引用类型(Reference Types)

引用类型包含：

- 字符串（string）
- 动态大小字节数组（bytes）
- 数组（Array）
- 结构体（Struts）

引用类型，赋值时，我们可以`值传递`，也可以`引用`即地址传递，如果是值传递，修改新变量时，不会影响原来的变量值，如果是`引用`传递，那么当你修改新变量时，原来变量的值会跟着变化，这是因为新就变量同时指向同一个地址的原因。

引用类型的变量有两种类型，分别是值传递的`memory`（深拷贝）和指针传递的`storage`（浅拷贝）。

一、memory（值传递）

当**引用类型作为函数参数**时，它的类型默认为`memory`，函数参数为`memory`类型的变量给一个变量赋值时，这个变量的类型必须和函数参数类型一致，所以我们可以写成`string memory name1 = name;`，或者`var name1 = name;`，**var**声明一个变量时，这个变量的类型最终由赋给它值的类型决定。

```
pragma solidity ^0.4.24;

contract Person {

    string public name;

    function Person() {
        name = "*kongyixueyuan";
    }

    function modifyName() {
        modify(name);
    }

    function modify(string str) private {
        // 不变
        // var name1 = str;
        // 不变
        string memory name1 = str;
        bytes(name1)[0] = '-';
    }

    function test() {
        // 会变
        // var name1 = name;
        // 会变
        // string name1 = name;
        // 不变
        string memory name1 = name;
        bytes(name1)[0] = '+';
    }
}
```

任何函数参数当它的类型为引用类型时，这个函数参数都默认为memory类型，memory类型的变量会临时拷贝一份值存储到内存中，当我们将这个参数值赋给一个新的变量，并尝试去修改这个新的变量的值时，最原始的变量的值并不会发生变化。

二、storage（指针传递）

当函数参数为`memory`类型时，相当于**值传递**，而`storage`类型的函数参数将是指针传递。

如果想要在`modifyName`函数中通过传递过来的指针修改`name`的值，那么必须将函数参数的类型显示设置为`storage`类型，`storage`类型拷贝的不是值，而是`name`指针，当调用`modifyName(name)`函数时，相当于同时有`str`，`name`,`name1`三个指针同时指向同一个对象，我们可以通过三个指针中的任何一个指针修改他们共同指向的内容的值。

```
pragma solidity ^0.4.24;

contract Person {

    string public name;

    function Person() {
        name = "*kongyixueyuan";
    }

    function modifyName() {
        modify(name);
    }

    function modify(string storage str) private {
        // 会变
        // var name1 = str;
        // 会变
        // string name1 = str;
        // 会变
        string storage name1 = str;
        bytes(name1)[0] = '-';
    }

    function test() {
        // 会变
        string storage name1 = name;
        bytes(name1)[0] = '+';
    }
}
```

函数默认为`public`类型，但是当我们的函数参数如果为`storage`类型时，函数的类型必须为`internal`或者`private`。

三、字符串（string）

字符串可以通过`""`来表示字符串的值，Solidity中的`string`字符串不像`C语言`一样以`\0`结束，字符串的长度就为我们所看见的字母的个数。

`string`字符串不能通过`length`方法获取其长度。

`string`字符串其实是一个特殊的可变字节数组。

```
pragma solidity ^0.4.24;

contract Demo{

    string public name; // 状态变量

    //构造函数
    function Demo() {
        name = "kongyixueyuan";
    }

    // set方法
    function setName(string str) {
        name = str;
    }

    // get方法
    function getName() constant returns (string) {
        return name;
    }

    function nameLength() constant returns (uint) {
        //报错
        //return name.length;
    }

}
```

四、动态大小字节数组（bytes）

- `bytes` 动态字节数组。
- `string` 是一个特殊的可变字节数组。

string动态大小字节数组

```
pragma solidity ^0.4.24;

contract Demo {

    string public name = "a!+&520";
    // string public name = "孔壹";
```

```
    function setNameFirstByteForL(bytes1 z) {
        bytes(name)[0] = z;
    }

    function nameBytes() constant returns (bytes) {
        return bytes(name);
    }

    function nameLength() constant returns (uint) {
        return bytes(name).length;
    }
}
```

不管是`字母`、数字还是特殊符号，每个字母对应一个byte（字节）。而一个汉字需要通过`3个字节`来进行存储。

bytes 动态字节数组

```
pragma solidity ^0.4.24;

contract Demo {

    // 初始化一个两个字节空间的字节数组
    bytes public name = new bytes(2);

    // 返回字节数组的长度
    function nameLength() constant returns (uint) {
        //报错
        return name.length;
    }

    // 设置字节数组的长度
    function setNameLength(uint len) {
        name.length = len;
    }

    // 往字节数组中添加字节
    function pushAByte(byte b) {
        name.push(b);
    }

}
```

bytes可变数组可使用length和push方法

通过`bytesI`来声明的字节数组为不可变字节数组，字节内容不可修改，字节数组长度不可修改。

通过`bytes name = new bytes(length)`， `length`为字节数组长度，来声明可变大小和可修改字节内容的可变字节数组。

五、固定大小字节数组、动态大小字节数组、字符串之间的转换

固定大小字节数组之间的转换

当`bytes9`转`bytes1`或者`bytes2`时，会进行低位截断。

当`bytes9`转换为`bytes32`时会以0进行低位补齐

```
pragma solidity ^0.4.24;

contract Demo {

   bytes6 name = 0xe5ad94e5a3b9;

   function bytes9ToBytes1() constant returns (bytes1) {
       return bytes1(name);
   }

   function bytes9ToBytes2() constant returns (bytes2) {
       return bytes2(name);
   }

   function bytes9ToBytes32() constant returns (bytes32) {
       return bytes32(name);
   }

}
```

固定大小字节数组转动态大小字节数组

`固定大小字节数组`不能直接转换成`动态大小字节数组`，正确方式如下

```
pragma solidity ^0.4.24;

contract Demo {

    bytes6 name = 0xe5ad94e5a3b9;

    function bytesIToBytes() constant returns (bytes) {
        //报错
        //return bytes(name);
```

```
        bytes memory names = new bytes(name.length);

        for(uint i = 0; i < name.length; i++) {
            names[i] = name[i];
        }

       return names;
   }
}
```

在上面的代码中，我们根据固定字节大小数组的长度来创建一个`memory`类型的动态类型的字节数组，然后通过一个`for循环`将固定大小字节数组中的字节按照索引赋给动态大小字节数组即可。

动态大小字节数组转string

因为string是特殊的动态字节数组，所以string只能和动态大小字节数组之间进行转换，不能和固定大小字节数组进行转行。

```
pragma solidity ^0.4.24;

contract Demo {

    bytes public names = new bytes(2);

    function Demo() {
        names[0] = 0x61;
        names[1] = 0x62;
    }

    function namesToString() constant returns (string) {
        return string(names);
    }
}
```

`string`本身是一个特殊的动态字节数组，所以它只能和`bytes`之间进行转换，不能和固定大小字节数组进行直接转换，如果是固定字节大小数组，需要将其转换为动态字节大小数组才能进行转换。

固定大小字节数组转换为string

固定大小字节数组不能直接转换为string。需要先将字节数组转动态字节数组，再转字符串。

```
pragma solidity ^0.4.4;

contract C {
   bytes9 name9 = 0x6c697975656368756e;
```

```
    function byte32ToString(bytes32 b) constant returns (string) {
        //报错
        // return string(name9);

        bytes memory names = new bytes(b.length);

        for(uint i = 0; i < b.length; i++) {
            names[i] = b[i];
        }

       return string(names);
    }
}
```

六、数组（Array）

数组可以声明时指定长度，也可以是动态变长。对storage存储的数组来说，元素类型可以是任意的，类型可以是数组，映射类型，结构体等。但对于memory的数组来说。如果作为public函数的参数，它不能是映射类型的数组，只能是支持ABI的类型（点击查看详情）。

固定长度的数组只是不可修改它的长度，不过可以修改它内部的值，而`bytes0 ~ bytes32`固定大小字节数组中，大小固定，内容固定，长度和字节均不可修改。

固定长度的数组（Arrays）

固定长度的数组不可修改它的长度，不过可以修改它内部的值，而`bytes0 ~ bytes32`固定大小字节数组中，大小固定，内容固定，长度和字节均不可修改。

固定大小的数组不能调用`push`方法向里面添加存储内容，声明一个固定长度的数组，比如：`uint[5] T`，数组里面的默认值为`[0,0,0,0,0]`，我们可以通过索引修改里面的值，但是不可修改数组长度以及不可通过`push`添加存储内容。

```
pragma solidity ^0.4.24;

contract Demo {

    // 数组的长度为5，数组里面的存储的值的类型为uint类型
    uint[5] public T = [1,2,3,4,5];

    function arrayLength() constant returns (uint) {
        return T.length;
    }

    function getArray() view returns (uint[5]) {
        return T;
    }
```

```
    function setTLength(uint len) public {
        //报错
        // T.length = len;
    }

    function setTIndexValue(uint index, uint num) public {
        T[index] = num;
    }

    function pushUintToT() public {
        //报错
        // T.push(6);
    }
}
```

可变长度的Arrays

`uint [] T = [1,2,3,4,5]`，这句代码表示声明了一个可变长度的`T`数组，因为我们给它初始化了`5`个无符号整数，所以它的长度默认为`5`。对可变长度的数组而言，可以通过`length`修改它的长度，也可修改元素值。也能通过push往数组中添加值，当往里面增加一个值，数组的个数就会加1。

```
pragma solidity ^0.4.24;

contract Demo {

    uint [] T = [1,2,3,4,5];

    function getArray() view returns (uint[]) {
        return T;
    }

    function TLength() constant returns (uint) {
        return T.length;
    }

    function setTLength(uint len) public {
        T.length = len;
    }

    function setTIndexValue(uint index, uint a) public {
        T[index] = a;
    }

    function pushUintToT(uint a) public {
        T.push(a);
    }
}
```

二维数组 - 数组里面放数组

```
pragma solidity ^0.4.24;

contract Demo {

    uint [2][3] T = [[1,2],[3,4],[5,6]];

    function TLength() constant public returns (uint) {

        return T.length; // 3
    }
}
```

`uint [2][3] T = [[1,2],[3,4],[5,6]]`这是一个三行两列的数组，你会发现和Java、C语言等的其它语言中二位数组里面的列和行之间的顺序刚好相反。在其它语言中，上面的内容应该是这么存储`uint [2][3] T = [[1,2,3],[4,5,6]]`。

上面的`数组T`是`storage`类型的数组，对于`storage`类型的数组，数组里面可以存放任意类型的值（比如：其它数组，结构体，字典 / 映射等等）。对于`memory`类型的数组，如果它是一个`public`类型的函数的参数，那么它里面的内容不能是一个`mapping(映射 / 字典)`，并且它必须是一个`ABI`类型。

创建 Memory Arrays

创建一个长度为`length`的`memory`类型的数组可以通过`new`关键字来创建。`memory`数组一旦创建，它不可通过`length`修改其长度。

```
pragma solidity ^0.4.24;

contract Demo {

    function f(uint len) {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // 在这段代码中 a.length == 7 、b.length == len
        a[6] = 8;
        // a.length = 2;
        b[1] = 2;
        // b.length = 3;
    }
}
```

memory类型的固定长度的数组不可直接赋值给storage/memory类型的可变数组

```
pragma solidity ^0.4.24;

contract Demo {
    function f() public {
        //报错
        // uint[] memory x = [uint(1), 3, 4];
        //报错
        // uint[] storage x = [uint(1), 3, 4];
        uint[3] memory x = [uint(1), 3, 4];
    }
}
```

数组字面量 Array Literals / 内联数组 Inline Arrays

```
pragma solidity ^0.4.24;

contract Demo {

    function f() public {
        //报错
        // g([1, 2, 3]);
        g([1, uint(2), 3]);
    }

    function g(uint[3] _data) {

    }
}
```

在上面的代码中，`[1, 2, 3]`是`uint8[3] memory`类型，因为`1、2、3`都是`uint8`类型，他们的个数为`3`，所以`[1, 2, 3]`是`uint8[3] memory`类型。但是在`g`函数中，参数类型为`uint[3]`类型，显然我们传入的数组类型不匹配，所以会报错。

将`g([1, 2, 3]);`替换成成`g([uint(1), 2, 3]);`

我们将`[1, 2, 3]`里面的第`0`个参数的类型强制转换为`uint`类型，所以整个`[uint(1), 2, 3]`的类型就匹配了`g`函数中的`uint[3]`类型。

创建固定大小字节数组 / 可变大小字节数组

`bytes0 ~ bytes32`我们可以通过`byte[len] b`来创建，`len`的范围为`0 ~ 32`。不过这两种方式创建的不可变字节数组有一小点区别，`bytes0 ~ bytes32`直接声明的不可变字节数组中，**长度不可变，内容不可修改**。而`byte[len] b`创建的字节数组中，**长度不可变，但是内容可修改**。

```
pragma solidity ^0.4.24;

contract Demo {
```

```
    bytes6 a = 0x6b6f6e677969;
    byte[6] aa = [byte(0x6b),0x6f,0x6e,0x67,0x79,0x69];

    byte[] cc = new byte[](10);

    function setAIndex0Byte() public {
        // 错误，不可修改
        // a[0] = 0x89;
    }

    function setAAIndex0Byte() public {
        aa[0] = 0x89;
    }

    function getAA() constant returns (byte[6]) {
        return aa;
    }

    function getCC() constant returns (byte[]) {
        return cc;
    }

    function setCC() {
        for(uint i = 0; i < a.length; i++) {
            cc.push(a[i]);
        }
    }
}
```

七、结构体（Struts）

结构体里面可以声明任意类型的变量，有如下两种方式可以实例化结构体。

```
pragma solidity ^0.4.24;

contract Demo {

    struct Person {
        uint age;
        uint stuID;
        string name;
    }

    Person public _person = Person(18,101,"孔壹学院");
    Person public _person2 = Person({age:19,stuID:102,name:"孔壹学院2"});
```

```
    function setPerson(Person p) internal {
        Person memory _person3 = Person(20,103,"孔壹学院3");
        Person memory _person4 = p;
    }
}
```

八、引用类型总结

当引用类型作为函数参数时，它的类型默认为`memory`。

当引用类型作为局部变量时，它的类型是`memory`。

当引用类型作为状态变量时，它的类型才是`storage`。

当引用类型作为函数参数指定为`storage`类型时，函数的类型必须为`internal`或者`private`。