**布尔类型**

bool ：可能的取值为字面常数值 true 和 false 。

运算符：

* ! （逻辑非）
* && （逻辑与， "and" ）
* || （逻辑或， "or" ）
* == （等于）
* != （不等于）

声明：

赋值：

修改：

**整型**

关键字 uint8 到 uint256 （无符号，从 8 位到 256 位）以及 int8 到 int256，以 8 位为步长递增。 uint 和 int 分别是 uint256 和 int256 的别名。

运算符：

* 比较运算符： <= ， < ， == ， != ， >= ， > （返回布尔值）
* 位运算符： & ， | ， ^ （异或）， ~ （位取反）
* 算数运算符： + ， - ， 一元运算 - ， 一元运算 + ， \* ， / ， % （取余） ， \*\* （幂）， << （左移位） ， >> （右移位）

声明：

赋值：

修改：

**定长浮点型**

Solidity 还没有完全支持定长浮点型。可以声明定长浮点型的变量，但不能给它们赋值或把它们赋值给其他变量。。

fixed / ufixed：表示各种大小的有符号和无符号的定长浮点型。

关键字 ufixedMxN 和 fixedMxN 中，M 表示该类型占用的位数，N 表示可用的小数位数。 M 必须能整除 8，即 8 到 256 位。 N 则可以是从 0 到 80 之间的任意数。

ufixed 和 fixed 分别是 ufixed128x19 和 fixed128x19 的别名。

运算符：

* 比较运算符：<=， <， ==， !=， >=， > （返回值是布尔型）
* 算术运算符：+， -， 一元运算 -， 一元运算 +， \*， /， % （取余数）

声明：

赋值：

修改：

**地址类型**

address：地址类型存储一个 20 字节的值（以太坊地址的大小）。 地址类型也有成员变量，并作为所有合约的基础。

运算符：

* <=， <， ==， !=， >= 和 >

声明：address x

赋值：address x = 0x123; address myAddress = this;

修改：同赋值

地址比较的意义：

**地址类型成员变量**

* balance 和 transfer

可以使用 balance 属性来查询一个地址的余额， 也可以使用 transfer 函数向一个地址发送 （以 wei 为单位）：

address myAddress = this;

* send

send 是 transfer 的低级版本。如果执行失败，当前的合约不会因为异常而终止，但 send 会返回 false。

* call， callcode 和 delegatecall

**定长字节数组**

关键字有：bytes1， bytes2， bytes3， ...， bytes32。byte 是 bytes1 的别名。

运算符：

* 比较运算符：<=， <， ==， !=， >=， > （返回布尔型）
* 位运算符： &， |， ^ （按位异或）， ~ （按位取反）， << （左移位）， >> （右移位）
* 索引访问：如果 x 是 bytesI 类型，那么 x[k] （其中 0 <= k < I）返回第 k 个字节（只读）。

声明：

赋值：

修改：

.length 表示这个字节数组的长度（只读）.

**变长字节数组**

bytes: 变长字节数组，参见 [数组](https://solidity-cn.readthedocs.io/zh/develop/types.html#arrays)。它并不是值类型。

声明：

赋值：

修改：

string: 变长 UTF-8 编码字符串类型，参见 [数组](https://solidity-cn.readthedocs.io/zh/develop/types.html#arrays)。并不是值类型。

声明：

赋值：

修改：

**地址字面常数（Address Literals）**

比如像 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF 这样的通过了地址校验和测试的十六进制字面常数属于 address 类型。 长度在 39 到 41 个数字的，没有通过校验和测试而产生了一个警告的十六进制字面常数视为正常的有理数字面常数。

**有理数和整数字面常数**

整数字面常数由范围在 0-9 的一串数字组成，表现成十进制。 例如，*69* 表示数字 69。 Solidity 中是没有八进制的，因此前置 0 是无效的。

十进制小数字面常数带有一个 .，至少在其一边会有一个数字。 比如：1.，.1，和 1.3。

科学符号也是支持的，尽管指数必须是整数，但底数可以是小数。 比如：2e10， -2e10， 2e-10， 2.5e1。

数值字面常数表达式本身支持任意精度，除非它们被转换成了非字面常数类型（也就是说，当它们出现在非字面常数表达式中时就会发生转换）。 这意味着在数值常量表达式中, 计算不会溢出而除法也不会截断。

**字符串字面常数**

字符串字面常数是指由双引号或单引号引起来的字符串（"foo" 或者 'bar'）。 不像在 C 语言中那样带有结束符；"foo" 相当于 3 个字节而不是 4 个。 和整数字面常数一样，字符串字面常数的类型也可以发生改变，但它们可以隐式地转换成 bytes1，……，bytes32，如果合适的话，还可以转换成 bytes 以及 string。

字符串字面常数支持转义字符，例如 \n，\xNN 和 \uNNNN。\xNN 表示一个 16 进制值，最终转换成合适的字节， 而 \uNNNN 表示 Unicode 编码值，最终会转换为 UTF-8 的序列。

**十六进制字面常数**

十六进制字面常数以关键字 hex 打头，后面紧跟着用单引号或双引号引起来的字符串（例如，hex"001122FF"）。 字符串的内容必须是一个十六进制的字符串，它们的值将使用二进制表示。

**枚举类型**

enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }

ActionChoices choice;

ActionChoices constant defaultChoice = ActionChoices.GoStraight;

choice = ActionChoices.GoStraight;

**函数类型**

函数类型是一种表示函数的类型。可以将一个函数赋值给另一个函数类型的变量，也可以将一个函数作为参数进行传递，还能在函数调用中返回函数类型变量。 函数类型有两类：*内部（internal）* 函数和 *外部（external）* 函数：

函数类型表示成如下的形式：

function (<parameter types>) {internal|external} [pure|constant|view|payable] [returns (<return types>)]

pure

constant

view

payable

实际使用形式：function newCampaign(address beneficiary, uint goal)

public

returns (uint campaignID) {

}

**数组**

数组可以在声明时指定长度，也可以动态调整大小。

数组元素类型可以是任意的（包括数组类型、映射类型、结构体）。

数组，元素类型不能是映射类型，

作为 public 函数的参数，它只能是 ABI 类型。

举个例子，一个长度为 5，元素类型为 uint 的动态数组的数组，应声明为 uint[][5]。 要访问第三个动态数组的第二个元素，你应该使用 x[2][1]（数组下标是从 0 开始的，且访问数组时的下标顺序与声明时相反，也就是说，x[2] 是从右边减少了一级）。

动态长度数组

动态长度数组的声明：元素类型为 T，动态数组声明为 T[]

如：bool[2][] m\_pairsOfFlags;

动态数组的赋值：m\_pairsOfFlags[index][0] = flagA;

m\_pairsOfFlags[index][1] = flagB;

动态数组值修改：m\_pairsOfFlags[index][0] = flagC;

m\_pairsOfFlags[index][1] = flagD;

固定长度数组

固定长度的数组声明：元素类型为 T，固定长度为 k 的数组可以声明为 T[k]。如：uint[2\*\*20] m\_aLotOfIntegers;

固定长度的数组赋值：m\_aLotOfIntegers[index] = 122

固定长度数组值修改：m\_aLotOfIntegers[index] = 1555

**创建内存数组**

可使用 new 关键字在内存中创建变长数组:

声明动态内存数组：uint[] memory a = new uint[](7);

声明动态长度字节内存数组：bytes memory b = new bytes(len);

**数组字面常数 / 内联数组**

[uint(1), 2, 3]

**成员**

**length**:

数组有 length 成员变量表示当前数组的长度。 动态数组可以在 （而不是 ）中通过改变成员变量 .length 改变数组大小。 并不能通过访问超出当前数组长度的方式实现自动扩展数组的长度。 一经创建， 数组的大小就是固定的（但却是动态的，也就是说，它依赖于运行时的参数）。

**push**:

变长的 数组以及 bytes 类型（而不是 string 类型）都有一个叫做 push 的成员函数，它用来附加新的元素到数组末尾。 这个函数将返回新的数组长度。

**结构体**

声明结构体： struct Request { bytes data;

function(uint) external callback;

}

声明结构体变量数组：Request[] private requests;

为数组添加数据源数：requests.push(Request(data, callback));

使用其中的元素 ：requests[requestID].callback(response);

**映射**

映射类型在声明时的形式为：

mapping(KeyType => ValueType) 修饰符 变量名。

KeyType 可以是除了映射、变长数组、合约、枚举以及结构体以外的几乎所有类型。

ValueType 可以是包括映射类型在内的任何类型。

声明映射：mapping (address => uint) public balances;

映射赋值：balances [msg.sender] = newBalance;

映射的修改：方式同赋值。

**及 LValues 的运算符**

如果 a 是一个 LValue（即一个变量或者其它可以被赋值的东西），以下运算符都可以使用简写：

a += e 等同于 a = a + e。 其它运算符 -=， \*=， /=， %=， |=， &= 以及 ^= 都是如此定义的。 a++ 和 a-- 分别等同于 a += 1 和 a -= 1，但表达式本身的值等于 a 在计算之前的值。 与之相反，--a 和 ++a 虽然最终 a 的结果与之前的表达式相同，但表达式的返回值是计算之后的值。

**删除**

delete a 的结果是将 a 的类型在初始化时的值赋值给 a。即对于整型变量来说，相当于 a = 0， 但 delete 也适用于数组，对于动态数组来说，是将数组的长度设为 0，而对于静态数组来说，是将数组中的所有元素重置。 如果对象是结构体，则将结构体中的所有属性重置。

uint x = data;

delete x; // 将 x 设为 0，并不影响数据

delete data; // 将 data 设为 0，并不影响 x，因为它仍然有个副本

uint[] storage y = dataArray;

delete dataArray;

// 将 dataArray.length 设为 0，但由于 uint[] 是一个复杂的对象，y 也将受到影响，

**基本类型之间的转换**

**隐式转换**

如果一个运算符用在两个不同类型的变量之间，那么编译器将隐式地将其中一个类型转换为另一个类型（不同类型之间的赋值也是一样）。 一般来说，只要值类型之间的转换在语义上行得通，而且转换的过程中没有信息丢失，那么隐式转换基本都是可以实现的： uint8 可以转换成 uint16，int128 转换成 int256，但 int8 不能转换成 uint256 （因为 uint256 不能涵盖某些值，例如，-1）。 更进一步来说，无符号整型可以转换成跟它大小相等或更大的字节类型，但反之不能。 任何可以转换成 uint160 的类型都可以转换成 address 类型。

**显式转换**

如果某些情况下编译器不支持隐式转换，但是你很清楚你要做什么，这种情况可以考虑显式转换。 注意这可能会发生一些无法预料的后果，因此一定要进行测试，确保结果是你想要的！ 下面的示例是将一个 int8 类型的负数转换成 uint：

int8 y = -3;

uint x = uint(y);

这段代码的最后，x 的值将是 0xfffff..fd （64 个 16 进制字符），因为这是 -3 的 256 位补码形式。

如果一个类型显式转换成更小的类型，相应的高位将被舍弃

uint32 a = 0x12345678;

uint16 b = uint16(a); // 此时 b 的值是 0x5678

**类型推断**

为了方便起见，没有必要每次都精确指定一个变量的类型，编译器会根据分配该变量的第一个表达式的类型自动推断该变量的类型

uint24 x = 0x123;

var y = x;

这里 y 的类型将是 uint24。不能对函数参数或者返回参数使用 var。