制約の力 - 状態を限定する -

PHP カンファレンス福岡 2023

@shin1x1

@shin1x1

- 新原雅司
- 大阪で、Web システムの開発や開発支援などを行なっています。
- PHP の現場
    - https://php-genba.shin1x1.com/

制約でコードに秩序を

@shin1x1

2016/05/21 PHPカンファレンス福岡 2016

制約で

想定された状態に限定する、

想定外の状態を排除する

と言う考え方

制約が無い例

- 整数に 100 を加えるだけの関数。

```php
function add100($v) {
    return $v + 100;
}

add100(10); // int(110)
```

- 型宣言が無いので、$v は全ての値を取り得る可能性がある。

- 取り得る値の例。

```
int:   10 + 100
float: 10.5 + 100;
bool:  true + 100;
bool:  false + 100;
null:  null + 100;
string: '1' + 100;
string: '10.5' + 100;
string: '1e2' + 100;
string: 'abc' + 100;
resource: STDIN + 100;
array:  [1] + 100;
object: new \stdClass() + 100;
```

- 全ての状態が起こり得る。

```
int:   10 + 100      // int(110)
float: 10.5 + 100    // float(110.5)
bool:  true + 100    // int(101)
bool:  false + 100   // int(100)
null:  null + 100    // int(100)
string: '10' + 100   // int(110)
string: '1e2' + 100  // float(200.0)
string: 'abc' + 100    // TypeError
resource: STDIN + 100  // TypeError
array:  [1] + 100      // TypeError
object: new \stdClass() + 100 // TypeError
```

制約が無い状態

- 全てが起こり得る状態。
- 起こり得る全てのことに対する考慮が必要。
  - 実装者はあり得ないと思っていても、後で読む人は想定せざるを得ない。
  - 不具合の調査や変更する場合、全ての可能性を検証する必要がある。
- 不要な複雑さを抱えることになり、それが持続してしまう。

状態を制限する

- 引数 $v の値を整数（想定された値）に限定する。
- 方法
  - 案 1: doc コメント
  - 案 2: if 文で判定
  - 案 3: 型宣言

案 1: doc コメント

```php
/**
 * @param int $v
 * @return int
 */
function add100($v) {
    return $v + 100;
}
```

- ✅ コメントなので多様な表現ができる。
  - 静的解析ツールと組み合わせるとジェネリクスや数値表現文字列など細かな型を指定できる。
- ❌ 強制力が無いので心許ない。
  - 実装（実体）と乖離する可能性。
  - 嘘の型指定やカバーしきれないアプリケーションもあるので、静的解析ツールのみでは不完全なケースがある。

案 2: if 文で判定

```php
function add100($v) {
    if (!is_int($v)) {
        throw new \InvalidArgumentException();
    }

    return $v + 100;
}
```

- ✅ 強制力がある。
- ✅ 表現力が高い（どのようなチェックも可能）。
- ⚠️ 実装を誤ったり、誤読する可能性がある。
  - テストでカバーできる。
- ❌ 判定コードを読むのに認知負荷がかかる。

案 3: 型宣言

```php
function add100(int $v): int {
    return $v + 100;
}
```

- ✅ 強制力がある。
- ✅ 記述も簡潔なので、認知負荷もかからない。
- ✅ 誤読する可能性もほぼ無い。
- ❌ 表現力に制限がある。
- 本ケースではこの方法が良い。

参考: GitHub Copilot

```php
/**
 * 整数に 10 を加算する関数
 */
function add10(int $num): int
{
    return $num + 10;
}
```

⚠️ 変数ではなく値の型である

- PHP では、変数に型があるのではなく、値に型がある。
- 引数の型宣言は、関数が呼び出された時点での型を指定しているもので、それ以降の型は変わる可能性があるので注意。

```php
function add100(int $v): int {
    // object を代入
    $v = new \stdClass();

    // TypeError
    return $v + 100;
}
```

⚠️ 演算結果が PHP_INT_MAX を超えると float になる

- int の上限を超えると float になる。

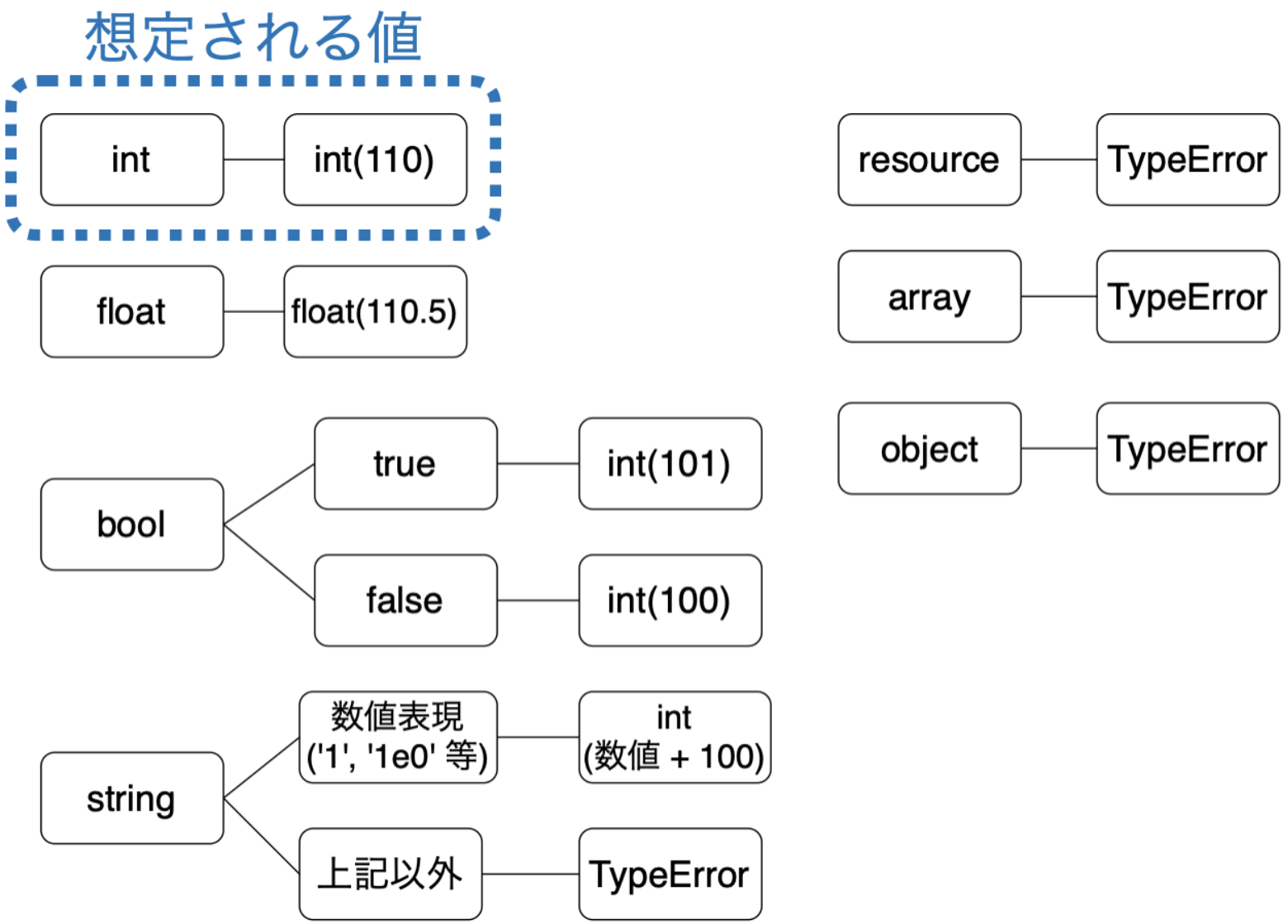
```php
function add100(int $v): int {
    return $v + 100;
}

// Uncaught TypeError: add100(): Return value must be of type int, float returned
var_dump(add100(PHP_INT_MAX - 99));
```

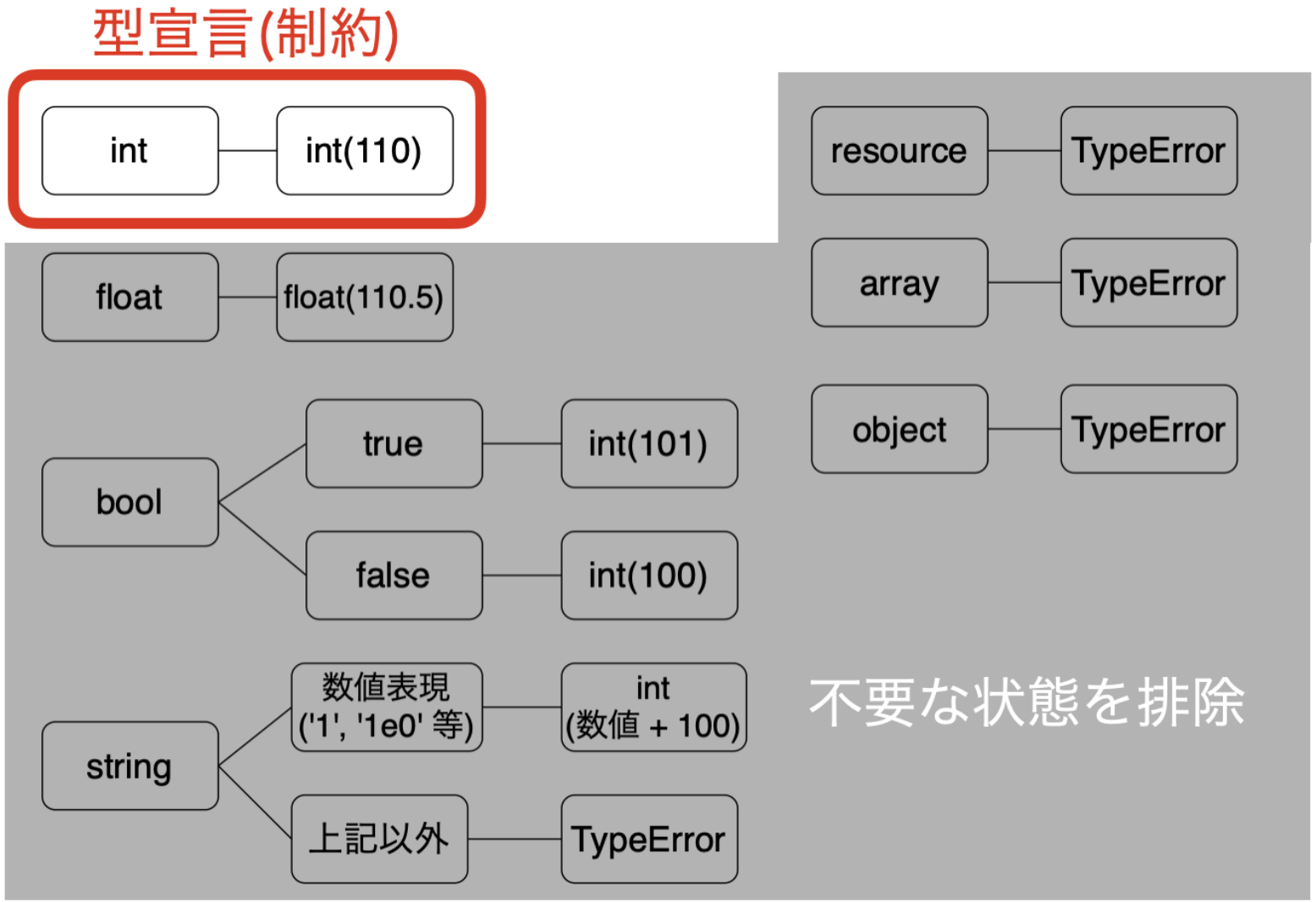
制約で状態を限定する

- その時点で想定される状態に限定し、そうでないものを排除する。
  - 不要な状態を考慮しなくて良くなる。
  - 理解容易性や変更容易性を阻害しない。
- 制約で想定された状態を保つ。
  - あるべき状態を制約という「枠」に 閉じ込めるイメージ。
  - 強制力があり、簡潔に表現できる方法が良い。

制約が無いイメージ


想定される値
int
int(110)
float
float(110.5)
bool
true
int(101)
false
int(100)
string
数値表現
('1', '1e0' 等)
int
(数値 + 100)
上記以外
TypeError
resource
TypeError
array
TypeError
object
TypeError

制約があるイメージ


型宣言(制約)
int
int(110)
float
float(110.5)
bool
true
int(101)
false
int(100)
string
数値表現
('1', '1e0' 等)
int
(数値 + 100)
上記以外
TypeError
resource
TypeError
array
TypeError
object
TypeError
不要な状態を排除

制約の活用

値の変更を排除する

- 不変（イミュータブル）にすれば、変更箇所を追う必要が無くなる。
  - 想定外の変更を避けられる。
- 方法
  - 定数 / クラス定数
  - イミュータブルプロパティ、オブジェクト

ex. readonly を使ったイミュータブルオブジェクト

- プロパティ（8.1 から）やクラス（8.2 から）の値をイミュータブルにする。
- incrementError() を実行すると、Fatal error。

```php
final readonly class ReadOnlyClass
{
    public function __construct(private int $point)
    {
    }

    public function addPoint(int $point): void
    {
        $this->point += $point; // Fatal error !!
    }
}
```

- プロパティを変更したい場合は、新しいインスタンスを返す。
- あくまで元のインスタンスのプロパティは変更されない。
- 意外に不変にできるところは多い。

```php
final readonly class ReadOnlyClass
{
    public function __construct(private int $point)
    {
    }

    public function addPoint(int $point): self
    {
        return new self($this->point + $point);
    }
}
```

⚠️ array やオブジェクトの要素は変更できる

- readonly プロパティへの再代入はできないが、すでに保持している値の要素は変更可能なので注意。

```php
final class ReadOnlyPropertyObject
{
    public function __construct(private readonly \stdClass $object)
    {
    }

    public function getObject(): \stdClass
    {
        return $this->object;
    }
}

$o = new ReadOnlyPropertyObject(new \stdClass());
$o->getObject()->i = 100;
var_dump($o->getObject()->i); // int(100)
```

値の構造を明確にする

- 万能な array だが、下記のような問題もある。
  - 要素構成が不明。
  - 要素値の型も不明。
- 要素の存在チェック、型検査（型変換）が必要になるケースもある。
- 方法
  - クラスにして、構造と要素の型を明確にする。

ex. データクラスにする

```php
final class User
{
    public function __construct(
        public readonly int $id,
        public readonly string $name,
        public readonly string $email,
    )
}

function doSomething(User $user): void {
    var_dump($user->name);
}
```

スコープを必要最小限に抑える

- スコープを小さくして、影響範囲を限定する。
- アクセス修飾子でスコープを制御。
- プロパティ
  - 基本は private。
  - データクラスでかつイミュータブルなら public で良い。
- メソッド
  - クラスの責務を実行するメソッドは public。
  - それを助けるメソッドは private。
  - public メソッドが多くあるなら、クラスを分割することを検討。

ライブラリの影響を限定

- OSS などサードパーティライブラリの更新時の影響を限定したい。
  - 影響範囲が広いと、アップデートや入れ替えが難しくなる。
- ラッパークラスで、ライブラリを内包する。
  - アプリケーションでは、このラッパークラスを利用する。
  - 直接利用している箇所は限定的なので、影響範囲を把握しやすい。
  - ラッパークラスのテストで、必要な挙動を確定できる。

ex. Chronos ラッパークラス

- Chronos をラッパークラス DateTime クラスに内包する。
- アプリケーションは、DateTime クラスを利用する。

```php
final class DateTime
{
    private Chronos $chronos;

    public function __construct(?Chronos $chronos = null)
    {
        $this->chronos = $chronos ?? Chronos::now();
    }

    public function toDateTimeString(): string
    {
        return $this->chronos->toDateTimeString();
    }

    // 必要なメソッドを実装
}
```

⚠️ ラッパークラス実装の注意点

- __call() を使った透過的なライブラリ呼び出しを避ける。
  - どのメソッドをどのように利用しているのかの把握が難しくなる。
  - 個別にメソッドを実装するのが良い。
- getter メソッドは実装しない。
  - ライブラリインスタンスをラッパークラス外に晒すと意味をなさなくなる。
- ラッパークラスは責務に応じて複数作るのもあり。
  - ex. DateTime クラスと Date クラス
  - ライブラリへ直接依存している箇所を限定できていれば良い。

RDB テーブル: 想定外の値を保存しない

- 想定外の値が含まれる場合、アプリケーション側でケアが必要となる。
  - ソースコードよりも制約の効果が大きく、持続する。
- 方法
  - 適切なデータ型を選択する。
  - テーブル制約を利用する。
    - 主キー制約、ユニーク制約
    - NOT NULL 制約
    - 外部キー制約

制約の活用例

- アプリケーションレイヤの依存関係
- テストコード（ユニットテスト、統合テスト）
    - テストコードで検証している挙動を枠にはめるイメージ
- 暗黙的型変換を排除
- リモート API スキーマ

静的解析ツールを活用

- 制約のチェックに静的解析ツールを活用する。
  - GitHub Actions などの CI 環境で自動実行しておくと良い。
- PHPStan / Psalm
  - 型の不一致、型宣言の有無、スコープチェックなど。
- php-cs-fixer
  - strict_types の強制、strict 引数の強制など。
- deptrac
  - ラッパークラス以外からのライブラリ利用をチェック。
  - アプリケーションレイヤの依存関係をチェック。

制約を入れるなら最初から

- 制約があることを前提に開発を進めることができる。
- 後で厳しくするより、緩める方が楽。
    - 制約を追加するとエラーになる可能性があるので、消極的になりがち。
- 想定された状態を知る必要があり、理解が深まる。

まとめ

- 制約という枠で、想定され状態に限定し、想定外を排除する。
- 制約は簡潔に記述でき、強制力を持つものが良い。
- システム開発は複雑なものなので、不必要な複雑さを減らして、本質的な複雑さに立ち向かおう。