第一部分　高　级　PHP

第1章　开　发　思　想

命名是所有事的开始。

要真正掌握一门编程语言，不仅要理解它的语法和语义，更重要的是掌握语言所体现的哲学思想、语言产生和发展的背景以及设计特点。

1.1　PHP与我

大家是否想过，为什么会有这么多的编程语言？除了所谓“主流语言”例如　C、C++、Pascal等之外，还有其他的如Logol、Cobol、Fortran、Simula和许多更加特殊的语言。当列出一个项目的梗概时，大多数软件开发者不会真正地考虑到可以使用多种编程语言；他们都有自己偏爱的语言（也许是公司指定的一种语言），了解它的优点和它的缺点，并根据语言的具体特点修正项目。但当克服所选语言的缺陷时，就可能会增加不必要的额外工作。

了解如何使用一门语言却缺乏其特定的概念知识，就好像一个开卡车的人想参加二轮马车比赛一样，当然，一般来讲他应该懂得如何驾驶二轮马车，他甚至可能在终点线上跻身前列，但他绝不可能成为一个出色的车手，除非他熟悉新车的独特之处。

类似地，当面向对象程序设计（oop）程序员编写一个应用程序的时候，他会尽力使程序满足项目要求，处理同一个任务，不同的程序员会运用不同的方式。哪种方式更好？每一个程序员会说他（她）的方法最好，但只有那些熟悉两种概念——oop和过程化编程——的人能够作出判断。

前面提到的每一种语言代表一种解决问题的特定方法，这些问题多属于具有特殊要求的某一特殊种类。因为这些语言集中在一个有限的应用领域内，他们的成功性也限制在这些领域。像C和Pascal这样的语言变得如此流行，就是因为它们被广泛应用，并且它们不针对特殊问题，却提供了能很好地解决普遍问题的工具。

那么PHP是如何适应这一体系的呢？尽管它被称之为一种语言，但 PHP并不是一种真正独立的语言，而是许多语言的混和体。它主要用 C的句法，但与C有很大不同。它是被解释的， PHP能识别不同的变量类型，但没有严格的类型检查， PHP识别类，但没有结构体类型，类似的例子很多，但你可能已领会到了关键一点： PHP融合了许多种不同的解决问题的思想，形成了一种全新的、独一无二的方法。

为了能够用PHP成功地开发Web应用程序，我们鼓励你首先回答下述问题： PHP是我的项目所需的理想语言吗？问得好。如果我们说不，那我们就会显得很愚笨（谁会去写一本关于他们

认为不好的东西的书呢？)。让我们重新阐述这个问题，对项目来说有比 PHP更好的语言吗？这次我们可以很有把握地回答，如果你正在从事网络应用程序的开发， PHP就是为你准备的最好的语言。

1.2 计划的重要性

你为什么应该阅读这一部分

即使你是一个很熟悉PHP的职业程序员，我们也建议你阅读下面的部分，因为这里包含了成功开发的基本知识，如果你对所讨论的题目已很熟悉，也应该花时间浏览一下，你可能会发现新的信息 —— 新的题观点、新的解决方法、新的答案，你对解决未来项目的不同方面的问题了解得越多，你就能越好地抓住关键点，并且用更好的方式处理。我们希望你信任我们是职业开发者，并相信我们的经验，这将使你在以后受益。

在深入探讨PHP特定问题之前，先让我们从一个更广泛的观点开始。不论你使用什么语言，也不论你在什么平台上开发。有一些问题在应用开发中是总会涉及到的。

当从事一个专业项目的时候，考虑一下你正在做什么是至关重要的，“ 了解你的敌人，永远不要低估它 ”。尽管你的项目并不是一个真正的敌人，这句话的寓意仍然适用，在转向其他题目时，要知道项目的所有技术条件、目标平台、用户，并且决不要低估那些没有考虑周全的小问题的重要性。

据我们的经验，计划占用了50%的开发时间。项目越大，它的纲要就应该越详尽。这一原则既适用于同你的顾客相联系并与他们密切合作以确定一个总的项目概要，又适用于与你的开发者探讨确定一个编码概要。在一致性和可维护性上花的气力越少，就越容易在重新打开旧文件并设法清除错误或添加新的特征时遇到问题。

计划所用时间与项目大小并不一定成比例，例如，想一下要设计的一个搜索算法。这一应用程序只需要在一堆信息中进行基本的，搜索并能根据规则抽取数据，由于数据已经存在，所以创建和输出将不会需要太多的努力。这一应用程序将把它的大部分运行时间花在搜索循环上。这个循环也许用不了100行代码，但是为一个优化的循环选择设计一个优化的算法很容易耗费一整天的时间，这个小小的循环也许是设计阶段最庞大的部分，但另一方面，你可以在不到一天的时间内策划好数千行的代码。

同样，我们假定需要一个小脚本来列出某个目录中的所有文件，你能够很快地完成它，使其能从事某一特定任务，在一个特定的目录列出所有文件，不必再担心它了 —— 问题已解决，可以转向其他任务，把你的程序抛在脑后。但另外一种策略是考虑一下以后的某个时间，甚至可能是在一个完全不同的项目中 —— 你可能会再一次需要一种类似的工具，仅仅一遍又一遍地重做目录列举器，每一个对应一个特定的任务，这简直是在浪费时间。因此，当首次遇到这种情况时，应该考虑到这一点，应从一个目录列举器中创建一个分离的模块，允许它列举不同的目录，有选择性地递推子目录，甚至允许使用通配符，你可以创建一个“ 防弹 ”函数，它即能处理大多数特例，又能完美地应付一个目录列举器的普通要求。采用这种策略经过几个项目之后，你将拥有一个工具参数的库，可以安全地重新使用和依赖这个库，从而可以极大地减省开发时间。

当然，有了一个日益增大的免费工具函数库，依然不能满足全部需要，也不能优化这个库以适应特殊需求，有些库太庞大以致不能随处安装，因为每一次选中都必须分析几百 K字节的代码，这将严重降低站点的性能。在这种情况下，需要用 100%自己创造的优化解决方案，以取代非最优解决方案。

更大的项目如果缺乏计划将导致更多的错误，在开发后期，可能会遇到没有或无法预见的困难，这是由于缺乏计划的时间和工作，这些困难可能会严重到让你彻底地重组整个项目。例如，对一个依赖额外数据库提取层的数据库支持的应用程序，其数据库提取层仅能接收文本数据，但后来你发现也需要用它接收数值性的数据，通过工作区转换，可以使它能够接收数值性数据。但后来你又感觉到这个工作区仍旧不能满足需要，这时唯一能做的就是改变数据库接口，这需要重构提取层并对所有主代码调用进行检查，当然也需要清除先前创建的工作区。

这样，数小时甚至整天的工作将不得不耗费在本来从一开始就可以避免的问题上，这些问题往往决定了程序开发的成败，因为“ 时间是你永远都不可能充分拥有的珍贵资源 ”。下面的内容将针对大部分基本的却是非常重要的开发中的实际问题进行讨论：改善代码质量以及基本设计和文件管理的问题。陈述完这些后，我们创建一个应用程序接口（ API），采取简单的、实用的方式使你熟悉这一新的思想，然后我们从头创建一个 API，在纸上从理论上开发它，并明确一些实用规则来帮助你实施下一个 API，例如风格问题、以及商业技巧等。

1.3 编码规范

好的编码和差的编码之间究竟有何区别呢？实际上，这个问题很简单。好的代码（确实好的代码）能够像一本书一样被阅读。你能从任何地方读起，并且能够时刻意识到你所读的这些行是干什么用的，它们在什么条件下执行，它们所要求的设置。即使你缺乏背景知识，遇到了一个错综复杂的算法，你也能很快看出它所从事的任务，以及它的风格。

举个例子，然后说“ 照着做 ”总是很容易的，但我想这一章应该使你打下写专业化代码的坚固基础，这一基础将区分真正精心编制的代码和一个草草完成的程序段。抱歉的是，由于篇幅所限，我们不能按我们所希望的那样详尽地讨论良好的代码书写风格的每一方面，但本章将给你一个很好的开始。我们期望你能迅速获得专用的材料，以熟悉软件设计和工程的每一要点。编码是一个很广的领域，几乎是一门独立的科学。有许多论文论述它，虽然这些论文大多很乏味，很理论化，但在应用中是不可放弃的。下面我们就最重要的问题进行最基本的讨论。

1.3.1 选择名字

选择变量名可能是程序员最常做、但却想得最少的。如果你已建立了这些在大项目中出现的变量名字、类型、定义位置的清单，那么你就创建了一个类似于小电话簿的东西，你想让你的清单成为什么样子呢？不同的命名方案已发展起来了，它们有不同的思想及各自的优点和缺点，这些方案一般分为两类：简短的变量和函数名及谈话式的变量和函数名（描述变量类型和目的的更长的名字）。

某个电话目录可能是这个样子的，如表 1-1所示。

表1-1 电话目录

姓名	地址	电话
J.D.	382W.S	－3951
M.S.	204E.R.	－8382

这份列表非常有意思：该列表有两个条目，但并没有更多的信息。人名只有首字母，没有全称；只有房间号，但没有街道名；只有电话号码的一部分，却没有完整的号码。

让我们看另外一个例子，如表1-2所示。

表1-2 电话目录

姓 名	地 址	电 话
ht5ft9in_age32_John	386 West Street，Los Angeles，California，	+1－555－304－3951
Doe_male_married	USA，Earth	
ht5ft6in_age27_Mary	204 East Road，Los Angeles，California，	+1－555－306－8382
Smith_female_single	USA，Earth	

在这个例子中，每个人的名字包括身高、年龄、性别及婚姻状况。地址中不但包括街道和城市，而且也包括州、国家、甚至星球。电话号码附加了国家和地区号。

第二种解决方案比第一种好吗？两个都不是最好的。在程序课上讲授的这两种解决方案，都不令人满意，定义一种类型 tpIntMyIntegerCounter，然后声明一个变量 instMYIntegerCyunterInstance。如果仅仅需要遍历一个数组并将所有元素都设为 0，这无疑显得太冗长了（见清单1-1）。

清单1-1 一个过于冗长的实例

```
for ( $instMyIntegerCounterInstance = 0;
      $instMyIntegerCounterInstance < MAXTPINTEGERCOUNTERRANGE;
      $instMyIntegerCounterInstance++)
         $instMyArrayInstance[$instMyCounterInstance] = 0;
```

另一方面，使用I、j、k（而不是像$instMyIntegerCounterInstance这样长的名字）也是不可接受的，尤其当我们从事的是像压缩这样复杂的缓冲操作的时候更是如此。

这只是普遍思想被误用的一个简单例子，该怎么办？解决的办法是选择好的整体思想，然后在适当的地方加以例外处理，当写一个应用程序时，应该知道你的代码从事的是什么工作，能够快速地从一点转到另一点 —— 但其他人可能认为这并不容易。如果你从开发组的某个人手中获得一个源文件并需要添加一些特征，首先必须对其进行整体把握，并区分代码的各个部分。理想情况下，这一过程将和阅读源文件平行进行，但由于在没有提示和公共样本帮你理清代码来阅读的情况下，这是不可能做到的，所以在源代码中包含尽可能多的额外信息，并且使得明显的事实不易于混淆就显得很重要了。

那么如何能查知这些信息，并将其合并入自己的代码呢？

- 使代码更易读。

- 如果可能，选择谈话式名字。
- 尽可能添加一些注释。
- 保持清晰、一致的函数接口。
- 把代码结构化成逻辑群。
- 抽出单独代码块。
- 使用文件来将函数分类。
- 编写文档。

下面将讨论上述各主题。

1.3.2 使代码更易读

在阅读的时候，为了理解文章的含义，你的大脑必须分析从你的眼睛里获得的信息，识别出重要的部分，然后把这些部分译成正确的代码。这个分析过程分两步执行：形式分析和逻辑分析。首先通过检查文章的可视结构来执行形式分析，例如：检查段落、行、列甚至词之间的空隙。这一过程打破了对文章的整体了解，将其分成更小块的树形结构。假想一个结构严密的树，有顶部的树节和底部的树叶，树的顶部包含着最一般的信息，例如，你要读段落顺序，树的底部是诸如一行中的词序或是一个词中的字母顺序的一些东西。

逻辑分析过程将提取这些形式信息，然后按顺序遍历此树，并设法将信息译成有意义的结果，这是一种语法上的翻译（这个句子有什么样的结构？），还是一种语境式的翻译（这句话是什么意思？）在此处讨论中并不重要。重要的是：形式分析的结果越好，逻辑分析就越容易、越快、越好。

逻辑分析能补偿形式分析中失去的信息，但仅仅是在一个有限的程度上补偿。

```
Asanexampletakethissentenceifyoucanreadityourlogicalanalyzerworksverywell.
```

你也许能读懂前面的这个句子，但要花费比读本书其他句子更长的时间和更多的注意力，在第一步分析中，一些重要的信息（间距）丢失了，你并不习惯这样。

我们可以通过添加一些标点使其变得更简单易懂。

```
Asanexample, takethissentence--ifyoucanreadit, yourlogicalanalyzerworksverywell.
```

标点是进行形式分析的有用信息。注意到阅读这一版本或把注意力集中在所选的任意一点上要容易得多。下一步：

```
As an example, take this sentence--if you can read it, your logical analyzer works very well.
```

这是你阅读句子的常规方式，即阅读文章时最习惯的方式，但我们也可用多行结构描述这个句子：

```
As an example,
take this sentence--
if you can read it,
your logical analyzer
works very well.
```

这是可以让你能尽快地理解这个句子极端的方法的一种，上面的断句阻碍了自然的阅读语序，因为你并不习惯读一个在句法上被拆成单元的句子，但对于源代码来说，这是一个优势，

因为源代码经常包含复杂的结构、公式等。使源代码保持清晰的外在形式、结构以帮助读者理解是很重要的，这可以通过使用缩进和在适当的位置放置编程语言的关键词来实现。

让我们看一个简短的PHP程序：

```
<?function myfunc($myvar){$somevar=$myvar*2;return($somevar+1);}print myfunc(1);?>
```

这个代码本身也许并不是智力劳动的精品，我们只观察一下它的结构，如果以前没有读过这个片段，你能够一下就指出主代码的起始处吗？你能标记出主代码中最初的和最后的说明吗？即使你能一下子找到想找的地方，你的眼睛也会不由自主的从行首开始从左到右的浏览，在你认为目标可能在的地方停下来。你的大脑也要重复读这一行，因为你会不时丢失形式分析得来的信息。为了弥补起步时信息的缺乏，你的大脑（逻辑分析区）也会采取这一步，并强调两次。正如电脑一样，你的头脑的能力是有限的。所以，当你的大脑确实想要理解和记忆源代码时，逻辑分析区就在缺乏能力的情况下承担了额外工作。但是理解和记忆恰恰是你想让人们在读你的源代码时所达到的，也是你在读别人的源代码时想要达到的。

因此，这就是为什么格式化源代码很有用的原因。还有别的原因吗？噢，是的，格式好的源代码看起来让人赏心悦目。

下面是一些指导原则，其中阐述了我们所认为的在格式化源代码时的最优风格。请注意，这些指导原则不是强制要求的，但可以认为是一般的规范，许多工业的和开放式的项目已经用这种方式将源代码格式化了。

并且，采用这种风格经常会带来收益。

- 块标志符（<?、?>、<?php、<%、%>、{、}等等）要放在不同的行里。
- 用tab 缩进所有的块（理想情况下，把 tab宽度改成不超过4的值）。
- 在关键词和关系对象符之间要留有空隙，特别是在进行计算时尤其要这样做。
- 将代码的逻辑块分别放在连续的行里，使逻辑块分组，并在块之间留有空行。
- 用空行的方式分隔各个块。
- 用空行的方式把函数头、函数脚和代码的其余部分分开(输入全局变量被看作是函数头的一部分)。
- 把每一块的注释并入代码。
- 在同一块内把所有行的注释放置在同样的一些列中。

作为一个例子，清单1-2给出了某段格式化的代码。

清单1-2 重新格式化的代码片断

```
<?

function myfunc($myvar)
{

    $somevar = $myvar * 2;

    return($somevar + 1);

}

print(myfunc(1));

?>
```

大家可以看到，这一小块代码读起来要容易得多。

在代码中，空格的使用可以进一步把参数和关键词分开：

```
<?

function myfunc ( $myvar )
{

    $somevar = $myvar * 2;

    return ( $somevar + 1 );

}

print ( myfunc ( 1 ) );

?>
```

以上看似毫无必要，不过要记住：这些代码要被嵌入几千行代码之中，所以必须改变你的观点。有些人说在书写源代码文本时，括号之间的空隙与其说有帮助不如说分散了人们的注意力——我们必须承认，有些时候这是事实，本书中的例子也并不都使用这种格式。我们认为，是否使用这种格式最后由你自己决定，最重要的则是：要保持一致性。一旦你决定采用某种风格，就一定要坚持至项目的完成。如果你在修改别人的源代码，你也要尽量遵守他们的风格。在职业开发中，一致性是最重要的原则之一。

要注意阅读所有源程序的例子，并尽量模仿他们的风格，调整你自己的风格直至和这些最初的例子很接近为止，一旦你对这种风格很熟悉，你会发现你所做出的努力没有白费。

在进一步阐述之前，我们举两个例子来更好地说明这一点，如图 1-1和图1-2所示。

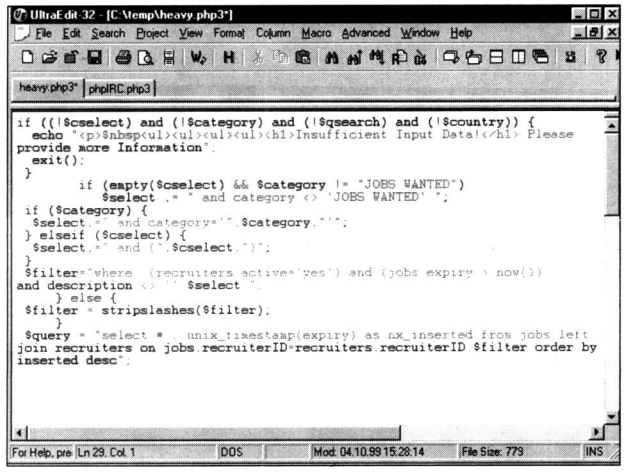

图1-1 坏的代码

图1-1中源代码是要建立一个SQL语句，除了最后的一行是把一个包含“ select * ”的字符串赋给一个名为$query的变量外，我们看不出图1-1中还有什么说明了该段代码的目的。与之相反，在图1-2中的代码中，你就比较容易理解代码的所有目的。

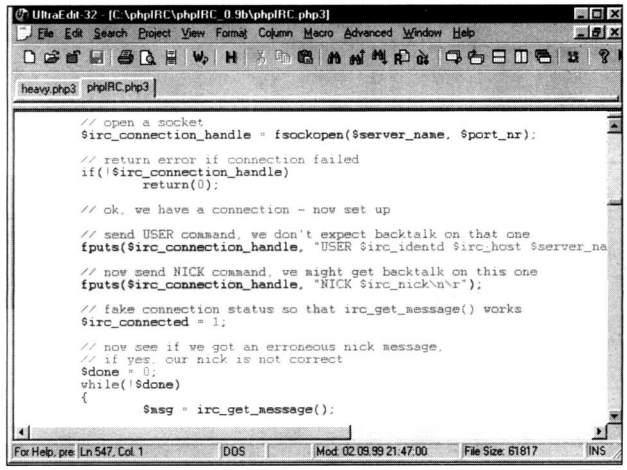

图1-2 好的代码

我认为代码就应该是这样，至少应该近似这样，代码应该有清楚的结构、很好的注释，并且很容易理解。

1.3.3 添加注释

我们无论怎样强调添加注释都不过分，尽管编程时你可能认为这是最微不足道的事情。在编写高质量的代码时，注释是很重要的。在解决复杂问题的时候，很少有两个人会有完全一样的想法，某些问题对于一个人可能是一目了然，而对于另外一个人可能是模糊不清的，在这种情况下，注释就是大有裨益的，只要需要，你都应该把它们添加到代码中。

目前主要有两种注释：头注释（例如文件头注释、模块头或函数头注释）和内部注释。头注释主要起介绍性作用，告诉读者一个文件要做哪些事情，或下面这一大段代码是关于什么的。内部注释用在函数内，或嵌入代码中以解释代码的某一行或某一块所做的工作。

下面介绍这些注释的外在感观及其所包含内容的概念。现在，这些注释通常可通过快速应用开发工具（RAD）或其他授权帮助工具来产生，但由于在撰写本书时仍没有适合 PHP的类似系统，所以这些注释应该是手编的，尽管这会增加一些额外的工作量。

下面按照注释类型的抽象程度，从最抽象的到最具体的来讨论。

保持注释不断更新

要记住在编写函数之中或之前就将其注释好，仅仅为了加注释而读一个文件是非常令人厌烦的工作。同时，要注意在以后的某个时候如果对进行函数的修改，就要适当地更新你的注释。例如，若增加或去掉全局变量，那么你也要在注释中对它们的使用注释进行更新；同样，如果参数顺序、类型等发生变化也是如此。

使用宏来加速你的注释

在你最喜欢的编辑器中，为每一种注释类型创建宏并给它们分配热键（例如，为文件头分配Ctrl+Alt+Fl，为模块头分配Ctrl+Alt+F2等等）。

如果编辑器支持的话，可以把变量引入注释中，这种创建详细的有大量信息的注释工作就变成了一个创建简短对话框问题。

1. 文件头注释

文件头可以像清单1-3那样编排

清单1-3 文件头注释

```
////////////////////////////////////////////////////////////////////////////
//
// phpIRC.php3 - IRC client module for PHP3 IRC clients
//
////////////////////////////////////////////////////////////////////////////
//
// This module will handle all major interfacing with the IRC server, making
// all IRC commands easily available by a predefined API.
//
// See phpIRC.inc.php3 for configuration options.
//
// Author: Till Gerken  Last modified: 09/17/99
//
// Copyright (c) 1999 by Till Gerken
//
////////////////////////////////////////////////////////////////////////////
```

你可能偏爱使用多行注释创建的对话框，有人认为这样美观 (如清单1-4所示)。

清单1-4 文件头注释（使用多行注释）

```
/*************************************************************************
 *                                                                       *
 * phpIRC.php3 - IRC client module for PHP3 IRC clients                  *
 *                                                                       *
 *************************************************************************
 *                                                                       *
 * This module will handle all major interfacing with the IRC server, making *
 * all IRC commands easily available by a predefined API.                *
 *                                                                       *
 * See phpIRC.inc.php3 for configuration options.                        *
 *                                                                       *
 * Author: Till Gerken  Last modified: 09/17/99                          *
 *                                                                       *
 * Copyright (c) 1999 by Till Gerken                                     *
 *                                                                       *
 *************************************************************************/
```

2. 在UNIX中提取块注释

在UNIX系统中，下面的grep命令从源程序中提取这样的块注释：

```
grep '^[\\/]*\*' source.php3
```

选择什么样的风格来格式化你的标题并不重要，但选择的由文件头包含的信息是很重要的。就像在上面例子中所看到的，标题应该包含一些整体信息，如：关于模块作者等的细节条目要按一种有意义的顺序放置（例如，包含一个长描述和一个短描述是没有意义的，当读完长描述后，就已经不再需要短描述了），下面的清单列出了我们所提倡的信息类型及其顺序：

1）模块文件名。
2）短模块描述（一行）。
3）长模块描述。
4）关于用法、要求、警告等的注释。
5）作者的名字和联系信息。
6）模块的创建和最后修改日期。
7）版权注意事项。
8）许可注意事项。
9）转变记录、主页、分配文件等的指针。
10）最后，如果需要，变化记录中的摘要。

如果这些听起来太多了，那么记住，宁可有多余的信息，也不要缺乏信息，当然，这并非在所有范围及所有条件下都合适，我们没有在前述的例子中包含所有情况。然而，你应该设法向你的标题中放置尽可能多的数据——这是一种良好的习惯。最坏的情况是有些人可能不去读它，但有可能有些人感激它——也许就是你自己，因为在一个商业化项目中，如果你忽视了版权和许可注意事项，而当别的程序员免费更新你的代码时，则会导致令人头疼的后果。

3．模块头注释

如果在一个文件中不止一个模块（例如，当某个模块组的一个模块包含三个函数时），应该在第一个函数前放一个信息量很大的标头。模块头形如清单 1-5所示。

清单1-5 模块头注释

```
//////////////////////////////////////////////////////////////////////////////
//
// Submodule for file access from main()
//
//////////////////////////////////////////////////////////////////////////////
//
// This submodule will provide functionality for easy file access,
// and includes error checking and reporting.
//
// Functions:
//
//    int file_open(string $file_name)
//    bool file_close(int $file_handle)
//    int file_read(int $file_handle, $nr_bytes)
//
// Remarks:
//
//    - provides no seek function
//    - does not allow write access
//
//////////////////////////////////////////////////////////////////////////////
```

这些标题按顺序可能包含如下各项：

1） 短模块描述。
2） 细节模块描述。

3） 函数原型清单。

4） 标记/注解。

多行注释再一次表现出其优越性。

4．函数头注释

函数头应足够细致地为每一个函数（见清单 1-6）描述句法、目的和必要的调用者信息。这些注释的重要性，相对于内部注释来说是次要的，函数头注释的目的是让程序员在模块开发和扩展中迅速了解每一个函数的要求，这些要求是为最初没有建立这些函数的“外人”所提供的，缺乏函数头注释的源代码经常需要开发者深入其中找到所要信息，而这一点经常会导致错误，因为不是所有隐藏的陷阱（有时它们隐藏得很好）都会被发现。

清单1-6 典型的函数头注释

```
//////////////////////////////////////////////////////////////////////////////
//
// int irc_get_channel_by_name(string $name)
//
//////////////////////////////////////////////////////////////////////////////
//
// Searches a channel by its name in the internal channel table and returns
// its handle.
//
//////////////////////////////////////////////////////////////////////////////
//
// Parameter:
//    $name - name of channel to search for
//
//////////////////////////////////////////////////////////////////////////////
//
// Return value:
//    Channel handle (numeric), 0 on error
//
//////////////////////////////////////////////////////////////////////////////
//
// Global references:
//    $irc_channel_array
//
//////////////////////////////////////////////////////////////////////////////
```

一个函数头注释应按顺序包含如下各项：

1）函数原型。

2）函数细节描述。

3）标记/注解。

4）参数描述。

5）返回值描述。

6）全局引用。

7）作者和最后一次修改的日期。

5．内部注释

内部注释直接放入代码中，并直接解释所有产生的问题。当你编写代码时，每件事你自己当然是很清楚的，这就是有人经常不写注释的一般原因。后来当你重新打开这个文件时（甚至也许是一年之后），你也许已遗忘你用的所有结构及使用它们的原因，这是我们经常遇到的一个问题。在我们自己的代码中或别人的代码中使用内部注释的原则是：注释越多越好这一原则的唯一例外是，注释不能被滥用到让人们对代码模糊不清的程度，同时，注意不要注释显而易见的东西。清单1-7列举了一些例子。

清单1-7 不好的内嵌注释

```
function calculate_next_index ( $base_index )
{

    $base_index = $base_index + 1;          // increase $base_index by one

    //
    //
    // Table of contents
    //
    // 1. Introduction
    // 2. About the authors
    [LOTS of lines cut out]
    //
    //
    $new_index = $base_index * COMPLICATED_NUMBER / 3.14 + sin($base_index);

}
```

在第一行中，因加1而增大的$base_index代码是需要注释的语句吗？我们表示怀疑。每一个人都能看得出$base_index正加上1。但它为什么加1？为什么正好加1？更好的注释大致是这样的：跳至我们所指的下一个指数，它仅有一个元素的距离。

第二个注解有同样的问题，但产生的原因不同。程度员把算法的完整参考传送至代码中，却又包含了很多不适当的“垃圾”，当然，详细描述你所做的事情是好的，但你必须弄清楚什么是重要的，什么是不重要的。

当你给代码添写注释时要考虑如下问题。

- 你在做什么？
- 为什么要做这件事？
- 为什么要采用这种方式做？
- 为什么要在这个地方做？
- 这个代码如何影响其他代码？
- 这个代码要求什么？
- 你的方法有什么缺陷吗？

例如，当你分析字符串的时候，记录输入串的格式，你的分析器的偏差（它对输入中的错误的反应）和它的输出。如果这些信息太多，以致不能直接嵌入你的代码，那么至少要安置一个指针，指向一个外部文件，在此文件中读者能够了解到分析器的各个方面。同时，要记住更新函数头注释，即设置一个对此文件的链接。

1.3.4 选择谈话式名字

正如前面所提到的，为函数和变量选择合适的名字在编程中是一个很重要的问题。一般情况下，当为一个变量选择名字时，首先要确定它是全局变量还是局部变量，如果此变量仅在函数的局部作用范围内可见，那么就给它选一个简洁、准确的名字来陈述此变量的内容或意义，这个变量名应该至少包含两个词，这两个词或者被下划线分开或者被大写字母分开，如清单 1-8 所示。

清单1-8 局部变量名实例

```
$counter
$next_index
$nrOptions
$cookieName
```

记住不要混用命名方案，要么都用小写字母来写变量名，用下划线来分隔词，要么使用大写字母来分隔词。不要用大写字母来分隔一个变量而用下划线来分隔另一个，这会导致错误，并且表现出不好的风格。一旦定好你自己的风格，就一直坚持到项目结束。

每一个全局变量都应该有一个前缀来标识它所属的模块，这一方案帮助把全局变量赋给它们的模块，同时也可避免出自不同模块的同名变量在全局范围内产生冲突。前缀应该用下划线和变量名分开，并应该包含一个词 —— 多数是一个缩写（见清单 1-9）。

清单1-9 全局变量名的例子

```
$phpPolls_setCookies
$phpPolls_lastIP
$db_session_id
$freakmod_last_known_user
```

小尺寸优势

创建更小的项目，每一个项目都用不同的命名风格。原因如下：

- 你能发现你偏爱的风格。
- 当你不得不适应别人风格时，能够很快变得熟练。

如上例所示，全局变量名倾向于比局部变量名长，这不仅是因为全局变量具有模块前缀，也是为了分清全局变量和局部变量。当一个变量的定义和初始化因隐藏在一个你接触不到的模块中而变得未知时，用变量的名字来思考它的意义和内容就显非常重要。这在实践中当然有个极限 —— 没人想记住多于四十个字母的名字 —— 但这只是一般意义上的极限。

从根本上讲，你应该命名全局变量就像向某人描述它一样。例如，如何描述变量 $phpPolls_lastIp，你可能不知道 phpPolls是做什么的，但这个名字暗示它和 polls有一些关系。lastIP意指它是最后一个 IP。哪一个IP，你不知道。显然，这个全局变量的名字选得不太好，因为它并没有准确地描述其内容。现在假定你问这个变量的含义是什么，答案是，它包含最后一个投票者的IP，现在想想该给它取一个什么名字？ $phpPolls_ last_voters_IP听起来如何？更好一点，不是吗？尽管这个名字可能很好，但它仍不合适，因为你曾见过另外两个同样出自 phpPolls

的全局变量，都以phpPolls_为前缀，然后紧跟一个词，出于一致性的考虑，你可以决定在名字内部仅用大写字母来分隔不同的词：$phpPolls_lastVotersIP。

函数名也应该用与全局变量名相同的相近风格加以处理，但略有不同。函数命名应描述它们的功能而且要符合语流，让名字符合语流是通过确定函数行为、并选择在该名字大量出现之处最适合的名字来实现的。

例如，如果用一个函数确定一个用户目前是否在线，它可能有以下名字中的一个：

```
function get_online_status($user_name);
function check_online_status($user_name);
function user_status($user_name);
function user_online($user_name);
function is_user_online($user_name);
```

考虑到返回值类型，上述清单中只有第一个和最后一个名字是合适的。假定函数将返回一个布尔值，那么它经常用在一个与if()语句的连接处。在那里，它一般是这样的：

选择 1：

```
if(user_status($user_name))
{
    // do something
}
```

选择 2：

```
if(is_user_online($user_name))
{
    // do something
}
```

在第一个选择中，函数名看起来不是很恰当，“If the user status of Jahn then do something.”再检查一下，第二种可能性：“If the John is online then do something.”，第二个观点没有打破语流，并且在第一眼见到的时候给人留下了更多印象。第一个选择把问题公开化：什么身份被谈及？该身份如何返回？第二个函数名清楚地表示这个函数会检查某人的在线状况并返回一个布尔值。

如果检查结果在函数的变量参数中返回又会怎样？

选择 1：

```
function user_status($user_name, &$status)
{
    // retrieve status and return in $status
}

$success = user_status($user_name, $user_online);
```

选择 2:

```
function get_online_status($user_name, &$status)
{
    // retrieve status and return in $status
}

$success = get_online_status($user_name, $user_online);
```

尽管user_status()并非一个不好的名字，但 get_online_status()更好一些。“ get ”这个词很清楚地表明函数检索在线状态并将其存于某个地方 —— 或者在一个全局变量中，或者在一个函数变量中。

对于仅仅进行数据处理的函数来说，要使用主动的而不是被动的名字。不要使用如huffman_encode()或database_checker()这样的名字 —— 将函数命名为 huffman_encode()和check_database()或将两个词交换顺序，这将很好地适应模块前缀。

你的代码是两种语言的还是三种语言的

对代码最普遍的批评之一涉及“ 民族化 ”，一种程序语言（起源于英语）与另一种程序语言搅合在一起。在我们的实际例子中，（Tobias源于意大利语，Ti源于德语），当我们检查各自国家程序员开发的项目时，我们发现他们喜欢使用德语和意大利语变量名和函数名而不是用英语。这导致了一种奇怪的混淆。正如你不会在你的日常信件中混用英语、法语、西班牙语等一样，所以，你在编程时也需要保持语言一致性，使用英文名字编写PHP程序，还有助于外国人理解你写的程序。

1.3.5 保持清晰一致的接口

你也许不愿意再看到“ 一致性 ”这个词，但对于接口设计来讲，它是编程基石中的关键一块。

非常不幸的是，PHP本身恰恰存在如何违反这一点的例子。

你在驾驶汽车的时候，油门在右而刹车板在左。当你换一辆车时，你希望情况也是如此，无论你在哪里，你都希望红灯意味着停止，而绿灯意味着前进。类似地，当你用一个库访问文件，且需要把一个文件句柄传给函数时，如果输出函数把文件 handu句柄作为第一个参数，输出函授将其作为最后一个参数，而另一个把它作为中间参数，那么这会令人感到莫名其妙。

当设计接口时，你应该首先考虑如下问题：

- 通过这个接口交换什么数据？
- 我到底需要什么参数？
- 大多数（或所有）的接口函数所共有的参数是什么？
- 这些参数最合乎逻辑的顺序是什么？

把它们牢记在心中，一旦你决定采用何种方式去做，你就应该在你的模块中保持参数一致性。即使内部函数也应遵从这一点。这一策略将使你以后能从接口中获得内部函数。另外，当开发组的同组成员要将新的代码加入你的模块中时，他也会很感谢你让他们省了不少功夫。

在PHP手册中，你会看到诸如strpos()、strchr()、strrchr()等一些字符串函数。它们之中，都会有参数string haystack，string needle等，其中haystack是在其中搜索的字符串，而 needle是要搜索到的字符串。现在，再看一下 str_replace()，这个函数不仅会突然采用了一种不同的命名方案，而且其参数也恰恰和其余的函数相反：它接受 strig needle和string haystack。

产生这种矛盾的原因是： str_replace()是对ereg_replace()的一个快速替代，并且大部分人会从调用ereg_replace()(接收相反顺序的参数)转到调用str_replace()。当然，这种说法有一定道理，

但是为什么regex函数按一种与字符串函数相反的顺序接收参数呢？因为在 PHP中，regex函数反映了在C中的相应函数。在开发一个应用程序的时候，看到 str_replace()从其余函数中突现出来是很别扭的事。在勾勒下一个接口的轮廓时，注意不要让这种情况发生在你的身上。

1.3.6 将代码结构化为逻辑群

应用程序通常包含不同的函数群，每一个函数完成一项特定的任务并（或）应用于特殊的应用领域。例如，在写一个支持数据库的应用程序时，一个函数群应该仅仅对处理数据库访问负责，这个代码确立了它自己的存在，能够安全地从程序的其余部分分离出来 —— 只要你设计得好。逻辑上只从事一项特定任务的函数群应该用某种方法设计，以使他们能够被独立地处理，这些函数在形式上也应该和主代码分开，建立一个模块。在运行一个应用程序之前，你应该建立一个能将所有函数归类在一起的函数清单，形成一个模块，并为每一个模块创建一个各自独立的设计计划。要注意创建详细的数据流程图，以便使模块能够满足应用程序的各种要求。做一个书面的整体计划，其重要性不可低估。由于篇幅所限，我们不能够再深入谈及这个问题，但我们建议你读一些关于设计方法的好书。

1.3.7 抽取单独的代码块

抽取代码块是一项在设计和实施阶段都应该做的事情，通常一个函数应该能完成以下工作：

1）开一个文件。
2）从文件中打读取数据。
3）证实数据（将数据合法化）。
4）更正数据中的错误。
5）将数据写入文件。
6）关闭文件。

每一步都可以“ 包装 ”成单独的一个程序块，抽取这些块并从中创建单独的函数是一种很好的方法。这不仅使你能够在别的函数中重新使用每一个程序块（你可能在别的地方也需要文件操作的支持），而且还能使代码更容易阅读和纠错，你可以使被抽取的部分“ 放弹 ”，给它们装备“ 纠错器 ”，以支持更多的东西。如果你采用内嵌法无法做到这一点，你的代码会很快变得异常庞大而冗赘，另外，如果你在其他的函数中，使用同样的程序块时产生需纠正的错误，你将不得不在使用此块的所有其他的函数中反复进行同样的纠正。

通过提取，可以把关键部分放在中心位置，只要更改一行程序，就可以改变所有相关函数的行为。

1.4 使用文件将函数分类

我们已经论述过对源代码使用复合式文件是有好处的，但我们也同样建议你为其他资源使用文件。这些资源可以是配置数据、客户标题、页脚或其他模板，以及任何从你的项目中可以抽出来作为一个单独实体而存在的东西。

在一个项目中使用模块有很多好处：

• 可以获得更小更容易维护的源代码文件。

• 可以对每一个文件进行不同的修改，而不必在整个项目中进行检查以进行一个微小的修改。

• 可以将部分资源从项目中分离出来，用在其他项目中。

• 许多开发组成员能够同时工作在一个项目上，而不必在检查时将所有的文件合并成修正控制系统。

以上论述适用于一个项目中存在的大部分资源。

文件应根据其内容加以命名。如果一些文件从属于一个更大的群体，可以给它们加一个共同的前缀，文件一般应该放在项目根目标的子目录下。例如，一个数据库提取层，其中有可访问不同数据库的模块，这些模块被“包装”成单独的文件， 每个文件名应冠以前缀dba_（这里dba代表database abstraction），这样你就得到了dba_mysql，dba_odbc，dba_oracle等名字。

要确保能够在将来通过使用配置模块目录来变更子目录，例如（注意 dba在本例中并不指代PHP的dba_*functions）：

```
<?
require("config.php3");

require("$dba_root/dba.php3");
require("$socket_root/socket.php3");
require("$phpPolls_root/phpPollUI.php3");

// [...]
?>
```

事例中的变量$dba_root、$socket_root和$phpPolls_root应该包含在一个中央配置文件中，该文件有对整个项目的全局化“选项”。该配置文件应该包含独立的源文件所需要大的，能使其在全局范围内可用的选项。这种“选项”可包括环境选项，如站点名、文件系统位置等等。

停留在（普通的）路径上

当某子目录包含配置文件时，要一直使用相对路径以确保项目在文件系统及用户系统上是灵活的 —— 不依赖开发环境的任何特定条件，就像在其他环境下一样。能保持一般化的东西就要尽量让它一般化。

1.5 编写文档

除了注释和结构化以外，文档也是值得注意的，一个项目的文件记录可能是你的用户将要见到的项目的第一部分，而第一部分是至关重要的。

规范化写出的文档应该是开发过程中惯例性的一步。正如你希望微型电话或其他哪怕是在很小的商店中购买的技术产品都有一本写得很好的手册一样，你的用户也希望从你那里得到较好的文档（更不用说他们可能会为此而付一大笔钱了）。

和注释一样，文件记录通常是在 RAD工具的帮助下产生的，很不幸，目前还不存在专为PHP设计的相应工具，所以写手册是一项费力不讨好，但却很有必要的一份工作。并且，这并不会影响你的工作效率。一个完整的手册应具有像书一样的内容结构，一般包括以下几项：

- 介绍。
- 内容表。
- 用户指导。
- 技术文件。
- 开发者指导。
- 完整的函数参考。

用户指导应该详细地描述为标准用户设置的应用程序接口（如果有的话）的所有特征，在这一部分不要太专业化，它应该仅仅是一个“如何”程度上的描述，但要确保每一方面都阐述得很详尽。技术文件应该为对技术感兴趣的用户和管理者而写，并应包含应用程序的技术要求、使用和引入的规范以及关于内在数据处理的信息（只要这是读者所感兴趣的）当然，这也要在你许可的允许范围之内。如果你允许用户看见和（或）修改源代码，那么编写一份开发者指导来解释项目的结构、数据流、内在关系以及列出所有的函数参考（包括内在函数），并要有完整的描述。

如果你在一个开发组中工作，职业技术作者将是这一群体的有力助手——他们有书写技术文件的经验，也有充裕的时间。让一个有开发任务的组员同时写文件记录会导致大量额外的压力，因为程序开发者总是很忙碌的，他们不想误期。

1.6 一个API设计实例

参照所有的理论，我们设计一个应用程序接口，以使你熟悉前面所讨论的思想和规范。请注意，这是一种实际的解决方法，而非一种理论上的方法。我们采用这种实际的方式是为了让你熟知每一步。在今后的项目中，你必然在纯理论的基础上设计 API，而不必首先看代码。关于理论方法的线索、提示和决窍，参见第 3章。

我们创建的API模块是用来处理一个简单的日程管理器。这个日程管理器函数的实际应用并不重要，记住，这恰恰是使用户模糊不清的地方。用户只是想管理一组约会，因此 API必须以这样的方式来设计，即提供一个约会管理的接口。无论你是在用 Julian或Gregorian日期还是你自己的格式，都不必通知基本系统的用户，在某些时候，你可能想给用户提供一套额外的功能（例如：日期格式转换），但如果你所需要的仅仅是管理约会，这就是完全不必要的。

另一方面，这并不意味着阻止甚至破坏这些功能的进一步使用，设计一个 API的技巧在于它恰好满足你一时的要求，即能够把 API扩展到最终需要的功能。这需要深入的计划和定义，正如本章一直在讨论的那样。

API是访问其自身所代表的模块功能的唯一途径，没有功能会丢失，也不会有任何不必要的功能会出现，甚至并不直属这一模块的功能都不会有。

一个简单日程管理器的要求如下：

- 增加一个事件。
- 删除一个事件。
- 检索即将发生事件的清单。

让我们首先为增加和删除事件定义原型；如清单 1-10所示，这些函数需要什么信息，又能提

供给我们什么返回值呢？

清单1-10 前两个函数的原型

```
void add_an_event(int day, int month, int year, int hour, int minutes,
➥int seconds, string description);
void delete_an_event(int day, int month, int year, int hour, int minutes,
➥int seconds);
```

由上我们最先得知的是：一个可接收“一般意义”参数清单的接口，即用日/月/年表示的日期和用小时/分钟/秒钟表示的时间，以及描述一个约会的字符串，这些函数无返回值，它们的名字是谈话式的。

谈话式的？是的，但是它们是很好的谈话式名字吗？ add_an_event()是谈话式的名字，但对这个函数来说并非最佳选择。首先，由这个函数的全局可见，它是 API的主要元素。既然这样，它就应该有一个名字前缀以清楚表明它本身也属于 API。应该加一个什么样的前缀呢？ calendar和scheduler是很好的方案，在这个例子中，我们选用 Calendar（见清单1-11）。

清单1-11 重命名后的函数原型

```
void calendar_add_an_event(int day, int month, int year, int hour, int minutes,
➥int seconds, string description);
void calendar_delete_an_event(int day, int month, int year, int hour, int minutes,
➥int seconds);
```

现在，我们有了个前缀，但名字仍让人不满意，在“ calendar_add_an_event()”和“calendar_delele_an_event”中的“an”是不必要的。它是产生于选择“过度谈话式”名字的遗迹，选择函数名时删除诸如a、an和the一类的词是一个很好的习惯。在大多数情况下，这些词占用空间但却起不到多大的区分作用，因为它们没有解释功能。特别地，当选择变量名时，这些词应该彻底避免，选择诸如$a_key或$the_key一类的名字是毫无意义的，因为key是显而易见的。选择一个可以解释什么key的名字会更有意义。如：$last_user_key。

清单1-12列出了重命名的函数。

清单1-12 最终函数名

```
void calendar_add_event(int day, int month, int year, int hour, int minutes,
➥int seconds, string description);
void calendar_delete_event(int day, int month, int year, int hour, int minutes,
➥int seconds);
```

下面转到另一个问题，这些函数有庞大的参数表，有这个必要吗？这些参数是根据一般的日期格式，即把日、月、年、小时、分钟、秒钟分开的格式选择的。然而，用一个接口来交换信息是不正规的，函数几乎不应该接收五个以上的参数。如果有更多的参数，你应该考虑使用结构体，结构体可以使接口变得清晰，这在很多时候是一个比避免初始化和（或）修改结构体而带来额外工作量更显得有意义。

在把所有的参数都放置到结构体中之前，仍有替换数据格式的可能性，为了将日期和时间代码化，你可能会使用BCD（Bianry Coded Digits）码或UNIX timestamps格式，这两种格式把

所有需要的变量“包装”到一个变量中。BCD码仍是广泛流行的代码，但在产生于UNIX式平台的PHP中，timestamps占据主导（见清单1-13）。如果你还没有遇到过timestamps，这里简单介绍一下:timestamps计算自1970年1月1日UTC零时以后的秒钟的数日，并以32位十进制数表示出来，这会在2106年导致一次回绕。但是由于PHP设有固定的32位类型来处理timestamp，PHP有可能把timestamps的大小转变为64位以保持Y2.106K适用性。你的应用程序不会觉察到这一点。Timestamp的另外一个优点是：有大量的PHP函数可以把它们转换成人们可读的日期或进行相反转换。用timestamp来进行计算也是很容易的，例如为了得到两个事件的不同点，你只需把一个timestamp从另一个里面扣除。

清单1-13 修正的API

```
void calendar_add_event(int timestamp, string description);
void calendar_delete_event(int timestamp, int seconds);
```

正如你所见到的那样，为了处理一种特殊的数据，进行现有格式和方法的检查是非常重要的。目前的格式不仅把参数清单缩小了350%，而且它也是一个处理日期和时间的基本结构的基本格式。检查文本格式和现存标准是一个在研究阶段永远都不该忽视的步骤。在开发阶段，也不应受任何偶然事件的影响。了解开发范围是必须的。

把这些牢记在心中，让我们看一下第三个必需的函数，它用来检索即将发生事件的清单，我们就要遇到问题了，因为返回值不是一个，而是一组相关变量的清单。

时间信息 1 => 描叙 1
时间信息 2 => 描叙 2
时间信息 3 => 描叙 3

这些数据可以通过用参考变量传递参数的方式返回（参见第2章，“高级语法”）：

```
//
// List function in pseudocode
//

function calendar_get_event_list($range, &$timestamp, &$description)
{

    while($current_timestamp < $range)
    {
        $timestamp[] = $next_event_timestamp;
        $description[] = $next_event_description;
    }

}
```

这个伪代码会在两个数组 $timestamp和$description中按请求顺序排列即将发生的事件。Index1将在$timestamp[0]中包含事件1的timestamp，在$description[0]中包含事件的描述。

然而，这仅是一个非最佳解决方案，因为让两个分离变量处理集群化的元素是一种不恰当的方法。为了处理集群化元素，应该使用集群化数据类型或者是一个类（这是PHP中唯一建立结构化类型的方法）或者是一个相关数组。

相关数组的优势是：即可被下标（索引组成元素——在通常数组中，一般是0、1、2、3等）搜索又可被值（信息量大的组成元素）搜索。但是此处，它们有一个变化的结构，这种结构能被改变，但会导致不合法结构数据的存在，并且处理起来有些笨拙。

类有完善展示自身结构的优势，但需要一个预先定义的数据类型。如果我们为返回值定义一个数据类型，出于一致性的考虑，我们也用这个数据类型来创建和删除事件。这反过来会要求我们修改现存的函数——仅仅填加一个函数是不会令人满意的，你现在可以看出事先进行的详细理论计划可以为我们节省宝贵的时间。在开始定义头两个函数前，定义一个结构化的数据类型将使我们在定义函数时可以使用这一类型，这样我们就有一个可以在清单函数中重新采用的一步到位的解决方法。

由于一个类将会向代码中引入一种风格，我们一般使用相关数组。清单函数将不会返回错误代码，所以我们使用函数的返回值来把数据传递给调用者。记住，如果你打算使用错误代码，你应该使所有函数返回错误代码，即使它们会永远成功，你也应该创建一种一致的错误代码方案因为通常地，你的API用户并不知道某一函数是否会成功运行。但他们希望如果函数运行错误时都会返回一个错误标志值。第3章有更多的关于这一点的内容。

返回到清单函数，下面是选用的函数类型：

```
function calendar_get_event_list($range)
{
    // Retrieve event list
}

$event_list = calendar_get_event_list($required_range);

for($i = 0; $i < count($event_list); $i++)
    print("Event at $event_list[$i]["time"]: $event_list[$i]["text"]<br>");
```

这些代码可能会产生如下结果：

```
Event at 95859383: Team meeting
Event at 95867495: Deadline for Telco project
Event at 95888371: XML Seminar
```

看起来不错，但在代码中有另一个主要的错误，在for()循环中，数据在二维数组中使用相关键标time和text来返回，这些变量在早些时候被分别命名，它们是针对时间的 $timestamp变量和针对描述文本的$description变量。当填充相关数组时要为键标使用与变量相应的名字。在这里for（）循环可以访问如下数组：

```
function calendar_get_event_list($range)
{
    // Retrieve event list
}

$event_list = calendar_get_even_list($required_range);

for($i = 0; $i < count($event_list); $i++)
    print("Event at $event_list[$i]["timestamp"]:
    ➥$event_list[$i]["description"]<br>");
```

1.7 小结

应用程序开发不仅仅是草草写下代码、使句法准确并保证软件运行。因为软件不仅要被计算机读，将来也要被程序员（或你自己）读，源代码应该清楚、准确、简洁、书写良好，容易阅读、有注释、使用自然语言表达， API应该构造清晰易懂、前后一致的接口；应该被结构化成逻辑单元，并在最后做出摘要。由于大的项目即使用最清晰的代码编写也不能不言自明，所以技术文件是必须撰写的。

本章介绍的编码规范，是以来自许多程序员所积累经验的一般意义上的指导原则为基础的，并不是强制性的规则。它们不难掌握，会使你和你的编程伙伴的生活更加轻松。