原码 反码 补码

原码：

就是一个二进制数字，从“数学观念”上来表达出的形式。其中，我们规定：

一个数字的最左边一位是“符号位”，0 表示正数，1 表示负数；

比如：

```
5的原码：
00000000 00000000 00000000 00000101
-3的原码：
10000000 00000000 00000000 00000011

```

反码：

正数的反码就是其本身（即不变）；

负数的反码是：符号位不变，其他位取反；

比如：

```
5的三码：
原码：   00000000 00000000 00000000 00000101
反码：   00000000 00000000 00000000 00000101

-3的三码
原码：   10000000 00000000 00000000 00000011
反码：   11111111 11111111 11111111 11111100
```

补码：

正数的补码就是其本身（即不变）；

负数的补码是：符号位不变，其他位取反后+1——即反码+1

```
5的三码：
原码：    00000000 00000000 00000000 00000101
反码：    00000000 00000000 00000000 00000101
补码：    00000000 00000000 00000000 00000101

-3的三码
原码：    10000000 00000000 00000000 00000011
反码：    11111111 11111111 11111111 11111100
补码：    11111111 11111111 11111111 11111101
```

提示：计算机内部的运算，实际全都是使用补码进行的，而且运算的时候，符号位不再区分，直接也当做数据参与运算

示例 1：

5+3：

```
演示5+3的cpu运算“过程”：
5的补码：     00000000 00000000 00000000 00000101
3的补码：     00000000 00000000 00000000 00000011
+
-----------------------------------------------------
结果：        00000000 00000000 00000000 00001000
可见结果是：2的3次方，即8
```

示例 2： 5-3：

实际上，cpu 内部，会将“减法”运算，转换为“加法运算”，即：5 + （-3）

```
演示5-3的cpu运算“过程”：实际进行的是：5 + （-3）
5的补码：     00000000 00000000 00000000 00000101
-3的补码：    11111111 11111111 11111111 11111101
+
-----------------------------------------------------
              00000000 00000000 00000000 00000010
可见，结果就是2。
```

位运算符的应用：管理一组食物的开关状态

什么是开关状态？

现实中，有很多数据都是只有 2 种结果（值）的，对应的其实就是我们的布尔类型的值。

这里，所谓管理一组事物的开关状态，应该理解为其实就是管理若干个只有 2 个状态的“数据符号”。

比如：有 5 个灯泡，对应 5 个状态数据。

这 5 个灯泡，就有 2^5 种状态呢？

这里的管理目标是：使用一个变量，就可以表达若干个数据的“当前状态”。具体有 3 个任务：

1，通过该变量，可以获知任何一个数据（灯泡）的当前状态。

2，通过该变量，可以将一个一个数据的状态“关闭”；

3，通过该变量，可以将一个一个数据的状态“开启”；

数组运算符

有这些：

+: 数组联合，也可以理解为“数组串联”。

将右边的数组项合并到左边数组的后面，得到一个新数组。如有重复键，则结果以左边的为准

```
$arr1 = array(5=>10,  8=>20,  10=>30);
$arr2 = array(3=>33,  2=>22);
$r1 = $arr1 + $arr2;//结果为：array(5=>10,  8=>20,  10=>30,  3=>33,  2=>22)
```

另一个有重复键的例子：

```
$arr1 = array(5=>10,  8=>20,  10=>30);
$arr2 = array(8=>33,  2=>22);
$r1 = $arr1 + $arr2;//结果为：array(5=>10,  8=>20,  10=>30,  2=>22)
```

==：如果两个数组具有相同的键名和键值（可以顺序不同，或类型不同），则返回 true

```
$arr1 = array(3=>33,  2=>22);
$arr2 = array(2=>”22”,  3=>”33”);
```

此时，$arr1,和 $arr2 是相等的（ == ）

!=

===： 如果两个数组具有相同的键名和键值且顺序和类型都一样，则返回 true

!==

错误控制运算符@：

通常就用在一个地方：

$link = @mysql_connect("数据库服务器地址","用户名", "密码");

作用是：

如果该连接数据的语句失败（比如连接不上），则屏蔽该失败的错误提示！

运算符的优先级

运算符，都有优先级问题！

记住以下几条就可以了：

- 要意识到运算符有优先级问题
- 括号最优先，赋值最落后（通常）
- 先乘除后加减
- 大致：单目运算符）算术运算符）比较运算符）逻辑运算符（除了“非”运算）

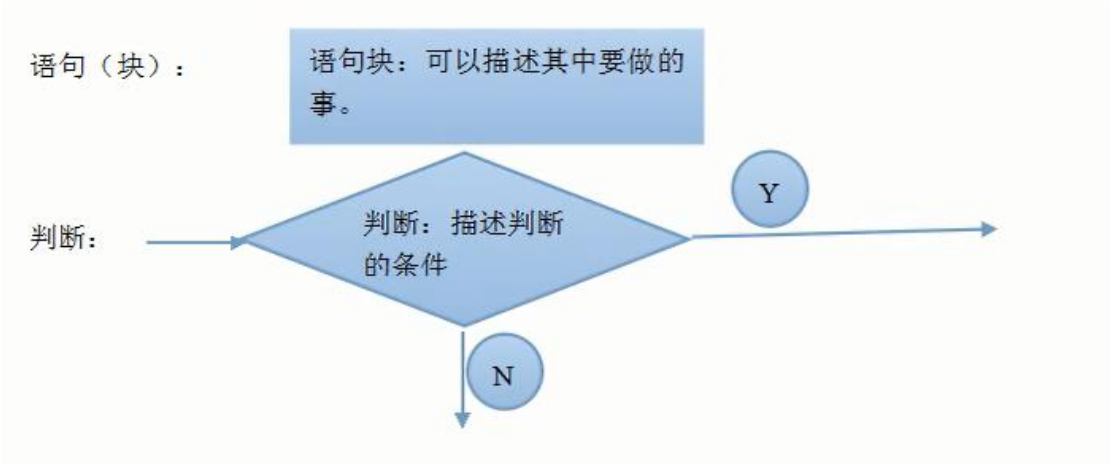
流程控制

流程图基本符号：

只是人们习惯上使用的一些图形符号，以代表一定的含义，帮组别人理解流程过程。

流程走向：

开始结束：



If 分支结构

基本语法形式如下：

```
if (条件判断 1）{
    分支 1；
}
else  if (条件判断 2）{
    分支 2；
}
else  if (条件判断 3）{
    分支 3；
}
。。。。。。
else {
    //else 分支
}
```

说明：

1，其中，绿色的 else if 部分可以重复若干次，也可以完全省略！

2，其中，紫色的 else 部分可以完全省略。

3，该 if 语句会从前往后（从上往下）依次判断条件，如果某个条件满足了，就会执行其中对应的分支，然后就结束 if 分支结构语句！

4，如果前面所有条件都不满足，就会执行最后的 else 分支（前提是有 else 分支）。

Switche 分支结构

形式：

```
switch ( 表达式 ){

    case  值 1：
        分支 1；
        【break；】  //是可以省略部分，不是语法所必须；
    case  值 2：
        分支 2；
        【break；】  //是可以省略部分，不是语法所必须；
    ……
    default ：
        default 分支；
}
```

说明：

1，将表达式的结果数据，跟“条件值 1”进行“相等判断”，如果相等，就执行分支 1，否则继续对后续值进行判断。。。。

2，如果某个分支判断为相等，则执行该分支语句后，并且如果其中没有 break 语句，则会直接进入下一个分支继续执行，而不会再去判断下一个分支的条件值了，并直到碰到 break 语句才会跳出。

For 循环结构

```php
//最基本for循环语句
for($i = 1; $i <= 9;  ++$i){
    echo $i;
    echo "<br />";
}

echo "<hr />";
//嵌套循环语句
for($i = 1; $i <= 9;  ++$i){
    //这里，可以看做是“输出一行”
    //一行输出若干个“*”
    for($k = 1; $k <= $i; ++$k){
        echo "*";
    }
    echo "<br />";  //一行结束（换行）
```

```php
//嵌套循环语句小应用：99乘法表
echo "<pre>";
for($i = 1; $i <= 9;  ++$i){
    //这里，可以看做是“输出一行”
    //一行输出若干个“*”
    for($k = 1; $k <= $i; ++$k){
        //echo "3 x 4 = 12";//要输出类似等式
        echo "$i x $k = " . ($i*$k) . "\t";
    }
    echo "<br />";  //一行结束（换行）
}
echo "</pre>";
```