solidity面试题(三)

这是根据一个面试题给出的问题总结 https://learnblockchain.cn/article/7076，我自己附上答案。另外还有问题或者答案里涉及到一些概念，在文中也会把它变成一个问题。这里是高级题的答案，如有不正确的地方，请在评论区指正。

1. 以太坊预编译合约的地址是什么?

答：由于 EVM 是一个基于堆栈的虚拟机，它根据交易所要执行的操作指令内容来计算 gas 消耗，如果计算非常复杂，在 EVM 中执行相关操作指令就会非常低效，而且会消耗大量的 gas。 例如，在 zk-snark 中，需要对椭圆曲线进行加减运算和配对运算。 在 EVM 中执行这些操作是非常复杂和不现实的。所幸以太坊还支持预编译合约。

预编译合约是 EVM 中用于提供更复杂库函数(通常用于加密、散列等复杂操作)的一种折衷方法，这些函数不适合编写操作码。 它们适用于简单但经常调用的合约，或逻辑上固定但计算量很大的合约。 预编译合约是在使用节点客户端代码实现的，因为它们不需要 EVM，所以运行速度很快。 与使用直接在 EVM 中运行的函数相比，它对开发人员来说成本也更低。

在代码层面，所谓的地址实际上是合约数组的索引，每一个索引唯一对应一个预编一个合约。

2. 当函数数量超过 4 个时，Solidity 如何管理函数选择器?

答：函数选择器是用来标识函数的 4 字节哈希值。当函数数量超过 4 个时，Solidity 会使用函数的哈希值来管理函数选择器。具体来说，Solidity 会将函数的哈希值与函数的签名进行映射，以便在编译时生成正确的函数选择器。在运行时，Solidity 会使用函数的哈希值来查找函数选择器，以便正确地调用函数。

3. 如果对一个合约进行委托调用，而该合约又对另一个合约进行委托调用，那么在代理合约、第一个合约和第二个合约中，msg.sender 是谁?

答：是调用代理合约的人，也就是代理合约中的msg.sender.

4. 如果有的话，ABI 编码在 calldata 和 memory 之间有何不同?

答：在 Solidity 中，ABI 编码是一种用于序列化和反序列化函数参数和返回值的标准格式。在函数调用时，函数参数被编码为 ABI 编码格式，并存储在 calldata 中。在函数内部，可以使用 msg.data 访问 calldata 中的数据。与之相反，memory 是 Solidity 中的一种数据位置，用于存储动态分配的内存。在函数内部，可以使用 memory 来存储和操作临时变量和数组。因此，ABI 编码和 memory 是两个不同的概念，它们之间没有直接的关系。

5. uint64 和 uint256 在 calldata 中的 ABI 编码有何不同?

答：在 Solidity 中，uint64 和 uint256 在 calldata 中的 ABI 编码有所不同。 当使用 uint64 时，编码器会将其填充到 32 字节，以与 EVM 的字长对齐。这样可以确保数据布局的协调，简化 EVM 中的操

作。当使用 uint256 时，编码器不需要额外的填充，因为它的字长与 EVM 的字长完全相同。可用 abi.encodePacked (number)，看一下uint64 number和 uint256 number的输出结果

6. 什么是只读重入?

答：只读重入是一种攻击方式，它利用了智能合约中的函数重入漏洞。攻击者会在调用一个只读函数时，通过在函数执行期间调用另一个函数来重入合约。由于只读函数不会修改合约状态，因此它们可以在调用期间被重入，而不会引起任何问题。然而，如果攻击者在重入期间调用了一个写入函数，那么他们就可以修改合约状态并窃取资金。

7. 为了防止只读重入攻击，开发者可以采取以下措施：

避免在只读函数中调用写入函数，以防止重入攻击。

使用 modifier 来限制只读函数的访问权限，以确保只有授权用户才能调用它们。

使用最新版本的 Solidity 编译器，因为它们包含了针对函数重入漏洞的修复。

8. 从不受信任的智能合约调用中读取（内存）字节数组的安全考虑是什么?

答：从不受信任的智能合约中读取（内存）字节数组可能会导致安全问题。攻击者可以通过构造恶意合约来读取合约中的敏感数据，例如私钥、密码等。为了防止这种情况发生：

使用 Solidity 中的 view 和 pure 函数来限制合约的访问权限。这些函数不会修改合约的状态，因此不会对合约中的数据造成任何影响。

使用 require 和 assert 函数来确保输入的数据符合预期，从而防止攻击者利用缺陷来读取合约中的数据。

使用加密技术来保护合约中的敏感数据，例如使用对称加密或非对称加密算法来加密数据。

9. 如果部署一个空的 Solidity 合约，在区块链上会有什么字节码?

答：部署一个空的 Solidity 合约，它将在区块链上占用一些空间，但不会有任何字节码。这是因为空合约不包含任何代码，因此它不需要在区块链上存储任何字节码。但是，即使是空合约也需要在区块链上分配地址，以便其他合约可以与它进行交互。因此，空合约将在区块链上占用一些空间，但这个空间非常小，可以忽略不计。

10. 以太虚拟机如何定价内存使用?

答：在以太坊中，内存使用的定价是动态的，它取决于当前内存使用情况。内存使用的定价是通过一种称为“内存燃料”的机制来实现的。内存燃料是一种虚拟的资源，它用于衡量内存使用的成本。在每个交易中，内存燃料的数量都是有限的，这意味着在使用大量内存时，交易的成本会相应地增加。内存燃料的价格是由矿工设置的，他们可以根据当前的内存使用情况来调整价格。EIP1559之后，价格由以太坊网络动态决定。

11. 智能合约的元数据部分存储了什么?

答：智能合约的元数据部分存储了一些关于合约的信息，例如合约的名称、版本、作者、编译器版本、编译时间等。这些信息可以帮助开发人员更好地理解和使用合约。此外，元数据还包括合约的ABI（应用程序二进制接口），它定义了与合约交互的方法和参数。具体可以看编译产生的_meta.json文件

12. 从 MEV 的角度来看，什么是叔块攻击?

答：从MEV（最大可提取价值）的角度来看，叔块攻击是指重新挖掘先前已经挖掘过的区块，以获取最大的套利机会并从协议的激励中获益，这可能会导致网络的显着不稳定性。攻击者可以通过在叔块中看到某些交易并将其前置来优化其收益，从而利用MEV。这种攻击可能会导致交易的顺序发生变化，从而影响交易的结果

13. 如何进行签名篡改攻击（malleability attack）?

答：签名篡改攻击是指攻击者通过更改交易的签名，从而生成一个新的交易，而不改变交易的有效性和语义。这种攻击可以导致交易的重放、取消或其他不良后果。攻击者可以通过多种方式进行签名篡改攻击，例如更改签名的字节、添加或删除签名的字节、更改签名的哈希值等。攻击者可以利用这些漏洞来欺骗合约，从而获得不当利益。

要进行签名篡改攻击，攻击者需要知道交易的签名，并且需要对签名进行修改。攻击者可以使用一些工具来执行这些操作，例如Bitcoin Core中的signrawtransaction命令。为了防止签名篡改攻击，开发人员可以使用一些技术，例如使用随机数生成签名、使用哈希函数对交易进行哈希、使用公钥加密等。这些技术可以增强交易的安全性，从而防止签名篡改攻击。

14. 在什么情况下，具有前导零的地址可以节省 gas，以及为什么?

答：在以太坊中，具有前导零的地址可以节省gas费用。这是因为在以太坊中，地址是由20个字节的哈希值表示的。如果地址的前几个字节为零，则可以省略这些零，从而减少交易的大小，进而减少gas费用。

例如，假设我们有一个地址为0x0000000000000000000000000000000000000000。如果我们将其转换为十六进制字符串，我们可以看到它的前12个字符都是零。在以太坊中，我们可以省略这些零，只使用0x0来表示该地址。这样，我们就可以减少交易的大小，从而减少gas费用。

需要注意的是，具有前导零的地址只能在特定情况下节省gas费用。如果地址的前几个字节不是零，那么省略这些字节将不会节省任何gas费用。此外，省略前导零可能会降低地址的可读性，因此需要在可读性和gas费用之间进行权衡。

15. `payable(msg.sender).call{value: value}("")` 和 `msg.sender.call{value: value}("")` 之间有什么区别?

答：payable(msg.sender).call{value: value}("")和msg.sender.call{value: value}("")之间的区别在于前者将msg.sender转换为address payable类型，而后者不会。在Solidity 0.8.0及更高版本中，msg.sender和tx.origin的类型都是address，而不是address payable。因此，如果您想使用msg.sender或tx.origin来接收以太币，您需要将它们转换为address payable类型，例如：payable(msg.sender).transfer(amount)或payable(tx.origin).transfer(amount)。

16. 一个字符串占用多少个存储槽?

答：字符串是动态大小的，因此其存储空间也是动态分配的。字符串的存储空间由其长度和一个额外的 32 字节的存储槽组成，用于存储字符串的长度。因此，字符串的存储空间可以计算为：`32 + (string_length + 31) / 32 * 32`

其中 string_length 是字符串的长度。例如，如果字符串的长度为 10，则其存储空间为 64 字节。请注意，这个公式只适用于字符串存储在状态变量中的情况。如果字符串存储在内存或 calldata 中，则不需要额外的存储槽。
字符串不超31个字节的占一个存储槽，最低位存储字符串长度，所以一个槽最多存31个字节。

17. Solidity 编译器中的--via-ir 功能是如何工作的?

答：--via-ir 是 Solidity 编译器的一个选项，它可以启用基于中间代码（IR）的代码生成器。使用 IR 代码生成器，Solidity 可以生成 Yul 中间代码，然后再将其转换为 EVM 字节码。这种方法的优点是，它可以实现更强大的跨函数优化通道，同时也使代码生成更加透明和可审计。

您可以在命令行中使用 --via-ir 或在 standard-json 中使用 {"viaIR": true} 选项来启用基于 IR 的编码。在 Hardhat 中，可以在 hardhat.config.js 的 settings 字段下加入配置，如：

```
solidity: {version: \"0.8.17\",settings: {\"viaIR\": true, //配置启用
IRoptimizer: {enabled: true,runs: 1000,
    },
  },
},
```

18. 函数修饰符是从右到左调用还是从左到右调用，还是不确定的?

答：Solidity中的函数修饰符是从右到左调用的，这是编译器决定的。

19. 如果对一个合约进行委托调用，而执行了指令 CODESIZE，将返回哪个合约的大小?

答：如果您对一个合约进行委托调用，并执行指令 CODESIZE，则返回的是被委托的合约的大小。CODESIZE指令返回的是指定合约的代码大小，而不是当前合约的大小。因此，如果您在一个合约中进行委托调用，CODESIZE指令将返回被委托的合约的大小，而不是当前合约的大小。

20. 为什么 ECDSA 对哈希而不是任意 bytes32 进行签名很重要

答：ECDSA是一种数字签名算法，它使用椭圆曲线加密来生成公钥和私钥。在以太坊中，ECDSA被用于验证交易的签名。ECDSA对哈希进行签名十分重要，而不是对任意的bytes32签名，因为这样可以避免安全漏洞。

如果ECDSA对任意的bytes32进行签名，那么攻击者可以构造一个新的bytes32，使得它的哈希值与原始bytes32的哈希值相同。这样，攻击者就可以使用原始签名来验证新的bytes32，从而导致安全漏洞。因此，ECDSA只对bytes32的哈希进行签名，以确保签名的安全性

21. 符号操作测试(symbolic manipulation testing)是如何工作的。

答：符号操作测试是一种基于符号计算的技术，用于在不实际执行程序的情况下探索程序的所有可能执行路径。在 Solidity 中，符号执行可以用于检测智能合约中的漏洞和错误。符号执行器将程序的输入参数表示为符号变量，然后通过解析程序的控制流图来生成程序的符号执行路径。这些路径可以用于检测程序中的漏洞和错误，例如整数溢出、未初始化的变量、未处理的异常等。

22. 复制内存区域的最有效方式是什么?

答:复制内存区域的最有效方式是使用 memcpy 函数。memcpy 函数可以将一个内存区域的内容复制到另一个内存区域。它的语法如下:

```
function memcpy(uint dest, uint src, uint len) internal {// Copy word-length chunks while possiblefor(; len= 32; len -= 32) {
        assembly {
            mstore(dest, mload(src))
        }
        dest += 32;
        src += 32;
    }
// Copy remaining bytesuint mask = 256 ** (32len) - 1;
    assembly {
        let srcpart := and(mload(src), not(mask))
        let destpart := and(mload(dest), mask)
        mstore(dest, or(destpart, srcpart))
    }
}
```

23. 如何在链上验证另一个智能合约是否触发了一个事件,而不使用预言机?

答:

产生事件合约:

```
pragma solidity ^0.8.0;

contract MyContract {event MyEvent(uint256 indexed id, string data);
function myFunction(uint256 id, string memory data) public {
        emit MyEvent(id, data);
    }
}
```

验证事件合约:

```
 pragma solidity ^0.8.0;

contract MyVerifier {function verifyEvent(address contractAddress, uint256 id, string memory data) public view returns (bool) {
        MyContract myContract = MyContract(contractAddress);uint256 fromBlock = block.number100;uint256 toBlock = block.number;bytes32 eventHash =
```

```
    keccak256(abi.encodePacked("MyEvent(uint256,string)"));bool eventFound =
    false;for (uint256 i = fromBlock; i <= toBlock; i++) {bytes32memory topics =
    new bytes32;
            topics[0] = bytes32(eventHash);
            topics[1] = bytes32(id);bytes memory data =
    abi.encodePacked(contractAddress);uint256memory eventIds =
    myContract.getPastEvents("MyEvent", i, i, data, topics);if (eventIds.length0) {
                eventFound = true;break;
            }
        }return eventFound;
    }
}
```

在上面的代码中，MyVerifier 合约包含一个名为 verifyEvent 的函数，该函数用于验证另一个智能合约是否触发了 MyEvent 事件。该函数接受三个参数：contractAddress 是要验证的智能合约地址，id 是要验证的事件 id 参数，data 是要验证的事件 data 参数。该函数使用 getPastEvents 函数来检索事件日志，并检查是否存在与指定参数匹配的事件。

24. 当调用 selfdestruct 时，以太何时转移？智能合约的字节码何时被擦除？

答：当调用 selfdestruct 函数时，合约账户上剩余的以太币会发送给指定的目标地址.、合约的存储和代码从状态中被移除，但是合约的地址仍然存在.。合约的字节码不会被擦除，但是合约的地址不再被使用，因此合约的代码也不再被执行。

25. 自由内存指针是什么？

答：自由内存指针是 Solidity 中的一个指针，它指向当前合约的内存空间中的下一个可用位置。在 Solidity 中，内存是一种临时存储空间，用于存储临时变量和函数参数。当您在 Solidity 中声明一个变量时，它将被分配到内存中，并且自由内存指针将被更新以指向下一个可用位置。在 Solidity 中，您可以使用 mload 和 mstore 操作码来读取和写入内存中的数据。自由内存指针是 Solidity 中的一个重要概念，因为它允许您在内存中动态分配空间，从而更有效地使用内存。

26. 在什么条件下，Openzeppelin 的 Proxy.sol 会覆盖自由内存指针？为什么这样做是安全的？

答：在 Proxy.sol 中，如果使用 delegatecall 调用另一个合约，那么在调用结束后，自由内存指针将被覆盖。这是因为 delegatecall 会将调用的合约的代码复制到当前合约的内存中，然后执行它。在执行期间，自由内存指针将被覆盖，因为它指向当前合约的内存空间。这是安全的，因为在 delegatecall 结束后，当前合约的状态不会被修改，而只会修改被调用的合约的状态。

27. 为什么 Solidity 废弃了"years"关键字？

答：在 Solidity 0.5.0 版本之前，Solidity 中有一个名为 “years” 的时间单位关键字。然而，由于闰年的存在，使用 “years” 作为时间单位会导致一些混淆和问题。因此，在 Solidity 0.5.0 版本中，Solidity 废弃了 “years” 关键字，建议使用 “days” 代替 “years” ，并将时间单位转换为天数。

例如，如果您想要表示一年的时间，您可以使用 “365 days” 来代替 “1 year”。这样做可以避免由于闰年而导致的时间计算错误。

28. verbatim 关键字的作用是什么，以及它可以在哪里使用?

答：智能合约中的 verbatim 关键字是 Solidity 语言的一部分。它的作用是将代码中的字符串字面量原样输出，而不进行转义或解析。这在编写智能合约时非常有用，因为它可以确保字符串的内容不会被意外地更改或破坏。例如，如果您需要在智能合约中编写一个包含 HTML 标记的字符串，那么使用 verbatim 关键字可以确保标记不会被解析或更改。verbatim 关键字可以在 Solidity 语言的任何地方使用，包括函数定义、变量声明和注释等。

29. 在调用另一个智能合约时可以转发多少 gas?

答：在调用另一个智能合约时，可以转发的 gas 数量取决于您的智能合约的 gas 限制和 gas 价格。如果您的智能合约的 gas 限制为 1000000，gas 价格为 20 Gwei，那么您可以转发的 gas 数量为 1000000。

30. 存储 -1 的 int256 变量在十六进制中是什么样子的?

答：0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

31. signextend 操作码有什么用?

答：用于将有符号整数的位数扩展到更高的位数。在 Solidity 中，有符号整数使用补码表示法，其中最高位表示符号位。当您使用 signextend 操作码时，它将检查有符号整数的符号位，并将其扩展到更高的位数。如果符号位为 1，则 signextend 操作码将在高位添加 1；否则，它将在高位添加 0。这样做可以确保有符号整数的符号位在扩展后保持不变。

32. 为什么 calldata 中的负数会消耗更多的 gas?

答：当您在 Solidity 中使用 calldata 传递负数时，Solidity 会将其转换为补码形式，并将其存储在 calldata 中。由于补码形式需要更多的位来表示负数，因此在 calldata 中存储负数需要更多的空间，从而消耗更多的 gas。这是因为在 Solidity 中，gas 的消耗是与数据大小成正比的。因此，存储更大的数据将导致更多的 gas 消耗。

33. 什么是 zk-friendly 哈希函数，它与非 zk-friendly 哈希函数有何不同?

答：zk-friendly 哈希函数是一种特殊的哈希函数，它具有一些特殊的性质，使其适用于零知识证明场景。与传统的哈希函数不同，zk-friendly 哈希函数通常具有以下特点：

- 低计算复杂度：zk-friendly 哈希函数通常具有较低的计算复杂度。
- 低内存占用：zk-friendly 哈希函数通常具有较低的内存占用。
- 低交互性：zk-friendly 哈希函数通常具有较低的交互性。
  与传统的哈希函数相比，zk-friendly 哈希函数通常具有更好的性能和更好的安全性。这使得它们在零知识证明中更容易使用，并且可以提供更好的隐私保护。

34. 在零知识的背景下，什么是 nullifier，它的用途是什么?

答：在零知识证明中，nullifier 是一个用于防止交易被重复使用的值。当一个交易被执行时，它会生成一个 nullifier 值，该值与交易相关联。如果交易被重复使用，nullifier 值将被公开，从而使交易无

效。这种方法可以确保数字货币只能被使用一次，从而防止数字货币被重复支付。