PHP 编码规范

最后修改日期: 2010-12-13

文件状态： [ ] 草稿 [√] 正在修改 [ ] 正式发布	小组：	OPED-DM
	项目：	ALL-公共项目文档
	编写：	余沛
	审核：	ALL

目 录

介绍

标准化的重要性

编码的标准化是大多数公司为之头痛、大多数程序员为之憎恨的问题（它强制改变程序员的个人习惯）。无论是在公司内部还是在互联网社区中，无数人针对标准化中的每一条字句进行着争论。标准化存在的目的不是为了剥夺个人书写代码的自由，而是为了减少困惑，使代码阅读变成一件愉悦而不是糟糕的体验。

优点

当一个编程项目尝试着遵守公用的编码标准时，会有以下好处：

- 新人可以很快的适应环境
- 防止新人自创出一套风格并养成终生的习惯
- 防止新人一次次的犯同样的错误
- 在一致的环境下，人们可以减少犯错的机会
- 阅读他人的代码变得更加容易
- 程序员们有了一致的敌人 :-)

缺点

当然也会产生一些不好的地方:

- 如果标准中的某条与你现有的习惯不符合，那么这条标准通常看上去很蠢
- 标准降低了创造力，限制了书写代码的自由
- 标准在长期合作并完全了解的人群中是没有必要的

- 标准强迫太多的格式，这种强迫性的工作会浪费点一些时间
- 得花更多的时间呼吁其它成员遵守标准，因为大家常常忽视标准

妥协

在编码工程中，经验告诉我们这样的结论：采用标准可以使项目更加顺利地完成。编码标准在很多时候并不是一个技术问题，而是一个有关个人习惯和团队信仰的问题。所以在一个团队中，如果产生了对既定标准的质疑，要尝试接受大多数人认同的方式。这种非技术性问题通常是靠**妥协**而不是无休止的**争论**来解决的。

解释

惯例

- 在本文档中使用“要”字所指的是使用本规范的所有项目需要强制遵守的内容。
- 使用“应该”一词的作用是指导项目定制项目细节规范。因为项目必须适当的包括(include)，排除(exclude)或定制（tailor）需求。
- 使用“可以”一词的作用与“应该”类似，因为它指明了可选的需求。

实施

- 首先应该在开发小组的内部找出产生困扰的重要所在，也许一份新标准对你的状况还不够恰当，也可能还有人对其中的问题表示强烈的反对，但它已经包括了那些重要的问题。
- 无论过程存在何种争议和不满，只要最后实施足够顺利的话，程序员们最终会认同它的合理性，并认为带着一些个人保留去遵循标准还是值得的。
- **如果没有合适的检查和惩罚机制，标准将成笑话。**

认同观点

1. 我呆过很多公司,写过很多代码,经验告诉我这行不通
2. 也许可行吧，但是它既不实用又无聊

如果您带着以上成见而来看待事物的话，请您保持开放的思想。你仍可以做出标准全是废话的结论，但是做出结论的方法就是让团队中的大部份人不仅看法上都站在你一边，而且在行动上愿意为你认为更好的方式去做出改变。

命名规则

合适的命名

命名是程序规划的核心。

名字就是事物在它所处的生态环境中一个长久而深远的结果。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。

名副其实

在为任何一个变量、函数、类命名前，首先要明确的想清楚它是做什么的，然后再为它取一个直达其意的名字。**如果看到一个命名的五秒钟内，你还是想不起来它是做什么的话，那么这个命名就是糟糕的。**如果还需要查手册才能明白它的含意时，这个命名的糟糕程度就与#$%^&无异。

坏例子： $u = ...

好例子： $userName = ...

避免误导

程序员必须避免留下掩藏代码本意的错误线索。应该避免使用与本意相悖的词。即使 PHP 中没有 List 类型，使用 accounts 或 accountGroup 也比 accountList 要好。因为 List 在大

多数程序员的印象中是一个数据类型。也要注意命名之间不要太相近，要程序员去留心 serviceTreeController 或 serviceTreeWsController 之间的差别并不是一件容易的事。

做有意义的区分

- 坏的区分：$a1,$a2,$a3…..
- 废话是另一种无意义的区分。假设你有一个 Procudct 类，如果还有一个 ProductInfo 或 ProductData 类，那么它们的名称虽然不同，意思却毫无区别。Info、Data 就像 a、the 一样，是多余的废话。NameString 会比 Name 好吗？难道 Name 不是 string 而会是一个 float 型不成？
- 保持一致性，如果要命名的对象属于同一性质，就不要分成两个名字。例如某一个类里负责更新的操作叫 update，而另一个类的类似操作却叫 edit。

避免使用编码

代码已经够多，无需再自找麻烦。把类型或作用域编到名称里面，徒然增加了编码的负担。也增加了维护的成本。语言和开发工具的发展，使得类似匈牙利标记法（Hungarian Notation,HN）这样的编码形式显得多余。它们增加了修改变量、函数或类名的难度。当 phoneNumber 由一个 int 转向 string 时，还需要在代码中到处寻找，将 phoneNumber 替换成 phoneString。

类命名

- 类名应该是名词或名词短语，如 Customer、WikiPage、Account，避免使用 Manager、Processor、Data 或 Info 这样的类名。类名不应该是动词。

- 在为类（class）命名前首先要知道它是什么。如果看到一个名字五秒钟内，你还是想不起来这个类是做什么的话，那么你的设计就做的不够好。

- 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。

方法和函数命名

- 通常每个方法和函数都是执行一个动作的，所以对它们以动宾结构进行命名会更清楚的说明它们是做什么的：比如用 CheckForErrors() 代替 ErrorCheck()，用 DumpDataToFile()代替 DataFile()。这么做也可以使功能和数据成为更可区分的物体。

- 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。

缩写词不要全部使用大写字母

- 对于缩写词，应该使用首字母大写、其余字母小写的方法来书写命名。
- 当命名含有缩略词时，人们似乎有着非常不同的直觉。统一规定是最好，这样一来，命名的含义就完全可以预知了。

  举个 NetworkABCKey 的例子，注意 C 是应该是 ABC 里面的 C 还是 key 里面的 C，这个是很令人费解的。有些人不在意这些，其他人却很讨厌这样。所以你会在不同的代码里看到不同的规则，使得你不知道怎么去叫它。

  例如： 使用: GetHtmlStatistic. 不使用: GetHTMLStatistic.

类命名

- 使用大写字母作为词的分隔，其他的字母均使用小写
- 名字的首字母使用大写
- 不要使用下划线('_')

理由

- 根据很多的命名方式，大部分人认为这样是最好的方式。

例如

```
class NameOneTwo


class Name
```

方法命名

- 使用动宾结构的语法命名
- 动词首字母采用小写，名词首字母采用大写的驼峰方式书写。
- 保护和私有方法,总是以(_)下划线开头

理由

- 使用所有不同规则的大部分人发现这是最好的折衷办法。
- 动宾结构命名法总是很容易让人理解到要做什么。
- 下划线能让人很清楚这个方法的作用域在哪里。

例如

```
class NameOneTwo
{
    Public function doIt() {};
```

```
    Private function _handleError() {};

}
```

类属性命名

- 保护或私有变量，总是在变量标记$后以（_）下划线开头。
- 其余与普通变量命名规则无异。

变量命名

- 首单词小写
- 单词间隔以单词首字母大写。

理由

- PHP 的变量总是以$开始,而$符后跟小写比大写更美观。
- 其余单词首字母大写分隔，容易辨认。

定义命名 / 全局常量

- 全局常量用'_'分隔每个单词。
- 很明显。常量单词全部采用大写格式。

理由

- 这是命名全局常量的传统。你要注意不要与其它的定义相冲突。

例如

```
define("A_GLOBAL_CONSTANT", "Hello world!");
```

函数命名

- 函数命名和方法一样，首字母小写，单词间隔首字母大写，动宾结构

理由

- 既然方法都这么做了，那么就保持命名的一致性。

例如

```
function testConnect(){

}
```

PHP 文件扩展名

我见过许多种 PHP 文件的扩展名（.html, .php, .php3, .php4, .phtml, .inc, .class...）

- 所有浏览者可见页面使用.html
- 所有类、函数库文件使用.php
- 独立的配置文件使用.inc.php

理由

- 一眼就能明白这是可见的(html),可解析的(.php)，还是用户可做为配置的(.inc.php)

类规则

别在对象构造方法做实际的工作

- 别在对象的构造方法中做真实的工作，在构造方法中只初始化变量和（或）做任何不会有失误的事情。

理由

- 构造方法不能返回错误 。

例如

```
class Device {
    function __Consturct()      { /* initialize and other stuff */ }
};

$dev = new Device();
```

胖（富）类和瘦类

一个类应该有多少个方法才叫适合？正确的回答是：有刚刚好的数量。但这个回答显示没有帮助。你必须根据环境做出正确的判断。这正是程序员们所孜孜追求的。这个追求所产生的两个极端就是胖类和瘦类。瘦类追求类的最小化、拥有尽可能少的方法。瘦类希望使用者通

过派生来增加所需要的功能。

瘦类看起来很干净，但其实并非仅仅如此。你不能用瘦类来做太多的事情。它主要的目的是建立一个模范。正因为瘦类的功能如此之少，所以项目中的程序员们会创建派生类，这将导致代码的复用和维护方面的问题。这也是我们从一开始就使用对象的部份原因。最简单的解决办法就是在基类中放入一些方法，再放入一些方法，等放到足够多的时候，它就变成了胖类。.

胖类有许多的方法，你能想到的它都有。为什么它会成为一个问题呢？如果这些方法都是与类有直接关系，那么它的确不是问题。真正的问题是：人们太懒了，把很多与对象不怎么相关的方法加入进去，但是这些方法拆分到另一个类里其实会更好。所以，在给一个类添加方法是，告诉自己：再确认一次吧，这个方法真的与这个类有直接的关系吗？

胖类还有其它的问题：当一个类变胖之后，它们可能变得更难理解。调试也会变得更困难，因为交互性不可预测。当一个方法发生变化时，尽管你可能不用它或者不关注，但仍需要被测试才能发布。

短方法

- 方法代码要限制在一页内：超过 20 行的代码就会让人头痛，超过 80 行的代码则会让人恶心，超过 200 行的代码就几乎没有人会去看。
- 确保一个方法只做一件事情。不仅类的职责要单一，方法的职责也应该单一。
- 参数要尽量的少，零参数是最完美的，三个以内还可以接受，五个以上就要考虑重新设计了。从长远来说，过多的无效参数是无益于扩展和使用的。

格式化

大括号 {} 规则

- 将大括号放置在关键词下方的同列处：

```
if ($condition)          while ($condition)
{                        {
    ...                      ...
}                        }
```

- 传统的 UNIX 的括号规则是，首括号与关键词同行，尾括号与关键字同列：

```
if ($condition) {        while ($condition) {
    ...                      ...
}                        }
```

- 引起剧烈争论的非原则的问题可通过折衷的办法解决 ,两种方法任意一种都是可以接受的。不过介于大多数人的习惯，**第二种**显然更受欢迎。

缩进/制表符/空格 规则

- 使用空格符代替制表符缩进。
- 每个层次缩进四个空格符。
- 不再使用只要一有需要就缩排的方法。对与最大缩进层数，并没有一个固定的规矩，但假如缩进层数大于四层的时候，你可以考虑着将代码分解。

理由

- 不同的系统中，对于制表符的显示宽度定义可能并不一致，但空格都很一致。
- 阅读缩进层次过多的代码会使人心烦意乱。

小括号、关键词和函数 规则

- 不要把小括号和关键词紧贴在一起，要用空格隔开它们。
- 函数中的多个参数，使用空格符分隔开
- 除非必要，不要在 Return 返回语句中使用小括号。

理由

- 关键字不是函数。如果小括号紧贴着函数名和关键字，二者很容易被看成是一体的。
- 分隔参数能更清楚的分辨出它们。

例如

```
if (condition) {

}


while (condition) {

}


strcmp($s, $s1);
```

return 1;

If Else 格式

总是将恒量放在等号/不等号的左边，例如：

if (‘something’ == $errorNum) ...

一个原因是假如你在等式中漏了一个等号，语法检查器会为你报错。第二个原因是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。

---

continue,break 和 ? 的使用:

Continue 和 Break

Continue 和 break 其实是变相的隐蔽的 goto 方法。

Continue 和 break 像 goto 一样，它们在代码中有魔力，所以要**尽可能少**的使用它们。

Continue 有两个主要的问题：

- 它可以绕过测试条件。
- 它可以绕过等/不等表达式。

看看下面的例子，考虑一下问题都在哪儿发生：

while (TRUE) {

```
    ...
    // A lot of code
    ...
    if (/* some condition */) {
        continue;
    }
    ...
    // A lot of code
    ...
    if ( $i++ > STOP_VALUE) break;
}
```

注意："A lot of code"是必须的，这是为了让程序员们不能那么容易的找出错误。

通过以上的例子，我们可以得出更进一步的规则：continue 和 break 混合使用是引起灾难的正确方法。

?:

麻烦在于人民往往试着在 ？ 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。

例如

```
(condition) ? funct1() : func2();
```

or

```
(condition)
    ? long statement
    : another long statement;
```

声明块的定位

- 声明代码块需要对齐。
- 变量初始化的代码块应该列表。
- 变量标记应该以类型为排序的邻近原则，而非名字。

理由

- 清晰。

例如

```
public          $Date
public          $Name
private         $_Day
private         $_Id
```

每行一个语句

除非这些语句有很密切的联系，否则每行只写一个语句。

每行最大字符数

每行**尽量**不要超过 80 个字符，80 个以内是个很好的习惯，100-120 个字符也还能接受。

如果超过 120，那就是故意使坏了。

记录所有的空语句

总是以注释记录下 for 或者是 while 的空语句块，以便清楚的知道该段代码是漏掉了，还是故意不写的。 这种空语句块，一定要将分号换行写，而不是与 while,if 写在同一行：

```
while ($dest++ = $src++)
    ;               // VOID
```

关键词后面用空格分隔

关键词的后面总是跟随一个空格，用以和函数进行区分。例如：

```
if  (1)
;    // void
```

文档规则

语言标签

- 总是保持以 <?php 的完整标签形式开始
- 对于一个以 PHP 为末尾的文件而言，不必使用 ?> 来结尾

理由

- 短标签会使一个 php 文件和 xml 文件无法区分，因为它们都以<?开头。
- 省略掉文件末尾的 ?> ，这样不期望的空格就不会出现在文件末尾，之后仍然可以输出响应标头。在使用输出缓冲时也很便利 ,就不会看到由包含文件生成的不期望的空格。

字符串引用

- 纯字符串使用单引号界定。
- 字符串中引用变量时，变量用{}界定。

理由

- 单引号界定的字符串，减少了转义和变量侦测的步骤。
- 用{}界定变量可以避免例如”$abc”到底是$a.’bc’还是$ab.’c’的疑惑。

目录文档

所有的目录下都需要具有 README 文档，其中包括：

- 该目录的功能及其包含内容
- 对主要文件的在线说明。
- 指导新接手的人如何找到相关资源（可选）：
  - 源文件索引
  - 在线文档
  - 纸文档
  - 设计文档
- 其他对读者有帮助的东西

考虑一下，当每个原有的工程人员走了，在 6 个月之内来的一个新人，那个孤独受惊吓的探险者通过整个工程的源代码目录树 ,阅读说明文件 ,源文件的标头说明等等做为地图，他应该有能力穿越整个工程。

注释规则

注释的原则

“别给糟糕的代码加注释---重新写吧！“

注释并不天然的是件好事，事实上，注释最多也就是一种必须的“恶“。若编码本身有足够的表达力。那么就不需要注释。所以**注释的第一原则就是：没有注释。**注释总是出现在我们试图用代码表达意图却遭遇到失败时。如果你发现自己需要写注释，就停顿一分钟，想想自己有没有办法翻个身子，用代码来表达这种意图。如果你能将代码重新整理得不需要注释，那么该夸奖下自己。反之，如果还是需要写注释，你应该做个鬼脸，以愧于自己在使用代码来表达事情上的失败。

另外，注释本身一定要简单清晰，如果一段注释本身还需要解释，那就是错误的。

注释的坏处

- 注释会撒谎：注释存在的时间越久，就离其所描述的代码越远。越来越变得不靠谱。原因很简单，程序员经常会改代码，但却不能保持同步的去更新注释。不准确的注释要比没注释还要坏，它们满口胡言的指引着后来阅读代码的人前往满是陷阱的森林。

- 注释不能美化糟糕的代码：我们编写一个模块，发现它令人困扰。或者太复杂。于是我们说：“噢，写点注释吧 “，不，最好是把代码弄干净。即使将注释写得像诗歌一样优美，也并不能对代码产生半点好处。

示例：一些不应该出现的注释

- 多余的注释/废话： public $userName; // the user’s name ,如果注释本身不能比代码提供更多的信息，也没有给出作者的意图。那么就应该删除掉。
- 日志式注释： 一些程序员喜欢在代码的开头，以注释的形式写上 changeLog，这在源码控制系统出现前是可以理解的，但现在有了 CVS 或 SVN，那么这些注释就会让模块变得凌乱不堪，应该全部删除掉。
- 归属与签名： /* add by peter */,这是版本控制系统做的事，没有必要用这些小小的签名来搞脏代码。而且事实上这这些注释会放在那里一年又一年，越来越和原作者没有关系。所以，源码控制系统才是这类信息最好的归属地。
- 注释掉的代码：不要注释掉一段看起来暂时无用的代码。如果它真的没有用，那么就果断的删除掉它。因为如果你不删除它，其它人也断然不敢删除这些注释掉的代码，他们会想：“代码放在那儿，一定有其原因。而且很重要，不能删除 “。我们有源码控制系统，它们丢不掉。大胆的删除掉它。
- 信息过多 注释最好不要超过三行 如果真有那么多值得描述的事情 将细节放在文档，而将注释指向文档位置。注释里只以最简单的语言告诉人们这是个什么即可。

注释该出现在哪里

- 对意图的解释。--用来解释语言以外的故事，在需要做出业务选择的地方加上注释，解释你选择了哪条路以及为什么要做这样的选择。**注释的作用应该是说明“为什么 “，而不是“是什么 “。**后来者会非常高兴你留下的这些信息，而不是像考古者一样需要借助放大镜来解读你的每一行代码。
- 警示：用来提醒后来的人，这儿可能存在的风险。例如 strcmp 返回 0 时并不代表匹配失败。或者使用 TODO 这样的标签告诉后来人这儿还没有完成、将来会是怎样。

其它

避免魔鬼数字

源代码中那些既无注释，又未定义的赤裸裸的数字就是魔鬼数字，因为包括代码的主人在内，没有人会在三个月后还知道它们的含义。例如：

```
if      (22 == $foo) { ..... }
else if (19 == $foo) { ..... }
```

在上例中 22 和 19 的含义是什么呢？如果一个数字改变了，或者这些数字只是简单的错误，你会怎么想？

使用魔鬼数字是代码主人属于业余程序员的重要标志，这样的程序员似乎不打算在团队工作，又或者是为了维持代码而不得不做的，否则他们永远不会做这样的事。

应该用 define()、CONST、或者至少用数组的 Key=>Value 来给想表示某样东西的数值一个真正的名字，而不是采用赤裸裸的数字，例如：

```
define("PRESIDENT_WENT_CRAZY", 22);
CONST   "WE_GOOFED"  =   19;
$status = array( 'stop'   => 2, ' start'   => 1);
```

```
if      (PRESIDENT_WENT_CRAZY == $foo) { …… }
else if (self::WE_GOOFED == $foo) { ……}
else if ($status[ 'stop' ] == $foo) { .....}
```

现在不是变得更好了么？

错误返回检测规则

- 检查所有的系统调用的错误信息，除非你要忽略错误。
- 为每条系统错误消息定义好错误文本。

不要采用缺省方法测试非零值

不要采用缺省值测试非零值，也就是使用：

```
if (FAIL != f())
```

比下面的方法好：

```
if (f())
```

即使 FAIL 可以含有 0 值 ，也就是 PHP 认为 false 的表示。在某人决定用-1 代替 0 作为失败返回值的时候，

一个显式的测试就可以帮助你了。就算是比较值不会变化也应该使用显式的比较 ,例如：

```
if (!($bufsize % strlen($str)))
```

应该写成：if (0 == ($bufsize % strlen($str)))以表示测试的数值（不是布尔）型。一个经常出问题的地方就是使用 strcmp 来测试一个字符等式，结果永远也不会等于缺省值。

非零测试采用基于缺省值的做法，那么其他函数或表达式就会受到以下的限制:

- 只能返回 0 表示失败，不能为/有其他的值。
- 命名以便让一个真(true)的返回值是绝对显然的，调用函数 IsValid()而不是 Checkvalid()。

布尔逻辑类型

大部分函数在FALSE的时候返回0，但是发挥非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 == func()) { ...
```

应该写成：

```
if (FALSE != func()) { ...
```

通常避免嵌入式的赋值

有时候在某些地方我们可以看到嵌入式赋值的语句，那些结构不是一个比较好的少冗余，可读性强的方法。

```
while ($a != ($c = getchar())) {
```

```
        process the character
    }
```

++和--操作符类似于赋值语句。因此，出于许多的目的，在使用函数的时候会产生副作用。使用嵌入式赋值提高运行时性能是可能的。无论怎样，程序员在使用嵌入式赋值语句时需要考虑在增长的速度和减少的可维护性两者间加以权衡。例如：

```
a = b + c;
d = a + r;
```

不要写成：

```
d = (a = b + c) + r;
```

虽然后者可以节省一个周期。但在长远来看，随着程序的维护费用渐渐增长，程序的编写者对代码渐渐遗忘，就会减少在成熟期的最优化所得。

重用自己或其他人的艰苦工作

跨工程的重用在没有一个通用结构的情况下几乎是不可能的。对象符合他们现有的服务需求，不同的过程有着不同的服务需求环境，这使对象重用变得很困难。

开发一个通用结构需要预先花费许多的努力来设计。当努力不成功的时候，无论出于什么原因，有几种办法推荐使用：

请教！给群组发 Email 求助

这个简单的方法很少被使用。因为有些程序员们觉得如果他向其他人求助，会显得自己水平低，这多傻啊!做新的有趣的工作，不要一遍又一遍的做别人已经做过的东西。 如果你需要某些事项的源代码，如果已经有某人做过的话，就向群组发 email 求助。结果会很惊喜！

在许多大的群组中，个人往往不知道其他人在干什么。你甚至可以发现某人在找一些东西做，并且自愿为你写代码，如果人们在一起工作，外面就总有一个金矿。

告诉！当你在做事的时候，把它告诉所有人

如果你做了什么可重用的东西的话，让其他人知道。别害羞，也不要为了保护自豪感而把你的工作成果藏起来。

一旦养成共享工作成果的习惯，每个人都会获得更多。

Don't be Afraid of Small Libraries

对于代码重用，一个常见的问题就是人们不利用原有的代码组合。一个可以被重用的类也许就隐蔽在一个神秘的目录，人们不喜欢做一个小库，对小库有一些不正确感觉。把这样的感觉克服掉吧，电脑才不关心你有多少个库呢。