一个完整的合约

```
pragma solidity ^0.4.24;

contract Counter {

    uint count = 0;
    address owner;

    function Counter(uint a) {
       count = a;
       owner = msg.sender;
    }

    function increment() public {
       if (owner == msg.sender) {
          count = count + 10;
       } else {
           count++;
       }
    }
```

```
    function getCount() constant returns (uint) {
       return count;
    }

    function kill() {
       if (owner == msg.sender) {
          selfdestruct(owner);
       }
    }
}
```

版本声明

```
pragma solidity ^0.4.4;
```

`pragma solidity`代表`solidity`版本声明，`0.4.4`代表`solidity`版本，`^`表示向上兼容，`^0.4.4`表示`solidity`的版本在`0.4.4 ~ 0.5.0(不包含0.5.0)`的版本都可以对上面的合约代码进行编译，`0.4.5`,`0.4.8`等等可以用来修复前面的`solidity`存在的一些`bug`。

合约声明

`contract`是合约声明的关键字，`Counter`是合约名字，`contract Counter`就是声明一个`Counter`合约。

`contract`相当于其他语言中的`class`，`Counter`相当于类名，`contract Counter`相当于`class Counter extends Contract`。

状态变量

状态变量是在合约内声明的公有变量。如

```
uint count = 0;
address owner;
```

`count`和`owner`就是状态变量，合约中的状态变量相当于`类`中的属性变量。

本地变量/局部变量

越过作用域即不可被访问，等待被回收。如`increment()`方法中声明的`step`就是局部变量。局部变量只在离它最近的`{}`内容使用。

构造函数（Contructor）

`function d()`函数名和合约名相同时，此函数是合约的构造函数，当合约对象创建时，会先调用构造函数对相关数据进行初始化处理。只会在合约部署时被调用一次。

成员函数

`function increment() public`和`function getCount() constant returns (uint)`都是`Counter`合约的成员函数，成员函数在iOS里面叫做方法、行为，合约实例可以调用成员函数处理相关操作。当调用`increment()`函数时，会让`状态变量count`增加`step`。当调用`getCount()`时会得到状态变量`count`的值。

析构函数（selfdestruct）

`析构函数`和`构造函数`对应，构造函数是初始化数据，而析构函数是销毁数据。在`counter`合约中，当我们手动调用`kill`函数时，就会调用`selfdestruct(owner)`销毁当前合约。

全局变量

```
msg.value ： 执行合约时，转账的eth数量，以wei为单位。
//payable ： 获取全局变量msg.value的方法必须使用payable标记。
msg.sender ： 执行合约的地址
```

类的多继承、重写

`solidity`类具有多继承的特性：

```
pragma solidity ^0.4.4;

contract Animal1 {
    function sayHelle() pure returns (string) {
        return "hello";
    }
}

contract Animal2 {
    function sayBye() pure returns (string) {
        return "bye";
    }
}

contract Dog is Animal1, Animal2 {
    // Dog 会继承 Animal1 及 Animal2 两个类
}
```

我们可以直接调用继承过来的函数，当然，我们还可以对继承过来的函数进行重写。重写与其他语言相通，即子类的同名函数会覆盖从父类继承的方法：

```
pragma solidity ^0.4.4;

contract Animal1 {
```

```
    function sayHelle() pure returns (string) {
        return "hello";
    }
}

contract Animal2 {
    function sayBye() pure returns (string) {
        return "bye";
    }
}

contract Dog is Animal1, Animal2 {
    function sayBye() pure returns (string) {
        return "bye Dog";
    }
}
```

函数的访问权限public、internal、private、external

- `private`: 私有函数。内部正常访问，外部无法访问，子类无法继承。
- `internal`: 内部函数。内部正常访问，外部无法访问，子类可继承。

  内部状态变量。内部正常访问，外部无法访问，子类可继承。
- `public`: 公共函数。内部正常访问，外部正常访问，子类可继承。
- `external`: 外部函数。内部不能访问（可通过this.()访问），外部正常访问，子类可继承。

函数默认为public类型，状态变量默认为internal类型。

```
pragma solidity ^0.4.4;

contract Animal {

    // public  公有：外部、内部、子类都可使用
    function testPublic() public pure returns (string) {
        return "public";
    }
    // private 私有：合约内部可以正常访问
    function testPrivate() private pure returns (string) {
        return "private";
    }
    // internal 内部：合约内部可以正常访问
    function testInternal() internal pure returns (string) {
        return "internal";
    }
    // external 外部：只能供外部访问
    function testExternal() external pure returns (string) {
```

```
        return "external";
    }

    function f() public  {
        testPublic();
        testInternal();
        testPrivate();
        this.testExternal();
    }
}

contract Dog is Animal {
    function call() {
        testPublic();
        testInternal();
        // testPrivate();
        this.testExternal();
    }
}

contract Pig {
    function call() {
        Animal animal = new Animal();
        animal.testPublic();
        // animal.testInternal();
        // animal.testPrivate();
        animal.testExternal();
    }
}
```

状态的访问权限

```
pragma solidity ^0.4.19;

contract Animal {
    int public a = 1;
    int private b = 2;
    int internal c = 3;
    int d = 4;
}

contract Dog is Animal {
    function Dog() {
        a = 11;
        // b = 12;
        c = 13;
        d = 14;
    }
```

```
    // 继承 public、internal、external 类型的状态变量
}
```

pure、view、constant、payable的区别

`solidity`函数的完整声明格式为：

```
function 函数名(参数)  public|private|internal|external  pure|view|constant|payable 无返回值|returns (返回值类型)
```

```
pragma solidity ^0.4.24;

contract Animal {
    string name = "kong yi xue yuan";

    // pure
    function getAge() public pure returns (uint) {
        return 30;
    }

    // constant/view
    function getName() public view returns (string) {
        return name;
    }

    // constant/view
    function getCurrentAddress() public constant returns (address) {
        return msg.sender;
    }

    function setName(string str) public {
        name = str;
    }

    function getMsgValue() payable returns (uint) {
        //使用msg.value所在方法若不用payabel标记也能编译通过
        //但是msg.value有值时运行会报错，为零时不会报错。
        return msg.value;
    }
}
```

结论如下：

- 只有当函数有返回值的情况下，才需要使用 `pure`、 `view`、 `constant`

- `pure`: 当函数返回值为自变量而非变量时，使用 `pure`
- `view`: 当函数返回值为全局变量或属性时，使用 `view`
- `constant`: 与 `view`是等价的，constant是view的别名。
- `payable`： 使用msg.value的所在方法若不用payabel标记也能编译通过。但是msg.value有值时运行会报错，为零时不会报错。

如果一个函数有返回值，函数中正常来讲需要有 `pure`、 `view`或 `constant`关键字，如果没有，在调用函数的过程中，需要主动去调用底层的call方法。

注： 如果一个函数中带了关键字 `view`或 `constant`，就不能修改状态变量的值。但凡是是带了这两个关键字，区块链就默认只是向区块链读取数据，读取数据不需要花gas，但是不花gas就不可能修改状态变量的值。写入数据或者是修改状态变量的值都需要花费gas。

memory、storage

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）。

值类型的变量，只能深拷贝。

详细内容将在后面讲述。