Solidity Documentation

发行版本 *0.8.20*

Ethereum

2023 年 09 月 23 日

基础知识

1 入门指南 **3**

2 翻译 **5**

3 目录 **7**

3.1 智能合约概述 . . . 7
3.2 Solidity 合约示例 . . . 16
3.3 安装 Solidity 编译器 . . . 40
3.4 Solidity 源文件结构 . . . 50
3.5 合约结构 . . . 54
3.6 类型 . . . 57
3.7 单位和全局可用变量 . . . 100
3.8 表达式和控制结构 . . . 108
3.9 合约 . . . 123
3.10 内联汇编 . . . 170
3.11 速查表 . . . 176
3.12 语法 . . . 181
3.13 使用编译器 . . . 207
3.14 分析编译器的输出结果 . . . 222
3.15 基于 Solidity 中间表征的 Codegen 变化 . . . 226
3.16 存储中的状态变量储存结构 . . . 231
3.17 内存中的存储结构 . . . 239
3.18 调用数据的存储结构 . . . 240
3.19 清理变量 . . . 241
3.20 源代码映射 . . . 242
3.21 优化器 . . . 243
3.22 合约的元数据 . . . 266

3.23 合约 ABI 规范 . . . 272
3.24 安全考虑 . . . 289
3.25 已知 bug 列表 . . . 297
3.26 Solidity v0.5.0 突破性变化 . . . 321
3.27 Solidity 0.6.0 版本突破性变化 . . . 330
3.28 Solidity v0.7.0 突破性变化 . . . 333
3.29 Solidity v0.8.0 突破性变化 . . . 336
3.30 风格指南 . . . 339
3.31 SMT 检查器和形式化验证 . . . 344
3.32 Yul . . . 364
3.33 导入路径解析 . . . 389
3.34 风格指南 . . . 400
3.35 通用模式 . . . 426
3.36 资源 . . . 433
3.37 贡献方式 . . . 436
3.38 语言的影响因素 . . . 445
3.39 Solidity 品牌指南 . . . 446

索引 **449**

警告： You are reading a community translation of the Solidity documentation. The Solidity team can give no guarantees on the quality and accuracy of the translations provided. The English reference version is and will remain the only officially supported version by the Solidity team and will always be the most accurate and most up-to-date one. When in doubt, please always refer to the English (original) documentation.

Solidity 是一门为实现智能合约而创建的面向对象的高级编程语言。智能合约是管理以太坊中账户行为的程序。

Solidity 是一种面向以太坊虚拟机 (EVM) 的 带花括号的语言。它受 C++，Python，和 JavaScript 的影响。您可以在语言的影响因素 部分中找到更多有关 Solidity 受哪些语言启发的细节。

Solidity 是静态类型语言，支持继承，库和复杂的用户自定义的类型以及其他特性。

使用 Solidity，您可以创建用于投票、众筹、秘密竞价（盲拍）以及多重签名钱包等用途的合约。

当开发智能合约时，您应该使用最新版本的 Solidity。除某些特殊情况之外，只有最新版本才会收到 安全修复。此外，突破性的变化以及新功能会定期引入。目前，我们使用 0.y.z 版本号 来表明这种快速的变化。

警告： Solidity 最近发布了 0.8.x 版本，该版本引入了许多重大更新。请务必阅读完整列表。

始终欢迎改进 Solidity 或此文档的想法，请阅读我们的贡献者指南 以了解更多细节。

提示： 您可以通过点击左下角的版本号弹出的菜单来选择首选的下载格式来下载该文档的 PDF，HTML 或 Epub 格式。

CHAPTER 1

入门指南

1. 了解智能合约基础知识

如果您是智能合约概念的新手，我们建议您从深入了解“智能合约介绍”部分开始，包括以下内容：

- 用 Solidity 编写的一个简单的智能合约例子。
- 区块链基础知识.
- 以太坊虚拟机.

2. 了解 Solidity

一旦您熟悉了基础知识，我们建议您阅读*"Solidity 示例"* 和“语言描述”部分，以了解该语言的核心概念。

3. 安装 Solidity 编译器

有多种方法可以安装 Solidity 编译器，只需选择您喜欢的选项，并按照安装页面 上提供的步骤操作即可。

提示： 您可以通过 Remix IDE 在浏览器中直接尝试代码示例。Remix 是一个基于网络浏览器的 IDE，允许您编写，部署和管理 Solidity 智能合约，无需在本地安装 Solidity。

警告： 由于人类编写的软件可能会存在错误，因此在编写智能合约时应遵循软件开发的最佳实践。这包括代码审查，测试，审计和正确性证明。智能合约用户有时对代码的信心甚至超过了作者，区块链和智能合约也有其独特的问题需要注意，因此在开始编写生产代码之前，请确保您已阅读安全考虑 部分。

4. 了解更多

如果您想更深入了解如何在以太坊上构建去中心化应用，以太坊开发者资源 可以为您提供有关以太坊的更多文档，以及各种教程、工具和开发框架。

如果您有任何问题，可以在 以太坊 StackExchange 上，或者在我们的 Gitter 频道 上搜索答案或提问。

CHAPTER 2

翻译

社区贡献者帮助将本文档翻译成多种语言。请注意，这些翻译的完整度和及时性各不相同。因此英文版才是参考的标准。

您可以通过点击左下角的语言切换器来切换语言。在弹出的菜单中，选择您需要的语言即可切换。

- 简体中文
- 法语
- 印度尼西亚语
- 日语
- 韩语
- 波斯语
- 俄语
- 西班牙语
- 土耳其语

备注： 我们建立了一个 GitHub 组织和翻译工作流程，以帮助简化社区的工作。请参考 solidity-文档组织 中的翻译指南，了解如何开启新的语言翻译或为社区翻译做出贡献。

CHAPTER 3

目录

关键字索引, 搜索页面

3.1 智能合约概述

3.1.1 简单的智能合约

让我们从一个基本的例子开始，这个例子设置了一个变量的值，并将其暴露给其他合约来访问。如果您现在不理解这些东西也没关系，我们稍后会讨论更多细节。

存储合约示例

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
```

(续下页)

(接上页)

```
        return storedData;
    }
}
```

第一行告诉您，源代码是根据 GPL3.0 版本授权的。在发布源代码是默认的情况下，机器可读的许可证说明是很重要的。

下一行指定源代码是为 Solidity 0.4.16 版本编写的，或该语言的较新版本，直到但不包括 0.9.0 版本。这是为了确保合约不能被新的（有重大改变的）编译器版本编译，在那里它可能会有不同的表现。*Pragmas* 是编译器关于如何处理源代码的常用指令（例如，pragma once ）。

Solidity 意义上的合约是代码（其 *函数*）和数据（其 *状态*）的集合，驻留在以太坊区块链的一个特定地址。这一行 `uint storedData;` 声明了一个名为 `storedData` 的状态变量，类型为 `uint` （*u*nsigned *int*eger，共 *256* 位）。您可以把它看作是数据库中的一个槽，您可以通过调用管理数据库的代码函数来查询和改变它。在这个例子中，合约定义了可以用来修改或检索变量值的函数 `set` 和 `get`。

要访问当前合约的一个成员（如状态变量），通常不需要添加 `this.` 前缀，只需要通过它的名字直接访问它。与其他一些语言不同的是，省略它不仅仅是一个风格问题，它导致了一种完全不同的访问成员的方式，但后面会有更多关于这个问题。

该合约能完成的事情并不多（由于以太坊构建的基础架构的原因），它能允许任何人在合约中存储一个单独的数字，并且这个数字可以被世界上任何人访问，且没有可行的办法阻止您发布这个数字。当然，任何人都可以再次调用 `set` ，传入不同的值，覆盖您的数字，但是这个数字仍会被存储在区块链的历史记录中。随后，我们会看到怎样施加访问限制，以确保只有您才能改变这个数字。

警告： 小心使用 Unicode 文本，因为有些字符虽然长得相似（甚至一样），但其字符码是不同的，其编码后的字符数组也会不一样。

备注： 所有的标识符（合约名称，函数名称和变量名称）都只能使用 ASCII 字符集。UTF-8 编码的数据可以用字符串变量的形式存储。

子货币（Subcurrency）例子

下面的合约实现了一个最简单的加密货币。这里，币确实可以无中生有地产生，但是只有创建合约的人才能做到（实现一个不同的发行计划也不难）。而且，任何人都可以给其他人转币，不需要注册用户名和密码，所需要的只是以太坊密钥对。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

```

(续下页)

(接上页)

```
contract Coin {
    // 关键字 "public" 使变量可以从其他合约中访问。
    address public minter;
    mapping(address => uint) public balances;

    // 事件允许客户端对您声明的特定合约变化做出反应
    event Sent(address from, address to, uint amount);

    // 构造函数代码只有在合约创建时运行
    constructor() {
        minter = msg.sender;
    }

    // 向一个地址发送一定数量的新创建的代币
    // 但只能由合约创建者调用
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }

    // 错误类型变量允许您提供关于操作失败原因的信息。
    // 它们会返回给函数的调用者。
    error InsufficientBalance(uint requested, uint available);

    // 从任何调用者那里发送一定数量的代币到一个地址
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });

        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

这个合约引入了一些新的概念，让我们逐一解读。

`address public minter;` 这一行声明了一个可以被公开访问的 *address* 类型的状态变量。`address` 类型是一个 160 位的值，且不允许任何算数操作。这种类型适合存储合约地址或外部账户 的密钥对。

关键字 `public` 自动生成一个函数，允许您在这个合约之外访问这个状态变量的当前值。如果没有这个关键字，其他的合约没有办法访问这个变量。由编译器生成的函数的代码大致如下所示（暂时忽略 `external` 和

view):

```
function minter() external view returns (address) { return minter; }
```

您可以自己添加一个类似上述的函数，但您会有同名的一个函数和一个变量。您不需要这样做，编译器会帮您解决这个问题。

下一行，`mapping(address => uint) public balances;` 也创建了一个公共状态变量，但它是一个更复杂的数据类型。映射 类型将地址映射到无符号整数。

映射可以被看作是 哈希表，它实际上是被初始化的，因此每一个可能的键从一开始就存在，并被映射到一个值，其字节表示为全零的值。然而，它既不可能获得一个映射的所有键的列表，也不可能获得所有值的列表。因此，要么记住您添加到映射中的内容，要么在不需要的情况下使用它。甚至更好的是，保留一个列表，或者使用一个更合适的数据类型。

而由 `public` 关键字创建的*getter* 函数 则是更复杂一些的情况，它大致如下所示：

```
function balances(address account) external view returns (uint) {
    return balances[account];
}
```

您可以用这个函数来查询单个账户的余额。

这一行 `event Sent(address from, address to, uint amount);` 声明了一个"事件"，它是在函数 `send` 的最后一行发出的。以太坊客户端，如网络应用，可以监听区块链上发出的这些事件，而不需要太多的成本。一旦发出，监听器就会收到参数 `from`，`to` 和 `amount`，这使得跟踪交易成为可能。

为了监听这个事件，您可以使用以下方法 JavaScript 代码，使用 web3.js 来创建 `Coin` 合约对象，然后在任何用户界面调用上面自动生成的 `balances` 函数：

```
Coin.Sent().watch({}, '', function(error, result) {
    if (!error) {
        console.log("Coin transfer: " + result.args.amount +
            " coins were sent from " + result.args.from +
            " to " + result.args.to + ".");
        console.log("Balances now:\n" +
            "Sender: " + Coin.balances.call(result.args.from) +
            "Receiver: " + Coin.balances.call(result.args.to));
    }
})
```

constructor 是一个特殊的函数，只在创建合约的过程中执行，事后不能再被调用。在这种情况下，它永久地存储了创建合约的人的地址。`msg` 变量（与 `tx` 和 `block` 一起）是一个特殊全局变量，其中包含允许访问区块链的属性。`msg.sender` 总是当前（外部）函数调用的地址。

最后，真正被用户或其他合约所调用的，以完成本合约功能的方法是 `mint` 和 `send`。

`mint` 函数发送一定数量的新创建的代币到另一个地址。*require* 函数调用定义了一些条件，如果不满足这些

条件就会恢复所有的变化。在这个例子中，`require(msg.sender == minter);` 确保只有合约的创建者可以调用 `mint`。一般来说，创建者可以随心所欲地铸造代币，但在某些时候，这将导致一种叫做” 溢出” 的现象。请注意，由于默认的检查过的算术，如果表达式 `balances[receiver] += amount;` 溢出，即当任意精度算术中的 `balances[receiver] + amount` 大于 `uint` 的最大值（`2**256 - 1`）时，交易将被恢复。对于函数 `send` 中的语句 `balances[receiver] += amount;` 也是如此。

错误（*Errors*） 允许您向调用者提供更多关于一个条件或操作失败原因的信息。错误与恢复状态 一起使用。`revert` 语句无条件地中止和恢复所有的变化，类似于 `require` 函数，但它也允许您提供错误的名称和额外的数据，这些数据将提供给调用者（并最终提供给前端应用程序或区块资源管理器），以便更容易调试失败或做出反应。

任何人（已经拥有一些这样的代币）都可以使用 `send` 函数来发送代币给其他任何人。如果发送者没有足够的代币可以发送，那么 `if` 条件就会为真。因此，`revert` 将导致操作失败，同时使用 `InsufficientBalance` 错误向发送者提供错误细节。

备注： 如果您用这个合约向一个地址发送代币，当您在区块链浏览器上查看该地址时，您不会看到任何东西，因为您发送代币的记录和变化的余额只存储在这个特定的代币合约的数据存储中。通过使用事件，您可以创建一个” 区块链浏览器”，跟踪您的新币的交易和余额，但您必须检查币合约地址，而不是币主的地址。

3.1.2 区块链基础

对于程序员来说，区块链这个概念并不难理解，这是因为大多数难懂的东西（挖矿，哈希 ，椭圆曲线密码学，点对点网络（P2P） 等）都只是用于提供特定的功能和承诺。您只需接受这些既有的特性功能，不必关心底层技术，比如，难道您必须知道亚马逊的 AWS 内部原理，您才能使用它吗？

交易/事务

区块链是全球共享的事务性数据库，这意味着每个人都可加入网络来阅读数据库中的记录。如果您想改变数据库中的某些东西，您必须创建一个被所有其他人所接受的事务。事务一词意味着您想做的（假设您想要同时更改两个值），要么一点没做，要么全部完成。此外，当您的事务被应用到数据库时，其他事务不能修改数据库。

举个例子，设想一张表，列出电子货币中所有账户的余额。如果请求从一个账户转移到另一个账户，数据库的事务特性确保了如果从一个账户扣除金额，它总被添加到另一个账户。如果由于某些原因，无法添加金额到目标账户时，源账户也不会发生任何变化。

此外，交易总是由发送人（创建者）签名。这样，就可非常简单地为数据库的特定修改增加访问保护机制。在电子货币的例子中，一个简单的检查可以确保只有持有账户密钥的人才能从中转账。

区块

要克服的一个主要障碍是（用比特币的术语）所谓的 “双花攻击 (double-spend attack)”：如果网络中存在两个交易，都想清空一个账户，会发生什么？只有其中一个交易是有效的，通常是最先被接受的那个。问题是，在点对点的网络中，” 第一” 不是一个客观的术语。

对此，抽象的答案是，您不必在意。一个全球公认的交易顺序将为您选择，解决这样的冲突。这些交易将被捆绑成所谓的” 区块”，然后它们将在所有参与节点中执行和分发。如果两个交易相互矛盾，最终排在第二位的那个交易将被拒绝，不会成为区块的一部分。

这些区块按时间形成了一个线性序列，这就是 “区块链” 一词的由来。区块每隔一段时间就会被添加到链上，但这些时间间隔在未来可能会发生变化。如需了解最新信息，建议在 Etherscan 等网站上对网络进行监控。

作为 “顺序选择机制”（也就是所谓的 “挖矿”）的一部分，可能有时会发生块（blocks）被回滚的情况，但仅在链的 “末端”。末端增加的块越多，其发生回滚的概率越小。因此您的交易被回滚甚至从区块链中抹除，这是可能的，但等待的时间越长，这种情况发生的概率就越小。

备注： 交易不保证被包括在下一个区块或任何特定的未来区块中，因为这不是由交易的提交者决定的，而是由矿工来决定交易被包括在哪个区块中。

如果您想安排您的合约的未来调用，您可以使用智能合约自动化工具或 oracle 服务。

3.1.3 以太坊虚拟机

概述

以太坊虚拟机或 EVM 是以太坊智能合约的运行环境。它不仅是沙盒封装的，而且实际上是完全隔离的，这意味着在 EVM 内运行的代码不能访问网络，文件系统或其他进程。甚至智能合约之间的访问也是受限的。

账户

在以太坊有两种共享同一地址空间的账户：**外部账户**，由公钥-私钥对（也就是人）控制；**合约账户**，由与账户一起存储的代码控制。

外部账户的地址是由公钥确定的，而合约的地址是在合约创建时确定的（它是由创建者地址和从该地址发出的交易数量得出的，即所谓的”nonce”）。

无论账户是否存储代码，这两种类型都被 EVM 平等对待。

每个账户都有一个持久的键值存储，将 256 位的字映射到 256 位的字，称为 **存储**。

此外，每个账户有一个以太 **余额**（balance）（单位是 “Wei”，`1 ether` 是 `10**18 wei`），余额会因为发送包含以太币的交易而改变。

交易

交易可以看作是从一个帐户发送到另一个帐户的消息（这里的账户，可能是相同的或特殊的零帐户，请参阅下文）。它能包含一个二进制数据（被称为“合约负载”）和以太。

如果目标账户含有代码，此代码会被执行，并以合约负载（二进制数据）作为入参。

如果目标账户没有设置（交易没有接收者或接收者被设置为 `null`），交易会创建一个 **新合约**。正如已经提到的，该合约的地址不是零地址，而是从发送者和其发送的交易数量（“nonce”）中得出的地址。这种合约创建交易的有效负载被认为是 EVM 字节码并被执行。该执行的输出数据被永久地存储为合约的代码。这意味着，为创建一个合约，您不需要发送实际的合约代码，而是发送能够产生合约代码的代码。

备注： 在合约创建的过程中，它的代码还是空的。所以直到构造函数执行结束，您都不应该在其中调用合约自己函数。

Gas

一经创建，每笔交易都会被收取一定数量的 **gas**，这些 gas 必须由交易的发起人（`tx.origin`）支付。在 EVM 执行交易时，gas 根据特定规则逐渐耗尽。如果 gas 在某一点被用完（即它会为负），将触发一个 gas 耗尽异常，这将结束执行并撤销当前调用栈中对状态所做的所有修改。

此机制激励了对 EVM 执行时间的经济利用，并为 EVM 执行器（即矿工/持币者）的工作提供补偿。由于每个区块都有最大 gas 量，因此还限制了验证块所需的工作量。

gas price 是交易发起人设定的值，他必须提前向 EVM 执行器支付 `gas_price * gas`。如果执行后还剩下一些 gas，则退还给交易发起人。如果发生撤销更改的异常，已经使用的 gas 不会退还。

由于 EVM 执行器可以选择包含一笔交易，因此交易发送者无法通过设置低 gas 价格滥用系统。

存储，内存和栈

以太坊虚拟机有三个存储数据的区域：存储器，内存和堆栈。

每个账户都有一个称为 **存储**的数据区，在函数调用和交易之间是持久的。存储是一个键值存储，将 256 位的字映射到 256 位的字。在合约中枚举存储是不可能的，读取的成本相对较高，初始化和修改存储的成本更高。由于这种成本，您应该把您存储在持久性存储中的内容减少到合约运行所需的程度。在合约之外存储像派生计算，缓存和聚合的数据。合约既不能读也不能写到除其自身以外的任何存储。

第二个数据区被称为 **内存**，合约在每次消息调用时都会获得一个新清除的实例。内存是线性的，可以在字节级寻址，但读的宽度限制在 256 位，而写的宽度可以是 8 位或 256 位。当访问（无论是读还是写）一个先前未触及的内存字（即一个字内的任何偏移）时，内存被扩展一个字（256 位）。在扩展的时候，必须支付 gas 成本。内存越大，成本就越高（它以平方级别扩展）。

EVM 不是基于寄存器的，而是基于栈的，因此所有的计算都在一个被称为 **栈（stack）** 的区域执行。栈最大有 1024 个元素，每个元素长度是一个字（256 位）。对栈的访问只限于其顶端，限制方式为：允许拷贝最顶

端的 16 个元素中的一个到栈顶，或者是交换栈顶元素和下面 16 个元素中的一个。所有其他操作都只能取最顶的两个（或一个，或更多，取决于具体的操作）元素，运算后，把结果压入栈顶。当然可以把栈上的元素放到存储或内存中。但是无法只访问栈上指定深度的那个元素，除非先从栈顶移除其他元素。

指令集

EVM 的指令集应尽量保持最小，以避免不正确或不一致的实现，这可能导致共识问题。所有的指令都是在基本的数据类型上操作的，256 位的字或内存的片断（或其他字节数组）。具备常用的算术，位，逻辑和比较操作。也可以做到有条件和无条件跳转。此外，合约可以访问当前区块的相关属性，比如它的编号和时间戳。

关于完整的列表，请参见操作码列表，它是内联汇编文档的一部分。

消息调用

合约可以通过消息调用的方式来调用其它合约或者发送以太币到非合约账户。消息调用和交易非常类似，它们都有一个源，目标，数据，以太币，gas 和返回数据。事实上每个交易都由一个顶层消息调用组成，这个消息调用又可创建更多的消息调用。

合约可以决定它剩余的 **gas** 有多少应该随内部消息调用一起发送，有多少它想保留。如果在内部调用中发生了 out-of-gas 的异常（或任何其他异常），这将由一个被压入栈顶的错误值来表示。在这种情况下，只有与调用一起发送的 gas 被用完。在 Solidity 中，在这种情况下，发起调用的合约默认会引起一个手动异常，所以异常会在调用栈上" 冒泡出来"。

如前文所述，被调用的合约（可以与调用者是同一个合约）将收到一个新清空的内存实例，并可以访问调用的有效负载-由被称为 **calldata** 的独立区域所提供的数据。在它执行完毕后，它可以返回数据，这些数据将被存储在调用者内存中由调用者预先分配的位置。所有这样的调用都是完全同步的。

调用被 **限制**在 1024 的深度，这意味着对于更复杂的操作，循环应优先于递归调用。此外，在一个消息调用中，只有 63/64 的 gas 可以被转发，这导致在实践中，深度限制略低于 1000。

委托调用和库

存在一种特殊的消息调用，被称为 **委托调用（delegatecall）**，除了目标地址的代码是在调用合约的上下文（即地址）中执行，`msg.sender` 和 `msg.value` 的值不会更改之外，其他与消息调用相同。

这意味着合约可以在运行时动态地从不同的地址加载代码。存储，当前地址和余额仍然指的是调用合约，只是代码取自被调用的地址。

这使得在 Solidity 中实现"库"的功能成为可能：可重复使用的库代码，可以放在一个合约的存储上，例如，用来实现复杂的数据结构的库。

日志

有一种特殊的可索引的数据结构，其存储的数据可以一路映射直到区块层级。这个特性被称为 **日志（logs）**，Solidity 用它来实现事件。合约创建之后就无法访问日志数据，但是这些数据可以从区块链外高效的访问。因为部分日志数据被存储在 布隆过滤器（bloom filter） 中，我们可以高效并且加密安全地搜索日志，所以那些没有下载整个区块链的网络节点（轻客户端）也可以找到这些日志。

创建

合约甚至可以通过一个特殊的指令来创建其他合约（不是简单的调用零地址）。创建合约的调用 **create calls** 和普通消息调用的唯一区别在于，负载会被执行，执行的结果被存储为合约代码，调用者/创建者在栈上得到新合约的地址。

停用和自毁

从区块链上删除代码的唯一方法是当该地址的合约执行 `selfdestruct` 操作。存储在该地址的剩余以太币被发送到一个指定的目标，然后存储和代码被从状态中删除。删除合约在理论上听起来是个好主意，但它有潜在的危险性，因为如果有人向被删除的合约发送以太币，以太币就会永远丢失。

> **警告：** 从 0.8.18 及更高版本开始，在 Solidity 和 Yul 中使用 `selfdestruct` 将触发弃用警告，因为 `SELFDESTRUCT` 操作码最终将经历 EIP-6049 中所述的行为的重大变化。

> **警告：** 即使一个合约通过 `selfdestruct` 删除，它仍然是区块链历史的一部分，可能被大多数以太坊节点保留。因此，使用 `selfdestruct` 与从硬盘上删除数据不一样。

备注： 尽管一个合约的代码中没有显式地调用 `selfdestruct` ，它仍然有可能通过 `delegatecall` 或 `callcode` 执行自毁操作。

如果您想停用您的合约，您可以通过改变一些内部状态来 **停用**它们，从而使再次调用所有的功能都会被恢复。这样就无法使用合约了，因为它立即返回以太。

预编译合约

有一小群合约地址是特殊的。`1` 和（包括）`8` 之间的地址范围包含"预编译合约"，可以像其他合约一样被调用，但它们的行为（和它们的 gas 消耗）不是由存储在该地址的 EVM 代码定义的（它们不包含代码），而是由 EVM 执行环境本身实现。

不同的 EVM 兼容链可能使用不同的预编译合约集。未来也有可能在以太坊主链上添加新的预编译合约，但您可以合理地预期它们总是在 `1` 和 `0xffff`（包括）之间。

3.2 Solidity 合约示例

3.2.1 投票合约

下面的合约相当复杂，但展示了 Solidity 的很多特性。它实现了一个投票合约。当然，电子投票的主要问题是如何将投票权分配给正确的人以及如何防止人为操纵。我们不会在这里解决所有的问题，但至少我们会展示如何进行委托投票，与此同时，使计票是 **自动且完全透明的**。

我们的想法是为每张选票创建一份合约，为每个选项提供一个简称。然后，作为合约的创造者——即主席，将给予每个地址单独的投票权。

地址后面的人可以选择自己投票，或者委托给他们信任的人来投票。

在投票时间结束时，`winningProposal()` 将返回拥有最大票数的提案。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title 委托投票
contract Ballot {
    // 这声明了一个新的复杂类型，用于稍后变量。
    // 它用来表示一个选民。
    struct Voter {
        uint weight; // 计票的权重
        bool voted;  // 若为真，代表该人已投票
        address delegate; // 被委托人
        uint vote;   // 投票提案的索引
    }

    // 提案的类型
    struct Proposal {
        bytes32 name;   // 简称（最长32个字节）
        uint voteCount; // 得票数
    }

    address public chairperson;
```

(续下页)

(接上页)

```
    // 这声明了一个状态变量，为每个可能的地址存储一个 `Voter`。
    mapping(address => Voter) public voters;

    // 一个 `Proposal` 结构类型的动态数组。
    Proposal[] public proposals;

    /// 为 `proposalNames` 中的每个提案，创建一个新的（投票）表决
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // 对于提供的每个提案名称，
        // 创建一个新的 Proposal 对象并把它添加到数组的末尾。
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` 创建一个临时 Proposal 对象
            // `proposals.push(...)` 将其添加到 `proposals` 的末尾
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // 给予 `voter` 在这张选票上投票的权利。
    // 只有 `chairperson` 可以调用该函数。
    function giveRightToVote(address voter) external {
        // 若 `require` 的第一个参数的计算结果为 `false`，
        // 则终止执行，撤销所有对状态和以太币余额的改动。
        // 在旧版的 EVM 中这曾经会消耗所有 gas，但现在不会了。
        // 使用 `require` 来检查函数是否被正确地调用，通常是个好主意。
        // 您也可以在 `require` 的第二个参数中提供一个对错误情况的解释。
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }
```

(续下页)

(接上页)

```
    /// 把您的投票委托给投票者 `to`。
    function delegate(address to) external {
        // 指定引用
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "You have no right to vote");
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is disallowed.");

        // 委托是可以传递的，只要被委托者 `to` 也设置了委托。
        // 一般来说，这样的循环委托是非常危险的，因为如果传递的链条太长，
        // 可能需要消耗的gas就会超过一个区块中的可用数量。
        // 这种情况下，委托不会被执行。
        // 但在其他情况下，如果形成闭环，则会导致合约完全被 "卡住"。
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // 不允许闭环委托
            require(to != msg.sender, "Found loop in delegation.");
        }

        Voter storage delegate_ = voters[to];

        // 投票者不能将投票权委托给不能投票的账户。
        require(delegate_.weight >= 1);

        // 由于 `sender` 是一个引用，
        // 因此这会修改 `voters[msg.sender]`。
        sender.voted = true;
        sender.delegate = to;

        if (delegate_.voted) {
            // 若被委托者已经投过票了，直接增加得票数。
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // 若被委托者还没投票，增加委托者的权重。
            delegate_.weight += sender.weight;
        }
    }

    /// 把您的票(包括委托给您的票)，
    /// 投给提案 `proposals[proposal].name`。
    function vote(uint proposal) external {
```

(续下页)

(接上页)

```
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;

        // 如果 `proposal` 超过了数组的范围,
        // 则会自动抛出异常, 并恢复所有的改动。
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev 结合之前所有投票的情况下, 计算出获胜的提案。
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // 调用 `winningProposal()` 函数以获取提案数组中获胜者的索引,
    // 并以此返回获胜者的名称。
    function winnerName() external view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

可能的优化

当前，为了把投票权分配给所有参与者，需要执行很多交易。此外，如果两个或更多的提案有相同的票数，`winningProposal()` 无法登记平局。您能想出一个办法来解决这些问题吗？

3.2.2 盲拍（秘密竞价）

在本节中，我们将展示如何轻松地在以太坊上创建一个盲拍的合约。我们将从一个公开拍卖开始，每个人都可以看到出价，然后将此合约扩展到盲拍合约，在竞标期结束之前无法看到实际出价。

简单的公开拍卖

下面这个简单的拍卖合约的总体思路是，每个人都可以在竞标期间发送他们的竞标。竞标已经包括发送资金/以太币，以便将竞标者与他们的竞标绑定。如果最高出价被提高，之前的最高出价者就会拿回他们的钱。竞价期结束后，受益人需要手动调用合约，才能收到他们的钱 - 合约不能自己激活接收。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // 拍卖的参数。
    // 时间是 unix 的绝对时间戳（自1970-01-01以来的秒数）
    // 或以秒为单位的时间段。
    address payable public beneficiary;
    uint public auctionEndTime;

    // 拍卖的当前状态。
    address public highestBidder;
    uint public highestBid;

    // 允许取回以前的竞标。
    mapping(address => uint) pendingReturns;

    // 拍卖结束后设为 `true`，将禁止所有的变更
    // 默认初始化为 `false`。
    bool ended;

    // 变化时将会发出的事件。
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // 描述失败的错误信息。

    // 三斜线的注释是所谓的 natspec 注释。
    // 当用户被要求确认一个交易或显示一个错误时，它们将被显示。

    /// 竞拍已经结束。
    error AuctionAlreadyEnded();
    /// 已经有一个更高的或相等的出价。
    error BidNotHighEnough(uint highestBid);
```

(续下页)

(接上页)

```
/// 竞拍还没有结束。
error AuctionNotYetEnded();
/// 函数 auctionEnd 已经被调用。
error AuctionEndAlreadyCalled();

/// 以受益者地址 `beneficiaryAddress` 创建一个简单的拍卖，
/// 拍卖时长为 `_biddingTime`。
constructor(
    uint biddingTime,
    address payable beneficiaryAddress
) {
    beneficiary = beneficiaryAddress;
    auctionEndTime = block.timestamp + biddingTime;
}

/// 对拍卖进行出价，具体的出价随交易一起发送。
/// 如果没有在拍卖中胜出，则返还出价。
function bid() external payable {
    // 参数不是必要的。因为所有的信息已经包含在了交易中。
    // 关键字 `payable` 是函数能够接收以太币的必要条件。

    // 如果拍卖已结束，撤销函数的调用。
    if (block.timestamp > auctionEndTime)
        revert AuctionAlreadyEnded();

    // 如果出价不高，就把钱送回去
    // (revert语句将恢复这个函数执行中的所有变化，
    // 包括它已经收到钱）。
    if (msg.value <= highestBid)
        revert BidNotHighEnough(highestBid);

    if (highestBid != 0) {
        // 简单地使用 highestBidder.send(highestBid)
        // 返还出价时，是有安全风险的，
        // 因为它可能执行一个不受信任的合约。
        // 让接收方自己取钱总是比较安全的。
        pendingReturns[highestBidder] += highestBid;
    }
    highestBidder = msg.sender;
    highestBid = msg.value;
    emit HighestBidIncreased(msg.sender, msg.value);
}
```

(续下页)

(接上页)

```
    /// 撤回出价过高的竞标。
    function withdraw() external returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // 将其设置为0是很重要的，
            // 因为接收者可以在 `send` 返回之前再次调用这个函数
            // 作为接收调用的一部分。
            pendingReturns[msg.sender] = 0;

            // msg.sender 不属于 `address payable` 类型，
            // 必须使用 `payable(msg.sender)` 明确转换，
            // 以便使用成员函数 `send()`。
            if (!payable(msg.sender).send(amount)) {
                // 这里不需抛出异常，只需重置未付款
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// 结束拍卖，并把最高的出价发送给受益人。
    function auctionEnd() external {
        // 对于可与其他合约交互的函数（意味着它会调用其他函数或发送以太币），
        // 一个好的指导方针是将其结构分为三个阶段：
        // 1. 检查条件
        // 2. 执行动作 (可能会改变条件)
        // 3. 与其他合约交互
        // 如果这些阶段相混合，其他的合约可能会回调当前合约并修改状态，
        // 或者导致某些效果（比如支付以太币）多次生效。
        // 如果合约内调用的函数包含了与外部合约的交互，
        // 则它也会被认为是与外部合约有交互的。

        // 1. 条件
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
        if (ended)
            revert AuctionEndAlreadyCalled();

        // 2. 影响
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);
```

(续下页)

(接上页)

```
        // 3. 交互
        beneficiary.transfer(highestBid);
    }
}
```

盲拍（秘密竞拍）

之前的公开拍卖接下来将被扩展为盲目拍卖。盲拍的好处是，在竞价期即将结束时没有时间压力。在一个透明的计算平台上创建一个盲拍可能听起来是一个矛盾，但加密技术可以实现它。

在 **竞标期间**，竞标者实际上并没有发送他们的出价，而只是发送一个哈希版本的出价。由于目前几乎不可能找到两个（足够长的）值，其哈希值是相等的，因此竞标者可通过该方式提交报价。在竞标结束后，竞标者必须公开他们的出价：他们发送未加密的值，合约检查出价的哈希值是否与竞标期间提供的值相同。

另一个挑战是如何使拍卖同时做到 **绑定和秘密**：唯一能阻止竞标者在赢得拍卖后不付款的方式是，让他们将钱和竞标一起发出。但由于资金转移在以太坊中不能被隐藏，因此任何人都可以看到转移的资金。

下面的合约通过接受任何大于最高出价的值来解决这个问题。当然，因为这只能在揭示阶段进行检查，有些出价可能是 **无效**的，而这是有目的的（它甚至提供了一个明确的标志，以便在高价值的转移中进行无效的出价）：竞标者可以通过设置几个或高或低的无效出价来迷惑竞争对手。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // 允许取回以前的竞标。
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);
```

(续下页)

(接上页)

```
// 描述失败的错误信息。

/// 该函数被过早调用。
/// 在 `time` 时间再试一次。
error TooEarly(uint time);
/// 该函数被过晚调用。
/// 它不能在 `time` 时间之后被调用。
error TooLate(uint time);
/// 函数 auctionEnd 已经被调用。
error AuctionEndAlreadyCalled();

// 使用 修饰符 (modifier) 可以更便捷的校验函数的入参。
// `onlyBefore` 会被用于后面的 `bid` 函数:
// 新的函数体是由 modifier 本身的函数体，其中`_`被旧的函数体所取代。
modifier onlyBefore(uint time) {
    if (block.timestamp >= time) revert TooLate(time);
    _;
}
modifier onlyAfter(uint time) {
    if (block.timestamp <= time) revert TooEarly(time);
    _;
}

constructor(
    uint biddingTime,
    uint revealTime,
    address payable beneficiaryAddress
) {
    beneficiary = beneficiaryAddress;
    biddingEnd = block.timestamp + biddingTime;
    revealEnd = biddingEnd + revealTime;
}

/// 可以通过 `_blindedBid` = keccak256(value, fake, secret)
/// 设置一个盲拍。
/// 只有在出价披露阶段被正确披露，已发送的以太币才会被退还。
/// 如果与出价一起发送的以太币至少为 "value" 且 "fake" 不为真，则出价有效。
/// 将 "fake" 设置为 true ，
/// 然后发送满足订金金额但又不与出价相同的金额是隐藏实际出价的方法。
/// 同一个地址可以放置多个出价。
function bid(bytes32 blindedBid)
    external
    payable
```

(续下页)

(接上页)

```
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: blindedBid,
            deposit: msg.value
        }));
    }

    /// 披露你的盲拍出价。
    /// 对于所有正确披露的无效出价以及除最高出价以外的所有出价，您都将获得退款。
    function reveal(
        uint[] calldata values,
        bool[] calldata fakes,
        bytes32[] calldata secrets
    )
        external
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(values.length == length);
        require(fakes.length == length);
        require(secrets.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (values[i], fakes[i], secrets[i]);
            if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake,␣
↪secret))) {
                // 出价未能正确披露。
                // 不返还订金。
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // 使发送者不可能再次认领同一笔订金。
            bidToCheck.blindedBid = bytes32(0);
        }
```

(续下页)

(接上页)

```
        payable(msg.sender).transfer(refund);
    }

    /// 撤回出价过高的竞标。
    function withdraw() external {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // 这里很重要，首先要设零值。
            // 因为，作为接收调用的一部分，
            // 接收者可以在 `transfer` 返回之前重新调用该函数。
            // (可查看上面关于 条件 -> 影响 -> 交互 的标注)
            pendingReturns[msg.sender] = 0;

            payable(msg.sender).transfer(amount);
        }
    }

    /// 结束拍卖，并把最高的出价发送给受益人。
    function auctionEnd()
        external
        onlyAfter(revealEnd)
    {
        if (ended) revert AuctionEndAlreadyCalled();
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }

    // 这是一个 "internal" 函数，
    // 意味着它只能在本合约（或继承合约）内被调用。
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // 返还之前的最高出价
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
```

(续下页)

(接上页)

```
    }
}
```

3.2.3 安全的远程购买

目前，远程购买商品需要多方相互信任。最简单的关系涉及一个卖家和一个买家。买方希望从卖方那里收到一件物品，卖方希望得到金钱（或等价物）作为回报。这里面有问题的部分是的运输。没有办法确定物品是否到达买方手中。

有多种方法来解决这个问题，但都有这样或那样的不足之处。在下面的例子中，双方都要把两倍价值于物品的资金放入合约中作为托管。只要发生这种情况，钱就会一直锁在合同里面，直到买方确认收到物品。之后，买方会得到退回的资金（他们押金的一半），卖方得到三倍的资金（他们的押金加上物品的价值）。这背后的想法是，双方都有动力去解决这个问题，否则他们的钱就会被永远锁定。

这个合约当然不能解决问题，但它概述了如何在合约内使用类似状态机的构造。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;

    enum State { Created, Locked, Release, Inactive }
    // 状态变量的默认值是第一个成员，`State.created`。
    State public state;

    modifier condition(bool condition_) {
        require(condition_);
        _;
    }

    /// 只有买方可以调用这个函数。
    error OnlyBuyer();
    /// 只有卖方可以调用这个函数。
    error OnlySeller();
    /// 在当前状态下不能调用该函数。
    error InvalidState();
    /// 提供的值必须是偶数。
    error ValueNotEven();

    modifier onlyBuyer() {
        if (msg.sender != buyer)
```

(续下页)

(接上页)

```
            revert OnlyBuyer();
        _;
    }

    modifier onlySeller() {
        if (msg.sender != seller)
            revert OnlySeller();
        _;
    }

    modifier inState(State state_) {
        if (state != state_)
            revert InvalidState();
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();
    event SellerRefunded();

    // 确保 `msg.value` 是一个偶数。
    // 如果是奇数，除法会截断。
    // 通过乘法检查它不是一个奇数。
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        if ((2 * value) != msg.value)
            revert ValueNotEven();
    }

    /// 终止购买并收回 ether。
    /// 只能由卖方在合同锁定前能调用。
    function abort()
        external
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // 我们在这里直接使用 `transfer`。
        // 它可以安全地重入。
        // 因为它是这个函数中的最后一次调用，
```

(续下页)

(接上页)

```
        // 而且我们已经改变了状态。
        seller.transfer(address(this).balance);
    }

    /// 买方确认购买。
    /// 交易必须包括 `2 * value` ether。
    /// Ether 将被锁住，直到调用 confirmReceived。
    function confirmPurchase()
        external
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
        state = State.Locked;
    }

    /// 确认您（买方）已经收到了该物品。
    /// 这将释放锁定的 ether。
    function confirmReceived()
        external
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // 首先改变状态是很重要的，否则的话，
        // 下面使用 `send` 调用的合约可以在这里再次调用。
        state = State.Release;

        buyer.transfer(value);
    }

    /// 该功能为卖家退款，
    /// 即退还卖家锁定的资金。
    function refundSeller()
        external
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // 首先改变状态是很重要的，否则的话，
        // 下面使用 `send` 调用的合约可以在这里再次调用。
```

(续下页)

(接上页)

```
        state = State.Inactive;

        seller.transfer(3 * value);
    }
}
```

3.2.4 微支付通道

在这一节中，我们将学习如何建立一个支付通道的实施实例。它使用加密签名，使以太币在同一当事人之间的重复转移变得安全、即时，并且没有交易费用。对于这个例子，我们需要了解如何签名和验证签名，并设置支付通道。

创建和验证签名

想象一下，Alice 想发送一些以太给 Bob，即 Alice 是发送方，Bob 是接收方。

Alice 只需要在链下发送经过加密签名的信息 (例如通过电子邮件) 给 Bob，它类似于写支票。

Alice 和 Bob 使用签名来授权交易，这在以太坊的智能合约中是可以实现的。Alice 将建立一个简单的智能合约，让她传输以太币，但她不会自己调用一个函数来启动付款，而是让 Bob 来做，从而支付交易费用。

该合约将按以下方式运作：

1. Alice 部署了 `ReceiverPays` 合约，附加了足够的以太币来支付将要进行的付款。
2. Alice 通过用她的私钥签署一个消息来授权付款。
3. Alice 将经过加密签名的信息发送给 Bob。该信息不需要保密（后面会解释），而且发送机制也不重要。
4. Bob 通过向智能合约发送签名的信息来索取他的付款，合约验证了信息的真实性，然后释放资金。

创建签名

Alice 不需要与以太坊网络交互来签署交易，这个过程是完全离线的。在本教程中，我们将使用 web3.js 和 MetaMask 在浏览器中签署信息。使用 EIP-712 中描述的方法，因为它提供了许多其他安全优势。

```
/// 先进行哈希运算使事情变得更容易
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount, function () { console.log(
↪"Signed"); });
```

备注： `web3.eth.personal.sign` 把信息的长度加到签名数据中。由于我们先进行哈希运算，消息的长度总是正好是 32 字节，因此这个长度前缀总是相同的。

签署内容

对于履行付款的合同，签署的信息必须包括：

1. 收件人的钱包地址。
2. 要转移的金额。
3. 重放攻击的保护。

重放攻击是指一个已签署的信息被重复使用，以获得对第二次交易的授权。为了避免重放攻击，我们使用与以太坊交易本身相同的技术，即所谓的 nonce，它是一个账户发送的交易数量。智能合约会检查一个 nonce 是否被多次使用。

另一种类型的重放攻击可能发生在所有者部署 `ReceiverPays` 合约时，先进行了一些支付，然后销毁该合约。后来，他们决定再次部署 `RecipientPays` 合约，但新的合约不知道以前合约中使用的 nonces，所以攻击者可以再次使用旧的信息。

Alice 可以通过在消息中包含合约的地址来防止这种攻击，并且只有包含合约地址本身的消息才会被接受。您可以在本节末尾的完整合约的 `claimPayment()` 函数的前两行找到这个例子。

组装参数

既然我们已经确定了要在签名信息中包含哪些信息，我们准备把信息放在一起，进行哈希运算，然后签名。简单起见，我们把数据连接起来。ethereumjs-abi 库提供了一个名为 `soliditySHA3` 的函数，模仿 Solidity 的 `keccak256` 函数应用于使用 `abi.encodePacked` 编码的参数的行为。这里有一个 JavaScript 函数，为 `ReceiverPays` 的例子创建了适当的签名。

```
// recipient， 是应该被支付的地址。
// amount， 单位是 wei， 指定应该发送多少ether。
// nonce， 可以是任何唯一的数字，以防止重放攻击。
// contractAddress， 用于防止跨合约的重放攻击。
function signPayment(recipient, amount, nonce, contractAddress, callback) {
    var hash = "0x" + abi.soliditySHA3(
        ["address", "uint256", "uint256", "address"],
        [recipient, amount, nonce, contractAddress]
    ).toString("hex");

    web3.eth.personal.sign(hash, web3.eth.defaultAccount, callback);
}
```

在 Solidity 中恢复信息签名者

一般来说，ECDSA 的签名由两个参数组成，r 和 s。以太坊的签名包括第三个参数 v，您可以用它来验证是哪个账户的私钥被用来签署信息，以及作为交易的发送者。Solidity 提供了一个内置函数*ecrecover*，它接受一个消息以及 r, s 和 v 参数，然后返回用于签署该消息的地址。

提取签名参数

web3.js 产生的签名是 r, s 和 v 的拼接的，所以第一步是把这些参数分开。您可以在客户端这样做，但在智能合约内这样做意味着你只需要发送一个签名参数而不是三个。将一个字节数组分割成它的组成部分是很麻烦的，所以我们在 splitSignature 函数中使用*inline assembly* 完成这项工作（本节末尾的完整合约中的第三个函数）。

计算信息哈希值

智能合约需要确切地知道哪些参数用于签名，因此它必须通过参数重新创建消息，并使用该消息进行签名验证。在 claimPayment 函数中，函数 prefixed 和 recoverSigner 做了这件事。

完整的合约

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes memory signature)␣
↪external {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // 这将重新创建在客户端上签名的信息。
        bytes32 message = prefixed(keccak256(abi.encodePacked(msg.sender, amount,␣
↪nonce, this)));

        require(recoverSigner(message, signature) == owner);
```

(续下页)

(接上页)

```
        payable(msg.sender).transfer(amount);
    }

    /// 销毁合约并收回剩余的资金。
    function shutdown() external {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

    /// 签名方法。
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // 前32个字节，在长度前缀之后。
            r := mload(add(sig, 32))
            // 第二个32字节。
            s := mload(add(sig, 64))
            // 最后一个字节（下一个32字节的第一个字节）。
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// 构建一个前缀哈希值，以模仿 eth_sign 的行为。
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

编写一个简单的支付通道合约

Alice 现在建立了一个简单但完整的支付通道的实现。支付通道使用加密签名来安全、即时地重复转移以太币，并且没有交易费用。

什么是支付通道？

支付通道允许参与者在不使用交易的情况下重复转移以太币。这意味着，你可以避免与交易相关的延迟和费用。我们将探讨两方（Alice 和 Bob）之间一个简单的单向支付通道。它涉及三个步骤：

1. Alice 用以太币为智能合约提供资金。这就" 打开" 了支付通道。
2. Alice 签署信息，说明欠接收者多少以太币。这个步骤对每一笔付款都要重复进行。
3. Bob " 关闭" 支付通道，取出他的那部分以太币，并将剩余部分发回给发送方。

备注： 只有步骤 1 和 3 需要以太坊交易，意味着步骤 2 中发送方可以通过链下方法（如电子邮件）向接收方发送加密签名的信息。这意味着只需要两个交易就可以支持任何数量的转移。

Bob 保证会收到他的资金，因为智能合约托管了以太币，并兑现了一个有效的签名信息。智能合约也强制执行超时，所以即使接收者拒绝关闭通道，Alice 也能保证最终收回她的资金。由支付通道的参与者决定保持通道的开放时间。对于一个短暂的交易，如向网吧支付每分钟的网络访问费，支付通道可以保持有限的开放时间。另一方面，对于经常性的支付，如向雇员支付每小时的工资，支付渠道可能会保持开放几个月或几年。

开通支付渠道

为了开通支付通道，Alice 部署了智能合约，添加了要托管的以太币，并指定了预期接收者和通道存在的最长时间。这就是本节末尾合同中的函数 `SimplePaymentChannel`。

进行支付

Alice 通过向 Bob 发送签名信息进行支付。这一步骤完全在以太坊网络之外进行。消息由发送方加密签名，然后直接传送给接收方。

每条信息包括以下信息：

- 智能合约的地址，用于防止跨合约重放攻击。
- 到目前为止，欠接收方的以太币的总金额。

一个支付通道只关闭一次，就是在一系列转账结束后。正因为如此，所发送的签名信息中只有一个能被赎回。这就是为什么每条签名信息都指定了一个累计的以太币欠款总额，而不是单个小额支付的金额。接收方自然会选择最新的签名信息来赎回，因为那是总额最高的签名信息。每个签名信息的 nonce 不再需要了，因为智能合约只兑现一个签名信息。智能合约的地址仍然被用来防止一个支付渠道的签名信息被用于另一个渠道。

下面是经过修改的 JavaScript 代码，用于对上一节中的信息进行加密签名：

```
function constructPaymentMessage(contractAddress, amount) {
    return abi.soliditySHA3(
        ["address", "uint256"],
        [contractAddress, amount]
    );
}

function signMessage(message, callback) {
    web3.eth.personal.sign(
        "0x" + message.toString("hex"),
        web3.eth.defaultAccount,
        callback
    );
}

// contractAddress， 是用来防止跨合同的重放攻击。
// amount，单位是wei，指定了应该发送多少以太。

function signPayment(contractAddress, amount, callback) {
    var message = constructPaymentMessage(contractAddress, amount);
    signMessage(message, callback);
}
```

关闭支付通道

当 Bob 准备好接收他的资金时，是时候通过调用智能合约上的 `close` 函数关闭支付通道了。关闭通道会向接收者支付欠他们的以太币，并销毁合约，将任何剩余的以太币送回给 Alice。为了关闭通道，Bob 需要提供一个由 Alice 签名的信息。

智能合约必须验证该消息是否包含发送者的有效签名。进行这种验证的过程与接收者使用签名的过程相同。Solidity 函数 `isValidSignature` 和 `recoverSigner` 的工作方式与上一节中的 JavaScript 对应函数一样，而后者的函数是从 `ReceiverPays` 合约中借用的。

只有支付通道的接收者可以调用 `close` 函数，他们自然会传递最新的支付信息，因为该信息带有最高的欠款总额。如果允许发送者调用这个函数，他们可以提供一个金额较低的签名消息，骗取接收者的欠款。

该函数会验证签名的信息与给定的参数是否相符。如果一切正常，接收者就会收到他们的那部分以太币，而剩下的以太币将通过 `selfdestruct` 发送给发送者。您可以在完整的合约中看到 `close` 函数。

通道到期

Bob 可以在任何时候关闭支付通道，但如果他们没有这样做，Alice 需要一个方法来收回她的托管资金。在合同部署的时候，设置了一个 到期时间。一旦达到这个时间，Alice 可以调用 `claimTimeout` 来收回她的资金。您可以在完整的合约中看到 `claimTimeout` 函数。

在这个函数被调用后，Bob 不能再接收任何以太。所以 Bob 必须在过期前关闭通道，这一点很重要。

完整的合约

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告
contract SimplePaymentChannel {
    address payable public sender;      // 发送付款的账户。
    address payable public recipient;   // 接收付款的账户。
    uint256 public expiration;  // 超时时间，以防接收者永不关闭支付通道。

    constructor (address payable recipientAddress, uint256 duration)
        payable
    {
        sender = payable(msg.sender);
        recipient = recipientAddress;
        expiration = block.timestamp + duration;
    }

    /// 接收者可以在任何时候通过提供发送者签名的金额来关闭通道，
    /// 接收者将获得该金额，其余部分将返回发送者。
    function close(uint256 amount, bytes memory signature) external {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// 发送者可以在任何时候延长到期时间。
    function extend(uint256 newExpiration) external {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }
```

(续下页)

(接上页)

```
    /// 如果达到超时时间而接收者没有关闭通道，
    /// 那么以太就会被释放回给发送者。
    function claimTimeout() external {
        require(block.timestamp >= expiration);
        selfdestruct(sender);
    }

    function isValidSignature(uint256 amount, bytes memory signature)
        internal
        view
        returns (bool)
    {
        bytes32 message = prefixed(keccak256(abi.encodePacked(this, amount)));

        // 检查签名是否来自付款方。
        return recoverSigner(message, signature) == sender;
    }

    /// 下面的所有功能是取自 '创建和验证签名' 的章节。

    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // 前32个字节，在长度前缀之后。
            r := mload(add(sig, 32))
            // 第二个32字节。
            s := mload(add(sig, 64))
            // 最后一个字节（下一个32字节的第一个字节）。
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
```

(续下页)

(接上页)

```
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// 构建一个前缀哈希值，以模仿eth_sign的行为。
    function prefixed(bytes32 hash) internal pure returns (bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n32", hash));
    }
}
```

备注： 函数 `splitSignature` 并没有使用所有的安全检查。真正的实现应该使用更严格的测试库，例如 openzepplin 的 这个版本 的这个代码。

验证付款

不同与上一节，支付通道中的信息不会马上被兑换。接收者会跟踪最新的信息，并在关闭支付通道的时候赎回它。这意味着接收者对每条信息进行自行验证是至关重要的。否则就不能保证接收者最终能够得到付款。

接收者应使用以下程序验证每条信息：

1. 验证签名信息中的合约地址是否与支付通道相符。
2. 验证新的总额是否为预期的数额。
3. 确认新的总额不超过代管的以太币数额。
4. 验证签名是否有效，是否来自于支付通道的发送方。

我们将使用 ethereumjs-util 库来编写这个验证。最后一步可以用多种方式完成，我们使用 JavaScript。下面的代码借用了上面 **JavaScript 代码**中加密签名的 `constructPaymentMessage` 函数。

```
// 这模拟了eth_sign 的JSON-RPC构建前缀的方法。
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}

function recoverSigner(message, signature) {
    var split = ethereumjs.Util.fromRpcSig(signature);
```

(续下页)

(接上页)

```
    var publicKey = ethereumjs.Util.ecrecover(message, split.v, split.r, split.s);
    var signer = ethereumjs.Util.pubToAddress(publicKey).toString("hex");
    return signer;
}

function isValidSignature(contractAddress, amount, signature, expectedSigner) {
    var message = prefixed(constructPaymentMessage(contractAddress, amount));
    var signer = recoverSigner(message, signature);
    return signer.toLowerCase() ==
        ethereumjs.Util.stripHexPrefix(expectedSigner).toLowerCase();
}
```

3.2.5 模块化合约

用模块化的方法来构建您的合约，可以帮助减少复杂性，提高可读性，这将有助于在开发和代码审查中发现错误和漏洞。如果您单独指定且控制每个模块的行为，您必须考虑的相互作用只是模块之间的相互作用，而不是合约的其他每个灵活模块函数。在下面的例子中，合约使用 Balances 库 的 move 方法来检查地址之间发送的余额是否符合您的期望。通过这种方式，Balances 库提供了一个独立的组件，可以正确地跟踪账户的余额。很容易验证 Balances 库永远不会产生负的余额或溢出，所有余额的总和在合约的有效期内是一个不变的量。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

library Balances {
    function move(mapping(address => uint256) storage balances, address from, address␣
↪to, uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping(address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);
```

(续下页)

(接上页)

```
    function transfer(address to, uint amount) external returns (bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;

    }

    function transferFrom(address from, address to, uint amount) external returns␣
↪(bool success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) external returns (bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }

    function balanceOf(address tokenOwner) external view returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

3.3 安装 Solidity 编译器

3.3.1 版本

Solidity 的版本遵循 语义化版本原则。此外，主版本（例如：0.x.y）的补丁级版本的发布不会包含重大更改。这意味着用 0.x.y 版本编译的代码可望用 0.x.z 版本编译，其中 z > y。

除了发行版本外，我们还提供 **每日开发构建版本（nightly development builds）**，目的是使开发人员能够轻松地试用即将推出的功能并提供早期反馈。然而，请注意，虽然每日开发构建版本通常是很稳定的，但它们包含了来自开发分支的前沿代码，并不保证总是有效的。尽管我们尽了最大努力，它们仍可能含有未记录的或重大的修改，这些修改不会成为实际发布版本的一部分。它们也不会用于生产。

当开发智能合约时，您应该使用最新版本的 Solidity。这是因为重大的改变，以及新的特性和错误修复是定期引入的。我们目前使用 0.x 版本号 来表示这种快速的变化的。

3.3.2 Remix

我们推荐使用 *Remix* 来开发简单合约和快速学习 *Solidity*。

Remix 可以在线使用，而无需安装任何东西。如果您想离线使用，可按 https://github.com/ethereum/remix-live/tree/gh-pages 的页面说明下载 `.zip` 文件来使用。Remix 也是一个方便的选择，可以在不安装多个 Solidity 版本的情况下测试每日开发构建版本。

本页的进一步选项详细说明了在您的计算机上安装 Solidity 命令行编译器。如果您刚好要处理大型合约，或者需要更多的编译选项，那么您应该选择使用一个命令行编译器。

3.3.3 npm / Node.js

使用 `npm` 可以便捷地安装 `solcjs`，它一个 Solidity 编译器。但该 *solcjs* 程序的功能比本页下面描述的访问编译器的方法要少。在使用命令行编译器 一章中，我们假定您使用的是全功能的编译器: `solc`。`solcjs` 的用法在它自己的 代码仓库 中记录。

注意: *solc-js* 项目是通过使用 Emscripten 从 C++ 版的 *solc* 衍生出来的，这意味着两者使用相同的编译器源代码。因此，*solc-js* 可以直接用于 JavaScript 项目（如 Remix）具体介绍请参考 *solc-js* 代码库。

```
npm install -g solc
```

备注： 在命令行中，可执行文件被命名为 `solcjs`。

`solcjs` 的命令行选项与 `solc` 和一些工具（如 `geth`）是不兼容的，因此不要期望 `solcjs` 能像 `solc` 一样工作。

3.3.4 Docker

Solidity 构建的 Docker 镜像可以使用从 `ethereum` 组织获得的 `solc` 镜像。使用 `stable` 标签获取最新发布的版本，使用 `nightly` 标签获取开发分支中潜在的不稳定变更的版本。

Docker 镜像会运行编译器可执行文件，所以您可以把所有的编译器参数传给它。例如，下面的命令提取了稳定版的 `solc` 镜像（如果您还没有），并在一个新的容器中运行它，同时传递 `--help` 参数。

```
docker run ethereum/solc:stable --help
```

您也可以在标签中指定发行的版本，例如，0.5.4 版本。

```
docker run ethereum/solc:0.5.4 --help
```

要使用 Docker 镜像来编译主机上的 Solidity 文件，请安装一个本地文件夹用于输入和输出，并指定要编译的合约。例如:

```
docker run -v /local/path:/sources ethereum/solc:stable -o /sources/output --abi --
↪bin /sources/Contract.sol
```

您也可以使用标准的 JSON 接口（当使用工具化的编译器时建议使用这种方式）。当使用这个接口时，不需要装载任何目录，只要输入的 JSON 是自成一体的（即它没有引用任何外部文件，而这些文件必须要被由导入回调）。

```
docker run ethereum/solc:stable --standard-json < input.json > output.json
```

3.3.5 Linux 包

Solidity 的二进制安装包可在 solidity/releases 找到。

对于 Ubuntu ，我们也提供 PPAs 。通过以下命令，可获取最新的稳定版本：

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo apt-get update
sudo apt-get install solc
```

您也可以使用以下命令安装每日开发构建版本：

```
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install solc
```

此外，一些 Linux 发行版提供了他们自己的软件包。这些软件包不是由我们直接维护的，而通常由各自的软件包维护者保持最新。

例如，Arch Linux 也有最新开发版本的软件包。

```
pacman -S solidity
```

还有一个 snap 包，然而，它 **目前没有维护**。它可以安装在所有 支持的 Linux 发行版 。通过以下命令，安装最新的稳定版本的 solc：

```
sudo snap install solc
```

如果您想测试 develop 分支下的最新变更，请使用以下方式：

```
sudo snap install solc --edge
```

备注： `solc` snap 使用严格的限制。这对 snap 包来说是最安全的模式但它也有一些限制，比如只能访问 `/home`

和 /media 目录下的文件。欲了解更多信息，请访问 Demystifying Snap Confinement。

3.3.6 macOS Packages

我们通过 Homebrew 作为从源头建立的版本, 发布 Solidity 编译器,。目前不支持预构建。

```
brew update
brew upgrade
brew tap ethereum/ethereum
brew install solidity
```

要安装最新的 0.4.x/0.5.x 版本的 Solidity，您也可以分别使用 `brew install solidity@4` 和 `brew install solidity@5`。

如果您需要特定版本的 Solidity，您可以直接从 Github 上安装一个 Homebrew 列表。

参见 solidity.rb 在 Github 上的提交情况.

复制您想要的版本的提交哈希值，然后在您的机器上检出该分支。

```
git clone https://github.com/ethereum/homebrew-ethereum.git
cd homebrew-ethereum
git checkout <your-hash-goes-here>
```

使用 `brew` 安装:

```
brew unlink solidity
# 例如, 安装 0.4.8
brew install solidity.rb
```

3.3.7 静态二进制文件

我们在 solc-bin 上维护了一个包含过去和现在编译器版本的静态构建的资源库，用于所有支持的平台。您也可以找到每日开发构建版本。

该资源库不仅是一个快速且简单的方法，让终端用户获得可以开箱即用的二进制文件，而且它对第三方工具也很友好：

- 这些内容被镜像到 https://binaries.soliditylang.org，在那里可以很容易地通过 HTTPS 下载，没有任何认证、速率或需要使用 git 的限制。
- 提供的内容具有正确的 *Content-Type* 请求头和宽松的 CORS 配置，因此它可以被运行在浏览器中的工具直接加载。
- 二进制文件不需要安装或解包（与必要的 DLLs 捆绑在一起的旧版 Windows 除外）。

- 我们努力争取高水平的向后兼容性。文件一旦被添加，在没有提供旧位置的链接/重定向的情况下，不会被删除或移动。它们也不会被修改，而且应始终与原始校验相匹配。唯一的例外是破损或无法使用的文件，如果保持原样，有可能造成更大的伤害。
- 文件是通过 HTTP 和 HTTPS 提供的。只要您以安全的方式获得文件列表（通过 git、HTTPS、IPFS 或者只是在本地的缓存），并在下载后验证二进制文件的哈希值，您就不必通过 HTTPS 获得二进制文件。

在大多数情况下，同样的二进制文件可以在 Github 上的 Solidity 发布页 中找到。不同的是，我们一般不更新 Github 已发布的旧版本。这意味着如果命名规则改变，我们不会重新命名，也不会为发布时不支持的平台添加构建。这只发生在 `solc-bin` 资源库里。

`solc-bin` 资源库包含几个顶级目录，每个目录代表一个平台。每个目录都包含一个 `list.json` 文件，列出可用的二进制文件。例如，在 `emscripten-wasm32/list.json` 中您会发现以下关于 0.7.4 版本的信息。

```
{
  "path": "solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256": "0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3",
  "sha256": "0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2",
  "urls": [
    "bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1",
    "dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

这意味着：

- 您可以在同一目录下找到二进制文件，名称为 solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. 注意，该文件可能是一个软链接，如果您没有使用 git 下载，或者您的文件系统不支持软链接，您需要自己解决。
- 该二进制文件也被镜像在 https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js. 在这种情况下，不需要 git，软链接的解决方式是显而易见的，要么提供一个文件的副本，要么返回一个 HTTP 重定向。
- 该文件也可在 IPFS 上找到，地址是 QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS.
- 该文件将来可能会存储在 Swarm 上，地址是 16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1.
- 您可以通过比较其 keccak256 哈希值来验证二进制文件的完整性 `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3`。哈希值可以在命令行上使用 sha3sum 提供的 `keccak256sum` 工具或在 JavaScript 中使用 *ethereumjs-util* 的 *keccak256()* 函数。
- 您也可以通过比较二进制文件的 sha256 哈希值来验证它的完整性 `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2`。

警告： 由于高度的向后兼容性要求，版本库包含一些遗留元素，但您在编写新工具时应避免使用它们：

- 如果您想获得最佳的性能，请使用 `emscripten-wasm32/`（有回退功能的 `emscripten-asmjs/`）而不是 `bin/`。在 0.6.1 版本之前，我们只提供 asm.js 二进制文件。从 0.6.2 开始，我们改用 WebAssembly builds，性能好得多。我们已经为 wasm 重建了旧版本，但原来的 asm.js 文件仍然在 `bin/` 下。新的文件必须放在一个单独的目录中，以避免名称冲突。
- 如果您想确定下载的是 wasm 还是 asm.js 二进制文件，请使用 `emscripten-asmjs/` 和 `emscripten-wasm32/` 而不是 `bin/` 和 `wasm/` 目录。
- 使用 `list.json` 代替 `list.js` 和 `list.txt`。JSON 列表格式包含了旧列表的所有信息。
- 使用 https://binaries.soliditylang.org，而不是 https://solc-bin.ethereum.org。为了使事情简单化，我们把几乎所有与编译器有关的东西都移到了新的域名 `soliditylang.org` 下，这也适用于 `solc-bin`。虽然推荐使用新的域名，但旧的域名仍然被完全支持，并保证指向同一位置。

警告： 二进制文件也可以在 https://ethereum.github.io/solc-bin/ 找到，但这个页面在 0.7.2 版本发布后就停止了更新，不会收到任何平台的新版本或每日开发构建版本，也不提供新的目录结构，包括非 emscripten 的构建。

如果您正在使用它，请切换到 https://binaries.soliditylang.org，它是一个直接的替代。这使我们能够以透明的方式对底层主机进行更改，并尽量减少干扰。与我们无法控制的 `ethereum.github.io` 域名不同，`binaries.soliditylang.org` 可以保证长期运行并保持相同的 URL 结构。

3.3.8 从源代码编译

先决条件 - 所有操作系统

以下是 Solidity 构建的所有依赖性：

软件	备注
CMake （在 Windows 上为 3.21.3 以上版本，其他为 3.13 以上版）	跨平台构建文件生成器。
Boost （Windows 系统为 1.77 版本，其他系统 1.65 以上版）	C++ 库。
Git	用于获取源代码的命令行工具。
z3 （4.8.16 以上版本, 可选）	与 SMT 检查器一起使用。
cvc4 （可选）	与 SMT 检查器一起使用。

备注： 0.5.10 之前的 Solidity 版本可能无法与 Boost 1.70 以上版本正确链接。一个可能的解决方法是，在运行 cmake 命令配置 Solidity 之前，暂时重命名 `<Boost install path>/lib/cmake/Boost-1.70.0`。

从 0.5.10 开始，针对 Boost 1.70 以上版本的链接应该无需人工干预。

备注： 默认的构建配置需要一个特定的 Z3 版本（在代码最后更新时的最新版本）。Z3 版本之间的变化常常导致返回的结果略有不同（但仍然有效）。我们的 SMT 测试没有考虑到这些差异，很可能会在不同的版本中失败，而不是为其编写的版本。这并不意味着使用不同版本的构建是有问题的。如果将 `-DSTRICT_Z3_VERSION=OFF` 选项传递给 CMake，您可以使用任何满足上表要求的版本进行构建。然而，如果您这样做，请记得在 `scripts/tests.sh` 中传递 `--no-smt` 选项以跳过 SMT 测试。

备注： 默认情况下，编译是以 语义模式进行的，这将启用额外的警告，并告诉编译器将所有警告视为错误。这迫使开发人员在警告出现时进行修复，因此它们不会累积到 "以后再修复"。如果您只对创建发布版本感兴趣，不打算修改源代码来处理这些警告，您可以向 CMake 传递 `-DPEDANTIC=OFF` 选项来禁用这种模式。一般情况下不建议这样做，但在使用我们没有测试过的工具链或试图用较新的工具构建旧版本时，可能需要这样做。如果您遇到这种警告，请考虑 报告它们。

最小编译器版本

以下 C++ 编译器及其最小版本可构建 Solidity 代码库：

- GCC, 8 以上版本
- Clang, 7 以上版本
- MSVC, 2019 以上版本

先决条件 - macOS

对于 macOS 的构建，确保最新版本的 Xcode 已安装。这包含了 Clang C++ 编译器，Xcode IDE 和其他苹果公司的开发工具，这些工具是在 OS X 上构建 C++ 应用程序所必须的。如果您是第一次安装 Xcode，或者刚刚安装了一个新的版本，那么您在使用命令行构建前，需同意使用协议：

```
sudo xcodebuild -license accept
```

我们的 OS X 构建脚本使用 the Homebrew 软件包管理器来安装外部依赖。如果您想从头开始的话，以下是如何 卸载 Homebrew。

先决条件 - Windows

您需要为 Solidity 的 Windows 版本安装以下依赖软件包:

软件	备注
Visual Studio 2019 Build Tools	C++ 编译器
Visual Studio 2019 （可选）	C++ 编译器和开发环境。
Boost （1.77 版本）	C++ 库文件。

如果您已经有一个 IDE 并且只需要编译器和库文件。您可以安装 Visual Studio 2019 构建工具。

Visual Studio 2019 同时提供 IDE 和必要的编译器和库。所以，如果您没有一个 IDE，并且想要开发 Solidity，那么 Visual Studio 2019 将是一个可以使您轻松获得一切设置的选择。

以下是应在 Visual Studio 2019 构建工具或 Visual Studio 2019 中安装的组件列表:

- Visual Studio C++ core features
- VC++ 2019 v141 toolset (x86,x64)
- Windows Universal CRT SDK
- Windows 8.1 SDK
- C++/CLI support

我们有一个辅助脚本，您可以用它来安装所有需要的外部依赖:

```
scripts\install_deps.ps1
```

这将安装 `boost` 和 `cmake` 到 `deps` 子目录。

克隆代码库

执行以下命令，克隆源代码:

```
git clone --recursive https://github.com/ethereum/solidity.git
cd solidity
```

如果您想帮助开发 Solidity，您可以分叉 Solidity，然后将您个人的分叉库作为第二远程源添加。

```
git remote add personal git@github.com:[username]/solidity.git
```

备注: 这种方法将导致一个预发布的构建，例如，在这种编译器产生的每个字节码中设置一个标志。如果您想重新构建一个已发布的 Solidity 编译器，那么请使用 github 发布页上的源压缩包:

https://github.com/ethereum/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz

(而不是由 github 提供的" 源代码")。

命令行构建

请确保在构建前安装外部依赖项（见上文）。

Solidity 项目使用 CMake 来配置构建。您可能想安装 ccache 以加快重复构建的速度。CMake 会自动使用它。在 Linux、macOS 和其他 Unix 系统上构建 Solidity 方式都差不多：

```
mkdir build
cd build
cmake .. && make
```

或者在 Linux 和 macOS 上有更简单的方式，您可以运行：

```
#注意：这将在 usr/local/bin 安装 solc 和 soltest 的二进制文件。
./scripts/build.sh
```

> **警告：** BSD 构建应该也可以工作，但是 Solidity 团队没有测试过。

对于 Windows 执行：

```
mkdir build
cd build
cmake -G "Visual Studio 16 2019" ..
```

如果您想使用由 `scripts\install_deps.ps1` 安装的 boost 版本，您需要额外传递 `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` 和 `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` 作为参数给 `cmake` 调用。

这将会导致在构建目录中创建 **solidity.sln** 文件。双击该文件，Visual Studio 就会启动。我们建议创建 **Release** 配置，但其他的配置也可以。

或者，您可以在命令行上为 Windows 构建，像这样：

```
cmake --build . --config Release
```

3.3.9 CMake 选项

如果您对 CMake 的可选项感兴趣，可以运行 `cmake ... -LH`。

SMT 解算器

Solidity 可以针对 SMT 解算器进行构建，如果它们在系统中被发现，将默认为是这样做的。每个解算器都可以通过 `cmake` 选项禁用。

注意：在某些情况下，这也可以是构建失败后，可能的变通方法。

在构建文件夹内，您可以禁用它们，因为它们是默认启用的:

```
# 只禁用Z3 SMT解算器。
cmake .. -DUSE_Z3=OFF

# 只禁用CVC4 SMT解算器。
cmake .. -DUSE_CVC4=OFF

# 同时禁用Z3和CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.3.10 版本号字符串详解

Solidity 版本名包含四部分:

- 版本号
- 预发布版本标签，通常为 `develop.YYYY.MM.DD` 或者 `nightly.YYYY.MM.DD`
- 以 `commit.GITHASH` 格式展示的提交号
- 由若干条平台、编译器详细信息构成的平台标识

如果有本地修改，提交将会有后缀 `.mod`。

这些部分按照 Semver 的要求来组合，其中 Solidity 预发布版标签等价于 Semver 预发布版标签，而 Solidity 提交号和平台标识则组成 Semver 的构建元数据。

发布版样例: `0.4.8+commit.60cc1668.Emscripten.clang`。

预发布版样例: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`。

3.3.11 关于版本管理的重要信息

在版本发布之后，补丁版本号会增加，因为我们假定接下来只有补丁级别的变更。当变更被合并后，版本应该根据 Semver 和变更的重要程度来提升。最后，发行版本总是与当前每日开发构建版本本的版本号一致，但没有 `prerelease` 指示符。

示例:

1. 0.4.0 版本发布。
2. 从现在开始，每晚构建一个 0.4.1 版本。
3. 引入非重大变更——不改变版本号。
4. 引入重大变更——版本号提升到 0.5.0。
5. 0.5.0 版本发布。

该方式与*version pragma* 一起运行良好。

3.4 Solidity 源文件结构

源文件可以包含任意数量的*contract 定义*, *import* 指令, *pragma* 指令和*using for* 指令和*struct*, *enum*, *function*, *error* 以及*constant 变量* 的定义。

3.4.1 SPDX 许可标识符

如果智能合约的源代码是公开的，就可以更好地建立对智能合约的信任。由于提供源代码总是涉及到版权方面的法律问题，Solidity 编译器鼓励使用机器可读的 SPDX 许可标识符 。每个源文件都应该以一个注释开始，表明其许可证

```
// SPDX-License-Identifier: MIT
```

编译器不会验证许可证是否属于 SPDX 许可的列表，但它确实包括在字节码元数据（*bytecode metadata*）提供的字符串中。

如果您不想指定一个许可，或者源代码不是开源的，请使用特殊值 `UNLICENSED`。请注意，`UNLICENSED`（不允许使用，不存在于 SPDX 许可证列表中）与 `UNLICENSE` （授予所有人所有权利）不同。Solidity 遵循 npm 的推荐。

提供这个注释并不能使您摆脱与许可有关的其他义务，如必须在每个源文件中提到特定的许可头或原始版权人。

编译器可以在文件的任何位置识别该注释，但建议把它放在文件的顶部。

关于如何使用 SPDX 许可证标识的更多信息可以在 SPDX 网站 中找到。

3.4.2 编译指示

`pragma` 关键字用于启用某些编译器特性或检查。一个 pragma 指令始终是源文件的本地指令，所以如果您想在整个项目中使用 pragma 指令，您必须在您的所有文件中添加这个指令。如果您*import* 另一个文件，该文件的 pragma 指令 不会自动应用于导入文件。

版本编译指示

源文件可以（而且应该）用版本 pragma 指令来注释，以拒绝用未来的编译器版本进行编译，因为这可能会引入不兼容的变化。我们力图把这类变更做到尽可能小，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。正因为如此，至少在包含重大变化的版本中，通读一下更新日志总是一个好主意。这些版本总是有 `0.x.0` 或 `x.0.0` 形式的版本。

版本编译指示使用如下：`pragma solidity ^0.5.2;`

带有上述代码的源文件在 0.5.2 版本之前的编译器上不能编译，在 0.6.0 版本之后的编译器上也不能工作（这第二个条件是通过使用 ^ 添加的）。因为在 `0.6.0` 版本之前不会有任何重大的变化，所以您可以确信您的代码是按照您的预期编译的。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。

可以为编译器版本指定更复杂的规则，这些规则与 npm 使用相同的语法。

备注： 使用版本 pragma 指令 不会改变编译器的版本。它也 不会启用或禁用编译器的功能。它只是指示编译器检查它的版本是否与编译指示所要求的版本一致。如果不匹配，编译器会发出一个错误。

ABI 编码编译指示

通过使用 `pragma abicoder v1` 或 `pragma abicoder v2` ，您可以选择 ABI 编码器和解码器的两种实现。

新的 ABI 编码器（v2）能够对任意嵌套的数组和结构进行编码和解码。除了支持更多的类型外，它还涉及更广泛的验证和安全检查，这可能导致更高的气体成本，但也提高了安全性。从 Solidity 0.6.0 开始，它被认为是非实验性的，并且从 Solidity 0.8.0 开始，它被默认启用。旧的 ABI 编码器仍然可以使用 `pragma abicoder v1;` 来选择。

新编码器所支持的类型集是旧编码器所支持的类型的一个严格超集。使用新编码器的合约可以与不使用新编码器的合约进行交互，没有任何限制。只有当非 `abicoder v2` 的合约不试图进行需要解码新编码器支持的类型的调用时，才有可能出现相反的情况。编译器可以检测到这一点，并会发出一个错误。只要为您的合同启用 `abicoder v2` ，就足以使错误消失。

备注： 这个编译指示适用于激活它的文件中定义的所有代码，无论这些代码最终在哪里结束。这意味着，一个合约的源文件被选择用 ABI 编码器 v1 编译，它仍然可以包含通过从另一个合约继承来使用新编码器的代码。如果新类型只在内部使用，而不是在外部函数签名中使用，这是被允许的。

备注： 到 Solidity 0.7.4 为止，可以通过使用 `pragma experimental ABIEncoderV2` 来选择 ABI 编码器 v2，但不可能明确选择编码器 v1，因为它是默认的。

实验性编译指示

第二个编译指示是实验性的编译指示。它可以用来启用编译器或语言中尚未默认启用的功能。目前支持以下实验性编译指示：

ABI 编码器 V2

因为 ABI 编码器 v2 不再被认为是实验性的，它可以通过 `pragma abicoder v2` （请见上文）从 Solidity 0.7.4 开始选择。

SMT 检查器

这个组件必须在构建 Solidity 编译器时被启用，因此它不是在所有 Solidity 二进制文件中都可用。构建说明 解释了如何激活这个选项。它在大多数版本中为 Ubuntu PPA 版本激活，但不用于 Docker 镜像、Windows 二进制文件或静态构建的 Linux 二进制文件。如果您在本地安装了 SMT 检查器并通过节点（而不是通过浏览器）运行 solc-js，可以通过 smtCallback 为 solc-js 激活它。

如果您使用 `pragma experimental SMTChecker;`，那么您会得到额外的安全警告。这些警告是通过查询 SMT 求解器获得的。该组件还不支持 Solidity 语言的所有功能，可能会输出许多警告。如果它报告不支持的功能，那么分析可能不完全正确。

3.4.3 导入其他源文件

语法与语义

Solidity 支持导入语句，以帮助模块化您的代码，这些语句与 JavaScript 中可用的语句相似 (从 ES6 开始)。然而，Solidity 并不支持 默认导出 的概念。

在全局层面，您可以使用以下形式的导入语句：

```
import "filename";
```

`filename` 部分被称为 导入路径。该语句将所有来自 “filename” 的全局符号（以及在那里导入的符号）导入到当前的全局范围（与 ES6 中不同，但对 Solidity 来说是向后兼容的）。这种形式不建议使用，因为它不可预测地污染了命名空间。如果您在 “filename” 里面添加新的顶层项目，它们会自动出现在所有像这样从 “filename” 导入的文件中。最好是明确地导入特定的符号。

下面的例子创建了一个新的全局符号 `symbolName`，其成员均来自 `"filename"` 中全局符号；

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

这意味着所有全局符号以 `symbolName.symbol` 的格式提供。

另一种语法不属于 ES6，但可能是有用的：

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

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

如果有命名冲突，您可以在导入的同时重命名符号。例如，下面的代码创建了新的全局符号 `alias` 和 `symbol2`，它们分别从 `"filename"` 里面引用 `symbol1` 和 `symbol2`。

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

导入路径

为了能够在所有平台上支持可重复的构建，Solidity 编译器必须抽象出存储源文件的文件系统的细节。由于这个原因，导入路径并不直接指向主机文件系统中的文件。相反，编译器维护一个内部数据库（*虚拟文件系统* 或简称 *VFS*），每个源单元被分配一个唯一的 *源单元名称*，这是一个不透明的、非结构化的标识。在导入语句中指定的导入路径被转译成源单元名称，并用于在这个数据库中找到相应的源单元。

使用*标准 JSON* API，可以直接提供所有源文件的名称和内容作为编译器输入的一部分。在这种情况下，源单元的名称确实是任意的。然而，如果您想让编译器自动查找并将源代码加载到 VFS 中，您的源单元名称需要以一种结构化的方式，使*回调引用* 能够定位它们。当使用命令行编译器时，默认的回调引用只支持从主机文件系统加载源代码，这意味着您的源单元名称必须是路径。一些环境提供了自定义的回调，其用途更广。例如，Remix IDE 提供了一个可以让您 从 HTTP、IPFS 和 Swarm URL 导入文件，或者直接引用 NPM 注册表中的包。

关于虚拟文件系统和编译器使用的路径解析逻辑的完整描述，请参见*路径解析*。

3.4.4 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。
*/
```

备注： 单行注释由 UTF-8 编码中的任何单码行结束符（LF、VF、FF、CR、NEL、LS 或 PS）结束。终结符在注释之后仍然是源代码的一部分，所以如果它不是一个 ASCII 符号（这些是 NEL、LS 和 PS），将导致解析

器错误。

此外，还有一种注释叫做 NatSpec 注释，在格式指南 中详细说明。它们用三斜线（/// ）或双星号块（/** ... */ ）来写，它们应该直接用在函数声明或语句的上方。

3.5 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，错误，结构类型 和枚举类型 的声明，且合约可以从其他合约继承。

还有一些特殊种类的合同，叫做库合约 和接口合约。

在关于合约 的部分包含比本节更多的细节，它的作用是提供一个快速的概述。

3.5.1 状态变量

状态变量是指其值被永久地存储在合约存储中的变量。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

有效的状态变量类型请参阅类型 章节，对状态变量可见性的可能选择请参阅可见性和 *getter* 函数。

3.5.2 函数

函数是代码的可执行单位。通常在合约内定义函数，但它们也可以被定义在合约之外。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

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

// 定义在合约之外的辅助函数
```

(续下页)

(接上页)

```
function helper(uint x) pure returns (uint) {
    return x * 2;
}
```

函数调用 可以发生在内部或外部，并且对其他合约有不同程度的可见性。函数 接受参数并返回变量，以便在它们之间传递参数和值。

3.5.3 函数修饰器

函数修饰器可以被用来以声明的方式修改函数的语义 (见合约部分的函数修饰器)。

重载，也就是具有同一个修饰器的名字但有不同的参数，是不可能的。

与函数一样，修饰器也可以被重载。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract Purchase {
    address public seller;

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

    function abort() public view onlySeller { // 修饰器的使用
        // ...
    }
}
```

3.5.4 事件

事件是能方便地调用以太坊虚拟机日志功能的接口。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

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

(续下页)

(接上页)

```
    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。

3.5.5 错误

错误 (类型) 允许您为失败情况定义描述性的名称和数据。错误 (类型) 可以在回滚声明 中使用。与字符串描述相比，错误 (类型) 要便宜得多，并允许您对额外的数据进行编码。您可以使用 NatSpec 格式来向用户描述错误。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// 没有足够的资金用于转账。要求 `requested`。
/// 但只有 `available` 可用。
error NotEnoughFunds(uint requested, uint available);

contract Token {
    mapping(address => uint) balances;
    function transfer(address to, uint amount) public {
        uint balance = balances[msg.sender];
        if (balance < amount)
            revert NotEnoughFunds(amount, balance);
        balances[msg.sender] -= amount;
        balances[to] += amount;
        // ...
    }
}
```

更多信息请参阅合约章节中的错误和恢复语句。

3.5.6 结构类型

结构类型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）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

3.5.7 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）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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

3.6 类型

Solidity 是一种静态类型语言，这意味着每个变量（状态变量和局部变量）都需要被指定类型。Solidity 提供了几种基本类型，可以用来组合出复杂类型。

除此之外，各个类型之间可以在包含运算符号的表达式中进行交互。关于各种运算符的快速参考，可以参考运算符的优先顺序。

Solidity 中不存在" 未定义" 或" 空" 值的概念，但新声明的变量总是有一个取决于其类型的默认值。为了处理任何意外的值，您应该使用*revert* 函数 来恢复整个事务，或者返回一个带有第二个 `bool` 值的元组来表示成功。

3.6.1 值类型

以下被称为值类型，因为它们的变量总是按值传递，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。关键字 `uint8` 到 `uint256`（无符号整型，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)
- 移位运算符：`<<`（左移），`>>`（右移）
- 算数运算符：`+`，`-`，一元运算 `-`（只适用于有符号的整数），`*`，`/`，`%` (取余)，`**` (幂)

对于一个整数类型 `X`，您可以使用 `type(X).min` 和 `type(X).max` 来访问该类型代表的最小值和最大值。

> **警告：** Solidity 中的整数被限制在一个特定的范围内。例如，对于 `uint32`，这是 `0` 到 `2**32 - 1`。有两种模式在这些类型上进行算术。“包装” 或 “不检查” 模式和 “检查” 模式。默认情况下，算术总是 “检查” 模式的，这意味着如果一个操作的结果超出了该类型的值范围，调用将通过一个失败的断言 而被恢复。您可以用 `unchecked { ... }` 切换到 “不检查” 模式。更多的细节可以在关于不检查 的章节中找到。

比较运算

比较的值是通过比较整数值得到的值。

位运算

位操作是在数字的二进制补码表示上进行的。这意味着，例如 `~int256(0) == int256(-1)`。

移位运算

移位操作的结果具有左操作数的类型，将结果截断以符合类型。右操作数必须是无符号类型，试图对有符号类型进行移位会产生一个编译错误。

移位可以通过以下方式用 2 的幂的乘法来" 模拟"。请注意，对左边操作数类型的截断总是在最后进行，但没有明确提及。

- `x << y` 等同于数学表达式 `x * 2**y`。
- `x >> y` 等同于数学表达式 `x / 2**y`，向负无穷远的方向取整。

> **警告：** 在 `0.5.0` 版本之前，负数 `x` 的右移 `x >> y` 相当于数学表达式 `x / 2**y` 向零舍入，即右移使用向上舍入（向零舍入）而不是向下舍入（向负无穷大）。

备注： 就像对算术操作那样，对移位操作从不进行溢出检查。相反，结果总是被截断的。

加法、减法和乘法

加法、减法和乘法具有通常的语义，在上溢和下溢方面有两种不同的模式：

默认情况下，所有的算术都会被检查是否有下溢或上溢，但这可以用 不检查限制 来禁用。这会导致包装的算术。更多细节可以在那一节中找到。

表达式 `-x` 等同于 `(T(0) - x)`，其中 `T` 是 `x` 的类型。它只能用于有符号的类型。如果 `x` 是负的，`-x` 的值就是正的。还有一个注意事项也是由二进制补码表示产生的：

如果您有（这样的表达式）`int x = type(int).min;`，那么 `-x` 就不符合正数范围。这意味着 `unchecked { assert(-x == x); }` 可以工作，而表达式 `-x` 在检查模式下使用时将导致断言失败。

除法

由于运算结果的类型总是操作数之一的类型，整数除法的结果总是一个整数。在 Solidity 中，除法是向零进位的。这意味着 `int256(-5) / int256(2) == int256(-2)`。

请注意，与此相反，在字面上 的除法会产生任意精度的分数值。

备注： 除以 0 会导致异常。这个检查 **不能**通过 `unchecked { ... }` 禁用。

备注： 表达式 `type(int).min / (-1)` 是除法导致溢出的唯一情况。在检查算术模式下，这将导致一个失败的断言，而在包装模式下，值将是 `type(int).min`。

取余

模数运算 `a % n` 是操作数 `a` 除以操作数 `n` 后产生余数 `r`，其中 `q = int(a / n)` 和 `r = a - (n * q)`。这意味着模数运算结果与它的左边操作数（或零）拥有相同的符号，`a % n == -(-a % n)` 对负的 `a` 来说成立。

- `int256(5) % int256(2) == int256(1)`
- `int256(5) % int256(-2) == int256(1)`
- `int256(-5) % int256(2) == int256(-1)`
- `int256(-5) % int256(-2) == int256(-1)`

备注： 对 0 取余会导致异常。这个检查 **不能**通过 `unchecked { ... }` 禁用。

幂运算

幂运算只适用于指数中的无符号类型。幂运算的结果类型总是等于基数的类型。请注意，它要足够大以容纳结果，并为潜在的断言失败或包装行为做好准备。

备注： 在检查模式下，幂运算只对小基数使用相对便宜的 `exp` 操作码。对于 `x**3` 的情况，表达式 `x*x*x` 可能更便宜。在任何情况下，气体成本测试和使用优化器都是可取的。

备注： 请注意，`0**0` 被 EVM 定义为 `1`。

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 `ufixed128x18` 和 `fixed128x18` 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 `-`, `*`, `/`, `%` (取余数)

> **备注:** 浮点型（在许多语言中的 `float` 和 `double` ，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

地址类型有两种基本相同的类型:

- `address`: 保存一个 20 字节的值（一个以太坊地址的大小）。
- `address payable`: 与 `address` 类型相同，但有额外的方法 `transfer` 和 `send`。

这种区别背后的想法是，`address payable` 是一个您可以发送以太币的地址，而您不应该发送以太币给一个普通的 `address`，例如，因为它可能是一个智能合约，而这个合约不是为接受以太币而建立的。

类型转换:

允许从 `address payable` 到 `address` 的隐式转换，而从 `address` 到 `address payable` 的转换必须通过 `payable(<address>)` 来明确。

对于 `uint160`、整数、`bytes20` 和合约类型，允许对 `address` 进行明确的转换和输出。

只有 `address` 类型和合约类型的表达式可以通过 `payable(...)` 显式转换为 `address payable` 类型。对于合约类型，只有在合约可以接收以太的情况下才允许这种转换，也就是说，合约要么有一个*receive* 函数，要么有一个 payable 类型的 fallback 的函数。请注意，`payable(0)` 是有效的，是这个规则的例外。

> **备注:** 如果您需要一个 `address` 类型的变量，并计划向其发送以太，那么就将其类型声明为 `address payable`，以使这一要求可行。另外，尽量尽早地进行这种区分或转换。

address 和 address payable 之间的区别是从 0.5.0 版本开始的。同样从该版本开始，合约不能隐式地转换为 address 类型，但仍然可以显式地转换为 address 或 address payable，如果它们有一个 receive 或 payable 类型的 fallback 函数的话。

运算符：

- <=, <, ==, !=, >= 和 >

> **警告：** 如果您使用较大字节的类型转换为 address，例如 bytes32，那么 address 就被截断了。为了减少转换的模糊性，从 0.4.24 版本开始，编译器将强迫您在转换中明确地进行截断处理。以 32 字节的值 0x111122223333444455556666777788889999AAAABBBBCCCCDDDDEEEEFFFFCCCC 为例。
>
> 您可以使用 address(uint160(bytes20(b)))，结果是 0x111122223333444455556666777788889999aAaa，或者您可以使用 address(uint160(uint256(b)))，结果是 0x777788889999AaAAbBbbCcccddDdeeeEfFFfCcCc。

备注： 符合 EIP-55 的混合大小写十六进制数字会自动被视为 address 类型的字面数字。参见地址字面类型。

地址类型成员变量

快速参考，请见地址类型的成员。

- balance 和 transfer

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

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

如果当前合约的余额不足，或者以太币转账被接收账户拒收，那么 transfer 功能就会失败。transfer 功能在失败后会被还原。

备注： 如果 x 是一个合约地址，它的代码（更具体地说：它的接收以太的函数，如果有的话，或者它的 *Fallback* 函数，如果有的话）将与 transfer 调用一起执行（这是 EVM 的一个特性，无法阻止）。如果执行过程中耗尽了气体或出现了任何故障，以太币的转移将被还原，当前的合约将以异常的方式停止。

- send

send 是 transfer 的低级对应部分。如果执行失败，当前的合约不会因异常而停止，但 send 会返回 false。

> **警告：** 使用 send 有一些危险：如果调用堆栈深度为 1024，传输就会失败（这可以由调用者强制执行），如果接收者的气体耗尽，也会失败。因此，为了安全地进行以太币转账，一定要检查 send 的返回值，或者使用 transfer，甚至使用更好的方式：使用收款人提款的模式。

- call, delegatecall 和 staticcall

为了与不遵守 ABI 的合约对接，或者为了更直接地控制编码，我们提供了 call，delegatecall 和 staticcall 函数。它们都接受一个 bytes memory 参数，并返回成功条件（作为一个 bool）和返回的数据（bytes memory）。函数 abi.encode, abi.encodePacked, abi.encodeWithSelector 和 abi.encodeWithSignature 可以用来编码结构化的数据。

示例：

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

> **警告：** 所有这些函数都是低级别的函数，应该谨慎使用。具体来说，任何未知的合约都可能是恶意的，如果您调用它，您就把控制权交给了该合约，而该合约又可能回调到您的合约中，所以要准备好在调用返回时改变您合约的状态变量。与其他合约互动的常规方法是在合约对象上调用一个函数（x.f()）。

备注： 以前的 Solidity 版本允许这些函数接收任意的参数，并且也会以不同的方式处理 bytes4 类型的第一个参数。这些边缘情况在 0.5.0 版本中被移除。

可以用 gas 修饰器来调整所提供的气体：

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature("register(string)",
↪"MyName"));
```

同样，所提供的以太值也可以被控制：

```
address(nameReg).call{value: 1 ether}(abi.encodeWithSignature("register(string)",
↪"MyName"));
```

最后，这些修饰器可以合并。它们的顺序并不重要：

```
address(nameReg).call{gas: 1000000, value: 1 ether}(abi.encodeWithSignature(
↪"register(string)", "MyName"));
```

以类似的方式，可以使用函数 delegatecall：不同的是，它只使用给定地址的代码，所有其他方面（存储，余额，...）都取自当前的合约。delegatecall 的目的是为了使用存储在另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合使用 delegatecall。

备注： 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`。这个功能在 0.5.0 版本中被移除。

从 byzantium 开始，也可以使用 `staticcall`。这基本上与 `call` 相同，但如果被调用的函数以任何方式修改了状态，则会恢复。

这三个函数 `call`，`delegatecall` 和 `staticcall` 都是非常低级的函数，只应该作为 *最后的手段* 来使用，因为它们破坏了 Solidity 的类型安全。

`gas` 选项在所有三种方法中都可用，而 `value` 选项只在 `call` 中可用。

备注： 最好避免在您的智能合约代码中依赖硬编码的气体值，无论状态是读出还是写入，因为这可能有很多隐患。另外，对气体的访问在未来可能会改变。

- `code` 和 `codehash`

您可以查询任何智能合约的部署代码。使用 `.code` 获得作为 `bytes memory` 的 EVM 字节码，这可能是空的。使用 `.codehash` 获得该代码的 Keccak-256 哈希值（作为 `bytes32`）。注意，使用 `addr.codehash` 比 `keccak256(addr.code)` 更便宜。

备注： 所有的合约都可以转换为 `address` 类型，所以可以用 `address(this).balance` 查询当前合约的余额。

合约类型

每个合约 都定义了自己的类型。您可以隐式地将一个合约转换为它们所继承的另一个合约。合约可以显式地转换为 `address` 类型，也可以从 `address` 类型中转换。

只有在合约类型具有 receive 或 payable 类型的 fallback 函数的情况下，才有可能明确转换为 `address payable` 类型和从该类型转换。这种转换仍然使用 `address(x)` 进行转换。如果合约类型没有一个 receive 或 payable 类型的 fallback 函数，可以使用 `payable(address(x))` 来转换为 `address payable`。您可以在地址类型 一节中找到更多信息。

备注： 在 0.5.0 版本之前，合约直接从地址类型派生出来，并且在 `address` 和 `address payable` 之间没有区别。

如果您声明了一个本地类型的变量（`MyContract c`），您可以调用该合约上的函数。注意要从相同合约类型的地方将其赋值。

您也可以实例化合约（这意味着它们是新创建的）。您可以在'*通过关键字 new 创建合约*'部分找到更多细节。

合约的数据表示与 `address` 类型相同，该类型也用于*ABI*。

合约不支持任何运算符。

合约类型的成员是合约的外部函数，包括任何标记为 `public` 的状态变量。

对于一个合约 `C`，您可以使用 `type(C)` 来访问关于该合约的类型信息 。

定长字节数组

值类型 `bytes1`, `bytes2`, `bytes3`, ..., `bytes32` 代表从 1 到 32 的字节序列。

运算符：

比较运算符：<=，<，==，!=，>=，>（返回布尔型）

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

移位运算符以无符号的整数类型作为右操作数（但返回左操作数的类型），它表示要移位的位数。有符号类型的移位将产生一个编译错误。

成员变量：

- `.length` 表示这个字节数组的长度（只读）.

备注： 类型 `bytes1[]` 是一个字节数组，但是由于填充规则，它为每个元素浪费了 31 个字节的空间（在存储中除外）。因此最好使用 `bytes` 类型来代替。

备注： 在 0.8.0 版本之前，`byte` 曾经是 `bytes1` 的别名。

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测试的十六进制字属于 `address` 类型。十六进制字数在 39 到 41 位之间，并且没有通过校验测试，会产生一个错误。您可以预加（对于整数类型）或附加（对于 bytesNN 类型）零来消除该错误。

备注： 混合大小写的地址校验和格式定义在 EIP-55。

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` 表示十进制数字 69。Solidity 中是没有八进制的，因此前置 0 是无效的。

小数字面常数由 `.` 和小数点后的至少一个数字组成。例如，`.1` 和 `1.3`` （但不是 ``1.`）。

也支持 `2e10` 形式的科学符号，其中尾数可以是小数，但指数必须是一个整数。字面的 `MeE` 相当于 `M * 10**E`。例子包括 `2e10`, `-2e10`, `2e-10`, `2.5e1`。

下划线可以用来分隔数字字面的数字，以帮助阅读。例如，十进制 `123_000`，十六进制 `0x2eff_abde`，科学十进制 `1_2e345_678` 都是有效的。下划线只允许在两个数字之间，并且只允许一个连续的下划线。含有下划线的数字字面没有额外的语义，下划线被忽略。

数值字面常数表达式保留任意精度，直到它们被转换为非字面常数类型（即通过与数字字面常数表达式以外的任何东西一起使用（如布尔字面常数）或通过显式转换）。这意味着在数值常量表达式中，计算不会溢出，除法不会截断。

例如，`(2**800 + 1) - 2**800` 的结果是常数 `1` （类型 `uint8`），尽管中间的结果甚至不符合机器字的大小。此外，`.5 * 8` 的结果是整数 `4` （尽管中间使用了非整数）。

警告： 虽然大多数运算符在应用于字面常数时都会产生一个字面常数表达式，但有一些运算符并不遵循这种模式：

- 三元运算符（`... ? ...: ...`）。
- 数组下标（`<array>[<index>]`）。

您可能期望像 `255 + (true ? 1 : 0)` 或 `255 + [1, 2, 3][0]` 这样的表达式等同于直接使用字面常数 256，但实际上它们是在 `uint8` 类型中计算的，可能会溢出。

只要操作数是整数，任何可以应用于整数的操作数也可以应用于数值字面常数表达式。如果两者中的任何一个是小数，则不允许进行位操作，如果指数是小数，则不允许进行幂运算（因为这可能导致无理数）。

以数值字面常数表达式为左（或基数）操作数，以整数类型为右（指数）操作数的移位和幂运算，总是在 `uint256` （非负数数值字面常数）或 `int256` （负数数值字面常数）类型中进行。无论右（指数）操作数的类型如何。

警告： 在 0.4.0 版本之前，Solidity 中整数字的除法会被截断，但现在它转换为一个有理数，即 `5 / 2` 不等于 `2`，而是 `2.5`。

备注： 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 3 的数值字面常数类型相同。

备注： 数字字面表达式一旦与非字面表达式一起使用，就会被转换为非字面类型。不考虑类型，下面分配给 `b` 的表达式的值被评估为一个整数。因为 `a` 的类型是 `uint128`，所以表达式 `2.5 + a` 必须有一个合适的类型。由于 `2.5` 和 `uint128` 的类型没有共同的类型，Solidity 编译器不接受这段代码。

```
uint128 a = 1;
uint128 b = 2.5 + a + 0.5;
```

字符串字面常数和类型

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'`）。它们也可以分成多个连续部分（`"foo" "bar"` 相当于 `"foobar"`），这在处理长字符串时很有帮助。它们不像在 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`。

例如，使用 `bytes32 samevar = "stringliteral"`，当分配给 `bytes32` 类型时，字符串字面常数被解释成原始字节形式。

字符串字面常数只能包含可打印的 ASCII 字符，也就是 0x20 ... 0x7E 之间的字符。

此外，字符串字元还支持以下转义字符：

- `\<newline>`（转义一个实际的换行）
- `\\`（反斜杠）
- `\'`（单引号）
- `\"`（双引号）
- `\n`（换行）
- `\r`（回车键）
- `\t`（制表）
- `\xNN`（十六进制转义，见下文）

- `\uNNNN` （unicode 转义，见下文）

`\xNN` 接收一个十六进制值并插入相应的字节，而 `\uNNNN` 接收一个 Unicode 编码点并插入一个 UTF-8 序列。

备注： 在 0.8.0 版本之前，有三个额外的转义序列。`\b`，`\f` 和 `v`。它们在其他语言中通常是可用的，但在实践中很少需要。如果您确实需要它们，仍然可以通过十六进制转义插入，即分别为 `\x08`，`x0c` 和 `\x0b`，就像其他 ASCII 字符一样。

下面例子中的字符串的长度为 10 个字节。它以一个换行字节开始，接着是一个双引号，一个单引号，一个反斜杠字符，然后（没有分隔符）是字符序列 `abcdef`。

```
"\n\"\'\\abc\
def"
```

任何非换行的 Unicode 行结束符（即 LF, VF, FF, CR, NEL, LS, PS）都被认为是字符串字面的结束。换行只在字符串字面内容前面没有 `\` 的情况下终止。

Unicode 字面量

普通字符串字面常数只能包含 ASCII 码，而 Unicode 字面常数 - 以关键字 `unicode` 为前缀 - 可以包含任何有效的 UTF-8 序列。它们也支持与普通字符串字面意义相同的转义序列。

```
string memory a = unicode"Hello �";
```

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（`hex"001122FF"`，`hex'0011_22_FF'`）。它们的内容必须是十六进制的数字，可以选择使用一个下划线作为字节边界之间的分隔符。字面的值将是十六进制序列的二进制表示。

由空格分隔的多个十六进制字面常数被串联成一个字面常数：`hex"00112233" hex"44556677"` 相当于 `hex"0011223344556677"`。

十六进制字面常数的行为与 字符串字面常数 类似，但是不能隐式转换为 `string` 类型。

枚举类型

枚举是在 Solidity 中创建用户定义类型的一种方式。它们可以显式地转换为所有整数类型，和从整数类型来转换，但不允许隐式转换。从整数的显式转换在运行时检查该值是否在枚举的范围内，否则会导致异常。枚举要求至少有一个成员，其声明时的默认值是第一个成员。枚举不能有超过 256 个成员。

数据表示与 C 语言中的枚举相同。选项由后续的从 `0` 开始无符号整数值表示。

使用 `type(NameOfEnum).min` 和 `type(NameOfEnum).max` 您可以得到给定枚举的最小值和最大值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

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)"。
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

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

    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

备注： 枚举也可以在文件级别上声明，在合约或库定义之外。

用户定义的值类型

一个用户定义的值类型允许在一个基本的值类型上创建一个零成本的抽象。这类似于一个别名，但有更严格的类型要求。

一个用户定义的值类型是用 `type C is V` 定义的，其中 `C` 是新引入的类型的名称，`V` 必须是一个内置的值类型（"底层类型"）。函数 `C.wrap` 被用来从底层类型转换到自定义类型。同样地，函数 `C.unwrap` 被用来从自定义类型转换到底层类型。

类型 `C` 没有任何运算符或附加成员函数。特别是，甚至运算符 `==` 也没有定义。不允许对其他类型进行显式

和隐式转换。

这种类型的值的数据表示是从底层类型中继承的，底层类型也被用于 ABI 中。

下面的例子说明了一个自定义类型 UFixed256x18，代表一个有 18 位小数的十进制定点类型和一个最小的库来对该类型做算术运算。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

// 使用用户定义的值类型表示一个18位小数，256位宽的定点类型。
type UFixed256x18 is uint256;

/// 一个在UFixed256x18上进行定点操作的最小库。
library FixedMath {
    uint constant multiplier = 10**18;

    /// 将两个UFixed256x18的数字相加。溢出时将返回，依靠uint256的算术检查。
    function add(UFixed256x18 a, UFixed256x18 b) internal pure returns (UFixed256x18)
↪{
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) + UFixed256x18.unwrap(b));
    }
    /// 将UFixed256x18和uint256相乘。溢出时将返回，依靠uint256的算术检查。
    function mul(UFixed256x18 a, uint256 b) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(UFixed256x18.unwrap(a) * b);
    }
    /// 对一个UFixed256x18类型的数字相下取整。
    /// @return 不超过 `a` 的最大整数。
    function floor(UFixed256x18 a) internal pure returns (uint256) {
        return UFixed256x18.unwrap(a) / multiplier;
    }
    /// 将一个uint256转化为相同值的UFixed256x18。
    /// 如果整数太大，则恢复计算。
    function toUFixed256x18(uint256 a) internal pure returns (UFixed256x18) {
        return UFixed256x18.wrap(a * multiplier);
    }
}
```

注意 UFixed256x18.wrap 和 FixedMath.toUFixed256x18 有相同的签名，但执行两个非常不同的操作。UFixed256x18.wrap 函数返回一个与输入的数据表示相同的 UFixed256x18，而 toUFixed256x18 则返回一个具有相同数值的 UFixed256x18。

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|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。

默认情况下，函数类型是内部函数，所以可以省略 `internal` 关键字。注意，这只适用于函数类型。对于合约中定义的函数，必须明确指定其可见性，它们没有默认类型。

转换：

当且仅当它们的参数类型相同，它们的返回类型相同，它们的内部/外部属性相同，并且 `A` 的状态可变性比 `B` 的状态可变性更具限制性时，一个函数类型 `A` 就可以隐式转换为一个函数类型 `B`。特别是：

- `pure` 函数可以转换为 `view` 和 `非 payable` 函数
- `view` 函数可以转换为 `非 payable` 函数
- `payable` 函数可以转换为 `非 payable` 函数

其他函数类型之间的转换是不可能的。

关于 `payable` 和 `非 payable` 的规则可能有点混乱，但实质上，如果一个函数是 `payable`，这意味着它也接受零以太的支付，所以它也是 `非 payable`。另一方面，一个 `非 payable` 的函数将拒收发送给它的以太，所以 `非 payable` 的函数不能被转换为 `payable` 的函数。声明一下，拒收以太比不拒收以太更有限制性。这意味着您可以用一个不可支付的函数覆写一个可支付的函数，但不能反过来。

此外，当您定义一个 `非 payable` 的函数指针时，编译器并不强制要求被指向的函数实际拒收以太。相反，它强制要求该函数指针永远不会被用来发送以太。这使得我们有可能将一个 `payable` 的函数指针分配给一个 `非 payable` 的函数指针，以确保这两种类型的函数表现相同，即都不能用来发送以太。

如果一个函数类型的变量没有被初始化，调用它将导致会出现异常。如果您在一个函数上使用了 `delete` 之后再调用它，也会发生同样的情况。

如果外部函数类型在 Solidity 的上下文中被使用，它们将被视为 `function` 类型，它将地址和函数标识符一起编码为一个 `bytes24` 类型。

请注意，当前合约的公共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` 。

一个内部类型的函数可以被分配给一个内部函数类型的变量，无论它在哪里被定义。这包括合约和库的私有，内部和公共函数，以及自由函数。另一方面，外部函数类型只与公共和外部合约函数兼容。

备注： 带有 `calldata` 参数的外部函数与带有 `calldata` 参数的外部函数类型不兼容。它们与相应的带有 `memory` 参数的类型兼容。例如，没有一个函数可以被 `function (string calldata) external` 类型的值所指向，而 `function (string memory) external` 可以同时指向 `function f(string memory) external {}` 和 `function g(string calldata) external {}`。这是因为对于这两个位置，参数是以同样的方式传递给函数的。调用者不能直接将其 calldata 传递给外部函数，总是 ABI 将参数编码到内存中。将参数标记为 `calldata` 只影响到外部函数的实现，在调用者一方的函数指针中是没有意义的。

库合约被排除在外，因为它们需要 `delegatecall`，并且对它们的选择器使用不同的 *ABI* 约定。接口中声明的函数没有定义，所以指向它们也没有意义。

成员：外部（或公共）函数有以下成员：

- `.address` 返回该函数的合约地址。
- `.selector` 返回 *ABI* 函数选择器

备注： 外部（或公共）函数曾经有额外的成员 `.gas(uint)` 和 `.value(uint)`。这些在 Solidity 0.6.2 中被废弃，并在 Solidity 0.7.0 中被移除。取而代之的是使用 `{gas: ...}` 和 `{value: ...}` 来分别指定发送到函数的气体量或以太（wei 为单位）量。参见外部函数调用 以获得更多信息。

以下例子展示如何使用这些成员：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.4 <0.9.0;

contract Example {
    function f() public payable returns (bytes4) {
        assert(this.f.address == address(this));
        return this.f.selector;
    }

    function g() public {
        this.f{gas: 10, value: 800}();
    }
}
```

以下例子展示如何使用内部函数类型：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library ArrayUtils {
    // 内部函数可以在内部库函数中使用，因为它们将是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);
```

(续下页)

(接上页)

```
    }

    function square(uint x) internal pure returns (uint) {
        return x * x;
    }

    function sum(uint x, uint y) internal pure returns (uint) {
        return x + y;
    }
}
```

另一个使用外部函数类型的例子:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


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

    Request[] private requests;
    event NewRequest(uint);

    function query(bytes memory data, function(uint) external callback) public {
        requests.push(Request(data, callback));
        emit NewRequest(requests.length - 1);
    }

    function reply(uint requestID, uint response) public {
        // 这里要检查的是调用返回是否来自可信的来源
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST =
↪Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa)); // 已知的合约
    uint private exchangeRate;

    function buySomething() public {
```

(续下页)

(接上页)

```
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

    function oracleResponse(uint response) public {
        require(
            msg.sender == address(ORACLE_CONST),
            "Only oracle can call this."
        );
        exchangeRate = response;
    }
}
```

备注： Lambda 或内联函数是计划中的，但还不支持。

3.6.2 引用类型

引用类型的值可以通过多个不同的名称进行修改。这与值类型形成鲜明对比，在值类型的变量被使用时，您会得到一个独立的副本。正因为如此，对引用类型的处理要比对值类型的处理更加谨慎。目前，引用类型包括结构、数组和映射。如果您使用一个引用类型，您必须明确地提供存储该类型的数据区域。`memory` （其寿命限于外部函数调用），`storage` （存储状态变量的位置，其寿命限于合约的寿命）或 `calldata` （包含函数参数的特殊数据位置）。

改变数据位置的赋值或类型转换将总是导致自动复制操作，而同一数据位置内的赋值只在某些情况下对存储类型进行复制。

数据位置

每个引用类型都有一个额外的属性，即" 数据位置"，关于它的存储位置。有三个数据位置。`memory`, `storage` 和 `calldata`。Calldata 是一个不可修改的、非持久性的区域，用于存储函数参数，其行为主要类似于 memory。

备注： 如果可以的话，尽量使用 `calldata` 作为数据位置，因为这样可以避免复制，也可以确保数据不能被修改。使用 `calldata` 数据位置的数组和结构也可以从函数中返回，但不可能分配这种类型。

备注： 在 0.6.9 版本之前，引用型参数的数据位置被限制在外部函数中的 `calldata`，公开函数中的 `memory`，以及内部和私有函数中的 `memory` 或 `storage`。现在 `memory` 和 `calldata` 在所有函数中都被允许使用，无论其可见性如何。

备注： 在 0.5.0 版本之前，数据位置可以省略，并且会根据变量的种类、函数类型等默认为不同的位置，但现在所有的复杂类型都必须给出一个明确的数据位置。

数据位置和分配行为

数据位置不仅与数据的持久性有关，而且也与分配的语义有关：

- 在 storage 和 memory 之间的分配（或从 calldata 中分配）总是创建一个独立的拷贝。
- 从 memory 到 memory 的赋值只创建引用。这意味着对一个内存变量的改变在所有其他引用相同数据的内存变量中也是可见的。
- 从 storage 到 **local** 存储变量的赋值也只赋值一个引用。
- 所有其他对 storage 的赋值总是拷贝的。这种情况的例子是对状态变量或存储结构类型的局部变量成员的赋值，即使局部变量本身只是一个引用。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // x 的数据存储位置是 storage。
    // 这是唯一可以省略数据位置的地方。
    uint[] x;

    // memoryArray 的数据存储位置是 memory。
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // 将整个数组拷贝到 storage 中，可行
        uint[] storage 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.pop(); // 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;
        // 同样， "delete y" 也是无效的，
        // 因为对引用存储对象的局部变量的赋值只能从现有的存储对象中进行。
        // 它将 “重置” 指针，但没有任何合理的位置可以指向它。
        // 更多细节见 "delete" 操作符的文档。
        // 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
    }
```

(续下页)

(接上页)

```
    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

数组

数组可以在声明时指定长度，也可以动态调整大小。

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[]`。

例如，一个由 5 个 `uint` 的动态数组组成的数组被写成 `uint[][5]`。与其他一些语言相比，这种记法是相反的。在 Solidity 中，`X[3]` 总是一个包含三个 `X` 类型元素的数组，即使 `X` 本身是一个数组。这在其他语言中是不存在的，如 C 语言。

索引是基于零的，访问方向与声明相反。

例如，如果您有一个变量 `uint[][5] memory x`，您用 `x[2][6]` 访问第三个动态数组中的第七个 `uint`，要访问第三个动态数组，用 `x[2]`。同样，如果您有一个数组 `T[5] a` 的类型 `T`，也可以是一个数组，那么 `a[2]` 总是有类型 `T`。

数组元素可以是任何类型，包括映射或结构体。并适用于类型的一般限制，映射只能存储在 `storage` 数据位置，公开可见的函数需要参数是*ABI 类型*。

可以将状态变量数组标记为 `public`，并让 Solidity 创建一个*getter* 函数。数字索引成为该函数的一个必要参数。

访问一个超过它的末端的数组会导致一个失败的断言。方法 `.push()` 和 `.push(value)` 可以用来在动态大小的数组末端追加一个新的元素，其中 `.push()` 追加一个零初始化的元素并返回它的引用。

备注： 动态大小的数组只能在存储中调整大小。在内存中，这样的数组可以是任意大小的，但是一旦分配了数组，就不能改变数组的大小。

`bytes` 和 `string` 类型的数组

`bytes` 和 `string` 类型的变量是特殊的数组。`bytes` 类似于 `bytes1[]`，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，但不允许用长度或索引来访问。

Solidity 没有字符串操作函数，但有第三方的字符串库。您也可以用 `keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2))` 来比较两个字符串的 keccak256-hash，用 `string.concat(s1, s2)` 来连接两个字符串。

您应该使用 `bytes` 而不是 `bytes1[]`，因为它更便宜，因为在 `memory` 中使用 `bytes1[]` 会在元素之间增加 31 个填充字节。请注意，在 `storage` 中，由于紧打包，没有填充，参见字节和字符串。一般来说，对于

任意长度的原始字节数据使用 `bytes`，对于任意长度的字符串（UTF-8）数据使用 `string`。如果您能将长度限制在一定的字节数，总是使用 `bytes1` 到 `bytes32` 中的一种值类型，因为它们更便宜。

备注： 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';`。注意这时您访问的是 UTF-8 形式的低级 bytes 类型，而不是单个的字符。

函数 `bytes.concat` 和 `string.concat`

您可以使用 `string.concat` 连接任意数量的 `string` 值。该函数返回一个单一的 `string memory` 数组，其中包含没有填充的参数内容。如果您想使用不能隐式转换为 `string` 的其他类型的参数，您需要先将它们转换为 `string`。

同样，`bytes.concat` 函数可以连接任意数量的 `bytes` 或 `bytes1 ... bytes32` 值。该函数返回一个单一的 `bytes memory` 数组，其中包含没有填充的参数内容。如果您想使用字符串参数或其他不能隐式转换为 `bytes` 的类型，您需要先将它们转换为 `bytes` 或 `bytes1` /…/ `bytes32`。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

contract C {
    string s = "Storage";
    function f(bytes calldata bc, string memory sm, bytes16 b) public view {
        string memory concatString = string.concat(s, string(bc), "Literal", sm);
        assert((bytes(s).length + bc.length + 7 + bytes(sm).length) ==
↪bytes(concatString).length);

        bytes memory concatBytes = bytes.concat(bytes(s), bc, bc[:2], "Literal",
↪bytes(sm), b);
        assert((bytes(s).length + bc.length + 2 + 7 + bytes(sm).length + b.length) ==
↪concatBytes.length);
    }
}
```

如果您不带参数调用 `string.concat` 或 `bytes.concat`，它们会返回一个空数组。

创建内存数组

具有动态长度的内存数组可以使用 new 操作符创建。与存储数组不同的是，**不可能**调整内存数组的大小（例如，.push 成员函数不可用）。您必须事先计算出所需的大小，或者创建一个新的内存数组并复制每个元素。

正如 Solidity 中的所有变量一样，新分配的数组元素总是以默认值 进行初始化。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

数组字面常数

数组字面常数表达式是一个逗号分隔的一个或多个表达式的列表，用方括号（[...]）括起来。例如，[1, a, f(3)]。数组字面常数的类型确定如下：

它总是一个静态大小的内存数组，其长度是表达式的数量。

数组的基本类型是列表上第一个表达式的类型，这样所有其他表达式都可以隐含地转换为它。如果不能做到这一点，则会有一个类型错误。

仅仅存在一个所有元素都可以转换的类型是不够的。其中一个元素必须是该类型的。

在下面的例子中，[1, 2, 3] 的类型是 uint8[3] memory，因为这些常量的类型都是 uint8。如果您想让结果是 uint[3] memory 类型，您需要把第一个元素转换为 uint。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

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

数组表达式 [1, -1] 是无效的，因为第一个表达式的类型是 uint8，而第二个表达式的类型是 int8，它们不能相互隐式转换。为了使其有效，例如，您可以使用 [int8(1), -1]。

由于不同类型的固定大小的内存数组不能相互转换（即使基类可以），如果您想使用二维数组字面常数，您必须总是明确指定一个共同的基类：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure returns (uint24[2][4] memory) {
        uint24[2][4] memory x = [[uint24(0x1), 1], [0xffffff, 2], [uint24(0xff), 3],
↪[uint24(0xffff), 4]];
        // 下面的方法不会起作用，因为一些内部数组的类型不对。
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2], [0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

固定大小的内存数组不能分配给动态大小的内存数组，也就是说，以下情况是不可能的：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

// 这不会被编译。
contract C {
    function f() public {
        // 下一行会产生一个类型错误，因为uint[3]内存不能被转换为uint[]内存。
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

计划在将来取消这一限制，但由于 ABI 中数组的传递方式，它产生了一些复杂的问题。

如果您想初始化动态大小的数组，您必须分配各个元素：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f() public pure {
        uint[] memory x = new uint[](3);
        x[0] = 1;
        x[1] = 3;
        x[2] = 4;
    }
```

(续下页)

(接上页)

```
}
```

数组成员

length:

数组有 `length` 成员变量表示当前数组的长度。一经创建，内存 memory 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():

动态存储数组和 `bytes` （不是 `string` ）有一个叫 `push()` 的成员函数，您可以用它在数组的末尾追加一个零初始化的元素。它返回一个元素的引用，因此可以像 `x.push().t = 2` 或 `x.push() = b` 那样使用。

push(x):

动态存储数组和 `bytes` （不是 `string` ）有一个叫 `push(x)` 的成员函数，您可以用它在数组的末端追加一个指定的元素。该函数不返回任何东西。

pop():

动态存储数组和 `bytes` （不是 `string` ）有一个叫 `pop()` 的成员函数，您可以用它来从数组的末端移除一个元素。这也隐含地在被删除的元素上调用*delete*。该函数不返回任何东西。

备注： 通过调用 `push()` 增加存储数组的长度有恒定的气体成本，因为存储是零初始化的，而通过调用 `pop()` 减少长度的成本取决于被移除元素的" 大小"。如果该元素是一个数组，它的成本可能非常高，因为它包括明确地清除被移除的元素，类似于对它们调用*delete*。

备注： 要在外部（而不是公开）函数中使用数组的数组，您需要激活 ABI coder v2。

备注： 在 Byzantium 之前的 EVM 版本中，不可能访问从函数调用返回的动态数组。如果您调用返回动态数组的函数，请确保使用设置为 Byzantium 模式的 EVM。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract ArrayContract {
    uint[2**20] aLotOfIntegers;
    // 请注意，下面不是一对动态数组，
    // 而是一个动态数组对（即长度为2的固定大小数组）。
    // 在 Solidity 中，T[k]和T[]总是具有T类型元素的数组，
```

(续下页)

(接上页)

```
    // 即使T本身是一个数组。
    // 正因为如此，bool[2][]是一个动态数组对，其元素是bool[2]。
    // 这与其他语言不同，比如C，
    // 所有状态变量的数据位置都是存储。
    bool[2][] pairsOfFlags;

    // newPairs被存储在memory中--这是公开合约函数参数的唯一可能性。
    function setAllFlagPairs(bool[2][] memory newPairs) public {
        // 赋值到一个存储数组会执行 ``newPairs`` 的拷贝，
        // 并替换完整的数组 ``pairsOfFlags``。
        pairsOfFlags = newPairs;
    }

    struct StructType {
        uint[] contents;
        uint moreInfo;
    }
    StructType s;

    function f(uint[] memory c) public {
        // 在 ``g`` 中存储一个对 ``s`` 的引用。
        StructType storage g = s;
        // 也改变了 ``s.moreInfo``.
        g.moreInfo = 2;
        // 指定一个拷贝，因为 ``g.contents`` 不是一个局部变量，
        // 而是一个局部变量的成员。
        g.contents = c;
    }

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

    function changeFlagArraySize(uint newSize) public {
        // 使用push和pop是改变数组长度的唯一方法。
        if (newSize < pairsOfFlags.length) {
            while (pairsOfFlags.length > newSize)
                pairsOfFlags.pop();
        } else if (newSize > pairsOfFlags.length) {
            while (pairsOfFlags.length < newSize)
                pairsOfFlags.push();
```

(续下页)

(接上页)

```
        }
    }

    function clear() public {
        // 这些完全清除了数组
        delete pairsOfFlags;
        delete aLotOfIntegers;
        // 这里有相同的效果
        pairsOfFlags = new bool[2][](0);
    }

    bytes byteData;

    function byteArrays(bytes memory data) public {
        // 字节数组 ("byte") 是不同的, 因为它们的存储没有填充,
        // 但可以与 "uint8[]"相同。
        byteData = data;
        for (uint i = 0; i < 7; i++)
            byteData.push();
        byteData[3] = 0x08;
        delete byteData[2];
    }

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

    function createMemoryArray(uint size) public pure returns (bytes memory) {
        // 使用 `new` 创建动态 memory 数组:
        uint[2][] memory arrayOfPairs = new uint[2][](size);

        //␣
↪内联数组总是静态大小的, 如果您只使用字面常数表达式, 您必须至少提供一种类型。
        arrayOfPairs[0] = [uint(1), 2];

        // 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] = bytes1(uint8(i));
        return b;
    }
}
```

对存储数组元素的悬空引用（Dangling References）

当使用存储数组时，您需要注意避免悬空引用。悬空引用是指一个指向不再存在的或已经被移动而未更新引用的内容的引用。例如，如果您将一个数组元素的引用存储在一个局部变量中，然后从包含数组中使用 .pop()，就可能发生悬空引用：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[][] s;

    function f() public {
        // 存储一个指向s的最后一个数组元素的指针。
        uint[] storage ptr = s[s.length - 1];
        // 删除s的最后一个数组元素。
        s.pop();
        // 写入已不在数组内的数组元素。
        ptr.push(0x42);
        // 现在向 ``s`` 添加一个新元素不会添加一个空数组，
        // 而是会产生一个长度为1的数组， 元素为 ``0x42``。
        s.push();
        assert(s[s.length - 1][0] == 0x42);
    }
}
```

ptr.push(0x42) 中的写法 **不会**恢复操作，尽管 ptr 不再指向 s 的一个有效元素。由于编译器假定未使用的存储空间总是被清零，随后的 s.push() 不会明确地将零写入存储空间，所以在 push() 之后，s 的最后一个元素的长度是 1，并且包含 0x42 作为其第一个元素。

注意，Solidity 不允许在存储中声明对值类型的引用。这类显式的悬空引用被限制在嵌套引用类型中。然而，当在数组赋值中使用复杂表达式时，悬空引用也会短暂发生：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

contract C {
    uint[] s;
    uint[] t;
    constructor() {
        // 向存储数组推送一些初始值。
        s.push(0x07);
        t.push(0x03);
    }
```

(续下页)

(接上页)

```
    function g() internal returns (uint[] storage) {
        s.pop();
        return t;
    }

    function f() public returns (uint[] memory) {
        // 下面将首先评估 ``s.push()` 到一个索引为1的新元素的引用。
        // 之后，调用 ``g`` 弹出这个新元素，
        // 导致最左边的元组元素成为一个悬空的引用。
        // 赋值仍然发生，并将写入 ``s`` 的数据区域之外。
        (s.push(), g()[0]) = (0x42, 0x17);
        // 随后对 ``s`` 的推送将显示前一个语句写入的值，
        // 即在这个函数结束时 ``s`` 的最后一个元素将有 ``0x42`` 的值。
        s.push();
        return s;
    }
}
```

每条语句只对存储进行一次赋值，并避免在赋值的左侧使用复杂的表达式，这样做总是比较安全的。

您需要特别小心处理对 `bytes` 数组元素的引用，因为 bytes 数组的 `.push()` 操作可能会*在存储中从短布局切换到长布局*。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0 <0.9.0;

// 这将报告一个警告
contract C {
    bytes x = "012345678901234567890123456789";

    function test() external returns(uint) {
        (x.push(), x.push()) = (0x01, 0x02);
        return x.length;
    }
}
```

这里，当第一个 `x.push()` 被运算时，`x` 仍然被存储在短布局中，因此 `x.push()` 返回对 `x` 的第一个存储槽中元素的引用。然而，第二个 `x.push()` 将字节数组切换为长布局。现在 `x.push()` 所指的元素在数组的数据区，而引用仍然指向它原来的位置，现在它是长度字段的一部分，赋值将有效地扰乱 `x` 的长度。为了安全起见，在一次赋值中最多只放大字节数组中的一个元素，不要在同一语句中同时对数组进行索引存取。

虽然上面描述了当前版本的编译器中悬空存储引用的行为，但任何带有悬空引用的代码都应被视为具有 未定义行为。特别的是，这意味着任何未来版本的编译器都可能改变涉及悬空引用的代码的行为。

请确保避免在您的代码中出现悬空引用。

数组切片

数组切片是对一个数组的连续部分的预览。它们被写成 `x[start:end]`，其中 `start` 和 `end` 是表达式，结果是 uint256 类型（或隐含的可转换类型）。分片的第一个元素是 `x[start]`，最后一个元素是 `x[end - 1]`。

如果 `start` 大于 `end`，或者 `end` 大于数组的长度，就会出现异常。

`start` 和 `end` 都是可选的：`start` 默认为 `0`，`end` 默认为数组的长度。

数组切片没有任何成员。它们可以隐含地转换为其底层类型的数组并支持索引访问。索引访问在底层数组中不是绝对的，而是相对于分片的开始。

数组切片没有类型名，这意味着任何变量都不能以数组切片为类型，它们只存在于中间表达式中。

备注： 到现在为止，数组切片只有 calldata 数组可以实现。

数组切片对于 ABI 解码在函数参数中传递的二级数据很有用：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.5 <0.9.0;
contract Proxy {
    /// @dev 由代理管理的客户合约的地址，即本合约的地址
    address client;

    constructor(address client_) {
        client = client_;
    }

    /// 转发对 "setOwner(address)" 的调用，
    /// 该调用在对地址参数进行基本验证后由客户端执行。
    function forward(bytes calldata payload) external {
        bytes4 sig = bytes4(payload[:4]);
        // 由于截断行为，bytes4(payload)的表现是相同的。
        // bytes4 sig = bytes4(payload);
        if (sig == bytes4(keccak256("setOwner(address)"))) {
            address owner = abi.decode(payload[4:], (address));
            require(owner != address(0), "Address of owner cannot be zero.");
        }
        (bool status,) = client.delegatecall(payload);
        require(status, "Forwarded call failed.");
    }
}
```

结构体

Solidity 提供了一种以结构形式定义新类型的方法，以下是一个结构体使用的示例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// 定义一个包含两个属性的新类型。
// 在合约之外声明一个结构，
// 可以让它被多个合约所共享。
// 在这里，这并不是真的需要。
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
    // 结构体也可以被定义在合约内部，这使得它们只在本合约和派生合约中可见。
    struct Campaign {
        address payable beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping(uint => Funder) funders;
    }

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

    function newCampaign(address payable beneficiary, uint goal) public returns (uint␣
↪campaignID) {
        campaignID = numCampaigns++; // campaignID 作为一个变量返回
        // 我们不能使用 "campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)"
        // 因为右侧创建了一个内存结构 "Campaign"，其中包含一个映射。
        Campaign storage c = campaigns[campaignID];
        c.beneficiary = beneficiary;
        c.fundingGoal = goal;
    }

    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。
        // 注意您也可以使用 Funder(msg.sender, msg.value) 来初始化。
        c.funders[c.numFunders++] = Funder({addr: msg.sender, amount: msg.value});
```

(续下页)

(接上页)

```
        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;
    }
}
```

上面的合约并没有提供众筹合约的全部功能，但它包含了理解结构体所需的基本概念。结构类型可以在映射和数组内使用，它们本身可以包含映射和数组。

结构体不可能包含其自身类型的成员，尽管结构本身可以是映射成员的值类型，或者它可以包含其类型的动态大小的数组。这一限制是必要的，因为结构的大小必须是有限的。

注意在所有的函数中，结构类型被分配到数据位置为 `storage` 的局部变量。这并不是拷贝结构体，而只是存储一个引用，因此对本地变量成员的赋值实际上是写入状态。

当然，您也可以直接访问该结构的成员，而不把它分配给本地变量，如 `campaigns[campaignID].amount = 0`。

备注： 在 Solidity 0.7.0 之前，包含仅有存储类型（例如映射）的成员的内存结构是允许的，像上面例子中的 `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` 这样的赋值是可以的，只是会默默地跳过这些成员。

3.6.3 映射类型

映射类型使用语法 `mapping(KeyType KeyName? => ValueType ValueName?)`，映射类型的变量使用语法 `mapping(KeyType KeyName? => ValueType ValueName?) VariableName` 声明。`KeyType` 可以是任何内置的值类型，`bytes`，`string`，或任何合约或枚举类型。其他用户定义的或复杂的类型，如映射，结构体或数组类型是不允许的。`ValueType` 可以是任何类型，包括映射，数组和结构体。`KeyName` 和 `ValueName` 是可选的（所以 `mapping(KeyType => ValueType)` 也可以使用），可以是任何有效的标识符，而不是一个类型。

您可以把映射想象成 哈希表，它实际上被初始化了，使每一个可能的键都存在，并将其映射到字节形式全是零的值，一个类型的默认值。相似性到此为止，键数据不存储在映射中，而是它的 `keccak256` 哈希值被用来查询。

正因为如此，映射没有长度，也没有被设置的键或值的概念，因此，如果没有关于分配的键的额外信息，就不能被删除（见清除映射）。

映射只能有一个 storage 的数据位置，因此允许用于状态变量，可作为函数中的存储引用类型，或作为库函数的参数。但它们不能被用作公开可见的合约函数的参数或返回参数。这些限制对于包含映射的数组和结构也是如此。

您可以把映射类型的状态变量标记为 public，Solidity 会为您创建一个*getter* 函数。KeyType 成为 getter 函数的参数，名称为 KeyName（如果指定）。如果 ValueType 是一个值类型或一个结构，getter 返回 ValueType，名称为 ValueName（如果指定）。如果 ValueType 是一个数组或映射，getter 对每个 KeyType 递归出一个参数。

在下面的例子中，MappingExample 合约定义了一个公共的 balances 映射，键类型是 address，值类型是 uint，将一个 Ethereum 地址映射到一个无符号整数值。由于 uint 是一个值类型，getter 返回一个与该类型相匹配的值，您可以在 MappingUser 合约中看到它返回指定地址对应的值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

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(address(this));
    }
}
```

下面的例子是一个简化版本的 ERC20 代币。_allowances 是一个映射类型在另一个映射类型中的例子。

在下面的例子中，为映射提供了可选的 KeyName 和 ValueName。它不影响任何合约的功能或字节码，它只是为映射的 getter 在 ABI 中设置输入和输出的 name 字段。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.18;

contract MappingExampleWithNames {
    mapping(address user => uint balance) public balances;
```

（续下页）

(接上页)

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

下面的例子使用 _allowances 来记录其他人可以从你的账户中提取的金额。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract MappingExample {

    mapping(address => uint256) private _balances;
    mapping(address => mapping(address => uint256)) private _allowances;

    event Transfer(address indexed from, address indexed to, uint256 value);
    event Approval(address indexed owner, address indexed spender, uint256 value);

    function allowance(address owner, address spender) public view returns (uint256) {
        return _allowances[owner][spender];
    }

    function transferFrom(address sender, address recipient, uint256 amount) public␣
↪returns (bool) {
        require(_allowances[sender][msg.sender] >= amount, "ERC20: Allowance not high␣
↪enough.");
        _allowances[sender][msg.sender] -= amount;
        _transfer(sender, recipient, amount);
        return true;
    }

    function approve(address spender, uint256 amount) public returns (bool) {
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    function _transfer(address sender, address recipient, uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");
        require(_balances[sender] >= amount, "ERC20: Not enough funds.");
```

(续下页)

(接上页)

```
        _balances[sender] -= amount;
        _balances[recipient] += amount;
        emit Transfer(sender, recipient, amount);
    }
}
```

递归映射

您不能对映射进行递归调用，也就是说，您不能列举它们的键。不过，可以在它们上层实现一个数据结构，并对其进行递归。例如，下面的代码实现了一个 IterableMapping 库，然后 User 合约将数据添加到该库中，sum 函数对所有的值进行递归调用去累加这些值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

struct IndexValue { uint keyIndex; uint value; }
struct KeyFlag { uint key; bool deleted; }

struct itmap {
    mapping(uint => IndexValue) data;
    KeyFlag[] keys;
    uint size;
}

type Iterator is uint;

library IterableMapping {
    function insert(itmap storage self, uint key, uint value) internal returns (bool␣
↪replaced) {
        uint keyIndex = self.data[key].keyIndex;
        self.data[key].value = value;
        if (keyIndex > 0)
            return true;
        else {
            keyIndex = self.keys.length;
            self.keys.push();
            self.data[key].keyIndex = keyIndex + 1;
            self.keys[keyIndex].key = key;
            self.size++;
            return false;
        }
    }
```

(续下页)

(接上页)

```
    function remove(itmap storage self, uint key) internal returns (bool success) {
        uint keyIndex = self.data[key].keyIndex;
        if (keyIndex == 0)
            return false;
        delete self.data[key];
        self.keys[keyIndex - 1].deleted = true;
        self.size --;
    }

    function contains(itmap storage self, uint key) internal view returns (bool) {
        return self.data[key].keyIndex > 0;
    }

    function iterateStart(itmap storage self) internal view returns (Iterator) {
        return iteratorSkipDeleted(self, 0);
    }

    function iterateValid(itmap storage self, Iterator iterator) internal view␣
↪returns (bool) {
        return Iterator.unwrap(iterator) < self.keys.length;
    }

    function iterateNext(itmap storage self, Iterator iterator) internal view returns␣
↪(Iterator) {
        return iteratorSkipDeleted(self, Iterator.unwrap(iterator) + 1);
    }

    function iterateGet(itmap storage self, Iterator iterator) internal view returns␣
↪(uint key, uint value) {
        uint keyIndex = Iterator.unwrap(iterator);
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }

    function iteratorSkipDeleted(itmap storage self, uint keyIndex) private view␣
↪returns (Iterator) {
        while (keyIndex < self.keys.length && self.keys[keyIndex].deleted)
            keyIndex++;
        return Iterator.wrap(keyIndex);
    }
}
```

(续下页)

(接上页)

```
// 如何使用
contract User {
    // 只是一个保存我们数据的结构体。
    itmap data;
    // 对数据类型应用库函数。
    using IterableMapping for itmap;

    // 插入一些数据
    function insert(uint k, uint v) public returns (uint size) {
        // 这将调用 IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // 我们仍然可以访问结构中的成员，
        // 但我们应该注意不要乱动他们。
        return data.size;
    }

    // 计算所有存储数据的总和。
    function sum() public view returns (uint s) {
        for (
            Iterator i = data.iterateStart();
            data.iterateValid(i);
            i = data.iterateNext(i)
        ) {
            (, uint value) = data.iterateGet(i);
            s += value;
        }
    }
}
```

3.6.4 运算符

即使两个操作数的类型不一样，也可以应用算术和位操作数。例如，您可以计算 `y = x + z`，其中 `x` 是 `uint8`，`z` 的类型为 `int32`。在这种情况下，下面的机制将被用来确定计算操作的类型（这在溢出的情况下很重要）和操作结果的类型：

1. 如果右操作数的类型可以隐式转换为左操作数的类型，则使用左操作数的类型，
2. 如果左操作数的类型可以隐式转换为右操作数的类型，则使用右操作数的类型，
3. 否则的话，该操作不被允许。

如果其中一个操作数是字面常数，它首先被转换为其"移动类型（mobile type）"，也就是能容纳该值的最小类型（相同位宽的无符号类型被认为比有符号类型"小"）。如果两者都是字面常数，那么运算的精度实际上是无限的，因为表达式被转换到任何必要的精度，所以当结果被用于非字面类型时，没有任何损失。

操作符的结果类型与操作的类型相同，除了比较操作符，其结果总是 bool。

运算符 ** （幂运算），<< 和 >> 使用左边操作数的类型进行运算和以其作为结果。

三元运算符

三元运算符用于形式为 < 条件表达式> ? <true 条件表达式> : <false 条件表达式>。它根据主要的 < 条件表达式> 的评估结果，计算后两个给定表达式中的一个。如果 < 条件表达式> 评估为 true，那么 <true 条件表达式> 将被计算，否则 <false 条件表达式> 被被计算。

三元运算符的结果没有有理数类型，即使它的操作数都是有理数字。结果类型是由两个操作数的类型决定的，方法同上，如果需要的话，首先转换为它们的可移动计算的类型。

因此，255 + (true ? 1 : 0) 将由于算术溢出而恢复计算。原因是 (true ? 1 : 0) 是 uint8 类型，这迫使加法也在 uint8 中进行，而 256 超出了这个类型允许的范围。

另一个结果是，像 1.5 + 1.5 这样的表达式是有效的，但 1.5 + (true ? 1.5 : 2.5) 却无效。这是因为前者是一个以无限精度计算的有理表达式，只有它的最终值才是重要的。后者涉及到将小数有理数转换为整数，这在目前是不允许的。

复数和增量/减量运算符

如果 a 是一个 LValue（即是一个变量或者是可以被分配的东西），下列运算符可以作为速记：

a += e 相当于 a = a + e，运算符 -=，*=，/=，%=，|=，&=，^=，<<= 和 >>= 都有相应的定义。a++ 和 a-- 相当于 a += 1 / a -= 1 但是表达式本身仍然是以前的值 a。相比之下，--a 和 ++a 对 a 有同样的作用，但返回改变后的值。

删除

delete a 为该类型分配初始值 a。例如，对于整数来说，它相当于 a = 0，但是它也可以用于数组，它指定一个长度为 0 的动态数组或者一个相同长度的静态数组，所有元素都设置为初始值。delete a[x] 删除数组中索引为 x 的元素，并保留所有其他元素和数组的长度不动。这特别意味着它在数组中留下一个缺口。如果您打算删除项目，一个映射类型 可能是一个更好的选择。

对于结构体，则将结构体中的所有属性重置。换句话说，在 delete a 之后，a 的值与 a 在没有赋值的情况下被声明是一样的，但有以下注意事项：

delete 对映射类型没有影响（因为映射的键可能是任意的，通常是未知的）。因此，如果您删除一个结构体，它将重置所有不是映射类型的成员，同时也会递归到这些成员，除非它们是映射。然而，单个键和它们所映射的内容可以被删除。如果 a 是一个映射，那么 delete a[x] 将删除存储在 x 的值。

值得注意的是，delete a 的行为实际上是对 a 的赋值，也就是说，它在 a 中存储了一个新的对象。当 a 是引用变量时，这种区别是明显的。它只会重置 a 本身，而不是它之前引用的值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // 将 x 设为 0，并不影响data变量
        delete data; // 将 data 设为 0，并不影响 x
        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，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。
        assert(y.length == 0);
    }
}
```

运算符的优先顺序

以下是按评估顺序列出的操作符优先级。

<table>
<tr><th>优先级</th><th>描述</th><th>操作符</th></tr>
<tr><td>1</td><td>后置自增和自减</td><td>++, --</td></tr>
<tr><td></td><td>创建类型实例</td><td>new < 类型名></td></tr>
<tr><td></td><td>数组元素</td><td>< 数组>[< 索引>]</td></tr>
<tr><td></td><td>访问成员</td><td>< 对象>.< 成员名></td></tr>
<tr><td></td><td>函数调用</td><td>< 函数>(< 参数...>)</td></tr>
<tr><td></td><td>小括号</td><td>(< 表达式>)</td></tr>
<tr><td>2</td><td>前置自增和自减</td><td>++, --</td></tr>
<tr><td></td><td>一元运算减</td><td>-</td></tr>
<tr><td></td><td>一元操作符</td><td>delete</td></tr>
<tr><td></td><td>逻辑非</td><td>!</td></tr>
<tr><td></td><td>按位非</td><td>~</td></tr>
<tr><td>3</td><td>乘方</td><td>**</td></tr>
<tr><td>4</td><td>乘、除和模运算</td><td>*, /, %</td></tr>
<tr><td>5</td><td>算术加和减</td><td>+, -</td></tr>
<tr><td>6</td><td>移位操作符</td><td><<, >></td></tr>
<tr><td>7</td><td>按位与</td><td>&</td></tr>
<tr><td>8</td><td>按位异或</td><td>^</td></tr>
<tr><td>9</td><td>按位或</td><td>|</td></tr>
<tr><td>10</td><td>非等操作符</td><td><, >, <=, >=</td></tr>
<tr><td>11</td><td>等于操作符</td><td>==, !=</td></tr>
<tr><td>12</td><td>逻辑与</td><td>&&</td></tr>
<tr><td>13</td><td>逻辑或</td><td>==</td></tr>
<tr><td>14</td><td>三元操作符</td><td>< 判断条件> ? < 如果为真时执行的表达式> : < 如果为假时执行的表达式></td></tr>
<tr><td></td><td>赋值操作符</td><td>=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=</td></tr>
<tr><td>15</td><td>逗号</td><td>,</td></tr>
</table>

3.6.5 基本类型之间的转换

隐式转换

在某些情况下，在赋值过程中，在向函数传递参数和应用运算符时，编译器会自动应用隐式类型转换。一般来说，如果在语义上有意义，并且不会丢失信息，那么值-类型之间的隐式转换是可能的。

例如，`uint8` 可以转换为 `uint16`，`int128` 可以转换为 `int256`，但是 `int8` 不能转换为 `uint256`，因为 `uint256` 不能容纳 `-1` 这样的值。

如果一个运算符被应用于不同的类型，编译器会尝试将其中一个操作数隐含地转换为另一个的类型（对于赋值也是如此）。这意味着操作总是以其中一个操作数的类型进行。

关于哪些隐式转换是可能的，请参考关于类型本身的章节。

在下面的例子中，y 和 z，即加法的操作数，没有相同的类型，但是 uint8 可以隐式转换为 uint16，反之则不行。正因为如此，y 被转换为 z 的类型，然后在 uint16 类型中进行加法。结果表达式 y + z 的类型是 uint16。因为它被分配到一个 uint32 类型的变量中，所以在加法后又进行了一次隐式转换。

```
uint8 y;
uint16 z;
uint32 x = y + z;
```

显式转换

如果编译器不允许隐式转换，但您确信转换会成功，有时可以进行显式类型转换。这可能会导致意想不到的行为，并使您绕过编译器的一些安全特性，所以一定要测试结果是否是您想要的和期望的！

以下面的例子为例，将一个负的 int 转换为 uint：

```
int  y = -3;
uint x = uint(y);
```

在这个代码片断的最后，x 变成 0xfffff..fd 的值（64 个十六进制字符），这在 256 位的二进制补码中表示是-3。

如果一个整数被明确地转换为一个较小的类型，高阶位就会被切断：

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b 现在会是 0x5678
```

如果一个整数被明确地转换为一个更大的类型，它将在左边被填充（即在高阶的一端）。转换的结果将与原整数比较相等：

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b 现在会是 0x00001234
assert(a == b);
```

固定大小的字节类型在转换过程中的行为是不同的。它们可以被认为是单个字节的序列，转换到一个较小的类型将切断序列：

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b 现在会是 0x12
```

如果一个固定大小的字节类型被明确地转换为一个更大的类型，它将在右边被填充。访问固定索引的字节将导致转换前后的数值相同（如果索引仍在范围内）：

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b 现在会是 0x12340000
assert(a[0] == b[0]);
assert(a[1] == b[1]);
```

于整数和固定大小的字节数组在截断或填充时表现不同，只有在整数和固定大小的字节数组具有相同大小的情况下，才允许在两者之间进行显式转换。如果您想在不同大小的整数和固定大小的字节数组之间进行转换，您必须使用中间转换，使所需的截断和填充规则明确：

```
bytes2 a = 0x1234;
uint32 b = uint16(a); // b 将会是 0x00001234
uint32 c = uint32(bytes4(a)); // c 将会是 0x12340000
uint8 d = uint8(uint16(a)); // d 将会是 0x34
uint8 e = uint8(bytes1(a)); // e 将会是 0x12
```

`bytes` 数组和 `bytes` calldata 切片可以明确转换为固定字节类型（`bytes1` /…/ `bytes32`）。如果数组比目标的固定字节类型长，在末端会发生截断的情况。如果数组比目标类型短，它将在末尾被填充零。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.5;

contract C {
    bytes s = "abcdefgh";
    function f(bytes calldata c, bytes memory m) public view returns (bytes16,␣
↪bytes3) {
        require(c.length == 16, "");
        bytes16 b = bytes16(m);  // 如果m的长度大于16, 将发生截断。
        b = bytes16(s);  // 右边进行填充, 所以结果是 "abcdefgh\0\0\0\0\0\0\0\0"
        bytes3 b1 = bytes3(s); // 发生截断, b1 相当于 "abc"
        b = bytes16(c[:8]);  // 同样用0进行填充
        return (b, b1);
    }
}
```

3.6.6 字面常数和基本类型之间的转换

整数类型

十进制和十六进制的数字字面常数可以隐含地转换为任何足够大的整数类型去表示它而不被截断：

```
uint8 a = 12; // 可行
uint32 b = 1234; // 可行
uint16 c = 0x123456; // 报错, 因为这将会截断成 0x3456
```

备注： 在 0.8.0 版本之前，任何十进制或十六进制的数字字面常数都可以显式转换为整数类型。从 0.8.0 开始，这种显式转换和隐式转换一样严格，也就是说，只有当字面意义符合所产生的范围时，才允许转换。

固定大小的字节数组

十进制数字字面常数不能被隐含地转换为固定大小的字节数组。十六进制数字字面常数是可以的，但只有当十六进制数字的数量正好符合字节类型的大小时才可以。但是有一个例外，数值为 0 的十进制和十六进制数字字面常数都可以被转换为任何固定大小的字节类型：

```
bytes2 a = 54321; // 不允许
bytes2 b = 0x12; // 不允许
bytes2 c = 0x123; // 不允许
bytes2 d = 0x1234; // 可行
bytes2 e = 0x0012; // 可行
bytes4 f = 0; // 可行
bytes4 g = 0x0; // 可行
```

字符串和十六进制字符串字面常数可以被隐含地转换为固定大小的字节数组，如果它们的字符数与字节类型的大小相匹配：

```
bytes2 a = hex"1234"; // 可行
bytes2 b = "xy"; // 可行
bytes2 c = hex"12"; // 不允许
bytes2 d = hex"123"; // 不允许
bytes2 e = "x"; // 不允许
bytes2 f = "xyz"; // 不允许
```

地址类型

正如在地址字面常数（*Address Literals*）中所描述的那样，正确大小并通过校验测试的十六进制字是 `address` 类型。其他字面常数不能隐含地转换为 `address` 类型。

只允许从 `bytes20` 和 `uint160` 显式转换到 `address`。

`address a` 可以通过 `payable(a)` 显式转换为 `address payable`。

备注： 在 0.8.0 版本之前，可以显式地从任何整数类型（任何大小，有符号或无符号）转换为 `address` 或 `address payable` 类型。从 0.8.0 开始，只允许从 `uint160` 转换。

3.7 单位和全局可用变量

3.7.1 以太坊（Ether）单位

一个字面常数可以带一个后缀 wei，gwei 或 ether 来指定一个以太坊的数量，其中没有后缀的以太数字被认为单位是 wei。

```
assert(1 wei == 1);
assert(1 gwei == 1e9);
assert(1 ether == 1e18);
```

单位后缀的唯一作用是乘以 10 的幂次方。

备注： 0.7.0 版本中删除了 finney 和 szabo 这两个单位。

3.7.2 时间单位

诸如 seconds，minutes，hours，days 和 weeks 等后缀在字面常数后面，可以用来指定时间单位，其中秒是基本单位，单位的考虑方式很直白：

- 1 == 1 seconds
- 1 minutes == 60 seconds
- 1 hours == 60 minutes
- 1 days == 24 hours
- 1 weeks == 7 days

如果您使用这些单位进行日历计算，请注意，由于 闰秒 会造成不是每一年都等于 365 天，甚至不是每一天都有 24 小时，而且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。

备注： 由于上述原因，在 0.5.0 版本中删除了后缀 years。

这些后缀单位不能应用于变量。例如，如果您想用时间单位（例如 days）来将输入变量换算为时间，您可以用以下方式：

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

(续下页)

(接上页)

```
    }
}
```

3.7.3 特殊变量和函数

有一些特殊的变量和函数总是存在于全局命名空间，主要用于提供区块链的信息，或者是通用的工具函数。

区块和交易属性

- `blockhash(uint blockNumber) returns (bytes32)`: 当 `blocknumber` 是最近的 256 个区块之一时，给定区块的哈希值；否则返回 0。
- `block.basefee (uint)`: 当前区块的基本费用（EIP-3198 和 EIP-1559）
- `block.chainid (uint)`: 当前链的 ID
- `block.coinbase (address payable)`: 挖出当前区块的矿工地址
- `block.difficulty (uint)`：当前块的难度（`EVM < Paris`）。对于其他 EVM 版本，它是为 `block.prevrandao` 的已废弃别名（EIP-4399）
- `block.gaslimit (uint)`: 当前区块 gas 限额
- `block.number (uint)`: 当前区块号
- `block.timestamp (uint)`: 自 unix epoch 起始到当前区块以秒计的时间戳
- `gasleft() returns (uint256)`：剩余的 gas
- `msg.data (bytes calldata)`: 完整的 calldata
- `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
- `tx.gasprice (uint)`: 随消息发送的 wei 的数量
- `tx.origin (address)`: 交易发起者（完全的调用链）

备注： 对于每一个 **外部（external）** 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。

备注： 当合约在链下而不是在区块中包含的交易的背景下计算时，您不应该认为 `block.*` 和 `tx.*` 是指任何特定区块或交易的值。这些值是由执行合约的 EVM 实现提供的，可以是任意的。

备注： 不要依赖 `block.timestamp` 和 `blockhash` 产生随机数，除非您知道自己在做什么。

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。

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

备注： 函数 `blockhash` 以前被称为 `block.blockhash`，在 0.4.22 版本中被废弃，在 0.5.0 版本中被删除。

备注： 函数 `gasleft` 的前身是 `msg.gas`，在 0.4.21 版本中被弃用，在 0.5.0 版本中被删除。

备注： 在 0.7.0 版本中，删除了别名 `now`` (用于 ``block.timestamp`)。

ABI 编码和解码函数

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: ABI-解码给定的数据，而类型在括号中作为第二个参数给出。例如：`(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`： 对给定的参数进行 ABI 编码
- `abi.encodePacked(...) returns (bytes memory)`：对给定参数执行紧打包编码。请注意，打包编码可能会有歧义！
- `abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory)`： 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 memory signature, ...) returns (bytes memory)`： 相当于 `abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`
- `abi.encodeCall(function functionPointer, (...)) returns (bytes memory)`： 对函数指针的调用进行 ABI 编码，参数在元组中找到。执行全面的类型检查，确保类型与函数签名相符。结果相当于 `abi.encodeWithSelector(functionPointer.selector, (...))`。

备注： 这些编码函数可用于制作外部函数调用的数据，而无需实际调用外部函数。此外，`keccak256(abi.encodePacked(a, b))` 是一种计算结构化数据的哈希值的方法（但是要注意有可能使用不同的函数参数类型会制作出一个" 哈希碰撞"）。

更多详情请参考*ABI* 和紧打包编码。

字节类型的成员

- `bytes.concat(...) returns (bytes memory)`: 将可变数量的字节和 *byte1, ..., byte32* 参数串联成一个字节数组

字符串的成员

- `string.concat(...) returns (string memory)`: 将可变数量的字符串参数串联成一个字符串数组

错误处理

关于错误处理和何时使用哪个函数的更多细节，请参见*assert* 和 *require* 的专门章节。

`assert(bool condition)`
如果条件不满足，会导致异常，因此，状态变化会被恢复 - 用于内部错误。

`require(bool condition)`
如果条件不满足，则恢复状态更改 - 用于输入或外部组件的错误。

`require(bool condition, string memory message)`
如果条件不满足，则恢复状态更改 - 用于输入或外部组件的错误，可以同时提供一个错误消息。

`revert()`
终止运行并恢复状态更改。

`revert(string memory reason)`
终止运行并恢复状态更改，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(bytes memory) returns (bytes32)
计算输入的 Keccak-256 哈希值。

备注： 以前 `keccak256` 的别名叫 `sha3`，在 0.5.0 版本中被删除。

---

sha256(bytes memory) returns (bytes32)
计算输入的 SHA-256 哈希值。

ripemd160(bytes memory) returns (bytes20)
计算输入的 RIPEMD-160 哈希值。

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。函数参数对应于签名的 ECDSA 值：

- `r` = 签名的前 32 字节
- `s` = 签名的第二个 32 字节
- `v` = 签名的最后 1 个字节

`ecrecover` 返回一个 `address`，而不是 `address payable`。参见地址类型 进行转换，以备您需要向恢复的地址转移资金。

更多细节，请阅读 使用示例.

警告： 如果您使用 `ecrecover`，请注意，一个有效的签名可以变成另一个有效的签名，而不需要知道相应的私钥。在 Homestead 硬分叉中，这个问题对 _transaction_ 签名进行了修复（见 EIP-2），但 ecrecover 函数仍然没有改变。

这通常不是一个问题，除非您要求签名是唯一的，或者用它们来识别个体。OpenZeppelin 有一个 ECDSA 辅助库，您可以用它作为 `ecrecover` 的包装，那样就没有这个问题。

备注： 当在 私有区块链上运行 `sha256`，`ripemd160` 或 `ecrecover` 时，您可能会遇到超出 gas（Out-of-Gas）的错误。这是因为这些功能是作为"预编译合约"实现的，只有在它们收到第一个消息后才真正存在（尽管它们的合约代码是硬编码的）。向不存在的合约发送消息的成本较高，因此执行时可能会遇到 Out-of-Gas 错误。这个问题的一个变通方法是，在您的实际合约中使用它们之前，先向每个合约发送 Wei（例如 1）。这在主网和测试网上都没有问题。

地址类型的成员

<address>.balance （uint256 ）
以 Wei 为单位的地址类型 的余额。

<address>.code （bytes memory ）
在地址类型 的代码（可以是空的）。

<address>.codehash （bytes32 ）
地址类型 的代码哈希值

<address payable>.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 payable>.send(uint256 amount) returns (bool)
向地址类型 发送数量为 amount 的 Wei，失败时返回 `false` 2300 gas 的矿工费用，不可调节。

<address>.call(bytes memory) returns (bool, bytes memory)
用给定的数据发出低级别的 `CALL`，返回是否成功的结果和数据，发送所有可用 gas，可调节。

<address>.delegatecall(bytes memory) returns (bool, bytes memory)
用给定的数据发出低级别的 `DELEGATECALL`，返回是否成功的结果和数据，发送所有可用 gas，可调节。

<address>.staticcall(bytes memory) returns (bool, bytes memory)
用给定的数据发出低级别的 `STATICCALL`，返回是否成功的结果和数据，发送所有可用 gas，可调节。

更多信息，请参见地址类型 一节。

> **警告：** 您应该尽可能避免在执行另一个合约函数时使用 `.call()`，因为它绕过了类型检查、函数存在性检查和参数打包。

> **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。为了保证以太坊转账安全，总是检查 `send` 的返回值，使用 `transfer` 或者下面更好的方式：用接收者提款的模式。

> **警告：** 由于 EVM 认为对一个不存在的合约的调用总是成功的，Solidity 在执行外部调用时使用 `extcodesize` 操作码进行额外的检查。这确保了即将被调用的合约要么实际存在（它包含代码），要么就会产生一个异常。
>
> 对地址而不是合约实例进行低级调用（即 `.call()`, `.delegatecall()`, `.staticcall()`, `.send()` 和 `.transfer()`）**不包括**这种检查，这使得它们在 gas 方面更便宜，但也更不安全。

备注： 在 0.5.0 版本之前，Solidity 允许地址成员被合约实例访问，例如 `this.balance`。现在这被禁止了，必须做一个明确的地址转换。`address(this).balance`。

备注： 如果状态变量是通过低级别的委托调用来访问的，那么两个合约的存储布局必须一致，以便被调用的合约能够正确地通过名称来访问调用合约的存储变量。当然，如果存储指针作为函数参数被传递的话，情况就不是这样了，就像高层库的情况一样。

备注： 在 0.5.0 版本之前，`.call`，`.delegatecall` 和 `.staticcall` 只返回成功状况，不返回数据。

备注： 在 0.5.0 版本之前，有一个名为 `callcode` 的成员，其语义与 `delegatecall` 相似但略有不同。

合约相关

`this`（当前合约类型）
当前合约，可以明确转换为地址类型

`super`
继承层次结构中更高一级的合约

`selfdestruct(address payable recipient)`
销毁当前合约，将其资金发送到给定的地址类型 并结束执行。注意，`selfdestruct` 有一些从 EVM 继承的特殊性：

- 接收合约的接收函数不会被执行。
- 合约只有在交易结束时才真正被销毁，任何一个 `revert` 可能会" 恢复" 销毁。

此外，当前合约的所有函数都可以直接调用，包括当前函数。

警告： 从 0.8.18 及以上版本开始，在 Solidity 和 Yul 中使用 `selfdestruct` 将触发一个已废弃警告，因为 `SELFDESTRUCT` 操作码最终会发生如 EIP-6049 中所述的行为上的重大变化。

备注： 在 0.5.0 版本之前，有一个叫做 `suicide` 的函数，其语义与 `selfdestruct` 相同。

类型信息

表达式 `type(X)` 可以用来检索关于 `X` 类型的信息。目前，对这一功能的支持是有限的（`X` 可以是合约类型或整数型），但在未来可能会扩展。

以下是合约类型 `C` 的可用属性：

type(C).name
合约的名称。

type(C).creationCode
内存字节数组，包含合约的创建字节码。可以在内联程序中用来建立自定义的创建程序，特别是通过使用 `create2` 操作码。这个属性 **不能**在合约本身或任何派生合约中被访问。它会导致字节码被包含在调用站点的字节码中，因此像这样的循环引用是不可能的。

type(C).runtimeCode
内存字节数组，包含合约运行时的字节码。通常是由 `C` 的构造函数部署的代码。如果 `C` 有一个使用内联汇编的构造函数，这可能与实际部署的字节码不同。还要注意的是，库合约在部署时修改其运行时字节码，以防止常规调用。与 `.creationCode` 相同的限制也适用于这个属性。

除了上述属性外，以下属性对接口类型 `I` 可用：

type(I).interfaceId:
一个 `bytes4` 值，是包含给定接口 `I` 的 EIP-165 接口标识符。这个标识符被定义为接口本身定义的所有函数选择器的 `XOR`，不包括所有继承的函数。

以下属性可用于整数类型 `T`：

type(T).min
类型 `T` 所能代表的最小值。

type(T).max
类型 `T` 所能代表的最大值。

3.7.4 保留关键词

这些关键字在 Solidity 中是保留的。它们在将来可能成为语法的一部分：

`after`, `alias`, `apply`, `auto`, `byte`, `case`, `copyof`, `default`, `define`, `final`, `implements`, `in`, `inline`, `let`, `macro`, `match`, `mutable`, `null`, `of`, `partial`, `promise`, `reference`, `relocatable`, `sealed`, `sizeof`, `static`, `supports`, `switch`, `typedef`, `typeof`, `var`。

3.8 表达式和控制结构

3.8.1 控制结构

大多数从大括号语言中知道的控制结构都可以在 Solidity 中使用：

有：if，else，while，do，for，break，continue，return，这些在 C 或者 JavaScript 中表达相同语义的关键词。

Solidity 也支持 try / catch 形式的语句的异常处理，但只适用于外部函数调用 和合约创建调用。可以使用恢复状态 来创建错误。

条件句 不能省略括号，但单句体周围可以省略大括号。

请注意，没有像 C 和 JavaScript 那样从非布尔类型到布尔类型的类型转换，所以 if (1) { ... } 在 Solidity 不是有效的。

3.8.2 函数调用

内部函数调用

当前合约中的函数可以直接（"从内部"）调用，也可以递归调用，就像下边这个荒谬的例子一样：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

// 这会有一个警告
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

这些函数调用在 EVM 内部被转化为简单的跳转。这样做的效果是，当前的内存不会被清空，也就是说，将内存引用传递给内部调用的函数是非常有效的。但只有同一合约实例的函数可以被内部调用。

您还是应该避免过度的递归调用，因为每个内部函数的调用都会占用至少一个堆栈槽，而可用的堆栈槽只有 1024 个。

外部函数调用

函数也可以使用 `this.g(8);` 和 `c.g(2);` 符号来调用，其中 `c` 是一个合约实例，`g` 是属于 `c` 的函数。通过这两种方式调用函数 `g` 会导致它被"外部"调用，使用消息调用而不是直接通过跳转。请注意，对 `this` 的函数调用不能在构造函数中使用，因为实际的合约还没有被创建。

其他合约的函数必须被外部调用。对于一个外部调用，所有的函数参数都必须被拷贝到内存中。

备注： 从一个合约到另一个合约的函数调用并不创建自己的交易，它是作为整个交易的一部分的消息调用。

当调用其他合约的函数时，您可以用特殊的选项 `{value: 10, gas: 10000}` 指定随调用发送的 Wei 或气体（gas）数量。请注意，不鼓励明确指定气体值，因为操作码的气体成本可能在未来发生变化。您发送给合约的任何 Wei 都会被添加到该合约的总余额中：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

您需要对 `info` 函数使用修饰符 `payable`，因为不这样的话，`value` 选项则不可用。

警告： 注意 `feed.info{value: 10, gas: 800}` 只在本地设置 `value` 和随函数调用发送的 `gas` 数量，最后的括号执行实际调用。所以 `feed.info{value: 10, gas: 800}` 不会调用函数，`value` 和 `gas` 的设置也会丢失，只有 `feed.info{value: 10, gas: 800}()` 执行了函数调用。

由于 EVM 认为对一个不存在的合约的调用总是成功的，Solidity 使用 `extcodesize` 操作码来检查即将被调用的合约是否真的存在（它包含代码），如果不存在就会引起异常。如果返回数据将在调用后被解码，则跳过该检查，因此 ABI 解码器将捕获不存在的合约的情况。

请注意，这个检查在低级调用 的情况下不执行，这些调用是对地址而不是合约实例进行操作。

备注： 在对预编译合约 使用高级调用时要小心，因为根据上述逻辑，编译器认为它们不存在，即使它们执行代码并可以返回数据。

如果被调用的合约本身抛出异常或超出了 gas 值，函数调用也会引起异常。

警告： 与另一个合约的任何互动都会带来潜在的危险，特别是当合约的源代码事先不知道的时候。当前的合约将控制权交给了被调用的合约，而这有可能做任何事情。即使被调用的合约继承自一个已知的父合约，继承的合约也只需要有一个正确的接口。然而，合约的实现完全可以是任意的，因此这会带来危险。此外，要做好准备，以防它调用到您系统中的其他合约，甚至在第一次调用返回之前就回到调用合约中。这意味着被调用的合约可以通过这个函数改变调用合约的状态变量。编写您的函数时，例如，对外部函数的调用发生在对您的合约中的状态变量的任何改变之后，这样您的合约就不会受到重入性漏洞的攻击。

备注： 在 Solidity 0.6.2 之前，指定以太值和气体值的推荐方法是使用 `f.value(x).gas(g)()`。这在 Solidity 0.6.2 中被废弃，并且从 Solidity 0.7.0 开始不再支持。

带命名参数的函数调用

函数调用参数可以用名字来表示，如果用 `{ }` 括起来的话，可以用任何顺序，如下面的例子所示。参数列表在名称上必须与函数声明中的参数列表相一致，但可以有任意的顺序。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }
}
```

函数定义中省略的名称

函数声明中的参数和返回值的名称可以省略。那些名字被省略的参数仍然会出现在堆栈中，但是无法通过名字访问。省略的返回值名称仍然可以通过使用 return 语句向调用者返回一个值。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    // 省略参数名称
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

3.8.3 通过 new 创建合约

一个合约可以使用 new 关键字创建其他合约。待创建合约的完整代码必须在创建的合约被编译时知道，所以递归的创建依赖是不可能的。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // 将作为合约 C 构造函数的一部分执行

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // 随合约的创建发送 ether
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

正如在例子中所看到的，在使用 value 选项创建 D 的实例时，可以发送以太，但不可能限制气体的数量。如果创建失败（由于堆栈耗尽，没有足够的余额或其他问题），会抛出一个异常。

加盐合约创建 / create2

当创建一个合约时，合约的地址是由创建合约的地址和一个计数器计算出来的，这个计数器在每次创建合约时都会增加。

如果您指定了选项 salt （一个 32 字节的值），那么合约的创建将使用一种不同的机制来得出新合约的地址。

它将从创建合约的地址、给定的盐值、创建合约的（创建）字节码和构造函数参数中计算出地址。

特别的是，计数器（"nonce"）没有被使用。这使得创建合约时有更多的灵活性。您能够在新合约创建之前得出它的地址。此外，在创建合约的同时创建其他合约的情况下，您也可以依赖这个地址。

这里的主要用例是做为链外互动的评判的合约，只有在有争议的时候才需要创建。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // 这个复杂的表达式只是告诉您如何预先计算出地址。
        // 它只是用于说明问题。
        // 实际上您只需要 ``new D{salt: salt}(arg)``。
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                abi.encode(arg)
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

> **警告：** 在用加盐方式创建合约时，有一些特殊性。一个合约可以在被销毁后在同一地址重新创建。然而，新创建的合约有可能具有不同的部署字节码，即使创建字节码是相同的（这是一个要求，否则地址会改变）。这是由于构造函数可以查询在两次创建之间可能发生变化的外部状态，并在存储之前将其纳入部署字节码。

3.8.4 表达式计算顺序

表达式的计算顺序不是特定的（更准确地说，表达式树中某节点的子节点间的计算顺序不是特定的，但它们的结算肯定会在节点自己的结算之前）。该规则只能保证语句按顺序执行，并对布尔表达式进行短路处理。

3.8.5 赋值

解构赋值和返回多个值

Solidity 内部允许元组 (tuple) 类型，也就是一个在编译时元素数量固定的对象列表，列表中的元素可以是不同类型的对象。这些元组可以用来同时返回多个数值，也可以用它们来同时赋值给多个新声明的变量或者既存的变量（或通常的 LValues）：

在 Solidity 中，元组不是适当的类型，它们只能被用来构建表达式的语法分组。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // 用类型声明的变量，并从返回的元组中分配，
        // 不是所有的元素都必须被指定（但数量必须匹配）。
        (uint x, , uint y) = f();
        // 交换数值的常见技巧 -- 对非数值存储类型不起作用。
        (x, y) = (y, x);
        // 元素可以不使用（也适用于变量声明）。
        (index, , ) = f(); // 将index设置为 7
    }
}
```

不可能混合使用声明和非声明变量赋值。例如，下面的方法是无效的。`(x, uint y) = (1, 2);`。

备注： 在 0.5.0 版本之前，给具有更少元素数的元组赋值都是可能的，要么在左边填充，要么在右边填充（无论哪个是空的）。现在这是不允许的，所以两边必须有相同数量的元素。

警告： 当涉及到引用类型时，在同时向多个变量赋值时要小心，因为这可能导致意外的复制行为。

数组和结构体的复杂情况

对于像数组和结构体这样的非值类型，包括 bytes 和 string，赋值的语义更为复杂，详见数据位置和赋值行为。

在下面的例子中，调用 g(x) 对 x 没有影响，因为它在内存中创建了一个独立的存储值的副本。然而，h(x) 成功地修改了 x，因为传递了一个引用而不是一个拷贝。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

3.8.6 作用域和声明

一个被声明的变量将有一个初始默认值，其字节表示为所有的零。变量的" 默认值" 是任何类型的典型" 零状态"。例如，bool 的默认值是 false。uint 或 int 类型的默认值是 0。对于静态大小的数组和 bytes1 到 bytes32，每个单独的元素将被初始化为与其类型相应的默认值。对于动态大小的数组，bytes 和 string，默认值是一个空数组或字符串。对于 enum 类型，默认值是其第一个成员。

Solidity 中的作用域规则遵循了 C99（与其他很多语言一样）：变量将会从它们被声明之后可见，直到一对 { } 块的结束。这一规则有个例外，在 for 循环语句中初始化的变量，其可见性仅维持到 for 循环的结束。

类似于参数的变量（函数参数、修改器参数、捕获（catch）参数......）在后面的代码块中是可见的--对于函数和修改器参数，在函数/修改器的主体中，对于捕获参数，在捕获块中。

在代码块之外声明的变量，例如函数、合约、用户定义的类型等，甚至在声明之前就已经可见。这意味着您可以在声明之前使用状态变量，并递归地调用函数。

因此，下面的例子在编译时不会出现警告，因为这两个变量的名字虽然相同，但作用域不同。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

作为 C99 作用域规则的特例，请注意在下边的例子里，第一次对 x 的赋值实际上将赋给外层变量而不是内层变量。在任何情况下，您都会得到一个关于外部变量被影射（译者注：就是说被在内部作用域中由一个同名变量所替代）的警告。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// 这将报告一个警告信息
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
```

(续下页)

(接上页)

```
            uint x;
        }
        return x; // x has value 2
    }
}
```

警告： 在 0.5.0 版本之前，Solidity 遵循与 JavaScript 相同的作用域规则，也就是说，在一个函数中的任何地方声明的变量都会在整个函数的作用域中，不管它是在哪里声明。下面的例子显示了一个曾经可以编译的代码片段，但从 0.5.0 版本开始导致了一个错误。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// 这将无法编译
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

3.8.7 检查或不检查的算术

上溢或下溢是指算术运算的结果值，当对一个不受限制的整数执行时，超出了结果类型的范围。

在 Solidity 0.8.0 之前，算术运算总是在下溢或上溢的情况下被包起来，这导致广泛使用引入额外检查的库。

从 Solidity 0.8.0 开始，在默认情况下所有的算术运算都会在上溢和下溢时还原，从而使这些库的使用变得没有必要。

为了获得以前的行为，可以使用一个 `未检查（unchecked）` 区块。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // 这个减法将在下溢时被包起来。
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // 这个减法在下溢时将被还原。
```

(续下页)

(接上页)

```
        return a - b;
    }
}
```

调用 `f(2, 3)` 将返回 `2**256-1`，而 `g(2, 3)` 将导致一个失败的断言。

`unchecked` 代码块可以在代码块内的任何地方使用，但不能替代代码块。它也不能被嵌套。

该设置只影响到在语法上位于代码块内的语句。从 `unchecked` 代码块内调用的函数不继承该属性。

备注： 为了避免歧义，您不能在一个 `unchecked` 代码块内使用 `_;`。

以下运算符在上溢或下溢时将导致一个失败的断言，如果在一个未检查的代码块内使用，将被包裹而不会出现错误。

`++`，`--`，`+`，二进制 `-`，单进制 `-`，`*`，`/`，`%`，`**`

`+=`，`-=`，`*=`，`/=`，`%=`

警告： 不能使用 `unchecked` 代码块来禁止检查除以 0 或对 0 取余数。

备注： 位操作符不执行上溢或下溢检查。这在使用位操作符移位（`<<`，`>>`，`<<=`，`>>=`）来代替整数除法和 2 的幂次方时尤其明显。例如 `type(uint256).max << 3` 不会恢复操作，尽管 `type(uint256).max * 8` 会恢复操作。

备注： `int x = type(int).min; -x;` 中的第二条语句将导致溢出，因为负数范围可以比正数范围多容纳一个值。

明确的类型转换将总是截断，并且永远不会导致失败的断言，但从整数到枚举类型的转换除外。

3.8.8 错误处理：Assert, Require, Revert and Exceptions

Solidity 使用状态恢复异常来处理错误。这种异常将撤消对当前调用（及其所有子调用）中的状态所做的所有更改，并且还向调用者标记错误。

当异常发生在子调用中时，它们会自动" 冒泡"（也就是说，异常被重新抛出），除非它们被 `try/catch` 语句捕获。这个规则的例外是 `send` 和低级函数 `call`，`delegatecall` 和 `staticcall`：它们在发生异常时返回 `false` 作为第一个返回值而不是" 冒泡"。

警告： 如果被调用的账户不存在，低级函数 `call`，`delegatecall` 和 `staticcall` 的第一个返回值为 `true`，这是 EVM 设计的一部分。如果需要的话，必须在调用之前检查账户是否存在。

异常可以包含错误数据，以错误实例 的形式传回给调用者。内置的错误 `Error(string)` 和 `Panic(uint256)` 被特殊函数使用，解释如下。`Error` 用于" 常规" 错误条件，而 `Panic` 用于在无错误代码中不应该出现的错误。

通过 `assert` 引起 Panic 异常和通过 `require` 引起 Error 异常

快捷函数 `assert` 和 `require` 可以用来检查条件，如果不符合条件就抛出一个异常。

`assert` 函数创建了一个 `Panic(uint256)` 类型的错误。在某些情况下，编译器也会产生同样的错误，如下所述。

Assert 应该只用于测试内部错误，以及检查不变量。正确运行的代码不应该创建一个 Panic 异常，甚至在无效的外部输入时也不应该。如果发生这种情况，那么您的合约中就有一个错误，您应该修复它。语言分析工具可以评估您的合约，以确定会导致 Panic 异常的条件和函数调用。

在下列情况下会产生一个 Panic 异常。与错误数据一起提供的错误代码表明 Panic 异常的种类。

1. 0x00：用于一般的编译器插入 Panic 异常的情况。
2. 0x01：如果您带参数调用 `assert` 时结果是 false。
3. 0x11：如果一个算术运算在一个 `unchecked { ... }` 代码块之外导致下溢或上溢。
4. 0x12：如果您对 0 做除法或者取余（例如 `5 / 0` 或者 `23 % 0`）。
5. 0x21：如果您把一个太大的或负数的值转换成一个枚举类型。
6. 0x22：如果您访问一个编码不正确的存储字节数组。
7. 0x31：如果您在一个空数组上调用 `.pop()`。
8. 0x32：如果您访问一个数组，`bytesN` 或一个数组切片索引超出数组长度或负索引（即 `x[i]`，其中 `i >= x.length` 或 `i < 0`）。
9. 0x41：如果您分配了太多的内存空间或创建了一个太大的数组。
10. 0x51：如果您调用一个零初始化的内部函数类型的变量。

`require` 函数要么创造一个没有任何数据的错误，要么创造一个 `Error(string)` 类型的错误。它应该被用来确保在执行之前无法检测到的有效条件。这包括对输入的条件或调用外部合约的返回值。

备注： 目前不能将自定义错误与 `require` 结合使用。请使用 `if (!condition) revert CustomError();` 代替。

在下列情况下，编译器会产生一个 `Error(string)` 异常（或者没有数据的异常）。

1. 调用 `require(x)`，其中 `x` 的值为 `false`。
2. 如果您使用 `revert()` 或 `revert(" 错误描述")`。
3. 如果您执行一个外部函数调用，目标是一个不包含代码的合约。
4. 如果您的合约通过一个没有 `payable` 修饰符的公开函数（包括构造函数和备用函数）接收以太。
5. 如果您的合约通过一个公共的 getter 函数接收以太。

对于以下情况，来自外部调用的错误数据（如果提供的话）会被转发。这意味着它既可以引起 `Error` 异常，也可以引起 `Panic` 异常（或提供的其他什么错误）。

1. 如果 `.transfer()` 失败。
2. 如果您通过消息调用一个函数，但它不能正常完成（即，耗尽了气体，没有匹配的函数，或自己抛出一个异常)，除非使用低级操作 `call`，`send`，`delegatecall`，`callcode` 或 `staticcall`。低级操作从不抛出异常，但通过返回 `false` 表示失败。
3. 如果您使用 `new` 关键字创建一个合约，但合约创建没有正常完成。

您可以选择为 `require` 提供一个信息字符串，但不能为 `assert` 提供。

备注： 如果您没有给 `require` 提供一个字符串参数，它将以空的错误数据进行还原，甚至不包括错误选择器。

下面的例子显示了如何使用 `require` 来检查输入的条件和 `assert` 进行内部错误检查。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // 由于转账失败后抛出异常并且不能在这里回调，
        // 因此我们应该没有办法仍然有一半的钱。
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

在内部，Solidity 会执行恢复操作（指令 `0xfd`）。这会导致 EVM 恢复对状态所做的所有更改。恢复的原因是不能继续安全地执行，因为没有实现预期的效果，还因为我们想保留交易的原子性，所以最安全的做法是恢复所有更改并使整个交易（或至少是调用）不产生效果。

在这两种情况下，调用者可以使用 `try`/ `catch` 对这种失败做出处理，但被调用者的变化将总是被恢复。

备注： 在 Solidity 0.8.0 之前，Panic 异常曾使用 `invalid` 操作码，它消耗了所有可用于调用的气体。在 Metropolis 发布之前，使用 `require` 的异常会消耗所有气体。

`revert`

可以使用 `revert` 语句和 `revert` 函数来触发直接恢复。

`revert` 语句将一个自定义的错误作为直接参数，没有括号：

revert CustomError(arg1, arg2);

出于向后兼容的原因，还有一个 `revert()` 函数，它使用圆括号并接受一个字符串：

revert(); revert("description");

错误数据将被传回给调用者，可以在那里捕获。使用 `revert()` 会导致没有任何错误数据的还原，而 `revert("description")` 将创建一个 `Error(string)` 错误。

使用一个自定义的错误实例通常会比字符串描述便宜得多，因为您可以使用错误的名称来描述它，它的编码只有四个字节。可以通过 NatSpec 提供更长的描述，这不会产生任何费用。

下面的例子显示了如何将一个错误字符串和一个自定义的错误实例与 `revert` 和相应的 `require` 一起使用。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // 另一种方法：
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // 执行购买。
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
```

(续下页)

(接上页)

```
    }
}
```

if (!condition) revert(...); 和 require(condition, ...); 这两种方式是等价的，只要 revert 和 require 的参数没有副作用，比如说它们只是字符串。

备注： require 函数和其他函数一样。这意味着在执行函数本身之前，所有参数都会被评估。特别是，在 require(condition, f()) 中，即使 condition 为真，函数 f 也被执行。

提供的字符串是*ABI 编码* 之后的，就像调用一个函数 Error(string) 一样。在上面的例子中，revert("Not enough Ether provided."); 返回以下十六进制作为错误返回数据：

```
0x08c379a0                                                         // Error(string)␣
↪的函数选择器
0x0000000000000000000000000000000000000000000000000000000000000020 //␣
↪数据的偏移量（32）
0x000000000000000000000000000000000000000000000000000000000000001a // 字符串长度（26）
0x4e6f7420656e6f7567682045746865722070726f76696465642e000000000000 // 字符串数据（
↪"Not enough Ether provided." 的 ASCII 编码，26字节）
```

调用者可以使用 try / catch 检索所提供的消息，如下所示。

备注： 以前有一个叫 throw 的关键字，其语义与 revert() 相同，在 0.4.13 版本中被弃用，在 0.5.0 版本中被删除。

try / catch

外部调用的失败可以用 try/catch 语句来捕获，如下所示：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // 如果有10个以上的错误，就永久停用该机制。
        require(errorCount < 10);
```

(续下页)

(接上页)

```
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // 如果在getData中调用revert，
            // 并且提供了一个原因字符串，
            // 则执行该命令。
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // 在发生Panic异常的情况下执行，
            // 即出现严重的错误，如除以零或溢出。
            // 错误代码可以用来确定错误的种类。
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // 在使用revert()的情况下，会执行这个命令。
            errorCount++;
            return (0, false);
        }
    }
}
```

`try` 关键字后面必须有一个表达式，代表外部函数调用或合约建（`new ContractName()`）。表达式中的错误不会被捕获（例如，如果它是一个复杂的表达式，也涉及到内部函数调用），只有外部调用本身发生恢复。接下来的 `returns` 部分（是可选的）声明了与外部调用返回的类型相匹配的返回变量。如果没有错误，这些变量将被分配，合约执行将在第一个成功代码块内继续。如果到达成功代码块的末端，则在 `catch` 块之后继续执行。

Solidity 根据错误的类型，支持不同种类的捕获块：

- `catch Error(string memory reason) { ... }`：这个 catch 子句会被执行，如果错误是由 `revert("reasonString")` 或 `require(false, "reasonString")` 造成的（或内部错误造成的）。
- `catch Panic(uint errorCode) { ... }`：如果错误是由 Panic 异常引起的，例如由失败的 `assert`、除以 0、无效的数组访问、算术溢出和其他原因引起的，这个 catch 子句将被运行。
- `catch (bytes memory lowLevelData) { ... }`：如果错误签名与其他子句不匹配，或者在解码错误信息时出现了错误，或者没有与异常一起提供错误数据，那么这个子句就会被执行。在这种情况下，声明的变量提供了对低级错误数据的访问。
- `catch { ... }`：如果您对错误数据不感兴趣，您可以直接使用 `catch { ... }`（甚至作为唯一的 catch 子句）来代替前面的子句。

计划在未来支持其他类型的错误数据。字符串 `Error` 和 `Panic` 目前是按原样解析的，不作为标识符处理。

为了捕捉所有的错误情况，您至少要有 `catch { ...}` 或 `catch (bytes memory lowLevelData) {`

`...` `}` 子句。

在 `returns` 和 `catch` 子句中声明的变量只在后面的代码块中有作用域。

备注： 如果在 try/catch 语句内部的返回数据解码过程中发生错误，这将导致当前执行的合约出现异常，正因为如此，它不会在 catch 子句中被捕获。如果在 `catch Error(string memory reason)` 的解码过程中出现错误，并且有一个低级的 catch 子句，那么这个错误就会在那里被捕获。

备注： 如果执行到一个 catch 代码块，那么外部调用的状态改变效果已经被恢复。如果执行到了成功代码块，那么这些影响就没有被还原。如果影响已经被还原，那么执行要么在 catch 代码块中继续，要么 try/catch 语句的执行本身被还原（例如由于上面提到的解码失败或者由于没有提供低级别的 catch 子句）。

备注： 调用失败背后的原因可能是多方面的。不要认为错误信息是直接来自被调用的合约：错误可能发生在调用链的更深处，被调用的合约只是转发了它。另外，这可能是由于消耗完气体值的情况，而不是故意的错误状况。调用方总是保留调用中至少 1/64 的气体值，因此，即使被调用合约没有气体了，调用方仍然有一些气体。

3.9 合约

Solidity 中的合约类似于面向对象语言中的类。它们在状态变量中包含持久的数据，以及可以修改这些变量的函数。在不同的合约（实例）上调用一个函数将执行一个 EVM 函数调用，从而切换上下文，使调用合约中的状态变量无法访问。一个合约和它的函数需要被调用才会发生。在以太坊中没有"cron" 的概念，在特定的事件中自动调用一个函数。

3.9.1 创建合约

可以通过以太坊交易 "从外部" 或从 Solidity 合约内部创建合约。

集成开发环境，如 Remix，使用 UI 元素使创建过程无缝化。

在以太坊上以编程方式创建合约的一种方法是通过 JavaScript API web3.js。它有一个名为 web3.eth.Contract 的函数，以方便创建合约。

当一个合约被创建时，它的构造函数（*constructor*）（一个用 `constructor` 关键字声明的函数）被执行一次。

构造函数是可选的。但是只允许有一个构造函数，这意味着不支持重载。

构造函数执行完毕后，合约的最终代码被存储在区块链上。这段代码包括所有公开和外部函数，以及所有通过函数调用可从那里到达的函数。部署的代码不包括构造函数代码或只从构造函数调用的内部函数。

在内部，构造函数参数在合约代码之后通过*ABI 编码* 传递，但是如果您使用 `web3.js` 则不必关心这个问题。

如果一个合约想创建另一个合约，创建者必须知道所创建合约的源代码（和二进制）。这意味着，循环的创建依赖是不可能的。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;


contract OwnedToken {
    // `TokenCreator` 是如下定义的合约类型。
    // 不创建新合约的话，也可以引用它。
    TokenCreator creator;
    address owner;
    bytes32 name;

    // 这是注册 creator 和设置名称的构造函数。
    constructor(bytes32 name_) {
        // 状态变量通过其名称访问，
        // 而不是通过例如 `this.owner` 的方式访问。
        // 函数可以直接或通过 `this.f` 访问。
        // 但后者提供了一个对函数的外部可视方法。
        // 特别是在构造函数中，您不应该从外部访问函数，
        // 因为该函数还不存在。
        // 详见下一节。
        owner = msg.sender;

        // 我们进行了从 `address` 到 `TokenCreator` 的显式类型转换，
        // 并假定调用合约的类型是 `TokenCreator`，
        // 没有真正的方法来验证，
        // 这并没有创建一个新的合约。
        creator = TokenCreator(msg.sender);
        name = name_;
    }

    function changeName(bytes32 newName) public {
        // 只有创建者可以改变名称。
        // 我们根据合约的地址进行比较，
        // 它可以通过显式转换为地址来检索。
        if (msg.sender == address(creator))
            name = newName;
    }

    function transfer(address newOwner) public {
        // 只有当前所有者才能发送 token。
```

(续下页)

(接上页)

```
        if (msg.sender != owner) return;

        // 我们通过使用下面定义的 `TokenCreator` 合约的一个函数
        // 来询问创建者合约是否应该进行转移。
        // 如果调用失败（例如由于气体值耗尽），
        // 这里的执行也会失败。
        if (creator.isTokenTransferOK(owner, newOwner))
            owner = newOwner;
    }
}


contract TokenCreator {
    function createToken(bytes32 name)
        public
        returns (OwnedToken tokenAddress)
    {
        // 创建一个新的 `Token` 合约并返回其地址。
        // 从JavaScript方面来看，
        // 这个函数的返回类型是 `address`，
        // 因为这是ABI中最接近的类型。
        return new OwnedToken(name);
    }

    function changeName(OwnedToken tokenAddress, bytes32 name) public {
        // 同样，`tokenAddress` 的外部类型是简单的 `address`。
        tokenAddress.changeName(name);
    }

    // 执行检查，以确定是否应该将代币转移到 `OwnedToken` 合约上。
    function isTokenTransferOK(address currentOwner, address newOwner)
        public
        pure
        returns (bool ok)
    {
        // 检查一个任意的条件，看是否应该进行转移。
        return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;
    }
}
```

3.9.2 可见性和 getter 函数

状态变量的可见性

public
: 公开状态变量与内部变量的不同之处在于，编译器会自动为它们生成*getter 函数*，从而允许其他合约读取它们的值。当在同一个合约中使用时，外部访问（例如 `this.x`）会调用 getter，而内部访问（例如 `x`）会直接从存储中获取变量值。Setter 函数没有被生成，所以其他合约不能直接修改其值。

internal
: 内部状态变量只能从它们所定义的合约和派生合约中访问。它们不能被外部访问。这是状态变量的默认可见性。

private
: 私有状态变量就像内部变量一样，但它们在派生合约中是不可见的。

> **警告：** 标记一些变量为 `private` 或 `internal`，只能防止其他合约读取或修改信息，但它仍然会被区块链之外的整个世界看到。

函数的可见性

Solidity 有两种函数调用：确实创建了实际 EVM 消息调用的外部函数和不创建 EVM 消息调用的内部函数。此外，派生合约可能无法访问内部函数。这就产生了四种类型的函数的可见性。

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）。

public
: 公开函数是合约接口的一部分，可以在内部或通过消息调用。

internal
: 内部函数只能从当前的合约或从它派生出来的合约中访问。它们不能被外部访问。由于它们没有通过合约的 ABI 暴露在外部，它们可以接受内部类型的参数，如映射或存储引用。

private
: 私有函数和内部函数一样，但它们在派生合约中是不可见的。

> **警告：** 标记一些变量为 `private` 或 `internal`，只能防止其他合约读取或修改信息，但它仍然会被区块链之外的整个世界看到。

在状态变量的类型之后，以及在函数的参数列表和返回参数列表之间，都会给出可见性指定符。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

在下面的例子中，合约 D, 可以调用 c.getData() 来检索状态存储中 data 的值，但不能调用 f。合约 E 是从合约 C 派生出来的，因此可以调用 compute。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint private data;

    function f(uint a) private pure returns(uint b) { return a + 1; }
    function setData(uint a) public { data = a; }
    function getData() public view returns(uint) { return data; }
    function compute(uint a, uint b) internal pure returns (uint) { return a + b; }
}

// 这将不会编译
contract D {
    function readData() public {
        C c = new C();
        uint local = c.f(7); // 错误: 成员 `f` 不可见
        c.setData(3);
        local = c.getData();
        local = c.compute(3, 5); // 错误: 成员 `compute` 不可见
    }
}

contract E is C {
    function g() public {
        C c = new C();
        uint val = compute(3, 5); // 访问内部成员 (从继承合约访问父合约成员)
    }
}
```

Getter 函数

编译器会自动为所有 **公开**状态变量创建 getter 函数。对于下面给出的合约，编译器将生成一个名为 `data` 的函数，它没有任何输入参数，并返回一个 `uint`，即状态变量 `data` 的值。状态变量在声明时可以被初始化。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public view returns (uint) {
        return c.data();
    }
}
```

getter 函数具有外部可见性。如果该符号被内部访问（即没有 `this.`），它被评估为一个状态变量。如果它被外部访问（即有 `this.`），它将被评价为一个函数。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    uint public data;
    function x() public returns (uint) {
        data = 3; // 内部访问
        return this.data(); // 外部访问
    }
}
```

如果您有一个数组类型的 `public` 状态变量，那么您只能通过生成的 getter 函数检索数组的单个元素。这种机制的存在是为了避免在返回整个数组时产生高额的气体成本。您可以使用参数来指定要返回的单个元素，例如 `myArray(0)`。如果您想在一次调用中返回整个数组，那么您需要写一个函数，例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {
    // 公开状态变量
    uint[] public myArray;

    // 编译器生成的getter函数
```

(续下页)

(接上页)

```
    /*
    function myArray(uint i) public view returns (uint) {
        return myArray[i];
    }
    */

    // 返回整个数组的函数
    function getArray() public view returns (uint[] memory) {
        return myArray;
    }
}
```

现在您可以使用 `getArray()` 来检索整个数组，而不是使用 `myArray(i)`，它每次调用只返回一个元素。

下一个例子稍微复杂一些：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Complex {
    struct Data {
        uint a;
        bytes3 b;
        mapping(uint => uint) map;
        uint[3] c;
        uint[] d;
        bytes e;
    }
    mapping(uint => mapping(bool => Data[])) public data;
}
```

它生成了一个如下形式的函数。结构中的映射和数组（字节数组除外）被省略了，因为没有好的方法来选择单个结构成员或为映射提供一个键：

```
function data(uint arg1, bool arg2, uint arg3)
    public
    returns (uint a, bytes3 b, bytes memory e)
{
    a = data[arg1][arg2][arg3].a;
    b = data[arg1][arg2][arg3].b;
    e = data[arg1][arg2][arg3].e;
}
```

3.9.3 函数修饰器

函数修饰器可以用来以声明的方式改变函数的行为。例如，您可以使用修饰器在执行函数之前自动检查一个条件。

修饰器是合约的可继承属性，可以被派生合约重载，但只有当它们被标记为 virtual 时，才能被重载。详情请见修饰器重载。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告

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

    // 这个合约只定义了一个修饰器，但没有使用它：
    // 它将在派生合约中使用。
    // 修饰器所修饰的函数体会被插入到特殊符号 `_;` 的位置。
    // 这意味着，如果所有者调用这个函数，这个函数就会被执行，
    // 否则就会抛出一个异常。
    modifier onlyOwner {
        require(
            msg.sender == owner,
            "Only owner can call this function."
        );
        _;
    }
}

contract destructible is owned {
    // 这个合约从 `owned` 合约继承了 `onlyOwner` 修饰器，
    // 并将其应用于 `destroy` 函数，
    // 只有在合约里保存的 owner 调用 `destroy` 函数，才会生效。
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract priced {
    // 修饰器可以接受参数：
    modifier costs(uint price) {
        if (msg.value >= price) {
            _;
        }
```

(续下页)

(接上页)

```
    }
}

contract Register is priced, destructible {
    mapping(address => bool) registeredAddresses;
    uint price;

    constructor(uint initialPrice) { price = initialPrice; }

    // 在这里也使用关键字 `payable` 非常重要，
    // 否则函数会自动拒绝所有发送给它的以太币。
    function register() public payable costs(price) {
        registeredAddresses[msg.sender] = true;
    }

    function changePrice(uint price_) public onlyOwner {
        price = price_;
    }
}

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    /// 这个函数受互斥量保护，这意味着 `msg.sender.call` 中的重入调用不能再次调用 ␣
↪`f`。
    /// `return 7` 语句指定返回值为 7，但修饰器中的语句 `locked = false` 仍会执行。
    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

如果您想访问定义在合约 C 中的修饰器 m，您可以使用 C.m 来引用它而不需要虚拟查询。只能使用定义在当前合约或其基础合约中的修饰器。修饰器也可以定义在库合约中，但其使用仅限于同一库合约的函数。

如果同一个函数有多个修饰器，它们之间以空格隔开，并按照所呈现的顺序进行评估运算。

修饰器不能隐式地访问或改变它们所修改的函数的参数和返回值。它们的值只能在调用的时候明确地传递给它们。

在函数修改器中，有必要指定何时运行应用修改器的函数。占位符语句（用单个下划线字符 _ 表示）用于表示被修改的函数主体应该插入的位置。请注意，占位符操作符与在变量名中使用下划线作为前导或尾随字符不同，后者是一种风格上的选择。

修饰器或函数体的显式返回只离开当前修饰器或函数体。返回变量会被赋值，但整个执行逻辑会从前一个修饰器中定义的 _ 之后继续执行。

> **警告：** 在 Solidity 的早期版本中，具有修饰器的函数中的 `return` 语句会表现的不同。

用 `return;` 从修饰器显式返回并不影响函数返回的值。然而，修饰器可以选择完全不执行函数主体，在这种情况下，返回变量被设置为默认值，就像函数有一个空主体一样。

_ 符号可以在修饰器中多次出现。每次出现都会被替换成函数体。

允许修饰器参数使用任意表达式，在这种情况下，所有从函数中可见的符号在修饰器中都是可见的。修饰器中引入的符号在函数中是不可见的（因为它们可能因重载而改变）。

3.9.4 Constant 和 Immutable 状态变量

状态变量可以被声明为 `constant` 或 `immutable`。在这两种情况下，变量在合约构建完成后不能被修改。对于 `constant` 变量，其值必须在编译时固定，而对于 `immutable` 变量，仍然可以在构造时分配。

也可以在文件级别定义 `constant` 变量。

编译器并没有为这些变量预留存储，它们的每次出现都会被替换为相应的常量表达式。

与普通的状态变量相比，常量变量（constant）和不可改变的变量（immutable）的气体成本要低得多。对于常量变量，分配给它的表达式被复制到所有访问它的地方，并且每次都要重新评估，这使得局部优化成为可能。不可变的变量在构造时被评估一次，其值被复制到代码中所有被访问的地方。对于这些值，要保留 32 个字节，即使它们可以装入更少的字节。由于这个原因，常量值有时会比不可变的值更便宜。

目前，并非所有的常量和不可变量的类型都已实现。唯一支持的类型是字符串类型（仅用于常量）和值类型。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
```

(续下页)

(接上页)

```
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint decimals_, address ref) {
        decimals = decimals_;
        // 对不可变量的赋值甚至可以访问一些全局属性。
        maxBalance = ref.balance;
    }

    function isBalanceTooHigh(address other) public view returns (bool) {
        return other.balance > maxBalance;
    }
}
```

Constant

对于 constant 变量，其值在编译时必须是一个常量，并且必须在变量声明的地方分配。任何访问存储、区块链数据（例如：block.timestamp，address(this).balance 或 block.number）或执行数据（msg.value 或 gasleft()）或者调用外部合约的表达式都是不允许的。但可能对内存分配产生副作用的表达式是允许的，但那些可能对其他内存对象产生副作用的表达式是不允许的。内置函数 keccak256，sha256，ripemd160，ecrecover，addmod 和 mulmod 是允许的（尽管除了 keccak256，它们确实调用了外部合约）。

允许在内存分配器上产生副作用的原因是，它应该可以构建复杂的对象，比如说查找表。这个功能现在还不能完全使用。

Immutable

声明为 immutable 的变量比声明为 constant 的变量受到的限制要少一些。不可变的变量可以在合约的构造函数中或在声明时被分配一个任意的值。它们只能被分配一次，并且从那时起，即使在构造时间内也可以被读取。

编译器生成的合约创建代码将在其返回之前修改合约的运行时代码，用分配给它们的值替换所有对不可变量的引用。当您将编译器生成的运行时代码与实际存储在区块链中的代码进行比较时，这一点很重要。

备注： 在声明时被分配的不可变量只有在合约的构造函数执行时才会被视为初始化。这意味着您不能在内联中用一个依赖于另一个不可变量的值来初始化不可变量。然而，您可以在合约的构造函数中这样做。

这是对状态变量初始化和构造函数执行顺序的不同解释的一种保障，特别是在继承方面。

3.9.5 函数

可以在合约内部和外部定义函数。

合约之外的函数，也称为" 自由函数"，总是隐含着 internal 的可见性。它们的代码包含在所有调用它们的合约中，类似于内部库函数。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory arr) pure returns (uint s) {
    for (uint i = 0; i < arr.length; i++)
        s += arr[i];
}

contract ArrayExample {
    bool found;
    function f(uint[] memory arr) public {
        // 这在内部调用自由函数。
        // 编译器会将其代码添加到合约中。
        uint s = sum(arr);
        require(s >= 10);
        found = true;
    }
}
```

备注： 在合约之外定义的函数仍然总是在合约的范围内执行。它们仍然可以调用其他合约，向它们发送以太，并销毁调用它们的合约，以及其他一些事情。与合约内定义的函数的主要区别是，自由函数不能直接访问变量 this，存储变量和不在其范围内的函数。

函数参数和返回变量

与许多其他语言不同，函数接受类型化的参数作为输入，也可以返回任意数量的值作为输出。

函数参数

函数参数的声明方式与变量相同，未使用的参数名称可以省略。

例如，如果您想让您的合约接受一种带有两个整数的外部调用，您可以使用类似以下的方式：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;
```

(续下页)

(接上页)

```
contract Simple {
    uint sum;
    function taker(uint a, uint b) public {
        sum = a + b;
    }
}
```

函数参数可以像任何其他局部变量一样使用，它们也可以被赋值。

返回的变量

函数的返回变量在 returns 关键字之后用同样的语法声明。

例如，假设您想返回两个结果：作为函数参数传递的两个整数的总和和乘积，那么您就使用类似的方法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
    {
        sum = a + b;
        product = a * b;
    }
}
```

返回变量的名字可以被省略。返回变量可以像其他本地变量一样使用，它们被初始化为相应的默认值，并且在它们被（重新）赋值之前拥有这个值。

您可以明确地赋值给返回变量，然后像上面那样结束函数，或者您可以用 return 语句直接提供返回值（单个或多个返回值）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Simple {
    function arithmetic(uint a, uint b)
        public
        pure
        returns (uint sum, uint product)
```

(续下页)

(接上页)

```
    {
        return (a + b, a * b);
    }
}
```

如果您过早使用 return 来结束一个有返回变量的函数，您必须在返回语句中同时提供返回值。

备注： 您不能从非内部函数返回某些类型。这包括下面列出的类型和任何递归地包含它们的复合类型：

- 映射，
- 内部函数类型，
- 参考类型，位置设置为 storage，
- 多维数组（仅适用于*ABI coder v1*），
- 结构体（仅适用于*ABI coder v1*）。

这个限制不适用于库函数，因为它们有不同的内部 *ABI*。

返回多个值

当一个函数有多个返回类型时，语句 return (v0, v1, ..., vn) 可以用来返回多个值。声明的数量必须与返回变量的数量相同，并且它们的类型必须匹配，有可能是经过隐式转换。

状态可变性

View 函数

函数可以被声明为 view，在这种情况下，它们承诺不修改状态。

备注： 如果编译器的 EVM 版本是 Byzantium 或更新的（默认），当调用 view 函数时，会使用操作码 STATICCALL，这使得状态作为 EVM 执行的一部分保持不被修改。对于库合约的 view 函数，会使用 DELEGATECALL，因为没有组合的 DELEGATECALL 和 STATICCALL。这意味着库合约中的 view 函数没有防止状态修改的运行时的检查。这应该不会对安全产生负面影响，因为库合约的代码通常在编译时就知道了，而且静态检查器也会进行编译时检查。

以下声明被认为是修改状态：

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。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

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

备注: 函数上的 constant 曾经是 view 的别名，但在 0.5.0 版本中被取消。

备注: Getter 方法被自动标记为 view。

备注: 在 0.5.0 版本之前，编译器没有为 view 函数使用 STATICCALL 操作码。这使得 view 函数通过使用无效的显式类型转换进行状态修改。通过对 view 函数使用 STATICCALL，在 EVM 层面上防止了对状态的修改。

Pure 函数

函数可以被声明为 pure，在这种情况下，它们承诺不读取或修改状态。特别是，应该可以在编译时评估一个 pure 函数，只给它的输入和 msg.data，但不知道当前区块链状态。这意味着读取 immutable 的变量可以是一个非标准 pure 的操作。

备注: 如果编译器的 EVM 版本是 Byzantium 或更新的（默认），则使用操作码 STATICCALL，这并不能保证不读取状态，但至少不能修改。

除了上面解释的状态修改语句列表外，以下内容被认为是从状态中读取的:

1. 读取状态变量。
2. 访问 address(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。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

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

当一个错误发生 时，Pure 函数能够使用 revert() 和 require() 函数来恢复潜在的状态变化。

恢复一个状态变化不被认为是”状态修改”，因为只有之前在没有 view 或 pure 限制的代码中对状态的改变才会被恢复，并且该代码可以选择捕捉 revert 而不传递给它。

这种行为也与 STATICCALL 操作码一致。

警告： 在 EVM 层面不可能阻止函数读取状态，只可能阻止它们写入状态（即只有 view 可以在 EVM 层面执行，pure 不可以）。

备注： 在 0.5.0 版本之前，编译器没有为 pure 函数使用 STATICCALL 操作码。这使得在 pure 函数中通过使用无效的显式类型转换进行状态修改。通过对 pure 函数使用 STATICCALL，在 EVM 层面防止了对状态的修改。

备注： 在 0.4.17 版本之前，编译器并没有强制要求 pure 不读取状态。这是一个编译时的类型检查，可以规避在合约类型之间做无效的显式转换，因为编译器可以验证合约的类型不做改变状态的操作，但它不能检查将在运行时被调用的合约是否真的属于该类型。

特殊的函数

接收以太的函数

一个合约最多可以有一个 `receive` 函数，使用 `receive() external payable { ... }` 来声明。（没有 `function` 关键字）。这个函数不能有参数，不能返回任何东西，必须具有 `external` 的可见性和 `payable` 的状态可变性。它可以是虚拟的，可以重载，也可以有修饰器。

receive 函数是在调用合约时执行的，并带有空的 calldata。这是在纯以太传输（例如通过 `.send()` 或 `.transfer()` ）时执行的函数。如果不存在这样的函数，但存在一个 payable 类型的 *fallback 函数*，这个 fallback 函数将在纯以太传输时被调用。如果既没有直接接收以太（receive 函数），也没有 payable 类型的 fallback 函数，那么合约就不能通过不代表支付函数调用的交易接收以太币，还会抛出一个异常。

在最坏的情况下，`receive` 函数只有 2300 个气体可用（例如当使用 `send` 或 `transfer` 时），除了基本的记录外，几乎没有空间来执行其他操作。以下操作的消耗气体将超过 2300 气体的规定：

- 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

警告： 当以太被直接发送到一个合约（没有使用函数调用，即发送者使用 `send` 或 `transfer`），但接收合约没有定义一个接收以太的函数或一个 payable 类型的 fallback 函数，会抛出一个异常，将以太送回（这在 Solidity v0.4.0 之前是不同的）。因此，如果您想让您的合约接收以太，您必须实现一个 receive 函数（不建议使用 payable 类型的 fallback 函数来接收以太，因为它不会因为接口混乱而失败）。

警告： 没有接收以太币功能的合约可以作为 *coinbase* 交易 *（又称 *矿工区块奖励）的接收者或作为 `selfdestruct` 的目的地接收以太币。

合约不能对这样的以太币转移做出反应，因此也不能拒绝它们。这是 EVM 的一个设计选择，Solidity 无法绕过它。

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

下面您可以看到一个使用 `receive` 函数的 Sink 合约的例子。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

// 这个合约会保留所有发送给它的以太币，没有办法返还。
```

(续下页)

(接上页)

```
contract Sink {
    event Received(address, uint);
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }
}
```

Fallback 函数

一个合约最多可以有一个 `fallback` 函数，使用 `fallback () external [payable]` 或 `fallback (bytes calldata input) external [payable] returns (bytes memory output)` 来声明（都没有 `function` 关键字）。这个函数必须具有 `external` 的函数可见性。一个 fallback 函数可以被标记为 virtual，可以标记为 override，也可以有修饰器。

如果其他函数都不符合给定的函数签名，或者根本没有提供数据，也没有接收以太的函数，那么 fallback 函数将在调用合约时执行。fallback 函数总是接收数据，但为了同时接收以太，它必须被标记为 `payable`。

如果使用带参数的版本，`input` 将包含发送给合约的全部数据（等于 `msg.data`），并可以在 `output` 中返回数据。返回的数据将不会被 ABI 编码。相反，它将在没有修改的情况下返回（甚至没有填充）。

在最坏的情况下，如果一个可接收以太的 fallback 函数也被用来代替接收功能，那么它只有 2300 气体是可用的（参见接收以太函数 对这一含义的简要描述）。

像任何函数一样，只要有足够的气体传递给它，fallback 函数就可以执行复杂的操作。

警告： 如果没有*receive 函数* 的存在，一个标记为 `payable` 的 fallback 函数也会在普通的以太传输时执行。如果您已经定义了一个 payable 类型的 fallback 函数，我们仍建议您也定义一个 receive 函数接收以太，以区分以太传输和接口混淆的情况。

备注： 如果您想对输入数据进行解码，您可以检查前四个字节的函数选择器，然后您可以使用 `abi.decode` 与数组切片语法一起对 ABI 编码的数据进行解码：`(c, d) = abi.decode(input[4:], (uint256, uint256));` 注意，这只能作为最后的手段，应该使用适当的函数来代替。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract Test {
    uint x;
    // 所有发送到此合约的消息都会调用此函数（没有其他函数）。
    // 向该合约发送以太币将引起异常，
```

(续下页)

(接上页)

```
    // 因为fallback函数没有 `payable` 修饰器。
    fallback() external { x = 1; }
}

contract TestPayable {
    uint x;
    uint y;
    // 所有发送到此合约的消息都会调用这个函数,
    // 除了普通的以太传输（除了receive函数，没有其他函数）。
    // 任何对该合约的非空的调用都将执行fallback函数（即使以太与调用一起被发送）。
    fallback() external payable { x = 1; y = msg.value; }

    // 这个函数是为纯以太传输而调用的,
    // 即为每一个带有空calldata的调用。
    receive() external payable { x = 2; y = msg.value; }
}

contract Caller {
    function callTest(Test test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // 结果是 test.x 等于 1。

        // address(test)将不允许直接调用 ``send``,
        // 因为 ``test`` 没有可接收以太的fallback函数。
        // 它必须被转换为 ``address payable`` 类型，才允许调用 ``send``。
        address payable testPayable = payable(address(test));

        // 如果有人向该合约发送以太币，转账将失败，即这里返回false。
        return testPayable.send(2 ether);
    }

    function callTestPayable(TestPayable test) public returns (bool) {
        (bool success,) = address(test).call(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // 结果是 test.x 等于 1，test.y 等于 0。
        (success,) = address(test).call{value: 1}(abi.encodeWithSignature(
↪"nonExistingFunction()"));
        require(success);
        // 结果是 test.x 等于 1，test.y 等于 1。
```

(续下页)

(接上页)

```
        // 如果有人向该合约发送以太币，TestPayable的receive函数将被调用。
        // 由于该函数会写入存储空间，它需要的气体比简单的 ``send`` 或 ``transfer``␣
↪要多。
        // 由于这个原因，我们必须要使用一个低级别的调用。
        (success,) = address(test).call{value: 2 ether}("");
        require(success);
        // 结果是 test.x 等于 1，test.y 等于 2 个以太。

        return true;
    }
}
```

函数重载

一个合约可以有多个同名的，但参数类型不同的函数。这个过程被称为" 重载"，也适用于继承的函数。下面的例子显示了在合约 A 范围内对函数 f 的重载。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

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

    function f(uint value, bool really) public pure returns (uint out) {
        if (really)
            out = value;
    }
}
```

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。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

// 这段代码不会编译
contract A {
    function f(B value) public pure returns (B out) {
        out = value;
    }

```

(续下页)

(接上页)

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

contract B {
}
```

以上两个 f 函数重载最终都接受 ABI 的地址类型，尽管它们在 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。

备注： 返回参数不作为重载解析的依据。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

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

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

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。

3.9.6 事件

Solidity 事件在 EVM 的日志功能之上给出了一个抽象。应用程序可以通过 Ethereum 客户端的 RPC 接口订阅和监听这些事件。

事件是合约的可继承成员。当您调用它们时，它们会导致参数被存储在交易的日志中--区块链中的一个特殊数据结构。这些日志与合约的地址相关联，被纳入区块链，只要有区块可以访问，就会留在那里（目前是永

远，但这可能会随着 Serenity 升级而改变）。日志及其事件数据不能从合约内部访问（甚至不能从创建它们的合约访问）。

有可能要求为日志提供 Merkle 证明，所以如果外部实体向合约提供这样的证明，它可以检查日志是否真的存在于区块链中。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。

您可以最多给三个参数添加 `indexed` 属性，将它们添加到一个特殊的数据结构中，称为*"topics"*，而不是日志的数据部分。一个 topic 只能容纳一个字（32 字节），所以如果您为一个索引参数使用引用类型，该值的 Keccak-256 哈希值将被存储为一个 topic 中。

所有没有 `indexed` 属性的参数都会被*ABI 编码* 到日志的数据部分。

Topics 允许您用来搜索事件，例如为特定的事件来过滤一系列的区块。您用来也可以通过发出事件的合约的地址来过滤事件。

例如，下面的代码使用 web3.js `subscribe("logs")` 方法 来过滤与某一地址值相匹配的日志：

```
var options = {
    fromBlock: 0,
    address: web3.eth.defaultAccount,
    topics: ["0x0000000000000000000000000000000000000000000000000000000000000000",␣
↪null, null]
};
web3.eth.subscribe('logs', options, function (error, result) {
    if (!error)
        console.log(result);
})
    .on("data", function (log) {
        console.log(log);
    })
    .on("changed", function (log) {
});
```

除非您用 `anonymous` 指定符声明事件，否则事件的签名的哈希值是 topic 之一。这意味着不可能通过名字来过滤特定的匿名事件，您只能通过合约地址来过滤。匿名事件的优点是，它们的部署和调用都比较便宜。它还允许您声明四个索引参数，而不是三个。

备注： 由于交易日志只存储事件数据而不存储类型，因此您必须知道事件的类型，包括哪个参数被索引以及事件是否是匿名的，以便正确解析数据。特别的是，有可能用一个匿名事件“伪造“另一个事件的签名。

事件类型的成员方法

- event.selector：对于非匿名事件，这是一个 bytes32 值，包含事件签名的 keccak256 哈希值，在默认 topic 中使用。

示例

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.21 <0.9.0;

contract ClientReceipt {
    event Deposit(
        address indexed from,
        bytes32 indexed id,
        uint value
    );

    function deposit(bytes32 id) public payable {
        // 事件是用 `emit` 发出的，后面是事件的名称和括号里的参数（如果有）。
        // 任何这样的调用（甚至是深度嵌套）都可以通过过滤 `Deposit`
        // 从JavaScript API中检测出来。
        emit Deposit(msg.sender, id, msg.value);
    }
}
```

在 JavaScript API 中的使用方式如下：

```
var abi = /* 由编译器产生的abi */;
var ClientReceipt = web3.eth.contract(abi);
var clientReceipt = ClientReceipt.at("0x1234...ab67" /* 地址 */);

var depositEvent = clientReceipt.Deposit();

// 监视变化
depositEvent.watch(function(error, result){
    // 结果包含非索引的参数和给 `Deposit` 调用的 topics。
    if (!error)
        console.log(result);
});


// 或者通过回调立即开始监视
var depositEvent = clientReceipt.Deposit(function(error, result) {
    if (!error)
```

(续下页)

(接上页)

```
        console.log(result);
});
```

上面的输出看起来像下面这样（经过修剪）：

```
{
   "returnValues": {
       "from": "0x1111…FFFFCCCC",
       "id": "0x50…sd5adb20",
       "value": "0x420042"
   },
   "raw": {
       "data": "0x7f…91385",
       "topics": ["0xfd4…b4ead7", "0x7f…1a91385"]
   }
}
```

了解事件类型的其他资料

- JavaScript 文档
- 事件的使用实例
- 如何在 js 中访问它们

3.9.7 错误和恢复语句

Solidity 中的错误提供了一种方便且省 gas 的方式来向用户解释为什么一个操作会失败。它们可以被定义在合约内部和外部（包括接口合约和库合约）。

它们必须与恢复语句 一起使用，它导致当前调用中的所有变化被恢复，并将错误数据传回给调用者。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

/// 转账的余额不足。需要 `required` 数量但只有 `available` 数量可用。
/// @param 可用的余额。
/// @param 需要要求的转帐金额。
error InsufficientBalance(uint256 available, uint256 required);

contract TestToken {
    mapping(address => uint) balance;
    function transfer(address to, uint256 amount) public {
```

(续下页)

(接上页)

```
        if (amount > balance[msg.sender])
            revert InsufficientBalance({
                available: balance[msg.sender],
                required: amount
            });
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
    // ...
}
```

错误不能被重载或覆盖，但是可以被继承。只要作用域不同，同一个错误可以在多个地方定义。错误的实例只能使用 `revert` 语句创建。

错误会创建数据，然后通过还原操作传递给调用者，使其返回到链下组件或在*try/catch 语句* 中捕获它。需要注意的是，一个错误只能在来自外部调用时被捕获，发生在内部调用或同一函数内的还原不能被捕获。

如果您不提供任何参数，错误只需要四个字节的数据，您可以像上面一样使用*NatSpec 语法* 来进一步解释错误背后的原因，这并不存储在链上。这使得这同时也是一个非常便宜和方便的错误报告功能。

更具体地说，一个错误实例在被 ABI 编码时，其方式与对相同名称和类型的函数的调用相同，然后作为 `revert` 操作码的返回数据。这意味着数据由一个 4 字节的选择器和*ABI 编码* 数据组成。选择器由错误类型的签名的 keccak256-hash 的前四个字节组成。

备注： 一个合约有可能因为同名的不同错误而恢复，甚至因为在不同地方定义的错误而使调用者无法区分。对于外部来说，即 ABI，只有错误的名称是相关的，而不是定义它的合约或文件。

如果您能定义 `error Error(string)`，那么语句 `require(condition, "description");` 将等同于 `if (!condition) revert Error("description")`。但是请注意，`Error` 是一个内置类型，不能在用户提供的代码中定义。

同样，一个失败的 `assert` 或类似的条件将以一个内置的 `Panic(uint256)` 类型的错误来恢复。

备注： 错误数据应该只被用来指示失败，而不是作为控制流的手段。原因是内部调用的恢复数据默认是通过外部调用链传播回来的。这意味着内部调用可以”伪造”恢复数据，使它看起来像是来自调用它的合约。

错误类型的成员

- error.selector：一个包含错误类型的选择器的 bytes4 值。

3.9.8 继承

Solidity 支持多重继承，包括多态性。

多态性意味着函数调用（内部和外部）总是执行继承层次结构中最新继承的合约中的同名函数（和参数类型）。但必须使用 virtual 和 override 关键字在层次结构中的每个函数上明确启用。参见函数重载 以了解更多细节。

通过使用 ContractName.functionName() 明确指定合约，可以在内部调用继承层次结构中更高的函数。或者如果您想在扁平化的继承层次中调用高一级的函数（见下文），可以使用 super.functionName()。

当一个合约继承自其他合约时，在区块链上只创建一个单一的合约，所有基础合约的代码被编译到创建的合约中。这意味着对基础合约的所有内部函数的调用也只是使用内部函数调用（super.f(..) 将使用 JUMP 而不是消息调用）。

状态变量的阴影被认为是一个错误。一个派生合约只能声明一个状态变量 x，如果在它的任何基类中没有相同名称的可见状态变量。

总的来说，Solidity 的继承系统与 Python 的继承系统 非常相似，特别是关于多重继承方面，但也有一些不同之处。

详细情况见下面的例子。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告


contract Owned {
    constructor() { owner = payable(msg.sender); }
    address payable 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 Destructible is Owned {
    // 关键字 `virtual` 意味着该函数可以在派生类中改变其行为 ("重载")。
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}
```

(续下页)

(接上页)

```
// 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。
abstract contract Config {
    function lookup(uint id) public virtual returns (address adr);
}


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


// 多重继承是可能的。请注意， `Owned` 也是 `Destructible` 的基类，
// 但只有一个 `Owned` 实例（就像 C++ 中的虚拟继承）。
contract Named is Owned, Destructible {
    constructor(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。
    // 如果您想重载这个函数，您需要使用 `override` 关键字。
    // 如果您想让这个函数再次被重载，您需要再指定 `virtual` 关键字。
    function destroy() public virtual override {
        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。
            Destructible.destroy();
        }
    }
}


// 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）。
```

(续下页)

(接上页)

```
contract PriceFeed is Owned, Destructible, Named("GoldFeed") {
    function updateInfo(uint newInfo) public {
        if (msg.sender == owner) info = newInfo;
    }

    // 在这里，我们只指定了 `override` 而没有 `virtual`。
    // 这意味着从 `PriceFeed` 派生出来的合约不能再改变 `destroy` 的行为。
    function destroy() public override(Destructible, Named) { Named.destroy(); }
    function get() public view returns(uint r) { return info; }

    uint info;
}
```

注意，在上面，我们调用 `Destructible.destroy()` 来" 转发" 销毁请求。这样做的方式是有问题的，从下面的例子中可以看出：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告

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

contract Destructible is owned {
    function destroy() public virtual {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* 清除操作 1 */ Destructible.
↪destroy(); }
}

contract Base2 is Destructible {
    function destroy() public virtual override { /* 清除操作 2 */ Destructible.
↪destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { Base2.destroy(); }
}
```

调用 Final.destroy() 时会调用最后的派生重载函数 Base2.destroy，但是会绕过 Base1.destroy，解决这个问题的方法是使用 super：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 这将报告一个由于废弃的 selfdestruct 而产生的警告

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

contract Destructible is owned {
    function destroy() virtual public {
        if (msg.sender == owner) selfdestruct(owner);
    }
}

contract Base1 is Destructible {
    function destroy() public virtual override { /* 清除操作 1 */ super.destroy(); }
}


contract Base2 is Destructible {
    function destroy() public virtual override { /* 清除操作 2 */ super.destroy(); }
}

contract Final is Base1, Base2 {
    function destroy() public override(Base1, Base2) { super.destroy(); }
}
```

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上一个基类合约中调用这个函数，所以它会调用 Base1.destroy() （注意最终的继承序列是——从最远派生合约开始：Final, Base2, Base1, Destructible, 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。

函数重载

如果基函数被标记为 virtual，则可以通过继承合约来改变其行为。被重载的函数必须在函数头中使用 override 关键字。重载函数只能将被重载函数的可见性从 external 改为 public。可变性可以按照以下顺序改变为更严格的可变性。nonpayable 可以被 view 和 pure 重载。view 可以被 pure 重写。payable 是一个例外，不能被改变为任何其他可变性。

下面的例子演示了改变函数可变性和可见性：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base
{
    function foo() virtual external view {}
}

contract Middle is Base {}

contract Inherited is Middle
{
    function foo() override public pure {}
}
```

对于多重继承，必须在 override 关键字后明确指定定义同一函数的最多派生基类合约。换句话说，您必须指定所有定义同一函数的基类合约，并且还没有被另一个基类合约重载（在继承图的某个路径上）。此外，如果一个合约从多个（不相关的）基类合约上继承了同一个函数，必须明确地重载它。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    function foo() virtual public {}
}

contract Base2
{
    function foo() virtual public {}
}

contract Inherited is Base1, Base2
{
    // 派生自多个定义 foo() 函数的基类合约，
    // 所以我们必须明确地重载它
    function foo() public override(Base1, Base2) {}
}
```

如果函数被定义在一个共同的基类合约中，或者在一个共同的基类合约中有一个独特的函数已经重载了所有其他的函数，则不需要明确的函数重载指定符。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;
```

(续下页)

(接上页)

```
contract A { function f() public pure{} }
contract B is A {}
contract C is A {}
// 无需明确的重载
contract D is B, C {}
```

更准确地说，如果有一个基类合约是该签名的所有重载路径的一部分，并且（1）该基类合约实现了该函数，并且从当前合约到该基类合约的任何路径都没有提到具有该签名的函数，或者（2）该基类合约没有实现该函数，并且从当前合约到该基类合约的所有路径中最多只有一个提到该函数，那么就不需要重载从多个基类合约继承的函数（直接或间接）。

在这个意义上，一个签名的重载路径是一条继承图的路径，它从所考虑的合约开始，到提到具有该签名的函数的合约结束，而该签名没有重载。

如果您不把一个重载的函数标记为 `virtual`，派生合约就不能再改变该函数的行为。

备注： 具有 `private` 可见性的函数不能是 `virtual`。

备注： 在接口合约之外，没有实现的函数必须被标记为 `virtual`。在接口合约中，所有的函数都被自动视为 `virtual`。

备注： 从 Solidity 0.8.8 开始，当重载一个接口函数时，不需要 `override` 关键字，除非该函数被定义在多个基础上。

如果函数的参数和返回类型与变量的 getter 函数匹配，公共状态变量可以重载为外部函数。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract A
{
    function f() external view virtual returns(uint) { return 5; }
}

contract B is A
{
    uint public override f;
}
```

备注： 虽然公共状态变量可以重载外部函数，但它们本身不能被重载。

修饰器重载

函数修改器可以相互重载。这与函数重载 的工作方式相同（除了对修改器没有重载）。virtual 关键字必须用在被重载的修改器上，override 关键字必须用在重载的修改器上：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

在多重继承的情况下，必须明确指定所有的直接基类合约。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Base1
{
    modifier foo() virtual {_;}
}

contract Base2
{
    modifier foo() virtual {_;}
}

contract Inherited is Base1, Base2
{
    modifier foo() override(Base1, Base2) {_;}
}
```

构造函数

构造函数是一个用 constructor 关键字声明的可选函数，它在合约创建时被执行，您可以在这里运行合约初始化代码。

在构造函数代码执行之前，如果您用内联编程的方式初始化状态变量，则将其初始化为指定的值；如果您不用内联编程的方式来初始化，则将其初始化为默认值。

构造函数运行后，合约的最终代码被部署到区块链上。部署代码的 gas 花费与代码长度成线性关系。这段代码包括属于公共接口的所有函数，以及所有通过函数调用可以到达的函数。但不包括构造函数代码或只从构造函数中调用的内部函数。

如果没有构造函数，合约将假定默认的构造函数，相当于 constructor() {}。比如说：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

abstract contract A {
    uint public a;

    constructor(uint a_) {
        a = a_;
    }
}

contract B is A(1) {
    constructor() {}
}
```

您可以在构造函数中使用内部参数（例如，存储指针）。在这种情况下，合约必须被标记为*abstract*，因为这些参数不能从外部分配有效的值，只能通过派生合约的构造函数来赋值。

> **警告：** 在 0.4.22 版本之前，构造函数被定义为与合约同名的函数。这种语法已被废弃，在 0.5.0 版本中不再允许。

> **警告：** 在 0.7.0 版本之前，您必须指定构造函数的可见性为 internal 或 public。

基本构造函数的参数

所有基类合约的构造函数将按照下面解释的线性化规则被调用。如果基类合约构造函数有参数，派生合约需要指定所有的参数。这可以通过两种方式实现：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base {
    uint x;
    constructor(uint x_) { x = x_; }
}

// 要么直接在继承列表中指定...
contract Derived1 is Base(7) {
    constructor() {}
}

// 或者通过派生构造函数的一个 "修改器"……
contract Derived2 is Base {
    constructor(uint y) Base(y * y) {}
}

// 或者将合约声明为abstract类型……
abstract contract Derived3 is Base {
}

// 并让下一个具体的派生合约对其进行初始化。
contract DerivedFromDerived is Derived3 {
    constructor() Base(10 + 10) {}
}
```

一种方式是直接在继承列表中给出（`is Base(7)`）。另一种是通过修改器作为派生构造函数的一部分被调用的方式（`Base(_y * _y)`）。如果构造函数参数是一个常量，并且定义了合约的行为或描述了它，那么第一种方式更方便。如果基类合约的构造函数参数依赖于派生合约的参数，则必须使用第二种方式。参数必须在继承列表中或在派生构造函数中以修饰器的形式给出。在两个地方都指定参数是一个错误。

如果一个派生合约没有指定其所有基类合约的构造函数的参数，那么它必须被声明为 abstract 类型。在这种情况下，当另一个合约从它派生时，其他合约的继承列表或构造函数必须为所有没有指定参数的基类合约提供必要的参数（否则，其他合约也必须被声明为 abstract 类型）。例如，在上面的代码片段中，可以查看合约 `Derived3` 和 `DerivedFromDerived`。

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。这最终实现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，您必须按照从"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。注意，这个顺序与 Python 中使用的顺序相反。

另一种简化的解释方式是，当一个函数被调用时，它在不同的合约中被多次定义，给定的基类以深度优先的方式从右到左（Python 中从左到右）进行搜索，在第一个匹配处停止。如果一个基类合约已经被搜索过了，它就被跳过。

在下面的代码中，Solidity 会给出"Linearization of inheritance graph impossible"这样的错误。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract X {}
contract A is X {}
// 这段代码不会编译
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`，这是一种无法解决的冲突。

由于您必须明确地重载一个从多个基类合约继承的函数，而没有唯一的重载，C3 线性化在实践中不是太重要。

继承的线性化特别重要的一个领域是，当继承层次中存在多个构造函数时，也许不那么清楚。构造函数将总是按照线性化的顺序执行，而不管它们的参数在继承合约的构造函数中是以何种顺序提供的。比如说：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract Base1 {
    constructor() {}
}

contract Base2 {
    constructor() {}
}

// 构造函数按以下顺序执行：
//  1 - Base1
//  2 - Base2
//  3 - Derived1
```

(续下页)

(接上页)

```
contract Derived1 is Base1, Base2 {
    constructor() Base1() Base2() {}
}

// 构造函数按以下顺序执行:
//  1 - Base2
//  2 - Base1
//  3 - Derived2
contract Derived2 is Base2, Base1 {
    constructor() Base2() Base1() {}
}

// 构造函数仍按以下顺序执行:
//  1 - Base2
//  2 - Base1
//  3 - Derived3
contract Derived3 is Base2, Base1 {
    constructor() Base1() Base2() {}
}
```

继承有相同名字的不同类型成员

由于继承的原因，当合约有以下任何一对具有相同的名称时，这是一个错误：

- 函数和修饰器
- 函数和事件
- 事件和修饰器

有一种例外情况，状态变量的 getter 可以重载一个外部函数。

3.9.9 抽象合约

当合约中至少有一个函数没有被实现，或者合约没有为其所有的基本合约构造函数提供参数时，合约必须被标记为 abstract。即使不是这种情况，合约仍然可以被标记为 abstract，例如，当您不打算直接创建合约时。抽象（abstract）合约类似于接口（*interface*）合约，但是接口（interface）合约可以声明的内容更加有限。

如下例所示，使用 `abstract` 关键字来声明一个抽象合约。注意，这个合约需要被定义为 abstract，因为函数 `utterance()` 被声明了，但没有提供实现（没有给出实现体 `{ }`）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

```

(续下页)

(接上页)

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

这样的抽象合约不能被直接实例化。如果一个抽象合约本身实现了所有定义的功能，这也是可以的。抽象合约作为基类的用法在下面的例子中显示：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

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

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

如果一个合约继承自一个抽象合约，并且没有通过重写实现所有未实现的函数，那么它也需要被标记为抽象的。

注意，没有实现的函数与函数类型 不同，尽管它们的语法看起来非常相似。

没有实现内容的函数的例子（一个函数声明）：

```
function foo(address) external returns (address);
```

类型为函数类型的变量的声明实例：

```
function(address) external returns (address) foo;
```

抽象合约将合约的定义与它的实现解耦，提供了更好的可扩展性和自我记录，促进了像 模板方法 这样的模式，并消除了代码的重复。抽象合约的作用与在接口中定义方法的作用相同。它是抽象合约的设计者说" 我的任何孩子都必须实现这个方法" 的一种方式。

备注： 抽象合约不能用一个未实现的 virtual 函数来重载一个已实现的 virtual 函数。

3.9.10 接口（interface）合约

接口（interface）合约类似于抽象（abstract）合约，但是它们不能实现任何函数。并且还有进一步的限制：

- 它们不能继承其他合约，但是它们可以继承其他接口合约。
- 在接口合约中所有声明的函数必须是 external 类型的，即使它们在合约中是 public 类型的。
- 它们不能声明构造函数。
- 它们不能声明状态变量。
- 它们不能声明修饰器。

将来可能会解除这些里的某些限制。

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

接口合约由它们自己的关键字表示：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface Token {
    enum TokenType { Fungible, NonFungible }
    struct Coin { string obverse; string reverse; }
    function transfer(address recipient, uint amount) external;
}
```

就像继承其他合约一样，合约可以继承接口合约。

所有在接口合约中声明的函数都是隐式的 `virtual` 的类型，任何重载它们的函数都不需要 `override` 关键字。这并不自动意味着一个重载的函数可以被再次重载--这只有在重载的函数被标记为 `virtual` 时才可能。

接口合约可以从其他接口合约继承。这与普通的继承有着相同的规则。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

interface ParentA {
    function test() external returns (uint256);
}

interface ParentB {
    function test() external returns (uint256);
}

interface SubInterface is ParentA, ParentB {
    // 必须重新定义test，以便断言父类的含义是兼容的。
```

(续下页)

(接上页)

```
    function test() external override(ParentA, ParentB) returns (uint256);
}
```

在接口合约和其他类似合约的结构中定义的类型可以从其他合约中访问：`Token.TokenType` 或 `Token.Coin`。

3.9.11 库合约

库合约与普通合约类似，但是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。因为每个库合约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）。如果库函数不修改状态（也就是说，如果它们是 `view` 或者 `pure` 函数），它们可以通过直接调用来使用（即不使用 `DELEGATECALL` 关键字），这是因为我们假定库合约是无状态的。特别的是，销毁一个库合约是不可能的。

备注： 在 0.4.20 版本之前，有可能通过规避 Solidity 的类型系统来破坏库合约。从该版本开始，库合约包含一个保护机制，不允许直接调用修改状态的函数（即没有 `DELEGATECALL` ）。

库合约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` 是库合约的话，可以使用 `L.f()` 调用库函数）。当然，需要使用内部调用约定来调用内部函数，这意味着所有的内部类型都可以被传递，类型存储在内存将被引用传递而不是复制。为了在 EVM 中实现这一点，从合约中调用的内部库函数的代码和其中调用的所有函数将在编译时包含在调用合约中，并使用常规的 `JUMP` 调用，而不是 `DELEGATECALL`。

备注： 当涉及到公共函数时，继承的类比就失效了。用 `L.f()` 调用公共库函数的结果是一个外部调用（准确地说，是 `DELEGATECALL` ）。相反，当 `A.f()` 是当前合约的基类合约时，`A.f()` 是一个内部调用。

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）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;


// 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;
}
```

(续下页)

(接上页)

```
library Set {
    // 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 {
    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));
    }
```

(续下页)

(接上页)

```
    // 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，您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` 都被编译为对外部合约/库的调用（`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`)。

下面的例子显示了如何使用存储在内存中的类型 和库合约中的内部函数，以实现自定义类型，而没有外部函数调用的开销：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;

struct bigint {
    uint[] limbs;
}

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

    function add(bigint memory a, bigint memory b) internal pure returns (bigint␣
↪memory 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 limbA = limb(a, i);
            uint limbB = limb(b, i);
            unchecked {
                r.limbs[i] = limbA + limbB + carry;

                if (limbA + limbB < limbA || (limbA + limbB == type(uint).max &&␣
↪carry > 0))
                    carry = 1;
                else
                    carry = 0;
            }
        }
        if (carry > 0) {
```

(续下页)

(接上页)

```
            // 太差了，我们需要增加一个 limb
            uint[] memory newLimbs = new uint[](r.limbs.length + 1);
            uint i;
            for (i = 0; i < r.limbs.length; ++i)
                newLimbs[i] = r.limbs[i];
            newLimbs[i] = carry;
            r.limbs = newLimbs;
        }
    }

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

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

contract C {
    using BigInt for bigint;

    function f() public pure {
        bigint memory x = BigInt.fromUint(7);
        bigint memory y = BigInt.fromUint(type(uint).max);
        bigint memory z = x.add(y);
        assert(z.limb(1) > 0);
    }
}
```

通过将库合约的类型转换为 `address` 类型，即使用 `address(LibraryName)`，可以获得一个库的地址。

由于编译器不知道库合约的部署地址，编译后的十六进制代码将包含 `__$30bbc0abd4d6364515865950d3e0d10953$__` 形式的占位符。占位符是完全等同于库合约名的 keccak256 哈希值的 34 个字符的前缀，例如 `libraries/bigint.sol:BigInt`，如果该库存储在 `libraries/` 目录下一个名为 `bigint.sol` 的文件中。这样的字节码是不完整的，不应该被部署。占位符需要被替换成实际地址。您可以在编译库的时候把它们传递给编译器，或者用链接器来更新已经编译好的二进制文件。参见库链接，了解如何使用命令行编译器进行链接。

与合约相比，库在以下方面受到限制：

- 它们不能有状态变量
- 它们不能继承，也不能被继承
- 它们不能接收以太

- 它们不能被销毁

(这些可能会在以后的时间里被解除)。

库合约中的函数签名和选择器

虽然对公共或外部库函数的外部调用是可能的，但这种调用的调用惯例被认为是 Solidity 内部的，与常规合约 *ABI* 所指定的不一样。外部库函数比外部合约函数支持更多的参数类型，例如递归结构和存储指针。由于这个原因，用于计算 4 字节选择器的函数签名是按照内部命名模式计算的，合约 ABI 中不支持的类型的参数使用内部编码。

签名中的类型使用了以下标识符：

- 值类型、非存储的 `string` 和非存储的 `bytes` 使用与合约 ABI 中相同的标识符。
- 非存储数组类型遵循与合约 ABI 中相同的惯例，即 `<type>[]` 用于动态数组，`<type>[M]` 用于 `M` 元素的固定大小数组。
- 非存储结构体用其完全等同于的名称来指代，即 `C.S` 代表 `contract C { struct S { ... } }`。
- 存储指针映射使用 `mapping(<keyType> => <valueType>) storage`，其中 `<keyType>` 和 `<valueType>` 分别是映射的键和值类型的标识。
- 其他存储指针类型使用其对应的非存储类型的类型标识符，但在其后面附加一个空格，即 `storage`。

参数的编码与普通合约 ABI 相同，除了存储指针，它被编码为一个 `uint256` 值，指的是它们所指向的存储槽。

与合约 ABI 类似，选择器由签名的 Keccak256-hash 的前四个字节组成。它的值可以通过使用 `.selector` 成员从 Solidity 获得，如下：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.14 <0.9.0;

library L {
    function f(uint256) external {}
}

contract C {
    function g() public pure returns (bytes4) {
        return L.f.selector;
    }
}
```

库的调用保护

正如介绍中提到的那样，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恢复，除非是对 `view` 或者 `pure` 函数的调用。

EVM 没有提供一个直接的方法让合约检测它是否被使用 `CALL` 调用，但是合约可以使用 `ADDRESS` 操作码来找出它当前运行的" 位置"。生成的代码将这个地址与构造时使用的地址进行比较，以确定调用的模式。

更具体地说，一个库合约的运行时代码总是以 push 指令开始，在编译时它是一个 20 字节的零。当部署代码运行时，这个常数在内存中被当前地址所取代，这个修改后的代码被存储在合约中。在运行时，这导致部署时的地址成为第一个被推入堆栈的常数，对于任何非-view 和非-pure 函数，调度器代码会将当前地址与这个常数进行比较。

这意味着一个存储在链上的库合约的实际代码，与编译器报告的 `deployedBytecode` 的代码不同。

3.9.12 Using For

指令 `using A for B` 可用于将函数（`A`）作为运算符附加到用户定义的值类型或作为成员函数附加到任何类型（`B`）。成员函数将调用它们的对象作为第一个参数（类似于 Python 中的 `self` 变量）。运算符函数将接收操作数作为参数。

它可以在文件级别或者在合约级别的合约内部有效。

第一部分，`A`，可以是以下之一：

- 一个函数列表，可选择分配运算符名称（例如 `using {f, g as +, h, L.t} for uint`）。如果未指定运算符，则该函数可以是库函数或自由函数，并将其作为成员函数附加到类型。否则，它必须是一个自由函数，并成为该类型上该运算符的定义。
- 一个库合约的名称（例如 `using L for uint`）- 该库合约的所有非私有函数都作为成员函数附加到该类型上。

在文件级别中，第二部分，`B`，必须是一个明确的类型（没有数据位置指定）。在合约内部，您也可以用 `*` 代替类型（例如 `using L for *;`），这样做的效果是，库合约 `L` 中所有的函数都会被附加到 所有类型上。

如果您指定了一个库合约，那么该库合约中的 所有非私有函数都会被附加到该类型上，即使是那些第一个参数的类型与对象的类型不匹配的函数。类型会在函数被调用的时候检查，并执行函数重载解析。

如果您使用一个函数列表（例如 `using {f, g, h, L.t} for uint`），那么类型（`uint`）必须可以隐式地转换为这些函数的第一个参数。即使这些函数都没有被调用，也要进行这种检查。请注意，只有当 `using for` 位于库合约内时，才能指定私有库函数。

如果您定义了一个操作符（例如 `using {f as +} for T`），那么类型（`T`）必须是一个用户定义的值类型，并且定义必须是一个 `pure` 函数。操作符定义必须是全局的。以下操作符可以用这种方式定义：

Category	Operator	Possible signatures
Bitwise	&	function (T, T) pure returns (T)
	|	function (T, T) pure returns (T)
	^	function (T, T) pure returns (T)
	~	function (T) pure returns (T)
Arithmetic	+	function (T, T) pure returns (T)
	-	function (T, T) pure returns (T)
		function (T) pure returns (T)
	*	function (T, T) pure returns (T)
	/	function (T, T) pure returns (T)
	%	function (T, T) pure returns (T)
Comparison	==	function (T, T) pure returns (bool)
	!=	function (T, T) pure returns (bool)
	<	function (T, T) pure returns (bool)
	<=	function (T, T) pure returns (bool)
	>	function (T, T) pure returns (bool)
	>=	function (T, T) pure returns (bool)

注意，一元和二元的 - 需要单独定义。编译器会根据操作符的调用方式选择正确的定义。

`using A for B;` 指令只在当前作用域（合约或当前模块/源单元）内有效，包括其中所有的函数，在使用它的合约或模块之外没有任何效果。

当在文件级别使用该指令并应用于在同一文件中用户定义类型时，可以在末尾添加 `global` 关键字。这将使函数和操作符附加到该类型的任何可用位置（包括其他文件），而不仅仅是在 using 语句的范围内。

下面我们将使用文件级函数来重写库合约 部分中的 set 示例。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

struct Data { mapping(uint => bool) flags; }
// 现在我们给这个类型附加上函数。
// 附加的函数可以在模块的其他部分使用。
// 如果您导入了该模块，
// 您必须在那里重复using指令，例如
//   import "flags.sol" as Flags;
//   using {Flags.insert, Flags.remove, Flags.contains}
//     for Flags.Data;
using {insert, remove, contains} for Data;

function insert(Data storage self, uint value)
    returns (bool)
```

(续下页)

(接上页)

```
{
    if (self.flags[value])
        return false; // 已经存在
    self.flags[value] = true;
    return true;
}

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

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


contract C {
    Data knownValues;

    function register(uint value) public {
        // 这里， 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));
    }
}
```

也可以通过这种方式来扩展内置类型。在这个例子中，我们将使用一个库合约。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.13;

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

(续下页)

(接上页)

```
    {
        for (uint i = 0; i < self.length; i++)
            if (self[i] == value) return i;
        return type(uint).max;
    }
}
using Search for uint[];

contract C {
    uint[] data;

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

    function replace(uint from, uint to) public {
        // 这将执行库合约中的函数调用
        uint index = data.indexOf(from);
        if (index == type(uint).max)
            data.push(to);
        else
            data[index] = to;
    }
}
```

注意，所有的外部库调用实际都是 EVM 函数调用。这意味着，如果传递内存或值类型，即使是 `self` 变量，也会执行复制。只有在使用存储引用变量或调用内部库函数时，才不会执行复制。

另一个展示了如何为用户定义的类型定义自定义操作符的示例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.19;

type UFixed16x2 is uint16;

using {
    add as +,
    div as /
} for UFixed16x2 global;

uint32 constant SCALE = 100;

function add(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    return UFixed16x2.wrap(UFixed16x2.unwrap(a) + UFixed16x2.unwrap(b));
```

(续下页)

(接上页)

```
}

function div(UFixed16x2 a, UFixed16x2 b) pure returns (UFixed16x2) {
    uint32 a32 = UFixed16x2.unwrap(a);
    uint32 b32 = UFixed16x2.unwrap(b);
    uint32 result32 = a32 * SCALE / b32;
    require(result32 <= type(uint16).max, "Divide overflow");
    return UFixed16x2.wrap(uint16(a32 * SCALE / b32));
}

contract Math {
    function avg(UFixed16x2 a, UFixed16x2 b) public pure returns (UFixed16x2) {
        return (a + b) / UFixed16x2.wrap(200);
    }
}
```

3.10 内联汇编

您可以用接近 Ethereum 虚拟机的语言，将 Solidity 语句与内联汇编交错使用。这给了您更精细的控制，这在您通过编写库来增强语言时特别有用。

在 Solidity 中用于内联汇编的语言被称为*Yul*，它在自己的章节中被记录。本节将只涉及内联汇编代码如何在 Solidity 代码内交互。

> **警告：** 内联汇编是一种在低等级上访问 Ethereum 虚拟机的方式。这绕过了 Solidity 的几个重要安全功能和检查。您应该只在需要它的任务中使用它，而且只有在您对使用它有信心的情况下。

一个内联汇编块由 `assembly { ... }` 标记的，其中大括号内的代码是*Yul* 语言中的代码。

内联汇编代码可以访问本地 Solidity 变量，如下所述。

不同的内联汇编块不共享名称空间，即不能调用或访问一个在不同内联汇编块中定义的 Yul 函数或变量。

3.10.1 例子

下面例子展示了一个库合约的代码，它可以取得另一个合约的代码，并将其加载到一个 `bytes` 变量中。通过使用 `<address>.code`，这在" 普通 Solidity" 中也是可能的。但这里的重点是，可重用的汇编库可以增强 Solidity 语言，而不需要改变编译器。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;
```

(续下页)

(接上页)

```
library GetCode {
    function at(address addr) public view returns (bytes memory code) {
        assembly {
            // 获取代码大小，这需要汇编语言
            let size := extcodesize(addr)
            // 分配输出字节数组 - 这也可以不用汇编语言来实现
            // 通过使用 code = new bytes(size)
            code := mload(0x40)
            // 包括补位在内新的 “memory end”
            mstore(0x40, add(code, and(add(add(size, 0x20), 0x1f), not(0x1f))))
            // 把长度保存到内存中
            mstore(code, size)
            // 实际获取代码，这需要汇编语言
            extcodecopy(addr, add(code, 0x20), 0, size)
        }
    }
}
```

在优化器不能产生高效代码的情况下，内联汇编也是有益的，例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;


library VectorSum {
    // 这个函数的执行效率比较低，
    // 因为优化器当前无法删除数组访问中的边界检查。
    function sumSolidity(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i)
            sum += data[i];
    }

    // 我们知道我们只能在数组范围内访问数组元素，所以我们可以避免检查。
    // 由于 ABI 编码中数组数据的第一个字（32 字节）的位置保存的是数组长度，
    // 所以我们在访问数组元素时需要加入 0x20 作为偏移量。
    function sumAsm(uint[] memory data) public pure returns (uint sum) {
        for (uint i = 0; i < data.length; ++i) {
            assembly {
                sum := add(sum, mload(add(add(data, 0x20), mul(i, 0x20))))
            }
        }
    }
```

(续下页)

(接上页)

```
    // 和上面一样，但在内联汇编内完成整个代码。
    function sumPureAsm(uint[] memory data) public pure returns (uint sum) {
        assembly {
            // 加载数组长度 (前 32 字节)
            let len := mload(data)

            // 略过长度字段。
            //
            // 保持临时变量以便它可以在原地增加。
            //
            // 注意：递增data会导致在这个汇编块之后出现一个无法使用的data变量。
            let dataElementLocation := add(data, 0x20)

            // 迭代到数组数据结束。
            for
                { let end := add(dataElementLocation, mul(len, 0x20)) }
                lt(dataElementLocation, end)
                { dataElementLocation := add(dataElementLocation, 0x20) }
            {
                sum := add(sum, mload(dataElementLocation))
            }
        }
    }
}
```

3.10.2 访问外部变量、函数和库

您可以通过使用其名称来访问 Solidity 变量和其他标识符。

值类型的局部变量可以直接用于内联汇编。它们既可以被读取也可以被赋值。

指向内存的局部变量是指内存中变量的地址，而不是值本身。这样的变量也可以被赋值，但请注意，赋值只会改变指针而不是数据，尊重 Solidity 的内存管理是您的责任。参见*Solidity 的惯例*。

同样地，引用静态大小的 calldata 数组或 calldata 结构的局部变量会指向 calldata 中变量的地址，而不是值本身。变量也可以被分配一个新的偏移量，但是请注意，没有进行验证以确保变量不会指向超过 `calldatasize()` 的地方。

对于外部函数指针，地址和函数选择器可以用 `x.address` 和 `x.selector` 来访问。选择器由四个右对齐的字节组成。两个值都可以被赋值。比如说：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.10 <0.9.0;

```

(续下页)

(接上页)

```
contract C {
    // 将一个新的选择器和地址分配给返回变量 @fun
    function combineToFunctionPointer(address newAddress, uint newSelector) public␣
↪pure returns (function() external fun) {
        assembly {
            fun.selector := newSelector
            fun.address  := newAddress
        }
    }
}
```

对于动态的 calldata 数组，您可以使用 `x.offset` 和 `x.length` 访问它们的 calldata 偏移量（字节）和长度（元素数）。这两个表达式也可以被赋值，但是和静态情况一样，不会进行验证以确保产生的数据区域在 `calldatasize()` 的范围内。

对于本地存储变量或状态变量，一个 Yul 标识符是不够的，因为它们不一定占据一个完整的存储槽。因此，它们的" 地址" 是由一个槽和槽内的字节偏移量组成。要检索变量 `x` 所指向的槽，您可以使用 `x.slot`，要检索字节偏移量，您可以使用 `x.offset` 。使用 `x` 本身会导致错误。

您也可以分配给本地存储变量指针的 `.slot` 部分。对于这些（结构、数组或映射），`.offset` 部分总是零。但不可能分配给状态变量的 `.slot` 或 `.offset` 部分。

本地 Solidity 变量可用于赋值，例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract C {
    uint b;
    function f(uint x) public view returns (uint r) {
        assembly {
            // 我们忽略了存储槽的偏移量，我们知道在这种特殊情况下它是零。
            r := mul(x, sload(b.slot))
        }
    }
}
```

警告： 如果您访问一个跨度小于 256 位的类型的变量（例如 `uint64`，`address`，或 `bytes16`），您不能对不属于该类型的编码的位做任何假设。特别是，不要假设它们是零。为了安全起见，在使用前一定要适当清除数据，因为这一点很重要：`uint32 x = f(); assembly { x := and(x, 0xffffff) /* 现在使用 x */ }` 为了清除有符号的类型，您可以使用 `signextend` 操作码。`assembly { signextend(<num_bytes_of_x_minus_one>, x) }`

自 Solidity 0.6.0 以来，内联汇编变量的名称不能影射内联汇编块范围内可见的任何声明（包括变量、合约和函数声明）。

自 Solidity 0.7.0 以来，在内联程序块内声明的变量和函数不能包含 `.`，但使用 `.` 可以有效地从内联程序块外访问 Solidity 变量。

3.10.3 需要避免的事情

内联汇编可能有一个相当高级的外观，但它实际上是非常低级的。函数调用、循环、if 条件和 switch 条件都可以通过简单的改写规则进行转换，之后，汇编器为您做的唯一事情就是重新安排函数式的操作码，为变量访问计算堆栈高度，并在达到汇编局部变量块的末端时移除堆栈槽。

3.10.4 Solidity 的的惯例

类型化变量的值

与 EVM 汇编相反，Solidity 有比 256 位更窄的类型，例如：`uint24`。为了提高效率，大多数算术运算忽略了类型可以短于 256 位的事实，高阶位在必要时被清理，例如，在它们被写入内存前不久或在执行比较之前。这意味着，如果您从内联汇编中访问这样的变量，您可能不得不先手动清理高阶位。

内存管理

Solidity 以下列方式管理内存。在内存中 `0x40` 的位置有一个" 空闲内存指针"。如果您想分配内存，从这个指针指向的地方开始使用内存，并更新它。不能保证该内存以前没有被使用过，因此您不能假设其内容为零字节。没有内置的机制来释放或释放分配的内存。下面是一段汇编代码，您可以用它来分配内存，它遵循上述的过程：

```
function allocate(length) -> pos {
  pos := mload(0x40)
  mstore(0x40, add(pos, length))
}
```

前 64 字节的内存可以作为短期分配的" 划痕空间（scratch space）"。空闲内存指针之后的 32 字节（即从 `0x60` 开始）是指永久为零，并作为空的动态内存数组的初始值使用。这意味着可分配的内存从 `0x80` 开始，也就是空闲内存指针的初始值。

Solidity 中内存数组中的元素总是占据 32 字节的倍数（对于 `bytes1[]` 来说也是如此，但对于 `bytes` 和 `string` 来说不是这样）。多维内存数组是指向内存数组的指针。一个动态数组的长度被存储在数组的第一个槽里，后面是数组元素。

> **警告：** 静态大小的内存数组没有长度字段，但以后可能会加入长度字段，以便在静态大小的数组和动态大小的数组之间有更好的转换性；所以，不要依赖这个特性。

内存安全

在不使用内联汇编的情况下，编译器可以依靠内存在任何时候都保持一个良好的定义状态。这对于通过 *Yul IR* 的新代码生成管道 来说尤其重要：这个代码生成路径可以将局部变量从堆栈转移到内存，以避免堆栈过深的错误，并执行额外的内存优化，如果它可以依赖于对内存使用的某些假设的话。

虽然我们建议始终尊重 Solidity 的内存模型，但内联汇编允许您以不兼容的方式使用内存。因此，在任何包含内存操作或在内存中分配给 Solidity 变量的内联汇编块存在的情况下，将堆栈变量移动到内存和额外的内存优化默认为全局禁用。

然而，您可以特别注释一个汇编块，以表明它实际上是遵循 Solidity 内存模型的，如下所示：

```
assembly ("memory-safe") {
    ...
}
```

特别的是，一个内存安全的汇编块只能访问以下内存范围：

- 通过上述 `allocate` 函数这样的机制由自己分配的内存。
- 由 Solidity 分配的内存，例如，在您引用的内存数组的范围内的内存。
- 上面提到的内存偏移量 0 和 64 之间的划痕空间。
- 位于汇编块开始时的空闲内存指针值 *之后* 的临时内存，即在空闲内存指针上 "分配" 而不更新空闲内存指针的内存。

此外，如果汇编块分配给内存中的 Solidity 变量，则需要确保对 Solidity 变量的访问只访问这些内存范围。

由于这主要是关于优化器的，所以这些限制仍然需要被遵守，即使汇编块回退或终止。举个例子，下面的汇编片段不是内存安全的，因为 `returndatasize()` 的值可能会超过 64 字节的划痕空间。

```
assembly {
  returndatacopy(0, 0, returndatasize())
  revert(0, returndatasize())
}
```

另一方面，下面的代码 *是* 内存安全的，因为超出空闲内存指针所指向的位置的内存可以安全地用作临时划痕空间：

```
assembly ("memory-safe") {
  let p := mload(0x40)
  returndatacopy(p, 0, returndatasize())
  revert(p, returndatasize())
}
```

注意，如果没有后续分配，则不需要更新空闲内存指针，但只能使用从空闲内存指针给出的当前偏移量开始的内存。

如果内存操作使用的长度为 0，那么也可以使用任意偏移量（不仅仅是落在了划痕空间中）：

```
assembly ("memory-safe") {
  revert(0, 0)
}
```

请注意，不仅内联汇编中的内存操作本身可以是内存不安全的，而且对内存中引用类型的 Solidity 变量的赋值也是如此。例如，以下内容就不是内存安全的：

```
bytes memory x;
assembly {
  x := 0x40
}
x[0x20] = 0x42;
```

既不涉及访问内存的任何操作，也不对内存中的任何 Solidity 变量进行赋值的内联汇编，自动被认为是内存安全的，不需要被注释。

> **警告：** 确保汇编块程序实际满足内存模型是您的责任。如果您将一个汇编块注释为内存安全的，但却违反了其中一个内存假设，那么这 **将**导致不正确的和未定义的行为，而这些行为不容易通过测试发现。

如果您正在开发一个要在多个 Solidity 版本之间兼容的库，您可以使用一个特殊的注释将一个汇编块注释为内存安全的：

```
/// @solidity memory-safe-assembly
assembly {
    ...
}
```

请注意，我们将在未来的突破性版本中不允许通过注释的方式进行注解；因此，如果您不关心与旧编译器版本的向后兼容问题，最好使用这种写法的代码字符串形式。

3.11 速查表

3.11.1 操作符的优先顺序

以下是按评估顺序列出的操作符优先级。

优先级	描述	操作符
1	后置自增和自减	++, --
	创建类型实例	new < 类型名>
	数组元素	< 数组>[< 索引>]
	访问成员	< 对象>.< 成员名>
	函数调用	< 函数>(< 参数...>)
	小括号	(< 表达式>)
2	前置自增和自减	++, --
	一元运算减	-
	一元操作符	delete
	逻辑非	!
	按位非	~
3	乘方	**
4	乘、除和模运算	*, /, %
5	算术加和减	+, -
6	移位操作符	<<, >>
7	按位与	&
8	按位异或	^
9	按位或	|
10	非等操作符	<, >, <=, >=
11	等于操作符	==, !=
12	逻辑与	&&
13	逻辑或	==
14	三元操作符	< 判断条件> ? < 如果为真时执行的表达式> : < 如果为假时执行的表达式>
	赋值操作符	=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=
15	逗号	,

3.11.2 ABI 编码和解码函数

- `abi.decode(bytes memory encodedData, (...)) returns (...)`: *ABI* - 对提供的数据进行解码。类型在括号中作为第二个参数给出。示例：`(uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes))`
- `abi.encode(...) returns (bytes memory)`: *ABI* - 对给定的参数进行编码。
- `abi.encodePacked(...) returns (bytes memory)`: 对给定的参数执行紧密打包。请注意，这种编码可能是不明确的！

- abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory): *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.encodeCall(function functionPointer, (...)) returns (bytes memory): 对 functionPointer 的调用进行 ABI 编码，参数在元组中找到。执行全面的类型检查，确保类型与函数签名相符。结果等于 abi.encodeWithSelector(functionPointer.selector(..))。
- abi.encodeWithSignature(string memory signature, ...) returns (bytes memory): 等价于 abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)

3.11.3 bytes 和 string 类型的属性

- bytes.concat(...) returns (bytes memory): 将可变数量的参数连接成一个字节数组
- string.concat(...) returns (string memory): 将可变数量的参数连接成一个字符串数组

3.11.4 address 类型的属性

- <address>.balance (uint256): 地址类型 的余额，以 Wei 为单位
- <address>.code (bytes memory): 在地址类型 的代码（可以是空的）
- <address>.codehash (bytes32): 地址类型 的代码哈希值
- <address payable>.send(uint256 amount) returns (bool): 向地址类型 发送给定数量的 Wei，失败时返回 false
- <address payable>.transfer(uint256 amount): 向地址类型 发送给定数量的 Wei，失败时会抛出错误

3.11.5 区块和交易属性

- blockhash(uint blockNumber) returns (bytes32): 给定区块的哈希值 - 只对最近的 256 个区块有效
- block.basefee (uint): 当前区块的基本费用（EIP-3198 和 EIP-1559 ）
- block.chainid (uint): 当前链的 ID
- block.coinbase (address payable): 当前区块矿工的地址
- block.difficulty (uint): 当前区块的难度值（EVM < Paris ）。对于其他 EVM 版本，它是 block.prevrandao 的一个废弃的别名，将在下一个重大改变版本中被删除。
- block.gaslimit (uint): 当前区块的 gas 上限
- block.number (uint): 当前区块的区块号

- `block.prevrandao` (`uint`): 由信标链提供的随机数（`EVM >= Paris`）（见 EIP-4399 ）。
- `block.timestamp` (`uint`): 当前区块的时间戳，自 Unix epoch 以来的秒数
- `gasleft() returns (uint256)`: 剩余 gas
- `msg.data` (`bytes`): 完整的调用数据
- `msg.sender` (`address`): 消息发送方（当前调用）
- `msg.sig` (`bytes4`): Calldata 的前四个字节（即函数标识符）。
- `msg.value` (`uint`): 随消息发送的 wei 的数量
- `tx.gasprice` (`uint`): 交易的 gas 价格
- `tx.origin` (`address`): 交易发送方（完整调用链上的原始发送方）

3.11.6 验证和断言

- `assert(bool condition)`: 如果条件为 `false`，则中止执行并恢复状态变化（用于内部错误）
- `require(bool condition)`: 如果条件为 `false`，则中止执行并恢复状态变化（用于错误的输入或外部组件的错误）
- `require(bool condition, string memory message)`: 如果条件为 `false`，则中止执行并恢复状态变化（用于错误的输入或外部组件的错误）。同时提供错误信息。
- `revert()`: 中止执行并恢复状态变化
- `revert(string memory message)`: 中止执行并恢复状态变化，提供一个解释性的字符串

3.11.7 数学和加密函数

- `keccak256(bytes memory) returns (bytes32)`: 计算输入的 Keccak-256 哈希值
- `sha256(bytes memory) returns (bytes32)`: 计算输入的 SHA-256 哈希值
- `ripemd160(bytes memory) returns (bytes20)`: 计算输入的 RIPEMD-160 的哈希值
- `ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`: 从椭圆曲线签名中恢复与公钥相关的地址，错误时返回 0
- `addmod(uint x, uint y, uint k) returns (uint)`: 计算 `(x + y) % k` 的值，其中加法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本开始会加入对 `k != 0` 的 assert（即会在此函数开头执行 `assert(k != 0);` 作为参数检查，译者注）。
- `mulmod(uint x, uint y, uint k) returns (uint)`: 计算 `(x * y) % k` 的值，其中乘法的结果即使超过 `2**256` 也不会被截取。从 0.5.0 版本开始会加入对 `k != 0` 的 assert（即会在此函数开头执行 `assert(k != 0);` 作为参数检查，译者注）。

3.11.8 合约相关函数

- `this` （当前合约的类型）：当前合约，可明确转换为 `address` 或 `address payable`
- `super`: 继承层次中高一级的合约
- `selfdestruct(address payable recipient)`: 销毁当前合约，将其资金发送到给定的地址

3.11.9 类型属性

- `type(C).name` (`string`): 合约的名称
- `type(C).creationCode` (`bytes memory`): 给定合约的创建字节码，参见类型信息。
- `type(C).runtimeCode` (`bytes memory`): 给定合约的运行时字节码，参见类型信息。
- `type(I).interfaceId` (`bytes4`): 包含给定接口的 EIP-165 接口标识符的值，参见类型信息。
- `type(T).min` (`T`): 整数类型 `T` 所能代表的最小值，参见类型信息。
- `type(T).max` (`T`): 整数类型 `T` 所能代表的最大值，参见类型信息。

3.11.10 函数可见性说明符

```
function myFunction() <visibility specifier> returns (bool) {
    return true;
}
```

- `public`：内部、外部均可见（参考为存储/状态变量创建*getter function* 函数）
- `private`：仅在当前合约内可见
- `external`：仅在外部可见（仅可修饰函数）——就是说，仅可用于消息调用（即使在合约内调用，也只能通过 `this.func` 的方式）
- `internal`：仅在内部可见（也就是在当前 Solidity 源代码文件内均可见，不仅限于当前合约内，译者注）

3.11.11 修改器

- `pure` 修饰函数时：不允许修改或访问状态。
- `view` 修饰函数时：不允许修改状态。
- `payable` 修饰函数时：允许从调用中接收以太币。
- `constant` 修饰状态变量时：不允许赋值（除初始化以外），不会占据存储插槽（storage slot）。
- `immutable` 修饰状态变量时：在构造时允许有一个确切的赋值，之后是恒定的。被存储在代码中。

- `anonymous` 修饰事件时：不把事件签名作为 topic 存储。
- `indexed` 修饰事件参数时：将参数作为 topic 存储。
- `virtual` 修饰函数和修改时：允许在派生合约中改变函数或修改器的行为。
- `override` 表示该函数、修改器或公共状态变量改变了基类合约中的函数或修改器的行为。

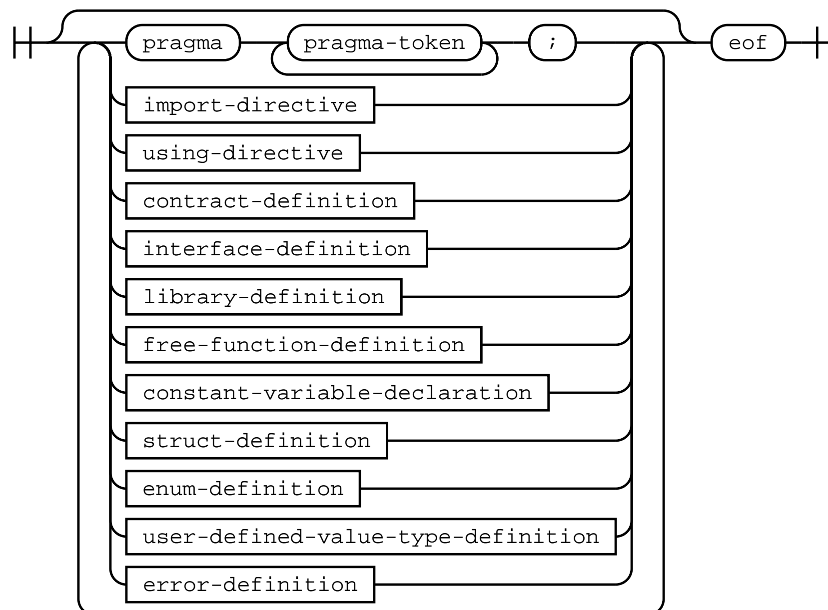
3.12 语法

parser grammar SolidityParser

Solidity 是一种静态类型的，面向合约的高级语言，用于在 Ethereum 平台上实现智能合约。

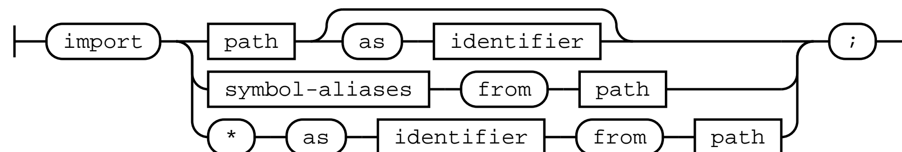
rule source-unit

在顶层，Solidity 允许 pragmas，导入语句，以及合约，接口，库，结构，枚举和常量的定义。



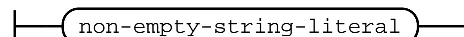
rule import-directive

导入指令从不同的文件中导入标识符。



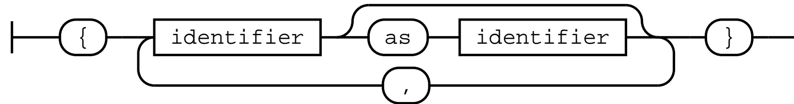
rule path

要导入的文件的路径。



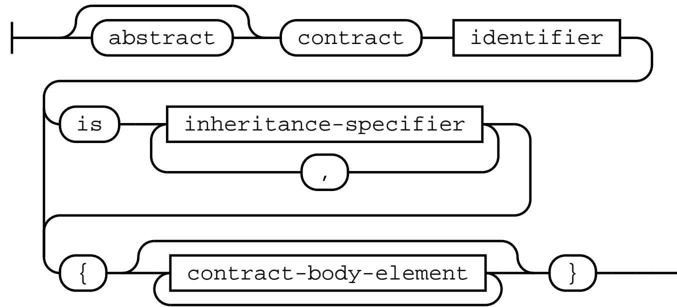
rule symbol-aliases

要导入的符号的别名列表。



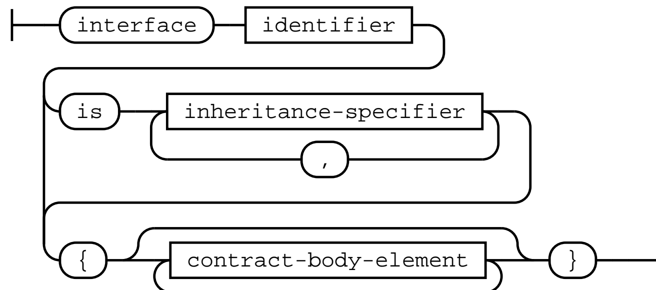
rule contract-definition

合约的顶层定义。



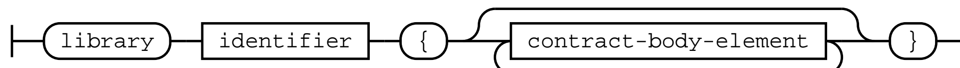
rule interface-definition

接口的顶层定义。



rule library-definition

一个库合约的顶层定义。



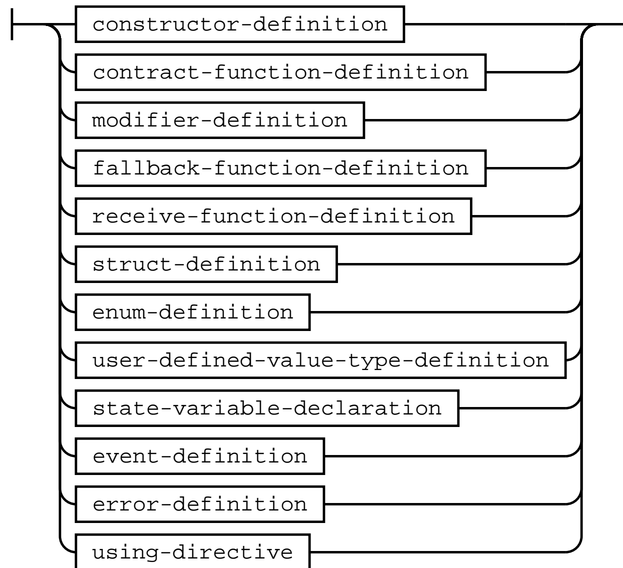
rule inheritance-specifier

合约和接口的继承指定器。可以有选择地提供基本构造函数参数。

rule contract-body-element

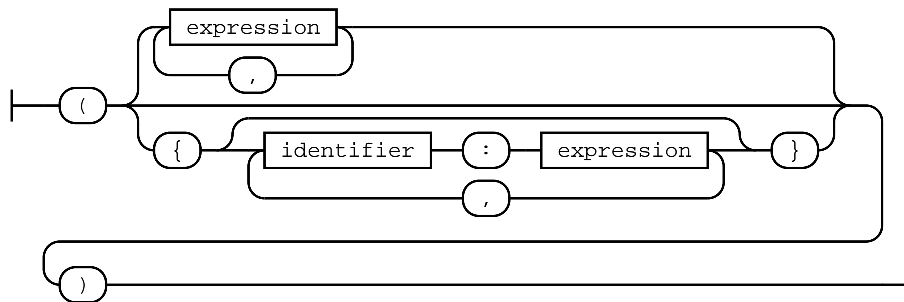
可以在合约，接口和库中使用的声明。

注意，接口和库不能包含构造函数，接口不能包含状态变量，库不能包含 fallback，receive 函数和非恒定状态变量。



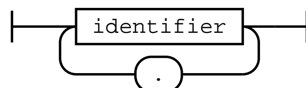
rule call-argument-list

调用一个函数或类似的可调用对象时的参数。参数要么以逗号分隔的列表形式给出，要么以命名参数的映射形式给出。



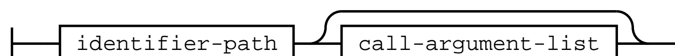
rule identifier-path

合格的名称。



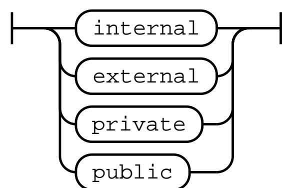
rule modifier-invocation

对一个修改器的调用。如果修改器不需要参数，参数列表可以完全跳过（包括开头和结尾的括号）。



rule visibility

函数和函数类型的可见性。

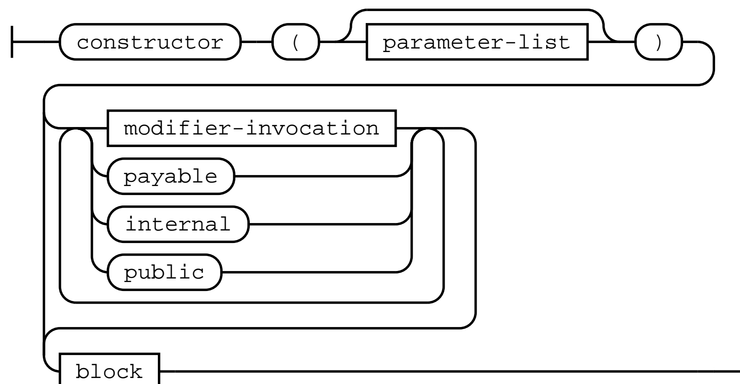


rule parameter-list

一个参数的列表，如函数参数或返回值。

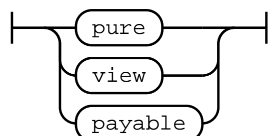
rule constructor-definition

一个构造函数的定义。必须始终提供一个实现。请注意，指定内部或公共可见性已被废弃。



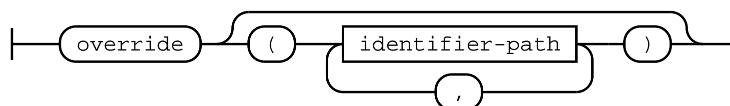
rule state-mutability

函数类型的状态可变性。如果没有指定可变性，则假定默认的可变性为"非 payable"。



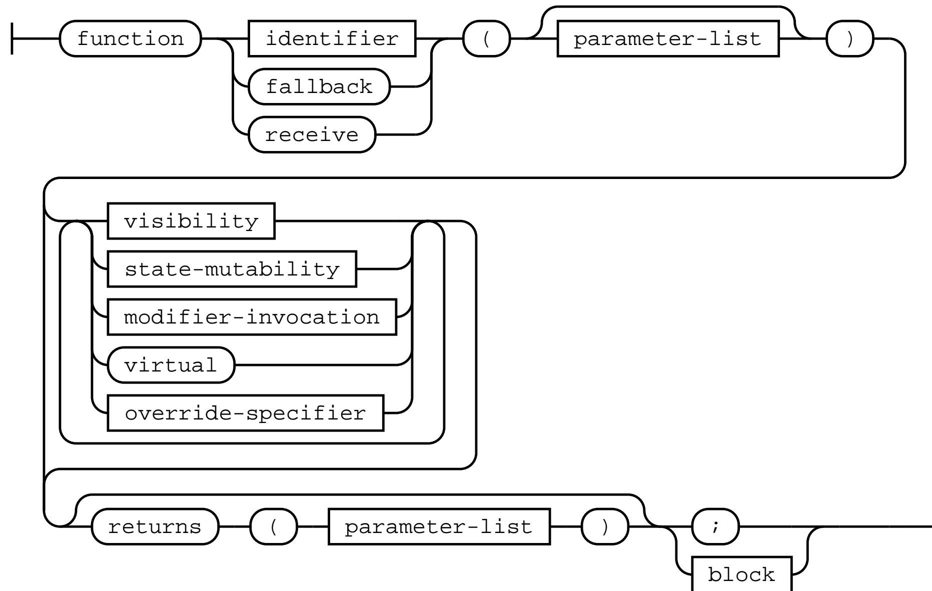
rule override-specifier

一个用于函数，修改器或状态变量的重载指定符。
如果在被重载的几个基础合约中存在不明确的声明，必须给出一个完整的基础合约清单。



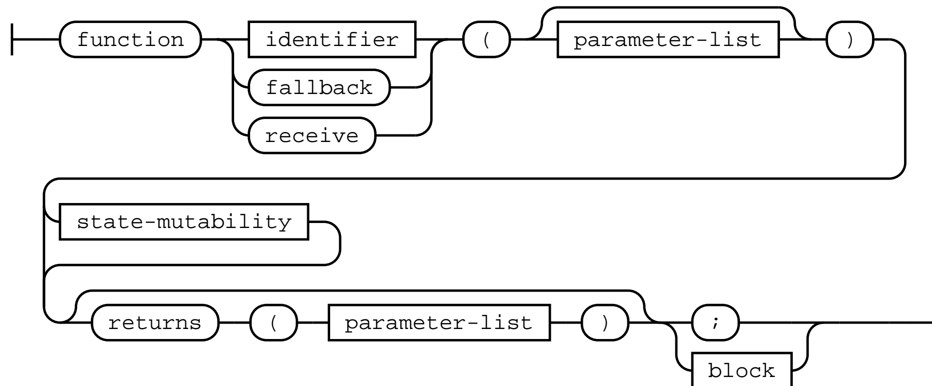
rule contract-function-definition

合约，库和接口功能的定义。根据定义函数的上下文，可能会有进一步的限制。例如，接口中的函数必须是未实现的，也就是说，不能包含主体块。



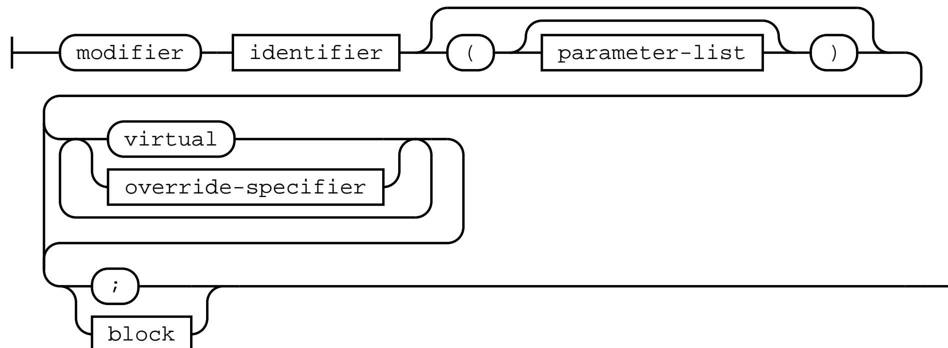
rule free-function-definition

自由函数的定义。



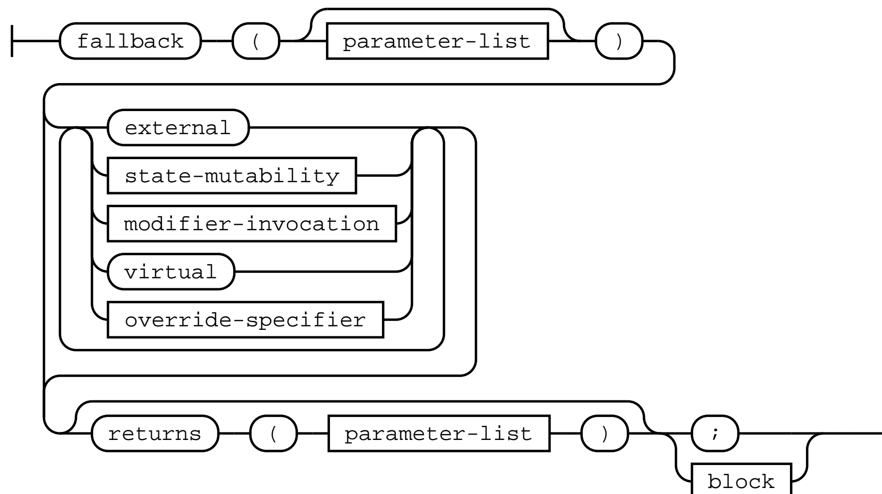
rule modifier-definition

修改器的定义。注意，在修改器的主体块中，下划线不能作为标识符使用，而是作为占位符语句，用于修改器所应用的函数主体。



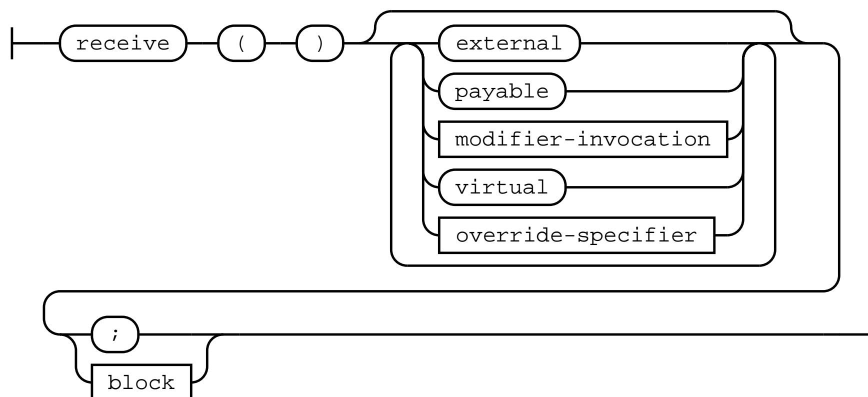
rule fallback-function-definition

特殊的 fallback 函数的定义。



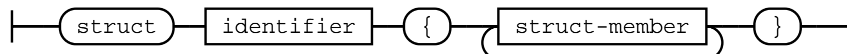
rule receive-function-definition

特殊的 receive 函数的定义。



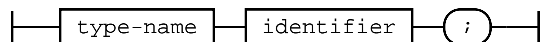
rule struct-definition

结构体的定义。可以出现在源代码单元的顶层，也可以出现在合约，库或接口中。



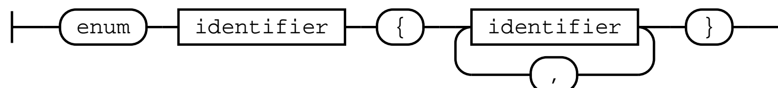
rule struct-member

一个命名的结构体成员的声明。



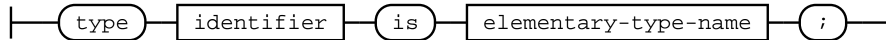
rule enum-definition

一个枚举的定义。可以出现在源代码单元的顶层，也可以出现在合约，库或接口中。



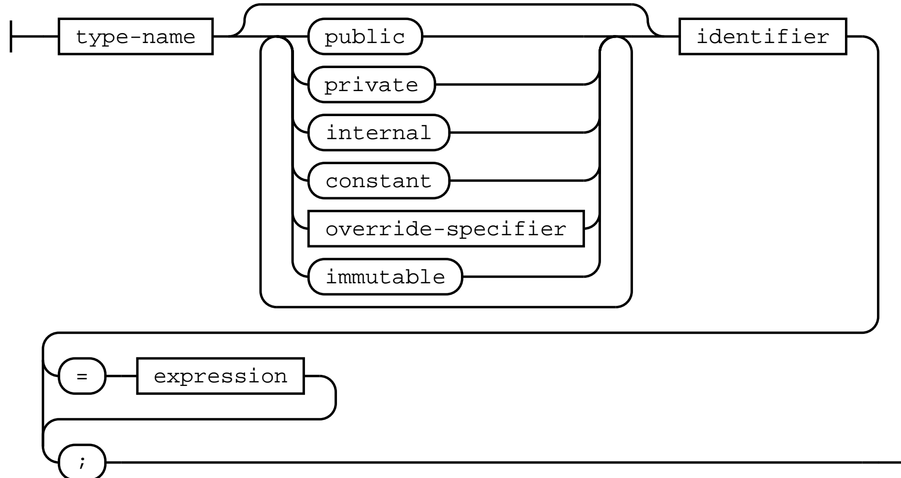
rule user-defined-value-type-definition

用户自定义的值类型的定义。可以出现在源代码单元的顶层，也可以出现在合约，库或接口中。



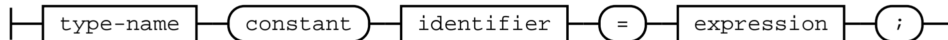
rule state-variable-declaration

一个状态变量的声明。



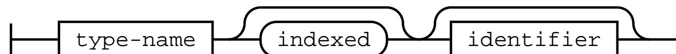
rule constant-variable-declaration

一个常量变量的声明。



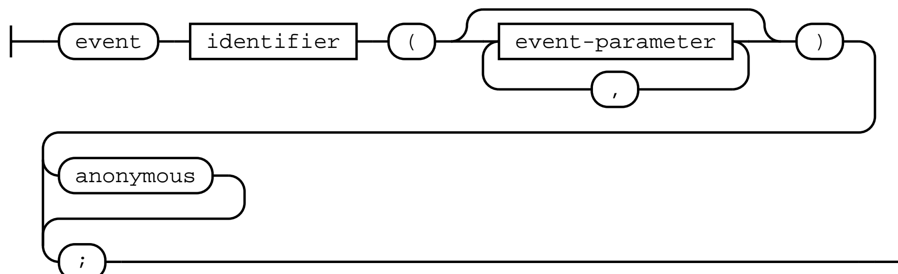
rule event-parameter

一个事件类型的参数。



rule event-definition

一个事件类型的定义。可以发生在合约，库或接口中。

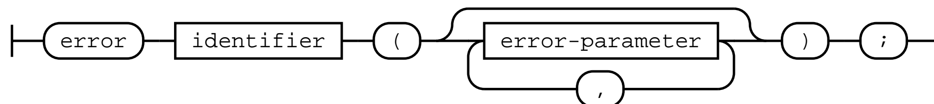


rule error-parameter

一个错误类型的参数。

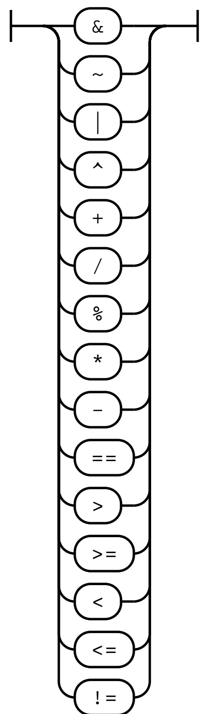
rule error-definition

错误类型定义。



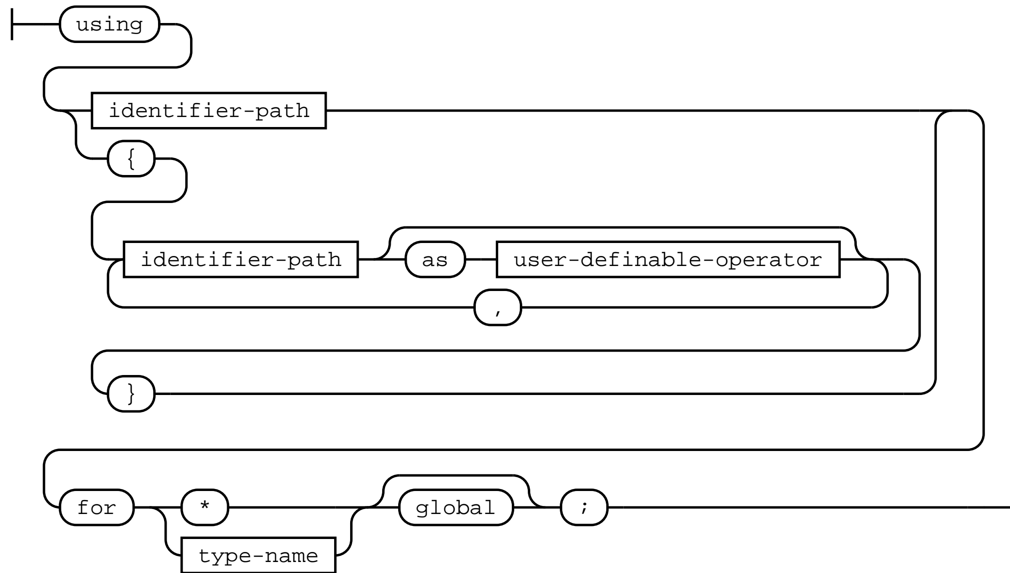
rule user-definable-operator

允许用户使用 *using for* 为某些类型实现的运算符。



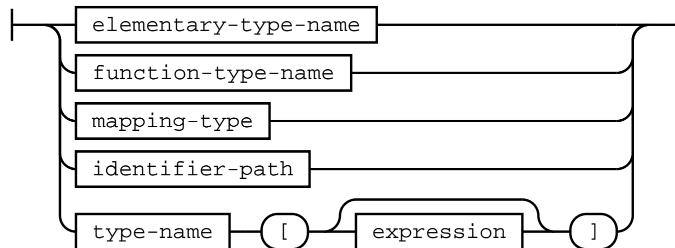
rule using-directive

使用指令将库函数和自由函数附加到类型上。可以在合约和库中以及文件层面中出现。

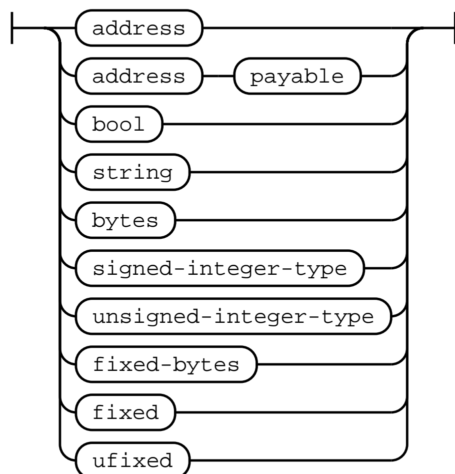


rule type-name

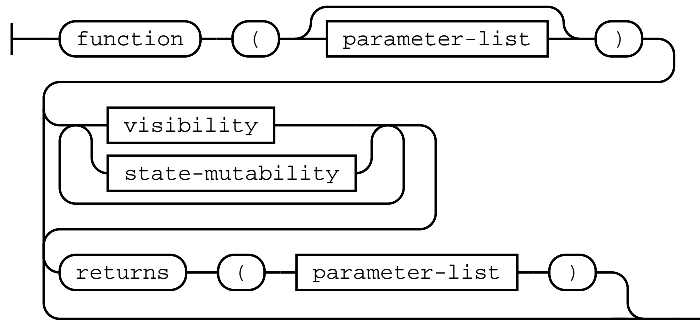
一个类型名称可以是一个基本类型，一个函数类型，一个映射类型，一个用户定义的类型（如合约类型或结构体类型）或一个数组类型。



rule elementary-type-name

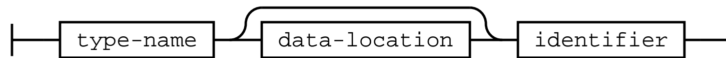


rule function-type-name

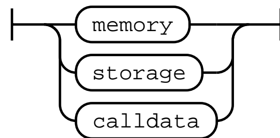


rule variable-declaration

单一变量的声明。



rule data-location

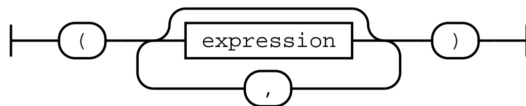


rule expression

复杂的表达式。可以是一个索引访问，一个索引范围访问，一个成员访问，一个函数调用（有可选的函数调用选项），一个类型转换，一个单数或双数表达式，一个比较或赋值，一个三元表达式，一个新的表达式（即一个合约的创建或动态内存数组的分配），一个元组，一个内联数组或一个主要表达式（即一个标识符，字面意思或类型名）。

expression [expression]
expression [expression : expression]
expression . identifier
address
expression { identifier : expression }
,
expression call-argument-list
payable call-argument-list
type (type-name)
++ expression
--
!
~
delete
-
expression ++
--
expression ** expression
expression * expression
/
%
expression + expression
-
expression << expression
>>
>>>
expression & expression
expression ^ expression
expression | expression
expression < expression
>
<=
>=
expression == expression
!=
expression && expression
expression || expression
expression ? expression : expression
expression = expression

rule tuple-expression

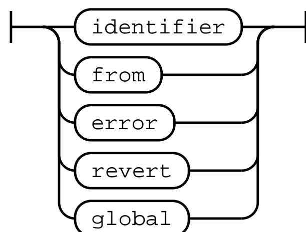


rule inline-array-expression

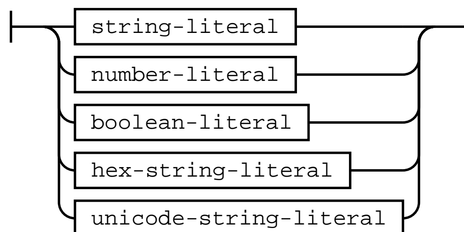
内联数组表达式表示一个静态大小的数组，它是所含表达式的共同类型。

rule identifier

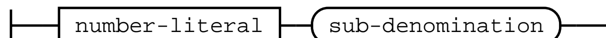
除了常规的非关键字标识符，一些关键字如 'from '和 'error '也可以作为标识符。



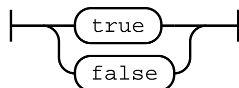
rule literal



rule literal-with-sub-denomination

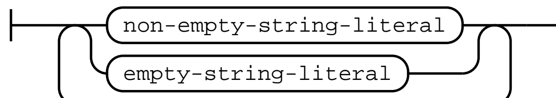


rule boolean-literal



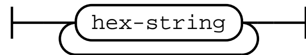
rule string-literal

一个完整的字符串字面量由一个或几个连续的引号字符串组成。



rule hex-string-literal

一个完整的十六进制字符串字面量由一个或几个连续的十六进制字符串组成。

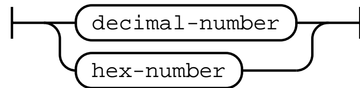


rule unicode-string-literal

一个完整的 unicode 字符串字面量由一个或几个连续的 unicode 字符串组成。

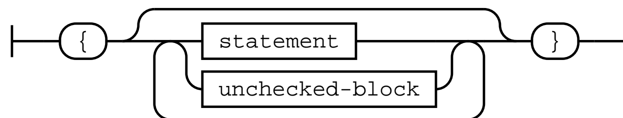
rule number-literal

数字字面量可以是带可选单位的十进制或十六进制数字。



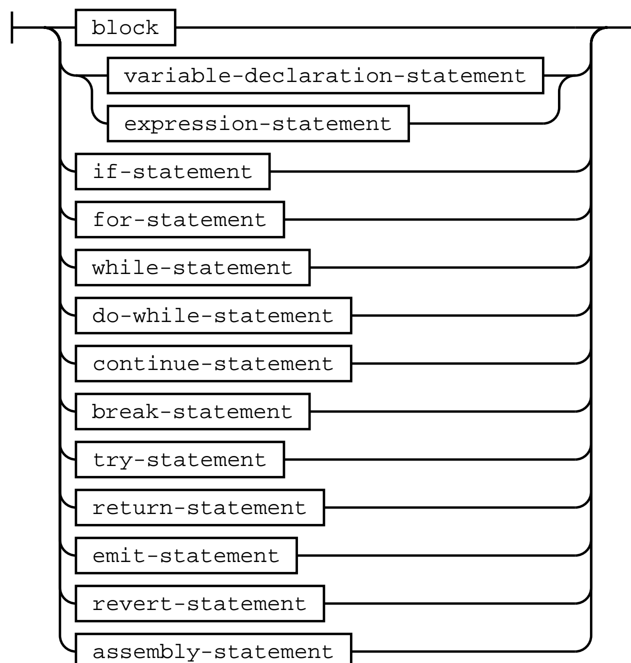
rule block

带花括号的语句块。可以打开自己的作用域。



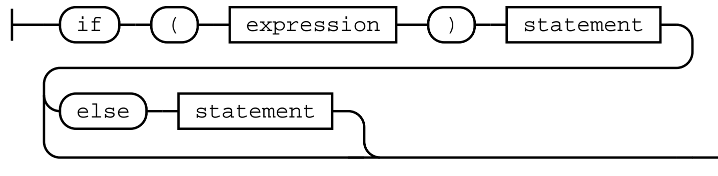
rule unchecked-block

rule statement



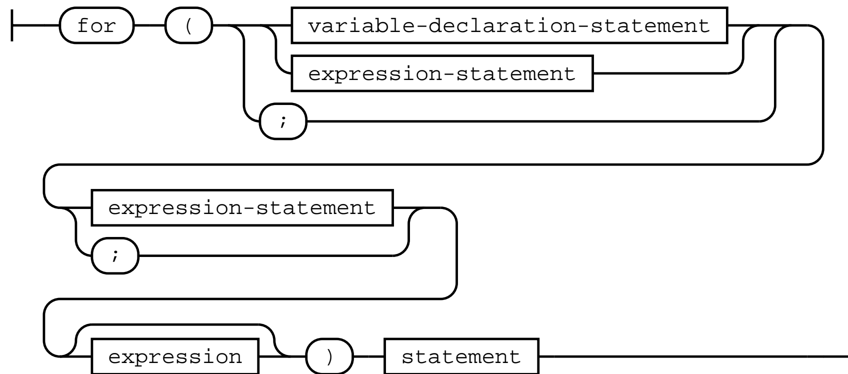
rule if-statement

带有可选的 else 部分的 If 语句。

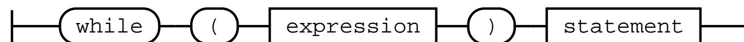


rule for-statement

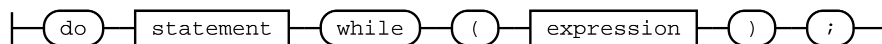
带有可选的初始值，循环条件和循环语句部分的 For 语句。



rule while-statement

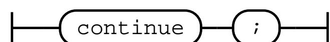


rule do-while-statement



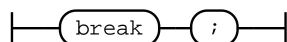
rule continue-statement

一个 continue 语句。只允许在 for、while 或 do-while 循环中使用。



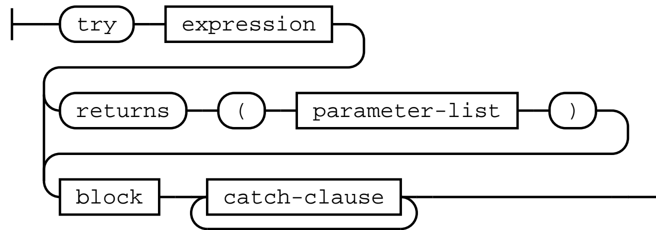
rule break-statement

一个 break 语句。只允许在 for，while 或 do-while 循环中使用。



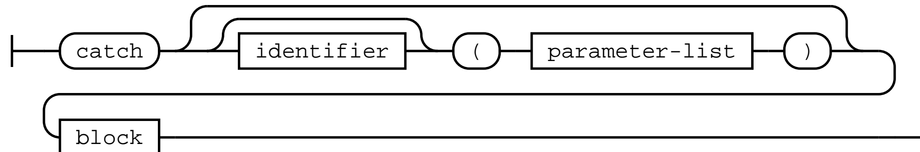
rule try-statement

一个 try 语句。包含的表达式需要是一个外部函数调用或合约创建。

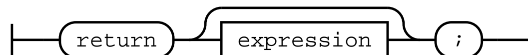


rule catch-clause

Try 语句的 catch 子句。

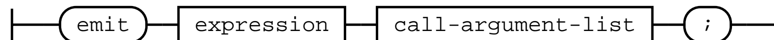


rule return-statement



rule emit-statement

一个发射语句。包含的表达式需要引用一个事件。

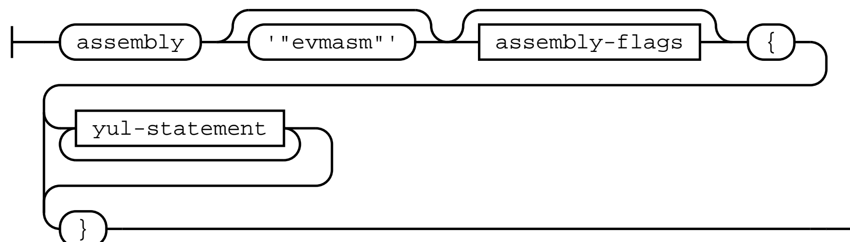


rule revert-statement

一个恢复语句。包含的表达式需要指向一个错误。

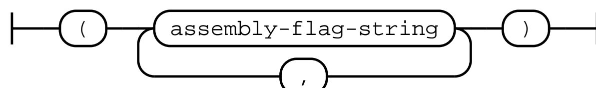
rule assembly-statement

一个内联汇编代码块。内联汇编块的内容使用一个单独的扫描器/读取器，也就是说，内联汇编块内的关键字和允许的标识符集是不同的。



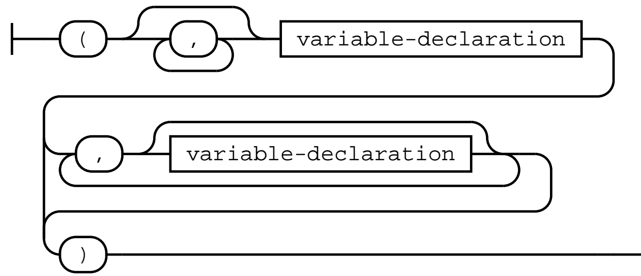
rule assembly-flags

内联标志。逗号分隔的双引号字符串列表作为标志。



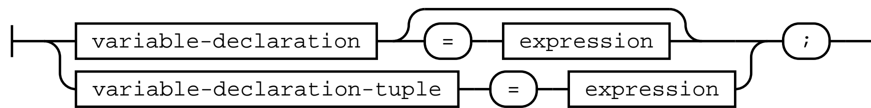
rule variable-declaration-tuple

在变量声明中使用的变量名元组。可能包含空字段。

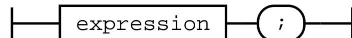


rule variable-declaration-statement

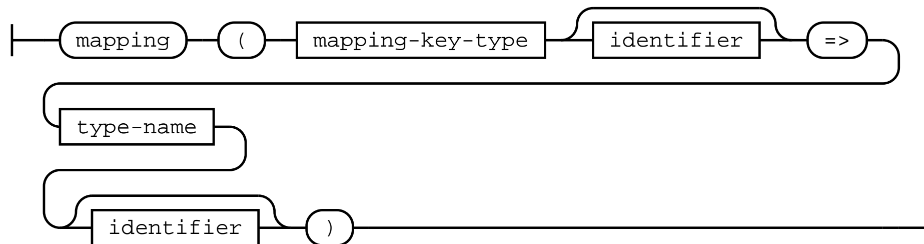
一个变量的声明语句。单个变量可以不带初始值声明，而变量的元组只能用初始值声明。



rule expression-statement

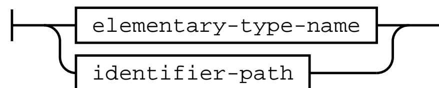


rule mapping-type



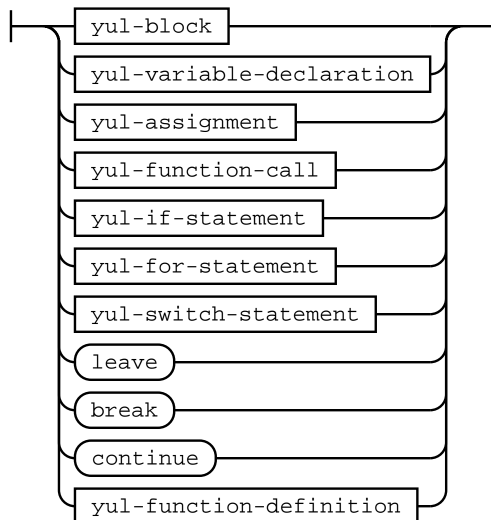
rule mapping-key-type

只有基本类型或用户定义的类型可以作为映射类型的键值。

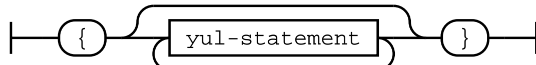


rule yul-statement

内联汇编块中的 Yul 语句。continue 和 break 语句只在 for 循环中有效。离开语句只在函数体内有效。

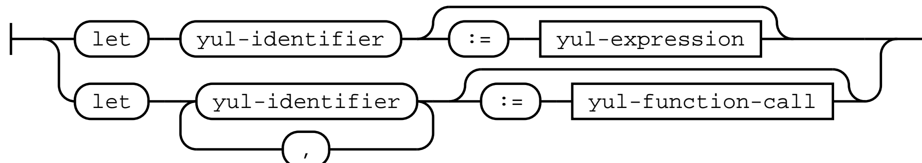


rule yul-block



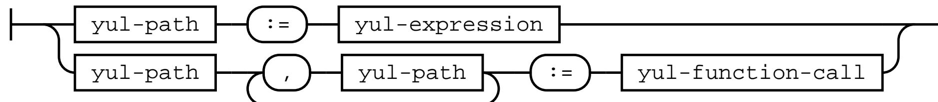
rule yul-variable-declaration

声明一个或多个具有可选的初始值的 Yul 变量。如果声明了多个变量，只有一个函数调用是有效的初始值。

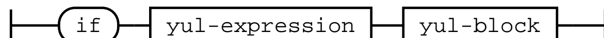


rule yul-assignment

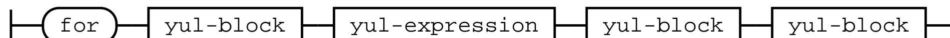
任何表达式都可以分配给一个 Yul 变量，而多分配则需要在右侧调用一个函数。



rule yul-if-statement

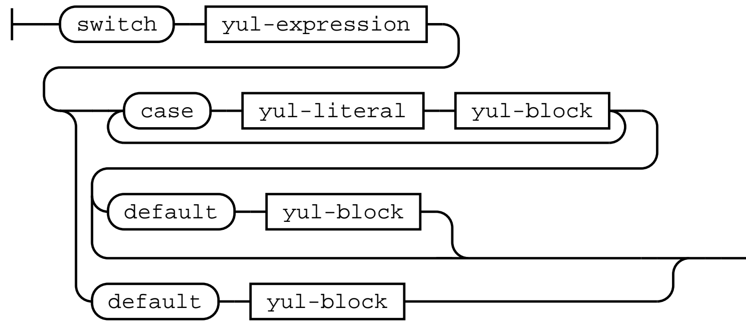


rule yul-for-statement

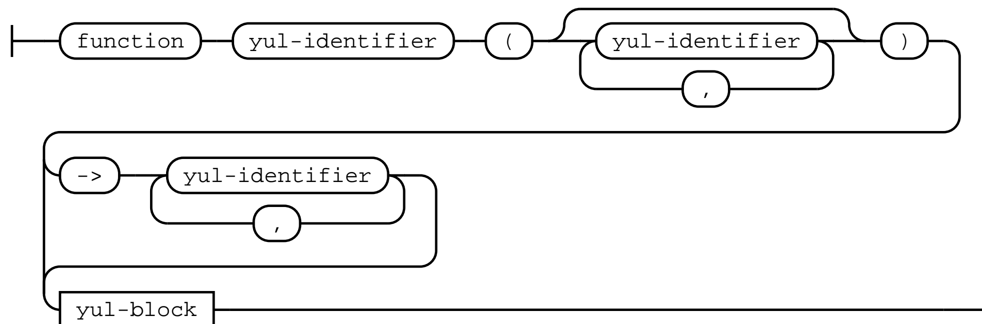


rule yul-switch-statement

Yul switch 语句可以只包括一个默认情况（已废弃）或一个或多个非默认情况，可选择紧跟一个默认情况。

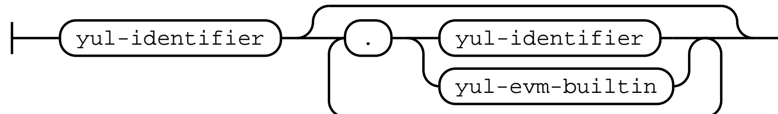


rule yul-function-definition



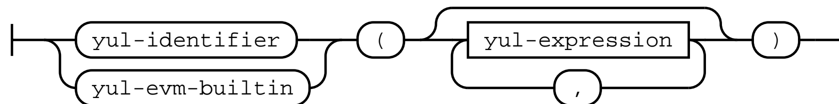
rule yul-path

虽然只有不带点的标识符可以在内联汇编中声明，但含有点的路径可以指内联汇编块之外的声明。

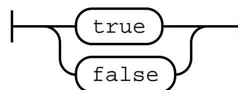


rule yul-function-call

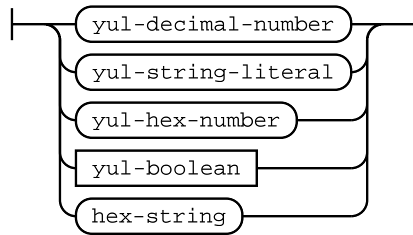
对带有返回值的函数的调用只能作为赋值或变量声明的右侧出现。



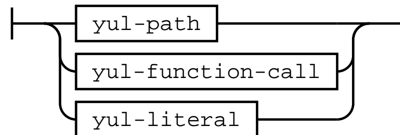
rule yul-boolean



rule yul-literal



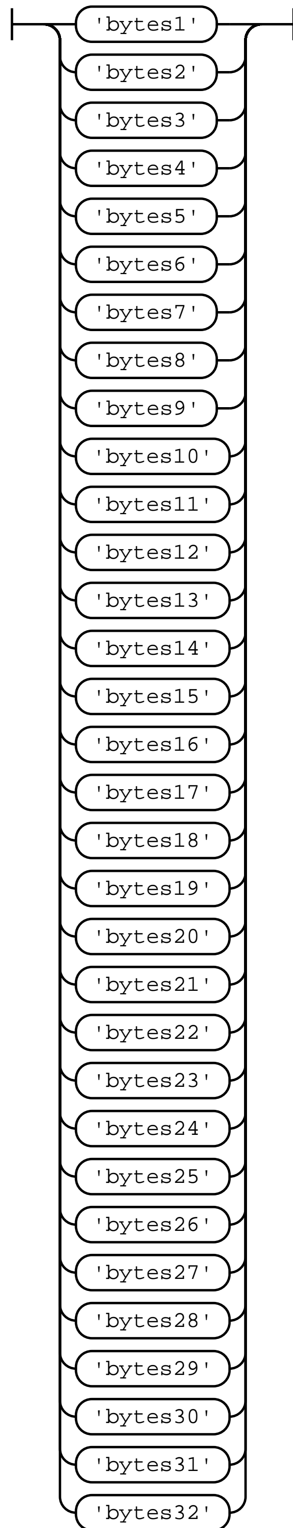
rule yul-expression



lexer grammar SolidityLexer

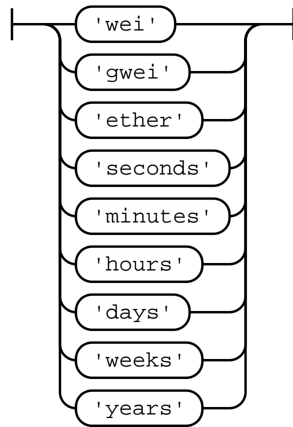
rule fixed-bytes

固定长度的字节类型。



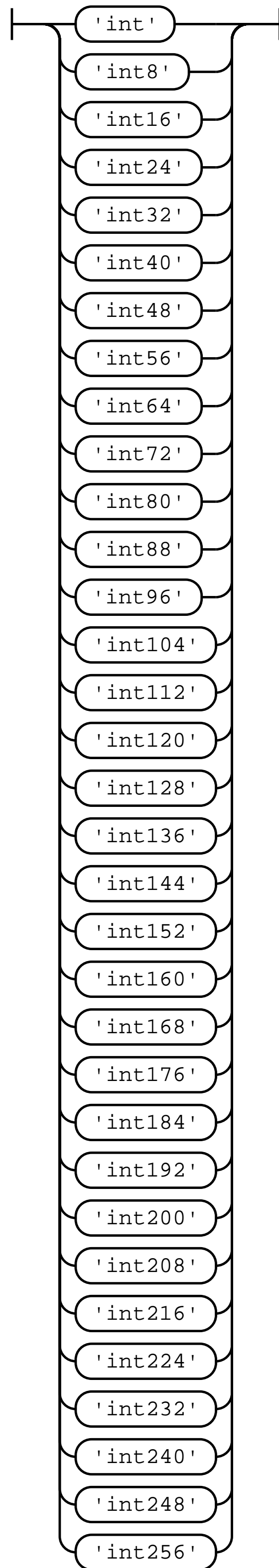
rule sub-denomination

数字的单位计价。



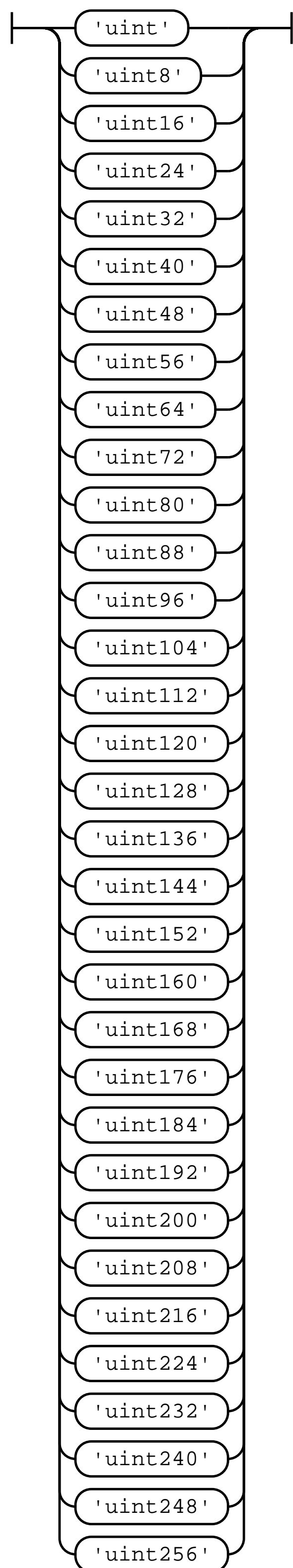
rule signed-integer-type

有符号的整数类型。int 是 int256 的一个别名。



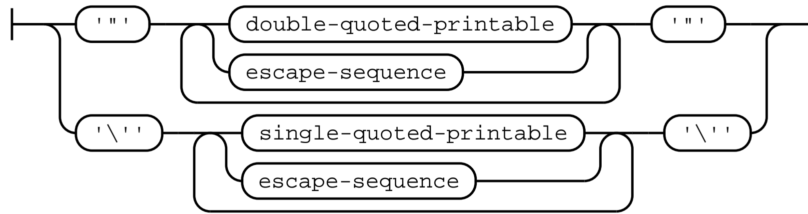
rule unsigned-integer-type

无符号整数类型。uint 是 uint256 的一个别名。



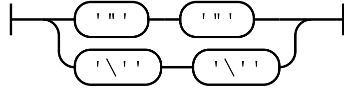
rule non-empty-string-literal

一个非空的带引号的字符串字面量，限制为可打印的字符。



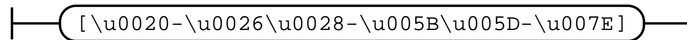
rule empty-string-literal

一个空的字符串字面量



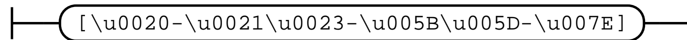
rule single-quoted-printable

除单引号或反斜线外的任何可打印字符。



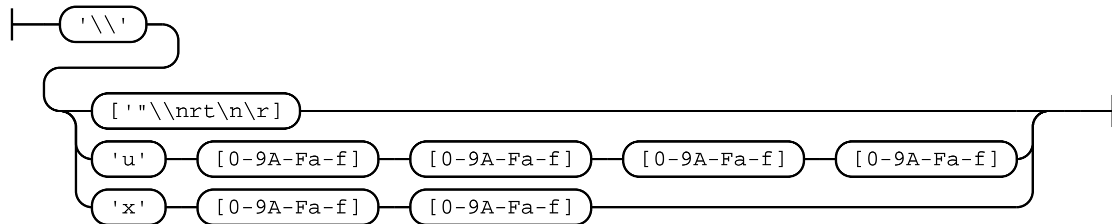
rule double-quoted-printable

除双引号或反斜线外的任何可打印的字符。



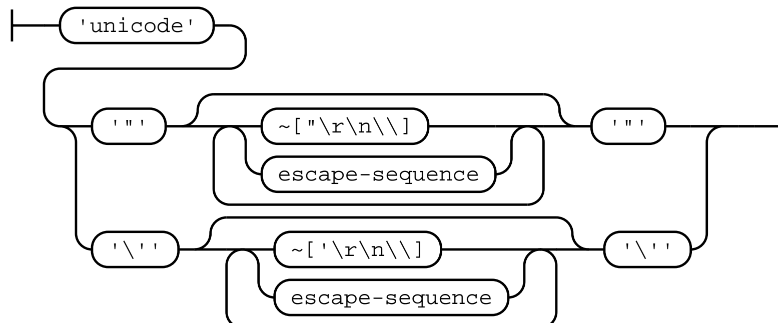
rule escape-sequence

转义序列。除了常见的单字符转义序列外，还可以转义换行，以及允许四个十六进制数字的 unicode 转义\uXXXX 和两个十六进制数字的转义序列\xXX。



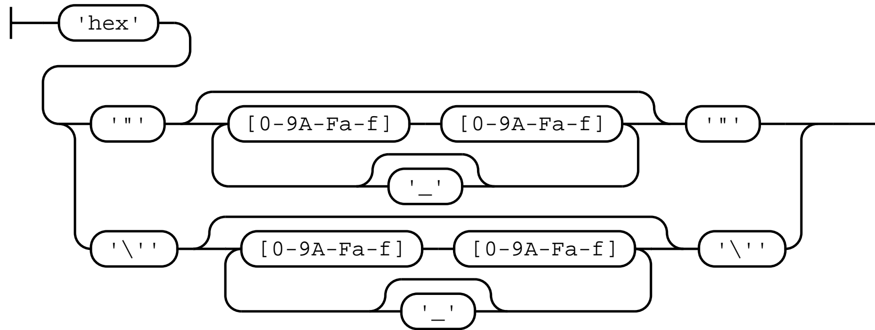
rule unicode-string-literal

单引号字符串字面量，允许任意的 unicode 字符。



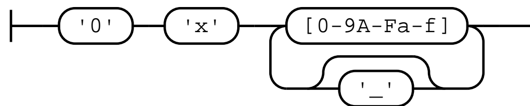
rule hex-string

十六进制字符串需要包含偶数个十六进制数字，可以使用下划线分组。



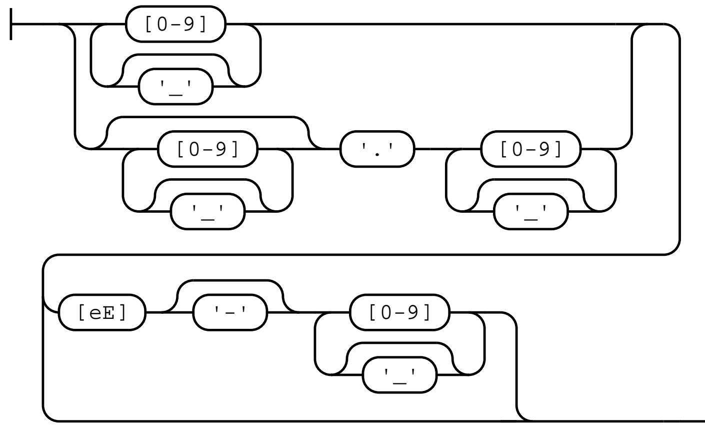
rule hex-number

十六进制数字由前缀和可以用下划线分隔的任意数量的十六进制数字组成。



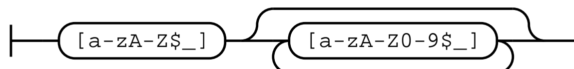
rule decimal-number

一个十进制数字的字面量由十进制数字组成，可以用下划线和一个可选的正负指数来分隔。如果这些数字包含一个小数点，则该数字具有定点类型。



rule identifier

solidity 中的标识符必须以字母，美元符号或下划线开头，并且可以在第一个符号之后再包含数字。



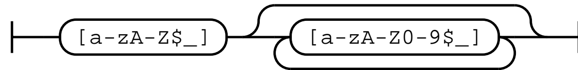
rule yul-evm-builtin

EVM Yul 语言的内置函数。

'stop'
'add'
'sub'
'mul'
'div'
'sdiv'
'mod'
'smod'
'exp'
'not'
'lt'
'gt'
'slt'
'sgt'
'eq'
'iszero'
'and'
'or'
'xor'
'byte'
'shl'
'shr'
'sar'
'addmod'
'mulmod'
'signextend'
'keccak256'
'pop'
'mload'
'mstore'
'mstore8'
'sload'
'sstore'
'msize'
'gas'
'address'
'balance'
'selfbalance'
'caller'
'callvalue'

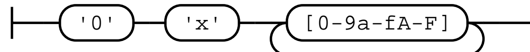
rule yul-identifier

Yul 标识符由字母，美元符号，下划线和数字组成，但不能以数字开头。在内联程序中，用户定义的标识符中不能有圆点。相反，对于由带点的标识符组成的表达式，请参阅 yulPath。



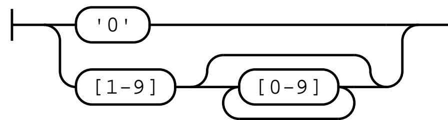
rule yul-hex-number

Yul 中的十六进制字由一个前缀和一个或多个十六进制数字组成。



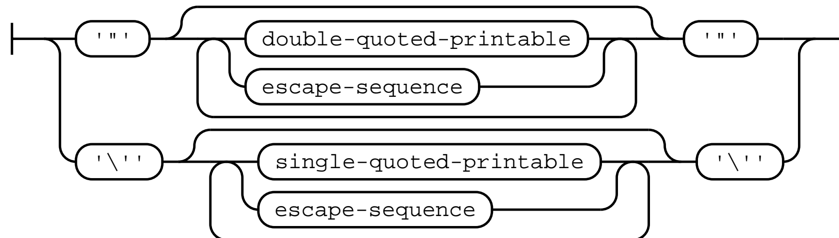
rule yul-decimal-number

Yul 中的小数字面量可以是零或任何不含前导零的小数位序列。



rule yul-string-literal

Yul 中的字符串字面量由一个或多个双引号或单引号字符串组成，这些字符串可能包含转义序列和可打印字符未转义的换行符或未转义的双引号或单引号除外。



rule pragma-token

编译指示令牌。可以包含除分号以外的任何类型的符号。注意，目前 solidity 解析器只允许它的一个子集。

3.13 使用编译器

3.13.1 使用命令行编译器

备注： 这一节并不适用于*solcjs*，即使在命令行模式下使用也不行。

基本用法

`solc` 是 Solidity 仓库的构建目标之一，它是 Solidity 命令行编译器。使用 `solc --help` 可以为您提供所有选项的解释。编译器可以产生各种输出，从简单的二进制文件和抽象语法树（解析树）上的汇编到 gas 使用量的估计。如果您只想编译一个文件，您可以运行 `solc --bin sourceFile.sol` 来生成二进制文件。如果您想通过 `solc` 获得一些更高级的输出信息，可以通过 `solc -o outputDirectory --bin --ast-compact-json --asm sourceFile.sol` 命令将所有的输出都保存到单独的文件中。

优化器选项

在您部署合约之前，在编译时使用 `solc --optimize --bin sourceFile.sol` 激活优化器。默认情况下，优化器将假设合约在其生命周期内被调用 200 次（更确切地说，它假设每个操作码被执行 200 次左右）。如果您想让最初的合约部署更便宜，而后来的函数执行更昂贵，请设置为 `--optimize-runs=1`。如果您期望有很多交易，并且不在乎更高的部署成本和输出大小，那么把 `--optimize-runs` 设置成一个高的数字。这个参数对以下方面有影响（将来可能会改变）：

- 函数调度程序中二进制搜索的大小
- 像大数字或字符串等常量的存储方式

基本路径和导入重映射

命令行编译器将自动从文件系统中读取导入的文件，但同时，它也支持通过如下方式，用 `prefix=path` 选项将路径重定向：

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

这实质上是指示编译器在 `/usr/local/lib/dapp-bin` 下搜索所有以 `github.com/ethereum/dapp-bin/` 开头的文件。

当访问文件系统搜索导入文件时，不以./或../开头的路径 被视为相对于使用 `--base-path` 和 `--include-path` 选项指定的目录（如果没有指定基本路径，则是当前工作目录）。此外，通过这些选项添加的路径部分将不会出现在合约元数据中。

出于安全考虑，编译器对它可以访问的目录有一些限制。在命令行中指定的源文件的目录和重映射的目标路径被自动允许被文件阅读器访问，但其他的都是默认为拒绝的。通过 `--allow-paths /sample/path,/another/sample/path` 语句可以允许额外的路径（和它们的子目录）。通过 `--base-path` 指定的路径内的所有内容都是允许的。

以上只是对编译器如何处理导入路径的一个简化。关于详细的解释，包括例子和边缘情况的讨论，请参考路径解析 一节。

库链接

如果您的合约使用库合约，您会注意到字节码中含有 `__$53aea86b7d70b31448b230b20ae141a537$__` 形式的字符串。这些是实际库的地址的占位符。此占位符是完全限定库名的 keccak256 散列的十六进制编码的 34 个字符前缀。字节码文件也将包含形式为 `// <placeholder> -> <fq library name>` 的代码行，以帮助识别占位符代表的库。注意，完全限定的库名是其源文件的路径和用 `:` 分隔的库名。您可以使用 `solc` 作为链接器，意味着您将在这些地方插入库的地址：

要么在您的命令中加入 `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890 file.sol:Heap=0xabCD567890123456789012345678901234567890"`，为每个库提供一个地址（用逗号或空格作为分隔符），要么将字符串存储在一个文件中（每行一个库），用 `-libraries fileName` 运行 `solc`。

备注： 从 Solidity 0.8.1 开始，接受 `=` 作为库和地址之间的分隔符，而 `:` 作为分隔符已被废弃。它将在未来被删除。目前 `--libraries "file.sol:Math:0x1234567890123456789012345678901234567890 file.sol:Heap:0xabCD567890123456789012345678901234567890"` 也可以工作。

如果调用 `solc` 时有 `--standard-json` 选项，它将在标准输入中期待一个 JSON 输入（如下所述），并在标准输出中返回一个 JSON 输出。这是对更复杂的，特别是自动化使用时的推荐接口。该进程将始终以“成功”状态终止，并通过 JSON 输出来报告任何错误。选项 `--base-path` 也以标准 JSON 模式处理。

如果调用 `solc` 时带有 `--link` 选项，所有输入文件都被编译成格式为 `__$53aea86b7d70b31448b230b20ae141a537$__` 形式的未链接的二进制文件（十六进制编码），并被本地链接（如果从标准输入（stdin）读取输入，则被写到标准输出（stdout））。在这种情况下，除了 `--libraries` 以外的所有选项都被忽略（包括 `-o`）。

警告： 不推荐在生成的字节码上手动链接库文件，因为它不会更新合约元数据。由于元数据包含在编译时指定的库的列表，而字节码包含元数据哈希，您将得到不同的二进制文件，并且这取决于何时进行链接。

您应该在编译合约时请求编译器链接库文件，方法是使用 `solc` 的 `--libraries` 选项或 `libraries` 键（如果您使用编译器的标准 JSON 接口）。

备注： 库的占位符曾经是库本身的完全限定名称，而不是它的哈希值。这种格式仍然被 `solc --link` 支持，但编译器将不再输出它。这一改变是为了减少库之间发生碰撞的可能性，因为只有完全限定的库名的前 36 个字符可以被使用。

3.13.2 将 EVM 版本设置为目标版本

当您编译您的合约代码时，您可以指定以太坊虚拟机版本来编译，以避免特定的功能或行为。

> **警告：** 在错误的 EVM 版本进行编译会导致错误，奇怪和失败的行为。请确保，特别是在运行一个私有链的情况下，您使用匹配的 EVM 版本。

在命令行中，您可以选择 EVM 的版本，如下所示：

```
solc --evm-version <VERSION> contract.sol
```

在标准 *JSON* 接口 中，使用 "settings" 字段中的键 "evmVersion"。

```
{
  "sources": {/* ... */},
  "settings": {
    "optimizer": {/* ... */},
    "evmVersion": "<VERSION>"
  }
}
```

EVM 版本选项

以下是一个 EVM 版本的列表，以及每个版本中引入的编译器相关变化。每个版本之间不保证向后兼容。

- **homestead**
  - （最老的版本）
- **tangerineWhistle**
  - 访问其他账户的 gas 成本增加，与 gas 估算和优化器有关。
  - 对于外部调用，所有 gas 都是默认发送的，以前必须保留一定的数量。
- **spuriousDragon**
  - exp 操作码的 gas 成本增加，与 gas 估计和优化器有关。
- **byzantium**
  - 在汇编中可使用操作码 returndatacopy，returndatasize 和 staticcall。
  - staticcall 操作码在调用非库合约 view 或 pure 函数时使用，它可以防止函数在 EVM 级别修改状态，也就是说，甚至适用于您使用无效的类型转换时。
  - 可以访问从函数调用返回的动态数据。
  - 引入了 revert 操作码，这意味着 revert 将不会浪费 gas。

- **constantinople**
  - 在汇编中可使用操作码 `create2`, `extcodehash`, `shl`, `shr` 和 `sar`。
  - 移位运算符使用移位运算码，因此需要的 gas 较少。
- **petersburg**
  - 编译器的行为与 constantinople 版本的行为相同。
- **istanbul**
  - 在汇编中可使用操作码 `chainid` 和 `selfbalance`。
- **berlin**
  - `SLOAD`，`*CALL`，`BALANCE`，`EXT*` 和 `SELFDESTRUCT` 的 gas 成本增加。编译器假设这类操作的 gas 成本是固定的。这与 gas 估计和优化器有关。
- **london**
  - 区块的基本费用（EIP-3198 和 EIP-1559 ）可以通过全局的 `block.basefee` 或内联汇编中的 `basefee()` 访问。
- **paris**
  - 引入了 `prevrandao()` 和 `block.prevrandao`，并改变了现在已经废弃的 `block.difficulty` 的语义，不允许在内联汇编中使用 `difficulty()` （见 EIP-4399 ）。
- `shanghai` （**默认项**）- 由于引入了 `push0`，代码尺寸更小，并且节省了 gas（参见 EIP-3855）。

3.13.3 编译器输入和输出 JSON 说明

推荐的与 Solidity 编译器连接的方式，特别是对于更复杂和自动化的设置，是所谓的 JSON 输入输出接口。编译器的所有发行版都提供相同的接口。

这些字段一般都会有变化，有些是可选的（如前所述），但我们尽量只做向后兼容的改动。

编译器 API 期望 JSON 格式的输入，并将编译结果输出为 JSON 格式的输出。不使用标准错误输出，进程将始终以“成功”状态终止，即使存在错误。错误总是作为 JSON 输出的一部分报告。

以下各小节通过一个例子来描述该格式。当然，注释是不允许的，在此仅用于解释。

输入说明

```
{
  // 必选: 源代码语言。目前支持的是 “Solidity“, “Yul“ 和 “SolidityAST” ␣
↪(实验性的)。
  "language": "Solidity",
  // 必选
```

(续下页)

(接上页)

```
"sources":
{
  // 这里的键值是源文件的 “全局“ 名称，
  // 导入文件可以通过重映射使用其他文件（见下文）。
  "myFile.sol":
  {
    // 可选： 源文件的kaccak256哈希值
    // 如果通过URL导入，它用于验证检索的内容。
    "keccak256": "0x123...",
    // 必选（除非声明了 "content" 字段，参见下文）: 指向源文件的URL。
    // 应按此顺序导入URL，并根据keccak256哈希值检查结果（如果有的话）。
    // 如果哈希值不匹配，或者没有一个URL(s)的结果是成功的，就应该产生一个错误。
    // 使用命令行界面只支持文件系统路径。
    //␣
↪通过JavaScript接口，URL将被传递给用户提供的读取回调，因此可以使用回调支持的任何URL。
    "urls":
    [
      "bzzr://56ab...",
      "ipfs://Qma...",
      "/tmp/path/to/file.sol"
      // 如果使用文件，其目录应通过 `--allow-paths <path>` 添加到命令行中。
    ]
    // 如果语言设置为 “SolidityAST”，则需要在 “ast” 字段下提供 AST。
    // 请注意，ASTs 的导入是试验性的，尤其是：
    // - 导入无效的 ASTs 会产生未定义的结果，并且
    // - 对无效的 ASTs 不提供适当的错误报告。
    // 此外，请注意 AST 导入只消耗编译器在 “stopAfter” （停止后）模式下生成的 AST␣
↪字段：
    // “parsing” 模式下生成的 AST 字段，然后重新执行分析，
    // 因此 AST 中任何基于分析的注释在导入时都会被忽略。
    "ast": { ... } // 格式化为 json ast 请求的 ``ast`` 输出选择。
  },
  "destructible":
  {
    // 可选：源文件的keccak256哈希值
    "keccak256": "0x234...",
    // 必选：（除非使用 “urls “）：源文件的字面内容
    "content": "contract destructible is owned { function shutdown() { if (msg.
↪sender == owner) selfdestruct(owner); } }"
  }
},
// 可选
"settings":
```

(续下页)

(接上页)

```
{
  // 可选： 在给定的阶段后停止编译。目前这里只有 “parsing” 有效。
  "stopAfter": "parsing",
  // 可选： 经过排序的重映射列表
  "remappings": [ ":g=/dir" ],
  // 可选： 优化器设置
  "optimizer": {
    // 默认情况下是禁用的。
    // 注意： enabled=false 仍然保留了一些优化功能。见下面的注解。
    // 警告： 在0.8.6版本之前，省略 “enabled“ 键并不等同于将其设置为false,
    // 实际上会禁用所有优化。
    "enabled": true,
    // 根据您打算运行代码的次数进行优化。
    // 较低的值将更多地针对初始部署成本进行优化，
    // 较高的值将更多地针对高频使用进行优化。
    "runs": 200,
    // 打开或关闭优化器组件的细节。
    // 上面的 “enabled“ 开关提供了两个默认值，
    // 可以在这里进行调整。如果给出了 “details“, “enabled“ 可以省略。
    "details": {
      // 如果没有给出 details, 窥视孔优化器总是打开的，使用 details 来关闭它。
      "peephole": true,
      // 如果没有给出 details, 内联器总是打开的，
      // 使用 details来关闭它。
      "inliner": true,
      // 如果没有给出 details, 未使用的跳板移除器总是打开的，
      // 使用 details来关闭它。
      "jumpdestRemover": true,
      // 在换元运算中，有时会对字词重新排序。
      "orderLiterals": false,
      // 移除重复的代码块
      "deduplicate": false,
      // 常见的子表达式消除，这是最复杂的步骤，但也能提供最大的收益。
      "cse": false,
      // 优化代码中字面数字和字符串的表示。
      "constantOptimizer": false,
      // 新的Yul优化器。主要在ABI coder v2 和 内联汇编的代码上运行。
      // 它与全局优化器设置一起被激活，并且可以在这里停用。
      // 在 Solidity 0.6.0 之前，它必须通过这个开关激活。
      "yul": false,
      // Yul优化器的调优选项。
      "yulDetails": {
        // 改善变量的堆栈槽的分配，可以提前释放堆栈槽。
```

(续下页)

(接上页)

```
          // 如果Yul优化器被激活，则默认激活。
          "stackAllocation": true,
          // 选择要应用的优化步骤。
          // 也可以同时修改优化序列和清理序列。
          // 每个序列的指令用 “:” 分隔，该值以 优化-序列:清理-序列 的形式提供。
          // 更多信息见 “优化器 > 选择优化”。
          // 这个字段是可选的，如果不提供，优化和清理的默认序列都会使用。
          // 如果只提供了其中一个选项，另一个将不会被运行。
          // 如果只提供分隔符 “:”，
          // 那么优化和清理序列都不会被运行。
          // 如果设置为空值，则只使用默认的清理序列，
          // 不应用任何优化步骤。
          "optimizerSteps": "dhfoDgvulfnTUtnIf..."
        }
      }
    },
    // 编译EVM的版本。
    // 影响到类型检查和代码生成。版本可以是 homestead,
    // tangerineWhistle, spuriousDragon, byzantium, constantinople, petersburg,␣
↪istanbul, berlin, london or paris
    "evmVersion": "byzantium",
    // 可选：改变编译管道以通过Yul的中间表示法。
    // 这在默认情况下是假的。
    "viaIR": true,
    // 可选： 调试设置
    "debug": {
      // 如何处理 revert (和require) 的原因字符串。设置是
      // "default", "strip", "debug" 和 "verboseDebug"。
      // "default" 不注入编译器生成的revert字符串，而是保留用户提供的字符串。
      // "strip"␣
↪删除所有的revert字符串（如果可能的话，即如果使用了字面意义），以保持副作用。
      // "debug" 为编译器生成的内部revert注入字符串，目前为ABI编码器V1和V2实现。
      // "verboseDebug" 甚至将进一步的信息附加到用户提供的revert字符串中（尚未实现）。
      "revertStrings": "default",
      // 可选：在产生的EVM汇编和Yul代码的注释中包括多少额外的调试信息。可用的组件是：
      // - `location`: `@src <index>:<start>:<end>` 形式的注解，
      //    表明原始 Solidity 文件中相应元素的位置，其中：
      //      - `<index>` 是与 `@us-src` 注释相匹配的文件索引。
      //      - `<start>` 是该位置的第一个字节的索引。
      //      - `<end>` 是该位置后第一个字节的索引。
      // - `snippet`: 来自 `@src` 所示位置的单行代码片断。
      //      该片段有引号，并跟随相应的 `@src` 注释。
      // - `*`: 通配符值，可用于请求所有的东西。
```

(续下页)

(接上页)

```
      "debugInfo": ["location", "snippet"]
    },
    // 元数据设置 (可选)
    "metadata": {
      // CBOR元数据默认是附加在字节码的最后。
      // 将此设置为false，会从运行时和部署时代码中省略元数据。
      "appendCBOR": true,
      // 只使用字面内容，不使用URL（默认为false）
      "useLiteralContent": true,
      // 对附加在字节码上的元数据哈希值使用给定的哈希值方法。
      // 元数据哈希可以通过选项 "none "从字节码中删除。
      // 其他选项是 "ipfs" 和 "bzzr1"。
      // 如果省略该选项，默认使用 "ipfs"。
      "bytecodeHash": "ipfs"
    },
    // 库的地址。如果这里没有给出所有的库，
    // 可能会导致未链接的对象，其输出数据是不同的。
    "libraries": {
      // 顶层键是使用该库的源文件的名称。
      // 如果使用了重映射，这个源文件应该与应用重映射后的全局路径一致。
      // 如果这个键是一个空字符串，那就是指一个全局水平。
      "myFile.sol": {
        "MyLib": "0x123123..."
      }
    },
    // 以下可用于根据文件和合约名称选择所需的输出。
    //
↪如果这个字段被省略，那么编译器就会加载并进行类型检查，但除了错误之外不会产生任何输出。
    // 第一层键是文件名，第二层键是合约名。
    // 一个空的合约名称用于不与合约绑定而是与整个源文件绑定的输出，如AST。
    // 以星号作为合约名称是指文件中的所有合约。
    // 同样地，以星形作为文件名可以匹配所有文件。
    // 要选择编译器可能产生的所有输出，
    // 使用 "outputSelection"。{ "*": { "*": [ "*" ], "": [ "*" ] } }",
    // 但要注意，这可能会不必要地减慢编译过程。
    //
    // 可用的输出类型如下：
    //
    // 文件级别（需要空字符串作为合约名称）：
    //   ast - 所有源文件的AST
    //
    // 合约级别（需要合约名称或 "*"）：
    //   abi - ABI
```

(续下页)

(接上页)

```
    //   devdoc - 开发者文档 (Natspec格式)
    //   userdoc - 用户文档 (Natspec格式)
    //   metadata - 元数据
    //   ir - 优化代码前的Yul中间表示法
    //   irOptimized - 优化后的中间表现
    //   storageLayout - 合约的状态变量的槽位、偏移量和类型
    //   evm.assembly - 新的汇编格式
    //   evm.legacyAssembly - JSON中的旧式汇编格式
    //   evm.bytecode.functionDebugData - 在函数层面的调试信息
    //   evm.bytecode.object - 字节码对象
    //   evm.bytecode.opcodes - 操作码列表
    //   evm.bytecode.sourceMap - 源码映射 (对调试有用)
    //   evm.bytecode.linkReferences - 链接引用 (如果是未链接的对象)
    //   evm.bytecode.generatedSources - 由编译器生成的源码
    //   evm.deployedBytecode* - 部署的字节码 (拥有evm.bytecode的所有选项) 。
    //   evm.deployedBytecode.immutableReferences - 从AST␣
↪id到引用不可变的字节码范围的映射
    //   evm.methodIdentifiers - 函数哈希值的列表
    //   evm.gasEstimates - 函数gas估计
    //   ewasm.wast - WebAssembly S-expressions格式的Ewasm
    //   ewasm.wasm - WebAssembly二进制格式的Ewasm
    //
    // 注意, 使用 `evm`, `evm.bytecode`, `ewasm` 等将选择该输出的每个目标部分。
    // 此外, `*` 可以作为通配符来请求所有东西。
    //
    "outputSelection": {
      "*": {
        "*": [
          "metadata", "evm.bytecode" // 启用每个合约的元数据和字节码输出。
          , "evm.bytecode.sourceMap" // 启用每个合约的源码映射输出。
        ],
        "": [
          "ast" // 启用每个文件的AST输出。
        ]
      },
      // 启用文件def中定义的MyContract的abi和opcodes输出。
      "def": {
        "MyContract": [ "abi", "evm.bytecode.opcodes" ]
      }
    },
    // modelChecker对象是实验性的, 可能会有变化。
    "modelChecker":
    {
```

(续下页)

(接上页)

```
      // 选择哪些合约应作为部署的合约进行分析。
      "contracts":
      {
        "source1.sol": ["contract1"],
        "source2.sol": ["contract2", "contract3"]
      },
      // 选择除法和模数运算的编码方式。
      // 当使用 `false` 时，它们被替换为与松弛变量的乘法。这是默认的。
      // 如果您使用CHC引擎而不使用Spacer作为Horn求解器（例如使用Eldarica），
      // 建议在这里使用 `true`。
      // 关于这个选项的更详细的解释，请参见形式验证部分。
      "divModWithSlacks": false,
      // 选择要使用的模型检查器引擎：所有（默认），bmc，chc，无。
      "engine": "chc",
      // 选择在编译时可获得被调用函数代码的情况下，
      // 外部调用是否应被视为可信。
      // 详情请参阅SMT检查器部分。
      "extCalls": "trusted",
      // 选择哪些类型的不变性应该报告给用户：合约，重入。
      "invariants": ["contract", "reentrancy"],
      // 选择是否输出所有验证过的目标。默认为 `false`。
      "showProved": true,
      // 选择是否输出所有未验证的目标。默认为 `false`。
      "showUnproved": true,
      // 选择是否输出所有不支持的语言功能。默认为 `false`。
      "showUnsupported": true,
      // 如果有的话，选择应该使用哪些求解器。
      // 关于求解器的描述，见形式验证部分。
      "solvers": ["cvc4", "smtlib2", "z3"],
      //␣
↪选择哪些目标应该被检查：常数条件，下溢，溢出，除以零，余额，断言，弹出空数组，界外。
      // 如果没有给出该选项，所有目标都被默认检查，除了 Solidity >=0.8.7 的下溢/溢出。
      // 目标描述见形式化验证部分。
      "targets": ["underflow", "overflow", "assert"],
      // 每个SMT查询的超时时间，以毫秒为单位。
      // 如果没有给出这个选项，SMTChecker将默认使用确定性的资源限制。
      // 给定超时为0意味着任何查询都没有资源/时间限制。
      "timeout": 20000
    }
  }
}
```

输出描述

```
{
  // 可选：如果没有遇到错误/警告/消息，则不存在。
  "errors": [
    {
      // 可选：在源文件中的位置。
      "sourceLocation": {
        "file": "sourceFile.sol",
        "start": 0,
        "end": 100
      },
      // 可选：更多的位置（如有冲突的声明的地方）。
      "secondarySourceLocations": [
        {
          "file": "sourceFile.sol",
          "start": 64,
          "end": 92,
          "message": "Other declaration is here:"
        }
      ],
      // 强制：错误类型，如 "TypeError"，"InternalCompilerError"，"Exception" ␣
↪等等。
      // 完整的类型清单见下文。
      "type": "TypeError",
      // 强制：发生错误的组件，例如 "general"，"ewasm" 等
      "component": "general",
      // 强制：错误的严重级别（"error"，"warning" 或␣
↪"info"，但请注意，这可能在未来被扩展。）
      "severity": "error",
      // 可选：错误原因的唯一代码
      "errorCode": "3141",
      // 强制
      "message": "Invalid keyword",
      // 可选：带错误源位置的格式化消息
      "formattedMessage": "sourceFile.sol:100: Invalid keyword"
    }
  ],
  // 这包含文件级的输出。
  // 它可以通过outputSelection设置进行限制/过滤。
  "sources": {
    "sourceFile.sol": {
      // 标识符（用于源码映射）
      "id": 1,
```

(续下页)

(接上页)

```
      // AST对象
      "ast": {}
    }
  },
  // 这里包含了合约级别的输出。
  // 它可以通过outputSelection设置进行限制/过滤。
  "contracts": {
    "sourceFile.sol": {
      // 如果使用的语言没有合约名称，则该字段应该留空。
      "ContractName": {
        // 以太坊合约的应用二进制接口（ABI）。如果为空，则表示为空数组。
        // 请参阅 https://docs.soliditylang.org/en/develop/abi-spec.html
        "abi": [],
        // 请参阅元数据输出文档（序列化的JSON字符串）
        "metadata": "{/* ... */}",
        // 用户文档（natspec）
        "userdoc": {},
        // 开发人员文档（natspec）
        "devdoc": {},
        // 中间表示形式 (string)
        "ir": "",
        // 请参阅 "存储布局" 文档。
        "storageLayout": {"storage": [/* ... */], "types": {/* ... */} },
        // EVM相关输出
        "evm": {
          // 汇编 (string)
          "assembly": "",
          // 旧风格的汇编 (object)
          "legacyAssembly": {},
          // 字节码和相关细节
          "bytecode": {
            // 在函数层面上调试数据。
            "functionDebugData": {
              // 接下来是一组函数，包括编译器内部的和用户定义的函数。
              // 这组函数不一定是完整的。
              "@mint_13": { // 函数的内部名称
                "entryPoint": 128, // 函数开始所在字节码的字节偏移量（可选）
                "id": 13, // 函数定义的AST ID，或者对于编译器内部的函数为空（可选）
                "parameterSlots": 2, // 函数参数的EVM堆栈槽的数量（可选）
                "returnSlots": 1 // 返回值的EVM堆栈槽的数量（可选）
              }
            },
            // 作为十六进制字符串的字节码。
```

(续下页)

(接上页)

```
            "object": "00fe",
            // 操作码列表（字符串）
            "opcodes": "",
            // 作为一个字符串的源映射。参见源映射的定义。
            "sourceMap": "",
            // 由编译器生成的源文件的数组。目前只包含一个Yul文件。
            "generatedSources": [{
              // Yul AST
              "ast": {/* ... */},
              // 文本形式的源文件（可能包含注释）。
              "contents":"{ function abi_decode(start, end) -> data { data :=␣
↪calldataload(start) } }",
              // 源文件ID，用于源引用，与Solidity源文件相同的 "命名空间"。
              "id": 2,
              "language": "Yul",
              "name": "#utility.yul"
            }],
            // 如果给定，这就是一个非链接的对象。
            "linkReferences": {
              "libraryFile.sol": {
                // 在字节码中的字节偏移量。
                // 链接取代了位于那里的20个字节。
                "Library1": [
                  { "start": 0, "length": 20 },
                  { "start": 200, "length": 20 }
                ]
              }
            }
          },
          "deployedBytecode": {
            /* ..., */ // 与上述布局相同。
            "immutableReferences": {
              // 有两个对AST ID为3的不可变的引用，都是32字节长。
              // 一个在字节码偏移量42，另一个在字节码偏移量80。
              "3": [{ "start": 42, "length": 32 }, { "start": 80, "length": 32 }]
            }
          },
          // 函数哈希值的列表
          "methodIdentifiers": {
            "delegate(address)": "5c19a95c"
          },
          // 函数gas估计
          "gasEstimates": {
```

(续下页)

(接上页)

```
          "creation": {
            "codeDepositCost": "420000",
            "executionCost": "infinite",
            "totalCost": "infinite"
          },
          "external": {
            "delegate(address)": "25000"
          },
          "internal": {
            "heavyLifting()": "infinite"
          }
        }
      },
      // Ewasm相关的输出
      "ewasm": {
        // S-expressions格式
        "wast": "",
        // 二进制格式（十六进制字符串）
        "wasm": ""
      }
    }
  }
}
}
```

错误类型

1. `JSONError`：JSON 输入不符合所需格式，例如，输入不是 JSON 对象，不支持的语言等。
2. `IOError`：IO 和导入处理错误，例如，在提供的源里包含无法解析的 URL 或哈希值不匹配。
3. `ParserError`：源代码不符合语言规则。
4. `DocstringParsingError`：注释块中的 NatSpec 标签无法解析。
5. `SyntaxError`：语法错误，例如 `continue` 在 `for` 循环外部使用。
6. `DeclarationError`：无效的，无法解析的或冲突的标识符名称比如 `Identifier not found`
7. `TypeError`：类型系统内的错误，例如无效类型转换，无效赋值等。
8. `UnimplementedFeatureError`：当前编译器不支持该功能，但预计将在未来的版本中支持。
9. `InternalCompilerError`：在编译器中触发的内部错误—应将此报告为一个 issue。
10. `Exception`：编译期间的未知失败—应将此报告为一个 issue。
11. `CompilerError`：编译器堆栈的无效使用—应将此报告为一个 issue。

12. `FatalError`：未正确处理致命错误—应将此报告为一个 issue。
13. `YulException`：在 Yul 代码生成过程中出现错误 - 这应该作为一个 issue 报告。
14. `Warning`：警告，不会停止编译，但应尽可能处理。
15. `Info`：编译器认为用户可能会在其中发现有用的信息，并不危险，也不一定需要处理。

3.14 分析编译器的输出结果

看一下编译器生成的汇编代码往往是有用的。生成的二进制文件，即 `solc --bin contract.sol` 的输出，通常很难阅读。建议使用标志 `--asm` 来分析汇编输出。即使是很大的合约，看一下改变前后的汇编结果的差异，往往是很有启发的。

以下合约（命名为 `contract.sol`）为例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function one() public pure returns (uint) {
        return 1;
    }
}
```

以下是 `solc --asm contract.sol` 的输出

```
======= contract.sol:C =======
EVM assembly:
    /* "contract.sol":0:86  contract C {... */
  mstore(0x40, 0x80)
  callvalue
  dup1
  iszero
  tag_1
  jumpi
  0x00
  dup1
  revert
tag_1:
  pop
  dataSize(sub_0)
  dup1
  dataOffset(sub_0)
  0x00
  codecopy
```

(续下页)

(接上页)

```
  0x00
  return
stop

sub_0: assembly {
        /* "contract.sol":0:86  contract C {... */
      mstore(0x40, 0x80)
      callvalue
      dup1
      iszero
      tag_1
      jumpi
      0x00
      dup1
      revert
    tag_1:
      pop
      jumpi(tag_2, lt(calldatasize, 0x04))
      shr(0xe0, calldataload(0x00))
      dup1
      0x901717d1
      eq
      tag_3
      jumpi
    tag_2:
      0x00
      dup1
      revert
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
    tag_3:
      tag_4
      tag_5
      jump  // in
    tag_4:
      mload(0x40)
      tag_6
      swap2
      swap1
      tag_7
      jump  // in
    tag_6:
      mload(0x40)
      dup1
```

(续下页)

(接上页)

```
      swap2
      sub
      swap1
      return
    tag_5:
        /* "contract.sol":53:57  uint */
      0x00
        /* "contract.sol":76:77  1 */
      0x01
        /* "contract.sol":69:77  return 1 */
      swap1
      pop
        /* "contract.sol":17:84  function one() public pure returns (uint) {... */
      swap1
      jump  // out
        /* "#utility.yul":7:125   */
    tag_10:
        /* "#utility.yul":94:118   */
      tag_12
        /* "#utility.yul":112:117   */
      dup2
        /* "#utility.yul":94:118   */
      tag_13
      jump  // in
    tag_12:
        /* "#utility.yul":89:92   */
      dup3
        /* "#utility.yul":82:119   */
      mstore
        /* "#utility.yul":72:125   */
      pop
      pop
      jump  // out
        /* "#utility.yul":131:353   */
    tag_7:
      0x00
        /* "#utility.yul":262:264   */
      0x20
        /* "#utility.yul":251:260   */
      dup3
        /* "#utility.yul":247:265   */
      add
        /* "#utility.yul":239:265   */
```

(续下页)

(接上页)

```
      swap1
      pop
        /* "#utility.yul":275:346   */
      tag_15
        /* "#utility.yul":343:344   */
      0x00
        /* "#utility.yul":332:341   */
      dup4
        /* "#utility.yul":328:345   */
      add
        /* "#utility.yul":319:325   */
      dup5
        /* "#utility.yul":275:346   */
      tag_10
      jump  // in
    tag_15:
        /* "#utility.yul":229:353   */
      swap3
      swap2
      pop
      pop
      jump  // out
        /* "#utility.yul":359:436   */
    tag_13:
      0x00
        /* "#utility.yul":425:430   */
      dup2
        /* "#utility.yul":414:430   */
      swap1
      pop
        /* "#utility.yul":404:436   */
      swap2
      swap1
      pop
      jump  // out

    auxdata:␣
↪0xa2646970667358221220a5874f19737ddd4c5d77ace1619e5160c67b3d4bedac75fce908fed32d98899864736f6c63782
}
```

另外，上述输出也可以从 Remix ，在编译合约后的” 编译细节” 选项下获得。

请注意，asm 输出以创建/构造器代码开始。部署代码是作为子对象的一部分提供的（在上面的例子中，它是子对象 sub_0 的一部分）。auxdata 字段对应于合约元数据 。汇编输出中的注释指向源文

件的位置。注意 `#utility.yul` 是一个内部生成的实用函数文件，可以使用标志 `--combined-json generated-sources,generated-sources-runtime` 获得。

类似地，可以通过 `solc --optimize --asm contract.sol` 命令获得优化后的程序集。通常情况下，观察两个不同的 Solidity 源是否会产生相同的优化代码是很有趣的。例如，查看表达式 `(a * b) / c`，`a * b / c` 是否生成相同的字节码。在可能的话，在剥离引用源位置的注释之后，通过获取相应程序集输出的 `diff` 很容易做到这一点。

备注： `--asm` 的输出不是设计成机器可读的。因此，在 solc 的各个小版本之间，输出可能会有重大的变化。

3.15 基于 Solidity 中间表征的 Codegen 变化

Solidity 可以通过两种不同的方式生成 EVM 字节码：要么直接从 Solidity 到 EVM 操作码（"旧编码"），要么通过在 Yul 中的中间表示法（"IR"）（"新编码" 或 "基于 IR 的编码"）。

引入基于 IR 的代码生成器的目的是，不仅使代码生成更加透明和可审计，而且能够实现更强大的跨函数的优化通道。

您可以在命令行中使用 `--via-ir` 或在 standard-json 中使用 `{"viaIR": true}` 选项来启用它，我们鼓励大家尝试一下！

由于一些原因，旧的和基于 IR 的代码生成器之间存在着微小的语义差异，主要是在那些我们无论如何也不会期望人们依赖这种行为的领域。本节强调了旧的和基于 IR 的代码生成器之间的主要区别。

3.15.1 仅有语义上的变化

本节列出了仅有语义的变化，从而有可能在现有的代码中隐藏新的和不同的行为。

- 在继承的情况下，状态变量初始化的顺序已经改变。

  以前的顺序是：

  - 所有的状态变量在开始时都被零初始化。
  - 从最终派生合约到最基础的合约评估基础构造函数参数。
  - 从最基础的继承关系到最终派生的继承关系初始化整个继承层次结构中的所有状态变量。
  - 如果存在，在线性化层次结构中从最基础的合约到最终派生的合约依次运行构造函数。

  新的顺序：

  - 所有的状态变量在开始时都被零初始化。
  - 从最终派生合约到最基础的合约评估基础构造函数参数。
  - 对于每一个合约，按照从最基础到最终派生的合约的线性化层次结构的顺序执行：

1. 初始化状态变量。
2. 运行构造函数（如果存在）。

这导致了合约中的差异，即一个状态变量的初始值依赖于另一个合约中构造函数的结果：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract A {
    uint x;
    constructor() {
        x = 42;
    }
    function f() public view returns(uint256) {
        return x;
    }
}
contract B is A {
    uint public y = f();
}
```

以前，y 会被设置为 0。这是由于我们会先初始化状态变量：首先，x 被设置为 0，当初始化 y 时，f() 将返回 0，导致 y 也为 0。在新的规则下，y 将被设置为 42。我们首先将 x 初始化为 0，然后调用 A 的构造函数，将 x 设置为 42。最后，在初始化 y 时，f() 返回 42，导致 y 为 42。

- 当存储结构被删除时，包含该结构成员的每个存储槽都被完全设置为零。以前，填充空间是不被触动的。因此，如果结构中的填充空间被用来存储数据（例如在合约升级的背景下），您必须注意，delete 现在也会清除添加的成员（而在过去不会被清除）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1;

contract C {
    struct S {
        uint64 y;
        uint64 z;
    }
    S s;
    function f() public {
        // ...
        delete s;
        // s只占用了32个字节槽的前16个字节
        // delete 将把零写到完整的插槽中
    }
}
```

我们对隐式删除也有同样的行为，例如当结构体的数组被缩短时。

- 关于函数参数和返回变量，函数修改器的实现方式略有不同。如果占位符 `_;` 在一个修饰符中被多次使用，这尤其有影响。在旧的代码生成器中，每个函数参数和返回变量在堆栈中都有一个固定的槽。如果因为多次使用 `_;` 而使函数运行多次，或者在一个循环中使用，那么函数参数或返回变量的值的变化在函数的下一次执行中是可见的。新的代码生成器使用实际的函数来实现修改器，并将函数参数传递下去。这意味着对一个函数主体的多次使用将得到相同的参数值，而对返回变量的影响是，它们在每次执行时都被重置为其默认值（零）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0;
contract C {
    function f(uint a) public pure mod() returns (uint r) {
        r = a++;
    }
    modifier mod() { _; _; }
}
```

如果您在旧的代码生成器中执行 `f(0)`，它将返回 `1`，而在使用新的代码生成器时，它将返回 `0`。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract C {
    bool active = true;
    modifier mod()
    {
        _;
        active = false;
        _;
    }
    function foo() external mod() returns (uint ret)
    {
        if (active)
            ret = 1; // 与 ``return 1`` 相同
    }
}
```

函数 `C.foo()` 返回以下值：

  - 旧的代码生成器：`1` 作为返回变量在第一次 `_;` 使用前只被初始化为 `0`，然后被 `return 1;` 覆盖。在第二次 `_;` 使用时，它没有被再次初始化，而且 `foo()` 也没有明确地分配给它（由于 `active == false`），因此它保持了它的第一个值。
  - 新的代码生成器：`0` 作为所有参数，包括返回参数，将在每次 `_;` 使用前被重新初始化。

- 对于旧的代码生成器，表达式的评估顺序是没有规定的。对于新的代码生成器，我们试图按照源顺序

(从左到右) 进行评估，但并不保证这一点。这可能会导致语义上的差异。

例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function preincr_u8(uint8 a) public pure returns (uint8) {
        return ++a + a;
    }
}
```

函数 preincr_u8(1) 返回以下值：

- 旧的代码生成器：3 (1 + 2)，但一般情况下返回值是不指定的
- 新的代码生成器：4 (2 + 2)，但不能保证返回值

另一方面，除了全局函数 addmod 和 mulmod 外，两个代码生成器对函数参数表达式的评估顺序是一样的。例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function add(uint8 a, uint8 b) public pure returns (uint8) {
        return a + b;
    }
    function g(uint8 a, uint8 b) public pure returns (uint8) {
        return add(++a + ++b, a + b);
    }
}
```

函数 g(1, 2) 返回以下值：

- 旧的代码生成器：10 (add(2+3, 2+3))，但返回值一般不指定。
- 新的代码生成器：10，但不能保证返回值

全局函数 addmod 和 mulmod 的参数由旧代码生成器从右向左评估，新代码生成器从左向右评估。例如：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f() public pure returns (uint256 aMod, uint256 mMod) {
        uint256 x = 3;
        // 旧的代码生成器： add/mulmod(5, 4, 3)
        // 新的代码生成器： add/mulmod(4, 5, 5)
```

(续下页)

(接上页)

```
        aMod = addmod(++x, ++x, x);
        mMod = mulmod(++x, ++x, x);
    }
}
```

函数 `f()` 返回以下值:

  - 旧的代码生成器: `aMod = 0` 和 `mMod = 2`
  - 新的代码生成器: `aMod = 4` 和 `mMod = 0`

- 新的代码生成器对自由内存指针施加了一个硬性限制 `type(uint64).max` (`0xffffffffffffffff`) 。其增加值超过这个限制的分配会被恢复。旧的代码生成器没有这个限制。

  例如:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.8.0;
contract C {
    function f() public {
        uint[] memory arr;
        // 分配空间: 576460752303423481
        // 假设freeMemPtr最初指向0x80
        uint solYulMaxAllocationBeforeMemPtrOverflow = (type(uint64).max - 0x80 -
↪31) / 32;
        // freeMemPtr 因 UINT64_MAX 限制溢出
        arr = new uint[](solYulMaxAllocationBeforeMemPtrOverflow);
    }
}
```

  函数 `f()` 的作用如下:

  - 旧的代码生成器: 在大内存分配后对数组内容进行清零时耗尽了 gas
  - 新的代码生成器: 由于自由内存指针溢出而还原 (不会耗尽 gas)。

3.15.2 内部结构

内部函数指针

旧的代码生成器对内部函数指针的值使用代码偏移量或标签。这一点特别复杂，因为这些偏移量在构造时和部署后是不同的，而且这些值可以通过存储跨越这个边界。正因为如此，这两个偏移量在构造时被编码为同一个值 (进入不同的字节)。

在新的代码生成器中，函数指针使用依次分配的内部 ID。由于通过跳转的调用是不可能的，通过函数指针的调用总是要使用内部调度函数，使用 `switch` 语句来选择正确的函数。

ID 0 是为未初始化的函数指针保留的，这些指针在被调用时，会引起调度函数的 panic 错误。

在旧的代码生成器中，内部函数指针是用一个特殊的函数初始化的，它总是引起 panic 错误。这导致在构造时对存储中的内部函数指针进行存储写入。

清理

旧的代码生成器只在操作前执行清理，而操作的结果可能会受到脏位值的影响。新的代码生成器在任何可能导致脏位的操作之后执行清理。我们希望优化器能够强大到足以消除多余的清理操作。

例如:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;
contract C {
    function f(uint8 a) public pure returns (uint r1, uint r2)
    {
        a = ~a;
        assembly {
            r1 := a
        }
        r2 = a;
    }
}
```

函数 f(1) 返回以下值:

- 旧的代码生成器:(fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe, 00000000000000000000000000000000000000000000000000000000000000fe)
- 新的代码生成器:(00000000000000000000000000000000000000000000000000000000000000fe, 00000000000000000000000000000000000000000000000000000000000000fe)

请注意，与新的代码生成器不同，旧的代码生成器在位取反赋值（a = ~a）后没有进行清理。这导致新旧代码生成器之间对返回值 r1 的赋值（在内联汇编块内）不同。然而，两个代码生成器在 a 的新值被分配到 r2 之前都进行了清理。

3.16 存储中的状态变量储存结构

合约的状态变量以一种紧凑的方式存储，这样多个值有时会使用同一个存储槽。除了动态大小的数组和映射（见下文）之外，数据是被逐项存储的，从第一个状态变量开始，它被存储在槽 0 中。对于每个变量，根据它的类型确定一个字节的大小。如果可能的话，需要少于 32 字节的多个连续项目被打包到一个存储槽中，根据以下规则:

- 存储插槽的第一项会以低位对齐（即右对齐）的方式储存。

- 值类型只使用存储它们所需的字节数。
- 如果一个值类型不适合一个存储槽的剩余部分，它将被存储在下一个存储槽。
- 结构和数组数据总是从一个新的存储槽开始，它们的项根据这些规则被紧密地打包。
- 结构或数组数据之后的变量总是开辟一个新的存储槽。

对于使用继承的合约，状态变量的排序是从最基础的合约开始，由合约的 C3 线性化顺序决定的。如果上述规则允许，来自不同合约的状态变量确实共享同一个存储槽。

结构体和数组中的元素都是顺序存储的，就像它们被明确给定的那样。

警告： 当使用小于 32 字节的元素时，您的合约的气体用量可能会更高。这是因为 EVM 每次对 32 字节进行操作。因此，如果元素小于这个大小，EVM 必须使用更多的操作，以便将元素的大小从 32 字节减少到所需的大小。

如果您处理的是存储值，使用缩小尺寸的类型可能是有益的，因为编译器会将多个元素打包到一个存储槽中，从而将多个读或写合并到一个操作中。如果您不是在同一时间读取或写入一个槽中的所有值，这可能会产生相反的效果，虽然。当一个值被写入一个多值存储槽时，该存储槽必须先被读取，然后与新值结合，这样同一槽中的其他数据就不会被破坏。

在处理函数参数或内存值时，因为编译器不会打包这些值，所以没有什么好处，

最后，为了让 EVM 对此进行优化，确保您的存储变量和 `struct` 成员的顺序，使它们能够被紧密地包装起来。例如，按照 `uint128, uint128, uint256` 的顺序声明您的存储变量，而不是 `uint128, uint256, uint128`，因为前者只占用两个存储槽，而后者则占用三个存储槽。

备注： 由于存储指针可以传递给库，存储中的状态变量的结构被认为是 Solidity 外部接口的一部分。这意味着对这一节中概述的规则的任何改变都被认为是对语言的重大改变，由于它的关键性质，在执行之前应该非常仔细地考虑。在发生这种重大变化的情况下，我们希望发布一种兼容模式，在这种模式下，编译器将生成支持旧结构的字节码。

3.16.1 映射和动态数组

由于映射和动态数组的大小是不可预知的，他们不能被存储在其前后的状态变量之间。相反，它们被认为只占用 32 个字节，与上述规则 有关，它们所包含的元素被存储在一个不同的存储槽，该存储槽是用 Keccak-256 哈希计算的。

假设映射或数组的存储位置在适应了存储结构规则 后，最终位于一个槽 `p`。对于动态数组，这个槽存储了数组中的元素数量（字节数组和字符串是一个例外，参见下文）。对于映射来说，这个槽保持空的状态，但是仍然需要它来确保即使有两个映射相邻，它们的内容最终也是在不同的存储位置。

数组数据从 `keccak256(p)` 开始，它的排列方式与静态大小的阵列数据相同：一个元素接着一个元素，如

果元素的长度不超过 16 字节，就有可能共享存储槽。包含动态数组的动态数组递归地应用这一规则。元素 `x[i][j]` 的位置为，其中 `x` 的类型是 `uint24[][]`，计算方法如下（同样，假设 `x` 本身存储在槽 `p`）：槽是 `keccak256(keccak256(p)+i)+ floor(j / floor(256 / 24))`，元素可以从槽数据 `v` 得到，使用 `(v >> ((j % floor(256 / 24)) * 24)) & type(uint24).max`。

对应于映射键 `k` 的值位于 `keccak256(h(k) . p)`，其中 `.` 是连接符，`h` 是一个函数，根据键的类型应用于键。

- 对于值类型，函数 `h` 将与在内存中存储值的相同方式来将值填充为 32 字节。
- 对于字符串和字节数组，`h(k)` 只是未填充的数据。

如果映射类型的值是一个非值类型，则计算的槽会标记为数据的开始位置。例如，如果值是结构体类型，您必须添加一个与结构体成员相对应的偏移量才能访问到该成员。

作为示例，参考以下合约：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;


contract C {
    struct S { uint16 a; uint16 b; uint256 c; }
    uint x;
    mapping(uint => mapping(uint => S)) data;
}
```

让我们计算一下 `data[4][9].c` 的存储位置。映射本身的位置是 `1`（变量 `x` 前面有 32 字节）。这意味着 `data[4]` 存储在 `keccak256(uint256(4) . uint256(1))`。`data[4]` 的类型还是一个映射，`data[4][9]` 的数据从 `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1)))` 槽开始。成员 `c` 在结构 `S` 中的槽位偏移是 `1`，因为 `a` 和 `b` 被装在一个槽位中。这意味着 `data[4][9].c` 的插槽是 `keccak256(uint256(9) . keccak256(uint256(4) . uint256(1))) + 1`。该值的类型是 `uint256`，所以它占用一个槽。

bytes 和 string

`bytes` 和 `string` 的编码是相同的。一般来说，编码与 `bytes1[]` 类似，即有一个槽用于存放数组本身和一个数据区，这个数据区是用该槽的位置的 `keccak256` 哈希值计算的。然而，对于较短的值（短于 32 字节），数组元素与长度一起存储在同一个槽中。

特别是：如果数据最多只有 `31` 字节长，元素被存储在高阶字节中（左对齐），最低阶字节存储值 `length * 2`。对于存储数据长度为 `32` 或更多字节的字节数，主槽 `p` 存储 `length * 2 + 1`，数据照常存储在 `keccak256(p)`。这意味着您可以通过检查最低位是否被设置来区分短数组和长数组：短数组（未设置）和长数组（设置）。

备注： 目前不支持处理无效编码的槽，但将来可能会加入。如果您通过 IR 进行编译，读取一个无效编码的

槽会导致 Panic(0x22) 错误。

3.16.2 JSON 输出

合约的存储结构可以通过标准的 *JSON* 接口 请求获得。输出的是一个 JSON 对象，包含两个键，storage 和 types。storage 对象是一个数组，每个元素都有以下形式：

```
{
    "astId": 2,
    "contract": "fileA:A",
    "label": "x",
    "offset": 0,
    "slot": "0",
    "type": "t_uint256"
}
```

上面的例子来自源于项目 fileA 的 contract A { uint x; } 的存储结构，并且

- astId 是状态变量声明的 AST 节点的 ID
- contract 是合约的名称，包括其路径作为前缀
- label 是状态变量的名称
- offset 是根据编码在存储槽中的字节偏移量
- slot 是状态变量所在或开始的存储槽。这个数字可能非常大，因此它的 JSON 值被表示为一个字符串
- type 是一个标识符，作为变量类型信息的关键（如下所述）

给定的 type，在这里是 t_uint256，代表 types 中的一个元素，它的形式是：

```
{
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32",
}
```

这里

- encoding 数据在存储中是如何编码的，可能的值是：
  - inplace：数据在存储中是连续布置的（参见上文）。
  - mapping：基于 Keccak-256 的哈希方法（参见上文）。
  - dynamic_array：基于 Keccak-256 的哈希方法（参见上文）。
  - bytes：单槽或基于 Keccak-256 哈希值，取决于数据大小（参见上文）。

- label 是典型的类型名称。
- numberOfBytes 是使用的字节数（十进制字符串）。注意，如果 numberOfBytes > 32 这意味着使用了一个以上的槽。

除了上述四种类型外，有些类型还有额外的信息。映射包含它的 key 和 value 类型（再次引用这个类型映射中的一个项），数组有它的 base 类型，结构体会列出它们的 成员，其格式与高层次的 storage 相同（参见上文）。

备注： 合约的存储结构的 JSON 输出格式仍被认为是实验性的，并且在 Solidity 的非重大版本中会有变化。

下面的例子显示了一个合约及其存储结构，包含值类型和引用类型，被编码打包的类型，以及嵌套的类型。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;
contract A {
    struct S {
        uint128 a;
        uint128 b;
        uint[2] staticArray;
        uint[] dynArray;
    }

    uint x;
    uint y;
    S s;
    address addr;
    mapping(uint => mapping(address => bool)) map;
    uint[] array;
    string s1;
    bytes b1;
}
```


```
{
  "storage": [
    {
      "astId": 15,
      "contract": "fileA:A",
      "label": "x",
      "offset": 0,
      "slot": "0",
      "type": "t_uint256"
    },
    {
```


（续下页）

(接上页)

```
      "astId": 17,
      "contract": "fileA:A",
      "label": "y",
      "offset": 0,
      "slot": "1",
      "type": "t_uint256"
    },
    {
      "astId": 20,
      "contract": "fileA:A",
      "label": "s",
      "offset": 0,
      "slot": "2",
      "type": "t_struct(S)13_storage"
    },
    {
      "astId": 22,
      "contract": "fileA:A",
      "label": "addr",
      "offset": 0,
      "slot": "6",
      "type": "t_address"
    },
    {
      "astId": 28,
      "contract": "fileA:A",
      "label": "map",
      "offset": 0,
      "slot": "7",
      "type": "t_mapping(t_uint256,t_mapping(t_address,t_bool))"
    },
    {
      "astId": 31,
      "contract": "fileA:A",
      "label": "array",
      "offset": 0,
      "slot": "8",
      "type": "t_array(t_uint256)dyn_storage"
    },
    {
      "astId": 33,
      "contract": "fileA:A",
      "label": "s1",
```

(续下页)

(接上页)

```
      "offset": 0,
      "slot": "9",
      "type": "t_string_storage"
    },
    {
      "astId": 35,
      "contract": "fileA:A",
      "label": "b1",
      "offset": 0,
      "slot": "10",
      "type": "t_bytes_storage"
    }
  ],
  "types": {
    "t_address": {
      "encoding": "inplace",
      "label": "address",
      "numberOfBytes": "20"
    },
    "t_array(t_uint256)2_storage": {
      "base": "t_uint256",
      "encoding": "inplace",
      "label": "uint256[2]",
      "numberOfBytes": "64"
    },
    "t_array(t_uint256)dyn_storage": {
      "base": "t_uint256",
      "encoding": "dynamic_array",
      "label": "uint256[]",
      "numberOfBytes": "32"
    },
    "t_bool": {
      "encoding": "inplace",
      "label": "bool",
      "numberOfBytes": "1"
    },
    "t_bytes_storage": {
      "encoding": "bytes",
      "label": "bytes",
      "numberOfBytes": "32"
    },
    "t_mapping(t_address,t_bool)": {
      "encoding": "mapping",
```

(续下页)

(接上页)


```
      "key": "t_address",
      "label": "mapping(address => bool)",
      "numberOfBytes": "32",
      "value": "t_bool"
    },
    "t_mapping(t_uint256,t_mapping(t_address,t_bool))": {
      "encoding": "mapping",
      "key": "t_uint256",
      "label": "mapping(uint256 => mapping(address => bool))",
      "numberOfBytes": "32",
      "value": "t_mapping(t_address,t_bool)"
    },
    "t_string_storage": {
      "encoding": "bytes",
      "label": "string",
      "numberOfBytes": "32"
    },
    "t_struct(S)13_storage": {
      "encoding": "inplace",
      "label": "struct A.S",
      "members": [
        {
          "astId": 3,
          "contract": "fileA:A",
          "label": "a",
          "offset": 0,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 5,
          "contract": "fileA:A",
          "label": "b",
          "offset": 16,
          "slot": "0",
          "type": "t_uint128"
        },
        {
          "astId": 9,
          "contract": "fileA:A",
          "label": "staticArray",
          "offset": 0,
          "slot": "1",
```


(续下页)

(接上页)

```
        "type": "t_array(t_uint256)2_storage"
      },
      {
        "astId": 12,
        "contract": "fileA:A",
        "label": "dynArray",
        "offset": 0,
        "slot": "3",
        "type": "t_array(t_uint256)dyn_storage"
      }
    ],
    "numberOfBytes": "128"
  },
  "t_uint128": {
    "encoding": "inplace",
    "label": "uint128",
    "numberOfBytes": "16"
  },
  "t_uint256": {
    "encoding": "inplace",
    "label": "uint256",
    "numberOfBytes": "32"
  }
 }
}
```

3.17 内存中的存储结构

Solidity 保留了四个 32 字节的插槽，具体的字节范围（包括端点）使用如下：

- `0x00 - 0x3f` （64 字节）：用于哈希方法的临时空间
- `0x40 - 0x5f` （32 字节）：当前分配的内存大小（又称空闲内存指针）。
- `0x60 - 0x7f` （32 字节）：0 值插槽

临时空间可以在语句之间使用（即在内联汇编之中）。0 值插槽则用来对动态内存数组进行初始化，且永远不会写入数据（因而可用的初始内存指针为 `0x80`）。

Solidity 总会把新对象保存在空闲内存指针的位置，所以这段内存实际上从来不会空闲（在未来可能会修改这个机制）。

Solidity 中内存数组中的元素总是占据 32 字节的倍数（对于 `bytes1[]` 来说也是如此，但对于 `bytes` 和 `string` 来说不是这样）。多维内存数组是指向内存数组的指针. 一个动态数组的长度被存储在数组的第一个槽里，后面是数组元素。

> **警告：** 在 Solidity 中，有一些操作需要一个大于 64 字节的临时内存区域，因此将不适合放在默认的临时空间中。它们将被放置在空闲内存指向的位置，但由于这种数据的生命周期较短，这个指针不会即时更新。这部分内存可能会被清零也可能不会。所以我们不应该期望这些所谓的空闲内存总会被清零。
>
> 虽然使用 `msize` 来到达一个绝对清零的内存区域似乎是个好主意，但在不更新空闲内存指针的情况下，非临时性地使用这样的指针会产生意想不到的结果。

3.17.1 与存储结构的区别

如上所述，内存中的存储结构与*存储（storage）* 中的存储结构是不同的。下面是一些例子。

在数组中的差异的例子

下面的数组在存储中占用 32 字节（1 个槽），但在内存中占用 128 字节（4 项，每个 32 字节）。

```
uint8[4] a;
```

在结构体中存储结构差异的例子

以下结构体在存储中占用 96 字节（3 个 32 字节的槽），但在内存中占用 128 字节（4 项，每个 32 字节）。

```
struct S {
    uint a;
    uint b;
    uint8 c;
    uint8 d;
}
```

3.18 调用数据的存储结构

一个函数调用的输入数据的格式被认为会遵循*ABI 规范* 所定义的格式。其中，ABI 规范要求参数被填充为 32 字节的倍数。而内部函数调用会使用不同规则。

合约的构造函数的参数直接附加在合约的字节码末尾，也是 ABI 编码的。构造函数将通过一个硬编码的偏移量来访问它们，而不是通过使用 `codesize` 操作码，因为在向代码追加数据时它会发生改变。

3.19 清理变量

最终，EVM 中的所有数值都被存储在 256 位的字中。因此，在某些情况下，当一个值的类型少于 256 比特时，有必要清理剩余的比特位。Solidity 编译器被设计为在任何可能受到剩余位中潜在垃圾的不利影响的操作之前执行这样的清理。例如，在将一个值写入内存之前，剩余的位需要被清除，因为内存的内容可以被用来计算哈希值或作为消息调用的数据发送。同样地，在将一个值存储到存储器中之前，剩余的位需要被清理，否则就会看到被混淆的数值。

注意，通过内联汇编的访问不被认为是这种操作。如果您使用内联汇编来访问短于 256 位的 Solidity 变量，编译器不保证该值被正确清理。

此外，如果接下来的操作不受影响，我们就不清理这些位。例如，由于任何非零值都被 `JUMPI` 指令认为是 `true`，所以在布尔值被用作 `JUMPI` 的条件之前，我们不对它们进行清理。

除了上面的设计原则外，Solidity 编译器在输入数据被加载到堆栈时也会对其进行清理。

下表描述了适用于不同类型的清理规则，其中 高位指的是在类型少于 256 位的情况下的剩余位。

类型	有效的值	清理无效值
n 个成员的枚举	0，直到 n - 1	抛出异常
布尔	0 或 1	结果为 1
有符号整数	高位设置为符号位	目前默许符号化扩展到一个有效值即所有高位被设置为符号位；将来可能会抛出一个异常。
无符号整数	高位被清零	目前默许为有效值，即所有高位被设置为零；将来可能会抛出一个异常。

请注意，有效和无效的值取决于其类型大小。假设类型为 `uint8`，即无符号的 8 位类型，它有以下有效值：

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 1111 1111
```

任何无效的值都将把高位设置为零：

```
0101...1101 0010 1010   无效的值
0000...0000 0010 1010   清理过的值
```

对于 `int8`，即有符号的 8 位类型，有效值是：

负数

```
1111...1111 1111 1111
1111...1111 1111 1110
```

(续下页)

(接上页)

```
....
1111...1111 1000 0000
```

正数

```
0000...0000 0000 0000
0000...0000 0000 0001
0000...0000 0000 0010
....
0000...0000 1111 1111
```

编译器将 符号化扩展（`signextend`）符号位，即负值为 1，正值为 0，覆盖高位：

负数

```
0010...1010 1111 1111   无效的值
1111...1111 1111 1111   清理过的值
```

正数

```
1101...0101 0000 0100   无效的值
0000...0000 0000 0100   清理过的值
```

3.20 源代码映射

作为 AST 输出的一部分，编译器提供了 AST 中相应节点所代表的源代码的范围。这可以用于各种目的，包括基于 AST 报告错误的静态分析工具和突出局部变量及其用途的调试工具。

此外，编译器还可以生成从字节码到生成该指令的源代码范围的映射。这对于在字节码层次上操作的静态分析工具和在调试器中显示源代码中的当前位置或断点处理来说，也是很重要的。这个映射还包含其他信息，如跳转类型和修改器深度（见下文）。

这两种源码映射都使用整数标识符来指代源码文件。源文件的标识符存储在 `output['sources'][sourceName]['id']` 中，其中 `output` 是标准 json 编译器接口的输出，被解析成 JSON。对于一些实用程序，编译器会生成" 内部" 源文件，这些文件不是原始输入的一部分，而是从源映射中引用的。这些源文件及其标识符可以通过 `output['contracts'][sourceName][contractName]['evm']['bytecode']['generatedSources']` 获得。

备注： 如果指令没有与任何特定的源文件相关联，源映射将分配一个整数标识符 `-1`。这可能发生在编译器生成的内联汇编语句所产生的字节码部分。

AST 内部的源映射使用以下符号:

`s:l:f`

其中，`s` 是源代码文件中范围起始处的字节偏移量，`l` 是以字节为单位的源代码范围的长度，`f` 是上述源代码索引。

源码映射中的字节码的编码更为复杂。它是一个由 `s:l:f:j:m` 组成的列表，用 `;` 分隔。这些元素中的每一个都对应着一条指令，也就是说，您不能使用字节偏移量，而必须使用指令偏移量（推送指令比单个字节长）。字段 `s`，`l` 和 `f` 同上。`j` 可以是 `i`，`o` 或 `-`，表示跳转指令是进入一个函数，从一个函数返回，还是作为一个循环的一部分的普通跳转。最后一个字段，`m`，是一个整数，表示" 修改器深度"。当占位符语句 (`_`) 进入修改器时，这个深度会增加，当它再次离开时，深度会减少。这使得调试器可以跟踪一些棘手的情况，如同一个修改器被使用两次，或在一个修改器中使用多个占位符语句。

为了压缩这些源码映射，特别是字节码的源码映射，使用了以下规则:

- 如果一个字段为空，则使用前一个元素的值。
- 如果缺少 `:`，以下所有字段都被认为是空的。

这意味着下面的源码映射代表了相同的信息:

`1:2:1;1:9:1;2:1:2;2:1:2;2:1:2`

`1:2:1;:9;2:1:2;;`

需要注意的是，当使用逐字（*verbatim*）内建程序时，源码映射将是无效的。内建程序被认为是一条指令，而不是潜在的多条指令。

3.21 优化器

Solidity 编译器使用两种不同的优化器模块。在操作码水平上操作的" 旧" 优化器和在 Yul IR 代码上操作的"新" 优化器。

基于操作码的优化器对操作码应用一套 简化规则。它还结合了相等的代码集并删除了未使用的代码。

基于 Yul 的优化器要强大得多，因为它可以跨函数调用工作。例如，任意跳转在 Yul 中是不可能的，所以有可能计算每个函数的副作用。假设有两个函数调用，其中第一个不修改存储，第二个修改存储。如果它们的参数和返回值不相互依赖，我们就可以对函数调用重新排序。同样地，如果一个函数是没有副作用的，而且其结果是乘以 0 的，就可以完全删除该函数调用。

目前，参数 `--optimize` 会为生成的字节码激活基于操作码的优化器，并为内部生成的 Yul 代码激活 Yul 优化器，例如当使用 ABI coder v2 时。您可以使用 `solc --ir optimized --optimize` 来为 Solidity 源码产生一个优化的 Yul IR。同样地，您可以使用 `solc --strict-assembly --optimize` 来产生一个独立的 Yul 模式。

备注: 窥视孔（peephole）优化器 和内联器总是默认启用的，只能通过标准 *JSON* 文件配置 关闭。

您可以在下面找到关于这两个优化器模块及其优化步骤的更多细节。

3.21.1 优化 Solidity 代码的好处

总的来说，优化器试图简化复杂的表达式，从而减少代码大小和执行成本，也就是说，它可以减少部署合约以及对合约进行外部调用所需的气体。它还会对函数进行专业化或内联化优化。特别是当函数内联一个可能导致更大的代码操作时，它经常这样做，因为这导致了更多简化的机会。

3.21.2 优化和非优化代码之间的差异

一般来说，最明显的区别是常量表达式在编译时被评估。当涉及到 ASM 输出时，人们也可以注意到等价或重复的代码块的减少（比较 `--asm` 和 `--asm --optimize` 标志的输出）。然而，当涉及到 Yul/中间代表时，可能会有明显的差异，例如，函数可能被内联，合并或重写以消除冗余等等（比较带有 `--ir` 和 `--optimize --ir-optimized` 标志的输出）。

3.21.3 优化器参数运行

运行次数（`--optimize-runs`）大致规定了在合约有效期内，所部署的代码的每个操作码被执行的频率。这意味着它是代码大小（部署成本）和代码执行成本（部署后的成本）之间的一个折衷参数。一个"运行"参数为"1"将产生简短的合约但昂贵的执行代码。相反，一个较大的"运行"参数将产生较大的合约但更省气体的执行代码。该参数的最大值为 `2**32-1`。

备注： 一个常见的误解是，这个参数指定了优化器的迭代次数。这是不正确的。优化器将始终运行尽可能多的次数来改进代码。

3.21.4 基于操作码的优化器模块

基于操作码的优化器模块对汇编代码进行操作。它在 `JUMPs` 和 `JUMPDESTs` 之间将指令序列分成基本块。在这些块中，优化器分析指令，并将对堆栈、内存或存储的每一次修改记录为一个表达式，该表达式由一条指令和一列参数组成，这些参数是指向其他表达式的指针。

此外，基于操作码的优化器使用了一个名为"通用子表达式消除器"的组件，它除其他任务外，还能找到总是相等的表达式(在每个输入上)，并将它们合并为一个表达式类。它首先尝试在一个已经知道的表达式列表中找到每个新的表达式。如果没有找到这样的匹配，它就根据 `constant + constant = sum_of_constants` 或 `X * 1 = X` 这样的规则简化表达式。由于这是一个递归过程，如果第二个因素是一个更复杂的表达式，并且知道这个表达式的值总是为 1，我们也可以应用后一个规则。

某些优化器步骤象征性地跟踪存储和内存位置。例如，这些信息被用来计算 Keccak-256 哈希值，可以在编译时进行评估。考虑一下这个序列：

```
PUSH 32
PUSH 0
CALLDATALOAD
PUSH 100
DUP2
MSTORE
KECCAK256
```

或着等同于 Yul 为

```
let x := calldataload(0)
mstore(x, 100)
let value := keccak256(x, 32)
```

在这种情况下，优化器跟踪位于内存位置 `calldataload(0)` 的值，然后意识到 Keccak-256 哈希值可以在编译时被评估。这只有在 `mstore` 和 `keccak256` 之间没有其他指令修改内存时才有效。因此，如果有一条指令写到内存（或存储），那么我们需要擦除对当前内存（或存储）的记忆。然而，这种擦除有一个例外，当我们可以很容易地看到指令没有写到某个位置。

示例，

```
let x := calldataload(0)
mstore(x, 100)
// 已知当前内存位置x -> 100
let y := add(x, 32)
// 没有清除 x -> 100 的记忆，因为y并没有写到[x, x+32)。
mstore(y, 200)
// 现在可以对这个Keccak-256进行计算了
let value := keccak256(x, 32)
```

因此，对存储和内存位置的修改，比如说位置 `l`，必须擦除关于可能等于 `l` 的存储或内存位置的记忆。更具体地说，对于存储，优化器必须删除所有可能等于 `l` 的符号位置的记忆，对于内存，优化器必须删除所有可能不超过 32 字节的符号位置的记忆。如果 `m` 表示一个任意的位置，那么这个擦除的决定是通过计算 `sub(l, m)` 的值来完成。对于存储，如果这个值被评估为一个非零的值，那么关于 `m` 的记忆将被保留。对于内存，如果这个值被评估为一个介于 `32` 和 `2**256 - 32` 之间的值，那么关于 `m` 的记忆将被保留。在所有其他情况下，关于 `m` 的记忆将被删除。

并且有一个对内存和存储的修改列表。这些信息与基本代码块一起存储并用来链接它们。此外，关于堆栈、存储和内存配置的记忆被转发给下一个（几个）块。

如果我们知道所有 `JUMP` 和 `JUMPI` 指令的目标，我们就可以构建一个完整的程序流程图。如果只有一个我们不知道的目标（原则上可能发生，跳转目标可以基于输入来计算），我们必须消除关于代码块输入状态的所有信息，因为它可能是未知的 `JUMP` 目标。如果一个 `JUMPI` 的条件等于一个常量，它将被转换为无条件跳转。

作为最后一步，每个块中的代码都会被完全重新生成。然后优化器会从代码块的结尾处在栈上的表达式开始创建依赖关系图，且不是该图组成部分的每个操作都会被丢弃。这样生成的代码将按照原始代码中的顺序对

内存和存储进行修改（舍弃不需要的修改）。最后，它生成了所有需要在堆栈中的正确位置的值。

这些步骤适用于每个基本代码块，如果代码块较小，则新生成的代码将用作替换。如果一个基本代码块在 JUMPI 处被分割，且在分析过程中被评估为一个常数，则会根据常量的值来替换 JUMPI，因此，类似于

```
uint x = 7;
data[7] = 9;
if (data[x] != x + 2) // 这个条件永远不会是真的
  return 2;
else
  return 1;
```

简化为这样：

```
data[7] = 9;
return 1;
```

简单内联

从 Solidity 0.8.2 版本开始，有另一个优化步骤，它用这些指令的拷贝来替换某些包含以“跳转”结束的“简单”指令的块的跳转。这相当于对简单的、小的 Solidity 或 Yul 函数进行内联。特别是，PUSHTAG(tag) JUMP 序列可以被替换，只要 JUMP 被标记为”进入”一个函数的跳转，并且在 tag 后面有一个基本块（如上面描述的“通用子表达式消除器”），它以另一个 JUMP 结束，被标记为“离开”一个函数的跳转。

特别是，考虑以下为调用内部 Solidity 函数而生成的汇编的原型例子:

```
  tag_return
  tag_f
  jump      // 从此进入
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // 从此退出
```

只要函数的主体是一个连续的基本块，“内联”就可以用位于 tag_f 处的块来代替 tag_f jump，结果是：

```
  tag_return
  ...body of function f...
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
```

(续下页)

(接上页)

```
  ...body of function f...
  jump      // 从此退出
```

现在，理想情况下，上述的其他优化器步骤将导致返回标签的推送被移向剩余的跳转，从而导致：

```
  ...body of function f...
  tag_return
  jump
tag_return:
  ...opcodes after call to f...

tag_f:
  ...body of function f...
  jump      // 从此退出
```

在这种情况下，"窥视孔优化器（PeepholeOptimizer）" 将删除返回跳转。理想情况下，所有对 `tag_f` 的引用都可以这样做，而不使用它，特别处理的话，它也可以被移除：

```
...body of function f...
...opcodes after call to f...
```

因此，对函数 `f` 的调用是内联的，可以删除 `f` 的原始定义。

无论何时，只要启发式算法表明，在合同的生命周期内，内联比不内联更便宜，就会尝试这样的内联。这种启发式方法取决于函数体的大小、对其标记的其他引用的数量（近似于函数调用的数量）以及合约的预期执行次数（全局优化器参数"runs"）。

3.21.5 基于 Yul 的优化器模块

基于 Yul 的优化器由几个阶段和组件组成，它们都以语义等效的方式转换 AST。我们的目标是，最终的代码要么更短，要么至少略长，但允许进一步的优化步骤。

> **警告：** 由于优化器正在进行大量开发，这里的信息可能已经过时。如果您依赖某项功能，请直接联系团队。

优化器目前遵循的是一种纯粹的贪婪策略，不做任何回溯。

下面将解释基于 Yul 的优化器模块的所有组件。以下的转换步骤是主要的组成部分：

- SSA 转换
- 通用子表达式消除器
- 表达式简化器

- 冗余赋值消除器
- 完全内联

优化器的步骤

这是按字母顺序排列的基于 Yul 的优化器的所有步骤的列表。您可以在下面找到更多关于各个步骤和它们的顺序的信息。

缩略语	全称
f	块展平器
l	循环引用程序
c	通用子表达式消除器
C	条件简化器
U	有条件的非对称性放大器
n	控制流简化器
D	死代码消除器
E	等价的存储清除器
v	等价函数组合器
e	表达式内联
j	表达式连接器
s	表达式简化器
x	表达式拆分器
I	循环条件进入正文
O	体外循环条件
o	循环初始重写器
i	完全内联
g	函数分组器
h	函数提升器
F	函数特殊化器
T	字面意义上的再物质化器（*LiteralRematerialiser*）
L	负载解析器
M	循环不变代码模式
r	冗余赋值消除器
R	基于推理的简化器 - 高度实验性
m	再物质化
V	*SSA* 反转器
a	*SSA* 转换
t	结构简化器
p	未使用的函数参数管理器

续下页

表 1 – 接上页

缩略语	全称
S	未使用的存储清除器
u	未使用过的处理器
d	初始化程序

一些步骤依赖于 BlockFlattener，FunctionGrouper，ForLoopInitRewriter 所保证的属性。由于这个原因，Yul 优化器总是在应用用户提供的任何步骤之前应用它们。

基于推理的简化器（ReasoningBasedSimplifier）是一个优化器步骤，目前在默认步骤集中没有启用。它使用一个 SMT 求解器来简化算术表达式和布尔条件。此外，它还没有得到彻底的测试或验证，可能会产生不可复现的结果，所以请谨慎使用!

选择优化方案

默认情况下，优化器将其预定义的优化步骤序列应用于生成的程序集。您可以使用 --yul-optimizations 选项来覆盖这个序列并提供您自己的序列：

```
solc --optimize --ir-optimized --yul-optimizations 'dhfoD[xarrscLMcCTU]uljmul:fDnTOc'
```

步骤的顺序很重要，会影响到输出的质量。此外，应用一个步骤可能为其他已经应用的步骤发现新的优化机会。因此，重复步骤往往是有益的。

[...] 里面的序列将在一个循环中多次应用，直到 Yul 代码保持不变或达到最大轮数（目前是 12）。方括号（[]）可以在一个序列中多次使用，但不能嵌套。

需要注意的一件事是，有一些硬编码的步骤总是在用户提供的序列之前和之后运行，如果用户没有提供序列，则是默认序列。

清理序列分界符 : 是可选的，用于提供一个自定义的清理序列，以取代默认序列。如果省略，优化器将简单地应用默认的清理序列。此外，定界符可以放在用户提供的序列的开头，这将导致优化序列为空，反之，如果放在序列的末尾，将被视为一个空的清理序列。

预处理

预处理组件进行转换，使程序变成某种更容易操作的正常形式。这种正常形式在剩下的优化过程中被保留。

消歧器

消歧器获取 AST 并返回一个新拷贝，其中所有标识符在输入 AST 中都有唯一的名称。这是所有其他优化器阶段的先决条件。其中一个好处是，标识符查找不需要考虑作用域，这简化了其他步骤所需的分析。

所有后续阶段都有一个属性，即所有的名字都保持唯一。这意味着如果需要引入一个新的标识符，就会产生一个新的唯一名称。

函数提升器

函数提升器将所有的函数定义移到最上面的块的末尾。只要在消歧义阶段之后进行，这就是一个语义上的等价转换。原因是，将一个定义移到更高层次的块中不能降低其可见性，而且不可能引用在不同函数中定义的变量。

这个阶段的好处是，可以更容易地查找函数定义，并且可以孤立地优化函数，而不必完全遍历 AST。

函数分组器

函数分组器必须在消歧义器和函数提升器之后应用。它的作用是将所有不是函数定义的最上面的元素移到一个单一的块中，这个块是根块的第一个语句。

在这一步之后，一个程序具有以下正常形式：

```
{ I F... }
```

其中 `I` 是一个（可能是空的）区块，不包含任何函数定义（甚至是递归的），`F` 是一个函数定义的列表，使得没有一个函数包含函数定义。

这个阶段的好处是，我们总是知道功能列表的开始位置。

循环条件进入正文

这种转换将 for 循环的循环迭代条件移动到循环体中。我们需要这种转换，因为表达式拆分器 将不适用于迭代条件表达式（以下示例中的 `C`）。

```
for { Init... } C { Post... } {
    Body...
}
```

被转化为

```
for { Init... } 1 { Post... } {
    if iszero(C) { break }
```

(续下页)

(接上页)

```
    Body...
}
```

当与 循环不变代码模式搭配时，这种转换也是有用的，因为循环不变条件中的不变量可以在循环之外进行。

循环初始重写器

这种转换将 for-loop 的初始化部分移到循环之前：

```
for { Init... } C { Post... } {
    Body...
}
```

被转化为

```
Init...
for {} C { Post... } {
    Body...
}
```

这简化了其余的优化过程，因为我们可以忽略 for 循环初始化块的复杂范围规则。

初始化程序

这一步重写了变量声明，使所有的变量都被初始化。像 `let x, y` 这样的声明被分割成多个声明语句。

目前只支持用零值初始化。

伪 SSA 转换

这个组件的目的是让程序变成一个较长的形式，以便其他组件能够更容易地与之配合。最终的表现形式将类似于静态单一赋值（SSA）的形式，不同的是，它不使用明确的"phi" 函数来合并来自控制流不同分支的值，因为 Yul 语言中不存在这样的功能。相反，当控制流合并时，如果一个变量在其中一个分支中被重新赋值，就会声明一个新的 SSA 变量来保持它的当前值，这样，下面的表达式仍然只需要引用 SSA 变量。

下面是一个转换的例子：

```
{
    let a := calldataload(0)
    let b := calldataload(0x20)
    if gt(a, 0) {
        b := mul(b, 0x20)
    }
```

(续下页)

(接上页)

```
    a := add(a, 1)
    sstore(a, add(b, 0x20))
}
```

应用以下所有转换步骤后，程序将如下所示：

```
{
    let _1 := 0
    let a_9 := calldataload(_1)
    let a := a_9
    let _2 := 0x20
    let b_10 := calldataload(_2)
    let b := b_10
    let _3 := 0
    let _4 := gt(a_9, _3)
    if _4
    {
        let _5 := 0x20
        let b_11 := mul(b_10, _5)
        b := b_11
    }
    let b_12 := b
    let _6 := 1
    let a_13 := add(a_9, _6)
    let _7 := 0x20
    let _8 := add(b_12, _7)
    sstore(a_13, _8)
}
```

请注意，此代码段中唯一重新分配的变量是 b。无法避免这种重新分配，因为根据控制流，b 具有不同的值。所有其他变量在定义后都不会改变其值。该属性的优点是，变量可以自由移动，对它们的引用可以通过它们的初始值进行交换（反之亦然），只要这些值在新上下文中仍然有效。

当然，这里的代码远远没有得到优化。相反，它要长得多。我们希望这段代码更容易使用，此外，还有一些优化器步骤可以撤销这些更改，并在最后使代码更加紧凑。

表达式拆分器

表达式拆分器将诸如 add(mload(0x123), mul(mload(0x456), 0x20)) 这样的表达式变成一连串独特变量的声明，这些变量被分配给该表达式的子表达式，这样每个函数调用只有变量作为参数。

上述内容将被转化为

```
{
    let _1 := 0x20
    let _2 := 0x456
    let _3 := mload(_2)
    let _4 := mul(_3, _1)
    let _5 := 0x123
    let _6 := mload(_5)
    let z := add(_6, _4)
}
```

请注意，这种转换并不改变操作码或函数调用的顺序。

它不适用于循环迭代条件，因为循环控制流不允许在所有情况下"概述"内部表达式。我们可以通过应用循环条件进入正文 将迭代条件移动到循环体中，从而避开这个限制。

最后一个程序的形式应确保（循环条件除外）函数调用不会嵌套在表达式中，所有函数调用参数都必须是变量。

这种形式的好处是，更容易重新排列操作码序列，也更容易执行函数调用内联。此外，也更简单地替换表达式的各个部分或重新组织"表达式树"。缺点是这样的代码对我们来说更难阅读。

SSA 转换

这个阶段尽可能地用新变量的声明来取代对现有变量的重复赋值。重新赋值仍然存在，但是所有对重新赋值的变量的引用都被新声明的变量所取代。

示例：

```
{
    let a := 1
    mstore(a, 2)
    a := 3
}
```

被转化为

```
{
    let a_1 := 1
    let a := a_1
```

(续下页)

(接上页)

```
    mstore(a_1, 2)
    let a_3 := 3
    a := a_3
}
```

精确语义:

对于任何在代码中被分配到某处的变量 a (带值声明且从未重新分配的变量不被修改), 执行以下转换:

- 将 let a := v 替换为 let a_i := v let a := a_i
- 将 a := v 替换为 let a_i := v a := a_i, 其中 i 是一个数字, 使得 a_i 尚未使用。

此外, 总是记录用于 a 的 i 的当前值, 并用 a_i 替换对 a 的每次引用。变量 a 的当前值映射在每个分配给它的块结束时被清除, 如果它被分配在 for 循环体或 post 块内, 则在 for 循环初始块结束时被清除。如果一个变量的值根据上面的规则被清除, 并且该变量被声明在块之外, 一个新的 SSA 变量将在控制流加入的位置被创建, 这包括循环后/体块的开始和 If/Switch/ForLoop/Block 语句之后的位置。

在此阶段之后, 建议使用冗余赋值消除器删除不必要的中间分配。

如果在这个阶段之前运行表达式拆分器和通用子表达式消除器, 那么这个阶段会提供最好的结果, 因为这样就不会产生过多的变量。另一方面, 如果在 SSA 转换之后运行通用子表达式消除器, 则效率更高。

冗余赋值消除器

SSA 转换总是生成 a := a_i 形式的赋值, 尽管这些赋值在许多情况下可能是不必要的, 比如下面的例子:

```
{
    let a := 1
    a := mload(a)
    a := sload(a)
    sstore(a, 1)
}
```

SSA 转换将这个片段转换为以下内容:

```
{
    let a_1 := 1
    let a := a_1
    let a_2 := mload(a_1)
    a := a_2
    let a_3 := sload(a_2)
    a := a_3
    sstore(a_3, 1)
}
```

冗余赋值消除器将删除对 `a` 的所有三个赋值，因为未使用 `a` 的值，因此将此代码段转换为严格的 SSA 形式为：

```
{
    let a_1 := 1
    let a_2 := mload(a_1)
    let a_3 := sload(a_2)
    sstore(a_3, 1)
}
```

当然，确定分配是否多余的错综复杂的部分与加入控制流有关。

该组件的详细工作情况如下：

AST 被遍历了两次：分别在在信息收集步骤和实际删除步骤中。在信息收集过程中，我们维护了一个从赋值语句到“未使用（unused）”，“未决定（undecided）”和“已使用（used）”三种状态的映射，这标志着分配的值是否会在以后被变量的引用使用。

当一个赋值被访问时，它被添加到处于“未决定”状态的映射中（见下面关于 for 循环的注释），而其他每个仍处于“未决定”状态的对同一变量的赋值被改为“未使用”。当一个变量被引用时，任何对该变量的赋值仍处于“未决定”状态，其状态被改变为“已使用”。

在控制流分叉的地方，映射的拷贝被移交给每个分支。在控制流汇合的地方，来自两个分支的两个映射以下列方式合并：只在一个映射中的语句或具有相同状态的语句不作改动地使用。冲突的值以如下方式解决：

- “未使用”，“未决定”-> “未决定”
- “未使用”，“已使用”-> “已使用”
- “未决定”，“已使用”-> “已使用”

对于 For 循环，考虑到条件下的连接控制流，将对条件、主体和后部进行两次访问。换句话说，我们创建了三条控制流路径：循环的零次运行、一次运行和两次运行，然后在最后合并它们。

不需要模拟第三次甚至更多的运行，这可以如下所示：

迭代开始时的赋值状态将决定性地导致该赋值在迭代结束时的状态。假如这个状态映射函数被称为 `f`。如上所述，三种不同状态 `unused`（未使用），`undecided`（未决定）和 `used`（已使用）的组合是 `最多（max）` 操作，其中 `unused = 0`，`undecided = 1`，`used = 2`。

正确的方法是计算

```
max(s, f(s), f(f(s)), f(f(f(s))), ...)
```

作为循环后的状态。因为 `f` 只是有三个不同的值的范围，迭代它必须在最多三个迭代后达到一个循环，因此 `f(f(f(s)))` 必须等于 `s`, `f(s)` 或 `f(f(s))` 其中之一，因此

```
max(s, f(s), f(f(s))) = max(s, f(s), f(f(s)), f(f(f(s))), ...).
```

总之，最多运行两次循环就足够了，因为只有三种不同的状态。

对于有" 默认" 情况的 switch 语句，没有跳过 switch 的控制流部分。

当一个变量超出范围时，所有仍处于" 未决定" 状态的语句都被改为" 未使用"，除非该变量是一个函数的返回参数--如何是这样，状态变为" 已使用"。

在第二次遍历中，所有处于" 未使用" 状态的赋值都被删除。

这一步通常是在 SSA 转换之后立即运行，以完成伪 SSA 的生成。

工具

可移动性

可移动性是表达式的一个属性。它大致上意味着表达式是没有副作用的，它的评估只取决于变量的值和环境的调用常数状态。大多数表达式都是可移动的。以下部分使表达式不可移动:

- 函数调用（如果函数中的所有语句都是可移动的，未来可能会放宽）
- 有副作用的操作码（如 `call` 或 `selfdestruct`）
- 读取或写入内存, 存储或外部状态信息的操作码
- 取决于当前 PC、内存大小或返回数据大小的操作码

数据流分析器

数据流分析器本身不是一个优化步骤，而是被其他组件作为工具使用。在遍历 AST 时，它跟踪每个变量的当前值，只要该值是一个可移动的表达式。它记录了作为表达式一部分的变量，这些表达式目前被分配给其他每个变量。在每次对变量 `a` 的赋值时，`a` 的当前存储值被更新，只要 `a` 是 `b` 当前存储表达式的一部分，变量 `b` 的所有存储值都被清除。

在控制流连接处，如果变量在任何控制流路径中已经或将要被分配，那么关于这些变量的记忆就会被清除。例如，在进入 for 循环时，所有将在主体或后块中分配的变量都被清除。

表达式的简化

这些简化过程会改变表达式，并用等效的、希望更简单的表达式替换它们。

通用子表达式消除器

这一步使用数据流分析器，用对某一变量的引用来替换语法上与该变量当前值相匹配的子表达式。这是一个等价转换，因为这种子表达式必须是可移动的。

如果值是一个标识符，所有本身是标识符的子表达式都被其当前值替换。

上述两条规则的结合允许计算出一个局部值的编号，这意味着如果两个变量有相同的值，其中一个将永远是未使用的。然后，未使用过的处理器或冗余赋值消除器将能够完全消除此类变量。

如果之前运行过表达式拆分器，则此步骤尤其有效。如果代码是伪 SSA 形式，那么变量值的可用时间更长，因此我们有更高的机会替换表达式。

如果通用子表达式消除器在它之前运行，表达式简化器将能够进行更好的替换。

表达式简化器

表达式简化器使用数据流分析器，并利用表达式的等价变换列表，如 `X + 0 -> X` 来简化代码。

它试图在每个子表达式上匹配诸如 `X + 0` 的模式。在匹配过程中，它将变量解析为当前分配的表达式，以便能够匹配更深入的嵌套模式，即使代码是伪 SSA 形式。

一些模式如 `X - X -> 0` 只能在表达式 `X` 是可移动的情况下应用，否则会删除其潜在的副作用。由于变量引用总是可移动的，即使它们的当前值可能不是，表达式简化器在拆分或伪 SSA 形式下又更加强大。

字面意义上的再物质化器（LiteralRematerialiser）

有待记录。

负载解析器

优化阶段，分别将 `sload(x)` 和 `mload(x)` 类型的表达式替换为当前存储和内存中的值，如果已知的话。

如果代码是 SSA 形式的，效果最好。

先决条件：消歧器，循环初始重写器。

基于推理的简化器

这个优化器使用 SMT 求解器来检查 `if` 条件是否为常数。

- 如果 限制条件和条件是不满足的（UNSAT），那么条件永远不会是真的，整个主体可以被删除。
- 如果 限制条件和非限制条件是不满足的（UNSAT），那么条件永远是真的，可以用 `1` 代替。

只有在条件是可移动的情况下，上面的简化才能适用。

它只对 EVM 语言有效，但在其他语言上使用是安全的。

先决条件：消歧器，SSA 转换。

声明规模的简化

循环引用程序

这个阶段删除了那些互相调用但既没有外部引用也没有从最外层上下文中引用的函数。

条件简化器

如果可以从控制流中确定数值，条件简化器就会插入对条件变量的赋值。

销毁 SSA 表格。

目前，这个工具是非常有限的，主要是因为我们还没有支持布尔类型。由于条件只检查表达式是否为非零，我们不能指定一个特定的值。

当前的特性：

- 切换条件：插入 “< 条件 > := < 条件标签 >”
- 在带有终止控制流的 if 语句后，插入 “< 条件 > : =0”

未来的特性：

- 允许用”1” 替换
- 考虑到用户定义的终止函数

如果之前已经运行过死代码的删除，那么使用 SSA 表单效果最好。

先决条件：消歧器。

有条件的非对称性放大器

条件简化器的反面。

控制流简化器

简化了几个控制流结构：

- 用 pop（条件）代替 if，用空的程序体代替 if
- 移除空的默认 switch 情况
- 如果不存在默认情况，则删除空的 switch 情况
- 用 pop（表达式）代替没有条件的 switch
- 把单例的 switch 变成 if
- 用 pop（表达式）和程序体代替 switch，只用默认情况

- 用匹配的条件程序体的常量表达式替换 switch
- 将 `for` 替换为终止控制流，在没有其他 break/continue 的情况下替换为 `if`
- 移除函数末尾的 `leave`

这些操作都不依赖于数据流。然而结构简化器执行类似的任务，确实依赖于数据流。

控制流简化器在其遍历过程中确实记录了是否存在 `break` 和 `continue` 语句。

先决条件：消歧器，函数提升器，循环初始重写器。重要提示：引入了 EVM 操作代码，因此目前只能用于 EVM 代码。

死代码消除器

这个优化阶段删除了不可到达的代码。

无法访问的代码是指在一个区块内的任何代码，其前面有 leave，return，invalid，break，continue，selfdestruct，revert 或调用用户定义的函数，并无限地递归。

函数定义被保留下来，因为它们可能被早期的代码调用，因此被认为是可访问的。

因为在 for 循环的 init 块中声明的变量，其范围会扩展到循环体，所以我们要求循环初始重写器在此步骤之前运行。

先决条件：循环初始重写器, 函数提升器, 函数分组器

等价的存储清除器

如果之前有对 `mstore(k, v)` / `sstore(k, v)` 的调用，但中间没有其他存储，并且 `k` 和 `v` 的值没有变化，则该步骤将删除 `mstore(k, v)` 和 `sstore(k, v)` 的调用。

如果在 SSA 转换和通用子表达式消除器之后运行，这个简单的步骤是有效的，因为 SSA 将确保变量不会改变，而通用子表达式消除器在已知值相同的情况下会重新使用完全相同的变量。

先决条件：消歧器, 循环初始重写器

未使用过的处理器

这一步删除了所有从未被引用的函数的定义。

它还删除了从未被引用的变量的声明。如果声明指定了一个不可移动的值，表达式将被保留，但其值将被丢弃。

所有可移动的表达式语句（未被赋值的表达式）都被删除。

结构简化器

这是一个一般的步骤，在结构层面上进行各种简化：

- 用 `pop(条件)` 代替 if 语句的空程序体。
- 用其主体替换带有真实条件的 if 语句
- 删除带有错误条件的 if 语句
- 把单例的 switch 变成 if
- 用 `pop(表达式)` 和程序体代替 switch，只用默认情况
- 通过匹配的条件程序体，用字面表达式替换 switch
- 用其初始化部分取代带有错误条件的 for 循环

该组件使用数据流分析器。

块展平器

这个阶段通过在外部块的适当位置插入内部块的语句来消除嵌套块。它依赖于函数分组器，并不对最外层的块进行展平，以保持函数分组器产生的形式。

```
{
    {
        let x := 2
        {
            let y := 3
            mstore(x, y)
        }
    }
}
```

被转化为

```
{
    {
        let x := 2
        let y := 3
        mstore(x, y)
    }
}
```

只要代码没有歧义，这就不会造成问题，因为变量的作用域只能增长。

循环不变代码模式

这种优化将可移动的 SSA 变量声明移到循环之外。

只有在循环体或后块中的最高级别的语句被考虑，即条件分支内的变量声明不会被移出循环。

要求：

- 消歧器, 循环初始重写器和函数提升器必须提前运行。
- 表达式拆分器和 SSA 转换应在前期运行以获得更好的结果。

函数级的优化

函数特殊化器

这一步是用字面参数来实现函数的专业化。

如果一个函数，例如，`function f(a, b) { sstore (a, b) }`，被调用时有字面参数，例如，`f(x, 5)`，其中 `x` 是一个标识符，可以通过创建一个新函数 `f_1` 来专门化，该函数只需要一个参数，即：

```
function f_1(a_1) {
    let b_1 := 5
    sstore(a_1, b_1)
}
```

其他优化步骤将能够对函数进行更多的简化。优化步骤主要对那些不会被内联的函数有用。

先决条件：消歧器，函数提升器

建议将字面意义上的再物质化器（LiteralRematerialiser）作为先决条件，尽管它不是正确性的必要条件。

未使用的函数参数管理器

这一步是删除一个函数中未使用的参数。

如果一个参数没有使用，比如在 `function f(a,b,c) -> x, y { x := div(a,b) }` 中的 `c` 和 `y`，我们删除该参数并创建一个新的" 连接" 函数，如下所示：

```
function f(a,b) -> x { x := div(a,b) }
function f2(a,b,c) -> x, y { x := f(a,b) }
```

并将所有对 `f` 的引用替换为 `f2`。之后应该运行内联，以确保所有对 `f2` 的引用都被 `f` 替换。

先决条件：消歧器，函数提升器，字面意义上的再物质化器

字面意义上的再物质化器这个步骤对于正确性来说不是必需的。它有助于处理诸如以下情况：`function f(x) -> y { revert(y, y} }` 其中字面意思 `y` 将被其值 `0` 取代，使我们能够重写该函数。

未使用的存储清除器

优化器组件，删除多余的 sstore 和内存存储语句。对于一个 sstore，如果所有传出的代码路径都恢复了（由于显式的的 revert()，invalid()，或无限递归）或导致另一个 sstore，优化器可以知道它将覆写第一个存储，该语句将被删除。然而，如果在初始 sstore 和恢复之间有读操作，或者覆写的 sstore，该语句将不会被删除。这样的读操作包括：外部调用，有任何存储访问的用户定义的函数，以及不能证明与初始 sstore 写的槽不同的 sload。

例如，下面的代码

```
{
    let c := calldataload(0)
    sstore(c, 1)
    if c {
        sstore(c, 2)
    }
    sstore(c, 3)
}
```

在运行未使用的存储消除器步骤后，将被转化为以下代码

```
{
    let c := calldataload(0)
    if c { }
    sstore(c, 3)
}
```

对于内存存储操作，事情一般比较简单，至少在最外层的 yul 块中是这样，因为如果在任何代码路径中从未被读取，所有这样的语句都将被删除。然而，在函数分析层面，其方法与 sstore 类似，因为我们不知道一旦离开函数的范围，内存位置是否会被读取，所以只有当所有的代码路径都导致内存被覆写时，语句才会被删除。

最好以 SSA 形式运行。

先决条件：Disambiguator, ForLoopInitRewriter.

等价函数组合器

如果两个函数在语法上是等价的，同时允许变量重命名，但不允许任何重新排序，那么对其中一个函数的任何引用都会被另一个函数取代。

实际删除的功能是由未使用过的处理器执行的。

函数内联

表达式内联

优化器的这个组件通过内联可以在函数表达式中内联的函数来执行限制性的函数内联，函数为：

- 返回一个单一的值。
- 有一个像 `r := < 函数表达式>` 的主体。
- 既没有提到自己，也没有提到右边的 `r`。

此外，对于所有的参数，以下各项都需要为真：

- 参数是可移动的。
- 该参数在函数体中被引用不到两次，或者该参数相当便宜（" 成本" 最多为 1，就像一个 0xff 以下的常数）。

例如：要被内联的函数的形式是：`function f(...) -> r { r := E }` 其中 `E` 是一个不引用 `r` 的表达式，函数调用中的所有参数都是可移动表达式。

这种内联的结果总是一个单一的表达式。

该组件只能用于具有唯一名称的源码。

完全内联

完全内联用函数的主体取代了某些函数的调用。这在大多数情况下是没有什么帮助的，因为它只是增加了代码的大小，但并没有什么好处。此外，代码通常是非常昂贵的，我们往往宁愿要更短的代码而不是更有效的代码。不过，在相同的情况下，内联一个函数可以对后续的优化步骤产生积极的影响。例如，如果一个函数参数是一个常数，就会出现这种情况。

在内联过程中，一个启发式方法被用来判断函数调用是否应该被内联。目前的启发式方法是不内联到" 大" 函数，除非被调用的函数很小。只使用一次的函数以及中等大小的函数被内联，而带有常数参数的函数调用允许稍大的函数。

在未来，我们可能会加入一个回溯组件，它不会立即对一个函数进行内联，而只是对其进行专业化处理，这意味着会生成一个函数的拷贝，其中某个参数总是被一个常数取代。之后，我们可以在这个专用函数上运行优化器。如果结果有很大的收益，那么这个专门化的函数就被保留下来，否则就用原来的函数代替。

清理

清理工作是在优化器运行结束时进行的。它试图将分割的表达式再次组合成深度嵌套的表达式，并且通过尽可能地消除变量来提高堆栈机的" 可编译性"。

表达式连接器

这是与表达式分割器相反的操作。它把正好有一个引用的变量声明序列变成一个复杂的表达式。这个阶段完全保留了函数调用和操作码执行的顺序。它不使用任何关于操作码的互换性的信息；如果将一个变量的值移到它的使用位置会改变任何函数调用或操作码执行的顺序，则不执行转换。

注意，组件不会移动变量赋值或被多次引用的变量的赋值。

片段 `let x := add(0, 2) let y := mul(x, mload(2))` 不能转换，因为它将导致调用操作码 `add` 和 `mload` 的顺序被调换--尽管这不会有什么影响，因为 `add` 是可移动的。

当像这样重排操作码时，变量引用和字面意义被忽略了。因此，片段 `let x := add(0, 2) let y := mul(x, 3)` 被转换为 `let y := mul(add(0, 2), 3)`，尽管 `add` 操作码将在计算字面意义 `3` 后执行。

SSA 反转器

这是一个微小的步骤，如果它与通用子表达式消除器和未使用过的处理器相结合，则有助于扭转 SSA 转换的影响。

我们生成的 SSA 形式对 EVM 和 WebAssembly 的代码生成是不利的，因为它生成了许多局部变量。最好的办法是用赋值重新使用现有的变量，而不是用新的变量声明。

SSA 转换改写

```
let a := calldataload(0)
mstore(a, 1)
```

为

```
let a_1 := calldataload(0)
let a := a_1
mstore(a_1, 1)
let a_2 := calldataload(0x20)
a := a_2
```

问题是在引用 `a` 时使用了变量 `a_1`，而不是 `a`。SSA 转换改变了这种形式的语句，只需将声明和赋值互换。上面的片段被转化为

```
let a := calldataload(0)
let a_1 := a
```

(续下页)

(接上页)

```
mstore(a_1, 1)
a := calldataload(0x20)
let a_2 := a
```

这是一个非常简单的等价转换，但是当我们现在运行通用子表达式消除器时，它将用 a 替换所有出现的 a_1（直到 a 被重新赋值）。然后，未使用过的处理器将完全消除变量 a_1，从而完全逆转 SSA 的转换。

堆栈压缩器

让以太坊虚拟机的代码生成变得困难的一个问题是，在表达式堆栈中，有 16 个插槽的硬性限制，可以向下延伸。这或多或少转化为 16 个局部变量的限制。堆栈压缩器采用 Yul 代码并将其编译为 EVM 字节码。每当堆栈差异过大时，它就会记录发生在哪个函数中。

对于每一个造成这种问题的函数，再物质化都会被调用，并提出特殊要求，以积极消除按其值的成本排序的特定变量。

一旦失败，这个程序会重复多次。

再物质化

再物质化阶段试图用最后分配给变量的表达式来替换变量引用。当然，这只有在这个表达式的评估费用相对较低的情况下才是有益的。此外，只有当表达式的值在赋值点和使用点之间没有变化时，它才具有语义上的等同性。这个阶段的主要好处是，如果它导致一个变量被完全消除，它可以节省堆栈槽（见下文），但是如果表达式非常便宜，它也可以在 EVM 上节省一个 DUP 操作码。

再物质化使用数据流分析器来跟踪变量的当前值，这些变量总是可移动的。如果数值非常便宜或者变量被明确要求消除，那么变量的引用就会被其当前值所取代。

体外循环条件

逆转体外循环条件的转换。

对于任何可移动的 c，它转换

```
for { ... } 1 { ... } {
if iszero(c) { break }
...
}
```

为

```
for { ... } c { ... } {
...
}
```

而它又转换

```
for { ... } 1 { ... } {
if c { break }
...
}
```

为

```
for { ... } iszero(c) { ... } {
...
}
```

字面意义上的再物质化器应在此步骤之前运行。

特定的 WebAssembly

主要功能

将最上面的块改变为一个具有特定名称（"main"）的函数，它没有输入和输出。

取决于函数分组器。

3.22 合约的元数据

Solidity 编译器自动生成一个 JSON 文件。该文件包含两种有关已编译合约的信息：

- 如何与合约交互：ABI 和 NatSpec 文档。
- 如何重现编译并验证已部署的合约：编译器版本、编译器设置和使用的源文件。

默认情况下，编译器会将元数据文件的 IPFS 哈希值附加到每个合约的运行时字节码（不一定是创建字节码）末尾，这样，如果发布了该文件，就可以通过验证的方式检索该文件，而无需求助于集中式数据提供者。其他可用选项包括 Swarm 哈希值和不在字节码中附加元数据哈希值。这些选项可通过*标准 JSON 接口* 进行配置。

您必须将元数据文件发布到 IPFS，Swarm 或其他服务，以便其他人可以访问它。您可以通过使用 `solc --metadata` 命令和 `--output-dir` 参数来创建该文件。如果没有这个参数，元数据将被写到标准输出。元数据包含 IPFS 和 Swarm 对源代码的引用，所以除了元数据文件外，您还必须上传所有的源文件。对于 IPFS，`ipfs add` 返回的 CID 中包含的哈希值（不是文件的直接 sha2-256 哈希值）应与字节码中包含的哈希值相匹配。

元数据文件的格式如下。下面的示例是以人类可读的方式呈现的。正确格式化的元数据应正确使用引号，尽量减少空白，并按字母顺序对所有对象的键值进行排序，以形成规范格式。不允许使用注释，此处注释仅用于解释目的。

```
{
  // 必选：编译器的详情，内容视语言而定。
  "compiler": {
    // 可选：生成此输出的编译器二进制文件的哈希值
    "keccak256": "0x123...",
    // 对 Solidity 来说是必选的：编译器的版本
    "version": "0.8.2+commit.661d1103"
  },
  // 必选：源代码的编程语言，一般会选择规范的“子版本”
  "language": "Solidity",
  // 必选：合约的生成信息
  "output": {
    // 必选：合约的 ABI 定义，见 “合约 ABI 规范”
    "abi": [/* ... */],
    // 必选：合约的开发者 NatSpec 文档，详见 https://docs.soliditylang.org/en/latest/
↪natspec-format.html
    "devdoc": {
      // 合约 @author NatSpec字段的内容
      "author": "John Doe",
      // 合约中 @dev NatSpec 字段的内容
      "details": "Interface of the ERC20 standard as defined in the EIP. See https://
↪eips.ethereum.org/EIPS/eip-20 for details",
      "errors": {
        "MintToZeroAddress()" : {
          "details": "Cannot mint to zero address"
        }
      },
      "events": {
        "Transfer(address,address,uint256)": {
          "details": "Emitted when `value` tokens are moved from one account (`from`)␣
↪toanother (`to`).",
          "params": {
            "from": "The sender address",
            "to": "The receiver address",
            "value": "The token amount"
          }
        }
      },
      "kind": "dev",
      "methods": {
        "transfer(address,uint256)": {
```

(续下页)

(接上页)

```
          // 方法的 @dev NatSpec 字段的内容
          "details": "Returns a boolean value indicating whether the operation␣
↪succeeded. Must be called by the token holder address",
          // 方法的 @param NatSpec 字段的内容
          "params": {
            "_value": "The amount tokens to be transferred",
            "_to": "The receiver address"
          },
          // @return NatSpec 字段的内容。
          "returns": {
            // 如果存在，返回var名称 (这里是␣
↪ "success" )。如果返回的var是未命名的， "_0" 作为键。
            "success": "a boolean value indicating whether the operation succeeded"
          }
        }
      },
      "stateVariables": {
        "owner": {
          // 状态变量的 @dev NatSpec 字段的内容
          "details": "Must be set during contract creation. Can then only be changed␣
↪by the owner"
        }
      },
      // 合约中 @title NatSpec 字段的内容
      "title": "MyERC20: an example ERC20",
      "version": 1 // NatSpec 版本
    },
    // 必选：合约的用户 NatSpec 文档。请参阅 "NatSpec 格式"
    "userdoc": {
      "errors": {
        "ApprovalCallerNotOwnerNorApproved()": [
          {
            "notice": "The caller must own the token or be an approved operator."
          }
        ]
      },
      "events": {
        "Transfer(address,address,uint256)": {
          "notice": "`_value` tokens have been moved from `from` to `to`"
        }
      },
      "kind": "user",
      "methods": {
```

(续下页)

(接上页)

```
        "transfer(address,uint256)": {
          "notice": "Transfers `_value` tokens to address `_to`"
        }
      },
      "version": 1 // NatSpec 版本
    }
  },
  // 必选： 编译器设置。反映编译时 JSON 输入的设置。
  // 查看标准 JSON 输入的 "setting" 字段文档
  "settings": {
    // 对 Solidity 来说是必选的： 文件路径以及为其创建的合约或库的名称。
    "compilationTarget": {
      "myDirectory/myFile.sol": "MyContract"
    },
    // 对 Solidity 来说是必选的。
    "evmVersion": "london",
    // 对 Solidity 来说是必选的： 使用的库合约地址。
    "libraries": {
      "MyLib": "0x123123..."
    },
    "metadata": {
      // 反映输入 json 中使用的设置，默认为 "true"
      "appendCBOR": true,
      // 反映输入 json 中使用的设置，默认为 "ipfs"
      "bytecodeHash": "ipfs",
      // 反映输入 json 中使用的设置，默认为 "false"
      "useLiteralContent": true
    },
    // 可选：优化设置。 "enabled" 和 "runs" 字段已弃用，仅用于向后兼容。
    "optimizer": {
      "details": {
        "constantOptimizer": false,
        "cse": false,
        "deduplicate": false,
        // inliner 默认为 "true"
        "inliner": true,
        // jumpdestRemover 默认为 "true"
        "jumpdestRemover": true,
        "orderLiterals": false,
        // peephole 默认为 "true"
        "peephole": true,
        "yul": true,
        // 可选： 仅当 "yul" 为 "true" 时才出现
```

(续下页)

(接上页)

```
        "yulDetails": {
          "optimizerSteps": "dhfoDgvulfnTUtnIf...",
          "stackAllocation": false
        }
      },
      "enabled": true,
      "runs": 500
    },
    // 对 Solidity 来说是必选的: 导入重新映射的排序列表。
    "remappings": [ ":g=/dir" ]
  },
  // 必选: 编译源文件/源单元, 键为文件路径
  "sources": {
    "destructible": {
      // 必选(除非使用了 "url"): 源文件的字面内容
      "content": "contract destructible is owned { function destroy() { if (msg.
↪sender == owner) selfdestruct(owner); } }",
      // 必选: 源文件的 keccak256 哈希值
      "keccak256": "0x234..."
    },
    "myDirectory/myFile.sol": {
      // 必选: 源文件的 keccak256 哈希值
      "keccak256": "0x123...",
      // 可选: 源文件中给出的 SPDX 许可证标识符
      "license": "MIT",
      // 必选(除非使用了 "content", 见上文): 指向源文件的排序 URL,
      // 协议可任意选择, 但建议使用 IPFS URL
      "urls": [ "bzz-raw://7d7a...", "dweb:/ipfs/QmN..." ]
    }
  },
  // 必选: 元数据格式的版本
  "version": 1
}
```

警告: 由于产生的合约的字节码默认包含元数据哈希值，对元数据的任何改变都可能导致字节码的改变。这包括对文件名或路径的改变，而且由于元数据包括所有使用的源的哈希值，一个空白的改变就会导致不同的元数据和不同的字节码。

备注: 上面的 ABI 定义没有固定的顺序。它可以随着编译器的版本而改变。不过，从 Solidity 0.5.12 版本开始，该数组保持一定的顺序。

3.22.1 在字节码中对元数据哈希值进行编码

编译器目前默认将规范元数据文件的 IPFS 哈希（in CID v0）和编译器版本附加到字节码末尾。也可选择使用 Swarm 哈希值代替 IPFS，或使用实验标志。以下是所有可能的字段：

```
{
  "ipfs": "<metadata hash>",
  // 如果编译器设置中的 "bytecodeHash" 为 "bzzr1"，此处不是 "ipfs" 而是 "bzzr1"
  "bzzr1": "<metadata hash>",
  // 以前的版本使用 "bzzr0" 而不是 "bzzr1"
  "bzzr0": "<metadata hash>",
  // 如果使用任何影响代码生成的实验性功能
  "experimental": true,
  "solc": "<compiler version>"
}
```

因为我们将来可能会支持以其他方式检索元数据文件，因此这些信息被存储为 CBOR - 编码。字节码中的最后两个字节表示 CBOR 编码信息的长度。通过查看这个长度，可以用 CBOR 解码器对字节码的相关部分进行解码。

请访问 Metadata Playground 查看实际操作。

SOLC 的发布版本使用如上所示的 3 个字节的版本编码（主要、次要和补丁版本号各一个字节），而预发布版本将使用一个完整的版本字符串，包括提交哈希和构建日期。

命令行标志 `--no-cbor-metadata` 可以用来跳过元数据在部署的字节码末端的附加。同样地，标准 JSON 输入中的布尔字段 `settings.metadata.appendCBOR` 可以设置为 false。

备注： CBOR 映射也可能包含其他键，因此最好通过查看字节码末尾的 CBOR 长度来完全解码数据，并使用适当的 CBOR 分析器。不要依赖以 `0xa264` 或 `0xa2 0x64 'i' 'p' 'f' 's'` 开头的数据。

3.22.2 自动化接口生成和 NatSpec 的使用方法

元数据的使用方式如下：一个想要与合约交互的组件（例如钱包）会检索合约的代码。它对包含元数据文件的 IPFS/Swarm 哈希的 CBOR 编码部分进行解码。通过该哈希值，元数据文件被检索出来。该文件被 JSON 解码成一个类似于上述的结构。

然后，该组件可以使用 ABI 为合约自动生成一个基本的用户界面。

此外，钱包还可以使用 NatSpec 用户文档，在用户与合约进行交互时，向用户显示一条可读的确认信息，同时请求交易签名进行授权。

有关其他信息，请阅读*以太坊自然语言规范（NatSpec）格式*。

3.22.3 源代码验证的用法

如果已固定/发布，则可以从 IPFS/Swarm 获取合约的元数据。元数据文件还包含源文件的 URL 或 IPFS 哈希值，以及编译设置，即重现编译所需的一切信息。

有了这些信息，就可以通过重现编译来验证合约的源代码，并将编译的字节码与已部署合约的字节码进行比较。

由于元数据和源代码的哈希值都是字节码的一部分，因此可以自动验证元数据和源代码。文件或设置的任何更改都会导致不同的元数据哈希值。这里的元数据是整个编译过程的指纹。

Sourcify 利用这一特性进行“完全/完美验证”，并将文件公开固定在 IPFS 上，以便使用元数据哈希值进行访问。

3.23 合约 ABI 规范

3.23.1 基本设计

合约应用二进制接口（ABI）是在以太坊生态系统中与合约交互的标准方式，包括从区块链外部和合约间的交互。数据根据其类型进行编码，如本规范中所述。编码不是自描述的，因此需要一种特定的概要（schema）来进行解码。

我们假设合约的接口函数是强类型的，在编译时就知道，并且是静态的。我们假设所有合约在编译时都有它们所调用的任何合约的接口定义。

本规范不涉及其接口是动态的或其他只有在运行时才知道的合约。

3.23.2 函数选择器

一个函数调用数据的前四个字节指定了要调用的函数。它是函数签名的 Keccak-256 哈希值的前 4 字节（高位在左的大端序）。签名被定义为基本原型的典型表达，没有数据位置的指定，也就是带有括号的参数类型列表的函数名。参数类型由一个逗号分割 - 不使用空格。

备注： 一个函数的返回类型不是这个签名的一部分。在*Solidity 的函数重载* 中，返回类型不被考虑。原因是为了保持函数调用解析与上下文无关。然而*JSON 描述的 ABI* 却同时包含了输入和输出。

3.23.3 参数编码

从第 5 字节开始是被编码的参数。这种编码也被用在其他地方，比如，返回值和事件的参数也会被用同样的方式进行编码，而用来指定函数的 4 个字节则不需要再进行编码。

3.23.4 类型

以下是基础类型：

- `uint<M>`：M 位的无符号整数，`0 < M <= 256`，`M % 8 == 0`。例如：`uint32`，`uint8`，`uint256`。
- `int<M>`：以 2 的补码作为符号的 M 位整数，`0 < M <= 256`，`M % 8 == 0`。
- `address`：除了字面上的意思和语言类型的区别以外，等价于 `uint160`，在计算和函数选择器中，通常使用 `address`。
- `uint`，`int`：`uint256`，`int256` 各自的同义词. 在计算和函数选择器中，通常使用 `uint256` 和 `int256`。
- `bool`：等价于 `uint8`，取值限定为 0 或 1。在计算和函数选择器中，通常使用 `bool`。
- `fixed<M>x<N>`：M 位的有符号的固定小数位的十进制数字，`8 <= M <= 256`，`M % 8 == 0`，且 `0 < N <= 80`，其中值 `v` 是 `v / (10 ** N)`。
- `ufixed<M>x<N>`：无符号的 `fixed<M>x<N>`.
- `fixed`，`ufixed`：`fixed128x18`，`ufixed128x18` 各自的同义词。在计算和函数选择器中，通常使用 `fixed128x18` 和 `ufixed128x18`。
- `bytes<M>`：M 字节的二进制类型，`0 < M <= 32`。
- `function`：一个地址（20 字节）之后紧跟一个函数选择器（4 字节）。编码之后等价于 `bytes24`。

以下是定长数组类型：

- `<type>[M]`：有 M 个元素的定长数组，`M >= 0`，数组元素为给定类型。

备注： 虽然这个 ABI 规范可以表达零元素的固定长度数组，但编译器不支持它们。

以下是非定长类型：

- `bytes`：动态大小的字节序列。
- `string`：动态大小的 unicode 字符串，通常呈现为 UTF-8 编码。
- `<type>[]`：元素为给定类型的变长数组。

可以将若干类型放到一对括号中，用逗号分隔开，以此来构成一个元组（tuple）：

- `(T1,T2,...,Tn)`：由 `T1`，…，`Tn`，`n >= 0` 构成的元组

用元组构成元组，用元组构成数组等等也是可能的。另外也可以构成零元组（当 `n == 0` 时）。

将 Solidity 映射到 ABI 类型

Solidity 支持上面介绍的除了元祖之外的所有同名类型。另一方面，一些 Solidity 类型不被 ABI 支持。下表在左栏显示了不属于 ABI 的 Solidity 类型，在右栏显示了代表它们的 ABI 类型。

Solidity	ABI
address payable	`address`
合约	`address`
枚举	`uint8`
用户自定义类型	其基本值类型
结构体	元组 (`tuple`)

> **警告：** 在 `0.8.0` 版本之前，枚举可以有超过 256 个成员，并由最小的整数类型表示，其大小刚好可以容纳任何成员的值。

3.23.5 编码的设计标准

编码被设计为具有以下属性，如果一些参数是嵌套的数组，这些属性特别有用：

1. 访问一个值所需的读取次数最多是参数数组结构内的值的深度，即需要四次读取次数来检索 `a_i[k][l][r]`。在 ABI 的前一个版本中，在最坏的情况下，读取次数的数量与动态参数的总数成线性比例。
2. 变量或数组元素的数据不与其他数据交错，它是可重定位的，即它只使用相对的 "地址"。

3.23.6 编码的形式化规范

我们区分了静态和动态类型。静态类型是直接编码的，而动态类型是在当前块之后的一个单独分配的位置进行编码。

定义： 以下类型被称为 "动态"：

- `bytes`
- `string`
- 任意类型 `T` 的数组 `T[]`
- 任意动态类型 `T` 的定长数组 `T[k]`，其中 `k >= 0`
- 由动态的 `Ti` (`1 <= i <= k`) 构成的元组 `(T1,...,Tk)`

所有其他类型都被称为 "静态"。

定义： `len(a)` 是一个二进制字符串 `a` 的字节长度。`len(a)` 的类型被呈现为 `uint256`。

我们把实际的编码 `enc` 定义为一个由 ABI 类型到二进制字符串的值的映射，因而，当且仅当 `X` 的类型是动态的，`len(enc(X))` 才会依赖于 `X` 的值。

定义： 对任意 ABI 值 `X`，我们根据 `X` 的实际类型递归地定义 `enc(X)`。

- `(T1,...,Tk)` 对于 `k >= 0` 且任意类型 `T1`，…，`Tk`

  `enc(X) = head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(k))`

  这里，`X = (X(1), ..., X(k))` 并且 `head` 和 `tail` 被定义为如下 `Ti`：

  如果 `Ti` 是静态类型：

  `head(X(i)) = enc(X(i))` 和 `tail(X(i)) = ""`（空字符串）

  否则，即 `Ti` 是动态类型时，它们被定义为：

  `head(X(i)) = enc(len( head(X(1)) ... head(X(k)) tail(X(1)) ... tail(X(i-1)) ))` `tail(X(i)) = enc(X(i))`

  注意，在动态类型的情况下，由于 head 部分的长度仅取决于类型而非值，所以 `head(X(i))` 是定义明确的。它的值是从 `enc(X)` 的开始算起的，`tail(X(i))` 的起始位在 `head(X(i))` 中的偏移量。

- `T[k]` 对于任意 `T` 和 `k`：

  `enc(X) = enc((X[0], ..., X[k-1]))`

  即，它就像是个由相同类型的 `k` 个元素组成的元组那样被编码的。

- `T[]` 当 `X` 有 `k` 个元素（`k` 的类型为 `uint256`）：

  `enc(X) = enc(k) enc((X[0], ..., X[k-1]))`

  也就是说，它被编码为具有相同类型的 `k` 元素的元组（即静态大小为 `k` 的数组），前缀为元素的数量。

- 具有 `k` 字节长度的 `bytes`，（假设其类型为 `uint256`）：

  `enc(X) = enc(k) pad_right(X)`，即，字节数被编码为 `uint256`，紧跟着实际的 `X` 的字节码序列，再在前边（左边）补上可以使 `len(enc(X))` 成为 32 的倍数的最少数量的 0 值字节数据。

- `string`：

  `enc(X) = enc(enc_utf8(X))`，即 `X` 被 UTF-8 编码，且在后续编码中将这个值解释为 `bytes` 类型。注意，在随后的编码中使用的长度是其 UTF-8 编码的字符串的字节数，而不是其字符数。

- `uint<M>`：`enc(X)` 是在 `X` 的大端序编码的高位（左侧）补充若干 0 值字节以使其长度成为 32 字节。
- `address`：与 `uint160` 的情况相同。
- `int<M>`：`enc(X)` 是在 `X` 的大端序的 2 的补码编码的高位（左侧）添加若干字节数据以使其长度成为 32 字节；对于负数，添加值为 `0xff` 的字节数据，对于正数，添加 0 值字节数据。
- `bool`：与 `uint8` 的情况相同，`1` 用来表示 `true`，`0` 表示 `false`。
- `fixed<M>x<N>`：`enc(X)` 就是 `enc(X * 10**N)`，其中 `X * 10**N` 可以理解为 `int256`。
- `fixed`：与 `fixed128x18` 的情况相同。

- `ufixed<M>x<N>`：`enc(X)` 就是 `enc(X * 10**N)`，其中 `X * 10**N` 可以理解为 `uint256`。
- `ufixed`：与 `ufixed128x18` 的情况相同。
- `bytes<M>`：`enc(X)` 就是 `X` 的字节序列加上为使长度成为 32 字节而添加的若干 0 值字节。

注意，对于任意的 `X`，`len(enc(X))` 都是 32 的倍数。

3.23.7 函数选择器和参数编码

总而言之，对带有参数 `a_1, ..., a_n` 的函数 `f` 的调用被编码为：

```
function_selector(f) enc((a_1, ..., a_n))
```

`f` 的返回值 `v_1, ..., v_k` 会被编码为：

```
enc((v_1, ..., v_k))
```

也就是说，返回值会被组合为一个元组（tuple）进行编码。

3.23.8 示例

给定一个合约：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract Foo {
    function bar(bytes3[2] memory) public pure {}
    function baz(uint32 x, bool y) public pure returns (bool r) { r = x > 32 || y; }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

因此，对于我们的例子 `Foo`，如果我们想用 `69` 和 `true` 做参数调用 `baz`，我们总共需要传送 68 字节，可以分解为：

- `0xcdcd77c0`：方法 ID。这源自 ASCII 格式的 `baz(uint32,bool)` 签名的 Keccak 哈希的前 4 字节。
- `0x0000000000000000000000000000000000000000000000000000000000000045`：第一个参数，一个被用 0 值字节补充到 32 字节的 uint32 值 `69`。
- `0x0000000000000000000000000000000000000000000000000000000000000001`：第二个参数，一个被用 0 值字节补充到 32 字节的 boolean 值 `true`。

合起来就是：

```
0xcdcd77c00000000000000000000000000000000000000000000000000000000000000045000000000000000000000000000
```

它返回一个 `bool`。比如它返回 `false`，那么它的输出将是一个字节数组 `0x0000000000000000000000000000000000000000000000000000000000000000`，一个 `bool` 值。

如果我们想用 `["abc", "def"]` 做参数调用 `bar`，我们总共需要传送 68 字节，可以分解为：

- `0xfce353f6`：方法 ID。源自 `bar(bytes3[2])` 的签名。
- `0x6162630000000000000000000000000000000000000000000000000000000000`：第一个参数的第一部分，一个 `bytes3` 值 `"abc"`（左对齐）。
- `0x6465660000000000000000000000000000000000000000000000000000000000`：第一个参数的第二部分，一个 `bytes3` 值 `"def"`（左对齐）。

合起来就是：

```
0xfce353f661626300000000000000000000000000000000000000000000000000000000006465660000000000000000000000000000000000000000000000000000000000
```

如果我们想用 `"dave"`，`true` 和 `[1,2,3]` 作为参数调用 `sam`，我们总共需要传送 292 字节，可以分解为：

- `0xa5643bf2`：方法 ID。这是从签名 `sam(bytes,bool,uint256[])` 中导出的。注意，`uint` 被替换为其典型代表 `uint256`。
- `0x0000000000000000000000000000000000000000000000000000000000000060`：第一个参数（动态类型）的数据部分的位置，即从参数编码块开始位置算起的字节数。在这里，是 `0x60` 。
- `0x0000000000000000000000000000000000000000000000000000000000000001`：第二个参数：boolean 的 true。
- `0x00000000000000000000000000000000000000000000000000000000000000a0`：第三个参数（动态类型）的数据部分的位置，由字节数计量。在这里，是 `0xa0`。
- `0x0000000000000000000000000000000000000000000000000000000000000004`：第一个参数的数据部分，以字节数组的元素个数作为开始，在这里，是 4。
- `0x6461766500000000000000000000000000000000000000000000000000000000`：第一个参数的内容：`"dave"` 的 UTF-8 编码（在这里等同于 ASCII 编码），并在右侧（低位）用 0 值字节补充到 32 字节。
- `0x0000000000000000000000000000000000000000000000000000000000000003`：第三个参数的数据部分，以数组的元素个数作为开始，在这里，是 3。
- `0x0000000000000000000000000000000000000000000000000000000000000001`：第三个参数的第一个数组元素。
- `0x0000000000000000000000000000000000000000000000000000000000000002`：第三个参数的第二个数组元素。
- `0x0000000000000000000000000000000000000000000000000000000000000003`：第三个参数的第三个数组元素。

合起来就是：

```
0xa5643bf200000000000000000000000000000000000000000000000000000000000000600000000000000000000000000000
```

3.23.9 动态类型的使用

用值为 `(0x123, [0x456, 0x789], "1234567890", "Hello, world!")` 的签名参数调用函数 `f(uint256,uint32[],bytes10,bytes)`，其的编码方式如下：

我们取 `keccak("f(uint256,uint32[],bytes10,bytes)")` 的前四个字节，即 `0x8be65246`。然后我们对所有四个参数的头部部分进行编码。对静态类型 `uint256` 和 `bytes10`，这些是我们要直接传递的值，而对于动态类型 `uint32[]` 和 `bytes`，我们使用其数据区开始的偏移量，从需编码的值的开始位置算起（即不计算包含函数签名哈希值的前四个字节）。也就是：

- `0x0000000000000000000000000000000000000000000000000000000000000123`（`0x123` 补充到 32 字节）
- `0x0000000000000000000000000000000000000000000000000000000000000080`（第二个参数的数据部分起始位置的偏移量，4*32 字节，正好是头部的大小）
- `0x3132333435363738393000000000000000000000000000000000000000000000`（`"1234567890"` 从右边补充到 32 字节）
- `0x00000000000000000000000000000000000000000000000000000000000000e0`（第四个参数的数据部分起始位置的偏移量 = 第一个动态参数的数据部分起始位置的偏移量 + 第一个动态参数的数据部分的长度 = 4*32 + 3*32，参考后文）

在此之后，跟着第一个动态参数的数据部分，`[0x456, 0x789]`：

- `0x0000000000000000000000000000000000000000000000000000000000000002`（数组元素个数，2）
- `0x0000000000000000000000000000000000000000000000000000000000000456`（第一个数组元素）
- `0x0000000000000000000000000000000000000000000000000000000000000789`（第二个数组元素）

最后，我们将第二个动态参数的数据部分 `"Hello, world!"` 进行编码：

- `0x000000000000000000000000000000000000000000000000000000000000000d`（元素个数，在这里是字节数：13）
- `0x48656c6c6f2c20776f726c642100000000000000000000000000000000000000`（`"Hello, world!"` 从右边补充到 32 字节）

最后，合并到一起的编码就是（为了清晰，在函数选择器和每 32 字节之后加了换行）：

```
0x8be65246
  0000000000000000000000000000000000000000000000000000000000000123
```

(续下页)

(接上页)

```
0000000000000000000000000000000000000000000000000000000000000080
3132333435363738393000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000e0
0000000000000000000000000000000000000000000000000000000000000002
0000000000000000000000000000000000000000000000000000000000000456
0000000000000000000000000000000000000000000000000000000000000789
000000000000000000000000000000000000000000000000000000000000000d
48656c6c6f2c20776f726c642100000000000000000000000000000000000000
```

让我们使用相同的原理来对一个签名为 `g(uint256[][],string[])` ，参数值为 `([[1, 2], [3]], ["one", "two", "three"])` 的函数来进行编码；但从最原子的部分开始：

首先我们将第一个根数组 `[[1, 2], [3]]` 的第一个嵌入的动态数组 `[1, 2]` 的长度和数据进行编码：

- `0x0000000000000000000000000000000000000000000000000000000000000002` (第一个数组中的元素数量 2；元素本身是 `1` 和 `2`)
- `0x0000000000000000000000000000000000000000000000000000000000000001` (第一个元素)
- `0x0000000000000000000000000000000000000000000000000000000000000002` (第二个元素)

然后我们对第一个根数组 `[[1, 2], [3]]` 的第二个嵌入式动态数组 `[3]` 的长度和数据进行编码：

- `0x0000000000000000000000000000000000000000000000000000000000000001` (第二个数组中的元素数量 1；元素数据是 `3`)
- `0x0000000000000000000000000000000000000000000000000000000000000003` (第一个元素)

然后我们需要为各自的动态数组 `[1, 2]` 和 `[3]` 找到偏移量 `a` 和 `b`。为了计算偏移量，我们可以看一下第一个根数组的编码数据 `[[1, 2], [3]]` 在编码中枚举每一行。

```
0 - a                                                                - [1, 2] 的偏移量
1 - b                                                                - [3] 的偏移量
2 - 0000000000000000000000000000000000000000000000000000000000000002 - [1, 2] 数组的计数
3 - 0000000000000000000000000000000000000000000000000000000000000001 - 1 的编码
4 - 0000000000000000000000000000000000000000000000000000000000000002 - 2 的编码
5 - 0000000000000000000000000000000000000000000000000000000000000001 - [3] 数组的计数
6 - 0000000000000000000000000000000000000000000000000000000000000003 - 3 的编码
```

偏移量 `a` 指向数组 `[1, 2]` 内容的开始位置，即第 2 行的开始（64 字节）；所以 `a = 0x0000000000000000000000000000000000000000000000000000000000000040`。

偏移量 `b` 指向数组 `[3]` 内容的开始位置，即第 5 行的开始（160 字节）；所以 `b = 0x00000000000000000000000000000000000000000000000000000000000000a0`。

然后我们对第二个根数组的嵌入字符串进行编码：

- `0x0000000000000000000000000000000000000000000000000000000000000003` （单词 `"one"` 中的字符个数）
- `0x6f6e650000000000000000000000000000000000000000000000000000000000` （单词 `"one"` 的 utf8 编码）
- `0x0000000000000000000000000000000000000000000000000000000000000003` （单词 `"two"` 中的字符个数）
- `0x74776f0000000000000000000000000000000000000000000000000000000000` （单词 `"two"` 的 utf8 编码）
- `0x0000000000000000000000000000000000000000000000000000000000000005` （单词 `"three"` 中的字符个数）
- `0x7468726565000000000000000000000000000000000000000000000000000000` （单词 `"three"` 的 utf8 编码）

作为与第一个根数组的并列，因为字符串也属于动态元素，我们也需要找到它们的偏移量 `c`, `d` 和 `e`：

```
0 - c                                                                - "one" 的偏移量
1 - d                                                                - "two" 的偏移量
2 - e                                                                - "three" 的偏移量
3 - 0000000000000000000000000000000000000000000000000000000000000003 - "one" 的字符计数
4 - 6f6e650000000000000000000000000000000000000000000000000000000000 - "one" 的编码
5 - 0000000000000000000000000000000000000000000000000000000000000003 - "two" 的字符计数
6 - 74776f0000000000000000000000000000000000000000000000000000000000 - "two" 的编码
7 - 0000000000000000000000000000000000000000000000000000000000000005 - "three" 的字符计数
8 - 7468726565000000000000000000000000000000000000000000000000000000 - "three" 的编码
```

偏移量 `c` 指向字符串 `"one"` 内容的开始位置，即第 3 行的开始（96 字节）；所以 `c = 0x0000000000000000000000000000000000000000000000000000000000000060`。

偏移量 `d` 指向字符串 `"two"` 内容的开始位置，即第 5 行的开始（160 字节）；所以 `d = 0x00000000000000000000000000000000000000000000000000000000000000a0`。

偏移量 `e` 指向字符串 `"three"` 内容的开始位置，即第 7 行的开始（224 字节）；所以 `e = 0x00000000000000000000000000000000000000000000000000000000000000e0`。

注意，根数组的嵌入元素的编码并不互相依赖，且具有对于函数签名 `g(string[],uint256[][])` 所相同的编码。

然后我们对第一个根数组的长度进行编码：

- `0x0000000000000000000000000000000000000000000000000000000000000002` (第一个根数组的元素数量 2；这些元素本身是 `[1, 2]` 和 `[3]`)

而后我们对第二个根数组的长度进行编码：

- `0x0000000000000000000000000000000000000000000000000000000000000003` (第二个根数组的元素数量 3；这些字符串本身是 `"one"`，`"two"` 和 `"three"`)

最后，我们找到根动态数组元素 `[[1, 2], [3]]` 和 `["one", "two", "three"]` 的偏移量 `f` 和 `g`。汇编数据的正确顺序如下：

```
0x2289b18c                                                          - 函数签名
 0 - f                                                              - [[1, 2], [3]] 的偏移量
 1 - g                                                              - ["one", "two", "three"] 的偏移量
 2 - 0000000000000000000000000000000000000000000000000000000000000002 - [[1, 2], [3]] 的元素计数
 3 - 0000000000000000000000000000000000000000000000000000000000000040 - [1, 2] 的偏移量
 4 - 00000000000000000000000000000000000000000000000000000000000000a0 - [3] 的偏移量
 5 - 0000000000000000000000000000000000000000000000000000000000000002 - [1, 2] 的元素计数
 6 - 0000000000000000000000000000000000000000000000000000000000000001 - 1 的编码
 7 - 0000000000000000000000000000000000000000000000000000000000000002 - 2 的编码
 8 - 0000000000000000000000000000000000000000000000000000000000000001 - [3] 的元素计数
 9 - 0000000000000000000000000000000000000000000000000000000000000003 - 3 的编码
10 - 0000000000000000000000000000000000000000000000000000000000000003 - ["one", "two", "three"] 的元素计数
11 - 0000000000000000000000000000000000000000000000000000000000000060 - 的偏移量"one"
12 - 00000000000000000000000000000000000000000000000000000000000000a0 - 的偏移量"two"
13 - 00000000000000000000000000000000000000000000000000000000000000e0 - 的偏移量"three"
14 - 0000000000000000000000000000000000000000000000000000000000000003 - "one" 的字符计数
15 - 6f6e650000000000000000000000000000000000000000000000000000000000 - "one" 的编码
16 - 0000000000000000000000000000000000000000000000000000000000000003 - "two" 的字符计数
17 - 74776f0000000000000000000000000000000000000000000000000000000000 - "two" 的编码
18 - 0000000000000000000000000000000000000000000000000000000000000005 - "three" 的字符计数
19 - 7468726565000000000000000000000000000000000000000000000000000000 - "three" 的编码
```

偏移量 `f` 指向数组 `[[1, 2], [3]]` 内容的开始位置，即第 2 行的开始（64 字节）；所以 `f = 0x0000000000000000000000000000000000000000000000000000000000000040`。

偏移量 `g` 指向数组 `["one", "two", "three"]` 内容的开始位置，即第 10 行的开始（320 字节）；所以 `g`

= 0x0000000000000000000000000000000000000000000000000000000000000140。

3.23.10 事件

事件是 Ethereum 日志/事件观察协议的一个抽象。日志条目提供了合约的地址，一系列最多四个主题和一些任意长度的二进制数据。事件利用现有的函数 ABI，以便将其（连同接口规范）解释为一个正确的类型化结构。

给定一个事件名称和一系列的事件参数，我们把它们分成两个子系列：那些有索引的和那些没有索引的。那些被索引的参数，可能多达 3 个（对于非匿名事件）或 4 个（对于匿名事件），与事件签名的 Keccak 散列一起使用，形成日志条目的主题。那些没有索引的则构成事件的字节数组。

实际上，使用该 ABI 的日志条目被描述为:

- `address`：合约的地址（由以太坊真正提供）；
- `topics[0]`：`keccak(EVENT_NAME+"("+EVENT_ARGS.map(canonical_type_of).join(",")+")")` `canonical_type_of` 是一个可以返回给定参数的权威类型的函数，例如，对 `uint indexed foo` 它会返回 `uint256`）。如果事件被声明为 `anonymous`，那么 `topics[0]` 不会被生成；
- `topics[n]`：如果事件没有被声明为 `anonymous`，则为 `abi_encode(EVENT_INDEXED_ARGS[n - 1])` 或者如果它被声明为该类型，则为 `abi_encode(EVENT_INDEXED_ARGS[n])`（`EVENT_INDEXED_ARGS` 是被索引的 `EVENT_ARGS` 的系列）；
- `data`：`EVENT_NON_INDEXED_ARGS` 的 ABI 编码（`EVENT_NON_INDEXED_ARGS` 是一系列没有索引的 `EVENT_ARGS`，`abi_encode` 是 ABI 编码函数，用于从一个函数返回一系列类型的值，如上所述）。

对于所有长度不超过 32 字节的类型，`EVENT_INDEXED_ARGS` 数组直接包含数值，填充或符号扩展（对于有符号整数）到 32 字节，就像常规 ABI 编码一样。然而，对于所有 “复杂” 类型或动态长度的类型，包括所有数组，`string`，`bytes` 和结构，`EVENT_INDEXED_ARGS` 将包含 *Keccak* 散列的特殊就地编码值（见索引事件参数的编码），而不是直接编码的值。这允许应用程序有效地查询动态长度类型的值（通过设置编码值的哈希值作为主题），但使应用程序无法解码他们没有查询到的索引值。对于动态长度类型，应用程序开发人员面临着对预定值的快速搜索（如果参数有索引）和任意值的可读性之间的权衡（这要求参数不被索引）。开发者可以通过定义具有两个参数的事件 -- 一个是索引的，一个是不索引的 -- 来克服这种权衡，实现高效搜索和任意可读性。

3.23.11 错误

在合约内部发生故障的情况下，合约可以使用一个特殊的操作码来中止执行，并恢复所有的状态变化。除了这些效果之外，描述性数据可以返回给调用者。这种描述性数据是对一个错误及其参数的编码，其方式与函数调用的数据相同。

作为一个例子，让我们考虑以下合约，它的 `transfer` 函数总是以” 余额不足” 的自定义错误返回。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
```

(续下页)

(接上页)

```
contract TestToken {
    error InsufficientBalance(uint256 available, uint256 required);
    function transfer(address /*to*/, uint amount) public pure {
        revert InsufficientBalance(0, amount);
    }
}
```

返回数据的编码方式与函数 `InsufficientBalance(0, amount)` 对函数 `InsufficientBalance(uint256,uint256)` 的调用方式相同。即 `0xcf479181`, `uint256(0)`, `uint256(amount)`。

错误选择器 `0x00000000` 和 `0xffffffff` 是保留给将来使用的。

> **警告：** 永远不要相信错误数据。默认情况下，错误数据通过外部调用在链向上冒泡产生，这意味着一个合约可能会收到一个它直接调用的任何合约中没有定义的错误。此外，任何合约都可以通过返回与错误签名相匹配的数据来伪造任何错误，即使该错误没有在任何地方定义。

3.23.12 JSON

合约接口的 JSON 格式是由一个函数，事件和错误描述的数组给出的。一个函数描述是一个带有字段的 JSON 对象：

- `type`: `"function"`, `"constructor"`, `"receive"`（*"接收以太币"函数*）或者 `"fallback"`（*"默认"函数*）;
- `name`: 函数名称;
- `inputs`: 数组对象，每个数组对象会包含:
  - `name`: 参数名称;
  - `type`: 参数的权威类型（详见下文）
  - `components`: 供元组（tuple）类型使用（详见下文）
- `outputs`: 一个类似于 `inputs` 的数组对象。
- `stateMutability`: 为下列值之一: `pure`（*指定为不读取区块链状态*）, `view`（*指定为不修改区块链状态*）, `nonpayable`（函数不接受以太币 - 默认选项）和 `payable`（函数可接收以太币）。

构造函数（constructor），receive 函数和 fallback 函数没有 `name` 或 `outputs` 属性。receive 函数和 fallback 函数也没有 `inputs` 属性。

备注： 向不接收以太币函数发送非零的以太币将使交易回滚。

备注： 在 Solidity 中，状态可变性 不可支付是完全不指定状态可变性时的修饰语。

一个事件描述是一个有极其相似字段的 JSON 对象：

- `type`：总是 `"event"`
- `name`：事件名称；
- `inputs`：对象数组，每个数组对象会包含：
  - `name`：参数名称。
  - `type`：参数的规范类型（详见下文）。
  - `components`：供元组（tuple）类型使用（详见下文）
  - `indexed`：如果该字段是日志主题的一部分，则为 `true`，如果它是日志数据段之一，则为 `false`。
- `anonymous`：如果事件被声明为 `anonymous`，则为 `true`。

错误消息如下：

- `type`：总是 `"error"`
- `name`：错误名称；
- `inputs`：对象数组，每个数组对象会包含：
  - `name`：参数名称。
  - `type`：参数的权威类型（相见下文）。
  - `components`：供元组（tuple）类型使用（详见下文）。

备注： 在 JSON 数组中可能有多个具有相同名称的错误，甚至具有相同的签名；例如，如果错误源自合约中的不同文件或从另一个合约引用。对于 ABI 来说，只有错误本身的名称是相关的，而不是它的定义位置。

例如，

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;


contract Test {
    constructor() { b = hex"12345678901234567890123456789012"; }
```

(续下页)

(接上页)

```
    event Event(uint indexed a, bytes32 b);
    event Event2(uint indexed a, bytes32 b);
    error InsufficientBalance(uint256 available, uint256 required);
    function foo(uint a) public { emit Event(a, b); }
    bytes32 b;
}
```

可由如下 JSON 来表示：

```
[{
"type":"error",
"inputs": [{"name":"available","type":"uint256"},{"name":"required","type":"uint256"}
↪],
"name":"InsufficientBalance"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event"
}, {
"type":"event",
"inputs": [{"name":"a","type":"uint256","indexed":true},{"name":"b","type":"bytes32",
↪"indexed":false}],
"name":"Event2"
}, {
"type":"function",
"inputs": [{"name":"a","type":"uint256"}],
"name":"foo",
"outputs": []
}]
```

处理元组类型

尽管名称被有意地不作为 ABI 编码的一部分，但将它们包含进 JSON 来显示给最终用户是非常合理的。其结构会按下列方式进行嵌套：

一个拥有 `name`，`type` 和潜在的 `components` 成员的对象描述了某种类型的变量。直至到达一个元组（tuple）类型且到那点的存储在 `type` 属性中的字符串以 `tuple` 为前缀，也就是说，在 `tuple` 之后紧跟一个 `[]` 或有整数 `k` 的 `[k]`，才能确定一个元组。元组的组件元素会被存储在成员 `components` 中，它是一个数组类型，且与顶级对象具有同样的结构，只是在这里不允许 已索引的 (`indexed`) 数组元素。

示例代码：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.5 <0.9.0;
pragma abicoder v2;

contract Test {
    struct S { uint a; uint[] b; T[] c; }
    struct T { uint x; uint y; }
    function f(S memory, T memory, uint) public pure {}
    function g() public pure returns (S memory, T memory, uint) {}
}
```

可由如下 JSON 来表示:

```
[
  {
    "name": "f",
    "type": "function",
    "inputs": [
      {
        "name": "s",
        "type": "tuple",
        "components": [
          {
            "name": "a",
            "type": "uint256"
          },
          {
            "name": "b",
            "type": "uint256[]"
          },
          {
            "name": "c",
            "type": "tuple[]",
            "components": [
              {
                "name": "x",
                "type": "uint256"
              },
              {
                "name": "y",
                "type": "uint256"
              }
            ]
          }
```

(续下页)

(接上页)

```
        ]
      },
      {
        "name": "t",
        "type": "tuple",
        "components": [
          {
            "name": "x",
            "type": "uint256"
          },
          {
            "name": "y",
            "type": "uint256"
          }
        ]
      },
      {
        "name": "a",
        "type": "uint256"
      }
    ],
    "outputs": []
  }
]
```

3.23.13 严格的编码模式

严格的编码模式是指导致与上述正式规范中定义的编码完全相同的模式。这意味着偏移量必须尽可能小，同时还不能在数据区域产生重叠，因此不允许有间隙。

通常，ABI 解码器是通过遵循偏移指针以简单的方式编写的，但有些解码器可能会强制执行严格模式。Solidity ABI 解码器目前并不强制执行严格模式，但编码器总是以严格模式创建数据。

3.23.14 非标准打包模式

通过 `abi.encodePacked()`，Solidity 支持一种非标准的打包模式，其中：

- 短于 32 字节的类型直接连接，没有填充或符号扩展。
- 动态类型是直接编码的，没有长度。
- 数组元素被填充，但仍被是直接编码

此外，不支持结构以及嵌套数组。

例如，对 `int16(-1), bytes1(0x42), uint16(0x03), string("Hello, world!")` 进行编码将生成如下结果

```
0xffff42000348656c6c6f2c20776f726c6421
  ^^^^                                 int16(-1)
      ^^                               bytes1(0x42)
        ^^^^                           uint16(0x03)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^ 字符串("Hello, world!") 没有长度字段
```

更具体地说：

- 在编码过程中，所有东西都是直接编码的。这意味着没有像 ABI 编码那样区分头和尾，也没有对数组的长度进行编码。
- `abi.encodePacked` 的直接参数被编码，只要不是数组（或 `string` 或 `bytes` ），就不需要填充。
- 一个数组的编码是其元素的编码 **与** 填充的连接。
- 动态大小的类型，如 `string`，`bytes` 或 `uint[]`，在编码时没有长度字段。
- `string` 或 `bytes` 的编码不会在末尾应用填充，除非它是数组或结构体的一部分（然后它被填充为 32 字节的倍数）。

一般来说，只要有两个动态大小的元素，编码就会模糊不清，因为缺少长度字段。

如果需要填充，可以使用明确的类型转换：`abi.encodePacked(uint16(0x12)) == hex"0012"`。

由于在调用函数时不使用打包编码，所以没有特别支持预留函数选择器。由于编码是模糊的，所以没有解码功能。

> **警告：** 如果使用 `keccak256(abi.encodePacked(a, b))` 并且 `a` 和 `b` 都是动态类型，那么通过将 `a` 的部分移动到 `b` 中，很容易在哈希值中产生冲突，反之亦然。更具体地说，`abi.encodePacked("a", "bc") == abi.encodePacked("ab", "c")`。如果你使用 `abi.encodePacked` 进行签名、认证或数据完整性，确保总是使用相同的类型，并检查其中最多一个是动态的。除非有令人信服的理由，否则应首选 `abi.encode`。

3.23.15 索引事件参数的编码

不属于值类型的索引事件参数，即数组和结构，不直接存储，而是存储一个编码的 Keccak-256 哈希值。这个编码的定义如下：

- `bytes` 和 `string` 值的编码只是字符串的内容，没有任何填充或长度前缀。
- 结构的编码是其成员编码的串联，总是填充为 32 字节的倍数（甚至是 `bytes` 和 `string`）。
- 数组的编码（包括动态和静态大小）是其元素编码的连接，总是填充为 32 字节的倍数（甚至是 `bytes` 和 `string`），没有任何长度前缀。

在上面，像往常一样，一个负数被填充符号扩展，而不是零填充。`bytesNN` 类型被填充在右边，而 `uintNN` / `intNN` 被填充在左边。

> **警告：** 如果一个结构包含一个以上的动态大小的数组，那么它的编码是不明确的。正因为如此，要经常重新检查事件数据，不要只依赖基于索引参数的搜索结果。

3.24 安全考虑

虽然通常很容易建立起按预期工作的软件，但要检查没有人能够以 **非**预期的方式使用它，就难得多了。

在 Solidity 中，这一点更加重要，因为您可以使用智能合约来处理代币，甚至可能是更有价值的东西。此外，智能合约的每一次执行都是公开的，除此之外，源代码也通常是容易获得的。

当然，您总是要考虑有多大的风险：您可以将智能合约与一个对公众开放（因此也对恶意行为者开放），甚至可能是开源的网络服务进行比较。如果您只在该网络服务上存储您的杂货清单，您可能不必太过小心，但如果您使用该网络服务管理您的银行账户，您就应该更加小心。

本节将列出一些陷阱和一般安全建议，但当然，这不可能是完整的。此外，请记住，即使您的智能合约代码没有错误，编译器或平台本身也可能有一个错误。编译器的一些公开的，与安全有关的错误列表可以在已知错误列表 中找到，它也是机器可读的。请注意，有一个涵盖 Solidity 编译器的代码生成器的 错误赏金计划。

像往常一样，对于开源文档，请帮助我们扩展这部分内容（尤其是，一些例子不会有什么影响）！

注意：除了下面的列表，您也可以在 Guy Lando 的知识列表 和 Consensys GitHub 代码仓库 中找到更多的安全建议和最佳实践。

3.24.1 陷阱

隐私信息和随机性

您在智能合约中使用的所有东西都是公开可见的，即使是标记为 `private` 的局部变量和状态变量。

如果你不希望区块构造者能够作弊，在智能合约中使用随机数是相当棘手的。

重入

一个合约（A）与另一个合约（B）的任何交互和任何以太币的转移都会将控制权交给该合约（B）。这使得 B 有可能在这个交互完成之前回调回 A。举个例子，下面的代码包含了一个错误（这只是一个片段，而不是一个完整的合约）：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;
```

(续下页)

(接上页)

```
// 此合约包含一个错误 - 请勿使用
contract Fund {
    /// @dev 合约的以太币份额的映射。
    mapping(address => uint) shares;
    /// 提取您的份额。
    function withdraw() public {
        if (payable(msg.sender).send(shares[msg.sender]))
            shares[msg.sender] = 0;
    }
}
```

这里的问题不是太严重，因为作为 send 的一部分，gas 有限，但它仍然暴露了一个弱点: 以太币的转移总是可以包括代码的执行，所以接收者可以是一个回调到 withdraw 的合约。这将让它获得多次退款，并基本上取回合约中的所有以太。特别的是，下面的合约将允许攻击者多次退款，因为它使用了 call，它会默认转发所有剩余 gas。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

//此合约包含一个错误 - 请勿使用
contract Fund {
    /// @dev 合约的以太币份额的映射。
    mapping(address => uint) shares;
    /// 提取您的份额。
    function withdraw() public {
        (bool success,) = msg.sender.call{value: shares[msg.sender]}("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

为了避免重入，您可以使用如下所示的检查-生效-交互（Checks-Effects-Interactions）模式:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Fund {
    /// @dev 合约的以太币份额的映射。
    mapping(address => uint) shares;
    /// 提取您的份额。
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        payable(msg.sender).transfer(share);
```

(续下页)

(接上页)

```
    }
}
```

检查-生效-交互模式确保所有通过合约的代码路径在修改合约的状态（检查）之前完成对所提供的参数的所有必要检查；只有这样它才会对状态进行任何改变（生效）；之后所有计划的状态改变被写入存储（交互）。这是一种常见的防止 *重入性攻击* 的万无一失的方法，在这种情况下，外部调用的恶意合约能够通过使用在原始合约最终完成交易之前回调原始合约的逻辑来重复花费授权额度，重复提取余额，以及其他事情。

请注意，重入不仅是对以太币转移的影响，也是对另一个合约的任何函数调用的影响。此外，您还必须考虑到多合约的情况。一个被调用的合约可以修改您所依赖的另一个合约的状态。

gas 限制和循环

对于没有固定迭代次数的循环，例如，依赖于存储值的循环，必须谨慎使用：由于块 gas 的限制，事务只能消耗一定量的 gas。无论是明确的还是仅仅由于正常的操作，循环中的迭代次数可以增长到超过块 gas 限制，这可能导致完整的合约在某一点上停滞。这可能不适用于只为从区块链上读取数据而执行的 `view` 函数。但是，这样的函数可能会被其他合约调用，作为链上操作的一部分，并使其停滞。请在您的合约文档中明确说明这种情况。

发送和接收以太币

- 无论是合约还是"外部账户"，目前都无法阻止有人向他们发送以太币。合约可以对普通的转账做出反应并拒绝，但有一些方法可以在不创建消息调用的情况下转移以太币。一种方法是简单地向合约地址"挖矿"，第二种方法是使用 `selfdestruct(x)`。
- 如果一个合约收到了以太（没有函数被调用），要么是执行*receive 方法*，要么执行*fallback* 函数。如果它没有 `receive` 也没有 `fallback` 函数，那么该以太将被拒绝（抛出一个异常）。在这些函数的执行过程中，合约只能依靠此时它所传递的"gas 津贴"（2300 gas）可用。但这个津贴不足以修改存储（但不要认为这是理所当然的，这个津贴可能会随着未来的硬分叉而改变）。为了确保您的合约能够以这种方式接收以太，请检查 receive 和 fallback 函数的 gas 要求（在 Remix 的"详细"章节会举例说明）。
- 有一种方法可以使用 `addr.call{value: x}("")` 将更多的 gas 转发给接收合约。这与 `addr.transfer(x)` 本质上是一样的，只是它转发了所有剩余的 gas，并为接收方提供了执行更昂贵的操作的能力（而且它返回一个失败代码，而不是自动传播错误）。这可能包括回调到发送合约或其他您可能没有想到的状态变化。因此，这种方法无论是给诚实用户还是恶意行为者都提供了极大的灵活性。
- 尽可能使用最精确的单位来表示 wei 的数量，因为您会因为缺乏精确性而失去任何四舍五入的结果。
- 如果您想用 `address.transfer` 来发送以太，有一些细节需要注意：
  1. 如果接收者是一个合约，它会导致其 receive 或 fallback 函数被执行，而该函数又可以回调发送以太的合约。

2. 发送以太可能由于调用深度超过 1024 而失败。由于调用者完全控制着调用深度，他们可以迫使传输失败；考虑到这种可能性，或者使用 `send`，并确保总是检查其返回值。更好的办法是，使用接收者可以提取以太币的模式来编写您的合约。

3. 发送以太也可能失败，因为接收合约的执行需要超过分配的 gas 值（确切地说，是使用了*require*，*assert*，*revert* 或者因为操作太昂贵）- 它“耗尽了 gas”（OOG）。如果您使用 `transfer` 或 `send`，并带有返回值检查，这可能为接收者提供一种手段来阻止发送合约的进展。同样，这里的最佳做法是使用"提款"模式而不是"发送"模式。

调用栈深度

外部函数调用随时都可能失败，因为它们超过了最大调用堆栈大小 1024 的限制。在这种情况下，Solidity 会抛出一个异常。恶意的行为者可能会在与您的合约交互之前，将调用堆栈逼到一个高值。请注意，由于 桔子哨子（Tangerine Whistle） 硬分叉，63/64 规则 使得调用栈深度攻击不切实际。还要注意的是，调用栈和表达式栈是不相关的，尽管两者都有 1024 个栈槽的大小限制。

注意 `.send()` 在调用栈被耗尽的情况下 **不会**抛出异常，而是会返回 `false`。低级函数 `.call()`，`.delegatecall()` 和 `.staticcall()` 也都是这样的。

授权的代理

如果您的合约可以作为一个代理，也就是说，如果它可以用用户提供的数据调用任意的合约，那么用户基本上可以承担代理合约的身份。即使您有其他的保护措施，最好是建立您的合约系统，使代理没有任何权限（甚至对自己也没有）。如果需要，您可以使用第二个代理来完成：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract ProxyWithMoreFunctionality {
    PermissionlessProxy proxy;

    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return proxy.callOther(addr, payload);
    }
    // 其他函数和其他功能
}

// 这是完整的合约，它没有其他功能，不需要任何权限就可以工作。
contract PermissionlessProxy {
    function callOther(address addr, bytes memory payload) public
            returns (bool, bytes memory) {
        return addr.call(payload);
    }
}
```

tx.origin

永远不要使用 tx.origin 做身份认证。假设您有一个这样的钱包合约:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 本合约包含一个错误 - 请勿使用
contract TxUserWallet {
    address owner;

    constructor() {
        owner = msg.sender;
    }

    function transferTo(address payable dest, uint amount) public {
        // 错误就在这里，您必须使用 msg.sender 而不是 tx.origin。
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

现在有人欺骗您，让您向这个攻击钱包的地址发送以太币:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
interface TxUserWallet {
    function transferTo(address payable dest, uint amount) external;
}

contract TxAttackWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        TxUserWallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

如果您的钱包检查了 msg.sender 的授权，它将得到攻击钱包的地址，而不是所有者地址。但是通过检查 tx.origin，它得到的是启动交易的原始地址，这仍然是所有者地址。攻击钱包会立即耗尽您的所有资金。

二进制补码 / 下溢 / 上溢

正如在许多编程语言中，Solidity 的整数类型实际上不是整数。当数值较小时，它们类似于整数，但也不能表示任意大的数字。

下面的代码会导致溢出，因为加法的结果太大，不能存储在 uint8 类型中：

```
uint8 x = 255;
uint8 y = 1;
return x + y;
```

Solidity 有两种模式来处理这些溢出。检查和不检查或 "包装" 模式。

默认的检查模式将检测到溢出并导致一个失败的断言。您可以使用 unchecked { ... }，使溢出被无声地忽略。上面的代码如果用 unchecked { ... } 包装，将返回 0。

即使在检查模式下，也不要认为您受到了保护，不会出现溢出错误。在这种模式下，溢出总是会被还原。如果无法避免溢出，这可能导致智能合约被卡在某个状态。

一般来说，请阅读关于二进制补码表示法的限制，它甚至对有符号的数字有一些更特殊的边缘情况。

尝试使用 require 将输入的大小限制在一个合理的范围内，并使用*SMT 检查器* 来发现潜在的溢出。

清除映射

Solidity mapping 类型（参见映射类型）是一个仅有存储空间的键值数据结构，它不跟踪被分配非零值的键。正因为如此，清理映射时不可能有关于写入键的额外信息。如果 mapping 被用作动态存储数组的基本类型，删除或弹出数组将不会对 mapping 元素产生影响。例如，如果一个 mapping 被用作一个 struct 的成员字段的类型，而该结构是一个动态存储阵列的基本类型，同样的情况也会发生。mapping 在包含 mapping 的结构或数组的分配中也会被忽略。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

contract Map {
    mapping(uint => uint)[] array;

    function allocate(uint newMaps) public {
        for (uint i = 0; i < newMaps; i++)
            array.push();
    }

    function writeMap(uint map, uint key, uint value) public {
        array[map][key] = value;
    }
```

(续下页)

(接上页)

```
    function readMap(uint map, uint key) public view returns (uint) {
        return array[map][key];
    }

    function eraseMaps() public {
        delete array;
    }
}
```

考虑一下上面的例子和下面的调用序列：allocate(10)，writeMap(4, 128, 256)。此时，调用 readMap(4, 128) 返回 256。如果我们调用 eraseMaps，状态变量 array 的长度被清零，但由于它的 mapping 元素不能被清零，它们的信息在合约的存储中仍然存在。删除 array 后，调用 allocate(5) 允许我们再次访问 array[4]，调用 readMap(4, 128) 则返回 256，即使没有再次调用 writeMap。

如果您的 mapping 信息必须被删除，可以考虑使用类似于 可迭代的映射 的库，它允许您在适当的 mapping 中遍历键并删除其值。

细枝末节

- 没有占满 32 字节的类型可能包含 "脏高位"。这在当您访问 msg.data 的时候尤为重要——它带来了延展性风险：您既可以用原始字节 0xff000001，也可以用 0x00000001 作为参数来调用函数 f(uint8 x) 以构造交易。这两个参数都会被正常提供给合约，就 x 而言，两者看起来都是数字 1，但 msg.data 将是不同的，所以如果您无论怎么使用 keccak256(msg.data)，您都会得到不同的结果。

3.24.2 推荐做法

认真对待警告

如果编译器警告您一些事情，您应该改变它。即使您不认为这个特定的警告有安全问题，但也可能在它下面埋藏着另一个问题。我们发出的任何编译器警告都可以通过对代码的轻微修改来消除。

始终使用最新版本的编译器，以获知所有最近引入的警告。

编译器发出的 info 类型的信息并不危险，只是代表编译器认为可能对用户有用的额外建议和可选信息。

限制以太币的数量

限制智能合约中可存储的以太币（或其他代币）的数量。如果您的源代码，编译器或平台有错误，这些资金可能会丢失。如果您想限制您的损失，就限制以太币的数量。

保持合约简练且模块化

保持您的合约短小而容易理解。把不相关的功能单独放在其他合约中或放在库合约中。关于源代码质量的一般建议当然也适用：限制局部变量的数量和函数的长度，等等。给您的函数添加注释，这样别人就可以看到您的意图是什么，并判断它是否与代码的作用不同。

使用"检查-生效-交互"（Checks-Effects-Interactions）模式

大多数函数会首先进行一些检查，并且应该首先完成这些检查（谁调用了这个函数，参数是否在范围内，他们是否发送了足够的以太，这个人是否有代币，等等）。

第二步，如果所有的检查都通过了，就应该对当前合约的状态变量进行影响。与其他合约的交互应该是任何函数的最后一步。

早期的合约延迟了一些效果，等待外部函数调用在非错误状态下返回。这往往是一个严重的错误，因为上面解释了重入问题。

请注意，对已知合约的调用也可能反过来导致对未知合约的调用，因此，最好总是应用这种模式。

包含故障-安全（Fail-Safe）模式

尽管将系统完全去中心化可以省去许多中间环节，但包含某种故障-安全模式仍然是好的做法，尤其是对于新的代码来说：

您可以在您的智能合约中添加一个功能，执行一些自我检查，如"是否有任何以太币泄漏?"，"代币的总和是否等于合约的余额?"或类似的事情。请记住，您不能为此使用太多的 gas，所以可能需要通过链外计算的帮助。

如果自我检查失败，合约会自动切换到某种"故障安全"模式，例如，禁用大部分功能，将控制权移交给一个固定的，可信赖的第三方，或者只是将合约转换为一个简单的"退回我的钱"的合约。

请求同行评审

检查一段代码的人越多，发现的问题就越多。要求其他人审查您的代码也有助于作为交叉检查，找出您的代码是否容易理解 - 这是好的智能合约的一个非常重要的标准。

3.25 已知 bug 列表

下面，您可以找到一个 JSON 格式的列表，其中包括 Solidity 编译器中一些已知的与安全有关的错误。该文件本身托管在 Github 仓库。该列表最早可以追溯到 0.3.0 版本，只有在此之前的版本中已知的错误没有列出。

还有一个文件叫 bugs_by_version.json，它可以用来检查哪些 bug 影响到特定版本的编译器。

合约源码验证工具以及其他与合约交互的工具应根据以下标准查阅此列表：

- 如果合约是用夜间编译器版本而不是发布版本编译的，那就有点可疑了。此列表不跟踪未发布或夜间版本。
- 如果合约的编译版本不是合约创建时的最新版本，则也有点可疑。对于从其他合约创建的合约，您必须按照创建链返回到事务，并使用该事务的日期作为创建日期。
- 如果合约是用包含已知 bug 的编译器编译的，并且合约是在已发布包含修复程序的较新编译器版本时创建的，则这是高度可疑的。

下面的已知错误的 JSON 文件是一个对象数组，每个错误都有一个对象，其键值如下：

uid
: 以 `SOL-<year>-<number>` 的形式给予该错误的唯一标识符。有可能存在多个具有相同 uid 的条目。这意味着多个版本范围受到同一错误的影响。

name
: 给予该错误的唯一名称

summary
: 对该错误的简短描述

description
: 该错误的详细描述

link
: 有更多详细信息的网站的 URL，可选

introduced
: 第一个包含该错误的发布的编译器版本，可选

fixed
: 第一个不再包含该错误的发布的编译器版本

publish
: bug 公开的日期，可选

severity
: bug 的严重程度：非常低，低，中，高。考虑合约测试中的可发现性，发生的可能性和错误造成的潜在损害。

conditions
: 必须满足的条件才能触发该错误。可以使用以下键：`optimizer`，布尔值，表示优化器必须打开才会出

现该错误。evmVersion，一个字符串，表示哪个 EVM 版本的编译器设置触发了该错误。这个字符串可以包含比较运算符。例如，">=constantinople" 表示当 EVM 版本设置为 constantinople 或更高时，该错误就会出现。如果没有给出条件，则假定该错误存在。

check

这个字段包含不同的检查，报告智能合约是否包含错误。第一种类型的检查是 JavaScript 正则表达式，如果存在该错误，将与源代码（"source-regex"）进行匹配。如果没有匹配，那么该漏洞很可能不存在。如果有一个匹配，则该错误可能存在。为了提高准确性，检查应该在剥离注释后应用于源代码。第二种类型的检查是在 Solidity 程序的紧凑 AST 上检查的模式（"ast-compact-json path"）。指定的搜索查询是一个 JsonPath 表达式。如果 Solidity AST 中至少有一个路径与该查询相匹配，则可能存在错误。

```
[
    {
        "uid": "SOL-2022-7",
        "name": "StorageWriteRemovalBeforeConditionalTermination",
        "summary": "Calling functions that conditionally terminate the external EVM␣
↪call using the assembly statements ``return(...)`` or ``stop()`` may result in␣
↪incorrect removals of prior storage writes.",
        "description": "A call to a Yul function that conditionally terminates the␣
↪external EVM call could result in prior storage writes being incorrectly removed by␣
↪the Yul optimizer. This used to happen in cases in which it would have been valid␣
↪to remove the store, if the Yul function in question never actually terminated the␣
↪external call, and the control flow always returned back to the caller instead.␣
↪Conditional termination within the same Yul block instead of within a called␣
↪function was not affected. In Solidity with optimized via-IR code generation, any␣
↪storage write before a function conditionally calling ``return(...)`` or ``stop()``␣
↪in inline assembly, may have been incorrectly removed, whenever it would have been␣
↪valid to remove the write without the ``return(...)`` or ``stop()``. In optimized␣
↪legacy code generation, only inline assembly that did not refer to any Solidity␣
↪variables and that involved conditionally-terminating user-defined assembly␣
↪functions could be affected.",
        "link": "https://blog.soliditylang.org/2022/09/08/storage-write-removal-
↪before-conditional-termination/",
        "introduced": "0.8.13",
        "fixed": "0.8.17",
        "severity": "medium/high",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-6",
        "name": "AbiReencodingHeadOverflowWithStaticArrayCleanup",
        "summary": "ABI-encoding a tuple with a statically-sized calldata array in␣
↪the last component would corrupt 32 leading bytes of its first dynamically encoded␣
```

(续下页)

(接上页)


```
↪component.",
        "description": "When ABI-encoding a statically-sized calldata array, the␣
↪compiler always pads the data area to a multiple of 32-bytes and ensures that the␣
↪padding bytes are zeroed. In some cases, this cleanup used to be performed by␣
↪always writing exactly 32 bytes, regardless of how many needed to be zeroed. This␣
↪was done with the assumption that the data that would eventually occupy the area␣
↪past the end of the array had not yet been written, because the encoder processes␣
↪tuple components in the order they were given. While this assumption is mostly true,
↪ there is an important corner case: dynamically encoded tuple components are stored␣
↪separately from the statically-sized ones in an area called the *tail* of the␣
↪encoding and the tail immediately follows the *head*, which is where the statically-
↪sized components are placed. The aforementioned cleanup, if performed for the last␣
↪component of the head would cross into the tail and overwrite up to 32 bytes of the␣
↪first component stored there with zeros. The only array type for which the cleanup␣
↪could actually result in an overwrite were arrays with ``uint256`` or ``bytes32``␣
↪as the base element type and in this case the size of the corrupted area was always␣
↪exactly 32 bytes. The problem affected tuples at any nesting level. This included␣
↪also structs, which are encoded as tuples in the ABI. Note also that lists of␣
↪parameters and return values of functions, events and errors are encoded as tuples.
↪",
        "link": "https://blog.soliditylang.org/2022/08/08/calldata-tuple-reencoding-
↪head-overflow-bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.16",
        "severity": "medium",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2022-5",
        "name": "DirtyBytesArrayToStorage",
        "summary": "Copying ``bytes`` arrays from memory or calldata to storage may␣
↪result in dirty storage values.",
        "description": "Copying ``bytes`` arrays from memory or calldata to storage␣
↪is done in chunks of 32 bytes even if the length is not a multiple of 32. Thereby,␣
↪extra bytes past the end of the array may be copied from calldata or memory to␣
↪storage. These dirty bytes may then become observable after a ``.push()`` without␣
↪arguments to the bytes array in storage, i.e. such a push will not result in a zero␣
↪value at the end of the array as expected. This bug only affects the legacy code␣
↪generation pipeline, the new code generation pipeline via IR is not affected.",
        "link": "https://blog.soliditylang.org/2022/06/15/dirty-bytes-array-to-
↪storage-bug/",
```


(续下页)

(接上页)

```
        "introduced": "0.0.1",
        "fixed": "0.8.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2022-4",
        "name": "InlineAssemblyMemorySideEffects",
        "summary": "The Yul optimizer may incorrectly remove memory writes from␣
↪inline assembly blocks, that do not access solidity variables.",
        "description": "The Yul optimizer considers all memory writes in the␣
↪outermost Yul block that are never read from as unused and removes them. This is␣
↪valid when that Yul block is the entire Yul program, which is always the case for␣
↪the Yul code generated by the new via-IR pipeline. Inline assembly blocks are never␣
↪optimized in isolation when using that pipeline. Instead they are optimized as a␣
↪part of the whole Yul input. However, the legacy code generation pipeline (which is␣
↪still the default) runs the Yul optimizer individually on an inline assembly block␣
↪if the block does not refer to any local variables defined in the surrounding␣
↪Solidity code. Consequently, memory writes in such inline assembly blocks are␣
↪removed as well, if the written memory is never read from in the same assembly␣
↪block, even if the written memory is accessed later, for example by a subsequent␣
↪inline assembly block.",
        "link": "https://blog.soliditylang.org/2022/06/15/inline-assembly-memory-side-
↪effects-bug/",
        "introduced": "0.8.13",
        "fixed": "0.8.15",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2022-3",
        "name": "DataLocationChangeInInternalOverride",
        "summary": "It was possible to change the data location of the parameters or␣
↪return variables from ``calldata`` to ``memory`` and vice-versa while overriding␣
↪internal and public functions. This caused invalid code to be generated when␣
↪calling such a function internally through virtual function calls.",
        "description": "When calling external functions, it is irrelevant if the data␣
↪location of the parameters is ``calldata`` or ``memory``, the encoding of the data␣
↪does not change. Because of that, changing the data location when overriding␣
↪external functions is allowed. The compiler incorrectly also allowed a change in␣
↪the data location for overriding public and internal functions. Since public␣
↪functions can be called internally as well as externally, this causes invalid code␣
```

(续下页)

(接上页)


```
↪to be generated when such an incorrectly overridden function is called internally␣
↪through the base contract. The caller provides a memory pointer, but the called␣
↪function interprets it as a calldata pointer or vice-versa.",
        "link": "https://blog.soliditylang.org/2022/05/17/data-location-inheritance-
↪bug/",
        "introduced": "0.6.9",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-2",
        "name": "NestedCalldataArrayAbiReencodingSizeValidation",
        "summary": "ABI-reencoding of nested dynamic calldata arrays did not always␣
↪perform proper size checks against the size of calldata and could read beyond␣
↪``calldatasize()``.",
        "description": "Calldata validation for nested dynamic types is deferred␣
↪until the first access to the nested values. Such an access may for example be a␣
↪copy to memory or an index or member access to the outer type. While in most such␣
↪accesses calldata validation correctly checks that the data area of the nested␣
↪array is completely contained in the passed calldata (i.e. in the range [0,␣
↪calldatasize()]), this check may not be performed, when ABI encoding such nested␣
↪types again directly from calldata. For instance, this can happen, if a value in␣
↪calldata with a nested dynamic array is passed to an external call, used in ``abi.
↪encode`` or emitted as event. In such cases, if the data area of the nested array␣
↪extends beyond ``calldatasize()``, ABI encoding it did not revert, but continued␣
↪reading values from beyond ``calldatasize()`` (i.e. zero values).",
        "link": "https://blog.soliditylang.org/2022/05/17/calldata-reencode-size-
↪check-bug/",
        "introduced": "0.5.8",
        "fixed": "0.8.14",
        "severity": "very low"
    },
    {
        "uid": "SOL-2022-1",
        "name": "AbiEncodeCallLiteralAsFixedBytesBug",
        "summary": "Literals used for a fixed length bytes parameter in ``abi.
↪encodeCall`` were encoded incorrectly.",
        "description": "For the encoding, the compiler only considered the types of␣
↪the expressions in the second argument of ``abi.encodeCall`` itself, but not the␣
↪parameter types of the function given as first argument. In almost all cases the␣
↪abi encoding of the type of the expression matches the abi encoding of the␣
↪parameter type of the given function. This is because the type checker ensures the␣
↪expression is implicitly convertible to the respective parameter type. However this␣
```


(续下页)

(接上页)


```
↪is not true for number literals used for fixed bytes types shorter than 32 bytes,␣
↪nor for string literals used for any fixed bytes type. Number literals were encoded␣
↪as numbers instead of being shifted to become left-aligned. String literals were␣
↪encoded as dynamically sized memory strings instead of being converted to a left-
↪aligned bytes value.",
        "link": "https://blog.soliditylang.org/2022/03/16/encodecall-bug/",
        "introduced": "0.8.11",
        "fixed": "0.8.13",
        "severity": "very low"

    },
    {
        "uid": "SOL-2021-4",
        "name": "UserDefinedValueTypesBug",
        "summary": "User defined value types with underlying type shorter than 32␣
↪bytes used incorrect storage layout and wasted storage",
        "description": "The compiler did not correctly compute the storage layout of␣
↪user defined value types based on types that are shorter than 32 bytes. It would␣
↪always use a full storage slot for these types, even if the underlying type was␣
↪shorter. This was wasteful and might have problems with tooling or contract␣
↪upgrades.",
        "link": "https://blog.soliditylang.org/2021/09/29/user-defined-value-types-
↪bug/",
        "introduced": "0.8.8",
        "fixed": "0.8.9",
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-3",
        "name": "SignedImmutables",
        "summary": "Immutable variables of signed integer type shorter than 256 bits␣
↪can lead to values with invalid higher order bits if inline assembly is used.",
        "description": "When immutable variables of signed integer type shorter than␣
↪256 bits are read, their higher order bits were unconditionally set to zero. The␣
↪correct operation would be to sign-extend the value, i.e. set the higher order bits␣
↪to one if the sign bit is one. This sign-extension is performed by Solidity just␣
↪prior to when it matters, i.e. when a value is stored in memory, when it is␣
↪compared or when a division is performed. Because of that, to our knowledge, the␣
↪only way to access the value in its unclean state is by reading it through inline␣
↪assembly.",
        "link": "https://blog.soliditylang.org/2021/09/29/signed-immutables-bug/",
        "introduced": "0.6.5",
        "fixed": "0.8.9",
```


(续下页)

(接上页)

```
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-2",
        "name": "ABIDecodeTwoDimensionalArrayMemory",
        "summary": "If used on memory byte arrays, result of the function ``abi.
↪decode`` can depend on the contents of memory outside of the actual byte array that␣
↪is decoded.",
        "description": "The ABI specification uses pointers to data areas for␣
↪everything that is dynamically-sized. When decoding data from memory (instead of␣
↪calldata), the ABI decoder did not properly validate some of these pointers. More␣
↪specifically, it was possible to use large values for the pointers inside arrays␣
↪such that computing the offset resulted in an undetected overflow. This could lead␣
↪to these pointers targeting areas in memory outside of the actual area to be␣
↪decoded. This way, it was possible for ``abi.decode`` to return different values␣
↪for the same encoded byte array.",
        "link": "https://blog.soliditylang.org/2021/04/21/decoding-from-memory-bug/",
        "introduced": "0.4.16",
        "fixed": "0.8.4",
        "conditions": {
            "ABIEncoderV2": true
        },
        "severity": "very low"
    },
    {
        "uid": "SOL-2021-1",
        "name": "KeccakCaching",
        "summary": "The bytecode optimizer incorrectly re-used previously evaluated␣
↪Keccak-256 hashes. You are unlikely to be affected if you do not compute Keccak-256␣
↪hashes in inline assembly.",
        "description": "Solidity's bytecode optimizer has a step that can compute␣
↪Keccak-256 hashes, if the contents of the memory are known during compilation time.␣
↪This step also has a mechanism to determine that two Keccak-256 hashes are equal␣
↪even if the values in memory are not known during compile time. This mechanism had␣
↪a bug where Keccak-256 of the same memory content, but different sizes were␣
↪considered equal. More specifically, ``keccak256(mpos1, length1)`` and␣
↪``keccak256(mpos2, length2)`` in some cases were considered equal if ``length1``␣
↪and ``length2``, when rounded up to nearest multiple of 32 were the same, and when␣
↪the memory contents at ``mpos1`` and ``mpos2`` can be deduced to be equal. You␣
↪maybe affected if you compute multiple Keccak-256 hashes of the same content, but␣
↪with different lengths inside inline assembly. You are unaffected if your code uses␣
↪``keccak256`` with a length that is not a compile-time constant or if it is always␣
↪a multiple of 32.",
```

(续下页)

(接上页)


```
        "link": "https://blog.soliditylang.org/2021/03/23/keccak-optimizer-bug/",
        "fixed": "0.8.3",
        "conditions": {
            "optimizer": true
        },
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-11",
        "name": "EmptyByteArrayCopy",
        "summary": "Copying an empty byte array (or string) from memory or calldata␣
↪to storage can result in data corruption if the target array's length is increased␣
↪subsequently without storing new data.",
        "description": "The routine that copies byte arrays from memory or calldata␣
↪to storage stores unrelated data from after the source array in the storage slot if␣
↪the source array is empty. If the storage array's length is subsequently increased␣
↪either by using ``.push()`` or by assigning to its ``.length`` attribute (only␣
↪before 0.6.0), the newly created byte array elements will not be zero-initialized,␣
↪but contain the unrelated data. You are not affected if you do not assign to ``.
↪length`` and do not use ``.push()`` on byte arrays, or only use ``.push(<arg>)`` or␣
↪manually initialize the new elements.",
        "link": "https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/",
        "fixed": "0.7.4",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-10",
        "name": "DynamicArrayCleanup",
        "summary": "When assigning a dynamically-sized array with types of size at␣
↪most 16 bytes in storage causing the assigned array to shrink, some parts of␣
↪deleted slots were not zeroed out.",
        "description": "Consider a dynamically-sized array in storage whose base-type␣
↪is small enough such that multiple values can be packed into a single slot, such as␣
↪`uint128[]`. Let us define its length to be `l`. When this array gets assigned from␣
↪another array with a smaller length, say `m`, the slots between elements `m` and␣
↪`l` have to be cleaned by zeroing them out. However, this cleaning was not␣
↪performed properly. Specifically, after the slot corresponding to `m`, only the␣
↪first packed value was cleaned up. If this array gets resized to a length larger␣
↪than `m`, the indices corresponding to the unclean parts of the slot contained the␣
↪original value, instead of 0. The resizing here is performed by assigning to the␣
↪array `length`, by a `push()` or via inline assembly. You are not affected if you␣
↪are only using `.push(<arg>)` or if you assign a value (even zero) to the new␣
↪elements after increasing the length of the array.",
```


(续下页)

(接上页)


```
        "link": "https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-
↪cleanup-bug/",
        "fixed": "0.7.3",
        "severity": "medium"
    },
    {
        "uid": "SOL-2020-9",
        "name": "FreeFunctionRedefinition",
        "summary": "The compiler does not flag an error when two or more free␣
↪functions with the same name and parameter types are defined in a source unit or␣
↪when an imported free function alias shadows another free function with a different␣
↪name but identical parameter types.",
        "description": "In contrast to functions defined inside contracts, free␣
↪functions with identical names and parameter types did not create an error. Both␣
↪definition of free functions with identical name and parameter types and an␣
↪imported free function with an alias that shadows another function with a different␣
↪name but identical parameter types were permitted due to which a call to either the␣
↪multiply defined free function or the imported free function alias within a␣
↪contract led to the execution of that free function which was defined first within␣
↪the source unit. Subsequently defined identical free function definitions were␣
↪silently ignored and their code generation was skipped.",
        "introduced": "0.7.1",
        "fixed": "0.7.2",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-8",
        "name": "UsingForCalldata",
        "summary": "Function calls to internal library functions with calldata␣
↪parameters called via ``using for`` can result in invalid data being read.",
        "description": "Function calls to internal library functions using the␣
↪``using for`` mechanism copied all calldata parameters to memory first and passed␣
↪them on like that, regardless of whether it was an internal or an external call.␣
↪Due to that, the called function would receive a memory pointer that is interpreted␣
↪as a calldata pointer. Since dynamically sized arrays are passed using two stack␣
↪slots for calldata, but only one for memory, this can lead to stack corruption. An␣
↪affected library call will consider the JUMPDEST to which it is supposed to return␣
↪as part of its arguments and will instead jump out to whatever was on the stack␣
↪before the call.",
        "introduced": "0.6.9",
        "fixed": "0.6.10",
        "severity": "very low"
    },
```


(续下页)

(接上页)

```
    {
        "uid": "SOL-2020-7",
        "name": "MissingEscapingInFormatting",
        "summary": "String literals containing double backslash characters passed␣
↪directly to external or encoding function calls can lead to a different string␣
↪being used when ABIEncoderV2 is enabled.",
        "description": "When ABIEncoderV2 is enabled, string literals passed directly␣
↪to encoding functions or external function calls are stored as strings in the␣
↪intemediate code. Characters outside the printable range are handled correctly, but␣
↪backslashes are not escaped in this procedure. This leads to double backslashes␣
↪being reduced to single backslashes and consequently re-interpreted as escapes␣
↪potentially resulting in a different string being encoded.",
        "introduced": "0.5.14",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-6",
        "name": "ArraySliceDynamicallyEncodedBaseType",
        "summary": "Accessing array slices of arrays with dynamically encoded base␣
↪types (e.g. multi-dimensional arrays) can result in invalid data being read.",
        "description": "For arrays with dynamically sized base types, index range␣
↪accesses that use a start expression that is non-zero will result in invalid array␣
↪slices. Any index access to such array slices will result in data being read from␣
↪incorrect calldata offsets. Array slices are only supported for dynamic calldata␣
↪types and all problematic type require ABIEncoderV2 to be enabled.",
        "introduced": "0.6.0",
        "fixed": "0.6.8",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2020-5",
        "name": "ImplicitConstructorCallvalueCheck",
        "summary": "The creation code of a contract that does not define a␣
↪constructor but has a base that does define a constructor did not revert for calls␣
↪with non-zero value.",
        "description": "Starting from Solidity 0.4.5 the creation code of contracts␣
```

(续下页)

(接上页)

```
↪without explicit payable constructor is supposed to contain a callvalue check that␣
↪results in contract creation reverting, if non-zero value is passed. However, this␣
↪check was missing in case no explicit constructor was defined in a contract at all,␣
↪but the contract has a base that does define a constructor. In these cases it is␣
↪possible to send value in a contract creation transaction or using inline assembly␣
↪without revert, even though the creation code is supposed to be non-payable.",
        "introduced": "0.4.5",
        "fixed": "0.6.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-4",
        "name": "TupleAssignmentMultiStackSlotComponents",
        "summary": "Tuple assignments with components that occupy several stack slots,
↪ i.e. nested tuples, pointers to external functions or references to dynamically␣
↪sized calldata arrays, can result in invalid values.",
        "description": "Tuple assignments did not correctly account for tuple␣
↪components that occupy multiple stack slots in case the number of stack slots␣
↪differs between left-hand-side and right-hand-side. This can either happen in the␣
↪presence of nested tuples or if the right-hand-side contains external function␣
↪pointers or references to dynamic calldata arrays, while the left-hand-side␣
↪contains an omission.",
        "introduced": "0.1.6",
        "fixed": "0.6.6",
        "severity": "very low"
    },
    {
        "uid": "SOL-2020-3",
        "name": "MemoryArrayCreationOverflow",
        "summary": "The creation of very large memory arrays can result in␣
↪overlapping memory regions and thus memory corruption.",
        "description": "No runtime overflow checks were performed for the length of␣
↪memory arrays during creation. In cases for which the memory size of an array in␣
↪bytes, i.e. the array length times 32, is larger than 2^256-1, the memory␣
↪allocation will overflow, potentially resulting in overlapping memory areas. The␣
↪length of the array is still stored correctly, so copying or iterating over such an␣
↪array will result in out-of-gas.",
        "link": "https://blog.soliditylang.org/2020/04/06/memory-creation-overflow-
↪bug/",
        "introduced": "0.2.0",
        "fixed": "0.6.5",
        "severity": "low"
    },
```

(续下页)

(接上页)


```
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and␣
↪continue and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared␣
↪inside a for loop if they can be removed in a control-flow branch that ends with␣
↪``break`` or ``continue`` even though they cannot be removed in other control-flow␣
↪branches. Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.6.0",
        "fixed": "0.6.1",
        "severity": "medium",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2020-2",
        "name": "privateCanBeOverridden",
        "summary": "Private methods can be overridden by inheriting contracts.",
        "description": "While private methods of base contracts are not visible and␣
↪cannot be called directly from the derived contract, it is still possible to␣
↪declare a function of the same name and type and thus change the behaviour of the␣
↪base contract's function.",
        "introduced": "0.3.0",
        "fixed": "0.5.17",
        "severity": "low"
    },
    {
        "uid": "SOL-2020-1",
        "name": "YulOptimizerRedundantAssignmentBreakContinue0.5",
        "summary": "The Yul optimizer can remove essential assignments to variables␣
↪declared inside for loops when Yul's continue or break statement is used. You are␣
↪unlikely to be affected if you do not use inline assembly with for loops and␣
↪continue and break statements.",
        "description": "The Yul optimizer has a stage that removes assignments to␣
↪variables that are overwritten again or are not used in all following control-flow␣
↪branches. This logic incorrectly removes such assignments to variables declared␣
↪inside a for loop if they can be removed in a control-flow branch that ends with␣
```


(续下页)

(接上页)

```
↪``break`` or ``continue`` even though they cannot be removed in other control-flow␣
↪branches. Variables declared outside of the respective for loop are not affected.",
        "introduced": "0.5.8",
        "fixed": "0.5.16",
        "severity": "low",
        "conditions": {
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-10",
        "name": "ABIEncoderV2LoopYulOptimizer",
        "summary": "If both the experimental ABIEncoderV2 and the experimental Yul␣
↪optimizer are activated, one component of the Yul optimizer may reuse data in␣
↪memory that has been changed in the meantime.",
        "description": "The Yul optimizer incorrectly replaces ``mload`` and␣
↪``sload`` calls with values that have been previously written to the load location␣
↪(and potentially changed in the meantime) if all of the following conditions are␣
↪met: (1) there is a matching ``mstore`` or ``sstore`` call before; (2) the contents␣
↪of memory or storage is only changed in a function that is called (directly or␣
↪indirectly) in between the first store and the load call; (3) called function␣
↪contains a for loop where the same memory location is changed in the condition or␣
↪the post or body block. When used in Solidity mode, this can only happen if the␣
↪experimental ABIEncoderV2 is activated and the experimental Yul optimizer has been␣
↪activated manually in addition to the regular optimizer in the compiler settings.",
        "introduced": "0.5.14",
        "fixed": "0.5.15",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true,
            "optimizer": true,
            "yulOptimizer": true
        }
    },
    {
        "uid": "SOL-2019-9",
        "name":
↪"ABIEncoderV2CalldataStructsWithStaticallySizedAndDynamicallyEncodedMembers",
        "summary": "Reading from calldata structs that contain dynamically encoded,␣
↪but statically-sized members can result in incorrect values.",
        "description": "When a calldata struct contains a dynamically encoded, but␣
↪statically-sized member, the offsets for all subsequent struct members are␣
↪calculated incorrectly. All reads from such members will result in invalid values.␣
```

(续下页)

(接上页)

```
↪Only calldata structs are affected, i.e. this occurs in external functions with␣
↪such structs as argument. Using affected structs in storage or memory or as␣
↪arguments to public functions on the other hand works correctly.",
        "introduced": "0.5.6",
        "fixed": "0.5.11",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-8",
        "name": "SignedArrayStorageCopy",
        "summary": "Assigning an array of signed integers to a storage array of␣
↪different type can lead to data corruption in that array.",
        "description": "In two's complement, negative integers have their higher␣
↪order bits set. In order to fit into a shared storage slot, these have to be set to␣
↪zero. When a conversion is done at the same time, the bits to set to zero were␣
↪incorrectly determined from the source and not the target type. This means that␣
↪such copy operations can lead to incorrect values being stored.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/
↪",
        "introduced": "0.4.7",
        "fixed": "0.5.10",
        "severity": "low/medium"
    },
    {
        "uid": "SOL-2019-7",
        "name": "ABIEncoderV2StorageArrayWithMultiSlotElement",
        "summary": "Storage arrays containing structs or other statically-sized␣
↪arrays are not read properly when directly encoded in external function calls or in␣
↪abi.encode*.",
        "description": "When storage arrays whose elements occupy more than a single␣
↪storage slot are directly encoded in external function calls or using abi.encode*,␣
↪their elements are read in an overlapping manner, i.e. the element pointer is not␣
↪properly advanced between reads. This is not a problem when the storage data is␣
↪first copied to a memory variable or if the storage array only contains value types␣
↪or dynamically-sized arrays.",
        "link": "https://blog.soliditylang.org/2019/06/25/solidity-storage-array-bugs/
↪",
        "introduced": "0.4.16",
        "fixed": "0.5.10",
        "severity": "low",
```

(续下页)

(接上页)


```
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-6",
        "name": "DynamicConstructorArgumentsClippedABIV2",
        "summary": "A contract's constructor that takes structs or arrays that␣
↪contain dynamically-sized arrays reverts or decodes to invalid data.",
        "description": "During construction of a contract, constructor parameters are␣
↪copied from the code section to memory for decoding. The amount of bytes to copy␣
↪was calculated incorrectly in case all parameters are statically-sized but contain␣
↪dynamically-sized arrays as struct members or inner arrays. Such types are only␣
↪available if ABIEncoderV2 is activated.",
        "introduced": "0.4.16",
        "fixed": "0.5.9",
        "severity": "very low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special␣
↪jump target only considered the situation after deployment.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-5",
        "name": "UninitializedFunctionPointerInConstructor_0.4.x",
        "summary": "Calling uninitialized internal function pointers created in the␣
↪constructor does not always revert and can cause unexpected behaviour.",
        "description": "Uninitialized internal function pointers point to a special␣
↪piece of code that causes a revert when called. Jump target positions are different␣
↪during construction and after deployment, but the code for setting this special␣
↪jump target only considered the situation after deployment.",
```


(续下页)

(接上页)


```
        "introduced": "0.4.5",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries",
        "summary": "Contract types used in events in libraries cause an incorrect␣
↪event signature hash",
        "description": "Instead of using the type `address` in the hashed signature,␣
↪the actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.5.0",
        "fixed": "0.5.8",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-4",
        "name": "IncorrectEventSignatureInLibraries_0.4.x",
        "summary": "Contract types used in events in libraries cause an incorrect␣
↪event signature hash",
        "description": "Instead of using the type `address` in the hashed signature,␣
↪the actual contract name was used, leading to a wrong hash in the logs.",
        "introduced": "0.3.0",
        "fixed": "0.4.26",
        "severity": "very low"
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but␣
↪can also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.0",
        "fixed": "0.5.7",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
```


(续下页)

(接上页)


```
        }
    },
    {
        "uid": "SOL-2019-3",
        "name": "ABIEncoderV2PackedStorage_0.4.x",
        "summary": "Storage structs and arrays with types shorter than 32 bytes can␣
↪cause data corruption if encoded directly from storage using the experimental␣
↪ABIEncoderV2.",
        "description": "Elements of structs and arrays that are shorter than 32 bytes␣
↪are not properly decoded from storage when encoded directly (i.e. not via a memory␣
↪type) using ABIEncoderV2. This can cause corruption in the values themselves but␣
↪can also overwrite other parts of the encoded data.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.4.19",
        "fixed": "0.4.26",
        "severity": "low",
        "conditions": {
            "ABIEncoderV2": true
        }
    },
    {
        "uid": "SOL-2019-2",
        "name": "IncorrectByteInstructionOptimization",
        "summary": "The optimizer incorrectly handles byte opcodes whose second␣
↪argument is 31 or a constant expression that evaluates to 31. This can result in␣
↪unexpected values.",
        "description": "The optimizer incorrectly handles byte opcodes that use the␣
↪constant 31 as second argument. This can happen when performing index access on␣
↪bytesNN types with a compile-time constant value (not index) of 31 or when using␣
↪the byte opcode in inline assembly.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.7",
        "severity": "very low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2019-1",
        "name": "DoubleShiftSizeOverflow",
```


(续下页)

(接上页)


```
        "summary": "Double bitwise shifts by large constants whose sum overflows 256␣
↪bits can result in unexpected values.",
        "description": "Nested logical shift operations whose total shift size is␣
↪2**256 or more are incorrectly optimized. This only applies to shifts by numbers of␣
↪bits that are compile-time constant expressions.",
        "link": "https://blog.soliditylang.org/2019/03/26/solidity-optimizer-and-
↪abiencoderv2-bug/",
        "introduced": "0.5.5",
        "fixed": "0.5.6",
        "severity": "low",
        "conditions": {
            "optimizer": true,
            "evmVersion": ">=constantinople"
        }
    },
    {
        "uid": "SOL-2018-4",
        "name": "ExpExponentCleanup",
        "summary": "Using the ** operator with an exponent of type shorter than 256␣
↪bits can result in unexpected values.",
        "description": "Higher order bits in the exponent are not properly cleaned␣
↪before the EXP opcode is applied if the type of the exponent expression is smaller␣
↪than 256 bits and not smaller than the type of the base. In that case, the result␣
↪might be larger than expected if the exponent is assumed to lie within the value␣
↪range of the type. Literal numbers as exponents are unaffected as are exponents or␣
↪bases of type uint256.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "fixed": "0.4.25",
        "severity": "medium/high",
        "check": {"regex-source": "[^/]\\*\\* *[^/0-9 ]"}
    },
    {
        "uid": "SOL-2018-3",
        "name": "EventStructWrongData",
        "summary": "Using structs in events logged wrong data.",
        "description": "If a struct is used in an event, the address of the struct is␣
↪logged instead of the actual data.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.4.17",
        "fixed": "0.4.25",
        "severity": "very low",
        "check": {"ast-compact-json-path": "$..[?(@.nodeType === 'EventDefinition')]..
↪[?(@.nodeType === 'UserDefinedTypeName' && @.typeDescriptions.typeString.startsWith(
```


(续下页)

(接上页)

```
↪'struct'))]"}
    },
    {
        "uid": "SOL-2018-2",
        "name": "NestedArrayFunctionCallDecoder",
        "summary": "Calling functions that return multi-dimensional fixed-size arrays␣
↪can result in memory corruption.",
        "description": "If Solidity code calls a function that returns a multi-
↪dimensional fixed-size array, array elements are incorrectly interpreted as memory␣
↪pointers and thus can cause memory corruption if the return values are accessed.␣
↪Calling functions with multi-dimensional fixed-size arrays is unaffected as is␣
↪returning fixed-size arrays from function calls. The regular expression only checks␣
↪if such functions are present, not if they are called, which is required for the␣
↪contract to be affected.",
        "link": "https://blog.soliditylang.org/2018/09/13/solidity-bugfix-release/",
        "introduced": "0.1.4",
        "fixed": "0.4.22",
        "severity": "medium",
        "check": {"regex-source": "returns[^;{]*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\
↪]\\s*\\[\\s*[^\\] \\t\\r\\n\\v\\f][^\\]]*\\][^{;]*[;{]"}
    },
    {
        "uid": "SOL-2018-1",
        "name": "OneOfTwoConstructorsSkipped",
        "summary": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored.",
        "description": "If a contract has both a new-style constructor (using the␣
↪constructor keyword) and an old-style constructor (a function with the same name as␣
↪the contract) at the same time, one of them will be ignored. There will be a␣
↪compiler warning about the old-style constructor, so contracts only using new-style␣
↪constructors are fine.",
        "introduced": "0.4.22",
        "fixed": "0.4.23",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-5",
        "name": "ZeroFunctionSelector",
        "summary": "It is possible to craft the name of a function such that it is␣
↪executed instead of the fallback function in very specific circumstances.",
        "description": "If a function has a selector consisting only of zeros, is␣
↪payable and part of a contract that does not have a fallback function and at most␣
```

(续下页)

(接上页)


```
↪five external functions in total, this function is called instead of the fallback␣
↪function if Ether is sent to the contract without data.",
        "fixed": "0.4.18",
        "severity": "very low"
    },
    {
        "uid": "SOL-2017-4",
        "name": "DelegateCallReturnValue",
        "summary": "The low-level .delegatecall() does not return the execution␣
↪outcome, but converts the value returned by the functioned called to a boolean␣
↪instead.",
        "description": "The return value of the low-level .delegatecall() function is␣
↪taken from a position in memory, where the call data or the return data resides.␣
↪This value is interpreted as a boolean and put onto the stack. This means if the␣
↪called function returns at least 32 zero bytes, .delegatecall() returns false even␣
↪if the call was successful.",
        "introduced": "0.3.0",
        "fixed": "0.4.15",
        "severity": "low"
    },
    {
        "uid": "SOL-2017-3",
        "name": "ECRecoverMalformedInput",
        "summary": "The ecrecover() builtin can return garbage for malformed input.",
        "description": "The ecrecover precompile does not properly signal failure for␣
↪malformed input (especially in the 'v' argument) and thus the Solidity function can␣
↪return data that was previously present in the return area in memory.",
        "fixed": "0.4.14",
        "severity": "medium"
    },
    {
        "uid": "SOL-2017-2",
        "name": "SkipEmptyStringLiteral",
        "summary": "If \"\" is used in a function call, the following function␣
↪arguments will not be correctly passed to the function.",
        "description": "If the empty string literal \"\" is used as an argument in a␣
↪function call, it is skipped by the encoder. This has the effect that the encoding␣
↪of all arguments following this is shifted left by 32 bytes and thus the function␣
↪call data is corrupted.",
        "fixed": "0.4.12",
        "severity": "low"
    },
    {
```


(续下页)

(接上页)

```
        "uid": "SOL-2017-1",
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the␣
↪code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode␣
↪by routines that compute them with less gas. For some special numbers, an incorrect␣
↪routine is generated. This could allow an attacker to e.g. trick victims about a␣
↪specific amount of ether, or function calls to call different functions (or none at␣
↪all).",
        "link": "https://blog.soliditylang.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-11",
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying␣
↪memory, ignored its return value. On the public chain, calls to the identity␣
↪precompile can be made in a way that they never fail, but this might be different␣
↪on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "uid": "SOL-2016-10",
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was simplified to just␣
↪use the empty state, but this implementation was not done properly. This bug can␣
↪cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
```

(续下页)

(接上页)


```
    },
    {
        "uid": "SOL-2016-9",
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly␣
↪and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same␣
↪32 byte storage slot, but storage writes always write 32 bytes. For some types, the␣
↪higher order bytes were not cleaned properly, which made it sometimes possible to␣
↪overwrite a variable in storage when writing to another one.",
        "link": "https://blog.soliditylang.org/2016/11/01/security-alert-solidity-
↪variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "uid": "SOL-2016-8",
        "name": "OptimizerStaleKnowledgeAboutSHA3",
        "summary": "The optimizer did not properly reset its knowledge about SHA3␣
↪operations resulting in some hashes (also used for storage variable positions) not␣
↪being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-
↪evaluating expressions whose value is already known. This knowledge was not␣
↪properly reset across control flow paths and thus the optimizer sometimes thought␣
↪that the result of a SHA3 operation is already present on the stack. This could␣
↪result in data corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "uid": "SOL-2016-7",
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that␣
↪received Ether.",
        "description": "Library functions are protected against sending them Ether␣
↪through a call. Since the DELEGATECALL opcode forwards the information about how␣
↪much Ether was sent with a call, the library function incorrectly assumed that␣
↪Ether was sent to the library and threw an exception.",
        "severity": "low",
```


(续下页)

(接上页)


```
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "uid": "SOL-2016-6",
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if␣
↪no Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a␣
↪certain amount of gas from the EVM to handle the transfer. In the case of a zero-
↪transfer, this gas is not provided which causes the recipient to throw an exception.
↪",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "uid": "SOL-2016-5",
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite␣
↪loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this␣
↪length is zero, code was generated that did not terminate and thus consumed all gas.
↪",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "uid": "SOL-2016-4",
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump␣
↪destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages.␣
↪At jump destinations, multiple code paths join and thus it has to compute a common␣
↪state from the incoming edges. Computing this common state was not done correctly.␣
↪This bug can cause data corruption, but it is probably quite hard to use for␣
↪targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
```


(续下页)

(接上页)


```
        "uid": "SOL-2016-3",
        "name": "CleanBytesHigherOrderBits",
        "summary": "The higher order bits of short bytesNN types were not cleaned␣
↪before comparison.",
        "description": "Two variables of type bytesNN were considered different if␣
↪their higher order bits, which are not part of the actual value, were different. An␣
↪attacker might use this to reach seemingly unreachable code paths by providing␣
↪incorrectly formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "uid": "SOL-2016-2",
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32␣
↪bytes did not correctly clean the higher order bits, causing corruption in other␣
↪array elements.",
        "description": "Multiple elements of an array of values that are shorter than␣
↪17 bytes are packed into the same storage slot. Writing to a single element of such␣
↪an array did not properly clean the higher order bytes and thus could lead to data␣
↪corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "uid": "SOL-2016-1",
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several␣
↪undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting␣
↪from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```

3.26 Solidity v0.5.0 突破性变化

本节强调了 Solidity 0.5.0 版本中引入的主要突破性变化，以及这些变化背后的原因和如何更新受影响的代码。对于完整的列表，请查看 版本更新日志。

备注： 用 Solidity v0.5.0 编译的合约仍然可以与合约甚至用旧版本编译的库对接，而无需重新编译或重新部署。将接口更改为包含数据位置，可见性和可变性说明符就足够了。参见下面的与旧合约的互操作性 部分。

3.26.1 仅有语义上的变化

本节仅列出了语义的变化，因此有可能在现有代码中隐藏新的且不同的行为。

- 有符号的右移现在使用正确的算术移位，即向负无穷大取整，而不是向零取整。有符号和无符号移位在君士坦丁堡（Constantinople）版本将有专门的操作码，目前由 Solidity 模拟。
- 在 `do...while` 循环中的 `continue` 语句现在跳转到条件，这是在这种情况下的常见行为。以前是跳到循环主体。因此，如果条件是假的，循环就终止了。
- 函数 `.call()`，`.delegatecall()` 和 `.staticcall()` 在给定一个 `bytes` 参数时，不再进行填充。
- 如果 EVM 的版本是拜占庭（Byzantium）或更高版本，现在调用 pure 和 view 函数时使用操作码 `STATICCALL` 而不是 `CALL`。这不允许在 EVM 层面上改变状态。
- 当在外部函数调用和 `abi.encode` 中使用时，ABI 编码器现在可以正确地对来自 calldata（`msg.data` 和外部函数参数）的字节数组和字符串进行填充。对于未填充的编码，请使用 `abi.encodePacked`。
- 如果传入的 calldata 太短或指向界外，ABI 解码器会在函数的开头和 `abi.decode()` 中回退。注意，脏的高阶位仍然会被忽略。
- 从蜜桔前哨（Tangerine Whistle）开始，用外部功能调用转发所有可用气体。

3.26.2 语义和语法的变化

本节重点介绍影响语法和语义的变化。

- 函数 `.call()`，`.delegatecall()`，`staticcall()`，`keccak256()`，`sha256()` 和 `ripemd160()` 现在只接受一个 `bytes` 参数。此外，该参数没有被填充。这样做是为了使参数的连接方式更加明确和清晰。将每个 `.call()` (和家族) 改为 `.call("")`，将每个 `.call(signature, a,b, c)` 改为 `.call(abi.encodeWithSignature(signature, a, b, c))` （最后一项只对值类型有效）。将每个 `keccak256(a, b, c)` 改为 `keccak256(abi.encodePacked(a, b, c))`。尽管这不是一个突破性的改变，建议开发者将 `x.call(bytes4(keccak256("f(uint256)")), a, b)` 改为 `x.call(abi.encodeWithSignature("f(uint256)", a, b))`。

- 函数 `.call()`，`.delegatecall()` 和 `.staticcall()` 现在返回 `(bool, bytes memory)` 以提供对返回数据的访问。将 `bool success = otherContract.call("f")` 改为 `(bool success, bytes memory data) = otherContract.call("f")`。
- Solidity 现在为函数局部变量实现了 C99 风格的范围规则，也就是说，变量只能在它们被声明后使用，并且只能在相同或嵌套的范围内使用。在 `for` 循环的初始化块中声明的变量在循环内部的任何一点都是有效的。

3.26.3 明确性要求

本节列出了现在的代码需要更加明确的变化。对于大多数的主题，编译器会提供建议。

- 明确的函数可见性现在是强制性的。在每个函数和构造函数中添加 `public`，在每个未指定可见性的回退或接口函数中添加 `external`。
- 所有结构，数组或映射类型的变量的明确数据位置现在是强制性的。这也适用于函数参数和返回变量。例如，将 `uint[] x = z` 改为 `uint[] storage x = z`，将 `function f(uint[] [] x)` 改为 `function f(uint[] [] memory x)`，其中 `memory` 是数据位置，可以相应地替换为 `storage` 或 `calldata`。注意，`external` 函数要求参数的数据位置为 `calldata`。
- 合约类型不再包括 `address` 成员，以便分离命名空间。因此，现在有必要在使用 `address` 成员之前，明确地将合约类型的值转换为地址。例如：如果 `c` 是一个合约，把 `c.transfer(...)` 改为 `address(c).transfer(...)`，把 `c.balance` 改为 `address(c).balance`。
- 现在不允许在不相关的合约类型之间进行显式的转换。您只能从一个合约类型转换到它的一个基础或祖先类型。如果您确定一个合约与您想转换的合约类型是兼容的，尽管它没有继承它，您可以通过先转换为 `address` 来解决这个问题。例如：如果 `A` 和 `B` 是合约类型，`B` 不继承 `A`，而 `b` 是 `B` 类型的合约，您仍然可以用 `A(address(b))` 将 `b` 转换成 `A` 类型。请注意，您仍然需要注意匹配的 payable 修饰的回退函数，如下文所述。
- `address` 类型被分成 `address` 和 `address payable`，其中只有 `address payable` 提供 `transfer` 功能。一个 `address payable` 可以直接转换为 `address`，但不允许以其他方式转换。将 `address` 转换为 `address payable` 是可以通过 `uint160` 转换的。如果 `c` 是一个合约，只有当 `c` 有一个 payable 修饰的回退函数时，`address(c)` 的结果是 `address payable`。如果您使用取回模式，您很可能不必改变您的代码，因为 `transfer` 只用于 `msg.sender` 而不是存储地址，而且 `msg.sender` 是一个 `address payable` 类型。
- 现在不允许不同位数的 `bytesX` 和 `uintY` 之间的转换了，因为 `bytesX` 会在右侧填充，`uintY` 会在左侧填充，这可能导致意外的转换结果。现在在转换前必须在类型内调整位数。例如，您想要将 `bytes4`（4 字节）转换为 `uint64`（8 字节），首先将 `bytes4` 变量转换为 `bytes8`，然后再转换为 `uint64`。当通过 `uint32` 转换时，您会得到相反的填充结果。在 v0.5.0 之前，任何 `bytesX` 和 `uintY` 之间的转换都要通过 `uint8X`。例如，`uint8(bytes3(0x291807))` 将被转换为 `uint8(uint24(bytes3(0x291807)))`（结果是 `0x07`）。
- 在非 payable 函数中使用 `msg.value`（或通过修改器引入）是不允许的，因为这是一个安全特性。将该函数变成 `payable`，或为程序逻辑创建一个新的内部函数，使用 `msg.value`。

- 为了清晰起见，如果使用标准输入作为源，命令行界面现在需要 `-`。

3.26.4 废弃的元素

这一节列出了废弃以前的功能或语法的变化。请注意，其中许多变化已经在实验模式 `v0.5.0` 中启用。

命令行和 JSON 接口

- 命令行选项 `--formal` （用于生成 Why3 输出以进一步形式化验证）已被废弃，现在已被删除。一个新的形式化验证模块，SMTChecker，可以通过 `pragma experimental SMTChecker;` 启用。
- 由于中间语言 `Julia` 更名为 `Yul`，命令行选项 `--julia` 被更名为 `--yul`。
- 删除了 `--clone-bin` 和 `--combined-json clone-bin` 命令行选项。
- 不允许使用空前缀的重映射。
- JSON AST 字段 `constant` 和 `payable` 被删除。这些信息现在出现在 `stateMutability` 字段中。
- `FunctionDefinition` 节点的 JSON AST 字段 `isConstructor` 被一个名为 `kind` 的字段取代，该字段的值可以是 `"constructor"`，`"fallback"` 或 `"function"`。
- 在非链接的二进制十六进制文件中，库地址占位符现在是完全等同的库名的 keccak256 哈希值的前 36 个十六进制字符，用 `$...$` 包围。以前，只使用完全等同的库名。这减少了碰撞的机会，特别是在使用长路径的时候。二进制文件现在也包含一个从这些占位符到完全等同名称的映射列表。

构造函数

- 现在必须使用 `constructor` 关键字来定义构造函数。
- 现在不允许在没有括号的情况下调用基本构造函数。
- 现在不允许在同一继承层次中多次指定基本构造函数参数。
- 现在不允许调用有参数但参数个数错误的构造函数。如果您只是想指定一个继承关系而不是给参数，完全不要提供括号。

函数

- 函数 `callcode` 现在被禁止使用（改用 `delegatecall`）。但仍然可以通过内联汇编使用它。
- 现在不允许使用 `suicide` （改用 `selfdestruct`）。
- 现在不允许使用 `sha3` （改用 `keccak256`）。
- 现在不允许使用 `throw` （改用 `revert`，`require` 和 `assert`）。

转换

- 现在不允许从数字到 bytesXX 类型的显性和隐性转换。
- 现在不允许从十六进制字数到不同大小的 bytesXX 类型的显性和隐性转换。

字面常量和后缀

- 由于闰年的复杂性和混乱性，现在不允许使用单位名称 years。
- 现在不允许出现后面没有数字的尾部圆点。
- 现在不允许将十六进制数字与单位值相结合（例如：0x1e wei）。
- 十六进制数字的前缀 0X 是不允许的，只能是 0x。

变量

- 为了清晰起见，现在不允许声明空结构。
- 现在不允许使用 var 关键字，以利于明确性。
- 现在不允许在具有不同组件数量的元组之间进行分配。
- 不允许使用不属于编译时常量的常量值。
- 现在不允许出现数值不匹配的多变量声明。
- 现在不允许出现未初始化的存储变量。
- 现在不允许使用空元组。
- 检测变量和结构中的循环依赖关系，在递归中被限制为 256 个。
- 现在不允许长度为零的固定长度数组。

语法

- 现在不允许使用 constant 作为函数状态的可变性修饰符。
- 布尔表达式不能使用算术运算。
- 现在不允许使用单数的 + 操作符。
- 如果没有事先转换为明确的类型，字面量不能再使用 abi.encodePacked。
- 现在不允许有一个或多个返回值的函数的空返回语句。
- 现在完全不允许使用" 松散汇编" 语法，也就是说，不能再使用跳转标签，跳转和非功能指令。使用新的 while，switch 和 if 结构代替。
- 没有实现的函数不能再使用修改器。

- 现在不允许具有命名返回值的函数类型。
- 现在不允许在不是程序块的 if/while/for 语句体中进行单语句变量声明。
- 新的关键字：calldata 和 constructor。
- 新的保留关键字：alias, apply, auto, copyof, define, immutable, implements, macro, mutable, override, partial, promise, reference, sealed, sizeof, supports, typedef 和 unchecked。

3.26.5 与旧合约的互操作性

通过为它们定义接口，仍然可以与为 0.5.0 之前的 Solidity 版本编写的合于对接（或者反过来）。考虑到您已经部署了以下 0.5.0 之前的合约：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// 在0.4.25版本的编译器之前，这将报告一个警告
// 这在0.5.0之后将无法编译。
contract OldContract {
    function someOldFunction(uint8 a) {
        //...
    }
    function anotherOldFunction() constant returns (bool) {
        //...
    }
    // ...
}
```

这将不再在 Solidity 0.5.0 版本中进行编译。然而，您可以为它定义一个兼容的接口：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}
```

请注意，我们没有声明 anotherOldFunction 是 view，尽管它在原始合约中被声明为 constant。这是由于从 Solidity 0.5.0 版本开始，staticcall 被用来调用 view 函数。在 0.5.0 版本之前，constant 关键字没有被强制执行，所以用 staticcall 调用一个被声明为 constant 的函数仍然可能被还原，因为 constant 函数仍然可能试图修改存储。因此，当为旧合约定义接口时，您应该只使用 view 来代替 constant，以防您绝对确定该函数能与 staticcall 一起工作。

有了上面定义的接口，您现在可以很容易地使用已经部署的 0.5.0 之前的合约：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

interface OldContract {
    function someOldFunction(uint8 a) external;
    function anotherOldFunction() external returns (bool);
}

contract NewContract {
    function doSomething(OldContract a) public returns (bool) {
        a.someOldFunction(0x42);
        return a.anotherOldFunction();
    }
}
```

同样，0.5.0 以前的库可以通过定义库的功能而不需要实现，并在连接时提供 0.5.0 以前的库的地址来使用（关于如何使用命令行编译器进行连接，请参见使用命令行编译器）。

```
// 这在0.6.0版本之后将无法编译。
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;

library OldLibrary {
    function someFunction(uint8 a) public returns(bool);
}

contract NewContract {
    function f(uint8 a) public returns (bool) {
        return OldLibrary.someFunction(a);
    }
}
```

3.26.6 示例

下面的例子显示了 Solidity 0.5.0 版本的合约及其更新版本，其中包括本节中列出的一些变化。

Old version:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.25;
// 这在0.5.0版本之后将无法编译。

contract OtherContract {
    uint x;
```

(续下页)

(接上页)

```
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract Old {
    OtherContract other;
    uint myNumber;

    // 没有提供函数的可变性，不是错误。
    function someInteger() internal returns (uint) { return 2; }

    // 没有提供函数的可见性，不是错误。
    // 没有提供函数的可变性，不是错误。
    function f(uint x) returns (bytes) {
        // 在这个版本中，var是可以使用的。
        var z = someInteger();
        x += z;
        // 在这个版本中，throw是可以使用的。
        if (x > 100)
            throw;
        bytes memory b = new bytes(x);
        y = -3 >> 1;
        // y == -1（错，应该是-2）。
        do {
            x += 1;
            if (x > 10) continue;
            // 'Continue' 会导致无限循环。
        } while (x < 11);
        // 调用只返回一个布尔值。
        bool success = address(other).call("f");
        if (!success)
            revert();
        else {
            // 局部变量可以在其使用后声明。
            int y;
        }
        return b;
    }

    //不需要为'arr'设置明确的数据位置
    function g(uint[] arr, bytes8 x, OtherContract otherContract) public {
```

(续下页)

(接上页)

```
        otherContract.transfer(1 ether);

        // 由于uint32 (4个字节) 小于byte8 (8个字节) ,
        // x的前4个字节将被丢失。
        // 这可能会导致意想不到的行为，因为bytesX是向右填充的。
        uint32 y = uint32(x);
        myNumber += y + msg.value;
    }
}
```

新版本:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.5.0;
// 这在0.6.0版本之后将无法编译。

contract OtherContract {
    uint x;
    function f(uint y) external {
        x = y;
    }
    function() payable external {}
}

contract New {
    OtherContract other;
    uint myNumber;

    // 必须指定函数的可变性。
    function someInteger() internal pure returns (uint) { return 2; }

    // 必须指定函数的可见性。
    // 必须指定函数的可变性。
    function f(uint x) public returns (bytes memory) {
        // 现在必须明确地给出类型。
        uint z = someInteger();
        x += z;
        // 现在不允许使用throw。
        require(x <= 100);
        int y = -3 >> 1;
        require(y == -2);
        do {
            x += 1;
            if (x > 10) continue;
```

(续下页)

(接上页)

```
            // 'Continue'跳转到下面的条件。
        } while (x < 11);

        // call返回值为(bool, bytes).
        // 必须指定数据位置。
        (bool success, bytes memory data) = address(other).call("f");
        if (!success)
            revert();
        return data;
    }

    using AddressMakePayable for address;
    // 必须指定'arr'的数据位置
    function g(uint[] memory /* arr */, bytes8 x, OtherContract otherContract,␣
↪address unknownContract) public payable {
        // 没有提供'otherContract.transfer'。
        // 由于'OtherContract'的代码是已知的，并且具有回退功能，
        // address(otherContract)具有'address payable'类型。
        address(otherContract).transfer(1 ether);

        // 没有提供'unknownContract.transfer'。
        // 没有提供'address(unknownContract).transfer'
        // 因为'address(unknownContract)'不是'address payable'类型。
        // 如果该函数需要一个您想发送资金的'address'类型，
        // 您可以通过'uint160'将其转换为'address payable'类型。
        // 注意：不建议这样做，应尽可能使用明确的'address payable'类型。
        //␣
↪为了提高明确性，我们建议使用一个库来进行转换（在这个例子中的合同后面提供）。
        address payable addr = unknownContract.makePayable();
        require(addr.send(1 ether));

        // 由于uint32（4字节）小于bytes8（8字节），
        // 所以不允许进行转换。
        // 我们需要先转换到一个通用的大小：
        bytes4 x4 = bytes4(x); // Padding happens on the right
        uint32 y = uint32(x4); // Conversion is consistent
        // 'msg.value'不能用在'非payable'类型的函数中。
        // 我们需要把函数变成payable类型
        myNumber += y + msg.value;
    }
}

// 我们可以定义一个库，将 ``address`` 类型明确转换为 ``address payable``␣
```

(续下页)

(接上页)

```
↪类型，作为一种变通方法。
library AddressMakePayable {
    function makePayable(address x) internal pure returns (address payable) {
        return address(uint160(x));
    }
}
```

3.27 Solidity 0.6.0 版本突破性变化

本节强调了 Solidity 0.6.0 版本中引入的主要突破性变化，以及这些变化背后的原因和如何更新受影响的代码。对于完整的列表，请查看 版本更新日志。

3.27.1 编译器可能不会发出警告的变化

本节列出了一些变化，在这些变化中，您的代码的行为可能会发生变化，而编译器不会告诉您。

- 指数运算的结果类型是基数的类型。就像对称运算一样，它曾经是可以同时容纳基数类型和指数类型的最小类型。此外，指数化的基数允许是有符号的类型。

3.27.2 显性要求

这一节列出了代码现在需要更显式的更改，但是语义没有改变。对于大多数主题，编译器会提供建议。

- 现在，只有当函数被标记为 `virtual` 关键字或被定义在一个接口中时，才能被重载。在接口之外没有实现的函数必须被标记为 `virtual`。当重载一个函数或修改器时，必须使用新的关键字 `override`。当重载一个定义在多个并行基类的函数或修改器时，所有基必须在关键字后面的括号中列出，像这样：`override(Base1, Base2)`。
- 成员对数组的 `length` 的访问现在总是只读的，即使是存储数组。不再可能通过给存储数组的长度分配一个新值来调整其大小。使用 `push()`，`push(value)` 或 `pop()` 代替，或者分配一个完整的数组，当然这将覆盖现有内容。这背后的原因是为了防止巨大的存储阵列的存储碰撞。
- 新的关键字 `abstract` 可以用来标记合约为抽象的。如果一个合约没有实现它的所有功能，就必须使用这个关键字。抽象合约不能用 `new` 操作符创建，在编译过程中也不可能为其生成字节码。
- 库合约必须实现其所有功能，而不仅仅是内部功能。
- 在内联汇编中声明的变量名称不能再以 `_slot` 或 `_offset` 结尾。
- 内联汇编中的变量声明不能再影射内联汇编块外的任何声明。如果变量名称中包含一个点，那么它的前缀到点的部分不能与内联汇编块外的任何声明冲突。
- 在内联汇编中，不带参数的操作码现在表示为"内置函数"而不是独立标识符。所以 `gas` 现在是 `gas()`。

- 现在不允许影子状态变量。一个派生合约只能声明一个状态变量 `x`，如果在它的任何基类合约中都没有同名的可见状态变量。

3.27.3 语义和语法变化

这一部分列出了您必须修改您的代码，而之后它又做了一些别的事情的变化。

- 现在不允许从外部函数类型转换为 `address` 类型。相反，外部函数类型有一个叫做 `address` 的成员，类似于现有的 `selector` 成员。
- 动态存储数组的函数 `push(value)` 不再返回新的长度（它什么也不返回）。
- 通常被称为" 回退函数" 的无名函数被拆分为一个新的回退函数，该函数使用 `fallback` 关键字定义，并使用 `receive` 关键字定义一个接收以太的函数。
  - 如果存在，每当调用数据为空时（无论是否收到以太），都会调用接收以太函数 receive。此函数是隐式的 `payable`。
  - 当没有其他函数匹配时，就会调用新的回退函数（如果接收以太的函数 receive 不存在，则包括调用数据为空的调用）。您可以让这个函数是 `payable` 函数或不是。如果它不是 `payable` 函数，那么不匹配任何其他发送价值的函数的交易将恢复。只有在采用升级或代理模式时，才需要实现新的回退函数。

3.27.4 新功能

本节列出了在 Solidity 0.6.0 之前不可能实现或很难实现的事情。

- *try/catch 语句* 允许您对失败的外部调用做出反应。
- `struct` 和 `enum` 类型可以在文件级别声明。
- 数组切片可以用于 calldata 数组，例如 `abi.decode(msg.data[4:], (uint, uint))` 是对函数调用有效负载进行解码的低级方法。
- Natspec 在开发者文档中支持多个返回参数，执行与 `@param` 相同的命名检查。
- Yul 和内联汇编有一个新的语句，叫做 `leave`，可以退出当前函数。
- 现在可以通过 `payable(x)` 将 `address` 转换为 `address payable`，其中 `x` 必须是 `address` 类型。

3.27.5 接口变化

本节列出与语言本身无关但对编译器接口有影响的更改。这些可能会改变您在命令行上使用编译器的方式，例如，您如何使用它的可编程接口，或者您如何分析它产生的输出。

新的错误报告器

引入一个新的错误报告器，其目的是在命令行上产生更易访问的错误消息。它在默认情况下是启用的，但是通过 `--old-reporter` 可以返回到弃用的旧错误报告器。

元数据哈希选项

编译器现在默认将元数据文件的 IPFS 哈希值附加到字节码的末尾（详见合约元数据 ）文档。在 0.6.0 之前，编译器默认附加了 Swarm 哈希值，为了仍然支持这种行为，引入了新的命令行选项 `--metadata-hash`。它允许您通过传递 `--metadata-hash` 命令行选项的 `ipfs` 或 `swarm` 值来选择要产生和附加的哈希。传递 `none` 则可以完全删除哈希。

这些变化也可以通过标准 *JSON* 接口 使用，并影响编译器生成的元数据 JSON。

读取元数据的推荐方法是读取最后两个字节，以确定 CBOR 编码的长度，并对该数据块进行适当的解码，这在元数据部分 中有所解释。

Yul 优化器

与传统的字节码优化器一起，*Yul* 优化器现在在用 `--optimize` 参数调用编译器时默认启用。可以通过 `--no-optimize-yul` 参数在调用编译器时禁用它。这主要影响到使用 ABI coder v2 的代码。

C API 变化

使用 `libsolc` 的 C API 的客户端代码现在可以控制编译器使用的内存。为了使这一变化保持一致，`solidity_free` 被重新命名为 `solidity_reset`，增加了函数 `solidity_alloc` 和 `solidity_free`，`solidity_compile` 现在返回一个必须通过 `solidity_free()` 显式释放的字符串。

3.27.6 如何更新您的代码

本节详细说明了如何为每一个突破性变化更新先前的代码。

- 将 `address(f)` 改为 `f.address`，因为 `f` 是外部函数类型。
- 用 `receive() external payable { ... }`，`fallback() external [payable] { ... }` 或这两个函数一起来替换 `function () external [payable] { ... }`。只要有可能，最好只使用 `receive` 函数。

- 将 `uint length = array.push(value)` 改为 `array.push(value);`。新的长度可以通过 `array.length` 访问。
- 将 `array.length++` 改为 `array.push()` 来增加数组长度，使用 `pop()` 来减少存储数组的长度。
- 在一个函数的 `@dev` 文档中，为每个命名的返回参数定义一个 `@return` 条目，将参数的名称作为第一个词。例如，如果您有定义为 `function f() public returns (uint value)` 的函数 `f()`，并且有 `@dev` 注释，那么记录它的返回参数如下：`@return value The return value.`。您可以混合使用命名的和未命名的返回参数文档，只要这些声明是按照它们在元组返回类型中出现的顺序即可。
- 为内联汇编中的变量声明选择唯一的标识符，不与内联汇编块外的声明冲突。
- 在每一个您打算重载的非接口函数上添加 `virtual`。在所有没有具体实现的接口之外的函数上添加 `virtual`。对于单继承，在每个重载的函数上添加 `override`。对于多重继承，添加 `override(A, B, ..)`，在括号中列出所有定义了重载函数的合约。当多个基类定义同一个函数时，继承的合约必须重载所有冲突的函数。
- 在内联汇编中，将 `()` 添加到所有不接受参数的操作码后。例如，将 `pc` 更改为 `pc()`，将 `gas` 更改为 `gas()`。

3.28 Solidity v0.7.0 突破性变化

本节强调了 Solidity 0.7.0 版本中引入的主要突破性变化，以及这些变化背后的原因和如何更新受影响的代码。对于完整的列表，请查看 版本更新日志。

3.28.1 语义的微小变化

- 用非字符量进行指数和移位（例如：`1 << x` 或 `2 ** x`）将总是使用 `uint256` 类型（对于非负数）或 `int256` 类型（对于负数）来执行操作。在此之前，该操作是在移位量/指数的类型中进行的，这可能会产生误导。

3.28.2 语法的变化

- 在外部函数和合约创建调用中，现在使用新的语法指定以太和 gas：`x.f{gas: 10000, value: 2 ether}(arg1, arg2)`。旧的语法 -- `x.f.gas(10000).value(2 ether)(arg1, arg2)` -- 会导致错误。
- 全局变量 `now` 已被弃用，应该使用 `block.timestamp` 来替换。对于一个全局变量来说，单一的标识符 `now` 太通用了，可能会让人觉得它在事务处理过程中发生变化，而 `block.timestamp` 正确地反映了它只是块的一个属性。
- 对变量的 NatSpec 注释只允许用于公共状态变量，而不允许用于本地或内部变量。
- 代号 `gwei` 现在是一个关键词（用于指定，例如 `2 gwei` 作为一个数字），不能作为一个标识符使用。

- 字符串现在只能包含可打印的 ASCII 字符，这也包括各种转义序列，如十六进制（`\xff`）和 unicode 转义（`\u20ac`）。
- 现在支持 Unicode 字符串文本来容纳有效的 UTF-8 序列。它们用 `unicode` 前缀来标识：`unicode"Hello 😃"`。
- 状态可变性：现在可以在继承过程中限制函数的状态可变性。具有默认状态可变性的函数可以被 `pure` 和 `view` 函数所覆盖，而 `view` 函数可以被 `pure` 函数所覆盖。同时，公共状态变量被认为是 `view`，甚至是 `pure`，如果它们是常量。

内联汇编

- 在用户定义的函数和变量名称中，不允许在内联汇编中使用 `.`。如果您在 "仅 Yul" 模式下使用 Solidity，它仍然有效。
- 存储指针变量 `x` 的槽和偏移量通过 `x.slot` 和 `x.offset` 访问，而不是 `x_slot` 和 `x_offset`。

3.28.3 移除未使用或不安全的功能

存储之外的映射关系

- 如果一个结构或数组包含一个映射，它只能在存储中使用。以前，映射成员在内存中被默默地跳过，这让人困惑，也容易出错。
- 如果存储中的结构或数组包含映射，则对其进行赋值是不可行的。以前，在复制操作过程中，映射会被默默地跳过，这是一种误导，而且容易出错。

函数和事件

- 构造函数不再需要可见性（`public`/`internal`）了。为了防止合约被创建，可以将其标记为 `abstract`。这使得构造函数的可见性概念变得过时了。
- 类型检查器：不允许库函数为 `virtual`：由于库合约不能被继承，库函数不应该被标记为 virtual。
- 不允许在同一继承层次中具有相同名称和参数类型的多个事件。
- `using A for B` 只影响到它所提到的合约。以前，这种影响是继承的。现在，您必须在所有使用该特性的派生合约中重复 `using` 语句。

表达式

- 有符号类型的移位是不允许的。以前，允许负数的移位，但它在运行时会被还原。
- `finney` 和 `szabo` 的面额被删除。它们很少被使用，并且不能使实际的金额清晰可见。相反，可以使用明确的数值，如 `1e20` 或非常常见的 `gwei`。

声明

- 关键字 `var` 不能再使用了。以前，这个关键词可以解析，但会导致一个类型错误，并建议使用哪种类型。现在，它导致一个解析器错误。

3.28.4 接口变化

- JSON AST：用 `kind: "hexString"` 来标记十六进制字符串文本。
- JSON AST：值为 `null` 的成员将从 JSON 输出中删除。
- NatSpec：构造器和函数有一致的用户文档输出。

3.28.5 如何更新您的代码

本节详细说明了如何为每一个突破性变化更新先前的代码。

- 将 `x.f.value(...)()` 改为 `x.f{value: ...}()`。类似地，`(new C).value(...)()` 改为 `new C{value: ...}()`，`x.f.gas(...).value(...)()` 改为 `x.f{gas: ..., value: ...}()`。
- 将 `now` 改为 `block.timestamp`.
- 将移位运算符中的右操作数的类型改为无符号类型。例如，将 `x >> (256 - y)` 改为 `x >> uint(256 - y)`。
- 如果需要，在所有派生合约中重复 `using A for B` 的语句。
- 从每个构造函数中删除 `public` 关键字。
- 从每个构造函数中删除 `internal` 关键字，并在合约中添加 `abstract` （如果还没有存在）。
- 将内联汇编中的 `_slot` 和 `_offset` 后缀分别改为 `.slot` 和 `.offset`。

3.29 Solidity v0.8.0 突破性变化

本节强调了 Solidity 0.8.0 版本中引入的主要突破性变化。对于完整的列表，请查看 版本更新日志。

3.29.1 语义的微小变化

本节列出了现有代码在编译器没有通知您的情况下改变其行为的更改。

- 算术操作在下溢和溢出时都会恢复。您可以使用 `unchecked { ... }` 来使用以前的包装行为。

  溢出的检查是非常普遍的，所以我们把它作为默认的检查，以增加代码的可读性，即使它是以略微增加 gas 成本为代价的。

- ABI 编码器 v2 默认是激活的。

  您可以使用 `pragma abicoder v1;` 来选择使用旧的行为。语句 `pragma experimental ABIEncoderV2;` 仍然有效，但它已被废弃，没有效果。如果您想显示使用，请使用 `pragma abicoder v2;` 代替。

  请注意，ABI coder v2 比 v1 支持更多的类型，并对输入进行更多的合理性检查。ABI coder v2 使一些函数调用更加昂贵，而且当合约中包含不符合参数类型的数据时，它还会使合约调用回退，而在 ABI coder v1 中则没有回退。

- 指数是右联的，也就是说，表达式 `a**b**c` 被解析为 `a**(b**c)`。在 0.8.0 之前，它被解析为 `(a**b)**c`。

  这是解析指数运算符的常用方法。

- 失败的断言和其他内部检查，如除以零或算术溢出，不使用无效的操作码，而是使用恢复操作码。更具体地说，它们将使用等于对 `Panic(uint256)` 的函数调用的错误数据，其错误代码是针对具体情况的。

  这将节省错误的 gas，同时它仍然允许静态分析工具将这些情况与无效输入的恢复区分开来，比如一个失败的 `require`。

- 如果访问存储中的一个字节数组，其长度被错误地编码，就会引起 panic 错误。合约不会出现这种情况，除非使用内联汇编来修改存储字节数组的原始表示。

- 如果常数被用于数组长度表达式中，Solidity 的先前版本将在评估树的所有分支中使用任意精度。现在，如果常量变量被用作中间表达式，它们的值将以与它们在运行时表达式中使用时相同的方式被正确舍入。

- 类型 `byte` 已经被删除。它是 `bytes1` 的别名。

3.29.2 新的限制条件

本节列出了可能导致现有合约不再编译的变化。

- 有一些与字面常量的显式转换有关的新限制。以前在以下情况下的行为可能是模糊的：
    1. 不允许从负数字段和大于 `type(uint160).max` 的字段显式转换为 `address`。
    2. 只有当字面常量位于 `type(T).min` 和 `type(T).max` 之间时，才允许字面常量与整数类型 `T` 之间的明确转换。特别的是，用 `type(uint).max` 代替 `uint(-1)` 的使用。
    3. 只有当字面常量能够代表枚举中的一个值时，才允许字面常量和枚举之间的显式转换。
    4. 字面常量和 `address` 类型之间的显式转换（例如，`address(literal)`）是 `address` 类型，而不是 `address payable` 类型。通过使用显式转换，即 `payable(literal)`，可以得到一个 payable 类型的地址类型。
- 地址字面常量 的类型是 `address`，而不是 `address payable`。它们可以通过显式的转换转换为 `address payable` 类型，例如：`payable(0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF)`。
- 对显式类型转换有新的限制。只有当符号，宽度或类型类别（`int`，`address`，`bytesNN` 等）有最多一次变化时，才允许进行转换。要执行多个变化，请使用多个转换。

  让我们使用符号 `T(S)` 来表示显式转换 `T(x)`，其中，`T` 和 `S` 是类型，`x` 是 `S` 类型的任何任意变量。这种不允许的转换的例子是 `uint16(int8)`，因为它同时改变了宽度（8 位到 16 位）和符号（有符号整数到无符号整数）。为了进行转换，我们必须通过一个中间类型。在前面的例子中，这将是 `uint16(uint8(int8))` 或者 `uint16(int16(int8))`。请注意，这两种转换方式将产生不同的结果，例如，对于 `-1`。下面是这个规则不允许的一些转换的例子。

  - `address(uint)` 和 `uint(address)`：同时转换类型和宽度。分别用 `address(uint160(uint))` 和 `uint(uint160(address))` 代替。
  - `payable(uint160)`，`payable(bytes20)` 和 `payable(integer-literal)`：同时转换了类型和状态可变性。分别用 `payable(address(uint160))`，`payable(address(bytes20))` 和 `payable(address(integer-literal))` 代替。请注意，`payable(0)` 是有效的，是规则的例外。
  - `int80(bytes10)` 和 `bytes10(int80)`：同时转换了类型和符号。分别用 `int80(uint80(bytes10))` 和 `bytes10(uint80(int80))` 代替。
  - `Contract(uint)`：同时转换类型和宽度。用 `Contract(address(uint160(uint)))` 代替。

  这些转换是不允许的，以避免歧义。例如，在表达式 `uint16 x = uint16(int8(-1))` 中，`x` 的值取决于是先应用符号还是宽度转换。
- 函数调用选项只能给出一次，即 `c.f{gas: 10000}{value: 1}()` 是无效的，必须改成 `c.f{gas: 10000, value: 1}()`。
- 全局函数 `log0`，`log1`，`log2`，`log3` 和 `log4` 已被删除。

这些都是低级别的函数，基本上没有被使用过。它们的行为可以通过内联汇编访问。

- `enum` 定义包含的成员不能超过 256 个。

  这将使我们可以安全地假设 ABI 中的底层类型总是 `uint8`。

- 除了公共函数和事件之外，不允许使用 `this`，`super` 和 `_` 的名称进行声明。这个例外是为了使声明用 Solidity 以外的语言实现的合约的接口成为可能，这些语言确实允许这种函数名称。

- 移除对代码中的 `\b`，`\f` 和 `\v` 转义序列的支持。它们仍然可以通过十六进制转义插入，例如：分别是 `\x08`，`\x0c`，和 `\x0b`。

- 全局变量 `tx.origin` 和 `msg.sender` 的类型是 `address` 而不是 `address payable`。我们可以通过显式转换将它们转换为 `address payable` 类型，即 `payable(tx.origin)` 或 `payable(msg.sender)`。

  做这个改变是因为编译器不能确定这些地址是否可以支付，所以现在需要一个明确的转换来使这个要求可见。

- 显式转换为 `address` 类型总是返回一个非-payable 类型的 `address`。特别是，以下显式转换的类型是 `address` 而不是 `address payable`：

  - `address(u)` 其中 `u` 是一个 `uint160` 类型的变量。我们可以通过两个显式转换将 `u` 转换为 `address payable` 类型，即 `payable(address(u))`。
  - `address(b)` 其中 `b` 是一个 `bytes20` 类型的变量。我们可以通过两个显式转换将 `b` 转换为 `address payable` 类型，即 `payable(address(b))`。
  - `address(c)` 其中 `c` 是一个合约。以前，这种转换的返回类型取决于合约是否可以接收以太（要么有一个 receive 函数，要么有一个 payable 类型的 fallback 函数）。转换 `payable(c)` 的类型为 `address payable`，只有当合约 `c` 可以接收以太时才允许。一般来说，人们总是可以通过使用以下显式转换将 `c` 转换为 `address payable` 的类型：`payable(address(c))`。请注意，`address(this)` 与 `address(c)` 属于同一类别，同样的规则也适用于它。

- 内联汇编中的 `chainid` 现在被认为是 `view` 而不是 `pure`。

- 一元求反不能再用于无符号整数，只能用于有符号整数。

3.29.3 接口变化

- `--combined-json` 的输出已经改变。JSON 字段 `abi`，`devdoc`，`userdoc` 和 `storage-layout` 现在是子对象。在 0.8.0 之前，它们曾被序列化为字符串。

- “传统 AST“已被删除（`--ast-json` 在命令行界面，`legacyAST` 用于标准 JSON）。使用“紧凑型 AST”（`--ast-compact-json` 参数. `AST`）作为替代。

- 旧的错误报告器（`--old-reporter` ）已经被删除。

3.29.4 如何更新您的代码

- 如果您依赖包装算术，请用 `unchecked { ... }` 包裹每个操作。
- 可选：如果您使用 SafeMath 或类似的库，将 `x.add(y)` 改为 `x + y`，`x.mul(y)` 改为 `x * y` 等等。
- 如果您想继续使用旧的 ABI 编码器，请添加 `pragma abicoder v1;`。
- 可以选择删除 `pragma experimental ABIEncoderV2` 或 `pragma abicoder v2` 因为它是多余的。
- 将 `byte` 改为 `bytes1`。
- 如果需要的话，添加中间显式类型转换。
- 将 `c.f{gas: 10000}{value: 1}()` 合并为 `c.f{gas: 10000, value: 1}()`。
- 将 `msg.sender.transfer(x)` 改为 `payable(msg.sender).transfer(x)` 或者使用 `address payable` 类型的存储变量。
- 将 `x**y**z` 改为 `(x**y)**z`。
- Use inline assembly as a replacement for `log0`, ..., `log4`.
- 将无符号整数取反的方法是从该类型的最大值中减去该整数，并加上 1（例如，`type(uint256).max - x + 1`，同时确保 *x* 不为零）

3.30 风格指南

Solidity 合约可以使用一种特殊形式的注释来为函数，返回变量等提供丰富的文档。这种特殊形式被命名为 Ethereum 自然语言规范格式（NatSpec）。

备注： NatSpec 是受 Doxygen 的启发。虽然它使用 Doxygen 风格的注释和标签，但并不打算与 Doxygen 保持严格的兼容性。请仔细检查下面列出的支持的标签。

该文件被划分为以开发人员为中心的信息和面向最终用户的信息。这些信息可以在终端用户（人类）与合约交互（即签署交易）时显示给他们。

建议 Solidity 合约使用 NatSpec 对所有公共接口（ABI 中的一切）进行完全地注释。

NatSpec 包括智能合约作者将使用的注释的格式，这些注释可被 Solidity 编译器理解。下面还详细介绍了 Solidity 编译器的输出，它将这些注释提取为机器可读的格式。

NatSpec 也可以包括第三方工具使用的注释。这些最可能是通过 `@custom:<name>` 标签完成的，一个好的用例是分析和验证工具就是如此。

3.30.1 文档示例

文档可以通过使用 Doxygen 符号格式来嵌入到每个 `contract`，`interface`，`library`，`function` 和 `event` 之上。在 NatSpec 中，`public` 状态变量等同于 `function`。

- 对于 Solidity，您可以选择 `///` 用于单行注释或以 `/**` 开始，并以 `*/` 结束的符号用于多行注释
- 对于 Vyper 来说，使用 `""""` 缩进到内部内容来裸注释（译者注：无标记符号注释）。参见 Vyper 文档。

下面的例子显示了一个合约和一个使用所有可用标记的函数。

备注： Solidity 编译器只在标签是外部或公共的情况下才进行解析。但也欢迎您为您的内部和私有函数使用类似的注释，不过这些不会被解析。

这在未来可能会发生变化。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.2 < 0.9.0;

/// @title 树的模拟器
/// @author Larry A. Gardner
/// @notice 您只能将此合约用于最基本的模拟。
/// @dev 目前所有的函数调用都是在没有副作用的情况下实现的
/// @custom:experimental 这是一个实验性的合约。
contract Tree {
    /// @notice 计算活体树木的树龄，按四舍五入计算
    /// @dev Alexandr N. Tetearing 算法可以提高精确度
    /// @param rings 树龄学样本的环数
    /// @return 树龄（岁），部分年份四舍五入
    function age(uint256 rings) external virtual pure returns (uint256) {
        return rings + 1;
    }

    /// @notice 返回该树的叶子数量。
    /// @dev 在此只是返回了一个固定的数字。
    function leaves() external virtual pure returns(uint256) {
        return 2;
    }
}

contract Plant {
    function leaves() external virtual pure returns(uint256) {
        return 3;
    }
}
```

(续下页)

(接上页)

```
contract KumquatTree is Tree, Plant {
    function age(uint256 rings) external override pure returns (uint256) {
        return rings + 2;
    }

    /// 返回这种特定类型的树的叶子数量。
    /// @inheritdoc Tree 合约
    function leaves() external override(Tree, Plant) pure returns(uint256) {
        return 3;
    }
}
```

3.30.2 标签

所有标签都是可选的。下表解释了每个 NatSpec 标签的目的和它可能被使用的地方。有一种特殊情况，如果没有使用标签，那么 Solidity 编译器将以同样的方式进行 `///` 或 `/**` 注释，如同它被标记为 `@notice`。

标签		应用于
`@title`	一个应该描述合约/接口的标题	contract, library, interface
`@author`	作者的名字	contract, library, interface
`@notice`	向终端用户解释这个东西的作用	contract, library, interface, function, public state variable, event
`@dev`	向开发人员解释任何额外的细节	contract, library, interface, function, state variable, event
`@param`	就像在 Doxygen 中一样记录一个参数（必须在参数名之后）	function, event
`@return`	记录一个合约的函数的返回变量	function, public state variable
`@inheritdoc`	从基本函数中复制所有缺失的标签（必须在合约名称之后）	function, public state variable
`@custom:...`	自定义标签，语义由应用程序定义	everywhere

如果您的函数返回多个值，如 `(int quotient, int remainder)` 那么使用多个 `@return` 语句，格式与 `@param` 语句相同。

自定义标签以 `@custom:` 开头，后面必须有一个或多个小写字母或连字符。然而，它不能以连字符开始。它们可以在任何地方使用，是开发者文档的一部分。

动态表达方式

Solidity 编译器将通过 NatSpec 文档从您的 Solidity 源代码传递到本指南所述的 JSON 输出。此 JSON 输出的使用者，例如最终用户的客户端软件，可以直接将其呈现给最终用户，或者它可以应用一些预处理。

例如，一些客户端软件会呈现为：

```
/// @notice 这个函数将使 `a` 乘以7
```

对终端用户来说，是：

```
这个函数将10乘以7
```

如果一个函数被调用，并且输入的 `a` 被赋值为 10。

继承说明

没有 NatSpec 的函数将自动继承其基函数的文档。这方面的例外情况是：

- 当参数名称不同时。
- 当有不止一个的基础函数时。
- 当有一个明确的 `@inheritdoc` 标签，指定了应该使用哪个合约来继承。

3.30.3 文件输出

当被编译器解析时，像上面例子中的文档将产生两个不同的 JSON 文件。一个是为了让终端用户在执行函数时作为通知使用，另一个是为了让开发人员使用。

如果上述合约被保存为 `ex1.sol`，那么您可以用以下方法生成文档：

```
solc --userdoc --devdoc ex1.sol
```

输出如下。

备注： 从 Solidity 0.6.11 版开始，NatSpec 输出也包含一个 `version`（版本号）和一个 `kind`（种类）字段。目前，`version` 被设置为 `1`，`kind` 必须是 `user`（用户）或 `dev`（开发者）之一。在未来，有可能会引入新的版本，淘汰旧的版本。

用户文档

上述文档将产生以下用户文档 JSON 文件作为输出：

```
{
  "version" : 1,
  "kind" : "user",
  "methods" :
  {
    "age(uint256)" :
    {
      "notice" : "计算活体树木的树龄，按四舍五入计算"
    }
  },
  "notice" : "您只能将此合约用于最基本的模拟。"
}
```

请注意，找到方法的关键是合约 *ABI* 中定义的函数的标准签名，而不是简单的函数名称。

开发者文档

除了用户文档文件，还应该产生一个开发者文档的 JSON 文件，看起来应该是这样的：

```
{
  "version" : 1,
  "kind" : "dev",
  "author" : "Larry A. Gardner",
  "details" : "目前所有的函数调用都是在没有副作用的情况下实现的",
  "custom:experimental" : "这是一个实验性的合约。",
  "methods" :
  {
    "age(uint256)" :
    {
      "details" : "Alexandr N. Tetearing 算法可以提高精确度",
      "params" :
      {
        "rings" : "树龄学样本的环数"
      },
      "return" : "树龄（岁），部分年份四舍五入"
    }
  },
  "title" : "树的模拟器"
}
```

3.31 SMT 检查器和形式化验证

使用形式化验证，有可能进行自动数学证明，证明您的源代码符合某种形式化规范。该规范仍然是正式的（就像源代码一样），但通常要简单得多。

请注意，形式化验证本身只能帮助您理解您所做的（规范）和您如何做的（实际实现）之间的区别。您仍然需要检查规范是否是您想要的，以及您没有遗漏任何意想不到的效果。

Solidity 实现了基于 SMT（可满足性模型理论（Satisfiability Modulo Theories） 和 Horn 解决的形式验证方法。SMT 检查器模块自动尝试证明代码满足由 `require` 和 `assert` 语句给出的规范。也就是说，它把 `require` 语句视为假设，并试图证明 `assert` 语句中的条件总是真的。如果发现断言失败，则可以向用户提供一个反例，说明断言是如何被违反的。如果 SMT 检查器对某一属性没有给出警告，这意味着该属性是安全的。

SMT 检查器在编译时检查的其他验证目标有：

- 算术上的下溢和溢出。
- 除以 0 的除法。
- 无用的条件和无法访问的代码。
- 弹出一个空数组。
- 超出界限的索引访问。
- 转账资金不足。

如果所有检查引擎都被启用，上述所有目标都被默认为自动检查，除了 Solidity >=0.8.7 的下溢和溢出。

SMT 检查器所报告的潜在警告是：

- `< 失败的属性> 发生在这里。`这意味着 SMT 检查器证明了某一属性失败。可能会给出一个反例，但是在复杂的情况下，也可能不会显示反例。在某些情况下，当 SMT 编码为 Solidity 代码添加了难以表达或无法表达的抽象时，这个结果也可能是一个假阳性。
- `< 失败的属性> 可能发生在这里。`这意味着求解器无法在给定的超时时间内证明两种情况。由于结果是未知的，SMT 检查器会报告潜在的健全性失败。这可以通过增加查询超时时间来解决，但问题也可能只是对引擎来说太难解决。

要启用 SMT 检查器，您必须选择应该运行哪一个引擎，其中默认的是没有引擎。选择引擎可以在所有文件上启用 SMT 检查器。

备注： 在 Solidity 0.8.4 之前，启用 SMT 检查器的默认方式是通过 `pragma experimental SMTChecker;` 并且只有包含 pragma 的合约才会被分析。该 pragma 已被弃用，尽管它仍能使 SMT 检查器向后兼容，但它将在 Solidity 0.9.0 中被移除。还要注意的是，现在即使在一个文件中使用 pragma，也会对所有文件启用 SMT 检查器。

备注： 假设 SMT 检查器和底层求解器中没有错误，那么验证目标没有警告就代表了一个无可争议的正确性

数学证明。请记住，这些问题在一般情况下是 很难的，有时是 不可能自动解决的。因此，有几个属性可能无法解决，或者可能导致大型合约的假阳性。每一个被证明的属性都应该被看作是一个重要的成就。对于高级用户，请参阅*SMT 检查器调优* 来了解一些可能有助于证明更复杂属性的选项。

3.31.1 教程

溢出

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        return add(x, y);
    }
}
```

上面的合约显示了一个溢出检查的例子。对于 Solidity >=0.8.7，SMT 检查器默认不检查下溢和溢出，所以我们需要使用命令行选项 `--model-checker-targets "underflow,overflow"` 或者 JSON 选项 `settings.modelChecker.targets = ["underflow", "overflow"]`。参见本节的目标配置。此处，它报告如下：

```
Warning: CHC: Overflow (resulting value larger than 2**256 - 1) happens here.
Counterexample:
x = 1, y =␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935
 = 0

Transaction trace:
Overflow.constructor(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935)
```

(续下页)

(接上页)

```
State: x = 1, y =␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935
Overflow.stateAdd()
    Overflow.add(1,␣
↪115792089237316195423570985008687907853269984665640564039457584007913129639935) --␣
↪internal call
 --> o.sol:9:20:
  |
9 |             return x_ + y_;
  |                    ^^^^^^^
```

如果我们添加了过滤掉溢出情况的 require 语句，SMT 检查器就会证明没有溢出是可以达到的（会通过不报告警告表现出来）。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Overflow {
    uint immutable x;
    uint immutable y;

    function add(uint x_, uint y_) internal pure returns (uint) {
        return x_ + y_;
    }

    constructor(uint x_, uint y_) {
        (x, y) = (x_, y_);
    }

    function stateAdd() public view returns (uint) {
        require(x < type(uint128).max);
        require(y < type(uint128).max);
        return add(x, y);
    }
}
```

断言

断言表示代码中的一个不变量：对于所有的事务，包括所有的输入和存储值，一个属性必须为真，否则就会出现错误。

下面的代码定义了一个保证没有溢出的函数 `f`。函数 `inv` 定义了 `f` 是单调递增的规范：对于每个可能的数值对 `(a, b)`，如果 `b > a`，那么 `f(b) > f(a)`。由于 `f` 确实是单调增长的，SMT 检查器证明了我们的属性是正确的。我们鼓励您试试这个属性和函数定义，看看会有什么样的结果！

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Monotonic {
    function f(uint x) internal pure returns (uint) {
        require(x < type(uint128).max);
        return x * 42;
    }

    function inv(uint a, uint b) public pure {
        require(b > a);
        assert(f(b) > f(a));
    }
}
```

我们还可以在循环中添加断言，以验证更多的复杂的属性。下面的代码搜索一个不受限制的数字数组的最大元素，并断言找到的元素必须大于或等于数组中的每个元素的属性。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m >= a[i]);

        return m;
    }
}
```

注意，在这个例子中，SMT 检查器将自动尝试证明三个属性：

1. 第一个循环中的 ++i 不会溢出。
2. 第二个循环中的 ++i 不会溢出。
3. 该断言始终是正确的。

备注： 这些属性涉及到循环，这使得它比前面的例子 *更加* 难了，所以要当心循环的问题！

所有的属性都被正确证明是安全的。可以随意改变属性和/或在数组上添加限制，以看到不同的结果。例如，将代码改为

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Max {
    function max(uint[] memory a) public pure returns (uint) {
        require(a.length >= 5);
        uint m = 0;
        for (uint i = 0; i < a.length; ++i)
            if (a[i] > m)
                m = a[i];

        for (uint i = 0; i < a.length; ++i)
            assert(m > a[i]);

        return m;
    }
}
```

我们得到的结果：

```
Warning: CHC: Assertion violation happens here.
Counterexample:

a = [0, 0, 0, 0, 0]
 = 0

Transaction trace:
Test.constructor()
Test.max([0, 0, 0, 0, 0])
  --> max.sol:14:4:
   |
14 |            assert(m > a[i]);
```

状态属性

到目前为止，这些例子只展示了 SMT 检查器在纯代码上的使用，证明了关于特定操作或算法的属性。智能合约中常见的属性类型是涉及合约状态的属性。对于这样的属性，可能需要多个交易来使断言失效。

举一个例子，考虑一个二维网格，其中两个轴的坐标都在（-2^128, 2^128 - 1）范围内。让我们在位置（0，0）放置一个机器人。该机器人只能在对角线上移动，一次只能走一步，不能在网格外移动。机器人的状态机可以用下面的智能合约来表示。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Robot {
    int x = 0;
    int y = 0;

    modifier wall {
        require(x > type(int128).min && x < type(int128).max);
        require(y > type(int128).min && y < type(int128).max);
        _;
    }

    function moveLeftUp() wall public {
        --x;
        ++y;
    }

    function moveLeftDown() wall public {
        --x;
        --y;
    }

    function moveRightUp() wall public {
        ++x;
        ++y;
    }

    function moveRightDown() wall public {
        ++x;
        --y;
    }

    function inv() public view {
        assert((x + y) % 2 == 0);
    }
```

(续下页)

(接上页)

```
}
```

函数 `inv` 代表状态机的一个不变量，即 `x + y` 必须是偶数。SMT 检查器设法证明，无论我们给机器人多少条命令，即使是无限多的命令，这个不变量都 不会失败。有兴趣的读者可能也想手动证明这个事实。提示：这个不变量是归纳性的。

我们也可以欺骗 SMT 检查器，让它给我们提供一条通往某个我们认为可能是可访问的位置的路径。我们可以通过添加以下函数，来增加 (2, 4) 是 不可访问的属性。

```
function reach_2_4() public view {
    assert(!(x == 2 && y == 4));
}
```

这个属性是假的，在证明这个属性是假的同时，SMT 检查器准确地告诉我们 *如何*访问到 (2, 4)。

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 2, y = 4

Transaction trace:
Robot.constructor()
State: x = 0, y = 0
Robot.moveLeftUp()
State: x = (- 1), y = 1
Robot.moveRightUp()
State: x = 0, y = 2
Robot.moveRightUp()
State: x = 1, y = 3
Robot.moveRightUp()
State: x = 2, y = 4
Robot.reach_2_4()
  --> r.sol:35:4:
   |
35 |            assert(!(x == 2 && y == 4));
   |            ^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

请注意，上面的路径不一定是确定的，因为还有其他路径可以访问（2，4)。选择哪条路径可能会根据所使用的解算器，其使用版本，或者只是随机地改变。

外部调用和重入

每个外部调用都被 SMT 检查器视为对未知代码的调用。这背后的原因是，即使被调用合约的代码在编译时是可用的，也不能保证部署的合约确实与编译时接口所在的合约相同。

在某些情况下，有可能在状态变量上自动推断出属性，即使外部调用的代码可以做任何事情，包括重新进入调用者合约，这些属性仍然是真的。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

interface Unknown {
    function run() external;
}

contract Mutex {
    uint x;
    bool lock;

    Unknown immutable unknown;

    constructor(Unknown u) {
        require(address(u) != address(0));
        unknown = u;
    }

    modifier mutex {
        require(!lock);
        lock = true;
        _;
        lock = false;
    }

    function set(uint x_) mutex public {
        x = x_;
    }

    function run() mutex public {
        uint xPre = x;
        unknown.run();
        assert(xPre == x);
    }
}
```

上面的例子显示了一个使用互斥标志来禁止重入的合约。解算器能够推断出，当 `unknown.run()` 被调用时，合约已经被“锁定”，所以无论未知的调用代码做什么，都不可能改变 `x` 的值。

如果我们“忘记”在函数 `set` 上使用 `mutex` 修饰符，SMT 检查器就能合成外部调用代码的行为，从而使断言失败。

```
Warning: CHC: Assertion violation happens here.
Counterexample:
x = 1, lock = true, unknown = 1

Transaction trace:
Mutex.constructor(1)
State: x = 0, lock = false, unknown = 1
Mutex.run()
    unknown.run() -- untrusted external call, synthesized as:
        Mutex.set(1) -- reentrant call
  --> m.sol:32:3:
   |
32 |                assert(xPre == x);
   |                ^^^^^^^^^^^^^^^^^
```

3.31.2 SMT 检查器选项和调试

超时

SMT 检查器使用了一个硬编码的资源限制（`rlimit`），这个限制是根据每个求解器选择的，与时间没有确切的关系。我们选择 `rlimit` 选项作为默认值，因为它比求解器内部的时间提供了更多的确定性保证。

这个选项大致转化为每个查询“几秒钟超时”。当然，许多属性非常复杂，需要大量的时间来解决，而决定并不重要。如果 SMT 检查器不能用默认的 `rlimit` 选项处理合约属性，则可以通过命令行界面（CLI）选项 `--model-checker-timeout <time>` 或 JSON 选项 `settings.modelChecker.timeout=<time>` 给出以毫秒为单位的超时。其中 0 表示不超时。

验证目标

SMT 检查器创建的验证目标的类型也可以通过命令行界面选项 `--model-checker-target <targets>` 或 JSON 选项 `settings.modelChecker.targets=<targets>` 来定制。在命令行界面情况下，`<targets>` 是一个没有空格的逗号分隔的一个或多个验证目标的列表，在 JSON 输入中是一个或多个作为字符串的目标数组。代表目标的关键词是：

- 断言：`assert`。
- 算术下溢：`underflow`。
- 算术溢出：`overflow`。
- 除以零：`divByZero`。
- 无用的条件和无法访问的代码：`constantCondition`。

- 弹出一个空数组：`popEmptyArray`。
- 越界的数组/固定字节索引访问：`outOfBounds`。
- 转账资金不足：`balance`。
- 以上都是：`default` （仅适用命令行界面）。

一个常见的目标子集可能是，例如：`--model-checker-targets assert,overflow`。

所有目标都被默认检查，除了 Solidity >=0.8.7 的下溢和溢出。

关于如何以及何时分割验证目标，没有精确的指导方法。但在处理大型合约时，它可能是有用的。

验证的目标

如果有任何已证明的目标，SMT 检查器会向每个引擎发出一个警告，说明有多少目标已证明。如果用户希望查看所有已证明的具体目标，可使用命令行选项 `--model-checker-show-proved` 和 JSON 选项 `settings.modelChecker.showProved = true`。

未验证的目标

如果有任何未验证的目标，SMT 检查器会发出一个警告，说明有多少个未验证的目标。如果用户希望看到所有具体的未验证的目标，可以使用命令行界面选项 `--model-checker-show-unproved` 和 JSON 选项 `settings.modelChecker.showUnproved = true`。

不支持的语言功能

SMT 检查器应用的 SMT 编码不完全支持某些 Solidity 语言特性，例如汇编块。不支持的构造通过过度近似进行抽象以保持健全性，这意味着即使不支持该特性，任何报告为安全的属性也是安全的。但是，当目标属性依赖于不支持特征的精确行为时，这种抽象可能会导致误报。如果编码器遇到这种情况，默认情况下会报告一个通用警告，说明它看到了多少不支持的特性。如果用户希望查看所有特定的不支持特性，可以使用命令行界面选项 `--model-checker-show-unsupported` 和 JSON 选项 `settings.modelChecker.showUnsupported = true`，它们的默认值是 `false`。

已验证过的合约

默认情况下，给定来源中的所有可部署合约都会被单独分析，正如将被部署的那一个合约一样。这意味着，如果一个合约有许多直接和间接的继承父类，所有这些都将被单独分析，尽管只有最终派生的合约可以在区块链上被直接访问。这给 SMT 检查器和求解器造成了不必要的负担。为了帮助缓解这样的情况，用户可以指定哪些合约应该作为部署的合约进行分析。当然，基类合约仍然被分析，但只是在分析最终派生的合约的情况下才进行，这可以减少编码和生成查询的复杂性。请注意，抽象合约在默认情况下不会被 SMT 检查器分析为最终派生的合约。

选择的合约可以通过命令行界面，用 <source>:<contract> 形式的键值对，以逗号分隔的列表（不允许有空格）给出：`--model-checker-contracts "<source1.sol:contract1>,<source2.sol:contract2>,<source2.sol:contract3>"`，以及通过*JSON 输入*中的对象 `settings.modelChecker.contracts`，它有如下格式：

```
"contracts": {
    "source1.sol": ["contract1"],
    "source2.sol": ["contract2", "contract3"]
}
```

可信的外部调用

默认情况下，SMT 检查器不会假定编译时可用代码与外部调用的运行时代码相同。以下面的合约为例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Ext {
    uint public x;
    function setX(uint _x) public { x = _x; }
}
contract MyContract {
    function callExt(Ext _e) public {
        _e.setX(42);
        assert(_e.x() == 42);
    }
}
```

当调用 `MyContract.callExt` 时，参数是一个地址。在部署时，我们无法确定地址 _e 是否确实包含合约 `Ext` 的部署。因此，如果 _e 包含除 `Ext` 以外的其他合约，SMT 检查器就会发出警告，称上述断言可能被违反。

不过，将这些外部调用视为可信调用可能很有用，例如，可以测试接口的不同实现是否符合合约相同属性。这意味着假设地址 _e 确实是作为合约 `Ext` 部署的。这种模式可以通过命令行界面选项 `--model-checker-ext-calls=trusted` 或 JSON 字段 `settings.modelChecker.extCalls: "trusted"` 来启用。

请注意，启用该模式会使 SMT 检查器分析的计算成本大大增加。

该模式的一个重要部分是，它适用于合约类型和对合约的高级外部调用，而不是诸如 `call` 和 `delegatecall` 之类的低级调用。地址的存储是按合约类型存储的，SMT 检查器假定外部调用的合约具有调用者表达式的类型。因此，将 `address` 或合约转换为不同的合约类型会产生不同的存储值，如果假设不一致，可能会产生不正确的结果，例如下面的示例：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract D {
    constructor(uint _x) { x = _x; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract E {
    constructor() { x = 2; }
    uint public x;
    function setX(uint _x) public { x = _x; }
}

contract C {
    function f() public {
        address d = address(new D(42));

        // `d` 是被部署的合约 `D`, 因此其 `x` 现在应为 42。
        assert(D(d).x() == 42); // 应该成立
        assert(D(d).x() == 43); // 应该不成立

        // 合约 E 和 合约 D 具有相同的接口, 因此
        // 下面的调用在运行时也会有效。
        // 然而, `E(d)` 的变化并没有反映在 `D(d)` 中。
        E(d).setX(1024);

        // 现在从 `D(d)` 读取将显示旧值。
        // 下面的断言本应在运行时失败,
        // 但在此模式的分析中却成功了（不健全）。
        assert(D(d).x() == 42);
        // 下面的断言在运行时应该成功,
        // 但在此模式的分析中却失败了（假阳性）。
        assert(D(d).x() == 1024);
    }
}
```

由于上述原因，请确保对某个 address 或 contract 类型变量的可信外部调用始终具有相同的调用表达式类型。

在继承的情况下，将被调用合约的变量转换为最派生类型的类型也很有帮助。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
```

(续下页)

(接上页)

```
interface Token {
    function balanceOf(address _a) external view returns (uint);
    function transfer(address _to, uint _amt) external;
}

contract TokenCorrect is Token {
    mapping (address => uint) balance;
    constructor(address _a, uint _b) {
        balance[_a] = _b;
    }
    function balanceOf(address _a) public view override returns (uint) {
        return balance[_a];
    }
    function transfer(address _to, uint _amt) public override {
        require(balance[msg.sender] >= _amt);
        balance[msg.sender] -= _amt;
        balance[_to] += _amt;
    }
}

contract Test {
    function property_transfer(address _token, address _to, uint _amt)␣
↪public {
        require(_to != address(this));

        TokenCorrect t = TokenCorrect(_token);

        uint xPre = t.balanceOf(address(this));
        require(xPre >= _amt);
        uint yPre = t.balanceOf(_to);

        t.transfer(_to, _amt);
        uint xPost = t.balanceOf(address(this));
        uint yPost = t.balanceOf(_to);

        assert(xPost == xPre - _amt);
        assert(yPost == yPre + _amt);
    }
}
```

请注意，在函数 `property_transfer` 中，外部调用是对变量 `t` 执行的。

这种模式的另一个注意事项是调用被分析合约之外的合约类型的状态变量。在下面的代码中，即使 `B` 部署了 `A`，存储在 `B.a` 中的地址也有可能在事务之间被 `B` 以外的任何人调用到 `B` 本身。为了反映 `B.a` 可能发生

的变化，编码允许从外部对 `B.a` 进行次数不限的调用。编码会跟踪 `B.a` 的存储情况，因此断言 (2) 应该成立。然而，目前的编码允许从概念上对 `B` 进行这种调用，因此断言 (3) 不成立。使编码在逻辑上更强大是可信模式的扩展，目前正在开发中。请注意，编码并不跟踪 `address` 变量的存储，因此，如果 `B.a` 的类型是 `address`，编码会假定它的存储在到 `B` 的事务之间不会改变。

```
pragma solidity >=0.8.0;

contract A {
    uint public x;
    address immutable public owner;
    constructor() {
        owner = msg.sender;
    }
    function setX(uint _x) public {
        require(msg.sender == owner);
        x = _x;
    }
}

contract B {
    A a;
    constructor() {
        a = new A();
        assert(a.x() == 0); // (1) 应该成立
    }
    function g() public view {
        assert(a.owner() == address(this)); // (2) 应该成立
        assert(a.x() == 0); // (3) 应该成立, 但失败了, 由于假阳性
    }
}
```

报告推断的归纳变量

对于用 CHC 引擎证明安全的属性，SMT 检查器可以检索由 Horn 求解器推断出的归纳不变式，作为证明的一部分。目前只能向用户报告两种类型的不变式：

- 合约不变量：这些是合约的状态变量的属性，在合约可能运行的每一个可能的事务之前和之后都是真的。例如，`x >= y`，其中 `x` 和 `y` 是一个合约的状态变量。
- 可重入性属性：它们代表了合约在存在对未知代码的外部调用时的行为。这些属性可以表达外部调用前后状态变量的值之间的关系，其中外部调用可以自由地做任何事情，包括对分析的合约进行可重入调用。导数变量代表所述外部调用后的状态变量的值。例如：`lock -> x = x'`。

用户可以使用命令行界面选项 `--model-checker-invariants "contract,reentrancy"` 来选择要报告的不变量类型，或者在*JSON 输入* 中的字段 `settings.modelChecker.invariants` 中作为数组。默

认情况下，SMT 检查器不报告不变量。

有松弛变量的除法和模数运算

Spacer 是 SMT 检查器使用的默认 Horn 求解器，它通常不喜欢 Horn 规则中的除法和模数操作。正因为如此，默认情况下，Solidity 的除法和模运算是用约束条件 `a = b * d + m` 来编码的，其中 `d = a / b` 和 `m = a % b`。然而，对于其他求解器，如 Eldarica，更喜欢语法上的精确操作。命令行标志 `--model-checker-div-mod-no-slacks` 和 JSON 选项 `settings.modelChecker.divModNoSlacks` 可以用来切换编码，这取决于所用求解器的偏好。

Natspec 标签函数抽象化

某些函数包括常见的数学方法，如 `pow` 和 `sqrt`，可能它们过于复杂，无法用完全自动化的方式进行分析。这些函数可以用 Natspec 标签进行注释，向 SMT 检查器表明这些函数应该被抽象化。这意味着在调用此函数时，不会使用函数的主体，函数将：

- 返回一个非决定性的值，如果抽象函数是 view/pure 类型的，则保持状态变量不变，否则会将状态变量设置为非决定性的值。可以通过注解 `//@custom:smtchecker abstract-function-nondet` 来使用。
- 作为一个未被解释的函数。这意味着函数的语义（由主体给出）会被忽略，这个函数的唯一属性是，给定相同的输入，它保证有相同的输出。这一点目前正在开发中，并将通过注解 `//@custom:smtchecker abstract-function-uf` 来使用。

模型检查引擎

SMT 检查器模块实现了两个不同的推理引擎，一个是有界模型检查器（Bounded Model Checker，BMC），一个是约束角条款（Constrained Horn Clauses，CHC）系统。这两个引擎目前都在开发中，并且有不同的特点。这两个引擎是独立的，每一个属性警告都说明它来自哪个引擎。请注意，上面所有带有反例的例子都是由 CHC 这个更强大的引擎报告的。

默认情况下，两个引擎都会被使用，其中首先运行 CHC，每一个没有被证明的属性都被传递给 BMC。您可以通过命令行界面选项 `--model-checker-engine {all,bmc,chc,none}` 或 JSON 选项 `settings.modelChecker.engine {all,bmc,chc,none}` 来选择一个特定的引擎。

有界模型检查器（BMC）

BMC 引擎单独地分析函数，也就是说，它在分析每个函数时不会考虑合约在多个交易中的整体行为。目前在这个引擎中循环也会被忽略了。只要不是直接或间接的递归，内部函数调用是内联的。如果可能的话，外部函数调用是内联的。有可能受重入影响的理论在此被忽略。

上述特点使 BMC 容易报告假阳性，但它也是轻量级的，应该能够快速找到小的局部 bug。

受约束的角条款（Constrained Horn Clauses，CHC）

合约的控制流程图（CFG）被建模为一个 Horn 条款系统，其中合约的生命周期由一个可以非确定性地访问每个公共/外部函数的循环表示。这样，在分析任何函数时都会考虑到整个合约在无限制数量的事务中的行为。这个引擎完全支持循环。支持内部函数调用，而外部函数调用假定被调用的代码是未知的，可以做任何事情。

在能够证明的内容方面，CHC 引擎要比 BMC 强大得多，但可能需要更多的计算资源。

SMT 和 Horn 求解器

上面详述的两个引擎使用自动定理证明器作为其逻辑后端。BMC 使用一个 SMT 求解器，而 CHC 使用一个 Horn 求解器。通常同一个工具可以同时充当这两种工具，如 z3，它主要是一个 SMT 求解器，并将 Spacer 作为一个 Horn 求解器使用，而 Eldarica 则同时做这两种工作。

如果求解器可用的话，用户可以通过命令行界面选项 `--model-checker-solvers {all,cvc4,eld,smtlib2,z3}` 或 JSON 选项 `settings.modelChecker.solvers=[smtlib2,z3]` 来选择应该使用哪个求解器，其中：

- `cvc4` 仅在使用 `solc` 编译二进制文件时可用。并且只有 BMC 使用 `cvc4`。
- `eld` 是通过其二进制文件使用的，必须安装在系统中。只有 CHC 使用了 `eld`，并且是在只有 `z3` 没有被启用的情况下。
- `smtlib2` 以 smtlib2 格式输出 SMT/Horn 查询。这些可以和编译器的 回调机制 一起使用，这样就可以采用系统中的任何求解器二进制来同步返回查询的结果给编译器。根据调用哪个求解器，BMC 和 CHC 都可以使用此方法。
- `z3` 是可用的情况
  - 如果 `solc` 与它一起被编译的话；
  - 如果 Linux 系统中安装了 4.8.x 及其以上版本的动态 `z3` 库（从 Solidity 0.7.6 开始）；
  - 在 `soljson.js` （从 Solidity 0.6.9 开始）中静态的，也就是编译器的 JavaScript 二进制。

备注： z3 4.8.16 版本破坏了与以前版本的 ABI 兼容性，不能与 solc <=0.8.13 版本一起使用。如果您正在使用 z3 >=4.8.16 的版本，请使用 solc >=0.8.14 的版本。反之，只使用旧的 z3 与旧的 solc 版本。我们也建议使用最新的 z3 版本，这也是 SMT 检查器的作用。

由于 BMC 和 CHC 都使用 `z3`，而且 `z3` 可以在更多的环境中使用，包括在浏览器中，大多数用户几乎不需要关心这个选项。更高级的用户可能会应用这个选项，在更复杂的问题上尝试其他求解器。

请注意，所选择的引擎和求解器的某些组合将导致 SMT 检查器不做任何事情，例如选择 CHC 和 `cvc4`。

3.31.3 抽象和假阳性结果

SMT 检查器以一种不完整但健全的方式实现了抽象：如果报告了一个 bug，它可能是由抽象引入的假阳性（由于删除了知识或使用了非精确类型）。如果它确定一个验证目标是安全的，那么它确实是安全的，也就是说，不存在假阴性（除非 SMT 检查器中存在一个 bug）。

如果一个目标不能被证明，您可以尝试通过使用上一节中的调整选项来帮助求解器。如果您确定是假阳性，在代码中加入有更多信息的 `require` 语句也可能给求解器带来一些更多的帮助。

SMT 的编码和类型

SMT 检查器编码试图尽可能精确，将 Solidity 类型和表达式映射到它们最接近的 SMT-LIB 表示法上，正如下表所示。

关于 SMT 编码内部如何工作的更多细节，请参阅论文 基于 SMT 的 Solidity 智能合约验证。

尚不支持的类型由一个 256 位无符号整数抽象出来，其不支持的操作被忽略。

关于 SMT 编码的内部工作方式的更多细节，请参见论文 基于 SMT 的 Solidity 智能合约验证。

函数调用

在 BMC 引擎中，当可能时，即当它们的实现可用时，对相同合约（或基础合约）的函数调用被内联。对其他合约中的函数的调用不被内联，即使它们的代码是可用的，因为我们不能保证实际部署的代码是相同的。

CHC 引擎创建了非线性的 Horn 选项，使用被调用函数的摘要来支持内部函数调用。外部函数调用被视为对未知代码的调用，包括潜在的可重入调用。

复杂的纯函数是由参数上的未转译函数（UF）抽象出来的。

方法	BMC/CHC 运行方式
assert	验证目标。
require	假设。
内部调用	BMC: 内联函数调用。CHC：函数摘要。
对已知代码的外部调用	BMC: 内联函数调用或抹去关于状态变量的记忆和本地存储引用。CHC: 假设被调用的代码是未知的。试图推断出在调用返回后仍然成立的不变性。
存储数组的压栈和出栈	精确地支持检查是否从一个空数组弹出。
ABI 函数	用 UF 函数进行抽象
addmod, mulmod	精确地支持
gasleft, blockhash, keccak256, ecrecover ripemd160	用 UF 函数进行抽象
无执行动作的纯函数（外部或复杂）。	用 UF 函数进行抽象
无执行动作的外部函数	BMC：擦除状态记忆并假定结果是不确定的。CHC：不确定的摘要。试图推断出在调用返回后仍然成立的不变性。
transfer	BMC：检查合约的余额是否足够。CHC：还不执行检查。
其他调用	目前不支持

使用抽象意味着失去精确的知识，但在许多情况下，这并不意味着失去证明力。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Recover
{
    function f(
        bytes32 hash,
        uint8 v1, uint8 v2,
        bytes32 r1, bytes32 r2,
        bytes32 s1, bytes32 s2
    ) public pure returns (address) {
        address a1 = ecrecover(hash, v1, r1, s1);
        require(v1 == v2);
        require(r1 == r2);
        require(s1 == s2);
        address a2 = ecrecover(hash, v2, r2, s2);
        assert(a1 == a2);
        return a1;
    }
}
```

在上面的例子中，SMT 检查器的表达能力不足以实际计算 ecrecover，但通过将函数调用建模为未转译的函数，我们知道在同等参数上调用时返回值是相同的。这就足以证明上面的断言总是正确的。

对于已知是确定性的函数，可以用 UF 来抽象一个函数调用，对于纯函数也很容易做到。然而，对于一般的外部函数来说，这是很难做到的，因为它们可能依赖于状态变量。

引用类型和别名

Solidity 为具有相同数据位置 的引用类型实现了别名。这意味着可以通过对同一数据区域的引用来修改一个变量。SMT 检查器并不跟踪哪些引用是指向相同的数据。这意味着每当分配一个局部引用或引用类型的状态变量时，所有关于相同类型和数据位置的变量的知识都会被抹去。如果类型是嵌套的，知识删除也包括所有的前缀基础类型。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;

contract Aliasing
{
    uint[] array1;
    uint[][] array2;
    function f(
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint[] storage d
    ) internal {
        array1[0] = 42;
        a[0] = 2;
        c[0][0] = 2;
        b[0] = 1;
        // 删除关于内存引用的记忆不应该删除关于状态变量的记忆。
        assert(array1[0] == 42);
        // 但是，对存储引用的赋值将相应地删除存储记忆。
        d[0] = 2;
        // 由于上面的分配，失败为假阳性。
        assert(array1[0] == 42);
        // 失败，因为 `a == b` 是可能的。
        assert(a[0] == 2);
        // 失败，因为 `c[i] == b` 是可能的。
        assert(c[0][0] == 2);
        assert(d[0] == 2);
        assert(b[0] == 1);
    }
    function g(
```

（续下页）

(接上页)

```
        uint[] memory a,
        uint[] memory b,
        uint[][] memory c,
        uint x
    ) public {
        f(a, b, c, array2[x]);
    }
}
```

在对 `b[0]` 进行赋值后，我们需要清除关于 `a` 的知识，因为它有相同的类型（`uint[]`）和数据位置（内存）。我们还需要清除关于 `c` 的知识，因为它的基本类型也是一个位于内存中的 `uint[]`。这意味着一些 `c[i]` 可能与 `b` 或 `a` 指的是同一个数据。

注意，我们没有清除关于 `array` 和 `d` 的知识，因为它们位于存储区，尽管它们也有 `uint[]` 类型。然而，如果 `d` 被分配，我们就需要清除关于 `array` 的知识，反之亦然。

合约余额

如果在部署交易中 `msg.value` > 0，则合约可能在部署时被发送资金。然而，合约的地址在部署前可能已经有了资金，这些资金由合约保存。因此，SMT 检查器在构造函数中假定 `address(this).balance >= msg.value`，以便与 EVM 规则一致。合约的余额也可能在不触发任何对合约的调用的情况下增加，如果

- `selfdestruct` 是由另一个合约执行的，被分析的合约是剩余资金的接收目标。
- 该合约是某个区块的 coinbase（即 `block.coinbase`）。

为了正确建模，SMT 检查器假设在每一笔新的交易中，合约的余额可能至少增长 `msg.value` 的值。

3.31.4 现实世界的假设

有些情况可以在 Solidity 和 EVM 中可以表达出，但可能在实践中不会发生。其中一种情况是动态存储数组的长度在压栈过程中溢出：如果 `push` 操作被应用于一个长度为 2^256 - 1 的数组，它的长度会悄悄溢出。然而，这在实践中不太可能发生，因为将数组增长到这一点所需的操作需要数十亿年的时间来执行。SMT 检查器采取的另一个类似的假设是，一个地址的余额永远不会溢出。

类似的想法在 EIP-1985 中提出过。

3.32 Yul

Yul（先前被也被称为 JULIA 或 IULIA）是一种可以编译到各种不同后端的中间语言。

它可以在独立模式下使用，也可以在 Solidity 内部用于“内联汇编”。编译器在基于 IR 的代码生成器（“新代码生成器”或“基于 IR 的代码生成器”）中使用 Yul 作为一种中间语言。Yul 是高级优化阶段的一个很好的目标，可以使所有的目标平台同样受益。

3.32.1 动机和高级别描述

Yul 的设计试图实现几个目标：

1. 用 Yul 编写的程序应该是可读的，即使代码是由 Solidity 或其他高级语言的编译器生成的。
2. 控制流应易于理解，以帮助人工检查、形式化验证和优化。
3. 从 Yul 到字节码的翻译应该尽可能的简单明了。
4. Yul 应该适用于整个程序的优化。

为了实现第一个和第二个目标，Yul 提供了高级别的结构，如 `for` 循环，`if` 和 `switch` 语句和函数调用。这些应该足以充分代表汇编程序的控制流。因此，没有提供 `SWAP`，`DUP`，`JUMPDEST`，`JUMP` 和 `JUMPI` 的明确语句，因为前两者混淆了数据流，后两者混淆了控制流。此外，`mul(add(x, y), 7)` 形式的函数语句比 `7 y x add mul` 这样的纯操作码语句更受欢迎，因为在第一种形式中，更容易看到哪个操作数用于哪个操作码。

尽管它是为堆栈机设计的，但 Yul 并没有暴露堆栈本身的复杂性。程序员或审计师不应该担心堆栈的问题。

第三个目标是通过以一种非常有规律的方式将高层结构编译成字节码来实现的。汇编器执行的唯一非本地操作是用户定义的标识符（函数、变量......）的名称查找和清理堆栈中的本地变量。

为了避免值和引用等概念之间的混淆，Yul 是静态类型的。同时，有一个默认的类型（通常是目标机的整数字），可以随时省略以帮助增加可读性。

为了保持语言的简单和灵活，Yul 在其纯粹的形式下没有任何内置的操作，函数或类型。在指定 Yul 的语言时，这些操作和语义被添加到一起，这使得 Yul 可以根据不同的目标平台和功能集的要求进行专业化。

目前，只有一种指定的 Yul 语言。这个语言使用 EVM 的操作码作为内建函数（见下文），并且只定义了 `u256` 类型，这是 EVM 的本地 256 位类型。正因为如此，我们将不在下面的例子中提供类型。

3.32.2 简单的例子

下面的例子程序是用 EVM 语言编写的，用来计算指数。它可以用 `solc --strict-assembly` 指令编译。内置函数 `mul` 和 `div` 分别计算乘法和除法。

```
{
    function power(base, exponent) -> result
    {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default
        {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

也可以用 for 循环而不是递归来实现同样的函数。这里，`lt(a, b)` 计算 `a` 是否小于 `b`。

```
{
    function power(base, exponent) -> result
    {
        result := 1
        for { let i := 0 } lt(i, exponent) { i := add(i, 1) }
        {
            result := mul(result, base)
        }
    }
}
```

在本节的末尾，可以找到 ERC-20 标准的完整实现。

3.32.3 单独使用

您可以使用 Solidity 编译器在 EVM 语言中以独立的形式使用 Yul。这将使用 *Yul 对象符号*，这样就有可能将代码作为数据引用到部署合约中。这种 Yul 模式可用于命令行编译器（使用 `--strict-assembly`）和标准-*json* 接口。

```
{
    "language": "Yul",
    "sources": { "input.yul": { "content": "{ sstore(0, 1) }" } },
```

(续下页)

(接上页)

```
    "settings": {
        "outputSelection": { "*": { "*": ["*"], "": [ "*" ] } },
        "optimizer": { "enabled": true, "details": { "yul": true } }
    }
}
```

> **警告：** Yul 正在积极开发中，只有以 EVM 1.0 为目标，Yul 的 EVM 语言才能完全实现字节码生成。

3.32.4 对 Yul 的非正式描述

在下文中，我们将谈论 Yul 语言的每个单独方面。在例子中，我们将使用默认的 EVM 语言。

语法

Yul 使用与 Solidity 相同的方式解析注释，字词和标识符，所以您可以使用 `//` 和 `/* */` 来表示注释。但是有一个例外，Yul 中的标识符可以包含圆点：`.`。

Yul 可以指定由代码，数据和子对象组成的 "对象"。请参阅*Yul 对象* 以了解这方面的详情。在本节中，我们只关注这样一个对象的代码部分。这个代码部分总是由一个大括号限定的块组成。大多数工具都支持只指定一个预期对象的代码块。

在一个代码块内，可以使用以下元素（更多细节见后面章节）：

- 字母，即 `0x123`，`42` 或 `"abc"`（最多 32 个字符的字符串）。
- 对内置函数的调用，例如 `add(1, mload(0))`
- 变量声明，例如 `let x := 7`，`let x := add(y, 3)` 或 `let x`（初始值为 0）
- 标识符（变量），例如：`add(3, x)`
- 赋值，例如：`x := add(y, 3)`
- 局部变量的作用域所在的代码块，例如 `{ let x := 3 { let y := add(x, 1) } }`
- if 语句，例如 `if lt(a, b) { sstore(0, 1) }`
- switch 语句，例如 `switch mload(0) case 0 { revert() } default { mstore(0, 1) }`
- for 循环，例如 `for { let i := 0} lt(i, 10) { i := add(i, 1) } { mstore(i, 7) }`
- 函数的定义，例如 `function f(a, b) -> c { c := add(a, b) }`

多个语法元素之间可以简单地用空格隔开，即不需要结尾的 `;` 或换行。

字面量

作为字面量，您可以使用。

- 以十进制或十六进制符号表示的整数常数。
- ASCII 字符串（例如 "abc"），可能包含十六进制转义 \xNN 和 Unicode 转义 \uNNNN，其中 N 是十六进制数字。
- 十六进制字符串（例如：hex"616263"）。

在 Yul 的 EVM 语言中，字面量表示 256 位的单词，如下所示：

- 十进制或十六进制的常量必须小于 2**256。它们以大端编码的无符号整数形式表示具有该值的 256 位字。
- 一个 ASCII 字符串首先被看作是一个字节序列，通过将非转义 ASCII 字符看作是一个单字节，其值是 ASCII 代码，转义 \xNN 是具有该值的单字节，转义 \uNNNN 是该代码点的 UTF-8 字节序列。字节序列不得超过 32 字节。字节序列在右边用零填充，以达到 32 个字节的长度；换句话说，字符串是以左对齐的方式存储。填充后的字节序列代表一个 256 位的字，其最有意义的 8 位是第一个字节的 1，也就是说，字节被解释为大端形式。
- 十六进制字符串首先被视为一个字节序列，将每一对连续的十六进制数字视为一个字节。字节序列不得超过 32 个字节（即 64 个十六进制数字），并按上述方法处理。

当为 EVM 编译时，这将被翻译成一个适当的 PUSHi 指令。在下面的例子中，3 和 2 相加的结果是 5，然后计算与字符串 “abc” 的按位 与（and）。最后的数值被分配到一个叫做 x 的局部变量。

上述 32 字节的限制并不适用于传递给需要字面参数的内置函数的字符串（例如，setimmutable 或 loadimmutable）。这些字符串最终不会出现在生成的字节码中。

```
let x := and("abc", add(3, 2))
```

除非是默认类型，否则字面的类型必须在冒号后指定：

```
// 这将不会被编译（u32和u256类型尚未实现）。
let x := and("abc":u32, add(3:u256, 2:u256))
```

函数调用

内置函数和用户定义的函数（见下文）都可以用前面例子中的相同方式调用。如果函数返回一个单一的值，它可以直接在一个表达式中再次使用。如果它返回多个值，则必须将它们分配给局部变量。

```
function f(x, y) -> a, b { /* ... */ }
mstore(0x80, add(mload(0x80), 3))
// 此处，用户定义的函数 `f` 返回两个值。
let x, y := f(1, mload(0))
```

对于 EVM 的内置函数，函数表达式可以直接转换为一系列操作码：您只需从右到左读取表达式，就可以得到操作码。在例子中的第二行，是 PUSH1 3 PUSH1 0x80 MLOAD ADD PUSH1 0x80 MSTORE。

对于调用用户定义的函数，参数也从右到左放在堆栈中，这是参数列表被评估的顺序。然而，返回值是在堆栈中从左到右，即在这个例子中，y 在堆栈的顶部，x 在其下方。

变量声明

您可以使用 let 关键字来声明变量。变量只在它所定义的 {...} 块内可见。当编译到 EVM 时，会创建一个新的堆栈槽，为该变量保留，并在到达块的末端时自动移除。您可以为该变量提供一个初始值。如果您不提供一个值，该变量将被初始化为零。

由于变量存储在堆栈中，它们不直接影响内存或存储，但它们可以在内置函数 mstore，mload，sstore 和 sload 中作为内存或存储位置的指针使用。未来的语言可能会为这种指针引入特定的类型。

当一个变量被引用时，其当前值被复制。对于 EVM 来说，这相当于一个 DUP 指令。

```
{
    let zero := 0
    let v := calldataload(zero)
    {
        let y := add(sload(v), 1)
        v := y
    } // y在这里被 “删除” 了
    sstore(v, zero)
} // v和zero在这里被 “删除” 。
```

如果声明的变量应该有一个与默认类型不同的类型，您可以用冒号表示。当您从一个返回多个值的函数调用中赋值时，您也可以在一条语句中声明多个变量。

```
// 这将不会被编译（u32和u256类型尚未实现）。
{
    let zero:u32 := 0:u32
    let v:u256, t:u32 := f()
    let x, y := g()
}
```

根据优化器的设置，编译器可以在变量被最后一次使用后释放堆栈槽，即使它仍然在范围内。

赋值

变量可以在其定义后使用 := 操作符进行赋值。可以在同一时间对多个变量进行赋值。为此，数值的数量和类型必须匹配。如果您想对一个有多个返回参数的函数进行赋值，您必须提供多个变量。同一变量不能多次出现在赋值的左侧，例如：`x, x := f()` 是无效的。

```
let v := 0
// 重新对v赋值
v := 2
let t := add(v, 2)
function f() -> a, b { }
// 赋予多个值
v, t := f()
```

If

if 语句可用于有条件地执行代码。不能定义 "else" 块。如果您需要多种选择条件，可以考虑使用 "switch" 来代替（见下文）。

```
if lt(calldatasize(), 4) { revert(0, 0) }
```

代码块的大括号是必需的。

Switch

您可以使用 switch 语句作为 if 语句的扩展版本。它获取一个表达式的值，并将其与几个字面常量进行比较，与匹配的常量相对应的分支被选中。与其他编程语言不同的是，出于安全考虑，控制流不会从一个条件延续到下一个条件。可以有一个叫 `default` 的回退或默认情况，如果没有一个字面常数匹配，就会采取这种情况。

```
{
    let x := 0
    switch calldataload(4)
    case 0 {
        x := calldataload(0x24)
    }
    default {
        x := calldataload(0x44)
    }
    sstore(0, div(x, 2))
}
```

条件的列表没有用大括号括起来，但条件的主体确实需要大括号。

循环

Yul 支持 for 循环，它由一个包含初始化部分的头，一个条件，一个后迭代部分和一个主体组成。条件必须是一个表达式，而其他三个是代码块。如果初始化部分在顶层声明了任何变量，这些变量的范围将延伸到循环的所有其他部分。

`break` 和 `continue` 语句可以在主体中使用，分别用于退出循环或跳到后部分。

下面的例子是计算内存中一个代码区域的总和。

```
{
    let x := 0
    for { let i := 0 } lt(i, 0x100) { i := add(i, 0x20) } {
        x := add(x, mload(i))
    }
}
```

For 循环也可以作为 while 循环的替代：只需将初始化和后迭代部分留为空即可。

```
{
    let x := 0
    let i := 0
    for { } lt(i, 0x100) { } {     // while(i < 0x100)
        x := add(x, mload(i))
        i := add(i, 0x20)
    }
}
```

函数声明

Yul 允许定义函数。这些不应该与 Solidity 中的函数相混淆，因为它们从来不是一个合约的外部接口的一部分，而是一个独立于 Solidity 函数的命名空间的一部分。

对于 EVM 来说，Yul 函数从堆栈中获取它们的参数（和一个返回的 PC），同时也将结果放到堆栈中。用户定义的函数和内置函数的调用方式完全相同。

函数可以在任何地方定义，并且在它们所声明的块中是可见的。在一个函数中，您不能访问在该函数之外定义的局部变量。

函数声明参数和返回变量，与 Solidity 类似。为了返回一个值，您可以把它分配给返回变量。

如果您调用一个返回多个值的函数，您必须用 `a, b := f(x)` 或 `let a, b := f(x)` 将它们分配给多个变量。

`leave` 语句可以用来退出当前函数。它的工作原理类似于其他语言中的 `return` 语句，只是它不需要返回值，它只是退出函数，函数将返回当前分配给返回变量的任何值。

注意，EVM 语言有一个内置的函数叫 return，它可以退出整个执行环境（内部消息调用），而不仅仅是当前的 yul 函数。

下面的例子通过平方和乘法实现了幂函数。

```
{
    function power(base, exponent) -> result {
        switch exponent
        case 0 { result := 1 }
        case 1 { result := base }
        default {
            result := power(mul(base, base), div(exponent, 2))
            switch mod(exponent, 2)
                case 1 { result := mul(base, result) }
        }
    }
}
```

3.32.5 Yul 形式规范

本章正式描述 Yul 代码。Yul 代码通常放置在 Yul 对象内，Yul 对象将在它们自己的章节中解释。

```
代码块 = '{' 语句* '}'
语句 =
    代码块 |
    函数定义 |
    变量声明 |
    赋值 |
    If |
    表达式 |
    Switch |
    For 循环 |
    循环中断 |
    退出
函数定义 =
    'function' 标识符 '(' 带类型的标识符列表? ')'
    ( '->' 带类型的标识符列表 )? 代码块
变量声明 =
    'let' 带类型的标识符列表 ( ':=' 表达式 )?
赋值 =
    标识符列表 ':=' 表达式
表达式 =
    函数调用 | 标识符 | 字面量
If 条件语句 =
```

(续下页)

(接上页)

```
    'if' 表达式 代码块
Switch 条件语句 =
    'switch' 表达式 ( Case+ Default? | Default )
Case =
    'case' 字面量 代码块
Default =
    'default' 代码块
For 循环 =
    'for' 代码块 表达式 代码块 代码块
循环中断 =
    'break' | 'continue'
退出 = 'leave'
函数调用 =
    标识符 '(' ( 表达式 ( ',' 表达式 )* )? ')'
标识符 = [a-zA-Z_$] [a-zA-Z_$0-9.]*
标识符列表 = 标识符 ( ',' 标识符)*
类型名 = 标识符
带类型的标识符列表 = 标识符 ( ':' 类型名 )? ( ',' 标识符 ( ':' 类型名 )? )*
字面量 =
    (数字字面量 | 字符串字面量 | True字面量 | False字面量) ( ':' 类型名 )?
数字字面量 = 十六进制数字 | 十进制数字
字符串字面量 = '"' ([^"\r\n\\] | '\\' .)* '"'
True字面量 = 'true'
False字面量 = 'false'
十六进制数字 = '0x' [0-9a-fA-F]+
十进制数字 = [0-9]+
```

语法层面的限制

除语法直接规定的限制外，还适用以下限制：

Switch 语句必须至少有一个判断条件（包括默认条件）。所有情况下的值都需要有相同的类型和不同的值。如果表达式类型的所有可能值都被覆盖，则不允许有默认情况（例如，一个带有 `bool` 表达式的 Switch 语句，如果有一个真和一个假的情况，则不允许有默认情况）。

每个表达式都评估为零或多个值。标识符和字面量精确地评估为一个值，而函数调用求值为所调用函数的返回值。

在变量声明和赋值中，右边的表达式（如果存在的话）必须求值到与左边的变量数量相等的数值。这是唯一允许对一个以上的值进行评估的表达式的情况。在赋值或变量声明的左侧，同一个变量名称不能出现多次。

也是语句的表达式（即在块级）必须评估为零值。

在所有其他情况下，表达式必须精确评估为一个值。

`continue` 或 `break` 语句只能在 for 循环的主体内使用，如下所示。考虑包含该语句的最内部循环。循环和

语句必须在同一个函数中，或者两者必须在最高层。该语句必须在循环的主体块中；不能在循环的初始化块或更新块中。值得强调的是，这个限制只适用于包含 `continue` 或 `break` 语句的最内层循环：这个最内层循环，以及 `continue` 或 `break` 语句，可以出现在外层循环的任何地方，可能是外层循环的初始化块或更新块中。例如，下面的例子是合法的，因为 `break` 出现在内循环的主体块中，尽管也出现在外循环的更新块中。

```
for {} true { for {} true {} { break } }
{
}
```

for 循环的条件部分必须精确评估为一个值。

`leave` 语句只能在一个函数内使用。

函数不能在 for 循环初始化块的任何地方定义。

字面量不可以大于它们本身的类型。已定义的最大类型宽度为 256 比特。

在赋值和函数调用过程中，各个值的类型必须匹配。没有隐式的类型转换。一般来说，只有当 EVM 语言提供一个适当的内置函数，接收一个类型的值并返回一个不同类型的值时，才能实现类型转换。

作用域规则

Yul 中的作用域是与块联系在一起的（函数和 for 循环是例外，下面会解释），所有的声明（函数定义 (`FunctionDefinition`)，变量声明 (`VariableDeclaration`)）都将新的标识符引入这些作用域。

标识符在其定义的块中是可见的（包括所有子节点和子块）。函数在整个块中是可见的（甚至在其定义之前），而变量只在 变量声明之后的语句中可见。

特别是，变量不能在其自身变量声明的右侧被引用。函数可以在其声明之前就被引用（如果它们是可见的）。

作为一般范围规则的一个例外，for 循环的"init"部分（第一个块）的范围延伸到 for 循环的所有其他部分。这意味着在 init 部分声明的变量（和函数）（但不在 init 部分的块内）在 for 循环的所有其他部分都是可见的。

在 for 循环的其他部分声明的标识符要遵守常规的句法范围规则。

这意味着一个 for 循环的形式 `for { I... } C { P... } { B... }` 等同于 `{ I... for {}. C { P... } { B... }.`

函数的参数和返回参数在函数体中是可见的，其名称必须是不同的。

在函数内部，不可能引用一个在该函数之外声明的变量。

影子变量是不允许的，也就是说，您不能在另一个同名的标识符也可见的地方声明一个标识符，即使因为它是在当前函数之外声明的而不可能引用它。

形式规范

我们通过提供一个在 AST 的各个节点上重载的评估函数 E 来正式指定 Yul。由于内置函数可能有副作用，E 接收两个状态对象和 AST 节点，并返回两个新的状态对象和数量不定的其他值。这两个状态对象是全局状态对象（在 EVM 的背景下，它是区块链的内存、存储和状态）和本地状态对象（本地变量的状态，即 EVM 中堆栈的一段）。

如果 AST 节点是一个语句，E 返回两个状态对象和一个 “mode”，该 mode 用于 `break`，`continue` 和 `leave` 语句。如果 AST 节点是一个表达式，E 返回两个状态对象和表达式所评估的数值。

全局状态的确切性质在这个高层次的描述中没有明确说明。本地状态 `L` 是标识符 `i` 到值 `v` 的映射，表示为 `L[i] = v`。

对于标识符 `v`, 我们用 `$v` 作为标识符的名字。

我们将为 AST 节点使用解构符号。

```
E(G, L, <{St1, ..., Stn}>: Block) =
    let G1, L1, mode = E(G, L, St1, ..., Stn)
    let L2 be a restriction of L1 to the identifiers of L
    G1, L2, mode
E(G, L, St1, ..., Stn: Statement) =
    if n is zero:
        G, L, regular
    else:
        let G1, L1, mode = E(G, L, St1)
        if mode is regular then
            E(G1, L1, St2, ..., Stn)
        otherwise
            G1, L1, mode
E(G, L, FunctionDefinition) =
    G, L, regular
E(G, L, <let var_1, ..., var_n := rhs>: VariableDeclaration) =
    E(G, L, <var_1, ..., var_n := rhs>: Assignment)
E(G, L, <let var_1, ..., var_n>: VariableDeclaration) =
    let L1 be a copy of L where L1[$var_i] = 0 for i = 1, ..., n
    G, L1, regular
E(G, L, <var_1, ..., var_n := rhs>: Assignment) =
    let G1, L1, v1, ..., vn = E(G, L, rhs)
    let L2 be a copy of L1 where L2[$var_i] = vi for i = 1, ..., n
    G1, L2, regular
E(G, L, <for { i1, ..., in } condition post body>: ForLoop) =
    if n >= 1:
        let G1, L1, mode = E(G, L, i1, ..., in)
        // 由于语法限制，mode 必须是规则的
        if mode is leave then
```

(续下页)

(接上页)

```
            G1, L1 restricted to variables of L, leave
        otherwise
            let G2, L2, mode = E(G1, L1, for {} condition post body)
            G2, L2 restricted to variables of L, mode
    else:
        let G1, L1, v = E(G, L, condition)
        if v is false:
            G1, L1, regular
        else:
            let G2, L2, mode = E(G1, L, body)
            if mode is break:
                G2, L2, regular
            otherwise if mode is leave:
                G2, L2, leave
            else:
                G3, L3, mode = E(G2, L2, post)
                if mode is leave:
                    G3, L3, leave
                otherwise
                    E(G3, L3, for {} condition post body)
E(G, L, break: BreakContinue) =
    G, L, break
E(G, L, continue: BreakContinue) =
    G, L, continue
E(G, L, leave: Leave) =
    G, L, leave
E(G, L, <if condition body>: If) =
    let G0, L0, v = E(G, L, condition)
    if v is true:
        E(G0, L0, body)
    else:
        G0, L0, regular
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn>: Switch) =
    E(G, L, switch condition case l1:t1 st1 ... case ln:tn stn default {})
E(G, L, <switch condition case l1:t1 st1 ... case ln:tn stn default st'>: Switch) =
    let G0, L0, v = E(G, L, condition)
    // i = 1 .. n
    // 对字面量求值，上下文无关
    let _, _, v1 = E(G0, L0, l1)
    ...
    let _, _, vn = E(G0, L0, ln)
    if there exists smallest i such that vi = v:
        E(G0, L0, sti)
```

(续下页)

(接上页)

```
    else:
        E(G0, L0, st')

E(G, L, <name>: Identifier) =
    G, L, L[$name]
E(G, L, <fname(arg1, ..., argn)>: FunctionCall) =
    G1, L1, vn = E(G, L, argn)
    ...
    G(n-1), L(n-1), v2 = E(G(n-2), L(n-2), arg2)
    Gn, Ln, v1 = E(G(n-1), L(n-1), arg1)
    Let <function fname (param1, ..., paramn) -> ret1, ..., retm block>
    be the function of name $fname visible at the point of the call.
    Let L' be a new local state such that
    L'[$parami] = vi and L'[$reti] = 0 for all i.
    Let G'', L'', mode = E(Gn, L', block)
    G'', Ln, L''[$ret1], ..., L''[$retm]
E(G, L, l: StringLiteral) = G, L, str(l),
    where str is the string evaluation function,
    which for the EVM dialect is defined in the section 'Literals' above
E(G, L, n: HexNumber) = G, L, hex(n)
    where hex is the hexadecimal evaluation function,
    which turns a sequence of hexadecimal digits into their big endian value
E(G, L, n: DecimalNumber) = G, L, dec(n),
    where dec is the decimal evaluation function,
    which turns a sequence of decimal digits into their big endian value
```

EVM 语言

目前 Yul 的默认语言是当前选择的 EVM 版本的 EVM 语言，与 EVM 的一个版本。该语言中唯一可用的类型是 `u256`，即 Ethereum 虚拟机的 256 位本地类型。因为它是该语言的默认类型，所以可以省略。

下表列出了所有内置函数（取决于 EVM 版本），并提供了函数/操作码的语义的简短描述。本文件并不想成为以太坊虚拟机的完整描述。如果您对精确的语义感兴趣，请参考另一份文件。

标有 `-` 的操作码不返回结果，所有其他操作码正好返回一个值。标有 `F`，`H`，`B`，`C`，`I`，`L``和 ``P` 的操作码分别是从 Frontier，Homestead，Byzantium，Constantinople，Istanbul，London 或 Paris 版本出现的。

在下文中，`mem[a...b)` 表示从位置 `a` 开始到不包括位置 `b` 的内存字节，`storage[p]` 表示插槽 `p` 的存储内容。

由于 Yul 管理着局部变量和控制流，所以不能使用干扰这些功能的操作码。这包括 `dup` 和 `swap` 指令，以及 `jump` 指令，标签和 `push` 指令。

指令			解释
stop()	-	F	停止执行，与 return(0, 0) 相同
add(x, y)		F	x + y
sub(x, y)		F	x - y
mul(x, y)		F	x * y
div(x, y)		F	x / y 或如果 y == 0，则为 0
sdiv(x, y)		F	x / y，对于有符号的二进制补数，如果 y == 0，则为 0
mod(x, y)		F	x % y, 如果 y == 0，则为 0
smod(x, y)		F	x % y, 对于有符号的二进制补数, 如果 y == 0，则为 0
exp(x, y)		F	x 的 y 次方
not(x)		F	x 的位" 非"（x 的每一个位都取反）
lt(x, y)		F	如果 x < y，则为 1，否则为 0
gt(x, y)		F	如果 x > y，则为 1，否则为 0
slt(x, y)		F	如果 x < y，则为 1，否则为 0，适用于有符号的二进制数
sgt(x, y)		F	如果 x > y，则为 1，否则为 0，适用于有符号的二进制补数
eq(x, y)		F	如果 x == y，则为 1，否则为 0
iszero(x)		F	如果 x == 0，则为 1，否则为 0
and(x, y)		F	x 和 y 的按位" 与"
or(x, y)		F	x 和 y 的按位" 或"
xor(x, y)		F	x 和 y 的按位" 异或"
byte(n, x)		F	x 的第 n 个字节，其中最重要的字节是第 0 个字节
shl(x, y)		C	将 y 逻辑左移 x 位
shr(x, y)		C	将 y 逻辑右移 x 位
sar(x, y)		C	将 y 算术右移 x 位
addmod(x, y, m)		F	(x + y) % m，采用任意精度算术，如果 m == 0 则为 0
mulmod(x, y, m)		F	(x * y) % m，采用任意精度算术，如果 m == 0 则为 0
signextend(i, x)		F	从第 (i*8+7) 位开始进行符号扩展，从最低符号位开始计算
keccak256(p, n)		F	keccak(mem[p...(p+n)))
pc()		F	代码中的当前位置
pop(x)	-	F	丢弃值 x
mload(p)		F	mem[p...(p+32))
mstore(p, v)	-	F	mem[p...(p+32)) := v
mstore8(p, v)	-	F	mem[p] := v & 0xff （(只修改了一个字节)）
sload(p)		F	storage[p]
sstore(p, v)	-	F	storage[p] := v
msize()		F	内存的大小，即最大的访问内存索引
gas()		F	仍可以执行的气体值
address()		F	当前合约/执行环境的地址
balance(a)		F	地址为 A 的余额，以 wei 为单位

指令			解释
selfbalance()		I	相当于 balance(address())，但更便宜
caller()		F	消息调用者（不包括 `delegatecall` 调用）。
callvalue()		F	与当前调用一起发送的 wei 的数量
calldataload(p)		F	从位置 p 开始的调用数据（32 字节）
calldatasize()		F	调用数据的大小，以字节为单位
calldatacopy(t, f, s)	-	F	从位置 f 的 calldata 复制 s 字节到位置 t 的内存中
codesize()		F	当前合约/执行环境的代码大小
codecopy(t, f, s)	-	F	从位置 f 的 code 中复制 s 字节到位置 t 的内存中
extcodesize(a)		F	地址为 a 的代码的大小
extcodecopy(a, t, f, s)	-	F	像 codecopy(t, f, s) 一样，但在地址 a 处取代码
returndatasize()		B	最后返回数据的大小
returndatacopy(t, f, s)	-	B	从位置 f 的 returndata 复制 s 字节到位置 t 的内存中
extcodehash(a)		C	地址 a 的代码哈希值
create(v, p, n)		F	用代码 mem[p...(p+n)) 创建新的合约，发送 v 数量的 wei 并返回新地址；错误
create2(v, p, n, s)		C	在 keccak256(0xff . this . s . keccak256(mem[p...(p+n))) 地址处创建代码为 mem
call(g, a, v, in, insize, out, outsize)		F	调用地址 a 上的合约，以 mem[in..(in+insize)) 作为输入一并发送 g 数量的 gas
callcode(g, a, v, in, insize, out, outsize)		F	相当于 `call` 但仅仅使用地址 a 上的代码，执行时留在当前合约的上下文当中
delegatecall(g, a, in, insize, out, outsize)		H	相当于 `callcode`，但同时保留 `caller` 和 `callvalue` 查看更多
staticcall(g, a, in, insize, out, outsize)		B	相当于 `call(g, a, 0, in, insize, out, outsize)` 但不允许状态
return(p, s)	-	F	终止执行，返回 mem[p..(p+s)) 上的数据
revert(p, s)	-	B	终止执行，恢复状态变更，返回 mem[p..(p+s)) 上的数据
selfdestruct(a)	-	F	终止执行，销毁当前合约，并且将余额发送到地址 a （已废弃）
invalid()	-	F	以无效指令终止执行
log0(p, s)	-	F	用 mem[p..(p+s)] 上的数据产生日志
log1(p, s, t1)	-	F	用 mem[p..(p+s)] 上的数据和 topic t1 产生日志
log2(p, s, t1, t2)	-	F	用 mem[p..(p+s)] 上的数据和 topic t1，t2 产生日志
log3(p, s, t1, t2, t3)	-	F	用 mem[p..(p+s)] 上的数据和 topic t1，t2，t3 产生日志
log4(p, s, t1, t2, t3, t4)	-	F	用 mem[p..(p+s)] 上的数据和 topic t1，t2，t3，t4 产生日志
chainid()		I	执行链的 ID（EIP-1344）
basefee()		L	当前区块的基本费用（EIP-3198 和 EIP-1559）
origin()		F	交易发送者
gasprice()		F	交易的气体价格
blockhash(b)		F	区块编号 b 的哈希值--只针对最近的 256 个区块，不包括当前区块。
coinbase()		F	目前的挖矿的受益者
timestamp()		F	自 epoch 开始的，当前块的时间戳，以秒为单位
number()		F	当前区块号
difficulty()		F	当前区块的难度（见下面的注释）

指令			解释
prevrandao()		P	由信标链提供的随机性（见下面的注释）
gaslimit()		F	当前区块的区块 gas 限制

备注： `call*` 指令使用 `out` 和 `outsize` 参数来在内存中定义的一个区域，用于放置返回或失败数据。这个区域的写入取决于被调用的合约返回多少字节。如果它返回更多的数据，只有第一个 `outsize` 字节被写入。您可以使用 `returndatacopy` 操作码访问其余的数据。如果它返回较少的数据，那么剩下的字节根本不被触及。您需要使用 `returndatacopy` 操作码来检查这个内存区域的哪一部分包含返回数据。剩下的字节将保留调用前的值。

备注： `difficulty()` 指令在 EVM >= Paris 版本中是不允许的。随着 Paris 网络的升级，以前被称为 `difficulty` 的指令的语义已经改变，该指令被重新命名为 `prevrandao`。它现在可以返回全 256 位范围内的任意值，而 Ethash 内部记录的最高难度值是 ~54 位。这一变化在 EIP-4399 中有所描述。请注意，与在编译器中选择哪个 EVM 版本无关，指令的语义取决于最终的部署链。

警告： 从 0.8.18 及更高版本开始，在 Solidity 和 Yul 中使用 `selfdestruct` 将触发弃用警告，因为 `SELFDESTRUCT` 操作码最终将经历 EIP-6049 中所述的行为的重大变化。

在一些内部语言中，还有一些额外的函数：

datasize, dataoffset, datacopy

函数 `datasize(x)`，`dataoffset(x)` 和 `datacopy(t, f, l)` 用来访问 Yul 对象的其他部分。

`datasize` 和 `dataoffset` 只能接受字符串字面量（其他对象的名称）作为参数，并分别返回数据区的大小和偏移量。对于 EVM，`datacopy` 函数等同于 `codecopy`。

setimmutable, loadimmutable

函数 `setimmutable(offset, "name", value)` 和 `loadimmutable("name")` 用于 Solidity 中的不可变机制，不能很好地映射到纯 Yul。对 `setimmutable(offset, "name", value)` 的调用假定包含给定不可变的命名的合约的运行时代码在偏移量 `offset` 处被复制到内存中，并将把 `value` 写到内存中的所有位置（相对于 `offset`），这些位置包含在运行时代码中为调用 `loadimmutable("name")` 产生的占位符。

linkersymbol

函数 `linkersymbol("library_id")` 是一个占位符，用来表示被链接器替换的地址字头。它的第一个也是唯一的参数必须是一个字符串字面量，并且唯一地代表要插入的地址。标识符可以是任意的，但是当编译器从 Solidity 源产生 Yul 代码时，它使用一个库名，并以定义该库的源单元的名称作为限定。要用一个特定的库地址链接代码，必须在命令行上的 `--libraries` 选项中提供相同的标识符。

例如，这段代码

```
let a := linkersymbol("file.sol:Math")
```

相当于

```
let a := 0x1234567890123456789012345678901234567890
```

当使用 `--libraries "file.sol:Math=0x1234567890123456789012345678901234567890` 选项调用链接器时。

请参阅使用命令行编译器 以了解有关 Solidity 链接器的详情。

memoryguard

调用 `let ptr := memoryguard(size)` 的调用者（其中 `size` 必须是一个数字字面量）承诺他们只使用 `[0, size]` 范围内的内存，或者从 `ptr` 开始的无界范围。

由于 `memoryguard` 调用的存在表明所有的内存访问都遵守这一限制，它允许优化器执行额外的优化步骤，例如堆栈限制规避器，它试图将原本无法到达的堆栈变量转移到内存中。

Yul 优化器承诺只使用内存范围 `[size, ptr)` 来实现其目的。如果优化器不需要保留任何内存，它认为 `ptr == size`。

`memoryguard` 可以被多次调用，但是需要在一个 Yul 子对象内有相同的字样作为参数。如果在一个子对象中发现至少一个 `memoryguard` 的调用，额外的优化步骤将在它身上运行。

verbatim

一组 `verbatim...` 内置函数可以让您为 Yul 编译器不知道的操作码创建字节码。它还允许您创建不会被优化器修改的字节码序列。

这些函数是 `verbatim_<n>i_<m>o("<data>", ...)`，其中

- `n` 是一个介于 0 和 99 之间的小数，指定输入栈槽/变量的数量
- `m` 是一个介于 0 和 99 之间的小数，指定输出栈槽/变量的数量
- `data` 是一个字符串字面量，包含字节的序列

例如，如果您想定义一个函数，将输入值乘以 2，而不需要优化器触及常数 2，您可以使用

```
let x := calldataload(0)
let double := verbatim_1i_1o(hex"600202", x)
```

这段代码将产生一个 `dup1` 操作码来检索 `x`（尽管优化器可能直接重新使用 `calldataload` 操作码的结果），后面直接是 `600202`。该代码被假定为消耗 `x` 的复制值，并在堆栈顶部产生结果。然后编译器生成代码，为 `double` 分配一个堆栈槽，并将结果存储在那里。

与所有的操作码一样，参数被安排在堆栈中，最左边的参数在最上面，而返回值则被假定是以最右边的变量在栈顶的方式排列的。

由于 `verbatim` 可以用来生成任意的操作码，甚至是 Solidity 编译器不知道的操作码，在与优化器一起使用 `verbatim` 时，必须小心。即使优化器被关闭，代码生成器也必须确定堆栈布局，这意味着，例如，使用 `verbatim` 来修改堆栈高度会导致未定义行为。

下面是一个不完全的列表，列出了对逐字字节码的限制，这些限制不被编译器检查。违反这些限制会导致未定义的行为。

- 控制流不应该跳入或跳出 verbatim 块，但它可以在同一个 verbatim 块内跳入。
- 除了输入和输出参数外，堆栈内容不应该被访问。
- 堆栈的高度差应该正好是 `m - n` （输出槽减去输入槽）。
- Verbatim 字节码不能对周围的字节码做任何假设。所有需要的参数都必须作为堆栈变量传入。

优化器不分析 verbatim 字节码，总是假设它修改了状态的所有方面，因此只能在 `verbatim` 函数调用中做很少的优化。

优化器将 verbatim 字节码视为一个不透明的代码块。它不会分割它，但可能会移动、重复或与相同的 verbatim 字节码块结合。如果一个 verbatim 的字节码块不能被控制流所触及。它可以被删除。

警告： 在讨论 EVM 的改进是否会破坏现有的智能合约时，`verbatim` 内部的功能不能得到与 Solidity 编译器本身使用的功能一样的考虑。

备注： 为了避免混淆，所有以字符串 `verbatim` 开头的标识符都被保留，不能用于用户定义的标识符。

3.32.6 Yul 对象的规范

Yul 对象被用来分组命名代码和数据部分。函数 `datasize`，`dataoffset` 和 `datacopy` 可以用来从代码中访问这些部分。十六进制字符串可用于指定十六进制编码的数据，普通字符串为本地编码。对于代码，`datacopy` 将访问其组装的二进制所表示的数据。

```
对象 = 'object' 字面量 '{' 代码 ( 对象 | 数据 )* '}'
代码 = 'code' 块
数据 = 'data' 字面量 ( 十六进制字面量 | 字面量 )
十六进制字面量 = 'hex' ('"' ([0-9a-fA-F]{2})* '"' | '\'' ([0-9a-fA-F]{2})* '\'')
字面量 = '"' ([^"\r\n\\] | '\\' .)* '"'
```

对于上面的 Block，指的是前一章解释的 Yul 代码语法中的 Block。

备注： 当一个对象的名称以 _deployed 结尾时，Yul 优化器将其视为部署的代码。这样做的唯一后果是优化器中的不同 gas 成本启发式算法。

备注： 可以定义名称中包含 . 的数据对象或子对象，但不可能通过 datasize，dataoffset 或 datacopy 访问它们，因为 . 是作为分隔符用来访问另一个对象内的对象。

备注： 被称为 ".metadata" 的数据对象有特殊意义：它不能从代码中访问，并且总是被附加到字节码的最末端，无论它在对象中的位置如何。

其他具有特殊意义的数据对象在未来可能会被添加，但它们的名字总是以 . 开头。

下面是一个 Yul 对象的例子：

```
// 一个合约由一个单一的对象组成，
// 其子对象代表要部署的代码或它可以创建的其他合约。
// 单个 “代码” 节点是该对象的可执行代码。
// 每一个 (其他) 命名的对象或数据部分都被序列化，
// 并被特殊的内置函数 datacopy / dataoffset / datasize 所访问
// 当前对象、子对象和当前对象内的数据项都在范围内。
object "Contract1" {
    // 这是合约的构造函数代码。
    code {
        function allocate(size) -> ptr {
            ptr := mload(0x40)
            // 请注意，Solidity 生成的 IR 代码也保留了内存偏移量 ``0x60``，但一个纯
↪Yul 对象可以自由地使用内存。
            if iszero(ptr) { ptr := 0x60 }
            mstore(0x40, add(ptr, size))
        }

        // 首先创建 “Contract2”
        let size := datasize("Contract2")
```

(续下页)

(接上页)

```
        let offset := allocate(size)
        // 这将转化为EVM的代码拷贝。
        datacopy(offset, dataoffset("Contract2"), size)
        // 构造函数参数是一个单一的数字 0x1234
        mstore(add(offset, size), 0x1234)
        pop(create(0, offset, add(size, 32)))

        // 现在返回运行时对象
        // 当前执行的代码是构造函数代码）。
        size := datasize("Contract1_deployed")
        offset := allocate(size)
        // 这将变成 Ewasm 的 内存->内存 复制
        // 和 EVM 的代码复制。
        datacopy(offset, dataoffset("Contract1_deployed"), size)
        return(offset, size)
    }

    data "Table2" hex"4123"

    object "Contract1_deployed" {
        code {
            function allocate(size) -> ptr {
                ptr := mload(0x40)
                // 请注意，Solidity 生成的 IR 代码也保留了内存偏移量␣
↪``0x60``，但一个纯 Yul 对象可以自由地使用内存。
                if iszero(ptr) { ptr := 0x60 }
                mstore(0x40, add(ptr, size))
            }

            // 运行时代码

            mstore(0, "Hello, World!")
            return(0, 0x20)
        }
    }

    // 嵌入对象。使用情况是，外面是一个工厂合约，
    // 而 Contract2 是由工厂创建的代码。
    object "Contract2" {
        code {
            // 此处是代码 ...
        }
```

(续下页)

(接上页)

```
        object "Contract2_deployed" {
            code {
                // 此处是代码 ...
            }
        }

        data "Table1" hex"4123"
    }
}
```

3.32.7 Yul 优化器

Yul 优化器对 Yul 代码进行操作，并对输入、输出和中间状态使用相同的语言。这使得优化器的调试和验证变得容易。

请参考一般的优化器文档，以了解关于不同优化阶段和如何使用优化器的更多细节。

如果您想在独立的 Yul 模式下使用 Solidity，您可以用 `--optimize` 激活优化器，并可选择用 `--optimize-runs` 指定预期合约执行次数：

```
solc --strict-assembly --optimize --optimize-runs 200
```

在 Solidity 模式下，Yul 优化器与常规优化器一起被激活。

优化步骤顺序

有关优化顺序的详细信息以及缩写列表可在优化器文档 中找到。

3.32.8 完整的 ERC20 示例（基于 yul）

```
object "Token" {
    code {
        // 将创建者存储在零号槽中。
        sstore(0, caller())

        // 部署合约
        datacopy(0, dataoffset("runtime"), datasize("runtime"))
        return(0, datasize("runtime"))
    }
    object "runtime" {
        code {
            // 防止发送以太的保护措施
```

(续下页)

(接上页)

```
        require(iszero(callvalue()))

        // 调度器
        switch selector()
        case 0x70a08231 /* "balanceOf(address)" */ {
            returnUint(balanceOf(decodeAsAddress(0)))
        }
        case 0x18160ddd /* "totalSupply()" */ {
            returnUint(totalSupply())
        }
        case 0xa9059cbb /* "transfer(address,uint256)" */ {
            transfer(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        case 0x23b872dd /* "transferFrom(address,address,uint256)" */ {
            transferFrom(decodeAsAddress(0), decodeAsAddress(1), decodeAsUint(2))
            returnTrue()
        }
        case 0x095ea7b3 /* "approve(address,uint256)" */ {
            approve(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        case 0xdd62ed3e /* "allowance(address,address)" */ {
            returnUint(allowance(decodeAsAddress(0), decodeAsAddress(1)))
        }
        case 0x40c10f19 /* "mint(address,uint256)" */ {
            mint(decodeAsAddress(0), decodeAsUint(1))
            returnTrue()
        }
        default {
            revert(0, 0)
        }

        function mint(account, amount) {
            require(calledByOwner())

            mintTokens(amount)
            addToBalance(account, amount)
            emitTransfer(0, account, amount)
        }
        function transfer(to, amount) {
            executeTransfer(caller(), to, amount)
        }
```

(续下页)

(接上页)

```
            function approve(spender, amount) {
                revertIfZeroAddress(spender)
                setAllowance(caller(), spender, amount)
                emitApproval(caller(), spender, amount)
            }
            function transferFrom(from, to, amount) {
                decreaseAllowanceBy(from, caller(), amount)
                executeTransfer(from, to, amount)
            }

            function executeTransfer(from, to, amount) {
                revertIfZeroAddress(to)
                deductFromBalance(from, amount)
                addToBalance(to, amount)
                emitTransfer(from, to, amount)
            }


            /* ---------- calldata 解码函数 ----------- */
            function selector() -> s {
                s := div(calldataload(0),␣
↪0x100000000000000000000000000000000000000000000000000000000)
            }

            function decodeAsAddress(offset) -> v {
                v := decodeAsUint(offset)
                if iszero(iszero(and(v,␣
↪not(0xffffffffffffffffffffffffffffffffffffffff)))) {
                    revert(0, 0)
                }
            }
            function decodeAsUint(offset) -> v {
                let pos := add(4, mul(offset, 0x20))
                if lt(calldatasize(), add(pos, 0x20)) {
                    revert(0, 0)
                }
                v := calldataload(pos)
            }
            /* ---------- calldata 编码函数 ---------- */
            function returnUint(v) {
                mstore(0, v)
                return(0, 0x20)
            }
```

(续下页)

(接上页)

```
            function returnTrue() {
                returnUint(1)
            }

            /* -------- 事件 ---------- */
            function emitTransfer(from, to, amount) {
                let signatureHash :=␣
↪0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
                emitEvent(signatureHash, from, to, amount)
            }
            function emitApproval(from, spender, amount) {
                let signatureHash :=␣
↪0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
                emitEvent(signatureHash, from, spender, amount)
            }
            function emitEvent(signatureHash, indexed1, indexed2, nonIndexed) {
                mstore(0, nonIndexed)
                log3(0, 0x20, signatureHash, indexed1, indexed2)
            }

            /* -------- 存储布局 ---------- */
            function ownerPos() -> p { p := 0 }
            function totalSupplyPos() -> p { p := 1 }
            function accountToStorageOffset(account) -> offset {
                offset := add(0x1000, account)
            }
            function allowanceStorageOffset(account, spender) -> offset {
                offset := accountToStorageOffset(account)
                mstore(0, offset)
                mstore(0x20, spender)
                offset := keccak256(0, 0x40)
            }

            /* -------- 存储访问 ---------- */
            function owner() -> o {
                o := sload(ownerPos())
            }
            function totalSupply() -> supply {
                supply := sload(totalSupplyPos())
            }
            function mintTokens(amount) {
                sstore(totalSupplyPos(), safeAdd(totalSupply(), amount))
            }
```

(续下页)

(接上页)

```
        function balanceOf(account) -> bal {
            bal := sload(accountToStorageOffset(account))
        }
        function addToBalance(account, amount) {
            let offset := accountToStorageOffset(account)
            sstore(offset, safeAdd(sload(offset), amount))
        }
        function deductFromBalance(account, amount) {
            let offset := accountToStorageOffset(account)
            let bal := sload(offset)
            require(lte(amount, bal))
            sstore(offset, sub(bal, amount))
        }
        function allowance(account, spender) -> amount {
            amount := sload(allowanceStorageOffset(account, spender))
        }
        function setAllowance(account, spender, amount) {
            sstore(allowanceStorageOffset(account, spender), amount)
        }
        function decreaseAllowanceBy(account, spender, amount) {
            let offset := allowanceStorageOffset(account, spender)
            let currentAllowance := sload(offset)
            require(lte(amount, currentAllowance))
            sstore(offset, sub(currentAllowance, amount))
        }

        /* ---------- 工具函数 ---------- */
        function lte(a, b) -> r {
            r := iszero(gt(a, b))
        }
        function gte(a, b) -> r {
            r := iszero(lt(a, b))
        }
        function safeAdd(a, b) -> r {
            r := add(a, b)
            if or(lt(r, a), lt(r, b)) { revert(0, 0) }
        }
        function calledByOwner() -> cbo {
            cbo := eq(owner(), caller())
        }
        function revertIfZeroAddress(addr) {
            require(addr)
        }
```

(续下页)

(接上页)

```
            function require(condition) {
                if iszero(condition) { revert(0, 0) }
            }
        }
    }
}
```

3.33 导入路径解析

为了能够在所有平台上支持可重复的构建，Solidity 编译器必须抽象出存储源文件的文件系统的细节。在导入中使用的路径必须在任何地方以同样的方式工作，而命令行界面必须能够与平台特定的路径一起工作，以提供良好的用户体验。本节旨在详细解释 Solidity 是如何协调这些要求的。

3.33.1 虚拟文件系统

编译器维护一个内部数据库（*虚拟文件系统或简称 VFS*），每个源单元被分配一个唯一的 *源单元名称*，这是一个不透明的非结构化的标识符。当您使用*import 语句* 时，您指定了引用源单元名称的 *导入路径*。

导入回调

VFS 最初只填充了编译器收到的输入文件。在编译过程中可以使用 *import 回调*加载其他文件，但这取决于您使用的编译器的类型（见下文）。如果编译器在 VFS 中没有找到任何与导入路径相匹配的源单元名称，它就会调用回调，负责获取要放在该名称下的源代码。一个导入回调可以自由地以任意方式解释源单元名称，而不仅仅是作为路径。如果在需要回调时没有可用的回调，或者无法找到源代码，编译就会失败。

命令行编译器提供了 *主机文件系统加载器* -- 一个基本的回调，它将源单元名称解释为本地文件系统中的一个路径。JavaScript 接口 默认不提供任何接口，但可以由用户提供一个。这个机制可以用来从本地文件系统以外的地方获得源代码（本地文件系统甚至可能无法访问，例如当编译器在浏览器中运行时）。例如，Remix IDE 提供了一个多功能的回调，让您 从 HTTP、IPFS 和 Swarm URL 导入文件，或直接引用 NPM 注册表中的包。

备注： 主机文件系统加载器的文件查找是依赖于平台的。例如，源单元名称中的反斜线可以被解释为目录分隔符，也可以不被解释为目录分隔符，查找时可以区分大小写，这取决于底层平台。

为了实现可移植性，我们建议避免使用只有在特定的导入回调中才能正常工作的导入路径，或者只在一个平台上使用。例如，您应该总是使用正斜线，因为它们在支持反斜线的平台上也能作为路径分隔符使用。

虚拟文件系统的初始内容

VFS 的初始内容取决于您如何调用编译器：

1. **solc / 命令行界面**

   当您使用编译器的命令行界面编译一个文件时，您提供一个或多个包含 Solidity 代码的文件的路径：

   ```
   solc contract.sol /usr/local/dapp-bin/token.sol
   ```

   以这种方式加载的文件的源单元名称是通过将其路径转换为规范的形式来构建的，如果可能的话，使其与基本路径或其中一个包含路径相对。参见*CLI 路径规范化和剥离* 以了解这一过程的详细描述。

2. **标准 JSON**

   当使用标准 *JSON* API 时通过 JavaScript 接口 或 *--standard-json* 命令行选项），您需提供 JSON 格式的输入，其中包含您所有源文件的内容。

   ```
   {
       "language": "Solidity",
       "sources": {
           "contract.sol": {
               "content": "import \"./util.sol\";\ncontract C {}"
           },
           "util.sol": {
               "content": "library Util {}"
           },
           "/usr/local/dapp-bin/token.sol": {
               "content": "contract Token {}"
           }
       },
       "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
   }
   ```

   上面的 `sources` 字典结构成为虚拟文件系统的初始内容，它的键被用作源单元名称。

3. **标准 JSON（通过导入回调）**

   通过标准 JSON，也可以告诉编译器使用导入回调来获得源代码：

   ```
   {
       "language": "Solidity",
       "sources": {
           "/usr/local/dapp-bin/token.sol": {
               "urls": [
                   "/projects/mytoken.sol",
                   "https://example.com/projects/mytoken.sol"
               ]
   ```

(续下页)

(接上页)

```
        }
    },
    "settings": {"outputSelection": {"*": { "*": ["metadata", "evm.bytecode"]}}}
}
```

如果导入回调是可用的，编译器将一个一个地给它 urls 中指定的字符串，直到有一个被成功加载或到达列表的末尾。

源单元名称的确定方式与使用 content 时相同 - 它们是 sources 字典结构的键，urls 的内容不会以任何方式影响它们。

4. **标准输入**

   在命令行中，也可以通过将源代码发送到编译器的标准输入来提供源代码:

```
echo 'import "./util.sol"; contract C {}' | solc -
```

   - 作为参数之一，指示编译器将标准输入的内容放在虚拟文件系统中的一个特殊的源单元名下：<stdin>。

初始化 VFS 之后，仍然可以向它添加其他文件，但只能通过导入回调的方式。

3.33.2 导入

导入语句指定了一个 导入路径。根据导入路径的指定方式，我们可以将导入分为两类：

- 直接导入，直接指定完整的源单元名称。
- 相对导入，指定一个以 ./ 或 ../ 开头的路径，与导入文件的源单元名称相结合。

列表 1: contracts/contract.sol

```
import "./math/math.sol";
import "contracts/tokens/token.sol";
```

在上面的 ./math/math.sol 和 contracts/tokens/token.sol 都是导入路径，然而它们转译成的源单元名分别是 contracts/math/math.sol 和 contracts/tokens/token.sol。

直接导入

不以 ./ 或 ../ 开头的导入是 直接导入。

```
import "/project/lib/util.sol";         // 源单元名称： /project/lib/util.sol
import "lib/util.sol";                  // 源单元名称： lib/util.sol
import "@openzeppelin/address.sol";     // 源单元名称： @openzeppelin/address.sol
import "https://example.com/token.sol"; // 源单元名称： https://example.com/token.sol
```

在应用任何导入重映射 之后，导入路径简单地成为源单元名称。

备注： 一个源单元的名字只是一个标识符，即使它的值碰巧看起来像一个路径，它也不受您在 shell 中通常期望的规范化规则的约束。任何 /./ 或 /../ 的注释段或多个斜线的序列都是它的一部分。当源是通过标准 JSON 接口提供的时候，完全有可能将不同的内容与源单元的名称联系起来，这些名称将指代磁盘上的同一个文件。

当源文件在虚拟文件系统中不可用时，编译器会将源单元名称传递给导入回调。主机文件系统加载器将尝试使用它作为路径并在磁盘上查找文件。在这一点上，平台特定的规范化规则开始发挥作用，在 VFS 中被认为是不同的名字实际上可能导致同一个文件被加载。例如，/project/lib/math.sol 和 /project/lib/../lib///math.sol 在 VFS 中被认为是完全不同的，但它们在磁盘上指向的是同一个文件。

备注： 即使一个导入回调最终从磁盘上的同一个文件中加载了两个不同的源单元名称的源代码，编译器仍然会将它们视为独立的源单元。重要的是源单元名称，而不是代码的物理位置。

相对导入

以 ./ 或 ./ 开头的导入是一个 相对导入。这种导入指定了一个相对于导入源单元的源单元名称的路径。

列表 2: /project/lib/math.sol

```
import "./util.sol" as util;    // 源单元名称： /project/lib/util.sol
import "../token.sol" as token; // 源单元名称： /project/token.sol
```

列表 3: lib/math.sol

```
import "./util.sol" as util;    // 源单元名称: lib/util.sol
import "../token.sol" as token; // 源单元名称: token.sol
```

备注： 相对导入 **总是**以 ./ 或 ../ 开始，所以与 import "./util.sol" 不同，import "util.sol" 是一个直接导入。虽然这两个路径在主机文件系统中被认为是相对的，但 util.sol 在 VFS 中实际上是绝对的。

让我们把 路径段定义为路径中不包含分隔符的任何非空部分，并以两个路径分隔符为界。分隔符是一个正斜杠或字符串的开头/结尾。例如，在 ./abc/..// 中，有三个路径段。.，abc 和 ..。

编译器根据导入路径将导入解析为一个源单元名称，方法如下：

1. 我们从导入源单元的源单元名称开始。
2. 最后一个带有斜线的路径段将从解析的名称中删除。
3. 然后，对于导入路径中的每一段，从最左边的一段开始：
   - 如果该段是 .，则跳过。
   - 如果该段是 ..，最后一个带有斜线的路径段将从解析的名称中删除。
   - 否则，该段（如果解析的名称不是空的，前面有一个单斜线）被附加到解析的名称上。

删除前面有斜线的最后一个路径段，可以理解为工作原理如下：

1. 超过最后一个斜线的所有内容都被删除（即 a/b//c.sol 变成 a/b//）。
2. 所有的尾部斜线被删除（即 a/b// 变成 a/b）。

请注意，该过程根据 UNIX 路径的通常规则对解析的源单元名称中来自导入路径的部分进行了规范化处理，即所有的 . 和 .. 被删除，多个斜线被压成一个。另一方面，来自导入模块的源单元名称的部分仍未被规范化。这确保了在导入文件被识别为 URL 时，protocol:// 部分不会变成 protocol:/。

如果导入路径已经规范化，则可以期望上述算法产生非常直观的结果。下面是一些例子，告诉您如果不是的话会发生什么：

列表 4: lib/src/../contract.sol

```
import "./util/./util.sol";         // 源单元名称: lib/src/../util/util.sol
import "./util//util.sol";          // 源单元名称: lib/src/../util/util.sol
import "../util/../array/util.sol"; // 源单元名称: lib/src/array/util.sol
import "../.././../util.sol";       // 源单元名称: util.sol
import "../../.././../util.sol";    // 源单元名称: util.sol
```

备注： 不建议使用使用包含前缀 `..` 的路径段。通过使用带有基本路径和包含路径 的直接导入，可以以更可靠的方式实现同样的效果。

3.33.3 基本路径和包含路径

基本路径和包含路径表示主机文件系统加载器将加载文件的目录。当一个源单元的名字被传递给加载器时，它把基本路径加到它的前面，并执行一个文件系统查找。如果查找不成功，也会对包含路径列表中的所有目录进行同样的处理。

建议将基本路径设置为您项目的根目录，并使用包含路径来指定可能包含您项目所依赖的库的其他位置。这可以让您以统一的方式从这些库中导入，无论它们在文件系统中相对于您的项目位于何处。例如，如果您使用 npm 安装包，而您的合约导入了 `@openzeppelin/contracts/utils/Strings.sol`，您可以使用这些选项来告诉编译器，该库可以在 npm 包目录中找到。

```
solc contract.sol \
    --base-path . \
    --include-path node_modules/ \
    --include-path /usr/local/lib/node_modules/
```

无论您是把库安装在本地还是全局包目录下，甚至直接安装在您的项目根目录下，您的合约都会被编译（具有完全相同的元数据）。

默认情况下，基本路径是空的，这使得源单元的名称没有变化。当源单元名称是一个相对路径时，这将导致文件在编译器被调用的目录中被查找。这也是唯一能使源单元名称中的绝对路径被实际解释为磁盘上的绝对路径的值。如果基本路径本身是相对的，则它被解释为相对于编译器的当前工作目录。

备注： 包含路径不能有空值，必须与非空的基本路径一起使用。

备注： 只要不使导入解析产生歧义，包含路径和基本路径可以重合。例如，您可以在基本路径内指定一个目录作为包含目录，或者有一个包含目录是另一个包含目录的子目录。只有传递给主机文件系统加载器的源单元名称在与多个包含路径或包含路径和基本路径结合代表一个现有路径时，编译器才会发出错误。

CLI 路径规范化和剥离

在命令行中，编译器的行为就像您对其他程序的期望一样：它接受平台的本地格式的路径，相对路径是相对于当前工作目录的。然而，分配给在命令行上指定了路径的文件的源单元名称，不应该因为项目在不同的平台上被编译，或者因为编译器碰巧从不同的目录被调用而改变。为了达到这个目的，来自命令行的源文件的路径必须被转换为规范的形式，如果可能的话，应使其与基本路径或包含路径之一相对。

规范化规则如下：

- 如果一个路径是相对路径，则通过在其前面加上当前工作目录使其成为绝对路径。
- 内部的 `.` 和 `..` 段被折叠起来。
- 平台特定的路径分隔符被替换为正斜杠。
- 多个连续路径分隔符的序列被压缩成一个分隔符（除非它们是 UNC 路径 的前导斜杠）。
- 如果路径中包含一个根名（例如 Windows 系统中的一个盘符），并且该根名与当前工作目录的根名相同，则根名将被替换为 `/`。
- 路径中的符号链接 **没有**解析。
  - 唯一的例外是在使相对路径成为绝对路径的过程中，对当前工作目录的路径进行了预处理。在一些平台上，工作目录总是用带有符号链接的解析来声明，所以为了保持一致性，编译器在任何地方都会解析它们。
- 即使文件系统对大小写不敏感，但 保留大小写 和磁盘上的实际大小写不同，是会保留路径的原始大小写。

备注： 有些情况下，路径不能独立于平台。例如，在 Windows 下，编译器可以通过将当前驱动器的根目录称为 `/` 来避免使用驱动器字母，但对于通往其他驱动器的路径来说，驱动器字母仍然是必要的。您可以通过确保所有的文件都在同一驱动器上的单一目录树内，来避免这种情况。

在规范化之后，编译器试图使源文件的路径变成相对的。它首先尝试基本路径，然后按照给出的顺序尝试包含路径。如果基本路径是空的或者没有指定，它将被视为等同于当前工作目录的路径（解决了所有符号链接）。只有当规范化的目录路径是规范化的文件路径的确切前缀时，才会接受这个结果。否则，文件路径仍然是绝对的。这使得转换毫不含糊，并确保相对路径不以 `../` 开头。产生的文件路径成为源单元名称。

备注： 剥离后产生的相对路径必须在基本路径和包含路径中保持唯一。例如，如果 `/project/contract.sol` 和 `/lib/contract.sol` 同时存在，编译器将对以下命令发出错误：

```
solc /project/contract.sol --base-path /project --include-path /lib
```

备注： 在 0.8.8 版本之前，CLI 路径剥离不被执行，唯一应用的规范化是路径分隔符的转换。当使用旧版本的

编译器时，建议从基本路径调用编译器，在命令行上只使用相对路径。

3.33.4 允许的路径

作为一项安全措施，主机文件系统加载器将拒绝从默认认为安全的几个位置之外的地方加载文件：

- 标准 JSON 模式之外：
  - 含有命令行上所列输入文件的目录。
  - 作为重映射 目标使用的目录。如果目标不是一个目录（即不以 `/`，`/.` 或 `/.` 结尾），则使用包含该目标的目录。
  - 基本路径和包含路径。
- 在标准 JSON 模式下：
  - 基本路径和包含路径。

可以使用 `--allow-paths` 选项将其他目录列入白名单。该选项接受一个用逗号分隔的路径列表：

```
cd /home/user/project/
solc token/contract.sol \
    lib/util.sol=libs/util.sol \
    --base-path=token/ \
    --include-path=/lib/ \
    --allow-paths=../utils/,/tmp/libraries
```

当用上面的命令调用编译器时，主机文件系统加载器将允许从以下目录导入文件：

- `/home/user/project/token/` （因为 `token/` 包含输入文件，也因为它是基本路径），
- `/lib/` （因为 `/lib/` 是包含路径之一），
- `/home/user/project/libs/` （因为 `libs/` 是一个包含重映射目标的目录），
- `/home/user/utils/` （因为 `.../utils/` 传给了 `-allow-paths` ），
- `/tmp/libraries/` （因为 `/tmp/libraries` 被传递到 `/tmp/libraries`），

备注： 编译器的工作目录是默认允许的路径之一，前提是它恰好是基本路径时（或者基本路径没有被指定或有一个空值）。

备注： 编译器不检查允许的路径是否真实存在以及它们是否是目录。不存在的或空的路径会被简单地忽略掉。如果一个被允许的路径与一个文件而不是一个目录相匹配，该文件也被视为白名单。

备注： 允许的路径是区分大小写的，即使文件系统不是这样的。大小写必须与您的导入中使用的大小写完全一致。例如 `--allow-paths tokens` 不会匹配 `import "Tokens/IERC20.sol"`。

警告： 只有通过允许的目录的符号链接才能到达的文件和目录不会被自动列入白名单。例如，如果上面的例子中的 `token/contract.sol` 实际上是一个指向 `/etc/passwd` 的符号链接，编译器将拒绝加载它，除非 `/etc/` 也是允许的路径之一。

3.33.5 导入重映射

导入重映射允许您将导入重定向到虚拟文件系统的不同位置。该机制通过改变导入路径和源单元名称之间的转换来工作。例如，您可以设置一个重映射，使任何从虚拟目录 `github.com/ethereum/dapp-bin/library/` 的导入被视为从 `dapp-bin/library/` 导入。

您可以通过指定 *context* 来限制重映射的范围。这允许创建仅适用于特定库或特定文件中的导入的重映射。如果没有 context 关键字指定，重映射将应用于虚拟文件系统中所有文件中的每个匹配的导入。

导入重映射的形式为 `context:prefix=target`：

- `context` 必须与包含导入文件的源单元名称的开头相匹配。
- `prefix` 必须与导入的源单元名称的开头相匹配。
- `target` 是前缀被替换的值。

例如，如果您在本地克隆 https://github.com/ethereum/dapp-bin/ 到 `/project/dapp-bin`，并用以下命令运行编译器：

```
solc github.com/ethereum/dapp-bin/=dapp-bin/ --base-path /project source.sol
```

您可以在您的源文件中使用以下内容：

```
import "github.com/ethereum/dapp-bin/library/math.sol"; // 源单元名称： dapp-bin/
↪library/math.sol
```

编译器将在 VFS 的 `dapp bin/library/math.sol` 下寻找该文件。如果那里没有该文件，源单元名称将被传递给主机文件系统加载器，然后它将在 `/project/dapp-bin/library/iterable_mapping.sol` 中寻找。

警告： 关于重映射的信息被存储在合约元数据中。由于编译器产生的二进制文件中嵌入了元数据的哈希值，对重映射的任何修改都会导致不同的字节码。

> 由于这个原因，您应该注意不要在重映射目标中包含任何本地信息。例如，如果您的库位于 /home/user/packages/mymath/math.sol，像 @math/=/home/user/packages/mymath/ 这样的重映射会导致您的主目录被包含在元数据中。为了能够在不同的机器上用这样的重映射重现相同的字节码，您需要在 VFS 和（如果您依赖主机文件系统加载器）主机文件系统中重新创建您的本地目录结构。
>
> 为了避免元数据中嵌入您的本地目录结构，建议将包含库的目录指定为 *include paths*。例如，在上面的例子中，--include-path /home/user/packages/ 会让您使用以 mymath/ 开始的导入。与重映射不同，该选项本身不会使 mymath 显示为 @math，但这可以通过创建符号链接或重命名软件包子目录来实现。

作为一个更复杂的例子，假设您依赖一个使用旧版 dapp-bin 的模块，您把它签出到 /project/dapp-bin_old，那么您可以运行：

```
solc module1:github.com/ethereum/dapp-bin/=dapp-bin/ \
     module2:github.com/ethereum/dapp-bin/=dapp-bin_old/ \
     --base-path /project \
     source.sol
```

这意味着 module2 的所有导入都指向旧版本，但 module1 的导入则指向新版本。

以下是关于重映射行为的详细规则：

1. **重新映射只影响导入路径和源单元名称之间的转换。**

   以任何其他方式添加到 VFS 的源单元名称不能被重新映射。例如，您在命令行上指定的路径和标准 JSON 中 sources.urls 中的路径不受影响。

   ```
   solc /project/=/contracts/ /project/contract.sol # 源单元名称： /project/contract.
   ↪sol
   ```

   在上面的例子中，编译器将从 /project/contract.sol 中加载源代码，并将其放在 VFS 中那个确切的源代码单元名下，而不是放在 /contract/contract.sol 中。

2. **上下文和前缀必须与源单元名称相匹配，而不是导入路径。**

   - 这意味着您不能直接重新映射 ./ 或 ./，因为它们在转译成源单元名称时被替换了，但您可以重新映射它们被替换的那部分名称：

     ```
     solc ./=a/ /project/=b/ /project/contract.sol # 源单元名称： /project/
     ↪contract.sol
     ```

     列表 5: /project/contract.sol

     ```
     import "./util.sol" as util; // 源单元名称： b/util.sol
     ```

   - 您不能重新映射基本路径或仅由导入回调内部添加的任何其他部分的路径。

```
solc /project/=/contracts/ /project/contract.sol --base-path /project # 源单元名称： contract.sol
```

列表 6: /project/contract.sol

```
import "util.sol" as util; // 源单元名称： util.sol
```

3. **目标直接插入源单元名称中，不一定是有效的路径。**

   - 只要导入回调能够处理它，它可以是任何东西。在主机文件系统加载器的情况下，这也包括相对路径。当使用 JavaScript 接口时，您甚至可以使用 URL 和抽象标识符，如果您的回调能够处理它们。
   - 重映射发生在相对导入已经被解析为源单元名称之后。这意味着以 `./` 和 `./` 开头的目标没有特殊含义，是相对于基本路径而不是源文件的位置。
   - 重映射目标没有被规范化，所以 `@root/=./a/b//` 将重映射 `@root/contract.sol` 到 `./a/b/contract.sol` 而不是 `a/b/contract.sol`。
   - 如果目标不以斜线结尾，编译器将不会自动添加一个斜线:

```
solc /project/=/contracts /project/contract.sol # 源单元名称： /project/contract.sol
```

列表 7: /project/contract.sol

```
import "/project/util.sol" as util; // 源单元名称： /contractsutil.sol
```

4. **上下文和前缀是匹配模式，匹配必须是精确的。**

   - `a//b=c` 不会匹配 `a/b`。
   - 源单元名称没有被规范化，所以 `a/b=c` 也不会匹配 `a//b`。
   - 文件和目录的部分名称是可以匹配。`/newProject/con:/new=old` 将匹配 `/newProject/contract.sol` 并将其重新映射到 `oldProject/contract.sol`。

5. **最多只有一个重映射被应用于单个导入。**

   - 如果多个重映射与同一个源单元名称相匹配，则选择具有最长匹配前缀的那个。
   - 如果前缀相同，则选择最后指定的那个。
   - 重映射对其他重映射不起作用。例如 `a=b b=c c=d` 不会导致 `a` 被重映射到 `d`。

6. **prefix 不能为空，但 context 和 target 是可选的。**

   - 如果 `target` 是空字符串，`prefix` 将从导入路径中删除。
   - 空的 `context` 意味着重新映射适用于所有源单元中的所有导入。

3.33.6 在导入中使用 url

大多数 URL 前缀，如 `https://` 或 `data://` 在导入路径中没有特殊含义。唯一的例外是 `file://`，它被主机文件系统加载器从源单元名称中剥离出来。

在本地编译时，您可以使用导入重映射，用本地路径替换协议和域名部分：

```
solc :https://github.com/ethereum/dapp-bin=/usr/local/dapp-bin contract.sol
```

注意前面的 `:`，当重映射上下文为空时，这是必要的。否则，`https:` 部分将被编译器解释为上下文。

3.34 风格指南

3.34.1 概述

本指南旨在为编写 Solidity 代码提供编码规范。这个指南应该被认为是一个不断发展的文件，随着有用的约定被发现和旧的约定被淘汰，它将随着时间而改变。

许多项目会实施他们自己的编码风格指南。如遇冲突，应优先使用具体项目的风格指南。

本风格指南中的结构和许多建议是取自 Python 的 pep8 风格指南 。

本指南并 *不是* 以指导正确或最佳的 solidity 编码方式为目的。本指南的目的是保持代码的 *一致性*。来自 Python 的参考文档 pep8，很好地阐述了这个概念。

备注： 风格指南是关于一致性的。重要的是与此风格指南保持一致，但项目中的一致性更重要。一个模块或功能内的一致性是最重要的。

但最重要的是：**知道什么时候不一致**——有时风格指南不适用。如有疑问，请自行判断。看看其他例子，并决定什么看起来最好，并应毫不犹豫地询问他人！

3.34.2 代码结构

缩进

每个缩进级别使用 4 个空格。

制表符或空格

空格是首选的缩进方法。

应该避免混合使用制表符和空格。

空行

在 solidity 源码中合约声明之间留出两个空行。

正确写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}


contract B {
    // ...
}


contract C {
    // ...
}
```

错误写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
}
contract B {
    // ...
}

contract C {
    // ...
}
```

在一个合约中的函数声明之间留有一个空行。

在相关联的各组单行语句之间可以省略空行。(例如抽象合约的 stub 函数)。

正确写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() public virtual pure;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }

    function ham() public pure override {
        // ...
    }
}
```

错误写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.0 <0.9.0;

abstract contract A {
    function spam() virtual pure public;
    function ham() public virtual pure;
}


contract B is A {
    function spam() public pure override {
        // ...
    }
    function ham() public pure override {
        // ...
    }
}
```

代码行的最大长度

最大建议行长度为 120 个字符。

折行时应该遵从以下指引。

1. 第一个参数不应该紧跟在左括号后边
2. 用一个，且只用一个缩进
3. 每个函数应该单起一行
4. 结束符号); 应该单独放在最后一行

函数调用

正确写法：

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

错误写法：

```
thisFunctionCallIsReallyLong(longArgument1,
                              longArgument2,
                              longArgument3
);

thisFunctionCallIsReallyLong(longArgument1,
    longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
    longArgument1, longArgument2,
    longArgument3
);

thisFunctionCallIsReallyLong(
longArgument1,
longArgument2,
longArgument3
);

thisFunctionCallIsReallyLong(
```

(续下页)

(接上页)

```
    longArgument1,
    longArgument2,
    longArgument3);
```

赋值语句

正确写法:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(
    argument1,
    argument2,
    argument3,
    argument4
);
```

错误写法:

```
thisIsALongNestedMapping[being][set][toSomeValue] = someFunction(argument1,
                                                                   argument2,
                                                                   argument3,
                                                                   argument4);
```

事件定义和事件发生

正确写法:

```
event LongAndLotsOfArgs(
    address sender,
    address recipient,
    uint256 publicKey,
    uint256 amount,
    bytes32[] options
);

LongAndLotsOfArgs(
    sender,
    recipient,
    publicKey,
    amount,
    options
);
```

错误写法:

```
event LongAndLotsOfArgs(address sender,
                        address recipient,
                        uint256 publicKey,
                        uint256 amount,
                        bytes32[] options);

LongAndLotsOfArgs(sender,
                  recipient,
                  publicKey,
                  amount,
                  options);
```

源文件编码格式

首选 UTF-8 或 ASCII 编码。

Imports 规范

Import 语句应始终放在文件的顶部。

正确写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";

contract A {
    // ...
}


contract B is Owned {
    // ...
}
```

错误写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract A {
    // ...
```

(续下页)

(接上页)

```
}


import "./Owned.sol";


contract B is Owned {
    // ...
}
```

函数顺序

排序有助于读者识别他们可以调用哪些函数，并更容易地找到构造函数和 fallback 函数的定义。

函数应根据其可见性和顺序进行分组：

- 构造函数
- receive 函数（如果存在）
- fallback 函数（如果存在）
- 外部函数
- 公共函数
- 内部函数
- 私有函数

在一个分组中，把 `view` 和 `pure` 函数放在最后。

正确写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {
    constructor() {
        // ...
    }

    receive() external payable {
        // ...
    }

    fallback() external {
        // ...
```

(续下页)

(接上页)

```
    }

    // 外部函数
    // ...

    // 是 view 修饰的外部函数
    // ...

    // 是 pure 修饰的外部函数
    // ...

    // 公共函数
    // ...

    // 内部函数
    // ...

    // 私有函数
    // ...
}
```

错误写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract A {

    // 外部函数
    // ...

    fallback() external {
        // ...
    }
    receive() external payable {
        // ...
    }

    // 私有函数
    // ...

    // 公共函数
    // ...

    constructor() {
```

(续下页)

(接上页)

```
        // ...
    }

    // 内部函数
    // ...
}
```

表达式中的空格

在以下情况下避免无关的空格:

除单行函数声明外，紧接着小括号，中括号或者大括号的内容应该避免使用空格。

正确写法:

```
spam(ham[1], Coin({name: "ham"}));
```

错误写法:

```
spam( ham[ 1 ], Coin( { name: "ham" } ) );
```

除外:

```
function singleLine() public { spam(); }
```

紧接在逗号，分号之前:

正确写法:

```
function spam(uint i, Coin coin) public;
```

错误写法:

```
function spam(uint i , Coin coin) public ;
```

赋值或其他操作符两边多于一个的空格:

正确写法:

```
x = 1;
y = 2;
longVariable = 3;
```

错误写法:

```
x            = 1;
y            = 2;
longVariable = 3;
```

在 receive 和 fallback 函数中不要包含空格:

正确写法:

```
receive() external payable {
    ...
}

fallback() external {
    ...
}
```

错误写法:

```
receive () external payable {
    ...
}

fallback () external {
    ...
}
```

控制结构

用大括号表示一个合约，库，函数和结构。应该为:

- 开括号与声明应在同一行。
- 闭括号在与之前函数声明对应的开括号保持同一缩进级别上另起一行。
- 开括号前应该有一个空格。

正确写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin {
    struct Bank {
        address owner;
        uint balance;
```

(续下页)

(接上页)

```
    }
}
```

错误写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract Coin
{
    struct Bank {
        address owner;
        uint balance;
    }
}
```

对于控制结构 if，else，while，和 for 的实施建议与以上相同。

另外，诸如 if，else，while，和 for 这类的控制结构和条件表达式的块之间应该有一个单独的空格，同样的，条件表达式的块和开括号之间也应该有一个空格。

正确写法:

```
if (...) {
    ...
}

for (...) {
    ...
}
```

错误写法:

```
if (...)
{
    ...
}

while(...){
}

for (...) {
    ...;}
```

对于控制结构，如果其主体内容只包含一行，则可以省略括号。

正确写法:

```
if (x < 10)
    x += 1;
```

错误写法:

```
if (x < 10)
    someArray.push(Coin({
        name: 'spam',
        value: 42
    }));
```

对于具有 else 或 else if 子句的 if 块，else 应该是与 if 的闭大括号放在同一行上。这一规则区别于其他块状结构。

正确写法:

```
if (x < 3) {
    x += 1;
} else if (x > 7) {
    x -= 1;
} else {
    x = 5;
}


if (x < 3)
    x += 1;
else
    x -= 1;
```

错误写法:

```
if (x < 3) {
    x += 1;
}
else {
    x -= 1;
}
```

函数声明

对于简短的函数声明，建议函数体的开括号与函数声明保持在同一行。

闭大括号应该与函数声明的缩进级别相同。

开大括号之前应该有一个空格。

正确写法：

```
function increment(uint x) public pure returns (uint) {
    return x + 1;
}

function increment(uint x) public pure onlyOwner returns (uint) {
    return x + 1;
}
```

错误写法：

```
function increment(uint x) public pure returns (uint)
{
    return x + 1;
}

function increment(uint x) public pure returns (uint){
    return x + 1;
}

function increment(uint x) public pure returns (uint) {
    return x + 1;
    }

function increment(uint x) public pure returns (uint) {
    return x + 1;}
```

一个函数的修饰符顺序应该是：

1. 可见性
2. 可变性
3. 虚拟性
4. 覆盖性
5. 自定义修饰符

正确写法：

```
function balance(uint from) public view override returns (uint)  {
    return balanceOf[from];
}

function shutdown() public onlyOwner {
    selfdestruct(owner);
}
```

错误写法:

```
function balance(uint from) public override view returns (uint)  {
    return balanceOf[from];
}

function shutdown() onlyOwner public {
    selfdestruct(owner);
}
```

对于长的函数声明，建议将每个参数放在自己的行中，与函数主体的缩进程度相同。闭小括号和开括号也应该放在自己的行中，与函数声明的缩进程度相同。

正确写法:

```
function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f
)
    public
{
    doSomething();
}
```

错误写法:

```
function thisFunctionHasLotsOfArguments(address a, address b, address c,
    address d, address e, address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(address a,
                                        address b,
```

(续下页)

(接上页)

```
                                                  address c,
                                                  address d,
                                                  address e,
                                                  address f) public {
    doSomething();
}

function thisFunctionHasLotsOfArguments(
    address a,
    address b,
    address c,
    address d,
    address e,
    address f) public {
    doSomething();
}
```

如果一个长函数声明有修饰符，那么每个修饰符都应该被丢到独立的一行。

正确写法：

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(
    address x,
    address y,
    address z
)
    public
    onlyOwner
    priced
    returns (address)
{
    doSomething();
}
```

错误写法:

```
function thisFunctionNameIsReallyLong(address x, address y, address z)
                                      public
                                      onlyOwner
                                      priced
                                      returns (address) {
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public onlyOwner priced returns (address)
{
    doSomething();
}

function thisFunctionNameIsReallyLong(address x, address y, address z)
    public
    onlyOwner
    priced
    returns (address) {
    doSomething();
}
```

多行输出参数和返回值语句应该遵从代码行的最大长度 一节的说明。

正确写法:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (
        address someAddressName,
        uint256 LongArgument,
        uint256 Argument
    )
{
    doSomething()

    return (
        veryLongReturnArg1,
        veryLongReturnArg2,
        veryLongReturnArg3
    );
```

(续下页)

(接上页)

```
}
```

错误写法:

```
function thisFunctionNameIsReallyLong(
    address a,
    address b,
    address c
)
    public
    returns (address someAddressName,
             uint256 LongArgument,
             uint256 Argument)
{
    doSomething()

    return (veryLongReturnArg1,
            veryLongReturnArg1,
            veryLongReturnArg1);
}
```

对于继承合约中需要参数的构造函数，如果函数声明很长或难以阅读，建议将基础构造函数像多个修饰符的风格那样，每个下沉到一个新行上书写。

正确写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
// 基础合约，为了使这段代码能被编译
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
```

(续下页)

(接上页)

```
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4)
    {
        // 用参数 param5 做一些事情
        x = param5;
    }
}
```

错误写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// 基础合约，为了使这段代码能被编译
contract B {
    constructor(uint) {
    }
}


contract C {
    constructor(uint, uint) {
    }
}


contract D {
    constructor(uint) {
    }
}


contract A is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
```

(续下页)

(接上页)

```
    B(param1)
    C(param2, param3)
    D(param4) {
        x = param5;
    }
}


contract X is B, C, D {
    uint x;

    constructor(uint param1, uint param2, uint param3, uint param4, uint param5)
        B(param1)
        C(param2, param3)
        D(param4) {
            x = param5;
        }
}
```

当用单个语句声明简短函数时，允许在一行中完成。

允许：

```
function shortFunction() public { doSomething(); }
```

这些函数声明的准则旨在提高可读性。因为本指南不会涵盖所有内容，作者应该自行作出最佳判断。

映射

在变量声明中，不要用空格将关键字 mapping 和其类型分开。不要用空格分隔任何嵌套的 mapping 关键词和其类型。

正确写法：

```
mapping(uint => uint) map;
mapping(address => bool) registeredAddresses;
mapping(uint => mapping(bool => Data[])) public data;
mapping(uint => mapping(uint => s)) data;
```

错误写法:

```
mapping (uint => uint) map;
mapping( address => bool ) registeredAddresses;
mapping (uint => mapping (bool => Data[])) public data;
mapping(uint => mapping (uint => s)) data;
```

变量声明

数组变量的声明在变量类型和括号之间不应该有空格。

正确写法：

```
uint[] x;
```

错误写法:

```
uint [] x;
```

其他建议

- 字符串应该用双引号而不是单引号。

正确写法：

```
str = "foo";
str = "Hamlet says, 'To be or not to be...'";
```

错误写法:

```
str = 'bar';
str = '"Be yourself; everyone else is already taken." -Oscar Wilde';
```

- 操作符两边应该各有一个空格。

正确写法：

```
x = 3;
x = 100 / 10;
x += 3 + 4;
x |= y && z;
```

错误写法:

```
x=3;
x = 100/10;
x += 3+4;
x |= y&&z;
```

- 为了表示优先级，高优先级操作符两边可以省略空格。这样可以提高复杂语句的可读性。您应该在操作符两边总是使用相同的空格数：

正确写法：

```
x = 2**3 + 5;
x = 2*y + 3*z;
x = (a+b) * (a-b);
```

错误写法:

```
x = 2** 3 + 5;
x = y+z;
x +=1;
```

3.34.3 布局顺序

按以下顺序布置合约的元素：

1. Pragma 语句
2. 导入语句
3. 接口
4. 库
5. 合约

在每个合约，库或接口内，使用以下顺序：

1. 类型声明
2. 状态变量
3. 事件
4. 错误
5. 修饰符
6. 函数

备注： 在接近事件或状态变量的使用时，声明类型可能会更清楚。

正确写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    error DivideByZero();
    function divide(int256 numerator, int256 denominator) public virtual returns␣
```

(续下页)

(接上页)

```
↪(uint256);
}
```

错误写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.4 <0.9.0;

abstract contract Math {
    function divide(int256 numerator, int256 denominator) public virtual returns␣
↪(uint256);
    error DivideByZero();
}
```

3.34.4 命名规范

当完全采纳和使用命名规范时会产生强大的作用。当使用不同的规范时，则不会立即获取代码中传达的重要元信息。

这里给出的命名建议旨在提高可读性，因此它们不是规则，而是透过名称来尝试和帮助传达最多的信息。

最后，基于代码库中的一致性，本文档中的任何规范总是可以被（代码库中的规范）取代。

命名方式

为了避免混淆，下面的名字用来指明不同的命名方式。

- `b` （单个小写字母）
- `B` （单个大写字母）
- `lowercase` （小写）
- `UPPERCASE` （大写）
- `UPPER_CASE_WITH_UNDERSCORES` （大写和下划线）
- `CapitalizedWords` （驼峰式，首字母大写）
- `mixedCase` （混合式，与驼峰式的区别在于首字母小写!）

备注: 当在驼峰式命名中使用缩写时，应该将缩写中的所有字母都大写。因此 HTTPServerError 比 HttpServerError 好。当在混合式命名中使用缩写时，除了第一个缩写中的字母小写（如果它是整个名称的开头的话）以外，其他缩写中的字母均大写。因此 xmlHTTPRequest 比 XMLHTTPRequest 更好。

应避免的名称

- l - el 的小写方式
- O - oh 的大写方式
- I - eye 的大写方式

切勿将任何这些用于单个字母的变量名称。他们经常难以与数字 1 和 0 区分开。

合约和库名称

- 合约和库名称应该使用驼峰式风格。比如：SimpleToken，SmartBank，CertificateHashRepository，Player，Congress，Owned.。
- 合约和库的名称也应与它们的文件名相符。
- 如果一个合约文件包括多个合约和/或库，那么文件名应该与 核心合约相匹配。但是，如果可以避免的话，不建议这样做。

如下面的例子所示，如果合约名称是 Congress，库名称是 Owned，那么它们的相关文件名应该是 Congress.sol 和 Owned.sol。

正确写法：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// Owned.sol
contract Owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

在 Congress.sol 合约里：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

import "./Owned.sol";


contract Congress is Owned, TokenRecipient {
    //...
}
```

错误写法:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

// owned.sol
contract owned {
    address public owner;

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}
```

在 Congress.sol 合约里:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;


import "./owned.sol";


contract Congress is owned, tokenRecipient {
    //...
}
```

结构体名称

结构体名称应该使用驼峰式风格。比如：MyCoin，Position，PositionXY。

事件名称

事件名称应该使用驼峰式风格。比如：Deposit，Transfer，Approval，BeforeTransfer，AfterTransfer。

函数名称

函数名称应该使用混合式命名风格。比如：getBalance，transfer，verifyOwner，addMember，changeOwner。

函数参数命名

函数参数命名应该使用混合式命名风格。比如：initialSupply，account，recipientAddress，senderAddress，newOwner。

在编写操作自定义结构的库函数时，这个结构体应该作为函数的第一个参数，并且应该始终命名为 self。

局部变量和状态变量名称

使用混合式命名风格。比如：totalSupply，remainingSupply，balancesOf，creatorAddress，isPreSale，tokenExchangeRate。

常量命名

常量应该全都使用大写字母书写，并用下划线分割单词。比如：MAX_BLOCKS，TOKEN_NAME，TOKEN_TICKER，CONTRACT_VERSION。

修饰符命名

使用混合式命名风格。比如：onlyBy，onlyAfter，onlyDuringThePreSale。

枚举变量命名

在声明简单类型时，枚举应该使用驼峰式风格。比如：TokenGroup，Frame，HashStyle，CharacterLocation。

避免命名冲突

- singleTrailingUnderscore_

当所需的名称与现有状态变量，函数，内置或其他保留关键字名称冲突时，建议使用此约定。

非外部函数和变量的下划线前缀

- _singleLeadingUnderscore

建议对非外部函数和状态变量（private 或 internal）使用此约定。默认情况下，没有指定可见性的状态变量是 internal。

在设计智能合约时，面向公众的 API（任何账户都可以调用的函数）是一个重要的考虑因素。前导下划线允许您立即识别此类函数的意图，但更重要的是，如果您将函数从非外部函数更改为外部函数（包括 public）并相应地重命名，这将迫使您在重命名时审查每个调用栈。这可能是针对非预期外部函数的重要手动检查，也是安全漏洞的常见来源（避免使用查找-替换-全部工具来进行此更改）。

3.34.5 NatSpec

Solidity 合约也可以包含 NatSpec 注释。它们用三重斜线（///）或双星号块（/** ... */）来写，它们应该直接用在函数声明或语句之上。

例如，来自一个简单的智能合约 的合约在添加了注释后看起来就像下面这个：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

/// @author Solidity团队
/// @title 一个简单的存储例子
contract SimpleStorage {
    uint storedData;

    /// 存储 `x`。
    /// @param x 要存储的新值
    /// @dev 将数字存储在状态变量 `storedData` 中
    function set(uint x) public {
        storedData = x;
    }
```

(续下页)

(接上页)

```
    /// 返回存储的值。
    /// @dev 检索状态变量 `storedData` 的值
    /// @return 存储的值
    function get() public view returns (uint) {
        return storedData;
    }
}
```

建议 Solidity 合约使用*NatSpec* 对所有公共接口（ABI 中的一切）进行完全注释。

请参阅关于*NatSpec* 的部分，以获得详细解释。

3.35 通用模式

3.35.1 从合约中提款

在某个操作之后发送资金的推荐方式是使用取回（withdrawal）模式。尽管在某个操作之后，最直接地发送以太币方法是一个 `transfer` 调用，但这并不推荐, 因为这会引入一个潜在的安全风险。您可能需要参考安全考虑 来获取更多信息。

下面是一个合约中实际提款模式的例子，其目标是向合约发送最多的钱，以成为"首富"，其灵感来自于 King of the Ether。

在下面的合约中，如果您不再是最富有的人，您将收到取代您成为"最富有"的人发送到合约的资金。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract WithdrawalContract {
    address public richest;
    uint public mostSent;

    mapping(address => uint) pendingWithdrawals;

    /// 发送的以太数量不高于目前的最高量。
    error NotEnoughEther();

    constructor() payable {
        richest = msg.sender;
        mostSent = msg.value;
    }
```

(续下页)

(接上页)

```
    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        pendingWithdrawals[richest] += msg.value;
        richest = msg.sender;
        mostSent = msg.value;
    }

    function withdraw() public {
        uint amount = pendingWithdrawals[msg.sender];
        // 记得在发送前将待处理的退款归零，
        // 以防止重入攻击
        pendingWithdrawals[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

下面是一个相反的直接使用发送模式的例子：

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract SendContract {
    address payable public richest;
    uint public mostSent;

    /// 发送的以太数量不高于目前的最高量。
    error NotEnoughEther();

    constructor() payable {
        richest = payable(msg.sender);
        mostSent = msg.value;
    }

    function becomeRichest() public payable {
        if (msg.value <= mostSent) revert NotEnoughEther();
        // 这一行会导致问题（详见下文）
        richest.transfer(msg.value);
        richest = payable(msg.sender);
        mostSent = msg.value;
    }
}
```

请注意，在这个例子中，攻击者可以通过使 `richest` 成为一个有 receive 或 fallback 函数的合约的地址而使合约陷入无法使用的状态（例如，通过使用 `revert()` 或只是消耗超过转给他们的 2300 gas 津贴）。这样，每

当调用 `transfer` 向"中毒"的合约交付资金时，它就会失败，因此 `becomeRichest` 也会失败，合约会永远被卡住。

相反，如果您使用第一个例子中的"取回（withdraw）"模式，那么攻击者只能使他/她自己的"取回"失败，并不会导致整个合约无法运作。

3.35.2 限制访问

限制访问是合约的一个常见模式。请注意，您永远无法限制任何人类或机器阅读您的交易内容或您的合约状态。您可以通过使用加密来增加一点难度，但如果您想让您的合约读取这些数据，那么其他人也将可以做到。

您可以限制 **其他合约**对您的合约状态的读取权限。这实际上是默认的，除非您声明您的状态变量为 `public`。

此外，您可以限制谁可以对您的合约的状态进行修改或调用您的合约的功能，这就是本节的内容。

使用 **函数修饰符**使这些限制变得非常明确。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract AccessRestriction {
    // 这些将在构造阶段被赋值
    // 其中，`msg.sender` 是
    // 创建这个合约的账户。
    address public owner = msg.sender;
    uint public creationTime = block.timestamp;

    // 现在列出了该合约可能产生的错误，
    // 并在特别注释中作了文字解释。

    /// 调用者未被授权进行此操作。
    error Unauthorized();

    /// 函数调用过早。
    error TooEarly();

    /// 函数调用时没有发送足够的以太。
    error NotEnoughEther();

    // 修饰器可以用来更改
    // 一个函数的函数体。
    // 如果使用这个修饰器，
    // 它会预置一个检查，仅允许
    // 来自特定地址的
    // 函数调用。
    modifier onlyBy(address account)
```

(续下页)

(接上页)

```
    {
        if (msg.sender != account)
            revert Unauthorized();
        // 不要忘记写 “_;” !
        // 它会被实际使用这个修饰器的
        // 函数体所替代。
        _;
    }

    /// 使 `newOwner` 成为这个合约的新所有者。
    function changeOwner(address newOwner)
        public
        onlyBy(owner)
    {
        owner = newOwner;
    }

    modifier onlyAfter(uint time) {
        if (block.timestamp < time)
            revert TooEarly();
        _;
    }

    /// 抹掉所有者信息。
    /// 仅允许在合约创建成功 6 周以后
    /// 的时间被调用。
    function disown()
        public
        onlyBy(owner)
        onlyAfter(creationTime + 6 weeks)
    {
        delete owner;
    }

    // 这个修饰器要求对函数调用
    // 绑定一定的费用。
    // 如果调用方发送了过多的费用,
    // 他/她会得到退款, 但需要先执行函数体。
    // 这在 0.4.0 版本以前的 Solidity 中很危险,
    // 因为很可能会跳过 `_;` 之后的代码。
    modifier costs(uint amount) {
        if (msg.value < amount)
            revert NotEnoughEther();
```

(续下页)

(接上页)

```
        _;
        if (msg.value > amount)
            payable(msg.sender).transfer(msg.value - amount);
    }

    function forceOwnerChange(address newOwner)
        public
        payable
        costs(200 ether)
    {
        owner = newOwner;
        // 这只是示例条件
        if (uint160(owner) & 0 == 1)
            // 这无法在 0.4.0 版本之前的
            // Solidity 上进行退还。
            return;
        // 退还多付的费用
    }
}
```

在下一个例子中，将讨论一种更专业的限制函数调用访问的方式。

3.35.3 状态机

合约通常会像状态机那样运作，这意味着它们有特定的 **阶段**，使它们有不同的表现或者仅允许特定的不同函数被调用。一个函数调用通常会结束一个阶段，并将合约转换到下一个阶段（特别是如果一个合约是以 **交互** 来建模的时候）。通过达到特定的 **时间**点来达到某些阶段也是很常见的。

一个典型的例子是盲拍（blind auction）合约，它起始于“接受盲目出价”，然后转换到“公示出价”，最后结束于“确定拍卖结果”。

函数修饰器可以用在这种情况下来对状态进行建模，并确保合约被正常的使用。

示例

在下边的示例中，修饰器 `atStage` 确保了函数仅在特定的阶段才可以被调用。

自动定时过渡是由修饰器 `timedTransitions` 处理的，它应该用于所有函数。

备注：修饰器的顺序非常重要. 如果 atStage 和 timedTransitions 要一起使用，请确保在 timedTransitions 之后声明 atStage，以便新的状态可以首先被反映到账户中。

最后，修饰器 `transitionNext` 能够用来在函数执行结束时自动转换到下一个阶段。

备注： 修饰器可以被忽略. 这只适用于 0.4.0 版本之前的 Solidity：由于修饰器是通过简单地替换代码而不是使用函数调用来应用的，如果函数本身使用 return，可以跳过 transitionNext 修饰器中的代码。如果您想这样做，请确保从这些函数中手动调用 nextStage。从 0.4.0 版本开始，即使函数明确返回，修饰器代码也会运行。

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract StateMachine {
    enum Stages {
        AcceptingBlindedBids,
        RevealBids,
        AnotherStage,
        AreWeDoneYet,
        Finished
    }
    /// 此阶段不能调用该函数。
    error FunctionInvalidAtThisStage();

    // 这是当前阶段。
    Stages public stage = Stages.AcceptingBlindedBids;

    uint public creationTime = block.timestamp;

    modifier atStage(Stages stage_) {
        if (stage != stage_)
            revert FunctionInvalidAtThisStage();
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    // 执行基于时间的阶段转换。
    // 请确保首先声明这个修饰器，
    // 否则新阶段不会被带入账户。
    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindedBids &&
                    block.timestamp >= creationTime + 10 days)
            nextStage();
        if (stage == Stages.RevealBids &&
```

(续下页)

(接上页)

```
                block.timestamp >= creationTime + 12 days)
            nextStage();
        // 由交易触发的其他阶段转换
        _;
    }

    // 这里的修饰器顺序非常重要！
    function bid()
        public
        payable
        timedTransitions
        atStage(Stages.AcceptingBlindedBids)
    {
        // 我们不会在这里实现实际功能（因为这仅是个代码示例，译者注）
    }

    function reveal()
        public
        timedTransitions
        atStage(Stages.RevealBids)
    {
    }

    // 这个修饰器在函数执行结束之后
    // 使合约进入下一个阶段。
    modifier transitionNext()
    {
        _;
        nextStage();
    }

    function g()
        public
        timedTransitions
        atStage(Stages.AnotherStage)
        transitionNext
    {
    }

    function h()
        public
        timedTransitions
        atStage(Stages.AreWeDoneYet)
```

(续下页)

(接上页)

```
        transitionNext
    {
    }

    function i()
        public
        timedTransitions
        atStage(Stages.Finished)
    {
    }
}
```

3.36 资源

3.36.1 一般资源

- Ethereum.org 开发者门户网站
- Ethereum StackExchange
- Solidity 门户网站
- Solidity 变更日志
- GitHub 上的 Solidity 源代码
- Solidity 语言用户聊天室
- Solidity 编译器开发人员聊天室
- 很棒的 Solidity
- 通过实例学 Solidity
- Solidity 文档社区翻译

3.36.2 集成（以太坊）开发环境

- **Brownie**

  面向以太坊虚拟机的基于 Python 的智能合约开发和测试框架。

- **Dapp**

  用于从命令行构建，测试和部署智能合约的工具。

- **Embark**

  构建和部署去中心化应用程序的开发者平台。

- **Foundry**

  用 Rust 编写的用于 Ethereum 应用开发的快速，可移植和模块化的工具包。

- **Hardhat**

  以太坊开发环境具有本地以太坊网络，调试功能和插件生态系统。

- **Remix**

  基于浏览器的 IDE，具有集成的编译器和 Solidity 运行环境，没有服务器端组件。

- **Truffle**

  以太坊开发框架。

3.36.3 编辑器集成

- Emacs
  - **Emacs Solidity**

    Emacs 编辑器的插件，提供语法高亮和编译错误报告。

- IntelliJ
  - **IntelliJ IDEA 插件**

    IntelliJ IDEA（和所有其他 JetBrains IDEs）的 Solidity 插件

- Sublime Text
  - **SublimeText 的软件包 - Solidity 语言语法**

    用于 SublimeText 编辑器的 Solidity 语法高亮。

- Vim
  - **Thesis 的 Vim Solidity**

    Vim 中 Solidity 的语法高亮。

  - **TovarishFin 的 Vim Solidity**

    Solidity 的 Vim 语法文件。

  - **Vim Syntastic**

    为 Vim 编辑器提供编译检查的插件。

- Visual Studio Code (VS Code)
  - **以太坊 Remix 的 Visual Studio 代码扩展包**

    VS Code 的以太坊 Remix 扩展包

  - **Juan Blanco 的 Solidity Visual Studio 代码扩展包**

    Microsoft Visual Studio Code 的 Solidity 插件，包括语法高亮和 Solidity 编译器。

  - **Nomic Foundation 的 Solidity Visual Studio 代码扩展包**

    由 Hardhat 团队提供的 Solidity 和 Hardhat 支持，包括：语法高亮，跳转到定义，重命名，快速修复和内联 solc 警告和错误。

– **Solidity 可视化审计扩展**

  在 Visual Studio Code 中增加了以安全为中心的语法和语义突出显示。

– **用于 VS Code 的 Truffle**

  在 Ethereum 和 EVM 兼容的区块链上构建，调试和部署智能合约。

3.36.4 Solidity 工具

- **ABI 到 Solidity 接口转换器**

  一个用于从智能合约的 ABI 生成合约接口的脚本。

- **abi-to-sol**

  从一个给定的 ABI JSON 生成 Solidity 接口源的工具。

- **Doxity**

  Solidity 的文档生成器。

- **Ethlint**

  识别和修复 Solidity 中的风格和安全问题的语法检查器。

- **evmdis**

  EVM 反汇编程序，对字节码进行静态分析，提供比原始 EVM 操作更高的抽象水平。

- **EVM Lab**

  丰富的工具包，与 EVM 互动。包括一个虚拟机、以太链 API，以及一个带有 gas 成本显示的跟踪查看器。

- **hevm**

  EVM 调试器和符号执行引擎。

- **leafleth**

  Solidity 智能合约的文档生成器。

- **PIET**

  一个通过简单图形界面开发，审计和使用 Solidity 智能合约的工具。

- **Scaffold-ETH**

  专注于产品快速迭代的可分叉的以太坊开发堆栈。

- **sol2uml**

  Solidity 合约的统一建模语言（UML）类图生成器。

- **solc-select**

  一个在 Solidity 编译器版本之间快速切换的脚本。

- **优化 Solidity 语言格式插件**

  Solidity 格式美化插件。

- **Solidity REPL**

  使用命令行 solidity 控制台立即尝试 solidity。

- **solgraph**
  可视化 Solidity 控制流并突出潜在的安全漏洞。
- **Solhint**
  Solidity 语法检查器，为智能合约的验证提供安全，风格指南和最佳实践规则。
- **Sourcify**
  去中心化的自动合约验证服务和合约元数据的公共存储库。
- **Sūrya**
  智能合约系统的实用工具，提供一些可视化输出和关于合约结构的信息。还支持查询函数调用图。
- **Universal Mutator**
  一个用于突变生成的工具，具有可配置的规则并支持 Solidity 和 Vyper。

3.36.5 第三方 Solidity 解析器和语法

- **用于 JavaScript 的 Solidity 解析器**
  一个建立在强大的 ANTLR4 语法之上的 JS Solidity 解析器。

3.37 贡献方式

对于大家的帮助，我们一如既往地欢迎。而且有很多选择可以为 Solidity 做出贡献。

特别是，我们感谢在以下领域的支持:

- 报告问题。
- 修复和响应 Solidity 的 GitHub 问题，特别是那些被标记为 " 很好的第一个问题"，这是作为外部贡献者的介绍性问题。
- 完善文档。
- 将文档 翻译 成更多的语言。
- 在 StackExchange 和 Solidity Gitter Chat 上回答其他用户的问题。
- 通过在 Solidity 论坛 上提出语言改进或新功能，并提供反馈来参与语言设计的过程。

为了开始参与，您可以尝试从源代码编译，以熟悉 Solidity 的组件和构建过程。此外，精通在 Solidity 中编写智能合约可能是有用的。

请注意，本项目发布时有一个 贡献者行为准则。参与此项目 - 在 issues，pull requests 或 Gitter channels 中 - 即表示您同意遵守其条款。

3.37.1 团队电话会议

如果您有问题或拉动请求要讨论，或有兴趣听听团队和贡献者正在做什么，您可以加入我们的公共团队电话会议：

- 每周三下午 3 点，中欧标准时间/中欧夏令时间。

会议在 Jitsi 举行。

3.37.2 如何报告问题

要报告一个问题，请使用 GitHub 问题跟踪器。当报告问题时，请提及以下细节：

- Solidity 版本。
- 源代码（如果可以的话）。
- 操作系统。
- 重现该问题的步骤。
- 实际行为与预期行为。

将导致问题的源代码减少到最低限度总是非常有帮助的，有时甚至可以澄清一个误解。

关于语言设计的技术讨论，去 Solidity 论坛 才是正确的选择（见*Solidity 语言设计*）。

3.37.3 拉取请求的工作流程

为了进行贡献，请 fork 一个 `develop` 分支并在那里进行修改。除了您 做了什么之外，您还需要在提交信息中说明，您 为什么做这些修改（除非只是个微小的改动）。

在进行了 fork 之后，如果您还需要从 `develop` 分支 pull 任何变更的话（例如，为了解决潜在的合并冲突），请避免使用 `git rebase` ，而是用 `git rebase` 您的分支。

此外，如果您正在编写一个新的功能，请确保您在 `test/` 下添加适当的测试案例（见下文）。

但是，如果您在进行一个更大的变更，请先与 Solidity Development Gitter channel （与上文提到的不同 - 这个变更侧重于编译器和编程语言开发，而不是编程语言的使用）进行咨询。

新的特性和 bug 修复会被添加到 `Changelog.md` 文件中：使用的时候请遵循上述方式。

最后，请确保您遵守了这个项目的 编码风格 。还有，虽然我们采用了持续集成测试，但是在提交 pull request 之前，请测试您的代码并确保它能在本地进行编译。

我们强烈建议在提交拉动请求之前，先看一下我们的 审查清单。我们会彻底审查每一个 PR，并会帮助您把它弄好，但有许多常见问题可以很容易地避免，使审查更加顺利。

感谢您的帮助！

3.37.4 运行编译器测试

先决条件

为了运行所有的编译器测试，您可能想选择性地安装一些依赖项（evmone，libz3，和 libhera）。

在 macOS 系统上，一些测试脚本需要安装 GNU 核心工具。可以使用 Homebrew 很简单地完成安装：`brew install coreutils`。

在 Windows 系统上，确保您有创建符号链接的权限，否则一些测试可能会失败。管理员应该有这个权限，但您也可以 将其授予其他用户 或 启用开发者模式。

运行测试

Solidity 包括不同类型的测试，其中大部分捆绑在 Boost C++ 测试框架 应用程序 `soltest`。运行 `build/test/soltest` 或其包装器 `scripts/soltest.sh` 对大多数变化来说是足够的。

`./scripts/tests.sh` 脚本自动执行大多数 Solidity 测试，包括那些捆绑在 Boost C++ 测试框架 应用程序 `soltest`（或其包装器 `scripts/soltest.sh`）中的测试，以及命令行测试和编译测试。

测试系统会自动尝试发现 evmone 的位置，以运行语义测试。

`evmone` 库必须位于当前工作目录相对的 `deps` 或 `deps/lib` 目录，其父级目录或其父级目录的父级目录中。另外，可以通过 `ETH_EVMONE` 环境变量指定 `evmone` 共享对象的显式位置。

`evmone` 主要用于运行语义和 gas 测试。如果您没有安装它，您可以通过向 `scripts/soltest.sh` 传递 `--no-semantic-tests` 标志来跳过这些测试。

运行 Ewasm 测试默认是禁用的，可以通过 `./scripts/soltest.sh --ewasm` 明确启用，要求 hera 被 `soltest` 找到。定位 `hera` 库的机制与 `evmone` 相同，只是用于指定明确位置的变量被称为 `ETH_HERA`。

`evmone` 和 `hera` 库的文件名后缀都应该是 Linux 上的 `.so`，Windows 系统上的 `.dll`，MacOS 上的 `.dylib`。

为了运行 SMT 测试，`libz3` 库必须被安装，并在编译器配置阶段被 `cmake` 可以找到。

如果您的系统没有安装 `libz3` 库，您应该在运行 `./scripts/tests.sh` 或 `./scripts/soltest.sh --no-smt` 之前，通过导出 `SMT_FLAGS=--no-smt` 来禁用 SMT 测试。这些测试是 `libsolidity/smtCheckerTests` 和 `libsolidity/smtCheckerTestsJSON`。

备注： 要获得 Soltest 运行的所有单元测试的列表，请运行 `./build/test/soltest --list_content=HRF`。

为了获得更快的结果，您可以运行一个子集，或特定的测试。

要运行测试的一个子集，可以使用过滤器：`./scripts/soltest.sh -t TestSuite/TestName`，其中 `TestName` 可以是通配符 `*`。

或者，举例来说，运行 yul 消歧义器的所有测试：`./scripts/soltest.sh -t "yulOptimizerTests/disambiguator/*" --no-smt`。

`./build/test/soltest --help` 有关于所有可用选项的广泛帮助。

尤其是可以查看：

- show_progress (-p) 来显示测试完成。
- run_test (-t) 来运行特定的测试案例，以及
- report-level (-r) 给出一个更详细的报告。

备注： 那些在 Windows 环境下使用的人，想在没有 libz3 的情况下运行上述基本集，可以使用 Git Bash，使用命令为：`./build/test/Release/soltest.exe -- --no-smt`。如果您在普通的命令提示符下运行，使用 `.\build\test\Release\soltest.exe -- --no-smt`。

如果您想使用 GDB 进行调试，确保您的构建方式与“通常”不同。例如，您可以在您的 `build` 文件夹中运行以下命令：

如果您想使用 GDB 进行调试，请确保您的构建方式与“通常”的构建方式不同。例如，您可以在 `build` 文件夹中运行以下命令：

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
make
```

这会创建了一些符号，所以当您使用 `--debug` 标志调试测试时，您可以访问其中的函数和变量，您可以用它来中断或打印。

CI 运行额外的测试（包括 `solc-js` 和测试第三方 Solidity 框架），需要编译 Emscripten 目标。

编写和运行语法测试

语法测试检查编译器是否对无效的代码产生正确的错误信息，并正确接受有效的代码。它们被保存在 `tests/libsolidity/syntaxTests` 文件夹下的单个文件中。这些文件必须包含注释，说明各自测试的预期结果。测试套件会根据给定的期望值进行编译和检查。

例如：`./test/libsolidity/syntaxTests/double_stateVariable_declaration.sol`

```
contract test {
    uint256 variable;
    uint128 variable;
}
// ----
// 声明错误： (36-52) 。标识符已被声明。
```

语法测试必须至少包含被测合约本身，后面是分隔符 //----。分隔符后面的注释是用来描述预期的编译器错误或警告的。数字范围表示错误发生在源代码中的位置。如果您希望合约在编译时没有任何错误或警告，您可以不使用分隔符和后面的注释。

在上面的例子中，状态变量 variable 被声明了两次，这是不允许的。这导致了一个 声明错误，说明标识符已经被声明。

用来进行那些测试的工具叫做 isoltest，可以在 ./build/test/tools/ 下找到。它是一个交互工具，允许您使用您喜欢的文本编辑器编辑失败的合约。让我们把第二个 variable 的声明去掉来使测试失败：

```
contract test {
    uint256 variable;
}
// ----
// 声明错误: (36-52) 。标识符已被声明。
```

再次运行 ./build/test/tools/isoltest 就会得到一个失败的测试：

```
syntaxTests/double_stateVariable_declaration.sol: FAIL
    Contract:
        contract test {
            uint256 variable;
        }

    Expected result:
        DeclarationError: (36-52): Identifier already declared.
    Obtained result:
        Success
```

isoltest 在获得的结果旁边打印出预期的结果，还提供了一个编辑，更新，跳过当前合约文件或退出应用程序的办法。

它为失败的测试提供了几种选择：

- edit：isoltest 试图在一个编辑器中打开合约，以便您可以调整它。它或者使用命令行上给出的编辑器（如 isoltest --editor /path/to/editor），或者在环境变量 EDITOR 中，或者只是 /usr/bin/editor （按这个顺序）。
- update：更新测试中的合约。这将会移除包含了不匹配异常的注解，或者增加缺失的预想结果。然后测试会重新开始。
- skip：跳过这一特定测试的执行。
- quit：退出 isoltest。

所有这些选项都适用于当前的合约，除了 quit，它可以停止整个测试过程。

在上边的情况自动更新合约会把它变为

```
contract test {
    uint256 variable;
}
// ----
```

并重新运行测试。它将会通过:

```
Re-running test case...
syntaxTests/double_stateVariable_declaration.sol: OK
```

备注: 为合约文件选择一个能解释其测试内容的名字,例如: `double_variable_declaration.sol`。不要把一个以上的合约放在一个文件中,除非您在测试继承或跨合约的调用。每个文件应该测试您的新功能的一个方面。

3.37.5 通过 AFL 运行 Fuzzer

Fuzzing 是一种测试技术,它可以通过运行多少不等的随机输入来找出异常的执行状态(片段故障、异常等等)。现代的 fuzzer 已经可以很聪明地在输入中进行直接的查询。我们有一个专门的程序叫做 `solfuzzer`,它可以将源代码作为输入,当发生一个内部编译错误,片段故障或者类似的错误时失败,但当代码包含错误的时候则不会失败。通过这种方法,fuzzing 工具可以找到那些编译级别的内部错误。

我们主要使用 AFL 来进行 fuzzing 测试。您需要手工下载和构建 AFL。然后用 AFL 作为编译器来构建 Solidity (或只是 `solfuzzer` 二进制文件):

```
cd build
# 如 果 需 要 的 话
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-gcc -DCMAKE_CXX_COMPILER=path/to/afl-g++
make solfuzzer
```

在这个阶段,您应该能够看到类似以下的信息:

```
Scanning dependencies of target solfuzzer
[ 98%] Building CXX object test/tools/CMakeFiles/solfuzzer.dir/fuzzer.cpp.o
afl-cc 2.52b by <lcamtuf@google.com>
afl-as 2.52b by <lcamtuf@google.com>
[+] Instrumented 1949 locations (64-bit, non-hardened mode, ratio 100%).
[100%] Linking CXX executable solfuzzer
```

如果指示信息没有出现,尝试切换指向 AFL 的 clang 二进制文件的 cmake 标志:

```
# 如果之前失败了
make clean
cmake .. -DCMAKE_C_COMPILER=path/to/afl-clang -DCMAKE_CXX_COMPILER=path/to/afl-clang++
make solfuzzer
```

否则，在执行时，fuzzer 就会停止，并出现错误，说二进制没有被检测到。

```
afl-fuzz 2.52b by <lcamtuf@google.com>
... (truncated messages)
[*] Validating target binary...

[-] Looks like the target binary is not instrumented! The fuzzer depends on
    compile-time instrumentation to isolate interesting test cases while
    mutating the input data. For more information, and for tips on how to
    instrument binaries, please see /usr/share/doc/afl-doc/docs/README.

    When source code is not available, you may be able to leverage QEMU
    mode support. Consult the README for tips on how to enable this.
    (It is also possible to use afl-fuzz as a traditional, "dumb" fuzzer.
    For that, you can use the -n option - but expect much worse results.)

[-] PROGRAM ABORT : No instrumentation detected
         Location : check_binary(), afl-fuzz.c:6920
```

接下来，您需要一些示例源文件。这使得 fuzzer 更容易发现错误。您可以从语法测试中复制一些文件，或者从文档或其他测试中提取测试文件。

```
mkdir /tmp/test_cases
cd /tmp/test_cases
# 从测试中提取：
path/to/solidity/scripts/isolate_tests.py path/to/solidity/test/libsolidity/
↪SolidityEndToEndTest.cpp
# 从文件中摘录：
path/to/solidity/scripts/isolate_tests.py path/to/solidity/docs
```

AFL 的文档指出，账册（初始的输入文件）不应该太大。每个文件本身不应该超过 1 kB，并且每个功能最多只能有一个输入文件；所以最好从少量的输入文件开始。此外还有一个叫做 `afl-cmin` 的工具，可以将输入文件整理为可以具有近似行为的二进制代码。

现在运行 fuzzer（`-m` 参数将使用的内存大小扩展为 60 MB）：

```
afl-fuzz -m 60 -i /tmp/test_cases -o /tmp/fuzzer_reports -- /path/to/solfuzzer
```

fuzzer 会将导致失败的源文件创建在 `/tmp/fuzzer_reports` 中。通常它会找到产生相似错误的类似的源文件。您可以使用 `scripts/uniqueErrors.sh` 工具来那些独特的错误。

3.37.6 Whiskers 系统

Whiskers 是一个类似于 Mustache 的字符串模板化系统。它被编译器用在不同的地方，以帮助代码的可读性，从而帮助代码的可维护性和可验证性。

该语法与 Mustache 有很大区别。模板标记 `{{` 和 `}}` 被 `<` 和 `>` 取代，以帮助解析并避免与 *Yul* 的冲突（符号 `<` 和 `>` 在内联汇编中是无效的，而 `{` 和 `}` 是用来限定块的）。另一个限制是，列表只能解决一个深度的问题，而且它们不会递归。这在将来可能会改变。

下面是一个粗略的说明：

任何出现的 `<name>` 的地方都会被提供的变量 `name` 的字符串值替换，没有任何转义，也没有迭代替换。可以用 `<#name>...</name>` 来划定一个区域。该区域中的内容将进行多次拼接，每次拼接会使用相应变量集中的值替换区域中的 `<inner>` 项，模板系统中提供了多少组变量集，就会进行多少次拼接。顶层变量也可以在这种区域内使用。

还有一些判断条件的表达式 `<?name <!name>...</name>`，根据布尔参数 `name` 的值，会在第一段或第二段继续递归地替换模板。如果使用 `<?+name>...<!+name>...</+name>` 这种表达式，那么检查的是字符串参数 `name` 是否为非空。

3.37.7 文档风格指南

在下面的部分，您可以找到专门针对 Solidity 文档贡献的风格建议。

英语

使用国际英语，除非使用项目或品牌名称。尽量减少使用当地的俚语和参考文化，尽量使您的语言对所有的读者都尽可能清晰。以下是一些参考资料，希望对大家有所帮助：

- 简化技术英语
- 国际英语

备注： 虽然官方的 Solidity 文档是用英语写的，但也有社区贡献的其他语言的翻译 可用。请参考 翻译指南 以了解如何为社区翻译作出贡献。

标题的大小写

在标题中使用 标题大小写。这意味着标题中的所有主词都要大写，但不包括冠词，连接词和介词，除非它们是标题的开头。

例如，下列各项都是正确的：

- Title Case for Headings.
- For Headings Use Title Case.
- Local and State Variable Names.
- Order of Layout.

扩写缩写

使用扩展的缩略语来表达单词，例如：

- ”Do not” 替代”Don't”。
- ”Can not” 替代”Can't”。

主动和被动语态

主动语态通常被推荐用于教程风格的文档，因为它有助于读者理解谁或什么在执行一项任务。然而，由于 Solidity 文档是教程和参考内容的混合物，被动语态有时更适用。

综上所述：

- 在技术参考方面使用被动语态，例如语言定义和 Ethereum 虚拟机的内部情况。
- 在描述关于如何应用 Solidity 某方面的建议时，使用主动语态。

例如，下面的内容是被动语态，因为它指定了 Solidity 的一个方面：

> 函数可以被声明为 `pure`，在这种情况下，它们承诺不读取或修改状态。

例如，下面是主动语态，因为它讨论了 Solidity 的一个应用：

> 在调用编译器时，您可以指定如何发现一个路径的第一个元素，也可以指定路径前缀的重映射。

常用术语

- “函数参数“和“返回变量“，而不是输入和输出参数。

代码示例

CI 进程在您创建 PR 时，使用 `./test/cmdlineTests.sh` 脚本测试所有以 `pragma solidity`，`contract`，`library` 或 `interface` 开头的代码块格式的示例代码。如果您正在添加新的代码实例，在创建 PR 之前确保它们能够工作并通过测试。

确保所有的代码实例以 `pragma` 版本开始，跨越合约代码有效的最大范围。例如 `pragma solidity >=0.4.0 <0.9.0;`。

运行文档测试

通过运行 `./scripts/docs.sh` 来确保您的贡献通过我们的文档测试，它安装了文档所需的依赖，并检查是否存在问题，如无效的链接或语法问题。

3.37.8 Solidity 语言设计

为了积极参与语言设计过程，并分享您关于 Solidity 未来的想法，请加入 Solidity 论坛。

Solidity 论坛作为提出和讨论新的语言功能及其在早期构思阶段的实现或现有功能的修改的一个地方。

一旦提案变得更加具体，它们的实施也将在 Solidity GitHub 仓库 中以问题的形式讨论。

除了论坛和问题讨论之外，我们还定期举办语言设计讨论会议，对选定的主题，问题或功能实现进行详细的辩论。这些会议的邀请函通过论坛共享。

我们也在论坛中分享反馈调查和其他与语言设计相关的内容。

如果您想知道团队在实施新功能方面的情况，您可以在 Solidity Github 项目 中关注实施状况。设计积压中的问题需要进一步规范，将在语言设计电话会议或常规团队电话会议中讨论。您可以通过从默认分支（*develop* ）到 breaking 分支 来查看下一个突破性版本即将发生的变化。

对于特殊情况和问题，您可以通过 Solidity-dev Gitter 频道 与我们联系，- 这是一个专门用于围绕 Solidity 编译器和语言开发的聊天室。

我们很高兴听到您对我们如何改进语言设计过程，使之更加协作和透明的想法。

3.38 语言的影响因素

Solidity 是一种 花括号语言，受到几种著名编程语言的影响和启发。

Solidity 受 C++ 的影响最深，但也借用了 Python，JavaScript 等语言的概念。

从变量声明的语法，for 循环，重载函数的概念，隐式和显式类型转换以及许多其他细节中可以看出 C++ 的影响。

这是由于变量的函数级范围和关键字 `var` 的使用。从 0.4.0 版本开始，JavaScript 的影响已经减少。现在，剩下的与 JavaScript 的主要相似之处是，使用关键字 `function` 来定义函数。Solidity 还支持导入语法和语义，

这些都与 JavaScript 中的相似。除了这些点，Solidity 看起来和其他大多数花括号语言一样，不再有主要的 JavaScript 影响。

对 Solidity 的另一个影响是 Python。Solidity 的修改器是为了模拟 Python 的装饰器而添加的，但其功能受到很大限制。此外，多重继承，C3 线性化和 `super` 关键字以及值和引用类型的一般赋值和复制语义都来自 Python。

3.39 Solidity 品牌指南

该品牌指南的特点是关于 Solidity 的品牌政策和标志使用指南的信息。

3.39.1 Solidity 品牌

Solidity 编程语言是一个开源的社区项目，由一个核心团队管理。该核心团队由 以太坊基金会 赞助。

本文件旨在提供有关如何最好地使用 Solidity 品牌名称和标识的信息。

我们鼓励您在使用该品牌名称或标志之前仔细阅读本文件。我们非常感谢您的合作!

3.39.2 Solidity 品牌名称

“Solidity”应该只用来指 Solidity 编程语言。

请不要按以下方式使用“Solidity”：

- 指的是任何其他编程语言。
- 以一种误导性的方式，或可能暗示不相关的模块，工具，文档或其他资源与 Solidity 编程语言的关联。
- 在方式上，混淆了社区对 Solidity 编程语言是否开源和免费使用的看法。

3.39.3 Solidity 标志许可

Solidity 标志是在 创意共享署名 4.0 国际许可协议 下发布和许可的。

这是最宽松的知识共享协议，允许为任何目的进行再利用和修改。

您可以自由选择以下方式：

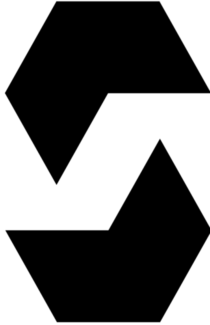
- **分享** - 以任何媒介或形式复制和重新分发材料。
- **修改** - 为任何目的，甚至为商业目的，对材料进行混音，改造和构建。

根据以下条款：

- **署名** - 您必须给予适当的信用，提供到许可证的链接，并说明是否进行了修改。您可以以任何合理的方式这样做，但不能以任何方式暗示 Solidity 核心团队认可您或您的使用。

在使用 Solidity 标识时，请尊重 Solidity 标识指南。

3.39.4 Solidity 标志指南



（右键点击标识即可下载。）

请不要这样做：

- 改变标志的比例（不要拉伸或切割）。
- 改变标志的颜色，除非是绝对必要。

3.39.5 信用

本文件部分内容来自于 Python 软件基金会商标使用政策 和 Rust 媒介指南 。

索引

符号

<stdin>, 391
--allow-paths, 208, **396**
--base-path, 208, 209, **394**
--include-path, 208, **394**
--libraries, **208**
--link, **208**
--standard-json, 209, **211**

A

abi, 101, 102, 272
    decode, 177
    encode, 177
    encodeCall, 177
    encodePacked, 177
    encodeWithSelector, 177
    encodeWithSignature, 177
ABI coder, **51**
abstract contract, 155, **158**
access
    restricting, 428
account, **12**
addmod, 103, 179
address, 12, 61, 65
    balance, 178
    code, 178
    codehash, 178
    send, 178
    transfer, 178
allowed paths, 208, **396**
analyse, 222
anonymous, 180
application binary interface, 272
array, 75, **77**, 135
    allocating, **78**
    dangling storage references, **83**
    length, **81**
    pop, **81**
    push, **81**
    slice, **85**
array of strings, 135
asm, **170**, 222, **364**
assembly, **170**, **364**
assembly-flags (*Antlr4 production rule*), 195
assembly-statement (*Antlr4 production rule*), 195
assert, 103, **117**, 179
assignment, 94, **113**
    destructuring, **113**
auction
    blind, 19
    open, 19

B

balance, 12, 61, 104
ballot, 16
base
    constructor, **155**
base class, **148**
base path, 208, **394**
blind auction, 19

block, **11**, 101, 178
 basefree, 178
 chainid, 178
 coinbase, 178
 difficulty, 178
 gaslimit, 178
 number, 101, 178
 prevrandao, 178
 timestamp, 101, 178
block (*Antlr4 production rule*), 193
blockhash, 178
bool, **58**
boolean-literal (*Antlr4 production rule*), 192
break, 108
break-statement (*Antlr4 production rule*), 194
Bugs, 296
byte array, 65
bytes, 68, **77**
 concat, 178
bytes members, 103
bytes32, 65
bytes-concat, **78**

C

C3 linearization, **156**
call, 61, 104
call-argument-list (*Antlr4 production rule*), 183
callcode, 104, 161
cast, **96**
catch-clause (*Antlr4 production rule*), 195
checked, 116
cleanup, 231
codehash, 104
coding style, 400
coin, 10
coinbase, 101
commandline compiler, **207**
comment, **53**
common subexpression elimination, 243
compile target, 209
compiler
 commandline, 207
compound operators, **94**
constant, **132**, 180
constant propagation, 243
constant-variable-declaration (*Antlr4 production rule*), 187
constructor, 123, **154**
 arguments, 123
constructor-definition (*Antlr4 production rule*), 184
continue, 108
continue-statement (*Antlr4 production rule*), 194
contract, 54, **123**
 abstract, 155, **158**
 base, **148**
 creation, **123**
 interface, **159**
 modular, 39
 precompiled, **15**
contract creation, 15
contract type, **64**
contract verification, 266
contract-body-element (*Antlr4 production rule*), 182
contract-definition (*Antlr4 production rule*), 182
contract-function-definition (*Antlr4 production rule*), 184
contracts
 creating, 111
creationCode, 106
cryptography, 103, 179
custom type, 69

D

data, 101
data-location (*Antlr4 production rule*), 190
days, **100**
deactivate, 15
decimal-number (*Antlr4 production rule*), 205
declarations, 114
default value, 114
delegatecall, 14, 61, 104, 161
delete, **94**

denomination, **99**
    ether, **100**
    time, **100**
deriving, **148**
difficulty, 101
direct import, **392**
dirty bits, 231
do/while, 108
do-while-statement (*Antlr4 production rule*), 194
double-quoted-printable (*Antlr4 production rule*), 204
dynamic array, 135

E

ecrecover, 103, 179
elementary-type-name (*Antlr4 production rule*), 189
else, 108
emit-statement (*Antlr4 production rule*), 195
empty-string-literal (*Antlr4 production rule*), 204
encode, 101
encoding, 102
enum, 54, 68
enum-definition (*Antlr4 production rule*), 186
error, **146**, 282
error-definition (*Antlr4 production rule*), 187
error-parameter (*Antlr4 production rule*), 187
errors, **117**
escape-sequence (*Antlr4 production rule*), 204
escrow, 27
ether, **100**
ethereum virtual machine, **12**
evaluation order
    expression, **228**
    function arguments, **229**
event, 10, 54, **143**
    anonymous, **143**
    indexed, **143**
    topic, **143**
event-definition (*Antlr4 production rule*), 187
event-parameter (*Antlr4 production rule*), 187
evm, **12**
EVM version, **209**
evmasm, **170**, **364**
exception, **117**
expression (*Antlr4 production rule*), 190
expression-statement (*Antlr4 production rule*), 196
external, 125, 180

F

fallback function, **140**
fallback-function-definition (*Antlr4 production rule*), 185
false, **58**
file://, 399
filesystem path, 53
finney, **100**
fixed, **60**
fixed point number, **60**
fixed-bytes (*Antlr4 production rule*), 199
for, 108
for-statement (*Antlr4 production rule*), 194
free-function-definition (*Antlr4 production rule*), 185
function, 54
    call, 14, **108**
    external, 108
    fallback, 140
    free, **133**, 166
    getter, **127**
    internal, 108
    modifier, 54, **129**, 428, 430
    pure, 137
    receive, 139
    view, 136
function parameter, 108
function pointers, 230
function type, **70**
function-type-name (*Antlr4 production rule*), 189
functions, **133**

G

gas, **13**, 101
gas price, **13**, 101
gasleft, 178
getter
    function, **127**
goto, 108
gwei, **100**

H

hex-number (*Antlr4 production rule*), 205
hex-string (*Antlr4 production rule*), 204
hex-string-literal (*Antlr4 production rule*), 192
Host Filesystem Loader, **389**
hours, **100**

I

identifier (*Antlr4 production rule*), 192, 205
identifier-path (*Antlr4 production rule*), 183
if, 108
if-statement (*Antlr4 production rule*), 193
import, **52**
    direct, 392
    path, 53, **391**
    relative, **392**
    remapping, **397**
import callback, 53, **389**
import-directive (*Antlr4 production rule*), 181
include paths, 208, **394**
indexed, 180
inheritance, **148**
    multiple, **156**
inheritance list, 155
inheritance-specifier (*Antlr4 production rule*), 182
inline
    arrays, **79**
inline-array-expression (*Antlr4 production rule*), 192
installing, **40**
instruction, **14**
int, **58**
integer, **58**, 66
interface contract, **159**
interface-definition (*Antlr4 production rule*), 182
internal, 125, 180
iterable mappings, **91**
iulia, 364

J

julia, 364

K

keccak256, 103, 179

L

length, 81
library, 14, **161**, 166
library-definition (*Antlr4 production rule*), 182
license, **50**
linearization, **156**
linker, **208**
literal
    address, **65**, 99
    array, **79**
    conversion, **98**
    hexadecimal, **68**, 99
    hexadecimal number, 98
    in Yul, **366**
    rational, **66**, 98
    string, **67**, 99
    unicode, **68**
literal (*Antlr4 production rule*), 192
literal-with-sub-denomination (*Antlr4 production rule*), 192
location, 75
log, 14
lvalue, 94

M

mapping, 10, **88**, 231
mapping-key-type (*Antlr4 production rule*), 196
mapping-type (*Antlr4 production rule*), 196
memory, **13**, 75

message call, **14**
metadata, 266
minutes, **100**
modifier-definition (*Antlr4 production rule*), 185
modifier-invocation (*Antlr4 production rule*), 183
modifiers, 180
modular contract, 39
module, 52
msg, 101
    data, 178
    sender, 178
    sig, 178
    value, 178
mulmod, 103, 179

N

natspec, 53
new, 78, **111**
non-empty-string-literal (*Antlr4 production rule*), 203
number, 101
number-literal (*Antlr4 production rule*), 193

O

open auction, 19
operator, **93**
    precedence, **95**, 176
    user-defined, **166**
optimiser, 243
optimizer, 243
origin, 101
overload, **142**
override-specifier (*Antlr4 production rule*), 184
overriding
    function, **151**
    modifier, **154**

P

packed, 102
parameter, **108**
    function, 108
    input, 108
    output, 108
parameter-list (*Antlr4 production rule*), 184
path (*Antlr4 production rule*), 181
payable, 180
pop, 81
pragma, **50**
    abicoder, **51**
    ABIEncoderV2, 51, **52**
    experimental, **52**
    SMTChecker, **52**
    version, **51**
pragma-token (*Antlr4 production rule*), 207
precompiled contracts, **15**
precompiles, **15**
prevrandao, 101
private, 125, 180
public, 125, 180
purchase, 27
pure, 180
pure function, **137**
push, 81

R

rational number, 66
receive, **139**
receive ether function, **139**
receive-function-definition (*Antlr4 production rule*), 186
reference type, **75**
relative import, **392**
remapping
    context, **397**
    import, **397**
    prefix, **397**
    target, 396, **397**
Remix IDE, 53, 399
remote purchase, 27
require, 103, **117**, 179
return, 108
return array, 135
return string, 135
return struct, 135

return variable, 108
return-statement (*Antlr4 production rule*), 195
revert, 103, **117**, 146, 179
revert-statement (*Antlr4 production rule*), 195
ripemd160, 103, 179
runtimeCode, 106

S

safe math, **116**
safemath, 116
scoping, **114**
seconds, **100**
selector
    of a function, 172, **272**
    of a library function, **165**
    of an error, **146**, 282
    of an event, **144**
selfdestruct, **15**, 106, 179
send, 61, 104
sender, 101
set, 161
sha256, 103, 179
signed-integer-type (*Antlr4 production rule*), 201
single-quoted-printable (*Antlr4 production rule*), 204
solc, **207**
SolidityLexer (*Antlr4 lexer grammar*), 199
SolidityParser (*Antlr4 parser grammar*), 181
source file, 52
source mappings, 242
source unit, 52
source unit name, 53, **389**
source-unit (*Antlr4 production rule*), 181
spdx, 50
stack, **13**
standard input, 391
standard JSON, **211**, 390
state machine, 430
state variable, 54, 231
state-mutability (*Antlr4 production rule*), 184
state-variable-declaration (*Antlr4 production rule*), 187
statement (*Antlr4 production rule*), 193
staticcall, 61, 104
stdin, 391
storage, 12, **13**, 75, 231
string, 67, **77**, 135
    concat, 178
string members, 103
string-concat, **78**
string-literal (*Antlr4 production rule*), 192
struct, 54, 75, **86**, 135
struct-definition (*Antlr4 production rule*), 186
struct-member (*Antlr4 production rule*), 186
style, 400
sub-denomination (*Antlr4 production rule*), 200
subcurrency, **8**
super, 106, 179
switch, 108
symbol-aliases (*Antlr4 production rule*), 181
szabo, **100**

T

this, 106, 179
throw, **117**
timestamp, 101
transaction, 11, **12**
transfer, 61, 104
true, **58**
try-statement (*Antlr4 production rule*), 194
tuple-expression (*Antlr4 production rule*), 192
tx
    gasprice, 178
    origin, 178
type, 57, 106
    contract, **64**
    conversion, **96**
    creationCode, 180
    function, **70**
    interfaceId, 180
    max, 180
    min, 180
    name, 180
    reference, **75**

runtimeCode, 180
struct, **86**
value, **57**
type-name (*Antlr4 production rule*), 189

U

ufixed, **60**
uint, **58**
unchecked, 116
unchecked-block (*Antlr4 production rule*), 193
unicode-string-literal (*Antlr4 production rule*), 193, 204
unsigned-integer-type (*Antlr4 production rule*), 202
unused store eliminator, **261**
user defined value type, **69**
user-definable-operator (*Antlr4 production rule*), 188
user-defined-value-type-definition (*Antlr4 production rule*), 186
using for, 161, **166**
using-directive (*Antlr4 production rule*), 188

V

value, 101
value type, **57**
variable
return, 108
variable-declaration (*Antlr4 production rule*), 190
variable-declaration-statement (*Antlr4 production rule*), 196
variable-declaration-tuple (*Antlr4 production rule*), 195
variably sized array, 135
VFS, **389**
view, 180
view function, **136**
virtual filesystem, 53, **389**
visibility, **125**, 180
visibility (*Antlr4 production rule*), 183
voting, 16

W

weeks, **100**
wei, **100**
while, 108
while-statement (*Antlr4 production rule*), 194
withdrawal, 426

Y

years, **100**
yul, **364**
yul-assignment (*Antlr4 production rule*), 197
yul-block (*Antlr4 production rule*), 197
yul-boolean (*Antlr4 production rule*), 198
yul-decimal-number (*Antlr4 production rule*), 207
yul-evm-builtin (*Antlr4 production rule*), 205
yul-expression (*Antlr4 production rule*), 199
yul-for-statement (*Antlr4 production rule*), 197
yul-function-call (*Antlr4 production rule*), 198
yul-function-definition (*Antlr4 production rule*), 198
yul-hex-number (*Antlr4 production rule*), 207
yul-identifier (*Antlr4 production rule*), 207
yul-if-statement (*Antlr4 production rule*), 197
yul-literal (*Antlr4 production rule*), 198
yul-path (*Antlr4 production rule*), 198
yul-statement (*Antlr4 production rule*), 196
yul-string-literal (*Antlr4 production rule*), 207
yul-switch-statement (*Antlr4 production rule*), 197
yul-variable-declaration (*Antlr4 production rule*), 197