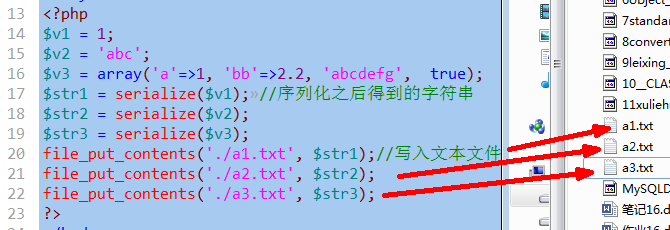
2，file\_put\_contents(“文本文件路径”, $str);

反序列化操作也同样简单：

3，$str = file\_get\_contents(“文本文件路径”); //先从文本文件中取得内容

4，$v1 = unserialize( $str); //反序列化，此时$v1这个变量中存储的就是原来变量内容（和类型）

举例：



然后在另一个文件中进行反序列化操作：



其他魔术方法

\_\_sleep(), 和 \_\_wakeup()

当对一个对象进行“序列化”操作的时候，会自动调用类中的\_\_sleep()方法；

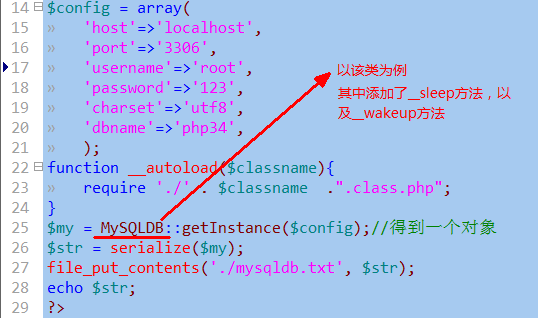
当“反序列化”一个对象的时候，会自动调用对应类中的\_\_wakeup()方法；

注意：反序列化操作中，必须已经具备了对应的类的声明。

\_\_sleep()方法中可以进行一些数据（资源）的清理工作，并返回一个数组，该数组可以存储一些想要进行序列化的对象的属性——即可以挑选属性进行序列化。

\_\_wakeup()方法中，可以在进行反序列化操作的时候进行某些有用的“数据恢复工作”，比如：连接数据库

这是“序列化”对象：



这是反序列化对象：



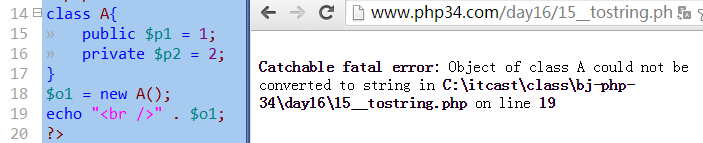
\_\_tostring()方法

当把一个对象当作一个“字符串”来看待（处理）的时候，会自动调用该魔术方法。

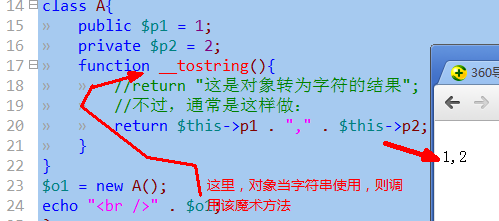
通过该方法，可以返回“合适”的字符串，也可以认为就是对象转换为字符串的结果。

如果没有改方法，会出错。

对象不能直接当字符串使用：



添加魔术方法：

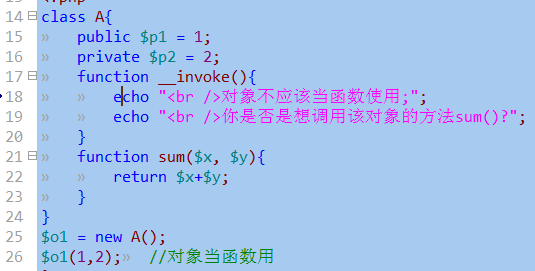


\_\_invoke()方法

对象当作一个“方法”（函数）的时候，会自动调用该魔术方法。

比如：

$v1 = $obj(1,2); //$obj其实是一个对象，这里就相当于将对象当函数用。





一些有关类和对象的系统函数和运算符

class\_exists(), ：判断某个类是否存在（定义过）

interface\_exists(), ：判断接口是否存在

get\_class()：获取某个对象的“所属类名”

get\_parent\_class(), ：获取某个对象的“所属父类的类名”

get\_class\_methods(), ：获取一个类的所有方法，返回一个索引数组，就是这些方法的名字。

get\_class\_vars(), ：获取一个类的所有属性，返回一个数组，下标为属性名，值为属性值。

get\_declared\_classes()：获得所有声明过的类（含系统中的类）

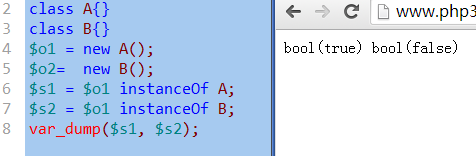
is\_object()：判断是否对象

get\_object\_vars()：获得对象的所有属性，返回一个数组，下标为属性名，值为属性值

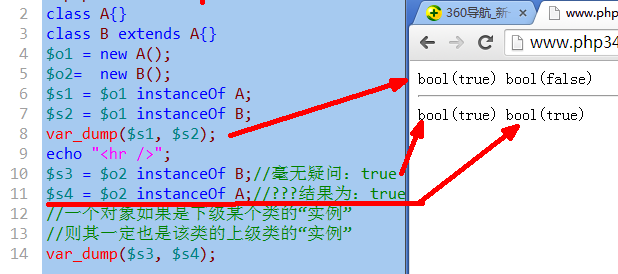
运算符：

new：

instanceof： 判断一个对象是否是某个类的“实例”



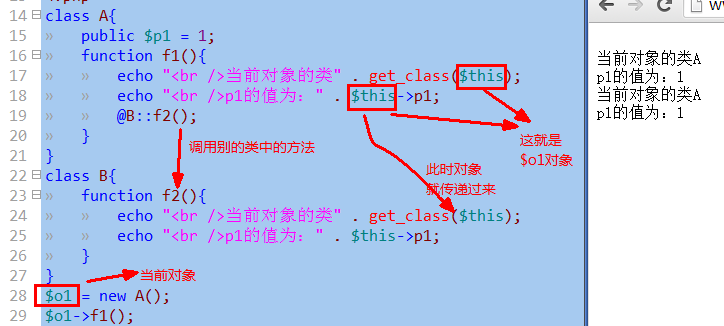
注意一个现象：



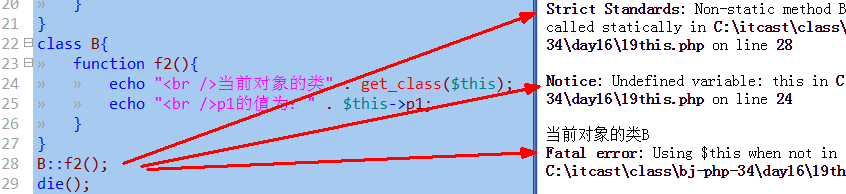
两个特定语法场景的辨析：

对象向下传递特性；

当一个对象（$o1) 调用一个实例方法，然后，在该实例方法（f1）中又去“静态”调用另一个类的方法（f2），则在方法f2中，会自动获得在方法f1中的那个$this对象。

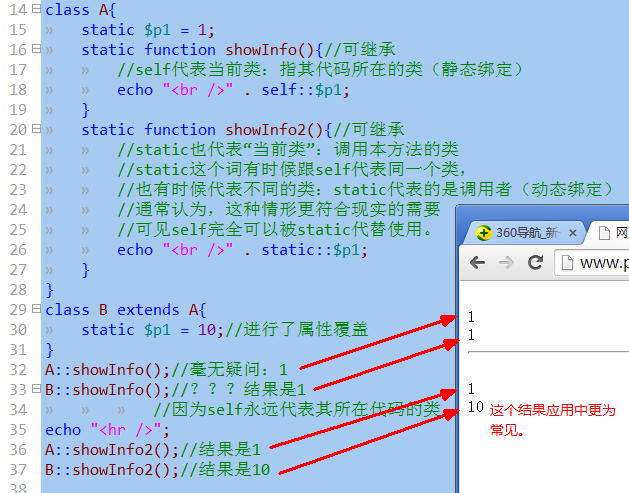


对比：



可见f2方法直接调用会产生严重错误（因为找不到$this)

static后期静态绑定特性：



可见static有3个不同含义的语法：

函数中的静态变量：

function f1(){

static $v1 = 1;

}

类中的静态成员：

class A{

static $v1 = 1;

static function f1(){}

}

方法中的动态指代“当前类”：

class A{

function f1(){

static::f2();//static此时指代调用f1这个方法的类（或对象的类）

self::f2(); //self这里永远指代当前A这个类。

}

}

面向对象3大思想特征介绍：

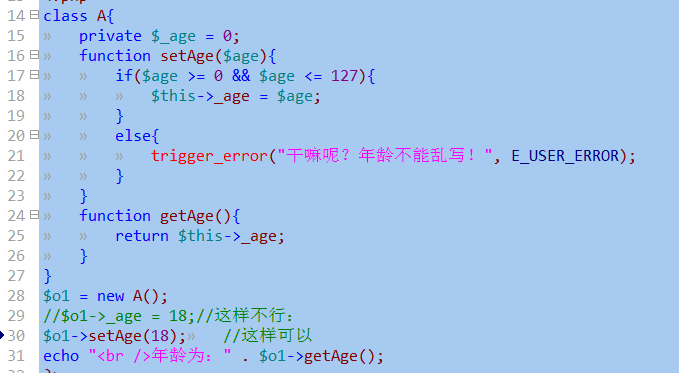
封装：

就是把数据“封闭”起来，尽量不给别人看到。

面向对象的基本语法“类的定义”形式，可以认为是最基本封装——封装了很多数据到某个类（对象）。

但：

更严格（更常说）的封装，是尽量将属性做出“私有的”——并通过共有的方法来向外提供对该属性的“可控性操作”：



继承：

多态：

多态：看起来用的是相同的东西（方法），但会得到不同的结果。

通常是指：一个对象可以使用相同的方法（动作）但得到不同的结果：

(这里是其他常见面向对象语法示例：)

class A{

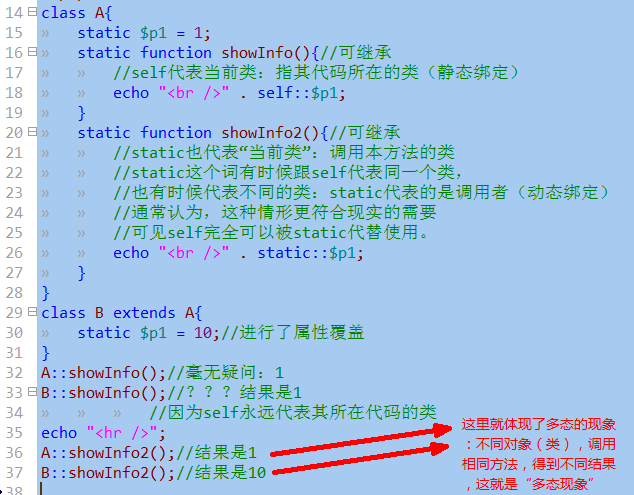
function f1($x ){....做事情1；。。。。}

function f1($x, $y ){....做事情2；。。。。}

function f1($x, $y, $z ){....做事情3；。。。。}

}

也指：不同的对象使用相同的方法，但得到不同的结果：



有空去网上搜搜这3个词的更多的乱七八糟解释。。。。。