

前言

为什么要写这本书

近几年，市场上关于PHP的书已经很多了，各种培训机构也如雨后春笋般不断增加。那为什么还要写这本书呢？这本书存在的意义又在哪里？这要从下面的几个问题说起。

有没有这样一本PHP教材，它不讲HTML和CSS，也不讲JavaScript基础，甚至不讲PHP语法基础？

有没有这样一本PHP教材，它不讲留言板或博客的开发，也不讲数据库的CRUD操作？

有没有这样一本PHP教材，它专注于Web开发技术的最前沿，深入浅出，适合中高级程序员的进阶和提高？

有没有这样一本PHP教材，它提倡面向对象的程序思想，提倡算法和数据结构的重要性，提倡对网络协议的深入理解，且没有大篇幅的代码，而是更多偏重于理论讲解？

有没有这样一本PHP教材，它探讨PHP的扩展开发，探讨高并发大流量的架构，深入探讨NoSQL的内部实现和细节？

以上几个问题也是我在早期PHP学习的过程中一直在寻找的答案，可是我并没有找到一本理想的PHP书籍，一本适合中高级程序员进阶的书籍。当怀着同样问题的旭松兄找到我时，我们不禁产生一个念头：“既然现在市场上缺少一本这样的书籍，我们何不自己写一本呢？利己利人的事值得去做。”然后一拍即合，说做就做，现在这本书经历长达一年多的酝酿和写作过程终于完稿了。

我是在大学期间接触到PHP语言的，并马上被其简洁的语法和极高的开发效率所吸引，一头扎进PHP开发的世界中。随着学习的深入，并经常关注PHP社区的动态，我很快意识到一些PHP社区普遍存在的问题。比如PHP社区一直争论算法重不重要，面向对象好不好，代码质量重要还是开发速度重要的问题。还有譬如为什么我去大型互联网公司应聘PHP程序员，却不考察我对PHP语法和函数的掌握情况，而是会问我C语言、算法、

网络协议、高并发处理、MVC理论这些看似和PHP不沾边的问题。

PHP到底要怎么学，学什么，一个高级PHP程序员应该是什么样的，我想这也是很多PHP新手和工作一两年的PHP开发者的疑惑。这本书所要解决的就是这一系列的问题。

在我看来，一本技术书籍的价值在于其对知识的提炼和与众不同的地方。举例来说，到一个书店去看书，你最想用笔抄下来或撕下来带走的那几页，就是对你帮助最大的东西，也是你认为这本书的价值所在。也是基于这个想法，我们思考这本书该写什么，怎么写，哪些地方对读者有帮助。我们试图从不同的角度带领读者来看PHP，进而给这本书注入一些不一样的东西。我们希望这是一件有意义的事。

本书适合的对象

PHP爱好者；

想进阶的初级PHP程序员；

对PHP扩展开发感兴趣的读者；

对高并发感兴趣的读者；

对NoSQL应用和实现原理感兴趣的读者；

从事PHP网络应用，想知道HTTP协议、Socket等更多细节的开发人员；

想就职于大型互联网公司的PHP程序员；

开设相关课程的大专院校的学生；

公司内部培训的学员。

如何阅读本书

本书一共有14章。每章节都可以单独阅读，由于部分知识点之间存在一定的衔接，故建议按先后顺序阅读。

第1章 为面向对象思想的核心概念。本章主要讲解面向对象开发的思想，重点讲解

面向对象模型的建立，以及面向对象的一些基础概念。通过大量对比和实例，尤其是与Java的对比，力图从不同角度讲解PHP面向对象的特性，让PHP程序员看到不同的面向对象。求同存异是本章的核心思想。

第2章 为用面向对象思维写程序。本章用简练的语言讲解了面向对象设计的五大原则，这五大原则也是理解设计模式的基础所在，帮助读者站在一个更高的角度思考面向对象。

第3章 为正则表达式技巧与实战。本章详细介绍了正则的基础语法，通过大量的示例、通俗的语言讲解正则概念，引导读者理解正则的一系列规则。接下来，结合实际工作用安全过滤、URL重写等实例，加深对正则的应用和掌握。最后给出正则效率优化的一些普遍技巧和替代方案，让读者对正则的使用得心应手。

第4章 为PHP网络技术及应用。本章着重介绍了HTTP协议、Socket开发、WebService、Cookie和Session使用等。结合实战向读者阐述网络开发的核心和重点，特别是对HTTP协议的理解。HTTP协议是Web开发的基石，也是各种面试和开发中必然遇到的知识点。而Socket则是应用交互的桥梁，保证了有用的可扩展性。

第5章 为PHP与数据库基础。本章从不同角度分析了MySQL，介绍了PDO、MySQL优化、存储过程、事件调度机制以及MySQL安全防范等内容。

第6章 为PHP模板引擎的原理与最佳实践。本章通过实现一个简单的模板引擎，学习模板引擎的原理和使用方法，然后对比几大流行的模板引擎实现方案，简单介绍了各种实现方案的思想 and 优缺点，最后探讨模板引擎的意义。

第7章 为PHP扩展开发。本章的知识是本书核心内容，介绍了PHP扩展开发的几个重要知识点，如扩展框架搭建、PHP生命周期、PHP变量在内核中的实现方式、Zend引擎、内存管理等，让读者深入PHP底层，知其然也知其所以然。

第8章 为缓存。本章主要介绍了缓存的基本原理和三个衡量指标，通过几个实例加深读者对缓存的理解。利用本章知识，读者应该能设计一个比较合理的缓存方案。

第9章 为Memcached应用与内幕。本章深入剖析了Memcached的实现和内部结构，从而使读者掌握Memcached的高级应用，对构建复杂环境的缓存层有个清晰的

认识。

第10章 为Redis应用与内幕。本章重点介绍了Redis的深入应用，如事务处理、主从同步、虚拟内存等，和第9章类似，探讨了Redis的实现内幕。合理利用Redis可以为我们解决大流量高并发的应用。

第11章 为高性能网站架构。本章探讨了高性能架构的基本出发点，重点以HandlerSocket、MySQL主从复制、反向代理缓存软件Varnish和任务分发框架Gearman为例，讲述几种高性能架构中会用到的技术。

第12章 为调试与测试。科学的调试方法有助于快速找出潜在的Bug、理解复杂应用的流程、提高开发效率。单元测试是代码质量的保障。在这一章的最后一节介绍了使用JMeter进行压力测试的方法。

第13章 为Hash算法与数据库的实现。本章介绍了Hash算法的基本原理，用此算法实现一个简单的、基于Hash的数据库，让读者意识到算法的重要性和可操作性。

第14章 为PHP编码规范。本章介绍了PHP开发中应遵循的基本代码规范，并提出合理建议。好的代码必然是规范的代码。

本书第1、2、3、5、6、8、12、14章由陈文撰写，第7、9、10、11、13章由列旭松撰写，第4章由两人共同完成。

勘误和支持

由于我们的水平和开发经验有限，同时计算机技术更新较快，书中难免存在不足之处，有些章节内容可能从未来的某一天开始不再适用，还望读者理解和体谅，并恳请读者批评指正。您若对本书有什么好的建议或者对书中部分内容有疑惑，可与我们联系，我们将尽量为读者提供最满意的解答。期待得到您的真挚反馈。我们的联系方式如下：

陈文：waitfox@qq.com

列旭松：liexusong@qq.com

感谢

首先要感谢PHP之父Rasmus Lerdorf，是他创建了这个简单、轻松、有趣、快速而又高效的语言；其次，感谢PHP社区每一位充满活力的朋友，和你们的交流使我学到很多，本书有不少内容就来自于社区的智慧。

在这里尤其要感谢机械工业出版社华章公司的大力支持，特别是杨福川和白宇两位编辑，在一年多的时间里，因为有了你们的耐心指导、逐字逐句认真审稿和改稿才有了本书的诞生。

最后，还要感谢家人和朋友的支持。

陈文

第1章 面向对象思想的核心概念

面向对象是什么？以下是维基百科对面向对象的解释：

面向对象程序设计（Object Oriented Programming, OOP）是一种程序设计范型，同时也是一种程序开发方法。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和可扩展性。

面向过程、面向对象以及函数式编程被人们称为编程语言中的三大范式（实际上，面向过程与面向对象都同属于命令式编程），是三种不同编码和设计风格。其中面向对象的核心思想是对象、封装、可重用性和可扩展性。

面向对象是一种更高级、更抽象的思维方式，面向过程虽然也是一种抽象，但面向过程是一种基础的抽象，面向对象又是建立在面向过程之上的更高层次的抽象，因此对面向对象的理解也就不是那么容易了。

面向对象和具体的语言无关。在面向对象的世界里，常常提到的两种典型语言——C++和Java。它们都是很好的面向对象的开发语言。实际上，像C语言这种大家普遍认为的面向过程开发的主打语言，也能进行面向对象的开发，就连JavaScript这门很久之前一直被视作面向过程编程的语言，人们对它的认识也发生了改变，逐渐承认其是面向对象的语言，并且也接受了JavaScript独特的面向对象的语法。所以我们说面向对象只是种程序设计的理念，和具体的语言无关。不同的程序员既可以用C语言写出面向对象的风格来，也可以用Java写成面向对象的风格。这里并不是说面向对象的风格要优于面向过程，而是二者各有自己所擅长的领域。OOPL（Object Oriented Programming Language）可以提高程序的封装性、复用性、可维护性，但仅仅是“可以”，能不能真正实现这些优点，还取决于编程和设计人员。就PHP而言，其不是一门纯的面向对象的语言，但是仍然可以使用PHP写出好的面向对象风格的代码。

实际开发中，面向对象为什么让我们觉得那么难？面向对象究竟难在什么地方？为什么面向对象开发在PHP里一直不是很受重视，并且没有得到普及和推广？PHP对面向对象的支持到底如何？怎么学习面向对象的思维？

在这里，我们将就面向对象一些概念展开讨论，其中重点讨论PHP特色的面向对象的风格和语法，并通过相互借鉴和对比，使读者认识PHP自身的特点，尤其是和其他语言中不同的地方。

1.1 面向对象的“形”与“本”

类是对象的抽象组织，对象是类的具体存在。

2200年前的战国时期，赵国平原君的食客公孙龙在骑着白马进城时，被守城官以马不能入城拦下，公孙龙即兴演讲，口述“白马非马”一论，守城官无法反驳，于是公孙龙就骑着他的白马（不是马的）进城去了。这就是历史上最经典的一次对面向对象思维的阐述。

公孙龙的“白马非马”论如下：

“白马非马”，可乎？曰：“可。”曰：“何哉？”曰：“马者，所以命形也；白者，所以命色也。命色者非命形也。故曰：‘白马非马’。”曰：“有白马不可谓无马也。不可谓无马者，非马也？有白马为有马，白之，非马何也？”曰：“求马，黄、黑马皆可致；求白马，黄、黑马不可致。使白马乃马也，是所求一也。所求一者，白者不异马也。所求不异，如黄、黑马有可有不可，何也？可与不可，其相非明。故黄、黑马一也，而可以应有马，而不可以应有白马，是白马之非马，审矣！”

公孙龙乃战国时期的“名家”，名家的中心论题是所谓“名”（概念）和“实”（存在）的逻辑关系问题。名者，抽象也，类也。实者，具体也，对象也。从这个角度讲，公孙龙是我国早期的最著名的面向对象思维的思想家。

“白马非马”这一论断的关键就在于“非”字，公孙龙一再强调白马与马的特征，通过把白马和马视为两个不同的类，用“非”这一关系，成功地把“白马”与“马”的关系由从属关系转移到“白马”这个对象与“马”这个对象的相等关系上，显然，二者不等，故“白马非马”。而我们正常的思维是，马是一个类，白马是马这个类的一个对象，二者属于从属关系。说“白马非马”，就是割裂马与白马之间的从属关系，偷换概念，故为诡辩也。

白马非马这个典故，我们可以称之为诡辩。但我们把这个问题抽象出来，实际上讨论的就是类与类之间的界定、类的定义等一系列问题，类应该抽象到什么程度，其中即涉及了类与对象的本质问题，也涉及了类的设计过程中的一些原则。

1.1.1 对象的“形”

要回答类与对象本质这个问题，我想可以先从“形”的角度来回答。本节以PHP为例，来探讨对象的“形”与“本”的问题。

类是我们对一组对象的描述。

在PHP里，每个类的定义都以关键字class开头，后面跟着类名，紧接着一对花括号，里面包含有类成员和方法的定义。如下面代码所示：

```
class person {
    public $ name;
    public $ gender;
    public function say () {
        echo $ this->name, "is", $ this->gender;
    }
}
```

在这里，我们定义了一个person类。代表了抽象出来的人这个概念，它含有姓名和性别这两个属性，还具有一个开口说话的方法，这个方法会告诉外界这个人的性别和姓名。我们接下来就可以产生这个类的实例：

```
$ student=new person ();
$ student->name= ' Tom ';
$ student->gender= ' male ';
$ student->say ();
$ teacher=new person ();
$ teacher->name= ' Kate ';
$ teacher->gender= ' female ';
```

```
$teacher->say ();
```

这段代码则实例化了person类，产生了一个student对象和teacher对象的实例。实际上也就是从抽象到具体的过程。现实世界中，仅仅说“人”是没有意义的，中国人把它叫“人”，美国人把它叫person或者human，如果高兴，把它叫“God”或者“板凳”都无所谓。但是只要你把“人”这个概念加上各种属性和方法，比如说有两条腿、直立行走、会说话，则无论是中国人，还是美国人，甚至外星人都是能理解你所描述的事物。所以，一个类的设计需要能充分展示其最重要的属性和方法，并且能与其他事物相区分。只有类本身有意义，从抽象到具体的实例化才会有意义。

根据上面的实例代码，可以有下面的一些理解：

类定义了一系列的属性和方法，并提供了实际的操作细节，这些方法可以用来对属性进行加工。

对象含有类属性的具体值，这就是类的实例化。正是由于属性的不同，才能区分不同的对象。在上面例子里，由于student和teacher的性别和姓名不一样，才得以区分开二人。

类与对象的关系类似一种服务与被服务、加工与被加工的关系，具体而言，就如同原材料与流水线的关系。只需要在对象上调用类中所存在的方法，就可以对类的属性进行加工，并且展示其功能。

类是属性和方法的集合，那么在PHP里，对象是什么呢？比较普遍的说法就是“对象由属性和方法组成”。对象是由属性组成，这很好理解，一个对象的属性是它区别于另一个对象的关键所在。由于PHP的对象是用数组来模拟的，因此我们把对象转为数组，就能看到这个对象所拥有的属性了。

继续使用上面代码，可以打印student对象：

```
print_r ((array)$student);  
var_dump ($student);
```

到这里，可以很直观地认识到，对象就是一堆数据。既然如此，可以把一个对象存储起来，以便需要时用。这就是对象的序列化。

所谓序列化，就是把保存在内存中的各种对象状态（属性）保存起来，并且在需要时可以还原出来。下面的代码实现了把内存中的对象当前状态保存到一个文件中：

```
$str=serialize ($ student );  
echo $ str;  
file _put _contents ( ' store.txt ', $ str );
```

输出序列化后的结果：

```
O: 6: "person": 2: { s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "mail"; }
```

在需要时，反序列化取出这个对象：

```
$str=file _get _contents ( ' store.txt ');  
$ student =unserialize ( $ str );  
$ student -> say ();
```

注意 在序列化和反序列化时都需要包含类的对象的定义，不然有可能出现在反序列化对象时，找不到该对象的类的定义，而返回不正确的结果。

可以看到，对象序列化后，存储的只是对象的属性。类是由属性和方法组成的，而对象则是属性的集合，由同一个类生成的不同对象，拥有各自不同的属性，但共享了类的代码空间中方法区域的代码。

1.1.2 对象的“本”

我们需要更深入地了解这种机制，看对象的“本”。对象是什么？对象在PHP中也是变量的一种，所以先看PHP源码中对变量的定义：

```
#zend/zend.h
typedef union _zvalue_value {
    long lval; /*long value*/
    double dval; /*double value*/
    struct {
        char*val;
        int len;
    } str;
    HashTable*ht; /*hash table value*/
    zend_object_value obj;
} zvalue_value;
```

zvalue_value，就是PHP底层的变量类型，zend_object_value obj就是变量中的一个结构。接着看对象的底层实现。

在PHP5中，对象在底层的实现是采取“属性数组+方法数组”来实现的。可以简单地理解为PHP对象在底层的存储如图1-1所示。



图 1-1 对象的组成

对象在PHP中是使用一种zend_object_value结构体来存储的。对象在ZEND（PHP底层引擎，类似Java的JVM）中的定义如下：

```
#zend/zend.h
```

```
typedef struct _zend_object {  
    zend_class_entry*ce; //这里就是类入口  
    HashTable*properties; //属性组成的HashTable  
    HashTable*guards; /*protects from get/set.....recursion*/  
} zend_object;
```

ce是存储该对象的类结构，在对象初始化时保存了类的入口，相当于类指针的作用。properties是一个HashTable，用来存放对象属性。guards用来阻止递归调用。

类的标准方法在zend/zend_object_handlers.h文件中定义，具体实现则是在zend/zend_

object_handlers.c文件中。关于PHP变量的存储结构的底层实现，将在第7章中进行更深入的介绍。

通过对上述源代码的简单阅读，可以更清晰地认识到对象也是一种很普通的变量，不同的是其携带了对象的属性和类的入口。

1.1.3 对象与数组

对象是什么，我们不好理解，也不容易回答，但是我们知道数组是什么。数组的概念比较简单。可以拿数组和对象对比来帮助我们理解对象。对象转化为数组，数组也能转换成对象。数组是由键值对数据组成的，数组的键值对和对象的属性/属性值对十分相似。对象序列化后和数组序列化后的结果是惊人的相似。如下面的代码所示：

```
$student_arr=array('name' => 'Tom', 'gender' => 'male');  
echo "\n";  
echo serialize($student_arr);
```

输出为：

```
a: 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male";}
```

可以很清楚地看出，对象和数组在内容上一模一样！

而对象和数组的区别在于：对象还有个指针，指向了它所属的类。在对student对象序列化时，我们看到了“person”这几个字符，这个标识符就标志了这个对象归属于person类，故在取出这个对象后，可以立即对其执行所包含的方法。如果对象中还包含对象呢？我们来看下一节的内容。

1.1.4 对象与类

在前面代码中定义了一个类，并创建了这个类的对象，把前面产生的对象作为这个新对象的一个属性，完整代码如代码清单1-1所示。

代码清单1-1 object.php

```
<? php
class person {
public $ name;
public $ gender;
public function say () {
echo $ this->name, " \ tis", $ this->gender, " \ r \ n";
}
}
class family {
public $ people;
public $ location;
public function construct ( $ p, $ loc ) {
$ this->people= $ p;
$ this->location= $ loc;
}
}
$ student=new person ();
$ student->name= ' Tom ';
$ student->gender= ' male ';
$ student->say ();
$ tom=new family ( $ student, ' peking ');
echo serialize ( $ student );
$ student_ arr=array ( ' name ' => ' Tom ', ' gender ' => ' male ');
echo" \ n";
echo serialize ( $ student_ arr );
print_ r ( $ tom );
echo" \ n";
echo serialize ( $ tom );
```

输出结果如下:

```
Tom is male
O: 6: "person": 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male";}
a: 2: {s: 4: "name"; s: 3: "Tom"; s: 6: "gender"; s: 4: "male";}
family Object
(
[people]=> person Object
(
[name]=> Tom
[gender]=> male
)
[location]=> peking
)
O: 6: "family": 2: {s: 6: "people"; O: 6: "person": 2: {s: 4: "name"; s: 3: "Tom"; s:
6: "gender"; s: 4: "male";} s: 8: "
location"; s: 6: "peking";}
```

可以看出，序列化后的对象会附带所属的类名，这个类名保证此对象能够在执行类的方法（也是自己所能执行的方法）时，可以正确地找到方法所在的代码空间（即对象所拥有的方法存储在类里）。另外，当一个对象的实例变量引用其他对象时，序列化该对象时也会对引用对象进行序列化。

基于如上的分析，可以总结出对象和类的概念以及二者之间的关系：

类是定义一系列属性和操作的模板，而对象则把属性进行具体化，然后交给类处理。

对象就是数据，对象本身不包含方法。但是对象有一个“指针”指向一个类，这个类里可以有方法。

方法描述不同属性所导致的不同表现。

类和对象是不可分割的，有对象就必定有一个类和其对应，否则这个对象也就成了没有亲人的孩子（但有一个特殊情况存在，就是由标量进行强制类型转换的object，没有一

个类和它对应。此时，PHP中一个称为“孤儿”的stdClass类就会收留这个对象)。

理解了以上四个概念，结合现实世界从实现和存储理解对象和类，这样就不会把二者看成一个抽象、神秘的东西，也就能写出符合现实世界的类了。

如果需要一个类，要从客观世界抽象出一套规律，就得总结这类事物的共性，并且让它可以与其他类进行区分。而这个区分的依据就是属性和方法。区分的办法就是实例化出一个对象，是骡子是马，拉出来遛遛。

现在，你是否对“白马非马”这个典故有了新的认识？

1.2 魔术方法的应用

魔术方法是以两个下划线“`__`”开头、具有特殊作用的一些方法，可以看做PHP的“语法糖”。

语法糖指那些没有给计算机语言添加新功能，而只是对人类来说更“甜蜜”的语法。语法糖往往给程序员提供了更实用的编码方式或者一些技巧性的用法，有益于更好的编码风格，使代码更易读。不过其并没有给语言添加什么新东西。PHP里的引用、SPL等都属于语法糖。

实际上，在1.1节代码中就涉及魔术方法的使用。family类中的construct方法就是一个标准魔术方法。这个魔术方法又称构造方法。具有构造方法的类会在每次创建对象时先调用此方法，所以非常适合在使用对象之前做一些初始化工作。因此，这个方法往往用于类进行初始化时执行一些初始化操作，如给属性赋值、连接数据库等。

以代码清单1-1所示代码为例，family中的construct方法主要做的事情就是在创建对象的同时对属性赋值。也可以这么使用：

```
$tom=new family ($student, 'peking ');  
$tom->people->say ();
```

这样做就不需要在创建对象后再去赋值了。有构造方法就有对应的析构方法，即destruct方法，析构方法会在某个对象的所有引用都被删除，或者当对象被显式销毁时执行。这两个方法是常见也是最有用的魔术方法。

1.2.1 set和get方法

set和get是两个比较重要的魔术方法，如代码清单1-2所示。

代码清单1-2 magic.php

```
<? php
class Account {
private $ user=1;
private $ pwd=2;
}
$a=new Account ();
echo $a->user;
$a->name=5;
echo $a->name;
echo $a->big;
```

运行这段代码会怎样呢？结果报错如下：

```
Fatal error: Cannot access private property Account: $user in G: \ bak \ temp \ tempcode \
sg.php on line 7
```

所报错误大致是说，不能访问Account对象的私有属性user。在代码清单1-2的类定义里增加以下代码，其中使用了set魔术方法。

```
public function set ($ name,$ value ) {
echo"Setting $ name to $ value \ r \ n";
$this-> $ name= $ value;
}
public function get ($ name ) {
if (! isset ( $ this-> $ name )) {
echo ' 未设置 ';
$this-> $ name="正在为你设置默认值";
}
return $ this-> $ name;
}
```

再次运行，看到正常输出，没有报错。在类里以两个下画线开头的方法都属于魔术方

法（除非是你自定义的），它们是PHP中的内置方法，有特殊含义。手册里把这两个方法归到重载。

PHP的重载和Java等语言的重载不同。Java里，重载指一个类中可以定义参数列表不同但名字相同的多个方法。比如，Java也有构造函数，Java允许有多个构造函数，只要保证方法签名不一样就行；而PHP则在一个类中只允许有一个构造函数。

PHP提供的“重载”指动态地“创建”类属性和方法。因此，set和get方法被归到重载里。

这里可以直观看到，若类中定义了set和get这一对魔术方法，那么当给对象属性赋值或者取值时，即使这个属性不存在，也不会报错，一定程度上增强了程序的健壮性。

我们注意到，在account类里，user属性的访问权限是私有的，私有属性意味着这个属性是类的“私有财产”，只能在类内部对其进行操作。如果没有set这个魔术方法，直接在类的外部对属性进行赋值操作是会报错的，只能通过在类中定义一个public的方法，然后在类外调用这个公开的方法进行属性读写操作。

现在有了这两个魔术方法，是不是对私有属性的操作变得更方便了呢？实际上，并没有什么奇怪的，因为这两个方法本身就是public的。它们和在对外的public方法中操作private属性的原理一样。只不过这对魔术方法使其操作更简单，不需要显式地调用一个public的方法，因为这对魔术方法在操作类变量时是自动调用的。当然，也可以把类属性定义成public的，这样就可以随意在类的外部进行读写。不过，如果只是为了方便，类属性在任意时候都定义成public权限显然是不合适的，也不符合面向对象的设计思想。

1.2.2 call和callStatic方法

如何防止调用不存在的方法而出错？一样的道理，使用call魔术重载方法。

call方法原型如下：

```
mixed call ( string $name, array $arguments )
```

当调用一个不可访问的方法（如未定义，或者不可见）时，call（）会被调用。其中name参数是要调用的方法名称。arguments参数是一个数组，包含着要传递给方法的参数，如下所示：

```
public function call ( $name, $arguments ) {  
    switch ( count ( $arguments ) ) {  
        case 2:  
            echo $arguments[0]* $arguments[1], PHP_EOL;  
            break;  
        case 3:  
            echo array_sum ( $arguments ), PHP_EOL;  
            break;  
        default:  
            echo ' 参数不对 ', PHP_EOL;  
            break;  
    }  
}  
$a->make ( 5 );  
$a->make ( 5, 6 );
```

以上代码模拟了类似其他语言中的根据参数类型进行重载。跟call方法配套的魔术方法是callStatic。当然，使用魔术方法“防止调用不存在的方法而报错”，并不是魔术方法的本意。实际上，魔术方法使方法的动态创建变为可能，这在MVC等框架设计中是很有

用的语法。假设一个控制器调用了不存在的方法，那么只要定义了call魔术方法，就能友好地处理这种情况。

试着理解代码清单1-3所示代码。这段代码通过使用callStatic这一魔术方法进行方法的动态创建和延迟绑定，实现一个简单的ORM模型。

代码清单1-3 simpleOrm.php

```
<php
abstract class ActiveRecord {
protected static $ table;
protected $ fieldvalues;
public $ select;
static function findById ( $ id ) {
    $ query="select*from"
    .static: $ table
    ."where id= $ id";
return self: createDomain ( $ query );
}
function get ( $ fieldname ) {
return $ this->fieldvalues[ $ fieldname];
}
static function callStatic ( $ method,$ args ) {
    $ field=preg_ replace ( ' / ^ findBy ( \ w * ) $ / ' , ' $ { 1 } ' , $ method );
    $ query="select*from"
    .static: $ table
    ."where $ field= ' $ args[0] ' ";
return self: createDomain ( $ query );
}
private static function createDomain ( $ query ) {
    $ klass=get _ called _ class ( );
    $ domain=new $ klass ( );
    $ domain->fieldvalues=array ( );
    $ domain->select= $ query;
foreach ( $ klass: $ fields as $ field=> $ type ) {
    $ domain->fieldvalues[ $ field]= ' TODO: set from sql result ' ;
}
}
```

```
return $domain;
}
}
class Customer extends ActiveRecord {
protected static $table= ' custdb ';
protected static $fields=array (

    ' id ' => ' int ',
    ' email ' => ' varchar ',
    ' lastname ' => ' varchar '
);
}
class Sales extends ActiveRecord {
protected static $table= ' salesdb ';
protected static $fields=array (
    ' id ' => ' int ',
    ' item ' => ' varchar ',
    ' qty ' => ' int '
);
}
assert ( "select*from custdb where id=123"==
Customer: findById ( 123 ) -> select );
assert ( "TODO: set from sql result"==
Customer: findById ( 123 ) -> email );
assert ( "select*from salesdb where id=321"==
Sales: findById ( 321 ) -> select );
assert ( "select*from custdb where Lastname= ' Denoncourt ' "==
Customer: findByLastname ( ' Denoncourt ' ) -> select );
```

再举个类似的例子。PHP里有很多字符串函数，假如要先过滤字符串首尾的空格，再求出字符串的长度，一般会这么写：

```
strlen ( trim ( $str ) );
```

如果要实现JS里的链式操作，比如像下面这样，应该怎么实现？

```
$str->trim()->strlen()
```

很简单，先实现一个String类，对这个类的对象调用方法进行处理时，触发call魔术方法，接着执行call_user_func即可。

1.2.3 toString方法

再看另外一个魔术方法__toString（在这里故意这么写，是要说明PHP中方法不区分大小写，但实际开发中还需要注意规范）。

当进行测试时，需要知道是否得出正确的数据。比如打印一个对象时，看看这个对象都有哪些属性，其值是什么，如果类定义了toString方法，就能在测试时，echo打印对象体，对象就会自动调用它所属类定义的toString方法，格式化输出这个对象所包含的数据。如果没有这个方法，那么echo一个对象将报错，例如“Catchable fatal error: Object of class Account could not be converted to string”语法错误，实际上这是一个类型匹配失败错误。不过仍然可以用print_r()和var_dump()函数输出一个对象。当然，toString是可以定制的，所提供的信息和样式更丰富，如代码清单1-4所示。

代码清单1-4 magic_2.php

```
<? php
class Account {
public $user=1; private $pwd=2;
//自定义的格式化输出方法
public function toString () {
return"当前对象的用户名是 {$this->user}，密码是 {$this->pwd} ";
}
}
$a=new Account (); echo $a;
echo PHP_EOL;
print_r ($a);
```

运行这段代码发现，使用toString方法后，输出的结果是可定制的，更易于理解。实际上，PHP的toString魔术方法的设计原型来源于Java。Java中也有这么一个方法，而且在Java中，这个方法被大量使用，对于调试程序比较方便。实际上，toString方法也是一种序列化，我们知道PHP自带serialize/unserialize也是进行序列化的，但是这组函数序列化时会产生一些无用信息，如属性字符串长度，造成存储空间的无谓浪费。因

此，可以实现自己的序列化和反序列化方法，或者json_encode/json_decode也是一个不错的选择为什么直接echo一个对象就会报语法错误，而如果这个对象实现toString方法后就可以直接输出呢？原因很简单，echo本来可以打印一个对象，而且也实现了这个接口，但是PHP对其做了个限制，只有实现toString后才允许使用。从下面的PHP源代码里可以得到验证：

```
ZEND_VM_HANDLER ( 40, ZEND_ECHO, CONST | TMP | VAR | CV, ANY )
{
    zend_op*opline=EX ( opline );
    zend_free_op free_op1;
    zval z_copy;
    zval*z=GET_OP1_ZVAL_PTR ( BP_VAR_R );
    //此处的代码预留了把对象转换为字符串的接口
    if ( OP1_TYPE! =IS_CONST&&
        Z_TYPE_P ( z ) ==IS_OBJECT&&Z_OBJ_HT_P ( z ) ->get_method! =NULL&&
        zend_std_cast_object_tostring ( z, &z_copy, IS_STRING TSRMLS_CC ) ==SUCCESS )
    {
        zend_print_variable ( &z_copy );
        zval_dtor ( &z_copy );
    } else {
        zend_print_variable ( z );
    }
    FREE_OP1 ();
    ZEND_VM_NEXT_OPCODE ();
}
```

由此可见，魔术方法并不神奇。

有比较才有认知。最后，针对本节代码给出一个Java版本的代码，供各位读者用来对比两种语言中重载和魔术方法的异同，如代码清单1-5所示。

代码清单1-5 Account.java

```
import org.apache.commons.lang3.builder.ToStringBuilder;
```

```
/**
 *类的重载演示Java版本
 *@author wfox
 *@date@verson
 */
public class Account {
    private String user; //用户名
    private String pwd; //密码
    public Account () {
        System.out.println ("构造函数");
    }
    public Account ( String user, String pwd ) {
        System.out.println ("重载构造函数");
        System.out.println ( user+"——"+pwd );
    }
    public void say ( String user ) {
        System.out.println ("用户是: "+user );
    }
    public void say ( String user, String pwd ) {
        System.out.println ("用户: "+user );
        System.out.println ("密码"+pwd );
    }
    public String getUser () {
        return user;
    }
    public void setUser ( String user ) {
        this.user=user;
    }
    public String getPwd () {
        return pwd;
    }
    public void setPwd ( String pwd ) {
    }
    @Override
    public String toString () {
        return ToStringBuilder.reflectionToString ( this );
    }
    public static void main ( String…… ) {

        Account account=new Account ();
```

```
account.setUser ("张三");  
account.setPwd ("123456");  
account.say ("李四");  
account.say ("王五", "123");  
System.out.println ( account );  
}  
}
```

运行上述代码，输出如图1-2所示。

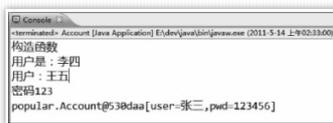


图 1-2 Java里的构造方法和重载演示

可以看出，Java的构造方法比PHP好用，PHP由于有了set/get这一对魔术方法，使得动态增加对象的属性字段变得很方便，而对Java来说，要实现类似的效果，就不得不借助反射API或直接修改编译后字节码的方式来实现。这体现了动态语言的优势，简单、灵活。

1.3 继承与多态

面向对象的优势在于类的复用。继承与多态都是对类进行复用，它们一个是类级别的复用，一个是方法级别的复用。提到继承必提组合，二者有何异同？PHP到底有没有多态？若没有，则为什么没有？有的话，和其他语言中的多态又有什么区别？这些都是本节所要讲述的内容。

1.3.1 类的组合与继承

在1.1节的代码中定义了两个类，一个是person，一个是family；在family类中创建person类中的对象，把这个对象视为family类的一个属性，并调用它的方法处理问题，这种复用方式叫“组合”。还有一种复用方式，就是继承。

类与类之间有一种父与子的关系，子类继承父类的属性和方法，称为继承。在继承里，子类拥有父类的方法和属性，同时子类也可以有自己的方法和属性。

可以把1.1节的组合用继承实现，如代码清单1-6所示。

代码清单1-6 family_extends.php

```
<? php
class person {
public $ name= ' Tom '; public $ gender;
static $ money=10000;
public function construct () {
echo ' 这里是父类 ', PHP _EOL;
}
public function say () {
echo $ this-> name, " \ tis", $ this-> gender, " \ r \ n";
}
}
class family extends person {
public $ name; public $ gender; public $ age;
static $ money=100000;
```

```
public function construct () {
parent: construct (); //调用父类构造方法
echo ' 这里是子类 ', PHP_EOL;
}
public function say () {
parent: say ();
echo $this->name, " \ tis \ t", $this->gender, ", and
is \ t", $this->age, PHP_EOL;
}
public function cry () {
echo parent: $money, PHP_EOL;
echo ' %> _ <% ', PHP_EOL;
echo self: $money, PHP_EOL; //调用自身构造方法
echo ' (*^ ^*) ';
}
}
$poor=new family ();
$poor->name= ' Lee ';
$poor->gender= ' female ';
$poor->age=25;
$poor->say ();
$poor->cry ();
```

运行上面的代码，可以得到如下输出结果：

```
这里是父类
这里是子类
Lee is female
Lee is female, and is 25
10000
%> _ <%
100000
(*^ ^*)
```

从上面代码中可以了解继承的实现。在继承中，用parent指代父类，用self指代自

身。使用“:”运算符（范围解析操作符）调用父类的方法。“:”操作符还用来作为类常量和静态方法的调用，不要把这两种应用混淆。

既然提到静态，就再强调一点，如果声明类成员或方法为static，就可以不实例化类而直接访问，同时也就不能通过一个对象访问其中的静态成员（静态方法除外），也不能用“:”访问一个非静态方法。比如，把上例中的`poor->cry()`；换成`poor:cry()`，按照这个规则，应该是要报错的。可能试验时，并没有报错，而且能够正确输出。这是因为用“:”方式调用一个非静态方法会导致一个E_STRICT级别的错误，而这里的PHP设置默认没有开启这个级别的报错提示。打开PHP安装目录下的php.ini文件，设置如下：

```
error_reporting=E_ALL | E_STRICT display_errors=On
```

再次运行，就会看到错误提示。因此，用“:”访问一个非静态方法不符合语法，但PHP仍然能够正确地执行代码，这只是PHP所做的一个“兼容”或者说“让步”。在开发时，设置最严格的报错等级，在部署时可适当调低。

组合与继承都是提高代码可重用性的手段。在设计对象模型时，可以按照语义识别类之间的组合关系和继承关系。比如，通过一些总结，得出了继承是一种“是、像”的关系，而组合是一种“需要”的关系。利用这条规律，就可以很简单地判断出父亲与儿子应该是继承关系，父亲与家庭应该是组合关系。还可以从另外一个角度看，组合偏重整体与局部的关系，而继承偏重父与子的关系，如图1-3所示。



图 1-3 继承和组合的对照

从方法复用角度考虑，如果两个类具有很多相同的代码和方法，可以从这两个类中抽象出一个父类，提供公共方法，然后两个类作为子类，提供个性方法。这时用继承语意更好。继承的UML图如图1-4所示。

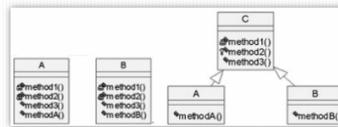


图 1-4 继承的UML图

而组合就没有这么多限制。组合之间的类可以关系（体现为复用代码）很小，甚至没有关系，如图1-5所示。

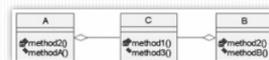


图 1-5 组合的UML图

然而在编程中，继承与组合的取舍往往并不是这么直接明了，很难说出二者是“像”的关系还是“需要”的关系，甚至把它拿到现实世界中建模，还是无法决定应该是继承还是组合。那应该怎么办呢？有什么标准吗？有的。这个标准就是“低耦合”。

耦合是一个软件结构内不同模块之间互连程度的度量，也就是不同模块之间的依赖关系。

低耦合指模块与模块之间，尽可能地使模块间独立存在；模块与模块之间的接口尽量少而简单。现代的面向对象的思想不强调为真实世界建模，变得更加理性化一些，把目标放在解耦上。

解耦是要解除模块与模块之间的依赖。

按照这个思想，继承与组合二者语义上难于区分，在二者均可使用的情况下，更倾向于使用组合。为什么呢？继承存在什么问题呢？

1) 继承破坏封装性。

比如，定义鸟类为父类，具有羽毛属性和飞翔方法，其子类天鹅、鸭子、鸵鸟等继承鸟这个类。显然，鸭子和鸵鸟不需要飞翔这个方法，但作为子类，它们却可以无区别地使用飞翔这个方法，显然破坏了类的封装性。而组合，从语义上来说，要优于继承。

2) 继承是紧耦合的。

继承使得子类和父类捆绑在一起。组合仅通过唯一接口和外部进行通信，耦合度低于继承。

3) 继承扩展复杂。

随着继承层数的增加和子类的增加，将涉及大量方法重写。使用组合，可以根据类型约束，实现动态组合，减少代码。

4) 不恰当地使用继承可能违反现实世界中的逻辑。

比如，人作为父类，雇员、经理、学生作为子类，可能存在这样的问题，经理一定是雇员，学生也可能是雇员，而使用继承的话一个人就无法拥有多个角色。这种问题归结起来就是“角色”和“权限”问题。在权限系统中很可能存在这样的问题，经理权利和职位大于主管，但出于分工和安全的考虑，经理没有权限直接操作主管所负责的资源，技术部经理也没权限直接命令市场部主管。这就要求角色和权限系统的设计要更灵活。不恰当的继承可能导致逻辑混乱，而使用组合就可以较好地解决这个问题。

当然，组合并非没有缺点。在创建组合对象时，组合需要一一创建局部对象，这一定程度上增加了一些代码，而继承则不需要这一步，因为子类自动有了父类的方法，如代码清单1-7所示。

代码清单1-7 mobile.php

```
<? php
//继承拥有比组合更少的代码量
class car {
public function addoil () {
echo"Add oil \ r \ n";
}
}
class bmw extends car {
}
class benz {
public $ car;
public function construct () {
```

```
$ this->car=new car;
}
public function addoil () {
$ this->car->addoil ();
}
}
$ bmw=new bmw;
$ bmw->addoil ();
$ benz=new benz ();
$ benz->addoil ();
```

显然，组合比继承增加了代码量。组合还有其他的一些缺点，不过总体说来，是优点大于缺点。

继承最大的优点就是扩展简单，但是其缺点大于优点，所以在设计时，需要慎重考虑。那应该如何使用继承呢？

精心设计专门用于被继承的类，继承树的抽象层应该比较稳定，一般不要多于三层。

对于不是专门用于被继承的类，禁止其被继承，也就是使用final修饰符。使用final修饰符既可防止重要方法被非法复写，又能给编辑器寻找优化的机会。

优先考虑用组合关系提高代码的可重用性。

子类是一种特殊的类型，而不只是父类的一个角色。

子类扩展，而不是复盖或者使父类的功能失效。

底层代码多用组合，顶层/业务层代码多用继承。底层用组合可以提高效率，避免对象臃肿。顶层代码用继承可以提高灵活性，让业务使用更方便。

思考题 设计一个log类，需要用到MySQL中的CURD操作，是应该使用继承呢还是组合？请给出理由。

继承并非一无是处，而组合也不是完美无缺的。如果既要组合的灵活，又要继承的代码简洁，能做到吗？

这是可以做到的，譬如多重继承，就具有这个特性。多重继承里一个类可以同时继承多个父类，组合两个父类的功能。C++里就是使用的这种模型来增强继承的灵活性的，但是多重继承过于灵活，并且会带来“菱形问题”，故为其使用带来了不少困难，模型变得复杂起来，因此在大多数语言中，都放弃了多重继承这一模型。

多重继承太复杂，那么还有其他方式能比较好地解决这个问题吗？PHP5.4引入的新的语法结构Traits就是一种很好的解决方案。Traits的思想来源于C++和Ruby里的Mixin以及Scala里的Traits，可以方便我们实现对象的扩展，是除extend、implements外的另外一种扩展对象的方式。Traits既可以使单继承模式的语言获得多重继承的灵活，又可以避免多重继承带来的种种问题。

1.3.2 各种语言中的多态

多态确切的含义是：同一类的对象收到相同消息时，会得到不同的结果。而这个消息是不可预测的。多态，顾名思义，就是多种状态，也就是多种结果。

以Java为例，由于Java是强类型语言，因此变量和函数返回值是有状态的。比如，实现一个add函数的功能，其参数可能是两个int型整数，也可能是两个float型浮点数，而返回值可能是整型或者浮点型。在这种情况下，add函数是有状态的，它有多种可能的运行结果。在实际使用时，编译器会自动匹配适合的那个函数。这属于函数重载的概念。需要说明的是，重载并不是面向对象里的东西，和多态也不是一个概念，它属于多态的一种表现形式。

多态性是一种通过多种状态或阶段描述相同对象的编程方式。它的真正意义在于：实际开发中，只要关心一个接口或基类的编程，而不必关心一个对象所属于的具体类。

很多地方会看到“PHP没有多态”这种说法。事实上，不是它没有，而是它本来就是多态的。PHP作为一门脚本语言，自身就是多态的，所以在语言这个级别上，不谈PHP的多态。在PHP官方手册也找不到对多态的详细描述。

既然说PHP没有多态这个概念（实际上是不需要多态这个概念），那为什么又要讲多态呢？可以看下面的例子，如代码清单1-8所示。

代码清单1-8 Polymorphism.php

```
<? php
class employee {
protected function working () {
echo ' 本方法需重载才能运行 ';
}
}
class teacher extends employee {
public function working () {
echo ' 教书 ';
}
}
```

```
}
class coder extends employee {
public function working () {
echo ' 敲代码 ';
}
}
function dprint ($obj) {
if ( get_class ($obj) == ' employee ') {
echo ' Error ';
} else {
$obj->working ();
}
}
dprint ( new teacher ());
dprint ( new coder ());
dprint ( new employee ());
```

通过判断传入的对象所属的类不同来调用其同名方法，得出不同结果，这是多态吗？如果站在C++角度，这不是多态，这只是不同类对象的不同表现而已。C++里的多态指运行时对象的具体化，指同一类对象调用相同的方法而返回不同的结果。看个C++的例子，如代码清单1-9所示。

代码清单1-9 C++多态的例子

```
#include <cstdlib>
#include <iostream>
/**
C++中用虚函数实现多态
*/
using namespace std;
class father {
public:
father (): age ( 30 ) { cout << "父类构造法, 年龄" << age << "\ n"; }
~father () { cout << "父类析构" << "\ n"; }
void eat () { cout << "父类吃饭吃三斤" << "\ n"; }
virtual void run () { cout << "父类跑10000米" << "\ n"; } //虚函数
```

```
protected:
int age;
};
class son: public father {
public:
son () { cout<<"子类构造法"<<"\n";}
~son () { cout<<"子类析构"<<"\n";}
void eat () { cout<<"儿子吃饭吃一斤"<<"\n";}
void run () { cout<<"儿子跑100米"<<"\n";}
void cry () { cout<<"哭泣"<<"\n";}
};
int main ( int argc, char*argv[])
{
father*pf=new son;
pf->eat ();
pf->run ();
delete pf;
system ("PAUSE");
return EXIT _SUCCESS;
}
```

上面的代码首先定义一个父类，然后定义一个子类，这个子类继承父类的一些方法并且有自己的方法。通过`father*pf=new son;`语句创建一个派生类（子类）对象，并且把该派生类对象赋给基类（父类）指针，然后用该指针访问父类中的`eat`和`run`方法。图1-6所示是运行结果。

由于父类中的`run`方法加了`virtual`关键字，表示该函数有多种形态，可能被多个对象所拥有。也就是说，多个对象在调用同一名字的函数时会产生不同的效果。

这个例子和PHP的例子有什么不同呢？C++的这个例子所创建的对象是一个指向父类的子对象，还可以创建更多派生类对象，然后上转型为父类对象。这些对象，都是同一类对象，但是在运行时，却都能调用到派生类同名函数。而PHP中的例子则是不同类的对象调用。



图 1-6 运行结果

由于PHP是弱类型的，并且也没有对象转型机制，所以不能像C++或者Java那样实现`father $pf=new son`；把派生类对象赋给基类对象，然后在调用函数时动态改变其指向。在PHP的例子中，对象都是确定的，是不同类的对象。所以，从这个角度讲，这还不是真正的多态。

代码清单1-8所示代码通过判断对象的类属性实现“多态”，此外，还可以通过接口实现多态，如代码清单1-10所示。

代码清单1-10 通过接口实现多态

```
<? php
interface employee {
public function working ();
}
class teacher implements employee {
public function working () {
echo '教书';
}
}
class coder implements employee {
public function working () {
echo '敲代码';
}
}
function doprint (employee $i) {
$i->working ();
}
$a=new teacher;
$b=new coder;
doprint ($a);
doprint ($b);
```

这是多态吗？这段代码和代码清单1-8相比没有多少区别，不过这段代码中`doprint`

函数的参数是一个接口类型的变量，符合“同一类型，不同结果”这一条件，具有多态性的一般特征。因此，这是多态。

如果把代码清单1-8中doprint函数的obj参数看做一种类型（把所有弱类型看做一种类型），那就可以认为代码清单1-8中的代码也是一种多态。

再次把三段代码放在一起品味，可以看出：区别是否是多态的关键在于看对象是否属于同一类型。如果把它们看做同一种类型，调用相同的函数，返回了不同的结果，那么它就是多态；否则，不能称其为多态。由此可见，弱类型的PHP里多态和传统强类型语言里的多态在实现和概念上是有一些区别的，而且弱类型语言实现起多态来会更简单，更灵活。

本节解决了什么是多态，什么不是多态的问题。至于多态是怎么实现的，各种语言的策略是不一样的。但是，最终的实现无非就是查表和判断。总结如下：

多态指同一类对象在运行时的具体化。

PHP语言是弱类型的，实现多态更简单、更灵活。

类型转换不是多态。

PHP中父类和子类看做“继父”和“继子”关系，它们存在继承关系，但不存在血缘关系。因此子类无法向上转型为父类，从而失去多态最典型的特征。

多态的本质就是if……else，只不过实现的层级不同。

1.4 面向接口编程

这里，首先强调一个概念，面向接口编程并不是一种新的编程范式。本章开头提到的三大范式中并没有提到面向接口。其次，这里是狭义的接口，即interface关键字。广义的接口可以是任何一个对外提供服务的出口，比如提供数据传输的USB接口、淘宝网对其他网站开放的支付宝接口。

1.4.1 接口的作用

接口定义一套规范，描述一个“物”的功能，要求如果现实中的“物”想成为可用，就必须实现这些基本功能。接口这样描述自己：

“对于实现我的所有类，看起来都应该像我现在这个样子”。

采用一个特定接口的所有代码都知道对于那个接口会调用什么方法。这便是接口的全部含义。接口常用来作为类与类之间的一个“协议”。接口是抽象类的变体，接口中所有方法都是抽象的，没有一个有程序体。接口除了可以包含方法外，还能包含常量。

比如用接口描述发动机，要求机动车必须要有“run”功能，至于怎么实现（摩托还是宝马），应该是什么样（前驱还是后驱），不是接口关心的。因为接口为抽象而生。作为质检总局，要判断这辆车是否合格，只要按“接口”的定义一条一条验证，这辆车不能“run”，那它就是废品，不能通过验收。但是，如果汽车实现了接口中本来不存在的方法music，并不认为有什么问题。接口就是一种契约。因此，在程序里，接口的方法必须被全部实现，否则将报fatal错误，如代码清单1-11所示。

代码清单1-11 interface.php

```
<? php
interface mobile
{
public function run (); //驱动方法
}
class plain implements mobile
```

```
{
public function run ()
{
echo"我是飞机";
}
public function fly ()
{
echo"飞行";
}
}
class car implements mobile {
public function run () {
echo"我是汽车 \r \n";
}
}
class machine
{
function demo ( mobile $a )
{
$a->fly (); //mobile接口是没有这个方法的
}
}
$obj=new machine ();
$obj->demo ( new plain ()); //运行成功
$obj->demo ( new car ()); //运行失败
```

在这段代码里，定义一个机动车接口，其中含有一个发动机功能。然后用一个飞机类实现这个接口，并增加了飞行方法。最后，在一个机械检测类中对机动车进行测试（用类型约束指定要测试的是机动车这个接口）。但是，此检测线测试的却是机动车接口中不存在的fly方法，直到遇到car的实例因不存在fly方法而报错。

这段代码实际上是错误的，不符合接口语义。但是在PHP里，对plain的实例进行检测时是可以运行的。也就是说，在PHP里，只关心是否实现这个方法，而并不关心接口语义是否正确。

按理说，接口应该是起一个强制规范和契约的作用，但是这里对接口的约束并没有起

效，也打破了契约，对检测站这个类的行为失去控制。可以看看在Java里是怎么处理的，如图1-7所示。

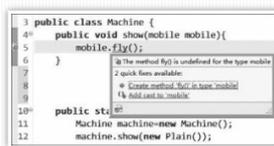


图 1-7 Java中接口是一种类型

Java认为，接口就是一种type，即类型。如果你打破了我们之间的契约，你的行为变得无法控制，那就是非法的。这符合逻辑，也符合现实世界。这就真正起到接口作为规范的作用了。

接口不仅规范接口的实现者，还规范接口的执行者，不允许调用接口中本不存在的方法。当然这并不是说一个类如果实现了接口，就只能实现接口中才有的方法，而是说，如果针对的是接口，而不是具体的类，则只能按接口的约定办事。这样的语法规定对接口的使用是有利的，让程序更健壮。根据这个角度讲，为了保证接口的语义，通常一个接口的实现类仅实现该接口所具有的方法，做到专一，当然这也不是一成不变的。

由上面的例子可以看出，PHP里接口作为规范和契约的作用打了折扣。上面例子实际就是一个典型面向接口编程的例子。根据这个例子，可以很自然地想到使用接口的场合，比如数据库操作、缓存实现等。不用关心我们所面对的数据库是MySQL还是Oracle，只需要关心面向Database接口进行具体业务的逻辑相关的代码，这就是面向接口编程的来历。

在这里，Database就如同employee一样，针对这个接口实现就好了。缓存功能也一样，我们不关注缓存是内存缓存还是文件缓存，或者是数据库缓存，只关注它是否实现了Cache接口，并且它只要实现了Cache接口，就实现了写入缓存和读取某条缓存中的数据及清除缓存这几个关键的功能点。

通常在大型项目里，会把代码进行分层和分工。核心开发人员和技术经理编写核心的流程和代码，往往是以接口的形式给出，而基础开发人员则针对这些接口，填充代码，如数据库操作等。这样，核心技术人员把更多精力投入到了技术攻关和业务逻辑中。前端针对接口编程，只管在Action层调用Service，不管实现细节；而后端则要负责Dao和

Service层接口实现。这样，就实现了代码分工与合作。

1.4.2 对PHP接口的思考

PHP的接口自始至终一直在被争议，有人说接口很好，有人说接口像鸡肋。首先要明白，好和不好的判断标准是什么。无疑，这是和Java/C++相比。在上面的例子中，已经讨论了PHP的接口在“面向契约编程”中是不足的，并没有起到应有的作用。

其实，在上面的interface.php代码中，machine类的声明应该在plain类前面。接口提供了一套规范，这是系统提供的，然后machine类提供一组针对接口的API并实现，最后才是自定义的类。在Java里，接口之所以盛行（多线程的runable接口、容器的collection接口等）就是因为系统为我们做了前面两部分的工作，而程序员，只需要去写具体的实现类，就能保证接口可用可控。

为什么要用接口？接口到底有什么好处？接口本身并不提供实现，只是提供一个规范。如果我们知道一个类实现了某个接口，那么就知道了可以调用该接口的哪些方法，我们只需要知道这些就够了。

PHP中，接口的语义是有限的，使用接口的地方并不多，PHP中接口可以淡化为设计文档，起到一个团队基本契约的作用，代码清单1-12所示。

代码清单1-12 cache_imp.php

```
<? php
interface cache {
/**
@describe: 缓存管理，项目经理定义接口，技术人员负责实现
**/

const maxKey=10000; //最大缓存量
public function getc ( $ key ); //获取缓存
public function setc ( $ key,$ value ); //设置缓存
public function flush ( ); //清空缓存
}
```

由于PHP是弱类型，且强调灵活，所以并不推荐大规模使用接口，而是仅在部分“内核”代码中使用接口，因为PHP中的接口已经失去很多接口应该具有的语义。从语义上考虑，可以更多地使用抽象类。至于抽象类和接口的比较，不再赘述。

另外，PHP5对面向对象的特性做了许多增强，其中就有一个SPL（标准PHP库）的尝试。SPL中实现一些接口，其中最主要的就是Iterator迭代器接口，通过实现这个接口，就能使对象能够用于foreach结构，从而在使用形式上比较统一。比如SPL中有一个DirectoryIterator类，这个类在继承SplFileInfo类的同时，实现Iterator、Traversable、SeekableIterator这三个接口，那么这个类的实例可以获得父类SplFileInfo的全部功能外，还能够实现Iterator接口所展示的那些操作。

Iterator接口的原型如下：

```
*current ()
This method returns the current index' s value.You are solely
responsible for tracking what the current index is as the
interface does not do this for you.
*key ()
This method returns the value of the current index' s key.For
foreach loops this is extremely important so that the key
value can be populated.
*next ()
This method moves the internal index forward one entry.
*rewind ()
This method should reset the internal index to the first element.
*valid ()
This method should return true or false if there is a current
element.It is called after rewind () or next () .
```

如果一个类声明了实现Iterator接口，就必须实现这五个方法，如果实现了这五个方法，那么就可以很容易对这个类的实例进行迭代。这里，DirectoryIterator类之所以拿来就能用，是因为系统已经实现了Iterator接口，所以可以像下面这样使用：

```
<? php
```

由于PHP是弱类型，且强调灵活，所以并不推荐大规模使用接口，而是仅在部分“内核”代码中使用接口，因为PHP中的接口已经失去很多接口应该具有的语义。从语义上考虑，可以更多地使用抽象类。至于抽象类和接口的比较，不再赘述。

另外，PHP5对面向对象的特性做了许多增强，其中就有一个SPL（标准PHP库）的尝试。SPL中实现一些接口，其中最主要的就是Iterator迭代器接口，通过实现这个接口，就能使对象能够用于foreach结构，从而在使用形式上比较统一。比如SPL中有一个DirectoryIterator类，这个类在继承SplFileInfo类的同时，实现Iterator、Traversable、SeekableIterator这三个接口，那么这个类的实例可以获得父类SplFileInfo的全部功能外，还能够实现Iterator接口所展示的那些操作。

Iterator接口的原型如下：

```
*current ()
This method returns the current index' s value.You are solely
responsible for tracking what the current index is as the
interface does not do this for you.
*key ()
This method returns the value of the current index' s key.For
foreach loops this is extremely important so that the key
value can be populated.
*next ()
This method moves the internal index forward one entry.
*rewind ()
This method should reset the internal index to the first element.
*valid ()
This method should return true or false if there is a current
element.It is called after rewind () or next () .
```

如果一个类声明了实现Iterator接口，就必须实现这五个方法，如果实现了这五个方法，那么就可以很容易对这个类的实例进行迭代。这里，DirectoryIterator类之所以拿来就能用，是因为系统已经实现了Iterator接口，所以可以像下面这样使用：

```
<? php
```

```
$dir=new DirectoryIterator ( dirname ( FILE ));
foreach ( $dir as $fileinfo ) {
if ( ! $fileinfo->isDir ( ) ) {
echo
$fileinfo->getFilename ( ), "\t", $fileinfo->getSize ( ), PHP_EOL;
}
}
```

可以想象，如果不用DirectoryIterator类，而是自己实现，不但代码量增加了，而且循环时候的风格也不统一了。如果自己写的类也实现了Iterator接口，那么就可以像Iterator那样工作。

为什么一个类只要实现了Iterator迭代器，其对象就可以被用作foreach的对象呢？其实原因很简单，在对PHP实例对象使用foreach语法时，会检查这个实例有没有实现Iterator接口，如果实现了，就会通过内置方法或使用实现类中的方法模拟foreach语句。这是不是和前面提到的toString方法的实现很像呢？事实上，toString方法就是接口的一种变相实现。

接口就是这样，接口本身什么也不做，系统悄悄地在内部实现了接口的行为，所以只要实现这个接口，就可以使用接口提供的方法。这就是接口“即插即用”思想。

我们都知道，接口是对多重继承的一种变相实现，而在讲继承时，我们提到了用来实现混入（Mixin）式的Traits，实际上，Traits可以被视为一种加强型的接口。

来看一段代码：

```
<? php
trait Hello {
public function sayHello ( ) {
echo ' Hello ';
}
}
trait World {
public function sayWorld ( )
echo ' World ';
```

```
}  
}  
class MyHelloWorld {  
    use Hello, World;  
    public function sayExclamationMark () {  
        echo '! ' ;  
    }  
}  
$o=new MyHelloWorld ();  
$o->sayHello ();  
$o->sayWorld ();  
$o->sayExclamationMark ();  
? >
```

上面的代码运行结果如下:

Hello World!

这里的MyHelloWorld同时实现了两个Traits，从而使其可以分别调用两个Traits里的代码段。从代码中就可以看出，Traits和接口很像，不同的是Traits是可以导入包含代码的接口。从某种意义上来说，Traits和接口都是对“多重继承”的一种变相实现。

总结关于接口的几个概念:

接口作为一种规范和契约存在。作为规范，接口应该保证可用性；作为契约，接口应该保证可控性。

接口只是一个声明，一旦使用interface关键字，就应该实现它。可以由程序员实现（外部接口），也可以由系统实现（内部接口）。接口本身什么都不做，但是它可以告诉我们它能做什么。

PHP中的接口存在两个不足，一是没有契约限制，二是缺少足够多的内部接口。

接口其实很简单，但是接口的各种应用很灵活，设计模式中也有很大一部分是围绕接口展开的。

1.5 反射

面向对象编程中对象被赋予了自省的能力，而这个自省的过程就是反射。

反射，直观理解就是根据到达地找到出发地和来源。比方说，我给你一个光秃秃的对象，我可以仅仅通过这个对象就能知道它所属的类、拥有哪些方法。

反射指在PHP运行状态中，扩展分析PHP程序，导出或提取出关于类、方法、属性、参数等的详细信息，包括注释。这种动态获取信息以及动态调用对象方法的功能称为反射API。

1.5.1 如何使用反射API

以1.1节的代码为模板，直观地认识反射的使用，如代码清单1-13所示。

代码清单1-13 reflection.php

```
<? php
class person {
public $ name;
public $ gender;
public function say () {
echo $ this->name, " \ tis", $ this->gender, " \ r \ n";
}
public function set ($ name, $ value ) {
echo "Setting $ name to $ value \ r \ n";
$ this-> $ name= $ value;
}
public function get ($ name ) {
if (! isset ( $ this-> $ name )) {
echo ' 未设置 ';
}
$ this-> $ name="正在为你设置默认值";
}
return $ this-> $ name;
```

```
}}
$student=new person ();
$student->name= ' Tom ';
$student->gender= ' male ';
$student->age=24;
```

现在，要获取这个student对象的方法和属性列表该怎么做呢？如以下代码所示：

```
//获取对象属性列表
$reflect=new ReflectionObject ($ student );
$props = $reflect->getProperties ();
foreach ($ props as $prop ) {
print $prop->getName () ." \ n";
}
//获取对象方法列表
$m= $reflect->getMethods ();
foreach ($ m as $prop ) {
print $prop->getName () ." \ n";
}
```

也可以不用反射API，使用class函数，返回对象属性的关联数组以及更多的信息：

```
//返回对象属性的关联数组
var _dump ( get _ object _ vars ( $ student ));
//类属性
var _dump ( get _ class _ vars ( get _ class ( $ student )));
//返回由类的方法名组成的数组
var _dump ( get _ class _ methods ( get _ class ( $ student )));
```

假如这个对象是从其他页面传过来的，怎么知道它属于哪个类呢？一句代码就可以搞定：

```
//获取对象属性列表所属的类
echo get_class ($student);
```

反射API的功能显然更强大，甚至能还原这个类的原型，包括方法的访问权限，如代码清单1-14所示。

代码清单1-14 使用反射API

```
//反射获取类的原型
$obj=new ReflectionClass (' person ');
$class_name=$obj->get_name ();
$Methods=$Properties=array ();
foreach ($obj->getProperties () as $v)
{
    $Properties[$v->get_name ()]=$v;
}
foreach ($obj->getMethods () as $v)
{
    $Methods[$v->get_name ()]=$v;
}
echo"class {$class_name} \n { \n";
is_array ($Properties) &&ksort ($Properties);
foreach ($Properties as $k=> $v)
{
    echo" \t";
    echo $v->isPublic ()? ' public ': ' ', $v->isPrivate ()? ' private ': ' ',
    $v->isProtected ()? ' protected ': ' ',
    $v->isStatic ()? ' static ': ' ';
    echo" \t {$k} \n";
}
echo" \n";
if ( is_array ($Methods)) ksort ($Methods);
foreach ($Methods as $k=> $v)
{
```

```
echo "\tfunction {$k} () {} \n";  
}  
echo"} \n";
```

输出如下:

```
class person  
{  
    public  gender  
    public  name  
    function get () {}  
    function set () {}  
    function say () {}  
}
```

不仅如此，PHP手册中关于反射API更是有几十个，可以说，反射完整地描述了一个类或者对象的原型。反射不仅可以用于类和对象，还可以用于函数、扩展模块、异常等。

1.5.2 反射有什么作用

反射可以用于文档生成。因此可以用它对文件里的类进行扫描，逐个生成描述文档。

既然反射可以探知类的内部结构，那么是不是可以用它做hook实现插件功能呢？或者是做动态代理呢？抛砖引玉，代码清单1-15是个简单的举例。

代码清单1-15 proxy.php

```
<? php
class mysql {
function connect ($db) {
echo"连接到数据库 $ { db[0]} \ r \ n";
}
}
class sqlproxy {
private $ target;
function construct ($ tar) {
$ this-> target[]=new $ tar ();
}
function call ($ name,$ args) {
foreach ($ this-> target as $ obj) {
$ r=new ReflectionClass ($ obj);
if ($ method=$ r->getMethod ($ name)) {
if ($ method->isPublic () &&! $ method->isAbstract ()) {
echo"方法前拦截记录LOG \ r \ n";
$ method->invoke ($ obj,$ args);
echo"方法后拦截 \ r \ n";
}
}
}
}
}
$ obj=new sqlproxy (' mysql ');
$ obj->connect (' member ');
```

这里简单说明一下，真正的操作类是mysql类，但是sqlproxy类实现了根据动态传入参数，代替实际的类运行，并且在方法运行前后进行拦截，并且动态地改变类中的方法和属性。这就是简单的动态代理。

在平常开发中，用到反射的地方不多：一个是对对象进行调试，另一个是获取类的信息。在MVC和插件开发中，使用反射很常见，但是反射的消耗也很大，在可以找到替代方案的情况下，就不要滥用。

PHP有Token函数，可以通过这个机制实现一些反射功能。从简单灵活的角度讲，使用已经提供的反射API是可取的。

很多时候，善用反射能保持代码的优雅和简洁，但反射也会破坏类的封装性，因为反射可以使本不应该暴露的方法或属性被强制暴露了出来，这既是优点也是缺点。

思考题 为什么要使用反射，反射存在的必要性是什么？或者说，反射为什么会存在？（已知一些情况：C语言是面向过程的编程语言，PHP、C++、Java是具有面向对象风格的编程语言。C语言和C++中没有对反射的原生支持，而PHP和Java具有反射API。可以思考一下，为什么C/C++语言里没有反射，以及C/C++语言里是否需要反射？）

1.6 异常和错误处理

在语言级别上，通常具有许多错误处理模式，但这些模式往往建立在约定俗成的基础上，也就是说这些错误都是预知的。但是在大型程序中，如果每次调用都去逐一检查错误，会使代码变得冗长复杂，到处充斥着if……else，并且严重降低代码的可读性。而且人的因素也是不可信赖的，程序员可能并不会把这些问题当一回事，从而导致业务异常。在这种背景下，就逐渐形成了异常处理机制，或者强迫消除这些问题，或者把问题提交给能解决它的环境。这就把“描述在正常过程中做什么事的代码”和“出了问题怎么办的代码”进行分离。

1.6.1 如何使用异常处理机制

异常的思想最早可以追溯到20世纪60年代，其在C++、Java中发扬光大，PHP则部分借鉴了这两种语言的异常处理机制。

PHP里的异常，是程序运行中不符合预期的情况及与正常流程不同的状况。一种不正常的情况，就是按照正常逻辑不该出错，但仍然出错的情况，这属于逻辑和业务流程的一种中断，而不是语法错误。PHP里的错误则属于自身问题，是一种非法语法或者环境问题导致的、让编译器无法通过检查甚至无法运行的情况。

在各种语言里，异常（exception）和错误（error）的概念是不一样的。在PHP里，遇到任何自身错误都会触发一个错误，而不是抛出异常（对于一些情况，会同时抛出异常和错误）。PHP一旦遇到非正常代码，通常都会触发错误，而不是抛出异常。在这个意义上，如果想使用异常处理不可预料的问题，是办不到的。比如，想在文件不存在且数据库连接打不开时触发异常，是不可行的。这在PHP里把它作为错误抛出，而不会作为异常自动捕获。

以经典的除零问题为例，如代码清单1-16所示。

代码清单1-16 exception.php

```
//exception.php
```

```
<? php
$a=null;
try {
$a=5/0;
echo $a, PHP_EOL;
} catch ( exception $e ) {
$e->getMessage ();
$a=-1;
}
echo $a;
```

运行结果如图1-8所示。



图 1-8 PHP里的除零错误

代码清单1-17所示是Java代码。

代码清单1-17 ExceptionTry.java

```
//ExceptionTry.java
public class ExcepetionTry {
public static void tp () throws ArithmeticException {
int a;
a=5/0;
System.out.println ("运算结果: "+a);
}
public static void main ( String[]args ) {
int a; try {
a=5/0;
System.out.println ("运算结果: "+a);
} catch ( ArithmeticException e ) {
e.printStackTrace ();} finally {
a=-1;
System.out.println ("运算结果: "+a);
}
```

```
try {
    ExceptionTry.tp ();
} catch (Exception e) {
    System.out.println ("异常被捕获");
}
}
```

运行结果如图1-9所示。



图 1-9 Java里的除零异常

把tp方法中的第二条语句改为如下形式：

```
a=5/1;
```

修改后的结果如图1-10所示。



图 1-10 Java里的异常

由以上运行结果可以看到，对于除零这种“异常”情况，PHP认为这是一个错误，直接触发错误（warning也是错误，只是错误等级不一样），而不会自动抛出异常使程序进入异常流程，故最终a值并不是预想中的-1，也就是说，并没有进入异常分支，也没有处理异常。PHP只有你主动throw后，才能捕获异常（一般情况是这样，也有一些异常PHP可以自动捕获）。

而对于Java，则认为除零属于ArithmeticException，会对其进行捕获，并对异常进行处理。

也就是说，PHP通常是无法自动捕获有意义的异常的，它把所有不正常的情况都视作了错误，你要想捕获这个异常，就得使用if……else结构，保证代码是正常的，然后判断

如果除数为0，则手动抛出异常，再捕获。Java有一套完整的异常机制，内置很多异常类会自动捕获各种各样的异常。但PHP这个机制不完善。PHP内建的常见异常类主要有pdoexception、reflection exception。

注意 其实PHP和Java之间之所以有这个差距，根本原因就在于，在Java里，异常是唯一的错误报告方式，而在PHP中却不是这样。通俗一点讲，就是这两种语言对异常和错误的界定存在分歧。什么是异常，什么是错误，两种语言的设计者存在不同的观点。

也就是说，PHP只有手动抛出异常后才能捕获异常，或者是有内建的异常机制时，会先触发错误，再捕获异常。那么PHP里的异常用法应该是什么样的呢？看下面的例子。

先定义两个异常类，它们需要继承自系统的exception，如代码清单1-18所示。

代码清单1-18 定义两个异常类

```
class emailException extends exception {
}
class pwdException extends exception {
function toString ()
{
return "<div class= \"error \">>Exception {$this->getCode ()}:
{$this->getMessage ()}
in File: {$this->getFile ()} on line: {$this->getLine ()} </div>";
//改写抛出异常结果
}
}
```

然后就是实际的业务，根据业务需求抛出不同异常，如代码清单1-19所示。

代码清单1-19 根据业务需求抛出不同异常

```
function reg ($reginfo=null) {
if (empty ($reginfo) | ! isset ($reginfo)) {
```

```
throw new Exception ("参数非法");
}
if ( empty ( $reginfo[ ' email ' ])) {
throw new emailException ( "邮件为空");
}
if ( $reginfo[ ' pwd ' ]! = $reginfo[ ' repwd ' ]) {
throw new pwdException ( "两次密码不一致");
}
echo ' 注册成功 ' ;
}
```

上面的代码判断传入的参数，根据业务进行异常分发。首先，如果没有传入任何参数（这个参数可以是POST进来，也可以是别的地方赋值得到），就把异常分发给exception超类，跳出注册流程；如果Email地址不存在，那么把异常分发给自定义的emailException异常，跳出注册流程；如果两次密码不一致，则将异常分发给自定义的pwdException，跳出注册流程。

现在异常分发了，但还不算完，还需要对异常进行分拣并做处理。代码如下所示：

```
try {
reg ( array ( ' email ' => ' waitfox@qq.com ' , ' pwd ' => 123456 , ' repwd ' =>
12345678 ));
//reg ();
} catch ( emailException $ee ) {
echo $ee->getMessage ();
} catch ( pwdException $ep ) {
echo $ep;
echo PHP_EOL, ' 特殊处理 ' ;
} catch ( Exception $e ) {
echo $e->getTraceAsString ();
echo PHP_EOL, ' 其他情况, 统一处理 ' ;
}
```

这一段代码用于捕获所抛出的各种异常，进行分门别类的处理。

提示 可以尝试不同注册条件，看看异常分拣的流程。需要注意，exception作为超类应该放在最后捕获。不然，捕获这个异常超类后，后面的捕获就终止了，而这个超类不能提供针对性的信息和处理。

在这里，对表单进行异常处理，通过重写异常类、手动抛出错误的方式进行异常处理。这是一种业务异常，可以人为地把所有不符合要求的情况都视作业务异常，和通常意义上的代码异常相区别。

那PHP里的异常应该怎么用？在什么时候抛出异常，什么时候捕获呢？什么场景下能应用异常？在下面三种场景下会用到异常处理机制。

1.对程序的悲观预测

如果一个程序员对自己的代码有“悲观情绪”，这里并不是指该程序员代码质量不高，而是他认为自己的代码无法一一处理各种可预见、不可预见的情况，那该程序员就会进行异常处理。假设一个场景，程序员悲观地认为自己的这段代码在高并发条件下可能产生死锁，那么他就会悲观地抛出异常，然后在死锁时进行捕获，对异常进行细致的处理。

2.程序的需要和对业务的关注

如果程序员希望业务代码中不会充斥大堆的打印、调试等处理，通常他们会使用异常机制；或者业务上需要定义一些自己的异常，这个时候就需要自定义一个异常，对现实世界中各种各样的业务进行补充。比如上班迟到，这种情况认为是一个异常，要收集起来，到月底集中处理，扣你工资；如果程序员希望有预见性地处理可能发生的、会影响正常业务的代码，那么它需要异常。在这里，强调了异常是业务处理中必不可少的环节，不能对异常视而不见。异常机制认为，数据一致很重要，在数据一致性可能被破坏时，就需要异常机制进行事后补救。

举个例子，比如有个上传文件的业务需求，要把上传的文件保存在一个目录里，并在数据库里插入这个文件的记录，那么这两步就是互相关联、密不可分的一个集成的业务，缺一不可。文件保存失败，而插入记录成功就会导致无法下载文件；而文件保存成功数据库写入失败，则会导致没有记录的文件成为死文件，永远得不到下载。

那么假设文件保存成功后没有提示，但是保存失败会自动抛出异常，访问数据库也一

样，插入成功没有提示，失败则自动抛出异常，就可以把这两个有可能抛出异常的代码段包在一个try语句里，然后用catch捕捉错误，在catch代码段里删除没有被记录到数据库的文件或者删除没有文件的记录，以保证业务数据的一致性。因此，从业务这个角度讲，异常偏重于保护业务数据一致性，并且强调对异常业务的处理。

如果代码中只是象征性地try……catch，然后打印一个报错，最后over。这样的异常不如不用，因为其没有体现异常思想。所以，合理的代码应该如下：

```
<? php
try {
//可能出错的代码段
if ( 文件上传不成功 ) throw ( 上传异常 );
if ( 插入数据库不成功 ) throw ( 数据库操作异常 );
} catch ( 异常 ) {
必须的补救措施，如删除文件、删除数据库插入记录，这个处理很细致
}
//……
? >
```

也可以如下：

```
<? php
上传 {
if ( 文件上传不成功 ) throw ( 上传异常 );
if ( 插入数据库不成功 ) throw ( 数据库操作异常 );
}
//其他代码……
try {
上传; 其他;
} catch ( 上传异常 ) {
必须的补救措施，如删除文件，删除数据库插入记录
} catch ( 其他异常 ) {
记录log
}
}
```

? >

上面两种捕获异常的方式中，前一种是在异常发生时立刻捕获；后一种是分散抛异常集中捕获。那到底应该是哪一种呢？

如果业务很重要，那么异常越早处理越好，以保证程序在意外情况下能保持业务处理的一致性。比如一个操作有多个前提步骤，突然最后一个步骤异常了，那么其他前提操作都要消除掉才行，以保证数据的一致性。并且在这种核心业务下，有大量的代码来做善后工作，进行数据补救，这是一种比较悲观的、而又重要的异常。我们应把异常消灭在局部，避免异常的扩散。

如果异常不是那么重要，并且在单一入口、MVC风格的应用中，为了保持代码流程的统一，则常常采用后一种异常处理方式。这种异常处理方式更多强调业务流程的走向，对善后工作并不是很关心。这是一种次要异常，其将异常集中处理从而使流程更专一。

异常处理机制可以把每一件事当做事务考虑，还可以把异常看成一种内建的恢复系统。如果程序某部分失败，异常将恢复到某个已知稳定的点上，而这个点就是程序的上下文环境，而try块里面的代码就保存catch所要知道的程序上下文信息。因此，如果很看重异常，就应该分散进行try……catch处理。

3. 语言级别的健壮性要求

在健壮性这点上，PHP是不足的。以Java为例，Java是一种面向企业级开发的语言，强调健壮性。Java中支持多线程，Java认为，多线程被中断这种情况是彻彻底底的无法预料和避免的。所以Java规定，凡是用了多线程，就必须正视这种情况。你要么抛出，不管它；要么捕获，进行处理。总之，你必须面对InterruptedException异常，不准回避。也就是异常发生后应对重要数据业务进行补救，当然你可以不做，但是你必须意识到，异常有可能发生。

这类异常是强制的。更多异常是非强制的，由程序员决定。Java对异常的分类和约束，保证了Java程序的健壮性。

异常就是无法控制的运行时错误，会导致出错时中断正常逻辑运行，该异常代码后面

的逻辑都不能继续运行。那么try……catch处理的好处就是：可以把异常造成的逻辑中断破坏降到最小范围内，并且经过补救处理后不影响业务逻辑的完整性；乱抛异常和只抛不捕获，或捕获而不补救，会导致数据混乱。这就是异常处理的一个重要作用，就是通过精确控制运行时的流程，在程序中断时，有预见地用try缩小可能出错的影响范围，及时捕获异常发生并做出相应补救，以使逻辑流程仍然能回到正常轨道上。

1.6.2 怎样看PHP的异常

PHP中的异常机制是不足的，绝大多数情况下无法自动抛出异常，必须用if……else先进行判断，再手动抛出异常。手动抛异常的意义不是很大，因为这意味着在代码里已经充分预期到错误的出现，也就算不上真正的“异常”，而是意料之中。同时，这种方式还会使你陷入纷繁复杂的业务逻辑判断和处理中。

Java语言做得比较好的就是定义了一堆内置的常见异常，不需要程序员判断各种异常情况后再手动抛出，编译器会代我们进行判断业务是否发生错误，若发生了，则自动抛出异常。作为程序员，只需要关心异常的捕获和随后补救，而不是像PHP那样关注到底会发生哪些异常，用if……else逐一判断，逐一抛出异常。

有没有什么机制使得PHP可以自动抛出异常呢？有，那就是结合PHP中的错误处理主动抛出异常。

使用异常能一定程度上会降低耦合性，但是也不能滥用。滥用异常的后果就是很可能导致代码被多处挂起，流程变得更复杂，难于理解。但是可以肯定，异常在PHP里有很大的价值，越复杂的应用，越需要合理考虑使用异常。

提示 需要提醒读者关注，SPL里定义了一大堆exception，如BadMethodCallException、LogicException等，同时这些异常之间还存在层级关系。这些异常只是一个空壳，什么方法都没有，需要自己填充。它们实际上起到一个命名参考的作用。

1.6.3 PHP中的错误级别

错误处理本来不属于面向对象的范畴，但是既然讲到异常，就不得不提及异常的同胞兄弟——错误。

PHP错误处理比异常的价值大得多。PHP错误的概念已经和异常做过比较，这里通过对PHP异常的认知，给PHP错误下个最直观最通俗的结论：PHP错误就是会使脚本运行不正常的情况。

PHP错误有很多种，包括warning、notice、deprecated、fatal error等。这和一般意义的错误概念有些差别。所以，notice不叫通知，而叫通知级别的错误，warning也不叫警告，而叫警告级别的错误。

错误大致分为以下几类。

deprecated是最低级别的错误，表示“不推荐，不建议”。比如在PHP 5中使用ereg系列的正则匹配函数就会报此类错误。这种错误一般由于使用不推荐的、过时的函数或语法造成的。其虽不影响PHP正常流程，但一般情况下建议修正。

其次是notice。这种错误一般告诉你语法中存在不当的地方。如使用变量但是未定义就会报此错。最常见的，数组索引是字符时没有加引号，PHP就视为一个常量，先查找常量表，找不到再视为变量。虽然PHP是脚本语言，语法要求不严，但是仍然建议对变量进行初始化。这种错误不影响PHP正常流程。

warning是级别比较高的错误，在语法中出现很不恰当的情况时才会报此错误，比如函数参数不匹配。这种级别的错误会导致得不到预期结果，故需要修改代码。

更高级别的错误是fatal error。这是致命错误，直接导致PHP流程终结，后面的代码不再执行。这种问题非改不可。

最高级别的错误是语法解析错误parse error。上面提到的错误都属于PHP代码运行期间错误，而语法解析错误属于语法检查阶段错误，这将导致PHP代码无法通过语法检查。错误级别不止这几个，最主要的都在前面提到了。PHP手册中一共定义了16个级别的错误，最常用的就这几个。代码清单1-20演示了常见级别的错误。

代码清单1-20 error.php

```
//Error.php
<? php
$date= ' 2012-12-20 ';
if ( ereg ( " ( [0-9] { 4 } ) - ( [0-9] { 1, 2 } ) - ( [0-9] { 1, 2 } ) ", $date, $regs )) {
echo "$regs[3]. $regs[2]. $regs[1]";
} else {
echo "Invalid date format: $date";
}
if ($i>5) {
echo ' $i没有初始化啊 ', PHP_EOL;
}
$a=array ( ' o ' =>2, 4, 6, 8);
echo $a[0];
$result=array_sum ( $a, 3 );
echo fun ();
echo ' 致命错误后呢? 还会执行吗? ';
//echo ' 最高级别的错误 ', $55;
```

这段代码演示至少四个级别的错误，如果看不全，应确保你的php.ini文件做了如下设定：

```
error_reporting=E_ALL | E_STRICT
display_errors=On
```

error_reporting指定错误级别，上面的设置是最严格的错误级别，具体设置可以参php.ini。

提示 有一个技巧我想你会用到，那就是在代码质量或者环境不可控时（比如数据库连接失败），使用error_reporting（0），这样就能屏蔽错误了，正式部署时可以采取这样的策略，防止错误消息泄露敏感信息。另外一个技巧就是在函数前加@符号，抑制错误

信息输出，如@mysql_connect()。

1.6.4 PHP中的错误处理机制

PHP里有一套错误处理机制，可以使用`set_error_handler`接管PHP错误处理，也可以使用`trigger_error`函数主动抛出一个错误。

`set_error_handler()` 函数设置用户自定义的错误处理函数。函数用于创建运行期间的用户自己的错误处理方法。它需要先创建一个错误处理函数，然后设置错误级别。语法如下：

```
set_error_handler ( error_function, error_types )
```

参数描述如下：

`error_function`：规定发生错误时运行的函数。必需。

`error_types`：规定在哪个错误报告级别会显示用户定义的错误。可选。默认为“E_ALL”。

提示 如果使用该函数，会完全绕过标准PHP错误处理函数，如果有必要，用户定义的错误处理程序必须终止（`die()`）脚本。

如果在脚本执行前发生错误，由于在那时自定义程序还没有注册，因此就不会用到这个自定义错误处理程序。这先实现一个自定义的异常处理函数，如代码清单1-21所示。

代码清单1-21 自定义的异常处理函数

```
<? php
function customError ($errno,$errstr,$errfile,$errline)
{
echo"<b>错误代码: </b>[{$errno}]{$errstr} \r\n";
echo"错误所在的代码行: {$errline} 文件 {$errfile} \r\n";
echo"PHP版本", PHP_VERSION, "(", PHP_OS, ") \r\n";
//die ();
```

```
}
set_error_handler ("customError", E_ALL | E_STRICT);
$a=array ('o' =>2, 4, 6, 8);
echo $a[0];
```

在这个函数里，可以对错误的详情进行格式化输出，也可以做任何要做的事情，比如判断当前环境和权限给出不同的错误提示，可使用`error_log`函数将错误记入log文件，还可以细化处理，针对`errno`的不同进行对应的处理。

自定义的错误处理函数一定要有这四个输入变量`errno`、`errstr`、`errfile`、`errline`。

`errno`是一组常量，代表错误的等级，同时也有一组整数和其对应，但一般使用其字符串值表示，这样语义更好一点。比如`E_WARNING`，其二进制掩码为4.，表示警告信息。

接下来，就是将这个函数作为回调参数传递给`set_error_handler`。这样就能接管PHP原生的错误处理函数了。要注意的是，这种托管方式并不能托管所有种类的错误，如`E_ERROR`、`E_PARSE`、`E_CORE_ERROR`、`E_CORE_WARNING`、`E_COMPILE_ERROR`、`E_COMPILE_WARNING`，以及`E_STRICT`中的部分。这些错误会以最原始的方式显示，或者不显示。

`set_error_handler`函数会接管PHP内置的错误处理，你可以在同一个页面使用`restore_error_handler()`；取消接管。

注意 如果使用自定义的`set_error_handler`接管PHP的错误处理，先前代码里的错误抑制`@`将失效，这种错误也会被显示。

在PHP异常中，异常处理机制是有限的，无法自动抛出异常，必须手动进行，并且内置异常有限。PHP把许多异常看做错误，这样就可以把这些“异常”像错误一样用`set_error_handler`接管，进而主动抛出异常。代码如下所示：

```
function customError ($errno,$errstr,$errfile,$errline)
{
//自定义错误处理时，手动抛出异常
```

```
throw new Exception ($level. ' | '. ${errstr});
}
set_error_handler ("customError", E_ALL | E_STRICT);
try
{
$a=5/0;
}
catch (Exception $e)
{
echo ' 错误信息: ', $e->getMessage ();
}
```

这样就能捕获到异常和非致命的错误，就能按照1.6.1节里讲述的方法进行了，这样可以弥补PHP异常处理机制的部分不足。

这种“曲折迂回”的处理方式存在的问题就是：必须依靠程序员自己来掌控对异常的处理，对于异常高发区、敏感区，如果程序员处理不好，就会导致前面所提到的业务数据不一致的问题。其优点在于，可以获得程序运行时的上下文信息，以进行针对性的补救。

fatal error这样的错误虽然捕获不到，也无法在发生此错误后恢复流程处理，但是还是可以使用一些特殊方法对这种错误进行处理的。这需要用到一个函数——register_shutdown_function，此函数会在PHP程序终止或者die时触发一个函数，给PHP来一个短暂的“回光返照”。在PHP 4的时代，类不支持析构函数，常用这个函数模拟实现析构函数。实例代码如下：

```
<? php
class Shutdown
{
public function stop ()
{
if ( error_get_last ())
{
print_r ( error_get_last ());
}
die ( ' Stop. ');
}
```

```
}  
}  
register_shutdown_function ( array ( new Shutdown (), ' stop '));  
$a=new a (); //将因为致命错误而失败  
echo ' 必须终止 ';
```

可以运行看看效果。对于fatal error还能做点收尾工作，但是PHP流程的终止是必然的。对于Parse error级别的错误，你只有傻眼了，除了可以修改配置文件php.ini，什么都做不了，修改的内容如下：

```
log_errors=On  
error_log=usr/log/php.log
```

这样一旦PHP发生了错误，就会被记入log文件，方便以后查询。

和exception类似，错误处理也有对应抛出错误的函数，那就是trigger_error函数，如下所示：

```
<? php  
$divisor=0;  
if ($divisor==0) {  
trigger_error ( "Cannot divide by zero", E_USER_ERROR );  
}  
echo ' break ';
```

关于错误处理，主要就是这些内容，还有一些错误处理和调试相关，我们将会放到后面的章节进行讲解。

提示 在PHP中，错误和异常是两个不同的概念，这种设计从根本上导致了PHP的异常和其他语言相异。以Java为例，Java中，异常是错误唯一的报告方式。说到底，两

者的区别就是对异常和错误的认识不同而产生的。PHP的异常绝大部分必须通过某种办法手动抛出，才能被捕获到，是一种半自动化的异常处理机制。

无论是错误还是异常，都可以使用handler接管系统已有的处理机制。

1.7 本章小结

本章主要介绍面向对象思想的程序的组成元素——类和对象。类是一个动作和属性的模板，对象是数据的集合。结合PHP自身实际情况，着重讲述PHP里面向对象的一些比较模糊的知识点，包括魔术方法、接口、多态、类的复用、反射、异常机制等。接口是一种类型，从接口的实现讲述接口是怎么实现“即插即用”的。

然后，对异常机制进行探讨。讲述异常应该是什么样的，应该怎么用，并且阐述了PHP中的异常为什么会这样，应该在什么场合使用异常等。PHP起初没有异常机制，后期为了进军企业级开发，才模仿Java加进去的，故有了错误处理和异常处理的并存，这种形式导致PHP异常处理不伦不类，通过和Java对比，让我们了解到了异常的真实含义。错误处理是对异常处理的一种补充。

到底面向过程和面向对象孰优孰劣呢？答案是：二者间并无高低优劣之别，它们各有优劣。

其实在OO发展中，暴露出一些问题，如深入对象内部读写状态存在的困难，现实和开发中不对应造成的建模困难，数据与逻辑绑定造成的类型臃肿。比如前面提到的反射，就是因为面向对象的封装导致读写内部状态比较困难而产生的。

面向对象存在的问题是越来越多的语言引入函数式编程的特征，如闭包、回调等。PHP也引入一些函数式编程的概念，有兴趣的读者可以自行研究。

第2章 面向对象的设计原则

第1章 已经说过，面向对象是一种高度抽象的思维。在面向对象设计中，类是基本单位，各种设计都是围绕着类来进行的。可以说，类与类之间的关系，构成了设计模式的大部分内容。

在初学阶段，可以认为类就是属性+函数组成的，实际上在底层存储上也确实是这样的，但是，这些仅仅是确定一个独立的类。而类与类之间的关系是设计模式所要探讨的内容。

经典的设计模式有23种，每种都是对代码复用和设计的总结，就设计模式而言，除了熟读GOF经典外，推荐《敏捷软件开发——原则、方法与实践》一书。本章并不就具体的设计模式展开讨论，而是讨论一些基本的设计原则，并给出一些小的实例，最后，作为前两章的总结，探讨一下PHP中的面向对象的一些问题。

2.1 面向对象设计的五大原则

在面向对象的设计中，如何通过很小的设计改变就可以应对设计需求的变化，这是设计者极为关注的问题。为此不少OO先驱提出了很多有关面向对象的设计原则用于指导OO的设计和开发。下面是几条与类设计相关的设计原则。

面向对象设计的五大原则分别是单一职责原则、接口隔离原则、开放-封闭原则、替换原则、依赖倒置原则。这五大原则也是23种设计模式的基础^[1]。

2.1.1 单一职责原则

亚当·斯密曾就制针业做过一个分工产生效率的例子^[2]。对于一个没有受过相应训练，又不知道怎样使用这种职业机械的工人来讲，即使他竭尽全力地工作，也许一天连一根针也生产不出来，当然更生产不出20根针了。但是，如果把这个行业分成各种专门的组织，再把这种组织分成许多个部门，其中大部分部门也同样分为专门的组织。把制针分为18种不同工序，这18种不同操作由18个不同工人来担任。那么，尽管他们的机器设备都

很差，但他们尽力工作，一天也能生产12磅针。每磅中等型号针有4000根，按这个数字计算，十多个人每天就可以制造48000根针，而每个人每天能制造4800根针。如果他们各自独立地工作，谁也不专学做一种专门的业务，那么他们之中无论是谁都绝不可能一天制造20根针，也许连1根针也制造不出来。这就是企业管理中的分工，在面向对象的设计里，叫做单一职责原则（Single Responsibility Principle, SRP）。

在《敏捷软件开发》中，把“职责”定义为“变化的原因”，也就是说，就一个类而言，应该只有一个引起它变化的原因。这是一个最简单，最容易理解却最不容易做到的一个设计原则。说得简单一点，就是怎样设计类以及类的方法界定的问题。这种问题是很普遍的，比如在MVC的框架中，很多人会有这样的疑惑，对于表单插入数据库字段过滤与安全检查应该是放在control层处理还是model层处理，这类问题都可以归到单一职责的范围。

再比如在职员类里，将工程师、销售人员、销售经理等都放在职员类里考虑，其结果将会非常混乱。在这个假设下，职员类里的每个方法都要用if……else判断是哪种情况，从类结构上来说将会十分臃肿，并且上述三种职员类型，不论哪一种发生需求变化，都会改变职员类，这是我们所不愿意看到的！

从上面的描述中应该能看出，单一职责有两个含义：一个是避免相同的职责分散到不同的类中，另一个是避免一个类承担太多职责。

那为什么要遵守SRP呢？

（1）可以减少类之间的耦合

如果减少类之间的耦合，当需求变化时，只修改一个类，从而也就隔离了变化；如果一个类有多个不同职责，它们耦合在一起，当一个职责发生变化时，可能会影响其他职责。

（2）提高类的复用性

修理电脑比修理电视机简单多了。主要原因就在于电视机各个部件之间的耦合性太高，而电脑则不同，电脑的内存、硬盘、声卡、网卡、键盘灯部件都可以很容易地单独拆卸和组装。某个部件坏了，换上新的即可。

上面的例子就体现了单一职责的优势。由于使用了单一职责，使得“组件”可以方便地“拆卸”和“组装”。

不遵守SRP会影响对该类的复用性。当只需要复用该类的某一个职责时，由于它和其他的职责耦合在一起，也就很难分离出。

遵守SRP在实际代码开发中有没有什么应用？有的。以数据持久层为例，所谓的数据持久层主要指的是数据库操作，当然，还包括缓存管理等。以数据库操作为例，如果是一个复杂的系统，那么就可能涉及多种数据库的相互读写等，这时就需要数据持久层支持多种数据库。应该怎么做？定义多个数据库操作类？你的想法已经很接近了，再进一步，就是使用工厂模式。

工厂模式（Factory）允许你在代码执行时实例化对象。它之所以被称为工厂模式是因为它负责“生产”对象。以数据库为例，工厂需要的就是根据不同的参数，生成不同的实例化对象。最简单的工厂就是根据传入的类型名实例化对象，如传入MySQL，就调用MySQL的类并实例化，如果是SQLite，则调用SQLite的类并实例化，甚至可以处理TXT、Excel等“类数据库”。工厂类也就是这样的一个类，它只负责生产对象，而不负责对象的具体内容。

先定义一个接口，规定一些通用的方法，如代码清单2-1所示。

代码清单2-1 定义一个适配器接口

```
<? php
interface Db_Adapter {
/**
 *数据库连接
 *@param $config数据库配置
 *@return resource
 */
public function connect ($config);
/**
 *执行数据库查询
 *@param string $query数据库查询SQL字符串
 *@param mixed $handle连接对象
```

```
*@return resource*/  
public function query ($ query,$ handle );  
}? >
```

这是一个简化的接口，并没有提供所有方法，其定义了MySQL数据库的操作类，这个类实现了Db_Adapter接口，具体如代码清单2-2所示。

代码清单2-2 定义MySQL数据库的操作类

```
<? php  
class Db_Adapter_Mysql implements Db_Adapter  
{  
private $_dbLink; //数据库连接字符串标示  
/**  
*数据库连接函数  
*  
*@param $ config数据库配置  
*@throws Db_Exception  
*@return resource*/  
public function connect ($ config )  
{  
if ( $ this-> _dbLink=@mysql_connect ( $ config->host.  
( empty ( $ config->port)? ' ': ' ' . $ config->port ),  
$ config->user,$ config->password, true )) {  
if ( @mysql_select_db ( $ config->database,$ this-> _dbLink)) {  
if ( $ config-> charset ) {  
mysql_query ( "SET NAMES ' { $ config-> charset } ' ", $ this-> _dbLink );  
}  
return $ this-> _dbLink;  
}}  
  
/**数据库异常*/  
throw new Db_Exception ( @mysql_error ( $ this-> _dbLink));  
}  
/**  
*执行数据库查询  
*  
*/
```

```
*@param string $query数据库查询SQL字符串
*@param mixed $handle连接对象
*@return resource
*/
public function query ( $query, $handle )
{
if ( $resource=@mysql_query ( $query, $handle ) ) {
return $resource;
}
}
}?? >
```

接下来是SQLite数据库的操作类，同样实现了Db_Adapter接口，如代码清单2-3所示。

代码清单2-3 SQLite数据库的操作类

```
<? php
class Db_Adapter_sqlite implements Db_Adapter
{
private $_dbLink; //数据库连接字符串标示
/**
*数据库连接函数
*@param $config数据库配置
*@throws Db_Exception
*@return resource*/
public function connect ( $config )
{
if ( $this->_dblink=mysqlite_open ( $config->file, 0666, $error ) ) {
return $this->_dblink;
}
/**数据库异常*/
throw new Db_Exception ( $error );
}
/**
*执行数据库查询
```

```
*@param string $query数据库查询SQL字符串

*@param mixed $handle连接对象
*@return resource
*/
public function query ( $query,$handle )
{
if ( $resource=@sqlite _query ( $query,$handle ) ) {
return $resource;
}
}
}
```

好了，如果现在需要一个数据库操作的方法的话怎么做？只需定义一个工厂类，根据传入不同的参数生成需要的类即可，如代码清单2-4所示。

代码清单2-4 定义一个工厂类

```
<? php
class sqlFactory
{
public static function factory ( $type )
{
if ( include _once ' Drivers/' . $type. ' .php ' ) {
$classname= ' Db _Adapter _ ' . $type;
return new $classname;
} else {
throw new Exception ( ' Driver not found ' );
}
}
}
}?
```

要调用时，就可以这么写：

```
$db=sqlFactory: factory ( ' MySQL ' );
```

```
$db=sqlFactory: factory ( ' SQLite ' );
```

我们把创建数据库连接这块程序单独拿出来，程序中的CURD就不用关心是什么数据库了，只要按照规范使用对应的方法即可。

工厂方法让具体的对象解脱了出来，使其并不再依赖具体的类，而是抽象。除了数据库操作这种显而易见的设计外，还有什么地方会用到工厂类呢？那就是SNS中的动态实现。

下面的图片来自国内某SNS网站，属于当前新鲜事页面，可以看到针对不同行为，其生成了不同动态。比如，参加了某个小组，动态显示的就是“XX参加了YY小组”；收到某某的礼物，别人看到的多台就是“XX收到了YY的ZZ礼物”，如图2-1所示。

以上这种动态应该怎么设计呢，最容易想到的就是用工厂模式，根据传入的操作不同，结合模板而生成不同的动态，如代码清单2-5所示。



图 2-1 某SNS网站的动态展示

代码清单2-5 工厂模式

```
<bead id="feedServiceFactory" class="FeedServiceFactory">
  <property name="feedMap">
    <map>
      <entry key="friend" value-ref="friendFeed"/>
      <entry key="album" value-ref="albumFeed"/>
      <entry key="reply" value-ref="replyFeed"/>
      <entry key="share" value-ref="shareFeed"/>
      <entry key="video" value-ref="videoFeed"/>
    </map>
  </property>
</bead>
```

```
<entry key="group"value-ref="groupFeed"/>
</map>
</property>
</bean>
```

以上代码是一个动态的生成配置，通过FEED的类型匹配到key，取到对应的bean，然后创建不同的动态，用的就是工厂模式。

设计模式里面的命令模式也是SRP的体现，命令模式分离“命令的请求者”和“命令的实现者”方面的职责。举一个很好理解的例子，就是你去餐馆吃饭，餐馆存在顾客、服务员、厨师三个角色。作为顾客，你只要列出菜单，传给服务员，由服务员通知厨师去实现。作为服务员，只需要调用准备饭菜这个方法（对厨师大喊“该炒菜了”），厨师听到要炒菜的请求，就立即去做饭。在这里，命令的请求和实现就完成了解耦。

模拟这个过程，首先定义厨师角色，厨师进行实际的做饭、烧汤的工作。详细代码如代码清单2-6所示。

代码清单2-6 餐馆的示例

```
/**
  厨师类，命令接受者与执行者
  ***/
class cook {
  public function meal () {
    echo ' 番茄炒鸡蛋 ', PHP_EOL;
  }
  public function drink () {
    echo ' 紫菜蛋花汤 ', PHP_EOL;
  }
  public function ok () {
    echo ' 完毕 ', PHP_EOL;
  }
}
//然后是命令接口
interface Command {
//命令接口
```

```
public function execute ( );  
}
```

现在轮到服务员出场，服务员是命令的传送者，通常你到饭馆吃饭都是叫服务员吧，不可能直接叫厨师，一般都是叫“服务员，给我来盘番茄炒西红柿”，而不会直接叫“厨师，给我来盘番茄炒西红柿”。所以，服务员是顾客和厨师之间的命令沟通者。模拟这个过程代码如代码清单2-7所示。

代码清单2-7 模拟服务员与厨师的过程

```
class MealCommand implements Command {  
private $ cook;  
//绑定命令接受者  
public function construct ( cook $ cook ) {  
$ this-> cook= $ cook;  
}  
public function execute ( ) {  
$ this-> cook-> meal ( ); //把消息传递给厨师，让厨师做饭，下同  
}  
}  
class DrinkCommand implements Command {  
private $ cook;  
//绑定命令接受者  
public function construct ( cook $ cook ) {  
$ this-> cook= $ cook;  
}  
public function execute ( ) {  
$ this-> cook-> drink ( );  
}}}
```

现在顾客可以按照菜单叫服务员了，如代码清单2-8所示。

代码清单2-8 模拟顾客与服务员的过程

```
class cookControl {
private $ mealcommand;
private $ drinkcommand;
//将命令发送者绑定到命令接收器上面来
public function addCommand ( Command $ mealcommand, Command $ drinkcommand ) {
    $ this-> mealcommand= $ mealcommand;
    $ this-> drinkcommand= $ drinkcommand;
}
public function callmeal () {
    $ this-> mealcommand->execute ();
}
public function calldrink () {
    $ this-> drinkcommand->execute ();
}}}
```

好了，现在完成整个过程，如代码清单2-9所示。

代码清单2-9 实现命令模式

```
$ control=new cookControl;
$ cook=new cook;
$ mealcommand=new MealCommand ( $ cook );
$ drinkcommand=new DrinkCommand ( $ cook );
$ control->addCommand ( $ mealcommand,$ drinkcommand );
$ control->callmeal ();
$ control->calldrink ();
```

从上面的例子可以看出，原来设计模式并非纯理论的东西，而是来源于实际生活，就连普通的餐馆老板都懂设计模式这门看似高深的学问。其实，在经济和管理活动中，对流程的优化就是对各种设计模式的摸索和实践。所以，设计模式并非计算机编程中的专利。事实上，设计模式的起源不是计算机学科，而是源于建筑学。

在设计模式方面，不仅以上这两种体现了SRP，还有别的（比如代理模式）也体现了SRP。SRP不只是对类设计有意义，对以模块、子系统为单位的系统架构设计同样有意

义。

模块、子系统也应该仅有一个引起它变化的原因，如MVC所倡导的各个层之间的相互分离其实就是SRP在系统总体设计中的应用。图2-2是来自CI框架的流程图。

SRP是最简单的原则之一，也是最难做好的原则之一。我们会很自然地将职责连接在一起。找到并且分离这些职责是软件设计需要达到的目的。

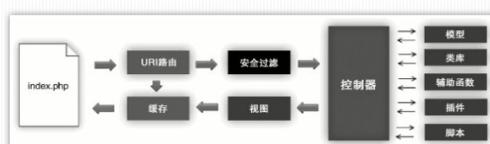


图 2-2 MVC中的流程

一些简单的应该遵循的做法如下：

根据业务流程，把业务对象提炼出来。如果业务流层的链路太复杂，就把这个业务对象分离为多个单一业务对象。当业务链标准化后，对业务对象的内部情况做进一步处理。把第一次标准化视为最高层抽象，第二次视为次高层抽象，以此类推，直到“恰如其分”的设计层次。

职责的分类需要注意。有业务职责，还要有脱离业务的抽象职责，从认识业务到抽象算法是一个层层递进的过程。就好比命令模式中的顾客，服务员和厨师的职责，作为老板（即设计师）的需要规划好各自的职责范围，既要防止越俎代庖，也要防止互相推诿。

[1]这些原则主要是由Robert C.Martin在《敏捷软件开发——原则、方法与实践》一书中总结出来的。

[2]见《国富论》第1章，分工理论是亚当·斯密的一个重要经济理论。

2.1.2 接口隔离原则

设计应用程序的时候，如果一个模块包含多个子模块，那么我们应该小心对该模块做出抽象。设想该模块由一个类实现，我们可以把系统抽象成一个接口。但是要添加一个新的模块扩展程序时，如果要添加的模块只包含原系统中的一些子模块，那么系统就会强迫我们实现接口中的所有方法，并且还要编写一些哑方法。这样的接口被称为胖接口或者被污染的接口，使用这样的接口将会给系统引入一些不当的行为，这些不当的行为可能导致不正确的结果，也可能导致资源浪费。

1. 接口隔离

接口隔离原则（Interface Segregation Principle, ISP）表明客户端不应该被强迫实现一些他们不会使用的接口，应该把胖接口中的方法分组，然后用多个接口代替它，每个接口服务于一个子模块。简单地说，就是使用多个专门的接口比使用单个接口要好得多。

ISP的主要观点如下：

1) 一个类对另外一个类的依赖性应当是建立在最小的接口上的。

ISP可以达到不强迫客户（接口的使用方）依赖于他们不用的方法，接口的实现类应该只呈现为单一职责的角色（遵守SRP原则）。

ISP还可以降低客户之间的相互影响——当某个客户程序要求提供新的职责（需求变化）而迫使接口发生改变时，影响到其他客户程序的可能性会最小。

2) 客户端程序不应该依赖它不需要的接口方法（功能）。

客户端程序不应该依赖它不需要的接口方法（功能），那依赖什么？依赖它所需要的接口。客户端需要什么接口就提供什么接口，把不需要的接口剔除，这就要求对接口进行细化，保证其纯洁性。

比如在应用继承时，由于子类将继承父类中的所有可用的方法；而父类中的某些方法，在子类中可能并不需要。例如，普通员工和经理都继承自雇员这个接口，员工需要每

天写工作日志，而经理则不需要。因此不能用工作日志来卡经理，也就是经理不应该依赖于提交工作日志这个方法。

可以看出，ISP和SRP在概念上是有一定交叉的。事实上，很多设计模式在概念上都有交叉，甚至你很难判断一段代码属于哪一种设计模式。

ISP强调的是接口对客户端的承诺越少越好，并且要做到专一。当某个客户程序的要求发生变化，而迫使接口发生改变时，影响到其他客户程序的可能性小。这实际上就是接口污染的问题。

2. 对接口的污染

过于臃肿的接口设计是对接口的污染。所谓接口污染就是为接口添加不必要的职责，如果开发人员在接口中增加一个新功能的主要目的只是减少接口实现类的数目，则此设计将导致接口被不断地“污染”并“变胖”。

接口污染会给系统带来维护困难和重用性差等方面的问题。为了能够重用被污染的接口，接口的实现类就被迫要实现并维护不必要的功能方法。

“接口隔离”其实就是定制化服务设计的原则。使用接口的多重继承实现对不同的接口的组合，从而对外提供组合功能——达到“按需提供服务”。

看下面这个例子，如图2-3所示。

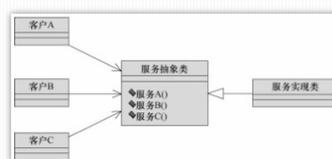


图 2-3 存在污染的接口设计

客户A需要A服务，只要针对客户A的方法发生改变，客户B和客户C就会受到影响。故这种设计需要对接口进行隔离，如图2-4所示。

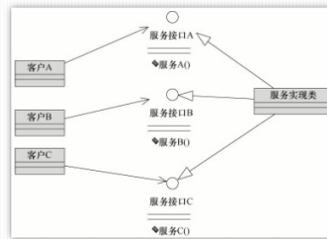


图 2-4 减少接口中的污染

由图2-4可知，如果针对客户A的方法发生改变，客户B和客户C并不会受到任何影响。你可能会想，这样做接口那岂不是会很多？这个问题问得很好，接口既要拆，但也不能拆得太细，这就得有个标准，这就是高内聚。接口应该具备一些基本的功能，能独立完成一个基本的任务。

图2-4所示只是个抽象的例子，在实际应用中，会遇到如下问题：比如，我需要一个能适配多种类型数据库的DAO实现，那么首先应实现一个数据库操作的接口，其中规定一些数据库操作的基本方法，如连接数据库、增删查改、关闭数据库等。这是一个最少功能的接口。对于一些MySQL中特有的而其他数据库不具有或性质不同的方法，如PHP里可能用到的MySQL的pconnect方法，其他数据库里并不存在和这个方法相同的概念，这个方法也就不应该出现在这个基本的接口里，那这个基本的接口应该有哪些基本的方法呢？PDO已经告诉你了。

PDO是一个抽象的数据接口层，它告诉我们一个基本的数据库操作接口应该实现哪些基本的方法。接口是一个高层次的抽象，所以接口里的方法应该是通用的、基本的、不易变化的。

还有一个问题，那些特有的方法应该怎么实现？根据ISP原则，这些方法可以在另一个接口中存在，让这个“异类”同时实现这两个接口。

对于接口的污染，可以考虑下面这两条处理方式：

利用委托分离接口。

利用多继承分离接口。

委托模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个

对象来处理，如策略模式、代理模式等中都应用到了委托的概念。至于其实现，在反射那一节其实已经实现了，这里就不再细讲了。

利用多继承分离接口，在接口一节也做了相应的讲解，这里不再重复。

2.1.3 开放-封闭原则

1.什么是“开放-封闭”

随着软件系统的规模不断增大，软件系统的维护和修改的复杂性不断提高，这种困境促使法国工程院院士Bertrand Meyer在1998年提出了“开放-封闭”（Open Close Principle, OCP）原则，这条原则的基本思想是：

Open（Open for extension）模块的行为必须是开放的、支持扩展的，而不是僵化的。

Closed（Closed for modification）在对模块的功能进行扩展时，不应该影响或大规模地影响已有的程序模块。

换句话说，也就是要求开发人员在不修改系统中现有功能代码（源代码或者二进制代码）的前提下，实现对应用系统的软件功能的扩展。用一句话概括就是：一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。

从生活中，最容易想到的例子就是电脑，我们可以轻松地对电脑进行功能的扩展，而只需通过接口连入不同的设备。

开放-封闭能够提高系统的可扩展性和可维护性，但这也是相对的，对于一台电脑不可能完全开放，有些设备和功能必须保持稳定才能减少维护上的困难。要实现一项新的功能，你就必须升级硬件，或者换一台更高性能的电脑。以电脑中的多媒体播放软件为例，作为一款播放器，应该具有一些基本的、通用的功能，如打开多媒体文件，停止播放、快进、音量调节等功能。但不论是什么播放器，不论是在什么平台下，遵循这个原则设计的播放器都应具有统一风格和操作习惯，无论换用哪一款播放器，都应保证操作者能快速上手。

以播放器为例，先定义一个抽象的接口，代码如下所示。

```
interface process {  
    public function process ();  
}
```

然后，对此接口进行扩展，实现解码和输出的功能，如代码清单2-10所示。

代码清单2-10 实现播放器的编码功能

```
class playerencode implements process {
public function process () {
echo"encode \r \n";
}
}
class playeroutput implements process {
public function process () {
echo"output \r \n";
}
}
```

对于播放器的各种功能，这里是开放的，只要你遵照约定，实现了process接口，就能给播放器添加新的功能模块。这里只实现解码和输出模块，还可以依据需求，加入更多新的模块。

接下来为定义播放器的线程调度管理器，播放器一旦接收到通知（可以是外部单击行为，也可以是内部的notify行为），将回调实际的线程处理，如代码清单2-11所示。

代码清单2-11 播放器的“调度管理器”

```
class playProcess {
private $message=null;
public function construct () {
}
public function callback ( event $ event ) {
$this->message=$event->click ();
if ( $this->message instanceof process ) {
$this->message->process ();
}
```

```
}}  
}
```

具体的产品出来了，在这里定义一个MP4类，这个类是相对封闭的，其中定义事件的处理逻辑，如代码清单2-12所示。

代码清单2-12 播放器的事件处理逻辑

```
class mp4 {  
public function work () {  
    $playProcess=new playProcess ();  
    $playProcess->callback ( new event ( ' encode ' ));  
    $playProcess->callback ( new event ( ' output ' ));  
    }  
}
```

最后为事件分拣的处理类，此类负责对事件进行分拣，判断用户或内部行为，以产生正确的“线程”，供播放器内置的线程管理器调度，如代码清单2-13所示。

代码清单2-13 播放器的事件处理类

```
class event {  
private $m;  
public function construct ($me) {  
    $this->m=$me;  
    }  
public function click () {  
    switch ($this->m) {  
    case ' encode ':  
        return new playerencode ();  
        break;  
    case ' output ':  
        }  
}
```

```
return new playeroutput ();  
break;  
}  
}  
}
```

最后，运行下下面的代码：

```
$ mp4=new mp4;  
$ mp4->work ();
```

输出结果如下：

```
encode output
```

这就实现了一个基本的播放器，此播放器的功能模块是对外开放的，但是内部处理应该是相对封闭和稳定的。但这个实现还存在一些问题，这就需要你来发现了。有时候为了降低系统的复杂性，也会不完全遵守设计模式，而是对其进行增删改。

2.如何遵守开放-封闭原则

实现开放-封闭的核心思想就是对抽象编程，而不对具体编程，因为抽象相对稳定。让类依赖于固定的抽象，这样的修改就是封闭的；而通过面向对象的继承和对多态机制，可以实现对抽象体的继承，通过复写其方法来改变固有行为，实现新的扩展方法，所以对于扩展就是开放的。

1) 在设计方面充分应用“抽象”和“封装”的思想。

一方面也就是要在软件系统中找出各种可能的“可变因素”，并将之封装起来；

另一方面，一种可变性因素不应当散落在多个不同代码模块中，而应当被封装到一个对象中。

2) 在系统功能编程实现方面应用面向接口的编程。

当需求发生变化时，可以提供该接口新的实现类，以求适应变化。

面向接口编程要求功能类实现接口，对象声明为接口类型。在设计模式中，装饰模式比较明显地用到了OCP。

2.1.4 替换原则

替换原则由MIT计算机科学实验室的Liskov女士在1987年的OOPSLA大会上的一篇文章《Data Abstraction and Hierarchy》中提出，主要阐述有关继承的一些原则，故又称里氏替换原则。

2002年，Robert C.Martin出版了一本名为《Agile Software Development Principles Patterns and Practices》的书，在书中他把里氏代换原则最终简化为一句话：“Subtypes must be substitutable for their base types”。（子类必须能够替换成它们的基类。）

1.LSP的内容

里氏替换原则（Liskov Substitution Principle, LSP）的定义和主要的思想如下：由于面向对象编程技术中的继承在具体的编程中过于简单，在许多系统的设计和编程实现中，我们并没有认真地、理性地思考应用系统中各个类之间的继承关系是否合适，派生类是否能正确地对其基类中的某些方法进行重写等问题。因此经常出现滥用继承或者错误地进行了继承等现象，给系统的后期维护带来不少麻烦。这就需要我们有一个设计原则来遵循，它就是替换原则。

LSP指出：子类型必须能够替换掉它们的父类型、并出现在父类能够出现的任何地方。它指导我们如何正确地进行继承与派生，并合理地重用代码。此原则认为，一个软件实体如果使用一个基类的话，那么一定适用于其子类，而且这根本不能察觉出基类对象和子类对象的区别。想一想，是不是和第一章提到的多态的概念比较像？

2.LSP主要是针对继承的设计原则

因为继承与派生是OOP的一个主要特性，能够减少代码的重复编程实现，从而实现系统中的代码复用，但如何正确地进行继承设计和合理地应用继承机制呢？

这就是LSP所要解决的问题：

如何正确地进行继承方面的设计？

最佳的继承层次如何获得？

怎样避免所设计的类层次陷入不符合OCP原则的状况？

那如何遵守该设计原则呢？

父类的方法都要在子类中实现或者重写，并且派生类只实现其抽象类中声明的方法，而不应当给出多余的方法定义或实现。

在客户端程序中只应该使用父类对象而不应当直接使用子类对象，这样可以实现运行期绑定（动态多态）。

如果A、B两个类违反了LSP的设计，通常的做法是创建一个新的抽象类C，作为两个具体类的超类，将A和B的公共行为移动到C中，从而解决A和B行为不完全一致的问题。

在前面的多态，继承这几节的内容里，已经涉及LSP，包括使用多态实现隐藏基类和派生类对象的区别，以及使用组合的方式解决继承中的基类与派生类（即子类）中的不符合语意的情况。PHP对LSP的支持并不好，缺乏向上转型等概念，只能通过一些曲折的方法实现。对于这个原则，这里就不再细讲了。

在接口那节提到了一个缓存的实现接口，试试用抽象类做基类，遵循LSP实现其设计。

这里给出其抽象类代码，如代码清单2-14所示。

代码清单2-14 缓存实现抽象类

```
<? php
abstract class Cache {
/**
 *设置一个缓存变量
 *
 *@param String $key 缓存Key
```

```
*@param mixed $ value 缓存内容
*@param int $ expire缓存时间 ( 秒 )
*@return boolean是否缓存成功
*/
public abstract function set ( $ key , $ value , $ expire=60 );
/**
*获取一个已经缓存的变量
*@param String $ key缓存Key
*@return mixed缓存内容
*/
public abstract function get ( $ key );
/**
*删除一个已经缓存的变量
*@return boolean 是否删除成功
*/
public abstract function del ( $ key );
/**
*删除全部缓存变量
*
*@return boolean 是否删除成功
*/
public abstract function delAll ( );
/**
*检测是否存在对应的缓存
**/
public abstract function has ( $ key );
}
```

如果现在要求实现文件、memcache、accelerator等各种机制下的缓存，只需要继承这个抽象类并实现其抽象方法即可。

现在，再来思考本书开头提到的白马非马的问题，试着用里氏替换原则阐释。

注意 LSP中代换的不仅仅是功能，还包括语意。试思考：白马可以代换马，而牛同样作为劳力，可代换马否？高跟鞋也是鞋子，男人穿高跟鞋又是否能接受？

2.1.5 依赖倒置原则

什么是依赖倒置呢？简单地讲就是将依赖关系倒置为依赖接口，具体概念如下：上层模块不应该依赖于下层模块，它们共同依赖于一个抽象（父类不能依赖子类，它们都要依赖抽象类）。

抽象不能依赖于具体，具体应该要依赖于抽象。

注意，这里的接口不是狭义的接口。

为什么要依赖接口？因为接口体现对问题的抽象，同时由于抽象一般是相对稳定的或者是相对变化不频繁的，而具体是易变的。因此，依赖抽象是实现代码扩展和运行期内绑定（多态）的基础：只要实现了该抽象类的子类，都可以被类的使用者使用。这里，我想强调一下扩展性这个概念。通常扩展性是指对已知行为的扩展，在讲述接口那一节，我也提到，接口应该是相对稳定的。这就告诉我们，无论使用多么先进的设计模式，也无法做到不需要修改代码即可达到以不变应万变的地步。在面向对象的这五大原则里，我认为依赖倒置是最难理解，也是最难实现的。

这个例子以前面提到的雇员类为蓝本，实现代码如代码清单2-15所示。

代码清单2-15 employee.php

```
<? php
interface employee {
public function working ();
}
class teacher implements employee {
public function working () {
echo 'teaching……';
}
}
class coder implements employee {
public function working () {
echo 'coding……';
}
}
```

```
}
class workA {
public function work () {
    $teacher=new teacher;
    $teacher->working ();
}
}
class workB {
private $e;
public function set ( employee $e ) {
    $this->e= $e;
}
public function work () {
    $this->e->working ();
}
}
$worka=new workA;
$worka->work ();
$workb=new workB;
$workb->set ( new teacher ());
$workb->work ();
```

在classA中，work方法依赖于teacher实现；在classB中，work转而依赖于抽象，这样可以把需要的对象通过参数传入。上述代码通过接口，实现了一定程度的解耦，但仍然是有限的。不仅是使用接口，使用工厂等也能实现一定程度的解耦和依赖倒置。

在workB中，teacher实例通过setter方法传入中，从而实现了工厂模式。由于这样的实现仍然是硬编码的，为了实现代码的进一步扩展，把这个依赖关系写在配置文件里，指明classB需要一个teacher对象，专门由一个程序检测配置是否正确（如所依赖的类文件是否存在）以及加载配置中所依赖的实现，这个检测程序，就称为IOC容器。

很多文章里看到IOC（Inversion of Control）概念，实际上，IOC是依赖倒置原则（Depend ence Inversion Principle, DIP）的同义词。而在提IOC的时候，你可能还会看到有人提起DI等概念。DI，即依赖注入，一般认为，依赖注入（DI）和依赖查找（DS）是IOC的两种实现。不过随着某些概念的演化，这几个概念之间的关系也变得很

模糊，也有人认为IOC就是DI。有人认为，依赖注入的描述比起IOC来更贴切，这里不纠缠于这几个概念之间的关系。

在经典的J2EE设计里，通常把DAO层和Service层细分为接口层和实现层，然后在配置文件里进行依赖关系的配置，这是最常见的DIP的应用。Spring框架就是一个很好的IOC容器，把控制权从代码剥离到IOC容器，这里是通过XML配置文件实现的，Spring在执行时期根据配置文件的设定，建立对象之间的依赖关系。

如下面代码所示：

```
<bean scope="prototype"
class="cn.notebook.action.NotebookListOtherAction" id="notebookListOtherAction">
  < property ref="userReplyService" name="userReplyService"/ > < property
ref="userService" name="userService"/>
  < property ref="permissionService" name="permissionService"/ > < property
ref="friendService" name="friendService"/>
</bean>
```

但是这样设置一样存在问题，配置文件会变得越来越大，其间关系会越来越复杂。同样逃脱不了随着应用和业务的改变，不断修改代码的恶魔（这里认为配置文件是代码的一部分。并且在实际开发中，很少存在单纯修改配置文件的情况。一般配置文件修改了，代码也会做相应修改）。

在PHP里，也有类似模仿Spring的实现，即把依赖关系写在了配置文件里，通过配置文件来产生需要的对象。我觉得这样的代码是还是为了实现而实现。在Spring里，配置文件里配置的不仅仅是一个类运行时的依赖关系，还可以实现事务管理、AOP、延迟加载等。而PHP要实现上面的种种特性，其消耗是巨大的。从语言层面讲，PHP这种动态脚本型语言在实现一些多态特性上和编译型的语言不同。其次PHP作为敏捷性的开发语言，更强调快速开发、逻辑清晰、代码简单易懂，如果再附加了各种设计模式的框架，从技术实现和运行效率上来看，都是不可取的。依赖倒置的核心原则是解耦。如果脱离这个最原始的原则，那就是本末倒置。

事实上，很多的设计模式里已经隐含了依赖倒置原则，我们也在有意或无意地做着一些依赖反转的工作。只是作为PHP，目前还没有一个比较完善的IOC容器，或许是PHP根本不需要。

如何满足DIP：

每个较高层次类都为它所需要的服务提出一个接口声明，较低层次类实现这个接口。

每个高层类都通过该抽象接口使用服务。

2.2 一个面向对象留言本的实例

在这一节，用面向对象的思想完成一个简单的留言本模型，这个模型不涉及实际的数据库操作以及界面显示，只是一个demo，用来演示面向对象的一些思维。

在面向过程的思维里，要设计一个留言本，一切都将以留言本为核心，抓到什么是什，按流程走下来，即按用户填写信息→留言→展示的流程进行。

现在用面向对象的思维思考这个问题，在面向对象的世界，会想尽办法把肉眼能看见的以及看不见的，但是实际存在的物或者流程抽象出来。既然是留言本，那么就存在留言内容这个实体，这个留言实体（domain）应该包括留言者的姓名、E-mail、留言内容等要素，如代码清单2-16所示。

代码清单2-16 留言实体类message.php

```
class message {
    public $ name; //留言者姓名
    public $ email; //留言者联系方式
    public $ content; //留言内容
    public function set ( $ name, $ value ) {
        $ this-> $ name= $ value;
    }
    public function get ( $ name ) {
        if ( ! isset ( $ this-> $ name )) {
            $ this-> $ name=NULL;
        }
    }
}
```

上面的类也就是所说的domain，是一个真实存在的、经过抽象的实体模型。然后，需要一个留言本模型，这个留言本模型包括留言本的基本属性和基本操作，如代码清单2-17所示。

代码清单2-17 留言本模型gbookModel.php

```
/**
 *留言本模型，负责管理留言本
 * $bookPath: 留言本属性
 */
class gbookModel {
    private $bookPath; //留言本文件
    private $data; //留言数据

    public function setBookPath ($bookPath) {
        $this->bookPath= $bookPath;
    }
    public function getBookPath () {
        return $this->bookPath;
    }
    public function open () {
    }
    public function close () {
    }
    public function read () {
        return file_get_contents ($this->bookPath);
    }
    //写入留言
    public function write ($data) {
        $this->data=self: safe ($data) ->name."&".self: safe ($data) ->email."\r\nsaid: \r\n".self: safe
        ($data) ->content;
        return
        file_put_contents ($this->bookPath,$this->data,FILE_APPEND);
    }
    //模拟数据的安全处理，先拆包再打包
    public static function safe ($data) {
        $reflect=new ReflectionObject ($data);
        $props = $reflect->getProperties (); $messagebox=new stdClass ();
        foreach ($props as $prop) {
            $ivar= $prop->getName ();
            $messagebox-> $ivar=trim ($prop->getValue ($data));
        }
    }
}
```

```
}  
return $messagebox;  
}  
public function delete () {  
file_put_contents ($this->bookPath, ' it ' s empty now ');}  
}
```

实际留言的过程可能会更复杂，可能还包括一系列准备操作以及Log处理，所以应定义一个类负责数据的逻辑处理，如代码清单2-18所示。

代码清单2-18 留言本业务逻辑处理leaveModel.php

```
class leaveModel {  
public function write ( gbookModel $gb, $data ) {  
$book= $gb->getBookPath ();  
  
$gb->write ( $data );  
//记录日志  
}  
}
```

最后，通过一个控制器，负责对各种操作的封装，这个控制器是直接面向用户的，所以包括留言本查看、删除、留言等功能。可以形象理解为这个控制器就是留言本所提供的直接面向使用者的功能，封装了操作细节，只需要调用控制器的相应方法即可，如代码清单2-19所示。

代码清单2-19 前端控制部分代码

```
class authorControl {  
public function message ( leaveModel $l, gbookModel $g, message $data ) {  
//在留言本上留言  
$l->write ( $g, $data );  
}  
}
```

```
public function view ( gbookModel $g ) {  
    //查看留言本内容  
    return $g->read ();  
}  
public function delete ( gbookModel $g ) {  
    $g->delete ();  
    echo self: view ( $g );  
}  
}
```

测试代码如下所示:

```
$message=new message;  
$message->name= ' phper ';  
$message->email= ' phper@php.net ';  
$message->content= ' a crazy phper love php so much. ';  
$gb=new authorControl (); //新建一个留言相关的控制器  
$pen=new leaveModel (); //拿出笔  
$book=new gbookModel (); //翻出笔记本  
$book->setBookPath ( "g: \ \ bak \ \ temp \ \ tempcode \ \ a.txt" );  
$gb->message ( $pen,$book,$message );  
echo $gb->view ( $book );  
$gb->delete ( $book );
```

这样看起来是不是比面向过程要复杂多了? 确实是复杂了, 代码量增多了, 也难以理解了。似乎也体现不出优点来。但是你思考过以下问题吗?

如果让很多人来负责完善这个留言本, 一部分负责实体关系的建立, 一部人负责数据操作层的代码, 这样是不是更容易分工了呢?

如果我要把这个留言本进一步开发, 实现记录在数据库中, 或者添加分页功能, 又该如何呢?

要实现上面第二个问题提出的功能, 只需在gbookModel类中添加分页方法, 代码

如下所示：

```
public function readByPage () {
    $handle=file ($this->bookPath); $count=count ($handle);
    $page=isset ($_GET['page'])? intval ($_GET['page']): 1;
    if ($page<1 || $page> $count)$page=1;
    $pnum=9;
    $begin= ($page-1) *$pnum;
    $end= ($begin+$pnum) > $count? $count: $begin+$pnum;
    for ($i=$begin; $i<$end; $i++) {
        echo ' <strong> ', $i+1, ' </strong> ', $handle[$i], ' <br/> ';
    }
    for ($i=1; $i<=ceil ($count/$pnum); $i++) {
        echo"<a href=? page= $ {i} > $ {i} </a>";
    }
}
```

然后到前端控制器里添加对应的action，代码如下所示：

```
public function viewByPage (gbookModel $g) {
    return $g->readByPage ();
}
```

运行结果如图2-5所示。

只需要这么简单的两步，就可实现所需要的分页功能，而且已有的方法都不用修改，只需在相关类中新增方法即可。当然，这个分页在实际点击时是有问题的，因为我没有把Action分开，而是通通放在一个页面里。对照着上面的思路，还可以把留言本扩展为MySQL数据库的。

在这个程序里只体现了非常简单的设计模式，这个程序还有许多要改进的地方，每个程序员心中都有一个自己的OO。项目越大越能体现模块划分、面向对象的好处。

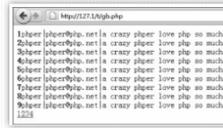


图 2-5 程序运行结果

思考 试着找找这个小程序里体现了哪些设计原则，并且试着加上一些异常处理等。

2.3 面向对象的思考

PHP的特色是简单、快速、适用。在PHP的世界里，一切以解决问题为主，所以很多设计方面的东西往往被忽视或排斥。虽然PHP的面向对象提出很多年了，但一直被排斥，很多人提倡原生态开发方式，甚至有人提倡彻底面向过程。伴随着对OO的质疑，PHP框架一方面如雨后春笋般遍地开花，另一方面一直受到抵制和质疑。

有一点是肯定的，PHP不是一门很好的面向对象的语言，因为其无法做到完全面向对象，也无法优雅实现面向对象。所以现在比较流行的还是以类为主的开放方式，即抛弃或精简经典的MVC理论，很少用和几乎不用设计模式，以类加代码模块的方式进行代码组织。这种开发方式在PHP的开源项目里是最流行的，也是最适合二次开发的，而比较纯的面向对象的产品有Zend Framework。这类产品入门的门槛比较高，代码看似“臃肿”，开发成本比较高，这类产品一般比较少见，市场占有率也比较低。

所有产品最终都是为市场服务的，PHP面向的是Web开发市场，所以并不需要高端的、复杂的设计和开发技巧。但是前面讲的那些并不是没有作用。

一些基本理论，在任何一门语言里都有共性。语法和函数库只是学好一门语言的必要条件，而不是充要条件。语法和函式只是表层的东西。只要掌握面向对象的思想，即使没有一点Java和.NET基础，也能看懂用它们写成的代码。

PHP只是一个脚本语言、一门工具而已。在Web开发中，PHP语言自身所占的分量越来越低，但却涉及程序设计的方方面面，而面向对象只是其中之一，也是最主要的一个方面。PHP是一种经典思想，能实现低耦合、易扩展的代码，其可用最经济的方式干一件事。

理论是重要的，但是理论也不是一成不变的。比如我们提到的一些设计模式，也没必要完全遵守，可以做一些精简和变形。

基于以上思考，我们认为在PHP的开发中应该灵活使用面向对象的特性和设计原则。

对于流程明确、需求清晰、需求变更风险小的业务逻辑，过程化开发（传统软件开发模式）最适合，这就像解一道数学题，总需要一步步去解，上一步的结果作为下一步的条

件。这个时候，面向过程的开发更符合人的思维。

但是对于流程复杂、需求不完善、存在很大需求变更风险的业务逻辑，此时用过程化开发将使程序变得非常的繁琐臃肿，实现难度很大，并且后期的维护代价高得惊人。此时，抽象思维将是最适合的，用面向对象的思维去抽象业务模型并随需求不断精化，最终交付使用，其扩展度和可维护性都要比过程化方法更好。

由于面向对象是更高一层的抽象，它有一些优点较之面向过程是比较突出的：

其一，新成员的加入和融合不再困难，高度抽象有利于高度总结。

其二，代码即文档，团队中的任何人都可以轻松地获得产品各个模块的基本信息，而不再需要通读大部分代码。

说到这里，可能就会有人有疑问了：本书一直在推崇面向对象的开发模式，说面向对象的好，说OO适合复杂的项目，那Linux这种复杂的项目，使用面向过程的C语言编写的，这又如何解释？

这个问题问得好，现解释如下：

其一，Linux虽然是用面向过程的C语言编写的，但是Linux的操作系统是使用内核+模块的方式构建的，这种模块化的思想是所有编程范式中的普适原则。

其二，面向对象和各种设计模式就是已经提供好的模式，使用已有的模式本比像Linux那样自己摸索出一个模式更方便快捷，开发成本更低，代码更易阅读。

其实，面向过程也好，面向对象也好，目的只有两个：一个是功能实现，一个是代码维护和扩展。只要能做好这两点，那就是成功的。

PHP不是一门很好的OOPL，但却是一门很好的Web设计语言。我们有理由相信，在Web开发领域，PHP还将继续发挥其作用，以其简单、快速吸引更多的开发者加入。

2.4 本章小结

本章主要讲解面向对象设计的五大原则，穿插一些设计模式的例子。在第1章的最后提到面向对象的设计思想存在一些问题，其本质在于面向对象强调对现实的建模，而现实和开发中并没有一一对应，因此五大原则和设计模式就是对OO的补充。

最后一节给出的留言本demo，只是一个很小的模型。一般来说，越是规模较大的项目，越能体现设计模式的前瞻性和必要性。

可能很多读者对一些设计模式有不同的见解和困惑，这是正常的。一段代码往往很难明确地归属于某一种设计模式，其可能有多种设计模式的影子。设计模式只是一种成熟的、可供借鉴的思考模式，而不是公式。

我们既要深入了解面向对象的思想，又不能执着于面向对象。

第3章 正则表达式基础与应用

正则表达式起源于科学家对人类神经系统工作原理的早期研究。美国新泽西州的Warren McCulloch和出生在美国底特律的Walter Pitts这两位神经生理方面的科学家，研究出一种用数学方式来描述神经网络的新方法，他们创新地将神经系统中的神经元描述成小而简单的自动控制元，从而做出一项伟大的工作革新。后来，数学科学家Stephen Kleene在Warren McCulloch和Walter Pitts早期工作的基础之上发表一篇论文，题目是《神经网络事件的表示法》，书中利用正则集合的数学符号描述此模型，引入正则表达式的概念。

3.1 认识正则表达式

正则表达式就是用某种模式去匹配一类字符串的一种公式。通俗地讲，就是用一个“字符串”描述一个特征，然后验证另一个“字符串”是否符合这个特征的公式。

比如“ab+”描述的特征是：一个a和任意个b。那么ab、abb、abbbbbbbbbbb都符合这个特征，而字符串ad显然是不符合的。

正则表达式可应用到各个方面。在常用的高级编辑器中，几乎都支持正则表达式，如Word、EditPlus、UltraEdit、Vim等。

正则表达式在编程语言中更是得到大规模推广。现在的语言几乎都是原生的，都可从语法上支持正则表达式，尤其在Perl的推动下，PHP、Java、.NET、JavaScript等语言都支持丰富的正则语法；不支持的可以通过一些包实现扩展。每种语言中对正则表达式的支持有所不同，其中Perl和.NET对正则表达式的支持最为强大，而JavaScript对正则表达式的支持则比较“朴素”。

注意 本节所讲的一些特性，并不是在所有语言中都支持。

3.1.1 PHP中的正则函数

正则表达式看起来总是那么古怪，以至于许多人对其望而生畏。首先要澄清一些概念：虽然不同语言间正则语法大同小异，但实际上正则表达式的实现有多种引擎（如非确定性有穷自动机NFA、确定性有穷自动机DFA），其表现又有多种风格（如JavaScript有自己的朴素正则、Perl有一套高级而强大的正则、.NET也有自己的一套正则风格）。另外，还有人可能容易混淆PHP中的preg和ereg。

简单地说，PHP中有两套正则函数，两者功能差不多：

1) 由PCRE库提供的函数，以“preg_”为前缀命名。

PCRE（Perl Compatible Regular Expression，兼容Perl的正则表达式）由Philip Hazel于1997年开发。现代的编程语言和软件中一般都使用PCRE库。

2) 由POSIX扩展提供的函数，以“ereg_”为前缀命名。

POSIX（Portable Operating System Interface of UNIX，UNIX可移植操作系统接口）由一系列规范构成，定义了UNIX操作系统应支持的功能，所以“POSIX风格的正则表达式”也就是“关于正则表达式的POSIX规范”，定义了BRE（Basic Regular Expression，基本型正则表达式）和ERE（Extended Regular Express，扩展型正则表达式）两大流派。通常UNIX的一些工具和较老的软件中会使用POSIX风格的正则。另外，一些数据库中也提供了POSIX风格的正则表达式。

自PHP 5.3以后，就不再推荐使用POSIX正则函数库，若程序中使用了则会报Deprecated级别的错误，这种情况通常在一些较老的代码中比较常见。其实使用或不使用POSIX正则函数库二者本质上没多大差别，主要是一些表现形式、语法和扩展功能的差别。

3.1.2 正则表达式的组成

在Windows资源管理器中查找文件以及批处理文件时，可使用通配符“?”和“*”表示匹配一组字符，这和正则表达式类似，“?”表示一个不确定的字符，而“*”则表示任意多个不确定字符。比如下面是删除本地垃圾文件批处理的部分代码：

```
del/f/s/q%systemdrive% \*.tmp
del/f/s/q%systemdrive% \*._mp
del/f/s/q%systemdrive% \*.log
del/f/s/q%systemdrive% \*.gid
del/f/s/q%systemdrive% \*.chk
del/f/s/q%systemdrive% \*.old
```

需要注意的是，这里的“?”和“*”称为“通配符”，而不是正则表达式。

在PHP里，一个正则表达式分为三个部分：分隔符、表达式和修饰符。

分隔符：可以是除了字母、数字、反斜线及空白字符以外的任何字符（比如/、!、#、%、|、~等）。经常使用的分隔符是正斜线（/）、hash符号（#）以及取反符号（~）。考虑到可读性，为了避免和反斜线混淆，一般不使用正斜线做分隔符。

表达式：由一些特殊字符和非特殊的字符串组成，比如“[a-z0-9_ --]+@[a-z0-9_ .]+”可以匹配一个简单的电子邮件字符串。

修饰符：用于开启或者关闭某种功能/模式。

3.1.3 测试工具的使用

在学习过程中，建议下载RegexTester工具验证和测试正则表达式，也可使用Firefox的扩展Regular Expression Tester进行测试，其界面如图3-1所示。



图 3-1 Firefox的扩展Regular Expression Tester

本书以后测试都将利用此工具进行，而不再写PHP代码测试。

注意 这个工具测试的代码不一定能在PHP中通过，反之PHP中合法的正则表达式在此工具里也不一定测试通过。其中的道理前面已经讲过了，不同语言实现的正则表达式略有区别。

下面，就来开始最简单的正则表达式入门的介绍。

3.2 正则表达式中的元字符

假设要在一篇文章里查找“he”，可以使用正则表达式“he”。这几乎是最简单的正则表达式，它可以精确匹配这样的字符串：由两个字符组成，前一个字符是“h”，后一个是“e”。通常，处理正则表达式的工具会提供一个忽略大小写的选项，如果选中这个选项，它可以匹配“he”、“HE”、“He”、“hE”这四种情况中的任意一种。

但是很多单词里包含“he”这两个连续的字符，比如“her”、“heet”等。用“he”来查找，这些单词中的“he”也会被找出来。如果要精确地查找“he”这个单词，应该使用以下形式：

```
\bhe\b
```

“\b”是正则表达式规定的一个特殊代码，代表单词的开头或结尾，也就是单词的分界处。虽然通常英文单词是由空格、标点符号或者换行来分隔，但是“\b”并不匹配这些单词分隔字符中的任何一个，它只匹配一个位置。

“\b”匹配位置的精确说法：前一个字符和后一个字符不全是（一个是，一个不是或不存在）“\w”。

假如要找“he”后面不远处跟着一个“is”，应该表示如下：

```
\bhe\b.*\bis\b
```

这里，点号(.)是元字符，匹配除了换行符以外的任意字符。“*”同样是元字符，不过它代表的不是字符，也不是位置，而是数量——它指定“*”前边的内容可以连续重复使用任意次以使整个表达式得到匹配。因此，“.”和“*”连在一起就意味着任意数量的、不包含换行的字符。现在，“\bhe\b.*\bis\b”的意思很明显：先是一个单词

he，然后是任意个任意字符（但不能是换行符），最后是is这个单词。

3.2.1 什么是元字符

元字符（Meta Characters）是正则表达式中具有特殊意义的专用字符，用来规定其前导字符（即位于元字符前面的字符）在目标对象中的出现模式。通过前面的例子，我们已经知道几个很有用的元字符。正则表达式里有很多元字符，常用元字符如表3-1所示。

元字符	描述
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
\$	匹配字符串的结束
-	表示范围
[]	匹配括号中的任意一个字符
*, +, ?	量词

下面看一些例子。

1) 匹配以字母“a”开头的单词：

```
\ba\w*\b
```

以上表达式先是某个单词开始处（\b），然后是字母“a”，接着是任意数量的字母或数字（\w*），最后是单词结束处（\b），匹配的单词如adandon、action、a等。

2) 匹配1个或更多连续的数字：

```
\d+
```

以上表达式可以匹配0、1、555等。这里的元字符+和*类似，不同的是，*匹配重复任意次（可能是0次），而+则匹配重复1次或更多次。

3) 匹配刚好6个字符的单词：

```
\b\w{6}\b
```

以上表达式匹配action、123456、ste_ph等。

注意 正则表达式里“单词”指不少于1个的连续字母和数字。

如果同时使用其他元字符，则能构造出功能更强大的正则表达式。比如下面这个例子：

```
0\d\d-\d\d\d\d\d\d\d\d
```

匹配字符串：以0开头，然后是2个数字，1个连字符，最后是8个数字，也就是中国部分地区的电话号码，如010 12345678。

这里“\d”是元字符，匹配1位数字（0、1、2……）。“-”不是元字符，只匹配它本身——连字符（或者减号，或者中横线，或者随你怎么称呼它）。

为了避免那么多烦人的重复，也可以这样写这个表达式：

```
0\d{2}-\d{8}
```

这里\d后面{2}和{8}的意思是，前面\d必须连续重复匹配2次和8次。

思考题 使用“he”、“\bhe\b”分别查找句子“he is a good student, the most proud of his mother. With him, she hold the hope.”有多少种匹配结果？

下面重点介绍几个常用元字符。

3.2.2 起始和结束元字符

元字符中有两个用来匹配位置：

^：匹配字符串的开始。

：匹配字符串的结束。

元字符“^”、“”与“\b”有点类似。“^”匹配字符串的开头，“”匹配结尾。这两个代码在验证输入内容时非常有用，比如某网站如果要求填写QQ号必须为5~11位数字时，可以使用：

```
^ \d{5, 11}$
```

这里 { 5, 11 } 表示重复次数不能少于5次，不能多于11次，否则都不匹配。因为使用“^”和“”，所以输入的整个字符串都要和 \d { 5, 11 } 匹配。也就是说，整个输入必须是5~11个数字，如果输入QQ号能匹配这个正则表达式，就符合要求。如果输入含有5~11个数字，但不是完整数字串，而只是一串字符的一部分，也不能匹配成功，如图3-2所示。



图 3-2 正则表达式匹配结果

从图中就能清晰地看出“^ \d { 5, 11 }”的确切含义。我想，你也能猜测到它和正则表达式 \d { 5, 11 } 的区别。为了加深印象，分别使用下面4个正则表达式看一下效果：

```
^ \d { 5, 11 } $ //匹配起始和结束位置都是数字的，且连续5~11位
```

```
\d{5,11}$ //匹配结束位置是数字的,且连续5~11位  
^\d{5,11} //匹配起始位置是数字的,且连续5~11位  
\d{5,11} //匹配连续的5~11位数字
```

很自然,在一行中,前三个正则表达式结果只可能有一个匹配结果,而最后一个正则表达式则可以有多于一个匹配成功的结果。因为一行只可能有一个开始位置和一个结束位置。

注意 我们在正则表达式处理工具处勾选Multiline选项,即多行选项,^和的意义就变成匹配行的开始处和结束处,否则将把整个输入视作一个字符串,忽视换行符。可以试着把多行选项去除后再看看效果。如果用过Vim编辑器,就知道命令“d^”和“d”的作用了。

3.2.3 点号

点号 (.) 是使用频率最高的元字符。例如，在做采集时抓取页面，要匹配某DIV里的内容，就需要用到点号匹配。下面代码是抓取本地HTML页面的一部分：

```
<html xmlns="http://www.w3.org/1999/xhtml">  
<head profile="http://gmpg.org/xfn/11">  
<meta http-equiv="content-type"content="text/html; charset=UTF-8"/>  
<title>我的博客</title>
```

要匹配这个网页的标题应该怎么办呢？很简单，使用点号匹配全部字符，如下：

```
<title>.*< \ /title>
```

这样就可以抓取你想要的任何内容了，包括DIV、SPAN等。

思考题 延伸思路，是不是还可以抓取页面的字符集？要判断这个页面有多少张图片是不是也很容易？只要找到特征字符就可以。试一下，看看和预想的结果是否一致。

3.2.4 量词

前面实际上已经涉及量词的概念，比如 `\d+`、`\d{5,11}` 等都应用了量词。正则表达式中的量词如表3-2所示。

下面是一些例子：

1) 匹配Windows后面跟1个或多个数字：

```
Windows \d+
```

2) 表示index后面紧跟0个或1个数字，：

```
index \d?
```

以上表达式匹配index、index1、index9这样的文件名，但不匹配index10、indexa这样的文件名。

3) 匹配一行第一个单词（或整个字符串第一个单词，具体匹配哪种，得看选项设置）：

限定符代码/语法	描 述
*	重复 0 次或更多次
+	重复 1 次或更多次
?	重复 0 次或 1 次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次

```
^ \w+
```

提示 在学习量词的过程中，要注意*和?这两个量词。前面提到过通配符的概念，通配符里也有这两个符号，要注意它们之间的区别。

3.3 正则表达式匹配规则

我们已经学习“*”、“-”、“?”等元字符，它们都有各自的特殊含义。如果想匹配没有预定义元字符的字符集合，或者表达式和已知定义相反，或者存在多种匹配情况，应该怎么办？本节就介绍几种常用匹配规则。

3.3.1 字符组

查找数字、字母、空白很简单，因为已经有了对应这些字符集合的元字符，但是如果匹配没有预定义元字符的字符集合（比如元音字母a、e、i、o、u），方法很简单，只需要在方括号里列出它们。

例如[aeiou]匹配任何一个英文元音字母，[.?!]匹配标点符号（“.”、“?”或“!”），c[aou]t匹配“cat”、“cot”、“cut”这三个单词，而“caout”则不匹配。

注意[]匹配单个字符，尽管看起来[]里有好多字符。

也可以指定字符范围，例如[0-9]的含意与\d完全一致：代表一位数字；同理[a-zA-Z_]完全等同于\w（如果只考虑英文）。

字符组很简单，但是一定要弄清楚字符组中什么时候需要转义。

3.3.2 转义

如果想查找或匹配元字符本身，比如查找*、? 等就出现问题：没办法指定，因为它们会被解释成别的意思。这时就使用 \ 来取消这些字符的特殊意义。因此，应该使用 \.和 *。当然，查找 \ 本身用 \\。这叫做转义。

通俗地讲，转义就是防止特殊字符被解析，或者说用某个符号表示另一个特殊符号。例如：unibetter \.com匹配unibetter.com, C: \ \ Windows匹配C: \ Windows。

在JavaScript或者PHP中都接触过转义的概念。例如，JavaScript中要弹出一个对话框，对话框中需要分成两行显示，用HTML的
 标签或者在源代码里手工换行都不行，应该用 \r \n表示换行并新起一行，如下所示：

```
alert ("警告:  
操作无效"); //错误  
alert ("警告<br>操作无效"); //错误  
alert ("警告\r\n操作无效"); //正确写法
```

在PHP里使用反斜杠 (\) 表示转义， \Q和 \E也可以在模式中忽略正则表达式元字符，比如：

```
\d+\Q.$\E$
```

以上表达式先匹配一个或多个数字，紧接着一个点号，然后一个，再然后一个点号，最后是字符串末尾。也就是说， \Q和 \E之间的元字符都会作为普通字符用来匹配。

正则表达式是不是遇到这些特殊字符就该转义呢？答案显然是否定的。转义只有在一定条件下，比如可能引起歧义或者被误解析的情况下才需要。有些情况并不需要转义这些“特殊”字符，并且在时转义也是无效的。这需要不断尝试并积累经验。看一个例子：

```
<? php
$reg="#[aby \ ]#";
$str= ' a \ bc[] {} ';
preg_match_all ($reg,$str,$m);
var_dump ($m);
```

在字符组中匹配“a”、“b”、“y”和“}”中任意一个，由于“}”是元字符，具有特殊意义，所以这里进行转义，使用“\ {}”表示“{}”。

但是实际上，这个转义是多余的。虽然“}”是元字符，具有特殊意义，但是在字符组中，“}”却无法发挥意义，不会引起歧义，所以不需要转义。在这里“\ {}”和“{}”是等价的。

既然转义符“\”是多余的，那么会不会被当作普通字符呢？字符串str里有“\”，但是可以从代码运行结果中看出，“\”字符并没有被匹配，也就是说正则表达式“#[abc \]#”中，虽然“\”转义符是多余的，但是也并没有被当作普通字符进行匹配。

如果确实要把“\”当作普通字符匹配，正则表达式需要写成：

```
#[} ab \ \ \ y]#
```

前面提到，不是所有出现特殊字符的地方都要转义。例如，以下正则表达式可以匹配“cat”、“c? t”、“c) t”等字符：

```
c[aou? *)]t
```

其中“?”和“*”等特殊字符都不需要转义。原因很简单，字符组里匹配的是单个字符，这些特殊字符不会引起歧义。

字符组里可以使用转义吗？可以，例如“c[\ d]d”可以匹配“c1d”、“c2d”等。

下面是复杂的表达式：

```
\ (? 0\d{2}[-])? \d{8}
```

“(”和“)”也是元字符（后面在分组章节会提到），所以在这里需要使用转义。这个表达式可以匹配几种格式的电话号码，例如（010）88886666、022 22334455或02912345678等。首先是转义符“\（”，表示出现0或1次（？），然后是一个0，后面跟着两个数字（\d{2}），然后是“)”、“-”或空格中的一个，出现1次或不出现（？），最后是八个数字（\d{8}）。

3.3.3 反义

有些时候，查找的字符不属于某个字符类，或者表达式和已知定义相反（比如除了数字以外其他任意字符），这时需要用到反义。常用反义如表3-3所示。

常用反义	描述
\W	匹配任意不是字母、数字、下划线、汉字的字符
\S	匹配任意不是空白符的字符
\D	匹配任意非数字的字符
\B	匹配不是单词开头或结束的位置
[^x]	匹配除了 x 以外的任意字符
[^a-z0-9]	匹配除了 a-z0-9 这几个字母以外的任意字符

反义有一个比较明显的特征，就是和一些已知元字符相反，并且为大写形式。比如“\d”表示数字，而“\D”就表示非数字。看一些实际的例子。

1) 不包含空白符的字符串:

```
\S+
```

2) 用尖括号括起来、以a开头的字符串:

```
<a[^>]+>
```

比如，要匹配字符串“百度”，这个正则表达式匹配的结果就是“”。

提示“^”在这里是“非”的意思，不要和表示开头的“^”混淆。那怎么区分呢？很简单，表示开始位置的“^”只能用在正则表达式最前端，而表示取反的“^”只用在字符组中，即只在中括号里出现。记住这一点，就不会搞混了。

日常工作中反义用得不多，因为扩大了范围。例如程序里的变量，第一个字符不允许是数字，一般使用“^[a-zA-Z_]”表示，而不会使用“\D”，因为“\D”扩大了范围，包括所有非数字的字符，显然，变量命名不仅仅要求第一个字符不是数字，也不能是

其他除了26个大小写字母和下画线以外的字符。因此，不要随意使用反义，以免无形中扩大范围，而使自己没有考虑到。

3.3.4 分支

分支就是存在多种可能的匹配情况。例如，匹配“cat”或者“hat”，可以写成`[ch]at`；要匹配“cat”、“hat”、“fat”、“toat”，很显然不能用字符组匹配的方式。这里表明前面的匹配字符可以是c、h、f或者to，而`[]`只能匹配单个字符，此时可用分支形式，即：

```
(c|h|f|to)at
```

其中括号里的表达式将视作一个整体（后面会讲到分组的概念），“|”表示分支，即可能存在的多种情况，可以匹配多个字符。分支的功能更强大，字符组方式只能对单个字符“分支”，而分支可以是多个字符以及更复杂的表达式。但对于单字符的情况，字符组的效率更高。也就是说，能使用字符组就不用分支。

看到这里，你可能会疑问：表达式“`[ch]at`”括号里面是可能的匹配，分支也是表示可能的匹配，那么“`[ch]at`”是否可以写成“`(c|h)at`”呢？答案显然是可以的，“`[ch]at=(c|h)at`”。

注意 括号匹配会捕获文本，如果不需要捕获文本，上面的例子可以使用“`(?:)`”，后面还会讲到。

正则表达式分支条件指有几种规则，无论满足其中哪一种规则都能匹配，具体方法是用“|”把不同规则分隔开，例如：

```
0\d{2}-\d{8}|0\d{3}-\d{7}
```

这个表达式能匹配两种以连字号分隔的电话号码：一种是3位区号，8位本地号（如010-12345678），一种是4位区号，7位本地号（如0376-2233445）。匹配3位区号的电话号码表达式如下：

```
\ (0\d{2} \)[-? \d{8} | 0\d{2}][-? \d{8}]
```

其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔。可以试试用分支条件把这个表达式扩展成同时支持4位区号。

例如，美国邮编规则是5位数字，或者用连字号间隔的9位数字。匹配表达式如下：

```
\d{5}-\d{4} | \d{5}
```

另外，使用分支条件时，要注意各个条件的顺序。如果改成以下形式，就只匹配5位邮编以及9位邮编的前5位：

```
\d{5} | \d{5}-\d{4}
```

注意 匹配分支条件时，将从左到右测试每个条件，如果满足某个分支，就不会再考虑其他条件。

3.3.5 分组

重复单个字符只需要直接在字符后面加上限定符，但如果想重复多个字符又该怎么办呢？可以用小括号指定子表达式，然后规定这个子表达式的重复次数，也可以对子表达式进行其他一些操作。这就是本节介绍的分组，常用分组语法如表3-4所示。

类别	代码/语法	描述
捕获	<code>(exp)</code>	匹配 <code>exp</code> ，并捕获文本到自动命名的组里
	<code>(?<name>exp)</code>	匹配 <code>exp</code> ，并捕获文本到名称为 <code>name</code> 的组里，也可以写成 <code>(?name=exp)</code>
	<code>(?:exp)</code>	匹配 <code>exp</code> ，不捕获匹配的文本，也不给此分组分配组号
零宽断言	<code>(?=exp)</code>	匹配 <code>exp</code> 前面的位置
	<code>(?<exp)</code>	匹配 <code>exp</code> 后面的位置
	<code>(?!exp)</code>	匹配后面跟的不是 <code>exp</code> 的位置
	<code>(?<!exp)</code>	匹配前面不是 <code>exp</code> 的位置
注释	<code>(?comment)</code>	提供注释辅助阅读，不对正则表达式的处理产生任何影响

例如，简单的IP地址匹配表达式如下：

```
(\d{1,3}\.){3}\d{1,3}
```

要理解以上表达式，应按下列顺序分析：

1) 匹配1~3位的数字：

```
\d{1,3}
```

2) 匹配3位数字加上1个英文句号（分组），重复3次（最后加上一个1~3位的数字）：

```
(\d{1,3}\.){3}
```

IP地址中每个数字都不能大于255，所以严格来说这个正则表达式是有问题的。因为它将匹配256.300.888.999这种不可能存在的IP地址。如果能使用算术比较，或许能简

单地解决这个问题，但是正则表达式中没有提供关于数学的任何功能，所以只能使用冗长的分组、选择、字符类来描述一个正确IP地址，如下所示：

```
((2[0-4]\d|25[0-5]|[01]? \d\d?)\.) {3} (2[0-4]\d|25[0-5]|[01]? \d\d?)
```

思考题 理解这个表达式的关键是理解“2[0-4]\d|25[0-5]|[01]? \d\d?”，读者应该能分析出它的意义。

默认情况下，每个分组会自动拥有一个组号，规则是：从左向右，以分组的左括号为标志，第一个出现的分组，其组号为1，第二个为2，以此类推；分组0对应整个正则表达式。

也可以自己指定子表达式的组名，语法如下：

```
? <Word> \w+
```

把尖括号换成单引号也行，如下所示：

```
? 'Word' \w+
```

这样就把 \w+组名指定为Word。

提示 组号分配远没有这么简单。组号分配过程是要从左向右扫描两遍：第一遍只给未命名组分配，第二遍只给命名组分配。因此，所有命名组的组号都大于未命名的组号。可以使用语法 (? : exp) 剥夺一个分组对组号分配的参与权。

3.3.6 反向引用

反向引用用于重复搜索前面某个分组匹配的文本。首先看示例，“\1”代表分组1匹配的文本：

```
\b(\w+)\b\s+\1\b
```

以上表达式可以匹配重复的单词，例如go go或者kitty kitty。首先这个表达式是一个单词，也就是单词开始处和结束处之间大于一个的字母或数字，即“\b(\w+)\b”，这个单词会被捕获到编号为1的分组中，然后是1个或几个空白符(\s+)，最后是分组1中捕获的内容（也就是前面匹配的那个单词），即\1，这样就相当于把所匹配的重复一次。

要反向引用分组捕获的内容，可以使用“\k<Word>”，所以上个例子也可以写成这样：

```
\b(?<Word>\w+)\b\s+\k<Word>\b
```

例如，要捕获字符串“\ "This is a ' string ' \ ”引号内的字符，如果使用以下正则表达式：

```
(\"|').*?(\"|')
```

将返回“"This is a ' ”。显然，这并不是我们想要的内容。这个表达式从第一个双引号开始匹配，遇到单引号之后就错误地结束匹配。这是因为表达式里包含"|'，也就是双引号(")和单引号(')均可。要修正这个问题，可以用到反向引用。

表达式“\1, \2, ……， \9”是对前面已捕获子内容的编号，可以作为对这些编

组的“指针”引用。在此例中，第一个匹配的引号就由1代表。可以这么写成：

```
(" | \ ' ) .*? \ 1
```

如果使用命名捕获组，可以写成：

```
(? P <quote> " | ' ) .*? (? P =quote )
```

看PHP使用反向引用的例子。

在很多论坛中都会看到UBB标签代码。UBB标签最早的设计是用来在论坛和留言本里代替HTML，实现一些简单的HTML效果，同时防止滥用HTML出现安全问题。例如，HTML中粗体的标签是：

```
<b>粗体</b>
```

或者：

```
<strong>粗体</strong>
```

而UBB标签则是：

```
[b]粗体[/b]
```

UBB标签以其更好的安全性，目前已经成为论坛发帖的代码标准，只不过不同论坛产品的叫法不一样而已。

最终，UBB标签还是要解析成HTML代码，才能让浏览器认识。这个过程是怎样实现的呢？下面以URL标签为例解释。

例如，UBB标签 “[url]1.gif[/url]” 用于插入表情。在解析时，需要把1.gif换成实际路径，并且需要用HTML的IMG标签进行替换，方法如下所示：

```
<? php
$str= ' [url]1.gif[/url][url]2.gif[/url][url]3.gif[/url] ';
$s=preg _ replace ( "# \ [url \ ] ( ? < WORD > \ d \ .gif ) \ [ \ /url \ ]#" , " < img
src=http: //image.ai.com/upload/ $1
> ", $str );
var _ dump ( $s );
```

运行结果如下：

```
string ( 141 ) "<img src=http: //image.ai.om/upload/1.gif >
<img src=http: //image.ai.com/upload/2.gif >
<img src=http: //image.ai.com/upload/3.gif > "
```

是不是很简单？一个简易表情标签就这样实现了。

这里再给出一个表达式实现同样的效果：

```
<? php
$str= ' [url]1.gif[/url][url]2.gif[/url][url]3.gif[/url] ';
$s=preg _ replace ( "# \ [url \ ] ( .*? ) \ [ \ /url \ ]#" , " < img
src=http: //image.ai.com/upload/ $1 > ", $str );
var _ dump ( $s );
```

提示 这个正则表达式涉及贪婪/懒惰匹配知识，后面会进一步介绍。

3.3.7 环视

断言用来声明一个应该为真的事实。正则表达式中，只有当断言为真时才会继续进行匹配。断言匹配的只是一个事实，而不是内容。本节介绍四个断言，它们用于查找在某些内容（但并不包括这些内容）之前或之后，也就是一个位置（如 `\b`、`^`、`.`）应该满足的一定条件（即断言），因此也称为零宽断言。

1. 顺序肯定环视 (`? =exp`)

零宽度正预测先行断言，又称顺序肯定环视，断言自身出现位置的后面能匹配表达式 `exp`。

比如，匹配以“ing”结尾的单词前面部分（除了“ing”以外的部分）：

```
\b\w+(?=ing\b)
```

以上表达式查找以下句子时，会匹配“sing”和“danc”：

```
I ' m singing while you ' re dancing.
```

2. 逆序肯定环视 (`? <=exp`)

零宽度正回顾后发断言，又称逆序肯定环视，断言自身出现位置的前面能匹配表达式 `exp`。

比如，以re开头的单词的后半部分（除了re以外的部分）：

```
(? <= \bre) \w+ \b
```

以上表达式在查找以下句子时匹配“ading”:

reading a book

假如在很长的数字中，每3位间加1个逗号（当然是从右边加起），可以在前面和里面添加逗号的部分：

```
((? <= \d) \d{3})+ \b
```

用以上表达式对“1234567890”进行查找，结果是“， 234， 567， 890”。这里的逗号只是匹配需要添加逗号的位置，还没有实际添加逗号。

下面这个例子同时使用这两种断言，匹配以空白符间隔的数字（再次强调，不包括这些空白符）：

```
(? <= \s) \d+ (? = \s)
```

前面提到过反义，用来查找不是某个字符或不在某个字符类里的字符。如果只是想要确保某个字符没有出现，但并不想去匹配它时怎么办？例如，如果想查找这样的单词——出现字母q，但是q后面跟的不是字母u。可以尝试这样：

```
\b \w*q[ ^u] \w* \b
```

以上表达式匹配包含后面不是字母u的字母q的单词。但是如果多做几次测试就会发现，如果q出现在单词的结尾，例如Iraq、Benq，这个表达式就会出错。这是因为[^u]

总要匹配一个字符，如果q是单词的最后一个字符，后面的 “[^u]” 将会匹配q后面的单词分隔符（可能是空格、句号或其他），后面的 “\w*\b” 将会匹配下一个单词，于是以上表达式就能匹配整个Iraq fighting。

逆序肯定环视能解决这样的问题，因为它只匹配一个位置，并不消费任何字符。现在，解决这个问题如下所示：

```
\b\w*q(?:u)\w*\b
```

3.顺序否定环视(?:exp)

零宽度负预测先行断言，又称顺序否定环视，断言此位置的后面不能匹配表达式“exp”。例如：

1) 匹配3位数字，而且这3位数字的后面不能是数字：

```
\d{3}(?!\d)
```

2) 匹配不包含连续字符串abc的单词：

```
\b(?:abc)\w*\b
```

如果匹配的单词是c开头、t结尾，中间有一个字符，但不能是u（也就是说，整个单词不能是cut），直接用“c[^u]t”就可以了，若中间的字符不能是a或u（也就是说，整个单词不能是cat或cut），则表达式改为“c[^au]t”。

如果认真读过关于排除型字符组的章节的读者肯定会知道，这个表达式能匹配的只是cot之类的单词，因为中间的排除型字符组“[^au]”必须匹配一个字符。可是，如果还

想匹配chart、conduct和court怎么办？最简单的想法是：去掉排除型字符组的长度限制，改成“c[^au]+t”。

不幸的是，这样行不通，因为这个表达式的意思是：c和t之间由多于一个“除a或u之外的字符”构成，而chart、conduct和court都包含a或u。

我们发现，其实要否定的是“单个出现的a或u”，而不仅仅是“出现的a或u”，所以才出现这样的问题。要解决这个问题，就应当把意思准确表达出来，变成“在结尾的t之前，不允许只出现一个a或u”。想到这一步，就可以用顺序否定环视(?! ……)来解决。表示在这个位置向右，不允许出现子表达式能够匹配的文本，把子表达式规定为“[au]t\b”（最后的“\b”很重要，它出现在t之后，保证t是单词的结尾字母）。有了限制，匹配a和t之间文本的表达式就随意很多，可以用匹配单词字符的简记法“\w”表示，于是整个表达式变成：

```
c(?! [au]t\b) \w+t
```

注意 这里出现的并不是排除型字符组“[^au]”，而是普通的字符组[au]，因为顺序否定环视本身已经表示了否定。

进一步思考，整个匹配文本中都不能出现字符串“cat”，要怎么办呢？这个正则表达式应该是：

```
^(?: (?! cat) .) +$
```

即在文本中的任意位置，都不能出现该字符串。

4. 逆序否定环视(? <! exp)

零宽度负回顾后发断言，又称逆序否定环视，可以用(? <! exp)断言此位置的前面不能匹配表达式exp。例如，前面不是小写字母的7位数字：

```
(? <! [a-z]) \d{7}
```

分析以下表达式，匹配不包含属性的简单HTML标签内的内容：

```
(? <=< (\w+) >).* (? =< \/\1>)
```

以上表达式最能表现零宽断言的真正用途。`(? (\w+) >)` 指定前缀为：被尖括号括起来的单词（比如可能是“”），然后是“.”（任意的字符串），最后是一个后缀 `(? =< \/\1>)`。注意后缀里的“\”，用到了前面提过的字符转义；“\1”则是反向引用，引用的正是捕获的第一组，即前面 `(\w+)` 匹配的内容，如果前缀实际上是“”，后缀就是“”。整个表达式匹配的是“”和“”之间的内容（再次提醒，不包括前缀和后缀本身）。

总体而言，环视相当于对“所在位置”附加一个条件，难点就在于找到这个“位置”。这一点解决了，环视就没有什么秘密可言了。

3.3.8 贪婪/懒惰匹配模式

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符。例如以下表达式将匹配以a开始，以b结束的最长字符串：

```
a.*b
```

如果用来搜索“aabab”，它会匹配整个字符串“aabab”。这就是贪婪匹配。

有时，需要匹配尽可能少的字符，也就是懒惰匹配。前面给出的限定符都可以转化为懒惰匹配模式，只要在后面加上一个问号。例如“.*?”就意味着匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复。例如，匹配以a开始、以b结束的最短字符串，正则表达式如下：

```
a.*? b
```

把上述表达式应用于aabab，如果只考虑“.*?”这个表达式，最先会匹配到aab（1~3字符）和ab（第2~3个字符）这两组字符。

为什么第一个匹配是aab（第1~3个字符）而不是ab（第2~3个字符）？简单地说，正则表达式有另一条规则，比懒惰/贪婪规则的优先级更高：最先开始的匹配拥有最高优先权。

常用懒惰限定符如表3-5所示。

懒惰限定符代码/语法	描述
*?	重复任意次，但尽可能少重复
+?	重复1次或更多次，但尽可能少重复
??	重复0次或1次，但尽可能少重复
{n,m}?	重复n到m次，但尽可能少重复
{n,}?	重复n次以上，但尽可能少重复

懒惰模式匹配原理简单来说，是在匹配和不匹配都可以的情况下，优先不匹配，记录备选状态，并将匹配控制交给正则表达式的下一个匹配字符。当后面的匹配失败时，回溯，进行匹配。关于回溯以及正则表达式效率等高级内容，可以查阅《精通正则表达式》一书。

在3.3.6节涉及懒惰匹配，把该节的例子稍作更改：

```
<? php
$str= '[url]1.gif/[url][url]2.gif/[url][url]3.gif/[url] ';
$ s=preg _ replace ( "# \ [url \ ] ( .* ) \ [ \ /url \ ]#" , " < img
src=http: //image.aiyooyo.om/upload/$1>", $ str);
var _ dump ( $ s);
```

在贪婪模式下，由于匹配表达式是“.*”，即任意字符出现任意次，这个正则表达式会一直匹配[url]后的内容，直到遇到结束条件“[/]”。匹配结果如图3-3所示。

图 3-3 运行结果

提示 实际开发中，涉及贪婪模式与懒惰模式的地方是很多的。在一定情况下，使用懒惰模式可以减少回溯，提高效率。

3.4 构造正则表达式

在构造和理解正则表达式的过程中，通常都是由简到繁的过程，如果理解正则表达式内部间的关系，就可以把比较复杂的正则表达式拆分成几个小块来理解，从而帮助消化。

3.4.1 正则表达式的逻辑关系

正则表达式之间的逻辑关系可以简单地用与、或、非来描述，如表3-6所示。

逻辑关系	描 述
与	在某个位置，某些元素（字符、字符组或者子表达式）必须出现
或	在某个位置，某个元素或许出现，或许不出现；或长度和出现次数不固定，或者是某几个元素中的一个
非	在某个位置，某些元素不出现

通常来说，正则表达式可以看做这三种逻辑关系的组合。下面分析这三种逻辑。

1. 与

“与”是正则表达式中最普遍的逻辑关系。一般来说，如果正则表达式中的元素没有任何量词（比如*、?、+）修饰，就是“与”关系。比如正则表达式：

```
abc
```

表示必须同时出现a、b、c三个字符。

连续字符是“与”关系的最佳代表。此外，有些环视结构也可以表达“与”关系。比如顺序肯定环视（?=exp）表示自身出现的位置后面能匹配表达式exp，换而言之，就是在它后面必须出现表达式exp。例如：

```
\w+(?=ing)
```

表示单词的后面必须是ing结尾。

除了顺序肯定环视外，逆序肯定环视也能表达“与”关系。

比如匹配DIV标签里的内容，例如<div>logo</div>中的logo，就可用以下正则表达式来匹配：

```
(? <=<div>) .* (? =</div>)
```

“(? <=<div>)”表示自身（即要匹配的部分）出现的位置前面匹配表达式“<div>”，“(? =</div>)”表示它的后面需要匹配表达式“</div>”，中间的“.*”就是匹配到的内容。

2.或

“或”是正则表达式中容易出现的逻辑关系。

如果“或”代表元素可以出现，也可以不出现，或者出现的次数不确定，可以用量词来表示“或”关系。比如以下表达式表示在此处，字符a可以出现，也可以不出现：

```
?a? ?
```

以下表达式表示在此处，字符串ab必然要出现1次，也可以出现无限多次：

```
[(ab) +]
```

如果“或”表示出现的是某个元素的一个，那么可以使用字符组。比如以下正则表达式表示此处出现的字符是a、b、c中的任何一个：

```
[abc]
```

如果要匹配多个字符，则使用分支结构 (…… |……)。比如匹配单词foot及其复数形式，就可以用正则表达式：

```
f(oo|ee)t
```

或者使用以下形式：

```
f[oe]{2}t
```

3.非

提到“非”，最容易想到正则表达式中的反义和“^”元字符。比如“\d”表示数字，那么其对应的\D就表示非数字；[a]表示a字符，那么[^a]就表示这个字符不是a。

“非”关系最常用来匹配成对的标签，例如双引号字符串的匹配，首尾两个双引号很容易匹配，其中的内容肯定不是双引号（暂不考虑转义的情况），所以可以用[^"]表示，其长度不确定，用*来限定，所以整个表达式如下：

```
[^"]*
```

比如，需要匹配HTML里成对的A标签，先匹配左尖括号，紧接着是a字符，后面可以是任意内容，最后是一个右尖括号。在这对括号之间可以出现空格、字母、数字、引号等字符，但是不能出现“>”字符，于是就可以用排除型字符组“[^>]”来表示。再加上后面的配对标签，整个表达式如下：

```
<a[ ^ >]*>.*< \ /a>
```

运行下面这段代码验证这个表达式:

```
<? php
$reg="#<a[ ^ >]*> (.*) < \ /a>#";
$str= ' <a href="http: //baidu.com">baidu< /a> some<a href="http: //sohu.com">sohu< /a
> ';
preg_match_all ($reg,$str,$m);
var_dump ($m);
```

运行结果如下:

```
array (2) {
  [0]=>
  array (1) {
    [0]=>
    string (74) "<a href="http: //baidu.com">baidu< /a> some<a href="http: //sohu.com">
sohu< /a>"
  }
  [1]=>
  array (1) {
    [0]=>
    string (43) "baidu< /a> some<a href="http: //sohu.com">sohu"
  }
}
```

发现结果不符合预期，出现了嵌套匹配。原因在于A标签之间的文本忘了做排除型匹配，于是修改后的正则表达式就成了“<a[^ >]*> ([^ < >]*) < \ /a>”。经过修改后就符合预期了。

除了反义和排除型字符组外，否定环视也能表示“非”这种关系。比如有一串文字：“ab<p>onecde<div>fgh</div>”。现在需要匹配除P标签外的所有标签。换言之，就是先匹配所有HTML标签，可以使用如下表达式：

```
</? \b[^\>]+>
```

匹配闭合的“<XXX>”或“</XXX>”标签，然后再排除“XXX”或“/XXX”部分是P的标签，于是使用顺序否定环视，用表达式：

```
(?! /? p\b)
```

排除了“<”或“</”这个位置后是P字符的情况，这样就满足需求了。最终的表达式则为：

```
<(?! /? p\b)[^\>]+>
```

通过上面的分析得到正则表达式中的“与或非”关系及其代表语法，如表3-7所示。

逻辑关系	代表语法
与	连续字符、肯定环视（顺序肯定、逆序肯定）
或	量词、字符组
非	排除型字符、反义、否定环视（顺序否定、逆序否定）

3.4.2 运算符优先级

正则表达式从左到右进行计算，并遵循优先级顺序，这与算术表达式非常类似。表3-8说明了各种正则表达式运算符的优先级顺序，其中优先级从上到下、由高到低排列。

运算符	描述
\	转义符
()、(?:)、(?=)、[]	括号和中括号
*, +, ?, , n , [n], [n, m]	限定符
^, \$, \b, \B, \d, \D, \w, \W, \s, \S	定位点和序列
.	替换

字符的优先级比替换运算符高，替换运算符允许 `m | food` 与 `m` 或 `food` 匹配。要匹配 `mood` 或 `food`，使用括号创建子表达式，从而产生如下表达式：

```
(m | f) ood
```

3.4.3 正则表达式的常用模式

模式 (Pattern Modifiers) 就是可以改变表达行为的字符, 用来关闭或打开某些特殊功能, 习惯上又称为正则修饰符。每种语言提供的正则表达式所支持的修饰符都不一样, 这节介绍一些基本修饰符及常用模式。

1. 忽略大小写模式 (i)

在此模式下, 正则匹配将不区分待匹配内容的大小写, 这在HTML里常用。由于HTML本身的容错性很好, 对大小写混用有很好的兼容处理能力, 也就经常会出现无论是标签还是内容的大小写混乱问题, 这时采用这种模式就能很好地处理这种情况。示例代码如下:

```
<div>gg< \ /div> 在忽略大小写模式下, 可匹配: <div>gG</Div>
<? php
$str= ' <div>gG</Div> ';
if ( preg_match ( ' %<div>gg< \ /div>%i ', "<div>gG</Div>", $arr )) {
echo"匹配成功". $arr[0];
} else {
echo"匹配不成功";
}
```

忽略大小写是针对整个表达式而言, 而不仅仅是欲匹配的部分。所以, 可以在代码里放心地使用此修饰符。但是出于效率的考虑, 尽量让正则表达式所指示的范围更精确。

注意, 修饰符对整个表达式有效。如果只想修饰部分表达式, 可以使用PCRE的内部选项——“局部修饰符”。比如下面表达式仅匹配abc和abC, 而不会匹配Abc:

```
#ab (? i) c#
```

也就是说，(? i) 只对它后面的字符c起作用。

2.多行模式 (m)

在讲起始和终止符时提过：“勾选Multiline选项，即多行选项。如果选中了这个选项，“^”和“\$”的意义就变成匹配行的开始处和结束处，否则将把整个输入视作一个字符串，忽视换行符。”也就是说，正则表达式默认开始^和结束只是对于正则字符串，如果在修饰符中加上m，开始和结束将会指字符串的每一行：即每一行的开头就是^，结尾就是\$。

需要注意，m表示多行匹配，而非跨行匹配。仅当表达式中出现^、中至少一个元字符且字符串有换行符“\n”时，m修饰符才起作用，否则被忽略，如代码清单3-1所示。

代码清单3-1 多行模式

```
<? php
$str="this is reg
Reg
this is
regexp turtor, oh reg";
if ( preg_match_all ( ' %.*reg $ %mi ', $ str, $ arr )) {
echo"匹配成功";
var_dump ( $ arr );
} else {
echo"匹配不成功";
}
```

在预想中，“.*reg\$”就是以reg结尾的行，由于加了m修饰符，按理应该匹配第一行和第四行，但实际结果呢？如下所示：

```
匹配成功array (1) {
[0]=>
array (1) {
```

```
[0]=>
string ( 20 ) "regexp turtor, oh reg"
}
}
```

可以看出，只匹配最后一行的reg，而第一行虽然也是以reg结尾，但是并没有被匹配。这里用到mi，也就是把修饰符m和i组合使用。事实上，本例中即使去掉m修饰符，最终结果也是一样，这说明\$只能表示最后一行。把正则表达式改为：

```
% ^ t.*%mi
```

匹配的结果将是：

```
匹配成功array (1) {
[0]=>
array (2) {
[0]=>
string (12) "this is reg"
[1]=>
string (8) "this is"
}
}
```

匹配到两行，去掉m修饰符只匹配到第一行。可见，即使加了m修饰符，也不是将整个字符串都匹配，这就是跨行与多行的区别。

另外，使用m模式匹配需要注意换行符是否真的有效。看代码清单3-2所示的例子。

代码清单3-2 多行模式的例子

```
<? php
$source1= ' abc \ nabcd ';
$source2="abc \ nabcd";
if ( preg_match_all ( ' % ^ abc % m ', $source1, $arr )) {
echo"匹配成功——";
var_dump ( $arr );
} else {
echo"匹配不成功——";
}
if ( preg_match_all ( ' % ^ abc % m ', $source2, $arr )) {
echo"匹配成功——";
var_dump ( $arr );
} else {
echo"匹配不成功——";
}
```

运行可以看到，由于source1字符串使用单引号，\n作为普通字符而非换行符，因此匹配到的结果和预期不符。

3.点号通配模式 (s)

点号通配模式的作用是使正则表达式里的点号元字符可以匹配换行符，如果没有这个修饰符，点号不匹配换行符。沿用上面的例子，如代码清单3-3所示。

代码清单3-3 点号通配模式

```
<? php
$str="this is reg
Reg
this is
regexp turtor, oh reg";
if ( preg_match_all ( ' % this . * ? reg % i ', $str, $arr )) {
echo"匹配成功";
var_dump ( $arr );
} else {
echo"匹配不成功";
```

```
}
```

前面学习过，“.”元字符表示除换行符以外的任意字符。所以按照这个推论，上述正则表达式应该能匹配到第一行，即“this is reg”，而第三行和第四行由于之间存在一个换行符，所以不能匹配。如果给上述正则表达式加上s修饰符，即“%this.*? reg%is”，那么匹配结果就不同了，如下所示：

```
匹配成功array (1) {  
  [0]=>  
  array (2) {  
    [0]=>  
    string (11) "this is reg"  
    [1]=>  
    string (13) "this is reg"  
  }  
}
```

可以看出，这个匹配跨行了。

我们只要牢记，s修饰符包括换行符。这个修饰符很有用，特别是在抓取一些文档时，由于存在不可见换行（这是很常见的），如果使用“.”匹配，就可能存在问题，这就需要表达式能匹配换行符。看代码清单3-4所示的例子。

代码清单3-4 匹配换行符

```
$ str= ' <body >  
<div class="content">  
<div class="head"> </div >  
<div class="body"> </div >  
<div class="foot"> </div >  
</div >  
</body > ';
```

```

$array=array ();
$array2=array ();
preg_match_all ( ' | <body> (.*) < /body> | ', $str, $array );
var_dump ( $array );
preg_match_all ( ' | <body> (.*) < /body> | s ', $str, $array2 );
var_dump ( $array2 );

```

结果是第一个正则表达式没有匹配到任何内容，而第二个正则表达式匹配<body>标签之内的全部内容。原因在于，要匹配的文本在<body>标签后紧接着是一个不可见换行符，从而导致第一个正则表达式匹配失败。从这个例子中更能深刻体会到s修饰符的作用。

4. 懒惰模式 (U)

“U”相当于前面提到的“?”，表示懒惰匹配，因此3.3.8节的例子也可以改写为如下代码：

```

<? php
$str= ' [url]1.gif/[url][url]2.gif/[url][url]3.gif/[url] ';
$s=preg_replace ( "# \ [url \ ] ( .* ) \ [ \ /url \ ]#U" , " < img
src=http: //image.aiyooyoo.om/upload/$1>" , $str );
var_dump ( $s );

```

在这里：以下两个表达式作用是等价的。

```
# \ [url \ ] ( .* ) \ [ \ /url \ ]#U
```

和

```
# \ [url \ ] ( .*? ) \ [ \ /url \ ]#
```

5. 结尾限制 (D)

如果使用限制结尾字符，则不允许结尾有换行。例如，以下正则表达式将匹配“abc”、“abc\n”这样的字符，即忽视结尾的换行：

```
%abc$%
```

如果使用此模式，限定其不可有换行，必须以abc结尾，如下所示：

```
%abc$%D
```

6. 支持UTF-8转义表达 (u)

u修饰符启用PCRE中与Perl不兼容的额外功能，模式字符串被当成UTF-8。u修饰符在UNIX下自PHP 4.1.0起可用，在Win32下自PHP 4.2.3起可用，自PHP 4.3.5起开始检查模式的UTF-8合法性。看一个例子：

```
$str="php编程";  
if ( preg_match ( "/^[ \x{4e00} - \x{9fa5} ]+$/u", $str )) {  
    echo "该字符串全部是中文";  
} else {  
    echo "该字符串不全部是中文";  
}
```

正则表达式中使用了u修饰符，就可以使用UTF-8的转义表达，这实际上是兼容问题。在PHP中，使用此修饰符即可实现在表达式中支持UTF-8。

提示 除了上述几个模式修饰符之外，PHP里还支持另外几个修饰符，如A、x等，但不是很常用，这里就不做讲解了。

3.5 正则在实际开发中的应用

前面几节主要讲解了正则表达式一些基础知识，本节几个例子将展示正则表达式在日常开发中的应用。

3.5.1 移动手机校验

首先，我们练习最常用的正则表达式：移动手机校验。

通过查阅相关资料知道移动的号段有：134、135、136、137、138、139、150、151、157、158、159，新增3G号182、183和188。手机号码一共11位，前3位为运营商号段，中间4位为号码归属地，后4位为随机号。那么可以写出如下正则表达式：

```
(13[4-9] | 15[01789] | 18[238]) \d{8}
```

规则很简单，匹配前3位是否在移动号码段里，后8位为数字即可。这里用到分支、分组和元字符这三个知识点，用PHP代码测试如代码清单3-5所示。

代码清单3-5 测试移动手机号匹配

```
<? php
$mobile= '13500000000';
$regex="! ^ (13[4-9] | 15[0189] | 188) \d{8}$! ";
if (! preg_match ($regex,$mobile)) {
die ( '错误的移动号码段! ');
} else {
die ( '移动手机. ');
}
```

运行结果符合预期。注意第三行代码，在前面已经说过，正则表达式的分隔符只要遵

循规则是可以随意的。所以这里用的是感叹号，而不是常见的斜杠，当然，用#、~也可以（为了书写美观，一般不用斜杠）。

如果只判断手机号就更简单了，如下所示：

```
1[358]\d{9}
```

3.5.2 匹配E-mail地址

运用学过的知识，我们实现简单的E-mail识别。

E-mail最简单的形式为user@domain.com。其中，user为用户名，domain为域名，com为后缀；当然，后缀还可以是net、name、cn等。一般用户名由3个以上的字母和数字组成，当然也不能太长，允许出现下画线。中间一个@符号，后面的域名长度为1~64位，后缀长度一般为2~5位，如下所示：

```
\w{3, 16}@\w{1, 64}\.\w{2, 5}
```

以上表达式匹配用户名长度3~16位，紧跟一个@符号，然后是1~64位的域名，再然后是dot(.号)，最后是2~5位的后缀。这个正则表达式还存在一些问题，比如xxx_yyy@yahoo.com.cn这样的地址，后面的.cn就无法匹配到。这就需要进一步学习。

Regular Expression Tester工具提供一些预定义的正则表达式，它提供的匹配E-mail的正则表达式如下：

```
^[a-z0-9_ \-]+(\.[a-z0-9 \-]+)*@([\a-z0-9 \-]+\.)+([a-z]{2} | aero | arpa | biz | com | coop | edu | gov | info | int | jobs | mil | museum | name | nato | net | org | pro | travel)$
```

这个表达式很长，使用字符组和分支条件语法，很好地解决了com.cn这样多个后缀域名的问题。相信现在读者对于匹配电话号码、QQ号等应该能得心应手了。

3.5.3 转义在数据安全中的应用

数据安全是任何一款软件设计中都需要考虑的问题。从技术层面讲，数据安全就是保护存储和使用的数据不被窃听、盗取和破坏。这可能是由外部因素造成的，比如由于过滤不严格造成SQL注入漏洞、提升脚本执行权限等，也有可能是由代码内部设计造成的，如死循环、低效率的语句造成服务器性能下降以致影响访问。

社会学意义上的数据安全则更广泛。比如，在在线购物商城的设计中，由于设计者错误地使用自增ID作为商品的单据流水号，竞争对手或有心人很容易分析出这个商城的每日销售量，进而估算出销售额、利润等商业机密数据。

在程序中要保证数据的安全，除了要保证代码内部运行的可靠外，最主要的就是严格处理外部数据，即秉持“一切输入输出都是不可靠的”理论，这就要求我们做好数据过滤和验证。PHP编程中最简单的过滤机制就是转义，即对用户的输入和输出进行转义和过滤。图3-4所示是简单的留言本。

简单留言本的演示代码如下：

```
<form action=""method="POST">
```



图 3-4 简单的留言本

```
<textarea name="content"> </textarea>
```

```
<input type="submit" value="留言"/>
```

```
<? php
```

```
echo $_POST[ ' content ' ];
```

```
? >
```

程序中没有任何的输入输出过滤，当在留言框中输入以下内容时得到如图3-5所示的页面。

```
<style>
Body { background: #000000 }
</style>
```

很显然，由于输入含有CSS代码和JavaScript代码，没有对其进行处理便原样输出，导致页面被篡改。这就是常说的“XSS攻击”。

针对上面的情况，只需在接收数据时使用htmlspecial chars函数，把代码中的特殊字符转为HTML实体，这样在输出时就不会使页面受影响了。这些特殊字符主要是“”、“'”、“&”、“<”、“>”。比如把“<script>alert(1); </script>”转为“<script>alert(1); </script>”，就可以阻止大部分XSS攻击。



图 3-5 被“攻击”后的留言本

注 HTML中规定字符实体引用（HTML Entities）还有对应的数字型实体（NCR），实际上是对这个实体的编号。比如，HTML Entities的格式“<”，NCR的格式“<”或“<”，均表示“<”字符。

有的时候，不希望出现这些无意义的字符，因为既然页面不允许这些HTML标签，那就干脆过滤掉，而不是显示出来，这样页面就不会留下恶意攻击者所留下的这些代码。要达到这个目的，就需要借助正则表达式。比如，要过滤所有HTML标签的正则表达式如下所示：

```
< \/? [^ >]+ > //过滤所有HTML标签
```

这个正则表达式匹配嵌套的尖括号，一个“\/?”表示斜杠可有可无，这样就匹配标

签的起始和关闭位置。“`[^>]+`”意思是：不是右尖括号的字符重复一次和更多次。为什么不是“`*`”呢？因为HTML标签里至少也是一个字符，如``，而`<>`不是合法HTML标签，最后关闭右尖括号。用下面字符测试：

```
<strong>strong</strong> <img src=a.jpg/>
```

其匹配情况如图3-6所示。



图 3-6 程序运行结果

从图中可以看出，运行结果基本符合我们的需求。同理，还可以只保留部分HTML标签，既不造成安全问题，又能使页面内容更丰富，这就可以利用UBB代码功能来实现。

前面提到过用正则表达式实现UBB标签的功能比较麻烦，但在这里只需要白名单功能，即只保留比较“安全”的标签，通过PHP内置的`strip_tags`函数可以容易做到。这个函数用于从字符串中去除HTML和PHP标记，仅保留参数中指定的标签。演示代码如下所示：

```
<? php
$text= ' <p>Test paragraph. <! ——Comment——> <a href="#fragment"> Other text </a>
';
echo strip_tags ($text);
echo "\n";
//允许<p>和<a>
echo strip_tags ($text, ' <p><a> ');
? >
```

另外，在SQL语句构造中，当字符含有引号的时候，可能造成SQL解析和执行失败。这样就要求转义。一种处理办法是把引号转义成实体，另一种处理办法是使用

addslashes函数把其转义。我们通常会使用后者。

addslashes函数返回一个字符串，该字符串为了数据库查询语句等的需要在某些字符前加上了反斜线。这些字符是单引号（'）、双引号（"）、反斜线（\）与NUL（NULL字符）。

注意 为了处理更多情况，还需要特别注意“%”、“'”等特殊字符。如果能转义则进行转义，没有对应的转义时，则进行过滤。

3.5.4 URL重写与搜索引擎优化

URL重写 (Rewrite) 是截取传入Web请求并自动将请求重定向到其他URL的过程。比如浏览器发来请求hostname/list_1, 服务器自动将这个请求中定向为http://hostname/list.php?id=1。该技术常用于搜索引擎优化 (Search Engine Optimization, SEO)。

伪静态也是重写的一种实现。例如, 某论坛网址为forum 17 1.html, 实际地址可能是forum.php?fid=17&page=1, 其中第一个数字是版块ID, 第二个数字是页数。这样的网址看起来比较短, 没有一大堆的&符号和GET参数, 不但用户喜欢这种友好的网址, 搜索引擎也喜欢这种简洁明了的网址。

URL重写有两种实现方式:

纯代码实现, 通过解析PATH_INFO实现。

服务器实现, 如利用Apache的mod_rewrite模块实现。

下面先介绍利用mod_rewrite实现重写的方法。例如, 实现列表页面list.php?Mode=A 重写为伪静态 list A.html, 步骤如下 (假设工程的根目录是E:/dev/www/php/new)。

首先, 配置Apache。在httpd.conf里把以下所示的这一行代码前面的#去掉, 启用Rewrite规则。

```
#LoadModule rewrite_module modules/mod_rewrite.so
```

在对应的<Directory"E:/dev/www/php">配置项下设置AllowOverride All。

在网站E:/dev/www/php/new目录下新建.htaccess文件, 并输入如下内容:

```
RewriteEngine on
```

```
RewriteRule index.html index.php  
RewriteRule list-([A-Z]+)\.html$ list.php? mode=$1[NC]
```

重启Apache。现在访问http://127.1/list A.html看看效果吧。可以看到访问http://127.1/list A.html和http://127.1/list.php? mode=A指向的是同一个页面，这就说明伪静态设置成功了。

现在逐条解释其原理。

Apache开启Rewrite模块，该模块默认是关闭的。

设置AllowOverride All的目的是让.htaccess文件生效；如果把AllowOverride设置成none，.htaccess文件将被完全忽略。

新建一个.htaccess文件。.htaccess文件是Apache中重要的配置文件，其格式为纯文本。它提供针对目录改变配置的方法，通过在一个特定的文档目录中放置包含一个或多个指令的文件，作用于此目录及其所有子目录。

注意 Windows资源管理器里不允许建立.htaccess文件，可以在命令行窗口输入“echo>.htaccess”达到新建的目的。

以下是对.htaccess中每一行代码的解释。

1) 打开运行时重写功能，其默认是关闭：

```
RewriteEngine on
```

2) 建立一条重写规则，把index.php重写为index.html：

```
RewriteRule index.html index.php
```

很显然，这是一个伪静态，而“index.html index.php”是最简单的正则表达式。

3) 把原地址list.php? mode=1形式重写成list ([A-Z]+) \.html:

```
RewriteRule list- ([A-Z]+) \.html$ list.php? mode= $1[NC]
```

实际上上述代码可以理解为，如果Apache遇到符合正则表达式list ([A-Z]+) \.html的请求，先捕获第一个括号里的值（这是紧挨着list后面，以.html结尾的一个字母），那么就重定向到真实的请求地址list.php? mode=1，而1是正则表达式里的反向引用。这就是重写的语意。括号中的“NC”表示大小写不敏感。

如果这个页面还有分页应该怎么写？比如list.php? mode=A& page=2。为了让URL更有意义，可以把这个地址重写为list A page 2.html。照葫芦画瓢，在已有基础上添加一条规则：

```
RewriteRule list- ([A-Z]+) -page- ( \d?) \.html$ list.php? mode= $1&page= $2
```

现在直接访问list A page 2.html试试。由于RewriteEngine为on，所以这一步并不需要重启Apache。虽然现在可以访问了，但是不是觉得这么写很累赘？

第二条和第三条重写规则都是针对list.php文件，可以合并成一条吗？答案是可以的。完整.htaccess文件如下：

```
RewriteEngine on
RewriteRule index.html index.php
#RewriteRule list- ([A-Z]+) \.html$ list.php? mode= $1[NC]这一行是注释
RewriteRule list- ([A-Z]+) (? : -page- )? ( \d?) \.html $ list.php? mode= $ 1 & page=
$ 2[NC]
```

新的重写规则就是最常用的正则表达式，如“?:”表示非捕获性匹配。只要理解了正则表达式，再辅之以Apache手册，掌握URL重写就会成为一件很轻松地事情。

提示 Nginx也能实现网址静态化，而且语法上几乎没什么差别，用到的都是基于正则表达式的语法。

使用URL重写能给网站带来哪些好处呢？

1) 有利于搜索引擎的抓取。因为现在大部分搜索引擎更喜欢抓取一些静态页面。而现在页面大部分数据都是动态显示的。这就需把动态页面变成静态页面，这样有利于搜索引擎抓取。

2) 用户更容易记忆。很少有用户关心网站页面地址，但对大中型网站来说，增强可读性还是必需的。

3) 隐藏实现技术。避免暴露采用的技术，给攻击网站增加阻碍。特别是前面讲的攻击方式，把参数隐藏起来，在一定程度上增加了攻击的难度。

重写还能帮助我们完成很多事情，比如简单下载处理。例如，现在需要对外提供文件下载服务，比如下载php.rar文件。在下载前还需要做许多额外工作，如下载权限的判断、服务器分流等，这就不是一个简单文件链接可以处理的，需要服务器端脚本做一些复杂处理。通过使用URL重写，可以把对php.rar文件的请求重定向到一个PHP文件的请求，在这个PHP文件中进行判断，满足所有判断后再输出文件。假定当前工程目录是E: \ www \ php \ download，现在要把对E: \ www \ php \ download \ php.rar的请求定向到一个PHP文件。添加下面的代码到当前目录的.htaccess文件中：

```
RewriteEngine on
RewriteRule (.*\.(rar|zip|chm))$down.php?file=$1[NC]
```

down.php中的代码如下：

```
<? php//echo"The file you wanna-download is \" ,$_GET[ ' file ' ];
```

```
$filename=basename($_GET['file']);
//其他逻辑处理,如检查是否有权限或者是否属于盗链,处理完成后提供下载
if(!is_file($filename)|!is_readable($filename)){
exit("没有找到指定的文件");
}
header("Content-type: application/octet-stream");
$ua=$_SERVER["HTTP_USER_AGENT"];
//处理中文文件名
$encode_filename=urlencode($filename);
$encode_filename=str_replace("+","%20",$filename);
if(preg_match("/MSIE",$ua)){
header('Content-Disposition: attachment; filename="'. $encode_filename. '");
} else if(preg_match("/Firefox",$ua)){
header("Content-Disposition: attachment; filename*= \ "utf8 ' ' ". $filename. '");
} else {
header('Content-Disposition: attachment; filename="'. $filename. '");
}
//如果服务器支持X_sendfile module,则使用X_sendfile头加速下载
$x_sendfile_supported=in_array('mod_xsendfile',apache_get_modules());
if(!headers_sent()&&$x_sendfile_supported){
header("X-Sendfile: {$filename}");
} else {
@readfile($filename);
}
```

再浏览<http://download.com/hello.rar> (已将虚拟主机域download.com映射到E: \ www \ php \

download目录),此请求将被重定向到down.php? file=hello.rar,这样就简单的实现了对文件下载的处理。

我们还能对重写做进一步扩充,来满足不同的需求。更多URL重写知识可以参考Apache 2手册。

3.5.5 删除文件中的空行和注释

不仅在代码中会用到正则表达式，其实在日常软件应用中也会涉及正则表达式。比如字处理软件、代码开发工具中都提供对正则表达式查找和替换的支持。

这里以UltraEdit为例来介绍正则表达式在日常软件中的应用。UltraEdit是一款功能强大的编辑器，支持正则表达式的使用。UltraEdit虽然和IDE无法相提并论，但是在处理一些小文件时，会显出其快速、轻量级的特点。

例如，PHP源文件中包含空行和注释，UltraEdit中的代码如图3-7所示。

这里面许多空行和注释，为了提高代码的可读性，需要去除大段空行。如果手工操作，必然很麻烦。此时，可以使用UltraEdit的正则表达式功能。在编辑菜单中，选择“替换”，输入如下表达式：

```
%[ ^t]++ ^p
```

注意，^ t前面的空格也要输入。单击替换所有，文件中的空行就删除了。如果还要删除注释，可以输入“//? *”，处理完成后的效果如图3-8所示。

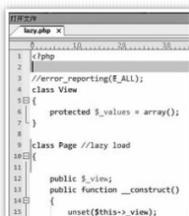


图 3-7 UltraEdit中的代码

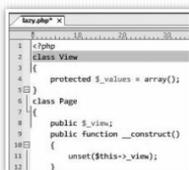


图 3-8 程序运行结果

这里使用 UltraEdit 的正则表达式，也可以选择 UNIX（POSIX 规范）和 Perl（PCRE 规范）风格的表达式，它们之间略有不同。

提示 有些框架为了尽力提升效率或者由于商业的原因，往往会在部署和发布时，通过解析 PHP 代码中的 token 清除源文件中表示空白和注释的 token。在这种情况下，使用代码的方式可能更好。

但有时无法使用代码完成这件事，我们不得不使用正则表达式。比如在使用 Word 保存资料的时候，文件中常常会带有大量的空白段落，通常只能手动删除这些空段落来调整格式，费时费力。在 Word 中，选择特殊字符，把 `^p^p` 替换成 `^p` 即可。Word 中这两个所谓的“特殊字符”，实际上就是正则表达式的一种体现。

3.6 正则表达式的效率与优化

正则表达式可以看做描述字符串匹配的算法代码，本质上说是一种有限状态机在计算机中的表示方法。状态机，表示有限个状态以及在这些状态之间进行转移和动作等行为的数学模型，在计算机中表示出来的就是有向图。而作为图就会涉及查找、回溯过程。不同查找的方式，其回溯过程也不一样，效率自然也是有区别的。要想弄明白正则表达式的效率，就得深入编译原理、数据结构等概念，这里不做原理性阐述，只介绍一些普遍原则。

注意 不同语言中有不同实现和限制，因此下面一些原则只是最基本的原则，不保证在所有实现中通用。有时候，某一种实现会对预知的情况进行优化，而另一种则不会。也就是说，正则表达式的效率不仅和正则引擎的种类有关，还和引擎具体实现有关。

1) 使用字符组代替分支条件。比如，使用[a-d]表示a~d之间的字母，而不是使用(a|b|c|d)。下面代码说明二者的效率差异。

```
<? php
$cnt=1000;
$testStr="";
for ($i=0; $i<1000; $i++) {
    $testStr.="abababcfg";
}
//第一种方案
$start=microtime ( TRUE );
for ($i=0; $i< $cnt; $i++) {
    preg_match ( '#^ ( a | b | c | d | e | f | g ) + $# ', $testStr );
}
echo ' waste time ( s ): ', microtime ( TRUE ) - $start, PHP_EOL;
//第二种方案
$start=microtime ( TRUE );
for ($i=0; $i< $cnt; $i++) {
    preg_match ( '#^[a-g]+ $# ', $testStr );
}
echo ' waste time ( s ): ', microtime ( TRUE ) - $start, PHP_EOL;
//第三种方案和第二种方案本质上是相同的
```

```
$start=microtime ( TRUE );  
for ($i=0; $i< $cnt; $i++) {  
    preg_match ( ' #^[abcdefg]+$ # ', $testStr);  
}  
echo ' waste time ( s): ', microtime ( TRUE ) - $start, PHP_EOL;
```

运行结果如下所示:

```
waste time ( s): 3.2742960453033  
waste time ( s): 0.059613943099976  
waste time ( s): 0.059823036193848
```

可以看出, [a g]和[abcdefg]这两种表达式的效率相当, 且使用字符组比分支条件的速度要快很多。这是由于在匹配单个字符的时候, 引擎会把[abc]这样的字符组视为1个元素, 而不是3个元素 (a、b、c)。整个元素作为匹配迭代的一个单元, 不需要进行三次迭代, 从而提高匹配效率。

2) 优先选择最左端的匹配结果。这在介绍分支条件匹配邮编的时候已经提到过。对于传统型NFA引擎来说, 这样改动对正则匹配的效率是有利的, 因为引擎一旦找到匹配结果就会停下来, 而不会去尝试正则表达式的每一种可能 (PHP中的preg函数就属于传统型NFA引擎)。

3) 标准量词是匹配优先的。若用量词约束某个表达式, 那么在匹配成功前, 进行的尝试次数有下限和上限。例如, 正则表达式为:

```
\w*(\d+)
```

字符串为copy2003y。这个正则匹配的1是多少? 如果回答2003就错了, 其结果应该是3。解释如下: 当正则引擎用 “\w*(\d+)” 匹配字符串copy2003y时, 会先用

“\w*”匹配字符串copy2003y。“\w*”会匹配字符串copy2003y的所有字符，然后再交给“\d+”匹配剩下的字符串，而剩下的没有了。这时，“\w*”规则会不情愿地吐出一个字符，给“\d+”匹配。同时，在吐出字符之前，记录一个点。这个点就是用于回溯的点，然后“\d+”匹配y，发现不能匹配成功，此时会要求“\w*”再吐出一个字符；“\w*”先记录一个回溯的点，再吐出一个字符。这时，“\w*”匹配结果只有copy200，已经吐出3y。“\d+”再去匹配3，发现匹配成功，会通知引擎，并且直接显示出来。所以，“(\d+)”的结果是3，而不是2003。

如果改为非贪婪模式呢？“\w*?(\d+)”匹配结果就应该是2003。由于“\w*?”是非贪婪，正则引擎会用表达式“\w+?”每次仅匹配一个字符串，然后再将控制权交给后面的“\d+”匹配下一个字符，同时记录一个点，用于匹配不成功时，返回这里再次匹配。

提示 尽量以组为单位进行匹配，使用固化分组就能避免无休止的匹配。

4) 谨慎用点号元字符，尽可能不用星号和加号这样的任意量词。只要能确定范围(例如“\w”), 就不要用点号; 只要能够预测重复次数, 就不要用任意量词。假设一条微博消息的XML正文部分结构如下:

```
<span class="msg">.....</span>
```

正文中无尖括号，写法如下:

```
<span class="msg">[^<]{1, 200}</span>
```

或者:

```
<span class="msg">.*</span>,</pre>

---


```

上述第一种代码的思路要好于第二种代码，原因有两个：

使用 “[^ <]”，保证了文本的范围不会超出下一个小于号所在位置。

明确长度范围 { 1, 200 }，依据是一条微博消息大致的字符长度范围是固定的，现在微博字数长度限制是140个字。

PHP的PCRE扩展中提供了两个设置项：

pcre.backtrack_limit//最大回溯数

pcre.recursion_limit//最大嵌套数

默认backtarck_limit是100000（10万），recursion_limit限制最大正则嵌套层数。在正则表达式的使用中，应尽量避免回溯次数过多等情况。

因回溯次数过多导致正则匹配失败的案例如下：

```
<? php
$a=range ( 1, 12636 );
shuffle ( $a );
$d=print_r ( $a, TRUE );
echo strlen ( $d ) /1024, PHP_EOL;
$a="<? xml version= ' 1.0 ' encoding= ' iso-8859-1 '? ><ppc>header". $d."tail</ppc>";
preg_match_all ( "/<ppc> (.*) ( \d* ) < \ /ppc>/s", $a, $m, PREG_SET_ORDER );
var_dump ( $m );
echo ' had result: ', PHP_EOL;
echo
strlen ( $ m[0][1] ), PHP_EOL, substr ( $ m[0][1], 0, 50 ), PHP_EOL, substr ( $ m[0]
[1], -50 ),
PHP_EOL;
//复制上面的代码，唯一的修改是增加字符串长度，使正则匹配爆栈
//ini_set ( ' pcre.backtrack_limit ', 100000000 ); //这里增加为1亿
$x2=range ( 1, 12638 );
shuffle ( $x2 );
$d2=print_r ( $x2, TRUE );
echo strlen ( $ d2 ) /1024, PHP_EOL;
```

```
$a2="<? xml version= ' 1.0 ' encoding= ' iso-8859-1 '? > <ppc>header". $d2."tail</ppc>";  
$ret=preg_match_all ( "/<ppc> ( .*? ) ( \ d* ) < \ /ppc> /s", $ a2, $ m2, PREG_SET_ORDER);  
var_dump ( $ m );  
echo preg_last_error ();  
echo ' had no result: ', PHP_EOL;  
echo  
strlen ( $ m2[0][1] ), PHP_EOL, substr ( $ m2[0][1], 0, 50 ), PHP_EOL, substr ( $ m2[0][1], -50 );
```

注意 程序的运行结果依赖于你的PHP的设置。

这个案例告诉我们，由于正则表达式使用不当导致匹配失败的情况是有可能发生的，特别是当Web文档比较大、结构比较复杂时。解决办法就是，把以下代码前面的注释符去除，给PHP配置更大的回溯栈空间：

```
ini_set ( ' pcre.backtrack_limit ', 100000000 );
```

但这是治标不治本的办法，最终解决方案还是优化正则表达式。

同理，能用懒惰匹配就坚决不用贪婪匹配。

5) 尽量使用字符串函数处理代替。使用字符串函数和正则表达式都可以处理字符串，两者相比，字符串函数处理的效率更高。当然，有些情况几乎是非正则表达式不能胜任的，或者不用正则表达式的成本太高，这些情况不得不用正则表达式，既然如此，就应该设计好。

6) 合理使用括号。每使用一个普通括号 ()，而不是非捕获型括号 (?: ……)，就会保留一部分内存等着再次访问。这样的正则表达式、无限次的运行次数，无异于一根根稻草的堆加，终将会把骆驼压死。

7) 起始、行锚点优化。能确定起始位置，使用 ^ 能提高匹配的速度。同理，使用标记结尾，正则引擎则会从符合条件的长度处开始匹配，略过目标字符串中许多可能的字

符。在写正则表达式时，应该将锚点独立出来，例如“`^(?: abc | 123)`”比“`^123 | ^ abc`”效率高，而“`^(abc)`”比“`^(abc)`”效率要高。

这个原则不适用于所有正则引擎。比如在PCRE中，二者效率相当。

8) 量词等价转换的效率差异。例如在PHP中，使用“`\d\d\d`”和“`\d{3}`”，或者“`====`”和“`= { 4 }`”，它们之间的效率几乎没有差别。但是换用其他语言可能就会有比较明显的性能差异了。

9) 对大而全的表达式进行拆分。

10) 使用正则以外的解决方案。前面已经提到，在有的场合可以使用字符串来代替正则表达式，此外，还有其他方案可以代替正则表达式。例如，在某项目中需要分析PHP代码，分离出对应的函数调用（以及源代码对应的位置）。虽然这使用正则表达式也可以实现，但无论从效率还是代码复杂度方面考虑，这都不是最优的方式。PHP已经内置解析器的接口PHP Tokenizer。

使用PHP Tokenizer能简单、高效、准确地分析出PHP源代码的组成。token_get_all函数参数为一段PHP代码，可提取出这段代码里的常量、变量、类名、函数等。这在编写phpdoc、代码优化提速、自动加载类时都可以用到。比如，在解析URL时没必要用正则表达式，使用parse_url函数即可；在获取HTTP头时，也可以使用get_headers函数。

在进行输入校验时，可以使用PHP 5提供的filter函数。例如，校验E-mail地址的代码如下：

```
filter_var ( ' admin@example.com ', FILTER_VALIDATE_EMAIL );
```

这样是不是好多了呢？如果在JavaScript里，可以使用DOM代替一些正则匹配。

这里总结几种正则表达式的取代方案，它们能部分取代正则表达式的实现。

PHP的字符串函数；

PHP的Tokenizer系列函数；

PHP的url函数及一些http函数；

PHP的filter系列函数；

JavaScript的DOM模型。

3.7 本章小结

本章中主要讲解了正则表达式的一些基本概念和语法，并且通过理论结合实际的方式，讲解了正则表达式在开发中的应用，最后介绍了正则表达式的效率优化和一些替代方案。

正则表达式中最容易混淆的就是转义，最难理解的就是环视，而环视在复杂的正则匹配中又是无法避免的，所以重点要掌握环视的概念和应用。

在实例讲解中以XSS攻击为例，强调数据过滤的必要性，“一切输入都是不可信任的”，在设计中，要始终保持高度警惕。当然，正则表达式只是一种解决方案。接下来着重讲了SEO中的静态化，通过Apache的URL重写把动态请求地址映射到一个静态HTML文件上，从而实现搜索引擎友好的地址。

正则优化的关键理念就是“减少回溯”，常见的手段就是减少分支、使用环视和懒惰匹配。最后列举几种正则表达式的替代方案，供读者参考。

正则表达式是一种抽象语法，要熟练掌握正则表达式，不但需要学会总结，还需要多多练习。

第4章 PHP网络技术及应用

作为专注于Web编程的PHP而言，简单的网络模型和接口，使得在PHP中实现套接字、cURL等都变得极其简单。本章主要从HTTP协议入手，逐步深入，展示PHP网络技术的实际应用。并且展开讨论Socket、网络协议分析、Cookie和Session等相关知识。

HTTP协议是整个Web的基础，是客户端和服务端协同工作的基石，要想了解Web的工作原理、优化Web应用，就要完全理解HTTP协议。

4.1 HTTP协议详解

简单来说，HTTP就是一个基于应用层的通信规范：双方要进行通信，大家都要遵守一个规范——HTTP协议。HTTP协议从WWW服务器传输超文本到本地浏览器，可以使浏览器更加高效。HTTP协议不仅保证计算机正确快速地传输超文本文档，还能确定传输文档中的哪一部分，以及哪部分内容首先显示（如文本先于图形）等。

4.1.1 HTTP协议与SPDY协议

HTTP（Hyper Text Transfer Protocol，超文本传输协议）是万维网协会（World Wide Web Consortium）和Internet工作小组（Internet Engineering Task Force, IETF）合作的结果，最终发布了一系列的RFC（Request For Comments）。RFC 1945定义了HTTP 1.0版本，最著名的是RFC 2616，其中定义了今天普遍使用的版本——HTTP 1.1。

HTTP是一个应用层协议，由请求和响应构成，是一个标准的客户端服务器模型。HTTP通常承载于TCP协议之上，有时也承载于TLS或SSL协议层之上，这个时候，就成了常说的HTTPS。默认HTTP的端口号为80，HTTPS的端口号为443。

HTTP协议在OSI模型中的位置如图4-1所示。

HTTP协议的模型就是客户端发起请求，服务器回送响应，如图4-2所示。HTTP协

议是一个无状态的协议，同一个客户端的这次请求和上次请求没有对应关系。



图 4-1 HTTP协议在OSI模型中的位置



图 4-2 HTTP请求响应模型

这种设计属于典型的“问答式”交互，客户端和服务端一问一答，使HTTP协议模型异常简单。这种设计也存在问题，比如服务器端不会主动向客户端PUSH，一问一答的轮询也会使TCP连接频繁建立和断开，导致其交互效率不高。基于以上缺点，SPDY协议应运而生。

SPDY协议是Google推出的协议，优化了浏览器和服务端之间的通信，支持流复用，具备优先级的请求、主动发起请求、强制SSL安全传输等先进的特性。目前，Chrome和Firefox浏览器的最新版均支持SPDY协议，一些服务器端软件也纷纷开始支持SPDY协议，如Jetty 8、Nginx 1.3.x等服务器的最新版本。可以预见，在未来的几年，SPDY协议将得到很大推广。

SPDY协议的应用需要客户端浏览器和服务端同时支持。目前，应用SPDY协议的主要是Google产品，如Google Plus。

4.1.2 HTTP协议如何工作

浏览网页是HTTP协议的主要应用，但是这并不代表HTTP协议就只能应用于网页的浏览。只要通信的双方都遵守HTTP协议，其就有用武之地。比如腾讯QQ、迅雷等软件都使用HTTP协议（当然还包括其他的协议）。

那么HTTP协议是如何工作的呢？

首先，客户端发送一个请求（Request）给服务器，服务器在接收到这个请求后将生成一个响应（Response）返回给客户端。一次HTTP操作称为一个事务，其工作过程可分为四步：

1) 客户机与服务器需要建立连接。单击某个超链接，HTTP协议的工作开始。

2) 建立连接后，客户机发送一个请求给服务器。格式为：前边是统一资源标识符（URL）、中间是协议版本号，后边是MIME信息（包括请求修饰符、客户机信息和可能的内容）。

3) 服务器接到请求后，给予相应的响应信息。格式为：首先是一个状态行（包括信息的协议版本号、一个成功或错误的代码），然后是MIME信息（包括服务器信息、实体信息和可能的内容）。

4) 客户端接收服务器返回的信息并显示在用户的显示屏上，然后客户机与服务器断开连接。

如果以上过程中的某一步出现错误，产生错误的信息将返回到客户端，由显示屏输出。对于用户来说，这些过程是由HTTP协议自己完成的，用户只要用鼠标单击，等待信息显示就可以了。

提示 怎样才能看到HTTP协议呢？可以查看RFC 2616文档，或者使用抓包软件。通用的抓包软件主要有IRIS、Wireshark等，专门抓取HTTP包的软件主要有HttpWatch、IE Analy zer、Fiddler、Charles等。本书使用Fiddler进行查看和调试，其体积小、功能强大，并且是免费的。在浏览器中使用Firefox的扩展Firebug查看HTTP请求。

下面简单介绍HTTP协议中一些主要的概念。

1.请求

在发起请求前，需要先建立连接。

连接是一个传输层的实际环流，它建立在两个相互通信的应用程序之间。在HTTP 1.1协议中，request和response头中都有可能出现一个connection的头，其决定当Client和Server通信时对于长链接如何处理。

HTTP 1.1协议中，Client和Server默认对方支持长链接，如果Client使用HTTP 1.1协议，但又不希望使用长链接，需要在header中指明connection的值为close；如果Server方也不想支持长链接，则在response中需要明确说明connection的值为close。不论request还是response的header中包含了值为close的connection，都表明当前正在使用的TCP连接在请求处理完毕后会断掉，以后Client再进行新的请求时必须创建新的TCP连接。

HTTP请求由三部分组成：请求行、消息报头、请求正文。请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：

```
Method Request-URI HTTP-Version CRLF
```

上述格式中各参数说明如下：

Method：请求方法。

Request URI：一个统一资源标识符。

HTTP Version：请求的HTTP协议版本。

CRLF：回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法（所有方法全为大写）有多种，各个方法的解释如下：

GET: 请求获取Request URI所标识的资源。

POST: 在Request URI所标识的资源后附加新的数据。

HEAD: 请求获取由Request URI所标识的资源的响应消息报头。

PUT: 请求服务器存储一个资源, 并用Request URI作为其标识。

DELETE: 请求服务器删除Request URI所标识的资源。

TRACE: 请求服务器回送收到的请求信息, 主要用于测试或诊断。

CONNECT: 保留以备将来使用。

OPTIONS: 请求查询服务器的性能, 或者查询与资源相关的选项和需求。

2. 响应

在接收和解释请求消息后, 服务器返回一个HTTP响应消息。HTTP响应也由三个部分组成, 分别是: 状态行、消息报头、响应正文。

状态行格式如下:

HTTP-Version Status-Code Reason-Phrase CRLF

上述格式中各参数说明如下:

HTTP Version: 服务器HTTP协议的版本。

Status Code: 服务器发回的响应状态代码。

Reason Phrase: 状态代码的文本描述。

状态代码由三位数字组成, 第一个数字定义了响应的类别, 有五种可能取值:

1xx: 指示信息——请求已接收，继续处理。

2xx: 成功——请求已被成功接收、理解、接受。

3xx: 重定向——要完成请求必须进行更进一步的操作。

4xx: 客户端错误——请求有语法错误或请求无法实现。

5xx: 服务器端错误——服务器未能实现合法的请求。

常见状态代码、状态描述和说明如下：

200 OK: 客户端请求成功。

400 Bad Request: 客户端请求有语法错误，不能被服务器所理解。

401 Unauthorized: 请求未经授权，这个状态代码必须和WWW Authenticate报头域一起使用。

403 Forbidden: 服务器收到请求，但是拒绝提供服务。

404 Not Found: 请求资源不存在，例如输入了错误的URL。

500 Internal Server Error: 服务器发生不可预期的错误。

503 Server Unavailable: 服务器当前不能处理客户端的请求，一段时间后可能恢复正常。

例如，响应信息“HTTP/1.1 200 OK (CRLF)”，表示响应请求到达服务器后被成功识别，返回成功标记。响应正文就是服务器返回的资源的内容。

3.报头

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。每个报头域组成形式如下：

名字+: +空格+值

注意 消息报头域的名字是不区分英文大小写的。报头都是自解释的，具体在这里就不描述了。

1) 普通报头中有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息（如缓存控制、连接控制等）。

2) 请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息（如UA头、Accept等）。

3) 响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request URI所标识的资源进行下一步访问的信息（如Location）。

4) 实体报头定义了关于实体正文和请求所标识的资源的元信息，例如有无实体正文。

一个报头的信息截图如图4-3所示。



图 4-3 一个HTTP请求的响应数据

比较重要的几个报头如下。

Host: 头域指定请求资源的Internet主机和端口号，必须表示请求URL的原始服务器或网关的位置。HTTP 1.1请求必须包含主机头域，否则系统会以400状态码返回。

User Agent: 简称UA，内容包含发出请求的用户信息。通常UA包含浏览者的信息，主要是浏览器的名称版本和所用的操作系统。在图4-3所示中可以看到，客户端使用的是Gecko渲染引擎的浏览器，这里是Firefox；操作系统为Windows NT 6.1的内核，即Windows 7操作系统（内核版本号和操作系统代号不是一一对应的）。这个UA头不仅

仅是使用浏览器才存在，只要使用了基于HTTP协议的客户端软件都会发送这个请求，无论是手机端还是PDA等。这个UA头是辨别客户端所用设备的重要依据。

Accept: 告诉服务器可以接受的文件格式。通常这个值在各种浏览器中都差不多。不过WAP浏览器所能接受的格式要少一些，这也是用来区分WAP和计算机浏览器的主要依据之一。随着WAP浏览器的升级，其已经和计算机浏览器越来越接近，因此这个判断所起的作用也越来越弱。

Cookie: Cookie分两种，一种是客户端向服务器端发送的，使用Cookie报头，用来标记一些信息；另一种是服务器发送给浏览器的，报头为Set Cookie。二者的主要区别是Cookie报头的value里可以有多个Cookie值，并且不需要显式指定domain等。而Set Cookie报头里一条记录只能有一个Cookie的value，需要指明domain、path等。

Cache Control: 指定请求和响应遵循的缓存机制。在请求消息或响应消息中设置Cache Control并不会修改另一个消息处理过程中的缓存处理过程。请求时的缓存指令包括no cache、no store、max age、max stale、min fresh、only if cached；响应消息中的指令包括public、private、no cache、no store、no transform、must revalidate、proxy revalidate、max age。

Referer: 头域允许客户端指定请求URI的源资源地址，这可以允许服务器生成回退链表，可用来登录、优化缓存等。也允许废除的或错误的连接由于维护的目的被追踪。如果请求的URI没有自己的URI地址，Referer不能被发送。如果指定的是部分URI地址，则此地址应该是一个相对地址。Referer通常是流量统计系统用来记录来访者地址的参数。

Content Length: 内容长度。

Content Range: 响应的资源范围。可以在每次请求中标记请求的资源范围，在连接断开重连时，客户端只请求该资源未下载的部分，而不是重新请求整个资源，实现断点续传。迅雷就是基于这个原理，使用多线程分段读取网络上的资源，最后再合并。

Accept Encoding: 指定所能接受的编码方式。通常服务器会对页面进行GZIP压缩后再输出以减少流量，一般浏览器均支持对这种压缩后的数据进行处理。但对于我们来说，如果不想接收到这些看似“乱码”的数据，可以指定不接受任何服务器端压缩处理，

要求其原样返回。

自定义报头：在HTTP消息中，也可以使用一些在HTTP 1.1正式规范里没有定义的头字段，这些头字段统称为自定义的HTTP头或者扩展头。比如server字段，在图4-3中，Google使用的是GWS服务器。这个报头一般是由服务器发送的。也可以定义一些“不正规”的报头，如“WEBMASTER: chen@qq.com”。在PHP里，使用header函数即可实现。图4-3中，X XSS Protection也是Google自定义的报头。

4.1.3 HTTP应用：模拟灌水机器人

垃圾评论和机器人一直是各大论坛和博主最头疼的问题，这些来势凶猛的数据是怎么提交到我们的服务器上的？是手工提交还是另有秘密武器？如果是用软件实现的，那么其实现原理是什么，又应该怎么防止？

为了解决这些头疼的问题，我们有必要先了解其产生的过程，然后有针对性地进行防御。知己知彼，百战不殆。

1.浏览器工作流程

其实，浏览器就是一个实现了HTTP协议的客户端软件，浏览器的工作流程如图4-4所示。

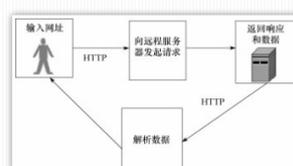


图 4-4 浏览器的工作流程

在整个过程中，浏览器扮演的角色始终是一个忠实的执行者。据此，我们很容易就能想到，只要遵循HTTP协议和服务器进行交互，就实现了一个最简单的浏览器，这个浏览器只提供对数据的收发而不提供解析功能。而对于服务器端的代码而言，是无法判断是真正的浏览器还是只是一个虚拟的浏览器。

2.PHP中和HTTP相关的函数

那怎么发送HTTP请求呢？可以使用代码发送HTTP头，如服务器端的PHP、客户端的AJAX，也可以使用各种抓包软件构造HTTP Request包。

在这之前，先介绍一些PHP中与HTTP协议相关的一系列函数。

`array get_headers (stringurl[, intformat])` 函数：取得服务器响应一个HTTP请求所发送的所有标头。通常用此函数请求一个URL，根据其返回的数据判断状态码是否为200，即可判断所请求的资源是否存在。

file系列函数: 包括fopen、file_get_contents等, 可以用来操作文件, 也可以请求一个网络上的资源。

stream_*系列函数: 发送请求, 包括但不限于HTTP协议。

socket系列函数: 通过Socket发送和请求数据, 包括但不限于HTTP协议。

cURL扩展库: PHP的一个扩展, 这是一个封装的函数库。可以用来模拟浏览器和服务进行交互, 功能比较强大。

header函数: PHP中可用此函数发送原始的HTTP头。需要注意的是, 这个函数之前不能有输出以及空格等。

这里用最简单的方式, 即使用PHP中的http_build_query和file系列函数发送HTTP请求。

我们经常用file_get_contents打开文件, 实际上用这个函数还可打开一个网络地址, 实现简单的网页抓取。用file_get_contents或者fopen、file、readfile等函数读取URL的时候, 会创建一个\$http_response_header变量保存HTTP响应的报头, 使用fopen等函数打开的数据流信息可以用stream_get_meta_data获取。简单的HTTP协议如代码清单4-1所示。

代码清单4-1 简单的HTTP协议使用示例

```
<? php
$html=file_get_contents (' http: //www.baidu.com/ ');
print_r ($http_response_header);
$fp=fopen (' http: //www.baidu.com/ ', ' r ');
print_r ( stream_get_meta_data ($fp));
fclose ($fp);
? >
```

3.模拟灌水机器人

PHP 5中新增的context参数使这些函数更加灵活, 通过该参数可以定制HTTP请求, 甚至POST数据。我们就利用这个函数向服务器发送请求。下面模拟一个机器人向一个博客发送留言。

首先打开某博客页面, 查看评论表单的字段, 为发帖机器人做准备, 如图4-5所示。

从图4-5可以看出, 一共有三个字段要填写, 且表单使用POST方法提交。构建表单值并提交, 如代码清单4-2所示。



图 4-5 表单字段

代码清单4-2 构造HTTP灌水机器人

```
<? php
$data=array ( ' author ' => ' 白菜大侠 ', ' mail ' => ' info@aiyooyo.com ', ' text ' => ' 博主很给力。 ');
$data=http_build_query ($ data );
$opts=array (
' http ' => array (
' method ' => ' POST ',
' header ' => "Content-type: application/x-www-form-urlencoded \r \n".
"Content-Length: ".strlen ($ data) ." \r \n",
' content ' => $ data )
);
$context=stream_context_create ($ opts );
$html=
@file_get_contents ( ' http : //aiyooyo.com/index.php/archives/7/comment ' ,
false,$ context );
? >
```

这段代码遵循HTTP请求的格式构造了一段报文。注明了请求方式为POST, 请求内

容为data。

注意 `http_build_query`函数并不是必需的, 这个函数仅仅是把传入的数组元素用`&`符号连接起来并编码, 也可以自己手工构造。这里使用这个函数仅仅是为了方便而已。

运行上面的代码, 刷新留言板。我们发现留言并没有提交, 怎么回事? 代码错了吗? 还是构造的请求不符合HTTP协议规范? 这里使用Firefox下最优秀的调试工具Firebug进行测试。打开Firebug, 在真实环境下发个评论, 看真实环境中页面向服务器提交了什么, 如图4-6所示。



图 4-6 Firebug抓到的数据

从图4-6中可以看到, 真实页面向服务器提交的数据如下:

```
author= %E7%99%BD %E8%8F %9C & mail=info %40aiyooyoo.com & url= & text=
%E6%88%91%E5%B0%B1%E8%A6%81%E6%9D %A5%E7%81%8C %E6%B0%B4%EF %BC
%8C%E8%B0%81%E4%B9%9F%E6%8C%A1%E4%B8%8D%E4%BD%8F%7E%7E
```

这一串就是当前页面向服务器传送的真实数据(注意: 中文和特殊字符被编码了)。从HTML代码我们知道, URL是一个非必需字段, 可写可不写, 所以可以确认POST参数这部分是没有问题的。

但是这还不够, 因为POST发送HTTP数据可能还需要Cookie、UA头等, 服务器才能识别, 判断结果是“这才是我要的数据”。至于是手工提交的还是机器提交的, 它是分不清的。看来上述问题可能是刚才构造的数据不够完整造成的。真实的header部分如图4-7所示。



图 4-7 Firebug中的请求头信息

把有用的数据添加到header部分以实现灌水机器人，如代码清单4-3所示。

代码清单4-3 灌水机器人的实现

```
<? php
$data=array ( ' author ' => ' 白菜大侠 ', ' mail ' => ' info@aiyooyo.com ', ' text ' => ' 博主很给力。 ');
$data=http_build_query ( $ data );
$opts=array (
' http ' => array (
' method ' => ' POST ',
' header ' => "Content-type: application/x-www-form-urlencoded \r \n".
"Content-Length: ".strlen ( $ data ) . " \r \n".
"Cookie: PHPSESSID=15vg6jsbbj5n0cjlg7pbioai85". " \r \n".
"User-Agent: Mozilla/5.0 ( Windows; U; Windows NT 6.1; zh-CN; rv: 1.9.2.13 ) Gecko/
20101203 Firefox/3.6.13". " \r \n".
"Referer: http: //aiyooyo.com/index.php/archives/7". " \r \n",
' content ' => $ data )
);
$context=stream_context_create ( $ opts );
$html=@file_get_contents ( ' http: //aiyooyo.com/index.php/archives/7/comment ' ,
false, $ context );
? >
```

Cookie中加了当前页面所必需的参数，还加了UA、Referer等参数。

提示 至于为什么要加这些参数、要加多少、为什么不用加Accept参数等问题都是经过使用总结出来的。因为很多参数实际上都是固定的而且非必需的，只提交最主要的参数即可。

以上代码不一定是完美的, 仅作演示而已。现在再来看看效果, 果然很“给力”, 确实可以发送了。一个粗糙的灌水机器人就制作成功了, 如图4-8所示。

整个过程耗时不到5分钟。可见file_get_contents是个好东西, 如果要做简单的页面抓取和数据提交, 可以考虑用这个函数。复杂的应用则需要使用cURL。但它们本质是一样的, 都需要了解HTTP协议。

4.使用抓包软件构造和提交HTTP请求

以Fiddler为例看看使用抓包软件怎么构造和提交HTTP请求。

假设已经安装了Fiddler, 这款软件安装时需要.NET运行时环境, Vista以上系统自带.NET运行时环境。在真实的环境下发表一个评论, 如图4-9所示。

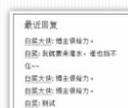


图 4-8 灌水机器人运行效果



图 4-9 表单示意

打开Fiddler, 默认是捕获所有数据包, 选菜单File→Capture Traffic命令, 取消默认的捕获状态。

提示 这一步操作不是必需的, 主要是为了停止捕获不需要的数据包, 减少后续的工作量。

由于发表帖子是发送请求, 并且是操作的起点, 所以这个请求通常就是第一个数据包。找到此数据包, 在右边查看Raw (原始数据), 由此可确认, 这就是刚才发出的HTTP请求, 如图4-10所示。



图 4-10 Fiddler抓取到的数据

在右边选择“Request Builder”的TAB选项，打开构造请求界面，选择Raw方式，即提交原始的HTTP请求，把上一步的数据复制到数据框中，如图4-11所示。



图 4-11 Fiddler中查看原始HTTP数据

单击“Execute”，数据包即发送到远程客户端。为了加强效果，可以多提交几次。打开网站后台看一下是否真的发送成功了，效果如图4-12所示。



图 4-12 程序运行结果

可以看到，我们发送的数据已经完全被服务器端接受了。

提示 使用代码提交HTTP请求比较麻烦，但是对流程的控制更强。比如用for循环不停地发送请求，效果更好。后面一种方式操作简单，不需要编码，对原始数据的操控性更强。二者各有优势，综合使用能逐渐从应用中了解HTTP协议。

4.1.4 垃圾信息防御措施

由上面的讲解就可以知道垃圾评论的大体来源了，那么怎么防御这样的攻击呢？

总结一下，防止这类垃圾评论与机器人的攻击的手段如下：

1) IP限制。其原理在于IP难以伪造。即使是对于拨号用户，虽然IP可变，但这也会大大增加攻击的工作量。

2) 验证码。其重点是让验证码难于识别，对于“字母+数字”的验证码，关键在于形变与重叠，增加其破解中切割和字模比对的难度，人眼尚且难以辨识，机器就更难处理了，再者是加大对于验证码的猜测难度。

3) Token和表单欺骗。通过加入隐藏的表单值或者故意对程序混淆表单值，进而达到判断是真实的用户还是软件提交的目的。

4) 审核机制。加大了管理人员的工作量，但理论上可以完全阻止垃圾评论，这是最无奈也是最有效的策略。

1.IP限制

HTTP协议是透明的、公开的，服务器端根本无法区分来源是真实的提交还是伪造的。所以通过判断Referer等手段是于事无补的，但是HTTP也有自己的局限。由于HTTP协议是应用层的协议，是基于TCP/IP协议的，故一些底层的東西HTTP协议是无法伪造的，比如IP。

IP在TCP层传递，其传输需要通过TCP的“三次握手”。在这个握手过程中，有一个校验过程，因此IP是很难伪造的。而HTTP层属于顶层，无法控制IP，所以最简单有效的办法就是对IP进行限制。

但是，不少代码会判断HTTP头中的HTTP_X_FORWARDED_FOR是否是代理过来的，若是，则把其记为IP。但是，HTTP_X_FORWARDED_FOR来自于HTTP请求中的X Forwarded For报头，而这个报头是可以修改的。这就可能造成潜在的攻击。所以合理的判断是完全不考虑代理，而使用SERVER变量中的HTTP_

CLIENT_IP或REMOTE_ADDR。二者是难以伪造的。出于安全考虑，凡是通过代理过来的请求或者HTTP头中包含X Forwarded For报头的，很多程序采取了拒绝的方案。这种处理方式比较简单，误差也较小。

2.Token法

要防止攻击关键就在于加大攻击的难度。最简单的是放一个隐藏可变的Token，每次提交都需要和服务器校对，若通不过，则为外部提交。

下面的代码称为Token法，可以阻止一些简单的重复，如代码清单4-4所示。

代码清单4-4 token.php

```
<? php
define ( ' SECRET ', "67%$#ap28");
function m_token () {
    $str=mt_rand ( 1000, 9999);
    $str2=dechex ( $_SERVER[ ' REQUEST_TIME ' ]-$str);
    return $str.substr ( md5 ( $str.SECRET ), 0, 10 ) .$str2;
}
echo m_token ();
echo ' <br/> ';
function v_token ( $str,$delay=300 ) {
    // $delay表示时间延迟，在不同的程序根据业务来自行修改
    $rs=substr ( $str, 0, 4 );
    $middle=substr ( $str, 0, 14 );
    $rs2=substr ( $str, 14, 8 );
    return ( $ middle== $ rs.substr ( md5 ( $ rs.SECRET ), 0, 10 ) ) && ( $_SERVER[ '
REQUEST_TIME ' ]-hexdec
    ( $rs2) - $rs < $delay );
}
var_dump ( v_token ( m_token ( ) ));
? >
```

这样，就造成了其构造HTTP请求的困难，使其难于通用。

3.验证码

对于预防一些灌水机器人，主要采取验证码一类的措施，包括中文验证码、回答验证，但是对于这两点，专业的灌水机器人都可以突破。但是对于技术含量不高的，可以起到一定的阻止作用，同时也降低了用户体验。

灌水机器人通常都会抓取页面的FROM表单，分析必填项与非必填项，对于必填项构造请求。如果存在普通的图形验证码，则启用图形识别模块、破解验证码、构造完整的数据包。对此，可以建立一个INPUT，用CSS或者间接地通过JavaScript设其页面为不可见，如果服务器收到的数据中包含有这个控件的值，那么肯定是来自机器提交。这样也能起到一定的效果。另外，对INPUT的值进行欺骗对博客的垃圾评论能起到一定的作用。例如：

用户名：或者可以做成图片防止机器的学习。

`<input type=text name=email/>` 实际上代表用户名。

`<input type=text name=url/>` 实际上是E-mail。

`<input type=text name=author/>` 实际上是URL，而且是隐藏的不能有任何输入的字段。

在服务器端，如果 `_POST[' email ']` 匹配的是一个电子邮件地址，那么其一定是灌水机器人。如果author字段有值，那么其也一定是灌水机器人。这叫做机器学习欺骗。同理，可以定期变更action的提交地址，加大灌水机器人的学习难度，也可以把所有数据改为后台审核。

但是，经过灌水机器人的学习和模型修改，过一段时间必将卷土重来。阻止外部提交一直是个难题。对于表单，因为它是可见的，所以很难阻止机器的猜解和模拟。目前，可行的办法是使用Active控件，使用对称加密和服务器通信，因为Active是不透明的。但是这样也存在问题，主要是IE Only和技术难度高，大多数的网站无法使用这样的技术，特别是个人站点。

使用JavaScript进行加密验证和非平衡图形验证码，可以阻止99%的外部提交。腾

讯的大部分产品都使用了这种技术，比如微博和邮箱。图4-13所示为移动139邮箱注册页面验证码。



图 4-13 移动139邮箱注册页面的验证码

移动的验证码比较有创意，其答案是四选一，猜中的概率为25%，这似乎并不能有效地阻止机器人的猜测。但是由于其需要手机验证码的参与，这是很难伪造的，或者说伪造成本很大，因此有很高的安全性。

Matlab中文论坛的注册验证问题如图4-14所示。



图 4-14 Matlab中文论坛的注册验证码

验证码看似用户不友好，但由于其本身就是专业论坛，“不专业”的注册会员并非其所需要的，因此不存在用户不友好这一说法。

提示 这两种验证机制都是可取的。有心的读者或许会联想到网络上流传的“最变态的验证码”系列图片，思路其实是一样的。

4.2 抓包工具

在介绍HTTP协议的时候涉及了一些抓包工具，并且简单地使用它进行了HTTP协议的学习。本节将进一步学习抓包工具的使用，并利用本节所学的知识介绍模拟登录和采集的开发思路和过程。

4.2.1 抓包工具分类

按照软件的功能，我们把抓包工具分为两类：

常规抓包工具：以IRIS、Wireshark为代表，这类软件可以抓取到整个局域网内所有的数据包，主要工作在数据传输层。

专用抓包工具：只抓取某一类协议，通常工作在应用层，最常见的就是对HTTP协议的抓取，如HttpWatch、Fiddler、IE Analyzer、Charles等。

由于Fiddler功能丰富，支持HTTP断点调试，并且是免费软件（作者一直在维护和更新），是所有同类软件中更新最及时、体积最小、功能最强大的免费软件之一。本节重点介绍Fiddler。

4.2.2 Fiddler功能与原理

Fiddler是用C#编写的免费HTTP/HTTPS网络调试器，以代理服务器的方式监听系统的网络数据流动。运行Fiddler后，就会在本地打开8888端口，网络数据流通过Fiddler进行中转时，可以监视HTTP/HTTPS数据流的记录并加以分析，甚至还可以修改发送和接收的数据。

Fiddler提供了清除IE缓存、请求构造器、文本编码转换工具等一系列工具，对前端开发工作很有价值。其工作原理是在浏览器（或者其他使用HTTP协议的进程）和服务器之间扮演代理的角色，这样所有的通信都要经过它。Fiddler工作原理如图4-15所示。

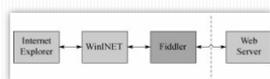


图 4-15 Fiddler工作原理

Fiddler以8888端口开本地代理服务器，并且支持HTTPS。所以，只要你的HTTP通信将代理设置为本地8888，Fiddler都能帮助你截获数据，然后中转给服务器或客户端。其最大的一个特点是可以中途修改HTTP通信内容。

提示 Sniffer和Fiddler的工作原理是一样的，但工作的网络层不同。

4.2.3 安装Fiddler

Fiddler下载地址：<http://www.fiddler2.com/fiddler2/version.asp>，约660KB，当前最新版本是V2.3.9.0。Fiddler支持Windows 2000/XP/2003/Vista/Windows 7操作系统，但需要电脑安装Microsoft.NET Framework v2.0以上版本的.NET运行库。高版本的操作系统如Windows 7已经自带了.NET运行库，如果是Windows XP、Windows 2003等，需要自行安装或更新本地.NET运行库。

下载Fiddler后双击安装程序，按照提示安装到指定无冲突即可。安装完成后运行软件，在浏览器中输入“<http://127.0.0.1:8888/>”，如果输出如图4-16所示界面，说明安装成功。



图 4-16 Fiddler运行截图

如果此时浏览网页，Fiddler就会抓取到你和服务器之间交流的数据包，如图4-17所示。

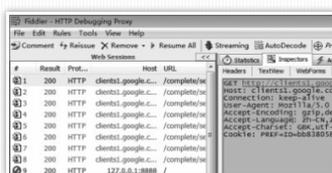


图 4-17 Fiddler抓取到的数据

打开Fiddler后，抓包默认是启用的。通常不需要特殊设置，在各种浏览器下Fiddler都能顺利工作。实际上，Fiddler和浏览器无关。读者在自己本机使用Fiddler时，可能抓取不到某个浏览器下的包，这不是Fiddler本身的问题，而是浏览器设置的问题。这种情况下，只需检查浏览器的代理设置。例如设置Firefox代理，选择“使用系统代理设置”，其他浏览器设置同理，如图4-18所示。



图 4-18 设置Firefox代理

4.2.4 Fiddler基本界面

Fiddler最基本的功能就是抓包与观察数据，下面简单介绍其界面和使用方法。

Fiddler的界面分为左右两栏：

左边为Web Sessions记录（注意，这里的Session表示一次HTTP对话，和PHP里提到的一般意义上的Session不是一个概念，这里只是沿用软件中的称呼），记录每个数据包的序号、Host、URL、资源类型、HTTP状态码、缓存状态等基本信息。

右边划分为上下两部分，上部分为请求数据，下部分为响应数据。通常在左端的Session列表里找到请求的URL，单击即可在右边看到请求数据的详细信息。常用的三种查看方式如下：

Header（header头格式）。

Raw（HTTP协议标准格式）。

Hex View（十六进制数据流）。

在Session栏里选择某一条请求，右击，通过弹出的快捷菜单可以对其执行保存、标记、删除、重放、注释等操作，如图4-19所示。

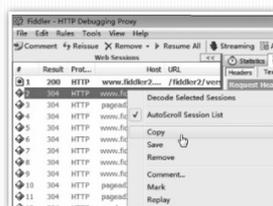


图 4-19 标记Session

下面简单介绍这几个常用功能。

保存：保存Session数据，方便以后查看或者做资源重定向。

标记：标记醒目的颜色方便查看，例如图中第一条Session标记为红色。

删除：删除不重要的或者认为可能会影响到你的Session。

重放：再次执行，这个Session的数据包将会重新发送一次。

注释：给Session添加注释。注释将显示在此Session的Comment列。

另外，在进行HTTP协议分析时，对于image、CSS类型的静态请求通常无助于我们进行协议分析，但这类Session往往还比较多，会干扰我们寻找需要的Session，故需要删除这类Session。是一条一条选择删除吗？可以这样做，但是效率不高。Fiddler提供了一个命令窗口，位于Session列表的最底部。Fiddler中内置了一些常用命令，方便管理Session，如图4-20所示。



图 4-20 Fiddler的命令窗口栏

要完成上述任务可以按以下步骤执行：

1) 在命令窗口输入 `select image`，自动选中所有image类型的Session，按 `delete` 键删除。

2) 输入 `select css`，选中所有css请求，按 `Delete` 键删除。

通过以上简单的两步，成功删除我们不需要的Session。如果当前网页有来自其他域名的Session，也会干扰我们的分析，可以输入 “`@google.hk.com`” 选中来自Google的Session，并删除。当要删除全部Session时，只需要输入 “`cls`” 并回车即可。Fiddler还支持其他命令，可以在官方网站上找到详细的帮助文档。

我们注意到Session列表的序号前都有一个小图标，其代表什么意义呢？

表4-1列举了各类Session前面请求图标所代表的含义。

图 标	含 义
	请求正在被发送到服务器
	正在从服务器下载响应
	请求被设置断点
	响应被设置断点
	请求使用了 HTTP HEAD 方法, 故响应应该为空
	该请求使用了 HTTPS 加密传输
	响应体为 HTML 文档
	响应体为图像
	响应体为脚本
	响应体为 CSS 样式表
	响应体为 XML 文档
	一个普通的请求响应成功
	响应状态码是 HTTP/300, 301, 302, 303 or 307 重定向
	响应状态码是 HTTP/304, 表示使用了缓存
	响应是一个来自于客户端代理的请求
	服务器端响应错误, 如 404
	Session 异常终止

掌握了Fiddler的Session管理以及HTTP协议, 即能使用好这款工具。

4.2.5 使用Fiddler进行HTTP断点调试

前面已经讲过了Fiddler的工作原理，相比其他HTTP抓包工具，Fiddler的优势就在于其特有的工作模式使其支持HTTP断点调试，这在很多场合是很有用的。下面通过一个例子学习。以Fiddler官方提供的测试页为例，其地址为：<http://www.fiddler2.com/sandbox/shop/>。在这个页面，下拉框选择“1”，单击check out提交请求。这是一个再正常不过的请求了，我们的输入通过表单被传递给服务器，服务器进行处理后返回给我们。图4-21是其返回界面。



图 4-21 返回界面

再看Fiddler中所记录请求数据包。单击左边的Session，右边窗体的数据如图4-22所示。

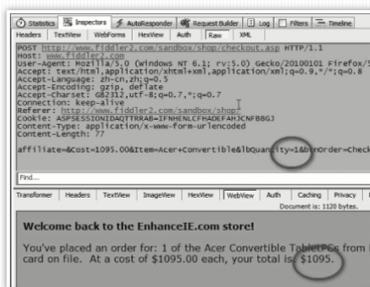


图 4-22 Fiddler抓取到的数据

注意图中标注的信息，lbQuantity是表单的参数，其值“1”是我们选择的，经服务器计算后得到结果为“1095”，并返回给浏览器。

接下来要给此响应设置断点，步骤如下：

- 1) 在菜单中选择Rules→Automatic Breakpoints→After Responses。
- 2) 回到订购页面，再次选择“1”，单击check out按钮。

3) 现在请求发出。经过Fiddler代理发送到服务器，服务器返回响应数据到Fiddler代理。

4) 此时由于设置了Responses断点，响应被挂起，就能在Fiddler中修改响应数据。

5) 提交。

此时响应由代理发送到客户端，就能看到响应数据了，如图4-23所示。

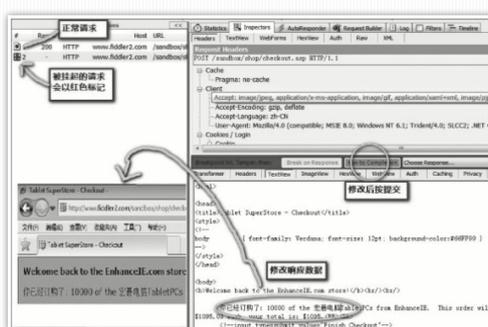


图 4-23 修改数据

可以看到，通过设置响应后（严格地说，应该是服务器响应到达Fiddler后，返回给浏览器前）断点，把HTTP发回来的响应中断并修改后才将它返回给浏览器。

还可以在请求发送前（严格地说，应该是客户端请求发送给Fiddler后，到达服务器前）设置断点，修改请求头，Fiddler代理将会把修改后的请求发送到服务器，然后读取服务器响应，中转并返回响应数据，如图4-24所示。

这两个特性对于调试AJAX程序特别有用，可以用来中断httpxml请求并修改请求内容、修改GET或者POST的数据以及header头等。这一点，也许你会觉得必要性不是太大，也许你会说：我可以自己在页面修改请求数据多次提交测试啊。但是再想想，Fiddler能够做到下面几条，这些在页面实现则较麻烦。



图 4-24 运行结果

- 1) 修改HTTP请求的原始数据，如UA、Cookie等。
- 2) 构造特殊请求数据。某些网页会通过使用JavaScript来限制用户在页面的数据输入。而Fiddler可以通过直接修改HTTP请求中的data突破限制，随意提交数据。
- 3) Fiddler通过拦截响应数据进行中断，可以修改响应体。

AJAX中有个回调函数，通常这个回调函数会根据服务器的返回值进行相应的客户端处理。如下面的代码所示：

```

$ ( function () {
    $ ( "#student \ \ .idcard" ) .click ( function () {
    if ( $ ( "#student \ \ .card" ) .val () ! =
    "" & & $ ( "#student \ \ .card" ) .val () .substr ( 0 , 1 ) == 0 ) {
    $ .get ( "/order/ajax/randcard.do" , function ( data ) {
    switch ( data . _rc ) {
    case "success" :
    $ ( "#student \ \ .card" ) .val ( data . randCard );
    break ;
    default :
    alert ( "处理失败" ); //other deal.....
    break ;
    }
    } );
    }
    } );
    } );
} );

```

这是一段生成随机不重复流水号的代码，在输入框里输入0后单击，即可通过AJAX请求服务器，返回一段JSON数据，然后根据JSON里的data._rc这个key的value进行不同的处理。如果是success，把返回的结果赋给该输入框，否则提示处理失败。

现在可以给该响应设置断点，修改返回的JSON数据，模拟各种情况，进而观察代码的运行情况。

上面这三个优势是不是很令你心动？你是不是灵光一闪地想到，从某种意义上来说，Fiddler这个断点功能可以帮助我们测试程序的安全性和健壮性呢？如果想到了这一点，再深入地想下去，我想你会得到更多。

除此之外，Fiddler还可以修改UA头。目前Fiddler支持对十几种浏览器UA的修改，包括IE 6~9、Firefox、Chrome、iPad、Windows Mobile以及Google的爬虫等，也可以自定义UA。使用这个功能，就能轻易地测试一些针对特定浏览器的hack，以及浏览一些需要手机才能浏览的页面。

4.3 Socket进程通信机制及应用

Socket通常称为“套接字”，用于描述IP地址和端口，是一个通信链的句柄。应用程序通过套接字向网络发出请求或者应答网络请求。Socket既不是一个程序，也不是一种协议，其只是操作系统提供的通信层的一组抽象API。

4.3.1 进程通信相关概念

进程通信的概念最初来源于单机系统。由于每个进程都在自己的地址范围内运行，为保证两个相互通信的进程之间既互不干扰又协调一致工作，操作系统为进程通信提供了相应设施，如UNIX BSD中的管道（pipe）、命名管道（named pipe）和软中断信号（signal），以及UNIX System V的消息（message）、共享存储区（shared memory）和信号量（semaphore）等，但这些都仅限于用在本机进程之间的通信。网间进程通信要解决的是不同主机进程间的相互通信问题（可把同机进程通信看作是其中的特例）。为此，首先要解决的是网间进程标识问题。同一主机上，不同进程可用唯一进程号（Process ID）标识。

网络环境下，各主机独立分配的进程号不能唯一标识该进程。例如，主机A赋予某进程号5，在B机中也可以存在5号进程，因此，“5号进程”就没有意义了。

操作系统支持的网络协议众多，不同协议的工作方式不同，地址格式也不同。因此，网间进程通信还要解决多重协议的识别问题。

为了解决上述问题，TCP/IP协议引入了下列概念。

1. 端口

网络中可以被命名和寻址的通信端口，是操作系统可分配的一种资源。

按照OSI七层协议的描述，传输层与网络层在功能上的最大区别是传输层提供进程通信能力。从这个意义上讲，网络通信的最终地址就不仅仅是主机地址了，还包括可以描述进程的某种标识符。为此，TCP/IP协议提出协议端口（Protocol Port，简称端口）的概念，用于标识通信的进程。

端口是一种抽象的软件结构（包括一些数据结构和I/O缓冲区）。应用程序（即进程）通过系统调用与某端口建立连接（binding）后，传输层传给该端口的数据都被相应进程所接收，相应进程发给传输层的数据都通过该端口输出。在TCP/IP协议的实现中，操作端口类似于一般的I/O操作，进程获取一个端口，相当于获取本地唯一I/O文件，可以用一般的读写原语访问。

类似于文件描述符，每个端口都拥有一个端口号，都是整数型标识符，用于区别不同端口。由于TCP/IP传输层的TCP协议和UDP协议是完全独立的两个软件模块，因此各自的端口号也相互独立，如TCP有一个255号端口，UDP也有一个255号端口，二者并不冲突。TCP与UDP段结构中端口的地址都是16比特，有0~65535个端口号。

对于这65536个端口号有以下使用规定：

端口号小于256的定义为常用端口，服务器一般都是通过常用端口号识别。任何TCP/IP实现所提供的服务都用1~1023之间的端口号，这是由IANA管理的。

客户端只需保证该端口号在本机上是唯一的。客户端端口号因存在时间很短暂，又称临时端口号。

大多数TCP/IP实现给临时端口号分配1024~5000之间的端口号。大于5000的端口号是为其他服务器预留的。

常见的端口有FTP的21号端口，HTTP服务的80号端口，SMTP（简单邮件传输服务，后面会详细介绍）的25号端口等。

2.地址

网络通信中通信的两个进程分别处在不同的机器上。遵循以下原则：

某台主机可与多个网络相连，必须指定一个特定网络地址。

网络上每台主机应有其唯一的地址。

每台主机上的每个进程应有在该主机上的唯一标识符。

通常主机地址由网络ID和主机ID组成，在TCP/IP协议中用32位整数值表示；TCP和

UDP均使用16位端口号标识用户进程。

3.连接

两个进程间的通信链路称为连接。连接表现为一些缓冲区和一组协议机制。

4.3.2 Socket演示：实现服务器端与客户端的交互

在讲解Socket的创建之前，先演示一下Socket为何物。这里用Java实现一个Socket的服务器端与客户端（也可以用任何其他支持Socket操作的语言实现），然后用PHP做为客户端请求该套接字。

在服务器端使用Socket开一个服务，端口是8001，这样就可以与多个客户端进行连接了。在客户端，向该Socket发送一条消息，服务器端在收到消息后，会根据情况进行一定的处理，返回给客户端，同时在服务器端打印所有收到的消息，如图4-25所示。



图 4-25 用Java实现一个Socket的服务器端与客户端

从图4-25中看到，服务器端和客户端都采用Java实现；一旦开启服务器端，这个服务就会被注册到Windows的网络服务中，端口为8001。使用netstat命令打印本机各端口的网络连接情况，在打印列表里看到此服务已经被注册了。一旦有客户端连接此Socket，操作系统就会为客户端自动分配一个随机端口，用来和服务器端8001端口进行通信。

既然此服务已经被注册到操作系统中，实际上此服务和腾讯QQ、FTP等是一个级别的，用它能够完成的事情很多。为了验证，使用Telnet连接，如图4-26所示。



图 4-26 Socket客户端与服务器端间的交互

由于Socket是开放的、透明的，一旦运行，任何操作Socket的语言都可以访问这个

开放的服务。图4-26所示是使用Java访问Socket的，也可以使用PHP、C、Python等任何提供SocketAPI的语言访问此服务。

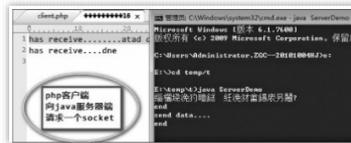
提示 Socket是一种服务，与其实现语言无关。基于这个性质，我们能实现不同服务之间、不同语言之间的互联互通。

代码清单4-5所示是PHP访问此Socket服务的代码。

代码清单4-5 PHP访问Socket

```
<? php
$sock=fsockopen ("192.168.0.2", 8001,$errno,$errstr, 1);
if (! $sock)
{
echo"$errstr ($errno) <br/> \n";
} else {
socket_set_blocking ($sock, false);
fwrite ($sock, "send data…… \r \n"); //注意：数据末尾需要加上" \r \n"提交此请求数据，否则可能将
//无法获取服务器端的回应，即使刷新缓冲也无效，这样就只有
//等到此连接关闭时才能获取到回应
fwrite ($sock, "end \r \n");
//使用end命令终止此客户端连接
while (! feof ($sock)) {
echo fread ($sock, 128);
flush ();
ob_flush ();
sleep (1);}
fclose ($sock);
}
```

运行结果如图4-27所示。



The screenshot shows two terminal windows. The left window is a PHP client script named 'demo.php' with the following content:

```
1 has receive.....atad  
2 has receive.....one  
3  
  
php客户端  
向java服务器端  
请求一个socket
```

 The right window is a Java server script named 'ServerDemo.java' with the following content:

```
Microsoft Windows [版本 6.0.6002.1.8000] (x86)  
版权所有 (c) 2009 Microsoft Corporation. 保留所有权利。  
C:\Users\Administrator.2009-2818180662>  
E:\cod temp\<br>E:\temp\>java ServerDemo  
服务端没打印任何数据  
read data.....  
read
```

图 4-27 程序运行结果

提示 本地进程间通过TCP通信，使用Wireshark等抓包工具是抓不到数据的。上面的例子中，客户端和服务器的数据无法抓取到。是因为回环接口的机制，这些包不会到达网卡，数据包直接被返回到传输层的输入队列中去了。而抓包工具要从网卡中获取数据。如果确实需要抓取这些数据，可以添加一条本地路由或者使用特殊抓包工具。如果是Linux系统，可以使用tcpdump命令来获取数据包。

4.3.3 Socket函数原型

看了这么多演示，大家对Socket应该有一个比较直观的认识了吧。Socket就是一种通信机制，类似于银行、电信这些部门的电话客服部门。打电话时，对方会分配一个坐席代表回答你的问题，客服部门就相当于Socket的服务器端，你就相当于客户端。在通话结束前，如果有人想找到和你通话的坐席代表是不可能的，因为你们正在通信，客服部门的电话交换机也不会重复分配。

Socket函数的原型定义如下：

```
SOCKET socket ( int af, int type, int protocol );
```

该函数共有三个参数：

af：指定应用程序使用的通信协议的协议族，对于TCP/IP协议族该参数置AF_INET，对于UNIX可建立本地Socket。

type：指定创建的Socket类型。有三种可选项。

流套接字类型 (SOCK_STREAM)：最常见的类型，基于TCP协议。

数据报套接字类型 (SOCK_DGRAM)：即UDP数据报。

原始套接字类型 (SOCK_RAW)：在IP层对套接字进行编程，实际上就是在IP层构造自己的IP包，然后把这个IP包发送出去。

protocol：指定应用程序所使用的通信协议。最常用的是TCP协议与UDP协议。

同样，可以把从TCP/UDP传输层过来的包抓取过来并进行分析。流套接字和数据报套接字不能完成的任务，可以在原始套接字中得以实现。所有语言提供的Socket API都是按照这个原型设计的。

提示 Socket从传输模式上又分为端对端和点对点的连接，流套接字和数据报套接

字都属于端对端的连接，因此需要绑定端口号。而原始套接字是基于IP协议的，属于点对点的传输模式，是没有端口这个概念的。比如常用的监测网络连接ping命令，就是基于ICMP协议的，它不存在端口概念。

4.3.4 PHP中的Socket函数

要创建基于Socket的应用程序，就需要详细地了解Socket的应用方法。这里以PHP为例介绍几个重要的Socket函数。

(1) resource socket_create

此函数用于创建一个Socket，代码如下：

```
resource socket_create ( int $ domain, int $ type, int $ protocol )
```

该函数共有三个参数，第一个参数指定Socket创建时所使用的通信协议族，其可选值和描述如表4-2所示。

参 数	描 述
AF_INET	基于 IP-v4 的 Internet 协议
AF_INET6	基于 IP-v6 的 Internet 协议，PHP 5.0 开始添加对其的支持
AF_UNIX	UNIX 本地通信协议

第二个参数指定Socket通信的交互类型，其可选的值如表4-3所示。

类 型	描 述
SOCK_STREAM	可靠的全双工连接，支持 TCP
SOCK_DGRAM	自动寻址信息功能，支持 UDP
SOCK_SEQPACKET	定序分程套接字
SOCK_RAW	构建传输层和网络层的原始套接字
SOCK_RDM	提供可靠的数据包传递

第三个参数指定Socket使用何种类型处理协议，包括ICMP、UDP、TCP，这里不再详细介绍。

(2) socket_bind

此函数用于将IP地址和端口绑定到socket_create函数所创建的句柄中。代码如下：

```
bool socket_bind ( resource $ socket, string $ address[, int $ port=0] )
```

socket_bind函数有三个参数:

第一个参数是必选参数, 其值是socket_create函数所创建的句柄。

第二个参数是必选参数, 其值是要绑定的IP地址。

第三个参数是可选参数, 其值是要绑定的端口号, 当socket_create函数所创建的第一个参数是AF_INET时, 需要指定这个参数。

(3) socket_listen

在绑定Socket后, 服务器端使用此函数监听客户端数据。函数原型如下:

```
bool socket_listen ( resource $ socket[, int $ backlog=0] )
```

第一个参数是socket_create函数创建的Socket句柄。

第二个参数是可选参数, 表示允许的最大连接数。

(4) socket_set_block

设置为非阻塞模式。函数原型如下:

```
bool socket_set_block ( resource $ socket )
```

非阻塞指在不能立刻得到结果之前, 该函数不会阻塞当前线程, 而会立刻返回。对应的概念是阻塞, 阻塞就是干不完不准回来, 必须得到对方的回应后才能继续下一步操作。特别是当用户比较多时, 设置成非阻塞是很必要的。如果是阻塞模式, 若两个客户端同时连接上, 服务器端在处理一个客户端请求时, 另外一个客户端的请求会被阻塞, 只有等到前一个客户端的事情处理完了, 后一个客户端的请求才会被响应。

(5) socket_write

使用此函数向Socket写入数据。函数原型如下：

```
int socket_write ( resource $ socket, string $ buffer[, int $ length=0] )
```

(6) socket_read

用此函数从Socket中读取指定长度的数据。函数原型如下：

```
string socket_read ( resource $ socket, int $ length[, int $ type=PHP_BINARY_READ] )
```

要注意第三个参数，指定要读取数据的类型，默认为PHP_BINARY_READ，安全读取二进制数据；另外一个值是PHP_NORMAL_READ，当遇到“r”或“\n”时停止。

(7) pfsockopen

实现长连接。Client方与Server方先建立通信连接，连接建立后不断开，然后再进行报文发送和接收。函数原型如下：

```
pfsockopen ( string $ hostname[, int $ port=-1[, int & $ errno[, string & $ errstr[, float $ timeout=ini_get  
("default_socket_timeout") ]]]])
```

(8) socket_set_option

设置Socket的控制选项，函数原型如下：

```
bool socket_set_option ( resource $ socket, int $ level, int $ optname, mixed $ optval )
```

例如设置 \$ socket发送超时1秒，接收超时3秒：

```
socket_set_option ( $ socket, SOL_SOCKET, SO_RCVTIMEO, array ( "sec"=>1, "usec"=>0 ));
socket_set_option ( $ socket, SOL_SOCKET, SO_SNDTIMEO, array ( "sec"=>3, "usec"=>0 ));
(9) socket_last_error
```

函数返回操作中任何socket函数产生的最后错误，返回值是一个int型的错误代号。函数原型如下：

```
int socket_last_error ( [resource $ socket] )
```

使用socket_strerror () 函数给出对错误码的字符串描述。具体定义在Windows和类UNIX系统略有差异，限于篇幅不再列举。PHP手册中也有列举，位于：

```
\ php 5.X.X \ ext \ sockets \ unix_socket_constants.h
\ php 5.X.X \ ext \ sockets \ win32_socket_constants.h
```

提示 要想深入Socket的内部实现机制是很困难的，作为一名非底层程序员，我们只要明白Socket是一套操作系统封装好的函数，会创建和调用就可以了。

代码清单4-6演示了在PHP里创建一个Socket的方法。

代码清单4-6 PHP创建Socket

```
<? php
$host="192.168.0.2";
$port=12345;
set_time_limit ( 0 ); //最好在CLI模式下运行, 保证服务不会超时
//创建Socket
$socket=socket_create ( AF_INET, SOCK_STREAM, 0 ) or die ( "Could not create socket
\n" );
///绑定socket到指定地址和端口
$result=socket_bind ( $socket, $host, $port ) or die ( "Could not bind to socket \n" );
//开始监听连接
$result=socket_listen ( $socket, 3 ) or die ( "Could not set up socket listener \n" );
//接收连接请求并调用另一个子Socket处理客户端——服务器间的信息
$spawn=socket_accept ( $socket ) or die ( "Could not accept incoming connection \n" );
//读取客户端输入
$input=socket_read ( $spawn, 1024 ) or die ( "Could not read input \n" );
//clean up input string
$input=trim ( $input );
//反转客户端输入数据, 返回服务端
$output=strrev ( $input ) . "\n";
socket_write ( $spawn, $output, strlen ( $output ) ) or die ( "Could not write output \n" );
//关闭sockets
socket_close ( $spawn );
socket_close ( $socket );
```

PHP的语言特性和自身定位决定了它只适合做客户端, 而不适合做服务器端。因为Socket主要面向底层和网络服务开发, 一般服务器端都是用C、Java等语言实现, 这样能更好地操纵底层, 对网络服务开发中遇到的问题(如并发、阻塞等)也有完善、成熟的解决方案, 而PHP显然不适合这种应用场景。

实际上, PHP操作MySQL数据库也是通过Socket进行的, 这正是由于Socket屏蔽了底层的协议, 使得网络服务之间的互联互通变得简单。

提示 除了传统的服务器端语言实现的Socket外, 随着HTML 5的流行, 浏览器客户端实现的WebSocket也逐渐兴起, 对于这一点值得关注。FlashSocket也是一个不错的解决方案。

4.3.5 Socket交互应用：使用Socket抓取数据

要在客户端操作Socket，可使用fsockopen、socket_create、stream_socket_client等函数实现。如果是PHP 5，建议使用stream_socket_client。

看一个实例，用Socket完成4.1.3节论坛灌水机器人的功能。

首先，到目的地（<http://typecho.org/archives/54>）真实提交一次，用Fiddler查看抓到的数据，如图4-28所示。



图 4-28 Fiddler抓包结果

使用Socket获取数据的实现，如代码清单4-7所示。

代码清单4-7 使用Socket获取数据

```
<? php
$post_=array ( ' author ' => ' 一直都在 ', ' mail ' => ' wait@qq.com ', ' url ' => ' ', '
text ' => ' 测试 ');
$data=http_build_query ($post_);
$fp=fsockopen ("typecho.org", 80,$errno,$errstr, 5);
$out="POST http://typecho.org/archives/54/comment HTTP/1.1\r\n";
$out.="Host: typecho.org\r\n";
$out.="User-Agent: Mozilla/5.0 ( Windows; U; Windows NT 6.1; zh-CN; rv: 1.9.2.13 )
Gecko/20101203 Firefox/3.6.13" . "\r\n";
$out.="Content-type: application/x-www-form-urlencoded\r\n";
$out.="Referer: http://typecho.org/archives/54/\r\n";
$out.="PHPSESSID=082b0cc33cc7e6df1f87502c456c3eb0\r\n";
$out.="Content-Length: ".strlen ($data) . "\r\n";
$out.="Connection: close\r\n\r\n";
$out=$data . "\r\n\r\n";
fwrite ($fp,$out);
while (! feof ($fp)) {
```

```
echo fgets ($fp, 1280);  
}  
fclose ($fp);
```

Web应用程序是无法区分机器人和人的，人和机器都是通过Socket提交数据，只不过人是通过浏览器调用操作系统的Socket提交，而机器人是通过自己写代码调用Socket提交。如果没有4.1.3节提到的防机器人设置，那么上面的代码可以很轻松地实现灌水。

最后，注意以下几点：

fsockopen的第一个参数hostname不要带“http://”字符串，除非使用SSL等。

Headers请求不一定都要按照抓包数据全部带上，除非调试不成功或者不熟练或者有特殊需求可以全部照搬，否则只带上几个核心的header。

在Connection和data后有两个换行，如图4-28所示。

有些表单请求可能有hidden值，务必仔细抓包。

注意编码问题。

前面说过，建议使用stream_socket实现。若使用stream_socket实现只改一行代码就行，如下：

```
$fp=stream_socket_client ("tcp://typecho.org: 80",$errno,$errstr, 3);
```

在PHP中，99.9%的Socket应用属于流套接字范畴，由于数据报套接字和原始套接字涉及比较底层的协议知识，这里就不介绍了，有兴趣的读者可以自行学习。

4.4 cURL工具及应用

在cURL还没有普及之前，常用Snoopy工具进行网络数据抓取。cURL是利用URL语法规则传输文件和数据的工具，支持很多协议，如HTTP、FTP、Telnet等。

cURL是一个通用的库，并非PHP独有。其实，很多功能用file、socket系列函数都可以实现，不过用cURL功能更全面，实现一些复杂的操作更简单，比如处理Cookie、验证、表单提交、文件上传等。

4.4.1 建立cURL请求的基本步骤

在学习更复杂的功能之前，先来看在PHP中建立cURL请求的基本步骤：

- 1) 初始化。
- 2) 设置选项，包括URL。
- 3) 执行并获取HTML文档内容。
- 4) 释放cURL句柄。

具体实现如代码清单4-8所示。

代码清单4-8 cURL的具体实现

```
<? php
//1.初始化
$ch=curl_init ();
//2.设置选项，包括URL
curl_setopt ($ch, CURLOPT_URL, "http: //www.php.net");
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1); //将curl_exec () 获取的信息以文件流
的形式返回，
//而不是直接输出
curl_setopt ($ch, CURLOPT_HEADER, 1);
//启用时会把头文件的信息作为数据流输出
```

```
//3.执行并获取HTML文档内容
$output=curl_exec ($ch);
//4.释放cURL句柄
curl_close ($ch);
echo $output;
```

很多时候并不需要header头，把CURLOPT_HEADER设为0或者不设置（默认为0）。

第二步最重要，也就是curl_setopt（）函数，一切玄妙均在此。有一长串cURL参数可供设置，它们能指定URL请求的各个细节。

要一次性全部看完并理解可能比较困难，这里只介绍一些常用方法，详情介绍可参考PHP手册。

4.4.2 检查cURL错误和获取返回信息

我们加一段检查错误的语句（虽然这并不是必需的）：

```
//.....  
$output=curl_exec ($ch);  
if ($output===FALSE) {  
echo"cURL Error: ".curl_error ($ch);  
}  
//.....
```

请注意，比较时用的是“===FALSE”，而非“==FALSE”。这是为了区分空输出和布尔值FALSE，因为布尔值才是真正的错误。另外，通过curl_getinfo()函数返回cURL执行后这一请求相关的信息，这对调试和排查错误是很有用的。代码如下所示：

```
//.....  
curl_exec ($ch);  
$info=curl_getinfo ($ch);  
echo '获取 ' . $info['url'] . ' 耗时 ' . $info['total_time'] . ' 秒 ';
```

返回的数组如代码清单4-9所示。

代码清单4-9 返回的数组

```
(  
[url]=> http: //www.php.net//资源网络地址  
[content_type]=> text/html; charset=utf-8//内容编码  
[http_code]=> 200//http状态码  
[header_size]=> 395//header的大小  
[request_size]=> 50//请求的大小  
[filetime]=> -1//文件创建时间
```

```
[ssl_verify_result]=>0//SSL验证结果
[redirect_count]=>0//跳转次数
[total_time]=>2.356//耗时
[namelookup_time]=>0//DNS查询时间
[connect_time]=>0.297//连接时间
[pretransfer_time]=>0.297//准备传输耗时
[size_upload]=>0//上传数据大小
[size_download]=>34738//下载数据大小
[speed_download]=>14744//下载速度
[speed_upload]=>0//上传速度
[download_content_length]=>-1//下载内容程度
[upload_content_length]=>0//上传内容长度
[starttransfer_time]=>0.921//开始传输耗时

[redirect_time]=>0//重定向耗时
[certinfo]=>Array//认证信息
(
)
```

这些信息在调试时是很有用的。当抓取数据出错的时候，就可以输出此数组查看具体的信息。例如，在cURL抓取的时候，可能由于网络等原因，时常出现抓取数据不完整的情况，我们就需要加一个校验。通过对所获取的数据计算filesize，然后和curl_getinfo获取的数据进行比较，如果大小相等，就认定下载正确，否则进行重复尝试。比如，三次尝试均不成功选择放弃或者放入失败队列，过一段时间再尝试。

看一个例子。使用cURL抓取网络上的一张图片，比较抓取图片的大小和文件头信息，目的是校验数据是否完整。详细代码如代码清单4-10所示。

代码清单4-10 cURL抓取图片

```
<? php
@header ( ' Content-type: image/png ');
//1.初始化
$ch=curl_init ();
//2.设置选项，包括URL
```

```
curl _ setopt ( $ ch, CURLOPT _
URL, "http: //renren.com/usr/uploads/2011/06/3230341841.png");
curl _ setopt ( $ ch, CURLOPT _ RETURNTRANSFER, 1); //将curl _ exec ( ) 获取的信息以文件流
的形式返回,
//而不是直接输出
//curl _ setopt ( $ ch, CURLOPT _ HEADER, 1);
//启用时会将头文件的信息作为数据流输出
//3.执行并获取内容
$output=curl _ exec ( $ ch);
//4.释放cURL句柄
$info=curl _ getinfo ( $ ch);
curl _ close ( $ ch);
file _ put _ contents ( "g: /bak/temp/1/a.png", $ output );
$ size=filesize ( "g: /bak/temp/1/a.png");
if ( $ size! = $ info[ ' size _ download ' ] ) {
echo ' 下载数据不完整 ';
//尝试再次下载, 最多三次不成功则放弃, 或加入失败队列
} else {
echo ' 下载数据完整, O ( n _ n ) O ~ ' ;
}
```

4.4.3 在cURL中伪造头信息

前面提到，头信息很重要，它是服务器端和客户端的身份证明和交流方式。本节用cURL模拟手机登录3g.qq.com。

首先在浏览器中输入“3g.qq.com”，会自动跳转到3gqq.qq.com，显示的不是我们要的内容。因为腾讯识别到我们在使用浏览器访问，自动转向其他地址，如图4-29所示。



图 4-29 手机腾讯网截图

要想实现手机访问的效果，就需要用cURL模拟手机UA访问它，如代码清单4-11所示。

代码清单4-11 cURL模拟手机UA

```
<? php
@header ( ' Content-type: text/html; charset=utf-8 ');
//第一次初始化
$ch=curl_init ();
curl_setopt ( $ch, CURLOPT_URL, "http://3g.qq.com");
curl_setopt ( $ch, CURLOPT_RETURNTRANSFER, 1);
$h=array ( ' HTTP_VIA : HTTP/1.1 SNXA-PS-WAP-GW21 ( infoX-WISG, Huawei
Technologies) ',
' HTTP_ACCEPT : application/vnd.wap.wmlscriptc, text/vnd.wap.wml, applica-
tion/vnd.wap.html+xml, appli
cation/xhtml+xml, text/html, multipart/mixed, */* ',
' HTTP_ACCEPT_CHARSET: ISO-8859-1, US-ASCII, UTF-8; Q=0.8, ISO-8859-15;
Q=0.8, ISO-
10646-UCS-2; Q=0.6, UTF-16; Q=0.6 ');
curl_setopt ( $ch, CURLOPT_HTTPHEADER,$h);
```

```

$output=curl_exec ($ch);
curl_close ($ch);
//第二次跳转
$ch=curl_init ();
curl_setopt ($ch, CURLOPT_URL, "http: //info50.3g.qq.com/g/s? aid=index& s_it=3
& g_from=
3gindex& & g_f=1283");
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt ($ch, CURLOPT_HTTPHEADER, $h);
$output=curl_exec ($ch);
curl_close ($ch);
echo $output;

```

运行结果如图4-30所示。



图 4-30 用cURL模拟手机UA访问结果

通过查看源代码可知，返回结果的确与手机设备相同。头信息来自诺基亚手机，通过访问一个输出\$_SERVER的页面获得。腾讯对不同的手机型号和分辨率使用了不同的视图。比如NOKIA 1681C手机，获得的信息如下：

```

HAZZ-PS-WAP1-GW14 ( infoX-WISG, Huawei Technologies ), HTTP_X_UP_DEVCAP_
SCREENDEPTH: 16 HTTP_X_UP_DEVCAP_SCREENPIXELS: 128, 160

```

“128，160”就是手机的屏幕尺寸。如果用宽屏手机如iPhone 4S或者安卓手机访问，由于其屏幕分辨率较大，得到的也应该是一个宽屏页面。

验证访问的页面是不是真的WAP页面，不需要看源代码，也不需要使用手机实际查

看，只要更换浏览器的UA头即可，而360浏览器、Opera等都是支持的，Firefox也有对应的插件。

注意 用这种方法“欺骗”移动、联通、电信的网站是不行的。以移动为例，因为实际上我们使用真实的手机访问网站时，所有数据都是被移动网关代理的，会带上一些特殊的头，如手机号、手机特征码、手机所在基站等，这些信息很难伪造（针对我们而言是一个黑盒），普通的网站是得不到这些特殊的头的（被屏蔽）。这就是为什么现在不能通过WAP页面获取手机号的原因。

如果你的网站需要获取来访者手机号，需要和移动签订合作协议，让移动网关对你的开发服务器所在IP放开约束。

移动公司由于拥有这些数据，就能开发手机定位等Web应用。因此，我们在自己的手机上，不输入手机号就能登录移动官网查询话费。

4.4.4 在cURL中用POST方法发送数据

当发起GET请求时，数据通过“查询字串”（query string）传递给一个URL。例如，在Google中搜索时，搜索关键字即为URL查询字串的一部分：

```
http: //www.google.com.hk/search? q=php
```

这种情况下可能并不需要cURL模拟，把这个URL丢给file_get_contents（）就能得到相同结果。

不过有一些HTML表单是用POST方法提交的。这种表单提交时，数据通过HTTP请求体（request body）发送，而不是查询字串。当然，前面介绍过用file和Socket系列函数处理POST，这里介绍cURL处理方式。可以用PHP脚本模拟这种URL请求。

首先，新建一个可以接受并显示POST数据的文件post_output.php：

```
print_r($_POST);
```

接下来，写一段PHP脚本执行cURL请求，如代码清单4-12所示。

代码清单4-12 PHP中使用cURL发送数据

```
$url="http: //localhost/post_output.php";  
$post_data=array (  
"foo"=> "bar",  
"query"=> "php",  
"action"=> "Submit"  
);  
$ch=curl_init ();  
curl_setopt ($ch, CURLOPT_URL, $url);
```

```
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);  
//设置为POST  
curl_setopt ($ch, CURLOPT_POST, 1);  
//把POST的变量加上  
curl_setopt ($ch, CURLOPT_POSTFIELDS, $post_data);  
$output=curl_exec ($ch);  
curl_close ($ch);  
echo $output;
```

执行代码后应该会得到以下结果:

```
Array  
(  
    [foo]=>bar  
    [query]=>php  
    [action]=>Submit  
)
```

这段脚本发送一个POST请求给post_output.php并返回，利用cURL捕捉了这个输出。

试着改写前面那个经典例子，向博客提交留言。现在已经使用file、Socket、cURL这三种方法实现了同一件事。

4.4.5 使用cURL上传文件

上传文件和POST十分相似，因为所有的文件上传表单都是通过POST方法提交的。首先新建一个接收文件的页面upload_output.php：

```
print_r($_FILES);
```

代码清单4-13所示是真正执行文件上传任务的脚本。

代码清单4-13 cURL上传文件

```
$url="http://localhost/upload_output.php";
$post_data=array (
"foo"=>"bar",
//要上传的本地文件地址
"upload"=>"@test.zip"
);
$ch=curl_init ();
curl_setopt ($ch, CURLOPT_URL, $url);
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt ($ch, CURLOPT_POST, 1);
curl_setopt ($ch, CURLOPT_POSTFIELDS, $post_data);
$output=curl_exec ($ch);
curl_close ($ch);
echo $output;
```

如果要上传一个文件，只需要把文件路径当作一个POST变量传过去，不过记得在前面加上@符号。执行这段脚本应该会得到如下输出：

```
Array
(
```

```
[upload]=> Array
(
  [name]=> test.zip
  [type]=> application/octet-stream
  [tmp_name]=> aabb.tmp
  [error]=> 0
  [size]=> 118364
)
```

提示 当POST的数据来自外部时，需要注意检查并过滤@符号，因为@符号在cURL中是有特殊作用的，而本身并不属于危险字符。

4.4.6 cURL批处理

cURL还有一个高级特性——批处理句柄（handle）。这一特性允许同时或异步地打开多个cURL连接。代码清单4-14所示是来自手册的示例代码。

代码清单4-14 cURL批处理示例代码

```
//创建两个cURL资源
$ch1=curl_init(); $ch2=curl_init();
//指定URL和适当的参数
curl_setopt($ch1, CURLOPT_URL, "http://lxr.php.net/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);
//创建cURL批处理句柄
$mh=curl_multi_init();
//加上前面两个资源句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);
//预定义一个状态变量
$active=null;
//执行批处理do {
$mrc=curl_multi_exec($mh,$active);
} while ($mrc==CURLM_CALL_MULTI_PERFORM);
while ($active&& $mrc==CURLM_OK) {
if (curl_multi_select($mh)! ==-1) {
do {
$mrc=curl_multi_exec($mh,$active);
} while ($mrc==CURLM_CALL_MULTI_PERFORM);
}}
//关闭各个句柄
curl_multi_remove_handle($mh,$ch1);
curl_multi_remove_handle($mh,$ch2);
curl_multi_close($mh);
```

这里要做的就是打开多个cURL句柄并指派给一个批处理句柄，然后只需在一个while循环里等它执行完毕。

这个示例中有两个主要循环：

第一个do……while循环重复调用curl_multi_exec()。这个函数是无隔断(non blocking)的，但会尽可能少地执行。它返回一个状态值，只要这个值等于常量CURLM_CALL_MULTI_PERFORM，就代表还有一些刻不容缓的工作要做(例如，把对应URL的HTTP头信息发送出去)。也就是说，需要不断调用该函数，直到返回值发生改变。

接下来的while循环，只在active变量为true时继续。这一变量之前作为第二个参数传给了curl_multi_exec()，代表只要批处理句柄中是否还有活动连接。接着调用curl_multi_select()，在活动连接(例如接受服务器响应)出现之前，它都是被“屏蔽”的。这个函数成功执行后，又会进入另一个do……while循环，继续下一条URL。

说明 很多人把这种方式称为cURL多线程处理，而curl_multi_exec并不是多线程，它属于异步处理的范畴。

4.4.7 cURL设置项

cURL有许多设置选项，这些选项才是cURL的灵魂。通过curl_setopt函数设置，原型如下：

```
bool curl_setopt ( resource $ch, int $option, mixed $value )
```

由于选项特别多，我们只介绍几个最常见而又很重要的。如果需要实现SSL传输、断点传输或者遇到获取页面出现乱码、欲获取服务器超时出错等，那么下面的选项对你会有很大帮助，如表4-4所示。

选项	描述
CURLOPT_AUTOREFERER	当根据 Location；重定向时，自动设置 header 中的 Referer；信息
CURLOPT_COOKIESESSION	启用时 cURL 会仅仅传递一个 Session Cookie，忽略其他 Cookie，默认状况下 cURL 会将所有 Cookie 返回给服务器端，Session Cookie 用来判断服务器端的 Session 是否有有效而存在的 Cookie
CURLOPT_FOLLOWLOCATION	启用时将服务器返回的“Location；”放在 header 中，递归地返回给服务器，使用 CURLOPT_MAXREDIRS 可以限定递归返回的数量
CURLOPT_HEADER	启用时将头文件的信息作为数据流输出
CURLOPT_RETURNTRANSFER	将 curl_exec() 获取的信息以文件流的形式返回，而不是直接输出
CURLOPT_INFILESIZE	设定上传文件的大小，单位为字节 (byte)
CURLOPT_MAXCONNECTS	允许的最大连接数量，超过会通过 CURLOPT_CLOSEPOLICY 决定应该停止哪些连接
CURLOPT_MAXREDIRS	指定 HTTP 重定向的最多数量，和 CURLOPT_FOLLOWLOCATION 一起使用
CURLOPT_COOKIE	设定 HTTP 请求中“Cookie；”部分的内容，多个 Cookie 用分号分隔，分号后带一个空格（例如，“test = apply; cookie = and”）
CURLOPT_COOKIEFILE	包含 Cookie 数据的文件名，Cookie 文件的格式可以是 Netscape 格式，或者只是纯 HTTP 头部信息存入文件
CURLOPT_COOKIEJAR	连接结束后保存 Cookie 信息的文件
CURLOPT_ENCODING	HTTP 请求头中“Accept-Encoding；”的值，支持的编码有“identity”，“deflate”和“gzip”，如果为空字符串“”，请求头会发送所有支持的编码类型
CURLOPT_POSTFIELDS	全部数据使用 HTTP 协议中的“POST”操作来发送，要发送文件，在文件名前面加上@前缀并使用完整路径，这个参数通过 urlencode 后的字符串类似 param = val1¶2 = val2...或使用一个以字段名为键值，字段数据为值的数组，如果 value 是一个数组，Content-Type 头将会被设置成 multipart/form-data
CURLOPT_RANGE	以“X-Y”的形式组成，其中 X 和 Y 都是可选项获取数据的范围，单位是字节，HTTP 传输线程也支持几个这样的重复项中间用逗号分隔如“X-Y,N-M”
CURLOPT_REFERER	HTTP 请求头中“Referer；”的内容

选项	描述
CURLOPT_HTTPHEADER	用来设置 HTTP 头字段的数组，数组形式如下： array (Content-type; text/plain; Content-length: 100)
CURLOPT_FILE	设置输出文件的位置，值是一个资源类型，默认为 STDOUT（浏览器）
CURLOPT_INFILE	在上传文件的时候需要读取的文件地址，值是一个资源类型
CURLOPT_HEADERFUNCTION	设置一个回调函数，其有两个参数：第一个是 cURL 的资源句柄，第二个是输出的 header 数据，header 数据的输出必须依赖这个函数，返回已写入的数据大小
CURLOPT_WRITEFUNCTION	拥有两个参数的回调函数：第一个是参数是会话句柄，第二是 HTTP 响应头信息的字符串，使用此回调函数，将自行处理响应头信息，响应头信息是整个字符串，设置返回值精确的已写入字符串长度，发生错误时传输线程终止

提示 如果觉得这个函数设置起来比较麻烦，使用curl_setopt_array函数可把所有设置项作为一个数组穿进去设置。

4.4.8 网络应用：使用cURL抓取腾讯微博

本小节综合运用前面所学的知识，使用cURL登录并且采集腾讯微博。

如果想要获取自己的微博内容，在微博服务提供商没有提供相应API时，我们能想到的方法就是模拟登录到我们自己的微博，然后抓取微博页面。本节以腾讯微博为例讲解通过cURL实现模拟登录的过程。如果读者能成功理解这一节的思路和代码，那么就会一通百通了，因为我想，没有比这个更复杂的例子了。

提示 尽管腾讯已经开放了微博API，对于那些没有开放API的应用，可以利用本实例的方法进行模拟登录、采集页面等操作。

先做一些准备工作：关掉其他无用的程序，防止干扰。打开Firefox的Firebug和Fiddler或者IE Analyzer（不推荐HttpWatch）准备抓包。

登录，抓包，发现一大堆数据（首页那个滚动的微博消息很烦人，能够产生一大堆的垃圾信息，注意避免受其干扰），停止抓包（这一步是为了防止截获到更多的数据包，增加辨别和处理工作量）。

说明 腾讯微博为了防止数字账号的枚举登录，做了一些限制，如果使用QQ号登录微博，会要求输入验证码，而使用注册账号登录，则可以避免输入。这就减少了工作量和难度。另外，如果一个账号频繁登录，腾讯就会强制要求输入验证码。对于这点，在初期的测试中，可能需要频繁重复这个登录过程，就会触发验证码机制。对于这点，一个策略是程序两次运行时间间隔不要太短，另一个策略是换其他账号测试。

1.分析抓包数据

首先看抓取到的第一条请求，下面是其请求的HTTP原始数据：

```
GET
http://ptlogin2.qq.com/login? u=@wait4you & p=0AC95B88839DF6AEB9DC1764BF3EBC88
& verifycode=! VKX& low_login_enable=1& low_login_hour=720& aid=46000101& u1=http%3A
%2F%2Ft.qq.com& ptredirect=1& h=1& from_ui=1& dumpy=& fp=loginerroralert HTTP/1.1
Host: ptlogin2.qq.com
```

```
User-Agent: Mozilla/5.0 ( Windows NT 6.1; rv: 5.0 ) Gecko/20100101 Firefox/5.0
Accept: */*
Accept-Language: zh-cn, zh; q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: GB2312, utf-8; q=0.7, *; q=0.7
Connection: keep-alive
Referer: http://t.qq.com/
Cookie:
ptvfsession-
=0b082f18cf393b9f4a6a28251fb0e5652271498e5f2f02cea686ed5bdb0decb086ad0072c3645
d77f24e7ec167b0c65d ; pgv _ pvid=2218166990 ; pgv _ flv=10.3 r181 ; pgv _
info=ssid=s8306834786 ; pgv _ r _ cookie=1171817943867 ; pt2gguin=o1281084745 ; ptisp= ;
ptuserinfo=e99988e88ba5e6b0b4 ;
ptcz=14f260c8f1b5708bf040f1288abc5a41bf683351da8e9f49274250ac1e23eda5 ; ptui _ login-
uin2=wait4 you; o_cookie=1281084745;
verifysession-
=h0024fe53e4dac61b1dcabd46b86de60ccd683b54dc8207bd88547efbf32d762856e99a5ff7a
0a07d8116e85e3c93a3509f
```

凭直觉和经验可以知道，这绝对不是平常所用的处理手段。在平常的应用中，都是直接POST输入表单值到服务器的，并且是明文。按照常规，我们看到的应该类似下面的数据：

```
POST: http://t.qq.com/login
data: u=wait4you&password=123456
```

很显然，腾讯在客户端动了手脚，唯一的可能就是在提交表单到服务器时，JavaScript拦截了默认的SUBMIT事件，对密码进行了处理，然后向服务器GET上面的地址，返回一个验证码。

我们看看这一步返回了什么，如下所示：

```
HTTP/1.1 200 OK
```

```
Date: Tue, 19 Jul 2011 00: 13: 39 GMT
Server: Tencent Login Server/2.0.0
Set-Cookie : pt2gguin=o1281084745 ; EXPIRES=Fri , 02-Jan-2020 00 : 00 : 00 GMT ;
PATH=/; DOMAIN=
qq.com;
Set-Cookie: uin=o1281084745; PATH=/; DOMAIN=qq.com;
Set-Cookie: skey=@M0runDjdY; PATH=/; DOMAIN=qq.com;
Set-Cookie : clientuin= ; EXPIRES=Fri , 02-Jan-1970 00 : 00 : 00 GMT ; PATH=/ ;
DOMAIN=qq.com;
Set-Cookie : clientkey= ; EXPIRES=Fri , 02-Jan-1970 00 : 00 : 00 GMT ; PATH=/ ;
DOMAIN=qq.com;
Set-Cookie : zzpaneluin= ; EXPIRES=Fri , 02-Jan-1970 00 : 00 : 00 GMT ; PATH=/ ;
DOMAIN=qq.com;
Set-Cookie : zzpanelkey= ; EXPIRES=Fri , 02-Jan-1970 00 : 00 : 00 GMT ; PATH=/ ;
DOMAIN=qq.com;
Set-Cookie: ptisp=; PATH=/; DOMAIN=qq.com;
Set-Cookie: luin=o1281084745; PATH=/; DOMAIN=qq.com; EXPIRES=Wed, 17-Aug-2011
16: 13:
39 GMT;
Set-Cookie:
lskey-
=00010000f3ee807847c962638171691f8116d23cd15297baa4175457e1a916b816486619271
bd8620096a7c2; PATH=/; DOMAIN=qq.com; EXPIRES=Wed, 17-Aug-2011 16: 13: 39 GMT;
Set-Cookie: ptuserinfo=e99988e88ba5e6b0b4; PATH=/; DOMAIN=ptlogin2.qq.com;
Pragma: no-cache
Cache-Control: no-cache; must-revalidate
Connection: Close
Content-Type: application/x-javascript; charset=utf-8
ptuiCB ( ' 0 ' , ' 0 ' , ' http: //t.qq.com ' , ' 1 ' , ' 登录成功! ' );
```

看来我们的猜想得到了证实，流程确实如此。接下来就看JavaScript文件是怎么处理的。

这些请求的数据是怎么来的呢？这个验证逻辑是怎么的呢？先看登录页面的处理。

2. 登录页面的处理

根据常识，腾讯在登录前会做许多登录的加密处理，QQ邮箱就是如此。前面提到，腾讯一定在登录前做了客户端处理，那我们肯定要从表单入手。在登录首页HTML代码中，看到了如下的处理代码：

```
onsubmit="if ( beforeSub () ) return false; if (! isAbleSubmit ) { return false }; return ptui_
checkValidate (); "
```

这就是 SUBMIT 的处理，顺藤摸瓜找到了 http://imgcache.qq.com/ptlogin/ac/v7/js/login_div.js?v=1.2.1 登录处理的 JavaScript 文件，下载此文件。

3.分析登录处理的JavaScript文件

这个JavaScript文件的逻辑比较复杂，通过分析找到下面的代码：

```
function ptui_needVC ( C, D ) {
if ( input_aid==D ) {
if ( ( C.indexOf ( "@" ) < 0 ) && isNaN ( C ) ) {
C="@"+C
}
}
var
B="http://ptlogin2."+g_domain+"/check?uin="+C+"&appid="+D+"&r="+Math.random ();
```

根据之前的推测，腾讯在我们登录微博时对客户端的密码进行了加密处理。所以在查看此JavaScript文件时，还要着重寻找和加密相关的代码。果不其然，我们找到了 `function md5_3 (B)` 这个函数，从函数名就可以知道，这是一个和加密处理相关的函数。

在上面的JavaScript文件里还找到了 `ptui_checkValidate` 函数，可以猜想，登录过程被此函数包装处理过了。我们怀疑在输入账号的时候，JavaScript脚本就悄悄地向

' , ' ! GZ1 ') 了，其第一个参数应该是1或者其他不等于0的参数，否则不论存在与否，都返回0，就没意义了。但是现在的事实是，不论输入的账号是否存在，返回的数据都是一样的。因此，`http://ptlogin2.qq.com/check?uin=@XXX`是验证账号是否存在的这个猜想被推翻了，那就只有另一种可能了，返回一个特殊的字符供前台使用。

提示 可能你会觉得这一步的猜想很不严谨，但是在实际开发中，从社会工程学的角度说，大胆假设是很有必要的。

截止这里，我们所做的都是猜想和假设。那怎么验证猜想呢？继续深入JavaScript代码。等一下，发现端倪了吗？

```
ptui_checkVC ( ' 0 ' , ' ! VKX ' );
```

再看第一个请求的HTTP原始数据，如图4-31所示。



图 4-31 抓取腾讯微博的数据包

第一个GET请求的参数`verifycode=!VKX`不就是在单击密码框时返回的数据吗？证实了前面的猜想，那一步返回的就是验证码！我们离成功不远了！

往前追溯，找到`ptui_needVC`函数（是不是和`ptui_checkVC`很像）。通过上面的分析和对源代码的阅读，大致得出流程如下：

- 1) 输入账号时利用JavaScript发出了一个请求，服务器端返回验证码。

- 2) 将这个验证码与密码一起进行加密，然后发送请求。上面那大串就是真正发送的数据。以下就是经过处理后的密码：

```
var f="";  
f+=e.verifycode.value;  
f=f.toUpperCase ();
```

服务器发起了请求，并返回了什么东西。

为了验证和查看这一步中腾讯究竟是怎么处理的，我们可以这么做：在首页输入我的账号，单击密码框（注意：只是单击密码框，不做输入和提交处理），这时看到了一条HTTP请求，地址如下：

```
http: //ptlogin2.qq.com/check? uin=@wait4you&appid=46000101&r=0.5118417510284644
```

然后看返回的数据：

```
ptui_checkVC ( ' 0 ' , ' ! VKX ' );
```

返回的是干什么？验证这个账号是否存在还是其他的事情？

这个疑问很好解答。我们输入账号tt55555555555555，单击输入框，返回的数据如下：

```
ptui_checkVC ( ' 0 ' , ' ! GZ1 ' );
```

两次输入返回的数据大同小异。按常理不可能有人使用tt55555555555555这样的数据作为账号，即难看又不好记。假如以下是验证wait4you账号是否存在的：

```
http: //ptlogin2.qq.com/check? uin=@wait4you&appid=46000101&r=0.5118417510284644
```

那么输入tt55555555555555这个不存在的账号时，也应该是验证这个账号是否存在的。这个账号按常理来说应该是不存在的，那么返回的就不应该是ptui_checkVC (' 0

```
b+=md5 ( md5_3 ( e.p.value ) +f)
```

也就是说，从登录的请求中找到verifycode参数，它就是验证码。从客户端将上面得到的b与verifycode都发送到服务器端，服务器端通过u值（账号）查出上面的密码B，然后在服务器端对B+verifycode进行md5求值，将结果与发送过来的D进行比较，一致则登录成功，反之失败。其他参数基本上都是固定值，除了获取验证码时传递的那个随机数r。

在发送了这个请求后，微博验证后会返回以下这样的数据，并且还将做一系列跳转：

```
ptuiCB ( ' 0 ', ' 0 ', ' http: //t.qq.com ', ' 1 ', ' 登录成功! ');
```

一切都恍然大悟了。其实，如果JavaScript功底足够的话，一看到ptui_checkVC (' 0 ', '! VKX ') 这样的返回值就知道，这其实是JavaScript里的动态执行JavaScript函数。也就是返回一段Script脚本，然后在客户端执行。通常返回的都是文本，所以对这种情况可能会不是很熟悉。

据此，大致可以写代码了，如代码清单4-15所示。

代码清单4-15 腾讯微博抓取

```
<? php
$t=360; //缓存时间，单位：秒
if (! is_file ( ' fetch.html ') || ( time () -filetime ( ' fetch.html ')) > $t) {
$qq= ' wait4you ';
$user=$_GET [ ' user ' ];
$pwd= ' 79F9BB25674886F8E1608F17F633B831 '; //以上参数不可修改
//调用方式：qq.php? user=XXX, XXX为你的微博用户名，注意不是你的QQ号
$verifyURL= ' http: //ptlogin2.qq.com/check? uin=@ ' . $qq. ' &appid=46000101 ' ;
$loginURL= ' http: //ptlogin2.qq.com/login? ' ;
//获取验证码及第一次cookie
$curl=curl_ init ( $verifyURL);
```

```
$cookie_jar=tempnam ( '. ', ' cookie ');
curl_setopt ( $curl, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
$verifyCode=curl_exec ( $curl );
curl_close ( $curl );
$verifyCode=strtoupper ( substr ( $verifyCode, 18, 4 ));
//发送登录请求并获取第二次cookie
$loginURL.= ' u=@ ' . $qq. ' & p= ' .md5 ( $ pwd. $ verifyCode ) . ' & verifycode= ' .
$verifyCode. ' &aid=46000101&u1=
http%3A%2F%2Ft.qq.com&h=1&from_ui=1&fp=loginerroralert ';
$curl=curl_init ( $loginURL );
curl_setopt ( $curl, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEFILE, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
$loginResult=curl_exec ( $curl ); curl_close ( $curl );
//echo ' 登录验证结果: ' . $loginResult;
//获取第三次cookie
$curl=curl_init ( ' http: //t.qq.com ' );
curl_setopt ( $curl, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEFILE, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
$loginResult=curl_exec ( $curl );
curl_close ( $curl );
//第四次
$curl=curl_init ( ' http: //t.qq.com/ ' . $user );
curl_setopt ( $curl, CURLOPT_RETURNTRANSFER, 1 );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEFILE, $cookie_jar );
curl_setopt ( $curl, CURLOPT_COOKIEJAR, $cookie_jar );
$loginResult=curl_exec ( $curl );
curl_close ( $curl );
unlink ( $cookie_jar );
file_put_contents ( ' fetch.html ', $loginResult );
}
@header ( ' Content-type: text/html; charset=utf-8 ' );
if ( is_file ( "fetch.html" )) {
$content=file_get_contents ( "fetch.html" );
$pattern="~<div class= \ "msgBox \ ">.*<div class= \ "pubInfo \ "> ~isU";
```

```
preg_match_all ($pattern,$content,$match);
$message1= $match[0][0]; //暂只抓取第一条信息
$message1=str_replace (' href="/', ' href="http: //t.qq.com/', $message1);
$message1=strip_tags ($message1, ' <a> ');
$m2=addslashes ($message1);
echo ' document.writeln (" ' . $m2. ' " ); ' ;
} else {
die ( ' 参数错误或者服务器不支持cURL ' );
}
? >
```

其中，后三次跳转如图4-32所示。



图 4-32 抓包数据

上面的PHP代码中，pwd参数由md5_3 ()得到，将微博页面的JavaScript下载到本地进行加密，或者自己用语言重新实现一次（基本上改动很少，这样就可以做成彻底的API了）。

由于登录逻辑比较复杂，主要由JavaScript处理客户端和服务端请求和返回数据，可能描述的还不够详细，不过你可以在本地多多练习，熟能生巧。

借助各种调试工具，能够在很大程度上帮助我们处理这类问题。这里最难的分析JavaScript文件，理清其处理逻辑。这一步不仅要求我们有丰富的HTTP协议和抓包分析的经验，还要求扎实的JavaScript基础。

现在要完成POST或者其他操作都随心所欲了。目前，网上有抓取QQ邮箱通信录的代码，思路是一样的。但是这样的代码早就失效了，如果想实现其功能，就必须自己分析登录逻辑，模拟登录。如果掌握了本节的知识技巧和技巧，就能轻松地实现任何功能。

提示 由于腾讯微博不断调整算法，此算法不保证一直可用，但是基本原理不变。另外，腾讯现在也开放了平台，可以直接调用微博API。

对于必须输入验证码的应用，能否模拟登录？答案仍然是肯定的，但是需要请求验证

码图片，返回，让使用者手工输入验证码，然后再发送请求。全自动地模拟登录还无法做到，因为想破解验证码难度是很大的。

思考 腾讯微博的做法是不是最大限度、最低成本地保障了客户资料的安全呢？是不是可以有效地避免某一类攻击呢？我们能从其设计中借鉴到什么呢？

4.5 简单邮件传输协议SMTP

很多应用需要发送邮件的功能。PHP有一个自带的mail()函数，很多新手会问：“为什么使用这个函数发不了邮件？”这是因为要想使用SMTP协议发送邮件，必须首先安装SMTP服务器。如果既不想安装SMTP邮件服务器，也没有条件安装，怎么办呢？这时，Socket就派上用场了。使用Socket连接一个已有的服务器，如163提供的SMTP服务器，然后用它发送邮件。应该怎么用呢？下面先来看SMTP协议是如何工作的。

4.5.1 SMTP协议如何工作

SMTP (Simple-mail Transfer Protocol, 简单邮件传输协议) 是由源地址到目的地址传送邮件的一组规则，用来控制信件的中转方式。SMTP协议属于TCP/IP协议族，其使每台计算机在发送或中转信件时能找到下一个目的地。通过使用指定的服务器，把E-mail寄到收信人的服务器上。

SMTP服务器是遵循SMTP协议的发送邮件服务器，用来发送或中转发出的电子邮件。客户端通过SMTP命令与SMTP服务器进行交互。首先，客户端需要建立一个与SMTP服务器的TCP连接，端口通常为25。在连接建立之后，客户端和服务器先执行一些应用层握手操作，让SMTP服务器知道客户端的信息，并且对客户端需求做出响应等。

在SMTP握手阶段，客户端向SMTP服务器分别指定发信人和收信人的电子邮件地址。握手阶段完毕，SMTP服务器把客户端发出的邮件消息添加到发信队列中，通过TCP提供的可靠数据传输服务把该消息无错地传送到服务器。如果客户还有其他邮件消息需发送到同一个服务器，就在同一个TCP连接重复上述过程；否则，指示TCP关闭该连接。

连接和发送过程如下：

1) 建立TCP连接。

2) 客户端发送HELO命令以标识发件人自己的身份，客户端发送MAIL命令。服务器以OK作为响应，表明准备接收。

3) 使用AUTH命令登录SMTP服务器，输入用户名和密码（注意，用户名和密码都需要使用base64加密）。

4) 客户端发送RCPT命令，标识该电子邮件的计划接收人，可以有多个RCPT行。服务器以OK作为响应，表示愿意为收件人发送邮件。

5) 协商结束后，使用DATA命令发送。

6) 以“.”号表示结束，输入内容一起发送出去，结束此次发送，用QUIT命令退出。

例如，使用Telnet创建一个SMTP会话，其中S代表服务器，C代表客户端，代码如下所示：

```
C: open smtp.qq.com 25
S: 220 esmtp4.qq.com Esmtp QQ Mail Server
C: HELO smtp.qq.com
S: 250 esmtp4.qq.com
C: AUTH login
S: 334 VXNlcm5hbWU6
C: 这里输入使用base64加密过的用户名
S: 334 UGFzc3dvcmQ6
C: 这里输入使用base64加密过的密码
S: 235 Authentication successful
C: MAIL FROM: <liexusong@qq.com>
S: 250 sender <liexusong@qq.com> OK
C: RCPT TO: <liexusong@163.com>
S: 250 recipient <liexusong@163.com> OK
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: This is example for smtp protocol
C: .
S: 250 message sent
C: QUIT
S: 221 goodbye
```

上述命令并不一定会一次性成功，服务器可能会返回错误响应，客户端应该按照协议规定的顺序输入后续的命令（或重复执行失败的命令、或重置会话、或退出会话等）。

4.5.2 SMTP协议常用命令

SMTP命令不区分大小写，但参数区分大小写。常用命令如表4-5所示。

命令	描述
DATA	开始信息写作
EXPN < string >	验证给定的邮件列表是否存在。扩充邮件列表，常被禁用
HELO < domain >	向服务器标识用户身份，返回邮件服务器身份
HELP < command >	查询服务器支持什么命令，返回命令中的信息
MAIL FROM: < host >	在主机上初始化一个邮件会话
NOOP	无操作，服务器响应 OK
QUIT	终止邮件会话
RCPT TO: < user >	标识单个的邮件接收人；MAIL 命令后面可有多个 rcpt to;
RSET	重置会话，当前传输被取消
SAML FROM: < host >	发送邮件到用户终端和邮箱
SEND FROM: < host >	发送邮件到用户终端
SOML FROM: < host >	发送邮件到用户终端或邮箱
TURN	接收端和发送端交换角色
VERFY < user >	用于验证指定的用户/邮箱是否存在；由于安全方面的原因，服务器常常禁止此命令

4.5.3 SMTP协议应用：使用Socket发送邮件

SMTP协议建立在TCP协议之上，所以原则上按照SMTP协议的规范，使用Socket跟SMTP服务器进行交互。下面使用PHP的Socket实现发送邮件的功能。本例中使用fsockopen()函数代替socket_*()类函数。

fsockopen()函数的好处是把Socket连接绑定到一个流上，然后使用各种操作流的函数操作这个Socket连接。下面看看fsockopen()函数的用法：

```
resource fsockopen ( string $ hostname, int $ port, int[ $ errno] , string[ $ errstr] ,
int[ $ timeout]);
```

参数说明如下：

hostname：要连接的服务器路径。

port：要绑定的端口。

errno：保存连接发生错误时的错误代号。

errstr：保存错误信息。

timeout：设置连接的超时时间，单位为秒。

使用fsockopen()函数创建的Socket连接句柄可以提供给诸如fgets()、fputs()、fwrite()、fread()、fclose()等流函数使用。

下面编写一个发送邮件的类，主要使用fsockopen()函数，按照SMTP协议实现。我们把这个类命名为smtp_mail。代码如下：

```
1.private $ host;
2.private $ port=25;
3.private $ user;
```

```
4.private $pass;
5.private $debug=false;
6.private $sock;
7.private $mail_format=0;
```

第1~7行定义了smtp_mail类需要用到的成员变量，具体包括：

host：保存要连接的SMTP服务器。

port：要绑定的端口，默认为25。

user和pass：要登录SMTP服务器的用户名和密码。

debug：标识是否开启调试模式，默认为关闭。

sock：保存与SMTP服务器连接的句柄。

mail_format：标志使用什么格式发送邮件，0为普通文本，1为HTML邮件。

```
8.function smtp_mail ($host,$port,$user,$pass,
9.$format=1,$debug=0)
9. {
10. $this->host=$host;
11. $this->port=$port;
12. $this->user=base64_encode ($user);
13. $this->pass=base64_encode ($pass);
14. $this->mail_format=$format;
15. $this->debug=$debug;
16. $this->sock=fsockopen ($this->host,$this->port,
17.&$errno, &$errstr, 10);
18.if (! $this->sock) {
19.exit ("Error number: $errno, Error message: $errstr \n");
20. }
21.$response=fgets ($this->sock);
22.if ( strstr ($response, "220") ===false) {
23.exit ("server error: $response \n");
24. }
```

```
24. }
```

第8~15行是smtp_mail类的构造函数，主要设置了smtp_mail类的成员变量。要注意的是，user和pass都需要使用base64加密，否则可能会登录失败。

第16~24行中的第16行使用fsockopen()函数连接SMTP服务器。如果连接失败，会显示错误信息，并且把程序挂起。第20行使用fgets()函数取得服务器的信息。如果服务器返回的信息中包含220，代表已经成功连接到服务器，否则表示连接错误。

```
25.private function show_debug ($message)
26. {
27.if ($this->debug) {
28.echo"<p>Debug: $message \n";
29.}
30.}
```

第25~30行中的show_debug()根据用户是否开启了调试模式显示调试信息，如果用户关闭了调试模式，将不会显示调试信息。调试模式在构造函数中设置，默认为关闭。

```
31.private function do_command ($cmd,$return_code)
32. {
33.fwrite ($this->sock,$cmd);
34.$response=fgets ($this->sock);
35.if ( strstr ($response, "$return_code") ===false) {
36.$this->show_debug ($response);
37.return false;
38.}
39.return true;
40.}
```

第31~40行中的do_command()方法的功能是把命令发送到服务器中执行，然后取得服务器的反馈消息，并且从服务器的反馈消息中判断命令是否执行成功。cmd参数就是要发送到服务器执行的命令，如果服务器的反馈消息中存在return_code，说明命令已经成功执行，否则就说明命令执行失败。

```
41.private function is_email ($email)
42. {
43. $pattren="/^[^_][\w]*@[ \w.]+[ \w]*[^\_]$/" ;
44. if ( preg_match ($pattren,$email,$matches) ) {
45. return true;
46. } else {
47. return false;
48. }
49. }
```

第41~49行中的is_email()方法用于验证邮箱地址是否合法。如果用户输入的邮箱是错误的，将拒绝发送邮件。这样可以把一些格式错误的邮箱剔除，减小出错的几率。

```
50.public function send_mail ($from,$to,$subject,$body)
51. {
52. if (! $this->is_email ($from) OR! $this->is_email ($to)) {
53. $this->show_debug ("Please enter valid from/to email.");
54. return false;
55. }
56. if ( empty ($subject) OR empty ($body)) {
57. $this->show_debug ("Please enter subject/content.");
58. return false;
59. }
60. $detail="From: ".$from."\r\n";
61. $detail.="To: ".$to."\r\n";
62. $detail.="Subject: ".$subject."\r\n";
63. if ($this->mail_format==1) {
64. $detail.="Content-Type: text/html; \r\n";
65. } else {
```

```
66.$detail="Content-Type: text/plain; \r\n";
67.}
68.$detail="charset=gb2312\r\n\r\n";
69.$detail=$body;
70.$this->do_command("HELO smtp.qq.com\r\n", 250);
71.$this->do_command("AUTH LOGIN\r\n", 334);
72.$this->do_command($this->user."\r\n", 334);
73.$this->do_command($this->pass."\r\n", 235);
74.$this->do_command("MAIL FROM: ".$from."\r\n", 250);
75.$this->do_command("RCPT TO: ".$to."\r\n", 250);
76.$this->do_command("DATA\r\n", 354);
77.$this->do_command($detail."\r\n.\r\n", 250);
78.$this->do_command("QUIT\r\n", 221);
79.return true;
80.}
```

第50~59行，from是发信人的邮箱，to是收信人的有效地址，subject是邮件的主题，body是邮件的内容。先使用is_email()方法验证发信人和收信人的邮箱是否有效，再验证邮件的主题和内容是否为空。只有这些信息都正确的情况下，才会发送邮件。

第60~69行，根据用户提供的信息生成邮件的主体内容。detail是要发送的主要内容，包括发信人、收信人、邮件主题、邮件内容等，另外还包括邮件的格式（HTML格式或者文本格式）和邮件的编码。

第70~80行，根据SMTP协议向服务器发送命令。例如70行发送HELO命令、71行发送AUTH命令等。特别注意，发送邮件主体的时候要以"\r\n.\r\n"作为结尾，因为SMTP服务器会以此作为邮件主体的结束标记，如果没有以此标记作为结尾，SMTP服务器会一直等待，直到超时为止。

SMTP测试代码如代码清单4-16所示。

代码清单4-16 SMTP测试代码

```
<? php
include ("smtp.class.php");
```

```
$host="smtp.qq.com";
$port=25;
$user="username";
$pass="password";
$from="send@qq.com";
$to="recv@qq.com";
$subject="Hello Body";
$content="This is example email for you";
$mail=new smtp_mail ($host,$port,$user,$pass);
$mail->send_mail ($from,$to,$subject,$content);
? >
```

测试结果如图4-33所示。

可以看出，原本单纯依靠PHP无法完成的任务，使用Socket轻而易举地实现了。假如现在有人问：PHP与C语言怎么交互？你可能会想到用C语言写PHP扩展，或者使用exec、system等命令，但是为什么不换个思路呢？用C语言提供服务，用PHP请求这个服务，PHP就能借助C语言完成许多看似不可能的任务。现在流行的WebGame正是利用C语言、Java等重量级语言在底层完成复杂的运算，通过Socket把接口提供给PHP使用。



图 4-33 使用Socket发送邮件测试结果

4.6 Webservice的前世今生

1998年，一个名叫Dave Winer的程序员设计出XMLRPC（XML Remote Procedure Call, XML远程方法调用），由于XML（eXtensible Markup Language, 可扩展置标语言）这个概念早期被炒作得很火，所以其名字沾了XML的光。RPC即远程服务调用的意思，其实这就是SOAP（Simple Object Access Protocol, 简单对象访问协议）。

之后SOAP迅速被捧红，各大巨头如IBM、微软都分别推出了自己的设计，市场一片混乱。很快大家就都认识到，必须有一个标准，于是这些巨头提议标准化，此时SOAP成型。

4.6.1 Webservice简介

Webservice的概念在1998年前后形成。微软使用SOAP推出了成熟的Webservice产品，此时Webservice传的是对象。

由于厂商之间博弈，SOAP被不断设计，最终形成现在的模样：直接传XML的Webservice。由于传对象无法实现互通，SOAP逐渐改为传字符串。

广义Webservice可以实现硬件和硬件、硬件和软件、软件和软件之间的通信。使不同的系统之间能够用“软件-软件对话”的方式相互调用，打破了软件应用、网站和各种设备之间的格格不入的状态，实现“基于Web无缝集成”的目标。

传统Webservice在Web领域已经渐渐被人抛弃，除了一些和硬件打交道的地方还在使用，已经很难见到这种极其笨重、低效、复杂的服务了。越来越多简单的Webservice产品被设计出来，这些产品凸显出简单高效的特性。简单、高效、易于理解才是我们需要的。

现在有很多开放的API基于简单XML。只需要读取服务器端生成的一个文件，得到一串XML，就可以用任何语言解析这个XML字符串。这种方式利用XML跨平台的通用性，使用简单的文本和HTTP请求实现了Webservice，这是一种对Web本质的回归——简

单、实用至上。

团购网从2009年一直火到现在，在网络上遍地开花，随之而来的是一种新的导航——团购导航站。团购导航其实就是把许多团购网信息收集起来展示，无非就是使用两种技术：采集和API。许多团购网站公开API供其他网站调用，例如拉手团购API（<http://www.lashou.com/openhtml.php>）。

要实现一个类似的需求很简单，即读取API，获取XML解析。用PHP解析XML很方便，一个simplexml_load_file函数就能搞定。只不过simplexml获取的是一个对象，更多人更习惯操作数组，那好办，把对象转为数组就OK了，如代码清单4-17所示。

代码清单4-17 把XML对象转换为数组

```
function simplexml_obj2array ($obj) {
    if ( count ($obj) >=1)
    {
        $result=$keys=array ();
        foreach ($obj as $key=> $value )
        {
            isset ($keys[$key])? ($keys[$key]+=1): ($keys[$key]=1);
            if ($keys[$key]==1)
            {
                $result[$key]=simplexml_obj2array ($value );
            }
            elseif ($keys[$key]==2)
            {
                $result[$key]=array ($result[$key], simplexml_obj2array ($value ));
            }
            else if ($keys[$key]>2)
            {
                $result[$key][]=simplexml_obj2array ($value );
            }
        }
        return $result;
    }
    else if ( count ($obj) ==0)
```

```

{
return ( string )$obj;
}}
echo ' <pre> ';
print_r ( simplexml_obj2array ( $xml ));
echo ' </pre> ';

```

程序运行结果如图4-34所示。



```

[0] => Array
(
    [url] => http://www.lashou.com/deal/feiJing/1200.html
    [xml] => Array
        (
            [docType] => Array
                (
                    [doctype] => 拾手网
                    [uri] => http://www.lashou.com/feiJing
                )
            [root] => 淘票
            [title] => 迅雷3000安全版包邮! 原价1100元的俄罗斯黑印纯黑
            [image] => Array ( [1] => 161; [2] => 200; [3] => webm; [4] => 90/90/90
            [price] => 1200.00000
            [modified] => 1202047200
            [url_md5] => 1550_00
            [url_id] => 3000_00
            [version] => 2.3
            [charset] => GB
            [meta1] => 迅雷3000安全版, 怎样保持你的迅雷速度, 享受快
        )
    [obj] => Array
        (
            [name] => 京东商城股份有限公司
            [url] => http://www.360buy.com
            [abbr] =>
        )
    )
)

```

图 4-34 程序运行结果

现在需要做就是HTML润色工作。这样，一个简单的API调用的核心部分就完成了。很简单吧，无非就是拉取XML解析，没有任何新名词、新概念。

API的另一个运用就是通过调用传统WebService实现服务。传统WebService以WSDL文件A作为参考传递XML。前面已经说过，这种方法比较笨重，也越来越不受欢迎。所以这里介绍一个轻量级并且高效的WebService解决方案—PHPRPC。

4.6.2 认识PHPRPC协议

PHPRPC是一个轻型的、安全的、跨网际的、跨语言的、跨平台的、跨环境的、跨域的协议，支持复杂对象传输、引用参数传递、内容输出重定向、分级错误处理、会话，是面向服务的高性能远程过程调用协议。

PHPRPC为什么快速，主要原因就在于其数据序列化和传输，这也是其优于传统WebService的原因。

PHPRPC支持十多种常见的编程语言，包括ASP、PHP、Java、C++、Delphi、JavaScript、ActionScript、Python、Ruby、Perl等。可以到官方网站了解更详细的应用：http://phprpc.org/zh_CN/。本节通过一个简单的应用演示此工具。假设有一个需求：由于某种原因，PHP需要调用Java或者Delphi的方法（因为PHP无法对硬件和系统资源进行操作，比如用纯PHP管理考勤机、打印机等，很难实现，所以说这种场景还是很正常的），就可以用Java或者Delphi发布一个方法，用简单的PHP调用。

提示 PHPRPC的功能不仅如此，现在已经演变为商业版本Hprose，其功能更强大。

如何用PHP调用Java类，通过php java bridge可以实现，还有另外一种方法就是使用Web Service，也可以使用ICE等中间件，这里通过PHPRPC实现。在学习这个例子前，先到PHPRPC官网下载对应的软件包，解压。其并不需要特殊的配置，使用非常简单。

首先在服务器端用Java写两个类，作为服务的发布：一个为MyHello类，简单地调用一个打印方法；另外一个类，使用Java从MySQL数据库中查询图书馆馆藏。提供打印的服务如下所示：

```
package cn.sin.book.dao;
public class MyHello {
public String say ( String name ) {
return"Hello"+name;
}
}
```

```
}
```

另外一个服务是提供图书查询的，如代码清单4-18所示。

代码清单4-18 图书服务实现的代码

```
package cn.sin.book.dao;
/**
 * @description: 简单图书查询类
 * @author waitfox@qq.com
 */
import java.sql.Connection;
import java.sql.SQLException;
import java.util.HashSet;
import java.util.Set;
import cn.sin.tool.ConnectionTool;
import cn.sin.book.domain.Book;
public class FindBooksDao {
public double allcount () { //查询馆藏图书数量
double pall=0;
ConnectionTool cfb=new ConnectionTool (); //连接数据库
Connection con=null;
try {
con=cfb.getConnection ();
cfb.setStmt ( con.createStatement ());

cfb.setRs ( cfb.getStmt () .executeQuery (
"select count ( bid ) as total from book" ));
if ( cfb.getRs () .next () ) {
pall=cfb.getRs () .getInt ( 1);
}
} catch ( SQLException e ) {
e.printStackTrace ();
}
try {
con.rollback ();
} catch ( SQLException e1 ) {
e1.printStackTrace ();
}
```

```
    }  
    } finally {  
    cfb.close ();  
    }  
    return pall;  
    }  
    public Set<Book> findbooks ( int begin, int offset ) { //查询图书详细列表  
    Set<Book> findbookSet=new HashSet<Book> ();  
    /*建立连接，此处代码与上一方法中相似部分省略*/  
    cfb.setPstmt ( con.prepareStatement ( "SELECT book.bid, book.bname, book.isbn, book.au-  
thor, book.press,  
    book.price, sum ( blink.statu ) AS kejie, count ( blink.bid ) AS zongshu FROM blink, book  
WHERE book.bid=blink.  
    bid GROUP BY book.bid LIMIT?,? " ));  
    /*装填模型，此处代码与上一方法中相似部分省略*/  
    findbookSet.add ( book );  
    }  
    } catch ( SQLException e ) {}  
    /*异常处理，此处代码与上一方法中相似部分省略*/  
    return findbookSet;  
    }  
    public String findbooks2 ( int begin, int offset ) { //如果类型未序列化，可使用此方法  
    FindBooksDao fBooksDao=new FindBooksDao ();  
    return fBooksDao.findbooks ( begin, offset ).toString ();  
    }  
    }
```

以上只给出了核心部分，省略了部分冗余代码，因为Java环境的部署和开发都不是关注的重点。

代码清单4-19所示是我们使用的模型，这和PHP中的对象是一个概念。

代码清单4-19 图书对象

```
package cn.sin.book.domain;  
import org.apache.commons.lang3.builder.ReflectionToStringBuilder;  
import org.apache.commons.lang3.builder.ToStringStyle;
```

```

import java.io.Serializable;
public class Book implements Serializable {

private int bid;
private String bname;
private String isbn;
private String author;
private String press;
private double price;
private int zongshu;
private int kejie;
/*getter, setter方法略*/
@Override
public String toString () {
return ReflectionToStringBuilder.toString ( this, ToStringStyle.MULTI_LINE_STYLE,

true, true );

}}

```

以上代码在Java的运行情况如图4-35所示。



图 4-35 Java程序运行结果

现在发布这个服务，分别让Java和PHP调用它。首先，建立rpc.jsp文件发布此服务，如代码清单4-20所示。

代码清单4-20 发布服务rpc.jsp文件

```

<%@page import="cn.sin.book.dao.MyHello"%>
<%@page import="cn.sin.book.dao.FindBooksDao"%>
<%@page import="org.phprpc.*"%>
<%
MyHello hello=new MyHello ();

```

```
FindBooksDao book=new FindBooksDao ();
PHPRPC_Server phprpc_server=new PHPRPC_Server ();
phprpc_server.add ( hello, MyHello.class );
phprpc_server.add ( book, FindBooksDao.class );
phprpc_server.start ( request, response );
%>
```

用Java客户端调用此服务，如代码清单4-21所示。

代码清单4-21 用Java客户端调用服务

```
package cn.sin.book.dao;
import java.util.Set;
import org.phprpc.*;
import cn.sin.book.domain.Book;
interface ihello {
//定义此服务所拥有的所有方法，我们看到即使不在一个类里定义的方法，也可以共同发布
public String say ( String name );
public double allcount ();
Set<Book> findbooks ( int begin, int offset );
public String findbooks2 ( int begin, int offset );
}
public class PhpRpcDemo {
public static void main ( String[]args ) {
PHPRPC_Client client=new PHPRPC_Client ( "http: //library.aiyooyo.com: 8080/action/-
book/
rpc.jsp");
ihello m= ( ihello ) client.useService ( ihello.class );
System.out.println ( m.say ( "白菜" ));
System.out.println ( m.allcount ( ));
System.out.println ( m.findbooks2 ( 0, 5 ));
System.out.println ( m.findbooks ( 0, 5 ));
}
}
```

运行情况如图4-36所示。



图 4-36 PHPRPC Java运行结果

轮到PHP出场了，用PHP作为客户端调用Java发布的方法，如代码清单4-22所示。

代码清单4-22 用PHP调用Java服务

```

<? php
include ( "phprpc/phprpc_client.php" );
$client=new PHPRPC_Client ( ' http://library.aiyooyo.com: 8080/action/book/rpc.jsp ' );
echo $client-> say ( "白菜" );
echo $client-> allcount ( );
echo $client-> findbooks ( 0, 5 );
? >
  
```

运行结果如图4-37所示。

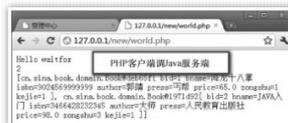


图 4-37 PHPRPC PHP运行结果

可以看出，这种方式无论是服务器端的发布还是客户端的调用都极其简单，没有烦琐的服务器端配置，也没有一堆难以理解的XML文件，速度也有很大优势。

如果用PHP客户端调用PHP服务器端呢？这更简单了，首先安装PHPRPC，需要以

下四个步骤。

步骤1 下载phprpc for php的安装包，解压。

其中，bigint.php、compat.php、phprpc_date.php、xxtea.php属于公共文件。不论是客户端还是服务器端都需要这些文件。

phprpc_client.php是客户端文件，如果只需要使用客户端，那么只要有上面那些公共文件和这个文件就可以使用了，使用时直接在程序中包含phprpc_client.php即可，公共文件不需要单独包含，phprpc_client.php会自动包含进去那些公共文件。

dhparams、dhparams.php、phprpc_server.php这三个文件是服务器端需要的文件。其中dhparams目录中包含的是加密传输时用生成密钥的参数，如果需要更详细的解释，请参见PHPRPC官方对加密传输的介绍，这里使用默认配置。

dhparams.php用来读取dhparams目录中文件的类。

phprpc_server.php是服务器端文件，如果使用PHP发布PHPRPC服务，只需要包含这个文件就可以了。公共文件和dhparams.php都不需要单独包含。

步骤2 把服务器端和客户端部署到同一台服务器上。

当然，实际情况中服务器端和客户端往往不在同一台服务器，或者不在同一个域名下。

步骤3 写服务器端代码并发布，如代码清单4-23所示。

代码清单4-23 PHP的RPC服务器端

```
<? php
include ("phprpc/phprpc_server.php");
class Hello {
static function HelloWorld () {
return ' Hello World! ';
}}
$server=new PHPRPC_Server ();
```

```
$server->add ( ' HelloWorld ', ' hello ');  
$server->start ();  
? >
```

步骤4 对客户端进行调用，如代码清单4-24所示。

代码清单4-24 PHP的RPC客户端

```
<? php  
include ( "phprpc/phprpc_client.php" );  
$client=new PHPRPC_Client ( ' http: //127.1/php/rtest.php ' );  
echo $client->HelloWorld ();  
? >
```

可见，用PHP写服务器端和客户端的代码都比较简洁，这正是PHP的优势。

4.6.3 Web服务的实现模式

PHP支持两种Web服务模式：WSDL和non WSDL，为了便于理解，先介绍Web服务的两种实现模式。Web服务主要有两种实现模式：

契约先行（Contract First）模式。

代码先行（Code First）模式。

契约先行模式的实现中，首要工作是定义针对这个Web服务接口的WSDL（Web Services Description Language，网页服务描述语言）文件。WSDL文件中描述了Web服务的位置、可提供的操作集，以及其他属性。WSDL文件也就是Web服务的“契约”。“契约”订立之后，再据此进行服务器端和客户端的应用程序开发。这种模式对应WSDL模式。

WSDL文件包括5部分：Types、Message、PortType、Binding和Service。

Types：类型定义独立于语言，对应于SOAP消息中要传输的元素信息的定义。

Message：每个Web方法对应两个Message定义in和out，而Message的定义包含头和体。

PortType：每个WebService对应一个PortType，其中又包含了对其发布的方法、操作。

Bindings：指定PortType中每个操作（类以及方法）的绑定信息，包含input和output消息的格式。

Service：每个WebService绑定的port信息。

WSDL文件的详细介绍可以查看：<http://www.w3school.com.cn/wsdl/>。

与契约先行模式不同，代码先行模式中，第一步工作是实现Web服务器端，根据服务器端的实现，用某种方法（自动生成或手工编写）生成WSDL文件。但是由于PHP本身并没有提供从Web服务实现代码中生成WSDL文件的方法，因此就要以non WSDL模

式连接服务器端，即不通过WSDL文件创建SoapServer和SoapClient示例，而是直接向构造函数传递必要的参数。当然，代码先行模式也有其他的解决方法，一些集成的PHP开发工具（如Zend Studio）提供了根据Web服务实现代码生成WSDL文件的功能。

有的情况下，对方约定必须使用SOAP通信，所以下面先来了解SOAP是什么。

4.6.4 简单对象访问协议SOAP

在PHP 5之前，使用第三方产品NuSOAP辅助开发WebService，不过这个产品已经停止开发了。PHP 5新增了内置的SOAP扩展（ext/soap），从此开发基于SOAP的应用程序不需要下载额外的扩展库或代码包了。

SOAP（Simple Object Access Protocol，简单对象访问协议）是基于XML的、可扩展的通信协议。SOAP提供了一种标准，使运行在不同平台，并使用不同编程语言编写的应用程序可以互相进行通信。SOAP具有可扩展性和平台无关性的特点，广泛用作WebService的通信协议。SOAP中有以下几个主要类。

1.SOAP中的主要类

（1）SOAP服务类SoapServer

SoapServer类用来开发Web服务器端应用程序。这个类中包含创建、设置和操纵Web服务的函数。向Web服务中添加操作（Operation）有两种方式：

直接添加已定义的函数；

添加已定义好的类，从而将该类的公有成员函数添加到Web服务中。

（2）SOAP客户端类SoapClient

SoapClient类用于开发Web服务的客户端程序。内置的可用的成员函数主要有创建客户端实例、调用可用操作、查询可用操作和数据类型等。除此之外还包括了可用于程序调试的函数、获取上次请求和应答的SOAP数据。

（3）SOAP参数类SoapHeader、SoapParam、SoapVar

SoapHeader类用来构造SOAP头，SOAP头可以对SOAP的能力进行必要的扩展，主要用于简单的身份认证。

SoapParam和SoapVar封装了放入SOAP请求中的数据，主要在non WSDL模式下使用。

提示 在WSDL模式下，SOAP请求的参数通过数组方式包装，SOAP扩展根据WSDL文件将这个数组转化成为SOAP请求中的数据部分，所以并不需要SoapParam和SoapVar这两个类。而非WSDL模式没有提供WSDL文件，所以必须通过这两个类进行包装。

(4) SOAP异常类SoapFault

SoapFault类继承自PHP的Exception类，用来实现SOAP异常处理机制，由SOAP服务器端抛出。SOAP客户端接收该类的实例，用于获取有用的调试信息。

2.SOAP演示

前面介绍了SOAP中的一些基本概念，现在看一个比较简单的演示。首先是服务器端功能实现，如代码清单4-25所示。

代码清单4-25 server.php

```
<? php
function GetTime () {
return date ( ' Y-m-d ', time ());
}
class member {
public function add ($x,$y) {
return $x+ $y;
}
}
```

接下来发布服务器端所提供的函数和类，如代码清单4-26所示。

代码清单4-26 proxy.php

```
<? php
include _once ( ' server.php ');
```

```
$ soap=new SoapServer ( null, array ( ' uri ' =>"http: //test-uri"));
$ soap->addFunction ( ' GetTime ');
$ soap->setClass ( ' member ');
$ soap->handle ();
```

客户端代码如代码清单4-27所示。

代码清单4-27 client.php

```
$ client=new SoapClient ( null, array ( ' location ' =>"http: //localhost/t/phptest/proxy.php", '
uri ' =>"http: //test-
uri",
"style"=>SOAP_RPC, "use"=>SOAP_ENCODED, "trace"=>1, "exceptions"=>0));
// $rt= $ client->GetTime ();
$ address= $ client->add ( 1, 6);
echo"获取是: ", $ address, "\r\n";
//echo"获取到时间是: ", $rt, "\r\n";
```

运行以上代码看结果。如果运行时报错，如 looks like we got no XML document，需要检查代码中是否存在空格和换行，建议把PHP末尾的“? >”标记符关闭。

注意 client.php中注释掉的那两行，不要在服务器端同时混合发布类和函数，也不要一个服务接口发布多个类。

由于PHP是弱类型语言，而SOAP协议中对类型的定义比较严格，所以PHP无法仅仅根据代码生成可供使用的WSDL文件，只能通过PHP DOC之类的机制在注释中声明，从而使辅助工具获得参数的类型。在此模式下，需要用类的方式发布服务，并且类中有对参数类型和返回值类型的定义。

注意 由于PHP没有其他语言里的某些类型（如date型），在参数构造方面很容易出问题，疑惑也最多，这个时候一般使用stdClass类构造任何服务器端需要的对象，另外一些情况，可以直接传递字符串。

尽管SOAP的名字叫做简单对象访问协议，实际上并不简单。这里不对SOAP进行深入研究，也不推荐使用SOAP。除了这两种之外，JSON、HTTP、Socket等在某种意义上都属于Web Service范畴。

4.6.5 调试工具soapUI

本节最后介绍WebService和SOAP的调试工具——soapUI。这款桌面软件由Java写成，可以方便地构造WebService请求，实现WebService的功能、负载、符合性测试。界面如图4-38所示。

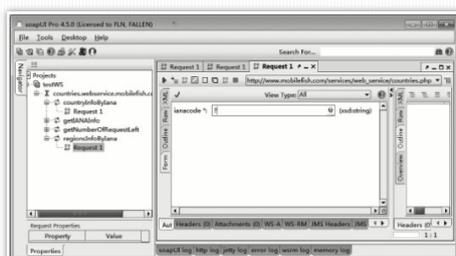


图 4-38 soapUI界面

可以到<http://www.soapui.org>网站下载soapUI，这里同时提供免费版和专业版下载。

看一个实际的例子，这个例子测试一个根据地区代码获取地区名和所在经纬度的应用。

WSDL 地址为 `http://www.mobilefish.com/services/web_service/countries.php?wsdl`。在浏览器中直接访问此地址，可以看出其可读性很差，很难一眼看出提供的方法，以及方法参数和返回值等信息。使用soapUI则让一切变得更简单。步骤如下。

步骤1 新建一个测试工程。

单击“file”→“New soapUI Project”菜单命令，或使用快捷键“Ctrl+N”新建一个测试工程，在弹出的对话框中依次输入工程名和WSDL文件地址，单击“OK”按钮完成，如图4-39所示。

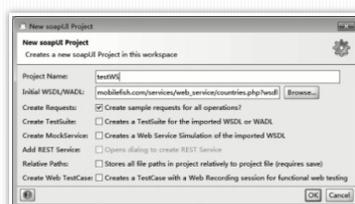


图 4-39 新建一个soapUI测试工程

创建完毕后，左侧导航栏中显示SOAP的方法列表。双击工程名，可以到管理界面中查看这个工程的详细信息。

步骤2 打开请求编辑器窗口。

展开导航栏中的“regionsInfoByIana”项，双击“Request 1”。右边的请求编辑器窗口将以Form、Raw、XML几种格式展示这个方法请求参数等关键信息，如图4-40所示。



图 4-40 请求编辑器窗口

请求编辑器分为三部分：

顶部的工具栏。包含一组与请求相关的动作、操作。

左边是请求区域。

右边是响应区域。

soapUI默认生成的请求中，“？”表示需要被替换的内容。根据需要，可以替换或者删除这些值。单击工具栏最左边的三角形按钮发送一个请求。这里在输入框中输入参数“SE”，单击按钮提交请求。

步骤3 查看测试结果，如图4-41所示。

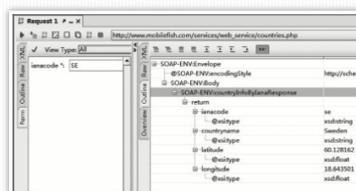


图 4-41 soapUI测试结果

由图4-41可以看出，输入参数为字符串类型的“SE”，返回3个结果，分别是字符串型的countyname和浮点型的latitude、longitude。测试结果符合预期。

当然，soapUI的功能远不止这些，如果希望了解更多使用技巧，可以到官方主页查看相关帮助文档。

4.7 Cookie详解

Cookie与Session是Web开发中必不可少的两个概念，不少人简单地把它们理解为一个是客户的存储机制，一个是服务器端的存储机制，而没有理解其中的原理和特性。实际上，Cookie和Session有着剪不断理还乱的关系，并没有想象中的那么简单。

随着Web应用的普及，SSO单点登录的推广，有必要深刻认识Cookie和Session的一些基础知识。理解了这些基础概念，可以使我们在开发中用到它们的时候得心应手。

本节首先结合HTTP协议和存储机制，详细讲解Cookie的基本概念和Web开发中的应用。

4.7.1 Cookie的基本概念及设置

Cookie在远程浏览器端存储数据并以此跟踪和识别用户的机制。从实现上说，Cookie是存储在客户端上的一小段数据，浏览器（即客户端）通过HTTP协议和服务器端进行Cookie交互。

注意 这里说的是客户端而不是浏览器，实际上能管理Cookie的不仅仅是浏览器，当然最常见的是由浏览器管理Cookie，后面的叙述中不再区分这两个概念。

Cookie独立于语言存在，也就是说，不论是PHP还是JSP种下的Cookie，其本质都是一样的，客户端脚本（如JavaScript）均能读取到。Cookie在很多语言里都有实现，比如PHP、ASP、Java、.NET。严格地说，Cookie并不是由这些语言实现的，而这些语言则是实现对Cookie的间接操作，即发送HTTP指令，浏览器收到指令便操作Cookie并返回给服务器。因此，Cookie是由浏览器实现和管理的。关于Cookie的RFC文档主要有：RFC 6265、RFC 2109。

举例来说，我们经常使用PHP设置Cookie，但实际上PHP并没有真正设置过Cookie，甚至可以说，PHP根本就没有这个能力设置Cookie。它只是发出命令让浏览器来做这件事而已，形象地说就是和“有关部门”打个招呼。了解这个概念，对我们后面的学习很重要。

Cookie主要是参照RFC 2109标准由客户端实现其生成、使用等整个管理过程，服务器端则参照此标准实现和客户端之间的交互指令。

在PHP中可以使用setcookie () 或setrawcookie () 函数设置Cookie。其函数原型如下：

```
Bool setcookie ( string $ name[ , string $ value[ , int $ expire=0[ , string $ path[ , string $ domain[ , bool $ secure =false[ , bool $ httponly=false] ] ] ] ] )
```

第一个参数是必选参数，其值是Cookie的名称，即 `_COOKIE` 这个全局数组的键值。

第二个参数用来设置Cookie的值。参数为空时，Cookie值为空。由于把Cookie的值设为false会使客户端尝试删除这个Cookie，所以要在Cookie上保存true或false时不应该直接使用boolean值，而应该用0表示false，用1表示true。只要愿意，用A表示false, B表示true也是可以的。

第三个参数用来设置有效时间，以秒为单位。这个值很重要，决定了Cookie的存储方式，后面会详细讲到。

第四个参数用来设置Cookie的有效目录，默认为“/”，即整个域名下有效。如果有需要，可以设置仅在某目录下有效。

第五个参数用来设置Cookie的作用域名，默认在本域名下。需要注意的是，在IE下，包括点号长度小于等于5的短域名如果带了domain参数，会导致Cookie设置失败。

第六个参数用来设置是否对Cookie进行加密传输，默认为false。如果设置为true，只有使用HTTPS，这个Cookie才会被设置。所以，通常情况下不设置此参数或使用默认值false。

第七个参数表示是否只使用HTTP访问Cookie。如果为1或者true，客户端的

JavaScript就无法操作这个Cookie。使用此参数可以减少XSS攻击的风险，但注意，不是所有的浏览器都支持这个参数。此参数只在PHP 5.2.0以上版本有效。（再次强调，Cookie和PHP没有任何关系，只和浏览器相关，PHP只负责跟客户端打招呼，具体的管理由客户端全程完成。）

提示 `setrawcookie`的功能和参数与`setcookie`基本一样，唯一区别是`setrawcookie`不会对Cookie中的value进行urlencode转码。

设置Cookie时需要注意以下几点：

这两个函数有一个返回值，如果是false，代表设置失败；如果为true，代表设置成功。但是这个返回值仅供参考，不代表客户端一定能接收到。

由PHP在当前页设置的Cookie不能立即生效，要等到下一个页面才能看到。这是由于设置的这个页面里的Cookie由服务器传递给客户浏览器，在下一个页面浏览器才能把Cookie从客户的机器里取出传回服务器。如果是JavaScript设置的，是立即生效的。

Cookie没有显式的删除函数。如果想删除Cookie，应该将Cookie的expire设置为过期时间，如1小时前、1970年等，这会触发浏览器的删除机制。

4.7.2 PHP和JavaScript对Cookie的操作

通过前面对HTTP协议的学习我们知道，Cookie是HTTP头的一部分，即先发送或请求Cookie，然后才是data域。因此setcookie（）等函数必须在其输出数据之前调用，这和header（）函数是相同的。不过也可以使用输出缓冲函数延迟脚本的输出，直到设置好所有Cookie和其他HTTP标头。

有的时候为了避免报如下的错误，除了可以在代码中控制外，还可以通过修改php.ini文件一劳永逸。将output_buffering的值改为on或者任何数字，默认是4096。

```
warning: Cannot add header information—headers already sent in……
```

可以借助header发送HTTP指令设置Cookie，但一般不推荐这么做，如：

```
header (“Set-Cookie: name= $ value[; path= $ path[; domain=xxx.com[; ……]”)
```

下面这段代码演示了使用PHP设置Cookie，然后用客户端的JavaScript读取Cookie。

```
<? php  
setcookie (“vote”, “k100”, time () +3600 );  
print_r ($ _COOKIE );
```

运行结果如图4-42所示。

由图4-42所示可知，客户端脚本读取到了通过PHP设置的Cookie。同理，JavaScript设置的Cookie也能被PHP读取到。在上面的例子中加入

“cookie ("name", "baicai");”，这是一个JavaScript的Cookie设置函数，此函数对JavaScript的Cookie函数进行简单封装，原型如代码清单4-28所示。



图 4-42 JavaScript操作Cookie

代码清单4-28 Cookie的JavaScript操作函数

```
<script>
function cookie ( name, value, options ) {
if ( typeof value! = ' undefined ' ) {
options=options | | { };
if ( value===null ) {
value= ' ' ;

options.expires=-1;
}
var expires= ' ' ;
if ( options.expires&& ( typeof options.expires== ' number ' | | options.expires.toUTC
String ) ) { var date;
if ( typeof options.expires== ' number ' ) {
date=new Date ( );
date.setTime ( date.getTime ( ) + ( options.expires*24*60*60*1000 ) );
} else {
date=options.expires;
}
expires= ' ; expires= ' +date.toUTCString ( );
}
var path=options.path? ' ; path= ' + ( options.path ): ' ' ;
var domain=options.domain? ' ; domain= ' + ( options.domain ): ' ' ;
var secure=options.secure? ' ; secure ': ' ' ;
document.cookie=[name, ' = ', encodeURIComponent ( value ), expires, path, domain, se
cure].join ( ' ' );} else {
```

```
var cookieValue=null;
if ( document.cookie&&document.cookie! = ' ') {
var cookies=document.cookie.split ( ';' );
for ( var i=0; i<cookies.length; i++) {
var cookie= ( cookies[i] ).replace ( / ^ \ s+ | \ s+ $ /g, "" );
if ( cookie.substring ( 0, name.length+1 ) == ( name+ ' = ' ) ) {
cookieValue=decodeURIComponent ( cookie.substring ( name.length+1 ) );
break;
}}
}
return cookieValue;
}}
cookie ( "name", "baicai" );
</script >
```

使用JavaScript设置Cookie后，刷新PHP页面打印Cookie，可以看到以下内容：

```
Array ( [vote]=>k100[name]=>baicai )
```

注意 要查看客户端的Cookie，如果使用Firefox，可以在网页中右击，在弹出的快捷菜单中选择“查看页面信息”，打开“安全”选项卡，单击“查看Cookie”即可。

这个例子告诉我们：该怎么操作就怎么操作，Cookie属于浏览器，而不属于哪一门具体的语言。

4.7.3 Cookie存储机制及应用

Cookie通常用来存储一些不是很敏感的信息，或者进行登录控制，也可用来记住用户名、记住免密码登录、防止刷票等。

以防止刷票为例，演示其使用方法，如代码清单4-29所示。

代码清单4-29 Cookie使用示例：防止刷票

```
function justByOnes () {
var wid= $ ("#articleId" ).val ();
if ( cookie ( "done" ) ==wid ) {
alert ( "你已经对本文投过票了，请不要重复提交" );
return false;
} else {
//如果没有Cookie或者当前Cookie不等于当前页面值
cookie ( "done", wid ); //设置Cookie，允许提交
return;
}
}
```

当然，用Cookie防止重复投票还不够，对于有些场合，比如文章投票、评论等，没必要做严格的判断，只要禁止刷新就够了。

前面提到过setcookie还有最后一个参数HttpOnly，如果设置这个参数，JavaScript就无法读取到这个Cookie。php.ini里也可以设置这个参数，效果如图4-43所示。

这里PHP设置了两个Cookie，而客户端JavaScript只读取到了一个。



图 4-43 运行截图

某些时候，这么做能增强网站的安全性。这个机制是怎么实现的呢？实际上没什么神奇的，就是通知浏览器给Cookie加上个特殊参数，屏蔽JavaScript脚本的读取，通过其他方法还是能看到的。前面也谈到，如果浏览器不支持，即使设置HttpOnly，客户端JavaScript也能读取到。这完全取决于浏览器怎么做。

每个域名下允许的Cookie是有限制的，根据浏览器这个限制也不同。例如IE 8一个域名可以存放50个Cookie，Firefox一个域名可以存150个Cookie（Mozilla官方文章指出每个域下默认允许50个Cookie，总共默认允许1000个Cookie，但又同时提到这个数目不是固定的，而且是可以修改的），超过后就删除旧的，且一个Cookie最大字节数为4097（随着浏览器的升级换代，这个上限有可能会增加）。

Cookie的中文译名叫“甜饼、点心”，Cookie确实能给我们带来许多好的用户体验和方便的功能。但Cookie不是越多越好，它会增加宽带。一旦Cookie被设置在域下，则请求该域名下的一个资源，浏览器和服务器之间都可能存在Cookie的上行与下行流量。看似一个很小的Cookie，在一个页面请求中就可能产生十几KB流量，所以不要滥用Cookie（为了验证此结论，可以使用Fiddler进行抓包）。

不要把Cookie当做客户端的存储器来用。

前面说过，Cookie是保存在客户端的一段数据，那究竟它保存在什么地方呢？有两种情况：一种是保存在文件中，一种是保存在浏览器内存中。

这两种情况的区别是什么呢？每种浏览器的策略都不尽相同，但都实现了对Cookie的管理。比如IE是每个域名下一个文本文件，而Firefox是全部放在SQLite数据库中管理。

Firefox把Cookie保存在C:\Documents and Settings\用户名\Application Data\Mozilla\Firefox\Profiles\随机目录，目录下可以看到一个cookie.sqlite文件，所有Cookie都是保存在这个文件中。用SQLite 3打开后如图4-44所示。



id	name	value	host	path	expiry	lastAccessed	isSecure	isHttpOnly
1298473959050000	PHP	66e9f9b	google.com.hk	/	1314285230	1298473959050000	0	1
1298473959050000	afmacid	880273	google.com	/	1298473959050000	1298473959050000	0	1
1298473959050000	PHP	44c22eae	google.com.hk	/wp/	1114285114	1298473959050000	0	1
1298473959050000	PHP	1500	localhost	/Alpha	1298473959050000	1298473959050000	0	0
1298473959050000	MOCKIDST	1393322	address.mozillat	/	1496257500	1298473959050000	0	0
1298473959050000	relaid=	36353	%2F%2F	/	1298473959050000	1298473959050000	0	0
1298473959050000	PHP	ED=KZ38	google.com	/	138154008	1298473959050000	0	0
1298473959050000	PHP	ED=KZ38	google.com.hk	/	138154008	1298473959050000	0	0
1298473959050000	PHP	9050798	yahoo.com	/	138154008	1298473959050000	0	0

图 4-44 Firefox 3下的Cookie

注意 此处的文件来自Firefox 3.6, Firefox 4以上版本, 对此文件进行了加密安全处理, 只有特定的API才能读取此文件, 但内容并没有改变。此处为了讲解方便我们使用Firefox 3。

关闭浏览器, Cookie并不会随之消失。除非设置该Cookie的expire为空, 即随着浏览器关闭而消失。细心的读者可能已经从图4-44中发现, vote这个Cookie还在, 而name这个Cookie不在了。这就是因为name这个Cookie没有设置过期时间, 随着浏览器关闭而消失, 即保存在内存中。

还有一种Cookie是Flash创建的, 称为Flash Shard Object, 又称Flash Cookie, 即使清空浏览器所有隐私数据, 这类顽固的Cookie还会存在硬盘上。因为它们不受浏览器管理, 只受Flash管理。很多网站采用这种技术识别用户。最新浏览器(如Firefox 6)支持删除Flash创建的Cookie(这是由于Adobe公司向Mozilla提供了对应的钩子)。

4.7.4 Cookie跨域与P3P协议

正常的Cookie只能在一个应用中共享，即一个Cookie只能由创建它的应用获得。实现Cookie的跨域，主要是为了统一应用平台，即实现目前最流行的单点登录。最简单的方式是使用P3P协议。

P3P (Platform for Privacy Preferences) 协议由万维网协会研制，为Web用户提供了对自己公开信息的更多控制。支持P3P协议的Web站点为浏览者声明它们的隐私策略。支持P3P协议的浏览器将Web站点策略与用户隐私偏好进行对比，并为用户提出不匹配的警告，通知用户有关Web隐私的处理方式。P3P协议的官网地址：<http://www.w3.org/P3P/P3P: CP= 'CURa AD Ma-dEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE COM NAV OTC NOI DSP COR'>。

Cookie跨域涉及两个不同的应用，习惯上称为第一方和第三方。

第一方Cookie来自当前正在查看的网站，或者发送到当前正在查看的网站。

第三方Cookie来自当前正在查看的网站以外的网站，或者发送到当前正在查看的网站以外的网站。

第三方网站通常提供正在查看的网站上的内容。例如，许多站点使用来自第三方网站的广告，或者IFRAME的别的网站的URL，这些第三方的网站可能使用的Cookie。

通过P3P使用户自己可以指定浏览器的隐私策略，达到存取第三方Cookie的目的。看到这你也许会觉得这跟Web应用毫无关系，真正的问题是如何让服务器指定用户浏览器的隐私策略，这就是P3P的使命，只要在响应用户请求时，在HTTP的头信息中增加关于P3P的配置信息就可以了，步骤如下。

步骤1 编辑hosts文件，加入测试域名 (C: \ WINDOWS \ system32 \ drivers \ etc \ hosts):

```
127.0.0.1 www.atest.com
127.0.0.1 www.btest.com
```

步骤2 访问www.atest.com时，调用btest.com的页面，种下Cookie：

```
<script src="http://www.btest.com/t/get.php? id=10000"> </script >
```

http://www. btest.com/t/get.php使用P3P种下本域名下的Cookie：

```
<? php
```

```
header ( ' P3P: CP="CURa ADMa-dEVa PSAo PSDo OUR BUS UNI PUR INT DEM STA PRE  
COM NAV OTC  
NOI DSP COR" ' );  
setcookie ( "p3p",$_GET[ ' id ' ], time () +3600, "/", ".btest.com");
```

访问btest域，Cookie已经生效。http://www.btest.com/look.php内容如下：

```
<? php  
var_dump ( $_COOKIE );
```

测试结果如图4-45所示。

作为对比，我们不使用P3P，直接在atest域下设置btest域下的Cookie，看看结果。

http://www. atest.com/t/_a.php中的代码如下：

```
<? php
```



图 4-45 测试结果

```
setcookie ("p3p", ' direct cookie ', time () +3600, "/", ".btest.com");
```

再次访问<http://www.btest.com/look.php>可以看到，打印出来的数组是空的，说明跨域设置失败。

修改一下，还是在atest域下调用btest域名，只不过<http://www.btest.com/t/get.php>页面不使用P3P。发现Cookie设置成功了。

说明 要设置第三方应用下的域名，应该采取间接方式，直接在第一方应用设置是无效的。

相关注意事项总结：

页面的Cookie不能是浏览器进程的Cookie（如不设置超时时间的Cookie），否则跨域会取不到。

利用IFRAME时，记得要在相应的动态页的页头添加一下P3P的信息，否则IE会把IFRAME框里的Cookie给阻止掉，产生问题本身未保存，自然就取不到了。这其实是FRAMESET和Cookie的问题，使用FRAME或者IFRAME时都会遇到。

IE对跨域访问Cookie限制比较严格，在Firefox、Chrome等浏览器下测试，即使不发送P3P头信息也能成功。

在实际应用中，如果仅仅有两个域名，很方便互相操作对方的Cookie。如果是多个工程，其之间的关系就变得非常复杂了。这个时候，就需要一个完整的SSO方案，通常是通过一个SSO Server系统进行中转。CAS就是一个比较成熟的SSO解决方案。

4.7.5 本地存储localStorage

一个域名的每个Cookie限制以4千字节（KB）键值对的形式存在。这样的存储量基本够用，不过如果服务器端也使用的话就很难说了，因为这里就不仅是前端要用了，还多了一方，这样的数据是不可靠的，永远无法知道是不是自己把它给堵满了，或者别人把你的数据改了。另外还有一点，如果用一些调试软件（如Firebug）抓头部会发现，在网站发送请求（POST）的时候会把Cookie也带过去，而这些东西并不需要，这样就会造成无谓的宽带开销。这些都是使用Cookie存储的一些顾虑。

针对以上弊端，一个新的解决方案为：本地存储localStorage。现在主流浏览器，包括IE 8+、Firefox 3+、Chrome 4+、Opera 10.5+、Safri 4+、iPhone 2+、Android 2+等，均支持localStorage，对于一些低级浏览器如IE 6、IE 7等，可以用Flash的SharedObject模拟一个本地存储的对象，或者用UserData实现本地大数据的存储。

可使用下面的代码检测浏览器是否支持localStorage：

```
if ( window.localStorage ) {  
    alert ( ' This browser supports localStorage ' );  
} else {  
    alert ( ' This browser does NOT support localStorage ' );  
}
```

存储数据的方法很简单，就是直接给window.localStorage添加一个属性，例如window.localStorage.a或者window.localStorage["a"]。读、写、删除操作以键值对的形式存在，如下面代码所示：

```
localStorage.a=3; //设置a为"3"  
console.log ( localStorage.a );  
localStorage["a"]="abc"; //设置a为"abc"，复盖上面的值  
console.log ( localStorage.a );  
localStorage.setItem ( "b", "i am b" ); //设置b为"i am b"  
console.log ( localStorage.b );
```

```
var a1=localStorage["a"]; //获取a的值
var a2=localStorage.a; //获取a的值
console.log ( a1+" \ t"+a2);
var b=localStorage.getItem ( "b" ); //获取b的值
console.log ( b);
localStorage.removeItem ( "b" ); //清除b的值
console.log ( localStorage.getItem ( "b" ));
```

运行结果如图4-46所示。

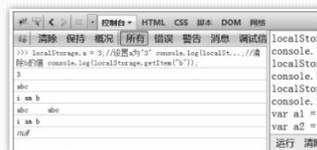


图 4-46 运行结果

如果希望一次性清除所有键值对，可以使用clear（）方法。另外，HTML 5提供了key（）方法，可以在不知道有哪些键值的时候使用，如下所示：

```
var showStorage=function () {
var storage=window.localStorage;
for ( var i=0; i<storage.length; i++) {
//key ( i ) 获得相应的键，再用getItem ( ) 方法获得对应的值
console.log ( storage.key ( i ) +": "+storage.getItem ( storage.key ( i ) ) +"<br>");
}
}
showStorage ( );
```

需要注意，HTML 5本地存储只能存字符串，任何格式存储的时候都会被自动转为字符串，所以读取的时候，需要自己进行类型转换。例如，用JSON.stringify（）方法序列化一个JSON对象存储到本地。

除了基本的读写操作外，HTML 5还提供了storage事件对键值对的改变进行监听，使用方法如下：

```
if ( window.addEventListener ) {  
window.addEventListener ( "storage", handle _ storage, false );  
} else if ( window.attachEvent ) {  
window.attachEvent ( "onstorage", handle _ storage );  
}  
function handle _ storage ( e ) {  
if (! e) { e=window.event; }  
//showStorage ();  
}
```

更多内容这里不再介绍，有兴趣的读者可以参考一些HTML 5文档和资料。

```
session_start (); $_SESSION[ ' user ' ]=$row[1];  
ob_end_flush ();
```

PHP的Session默认通过文件的方式实现，即存储在服务器端的Session文件，每个Session一个文件。一般文件名形式如下：

```
sess_4c83638b3b777545583181c2f89168ec
```

后面是随机的、32位编码的字符串。用编辑器打开它，一般内容结构如下：

```
变量名 | 类型: 长度: 值;
```

4.8 Session详解

Session即会话，指一种持续性的、双向的连接。Session和Cookie在本质上没什么区别，都是针对HTTP协议的局限性而提出的一种保持客户端和服务器间保持会话连接状态的机制。

Session的实现可以有多种，如URL重写、Cookie，通过在Cookie中存储sessionID实现Session传递。

实际上Session的实现遵照RFC 2965、RFC 2109等文档的建议（文档中仅给出了关于实现Session和Cookie存储内容、原理等的建议，并没有对具体实现做硬性规定）。

4.8.1 Session的基本概念及设置

和Cookie一样，Session也是一个通用标准，但在不同的语言中实现有所不同。由于其广泛的应用性，PHP作为一种Web开发语言，完全支持Session机制。针对Web网站来说，Session指用户在浏览某个网站时，从进入网站到浏览器关闭这段时间内的会话。由此可知，Session实际上是一个特定的时间概念。使用Session可以在网站的上下文不同页面间传递变量、用户身份认证、程序状态记录等。常见的形式就是配合Cookie使用，实现保存用户登录状态功能。和Cookie一样，`session_start()`必须在程序最开始执行，前面不能有任何输出内容，否则就会出现以下警告：

Warning: Cannot send session cookie-headers already sent

类似这样的警告信息通常是因为当前页面被包含或者包含了BOM头。

有时候，确实需要输出或者不能控制前面是否有输出，可以这么做：

```
ob_start();
```

4.8.2 Session的工作原理

我们看到，HTTP协议本身并不能支持服务器端保存客户端的状态信息。为了解决这一问题，于是引入了Session的概念，用其来保存客户端的状态信息。

Session通过一个称为PHPSESSID的Cookie和服务器联系。Session是通过sessionID判断客户端用户的，即Session文件的文件名。

用一个形象的比喻解释Session的工作方式。假设Web Server是一个商场的存包处，一个顾客（HTTP Request）第一次来到存包处，管理员把顾客的物品存放在某柜子里面（这个柜子就相当于Session），把一个号码牌交给这个顾客，作为取包凭证（这个号码牌就是sessionID）。顾客下一次来的时候，要把号码牌交给存包处的管理员。管理员根据号码牌找到相应的柜子，根据顾客请求，取出、更换、添加柜子中的物品，存包处也可以让顾客的号码牌和号码牌对应的柜子失效。顾客的忘性很大，管理员在顾客回去的时候都要提醒顾客记住自己的号码牌。这样，顾客下次来的时候，就会带着号码牌回来。

sessionID实际上是在客户端和服务器端之间通过HTTP Request和HTTP Response传来传去。sessionID按照一定的算法生成，必须包含在HTTP Request里面，保证唯一性和随机性，以确保Session的安全。如果没有设置Session的生存周期，sessionID存储在内存中，关闭浏览器后该ID自动注销；重新请求该页面，会重新注册一个sessionID。如果客户端没有禁用Cookie，Cookie在启动Session会话的时候扮演的是存储sessionID和Session生存期的角色。可以手动设置Session的生存期，代码如下：

```
$lifeTime=24*3600;  
setcookie ( session_name (), session_id (), time () + $lifeTime, " ");
```

也可使用session_set_cookie_params ()函数设置Session的生存期。Session过期后，PHP会对其进行回收。因此，Session并非都随着浏览器的关闭而消失的。

假设客户端禁用Cookie怎么办? 没办法, 所有生存周期都是浏览器进程, 只要关闭浏览器, 再次请求页面又要重新注册Session。

那么怎么传递sessionID呢? 通过URL或者隐藏表单。我们做个试验, 先禁止浏览器的Cookie, 如代码清单4-30所示。

代码清单4-30 a.php

```
echo"<a href=new.php>new1</a> </br>";  
$a=session_name ();  
$b=session_id ();  
echo"<a href=new.php? $a=".$b.">new2</a>";
```

然后在下一个页面查看我们传递来的Session, 如代码清单4-31所示。

代码清单4-31 new.php

```
$sessionName=session_name ();  
//取得sessionID  
$sessionID=$_GET[$sessionName];  
//使用session_id () 设置获得的Session  
session_id ($sessionID);  
session_start ();  
print_r ($_COOKIE);  
var_dump ($_SESSION);
```

可以看到, 从new1连接点进去是空的, 而从new2连接点进去, 即使浏览器禁用Cookie, 仍然可以得到Session。因此, 浏览器禁用Cookie, Session也会受影响, 但可以通过其他方式传值, 比如手工添加, 也可设置自动添加。php.ini中把session.use_trans_sid设成1, 那么连接的后就会自己加Session的ID。

Session以文件的形式存放在本地硬盘的一个目录中, 所以当Session比较多时, 磁

盘读取文件就会比较慢。经验告诉我们，当一个目录的文件数超过2000时，读写这个目录就会很慢。于是想到把Session分目录存放。

php.ini 里 Session 设置部分中有一项：`session.save_path="N;MODE;/path"`。这项设置可以给Session存放目录进行多级散列，其中“N”表示要设置的目录级数，“MODE”表示目录的权限属性，默认为600。Windows基本不用设置。“/path”表示Session文件存放的根目录路径，比如设置为下面的格式：

```
session.save_path="2; /tmp/phpsession"
```

上述代码表示把/tmp/phpsession目录作为PHP的Session文件存放根目录，在该目录下进行两级目录散列，每一级目录分别以0~9和a~z共36个字符作为目录名，这样存放Session的目录可以达到3636个。

注意 这里的子目录需要自己手工创建，当然是写代码。以后的Session将按sid的前两位存入对应的目录中去。

Session的回收是被动的，为了保证过期的Session能被正常回收，可以修改PHP配置文件中的`session.gc_divisor`参数以提高回收率（太大了会增加负载），或者设置一个变量判断是否过期。对于设置分级目录存储的Session，PHP不会自动回收，需要自己实现其回收机制。

4.8.3 Session入库

对于访问量大的站点，用默认的Session存储方式并不适合，较优的方法是用Data Base（可以是普通数据库、内存表、APC等）存取Session。解决这个问题的方案就是session_set_save_handler函数，代码如下：

```
bool session_set_save_handler ( callback open, callback close, callback read, callback write,
callback destroy, call
back gc )
```

只需实现这几个接口，PHP就能帮助我们进行Session管理。

举例说明，首先在数据库里执行如下语句新建一个Session数据表，如代码清单4-32所示。

代码清单4-32 新建一个Session数据表

```
CREATE TABLE 'sessions' (
' sid ' CHAR ( 40 ) NOT NULL COMMENT ' session名 ',
' data ' VARCHAR ( 200 ) NOT NULL COMMENT ' session值 ',
' update ' INT ( 10 ) UNSIGNED NOT NULL DEFAULT ' 0 ' COMMENT ' 更新时间 ',
UNIQUE INDEX ' sid ' ( ' sid ' ))
COLLATE= ' utf8_general_ci ' ENGINE=MEMORY
ROW_FORMAT=DEFAULT
```

MEMORY引擎采用内存表，所有数据存储在内存在，操作速度快，对于Session这种形式的数据正好适用。由于代码量比较多，这里只给出关键代码。为了更清楚地了解session_set_save_handler的处理流程，建议用Xdebug进行单步跟踪调试。

以上Session处理只实现了最基础的功能，可以在此基础上进行改进，如代码清单4-33所示（注释很详细）。

代码清单4-33 Session的MySQL实现

```
<? php
/**
 *@author: 猪也知道
 *@version: 2011-03-05 beta0.1
 *@description: session的数据库实现方案
 *@todo: 回收机制不完善, 存在bug。部分处理比较粗糙
 */
class SessionSaveHandle {
public $ lifeTime; //Session生命周期
private $ tosql;
public $ db;
//数据库句柄
private $ sessiondata; //Session数据
private $ lastflush;
private $ sessName= ' PHPSESSID';
public function construct ( ) {
    $ this->db=Database: getInstance ( );
    $ this->lifeTime=get_cfg_var ( "session.gc_maxlifetime" );
}
function open ( $ savePath,$ sessName ) {

return true;
}
function close ( ) {
// $ this->gc ( ini_get ( ' session.gc_maxlifetime ' ));
return true;
}
function read ( $ sid ) {
    $ format="SELECT data FROM sessions WHERE sid= ' %s' LIMIT 1";
    $ this->tosql=sprintf ( $ format,$ sid );
    $ result= $ this->db->getOne ( $ this->tosql );
    if ( ! empty ( $ result ) ) {
return $ this->sessiondata= $ result; //若需要更多数据,$ result[ ' XXX ' ]
}
}
/**
if ( ! empty ( $ result ) ) {
```

```
$this->lastflush=$result[ ' update ' ];
$this->sessiondata=$result[ ' data ' ];
return true;
//若需要更多数据,$result[ ' XXX ' ]
}
*@param type $sessID
*@param type $sessData
*@return type
*/
}
function write ( $sessID,$sessData ) {
    $now=time ( );
    $newExp= $now+ $this-> lifeTime;
    $this-> tosql="SELECT*FROM sessions WHERE sid= ' $sessID ' ";
    $result= $this-> db-> getOne ( $this-> tosql );
    if ( $sessData== ' ' | ! isset ( $sessData ) ) {
        $sessData= $this-> sessiondata;
    }
    if ( $result ) {
        $this-> db-> execute ( "UPDATE sessions
SET ' update ' = ' $newExp ',
data= ' $sessData '
WHERE sid= ' $sessID ' ");
        if ( mysql_affected_rows ( ) ) {
            return true;
        }
    } else {
        $this-> db-> insert ( "INSERT INTO sessions (
sid,
' update ',
data)

VALUES (
' $sessID ',
' $now ',
' $sessData ' ) ");
        if ( mysql_affected_rows ( ) ) {
            return true;
        }
    }
}
```

```
return false;
}
function destroy ( $sessID ) {
    $this->tosql="DELETE FROM sessions WHERE sid= ' $sessID ' ";
    if ( $this->db->execute ( $this->tosql ) ) {
        return true;
    } else {
        return false;
    }
}
function gc ( $sessMaxLifeTime ) {
    $t=time ( );
    $this->tosql="DELETE FROM sessions WHERE $t- ' update ' > $ { sessMaxLifeTime } ";
    var _dump ( $this->tosql );
    $this->db->execute ( $this->tosql );
    if ( mysql_affected_rows ( ) >0 ) {
        return TRUE;
    } else {
        return FALSE;
    }
}
}
```

测试代码如下:

```
<? php
include ' config.php ';
include ' mysql.php ';
include ' sessionHandle.php ';
$session=new SessionSaveHandle ( );
ini_set ( ' session.use_trans_sid ', 0 );
ini_set ( ' session.use_cookies ', 1 );
ini_set ( ' session.cookie_path ', ' / ' );
ini_set ( ' session.save_handler ', ' user ' );
session_module_name ( ' user ' );
session_set_save_handler ( array ( $session, "open" ), array ( $session, "close" ),
array ( $ session , "read" ), array ( $ session , "write" ), array ( $ session , "destroy" ),
```

```
array ($session, "gc"));
    session_start ();
    var_dump ($_SESSION);
    echo $_SESSION[ ' age ' ];
```

session类的使用方法如代码清单4-34所示。

代码清单4-34 session类的使用

```
<? php
/*
*session类的使用
*/
include ' config.php ';
include ' mysql.php ';
include ' sessionHandle.php ';
$session=new SessionSaveHandle ();
ini_set ( ' session.save_handler ', ' user ');
session_set_save_handler ( array ($session, "open"), array ($session, "close"),
array ($ session , "read" ), array ( $ session , "write" ), array ( $ session , "destroy" ),
array ($session, "gc"));
session_start ();
$_SESSION[ ' name ' ]= ' baicai ';
echo $_SESSION[ ' name ' ];
$_SESSION[ ' age ' ]= ' 19 ';
echo $_SESSION[ ' age ' ];
echo ' <br/>SessionID: ' .session_id ();
echo ' <a href=session2.php>session2.php</a> ';
```

在大流量的网站中，Session入库存存在效率不高、占据数据库connection资源等问题。针对这种情况，可以使用Memcached、Redis等Key Value数据存储方案实现高并发、大流量的Session存储。

4.8.4 Cookie与Session问答

Q1: Cookie运行在客户端, Session运行在服务器端, 对吗?

A: 不完全正确。Cookie是运行在客户端, 由客户端进行管理; Session虽然是运行在服务器端, 但是sessionID作为一个Cookie是存储在客户端的。

Q2: 浏览器禁止Cookie, Cookie就不能用了, 但Session不会受浏览器影响, 对吗?

A: 错。浏览器禁止Cookie, Cookie确实不能用了, Session会受浏览器端的影响。很简单的实验, 在登录一个网站后, 清空浏览器的Cookie和隐私数据, 单击后台的连接, 就会因为丢失Cookie而退出。当然, 有办法通过URL传递Session, 前文已有叙述。

Q3: 浏览器关闭后, Cookie和Session都消失了, 对吗?

A: 错。存储在内存中的Cookie确实会随着浏览器的关闭而消失, 但存储在硬盘上的不会。更顽固的是Flash Cookie, 夸张地说只有格式化硬盘, 它才会消失。不过现在很多系统软件如360安全卫士和新版浏览器已经支持删除Flash Cookie。百度采用了这样的技术记忆用户: Session在浏览器关闭后也不会消失, 除非正常退出, 代码中使用了显示的unset删除Session。否则Session可能被回收, 也有可能永远残留在系统中。

Q4: Session比Cookie更安全吗? 不应该大量使用Cookie吗?

A: 错误。Cookie确实可能存在一些不安全因素, 但和JavaScript一样, 即使突破前端验证, 还有后端保障安全。一切都要看设计, 尤其是涉及权限的时候, 特别需要注意。这个问题在正则表达式那章有过讲解。通常情况下, Cookie和Session是绑定的, 获得Cookie就相当于获得了Session, 客户端把劫持的Cookie原封不动地传给服务器, 服务器收到后, 原封不动地验证Session, 若Session存在, 就实现了Cookie和Session的绑定过程。因此, 不存在Session比Cookie更安全这种说法。如果说不安全, 也是由于代码不安全, 错误地把用作身份验证的Cookie作为权限验证来使用。

Q5: Session是创建在服务器上的, 应该少用Session而多用Cookie, 对吗?

A: 错。Cookie可以提高用户体验，但会加大网络之间的数据传输量，应尽量在Cookie中仅保存必要的数据库。

Q6: 如果把别人机器上的Cookie文件复制到我的电脑上（假设使用相同的浏览器），是不是能够登录别人的账号呢？如何防范？

A: 是的。这属于Cookie劫持的一种做法。要避免这种情况，需要在Cookie中针对IP、UA等加上特殊的校验信息，然后和服务器端进行比对。

Q7: 在IE浏览器下登录某网站，换成Firefox浏览器是否仍然是未登录状态？使用IE登录了腾讯网站后，为什么使用Firefox能保持登录状态？

A: 前面已经提过，不同浏览器使用不同的Cookie管理机制，无法实现共用Cookie。如果使用IE登录腾讯网站，使用Firefox也能登录，这是由于在安装腾讯QQ软件时，你的电脑上同时安装了针对这两个浏览器的插件，可以识别本地已登录QQ号码进而自动登录。本质上，不属于共用Cookie范畴。

4.9 本章小结

本章介绍了PHP网络应用的方方面面，从HTTP协议开始讲到了Socket、cURL、WebService、Cookie和Session。熟悉HTTP协议基本概念和使用方法对高效使用网站、优化网站性能有很大的帮助。另外，SPDY也是值得关注的新生事物。最后，分别详细阐述了Cookie和Session的概念，并且回答了一些常见的问题。

通过本章我们知道，如果所有核心业务都基于HTTP，那么架构将不具备伸缩性，而Socket可以很好地解决这个问题，通过Socket可以实现高效的数据交互。通过SMTP的例子讲解了Socket操纵已知协议进行跨应用、跨服务器的交互。同时，提醒读者关注HTML 5中最新的WebSocket应用。随着Web App的蓬勃发展，各种App盛行，如新浪微博App、支付宝接口等。要和这些App打交道，cURL是一个很好的解决方案。无论是HTTP协议，还是Socket、cURL，都需要一些辅助工具，而Fiddler就是一款专注于HTTP协议的抓包工具，本章从安装、工具界面以及功能等方面对其做了介绍。

掌握本章内容，能让你在日常的PHP开发中事半功倍。

第5章 PHP与数据库基础

数据库是Web应用的重点，往往是一个系统的瓶颈所在。掌握一些数据库设计的基本理论和优化技巧，对于提升应用的处理能力是有帮助的。数据库知识不仅是DBA的知识范畴，也是程序员必须掌握的。由于程序员更接近应用，更容易知道应用的瓶颈，所以他们更容易犯错。

本章仅针对工作中可能接触到的知识点展开介绍。讲述基本的数据库设计应遵循的一些原则和优化技巧。

5.1 什么是PDO

如果说PHP是脚本语言中的利器，那么对数据库的支持无疑让PHP如虎添翼。最初，“支持几乎市面上所有的数据库”成了PHP最大的卖点。对数据库的支持使PHP的应用范围更广。事实上，PHP对数据库支撑并不好，最重要的一点就是抽象度不够，访问接口不统一。

PHP针对每种数据库都有一个独立的模块、一组独立的函数。这样的结构和设计让PHP兼容多种数据库变得困难。一旦要将一个应用移到另外一种数据库环境中，或者需要添加新的数据库支持，就不得不重新编写和数据库相关的操作。通常编写多个类，用适配器模式实现。在这个历史背景下PDO出现了。PDO（PHP Data Objects）提供一个通用接口访问多种数据库，即抽象的数据模型支持连接多种数据库。有了PDO使代码变得更简洁、更安全。

在PHP中，连接MySQL数据库通常有三种选择：

MySQL系列函数：最常用，是过程式风格的一组应用。

MySQLi系列函数：是MySQL函数的增强改进版，提供了过程化和面向对象两种风格的API，增加了预编译和参数绑定等新的特性。

PDO: 从语法上讲, PDO更接近MySQLi (PHP 5.1及以上版本支持)。

PDO扩展为PHP定义了一个访问数据库的轻量、持久的接口。实现PDO接口的每一种数据库驱动都能以正则扩展的形式把各自的特色表现出来。

注意 PDO扩展只是一个抽象的接口层, 利用PDO扩展本身并不能实现任何数据库操作, 必须使用一个特定的数据库PDO驱动访问数据库。

PDO提供一个数据访问抽象层, 这就意味着不管使用哪种数据库, 都可以用同样一组API对数据进行操作, 保证了可抽象性和访问接口的一致性。

Windows上启用PDO很简单, 在php.ini文件里找到以下语句, 把前面的分号去掉, 同时选择一个特定的数据库类型即可:

```
; extension=php_pdo.dll
```

Linux操作系统上启用的方法与上述方法类似。

5.1.1 PDO预定义类

PDO中包含三个预定义类: PDO、PDOStatement和PDOException, 下面分别简单介绍。PDOException类是对exception的简单重写, 这里不做介绍。

1. PDO类

PDO类代表一个PHP和数据库之间的连接, PDO类所拥有的方法如下:

PDO: 构造器, 构建一个新的PDO对象。

beginTransaction: 开始事务。

commit: 提交事务。

errorCode: 从数据库返回一个错误代号, 如果有的话。

`errorInfo`: 从数据库返回一个含有错误信息的数组, 如果有的话。

`exec`: 执行一条SQL语句并返回影响的行数。

`getAttribute`: 返回一个数据库连接属性。

`lastInsertId`: 返回最新插入到数据库的行 (ID)。

`prepare`: 为执行准备一条SQL语句, 返回语句执行后的联合结果集 (`PDOStatement`)。

`query`: 执行一条SQL语句并返回一个结果集。

`quote`: 返回添加引号的字符串, 使其可用于SQL语句中。

`rollBack`: 回滚一个事务。

`setAttribute`: 设置一个数据库连接属性。

2. `PDOStatement`类

`PDOStatement`类代表一条预处理语句以及语句执行后的联合结果集 (`associated result set`), 其拥有的方法如下:

`bindColumn`: 绑定一个PHP变量到结果集中的输出列。

`bindParam`: 绑定一个PHP变量到一个预处理语句中的参数。

`bindValue`: 绑定一个值到与处理语句中的参数。

`closeCursor`: 关闭游标, 使语句可以再次执行。

`columnCount`: 返回结果集中的列的数量。

`errorCode`: 从语句中返回一个错误代号, 如果有的话。

`errorInfo`: 从语句中返回一个包含错误信息的数组, 如果有的话。

`execute`: 执行一条预处理语句。

`fetch`: 从结果集中取出一行。

`fetchAll`: 从结果集中取出一个包含所有行的数组。

`fetchColumn`: 返回结果集中某一列中的数据。

`getAttribute`: 返回一个PDOStatement属性。

`getColumnMeta`: 返回结果集中某一列的结构 (metadata?)。

`nextRowset`: 返回下一个结果集。

`rowCount`: 返回SQL语句执行后影响的行数。

`setAttribute`: 设置一个PDOStatement属性。

`setFetchMode`: 为PDOStatement设定获取数据的方式。

5.1.2 如何使用PDO

使用PDO的第一步也是最主要的一步就是配置数据源，之后和通常使用的MySQL扩展操作数据库的方法没有什么区别，只是使用面向对象风格的API。看一个示例，如代码清单5-1所示。

代码清单5-1 PDO使用示例

```
<? php
/**
 @author: waitfox
 @describe: PDO演示
 **/
try {
 $dsn="mysql: host=localhost; dbname=test"; //配置PDO的数据源
 $db=new PDO ($dsn, ' root ', ' 1234 ');
 //构造方法
 //设置异常可捕获
 $db->setAttribute ( PDO: ATTR_ERRMODE, PDO: ERRMODE_EXCEPTION );
 $db->exec ( "SET NAMES ' UTF8 ' " );
 $sql="INSERT INTO tc0 _log ( name, content, ip, datetime ) values ( ' admin ', ' 插入一条记录
', ' {$_SERVER[ ' RE
MOTE_ADDR ' ]} ', now ( ) )";
 //插入到日志表里
 $db->exec ( $sql );
 //使用预处理语句
 $ insert= $ db- > prepare ( "INSERT INTO tc0 _ log ( name, content, ip, datetime )
values ( ?, ?, ?, now ( ) ) " );
 $ insert->execute ( array ( ' admin ', ' 插入一条记录1111 ', " {$_SERVER[ ' REMOTE _
ADDR ' ]} " ) );
 $ insert->execute ( array ( ' admin ', ' 插入一条记录2222 ', " {$_SERVER[ ' REMOTE _
ADDR ' ]} ", 8, 9 ) ); //异常
 $ sql="select name, content, ip, datetime from tc0 _log";
 $ query= $ db->prepare ( $ sql );
 $ query->execute ( );
 var _dump ( $ query->fetchAll ( PDO: FETCH_ASSOC ) );
 } catch ( PDOExceptionn $err ) {
```

```
echo $err->getMessage ();  
}
```

是不是很简单？

5.1.3 PDO参数绑定与预编译

PDO最大的特点是引入参数绑定和预编译。参数绑定和预编译是什么关系？其实二者是同一件事的不同阶段。PDO参数绑定示例如代码清单5-2所示。

代码清单5-2 PDO参数绑定示例

```
<? php
/*Execute a prepared statement by binding PHP variables*/
$calories=150;
$colour= ' red ';
$sth=$dbh->prepare ( ' SELECT name, colour, calories
FROM fruit
WHERE calories<: calories AND colour=: colour ');
$sth->bindParam ( ': calories ', $calories, PDO: PARAM_INT );
$sth->bindParam ( ': colour ', $colour, PDO: PARAM_STR, 12);
$sth->execute ();
$sth=$dbh->prepare ( ' SELECT name, colour, calories
FROM fruit
WHERE calories<? AND colour=? ');
$sth->bindParam ( 1, $calories, PDO: PARAM_INT );
$sth->bindParam ( 2, $colour, PDO: PARAM_STR, 12);
$sth->execute ();
```

在MySQL应用中，为了防止注入攻击，通常使用intval、addslashes等函数对传入的参数进行转义，转变成SQL中合法的参数类型，这种方法较复杂，而使用PDO的bindParam方法，一切变得更简单快捷，只需在函数中指定第三个参数，即可把传入的参数转换为需要的类型并拼接到原先的SQL语句中。那这个功能是怎么实现的呢？用PHP模拟这个过程，如代码清单5-3所示。

代码清单5-3 PHP模拟PDO参数绑定功能

```
<? php
```

```
function bindParam (& $sql,$location,$var,$type) {
//确定类型
switch ($type) {
//字符串
default: //默认使用字符串类型
case ' STRING ':
$var=addslashes ($var); //转义
$var="'".$var."'";
//加上单引号.SQL语句中字符串插入必须加单引号
break;
case ' INTEGER ': case ' INT ':
$var= (int)$var;
//强制转换成int
//还可以增加更多类型..
}
for ($i=1,$pos=0; $i<=$location; $i++) {
$pos=strpos ($sql, '? ', $pos+1);
}
//替换问号
$sql=substr ($sql, 0,$pos) .$var.substr ($sql,$pos+1);
}
```

使用下面的代码试着传入一些非法参数:

```
<? php
$uid=10086;
$pwd="pwd";
$sql="SELECT*FROM table WHERE uid=? AND pwd=? ";
bindParam ($sql, 1,$uid, ' INT '); //在第一个问号处绑定 $uid为INT型变量
bindParam ($sql, 2,$pwd, ' STRING '); //在第二个问号处绑定 $pwd为STRING型变量
echo $sql;
```

非法参数例如:

```
uid="good", pwd="123456 ' or ' 1 ' = ' 1"
```

可以看到，生成的SQL语句做了对应的转义和过滤处理。

其实，PHP的PDO思想来自于JDBC。JDBC（Java-dataBase Connectivity, Java数据库连接）是用于执行SQL语句的Java API，为多种关系数据库提供统一访问。JDBC有一个PreparedStatement类，用于实现JDBC的参数绑定。由于Java与Oracle数据库搭配比较紧密，Oracle自身实现了参数绑定这一特性，使SQL语句查询效率提高10~20倍。故在JDBC中，默认实现了这个类。

由于PHP自身和MySQL实现的原因，PDO中的参数绑定无法起到JDBC的效果，实际上其主要作用是数据过滤与安全。MySQL支持预编译，但是很弱，仅仅是Session级别，对执行效率提升不明显。

预编译负责两件事，转义和软解析提速。程序要支持预编译，除了数据库支持外，还需要驱动支持（PDO和MySQLi均支持）。

注意 使用PDO从MySQL数据库查询的数据都是string，在某些特殊应用下，可能需要转换格式。

5.1.4 PDO事务处理

一个事务中所有的工作在提交时，即使是分阶段执行，也要保证安全地应用于数据库，不被其他的连接干扰。事务工作可以在请求发生错误时轻松地自动取消。

事务的主要特性：原子性、一致性、独立性和持久性（Atomicity, Consistency, Isolation and Durability, ACID）。典型运用就是通过把批量的改变“保存”然后立即执行，这样就能提高效率了。一旦事务不成功，将回滚到初始状态，保证数据的一致性。

SQL通常工作在“自动提交”模式下，这意味着执行的每个查询都有自己隐含的事务处理，无论是数据库支持事务，还是因数据库不支持而不存在事务，DML语句执行的结果都将立即生效并不可更改。例如，在MySQL中执行一条update或者delete语句，其功能将立即生效，并且是永久性的，不可更改的。而在Oracle数据库中，默认是事务模式，要delete一条数据，数据并不会被永久性删除，只有执行了commit命令后才会生效。

PDO中使用beginTransaction（）方法创建事务。在一个事务中，使用commit（）或rollback（）方法结束事务，具体应用哪种方法这取决于事务中代码运行是否成功。脚本结束或一个连接要关闭时，如果还有一个未处理完的事务，PDO自动将其回滚。这对于脚本意外终止情况来说是一个安全方案，如果没有明确地提交事务，它将假设发生一些错误，为数据的安全执行回滚。

自动回滚仅发生于通过beginTransaction（）建立的事务。如果用手动方式执行一个开始事务的查询，PDO无法知道它的情况，故无法回滚。例如：

```
$db->beginTransaction(); //开启事务
$db->exec("INSERT INTO tc0_log ( name, content, ip, datetime ) values ( ' admin ', ' bb
', ' {$_SERVER[ ' REMOTE_
ADDR ' ]} ', now ( ) )");
//正常执行的SQL语句
$db->exec("insert into staff XXX"); //这条语句是错误的，所以无法执行
$db->commit();
```

上述代码中，因为开启了事务，第二条SQL执行失败会导致所有SQL回滚而无法执行。因此，第一条数据也没有插入。这就是事务原子性的体现：要么全部成功，要么全部失败。

提示 因为使用了事务，要么成功，要么失败。如果测试的时候发现第一条执行了，而第二条SQL语句却失败了，则应该检查表类型是否为MyISAM。MyISAM引擎是不支持事务的，请改用InnoDB存储引擎或其他支持事务的引擎。这点经常被遗忘。

5.1.5 PDO的效率问题

既然PDO有如此多的特性，那么应不应该马上使用PDO呢？首先需要考虑效率问题。PDO效率到底怎样？

基于PHP 5.3，我使用一个包含60多个表、大小为2GB数据库进行本地测试，经过反复测试，PDO的CRUD（增删改查）效率比MySQL直连低5%~15%，并且方差大于MySQL直连。如果项目对运行效率要求严格，则应使用MySQL或MySQLi。

至于负载方面，笔者未能进行生产过程中的实测，但根据众多使用者的测试，PDO开启长连接后负载高于MySQL且比较稳定。另有使用者测试，PDO连接MySQL、Oracle速度比直连有优势。

应用会迁移到其他数据库吗？实际应用中，90%的程序是不会进行数据库迁移的，有数据库迁移的应用程序少之又少。由于每种数据库特性的千差万别，在语法和优化上更是不可能一致的，PDO无法做到一处编写，到处应用。所以从这个角度看，似乎必要性也不是很大。这些顾虑也导致PDO推广的困难。

综上所述，即使是在Oracle中，预编译和参数绑定也不一定就能提速，反而有可能会因为执行计划被改变造成效率低下。有些数据库预编译反而会造成效率下降。推荐在新应用中尝试使用PDO，旧的应用则没必要进行重构。

5.2 数据库应用优化

数据库的优化主要包括两个方面，一方面是SQL程序语句的优化，另一方面是数据库服务器和配置的优化。查询语句优化主要涉及两个方面：一些普遍遵循的原则，以及怎样对查询语句进行性能分析。

为了更好地体会SQL语句带来的效率差异，建议手工创建几个结构复杂的表，多导入一些数据（例如导入10万条以上）进行测试，效果会比较直观。

5.2.1 基本语句优化10个原则

在数据库的应用中，程序员们通过不断地实践总结了很多经验，这些经验是一些普遍适用的规则。每一个程序员都应该了解并记住它们，在构造SQL时，养成良好的习惯。以下列举10个比较重要的原则供大家参考。

原则1：尽量避免在列上进行运算，这样会导致索引失效。例如原句为：

```
SELECT * FROM t WHERE YEAR ( d ) >= 2011;
```

优化为：

```
SELECT * FROM t WHERE d >= ' 2011-01-01 ';
```

原则2：使用JOIN时，应该用小结果集驱动大结果集。同时把复杂的JOIN查询拆分成多个Query。因为JOIN多个表时，可能导致更多的锁定和堵塞。例如：

```
SELECT * FROM a JOIN b ON a.id=b.id LEFT JOIN c ON c.time=a.date  
LEFT JOIN d ON c.pid=b.aid LEFT JOIN e ON e.cid=a.did
```

原则3：注意LIKE模糊查询的使用，避免%%。例如原句为：

```
SELECT*FROM t WHERE name LIKE ' %de% '
```

优化为：

```
SELECT*FROM t WHERE name >= ' de ' AND name < ' df '
```

原则4：仅列出需要查询的字段，这对速度不会有明显影响，主要考虑节省内存。例如原句为：

```
SELECT*FROM Member;
```

优化为：

```
SELECT id, name, pwd FROM Member;
```

原则5：使用批量插入语句节省交互。例如原句为：

```
INSERT INTO t ( id, name ) VALUES ( 1, ' a ' );  
INSERT INTO t ( id, name ) VALUES ( 2, ' b ' ); INSERT INTO t ( id, name ) VALUES ( 3, ' c ' );
```

优化为:

```
INSERT INTO t (id, name) VALUES (1, 'a'), (2, 'b'), (3, 'c');
```

原则6: limit的基数比较大时使用between。例如原句为:

```
select*from article as article order by id limit 1000000, 10
```

优化为:

```
select*from article as article where id between 1000000 and 1000010 order by id
```

between限定比limit快，所以在海量数据访问时，建议用between或是where替换掉limit。但是between也有缺陷，如果id中间有断行或是中间部分id不读取的情况，总读取的数量会少于预计数量！

在取比较后面的数据时，通过desc方式把数据反向查找，以减少对前段数据的扫描，让limit的基数越小越好！

原则7: 不要使用rand函数获取多条随机记录。例如:

```
select*from table order by rand () limit 20;
```

使用下面的语句代替:

```
SELECT * FROM ' table ' AS t1 JOIN ( SELECT ROUND ( RAND ( ) * ( ( SELECT MAX ( id ) FROM ' table ' ) - ( SE LECT MIN ( id ) FROM ' table ' ) ) + ( SELECT MIN ( id ) FROM ' table ' ) ) AS id ) AS t2 WHERE t1.id > =t2.id ORDER BY t1.id LIMIT 1;
```

这是获取一条随机记录，这样即使执行20次，也比原来的语句高效。或者先用PHP产生随机数，把这个字符串传给MySQL，MySQL里用in查询。

原则8：避免使用NULL。

原则9：不要使用count (id)，而应该是count (*)。

原则10：不要做无谓的排序操作，而应尽可能在索引中完成排序。

5.2.2 索引与性能分析

怎么知道SQL执行效率高呢？例如以下这段代码：

```
Mysql> set @@profiling=1;
Mysql> select * from typecho_comments order by mail limit 10, 30;
Mysql> show profiles;
```

直观的方法就是看客户端返回的执行时间。此外，还有更精确的做法，就是查看性能报告。以笔者的博客为例，打开性能分析开关，就能看到MySQL对每一条执行计划的详细报告，如图5-1所示。

```
4 | 0.0334545 | select * from typecho_comments order by mail limit 10,30 |
5 | 0.00219125 | select * from typecho_comments order by mail limit 10,30 |
-----
5_rows_in_set
```

图 5-1 SQL性能分析报告

对于同一条语句，第二次查询明显比第一次快，因为MySQL缓存了查询。查看第四条SQL语句执行的细节：

```
Mysql> show profile for query 4;
```

用这种方法可以定位性能瓶颈，如图5-2所示。

Status	Duration	Status	Duration
starting	7.1E-5	starting tables	0.000102
opening tables	1.1E-5	opening tables	2.4E-5
system lock	3E-6	system lock	9E-6
table lock	8E-6	table lock	1.2E-5
init	2.4E-5	init	4E-5
optimizing	4E-6	optimizing	6E-6
statistics	1.3E-5	statistics	1.3E-5
preparing	7E-6	preparing	1.1E-5
executing	1E-6	executing	2E-6
sorting result	0.03728	sorting result	0.0008
sending data	0.00041	sorting data	0.000908
end	3E-6	end	1.3E-5
query end	3E-6	query end	3E-6
freeing items	8.2E-5	freeing items	0.000242
logging slow query	1E-6	logging slow query	3E-6
cleaning up	8E-6	cleaning up	5E-6
16 rows in set		16 rows in set	

图 5-2 SQL语句执行细节对比

从图5-2所示的结果看到，两条完全相同的语句，第二次执行明显比第一次快多。这

就是SQL缓存的结果。

MySQL执行计划就是在一条SELECT语句前放上关键词EXPLAIN, MySQL解释它将如何处理SELECT, 提供有关表如何联合和以什么次序联合的信息。借助于EXPLAIN可以知道:

什么时候必须为表加入索引, 以得到一个使用索引找到记录的更快的SELECT方法。

优化器是否以一个最佳次序联结表。

为了强制优化器对一个SELECT语句使用一个特定联结次序, 需增加一个STRAIGHT_JOIN子句, 方法如下:

```
EXPLAIN[EXTENDED]SELECT select_options
```

对比较复杂的查询进行计划分析时, 可能会得到多条执行计划, 例如:

```
Mysql>explain select*from event;
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | event | ALL | NULL | NULL | NULL | NULL | 13 | |
```

各属性的含义如下:

id: 查询的序列号。

select_type: 查询的类型, 主要包括普通查询、联合查询和子查询。

table: 所访问的数据库中表的名称。

type: 联合查询使用的类型。

possible_keys: 指出MySQL能使用哪个索引在该表中找到该行。如果这个值是空的,则表示没有相关的索引。这时要提高性能,可通过检验WHERE子句,看是否引用了某些字段,或者检查字段是否适合索引。

key: 显示MySQL实际决定使用的键。如果没有索引被选择,键是NULL。

key_len: 显示MySQL决定使用的键长度。如果键是NULL,长度就是NULL。注意,这个值可以反映出多重主键里MySQL实际使用了哪部分。

ref: 显示哪个字段或常数与key一起被使用。

rows: 这个值表示MySQL要遍历多少数据才能找到所需的结果集,其在InnoDB上是不准确的。

Extra: 如果是Only index,意味着信息只能用索引树中的信息检索,这比扫描整个表要快;如果是where used,则表示使用了where限制,但是用索引还不够;如果是impossible where,则表示通过收集到的统计信息判断出不可能存在的结果。除此之外,Extra还有下面一些可能值:

Using filesort: 表示包含orderby且无法使用索引进行派讯操作时,不得不使用相应的派讯算法实现。

using temporary: 使用临时表,常见于orderby和group by。

select tables optimized way: 使用聚合函数,并且MySQL进行了快速定位。通常是MAX, MIN, COUNT(*)等函数。

需要说明的是, type显示的访问类型是较重要的指标,结果值从好到坏依次是: system(系统表)、const(读常量)、eq_ref(最多一条匹配结果,通常是通过主键访问)、ref(被驱动表索引引用)、fulltext(全文索引检索)、ref_or_null(带空值的索引查询)、index_merge(合并索引结果集)、unique_subquery(子查询中返回的字段是唯一组合或索引)、index_subquery(子查询返回的是索引,但非主键)、range(索引范围扫描)、index(全索引扫描)、ALL(全表扫描)。

一般来说，保证查询至少达到range级，最好能达到ref级。ALL为全表扫描，是最坏的情况，这种情况往往是用不上索引。

MySQL索引建立和使用的基本原则如下：

合理设计和使用索引。

在关键字段的索引上，建与不建索引，查询速度相差近100倍。

差的索引和没有索引效果一样。

索引并非越多越好，因为维护索引需要成本。

每个表的索引应在5个以下，应合理利用部分索引和联合索引。

不在结果集中的结果单一的列上建索引。比如性别字段只有0和1两种结果集，在这个字段上建索引并不会太多帮助。

建索引的字段结果集最好分布均匀，或者符合正态分布。

5.2.3 服务器和配置的优化

MySQL中有多种存储引擎，每种存储引擎都有自己的特色。想要好的性能，第一步就是选择合适的数据库引擎。MySQL中常见的三种引擎特点如表5-1所示。

	MyISAM	Memory	InnoDB
用途	快速	内存数据	完整的事务支持
锁	全表锁定	全表锁定	多种隔离级别的行锁
持久性	基于表恢复	无磁盘 I/O, 无可持久性	基于日志的恢复
事务特性	不支持	不支持	支持
支持索引类型	B-tree/FullText/B-tree	Hash/B-tree	Hash/B-tree

通常的观点是MyISAM注重性能，InnoDB注重事务，故一般使用MyISAM类的表做非事务型的业务。

这种观点产生于早期InnoDB引擎还不成熟的时候，而事实上并不是这样。MySQL在高并发下的性能瓶颈很明显，主要原因就是锁定机制导致的堵塞。而InnoDB在锁定机制上采用了行级锁，不同于MyISAM的表级锁，行级锁在锁定上带来的消耗大于表级锁，但是在系统并发访问量较高时，InnoDB整体性能远高于MyISAM。同时，InnoDB的索引不仅缓存索引本身，也缓存数据，所以InnoDB需要更大的内存，应在这个年代，内存是很廉价的。

选择最合适的存储引擎是优化的第一步。

1. 存储引擎的选择方法

在介绍存储引擎的选择方法之前，首先了解读写比（R/W）的概念。通过在数据库中执行show global status得到系统当前状态。在这些变量中，形如Com_XXX的语句表示XXX语句执行的次数。如Com_select表示SELECT语句执行的次数，以此类推。通过计算读类型和写类型语句的比例，即可确定一个粗糙的读写比例。理想的读写比为100:1，当读写比达到10:1的时候，就认为是以写为主的数据库了。一般来说，这个值在30:1左右，如图5-3所示。

Command type	Total count	Average per hour	Average per second	Percentage
select	225,949,258	463,675.5	122.1	48.5%
insert	286,009,524	368,052.9	102.2	36.9%
commit	150,361,256	397,168.9	54.8	18.8%
update	5,158,839	9,238.3	2.6	0.9%
insert	41,772,827	7,962.7	2.1	0.7%
show variables	939,320	1,678.2	0.5	0.2%
show columns	938,676	1,677.4	0.5	0.2%
show execute	637,797	1,129.5	0.3	0.1%
show process	637,797	1,129.5	0.3	0.1%
show slave	629,191	1,124.5	0.3	0.1%
insert select	574,109	1,025.7	0.3	0.1%
delete	522,877	934.2	0.3	0.1%
begin	463,641	828.3	0.2	0.1%

图 5-3 查询结果

选择存储引擎的基本原则如下。

(1) 采用MyISAM引擎

R/W > 100:1且update相对较少;

并发不高, 不需要事务;

表数据量小;

硬件资源有限。

(2) 采用InnoDB引擎

R/W比较小, 频繁更新大字段;

表数据量超过1000万, 并发高;

安全性和可用性要求高。

(3) 采用Memory引擎

有足够的内存;

对数据一致性要求不高, 如在线人数和Session等应用;

需要定期归档的数据。

2.MySQL服务器调整优化措施

1) 关闭不必要的二进制日志和慢查询日志, 仅在内存足够或开发调试时打开它们。
使用下面的语句查看查询是否打开。

```
show variables like ' %slow% ';
```

还可以使用下面的语句查看慢查询的条数，定期打开方便优化。

```
Mysql> show global status like ' %slow% ';
```

但是慢查询也会带来一些CPU损耗。建议间断性打开慢查询日志来定位性能瓶颈。

2) 适度使用Query Cache。

3) 增加MySQL允许的最大连接数。可用下面的语句查看MySQL允许的最大连接数。

```
Mysql> show variables like ' max_ connections ';
```

4) 对于MyISAM表适当增加key_buffer_size。当然这需要根据key_cache的命中率进行计算，例如：

```
Mysql> show global status like ' key_ read% ' ;  
key_ cache_ miss_ rate=Key_ reads/Key_ read_ requests*100%;
```

当key_cache_miss_rate值大于1%时就需要适当增加key_buffer_size了。

对于MyISAM，还需要注意table_cache的设置。当table_cache不够用的时候，MySQL会采用LRU算法踢掉最长时间没有使用的表；如果table_cache设置过小，MySQL就会反复打开、关闭FRM文件，造成一定的性能损失；如果table_cache设置过大，MySQL将会消耗很多CPU资源去处理table_cache的算法。因此table_cache值一定要设置合理，可以参考opened_tables参数的值，如果这个值一直增长，就需要适当增加table_cache值。

对于InnoDB，需要重点关注`innodb_buffer_pool_size`参数。

5) 从表中删除大量行后，可运行`OPTIMIZE TABLE TableName`进行碎片整理。

5.2.4 MySQL瓶颈及应对措施

MySQL是存在瓶颈的。当MySQL单表数据量达到千万级别以上时，无论如何对MySQL进行优化，查询如何简单，MySQL的性能都会显著降低，这时候可以采取以下措施。

1) 增加MySQL配置中buffer和Cache的数值，增加服务器CPU数量和内存的大小，这样能很大程度上应对MySQL的性能瓶颈。性能优化中，效果最显著、成本最低的当属硬件和服务器优化，所以这是应该首先考虑的。

2) 使用第三方引擎或衍生版本。如Percona在功能和性能上较MySQL有着很显著的提升；由MySQL创始人开发的免费MariaDB在InnoDB引擎上的性能也比MySQL优秀；据官网介绍，另一款改进的产品TokudB，性能是MySQL的10倍以上。以上这些衍生版或改进版的MySQL主要都是针对MySQL的InnoDB引擎。InnoDB每次提交事务时，为了保证数据已经持久化到磁盘（Durable），需要调用一次fsync告知文件系统，将可能在缓存中的数据刷新到磁盘。而fsync操作本身是非常“昂贵”的（消耗较多的I/O资源，响应较慢），如果每次事务提交都单独做fsync操作，那么这里将是系统TPS的一个瓶颈，所以就有了Group Commit的概念。MySQL 5.0后，出于分布式架构的考虑，为了保证InnoDB内部Commit和MySQL日志的顺序一致，InnoDB被迫放弃支持Group Commit。之后，就一直没有好的解决方案了。直到出现MariaDB，才比较完美地解决了这个问题。

3) 迁移到其他数据库。如开源的PostgreSQL和商业的Oracle。与Oracle和PostgreSQL相比，MySQL属于线程模式，并且采用了插件引擎的架构。这种实现的确有自己的优势：轻巧快速、系统资源消耗小、支持更多并发连接，但进程模式能更充分地应用CPU资源，在应付复杂业务查询上更有优势。比如，在常规优化的前提下，Oracle的单表性能瓶颈经验值在2亿数据量的级别，远远优于MySQL。不仅如此，在关联查询和内置函数等功能上，Oracle都是完胜MySQL数据库的。再比如，PostgreSQL数据库相比MySQL，拥有更强大的查询优化器，不会频繁重建索引，支持物化视图等优势。当然，迁移到其他数据库还要看应用的类型，比如是偏向OLTP（On Line Transaction Processing，联机事务处理）的应用还是偏向OLAP（On Line Analytical Processing，联机分析处理）应用。

4) 对数据库进行分区、分表操作，减少单表体积。

5) 使用NoSQL等辅助解决方案，如Memcached、Redis。

6) 使用中间件做数据拆分和分布式部署，这方面的典型案例有阿里巴巴对外开源的数据库中间件Cobar。

7) 使用数据库连接池技术。在第3点，我们讲过，MySQL是线程模式，可以支持更多的并发连接数。MySQL能支持远比Oracle和PostgreSQL更多的连接。所以Oracle和PostgreSQL应用中通常用数据库连接池技术来复用连接。那么MySQL为什么还需要用这些数据库应用中常见的数据库连接池技术呢？原因在于MySQL的锁机制还不够完善，同时程序中的一些问题都有可能导致MySQL数据库连接阻塞，在并发大的情况下，就会浪费很多链接资源和反复连接的消耗。使用数据库连接池，让连接进行排队和复用，一定程度上可以缓解高并发下的连接压力。

MySQL瓶颈是真实存在的，但是不少大型互联网公司仍然在使用MySQL，并且能使用的很好，这一方面是因为这些公司的技术实力足以对MySQL进行二次开发和改进，另一方面则得益于其成熟的架构。所以，一个工具能不能用好，人的因素占很大的比重。

5.3 数据库设计

5.3.1 范式与反范式

数据库范式是数据库设计中必须掌握的知识，没有对范式的理解，就无法设计出高效率、优雅的数据库。

为了规范数据库的设计，在数据库理论发展的过程中，逐渐形成了数据库范式的理论。到目前为止，一般认为数据库设计中有五大范式，这五大范式又是层次递进的。

但是为什么又要反范式呢？甚至连第三范式都要反？这就是反范式所要讲的内容。

到了第三范式，通常已经足够满足业务需求了，表之间的关系也比较清晰，容易维护。那究竟是什么原因使得我们要反范式呢？要明白这个原因，首先应介绍范式理论的背景。

数据库范式理论在20世纪70年代初提出，并且在20世纪80年代基本完善定型。那个时候的系统存在如下特征：可用的存储器资源极其有限，几百兆字节大小的磁盘就算大的了，而现在的硬盘动辄几百GB，甚至上TB；同时，那个时候网络还不成熟，能使用网络的人很少，通常只是涉及单机的计算性能。因此，数据库范式理论强调减少依赖、降低冗余是有其历史背景的。而现在，硬盘容量比当时大了几万倍，硬盘变得廉价，数据存储不再是问题；同时，面临着高并发，业务逻辑极度复杂，低延迟要求的情况，此时，还一味地固执遵循范式设计理论是不当的。适当降低范式，增加冗余，用空间来换时间是值得的。最低可以把范式降低到第一范式。

通常在设计数据库时需要遵循如下原则。

1) 核心业务使用范式。在类似交易有关的这种敏感和核心业务中，强调数据安全和一致性，需要遵循范式保证机密数据不被破坏，核心业务不出现不一致的情况。什么样的业务才能叫核心业务，这要依照实际情况而定。

2) 弱一致性需求——反ACID。在一些对数据一致性要求不高的场合，不必完全遵循ACID，出现适当的数据不一致是可以容忍的。如在线人数统计，静态页等。最近几年流

行的NoSQL技术，就是基于弱一致性需求，降低数据完整性和一致性换取效率的。

3) 空间换时间，冗余换效率。由于一条可见记录被拆分到了多个表中进行记录，当数据量比较大的时候，联表查询就变得比较费时，SQL语句也变得很复杂，难于优化。此时就需要适当的冗余了。在统计报表，视图中就是对这一规则的具体体现。统计报表通常会有很多列，有的甚至多到上百列，需要关联几个甚至十几个表进行查询。如果每次查看统计信息都进行表关联查询，速度缓慢不说，更严重的情况是使用的人一多，就可能导致数据库服务器宕机。这种情况就需要冗余表了，冗余表一般符合第一或第二范式。那冗余表怎么处理？一般是定期转储。很少有人会去实时查询3年前某个月的销售数据明细表，因为他只要看报表就可以了。

4) 避免不必要的冗余。范式理论不是想反就能反的，反范式理论不是说不要范式，而是在必要时创建冗余表或总结表。不必要的冗余仍然是要避免的。所有的规则都是有使用场景的，我们不应该固守规则，在某些情况下，应懂得变通。

5.3.2 数据库分区

所谓分区，就是把一个数据表的文件和索引分散存储在不同的物理文件中。MySQL5.1及以上版本才支持分区。

可使用如下命令确认版本是否支持分区：

```
Mysql> SHOW VARIABLES LIKE '%partition%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_partitioning | YES |
+-----+-----+
```

MySQL支持的分区类型包括Range、List、Hash、Key，其中Range最常用：

```
CREATE TABLE foo (
id INT NOT NULL AUTO_INCREMENT,
created DATETIME,
PRIMARY KEY (id, created)
) ENGINE=INNODB PARTITION BY RANGE ( TO_DAYS ( created )) (
PARTITION foo_1 VALUES LESS THAN ( TO_DAYS ( ' 2009-01-01 ' )),
PARTITION foo_2 VALUES LESS THAN ( TO_DAYS ( ' 2010-01-01 ' ))
)
```

即便创建完分区，也能在后期管理，比如添加一个新的分区，代码如下：

```
ALTER TABLE foo ADD PARTITION (
PARTITION foo_3 VALUES LESS THAN ( TO_DAYS ( ' 2011-01-01 ' ))
)
```

删除一个分区，代码如下：

```
ALTER TABLE FOO DROP PARTITION foo_3;
```

检索information_schema数据库能看到刚刚创建的分区信息，检索方法如下：

```
SELECT * FROM PARTITIONS WHERE PARTITION_NAME IS NOT NULL
```

此时，打开MySQL数据目录（SHOW VARIABLES LIKE 'datadir'），如果MySQL配置设置in_nodb file per table为ON（由于前面定义的是InnoDB），则会发现：

```
foo#p#foo_1.ibd foo#p#foo_2.ibd
```

如果创建的是MyISAM表类型，会发现：

```
foo#P#foo_1.MYD  
foo#P#foo_1.MYI  
foo#P#foo_2.MYD  
foo#P#foo_2.MYI
```

由此可知，MySQL通过分区把数据保存到不同数据文件里，同时索引也是分区的。相对未分区的表来说，分区后单独的数据文件和索引文件的大小都明显降低，效率则明显提升。为了验证这一点，可做如下实验：

```
INSERT INTO 'foo' ('id', 'created') VALUES (1, '2008-01-02 00: 00: 00'), (2, '2009-01-02 00: 00: 00');
```

然后执行如下SQL语句：

```
EXPLAIN PARTITIONS SELECT*FROM foo WHERE created= '2008-01-02';
```

看到MySQL仅仅在foo_1分区执行这条查询。

理论上效率肯定会高一些，至于具体高多少，就要看数据量了。实际应用分区时，通过DATA-DIRECTORY和INDEX DIRECTORY选项把不同分区分散到不同磁盘上，从而进一步提高系统的I/O吞吐量。

提示 使用分区功能之后，相关查询最好都用EXPLAIN PARTITIONS过一遍，确认分区是否生效。

到底应该采用哪种分区类型呢？

通常使用Range类型，不过有些情况，比如说在主从结构中，主服务器由于很少使用SELECT查询，在主服务器上使用Range类型的分区通常并没有太大意义，此时使用Hash类型的分区相对更好，假设使用：

```
PARTITION BY HASH (id) PARTITIONS 10
```

当插入新数据时，会根据id把数据平均分散到各个分区上，由于文件小、效率高，更新操作会变得更快速。

到底应该按哪个字段分区呢？

通常情况下，按时间字段分区，不过具体情况还是应该按需求而定。划分应用的方式

有很多种，比如按时间或用户，哪种用得更多，就选哪种分区。如果使用主从结构可能更灵活些，有的“从服务器”使用时间分区，有的“从服务器”使用用户分区。不过如此一来，当执行查询时，程序里应该负责选择正确的“从服务器”查询，写个MySQL Proxy脚本应该可以透明实现。

分区虽然很好，但目前的实现还有很多限制，例如：

主键或者唯一索引必须包含分区字段，如PRIMARY KEY (id, created)，不过对InnoDB来说，大主键性能不好。

很多时候，使用分区就不要再使用主键，否则可能影响性能。

只能通过int类型的字段或者返回int类型的表达式来分区，通常使用YEAR或TO_DAYS等函数（MySQL5.6对这个限制开始放开）。

每个表最多1024个分区，不可能无限制扩展分区，而且过度使用分区往往会消耗大量系统内存。

采用分区的表不支持外键，相关的约束逻辑必须通过程序来实现。

分区后，可能会造成索引失效，需要验证分区可行性。

5.3.3 分表的应用

分表思想和分区类似，区别是：分区是把一个逻辑表文件分成了几个物理文件后进行存储，而分表则是把原先一个表分成几个表。进行分表查询时，可以union或者做一个视图。

分表又分垂直切分和水平切分，其中水平切分最常用。水平切分通常指切分到另外数据库或表中。例如，对于一个论坛的会员表，按对10的模进行切分：

```
table=uid%10
```

如果 $uid\%10=0$ ，则将用户数据放置到`uid_0`表中；如 $uid\%10=1$ ，则将用户数据放置到`uid_1`表中，依次类推。

有个问题，这个uid应该是所有会员按序增长的，可它是怎么得到的呢？使用自增键显然是不行的，这时就要用到序列了。

对于一些流量统计系统，其数据量比较大，并且对过往数据的关注度不高，这时按年、月、日进行分表，将每日统计信息放到一个以日期命名的表中；或者按照增量进行分表，如每个表100万条数据，超过100万的放入第二个表中。还可以按Hash进行分表，但是按日期和取模余数分表最常见，也较容易扩展。

分表后可能会遇到新的问题，那就是查询、分页和统计。通用的方法是在程序中进行处理，辅助视图。

使用分表的场景举例如下。

1.场景一

会员数据对5取模，放在5个数据表中，如何查询会员数据？

1) 已知uid查询会员数据，代码如下：

```
<? php
$member_table= ' member ' .uid%5;
$sql="select*from { $member_table } ";
//查询全部会员数据
$table=array ( ' member0 ', ' member1 ', ' member2 ', ' member3 ', ' member4 ');
foreach ( $table as $v ) {
    $sql.="select*from { $v } union";
}
$sql=substr ( $sql, 0, -5 );
```

这样就关联了所有的表。同理，分页的话在这个大集合上做limit即可。如果分表后的数量相对固定，也可以通过一个固定的视图完成。这里可能有人要问，这样做岂不是回到起点，又要关联所有数据？这和分表没差别啊。其实不是这样的，在实际应用中，不可能查看所有的会员资料，一次查看20个，然后分页。完全没有必要做union，仅查询一个表就够了，唯一需要考虑的是在分页临界点的衔接。其实，这个衔接是否那么重要呢？即使偶尔出现几条数据的差异，也不会对业务有任何影响。

2) 和其他表进行关联。这其实和1)类似，就不再赘述了。

3) 根据会员姓名搜索用户信息。在这种需求下，需要搜索所有的表，并对结果进行汇总。虽然这样做产生了多次查询，但并不代表效率就低。好的SQL语句执行10次也比差的SQL语句执行1次还快。

2.场景2

在一个流量监控系统中，由于网络流量巨大，统计数据很庞大，需要按天分表。现要得到任意日、周、月的数据。

1) 需要任意一天的数据。直接查询当天的数据表即可。

2) 需要几天的数据。分开查询这几天的数据，然后进行汇总。

3) 需要查询一周的数据。对一周数据定期汇总到一个week表，从此表查询数据。这个汇总过程可由一个外部程序完成，也可以是定期执行的脚本，或者一个数据库的JOB。

4) 查询一月的数据。汇总本月所有数据到month表，从此表查询。

5) 查询5个月内的详细数据。不支持。仅支持最多3个月内的详细数据。数据每3个月归档一次。在大数据的处理中，必须做出一些牺牲。对于超出3个月的数据，仅提供统计数据，详细数据需要查看归档。90天或者180天，给数据保存设个界限，也是大部分这类系统的常规做法，超出90天的数据就不再提供详单。比如，移动的通话记录最多保存半年，即180天，超出这个范围的数据不提供查询。如果你实在需要，可能就要联系移动的工程师了。

总结 分表前应尽量按照实际业务来分表，参考依据就是哪些字段在查询中起主要作用，那就按这些段来分表，并且需要在分表前就估计好规模，也就是先确定好规则再分表。

对于分表后的操作，依然是联合查询、视图等基本操作，或者使用Merge引擎合并数据，并在此表中查询。复杂一些的操作需要借助存储过程来完成，借助外部工具实现对分表的管理，但由于其适用性不强，不能得到广泛的推广。

对于较庞大的数据，不论是否进行分表，都必须考虑功能和效率的平衡性，并在功能上做出让步。我们不能事事迁就用户，而应该对某些影响效率的功能做出限制。如移动公司的180天限制、论坛禁止对老帖进行回复等。

5.4 MySQL的高级应用

本节将介绍一些MySQL中不常用但却能在关键时刻帮上大忙的应用。主要是MySQL一些高级特性，如序列、视图、存储过程等。

5.4.1 MySQL自增长序列

MySQL通过AUTO_INCREMENT自增长字段实现插入1条记录，自动增加1。而这个功能在Oracle中是通过序列（sequence）完成的。这样看，似乎MySQL自增长比Oracle序列的实现更方便。

首先要说明的是，MySQL自增长“序列”和序列是两回事，MySQL本身不提供序列机制。

MySQL使用AUTO_INCREMENT设置起始值，也能通过修改系统变量auto_increment_increment设置步长，这是一个全局设置。MySQL一个表只能有一个自增长字段。自增长只能分配给固定表的固定字段，不能被多个表共用，并且只能是数字型。另外，自增主键往往是没意义的。

在下列情况可能需要使用序列。

业务复杂，需要定制和控制主键时（自增主键只能是按数字递增的，但是序列可以随心所欲地变化，比如按照年、月、日生成主键）；

希望手工维护自增长，方便数据迁移时；

当事务跨多表，期望事务可靠性时；

需要一个业务上有意义的主键时，比如单据号等（若只是一个流水号，那么用自增长更方便）；

主键很明确地需要和其他表关联时；

期望主键是唯一的，不需要重复利用时。

当然，序列也有缺点，主要是程序处理麻烦。Oracle自增有主键的缓存与之对应，不用担心效率问题，而MySQL只能通过触发器模拟，会有一些性能损失。

下面我们来实现MySQL中的序列。首先建一个序列表，代码如下：

```
DROP TABLE IF EXISTS ' sequence ' ;
CREATE TABLE IF NOT EXISTS ' sequence ' (
  ' name ' varchar ( 50 ) NOT NULL ,
  ' current_value ' int ( 11 ) NOT NULL ,
  ' increment ' int ( 11 ) NOT NULL DEFAULT ' 1 '
) ENGINE=MyISAM DEFAULT CHARSET=utf8 CHECKSUM=1 DELAY _ KEY _ WRITE=1
ROW_FORMAT
=DYNAMIC COMMENT= ' 序列表，命名s_[table_name] ' ;
INSERT INTO ' sequence ' ( ' name ' , ' current_value ' , ' increment ' ) VALUES ( ' s_
blog_account ' , 0 , 1 )
```

name是序列名，current_value是序列当前值，increment是每次增长的步长。

再写几个函数（取当前值）：

```
DROP FUNCTION IF EXISTS ' currval ' ;
DELIMITER//
CREATE FUNCTION ' currval ' ( seq_name VARCHAR ( 50 )) RETURNS int ( 11 )
READS SQL DATA
DETERMINISTIC BEGIN
DECLARE VALUE INTEGER;
SET VALUE=0;
SELECT current_value INTO VALUE FROM sequence WHERE NAME=seq_name;
RETURN VALUE;
END//
DELIMITER;
```

取下一个值：

```
DROP FUNCTION IF EXISTS ' nextval ' ;
DELIMITER//
CREATE FUNCTION ' nextval ' ( seq_name VARCHAR ( 50 )) RETURNS int ( 11 )
DETERMINISTIC BEGIN
UPDATE sequence SET current _ value=current _ value +increment WHERE NAME=seq _
name;
RETURN currval ( seq_name );
END//
DELIMITER;
```

需要插入时，只需加下面一句，用nextval函数的返回值作为主键插入到对应的表中：

```
select nextval ( "s_blog_account" )
```

当对数据进行了归档和删除后，更改sequence表对应的序列值，让其重新从零开始记录。而自增主键是不能实现归零的，MySQL会记下每次操作前的自增主键，即使删除了记录，新插入的数据自增主键也会紧跟在删除前的数据之后，这样，频繁更新的数据表的自增主键显得比较离散，不连续。

前面曾经提到过使用between优化limit，但是对于主键不连续的记录就没办法了。而sequence不会存在这样的问题，即使存在，也可以进行更新。

比较MySQL中自增主键和序列之间的异同以及优缺点，似乎感觉在MySQL中使用自增主键更方便。但这里仍然推荐使用序列，因为其有更高的定制性和可控性。并且，当转用其他数据库进行开发时，也能更快地熟悉一些特性。

不过，在Log表和无意义字段中仍推荐使用自增，因为这些字段实在没有任何意义需要对其进行标记，这里的主键仅仅是为了做索引。需要注意的是，不要在一个表的字段上同时使用自增和“序列”。

对于InnoDB引擎，虽然序列的可定制性很强，但是如果使用序列，则不推荐使用char等非number型数据做主键，因为这样会明显影响InnoDB的插入性能。应该保证序列是有序的number.

5.4.2 MySQL视图

视图是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。但是，在数据库中视图并不以存储的数据值集形式存在（这里仅仅是针对MySQL数据库而言的，在Oracle中存在“物化视图”，视图本身还能存储数据）。行和列数据来自自定义视图的查询所引用的表，并且在引用视图时动态生成。

对其中所引用的基础表来说，视图的作用类似于筛选，筛选当前或其他数据库的一个或多个表，或者其他视图。通过视图进行查询没有任何限制，进行数据修改时的限制也很少。

创建视图使用CREATE VIEW，代码如下：

```
CREATE[OR REPLACE][ALGORITHM= { UNDEFINED | MERGE | TEMPTABLE } ]
VIEW[db_name.]view_name[ ( column_list ) ]
AS select_statement
[WITH[CASCADED | LOCAL]CHECK OPTION]
```

该语句若给定[OR REPLACE]，表示当已有同名的视图时，将复盖原视图。使用select_statement语句可从表或其他视图中查询。视图属于数据库，因此需要指定数据库的名称，若未指定，表示在当前的数据库创建新视图。

表和数据库共享数据库中相同的名称空间，因此，数据库不能包含相同名称的表和视图，并且，视图的列名也不能重复。举个例子，先建立好友表：

```
CREATETABLE ' e_friend ' (
  ' uid ' INT ( 10 ) UNSIGNEDNOTNULLCOMMENT ' 用户ID ',
  ' fri_uid ' INT ( 11 ) NOTNULLCOMMENT ' 好友的用户ID ',
  ' fri_nickname ' VARCHAR ( 64 ) NULLDEFAULTNULLCOLLATE ' gbk_bin ',
  ' fri_group_id ' MEDIUMINT ( 9 ) NULLDEFAULTNULLCOMMENT ' 好友分组 ',
  PRIMARYKEY ( ' uid ', ' fri_uid '),
  INDEX ' idx_friend_fri_group_id ' ( ' fri_group_id ')
```

```
)  
COMMENT= '好友表 (一对好友关系2条记录) '  
COLLATE= 'gbk_bin '  
ENGINE=InnoDB;
```

接下来建立消息表，此表存放用户发布的消息：

```
CREATETABLE 'e_feed' (  
  'feed_id' INT (10) UNSIGNEDNOTNULLCOMMENT '动态ID',  
  'uid' INT (10) UNSIGNEDNULLDEFAULTNULLCOMMENT '用户ID',  
  'feed_data' VARCHAR (1024) NULLDEFAULTNULLCOMMENT '动态数据内容' COLLATE  
'gbk_bin',  
  PRIMARYKEY ('feed_id'),  
)  
COMMENT= '动态 '  
COLLATE= 'gbk_bin '  
ENGINE=InnoDB  
CHECKSUM=1;
```

现在要查询某用户的好友发布的全部消息，通常这样写：

```
select u.uid, u.fri_uid, u.fri_nickname, f.feed_id, f.feed_data from e_friend u  
left join e_feed f  
on u.fri_uid=f.uid  
and u.uid=1
```

如果采用视图，就这样写：

```
create view freind_feed as select u.uid, u.fri_uid, u.fri_nickname, f.feed_id, f.feed_data  
from e_friend u  
left join e_feed f
```

```
on u.u.fri_uid=f.uid
```

如果要查询某会员好友的消息记录，直接从此视图表中查询：

```
select*from friend_feed where uid=1
```

从视觉上看，这样的语句更简洁。视图实际上等价于一条SQL语句，只是把一连串的SQL语句缩写。实际上在MySQL中，视图等价于依据SQL查询语句，MySQL在处理视图查询时，只是把视图展开成其定义的语句，并在其上查询。因此，MySQL视图不可能提高查询效率，其只是到另一个查询语句的映射。

MySQL里的视图只是一个虚拟表，只包含定义，而不含有任何数据。

前面一直强调报表的应用，在报表中，视图的应用是最普遍的。报表涉及的SQL语句比较复杂，能用一个简单的视图查询来替代冗长的SQL。如果是Oracle，还可以在视图中存放实际的数据，加快查询速度。

创建视图的注意事项如下：

SELECT语句不能包含FROM子句中的子查询。

SELECT语句不能引用系统或用户变量。

SELECT语句不能引用预处理语句参数。

在存储子程序内，定义不能引用子程序参数或局部变量。

在定义中引用的表或视图必须存在。但是，创建视图后，能够舍弃定义引用的表或视图。要想检查视图定义是否存在这类问题，可使用CHECK TABLE语句。

在定义中不能引用TEMPORARY表，不能创建TEMPORARY视图。

在视图定义中命名的表必须已存在。

不能将触发程序与视图关联在一起。

在视图定义中允许使用ORDER BY，但是，如果从特定视图进行了选择，而该视图又使用了具有自己ORDER BY的语句，它将被忽略。

不能给视图添加索引。

视图通常是不允许更新的，能更新的视图比较少，条件也比较苛刻。

只要SQL语句中含有下面任何一个子句，都会限制视图的更新：

聚合函数（SUM（）、MIN（）、MAX（）、COUNT（）等）。

DISTINCT.

GROUP BY.

HAVING.

UNION或UNION ALL。

位于选择列表中的子查询。

Join.

FROM子句中的不可更新视图。

WHERE子句中的子查询，引用FROM子句中的表。

仅引用文字值（在该情况下，没有要更新的基本表）。

ALGORITHM=TEMPTABLE（使用临时表总会使视图成为不可更新的）。

关于可插入性（可用INSERT语句更新），如果满足关于视图列的下述额外要求，可更新的视图也是可插入的：

不得有重复的视图列名称。

视图必须包含没有默认值的基表中的所有列。

视图列必须是简单的列引用而不是导出列。

其实以上要求综合起来就是一句话：视图的数据和基本表的数据不一样。因此视图一般用于查询的目的。那么使用视图有什么好处呢？

首先，实现了更细致的权限控制。比如你查询学生表，允许你知道学生姓名，但不允许看到用户的联系方式。这时就可以定义一个视图，仅包含学生姓名，并赋予使用者此视图的权限。这样就屏蔽了使用者对基础表的使用权限，仅暴露给其需要的字段。虽然给表字段分配用户权限也能达到同样的效果，但是如果涉及多表关联时，远没有视图灵活和直观。

其次，把业务中常用的SQL语句用一个视图来表示，可使查询语句更直观。

最后，性能优势虽在MySQL中不存在，但在Oracle等商业数据库中，视图支持更多的特性，如物化视图。

5.4.3 MySQL存储过程和事件调度

SQL语言首先是一门编程语言，其次才是查询语言。利用SQL语言完成复杂的数据库层业务操作，这往往通过编写存储过程来实现。在一些应用中，数据库编程占整个应用的大部分，其编程能力起决定性作用。MySQL 5支持存储过程、函数、视图、JOB等数据库应有的基本特性。

Web应用中经常会要求某个任务常驻内存或者定期执行。我们知道，如果PHP运行在CLI下，是不限时的，所以用来做计划任务。也就是在未来某个时间自动执行某个任务。网页的话，修改程序运行的最大时间，就能无限期执行下去。但是网页请求有可能在执行的过程中被中断，导致客户端和服务端（如Apache）的交互中断，从而使Apache放弃执行PHP脚本。这时，通过如下脚本实现脚本的不中断执行：

```
<? php
ignore_user_abort ( true );
set_time_limit ( 0 );
if ( date ( "m" ) %5==0 ) {
//do something }
```

加入`ignore_user_abort (true)`可以保证用户在关闭网页后，程序在后台执行而不会立即中断。计划任务由用户或搜索引擎的蜘蛛触发，也可以对某些任务采取伪任务的方式（即若任务没有被外界触发，那么在后台查看任务时，先标记任务为已执行，然后再触发，慢慢地去执行。因为很多任务的实时性并不高）。如果是UNIX系操作系统，则用Crontab执行。

如果涉及单纯的数据库操作，由MySQL 5的event定时触发。这样比用PHP更直接。既然是数据库操作，为什么不直接用MySQL呢？何必再多此一举？下面举个例子，同时介绍MySQL的存储过程和定时器。

比如，现在要定期删除处理过的日志，而这些日志存储在数据库里，表结构如下：

```
CREATE TABLE 'log' (
```

```
' id ' INT ( 10 ) NOTNULLAUTO _ INCREMENT ,
' message ' VARCHAR ( 200 ) NULLDEFAULT ' 0 ' ,
' isread ' INT ( 11 ) NULLDEFAULT ' 0 ' ,
PRIMARYKEY ( ' id ' )
)
COLLATE= ' utf8 _general _ci '
```

现在需要定期删除看过的日志文件，其存储过程如下：

```
DELIMITER//
CREATE _ PROCEDURE ' utable ' ( IN ' $tname ' VARCHAR ( 20 ) , IN $fieldvarchar ( 10 ) )
BEGIN
SET @sqlcmd=CONCAT ( "delete from", $tname, "where", $field, "=1" );
PREPARE stmt FROM @sqlcmd;
EXECUTE stmt;
DEALLOCATEPREPARE stmt;
END
//
```

需要注意的是，存储过程支持IF……ELSE、循环等基本语法，但和编程语言的语法有所区别。创建一个存储过程有固定的格式和套路。通常分以下几步。

1) 确定输入/输出参数和类型。如上述过程中的代码：

```
IN ' $tname ' VARCHAR ( 20 )
```

其中，IN表示输入参数，tname是参数名，VARCHAR (20) 是参数类型。

2) 定义变量和赋值。使用declare关键字定义变量，set关键字用来赋值，如：

```
declare a float;
```

```
set a=0;
```

declare语法比较简单易懂，这里就不介绍了。在set赋值语句中可以使用SQL中的函数以及SELECT语句，如：

```
declare a float; set a=rand ();
```

如果set的变量前有@符号，表示这是一个会话变量。否则，只是一个局部变量。

3) 过程的主体部分。可以是各种运算，也可以是数据库操作。

4) 程序的返回值。可以有返回值，也可以无返回值。

看一个有返回值的简单例子：

```
CREATE DEFINER=' root ' @ ' localhost ' PROCEDURE ' proc _ test ' ( )  
LANGUAGE SQL  
NOT DETERMINISTIC CONTAINS SQL  
SQL SECURITY DEFINER  
COMMENT ' '  
begin  
declare a float;  
set a=rand ();  
select a;  
end
```

我们定义的存储过程的调用如下：

```
CALL ' utable ' ( ' log ', ' isread ' )
```

基于此，创建一个事件：

```
CREATE EVENT IF NOT EXISTS event_log ON SCHEDULE EVERY 100 SECOND
ONCOMPLETION PRESERVE
DOCALL 'utable' ('log', 'isread');
```

EVENT就是我们常说的JOB，即调度，表示定期执行某个操作。上面event_log的含义是每100秒执行一次，调用utable存储过程。当然，这里可根据需求定制执行间隔。事件一旦创建就会自动执行。假设事先在log表里插入一些数据，过100秒后看看是不是符合条件的数据都被删除了？关闭event语句，方法如下：

```
ALTER EVENT event_log ON COMPLETION PRESERVE DISABLE;
```

我们可能会想，没必要这么麻烦吧？使用PHP很简单啊。但是对于更精确的需求，单纯依靠PHP脚本不能保证精确定时执行，效率也不够好。其实存储过程主要用于队列服务。在用MySQL模拟队列的时候，就需要用到存储过程、定时器等工具。而这在SNS、微博等应用中很常见，也是最简单的方式。

如果创建的event执行不了，可能是因为MySQL没有打开event功能，此功能默认是关闭的。查看是否打开的方法如下：

```
SHOW VARIABLES LIKE '%sche%';
```

打开event功能的方法如下：

```
SET GLOBAL event_scheduler=1;
```

上面只简单介绍了存储过程和调度。在实际的业务中，存储过程得到了大量的应用，特别是Oracle这类的商业数据库，很多业务逻辑都由存储过程完成。因此，存储过程能提升效率，且还具有模块化、易于复用、可以移植等特点。

但是MySQL的存储过程还存在两个问题：一是不像Oracle那样有强大而且成熟的调试工具，只能靠经验和一些软件辅助；二是MySQL的编程能力远弱于Oracle等商业数据库。但是作为一款开源、免费的数据库产品，它提供的功能已经足够强大了。

5.4.4 用MySQL模拟消息队列

队列（queue）和栈一样，是一种线性表结构，不过队列是一种先进先出（FIFO）的数据结构。队列只允许在后端（rear）进行插入操作，在前端（front）进行删除操作。

最近一两年，团购、秒杀等购物应用异常火爆，既然购物，就涉及数据库操作。那么这么大的并发是怎么处理的？不会把数据库压垮吗？答案是会的。一个解决办法就是把HTTP请求放入内存中的高速队列，然后对队列里的数据按一定的规则进行分流处理，这就是HTTP请求队列。

比如，微博和SNS通常拥有上亿的受众数量。一个明星或公众人物可能有几千万的粉丝。如若一个公众人物发了条微博，那么就得推送到所有关注者那里（有推策略、拉策略、随机策略、优先策略等）。这时就必须用到队列。

由上面的应用实例可知，面对大数据量和高并发的Web应用，队列工具可以大展拳脚。

下面以SNS中的动态分发的设计过程为例，讲解一个简单的队列在消息调度机制中的实现和使用。

当某个人发布一条动态的时候，首先给把这条动态插入feed表，然后判断当前用户组，如果是普通好友，则给所有好友发布广播，往feedbroadcast表写入冗余数据。如果是公共主页，则给好友和关注者发送广播。可是这样会存在一个问题，这种频繁不间断的读写数据库，会给服务器造成很大压力。此时我们注意到，好友动态本身不必具有十分高的实时要求。所以，这里采用异步的推，把动态推给好友，而不是好友主动拉取。

随着网站用户数量的增加，若在前端页面直接写入大量数据，会延长用户的等待时间，在多并发的情况下，页面效率低、压力集中。为了解决上述问题，考虑使用数据的异步处理。而消息队列的背后实质就是一种“异步处理”的思想。

“消息队列”是在消息的传输过程中保存消息的容器。消息队列管理器在将消息从它的源中继到它的目标时充当中间人的角色。队列主要提供路由并保证消息的传递；如果发送消息时接收者不可用，消息队列会保留消息，直到成功地传递。利用消息队列可以很好地异步处理数据传送和存储，当频繁地向数据库中插入数据时，就可采取消息队列异步插

入。另外，可将较慢的处理逻辑、有并发数量限制的处理逻辑，通过消息队列放在后台处理，例如视频转换、发送手机短信等。

消息发送很简单，在动态产生之后，直接在DATABASE中插入一条记录即可。这里，完全依赖数据库来模拟实现消息队列。

消息的接收过程是通过定时的轮询机制，消息接收者从消息队列表中获取对应topic类型的“未处理”消息。

接收者从已获取的消息列表中取出下一条消息，进行业务逻辑处理，成功后将该消息标记为“已处理”。通过这种简单的ACK机制保证消息不会丢失或者遗漏。

注意 在队列中存储的是消息，而不是实际要分发和处理的数据。秒杀队列中存储的仅仅是一个HTTP请求，SNS队列中存储的仅仅是一条制造出来的动态，而不是所有要分发的动态。动态的分发不是消息队列所负责的，其由另外一个程序处理。

用数据库模拟消息队列。新建一个消息对列表，其结构如下：

```
DROP TABLE IF EXISTS ' eventqueue ' ;
CREATE TABLE IF NOT EXISTS ' eventqueue ' (
  ' qid ' int ( 11 ) NOTNULLAUTO _ INCREMENTCOMMENT ' 消息序列 ' ,
  ' topic ' tinyint ( 4 ) NOTNULLCOMMENT ' 应用类别 ' ,
  ' status ' tinyint ( 4 ) NOTNULLCOMMENT ' 状态, 标识是否进行分发, 0未分发, 1已分发 ' ,
  ' data ' varchar ( 1024 ) COLLATEgbk _ binDEFAULTNULLCOMMENT ' 消息内容 ' ,
  ' uid ' int ( 11 ) NOTNULLCOMMENT ' 动态产生者的uid ' ,
  ' create _ date ' timestamp NOT NULL DEFAULT CURRENT _ TIMESTAMP ON UPDATE
CURRENT _ TIMESTAMP COMMENT ' 发送时间 ' ,
  PRIMARYKEY ( ' qid ' )) ENGINE=MEMORY DEFAULT CHARSET=gbk COLLATE=gbk _ bin
COMMENT= ' 消息
队列表, 用以记录消息的属性及内容 '
```

上述结构中存放消息的产生者的是uid，而不是具体的动态内容。Data字段是消息的内容，并非动态的内容。动态的内容存储在feed表中。

消息队列有了，还需要一个对消息队列进行轮询调度的程序。对此，在数据库中建三个存储过程和一个定时器来实现模拟，具体说明如表5-2所示。

存储过程名	作 用	说 明
Proc_msg_receiver_friend	接收并处理好友的动态分发	从消息队列中获取消息 业务处理：给好友发动态
Proc_msg_receiver_tip	接收并处理公共主页的动态分发	从消息队列中获取消息 业务处理：给关注者发消息
msg_scheduler	调度程序	根据服务器访问的高峰期不同，调度消息接收者从消息队列中获取消息
Proc_msg_cleaner	消息删除程序	定期删除消息队列中已经接收的数据

下面来看其中一个消息分发过程，这是通过一个存储过程实现的，代码如下：

```

DROP PROCEDURE IF EXISTS ' proc_msg_receiver_friend ' ;
DELIMITER//
CREATE DEFINER=' TEST ' @ ' % ' PROCEDURE ' proc_msg_receiver_friend ' ( )
COMMENT ' 消息队列中好友分发存储过程 '
BEGIN
DECLARE CONTINUE HANDLER FOR SQL EXCEPTION ROLLBACK;
START TRANSACTION;
——好友动态分发
INSERT INTO feed_broadcast
SELECT feed_id, ef.fri_uid, temp.create_date, app_id, src_type
FROM ( SELECT * FROM eventqueue eq WHERE eq.status=0 AND eq.topic=1
ORDER BY create_date DESC LIMIT 100 ) temp, friend ef WHERE temp.uid=ef.uid;
——更改动态消息状态
UPDATE eventqueue eq SET status=1
WHERE eq.status=0 AND eq.topic=1
ORDER BY create_date DESC LIMIT 100;
COMMIT;
END//
DELIMITER;

```

有了消息和消息的分发，接下来需要通过调度决定分发的策略、何时分发和频率如何。利用MySQL的event实现调度。在event里设定消息调度的频率，比如每5分钟调度一次存储过程，对消息队列里的数据进行分发。下面是消息清除器的event：

```
DROP EVENT IF EXISTS ' event_msg_cleaner ' ;
DELIMITER//
CREATE EVENT ' event_msg_cleaner ' ONSCHEDULEEVERY 1 HOURSTARTS ' 2011-08-
01 00: 00: 00 ' ON COMPLETION NOT PRESERVE ENABLE COMMENT ' 删除已处理消息调度 '
DO DELETE FROM ens_eventqueue WHERE STATUS=1//
DELIMITER;
```

看到这里，你会觉得这个思路其实和常规实现没多大区别，只不过把页面上的拉取操作变为服务器上的推操作，还麻烦了很多。但是进一步想一下，通过定制的调度把原先一瞬间可能发生的成千上万条动态分发的insert操作延迟了，并且是定制分量，把高峰期的负载分一部分到低谷时间段进行处理，这样能减少很多负担。使用MySQL的表来存储和处理消息队列确实不够快并且会造成数据库的压力，但我们可以把消息放到以快著称的内存缓存中，如Memcached。如果再配合上作业管理系统，这个分发机制的策略还能更复杂，甚至可进行优先级管理。

在这里，队列的“异步操作”特点就体现出来了，对于VIP用户或者活跃用户优先分发，对“垃圾”用户延迟分发或者不分发。比如某公共主页有100万粉丝，假如规定1个月内没有登录或者每个月登录次数少于3次的用户属于“僵尸”用户，在进行动态分发的时候故意漏掉这些用户。通过指定消息的优先级就能分清轻重缓急，合理分摊服务器负担。大致流程如下。

1) 存储过程Proc_msg_receiver_friend从eventqueue表选择未处理的状态，联合friend表的接收者，对发送者的每个好友插入冗余表feed_broadcast，然后处理eventqueue消息队列表，更改动态消息状态为已分发。

2) 调用event_msg_cleaner事件每1小时执行一次，删除队列表中已处理的消息调度。

3) 使用作业管理系统定时调用存储过程来分发。

4) 动态不是实时的，而是由服务器决定是否分发给接收者，动态全部进入消息队列，由调度程序决定是否分发和处理。

队列的异步操作的优点：部署简单，仅使用MySQL的内置存储过程和定时器即可完

成大部分调度，服务器部署也很简单，容易迁移。

队列的异步操作的缺点：对于大数据量、上千万的消息队列，即使用数据库模拟消息队列也力不从心。另外这种方法实时性不高，对于动态分发、邮件、小纸条、短信等实时性相对略低的应用，可以采用这种方案进行处理，但是不能用于对实时性要求很高的场所。

提示 当消息很大时，可把消息放入内存，或者进行集群分发。

实际上，消息队列的应用场景取决于消息的重要性。从消息的重要性角度讲，动态属于纯娱乐性质的消息，本质上是一种垃圾信息。对于一条动态，可以分发或者不分发，动态的受体不会去关注自己是否收到了所有关注者的动态，对于漏掉的动态也没有感知。

做个试验，用自己的微博加2000个关注者，假设这2000个关注者都是微博控，那么在1小时内将可能产生至少10000条动态。过1小时后再看微博页面，你根本不会收到10000条动态，通常只收到最近的几十条动态，其他过期的、非活跃用户发布的动态被丢弃了，没有分发。但是用户发给自己的动态是要保留的，这也是前面的分发程序中所考虑到的。

如果是重要的消息，消息队列就必须一条一条的分发，不允许丢弃。

例如，淘宝秒杀和上述消息队列本质上是一样的，但是对实时性要求更高，直接在HTTP层甚至TCP/IP层进行队列处理能达到很高的效率。

关于HTTP队列，金山公司内部有一个开源作品HTTPSQL，其基于Tokyo Cabinet的B+tree Key/Value数据库做数据的持久化存储。这个作品正是把这里的队列从MySQL移到内存数据库中，并且由一个专门的进程负责队列的操作，客户端则通过Socket请求进行读写。

在这里，HTTPSQL和消息队列的实现思路是一样的。只不过结合了NoSQL，能得到更快的速度，并支撑更高的并发量。在对并发和数据量要求苛刻的情况下，就需要自己实现一个轻量级的基于内存的消息队列。对此，可以选用不少开源的消息队列产品。比如ZeroMQ就是一个轻量级异步消息队列，提供了一系列的API来实现消息的传输，并且提供了C/C++、PHP、Java、Python等语言的接口。腾讯曾在自己的产品中使用过

ZeroMQ。此外，JMS也是一个简单快速、功能强大的消息队列。

5.4.5 SQL注入漏洞与防范

MySQL注入又称SQL Injection，通过构造特定的SQL语句获取权限外的数据。随着程序员安全意识的提高，如今SQL注入攻击得逞的几率越来越小。“知己知彼，百战不殆”，本节以MySQL为例，介绍SQL注入原理，以帮助开发者应对这种攻击。本节的知识不作为读者进行攻击的参考。

SQL注入的原理其实很简单，就是在原有SQL语句上附加一段SQL代码，构造特殊的SQL语句，利用应用程序自身的权限实现所需要的操作。为了便于理解，先建一个用户表，方法如下：

```
CREATE TABLE 'user' (  
  'uid' INT (10) NULL AUTO_INCREMENT,  
  'username' CHAR (50) NULL DEFAULT '0',  
  'pwd' CHAR (16) NULL DEFAULT '0',  
  PRIMARY KEY ('uid')  
)  
COLLATE='utf8_general_ci' ENGINE=InnoDB;  
insert into user (username,pwd) values ('admin','admin888');  
INSERT INTO 'user' ('username','pwd') VALUES ('custom','123456');
```

然后自行插入几条数据以便测试。作为登录应用，在程序里一般执行如下SQL语句判断账号密码是否匹配。用PHP代码表示如下（为了突出重点，这里采用最简单的实现方法，简化了登录模型）：

```
<? php  
$host="localhost";  
$username="root";  
$dbpwd="123";  
$dbname="test";  
$conn=mysql_connect ($host,$username,$dbpwd) or die ("#fail to connect to db.");  
mysql_select_db ($dbname,$conn) or die ("#2: failed to open database! ");
```

```
$user=$_GET['user'];
$pwd=$_GET['pwd'];
$sql="SELECT*FROM user WHERE username= '{ $user } ' AND pwd= '{ $pwd } ' ";
$result=mysql_query ($sql,$conn);
$num_rows=mysql_num_rows ($result);
if ($num_rows<1)
{
echo"fail";
} else {
echo"succes";
}
while ($row=mysql_fetch_array ($result)) {
var_dump ($row);
}
echo"<code>SQL: $sql<code>";
```

在浏览器里执行以下请求:

<http://127.1/t/inject.php?user=admin&pwd=admin888>

显然这是一条正确的语句, 其相当于:

```
SELECT*FROM user WHERE username= ' admin ' AND pwd= ' admin888 '
```

上述两条语句将得到正确的执行结果。

对于如下请求:

<http://127.1/t/inject.php?user=admin&pwd=admin>

由于密码和用户名不匹配，将得不到任何查询结果集。通过MySQL客户端也可以得到验证：

```
Mysql> SELECT * FROM user WHERE username= ' admin ' AND pwd= ' admin '
->;
Empty set ( 0.00 sec )
```

在MySQL客户端构造特别的查询语句可以绕过这个验证，如下：

```
SELECT * FROM user WHERE username= ' admin ' AND pwd= ' admin ' or 1=1
Mysql> SELECT * FROM user WHERE username= ' admin ' AND pwd= ' admin ' or 1=1;
+----+----+----+
| uid | username | pwd |
+----+----+----+
| 1 | admin | admin888 |
| 2 | custom | 123456 |
+----+----+----+
2 rows in set ( 0.00 sec )
```

很明显，由于or是一个可选性分支，1=1恒成立，所以SQL语句得以执行。如果是在客户端呢？试着如此构造查询条件：

```
http: //127.1/t/inject.php? user=admin&pwd=admin%20or%201=1
```

上述语句没有任何输出结果，通过页面打印看到最终构造的SQL语句如下：

```
SELECT * FROM user WHERE username= ' admin ' AND pwd= ' admin or 1=1 '
```

由于引号导致SQL构造不正确，需要绕过引号，让引号闭合，使最终结果和在客户端输入一致。

对比MySQL客户端的语句，执行如下请求：

http://127.1/t/inject.php? user=admin&pwd=admin or ' 1=1

成功了！程序运行结果如图5-4所示。



图 5-4 程序运行结果

至此，一个最简单的MySQL注入就成功了。不知道管理员密码就能登录其账号。

既然MySQL注入这么好用，还可以做些什么呢？现在利用这个漏洞获取另外一个表的数据。我系统里还有一个path表，其结构如下：

```
CREATE TABLE ' path ' (  
  ' id ' INT ( 10 ) NOT NULL DEFAULT ' 0 ' ,  
  ' type ' CHAR ( 50 ) NULL DEFAULT NULL ,  
  PRIMARY KEY ( ' id ' ),  
  INDEX ' id ' ( ' id ' )  
 )  
COLLATE= ' utf8 _general _ci '  
ENGINE=InnoDB;
```

用union联合查询如下：

```
Mysql> SELECT*FROM user WHERE username= ' admin ' AND pwd= ' admin ' union selec-  
t*from path;
```

报错:

```
ERROR 1222 ( 21000 ): The used SELECT statements have a-different number of columns
```

原因是union关联的表字段数量必须相等，user表是三个字段，而path表只有两个字段，无法union，修改一下即可，如下所示：

```
Mysql > SELECT*FROM user WHERE username= ' admin ' AND pwd= ' admin ' union  
select*, 1 from path; +----+----+----+  
| uid | username | pwd | |  
+----+----+----+  
| 1 | 1, 2, 8 | 1 | |  
| 2 | 2, 9, 10 | 1 | |  
| 3 | 1, 9, 10 | 1 | |  
| 4 | 9, 12 | 1 | |  
| 5 | 2, 9 | 1 | |  
| 6 | 9, 2 | 1 | |  
| 7 | 9, 82 | 1 | |  
| 8 | 4, 9, 2, 7 | 1 | |  
+----+----+----+8 rows in set ( 0.00 sec )
```

上述做法的原理就在于SELECT一个常量，手动为关联查询增加一个虚拟字段，现在它们的字段数相等，得到正确的查询结果。如果不知道user表和path表有多少个字段，需要尝试：

```
SELECT*FROM user WHERE username= ' admin ' AND pwd= ' admin ' union select 1, 1, 1  
from path;
```

手动创建三个虚拟的字段，和union前面部分的字段数一样就能查询了。这样查询到的结果是没有意义的，但是却根据这个猜到了path表的字段数，这样就能构造联合查询了。实际上，准确的说法是要猜到union前半部分的字段数，由于上面的查询使用SELECT*FROM user，即查出全部三个字段，如果是SELECT username, pwd FROM user，只查询两个字段，那么union后面一半的结果集也只应该有两列。

如果不知道数据类型和字段数量，用1充当一个字段，只要改变数量尝试，一定会猜到。如果是开源代码，预先知道表结构，那么省略尝试这一步。

下面给出在网页中应该构造的请求：

```
http: //127.1/t/inject.php? user=admin&pwd=admin ' union select*, 1 from path where ' 1=1
```

这样，只要找到一个注入点，就可以查询任何一个表的内容了。

这种构造SQL实现特殊目的应用，只是MySQL注入攻击的一部分，构造SQL实现攻击的例子还有很多，如搜索页面的模糊匹配，如果构造LIKE ' %%' 这样的查询条件，虽然不会造成隐私泄露等安全问题，但显然会加重MySQL服务器的负担，这也算SQL注入攻击的一种。

除了构造特殊的SELECT语句外，构造insert、delete等语句危害更大。在实际应用中，很难找到这么明显的注入点，同时需要配合其他手段。本节只介绍基本原理，不会过多涉及攻击方面的内容，有兴趣的读者自行查阅相关资料。

最后，看PHP一句话木马注入MySQL数据库的例子，如图5-5所示。

实际上，SQL注入技术含量并不高。那怎么防范呢？其实很简单

1) 如果是整型变量或字段，使用intval () 函数把所有传入的参数转化为一个数值。比如翻页、按ID浏览文章等。

2) 对于字符型变量，用addslashes () 会把所有 ' (单引号)、" (双引号)、\ (反斜线) 和空字符转为含有反斜线的溢出字符。或者使用PDO的参数绑定来提高安全性。

3) 转义或过滤一些特殊字符，如%等。

4) 保护表结构等关键信息 (对于开源程序此条不成立，这仅是无奈之举。例如有人会故意把字段名命名搞得很古怪，这是不可取的，最可靠的还是在代码级别上把好关)。

5) 任何情况下都要做好数据备份，以防万一。



图 5-5 SQL注入

5.5 本章小结

本章笼统地讲了一些MySQL的基本概念，这些也是最常用的概念。PDO类似ODBC、JD BC，通过PDO的参数绑定可以极大地提高应用的安全性。在性能要求不是很苛刻的环境下，使用PDO带来的收益会大于其损失。本章着重讨论了MySQL的优化，最后介绍了MySQL里的一些高级应用，如事件调度、视图和注入防范等。

关于MySQL引擎的底层和原理、分布式架构和部署方案、数据恢复和备份迁移、锁机制和监控等更深入的知识，后面的章节会逐一介绍。

第6章 PHP模板引擎的原理与实践

模板引擎思想来自经典的MVC（Model View Controller）模型，即模型层-视图层-控制器模型。MVC本来存在于桌面程序中，M指数据模型，V指用户界面，C则指控制器。使用MVC的目的是将M和V的实现代码分离，从而使同一个程序可以使用不同的表现形式。

随着Web的流行，这一模型被引入Web开发中。此时，V即视图层，也就是通常所说的模板。视图层实现了数据生成和数据展示的分隔。早期的视图通常由HTML元素控制界面，随着互联网的发展，一些新的表现层技术（比如Flex等）流行起来，MVC模型使得数据和表现得到分离，一套数据可以用于多种表现层而不用修改逻辑层的代码。比如，可以在模型层生成新闻列表的数据，然后分别在传统的Web网页、RIA应用、手机应用中用不同表现层技术来展示数据，而不用修改逻辑层代码，让前端和后端分离。

同时，AJAX应用的流行、jQuery库的普及，使早期HTML、JavaScript、PHP代码混写的情况得到改观。AJAX技术的应用使数据的请求和生成、展示实现了分离，促进了表现层和代码层的分离。

模板引擎作为视图层和模型层分离的一种解决方案，让前后端更好地分工协作。PHP开发经历了前后端混编，到极力推崇模板引擎（以Smarty为代表），再到现在的回归自然、质疑PHP模板引擎存在的必要性这几个发展阶段。

PHP中到底有没有必要使用模板引擎呢？模板引擎的原理是什么呢？为什么Smarty会那么流行，而如今在PHP的社区一提到Smarty就会引起争论呢？从受宠到失宠，这期间都发生了什么？这些都是本章要探讨的话题。

6.1 代码分层的思想

PHP作为轻巧灵活的脚本语言，非常适合Web开发这种开发周期短、需求变化快、强调用户体验的需求和业务。PHP早期开发，通常都是PHP代码与HTML混写，代码中充斥着数据库操作、逻辑判断、HTML代码生成，甚至JavaScript代码等。当项目规模

不大时，这种编程风格能够很快速地完成需求。但是一旦项目规模扩大后，再加上前端逻辑越来越复杂，这种完全混写的方式必将带来代码可读性差和后期维护困难等问题。

从前，我们可能是这么写代码，如代码清单6-1所示。

代码清单6-1 PHP和HTML混写示例

```
<html>
<head>
<meta http-equiv="content-type"content="text/html; charset=utf-8">
<title>demo.php</title> </head>
<body>
会员信息
<table>
<tr>
<td>ID</td>
<td>姓名</td>
</tr>
<? php
mysql_connect ("localhost", "root", "123") or die ("Could not connect: ".mysql_error ());
mysql_select_db ("test");
$result=mysql_query ("SELECT uid, username FROM user");
while ($row=mysql_fetch_array ($result)) {
echo"<tr> <td>". $row[ ' uid ' ]."</td> <td>". $row[ ' username ' ]."</td> </tr>";
}
mysql_free_result ($result);
? >
</table>
<? php
//其他模块
? >
</body> </html>
```

需要想办法实现数据生成和显示的分离，此时通常会把和数据库打交道的部分放到一个文件中，显示数据的部分放到另外一个文件中。新建文件u data.php并获取数据，如

代码清单6-2所示。

代码清单6-2 数据获取u data.php

```
<? php
mysql_connect ( "localhost", "root", "123" ) or die ( mysql_error ( ) );
mysql_select_db ( "test" );
$result=mysql_query ( "SELECT uid, username FROM user" );
$data=array ( );
while ( $row=mysql_fetch_array ( $result ) ) {
    $data[]=$row;
}
mysql_free_result ( $result );
//其他数据获取模块
```

然后在另一个文件中引入该文件的显示数据，如代码清单6-3所示。

代码清单6-3 显示部分

```
<html>
<head>
<meta http-equiv="content-type"content="text/html; charset=utf-8">
<title>demo.php</title> </head>
<body>
会员信息
<table>
<tr>
<td>ID</td>
<td>姓名</td>
</tr>
<? php
include ' u-data.php ';
foreach ( $data as $row ) {
echo"<tr> <td>". $row[ ' uid ' ]."</td> <td>". $row[ ' username ' ]."</td> </tr>";
}
}
```

```
//展示其他数据
? >
</table>
</body> </html>
```

这两种做法都没有实现PHP代码和HTML代码的分离，但是第二段代码相比第一段实现了数据获取和数据显示的分离。在第二种处理方式中，其中一个文件专门负责和数据库等打交道，获取和加工数据，然后交由另一个文件进行显示，这个负责显示的文件只是进行一些简单的逻辑操作，如循环、判断、输出等。

显然，第二种处理方式相比第一种要更加易于维护。对于第二种处理方式，还可以使用HereDoc语法使其更简洁，如下面代码所示：

```
<? php foreach ($data as $row) {
echo <<<TM
<tr> <td> $row[uid]</td> <td> $row[username]</td> </tr>
TM;
}? >
```

注意 PHP手册中提到，一般在数组中，对于非数字索引的键值应该加上引号，以避免产生警告，报告找不到key（此时，PHP会将key视为一个常量，优先寻找对应常量的值，若找不到则视为数组的key值），但如果是在字符串中，是可以省略引号的。

第二种处理方式即原生态模板机制，是当前一些PHP程序所采用的模板机制，特点是简单灵活、符合PHP的语法和使用习惯、学习成本低。缺点就是缺少一些更高级的功能。

想想上面的处理方式，我们在模板页面中包含了和数据库操作相关的页面，这似乎不应该出现在模板代码中，模板中只应该负责数据的展示，任何和数据显示不相关的代码，都应该从模板文件中拿出去，所以把前面的逻辑反过来，在负责数据操作的文件中包含模板，然后模板中只负责显示，这样看起来会更好一些。

6.2 实现一个简单的模板引擎骨架

想让模板机制更强大易用怎么办呢？比如想让模板引擎配置化，在使用时，只需刷新一下即可使用；想让模板引擎支持缓存；另外，还想在模板文件中检测变量类型及是否为空等，使模板文件进一步脱离PHP语法。这些都是可以做到的。

6.2.1 搭建模板引擎基础类骨架

先搭建一个模板引擎的骨架，让模板引擎变得可配置，这样会让其变得更灵活、更容易扩展。此时需要一个arrayConfig变量，用来存放模板的配置，对应的还应该有读取和写入配置的方法，代码如下所示：

```
class Template
{
private $ arrayConfig=array (
    ' suffix ' => ' .m ', //设置模板文件的后缀
    ' templateDir ' => ' template/ ', //设置模板所在的文件夹
    ' compiledir ' => ' cache/ ', //设置编译后存放的目录
    ' cache_ htm ' => false, //是否需要编译成静态的HTML文件
    ' suffix_ cache ' => ' .htm ', //设置编译文件的后缀
    ' cache_ time ' => 7200//多长时间自动更新, 单位秒
);
public $ file; //模板文件名
static private $ instance=null;
public function construct ( $ arrayConfig=array ()) {
    $ this-> arrayConfig= $ arrayConfig+ $ this-> arrayConfig;
}
/**
 *取得模板引擎的实例
 *
 * @return object
 * @access public
 * @static
 */
public static function getInstance () {
```

```
if ( is_null ( self: $instance )) {
self: $instance=new Template ( );
}
return self: $instance;
}
/*单步设置引擎*/
public function setConfig ( $key,$value=null )
{
if ( is_array ( $key ))
{
$this->arrayConfig= $key+ $this->arrayConfig;
} else {
$this->arrayConfig[ $key]= $value;
}}
/*获取当前模板引擎配置, 仅供调试使用*/
public function getConfig ( $key=null )
{
if ( $key ) {
return $this->arrayConfig[ $key];
} else {
return $this->arrayConfig; }
}
```

上面的代码只负责生成模板引擎的实例, 以及读取和设置一些模板引擎的基本属性, 还没有做实质性的工作。通过下面的代码来检测其是否正常工作:

```
<? php
include ' Template.php ';
$tpl=new Template ( );
var _dump ( $tpl->getConfig ( ));
```

上面的代码打印出该模板引擎的基本配置信息, 也可以在其构造方法中设置相关参数, 这会让模板变更比较灵活。接下来, 向模板引擎中添加变量注入和展示功能, 这需要在Template类中增加一个类变量, 用来容纳所有要注入的代码; 再增加一组方法, 用来

向模板文件中注入变量。代码如下所示：

```
private $ value=array ( );
/**
 *注入单个变量
 *
 *@param string $ key模板变量名
 *@param mixed $ value模板变量的值
 *@return void
 */
public function assign ( $ key,$ value )
{
    $ this-> value[ $ key]= $ value;
}
/**
 *注入数组变量
 *
 *@param array $ array*/
public function assignArray ( $ array )
{
    if ( is_ array ( $ array )) {
        foreach ( $ array as $ k=> $ v ) {
            $ this-> value[ $ k]= $ v;
        }
    }
}
```

可以一次注入一个变量，也可以一次注入多个变量，这样将会精简代码，然后需要一个show方法，展示我们的模板，代码如下所示：

```
public function show ( $ file )
{
    $ this-> file= $ file;
    if ( ! is_ file ( $ this-> path ( ))) {
        exit ( ' 找不到对应的模板 ');
    }
}
```

```
$compileFile= $this->arrayConfig[ 'compiledir ' ]. ' / ' .md5 ( $file ) . ' .php ' ;  
var_dump ( $compileFile );  
var_dump ( $this->path ( ) ); if ( ! is_file ( $compileFile ) ) {  
    mkdir ( $this->arrayConfig[ 'compiledir ' ] );  
    $this->compileTool->compile ( $this->path ( ), $compileFile );  
} else {  
    readfile ( $compileFile );  
}  
}
```

这个方法的流程如下：

- 1) 检查是否存在模板文件，如果不存在则退出，若存在则转到第二步；
- 2) 检查对应的模板是否已经被编译，如果未被编译，引入编译工具类，对其进行编译，若已编译过，则表示存在，转到第三步；
- 3) 读取编译后的文件。

6.2.2 编译类骨架

接下来，引入一个外部的编译类对模板进行编译。下面是编译类的代码：

```
<? php
/**
 *Created by waterfox.
 *Date: 11-12-12
 *Time: 上午2: 40
 *Version: 1.0
 *Description: 模板编译工具类
 */
class CompileClass {
private $ template; //待编译的文件
private $ content; //需要替换的文本
private $ comfile; //编译后的文件
private $ left= ' { '; //左定界符
private $ right= ' } '; //右定界符
private $ value=array (); //值栈construct () {
public function
}
public function compile ($ source,$ destFile ) {
file _put _contents ( $ destFile, file _get _contents ( $ source ));
}
}
```

这个类中的编译方法只是一个临时方法，把模板文件复制一份作为已编译文件，具体实现还需要到下一步。

到此为止，我们已完成了这个模板引擎的基本骨架，尽管有的方法还没有用到或没完善，但是已经可以工作了。Template类的完整代码如代码清单6-4所示。

代码清单6-4 模板引擎编译类

```
<? php
```

```
/**
 *一个简单的模板引擎
 */
class Template
{
private $ arrayConfig=array (
    ' suffix ' => ' .m ', //设置模板文件的后缀
    ' templateDir ' => ' template/ ', //设置模板所在的文件夹
    ' compiledir ' => ' cache ', //设置编译后存放的目录
    ' cache_ htm ' => false, //是否需要编译成静态的HTML文件
    ' suffix_ cache ' => ' .htm ', //设置编译文件的后缀
    ' cache_ time ' => 7200//多长时间自动更新, 单位秒
);
public $ file; //模板文件名, 不带路径
private $ value=array ();
private $ compileTool; //编译器
static private $ instance=null;
public function construct ( $ arrayConfig=array ())
{
    $ this-> arrayConfig= $ arrayConfig+ $ this-> arrayConfig;
    include ( ' CompileClass.php ');
    $ this-> compileTool=new CompileClass;
}
/**
 *取得模板引擎的实例
 *
 *@return object

 *@access public
 *@static
 */
public static function getInstance ()
{
    {
    if ( is_ null ( self: $ instance )) {
    self: $ instance=new Template ();
    }
    return self: $ instance;
    }
}
/*单步设置引擎*/
public function setConfig ( $ key, $ value=null )
```

```
{
if ( is_array ( $key )) {
    $this->arrayConfig= $key+ $this->arrayConfig;
} else {
    $this->arrayConfig[ $key]= $value;
}
}
/*获取当前模板引擎配置, 仅供调试使用*/
public function getConfig ( $key=null )
{
    if ( $key ) {
        return $this->arrayConfig[ $key];
    } else {
        return $this->arrayConfig;
    }
}
/**
*注入单个变量
*
*@param string $key模板变量名
*@param mixed $ value模板变量的值
*@return void
*/
public function assign ( $key, $value )
{
    $this->value[ $key]= $value;
}
/**
*注入数组变量
*
*@param array $ array*/

public function assignArray ( $ array )
{
    if ( is_array ( $ array )) {
        foreach ( $ array as $k=> $v ) {
            $this->value[ $k]= $v;
        }
    }
}
```

```
public function path ()
{
return $this->arrayConfig[ ' templateDir ' ]. $this->file. $this->arrayConfig[ ' suffix ' ];
}
public function show ( $ file )
{
$ this->file= $ file;
if (! is_file ( $ this->path ())) {
exit ( ' 找不到对应的模板 ' );
}
$ compileFile= $ this->arrayConfig[ ' compiledir ' ]. ' / ' .md5 ( $ file ) . ' .php ' ;
var _dump ( $ compileFile );
var _dump ( $ this->path ());
if (! is_file ( $ compileFile)) {
mkdir ( $ this->arrayConfig[ ' compiledir ' ]);
$ this->compileTool->compile ( $ this->path (), $ compileFile );
readfile ( $ compileFile );
} else {
readfile ( $ compileFile );
}
}
}
```

6.2.3 测试模板引擎

现在测试模板引擎，在template目录下新建一个member.m模板文件，内容如下：

```
<html>
{ data }
</html>
```

进行简单的调用：

```
<? php
include ' Template.php ';
$tpl=new Template ();
$tpl->show ( ' member ' );
```

运行以上的代码，效果如图6-1所示，可以看到生成了静态HTML文件（图中左侧的目录树中多了一个静态文件）。

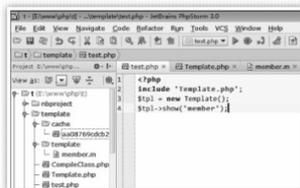


图 6-1 简单模板引擎运行效果

综上，模板引擎的基本开发思路如下：

- 1) 模板引擎要做的事情就是把逻辑层和表现层的代码分离，这是一个大原则。
- 2) 作为一个工具类，应该满足一些基本要求，其中之一就是可配置，像前面的代码中有很很大一部分就是实现了模板引擎的参数设置和读取，这为后期的方法调用和调试提供

了方便。

3) 确定模板引擎到底需要一些什么功能，需要哪些特性。例如，希望能够在HTML文件中编写页面，这个页面中可能会有一些PHP传来的数据，只需负责展示这部分数据即可，剩下的事就和我无关了。HTML文件里的数据并不被PHP引擎认识，只有翻译成PHP代码后才可能被执行。所以，需要一个翻译的过程，下面就来实现这个过程。

6.3 模板引擎的编译

模板引擎的编译实际上是一个“翻译”的过程，即把模板文件转换成PHP文件。因为模板文件中有一些变量和简单的逻辑，这些变量值来自逻辑层，而最终需要展示在浏览器中，因此就需要把这些值注入模板中，并最终转换成可以被执行的PHP代码。

6.3.1 实现变量标签

先实现一个最简单的功能，如果在模板文件中定义一个变量标签，模板引擎可以将它转化为PHP代码，然后输出。

以上一节的代码为例，模板文件中存在 { data } 标签，其作用就是输出对应的值，如果data为5，最终输出结果就应该是5。改造现有模板编译类，为其添加一个编译变量标签功能，这需要用到正则知识。我们想要实现如下的效果：

```
{ $data } → <? php echo $data;? >
```

在编译类中增加如下函数：

```
public function c_var () {
    $patten="# \ { \ \ $ ( [a-zA-Z_ \ x7f- \ xff][a-zA-Z0-9_ \ x7f- \ xff]* ) \ } #";
    if ( strpos ( $this->content, ' { $ ' )! ==false ) {
        $ this-> content=preg _ replace ( $ patten , " <? php echo \ $ this-> value [ ' \ \ 1 ' ] ;?
    > " , $ this-> con
        tent );
    }
}
```

这个方法用来匹配 { var } 这种模式，即匹配以大括号包含着变量的模式。若把它替换成 <? php echo var;? > 模式，则可用以下正则表达式来匹配：

```
# \ { \ $ ([a-zA-Z_ \x7f- \xff][a-zA-Z0-9_ \x7f- \xff]*) \} #
```

“\ {”匹配大括号的左边，“\ \”匹配美元符号，“\}”匹配大括号的右边。PHP的变量命名规则如下：

```
([a-zA-Z_ \x7f- \xff][a-zA-Z0-9_ \x7f- \xff]*)
```

即一个有效的变量名由字母或者下划线开头，后面跟上任意数量的字母、数字，或者下划线。

匹配后就是替换，使用正则中的反向引用完成。这里，\ \ 1表示第一个匹配结果，以此类推。相应的，Template类也需要做一些调整。

6.3.2 实现foreach标签

接下来实现循环标签。第一步仍然是指定规则，假设想要实现如下的效果：

```
{ foreach $b } { V } { /foreach }
→ <? php foreach ( $b as $k => $v ) {
echo $v;
}? >
```

其中，V表示数组的value；k表示数组的键值，具有特殊含义。另外，foreach可以写成loop标签，比如下面的模板代码：

```
{ loop $b } { K } — { V } { /loop }
```

上述代码可等价转换为：

```
<? php foreach ( $b as $k => $v ) {
echo $k, ' — ', $v;
}? >
```

想要实现这个效果，可以用正则表达式来判断是否存在foreach语句，以及是否闭合foreach语句。如果存在foreach语句，就用对应的PHP语法替换。判断foreach的起始可以用以下正则表达式：

```
"# \ { ( loop | foreach ) \ \ $ ( [a-zA-Z_ \x7f- \xff][a-zA-Z0-9_ \x7f- \xff]* ) } #"
```

判断是否闭合foreach，先要判断是否以“{”开始，然后判断foreach语句，中间一个空格，接下来判断PHP的变量标记“”符号，然后是变量名、“}”闭合符号。判断闭合符号比较简单，使用以下正则表达式即可：

```
"#\ {\/(loop|foreach)}#"
```

判断完匹配规则，就要替换对应的规则，把循环头替换成如下形式：

```
"<? php foreach ( ( array ) \ $this->value[ ' \ \ 2 ' ]as \ $K=> \ $V ) {? >"
```

循环尾部闭合处替换成如下形式：

```
"<? php }? >"
```

键值对如何处理呢？在这里，用“#\ {([K|V])\}#”匹配“K”或“V”符号，然后替换成输出语句。

foreach标签的编译函数如下：

```
public function c_foreach () {
    $patten1="#\ {(loop|foreach) \ \ $ (.*)}#";
    $patten2="#\ {\/(loop|foreach)}#";
    $patten3="#\ {([K|V])\}#";
    $this->content=preg_replace ($ patten1, "<? php foreach ( ( array ) \ \ 2 as \ $K=> \ $V ) {? >",$this->content);
    $this->content=preg_replace ($ patten2, "<? php }? >",$this->content);
    $this->content=preg_replace ($ patten3, "<? php echo \ $ \ \ 1;? >",$this->content);
}
```

```
}
```

写到这里，相信读者已发现规律并掌握大致思路了，无非就是匹配和替换。如果新增一种标签语法，那么对应新增一个解析函数即可。但这里存在一个问题，针对每一种模板语法都要使用一个函数，是不是很烦琐？而且有的匹配规则是可以复用的。这时考虑重构代码。

首先，给类增加两组变量，分别代表匹配规则和替换规则：

```
private $T_P=array (); private $T_R=array ();
```

然后，在构造函数中一次性初始化规则，将规则放到一个数组中，如下所示：

```
$this->T_P[]="# \ { \ \ $ ( [a-zA-Z_ \ x7f- \ xff][a-zA-Z0-9_ \ x7f- \ xff]* ) \ } #";  
$this->T_P[]="# \ { ( loop | foreach ) \ \ $ ( [a-zA-Z_ \ x7f- \ xff][a-zA-Z0-9_ \ x7f- \ xff]* ) \ } #";  
$this->T_P[]="# \ { \ / ( loop | foreach ) \ } #";  
$this->T_P[]="# \ { ( [K | V] ) \ } #";  
$this->T_R[]="<? php echo \ $this->value[ ' \ \ 1 ' ]? >";  
$this->T_R[]="<? php foreach ( ( array ) \ $this->value[ ' \ \ 2 ' ]as \ $K=> \ $V ) { ?>";  
$this->T_R[]="<? php }? >";  
$this->T_R[]="<? php echo \ $ \ \ 1;? >";
```

现在不再针对每一个标签语法写解析函数了，而是统一处理。

```
public function compile () {  
    $this->content=preg_replace ( $this->T_P,$this->T_R,$this->content );  
}
```

为了保持模板的简单，while、for等同样具有循环功能的语法就不在这里用标签实现了，而是统一采用foreach/loop标签实现。

6.3.3 实现if……else标签

继续完善模板引擎，为其增加if……else标签。根据前面的思路，主要还是正则判断。第一步仍然是确定规则。PHP中的if……else语法如下所示：

```
if ( a ) {  
  //do sth  
} else if {  
  //do another thing  
} else {  
  //do default thing  
}
```

模板中的标签语法如下：

```
{ if $data== ' abc ' }  
我是abc  
{ elseif $data== ' def ' }  
我是def  
{ else }  
我就是我， { $data }  
{ /if }
```

实现if……else标签的具体步骤如下：

1) 解析 { if data== ' abc ' } 标签，观察其形式为两边一个大括号，然后是if关键字，接下来是条件语句，因此可以得出匹配正则：

```
"#\ { if (.*?) \} #"
```

2) 解析 { elseifdata== ' def ' } 标签, 使用以下正则表达式匹配:

```
"#\ { ( else if | elseif) (.*) \} #"
```

3) 解析 { else } 标签, 使用以下正则表达式匹配:

```
"#\ { else \} #"
```

4) 解析 { /if } 标签, 使用以下正则表达式匹配:

```
"#\ { \ /if } #"
```

因为前面foreach标签也用到了关闭标签, 故可以把它们合起来用一个正则表达式进行匹配。

把上面正则表达式捕获的内容使用反向引用替换, 得到对应的正则匹配和替换规则, 如下所示:

```
$this->T_P[]="#\ { if (.*) \} #";  
$this->T_P[]="#\ { ( else if | elseif) (.*) \} #";  
$this->T_P[]="#\ { else \} #";  
$this->T_R[]=' <? php if ( \ \1) {? > ';  
$this->T_R[]=' <? php } else if ( \ \2) {? > ';  
$this->T_R[]=' <? php } else {? > ';
```

因此, 示例中的标签语法将被解析成如下的PHP代码:

```
<? php if ( $data== ' abc ' ) {? >  
我是abc  
<? php } else if ( $data== ' def ' ) {? >  
我是def  
<? php } else {? >  
我就是我, <? php echo $this->value[ ' data ' ];? >  
<? php }? >
```

6.3.4 对PHP原生语法的支持

现在已经给这个模板引擎添加了两个自定义标签，但是有时模板引擎还是不够灵活，且仅使用有限的标签不易完成复杂的语法，故还需要增加对PHP原生语法的支持。

原生的PHP就是能够直接在模板里写PHP代码，直接使用PHP函数，这样虽然破坏了模板的整洁，会带来一些安全问题（主要体现在如果使用别人制作的模板，可能会被嵌入恶意代码），但只要在可控范围内，还是可以接受的，而且PHP函数也很方便使用。

要实现对PHP原生语法的支持，关键一步在于如果PHP代码里使用了变量，则变量不是直接定义，而是放在了“this->value”这个值栈里，由于变量是分配进去的，因此不能直接使用，那么可以使用extract函数把值栈导入符号变量中。同时，在模板引擎中添加一个开关变量，控制是否允许在模板文件中使用PHP代码。如果不允许，遇到PHP代码则对其进行转义或屏蔽，其实现如下：

```
//配置中增加开关变量
'php_turn' => true //是否支持原生PHP代码
//编译时，导出符号表变量
extract ($this->value, EXTR_OVERWRITE);
//根据开关变量，选择性地进行转义
if ($config['php_turn'] === false) {
    $this->T_P[] = "# < \? (= | php |) (.+?) \? > #is";
    $this->T_R[] = "&lt;? \ \ 1 \ \ 2? &gt; ";
}

```

这样就能选择性地让模板引擎支持PHP原生代码了。如果不想支持，只需要在实例化类的时候传入对应的参数即可。

有时可能还需要在模板中使用注释，可通过下面的正则表达式进行匹配：

```
$this->T_P[] = "# \ { (\ \ # | \ *) (.*) (\ \ # | \ *) \} #";

```

由于模板编译后的PHP文件不需要阅读，所以注释不出现在编译后的PHP代码里，可以将其替换为空。当然，可以使用HTML形式的注释“<! -注释- >”，不需要我们的类做任何工作，这样更方便阅读。最后，如果模板中需要调用PHP函数，可以使用原生的PHP语法。

模板中的标签不一定非要对PHP语法进行翻译，可以利用模板标签实现一些特定功能。比如前端开发中，为了防止JavaScript等静态文件被浏览器缓存，往往会在JavaScript文件名后加一个版本号或时间戳，现在这个功能也可以用模板引擎来完成，如下所示：

```
public function c_staticFile () {
    $this->content=preg_replace ( '# \ { \! ( .*? ) \! \} # ', ' <script src= \ \ 1 ' . '? t=
.time ( ) . ' > </script> ',
    $this->content );
}
```

6.4 完善模板引擎

我们已经掌握了一个PHP模板引擎的基本实现方法，并且实现了变量输出、循环、判断等常用PHP逻辑，如果需求复杂，也可以使用原生PHP代码。接下来要做的是一些完善性的工作，如debug和缓存功能实现等。

6.4.1 模板缓存机制的实现

前面实现了模板引擎对自定义标签的解析，但这意义不大。为什么呢？因为这样一个模板引擎实现了从模板语法到PHP语法的翻译过程，而PHP本身的特点决定了这是一个低效和耗时的过程。

我们知道，PHP代码每运行一次，就要从头再来，从变量生成、逻辑处理一直到输出页面，这个过程很耗时。模板引擎只实现了模板语法到PHP语法的翻译，一定程度上对于美工是友好的，但是加重了PHP的负担。由于这个翻译过程是由正则表达式来实现，而正则表达式的效率比较低，因此使用模板引擎会造成网站效率降低。要解决这个问题，就需要使用缓存。

把模板翻译后的PHP文件所执行的结果保存为HTML静态文件，在下一次请求到来时，直接给出HTML文件，而省去翻译和执行的过程，从而大大提高网站运行效率，这也是使用模板引擎的初衷之一。

要减少模板引擎的开销，首先要减少“编译”的开销。因此通过判断编译后的PHP文件和模板文件的修改时间来确定是否“编译”。如果“编译”后PHP文件的修改时间早于模板文件的修改时间，意味着在“编译”文件生成后模板做了修改，需要重新编译；反之，则直接加载编译后的PHP文件。这个过程可以通过如下的代码实现：

```
if (! is_file ($compileFile) || filemtime ($compileFile) < filemtime ($this->path ())) {  
    //重新编译  
} else {  
    //直接加载编译后文件  
}
```

这样就省去编译标签的时间了，然后加上静态HTML缓存的功能。思路就是先判断设置中是否开启缓存静态文件，如果开启则判断缓存文件是否存在且未过期，若满足条件，直接读取缓存的HTML文件，否则直接执行编译后的PHP文件或生成新的HTML缓存。这个过程的部分代码如下：

```
public function reCache ($ file) {
    $ flag=false;
    $ cacheFile= $ this->arrayConfig[ ' compiledir ' ]. ' / ' .md5 ( $ file) . ' .htm ' ;
    if ( $ this->arrayConfig[ ' cache_ htm ' ]===true) { //是否需要缓存
        $ timeFlag= ( time () -@filemtime ( $ cacheFile) ) < $ this->arrayConfig[ ' cache_ time ' ]?
true: false;
        if ( is_ file ( $ cacheFile) && filesize ( $ cacheFile) >1&& $ timeFlag) { //缓存存在未过期
            $ flag=true;
        } else {
            $ flag=false;
        }
    }
    return $ flag;
}
```

将执行结果保存为HTML格式，这可以使用下面两个函数实现：

```
ob_ start (); //开始捕获输出缓冲
ob_ get_ contents () //获取输出缓冲
```

通过上面的几个逻辑，就实现了简单的模板缓存机制。最后，还可以给模板加上缓存清理机制，针对单个文件或全部文件进行清理，关于这部分内容这里不再赘述。

6.4.2 调试和缓存清理

接下来要完善模板的调试和错误处理功能，处理模板解析和路径问题。在PHP中可以使用相对路径，也可以使用绝对路径，但是相对路径比较容易导致路径混乱，因此为了程序的健壮性，我们在模板中使用绝对路径来定位文件。

注意，对于模板文件夹和缓存文件夹的处理是不同的。如果模板文件夹不存在，模板文件也就不存在，此时应该给出明确的错误提示；而当缓存文件夹不存在时，应该新建一个文件夹。还可以加上模板运行时间的计算等作为统计信息。用到的部分代码大致如下：

```
public $ debug=array ();
$ this-> debug[ ' begin ' ]=microtime ( true );
/*
*路径处理为绝对路径
*/
public function getPath () {
$ this-> arrayConfig[ ' templateDir ' ]=strtr ( realpath ( $ this-> arrayConfig[ ' templateDir
' ]), ' \ \ ', ' / ' ) . ' / ' ;
$ this-> arrayConfig[ ' compiledir ' ]=strtr ( realpath ( $ this-> arrayConfig[ ' compiledir ' ]), '
\ \ ', ' / ' ) . ' / ' ;
}
//.....
$ this-> debug[ ' spend ' ]=microtime ( true ) - $ this-> debug[ ' begin ' ];
$ this-> debug[ ' count ' ]=count ( $ this-> value );
//.....
public function debug _ info () { }
```

最后是缓存清理函数，遍历缓存文件下指定路径的全部文件并删除：

```
/*
*清理缓存的HTML文件
*/
public function clean ( $ path=null ) {
```

```
if ( $path===null ) {
    $path= $this->arrayConfig[ ' compiledir ' ];
    $path=glob ( $path. ' * ' . $this->arrayConfig[ ' suffix_cache ' ] );
} else {
    $path= $this->arrayConfig[ ' compiledir ' ].md5 ( $path ) . ' .htm ' ;
}
foreach ( ( array ) $path as $v ) {
    unlink ( $v );
}
}
```

到此为止，一个简单的模板引擎就完成了，这个模板引擎由两个类组成：Template.php类和CompileClass.php类。Template.php类的完整源代码如代码清单6-5所示。

代码清单6-5 Template.php类的完整源代码

```
<? php
/**
 *一个简单的模板引擎
 */
class Template
{
private $ arrayConfig=array (
    ' suffix ' => ' .m ' , //设置模板文件的后缀
    ' templateDir ' => ' template/ ' , //设置模板所在的文件夹
    ' compiledir ' => ' cache/ ' , //设置编译后存放的目录
    ' cache_ htm ' => false, //是否需要编译成静态的HTML文件
    ' suffix_cache ' => ' .htm ' , //设置编译文件的后缀
    ' cache_time ' => 2000, //多长时间自动更新, 单位秒
    ' php_turn ' => true, //是否支持原生PHP代码

    ' cache_control ' => ' control.dat ' ,
    ' debug ' => false
);
public $ file; //模板文件名, 不带路径
```

```
private $value=array (); //值栈
private $compileTool; //编译器
static private $instance=null;
public $debug=array (); //调试信息
private $controlData=array ();
public function construct ( $ arrayConfig=array ())
{
    $this->debug[ ' begin ']=microtime ( true );
    $this->arrayConfig= $ arrayConfig+ $ this->arrayConfig;
    $this->getPath ();
    if ( ! is_dir ( $ this->arrayConfig[ ' templateDir ' ])) {
        exit ( "template dir isn ' t found");
    }
    if ( ! is_dir ( $ this->arrayConfig[ ' compiledir ' ])) {
        mkdir ( $ this->arrayConfig[ ' compiledir ' ], 0770, true );
    }
    include ( ' CompileClass.php ');
}
/**
 *路径处理为绝对路径
 */
public function getPath () {
    $this->arrayConfig[ ' templateDir ']=strtr ( realpath ( $ this->arrayConfig[ ' templateDir
' ]), ' \ \ ', ' / ' ). ' / ' ;
    $this->arrayConfig[ ' compiledir ']=strtr ( realpath ( $ this->arrayConfig[ ' compiledir ' ]), '
\ \ ', ' / ' ). ' / ' ;
}
/**
 *取得模板引擎的实例
 *
 *@return object
 *@access public
 *@static
 */
public static function getInstance ()
{
    {
    if ( is_null ( self: $instance )) {
        self: $instance=new Template ();
    }
    return self: $instance;
}
```

```
}

/**
 *设置模板引擎的参数
 *@param $key
 *@param null $value*/
public function setConfig ( $key , $value=null )
{
if ( is_array ( $key ) ) {
$ this-> arrayConfig= $ key+ $ this-> arrayConfig;
} else {
$ this-> arrayConfig[ $ key]= $ value;
}}
/**
 *获取当前模板引擎实例的配置， 仅供调试信息时使用
 *@param null $key
 *@return array*/
public function getConfig ( $key=null )
{
if ( $key ) {
return $ this-> arrayConfig[ $ key];
} else {
return $ this-> arrayConfig;
}}
/**
 *注入单个变量
 *@param string $key模板变量名
 *@param mixed $value模板变量的值
 *@return void
 */
public function assign ( $key , $value )
{
$ this-> value[ $ key]= $ value;
}
/**
 *注入数组变量
 *@param array $array*/
public function assignArray ( $array )
{
```

```
if ( is_array ( $array )) {
    foreach ( $array as $k => $v ) {
        $this->value[ $k ] = $v;
    }
}

public function path ( )
{
    return $this->arrayConfig[ ' templateDir ' ]. $this->file. $this->arrayConfig[ ' suffix ' ];
}
/*
*判断是否开启了缓存
*/

public function needCache ( ) {
    return $this->arrayConfig[ ' cache_ htm ' ];
}
/**
*是否需要重新生成静态文件
*@param $ file
*@return bool
*/

public function reCache ( $ file ) {
    $ flag = false;
    $ cacheFile = $this->arrayConfig[ ' compiledir ' ]. md5 ( $ file ) . ' .htm ' ;
    if ( $this->arrayConfig[ ' cache_ htm ' ] === true ) { //是否需要缓存
        $ timeFlag = ( time ( ) - @filemtime ( $ cacheFile ) ) < $this->arrayConfig[ ' cache_ time ' ]?
true: false;
        if ( is_file ( $ cacheFile ) && filesize ( $ cacheFile ) > 1 && $ timeFlag ) { //缓存存在未过期
            $ flag = true;
        } else {
            $ flag = false;
        }
    }
    return $ flag;
}
/**
*显示模板
*@param $ file
*/

public function show ( $ file )
{

```

```
$this->file= $file;
if (! is_file ($this->path ())) {
    exit ( ' 找不到对应的模板 ');
}
$compileFile= $this->arrayConfig[ ' compiledir ' ].md5 ( $file ) . ' .php ';
$cacheFile= $this->arrayConfig[ ' compiledir ' ].md5 ( $file ) . ' .htm ';

if ( $this->reCache ( $file ) ===false ) {
    $this->debug[ ' cached ' ]= ' false ';
    $this-> compileTool=new CompileClass ( $this-> path ( ) , $ compileFile , $ this->
arrayConfig );
    if ( $this->needCache ( ) ) { ob_start ( );
    extract ( $this->value, EXTR_ OVERWRITE );
    if (! is_file ( $compileFile ) || filemtime ( $compileFile ) < filemtime ( $this->path ( ))) {
        $this-> compileTool->vars= $this->value;
        $this-> compileTool->compile ( );
        include $compileFile;
    } else {
        include $compileFile;
    }
    if ( $this->needCache ( ) ) {
        $message=ob_get_contents ( );
        file_put_contents ( $cacheFile, $message );
    }
    } else {
        readfile ( $cacheFile );
        $this->debug[ ' cached ' ]= ' true ';
    }
    $this->debug[ ' spend ' ]=microtime ( true ) - $this->debug[ ' begin ' ];
    $this->debug[ ' count ' ]=count ( $this->value );
    $this->debug_info ( );
}
public function debug_info ( ) {
    if ( $this->arrayConfig[ ' debug ' ]===true ) {
        echo PHP_EOL, ' ——debug info—— ', PHP_EOL;
        echo ' 程序运行日期: ', date ( "Y-m-d h: i: s" ), PHP_EOL;
        echo ' 模板解析耗时: ', $this->debug[ ' spend ' ], ' 秒 ', PHP_EOL;
        echo ' 模板包含标签数目: ', $this->debug[ ' count ' ], PHP_EOL;
        echo ' 是否使用静态缓存: ', $this->debug[ ' cached ' ], PHP_EOL;
        echo ' 模板引擎实例参数: ', var_dump ( $this->getConfig ( ) );
    }
}
```

```
    }
  }
  /**
  *清理缓存的HTML文件
  *@param null $path
  */
  public function clean ( $path=null ) {
  if ( $path===null ) {
  $path= $this->arrayConfig[ ' compiledir ' ];
  $path=glob ( $path. ' * ' . $this->arrayConfig[ ' suffix_cache ' ] );
  } else {
  $path= $this->arrayConfig[ ' compiledir ' ].md5 ( $path ) . ' .htm ' ;
  }
  foreach ( ( array ) $path as $v ) {
  unlink ( $v );
  }
  }
}
```

CompileClass.php类负责模板解析的文件，主要由正则表达式实现，完整源代码如代码清单6-6所示。

代码清单6-6 模板引擎编译CompileClass.php类的完整源代码

```
<? php
/**
*Created by waterfox.
*Version: 1.0
*Description:
*/
class CompileClass
{
private $ template; //待编译的文件
private $ content; //需要替换的文本
private $ comfile; //编译后的文件
```

```

private $left= ' {'; //左定界符
private $right= ' } '; //右定界符
private $value=array (); //值栈
private $phpTurn;
private $T_P=array ();
private $T_R=array ();
public function
construct ( $ template, $ compileFile, $ config )
{
    $ this-> template= $ template;
    $ this-> comfile= $ compileFile;
    $ this-> content=file _get _contents ( $ template );
    if ( $ config[ ' php _ turn ' ]==false ) {
        $ this->T _P[]="# < \? ( = | php | ) ( .+? ) \? > #is";
        $ this->T _R[]="# &lt;? \ \ 1 \ \ 2? &gt; ";
    }
    $ this->T _P[]="# \ { \ \ $ ( [a-zA-Z _ \ x7f- \ xff][a-zA-Z0-9 _ \ x7f- \ xff]* ) \ } #";
    $ this->T _P[]="# \ { ( loop | foreach ) \ \ $ ( [a-zA-Z _ \ x7f- \ xff][a-zA-Z0-9 _ \ x7f- \
xff]* ) } #i";
    $ this->T _P[]="# \ { \ / ( loop | foreach | if ) } #i";
    $ this->T _P[]="# \ { ( [K | V] ) \ } #";
    $ this->T _P[]="# \ { if ( .*? ) \ } #i";
    $ this->T _P[]="# \ { ( else if | elseif ) ( .*? ) \ } #i";
    $ this->T _P[]="# \ { else \ } #i";

    $ this->T _P[]="# \ { ( \ # | \ * ) ( .*? ) ( \ # | \ * ) \ } #";
    $ this->T _R[]="# <? php echo \ $ this->value[ ' \ \ 1 ' ]? >";
    $ this->T _R[]="# <? php foreach ( ( array ) \ $ this->value[ ' \ \ 2 ' ]as \ $ K=> \ $ V ) {?
>";
    $ this->T _R[]="# <? php }? >";
    $ this->T _R[]="# <? php echo \ $ \ \ 1;? >";
    $ this->T _R[]=' <? php if ( \ \ 1 ) {? > ';
    $ this->T _R[]=' <? php } else if ( \ \ 2 ) {? > ';
    $ this->T _R[]=' <? php } else {? > ';
    $ this->T _R[]=' ' ;
}
public function compile ( )
{
    $ this->c _var2 ( );
    $ this->c _staticFile ( );

```

```
file_put_contents ($this->comfile,$this->content);
}
public function c_var2 ()
{
$this->content=preg_replace ($this->T_P,$this->T_R,$this->content);
}
//加入对静态JavaScript文件的解析
public function c_staticFile () {
$this->content=preg_replace ('#\{!(.*?)!\}#','<script src=\\1'. '? t=
.time () .' ></script>', $this
->content);
}
public function set ($name,$value)
{
$this->$name= $value;
}
public function get ($name)
{
return $this->$name;
}
}
```

6.4.3 如何使用模板

首先建立静态HTML模板，如代码清单6-7所示。

代码清单6-7 模板文件Member.m

```
<html>
  {! like.js!}
  {$data }, {$person }
  <ul>
    { loop $b } <li> { V } </li> { /loop }
  </ul>
  <? php
  echo $pai*2;? >
  { if $data== ' abc ' }
  我是abc
  { elseif $data== ' def ' }
  我是def
  { else }
  我就是我, {$data }
  { /if }
  { #注释不会出现在编译后的PHP代码中# }
  123466——</html>
```

与此模板对应的动态PHP文件内容如代码清单6-8所示。

代码清单6-8 test.php

```
<? php
include ' Template.php ' ;
$tpl=new Template ( array ( ' php_turn ' => true, ' debug ' => true ));
$tpl->assign ( ' data ', ' hello world ');
$tpl->assign ( ' person ', ' cafeCAT ');
$tpl->assign ( ' pai ', 3.14 );
```

```
$arr=array (1, 2, 3, 4, "hahattt", 6);  
$tpl->assign ( ' b ', $arr );  
$tpl->show ( ' member ' );  
? >
```

运行结果这里不再截图展示。

实现一个模板引擎主要涉及正则表达式和文件操作，整个模板引擎约300行代码，基本能满足常规需求，并且扩展性非常好。

6.5 常用模板引擎

当前PHP应用中使用广泛的模板引擎有：Discuz、Smarty、织梦（DedeCms）、Blitz。这四种模板引擎代表了四种设计思想。我们着重关注其特性及实现。

6.5.1 Discuz模板引擎

Discuz是康盛在其Discuz系列产品中所使用的模板引擎，特点是体积小、语法简单，是一个轻量级的模板引擎，是与本书中所设计的模板引擎最接近的产品。Discuz模板引擎是通过特殊的Discuz! 模板标签完成特定的逻辑判断和输出操作的。比如，下面的语法：

1) 直接输出变量：

```
{ $abc }  
{ echo $abc }
```

2) 输出数组中某个变量：

```
{ $arr[0] }
```

3) 判断语句：

```
<! —— { if $ abc < 1 } —— >  
abc小于1  
<! —— { elseif $ abc == 1 } —— >  
abc等于1  
<! —— { else } —— >  
abc大于1
```

```
<! —— {/if} —— >
```

4) 循环语句

```
<! —— { loop $abc $key $val } —— >  
这里是 {$val}:  
<! —— {/loop} —— >
```

5) 语言包支持:

```
{ lang china }
```

等同于读取:

```
$language[ ' china ' ]
```

可以看出,和之前设计的模板引擎语法很接近,只不过Discuz模板引擎多了一些功能,如内嵌eval语句、引入其他模板、多语言支持等。当然,也可以在我们的模板基础上添加对应的解析功能或者变通实现。Discuz模板引擎对应的文件是class _template.php,其中部分代码如下:

```
function languagevar ($var) {  
    $vars=explode ( ': ', $var );  
    $isplugin=count ( $vars ) ==2;  
    if (! $isplugin) {  
        ! isset ( $this->language[ ' inner ' ]) && $this->language[ ' inner ' ]=array ();  
        $langvar=& $this->language[ ' inner ' ];
```

```
    } else {  
        ! isset ( $ this- > language[ ' plugin ' ] [ $ vars[0] ] ) && $ this- > language[ ' plugin ' ]  
        [ $ vars[0] ] = array ( );  
        $ langvar = & $ this- > language[ ' plugin ' ] [ $ vars[0] ];  
        $ var = & $ vars[1];  
    }  
    //……其余代码略
```

该函数的功能是我们的模板引擎中所没有的，其实现过程为：在进入函数后，根据“:”把参数转化成数组，判断数组中元素的个数，如果为2，则认为是插件的模板，否则认为是内部的模板。在此之前，需要了解Discuz模板引擎语言包的使用。在模板中通过下面代码使用语言包中的某个值：

```
{ lang china }
```

语言包在./source/language/目录下，以PHP数组形式存放。而如果是插件模板，语言包的使用又有一些不同，模板中通过以下方法调用插件语言包：

```
{ template identifier: test }
```

插件语言包存放在类似data/plugindata/identifier.lang.php文件中。理解了Discuz中语言包的使用后，我们豁然开朗，明白了这个函数的用意。后面是加载对应的语言包变量数组，然后赋值。

再看Discuz是如何解析 { lang china } 标签到PHP语法的。找到这个类中的parse_template函数，看到这一行就清楚了：

```
$ template = preg_replace ( "/\ { lang \ s+ ( .+? ) \} /ies", "\ $ this- > languagevar ( ' \ \ 1 ' )", $ template );
```

把 { lang china } 标签中的china变量提取出来，然后调用languagevar函数，返回china键在语言包变量数组里对应的值。这和之前的思路也差不多。其他标签的实现大同小异。

Discuz模板引擎优点是小巧、简单、易用。缺点是和自身产品结合较紧密、扩展性差，如果要抽离出来，需要做一些修改。另外，源代码可读性极差，没有任何注释。在没有阅读相关的文档之前，较难看懂。

Discuz 产 品 相 关 代 码 下 载 地 址：
<http://www.comsenz.com/products/discuzx>。

6.5.2 Smarty模板引擎

提起Smarty, PHP用户几乎无人不知, 它是最早被规模使用、最知名的模板引擎, 尽管目前其市场份额下滑很多, 但是在一些比较老的项目中, 还是能看到它的身影。Smarty是一个开源项目, 官方网站是<http://www.smarty.net/>, 当前最新版本是3.1.7。

Smarty这款专业的模板引擎是重量级产品, 也是所有模板引擎中最复杂的, 在使用方法上与其他没有区别, 只是功能更多、语法相应复杂。

在基础语法上, Smarty和其他模板引擎没有大的差别。比如下面这段PHP代码:

```
<? php for ($x=0; $x<20; $x+=2):? >
<php echo $x+1;? >
<? php endfor;? >
```

用Smarty语法表示如下:

```
{ for $x=1 to 20 step 2 }
{$x}
{/for}
```

从这个语法可以看出Smarty的强大和复杂, 仅一个循环语句, Smarty就有for、section、foreach等多种标签, 还支持{break}、{continue}等标签, 完全可以替代PHP中对应的语法, 而其他模板引擎往往支持的语法比较单一。

Smarty的强大和复杂还不止这些, 其还支持大量的内建函数、自定义函数以及简单UI功能, 甚至可以在Smarty基础上开发新插件, 进一步扩展Smarty功能。例如, 下面的代码可实现一个简单的表格:

```
<? php
$smarty->assign ( ' data ', array ( 1, 2, 3, 4, 5, 6, 7, 8, 9 ));
$smarty->assign ( ' tr ', array ( ' bgcolor="#eeeeee" ', ' bgcolor="#dddddd" ' ));
$smarty->display ( ' index.tpl ');
? >
//template.htm
{ html_ table loop= $ data }
```

此外，通过smarty_function_name方法能够自定义标签，增强Smarty的功能。这里不做详细介绍，感兴趣的朋友可以阅读官方手册。

Smarty的特点是强大、复杂，学习成本高。由于其功能比较强大，文档比较完善，历史比较久，事实上已经成为PHP模板引擎的代表，比较适合多个团队间协作的使用。

6.5.3 DedeCms模板引擎

DedeCms（织梦）是一套PHP的内容管理系统，其提供了一套模板，这套模板有自己的特色，以模块化的数据标签作为模板的设计思路。也就是说，织梦模板主要是以模块为标签的基本元素，比如一个标签代表一篇文章或者一个栏目的显示。这是由于其作为一款CMS软件，以内容展示为主的性质决定的。

例如，Arclist标记在织梦模板中专门用来获取文章，基本语法如下：

```
{ dede: arclist typeid= ' ' row= ' ' col= ' ' titlelen= ' ' infolen= ' '
imgwidth= ' ' imgheight= ' ' type= ' ' orderby= ' ' keyword= ' ' }
底层模板 (InnerText)
{/dede: arclist }
```

其中各属性如下：

typeid= ' '，栏目ID，在列表模板和档案模板中一般不需要指定，在封面模板中允许用“，”分开表示多个栏目；

row= ' '，返回文档列表总数；

col= ' '，分多少列显示（默认为单列）；

titlelen= ' '，标题长度；

infolen= ' '，内容简介长度；

imgwidth= ' '，缩略图宽度；

imgheight= ' '，缩略图高度；

type= ' '，表示档案类型，其中空值表示不使用这个属性，type= ' all ' 时为普通文档；

orderway= ' ' 值为desc或asc，指定排序方式是降序还是顺向排序，默认为降序；

keyword= ' ' 表示含有指定关键字的文档列表，多个关键字用“,”分开。

织梦模板引擎实际上是围绕内容展开的，而不是逻辑操作。因此，使用织梦模板不需要懂PHP编程、数据库CURD，只要按照模板标签配置对应的参数，即可取出想要的内容并展示，掌握基本HTML知识即可。织梦的模板标签供使用织梦CMS建站的站长使用。大多数主流CMS都使用织梦风格的标签。当然，这种形式的标签并不是织梦首创，之所以选择织梦，是由于其在国内普及率比较高，有代表性。

织梦这种类型的标签最终通过正则表达式匹配出参数，调用相应的方法获取数据。

织梦风格模板引擎的特点是：只可在自身产品中使用、不具有公用性、以内容展示为主，常见于CMS类型网站，学习和使用门槛低，不需要编程基础。

6.5.4 Blitz模板引擎

可能使用Blitz模板引擎的人并不多，这是一款特殊的模板引擎，由C语言实现，作为PHP的扩展使用。官方网站是http://alexeyrybak.com/blitz/blitz_en.html，当前最新版本为0.7.1.12。

具体安装过程不再赘述。看一个例子，代码如下：

```
$View=new Blitz ();
$View->load ( ' Where is the { {$what }}, Lebowski? ');
$View->display ( array ( ' what ' => ' money ' ));
```

当然，还可以这么写：

```
$View=new Blitz ( ' demo.tpl ');
$View->display ( array ( ' what ' => ' money ' ));
//demo.tpl中
Where is the { {$what }}, Lebowski?
```

或者使用set方法，这样比较符合我们的习惯。

Blitz模板引擎特点：功能足够强大，但由于服务器的限制，C扩展形式的模板引擎很难推广使用，DIY比较困难，适合个人项目，或者有实力的团队对其修改后内部使用。

除了上面介绍的几种模板引擎外，ThinkPHP模板引擎也比较有特色，扩充性也很好。

如果想进一步提升模板引擎的性能，可以使用以下手段：

- 1) 缓存AST（抽象语法树，Abstract Syntax Tree）结果；

2) 使用字节流的方式输出。

6.5.5 模板引擎的一些思考

随着Web的发展，仅一门语言或一种技术已经不能满足需求，分层架构显得越来越重要。在大型架构中，从不会简单地用PHP从头到尾实现一个完整的MVC架构，可能底层是C/Java支撑，负责密集运算和与数据库交互；表现层用JavaScript+PHP完成，JavaScript负责前端大部分的业务逻辑和数据发送，数据经由PHP达到后端。比如淘宝，使用PHP作为前端，Java是主要的后端语言，实现密集运算和数据中心。再如百度，主要逻辑和数学运算由C完成，PHP只是前端展示语言，并且在代码中大量使用C完成PHP扩展。

越是大型应用，PHP所扮演的角色越向前端靠近。PHP仅负责数据的传输和简单业务判断，配合JavaScript完成表现层的工作，此时PHP的角色是视图层。PHP本身是一套C函数的抽象，可以理解为PHP就是一个C写成的模板语言，此时再用PHP实现一个模板引擎已经不那么必要了。PHP代码和HTML代码混写也可以接受，况且PHP的强项就是字符串和数组处理，作为前端模板语言是最合适的。

从另一个角度讲，模板引擎真的对美工友好吗？其实模板引擎的语法不比PHP原生语法简单，况且很多情况下，这部分代码是由程序员自己完成，美工只负责基本的切图和CSS。对于程序员而言，每个模板引擎的语法都不一样，每学一种都得付出一定的学习成本。而原生的PHP语法在任何地方都通用，没有学习成本，只要把缓存做好就可以。

随着AJAX、JSON等开始流行，使用JavaScript负责部分前端数据展示变得流行，特别是jQuery的推波助澜，一些JavaScript引擎也引起了开发者的兴趣。比如，登录QQ选择“访问群社区”，即到类似地址<http://qun.qq.com/air/#3011111>。查看这群社区页面源代码：

```
<! ——user status——>
<p style="display: none"> <textarea id="tUserStatus"rows="0"cols="0"> <! [CDATA[
{ #template MAIN }
{ #if ' 0 ' == $T.id }
<div class="fl"> <a href="/air/#"rel="signin"title="登录">登录</a> </div>
{ #else }
<div class="fl">欢迎您, &nbsp; &nbsp; { GF.nick ($T.nick)} </div>
```

```
| T_DIS
  PATCH: 0.0026S |",
  module: "default", controller: "index", action: "frame",
  env: "live", way: "frame", language: "zh-cn",
  user: {
  id: "363575104",
  nick: "%E7%8C%AA%E8%82%89%E6%9C%89%E6%AF%92",
  gkey: ""
  },
```

PHP只需要负责生成数据，至于数据的展示和逻辑判断等则由JavaScript处理。

在多层架构体系中，PHP扮演的角色越来越少，所以PHP也就用不到太多“道具”了。我们编程时应做多种语言协作、前端后端协作，百花争鸣，让每一种语言和工具都扮演好自己的角色，做自己最擅长的事，合理分配负载。

总之，数据和代码分离的思想很重要。

```

<div class="fl"> &nbsp; &nbsp; &nbsp; [ <a href="/air/#"rel="signout"title="退出">退出</a> ]</div >
{ #if }
<span> &nbsp; &nbsp; &nbsp; | &nbsp; &nbsp; &nbsp; </span>
{ #/template MAIN }
]]> </textarea>
//.....
<div class="groupList"id="groupList">
<table border="0"cellpadding="0"cellspacing="0"width="300">
<tr>
<th align="left">我加入的群</th>
<th align="right">创建日期</th>
</tr>
{ #foreach $ T.c as record }
{ #include ROW root= $ T.record }
{ #for } </table>
</div> </div>
{ #/template MAIN }
{ #template ROW }

```

在页面底端看到了这样的代码:

```

<script type="text/javascript"src="/god/m/js/loader.zh-cn.js"> </script >
<script type="text/javascript">
document.domain="qq.com";
jL.script ( { mark: "jquery", uri: "/god/m/js/jquery.zh-cn.js", depend: true } );
jL.script ( { mark : "jquery-plugins", uri : "/god/m/js/jquery/plugins.zh-cn.js", depend :
true } );
jL.script ( { mark: "frame", uri: "/god/m/js/frame.zh-cn.js? __=96464", depend: true, onload:
function ( p ) {
if ( $.browser.msie&&"6.0"== $.browser.version ) {
document.execCommand ( "BackgroundImageCache", false, true );
}
}
G.run ( p );
}, params: {
request: location.href,
domain: "qun.qq.com", server: "70.75", client: "123.160.133.18",
elapsed: "0.0049", memory: "0.48MB", profile: "T_LOAD: 0.0007S | T_ROUTE: 0.0005S

```

6.6 本章小结

本章实现了一个简单的模板引擎，通过其实现我们可以看到，不管是什么模板引擎，其思路主要就是模板语法解析，通过把模板标签解释为实际的语法去执行。生成静态文件的思路也是一致的，即获取程序运行时的输出缓冲区，保存到静态文件中。

实际上，PHP就是一个C应用在Web上的模板，好比JSP是Java Servlet的模板一样，复杂的模板语法不会受开发者的欢迎。模板引擎有两个发展思路：一个是简洁，尽量接近PHP原生语法，提供少而精、易于学习的模板语法；另一个是特化，就是模板引擎的语法和当前应用紧密结合，这也是CMS中比较常见的做法。

第7章 PHP扩展开发

学习本章前，你应已熟识C语言，否则建议先去学习C语言的基本知识，因为本章所有知识都是建立在C语言之上的。如果你之前从未编写过PHP扩展，但是对编写PHP扩展非常感兴趣的话，通过学习本章之后，你就能掌握编写PHP扩展的方法。

7.1 为什么要开发PHP扩展

PHP有丰富的函数库，一般情况下已经足够我们使用。为什么还要开发PHP扩展呢？主要有以下几个原因：

如果应用是非常注重效率的，如复杂的图像算法，需要编写成PHP扩展。

有些系统调用不能用PHP直接访问的，需要编写成扩展，例如使用Linux下的fork()函数创建一个进程。

如果想商业化一个应用，但是又不想暴露源代码，可以编写成扩展（当然还可以使用Zend公司的加密工具Zend Guard）。

当然还有其他原因，比如想显示自己的技术等。不管是基于什么目的去（学习）编写PHP扩展，最终都是有益的，因为你会从中得到很多在PHP语言层学不到的知识。

下面让我们就开始PHP扩展之旅吧。

7.2 搭建PHP扩展框架

在编写PHP扩展时，有很多枯燥而烦琐的工作需要完成，庆幸的是PHP提供了一些有用的工具帮我们完成这些工作。接下来介绍这些工具的使用方法。

7.2.1 PHP源代码目录

在编写扩展之前，先了解一下PHP源目录。PHP源码目录中的文件如图7-1所示。



图 7-1 PHP源码目录文件

表7-1中列出了PHP源代码的一些主要目录及描述。

目录	描述
ext	这是存放动态和内置模块的目录，在这里可以找到所有的 PHP 官方扩展，并且以后也会在这里编写扩展
main	包含 PHP 的主要宏定义
pear	该目录就是“PHP 扩展与应用库”目录，包含 PEAR 核心文件
sapi	包含不同服务器抽象层的代码
TSRM	Zend 和 PHP 的“线程安全管理器”目录
Zend	包含 Zend 引擎的所有文件，在这里你可以找到所有的 Zend API 定义和宏等

除以上源码目录之外，还需要了解以下几个重要的头文件，因为在编写扩展的过程中，一般要把这些文件包含进来：

main/php.h：位于main目录下。包含绝大部分PHP宏及PHP API定义。

Zend/zend.h：位于Zend目录下。包含绝大部分Zend宏及Zend API定义。

Zend/zend_API.h：位于Zend目录下。包含Zend API的定义。

了解了PHP源代码目录树和主要文件之后，现在可以开始尝试在Windows平台和Linux平台创建PHP扩展的框架了。

7.2.2 ext_skel工具

PHP内核开发人员为编写PHP扩展提供一个很好用的“自动构建系统”工具“ext_skel”。使用ext_skel可以很方便地搭建一个扩展框架。

ext_skel工具在PHP源代码目录下的ext目录下。ext目录下有两个文件，ext_skel和ext_skel_win32.php。第一个在Linux系统下使用，而ext_skel_win32.php则是在Windows系统下使用的。

7.2.3 Windows平台环境配置

在Windows平台配置编写PHP扩展的环境比较简单，只需要安装VC++就行了，版本建议使用6.0，因为VC++6.0速度比较快，而且比较轻巧。另外建议安装“Visual Assist X”，这样会使编写PHP扩展变得非常容易。

除了安装VC++之外，还需要一样非常重要的东西，那就是PHP源代码。没有PHP源代码就不能编写PHP扩展。所以需要下载PHP的源代码，本书使用的PHP源代码版本是PHP 5.10，大家可以去官方网站下载。

有了VC++和PHP源代码，就可以开始编写扩展了。

1.使用ext_skel工具

下面就来看怎么使用ext_skel工具创建一个PHP扩展的框架。

首先在命令行下进入到PHP源代码目录的ext目录下，如图7-2所示。



图 7-2 进入PHP的源码目录

输入以下命令创建扩展框架：

```
php ext_skel_win32.php—extname=myext
```

结果如图7-3所示。

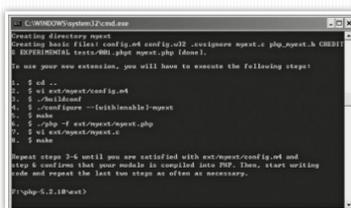


图 7-3 创建扩展框架

注意 如果提示“php不是内部或外部命令，也不是可运行的程序或批处理文件”，请先安装PHP并且把PHP安装目录绑定到Windows的环境变量中。

成功之后会发现ext目录下多了一个myext目录，该目录的结构如图7-4所示。



图 7-4 扩展框架文件myext目录

myext目录中，myext.dsp是VC++的工程文件。

2.编译安装扩展

用VC++打开myext.dsp工程文件，按编译按钮开始编译这个扩展。在编译的过程中，会发现编译出错了，提示的错误信息如图7-5所示。

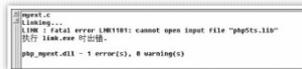


图 7-5 编译扩展出错

错误提示找不到php5ts.lib文件。从PHP的安装目录（注意不是源代码目录）下的dev目录中，把php5ts.lib文件复制到创建的扩展目录（myext）下，然后再次按编译按钮，这次成功编译扩展。

编译扩展之后，源代码的根目录下多了一个Release_TS目录，在该目录中有刚刚编译好的动态扩展文件（php_myext.dll），如图7-6所示。



图 7-6 编译完成的扩展文件

注意 如果在源代码根目录下找不到Release_TS目录，而找到了Debug_TS目录的话，请把VC++的编译模式改为Release。

至此，已在Windows环境下创建一个PHP的扩展，现在把这个扩展安装到PHP中。

安装也很简单，只要把编译好的dll文件复制到PHP安装目录的ext目录下，然后在php.ini配置文件中添加“extension=php_myext.dll”，并重启Web服务器（nginx或者Apache）就可以。

安装完成后，可以在phpinfo中看到扩展信息，如图7-7所示。



图 7-7 phpinfo中显现的扩展信息

可以调用confirm_myext_compiled()函数测试我们的扩展：

```
<? php
echo confirm_myext_compiled ("hello myext");
? >
```

如果输出以下信息，说明已经成功编写并且安装一个PHP扩展：

```
Congratulations! You have successfully modified ext/myext/config.m4.Module hello myext is
now compiled into PHP.
```

7.2.4 Linux平台环境配置

在Linux平台下搭建PHP扩展的开发环境比在Windows平台下容易，本节以Ubuntu系统为例，介绍怎么在Linux系统下搭建PHP扩展的开发环境。

1.安装php dev包

使用php dev包中的phpize工具可以减少很多烦琐的步骤。如果使用PHP源码编译的话，就不用安装php dev包，因为PHP源码中已经有phpize工具。

在Ubuntu下安装php dev包可以使用以下命令：

```
$ sudo apt-get install php5-dev
```

安装完毕后，使用以下命令查看是否安装成功：

```
$ phpize--version Configuring for:
PHP Api Version: 20090626
Zend Module Api No: 20090626
Zend Extension Api No: 220090626
```

2.使用ext_skel工具

Linux平台也有ext_skel工具，在PHP源码的ext目录下有个ext_skel文件（对应Windows下的ext_skel_win32.php文件，用法相似），使用它可以方便地搭建一个PHP扩展的框架，例如创建一个myext扩展：

```
$/ext_skel—extname=myext
Creating directory myext
Creating basic files: config.m4 config.w32.cvsignore myext.c
```

```
php_myext.h CREDITS EXPERIMENTAL tests/001.phpt myext.php[done].
```

To use your new extension, you will have to execute the following steps:

1. `$ cd..`
2. `$ vi ext/myext/config.m4`
3. `$./buildconf`
4. `$./configure—[with | enable]-myext`
5. `$ make`
6. `$./php-f ext/myext/myext.php`
7. `$ vi ext/myext/myext.c`
8. `$ make`

Repeat steps 3–6 until you are satisfied with `ext/myext/config.m4` and step 6 confirms that your module is compiled into PHP. Then, start writing code and repeat the last two steps as often as necessary.

生成扩展框架后，需要修改扩展的m4文件。打开config.m4文件，去掉以下配置前的dnl:

```
dnl PHP_ARG_ENABLE ( hello, whether to enable hello support,  
dnl[—enable-hello Enable hello support] )
```

修改后如下:

```
PHP_ARG_ENABLE ( hello, whether to enable hello support,  
[—enable-hello Enable hello support] )
```

3.编译安装扩展

创建好扩展框架后，进入到扩展的目录下，使用phpize命令生成扩展的配置工具，然后编译和安装，过程如下:

```
$ cd myext
$ phpize
$ ./configure——with-php-config=/usr/local/php5/bin/php-config
$ make
$ make test
$ make install
```

编译安装后，可以在/usr/local/php5/lib/php/extensions/no debug non zts 20060613/目录下看到生成的扩展文件myext.so。接着在php.ini文件中添加扩展信息：

```
extension=myext.so
```

最后使用“php m”命令查看扩展是否安装成功，如下：

```
$/usr/local/php5/bin/php-m
[PHP Modules]
.....
myext
.....
[Zend Modules]
```

7.2.5 PHP的生命周期

一个PHP实例，无论是从init脚本中调用的，还是从命令行启动的，都会依次经过Module init、Request init、Request shutdown、Module shutdown四个过程。当然，这之间还会执行脚本自己的逻辑。两种init和两种shutdown各会执行多少次、各自的执行频率有多少，取决于PHP是用什么SAPI与宿主通信的。最常见的四种启动PHP的方式如下：

直接以CLI/CGI模式调用；

多进程模块；

多线程模块；

Embedded（嵌入式，在自己的C程序中调用Zend Engine）。

SAPI（Server abstraction API，服务器抽象化程序接口）提供一个接口，使得PHP可以和其他应用进行交互数据。也就是说，PHP能够跟其他程序（如Apache）交互就是这个接口起的作用。

在命令行模式下运行一个PHP程序的主要流程如图7-8所示。

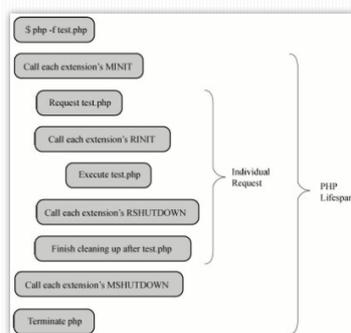


图 7-8 PHP生命周期

从图7-8中可以看到，当在命令行中敲入“php f test.php”时，会发生如下过程。

(1) Call each extension 's MINIT 这个过程在扩展被载入时调用。一般写在扩

展的以下函数中:

```
PHP_MINIT_FUNCTION ( myext )
{
//注册常量或者类等初始化操作
return SUCCESS;
}
```

(2) Request test. php

请求test.php文件。当请求到达后，PHP会初始化执行脚本的基本环境，例如创建一个执行环境，包括保存PHP运行过程中变量名称和变量值内容的符号表，以及当前所有的函数以及类等信息的符号表。然后PHP会调用所有模块RINIT函数，在这个阶段各个模块也可以执行一些相关的操作，模块的RINIT函数和MINIT函数类似:

```
PHP_RINIT_FUNCTION ( myext )
{
/*例如记录请求开始时间*/
/*随后在请求结束的时候记录结束时间*/
/*这样就能够记录下处理请求所花费的时间了*/
return SUCCESS;
}
```

(3) Execute test. php

执行test.php阶段，主要是把PHP文件编译成Opcodes，然后在PHP虚拟机下执行。

(4) Call each extension ' s RSHUTDOWN

请求处理完后进入结束阶段，一般脚本执行到末尾或者通过调用exit () 或者die () 函数，PHP都将进入结束阶段。和开始阶段对应，结束阶段也分为两个环节，一个在请求

结束后 (RSHUTDOWN), 一个在SAPI生命周期结束时 (MSHUTDOWN)。

RSHUTDOWN类似如下:

```
PHP_RSHUTDOWN_FUNCTION (myext)
{
    /*例如记录请求结束时间, 并把相应的信息
    写入到日至文件中*/
    return SUCCESS;
}
```

MSHUTDOWN类似如下:

```
PHP_MSHUTDOWN_FUNCTION (myext)
{
    /*注销一些持久化的资源*/
    return SUCCESS;
}
```

在请求一个PHP页面时, PHP基本上是按照这个流程执行的。所以在以后编写扩展时, 可以按照这个流程做一些初始化工作等。打开刚才创建的扩展的myext.c文件, 可以看到类似以上的结构, 如图7-9所示所示。

这个扩展例子中并没有做任何的操作, 在以后的扩展中会在这些函数中增加一些操作。

```
181 /* {{{ PHP_INIT_FUNCTION
182 */
183 PHP_INIT_FUNCTION(myext)
184 {
185     /* If you have INI entries, uncomment these lines
186     REGISTER_INI_ENTRIES();
187     */
188     REGISTER_INI_ENTRIES();
189     return SUCCESS;
190 }
191 /* }}} */
192
193 /* {{{ PHP_SHUTDOWN_FUNCTION
194 */
195 PHP_SHUTDOWN_FUNCTION(myext)
196 {
197     /* uncomment this line if you have INI entries
198     UNREGISTER_INI_ENTRIES();
199     */
200     UNREGISTER_INI_ENTRIES();
201     return SUCCESS;
202 }
203 /* }}} */
204
205 /* Remove if there's nothing to do at request start */
206 /* {{{ PHP_REQUEST_FUNCTION
207 */
208 PHP_REQUEST_FUNCTION(myext)
209 {
210     return SUCCESS;
211 }
212 /* }}} */
```

图 7-9 PHP生命周期相关函数

7.3 PHP内核中的变量

要编写PHP扩展不可避免要接触PHP内核中变量的表示方式，所以必须了解PHP变量在内核中的存储方式和使用方法。

7.3.1 PHP变量在内核中的存储方式

PHP是弱类型语言，也就是说一个PHP变量可以保存任何的数据类型。但是PHP是使用C语言编写的，而C语言是强类型语言，每个变量都有固定类型，不能随意改变变量的类型（可以通过强类型转换改变，不过有可能出现问题），在Zend引擎中是怎么可以做到一个变量保存任何的数据类型呢？

打开Zend/zend.h头文件，会发现以下一些结构体：

```
typedef struct _zval_struct zval;
typedef union _zvalue_value {
    long lval; /*long value*/
    double dval;
    /*double value*/
    struct {
        char*val;
        int len;
    } str;
    HashTable*ht;
    /*hash table value*/
    zend_object_value obj;
} zvalue_value;
struct _zval_struct {
    /*Variable information*/
    zvalue_value value;
    /*value*/
    zend_uint refcount;
    zend_uchar type;
    /*active type*/
    zend_uchar is_ref;
```

```
};
```

zval结构体就是通常用到的PHP变量（如variable）在内核中的表示方式。在zval结构体中，可以看到4个成员变量，分别是：

zvalue_value value：变量的值，PHP变量的值就保存在这里。

zend_uint refcount：变量引用数，变量引用计算器。

zend_uchar type：变量的类型，下面详细说明。

zend_uchar is_ref：变量是否被引用。

zval结构体的value成员变量是一个zvalue_value联合体，PHP能够保持任何的结构类型就是因为这个联合体。从zvalue_value联合体的成员变量中可以看到，不同的类型会保存到不同的成员变量中，这样就实现了PHP变量可以存储任何数据类型。例如，当变量是整数类型时，会保存到value的ival成员变量中；而当变量的类型是字符串时，又会保存到value的str成员变量中。表7-2展示了不同类型保存到对应的成员变量中。

PHP 语言层类型	保存在 zvalue_value 的成员变量
long, bool, resource	ival
double	dval
string	str (low 保存字符串的长度, val 保存字符串的值)
array	ht
object	obj

现在已解决了一个PHP变量可以保存任意类型的问题，但是另外一个问题又出现了，就是Zend引擎是怎么知道这个变量保存的是什么类型呢？我们注意到，zval结构体中有个type成员变量，这个成员变量就是保存一个PHP变量的类型。

Zend引擎定义了几种变量类型，如下所示：

```
#define IS_NULL 0
#define IS_LONG 1
#define IS_DOUBLE 2
#define IS_STRING 3
#define IS_ARRAY 4
```

```
#define IS_OBJECT 5
#define IS_BOOL 6
#define IS_RESOURCE 7
```

每一个宏定义对应PHP语言层的一种类型，例如当zval的type成员变量等于IS_STRING时（zval.type==IS_STRING），说明这个变量的类型是字符串类型。表7-3所示是每个宏定义对应的PHP类型。

宏定义	表示类型	宏定义	表示类型
IS_NULL	NULL类型 (null)	IS_ARRAY	数组类型 (array)
IS_LONG	整数类型 (int)	IS_OBJECT	对象类型 (object)
IS_DOUBLE	浮点型类型 (float)	IS_BOOL	布尔类型 (bool)
IS_STRING	字符串类型 (string)	IS_RESOURCE	资源类型 (resource)

可以通过下面的代码打印一个zval的类型：

```
switch ( zval.type ) {
case IS_NULL:
php_printf ( "zval type is null \n" );
break;
case IS_STRING:
php_printf ( "zval type is string \n" );
break;
case IS_LONG:
php_printf ( "zval type is long \n" );
break;
case IS_ARRAY:
php_printf ( "zval type is array \n" );
break;
.....
}
```

7.3.2 PHP内核变量访问宏

使用“`zval.type=IS_LONG`”方式可以设置一个变量的类型，不过这样做不是很合适，因为不能预测PHP以后的版本会发生什么变化，有可能在以后的版本中把`type`成员变量的名字改成`type_gc`或者其他的名字，那么之前写好的扩展就不能适应于这些版本了。为了解决这个问题，PHP内核提供一个访问和设置变量类型的方法，具体如下：

```
Z_TYPE ( zval ) 对应zval结构体的实体
Z_TYPE_P ( &zval ) 对应zval结构体的指针
Z_TYPE_PP ( &&zval ) 对应zval结构体的二级指针
```

可以用以下方式设置变量的类型：

```
Z_TYPE ( zval ) =IS_LONG;
```

用以下方法访问变量的类型：

```
if ( Z_TYPE ( zval ) ==IS_LONG ) {
printf ( "is long \n" );
}
```

这样，就算以后`zval`结构体的`type`成员变量改名，我们的扩展也可以继续使用。

与变量的类型一样，变量的值也有相应的访问宏定义，如表7-4所示。

使用表7-4中的这些宏可以设置一个变量的类型和值。例如，创建一个值为10的整数变量`lvar`：

```
zval lvar;
```

```
Z_TYPE (lvar) =IS_LONG;
Z_LVAL (lvar) =10;
```

如果用PHP脚本的话，相当于以下的代码：

类 型	访问宏
整数类型	Z_LVAL (eval)
	Z_LVAL_P (&eval)
	Z_LVAL_PP (&&eval)
浮点类型	Z_DVAL (eval)
	Z_DVAL_P (&eval)
	Z_DVAL_PP (&&eval)
布尔类型	Z_BVAL (eval)
	Z_BVAL_P (&eval)
	Z_BVAL_PP (&&eval)
字符串类型	取得值:
	Z_STRVAL (eval)
	Z_STRVAL_P (&eval)
	Z_STRVAL_PP (&&eval)
	取得长度:
	Z_STRLEN (eval)
Z_STRLEN_P (&eval)	
Z_STRLEN_PP (&&eval)	
数组类型	Z_ARRVAL (eval)
	Z_ARRVAL_P (&eval)
	Z_ARRVAL_PP (&&eval)
资源类型	Z_RESVAL (eval)
	Z_RESVAL_P (&eval)
	Z_RESVAL_PP (&&eval)

```
$lvar=10;
```

注意 PHP内核还利用上面的宏包装一些更方便的宏。考虑到上面的宏已经足够我们使用，这里不再详细说明。

7.3.3 引用计数器与写时复制

众所周知，PHP是不支持指针的，但是如果希望两个变量同时指向同一内存块怎么办呢？为了解决这个问题，PHP内核里使用了引用计数器。

7.3.1 节已经了解过PHP变量在内核中的存储方式了，zval结构中有以下两个成员变量用于引用计数器：

is_ref: BOOL值，标识变量是否是引用集合。

refcount: 计算指向引用集合的变量个数。

看下面这段PHP代码：

```
<? php
$a="this is variable";
? >
```

一个zval结构的实体称为zval容器。在PHP语言层创建一个变量就会相应地在PHP内核中创建一个zval容器。因为上面的代码创建一个变量\$a，所以在PHP内核中会创建一个zval容器。又因为这个变量不是一个引用，所以zval容器的is_ref等于FALSE，并且refcount等于1。

再看下面一段代码：

```
<? php
$a="this is variable";
$b=$a;
? >
```

上面这段代码创建了两个变量a和b，所以在PHP内核会创建两个zval容器来保存它

们。变量b被赋予变量a的值，那么现在变量a对应的zval容器的is_ref字段为何值呢？

由于变量b并不是引用变量a，所以变量a的is_ref字段的值为FALSE，这个容易理解。但是如果使用xdebug打印变量a的话，会发现refcount等于2。这就有点奇怪了，为什么ref count是2的？既然不是引用，refcount也应该是1的啊。要明白为什么refcount等于2，首先要知道PHP的写时复制（copy on write）机制。

写时复制是一个解决内存复用的方法。例如上面代码，如果简单地把a的值赋给b，就有两个"this is variable"字符串的复制，这样不利于内存复用。因为完全可以使用一个"this is variable"字符串的复制完成工作。所以简单的赋值是非常耗内存的，写时复制就是为了解决这种问题而创造的，那什么是写时复制呢？顾名思义，就是当变量的值改变时才进行内存的复制。要理解写时复制，先看下面这段代码：

```
<? php
$a="this is variable";
xdebug_debug_zval('a');
$b=$a;
xdebug_debug_zval('a');
$a="changed value";
xdebug_debug_zval('a');
? >
```

上面这段代码使用xdebug调试工具。输出如图7-10所示。



```
a
[refcount=1, is_ref=0, memory 'this is variable' (length=14)]
a
[refcount=2, is_ref=0, memory 'this is variable' (length=14)]
a
[refcount=1, is_ref=0, memory 'changed value' (length=12)]
```

图 7-10 使用xdebug打印变量

如图7-10所示，当将变量a的值赋给变量b时，变量a的refcount增加1，所以这时候变量a跟变量b是指向同一内存块的。当改变变量a的值时，发现refcount的值变回1，所以这时候变量a和变量b指向不同的内存块，这就是写时复制机制。就是两个指向同一内存块的变量，当其中一个变量的值发生变化，才会另外创建一个内存块去保存新的值。聪明

的读者会发现其实写时复制也是一种引用，只不过这种引用会受变量值的改变而破坏罢了。

看最后一种情况，如果用户在PHP脚本中显式地让一个变量引用另一个变量，PHP内核会如何处理呢？看下面一段代码：

```
<? php
$a=1;
xdebug_debug_zval('a');
$b=&$a;
xdebug_debug_zval('a');
$b+=5;
xdebug_debug_zval('a');
? >
```

上面的代码输出如下：

```
a:
 (refcount=1, is_ref=0), int 1
a:
 (refcount=2, is_ref=1), int 1
a:
 (refcount=2, is_ref=1), int 6
```

可以看到，当显式地让一个变量引用另一个变量时，变量的`is_ref`字段会设置为1，表示此变量被引用，另外引用计数器（`refcount`）也相应地加1。而在PHP内核中通过以下代码判断是否复制变量：

```
if (( *varval ) ->is_ref || ( *varval ) ->refcount < 2) {
return *varval;
}
```

从上面的代码中可以知道，当变量被引用或者引用计数器小于2时会直接返回变量的指针（直接返回变量的实体，而不复制变量的值）。当修改一个被引用变量的值时，所有引用它的变量其值也会被修改，因为它们指向同一个zval容器。

7.4 PHP内核中的HashTable分析

HashTable是PHP的灵魂，因为在Zend引擎中大量地使用了HashTable，如变量表、常量表、函数表等，这些都是使用HashTable保存的，另外PHP的数组也是使用HashTable实现的。所以了解了PHP的HashTable才能真正了解PHP。

PHP内核的HashTable跟传统HashTable实现基本一样，但是增加一些特性（如同时维护一个双链表），所以了解传统HashTable实现对了解PHP的HashTable有非常大的帮助。

下面了解PHP内核的HashTable。

7.4.1 PHP内核HashTable的数据结构

PHP中的HashTable实现代码保存在Zend/zend_hash.h和Zend/zend_hash.c这两个文件中，首先看HashTable的数据结构定义（Zend/zend_hash.h）：

```
typedef struct bucket {
    ulong h; /*用于存储key的Hash值*/
    uint nKeyLength;
    void*pData;
    void*pDataPtr;
    struct bucket*pListNext;
    struct bucket*pListLast;
    struct bucket*pNext;
    struct bucket*pLast;
    char arKey[1]; /*Must be last element*/
} Bucket;

typedef struct _hashtable {
    uint nTableSize;
    uint nTableMask;
    uint nNumOfElements;
    ulong nNextFreeElement;
    Bucket*pInternalPointer; /*Used for element traversal*/
```

```
Bucket*pListHead;
Bucket*pListTail;
Bucket**arBuckets;
dtor_func_t pDestructor;
zend_bool persistent;
unsigned char nApplyCount;
zend_bool bApplyProtection;
} HashTable;
```

在上面的数据结构定义中看到，Bucket（桶）和HashTable这两个结构体构成一个完整的HashTable。

1. Bucket结构体分析

先分析Bucket结构体中成员变量的作用：

ulong h: 保存经过hash函数处理之后的hash值。

uint nKeyLength: 保存索引（key）的长度。

void*pData: 指向要保存的内存块地址。

void*pDataPtr: 保存指针数据。

struct bucket*pListNext: 指向双向链表的下一个元素。

struct bucket*pListLast: 指向双向链表的上一个元素。

struct bucket*pNext: 指向具有同一个hash值的下一个元素。

struct bucket*pLast: 指向具有同一个hash值的上一个元素。

char arKey[1]: 保存索引（key），而且必须为最后一个成员。

在本节开始时说过，PHP的HashTable同时维护一个双向链表（至于为什么要同时维护一个双向链表，以后会说到），而这个双向链表就是通过pListNext和pListLast这两个成员变量维护的。

在这些成员变量中，pData和pDataPtr比较容易混淆。这两个成员到底有什么用呢？pData指向的是想要保存的内存块地址，一般是通过malloc之类的系统调用分配出来。但是有时候只想保存一个指针，如果这样也去调用malloc分配内存的话就会造成很多细小的内存块，从而导致产生内存碎片。这种情况PHP内核是不能容忍的，所以出现了pDataPtr指针。pDataPtr的作用就是当想要保存的数据是一个指针类型的数据时，就直接保存到pDataPtr成员变量中，而不用调用malloc分配内存，从而防止内存碎片的产生，然后pData直接指向pDataPtr。当要保存的数据不是指针时，pDataPtr被设置成NULL。

知道数据保存在哪里之后，怎么找到这个数据呢？答案是通过索引（key），每一个元素都有一个索引，且彼此不同。通过索引可以找到这个元素，并且取得保存在里面的数据。索引保存在哪里呢？注意Bucket结构的最后一个成员arKey，很明显就是保存在这里的。但是arKey只有一个字节，如果索引不止一个字节怎么办？这里PHP使用C语言的一个常用技巧（flexible array），通过申请sizeof（Bucket）+nKeyLength大小的内存（其中nKeyLength就是索引的长度），然后把索引保存到arKey成员中，而nKeyLength保存索引的长度。

但是如果索引是整数怎么办呢？本来也可以把整数当成字符串看待的，但是PHP不这样做，而是利用一个技巧解决。当索引是整数时，PHP把索引保存到Bucket结构体的h成员变量中，然后把nKeyLength设置为0，表示这个索引是一个整数而不是字符串。所以当nKeyLength大于0时，可以在arKey中取到索引，而nKeyLength等于0时，就在h中取得索引。当nKeyLength大于0时，h成员变量是不是没有用呢？其实不然，当nKeyLength大于0时（就是索引是字符串时），h成员保存的是索引经过hash函数处理之后的值。这样做的好处就是，在重hash时不用重新计算索引的hash值。

最后要说明的是，pNext和pLast指向的是具有同一个hash值的下一个元素和上一个元素，这是解决hash冲突的一种方法，俗称拉链法。

综合上面的分析，Bucket结构体应该如图7-11所示。

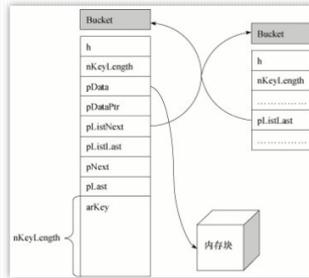


图 7-11 Bucket结构示意图

2.HashTable结构体分析

上面已经分析过Bucket结构，一个Bucket只能保存一个数据，而HashTable的目的就是通过索引（key）把每个元素分散到一个唯一的位置（当没有冲突时），这是怎么做到的呢？答案是通过hash算法把索引处理成一个int的数，然后定位到一个Bucket数组的其中一个元素中，如图7-12所示。

由图7-12所示可知，一个字符串的索引（图中的Key）经过hash函数处理之后会返回一个int的索引，通过int索引定位到Bucket数组的其中一个元素。字符串索引经过hash函数处理后，返回一个值为1的int索引，然后从Bucket数组中取得索引为1的元素（第二个元素），这就是HashTable的原理和实现方法。

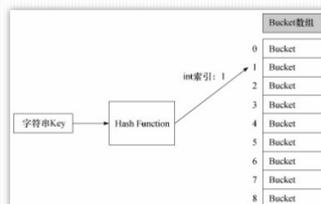


图 7-12 HashTable原理

PHP内核通过HashTable结构管理Bucket数组，下面列出HashTable结构体的主要成员变量的作用：

uint nTableSize：记录Bucket数组的大小。

uint nNumOfElements：记录HashTable中元素个数。

uint nNextFreeElement：下一个可用Bucket位置。

Bucket* pInternalPointer: 遍历HashTable元素。

Bucket* pListHead: 双链表表头。

Bucket* pListTail: 双链表表尾。

Bucket** arBuckets: Bucket数组。

arBuckets中保存的就是图7-12所示的Bucket数组，而nTableSize记录arBuckets数组的大小。现在可以想象PHP内核是怎样通过一个字符串索引定位到Bucket数组（arBuckets）中的一个元素的，代码如下：

```
h=hash ( key );
pos=h%nTableSize;
bucket=arBuckets[pos];
```

其中的hash就是把一个字符串处理成一个整数的函数，具体如下：

```
int hash ( char*key ) {
int h=0;
char*p=key;
while ( *p ) {
h+=*p;
p++;
}
return h;
}
```

上面的hash算法只展示了hash函数是怎么把一个字符串处理成一个整数，实际上PHP内核的hash算法会更复杂。

nNumOfElements记录HashTable中保存元素的个数，值得注意的是

nNumOfElements与nTableSize的区别，图7-13就展示了它们之间的区别。

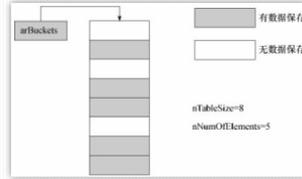


图 7-13 Bucket存储数据示意图

pListHead和pListTail分别是双向链表的表头和表尾指针。传统的HashTable是不会同时维护双向链表的，PHP这样做主要是用于有序遍历HashTable里的元素。图7-14展示了这个双向链表。

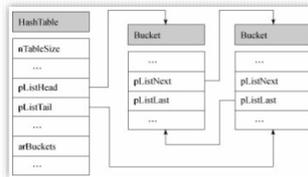


图 7-14 HashTable中的双向链表

至此，已经对PHP内核的HashTable有比较深入的了解，最后用图7-15展示PHP的Hash Table全貌。

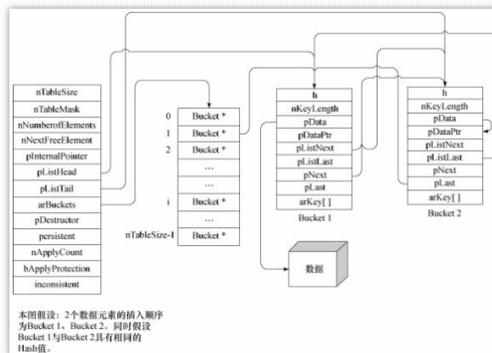


图 7-15 HashTable结构总览

7.4.2 HashTable的代码实现

本节主要介绍PHP内核是怎样实现HashTable的，下面的代码摘自Zend/zend_hash.c文件。有了前面对HashTable的分析，很容易理解下面的内容。

1.HashTable的初始化

PHP为HashTable的初始化提供了一个接口zend_hash_init，这个接口主要是把HashTable结构体的成员变量初始化，并且初始化桶数组（arBuckets），实现如下：

```
ZEND_API int zend_hash_init ( HashTable*ht, uint nSize,
hash_func_t pHashFunction, dtor_func_t pDestructor,
zend_bool persistent ZEND_FILE_LINE_DC )
{
    uint i=3;
    Bucket**tmp;
    SET_INCONSISTENT ( HT_OK );
    while ( ( 1U < i ) < nSize ) {
        i++;
    }
    ht->nTableSize=1 << i;
    ht->nTableMask=ht->nTableSize-1;
    ht->pDestructor=pDestructor;
    ht->arBuckets=NULL;
    ht->pListHead=NULL;
    ht->pListTail=NULL;
    ht->nNumOfElements=0;
    ht->nNextFreeElement=0;
    ht->pInternalPointer=NULL;
    ht->persistent=persistent;
    ht->nApplyCount=0;
    ht->bApplyProtection=1;
    /*Uses ecalloc ( ) so that Bucket*==NULL*/
    if ( persistent ) {
        tmp= ( Bucket** ) calloc ( ht->nTableSize, sizeof ( Bucket* ) );
        if ( ! tmp ) {
            return FAILURE;
        }
    }
}
```

```
    }
    ht->arBuckets=tmp;
    } else {
    tmp= ( Bucket**) ecalloc _rel ( ht->nTableSize, sizeof ( Bucket* ));
    if ( tmp ) {
    ht->arBuckets=tmp;
    }
    }
    return SUCCESS;
    }
```

参数nSize是要申请的桶数组大小，但这并不是PHP实际申请的大小，因为PHP内部会计算出一个不小于nSize并且是2的n次方的数作为实际申请的大小。

参数persistent判断是否使用PHP内存管理，如果为FALSE就是使用操作系统的内存管理，否则就是使用PHP内存管理。一般把这个参数设置为TRUE（就是使用PHP的内存管理），因为使用操作系统内存管理需要自己释放内存，容易造成内存泄漏，所以最好是交给PHP去管理内存。

2.HashTable的插入操作

向HashTable插入一个元素的第一步，就是要找到这个元素应该存放到arBuckets数组的哪个位置，前面已经分析过怎么根据索引定位到arBuckets数组的其中一个元素。因为要确保索引在HashTable中是唯一的，所以要比较相同hash值的链表上的所有元素是否有已经存在此索引，不存在才插入到HashTable中。实现如下：

```
ZEND_API int _zend_hash_add_or_update ( HashTable*ht, char*arKey, uint nKeyLength,
void*pData, uint nDataSize, void**pDest, int flag ZEND_FILE_LINE_DC )
{
    ulong h;
    uint nIndex;
    Bucket*p;
    IS_CONSISTENT ( ht );
    if ( nKeyLength <= 0 ) {
        return FAILURE;
    }
```

```

}
h=zend_inline_hash_func ( arKey, nKeyLength );
nIndex=h&ht->nTableMask;
p=ht->arBuckets[nIndex];
while ( p! =NULL ) {
if ( ( p->h==h ) && ( p->nKeyLength==nKeyLength ) ) {
if ( ! memcmp ( p->arKey, arKey, nKeyLength ) ) {
if ( flag&HASH_ADD ) {
return FAILURE;
}
HANDLE_BLOCK_INTERRUPTS ();
if ( ht->pDestructor ) {
ht->pDestructor ( p->pData );
}
UPDATE_DATA ( ht, p, pData, nDataSize );
if ( pDest ) {
*pDest=p->pData;
}
HANDLE_UNBLOCK_INTERRUPTS ();
return SUCCESS;
}
}
p=p->pNext;
}
p= ( Bucket* ) pemalloc ( sizeof ( Bucket ) -1+nKeyLength, ht->persistent );
if ( ! p ) {
return FAILURE;
}
memcpy ( p->arKey, arKey, nKeyLength );
p->nKeyLength=nKeyLength;
INIT_DATA ( ht, p, pData, nDataSize );
p->h=h;
CONNECT_TO_BUCKET_DLLIST ( p, ht->arBuckets[nIndex] );
if ( pDest ) {
*pDest=p->pData;
}
HANDLE_BLOCK_INTERRUPTS ();
CONNECT_TO_GLOBAL_DLLIST ( p, ht );
ht->arBuckets[nIndex]=p;
HANDLE_UNBLOCK_INTERRUPTS ();

```

```
ht->nNumOfElements++;
ZEND_HASH_IF_FULL_DO_RESIZE(ht); /*如果Hash Table的桶 ( buckets ) 满了, 扩大桶
的大小*/
return SUCCESS;
}
```

该函数以字符串作为索引, 所以必须判断nKeyLength (字符串长度) 是否大于0, 不是的话就直接退出。然后通过zend_inline_hash_func函数计算arKey的hash值h, 并与nTableMask进行按位“与”操作得到一个0 ~ nTableMask的整数nIndex, nIndex就是要插入的arBuckets数组的位置。

通过nIndex取得arBuckets数组的第nIndex个元素 (arBuckets[nIndex]), 这是一个具有相同hash值的双向链表, 通过遍历这个链表查看当中是否已经存在要插入的索引。如果存在并且要进行更新操作 (flags参数等于HASH_UPDATE), 就把旧值替换成新值, 并且释放旧值的内存。如果不存在要插入的索引, 就进行插入操作。以上两种情况都不是的话就会返回FAILURE (失败)。

插入过程如下:

步骤1 申请一个新的Bucket保存索引和值。

步骤2 把索引复制到arKey字段中。

步骤3 把pData指针指向值。

步骤4 把新Bucket添加到具有相同hash值的双向链表中 (主要通过宏CONNECT_TO_BUCKET_DLLIST完成)。

步骤5 把新Bucket添加到HashTable双向链表中 (通过CONNECT_TO_GLOBAL_DLLIST宏完成)。

步骤6 插入元素完毕后, 如果HashTable的元素个数大于arBuckets数组的大小 (nNumOfElements大于nTableSize), 将进行重hash操作 (HashTable的arBuckets数组大小翻倍)。

另外PHP内核提供很多用于插入/更新的函数，流程基本跟上面一样，所以这里就不再详细分析了，有兴趣的话可以参考Zend/zend_hash.c和Zend/zend_hash.h这两个文件。

3.HashTable的元素访问

元素访问是HashTable中最简单的操作，跟插入操作相似，只是把插入替换成取值而已，代码如下：

```
ZEND_API int zend_hash_find ( HashTable*ht, char*arKey, uint nKeyLength, void**pData )
{
    ulong h;
    uint nIndex;
    Bucket*p;
    IS_CONSISTENT ( ht );
    h=zend_inline_hash_func ( arKey, nKeyLength );
    nIndex=h&ht->nTableMask;
    p=ht->arBuckets[nIndex];
    while ( p! =NULL ) {
        if ( ( p->h==h ) && ( p->nKeyLength==nKeyLength ) ) {
            if ( ! memcmp ( p->arKey, arKey, nKeyLength ) ) {
                *pData=p->pData;
                return SUCCESS;
            }
        }
        p=p->pNext;
    }
    return FAILURE;
}
```

注意参数pData是个二级指针，用于保存取得的值。有时候我们会忘记这个参数是二级指针而发生错误，具体如下：

```
void*retval=NULL;
zend_hash_find ( ht, "name", strlen ( "name" ), retval ); /*错误*/
```

这样就会取不到数据，正确的应该是：

```
void*retval=NULL;
zend_hash_find ( ht, "name", strlen ( "name" ), &retval); /*正确*/
```

这个地方是值得注意的。

4.HashTable的元素删除

HashTable的元素删除也比较简单，主要调用zend_hash_del_key_or_index () 函数完成，代码实现跟元素的插入和访问差不多，所以这里就不再分析了，有兴趣可以看看源代码。

7.5 Zend API详解与扩展编写

虽然我们已了解了编写扩展的相关基础知识，但是还没有真正编写一个有用的扩展，从本节开始，就开始学习使用Zend API编写PHP扩展。

Zend引擎提供很多API扩展PHP的功能，有了这些API，就能很轻松地编写出我们的扩展，不过要编写稳定、快速的扩展，还要多写多练才能达到，下面首先学习Zend引擎。

7.5.1 什么是Zend引擎

Zend引擎是脚本语言引擎（解析器+虚拟机），主要的工作就是解析、翻译和执行PHP脚本。图7-16所示为Zend引擎的流程。

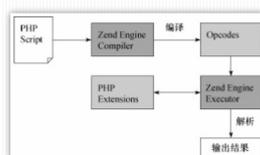


图 7-16 Zend引擎执行流程

从图7-16中可以看出Zend引擎要做两个工作：

编译PHP脚本，输出Opcodes。

解析执行Opcodes，输出结果。

在解析执行的过程中Zend引擎可以调用到所有已经载入到PHP环境的扩展库。

7.5.2 Zend引擎内存管理

PHP扩展是使用C语言开发的，而C语言最令人望而生畏的就是内存管理，因为有时候可能会忘记释放某些内存，或者释放同一内存块两次等，这些都是致命的错误。

为了防止这些情况出现，Zend引擎提供一些与内存管理相关的接口，使用这些接口后就不用管内存有没有释放，因为Zend引擎会为你管理所有通过Zend API申请的内存，从而避免了内存泄漏。表7-5列出了一些常用的内存管理接口。

接 口	描 述
<code>emalloc(size_t size)</code>	替代 <code>malloc</code> ，申请一块大小为 <code>size</code> 的内存块
<code>efree(void * ptr)</code>	替代 <code>free</code> ，释放 <code>ptr</code> 指向的内存块
<code>estrdup(char * str)</code>	替代 <code>strdup</code> ，申请一块跟 <code>str</code> 字符串一样大小的内存块，并复制 <code>str</code> 字符串到新的内存块中
<code>estrndup(char * str, int slen)</code>	替代 <code>strndup</code> ，跟 <code>estrdup</code> 差不多，不过此函数需要指定字符串的长度。此外， <code>estrndup</code> 函数比 <code>strndup</code> 要快并且是二进制安全的
<code>ecalloc(size_t numOfElem, size_t sizeOfElem)</code>	替代 <code>calloc</code> ，申请 <code>numOfElem</code> 份大小为 <code>sizeOfElem</code> 的内存块（也就是申请大小为 <code>numOfElem * sizeOfElem</code> 的内存块），并且把所有位置为0
<code>erealloc(void * ptr, size_t newSize)</code>	替代 <code>realloc</code> ，把 <code>ptr</code> 指向的内存块的大小扩展到 <code>newSize</code>

注意 上面提到的内存管理函数所有申请的内存仅对当前本地的请求有效，并且会在脚本执行完毕，处理请求终止时被释放。

7.5.3 PHP扩展的架构

至此已经有足够的基础编写扩展了。在接下来编写扩展的过程中，我们会把扩展对应的PHP代码也写出来，希望这样做能够让大家更容易接受。

在学习之前，先看一个扩展的架构，如代码清单7-1所示。

代码清单7-1 PHP扩展的架构

```
1.#include"php.h"
2.#include"php_myext.h"
3.static int le_myext;
4.PHP_FUNCTION ( confirm_myext_compiled );
5.zend_function_entry myext_functions[]= {
6.PHP_FE ( confirm_myext_compiled, NULL )
7. { NULL, NULL, NULL }
8. };
9.zend_module_entry myext_module_entry= {
10.STANDARD_MODULE_HEADER,
11."myext",
12.myext_functions,
13.PHP_MINIT ( myext ),
14.PHP_MSHUTDOWN ( myext ),
15.PHP_RINIT ( myext ),
16.PHP_RSHUTDOWN ( myext ),
17.PHP_MININFO ( myext ),
18."0.1",
19.STANDARD_MODULE_PROPERTIES
20. };
21.#ifdef COMPILE_DL_MYEXT
22.ZEND_GET_MODULE ( myext )
23.#endif
24.PHP_MINIT_FUNCTION ( myext )
25. {
26.
return SUCCESS;
27. }
```

```
28.PHP_MSHUTDOWN_FUNCTION ( myext )
29. {
30.
return SUCCESS;
31. }
32.PHP_RINIT_FUNCTION ( myext )
33. {
34.
return SUCCESS;
35. }
36.PHP_RSHUTDOWN_FUNCTION ( myext )
37. {
38.
return SUCCESS;
39. }
40.PHP_MINFO_FUNCTION ( myext )
41. {
42.php_info_print_table_start ();
43.php_info_print_table_header ( 2, "myext support", "enabled" );
44.php_info_print_table_end ();
45. }
46.PHP_FUNCTION ( confirm_myext_compiled )
47. {
48.char*arg=NULL;
49.int arg_len, len;
50.char*strg;
51.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC,
"s", &arg, &arg_len ) ==FAILURE ) {
52.
return;
53. }
54.len=sprintf (&strg, 0, "Hello%s, %s", "myext", arg );
55.RETURN_STRINGL ( strg, len, 0 );
56. }
```

如代码清单7-1所示，所有的PHP扩展通常都包含以下几个部分：

包含头文件（引入所需的宏、API定义等，第1~2行）；

声明导出函数（Zend函数块的声明，第4行）；

声明Zend函数块（第5~8行）；

声明Zend模块（第9~20行）；

实现get_module（）函数（第21~23行）；

实现导出函数（第46~56行）。

上面提到的这些结构在PHP扩展中都是必不可少的，PHP内核就是通过这些约定的规则与扩展通信的，如果想学会编写PHP扩展，首先必须了解这些结构。下面详细介绍这些结构。

1.包含头文件

在PHP扩展中，必须要包含的头文件只有一个php.h，它位于main目录下。这个文件包含构建扩展时所必需的各种宏和API定义。要包含这个头文件使用如下宏：

```
#include"php.h"
```

提示 应该为自己编写的扩展创建一个包含其特有信息的头文件。这个头文件应该包含php.h和所有导出函数的定义。如果使用ext_skel工具创建扩展的话，可能已经拥有这个头文件，因为ext_skel工具会自动创建。例如，使用ext_skel创建一个myext扩展，在这个扩展目录下就可能有个php_myext.h头文件，这就是上面所提到的头文件。

2.声明导出函数

编写扩展的目的是能够让PHP脚本调用扩展中的函数和类，那怎么才能让PHP脚本调用编写的扩展呢？答案就是声明和实现“导出函数”。

那什么是“导出函数”呢？“导出函数”其实就是按照PHP内核规定的准则编写的函数，形式如下：

```
void zif_ext_function (
int ht,
zval*return_value,
zval*this_ptr,
int return_value_used,
zend_executor_globals*executor_globals
);
```

PHP脚本就可以使用下面的代码调用上面的函数：

```
<? php
ext_function (……);
? >
```

因为“导出函数”形式都是固定的，所以Zend引擎提供一个方便的宏声明，具体如下：

```
ZEND_FUNCTION ( function_name );
```

function_name就是在PHP脚本中调用的函数名。ZEND_FUNCTION宏会把导出函数补充成以下形式：

```
void zif_function_name (
int ht,
zval*return_value,
zval*this_ptr,
int return_value_used,
zend_executor_globals*executor_globals
```

);

这样就可以免去烦琐的声明过程。

从上面“导出函数”声明的形式可以看出，“导出函数”没有返回值（void），其以“zif_”作为前缀，并且有五个参数。

导出函数的五个参数在编写扩展中有非常重要的作用，所以必须了解它们的作用，具体如下如表7-6所示。

参 数	描 述
ht	保存扩展函数参数的个数，但不应该直接访问这个值，而是通过 ZEND_NUM_ARGS() 来获取参数的个数。如 PHP 脚本代码 showmessage(\$arg1, \$arg2, \$arg3) 有 3 个参数，ht 等于 3
return_value	用来保存扩展函数向 PHP 脚本返回的值（访问这个变量的最佳方式也是用一系列的宏，后面会有详细说明）
this_ptr	根据这个参数可以访问该函数所在的对象（换句话说，此时这个函数应该是一个类的“方法”），推荐使用函数 getThis() 得到这个值
return_value_used	用来标识函数的返回值是否被脚本所使用，0 表示脚本不使用其返回值，而 1 则相反
Executor_globals	指向 Zend 引擎的全局设置，在创建新变量时很有用。在函数中使用宏 TSRMLS_FETCH() 引用这个值

3.声明Zend函数块

现在已经知道怎么声明导出函数，但Zend引擎是不会自动引入声明的导出函数的。那怎么才能把编写的函数引入到Zend引擎呢？答案就是zend_function_entry结构体。

先看zend_function_entry结构体的定义：

```
typedef struct _zend_function_entry {
    char*fname;
    void (*handler) (INTERNAL_FUNCTION_PARAMETERS);
    unsigned char*func_arg_types;
} zend_function_entry;
```

zend_function_entry结构体成员变量的作用如表7-7所示。

字 段	描 述
fname	指定在 PHP 脚本里调用的函数名（比如 fopen, mysql_connect, fsockopen 等）
handler	指向对应 C 函数的句柄，就是声明的导出函数句柄
func_arg_types	标识一些参数是否需要强制性按引用方式进行传递。通常应将其设定为 NULL

Zend引擎通过zend_function_entry结构数组把声明的导出函数引入内部，方法如下：

```
zend_function_entry myext_functions[] = {
    PHP_FE ( myext, NULL )
    { NULL, NULL, NULL }
};
```

Zend引擎在载入扩展时，自动把这个数组中的所有函数引入到函数表中，这样PHP脚本就可以调用这些函数了。

注意，zend_function_entry结构数组的最后一项是 { NULL, NULL, NULL }，Zend引擎就是靠这个判断导出函数列表是否完毕的，没有这一项的话，Zend引擎是不知道到哪里才结束的，从而导致内存溢出。导入过程大概如下：

```
int i;
for ( i=0; myext_functions[i].handler; i++ ) {
    /*把导入函数载入到内核函数表中*/
    .....
}
```

另外需要了解ZEND_FE宏，ZEND_FE宏会把zend_function_entry结构补充完整，例如：

```
PHP_FE ( myext, NULL )
```

会补充为：

```
{ "myext", zif_myext, NULL }
```

注意 从上面展开的形式中可以看出，可以手工把zend_function_entry结构填充完整，不过还是建议使用ZEND_FE宏来补充，因为以后Zend引擎发生改变的话，扩展还可以继续使用，而手工填充就不一定能够使用了。

4.声明Zend模块

PHP扩展的信息被保存在zend_module_entry结构中，这个结构包含所有需要向Zend引擎提供的模块信息，下面是zend_module_entry结构的定义：

```
typedef struct _zend_module_entry zend_module_entry;
struct _zend_module_entry {
    unsigned short size;
    unsigned int zend_api;
    unsigned char zend_debug;
    unsigned char zts;
    char*name;
    zend_function_entry*functions;
    int (*module_startup_func) (INIT_FUNC_ARGS);
    int (*module_shutdown_func) (SHUTDOWN_FUNC_ARGS);
    int (*request_startup_func) (INIT_FUNC_ARGS);
    int (*request_shutdown_func) (SHUTDOWN_FUNC_ARGS);
    void (*info_func) (ZEND_MODULE_INFO_FUNC_ARGS);
    char*version;
    .....//其余的一些我们不感兴趣的信息
};
```

表7-8列出了zend_module_entry结构体成员变量的作用。

字 段	描 述
size	通常用"STANDARD_MODULE_HEADER"填充，指定模块的四个成员：标识整个模块结构大小的 size，值为 ZEND_MODULE_API_NO 常量的 zend_api，标识是否为调试版本（使用 ZEND_DEBUG 进行编译）的 zend_debug，还有一个用来标识是否启用 ZTS（Zend 线程安全，使用 ZTS 或 USING_ZTS 进行编译）的 zts。
zend_api	
zend_debug	
zts	

(续)	
字段	描述
name	模块名称 (如 File functions, Socket functions, Crypt 等)。这个名字就是使用 <code>phpinfo()</code> 函数后在 "Additional Modules" 部分所显示的名称
functions	Zend 函数表的指针, 前面已经讨论过
module_startup_func	模块启动函数。仅在模块初始化时被调用, 通常用于一些与整个模块相关的初始化的工作 (比如申请初始化的内存等)。如果想表明模块函数调用失败或请求初始化失败请返回 FAILURE, 否则请返回 SUCCESS。可以通过宏 ZEND_MINIT 声明一个模块启动函数。如果不想使用, 请将其设定为 NULL
module_shutdown_func	模块关闭函数。仅在模块卸载时被调用, 通常用于一些与模块相关的反初始化的工作 (比如释放已申请的内存等)。这个函数和 <code>module_startup_func()</code> 相对应。如果想表明函数调用失败或请求初始化失败请返回 FAILURE, 否则请返回 SUCCESS。可以通过宏 ZEND_MSHUTDOWN 声明一个模块关闭函数。如果不想使用, 请将其设定为 NULL
request_startup_func	请求启动函数。在每次有页面请求时被调用。通常用于与该请求相关的初始化工作。如果想表明函数调用失败或请求初始化失败请返回 FAILURE, 否则请返回 SUCCESS。注意: 如果该模块是在一个页面请求中被动态加载的, 这个模块的请求启动函数将晚于模块启动函数的调用 (其实这两个初始化事件是同时发生的)。可以使用宏 ZEND_RINIT 声明一个请求启动函数。若不想使用, 请将其设定为 NULL
request_shutdown_func	请求关闭函数。在每次页面请求处理完毕后被调用, 与 <code>request_startup_func()</code> 相对应。如果想表明函数调用失败或请求初始化失败请返回 FAILURE, 否则请返回 SUCCESS。注意: 当在页面请求作为动态模块加载时, 这个请求关闭函数先于模块关闭函数的调用 (其实这两个反初始化事件是同时发生的)。可以使用宏 ZEND_RSHUTDOWN 声明这个函数。若不想使用, 请将其设定为 NULL
info_func	模块信息函数。当脚本调用 <code>phpinfo()</code> 函数时, Zend 会遍历所有已加载的模块, 并调用它们的这个函数。每个模块都有机会输出自己的信息。通常情况下, 这个函数用来显示一些环境变量或静态信息。可以使用宏 ZEND_MINFO 声明这个函数。若不想使用, 请将其设定为 NULL
version	模块的版本号。如果暂时还不想给某块设置一个版本号的话, 可以将其设定为 NO_VERSION_YET。但倘若在此添加一个字符单作为其版本号, 版本号通常类似: 2.5-dev, 2.5RC1, 2.5 或者 2.5p3 等
remaining_structure_elements	通常在模块内部使用。通常使用宏 STANDARD_MODULE_PROPERTIES 填充。注意, 也不应该将它们设定别的值

下面看一个例子:

```
zend_module_entry myext_module_entry = {
    STANDARD_MODULE_HEADER,
    "myext",
    myext_functions,
    PHP_MINIT ( myext ),
    PHP_MSHUTDOWN ( myext ),
    PHP_RINIT ( myext ),
    PHP_RSHUTDOWN ( myext ),
    PHP_MINFO ( myext ),
    "0.1",
    STANDARD_MODULE_PROPERTIES
};
```

在上面的例子中, 扩展命名为 "myext", 把导入函数列表设置为 `myext_functions`, 并设置相应的启动和关闭函数。

关于设置启动和关闭函数的宏如表7-9所示。

宏	描述
ZEND_MINIT(<i>module</i>)	声明一个模块的启动函数。函数名被自动设定为 <code>zend_init_<module></code> (比如 <code>zend_init_myext</code>)。通常与 <code>ZEND_MINIT_FUNCTION</code> 搭配使用
ZEND_MSHUTDOWN(<i>module</i>)	声明一个模块的关闭函数。函数名被自动设定为 <code>zend_mshutdown_<module></code> (比如 <code>zend_mshutdown_myext</code>)。通常与 <code>ZEND_MSHUTDOWN_FUNCTION</code> 搭配使用
ZEND_RINIT(<i>module</i>)	声明一个请求的启动函数。函数名被自动设定为 <code>zend_rinit_<module></code> (比如 <code>zend_rinit_myext</code>)。通常与 <code>ZEND_RINIT_FUNCTION</code> 搭配使用
ZEND_RSHUTDOWN(<i>module</i>)	声明一个请求的关闭函数。函数名被自动设定为 <code>zend_rshutdown_<module></code> (比如 <code>zend_rshutdown_myext</code>)。通常与 <code>ZEND_RSHUTDOWN_FUNCTION</code> 搭配使用
ZEND_MINFO(<i>module</i>)	声明一个输出模块信息的函数。用于 <code>phpinfo()</code> 。函数名被自动设定为 <code>zend_info_<module></code> (比如 <code>zend_info_myext</code>)。通常与 <code>ZEND_MINFO_FUNCTION</code> 搭配使用

我们在7.3节已经介绍过启动和关闭函数的实现，这里不再重复。

5.实现get_module()函数

当扩展被动态加载时会调用`get_module()`函数。下面先来看如何创建这个函数，具体如下（代码清单7-1中的第21~23行）：

```
#ifdef COMPILE_DL_MYEXT
ZEND_GET_MODULE(myext)
#endif
```

通过上面的代码可以看出，`get_module()`函数是通过`ZEND_GET_MODULE`宏创建的。`ZEND_GET_MODULE`宏定义如下：

```
#define ZEND_GET_MODULE(name) \
ZEND_DLEXPORT zend_module_entry*get_module(void) { \
return&name##_module_entry; \
}
```

从上面的宏定义可以看出，`ZEND_GET_MODULE`宏展开后就是`get_module()`函数，这个函数返回一个`zend_module_entry`指针。聪明的读者可能会发现，这个指针就是上面定义的Zend模块。这样PHP内核就可以通过调用`get_module()`函数取得编写的扩展的信息。现在终于知道PHP内核跟扩展之间的通信渠道是什么了，就是`get_module()`函数。

注意 使用ext_skel工具创建的扩展框架会包含get_module()函数，所以一般不用手动添加。

另外注意到get_module()函数被一条件编译宏包围着，说明get_module()函数应该在某些情况下不会被实现的。到底什么情况不会被实现呢？答案是：当扩展被编译成一个内建的模块时，get_module()函数不会被实现。

6.实现导出函数

导出函数的实现是构建扩展的最后一步。导出函数是可以在PHP脚本中调用的函数，而实现这些函数才能发挥作用。下面学习怎么实现导出函数，如代码清单7-1中第46~56行所示：

```
PHP_FUNCTION ( confirm_myext_compiled )
{
    char*arg=NULL;
    int arg_len, len;
    char*strg;
    if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC,
    "s", &arg, &arg_len ) ==FAILURE ) {
        return;
    }
    len=sprintf (&strg, 0, "Hello%s, %s", "myext", arg);
    RETURN_STRINGL ( strg, len, 0);
}
```

实现导出函数跟声明导出函数一样都是使用PHP_FUNCTION宏（前面已经说过）定义一个导出函数，然后在函数体里面实现想要的功能。

注意 在声明的Zend函数块中的所有函数都必须实现，不然会出现严重的错误。例如，在Zend函数块中声明一个ZEND_FE (show_message, NULL)函数，就必须实现ZEND_FUNCTION (show_message)函数。

至此，已经介绍了PHP扩展结构的全部内容，现在应该知道代码清单7-1中每一行代

码的作用了。接下来就是学习怎么在导出函数中实现我们需要的功能。

7.5.4 接收用户传递的参数

一般来说，编写扩展的目的是希望使用C代码层来处理我们提供的数据。所以对于扩展来说，最重要的事情就是如何接收和处理那些通过函数参数传递过来的数据。扩展层是怎么取得PHP语言层传递过来的参数呢？下面详细说明。

1.取得参数个数

因为PHP是支持可变参数的，所以参数的个数在没有被调用之前是不确定的。那就不可能在声明导出函数的时候同时声明参数列表，所以不得不在导出函数内部取得参数的个数。可以使用ZEND_NUM_ARGS宏实现。其实这个宏等于参数ht，在讲解导出函数时已经说过这个参数的作用。

ZEND_NUM_ARGS宏定义如下（在Zend/zend_API.h文件中）：

```
#define ZEND_NUM_ARGS () (ht)
```

下面的代码展示了判断传入的参数个数是否正确的方法：

```
if (ZEND_NUM_ARGS () != 3) WRONG_PARAM_COUNT;
```

如果没有传递3个参数给该函数，就会打印一个错误消息，并且退出该函数。在上面的代码中，使用一个宏WRONG_PARAM_COUNT，主要作用是打印一个类似“Warning: Wrong param

eter count for myext () in/home/www/myext. php on line 3”的错误信息，然后返回到PHP内核状态。可以在Zend/zend_API.h中找到宏的定义：

```
ZEND_API void wrong_param_count ( void );
```

```
#define WRONG_PARAM_COUNT { wrong_param_count (); return; }
```

从以上的代码可以看到，WRONG_PARAM_COUNT宏展开之后调用一个内部函数wrong_param_count()，然后返回到PHP内核状态。wrong_param_count()函数是打印一个警告信息。至于怎么打印错误信息，后面有详细的说明。

2.取得参数实体

取得参数的实体才是我们需要的。跟取得参数的个数一样，不能通过导出函数的参数列表取得，所以Zend引擎提供一个取得函数实体的API，这个API的声明如下：

```
int zend_parse_parameters ( int num_args TSRMLS_CC,  
char*type_spec, ……);
```

第一个参数num_args表明想要接收的参数个数，一般使用ZEND_NUM_ARGS()宏表示获取所有传递过来的参数。第二个参数必须是宏TSRMLS_CC，这个宏会根据编译时是否启用线程安全来传递参数，在没有启动线程安全时，这个宏会被忽略掉，而当启动线程安全时，这个宏会被替换成tsrm_ls。第三个参数type_spec是一个字符串，用来指定所期待接收的各个参数的类型。剩下的参数就是用来接收PHP语言层传递过来的参数的变量指针。

zend_parse_parameters()在接收参数时，会尽量地把参数的类型转换成你期望的类型。例如，你期望得到一个字符串的参数，但是用户却传递你一个整型的参数，zend_parse_parameters()会帮把这个整型的参数转换成字符串的参数。

注意 任何一种标量类型都可以转换成另外一种标量类型（如字符串、整型、浮点型等），但是不能在标量类型与复杂类型（如数组、对象和资源等）之间进行转换。

如果zend_parse_parameters()函数成功地接收了参数，并在转换参数类型的过程中没有发生错误，这个函数会返回SUCCESS，否则就返回FAILURE。

现在讲解type_spec参数。type_spec参数是一个字符串，跟printf()函数的格式化参数(第一个参数)很相似，都是用来指定后面参数的类型的。不过值得庆幸的是，type_spec参数比printf()的格式化参数简单。这个参数的使用方法如下：

```
long lptr;
double dptr;
zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "ld", &lptr, &dptr );
```

上面的代码中，type_spec参数被设置为“ld”，表示要接收一个长整型(long)的参数和一个浮点型(double)的参数，其中“l”代表long，“d”代表double。

在type_spec参数之后，使用两个变量lptr和dptr保存传递过来的参数。要注意的是，用来保存参数的变量的类型要与在type_spec设置的类型相一致。

下面列举可以指定接收的参数类型：

l: 长整型。

d: 双精度浮点类型。

s: 字符串和其长度(要使用两个变量保存)。

b: 布尔型。

r: 资源，保存在zval*中。

a: 数组，保存在zval*中。

o: (任何类的)对象，保存在zval*中。

O: (由class entry指定的类的)对象，保存在zval*中。

z: 实际的zval*。

下面的一些字符在类型说明字符串（就是那个char*type_spec）中具有特别的含义：

|: 表明剩下的参数都是可选参数。如果用户没有传进来这些参数值，这些值就会被初始化成默认值。

/: 表明参数解析函数将会对剩下的参数以SEPARATE_ZVAL_IF_NOT_REF()的方式提供这个参数的一份拷贝，除非这些参数是一个引用。

!: 表明剩下的参数允许被设定为NULL（仅用在a、o、O、r和z上）。如果用户传进来一个NULL值，则存储该参数的变量将会设置为NULL。

为了让大家对zend_parse_parameters()函数有更深入的了解，下面举几个例子。

例1 取得一个字符串和一个布尔值：

```
int slen;
char*str;
zend_bool b;
if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC,
"sb", &str, &slen, &b) ==FAILURE ) {
return;
}
```

在上面的代码中，使用"sb"指定要传递一个字符串的参数和一个布尔值的参数。值得注意的是，在接收字符串时，除了要接收字符串的地址之外，还要接收字符串的长度。如上例中，使用str保存字符串的地址，用slen保存字符串的长度。

例2 取得一个数组和一个可选的长整型参数：

```
zval*array;
long l;
```

```
if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC,
    "a|!", &array, &l) ==FAILURE ) {
    return;
}
```

上面的代码中，使用"a|!"指定要传递一个数组和一个长整型的参数。在“a”之后使用竖杠，表示竖杠之后的参数为可选，在调用时可以不用传递这些参数。如果不传递这些参数，PHP内核会自动设置为默认值。例如可以这样调用：

```
<? php
myext ( array ( 1, 2, 3 )); //忽略第二个参数
myext ( array ( 1, 2, 3 ), 10); //传递第二个参数
? >
```

例3 取得一个对象或者NULL：

```
zval*object;
if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC,
    "o! ", &object) ==FAILURE ) {
    return;
}
```

在上面的代码中，使用"o! "指定要传递的一个对象。另外，使用感叹号修饰这个参数，表示如果传递进来的是一个空对象，object就会被设置为NULL。

3.接收可变参数

PHP支持可变参数，可以使用func_get_args ()和func_num_args ()这两个函数实现，代码如下：

```
<? php
```

```
function mutative () {
    $total=0;
    $nums=func_num_args ();
    $arglist=func_get_args ();
    for ($i=0; $i<$nums; $i++) {
        $total+= $arglist[$i];
    }
    return $total;
}
echo mutative (1, 2, 3, 4, 5, 6); //输出21
? >
```

上面的代码实现了可变参数功能，`func_num_args()` 函数可以取得参数的数量，`func_get_args()` 函数可以取得所有的参数，并放到一个数组里面。通过这两个函数就可以很简单地实现可变参数。

既然在PHP语言层可以轻松实现可变参数，那么在扩展中怎样可以实现可变参数呢？如果在PHP内核也提供一个跟`func_get_args()` 函数一样的接口，那就容易多了。幸运的是，PHP内核中真的有一个跟`func_get_args()` 函数差不多的接口，这个接口如下：

```
int zend_get_parameters_array_ex (int param_count,
    zval***argument_array TSRMLS_DC);
```

第一个参数`param_count`表示要接收的参数个数，第二个参数`argument_array`用于保存参数列表的数组。跟`func_get_args()` 函数一样，这个接口也是取得一个参数列表的数组。下面是`zend_get_parameters_array_ex()` 的使用方法：

```
zval**parameter_list[3];
int num_args=ZEND_NUM_ARGS ();
if ( num_args>3 ) {
    WRONG_PARAM_COUNT;
```

```
}  
if ( zend_get_parameters_array_ex ( num_args,  
parameter_list ) ==FAILURE ) {  
return;  
}
```

上面代码中，首先判断参数的个数是否小于等于3个，因为保存参数的数组只有3个元素，如果参数大于3个就会出现段错误这种致命的错误。所以为了防止段错误，必须先判断参数个数是否小于等于3个。如果参数个数符合要求，就可以使用`zend_get_parameters_array_ex()`函数接受参数列表。

在接收到参数列表之后（保存在`parameter_list`中）就可以使用它，代码如下：

```
switch ( num_args ) {  
case 3:  
php_printf ( "Third parameter is: %s \n", *parameter_list[2] );  
case 2:  
php_printf ( "Second parameter is: %s \n", *parameter_list[1] );  
case 1:  
php_printf ( "First parameter is: %s \n", *parameter_list[0] );  
break;  
}
```

上面的代码中使用一个`switch`处理参数列表。`switch`结构的特点是：如果没有使用`break`跳出这个`case`，后面的`case`也会执行。上面的代码中都没有使用`break`跳出本次`case`，所以如果传入3个参数的话，会执行全部`case`，如果传入2个参数的话，就执行除`case3`之外的所有`case`。

4. 参数类型转换

前面学习了怎么接收可变参数，但是细心的读者会发现，接收的参数都没有类型限制的，就是说可以传递任何类型的参数。这样就出现问题了，如果希望第一个参数传入的是数组，但是用户却传递了一个字符串，那会出现什么问题呢？如果使用处理数组的函数去

处理字符串的话，会出现严重的错误。所以要杜绝这种情况的出现。有什么办法去解决这种情况呢？要解决这种情况，可以使用表7-10所列的参数类型转换函数。

函 数	描 述
<code>convert_to_boolean_ex()</code>	强制转换为布尔类型。若原来是布尔值则保留，不做改动。长整型值 0、双精度型值 0.0、空字符串或字符串 "0"、还有空值 NULL 都将被转换为 FALSE（本质上是一个整数 0）；数组和对像若为空则转换为 FALSE，否则转换为 TRUE；除此之外的所有值均转换为 TRUE（本质上是一个整数 1）。
<code>convert_to_long_ex()</code>	强制转换为长整型。这也是默认的整数类型。如果原来是空值 NULL、布尔型、资源、当然还有长整型，则其值保持不变（因为本质上都是整数 0）。双精度型则被简单取整。包含有一个整数的字符串将会被转换为对应的整数，否则转换为 0。空的数组和对像将被转换为 0，否则将被转换为 1。
<code>convert_to_double_ex()</code>	强制转换为一个双精度型。这是默认的浮点数据类型。如果原来是空值 NULL、布尔值、资源和双精度型则其值保持不变（只变一个变量类型）。包含有一个数字的字符串将被转换成相应的数字，否则被转换为 0.0。空的数组和对像将被转换为 0.0，否则将被转换为 1.0。
<code>convert_to_string_ex()</code>	强制转换为字符串。空值 NULL 将被转换为空字符串。布尔值 TRUE 将被转换为 "1"，FALSE 则被转换为一个空字符串。长整型和双精度型会分别转换为对应的字符串。数组将会被转换为字符串 "Array"，而对像则被转换为字符串 "Object"。
<code>convert_to_array_ex (value)</code>	强制转换为数组。若原来就是一数组则不改动。对像将被转换为一个以其属性为键名，以其属性值为键值的数组。空值 NULL 将被转换为一个空数组。除此之外的所有值都将被转换为仅有一个元素（下标为 0）的数组，并且该元素即为该值。
<code>convert_to_object_ex (value)</code>	强制转换为对像。若原来就是对像则不改动。空值 NULL 将被转换为一个空对像。数组将被转换为一个以其键名为属性、键值为其属性值的对像。其他类型则被转换为一个具有 "value" 属性的对像，"value" 属性的值即为该值本身。
<code>convert_to_null_ex (value)</code>	强制转换为空值 NULL。

注意 表中所有参数类型转换函数都是以一个**zval作为参数的。

下面举个例子：

```

zval**parameter[1];
if ( ZEND_NUM_ARGS () != 1 ) {
    WRONG_PARAM_COUNT;
}
if ( zend_get_parameters_array_ex ( 1, parameter ) == FAILURE ) {
    return;
}
convert_to_long_ex ( parameter[0] );
php_printf ( "After change: %ld \n", Z_LVAL_PP ( parameter[0] ));

```

在上例中，接收到参数之后就使用`convert_to_long_ex()`函数把参数类型转换成长整型。如果传递的参数是字符串"10HELLO"，就会被转换成长整型10。

5.处理通过引用传递的参数

PHP可以通过引用传递的参数，例如：

```
<? php
```

```
function swap (&$a, &$b) {  
    $tmp=$b;  
    $b=$a;  
    $a=$tmp;  
}  
$a=10;  
$b=12;  
swap ($a,$b);  
? >
```

因为修改通过引用传递的参数会改变原来变量的数据，所以在扩展里修改通过引用传递过来的参数时要特别的小心。

本章7.3.3节已经了解过引用计数器。其实为了节省内存，PHP内核在传递参数时也使用引用计数的方式，所以不管是否使用引用传递参数，它们指向的值都是相同的（就是指向同一块内存块），不同的是is_ref字段（zval结构的is_ref字段）。引用传递的参数is_ref字段为1，而非引用传递的参数is_ref字段为0。

这样，当要修改引用传递过来的参数时，可以直接修改参数的值（因为参数的值跟原来变量的值是相同的，也就是说内存地址是一样的），就可以修改原来变量的值。但是当参数不是使用引用传递过来的，而又要修改它的值的话就可能要出现问题了，因为PHP规定修改非引用传递的参数值是不会影响原来变量的值，但是PHP内核却是使用引用传递参数的，所以修改参数的值必定会影响到原来变量的值。为了解决这个问题，PHP内核中实现“zval分离”。

“zval分离”就是写时复制，即当需要修改非引用传递的参数值时，从原来变量中复制一份新的值，并让参数指向这个新的值。这样修改参数的值时，就不会影响到原来变量的值。

使用PZVAL_IS_REF (zval*) 宏可判断传递过来的参数是否使用引用方式，而使用SEPARATE_ZVAL (zval**) 宏可以分离一个zval。

使用PZVAL_IS_REF宏的例子如下：

```
zval*parameter;
```

```
if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "z",
&parameter ) ==FAILURE ) {
return;
}
if (! PZVAL_IS_REF ( parameter )) {
zend_error ( E_WARNING, "Parameter wasn't passed by reference");
RETURN_NULL ();
}
```

PZVAL_IS_REF宏判断传递过来的参数是否通过引用方式。它的定义很简单，具体如下：

```
#define PZVAL_IS_REF ( z ) (( z )->is_ref)
```

上边这条语句主要就是判断参数的is_ref字段是否为真。

使用SEPARATE_ZVAL宏的例子如下：

```
zval**parameter;
zend_get_parameters_ex ( 1, &parameter );
SEPARATE_ZVAL ( parameter );
```

因为SEPARATE_ZVAL()宏会通过emalloc()函数申请一个新的zval，所以这也就意味着，如果不主动释放这段内存的话，它就会直到脚本结束时才会被释放。所以如果大量调用这个宏而没有主动释放的话，可能会把内存耗光。

通过SEPARATE_ZVAL()宏分离过的参数，可以放心地修改它的值，因为分离的参数已和任何的变量都没有关系了。

7.5.5 在PHP扩展中创建变量

在编写PHP自定义函数时，一般都会用到自己创建的一些变量，所以在编写扩展时我们也希望创建自定义的变量。在扩展中创建自定义变量有些麻烦，下面就介绍怎么在扩展里创建变量。

1.创建局部变量

在自定义函数中，只要不是使用global关键字声明的变量都是局部变量，例如下面代码：

```
1.<? php
2.error_reporting ( E_ALL );
3.
4.function myfunc ( ) {
5.    $msg="this is my function";
6.    echo $msg;
7.}
8.
9.myfunc ( );
10.echo $msg;
11.? >
```

上面的代码中，在myfunc () 函数中创建一个局部变量msg，在第9行调用这个函数，然后在第10行打印变量msg，此时会发现在第10行出现错误，如图7-17所示。



图 7-17 打印局部变量出错

从图7-17中可以看出，在自定义函数中创建的局部变量不能在全局环境中使用，否则会提示变量没有定义。

那在编写扩展时，应该怎样去创建一个局部变量呢？要创建一个能够被PHP脚本访问

的局部变量，要先创建一个zval容器，然后对这个zval容器进行必要的填充，最后把它引入到Zend引擎的内部符号表中，代码如下：

```
zval*new_var;
/*申请并初始化一个新的zval容器*/
MAKE_STD_ZVAL ( new_var );
/*将“new_var”变量引入到当前活动符号表中*/
ZEND_SET_SYMBOL ( EG ( active_symbol_table ), "new_var", new_var );
/*现在就可以在脚本中用$new_var访问这个变量了*/
```

注意 符合表就像一本字典。在PHP内核中，所有的变量都是保存在这个符号表中，PHP内核就是在这个符号表中查找相应的变量的。

MAKE_STD_ZVAL () 宏会通过ALLOC_ZVAL () 宏申请一个新的zval容器的内存空间，并且调用INIT_PVAL () 宏初始化这个zval容器。MAKE_STD_ZVAL 宏代码如下：

```
#define MAKE_STD_ZVAL ( zv ) \
ALLOC_ZVAL ( zv ); \
INIT_PZVAL ( zv );
```

INIT_PVAL () 宏负责把zval容器的引用计算器 (refcount) 置为1，然后把引用标识设置为0 (即把is_ref设置为0)。INIT_PVAL () 宏代码如下：

```
#define INIT_PZVAL ( z ) \
( z )->refcount=1; \
( z )->is_ref=0;
```

当创建一个zval容器之后，必须把它引入符号表中才能在PHP脚本中使用。ZEND_

`SET_SYMBOL()` 宏负责把新建的变量引入到Zend引擎内部的符号表中。这个宏首先会检查这个变量是否已经存在于符号表中，如果已经存在则将其替换成另一个引用变量（同时会自动销毁原有的zval容器）。

`ZEND_SET_SYMBOL()` 宏是通过EG宏访问Zend执行器（执行Opcodes的部分）的全局结构的。如果使用的是EG（`active_symbol_table`），则可以访问当前的活动符号表（通常就是局部变量表），从而可以处理一些局部变量。

如果你很在意程序的运行效率，并且不太在乎内存的话，可以使用`zend_hash_update()`函数跳过检查变量名是否存在于符号表中，而强行将变量名插入到符号表中，代码如下：

```
zval*new_var;
MAKE_STD_ZVAL(new_var);
zend_hash_update(
    EG(active_symbol_table),
    "new_var",
    sizeof("new_var"),
    &new_var,
    sizeof(zval*),
    NULL
);
```

2.创建全局变量

其实全局变量跟局部变量在PHP内核中看是一样的，不一样的只是它们保存在不同的符号表中，局部变量保存在`active_symbol_table`中，而全局变量就保存在`symbol_table`中。所以可以使用上面的代码，只要把`active_symbol_table`替换成`symbol_table`就可以了，代码如下：

```
zval*new_var;
MAKE_STD_ZVAL(new_var);
```

```
ZEND_SET_SYMBOL (&EG ( symbol_table ), "new_var", new_var);
```

注意 除了符号表把`active_symbol_table`替换成`symbol_table`之外，应该还要注意`active_symbol_table`是一个指针，而`symbol_table`不是。所以还需要增加取地址操作“`&EG (symbol_table)`”，因为`ZEND_SET_SYMBOL`宏需要一个指针作为参数。

下面编写一个导出函数，主要工作是创建一个长整型的全局变量`new_var`，代码如下：

```
PHP_FUNCTION ( myext )
{
    zval*new_var;
    MAKE_STD_ZVAL ( new_var );
    ZVAL_LONG ( new_var, 10 );
    ZEND_SET_GLOBAL_VAR ( "new_var", new_var );
}
```

在上面导出函数中创建了一个`new_var`的全局变量，所以在PHP脚本中就可以使用这个变量了，代码如下：

```
<? php
error_reporting ( E_ALL );
myext ();
var_dump ( $new_var );
? >
```

输出如图7-18所示。



图 7-18 在扩展里创建全局变量

值得注意的是，必须先调用myext()函数，因为变量new_var就是在这个函数里面创建的，所以调用它才会创建全局变量new_var；如果不调用myext()函数就直接打印new_var变量，会发生错误（提示变量new_var没有定义），如图7-19所示。



图 7-19 没定义变量的错误提示

注意 如果上面的例子在VC++6.0中编译不通过，提示“unresolved external symbol _executor_globals_id”错误，可以按以下步骤处理，选择“工程”→“设置”→“C/C++”菜单命令，在“预处理程序定义”中把LIBZEND_EXPORTS选项删除即可。

7.5.6 在PHP扩展中为变量赋值

变量的作用就是存放数据。在PHP中变量不但保存着值，还保存着类型，所以不但要为变量赋值，同时还要为变量设置类型。下面学习如何为变量赋值。

1.长整型（整型）类型变量

在PHP内核中，整数全部都是长整型（long），其值的存储方法也是非常简单的。前面讨论过zval容器的结构，当中的value字段就是保存数据的。所有整数都是直接保存在这个联合体中的lval字段中，相应的数据类型（type字段）为IS_LONG，代码如下：

```
zval*new_var;  
MAKE_STD_ZVAL ( new_var );  
new_var->value.lval=12;  
new_var->type=IS_LONG;
```

但是为了兼容性，最好使用ZVAL_LONG宏赋值：

```
zval*new_var;  
MAKE_STD_ZVAL ( new_var );  
ZVAL_LONG ( new_var, 12);
```

上面的例子跟以下的PHP代码效果相同：

```
<? php  
$new_var=12;  
? >
```

2.双精度（浮点数）类型变量

跟赋值整数差不多，为变量赋值浮点型时，只需把数据直接存放在value联合体的dval字段中，然后把对应的数据类型设置为IS_DOUBLE即可，代码如下：

```
zval*new_var;  
MAKE_STD_ZVAL ( new_var );  
new_var->value.dval=12.56;  
new_var->type=IS_DOUBLE;
```

同样为了兼容性，最好使用ZVAL_DOUBLE宏赋值：

```
zval*new_var;  
MAKE_STD_ZVAL ( new_var );  
ZVAL_DOUBLE ( new_var, 12.56 );
```

上面的例子跟以下的PHP代码效果相同：

```
<? php  
$new_var=12.56;  
? >
```

3.字符串类型变量

为变量赋字符串类型的值时可能会比较麻烦。除了保存字符串的值外，还要保存字符串的长度（提供给strlen（）等函数使用）。赋值时，字符串的值保存在zval.value联合体的str结构体中的val字段中，而字符串长度保存在str结构体中的len字段中，并且把相应的数据类型设置为IS_STRING。

另外值得注意的是，用来保存字符串值的内存块应该使用Zend引擎内存管理函数去申请，这样可以避免自己管理这些内存，还可以让Zend引擎处理起来更方便，代码如下：

```
zval*new_var;
char*str="This is a new string variable";
MAKE_STD_ZVAL ( new_var );
new_var->value.str.len=strlen ( str );
new_var->value.str.val=estrndup ( str );
new_var->type=IS_STRING;
```

请注意，上面使用`estrndup()`函数创建一块新的内存块，`estrndup()`函数是Zend引擎的内存管理函数。还可以使用`ZVAL_STRING`宏完成上面的工作：

```
zval*new_var;
char*str="This is a new string variable";
MAKE_STD_ZVAL ( new_var );
ZVAL_STRING ( new_var, str, 1);
```

`ZVAL_STRING`宏的第三个参数指明该字符串是否需要被复制（使用Zend引擎的内存管理函数）。当设置为1时，将会复制第二个参数指向的字符串。如果设置为0时，直接把`val`字段指向第二个参数指向的字符串。

如果只想取得字符串的一部分或者已经知道字符串的长度，可以使用宏`ZVAL_STRINGL (zval, string, length, duplicate)`完成这项工作，参数`length`就是指定字符串的长度。`ZVAL_STRINGL`宏比`ZVAL_STRING`宏快，`ZVAL_STRINGL`宏的定义如下：

```
#define ZVAL_STRINGL ( z, s, l, duplicate ) { \
char*s= ( s ); int l=l; \
(z)->value.str.len=l; \
(z)->value.str.val= ( duplicate? estrndup ( s,l): s ); \
(z)->type=IS_STRING; \
}
```

从定义可以看出，字符串的长度（len字段）被直接设置成length参数的值，所以不用使用strlen（）函数去计算字符串的长度。

如果想创建一个空字符串，可以将其长度设置为0，并且把empty_string作为字符串的值即可：

```
zval*new_var;
MAKE_STD_ZVAL ( new_var );
new_var->value.str.len=0;
new_var->value.str.val=empty_string;
new_var->type=IS_STRING;
```

或者可以使用ZVAL_EMPTY_STRING宏完成：

```
zval*new_var;
MAKE_STD_ZVAL ( new_var );
ZVAL_EMPTY_STRING ( new_var );
```

4.布尔类型变量

布尔类型变量的赋值跟长整型的差不多，只是把数据类型字段设置为IS_BOOL，并且lval字段的值只能是0或者1，代码如下：

```
zval*new_var;
MAKE_STD_ZVAL ( new_var );
new_var->value.lval=1;
new_var->type=IS_LONG;
```

同样，也可以使用ZVAL_BOOL宏完成这项工作，或者可以直接使用ZVAL_TRUE和ZVAL_FALSE宏直接将其值设置为TRUE和FALSE：

```
zval*new_var;  
MAKE_STD_ZVAL (new_var);  
ZVAL_BOOL (new_var, 1); /*或者ZVAL_BOOL (new_var, 0)*/
```

5.数组类型变量

数组在PHP中扮演着重要的角色，可以说PHP的强大全是因为数组的灵活。而在PHP内核中，数组是使用哈希表（HashTable）存储的。在7.4节中详细介绍了PHP的HashTable，所以现在学习数组就比较简单了。

在为变量赋值数组类型时，先要创建一个HashTable，然后将其保存在zval.val容器的ht字段中。对于这项工作，Zend引擎提供了一个简单的接口——array_init()。代码如下：

```
zval*new_var;  
MAKE_STD_ZVAL (new_var);  
array_init (new_array);
```

其实上面的代码就是创建一个空的数组，效果跟下面的PHP代码相同：

```
<? php  
$new_var=array ();  
? >
```

创建一个空的数组之后，就可以往数组中添加元素了。往数组添加元素有一系列的

API可以使用，下面列出可以使用的所有API（所有的这些API在调用成功时都是返回SUCCESS，调用失败时都是返回FAILURE）：

关联数组的API如表7-11所示。

表 7-11 关联数组的 API

函 数	描 述
add_assoc_long (zval * array, char * key, long n);	添加一个长整型元素 相当于\$array["key"] = 10;
add_assoc_unset (zval * array, char * key);	添加一个 unset 元素 相当于\$array["key"] = NULL;
add_assoc_bool (zval * array, char * key, int b);	添加一个布尔值 相当于\$array["key"] = TRUE;
add_assoc_resource (zval * array, char * key, int r);	添加一个资源 相当于\$array["key"] = 6021;
add_assoc_double (zval * array, char * key, double d);	添加一个浮点值 相当于\$array["key"] = 10.12;
add_assoc_string (zval * array, char * key, char * str, int duplicate);	添加一个字符串，duplicate 表明这个字符串是否被复制到 Zend 引擎的内部内存 相当于\$array["key"] = "string";

(续)

函 数	描 述
add_assoc_stringl (zval * array, char * key, char * str, uint length, int duplicate);	添加一个指定长度的字符串 相当于 add_assoc_string()
add_assoc_zval (zval * array, char * key, zval * value);	添加一个 zval 结构。这在添加另外一个数组、对象或流等数据时会有用 相当于\$array["key"] = \$value;

索引数组的API如表7-12所示。

表 7-12 索引数组的 API

函 数	描 述
add_index_long (zval * array, uint idx, long n);	添加一个长整型元素 相当于\$array[0] = 10;
add_index_unset (zval * array, uint idx);	添加一个 unset 元素 相当于\$array[0] = NULL;
add_index_bool (zval * array, uint idx, int b);	添加一个布尔值 相当于\$array[0] = TRUE;
add_index_resource (zval * array, uint idx, int r);	添加一个资源 相当于\$array[0] = 6021;
add_index_double (zval * array, uint idx, double d);	添加一个浮点值 相当于\$array[0] = 10.12;
add_index_string (zval * array, uint idx, char * str, int duplicate);	添加一个字符串，duplicate 表明这个字符串是否被复制到 Zend 引擎的内部内存 相当于\$array[0] = "string";
add_index_stringl (zval * array, uint idx, char * str, uint length, int duplicate);	添加一个指定长度的字符串 其余跟 add_index_string() 相同
add_index_zval (zval * array, uint idx, zval * value);	添加一个 zval 结构。这在添加另外一个数组、对象或流等数据时会有用 相当于\$array[0] = \$value;
add_next_index_long (zval * array, long n);	添加一个长整型元素 相当于\$array[] = 10;
add_next_index_unset (zval * array);	添加一个 unset 元素 相当于\$array[] = NULL;
add_next_index_bool (zval * array, int b);	添加一个布尔值 相当于\$array[] = TRUE;
add_next_index_resource (zval * array, int r);	添加一个资源 相当于\$array[] = 6021;
add_next_index_double (zval * array, double d);	添加一个浮点值 相当于\$array[] = 10.12;
add_next_index_string (zval * array, char * str, int duplicate);	添加一个字符串，duplicate 表明这个字符串是否被复制到 Zend 引擎的内部内存 相当于\$array[] = "string";

(续)

函 数	描 述
add_next_index_stringl (zval * array, char * str, uint length, int duplicate);	添加一个指定长度的字符串 其余跟 add_next_index_string() 相同
add_next_index_zval (zval * array, zval * value);	添加一个 zval 结构。这在添加另外一个数组、对象或流等数据时会有用 相当于\$array[] = \$value;

上面所提到的API只是抽象HashTable的API函数而已。所以也可以直接使用HashTable的API函数进行操作。例如，如果想添加一个元素到数组，可以直接使用zend_hash_update()函数进行操作。

注意 关联数组是指使用字符串作为key的数组，而索引数组是指使用整数作为key的数组。

下面看添加一个元素到数组的例子：

```
zval*array, *element;
char*key="key_for_search";
char*value="value_for_element";
MAKE_STD_ZVAL ( array );
MAKE_STD_ZVAL ( element );
array_init ( array );
ZVAL_STRING ( element, value, 1 );
add_assoc_zval ( array, key, element );
```

效果类似于以下的PHP代码：

```
<? php
$array=array ();
$element="value_for_element";
$array["key_for_search"]= $element;
? >
```

6.对象类型变量

在讲为变量赋予对象类型的值之前，先要了解在PHP中对象跟数组的关系。看以下PHP代码：

```
<? php
class myclass {
public $property1;
public $property2;
public $property3;
}
$object=new myclass ();
$object->property1="property1";
$object->property2="property2";
$object->property3="property3";
var_dump ($object);
$object= (array)$object;
var_dump ($object);
? >
```

输出的结果如图7-20所示。



图 7-20 测试结果

从上面的代码可以看出，对象可以转换成数组（此过程不可逆，也就是说转换成数组之后不能再恢复到原来的对象，因为会丢失方法），所以对象和数组有很多相似的地方。实际上，对象的属性跟数组是可以相互转换的。下面看创建对象的API。

可以调用object_init()函数初始化一个对象：

```
zval*new_object;
MAKE_STD_ZVAL(new_object);
if (object_init(new_object) != SUCCESS)
{
RETURN_NULL();
}
```

Zend引擎有一套为对象添加属性的API，如表7-13所示。

函 数	描 述
<code>add_property_long(zval *object, char *key, long l);</code>	添加一个长整型类型的属性值
<code>add_property_int(zval *object, char *key);</code>	添加一个 <code>int</code> 类型的属性值
<code>add_property_bool(zval *object, char *key, int b);</code>	添加一个布尔类型的属性值
<code>add_property_double(zval *object, char *key, long d);</code>	添加一个双精度类型的属性值
<code>add_property_double(zval *object, char *key, double d);</code>	添加一个浮点类型的属性值
<code>add_property_string(zval *object, char *key, char *str, int duplicate);</code>	添加一个字符串类型的属性值
<code>add_property_string(zval *object, char *key, char *str, uint length, int duplicate);</code>	添加一个指定长度的字符串类型的属性值，速度比 <code>add_property_string()</code> 函数快，而且是二进制安全的
<code>add_property_zval(zval *object, char *key, zval *container);</code>	添加一个 <code>zval</code> 结构的属性值。这在添加另外一个数组、对象等数据时很有用

在上面所有的API中，第一个参数都是要添加属性的对象，第二个参数是属性的名字，最后一个参数是属性的值。例如添加一个名字为“name”，类型为字符串的属性：

```
zval*new_object;
MAKE_STD_ZVAL ( new_object );
if ( object_init ( new_object ) != SUCCESS )
{
RETURN_NULL ();
}
add_property_string ( new_object, "name", "James", 1);
```

7.资源类型变量

创建资源类型的变量比创建其他类型的变量烦琐一点。严格来说，资源不是数据类型，它是一个可以维护任何数据类型的抽象（就像C语言中的指针）。所有的资源都是保存在一个Zend内部的资源列表中，列表中的每份资源都有一个指向实际数据的指针。如果对PHP内核足够了解的话，可以直接访问这些资源，不过为了兼容性和安全性，还是建议使用Zend引擎提供的API访问它们。

如果某一个资源失去所有的引用，就会触发相应的析构函数，而这个析构函数是由资源自己提供的。为什么要提供析构函数呢？因为Zend引擎不能管理资源的实际数据，所以为了防止内存泄漏，就必须提供析构函数释放这些内存。

例如，数据库连接和文件描述符在PHP内部都是使用资源类型保存的。另外，可以保存任意的数据类型，如自定义的数据结构等。

使用 `zend_register_list_destructors_ex()` 函数可以用来注册一个资源变量的析构函数，该函数返回一个资源的句柄。这个资源句柄的作用是把资源与资源的析构函数相关联。以下是 `zend_`

`register_list_destructors_ex` 函数的定义：

```
ZEND_API int zend_register_list_destructors_ex ( rsrc_dtor_func_t ld,
rsrc_dtor_func_t pld, char*type_name, int module_number );
```

`zend_register_list_destructors_ex()` 函数的参数说明如表7-14所示。

参 数	描 述
ld	普通资源的析构函数
pld	持久化资源的析构函数
type_name	为资源指定一个名称。在 PHP 内部为某个资源类型起个名字是个好习惯（当然名字不能重复）。用户调用 <code>var_dump(\$resource)</code> 时就可取得该资源的名称
module_number	在模块的 <code>PHP_MINIT_FUNCTION</code> 函数中会自动定义，因此可将其忽略

从 `zend_register_list_destructors_ex()` 函数的定义可以看出，函数需要提供两个不同的资源析构函数：一个是普通资源的析构函数，另一个是持久化资源的析构函数。在注册资源析构函数时，这两个析构函数至少要提供一个，另外一个析构函数可以简单地设为 `NULL`。

资源的析构函数原型必须如下定义：

```
void resource_destruction_handler ( zend_rsrc_entry* rsrc TSRMLS_DC );
```

参数 `rsrc` 是一个指向 `zend_rsrc_list_entry` 结构体的指针：

```
typedef struct _zend_rsrc_list_entry {
void*ptr;
int type;
int refcount;
```

```
} zend_rsrc_list_entry;
```

其中ptr字段用于保存资源的真正数据的地址，一般在析构函数中释放ptr字段指向的内存。

例如定义一个链表数据结构如下：

```
typedef struct _ListNode {  
    struct _ListNode*next;  
    void*data;  
} ListNode;
```

然后资源变量保存的就是这个链表的头指针，析构函数可以这样编写：

```
void list_destroy_handler ( zend_rsrc_list_entry* rsrc TSRMLS_DC )  
{  
    ListNode*current, *next;  
    current= ( ListNode* ) rsrc->ptr;  
    while ( current ) {  
        next=current->next;  
        free ( current );  
        current=next;  
    }  
}
```

这样就可以释放整个链表的内存。

一般需要一个模块全局变量保存zend_register_list_destructors_ex()函数返回的资源句柄。如果用ext_skel工具生成扩展框架的话，这个全局变量已经自动生成，以“le_”作为前缀。例如，使用ext_skel工具创建一个myext扩展，就会自动生成一个类型为int的全局变量le_myext，用来保存zend_register_list_destructors_

ex () 函数返回的资源句柄。

一般在MINIT阶段注册析构函数，代码如下：

```
static int le_myext;
typedef struct _ListNode {
    struct _ListNode*next;
    void*data;
} ListNode;
void list_destroy_handler ( zend_rsrc_list_entry* rsrc TSRMLS_DC )
{
    ListNode*current, *next;
    current= ( ListNode* ) rsrc->ptr;
    while ( current ) {
        next=current->next;
        free ( current );
        current=next;
    }
}
PHP_MINIT_FUNCTION ( myext )
{
    le_myext=zend_register_list_destructors ( list_destroy_handler,
    NULL, "list_resource", module_number );
    return SUCCESS;
}
```

在注册完资源析构函数之后，还需要把真正的资源与这个资源句柄关联起来（这样当资源没有用时，就会自动调用析构函数）。可以使用zend_register_resource () 函数或者ZEND_REGISTER_RESOURCE () 宏实现。ZEND_REGISTER_RESOURCE () 宏比zend_register_resource () 函数兼容性跟好，所以最好使用前者。定义如下：

```
int ZEND_REGISTER_RESOURCE ( zval* rsrc_result, void* rsrc_pointer,
int rsrc_type );
```

zend_register_resource () 函数参数说明如表7-15所示。

参 数	描 述
zval* rsrc_result	用于存储 zend_register_resource () 函数返回的结果
void* rsrc_pointer	指向所保存的资源
int rsrc_type	是在注册函数析构函数时返回的资源句柄。注册成功之后会把资源与析构函数关联起来。对上 面的代码来说就是 le_stream

zend_register_resource () 函数的返回值表示这个资源的唯一ID，函数的定义如下：

```
int zend_register_resource ( zval* rsrc_result, void* rsrc_pointer, int rsrc_type )
{
    int rsrc_id;
    rsrc_id = zend_list_insert ( rsrc_pointer, rsrc_type );
    if ( rsrc_result ) {
        rsrc_result->value.lval = rsrc_id;
        rsrc_result->type = IS_RESOURCE;
    }
    return rsrc_id;
}
```

zend_list_insert () 函数把资源插入到资源列表中，并且返回资源所在的列表位置（资源ID）。之后就可以通过这个ID访问资源了。可以通过 RETURN_RESOURCE () 宏返回这个资源给用户，方法如下：

```
RETURN_RESOURCE ( rsrc_id );
```

当注册一个资源之后，Zend引擎就会一直监视这个资源的引用，当这个资源的引用为0时，Zend引擎就会自动调用资源析构函数（就是注册的那个资源析构函数）。这样就不用担心内存泄露了。

通过 RETURN_RESOURCE () 宏把资源ID返回给用户之后，用户就可以根据这个资源ID取得所需的资源。这个操作可以通过 ZEND_FETCH_RESOURCE () 宏实

现:

```
ZEND_FETCH_RESOURCE ( rsrc, rsrc_type, rsrc_id, default_rsrc_id,
resource_type_name, resource_type )
```

ZEND_FETCH_RESOURCE () 宏参数说明如表7-16所示。

参 数	描 述
rsrc	保存取得的资源
rsrc_type	表明想要的资源类型。比如 LibNuke 等
rsrc_id	用户通过 PHP 脚本传递给你的资源 ID
default_rsrc_id	假如没有取得资源时默认指定的资源标识符。通常为 -1
resource_type_name	所请求的资源类型名称。当不能找到资源时, 就用这个字符串去填充系统由于维护而输出的错误信息
resource_type	可以取回在注册资源析构函数时返回的资源句柄。本例是 le_mysql

大多数创建资源的函数都有相应用于释放资源的函数。例如mysql_connect () 有mysql_close ()、fopen () 有fclose () 等。或者直接unset () 一个资源变量, 也可以直接关闭和释放它, 如下所示:

```
<? php
$fp=fopen ( "file.txt", "r" );
unset ( $fp );
? >
```

在上面的代码中, 没有调用fclose () 函数关闭文件句柄, 但是结果却是这个文件句柄被关闭。为什么会这样呢? 还记得本节刚开始所说的资源析构函数吗? 奥秘就在这里, 当unset () 一个资源变量时, 会把这个资源变量的引用减1, 当变量的引用等于0时就会自动调用注册的资源析构函数把资源释放掉。这是利用了PHP内核的垃圾回收机制。

但是如果强制释放资源又怎么办呢? 这时候可以使用zend_list_delete () 函数完成。zend_list_delete () 原型如下:

```
ZEND_API int zend_list_delete ( int id TSRMLS_DC );
```

该函数的作用就是把资源变量的引用减1，然后判断引用是否小于等于0，如果是的话就把资源删除。zend_list_delete()代码如下：

```
#define zend_list_delete ( id ) _zend_list_delete ( id TSRMLS_CC )
ZEND_API int _zend_list_delete ( int id TSRMLS_DC )
{
    zend_rsrc_list_entry *le;
    if ( zend_hash_index_find ( &EG ( regular_list ),
        id, ( void** ) &le ) == SUCCESS ) {
        if ( --le->refcount <= 0 ) {
            return zend_hash_index_del ( &EG ( regular_list ), id );
        } else {
            return SUCCESS;
        }
    } else {
        return FAILURE;
    }
}
```

因为该函数只是把资源变量从资源列表中删除，资源占用的内存是不会被释放的，所以删除资源变量之后还要释放资源占用的内存。

7.5.7 错误和输出API

在PHP内核中，不能简单地使用printf()函数打印数据。因为PHP有自己一套管理输出流的函数，如果调用printf()函数打印数据，很有可能出现意想不到的结果。下面了解PHP的输出API。

1.php_printf()函数

php_printf()函数跟C语言标准库的printf()函数很相似，唯一不同的就是php_printf()函数指向的是Zend的输出流，其原型如下：

```
PHPAPI int php_printf ( const char*format, ……);
```

第一个参数指定输出字符串的格式，格式参数跟printf()函数一样，例如：

```
php_printf ("This is number: %d \n", 10);
```

2.zend_error()函数

zend_error()函数用于输出一个错误信息，原型如下：

```
ZEND_API void zend_error ( int type, const char*format, ……);
```

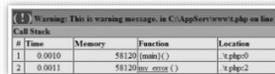
第一个参数指定错误信息的类型，第二个参数指定错误提醒消息。表7-17列举出了所有的错误信息类型。

错误信息类型	描述
E_ERROR	输出一个错误，然后立即中止脚本的执行
E_WARNING	输出一个一般性的警告，脚本会继续执行
E_NOTICE	输出一个通知，脚本会继续执行。注意，默认情况下 php.ini 会关闭显示这种错误
E_CORE_ERROR	输出一个 PHP 内核错误。通常情况下这种错误类型不应该被用户自己编写的模块引用
E_COMPILE_ERROR	输出一个编译器内部错误。通常情况下这种错误类型不应该被用户自己编写的模块引用
E_COMPILE_WARNING	输出一个编译器内部警告。通常情况下这种错误类型不应该被用户自己编写的模块引用

例如抛出一个警告类型的错误消息的代码如下：

```
zend_error ( E_WARNING, "This is warning message.");
```

输出如图7-21所示。



#	Time	Memory	Function	Location
1	0.0010	58120	(main())	x.php:0
2	0.0011	58120	zend_error()	x.php:2

图 7-21 PHP错误提示

3.向phpinfo () 中输出信息

一般完成一个扩展之后，希望使用者了解到这个扩展的一些信息，最直接的方法就是用phpinfo () 函数打印。怎样可以向phpinfo () 中添加信息呢？答案就是实现ZEND_MININFO () 函数。当在PHP脚本中调用phpinfo () 函数时，ZEND_MININFO () 函数会自动被调用。

如果指定ZEND_MININFO () 函数，phpinfo () 会自动打印一小节，小节的头部就是模块名。如果还想显示更多的信息就要自己去指定。

PHP内核提供了一些以表格格式显示信息的接口，可以使用这些接口格式化要显示的信息。一般情况下，需要完成三个步骤。

步骤1 调用php_info_print_table_start函数制定表格。

步骤2 调用php_info_print_table_header和php_info_print_table_row这两个函数打印表格具体的行列信息。

步骤3 调用php_info_print_table_end函数结束表格。

php_info_print_table_start () 和php_info_print_table_end () 这两个函数没有参数，而php_info_print_table_header () 和php_info_print_table_row () 这两个函数需要的参数会根据的需要而变化，原型如下：

```
void php_info_print_table_header ( int cols, .....);
void php_info_print_table_row ( int cols, .....);
```

第一个参数是希望表格拥有的列数，而后面的参数都是字符串类型的，参数个数根据想要的列数而变化。例如希望表格有两列的话，就需要提供两个字符串的参数，如下：

```
php_info_print_table_header ( 2, "this is first", "this is second" );
```

这些函数的作用如图7-22所示。



图 7-22 phpinfo图解

从图7-22中可以看出，php_info_print_table_header () 函数跟php_info_print_row () 函数不同的地方就是它们打印的颜色不一样。下面自己创建一个phpinfo () 的信息表格：

```
php_info_print_table_start ();
php_info_print_table_header ( 2, "First column", "Second column" );
php_info_print_table_row ( 2, "Entry in first row", "Another entry" );
php_info_print_table_row ( 2, "Just to fill", "another row here" );
php_info_print_table_end ();
```

输出的结果如图7-23所示。



图 7-23 测试结果

7.5.8 运行时信息函数

如果想了解当前正在执行的文件名或者正在进行的函数名，则可通过PHP内核提供的一系列查看这些信息的函数进行。

如果要查看当前正在执行的函数名，可以使用`get_active_function_name()`函数获取。该函数需要传入`TSRMLS_C`宏作为参数访问执行器（Executor）全局变量，返回值为函数名的指针。

如果要查看当前执行的文件名，可以使用`zend_get_executed_filename()`获取。该函数也需要传入`TSRMLS_C`宏作为参数，返回值为文件名的指针。

如果要查看当前执行到哪一行代码了，可以使用`zend_get_executed_lineno()`函数获取。该函数同样需要传入`TSRMLS_C`宏作为参数，返回值为当前进行代码的行数。

下面是这些函数使用的例子：

```
php_printf ( "The name of current function is: %s \n",
get_active_function_name ( TSRMLS_C ));
php_printf ( "The file currently executed is: %s \n",
zend_get_executed_filename ( TSRMLS_C ));
php_printf ( "The current line being executed is: %i \n",
zend_get_executed_lineno ( TSRMLS_C ));
```

输出结果如图7-24所示。



```
The name of current function is: conf_mymem_compiled
The file currently executed is: C:\php\phpwin32.php
The current line being executed is: 2
```

图 7-24 测试结果

7.5.9 调用用户自定义函数

如果你喜欢逆向思维的话，可能会问，能不能在扩展里面调用用户在PHP脚本中定义的函数呢？答案是肯定的，PHP允许在扩展里面调用PHP脚本中定义的函数，实现一些回调机制之类的功能。

要在扩展中调用一个PHP脚本定义的函数，可以使用`call_user_function_ex()`函数实现。函数的原型如下：

```
int call_user_function_ex ( HashTable*function_table,
zval**object_pp, zval*function_name,
zval**retval_ptr_ptr, zend_uint param_count,
zval**params[], int no_separation,
HashTable*symbol_table TSRMLS_DC );
```

表7-18展示了`call_user_function_ex()`函数的各个参数的作用。

参 数	描 述
function_table	指定要访问的函数表指针
object_pp	如果你调用的是一个方法，需要设置此参数指定要调用方法的对象，否则设置为 NULL
function_name	指定要调用的函数名（或方法名）
retval_ptr_ptr	保存返回值的指针
param_count	指定参数的个数
params	指定参数列表
no_separation	指定是否要禁止 eval 分离操作
symbol_table	符号表，一般设置为 NULL

注意 `function_table`和`object_pp`这两个参数只需要指定其中一个就行了。如果要调用的是一个函数，就需要指定`function_table`，把`object_pp`设置为NULL；如果想调用一个方法的话，就必须指定`object_pp`，Zend引擎会自动把函数表设置为当前对象的函数表。

另外需要特别注意`no_separation`参数，该参数指定是否要禁止进行zval分离操作，一般情况下都是设置为0。如果设置为1的话，函数会更省空间，但是当任何一个参数需要进行zval分离时就会导致操作失败。

下面举个例子说明`call_user_function_ex()`函数的使用方法：

```
PHP_FUNCTION ( my_call_user_function )
{
    zval**function_name;
    zval*retval;
    if ( ZEND_NUM_ARGS () != 1 ||
        ( zend_get_parameters_ex ( 1, &function_name ) != SUCCESS ) )
    {
        WRONG_PARAM_COUNT;
    }
    if ( ( *function_name )->type != IS_STRING )
    {
        zend_error ( E_ERROR, "Function requires string argument" );
    }
    if ( call_user_function_ex ( CG ( function_table ),
        NULL, *function_name, &retval,
        0, NULL, 0, NULL TSRMLS_CC ) != SUCCESS )
    {
        zend_error ( E_ERROR, "Function call failed" );
    }
    *return_value=*retval;
    zval_copy_ctor ( return_value );
    zval_ptr_dtor ( &retval );
}
```

上面的例子中，创建一个my_call_user_function扩展函数，在函数中调用call_user_function_

ex () 函数调用用户在PHP脚本中定义的函数。要注意的是CG (function_table) 参数，它是全局函数表。然后可以使用以下的PHP脚本调用此扩展函数：

```
<? php
function myfunc ()
{
    echo"call user function success! \n";
    return"this is myfunc () ";
}
```

```
$retval=my_call_user_function ("myfunc");  
echo"Return value: $retval";  
? >
```

输出结果如图7-25所示。



图 7-25 测试结果

7.5.10 PHP配置项

配置功能对于一个程序来说是非常重要的。例如，有时候想把数据保存到D盘，而有时候却想把数据保存到F盘，就需要配置功能。修改程序代码可以实现，不过这样做太麻烦（如果是编译型语言需要重新编译）。最好的方法就是创建一个配置文件，让程序读取配置文件的内容。以后想修改某些配置时，只需修改配置文件就可以。

如果想为你的扩展创建一个.ini文件的配置节，可以使用PHP_INI_BEGIN()宏标识这个节的开始，并用PHP_INI_END()宏标识配置节的结束。可以在两者之间使用PHP_INI_ENTRY()宏创建具体的配置项，具体如下：

```
PHP_INI_BEGIN()  
PHP_INI_ENTRY("myext.setting", "Hello World", PHP_INI_ALL, NULL)  
PHP_INI_END()
```

PHP_INI_ENTRY()宏接收4个参数：配置项名称、初始值、改变这个值所需的权限以及改变这个值时的回调函数句柄。配置项名称和初始值必须是一个字符串，即使它们是一个整数也要转换为字符串。

改变配置值所需的权限可以分为三种：

PHP_INI_SYSTEM，只允许在php.ini中修改这些值。

PHP_INI_USER，允许用户在运行像.htaccess这样的文件时重写其值。

PHP_INI_ALL，允许随意修改这些值。

第四个参数指定当初始值被修改后的回调函数句柄。一旦某个配置项的初始值被修改，相应的回调函数将会被调用。可以使用PHP_INI_MH()宏定义函数：

```
PHP_INI_MH(OnChangeSetting);  
PHP_INI_MH(OnChangeSetting)
```

```

{
php_printf ( "Our ini entry has been changed to%s \n", new_value );
return ( SUCCESS );
}

```

改变之后的新值会通过一个new_value字符串指针传递个函数。在上面的例子中，直接打印这个值。

要让扩展能够读到配置项的值，就必须把整个初始化配置项引入PHP内核中，这项工作可以在模块的起始和结束函数中使用REGISTER_INI_ENTRIES()宏和UNREGISTER_INI_ENTRIES()

宏完成：

```

ZEND_MINIT_FUNCTION ( myext )
{
REGISTER_INI_ENTRIES ();
}
ZEND_MSHUTDOWN_FUNCTION ( myext )
{
UNREGISTER_INI_ENTRIES ();
}

```

创建好配置项之后，可以使用一系列的宏访问这些配置项，这些宏如表7-19所示。

宏	描述
INI_INT(name)	将配置项 name 的当前值以长整型数返回
INI_FLT(name)	将配置项 name 的当前值以双精度浮点型数返回
INI_STR(name)	将配置项 name 的当前值以字符串返回。注意：字符串不是复制过的字符串，而是直接指向内部数据。如果你需要进行进一步的访问的话，那就需要再进行复制一下
INI_BOOL(name)	将配置项 name 的当前值以布尔值返回（返回值被定义为 zend_bool，也就是说是一个 unsigned char）
INI_ORIG_INT(name)	将配置项 name 的初始值以长整型数返回
INI_ORIG_FLT(name)	将配置项 name 的初始值以双精度浮点型数返回
INI_ORIG_STR(name)	将配置项 name 的初始值以字符串返回。注意：字符串不是复制过的字符串，而是直接指向内部数据。如果你需要进行进一步的访问的话，那就需要再进行复制
INI_ORIG_BOOL(name)	将配置项 name 的初始值以布尔值返回（返回值被定义为 zend_bool，也就是说是一个 unsigned char）

下面举例说明配置的使用。

1) 包含头文件:

```
#include"php.h"  
#include"php_ini.h"  
#include"ext/standard/info.h"#include"php_myext.h"
```

2) 定义一些必须的变量和结构:

```
static int le_myext;  
zend_function_entry myext_functions[] = {  
    PHP_FE ( myext, NULL )  
    { NULL, NULL, NULL }  
};  
zend_module_entry myext_module_entry = {  
    #if ZEND_MODULE_API_NO >= 20010901  
    STANDARD_MODULE_HEADER,  
    #endif  
    "myext",  
    myext_functions,  
    PHP_MINIT ( myext ),  
    PHP_MSHUTDOWN ( myext ),  
    PHP_RINIT ( myext ),  
    PHP_RSHUTDOWN ( myext ),  
    PHP_MINFO ( myext ),  
    #if ZEND_MODULE_API_NO >= 20010901  
    "0.1", #endif  
    STANDARD_MODULE_PROPERTIES  
};  
#ifdef COMPILE_DL_MYEXT  
ZEND_GET_MODULE ( myext )  
#endif
```

3) 定义修改配置默认值时的回调函数:

```
PHP_INI_MH ( OnChangeSetting )
{
php_printf ( "Our ini entry has been changed to%s \n", new_value );
return ( SUCCESS );
}
```

4) 扩展创建配置项:

```
PHP_INI_BEGIN ( )
PHP_INI_ENTRY ( "myext.setting", "Hello World",
PHP_INI_ALL, OnChangeSetting )
PHP_INI_END ( )
```

5) 定义模块初始化函数, 并把配置项引入到PHP内核中:

```
PHP_MINIT_FUNCTION ( myext )
{
REGISTER_INI_ENTRIES ( );
return SUCCESS;
}
PHP_MSHUTDOWN_FUNCTION ( myext )
{
UNREGISTER_INI_ENTRIES ( );
return SUCCESS;
}
```

6) 定义请求初始化函数:

```
PHP_RINIT_FUNCTION ( myext )
```

```
{
return SUCCESS;
}
PHP_RSHUTDOWN_FUNCTION ( myext )
{
return SUCCESS;
}
```

7) 定义在phpinfo () 中显示的信息:

```
PHP_MININFO_FUNCTION ( myext )
{
php_info_print_table_start ();
php_info_print_table_header ( 2, "myext support", "enabled" );
php_info_print_table_end ();
}
```

8) 取得并打印配置项的值:

```
PHP_FUNCTION ( myext )
{
php_printf ( "myext.setting: %s", INI_STR ( "myext.setting" ));
RETURN_TRUE;
}
```

上述程序的运行结果如图7-26所示。

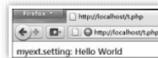


图 7-26 测试结果

7.5.11 创建常量的宏

PHP支持常量的定义，在编写扩展时可能希望定义一些预定义的常量给用户使用，这时候就要使用创建常量的宏。

PHP的常量是不需要使用“”作为前缀的，而且是全局有效的。比如TRUE和FALSE这两个常量。可以使用以下的宏创建常量，如表7-20所示。

宏	描述
REGISTER_LONG_CONSTANT (name, value, flags)	新建一个长整型常量
REGISTER_DOUBLE_CONSTANT (name, value, flags)	新建一个双精度型常量
REGISTER_STRING_CONSTANT (name, value, flags)	新建一个字符串型常量。给定的字符串的空间必须在 Zend 内部内存
REGISTER_STRINGL_CONSTANT (name, value, length, flags)	新建一个指定长度的字符串型常量。同样，给定字符串的空间必须在 Zend 内部内存

上面的所有宏在创建常量时必须指定一个名称和值。还可以通过第三个参数（flags）指定常量的特性。flags参数可以设置以下标识：

CONST_CS：设置此标识的常量是大小写敏感的。

CONST_PERSISTENT：设置此标识的常量是持久化的。也就是说常量在本次请求完毕之后也不会被释放，其他请求还可以继续使用。

使用二进制的“或”操作可以指定一个变量具有两种标识的特性，如下：

```
REGISTER_LONG_CONSTANT ("MY_NEW_CONSTANT", 10, CONST_CS | CONST_PERSISTENT);
```

上面的代码创建了一个长整型常量MY_NEW_CONSTANT，并且是大小写敏感和持久化的。

7.6 编写一个完整的扩展

至此，我们已经把基本的Zend API介绍完毕了。掌握这些Zend API之后，就可以开始编写扩展了。下面编写一个完整的扩展。

PHP只支持HashTable数据结构，若想使用其他的数据结构则会显得很无助。所以可以为PHP编写一个链表结构的扩展弥补它的不足。

7.6.1 链表结构的实现

因为要实现一个链表结构的扩展，所以要先了解链表数据结构，如果你已对链表结构有比较深入的认识，可以跳过本节。

链表是最基础的数据结构之一。它是一种线性数据结构，但并不会按照线性顺序存储数据，而是通过一个指向下一个节点的指针把所有的节点连接起来。结构如图7-27所示。



图 7-27 链表结构

双向链表比单向链表具有更高的灵活性，下面以双向链表作为例子。双向链表中，每个结点都有两个指针域，一个指针指向其后继结点，另一个指针指向其前驱结点，结构如图7-28所示。

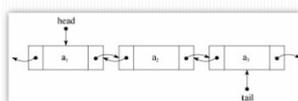


图 7-28 双向链表结构图

使用下面的结构体表示一个节点：

```
typedef struct list_node {
    zval*value;
    struct list_node*prev;
    struct list_node*next;
} list_node;
```

在上面的list_node结构体中，value字段用来存储数据，而prev字段指向前驱节点，next字段指向后继节点。

使用下面的结构体保存一个链表的信息：

```
typedef struct list_head {
    int size;
    list_node*head;
    list_node*tail;
} list_head;
```

size字段是记录链表的节点数，head字段用于保存链表头结点，而tail字段用于保存链表尾节点。

下面看链表的操作。

1.创建一个链表

创建链表操作比较简单，通过malloc()系统调用申请一个链表结构并初始化它所有字段即可。用函数返回这个链表指针，如下所示：

```
list_head*list_create()
{
    list_head*head;
    head=(list_head*) malloc(sizeof(list_head));
    if(head){
        head->size=0;
        head->head=NULL;
        head->tail=NULL;
    }
    return head;
}
```

2.添加一个节点到链表的表头

使用list_add_head()函数把节点添加到链表的表头。通过malloc()系统调用来申请一个新节点空间,然后把新节点的后继节点指针指向链表表头节点,把链表表头节点的前驱指针指向新节点。接着把链表表头重新设置为新节点。具体实现如下所示:

```
int list_add_head ( list_head*head, zval*value )
{
    list_node*node;
    node= ( list_node* ) malloc ( sizeof ( *node ) );
    if ( ! node ) {
        return 0;
    }
    node->value=value;
    node->prev=NULL;
    node->next=head->head;
    if ( head->head )
    {
        head->head->prev=node;
    }
    head->head=node;
    if ( ! head->tail )
    {
        head->tail=head->head;
    }
    head->size++;
    return 1;
}
```

把新节点添加到链表的表头,如图7-29所示。

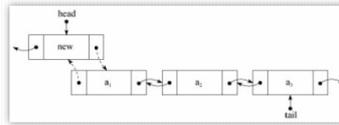


图 7-29 添加新节点到链表表头

注意 当链表只有一个节点时，链表表尾与链表表头指向同一个节点。

3.添加一个节点到链表的表尾

添加新节点到链表表尾操作与添加到表头的操作差不多，主要是把新节点的前驱节点指针指向链表表尾，然后把链表表尾的后继指针指向新节点，最后重新设置链表表尾为新节点即可。具体实现如下所示：

```
int list_add_tail ( list_head*head, zval*value )
{
    list_node*node;
    node= ( list_node* ) malloc ( sizeof ( *node ) );
    if ( ! node ) {
        return 0;
    }
    node->value=value;
    node->prev=head->tail;
    node->next=NULL;
    if ( head->tail )
    {
        head->tail->next=node;
    }
    head->tail=node;
    if ( ! head->head )
    {
        head->head=head->tail;
    }
    head->size++;
    return 1;
}
```

4.删除指定位置的节点

`list_delete_index()` 函数的作用是找到指定位置的那个节点，然后把此节点删除。具体实现如下所示：

```
int list_delete_index ( list_head*head, int index )
{
    list_node*curr;
    if ( index < 0 )
    {
        index = ( -index ) - 1;
        curr = head -> tail;
        while ( index > 0 )
        {
            curr = curr -> prev;
            index --;
        }
    } else {
        curr = head -> head;
        while ( index > 0 )
        {
            curr = curr -> next;
            index --;
        }
    }
    if ( ! curr || index > 0 ) return 0;
    if ( curr -> prev ) {
        curr -> prev -> next = curr -> next;
    } else {
        head -> head = curr -> next;
    }
    if ( curr -> next ) {
        curr -> next -> prev = curr -> prev;
    } else {
        head -> tail = curr -> prev;
    }
    return 1;
}
```

```
}

```

`list_delete_index()` 函数支持正负数两种索引：当索引是正数时，表示从链表的表头向表尾方向开始计算，删除第 `index` 个节点；而当索引是负数时，表示从链表表尾向表头方向开始计算，删除第 $(-index) - 1$ 个元素。

值得注意的是，如果删除的是头节点，需要把头节点指针指向头节点的下一个节点。如果删除的是尾节点，则需要把尾节点指针指向尾节点的上一个节点。删除过程如图 7-30 所示。

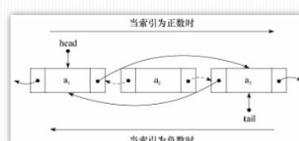


图 7-30 删除节点过程

5. 获取指定位置节点

`list_fetch()` 函数取得指定位置的节点，函数成功返回 1，并且把数据保存到 `retval` 变量中，如果失败则返回 0。`list_fetch()` 函数也支持正负数索引。具体实现如下所示：

```
int list_fetch ( list_head*head, int index, zval**retval )
{
    list_node*node;
    if ( index > 0 ) {
        node = head->head;
        while ( node && index > 0 ) {
            node = node->next;
            index--;
        }
    } else {
        index = (-index) - 1;
        node = head->tail;
        while ( node && index > 0 ) {
            node = node->prev;
            index--;
        }
    }
    *retval = node->data;
    return 1;
}

```

```
}  
}  
if (! node || index>0) return 0;  
*retval=node->value;  
return 1;  
}
```

6.取得链表的节点个数

虽然取得链表的节点个数非常简单，但这却非常重要。因为必须知道链表的节点个数才能保证获取指定节点时不会发生错误。取得链表的节点个数的具体实现如下所示：

```
int list_length ( list_head*head )  
{  
if ( head ) {  
return head->size;  
} else {  
return 0;  
}}  
}
```

7.释放链表

释放链表的操作很简单，主要过程是遍历链表，然后释放所有的节点，最后把链表结构释放。具体实现如下所示：

```
void list_destroy ( list_head*head )  
{  
list_node*curr, *next;  
curr=head->head;  
while ( curr )  
{  
next=curr->next;  
free ( curr );  
}}
```

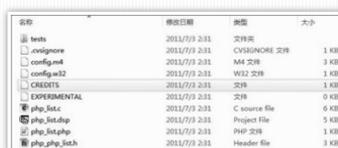
```
curr=next;  
}  
free ( head );  
}
```

7.6.2 创建PHP扩展框架

本节创建的扩展叫“php_list”，表示这是一个链表结构的扩展。下面以Windows操作系统作为编写扩展的平台。首先使用ext_skel工具创建扩展框架，命令如下：

```
php ext_skel_win32.php—extname=php_list
```

完成之后，会得到扩展的所有文件和配置，如图7-31所示。



名称	修改日期	类型	大小
tests	2011/7/3 2:31	文件夹	
.cvsignore	2011/7/3 2:31	CVSIGNORE 文件	1 KB
config.m4	2011/7/3 2:31	M4 文件	3 KB
config.w32	2011/7/3 2:31	W32 文件	1 KB
CREDITS	2011/7/3 2:31	文件	1 KB
EXPERIMENTAL	2011/7/3 2:31	文件	0 KB
php_list.c	2011/7/3 2:31	C 源文件	6 KB
php_list.dep	2011/7/3 2:31	Project File	5 KB
php_list.php	2011/7/3 2:31	PHP 文件	1 KB
php_list.h	2011/7/3 2:31	Header file	3 KB

图 7-31 php_list扩展框架的结构

7.6.3 编写代码

在编写代码之前，应该先构思好扩展应该有哪些功能。本例中的扩展提供以下功能：

创建链表结构资源：resource list_create ();

添加数据到链表表头：bool list_add_head (list, value);

从链表中取得头节点的数据：mixed list_fetch_head (list);

添加数据到链表表尾：bool list_add_tail (list, value);

从链表中取得尾节点的数据：mixed list_fetch_tail (list);

从链表中取得指定位置的数据：mixed list_fetch_index (list, index);

取得链表节点个数：int list_element_nums (list);

从链表中删除指定的节点：bool list_delete_index (list, index);

释放链表：void list_destroy (list);

注意 上面所有函数的list参数都是指链表结构资源。

现在开始编写代码，下面是代码的第1部分：

```
1. #ifdef HAVE_CONFIG_H
2. #include "config.h"
3. #endif
4. #include "php.h"
5. #include "php_ini.h"
6. #include "ext/standard/info.h"
7. #include "php_php_list.h"
8. #include "list.h"
9. static int le_php_list;
10. static int freed=0;
```

上述代码中1~8行包含一些主要的头文件。list.h是编写的链表结构的头文件，必须把这个头文件包含进来才能用链表结构的接口。

上述代码第9~10行中le_php_list是资源句柄，用于保存创建的链表结构资源。而freed用于标识链表结构是否已经释放，因为多次释放同一块内存会导致致命的错误，所以使用这个标识防止多次释放链表结构资源。

下面是代码的第2部分：

```
11.void list_destroy_handler ( zend_rsrc_list_entry* rsrc TSRMLS_DC )
12. {
13.if (! freed) {
14.list_head* list;
15.list = ( list_head* ) rsrc->ptr;
16.list_destroy ( list );
17.freed=1;
18. }
19. }
```

上述代码第11~19行中list_destroy_handler ()是链表资源的析构函数。在PHP生命周期结束之前，会自动调用list_destroy_handler ()函数。list_destroy_handler ()函数的主要工作是调用list_destroy ()函数释放链表结构占用的内存。

可以看到，只有当freed不为真时才会释放链表。而释放链表之后，应该把freed设置为真，表示链表已经释放过，这样就可以防止多次释放同一块内存。

下面是代码的第3部分：

```
20.zend_function_entry php_list_functions[] = {
21.PHP_FE ( list_create, NULL )
22.PHP_FE ( list_add_head, NULL )
23.PHP_FE ( list_fetch_head, NULL )
24.PHP_FE ( list_add_tail, NULL )
```

```
25.PHP_FE ( list_fetch_tail, NULL )
26.PHP_FE ( list_fetch_index, NULL )
27.PHP_FE ( list_delete_index, NULL )
28.PHP_FE ( list_destroy, NULL )
29.PHP_FE ( list_element_nums, NULL )
30. { NULL, NULL, NULL }
31.};
```

上述代码中第20~31行声明了Zend函数块，它的主要作用是让Zend引擎知道扩展中有哪些是可以让PHP脚本调用的函数。随后还要把它引入到Zend引擎中。

下面是代码的第4部分：

```
32.zend_module_entry php_list_module_entry= {
33.#if ZEND_MODULE_API_NO >= 20010901
34.STANDARD_MODULE_HEADER,
35.#endif
36."php_list",
37.php_list_functions,
38.PHP_MINIT ( php_list ),
39.PHP_MSHUTDOWN ( php_list ),
40.PHP_RINIT ( php_list ),
41.PHP_RSHUTDOWN ( php_list ),
42.PHP_MINFO ( php_list ),
43.#if ZEND_MODULE_API_NO >= 20010901
44."0.1",
45.#endif
46.STANDARD_MODULE_PROPERTIES 47. };
48.#ifdef COMPILE_DL_PHP_LIST
49.ZEND_GET_MODULE ( php_list )
50.#endif
```

上述代码中第32~47行声明的是Zend模块入口块 (Zend Module Block)，它包含模块的信息。在第37行设置Zend函数块，这样就把Zend函数块引入到Zend引擎中。

上述代码中第48~50行实现了`get_module()`函数。当使用动态加载的方式加载模块的话，就会自动调用这个函数。函数的作用就是返回Zend模块入口块给Zend引擎，过程如图7-32所示（在本例中Zend模块入口块就是`php_list_module_entry`）。

下面是代码的第5部分：

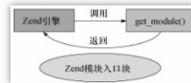


图 7-32 `get_module()` 函数调用过程

```
51.PHP_MINIT_FUNCTION (php_list)
```

```
52. {
53.le_php_list=zend_register_list_destructors_ex (
list_destroy_handler, NULL, "list_resource", module_number);
54.return SUCCESS;
55. }
56.PHP_MSHUTDOWN_FUNCTION (php_list)
57. {
58.return SUCCESS;
59. }
60.PHP_RINIT_FUNCTION (php_list)
61. {
62.return SUCCESS;
63. }
64.PHP_RSHUTDOWN_FUNCTION (php_list)
65. {
66.return SUCCESS;
67. }
```

上述代码中第51~67行主要实现了PHP生命周期各个阶段的初始化函数和析构函数。在7.2.5节中已经详细的介绍过这些函数的作用了。

在扩展被载入阶段的处理函数中（第53行），注册链表资源的析构函数。当PHP生命周期结束时，就会自动调用这个析构函数释放链表资源。其他阶段的处理函数，直接返回

SUCCESS。

下面是代码的第6部分：

```
68.PHP_MINFO_FUNCTION ( php_list )
69. {
70.php_info_print_table_start ();
71.php_info_print_table_header ( 2, "php_list support", "enabled" );
72.php_info_print_table_end ();
73. }
```

上述代码中第68~73行定义的是在phpinfo中显示的信息。

以上所述内容一般由ext_skel工具自动生成，只需修改部分代码就可以了。下面开始讲解导出函数的实现部分。

下面是代码的第7部分：

```
74.PHP_FUNCTION ( list_create )
75. {
76.list_head*list;
77.list=list_create ();
78.if (! list) {
79.RETURN_NULL ();
80. } else {
81.ZEND_REGISTER_RESOURCE ( return_value, list, le_php_list );
82. }
83. }
```

上述代码第74~83行中，list_create () 导出函数的功能是创建一个新的链表结构，然后使用ZEND_REGISTER_RESOURCE () 宏把它注册到Zend引擎的资源列表中（见第81行）。

下面是代码的第8部分:

```
84.PHP_FUNCTION ( list_add_head )
85. {
86.zval*value;
87.zval*lrc;
88.list_head*list;
89.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "rz",
&lrc, &value ) ==FAILURE ) {
90.RETURN_FALSE;
91. }
92.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
"List Resource", le_php_list );
93.list_add_head ( list, value );
94.RETURN_TRUE;
95. }
```

上述代码第84~88行中, `list_add_head()` 导出函数的功能是把一条记录插入到链表的表头, 对应的PHP原型为: `bool list_add_head (list, value)`。list为链表资源, value为要插入的数据。

上述代码第89行中使用`zend_parse_parameters()`函数接收从PHP脚本传递进来的参数。这里要接收两个参数: 一个是链表资源, 一个是要插入的数据。

上述代码第92~95行中使用`ZEND_FETCH_RESOURCE()`宏取得链表的实体指针, 然后通过`list_add_head()`链表库接口把记录插入到链表中。

下面是代码的第9部分:

```
96.PHP_FUNCTION ( list_fetch_head )
97. {
98.zval*lrc, *retval;
99.list_head*list;
100.int res;
```

```
101.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "r",
102.&lrc ) ==FAILURE ) {
103.RETURN_FALSE;
104.}
105.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
106."List Resource", le_php_list );
107.res=list_fetch ( list, 0, &retval );
108.if (! res) {
109.RETURN_NULL (); 108. } else {
110.RETURN_ZVAL ( retval, 1, 0 );
111. }
```

上述代码第96~104行中，`list_fetch_head()`导出函数的功能是取得链表头结点的数据，对应的PHP原型为`mixed list_fetch_head(list)`，在调用此函数时需要以链表资源作为参数。先使用`ZEND_FETCH_RESOURCE()`宏取得链表的实际指针。

上述代码中第105~111行中，通过`list_fetch()`函数取得链表的头结点数据。如果成功获取，即返回此数据，否则返回NULL。

下面是代码的第10部分：

```
112.PHP_FUNCTION ( list_add_tail )
113. {
114.zval*value;
115.zval*lrc;
116.list_head*list;
117.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "rz",
118.&lrc, &value ) ==FAILURE ) {
119.RETURN_FALSE;
120.}
121.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
122."List Resource", le_php_list );
123.list_add_tail ( list, value );
124.RETURN_TRUE;
```

```
123. }
```

上述代码第112~120行中，`list_add_tail()`导出函数的功能是把一条记录插入到链表的表尾。对应的PHP原型为`bool list_add_tail(list, value)`，`list`为链表资源，`value`为要插入的数据。先使用`zend_parse_parameters()`函数接收从PHP脚本传递进来的参数。这里要接收两个参数，一个是链表资源，一个是要插入的数据。

上述代码第121~123行中，使用`list_add_tail()`函数把数据添加到链表的表尾，并返回TRUE。下面是代码的第11部分：

```
124.PHP_FUNCTION (list_fetch_tail)
125. {
126.zval*lrc, *retval;
127.list_head*list;
128.int res;
129.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "r",
130.&lrc ) ==FAILURE ) {
131.RETURN_FALSE;
132.}
133.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
134."List Resource", le_php_list );
135.res=list_fetch ( list, list_length ( list ) -1, &retval );
136.if (! res) {
137.RETURN_NULL ();
138.} else {
139.RETURN_ZVAL ( retval, 1, 0 );
140.}
141.}
```

上述代码第124~132行中，`list_fetch_tail()`导出函数的功能是取得链表尾节点的数据。对应的PHP原型为`mixed list_fetch_tail(list)`，`list`为链表资源。使用`zend_parse_parameters()`函数接收从PHP脚本传递进来的链表资源。

上述代码第133~139行中，使用list_fetch()函数取得链表的尾节点数据。如果成功获取，即返回此数据，否则返回NULL。

下面是代码的第12部分：

```
140.PHP_FUNCTION ( list_fetch_index )
141. {
142.zval*lrc, *retval;
143.list_head*list;
144.long index;
145.int res;
146.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "rl",
&lrc, &index ) ==FAILURE ) {
147.RETURN_FALSE;
148. }
149.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
"List Resource", le_php_list );
150.res=list_fetch ( list, index, &retval );
151.if (! res) {
152.RETURN_NULL ();
153. } else {
154.RETURN_ZVAL ( retval, 1, 0 );
155. }
156. }
```

上述代码第140~149行中，list_fetch_index()导出函数的功能是取得链表指定位置的数据。对应的PHP原型为mixed list_fetch_index (list, index)，list为链表资源，index为要取得数据的位置。先使用zend_parse_parameters()函数接收从PHP脚本传递进来的链表资源。

上述代码第150~156行中，使用list_fetch()函数取得指定位置的数据，并把数据返回给用户。下面是代码的第13部分：

```
157.PHP_FUNCTION ( list_element_nums )
```

```
158. {
159.zval*lrc;
160.list_head*list;
161.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "r",
162.&lrc ) ==FAILURE ) {
163.}
164.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
165."List Resource", le_php_list );
166.}
```

上述代码第157~166行中，`list_element_nums()`导出函数的功能是取得链表的元素个数。对应的PHP原型为`int list_element_nums (list)`。该函数主要使用`list_length()`函数获取链表的元素个数。

下面是代码的第14部分：

```
167.PHP_FUNCTION ( list_delete_index )
168. {
169.zval*lrc;
170.list_head*list;
171.long index;
172.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "rl",
173.&lrc, &index ) ==FAILURE ) {
174.}
175.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
176."List Resource", le_php_list );
177.if ( list_delete ( list, index ) ) {
178.} else {
179.}
180.}
181.}
```

上述代码第167~181行中，`list_delete_index()` 导出函数的功能是删除链表指定位置的元素。对应的PHP原型为`bool list_delete_index(list, index)`，`index`参数是要删除元素的位置。使用`list_delete()` 函数删除指定位置的元素。

下面是代码的第15部分：

```
182.PHP_FUNCTION ( list_destroy )
183. {
184.zval*lrc;
185.list_head*list;
186.if ( zend_parse_parameters ( ZEND_NUM_ARGS () TSRMLS_CC, "r",
187.&lrc ) ==FAILURE ) {
187.RETURN_FALSE;
188. }
189.ZEND_FETCH_RESOURCE ( list, list_head*, &lrc, -1,
190."List Resource", le_php_list );
191.if (! freed) {
191.list_destroy ( list );
192.freed=1;
193. }
194. }
```

上述代码第182~194行中，`list_destroy()` 导出函数的功能是释放链表占用的内存。对应的PHP原型为`bool list_destroy(list)`，主要使用`list_destroy()` 函数释放链表。我们注意到，当`freed`为0时才会释放链表，并把`freed`设置为1，这样做的目的是防止多次释放链表而导致出错。

7.6.4 编译安装扩展

扩展写好了，现在要做的工作就是编译扩展和安装扩展。在编译扩展之前，应该把链表库文件引入到工程中，如图7-33所示。

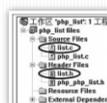


图 7-33 引入链表库文件到工程中

把组建模式设置为“Win32 Release_TS”，如图7-34所示。

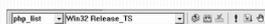


图 7-34 设置组建模式

现在可以编译扩展了，只要按编译按钮即可。之后就可以在Release_TS目录下找到编译的动态扩展库，如图7-35所示。



图 7-35 动态扩展库文件

把编译好的动态扩展库复制到PHP安装路径的ext目录下。然后修改“php.ini”配置文件，把编写的扩展添加进去，配置如下：

```
extension=php_php_list.dll
```

完成后重启Web服务器。

7.6.5 测试扩展

扩展安装完成，现在可以开始编写测试程序验证编写的扩展是否能够正常工作。测试代码如下：

```
<? php
$list=list_create ();
for ($i=0; $i<10; $i++) {
list_add_head ($list, "elements[$i]");
}
$list_nums=list_element_nums ($list);
echo"numbers of list: ",$list_nums;
for ($i=$list_nums-1; $i>=0; $i——) {
echo list_fetch_index ($list,$i);
}? >
```

在上面的代码中，首先使用list_create()函数创建一个链表资源，使用list_add_head()函数向链表中添加10个元素；然后通过list_element_nums()函数取得链表的元素个数；最后使用list_fetch_index()函数遍历链表的所有元素。测试结果图7-36所示。



图 7-36 链表扩展测试结果

7.7 本章小结

本章主要介绍了编写PHP扩展的详细过程和大部分的Zend API等。Zend引擎提供的扩展API相当丰富，所以在本章中不可能把所有的内容都完整地介绍一遍，如果想更深入地了解Zend引擎或者更高级的Zend API，可以浏览PHP的源代码。

通过对本章的学习，你应不仅学会开发PHP扩展的技能，而且对PHP内部实现原理有了更进一步的了解。现在，你应已经学到足够的知识创建自己的扩展了。如果以后需要编写效率非常高的程序，可以考虑把程序编写成PHP扩展的形式。特别要注意的是，PHP扩展更接近PHP底层，如果编写的扩展有错误，很有可能会导致PHP崩溃。所以，在编写PHP扩展时要非常细心，可以的话尽量使用PHP提供的API代替操作系统的API。例如，尽量使用PHP的内存管理函数代替操作系统的内存管理函数。

第8章 缓存详解

随着网络的发展，数据越来越多，从而导致运算的压力越来越大，为了解决这一问题，就需要合理分级计算资源，充分利用已有资源。缓存的工作实际上就是资源的合理分配。

8.1 认识缓存

缓存（Cache）原意是指可以进行高速数据交换的存储器。当CPU处理数据时，先到Cache中寻找，如果数据因之前的操作已经读取而被暂存其中，就不需要再从随机存取存储器（Random Access Memory, RAM）中读取数据了。

现在缓存的概念已被扩充，不仅在CPU和主内存之间有Cache，而且在内存和硬盘之间也有Cache（磁盘高速缓存）——凡是位于速度相差较大的两种介质之间，用于协调两者数据传输速度差异的结构，均可称为Cache。按照这个概念，只要两种介质之间存在传输速率差，速率较高的一方即可作为对应的缓存。自底向上，可以简单认为，计算机系统中每一层的速率相差一个数量级（如CPU的响应速率是内存的10倍，内存则是硬盘的10倍等）都会用到Cache。

本章将以网络应用的缓存为主进行介绍。

一个网站或者应用的一般形式是：浏览器向应用服务器发出请求，应用服务器做一堆计算和逻辑判断后再向数据存储层发出请求，数据存储层（以下统称为数据库）收到请求后再经计算把数据返回给应用服务器，应用服务器再次计算后把数据返回给浏览器。这是标准流程。

8.1.1 为什么使用缓存

随着Web业务的复杂和并发的增加，应用服务器和数据库服务器所做的计算也越来越多，但是我们的应用服务器资源是有限的，数据库每秒接受并处理请求的次数也是有限的。如何利用有限的资源提供尽可能大的吞吐量呢？

一个办法：减少计算量，缩短请求流程（减少网络I/O或者硬盘I/O），这时缓存就可以大展身手了。缓存的基本原理就是打破标准流程，在标准流程中任何环节都可以被切断。请求可以从缓存里取到数据直接返回。这样能够节省时间、提高响应速度、节省硬件资源，可以让有限的硬件资源服务更多的用户。

在Web世界，理论上每一层都可以被缓存。以PHP应用为例：

底层有CPU缓存、磁盘文件系统缓存。

应用层有Zend虚拟机的变量缓存，有Memcached这样的Key Value内存缓存，有APC、eAccelerator这类基于Opcode字节码的缓存。

数据库层有Table Cache、Thread Cache、Query Cache。

Servlet容器层有Apache的缓存。

Servlet再上去一点，有一个Web Cache层（如Squid、Varnish等），然后应用程序代码级别的Smarty实现的文件缓存。基于HTTP协议和浏览器自身实现的浏览器缓存。当然，上面的分类只是大致描述，其中存在重复和交叉。

此处，NoSQL系列缓存（NoSQL不是Not SQL，而是Not only SQL），包括Memcached、Cas

sandra、MongoDB、Redis、Tokyo Tyrant等产品。NoSQL同样是一种存储数据的数据库，主要基于内存和Key Value模式（此外，还有基于document、XML的）。由于NoSQL通常基于内存，所以同时还具有内存缓存的作用。故常有人困惑或争执其是缓存软件还是数据库软件。其实这可以从其命名看出，如Memcached偏向于缓存，追求速度和性能；MongoDB偏向于数据库，数据类型较丰富。因此，NoSQL作为一种统称，同时具有Key Value数据库和内存缓存的功能。第9章将详细介绍其原理和使用方法。

缓存放在什么地方呢？无非就是内存和硬盘。文件缓存（如模板机制）自然缓存在硬盘上，而一些需要高速存取的变量则缓存在内存中。模板的缓存把动态代码编译成静态文件放入硬盘，不用每次访问都编译，直接读出即可。

通常来说，缓存组件都是同时结合内存和硬盘的，当内存满后，把部分数据持久化存到硬盘，或定期dump把内存的数据写入硬盘，防止数据丢失。

缓存有三个要素：命中率、缓存更新策略、缓存最大数据量。下面分别介绍。

8.1.2 命中率

通常通过命中率衡量缓存机制的好坏和效率。

命中率指请求缓存次数和缓存返回正确结果次数的比例。比例越高，证明缓存的使用率越高。缓存机制比较好理解，也很简单。以Query Cache为例。

MySQL的Query Cache用来缓存和Query相关的数据。具体来说，Query Cache缓存客户端提交给MySQL的SELECT语句以及该语句的结果集。就是将SELECT语句和语句的结果做Hash映射关系后保存在一定的内存区域中。

MySQL提供一系列Global Status记录Query Cache当前状态，具体如下：

Qcache_free_blocks：目前处于空闲状态的Query Cache中内存block数目。

Qcache_free_memory：目前处于空闲状态的Query Cache内存总量。

Qcache_hits：Query Cache命中次数。

Qcache_inserts：向Query Cache中插入新的Query Cache次数，也就是没有命中的次数。

Qcache_lowmem_prunes：当Query Cache内存容量不够，需要从中删除旧的Query Cache以给新Cache对象使用的次数。

Qcache_not_cached：没有被Cache的SQL数，包括无法被Cache的SQL以及由于query_

cache_type设置而不会被Cache的SQL。

Qcache_queries_in_cache：目前在Query Cache中的SQL数量。

Qcache_total_blocks：Query Cache中总的block数量。

提示 可以通过执行SHOW GLOBAL STATUS查看GLOBAL STATUS。

按照SQL Cache规则，只有SELECT语句被缓存。插入、删除、更新不需要进行缓存，同时SHOW命令和存储过程（包括存储过程中的SELECT）也不会进入缓存结果集。Query Cache命中率计算公式如下：

$$\text{hit rate} = \text{Qcache_queries_in_cache} / \text{Com_select}$$

正常缓存命中率是多少呢？不同缓存应用中其值大相径庭。以Query Cache为例，经过服务器一段时间的运行和积累，其命中率通常都能达到98%以上。而对于另外一些缓存应用，缓存命中率能达到85%以上就已经很高了，达到98%是理想状态。这跟缓存机制的实现有很大关系。一般越复杂的缓存机制，越难保证命中率，并且命中率的提高还需要积累和练习。随着服务器的运行和积累，缓存命中率会逐渐增长直至稳定状态。

需要注意的是，如果数据频繁更新，就需要考虑缓存的合理性。因为频繁更新会使命中率大大降低。以Query Cache为例，Query Cache具有失效机制，如果表中的数据变化比较频繁，大量Query Cache频繁失效，命中率就可能比较低。这种场景下，Query Cache不仅不能提高效率，反而可能造成负面影响。

8.1.3 缓存更新策略

MySQL的Query Cache缓存更新策略很简单。在MySQL中，可以设置Query Cache所使用的总内存，MySQL会把默认可以进行缓存的SQL语句的结果集进行缓存，一旦内存塞满后，就会剔除老的Query Cache对象。同时，为了保证Query Cache中的内容与是实际数据绝对一致，当表中的数据有任何变化，包括新增、修改、删除等，都会使所有引用到该表的SQL的Query Cache失效。

一般把缓存更新策略归纳为以下几种：

FIFO[First In First Out]。最先进入缓存得数据在缓存空间不够情况下（超出最大元素限制时）会被首先清理出去。

LFU[Less Frequently Used]。最少使用的元素会被清理掉。这要求缓存的元素有hit属性，在缓存空间不够得情况下，hit值最小的将会被清出缓存。

LRU[Least Recently Used]。最近最少使用的元素被清理。缓存的元素有一个时间戳，当缓存容量满了，而又需要腾出地方缓存新元素时，现有缓存元素中时间戳离当前时间最远的元素将被清出缓存。

思考 MySQL的Query Cache属于什么策略？

根据前面的描述，可以判断出，MySQL没有对每一条Query Cache的使用进行维护，当内存满后，简单地清除最早的数据，那么，它应该属于FIFO策略，即队列清除。不少缓存组件都使用了队列这种简单的策略。

8.1.4 缓存最大数据量

缓存最大数据量是在缓存中能处理元素的最大个数或所能使用的最大存储空间。通常各种缓存机制都会对缓存最大数据量进行限定，可以是固定大小的存储空间、集合个数，或者由操作系统所能分配和处理的存储空间决定。

例如，MySQL的Query Cache缓存最大数据量由`query_cache_size`参数决定，并且可以修改。而基于内存的Key Value实施方案Memcached，其缓存最大数据量可使用内存由操作系统决定，默认为64MB，每次最大可申请内存为2MB。

超过缓存机制所允许的最大数据量系统会进行相应处理，一般有四种处理方式：

停止缓存服务，所有缓存数据被清空。

拒绝写入，不再对缓存数据进行更新。

根据缓存更新策略清除旧数据。

在方式3基础上，将淘汰的数据备份，腾出新的空间。

实际应用中，通常以方式3、方式4最为常见。

8.2 文件缓存

文件缓存是把缓存数据存储到文件系统即硬盘中。与内存相比，硬盘属于比较慢的存储设备。那为什么还需要用到文件缓存呢？原因如下：

磁盘容量大，可以存放足够多的数据。现在的常规硬盘已经进入TB级别，但内存还处于GB级别（1TB=1024GB）。磁盘价格远远低于内存价格，通常只有同样大小内存售价的百分之一到十分之一。

磁盘与内存相比更稳定更可靠，断电后数据不丢失，存储也比较简单可靠。

随着制造技术的进步，出现了固态硬盘（Solid State Disk, SSD），使硬盘的读取和写入速度得到极大的提高，能达到500Mb/s。

扩展容易。可以使用磁盘阵列、分布式处理等进行大规模的存储和管理。

由于以上几点原因，文件缓存在Web应用中常见于模板引擎和配置文件的处理中。

8.2.1 文件缓存机制

文件缓存机制逻辑很清晰，模板的作用之一就是把动态PHP代码“编译”成静态的HTML文件，当下次读取时不用再“编译”直接读取静态文件。只需要一系列文件函数就能处理。

下面是一个简化的模板引擎进行文件缓存的实现。此模板引擎没有实现标签、局部编译等功能，只把PHP文件编译成HTML静态文件，实现最简单的文件缓存。具体如代码清单8-1所示。

代码清单8-1 简单的文件缓存实现demo

```
<? PHP
//部分代码略
//返回编译文件的文件名，以后存在路径处理故封装
```

```
public function path () {
return $this->cache_file=$this->arrayConfig[ ' cachedir ' ].
$this->file.$this->arrayConfig[ ' suffix_cache ' ];
}
//开始编译
public function compile () {
$this->message=ob_get_contents ();
@file_put_contents ( $this->path (), $this->message );
}
/**
*显示模板
*@param string $file模板名
*/
public function show ( $file ) {
$this->file= $file;
$PIG_TPL_FILE= $this->arrayConfig[ ' templateDir ' ]. $file. $this->arrayConfig[ ' suffix
'];
//先判断缓存文件是否存在，再看是否过期
if ( is_file ( $this->path () ) ) {
$old= $_SERVER[ ' REQUEST_TIME ' ]-filetime ( $this->path () ) >= $this->
arrayConfig[ ' cache_
time ' ]? 1: 0; //判断缓存是否过期
} else {
$old=0;}
if ( true== $this->arrayConfig[ ' cache_htm ' ] ) { //如果需要静态编译
if ( is_file ( $this->path () ) &&! $old ) {
readfile ( $this->path () ); //静态编译文件存在且未过期
} else {
if ( is_file ( $PIG_TPL_FILE ) ) {
ob_start ();
extract ( $this->value );
include ( $PIG_TPL_FILE ); //否则就重新编译
$this->compile ();
} else { die ( ' 找不到模板文件. ' ); }
}
} else {
//不需要编译
if ( is_file ( $PIG_TPL_FILE ) ) {
extract ( $this->value );
include ( $PIG_TPL_FILE ); //否则就加载原始文件并保存
```

```
} else { die ( ' 找不到模板文件: ' . $PIG_TPL_FILE ); }  
}  
} //end show  
//end file? >
```

这个简单的模板引擎使用最简单的文件缓存，思路如下：

1) 先根据配置文件判断是否要进行缓存，若不需要缓存，则直接include加载PHP文件。若需要缓存，则转到下一步。

2) 判断对应的静态文件，此处即缓存文件是否存在，若不存在，则进行“编译”，将编译内容保存为静态的HTML文件。否则，转到下一步。

3) 判断静态文件是否过期，若未过期则读取，否则重新编译。

尽管文件存储受到磁盘I/O效率较低的影响，但是在Web中，类似模板编译后存储到静态文件这样的做法有很大优势。用较小I/O换取Web服务器与数据库的交互，以及PHP解释引擎对脚本进行解释的消耗，而得到较大回报。

有时，我们会把一些常用的或比较耗费数据库资源的大变量缓存起来。比如在无限分类的应用中，当分类很多时，每次查询都要动用数据库，而分类又很少改动。偏偏对分类的查询和操作涉及的数据量都很大，往往效率极低。这时，常常这么做：

```
$data = $db->fetchAll ($sql);  
file_put_contents ( ROOT. ' cache/cat.PHP ', var_export ($data, TRUE ));
```

需要时可打开文件，一句代码就可以解开数据：

```
eval ( "\ $data = $data; ")
```

8.2.2 文件缓存开源产品Secache

这里介绍一个开源产品Secache，该产品是由ShopEx团队开发，下载地址：
http: //

code.google.com/p/secache/.

Secache是文件型缓存解决方案，其特点如下：

纯PHP实现，无须任何扩展，支持PHP4/5。

使用LRU算法自动清理过期内容。

最大支持1GB缓存文件。

使用hash定位，读取迅速。

在虚拟主机不支持Memcached等高速缓存的情况下，可以考虑采用Secache。

简单地说，Secache是用PHP实现的Key Value数据库，把数据按照键值对方式存储到单文件中。使用方式很简单，如代码清单8-2所示。

代码清单8-2 Secache使用示例

```
require ( '../Secache/Secache.PHP ');
$cache=new Secache;
$cache->workat ( ' cachedata ');
$key=md5 ( ' test '); //必须自己做hash, 前4位是16进制0-f, 最长32位
$value= ' 值数据 '; //必须是字符串
$cache->store ( $key,$value );
if ( $cache->fetch ( $key,$return )) {
echo ' <li> ' . $key. ' => ' . $return. ' </li> ';
} else {
echo ' <li>Data get failed! <b> ' . $key. ' </b> </li> ';
}
```

Secache的使用方法如我们在PHP中使用数组一样简单，在一些小型应用中是很好的缓存解决方案。

Secache模仿Memcached存储机制，是一个Key Value二进制数据库，使用Hash索引方式快速查找定位到相应文件在数据文件中的位置，并且采用拉链法解决冲突。

Secache使用LRU缓存策略。当Secache data中某一存储区中数据超过限制时，会根据LRU算法得出该存储区中需要淘汰的缓存文件供新文件存入。

和Memcached类似，Secache使用Slab（数据块）形式存储数据。把整个文件分成等级不同的存储区，每个存储区有大小相同的多个数据块，按照存储数据的大小选择最合适的存储区，按顺序把要存储的数据放入数据块中，以充分利用存储空间，并使文件结构更紧凑。文件结构如图8-1所示。



图 8-1 Secache文件结构图

从某种意义上说，数据库存储也是文件存储的一种。数据库可以看做优化过的、高效的文件存储系统。什么样的数据适合存放到数据库中，什么样的数据又适合存放到文件中呢？

对于变动比较少，体积比较大的数据适宜使用文件存储；反之，使用数据库存储比较合适。另外，数据库解决了文件存储中很难解决的数据同步和锁的问题。

8.3 Opcode缓存

一个PHP程序在运行完后，内存马上释放，基本上所有数据都在此时销毁（仅有极少数据会缓存），也就是说此时计算机内存中基本上不存在这个PHP代码中的数据，这和常驻内存的Java等语言有显著区别。这种缓存机制具有如下的优缺点：

优点：有效避免内存泄露，内存回收机制更简单，避免因为一个程序的问题而连累整个服务器。

缺点：无法复用已有数据，每个PHP请求都得重复执行请求-翻译-执行的过程，重复过多。

Opcode（Operation Code，操作码）缓存就是虚拟机把PHP代码编译成一种中间码的结果缓存起来（可以缓存到硬盘或内存）。下一次PHP运行此页面时，只要直接解释这些代码就行了。这样省去了Flex语法器进行语法编译和大部分语法检查（这个语法检查在多个阶段均存在）的过程，一定程度上提高了PHP运行速度，减轻了服务器负荷。

注意 Opcode不是PHP的专有名词。PHP的Opcode是一种PHP脚本编译后的中间语言，就像Java的ByteCode，PHP的语言引擎Zend执行PHP代码时，会把PHP代码经过分成Token，词法分析的过程转成Opcode，然后顺序执行。

8.3.1 eAccelerator下载及使用

eAccelerator工具能够起到“常驻内存”的作用。

eAccelerator支持Linux和Windows系统，可到<http://eaccelerator.net/>处下载对应PHP版本的Linux源码，可到http://www.sitebuddy.com/PHP/Accelerators/eAccelerator_windows_binaries_

[builds](#)处下载对应PHP版本的Windows源码。

eAccelerator使用和配置都很简单，以Windows版为例，把下载的对应该版本的

DLL文件拷贝到PHP的ext目录下，在php.ini文件中增加如下配置项：

```
[eaccelerator]
extension="E: \ dev \ PHP \ ext \ eAccelerator533ts.dll"//eaccelerator可安装为Zend扩展或普通
扩展
eaccelerator.shm_size="16"//共享内存大小
eaccelerator.cache_dir="e: \ temp \ eaccelerator"//缓存文件的目录
eaccelerator.enable="1"//1启动eaccelerator, 0关闭
eaccelerator.optimizer="1"//允许加速脚本执行
eaccelerator.log_file="E: \ debug \ PHP_ex.log"; //日志文件, 存放缓存命中的文件
eaccelerator.name_space=""//缓存中的key前缀, 在虚拟主机中可在.htaccess里设置
eaccelerator.check_mtime="1"//是否自动检查缓存过期, 建议关闭
eaccelerator.debug="0"//不记录日志
eaccelerator.filter=""
eaccelerator.shm_max="0"//共享内存中可put进来的最大尺寸
eaccelerator.shm_ttl="0"//回收内存在未被使用过的文件
eaccelerator.shm_prune_period="0"//多少秒回收一次内存在未被使用过的文件
eaccelerator.shm_only="0"//1禁止缓存在磁盘上编译过后的文件, 0允许
eaccelerator.compress="1"
eaccelerator.compress_level="9"//最大压缩
```

在服务器第一次请求PHP文件时，会对PHP文件的Opcode进行缓存；再次请求时，直接取出缓存的Opcode，由Zend虚拟机直接执行，从而节省了语法解析的消耗。

8.3.2 如何查看Opcode

首先安装VLD扩展 (Vulcan Logic Disassembler), 用来检测PHP脚本的执行情况。VLD扩展没有编译好的DLL或SO文件, 需要自行编译。

Linux下安装VLD的方法如下:

```
wget http://pecl.PHP.net/get/vld
tar zxvf vld-0.9.1.tgz
cd vld-0.9.1
phpize
./configure
make install
```

编辑php.ini文件激活VLD扩展的方法如下:

```
extension=vld.so
```

如果在Windows下, 需要自行配置PHP编译环境, 整个编译过程比较烦琐, 详细参考网上的文章。现在给出一个文件测试代码:

```
<? PHP
echo"hellword \ r";
$data[ ' first ' ]= ' Hello ';
$data[ ' first ' ][ ' second ' ]= ' world ';
echo $data[ ' first ' ];
```

在命令行下执行:

```
php-dvld.active=1 g: \ bak \ temp \ tempcode \ str.php
```

得到输出如图8-2所示。

这段代码分成13步执行。学过汇编和编译原理课程的读者，对这样形式的代码应该比较熟悉。没错，Opcode就是PHP“汇编代码”。

配置好eAccelerator，运行PHP就能在预设的缓存目录找到被eAccelerator缓存的文件。可以到官方网站（<http://eaccelerator.net/browser/eaccelerator/trunk/control.php>）下载control.php文件，把它和源代码中的dasm.php、info.php放到一个目录下（注意，只有Linux版本源码包才有这两个文件，Windows下复制过去就可以），修改PHP配置文件，设置允许管理员进行操作的执行路径，添加以下字段：

```
eaccelerator.allowed_admin_path=/var/www/html/
```

```

$? \bak\temp\tempcode\str.php
Files used: G:\bak\temp\tempcode\str.php
Function name: Call11
Number of ops: 13
Compiled code: TM - $data
Line  # * op          fetch      ext  return  operands
-----
2  0  >  EXI_STMT
1  ECHO                                           'hello'
opcode
3  2  EXI_STMT
3  ZAMP_SESSION_DIM      18. 'f1
'f1'  4  ZAMP_OP_DIMA
'f1'  5  EXI_STMT
6  FETCH_DIM_U           52. 'f1
'f1'  7  ZAMP_SESSION_DIM
'op'  8  ZAMP_OP_DIMA
'op'  9  EXI_STMT
10  FETCH_DIM_R           55. 'f1
'f1'  11  ECHO
6  12  RETURN                                           1
Branches: # W: line: 2- G: op:  W: op:  E2
path: W: #.
use: l: none
$? \bak\temp\tempcode\str.php
  
```

图 8-2 Opcode示意图

运行以下代码，登录即可在Web界面查看eAccelerator运行信息并进行缓存管理：

```
127.1/control.php, 以admin/eAccelerator账号登录
```

eAccelerator管理界面如图8-3所示。

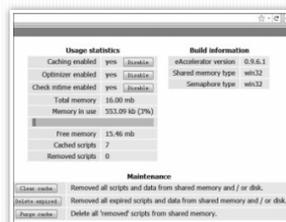


图 8-3 eAccelerator管理界面

eAccelerator缓存Opcode，同时提供一组API用来操作缓存数据。变相实现了类似Java中的“共享变量”功能。我们知道，PHP本身无法实现共享变量，变量在请求完成后就被销毁，无法在另一个进程中得到其数据。借助eAccelerator就能实现类似功能。这主要由下面两个API实现（更多的API接口可以参考官方文档）：

`eaccelerator_put (key, value, ttl=0)`：将value存储在共享内存中，并存储ttl秒。

`eaccelerator_get (key)`：从共享内存中返回eaccelerator_put () 函数所存储的缓存数值，如果不存在或者已经过期，则返回null。

无论eAccelerator还是APC, Opcode缓存软件实现的都只是简单的“共享变量”，不能和具有完整GC功能的Java或.NET中的内存变量相提并论，但是这确实提高了PHP的执行效率，使其接近编译型语言的速度。

8.4 客户端缓存

缓存最后一层，是直接面对客户端的客户端缓存。通常也把这部分称为Web缓存。Web缓存位于客户端。缓存会根据进来的请求保存输出内容的副本，例如HTML页面、图片、文件（统称为副本）等，然后，当下一个请求来到时，如果是相同的URL，缓存直接使用副本响应访问请求，而不是向源服务器再次发送请求。

Web缓存的具体实现是由浏览器来实现的。浏览器在计算机上开辟一块硬盘空间用于存储已经看过的网站的副本。浏览器缓存根据非常简单的规则进行工作：在同一个会话过程中（即当前浏览器没有被关闭之前）会检查一次并确定缓存的副本足够新。这个缓存对于用户单击“后退”或者单击刚访问过的链接特别有用，如果你浏览过程中访问到同一个图片，这些图片可以从浏览器缓存中调出并即时显现。

8.4.1 客户端缓存规则

前端页面缓存主要遵循HTTP协议和客户端（或服务器端）的设置工作。通常遵循的规则如下：

如果响应头信息告诉缓存器不要保留缓存，缓存器就不会缓存相应内容。

如果请求信息需要认证或者安全加密，相应内容也不会被缓存。

如果在回应中不存在校验器（ETag或者Last Modified头信息），缓存服务器会认为缺乏直接的更新度信息，内容将会被认为不可缓存。

一个缓存的副本如果含有以下信息，内容将会被认为足够新：

含有完整的过期时间和寿命控制头信息，并且内容仍在保鲜期内。

浏览器已经使用过缓存副本，并且在一个会话中已经检查过内容的新鲜度。

缓存代理服务器近期内已经使用过缓存副本，并且内容的最后更新时间在上次使用期之前。

够新的副本将直接从缓存中送出，而不会向源服务器发送请求。

如果缓存的副本已经太旧了，缓存服务器将向源服务器发出校验请求，用于确定是否可以继续使用当前拷贝继续服务。

8.4.2 HTTP协议中的缓存使用

HTTP协议中有很多篇幅用来描述缓存。HTML文档的< head > 区域中加入描述文档的各种属性，这些Meta标签常常被用于标记文档不可以被缓存或者标记多长时间后过期。

当浏览器读取到此HTML页面时，会遵循这个标记，在客户端采取针对性的缓存措施。除了直接在HTML文档里进行标记外，还可以在Web服务器端进行标记。

在客户端，通过浏览器发出第一次请求，请求某一个URL时，根据HTTP协议的规定，浏览器会向服务器传送报头（HTTP Request Header），服务器端响应同时记录相关属性标记（HTTP Reponse Header），服务器端的返回状态会是200，格式类似如下：

```
HTTP/1.1 200 OK
Date: Tue, 03 Mar 2009 04: 58: 40 GMT
Content-Type: image/jpeg
Content-Length: 83185
Last-Modified: Tue, 24 Feb 2009 08: 01: 04 GMT
Cache-Control: max-age=2592000
Expires: Thu, 02 Apr 2009 05: 14: 08 GMT
Etag: "5d8c72a5edda8d6a: 3239"
```

客户端第二次请求此URL时，根据HTTP协议的规定，浏览器会向服务器传送报头（HTTP Request Header），服务器端响应请求，并查询到标记文件没有发生改动，服务器端返回304，浏览器收到此状态码后，直接从本地缓存中读取：

```
HTTP/1.x 304 Not Modified
Date: Tue, 03 Mar 2009 05: 03: 56 GMT
Content-Type: image/jpeg
Content-Length: 83185
Last-Modified: Tue, 24 Feb 2009 08: 01: 04 GMT
```

```
Cache-Control: max-age=2592000
Expires: Thu, 02 Apr 2009 05: 14: 08 GMT
Etag: "5d8c72a5edda8d6a: 3239"
```

其中Last Modified、Expires和ETag是页面缓存标识。

1.Expires

Expires（过期时间）属性是HTTP控制缓存的基本手段，其告诉缓存器相关副本在多长时间是新鲜的。过了这个时间，缓存器就会向服务器发送请求，检查文档是否被修改。几乎所有缓存服务器都支持Expires属性。Expires头信息对于设置静态图片文件可缓存特别有用。因为这些图片修改很少，你可以给它们设置一个特别长的过期时间，这会使用户网站的用户相应变得非常快。

过期时间头信息属性值只能是HTTP格式的日期时间，其他的都会被解析成当前时间“之前”，副本会过期。记住：HTTP的日期时间必须是格林威治时间（GMT），而不是本地时间，例如：Expires: Fri, 30 Oct 1998 14: 19: 41 GMT。

2.Cache Control

HTTP 1.1介绍了另外一组头信息属性——Cache Control响应头信息。通过这个属性可让网站发布者全面控制内容，并定位过期时间的限制。

有用的Cache Control响应头信息包括：

max age=[秒]：缓存过期时间，这个参数是基于请求时间的相对时间间隔，而不是绝对过期时间，[秒]是一个数字，单位是秒：从请求时间开始到过期时间之间的秒数。

s maxage=[秒]：类似于max age属性，不同的是，它应用于共享（如代理服务器）缓存。

public：标记认证内容也可以被缓存，一般来说：经过HTTP认证才能访问的内容，输出是自动不可以缓存的。

no cache：强制每次请求直接发送给源服务器，而不经本地缓存版本的校验。这

对于需要认证的应用（可以和public结合使用）或者严格要求使用最新数据的应用（不惜牺牲使用缓存的所有好处）很有用。

no store: 强制缓存在任何情况下都不要保留任何副本。

must revalidate: 如果服务器端明确指出资源的过期时间或者保鲜时间，而且声明了资源的修改时间或者ETag之类的标识，那么就有一个问题了，在保鲜时间内，如果用到了该资源，是不是要到服务器确认一下资源是否最新的？如果服务器声明了must revali date，则每次使用该资源都需要确认资源新鲜性。

proxy pevalidate: 和must revalidate类似，只对缓存代理服务器起作用。

举例:

```
Cache-Control: max-age=3600, must-revalidate
```

Last Modified: 文档最后修改时间。

在浏览器第一次请求某一个URL时，服务器端的返回状态会是200，内容是你请求的资源，同时有一个Last Modified的属性标记（HTTP Reponse Header）此文件在服务期端最后被修改的时间，格式类似这样:

```
Last-Modified: Tue, 24 Feb 2009 08: 01: 04 GMT
```

客户端第二次请求此URL时，根据HTTP协议的规定，浏览器会向服务器传送If Modified Since报头（HTTP Request Header），询问该时间之后的文件是否有被修改过，例如:

```
If-Modified-Since: Tue, 24 Feb 2009 08: 01: 04 GMT
```

如果服务器端的资源没有变化，则自动返回HTTP 304 (Not Changed.) 状态码，内容为空，这样就节省了传输数据量。当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

注意 如果If Modified Since的时间比服务器当前时间（当前的请求时间request_time）还晚，会认为是个非法请求。

3.ETag

服务器生成的唯一标识符ETag，每次副本的标签都会变化。HTTP协议规格说明定义ETag为“被请求变量的实体标记”。简单点说即服务器响应时给请求URL标记，并在HTTP响应头中将其传送到客户端，类似服务器端返回的格式，例如：

```
Etag: "5d8c72a5edda8d6a: 3239"
```

客户端的查询更新格式是这样的：

```
Etag: "5d8c72a5edda8d6a: 3239"  
If-None-Match: "5d8c72a5edda8d6a: 3239"
```

如果ETag没改变，返回状态304。在客户端发出请求后，HTTP Reponse Header中包含ETag: "5d8c72a5edda8d6a: 3239"标识，等于告诉Client端，你拿到的这个资源又表示ID: 5d8c72a5edda8d6a: 3239。下次需要发请求并索要同一个URI时，浏览器同时发出一个If None Match报头（HTTP Request Header），此时报头信息包含上次访问得到的ETag: "5d8c72a5edda8d6a: 3239"标识。

```
If-None-Match: "5d8c72a5edda8d6a: 3239"
```

这样Client端等于缓存两份，服务器端就会比对二者的ETag。如果If None Match为False，不返回200，返回304 (Not Modified) Response。

ETag和Last Modified都能起到文档唯一性标识的作用。

小窍门

- 1) 不经常改变的图片/页面启用缓存，使用Cache Control: max age属性设置一个较长的过期时间。
- 2) 定期更新的内容设置一个缓存服务器可识别的max age属性或过期时间。
- 3) 尽量避免使用POST，除非万不得已，POST模式的返回内容大部分缓存服务器不会保存，而如果发送内容通过URL或GET模式发送，那么发送的内容可以缓存下来供以后使用。
- 4) 不要在URL中加入针对每个用户的识别信息：除非内容是针对不同用户的。

8.4.3 HTTP缓存实例

最后，通过一个实例演示HTTP缓存的应用。

可以在HTML页面利用meta tag和在PHP程序中通过header来控制缓存的生成和过期策略。例如：

```
<? PHP
header ( ' Cache-Control: max-age=86400, must-revalidate '); //24小时
header ( ' Last-Modified: ' . gmdate ( ' D, d M Y H: i: s ' ) . ' GMT ');
header ( ' Expires: ' . gmdate ( ' D, d M Y H: i: s ', time () + ' 86400 ' ) . ' GMT ');
echo ' 我不刷新 ';
```

写个HTML文件c.htm:

```
<html> <body>
haha, <a href=cache.PHP>go </a>
</body> </html>
```

请求127.1/c.htm，单击页面中的链接，利用浏览器的回退按钮返回c.htm，再单击此页中的链接，如图8-4所示，当添加缓存指令后，无论如何后退和单击链接，网络请求URL这一条始终为灰色，表示浏览器并没有发起实际的网络请求，而是直接调用存储在用户电脑中的缓存页，除非缓存时间过期。在这期间，即使是实际内容改变了，浏览器也不会去重新读取我们在服务器上的资源。可以把echo那一句修改后，再单击会发现网络请求仍然为灰色。



图 8-4 浏览器缓存示意图

只有出现三种情况浏览器才会更新缓存：缓存到期；缓存被清除；按F5或Ctrl+F5键强制刷新。（这一点各种浏览器处理可能存在差异，本机测试环境是Firefox 4）。

屏蔽上面的header指令或者改用如下代码时，浏览器会在每次请求时去服务器读取资源：

```
//告诉客户端浏览器不使用缓存，HTTP 1.1协议
```

```
header ("Cache-Control: no-cache, must-revalidate");
```

```
//告诉客户端浏览器不使用缓存，兼容HTTP 1.0协议
```

```
header ("Pragma: no-cache");
```

这里，大致了解浏览器对页面的缓存处理。通过这部分介绍，我们很容易就能联想到缓存的使用场景，有时候应用需要缓存，有时候又不需要，这需要根据场景具体的使用而定。浏览器缓存的作用虽然很微小，但能省一点算一点，而且浏览器端缓存是免费的。

有些浏览器会缓存静态资源，并且普通用户很少会对缓存进行清理，这会导致虽然开发人员已在服务器上更新了文件，但是用户一直没有看到更新情况。这种情况下，需要用户强制刷新或者清空浏览器本地缓存才能使缓存失效。此时我们通常采取在静态资源后附加一个时间戳的方式避免缓存，如：

```
<script src="link.js? d=1923454332"> </script >
```

在JavaScript文件后加上问号的那一串字符并没有实际意义，只是为了避免JavaScript被缓存到本地而不能及时更新，其算是给JavaScript文件加的一个“版本号”，通常为时间戳或者版本标记（如link.js? Version=1.2.0）。这样浏览器就会认为这是一个新资源，从而发起请求，更新本地缓存。

需要注意的是，有时这种方法仍然不会刷新缓存，如有的服务器加了代理服务器，其忽略静态文件名后的字符串，对请求不予响应。这时，最好直接更改文件名，如link_20122.js。

既然前端页面缓存的目的是减少请求，那么，在服务器端压缩数据，就是为了减少数据传输量。使用一些工具可以对CSS和JavaScript代码进行压缩。比如jQuery开发版就是经过压缩的，这个压缩比例往往还很大。通常使用jsmin、jspacker进行压缩。

除了对CSS、JavaScript进行压缩外，还可以使用服务器端的Gzip功能对页面进行压缩，相应的压缩文件可由浏览器端进行解压，这将很大程度上减少浏览器和服务器端的数据传输量，但是一定程度上会加剧服务器的负担。

图片资源也能进行压缩和优化。比如使用CSS Sprite技术减少HTTP请求次数，对体积较大的照片等使用GD和ImageMagick生成缩略图。

8.4.4 HTML 5中的Application Cache

HTML 5中新增一个特性Application Cache接口，用来处理离线应用中的一些问题。使用这个接口会给应用带来三方面的优势：

离线浏览：用户在不能联网时依然能浏览整个站点。

高速：缓存资源是存储在本地的，能更快加载。

更小的服务器负载：浏览器只需从服务器端下载有改变的资源，相同资源不重复下载。

Application Cache（或AppCache）指定浏览器需要保存哪个文件。用户在离线情况下，即使按了刷新按钮，应用也能正确加载和工作。

Application Cache配置比较简单，指定一个Manifest文件配置本地缓存即可，具体如下：

```
<html manifest="example.mf">  
.....  
</html>
```

如果一个页面没有Manifest属性，将不会被缓存（除非在Manifest文件中显式指定这个页面）。这意味着只要用户访问的页面包含Manifest属性，都将被加入Application Cache中。这样，不用在Manifest文件中指定需要缓存哪些页面。

Manifest属性可以指定一个绝对URL或一个相对路径，但是，一个绝对URL需要和Web App同源。一个Manifest文件可以是任何扩展文件类型，但必须有正确的MIME Type。可以在Web服务器端的配置文件加上这个属性。以Apache为例，在httpd.conf里加上一句话：

```
AddType text/cache-manifest.mf
```

Manifest文件内容如下所示:

```
CACHE MANIFEST
index.html
style.css
images/logo.jpg
scripts/main.js
```

需要注意，第一行“CACHE MANIFEST”字符串是必不可少的。在Manifest文件中，还可以使用#添加注释，注释内容可以是时间戳等，以通知浏览器更新策略。

一个应用只有在Manifest文件发生变化时才会更新缓存。例如，如果编辑图像或改写一个JavaScript函数，缓存并不会发生更新。必须改写Manifest文件本身来通知浏览器需要更新缓存文件，缓存才会更新。

一个应用在离线情况会保持它的缓存状态，除非有以下事件发生则会更改其缓存状态:

用户清除浏览器中存储的站点数据。

Manifest file被修改。

注意 修改了在Manifest文件中列出的某个文件并不会让浏览器重新缓存资源。必须是Manifest文件本身改变了，才会重新进行缓存。

App Cache通过编程更新了。

要在脚本中更新缓存也比较简单，只要操纵window.applicationCache对象就可以了，相应代码如下:

```
var appCache=window.applicationCache;
```

```
var status=appCache.status; //缓存状态, 0-5之间
console.log ( status );
appCache.update ( ); //更新缓存
if ( appCache.status==window.applicationCache.UPDATEREADY ) {
appCache.swapCache ( ); //如果获取成功, 清除旧数据, 用新数据取代
}
```

要了解关于Application Cache的更多内容, 可以阅读HTML 5相关文档。

8.5 Web服务器缓存

在Web层面上的缓存，除了基于HTTP协议加浏览器实现外，还可通过一些Web服务器自带的缓存组件，以及服务器和浏览器之间的代理服务器提供的缓存功能实现。下面对这类缓存做简单介绍。

8.5.1 Apache缓存

Apache的Expires和Cache Control模块包含控制缓存的信息。这些模块需要和Apache一起编译。虽然它们已经包含在发布版本中，但默认并没有启用。确定相应模块已经被启用的方法是：找到httpd程序并运行`httpd -l`会列出可用模块，要用的模块是`mod_expires`和`mod_headers`。

Apache一旦启用了相应模块，就可以在`.htaccess`文件或者服务器的`access.conf`文件中，通过`mod_expires`设置副本过期时间了，包括设置控制应答时的Expires头内容和Cache Control头的`max age`指令，代码如下：

```
ExpiresActive On
ExpiresByType image/gif"access plus 1 month"
ExpiresByType image/jpg"access plus 1 month"
ExpiresByType image/jpeg"access plus 1 month"
ExpiresByType image/x-icon"access plus 1 month"
ExpiresByType image/bmp"access plus 1 month"
ExpiresByType image/png"access plus 1 month"
ExpiresByType text/html"access plus 30 minutes"
ExpiresByType text/css"access plus 30 minutes"
ExpiresByType text/txt"access plus 30 minutes"
ExpiresByType text/js"access plus 30 minutes"
ExpiresByType application/x-javascript"access plus 30 minutes"
ExpiresByType application/x-shockwave-flash"access plus 30 minutes"
```

也可用以下代码进行设置：

```
< ifmodule mod_expires.c >
< filesmatch "\.(jpg | gif | png | css | js)$" >
ExpiresActive on
ExpiresDefault "access plus 1 year"
< /filesmatch >
< /ifmodule >
```

设置Expires后，系统会自动输出Cache Control的max age信息，关于Expires详细内容可以查看Apache官方文档。

如果在Windows系统下使用Apache，建议到<http://www.apachelounge.com/download/>处下载改进的Apache服务器以及相关的缓存组件。Apache lounge是Apache的改进版，在性能、稳定性和内存管理上都超过基于VC 6编译的Apache官方版本。Apache lounge版本随着Apache官方版本同步更新，所以不需要担心版本和安全问题。

8.5.2 Nginx缓存

Nginx (eNginx) 是高性能HTTP和反向代理服务器，也是IMAP/POP3/SMTP代理服务器。由Igor Sysoev为Rambler.ru站点（俄罗斯访问量第二）开发，它已经在该站点运行超过四年。Igor将源代码以类BSD许可证的形式发布。Nginx因为稳定性、丰富的功能集、示例配置文件和低系统资源消耗而闻名。目前，国内各大门户网站已经部署了Nginx，如新浪、网易、腾讯等；国内几个重要视频分享网站也部署了Nginx。Nginx技术在国内日趋火热，越来越多的网站开始部署Nginx。

Nginx体积小、配置简单、扩展性强，通过众多开源模块发挥强大的功能。Nginx性能远远超越传统的Apache。Nginx配合PHP的FastCGI模式，充分利用PHP天生的优势，具有极大的负载能力。

Nginx还是反向代理软件，可实现负载均衡和集群。

在Nginx中可以实现传统的缓存以及基于proxy_cache的缓存。

从Nginx 0.7.44版开始支持类似Squid的较正规的缓存功能，但是目前还处于开发阶段，支持相当有限。这个缓存把链接用md5编码经哈希后保存，所以支持任意链接，同时支持404/301/302这样的非200状态。

配置一个缓存空间的代码如下：

```
proxy_cache_path/path/to/cache levels=1: 2 keys_zone=NAME: 10m inactive=5m max_size=2m clean_time=1m;
```

注意，这个配置在server标签外，levels指定该缓存空间有两层hash目录，第一层1个字母，第二层2个字母，保存的文件名类似于/path/to/cache/c/29/b7f54b2df7773722d382f4809d65029c；keys_zone为这个空间起个名字，10m指空间大小为10MB；inactive的5m指缓存默认时长为5分钟；max_size的2m指单个文件超过2MB就不缓存；clean_time指定1分钟清理一次缓存。

```
location/ {  
    proxy_pass http://www.linuxidc.com/;  
    proxy_cache NAME; #使用NAME这个keys_zone  
    proxy_cache_valid 200 302 1h; #200和302状态码保存1小时  
    proxy_cache_valid 301 1d; #301状态码保存1天  
    proxy_cache_valid any 1m; #其他的保存1分钟  
}
```

此外，还有基于第三方插件的缓存，例如新浪开发的针对Nginx 0.6.39及以下版本的NCache插件，利用Nginx和Memcached实现一部分类似Squid的缓存功能。此插件现在已经停止维护，并且在最新的Nginx稳定版本已经不兼容此插件。因Nginx集成的proxy_cache已经能满足大部分需求，所以其成为在Nginx服务器上实现缓存机制的最流行、最简单的选择。

还有一些代理服务器及加速器等也能实现缓存管理，如Squid、Vanish等，但这类软件和实现方案都有一定的局限性。比如Squid部署虽然比较简单，但是处理能力低；而Vanish虽然处理能力和连接速度都很出色，但是在高负载下存在丢包问题。而Apache模块众多，功能丰富，却损失了处理速度和负载能力。我们应该根据自己实际需求，在充分测试的前提下，选择最合适我们的方案。

8.6 本章小结

本章主要介绍缓存的原理，从不同层次介绍缓存的实现。书中从服务器端介绍到客户端缓存，综合缓存各个阶段，合理运用各种缓存工具，以达到提高性能和速度，降低消耗的目的。

缓存的本质就是在存在两种不同速率介质的情况下，利用速率较大的介质平衡性能。因此，在开发中时刻要想到这一点，即系统中是否存在速率差，怎么利用这种速率差。这就需要开发者拓宽自己的视野，从最前端考虑到最后端。比如，在一些大的互联网公司，对缓存的设计已经不再停留在简单的文件缓存和内存缓存，而是深入到了CPU缓存的级别，通过合理规划变量的内存布局，充分利用CPU的高速缓存区。

缓存的目的就是为了获得性能的提升，从实现成本的角度来说，缓存是实现成本最低的手段。在缓存的策略的设计上，应该以缓存的三要素为参考，做到以最小的付出得到最大的回报。本章中花了较多篇幅讲解客户端缓存，这是一种实现成本最低的方案。

第9章 Memcached使用与实践

Memcached是以LiveJournal旗下Danga Interactive公司的Brad Fitzpatric为首开发的一款内存缓存软件，现在已在Mixi、Hatena、Facebook、VOX、LiveJournal等众多服务中用于提高Web应用扩展性。

9.1 为什么要用Memcached

随着互联网的发展，特别是Web 2.0网站的兴起，传统的关系型数据库（如MySQL、Oracle）开始出现瓶颈，很多方面不能满足我们的要求。例如：

1) 对数据库的高并发读写。

关系型数据库本身就是个庞然大物，处理过程非常复杂和耗时（如解析SQL语句、事务处理等）。如果对关系型数据库进行高并发读写（每秒上万次的访问），那么关系型数据库是不能承受的。

2) 对海量数据的处理。

对于大型SNS网站，每天有上千万条的数据产生（如微博）。对于关系型数据库，如果在一个有上亿条数据的数据表中查找某条记录，效率会低得难以忍受。

使用Memcached能够很好地解决以上问题。

9.2 Memcached的安装及使用

Memcached是高性能的分布式内存缓存服务器，通过缓存数据库查询结果，减少数据库访问次数，以提高动态Web应用的速度和可扩展性。

Memcached有如下特点：

协议简单；

基于libevent的事件处理；

内置内存存储方式；

采用不互相通信的分布式。

Memcached以守护程序的方式运行于一个或多个服务器中，随时接受客户端的连接操作，客户端可以由各种语言编写，如Perl、PHP、Python、Ruby、Java、C#、C等。客户端在与Memcached服务建立连接之后，接下来的事情就是存取对象了。每个被存储的对象都有一个唯一标识符key与之相关联，通过key可以对对象进行存取操作。保存在Memcached的对象实际上放置在内存中，这也是Memcached如此高效快速的原因。

要注意的是，存储这些对象并不是持久的，服务停止之后，里边的数据就会丢失。典型的应用模型如图9-1所示。

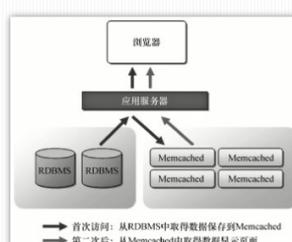


图 9-1 Memcached应用模型

为了提高性能，Memcached把数据存储于内存中。由于数据仅存在于内存中，因此重启Memcached或者操作系统会导致全部数据消失。另外，内存容量达到指定值后，

就会使用LRU (Least Recently Used) 算法自动删除不使用的 (或者很少使用的) 缓存。Memcached本身是为缓存而设计的服务器，因此并没有过多考虑数据的持久化问题。

9.2.1 安装Memcached服务器

安装Memcached比较简单，这里以Ubuntu系统作为安装平台，安装的步骤如下。

1) 在安装Memcached之前，必须先安装依赖库libevent，安装方法如下：

```
$ wget https://github.com/downloads/libevent/libevent/libevent-2.0.15-stable.tar.gz
$ tar -zxvf libevent-2.0.15-stable.tar.gz
$ cd libevent-2.0.15-stable
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

2) libevent安装完毕，开始安装Memcached服务器，安装过程如下：

```
$ wget http://memcached.googlecode.com/files/memcached-1.4.9.tar.gz
$ tar -zxvf memcached-1.4.9.tar.gz
$ cd memcached-1.4.9
$ ./configure --prefix=/usr/local/memcached
$ make
$ sudo make install
```

3) Memcached安装完毕，使用以下命令启动Memcached服务器：

```
$/usr/local/memcached/bin/memcached -d -m 128 -u root -p 11211
```

Memcached常用启动选项及其描述如表9-1所示。

选项	描述
-d	以守护程序 (daemon) 方式运行 Memcached
-m	设置 Memcached 可以使用的内存大小, 单位为 MB
-l	设置监听的 IP 地址, 如果是本机的话, 通常不设置此参数
-p	设置监听的端口, 默认为 11211, 也可以不设置此参数
-u	指定用户, 如果当前为 root, 需要使用此参数指定用户

除以上5个常用启动选项外, 还有很多其他选项, 通过以下命令可了解更多的启动选项信息:

```
$/usr/local/memcached/bin/memcached-h
```

9.2.2 安装Memcached客户端

Memcached客户端非常丰富，本节使用PHP的memcache扩展进行解说。

1) 安装PHP的memcache扩展过程如下：

```
$ wget http://pecl.php.net/get/memcache-2.2.5.tgz
$ tar-zxvf memcache-2.2.5.tar
$ cd memcache-2.2.5 $ phpize
$ ./configure --enable-memcache --with-php-config=/usr/local/php/bin/php-config --
with-zlib-dir
$ make
$ sudo make install
```

2) 在PHP配置文件php.ini中加入以下配置：

```
extension=/usr/local/php/lib/php/extensions/no-debug-non-zts-20120625/memcache.so
```

3) 重启Web服务器。

如果安装成功，可以通过phpinfo()获得该扩展的相关信息，如图9-2所示。

安装完成后，便可以使用memcache扩展与Memcached服务器进行交互了。

下面使用memcache扩展编写一个简单例子：

```
<? PHP
$mc=new Memcache (); //创建Memcache对象
$mc->connect ( ' 127.0.0.1 ', 11211 ); //连接Memcached服务器
$mc->set ( ' key ', ' value ', 0, 10 ); //存储数据
$val=$mc->get ( ' key '); //获取数据
```

```
$ mc->delete ( ' key ' ); //删除数据  
$ mc->flush ( ); //强制刷新全部缓存, 即清空Memcached服务器  
$ mc->close ( ); //断开与Memcached服务器的连接  
? >
```

可以看出, 使用memcache扩展非常简单。

memcache		
memcache support	enabled	
Active persistent connections	0	
Version	2.2.5	
Revision	[Revision: 1.113.9]	
Directive	Local Value	Master Value
memcache.allow_failover	1	1
memcache.chunk_size	8192	8192
memcache.default_port	11211	11211
memcache.default_timeout_ms	1000	1000
memcache.hash_function	md5	md5
memcache.hash_strategy	standard	standard
memcache.max_failover_attempts	20	20

图 9-2 memcache扩展信息

memcache扩展还有很多有用的方法, 下面就介绍这些方法。

9.2.3 使用memcache扩展访问Memcached服务器

memcache扩展提供了非常丰富的方法访问Memcached服务器。下面介绍memcache扩展的一些常用的方法。

1.Memcache: connect

connect方法用于连接Memcached服务器，用法如下：

```
bool Memcache: connect ( string $host[, int $port[, int $timeout]])
```

host: 服务器域名或IP。

port: 服务器TCP端口号，默认值是11211。

timeout: 连接持续（超时）时间，单位为秒。默认值1秒，修改此值之前要慎重，过长的连接持续时间可能会导致失去所有的缓存优势。

2.Memcache: addServer

addServer方法用于向对象添加一个服务器（注：addServer没有连接到服务器的动作，所以在Memcached进程没有启动的时候，执行addServer成功也会返回TRUE），用法如下：

```
bool Memcache: addServer ( string $host[, int $port[, bool $persistent[, int $weight[, int $timeout[, int $retry_interval[, bool $status[, callback $failure_callback]]]]]])
```

host: 服务器域名或IP。

port: 端口号，默认为11211。

persistent: 是否使用常连接, 默认为TRUE。

weight: 权重, 在多个服务器设置中占的比重。

timeout: 连接Memcached服务器失效的秒数。

retry_interval: 服务器连接失败时的重试频率, 默认是15秒一次, 如果设置为-1将禁止自动重试。

status: 控制服务器是否被标记为online, 设置这个参数为FALSE并设置**retry_interval**为-1可以使连接失败的服务器被放到一个描述不响应请求的服务器池中, 对这个服务器的请求将失败。默认参数为TRUE, 代表该服务器可以被定义为online。

failure_callback: 失败时的回调函数, 函数的两个参数为失败服务器的**hostname**和**port**。
3.Memcache: add add方法用于向Memcached服务器添加一个要缓存的数据 (如果Memcached服务器已经存在要存储的key, add方法调用失败)。用法如下:

```
bool Memcache: add ( string $key, mixed $var[, int $flag[, int $expire]])
```

key: 缓存数据的键, 其长度不能超过250字节。

var: 缓存数据的值, 其值最大为1MB。

flag: 是否使用ZLib压缩。把flag设置为MEMCACHE_COMPRESSED时, 如果要缓存的数据很小, 不会采用ZLib压缩, 只有数据达到一定大小时才对数据进行ZLib压缩。

expire: 缓存数据的过期时间。0为永不过期, 可使用UNIX时间戳格式或距离当前时间的秒数, 设为秒数时不能大于2592000 (30天)。

注意 key的最大长度为250字节可以从Memcached源码中看到 (版本1.2.8):

```
static void process_update_command ( conn*c, token_t*tokens, const size_t ntokens, int
```

```
comm, bool handle_cas) {  
    .....  
  
    if ( tokens[KEY _ TOKEN].length>KEY _ MAX _ LENGTH ) {  
        out_string ( c, "CLIENT _ ERROR bad command line format" );  
        return;  
    }  
    .....  
}
```

代码中的tokens[KEY _ TOKEN].length就是key的长度，而KEY _ MAX _ LENGTH被定义为250，即key的长度不能大于250字节。因此，为了节省内存和带宽，我们应该尽量使用较短的key。

4.Memcache: replace

replace方法用于替换一个已存在key的缓存内容，用法如下：

```
bool Memcache: replace ( string $key, mixed $ var[, int $ flag[, int $ expire]])
```

replace方法跟add方法的参数相同，这里省略参数说明。

5.Memcache: set

set方法用于设置一个指定key的缓存内容，set方法是add方法与replace方法的集合体（要设置的key不存在时，set方法与add方法的效果一样；要设置的key存在时，set方法与re place方法的效果一样）。用法如下：

```
bool Memcache: set ( string $key, mixed $ var[, int $ flag[, int $ expire]])
```

set方法跟add方法的参数相同，这里省略参数说明。

6.Memcache: get

get方法用于获取某key的缓存内容。用法如下:

```
string Memcache: get ( string $key[, int& $flags])  
array Memcache: get ( array $keys[, array& $flags])
```

key: 要获取值的key或key数组。

flags: 如果给定这个参数 (以引用方式传递), 该参数会被写入一些与key对应的信息。这些标记和set方法中的同名参数意义相同。

7.Memcache: delete

delete方法用于删除某key的缓存, 用法如下:

```
bool Memcache: delete ( string $key[, int $timeout])
```

key: 要删除元素的key。

timeout: 删除该元素的执行时间。如果timeout设置为0, 则表示该元素立即删除; 如果timeout设置为30, 该元素会在30秒内被删除。

8.Memcache: flush

flush方法的作用是立即使所有已经存在的缓存失效。flush方法不会真正释放任何资源, 仅仅标记所有缓存失效, 用法如下:

```
bool Memcache: flush ( void )
```

9.Memcache: getServerStatus

getServerStatus方法用于获取一个服务器的在线/离线状态（0表示服务器离线，非0表示在线），用法如下：

```
int Memcache: getServerStatus ( string $host[, int $port] )
```

host: 服务器域名或IP。

port: 服务器侦听的端口，默认为11211。

10.Memcache: getStats

getStats方法用于获取服务器的统计信息。getStats方法返回一个关联数组形式的服务器统计信息。数组的键是统计信息名，值就是统计信息的值，用法如下：

```
array Memcache: getStats ( [string $type[, int $slabid[, int $limit=100]] ] )
```

type: 期望获取的统计信息类型，可以使用的值有reset、malloc、maps、cachedump、slabs、items、sizes。

slabid: 与type参数联合从指定slab分块拷贝数据。

limit: 与type参数联合设置cachedump时从服务器端获取的实体条数。

11.Memcache: close

close方法用于关闭与Memcached服务器的连接，用法如下：

```
bool Memcache: close ( void )
```

9.2.4 使用Memcached加速Web应用

在实际应用中，通常会把数据库查询的结果保存到Memcached中，下次访问时直接从Memcached中获取，而不再进行数据库查询操作，这样在很大程度上能够减轻数据库的负担。

前面介绍了memcache扩展的使用方法，下面通过一个实例介绍使用Memcached缓存数据库的查询结果，从而达到加速的效果。代码如下：

```
<? php
$mc=new Memcache (); //创建Memcache对象
$mc->connect ( ' 127.0.0.1 ', 11211);
$uid= (int)$_GET[ ' uid ' ];
$sql="SELECT*FROM users WHERE uid= ' $uid ' ";
$key=md5 ( $sql);
//数据库查询结果是否已经缓存到Memcached服务器中
if ( ! ( $datas= $mc->get ( $key))) {
//在Memcached中未获取缓存数据，则使用数据库查询获取记录集
$conn=mysql_connect ( ' localhost ', ' test ', ' test ');
mysql_select_db ( ' test ');
$result=mysql_query ( $sql);
while ( $row=mysql_fetch_object ( $result)) {
    $datas[]= $row;
}
//将从数据库中获取的结果集数据保存到Memcached中，以供下次访问时使用
$mc->add ( $key, $datas );
}
var_dump ( $datas );
? >
```

首先通过md5 () 函数把要查询的SQL语句加密成一个唯一的key，使用这个key去Memcached服务器中进行查询。如果Memcached已经缓存此SQL查询的结果，则直接返回给用户。否则，从数据库中查询结果并缓存到Memcached服务器中。

9.3 深入了解Memcached

前面已经介绍过Memcached的使用，现在对Memcached应该有一定的认识了。接下来将深入探索Memcached的内部实现，以便更深入地了解Memcached。

9.3.1 Memcached如何支持高并发

Memcached使用多路复用I/O模型（如epoll、select等）。传统阻塞I/O中，系统可能会因为某个用户连接还没做好I/O准备而一直等待，直到这个连接做好I/O准备。如果这时有其他用户连接到服务器，很可能会因为系统阻塞而得不到响应。

而多路复用I/O是一种消息通知模式，用户连接做好I/O准备后，系统会通知我们这个连接可以进行I/O操作，这样就不会阻塞在某个用户连接。因此，Memcached才能支持高并发。

此外，Memcached使用了多线程模式。在开启Memcached服务器时通过使用“t”参数指定要开启的线程数。但并不是线程数越多越好，一般设置为CPU核数，这样效率最高。因为线程数越多，系统需要的线程调度时间就越多。而把线程数设置成CPU核数，系统需要的线程调度时间最少。

9.3.2 使用Slab分配算法保存数据

Memcached默认只能存储不大于1MB的数据（修改Memcached源码可以打破这个限制，只需把POWER_BLOCK宏设置为更大的数）。为什么Memcached会有这个限制呢？因为Memcached在存储数据时使用Slab内存分配算法。使用这种算法可以减少生成内存碎片，提高内存使用效率等。但使用这种算法也导致Memcached只能使用不大于1MB的内存。

下面简单介绍Memcached的Slab分配算法。该算法所有源代码都在slabs.c文件中。

Slab分配算法的原理是，把固定大小（1MB）的内存块划分成n小块，如图9-3所示。

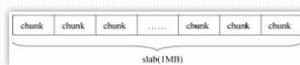


图 9-3 slab页内存结构

Slab分配算法把每1MB大小的内存块称为一个slab页，每次向系统申请一个slab页，然后再通过分割算法把这个slab页分割成若干小块的chunk，然后把这些chunk分配给用户使用，如图9-2所示。分割算法如下：

```
#define POWER_SMALLEST 1
#define POWER_LARGEST 200
#define POWER_BLOCK 1048576 /*slab页的大小*/
slabclass_t slabclass[POWER_LARGEST+1];
void slabs_init ( const size_t limit, const double factor, const bool prealloc ) {
    unsigned int size=sizeof ( item ) +48;
    int i=POWER_SMALLEST-1;
    double factor=1.25;
    while ( ++i<POWER_LARGEST && size <=POWER_BLOCK/2 ) {
        if ( size%CHUNK_ALIGN_BYTES )
            size+=CHUNK_ALIGN_BYTES- ( size%CHUNK_ALIGN_BYTES );
        slabclass[i].size=size;
        slabclass[i].perslab=POWER_BLOCK/slabclass[i].size;
        size*=factor; /*每次增大factor倍*/
    }
}
```

```
}
power_largest=i;
slabclass[power_largest].size=POWER_BLOCK;
slabclass[power_largest].perslab=1;
}
```

slabclass是一个类型为slabclass_t结构的数组，slabclass_t结构体定义如下：

```
typedef struct {
unsigned int size;
unsigned int perslab;
void**slots;
unsigned int sl_total;
unsigned int sl_curr;
void*end_page_ptr;
unsigned int end_page_free;
unsigned int slabs;
void**slab_list;
unsigned int list_size;
unsigned int killing;
} slabclass_t;
```

size字段保存一个chunk块的大小；perslab字段保存一个slab页中有多少个chunk块；slots字段保存被回收的chunk块列表；sl_total字段保存slots列表中chunk块的个数；sl_curr字段记录着slots列表中下一个可以使用的chunk块；end_page_ptr字段保存着下一个可以使用的空闲chunk块；slab_list字段保存着所有从操作系统申请的slab页列表。

由分割算法的源代码可以知道，Slab算法按照不同大小的chunk块分割slab页，而不同大小的chunk块以factor（默认为1.25）倍增大。perslab的计算方法就是使用slab页的大小（1MB）除以chunk块的大小。代码如下：

```
slabclass[i].size=size;
slabclass[i].perslab=POWER_BLOCK/slabclass[i].size;
size*=factor;
```

其中POWER_BLOCK就是slab页的大小，默认为1048576字节（1MB）。使用“memcached vv”命令查看内存分配情况，如图9-4所示。

```
hexusong@hexusong-desktop: ~$ memcached -vv
slab class 1: chunk size 80 perslab 13197
slab class 2: chunk size 104 perslab 10082
slab class 3: chunk size 136 perslab 7710
slab class 4: chunk size 176 perslab 5957
slab class 5: chunk size 224 perslab 4682
slab class 6: chunk size 280 perslab 3744
slab class 7: chunk size 352 perslab 2970
slab class 8: chunk size 440 perslab 2383
slab class 9: chunk size 560 perslab 1890
slab class 10: chunk size 720 perslab 1456
slab class 11: chunk size 928 perslab 1125
slab class 12: chunk size 1184 perslab 885
slab class 13: chunk size 1536 perslab 687
slab class 14: chunk size 1984 perslab 529
slab class 15: chunk size 2592 perslab 413
slab class 16: chunk size 3328 perslab 320
slab class 17: chunk size 4352 perslab 245
slab class 18: chunk size 5728 perslab 188
slab class 19: chunk size 7424 perslab 145
slab class 20: chunk size 9664 perslab 111
slab class 21: chunk size 12608 perslab 83
slab class 22: chunk size 16512 perslab 63
slab class 23: chunk size 21632 perslab 48
slab class 24: chunk size 28416 perslab 36
slab class 25: chunk size 37248 perslab 27
slab class 26: chunk size 48832 perslab 20
slab class 27: chunk size 64256 perslab 15
slab class 28: chunk size 84384 perslab 11
slab class 29: chunk size 110528 perslab 8
slab class 30: chunk size 144704 perslab 6
slab class 31: chunk size 191680 perslab 4
slab class 32: chunk size 252224 perslab 3
slab class 33: chunk size 330688 perslab 2
slab class 34: chunk size 432912 perslab 1
slab class 40: chunk size 1048576 perslab 1
```

图 9-4 Slab算法内存分配情况

从图9-4中可以看出，默认情况下Memcached可分为40种slab页（slab class 1 ~ slab class 40），每种slab页的chunk块大小都不相同。如slab class 1的chunk块大小为80字节，slab class 2的chunk块大小为104字节，但由公式

$$\text{next_slabclass_size} = \text{current_slabclass_size} * 1.25$$

可得slab class 2的chunk块大小应该为 $80 * 1.25 = 100$ 。为什么slab class 2的chunk块大小是104呢？原因是字节对齐，Memcached规定8字节对齐，也就是说chunk块的大小必须被8整除。字节对齐操作如下：

```
if ( size%CHUNK_ALIGN_BYTES )
size+=CHUNK_ALIGN_BYTES- ( size%CHUNK_ALIGN_BYTES );
```

CHUNK_ALIGN_BYTES等于8，上面的操作变为：

```
size+=8- ( size%8);
```

所以100字节经过字节对齐处理之后变成104。

通过以上分析，可以使用图9-5表示slabclass数组。

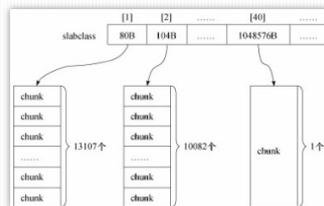


图 9-5 slabclass数组结构

Memcached向Slab层申请内存存储数据时，Slab层从slabclass中找到一个合适的slab页，然后分配其中一个空闲的chunk块给Memcached使用。此过程通过slabs_clsidx()和do_slabs_alloc()函数完成，代码如下：

```
unsigned int slabs_clsidx ( const size_t size ) {
    int res=1;
    if ( size==0 )
        return 0;
    while ( size > slabclass[res].size )
        if ( res++==power_largest ) return 0;
    return res;
}

void*do_slabs_alloc ( const size_t size, unsigned int id ) {
    slabclass_t*p;
    void*ret=NULL;
    .....

    p=&slabclass[id]; /*大小合适的slabclass*/
    if ( ! ( p->end_page_ptr! =0 | | p->sl_curr! =0 | |
do_slabs_newslab ( id )! =0 ) ) { //向操作系统申请新的slab页
```

```
ret=NULL;
} else if (p->sl_curr! =0) {
ret=p->slots[——p->sl_curr]; //被回收的chunk块
} else {
ret=p->end_page_ptr; //空闲的chunk块
if (——p->end_page_free! =0) {
p->end_page_ptr+=p->size;
} else {
p->end_page_ptr=0;
}
}
return ret;
}
```

slab_clsid () 函数的作用是根据参数size找到一个chunk块大小刚好合适的slabclass。找到合适的slabclass后，便会把其中一个可用的chunk块分配给Memcached使用，分配工作由do_slabs_alloc () 函数完成。

do_slabs_alloc () 函数首先尝试从slots列表（被回收的chunk块列表）中获取可用的chunk块，如果有可用的chunk块便返回，否则就从空闲的chunk块列表中获取可用的chunk块并返回。

因为最大的chunk块大小为1MB，所以Memcached只能存储不大于1MB的数据。

9.3.3 删除过期item

Memcached为每个item设置一个过期时间，但不是到期就把item从内存中删除，而是访问item时如果到了有效期，才把item从内存中删除。实现代码如下（版本1.2.8）：

```

item*do_item_get_notdeleted ( const char*key, const size_t nkey, bool*delete_locked )
{
    item*it=assoc_find ( key, nkey );
    if ( delete_locked ) *delete_locked=false;
    if ( it!=NULL&&( it->it_flags&ITEM_DELETED ) ) {
        if ( ! item_delete_lock_over ( it ) ) {
            if ( delete_locked ) *delete_locked=true;
            it=NULL;
        }
    }
    if ( it!=NULL && settings.oldest_live!=0 && settings.oldest_live<=current_time && it->
time<=set
tings.oldest_live ) {
        do_item_unlink ( it );
        it=NULL;
    }
    if ( it!=NULL&&it->exptime!=0&&it->exptime<=current_time ) {
        do_item_unlink ( it );
        it=NULL;
    }
    if ( it!=NULL ) {
        it->refcount++;
        DEBUG_REFCNT ( it, ' + ' );
    }
    return it;
}

```

使用do_item_get_notdeleted函数在Memcached中查找指定的item，加粗代码部分功能为删除过期item。从上面的代码可知，当item过期时间早于当前时间时，

便会删除此item。

提示 延迟删除过期item到查找时进行，可以提高Memcached效率。这样不必时时刻刻检查过期item，从而提高CPU工作效率。

9.3.4 使用LRU算法淘汰数据

当Memcached使用内存数大于设置的最大内存使用数时，为了腾出内存空间来存放新的数据项，Memcached会启动LRU算法淘汰旧的数据项。那么，Memcached会淘汰哪些数据项呢？下面是Memcached淘汰数据的源码（版本1.2.8）：

```
it=slabs_alloc ( ntotal, id );
if ( it==0 ) {
int tries=50;
.....

for ( search=tails[id]; tries>0&&search!=NULL; tries--, search=search->prev )
{
//查找一个引用计数器为0的item
if ( search->refcount==0 ) {
.....

do_item_unlink ( search );
break;
}
}
it=slabs_alloc ( ntotal, id );
if ( it==0 ) {
tries=50;
for ( search=tails[id]; tries>0&&search!=NULL; tries--, search=search->prev )
{
//查找一个3小时没有被访问的item
if ( search->refcount!=0&&search->time+10800<current_time ) {
search->refcount=0;
do_item_unlink ( search );
break;
}
}
it=slabs_alloc ( ntotal, id );
if ( it==0 ) {
return NULL;
}
}
}
```

从上述代码中可以看到，使用slabs _ alloc函数申请内存失败时，就开始淘汰数据了。淘汰规则是，从数据项列表（item list）尾部开始遍历，在列表中查找一个引用计数器（refcount）为0的item，把此item释放掉。

为什么要从item列表尾部开始遍历呢？因为Memcached会把刚刚访问过的item放到item列表头部，所以尾部的item都是没有被访问过的（或者很少被访问），这就是LRU算法的精髓。

如果在item列表中找不到引用计数器为0的item，就查找一个3小时没有访问过的item，把它释放。如果还是找不到，就返回NULL（申请内存失败）。

从上面的分析可以知道，当内存不足时，Memcached会把访问比较少或者一段时间没有访问的item淘汰，以便腾出内存空间存放新的item。

9.3.5 Memcached多线程模型

Memcached是一个多线程的缓存服务器程序。在Memcached内部，线程分为：

主线程（main thread）：接收客户端连接，并把连接分配给工作线程处理。

工作线程（worker thread）：处理客户端连接请求。

Memcached多线程模型原理如图9-6所示。

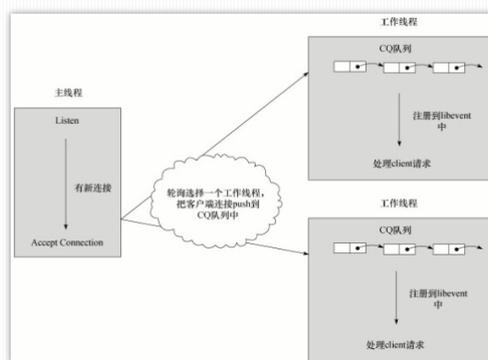


图 9-6 Memcached多线程模型原理

从图9-6中可以看出，主线程主要侦听客户端连接，有客户端连接到Memcached时，Memcached会调用accept函数接收到来的连接，把连接push到工作线程的CQ队列中，并向工作线程发送一个信号，通知工作线程有新的客户端连接需要处理。

当工作线程接收到主线程的信号后，便会把CQ队列上的客户端连接注册到libevent进行侦听，libevent会侦听客户端连接的读写事件，并调用相关的回调函数（drive_machine）进行处理。

主线程接收到客户端连接时的处理代码如下：

```
static void drive_machine ( conn*c ) {
.....

while (! stop) {
switch ( c->state ) {
```

```
case conn_listening:
    addrlen=sizeof ( addr );
    if ( ( sfd=accept ( c->sfd, ( struct sockaddr* ) &addr, &addrlen ) ) ==-1 ) {
    if ( errno==EAGAIN | | errno==EWOULDBLOCK ) {
        stop=true;
    } else if ( errno==EMFILE ) {
        accept_new_conns ( false );
        stop=true;
    } else {
        perror ( "accept ( )" );
        stop=true;
    }
    break;
    }
    if ( ( flags=fcntl ( sfd, F_GETFL, 0 ) ) <0 | |
    fcntl ( sfd, F_SETFL, flags | O_NONBLOCK ) <0 ) {
        perror ( "setting O_NONBLOCK" );
        close ( sfd );
        break;
    }
    dispatch_conn_new ( sfd, conn_read, EV_READ | EV_PERSIST,
    DATA_BUFFER_SIZE, false );
    break;
    .....
    }}
    .....
}
```

从上面的代码可以知道，当有客户端连接到Memcached时，主线程会调用accept函数接收客户端连接，然后调用dispatch_conn_new函数把连接交给工作线程处理。

dispatch_conn_new函数的主要工作是选择一个工作线程，把客户端连接push到此工作线程的CQ队列中，接着发送一个信号通知工作线程有新的客户端连接需要处理，dispatch_conn_new函数的代码如下：

```
void dispatch_conn_new ( int sfd, int init_state, int event_flags, int read_buffer_size, int is
```

```
_udp) {
    CQ_ITEM*item=cqi_new ();
    int tid=last_thread% ( settings.num_threads-1);
    tid++;
    LIBEVENT_THREAD*thread=threads+tid;
    last_thread=tid;
    item->sfd=sfd;
    item->init_state=init_state;
    item->event_flags=event_flags;
    item->read_buffer_size=read_buffer_size;
    item->is_udp=is_udp;
    cq_push (&thread->new_conn_queue, item);
    MEMCACHED_CONN_DISPATCH ( sfd, thread->thread_id);
    if ( write ( thread->notify_send_fd, "", 1) !=1) {
        perror ("Writing to thread notify pipe");
    }
}
```

当工作线程接收到主线程发送过来的信号时，便会调用thread_libevent_process函数进行处理，代码如下：

```
static void thread_libevent_process ( int fd, short which, void*arg) {
    LIBEVENT_THREAD*me=arg;
    CQ_ITEM*item;
    char buf[1];
    if ( read ( fd, buf, 1) !=1)
        if ( settings.verbose>0)
            fprintf ( stderr, "Can ' t read from libevent pipe \ n");
    item=cq_peek (&me->new_conn_queue);
    if ( NULL! =item) {
        conn*c=conn_new ( item->sfd, item->init_state, item->event_flags,
            item->read_buffer_size, item->is_udp, me->base);
        if ( c==NULL) {
            if ( item->is_udp) {
                fprintf ( stderr, "Can ' t listen for events on UDP socket \ n");
                exit ( 1);
            }
        }
    }
}
```

```
    } else {  
    if ( settings.verbose>0 ) {  
    fprintf ( stderr, "Can ' t listen for events on fd%d \ n",  
    item->sfd );  
    }  
    close ( item->sfd );  
    }  
    }  
    cqi_free ( item );  
    }  
    }
```

thread_libevent_process函数的主要工作是从管道读取一个字节的数据，然后从CQ队列中取得一个连接，并调用conn_new函数把此连接注册到libevent中进行侦听。当客户端连接可读或者可写时，Memcached便会调用drive_machine函数进行处理。

9.4 Memcached分布式布置方案

有时1台Memcached服务器不能满足我们的要求，需要布置多台Memcached服务器。但是有个问题，怎么确定一个数据应该保存到哪台服务器上面呢？

有两种方案，第一种是普通Hash分布，第二种是一致性Hash分布。下面通过PHP作为客户端来说明这两种方案。

9.4.1 普通Hash分布

普通Hash分布比较简单，Hash函数大概如下：

```
function mHash ($key) {
    $md5=substr ( md5 ($key), 0, 8);
    $seed=31;
    $hash=0;
    for ($i=0; $i<8; $i++) {
        $hash=$hash*$seed+ord ($md5 {$i});
        $i++;
    }
    return $hash&0x7FFFFFFF;
}
```

首先通过md5把key处理成一个32位字符串，取其前8字符。再经过Hash算法处理成一个整数并返回，然后映射到其中一台Memcached服务器。

假设配置两台Memcached服务器，可以使用下面的代码映射：

```
<? php
$servers=array (
    array ( "host"=>"192.168.1.12", "port"=>6397),
    array ( "host"=>"192.168.1.20", "port"=>6397),
```

```
);  
$key="TheKey";  
$value="TheValue";  
$sc=$servers[mHash($key)%2];  
$memcached=new Memcached($sc);  
$memcached->set($key,$value);  
? >
```

通过Hash函数把key转化成整数后，利用这个整数与Memcached服务器数量取模，如下面代码所示：

```
$sc=$servers[mHash($key)%2];
```

这样得到的是其中一台服务器的配置，利用这个配置连接Memcached服务器。这样就完成了分布式布置。取数据跟保存数据的方法一样，把set命令改为get命令就可以了。

9.4.2 一致性Hash分布

在服务器数量不发生改变的时候，普通Hash分布可以很好地运作。当服务器数量发生改变时，问题就出来了。试想，增加一台服务器时，同一个key经过Hash之后，与服务器数取模的结果跟没增加服务器之前的结果会不一样，这就导致之前保存的数据丢失。为了把丢失的数据减到最少，可以采取一致性Hash（Consistent Hashing）分布算法解决。

一致性Hash分布算法分6个步骤，如下所示。

步骤1 将一个32位整数（即 $0 \sim 2^{32}-1$ ）想象成一个环，如图9-7所示。

将0作为圆环的头， $2^{32}-1$ 作为圆环的尾，把它连接起来，就成为环。当然这只是想象。

步骤2 通过Hash函数把key处理成整数。

例如把4个key（key1~key4）通过Hash函数处理成整数：

```
$ key1=mHash ("key1");  
$ key2=mHash ("key2");  
$ key3=mHash ("key3");  
$ key4=mHash ("key4");
```

把key处理成整数之后，就可以在环中找到一个位置与之对应，如图9-8所示。

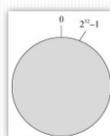


图 9-7 将32位整数想象成一个环

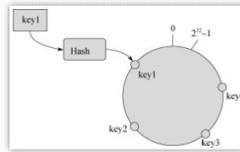


图 9-8 4个key处理成整数在环中的位置

步骤3 把Memcached群映射到环上，使用Hash函数处理服务器所使用的IP地址。

例如有3台服务器，分别使用IP（192.168.1.1）、IP（192.168.1.2）、IP（192.168.1.3），使用下面的方法映射到环上

```
$ server1=mHash ("192.168.1.1");
$ server2=mHash ("192.168.1.2");
$ server3=mHash ("192.168.1.3");
```

把服务器映射到环上，如图9-9所示。

经过上面几个步骤，我们把数据的key和服务器都映射到同一个环上。下面考虑如何把数据映射到服务器上。

步骤4 把数据映射到服务器上。

沿着圆环顺时针方向的key出发（key4），直到遇到一个服务器为止（server2），把key对应的数据保存到这个服务器上。根据上面的方法，key4和key3保存到server2上，key2保存到server1上，key1保存到server3上，如图9-10所示。

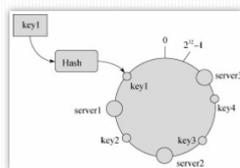


图 9-9 把服务器映射到环上

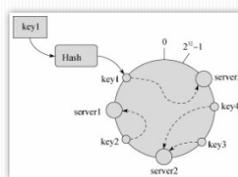


图 9-10 把数据映射到服务器上

步骤5 移除服务器。

考虑一下，如果现在server2服务器崩溃了，那么受到影响的仅是那些沿着server2逆时针出发直到遇到下一个服务器（server3）之间的数据，也就是映射到server2上的那些数据。

在上例中，需要进行变动的有key3和key4对应的数据，把这些数据重新映射到server1上即可，如图9-11所示。

步骤6 添加服务器。

再考虑一下，如果现在需要增加1台服务器server4，用之前的方法把它映射到key3与key4之间。这时受到影响的是沿着server4逆时针出发直到遇到下一个服务器（server3）之间的数据（它们原本是映射到server2上的一部分数据），把这些数据重新映射到server4上即可。

在这里仅需要变动的只有key4对应的数据，将其重新映射到server4上即可，如图9-12所示。

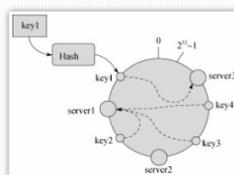


图 9-11 移除服务器

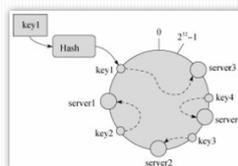


图 9-12 添加服务器



9.4.3 一致性Hash分布算法实例

下面使用PHP实现一致性Hash分布算法。

创建一个FlexiHash类，此类是实现一致性Hash分布算法的实体，它有两个成员变量：

serverList：保存服务器列表。

isSorted：记录服务器列表是否已经排过序。

此外，FlexiHash类有三个方法：

addServer：添加一个服务器到服务器列表中；

removeServer：从服务器列表中删除一个服务器；

lookup：在当前的服务器列表中找到合适的服务器存放数据。

使用PHP实现一致性Hash分布算法的代码如下：

```
<? php
class FlexiHash
{
private $ serverList=array ();
private $ isSorted=FALSE;
function addServer ($ server) {……}
function removeServer ($ server) {……}
function lookup ($ key) {……}
}? >
```

1) addServer方法实现：

```
function addServer ($ server) {
```

```
$hash=mHash($server);  
if(!isset($this->serverList[$hash])){  
    $this->serverList[$hash]=$server;  
}  
$this->isSorted=FALSE;  
return TRUE;  
}
```

addServer方法实现很简单。首先通过mHash函数计算出服务器的Hash值，通过此Hash值定位服务器列表上的某个位置。因为此时服务器列表发生了变化，所以应该把排序标识\$isSorted设置为FALSE。

2) removeServer方法实现:

```
function removeServer($server){  
    $hash=mHash($server);  
    if(isset($this->serverList[$hash])){  
        unset($this->serverList[$hash]);  
    }  
    $this->isSorted=FALSE;  
    return TRUE;  
}
```

removeServer方法实现跟addServer方法实现差不多。首先通过mHash函数计算出服务器的Hash值，以此Hash值作为要删除服务器的索引，把此服务器从服务器列表中删除。因为此时服务器列表发生了变化，所以把排序标识isSorted设置为FALSE。

3) lookup方法实现:

```
function lookup($key){  
    $hash=mHash($key);  
    if(!$this->isSorted){  
        krsort($this->serverList, SORT_NUMERIC);  
    }  
}
```

```
$this->isSorted=TRUE;
}
foreach ($this->serverList as $pos=>$server) {
    if ($hash>=$pos) return $server;
}
return $this->serverList[count($this->serverList)-1];
}
```

lookup方法首先通过mHash函数计算出key的Hash值，然后判断服务器列表是否排过序，如果没有，就先对服务器列表进行倒序排序操作。倒序排序的作用是把服务器列表转化成一个逆时针的圆环。最后遍历服务器列表，找到一个合适的服务器返回。

4) 测试代码:

```
<? php
$hserver=new FlexiHash ();
$hserver->addServer ("192.168.1.1");
$hserver->addServer ("192.168.1.2");
$hserver->addServer ("192.168.1.3");
$hserver->addServer ("192.168.1.4");
$hserver->addServer ("192.168.1.5");
echo"save key1 in server: ",$hserver->lookup (' key1 ');
echo"save key2 in server: ",$hserver->lookup (' key2 ');
echo ' ===== ';
$hserver->removeServer ("192.168.1.4");
echo"save key1 in server: ",$hserver->lookup (' key1 ');
echo"save key2 in server: ",$hserver->lookup (' key2 ');
echo ' ===== ';
$hserver->addServer ( ' 192.168.1.6 ');
echo"save key1 in server: ",$hserver->lookup (' key1 ');
echo"save key2 in server: ",$hserver->lookup (' key2 ');
? >
```

测试结果如图9-13所示。

```
save key1 in server: 192.168.1.4  
save key2 in server: 192.168.1.2  
=====  
save key1 in server: 192.168.1.3  
save key2 in server: 192.168.1.2  
=====  
save key1 in server: 192.168.1.3  
save key2 in server: 192.168.1.2
```

图 9-13 测试结果

从上面的测试结果可以看出，在增加或者减少服务器的时候，一致性Hash算法只会改变很少一部分数据的存储服务器，从而减少了数据丢失的情况。

9.5 本章小结

本章主要介绍Memcached的使用和原理。大多数Web应用的瓶颈在于数据库访问，而使用Memcached可以减少对数据的访问，从而提高Web应用效率。另外，了解Memcached内部实现的原理，可以帮助我们解决一些使用Memcached时遇到的问题。

第10章 Redis使用与实践

本章介绍功能比Memcached更强大的数据库：Redis。本质上，Redis是Key Value类型的内存数据库，其中Value可以是String、List、Set、Sorted Set、Hash等类型的数据结构。目前，国内新浪微博大量使用Redis存储数据，很多知名网站也纷纷加入Redis阵营。

10.1 Redis的安装及使用

Redis把整个数据库全加载到内存当中进行操作，通过异步操作定期把数据库数据flush到硬盘上保存。因为是纯内存操作，所以Redis的性能非常出色，每秒处理10万次以上的读写操作，是已知性能最快的Key Value数据库。

Redis有以下优点：

支持丰富的数据类型：如String、List、Set、Sorted Set、Hash等。

支持两种数据持久化方式：Snapshotting（快照）和Append Only file（追加）。

支持主从复制。

10.1.1 Redis安装步骤

1) 下载Redis的稳定版本，地址如下：

```
$ wget http://redis.googlecode.com/files/redis-2.2.10.tar.gz
```

2) 解压缩，方法如下：

```
$ tar zvxf redis-2.2.10.tar.gz
```

3) 编译Redis, 方法如下:

```
$ cd redis-2.2.10 $ make
```

make命令执行完成后, 在当前目录下生成几个可执行文件, 分别是redis server、redis cli、redis benchmark、redis check dump和redis check aof, 使用“ls./src F | grep ' * ’”命令把它们列出来, 如图10-1所示。



```
~/src/redis-2.2.10 $ ls ./src | grep '* *'
redis-benchmark
redis-check-aof
redis-check-dump
redis-cli
redis-server
```

图 10-1 Redis可执行文件列表

其中比较重要的有以下3个, 具体作用如下:

redis server: Redis服务器程序。

redis cli: Redis命令行操作工具。也可以用telnet根据其纯文本协议操作。

redis benchmark: Redis性能测试工具, 测试Redis在你的系统及配置下的读写性能。

4) 安装Redis。

Redis没有提供make install命令, 需要手动安装, 安装过程如下:

```
$ sudo cp redis.conf/etc/
```

```
$ sudo cp redis-benchmark redis-cli redis-server/usr/bin/
```

10.1.2 修改Redis配置文件

接下来修改/etc/redis.conf文件调整Redis配置。表10-1列举了Redis中一些常用的配置项。

配置项	描述
daemonize < yes/no >	是否以后台 daemon 方式运行
pidfile < * .pid >	pid 文件位置, daemonize 为 yes 时才起作用
port < port >	监听的端口号 (默认 6379)
timeout < seconds >	请求超时时间 (单位: 秒)
loglevel < debug/verbose/notice/warning >	log 信息级别, 选项有 debug, verbose, notice, warning
logfile < filename >	log 文件位置 (默认 stdout)
databases < number >	开启数据库的数量
save < seconds > < changes >	保存快照的频率, 在一定时间内执行一定数量的写操作时, 自动保存快照, 可设置多个条件
slaveof < masterip > < masterport >	当本机为从服务时, 设置主服务的 IP 及端口
masterauth < master-password >	当本机为从服务时, 设置主服务的连接密码
requirepass	连接密码
maxclients < number >	最大客户端连接数, 0 为 unlimited
maxmemory < bytes >	设置最大内存, 达到最大内存设置后, Redis 先尝试清除已过期或即将过期的 key, 当此方法处理后, 仍到达最大内存设置, 将无法再进行写入操作
rdbcompression < yes/no >	是否使用压缩
rdbfilename < * .rdb >	数据快照文件名 (只是文件名, 不包括目录)
dir < directory >	数据快照的保存目录
appendonly < yes/no >	是否开启 appendonly log, 如果开启每次写操作记一条 log, 会提高数据抗风险能力, 但影响效率
appendfilename < * .aof >	更新日志文件名
appendfsync < always/everysec/no >	appendonly log 如何同步到磁盘 (always: 每次写都强制调用 fsync; everysec: 每秒调用一次 fsync; no: 不调用 fsync 等待系统自己同步)

配置项	描述
vm-enabled < yes/no >	是否启用虚拟内存
vm-swapfile < * .swap >	交换分区文件 (启用虚拟内存时使用)
vm-max-memory < number >	将所有大于 vm-max-memory 的数据存入虚拟内存, 无论 vm-max-memory 设置多少, 所有索引数据都是内存存储的 (Redis 的索引数据就是 key), 也就是说, 当 vm-max-memory 设置为 0 时, 其实是所有 value 都存在于磁盘上
vm-page-size < bytes >	SWAP 文件页大小 (根据存储的值设置, 单位: byte)
vm-pages < number >	交换分区文件中内存页的数量
vm-max-threads < number >	对 SWAP 文件操作的最大线程数 (最好为 CPU 的数量)

例如, 希望以daemon方式运行Redis, 应该把daemonize配置项修改为yes; 希望请求不能超过5秒, 应该把timeout配置项修改为5。

10.1.3 运行Redis服务器

使用以下命令运行Redis服务器:

```
$/usr/bin/redis-server/etc/redis.conf
```

使用以下命令检查Redis是否启动:

```
$ ps-x | grep redis  
1411 pts/0 S+ 0: 00/usr/bin/redis-server/etc/redis.conf
```

10.1.4 key相关命令

Redis本质上是Key Value数据库，所以先了解key的相关操作。在Redis中，key使用字符串存储，但是key中不能出现空格或者换行符“\n”，原因是空格和换行符都是Redis的特殊字符，但只限于key，value可以使用任何字符。

注意 Redis以“\n”作为命令结束符，所以在key中不能存在“\n”，否则就会出错。此外，Redis以空格作为命令和参数的分隔符，所以在key中也不能存在空格。

key相关的命令如表10-2所示。

命令	描述
exists key	测试指定 key 是否存在，返回 1 表示存在，返回 0 代表不存在
del key1 key2...keyN	删除给定 key，返回删除 key 的数目，返回 0 表示给定 key 都不存在
type key	返回给定 key 的 value 类型，返回 none 表示不存在 key，String 为字符串类型，List 为链表类型，set 为无序集合类型...

命令	描述
keys pattern	返回匹配指定模式的所有 key
expire key seconds	设置给定 key 的过期时间
randomkey	返回从当前数据库中随机选择的一个 key，如果当前数据库是空的，返回空串
rename oldkey newkey	重命名 key，如果 newkey 存在，将被覆盖，返回 1 表示成功，返回 0 表示失败，若失败，则可能是 oldkey 不存在或者和 newkey 相同
renamex oldkey newkey	同上，如果 newkey 存在返回失败
ttl key	返回设置过期时间 key 的剩余秒数，-1 表示 key 不存在或者没有设置过期时间
move key db-index	将 key 从当前数据库移动到指定数据库，返回 1 成功，返回 0 表示 key 不存在或者已经在指定数据库中

下面使用Redis提供的命令行操作工具redis cli与Redis服务器进行交互，并且测试上面介绍的命令，使用下面的命令启动redis cli：

```

$ /usr/bin/redis-cli
启动redis-cli后，测试上述命令：
redis> set key1 value1
OK
redis> set key2 value2
OK
redis> set key3 value3
OK
redis> randomkey
"key2"
redis> keys key*
1."value2"

```

```
2."value3"  
3."value1"  
redis> type key1  
string  
redis> del key1 key2  
( integer ) 2  
redis> expire key3 600  
( integer ) 1  
redis> rename key1 key11  
OK
```

应该尽量使用较短的key，因为较短的key可以节省内存和带宽。

10.1.5 Redis支持的数据类型

Redis支持多种数据类型，如String、List、Set、Sorted Set、Hash等。每种数据类型都有其各自的特点，下面介绍Redis所支持的数据类型的特点和使用方法。

1.String类型

String类型是二进制安全的，可以把图片和视频文件保存到String中，定义如下：

```
struct sdshdr {  
    long len;  
    long free;  
    char buf[];  
};
```

buf数组：字符串的实体，保存字符串的内容。

len字段：记录buf数组大小。

free字段：记录buf数组还有多少可用空间。

String内存结构如图10-2所示。

因为有len和free字段记录字符串信息，所以不必使用一般的nil字符作为结束，从而实现二进制安全。为了提高网站的运行速度，可以使用String类型缓存一些静态文件，如图片文件、CSS文件等。

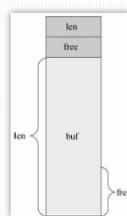


图 10-2 String内存结构

String类型支持的命令如表10-3所示。

命令	描述
set key value	设置 key 对应 String 类型的值。返回 1 表示成功，返回 0 表示失败
setnx key value	如果 key 不存在，设置 key 对应 String 类型的值。如果 key 已经存在，返回 0
get key	获取 key 对应的 String 值。如果 key 不存在返回 nil
getset key value	先获取 key 的值，再设置 key 的值。如果 key 不存在返回 nil
mget key1 key2...keyN	一次获取多个 key 的值。如果对应 key 不存在，则对应返回 nil
mset key1 value1...keyN valueN	一次设置多个 key 的值。成功返回 1，表示所有的值都设置；失败返回 0，表示没有任何值被设置
msetnx key1 value1...keyN valueN	一次设置多个 key 的值，但是不会覆盖已经存在的 key
incr key	向 key 对应的值加 1，并返回新的值。注意 incr 一个不是 int 的 value 会返回错误。incr 一个不存在的 key，则设置 key 值为 1
decr key	向 key 对应的值减 1，decr 一个不存在的 key，则设置 key 值为 -1
incrby key integer	向 key 对应的值加上一个指定整数 integer，key 不存在时会设置 key，并认为原来的 value 是 0
decrby key integer	向 key 对应的值减去一个指定整数 integer，decrby 完全是为了可读性，通过 incrby 一个负值实现同样效果，反之一样

下面使用redis cli测试String类型的操作命令：

```
redis> set mykey1 myvalue1 OK
redis> get mykey1"myvalue1"
redis> mset mykey2 myvalue2 mykey3 myvalue3
OK
redis> mget mykey2 mykey3
1."myvalue2"
2."myvalue3"
redis> set counter 1
OK
redis> incr counter
(integer) 2
redis> decr counter
(integer) 1
```

String类型支持incr操作，可以用做统计计算，如统计网站访问次数、博客访问次数等。

2.List类型

List数据类型指key对应的value是一个双向链表结构，所以List类型提供链表支持的所有操作。List类型在互联网应用中非常有用，例如存放微博中“我关注的列表”，或者论坛中所有回帖ID。

List类型支持的命令如表10-4所示。

命令	描述
lpush key string	向 key 对应 List 头部添加一个字符串元素。成功返回 1，失败返回 0
rpush key string	向 key 对应 List 尾部添加一个字符串元素。成功返回 1，失败返回 0
llen key	返回 key 对应 List 的长度。如果 key 不存在返回 0，如果 key 对应类型不是 List 返回错误
lrange key start end	返回指定区间内 (start-end) 的元素。下标从 0 开始，负值表示从列表尾部开始计算。-1 表示倒数第一个元素。key 不存在返回空列表
ltrim key start end	截取 List 指定区间内 (start-end) 元素。成功返回 1，key 不存在返回错误
lset key index value	设置 List 中指定下标的元素值。成功返回 1，key 或者下标不存在返回错误
lrem key count value	从 List 头部 (count 正数) 或尾部 (count 负数) 删除一定数量 (count 绝对值) 的匹配 value 的元素，返回删除的元素数量。count 为 0 时候删除全部
lpop key	从 List 头部删除并返回删除元素。如果 key 对应 List 不存在或者空返回 nil。如果 key 对应值不是 List 返回错误
rpop key	从 List 尾部删除并返回删除元素
blpop key1...keyN timeout	从左到右扫描 key1...keyN，返回对第一个非空 List 进行 lpop 操作并返回。如果所有 List 都是空或不存在，阻塞 timeout 秒。timeout 为 0 表示一直阻塞。阻塞时，如果有其他 client 对 key1...keyN 中任意一个 key 进行 push 操作，阻塞解除并返回。如果超时发生，则返回 nil
brpop key1...keyN timeout	功能与 blpop 相似。不同的是 blpop 从头部删除，而 brpop 从尾部删除

下面使用redis cli测试List类型的操作命令：

```
redis> lpush mylist1 value1
(integer) 1
redis> llen mylist1
(integer) 1
redis> lpop mylist1"value1"
redis> lpush mylist1 value1
(integer) 1
redis> lpush mylist1 value2
(integer) 2
redis> lpush mylist1 value3
(integer) 3
redis> lrange mylist1 0 2
1."value3"
2."value2"3."value1"
```

另外，使用List还可以实现消息队列功能，减轻数据库的压力。消息队列类似于现实生活中的排队，每次有消息到达时就把消息放进队列尾部，取出消息时就从队列头部取出。要用List实现消息队列，先用rpush命令把消息放进队列尾部，然后使用lpop命令把消息从队列头部中取出。

3.Set类型

Set数据类型是一种无序集合，在Redis内部通过HashTable实现，查找和删除元素的时间复杂度为 $O(1)$ 。Set数据类型的优点是快速查找元素是否存在，用于记录一些不能重复的数据。例如，在网站注册账号时用户名不能重复，使用Set记录注册用户，如果注册的用户名已经存在于Set中，则拒绝此用户注册，如图10-3所示。



图 10-3 判断用户名是否注册

Set类型支持的命令如表10-5所示。

命令	描述
sadd key member	添加一个 String 元素到 key 对应的 set 集合中。成功返回 1，如果元素在集合中返回 0，key 对应的 set 不存在返回错误
srem key member	从 key 对应 set 中移除给定元素，成功返回 1，如果 member 在集合中不存在或者 key 不存在返回 0，如果 key 对应的不是 set 类型的值返回错误
spop key	删除并返回 key 对应 set 中随机的一个元素，如果 set 是空或者 key 对应的 set 不存在返回 nil
srandmember key	同 spop，随机取 set 中的一个元素，但是不删除元素
smove srckey dstkey member	从 srckey 对应 set 中移除 member 并添加到 dstkey 对应 set 中，整个操作是原子的。成功返回 1，如果 member 在 srckey 中不存在返回 0，如果 key 对应的值不是 set 类型，返回错误

(续)	
命令	描述
scard key	返回 set 的元素个数，如果 set 是空或者 key 不存在返回 0
sismember key member	判断 member 是否在 set 中，存在返回 1，不存在或者 key 对应的 set 集合不存在返回 0
sinter key1 key2...keyN	返回所有给定 key 的交集
sinterstore dstkey key1...keyN	同 sinter，同时将交集存到 dstkey 对应的 set 集合中
sunion key1 key2...keyN	返回所有给定 key 的并集
sunionstore dstkey key1...keyN	同 sunion，同时将并集保存到 dstkey 对应的 set 集合中
sdiff key1 key2...keyN	返回所有给定 key 的差集
sdiffstore dstkey key1...keyN	同 sdiff，同时将差集保存到 dstkey 对应的 set 集合中
smembers key	返回 key 对应 set 的所有元素，结果是无序的

下面使用redis cli测试Set类型的操作命令：

```
redis> sadd myset1 member1
(integer) 1
redis> sadd myset1 member2
(integer) 1
redis> sadd myset1 member3
(integer) 1
redis> scard myset1
(integer) 3
redis> sismember myset1 member1
(integer) 1
redis> sismember myset1 member5
(integer) 0
redis> smembers myset1
```

- 1."member3"
- 2."member1"
- 3."member2"

Set类型通常用于记录做过某些事情。例如在某些投票系统中，每个用户一天只能投票一次，那么可以使用Set来记录某个用户的投票情况，只需要以日期作为Set的key，则将用户ID作为member即可。要查看某个用户今天是否投过票，只需以今天的日期作为key去查询此用户ID是否存在。

4.Sorted Set类型

Sorted Set类型与Set类型相似，都是String类型元素的集合，不同的是Sorted Set属于有序集合，而Sorted Set通过一个double类型的整数score进行排序。Sorted Set通过SkipList（跳跃表）和HashTable组合完成。SkipList负责排序功能，而HashTable负责保存数据。

因为Sorted Set是排序的Set，所以Set能做的事情Sorted Set也可以做。而且Sorted Set还可以完成一些Set不能做的事情，例如，使用Sorted Set构建一个具有优先级的队列，这也是List类型不能实现的。

Sorted Set类型支持的命令如表10-6所示。

命 令	描 述
<code>zadd key score member</code>	添加元素 member 到集合，元素在集合中存在则更新对应 score
<code>zrem key member</code>	删除指定元素，1 表示成功，如果元素不存在返回 0
<code>zincrby key incr member</code>	增加对应 member 的 score 值，并且重新排序，返回更新后的 score 值
<code>zrank key member</code>	返回指定元素在集合中的排名（下标），集合中元素按 score 从小到大排序的
<code>zrevrank key member</code>	同上，但是集合中元素按 score 从大到小排序
<code>zrange key start end</code>	从集合中指定区间的元素，返回结果按 score 顺序排列
<code>zrevrange key start end</code>	同上，返回结果按 score 逆序排列
<code>zrangebyscore key min max</code>	返回集合中 score 在给定区间的元素
<code>zcount key min max</code>	返回集合中 score 在给定区间的数量
<code>zcard key</code>	返回集合中元素个数
<code>zscore key element</code>	返回给定元素对应的 score
<code>zremrangebyrank key min max</code>	删除集合中排名在给定区间的元素
<code>zremrangebyscore key min max</code>	删除集合中 score 在给定区间的元素

使用redis cli测试Sorted Set类型的操作命令：

```
redis> zadd myzset 1 one
(integer) 1
redis> zadd myzset 2 two
```

```
(integer) 1
redis> zadd myzset 3 three
(integer) 1
redis> zrange myzset 0 2
1."one"
2."two"
3."three"
redis> zrem myzset one
(integer) 1
redis> zrange myzset 0 1
1."two"
2."three"
```

Sorted Set类型在Web应用中非常有用。例如排行榜应用中按“顶帖”次数排序，方法是：将排序的值设置成Sorted Set的score值，将具体数据设置成相应的value，用户每次按“顶帖”按钮时，只需执行zadd命令修改score值。

5.Hash类型

Hash类型是每个key对应一个HashTable，添加、删除和修改操作的时间复杂度都是 $O(1)$ 。Hash类型适合应用于存储对象，例如用户信息对象。把用户ID作为key，可以把用户信息保存到Hash类型中，Hash类型存储结构如图10-4所示。



图 10-4 Hash类型存储结构

新建一个Hash类型对象时，为了节省内存，Redis使用zipmap存储数据。这个zipmap并不是真正的HashTable，添加、删除和修改操作的时间复杂度都是 $O(n)$ ，但是相比普通Hash Table，zipmap节省不少内存。如果field或者value大小超出一定限制，Redis在内部自动将zip map替换成正常的HashTable存储。修改配置文件的hash max zipmap entries和hash max zipmap value选项设置这两个限制。

Hash类型支持的命令如表10-7所示。

命令	描述
hset key field value	设置 key 对应的 Hash 对象中指定域的值。如果 key 对应的 Hash 对象不存在，将创建此 Hash 对象。如果指定的域已经存在，其值将被重写
hget key field	返回与 field 域关联的值。如果该域不存在或者 key 对应的 Hash 对象不存在，返回值为 nil
hgetall key field1...fieldN	返回存储在 key 对应的 Hash 对象中各个指定域相关的值。对于在 Hash 对象中不存在的域，返回值为 nil
hmset key field1 value1...fieldN valueN	设置存储在 key 对应的 Hash 对象中指定域的值。该命令会复写 Hash 中已经存在的域。如果 key 对应的 Hash 对象不存在，将创建此 Hash 对象
hincrby key field integer	将存储在 key 对应的 Hash 对象中 field 域相关的值加上由 integer 指定的值。如果 key 对应的 Hash 对象不存在，则创建此 Hash 对象。如果 field 域不存在或者具有一个不能表示为整型的字符串值，在执行该操作前设置为 0
hexists key field	查看指定 field 域是否存在
hdel key field	删除指定的 field 域。返回 1，如果指定的域不存在或者 key 不存在，返回值为 0
hlen key	返回 key 对应的 Hash 对象中 field 数。如果 key 不存在，返回值为 0
hkeys key	返回 key 对应的 Hash 对象中所有 field 名称
hvals key	返回 key 对应的 Hash 对象中所有值
hgetall key	返回 key 对应的 Hash 对象中所有域和相关联的值。在返回值中，每个域名称后面跟着相关联的值

使用redis cli测试Hash类型的操作命令：

```
redis > hset myhash name liexusong
(integer) 1
redis > hset myhash sex man
(integer) 1
redis > hset myhash age 25
(integer) 1
redis > hget myhash name
"liexusong"
redis > hkeys myhash
1."name"
2."sex"
3."age"
redis > hvals myhash
1."liexusong"
2."man"
3."25"
redis > hexists myhash name
(integer) 1
```

10.1.6 Redis排序命令详解

Redis支持对List、Set、Sorted Set类型进行排序，sort命令完整格式如下：

```
SORT key[BY pattern][LIMIT start count][GET pattern][ASC | DESC][ALPHA][STORE dstkey]
```

下面详细说明sort命令各选项。

1.SORT key

这是最简单的情况，不设置任何选项就是对集合元素进行简单排序，并返回排序后的结果，例如：

```
redis> lpush mylist 2
(integer) 1
redis> lpush mylist 1
(integer) 2
redis> lpush mylist 3
(integer) 3
redis> sort mylist
1) "1"
2) "2"
3) "3"
```

2.[ASC | DESC][ALPHA]

sort命令默认排序方式从小到大（ASC），也可以按照逆序或者按字母顺序排序。逆序在sort命令后加上DESC选项，如果想按字母顺序排序，加上ALPHA选项。ALPHA和DESC可以同时使用，例如：

```
redis> lpush namelist Tom
(integer) 1
redis> lpush namelist Jack
(integer) 2
redis> lpush namelist Mick
(integer) 3
redis> lpush namelist Wade
(integer) 4
redis> sort namelist
1) "Wade"
2) "Mick"
3) "Jack"
4) "Tom"
redis> sort namelist alpha
1) "Jack"
2) "Mick"
3) "Tom"
4) "Wade"
redis> sort namelist desc alpha
1) "Wade"
2) "Tom"
3) "Mick"
4) "Jack"
```

3.[BY pattern]

sort命令可以按照集合元素自身的值排序，还可以按给定模式（pattern）将集合元素内容组合成新key，并按新key对应的内容排序，完成后返回排好序的集合元素。下面使用第一个例子中的mylist集合作为排序对象：

```
redis> set name1 102
OK
redis> set name2 103
OK
redis> set name3 101
OK
```

```
redis> sort mylist by name*  
1) "3"  
2) "1"  
3) "2"
```

模式“name*”代表使用mylist集合中元素的值填充“*”，按照name1、name2、name3这3个key对应的值排序，返回的是排序后mylist集合中的元素。

4.[LIMIT start count]

上面的例子返回结果都是排序后集合的全部元素，使用limit选项能够限定返回结果的数量，例如：

```
redis> sort mylist limit 0 2  
1) "1"2) "2"
```

start下标从0开始，上面例子中limit选项意思是从第1个元素开始，获取2个元素。

5.[STORE dstkey]

经常对集合按照固定模式排序，把排序结果缓存起来可以减少CPU的开销。使用STORE选项将排序结果保存到指定的key中，例如：

```
redis> sort mylist by name*store cachelist  
( integer ) 3  
redis> type cachelist  
list  
redis> lrange cachelist 0-1  
1) "3"  
2) "1"  
3) "2"
```

上面的例子把排序结果保存到cachelist中。使用type命令查看cachelist的类型是List。

10.2 事务处理

Redis目前对事务的支持比较简单，只能保证一个客户端连接发起事务中的命令可以连续执行，而中间不会插入其他客户端连接的命令。

10.2.1 事务处理原理

一般情况下，Redis接收到一个客户端连接发来的命令，立刻执行并返回结果。但是当连接发出multi命令时，此连接便进入一个事务上下文，Redis把此连接发来的命令存入一个队列中。当此连接发出exec命令，Redis便开始按顺序执行队列中的所有命令，并将所有命令执行的结果打包一起返回给客户端连接，然后此连接便结束事务上下文。例如：

```
redis> multi
OK
redis> set num 1
QUEUED
redis> incr num
QUEUED
redis> exec
1) OK
2) (integer) 2
```

从上面的例子看到set num 1和incr num命令发出以后并没有立刻执行，而是存放到队列中。调用exec命令，这两个命令才开始连续执行，最后返回这两个命令执行后的结果。

可调用discard命令取消一个事务，例如：

```
redis> multi
OK
```

```
redis> set count 100
QUEUED
redis> incr count
QUEUED
redis> discard
OK
redis> get count
(nil)
```

从上面的例子看到，set count 100和incr count命令都没有执行。discard命令的作用是清空事务的命令队列并退出事务上下文。

另外，Redis只能保证事务的每个命令能够连续执行，但是如果事务中有命令执行失败，Redis不进行回滚操作。

10.2.2 事务处理实现

客户端连接执行multi命令后便进入一个事务上下文，multi命令代码如下：

```
void multiCommand ( redisClient*c ) {
    if ( c->flags&REDIS_MULTI ) {
        addReplyError ( c, "MULTI calls can not be nested" );
        return;
    }
    c->flags |= REDIS_MULTI; //添加事务处理标志
    addReply ( c, shared.ok );
}
```

multi命令为客户端连接添加“事务处理”标志，接下来Redis便知道此连接正在进行事务处理，便把连接发送的命令存放到命令队列中，代码如下：

```
int processCommand ( redisClient*c ) {
    .....

    /*Exec the command*/
    if ( c->flags&REDIS_MULTI&&
        c->cmd->proc! =execCommand&&c->cmd->proc! =discardCommand&&
        c->cmd->proc! =multiCommand&&c->cmd->proc! =watchCommand )
    {
        queueMultiCommand ( c );
        addReply ( c, shared.queued );
    } else {
        if ( server.vm_enabled&&server.vm_max_threads>0&&
            blockClientOnSwappedKeys ( c ) ) return REDIS_ERR;
        call ( c );
    }
    .....
}
```

processCommand函数在客户端连接执行时被调用。processCommand函数判断连接是否有事务处理标志，如果有就表示此连接正在进行事务处理，便调用queueMultiCommand函数把命令存放到命令队列中。

exec命令的代码如下：

```
void execCommand ( redisClient*c ) {
    int j;
    robj**orig_argv;
    int orig_argc;
    struct redisCommand*orig_cmd;
    //如果没有调用multi命令，返回错误信息
    if ( ! ( c->flags&REDIS_MULTII ) ) {
        addReplyError ( c, "EXEC without MULTI" );
        return;
    }
    .....

    /*Exec all the queued commands*/
    unwatchAllKeys ( c ); /*Unwatch ASAP otherwise we ' ll waste CPU cycles*/
    orig_argv=c->argv;
    orig_argc=c->argc;
    orig_cmd=c->cmd;
    addReplyMultiBulkLen ( c, c->mstate.count );
    /*执行命令队列中的命令*/
    for ( j=0; j<c->mstate.count; j++ ) {
        c->argc=c->mstate.commands[j].argc;
        c->argv=c->mstate.commands[j].argv;
        c->cmd=c->mstate.commands[j].cmd;
        call ( c ); //执行命令
        c->mstate.commands[j].argc=c->argc;
        c->mstate.commands[j].argv=c->argv;
        c->mstate.commands[j].cmd=c->cmd;
    }
    c->argv=orig_argv;
    c->argc=orig_argc;
    c->cmd=orig_cmd;
    /*释放命令队列*/
```

```
freeClientMultiState ( c );  
initClientMultiState ( c );  
/*去除事务处理标志*/  
c->flags&=~ ( REDIS_MULTI | REDIS_DIRTY_CAS );  
server.dirty++;  
}
```

当客户端连接调用exec命令，Redis遍历命令队列并且调用队列中的命令（加粗部分代码）。执行完队列中所有命令后，Redis释放命令队列占用的内存，并且去除连接的事务处理标志，退出事务处理上下文。

10.3 持久化

Redis是基于内存的数据库，内存数据库有个严重的弊端：突然宕机或者断电时，内存的数据不会保存。为了解决这个问题，Redis提供两种持久化方式：内存快照（Snapshotting）和日志追加（Append only file）。下面介绍这两种持久化方式。

10.3.1 内存快照

内存快照方式是将内存中的数据以快照方式写入二进制文件中，默认文件名为dump.rdb。

Redis每隔一段时间进行一次内存快照操作，客户端使用save或者bgsave命令告诉Redis需要做一次内存快照操作。save命令在主线程中保存内存快照，Redis由单线程处理所有请求，执行save命令可能阻塞其他客户端请求，从而导致不能快速响应请求，所以建议不要使用save命令。另外要注意，内存快照每次都把内存数据完整地写入硬盘，而不是只写入增量数据。所以如果数据量大，写入操作比较频繁，从而严重影响性能。

与内存快照相关的配置选项如下：

```
save <seconds> <changes>
```

上面的配置表示经过seconds秒或数据更改changes次进行一次内存快照操作。例如下面配置：

```
save 900 1
```

表示经过900秒或数据更改1次就进行一次内存快照操作。设置多个这样的条件实现不同内存快照方案如下：

```
save 900 1
save 300 10
save 60 10000
```

这样，其中一个条件成立，Redis都进行一次内存快照操作。

10.3.2 日志追加

日志追加（aof）方式是把增加、修改数据的命令通过write函数追加到文件尾部（默认是appendonly.aof）。Redis重启时读取appendonly.aof文件中的所有命令并且执行，从而把数据写入内存中。

另外，操作系统内核的I/O接口可能存在缓存，所以日志追加方式不可能立即写入文件中，这样就有可能丢失部分数据。幸运的是Redis提供了解决方法，通过修改配置文件告诉Redis应该在什么时候使用fsync函数强制操作系统把缓存写入磁盘。有以下三种方法：

```
appendonly yes#启动日志追加持久化方式（yes | no）
#appendfsync always#每次收到增加或者修改命令就立刻强制写入磁盘
appendfsync everysec#每秒强制写入磁盘一次
#appendfsync no#是否写入磁盘完全依赖操作系统
```

日志追加方式有效降低数据丢失的风险，同时也带来另一个问题，即持久化文件（appendonly.aof）不断膨胀。例如调用incr nums命令100次，文件就会保存100条incr nums命令，其实99条都是多余的，因为要恢复数据只需要set nums 100。

为了压缩日志文件，Redis提供bgrewriteaof命令。当Redis收到此命令，就使用类似于内存快照方式将内存的数据以命令的方式保存到临时文件中，最后替换原来的日志文件。

提示 内存快照和日志追加都存在缺点，要选择哪种持久化方式需要自己衡量。也可以把这两种持久化都关闭，实现自己的持久化方式，如使用Berkeley DB或者Tokyo Cabinet。

10.4 主从同步

主从同步可以防止主机坏掉导致网站不能正常运作，这种方法即把从机设置为主机即可。Redis支持主从同步，而且配置很简单。Redis主从同步的优点如下：

Master可以有多个Slave。

多个Slave连接到相同Master, Slave还可以连接其他Slave形成图形结构。

不会阻塞Master。当一个或者多个Slave与Master进行初次同步数据时，Master可以继续处理客户端的请求。相反，Slave在初次同步数据时会阻塞从而不能处理客户端的请求（2.2版本后不再阻塞）。

主从同步用来提高系统的伸缩性，比如多个Slave专门用于客户端的读请求。

在Master服务器上禁止数据持久化（注释配置文件中所有save配置选项），只在Slave服务器上进行数据持久化。

10.4.1 Redis主从同步原理

Redis主从同步设置很简单，设置好Slave服务器后，Slave自动和Master建立连接，发送SYNC命令。无论是第一次同步建立的连接还是连接断开后重新建立的连接，Master都启动一个后台进程，将内存数据以快照方式写入文件中，同时Master主进程开始收集新的写命令并且缓存起来。Master后台进程完成内存快照操作后，把数据文件发给Slave, Slave将文件保存到磁盘上，然后把数据加载到内存中。接着Master把缓存的命令发给Slave，后续Master收到的写命令都通过开始建立的连接发送给Slave。当Master与Slave断开连接，Slave自动重新建立连接。如果Master同时收到多个Slave发来的同步请求，其只启动一个进程写数据库镜像，然后发送给所有Slave。

Redis主从同步原理如图10-5所示。



图 10-5 Redis主从同步原理

Redis主从同步过程分两个阶段，第一阶段如下：

- 1) Slave服务器主动连接到Master服务器。
- 2) Slave服务器发送SYNC命令到Master服务器请求同步。
- 3) Master服务器备份数据库到rdb文件。
- 4) Master服务器把rdb文件传输给Slave服务器。
- 5) Slave服务器清空数据库数据，把rdb文件数据导入数据库中。

完成第一阶段，接下来Master服务器把用户所有更改数据的操作，通过命令的形式转发给所有Slave服务器，Slave服务器只需执行Master服务器发送过来的命令就可以达到同步的效果。

相对于MySQL来说，Redis主从复制配置很简单，只需在Slave服务器配置文件中加入以下配置项：

```
slaveof 192.168.1.1 6379#指定Master的IP和端口
```

10.4.2 Slave端的工作流程

Slave服务器端的第一步是连接Master服务器，代码如下：

```
void replicationCron ( void ) {
.....

if ( server.replstate==REDIS_REPL_CONNECT ) {
redisLog ( REDIS_NOTICE, "Connecting to MASTER....." );
if ( syncWithMaster () ==REDIS_OK ) {
redisLog ( REDIS_NOTICE, "MASTER<->SLAVE sync started: SYNC sent" );
if ( server.appendonly ) rewriteAppendOnlyFileBackground ();
}
}
.....
}
```

replicationCron函数每隔1秒运行一次（10个定时器周期），如果Slave服务器的同步状态为REDIS_REPL_CONNECT（还没有连接到Master），调用syncWithMaster函数进行连接。

syncWithMaster函数的作用是连接Master服务器，并且发送SYNC命令请求Master服务器进行同步。syncWithMaster函数代码如下：

```
int syncWithMaster ( void ) {
char buf[1024], tmpfile[256], authcmd[1024];
//连接到Master服务器
int fd=anetTcpConnect ( NULL, server.masterhost, server.masterport );
int dfd, maxtries=5;
//连接Master服务器失败
if ( fd==-1 ) {
redisLog ( REDIS_WARNING, "Unable to connect to MASTER: %s", strerror ( errno ) );
return REDIS_ERR;
}
}
```

```
//如果Master需要认证
if ( server.masterauth ) {
    snprintf ( authcmd, 1024, "AUTH%s \r \n", server.masterauth );
    //发送认证信息
    if ( syncWrite ( fd, authcmd, strlen ( server.masterauth ) +7, 5 ) ==-1 ) {
        close ( fd );
        redisLog ( REDIS _ WARNING, "Unable to AUTH to MASTER: %s",
            strerror ( errno ));
        return REDIS _ ERR;
    }
    if ( syncReadLine ( fd, buf, 1024, 3600 ) ==-1 ) {
        close ( fd );
        redisLog ( REDIS _ WARNING , "I/O error reading auth result from MASTER : %s" ,
            strerror ( errno ));
        return REDIS _ ERR;
    }
    if ( buf[0]! = ' + ' ) { //如果认证失败, 返回错误
        close ( fd );
        redisLog ( REDIS _ WARNING , "Cannot AUTH to MASTER, is the masterauth password
            correct? " );
        return REDIS _ ERR;
    }
}
//发送SYNC命令请求Master进行同步
if ( syncWrite ( fd, "SYNC \r \n", 7, 5 ) ==-1 ) {
    close ( fd );
    redisLog ( REDIS _ WARNING, "I/O error writing to MASTER: %s", strerror ( errno ));
    return REDIS _ ERR;
}
//创建内存快照临时文件
while ( maxtries—— ) {
    snprintf ( tmpfile, 256,
        "temp-%d.%d.rdb", ( int ) time ( NULL ), ( long int ) getpid ( ));
    dfd=open ( tmpfile, O _ CREAT | O _ WRONLY | O _ EXCL, 0644 );
    if ( dfd! =-1 ) break;
    sleep ( 1 );
}
if ( dfd==-1 ) {
    close ( fd );
    redisLog ( REDIS _ WARNING , "Opening the temp file needed for MASTER < - > SLAVE
```

```

synchronization:
    %s", strerror ( errno ));
    return REDIS_ERR;
}
//侦听Master服务器是否可读, 当可读时调用readSyncBulkPayload () 函数
if ( aeCreateFileEvent ( server.el, fd, AE_READABLE, readSyncBulkPayload, NULL ) == AE_ERR ) {
    close ( fd );
    redisLog ( REDIS_WARNING, "Can ' t create readable event for SYNC" );
    return REDIS_ERR;
}
//设置同步状态为REDIS_REPL_TRANSFER
server.replstate=REDIS_REPL_TRANSFER;
server.repl_transfer_left=-1;
server.repl_transfer_s=fd;
server.repl_transfer_fd=dfd;
server.repl_transfer_lastio=time ( NULL );
server.repl_transfer_tmpfile=zstrdup ( tmpfile );
return REDIS_OK;
}

```

syncWithMaster函数首先调用anetTcpConnect函数连接到Master服务器, 发送SYNC命令到Master服务器请求同步操作 (如果Master需要认证, 先进行认证操作)。然后侦听Master服务器是否可读, 如果可读, 调用readSyncBulkPayload函数从Master服务器处读取数据。最后设置Slave服务器的同步状态为REDIS_REPL_TRANSFER, 表示正在从Master服务器处读取数据。

readSyncBulkPayload函数从Master服务器处读取内存快照文件 (rdb文件) 数据, 并保存到本地内存快照文件中。数据传输完毕后, Slave服务器把数据库清空, 并且把内存快照文件数据导入数据库。代码如下:

```

void readSyncBulkPayload ( aeEventLoop*el, int fd, void*privdata, int mask ) {
    .....

    //获取内存快照文件的大小
    if ( server.repl_transfer_left == -1 ) {

```

```
if ( syncReadLine ( fd, buf, 1024, 3600 ) == -1 ) {
    replicationAbortSyncTransfer ( );
    return;
}
.....

//内存快照文件大小
server.repl_ transfer_ left= strtol ( buf+1, NULL, 10 );
return;
}
readlen= ( server.repl_ transfer_ left < ( signed ) sizeof ( buf ) ) ?
server.repl_ transfer_ left : ( signed ) sizeof ( buf );
nread= read ( fd, buf, readlen ); //从Master服务器处读取数据
if ( nread <= 0 ) {
    replicationAbortSyncTransfer ( );
    return;
}
server.repl_ transfer_ lastio= time ( NULL );
//把接收到的数据写入临时内存快照文件中
if ( write ( server.repl_ transfer_ fd, buf, nread ) != nread ) {
    replicationAbortSyncTransfer ( );
    return;
}
server.repl_ transfer_ left -= nread;
//如果数据传输完毕
if ( server.repl_ transfer_ left == 0 ) {
    if ( rename ( server.repl_ transfer_ tmpfile, server.dbfilename ) == -1 ) {
        replicationAbortSyncTransfer ( );
        return;
    }
    emptyDb ( );
    //清空数据库
    aeDeleteFileEvent ( server.el, server.repl_ transfer_ s, AE_ READABLE );
    if ( rdbLoad ( server.dbfilename ) != REDIS_ OK ) {
        replicationAbortSyncTransfer ( );
        return;
    }
}
.....

//创建与Master服务器连接的对象
server.master= createClient ( server.repl_ transfer_ s );
server.master-> flags |= REDIS_ MASTER;
```

```
server.master->authenticated=1;
//设置同步状态为REDIS_REPL_CONNECTED, 表示已连接
server.replstate=REDIS_REPL_CONNECTED;
redisLog ( REDIS_NOTICE, "MASTER<->SLAVE sync: Finished with success");
}
}
```

因为在导入内存快照文件的数据时清空数据库，所以在这个过程中对Slave服务器进行的数据更新都将失去作用。因此，如果要对Slave服务器的数据进行修改，就必须在同步的第一个阶段完成之后进行。

前面对Slave服务器端的工作流程进行了分析，接着来分析Master服务器端的工作流程。

10.4.3 Master端的工作流程

当Master服务器接收到Slave服务器的SYNC命令后，便调用syncCommand函数进行Master端同步工作。syncCommand函数代码如下：

```
void syncCommand ( redisClient*c ) {
    if ( c->flags&REDIS_SLAVE ) return;
    //如果此服务器同是Master和Slave
    //必须等待Slave端工作完毕才能进行Master端工作
    if ( server.masterhost&&server.replstate!=REDIS_REPL_CONNECTED ) {
        addReplyError ( c, "Can ' t SYNC while not connected with my master" );
        return;
    }
    if ( listLength ( c->reply ) !=0 ) {
        addReplyError ( c, "SYNC is invalid with pending input" );
        return;
    }
    //如果服务器正在内存快照备份
    if ( server.bgsavechildpid!= -1 ) {
        redisClient*slave;
        listNode*ln;
        listIter li;
        //判断备份是否由Slave触发的
        listRewind ( server.slaves, &li );
        while ( ( ln=listNext ( &li ) ) ) { //遍历Slave列表
            slave=ln->value;
            if ( slave->replstate==REDIS_REPL_WAIT_BGSAVE_END ) break;
        }
        if ( ln ) {
            //如果此次备份是由其他Slave触发
            listRelease ( c->reply );
            c->reply=listDup ( slave->reply ); //复制Master回复数据
            c->replstate=REDIS_REPL_WAIT_BGSAVE_END;
        } else {
            //如果不是由Slave触发
            //设置同步状态为等待数据库备份开始
            c->replstate=REDIS_REPL_WAIT_BGSAVE_START;
        }
    }
}
```

```
    }
  } else {
    //如果服务器还没有进行内存快照备份，就开启备份工作
    redisLog ( REDIS_NOTICE, "Starting BGSAVE for SYNC" );
    if ( rdbSaveBackground ( server.dbfilename ) != REDIS_OK ) {
      addReplyError ( c, "Unable to perform background save" );
      return;
    }
    //设置同步状态为等待数据库备份完毕
    c->replstate=REDIS_REPL_WAIT_BGSAVE_END;
  }
  c->repldbfd=-1;
  c->flags |= REDIS_SLAVE;
  c->slaveseldb=0;
  //把连接添加到服务器的slaves列表中
  listAddNodeTail ( server.slaves, c );
  return;
}
```

syncCommand函数主要工作是发起一个内存快照备份。

如果此时服务器已经在进行内存快照备份，就要判断此次备份操作是否由其他Slave发起。如果由其他Slave发起，不必发起备份，否则就要等待此次备份完毕后再次发起新的内存快照备份。如果服务器没有进行内存快照备份，即可发起新的内存快照备份。最后把此Slave连接添加到slaves列表中。

内存快照备份完毕后，Master服务器便调用backgroundSaveDoneHandler函数进行一些后续工作，并且调用updateSlavesWaitingBgsave函数进行下一步同步工作。

updateSlavesWaitingBgsave函数的代码如下：

```
void updateSlavesWaitingBgsave ( int bgsaveerr ) {
  listNode*ln;
  int startbgsave=0;
```

```

listIter li;
//遍历slaves列表
listRewind ( server.slaves, &li);
while ( (ln=listNext (&li)) ) {
redisClient*slave=ln->value;
//如果有Slave需要发起一个新的内存快照备份, 把startbgsave设置为1
if ( slave->replstate==REDIS_REPL_WAIT_BGSAVE_START ) {
startbgsave=1;
slave->replstate=REDIS_REPL_WAIT_BGSAVE_END;
} else if ( slave->replstate==REDIS_REPL_WAIT_BGSAVE_END ) {
struct redis_stat buf;
if ( bgsaveerr!=REDIS_OK ) {
freeClient ( slave );
continue;
}
if ( ( slave->repldbfd=open ( server.dbfilename, O_RDONLY )) ==-1 ||
redis_fstat ( slave->repldbfd, &buf ) ==-1 ) {
freeClient ( slave );
continue;
}
slave->repldboff=0;
slave->repldbsize=buf.st_size;
slave->replstate=REDIS_REPL_SEND_BULK;
aeDeleteFileEvent ( server.el, slave->fd, AE_WRITABLE );
if ( aeCreateFileEvent ( server.el, slave->fd, AE_WRITABLE, sendBulkToSlave, slave )
==AE_
ERR ) {
freeClient ( slave );
continue;
}
}
}
//如果有Slave服务器需要发起新的内存快照备份
if ( startbgsave ) {
//发起一个新的内存快照备份
if ( rdbSaveBackground ( server.dbfilename )!=REDIS_OK ) {
listIter li;
listRewind ( server.slaves, &li );
redisLog ( REDIS_WARNING, "SYNC failed.BGSAVE failed" );
while ( ( ln=listNext (&li)) ) {

```

```
redisClient*slave=ln->value;
if ( slave->replstate==REDIS_REPL_WAIT_BGSAVE_START )
freeClient ( slave );
}
}
}
}
```

updateSlavesWaitingBgsave函数遍历Slave列表，为每个Slave连接创建一个写事件，事件回调函数为sendBulkToSlave。如果Slave需要发起一个新的内存快照备份，那么就调用rdbSave

Background函数发起一个新的内存快照备份。

sendBulkToSlave函数把内存快照文件发送给Slave服务器。sendBulkToSlave函数的代码如下：

```
void sendBulkToSlave ( aeEventLoop*el, int fd, void*privdata, int mask ) {
redisClient*slave=privdata;
char buf[REDIS_IOBUF_LEN];
ssize_t nwritten, buflen;
//发送内存快照文件的大小给Slave服务器
if ( slave->repldoff==0 ) {
sds bulkcount;
bulkcount=sdscatprintf ( sdsempty ( ), " $ %lld \r \n", ( unsigned long long ) slave->
repldbsize );
if ( write ( fd, bulkcount, sdslen ( bulkcount )) != ( signed ) sdslen ( bulkcount ))
{
sdsfree ( bulkcount );
freeClient ( slave );
return;
}
sdsfree ( bulkcount );
}
//读取内存快照文件数据
lseek ( slave->repldbufd, slave->repldoff, SEEK_SET );
```

```
buflen=read ( slave->repldbfd, buf, REDIS_IOBUF_LEN );
if ( buflen<=0 ) {
redisLog ( REDIS_WARNING, "Read error sending DB to slave: %s",
( buflen==0)? "premature EOF": strerror ( errno ) );
freeClient ( slave );
return;
}
//发送内存快照文件数据给Slave服务器
if ( ( nwritten=write ( fd, buf, buflen ) ) ==-1 ) {
redisLog ( REDIS_VERBOSE, "Write error sending DB to slave: %s",
strerror ( errno ) );
freeClient ( slave );
return;
}
slave->repldboff+=nwritten;
if ( slave->repldboff==slave->repldbsize ) { //rdb文件传输完毕
close ( slave->repldbfd );
slave->repldbfd=-1;
aeDeleteFileEvent ( server.el, slave->fd, AE_WRITABLE );
slave->replstate=REDIS_REPL_ONLINE;
if ( aeCreateFileEvent ( server.el, slave->fd, AE_WRITABLE,
sendReplyToClient, slave ) ==AE_ERR ) {
freeClient ( slave );
return;
}
addReplySds ( slave, sdsempty ( ) );
}
}
```

sendBulkToSlave函数发送内存快照文件给Slave服务器，发送完毕后，同步的第一阶段告一段落。接下来便进入同步的第二阶段。

同步第二阶段过程中，Master服务器把客户端发送的命令（对数据有修改的命令）转发给Slave服务器，从而实现同步的效果。在call函数中可以看到以下代码：

```
void call ( redisClient*c, struct redisCommand*cmd ) {
```

```
long long dirty;
dirty=server.dirty;
cmd->proc ( c );
dirty=server.dirty-dirty;
.....

if ( ( dirty | | cmd->flags&REDIS_CMD_FORCE_REPLICATION ) &&
listLength ( server.slaves ) ) {
replicationFeedSlaves ( server.slaves, c->db->id, c->argv, c->argc );
}
.....
}
```

dirty变量用于判断命令是否对数据进行修改，如果命令修改了数据，就把此命令转发给Slave。replicationFeedSlaves函数把命令转发给所有Slave服务器。同步的第二个阶段一直进行下去，直到Master服务器停止服务。至此，同步的所有流程分析完毕。

10.5 虚拟内存

Redis的数据保存在内存中，可能出现物理内存不足的情况。物理内存不足时，Redis使用什么方法解决问题呢？答案是使用“虚拟内存”（VM）。

VM是Redis 2.0新增功能。之前，Redis把数据库中的所有数据放在内存中。随着Redis的不断运行，使用的内存会越来越大，最终导致内存不足的情况。Redis的VM与操作系统的VM相似，把很少访问的value保存到磁盘中。与此同时，Redis把value对应的key都放在内存中，为了能够让Redis快速定位到被换出的value所在磁盘位置，从而将其导入到内存中。

操作系统也有虚拟内存功能，为什么Redis要重复“制造轮子”呢？主要有以下两点原因：操作系统的VM是基于页的概念，比如Linux系统中每个页是4KB，而Redis大多数对象远小于4KB，一页上可能有多个Redis对象。另外Redis的集合对象类型如List、set可能存在于多个页上。故Redis自己实现可达到控制换入的粒度。

Redis将交换到磁盘的对象压缩，保存到磁盘的对象可以去除指针和对象元数据信息。一般压缩后的对象比内存中的对象小10倍，这样Redis的VM比操作系统的VM少做很多I/O操作。

10.5.1 配置文件信息

要使用VM，在配置文件中开启相关的配置项：

```
#开启VM功能
vm-enabled yes
#交换出来的value保存的文件路径/tmp/redis.swap
vm-swap-file/tmp/redis.swap
#Redis使用的最大内存上限，超过上限后Redis开始交换value到磁盘文件中
vm-max-memory 268435456
#设置每个页面的大小为32个字节
vm-page-size 32
#最多在文件中使用多少页面，SWAP文件的大小等于vm-page-size*vm-pages
```

```
vm-pages 134217728
#用于执行value对象换入换出的工作线程数量。0表示不使用工作线程
vm-max-threads 4
```

Redis的VM只把value交换到磁盘中，而key依然存储在内存中，目的是让开启VM的Redis和完全使用内存的Redis性能基本保持一致。如果由于太多key而造成内存不足问题，Redis的VM并不能解决。

和操作系统一样，Redis也按照页（page）交换对象。一页只能保存一个对象，但是一个对象可以保存在多页中。

当Redis使用的内存没有超过设置的vm max memory之前，不把任何value交换到磁盘中；当超过最大内存限制后，Redis根据以下算法寻找一个对象交换到磁盘中：

```
swappability=age*log ( size_in_memory );
```

其中age代表这个对象距离上一次被访问的时间，size_in_memory是这个对象在内存中占用的空间大小。Redis采取的策略是把那些很少访问，而且占用内存又比较大的对象交换到磁盘中，但是第二个因素所占的权重更低，所以在公式中取log值。因为交换大对象时，需要占用更多的I/O和CPU资源。

应该根据自己的应用设置vm page size，设置太大浪费磁盘空间，设置太小造成SWAP文件出现过多碎片。

vm max threads表示用于交换任务的工作线程数量，建议不要将vm max threads设置为0，设置为0时交换过程在主线程进行，从而阻塞其他用户。但也不是设置越大越好，因为太多的工作线程导致操作系统使用更多时间来切换线程，从而降低效率。推荐把vm max threads设置为服务器的CPU核心数。

为了理解VM子系统如何工作，需要了解对象在SWAP文件中如何存储。

SWAP文件中采用rdb文件的存储格式。SWAP文件被分割成固定数量的页，每页占

用指定数量的字节空间。在redis.conf中根据自己业务需求配置以下两个参数：

vm page size设置每页的大小，默认值为32。

vm pages设置能够使用的页数，默认值为134217728。

Redis在内存中保存一个bitmap以映射这些页是否被占用，每bit代表对应磁盘空间的页是否被使用。内存中保存这样一份映射表极大增强了Redis性能，同时，对内存的使用又非常少。

10.5.2 开启VM的后台操作

Redis默认持久化方式是开启一个子进程创建rdb文件。Redis调用系统的fork函数创建一个子进程，这样得到当前在内存中数据库的一个完全一致的拷贝，因为fork函数复制了当前进程的整个编程内存空间（fork中采用Copy on Write机制，这里调用fork不会消耗太多内存）。

子进程中持有数据库在某时刻的数据备份，其他客户端提交的命令都将由父进程提供，所以并不会修改子进程的数据。在子进程中，仅仅把整个数据写入dump.rdb文件，然后退出。但是当VM开启时，value可能被交换出内存，并不是在内存中保存所有数据。子进程在执行保存时，与父进程共享同一个SWAP文件，因为：

有命令需要请求交换出内存的数据时，父进程需要把这些对象换回到内存中。

子进程需要访问SWAP文件获得完整数据集。

为了避免两个进程访问同一个SWAP文件，Redis采取一个很简单的方式：当有后台子进程在做快照保存时，父进程不允许把内存中的对象交换到SWAP文件。这样，两个进程都将采取只读模式访问SWAP文件。

这种方式下存在一个问题：有子进程进行内存快照时，因为不能将新对象换出，Redis会申请更多内存，虽然已经超过vm max memory设置。

当然，通常不会出现这个问题，因为后台保存进程能够在很短时间内执行结束。通过vmCanSwapOut函数便知道是否可以VM交换，vmCanSwapOut代码如下：

```
int vmCanSwapOut ( void ) {  
    return ( server.bgsavechildpid==-1 && server.bgrewritechildpid==-1 );  
}
```

从上面的代码知道，Redis正在进行后台内存快照或者后台重写aof时，不能VM交换。

VM系统设计将Redis中的Object从内存中交换到硬盘，以释放内存。Redis只把关联到value的Object交换到硬盘。为了更好地理解这个概念，下面使用Redis内部的DEBUG命令展示一个Object在Redis内部的样子：

```
redis> set foo bar OK
redis> debug object foo
Key at : 0x100101d00 refcount : 1 , value at : 0x100101ce0 refcount : 1 encoding : raw
serializedlength: 4
```

从输出看，Redis最顶层的HashTable中，记录从Redis Objects（keys）到Redis Objects（values）的映射关系。VM系统只能把value对应的Object交换到硬盘中，而把所有关联key的Object放在内存中。这样做很好地保证了查询性能，这也是Redis的VM系统主要设计目的：让开启VM系统的Redis和完全使用内存的Redis保持基本一致的性能。

10.5.3 Redis Object和VM Pointer

当一个value交换到磁盘时，value对应的Redis Object将被替换成VM Pointer，VM Pointer保存value在磁盘的信息，Redis Object和VM Pointer的定义如下：

```
/*Redis Object定义*/
typedef struct redisObject {
    unsigned type: 4;
    unsigned storage: 2; /*REDIS_VM_MEMORY or REDIS_VM_SWAPPING*/
    unsigned encoding: 4;
    unsigned lru: 22;
    int refcount;
    void*ptr;
} robj;
/*VM Pointer定义*/
typedef struct vmPointer {
    unsigned type: 4;
    unsigned storage: 2; /*REDIS_VM_SWAPPED or REDIS_VM_LOADING*/
    unsigned notused: 26;
    unsigned int vtype; /*type of the object stored in the swap file*/
    off_t page; /*the page at witch the object is stored on disk*/
    off_t usedpages; /*number of pages used on disk*/
} vmpointer;
```

从上面代码可以看出，Redis Object和VM Pointer都有一个storage字段，用于标识value存储位置，这个字段可能的值有：

REDIS_VM_MEMORY：该key所关联的value存储在内存里。

REDIS_VM_SWAPPED：value已经被交换到磁盘，Hash表中的value值已被设置为NULL。

REDIS_VM_LOADING：value已经被交换至磁盘，但是现在有一个任务正在把它从磁盘中加载到内存（这个值只在threaded VM开启的时候用到）。

REDIS_VM_SWAPPING: value在内存中，但是有一个任务正在把它从内存中写入磁盘。

所以，当value存储在内存中时，Redis使用一个Redis Object与之关联；而当value存储在磁盘中时，Redis使用一个VM Pointer与之关联。要知道value存储在磁盘中还是内存中，只需判断storage字段。

VM Pointer中主要记录value在磁盘中的信息。

vtype字段：设置成交换出去的对象类型。

page字段：记录对象在SWAP文件中从哪页开始。

usedpages字段：记录对象共包含几页。

10.5.4 交换过程

1. 交换对象至SWAP文件

在单线程VM环境下，交换过程分为以下3个步骤：

步骤1 计算保存这个对象需要占用SWAP文件中的多少页。

在代码实现中，调用rdbSavedObjectPages函数计算一个对象需要占用的磁盘页大小。

步骤2 在SWAP文件中寻找一段连续页空间保存这个对象。

这个任务通过调用vmFindContiguousPages函数实现。当SWAP文件已经用完或者空闲空间过于碎片化，这个函数就会查找失败。这种情况发生时，Redis放弃将该对象保存到SWAP文件，这个对象就会在内存中。vmFindContiguousPages函数代码如下：

```
int vmFindContiguousPages ( off_t*first, off_t n ) {
    off_t base, offset=0, since_jump=0, numfree=0;
    if ( server.vm_near_pages==REDIS_VM_MAX_NEAR_PAGES ) {
        server.vm_near_pages=0;
        server.vm_next_page=0;
    }
    server.vm_near_pages++;
    base=server.vm_next_page;
    while ( offset < server.vm_pages ) {
        off_t this=base+offset;
        /*If we overflow, restart from page zero*/
        if ( this >= server.vm_pages ) {
            this -= server.vm_pages;
            if ( this == 0 ) {
                /*Just overflowed, what we found on tail is no longer
                *interesting, as it ' s no longer contiguous.*/
                numfree=0;
            }
        }
    }
}
```

```
if ( vmFreePage ( this ) ) {
    /*This is a free page*/
    numfree++;
    /*Already got N free pages? Return to the caller, with success*/
    if ( numfree==n ) {
        *first=this- ( n-1);
        server.vm _next _page=this+1;
        redisLog ( REDIS _DEBUG, "FOUND CONTIGUOUS PAGES: %lld pages at%lld \ n", ( long
long )
n, ( long long ) *first);
        return REDIS _OK;
    }
    } else {
        /*The current one is not a free page*/
        numfree=0;
    }
    /*Fast-forward if the current page is not free and we already
    *searched enough near this place.*
    since _jump++;
    if ( ! numfree&&since _jump>=REDIS _VM _MAX _RANDOM _JUMP/4 ) {
        offset+=random ( ) %REDIS _VM _MAX _RANDOM _JUMP;
        since _jump=0;
        /*Note that even if we rewind after the jump, we are don ' t need
        *to make sure numfree is set to zero as we only jump*if*it
        *is set to zero.*
    } else {
        /*Otherwise just check the next page*/
        offset++;
    }
    }
    return REDIS _ERR;
}
```

从上面代码看出，vmFindContiguousPages函数从全部磁盘页中查找n块连续的空闲页。成功返回REDIS_OK，并把first参数设置为连续页的开始地址，失败返回REDIS_ERR。

步骤3 把对象写入SWAP文件。

调用vmWriteObjectOnSwap函数如下:

```
int vmWriteObjectOnSwap ( robj*o, off_t page ) {
//为SWAP文件加上文件锁
if ( server.vm_enabled ) pthread_mutex_lock (&server.io_swapfile_mutex);
if ( fseeko ( server.vm_fp, page*server.vm_page_size, SEEK_SET ) ==-1 ) {
if ( server.vm_enabled )
pthread_mutex_unlock (&server.io_swapfile_mutex);
redisLog ( REDIS_WARNING,
"Critical VM problem in vmWriteObjectOnSwap (): can ' t seek: %s",
strerror ( errno ));
return REDIS_ERR;
}
rdbSaveObject ( server.vm_fp, o );
fflush ( server.vm_fp );
//为SWAP文件解锁
if ( server.vm_enabled ) pthread_mutex_unlock (&server.io_swapfile_mutex);
return REDIS_OK;
}
```

vmWriteObjectOnSwap函数首先锁住SWAP文件，防止其他线程进行读写操作。接着把文件指针移动到要写入页的开始地址，并调用rdbSaveObject函数把对象写入SWAP文件。

对象保存到SWAP文件之后，便从内存中释放。同时，redisObject被替换成vmPointer。并且将storage字段设置为REDIS_VM_SWAPPED，表示此对象已经被交换到磁盘。

2.从SWAP文件中加载对象

从SWAP文件中加载对象更简单，因为在VM Pointer中已经记录对象在SWAP文件的页起始地址和所占页数，只要调用vmLoadObject就可以了。而vmLoadObject最

终调用vmReadObject

FromSwap把对象从磁盘载入内存，vmReadObjectFromSwap代码如下：

```
robj*vmReadObjectFromSwap ( off_t page, int type ) {
    robj*o;
    if ( server.vm_enabled ) pthread_mutex_lock ( &server.io_swapfile_mutex );
    if ( fseeko ( server.vm_fp, page*server.vm_page_size, SEEK_SET ) == -1 ) {
        redisLog ( REDIS_WARNING, "Unrecoverable VM problem in vmReadObjectFromSwap ( ):
can't seek: %
s", strerror ( errno ));
        _exit ( 1 );
    }
    o=rdbLoadObject ( type, server.vm_fp );
    if ( o==NULL ) {
        redisLog ( REDIS_WARNING, "Unrecoverable VM problem in vmReadObjectFromSwap ( ):
can't load ob
ject from swap file: %s", strerror ( errno ));
        _exit ( 1 );
    }
    if ( server.vm_enabled ) pthread_mutex_unlock ( &server.io_swapfile_mutex );
    return o;
}
```

vmReadObjectFromSwap函数首先锁住SWAP文件，避免和其他线程发生冲突。接着调用rdbLoadObject函数把对象从rdb文件载入内存。

10.5.5 阻塞式VM

开启阻塞式VM，需要在配置文件中设置vm max threads为0。配置文件中还有一个设置比较重要——vm max memory，只有当内存占用超过该设定值时，Redis才触发VM交换。

将对象从内存交换至SWAP文件发生在cron任务中，该任务每100毫秒执行一次。如果发现内存使用超过设定值，Redis便循环调用vmSwapOneObject把对象交换至SWAP文件。这个函数接受一个参数，如果传入0，则采取阻塞式交换方式，否则使用I/O线程。vmSwapOneObject代码如下：

```
int vmSwapOneObject ( int usethreads ) {
    int j, i;
    struct dictEntry*best=NULL;
    double best_swappability=0;
    redisDb*best_db=NULL;
    robj*val;
    sds key;
    for ( j=0; j<server.dbnum; j++ ) {
        redisDb*db=server.db+j;
        /*Why maxtries is set to 100?
        *Because this way ( usually ) we ' ll find 1 object even if just 1%-2%
        *are swappable objects*/
        int maxtries=100;
        if ( dictSize ( db->dict ) ==0 ) continue;
        for ( i=0; i<5; i++ ) {
            dictEntry*de;
            double swappability;
            if ( maxtries ) maxtries--;
            de=dictGetRandomKey ( db->dict );
            val=dictGetEntryVal ( de );
            /*Only swap objects that are currently in memory.
            *
            *Also don ' t swap shared objects: not a good idea in general and
            *we need to ensure that the main thread does not touch the
            *object while the I/O thread is using it, but we can ' t
```

```
*control other keys without adding additional mutex.*/  
if ( val->storage! =REDIS_VM_MEMORY | | val->refcount! =1 ) {  
if ( maxtries ) i——; /*don ' t count this try*/  
continue;  
}  
swappability=computeObjectSwappability ( val );  
if ( ! best | | swappability>best_swappability ) {  
best=de;  
best_swappability=swappability;  
best_db=db;  
}}  
}  
if ( best==NULL ) return REDIS_ERR;  
key=dictGetEntryKey ( best );  
val=dictGetEntryVal ( best );  
redisLog ( REDIS_DEBUG, "Key with best swappability: %s, %f",  
key, best_swappability );  
/*Swap it*/  
vmpointer*vp;  
if ( ( vp=vmSwapObjectBlocking ( val ))! =NULL ) {  
dictGetEntryVal ( best ) =vp;  
return REDIS_OK;  
} else {  
return REDIS_ERR;  
}  
}
```

vmSwapOneObject操作可分解为以下4个步骤:

- 1) 找到一个较优的用来交换至SWAP分区的候选对象。
- 2) 调用vmSwapObjectBlocking将对象所关联的value值交换至硬盘，此函数返回一个vm Pointer指针，用于存储value在磁盘中的信息。
- 3) 使用 vmSwapObjectBlocking 返回的 vmPointer 替换 value 对应的 redisObject，并把storage字段设置为REDIS_VM_SWAPPED，表示此value已经交换到磁盘。

4) 释放对象所关联的value占用的内存。

这个函数一直被反复调用，直到SWAP文件已经占满或者内存占用降低至vm max memory所设置的值之下。我们来看看VM交换的触发过程：

```
int serverCron ( struct aeEventLoop*eventLoop, long long id, void*clientData ) {
.....

if ( vmCanSwapOut () ) {
//使用内存超出vm-max-memory设置的最大值
while ( server.vm_ _enabled&& zmalloc_ _used_ _memory () >
server.vm_ _max_ _memory )
{
int retval= ( server.vm_ _max_ _threads==0)?
vmSwapOneObjectBlocking () : /*阻塞式VM*/
vmSwapOneObjectThreaded () ; /*非阻塞式VM*/
if ( retval==REDIS_ _ERR&&! ( loops%300 ) &&
zmalloc_ _used_ _memory () >
( server.vm_ _max_ _memory+server.vm_ _max_ _memory/10 ) )
{
redisLog ( REDIS_ _WARNING, "WARNING: vm-max-memory limit exceeded by more than
10%%
but unable to swap more objects out! " );
}
/*Note that when using threade I/O we free just one object ,
*because anyway when the I/O thread in charge to swap this
*object out will finish, the handler of completed jobs
*will try to swap more objects if we are still out of memory.*!
if ( retval==REDIS_ _ERR | | server.vm_ _max_ _threads>0 ) break;
}
}
.....
}
```

从上面的代码可以知道，当Redis使用的内存超出vm max memory设置的最大值时，VM交换便会触发。阻塞式VM使用vmSwapOneObjectBlocking进行，而非阻塞

式VM使用vmSwapOne

ObjectThreaded进行。

10.5.6 非阻塞式VM

阻塞式VM的优点是简单，但是在阻塞式VM条件下，有客户端访问被换出的数据，或者Redis正在将数据换出时，就意味着Redis将不能处理其他客户端请求，从而导致阻塞其他客户端。

换出操作应该在后台运行，同时，在当有客户端访问已换出的数据时，其他也在访问的客户端得到数据的速度应该与不开启VM时一样。只有那些访问已换出数据的客户端可以延迟。由于这些限制，Redis实现了非阻塞式VM。非阻塞式VM使用I/O线程实现。

Redis使用一个任务队列进行主线程和I/O线程的通信。当主线程需要在后台使用I/O线程完成一些任务时，便push一个I/O任务到server.io_newjobs队列。当系统中还不存在活动I/O线程时，便新建一个。这时，I/O线程会执行这些I/O任务，在任务完成之后把结果push到server.io_processed队列。I/O线程使用UNIX管道给主线程发送1字节的信号，通知主线程有一个新任务已经完成。

iojob（I/O任务）结构体定义如下：

```
typedef struct iojob {
    int type; /*I/O任务的类型*/
    redisDb*db; /*所在数据库*/
    robj*key; /*进行VM交换的key*/
    robj*id;
    robj*val;
    off_t page; /*对象所在页的开始地址*/
    off_t pages; /*对象占用多少页*/
    int canceled; /*I/O任务是否被取消*/
    pthread_t thread; /*线程ID*/
} iojob;
```

I/O线程执行3种类型的任务：

REDIS_IOJOB_LOAD：从SWAP文件中加载一个对象到内存。

REDIS_IOJOB_PREPARE_SWAP: 计算一个对象在保存到SWAP文件时需要占用多少页的空间。

REDIS_IOJOB_DO_SWAP: 把对象从内存中保存到SWAP文件。

而其他工作都由主线程自身处理，如在SWAP文件中找到一段合适的空闲页，确定哪些对象需要交换等。

在非阻塞式VM环境下，有对象需要VM交换时，Redis调用vmSwapObjectThreaded发起一个I/O任务，代码如下：

```
int vmSwapObjectThreaded ( robj*key, robj*val, redisDb*db ) {
    iojob*j;
    j=zmalloc ( sizeof ( *j ) );
    j->type=REDIS_IOJOB_PREPARE_SWAP;
    j->db=db;
    j->key=key;
    incrRefCount ( key );
    j->id=j->val=val;
    incrRefCount ( val );
    j->canceled=0;
    j->thread= ( pthread_t ) -1;
    val->storage=REDIS_VM_SWAPPING;
    lockThreadedIO ();
    queueIOJob ( j );
    unlockThreadedIO ();
    return REDIS_OK;
}
```

vmSwapObjectThreaded函数首先创建一个I/O任务，然后设置I/O任务各属性，接着添加到任务队列。queueIOJob函数负责把I/O任务添加到任务队列，并且在I/O线程少于vm max threads时创建新I/O线程，代码如下：

```
void queueIOJob ( iojob*j ) {
redisLog ( REDIS_DEBUG, "Queued IO Job%p type%d about key ' %s ' \ n",
( void* ) j, j->type, ( char* ) j->key->ptr);
listAddNodeTail ( server.io_newjobs, j);
if ( server.io_active_threads < server.vm_max_threads )
spawnIOThread ();
}
```

上面代码中，spawnIOThread函数用于创建新的I/O线程。I/O线程处理任务队列中的I/O任务，工作代码如下：

```
void*IOThreadEntryPoint ( void*arg ) {
iojob*j;
listNode*ln;
REDIS_NOTUSED ( arg);
pthread_detach ( pthread_self ());
while ( 1 ) {
/*Get a new job to process*/
lockThreadedIO ();
if ( listLength ( server.io_newjobs ) == 0 ) {
/*No new jobs in queue, exit.*/
redisLog ( REDIS_DEBUG, "Thread%d exiting, nothing to do",
( long ) pthread_self ());
server.io_active_threads--;
unlockThreadedIO ();
return NULL;
}
/*从任务队列中取出一个I/O任务*/
ln=listFirst ( server.io_newjobs);
j=ln->value;
listDelNode ( server.io_newjobs, ln);
/*Add the job in the processing queue*/
j->thread=pthread_self ();
listAddNodeTail ( server.io_processing, j);
ln=listLast ( server.io_processing);
unlockThreadedIO ();
```

```

redisLog ( REDIS _ DEBUG , "Thread%d got a new job ( type%d ): %p about key ' %s ' " ,
(long ) pthread _ self
( ) , j - > type , ( void * ) j , ( char * ) j - > key - > ptr );
/*处理I/O任务*/
if ( j - > type == REDIS _ IOJOB _ LOAD ) {
vmpointer * vp = ( vmpointer * ) j - > id ;
j - > val = vmReadObjectFromSwap ( j - > page , vp - > vtype );
} else if ( j - > type == REDIS _ IOJOB _ PREPARE _ SWAP ) {
j - > pages = rdbSavedObjectPages ( j - > val );
} else if ( j - > type == REDIS _ IOJOB _ DO _ SWAP ) {
if ( vmWriteObjectOnSwap ( j - > val , j - > page ) == REDIS _ ERR )
j - > canceled = 1 ;
}
/*Done: insert the job into the processed queue*/
lockThreadedIO ( ) ;
listDelNode ( server.io _ processing , ln );
/*把处理完成的I/O任务添加到io _ processed队列*/
listAddNodeTail ( server.io _ processed , j );
unlockThreadedIO ( ) ;
/*通知主线程有已经完成I/O任务*/
redisAssert ( write ( server.io _ ready _ pipe _ write , "x" , 1 ) == 1 );
}
return NULL ; /*never reached*/
}

```

I/O线程处理过程如下:

- 1) 从io _ newjobs任务队列中取出一个I/O任务。
- 2) 根据I/O任务的类型进行不同操作。
- 3) 把处理完成的I/O任务放进io _ processed任务队列, 同时通知主线程有已经完成的I/O任务 (通过UNIX管道)。

当主线程接收I/O线程发送的信号时, 便调用vmThreadedIOCompletedJob函数对io _ processed任务队列中的I/O任务进行最后的处理。

10.6 扩展库phpredis安装及使用

phpredis是PHP与Redis交互的扩展库，方便与Redis服务器通信。下面看看在Linux下怎么安装phpredis。

1) 下载phpredis源码。

从地址<https://github.com/owlient/phpredis/downloads>处下载最新版本的phpredis源码。

2) 解压，代码如下：

```
$ tar-xzvf phpredis.tar.gz
```

3) 编译安装，代码如下：

```
$ cd phpredis
$ /usr/local/php/bin/phpize
$ ./configure-with-php-config=/usr/local/php/bin/php-config
$ make
$ make install
```

4) 修改php.ini。

最后在php.ini配置文件中添加“extension=redis.so”，重启Web服务器。在phpinfo中看到的的信息如图10-6所示。



redis	
Redis Support	enabled
Redis Version	2.1.0

图 10-6 phpredis信息

对应Redis的每个命令phpredis都实现一个相应方法（更详细的介绍访问网址<https://github.com/nicolasff/phpredis>），下面是一个简单的例子：

```
<? php
$redis=new Redis (); //实例化Redis对象
$redis->connect ("127.0.0.1", 6379); //连接Redis服务器
$redis->set ("key", "value"); //调用set命令存储数据
$value=$redis->get ("key"); //调用get命令获取数据
echo $value; //打印value的值
$redis->close (); //关闭与Redis的连接
? >
```

10.7 Redis应用实践

Redis非常灵活，能够解决许多实际问题。下面具体介绍其在实际中的应用。

10.7.1 使用消息队列发布微博

在一些用户创造内容的应用中（如SNS、微博），可能出现1秒有上万个用户同时发布消息的情况，此时如果使用MySQL很可能出现“too many connections”错误，当然，把MySQL的max_connections参数设置为更大数，不过这是一个治标不治本的方法，这时可以考虑使用Redis。

使用Redis的List类型作为消息队列，把用户发布的消息暂时存储在消息队列中，接着使用一个cron程序把消息队列中的消息插入MySQL。这样有效降低MySQL的并发量。

例如发布一条微博使用以下接口：

```
<? php
$uid=get_uid ();
$content=get_content ();
$timestamp=time ();
$weibo=new Weibo (); //创建Weibo对象
$weibo->post ($uid,$content,$timestamp);
? >
```

Weibo对象的post方法就是发布微博的接口，它直接把微博写入MySQL。参数uid是用户的UID，content是微博的内容，timestamp是发表的时间戳。

为了降低MySQL的并发数，先把用户发布的微博存在Redis中，代码如下：

```
<? php
```

```
$redis=new Redis (' 127.0.0.1 ', 6379); //创建Redis对象
$redis->connect ();
$weibo_info=array (
    ' uid ' =>get_uid (),
    ' content ' =>get_content (),
    ' timestamp ' =>time ()
);
$redis->lpush (' weibo_list ', json_encode ($weibo_info));
$redis->close ();
? >
```

先把微博信息使用`json_encode`编码成JSON格式（JSON是可存储格式，现在流行的编程语言基本都支持），然后使用Redis对象的`lpush`方法把微博信息插入到`weibo_list`队列。

把微博存到Redis以后，编写一个cron程序把Redis中的微博信息插入到MySQL，代码如下：

```
<? php
$redis=new Redis (' 127.0.0.1 ', 6379); //创建Redis对象
$redis->connect ();
$weibo=new Weibo (); //创建Weibo对象
while ( TRUE ) {
    if ( $redis->lsize (' weibo_list ') > 0 ) {
        $info=$redis->rpop (' weibo_list ');
        $info=json_decode ($info);
        $weibo->post ($info->uid,$info->content,$info->timestamp);
    } else {
        sleep ( 1 ); //如果队列中没有任务的时候，睡眠1s，让出CPU给其他进程
    }
}
$redis->close ();
? >
```

在cron程序中，先使用Redis对象的`rpop`（）方法从`weibo_list`列表中取得一条微

博信息，然后使用`json_decode()`函数解码，最后调用Weibo对象的`post`方法把微博信息插入MySQL。

使用消息队列有一个缺点，就是“延时”。为了把延时降到最低，运行多个cron程序同时把消息队列中的数据插入MySQL。使用消息队列的好处是扩展性好，当一台Redis服务器不能应付大量并发时，使用“一致性Hash算法”把并发分发到不同Redis服务器。

10.7.2 Redis替代文件存储Session

PHP默认使用文件存储Session，如果并发量大，效率非常低。而Redis对高并发的支持非常好，所以，可以使用Redis替代文件存储Session。

在讲解实例之前，先了解PHP的`session_set_save_handler`函数的作用和使用方法。该函数定义用户级Session保存函数（如打开、关闭、写入等）。原型如下：

```
bool session_set_save_handler ( callback open, callback close, callback read, callback write,
callback destroy, call back gc )
```

`session_set_save_handler`函数各参数作用如表10-8所示。

参 数	描 述
open	当 Session 打开时调用此函数。接收两个参数，第一个参数是保持 Session 的路径，第二个参数是 Session 的名字。
close	当 Session 操作完成时调用此函数。不接收参数。

参 数	描 述
read	以 Session ID 作为参数，通过 Session ID 从数据存储方中取得数据，并且返回此数据。如果数据为空，可以返回一个空字符串。此函数在调用 <code>session_start</code> 之前被触发。
write	当数据存储时调用。有两个参数，一个是 Session ID，另外一个为 Session 的数据。
destroy	当调用 <code>session_destroy</code> 函数时触发 <code>destroy</code> 函数。只有一个参数 Session ID。
gc	当 PHP 执行 Session 垃圾回收机制时触发。

注意 在使用该函数前，先把`php.ini`配置文件的`session.save_handler`选项设置为`user`，否则`session_set_save_handler`不会生效。

编写一个Session管理类SessionManager，代码如下：

```
<? php
class SessionManager {
private $redis;
private $sessionSavePath;
private $sessionName;
private $sessionExpireTime=30;
```

```
public function construct () {
    $this->redis=new Redis ();
    $this->redis->connect ( ' 127.0.0.1 ', 6379);
    $retval=session_set_save_handler (
    array ($this, "open"),
    array ($this, "close"),
    array ($this, "read"),
    array ($this, "write"),
    array ($this, "destroy"),
    array ($this, "gc")
    );
    session_start ();
}
public function open ($path,$name) {
    return true;
}
public function close () {
    return true;
}
public function read ($id) {
    $value=$this->redis->get ($id);
    if ($value) {
        return $value;
    } else {
        return ' ';
    }
}
public function write ($id,$data) {
    if ($this->redis->set ($id,$data)) {
        $this->redis->expire ($id,$this->sessionExpireTime);
        return true;
    }
    return false;
}
public function destroy ($id) {
    if ($this->redis->delete ($id)) {
        return true;
    }
    return false;
}
```

```
public function gc ( $maxlifetime ) {  
    return true;  
}  
public function destruct ( ) {  
    session_write_close ( );  
}  
}  
?  
>
```

SessionManager构造函数主要用来连接Redis服务器，使用session_set_save_handler函数设置Session回调函数，并调用session_start函数开启Session功能。因为本例中open、close和gc回调函数的作用不是很大，所以直接返回true。

在write回调函数中，以Session ID作为key，把Session的数据作为value存储到Redis服务器，设置Session的过期时间为30秒。在read回调函数中，以Session ID作为key从Redis服务器中读取数据，并返回此数据。而在destroy回调函数中，则以Session ID作为key从Redis服务器中删除对应的Session数据。

使用时只需包含SessionManager类，然后实例化一个SessionManager对象。下面例子使用SessionManager来管理Session，首先创建一个session_set.php文件，输入如下代码：

```
<? php  
include ( "SessionManager.php" );  
new SessionManager ( ); //开启Session管理  
$_SESSION["username"]="Liexusong"; //创建一个名为username的Session变量  
?  
>
```

然后再创建一个session_get.php文件，输入如下代码：

```
<? php  
include ( "SessionManager.php" );
```

```
new SessionManager ();  
echo $_SESSION["username"];  
? >
```

测试时，先访问session_set.php再访问session_get.php，输出结果如图10-7所示。



图 10-7 使用SessionManager管理Session

从测试结果可以看出，Session变量能够跨页面访问，说明SessionManager能够正常工作。另外，可以通过Session ID从Redis查询到相关的Session数据。

10.8 深入了解Redis内核

要更好地使用Redis，必须了解Redis原理。接下来通过源码层深入了解Redis内核。

10.8.1 内存淘汰

Redis是一个内存数据库，当内存不足时，Redis有两种处理方式：

启用虚拟内存：将vm enabled配置项设置为yes。

启用内存淘汰：将maxmemory配置项设置为一个大于0的整数。在内存不足时，内存淘汰是怎么处理的？

在processCommand函数中可以看到以下代码（用户调用命令时触发）：

```
int processCommand ( redisClient*c ) {
.....
if ( server.maxmemory ) freeMemoryIfNeeded ( );
.....
}
```

设置maxmemory选项时调用freeMemoryIfNeeded函数进行内存淘汰，代码如下：

```
void freeMemoryIfNeeded ( void ) {
if ( server.maxmemory_policy==REDIS_MAXMEMORY_NO_EVICTION )
return;
while ( server.maxmemory && zmalloc_used_memory ( ) > server.maxmemory ) {
int j, k, freed=0;
for ( j=0; j<server.dbnum; j++ ) {
long bestval=0; /*just to prevent warning*/
sds bestkey=NULL;
```

```

struct dictEntry*de;
redisDb*db=server.db+j;
dict*dict;
if ( server.maxmemory_policy==REDIS_MAXMEMORY_ALLKEYS_LRU ||
server.maxmemory_policy==REDIS_MAXMEMORY_ALLKEYS_RANDOM )
{
dict=server.db[j].dict;
} else {
dict=server.db[j].expires;
}
if ( dictSize ( dict ) ==0 ) continue;
/*===== ( 1 ) 随机淘汰算法=====*/
if ( server.maxmemory_policy==REDIS_MAXMEMORY_ALLKEYS_RANDOM ||
server.maxmemory_policy==REDIS_MAXMEMORY_VOLATILE_RANDOM )
{
de=dictGetRandomKey ( dict ); //随机查找一个key
bestkey=dictGetEntryKey ( de );
}
/*===== ( 2 ) LRU淘汰算法=====*/
else if ( server.maxmemory_policy==REDIS_MAXMEMORY_ALLKEYS_LRU || server.
maxmemo
ry_policy==REDIS_MAXMEMORY_VOLATILE_LRU )
{
//查找一个最近最少访问的key
for ( k=0; k<server.maxmemory_samples; k++ ) {
sds thiskey;
long thisval;
robj*o;
de=dictGetRandomKey ( dict );
thiskey=dictGetEntryKey ( de );
if ( server.maxmemory_policy==REDIS_MAXMEMORY_VOLATILE_LRU )
de=dictFind ( db->dict, thiskey );
o=dictGetEntryVal ( de );
thisval=estimateObjectIdleTime ( o );
if ( bestkey==NULL || thisval>bestval ) {
bestkey=thiskey;
bestval=thisval;
}
}
}
}

```

```
/*===== ( 3 ) TTL淘汰算法=====*/
else if ( server.maxmemory_policy==REDIS_MAXMEMORY_VOLATILE_TTL )
{
//查找一个最快过期的key
for ( k=0; k<server.maxmemory_samples; k++ ) {
sds thiskey;
long thisval;
de=dictGetRandomKey ( dict );
thiskey=dictGetEntryKey ( de );
thisval= ( long ) dictGetEntryVal ( de );
if ( bestkey==NULL || thisval<bestval ) {
bestkey=thiskey;
bestval=thisval;
}
}
}
/*删除key.*/
if ( bestkey ) {
robj*keyobj=createStringObject ( bestkey, sdslen ( bestkey ) );
dbDelete ( db, keyobj );
server.stat_evictedkeys++;
decrRefCount ( keyobj );
freed++;
}
}
if ( ! freed ) return;
}
}
```

zmalloc_used_memory函数用来取得Redis使用的内存数，例如以下代码：

```
while ( server.maxmemory&&zmalloc_used_memory ( ) > server.maxmemory )
```

当Redis使用的内存数大于可使用的最大内存数时，进行内存淘汰。

相对于Memcached来说（只有LRU淘汰算法），Redis的淘汰算法较丰富，主要有3种：随机淘汰算法：从数据库中随机删除一个key。

LRU淘汰算法：从数据库中删除一个最近最少访问的key。

TTL淘汰算法：从数据库中删除一个最快过期的key。

通过maxmemory policy配置项指定使用的淘汰算法。至于使用哪种淘汰算法，应根据自己的需求设定。

10.8.2 对象引用计数器

设想以下情景，一个客户端调用get命令获取一个较大key时（不能通过一次网络I/O把数据传输完毕），另一个客户端调用del命令删除此key，如果此时没有对key进行任何保护，get操作就有可能导致内存段错误（因为del操作已经把key从内存删除，而get操作还在进行，这样get操作就会访问到非法内存地址）。

为了解决这个问题，Redis使用对象引用计数器。原理是：给对象添加一个引用计数器，每当有地方引用它时，计数器值就加1，当引用失效时，计数器值就减1。当引用计数器为0时，Redis便把此对象从内存中删除。

引用计数器巧妙地解决了get命令和del命令的冲突问题。原理用图10-8所示进行描述。

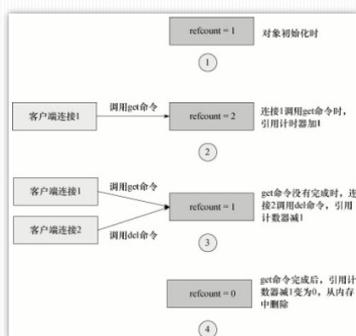


图 10-8 引用计数器原理

要理解对象引用计数器，先了解Redis的对象。在Redis中，所有key和value都是通过对象（Redis Object）进行存储的，对象的结构体定义如下：

```
typedef struct redisObject {
    unsigned type: 4;
    unsigned storage: 2;
    unsigned encoding: 4;
    unsigned lru: 22;
    int refcount;
    void*ptr;
}
```

```
} robj;
```

robj结构体中的ptr字段指向数据的内存地址，refcount字段就是引用计数器。下面看看Redis如何通过引用计数器解决上面的问题。

当Redis创建一个对象时，把对象的引用计数器初始化为1，代码如下：

```
robj*createObject ( int type, void*ptr ) {  
    robj*o=zmalloc ( sizeof ( *o ));  
    o->type=type;  
    o->encoding=REDIS_ENCODING_RAW;  
    o->ptr=ptr;  
    o->refcount=1;  
    o->lru=server.lruclock;  
    o->storage=REDIS_VM_MEMORY;  
    return o;  
}
```

当客户端调用get命令获取一个key时，调用incrRefCount函数把对象的引用计数器加一，incrRefCount函数的代码如下：

```
void incrRefCount ( robj*o ) {  
    o->refcount++;  
}
```

所以，只有一个客户端使用get命令获取key时，此key的对象引用计数器应该为2。通过“object refcount”命令查看一个key的引用计数器（2.2.3以上版本为object命令）。

当客户端调用del命令时，Redis调用decrRefCount函数把对象的引用计数器减1，

如果对象的引用计数器等于0，Redis才把对象从内存中删除。decrRefCount函数代码如下：

```
void decrRefCount ( void*obj ) {
.....

if (—— ( o->refcount ) ==0 ) { //如果引用计数器为0，就释放对象
switch ( o->type ) {
case REDIS_STRING: freeStringObject ( o ); break;
case REDIS_LIST: freeListObject ( o ); break;
case REDIS_SET: freeSetObject ( o ); break;
case REDIS_ZSET: freeZsetObject ( o ); break;
case REDIS_HASH: freeHashObject ( o ); break;
default: redisPanic ( "Unknown object type" ); break;
}
o->ptr=NULL;
zfree ( o );
}
}
```

从上面的分析可知，使用对象引用计数器后，当get操作还没完成时调用del命令也不会出现内存段错误。这是因为调用get命令后对象的引用计数器加1，所以此时调用del命令也不会使其引用计数器变为0（不会从内存中删除此key）。

调用del命令后在什么时候才把这个key从内存中删除呢？答案是等待get操作完成后（如果同时有多个客户端调用get命令，就要等待所有客户端完成后）。当一个get操作完成后，Redis把此key的引用计数器减1。而当所有的get操作都完成后，此key的引用计数器将变为0，此时，Redis就把这个key从内存中删除。

10.8.3 自动关闭超时连接

Redis有自动关闭超时连接的功能，当有一个客户端连接在一定时间内不进行任何操作时，Redis自动关闭它。因为Redis只能处理有限的客户端连接，这个功能有效地防止一些用户恶意占用连接。

开启这个功能在配置文件中把timeout置为大于0的整数，Redis根据客户端最后一次操作时间，判断是否超过timeout配置时间，如果超过timeout配置时间，Redis就关闭此连接。

看看Redis是怎么自动关闭超时的连接，serverCron函数代码如下：

```
int serverCron ( struct aeEventLoop*eventLoop, long long id, void*clientData ) {
.....

if ( ( server.maxidletime&&! ( loops%100 ) ) || server.bpop_blocked_clients )
closeTimedoutClients ( );
.....

}
```

server.maxidletime是timeout设置的时间，如果把timeout设置为0时，这个功能就不会启用。否则，调用closeTimedoutClients函数关闭超时的连接：

```
void closeTimedoutClients ( void ) {
redisClient*c;
listNode*ln;
time_t now=time ( NULL );
listIter li;
listRewind ( server.clients, &li );
while ( ( ln=listNext ( &li ) ) !=NULL ) {
c=listNodeValue ( ln );
if ( server.maxidletime&&
! ( c->flags&REDIS_SLAVE ) &&
! ( c->flags&REDIS_MASTER ) &&
```

```

! ( c->flags&REDIS_BLOCKED ) &&
dictSize ( c->pubsub_channels ) ==0&&
listLength ( c->pubsub_patterns ) ==0&&
( now-c->lastinteraction>server.maxidletime )
{
redisLog ( REDIS_VERBOSE, "Closing idle client" );
freeClient ( c );
} else if ( c->flags&REDIS_BLOCKED ) {
if ( c->bpop.timeout! =0&&c->bpop.timeout<now ) {
addReply ( c, shared.nullmultibulk );
unblockClientWaitingData ( c );
}
}
}
}

```

closeTimedoutClients函数首先遍历所有客户端连接，然后用现在的时间减去最后一次操作的时间再与timeout配置的时间比较，代码如下：

```

if ( now-c->lastinteraction>server.maxidletime )

```

如果超过timeout配置的时间，调用freeClient函数关闭客户端连接。

例如，使用PHP连接Redis后不做任何操作等待一段时间。超过Redis配置的timeout时间后，PHP自动与Redis断开连接，如图10-9所示。

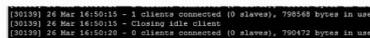


图 10-9 Redis关闭超时连接

图10-9中的“Closing idle client”信息就是关闭超时连接时的调试信息。如果要使用连接池，就必须把timeout配置项设置为0，否则连接池中的连接会因为超时而失效。

10.8.4 清除过期数据

Redis为每个存储的数据设定一个过期的时间，当到达这个设定的过期时间后，Redis便会把过期的数据从内存中删除。

为了提高效率，Redis使用一个HashTable存储数据的过期时间，把数据的key与过期时间相关联，这样就可以通过key来查询数据的过期时间了。

Redis并不会每时每刻去检查数据是否过期，因为这样做效率太低。Redis清除过期数据分两个阶段进行，第一个阶段在定时器中进行（serverCron），第二个阶段在用户获取数据时进行。

在Redis的定时器中，每隔100毫秒进行一次清理过期数据的动作，代码如下：

```
int serverCron ( struct aeEventLoop*eventLoop, long long id, void*clientData ) {
.....
if ( server.masterhost==NULL ) activeExpireCycle ();
.....
}
```

每隔100毫秒Redis便会调用activeExpireCycle函数清理过期数据，activeExpireCycle函数的

代码如下：

```
void activeExpireCycle ( void ) {
int j;
for ( j=0; j<server.dbnum; j++ ) {
int expired;
redisDb*db=server.db+j;
/*Continue to expire if at the end of the cycle more than 25%
*of the keys were expired.*/
do {
```

```
long num=dictSize ( db->expires );
time_t now=time ( NULL );
expired=0;
if ( num>REDIS_EXPIRELOOKUPS_PER_CRON )
num=REDIS_EXPIRELOOKUPS_PER_CRON;
while ( num—— ) {
dictEntry*de;
time_t t;
//随机获取一个设定的过期时间
if ( ( de=dictGetRandomKey ( db->expires ) ) ==NULL ) break;
t= ( time_t ) dictGetEntryVal ( de );
if ( now>t ) { //如果数据已经过期
sds key=dictGetEntryKey ( de );
robj*keyobj=createStringObject ( key, sdslen ( key ) );
propagateExpire ( db, keyobj );
dbDelete ( db, keyobj ); //删除value
decrRefCount ( keyobj );
expired++;
server.stat_expiredkeys++;
}
}
} while ( expired>REDIS_EXPIRELOOKUPS_PER_CRON/4 );
//超过25%数据到期时, 继续清除过期的数据
}
```

activeExpireCycle函数随机获取一些数据的过期时间, 如果当前时间大于数据设定的过期时间, 就把此数据从内存中删除。Redis每次随机获取10个数据的过期时间, 如果这10个数据中有超过25%的数据到达过期时间 (也就是大于等于3个), 这个清理过程会一直进行下去。目的是尽可能删除更多的过期数据, 节省内存的空间。

用户获取数据时也会进行过期数据的清理。

使用get、hget等命令获取数据时, Redis调用expireIfNeeded函数删除过期的数据 (但仅限于当前获取的数据)。该函数判断当前获取的数据是否过期, 如果已经过期, 就从内存中删除。expireIfNeeded函数的代码如下:

```
int expireIfNeeded ( redisDb*db, robj*key ) {
    time_t when=getExpire ( db, key ); //获取数据的过期时间
    if ( when<0 ) return 0; /*如果没有设定过期时间*/
    .....

    if ( time ( NULL ) <=when ) return 0; /*如果没有到达过期时间*/
    /*Delete the key*/
    server.stat_expiredkeys++;
    propagateExpire ( db, key );
    return dbDelete ( db, key ); /*删除过期的数据*/
}
```

首先判断数据是否已经过期，如果已经过期，调用dbDelete函数把它从数据库中删除。db Delete函数除了删除数据外，还会把此数据的过期时间信息删除。

可能大家会有疑问，在定时器中已经把过期的数据删除了，为什么还要在获取数据时对数据进行过期判断呢？因为在定时器中只是随机删除一些过期数据，不可能把所有的过期数据完全删除。

10.9 本章小结

本章主要介绍Redis的使用和原理，从实现上剖析其内部存储和高性能的秘密。只有会使用它，并了解其内部实现原理，才能把它的性能发挥到极致，在遇到问题时才能够快速找到解决方案。

第11章 高性能网站架构方案

Web应用中访问数据库的开销比较大，要提高应用的性能就要避免过多地访问数据库。纵使访问数据库会降低应用性能，有时大量的数据库访问依然在所难免，原因在于：

虽然缓存可以降低数据库的访问次数，但是缓存过期后仍然需要访问数据库，因此数据库依然可能成为应用瓶颈。

数据库的写操作通常不能引入缓存策略。

没有搭建缓存环境，而直接对数据库进行操作。

数据库性能不容忽视，本章将深入介绍几种有效的MySQL性能优化方案。

11.1 如何优化网站响应时间

一个网站能不能留住用户，要看网站的内容，另外一个重要因素就是网站的响应时间。让用户等待的时间过长，最严重的后果是用户流失。所以要经营好一个网站，更重要的是优化网站的响应时间。

有以下几个方法可以优化网站的响应时间。

(1) 减少HTTP请求

下载一个CSS文件或者图片需要一个独立的HTTP请求，而解析HTTP协议的过程需要时间。处于相对顶层的HTTP协议占用更多传输量，所以减少HTTP请求可以降低网站的响应时间和减少传输的数据。要减少HTTP请求，主要的方法有：

将多个图片合并成为一个文件，利用CSS背景图片的偏移技术呈现在网页中，从而减少图片下载的请求。

合并JavaScript脚本和CSS样式文件。

利用浏览器的Cache功能，避免重复下载相同文件。

(2) 动态内容静态化

如果应用中一些内容不经常改动，动态内容静态化是非常有效的加速方法。例如在新闻发布系统中，把发布的新闻生成静态HTML文件，减少服务器脚本的计算时间，从而降低服务器的响应时间。不过这种方法不能用于内容经常变化的应用，如SNS（社交网络）。

(3) 优化数据库

对于使用数据库的Web站点来说，数据库性能关系整个Web应用的性能，如果数据库性能不佳，其他的优化工作也是徒劳无功。所以优化数据库性能，对提高整个Web应用的效率有着举足轻重的作用。

(4) 使用负载均衡

单台Web服务器处理能力有限，单台服务器承受的压力达到极限时，需要有更多的服务器分担工作，我们需要想办法将流量合理分配到更多的服务器上。

实现负载均衡有多种方法，如HTTP重定向、基于DNS的轮询解析、反向代理服务。无论哪种方法，最终目的都是把流量分配到更多服务器上，从而降低单台服务器的压力。

(5) 使用缓存

缓存把需要花费昂贵开销的计算结果保存起来，在以后需要时直接取出，从而避免重复计算。而在Web应用中，数据库的访问耗时相对较多，所以减少数据库的访问次数可以有效提高应用性能。

缓存的方法有很多，可以把查询数据库的结果存储为一个PHP文件，需要时直接include到程序中即可。也可以把查询结果存储在Memcached，需要时直接从Memcached中读取。

11.1.1 吞吐率

吞吐率指单位时间内服务器处理的请求数，通常使用“reqs/s”（服务器每秒处理请求的数量）表示。在一些常见Web服务器软件中，通常提供当前服务器运行状态以及吞吐率的查看方法。

如使用Lighttpd的mod_status模块监控Lighttpd的吞吐率。表11-1展示了Lighttpd最近5s的吞吐率。

从表中可以得知Lighttpd服务器从启动到目前时刻的吞吐率以及最近5s的吞吐率，当中包括单位时间流出数据量。从表中看出，最近5s吞吐率为188 reqs/s，而所有时间的平均吞吐率为132 reqs/s。

吞吐率只描述服务器在实际运行期间单位时间内处理的请求数，而我们更加关心服务器并发处理能力的上限，即单位时间内服务器能够处理的最大请求数（即最大吞吐率）。但是在测试时，很难调动足够的人测试服务器的最大吞吐率。所以，需要使用某些方法模拟足够数目的并发用户数，这种方法称为“压力测试”。

Hostname	localhost
Uptime	41 days 23 hours 5 min 3s
Started at	2011-10-02 15: 23: 12
absolute (since start)	
Requests	439 Memq
Traffic	1. 28 TB
average (since start)	
Requests	132 reqs/s
Traffic	379. 50 KB/s
average (5s sliding average)	
Requests	188 reqs/s
Traffic	335. 09 KB/s

11.1.2 压力测试

压力测试工具很多，如LoadRunner、JMeter和ab等。由于LoadRunner和JMeter使用过于复杂，本书中大部分使用ab。

ab (Apache Bench) 是Apache附带的压力测试软件，容易使用，功能完全能够满足我们的要求。本书中使用Apache 2.2.11中附带的ab，其版本信息如下所示：

```
$ ab-V
This is ApacheBench, Version 2.0.40-dev < $Revision: 1.146 $ > apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.apache.org/
```

ab参数说明如下：

```
$ ab-h
Usage: ab[options][http: //]hostname[: port]/path
Options are:
-n requests  Number of requests to perform
-c concurrency Number of multiple requests to make
-t timelimit Seconds to max.wait for responses
-p postfile File containing data to POST
-T content-type Content-type header for POSTing
-v verbosity How much troubleshooting info to print
-w Print out results in HTML tables
-l Use HEAD instead of GET
-x attributes String to insert as table attributes
-y attributes String to insert as tr attributes
-z attributes String to insert as td or th attributes
-C attribute Add cookie, eg. ' Apache=1234. ( repeatable )
-H attribute Add Arbitrary header line, eg. ' Accept-Encoding: gzip '
Inserted after all normal header lines. ( repeatable )
-A attribute Add Basic WWW Authentication, the attributes
are a colon separated username and password.
```

-P attribute Add Basic Proxy Authentication, the attributes are a colon separated username and password.
-X proxy: port Proxyserver and port number to use
-V Print version number and exit
-k Use HTTP KeepAlive feature
-d Do not show percentiles served table.
-S Do not show confidence estimators and warnings.
-g filename Output collected data to gnuplot format file.
-e filename Output CSV file with percentages served
-h Display usage information (this message)

ab的参数比较多，常用的只有以下几个：

n: 在测试会话中执行的请求个数，默认执行一个请求。

c: 要创建的并发用户数，默认创建一个用户数。

t: 等待Web服务器响应的最大时间（单位：秒），默认没有时间限制。

k: 使用Keep Alive特性。

c: 对请求附加一个Cookie，形式为name=value。

下面使用ab进行一次压力测试：

```
$ ab-c10-n1000 http://localhost/index.php
This is ApacheBench, Version 2.0.40-dev < $Revision: 1.146 $ > apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 2006 The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost ( be patient )
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
```

```
Completed 800 requests
Completed 900 requests
Finished 1000 requests
Server Software: Apache/2.2.8
Server Hostname: localhost
Server Port: 80
Document Path: /index.php
Document Length: 3607 bytes
Concurrency Level: 10
Time taken for tests: 3.73205 seconds
Complete requests: 1000
Failed requests: 0
Write errors: 0
Total transferred: 3794000 bytes
HTML transferred: 3607000 bytes
Requests per second: 325.39[#/sec] ( mean )
Time per request: 30.732[ms] ( mean )
Time per request: 3.073[ms] ( mean, across all concurrent requests )
Transfer rate: 1205.58[Kbytes/sec]received
Connection Times ( ms )
min  mean[+/-sd]median  max
Connect: 0  0  3.9  0  15
Processing: 0  28  11.6  31  78
Waiting: 0  27  11.7  31  78
Total: 0  29  11.6  31  78
Percentage of the requests served within a certain time ( ms )
50%31
66%31
75%31
80%31
90%46
95%46
98%62
99%62
100%78 ( longest request )
```

测试结果一目了然，吞吐率为325.39reqs/s。

测试结果中关注以下内容：

Server Software：被测试Web服务器的软件名称。

Server Hostname：请求URL中的主机名称。

Server Port：被测试Web服务器的侦听端口。

Document Path：请求的URL的绝对路径。

Document Length：HTTP响应数据的正文长度。

Concurrency Level：并发用户数，设置的“c”参数。

Time taken for tests：所有请求处理完成所花费的总时间。

Complete requests：总请求数，设置的“n”参数。

Failed requests：失败的请求数。

Total transferred：所有请求的响应数据长度总和。

HTML transferred：所有请求的响应数据中正文数据的总和。也就是减去Total transferred中HTTP响应数据中头信息的长度。

Requests per second：Web服务器的吞吐率，等于Complete requests/Time taken for tests。Time per request：用户平均请求等待时间，等于Time taken for tests/ (Complete requests/Concurrency Level)。

Transfer rate：这些请求在单位时间内从服务器取得的数据长度，等于Total transferred/Time taken for tests。

11.1.3 持久连接

持久连接（Keep Alive）又称长连接，指TCP连接中持续发送多份数据而不断开的连接，与之对应的称为短连接，就是连接后发送一次数据便断开的连接。

建立一个TCP连接需要进行三次握手（Three Way Hand Shake），如图11-1所示。

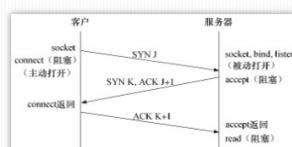


图 11-1 TCP三次握手

从图11-1中看出，TCP三次握手需要交换三个分组的数据，而交换数据需要耗费一定时间。在允许的情况下，连接次数越少，越有利于性能的提升。

由于HTTP具有无状态特性，因此不依赖于TCP的长连接。通常情况一次TCP连接只处理一个HTTP请求，请求处理完毕后马上关闭此次TCP连接。所以，如果HTTP支持长连接，好处是显而易见的，特别对于密集型图片或者网页等小数据请求处理有明显的加速作用。幸运的是，HTTP 1.1对长连接有完整定义，同时，很多浏览器和Web服务器也纷纷提供对长连接的支持。

HTTP长连接的实施需要浏览器和Web服务器共同协助完成。一方面，浏览器需要保持一个TCP连接不能释放，并且不断地发生多个请求。另一方面，服务器不能过早地主动关闭连接。目前的浏览器普遍支持长连接，只需要在发出的HTTP请求数据头中包含相关长连接声明即可，如下所示：

```
Connection: Keep-Alive
```

这个声明告诉服务器：如果可以，请让我重用这个连接。告诉服务器不要在处理完当前请求后马上关闭连接。

同时，Web服务器必须打开长连接的支持。目前所有主流Web服务器软件都支持长连接，如开启Apache 2.2.11对长连接的支持的方法如下：

```
KeepAlive On
```

每个长连接都需要设置一个超时时间，这个设置同时出现在Web浏览器和服务器上，所以双方都可以主动关闭连接。对于IE 6来说，默认超时时间为1分钟，Firefox默认超时时间为115秒，通过注册表或浏览器配置修改超时时间。比如在Firefox浏览器地址栏输入about: config进入设置页面，修改network.http.keep alive.timeout参数。

对于Web服务器，通常提供超时时间的配置参数，如Apache中修改KeepAliveTimeout参数如下所示：

```
KeepAliveTimeout 20
```

设置超时时间为20秒，而在默认情况下为5秒。

注意 TCP连接是双向的，任何一方都可以主动关闭连接。所以，当浏览器与Web服务器设置的超时时间不一致时，将以最短的超时时间为准。

在请求大量小文件时，长连接能够有效地减少重新建立连接的开销。在启动ab时加上“k”参数可以使用Keep Alive模式进行测试。

例如，先不使用长连接请求一个567字节的图片，测试结果如下：

```
$ ab-c10-n10000 http://localhost/image.png
Server Software: Apache/2.2.8
Server Hostname:
localhost
Server Port: 80
```

```
Document Path: /image.png
Document Length: 576 bytes
Concurrency Level: 10
Time taken for tests: 14.430026 seconds
Complete requests: 10000
Failed requests: 0
Write errors: 0
Total transferred: 8470000 bytes
HTML transferred: 5760000 bytes
Requests per second: 693.00[#/sec] ( mean )
Time per request: 14.430[ms] ( mean )
Time per request: 1.443[ms] ( mean, across all concurrent requests )
Transfer rate: 573.18[Kbytes/sec]received
Connection Times ( ms )
min mean[+/-sd]median max
Connect: 0 0 3.3 0 15
Processing: 0 12 7.6 15 46
Waiting: 0 12 7.4 15 46
Total: 0 13 7.4 15 46
Percentage of the requests served within a certain time ( ms )
50%15
66%15
75%15
80%15
90%15
95%31
98%31
99%31
100%46 ( longest request )
```

从测试结果看出，在不使用Keep Alive时的吞吐率为693.00reqs/s。使用Keep Alive模式进行测试，结果如下所示：

```
$ ab-c10-n10000-k http://localhost/image.png
Server Software: Apache/2.2.8
Server Hostname: localhost
Server Port: 80
```

```
Document Path/image.png
Document Length: 576 bytes
Concurrency Level: 10
Time taken for tests: 8.502015 seconds
Complete requests: 10000
Failed requests: 0
Write errors: 0
Keep-Alive requests: 9905
Total transferred: 8829362 bytes
HTML transferred: 5762304 bytes
Requests per second: 1176.19[#/sec] ( mean )
Time per request: 8.502[ms] ( mean )
Time per request: 0.850[ms] ( mean, across all concurrent requests )
Transfer rate: 1014.11[Kbytes/sec]received
Connection Times ( ms )
min mean[+/-sd]median max
Connect: 0 0 0.4 0 15
Processing: 0 8 7.5 15 31
Waiting: 0 8 7.5 15 31
Total: 0 8 7.5 15 31
Percentage of the requests served within a certain time ( ms )
50%15
66%15
75%15
80%15
90%15
95%15
98%15
99%15
100%31 ( longest request )
```

从测试结果看出，使用Keep Alive模式的吞吐率为1176.19reqs/s，与之前相比有了大幅度的提高。在此测试中，使用长连接的请求数为9905个，也就是说有9905个请求重用同一个TCP连接。

注意 长连接不一定是正效应，也可能影响服务器的并发性能。

11.2 MySQL响应速度提高方案：HandlerSocket

日本DeNA公司架构师Yoshinori开发的HandlerSocket，以MySQL Plugin的形式运行，在MySQL体系架构中绕开SQL解析这层，使得应用程序直接和InnoDB存储引擎交互，通过合并写入、简单协议等手段提高数据访问的性能，在CPU密集型应用中优势尤其明显。

HandlerSocket解决了缓存问题，InnoDB有成熟的解决方案，通过参数配置用于缓存数据的内存大小。只要参数分配合理，应用程序就能在无须干涉的情况下实现热点数据的缓存，降低缓存维护成本。

11.2.1 HandlerSocket工作原理

HandlerSocket是MySQL的一个Plugin，集成在mysqld进程中；NoSQL无法实现的复杂查询等操作，仍然使用MySQL自身的关系型数据库实现。在运维层面，原来广泛使用的MySQL主从复制等经验继续发挥作用，相比其他NoSQL产品，数据安全更有保障。HandlerSocket原理如图11-2所示。

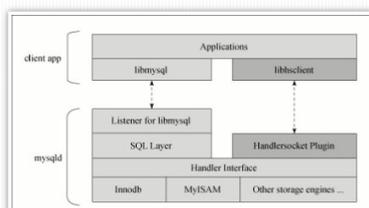


图 11-2 HandlerSocket原理

从图11-2所示中看出，HandlerSocket绕过MySQL的SQL解析层（SQL Layer），直接访问MySQL存储层。另外，HandlerSocket采用epoll和worker thread/thread pooling网络架构，性能更高。

11.2.2 HandlerSocket安装和配置

首先确定系统已经安装MySQL 5.1或以上版本，下载与系统MySQL版本一致的MySQL源代码和HandlerSocket源代码（使用命令“mysql V”查看MySQL版本），如下：

```
$> wget http://downloads.mysql.com/archives/mysql-5.1/mysql-5.1.37.tar.gz
$
>
https://nodeload.github.com/ahiguti/HandlerSocket-Plugin-for-MySQL/tarball/master wget
```

解压MySQL源码和HandlerSocket源码，如下：

```
$> tar xzf mysql-5.1.37.tar.gz
$> tar xzf ahiguti-HandlerSocket-Plugin-for-MySQL-1.0.6-76-gf5f7443.tar.gz
```

编译HandlerSocket，如下：

```
$> cd ahiguti-HandlerSocket-Plugin-for-MySQL-f5f7443 $> ./autogen.sh
$> ./configure --with-mysql-source=../mysql-5.1.37 \
--with-mysql-bindir=/usr/bin \
--with-mysql-plugindir=/usr/lib/mysql/plugin
```

上述代码中参数含义如下：

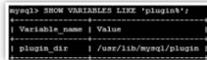
with mysql source: MySQL源代码目录。

with mysql bindir: MySQL二进制可执行文件目录（mysql_config所在目录）。

with mysql plugindir: MySQL插件目录。

按如下方法查询MySQL插件目录，如图11-3所示。

如果在运行configure脚本的过程中出现“MySQL source version does not match MySQL binary version”错误（原因是缺失VERSION文件），可以在MySQL源代码目录下创建一个名为“VERSION”的文件，并且写入以下内容：



```
mysql> SHOW VARIABLES LIKE 'plugin*'
+-----+-----+
Variable_name | Value
+-----+-----+
plugin_dir    | /usr/lib/mysql/plugin
+-----+-----+
```

图 11-3 查询MySQL插件所在目录

```
MYSQL_VERSION_MAJOR=5
MYSQL_VERSION_MINOR=1
MYSQL_VERSION_PATCH=37
MYSQL_VERSION_EXTRA=
```

然后再次运行configure脚本，编译源码，如下：

```
$> make
$> make install
```

编译之后HandlerSocket还不能使用，还要在MySQL配置文件（my.cnf）中增加以下HandlerSocket相关配置项：

```
[mysqld]
#绑定读请求端口
loose_handlersocket_port=9998
#绑定写请求端口
loose_handlersocket_port_wr=9999
#读请求线程数
loose_handlersocket_threads=16
```

```
#写请求线程数  
loose_handlersocket_threads_wr=16  
#设置HandlerSocket最大接收连接数open_files_limit=65535
```

这里增加的这些主要是针对HandlerSocket的配置，它有2个端口，9998读数据，9999读写均可，但是通过9998读的效率更高。这里设置处理读写的线程数均为16个，另外为了处理更多并发连接，设置能打开的文件描述符个数为65535。

此外，InnoDB的innodb_buffer_pool_size或MyISAM的key_buffer_size配置选项关系到缓存索引，所以尽可能设置大一些，这样才能发挥HandlerSocket的潜力。下面把InnoDB的innodb_buffer_pool_size设置为2GB：

```
innodb_buffer_pool_size=2G
```

把所有的准备工作做好之后，现在可以激活HandlerSocket插件了。首先登录MySQL，使用以下命令激活HandlerSocket：

```
mysql> INSTALL PLUGIN handlersocket soname 'handlersocket.so';  
Query OK, 0 rows affected (0.01 sec)
```

如果没有错误发生，能够在MySQL里看到HandlerSocket的线程，使用以下命令查看：

```
mysql> SHOW PROCESSLIST;  
mysql> SHOW PLUGINS;
```

11.2.3 PHP HandlerSocket性能测试

本次测试使用PHP的扩展包PHP HandlerSocket，安装过程如下：

```
$> wget http://php-handlersocket.googlecode.com/files/php-handlersocket-0.3.0.tar.gz
$> tar zxvf php-handlersocket-0.3.0.tar.gz
$> cd php-handlersocket-0.3.0
$> phpize
$> ./configure
$> make
$> make install
```

编译完成后创建一个handlersocket.so文件，把此文件复制到PHP的扩展目录下，然后在php.ini配置文件中添加extension=handlersocket.so配置项，重启PHP服务即可。PHP Handler Socket 详细信息参考 <http://code.google.com/p/php-handlersocket/>。

本次测试主要比较SQL查询和HandlerSocket性能差异。测试结果如图11-4所示。

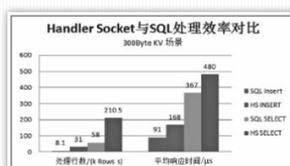


图 11-4 测试结果

从测试结果看出，HandlerSocket性能大概是SQL的4倍，在性能上可能比其他NoSQL产品稍微逊色，但是HandlerSocket支持很多其他NoSQL产品不支持的特性，如范围查询和Limit。HandlerSocket有成熟的数据库引擎（InnoDB）支持，在稳定性和数据安全性方面表现出色。

11.3 MySQL稳定性提高方案：主从复制

主从复制功能通过在主服务器和从服务器之间切分处理客户查询的负荷，可以得到更好的客户响应时间。SELECT查询可以发送到从服务器，以降低主服务器的查询处理负荷。修改数据的语句仍然发送到主服务器，以使主、从服务器保持同步。如果非更新查询为主（如SELECT查询），该负载均衡策略很有效。

MySQL主从复制的优点如下：

增加健壮性。主服务器出现问题时，切换到从服务器作为备份。

优化响应时间。不要同时在主从服务器上进行更新，这样可能引起冲突。

在从服务器备份过程中，主服务器继续处理更新。

11.3.1 主从复制工作原理

主从复制通过3个过程实现，其中一个过程发生在主服务器上，另外两个过程发生在从服务器上。具体情况如下：

主服务器将用户对数据库更新的操作以二进制格式保存到Binary Log日志文件中，然后由Binlog Dump线程将Binary Log日志文件传输给从服务器。

从服务器通过一个I/O线程将主服务器的Binary Log日志文件中的更新操作复制到一个叫Relay Log的中继日志文件中。

从服务器通过另一个SQL线程将Relay Log中继日志文件中的操作依次在本地执行，从而实现主从之间数据的同步。

主从复制详细过程如图11-5所示。

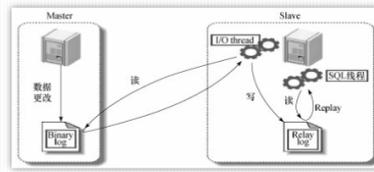


图 11-5 主从复制过程

(1) BinLog Dump线程

BinLog Dump线程运行在主服务器上，主要工作是把Binary Log二进制日志文件的数据发送给从服务器。

使用SHOW PROCESSLIST语句查看该线程是否正在运行。

(2) I/O线程

从服务器执行START SLAVE语句后，创建一个I/O线程。此线程运行在从服务器上，与主服务器建立连接，然后向主服务器发出更新请求。之后，I/O线程将主服务器发送的更新操作复制到本地Relay Log日志文件中。

使用SHOW SLAVE STATUS语句查看I/O线程状态。

(3) SQL线程

SQL线程运行在从服务器上，主要工作是读取Relay Log日志文件中的更新操作，并将这些操作依次执行，从而使主从服务器数据得到同步。

11.3.2 主从复制配置

需要在主、从服务器分别进行相关配置，方法如下：

1) 确认主从服务器的MySQL版本。

MySQL不同版本的BinLog格式可能不一样，最好采用相同版本。如果达不到要求，必须保证主服务器版本不高于从服务器版本。用mysql V命令查看。

2) 在主服务器上为从服务器设置一个连接账户，授予REPLICATION SLAVE权限。

每个从服务器使用标准MySQL用户名和密码连接主服务器。假定域为mydomain.com，要创建用户名为“repl”的账户，从服务器使用该账户从域内任何主机使用密码“pass4slave”访问主服务器。创建该账户使用GRANT语句：

```
mysql> GRANT REPLICATION SLAVE ON *.* TO 'repl' @ '%.mydomain.com'
-> IDENTIFIED BY 'pass4slave';
```

3) 配置主服务器。

打开二进制日志，指定唯一Server ID。例如，在my.cnf配置文件中加入如下值：

```
[mysqld]
log-bin=mysql-bin server-id=1
```

4) 重启主服务器。

运行SHOW MASTER STATUS语句，输出如图11-6所示。

```
mysql> show master status;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000001 | 98       |              |                  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

图 11-6 Master status

File表示主服务器正在使用的binlog文件；Position的值与binlog文件的大小相同，表示下一个被记录事件的位置；Binlog_Do_DB和Binlog_Ignore_DB是主服务器控制写入binlog文件内容的过滤选项，默认为空，表示不做任何过滤。

File和Position两个字段指明从服务器将从哪个binlog文件中复制，以及复制的开始位置，它们也是CHANGE MASTER TO语句的参数。

5) 配置从服务器。

从服务器的配置与主服务器类似，必须提供一个唯一Server ID（不能跟主服务器ID相同），配置完毕后同样需要重启MySQL服务器，配置如下：

```
[mysqld]
log-bin=mysql-bin server-id=2
```

6) 启动从服务器。

接下来让从服务器连接主服务器，并开始重做主服务器binlog文件中的事件。

7) 指定主服务器信息。

使用CHANGE MASTER TO语句指定主服务器的信息，不要在配置文件中设置。该语句可以替代在配置文件中提供主服务器的信息，另外，不需要停止服务器，便可以为从服务器指定不同主服务器，语句如下：

```
mysql> CHANGE MASTER TO MASTER_HOST= ' 192.168.1.100 ',
->MASTER_USER= ' repl ',
->MASTER_PASSWORD= ' pass4slave ',
->MASTER_LOG_FILE= ' mysql-bin.000001 ',
```

```
-> MASTER_LOG_POS=0;
```

此处指定MASTER_LOG_POS的值为0，因为要从日志的开始位置开始读。8) 查看从服务器的设置是否正确。

使用SHOW SLAVE STATUS语句查看从服务器的设置是否正确：

```
mysql> SHOW SLAVE STATUS \ G
*****1.row*****
Slave_IO_State:
Master_Host: 192.168.1.100
Master_User: repl
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000001
Read_Master_Log_Pos: 4
Relay_Log_File: mysql-relay-bin.000001
Relay_Log_Pos: 4
Relay_Master_Log_File: mysql-bin.000001
Slave_IO_Running: No
Slave_SQL_Running: No
.....omitted.....
Seconds_Behind_Master: NULL
```

Slave_IO_State、Slave_IO_Running和Slave_SQL_Running表明从服务器还没有开始复制过程。

注意 日志位置为4而不是0，因为0只是日志文件的开始位置，并不是日志记录事件的开始位置。实际上，MySQL知道的第一个事件的位置是4。

11.3.3 连接主从服务器

1) 执行START SLAVE语句开始复制:

```
mysql> START SLAVE;
```

2) 运行SHOW SLAVE STATUS查看输出结果:

```
mysql> SHOW SLAVE STATUS \ G
*****1.row*****
Slave_IO_State: Waiting for master to send event
Master_Host: 192.168.1.100
Master_User: repl
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000001
Read_Master_Log_Pos: 164
Relay_Log_File: mysql-relay-bin.000001
Relay_Log_Pos: 164
Relay_Master_Log_File: mysql-bin.000001
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
.....omitted.....
Seconds_Behind_Master: 0
```

从结果看出，从服务器的I/O线程和SQL线程都已经开始运行，而且Seconds_Behind_Mas ter不再是NULL。日志位置增加意味着一些事件被获取并执行。如果在主服务器上进行修改，可以在从服务器上看到各种日志文件位置的变化，以及数据库中数据的变化。

使用SHOW PROCESSLIST语句查看主、从服务器的线程状态。

3) 在主服务器上，可以看到从服务器的I/O线程创建的连接：

```
mysql> SHOW PROCESSLIST \ G
*****1.row*****
Id: 1
User: root
Host: localhost: 2096
db: test
Command: Query
Time: 0
State: NULL
Info: show processlist
*****2.row*****
Id: 2
User: repl
Host: localhost: 2144
db: NULL
Command: Binlog Dump
Time: 1838
State: Has sent all binlog to slave; waiting for binlog to be updated
Info: NULL
2 rows in set ( 0.00 sec )
```

第2行就是处理从服务器的I/O线程的连接。

4) 在从服务器上运行SHOW PROCESSLIST语句：

```
mysql> SHOW PROCESSLIST \ G
*****1.row*****
Id: 1
User: system user
Host:
db: NULL
Command: Connect
Time: 2291
```

```
State: Waiting for master to send event
Info: NULL
*****2.row*****
Id: 2
User: system user
Host:
db: NULL
Command: Connect
Time: 1852
State: Has read all relay log; waiting for the slave I/O thread to update it
Info: NULL
*****3.row*****
Id: 5
User: root
Host: localhost: 2152
db: test
Command: Query
Time: 0
State: NULL
Info: show processlist
3 rows in set ( 0.00 sec )
```

第1行为I/O线程状态，第2行为SQL线程状态。

注意 从服务器通过读主服务器的二进制日志实现自我更新，对数据库进行修改的操作都要放在主服务器上执行，而从服务器只用来查询（只读不写的数据库操作）。

11.4 Web应用加速方案: Varnish

在没有任何优化的情况下, 每一个HTTP请求, Web服务器都必须从服务器磁盘中读取请求页面的数据, 然后发送给客户端。相对内存访问速度来说, 磁盘访问的速度极其缓慢(内存访问速度是磁盘访问速度的105106倍)。把访问过的页面缓存到内存中, 下次访问直接从内存中读取, 能有效加快Web应用的访问速度。

11.4.1 传统代理与反向代理

一般情况下, 使用浏览器直接连接其他Internet站点取得网络信息, 直接联系到目的站点服务器, 目的站点服务器把信息传送回来。

介于客户端和Web服务器之间的另一台服务器称为代理服务器, 浏览器不直接到Web服务器取回网页, 而向代理服务器发出请求, 信号先送到代理服务器, 由代理服务器取回浏览器所需要信息并传送给浏览器。工作流程如图11-7所示。

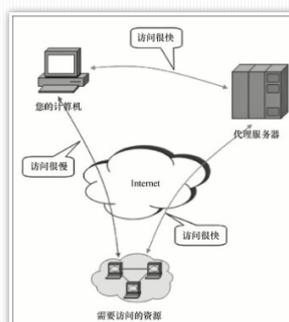


图 11-7 代理原理

大部分代理服务器具有缓冲功能, 好像一个Cache, 有很大存储空间, 不断将新取得的数据储存到本机的存储器上, 如果本机存储器上已经存在用户请求的数据而且是最新的, 直接将存储器上的数据发送给用户, 这样就能显著提高浏览速度和效率。

代理服务器所做的工作是将用户HTTP请求转发给Web服务器, 再将从Web服务器处收到的响应数据发送给用户浏览器。所以, 从Web服务器的角度看, 代理服务器和用户浏览器的本质是一样的, 他们都扮演着HTTP代理的角色。

反向代理 (Reverse Proxy) 与传统代理原理相似, 只是使用对象不同。

传统代理使用对象是客户端程序, 而反向代理使用对象是服务端程序。这也是其称为“反向代理”的根本原因。

引入反向代理后, 用户通过反向代理服务器间接访问Web服务器, 从而把后端Web服务器隐藏。不过用户并不关心这些, 反向代理服务器完美地充当用户心中的Web服务器。至于反向代理服务器和后端Web服务器的沟通, 和传统代理的本质一样, 即使用HTTP协议。

反向代理服务器原理如图11-8所示。

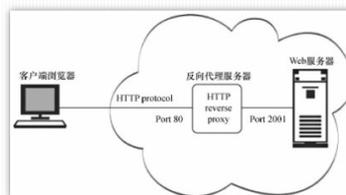


图 11-8 反向代理原理

反向代理有两个作用:

利用反向代理服务器的安全特性处理事务。

利用高速缓存特性在高并发量的服务器上加速。

11.4.2 Varnish安装和配置

反向代理服务器很多，比如Nginx（同时也是高性能的Web服务器）、Squid和Varnish。Squid的功能非常强大，可以做很多其他工作，比如传统代理、访问控制、身份验证和流量管理等。因此，Squid的体积非常庞大，配置过于复杂。而Varnish更专注于反向代理，更简单、效率更高。本节介绍如何使用反向代理服务器Varnish加速Web应用。

下载源文件（<http://www.varnish-cache.org/releases>），按照以下过程安装：

```
$ tar xzf varnish-3.0.1.tar.gz
$ cd varnish-3.0.1
```

假设安装在/usr/local/varnish目录下：

```
$ ./configure——prefix=/usr/local/varnish
$ make
$ make install
```

安装完毕，先把Varnish配置为Apache的代理服务器，修改Varnish配置文件（/usr/local/varnish/etc/default.vcl），如下所示：

```
backend default {
    .host="127.0.0.1";
    .port="80";
}
```

上面的配置中，host为Web服务器IP地址，port为Web服务器侦听端口。使用如下

命令启动Varnish:

```
$ varnishd -f /usr/local/varnish/etc/default.vcl -s malloc, 128M \  
-T 127.0.0.1: 2000 -a: 8080
```

f: 指定Varnish使用的配置文件。

s: 确定Varnish使用的存储类型和存储容量。这里使用malloc类型（内存缓存），也可以使用文件缓存，只需把malloc改为file即可。128M指定Varnish使用多少内存作为缓存。

T: 指定管理程序侦听的地址和端口。在不重启Varnish的情况下，通过这个程序管理Varnish。

a: 指定Varnish侦听的地址和端口。

11.4.3 Varnish性能测试

为了测试反向代理服务器带来的性能提升，需要一些对比的数据，主要包括：

不使用反向代理时的性能数据；

使用反向代理时的性能数据。

1. 不使用反向代理

本次测试对一幅7968字节的图片进行1000次请求，并发数为100。性能报告如下：

```
$ ab-c100-n1000 http://localhost/image.png
Server Software: Apache/2.2.12
Server Hostname: localhost
Server Port: 80
Document Path: /image.png
Document Length: 7968 bytes
Concurrency Level: 100
Time taken for tests: 1.139 seconds
Complete requests: 1000
Failed requests: 0
Write errors: 0
Total transferred: 8289792 bytes
HTML transferred: 8031744 bytes
Requests per second: 878.09[#/sec] ( mean )
Time per request: 113.883[ms] ( mean )
Time per request: 1.139[ms] ( mean, across all concurrent requests )
Transfer rate: 7108.62[Kbytes/sec]received
Connection Times ( ms )
min mean[+/-sd]median max
Connect: 0 6 4.4 6 23
Processing: 12 101 18.9 106 122
Waiting: 3 96 18.8 101 121
Total: 22 107 16.3 113 123
```

测试结果显示，在没有使用反向代理的情况下，Apache的效率为每秒处理878个请求。虽然效率不是很高，但是Apache已经尽力了。

2.使用反向代理

使用反向代理情况下再次对刚才的内容进行压力测试，结果如下：

```
$ ab-c100-n1000 http://localhost:8080/image.png
Server Software: Apache/2.2.12
Server Hostname: localhost
Server Port: 8080
Document Path: /image.png
Document Length: 7968 bytes
Concurrency Level: 100
Time taken for tests: 0.345 seconds
Complete requests: 1000
Failed requests: 0
Write errors: 0
Total transferred: 8283000 bytes
HTML transferred:
7968000 bytes
Requests per second: 2899.28[#/sec] ( mean )
Time per request: 34.491[ms] ( mean )
Time per request: 0.345[ms] ( mean, across all concurrent requests )
Transfer rate: 23451.91[Kbytes/sec]received
Connection Times ( ms )
min mean[+/-sd]median max
Connect: 3 15 7.0 15 28
Processing: 5 18 6.8 19 31
Waiting: 1 13 7.6 12 28
Total: 23 33 3.5 33 43
```

从测试结果看，使用Varnish的情况下，效率是原来的3倍，结果值得我们兴奋。

在测试结果中，Server Software依然显示Apache。我们明明对Varnish进行测试

试，但是为什么显示Apache呢？原因是Varnish为了让自己足够透明，尽量隐藏自己。

通过响应的HTTP头体现Varnish的存在，如图11-9所示。

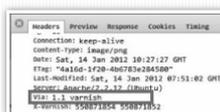


图 11-9 Varnish信息

响应的HTTP头中有一个Via标签，值为varnish，说明正在使用Varnish反向代理服务。

11.4.4 修改缓存规则

如果开启Varnish反向代理，我们请求time.php，而time.php返回当前服务器的时间戳，会出现什么情况呢？很简单，每次请求返回的时间都一样。为什么？

由于Varnish把time.php第一次返回的内容缓存起来，以后访问的都是第一次请求的缓存，所以每次返回的时间都一样。

我们并不希望Varnish缓存动态内容，应该怎么做呢？答案是：修改Varnish配置文件。

Varnish 配置文件使用简单的配置语言 VCL (Varnish Configuration Language)，将反向代理的工作按照时间点分为多个阶段，并在各个阶段通过事件函数（或者回调函数）的风格，将控制权交给我们。下面以最常用的两个阶段为例。

1.vcl_recv函数配置

请求刚刚到达Varnish时调用vcl_recv函数，这里要做的是：告诉Varnish哪些请求需要先查找缓存，哪些请求直接转发到后端服务器。比如以下配置：

```
sub vcl_recv
{
if ( req.request! ="GET"&&
req.request! ="HEAD"&&
req.request! ="PUT"&&
req.request! ="POST"&&
req.request! ="TRACE"&&
req.request! ="OPTIONS"&&
req.request! ="DELETE" ) {
/*Non-RFC2616 or CONNECT which is weird.*/
return ( pipe );
}
if ( req.request! ="GET"&&req.request! ="HEAD" ) {
/*We only deal with GET and HEAD by default*/
return ( pass );
```

```
}  
if ( req.http.Authorization || req.http.Cookie ) {  
/*Not cacheable by default*/  
return ( pass );  
}  
return ( lookup );  
}
```

上面的配置包含3个条件判断语句，在满足条件的情况下会终止函数，返回一个指定值，告诉Varnish接下来应该进入哪个阶段。

比如，pass代表不查找缓存直接将请求转发给后端服务器，接着进入vcl_pass阶段。lookup告诉Varnish需要在缓存中查找内容，然后可能进入vcl_hit阶段，表示缓存命中，或者进入vcl_miss阶段，表示缓存没有命中。

在上面的配置中有一些变量，比如req.request代表来自浏览器的HTTP请求类型，如GET、POST等；req.http.Cookie代表HTTP请求中发送的Cookie内容。

此外，Varnish定义一系列变量，可以在事件函数中获取或者重写（更多Varnish变量可参考Varnish在线手册）：

client.ip: 客户端IP地址。

req.url: 用户请求的URL地址。

req.http.[header key]: 用户HTTP请求头中的信息，如req.http.host代表请求的主机名，

req.http.cookie代表请求发送的Cookie内容。

obj.status: 从后端服务器返回的HTTP状态码，比如200。

obj.response: 从后端服务器返回的HTTP状态信息，比如Not Found。

obj.cacheable: 从服务器返回的内容可以被缓存。

2.vcl_fetch函数配置

Varnish从后端服务器获取内容后调用vcl_fetch函数，在这里决定哪些内容需要缓存，哪些不需要缓存。配置如下：

```
sub vcl_fetch
{
if (! obj.cacheable) {
return ( pass );
}
if ( obj.http.Set-Cookie ) {
return ( pass );
}
set obj.prefetch=-30s;
return ( deliver );
}
```

这里的返回值代表另一些含义，pass代表将内容直接传递给浏览器，而不需要被缓存。deliver代表将内容写入缓存。上面的配置中，第一个条件，判断内容是否可以被缓存，如果不可以，直接发送给浏览器。第二个条件，如果从后端服务器获取的内容包含set cookie的HTTP头，也不需要缓存内容。

3.修改Varnish配置

修改Varnish的配置禁止其缓存动态内容，在vcl_recv函数中加入以下配置：

```
sub vcl_recv {
.....

if ( req.request=="GET"&&req.url~"\.(php)($|\?)" )
{
return ( pass );
}
.....
}
```

加粗代码是新添加的配置，意思是，以GET方式、并且url里以“.php”或者“.php?”结尾的请求，直接转发给后端服务器。

测试以上配置，发现Varnish不再缓存动态内容。

11.4.5 监控Varnish运行状态

有时候需要知道Varnish的运行状况（如缓存命中率和丢失率），这时打开varnish-stat监控程序，如下所示：

```

0+00: 10: 16
Hitrate ratio: 5 5 5
Hitrate avg: 0.8000 0.8000 0.8000
2 0.00 0.00 client_conn-Client connections accepted
12 0.00 0.02 client_req-Client requests received
10 0.00 0.02 cache_hit-Cache hits
2 0.00 0.00 cache_miss-Cache misses
1 0.00 0.00 backend_conn-Backend conn.success
1 0.00 0.00 backend_reuse-Backend conn.reuses
2 0.00 0.00 backend_recycle-Backend conn.recycles
2 0.00 0.00 fetch_length-Fetch with Length
5..n_sess_mem-N struct sess_mem
0..n_sess-N struct sess
0..n_object-N struct object
2..n_objectcore-N struct objectcore
4..n_objecthead-N struct objecthead
2..n_waitinglist-N struct waitinglist
1..n_vbc-N struct vbc
10..n_wrk-N worker threads
10 0.00 0.02 n_wrk_create-N worker threads created
1..n_backend-N backends
2..n_expired-N expired objects
2..n_lru_moved-N LRU moved objects
6 0.00 0.01 n_objwrite-Objects sent with write
2 0.00 0.00 s_sess-Total Sessions
12 0.00 0.02 s_req-Total Requests
2 0.00 0.00 s_fetch-Total fetch
3716 0.00 6.03 s_hdrbytes-Total header bytes
1440 0.00 2.34 s_bodybytes-Total body bytes
1 0.00 0.00 sess_closed-Session Closed
12 0.00 0.02 sess_linger-Session Linger
8 0.00 0.01 sess_herd-Session herd

```

```
978 0.00 1.59 shm_records-SHM records
458 0.00 0.74 shm_writes-SHM writes
2 0.00 0.00 backend_req-Backend requests made
```

上面的信息很多，黑体字部分解说：

Client requests received：目前为止，浏览器向Varnish发送HTTP请求的累积次数。如果使用长连接，结果可能大于上面的**Client connections accepted**。

Cache hits：Varnish在缓存区中查找并且命中缓冲的次数。

Cache misses：Varnish在缓存区中查找但是并没有命中缓冲的次数。

N expired objects：过期缓存内容的个数。

N LRU moved objects：淘汰缓存内容的个数。

Total header bytes：缓存区中所有缓存内容的HTTP头信息长度。

Total body bytes：缓存区中所有缓存内容的正文长度。

缓存命中率的高低直接说明Varnish的运行状态和效果，较高的缓存命中率说明Varnish运行状态良好。反之，过低的缓存命中率说明Varnish的配置可能存在问题，需要进行调整，因此，从整体上了解Varnish命中率和缓存状态，对于优化和调整Varnish至关重要。

11.5 异步计算方案: Gearman

Web应用中比较耗时的操作有:

裁剪用户上传的图片, 或者生成缩略图。

用户上传的文件分发到多台服务器上。

对上传的视频进行转码。

我们应该把耗时的计算交给后台服务器处理。这时异步计算正好派上用场。

11.5.1 Gearman工作原理

Gearman是一个分发任务的程序框架, 与Memcached出于同门。最初用于LiveJournal的图片resize功能。由于图片resize需要消耗大量计算资源, 需要调度后端多台服务器执行, 完成任务之后返回前端再呈现到界面。

Gearman体系包括三个部分:

Client: 创建并发起一个Job请求。

Job Server: 找到合适的Worker, 并把Job交给Worker。

Worker: 执行Job, 事实上所有Job都是由Worker完成。

Gearman提供一系列API让Client、Worker能够与Job Server通信。Gearman工作原理如图11-10所示。

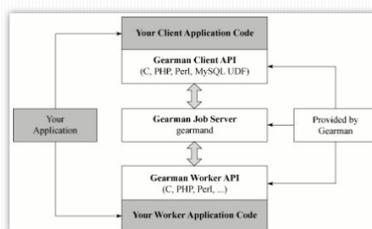


图 11-10 Gearman工作原理

图11-10中灰色部分是我们编写的程序代码，白色部分是由Gearman或者第三方提供的API。只要符合Gearman协议规范就可以跟Gearman沟通，所以Client和Worker并不需要使用同种程序语言实现。例如，可在Client端使用PHP编写程序，而在Worker端使用C或者Java编写。Gearman工作流程如图11-11所示。

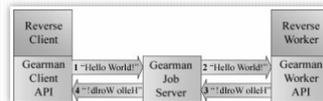


图 11-11 Gearman工作流程

Client负责把一个Job发送给Job Server, Job Server从Worker群中找到合适的Worker并发送Job。Worker负责处理Job, 完成后把结果发送给Job Server, 接着Job Server把Worker发送的结果返回给Client。另外Gearman支持异步模式, Client不必等待Worker返回处理结果而直接返回, 从而实现异步计算。

11.5.2 安装Gearman和PHP扩展

在Ubuntu下安装Gearman 0.20版本（下载地址<http://gearman.org/>）。安装过程如下：

```
$ wget http://launchpad.net/gearmand/trunk/0.20/+download/gearmand-0.20.tar.gz
$ tar xzf gearmand-0.20.tar.gz
$ cd gearmand-0.20
$ ./configure --prefix=/usr/local/gearman
$ sudo make
$ sudo make install
```

其中“/usr/local/gearman”是安装路径。安装Gearman时可能缺失某些依赖库，如lib boost、libevent和uuid等，使用以下命令安装：

```
sudo apt-get install libboost-program-options-dev
sudo apt-get install libevent-dev
sudo apt-get install uuid-dev
```

安装Gearman之后，还要安装Gearman的PHP扩展才可以轻松与Gearman沟通，安装过程如下：

```
$ wget http://pecl.php.net/get/gearman-0.8.0.tgz
$ tar xzf gearman-0.8.0.tgz
$ cd gearman-0.8.0 $ phpize
$ ./configure $ sudo make
$ sudo make install
```

安装Gearman和PHP扩展后，启动Gearman Job Server，命令如下：

```
$/usr/local/gearman/sbin/gearmand-d  
Method for libevent: epoll  
Listening on port 4730
```

11.5.3 使用Gearman异步发送邮件

邮件发送比较耗时，并发量较大时可能出现延时现象。让用户等待邮件发送成功再提示，会大大降低用户体验。所以，使用Gearman实现邮件的异步发送。根据前面的介绍，必须编写Client端和Worker端。

1.Client端程序

Client端程序如下：

```
<? php
$client=new GearmanClient ();
$client->addServer ("127.0.0.1", 4730 );
$info=array (
"to"=>"to@gmail.com",
"subject"=>"Test send email",
"message"=>"Hello! This is a simple email message.",
"headers"=>"From: from@yahoo.com.cn"
);
echo $client->do ("sendmail", serialize ($info));
? >
```

首先实例化一个GearmanClient对象，然后连接到Gearman Job Server；调用do ()方法发送一个名为“sendmail”的Job，功能是发送一封邮件。邮件信息保存在\$info数组中，使用serialize ()函数把\$info数组序列化成字符串。

do ()方法有两个参数，第一个参数告诉Job Server要执行哪个功能，这个功能由Worker提供。第二个参数是传递给Worker的数据，必须是字符串，如果是数组，需要使用PHP的serialize ()函数序列化成字符串。

do ()方法不是异步的，调用该方法时会阻塞，直到Job Server把Worker的结果返回给它为止。如果想使用异步方式，则使用doBackground ()方法发送。详细API参考地址<http://php.net/>

manual/en/book.gearman.php处的相关内容。

编写完Client端程序，接着编写Worker端程序。

2.Worker端程序

Worker端程序从Job Server中取得任务，然后处理任务。

在本例中的任务是发送邮件，代码如下：

```
<? php
$worker=new GearmanWorker ();
$worker->addServer ("127.0.0.1", 4730);
$worker->addFunction (' sendmail ', ' doSendMail ');
while ($worker->work ()); //循环处理任务
function doSendMail ($job) {
$email=unserialize ($job->workload ());
return mail ($email[ ' to ' ],$email[ ' subject ' ],$email[ ' message ' ],$email[ ' headers ' ]);
}? >
```

首先实例化一个GearmanWorker对象，然后连接到Gearman Job Server。调用addFunction ()方法把任务处理函数绑定到Worker中。

addFunction ()方法接受两个参数：第一个是功能名称，对应GearmanClient:do ()方法的第一个参数。第二个是callback函数，用来处理任务的具体实现。

最后，使用一个无限循环接受Gearman Job Server发送的任务（字体加粗部分），调用处理函数doSendMail处理这些任务。

3.运行测试

完成Client端和Worker端程序后，先启动Worker端程序，启动方法如下：

```
$/usr/local/php5/bin/php worker.php
```

接着启动Client端程序，启动方法如下：

```
$/usr/local/php5/bin/php client.php
```

如果并发量不大，邮件应该很快发送出去。但是在并发量非常大的情况下，邮件发送可能会延时。由于采用异步方式发送，用户不必等待邮件发送完毕就可以继续做其他事情。

11.6 本章小结

本章主要介绍网站高性能架构方面的知识。高性能架构的关键是找到性能瓶颈，一般来说，网络通信环节最容易成为Web应用中的瓶颈，例如程序和数据库的通信、程序和内存之间的数据交换阶段，还有可能是I/O的瓶颈。

本章通过几种常见的高性能工具实现高性能网站架构。HandlerSocket和MySQL主从复制保证了MySQL高效读写和稳定性，Varnish反向代理实现服务器端的高效缓存，而Gearman框架则实现异步任务管理。

一个高性能应用不仅仅是高性能工具的简单堆砌，找到瓶颈，有的放矢才是关键。

第12章 代码调试和测试

在程序的开发阶段，必然涉及程序的调试，调试可以减少代码中存在的隐患，帮助程序员更好理清程序的流程，对代码的调试伴随着整个开发过程。当程序基本开发完成后，开发者就需要对代码进行测试，既有针对基本功能的冒烟测试，也有针对模块的单元测试，还有安全测试，压力测试等。通过在开发中和开发后对程序的排错、除错，尽最大努力保证代码的可用性。本章将介绍一些调试和测试的基本概念及操作，以一些典型的工具演示测试方法。

12.1 调试PHP代码

PHP语言没有多线程、同步等复杂概念，大多由函数和类组成，数据类型和语法比较简单，利用其丰富的内置函数即可实现大部分测试功能。因此PHP代码比较容易理解，也很容易调试。

12.1.1 PHP调试函数

对于一般PHP代码，使用几个常用输出函数即可完成代码调试。常用的函数有echo、print、print_r、die、var_dump等。其中echo和var_dump最常用。echo作为一个输出语句，可以放在代码的任何位置，输出需要调试的变量，十分简单且不会终止正常流程。

(1) debug函数

在实际应用中，除了需输出当前变量值外，还需输出一些额外信息，如当前代码所在行号以及所在函数名称。通过代码清单12-1所示代码实现。

代码清单12-1 简单的debug函数示例

```
function debug () {
```

```
$numargs=func_num_args();
$arg_list=func_get_args();
for ($i=0; $i<$numargs; $i++) {
echo"第{$i}个变量的值为: ",$arg_list[$i], PHP_EOL;
}
echo '当前所处的文件名为: ', FILE, PHP_EOL;
}
```

这段代码使用可变参数的方法，输出任意变量的当前值，在类似循环等结构的代码块中调试比较方便。结合一段递归代码跟踪变量的值，相关的代码如代码清单12-2所示。

代码清单12-2 在递归中使用debug函数

```
<? php
//debug函数原型略function factor1 ($n) {
$factor=1;
for ($i=1; $i<=$n; $i++) {
$factor*=$i;
debug ($factor,$i);
}
return $factor;
}
$result=factor1 (4);
? >
```

输出如下：

```
第0个变量的值为: 1第1个变量的值为: 1
当前所处的文件名为: E: \ data \ lua \ d1.php
的值为: 2
第1个变量的值为: 2
当前所处的文件名为: E: \ data \ lua \ d1.php
第0个变量的值为: 6
第1个变量的值为: 3
```

```
当前所处的文件名为: E: \ data \ lua \ d1.php  
第0个变量的值为: 24  
第1个变量的值为: 4  
当前所处的文件名为: E: \ data \ lua \ d1.php
```

通过输出可很直观地看到代码运行过程中变量值的变化过程。

(2) var_dump函数

有的情况，需要知道变量的类型，如果变量是一个矢量类型数据（数组、对象等），可以使用var_dump函数。这个函数在输出变量值的同时会打印变量的数据类型，并且排列工整。在实际应用中，var_dump函数比较常用。

如果要在输出变量值后终止流程，应该使用die、exit语句。die和exit在功能上基本是等价的，都是语句结构，而不是函数。因此，下面的写法都是正确的：

```
exit;  
exit (); exit (0);  
exit ( '程序中止' );
```

注意 尽管调用exit ()，shutdown函数以及对象析构函数总是会被执行。

(3) debug_zval_dump函数

除了以上常见函数和语句外，还有一些不太常用的函数，其中比较重要的是debug_zval_dump函数。这个函数输出结果跟var_dump类似，唯一不同的是增加一个值refcount，记录一个变量被引用多少次，这是PHP写时复制机制的一个重要特点，和PHP变量在底层的实现以及内存回收有关。这个函数可以帮助我们了解PHP的引用机制。

接下来看代码清单12-3所示的代码。

代码清单12-3 PHP中的引用机制

```
<? php
$debugArray=array (1, 2, 3);
foreach ($debugArray as $v) {
    $v*=2;
}
var_dump ($debugArray);
$debugArray2=array (1, 2, 3);
foreach ($debugArray2 as& $v) {
    $v*=2;
}
var_dump ($debugArray2);
```

输出结果如图12-1所示。



图 12-1 PHP中的引用机制

很清晰地看到，在foreach中，由于操作的只是数组的一个副本，对于数组的赋值操作并不会改变数组自身的值。如果需要在foreach中改变数组自身的值，需要使用引用或者改为在for循环中操作数组。使用引用是一个高效的办法。为了验证我们的推断，使用debug_zval_dump函数查看变量的引用次数，如代码清单12-4所示。

代码清单12-4 debug_zval_dump函数的使用示例

```
$debugArray=array (1, 2, 3);
foreach ($debugArray as $v) {
    $v*=2;
    debug_zval_dump ($v);
}
var_dump ($debugArray);
$debugArray2=array (1, 2, 3);
foreach ($debugArray2 as& $v) {
```

```
$v*=2;
debug_zval_dump($v);
}
var_dump($debugArray2);
```

程序输出如图12-2所示。



图 12-2 PHP引用机制

两段代码唯一区别在于引用次数，由于使用了&，操作的是数组自身而不是数组的副本，所以比不用引用的foreach循环少一次变量引用。

(4) debug_print_backtrace函数

在遇到一些递归引用或者函数嵌套时，debug_print_backtrace函数能帮助我们查看程序的调用栈，方便理清程序执行的上下文环境。示例代码如代码清单12-5所示。

代码清单12-5 debug_print_backtrace函数使用示例

```
<? php
function a () {
    b ();
}
function b () {
    c ();
}
function c () {
    debug_print_backtrace ();
}
a ();
```

输出结果如下:

```
#0 c () called at[E: \data \lua \d1.php: 6]
#1 b () called at[E: \data \lua \d1.php: 3]
#2 a () called at[E: \data \lua \d1.php: 11]
```

输出直观地告诉我们，函数c的调用堆栈，根据栈的先进后出特点，输出从下往上看就能理清函数c的调用过程。

如果再加上前面讲过的错误和异常处理中的相关函数，常见的调试就都能得心应手了。

12.1.2 断点调试与变量跟踪工具Xdebug

现在已经能处理绝大多数常见问题了，但是在某些时候还需要更强大有效的调试手段，这就需要利用一些支持断点调试的工具。

当代码涉及很多文件时，比如在MVC中，Action层代码可能会涉及几个甚至十几个文件，这些文件包括框架的核心配置初始化文件、类文件、路由控制文件等，如果用手工调试会比较累。另外，某些应用可能比较复杂，我们并不清楚流程是怎样的，也不知道究竟是哪一步出了问题，该去哪里追踪变量。用手工方法一步一步输出变量显然是笨办法。这些情况下，就轮到断点调试工具Xdebug上场了。

这里使用的IDE是Oracle的NetBeans for PHP 7.0。NetBeans支持多平台的多语言，功能强大，支持的插件比较丰富，是一款和Zend Studio/Eclipse系列同样强大的产品。在调试前，需要先安装Xdebug这款Zend扩展。

以Windows系统为例，到Xdebug官网（<http://xdebug.org/>）下载和PHP版本对应的扩展，尤其注意扩展所适用的PHP版本号、TS（线程安全）版还是NTS（非线程安全）版，以及VC6版本还是VC9版本，这些都应该和安装的PHP对应，否则可能无法使用。这些都可以从扩展的名字里看出。这里，PHP版本是php5.3.8的TS版本，下载的扩展文件是php_Xdebug 2.1.2 5.3 vc9 nts.dll。

下载后，将DLL文件放到ext目录下，编辑php.ini文件，增加下面的配置（已对配置加上注释）：

```
[Xdebug]
; 用中括号表示出来的是模块名称，它会在你的phpinfo信息中作为大的分隔标题显示出来
zend_extension=E:\dev\php538\ext\php_Xdebug-2.1.2-5.3-vc9.dll
; 设置php_Xdebug的DLL文件路径和名称
Xdebug.auto_trace=On
; Xdebug会将php对函数调用的监测信息用文件格式输出来
Xdebug.collect_params=On
; Xdebug会将php对函数调用的参数加入到函数过程调用的监测中
Xdebug.collect_return=On
; 将函数调用的返回值加入到函数过程调用的监测中
```

```

Xdebug.trace_output_dir="E:/debug/phppro"
; 设置的函数调用监测信息的输出路径
Xdebug.profiler_enable=On
; 这是效能监测的设置开关
Xdebug.profiler_output_dir="E:/debug/phppro"
; 这是效能监测信息设置为on的情况下，写入到profiler_output_dir设置的路径中，会生成一个相应的文件
Xdebug.profiler_output_name="cachegrind.out.%p"
Xdebug.remote_enable=on
Xdebug.remote_handler=dbgp
Xdebug.remote_host=localhost
Xdebug.remote_port=9000
xdebug.var_display_max_depth=10; 复杂（多维）变量显示的深度，默认为3，最好将其设置得大一点，方便
; 查看较复杂的变量

```

最后四行是为了让IDE与Xdebug协作。

在NetBeans中调试文件，按F7键表示逐行跟踪文件的运行，可看到系统的每一步时的输出。如果在代码当前位置中断持续运行，进行调试，只需在当前代码处下一个断点即可。

如图12-3所示，在IDE中，在我们需要进行断点调试的行前面双击，即可在此行设置一个断点，一个文件中可以设置多个断点。

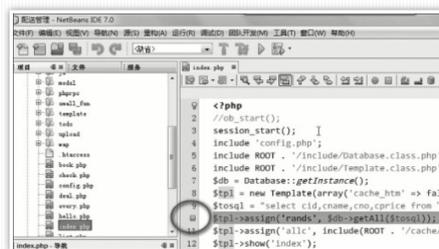


图 12-3 在NetBeans中使用Xdebug进行调试

图12-3中，反色显示的第10行代码即为加断点的代码，现在只想看当程序运行到第10行时，`tosql`的值是什么，可以按以下步骤进行：

1) 在菜单栏中单击“调试”→“调试文件”命令，开始调试当前文件。

2) 单击“步过”按钮，或者按F8键，让调试器快速在当前文件中逐行调试，一直按F8键，直到程序步入调试的第10行。每行最前面小箭头告诉我们当前调试器正在调试的行。

3) 在“步过”过程中，每一步都会在IDE底部的变量视图中显示页面中当前已加载变量的类型和值。当“步过”到第10行时，`tosql`已被赋值，此时就能看到其变量值了，如图12-4所示。

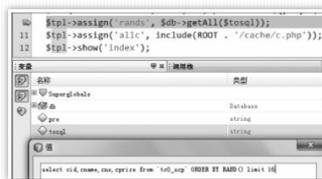


图 12-4 debug截图（一）

可以看到原先`tosql`的变量值已经被真实值替换，如果程序得不到预想结果，在这里要随机输出16条记录，首先看一下SQL语句是否拼装正确，然后放到MySQL客户端执行，借此判断是数据库中数据问题还是PHP语句问题。

平时使用`var_dump(db->getAll($tosql))`来看从数据库中获取的结果集是否正确，但是用了断点调试工具，就不用这么麻烦了，只需要不停地按F8键，观察变量就可以，如图12-5所示。

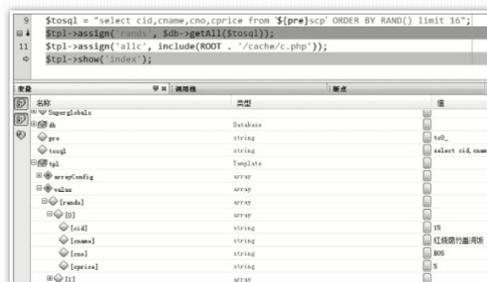


图 12-5 debug截图（二）

使用Xdebug能减少工作量，可以看到当前上下文环境中所有变量的值，对于查错排错有很大帮助。另外，在代码中使用模板引擎，对于Xdebug来说都不是问题，所有变量

值都能在一个框里看到。例如config.php里的配置信息、Template.class.php文件里的模板配置信息（tpl对象的arrayConfig变量）。如果是以前，开发者不得不在文件里放置大堆逐条打印语句，特别是流程比较复杂时，就会显得很麻烦，然后这种情况更能体现断点调试的优势。

比“步过”更精细的是“步入”，如果在调试过程中使用“步入”，Xdebug就会真正一条语句一条语句进入，遇到加载的外部文件时，也会进入其中。这样，对整个程序的运行流程就能做到真正的一清二楚，但同时也使整个debug流程变得冗长。

通过分别使用上面两种不同的调试，大家应该已经理解“步过”和“步入”的区别了。“步过”指跳过当前代码，执行下一行代码，不会虑当前代码的内部执行细节；而“步入”是进入当前代码内部，逐条执行每一条指令。开发者只想知道程序的大致流程时，应该使用“步过”跳过细节，避免花费过多不必要时间。而当代码遇到一些很难排除的bug时，需要精确地知道每条指令的执行细节，此时则选择“步入”。

12.2 前端调试

前端调试主要指对HTML、CSS、JavaScript等调试，相比PHP调试显得更简单。这里，开发者可以选用Firefox最经典的工具Firebug完成前端调试。

Firebug是Firefox下一款开发类插件，属于Firefox五星级插件之一。它集HTML查看和编辑、JavaScript控制台、网络状况监视器于一体，是开发JavaScript、CSS、HTML和AJAX的得力助手。Firebug从各个不同的角度剖析Web页面内部的细节层面，给Web开发者带来很大便利。此外，基于Firebug衍生了FirePHP、FireCookie等系列工具。

12.2.1 Firebug调试API

Firebug提供一些专门用于输出调试信息的API。

在控制台命令行窗口输入console，单击“运行”，可看到Firebug内置对象的一些方法。比如最常用的：

```
console.info ("输出info");  
console.warn ("输出警告");  
console.error ("输出错误");
```

Firebug以不同图标显示这三种不同级别的调试信息，如图12-6所示。



图 12-6 Firebug的console对象

有时需要查看一个对象的详细信息，可使用console.dir () 函数如下所示：

```
var a=new String ("abc");
```

```
console.dir ( a );
console.dir ( $ ( ' nickname ' ));
```

Firebug在显示复杂对象时比较有用，因为用它显示结构会很清晰，而且会显示很多附加信息，如图12-7所示。



图 12-7 Firebug中查看对象

使用`console.dirxml()`函数可显示网页某个节点(`node`)包含的HTML/XML代码。

使用`console.table()`函数可显示对象属性，这个函数和`console.dir()`作用类似，只是显示形式不一样。它以表格的形式把对象打印出来。对于一个对象，使用此函数不仅可以看到该对象的`value`，还可以看到该对象所拥有的方法。比如下面的代码：

```
var table1=new Array ( 5 );
for ( var i=0; i<table1.length; i++ )
table1[i]=[i+1, i+2, i+3, i+4, i+5, i+6, i+7];
console.table ( table1 );
```

输出如图12-8所示。

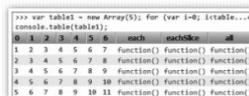


图 12-8 Firebug的调试

`console.trace()`用于追踪函数的调用轨迹，`console.time()`和`console.timeEnd()`用于显示代码的运行时间。看一下这段代码：

```
console.time ( "循环100w" );
```

```
for (var i=0; i<1000; i++) {  
  for (var j=0; j<1000; j++) {}  
}  
console.timeEnd ("循环100w");
```

在Firebug的命令栏中运行这段代码，得到如图12-9所示的结果。



图 12-9 使用Firebug计时器计算循环100万次耗时

除此之外，用console.profile () 函数实现对某个函数的性能分析。

其实不止Firefox，几乎所有主流浏览器都带有调试工具，比如IE的开发人员工具、Opera的Dragonfly等。这些工具的使用方法和界面与Firebug大同小异。以上提到的各种调试工具各有优势。比如，如果要调试IE下的JavaScript代码，首选IE的开发人员工具，同样是按F12键调出此工具，功能不逊于Firebug。

除了浏览器附带的调试工具外，一些专业的JavaScript开发工具也能提供很好的调试体验，如Apanta、WebStorm。

12.2.2 使用Firebug调试DOM结构

DOM调试指对HTML文档的修改与调试，与之相关的还有对CSS的修改、调试。在一个页面完成后，如果觉得效果不理想，或者想看其他效果，最笨的办法就是修改文件中对应的HTML代码以及样式表文件。

当然，这很浪费时间和精力。特别是在调CSS样式的情况下，无法一次到位，通常需要调很多次，不得不经历“修改文件→刷新浏览器观察→再修改文件→再刷新浏览器观察→再修改文件”这一个反复过程，工作效率低下。有了Firebug，一切都改变了。

安装Firebug后，使用快捷键F12即可打开Firebug。在Firebug中要修改HTML文档结构是件很简单的事。如图12-10所示，开发者很方便在Firebug中查看任何DOM结构。



图 12-10 Firebug中查看DOM结构

单击最左侧第二个按钮，用最方便的形式查看网页中的元素。鼠标所到之处，其所在的DOM块都会被一个边框标记，开发者可以看到当前页面元素在HTML源代码里的位置，如图12-11所示。



图 12-11 DOM界面截图

要使HTML文档结构变得简单，只需双击要修改的元素，在Firebug下面的代码栏中定位到元素在HTML文档的位置，直接修改HTML源代码。例如，这里把“暂时还没有留言”改成其他的文字，如图12-12所示。



图 12-12 在Firebug中修改DOM

修改后，页面效果立即显现，如图12-13所示。

据此，开发者可随时随地对页面文档结构进行调整，并立即显示。有时还需做一些样式修改，选择要修改样式的元素，在Firebug右侧看到其对应的CSS代码，同样可以修改并立即在浏览器中得到体现，如图12-14所示。



图 12-13 在Firebug中修改DOM



图 12-14 在Firebug中调试CSS

在Firebug中看清整个DOM层级关系以及任意HTML元素的CSS描述，对于DIV+CSS构造的前端页面特别有用，可帮助我们省去修改源代码的麻烦。

12.2.3 使用Firebug调试JavaScript

JavaScript调试一直是一件烦琐的事情，之前通常在源文件中加入alert语句查看当前变量值，一步一步找到JavaScript代码中存在的问题。现在使用Firebug，一切都变得简单了。

Firebug控制台选项卡中多了一个JavaScript代码调试窗口，在代码栏中输入JavaScript代码并执行，不需要把JavaScript代码写到HTML文件里，然后再用浏览器去看执行效果这么麻烦。更方便的是，这里输入的JavaScript代码和当前页面进行交互，彻底省去开发者修改源文件的烦恼。

下面是一个留言本页面，选择提交时，如果昵称一栏为空，就会提示填写昵称，如图12-15所示。

上面的效果通过下面代码实现判断：

```
function submitForm () {  
  if (! trim ($ ( ' nickname ' ).value)) {  
    alert ( ' 请填写昵称 ' );  
    $ ( ' nickname ' ).focus ();  
    return false;  
  }  
}
```

现在假设要对此函数进行一些修改，增加一个功能，即不允许访客填写昵称为admin。先不用忙着修改JavaScript源代码，在Firebug先测试代码和思路是否可行。

把判断的逻辑改为如下代码：

```
if ( trim ($ ( ' nickname ' ).value) == "" || $ ( ' nickname ' ).value == "admin" ) {  
  alert ( ' 昵称不合法 ' );  
}
```

在Firebug中测试，看看是否正确，如图12-16所示。



图 12-15 程序运行截图



图 12-16 Firebug命令窗口

在昵称一栏先留空，单击“运行”，弹出提示框，提示昵称不合法；然后输入admin，再次运行，仍然是提示昵称不合法。看起来满足了需求，在昵称框中再次输入admin5，仍提示昵称不合法。按照起初的想法，既不是空字符串，也不等于admin，这次理应通过检测，问题出在什么地方呢？再加几行代码打印变量，看看问题到底在哪：

```

alert ( trim ( $ ( ' nickname ' ) . value ) );
alert ( trim ( $ ( ' nickname ' ) . value ) == "" );
alert ( $ ( ' nickname ' ) . value == "admin" );
if ( trim ( $ ( ' nickname ' ) . value == "" ) || $ ( ' nickname ' ) . value == "admin" ) {
    alert ( ' 昵称不合法 ' );
} else {
    alert ( 1 );
}

```

输入昵称admin5，前三次提示分别是“admin5”、“false”、“false”。证明思路是正确的，可是到下面的判断语句还是提示昵称不合法，看来应该是if条件语句的问题。经过仔细检查，发现问题出在if语句。

```

if ( trim ( $ ( ' nickname ' ) . value == "" ) || $ ( ' nickname ' ) . value == "admin" ) {

```

正确语句应该是：

```
if ( trim ( $ ( ' nickname ' ) .value ) == "" | | $ ( ' nickname ' ) .value=="admin" ) {
```

原先的判断括号用错位置，`trim ((' nickname ') .value=="")` 这条语句的意思变成判断trim函数是否执行成功，因为条件始终为真，导致判断出现问题。这是逻辑错误，而不是语法错误，只有经过不断调试才能发现。如果用传统办法，不断修改JavaScript文件里的源代码来调试，会很麻烦。

这么做还不够犀利？试试Firebug断点调试功能。找到处理提交表单的JavaScript文件，在获取昵称的位置打上断点，如图12-17所示。



图 12-17 在Firebug中断点调试JavaScript

回到页面中，昵称一栏什么都不填，提交触发断点，此时Firebug已经捕获这个事件，并在右侧窗口显示。选择单步进入，看 `()` 函数的定义，或者选择继续，进入下一步流程。右侧监控区显示当前断点处的变量值、调用堆栈等，展开就能看清楚。

提示 由于Firebug的一些问题，对一个JavaScript文件进行调试后，想要再次调试就会发现调试按钮变灰无法debug。此时，可清空浏览器缓存，重启浏览器再进行调试。

12.2.4 使用Fiddler调试远程服务器上的文件

在项目开发时，通常会经过“本地服务器”→“测试服务器”→“半正式环境”→“正式环境”这样的过程，主要为了保证正式服务器的安全运行，防止错误代码直接上传到正式服务器。如果是服务器端文件，这样做是很必要的，虽然烦琐但是不会影响到正式服务器；如果对于一些静态资源，这样做就会显得麻烦些了。

使用Firebug直接在正式环境下修改DOM、CSS等静态文件可以立即看到效果，而且可以编辑、新增、删除CSS规则。但是Firebug目前尚不支持修改JavaScript文件，如果JavaScript文件的修改量比较大，又想直接看到在正式服务器上的效果，Firebug就不是很方便了。这个时候可以借助Fiddler工具。

此前已经介绍过Fiddler，并使用Fiddler进行了抓包，现在介绍Fiddler资源重定向功能。

假设需要对用户表单提交页面的JavaScript文件做较大幅度的修改，但是又不想在本地修改后传到服务器测试。这时，Fiddler资源重定向功能就派上用场了。打开Fiddler，请求网页，在左侧Session窗口中选择要替换的本地资源，在右侧窗口选择AutoResponder选项卡，勾选“Enable automatic responses”配置规则，这里用的是精确匹配，选择一个本地文件替换服务器上的文件请求，如图12-18所示。

当请求服务器上的资源时，Fiddler会用本地资源替换它，故Fiddler适用于对文件进行较多修改，如图12-19所示。



图 12-18 Fiddler重定向



图 12-19 重定向后的JavaScript代码

从Firebug中看到，对远程服务器上JavaScript文件的请求已经被重定向为对本地文

件的请求。有时不想因为仅仅是静态资源的修改就在本地搭一个运行环境，然后开启各种服务，就可以使用Fiddler把对正式服务器的请求重定向到对本地文件的请求，再结合Firebug，即可实现完美的调试。

不仅是JavaScript文件，对于image、CSS文件等Fiddler都可以进行重定向。这对AJAX开发以及需求频繁变动的前端开发来说比较方便。

12.3 日志管理

在开发时，通常需要记录一些Log方便后期排错和优化。无论是PHP、Apache，还是数据库，都提供记录Log功能，在适当的时候打开Log记录功能，有助于我们发现代码中的各种问题。

12.3.1 PHP日志

PHP本身不是一种严谨的语言，在产品上线时有可能发生各种运行时错误或者警告，虽然可能不会影响产品的使用，但这也许就是潜在的安全问题或性能问题。所以即使产品已经通过测试并且上线，仍然有必要定期记录日志，定期检查Log文件。

打开PHP的Log记录，只需要在php.ini中设置如下选项：

； 开启日志记录和记录的错误等级

```
Log_errors=On
LogLevel warn
； 记录Log的位置，一般记录在文件中这样比较通用。
error_log=e: /dev/temp/php_errors.Log
```

其他选项保持默认。一旦PHP在运行时遇到错误，就会被记录。Log文件格式：时间+错误级别+错误信息+发生错误的文件+错误所在的代码行。例如：

```
[18-Aug-2011 15: 57: 12]PHP Fatal error: Class ' Object ' not found in C: \ tempdoc \
0815.php on line 2
[20-Aug-2011 10: 30: 19]PHP Deprecated: Call-time pass-by-reference has been depre-
cated in C: \ temp doc \ 0815.php on line 13
```

注意 Log文件的体积可能很大，特别是在循环语句中出现错误时，所以需要定期检查Log文件，并定期清理。记录Log不会对PHP运行效率产生实质性的影响。

12.3.2 Apache服务器日志

Apache默认是开启Log功能的，在conf/httpd.conf中进行如下配置：

```
ErrorLog "Logs/error.Log"  
CustomLog "Logs/access.Log" common
```

错误日志是最重要的Log文件，其文件名和位置取决于ErrorLog指令。Apache httpd将在这个文件中存放诊断信息和处理请求中出现的错误，由于这里经常包含出错细节以及解决方案，故如果服务器启动或运行中有问题，应首先查看这个错误日志。

错误日志格式相对灵活，可以附加文字描述。某些信息会出现在绝大多数记录中，典型的例子如下：

```
[Wed Oct 11 14 : 32 : 52 2000] [error] [client 127.0.0.1]client denied by server  
configuration: /export/home/live/ap/htdocs/test
```

其中，第一项是错误发生的日期和时间；第二项是错误的严重性，LogLevel指令使只有高于指定严重性级别的错误才会被记录；第三项是导致错误的IP地址；此后是信息本身。

比如，某次启动Apache时发现无法启动，经过检测错误日志，发现“Unclean shutdown of previous Apache run?”提示，出现这种错误最大可能就是新增加的扩展不兼容PHP对应版本，以及扩展找不到等。在命令行下运行php.exe, PHP会自动检测扩展的兼容性并给出提示。

CustomLog是Apache的访问日志，格式由LogFormat参数控制，默认记录形式如下：

127.0.0.1—[10/Sep/2011: 16: 37: 09+0800]"GET/index.php HTTP/1.0"200 69843

其中主要项说明如下:

127.0.0.1: 发送请求到服务器的客户IP地址。

[10/Sep/2011: 16: 37: 09+0800]: 服务器完成请求处理时的时间。

"GET/index.php HTTP/1.0": 客户端发出的请求行, 包含动作、请求地址、HTTP协议版本号等。

200: 服务器返回给客户端的状态码。这个信息非常有价值, 因为它指示了请求结果、被成功响应(以2开头)、重定向(以3开头)、出错(以4开头)、产生服务器端错误(以5开头)等状态。完整状态码列表可参见HTTP规范。

69843: 返回给客户端的不包括响应头的字节数。

这几项输出格式大多可以定制, 修改LogFormat参数即可。

CustomLog有很重要的作用, 体现在下列几点:

SEO和网站优化。通过Log查看死链, 以及来访的蜘蛛等, 从而对网站进行针对性的优化。

发现潜在安全问题。由于此Log记录HTTP请求头, 里面有请求地址。帮助我们发现一些攻击者构造的非法请求, 尤其是以数据库注入为主的攻击。

注意, Apache访问日志默认记录的内容可能不满足我们的需求, LogFormat记录如下:

```
LogFormat"%h%l%u%t \ "%r \ "%>s%b \ "% { Referer } i \ " \ "% { User-Agent } i \ ""combined
```

修改为:

```
LogFormat "%h%l%u%t \ "%m%U%q%H \ "%>s%b \ "% { Referer } i \ " \ "% { User-Agent } i  
 \ ""combined
```

然后注释掉http.conf中的#CustomLog"Logs/access.Log"common，去除#CustomLog"Logs/ac

cess. Log"combined这行前面的注释。现在就能记录请求中的查询字符串以及User Agent等信息了。

对于Apache访问日志，量比较大时，人工查看和分析都不方便，可以借助一些现有的软件。

另外，访问量比较大时按日期或文件大小分文件存储日志，Apache自带分卷工具rotate Logs，其使用方法如下：

```
CustomLog" | bin/rotateLogs.exe -l e: /dev/Logs/apache/access%Y-%m-%d.Log 86400"-  
common
```

12.3.3 MySQL日志

任何一种数据库都有日志。MySQL中有4种日志，分别是错误日志、二进制日志、查询日志和慢查询日志，记录MySQL数据库不同方面的踪迹。这些Log都可以在MySQL配置文件my.cnf (Windows下是my.ini) 配置，或者在每次启动MySQL时作为参数传入。

1. 错误日志

错误日志记录MySQL启动和停止，以及服务器在运行过程中发生的任何错误的相关信息。在配置文件里配置“Log error=[file name]”指定错误日志存放的位置。如果没有指定[file name]，默认hostname.err为文件名，默认存放在DATADIR目录中，这是一个文本文件，可用文本编辑器直接打开查看。记录的内容如下所示：

```
110223 15: 58: 08[Note]Plugin ' FEDERATED ' is disabled.
110223 15: 58: 08 InnoDB: Initializing buffer pool, size=17.0M
110223 15: 58: 08 InnoDB: Completed initialization of buffer pool
InnoDB: No valid checkpoint found.
InnoDB: If this error appears when you are creating an InnoDB database,
InnoDB: the problem may be that during an earlier attempt you managed
InnoDB: to create the InnoDB data files, but Log file creation failed.
InnoDB: If that is the case, please refer to
InnoDB: http://dev.MySQL.com/doc/refman/5.1/en/error-creating-innodb.html
110223 15: 58: 08[ERROR]Plugin ' InnoDB ' init function returned error.
110223 15: 58: 08[ERROR]Plugin ' InnoDB ' registration as a STORAGE ENGINE failed.
110223 15: 58: 08[ERROR]Unknown/unsupported table type: innodb
110223 15: 58: 08[ERROR]Aborting
```

若遇到MySQL无法启动的情况，应该首先看这个文件。上面的Log文件就是一次失败的启动记录。注意这一行：

```
110223 15: 58: 08[ERROR]Plugin ' InnoDB ' registration as a STORAGE ENGINE failed.
```

从这里可以看出MySQL启动失败的原因是InnoDB引擎注册失败，应该怎么解决呢？第一个想到的临时解决方案是：启动时先禁止InnoDB引擎。

找到my.ini文件，更改default storage engine=innodb为default storage engine=myisam。这样启动时就不会报错了，但是在建表时InnoDB引擎也就没有用了。继续从错误日志入手，注意到下面这几行记录：

```
InnoDB: to create the InnoDB data files, but Log file creation failed
InnoDB: http: //dev.MySQL.com/doc/refman/5.1/en/error-creating-innodb.html
```

这里告诉我们解决问题的办法，指出是InnoDB日志文件的问题，由于InnoDB和MyISAM的工作原理有很大区别，InnoDB数据依赖其Log文件，所以错误的配置会造成InnoDB引擎初始化失败。通过MySQL官方提示，修改下面几个参数：

```
innodb_buffer_pool_size=512M
innodb_log_file_size=128M
```

把这几个参数的值调大，然后在配置文件里加入tmpdir="usr/tmp"路径，因为InnoDB还需要一个临时的文件缓存区。经过这样处理后，再次启动MySQL，一切正常了。

2.二进制日志

二进制日志通常称为binLog，包含所有更新数据或者已经潜在更新数据的所有语句。语句以“事件”的形式保存，描述数据更改，还包含关于每个更新数据库的语句的执行时间信息。二进制日志其记着所有的DDL和DML，但不包括数据查询语句。

在配置文件里配置Log bin开启，MySQL在每个binLog名后面添加一个数字扩展名，同时创建一个binLog索引文件binLog.index，包含所有使用的binLog文件名。

binLog以binary方式存取，不能直接查看，需要用MySQL提供的mysqlbinlog工具查看。

使用方法如下所示：

```
>mysqlbinlog D: \ data \ data \ binLog.000001
```

显示如下（部分数据）：

```
1.#at 441
2.#111014 10: 51: 04 server id 1 end_Log_pos 469 Intvar
3.SET INSERT_ID=1/*! */;
4.#at 469
5.#111014 10: 51: 04 server id 1 end_Log_pos 615 Query thread_id=1 exec_tim
6.e=0 error_code=0
7.SET TIMESTAMP=1318560664/*! */;
8.INSERT INTO ' admin ' ( ' admin_username ', ' admin_password ') VALUES ( ' admin
', ' 12345
9.6 ')
10./*! */;
```

第1行，at后边的441标明当前事件出现在binLog文件中的位置（二进制中的位置，不是行数）。

第2行，行首的时间记录这个事件在服务器上产生的时间，在replicatin中，这个timestamp会被传送到Slave上。serverid是产生这条Log的服务器的server_id。end_Log_pos标明下一条Log起始位置。thread_id指明哪个线程产生了这条Log。exec_time在Master上表示这条语句执行所用的时间，在Slave上表示这条语句在Slave执行完成的时间点与这条语句在Master上执行完成时间点之间的差值。error_code表示这条语句执行产生的错误代码，0表示没有错误。

第3行，由第2行中行首的时间产生的timestamp值。

第4行，产生这条Log的Query。

binLog的主要目的是在恢复数据库时能够最大可能地保证数据完整，因为binLog包含备份后进行的所有更新，另外，其还用在MySQL的主从配置上。

3.查询日志

查询日志记录Client的所有语句，在配置文件里通过配置Log选项启用，启用方法如下：

```
Log="D: /data/Data/MySQLquery.Log"
```

查询日志也是一个文本文件，可用文本编辑器直接打开查看。由于查询日志记录数据库所有操作，对于访问频繁的系统，此种日志会造成性能影响，建议关闭或者间歇性打开。如果遇到安全问题，也可以从这个日志中进行排除。

4.慢查询日志

慢查询日志记录执行时间超过参数long_query_time（单位是秒）所设定值的SQL语句日志。通过配置Log slow_queries选项开启，如下所示：

```
#日志存放的文件的位置
Log-slow-queries="D: /data/Data/slow.Log"
#查询超过这个时间的将会被记录，单位为秒
long_query_time=2
```

MySQL慢查询日志对于跟踪有问题的查询非常有用，可以分析出当前程序里有很耗费资源的SQL语句。慢查询对MySQL的性能影响不大，建议打开。慢查询日志是一个文本文件，可用文本编辑器直接打开查看，也可以用mysqldumpslow工具进行分析。可以

对 Log 进行排序，找出最慢的或者返回记录集最大的前几条语句。使用“mysqldumpslow——help”可以查看其所有参数。

根据性能优化常识，优化执行时间最长的语句比优化执行频率高但优化效果不明显的语句获得的收益更大。因此，慢查询日志是MySQL优化的关键依据之一。

12.4 代码性能测试技术

对代码进行测试是产品在发布前必不可少的环节，需要保证产品的可用性、易用性、健壮性和安全性。这也是产品测试要达成的目标。软件测试是一个大的方向，专业性很强，在本节中，将介绍一些基本测试方法。

12.4.1 时间点测试

性能测试主要目的是测试代码的执行效率，找到瓶颈进行优化。性能测试最常用的办法就是利用时间点，计算一段代码在运行前和运行完毕后的时间差。代码一般是这样写的，如代码清单12-6所示。

代码清单12-6 简单的时间点测试

```
<? php
$btime=microtime ( TRUE );
//实际的代码
$etime=microtime ( TRUE );
echo $etime-$btime;
? >
```

如果运行的代码本身属于耗时较少的代码，只运行一次测试的结果显然不够可靠，往往是会运行多次，比如运行10000次后取平均值的办法计算平均耗时。

当开发者需要分析一个文件中多个代码块效率时，就得写上多个这样的语句，显得比较麻烦，同时粒度也不够细。

之前介绍的Xdebug具有断点调试功能，还能对程序执行的效率进行分析。要让Xdebug支持性能分析，首先应开启profile功能，即配置：

```
Xdebug.profiler_enable=on
```

配置打开trace以及trace文件的输出目录，代码如下：

```
Xdebug.auto_trace=On
Xdebug.trace_output_dir="E: /debug/phppro"
```

配置性能分析产生的分析文件所存放的目录和文件的命名规则。

```
Xdebug.profiler_output_dir="E: /debug/phppro"
Xdebug.profiler_output_name="cachegrind.out.%p"
```

配置几个基本选项后，Xdebug会在请求PHP文件时，在每个函数运行前和运行后插入一个监控点，计算每个函数的运行时间，然后将统计结果输出到一个文件名形如“cachegrind.out.2440”这样的文件中，如图12-20所示。

此文件记录了PHP的调用堆栈以及消耗的时间等信息，但是其格式对于我们来说不好理解，因此还需要一个工具帮助我们查看这种类型的文件，这就是WinCacheGrind。



图 12-20 Xdebug生成的性能数据文件

12.4.2 文件查看工具WinCacheGrind

用WinCacheGrind打开这些文件，能看到每个文件中每个函数的运行消耗，如图12-21所示。

Function	Avg Self	Avg Cum	Total Self	Total Cum
include:	50ms	228ms	95ms	228ms
include: config.php	30ms	68ms	30ms	68ms
php_mysql_query	7.8ms	7.8ms	24ms	24ms
php_date_default_timezone_set	22ms	22ms	22ms	22ms
php_session_start	22ms	22ms	22ms	22ms
php_mysql_connect	7.8ms	7.8ms	15ms	15ms

图 12-21 用WinCacheGrind查看Xdebug生成的文件

在这里，详细地给出了每条语句所消耗的时间，调用的次数，以及整个页面运行消耗的时间。从图中可以看出，页面总共耗时228ms，其中有一个比较耗时的操作竟然是include: config.php。为什么include一个文件耗时会这么多呢？下面是config.php文件的源代码：

```
<? php
header ( ' Content-type: text/html; charset=utf-8 ' );
//header ( "Cache-Control: max-age=10000000" );
define ( "DBHOST", "localhost", true );
define ( "DBUSER", "root", true );
define ( "DBNAME", "test", true );
define ( "DBPASSWORD", "123", true );
@date_default_timezone_set ( ' Asia/Chongqing ' );
$pre="tc0_";
define ( "PRE", "tc0_", true );
$conn=@mysql_connect ( DBHOST, DBUSER, DBPASSWORD ) or die ( ' #1: failed on connecting database! ' );
mysql_select_db ( DBNAME, $conn ) or die ( " #2: failed on connecting database! " );
mysql_query ( "SET NAMES ' UTF8 ' " );
$salt= ' 1FE3126660#@54qwfhayuZ6K66ZeuKOWNg+asoXzllbpkp8pJ ' ;
define ( "ROOT", $_SERVER[ ' DOCUMENT_ROOT ' ]. ' /new/ ' );
? >
```

这个文件似乎没什么特别，在WinCacheGrind中跟进看这个文件的运行情况，如图12-22所示。

Function	Avg Self	Avg Cum.	Total Self	Total Cum.	Calls
php: mysql_query	7.6ms	7.6ms	24ms	24ms	3
php: date_default_timezone_set	23ms	23ms	23ms	23ms	1
php: mysql_connect	7.6ms	7.6ms	15ms	15ms	2
php: mysql_select_db	0.5ms	0.5ms	1.1ms	1.1ms	2
php: date	-	-	-	-	2
php: header	-	-	-	-	1

图 12-22 查看代码执行细节

可以看到，比较耗时的两个操作一个是date_default_timezone_set函数，耗时23ms，另一个是mysql_query函数，耗时24sm，共调用3次。

先看mysql_query的耗时在哪些地方，双击“php: mysql_query”展开调用堆栈，看到的信息如图12-23所示。

Function	Avg Self	Avg Cum.	Total Self	Total Cum.
php: date_default_timezone_set	23ms	23ms	23ms	23ms
Sum of total self time: 63ms (27.6%)				
Sum of calls: 16				
Rank	Self	Cum.	Called by	Callstack
1	0.7ms	0.7ms	include-config.php	config.php (13)
2	0.4ms	0.4ms	Database->connect	Database.class.php (54)
3	22ms	22ms	Database->query	Database.class.php (62)

图 12-23 mysql_query函数执行细节

看来在查询商品时消耗，这应该属于正常消耗。那么date_default_timezone_set消耗23ms，正常吗？我们感觉问题应该不在这里。为了再次确认时间到底消耗在什么地方，刷新请求，Xdebug就会立即生成新的性能分析文件。在WinCacheGrind中选择“file”菜单，单击“reload”（或者使用快捷键Ctrl+R），重新加载新的profile文件，再看新的性能分析报告。

新的性能分析报告显示整个页面加载只用17ms，远远小于第一次运行时的228ms，通过分析，耗时比较多的还是MySQL_connect和database->getall这两条语句，这个结果也符合我们的直觉。经过多次刷新，这个统计结果基本趋于稳定。

上面的实验告诉我们，要分析一个文件的运行效率，仅仅进行一次明显不够，尤其是在第一次请求时，会有个“冷启动”的消耗，所以才会导致第一次请求耗时比较长。经过多次刷新后，得出的结果趋于稳定也符合我们的直觉。

所以在进行这样的性能监测和分析时，一定要多次采样，样本值才会比较稳定。除了

Win CacheGrind外，网页版的WebGrind能用于在线性能分析。只需下载WebGrind压缩包，解压后配置config.php文件中profiledir的位置，即可通过浏览器查看输出数据。

此外，Facebook开源的XHProf也可以对PHP进行性能分析，和Xdebug相比更加轻量级，并且对系统消耗更小，适合在线上使用。这里不再赘述。

12.4.3 性能测试注意事项

Xdebug的例子告诉我们，如果仅仅把一次运行结果作为性能分析的依据显然有失偏颇，且会造成很大的误差，导致错误的结论。因此在性能测试时，务必要多次测试，求取平均值。对于多次测试，最容易想到的就是在for循环中指定次数。在这样的测试中，有一些注意事项有必要提出，以保证测试的公平和准确性。

1) 如果进行对比测试，首先应该保证可比性。

也就是说待测试的两个对象应该是可以进行比较的，并且这样的比较是有意义的。同时必须做到被测试对象具有相同的运行环境，如果做不到运行环境相同，也应该是相当的。

2) 排除不相关因素。

例如，不相关的网络延迟、输出缓冲、缓存等，比如缓存会使测试结果过于乐观。

3) 过多与过少。

测试次数过少，样本量不足，容易造成误差比较大；而测试次数过多，容易造成压力下的突变，除非进行压力测试。测试次数通常介于1000~1000000之间，当然应依据不同应用合理选择。

4) 考虑热启动时间。

所有测试应该在服务器运行平稳后进行，测试代码要考虑脚本的初始化消耗，如下所示。

测试方案1:

```
//运行环境初始化
//多次循环测试A
//多次循环测试B
测试方案2:
//运行环境初始化
```

```
//多次循环测试B  
//多次循环测试A
```

这两个方案得到的结论可能不一致，尽管这种情况在PHP这种语言里不多见，但是对于某些编译比较耗时的语言来说，这一点也是需要考虑的。

5) 指标的全面性。

很多测试使用的是时间点方法，仅仅考虑被测试对象的运算/执行速度，而没有全面地考察其内存消耗、CPU占用，这是不够科学的。

6) 关于时间精度的说明。

相比较而言，Linux上测试所得到的时间精度要比Windows高，尤其是在多核的情况下，比如多核下Windows XP的计时可能会存在同步问题。另外，如果想获得更精细的程序运行时间，排除其他进程的影响，可以用Linux下的time命令。

12.5 单元测试

计算机编程中，单元测试（又称为模块测试）是针对程序模块（软件设计的最小单位）进行正确性检验的测试。

程序单元是应用的最小可测试部件。在过程化编程中，一个单元就是单个程序、函数、过程等；对于面向对象的编程，最小单元就是方法，包括基类（超类）、抽象类、或者派生类（子类）中的方法。

单元测试是开发者编写的一小段代码，用于检验代码一个很小的、很明确的功能是否正确。通常而言，一个单元测试用于判断某个特定条件（或者场景）下某个特定函数的行为。单元测试是由程序员自己完成，最终受益的也是程序员自己。可以说，程序员有责任编写功能代码，同时也就有责任为自己的代码编写单元测试。执行单元测试，就是证明这段代码的行为和期望的一致。

单元测试具有很多优点，比如可以提高代码后期的维护性。我们可以把程序设计成易于调用和可测试的，即迫使解除软件中的耦合。另外，自动化的单元测试避免代码出现回归，编写完成之后，可以随时随地快速运行测试。

12.5.1 单元测试框架PHPUnit的安装

PHPUnit是一个轻量级PHP单元测试框架，是PHP5下对JUnit3系列版本的完整移植，是xUnit测试框架家族的一员（基于Kent Beck设计，Kent Beck是业界公认的敏捷开发之父，也是JUnit的作者之一），当前最新版是3.6。

安装PHPUnit之前，要保证系统中安装了PHP的pear包。PHPUnit官方指定通过pear包管理器安装，也可以下载源码包放在指定目录安装，但是配置比较麻烦。所以在这里采用pear方式安装。

pear是“PHP Extension and Application Repository”的缩写，是一个PHP代码包的分发和管理工具，类似Linux系统的rpm包管理器，提供一些可以复用的组件和库。

在UNIX/Linux/BSD等类UNIX系统中，pear默认安装好，如果需要重新安装，可

以用如下命令：

```
$ wget http://pear.php.net/go-pear.phar
$ php go-pear.phar
```

如果是Windows系统，在命令行窗口中找到PHP的安装目录，运行go_pear.bat批处理文件即可。如果当前的PHP环境中没有go_pear.bat文件和pear目录，到<http://pear.php.net/go>

pear.phar处下载相应文件，放入自行建立的pear目录，新建go_pear.bat文件，输入如下内容：

```
@ECHO OFF
set PHP_BIN=php.exe
%PHP_BIN% -d output_buffering=0 PEAR \ go-pear.phar
Pause
```

运行go_pear.bat，按提示操作即可安装好pear包，如图12-24所示。



图 12-24 安装pear

安装pear包后，分别输入如下命令即可安装PHPUnit：

```
pear config-set auto_discover 1
pear install pear.PHPUnit.de/PHPUnit
```

如果网络正常的话，一两分钟后即可自动下载并安装成功。在当前命令行下输入

PHPUnit, 如果有输出, 则证明安装成功。

12.5.2 结合NetBeans使用PHPUnit进行单元测试

这里结合NetBeans使用PHPUnit进行单元测试。确认已经安装PHPUnit后，打开NetBeans，单击“工具”→“选项”命令，选择“PHP”→“单元测试”，填入PHPUnit脚本的路径，如图12-25所示。



图 12-25 NetBeans中设置PHPUnit路径

步骤1 新建项目命名unitest，新建Caculator类，输入如下内容，如代码清单12-7所示。

代码清单12-7 待测试代码Calculator.php

```
<? php
class Calculator {
public function add ($a,$b) {
return $a+ $b;
}
public function add2 ($a,$b) {
return $a+ $b;
}
}
? >
```

步骤2 为当前类添加包含断言的注释块，如代码清单12-8所示。

代码清单12-8 添加断言注释

```
<? php
class Calculator {
```

```
/**
 * @assert (0, 0) ==0
 * @assert (0, 1) ==1
 * @assert (1, 0) ==1
 * @assert (1, 1) ==2
 * @assert (1, 2) ==4
 */
public function add ($a,$b) {
    return $a+ $b;
}
public function add2 ($a,$b) {
    return $a+ $b;
}
}
? >
```

步骤3 在“项目”窗口中右击“Calculator.php”节点，在弹出的快捷菜单中选择“工具”→“创建PHPUnit测试”命令。请注意，可以在“源文件”节点的上下文菜单中为项目中的所有文件创建测试。

第一次创建测试时，会打开一个对话框，询问要存储测试文件的目录。本示例中新建一个tests目录，如图12-26所示。



图 12-26 在NetBeans中使用PHPUnit

IDE将在一个名为CalculatorTest.php的文件中生成框架测试类，此类中将为每个@assert标注创建一个测试。如果这一步操作失败，很有可能是PHPUnit.bat文件中PHP_BIN设置的问题，可自行修改其为绝对路径。目录结构可参考图12-27所示。

现在，既可以测试单个文件，也可以测试整个项目。要测试项目，右击项目的父节点，然后选择“测试”命令或按Alt+F6组合键。要测试Calculator.php文件，右击该文件的节点，然后选择“测试”或按Alt+F6组合键。本示例只有一个文件，且该文件中只有一个类，因此两种测试结果相同。IDE会运行测试并在“测试结果”窗口中显示结果。

运行测试，看到IDE报错，测试失败，错误提示如下：



图 12-27 目录结构示意图

```
Fatal error: Cannot redeclare CalculatorTest: testAdd2 () inE: \ www \ php \ unittest \ tests
\
CalculatorTest.php on line 80
```

经过查看PHPUnit生成的自动测试代码，我们发现CalculatorTest类中testAdd2()方法的定义重复了。这是由于被测试类中有一个add2方法，且PHPUnit按照注释中的断言先后顺序自动生成的函数名testAdd2与对add2方法生成的测试方法testAdd2重名了。只需命名Calculator.php中的add2方法即可，或者屏蔽CalculatorTest.php最后一个testAdd2的代码即可。处理后再次单击测试Calculator.php，测试结果如图12-28所示。



图 12-28 单元测试结果

如图12-28所示，5个测试单元中，4个通过，1个失败。失败的测试单元为testAdd5函数：

```
public function testAdd5 () {
    $this->assertEquals (
    4,$this->object->add ( 1, 2 )
    );
}
```

很明显， $1+2$ 的结果不等于4，因此测试失败。到这里为止，针对Caculator.php文件的单元测试即告完成。

12.5.3 PHPUnit中的断言函数

上节的例子中用到“断言”这个概念。编写代码时，总是会做出一些假设，断言就是用于在代码中捕捉这些假设，可将断言看做异常处理的一种高级形式。断言表示为一些布尔表达式，程序员相信在程序中的某个特定点该表达式值为真，可以在任何时候启用和禁用断言验证，因此在测试时启用断言而在部署时禁用断言。同样，程序投入运行后，最终用户在遇到问题时可以重新启用断言。

断言是任何xUnit框架的核心。PHP中也内置断言函数，如下面代码片段所示：

```
<? php
$array=array (1, 3, 5);
assert ( array_search (5,$array));
asser (2==3);
```

返回如下结果：

```
Fatal error: Call to undefined function asser () in E: \ www \ php \ t \ request.php on line 4
```

简单理解，断言就是假设表达式为真，如果为假，则断言失败。此外，PHPUnit还有其他30多个断言，例如：

```
public function testFailure ()
{
    $this->assertArrayHasKey ( ' baar ', array ( ' bar ' => ' baz '));
}
```

把这段代码加入CalculatorTest.php，重新运行测试，即可看到结果。从方法命名

上看，这个断言是用来判断指定的键是否存在于数组中。下面是PHPUnit中部分断言的用法。

1) 断言类className含有静态属性attributeName:

```
assertClassHasStaticAttribute ( string $ attributeName, string $ className[, string $message= ' '])
```

2) 断言迭代器对象haystack/数组haystack含有needle:

```
assertContains ( mixed $ needle, Iterator | array $ haystack[, string $message= ' '])
```

3) 断言迭代器对象haystack/数组haystack不含有needle:

```
assertNotContains ( mixed $ needle, Iterator | array $ haystack[, string $message= ' '])
```

4) 断言needle为一个类/对象haystack可访问到的属性 (public、protected和private):

```
assertAttributeContains ( mixed $ needle, Class | Object $ haystack[, string $message= ' '])
```

5) 断言needle不是一个类/对象haystack可访问到的属性:

```
assertAttributeNotContains ( mixed $ needle, Class | Object $ haystack[, string $message= ' '])
```

6) 断言字符串needle在字符串haystack中:

```
assertContains ( string $ needle, string $ haystack[, string $ message= ' ' ])
```

7) 断言对象的属性只有type类型和非含有type类型:

```
assertAttributeContainsOnly ( $ type , $ haystackAttributeName , $ haystackClassOrObject  
,$isNativeType=NULL,  
$ message= ' ' )
```

8) 断言actual为空:

```
assertEmpty ( mixed $ actual[, string $ message= ' ' ])
```

9) 断言actual非空:

```
assertNotEmpty ( mixed $ actual[, string $ message= ' ' ])
```

10) 断言对象的所有属性为空:

```
assertAttributeEmpty ( $ haystackAttributeName,$ haystackClassOrObject,$ message= ' ' )
```

11) 断言对象的所有属性非空:

```
assertAttributeNotEmpty ( $haystackAttributeName, $haystackClassOrObject, $message= ' ' )
```

12) 断言Dom节点actualNode和DOM节点expectedNode相同，checkAttributes为FALSE不断言节点属性，TRUE则断言属性。

```
assertEqualXMLStructure ( DOMNode $ expectedNode, DOMNode $ actualNode[, boolean $checkAttributes= FALSE, string $message= ' ' ])
```

13) 断言数组actual和数组expected相同：

```
assertEquals ( array $ expected, array $ actual[, string $message= ' ' ])
```

14) 断言condition的结果为false：

```
assertFalse ( bool $ condition[, string $message= ' ' ])
```

15) 断言文件actual和expected相同：

```
assertFileEquals ( string $ expected, string $ actual[, string $message= ' ' ])
```

16) 断言文件filename存在：

```
assertFileExists ( string $ filename[, string $message= ' ' ])
```

17) 断言actual比expected大:

```
assertGreaterThan ( mixed $ expected, mixed $ actual[, string $ message= ' ' ])
```

如果用于断言类的属性, 则用:

```
assertAttributeGreaterThan ( )
```

18) 断言actual大于等于expected:

```
assertGreaterThanOrEqual ( mixed $ expected, mixed $ actual[, string $ message= ' ' ])
```

19) 断言actual为expected的实例:

```
assertInstanceOf ( $ expected, $ actual[, $ message= ' ' ])
```

20) 断言variable的值为null:

```
assertNull ( mixed $ variable[, string $ message= ' ' ])
```

21) 断言字符串string符合正则表达式pattern:

```
assertRegExp ( string $ pattern, string $ string[, string $ message= ' ' ])
```

22) 断言actual和expected的类型和值相同:

```
assertSame ( mixed $ expected, mixed $ actual[, string $ message= ' ' ])
```

23) 断言对象actual和对象expected相同:

```
assertSame ( object $ expected, object $ actual[, string $ message= ' ' ])
```

注意，大部分断言都是成对出现的。有A方法，就有一个Not A方法，出于篇幅考虑，上面只列出部分常用方法，更多断言可以参考PHPUnit官方手册。

12.5.4 PHPUnit常用方法

这里介绍几个PHPUnit方法，这些方法多数也是xUnit框架中所共有的特性。

(1) setUp ()

通用初始化方法，在单元测试模块的开始部分，做一些初始化工作，如创建连接、获取bean等。在每一个测试case之前执行。

(2) setUpBeforeClass

和setUp类似，但此方法只执行一次。

(3) tearDown

拆除方法，通常用在单元测试模块的结束部分，做一些收尾工作，如关闭连接等。在每一个测试case之后执行。无论测试成功或失败，该方法都会执行。

(4) tearDownAfterClass

和tearDown类似，但此方法只执行一次。

(5) markTestSkipped

使用此方法，可以跳过或标记不完全测试，简单地跳过暂时不需要或者不满足测试条件的部分代码，继续目前的测试程序，运行特定的测试。

12.5.5 PHPUnit常用注解

注解（Annotations）是PHP源码注释块中的特殊标记，PHPUnit通过解析注释块读取其中的注解，并执行一些特定方法。在前面的例子中，已经接触过@assert注解，常用的还有@depends、@test。

1.@depends

@depends定义一种依赖关系，即一个测试方法的参数内容和是否会运行依赖于另外一个测试方法的结果。这个特性一般用于检查代码的逻辑过程，一个逻辑执行的前提是另外一个逻辑的执行结果。下面的例子来自PHPUnit官方手册，如代码清单12-9所示。

代码清单12-9 注解@depends使用示例

```
<? php
class StackTest extends PHPUnit_Framework_TestCase
{
    public function testEmpty ()
    {
        $stack=array ();
        $this->assertEmpty ($stack);
        return $stack;
    }
    /**
     *@depends testEmpty*/
    public function testPush ( array $stack )
    {
        array_push ($stack, ' foo ');
        $this->assertEquals ( ' foo ', $stack[count ($stack) -1]);
        $this->assertNotEmpty ($stack);
        return $stack;
    }
    /**
     *@depends testPush*/
    public function testPop ( array $stack )
    {
```

```
$this->assertEquals('foo', array_pop($stack));  
$this->assertEmpty($stack);  
}  
}  
? >
```

在上面测试类 StackTest 中，定义了2个依赖测试方法：testPush 依赖于 testEmpty, testPop 依赖于 testPush。在测试运行时，testEmpty 中的断言执行完毕也没有出现问题时，方法中的 return 语句会将 stack 传给依赖于它的 testPush，作为 testPush 的参数传入 testPush 中；testPush 执行完毕之后，也会将 stack 传给依赖于它的 testPop。只要断言检查没有出现异常，PHPUnit 就会根据依赖关系依次执行依赖的测试方法，直到依赖关系结束为止。如果某个测试方法依赖的方法测试没有通过，PHPUnit 会自动跳过后面的所有依赖测试。

2.@test

在 PHPUnit 中，另一个比较有用的注解是 @test。只有以 test 字符开头的方法才会被测试，比如下面的代码将不会被测试到：

```
public function haha () {  
    $this->assertEquals(1, 6);  
}
```

增加 @test 注解，标明这是需要测试的方法，该方法会被测试到：

```
/**  
 *@test  
 */  
public function haha () {  
    $this->assertEquals(1, 6);  
}
```

12.6 压力测试

在PHP的开发中，有时需要测试当前服务器所能承受的荷载，这通常就需要用到压力测试了。性能测试偏重于代码，压力测试偏重于服务器。

最常用也是最简单的压力测试工具就是Apache自带的ab工具，之前章节已经做过介绍，这里就不再复述。ab虽然使用方法很简单，但功能也弱，对HTTP请求进行压力和性能测试，除了ab外，还有一款更专业的工具JMeter。

JMeter是Apache的开源项目，使用Java编写，是一个功能强大的性能测试工具，可以对HTTP请求、FTP请求、数据库连接（使用JDBC）等进行测试，并且其HTTP测试的功能更强大，更友好地支持GET/POST定制。

12.6.1 使用JMeter压力测试HTTP

先到官方网站下载最新版本JMeter：
http://jakarta.apache.org/site/downloads/downloads_jmeter.cgi，本书使用2.5.1版本。下载后直接解压，运行bin/jmeter.bat即可。在使用JMeter前，先了解几个术语：

线程组：测试里每个任务都要线程处理，所有后来的任务必须在线程组下面创建。通过执行“测试计划”→“添加”→“线程组”命令建立线程组，在线程组面板里有几个输入栏——线程数、Ramp Up Period（in seconds）、循环次数，其中Ramp Up Period（in seconds）表示在这时间内创建完所有的线程。例如有8个线程，Ramp Up=200秒，线程的启动时间间隔为200/8=25秒，这样的好处是，一开始不会使服务器有太大的负载。

取样器（Sampler）：可以认为所有测试任务都由取样器承担，如HTTP请求。

断言：对取样器返回的请求结果给出判断，判断其是否正确。

监听器：对取样器的请求结果进行显示、并统计一些数据（吞吐量、KB/S等）。测试步骤如下：

步骤1 建立测试计划。

测试计划描述执行测试过程中JMeter的执行过程和步骤，完整的测试计划包括一个或者多个线程组（Thread Groups）、逻辑控制（Logic Controller）、实例产生控制器（Sample Generating Controllers）、监听器（Listener）、定时器（Timer）、断言（Assertions）等。打开JMeter时，已经建立一个默认的测试计划，一个JMeter应用的实例只能建立或者打开一个测试计划。

现在开始填充一个测试计划的内容，这个测试计划向一个文件发出POST请求，需要JMeter模拟50个请求者（也就是50个线程），每个请求者连续请求两次。

待测试的脚本内容如下：

```
<? php
$num=4;
if (isset($_POST['num']))$num=$_POST['num'];
$sum=0;
for ($i=0; $i<=$num; $i++) {
    $sum+=$i;
}
echo $sum;
```

步骤2 添加线程组。

右击测试计划，在弹出的快捷菜单中依次选择“添加”→“线程组”命令，按照测试计划依次填入以下参数，线程数为50，Ramp Up=0，循环次数=4，如图12-29所示。

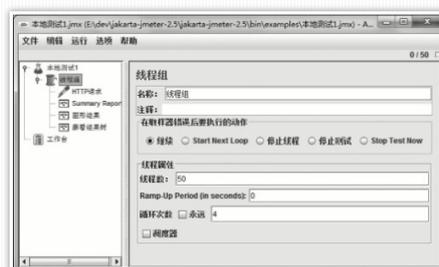


图 12-29 JMeter中设置线程组

步骤3 添加取样器。

右击线程组，在弹出的快捷菜单中依次选择“添加”→“Sample”→“HTTP请求”命令，Web服务器名称设为localhost，端口设为80，协议设为http，路径设为/t/test_1.php，方法设为POST，然后填入POST参数，这也是脚本所要接收的参数，如图12-30所示。



图 12-30 在JMeter中使用设置HTTP请求

步骤4 添加监听器。

右击线程组在弹出的快捷菜单中依次选择“添加”→“监听器”→“图形结果”（第二个）命令，然后再添加一个Summary report monitor以及“查看结果树”。然后到菜单中选择运行即可。

现在，只需要查看监听器就可以看到测试结果。测试结果给出了吞吐量、偏离等重要指标，如图12-31所示。

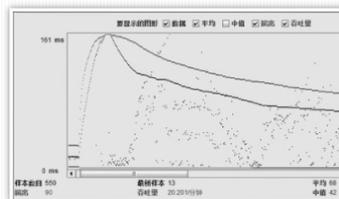


图 12-31 测试结果

到“察看结果树”里看测试结果是否符合预期，如图12-32所示。



图 12-32 测试结果

根据代码得知，其是一段求和的代码，当输入参数是100时，结果为5050，测试结果符合预期。单击“请求”选项卡，我们还能看到发送的请求头数据。可以通过增加线程数以及传入参数的手段来测试不同条件下的负载。

JMeter的主要测试组件如下：

测试计划：使用JMeter进行测试的起点，是其他JMeter测试元件的容器。

线程组：代表一定数量的并发用户，用来模拟并发用户发送请求。实际请求内容在Sampler

中定义，它被线程组包含。

monitor：负责收集测试结果，同时被告知结果显示的方式。

逻辑控制器：自定义JMeter发送请求的行为逻辑，与Sampler结合使用模拟复杂的请求序列。

断言：用来判断请求响应的结果是否如用户所期望的。可以隔离问题域，即在确保功能正确的前提下执行压力测试。这个限制对于有效测试非常有用。

配置元件：维护Sampler需要的配置信息，根据实际需要修改请求的内容。

前置处理器和后置处理器：在生成请求之前和之后完成工作。前置处理器常用来修改请求的设置，后置处理器常用来处理响应的数据。

定时器：负责定义请求之间的延迟间隔。

为了方便下次使用，可以保存当次测试计划。

注意 压力测试结果的可靠性很大程度上依赖测试软件自身的性能。JMeter由Java开发，受限于JVM，自身内存消耗比较大，同时提供的压力不是很大。为了保证测试的准确性，需要把测试环境和被测试环境分开。如果需要更大的压力，应考虑使用CurlLoader。

12.6.2 压力测试MySQL

除了对Web服务器进行测试外，数据库服务器也是系统的瓶颈之一，因此，我们还有可以针对性地对数据库进行压力测试。前面讲过MySQL的慢查询、explain分析等，这些主要是针对单条SQL语句的，其结果反映了SQL语句的优劣；而对MySQL进行的压力测试，则反映的是SQL语句本身的优劣以及MySQL服务器的性能。

1.利用mysqlslap工具测试MySQL

mysqlslap是一个MySQL官方提供的压力测试工具，通过模拟多个并发客户端访问MySQL来执行测试，使用起来非常的简单。通过mysqlslap help可以获得可用的选项。

下面我们就来看看一些比较重要的参数：

defaults file：配置文件存放位置。

create schema：所要测试的数据库。

concurrency：并发数。

engines：测试引擎，可以有多个，用分隔符隔开。

iterations：迭代的实验次数。

socket：用来连接数据库的socket文件位置。

debug info：打印内存和CPU的信息。

only print：只打印测试语句而不实际执行。

auto generate sql：自动产生测试SQL。

auto generate sql load type：测试SQL的类型，类型有mixed、update、write、key、read。

number of queries：执行的SQL总数量。

number int cols: 表内int列的数量。

number char cols: 表内char列的数量。

query=name: 使用自定义脚本执行测试, 例如可以调用自定义的一个存储过程或者SQL语句来执行测试。

看一个例子:

```
mysqlslap-a-concurrency=50—number-of-queries 300-T-hlocalhost-uroot-p123
```

此处表示自动测试, MySQL将自动创建数据库, 自动插入以及读取, 给出最终的执行结果, 其执行结果可能如下:

```
Benchmark
Running for engine myisam
Average number of seconds to run all queries: 0.087 seconds
Minimum number of seconds to run all queries: 0.087 seconds
Maximum number of seconds to run all queries: 0.087 seconds
Number of clients running queries: 200
Average number of queries per client: 1
```

也可以指定自己的SQL语句, 使用很简单, 这里不举例。

2.使用JMeter测试MySQL

使用mysqlslap工具测试数据库的性能, 命令行方式不太友好, 这里介绍用JMeter测试MySQL。JMeter使用JDBC技术, 可以测试任何支持JDBC技术的数据库, 包括MySQL、Oracle、PgSQL、MSSQL等。我们分步演示JMeter测试MySQL的方法。

步骤1 新建测试计划。

新建一个线程组，线程数设置为50，不循环，如图12-33所示。



图 12-33 在JMeter中新建数据库测试计划

步骤2 配置JDBC。

在线程组上右击，在弹出的快捷菜单中选择“添加”→“配置元件”→“JDBC Connection Configuration”命令。配置各项参数，其中Variable Name可以随便填写，此处填写MySQL。最主要的配置是Database Connection Configuration。

Database URL：JDBC格式的数据库连接，每种数据库的URL都有所区别。MySQL的URL如下，jdbc: MySQL: //localhost: 3306/test, test为数据库名称。

JDBC Driver class：JDBC驱动的名称。

Username：数据库用户名。

Password：数据库密码。

其他配置项保持默认即可，配置可参考图12-34。

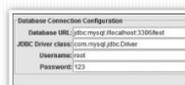


图 12-34 配置JDBC

步骤3 添加JDBC参数后，还需要添加MySQL的JDBC驱动。

因为JMeter没有自带JAR包。到MySQL官方网站找到MySQL的JDBC驱动，在“测试计划”中添加class path，如图12-35所示。



图 12-35 指定JDBC的JAR包

如果忘了这一步，将导致运行失败。

步骤4 添加JDBC请求。

需要修改的参数包括：

Variable Name：和JDBC Connection Configuration填写同样的内容，表示JDBC Connection Configuration建立一个名为MySQL的连接池，之后其他的JDBC Request都共用这个连接池。

Query：所要测试的SQL语句，在这里输入：INSERT INTO ' user ' (' user-name ', ' pwd ', ' sint ', ' bint ') VALUES (' testhaha ', ' 123 ', 6, 7);。在这里选择查询、更新、存储过程等条件，如图12-36所示。



图 12-36 设置界面

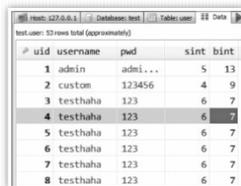
步骤5 增加两个监听器，这里凭个人喜好和需求任意添加。

运行结果如图12-37所示。线程前面有对号的表示该线程执行成功。



图 12-37 测试过程

到MySQL中看一下，数据的确添加成功，如图12-38所示。



The screenshot shows a MySQL database window with a table containing test data. The table has columns for uid, username, pwd, sint, and bint. The data is as follows:

uid	username	pwd	sint	bint
1	admin	admin...	5	13
2	custom	123456	4	9
3	testhaha	123	6	7
4	testhaha	123	6	7
5	testhaha	123	6	7
6	testhaha	123	6	7
7	testhaha	123	6	7
8	testhaha	123	6	7

图 12-38 验证测试结果

到此为止，JMeter测试MySQL的步骤完毕。

12.6.3 JMeter+Badboy组合测试

我们可能会遇到下面这种场景：模拟现实操作进行性能测试。需要模拟一个完整流程的用户操作，模拟一定量的并发请求，这样的测试结果比较接近于生产环境。

使用JMeter完成这个任务比较烦琐，因为JMeter的录制脚本过于笨拙复杂。

Badboy是用C++开发的免费软件，用于测试和开发复杂的动态应用。它提供强大的屏幕录制和回放功能，同时提供了丰富的图形结果分析功能。可以用Badboy录制脚本，将录制的脚本导出为JMeter格式的脚本，最后将该脚本导入JMeter，借助于JMeter强大的测试功能模拟大量的虚拟用户，进行复杂的性能测试。下面演示测试流程。

步骤1 安装Badboy。

Badboy的下载地址为<http://badboy.com.au/download/index>，这是一个可执行程序，下载之后双击，根据安装向导很容易安装完毕。

步骤2 用Badboy录制脚本并导出为JMeter格式的脚本。

在开始菜单中启用Badboy，单击工具栏上的红色圆饼按钮开始录制，然后在URL框中输入要测试站点的URL。随便单击几个链接，然后单击工具栏上的黑色方框按钮，停止录制。单击Play All播放录制过程，不满意可以重新录制。最后单击“File”→“Export to JMeter……”命令，将录制的脚本导出为JMeter格式脚本，如图12-39所示。



图 12-39 Badboy界面

步骤3 启用JMeter。

导入Badboy录制的脚本，并设置测试计划，然后按照之前的方法，添加线程组和监视器即可。这里不再复述。

如果对测试数据比较感兴趣，可以使用Jmeter+Sigar组合收集更多的测试结果。Sigar全名是System Information Gatherer And Reporter，中文名是系统信息收集和报表工具，其是一个跨平台开源的工具。

提示 JMeter是一个专业的测试工具，使用起来比较复杂，难点在于对各种专业名词的理解上。JMeter和LoadRunner的设计理念比较接近。其优点在于开源免费、功能强大，不比商业软件LoadRunner逊色，其同时秉承了Apache组织的一贯风格，文档丰富详细。当然，测试本身就是一个艺术，所以JMeter入门和掌握其操作有一定难度。

12.7 本章小结

本章主要讲解基本的调试和测试技术，如前端调试和PHP代码调试，以及Firebug、Fiddler、Xdebug的使用方法。掌握基本调试技能对于开发过程中排错有很大帮助，也有助于我们阅读他人的源代码。其次，讲解PHP、MySQL、Apache这三类软件的日志管理，有助于我们从日志中发现和解决问题。

本章测试环节以PHPUnit单元测试和JMeter为主，介绍了基本的测试技术。单元测试是开发中一个重要环节，可以保证产品基本功能的可用性，有利于团队开发中的协作和分工。JMeter是强大的测试工具，其能更好地模拟生产环境进行功能测试和压力测试，并且支持多种数据库和协议，是一个比较理想的测试解决方案。JMeter配合Badboy可以使测试过程变得更简单。

第13章 Hash算法与数据库实现

Hash表 (HashTable) 又称散列表, 通过把关键字Key映射到数组中的一个位置来访问记录, 以加快查找的速度。这个映射函数称为Hash函数, 存放记录的数组称为Hash表。

13.1 Hash函数

Hash函数的作用是任意长度的输入, 通过Hash算法变换成固定长度的输出, 该输出就是Hash值。这种转换是一种压缩映射, 也就是Hash值的空间通常远小于输入的空间, 不同的输入可能会散列成相同的输出, 而不可能从Hash值来唯一地确定输入值。

一个好的Hash函数应该满足以下条件: 每个关键字都可以均匀地分布到Hash表任意一个位置, 并与其他已被散列到Hash表中的关键字不发生冲突。这是Hash函数最难实现的。

13.2 Hash算法

关键字k可能是整数或者字符串，可以按照关键字的类型设计不同的Hash算法。整数关键字的Hash算法有以下几种。

13.2.1 直接取余法

直接取余法原理比较简单，直接用关键字k除以Hash表的大小m取余数，算法如下：

$$h(k) = k \bmod m$$

例如，如果Hash表的大小为 $m=12$ ，所给关键字为 $k=100$ ，则 $h(k)=4$ 。这种算法只需要一个求余操作，速度比较快。

13.2.2 乘积取整法

乘积取整法首先使用关键字 k 乘以一个常数 A ($0 < A < 1$), 并抽取出 kA 的小数部分。然后用Hash表大小 m 乘以这个值, 再取整数部分即可。算法如下:

$$h(k) = \text{floor}(m * (kA \bmod 1))$$

其中, $kA \bmod 1$ 表示 kA 的小数部分, floor 是取整操作。

当关键字是字符串的时候, 就不能使用上面的Hash算法。因为字符串是由字符组成, 所以可以把字符串所有字符的ASCII码加起来得到一个整数, 然后再按照上面的Hash算法去计算即可。算法如下:

```
function hash ($key, $m) {
    $strlen=strlen($key); $hashval=0;
    for ($i=0; $i<$strlen; $i++) {
        $hashval+=ord($key{$i});
    }
    return $hashval%$m;
}
```

虽然这是最简单的Hash算法, 而且效果也不好, 但可以描述Hash算法的基本原理。

13.2.3 经典Hash算法Times33

经过计算机科学家们多年的研究，创造了一些非常有效的Hash算法，比较有名的包括：ELFHash、APHash和Times33等。下面是经典的Times33算法：

```
unsigned int DJBHash ( char*str ) {  
    unsigned int hash=5381; while ( *str ) {  
        hash+= ( hash<<5 ) + ( *str++ );  
    }  
    return ( hash&0x7FFFFFFF );  
}
```

Times33算法思路就是不断乘以33，其效率和随机性都非常好，广泛运用于多个开源项目中，如Apache、Perl和PHP等。

13.3 Hash表

Hash表是计算机科学中最为重要的数据结构之一。

13.3.1 Hash表结构

Hash表的时间复杂度为 $O(1)$ ，Hash表结构可以用图13-1展示。

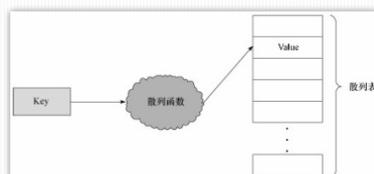


图 13-1 Hash表结构

从图13-1中可以看出，要构造一个Hash表必须创建一个足够大的数组用于存放数据，另外还需要一个Hash函数把关键字Key映射到数组的某个位置。

Hash表的实现步骤如下：

- 1) 创建一个固定大小的数组用于存放数据。
- 2) 设计Hash函数。
- 3) 通过Hash函数把关键字映射到数组的某个位置，并在此位置上进行数据存取。

下面使用PHP实现一个Hash表。

13.3.2 使用PHP实现Hash表

首先创建一个HashTable类，有两个属性buckets和size。buckets用于存储数据的数组，size用于记录buckets数组大小。然后在构造函数中为buckets数组分配内存。代码如下：

```
<? php
class HashTable {
private $ buckets;
private $ size=10;
public function construct ( ) {
    $ this-> buckets=new SplFixedArray ( $ this-> size );
}
}
```

在构造函数中，为buckets数组分配了一个大小为10的数组。在这里使用了SPL扩展的SplFixedArray数组而不是一般的数组（array），这是因为SplFixedArray数组更接近于C语言的数组，而且效率更高。在创建SplFixedArray数组时需要为其提供一个初始化的大小。

注意 要使用SplFixedArray数组必须安装并开启SPL扩展，PHP5.3或之后版本默认开启此扩展。如果没有开启SPL扩展，可以使用一般的数组代替SplFixedArray数组。

接着为Hash表制定一个Hash函数，为了简单起见，这里使用最简单的Hash算法，也就是上面提到的把字符串的所有字符加起来再取余，代码如下：

```
private function hashfunc ( $key ) {
    $ strlen=strlen ( $key ); $ hashval=0;
    for ( $ i=0; $ i< $ strlen; $ i++ ) {
        $ hashval+=ord ( $key { $ i } );
    }
    return $ hashval% $ this-> size;
}
```

有了Hash函数，就可以实现插入和查找方法。插入数据时，先通过Hash函数计算关键字所在Hash表的位置，然后把数据保存到此位置即可。代码如下：

```
public function insert ( $key, $val ) {  
    $index= $this->hashfunc ( $key );  
    $this->buckets[ $index]= $val;  
}
```

查找数据方法与插入数据方法相似，先通过Hash函数计算关键字所在Hash表的位置，然后返回此位置的数据即可。代码如下：

```
public function find ( $key ) {  
    $index= $this->hashfunc ( $key );  
    return $this->buckets[ $index];  
}
```

至此，一个简单的Hash表编写完成。下面测试这个Hash表，如代码清单13-1所示：

代码清单13-1 测试代码

```
<? php  
$ht=new HashTable ();  
$ht->insert ( ' key1 ', ' value1 '); /*插入key1=>value1*/  
$ht->insert ( ' key2 ', ' value2 '); /*插入key2=>value2*/  
echo $ht->find ( ' key1 '); /*查找key1对应的数据*/  
echo $ht->find ( ' key2 '); /*查找key2对应的数据*/  
? >
```

输出结果如图13-2所示。



图 13-2 Hash表的测试结果

13.3.3 Hash表冲突

从代码清单13-1的测试结果来看，Hash表好像能够正常工作。但使用代码清单13-2进行测试时就会发现问题。

代码清单13-2 Hash表冲突测试代码

```
<? php
$ht=new HashTable ();
$ht->insert ( ' key1 ', ' value1 ');
$ht->insert ( ' key12 ', ' value12 ');
echo $ht->find ( ' key1 ');
echo $ht->find ( ' key12 ');
? >
```

输出结果如图13-3所示。



图 13-3 Hash表冲突测试结果

代码清单13-2中，存储“key1/value1”和“key12/value12”这两对数据，按道理，获取数据的时候应该得到“value1”和“value12”，但为什么得到的都是“value12”呢？

这个问题称为Hash表的冲突。冲突的原因是：不同的关键字通过Hash函数计算出来的Hash值相同。通过打印“key1”和“key12”的Hash值，可以发现它们都为8。也就是说“value1”和“value12”同时被存储在Hash表的第9个位置上（索引从0开始，索引8表示第9个位置），所以“value1”的值被“value12”复盖了。

解决冲突常用的方法有：开放定址法和拉链法。因为拉链法容易理解，而且实现起来比较简单，所以下面以拉链法来解决冲突问题。

13.3.4 拉链法解决冲突

拉链法解决冲突的做法是：将所有相同Hash值的关键字节节点链接在同一个链表中，如图13-4所示。

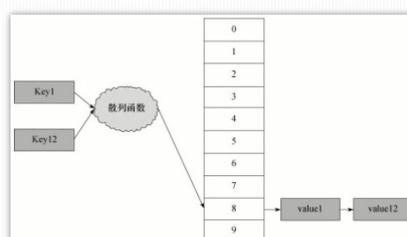


图 13-4 拉链法

从图13-4中可以看出，拉链法把相同Hash值的关键字节节点以一个链表连接起来，那么在查找元素时必须遍历这条链表，比较链表中每个元素的关键字与查找的关键字是否相等，如果相等就是我们要查找的元素。

因为节点需要保存关键字（Key）和数据（Value），同时还要记录具有相同Hash值的节点。所以创建一个HashNode类存储这些信息，HashNode结构如下：

```
<? php
class HashNode {
public $key; public $value;
public $nextNode;
public function construct ( $key, $value, $nextNode=NULL ) {
    $this->key= $key;
    $this->value= $value;
    $this->nextNode= $nextNode;
}}
? >
```

HashNode有3个属性：key、value和nextNode。key是节点的关键字，value是节点的值，而nextNode是指向具有相同Hash值节点的指针。现在把插入方法修改为

如下形式:

```
public function insert ($key, $value) {
    $index = $this->hashfunc($key);
    /*新建一个节点*/
    if (isset($this->buckets[$index])) {
        $newNode = new HashNode($key, $value, $this->buckets[$index]);
    } else {
        $newNode = new HashNode($key, $value, NULL);
    }
    $this->buckets[$index] = $newNode; /*保存新节点*/
}
```

修改后的插入算法流程如下:

- 1) 使用Hash函数计算关键字的Hash值, 通过Hash值定位到Hash表的指定位置。
- 2) 如果此位置已经被其他节点占用, 把新节点的nextNode指向此节点, 否则把新节点的nextNode设置为NULL。
- 3) 把新节点保存到Hash表的当前位置。

经过这3个步骤, 相同Hash值的节点会被连接在同一个链表。

查找算法相应修改为如下形式:

```
public function find ($key) {
    $index = $this->hashfunc($key);
    $current = $this->buckets[$index];
    while (isset($current)) { /*遍历当前链表*/
        if ($current->key == $key) { /*比较当前节点的关键字*/
            return $current->value; /*查找成功*/
        }
        $current = $current->nextNode; /*比较下一个节点*/
    }
}
```

```
}  
return NULL; /*查找失败*/  
}
```

修改后的查找算法流程如下：

- 1) 使用Hash函数计算关键字的Hash值，通过Hash值定位到Hash表的指定位置。
- 2) 遍历当前链表，比较链表中每个节点的关键字与查找关键字是否相等。如果相等，查找成功。
- 3) 如果整个链表都没有要查找的关键字，查找失败。

测试代码如下：

```
<? php  
$ht=new HashTable ();  
$ht->insert ( ' key1 ', ' value1 ');  
$ht->insert ( ' key12 ', ' value12 ');  
echo $ht->find ( ' key1 ');  
echo $ht->find ( ' key12 ');  
? >
```

输出结果如图13-5所示。



图 13-5 使用拉链法解决冲突

从图13-5可以看出，使用拉链法解决了冲突问题。

13.4 一个小型数据库的实现

本节开发一个Key Value形式的小型数据库来巩固Hash算法。

企业级数据库通常使用B+Tree，或者动态Hash技术作为索引算法。这些算法的检索速度非常快，但同时也非常复杂和烦琐。本节的目的并不是要创造专业的数据库，所以我们将使用较为简单的静态Hash算法作为索引算法。

在编写代码的过程中，要用到pack、unpack这两个函数。这两个函数并不常用，要了解这两个函数的使用方法，首先看PHP与二进制的关系。

一般情况下，使用PHP向文件写入一个整数时，PHP会先把整数转换成字符串，然后再进行写入操作。如代码清单13-3：

代码清单13-3 向文件写入整数

```
<? php
$fp=fopen("data.dat", "wb");
fwrite($fp, 12, 4);
fclose($fp);
? >
```

在上面的程序中，我们期望把整数12的二进制代码写入文件data.dat中，但实际上写入的却是字符串"12"的二进制代码（十六进制为31、32），如图13-6所示。



图 13-6 data.dat文件的二进制代码

所以如果想用PHP向文件写入一个整数的二进制代码时，就不能使用上面的方法。为了解决这个问题，PHP提供了两个打包和解包二进制代码的函数，即pack和unpack函数。

13.4.1 pack函数的用法

pack函数的作用是把数据装入一个二进制字符串，语法如下：

```
string pack ( string $format[, mixed $args[, mixed $……]] );
```

参数说明如表13-1所示。

参 数	说 明
format	必需。规定在包装数据时所使用的格式
args	可选。规定被包装的一个或多个参数

format参数字符格式含义如表13-2所示。

格 式	说 明
a	一个填充空的字节串
A	一个填充空格的字节串
b	一个位串，在每个字节里位的顺序都是升序
B	一个位串，在每个字节里位的顺序都是降序
c	一个有符号 char (8 位整数) 值
C	一个无符号 char (8 位整数) 值
d	本机格式的双精度浮点数
f	本机格式的单精度浮点数
k	十六进制串，低 4 位在前
H	十六进制串，高 4 位在前
i	有符号整数，本机格式
I	无符号整数，本机格式
l	有符号长整型，总是 32 位
L	无符号长整型，总是 32 位
n	16 位短整型，“网络”字节序 (大头在前)
N	32 位短整型，“网络”字节序 (大头在前)
p	指向空结尾的字符串的指针
P	指向定长字符串的指针
q	有符号 4 倍 (64 位整数) 值

格 式	说 明
Q	无符号 4 倍 (64 位整数) 值
s	有符号短整型，总是 16 位
S	无符号短整型，总是 16 位
u	无编码的字符串
U	Unicode 字符串
v	“VAX”字节序 (小头在前) 的 16 位短整型
V	“VAX”字节序 (小头在前) 的 32 位短整型
w	BEH 压缩的整数
x	空字节 (向前忽略一个字节)
X	备份 1 字节
Z	空结束的 (和空填充的) 字符串
@	用空字节填充绝对位置

下面使用pack函数修改代码清单13-3，如代码清单13-4所示。

代码清单13-4 pack函数使用示例

```
<? php
```

```
$fp=fopen ("data.dat", "wb");  
$bin=pack ("L", 12);  
fwrite ($fp,$bin, 4);  
fclose ($fp);  
? >
```

现在使用十六进制查看器查看由程序生成的data.dat文件的二进制代码，如图13-7所示。



图 13-7 data.dat文件的二进制代码

从图13-7中可以看出，现在写入的是整数12的二进制代码（十六进制为0C000000）。pack（）函数的format参数指定以什么样的方式来打包数据。在代码清单13-4中的“L”，代表了以一个无符号长整型来打包数据，所以最后写入的是类型为长整型的整数12，占4字节。

如果想写入3个长整型的整数，可以使用代码清单13-5所示代码。

代码清单13-5 写入3个长整型整数

```
<? php  
$fp=fopen ("data.dat", "wb");  
$bin=pack ("LLL", 12, 12, 12);  
fwrite ($fp,$bin, 12);  
fclose ($fp);  
? >
```

二进制代码如图13-8所示。

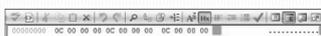


图 13-8 data.dat文件的二进制代码

可以把代码清单13-5中pack函数的format参数改为“L3”，表示有3个长整型类型

的整数需要打包。

13.4.2 unpack函数的用法

下面继续看pack函数的逆向操作函数unpack。

unpack函数的作用是从二进制字符串对数据进行解包，语法如下：

```
array unpack ( string $format, string $data );
```

参数说明如表13-3所示。

参 数	描 述
format	必需。规定在包装数据时所使用的格式
data	可选。规定要解包的二进制数据

unpack函数的format参数跟pack函数的format参数相同，详细说明可以参考表13-1。unpack函数把二进制的data参数解包成一个数组。unpack函数的使用方法如代码清单13-6所示。

代码清单13-6 unpack函数使用示例

```
<? php
$fp=fopen ("data.dat", "wb");
$bin=pack ("L", 12);
fwrite ($fp,$bin, 4);
fclose ($fp);
$fp=fopen ("data.dat", "rb");
$bin=fread ($fp, 4);
$pack=unpack ("L",$bin);
fclose ($fp);
print_r ($pack);
? >
```

以上代码的输出结果如图13-9所示。



图 13-9 代码清单13-6的输出结果

13.4.3 索引文件和数据文件

我们使用两个文件存储信息：一个索引文件和一个数据文件。索引文件包含索引值（即键）和指向数据文件中对应数据记录的指针（即偏移量）。有许多技术可以用来组织索引文件，以提高按键查询的速度和效率。本例使用固定大小的Hash表组织索引文件，采用拉链法处理Hash冲突。索引文件以idx作为后缀，数据文件以dat为后缀。

这里指定键不能超过128个字符，并且以NULL（也就是二进制值为0）作为结尾。在存储数据时，要求一个键只能对应一条记录，也就是说键是唯一的。保证键的唯一性可以提高查询效率。

图13-10所示是数据库实现的基本结构。索引文件由三部分组成：空闲链表指针，Hash表和索引记录。图中所有的指针字段实际存储的是文件的偏移量，用一个int类型（4字节）的整数来存储。

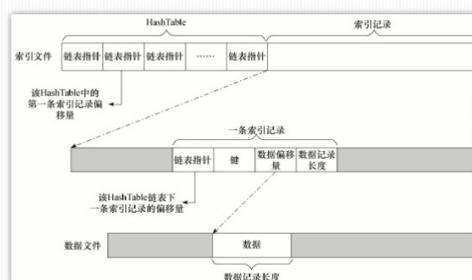


图 13-10 索引文件和数据文件结构

在图13-10中，键的大小固定为128字节，所以存储数据时键的大小不能超过128字节，否则程序会出错。

给定一个键要在数据库中寻找一条记录时，程序根据该键计算出Hash值，由此Hash值可以确定Hash表中的一条Hash链表（如果确定链表指针字段为0，表示该链表为空链表）。沿着这条Hash链表可以找到所有具有该Hash值的索引记录。当遇到一条索引记录的链表指针字段为0时，表示到达了此Hash链表的末尾。查找过程如图13-11所示。

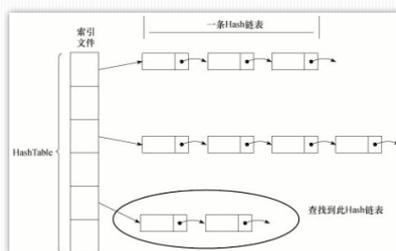


图 13-11 查找记录过程

13.4.4 数据库接口方法

以面向对象的方式编写这个数据库程序，先来看这个数据库对象有哪些方法，下一节再讨论其实际的实现过程。

1) 打开关闭数据库:

```
int db: open ( string $pathname, int $oflag, ...../*int mode*/);  
void db: close ();
```

如果 open 方法成功返回，建立两个文件：pathname.idx（索引文件）和 pathname.dat（数据文件）。

open 方法的第二个参数 oflag 指定这些文件的打开模式（只读、读/写、如果文件不存在则创建等）。如果需要创建新文件，mode 将作为第三个参数传递给 open 方法（文件访问权限）。

不再使用数据库时，调用 close 方法关闭数据库。close 方法将关闭索引文件和数据文件。

2) 插入数据到数据库:

```
int db: insert ( string $key, string $data, int flag );
```

当向数据库插入一条新记录时，必须指明此记录的键，以及与此键相联系的数据。如果此记录存储的是学生成绩，那么键就可以是学生的名字，数据就是此学生的成绩。我们的实现要求每条记录的键必须是唯一的。

flag 参数只能是 DB_INSERT（添加一条新记录），DB_REPLACE（替换一条已有的记录）或者 DB_STORE（添加一条新记录或者替换一条已有记录）。这三个常量是

我们自定义的。3) 取得一条记录:

```
string db: fetch ( string $key );
```

通过提供参数key取得一条记录。如果找到记录，返回此记录的数据；否则返回false。

4) 删除一条记录:

```
int db: delete ( string $key );
```

通过提供参数key删除指定的一条记录。如果此记录存在，删除并返回true；否则就返回false。

上述这4个方法就是数据库的主要功能。接下来开始实现这些方法。

13.4.5 源代码解析

本节对编写的数据库类源代码进行了解说。在解说源码时会为源码加上行号，这样讨论更加方便和清晰。首先看一些常量的定义：

```
<? php
1.define ( ' DB_INSERT ', 1);
2.define ( ' DB_REPLACE ', 2);
3.define ( ' DB_STORE ', 3);
4.define ( ' DB_BUCKET_SIZE ', 262144);
5.define ( ' DB_KEY_SIZE ', 128);
6.define ( ' DB_INDEX_SIZE ', DB_KEY_SIZE+12);
7.define ( ' DB_KEY_EXISTS ', 1);
8.define ( ' DB_FAILURE ', -1);
9.define ( ' DB_SUCCESS ', 0);
```

代码第1~3行定义了传递给insert方法的合法标志，前面已经介绍过这3个标志的作用。

代码第4~6行Hash表的桶大小由DB_BUCKET_SIZE指定，桶中有262144个元素指针，每个指针是一个int类型的整数，存储Hash链表的第一个元素的文件偏移量。

DB_KEY_SIZE常量指定键的长度，这里定义了128字节，如果需要更多的键，可以把DB_KEY_SIZE设置为更多的值。DB_INDEX_SIZE常量指定一条索引记录的长度。

代码第7~9行定义了3个返回信息常量，当方法调用成功时返回DB_SUCCESS，调用失败时返回DB_FAILURE。而在调用insert方法时，如果键重复就会返回DB_KEY_EXISTS。

下面继续讲解数据库类的成员变量：

```
10.class DB {
11.private $idx_fp;
12.private $dat_fp;
13.private $closed;
```

代码第10~13行为我们编写的数据库类名为DB。其包含3个私有的成员变量：`idx_fp`保存索引文件的句柄，`dat_fp`保存数据文件的句柄，`closed`记录数据库是否已经关闭。

```
14.public function open ($pathname) {
15. $idx_path= $pathname. '.idx ';
16. $dat_path= $pathname. '.dat ';
17.if (! file_exists ($idx_path)) {
18. $init=true;
19. $mode="w+b";
20. } else {
21. $init=false;
22. $mode="r+b";
23. }
```

代码第14~16行中`open`方法用于打开一个数据库，参数`pathname`就是数据库名字。变量`idx_path`保存的是索引文件的路径，而变量`dat_path`保存的是数据文件的路径。

代码第17~23行中程序判断索引文件是否存在。索引文件是否存在直接影响到我们的初始化工作。`mode`变量就是保存以什么模式打开索引文件：`w+b`模式以读写的方式打开文件，若文件不存在则创建此文件；`r+b`模式以读写的方式打开文件，但是此文件必须存在。变量`init`标志是否需要初始化索引文件。

```
24. $this->idx_fp=fopen ($idx_path,$mode);
25.if (! $this->idx_fp) {
26.return DB_FAILURE;
```

```
27. }
28. if ($init) {
29.   $elem=pack('L', 0x00000000);
30.   for ($i=0; $i<DB_BUCKET_SIZE; $i++) {
31.     fwrite($this->idx_fp,$elem, 4);
32.   }
33. }
34. $this->dat_fp=fopen($dat_path,$mode);
35. if (! $this->dat_fp) {
36.   return DB_FAILURE;
37. }
38. return DB_SUCCESS;
39. }
```

代码第24~27行，程序使用上面指定的打开模式打开索引文件。索引文件存在时，程序以r+b模式打开索引文件，否则以w+b模式打开索引文件。

代码第28~33行，因为索引文件在刚创建的时候还没有索引块（也就是图12-5的Hash表），程序首先要判断是否需要初始化索引文件，如果需要初始化，就把索引块写入到索引文件中。

在本例中，把262144个数值为0的长整型数字（占4字节）写入文件中，共占1MB空间。所以在没有任何数据插入到数据库的情况下，索引文件也占1MB空间。

代码第34~39行，打开索引文件之后，打开数据文件。打开模式的方法跟打开索引文件的方法相同。

```
40. private function _hash($string) {
41.   $string=substr(md5($string), 0, 8);
42.   $hash=0;
43.   for ($i=0; $i<8; $i++) {
44.     $hash+=33*$hash+ord($string{$i});
45.   }
46.   return $hash&0x7FFFFFFF;
```

```
47. }
```

代码第40~47行，`_hash`方法根据给定的字符串计算Hash值。

此方法先通过MD5函数把字符串处理成一个32个字符的字符串。取前8个字符作为计算串，利用Times33算法把它处理成一个整数并返回。Times33算法的优点是分布比较均匀，而且速度非常快。

```
48. public function fetch ($key) {
49.     $offset= ($this->_hash($key)%DB_BUCKET_SIZE)*4;
50.     fseek($this->idx_fp,$offset,SEEK_SET);
51.     $pos=unpack('L',fread($this->idx_fp,4));
52.     $pos=$pos[1];
```

代码第48~49行，`fetch()`方法根据给定的键从数据库中查询到指定的一条记录。

此方法首先通过`_hash`方法计算键的Hash值。通过此Hash值计算出记录所在的Hash链的文件偏移量。计算方法是：Hash值除以Hash桶大小再乘以4。乘以4是因为每个链表指针的大小为4字节。

代码第50~52行，通过`fseek`函数把文件指针移动到目标位置，并通过`fread`函数读取4字节，这4字节就是目标Hash链表的文件偏移量（见图12-6）。因为我们是二进制形式把链表指针写入到索引文件中的，所以我们需要通过`unpack`把它解包成数字。这样我们就找到了目标Hash链表在索引文件中的偏移量。

```
53. $found=false;
54. while ($pos) {
55.     fseek($this->idx_fp,$pos,SEEK_SET);
56.     $block=fread($this->idx_fp,DB_INDEX_SIZE);
57.     $cpkey=substr($block,4,DB_KEY_SIZE);
58.     if (! strcmp($key,$cpkey,strlen($key))) {
59.         $dataoff=unpack('L',substr($block,DB_KEY_SIZE+4,4));
```

```
60. $dataoff= $dataoff[1];
61. $datalen=unpack('L', substr($block, DB_KEY_SIZE+8, 4));
62. $datalen= $datalen[1];
63. $found=true;
64. break;
65. }
66. $pos=unpack('L', substr($block, 0, 4));
67. $pos= $pos[1];
68. }
69. if (! $found) {
70. return NULL;
71. }
72. fseek($this->dat_fp, $dataoff, SEEK_SET);
73. $data=fread($this->dat_fp, $datalen);
74. return $data;
75. }
```

代码第53~57行，使用found变量来标志是否找到指定的记录。使用一个while循环遍历记录所在的Hash链表。

while循环的结束条件有两个：一个是Hash链表遍历完毕，另外一个找到指定的记录。

当索引记录偏移量（pos变量）不为0时说明Hash链表还没有遍历完毕，需要继续遍历Hash链表。通过fseek（）函数把索引文件指针移动到pos位置，再通过fread（）函数读取一条索引记录大小的数据进行分析。

代码第58~68行，比较当前索引记录的key与查找的key，如果相等表示已经找到了。可以从当前索引记录中读取数据在数据文件中的偏移量（dataoff）和长度（datalen），并把found设置为true表示已经找到了数据记录，退出while循环。否则把pos设置为下一条索引记录的偏移量。

代码第69~71行，如果没有找到指定的记录，返回NULL。

代码第72~75行，如果找到记录，就可以从数据文件中读取数据。

读取数据的过程如下：首先使用fseek（）函数把数据文件的文件指针定位到我们需要的数据记录上，然后使用fread（）函数读取指定长度数据，并且以此数据记录作为返回值。

```
76.public function insert ($key,$data) {
77.$offset= ($this->_hash ($key) %DB_BUCKET_SIZE) *4;
78.$idxoff=fstat ($this->idx_fp);
79.$idxoff=intval ($idxoff[ 'size' ]);
80.$datoff=fstat ($this->dat_fp);
81.$datoff=intval ($datoff[ 'size' ]);
82.$keylen=strlen ($key);
83.if ($keylen>DB_KEY_SIZE) {
84.return DB_FAILURE;
85.}
```

代码第76~81行，insert（）方法把一条记录插入到数据库中。key参数是记录的键，data参数是记录的数据。insert（）方法会把key存放到索引文件中，把data存放到数据文件中。

首先通过key计算出索引记录所在的Hash链表的文件偏移量offset。通过fstat（）函数获取到索引文件和数据文件的下一个空闲空间的文件偏移量idxoff和datoff。

代码第82~85行，比较要插入的索引记录的key长度是否大于限定的最大长度，如果是直接返回插入失败。

```
86.$block=pack ('L', 0x00000000);
87.$block.=$key;
88.$space=DB_KEY_SIZE-$keylen;
89.for ($i=0; $i<$space; $i++) {
90.$block.=pack ('C', 0x00);
91.}
92.$block.=pack ('L',$datoff);
93.$block.=pack ('L', strlen ($data));
```

代码第86~93行，构造一个索引记录块block。

从图13-10中可以知道，一条索引记录有以下几个域：指向下一条索引记录的指针、键、数据记录所在数据文件的偏移量和数据记录的长度。所以要构造一条索引记录，必须把这些域填充完整。

第86行把“指向下一条索引记录的指针”域填充为一个类型为长整型的数字0，表示已经没有下一条索引记录了，也就是说当前索引记录是Hash链的最后一个记录。

在第87行，把“键”域填充为我们要插入的key。要注意的是，如果key没有达到我定的最大长度，就使用字符0作为填充，直到达到键的最大长度为止。

第92和93行，把“数据记录所在数据文件的偏移量”域和“数据记录的长度”域分别填充为数据文件的下一个空闲空间的文件偏移量（datoff）和要插入的数据记录的长度。

```
94.fseek ($this->idx_fp,$offset,SEEK_SET);
95.$pos=unpack('L',fread($this->idx_fp,4));
96.$pos=$pos[1];
97.if($pos==0){
98.fseek($this->idx_fp,$offset,SEEK_SET);
99.fwrite($this->idx_fp,pack('L',$idxoff),4);
100.fseek($this->idx_fp,0,SEEK_END);
101.fwrite($this->idx_fp,$block,DB_INDEX_SIZE);
102.fseek($this->dat_fp,0,SEEK_END);
103.fwrite($this->dat_fp,$data,strlen($data));
104.return DB_SUCCESS;
105.}
106.$found=false;
107.while($pos){
108.fseek($this->idx_fp,$pos,SEEK_SET);
109.$tmp_block=fread($this->idx_fp,DB_INDEX_SIZE);
110.$cpkey=substr($tmp_block,4,DB_KEY_SIZE);
111.if(!strcmp($key,$cpkey,strlen($key))){
```

```
112.$dataoff=unpack('L', substr($tmp_block, DB_KEY_SIZE+4, 4));
113.$dataoff=$dataoff[1];
114.$datalen=unpack('L', substr($tmp_block, DB_KEY_SIZE+8, 4));
115.$datalen=$datalen[1];
116.$found=true;
117.break;
118.}
119.$prev=$pos;
120.$pos=unpack('L', substr($tmp_block, 0, 4));
121.$pos=$pos[1];
122.}
123.if($found){
124.return DB_KEY_EXISTS;
125.}
126.fseek($this->idx_fp,$prev,SEEK_SET);
127.fwrite($this->idx_fp,pack('L',$idxoff),4);
128.fseek($this->idx_fp,0,SEEK_END);
129.fwrite($this->idx_fp,$block,DB_INDEX_SIZE);
130.fseek($this->dat_fp,0,SEEK_END);
131.fwrite($this->dat_fp,$data,strlen($data));
132.return DB_SUCCESS;
133.}
```

代码第94~96行，把索引文件的文件偏移量移动到索引记录所在的Hash链表位置上。读取Hash链表的开始索引记录的文件偏移量pos。

代码第97~105行，如果Hash链表的开始索引记录的文件偏移量(pos)等于0，表示此Hash链表为空，把新的索引记录插入到索引文件的空闲位置上，并且修改Hash链表的开始索引记录的文件偏移量为新插入的索引记录的位置。

代码第106~133行，如果Hash链表的开始索引记录的文件偏移量(pos)不等于0，则表示此Hash链表不为空。那么遍历此Hash链表，查找Hash链表中是否已经存在要插入的key，如果已经存在，则说明插入操作失败。否则，把新的索引记录插入到索引文件的空闲位置上，并且把Hash链表最后一个索引记录节点的next指针(指向下一个索引记录节点的文件偏移量)修改为新插入的索引记录的位置，使其成为Hash链表的最后一个节点。

```
134.public function delete ($key) {
135.$offset= ($this->_hash ($key) %DB_BUCKET_SIZE) *4;
136.fseek ($this->idx_fp,$offset,SEEK_SET);
137.$head=unpack ('L', fread ($this->idx_fp, 4));
138.$head= $head[1];
139.$curr= $head;
140.$prev=0;
```

代码第134~136行，delete () 方法从数据库中删除一条记录。key参数是要删除记录的键。首先通过key计算索引记录所在的Hash链表的文件偏移量offset。通过fseek () 函数把索引文件的文件偏移量移动到offset处。

代码第137~140行，通过fread () 函数和unpack () 函数读取Hash链表头节点的文件偏移量，并保存到head变量中。把当前节点curr设置为head，前一个节点prev设置为NULL。

```
141.while ($curr) {
142.fseek ($this->idx_fp,$curr,SEEK_SET);
143.$block=fread ($this->idx_fp,DB_INDEX_SIZE);
144.$next =unpack ('L', substr ($block, 0, 4));
145.$next = $next[1];
146.$cpkey=substr ($block, 4, DB_KEY_SIZE);
147.if (! strcmp ($key,$cpkey, strlen ($key))) {
148.$found=true;
149.break;
150.}
151.$prev= $curr;
152.$curr= $next; 153.}
```

代码第141~145行，进行while循环，直到当前节点curr为空。

把索引文件的文件指针移动到当前节点处，通过fread () 函数读取一个索引记录

block。再从索引记录中读取下一个节点的文件偏移量next。

代码第146~150行，比较当前索引记录的key是否与我们要删除记录的key相等，如果相等就表明此记录是要删除的记录，就把found设置为true表示已经找到此索引记录，并退出循环。

代码第151~153行，否则把上一个节点的文件偏移量prev设置为当前节点的文件偏移量curr，再把当前节点的文件偏移量curr设置为下一个节点的文件偏移量next，继续循环下去。

```
154.if (! $found) {
155.return DB_FAILURE;
156.}
157.if ($prev==0) {
158.fseek ($this->idx_fp,$offset,SEEK_SET);
159.fwrite ($this->idx_fp,pack('L',$next),4);
160.} else {
161.fseek ($this->idx_fp,$prev,SEEK_SET);
162.fwrite ($this->idx_fp,pack('L',$next),4);
163.}
164.return DB_SUCCESS;
165.}
```

代码第154~156行，如果没有找到要删除的记录，就返回DB_FAILURE，表示删除失败。

代码第157~165行，则表示找到要删除的记录。删除过程要分两种情况处理：

1) 要删除节点的前一个节点prev为空，表示要删除的节点是Hash链表的头节点，那么需要把Hash链表的头节点修改成要删除节点的下一个节点next。

2) 要删除节点的前一个节点prev不为空，那么只需把其指向下一个节点的指针修改成要删除节点的下一个节点next即可。

```
166.public function close () {  
167.if (! $this->closed) {  
168.fclose ($this->idx_fp);  
169.fclose ($this->dat_fp);  
170.$this->closed=true;  
171.}  
172.}
```

代码第166~172行，最后是关闭数据库的方法close ()。

close () 方法首先判断数据库是否已经关闭，如果还没有关闭，就使用fclose () 函数关闭索引文件和数据文件。接着把closed设置为true，表示数据库已经关闭，通过这种方法防止多次关闭文件导致错误发生。

13.4.6 测试代码

至此，已经把小型数据库的原理和实现讲解完毕，下面测试效率如何。1) 插入记录效率测试：

```
<? php
include ( ' db.class.php ');
$db=new DB ();
$db->open ( ' dbtest ');
$start_time=explode ( ' ', microtime ( ));
$start_time= $ start_time[0]+ $ start_time[1];
for ( $ i=0; $ i<10000; $ i++) {
    $db->insert ( "key". $ i, "value". $ i);
}
$end_time=explode ( ' ', microtime ( ));
$end_time= $ end_time[0]+ $ end_time[1];
$db->close ( );
echo ' process time in ' . ( $ end_time- $ start_time ) . ' seconds ' ;
? >
```

在上面的代码中插入10000条记录，测试效果如图13-12所示。



图 13-12 插入记录效率测试结果

2) 查询记录效率测试：

```
<? php
include ( ' db.class.php ');
$db=new DB ();
$db->open ( ' dbtest ');
$start_time=explode ( ' ', microtime ( ));
$start_time= $ start_time[0]+ $ start_time[1];
for ( $ i=0; $ i<10000; $ i++) {
```

```
$db->fetch ("key $i");  
}  
$end_time=explode (' ', microtime ());  
$end_time=$end_time[0]+$end_time[1];  
$db->close ();  
echo ' process time in ' . ($end_time-$start_time) . ' seconds ' ;  
? >
```

在上面的代码中查询了10000条记录，测试效果如图13-13所示。



图 13-13 查询记录测试结果

从上面的测试结果来看，效率还是不错的。把这个小型数据库作为本地缓存来使用，可以减少数据库的查询次数，达到加速Web应用的效果。

13.5 本章小结

本章主要介绍了Hash算法的原理和Hash表的实现，并以实现一个小型数据库为例来巩固Hash算法的相关知识。Hash算法的应用离我们不远，MD5散列函数、Mem-cache内存缓存系统、简单的数据库分表设计都存在Hash算法应用的影子。

第14章 PHP编码规范

为了提高工作效率，保证开发的有效性和合理性，并最大程度提高程序代码的可读性和可重复利用性，提高沟通效率，需要一份代码编写规范。本章介绍的规范参考了Zend与PEAR编码规范，并进行了适当的简化和调整，尽量保持简约易懂。通过代码规范，让大家养成良好的代码编写习惯，同时减少代码中的bug。

本规范包含PHP开发时程序编码中命名规范、代码缩进规则、控制结构、函数调用、函数定义、注释、包含代码、PHP标记、常量命名等方面的规则。

依据“约定大于规范”原则，本规范不强制指定和推荐某种格式，并就实际开发中个人习惯做了一些调整。因此，本书中部分代码也不完全符合本规范，这些规范仅供大家参考。

14.1 文件格式

14.1.1 文件标记

所有PHP文件，其代码标记均使用完整PHP标签，不建议使用短标签，例如：

```
<? php
echo ' Hello world! ';
? >
<?
//短标签格式不推荐
echo ' not suggest ';
? >
```

使用短标签格式容易和XML混淆，并且不是所有PHP版本和服务器都默认支持或打

开短标签选项（从PHP 5.4开始，php.ini中的短标签选项不影响短标签的使用）。

对于只含有PHP代码的文件，将在文件结尾处忽略“? >”。这是为了防止多余空格或者其他字符影响到代码。本书中大部分代码都是这么做的。

实际上这个问题只有在不开启压缩或缓存输出时才会出现，例如：

```
php.ini-禁止压缩输出及缓存输出
zlib.output_compression=off
output_buffering=off
```

foo.php，注意这个时候有一些空格或换行符掉在了“? >”之后，当然这在页面上是看不到的。

```
<? php
$foo= ' foo ';
? >
```

index.php，在包含foo.php的同时，实际上已经输出一些空格或换行了。

```
<? php
include ' foo.php ';
session_start ();
? >
```

这时将看到一个警告（warning）：“……Cannot send session cache limiter

headers already sent……”，之所以出现这个警告，是因为在`session_start()`之前输出一些看不到的字符。

14.1.2 文件和目录命名

程序文件名和目录名均采用有意义的英文命名，不使用拼音或无意义的字母，只允许出现字母、数字、下画线和中画线字符，同时必须以“.php”结尾（模板文件除外）。多个词间使用驼峰法命名。

```
//类统一采用
DemoTest.class.php
//接口统一采用
DemoTest.interface.php
//其他按照各自的方式
demoTest. { style } .php
//其他一些文件按照
demoTest.inc.php zend/demo.lib.php
```

14.1.3 文件目录结构

在开发规范、独立的PHP项目时，使用规范的文件目录结构，有助于提高项目逻辑结构合理性，对于扩展和合作以及团队开发均有好处。

通常一个完整独立的PHP项目，其文件和目录结构如下：

```
└─app//独立的应用
└─class//单个的类文件，共用的类文件（比如工具类）
└─conf/inc//配置文件目录
└─data//数据文件目录
└─doc//程序相关文档
└─htdocs//document_root
└─images//所有图片文件存放路径（在里面根据目录结构设立子目录）
└─css//CSS文件
└─js//JavaScript脚本文件
└─lib//共用类库
└─template//模板文件
└─temp//临时文件目录
  | └─cache | └─session
  | └─template_c
  | └─other
└─upload//上传文件（按特定规则分目录存放）
└─manage//后台管理文件存放目录
```

根据具体应用情况考虑目录结构，不用完全遵循，但是要尽量做到规范化。

14.2 命名规范

14.2.1 变量命名

PHP中的变量用一个美元符号后面跟变量名表示。变量名区分大小写。一个有效变量名由字母或者下划线开头，后面跟任意数量的字母、数字、下划线。正常的正则表达式将表述为：

```
[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*
```

不应该在变量中使用中文等非ASCII字符。

1.程序整体

程序整体以驼峰法命名，以小写字母开始，同时命名要有意义，如：

```
function displayName ($name) {  
    echo $name;  
}
```

2.PHP全局变量键值

PHP全局变量键值两边都有“_”，中间使用驼峰法命名，如：

```
$_GLOBAL[ '_beginTime_' ];
```

3.普通变量

普通变量整体采用驼峰法，建议在变量前加表示类型的前缀。不确定类型的以大写字母开头，函数参数不确定的类型以大写字母开头，其他地方的变量以小写字母开头。对于一些常见变量，按照约定命名，并避免使用常用关键字或存在模糊意义的单词。变量应该以名词为主。推荐：

字符串→sMyName

数组→arrMyArray

对象→oMyObject

资源→resource

布尔值→flag

不推荐：

yes：不应该使用其作为bool型变量，因为变量很可能被改变，其可能使得yes=false，而让其代码逻辑变得混乱。

sex：具有模糊意义且不地道的英文单词，性别的命名应该是gender。

4.函数名

函数名既要有意义，一看就知道要干什么，也要尽量缩写。建议采用动词或动词加形容词的命名方式，如showMsg。不建议下面这样的函数名：getPublishedAdvertisementBy

CategoryAndCategoryIdAndPosition.

上面的函数名可以提炼为：getAd (category , categoryid , position , published)。

5.类中的属性

类中的属性变量遵守普通变量的命名规则。

14.2.2 类及接口命名

在面向风格的代码中，其命名方式一些特殊。类的命名遵循以下规则：

以大写字母开头；

多个单词组成的变量名，单词之间不用间隔，各个单词首字母大写；

类名与类文件保持一致；

程序中所有类名唯一；

抽象类应以Abstract开头。

例如，定义class MyClass或class Database等对应的文件为：

```
MyClass.class.php
Database.class.php
```

类中的方法和函数采用相同的命名规则。

接口命名遵循以下规则：

- 1) 采取和类相同的命名规则，但在其命名前加i字符，表示接口。如iDataBae。
- 2) 尽量保持和实现它的类命名一致。

接口文件可使用“iDatabase.interface.php”命名。

14.2.3 数据库命名

在数据库相关的命名中，一律不出现大写。详细命名规则如下所示。

1) 数据表命名遵循以下规范：

表名均使用小写字母；

表名字使用统一的前缀，且前缀不能为空（模块化，且可有效规避MYSQL保留字）；

对于多个单词组成的表名，使用“_”间隔。例如：

```
blog_user_info blog_link
```

2) 表字段命名遵循如下规则：

全部使用小写字母命名；

多个单词不用下画线进行分割；

如果有必要，给常用字段加上表名首字母作为前缀；

避免使用关键字和保留字，但约定俗成的除外。

例如：

```
//表  
blog_user_info//字段  
uid  
opentime time  
from//不建议，和保留字冲突
```

3) 存储过程、触发器、event以及视图的命名在表的命名规则的基础上，遵循以下规则：

存储过程以proc_开头，如“proc_syn_nick_name_friend”；

触发器以tri_开头，如“tri_blog_user_u”；

Event调度以event_开头，如“event_rank”；

视图以view_开头，如“view_blog_user”。

14.2.4 习惯与约定

通常变量的命名应该是有意义的单词，但在循环体中的临时变量采用“IN规则”。

IN规则原本来自FORTRAN，在FORTRAN中，以字母表中I~N范围内字母开头的变量默认为整形变量。循环体中一般是整形变量，故习惯用I~N字母作为循环体中的变量命名。同时，I是标识符（Identify）首字母。如下所示：

```
function bubble_sort ($array) {
    $count=count ($array);
    for ($i=0; $i< $count; $i++) {
        for ($j= $count-1; $j> $i; $j--) {
            if ($array[$j]< $array[$j-1]) {
                $tmp= $array[$j];
                $array[$j]= $array[$j-1];
                $array[$j-1]= $tmp;
            }
        }
    }
    return $array;
}
```

这个嵌套循环中，使用i、j这样无意义的变量是允许的，并且也是能被普遍接受的。

1.缩写

按照习惯，同时为了减小变量的长度，在不影响可读性的前提下，习惯对变量进行缩写，常见于一些函数的参数。此规则适用于PHP之外的代码，如JavaScript代码、数据库表字段命名等，例如：

Image→img

string →str

database→db

array →arr

count →cnt

temporary →temp或者tmp

password →passwd或者pwd

message →msg

2.魔术数字

魔术数字 (magic number) 指直接写在程式码里的具体数值 (如10、123等以数字直接写出的值)。虽然程序作者写的时候自己能了解数值的意义, 但对其他程序员而言, 甚至经过一段时间后, 作者本人都难以记得这个数值的用途, 只能苦笑讽刺“这个数值的意义虽然不懂, 不过至少程式能够执行, 真是个魔术般的数字”。

一般认为程式码中不应该含有魔术数字, 理由是:

- 1) 数值的意义难以了解, 比如一些专业的数字。
- 2) 数值需要变动时, 可能不只改一个地方。

看下面这个例子:

```
$price_tax=1.05*price
```

上面代码计算输入的价格 (price) 考虑了税率后的 (price_tax) 售价。当政府调整税率时, 这段程序就必要修改。这里1.05就是魔术数字, 我们可通过常量定义避免这种情况, 具体如下:

```
define (TAX, 1.05)  
$price_tax=TAX*price
```

14.3 注释规范

每个程序必须提供必要的注释，包括文件注释、代码块注释、函数注释等。

14.3.1 程序注释

程序注释原则如下：

写在被注释代码前面，而不是后面。但对于单行语句，按照习惯可以把注释放在语句末尾。

对于大段注释，使用`/**/`格式，通常在文件和函数注释中使用，而代码内部统一使用`//`注释，因为其写起来简单。

注释不宜太多，大家都能看懂的行不必注释。

例如：

```
/**
 *初始化过程
 *@access public
 *@return void
 */
function init () {
//列表文件不存在，则重新编译
if (! is_file ( DATA_ DIR. ' list.php ')) {
APP: clear (); //必须先删除缓存，防止数据不一致
APP: build ();
}
//other things
}
```

代码注释应该描述为什么，而不是做什么，给代码阅读者提供最主要的信息，不能为

了注释而注释。

注意：不要注释大段代码，因为这样容易分散阅读者的注意力。如果觉得这段代码现在用不到，但将来可能用到，这时需要VCS（Version Control System，版本控制系统）管理软件，比如SVN、GIT。IDE自带local history功能，可以作为本地的VCS系统，如图14-1所示。

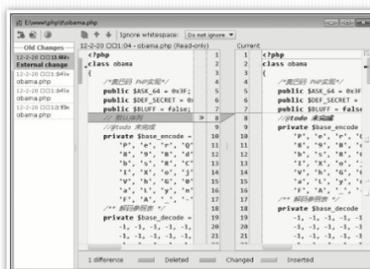


图 14-1 PhpStorm的local history功能

14.3.2 文件注释

文件注释通常放在整个PHP文件头部，其内容包括文件版权、作者、编写日期、版本号等重要信息。PHP中，可以参照phpdocument规范，便于利用程序自动生成文档。

文件注释遵循以下规则：

必须包含本程序的描述；

必须包含作者；

必须包含项目名称；

必须包含文件的名称；

可以包含书写日期；

可以包含版本信息；

可以包含重要的使用说明，如类的调用方法、注意事项等。例如：

```
<? php
/**
 *ZOC项目
 *
 *LICENSE*
 *本代码来自于ZOC项目，遵循GNU自由软件协议
 *
 *____
 *别问我为什么这个项目叫做ZOC，其实我也不明白
 *____
 *@description 模型初始化，按需加载所需要的资源
 *@package  MODELINI
 *@copyright  Copyright (c) 2005-2111阿里叔叔Inc.
 *@author  番茄炒西红柿tomato@qq.com
 *@version  1.01
 *@modify  优化了compile方法，提高效率。by: 西红柿炒番茄
```

```
*/
```

需要注意，文件一定要加上作者信息，这样有利于划分代码责任，同时方便代码阅读者联系作者。另外，版本号需要随着每次更新进行改动，并且加上modify注释，表明每次改动什么地方。

14.3.3 类/接口注释

类和接口的注释应该尽量简洁。按照一般的习惯，一个文件只包含一个类，在类注释中通常不需要再加上作者和版本等信息，加上可见性和简单的描述即可。如果文件注释已经足够详细，可以不用给类写注释。如果同时存在接口和接口的实现类，通常做法是仅在接口中进行注释。

14.3.4 方法和函数注释

方法和函数的注释写在前面，通常需要标明的信息主要是可见性、参数类型和返回值的类型，例如：

```
/**
 *连接数据库
 *@param string $dbhost 数据库服务器地址
 *@param string $dbuser 数据库用户名
 *@param string $dbpw 数据库密码
 *@param string $dbname 数据库名
 *@param string $charset数据库的编码（推荐使用UTF8）
 *@access public
 *@return void*/
public function construct ( $dbhost, $dbuser, $dbpw, $dbname= ' ', $charset= ' utf8 ' ) { //
}
```

如果参数使用PHP中的type hint，可以不对一些参数类型进行注释。

14.3.5 标注的使用

IDE支持一些特殊注释，可以列出整个项目中的特殊注释，方便后期维护和代码检查，例如：

//@fixMe表示需要修复项。如：修复了IP获取的一个安全漏洞

//@todo表示需要完善的地方。如：这个函数的效率太低，需要改进

上述注释和Java中的标注及注解比较相像。NetBeans中标注的效果如图14-2所示。

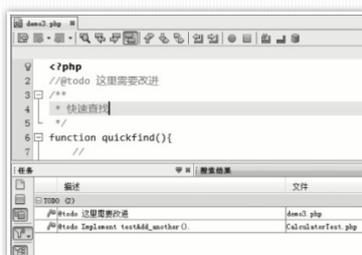


图 14-2 NetBeans中的标注

不同IDE对这类特殊注释的支持程度不同。为了编码效率和团队协作，建议在项目开发中使用大型IDE进行项目管理。

对于代码不推荐使用的函数或方法，使用@Deprecated注释；对于重载方法，使用@over load注释。NetBeans中在“工具”→“选项卡”中设置TODO模板，如图14-3所示。可以在TODO模式中自行添加标注。



图 14-3 在NetBeans中自定义标注

14.4 代码风格

14.4.1 缩进和空格

在书写代码的时候，必须注意代码的缩进规则：

使用4个空格作为缩进，而不使用tab缩进（如在UltraEdit中可以进行预先设置）。

变量赋值时，等号左右留出空格。

例如：

```
$url= $_GET[ ' url ' ]; //推荐  
$url= $_GET[ ' url ' ]; //不推荐
```

为了最大程度减轻工作量，保持代码美观，建议使用大型IDE管理代码。比如，在Net Beans中，使用Alt+Shift+F组合键对代码进行格式化。

14.4.2 语句断行

代码书写中应遵循以下原则：

尽量保证程序语句一行就是一句；

尽量不要使一行的代码太长，一般控制在80个字符以内；

如果一行代码太长，请使用类似“.”的方式断行书写；

执行数据库的SQL语句操作时，尽量不要在函数内写SQL语句，而先用变量定义SQL语句，然后在执行操作的函数中调用定义的变量。例如：

```
//代码分割
$sSql="SELECT username, password, address, age, postcode FROM test_t";
$sSql.="WHERE username= $ { user } ";
$hRes=mysql_query ($ sSql);
```

14.4.3 更好的习惯

在代码中，使用下面列举的写法，可以使代码更优雅。

1) 多使用PHP中已经存在的常量，而不要自己定义，例如：

```
//换行
echo $msg.\r\n";
echo $msg, PHP_EOL;
```

PHP中，PHP_EOL是一个预定义常量，表示一行结束，随着所使用系统的不同，使用PHP_EOL会让代码更具有可移植性。

2) 在echo中使用逗号做字符串连接符，比使用点号代码更美观。

单引号的效率优于双引号，但不要刻意使用单引号，一则不美观，二则二者在用法上有区别，稍有不慎就容易犯错。尽管echo语句效率更高那么一点点，但请学会使用printf函数。把效率用在“刀刃”上。

```
//丑陋的echo
echo '每个'. $school. '大约有'. floor($avg). '个学生';
//优雅的printf
$format= '每个%s大约有%d个学生';
printf($format, $school, $avg);
//其实printf函数用得最多的地方是拼装SQL语句
```

3) 更详尽的注释。

注释是一门艺术，好的注释可以比代码更精彩。不用担心效率问题。一则注释对代码的效率影响不大，其次在正式产品中可以对代码中的注释进行批量删除。注释做到极致和完美的典型代表是Apache组织各种产品的源代码。

4) 不要滥用语法糖。

语法糖也就是语言中的潜规则，即不具有普遍代表性的语法。少量使用语法糖会尝到甜头，大量使用则是一种灾难。

14.5 本章小结

在开头就提到编码规范的目的是提高工作效率，保证开发的有效性和合理性，并可最大程度提高程序代码的可读性和可重复利用性，减少返工，提高沟通效率。好的代码风格，见代码如见人，读代码时就好像和作者本人在亲切交谈。

PHP是一门脚本语言，用于网站开发，和JavaScript类似，本身就是一门不严谨的语言，有人评价PHP是“dirty and fast language”（脏而快的开发语言），所以规范显得更重要。

最后要说明的是，本规范不是强制，也不是标准。“约定大于规范”，如果有的规范太死板，不适应你所在团队，你可以不采用，而是按照自己的规范进行。

强烈建议使用大型IDE进行代码开发和管理，并使用合适的版本控制软件。