「PSR 规范」PSR-1 基础编码规范

基本代码规范

本篇规范制定了代码基本元素的相关标准，以确保共享的 PHP 代码间具有较高程度的技术互通性。

关于「能愿动词」的使用

为了避免歧义，文档大量使用了「能愿动词」，对应的解释如下：

- 必须 (MUST)：绝对，严格遵循，请照做，无条件遵守；
- 一定不可 (MUST NOT)：禁令，严令禁止；
- 应该 (SHOULD) ：强烈建议这样做，但是不强求；
- 不该 (SHOULD NOT)：强烈不建议这样做，但是不强求；
- 可以 (MAY) 和 可选 (OPTIONAL) ：选择性高一点，在这个文档内，此词语使用较少；

1. 概览

- PHP 代码文件 必须 以 <?php 或 <?= 标签开始；

- PHP 代码文件 必须 以 不带 BOM 的 UTF-8 编码；

- PHP 代码中 应该 只定义类、函数、常量等声明，或其他会产生 副作用 的操作（如：生成文件输出以及修改 .ini 配置文件等），二者只能选其一；

- 命名空间以及类 必须 符合 PSR 的自动加载规范：PSR-4 中的一个；

- 类的命名 必须 遵循 StudlyCaps 大写开头的驼峰命名规范；

- 类中的常量所有字母都 必须 大写，单词间用下划线分隔；

- 方法名称 必须 符合 camelCase 式的小写开头驼峰命名规范。

2. 文件

2.1. PHP 标签

PHP 代码 **必须** 使用 <?php ?> 长标签 或 <?= ?> 短输出标签； **一定不可** 使用其它自定义标签。

2.2. 字符编码

PHP 代码 **必须** 且只可使用 不带 BOM 的 UTF-8 编码。

2.3. 副作用

一份 PHP 文件中 **应该** 要不就只定义新的声明，如类、函数或常量等不产生 副作用 的操作，要不就只书写会产生 副作用 的逻辑操作，但 **不该** 同时具有两者。

「副作用」(side effects) 一词的意思是，仅仅通过包含文件，不直接声明类、函数和常量等，而执行的逻辑操作。

「副作用」包含却不仅限于：

- 生成输出
- 直接的 *require* 或 *include*
- 连接外部服务
- 修改 *ini* 配置
- 抛出错误或异常
- 修改全局或静态变量
- 读或写文件等

以下是一个 反例，一份包含「函数声明」以及产生「副作用」的代码：

```
<?php

// 「副作用」：修改 ini 配置 ini_set('error_reporting', E_ALL);

// 「副作用」：引入文件 include "file.php";

// 「副作用」：生成输出 echo "<html>\n";

// 声明函数 function foo(){

    // 函数主体部分}
```

下面是一个范例，一份只包含声明不产生「副作用」的代码：

```
<?php

// 声明函数 function foo(){

    // 函数主体部分}

// 条件声明 **不** 属于「副作用」if (! function_exists('bar')) {

    function bar()

    {

        // 函数主体部分    }}
```

3. 命名空间和类

命名空间以及类的命名必须遵循 *PSR-4*。

根据规范，每个类都独立为一个文件，且命名空间至少有一个层次：顶级的组织名称（*vendor name*）。

类的命名 **必须** 遵循 *StudlyCaps* 大写开头的驼峰命名规范。

PHP 5.3 及以后版本的代码 **必须** 使用正式的命名空间。

例如：

```
<?php

// PHP 5.3 及以后版本的写法 namespace Vendor\Model;

class Foo{}
```

5.2.x 及之前的版本 **应该** 使用伪命名空间的写法，约定俗成使用顶级的组织名称（*vendor name*）如 *Vendor_* 为类前缀。

```
<?php

// 5.2.x 及之前版本的写法 class Vendor_Model_Foo{}
```

4. 类的常量、属性和方法

此处的「类」指代所有的类、接口以及可复用代码块（*traits*）。

4.1. 常量

类的常量中所有字母都 **必须** 大写，词间以下划线分隔。

参照以下代码：

```
<?phpnamespace Vendor\Model;

class Foo{

    const VERSION = '1.0';

    const DATE_APPROVED = '2012-06-01';}
```

4.2. 属性

类的属性命名 **可以** 遵循：

- 大写开头的驼峰式 (*$StudlyCaps*)
- 小写开头的驼峰式 (*$camelCase*)
- 下划线分隔式 (*$under_score*)

本规范不做强制要求，但无论遵循哪种命名方式，都 **应该** 在一定的范围内保持一致。这个范围可以是整个团队、整个包、整个类或整个方法。

4.3. 方法

方法名称 **必须** 符合 *camelCase()* 式的小写开头驼峰命名规范。

「PSR 规范」PSR-2 编码风格规范

编码风格指南

本篇规范是 [PSR-1][] 基本代码规范的继承与扩展。

本规范希望通过制定一系列规范化 PHP 代码的规则，以减少在浏览不同作者的代码时，因代码风格的不同而造成不便。

当多名程序员在多个项目中合作时，就需要一个共同的编码规范， 而本文中的风格规范源自于多个不同项目代码风格的共同特性， 因此，本规范的价值在于我们都遵循这个编码风格，而不是在于它本身。

1. 概览

- 代码 必须 遵循 PSR-1 中的编码规范 。

- 代码 必须 使用 4 个空格符而不是「Tab 键」进行缩进。

- 每行的字符数 应该 软性保持在 80 个之内，理论上 一定不可 多于 120 个，但 一定不可 有硬性限制。

- 每个 *namespace* 命名空间声明语句和 *use* 声明语句块后面，必须 插入一个空白行。

- 类的开始花括号（{） 必须 写在函数声明后自成一行，结束花括号（}）也 必须 写在函数主体后自成一行。

- 方法的开始花括号（{） 必须 写在函数声明后自成一行，结束花括号（}）也 必须 写在函数主体后自成一行。

- 类的属性和方法 必须 添加访问修饰符（*private*、*protected* 以及 *public*），*abstract* 以及 *final* 必须 声明在访问修饰符之前，而 *static* 必须 声明在访问修饰符之后。

- 控制结构的关键字后 必须 要有一个空格符，而调用方法或函数时则 一定不可 有。

- 控制结构的开始花括号（{） 必须 写在声明的同一行，而结束花括号（}） 必须 写在主体后自成一行。

- 控制结构的开始左括号后和结束右括号前，都 一定不可 有空格符。

1.1. 例子

以下例子程序简单地展示了以上大部分规范：

```
<?phpnamespace Vendor\Package;

use FooInterface;use BarClass as Bar;use
OtherVendor\OtherPackage\BazClass;

class Foo extends Bar implements FooInterface{

    public function sampleFunction($a, $b = null)

    {

        if ($a === $b) {

            bar();

        } elseif ($a > $b) {

            $foo->bar($arg1);

        } else {

            BazClass::bar($arg2, $arg3);

        }

    }


    final public static function bar()

    {

        // 方法的内容    }}
```

2. 通则

2.1 基本编码准则

代码 **必须** 符合 *PSR-1* 中的所有规范。

2.2 文件

所有 PHP 文件 **必须** 使用 Unix LF (linefeed) 作为行的结束符。

所有 PHP 文件 **必须** 以一个空白行作为结束。

纯 PHP 代码文件 **必须** 省略最后的 ?> 结束标签。

2.3. 行

行的长度 **一定不可** 有硬性的约束。

软性的长度约束 **必须** 要限制在 120 个字符以内，若超过此长度，带代码规范检查的编辑器 **必须** 要发出警告，不过 **一定不可** 发出错误提示。

每行 **不该** 多于 80 个字符，大于 80 字符的行 **应该** 折成多行。

非空行后 **一定不可** 有多余的空格符。

空行 **可以** 使得阅读代码更加方便以及有助于代码的分块。

每行 **一定不可** 存在多于一条语句。

2.4. 缩进

代码 **必须** 使用 4 个空格符的缩进，**一定不可** 用 tab 键。

备注：使用空格而不是「tab 键缩进」的好处在于， 避免在比较代码差异、打补丁、重阅代码以及注释时产生混淆。 并且，使用空格缩进，让对齐变得更方便。

2.5. 关键字 以及 True/False/Null

PHP 所有 关键字 **必须** 全部小写。

常量 true 、false 和 null 也 **必须** 全部小写。

3. namespace 以及 use 声明

namespace 声明后 **必须** 插入一个空白行。

所有 *use* **必须** 在 *namespace* 后声明。

每条 *use* 声明语句 **必须** 只有一个 *use* 关键词。

use 声明语句块后 **必须** 要有一个空白行。

例如：

```
<?phpnamespace Vendor\Package;

use FooClass;use BarClass as Bar;use OtherVendor\OtherPackage\BazClass;

// ... 更多的 PHP 代码在这里 ...
```

4. 类、属性和方法

此处的「类」泛指所有的「*class* 类」、「接口」以及「*traits* 可复用代码块」。

4.1. 扩展与继承

关键词 *extends* 和 *implements* **必须** 写在类名称的同一行。

类的开始花括号 **必须** 独占一行，结束花括号也 **必须** 在类主体后独占一行。

```
<?phpnamespace Vendor\Package;

use FooClass;use BarClass as Bar;use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable{

    // 这里面是常量、属性、类方法}
```

implements 的继承列表也 **可以** 分成多行，这样的话，每个继承接口名称都 **必须** 分开独立成行，包括第一个。

```
<?phpnamespace Vendor\Package;

use FooClass;use BarClass as Bar;use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements

    \ArrayAccess,
```

```
    \Countable,
    \Serializable{
    // 这里面是常量、属性、类方法}
```

4.2. 属性

每个属性都 **必须** 添加访问修饰符。

一定不可 使用关键字 *var* 声明一个属性。

每条语句 **一定不可** 定义超过一个属性。

不该 使用下划线作为前缀，来区分属性是 *protected* 或 *private*。

以下是属性声明的一个范例：

```
<?phpnamespace Vendor\Package;
class ClassName{
    public $foo = null;}
```

4.3. 方法

所有方法都 **必须** 添加访问修饰符。

不该 使用下划线作为前缀，来区分方法是 *protected* 或 *private*。

方法名称后 **一定不可** 有空格符，其开始花括号 **必须** 独占一行，结束花括号也 **必须** 在方法主体后单独成一行。参数左括号后和右括号前 **一定不可** 有空格。

一个标准的方法声明可参照以下范例，留意其括号、逗号、空格以及花括号的位置。

```
<?phpnamespace Vendor\Package;
class ClassName{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body    }}
```

4.4. 方法的参数

参数列表中，每个逗号后面 **必须** 要有一个空格，而逗号前面 **一定不可** 有空格。

有默认值的参数，**必须** 放到参数列表的末尾。

```
<?phpnamespace Vendor\Package;

class ClassName{

    public function foo($arg1, &$arg2, $arg3 = [])

    {

        // method body    }}
```

参数列表 **可以** 分列成多行，这样，包括第一个参数在内的每个参数都 **必须** 单独成行。

拆分成多行的参数列表后，结束括号以及方法开始花括号 **必须** 写在同一行，中间用一个空格分隔。

```
<?phpnamespace Vendor\Package;

class ClassName{

    public function aVeryLongMethodName(

        ClassTypeHint $arg1,

        &$arg2,

        array $arg3 = []

    ) {

        // 方法的内容    }}
```

4.5. *abstract* 、 *final* 、 以及 *static*

需要添加 *abstract* 或 *final* 声明时，**必须** 写在访问修饰符前，而 *static* 则 **必须** 写在其后。

```
<?phpnamespace Vendor\Package;
```

```
abstract class ClassName{

    protected static $foo;


    abstract protected function zim();


    final public static function bar()

    {

        // method body    }}
```

4.6. 方法及函数调用

方法及函数调用时，方法名或函数名与参数左括号之间 **一定不可** 有空格，参数右括号前也 **一定不可** 有空格。每个参数前**一定不可** 有空格，但其后 **必须** 有一个空格。

```
<?phpbar();$foo->bar($arg1);Foo::bar($arg2, $arg3);
```

参数 **可以** 分列成多行，此时包括第一个参数在内的每个参数都 **必须** 单独成行。

```
<?php$foo->bar(

    $longArgument,

    $longerArgument,

    $muchLongerArgument);
```

5. 控制结构

控制结构的基本规范如下：

- 控制结构关键词后 **必须** 有一个空格。
- 左括号 (后 **一定不可** 有空格。
- 右括号) 前也 **一定不可** 有空格。
- 右括号) 与开始花括号 { 间 **必须** 有一个空格。

- 结构体主体 **必须** 要有一次缩进。
- 结束花括号 } **必须** 在结构体主体后单独成行。

每个结构体的主体都 **必须** 被包含在成对的花括号之中， 这能让结构体更加结构话，以及减少加入新行时，出错的可能性。

5.1. *if* 、*elseif* 和 *else*

标准的 *if* 结构如下代码所示，请留意「括号」、「空格」以及「花括号」的位置， 注意 *else* 和 *elseif* 都与前面的结束花括号在同一行。

```
<?phpif ($expr1) {
    // if body} elseif ($expr2) {
    // elseif body} else {
    // else body;}
```

应该 使用关键词 *elseif* 代替所有 *else if* ，以使得所有的控制关键字都像是单独的一个词。

5.2. *switch* 和 *case*

标准的 *switch* 结构如下代码所示，留意括号、空格以及花括号的位置。*case* 语句 **必须** 相对 *switch* 进行一次缩进，而 *break* 语句以及 *case* 内的其它语句都 **必须** 相对 *case* 进行一次缩进。

如果存在非空的 *case* 直穿语句，主体里 **必须** 有类似 *// no break* 的注释。

```
<?phpswitch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
```

```
        // no break    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;}
```

5.3. while 和 do while

一个规范的 while 语句应该如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?phpwhile ($expr) {
    // structure body}
```

标准的 do while 语句如下所示，同样的，注意其「括号」、「空格」以及「花括号」的位置。

```
<?phpdo {
    // structure body;} while ($expr);
```

5.4. for

标准的 for 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?phpfor ($i = 0; $i < 10; $i++) {
    // for body}
```

5.5. foreach

标准的 foreach 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?phpforeach ($iterable as $key => $value) {
```

```
    // foreach body}
```

5.6. try, catch

标准的 try catch 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?phptry {
    // try body} catch (FirstExceptionType $e) {
    // catch body} catch (OtherExceptionType $e) {
    // catch body}
```

6. 闭包

闭包声明时，关键词 function 后以及关键词 use 的前后都 **必须** 要有一个空格。

开始花括号 **必须** 写在声明的同一行，结束花括号 **必须** 紧跟主体结束的下一行。

参数列表和变量列表的左括号后以及右括号前，**一定不可** 有空格。

参数和变量列表中，逗号前 **一定不可** 有空格，而逗号后 **必须** 要有空格。

闭包中有默认值的参数 **必须** 放到列表的后面。

标准的闭包声明语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php$closureWithArgs = function ($arg1, $arg2) {
    // body};
$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
    // body};
```

参数列表以及变量列表 **可以** 分成多行，这样，包括第一个在内的每个参数或变量都 **必须** 单独成行，而列表的右括号与闭包的开始花括号 **必须** 放在同一行。

以下几个例子，包含了参数和变量列表被分成多行的多情况。

```
<?php$longArgs_noVars = function (
```

```
    $longArgument,
    $longerArgument,
    $muchLongerArgument) {
   // body};
$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3) {
   // body};
$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3) {
   // body};
$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument) use ($var1) {
   // body};
$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
```

```
    $muchLongerVar3) {
    // body};
```

注意，闭包被直接用作函数或方法调用的参数时，以上规则仍然适用。

```
<?php$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // body    },
    $arg3);
```

总结

- 以上规范难免有疏忽，其中包括但不仅限于：

- 全局变量和常量的定义

- 函数的定义

- 操作符和赋值

- 行内对齐

- 注释和文档描述块

- 类名的前缀及后缀

「PSR 规范」PSR-3 日志接口规范

日志接口规范

本文制定了日志类库的通用接口规范。

本规范的主要目的，是为了让日志类库以简单通用的方式，通过接收一个 Psr\Log\LoggerInterface 对象，来记录日志信息。 框架以及 CMS 内容管理系统如有需要，**可以** 对此接口进行扩展，但需遵循本规范， 这才能保证在使用第三方的类库文件时，日志接口仍能正常对接。

1. 规范说明

1.1 基本规范

- LoggerInterface 接口对外定义了八个方法，分别用来记录 RFC 5424 中定义的八个等级的日志：debug、 info、 notice、 warning、 error、 critical、 alert 以及 emergency 。

- 第九个方法 —— log，其第一个参数为记录的等级。可使用一个预先定义的等级常量作为参数来调用此方法，必须与直接调用以上八个方法具有相同的效果。如果传入的等级常量参数没有预先定义，则 必须 抛出 Psr\Log\InvalidArgumentException 类型的异常。在不确定的情况下，使用者 不该 使用未支持的等级常量来调用此方法。

1.2 记录信息

- 以上每个方法都接受一个字符串类型或者是有 __toString() 方法的对象作为记录信息参数，这样，实现者就能把它当成字符串来处理，否则实现者 必须 自己把它转换成字符串。

- 记录信息参数 可以 携带占位符，实现者 可以 根据上下文将其它替换成相应的值。

- 其中占位符 必须 与上下文数组中的键名保持一致。

- 占位符的名称 必须 由一个左花括号 { 以及一个右括号 } 包含。但花括号与名称之间 一定不可有空格符。

- 占位符的名称 应该 只由 A-Z、a-z、0-9、下划线 _、以及英文的句号 . 组成，其它字符作为将来占位符规范的保留。

- 实现者 可以 通过对占位符采用不同的转义和转换策略，来生成最终的日志。 而使用者在不知道上下文的前提下，不该 提前转义占位符。

- 以下是一个占位符使用的例子：

```
/**
* 用上下文信息替换记录信息中的占位符
*/function interpolate($message, array $context = array()){
  // 构建一个花括号包含的键名的替换数组  $replace = array();
  foreach ($context as $key => $val) {
      $replace['{' . $key . '}'] = $val;
  }
```

```
  // 替换记录信息中的占位符，最后返回修改后的记录信息。   return strtr($message, $replace);}

// 含有带花括号占位符的记录信息。$message = "User {username} created";

// 带有替换信息的上下文数组，键名为占位符名称，键值为替换值。$context = array('username' => 'bolivar');

// 输出 "Username bolivar created"echo interpolate($message, $context);
```

1.3 上下文

- 每个记录函数都接受一个上下文数组参数，用来装载字符串类型无法表示的信息。它 可以 装载任何信息，所以实现者 必须 确保能正确处理其装载的信息，对于其装载的数据， 一定不可 抛出异常，或产生 PHP 出错、警告或提醒信息（error、warning、notice）。

- 如需通过上下文参数传入了一个 Exception 对象，必须 以 exception 作为键名。记录异常信息是很普遍的，所以如果它能够在记录类库的底层实现，就能够让实现者从异常信息中抽丝剥茧。 当然，实现者在使用它时，必须 确保键名为 exception 的键值是否真的是一个 Exception，毕竟它 可以 装载任何信息。

1.4 助手类和接口

- Psr\Log\AbstractLogger 类使得只需继承它和实现其中的 log 方法，就能够很轻易地实现 LoggerInterface 接口，而另外八个方法就能够把记录信息和上下文信息传给它。

- 同样地，使用 Psr\Log\LoggerTrait 也只需实现其中的 log 方法。不过，需要特别注意的是，在 traits 可复用代码块还不能实现接口前，还需要 implement LoggerInterface。

- 在没有可用的日志记录器时，Psr\Log\NullLogger 接口 可以 为使用者提供一个备用的日志「黑洞」。不过，当上下文的构建非常消耗资源时，带条件检查的日志记录或许是更好的办法。

- Psr\Log\LoggerAwareInterface 接口仅包括一个 setLogger(LoggerInterface $logger) 方法，框架可以使用它实现自动连接任意的日志记录实例。

- Psr\Log\LoggerAwareTrait trait 可复用代码块可以在任何的类里面使用，只需通过它提供的 $this->logger，就可以轻松地实现等同的接口。

- Psr\Log\LogLevel 类装载了八个记录等级常量。

2. 包

上述的接口、类和相关的异常类，以及一系列的实现检测文件，都包含在 psr/log 文件包中。

3. Psr\Log\LoggerInterface

```
<?php
```

```php
namespace Psr\Log;

/**

 * 日志记录实例

 *

 * 日志信息变量 —— message，**必须** 是一个字符串或是实现了 __toString() 方法的对象。

 *

 * 日志信息变量中 **可以** 包含格式如 “{foo}” (代表 foo) 的占位符，

 * 它将会由上下文数组中键名为「foo」的键值替代。

 *

 * 上下文数组可以携带任意的数据，唯一的限制是，当它携带的是一个 exception 对象时，它的键名 **必须** 是 "exception"。

 *

 * 详情可参阅：https://github.com/PizzaLiu/PHP-FIG/blob/master/PSR-3-logger-interface-cn.md

 */interface LoggerInterface{

    /**

     * 系统不可用

     *

     * @param string $message

     * @param array $context

     * @return null

     */

    public function emergency($message, array $context = array());



    /**
```

```
     *  **必须** 立刻采取行动
     *
     * 例如：在整个网站都垮掉了、数据库不可用了或者其他的情况下， **应该** 发送一条警报短信把你叫醒。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function alert($message, array $context = array());

    /**
     * 紧急情况
     *
     * 例如：程序组件不可用或者出现非预期的异常。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function critical($message, array $context = array());

    /**
     * 运行时出现的错误，不需要立刻采取行动，但必须记录下来以备检测。
     *
     * @param string $message
```

```
     * @param array $context
     * @return null
     */
    public function error($message, array $context = array());

    /**
     * 出现非错误性的异常。
     *
     * 例如：使用了被弃用的 API、错误地使用了 API 或者非预想的不必要错误。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function warning($message, array $context = array());

    /**
     * 一般性重要的事件。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function notice($message, array $context = array());
```

```php
    /**
     * 重要事件
     *
     * 例如：用户登录和 SQL 记录。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function info($message, array $context = array());

    /**
     * debug 详情
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function debug($message, array $context = array());

    /**
     * 任意等级的日志记录
     *
     * @param mixed $level
     * @param string $message
```

```
     * @param array $context
     * @return null
     */
    public function log($level, $message, array $context = array());}
```

4. Psr\Log\LoggerAwareInterface

```
<?php
namespace Psr\Log;
/**
 * logger-aware 定义实例
 */interface LoggerAwareInterface{
    /**
     * 设置一个日志记录实例
     *
     * @param LoggerInterface $logger
     * @return null
     */
    public function setLogger(LoggerInterface $logger);}
```

5. Psr\Log\LogLevel

```
<?php
```

```php
namespace Psr\Log;

/**

 * 日志等级常量定义

 */class LogLevel{

    const EMERGENCY = 'emergency';

    const ALERT     = 'alert';

    const CRITICAL  = 'critical';

    const ERROR     = 'error';

    const WARNING   = 'warning';

    const NOTICE    = 'notice';

    const INFO      = 'info';

    const DEBUG     = 'debug';}
```

「PSR 规范」PSR-4 自动加载规范

1. 概述

本 PSR 是关于由文件路径 自动载入 对应类的相关规范， 本规范是可互操作的，可以作为任一自动载入规范的补充，其中包括 PSR-0，此外， 本 PSR 还包括自动载入的类对应的文件存放路径规范。

2. 详细说明

1. 此处的「类」泛指所有的「Class 类」、「接口」、「traits 可复用代码块」以及其它类似结构。

2. 一个完整的类名需具有以下结构:

```
\<命名空间>(\<子命名空间>)*\<类名>
```

   1) 完整的类名 必须 要有一个顶级命名空间，被称为 "vendor namespace"；

   2) 完整的类名 可以 有一个或多个子命名空间；

   3) 完整的类名 必须 有一个最终的类名；

   4) 完整的类名中任意一部分中的下滑线都是没有特殊含义的；

   5) 完整的类名 **可以** 由任意大小写字母组成；

   6) 所有类名都 **必须** 是大小写敏感的。

3. 当根据完整的类名载入相应的文件

1) 完整的类名中，去掉最前面的命名空间分隔符，前面连续的一个或多个命名空间和子命名空间，作为「命名空间前缀」，其必须与至少一个「文件基目录」相对应；

2) 紧接命名空间前缀后的子命名空间 必须 与相应的「文件基目录」相匹配，其中的命名空间分隔符将作为目录分隔符。

3) 末尾的类名 必须 与对应的以 *.php* 为后缀的文件同名。

4) 自动加载器（*autoloader*）的实现 一定不可 抛出异常、一定不可 触发任一级别的错误信息以及 不应该 有返回值。

3. 例子

下表展示了符合规范完整类名、命名空间前缀和文件基目录所对应的文件路径。

完整类名	命名空间前缀	文件基目录	文件路径
\Acme\Log\Writer\File_Writer	Acme\Log\Writer	./acme-log-writer/lib/	./acme-log-writer/lib/File_Writer.php
\Aura\Web\Response\Status	Aura\Web	/path/to/aura-web/src/	/path/to/aura-web/src/Response/Status.php
\Symfony\Core\Request	Symfony\Core	./vendor/Symfony/Core/	./vendor/Symfony/Core/Request.php
\Zend\Acl	Zend	/usr/includes/Zend/	/usr/includes/Zend/Acl.php

「PSR 规范」PSR-6 缓存接口规范

介绍

缓存是提升应用性能的常用手段，为框架中最通用的功能，每个框架也都推出专属的、功能多 样的缓存库。这些差别使得开发人员不得不学习多种系统，而很多可能是他们并不需要的功能。 此外，缓存库的开发者同样面临着一个窘境，是只支持有限数量的几个框架还是创建一堆庞 大的适配器类。

一个通用的缓存系统接口可以解决掉这些问题。库和框架的开发人员能够知道缓存系统会按照他们所 预期的方式工作，缓存系统的开发人员只需要实现单一的接口，而不用去开发各种各样的适配器。

目标

本 PSR 的目标是：创建一套通用的接口规范，能够让开发人员整合到现有框架和系统，而不需要去 开发框架专属的适配器类。

定义

- 调用类库 (Calling Library) - 调用者，使用缓存服务的类库，这个类库调用缓存服务，调用的 是此缓存接口规范的具体「实现类库」，调用者不需要知道任何「缓存服务」的具体实现。

- 实现类库 (Implementing Library) - 此类库是对「缓存接口规范」的具体实现，封装起来的缓存服务，供「调用类库」使用。实现类库 必须 提供 PHP 类来实现 Cache\CacheItemPoolInterface 和 Cache\CacheItemInterface 接口。实现类库 必须 支持最小的如下描述的 TTL 功能，秒级别的精准度。

- 生存时间值 (TTL - Time To Live) - 定义了缓存可以存活的时间，以秒为单位的整数值。

- 过期时间 (Expiration) - 定义准确的过期时间点，一般为缓存存储发生的时间点加上 TTL 时 间值，也可以指定一个 DateTime 对象。

- 假如一个缓存项的 TTL 设置为 300 秒，保存于 1:30:00 ，那么缓存项的过期时间为 1:35:00。

- 实现类库 可以 让缓存项提前过期，但是 必须 在到达过期时间时立即把缓存项标示为 过期。如果调用类库在保存一个缓存项的时候未设置「过期时间」、或者设置了 null 作为过期 时间（或者 TTL 设置为 null），实现类库 可以使用默认自行配置的一个时间。如果没 有默认时间，实现类库 必须把存储时间当做 永久性 存储，或者按照底层驱动能支持的 最长时间作为保持时间。

- 键 (KEY) - 长度大于 1 的字串，用作缓存项在缓存系统里的唯一标识符。实现类库 必须 支持「键」规则 A-Z, a-z,0-9, _, 和 . 任何顺序的 UTF-8 编码，长度 小于 64 位。实现类库 可以 支持更多的编码或者更长的长度，不过 必须 支持至少以上指定 的编码和长度。实现类库可自行实现对「键」的转义，但是 必须 保证能够无损的返回「键」字串。以下 的字串作为系统保留: {}()/\@:，一定不可 作为「键」的命名支持。

- 命中 (*Hit*) - 一个缓存的命中，指的是当调用类库使用「键」在请求一个缓存项的时候，在缓存 池里能找到对应的缓存项，并且此缓存项还未过期，并且此数据不会因为任何原因出现错误。调用类 库 应该 确保先验证下 *isHit()* 有命中后才调用 *get()* 获取数据。

- 未命中 (*Miss*) - 一个缓存未命中，是完全的上面描述的「命中」的相反。指的是当调用类库使用「键」在请求一个缓存项的时候，在缓存池里未能找到对应的缓存项，或者此缓存项已经过期，或者此数据因为任何原因出现错误。一个过期的缓存项，必须 被当做 未命中 来对待。

- 延迟 (*Deferred*) - 一个延迟的缓存，指的是这个缓存项可能不会立刻被存储到物理缓存池里。一个 缓存池对象 可以对一个指定延迟的缓存项进行延迟存储，这样做的好处是可以利用一些缓存服务器提供 的批量插入功能。缓存池 必须 能对所有延迟缓存最终能持久化，并且不会丢失。可以 在调用类库还未发起保存请求之前就做持久化。当调用类库调用 *commit()* 方法时，所有的延迟缓存都 必须 做持久化。实现类库 可以 自行决定使用什么逻辑来触发数据持久化，如对象的 析构方法 (*destructor*) 内、调用 *save()* 时持久化、倒计时保存或者触及最大数量时保存等。当请求一个延迟 缓存项时，必须 返回一个延迟，未持久化的缓存项对象。

数据

实现类库 **必须** 支持所有的可序列化的 *PHP* 数据类型，包含：

- **字符串** - 任何大小的 *PHP* 兼容字符串
- **整数** - *PHP* 支持的低于 *64* 位的有符号整数值

- **浮点数** - 所有的有符号浮点数
- **布尔** - *true* 和 *false*.
- **Null** - *null* 值
- **数组** - 各种形式的 *PHP* 数组
- **对象（*Object*）** - 所有的支持无损序列化和反序列化的对象，如：*$o == unserialize(serialize($o))* 。对象 **可以** 使用 *PHP* 的 *Serializable* 接口，*__sleep()* 或者 *__wakeup()* 魔术方法，或者在合适的情况下，使用其他类似的语言特性。

所有存进实现类库的数据，都 必须 能做到原封不动的取出。连类型也 必须 是完全一致，如果 存进缓存的是字符串 *5*，取出来的却是整数值 *5* 的话，可以算作严重的错误。实现类库 **可以** 使用 *PHP* 的「*serialize()/unserialize()* 方法」作为底层实现，不过不强迫这样做。对于他们的兼容性，以能支持所有数据类型作为基准线。

实在无法「完整取出」存入的数据的话，实现类库 **必须** 把「缓存丢失」标示作为返回，而不是损坏了的数据。

主要概念

缓存池 *Pool*

缓存池包含缓存系统里所有缓存数据的集合。缓存池逻辑上是所有缓存项存储的仓库，所有存储进去的数据， 都能从缓存池里取出来，所有的对缓存的操作，都发生在缓存池子里。

缓存项 *Items*

一条缓存项在缓存池里代表了一对「键/值」对应的数据，「键」被视为每一个缓存项主键，是缓存项的 唯一标识符，**必须** 是不可变更的，当然，「值」**可以** 任意变更。

错误处理

缓存对应用性能起着至关重要的作用，但是，无论在任何情况下，缓存 **一定不可** 作为应用程序不 可或缺的核心功能。

缓存系统里的错误 **一定不可** 导致应用程序故障，所以，实现类库 **一定不可** 抛出任何除了此接口规范定义的以外的异常，并且 **必须** 捕捉包括底层存储驱动抛出的异常，不让其冒泡至超 出缓存系统内。

实现类库 **应该** 对此类错误进行记录，或者以任何形式通知管理员。

调用类库发起删除缓存项的请求，或者清空整个缓冲池子的请求，「键」不存在的话 **必须** 不能 当成是有错误发生。后置条件是一样的，如果取数据时，「键」不存在的话 **必须** 不能当成是有错误发生

接口

CacheItemInterface

CacheItemInterface 定义了缓存系统里的一个缓存项。每一个缓存项 **必须** 有一个「键」与之相 关联，此「键」通常是通过 Cache\CacheItemPoolInterface 来设置。

Cache\CacheItemInterface 对象把缓存项的存储进行了封装，每一个 Cache\CacheItemInterface 由一个 Cache\CacheItemPoolInterface 对象生成，CacheItemPoolInterface 负责一些必须的设置，并且给对象设置具有 唯一性的「键」。

Cache\CacheItemInterface 对象 **必须** 能够存储和取出任何类型的，在「数据」章节定义的 PHP 数值。

调用类库 **一定不可** 擅自初始化「CacheItemInterface」对象，「缓存项」只能使用「CacheItemPoolInterface」对象的 getItem() 方法来获取。调用类库 **一定不可** 假设 由一个实现类库创建的「缓存项」能被另一个实现类库完全兼容。

```
namespace Psr\Cache;

/**

 * CacheItemInterface 定了缓存系统里对缓存项操作的接口
```

```
*/interface CacheItemInterface{

    /**

     * 返回当前缓存项的「键」

     *

     * 「键」由实现类库来加载，并且高层的调用者（如：CacheItemPoolInterface）

     *  **应该** 能使用此方法来获取到「键」的信息。

     *

     * @return string

     *   当前缓存项的「键」

     */

    public function getKey();


    /**

     * 凭借此缓存项的「键」从缓存系统里面取出缓存项。

     *

     * 取出的数据 **必须** 跟使用 `set()` 存进去的数据是一模一样的。

     *

     * 如果 `isHit()` 返回 false 的话，此方法必须返回 `null`，需要注意的是
`null`

     * 本来就是一个合法的缓存数据,所以你 **应该** 使用 `isHit()` 方法来辨别到
底是

     * "返回 null 数据" 还是 "缓存里没有此数据"。

     *

     * @return mixed

     *   此缓存项的「键」对应的「值」，如果找不到的话，返回 `null`

     */
```

```
    public function get();


    /**
     * 确认缓存项的检查是否命中。
     *
     * 注意：调用此方法和调用 `get()` 时 **一定不可** 有先后顺序之分。
     *
     * @return bool
     *   如果缓冲池里有命中的话，返回 `true`，反之返回 `false`
     */
    public function isHit();


    /**
     * 为此缓存项设置「值」。
     *
     * 参数 $value 可以是所有能被 PHP 序列化的数据，序列化的逻辑
     * 需要在实现类库里书写。
     *
     * @param mixed $value
     *   将被存储的可序列化的数据。
     *
     * @return static
     *   返回当前对象。
     */
    public function set($value);
```

```
    /**
     * 设置缓存项的准确过期时间点。
     *
     * @param \DateTimeInterface $expiration
     *
     *   过期的准确时间点,过了这个时间点后,缓存项就 **必须** 被认为是过期了的。
     *   如果明确的传参 `null` 的话，**可以** 使用一个默认的时间。
     *   如果没有设置的话，缓存 **应该** 存储到底层实现的最大允许时间。
     *
     * @return static
     *   返回当前对象。
     */
    public function expiresAt($expiration);

    /**
     * 设置缓存项的过期时间。
     *
     * @param int|\DateInterval $time
     *   以秒为单位的过期时长，过了这段时间后，缓存项就 **必须** 被认为是过期了的。
     *   如果明确的传参 `null` 的话，**可以** 使用一个默认的时间。
     *   如果没有设置的话，缓存 **应该** 存储到底层实现的最大允许时间。
     *
     * @return static
```

```
     *   返回当前对象
     */
    public function expiresAfter($time);
}
```

CacheItemPoolInterface

Cache\CacheItemPoolInterface 的主要目的是从调用类库接收「键」，然后返回对应的 Cache\CacheItemInterface 对象。

此接口也是作为主要的，与整个缓存集合交互的方式。所有的配置和初始化由实现类库自行实现。

```
namespace Psr\Cache;
/**
 * CacheItemPoolInterface 生成 CacheItemInterface 对象
 */interface CacheItemPoolInterface{
    /**
     * 返回「键」对应的一个缓存项。
     *
     * 此方法 **必须** 返回一个 CacheItemInterface 对象，即使是找不到对应的缓存项
     * 也 **一定不可** 返回 `null`。
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
     * @throws InvalidArgumentException
```

```
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会被抛出。
     *
     * @return CacheItemInterface
     *   对应的缓存项。
     */
    public function getItem($key);

    /**
     * 返回一个可供遍历的缓存项集合。
     *
     * @param array $keys
     *   由一个或者多个「键」组成的数组。
     *
     * @throws InvalidArgumentException
     *   如果 $keys 里面有哪个「键」不是合法，\Psr\Cache\InvalidArgumentException 异常
     *   会被抛出。
     *
     * @return array|\Traversable
     *   返回一个可供遍历的缓存项集合，集合里每个元素的标识符由「键」组成，即使即使是找不到对
     *   的缓存项，也要返回一个「CacheItemInterface」对象到对应的「键」中。
     *   如果传参的数组为空，也需要返回一个空的可遍历的集合。
     */
    public function getItems(array $keys = array());
```

```
    /**
     * 检查缓存系统中是否有「键」对应的缓存项。
     *
     * 注意: 此方法应该调用 `CacheItemInterface::isHit()` 来做检查操作，而不是
     * `CacheItemInterface::get()`
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
     * @throws InvalidArgumentException
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会被抛出。
     *
     * @return bool
     *   如果存在「键」对应的缓存项即返回 true，否则 false
     */
    public function hasItem($key);


    /**
     * 清空缓冲池
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
```

```
    public function clear();


    /**
     * 从缓冲池里移除某个缓存项
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
     * @throws InvalidArgumentException
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会
被抛出。
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function deleteItem($key);


    /**
     * 从缓冲池里移除多个缓存项
     *
     * @param array $keys
     *   由一个或者多个「键」组成的数组。
     *
     * @throws InvalidArgumentException
     *   如果 $keys 里面有哪个「键」不是合法，
\Psr\Cache\InvalidArgumentException 异常
```

```
     *   会被抛出。
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function deleteItems(array $keys);

    /**
     * 立刻为「CacheItemInterface」对象做数据持久化。
     *
     * @param CacheItemInterface $item
     *   将要被存储的缓存项
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function save(CacheItemInterface $item);

    /**
     * 稍后为「CacheItemInterface」对象做数据持久化。
     *
     * @param CacheItemInterface $item
     *   将要被存储的缓存项
     *
     * @return bool
```

```
     *   成功返回 true，有错误发生返回 false
     */
    public function saveDeferred(CacheItemInterface $item);


    /**
     * 提交所有的正在队列里等待的请求到数据持久层，配合 `saveDeferred()` 使用
     *
     * @return bool
     *  成功返回 true，有错误发生返回 false
     */
    public function commit();}
```

CacheException

此异常用于缓存系统发生的所有严重错误，包括但不限制于 缓存系统配置，如连接到缓存服务器出错、错 误的用户身份认证等。

所有的实现类库抛出的异常都 **必须** 实现此接口。

```
namespace Psr\Cache;

/**
 * 被所有的实现类库抛出的异常继承的「异常接口」
 */interface CacheException{}
```

InvalidArgumentException

```
namespace Psr\Cache;

/**
 * 传参错误抛出的异常接口
 *
```

```
* 当一个错误或者非法的传参发生时，**必须** 抛出一个继承了

* Psr\Cache\InvalidArgumentException 的异常

*/interface InvalidArgumentException extends CacheException{}
```