Typage en PHP comment ça fonctionne ?

Dissection et explication du système de type de PHP

George Peter Banyard

14 Octobre 2022

The PHP Foundation

À propos de moi

Twitter: @Girgias

GitHub: Girgias

Site: *https://gpb.moe*

The PHP Foundation

- A étudié les maths pures
- Core dev de PHP financé à temps partiel par The PHP Foundation
- Mainteneur principal de la documentation FR de PHP
- Auteur des RFCs:
  - Saner Numeric Strings [5]
  - Pure Intersection Types [4]
  - DNF Types [3]
  - Allow null and false as stand-alone types [2]
  - True type [1]
  - Et d'autres...

Table des matières

Terminologie et Symboles

Terminologie

Un système de type est un système logique constitué d'un ensemble de règles qui assigne la propriété appelée un ***type*** *à chaque "terme" (un mot, phrase, ou autre ensemble de symbole). Généralement les termes sont des constructions diverses d'un programme informatique, tels que les variables, expressions, fonctions, ou modules. Un système de type dicte les* ***opérations*** *qui peuvent être réalisées sur un terme. Pour les variables, le système de type détermine les valeurs permissent de ce terme. [9]*

Terminologie et Symboles

Symboles mathématiques :

$\forall$ Pour tout

$\exists$ Il existe

$s \in S$ s appartient à S

Relations entre Types :

$U <: V$ U est un sous-type de V

$X :> Y$ X est un super-type de Y

Types Fondamentaux :

Type universel $\top$: $\forall T, T <: \top$

Type unitaire () : type qui n'a qu'une seule valeur

Type vide $\bot$: $\forall T, T :> \bot$

Système de Type de PHP

Système de Type de PHP

En PHP un paramètre de fonction, la valeur de retour d'une fonction, ou une propriété d'une classe peut définir un type.

Les types disponibles sont :

- Types Primitifs
- Types définit en espace Utilisateur
- Types Littéraux
- Type **`callable`**
- Types Composés
- Alias de Type

Types Primitifs

Type universel	**`mixed`** (PHP 8.0)
Type objet	**`object`** (PHP 7.2)
Type tableau	**`array`**
Types scalaires	**`bool`**, **`int`**, **`float`**, **`string`**
Type unitaire	**`null`** (PHP 8.0*)
Type vide	**`never`** (PHP 8.1)

Et un type de retour spécial:

`void` (PHP 7.1)

Types définis en espace Utilisateur

Aussi appelés class-types, ce sont les:

- Interfaces
- Classes
- Énumérations (PHP 8.1)

Types de classe relatifs :

- `self`
- `parent`
- `static` (PHP 8.0 en type de retour uniquement)

Types littéraux

Un type littéral est un sous-type concret d'un type, c.à.d. la valeur d'un type.

- `false` (PHP 8.0*)
- `true` (PHP 8.2)

Attention
Il est impossible de définir un type littéral en espace utilisateur.
Créez une énumération à la place

Type `callable`

Type qui représente une fonction:

- Une chaîne de caractères: *`"strlen"`*
- Une paire objet/nom de méthode: *`[$instance, "method"]`*
- Un objet qui implémente *`__invoke()`*
- Une Closure, récupérable avec la syntaxe: *`strlen(...)`* (PHP 8.1)

Attention
Il est impossible de définir une propriété de classe comme **`callable`**

Types composés

Un type composé est un type formé de plusieurs autres types.

Type d'intersection : `A&B` (PHP 8.1)

Type d'union simple : `T|U` (PHP 8.0)

Type FND : `(X&Y)|(V&W)` (PHP 8.2)

Forme Normale Disjonctive
En calcul des propositions, une Forme Normale Disjonctive ou FND (en anglais, *disjunctive normal form* ou *DNF*) est une normalisation d'une expression logique qui est une disjonction de clauses conjonctives. [6]

Alias de type

À partir de PHP 8.2, **`iterable`** est un alias de type résolu lors de la compilation.

$$\texttt{iterable} := \texttt{Traversable}|\texttt{array}$$

Avant c'était un pseudo-type primitif.

Attention
Il est impossible de définir un alias de type en espace utilisateur.

Implémentation: Vérification du type d'une valeur

Chaque valeur en PHP est représentée par une Zval.

```c
struct _zval_struct {
  zend_value value;
  union {
    uint32_t type_info;
    struct {
     /* ... */
    } v;
  } u1;
  /* ... */
};
```

```c
#define IS_UNDEF        0
#define IS_NULL         1
#define IS_FALSE        2
#define IS_TRUE         3
#define IS_LONG         4
#define IS_DOUBLE       5
#define IS_STRING       6
#define IS_ARRAY        7
#define IS_OBJECT       8
#define IS_RESOURCE     9
#define IS_REFERENCE    10
#define IS_CONSTANT_AST 11
/* ... */
```

Implémentation: Vérification du type d'un objet

Structure de classe

```
struct _zend_class_entry {
  char type;
  zend_string *name;
  zend_class_entry *parent;
  uint32_t ce_flags;
  uint32_t num_interfaces;
  zend_class_entry **interfaces;
  /* ... */
};
```

Implémentation de `instanceof`

```
static zend_always_inline bool
instanceof_function(
  const zend_class_entry *instance_ce,
  const zend_class_entry *ce
) {
  return instance_ce == ce
    || instanceof_function_slow(
                instance_ce, ce);
}
```

```
ZEND_API bool ZEND_FASTCALL instanceof_function_slow(
  const zend_class_entry *instance_ce, const zend_class_entry *ce)
{
  if (ce->ce_flags & ZEND_ACC_INTERFACE) {
    if (instance_ce->num_interfaces) {
      for (uint32_t i = 0; i < instance_ce->num_interfaces; i++) {
        if (instance_ce->interfaces[i] == ce) {
          return 1;
        }
      }
    }
    return 0;
  } else {
    while (1) {
      instance_ce = instance_ce->parent;
      if (instance_ce == ce) {
        return 1;
      }
      if (instance_ce == NULL) {
        return 0;
      }
    }
  }
}
```

zend_type la représentation d'un type en interne

Le type d'un paramètre, valeur de retour ou d'une propriété est représenté par un zend_type.

```
typedef struct {
  void *ptr;
  uint32_t type_mask;
} zend_type;
```

ptr est soit une classe sous forme de chaîne soit une liste de type:

```
typedef struct {
  uint32_t num_types;
  zend_type types[1];
} zend_type_list;
```

Masque de bit pour les types primitifs

```
#define MAY_BE_UNDEF     (1 << IS_UNDEF)
#define MAY_BE_NULL      (1 << IS_NULL)
#define MAY_BE_FALSE     (1 << IS_FALSE)
#define MAY_BE_TRUE      (1 << IS_TRUE)
#define MAY_BE_BOOL      (MAY_BE_FALSE|MAY_BE_TRUE)
#define MAY_BE_LONG      (1 << IS_LONG)
#define MAY_BE_DOUBLE    (1 << IS_DOUBLE)
#define MAY_BE_STRING    (1 << IS_STRING)
#define MAY_BE_ARRAY     (1 << IS_ARRAY)
#define MAY_BE_OBJECT    (1 << IS_OBJECT)
#define MAY_BE_RESOURCE  (1 << IS_RESOURCE)
#define MAY_BE_ANY
↪    (MAY_BE_NULL|MAY_BE_FALSE|MAY_BE_TRUE|⌋
↪    MAY_BE_LONG|MAY_BE_DOUBLE|MAY_BE_STRING|⌋
↪    MAY_BE_ARRAY|MAY_BE_OBJECT|MAY_BE_RESOURCE)
#define MAY_BE_CALLABLE  (1 << IS_CALLABLE)
#define MAY_BE_VOID      (1 << IS_VOID)
#define MAY_BE_NEVER     (1 << IS_NEVER)
#define MAY_BE_STATIC    (1 << IS_STATIC)
```

Héritage et Principe de Substitution de Liskov

Principe de Substitution de Liskov

Le principe de substitution de Liskov (en anglais *Liskov Substitution Principle* ou LSP) est, en programmation orientée objet, une définition particulière de la notion de sous-type. Il a été formulé par Barbara Liskov et Jeannette Wing en 1993. [7]

La formulation condensée est:

> *Si* $\phi(\mathsf{x})$ *est une propriété démontrable pour tout objet* x *de type* T*, alors* $\phi(\mathsf{y})$ *est vraie pour tout objet* y *de type* S *tel que* S *est un sous-type de* T

Principe de Substitution de Liskov: Simplifié

LSP est un principe à propos du remplacement d'un type par un autre tel que les interactions avant et après ne soient pas perturbées.

Les pré-conditions ne peuvent pas être renforcées dans une sous-type

Les post-conditions ne peuvent pas être affaiblies dans un sous-type

Les invariants doivent être conservé dans un sous-type

Principe de Substitution de Liskov: Simplifié

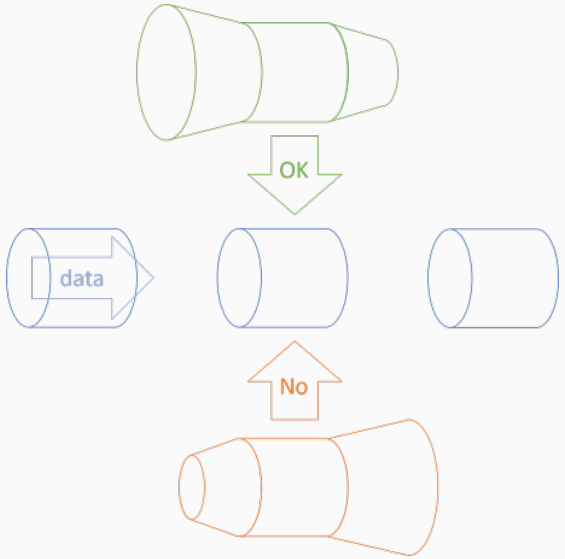

Figure 1: Visualisation du Principe du Substitution de Liskov comme un tuyau. [8]

Effet de LSP sur les signatures

Les méthodes ne peuvent pas ajouter de paramètres obligatoires.

Le type des paramètres des méthodes doit être *contra-variant*,
c.à.d. un super-type.

Le type de retour des méthodes doit être *co-variant*,
c.à.d. un sous-type.

Le type des propriétés doit être *co* **et** *contra-variant*.

Implémentation

Pour la vérification de la compatibilité des types on a juste besoin de:

```
static inheritance_status zend_perform_covariant_type_check(
    zend_class_entry *fe_scope, zend_type fe_type,
    zend_class_entry *proto_scope, zend_type proto_type)
{
    /* ... */
}
```

où *fe* représente le sous-type, et *proto* le super-type

Implémentation: covariance de types primitifs

```c
/* Apart from void, everything is trivially covariant to the mixed type.
 * Handle this case separately to ensure it never requires class loading. */
if (ZEND_TYPE_PURE_MASK(proto_type) == MAY_BE_ANY &&
        !ZEND_TYPE_CONTAINS_CODE(fe_type, IS_VOID)) {
    return INHERITANCE_SUCCESS;
}

/* Builtin types may be removed, but not added */
uint32_t fe_type_mask = ZEND_TYPE_PURE_MASK(fe_type);
uint32_t proto_type_mask = ZEND_TYPE_PURE_MASK(proto_type);
uint32_t added_types = fe_type_mask & ~proto_type_mask;
if (added_types) {
    if ((added_types & MAY_BE_STATIC) &&
            zend_type_permits_self(proto_type, proto_scope, fe_scope)) {
        /* Replacing type that accepts self with static is okay */
        added_types &= ~MAY_BE_STATIC;
    }
    if (added_types == MAY_BE_NEVER) {
        /* never is the bottom type */
        return INHERITANCE_SUCCESS;
    }
    if (added_types) {
        /* Otherwise adding new types is illegal */
        return INHERITANCE_ERROR;
    }
}
```

U un type d'union, est ce que $U <: V$?

$U_i \in U, V_j \in V$

$$\text{si } \forall U_i, \exists V_j : U_i <: V_j \implies U_1 \mid \cdots \mid U_n <: V_1 \mid \cdots \mid V_m \quad (1)$$

$$\text{si } \forall U_i, \forall V_j : U_i <: V_j \implies U_1 \mid \cdots \mid U_n <: V_1 \,\&\, \cdots \,\&\, V_m \quad (2)$$

On doit itérer sur l'ensemble de type U d'abord, et juste vérifier si chaque U_i est un sous-type de V.

Implémentation: U un type d'union, est-ce que $U <: V$?

```
early_exit_status = INHERITANCE_ERROR;
ZEND_TYPE_FOREACH(fe_type, single_type) {
    inheritance_status status;
    /* Union has an intersection type as it's member */
    if (ZEND_TYPE_IS_INTERSECTION(*single_type)) {
        status = zend_is_intersection_subtype_of_type(
            fe_scope, *single_type, proto_scope, proto_type);
    } else {
        zend_string *fe_class_name = get_class_from_type(fe_scope, *single_type);
        if (!fe_class_name) {
            continue;
        }

        status = zend_is_class_subtype_of_type(
            fe_scope, fe_class_name, proto_scope, proto_type);
    }

    if (status == early_exit_status) {
        return status;
    }
} ZEND_TYPE_FOREACH_END();
```

U un type d'intersection, est-ce que $U <: V$?

Un type d'intersection en PHP est forcément un objet, donc si $\texttt{object} \in V$ alors $U <: V$.

Dans le cas général: $U_i \in U, V_j \in V$

$$\text{si } \exists V_j, \exists U_i : U_i <: V_j \implies U_1 \,\&\, \cdots \,\&\, U_n <: V_1 \mid \cdots \mid V_m \tag{3}$$

$$\text{si } \forall V_j, \exists U_i : U_i <: V_j \implies U_1 \,\&\, \cdots \,\&\, U_n <: V_1 \,\&\, \cdots \,\&\, V_m \tag{4}$$

Comme l'*ordre* des quantificateurs est inversé comparé au cas d'un type d'union, on doit d'abord itérer sur V. Si V est un type d'union, il suffit d'une seule paire $(i; j)$ qui satisfait $U_i <: V_j$ pour que $U <: V$. Dans le cas contraire il faut que chaque V_j soit satisfait par au moins un U_i pour que $U <: V$.

Implémentation: U un type d'intersection, est ce que $U <: V$?

```
static inheritance_status zend_is_intersection_subtype_of_type(
    zend_class_entry *fe_scope, zend_type fe_type,
    zend_class_entry *proto_scope, zend_type proto_type)
{
    inheritance_status early_exit_status =
        ZEND_TYPE_IS_INTERSECTION(proto_type) ? INHERITANCE_ERROR : INHERITANCE_SUCCESS;
    ZEND_TYPE_FOREACH(proto_type, single_type) {
        inheritance_status status;

        if (ZEND_TYPE_IS_INTERSECTION(*single_type)) {
            status = zend_is_intersection_subtype_of_type(
               fe_scope, fe_type, proto_scope, *single_type);
        } else {
            zend_string *proto_class_name = get_class_from_type(proto_scope, *single_type);
            if (!proto_class_name) {
                continue;
            }

            zend_class_entry *proto_ce = NULL;
            status = zend_is_intersection_subtype_of_class(
                fe_scope, fe_type, proto_scope, proto_class_name, proto_ce);
        }

        if (status == early_exit_status) {
            return status;
        }
    } ZEND_TYPE_FOREACH_END();
    return early_exit_status == INHERITANCE_ERROR ? INHERITANCE_SUCCESS : INHERITANCE_ERROR;
}
```

Le futur?

Le futur?

Alias de type en espace utilisateur

```
typedef numeric int|float
```

Type de fonction

```
foo(fn<int,string>:bool $callable) {}
```

Type générique

```
class Collection<T> {
    private array<T> $stack = [];
    public function add(T $v) {
        $this->stack[] $v;
    }
}
```

Paramètres *in-out*

```
function foo(inout array $v) { $v = 5; }
$a = [];
foo($a); // TypeError: inout argument 1 passed to foo()
↪  must be of the type array, int assigned
```

Merci beaucoup!

Twitter: @Girgias

GitHub: Girgias

Site: `https://gpb.moe`

Feedback:

Coercitions de types scalaire

Par défaut PHP coerce les valeurs scalaires vers un type scalaire autorisé. S'il y a plusieurs types scalaires alors l'ordre est le suivant :

1. **`int`**
2. **`float`**
3. **`string`**
4. **`bool`**

Note
Si la valeur est une **`string`** et le type déclaré a **`int`** et **`float`** alors la sémantique de chaînes numérique décide du type de destination.

Note
Les fonctions internes coerce **`null`**, obsolète à partir de PHP 8.1.

`strict_types`

Que fait *`strict_types`* ?

`strict_types`

Que fait *`strict_types`* ?

Plus de coercitions pour les types scalaires dans les cas suivant uniquement :

- Appel de fonction dans le fichier qui définit *`strict_types`*
- Valeur de retour pour une fonction définit en espace utilisateur
- Assignation de valeur à une propriété typé

strict_types

strict_types ne change pas le comportement :

- Des opérateurs binaires:

```php
<?php
declare(strict_types=1);
var_dump(10 + "45"); // int(55)
var_dump(true . " hello"); // string(7) "1 hello"
```

- Des opérateurs de comparaison

```php
<?php
declare(strict_types=1);
var_dump("1" == "01"); // bool(true)
var_dump(014 == "14"); // bool(false)
var_dump(14 == "014"); // bool(true)
```

- Des constructions de langages

```php
<?php
declare(strict_types=1);
exit(true); // Écrira "1" et non pas un code de sortie 1 comme si
↪  exit(1) aurait été utilisé
```

strict_types

strict_types ne change pas le comportement :

- De la coercitions des types scalaires pour les fonctions appelées par le moteur

```php
<?php
declare(strict_types=1);
$f = fn (int $i): bool => (bool) ($i % 2);
$a = ['1', '2', 3, 4, '5.0', '6.0'];
var_dump(array_filter($a, $f));
```

Résultat :

```
array(3) {
  [0]=>
  string(1) "1"
  [2]=>
  int(3)
  [4]=>
  string(3) "5.0"
}
```

Exemples LSP

```php
<?php
interface I {}
class U implements I {}
class V implements I {}

class Super {
    public function foo(): I {}
}
class Sub extends Super {
    public function foo(): U|V {}
}
```

```php
<?php
interface A {}
interface B {}
class X implements A, B {}
class Y implements A, B {}

class Super {
    public function foo(): A&B {}
}
class Sub extends Super {
    public function foo(): X|Y {}
}
```

References

[1] George Peter Banyard. *Add true type*. Apr. 7, 2022. URL: *https://wiki.php.net/rfc/true-type* (visited on 10/10/2022).

[2] George Peter Banyard. *Allow null and false as stand-alone types*. Feb. 20, 2022. URL: *https://wiki.php.net/rfc/null-false-standalone-types* (visited on 10/10/2022).

[3] George Peter Banyard. *Disjunctive Normal Form Types*. Apr. 7, 2022. URL: *https://wiki.php.net/rfc/dnf_types* (visited on 10/10/2022).

References ii

[4] George Peter Banyard. *Pure intersection types*. Mar. 23, 2021. URL: *https://wiki.php.net/rfc/pure-intersection-types* (visited on 10/10/2022).

[5] George Peter Banyard. *Saner numeric strings*. June 28, 2020. URL: *https://wiki.php.net/rfc/saner-numeric-strings* (visited on 10/10/2022).

[6] *Forme normale disjonctive*. In: *Wikipédia*. Page Version ID: 193799539. May 18, 2022. URL: *https://fr.wikipedia.org/w/index.php?title=Forme_normale_disjonctive&oldid=193799539* (visited on 10/10/2022).

References iii

[7] Barbara H. Liskov and Jeannette M. Wing. “A behavioral notion of subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1, 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: *10.1145/197320.197383*. URL: *https://doi.org/10.1145/197320.197383* (visited on 10/10/2022).

[8] Mark Seemann. *The Liskov Substitution Principle as a profunctor*. ploeh blog danish software design. Dec. 6, 2021. URL: *https://blog.ploeh.dk/2021/12/06/the-liskov-substitution-principle-as-a-profunctor/* (visited on 10/10/2022).

References iv

[9] *Type system*. In: *Wikipedia*. Page Version ID: 1114430701. Oct. 6, 2022. URL: *https://en.wikipedia.org/w/index.php?title=Type_system&oldid=1114430701* (visited on 10/10/2022).