Solidity 源文件结构

源文件中可以包含任意多个合约定义、导入指令和杂注指令。

版本杂注

为了避免未来被可能引入不兼容变更的编译器所编译，源文件可以（也应该）被所谓的版本 所注解。 我们力图把这类变更做到尽可能小，特别是，我们需要以一种当修改语义时必须同步修改语法的方式引入变更，当然这有时候也难以做到。 因此，至少对含重大变更的版本，通读变更日志永远是好办法。 这些版本的版本号始终是 `0.x.0` 或者 `x.0.0` 的形式。

版本杂注使用如下:

```
pragma solidity ^0.4.0;
```

这样，源文件将既不允许低于 0.4.0 版本的编译器编译， 也不允许高于（包含） `0.5.0` 版本的编译器编译（第二个条件因使用 `^` 被添加）。 这种做法的考虑是，编译器在 0.5.0 版本之前不会有重大变更，所以可确保源代码始终按预期被编译。 上面例子中不固定编译器的具体版本号，因此编译器的补丁版也可以使用。

可以使用更复杂的规则来指定编译器的版本，表达式遵循 npm 版本语义。

注解

Pragma 是 pragmatic information 的简称，微软 Visual C++ 文档 中译为杂注。Solidity 中沿用 C ，C++ 等中的编译指令概念，用于告知编译器 **如何** 编译。 ——译者注
导入其他源文件

语法与语义

虽然 Solidity 不知道 "default export" 为何物， 但是 Solidity 所支持的导入语句，其语法同 JavaScript（从 ES6 起）非常类似。

注解

ES6 即 ECMAScript 6.0，ES6 是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布。 ——译者注
在全局层面上，可使用如下格式的导入语句：

```
import "filename";
```

此语句将从 “filename” 中导入所有的全局符号到当前全局作用域中（不同于 ES6，Solidity 是向后兼容的）。

```
import * as symbolName from "filename";
```

...创建一个新的全局符号 `symbolName`，其成员均来自 `"filename"` 中全局符号。

```
import {symbol1 as alias, symbol2} from "filename";
```

...创建新的全局符号 `alias` 和 `symbol2`，分别从 `"filename"` 引用 `symbol1` 和 `symbol2` 。

另一种语法不属于 ES6，但或许更简便：

```
import "filename" as symbolName;
```

这条语句等同于 `import * as symbolName from "filename";`。

路径

上文中的 filename 总是会按路径来处理，以 `/` 作为目录分割符、以 `.` 标示当前目录、以 `..` 表示父目录。 当 `.` 或 `..` 后面跟随的字符是 `/` 时，它们才能被当做当前目录或父目录。 只有路径以当前目录 `.` 或父目录 `..` 开头时，才能被视为相对路径。

用 `import "./x" as x;` 语句导入当前源文件同目录下的文件 `x` 。 如果用 `import "x" as x;` 代替，可能会引入不同的文件（在全局 `include directory` 中）。

最终导入哪个文件取决于编译器（见下文）到底是怎样解析路径的。 通常，目录层次不必严格映射到本地文件系统， 它也可以映射到能通过诸如 ipfs，http 或者 git 发现的资源。

在实际的编译器中使用

当运行编译器时，它不仅能指定如何发现路径的第一个元素，还可指定路径前缀 。 例如，`github.com/ethereum/dapp-bin/library` 会被重映射到 `/usr/local/dapp-bin/library` ， 此时编译器将从重映射位置读取文件。如果重映射到多个路径，优先尝试重映射路径最长的一个。 这允许将比如 `""` 被映射到 `"/usr/local/include/solidity"` 来进行“回退重映射”。 同时，这些重映射可取决于上下文，允许你配置要导入的包，比如同一个库的不同版本。

solc:

对于 solc（命令行编译器），这些重映射以 `context:prefix=target` 形式的参数提供。 其中，`context:` 和 `=target` 部分是可选的（此时 target 默认为 prefix ）。 所有重映射的值都是被编译过的常规文件（包括他们的依赖），这个机制完全是向后兼容的（只要文件名不包含 = 或 : ）， 因此这不是一个破坏性修改。
在 `content` 目录或其子目录中的源码文件中，所有导入语句里以 `prefix` 开头的导入文件都将被用 `target` 替换 `prefix` 来重定向。

举个例子，如果你已克隆 `github.com/ethereum/dapp-bin/` 到本地 `/usr/local/dapp-bin` ，可在源文件中使用：

```
import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;
```

然后运行编译器：

```
solc github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ source.sol
```

举个更复杂的例子，假设你依赖了一些使用了非常旧版本的 dapp-bin 的模块。旧版本的 dapp-bin 已经被 checkout 到 `/usr/local/dapp-bin_old` ，此时你可使用：

```
solc module1:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin/ \
module2:github.com/ethereum/dapp-bin/=/usr/local/dapp-bin_old/ \
source.sol
```

这样， `module2` 中的所有导入都指向旧版本，而 `module1` 中的导入则获取新版本。

注意， solc 只允许包含来自特定目录的文件：它们必须位于显式地指定的源文件目录（或子目录）中，或者重映射的目标目录（或子目录）中。 如果你想直接用绝对路径来包含文件，只需添加重映射 `=/`。

如果有多个重映射指向一个有效文件，那么具有最长公共前缀的重映射会被选用。

Remix:

Remix 提供一个为 github 源代码平台的自动重映射，它将通过网络自动获取文件： 比如，你可以使用 `import "github.com/ethereum/dapp-bin/library/iterable_mapping.sol" as it_mapping;` 导入一个 map 迭代器。

未来， Remix 可能支持其他源代码平台。

注释

可以使用单行注释（`//`）和多行注释（`/*...*/`）

```
// 这是一个单行注释。

/*
这是一个
多行注释。
*/
```

此外，有另一种注释称为 natspec 注释，其文档还尚未编写。 它们是用三个反斜杠（`///`）或双星号开头的块（`/** ... */`）书写，它们应该直接在函数声明或语句上使用。 可在注释中使用 Doxygen 样式的标签来文档化函数、 标注形式校验通过的条件，和提供一个当用户试图调用一个函数时显示给用户的 **确认文本**。

在下面的例子中，我们记录了合约的标题、两个入参和两个返回值的说明：

```
pragma solidity ^0.4.0;
```

```
/** @title 形状计算器。  */
contract shapeCalculator {
    /** @dev 求矩形表明面积与周长。
    * @param w 矩形宽度。
    * @param h 矩形高度。
    * @return s 求得表面积。
    * @return p 求得周长。
    */
    function rectangle(uint w, uint h) returns (uint s, uint p) {
        s = w * h;
        p = 2 * (w + h);
    }
}
```

Next Previous

合约结构

在 Solidity 中，合约类似于面向对象编程语言中的类。 每个合约中可以包含 状态变量、 函数、 函数修饰器、事件、 结构类型、 和 枚举类型 的声明，且合约可以从其他合约继承。

状态变量

状态变量是永久地存储在合约存储中的值。

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData; // 状态变量
    // ...
}
```

有效的状态变量类型参阅 类型 章节， 对状态变量可见性有可能的选择参阅 可见性和 getter 函数 。

函数

函数是合约中代码的可执行单元。

```
pragma solidity ^0.4.0;

contract SimpleAuction {
    function bid() public payable { // 函数
        // ...
    }
}
```

函数调用 可发生在合约内部或外部，且函数对其他合约有不同程度的可见性（可见性和 getter 函数）。

函数修饰器

函数修饰器可以用来以声明的方式改良函数语义（参阅合约章节中 函数 ）。

```
pragma solidity ^0.4.22;

contract Purchase {
    address public seller;

    modifier onlySeller() { // 修饰器
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    function abort() public onlySeller { // Modifier usage
        // ...
    }
}
```

事件

事件是能方便地调用以太坊虚拟机日志功能的接口。

```
pragma solidity ^0.4.21;
contract SimpleAuction {
```

```
    event HighestBidIncreased(address bidder, uint amount); // 事件

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); // 触发事件
    }
}
```

有关如何声明事件和如何在 dapp 中使用事件的信息，参阅合约章节中的 事件。

结构类型

结构是可以将几个变量分组的自定义类型（参阅类型章节中的 结构体）。

```
pragma solidity ^0.4.0;

contract Ballot {
    struct Voter { // 结构
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

枚举类型

枚举可用来创建由一定数量的“常量值”构成的自定义类型（参阅类型章节中的 枚举类型）。

```
pragma solidity ^0.4.0;

contract Purchase {
    enum State { Created, Locked, Inactive } // 枚举
```

值类型

以下类型也称为值类型，因为这些类型的变量将始终按值来传递。 也就是说，当这些变量被用作函数参数或者用在赋值语句中时，总会进行值拷贝。

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）

运算符 `||` 和 `&&` 都遵循同样的短路（ short-circuiting ）规则。就是说在表达式 `f(x) || g(y)`中， 如果 `f(x)` 的值为 `true` ，那么 `g(y)` 就不会被执行，即使会出现一些副作用。

整型

`int` / `uint` ： 分别表示有符号和无符号的不同位数的整型变量。 支持关键字 `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）

除法总是会截断的（仅被编译为 EVM 中的 `DIV` 操作码）， 但如果操作数都是字面常数（literals） （或者字面常数表达式），则不会截断。

除以零或者模零运算都会引发运行时异常。

移位运算的结果取决于运算符左边的类型。 表达式 `x << y` 与 `x * 2**y` 是等价的，`x >> y` 与 `x / 2**y` 是等价的。这意味对一个负数进行移位会导致其符号消失。 按负数位移动会引发运行时异常。

警告

由有符号整数类型负值右移所产生的结果跟其它语言中所产生的结果是不同的。 在 Solidity 中，右移和除是等价的，因此对一个负数进行右移操作会导致向 0 的取整（截断）。 而在其它语言中， 对负数进行右移类似于（向负无穷）取整。
定长浮点型

警告

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。 在关键字 `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）

注解

浮点型（在许多语言中的 `float` 和 `double` 类型，更准确地说是 IEEE 754 类型）和定长浮点型之间最大的不同点是， 在前者中整数部分和小数部分（小数点后的部分）需要的位数是灵活可变的，而后者中这两部分的长度受到严格的规定。 一般来说，在浮点型中，几乎整个空间都用来表示数字，但只有少数的位来表示小数点的位置。
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 `>`

注解

从 0.5.0 版本开始，合约不会从地址类型派生，但仍然可以显式地转换成地址类型。

地址类型成员变量

- `balance` 和 `transfer`

快速参考，请见 地址相关。

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 x = 0x123;
address myAddress = this;
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

注解

如果 `x` 是一个合约地址，它的代码（更具体来说是它的 fallback 函数，如果有的话）会跟 `transfer` 函数调用一起执行（这是 EVM 的一个特性，无法阻止）。 如果在执行过程中用光了 gas 或者因为任何原因执行失败， 交易会被打回，当前的合约也会在终止的同时抛出异常。

- `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`。

警告

在使用 `send` 的时候会有些风险：如果调用栈深度是 1024 会导致发送失败（这总是可以被调用者强制），如果接收者用光了 gas 也会导致发送失败。 所以为了保证 发送的安全，一定要检查 `send` 的返回值，使用 `transfer` 或者更好的办法：使用一种接收者可以取回资金的模式。

- `call`， `callcode` 和 `delegatecall`

此外，为了与不符合 的合约交互，于是就有了可以接受任意类型任意数量参数的 `call` 函数。 这些参数会被打包到以 32 字节为单位的连续区域中存放。 其中一个例外是当第一个参数被编码成正好 4 个字节的情况。 在这种情况下，这个参数后边不会填充后续参数编码，以允许使用函数签名。

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
```

```
nameReg.call("register", "MyName");
nameReg.call(bytes4(keccak256("fun(uint256)")), a);
```

call 返回的布尔值表明了被调用的函数已经执行完毕（true）或者引发了一个 EVM 异常（false）。 无法访问返回的真实数据（为此我们需要事先知道编码和大小）。

可以使用 .gas() 调整提供的 gas 数量

```
namReg.call.gas(1000000)("register", "MyName");
```

类似地，也能控制提供的 的值

```
nameReg.call.value(1 ether)("register", "MyName");
```

最后一点，这些 可以联合使用。每个修改器出现的顺序不重要

```
nameReg.call.gas(1000000).value(1 ether)("register", "MyName");
```

注解

目前还不能在重载函数中使用 gas 或者 value 。

一种解决方案是给 gas 和值引入一个特例，并重新检查它们是否在重载的地方出现。
类似地，也可以使用 delegatecall： 区别在于只使用给定地址的代码，其它属性（存储，余额，.....）都取自当前合约。 delegatecall 的目的是使用存储在另外一个合约中的库代码。 用户必须确保两个合约中的存储结构都适用于 delegatecall。 在 homestead 版本之前，只有一个功能类似但作用有限的 callcode 的函数可用，但它不能获取委托方的 msg.sender 和 msg.value。

这三个函数 call， delegatecall 和 callcode 都是非常低级的函数，应该只把它们当作最后一招 来使用，因为它们破坏了 Solidity 的类型安全性。

注解

所有合约都继承了地址（address）的成员变量，因此可以使用 this.balance 查询当前合约的余额。

注解

不鼓励使用 callcode，在未来也会将其移除。

警告

这三个函数都属于低级函数，需要谨慎使用。 具体来说，任何未知的合约都可能是恶意的。 你在调用一个合约的同时就将控制权交给了它，它可以反过来调用你的合约， 因此，当调用返回时要为你的状态变量的改变做好准备。
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）。

该类型可以和作为右操作数的任何整数类型进行移位运算（但返回结果的类型和左操作数类型相同），右操作数表示需要移动的位数。 进行负数位移运算会引发运行时异常。

成员变量：

- `.length` 表示这个字节数组的长度（只读）.

注解

可以将 `byte[]` 当作字节数组使用，但这种方式非常浪费存储空间，准确来说，是在传入调用时，每个元素会浪费 31 字节。 更好地做法是使用 `bytes`。
变长字节数组

`bytes`:

变长字节数组，参见 数组。它并不是值类型。

`string`:

变长 UTF-8 编码字符串类型，参见 数组。并不是值类型。

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。

注解

混合大小写的地址校验和格式定义在 EIP-55 中。
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。

例如， `(2**800 + 1) - 2**800` 的结果是字面常数 `1` （属于 `uint8` 类型），尽管计算的中间结果已经超过了 的机器字长度。 此外， `.5 * 8` 的结果是整型 `4` （尽管有非整型参与了计算）。

只要操作数是整型，任意整型支持的运算符都可以被运用在数值字面常数表达式中。 如果两个中的任一个数是小数，则不允许进行位运算。如果指数是小数的话，也不支持幂运算（因为这样可能会得到一个无理数）。

注解

Solidity 对每个有理数都有对应的数值字面常数类型。 整数字面常数和有理数字面常数都属于数值字面常数类型。 除此之外，所有的数值字面常数表达式（即只包含数值字面常数和运算符的表达式）都属于数值字面常数类型。 因此数值字面常数表达式 `1 + 2` 和 `2 + 1` 的结果跟有理数三的数值字面常数类型相同。

警告

在早期版本中，整数字面常数的除法也会截断，但在现在的版本中，会将结果转换成一个有理数。即 `5 / 2` 并不等于 `2`，而是等于 `2.5`。

注解

数值字面常数表达式只要在非字面常数表达式中使用就会转换成非字面常数类型。 在下面的例子中，尽管我们知道 `b` 的值是一个整数，但 `2.5 + a` 这部分表达式并不进行类型检查，因此编译不能通过。

```
uint128 a = 1;
```

```
uint128 b = 2.5 + a + 0.5;
```

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。

十六进制字面常数跟字符串字面常数很类似，具有相同的转换规则。

枚举类型

```
pragma solidity ^0.4.16;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // 由于枚举类型不属于 |ABI| 的一部分，因此对于所有来自 Solidity 外部的调用，
    // "getChoice" 的签名会自动被改成 "getChoice() returns (uint8)"。
    // 整数类型的大小已经足够存储所有枚举类型的值，随着值的个数增加，
    // 可以逐渐使用 `uint16` 或更大的整数类型。
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }
```

```
    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

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：

内部函数只能在当前合约内被调用（更具体来说，在当前代码块内，包括内部库函数和继承的函数中），因为它们不能在当前合约上下文的外部被执行。 调用一个内部函数是通过跳转到它的入口标签来实现的，就像在当前合约的内部调用一个函数。

外部函数由一个地址和一个函数签名组成，可以通过外部函数调用传递或者返回。

函数类型表示成如下的形式

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

与参数类型相反，返回类型不能为空 —— 如果函数类型不需要返回，则需要删除整个 `returns (<return types>)` 部分。

函数类型默认是内部函数，因此不需要声明 `internal` 关键字。 与此相反的是，合约中的函数本身默认是 public 的，只有当它被当做类型名称时，默认才是内部函数。

有两种方法可以访问当前合约中的函数：一种是直接使用它的名字，`f` ，另一种是使用 `this.f` 。 前者适用于内部函数，后者适用于外部函数。

如果当函数类型的变量还没有初始化时就调用它的话会引发一个异常。 如果在一个函数被 `delete` 之后调用它也会发生相同的情况。

如果外部函数类型在 Solidity 的上下文环境以外的地方使用，它们会被视为 `function` 类型。 该类型将函数地址紧跟其函数标识一起编码为一个 `bytes24` 类型。。

请注意，当前合约的 public 函数既可以被当作内部函数也可以被当作外部函数使用。 如果想将一个函数当作内部函数使用，就用 `f` 调用，如果想将其当作外部函数，使用 `this.f` 。

除此之外，public（或 external）函数也有一个特殊的成员变量称作 `selector`，可以返回 ABI 函数选择器:

```
pragma solidity ^0.4.16;

contract Selector {
    function f() public view returns (bytes4) {
        return this.f.selector;
    }
}
```

如果使用内部函数类型的例子:

```
pragma solidity ^0.4.16;

library ArrayUtils {
    // 内部函数可以在内部库函数中使用，
    // 因为它们会成为同一代码上下文的一部分
    function map(uint[] memory self, function (uint) pure returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }
    function reduce(
        uint[] memory self,
        function (uint, uint) pure returns (uint) f
    )
        internal
        pure
```

```
        returns (uint r)
    {
        r = self[0];
        for (uint i = 1; i < self.length; i++) {
            r = f(r, self[i]);
        }
    }
    function range(uint length) internal pure returns (uint[] memory r) {
        r = new uint[](length);
        for (uint i = 0; i < r.length; i++) {
            r[i] = i;
        }
    }
}

contract Pyramid {
    using ArrayUtils for *;
    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }
    function square(uint x) internal pure returns (uint) {
        return x * x;
    }
    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

另外一个使用外部函数类型的例子:

```
pragma solidity ^0.4.11;

contract Oracle {
    struct Request {
        bytes data;
        function(bytes memory) external callback;
    }
```

```
    Request[] requests;
    event NewRequest(uint);
    function query(bytes data, function(bytes memory) external callback) public {
        requests.push(Request(data, callback));
        NewRequest(requests.length - 1);
    }
    function reply(uint requestID, bytes response) public {
        // 这里要验证  reply 来自可信的源
        requests[requestID].callback(response);
    }
}

contract OracleUser {
    Oracle constant oracle = Oracle(0x1234567); // 已知的合约
    function buySomething() {
        oracle.query("USD", this.oracleResponse);
    }
    function oracleResponse(bytes response) public {
        require(msg.sender == address(oracle));
        // 使用数据
    }
}
```

注解

Lambda 表达式或者内联函数的引入在计划内，但目前还没支持。
引用类型

比起之前讨论过的值类型，在处理复杂的类型（即占用的空间超过 256 位的类型）时，我们需要更加谨慎。 由于拷贝这些类型变量的开销相当大，我们不得不考虑它的存储位置，是将它们保存在 ** ** （并不是永久存储）中， 还是 ** ** （保存状态变量的地方）中。

数据位置

所有的复杂类型，即 数组 和 结构 类型，都有一个额外属性，“数据位置”，说明数据是保存在 中还是 中。 根据上下文不同，大多数时候数据有默认的位置，但也可以通过在类型名后增加关键字 `storage` 或 `memory` 进行修改。 函数参数

（包括返回的参数）的数据位置默认是 `memory`， 局部变量的数据位置默认是 `storage`，状态变量的数据位置强制是 `storage` （这是显而易见的）。

也存在第三种数据位置， `calldata` ，这是一块只读的，且不会永久存储的位置，用来存储函数参数。 外部函数的参数（非返回参数）的数据位置被强制指定为 `calldata` ，效果跟 `memory` 差不多。

数据位置的指定非常重要，因为它们影响着赋值行为： 在 和 之间两两赋值，或者 向状态变量（甚至是从其它状态变量）赋值都会创建一份独立的拷贝。 然而状态变量向局部变量赋值时仅仅传递一个引用，而且这个引用总是指向状态变量，因此后者改变的同时前者也会发生改变。 另一方面，从一个 存储的引用类型向另一个 存储的引用类型赋值并不会创建拷贝。

```
pragma solidity ^0.4.0;

contract C {
    uint[] x; // x 的数据存储位置是 storage

    // memoryArray 的数据存储位置是 memory
    function f(uint[] memoryArray) public {
        x = memoryArray; // 将整个数组拷贝到 storage 中，可行
        var y = x; // 分配一个指针（其中 y 的数据存储位置是 storage），可行
        y[7]; // 返回第 8 个元素，可行
        y.length = 2; // 通过 y 修改 x，可行
        delete x; // 清除数组，同时修改 y，可行
        // 下面的就不可行了；需要在 storage 中创建新的未命名的临时数组， /
        // 但 storage 是“静态”分配的：
        // y = memoryArray;
        // 下面这一行也不可行，因为这会“重置”指针，
        // 但并没有可以让它指向的合适的存储位置。
        // delete y;

        g(x); // 调用 g 函数，同时移交对 x 的引用
        h(x); // 调用 h 函数，同时在 memory 中创建一个独立的临时拷贝
    }

    function g(uint[] storage storageArray) internal {}
```

```
    function h(uint[] memoryArray) public {}
}
```

总结

强制指定的数据位置：

- 外部函数的参数（不包括返回参数）： calldata
- 状态变量： storage

默认数据位置：

- 函数参数（包括返回参数）： memory
- 所有其它局部变量： storage

数组

数组可以在声明时指定长度，也可以动态调整大小。 对于 的数组来说，元素类型可以是任意的（即元素也可以是数组类型，映射类型或者结构体）。 对于 的数组来说，元素类型不能是映射类型，如果作为 public 函数的参数，它只能是 ABI 类型。

一个元素类型为 `T`，固定长度为 `k` 的数组可以声明为 `T[k]`，而动态数组声明为 `T[]`。 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]` （注意这里跟其它语言比，数组长度的声明位置是反的）。 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）。。

`bytes` 和 `string` 类型的变量是特殊的数组。 `bytes` 类似于 `byte[]`，但它在 calldata 中会被“紧打包”（译者注：将元素连续地存在一起，不会按每 32 字节一单元的方式来存放）。 `string` 与 `bytes` 相同，但（暂时）不允许用长度或索引来访问。

注解

如果想要访问以字节表示的字符串 `s`，请使用 `bytes(s).length` / `bytes(s)[7] = 'x';`。 注意这时你访问的是 UTF-8 形式的低级 bytes 类型，而不是单个的字符。
可以将数组标识为 `public`，从而让 Solidity 创建一个 getter。 之后必须使用数字下标作为参数来访问 getter。

创建内存数组

可使用 `new` 关键字在内存中创建变长数组。 与 数组相反的是，你 不能 通过修改成员变量 `.length` 改变 数组的大小。

```
pragma solidity ^0.4.16;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        // 这里我们有 a.length == 7 以及 b.length == len
        a[6] = 8;
    }
}
```

数组字面常数 / 内联数组

数组字面常数是写作表达式形式的数组，并且不会立即赋值给变量。

```
pragma solidity ^0.4.16;

contract C {
    function f() public pure {
        g([uint(1), 2, 3]);
    }
    function g(uint[3] _data) public pure {
        // ...
    }
}
```

数组字面常数是一种定长的 数组类型，它的基础类型由其中元素的普通类型决定。 例如，`[1, 2, 3]` 的类型是 `uint8[3] memory`，因为其中的每个字面常数的类型都是 `uint8`。 正因为如此，有必要将上面这个例子中的第一个元素转换成 `uint` 类型。 目前需要注意的是，定长的 数组并不能赋值给变长的 数组，下面是个反例：

```
// 这段代码并不能编译。

pragma solidity ^0.4.0;
```

```
contract C {
    function f() public {
        // 这一行引发了一个类型错误，因为 unint[3] memory
        // 不能转换成 uint[] memory。
        uint[] x = [uint(1), 3, 4];
    }
}
```

已经计划在未来移除这样的限制，但目前数组在 ABI 中传递的问题造成了一些麻烦。

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。

警告

在外部函数中目前还不能使用多维数组。

警告

由于 的限制，不能通过外部函数调用返回动态的内容。 例如，如果通过 web3.js 调用 `contract C { function f() returns (uint[]) { ... } }` 中的 `f` 函数，它会返回一些内容，但通过 Solidity 不可以。

目前唯一的变通方法是使用大型的静态数组。

```
pragma solidity ^0.4.16;

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // 注意下面的代码并不是一对动态数组，
    // 而是一个数组元素为一对变量的动态数组（也就是数组元素为长度为 2 的定长数组的动态数组）。
```

```
bool[2][] m_pairsOfFlags;
// newPairs 存储在 memory 中 —— 函数参数默认的存储位置

function setAllFlagPairs(bool[2][] newPairs) public {
    // 向一个 storage 的数组赋值会替代整个数组
    m_pairsOfFlags = newPairs;
}

function setFlagPair(uint index, bool flagA, bool flagB) public {
    // 访问一个不存在的数组下标会引发一个异常
    m_pairsOfFlags[index][0] = flagA;
    m_pairsOfFlags[index][1] = flagB;
}

function changeFlagArraySize(uint newSize) public {
    // 如果 newSize 更小，那么超出的元素会被清除
    m_pairsOfFlags.length = newSize;
}

function clear() public {
    // 这些代码会将数组全部清空
    delete m_pairsOfFlags;
    delete m_aLotOfIntegers;
    // 这里也是实现同样的功能
    m_pairsOfFlags.length = 0;
}

bytes m_byteData;

function byteArrays(bytes data) public {
    // 字节的数组（语言意义中的 byte 的复数 ``bytes``）不一样，因为它们不是填充式存储的，
    // 但可以当作和 "uint8[]" 一样对待
    m_byteData = data;
    m_byteData.length += 7;
    m_byteData[3] = byte(8);
    delete m_byteData[2];
```

```
    }

    function addFlag(bool[2] flag) public returns (uint) {
        return m_pairsOfFlags.push(flag);
    }

    function createMemoryArray(uint size) public pure returns (bytes) {
        //使用 `new` 创建动态 memory 数组：
        uint[2][] memory arrayOfPairs = new uint[2][](size);
        //创建一个动态字节数组：
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = byte(i);
        return b;
    }
}
```

结构体

Solidity 支持通过构造结构体的形式定义新的类型，以下是一个结构体使用的示例：

```
pragma solidity ^0.4.11;

contract CrowdFunding {
    // 定义的新类型包含两个属性。
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }
```

```
    uint numCampaigns;
    mapping (uint => Campaign) campaigns;

    function newCampaign(address beneficiary, uint goal) public returns (uint campaignID) {
        campaignID = numCampaigns++; // campaignID 作为一个变量返回
        // 创建新的结构体示例，存储在 storage 中。我们先不关注映射类型。
        campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0);
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // 以给定的值初始化，创建一个新的临时 memory 结构体，
        // 并将其拷贝到 storage 中。
        // 注意你也可以使用 Funder(msg.sender, msg.value) 来初始化。
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
        c.amount += msg.value;
    }

    function checkGoalReached(uint campaignID) public returns (bool reached) {
        Campaign storage c = campaigns[campaignID];
        if (c.amount < c.fundingGoal)
            return false;
        uint amount = c.amount;
        c.amount = 0;
        c.beneficiary.transfer(amount);
        return true;
    }
}
```

上面的合约只是一个简化版的众筹合约，但它已经足以让我们理解结构体的基础概念。 结构体类型可以作为元素用在映射和数组中，其自身也可以包含映射和数组作为成员变量。

尽管结构体本身可以作为映射的值类型成员，但它并不能包含自身。 这个限制是有必要的，因为结构体的大小必须是有限的。

注意在函数中使用结构体时，一个结构体是如何赋值给一个局部变量（默认存储位置是 ）的。 在这个过程中并没有拷贝这个结构体，而是保存一个引用，所以对局部变量成员的赋值实际上会被写入状态。

当然，你也可以直接访问结构体的成员而不用将其赋值给一个局部变量，就像这样，`campaigns[campaignID].amount = 0`。

映射

映射类型在声明时的形式为 `mapping(_KeyType => _ValueType)`。 其中 `_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。

映射可以视作 哈希表 <https://en.wikipedia.org/wiki/Hash_table>，它们在实际的初始化过程中创建每个可能的 key， 并将其映射到字节形式全是零的值：一个类型的 默认值。然而下面是映射与哈希表不同的地方： 在映射中，实际上并不存储 key，而是存储它的 `keccak256` 哈希值，从而便于查询实际的值。

正因为如此，映射是没有长度的，也没有 key 的集合或 value 的集合的概念。

只有状态变量（或者在 internal 函数中的对于存储变量的引用）可以使用映射类型。。

可以将映射声明为 `public`，然后来让 Solidity 创建一个 getter。 `_KeyType` 将成为 getter 的必须参数，并且 getter 会返回 `_ValueType`。

`_ValueType` 也可以是一个映射。这时在使用 getter 时将将需要递归地传入每个 `_KeyType` 参数。

```
pragma solidity ^0.4.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}

contract MappingUser {
```

```
    function f() public returns (uint) {
        MappingExample m = new MappingExample();
        m.update(100);
        return m.balances(this);
    }
}
```

注解

映射不支持迭代，但可以在此之上实现一个这样的数据结构。 例子可以参考可迭代的映射。

涉及 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。

`delete` 对整个映射是无效的（因为映射的键可以是任意的，通常也是未知的）。因此在你删除一个结构体时，结果将重置所有的非映射属性，这个过程是递归进行的，除非它们是映射。 然而，单个的键及其映射的值是可以被删除的。

理解 `delete a` 的效果就像是给 `a` 赋值很重要，换句话说，这相当于在 `a` 中存储了一个新的对象。

```
pragma solidity ^0.4.0;

contract DeleteExample {
    uint data;
```

```
    uint[] dataArray;

    function f() public {
        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，
        // 因为它是一个存储位置是 storage 的对象的别名。
        // 另一方面："delete y" 是非法的，引用了 storage 对象的局部变量只能由已有的 storage 对象
赋值。
    }
}
```

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`。

警告

类型只能从第一次赋值中推断出来，因此以下代码中的循环是无限的， 原因是 ``i`` 的类型是 `uint8`，而这个类型变量的最大值比 `2000` 小。 `for (var i = 0; i < 2000; i++) { ... }`

单元和全局变量

单位

单位之间的换算就是在数字后边加上 `wei`、 `finney`、 `szabo` 或 `ether` 来实现的，如果后面没有单位，缺省为 Wei。例如 `2 ether == 2000 finney` 的逻辑判断值为 `true`。

时间单位

秒是缺省时间单位，在时间单位之间，数字后面带有 `seconds`、 `minutes`、 `hours`、 `days`、 `weeks` 和 `years` 的可以进行换算，基本换算关系如下：

- `1 == 1 seconds`
- `1 minutes == 60 seconds`
- `1 hours == 60 minutes`
- `1 days == 24 hours`
- `1 weeks == 7 days`
- `1 years == 365 days`

由于闰秒造成的每年不都是 365 天、每天不都是 24 小时 leap seconds，所以如果你要使用这些单位计算日期和时间，请注意这个问题。因为闰秒是无法预测的，所以需要借助外部的预言机（oracle，是一种链外数据服务，译者注）来对一个确定的日期代码库进行时间矫正。

注解

`years` 后缀已经不推荐使用了，因为从 0.5.0 版本开始将不再支持。
这些后缀不能直接用在变量后边。如果想用时间单位（例如 days）来将输入变量换算为时间，你可以用如下方式来完成：

```
function f(uint start, uint daysAfter) public {
        if (now >= start + daysAfter * 1 days) {
                //...
        }
}
```

特殊变量和函数

在全局命名空间中已经存在了（预设了）一些特殊的变量和函数，他们主要用来提供关于区块链的信息或一些通用的工具函数。

区块和交易属性

- `block.blockhash(uint blockNumber) returns (bytes32)`：指定区块的区块哈希——仅可用于最新的 256 个区块且不包括当前区块；而 blocks 从 0.4.22 版本开始已经不推荐使用，由 `blockhash(uint blockNumber)` 代替
- `block.coinbase` (`address`): 挖出当前区块的矿工地址
- `block.difficulty` (`uint`): 当前区块难度
- `block.gaslimit` (`uint`): 当前区块 gas 限额
- `block.number` (`uint`): 当前区块号
- `block.timestamp` (`uint`): 自 unix epoch 起始当前区块以秒计的时间戳
- `gasleft() returns (uint256)`：剩余的 gas
- `msg.data` (`bytes`): 完整的 calldata

- `msg.gas` (`uint`): 剩余 gas - 自 0.4.21 版本开始已经不推荐使用，由 `gesleft()` 代替
- `msg.sender` (`address`): 消息发送者（当前调用）
- `msg.sig` (`bytes4`): calldata 的前 4 字节（也就是函数标识符）
- `msg.value` (`uint`): 随消息发送的 wei 的数量
- `now` (`uint`): 目前区块时间戳（`block.timestamp`）
- `tx.gasprice` (`uint`): 交易的 gas 价格
- `tx.origin` (`address`): 交易发起者（完全的调用链）

注解

对于每一个**外部函数**调用，包括 `msg.sender` 和 `msg.value` 在内所有 `msg` 成员的值都会变化。这里包括对库函数的调用。

注解

不要依赖 `block.timestamp`、 `now` 和 `blockhash` 产生随机数，除非你知道自己在做什么。

时间戳和区块哈希在一定程度上都可能受到挖矿矿工影响。例如，挖矿社区中的恶意矿工可以用某个给定的哈希来运行赌场合约的 payout 函数，而如果他们没收到钱，还可以用一个不同的哈希重新尝试。

当前区块的时间戳必须严格大于最后一个区块的时间戳，但这里唯一能确保的只是它会是在权威链上的两个连续区块的时间戳之间的数值。

注解

基于可扩展因素，区块哈希不是对所有区块都有效。你仅仅可以访问最近 256 个区块的哈希，其余的哈希均为零。

ABI 编码函数

- `abi.encode(...) returns (bytes)`： ABI - 对给定参数进行编码
- `abi.encodePacked(...) returns (bytes)`：对给定参数执行 紧打包编码
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes)`： ABI - 对给定参数进行编码，并以给定的函数选择器作为起始的 4 字节数据一起返回

- `abi.encodeWithSignature(string signature, ...) returns (bytes)`：等价于 `abi.encodeWithSelector(bytes4(keccak256(signature), ...)`

注解

这些编码函数可以用来构造函数调用数据，而不用实际进行调用。此外，`keccak256(abi.encodePacked(a, b))` 是更准确的方法来计算在未来版本不推荐使用的 `keccak256(a, b)`。
更多详情请参考 ABI 和 紧打包编码。

错误处理

`assert(bool condition)`:

> 如果条件不满足，则使当前交易没有效果 — 用于检查内部错误。

`require(bool condition)`:

> 如果条件不满足则撤销状态更改 - 用于检查由输入或者外部组件引起的错误。

`require(bool condition, string message)`:

> 如果条件不满足则撤销状态更改 - 用于检查由输入或者外部组件引起的错误，可以同时提供一个错误消息。

`revert()`:

> 终止运行并撤销状态更改。

`revert(string reason)`:

> 终止运行并撤销状态更改，可以同时提供一个解释性的字符串。

数学和密码学函数

`addmod(uint x, uint y, uint k) returns (uint)`:

> 计算 `(x + y) % k`，加法会在任意精度下执行，并且加法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本的编译器开始会加入对 `k != 0` 的校验（assert）。

`mulmod(uint x, uint y, uint k) returns (uint)`:

> 计算 `(x * y) % k`，乘法会在任意精度下执行，并且乘法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本的编译器开始会加入对 `k != 0` 的校验（assert）。

`keccak256(...) returns (bytes32)`:

计算 (tightly packed) arguments 的 Ethereum-SHA-3 （Keccak-256）哈希。

sha256(...) returns (bytes32):

计算 (tightly packed) arguments 的 SHA-256 哈希。

sha3(...) returns (bytes32):

等价于 keccak256。

ripemd160(...) returns (bytes20):

计算 (tightly packed) arguments 的 RIPEMD-160 哈希。

ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)：

利用椭圆曲线签名恢复与公钥相关的地址，错误返回零值。 (example usage)

上文中的“tightly packed”是指不会对参数值进行 padding 处理（就是说所有参数值的字节码是连续存放的，译者注），这意味着下边这些调用都是等价的：

keccak256("ab", "c") keccak256("abc") keccak256(0x616263) keccak256(6382179) keccak256(97, 98, 99)

如果需要 padding，可以使用显式类型转换：`keccak256("\x00\x12")` 和 `keccak256(uint16(0x12))`是一样的。

请注意，常量值会使用存储它们所需要的最少字节数进行打包。例如：`keccak256(0) == keccak256(uint8(0))`， `keccak256(0x12345678) == keccak256(uint32(0x12345678))`。

在一个私链上，你很有可能碰到由于 `sha256`、 `ripemd160` 或者 `ecrecover` 引起的 Out-of-Gas。这个原因就是他们被当做所谓的预编译合约而执行，并且在第一次收到消息后这些合约才真正存在（尽管合约代码是硬代码）。发送到不存在的合约的消息非常昂贵，所以实际的执行会导致 Out-of-Gas 错误。在你的合约中实际使用它们之前，给每个合约发送一点儿以太币，比如 1 Wei。这在官方网络或测试网络上不是问题。

地址相关

<address>.balance (uint256):

以 Wei 为单位的 地址类型 的余额。

<address>.transfer(uint256 amount):

向 地址类型 发送数量为 amount 的 Wei，失败时抛出异常，发送 2300 gas 的矿工费，不可调节。

<address>.send(uint256 amount) returns (bool):

向 地址类型 发送数量为 amount 的 Wei，失败时返回 false，发送 2300 gas 的矿工费用，不可调节。

<address>.call(...) returns (bool):

发出低级函数 CALL，失败时返回 false，发送所有可用 gas，可调节。

<address>.callcode(...) returns (bool):

发出低级函数 CALLCODE，失败时返回 false，发送所有可用 gas，可调节。

<address>.delegatecall(...) returns (bool):

发出低级函数 DELEGATECALL，失败时返回 false，发送所有可用 gas，可调节。

更多信息，参考 地址类型 部分：

警告

使用 send 有很多危险：如果调用栈深度已经达到 1024（这总是可以由调用者所强制指定），转账会失败；并且如果接收者用光了 gas，转账同样会失败。为了保证以太币转账安全，总是检查 send 的返回值，利用 transfer 或者下面更好的方式： 用这种接收者取回钱的模式。

注解

如果在通过低级函数 delegatecall 发起调用时需要访问存储中的变量，那么这两个合约的存储中的变量定义顺序需要一致，以便被调用的合约代码可以正确地通过变量名访问合约的存储变量。 这当然不是指像在高级的库函数调用时所传递的存储变量指针那样的情况。

注解

不鼓励使用 callcode，并且将来它会被移除。

合约相关

this (current contract's type):

当前合约，可以明确转换为 地址类型。

selfdestruct(address recipient):

销毁合约，并把余额发送到指定 地址类型。

`suicide(address recipient)`:

与 selfdestruct 等价，但已不推荐使用。

此外，当前合约内的所有函数都可以被直接调用，包括当前函数

由于 Solidity 有两种函数调用（内部调用不会产生实际的 EVM 调用或称为“消息调用”，而外部调用则会产生一个 EVM 调用）， 函数和状态变量有四种可见性类型。 函数可以指定为 `external`， `public` ， `internal` 或者 `private`，默认情况下函数类型为 `public`。 对于状态变量，不能设置为 `external` ，默认是 `internal` 。

`external`：

外部函数作为合约接口的一部分，意味着我们可以从其他合约和交易中调用。 一个外部函数 `f` 不能从内部调用（即 `f` 不起作用，但 `this.f()` 可以）。 当收到大量数据的时候，外部函数有时候会更有效率。

`public`：

public 函数是合约接口的一部分，可以在内部或通过消息调用。对于公共状态变量， 会自动生成一个 getter 函数（见下面）。

`internal`：

这些函数和状态变量只能是内部访问（即从当前合约内部或从它派生的合约访问），不使用 `this` 调用。

`private`：

private 函数和状态变量仅在当前定义它们的合约中使用，并且不能被派生合约使用

函数

View 函数

可以将函数声明为 `view` 类型，这种情况下要保证不修改状态。

下面的语句被认为是修改状态：

1. 修改状态变量。
2. 产生事件。
3. 创建其它合约。

4. 使用 `selfdestruct`。
5. 通过调用发送以太币。
6. 调用任何没有标记为 `view` 或者 `pure` 的函数。
7. 使用低级调用。
8. 使用包含特定操作码的内联汇编。

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public view returns (uint) {
        return a * (b + 42) + now;
    }
}
```

注解

`constant` 是 `view` 的别名。

注解

Getter 方法被标记为 `view`。

警告

编译器没有强制 `view` 方法不能修改状态。

Pure 函数

函数可以声明为 `pure` ，在这种情况下，承诺不读取或修改状态。

除了上面解释的状态修改语句列表之外，以下被认为是从状态中读取：

1. 读取状态变量。
2. 访问 `this.balance` 或者 `<address>.balance`。
3. 访问 `block`， `tx`， `msg` 中任意成员 （除 `msg.sig` 和 `msg.data` 之外）。
4. 调用任何未标记为 `pure` 的函数。

5. 使用包含某些操作码的内联汇编。

```
pragma solidity ^0.4.16;

contract C {
    function f(uint a, uint b) public pure returns (uint) {
        return a * (b + 42);
    }
}
```

警告

编译器没有强制 `pure` 方法不能读取状态。

Fallback 函数

合约可以有一个未命名的函数。这个函数不能有参数也不能有返回值。 如果在一个到合约的调用中，没有其他函数与给定的函数标识符匹配（或没有提供调用数据），那么这个函数（fallback 函数）会被执行。

除此之外，每当合约收到以太币（没有任何数据），这个函数就会执行。此外，为了接收以太币，fallback 函数必须标记为 `payable`。 如果不存在这样的函数，则合约不能通过常规交易接收以太币。

在这样的上下文中，通常只有很少的 gas 可以用来完成这个函数调用（准确地说，是 2300 gas），所以使 fallback 函数的调用尽量廉价很重要。 请注意，调用 fallback 函数的交易（而不是内部调用）所需的 gas 要高得多，因为每次交易都会额外收取 21000 gas 或更多的费用，用于签名检查等操作。

具体来说，以下操作会消耗比 fallback 函数更多的 gas：

- 写入存储
- 创建合约
- 调用消耗大量 gas 的外部函数
- 发送以太币

请确保您在部署合约之前彻底测试您的 fallback 函数，以确保执行成本低于 2300 个 gas。

注解

即使 fallback 函数不能有参数，仍然可以使用 `msg.data` 来获取随调用提供的任何有效数据。

警告

一个没有定义 fallback 函数的合约，直接接收以太币（没有函数调用，即使用 `send` 或 `transfer`）会抛出一个异常， 并返还以太币（在 Solidity v0.4.0 之前行为会有所不同）。所以如果你想让你的合约接收以太币，必须实现 fallback 函数。

警告

一个没有 payable fallback 函数的合约，可以作为 coinbase transaction （又名 miner block reward）的接收者或者作为 `selfdestruct` 的目标来接收以太币。

一个合约不能对这种以太币转移做出反应，因此也不能拒绝它们。这是 EVM 在设计时就决定好的，而且 Solidity 无法绕过这个问题。

这也意味着 `this.balance` 可以高于合约中实现的一些手工记帐的总和（即在 fallback 函数中更新的累加器）。

```
pragma solidity ^0.4.0;

contract Test {
    // 发送到这个合约的所有消息都会调用此函数（因为该合约没有其它函数）。
    // 向这个合约发送以太币会导致异常，因为 fallback 函数没有 `payable` 修饰符
    function() public { x = 1; }
    uint x;
}


// 这个合约会保留所有发送给它的以太币，没有办法返还。
contract Sink {
    function() public payable { }
}

contract Caller {
    function callTest(Test test) public {
        test.call(0xabcdef01); // 不存在的哈希
        // 导致 test.x 变成 == 1。
```

```
        // 以下将不会编译，但如果有人向该合约发送以太币，交易将失败并拒绝以太币。
        // test.send(2 ether）；
    }
}
```

函数重载

合约可以具有多个不同参数的同名函数。这也适用于继承函数。以下示例展示了合约 `A` 中的重载函数 `f`。

```
pragma solidity ^0.4.16;

contract A {
    function f(uint _in) public pure returns (uint out) {
        out = 1;
    }

    function f(uint _in, bytes32 _key) public pure returns (uint out) {
        out = 2;
    }
}
```

重载函数也存在于外部接口中。如果两个外部可见函数仅区别于 Solidity 内的类型而不是它们的外部类型则会导致错误。

```
// 以下代码无法编译
pragma solidity ^0.4.16;

contract A {
    function f(B _in) public pure returns (B out) {
        out = _in;
    }

    function f(address _in) public pure returns (address out) {
        out = _in;
    }
}

contract B {
}
```

以上两个 `f` 函数重载都接受了 ABI 的地址类型，虽然它们在 Solidity 中被认为是不同的。

重载解析和参数匹配

通过将当前范围内的函数声明与函数调用中提供的参数相匹配，可以选择重载函数。 如果所有参数都可以隐式地转换为预期类型，则选择函数作为重载候选项。如果一个候选都没有，解析失败。

注解

返回参数不作为重载解析的依据。

```
pragma solidity ^0.4.16;

contract A {
    function f(uint8 _in) public pure returns (uint8 out) {
        out = _in;
    }

    function f(uint256 _in) public pure returns (uint256 out) {
        out = _in;
    }
}
```

调用 `f(50)` 会导致类型错误，因为 `50` 既可以被隐式转换为 `uint8` 也可以被隐式转换为 `uint256`。 另一方面，调用 `f(256)` 则会解析为 `f(uint256)` 重载，因为 `256` 不能隐式转换为 `uint8`。

事件

事件允许我们方便地使用 EVM 的日志基础设施。 我们可以在 dapp 的用户界面中监听事件，EVM 的日志机制可以反过来“调用”用来监听事件的 Javascript 回调函数。

事件在合约中可被继承。当他们被调用时，会使参数被存储到交易的日志中 —— 一种区块链中的特殊数据结构。 这些日志与地址相关联，被并入区块链中，只要区块可以访问就一直存在（在 Frontier 和 Homestead 版本中会被永久保存，在 Serenity 版本中可能会改动)。 日志和事件在合约内不可直接被访问（甚至是创建日志的合约也不能访问）。

对日志的 SPV（Simplified Payment Verification）证明是可能的，如果一个外部实体提供了一个带有这种证明的合约，它可以检查日志是否真实存在于区块链中。 但需要留意的是，由于合约中仅能访问最近的 256 个区块哈希，所以还需要提供区块头信息。

最多三个参数可以接收 indexed 属性，从而使它们可以被搜索：在用户界面上可以使用 indexed 参数的特定值来进行过滤。

如果数组（包括 string 和 bytes）类型被标记为索引项，则它们的 keccak-256 哈希值会被作为 topic 保存。

除非你用 anonymous 说明符声明事件，否则事件签名的哈希值是 topic 之一。同时也意味着对于匿名事件无法通过名字来过滤。

所有非索引参数都将存储在日志的数据部分中。

注解

索引参数本身不会被保存。你只能搜索它们的值（来确定相应的日志数据是否存在），而不能获取它们的值本身。

```
pragma solidity ^0.4.0;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        // 我们可以过滤对 `Deposit` 的调用，从而用 Javascript API 来查明对这个函数的任何调用（甚至是深度嵌套调用）。
        Deposit(msg.sender, _id, msg.value);
    }
}
```

使用 JavaScript API 调用事件的用法如下：

```
var abi = /* abi 由编译器产生 */;
var ClientReceipt = web3.eth.contract(abi);
```

```
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* 地址 */);

var event = clientReceipt.Deposit();

// 监视变化
event.watch(function(error, result){
    // 结果包括对 `Deposit` 的调用参数在内的各种信息。
    if (!error)
        console.log(result);
});

// 或者通过回调立即开始观察
var event = clientReceipt.Deposit(function(error, result) {
    if (!error)
        console.log(result);
});
```

日志的底层接口

通过函数 log0，log1， log2， log3 和 log4 可以访问日志机制的底层接口。 logi 接受 i + 1 个 bytes32 类型的参数。其中第一个参数会被用来做为日志的数据部分，其它的会做为 topic。上面的事件调用可以以相同的方式执行。

```
pragma solidity ^0.4.10;

contract C {
    function f() public payable {
        bytes32 _id = 0x420042;
        log3(
            bytes32(msg.value),

bytes32(0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20),
            bytes32(msg.sender),
            _id
        );
    }
}
```

其中的长十六进制数的计算方法是 `keccak256("Deposit(address,hash256,uint256)")`，即事件的签名。

其它学习事件机制的资源

- Javascript 文档
- 事件使用例程
- 如何在 js 中访问它们

继承

通过复制包括多态的代码，Solidity 支持多重继承。

所有的函数调用都是虚拟的，这意味着最远的派生函数会被调用，除非明确给出合约名称。

当一个合约从多个合约继承时，在区块链上只有一个合约被创建，所有基类合约的代码被复制到创建的合约中。

总的来说，Solidity 的继承系统与 Python 的继承系统 ，非常 相似，特别是多重继承方面。

下面的例子进行了详细的说明。

```
pragma solidity ^0.4.16;

contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

// 使用 is 从另一个合约派生。派生合约可以访问所有非私有成员，包括内部函数和状态变量，
// 但无法通过 this 来外部访问。
contract mortal is owned {
    function kill() {
        if (msg.sender == owner) selfdestruct(owner);
    }
```

```
}

// 这些抽象合约仅用于给编译器提供接口。
// 注意函数没有函数体。
// 如果一个合约没有实现所有函数，则只能用作接口。
contract Config {
    function lookup(uint id) public returns (address adr);
}

contract NameReg {
    function register(bytes32 name) public;
    function unregister() public;
  }

// 可以多重继承。请注意，owned 也是 mortal 的基类，
// 但只有一个 owned 实例（就像 C++ 中的虚拟继承）。
contract named is owned, mortal {
    function named(bytes32 name) {
        Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
        NameReg(config.lookup(1)).register(name);
    }

    // 函数可以被另一个具有相同名称和相同数量/类型输入的函数重载。
    // 如果重载函数有不同类型的输出参数，会导致错误。
    // 本地和基于消息的函数调用都会考虑这些重载。
    function kill() public {
        if (msg.sender == owner) {
            Config config = Config(0xD5f9D8D94886E70b06E474c3fB14Fd43E2f23970);
            NameReg(config.lookup(1)).unregister();
            // 仍然可以调用特定的重载函数。
            mortal.kill();
        }
    }
}

// 如果构造函数接受参数，
```

```
// 则需要在声明（合约的构造函数）时提供，
// 或在派生合约的构造函数位置以修饰器调用风格提供（见下文）。
contract PriceFeed is owned, mortal, named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    function get() public view returns(uint r) { return info; }

    uint info;
}
```

注意，在上边的代码中，我们调用 `mortal.kill()` 来“转发”销毁请求。 这样做法是有问题的，在下面的例子中可以看到:

```
pragma solidity ^0.4.0;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* 清除操作 1 */ mortal.kill(); }
}

contract Base2 is mortal {
    function kill() public { /* 清除操作 2 */ mortal.kill(); }
}

contract Final is Base1, Base2 {
}
```

调用 `Final.kill()` 时会调用最远的派生重载函数 `Base2.kill`，但是会绕过 `Base1.kill`， 主要是因为它甚至都不知道 `Base1` 的存在。解决这个问题的方法是使用 `super`:

```
pragma solidity ^0.4.0;

contract owned {
    function owned() public { owner = msg.sender; }
    address owner;
}

contract mortal is owned {
    function kill() public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is mortal {
    function kill() public { /* 清除操作 1 */ super.kill(); }
}


contract Base2 is mortal {
    function kill() public { /* 清除操作 2 */ super.kill(); }
}

contract Final is Base1, Base2 {
}
```

如果 `Base2` 调用 `super` 的函数，它不会简单在其基类合约上调用该函数。 相反，它在最终的继承关系图谱的下一个基类合约中调用这个函数，所以它会调用 `Base1.kill()` （注意最终的继承序列是——从最远派生合约开始：Final, Base2, Base1, mortal, ownerd）。 在类中使用 super 调用的实际函数在当前类的上下文中是未知的，尽管它的类型是已知的。 这与普通的虚拟方法查找类似。

基类构造函数的参数

派生合约需要提供基类构造函数需要的所有参数。这可以通过两种方式来完成:

```
pragma solidity ^0.4.0;
```

```
contract Base {
        uint x;
        function Base(uint _x) public { x = _x; }
}

contract Derived is Base(7) {
        function Derived(uint _y) Base(_y * _y) public {
        }
}
```

一种方法直接在继承列表中调用基类构造函数（`is Base(7)`）。 另一种方法是像使用方法一样， 作为派生合约构造函数定义头的一部分， （`Base(_y * _y)`）。 如果构造函数参数是常量并且定义或描述了合约的行为，使用第一种方法比较方便。 如果基类构造函数的参数依赖于派生合约，那么必须使用第二种方法。 如果像这个简单的例子一样，两个地方都用到了，优先使用 风格的参数。

多重继承与线性化

编程语言实现多重继承需要解决几个问题。 一个问题是 钻石问题。 Solidity 借鉴了 Python 的方式并且使用“ C3 线性化 ”强制一个由基类构成的 DAG（有向无环图）保持一个特定的顺序。 这最终反映为我们所希望的唯一化的结果，但也使某些继承方式变为无效。尤其是，基类在 `is` 后面的顺序很重要。 在下面的代码中，Solidity 会给出“ Linearization of inheritance graph impossible ”这样的错误。

```
// 以下代码编译出错

pragma solidity ^0.4.0;

contract X {}
contract A is X {}
contract C is A, X {}
```

代码编译出错的原因是 `C` 要求 `X` 重写 `A` （因为定义的顺序是 `A, X` ）， 但是 `A` 本身要求重写 `X`，无法解决这种冲突。

可以通过一个简单的规则来记忆： 以从“最接近的基类”（most base-like）到“最远的继承”（most derived）的顺序来指定所有的基类。

继承有相同名字的不同类型成员

当继承导致一个合约具有相同名字的函数和 时，这会被认为是一个错误。 当事件和 同名，或者函数和事件同名时，同样会被认为是一个错误。 有一种例外情况，状态变量的 getter 可以覆盖一个 public 函数。

抽象合约

合约函数可以缺少实现，如下例所示（请注意函数声明头由 ; 结尾）：

```
pragma solidity ^0.4.0;

contract Feline {
        function utterance() public returns (bytes32);
}
```

这些合约无法成功编译（即使它们除了未实现的函数还包含其他已经实现了的函数），但他们可以用作基类合约:

```
pragma solidity ^0.4.0;

contract Feline {
        function utterance() public returns (bytes32);
}

contract Cat is Feline {
        function utterance() public returns (bytes32) { return "miaow"; }
}
```

如果合约继承自抽象合约，并且没有通过重写来实现所有未实现的函数，那么它本身就是抽象的。

接口

接口类似于抽象合约，但是它们不能实现任何函数。还有进一步的限制：

1. 无法继承其他合约或接口。
2. 无法定义构造函数。
3. 无法定义变量。

4. 无法定义结构体

5. 无法定义枚举。

将来可能会解除这里的某些限制。

接口基本上仅限于合约 ABI 可以表示的内容，并且 ABI 和接口之间的转换应该不会丢失任何信息。

接口由它们自己的关键字表示：

```
pragma solidity ^0.4.11;

interface Token {
    function transfer(address recipient, uint amount) public;
}
```

就像继承其他合约一样，合约可以继承接口。

库

库与合约类似，它们只需要在特定的地址部署一次，并且它们的代码可以通过 EVM 的 `DELEGATECALL` (Homestead 之前使用 `CALLCODE` 关键字)特性进行重用。 这意味着如果库函数被调用，它的代码在调用合约的上下文中执行，即 `this` 指向调用合约，特别是可以访问调用合约的存储。 因为每个库都是一段独立的代码，所以它仅能访问调用合约明确提供的状态变量（否则它就无法通过名字访问这些变量）。 因为我们假定库是无状态的，所以如果它们不修改状态（也就是说，如果它们是 `view` 或者 `pure` 函数）， 库函数仅可以通过直接调用来使用（即不使用 `DELEGATECALL` 关键字）， 特别是，除非能规避 Solidity 的类型系统，否则是不可能销毁任何库的。

库可以看作是使用他们的合约的隐式的基类合约。虽然它们在继承关系中不会显式可见，但调用库函数与调用显式的基类合约十分类似 （如果 `L` 是库的话，可以使用 `L.f()` 调用库函数）。此外，就像库是基类合约一样，对所有使用库的合约，库的 `internal` 函数都是可见的。 当然，需要使用内部调用约定来调用内部函数，这意味着所有内部类型，内存类型都是通过引用而不是复制来传递。 为了在 EVM 中实现这些，内部库函数的代码和从其中调用的所有函数都在编译阶段被拉取到调用合约中，然后使用一个 `JUMP` 调用来代替 `DELEGATECALL`。

下面的示例说明如何使用库（但也请务必看看 using for 有一个实现 set 更好的例子）。

```
pragma solidity ^0.4.16;

library Set {
  // 我们定义了一个新的结构体数据类型，用于在调用合约中保存数据。
  struct Data { mapping(uint => bool) flags; }

  // 注意第一个参数是“storage reference”类型，因此在调用中参数传递的只是它的存储地址而不是内容。
  // 这是库函数的一个特性。如果该函数可以被视为对象的方法，则习惯称第一个参数为 `self`。
  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // 已经存在
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // 不存在
      self.flags[value] = false;
      return true;
  }

  function contains(Data storage self, uint value)
      public
      view
      returns (bool)
  {
```

```
            return self.flags[value];
        }
    }

    contract C {
        Set.Data knownValues;

        function register(uint value) public {
            // 不需要库的特定实例就可以调用库函数，
            // 因为当前合约就是“instance”。
            require(Set.insert(knownValues, value));
        }
        // 如果我们愿意，我们也可以在这个合约中直接访问 knownValues.flags。
    }
```

当然，你不必按照这种方式去使用库：它们也可以在不定义结构数据类型的情况下使用。 函数也不需要任何存储引用参数，库可以出现在任何位置并且可以有多个存储引用参数。

调用 `Set.contains`， `Set.insert` 和 `Set.remove` 都被编译为外部调用（ `DELEGATECALL` ）。 如果使用库，请注意实际执行的是外部函数调用。 `msg.sender`， `msg.value` 和 `this` 在调用中将保留它们的值， （在 Homestead 之前，因为使用了 `CALLCODE`，改变了 `msg.sender` 和 `msg.value`)。

以下示例展示了如何在库中使用内存类型和内部函数来实现自定义类型，而无需支付外部函数调用的开销：

```
    pragma solidity ^0.4.16;

    library BigInt {
        struct bigint {
            uint[] limbs;
        }

        function fromUint(uint x) internal pure returns (bigint r) {
            r.limbs = new uint[](1);
            r.limbs[0] = x;
        }
```

```
        function add(bigint _a, bigint _b) internal pure returns (bigint r) {
                r.limbs = new uint[](max(_a.limbs.length, _b.limbs.length));
                uint carry = 0;
                for (uint i = 0; i < r.limbs.length; ++i) {
                        uint a = limb(_a, i);
                        uint b = limb(_b, i);
                        r.limbs[i] = a + b + carry;
                        if (a + b < a || (a + b == uint(-1) && carry > 0))
                                carry = 1;
                        else
                                carry = 0;
                }
                if (carry > 0) {
                        // 太差了，我们需要增加一个 limb
                        uint[] memory newLimbs = new uint[](r.limbs.length + 1);
                        for (i = 0; i < r.limbs.length; ++i)
                                newLimbs[i] = r.limbs[i];
                        newLimbs[i] = carry;
                        r.limbs = newLimbs;
                }
        }

        function limb(bigint _a, uint _limb) internal pure returns (uint) {
                return _limb < _a.limbs.length ? _a.limbs[_limb] : 0;
        }

        function max(uint a, uint b) private pure returns (uint) {
                return a > b ? a : b;
        }
}

contract C {
        using BigInt for BigInt.bigint;

        function f() public pure {
```

```
            var x = BigInt.fromUint(7);
            var y = BigInt.fromUint(uint(-1));
            var z = x.add(y);
        }
}
```

由于编译器无法知道库的部署位置，我们需要通过链接器将这些地址填入最终的字节码中 （请参阅 使用命令行编译器 以了解如何使用命令行编译器来链接字节码）。 如果这些地址没有作为参数传递给编译器，编译后的十六进制代码将包含 `__Set_____` 形式的占位符（其中 `Set` 是库的名称）。 可以手动填写地址来将那 40 个字符替换为库合约地址的十六进制编码。

与合约相比，库的限制：

- 没有状态变量
- 不能够继承或被继承
- 不能接收以太币

（将来有可能会解除这些限制）

库的调用保护

如果库的代码是通过 `CALL` 来执行，而不是 `DELEGATECALL` 或者 `CALLCODE` 那么执行的结果会被回退， 除非是对 `view` 或者 `pure` 函数的调用。

EVM 没有为合约提供检测是否使用 `CALL` 的直接方式，但是合约可以使用 `ADDRESS` 操作码找出正在运行的“位置”。 生成的代码通过比较这个地址和构造时的地址来确定调用模式。

更具体地说，库的运行时代码总是从一个 push 指令开始，它在编译时是 20 字节的零。当部署代码运行时，这个常数 被内存中的当前地址替换，修改后的代码存储在合约中。在运行时，这导致部署时地址是第一个被 push 到堆栈上的常数， 对于任何 non-view 和 non-pure 函数，调度器代码都将对比当前地址与这个常数是否一致。

Using For

指令 `using A for B;` 可用于附加库函数（从库 `A`）到任何类型（`B`）。 这些函数将接收到调用它们的对象作为它们的第一个参数（像 Python 的 `self` 变量）。

`using A for *;` 的效果是，库 `A` 中的函数被附加在任意的类型上。

在这两种情况下，所有函数都会被附加一个参数，即使它们的第一个参数类型与对象的类型不匹配。 函数调用和重载解析时才会做类型检查。

`using A for B;` 指令仅在当前作用域有效，目前仅限于在当前合约中，后续可能提升到全局范围。 通过引入一个模块，不需要再添加代码就可以使用包括库函数在内的数据类型。

让我们用这种方式将 库 中的 set 例子重写:

```
pragma solidity ^0.4.16;

// 这是和之前一样的代码，只是没有注释。
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
      public
      returns (bool)
  {
      if (self.flags[value])
          return false; // 已经存在
      self.flags[value] = true;
      return true;
  }

  function remove(Data storage self, uint value)
      public
      returns (bool)
  {
      if (!self.flags[value])
          return false; // 不存在
```

```
        self.flags[value] = false;
        return true;
    }

    function contains(Data storage self, uint value)
        public
        view
        returns (bool)
    {
        return self.flags[value];
    }
}

contract C {
    using Set for Set.Data; // 这里是关键的修改
    Set.Data knownValues;

    function register(uint value) public {
        // Here, all variables of type Set.Data have
        // corresponding member functions.
        // The following function call is identical to
        // `Set.insert(knownValues, value)`
        // 这里， Set.Data 类型的所有变量都有与之相对应的成员函数。
        // 下面的函数调用和 `Set.insert(knownValues, value)` 的效果完全相同。
        require(knownValues.insert(value));
    }
}
```

也可以像这样扩展基本类型:

```
pragma solidity ^0.4.16;

library Search {
    function indexOf(uint[] storage self, uint value)
        public
        view
        returns (uint)
```

```
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return uint(-1);
    }
}

contract C {
    using Search for uint[];
    uint[] data;

    function append(uint value) public {
        data.push(value);
    }

    function replace(uint _old, uint _new) public {
        // 执行库函数调用
        uint index = data.indexOf(_old);
        if (index == uint(-1))
            data.push(_new);
        else
            data[index] = _new;
    }
}
```

注意，所有库调用都是实际的 EVM 函数调用。这意味着如果传递内存或值类型，都将产生一个副本，即使是 `self` 变量。 使用存储引用变量是唯一不会发生拷贝的情况

https://ethereum.github.io/browser-solidity/#optimize=false&version=soljson-v0.4.25+commit.59dbf8f1.js

1. 我们有决定函数何时和被谁调用的可见性修饰符: private 意味着它只能被合约内部调用； internal 就像 private 但是也能被继承的合约调用； external 只能从合约外部调用；最后 public 可以在任何地方调用，不管是内部还是外部。

2. 我们也有状态修饰符， 告诉我们函数如何和区块链交互: view 告诉我们运行这个函数不会更改和保存任何数据； pure 告诉我们这个函数不但不会往区块链写数据，它甚至不从区块链读取数据。这两种在被从合约外部调用的时候都不花费任何 gas（但是它们在被内部其他函数调用的时候将会耗费 gas）。

```
  require(msg.sender == zombieToOwner[_zombieId]);
```

如果你对以太坊的世界有一些了解，你很可能听过人们聊到代币——尤其是 **ERC20** 代币.

一个 _代币_ 在以太坊基本上就是一个遵循一些共同规则的智能合约——即它实现了所有其他代币合约共享的一组标准函数，例如 transfer(address _to, uint256 _value) 和 balanceOf(address _owner).

在智能合约内部，通常有一个映射， mapping(address => uint256) balances，用于追踪每个地址还有多少余额。

所以基本上一个代币只是一个追踪谁拥有多少该代币的合约，和一些可以让那些用户将他们的代币转移到其他地址的函数。

由于所有 ERC20 代币共享具有相同名称的同一组函数，它们都可以以相同的方式进行交互。

这意味着如果你构建的应用程序能够与一个 ERC20 代币进行交互，那么它就也能够与任何 ERC20 代币进行交互。 这样一来，将来你就可以轻松地将更多的代币添加到你的应用中，而无需进行自定义编码。 你可以简单地插入新的代币合约地址，然后哗啦，你的应用程序有另一个它可以使用的代币了。

其中一个例子就是交易所。 当交易所添加一个新的 ERC20 代币时，实际上它只需要添加与之对话的另一个智能合约。 用户可以让那个合约将代币发送到交易所的钱包地址，然后交易所可以让合约在用户要求取款时将代币发送回给他们。

交易所只需要实现这种转移逻辑一次，然后当它想要添加一个新的 ERC20 代币时，只需将新的合约地址添加到它的数据库即可。

ABI 意为应用二进制接口（Application Binary Interface）。基本上，它是以 JSON 格式表示合约的方法，告诉 Web3.js 如何以合同理解的方式格式化函数调用。

View 修饰符

函数定义为 view, 意味着它只能读取数据不能更改数据:

```
function sayHello() public view returns (string) {
```

Solidity 还支持 **pure** 函数, 表明这个函数甚至都不访问应用里的数据, 例如:

```
function _multiply(uint a, uint b) private pure returns (uint) {
  return a * b;
}
```

这个函数甚至都不读取应用里的状态 — 它的返回值完全取决于它的输入参数，在这种情况下我们把函数定义为 **pure**.

如何让 _generateRandomDna 函数返回一个全(半) 随机的 uint?

Ethereum 内部有一个散列函数 keccak256，它用了 SHA3 版本。一个散列函数基本上就是把一个字符串转换为一个 256 位的 16 进制数字。字符串的一个微小变化会引起散列数据极大变化。

这在 Ethereum 中有很多应用，但是现在我们只是用它造一个伪随机数。

例子:

```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0-ba58b8748c00d2e5

keccak256("aaaab");

//
b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9

keccak256("aaaac");
```

显而易见，输入字符串只改变了一个字母，输出就已经天壤之别了。

第 13 章: 事件

我们的合约几乎就要完成了！ 让我们加上一个事件.

事件 是合约和区块链通讯的一种机制。你的前端应用“监听”某些事件，并做出反应。

例子:

```
// 这里建立事件

event IntegersAdded(uint x, uint y, uint result);


function add(uint _x, uint _y) public {
  uint result = _x + _y;
  //触发事件，通知 app
  IntegersAdded(_x, _y, result);
  return result;
}
```

你的 app 前端可以监听这个事件。JavaScript 实现如下:

```
YourContract.IntegersAdded(function(error, result) {
  // 干些事
}
```

```
pragma solidity ^0.4.19;

contract ZombieFactory {
    //这里定义事件
    event NewZombie(uint zombieId, string name, uint dna);
```

```
    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) -1 ;
        //这里触发事件
        NewZombie(id, _name, _dna);
    }

  //生成哈希值
    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }


    function createRandomZombie(string _name) public {
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }

}
```

Mapping（映射）
在第 1 课中，我们看到了 _ 结构体 _ 和 _ 数组 _ 。_映射_ 是另一种在 Solidity 中存储有组织数据的方法。
映射是这样定义的：

```
//对于金融应用程序，将用户的余额保存在一个 uint 类型的变量中：
mapping (address => uint) public accountBalance;
//或者可以用来通过 userId 存储/查找的用户名
mapping (uint => string) userIdToName;
```

映射本质上是存储和查找数据所用的键-值对。在第一个例子中，键是一个 address，值是一个 uint，在第二个例子中，键是一个 uint，值是一个 string。

M

sg.sender

在 Solidity 中，有一些全局变量可以被所有函数调用。 其中一个就是 msg.sender，它指的是当前调用者（或智能合约）的 address。

注意：在 *Solidity* 中，功能执行始终需要从外部调用者开始。 一个合约只会在区块链上什么也不做，除非有人调用其中的函数。所以 *msg.sender* 总是存在的。

equire 使得函数在执行过程中，当不满足某些条件时抛出错误，并停止执行：

```
function sayHiToVitalik(string _name) public returns (string) {
  // 比较 _name 是否等于 "Vitalik". 如果不成立，抛出异常并终止程序
  // (敲黑板: Solidity 并不支持原生的字符串比较, 我们只能通过比较
  // 两字符串的 keccak256 哈希值来进行判断)
  require(keccak256(_name) == keccak256("Vitalik"));
  // 如果返回 true, 运行如下语句
  return "Hi!";
}
```

通过 **is** 关键字 来标明继承：

```
contract ZombieFeeding is ZombieFactory {
}
```

在 Solidity 中，有两个地方可以存储变量 —— storage 或 memory。

在 Solidity 中，当你有多个文件并且想把一个文件导入另一个文件时，可以使用 import 语句：

```
import "./someothercontract.sol";
```

internal 和 external

除 public 和 private 属性之外，Solidity 还使用了另外两个描述函数可见性的修饰词：internal（内部） 和 external（外部）。

internal 和 private 类似，不过， 如果某个合约继承自其父合约，这个合约即可以访问父合约中定义的“内部”函数。（嘿，这听起来正是我们想要的那样！）。

external 与 public 类似，只不过这些函数只能在合约之外调用 - 它们不能被合约内的其他函数调用

Storage 变量是指永久存储在区块链中的变量。 Memory 变量则是临时的，当外部函数对某合约调用完成时，内存型变量即被移除。你可以把它想象成存储在你电脑的硬盘或是 RAM 中数据的关系

汇报

区块链介绍
定义（结合例子去理解）发展
挖矿算法 实践 安全分析 区块链分叉 重放攻击 日蚀攻击
区块链应用前瞻 DNS（去中心化） 抗篡改性（验证溯源，电子病历）

村记账的故事
以太坊挖矿 随机数的不安全性能 掷骰子 类比挖矿
区块链和分布式数据库的区别
ES33%攻击 反对 ES 33%攻击 估计这家伙持有比特币

当我们在谈论区块链，在谈论它的挖矿时，究竟是在谈论什么
之所以坚定相信未来，是因为未来人们的眼睛有拨开历史风尘的睫毛，有看透岁月篇章的瞳孔，真正决定走向的是人们
不久之后，当我们再谈到区块链，我们会觉得它也是个普通的东西罢了
就像以前的因特网，也一样会”飞入平常百姓家”

采用 POW 共识机制来判断谁记账 共识机制：区块链节点竞争来进行记账 竞争评判的标准是共识机制

系统提出一个 难于计算但易于验证的数学难题 大家证明 证明者 得到奖励 计算能力越强越有机会赢得证明

这个过程就叫挖矿　挖矿使用的设备叫矿机　设计出来的专业计算机 就叫矿机

学业考试 短时间 考出好成绩 而非碰运气

0x19	03a30c
前两位幂	后六位为系数
target = coefficient * 2^(8 * (exponent – 3))	
Target=0x0000000000000003A30C000000000000000000000000000000000000	
高度为277,316 的有效区块的头信息哈希值是小于这个目标值的。这个数字的二进制表示位都是0在这个难度上，一个每秒可以处理1万亿个哈希计算的矿工（1 tera-hash per sec TH/sec）平均每8,496个区块才能找到一个正确结果，换句话说，平均每59 天，才能为某块找到正确的哈希值。	