ThinkPHP Framework 1.5
Programming standard

ThinkPHP 1.5
编码规范

编写：ThinkPHP 文档组

最后更新：2008-12-16

目录

1 概述

1.1 适用范围

本文档提供的代码格式和文档的指南是给参与 ThinkPHP 的个人和团队使用的，对许多 ThinkPHP 的项目开发者也有帮助，可以和 ThinkPHP 的代码保持一致。 注：有时候开发者认为在最详细的设计级别上标准的建立比标准所建议的更重要。

ThinkPHP 编码标准的话题包括：

文件格式

命名约定

编码风格

注释文档

格式规范

开发规划

1.2 目标

本编码规范的形成旨在为 ThinkPHP 开发框架和以后在此之上的应用开发建立一个可操作的编程标准、约定和指南，以规范我们的代码开发工作。提高代码的可读性，提高系统的健壮性、稳定性、可靠性。通过遵循这些程序设计标准，作为一个 PHP 开发者的生产效率会有显著提高。经验证明，若从一开始就花时间编写高质量的代码，则在软件开发阶段，对代码的修改要容易很多。

最后，遵循一套通用的程序设计标准将带来更大的一致性，使软件开发团队的效率明显提高。

1.3 原则

本编码规范遵循几个主要原则：

- ✧ 可维护性；
- ✧ 文档化；
- ✧ 提高效率；

1.4 参考

在编写本编码规范的时候，参考了

PHP 编码规范 [第一版]
PHPDocument 使用说明规范
Java 编码规范
PHP 编程标准 2000-11-16 Fredrik Kristiansen

2 约定规则

2.1 给代码注释

记住：如果你的代码不值得注释，那么它就不值得保留。当正确地使用了本文提到的注释标准和方针，就可以大幅度地提高代码质量。

2.2 让代码分段和缩进

一种提高代码可读性的方法是给代码分段 ,换句话说 ,就是在代码块内让代码缩进。所有在括号 { 和 } 之内的代码，构成一个块。基本思想是，块内的代码都应统一地缩进去一个单位。缩进由四个空格组成，禁止使用制表符 TAB 。

一行 80 字符以内是比较合适，在有些情况下，长点也可以, 但最多为 120 个字符。

2.3 行结束标志

行结束标志遵循 Unix 文本文件的约定 ,行必需以单个换行符(LF)结束。换行符在文件中表示为 10，或 16 进制的 0x0A。

注：不要使用 苹果操作系统的回车（0x0D）或 Windows 电脑的回车换行组合如（0x0D,0x0A）。

2.4 在代码中使用空白

在代码中加入几个空行，也叫空白，将代码分为一些小的、容易理解的部分，可以使它更加可读。建议采用一个空行来分隔代码的逻辑组，例如控制结构，采用两个空行来分隔成员函数定义。没有空白的代码很难读，很难理解。

2.5 遵循 30 秒法则

其他的程序员应能在少于 30 秒钟的时间内完全理解你的成员函数，理解它做什么，为什么这样做，它是怎样做的。如果他们做不到，说明你的代码太难维护，应加以改进。30 秒钟，明明白白。 一个好的经验法则是：如果一个成员函数一个屏幕装不下，那么它就很可能太长了。

2.6 每行只做一件事情

每一行代码只做一件事情。在依赖于穿孔卡片的计算机发展的早期，想让一行代码完成尽量多的功能的想法是可以理解的。若想在一行里做多件事情，就会使代码难于理解。为什么要这样呢？我们应使代码尽量容易理解，从而更容易维护和改进。正如同一个成员函数应该并且只能做一件事一样，一行代码也只应做一件事情。

此外，应让代码在一个屏幕内可见。 也不应向右滚动编辑窗口来读取一整行代码，包括含有行内注释语句的代码。

2.7 说明运行顺序

提高代码可读性的一个相当简单的方法是使用圆括号来说明 PHP 代码运行的准确顺序。如果为了理解你的源码而必须了解编程语言的操作顺序，那么这说明源码中一定有什么重要的东西做的不对。 这大多是在 AND 或者 OR 其它几个比较关系处产生的逻辑比较上的问题。

注意：如果你象前文所建议的那样，采用短小单独的命令行，那么就不会产生这个问题。

2.8 公共和保护接口最小化原则

面向对象程序设计的基本点之一是最小化一个类的公共接口。这样做有几个理由：

- ✧ 可学习性。要了解如何使用一个类，只需了解它的公共接口即可。公共接口越小，类越容易学习。
- ✧ 减少耦合。当一个类的实例向另一个类的实例或者直接向这个类发送一条消息时，这两个类变得耦合起来。最小化公共接口意味着将耦合的可能降到最低。
- ✧ 更大的灵活性。这直接与耦合相联系。一旦想改变一个公共接口的成员函数的实现方法，如你可能想修改成员函数的返回值，那么你很可能不得不修改所有调用了该成员函数的代码。公共接口越小，封装性就越大，代码的灵活性也越大。

- ✧ 尽力使公共接口最小化这一点明显地很值得你的努力，但通常不明显的是也应使被保护接口最小化。基本思想是，从一个子类的角度来看，它所有超类的被保护接口是公共的。任何在被保护接口内的成员函数可被一个子类调用。所以，出于与最小化公共接口同样的理由，应最小化类的被保护接口。
- ✧ 首先定义公共接口。大多数有经验的开发者在开始编写类的代码之前就先定义类的公共接口。第一，如果你不知道一个类要完成怎样的服务/行为，你仍有一些设计工作要做。第二，这样做使这个类很快地初具雏形，以便其他有赖于该类的开发者在"真正的"类被开发出来以前至少可以用这个雏形开始工作。 第三，这种方法给你提供了一个初始框架，围绕着这个框架你构造类。

3 命名规范

3.1.1 合适的命名

命名是程序规划的核心。古人相信只要知道一个人真正的名字就会获得凌驾于那个人之上的不可思议的力量。只要你给事物想到正确的名字，就会给你以及后来的人带来比代码更强的力量。总的来说，只有了解系统的程序员才能为系统取出最合适的名字。如果所有的命名都与其自然相适合，则关系清晰，含义可以推导得出，一般人的推想也能在意料之中。如果你发觉你的命名只有少量能和其对应事物相匹配的话， 最好还是重新好好再看看你的设计吧。

3.1.2 文件后缀

为了更好的识别文件类型和功能，对 ThinkPHP 框架中使用的 php 文件后缀名定义如下：

- ✧ 类和接口文件的后缀统一使用 .class.php
- ✧ 其他文件的后缀统一用 .php

3.1.3 目录和文件命名

目录名和文件名命名应注意：

- ✧ 采用大小写结合的方式，并且首字母也大写；
- ✧ 一般不建议使用下划线‘_’，除非特殊需要；
- ✧ 不允许使用点‘.’，防止 import 错误；
- ✧ 不超过 20 个英文字符，不使用中文；
- ✧ 如果文件是一个类或者接口，那么应该和类名保持一致；

3.1.4 包的命名

包在 ThinkPHP 中意义比较特殊，包的名称是由目录或者目录加文件名的方式组成，所以其相应的命名遵循目录和文件名的命名规范。每个包名称之间用点号‘.’分隔开来。如 Think.Core.Action Think.Util.ArrayList 等。

全局包的名字用所在机构的 Internet 保留域名开头。如：Com.Sina，Org.Util 等等；

(注意： 包的命名是需要和目录以及文件相对应的)

项目中包的命名方式是（也就代表了相应的目录结构）

域名＋项目名＋模块名＋类型＋类名

例如

Com.Liu21st.Lab.Info.Action.UserAction

Com.Liu21st.Lab.Info.Common. Filter

3.1.5 类和接口命名

- ✧ 类和接口的命名规范同文件名的命名，并且类和接口的命名应该和该文件名一致（包括大小写）

- ✧ 尽量采用该领域的术语来命名类和接口。如果用户称他们的"客户"为"顾客" ，那么就采用术语 Customer 来命名这个类，而不用 Client。许多程序开发者会犯的一个错误是，不去使用工业或领域里已经存在着很完美的术语时，却生造出一些普通词汇。
- ✧ 避免使用长名字（最好不超过 15 个字母）。 虽然 PhysicalOrVirtualProductOrService 看起来似乎是个不错的类名，但是这个名字太长了，应该考虑重新给它起个短一点的名字，比如象 Offering。
- ✧ 在为类命名前首先要知道它是什么。如果通过类名的提供的线索，你还是想不起这个类是什么的话，那么你的设计就还做的不够好。
- ✧ 超过三个词组成的混合名是容易造成系统各个实体间的混淆，再看看你的设计，尝试使用（CRC Se-ssion card)看看该命名所对应的实体是否有着那么多的功用。
- ✧ 对于派生类的命名应该避免带其父类名的诱惑，一个类的名字只与它自身有关，和它的父类叫什么无关。
- ✧ 有时后缀名是有用的，例如：如果你的系统使用了代理（agent ），那么就把某个部件命名为"下载代理"（DownloadAgent）用以真正的传送信息。不要使用下划线 "_"；

3.1.6 变量和属性命名

所有变量和属性均使用完整的英文单词或缩写词方式命名，并附有中文注释。其命名原则有：

- ✧ 使用可以准确说明变量/属性的完整的英文描述符。例如，采用类似 firstName，grandTotal 或 corporateCustomer 这样的名字。虽然象 x1，y1 或 fn 这样的名字很简短，输入起来容易，但是我们难以知道它们代表什么、结果是什么含义，因而使代码以理解、维护和改进。
- ✧ 采用大小写混合，提高名字的可读性。首字母一般应该采用小写字母。
- ✧ 尽量少用缩写，但如果需要使用，可以选择单词的头 4 个字母作为缩写。如：deptInfoValue，mapActi, mapActiInst 等等。

- ✧ 避免使用相似或者仅在大小写上有区别的名字。例如，不应同时使用变量名 persistentObject 和 perSistentObject，以及 anSqlDatabase 和 anSQLDatabase。
- ✧ 避免使用下划线作为名字的首末字母。以下划线为首末字母的名字通常为系统保留，除预处理定义之外，一般不用作用户命名。更重要的是，下划线经常造成麻烦而且难输入，所以尽量避免使用。
- ✧ 下划线“_”打头的变量和属性属于私有属性
- ✧ 以双下划线“__”打头的变量和属性作为特殊变量使用时考虑（例如魔法变量）

3.1.7 类的方法命名

成员函数名称的第一个单词常常采用一个有强烈动作色彩的动词。如：openAccount()、printMailingLabel()、createUser()、delete()等。并且采用大小写结合的方式。这种约定常常使人一看到成员函数的名称就能判断它的功能。虽然这种约定要使开发者多做一些输入的工作，因为函数名常常较长，但是回报是提高代码的可理解性。

有时后缀名是有用的:

- Max - 含义为某实体所能赋予的最大值。
- Cnt - 一个运行中的计数变量的当前值。
- Key - 键值。

例如：retryMax 表示最多重试次数，retryCnt 表示当前重试次数。

- 有时前缀名是有用的：
  - is - 含义为问一个关于某样事物的问题。无论何时，当人们看到 Is 就会知道这是一个问题。
  - get - 含义为取得一个数值。

- set - 含义为设定一个数值

例如：isHitRetryLimit。

✧ 下划线“_”打头的方法属于私有方法

✧ 以双下划线“__”打头的变量和属性作为魔法方法

3.1.8 函数命名

函数命名应采用完整的英文描述符，全部使用小写，并且用下划线“_”连接。

虽然 PHP 对方法大小写没有区分，但是这样有助于帮助我们区分函数和（类）方法。

如 file_get_content import

特例

单字母函数或者方法属于 ThinkPHP 特殊方法，通常是某些操作的简化定义，而且用大写，例如 ThinkPHP 中的 ADSL 方法。

以__开头的函数或者方法属于魔术方法或者特殊方法对待，其规则可以不受函数和方法命名约束。

第三方代码的命名原则上可以不受以上规则约束。

3.1.9 常量命名

✧ 常量命名采用大写字母，用下划线“_”分割

✧ 尽量使用全称，如果需要缩写，用首 4 个字符

✧ 系统常量以 THINK_打头，例如：THINK_VERSION、THINK_PATH

✧ 类的常量以类的名称打头，例如 DB_TYPE、LOG_WRITE_METHOD

✧ 模块常量以模块名称打头，例如 INFO_TYPE USER_ID

3.1.10 局部变量命名

一般说来，命名局部变量遵循与命名变量和属性一样的约定，即使用完整的英文描述符，任何非开头的单词的第一个字母要大写。但是为方便起见，对于如下几个特殊的局部变量类型，这个约定可以放宽：

✧ 循环计数器

✧ 异常

命名循环计数器

因为局部变量常用作循环计数器，并且它为 C/C++ 所接受，所以在 Java 编程中，可以采用 i、j、k、nIndex、nLoop 作为循环计数器 [GOS96]。 若采用这些名字作为循环计数器，要始终使用它们。概括起来说，i、j、k、nIndex、nLoop 作为计数器时， 它们可以很快被输入，它们被广泛的接受。

命名异常对象

因为在 PHP 代码中异常处理也非常普遍，所以字母 e 作为**一般的异常**符被广泛地接受。如果有多个异常同时出现，应该使用该类型异常全称的缩写表示。catch(SQLException sqlexception)，一般情况下，每个应用模块应有个异常类。如 OA 系统分为公文流转、用户机构、系统权限、档案管理等模块，这时应相应地有:FlowException、OrgException、RightException、FileException 等异常类，以便系统产生异常时，能从例外快速地定位问题发生在哪个模块。

3.1.11 方法参数命名

✧ 如果是函数的参数或者非类的属性则不做特殊约定，按照变量和属性的命名规范执行；

✧ 如果是成员函数的参数，则首先使用和类的属性一致的命名；

例如

参数作为类的属性

```
function setLabel($label){
    $this->label = $label;
}
```

其它类型参数

```
function createVo($typeId,$voName){

}
```

3.1.12 数据表命名

建议使用项目名_模块名_表名 来作为数据表的完整名称，尽量使用英文全名，并且采用小写和下划线分割的规则，如果需要缩写，采用首 4 个字母，如

oa_info_user

think_common_file

3.1.13 数据表字段命名

数据表的字段名称尽量采用规范的英文名称，全部采用小写格式，并且以下划线"_"分割。

如 user_id room_type

4 注释规范

4.1.1注释约定

以 /** 开始， */ 结束的文档注释；

以 /* 开始，以 */ 结束的 C 语言风格注释；

以及以 // 开始，代码行末尾结束的单行注释。

下表是对各类注释语句建议用法的一个概括，也给出了几个例子。

注释语句类型	用法	示例
文档注释	在接口、类、成员函数和字段声明之前紧靠它们的位置用文档注释进行说明。文档注释由 *phpdoc* 处理，为一个类生成外部注释文档。	/** Customer （顾客）.顾客是指作为我们的服务及产品的销售对象的任何个人或组织。 @author 流年 */
C 语言风格注释	采用 C 语言风格的注释语句将无用的代码注释掉。保留这些代码是因为用户可能改变想法，或者只是想在调试中暂时不执行这些代码。	/* 这部分代码已被它前面的代码替代，所以于 1999 年 6 月 4 日被 B. Gustafsson 注释掉。如果两年之后仍未用这些代码，将其删除。 . . . （源代码） */

单行注释	在成员函数内部采用单行注释语句对业务逻辑、代码片段和临时变量声明进行说明。	// 因为让利活动
		// 从 1995 年 2 月开始，
		// 所以给所有超过 $1000 的
		// 发货单 5% 的折扣。

4.1.2文档注释

每个文件都应该在开头部分使用文档注释，列出其类名、作用以及版权和版本信息。

可以参考 ThinkPHP 核心类库遵循的文档注释格式。

4.1.3类和接口注释

以下的信息应写在文档注释中紧靠类的定义的前面：

- ✧ 类的目的和作用。开发者需要了解一个类的一般目的，以判断这个类是否满足他们的需求。养成一个注释与类有关的任何好东西的习惯。
- ✧ 已知的问题。如果一个类有任何突出的问题，应说明出来，让其他的开发者了解这个类的缺点/难点。此外，还应注明为什么不解决问题的原因。
- ✧ 类的开发/维护历史。通常要包含一个历史记录表，列出日期、类的作者和修改概要。这样做的目的是让进行维护的程序员了解过去曾对一个类所做的修改，是谁做了什么样的修改。
- ✧ 版权信息。

4.1.4成员函数注释

注释一个成员函数是为了函数更加可被理解，进而可维护和可扩展。

每一个 PHP 成员函数都应包含某种称之为"成员函数文档"的函数头。这些函数头在源代码的前面，用来记录所有重要的有助于理解函数的信息。这些信息包含但不仅仅局限于以下内容：

- 成员函数做什么以及它为什么做这个。通过给一个成员函数加注释，让别人更加容易判断他们是否可以复用代码。注释出函数为什么做这个可以让其他人更容易将你的代码放到程序的上下文中去。也使其他人更容易判断是否应该对你的某一段代码加以修改（有可能他要做的修改与你最初为什么要写这一段代码是相互冲突的）。
- 哪些参数必须传递给一个成员函数。还必须说明，如果带参数，那么什么样的参数必须传给成员函数，以及成员函数将怎样使用它们。这个信息使其他程序员了解应将怎样的信息传递给一个成员函数。phpdoc @param 标识便用于该目的。
- 成员函数返回什么。如果成员函数有返回值，则应注释出来，这样可以使其他程序员正确地使用返回值/对象。phpdoc @return 标识便用于此目的。
- 已知的问题。成员函数中的任何突出的问题都应说明，以便让其他程序开发者了解该成员函数的弱点和难点。如果在一个类的多个成员函数中都存在着同样的问题，那么这个问题应该写在类的说明里。
- 任何由某个成员函数抛出的异常。应说明成员函数抛出的所有异常，以便使其他程序员明白他们的代码应该捕获些什么。phpdoc @exception 标识便用于此目的。
- 如何在适当情况下调用成员函数的例子。最简单的确定一段代码如何工作的方法是看一个例子。考虑包含一到两个如何调用成员函数的例子。
- 可用的前提条件和后置条件。 前提条件是指一个成员函数可正确运行的限制条件；后置条件是指一个成员函数执行完以后的属性或声明。前提条件和后置条件以多种方式说明了在编写成员函数过程中所做的假设，精确定义了一个成员函数的应用范围。
- 成员函数的开发/维护历史。通常要包含一个历史记录表，列出日期、修改的作者和修改概要。这样做的目的是让进行维护的程序员了解过去曾对一个成员函数所做的修改，是谁做了什么样的修改。
- 除成员函数注释以外，在成员函数内部还需加上注释语句来说明你的工作。 目的是使成员函数更易

理解、维护和增强。

✧ 内部注释应采用两种方式：C 语言风格的注释（/* 和 */）和单行注释（//）。正如上述所讨论的，应认真考虑给代码的业务逻辑采用一种风格的注释，给要注释掉的无用代码采用另外一种风格的注释。对业务逻辑采用单行注释，因为它可用于整行注释和行末注释。采用 C 语言风格的注释语句去掉无用的代码，因为这样仅用一个语句就可以容易地去掉几行代码。此外，因为 C 语言风格的注释语句很象文档注释符。它们之间的用法易混淆，这样会使代码的可理解性降低。所以，应尽量减少使用它们。

在函数内，需要注释说明的地方：

✧ 控制结构。说明每个控制结构，例如比较语句和循环。你无须读完整个控制结构内的代码才判断它的功能，而仅需看看紧靠它之前的一到两行注释即可。

✧ 代码做了些什么以及为什么这样做。通常你常能看懂一段代码做了什么，但对于那些不明显的代码，你很少能判断出它为什么要那样做。例如，看完一行代码，你很容易地就可以断定它是在定单总额上打了 5% 的折扣。这很容易。不容易的是为什么要打这个折扣。显然，肯定有一条商业法则说应打折扣，那么在代码中至少应该提到那条商业法则，这样才能使其他开发者理解你的代码为什么会是这样。

✧ 局部变量。在一个成员函数内定义的每一个局部变量都应在它代码的所在行声明，并且应采用一个行内注释说明它的用法。

✧ 难或复杂的代码。若发现不能或者没有时间重写代码，那么应将成员函数中的复杂代码详细地注释出来。一般性的经验法则是，如果代码并非显而易见的，则应说明。

✧ 处理顺序。如果代码中有的语句必须在一个特定的顺序下执行，则应保证将这一点注释出来。没有比下面更糟糕的事了：你对一段代码做一点简单的改动，却发现它不工作，于是花了几个小时查找

问题，最后发现原来是搞错了代码的执行顺序。

✧ 在闭括号后加上注释。常常会发现你的控制结构内套了一个控制结构，而在这个控制结构内还套了一个控制结构。虽然应该尽量避免写出这样的代码，但有时你发现最好还是要这样写。问题是闭括号 } 应该属于哪一个控制结构这一点就变得混淆了。一个好消息是，有一些编辑器支持一种特性：当选用了一个开括号后，它会自动地使相应得闭括号高亮显示；一个坏消息是，并非所有的编辑器都支持这种属性。我发现通过将类似 //end if，//end for，//end switch 这样的注释加在闭括号所在行的行后，可以使代码更易理解。

4.1.5变量和属性注释

属性（字段）都应很好地加以注释，以便其他开发者理解它。要想有效地注释，以下的部分需要说明：

✧ 通过说明属性（字段）值的限制条件，有助于定义重要的业务规则，使代码更易理解（针对 PHP5）。

4.1.6局部变量注释

在 PHP 中声明和注释局部变量有几种约定。这些约定是：

一行代码只声明一个局部变量。这与一行代码应只有一个语句相一致，并使得对每个变量采用一个行内注释成为可能。

用一个行内注释语句说明局部变量。行内注释是一种紧接在同一行的命令代码后，用符号 // 标注出来的单行注释风格（它也叫"行末注释"）。应注释出一个局部变量用于做什么、在哪里适用、为什么要用等等，使代码易读。

如：

```
$count = 1;   //记录循环次数
$pageSize = 5;   //页面的大小
```

5 格式规范

5.1 排版格式

- 关键词和操作符之间加适当的空格。
- 相对独立的程序块与块之间加空行。
- 较长的语句、表达式等要分成多行书写。
- 划分出的新行要进行适应的缩进，使排版整齐，语句可读。
- 长表达式要在低优先级操作符处划分新行，操作符放在新行之首。
- 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分。
- 若函数或过程中的参数较长，则要进行适当的划分。
- 不允许把多个短语句写在一行中，即一行只写一条语句。
- 函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进风格。
- C/C++语言是用大括号'{'和'}'界定一段程序块的，编写程序块时'{'和'}'应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类或接口的定义、方法定义都应该遵循这个格式；
- if、for、do、while、switch、foreach 语句中的起始'{'紧跟其后，结束'}'保持单独一行，并且关键字对齐。
- 使用四个空格为每层次缩进（在有些编辑器中可以设置插入空格代替制表符）。
- 对与最大缩进层数，并没有一个固定的规矩，假如缩进层数大于四层的时候，你可以考虑着将代码因数分解(factoring out code)。
- 不要把小括号和关键词紧贴在一起，要用空格隔开它们。
- 不要把小括号和函数名紧贴在一起。

✧ 除非必要，不要在 Return 返回语句中使用小括号。

5.2 代码样式

5.2.1 代码划分

PHP 代码总是用完整的标准的 PHP 标签定界：

<?php

?>

禁止使用短标签

5.2.2 变量替换

使用下面方式的变量替换：

$string = "Hello $name, welcome to ThinkPHP!";

$string = "Hello {$name}, welcome to ThinkPHP!";

5.2.3 访问级别

成员变量不允许使用 var，要用 private、 protected 或 public 访问级别定义。 直接访问 public 变量是允许的但不鼓励，最好使用访问器 （set/get）。方法也需要加上类似的访问级别。

5.2.4 条件格式

总是将恒量放在等号/不等号的左边，例如：

if (6 == $errorNum) ...

理由是：

✧ 假如你在等式中漏了一个等号，语法检查器会为你报错。

✧ 你能立刻找到数值而不是在你的表达式的末端找到它。

需要一点时间来习惯这个格式，但是它确实很有用。

5.2.5 使用？：

经常开发人员往往试着在 ？ 和 ： 之间塞满了许多的代码。

以下的是一些清晰的连接规则：

- ✧ 把条件放在括号内以使它和其他的代码相分离。
- ✧ 如果可能的话，动作可以用简单的函数。
- ✧ 把所做的动作，“?”，“:”放在不同的行，除非他们可以清楚的放在同一行。

5.2.6 Continue 和 break

Continue 和 break 像 goto 一样，它们在代码中是有魔力的，所以要节俭（尽可能少）的使用它们。

使用了这一简单的魔法，由于一些未公开的原因，读者将会被定向到只有上帝才知道的地方去。

continue 和 break 混合使用是引起灾难的正确方法。

5.2.7 布尔逻辑判断

大部分函数在 FALSE 的时候返回 0，但是并不是非 0 值就代表 TRUE，因而不要用 1（TRUE，YES，诸如此类）等式检测一个布尔值，应该用 0（FALSE，NO，诸如此类）的不等式来代替：

```
if (TRUE == $data) { ...
```

应该写成：

```
if (FALSE != $data) { ...
```

在严格需要区分布尔类型的地方应该采用恒等判断，如

```
if (FALSE !== $data) { ...
```

5.3 文档化

必须用 PhpDocument 来为类生成文档。不仅因为它是标准 ,这也是被各种 php 编译器都认可的方法。

因此我们在对文档进行注释的时候，需要使用 PhpDocument 支持的标签。

下面是一下常用的 PhpDocument 标签：

@abstract 申明变量/类/方法

@access 指明这个变量、类、函数/方法的存取权限

@author 函数作者的名字和邮箱地址

@category 组织 packages

@copyright 指明版权信息

@const 指明常量

@deprecate 指明不推荐或者是废弃的信息

@example 示例

@exclude 指明当前的注释将不进行分析，不出现在文挡中

@final 指明这是一个最终的类、方法、属性，禁止派生、修改。

@global 指明在此函数中引用的全局变量

@include 指明包含的文件的信息

@link 定义在线连接

@module 定义归属的模块信息

@modulegroup 定义归属的模块组

@package 定义归属的包的信息

@param 定义函数或者方法的参数信息

@return 定义函数或者方法的返回信息

@see 定义需要参考的函数、变量，并加入相应的超级连接。

@since 指明该 api 函数或者方法是从哪个版本开始引入的

@static 指明变量、类、函数是静态的。

@throws 指明此函数可能抛出的错误异常,极其发生的情况

@todo 指明应该改进或没有实现的地方

@var 定义说明变量/属性。

@version 定义版本信息

6 开发规范

6.1 架构函数

别在对象架构期做真实的工作，在架构期初始化变量或做任何不会有失误的事情。

6.2 封装业务方法

尽量在模型类里面封装业务方法，而 Action 类只是负责业务的流程。

6.3 异常处理规范

为了系统的稳定性，应该保证代码的健壮性。时时刻刻检查是否有异常产生，并按以下的原则处

理。

- ✧ 在底层的类中，如果没有显示的界面，应该在相关的 log 日志文件中记录错误；
- ✧ 如果有显示界面，应该在记录 log 日志文件的同时，显示相关的出错信息。
- ✧ 在成员函数中应该检查参数的合法性，运行状态中参数的合法性，执行结果是否为 null 等。
- ✧ 不同模块应定义自己的异常类,按照实际运行过程可能出现的。

6.4 测试维护规范

- ✧ 单元测试要求至少达到语句覆盖。
- ✧ 单元测试开始要跟踪每一条语句，并观察数据流及变量的变化。
- ✧ 清理、整理或优化后的代码要经过审查及测试。
- ✧ 代码版本升级要经过严格测试。
- ✧ 在开发过程中多利用 ThinkPHP 的调试机制（调试模式、调试方法和日志信息）

6.5 性能约束

在写代码的时候，从头至尾都应该考虑性能问题。这不是说时间都应该浪费在优化代码上，而是我们时刻应该提醒自己要注意代码的效率。比如：如果没有时间来实现一个高效的算法，那么我们应该在文档中记录下来，以便在以后有空的时候再来实现她。

不是所有的人都同意在写代码的时候应该优化性能这个观点的，他们认为性能优化的问题应该在项目的后期再去考虑，也就是在程序的轮廓已经实现了以后。