

19/08/09 - PHP 5.3

Llego algo tarde, cosas del verano, pero voy a comentar algunas de las mejoras que vienen incluidas en la nueva liberación de PHP, la 5.3.

Para mi, esta versión, además de corregir fallos de la anterior, aporta características que le hacen acercarse aún más a la forma de programación profesional que viene dada con otros lenguajes como Java, C#, C++, Python, Ruby...

Espacios de nombres (namespaces)

Los espacios de nombres, llamados en otros lenguajes: paquetes, módulos, ...; con esto, podemos tener organizado nuestro código, no solo en directorios, sino también bien delimitado su espacio de nombres mediante el uso del *namespace*.

Por ejemplo, si tenemos dos directorios: *File* y *BD*; que pertenecen a un programa y cada uno de ellos tiene un archivo para contener la clase *Output*, en las versiones anteriores de PHP habría que incluir uno u otro, o modificar el nombre para que fuese *FileOutput* y *DBOutput*, redundando la ruta con el nombre de la clase.

Ahora, con el uso de namespaces, se puede delimitar con solo agregar una sección de código como esta:

```
namespace File {Array Array    class Output {Array          /*
... */Array    }Array}Array Arraynamespace {Array    $f = new
File\Output();Array}
```

Static... ahora sí

En las versiones anteriores de PHP, los valores *static* no eran tratados del todo bien, no permitiéndose algunos usos que parecían lógicos y no produciéndose, sobre todo en la herencia, los resultados que se esperaban.

Ahora, ya se permite el uso de variables para contener el nombre de la clase a la que llamar de forma estática, es decir, ya se permite este uso:

```
class A {Array    public static function say() {Array
```

```
echo "Hola\n";Array      }Array}Array Array$class =
"A";Array$class::say();
```

Así mismo, el uso del *late state binding*, es posible mediante la palabra clave *static*, de modo que, si se llama a un método estático como *static::metodo()* en lugar de como *self::metodo()*, se llama al método de la clase hija que haya sobrecargado al método que se llama. Un ejemplo:

```
class A {Array      public static function who() {Array
echo __CLASS__;Array      }Array      public static function
test() {Array      static::who(); // Here comes Late Static
BindingsArray      }Array}Array Arrayclass B extends A {Array
public static function who() {Array      echo
__CLASS__;Array      }Array}Array ArrayB::test(); // output: B
```

Por último, la función especial `__call` no se llama cuando no hay un método de clase que no exista, en su lugar se llamará a `__callStatic`, de modo que se pueda hacer diferencia de cuando se está llamando a un método de objeto y cuando se llama a un método de clase.

Más de Late State Binding

Hasta ahora, cuando se quería hacer una clase abstracta que llamase a funciones de sus clases hijas, que aún no han sido declaradas, éstas debían ser abstractas. Pero si la clase no era abstracta y el método a llamar no constaba como abstracto, la llamada no se realizaba a esa función, sino a la del padre.

El sistema de *late state binding*, asegura que la llamada a métodos se hará siempre de abajo a arriba, es decir, si existe un método en la clase hija que se instanció, aunque el método en ejecución sea el del padre, el método que se llama es el de la clase hija. Más o menos lo que se vió con los métodos estáticos, pero esta vez con métodos.

Funciones anónimas

El uso de funciones anónimas (*closures* o *lambda*) es una técnica de programación que permite completar código escrito, mediante el desarrollo parcial de algún algoritmo. Imagina que quieres hacer un código en el que tengas que hacer algo, un paso inicial, un paso intermedio y, en cada iteración una ejecución específica,

para terminar con un último paso tras esa iteración.

El código concreto dentro de la iteración puede variar... y de hecho variará en cada implementación, pero el resto no, el resto se mantiene de forma fija. Pues, se puede implementar el esqueleto del programa, y aceptar como parámetro de la función que se haga, una variable que contenga el código a ejecutar. La función que completaría nuestro código se puede hacer así:

```
$code = function ($dato1, $dato2) {Array      echo $dato1 . "--"
$dato2 .
"\n";Array};Array Arrayalgoritmo($code);Array Arrayfunction
algoritmo( $func ) {Array      for ($i=0; $i<100; $i++)
{Array          $func($i, $i*$i);Array      }Array}
```

También se pueden emplear funciones como *array_walk*, *preg_replace_callback*, *uasort*, etc.

Esto permite realizar códigos como los que se realizan en Ruby o en los lenguajes funcionales y declarativos.

Recolector de Basura de Referencias Circulares opcional

Se deja al programador la decisión de si quiere activar el recolector de referencias circulares del *garbage collector*, o no. Por defecto viene activado y actúa de la siguiente forma, con este código:

```
class A {Array      function __construct () {Array
$this->b = new B($this);Array      }Array}Array Arrayclass B
{Array      function __construct ($parent = NULL) {Array
$this->parent = $parent;Array      }Array}Array Arrayfor ($i = 0
; $i < 1000000 ; $i++) {Array      $a = new A();Array
unset($a);Array}Array Arrayecho
number_format(memory_get_usage());
```

Si se ejecuta desde consola, con una versión de PHP anterior a la 5.3, se verá en pantalla un error fatal, de que el límite de memoria se ha superado.

Esto es debido a que la liberación de memoria, cuando se va a proceder a liberar la clase B, ve que tiene una referencia a A, que ya va a ser liberada y crea un ciclo,

con lo que, para evitarlo, no libera B. En las versiones de PHP 5.3 en adelante, se detecta este ciclo como tal y se liberan ambas.

También existe la posibilidad de desactivar este comportamiento o ver si está activo, con las funciones `gc_enable`, `gc_enabled` y `gc_disable`.

Nowdoc y Heredoc

Hasta ahora, los bloques de tipo *heredoc* eran los únicos que permitían escribir de forma libre un texto para después usarlo como variable. Ahora también disponemos de los *nowdoc*, que son iguales, salvo que no se hace *parseo* de variables. Un ejemplo:

```
$hola = "Hi, ";
Array Array$hd = <<<ENDArraytexto
$holaArrayEND;
Array Array$nd = <<<'END'Arraytexto
$holaArrayEND;
Array Arrayecho $hd; // output: texto
Hi,Arrayecho $nd; // output: texto $hola
```

Constantes

Ahora, la palabra clave *const* puede ser empleada fuera del alcance de una clase, con lo que, en lugar de usar *define* se puede emplear esta forma:

```
// antesArray
// define("CONSTANTE", "Hola
mundo!");
Array Array
// ahoraArray
const CONSTANTE = "Hola
mundo!";
```

Operador Ternario simplificado

El operador ternario $(expr1)?(expr2):(expr3)$ ahora permite dejar vacío el espacio correspondiente a $(expr2)$ de modo que si $(expr1)$ es verdadero, se retorna $(expr1)$ y si es falso, se retorna $(expr3)$.

Nuevos Módulos

PHPar

Al igual que JAR, PHPar sirve para empaquetar los PHP en un solo fichero, con lo que se mejora el despliegue de las aplicaciones, la organización del código, etc.

Intl

Mejores funciones de internacionalización para PHP. Hasta ahora, en PHP el único soporte de internacionalización que había disponible era *gettext*, ahora, con el uso de Intl y sus clases, se facilita internacionalizar una aplicación web, ya que tiene soporte para numeración, fecha, etc.

FileInfo

Da información sobre ficheros, usando la cabecera *magic* del propio fichero, intenta localizar de forma heurística su tipo, pudiendo retornarlo en formato MIME.

Etiquetas de Salto

Bueno, como en todo, hay avances y *retrocesos*, esto de proporcionar a PHP un elemento de código *spaguetti* como es el goto, no hace sino potenciar los malos usos, por lo que yo descartaría su uso.

Migración

Como cualquier liberación de lenguaje, PHP viene con mejoras, agregados y, además, cambios que implican que códigos anteriores puedan dejar de funcionar, con lo que es aconsejable leerse bien la guía de migración, para ver los cambios que se introducen, lo que llega a ser *deprecated*, etc.

Conclusiones

Con esta liberación, PHP da varios matices que le hacen acercarse más a lenguajes como C++ y Java, agregando cosas como PHPar y los namespaces, así como mejorando su implementación de POO.

Considero que el cambiar a esta versión y desarrollar con las nuevas implementaciones hará que los códigos desarrollados sean más claros y, sobre todo, más organizados, por lo que es una buena baza para realizar el cambio.

Además, viendo la guía de migración, se deja entrever que los cambios de la

versión anterior, 5.2.x, a esta nueva rama, son mínimos, con lo que, si se ha desarrollado código acorde a la versión anterior, realizar los cambios para adecuarse a esta versión, no es nada complicado.

No hay entradas relacionadas