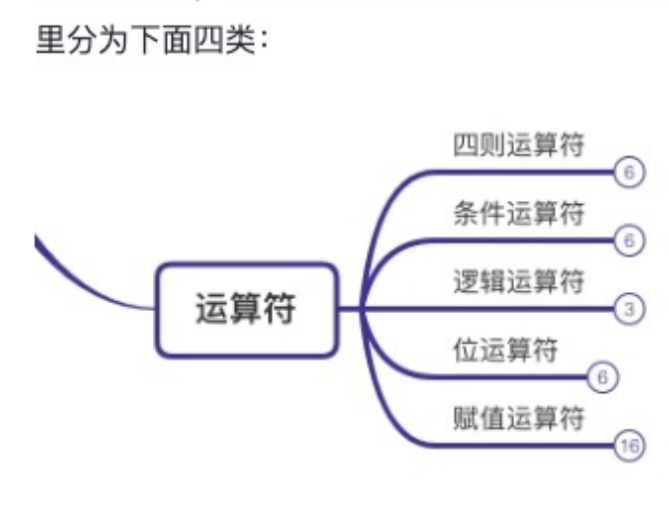
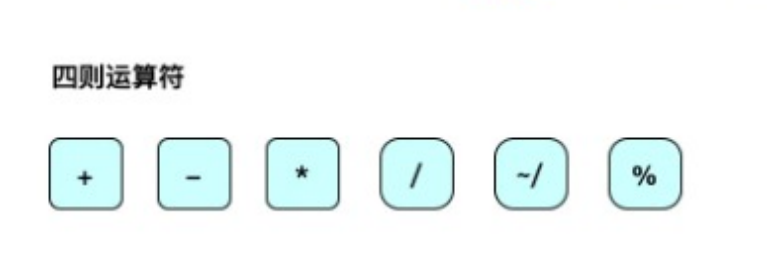
如果数学中没有运算符号，那么整个数学体系都会土崩瓦解。编程中也一样，运算符在逻辑实现中无处不在，它们就像逻辑的血肉一般。运算符的学习对接受过九年义务教育的人来说，应该是非常简单的，它很符合我们人类对事物的认知，所以也不用太大的压力。对于运算符的种类，这里分为下面四类：



---

1. 四则运算

四则运算是最基础的 `量` 之间的逻辑处理，包括下面的：`加` 、`减` 、`乘` 、`除` 、`商` 、`余` 六种运算。四则运算符的左右两侧是 `变量` ，会产出运算出的另一变量。



下面是对六个 `四则运算符` 的使用测试，其中 `~/` 表示 `取商` ，`%` 表示取 `余数` ，也有的人叫它取 `模` 。对于除法 `/` 来说，返回值类型为 `double` ，这和一些整数相除取商的语言有一点差别：

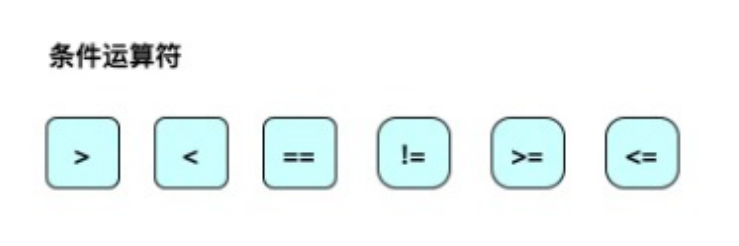
```dart
---->[grammar/operator/four_arithmetic.dart]----
void foo1() {
  int a = 10;
  int b = 3;
  print(a + b); // 13
  print(a - b); // 7
  print(a * b); // 30
  print(a / b); // 3.3333333333333335
  print(a ~/ b); // 3
  print(a % b); // 1
}
```

另外，要注意一点：对于浮点型的运算可能会有精度损失的问题，这点是浮点型本身的问题，任何编程语言都无法避免。在日常生活中，我们一般使用的小数大多不会绝对的精确，比如记录跑步的路程、身高、体重等，一般会保留几位小数。当对数字计算精度要求非常高的时候，比如金额的计算，可以使用 `decimal` 三方库。

```dart
void foo2(){
  double a = 10.2;
  int b = 3;
  print(a + b); // 13.2
  print(a - b); // 7.199999999999999
  print(a * b); // 30.599999999999998
  print(a / b); // 3.4
  print(a ~/ b); // 3
  print(a % b); // 1.1999999999999993
}
```

2.条件运算符

`条件运算` 符顾名思义，是用来校验条件的，这和我们常规的思维是一致的。条件运算符的左右两侧是 `变量` ，会产出一个 `bool` 值。



下面是对六个 `条件运算符` 的使用测试，使用很简单，了解一下语法即可：

```dart
---->[grammar/operator/condition.dart]----
void foo1() {
  int a = 5;
  int b = 6;
  print(a > b); // false
  print(a < b); // true
  print(a == b); // false
  print(a != b); // true
  print(a >= b); // false
  print(a <= b); // true
}
```

在这里引伸一下三目运算符的使用，如下 `getLager` 方法，返回两个数较大的值。要分两步：首先比较是否 `a` 比 `b` 大，然后根据比较的结果返回较大值。

```dart
num getLager(num a , num b){
  bool result = a > b;
  if(result){
    return a;
  }else{
    return b;
  }
}
```

三目运算符就相当于对上面两步的简写形式，整个表达式返回的是 `某类型对象` 。表达式的主体分为三段，`condition` 后加 `?` ，之后两对象通过 `:` 分隔。如果 `condition` 为 `true` ，返回 `:` 前的对象，反之取后。

```dart
Type#0 = condition ? Type#1 : Type#2
```

下面是 `getLager` 方法使用 `三目运算符` 的简写代码。另外要注意，`condition` 不要写的过于复杂，不然可读性会非常差。如果条件确实需要多段判断，可以先提取为局部变量，这样方便阅读。

```css
num getLager(num a, num b) {
  num result = a > b ? a : b;
  return result;
}
```

3.逻辑运算符

逻辑运算符是对 `bool` 值的运算有如下三种，分别是 `与` 、`或` 、`非` 。在使用时，`&&` 和 `||` 左右两侧连接 `bool` 值；`!` 后方连接 `bool` 值。

`&&` 的运算特点是：只有当左右都为 `true` 时，运算结果才为 `true` 。在生活中，这表示 `并且` 的意思，比如下面代码中，`allow` 表示：年龄是否小于 `14` ，并且身高小于 `1.4 m` 。`allow` 为 `true` 需要两个条件都满足。

```dart
---->[grammar/operator/logic.dart]----
int age = 13;
double height = 1.39;
bool allow = age < 14 && height < 1.40;
```

`||` 的运算特点是：当左右任意一个为 `true` 时，运算结果就为 `true` 。在生活中，这表示 `或者` 的意思，比如下面代码中，`allow` 表示：年龄是否小于 `14` ，或者身高小于 `1.4 m` 。`allow` 为 `true` 只要任意条件满足即可。

```dart
int age = 13;
double height = 1.59;
bool allow = age < 14 || height < 1.4;
```

`!` 的运算特点是：运算结果是 `bool` 值的反值。在生活中，这表示 `不是` 的意思。如下 `tag1` 处的 `!allow` 表示，不是 `allow` 的情况：

```dart
int age = 14;
double height = 1.59;
bool allow = age < 16 || height < 1.4;

if(!allow){ // tag1
  print("不允许入内!");
}
```

单个的逻辑运算符很容易理解，符合人类的思维，但如果是多个运算符叠加，就会使逻辑比较混乱。此时建议合理拆分，不要让一个逻辑运算的表达式过为冗长。

最后，来说一下逻辑运算符的 `“断路”` 特点。比如对于 `||` 而言，只要条件之一满足即为 `true` 。所以当第一个条件满足，后面的条件是什么就无所谓了，也就没必要去处理。这结合现实也很容易理解，比如说动物园，年龄小于 `12` 岁，或者身高小于 `1.2` 的人可以免费进入。当查看年龄符合条件，就没必要再检测身高。

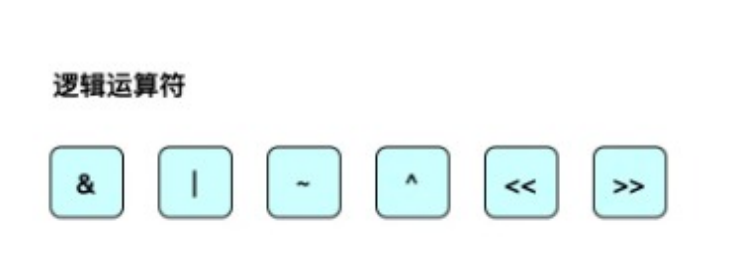
```dart
void foo3() {
  int age = 10;
  bool free = age < 12 || check();
}

bool check() {
  print("call check");
  return true;
}
```

如上代码中，`age<12` 满足，`check` 方法就不会触发。对于 `&&` 运算符也是类似，它的特点是 `一假全假` ，当第一个值为假，后面就不会再校验。逻辑运算符是比较贴近生活，符合人类思维的。

4.位运算符

我们知道在计算机中，数据是以 `二进制` 的形式进行存储的。而位运算符针对的就是 `二进制位` 进行的计算。在日常开发中位运算符的使用场景并不是太多，但也有必要了解一下。



拿 `int` 型的整数为例，它在内存中占据 `4` 字节，每字节 (`byte`) 由 `8` 个二进制数字 (`bit`) 表示。如下 `65` 、`11` 在计算机中以 `32` 个二进制位来记录：

```dart
十进制: 65
二进制: 0000 0000 0000 0000 0000 0000 0100 0001

十进制: 11
二进制: 0000 0000 0000 0000 0000 0000 0000 1011
```

`&` 左右连接值，是个二元运算符。其运算特点是：将左右值的位依次运算，只有对应位上下的数字都为 `1` 时，运算结果为 `1` ，其他情况为 `0` 。比如下面的示意中，依次对比上下两行，第三行是运算结果：

```dart
---->[grammar/operator/bit.dart]----
int a = 65;
int b = 11;
int c = a & b;

  0000 0000 0000 0000 0000 0000 0100 0001     a = 65
& 0000 0000 0000 0000 0000 0000 0000 1011     b = 11
  0000 0000 0000 0000 0000 0000 0000 0001     c = 1
```

`|` 左右连接值，也是个二元运算符。其运算特点是：将左右值的位依次运算，只要对应位上下的数字有一个是 `1` ，运算结果为 `1` ，其他情况为 `0` 。比如下面的示意中，依次对比上下两行，第三行是运算结果：

```dart
int a = 65;
int b = 11;
int c = a | b;

  0000 0000 0000 0000 0000 0000 0100 0001     a = 65
| 0000 0000 0000 0000 0000 0000 0000 1011     b = 11
  0000 0000 0000 0000 0000 0000 0100 1011     c = 75
```

`~` 右侧连接一个值，是个 `一元` 运算符。其运算特点是：将每位数字取反，也就是遇 `0` 写 `1` ，遇 `1` 写 `0` 。如下示意中，第二行是运算结果，应该非常清晰。

```dart
int c = ~b;
  0000 0000 0000 0000 0000 0000 0000 1011  b = 11
~ 1111 1111 1111 1111 1111 1111 1111 0100  c = -12
```

`*番外小知识====start*`

至于 `1111 1111 1111 1111 1111 1111 1111 0100` 为什么是 `-12` ，这里简单拓展一下，看不懂的也无所谓， 这属于计算机存储形式的知识，是比较底层的基础常识。对 `应用层` 来说并不是非常重要，并不影响应用开发。这就像人们不了解盐、水、酒精、塑料的元素构成、分子结构，并不会影响日常对这些事物的使用。但对于生物、化学方面的研究人员，理解分子结构就非常重要。适合的时期，学适合的东西，也是非常重要的。

`原码` 在形式上表现为 `符号 + 绝对值` ，比如 `-12` 用原码表示为：其中左侧第一位是符号位，`1` 表示负数，`0` 表示正数。

```dart
-12 原码:  1000 0000 0000 0000 0000 0000 0000 1100
```

`反码` 在形式上表现为：正数 `反码 = 原码` ；负数符号位与原码相同，其余位取反：

```dart
-12 原码:  1000 0000 0000 0000 0000 0000 0000 1100
-12 反码:  1111 1111 1111 1111 1111 1111 1111 0011
```

`补码` 在形式上表现为：正数 `反码 = 原码 = 补码` ；负数补码为 `反码 + 1` ：

```dart
-12 原码:  1000 0000 0000 0000 0000 0000 0000 1100
-12 反码:  1111 1111 1111 1111 1111 1111 1111 0011
-12 补码:  1111 1111 1111 1111 1111 1111 1111 0100
```

通过日志能看出，在计算机内存中存储的是 `1111 1111 1111 1111 1111 1111 1111 0100` ，也就是 `补码` 。至于为什么存储补码，感兴趣的可以自己了解一下。

`*番外小知识====end*`

`^` 左右连接值，是个二元运算符。其运算特点是：将左右值的位依次运算，只有对应位上下的数字不同时，运算结果为 `1` ，其他情况为 `0` 。比如下面的示意中，依次对比上下两行，第三行是运算结果：

```dart
int a = 65;
int b = 11;
int c = a ^ b;

  0000 0000 0000 0000 0000 0000 0100 0001     a = 65
^ 0000 0000 0000 0000 0000 0000 0000 1011     b = 11
  0000 0000 0000 0000 0000 0000 0100 1010     c = 74
```

`<<` 和 `>>` 是位移运算符，左侧是 `值` ，右侧是 `位移的数量` ，也是个二元运算符。 如下对 `65` 左移两位，就是把二进制数字向左移动两位，高位移出的舍弃，低位不够的补 `0` 。右移同理。

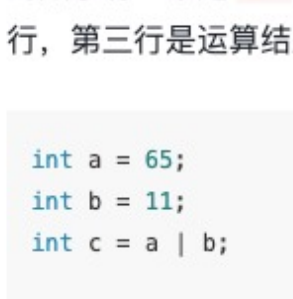
```dart
int a = 65;
int c = a << 2;

0000 0000 0000 0000 0000 0000 0100 0001     a = 65
0000 0000 0000 0000 0000 0001 0000 0100     c = 260
```

到这，六个位运算符就简单介绍了一下，现在看不懂也没有太大关系，了解它们的基本作用即可。

5.赋值运算符

赋值运算符的语法非常简单：左侧是变量名，右侧是值，简而言之就是为变量进行赋值。



```dart
---->[grammar/operator/equal.dart]----
int a = 10;
int b = a + 40;
```

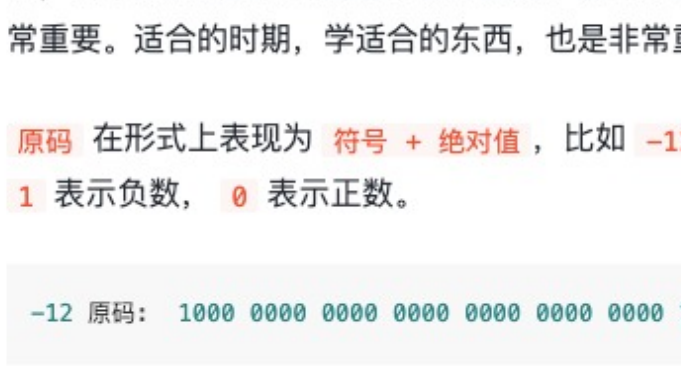
其中 `??=` 相当于一种简写形式。如下 `??=` 前面是变量 `a`，后面是默认值。表示当 `a` 为 `null` 时，为 `b` 赋值为默认值。如果 `a` 不为 `null` ，为 `b` 赋值为 `a` 。

```dart
int b = a ??= 20;

等价于:

int? a;
int b;
if(a != null){
  b = a;
}else{
  b = 20;
}
```

`四则运算` 和 `位运算` 中的二元运算符，都有相关的赋值运算符。这些也都比较简单，只是相当于一种简写形式而已。



如下所示：

```dart
int a = 10;
a += 20; // 等价于 a = a + 20;
a &= 10; // 等价于 a = a & 20;
```

其余的都是类似的，就不一一赘述了。

到这里 `Dart` 中的所有运算符就介绍完毕了，总的来说，这些运算符很符合我们的思维，理解上并没有什么大问题。不过如何合理地使用这些符号来完成需求，需要你在编程的生涯中慢慢体会。知道字怎么写，只是第一步，距离能写出一篇好文章还有很长的路要走。下面我们将进入下一个阶段：`面向对象` 。

留言

输入评论（Enter换行，⌘ + Enter发送）

发表评论

全部评论（6）

ObliviateOnline | 安卓开发工程师 @ 某监... | 3月前

很好，温故而知新，计算机存储补码原因如下：
1、计算机为了设计简单，只设计了加数寄存器，通过补码的形式可以将减法转换为加法，计算机只能进行加法操作
2、如果计算机中存储数据使用原码，则会导致两个问题：
1. 有负数参与的运算结果不正确；
1+（-1）的正确结果应该为0，而用原码存储时，得到的结果为-2...

展开

3 回复

蚊香酱 | iOS开发工程师 @ 自由... | 5月前

番外小知识这一节，计算机中存储的是补码而不是原码，这部分内容说反了

点赞 1

> **张风捷特烈**（作者） | 5月前
>
> 是我没理清，已改正。感谢指正
>
> 点赞 回复

胜2015 | 5月前

“拿 int 型的整数为例，它在内存中占据 8 字节，每位由 4 位二进制数字表示”
通常占4个字节，每个字节 8 个bit位（8个二进制数字），这样的?

点赞 1

> **张风捷特烈**（作者） | 5月前
>
> 我的锅，已更正
>
> 点赞 回复

故乡的原风景 | 5月前

已阅😁

点赞 回复