

# 函数

## 函数基础

### 函数的定义：

形式：

```
function 函数名 ( 【$形参 1】 【, $形参 2】 【, .... 】 ){  
    //函数体。。。。。  
}
```

说明：

- 1，定义时使用的形参，其实就是一个变量——只能在该函数内部使用的变量
- 2，形参作为变量，其名字是“自己定义”——自然应该遵循命名规范；

### 函数的调用：

```
函数名 ($实参 1, $实参 2, ..... );
```

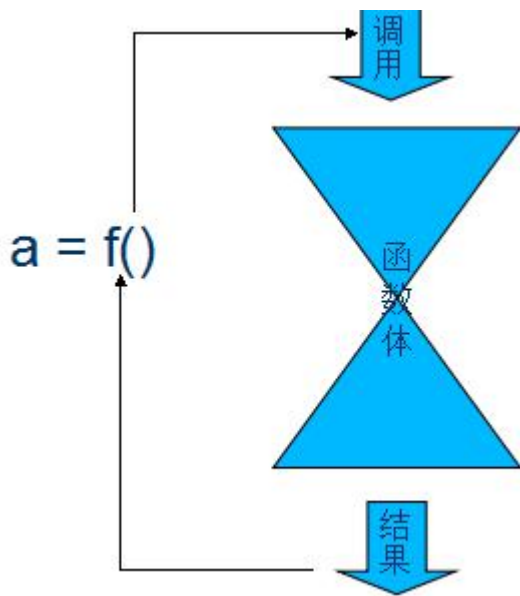
说明：

- 1，实参应该跟要调用的函数的形参“一一对应”；
- 2，实参就是“数据值”，可能是直接值（比如 5，“abc”），也可能是变量值(比如\$v1)

```
14 function f1($x, $y){  
15  
16     $s = $x * $x + $y * $y;  
17     $result = sqrt($s); //求其开方  
18     return $result; //返回该数据值  
19 }  
20 $v1 = f1(3,4);  
21 echo "v1 = $v1";  
22 echo "f1(5,6) = " . f1(5, 6);
```

### 函数调用详细过程

- 1，首先，将函数调用时的实参数据，传递（赋值）给函数的形参（变量）；
- 2，程序的执行流程，进入到函数内部——此时可以认为是一个跟外界“隔离”的“独立运行空间”。
- 3，在函数内部，按正常的流程顺序，执行其中的代码；
- 4，直到函数结束，则退出该运行空，而返回到原来调用函数的位置，继续执行后续代码！
- 5，如果在函数内部执行的过程中，有 **return** 语句，则也会立即终止函数，并回到函数调用位置。



## 函数的参数问题

### 函数形参的默认值问题

我们可以给一个函数定义时的形参，赋值一个“默认值”，则这个函数调用的时候，该形参对应的实参，可以不给值。

```

14 function f1($x = 3, $y = 4){
15     $s = $x * $x + $y * $y;
16     $result = sqrt($s); //求其开方
17     return $result; //返回该数据值
18 }
19 $v1 = f1(30, 40); //传过去2个数据分别给予x和y
20 $v2 = f1(30); //传过去1个数据，给予x，y自动获得默认值4
21 $v3 = f1(); //没有传值过去，x自动获得3，y自动获得4
  
```

函数形参的默认值，可以只给部分形参设置默认值，但设置默认值性的形参，都要放在“右边”（后边）：

```

24 function f2($a, $b=3, $c = 'abc'){
25     echo "<br />这是只是演示多个形参，部分有默认值情况";
26     echo "<br />a=$a, b=$b, c=$c";
27 }
28 f2(1);
29 f2(1,2);
30 f2(1,2, 'xyz');
31 f2(); //这种做法是错误的！
  
```

## 形参的传值问题

一句话：形参的传值问题，其实就是“变量之间的传值问题”：

其实无非就是实参变量，传值给形参变量的问题。

即：

此时，也同样有两种传值方式：

值传递：

这是默认值。如果没有特别设定，参数传值都是值传递。

引用传递：

需要在形参的前面加 & 符号：

```
33 //演示形参的引用传递问题
34 function f3($a, &$b){
35     $a = $a*$a;
36     $b = $b*$b;
37     return $a+$b;
38 }
39 // $v1 = f3(3, 4); // 这里报“致命错误”，因为4不能当做对应引用传递的形参的对应实参
40 // 这里，$b这个形参对应的实参，必须是一个“变量”，如下一行调用：
41 $s1 = 3;
42 $s2 = 4;
43 $v2 = f3($s1, $s2);
44 echo "<br /><br />v2 = $v2";
45 echo "<br />此时：s1 = $s1, s2 = $s2";
46 ?>
47 </body>
```

可见，值传递的实参变量，即使在函数内部对应的形参变量改变了其值，也不会改变该实参变量的值。  
相反，引用传递的实参变量，如果在函数内部对应的形参变量的值发生改变，则也就会改变该实参变量的值

v2 = 25  
此时：s1 = 3, s2 = 16

## 函数参数的数量问题

- 1，通常，函数调用时的实参数量，应该跟函数定义时的形参数量保持一致。
- 2，如果函数定义时，形参有默认值，则对应的实参就可以进行一定程度的省略：  
**注意：省略只能从右往左进行。**
- 3，有一种定义和使用函数的特别形式（并不常见）：它不定义形参，而实参任意给出。

其实，系统中也有类似的函数：，比如：

```
var_dump($v1);
```

```
var_dump($v1, $v2, $v3); //ok!
```

可见，该函数就可以接受任意个数的实参；

我们自己也可以定义这种函数。其实，这种函数，依赖的是以下 3 个系统函数来获取相应的信息，以得到实参数据的处理：


- 1: func\_get\_args(); //获取实参数据列表，成为一个数组
- 2: func\_get\_arg(\$i); //获取第\$i 个实参数据，\$i 从 0 开始算起；
- 3: func\_num\_args(); //获取实参的数量（个数）

下面就是例子：

```

48 //定义一个没有形参的函数
49 //但其可以接收任意个数的实参
50 function f4(){
51     //系统函数func_get_args()可以获取函数调用时传递过来的
52     //所有实参数据，并且都放入一个数组中！
53     $arr = func_get_args();
54     echo "<p>函数f4被调用，其实参为：";
55     foreach($arr as $value){
56         echo $value . " ";
57     }
58 }
59 f4(1, 2, 3);
60 f4('aa', 'bb');
61 ?>
62 </body>

```



## 函数的返回值问题

一个观念问题：

函数的返回值，不是语法规定，而是应用所需：需要就返回，不需要就无需返回。

返回值，一定是通过 `return` 语句！

形式：

```

function 函数名(...)
{
    //。。。。。
    return XX 数据;
}

```

注意：

`return` 语句的作用，不管后面跟不跟数据值，都会立即终止函数的执行，返回到函数调用的位置并继续后续工作。

## 函数的其他形式：

### 可变函数

先想想可变变量：

```
$v1 = "abc";
```

```
$abc = 123;
```



echo \$\$v1; //输出 123，这就是所谓的可变变量。

可变变量：一个变量的名字还是一个变量！

可变函数：一个函数的名字是一个变量！

```
14 function f1(){
15     echo "<br />这是一个普通的函数而已！";
16 }
17 $v1 = "f1";
18 $v1(); //这就是可变函数！
19 //可变函数其实就是在调用函数的时候，使用一个变量名而已。
20 //该变量的内部，就是该函数名！
21
```

演示可变函数的一个灵活性使用：

```
23 function jpg(){ echo "<br />处理jpg图片";}
24 function png(){ echo "<br />处理png图片";}
25 function gif(){ echo "<br />处理gif图片";}
26 $file = "abc.png"; //代表用户上传的图片；
27     //其后缀肯能是png, jpg, gif等
28 $houzhui = strrchr($file, ".");
29     //strrchr($s1,$s2)函数用于获取字符串$s1中最后一次
30     //出现的字符$s2之后的所有字符内容（含$s2本身）
31 //echo "<br />$houzhui";
32 $houzhui = substr($houzhui,1); //获得该字符串从位置1开始之后的所有字符
33 $houzhui(); //可变函数！
```

## 匿名函数

匿名函数就是没有名字的函数。

有 2 种形式的匿名函数：

形式 1：将一个匿名函数“赋值”给一个变量——此时该变量就代表该匿名函数了！

```

14 //将一个匿名函数，赋值给一个变量f1
15 $f1 = function (){
16     echo "<br />这是一个匿名函数! ";
17 };
18 $f1(); //调用该匿名函数，就使用该变量
19 //可见其形式跟调用可变函数一样!
20
21 //在演示一个带参数的匿名函数：
22 $f2 = function ($p1, $p2){
23     $result = $p1 + $p2;
24     return $result;
25 }
26 $re1 = $f2(3,4); //将函数的返回值赋值给$re1;

```

形式 2:

是直接将一个匿名函数，当做“实参”来使用！——即调用“别的函数 A”的时候，使用一个匿名函数来当做实参。自然，在该函数 A 中，也就应该对该匿名函数当做一个函数来用！

```

30 function func1( $x, $y, $z ){
31     $s1 = $x+$y;
32     $s2 = $x-$y;
33     $z($s1, $s2);
34 }
35
36 func1(3, 4,
37
38     function ($m1, $m2){
39         echo "m1=$m1, m2 = $m2";
40     }
41 );

```

这里，其实\$z()就是调用一个匿名函数，而且是传过来的匿名函数

就是它

```

30 function func1( $x, $y, $z ){
31     $s1 = $x+$y;
32     $s2 = $x-$y;
33     $z($s1, $s2);
34 }
35
36 func1(3, 4,
37
38     function ($m1, $m2){
39         $n = $m1 * $m2;
40         echo "<br />两个数是和乘以2个数的差的结果为: $n";
41     }
42 );

```

此时，执行这个\$z()函数，做的事情就跟前一次不一样了！

## 变量的作用域问题

变量的作用域，就是指：一个变量，在什么范围中可以使用的情况。

php 中，有 3 中变量作用域：

局部作用域：

就是指一个函数的内部范围。

对应这样的变量，就称为“局部变量”；

超全局作用域：

就是指所有的代码范围。

对应这样的变量，就称为“超全局变量”；

其实只有系统预定义的几个：\$\_GET, \$\_POST, \$\_SERVER, \$\_REQUEST, \$GLOBALS, \$\_SESSION, \$\_COOKIE, \$\_FILES

全局作用域：

就是不在函数内部的范围——函数外部。

对应这样的变量，就称为“全局变量”；



通常，

1，全局范围不能访问局部变量：

```
14 $v1 = 1;
15 function f1(){
16     echo "<br />在函数内部访问外部: v1 = $v1;";
17 }
18 f1();
19
```

**Notice:** Undefined variable: v1 in  
在函数内部访问外部: v1 = ;

2，局部范围不能访问全局变量：

```
22 //这里演示全局访问局部变量，出错：
23 function f2(){
24     $v2 = 1;
25 }
26 f2();
27 echo "<br />在函数外部访问局部变量: v2 = $v2;";
28
```

**Notice:** Undefined variable: v2  
在函数外部访问局部变量: v2 = ;



3, 函数内部的变量（局部变量），通常在函数调用执行结束后，就被“销毁”了。

4, 但有一种局部变量，在函数调用结束后不被销毁：它叫做“静态变量”；

使用形式：

```
function 函数名 (...){  
    static $变量名 = 初始值;    //这就是静态变量!  
    . . . . .  
}
```

```
30 function f3(){  
31     static $c = 0; //静态局部变量，它的值会保留  
32     //而且，此行的赋初值，只会执行第一次  
33     $c++;  
34     $d = 0;      //此行，每次进入函数，都会执行  
35     $d++;  
36     echo "<br />c=$c, d=$d, (此函数被调用次数为: $c)";  
37     //到此行之后，函数结束，其中的$d变量就被销毁了!  
38 }  
39 f3();  
40 f3();  
41 f3();
```

c=1, d=1, (此函数被调用次数为: 1)  
c=2, d=1, (此函数被调用次数为: 2)  
c=3, d=1, (此函数被调用次数为: 3)

可见，静态变量可以用于统计（计算）一个函数被调用了多少次。  
当然，也而已用于其它需要进行数据保留的函数调用过程。





## 如果在局部作用域使用（访问）全局变量？（常见需求）

有 2 种做法：

做法 1：

使用 global 关键字来实现：

```
43 echo "<hr />";
44 $v4 = 4;
45 function f4(){
46     //在函数中，使用global来声明一个要使用的全局变量的同名局部变量。
47     global $v4; //这里，$v4是局部变量，只是跟全局的v4同名
48     //实际情况是：此时外部v4变量跟内部的v4变量，共同
49     //指向一个数据区—引用关系！
50     echo "<br />在局部访问全局变量v4 = $v4";
51     $v4 = 44; //修改其值；
52 } //如果这里unset($v4); 不影响外部$v4的使用
53 f4();
54 echo "<br />在全局再次访问v4 = $v4";
55
56 ?>
```

全局变量\$v4

数据: 4 → 44

局部变量\$v4

在局部访问全局变量v4 = 4  
在全局再次访问v4 = 44

做法 2：

使用\$GLOBALS 超全局变量来实现：

```
57 echo "<hr />";
58 $v5 = 5; //全局变量
59 function f5(){
60     // $GLOBALS可以认为是全局变量的另一种使用形式！
61     echo "<br />在局部访问全局变量v5 = " . $GLOBALS['v5'];
62     $GLOBALS['v5'] = 55; //修改其值；
63 }
64 f5();
65 echo "<br />在全局再次访问v5 = " . $v5;
66 echo "<br />在全局再次访问v5 = " . $GLOBALS['v5'];
67
68 ?>
```

在局部访问全局变量v5 = 5  
在全局再次访问v5 = 55  
在全局再次访问v5 = 55

但，如果我们对\$GLOBALS 变量的某个单元（也即下标）进行 unset，则其就会完全对应销毁该变量。

这是因为，\$GLOBALS 对全局变量的使用可以看做是全局变量的另一种语法形式而已，而不是“引用关系”，举例如下：

```
71 $v6 = 6;    //全局变量
72 function f6(){
73     //GLOBALS可以认为是全局变量的另一种使用形式!
74     echo "<br />在局部访问全局变量v6 = " . $GLOBALS['v6'];
75     $GLOBALS['v6'] = 66;    //修改其值;
76     unset($GLOBALS['v6']);
77 }
78 f6();
79 echo "<br />在全局再次访问v6 = " . $v6;
80 ?>
81 </body>
82 </html>
```

此时就是实实在在地销毁了该全局变量

在局部访问全局变量v6 = 6

Notice: Undefined variable: v6 in

在全局再次访问v6 =

## 有关函数的系统函数：

- `function_exists()` :判断一个函数是否被定义过。其中使用的参数为“函数名”：

```
14 if( function_exists("f1") == false ){
15     function f1(){
16         echo "<br />这个函数我自己定义了!";
17     }
18 }
19 f1();    //这里就可以放心使用该函数了!
```

- `func_get_arg($i)` : 获取第 *i* 个实参值
- `func_get_args()` : 获取所有实参 ( 结果是一个数组 )
- `func_num_args()` : 获取所有实参的个数。

## 其他系统函数：

自己会查，并需要去查：

- 字符串函数：
  - 输出与格式化：`echo` , `print`, `printf`, `print_r`, `var_dump`.

- 字符串去除与填充：trim, ltrim, rtrim, str\_pad
- 字符串连接与分割：implode, join , explode, str\_split
- 字符串截取：substr, strchr, strrchr,
- 字符串替换：str\_replace, substr\_replace
- 字符串长度与位置： strlen, strpos, strrpos,
- 字符转换：strtolower, strtoupper, lcfirst, ucfirst, ucwords
- 特殊字符处理：nl2br, addslashes, htmlspecialchars, htmlspecialchars\_decode,
- [时间函数：](#)
  - time, microtime, mktime, date, idate, strtotime, date\_add, date\_diff, date\_default\_timezone\_set, date\_default\_timezone\_get
- [数学函数：](#)
  - max, min, round, ceil, floor, abs, sqrt, pow, round, rand

## 有关函数的编程思想

### 递归思想——递归函数

递归函数，就是：在一个函数内部调用它自己的函数！

先考察一个最简单的函数：

```
function fl($n){
    echo $n;
    $n++;
    fl($n);
}
```



```
}
```

fl(1);

从这个简单的函数可以看出，该函数调用是“永无止境”的（没完没了），最终会将内存消耗完毕。  
显然，这不是一个正常的做法！

实用的递归函数是：能够控制这个调用的过程中，会在某个时刻（条件下）停下来！

实例演示：

求 5 的阶乘。

数学上，有这样两个有关阶乘的基本规则：

1, n 的阶乘，是 n-1 的阶乘，乘以 n 的结果。

2, 1 的阶乘是 1；

现在，假设，有一个函数，该函数“能够”计算 n 的阶乘。

```
function jiecheng( $n ){
```

```
//.....
```

```
}
```

```
$v1 = jiecheng(8); //结果应该是 8 的阶乘
```

```
$v2= jiecheng(5); //结果应该是 5 的阶乘
```

```
14 //现在，假设，有一个函数，该函数“能够”计算n的阶乘。
15 function jiecheng( $n ){
16     echo "<br />开始：有人要求{$n}的阶乘";
17     if( $n == 1){
18         echo "<br />结束：终于求到了{$n}的阶乘：：1";
19         return 1;
20     }
21     $jieguo = $n * jiecheng($n-1);
22     echo "<br />结束：终于求到了{$n}的阶乘：：$jieguo";
23     return $jieguo;
24 }
25 $v2= jiecheng(5); //结果应该是5的阶乘
26 /*
27 演示调用过程：
28 $v2 = jiecheng(5)相当于：
29 $v2 = 5 * jiecheng(4)==>>
30 $v2 = 5 * (4 * jiecheng(3) ) ==>>
31 $v2 = 5 * (4 * (3 * jiecheng(2) ) ) ==>>
32 $v2 = 5 * (4 * (3 * (2 * jiecheng(1) ) ) ) ==>>
33 $v2 = 5 * (4 * (3 * (2 * 1 ) ) ) ==>>
34 $v2 = 5 * (4 * (3 * 2 ) ) ==>>
35 $v2 = 5 * (4 * 6 ) ==>>
36 $v2 = 5 * 24 ==>>
37 $v2 = 120
38 */
39 echo "<br />v2 = $v2";
40 }
```

开始：有人要求5的阶乘  
开始：有人要求4的阶乘  
开始：有人要求3的阶乘  
开始：有人要求2的阶乘  
开始：有人要求1的阶乘  
结束：终于求到了1的阶乘：：1  
结束：终于求到了2的阶乘：：2  
结束：终于求到了3的阶乘：：6  
结束：终于求到了4的阶乘：：24  
结束：终于求到了5的阶乘：：120  
v2 = 120

递归思想总结：

当面对一个“大问题”，该大问题可以经由该问题的同类问题的“小一级问题”而经过简单计算获得，

而且，可以获知（已知）这类问题的“最小一级问题”的答案。则，此时就可以使用递归方法来解决该问题。

则此时该函数的基本模式是：

```
function digui( $n ){  
    if(是最小一级){  
        return 已知的答案;  
    }  
    $jieguo = 对 digui($n-1) 进行简单运算;  
    return $jieguo;  
}
```

课间练习：

以下数列：1， 1， 2， 3， 5， 8， 13， .....

说明：

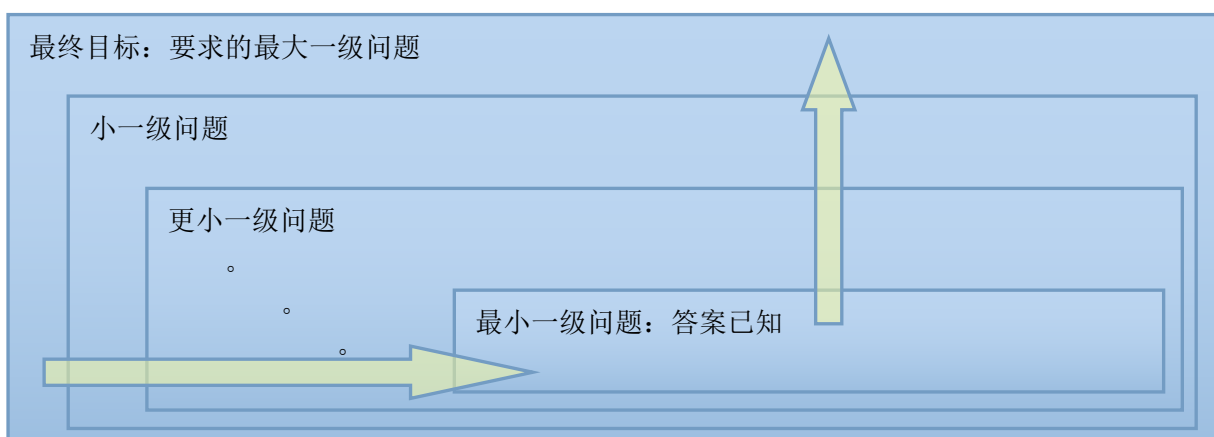
第1项是1，第2项也是1（都是已知）；

其他项，是其前两项的和；

求：第20项；

```
function shulie( $n ){ //把 n 理解地第几项;  
    if ( $n==1 || $n==2 ){  
        return 1;  
    }  
    $jieguo = shulie($n-2) + shulie($n-1);  
    return $jieguo;  
}  
$v1 = shulie( 20 );
```

递归思想图示：



## 递推（迭代）思想

也同样思考这个问题：

求 5 的阶乘：

先演示最初级的做法：

```
14 //演示递推思想：
15 //目标：要求5的阶乘：
16 $jiecheng1 = 1; //表示1的阶乘（已知）；
17 $jiecheng2 = $jiecheng1 * 2; //2的阶乘；
18 $jiecheng3 = $jiecheng2 * 3; //3的阶乘；
19 $jiecheng4 = $jiecheng3 * 4; //4的阶乘；
20 $jiecheng5 = $jiecheng4 * 5; //5的阶乘；
21 ?>
```

将上述代码，使用一个变量，也同样能完成：

```
14 //演示递推思想：
15 //目标：要求5的阶乘：
16 $jiecheng = 1; //表示1的阶乘（已知）；
17 $jiecheng = $jiecheng * 2; //2的阶乘；
18 $jiecheng = $jiecheng * 3; //3的阶乘；
19 $jiecheng = $jiecheng * 4; //4的阶乘；
20 $jiecheng = $jiecheng * 5; //5的阶乘；
21 ?>
```

然后，将上述代码的规律性体现出来——就是使用循环：

```
16 $jiecheng = 1; //表示1的阶乘（已知）；
17 for($i = 2; $i <= 5; ++$i){
18     //此循环会从2的阶乘开始，一次次求得
19     //“更大”一个数的阶乘，直到5的阶乘
20     $jiecheng = $jiecheng * $i; //i的阶乘；
21 }
```

然后，将该语句，再次进行转换，使用递推思想中的 2 个观念：前一个答案，后一个答案：

```
16 $qian = 1; //表示“前一个已知的答案”：这里是第一个，就是1的阶乘
17 for($i = 2; $i <= 5; ++$i){//意图求得从2开始到5的“每一个数阶乘”
18     //此循环会从2的阶乘开始，一次次求得
19     //“更大”一个数的阶乘，直到5的阶乘
20     $jieguo = $qian * $i; //要求的结果是由“前一个结果”经过简单乘法运算而得到
21     echo "<br />{$i}的阶乘是$jieguo";
22     $qian = $jieguo; //将当前求得的“结果”，又当成“前一个”，以供下一次使用！
23 }
24 echo "<br />结果为： " . $jieguo;
```

递推总结：

如果要求一个“大问题”，且该问题有如下 2 个特点：

1，已知该问题的同类问题的最小问题的答案。

2，如果知道这种问题的小一级问题的答案，就可以轻松求得其“大一级”问题的答案，并且此问题的级次有一定的规律；

则此时就可以使用递推思想来解决该问题，代码模式为：

\$qian = 已知的最小一级问题的答案；

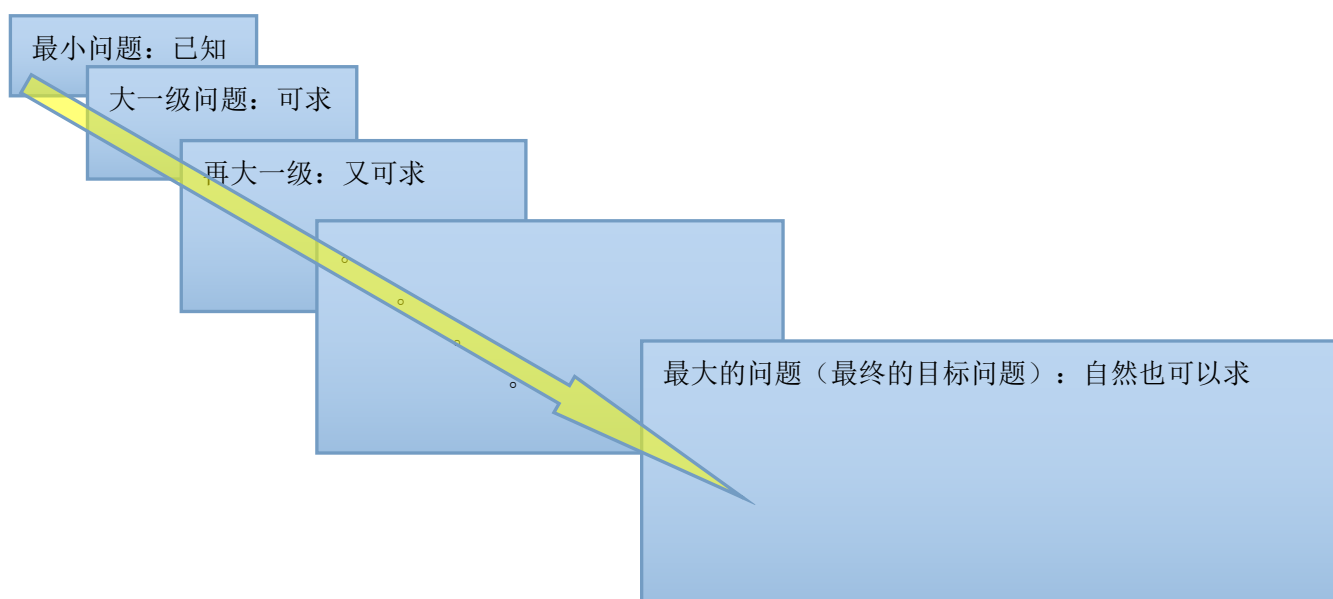
for( \$i = 最小一级的下一级； \$i <= 最大一级的级次； ++\$i ) {

```

$jieguo = 对 $qian 进行一定的计算，通常需要使用到$i;
$qian = $jieguo;
}
echo “结果为: ” . $jieguo;

```

递推思想图示：



通常，如果一个问题，既能使用递归算计解决，又能使用递推算法解决，则应该使用递推算法。

下面用递推思想来完成刚才的数列题：

以下数列：1， 1， 2， 3， 5， 8， 13， .....

求第 20 项：



```
26  /*
27  下面用递推思想来完成刚才的数列题:
28  以下数列: 1, 1, 2, 3, 5, 8, 13, .....
29  求第20项:
30  */
31  $qian1 = 1;
32  $qian2 = 1;
33  for($i = 3; $i <= 20; ++$i){
34      $jieguo = $qian1 + $qian2; //第3项
35      $qian1 = $qian2;
36      $qian2 = $jieguo;
37  }
38  echo "<br />数列的第20项为: " . $jieguo;
39
```