

第三章 PHP5 面向对象高级类特性

这一周来正好白天没课，赶出来这章。

宝宝出生将近一个月，快 10 斤重了。

宝宝快张大，你想编程么？宝宝妈说，做这个行业太累了。

刀客羽朋
于石家庄 2006-11-23

目录

3.1 static变量、方法	3
3.1.1 静态属性公用特性	4
3.1.2 静态属性直接调用	5
3.1.3 静态方法	6
3.1.4 静态方法如何调用静态方法	7
3.1.5 静态方法调用静态属性	8
3.1.6 静态方法不能调用非静态属性	9
3.1.6 静态方法调用非静态方法	10
3.1.7 设计模式之单件模式	12
3.2 final类、final方法和常量	16
3.2.1 final类的不能被继承	16
3.2.2 final方法不能被重写	17
3.2.3 PHP5 中的常量	18
3.3 abstract类和abstract方法	20
3.3.1 abstract 抽象类	21
3.3.2 abstract 抽象方法	23
3.3.3 抽象类继承抽象类	27
3.3.4 静态抽象方法	29
3.3.5 PHP5.2.0 中的静态抽象方法	30
3.4 设计模式之模版模式	31
3.4.1 模版模式实例	31

3.1 **static** 变量、方法

- **static** 关键字用来修饰属性、方法，称这些属性、方法为静态属性、静态方法。
- **static** 关键字声明一个属性或方法是和类相关的，而不是和类的某个特定的实例相关，因此，这类属性或方法也称为“**类属性**”或“**类方法**”
- 如果访问控制权限允许，可不必创建该类对象而直接使用类名加两个冒号“**::**”调用。

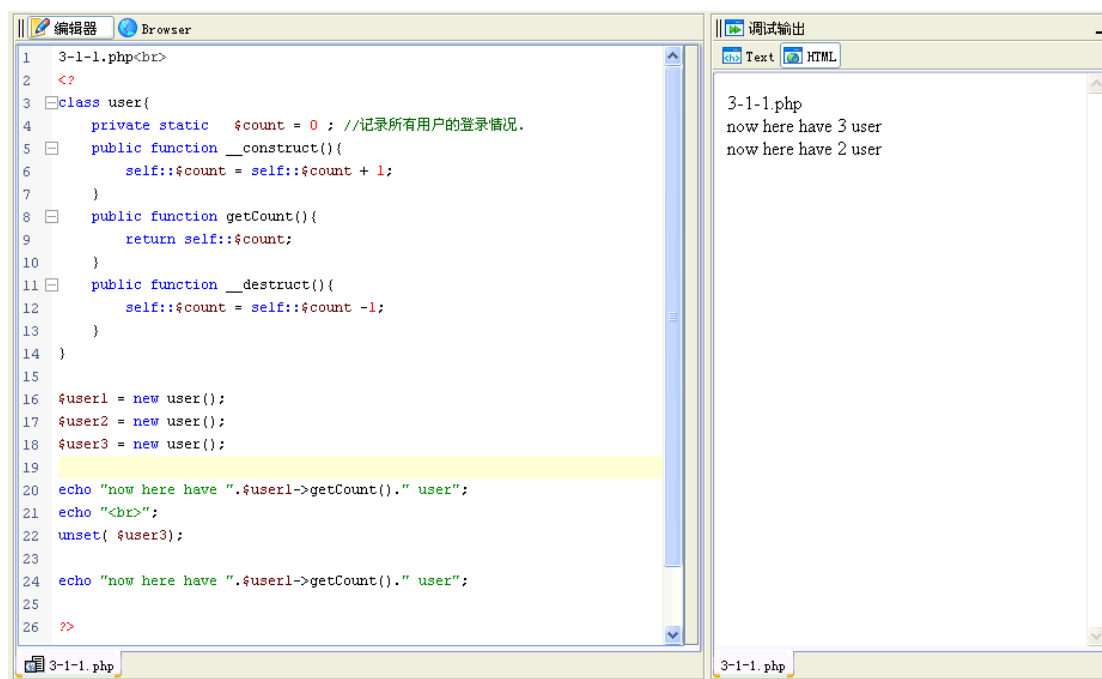
- **static** 关键字可以用来修饰变量、方法。
- 不经过实例化，就可以直接访问类中 **static** 的属性和 **static** 的方法。
- **static** 的属性和方法，只能访问 **static** 的属性和方法，不能类访问非静态的属性和方法。因为静态属性和方法被创建时，可能还没有任何这个类的实例可以被调用。
- **static** 的属性，在内存中只有一份，为所有的实例共用。
- 使用 **self::** 关键字访问当前类的静态成员。

3.1.1 静态属性公用特性

- 一个类的所有实例，共用类中的静态属性。
- 也就是说，在内存中即使有多个实例，静态的属性也只有一份。

下面例子中的设置了一个计数器\$count 属性，设置 private 和 static 修饰。这样，外界并不能直接访问\$count 属性。而程序运行的结果我们也看到多个实例在使用同一个静态的\$count 属性。

例 3-1-1.php



```
1 3-1-1.php<br>
2 <?
3 class user{
4     private static $count = 0 ; //记录所有用户的登录情况.
5     public function __construct(){
6         self::$count = self::$count + 1;
7     }
8     public function getCount(){
9         return self::$count;
10    }
11    public function __destruct(){
12        self::$count = self::$count -1;
13    }
14 }
15
16 $user1 = new user();
17 $user2 = new user();
18 $user3 = new user();
19
20 echo "now here have " . $user1->getCount() . " user";
21 echo "<br>";
22 unset( $user3);
23
24 echo "now here have " . $user1->getCount() . " user";
25
26 ?>
```

调试输出

```
3-1-1.php
now here have 3 user
now here have 2 user
```

3.1.2 静态属性直接调用

- 静态属性不需要实例化就可以直接使用，在类还没有创建时就可以直接使用。
- 使用的方式是 **类名::静态属性名**

例 3-1-2.php

```

1 3-1-2.php<br>
2 <?
3 class Math{
4     public static $pi = 3.14;
5
6 }
7 //求一个半径3的圆的面积。
8 $r = 3;
9 echo "半径是 $r 的面积是<br>";
10 echo Math::$pi * $r * $r ;
11
12 echo "<br><br>";
13
14 //这里我觉得 3.14 不够精确，我把它设置的更精确。
15 Math::$pi = 3.141592653589793;
16 echo "半径是 $r 的面积是<br>";
17 echo Math::$pi * $r * $r ;
18
19
20 ?>
  
```

调试输出

```

3-1-2.php
半径是 3 的面积是
28.26

半径是 3 的面积是
28.274333882308
  
```

类没有创建，静态属性就可以直接使用。

那静态属性在什么时候在内存中被创建？

在 PHP 中没有看到相关的资料。引用 Java 中的概念，来解释应该也具有通用性。

静态属性和方法，在类被调用时创建。

类没有创建，静态属性就可以直接使用。那静态属性在什么时候在内存中被创建？在 PHP 中没有看到相关的资料。我们引用 Java 中的概念，来解释 PHP 的静态修饰符，应该也具有通用性。
静态属性和方法，在类被调用时创建。
类被调用，是指类被创建或者类中的任何静态成员被调用。

3.1.3 静态方法

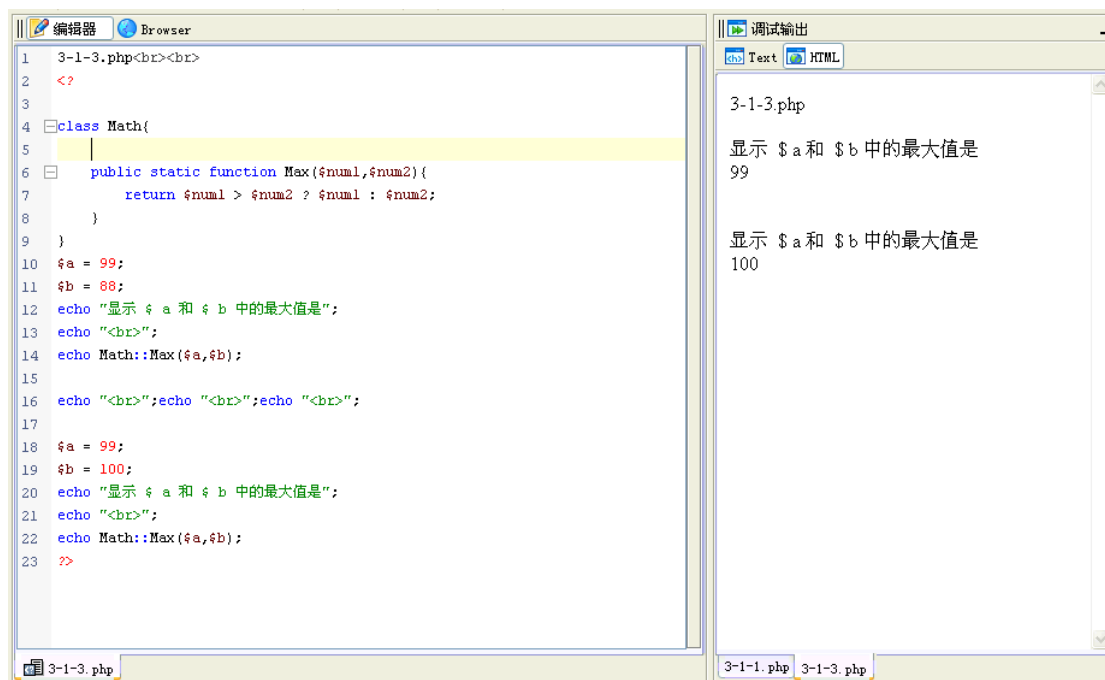
- 静态方法不需要所在类被实例化就可以直接使用。
- 使用的方式是 **类名::静态方法名**

下面我们继续写这个 `Math` 类，用来进行数学计算。我们设计一个方法用来算出其中的最大值。

既然是数学运算，我们也没有必要去实例化这个类，如果这个方法可以拿过来就用就方便多了。

我们这只是为了演示 `static` 方法而设计的这个类。在 PHP 提供了 `max()` 函数比较数值。

例 3-2-3.php



```
1 3-1-3.php<br><br>
2 <?
3
4 class Math{
5
6     public static function Max($num1,$num2){
7         return $num1 > $num2 ? $num1 : $num2;
8     }
9 }
10 $a = 99;
11 $b = 88;
12 echo "显示 $ a 和 $ b 中的最大值是";
13 echo "<br>";
14 echo Math::Max($a,$b);
15
16 echo "<br>";echo "<br>";echo "<br>";
17
18 $a = 99;
19 $b = 100;
20 echo "显示 $ a 和 $ b 中的最大值是";
21 echo "<br>";
22 echo Math::Max($a,$b);
23 ?>
```

调试输出

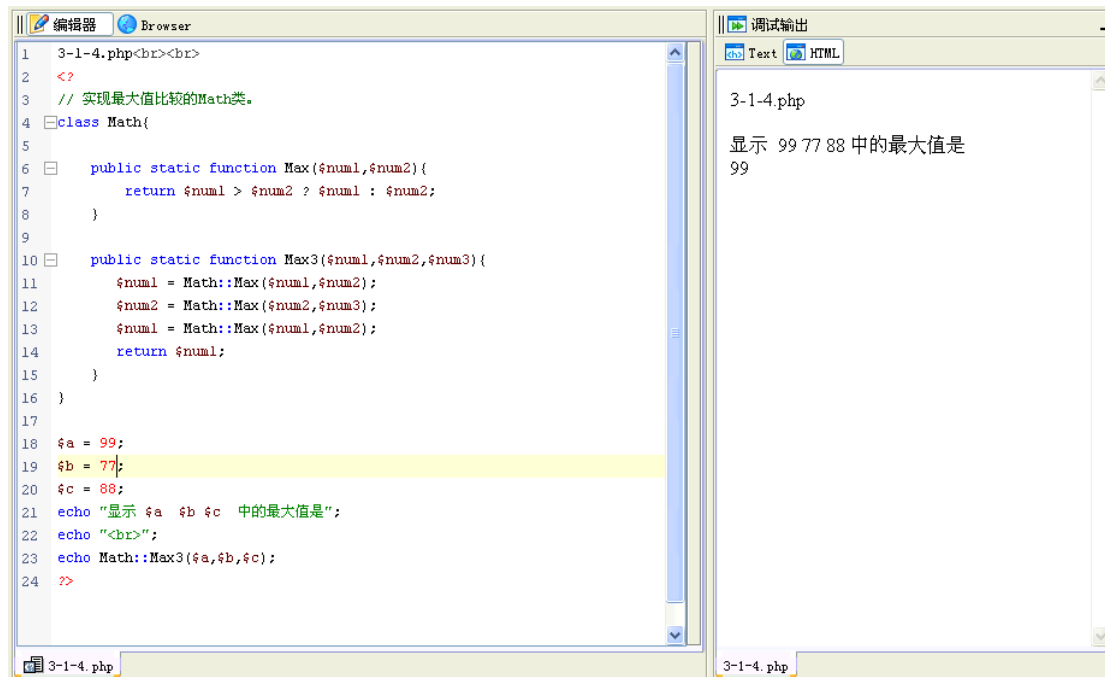
```
3-1-3.php
显示 $ a 和 $ b 中的最大值是
99

显示 $ a 和 $ b 中的最大值是
100
```

3.1.4 静态方法如何调用静态方法

第一个例子，一个静态方法调用其它静态方法时，直接使用 类名

例 3-1-4.php



```
1 3-1-4.php<br><br>
2 <?
3 // 实现最大值比较的Math类。
4 class Math{
5
6     public static function Max($num1,$num2){
7         return $num1 > $num2 ? $num1 : $num2;
8     }
9
10    public static function Max3($num1,$num2,$num3){
11        $num1 = Math::Max($num1,$num2);
12        $num2 = Math::Max($num2,$num3);
13        $num1 = Math::Max($num1,$num2);
14        return $num1;
15    }
16 }
17
18 $a = 99;
19 $b = 77;
20 $c = 88;
21 echo "显示 $a $b $c 中的最大值是";
22 echo "<br>";
23 echo Math::Max3($a,$b,$c);
24 ?>
```

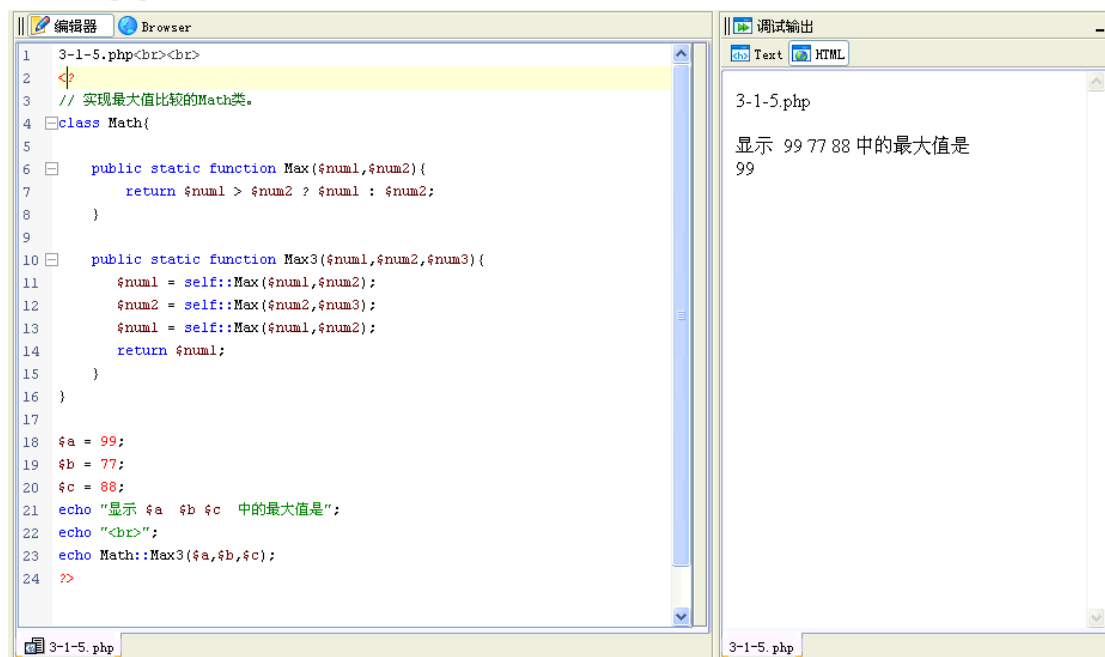
调试输出

3-1-4.php

显示 99 77 88 中的最大值是
99

也可以使用 **self::** 调用当前类中的其它静态方法。(建议)

例 3-1-5.php



```
1 3-1-5.php<br><br>
2 <?
3 // 实现最大值比较的Math类。
4 class Math{
5
6     public static function Max($num1,$num2){
7         return $num1 > $num2 ? $num1 : $num2;
8     }
9
10    public static function Max3($num1,$num2,$num3){
11        $num1 = self::Max($num1,$num2);
12        $num2 = self::Max($num2,$num3);
13        $num1 = self::Max($num1,$num2);
14        return $num1;
15    }
16 }
17
18 $a = 99;
19 $b = 77;
20 $c = 88;
21 echo "显示 $a $b $c 中的最大值是";
22 echo "<br>";
23 echo Math::Max3($a,$b,$c);
24 ?>
```

调试输出

3-1-5.php

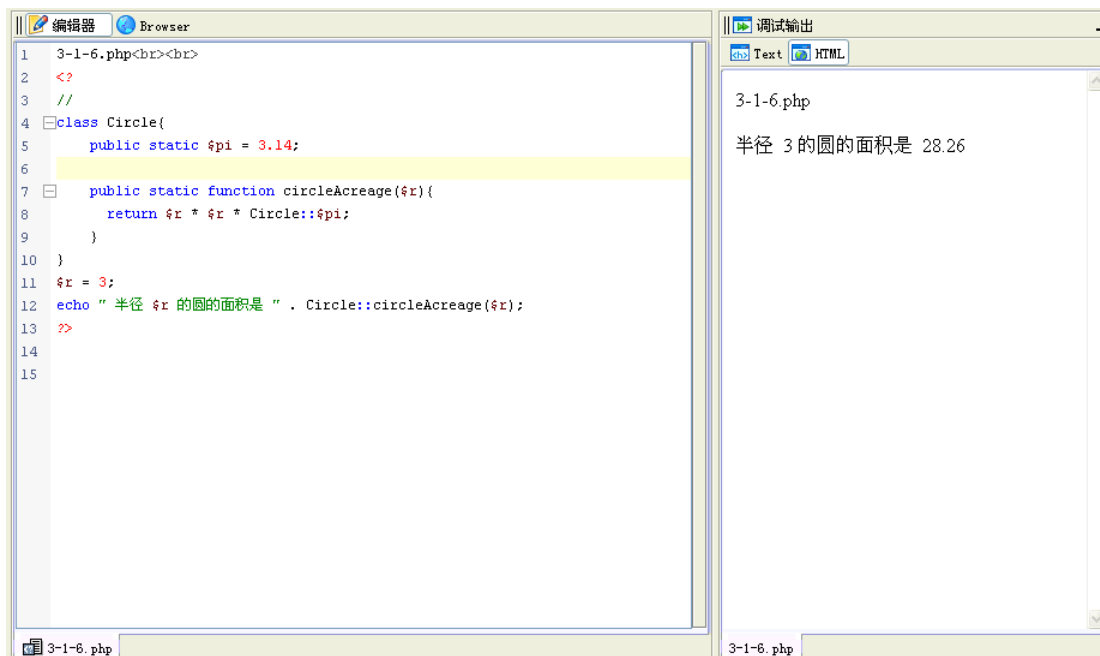
显示 99 77 88 中的最大值是
99

记住这个 **self::** 它表示当前类的静态成员，与 **\$this** 不同，**\$this** 指当前对象。

3.1.5 静态方法调用静态属性

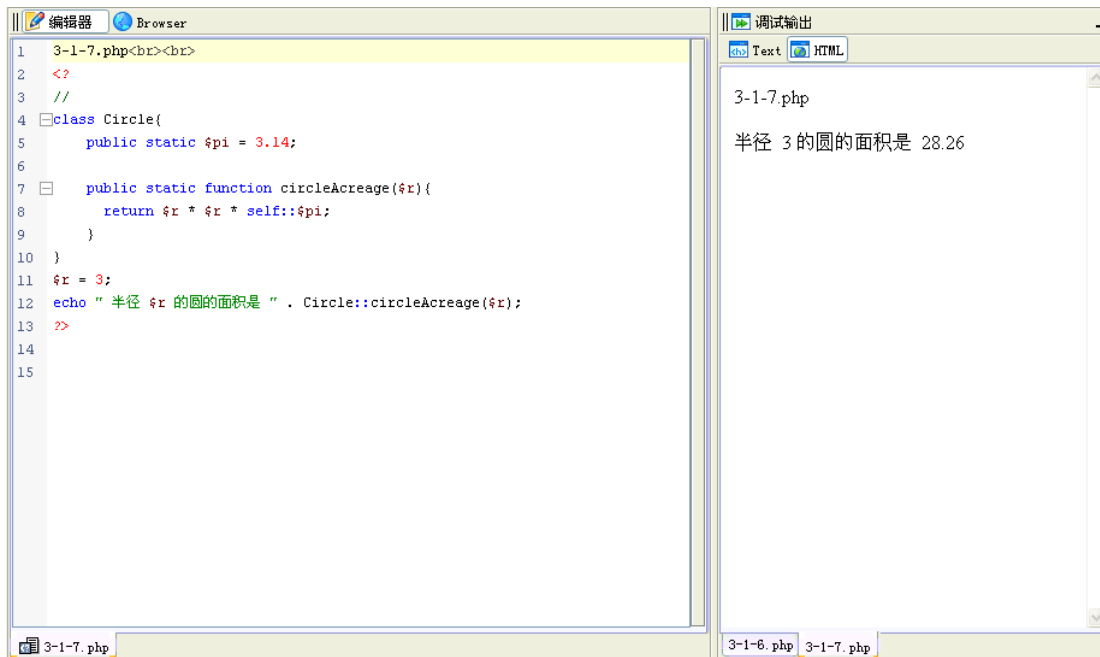
使用 **类名::静态属性名** 调用本类中的静态属性。

例 3-1-6.php



使用 **self::** 调用本类的静态属性。（建议）

例 3-1-7.php

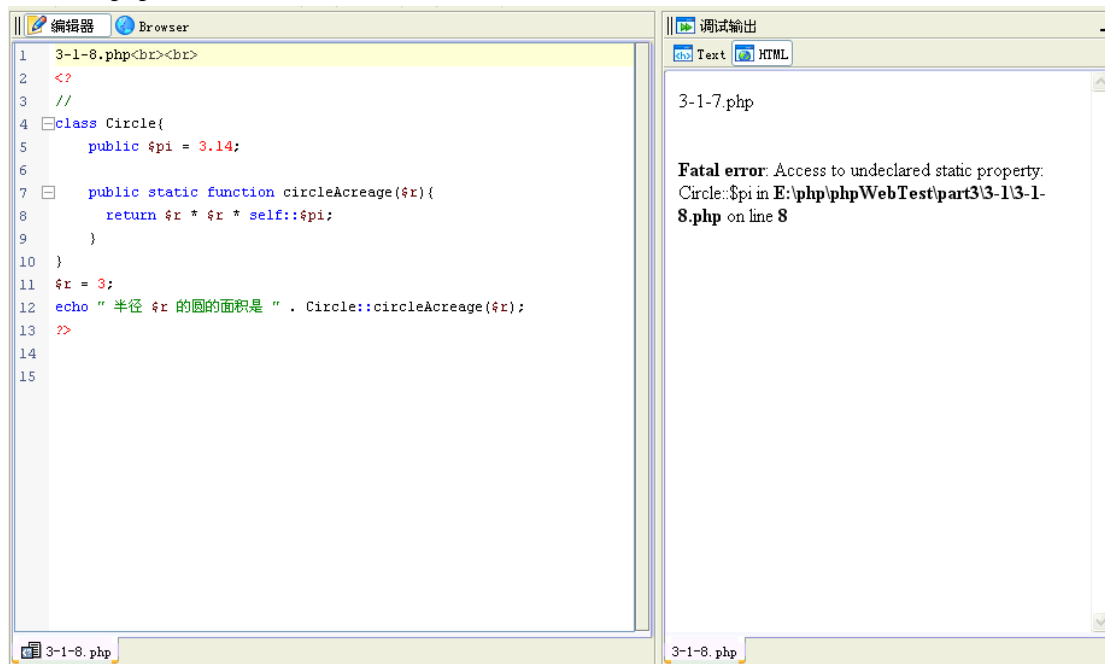


3.1.6 静态方法不能调用非静态属性

- 静态方法不能调用非静态的属性。

不能使用 `self::` 调用非静态属性。

例 3-1-8.php



The screenshot shows a PHP IDE with two panes. The left pane is a code editor for '3-1-8.php' containing the following code:

```
1 3-1-8.php<br><br>
2 <?
3 //
4 class Circle{
5     public $pi = 3.14;
6
7     public static function circleAcreage($r){
8         return $r * $r * self::$pi;
9     }
10 }
11 $r = 3;
12 echo " 半径 $r 的圆的面积是 " . Circle::circleAcreage($r);
13 >>
14
15
```

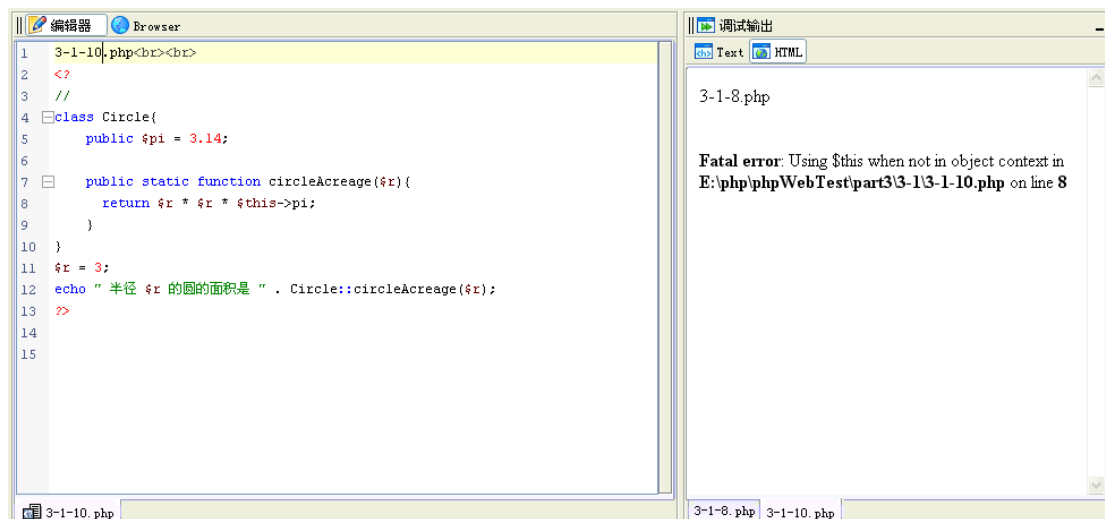
The right pane is the '调试输出' (Debug Output) window, showing the output of the script:

```
3-1-7.php

Fatal error: Access to undeclared static property:
Circle::$pi in E:\php\phpWebTest\part3\3-1\3-1-
8.php on line 8
```

也不能使用 `$this` 获取非静态属性的值。

例 3-1-10



The screenshot shows a PHP IDE with two panes. The left pane is a code editor for '3-1-10.php' containing the following code:

```
1 3-1-10.php<br><br>
2 <?
3 //
4 class Circle{
5     public $pi = 3.14;
6
7     public static function circleAcreage($r){
8         return $r * $r * $this->pi;
9     }
10 }
11 $r = 3;
12 echo " 半径 $r 的圆的面积是 " . Circle::circleAcreage($r);
13 >>
14
15
```

The right pane is the '调试输出' (Debug Output) window, showing the output of the script:

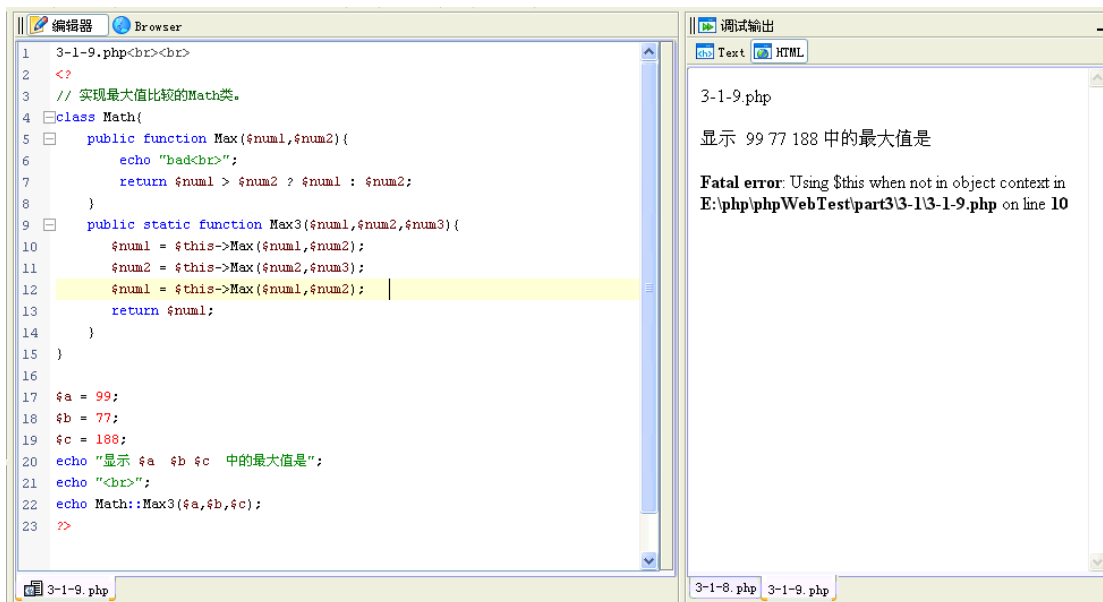
```
3-1-8.php

Fatal error: Using $this when not in object context in
E:\php\phpWebTest\part3\3-1\3-1-10.php on line 8
```

3.1.6 静态方法调用非静态方法

PHP5 中，在静态方法中不能使用 `$this` 标识调用非静态方法。

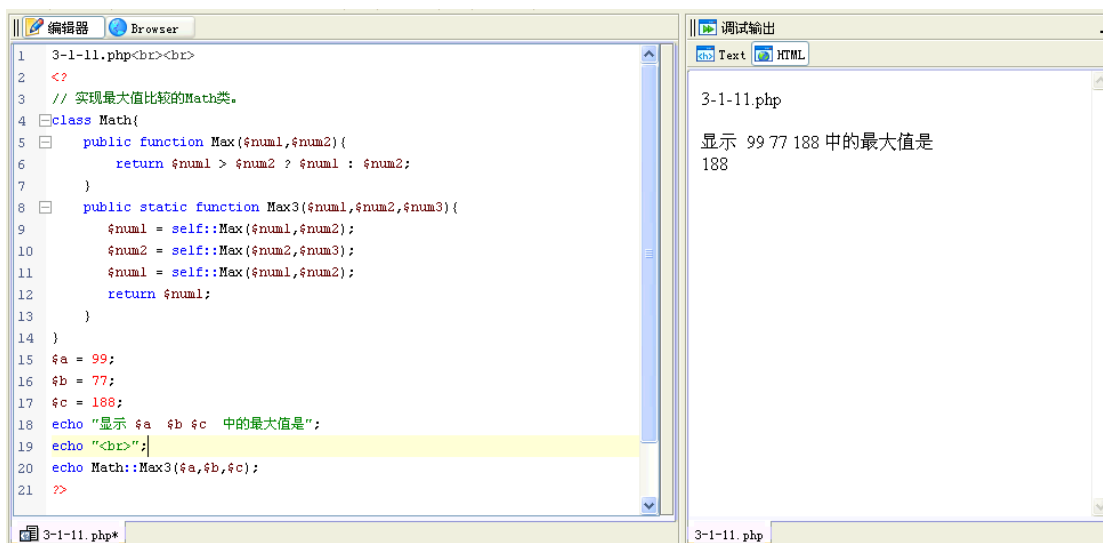
例 3-1-9



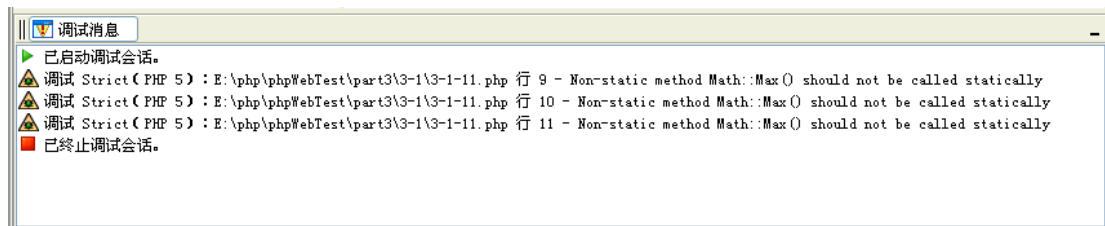
- 当一个类中有非静态方法被 `self::` 调用时，系统会自动将这个方法转换为静态方法。

下面的代码被执行了，而且有结果。因为 `Max` 方法被系统转换为静态方法了。

例：3-1-11.php



在 Zend Studio 的调试部分显示出了这些错误的信息。

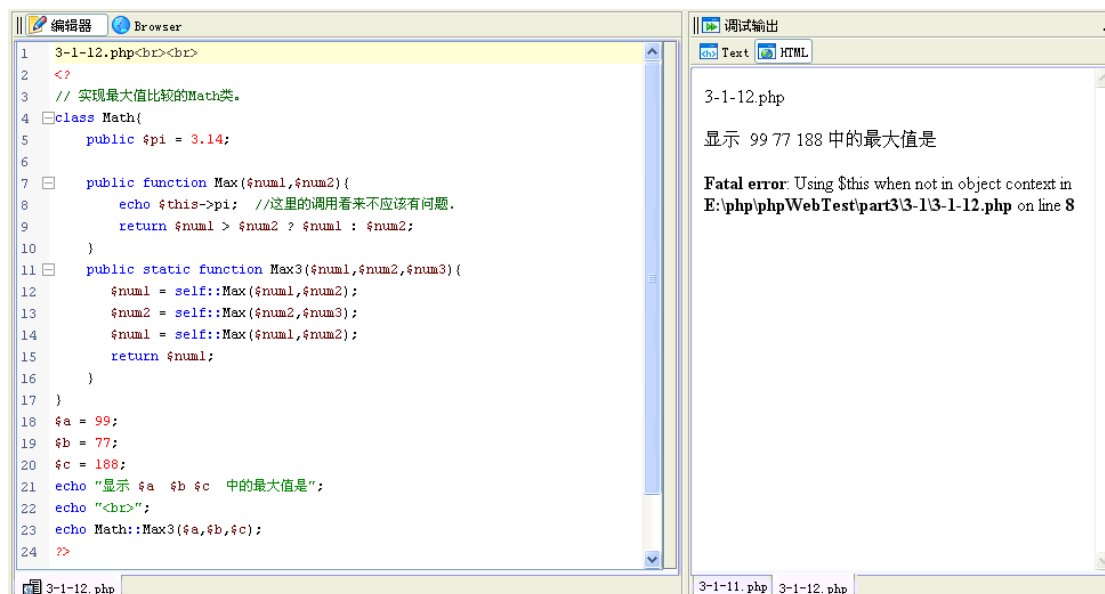


下面的例子中，我们让静态方法 Max3 用过 self::调用了非静态方法 Max，有让非静态方法 Max 通过\$this 调用非静态属性 \$pi 。

在运行是报出了错误，这个错误和前一个例子 3-1-9.php 一样，这次倒是非静态方法 Max 报出了静态方法调用非静态属性的错误。

这倒是证明了一点，在这里我们定义的非静态方法 Max 被系统自动转换成静态方法了。

例：3-1-12.php



3.1.7 设计模式之单件模式

单件模式要解决的问题就是“如何让这个类只有一个实例”。

我们的 web 应用中，大量使用了数据库连接，如果反复建立与数据库的连接必然消耗更多的系统资源。

我们如何解决这个问题，建立唯一的数据库连接是必要的方式。

我们又如何知道与这个数据库的连接是否已经建立？还是需要现在建立？

单件模式可以解决这个问题。

先假设我们需要一个类完成在内存中只有一份的功能，我们该如何做呢？

我们一步一步的使用前面学过的知识来写一个单件的例子。

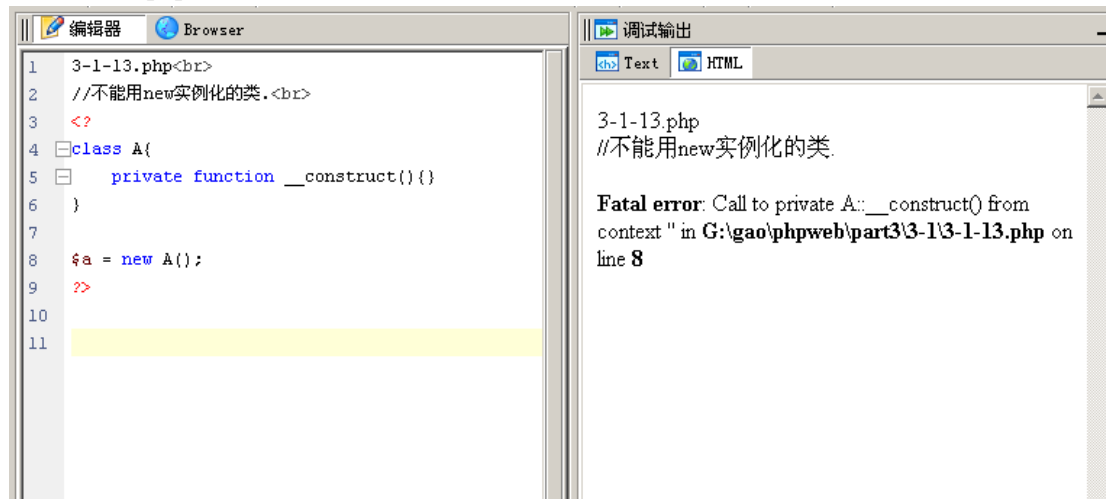
问题 1:

- 前面学过，每次用 `new 类名` 的方式，就可以创建一个对象。
- 我们必须禁止外部程序用 `new` 类名的方式来创建多个实例。

解决办法:

- 我们将构造函数设置成 `private`，让构造函数只能在内部被调用，而外部不能调用。
- 这样，这个类就不能被外部用 `new` 的方式建立多个实例了。

例：3-1-13.php 不能被外部用 `new` 实例化的类。



```
1 3-1-13.php<br>
2 //不能用new实例化的类.<br>
3 <?
4 class A{
5     private function __construct(){
6     }
7
8     $a = new A();
9     ?>
10
11
```

```
3-1-13.php
//不能用new实例化的类.

Fatal error: Call to private A::__construct() from
context " in G:\gao\phpweb\part3\3-1-13.php on
line 8
```

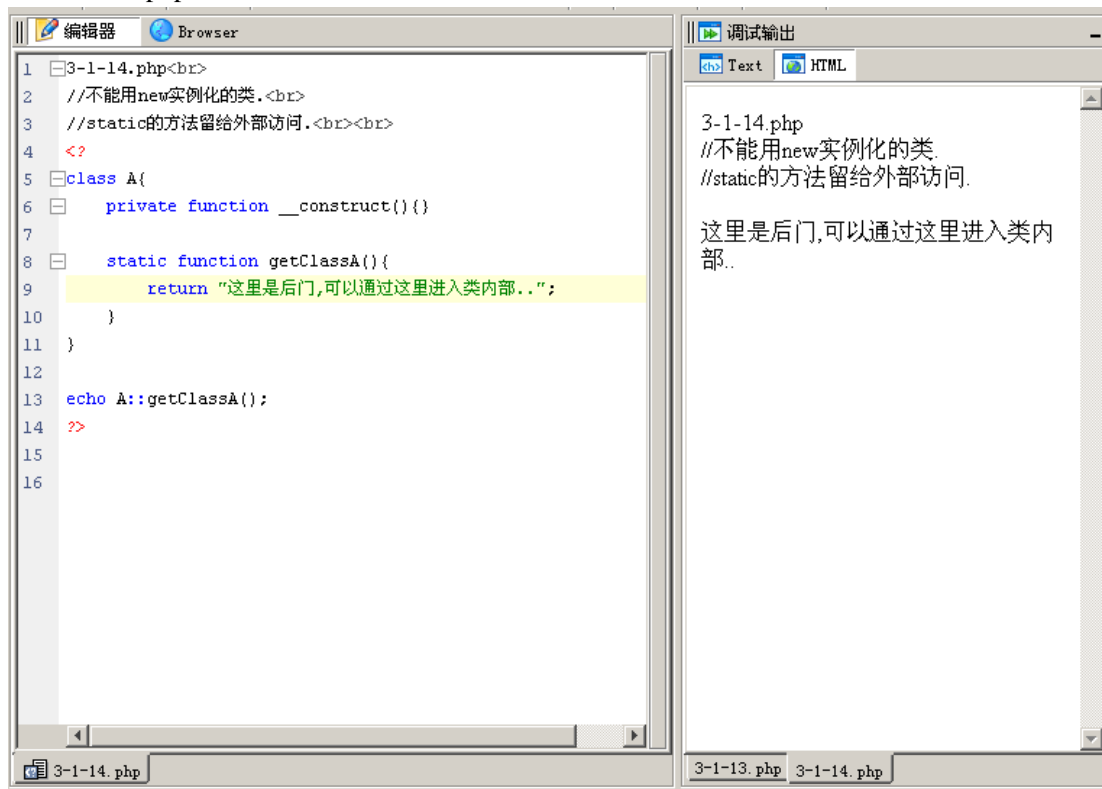
问题 2:

- 我们已经禁止外部用 `new` 实例化这个类，我们改如何让用户访问这个类呢？
- 前门堵了，我们需要给用户留个后门。

解决办法:

- `static` 修饰的方法，可以不经实例化一个类就可以直接访问这个方法。
- 后门就在这里。

例：3-1-14.php



```
1 3-1-14.php<br>
2 //不能用new实例化的类.<br>
3 //static的方法留给外部访问.<br><br>
4 <?
5 class A{
6     private function __construct(){
7
8     static function getClassA(){
9         return "这里是后门,可以通过这里进入类内部..";
10    }
11 }
12
13 echo A::getClassA();
14 ?>
15
16
```

3-1-14.php
//不能用new实例化的类。
//static的方法留给外部访问。

这里是后门,可以通过这里进入类内部..

问题 3:

- 虽然我们已进入类内部，但我们要的是这个类的唯一实例？
- 先不管别的，我们先需要一个实例。
- 通过这个 `static` 的方法返回这个实例，如何做呢？

解决办法:

- `private` 的构造函数，不能被外部实例化。
- 但是我们已经成功潜入类的内部了（间谍？007？），我们在内部当然可以调用 `private` 的方法创建对象。
- 我们这样做看看。

例 3-1-15.php

下面的例子我们确实返回了 A 类的实例，但注意两次执行返回的不是同一个实例。

```

1  3-1-15.php<br>
2  //不能用new实例化的类.<br>
3  //static的方法留给外部访问.<br>
4  //在方法内部返回实例.<br><br>
5  <?>
6  class A{
7  private function __construct(){}
8  static function getClassA(){
9      $a = new A();
10     return $a;
11 }
12 }
13 // 看到这里确实返回的是 A 的实例.但不是同一个对象.
14 $a1 = A::getClassA();
15 $a2 = A::getClassA();
16 echo "\$a1 的类是 ".get_class($a1)." , \$a2 是 ".get_class($a2);
17 if($a1 === $a2){
18     echo "<br> \$a1 \$a2 指向同一对象.";
19 }else{
20     echo "<br> \$a1 \$a2 不是一个对象.";
21 }
22 <?>
23
24

```

```

3-1-15.php
//不能用new实例化的类.
//static的方法留给外部访问.
//在方法内部返回实例.

$a1 的类是 A, $a2 是 A
$a1 $a2 不是一个对象.

```

问题 4:

- 我们已经通过 `static` 方法返回了 A 的实例。但还有问题。
- 我们如何保证我们多次操作获得的是同一个实例的呢？

解决办法:

- `static` 的属性在内部也只有一个。
- `static` 属性能有效的被静态方法调用。
- 将这个属性也设置成 `private`，以防止外部调用。
- 先将这个属性设置成 `null`。
- 每次返回对象前，先判断这个属性是否为 `null`。
- 如果为 `null` 就创建这个类的新实例，并赋值给这个 `static` 属性。
- 如果不为空，就返回这个指向实例的 `static` 属性。

例：3-1-16.php

```

1 3-1-16.php<br>
2 //不能用new实例化的类.<br>//static的方法留给外部访问.<br>
3 //在方法内部返回实例.<br>
4 //定义静态属性保证这个实例能被静态方法调用.<br>
5 //增加判断部分.<br><br>
6 <?
7 class A{
8     private static $a = null;
9     private function __construct(){}
10    static function getClassA(){
11        if( null == self::$a){
12            self::$a = new A();
13        }
14        return self::$a;
15    }
16 }
17 // 看到这里确实返回的是 A 的实例.但不是同一个对象.
18 $a1 = A::getClassA();
19 $a2 = A::getClassA();
20 echo "\$a1 的类是 ".get_class($a1)." , \$a2 是 ".get_class($a1);
21 if($a1 === $a2){
22     echo "<br> \$a1 \$a2 指向同一对象.";
23 }else{
24     echo "<br> \$a1 \$a2 不是一个对象.";
25 }
26 ?>

```

调试输出

```

3-1-16.php
//不能用new实例化的类.
//static的方法留给外部访问.
//在方法内部返回实例.
//定义静态属性保证这个实例能被静态方法调用.
//增加判断部分.

$a1 的类是 A, $a2 是 A
$a1 $a2 指向同一对象.

```

- 到此，我们写了一个最简单的 **单件模式**。
- 现在，你可以尝试写一个应用单件设计模式的数据库连接类。
- 要记住**单件模式**的使用效果和书写方式。

3.2 final 类、final 方法和常量

- final---用于类、方法前。
- final 类---不可被继承。
- final 方法---不可被覆盖。

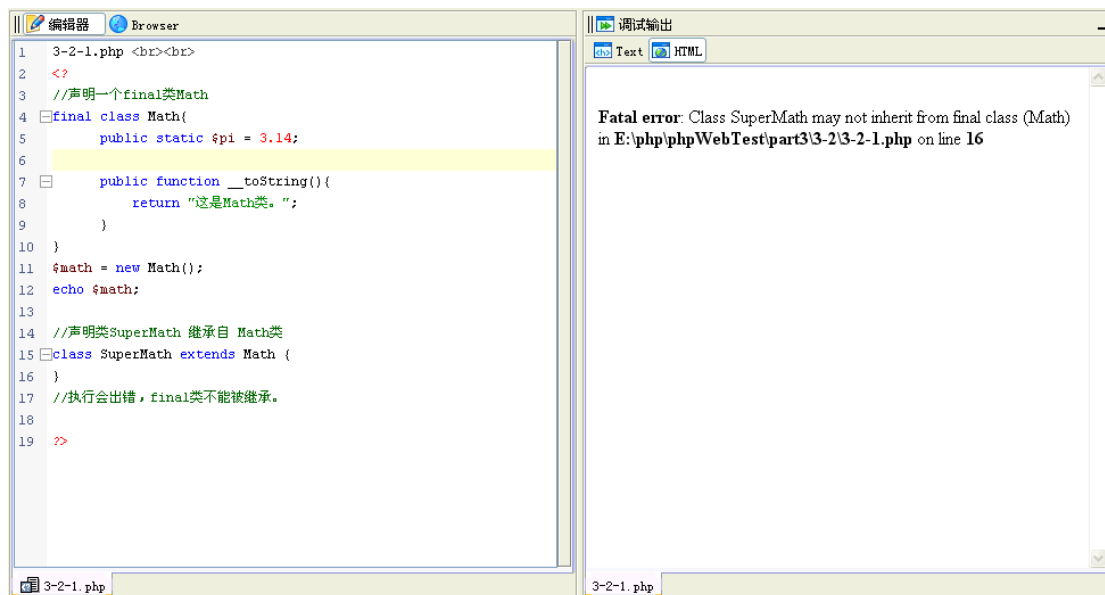
3.2.1 final 类的不能被继承

如果我们不希望一个类被继承，我们使用 final 来修饰这个类。

于是这个将无法被继承。

比如我们设定的 Math 类，涉及了我们要做的数学计算方法，这些算法也没有必要修改，也没有必要被继承，我们把它设置成 final 类型。

例 3-2-1.php



```
1 3-2-1.php <br><br>
2 <?
3 //声明一个final类Math
4 final class Math{
5     public static $pi = 3.14;
6
7     public function __toString(){
8         return "这是Math类。";
9     }
10 }
11 $math = new Math();
12 echo $math;
13
14 //声明类SuperMath 继承自 Math类
15 class SuperMath extends Math {
16 }
17 //执行会出错，final类不能被继承。
18
19 >>
```

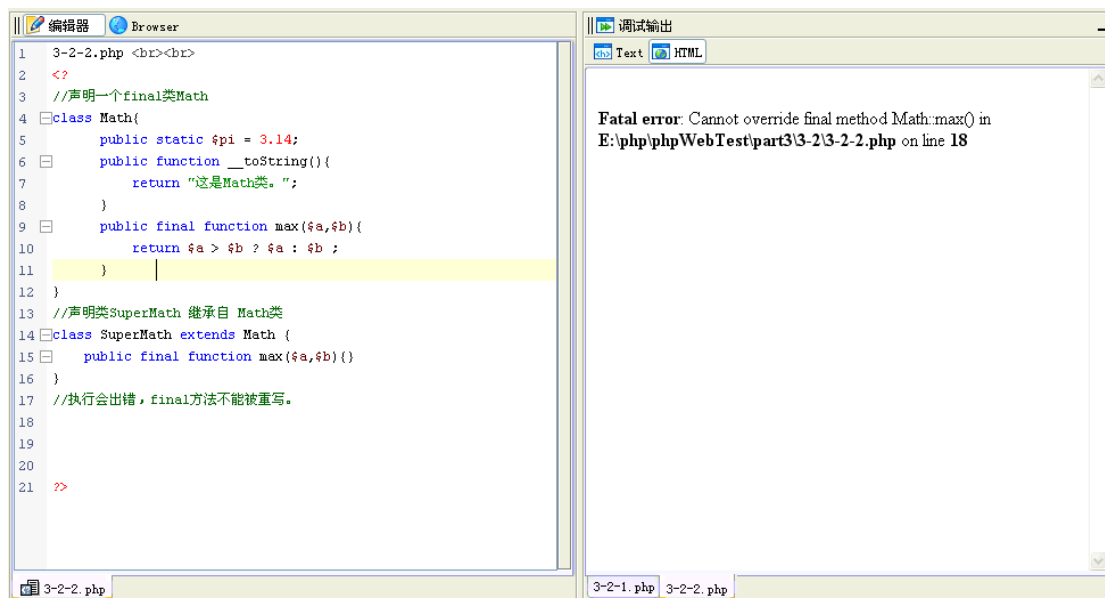
Fatal error: Class SuperMath may not inherit from final class (Math) in E:\php\phpWebTest\part3\3-2\3-2-1.php on line 16

3.2.2 final 方法不能被重写

如果不希望类中的某个方法被子类重写，我们可以设置这个方法为 `final` 方法，只需要在这个方法前加上 `final` 修饰符。

如果这个方法被子类重写，将会出现错误。

例：3-2-1.php



The screenshot shows a PHP IDE with two panes. The left pane is the code editor, and the right pane is the debug output window.

```
1 3-2-2.php <br><br>
2 <?
3 //声明一个final类Math
4 class Math{
5     public static $pi = 3.14;
6     public function __toString(){
7         return "这是Math类。";
8     }
9     public final function max($a,$b){
10        return $a > $b ? $a : $b ;
11    }
12 }
13 //声明类SuperMath 继承自 Math类
14 class SuperMath extends Math {
15     public final function max($a,$b){}
16 }
17 //执行会出错，final方法不能被重写。
18
19
20
21 >>
```

The right pane shows the following error message:

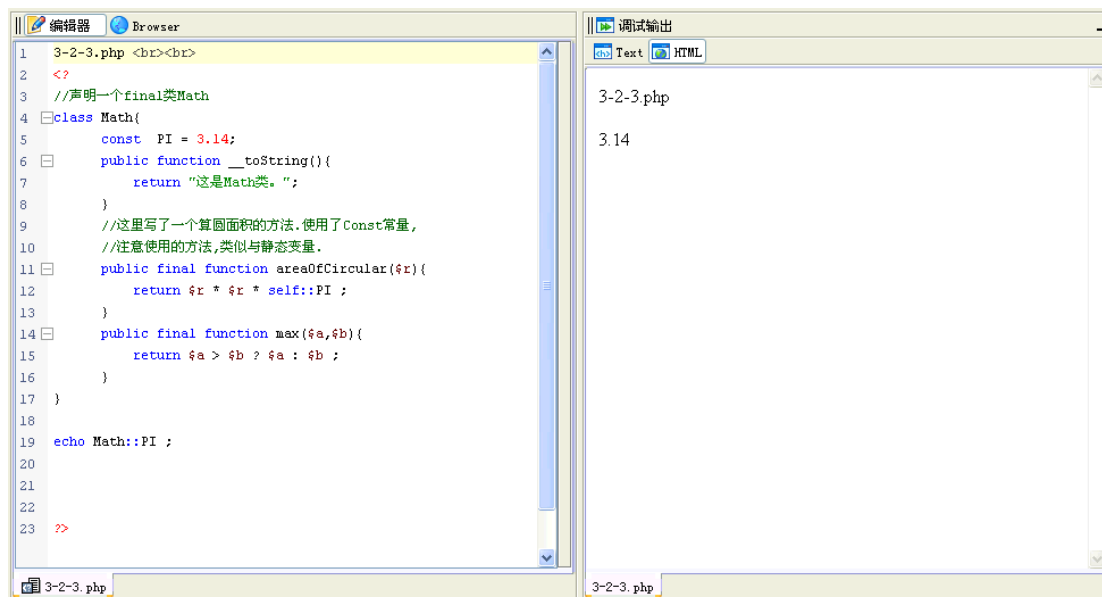
```
Fatal error: Cannot override final method Math::max() in
E:\php\phpWebTest\part3\3-2\3-2-2.php on line 18
```

3.2.3 PHP5 中的常量

- 在 PHP5 类中继续使用 `const` 修饰常量。
- 我们使用 `const` 定义一个常量，定义的这个常量不能被改变。
- 在 PHP5 中 `const` 定义的常量与定义变量的方法不同，不需要加 `$` 修饰符。
`const PI = 3.14;` 这样就可以。

- 而使用 `const` 定义的常量名称一般都大写，这是一个约定，在任何语言中都是这样。
- 如果定义的常量由多个单词组成，使用 `_` 连接，这也是约定。
- 比如，`MAX_MUMBER` 这样的命名方式。一个好的命名方式，是程序员必须注意的。
- 类中的常量使用起来类似静态变量，不同点只是它的值不能被改变。我们使用 `类名::常量名` 来调用这个常量。

例 3-2-3.php



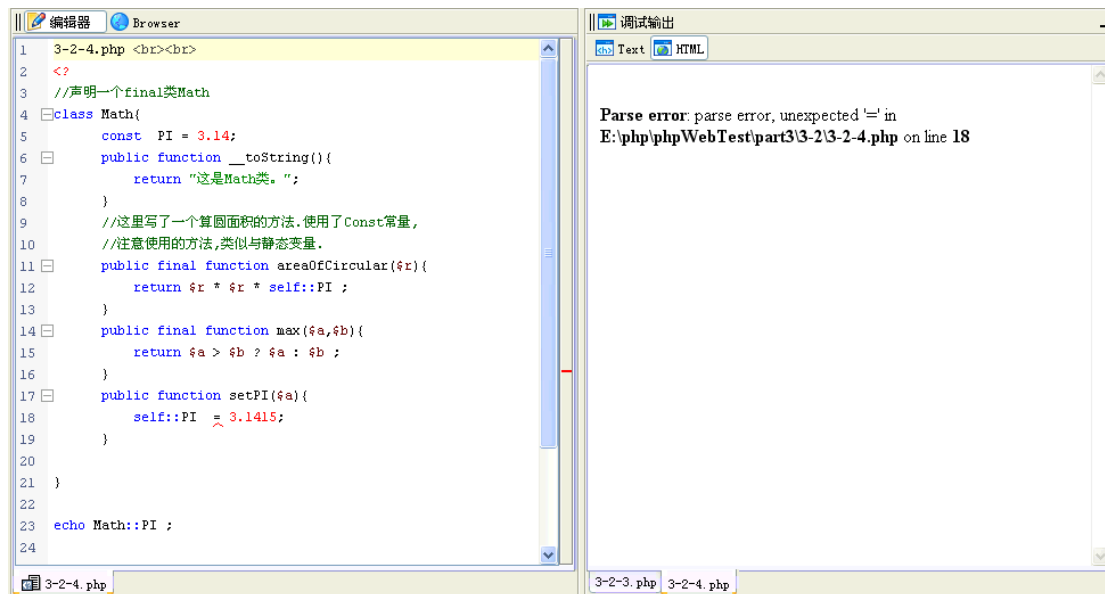
```
1 3-2-3.php <br><br>
2 <?
3 //声明一个final类Math
4 class Math{
5     const PI = 3.14;
6     public function __toString(){
7         return "这是Math类。";
8     }
9     //这里写了一个算圆面积的方法.使用了Const常量,
10    //注意使用的方法,类似与静态变量.
11    public final function areaOfCircular($r){
12        return $r * $r * self::PI ;
13    }
14    public final function max($a,$b){
15        return $a > $b ? $a : $b ;
16    }
17 }
18
19 echo Math::PI ;
20
21
22
23 ?>
```

调试输出

```
3-2-3.php
3.14
```

尝试为 const 定义的常量赋值，将会出现错误。

例 3-2-4.php



The screenshot shows a PHP IDE with two panes. The left pane is the editor, and the right pane is the debug output window.

```
1 3-2-4.php <br><br>
2 <?
3 //声明一个final类Math
4 class Math{
5     const PI = 3.14;
6     public function __toString(){
7         return "这是Math类。";
8     }
9     //这里写了一个算圆面积的方法.使用了Const常量,
10    //注意使用的方法,类似与静态变量.
11    public final function areaOfCircular($r){
12        return $r * $r * self::PI ;
13    }
14    public final function max($a,$b){
15        return $a > $b ? $a : $b ;
16    }
17    public function setPI($a){
18        self::PI = 3.1415;
19    }
20
21 }
22
23 echo Math::PI ;
24
```

The debug output window shows the following error message:

```
Parse error: parse error, unexpected '=' in
E:\php\phpWebTest\part3\3-2\3-2-4.php on line 18
```

3.3 abstract 类和 abstract 方法

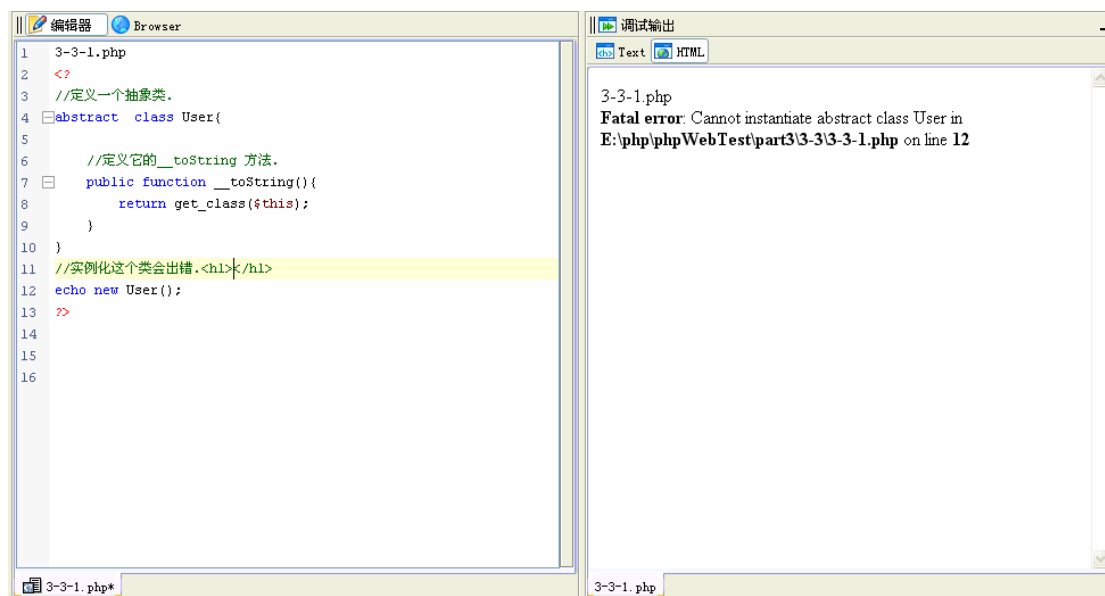
- 可以使用 **abstract** 来修饰一个类或者方法。
- 用 **abstract** 修饰的类表示这个类是一个抽象类，用 **abstract** 修饰的方法表示这个方法是一个抽象方法。
- 抽象类不能被实例化。
- 抽象方法是只有方法声明，而没有方法的实现内容。

3.3.1 abstract 抽象类

- 可以使用 **abstract** 来修饰一个类。
- 用 **abstract** 修饰的类表示这个类是一个抽象类。
- 抽象类不能被实例化。

这是一个简单抽象的方法，如果它被直接实例化，系统会报错。

例：3-3-1.php



The screenshot shows a PHP IDE with two windows. The left window is the '编辑器' (Editor) showing the code for '3-3-1.php'. The right window is '调试输出' (Debug Output) showing the error message.

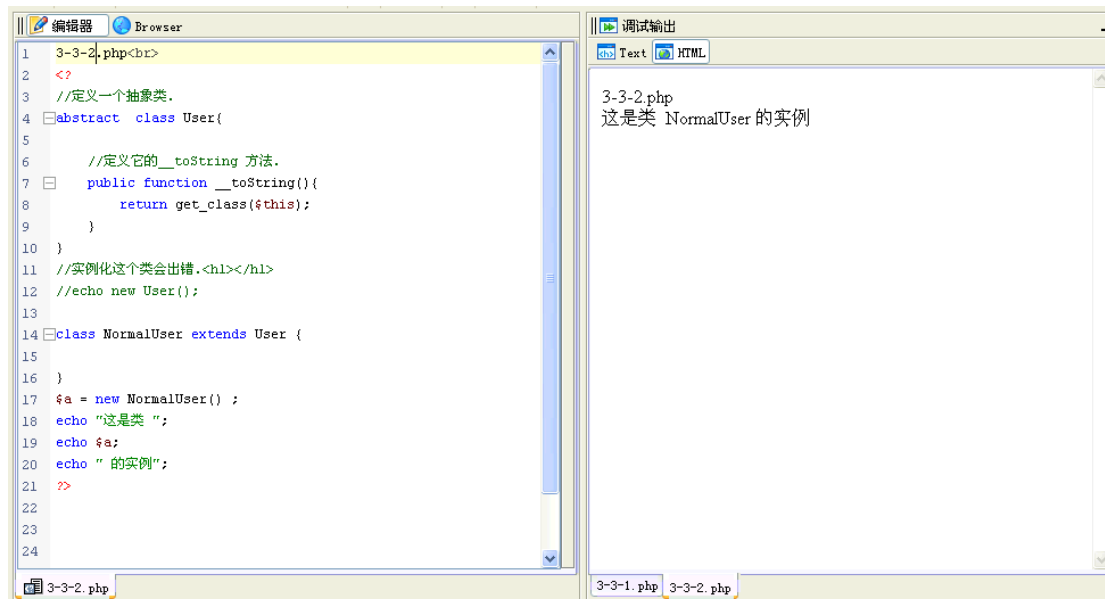
```
1 3-3-1.php
2 <?
3 //定义一个抽象类.
4 abstract class User{
5
6     //定义它的__toString方法.
7     public function __toString(){
8         return get_class($this);
9     }
10 }
11 //实例化这个类会出错.<h1>|</h1>
12 echo new User();
13 >>
14
15
16
```

调试输出

```
3-3-1.php
Fatal error: Cannot instantiate abstract class User in
E:\php\phpWebTest\part3\3-3\3-3-1.php on line 12
```

例：3-3-2.php

下面例子的 NormalUser 继承自 User 类，就可以被实例化了。



```
1 3-3-2.php<br>
2 <?
3 //定义一个抽象类.
4 abstract class User{
5
6     //定义它的__toString方法.
7     public function __toString(){
8         return get_class($this);
9     }
10 }
11 //实例化这个类会出错.<h1></h1>
12 //echo new User();
13
14 class NormalUser extends User {
15
16 }
17 $a = new NormalUser() ;
18 echo "这是类 ";
19 echo $a;
20 echo " 的实例";
21 >>
22
23
24
```

调试输出

```
3-3-2.php
这是类 NormalUser 的实例
```

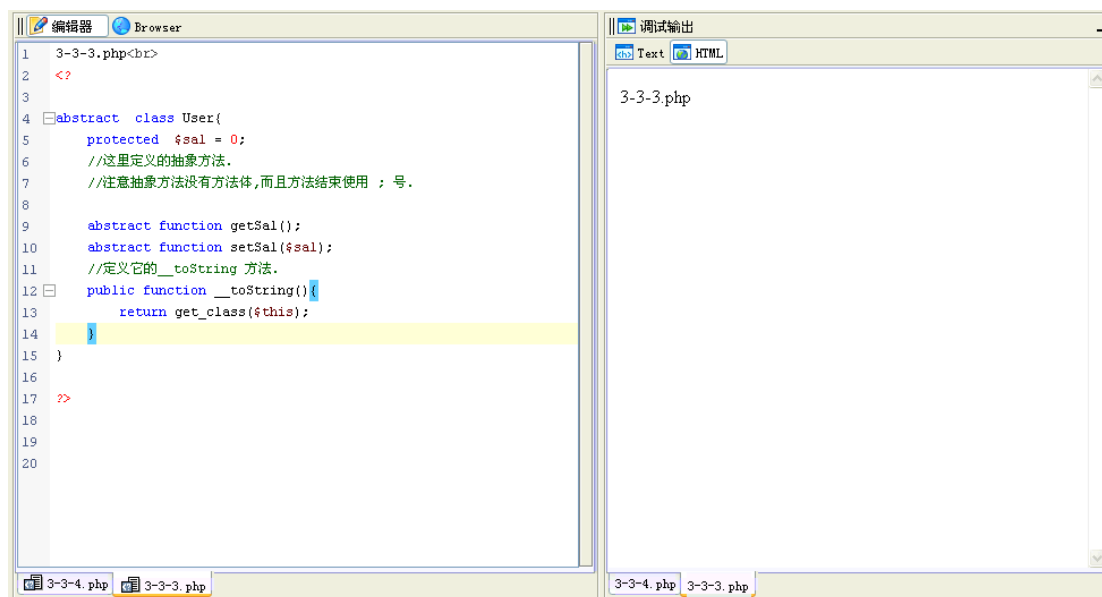
单独设置一个抽象类是没有意义的，只有有了抽象方法，抽象类才有了血肉。
下面介绍抽象方法。

3.3.2 abstract 抽象方法

- 用 **abstract** 修饰的类表示这个方法是一个抽象方法。
- 抽象方法，只有方法的声明部分，没有方法体。
- 抽象方法没有 **{ }**，而采用 **;** 结束。
- 一个类中，只要有一个抽象方法，这个类必须被声明为抽象类。
- 抽象方法在子类中必须被重写。

下面是一个抽象类，其中有两个抽象方法，分别是 `setSal()` 和 `getSal()`。用来取回\$sal 员工的工资。

例：3-3-3.php



```
1 3-3-3.php<br>
2 <?
3
4 abstract class User{
5     protected $sal = 0;
6     //这里定义的抽象方法.
7     //注意抽象方法没有方法体,而且方法结束使用 ; 号.
8
9     abstract function getSal();
10    abstract function setSal($sal);
11    //定义它的__toString 方法.
12    public function __toString(){
13        return get_class($this);
14    }
15 }
16
17 >>
18
19
20
```

调试输出

```
3-3-3.php
```

既然 User 类不能被直接继承，我们写一个 NormalUser 类继承自 User 类。

当我们写成如下代码时，系统会报错。

这个错误告诉我们，在 User 类中有两个抽象方法，我们必须在子类中重写这两个方法。

例：3-3-4.php

```

1 3-3-4.php<br>
2 <?
3 abstract class User{
4     protected $sal = 0;
5     //这里定义的抽象方法.
6     //注意抽象方法没有方法体,而且方法结束使用 ; 号.
7
8     abstract function getSal();
9     abstract function setSal($sal);
10    //定义它的__toString 方法.
11    public function __toString(){
12        return get_class($this);
13    }
14 }
15
16 class NormalUser extends User {
17 }
18
19
20
21
22
23

```

Fatal error: Class NormalUser contains 2 abstract methods and must therefore be declared abstract or implement the remaining methods (User::getSal, User::setSal) in E:\php\phpWebTest\part3\3-3\3-3-4.php on line 18

例：3-3-5.php

下面例子，重写了这两个方法，虽然方法体里面 {} 的内容是空的，也算重写了这个方法。注意看重写方法的参数名称，这里只要参数数量一致就可以，不要求参数的名称必须一致。

```

1 3-3-5.php<br>
2 <?
3 abstract class User{
4     protected $sal = 0;
5     //这里定义的抽象方法.
6     //注意抽象方法没有方法体,而且方法结束使用 ; 号.
7
8     abstract function getSal();
9     abstract function setSal($sal);
10    //定义它的__toString 方法.
11    public function __toString(){
12        return get_class($this);
13    }
14 }
15
16 class NormalUser extends User {
17     //虽然是空的方法体,但依然是重写了.
18     function getSal(){}
19     function setSal($sal){}
20 }
21 //这样就不会出错了.
22
23
24

```

3-3-5.php

下面 19-21 行，三种写重写的方式都会报错。

- 19 行，缺少参数。
- 20 行，参数又多了。
- 21 行，参数类型不对。（这种写法在以后章节介绍）

例:3-3-3.php

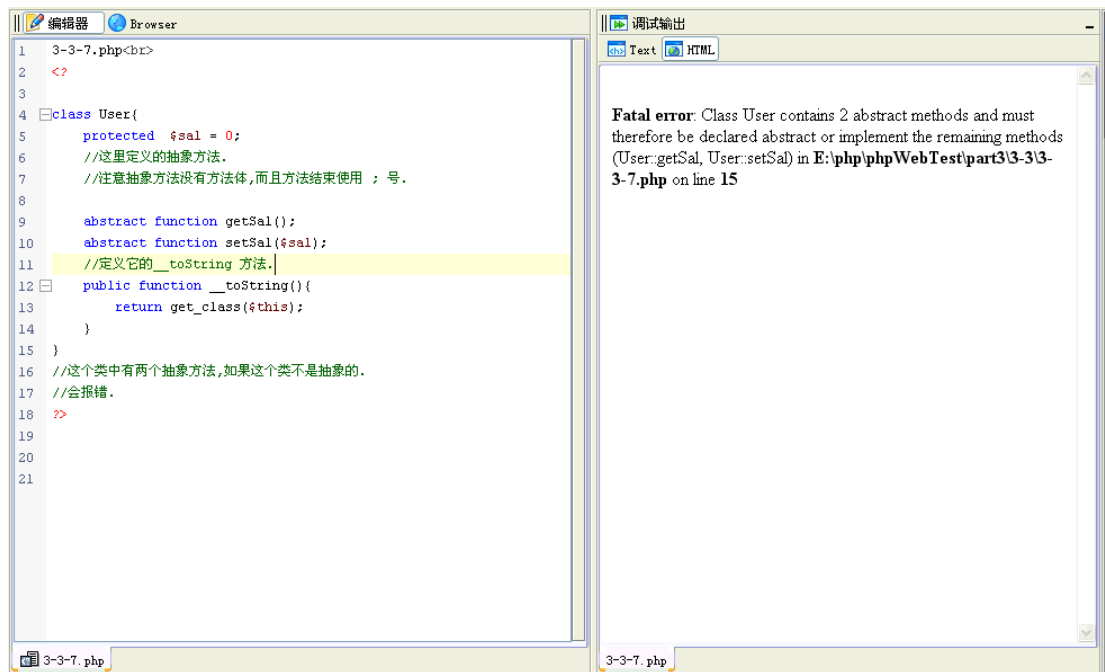
```
1 3-3-6.php<br>
2 <?
3 abstract class User{
4     protected $sal = 0;
5     //这里定义的抽象方法.
6     //注意抽象方法没有方法体,而且方法结束使用 ; 号.
7     abstract function getSal();
8     abstract function setSal ($sal);
9     //定义它的__toString 方法.
10 public function __toString(){
11     return get_class($this);
12 }
13 }
14
15 class NormalUser extends User {
16     //虽然是空的方法体,但依然是重写了.
17     function getSal(){
18
19         //function setSal(){
20         //function setSal($sal1,$sal2){
21         function setSal( User $sal){
22
23     }
24     //这样就不会出错了.
25 >>
26
27
28
```

Fatal error: Declaration of NormalUser::setSal() must be compatible with that of User::setSal() in E:\php\phpWebTest\part3\3-3\3-3-6.php on line 23

- 一个类中，如果有一个抽象方法，这个类必须被声明为抽象类。

下面这个类不是抽象类，其中定义了一个抽象方法，会报错。

例：3-3-7.php



The screenshot shows a PHP IDE with two windows. The left window is the editor for '3-3-7.php' and the right window is the '调试输出' (Debug Output) window.

```
1 3-3-7.php<br>
2 <?
3
4 class User{
5     protected $sal = 0;
6     //这里定义的抽象方法。
7     //注意抽象方法没有方法体,而且方法结束使用 ; 号。
8
9     abstract function getSal();
10    abstract function setSal($sal);
11    //定义它的__toString方法。
12    public function __toString(){
13        return get_class($this);
14    }
15 }
16 //这个类中有两个抽象方法,如果这个类不是抽象的。
17 //会报错。
18 ?>
19
20
21
```

The Debug Output window shows the following error message:

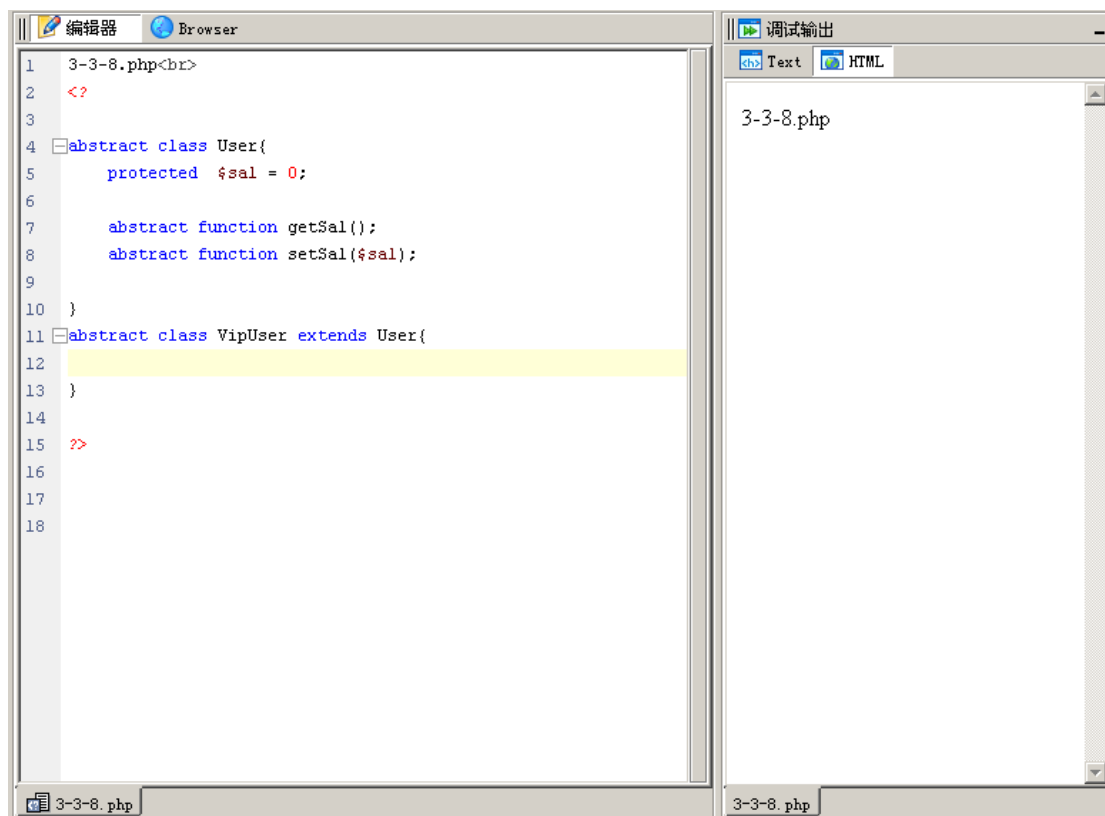
```
Fatal error: Class User contains 2 abstract methods and must therefore be declared abstract or implement the remaining methods (User::getSal, User::setSal) in E:\php\phpWebTest\part3\3-3-3-7.php on line 15
```

3.3.3 抽象类继承抽象类

- 抽象类继承另外一个抽象类时，不用重写其中的抽象方法。
- 抽象类中，不能重写抽象父类的抽象方法。
- 这样的用法，可以理解为对抽象类的扩展。

下面的例子，演示了一个抽象类继承自另外一个抽象类时，不需要重写其中的抽象方法。

例：3-3-8.php



```
1 3-3-8.php<br>
2 <?
3
4 abstract class User{
5     protected $sal = 0;
6
7     abstract function getSal();
8     abstract function setSal($sal);
9
10 }
11 abstract class VipUser extends User{
12
13 }
14
15 ?>
16
17
18
```

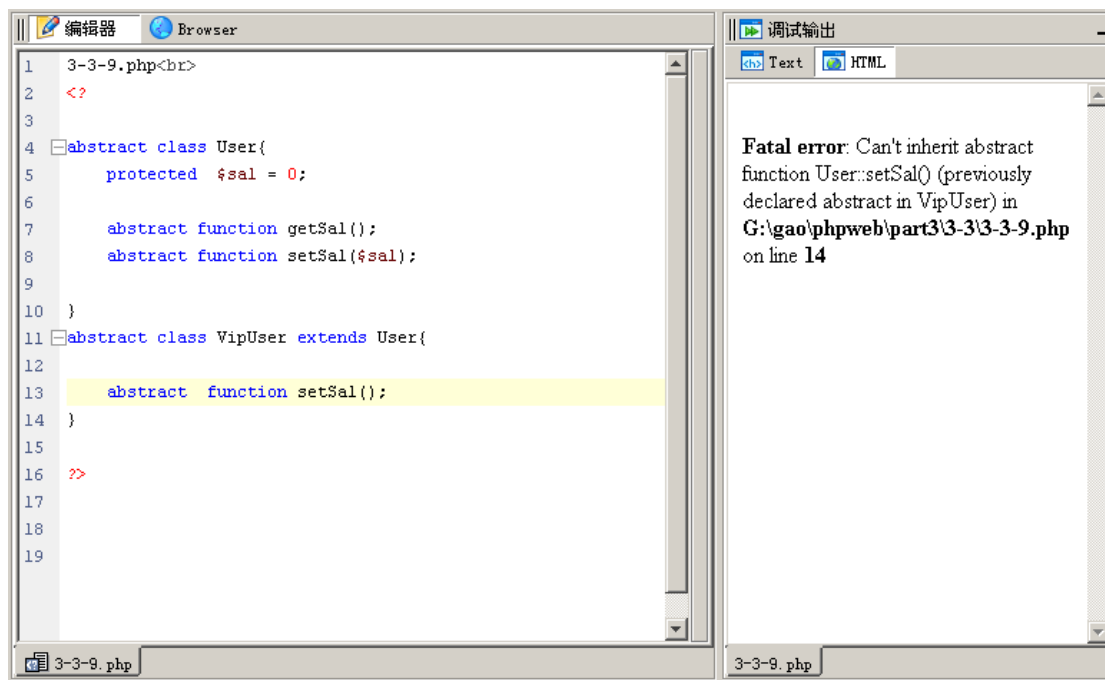
调试输出

3-3-8.php

抽象类在被继承后，其中的抽象方法不能被重写。

如果发生重写，系统会报错。

例 3-3-9.php

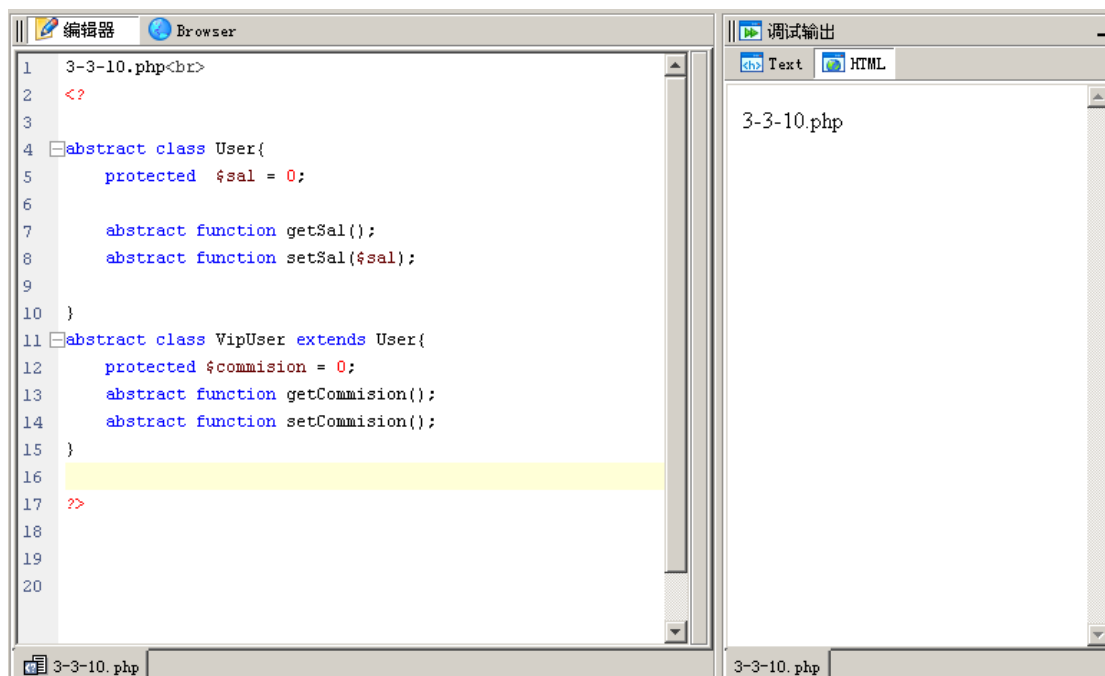


```
1 3-3-9.php<br>
2 <?
3
4 abstract class User{
5     protected $sal = 0;
6
7     abstract function getSal();
8     abstract function setSal($sal);
9
10 }
11 abstract class VipUser extends User{
12
13     abstract function setSal();
14 }
15
16 >>
17
18
19
```

Fatal error: Can't inherit abstract function User::setSal() (previously declared abstract in VipUser) in G:\gao\phpweb\part3\3-3\3-3-9.php on line 14

抽象类继承抽象类，目的对抽象类的扩展。

例 3-3-10.php



```
1 3-3-10.php<br>
2 <?
3
4 abstract class User{
5     protected $sal = 0;
6
7     abstract function getSal();
8     abstract function setSal($sal);
9
10 }
11 abstract class VipUser extends User{
12     protected $commission = 0;
13     abstract function getCommission();
14     abstract function setCommission();
15 }
16
17 >>
18
19
20
```

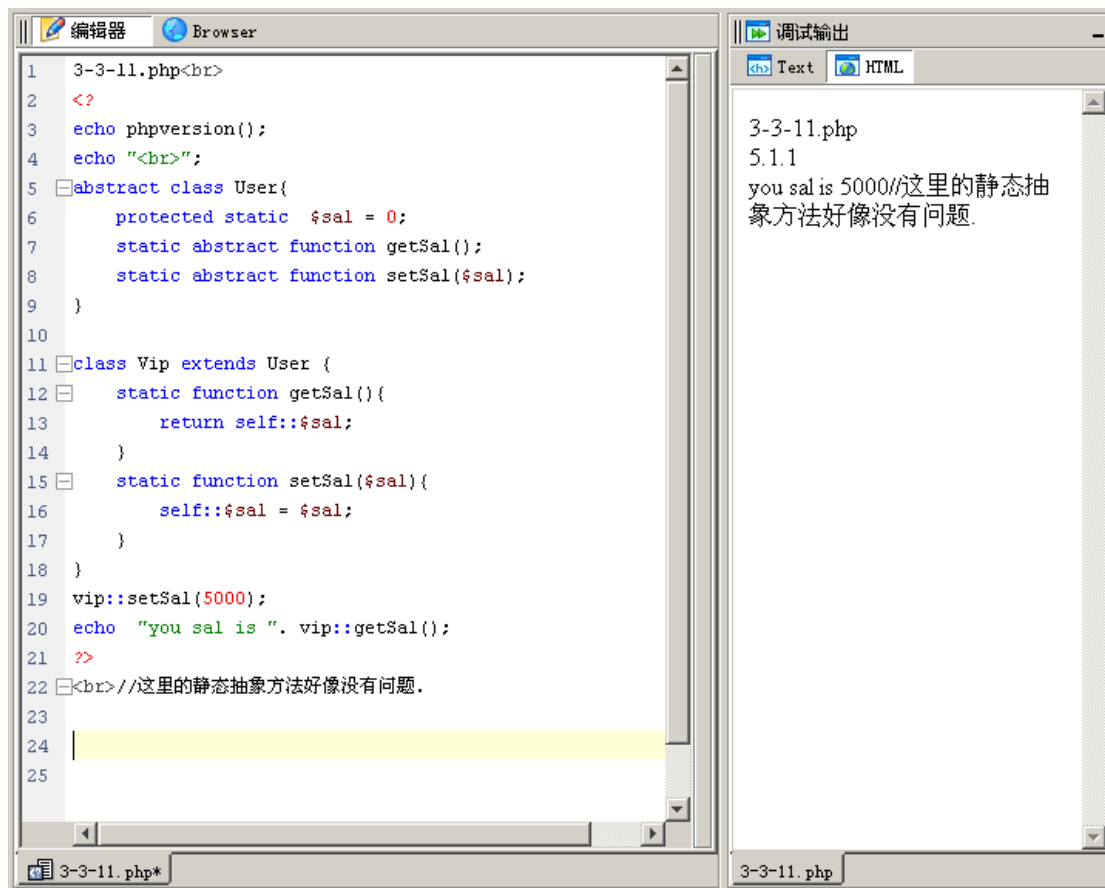
3-3-10.php

3.3.4 静态抽象方法

在 PHP5.1 中，抽象类中支持静态抽象方法。

下面这个例子，看到静态抽象方法可以声明。实现这个方法时，必须是静态的方法。

例：3-3-11.php



```
1 3-3-11.php<br>
2 <?
3 echo phpversion();
4 echo "<br>";
5 abstract class User{
6     protected static $sal = 0;
7     static abstract function getSal();
8     static abstract function setSal($sal);
9 }
10
11 class Vip extends User {
12     static function getSal(){
13         return self::$sal;
14     }
15     static function setSal($sal){
16         self::$sal = $sal;
17     }
18 }
19 vip::setSal(5000);
20 echo "you sal is ". vip::getSal();
21 ?>
22 <br> //这里的静态抽象方法好像没有问题。
23
24
25
```

调试输出

```
3-3-11.php
5.1.1
you sal is 5000//这里的静态抽象方法好像没有问题。
```

3.3.5 PHP5.2.0 中的静态抽象方法

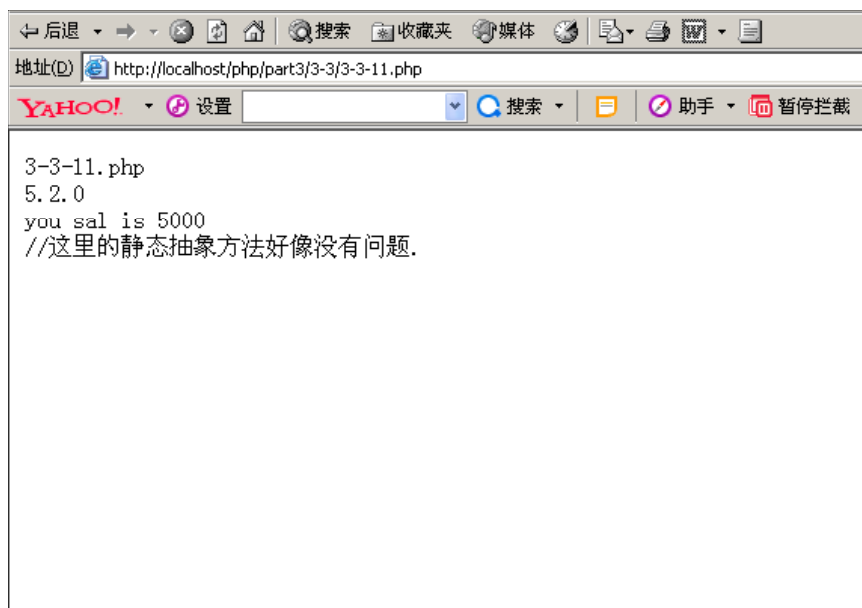
在 PHP5.2.0 的 更新信息中有这样一段话:

```
- Dropped abstract static class functions. (Marcus)

Due to an oversight, PHP 5.0 and 5.1 allowed abstract static functions in
classes. In PHP 5.2, only interfaces can have them.
```

译文: (因为疏漏, 在 PHP5.0 和 PHP5.1 的类中允许静态抽象方法。
在 PHP5.2 中, 只有接口可以拥有静态抽象方法。)

在 PHP5.2.0 版本中测试, 刚才的代码执行正常。在 PHP5.2.0 中是依然兼容。



3.4 设计模式之模版模式

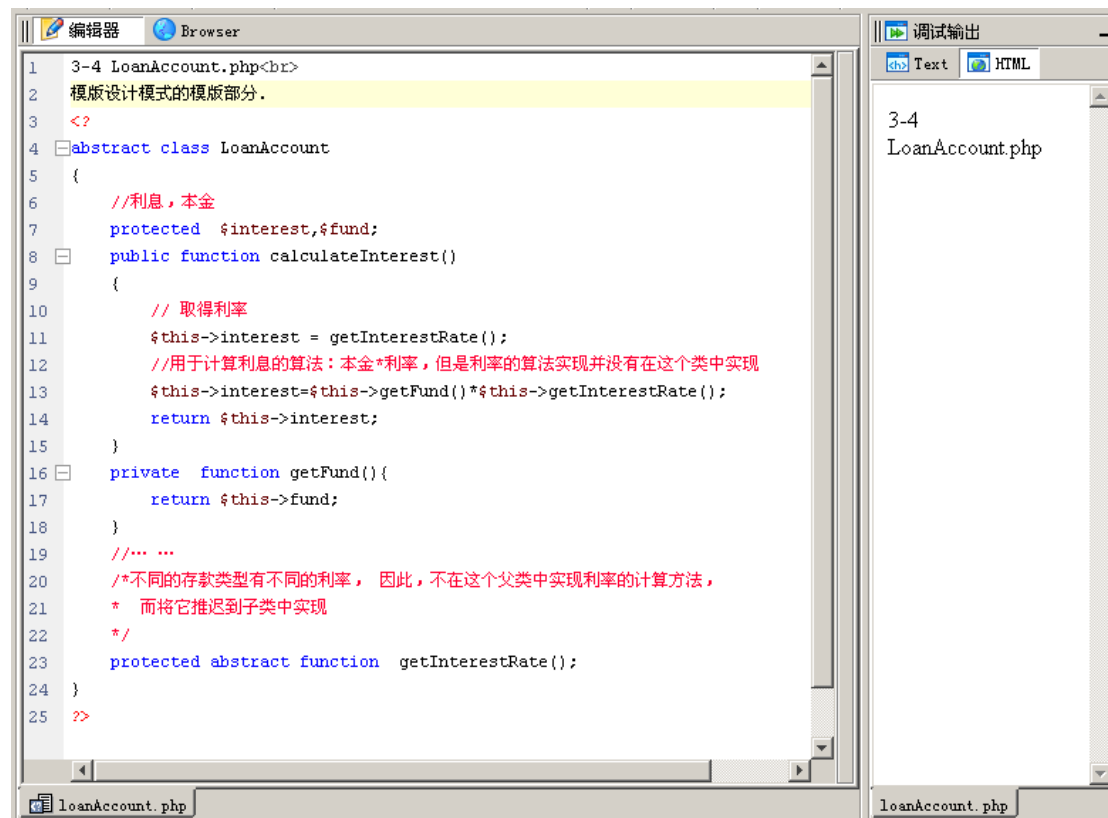
抽象类的应用就是典型的模版模式，先声明一个不能被实例化的模版，在子类中去依照模版实现具体的应用。

3.4.1 模版模式实例

我们写这样一个应用：

银行计算利息，都是利率乘以本金和存款时间，但各种存款方式计算利率的方式不同，所以在账户这个类的相关方法里，只搭出算法的骨架，但不具体实现。具体实现由各个子类来完成。

例：LoanAccount.php



```
1 3-4 LoanAccount.php<br>
2 模版设计模式的模版部分。
3 <?
4 abstract class LoanAccount
5 {
6     //利息，本金
7     protected $interest,$fund;
8     public function calculateInterest()
9     {
10         // 取得利率
11         $this->interest = getInterestRate();
12         //用于计算利息的算法：本金*利率，但是利率的算法实现并没有在这个类中实现
13         $this->interest=$this->getFund()*$this->getInterestRate();
14         return $this->interest;
15     }
16     private function getFund(){
17         return $this->fund;
18     }
19     //... ..
20     /*不同的存款类型有不同的利率，因此，不在这个父类中实现利率的计算方法，
21     * 而将它推迟到子类中实现
22     */
23     protected abstract function getInterestRate();
24 }
25 ?>
```

调试输出

```
3-4
LoanAccount.php
```

以后，所有和计算利息的类都继承自这个类，而且必须实现其中的 `getInterestRate()` 方法，这种用法就是模版模式。

下一章将介绍面向对象语言的其它重要组成 **接口、多态**。