مستند سالیدیتی

نسخه 0.8.4

این بخش از مستند توسط سارا باوی فرد ترجمه شده است.

راه ارتباطی جهت بهبود داکیومنت

mailto:soliditylang.fa@gmail.com

سالیدیتی[1] یک زبان شیء گرا[2] و سطح بالا برای پیاده سازی قراردادهای هوشمند[3] می‌باشد. قراردادهای هوشمند، برنامه‌هایی هستند که رفتار حساب‌ها[4] در داخل حالت اتریوم[5] را کنترل می‌کنند.

سالیدیتی یک زبان آکلادی[6] می‌باشد که از زبان‌هایی مانند سی پلاس پلاس[7]، پایتون و جاوا اسکریپت تأثیر گرفته و برای هدف قراردادن EVM یا ماشینِ مجازیِ اتریوم[8] طراحی شده‌است.

سالیدیتی از نوع استاتیک[9] می‌باشد. از ویژگی‌های ارث بری[10]، کتابخانه‌ها[11] و انواع نوع‌های پیچیده تعریف شده توسط کاربر[12] پشتیبانی می‌کند.

با سالیدیتی می‌توانید قراردادهایی را برای کاربردهایی از قبیل رأی‌گیری[13]، سرمایه گذاری جمعی[14]، مزایده کور[15] و کیف پول با امضای چندگانه[16] استفاده کنید.

هنگام استقرار[17] قراردادها، باید از آخرین نسخه سالیدیتی منتشر شده استفاده کنید. به این دلیل که تغییرات جدید[18]، ویژگی‌های جدید[19] و رفع باگ‌ها[20] به طور منظم معرفی می‌شوند. ما در حال حاضر از نسخه 0.X برای نشان دادن این تغییرات سریع استفاده می‌کنیم.

هشدار
سالیدیتی به تازگی نسخه 0.8.X را منتشر کرده که تغییرات جدید را معرفی می‌کند. حتماً لیست کامل را مطالعه کنید.

[1] Solidity
[2] object-oriented
[3] smart contacts
[4] accounts
[5] Ethereum state
[6] curly-bracket language
[7] C++
[8] Ethereum Virtual Machine (EVM)
[9] statically typed
[10] Voting
[11] libraries
[12] complex user-defined types
[13] voting
[14] crowdfunding
[15] blind auctions
[16] multi-signature wallets
[17] deploy
[18] breaking changes
[19] new features
[20] bug fixes

ایده‌های بهبود سالیدیتی یا این مستند همیشه مورد استقبال قرار میگیرد، برای جزئیات بیشتر راهنمای همکاری را مطالعه کنید.

شروع

1. درک مبانی قراردادهای هوشمند

اگر با مفهوم قراردادهای هوشمند آشنا هستید، به شما توصیه می‌کنیم که با جستجوی بخش "معرفی قراردادهای هوشمند" شروع به کار کنید، که شامل موارد زیر است:

- یک مثال ساده از قرارداد هوشمند که با سالیدیتی نوشته شده‌است.
- مبانی بلاکچین.
- ماشین مجازی اتریوم.

2. آشنایی با سالیدیتی

هنگامی که با مبانی اولیه آشنا شدید، توصیه می‌کنیم برای درک مفاهیم اصلی زبان، بخش‌های "سالیدیتی با مثال" و "شرح زبان" را بخوانید.

3. نصب کامپایلر[1] سالیدیتی

روش های مختلفی برای نصب کامپایلر سالیدیتی وجود دارد، به سادگی گزینه مورد نظر خود را انتخاب کنید و مراحل ذکر شده در صفحه نصب را دنبال کنید.

[1] Compiler

تذکر:

می‌توانید نمونه‌های کد را مستقیماً در مرورگر خود با ویرایشگر کد ریمیکس[1] امتحان کنید. ریمیکس یک ویرایشگر کد مبتنی بر مرورگر وب است که به شما امکان می‌دهد، بدون نیاز به نصب سالیدیتی به صورت محلی، قراردادهای هوشمند سالیدیتی را بنویسید، دیپلوی و مدیریت کنید.

هشدار:

نرم افزار به عنوان نوشته‌ی انسان، می‌تواند باگ داشته باشد. هنگام نوشتن قراردادهای هوشمند خود، باید بهترین شیوه‌های توسعه نرم افزار را دنبال کنید. شیوه‌های توسعه نرم افزار شامل بازبینی[2]، آزمایش[3]، حسابرسی[4] و اثبات صحتِ[5] کد می‌باشد. کاربران قرارداد هوشمند گاهی اوقات به خودِ کد نسبت به نویسندگان آن‌ها اطمینان بیشتری دارند. بلاکچین‌ها و قراردادهای هوشمند مسائل منحصر به فرد خود را دارند، که باید مراقب آنها باشید. بنابراین قبل از کار بر روی تولید کد، حتماً قسمت ملاحظات امنیتی را مطالعه کنید.

4. یادگیری بیشتر

اگر می‌خواهید در مورد ساخت برنامه‌های غیرمتمرکز در اتریوم اطلاعات بیشتری کسب کنید، منابع توسعه دهنده اتریوم می‌توانند به شما در تهیه مستندِ عمومیِ بیشتر در مورد اتریوم و انتخاب گسترده‌ای از آموزش‌ها، ابزارها و چارچوب‌های توسعه کمک کنند.

[1] Remix IDE
[2] review
[3] Testing
[4] audits
[5] correctness proofs

اگر سؤالی دارید، می‌توانید جواب‌ها را جستجو کنید یا از طریق Ethereum StackExchange یا کانال Gitter ما بپرسید.

فصل دوم

ترجمه‌ها

داوطلبان جامعه سالیدیتی به ترجمه این مستند به چندین زبان کمک می‌کنند. این سندها سطوح مختلفی از کامل و بروز بودن را دارند. نسخه انگلیسی به عنوان مرجع می‌باشد.

- فرانسوی (در حال انجام)
- ایتالیایی (در حال انجام)
- ژاپنی
- کره‌ای (در حال انجام)
- روسی (نسبتاً قدیمی)
- چینی ساده شده (در حال انجام)
- اسپانیایی
- ترکی (جزئی)

فصل سوم

محتوا

3.1 مقدمه‌ای بر قرادهای هوشمند

3.1.1 یک قرارداد هوشمند ساده

بیایید با یک مثال ابتدایی شروع کنیم که مقدار یک متغیر را تعیین می‌کند و آن را در معرض دسترسی سایر قراردادها قرار می‌دهد. اینکه الان شما متوجه چیزی نمی‌شوید طبیعی می‌باشد، بعداً به جزئیات بیشتری خواهیم پرداخت.

مثال ذخیره سازی

```
// 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) {
        return storedData;
    }
}
```

خط اول به شما می‌گوید کد منبع[1]، تحت مجوز GPL نسخه 3.0 می‌باشد. در جایی که انتشار کد منبع به صورت پیشفرض[2] باشد، مشخص کننده‌هایِ مجوز قابلِ خواندن توسطِ ماشین[3] مهم هستند.

[1] source code
[2] default
[3] Machine-readable license specifiers

خط بعدی مشخص می‌کند کد منبع برای سالیدیتی نسخه 0.4.16 یا نسخه جدیدتر زبان نوشته شده است، اما شامل نسخه 0.9.0 نمی‌باشد. بخاطر اینکه قرارداد هوشمند می‌تواند رفتار متفاوتی داشته باشد، به این هدف که اطمینان حاصل شود قرارداد هوشمند نتواند با نسخه جدید کامپایلر، کامپایل شود. پراگما (Pragmas) دستورالعمل‌های رایج برای کامپایلرها در مورد نحوه برخورد با کد منبع می‌باشند (به عنوان مثال پراگما یکبار(pragma once)).

قراردادها در سالیدتی به معنای مجموعه‌ای از کد (توابع آن‌ها) و داده‌ها (حالت آن‌ها) است که در یک آدرس خاص در بلاکچین اتریوم قرار دارند. خط `uint storedData;` یک متغیر حالت[1] به نام `storedData` از نوع `uint` (عددِ صحیحِ بدونِ علامت[2] 256 بیتی) را مشخص می‌کند. می‌توان آن را به عنوان یک اسلات[3] در پایگاه داده در نظر بگیرید که می‌توانید با فراخوانی توابع کدی که پایگاه داده را مدیریت می‌کند، آن‌ها را جستوجو کرده و ویرایش کنید. در این مثال، قرارداد توابع `set` و `get` را تعریف می‌کند که برای ویرایش[4] یا بازیابی[5] مقدار متغیر استفاده شود.

همانند سایر زبان‌های رایج، برای دسترسی به یک متغیر حالت، نیازی به پیشوند `this.` ندارید. این قرارداد جدا از اینکه هنوز کار زیادی انجام نداده‌است (به دلیل زیرساخت‌های ساخته شده توسط اتریوم) اجازه می‌دهد هر کس یک عدد را ذخیره کند، که این عدد توسط هر کسی در دنیا بدون هیچ روش امکان پذیر برای جلوگیری از انتشار آن قابل دسترس می‌باشد. هر کسی می‌تواند مجدداً تابع `set` را فراخوانی کند و عدد را رونویسی کند، اما عدد در تاریخچهِ بلاکچین هنوز ذخیره بماند. بعداً خواهید دید که چگونه می‌توانید محدودیت‌های دسترسی را اعمال کنید تا فقط شما بتوانید عدد را تغییر دهید.

---

[1] state variable
[2] unsigned integer
[3] slot
[4] modify
[5] retrieve

هشدار
در هنگام استفاده از متن Unicode مراقب باشید، زیرا نویسه‌های[1] مشابه (یا حتی یکسان) می‌توانند دارای نکته‌های کدی[2] متفاوتی باشند و همینطور به عنوان یک آرایه بایت[3] متفاوت کدگذاری شوند.

توجه داشته باشید
همه شناسه‌ها[4] (نام قرارداد، نام تابع و نام متغیر) به مجموعه کاراکترهای ASCII محدود می‌شوند. ذخیره داده‌های رمزگذاری شده UTF-8 در متغیرهای رشته‌ای[5] امکان پذیر است.

مثال ساب کارنسی یا زیر ارز[6]

قرارداد زیر ساده ترین شکل ارز رمزنگاری شده[7] را پیاده سازی می‌کند. این قرارداد فقط به سازنده آن اجازه می‌دهد سکه[8]های جدید ایجاد کند (طرح‌های مختلف صدور امکان پذیر است). هر کسی می‌تواند بدون نیاز به ثبت نام با نام کاربری و رمز عبور، سکه برای یکدیگر ارسال کنند، تمام آنچه شما نیاز دارید یک جفت کلید اتریوم است.

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

contract Coin {
    // The keyword "public" makes
    variables // accessible from other
```

[1] characters
[2] code points
[3] byte array
[4] identifier
[5] string
[6] Subcurrency
[7] cryptocurrency
[8] coin

```
    contracts address public minter;
    mapping (address => uint) public
    balances;

    // Events allow clients to react to specific
    // contract changes you declare event
    Sent(address from, address to, uint amount);

    // Constructor code is only run when the contract
    // is created
    constructor() {
    minter =
    msg.sender;
    }

    // Sends an amount of newly created coins to an address
// Can only be called by the contract creator function mint(address receiver, uint
amount) public { require(msg.sender == minter); require(amount < 1e60);
        balances[receiver] += amount;
    }

    // Errors allow you to provide information about
    // why an operation failed. They are returned // to
    the caller of the function. error
    InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins // from any
    caller to an address 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); }
}
```

این قرارداد مفاهیم جدیدی را معرفی می‌کند، اجازه دهید یکی یکی آنها را مرور کنیم.

خط `address public minter;` متغیر حالت از نوع `address` را مشخص می‌کند. نوع آدرس یک مقدار 160 بیتی است که اجازه هیچ گونه عملیات حسابی را نمی‌دهد. متغییر `address` برای ذخیره آدرس قراردادها یا یک هش از نیمه عمومیِ[1] یک جفت کلید[2] متعلق به حساب‌های خارجی[3] مناسب است.

کلمه کلیدی `public` به طور خودکار تابعی را ایجاد می‌کند که به شما امکان می‌دهد، از خارج از قرارداد به مقدار فعلی متغیر حالت[4] دسترسی پیدا کنید. بدون این کلمه کلیدی، سایر قراردادها راهی برای دسترسی به متغیر ندارند. کد تابع که توسط کامپایلر تولید می‌شود معادل موارد زیر است (فعلاً از `external` و `view` چشم پوشی کنید):

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

می‌توانید توابعی مانند موارد فوق را خود اضافه کنید، اما یک متغیرحالت و تابعی با همان نام خواهید داشت. اما نیازی به این کار نیست، کامپایلر آن را برای شما مشخص می‌کند.

---

[1] public half
[2] keypair
[3] external accounts
[4] state variable

خط بعدی، `mapping (address => uint) public balances;` یک متغیرِ حالتِ عمومی[1] ایجاد می‌کند، اما یک نوع داده[2] پیچیده‌تر است. نوع mapping آدرس‌ها را به اعداد صحیح بدون علامت[3] (unsigned integers) نگاشت[4] می‌کند.

Mapping[5]ها را می‌توان به عنوان جداول هش[6] مشاهده کرد که عملاً مقداردهی اولیه شده‌اند، به طوری که همه کلیدهای ممکن از همان ابتدا وجود داشته و به مقداری که همهِ نمایشِ بایت[7] آن‌ها صفر است نگاشت شده باشند. با این حال، نه می‌توان لیستی از تمام کلیدهای Mapping و نه لیستی از تمام مقادیر را بدست آورد. آنچه را که به Mapping اضافه کرده‌اید، ثبت کنید یا از آن در زمینه‌ای که نیازی به آن مقدار نیست استفاده کنید. یا حتی بهتر است که یک لیست نگهدارید یا از نوع داده[8] مناسب استفاده کنید.

تابع getter ایجاد شده توسط کلمه کلیدی `public` در Mapping پیچیده‌تر است. به شرح زیر است:

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

شما می‌توانید برای جستجوی بالانس[9] یک حساب از این تابع استفاده کنید.

خط `event Sent(address from, address to, uint amount);` یک "event" را مشخص می‌کند که در آخرین خط با تابع `send` منتشر می‌شود. کلاینت اتریوم مانند برنامه‌های کاربردی[10] وب می‌توانند به این رویداد[11]ها که در بلاکچین منتشر شده‌اند، بدون هزینه زیاد گوش دهند. شنونده

---

[1] public state variable
[2] datatype
[3] unsigned integers
[4] map
[5] نگاشت
[6] hash tables
[7] byte-representation
[8] data type
[9] balance
[10] web applications
[11] event

به محض انتشار، آرگومان‌های `from` ، `to` ، `amount` را دریافت می‌کند، که امکان ردیابی تراکنش‌ها را فراهم می‌کند.

برای گوش دادن به این event، می‌توانید از کد جاوا اسکریپت زیر استفاده کنید که از web3.js برای ایجاد شیء قرارداد[1] `Coin` استفاده می‌کند و هر رابط کاربری تابع `balances` که به صورت خودکار ایجاد شده را از بالا فراخوانی می‌کند:

```
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 یک تابع خاص است که در هنگام ایجاد قرارداد اجرا می‌شود و پس از آن نمی‌توان آن را فراخوانی کرد. در این مورد، constructor آدرس شخص ایجاد کننده قرارداد را برای همیشه ذخیره می‌کند. متغیر `msg` (همراه با `tx` و `block`) یک متغیر جهانی خاص[2] که شامل خصوصیاتی می‌باشد و امکان دسترسی به بلاکچین را فراهم می‌کند. `msg.sender` همیشه آدرسی است که فراخوانی تابع فعلی (خارجی[3]) از آن گرفته شده‌است.

[1] contract object [2] special global variable [3] external

توابعی که قرارداد را تشکیل می‌دهند و کاربران و قراردادها می‌توانند آنها را فراخوانی کنند، `mint` و `send` هستند.

تابع `mint` مقداری سکه تازه ایجاد شده را به آدرس دیگری می‌فرستد. فراخوانی تابع require شرایطی را تعریف می‌کند که در صورت عدم تحقق همه تغییرات را برمی‌گرداند[1]. در این مثال، `require(msg.sender == minter);` تضمین می‌کند که فقط سازنده قرارداد می‌تواند با `mint` قرارداد را فراخوانی کند و `require(amount < 1e60);` حداکثر مقدار توکن[2]ها را تضمین می‌کند. این مورد اطمینان می‌دهد که در آینده هیچ خطای سرریز[3] وجود نخواهد داشت.

خطاها به شما امکان می‌دهند اطلاعات بیشتری در مورد علت شرایط یا شکست عملیات به فراخوانی کننده[4] ارائه دهید. خطاها همراه با دستورات revert استفاده می‌شوند. دستورات revert بدون تغییر و بدون قید و شرط، تمام تغییرات مشابه با تابع `require` را نابود و برمی‌گردانند[5]، اما همچنین به شما امکان ارائه نام خطا و داده‌های اضافی را که به فراخوانی کننده (و در نهایت به برنامه سمت کاربر[6] یا جستوجوگر بلاک[7]) نشان دهید، را می‌دهند. به طوری که یک شکست[8] را می‌توان به راحتی عیب یابی[9] کرد یا به آن واکنش نشان داد.

تابع `send` می‌تواند توسط هر کسی (که قبلاً برخی از این سکه‌ها را در اختیار داشته است) برای ارسال سکه به شخص دیگر استفاده شود. اگر فرستنده، سکه کافی برای ارسال نداشته باشد، فراخوانی `require` شکست می‌خورد و یک رشته پیام خطای[10] مناسب برای فرستنده ارائه می‌دهد.

توجه داشته باشید

[1] revert
[2] token
[3] overflow
[4] caller
[5] revert
[6] front-end
[7] block explorer
[8] failure
[9] debug
[10] error message string

اگر از این قرارداد برای ارسال سکه به یک آدرس استفاده کنید، وقتی به آن آدرس در یک مرورگر بلاکچین نگاه کنید، چیزی مشاهده نخواهید کرد. زیرا تاریخچه ارسال سکه و بالانس تغییر یافته و فقط در فضای ذخیره سازی داده[1] این قراردادِ خاصِ سکه ذخیره می‌شود. با استفاده از رویداد[2]ها، می‌توانید یک "جستجوگر بلاکچینی" ایجاد کنید که تراکنش‌ها و بالانس‌های سکه جدید شما را ردیابی می‌کند، اما باید آدرس قرارداد سکه و نه آدرس صاحبان سکه را بررسی کنید.

3.1.2 مبانی بلاکچین

درک بلاکچین به عنوان یک مفهوم برای برنامه نویسان خیلی دشوار نمی‌باشد. به این دلیل که بیشتر پیچیدگی در مفهوم (استخراج[3]، هش کردن[4]، رمزنگاری منحنی بیضوی[5]، شبکه های همتا به همتا[6] و غیره) می‌باشد که فقط برای ارائه مجموعه خاصی از ویژگی‌ها و وعده‌ها برای پلتفرم می‌باشد. پس از پذیرش این ویژگی‌ها، دیگر لازم نیست نگران فناوری زیر ساخت باشید - یا برای استفاده از آن باید بدانید که AWS آمازون چگونه کار می‌کند؟

تراکنش‌ها[7]

بلاکچین یک پایگاه داده تراکنشی[8] مشترک جهانی[9] است. این بدان معناست که هر کس فقط با شرکت در شبکه می‌تواند ورودی‌های پایگاه داده را بخواند. اگر می‌خواهید چیزی را در پایگاه داده تغییر دهید، باید به اصطلاح، تراکنش ایجاد کنید، باید توسط دیگران پذیرفته شود. کلمه تراکنش به تغییری که می‌خواهید ایجاد کنید که یا اصلاً انجام نشده یا کاملاً اعمال شده اشاره دارد (فرض کنید می‌خواهید همزمان دو مقدار را تغییر دهید). علاوه بر این، زمانی که تراکنش شما در پایگاه داده اعمال می‌شود، هیچ تراکنش دیگری نمی‌تواند آن را تغییر دهد.

---

[1] data storage
[2] event
[3] Mining
[4] hashing
[5] elliptic-curve cryptography
[6] peer-to-peer network
[7] Transaction
[8] transactional database
[9] globally shared

به عنوان مثال، جدولی را تصور کنید که بالانس تمام حساب‌ها را در یک ارز الکترونیکی[1] فهرست می‌کند. اگر انتقال از یک حساب به حساب دیگر درخواست شود، ماهیت تراکنشی پایگاه داده تضمین می‌کند که اگر مبلغ از یک حساب کم شود، همیشه به حساب دیگر اضافه می‌شود. اگر به هر دلیلی، افزودن مبلغ به حساب مقصد[2] امکان پذیر نباشد، حساب مبدأ[3] نیز ویرایش نمی‌شود.

علاوه بر این، یک تراکنش همیشه به صورت رمزنگاری توسط فرستنده (سازنده[4]) امضا می‌شود. این امر باعث می‌شود محافظت از دسترسی به تغییرات خاص پایگاه داده آسان باشد. در مثال ارز الکترونیکی[5]، یک بررسی ساده تضمین می‌کند که فقط شخصی که کلیدهای حساب را دارد می‌تواند از آن پول انتقال بدهد.

بلاک‌ها[6]

یک مانع عمده برای غلبه بر چیزی که (در اصطلاحات بیتکوین) "حمله دو بار خرج کردن[7] " نامیده می‌شود : اگر دو تراکنش در شبکه وجود داشته باشد که هر دو بخواهند یک حساب را خالی کنند چه اتفاقی می‌افتد؟ فقط یکی از تراکنش‌ها می‌تواند معتبر باشد، به طور معمول تراکنشی که ابتدا پذیرفته می‌شود. مسئله این است که "First" یک اصطلاح عَملی در یک شبکه همتا به همتا[8] نیست.

پاسخ خلاصه این است که شما نیاز ندارید مراقبت باشید. یک ترتیب از تراکنش‌های پذیرفته شده به صورت جهانی برای شما انتخاب می‌شود، که اختلافات را حل می‌کند. تراکنش‌ها به صورت چیزی که "بلاک" نام دارد، بسته و سپس اجرا می‌شوند و در بین گره‌های مشارکت کننده توزیع می‌شوند. اگر دو تراکنش با یکدیگر مغایرت داشته باشند، تراکنشی که در نهایت دوم شود رد می‌شود و بخشی از بلاک نمی‌شود.

---

[1] electronic currency
[2] target account
[3] source account
[4] creator
[5] electronic currency
[6] Blocks
[7] double-spend attack
[8] peer-to-peer network

این بلاک‌ها از نظر زمانی یک توالی خطی[1] را تشکیل می‌دهند و بخاطر همین است که کلمه "بلاکچین" از آن گرفته می‌شود. بلاک‌ها در فواصل نسبتاً منظمی به زنجیره اضافه می‌شوند - برای اتریوم تقریباً هر 17 ثانیه می‌باشد.

به عنوان بخشی از "مکانیزم انتخاب ترتیبی[2] " (که "استخراج[3]" نامیده می‌شود) ممکن است فقط در "نوک[4]" زنجیره، برگرداندن بلاک‌ها هرزگاهی اتفاق بیفتد. هرچه تعداد بلاک‌های اضافه شده در بالای یک بلاکِ خاص بیشتر باشد، احتمال برگردانندن آن بلاک کمتر است. بنابراین ممکن است تراکنش شما برگردانده شود و حتی از بلاکچین حذف شود، اما هرچه بیشتر منتظر بمانید، احتمال آن کمتر است.

توجه داشته باشید
تضمین نمی‌شود که تراکنش در بلاک بعدی یا هر بلاک مشخص خاص در آینده لحاظ شود، زیرا این کار به عهده ارسال کننده نمی‌باشد، بلکه ماینرها باید تعیین کنند که تراکنش در کدام بلاک لحاظ شود. اگر می‌خواهید فراخوانی قراردادتان را در آینده زمان بندی کنید، می توانید از alarm clock یا سرویس اوراکل مشابه استفاده کنید.

3.1.3 ماشین مجازی اتریوم

مرور کلی

ماشین مجازی اتریوم[5] یا EVM محیط زمان اجرای قراردادهای هوشمند در اتریوم می‌باشد. EVM نه تنها یک جعبه است بلکه در واقع کاملاً ایزوله شده است، به این معنی که کدی که در داخل EVM اجرا می شود، به شبکه

[1] linear sequence
[2] order selection mechanism
[3] mining
[4] tip
[5] Ethereum Virtual Machine

و فایل سیستم[1] یا سایر فرآیندها دسترسی ندارد. در قراردادهای هوشمند حتی دسترسی به سایر قراردادهای هوشمند محدود است.

حساب‌ها

در اتریوم دو نوع حساب وجود دارد که فضای آدرس یکسانی را به اشتراک می‌گذارند: حساب های خارجی[2] که توسط جفت کلید عمومی-خصوصی[3] (به عنوان مثال انسان) کنترل می‌شوند و حساب‌های قراردادی[4] که توسط کدی که همراه با حساب ذخیره می‌شود کنترل می‌شوند.

آدرس یک حساب خارجی[5] از کلید عمومی مشخص می‌شود در حالی که آدرس قرارداد در زمان ایجاد قرارداد مشخص می‌شود (از آدرس سازنده و تعداد تراکنش‌های ارسال شده از آن آدرس مشتق گرفته شده است، که به اصطلاح " نانس[6]" نامیده می‌شود).

صرف نظر از اینکه حساب، کد را ذخیره می‌کند یا نه، EVM با این دو نوع حساب به صورت برابر رفتار می‌کند.

هر حساب دارای یک حافظه کلید-مقدار[7] ثابت است که کلمات 256 بیتی را با کلمات 256 بیتی که فضای ذخیره سازی[8] نامیده می‌شود.

علاوه بر این، هر حساب بالانسی معادل به اتر دارد (به طور دقیق در "wei"، معادل `1 ether` است) که می‌تواند با ارسال تراکنش‌هایی که شامل اتر هستند، اصلاح شود.

تراکنش‌ها

تراکنش پیامی است که از یک حساب به حساب دیگر ارسال می‌شود (که ممکن است یکسان یا خالی باشد، به مطالب زیر مراجعه کنید). تراکنش‌ها می‌تواند شامل داده‌های باینری (که "پیلود[9]" نامیده می‌شود) و اتر باشند.

---

[1] filesystem
[2] External accounts
[3] public-private key pairs
[4] contract accounts
[5] External accounts
[6] nonce
[7] key-value store
[8] storage
[9] payload

اگر حساب مقصد حاوی کد باشد، آن کد اجرا می شود و پیلود به عنوان داده ورودی ارائه می‌شود.

اگر حساب مقصد[1] تنظیم نشده باشد (یعنی تراکنش گیرنده نداشته باشد یا `null` تنظیم شده باشد)، تراکنش قرارداد جدید ایجاد می‌کند. همانطور که قبلاً ذکر شد، آدرس آن قرارداد آدرس صفر[2] نمی‌باشد، بلکه آدرسی است که از فرستنده و تعداد تراکنش‌های ارسال شده آن ("نانس") گرفته شده است. پیلود چنین تراکنش ایجاد قرارداد به عنوان بایت کد[3] EVM در نظر گرفته شده و اجرا می‌شود. داده‌های خروجی این اجرا به عنوان کدِ قرارداد برای همیشه ذخیره می‌شود. این بدان معناست که برای ایجاد قرارداد، شما کد واقعی قرارداد را ارسال نمی‌کنید، بلکه در واقع کدی که هنگام اجرا، آن کد را برمی‌گرداند[4].

توجه داشته باشید
هنگامی که یک قرارداد در حال ایجاد باشد، کد آن هنوز خالی است. به همین دلیل، تا زمانی که سازنده آن، اجرای آن را به اتمام نرسانده، نباید مجدداً قرارداد در حال ساخت را فراخونی مجدد کنید.

گاز[5]

به محض ایجاد، هر تراکنش با مقدار مشخصی گاز شارژ می شود، هدف آن محدود کردن میزان کار مورد نیاز برای انجام تراکنش و پرداخت همزمان هزینه این اجرا است. هنگامی که EVM تراکنش را انجام می‌دهد، گاز طبق قوانین خاص به تدریج خالی می‌شود.

هزینه گاز مقداری است که توسط سازنده تراکنش تنظیم می‌شود و باید `gas_price * gas` را قبل از حساب ارسالی پرداخت کند. اگر پس از اجرا مقداری گاز باقی مانده باشد، به همان روش به سازنده بازپرداخت می‌شود.

---

[1] target account
[2] zero address
[3] bytecode
[4] return
[5] Gas

اگر گاز در هر نقطه مصرف شود (یعنی منفی باشد)، یک استثناء اتمام گاز ایجاد می‌شود، که تمام تغییرات و اصلاحات ایجاد شده در حالت[1] را در این چارچوب فراخوانی فعلی، باز می‌گرداند[2].

فضای ذخیره سازی[3] ، حافظه مِمُوری[4] و پشته[5]

ماشین مجازی اتریوم دارای سه فضا برای ذخیره سازی داده می‌باشد: فضای ذخیره سازی یا storage، حافظه موقت یا مِمُوری و پشته، که در پاراگراف‌های زیر توضیح داده شده‌اند.

هر حساب دارای یک فضای داده به نام storage می‌باشد، که بین تابع فراخوانی و تراکنش‌ها ثابت است. storage یک حافظه کلید_مقدار[6] می‌باشد که کلمات 256 بیتی را با کلمات 256 بیتی نگاشت[7] می‌کند. محاسبه storage از طریق قرارداد امکان پذیر نیست، خواندن آن نسبتاً پر هزینه است و برای مقدار دهی و ویرایش کردن پر هزینه‌تر هم می‌باشد. به دلیل این هزینه‌ها، شما باید آنچه را که در storage ذخیره می‌کنید، به میزانِ نیازِ قرارداد برای انجام این کار به حداقل برسانید. . داده‌هایی مانند محاسبه مشتق، کش کردن[8] و جمع آوری داده‌های خارج از قرارداد، را در آن ذخیره کنید. یک قرارداد نمی‌تواند در هر storage ای به جز storage خودش بنویسد یا بخواند.

دومین فضای داده مِمُوری[9] نامیده می‌شود، که هر قرارداد یک نمونهِ پاکِ تازه برای هر فراخونی پیام بدست می‌آورد. مِمُوری خطی است و می‌توان آن را در سطح بایت آدرس دهی کرد، اما خواندن مِمُوری به عرض 256 بیت محدود می‌شود، در هنگام نوشتن می‌تواند 8 بیت یا 256 بیت عرض داشته باشد. هنگام دسترسی (خواندن یا نوشتن) به یک کلمه مِمُوری که قبلا دست نخورده است (یعنی هر آفست[10] درون یک کلمه)، مِمُوری با یک

---

[1] state
[2] revert
[3] Storage
[4] Memory
[5] the Stack
[6] key-value store
[7] maps
[8] caching
[9] memory
[10] offset

کلمه (256 بیتی) گسترش می‌یابد. در زمان گسترش، هزینه گاز باید پرداخت شود. حافظه هرچه بزرگتر شود گرانتر است (هزینه گاز به صورت درجه دو اندازه گیری می‌شود).

EVM یک ماشین ثبت نیست بلکه یک ماشین پشته است، بنابراین تمام محاسبات در یک فضای داده به نام پشته انجام می‌شود. حداکثر اندازه آن 1024 عنصر است و شامل کلمات 256 بیتی است. دسترسی به پشته به روش "بالا انتها" که در زیر توضیح داده شده‌است، محدود می‌باشد:

امکان کپی کردن یکی از 16 عنصر بالاتر در بالای پشته یا تعویض بالاترین عنصر با یکی از 16 عنصر زیر آن وجود دارد. تمام عملیات دیگر دو عنصر بالاتر (یا یک یا بیشتر، بسته به عملیات) را از پشته گرفته و نتیجه را در پشته درج میکنند. مطمئناً امکان دسترسی عناصر پشته به storage یا مِمُوری برای دستیابی عمیق تر به پشته وجود دارد، اما دسترسی به عناصر دلخواه در اعماق پشته بدون برداشتن قسمت بالای پشته امکان پذیر نیست.

مجموعه دستورالعمل‌ها

مجموعه دستورالعمل‌های EVM برای جلوگیری از پیاده سازی‌های نادرست یا ناسازگار که می‌توانند مشکلات اجماعی[1] ایجاد کنند، در حداقل نگه داشته می‌شوند. همه دستورالعمل‌ها بر اساس نوع داده اصلی[2]، کلمات 256 بیتی یا برشی از مِمُوری[3] (یا سایر آرایه‌های بایت[4]) کار می‌کنند. عملیات حسابی[5]، بیتی[6]، منطقی[7] و مقایسه‌ای [8] معمول وجود دارد. پرش‌های شرطی و غیر شرطی[9] امکان پذیر است. علاوه بر این، قراردادها می‌توانند به ویژگی‌های مربوط به بلاک فعلی مانند شماره[10] و برچسب زمان[11] آن دسترسی داشته باشند. برای لیست کامل، لطفاً به لیست آپکد به عنوان بخشی از مستند اسمبلی داخلی[12] مراجعه کنید.

---

[1] consensus problems
[2] basic data type
[3] slices of memory
[4] byte arrays
[5] arithmetic
[6] bit
[7] logical
[8] comparison
[9] Conditional and unconditional jumps
[10] number
[11] timestamp
[12] inline assembly

پیام های فراخوانی[1]

قراردادها می‌توانند سایر قراردادها را فراخوانی کنند یا اتر را از طریق پیام‌های فراخوانی به حساب‌های غیر قراردادی بفرستند. پیام‌های فراخوانی مانند تراکنش‌ها هستند، بدین معنی که دارای منبع[2]، مقصد[3]، پیلود داده[4]، اتر[5]، گاز[6] و داده‌های برگشتی هستند. در واقع، هر تراکنش از یک پیام فراخوانی سطح بالا[7] تشکیل شده است که به نوبه خود می‌تواند پیام‌های فراخوانی دیگری ایجاد کند.

یک قرارداد می‌تواند تصمیم بگیرد که چه مقدار از گاز باقیمانده آن باید با پیام فراخوانی داخلی ارسال شود و چه مقدار آن را می‌خواهد نگهداری کند. اگر یک استثنای اتمام-گاز[8] در فراخوانی داخلی (یا هر استثناء دیگر) اتفاق بیفتد، با یک مقدار خطا روی پشته[9] نشان داده می‌شود. در این حالت، فقط گاز ارسالی همراه با فراخوانی مصرف می‌شود. در سالیدیتی، فراخوانی قرارداد به طور پیش فرض در چنین شرایطی باعث یک استثناء دستی می‌شود، به طوری که استثناء فراخوانی، پشته را به صورت " پنجره نمایش[10]" نمایش می‌دهد.

همانطور که قبلاً گفته شد، قرارداد فراخوانی شده (که می‌تواند همان فراخوانی کننده باشد) یک مِمُوری تازه پاک شده را دریافت می‌کند و به فراخوانی پیلود[11] دسترسی دارد- که در یک فضای جداگانه به نام فراخوانی داده یا calldata ارائه می‌شود. پس از پایان اجرا، داده‌هایی را که در مکانی از مِمُوری فراخوانی کننده که از قبل توسط فراخوانی کننده اختصاص داده شده‌است و در آن ذخیره خواهند شد، را می‌تواند برگرداند. همه‌ی این تماس‌ها کاملاً همگام هستند.

فراخوانی‌ها به عمق 1024 محدود می‌شوند، این بدان معنی است که برای انجام عملیات پیچیده‌تر، حلقه‌ها[12] بر فراخوانی‌های مکرر باید ترجیح داده شوند. بعلاوه، فقط 64/63امین از گاز می‌تواند در یک پیام فراخوانی فوروارد شود، که در عمل باعث ایجاد عمق کمتر از 1000 می‌شود.

---

[1] Message Calls
[2] source
[3] target
[4] data payload
[5] Ether
[6] gas
[7] top-level message call
[8] out-of-gas exception
[9] stack
[10] bubble up
[11] call payload
[12] loops

Delegatecall / Callcode و کتابخانه ها

نوع خاصی از پیام های فراخوانی به نام delegatecall وجود دارد، که همانند یک پیام فراخوانی می‌باشند، جدا از این واقعیت که کد در آدرس مقصد در قرارداد فراخوانی کننده اجرا می‌شود و `msg.sender` و `msg.value` مقادیر خود را تغییر نمی‌دهند.

این بدان معناست که یک قرارداد می‌تواند به صورت پویا در زمان اجرا، کد را از آدرس دیگری بارگیری کند. Storage، آدرس فعلی و بالانس هنوز به قرارداد فراخوانی کننده اشاره دارد و فقط کد از آدرس فراخوانی شده گرفته شده‌است.

delegatecall امکان ویژگی "کتابخانه" در سالیدیتی را فراهم می‌کند: کد کتابخانه قابل استفاده مجدد می‌باشد که می‌تواند در storage قرارداد اعمال شود. به عنوان مثال به منظور پیاده سازی یک ساختار داده پیچیده[1].

گزارش‌ها[2]

می‌توان داده‌ها را در یک ساختار داده‌ای با نمایه خاص ذخیره کرد که همه راه‌ها تا سطح بلوک را نشان می‌دهد. این ویژگی لاگ یا گزارش نامیده شده‌است که توسط سالیدیتی به منظور اجرای رویدادها استفاده می‌شود. قراردادها پس از ایجاد نمی‌توانند به لاگ داده‌ها دسترسی داشته باشند، اما از خارج از بلاکچین می‌توان به صورت کارآمد به آنها دسترسی داشته باشند. از آنجا که بخشی از لاگ داده‌ها در فیلترهای بلوم ذخیره می‌شوند، جستجوی این داده‌ها به روشی کارآمد و رمزنگاری شده امکان پذیراست، بنابراین جفت‌های شبکه‌ای که کل بلاکچین را بارگیری نمی‌کنند (به اصطلاح "کلاینت لایت") هنوز هم می‌توانند این لاگ‌ها را پیدا کنید.

[1] complex data structure

[2] Logs

ایجاد کردن[1]

قراردادها حتی می‌توانند قراردادهای دیگری را با استفاده از یک آپکد خاص[2] ایجاد کنند (یعنی آدرس صفر[3] را به عنوان یک تراکنش به سادگی صدا نمی‌زنند). تنها تفاوت بین فراخوانی‌های ایجاد[4] و پیام‌های فراخوانی عادی[5] این است که داده‌های پیلود[6] اجرا می‌شوند و نتیجه به عنوان کد[7] و فراخوانی کننده[8]، ذخیره می‌شود. ایجاد کننده، آدرس قرارداد جدید را روی پشته[9] دریافت می‌کند.

غیرفعال کردن و خود تخریبی

تنها راه حذف کد از بلاکچین زمانی است که قراردادی در آن آدرس عملیات `selfdestruct` را انجام دهد. باقی مانده اتر ذخیره شده در آن آدرس به مقصد تعیین شده ارسال می‌شود و سپس storage و کد از حالت[10] خارج می‌شود. حذف قرارداد از نظر تئوری یک ایده خوب به نظر می‌رسد، اما به طور بالقوه خطرناک است، زیرا اگر کسی اتر را به قراردادهای حذف شده بفرستد، اتر برای همیشه از بین می‌رود.

هشدار
حتی اگر قراردادی با `selfdestruct` حذف شود، هنوز به عنوان بخشی از تاریخچهِ بلاکچین باقی می‌ماند و احتمالاً توسط اکثر گره‌های اتریوم نگهداری شود. بنابراین استفاده از `selfdestruct` با حذف داده‌ها از روی هارد دیسک یکسان نیست.

توجه داشته باشید
حتی اگر کد قرارداد فاقد فراخوانی `selfdestruct` باشد، باز هم می‌تواند آن عملیات را با استفاده از `delegatecall` یا `callcode` انجام دهد.

[1] Create
[2] special opcode
[3] zero address
[4] create calls
[5] normal message calls
[6] payload data
[7] code
[8] caller
[9] stack
[10] state

اگر می‌خواهید قراردادهای خود را غیرفعال[1] کنید، در عوض باید آنها را با تغییر حالت داخلی[2] که باعث برگشت همه توابع می‌شود، غیرفعال کنید. این امر استفاده از قرارداد را غیرممکن می‌کند، زیرا بلافاصله اتر را برمی‌گرداند.

3.2 نصب کامپایلر سالیدیتی

3.2.1 نسخه بندی[3]

نسخه‌های سالیدیتی از نسخه سینتکسی[4] پیروی می‌کنند و علاوه بر این نسخه‌ها، نسخه‌های نسخه شبانه[5] نیز در دسترس هستند. کارکردن نسخه‌های نسخه شبانه تضمین نمی‌شود و علی‌رغم همه تلاش‌ها ممکن است تغییرات غیرمستند و یا تغییرات جدید[6] را شامل شوند. توصیه می‌کنیم از آخرین نسخه استفاده کنید. بسته‌های نصبی زیر از آخرین نسخه استفاده خواهند کرد.

3.2.2 ریمیکس[7]

ریمیکس را برای قراردادهای کوچک و برای یادگیری سریع سالیدیتی توصیه می‌کنیم.

می‌توانید به صورت آنلاین به ریمیکس دسترسی داشته باشید و نیازی به نصب هیچ چیزی ندارید. اگر می‌خواهید بدون اتصال به اینترنت از آن استفاده کنید، به https://github.com/ethereum/remix-live/tree/gh-pages مراجعه کنید و فایل `.zip` را همانطور که در آن صفحه توضیح داده شده بارگیری کنید. ریمیکس همچنین یک گزینه مناسب برای تست نسخه شبانه بدون نصب چندین نسخه سالیدیتی است.

---

[1] disable
[2] internal state
[3] Versioning
[4] semantic versioning
[5] nightly development builds
[6] broken changes
[7] Remix

این صفحه گزینه‌های بیشتر برای نصب نرم افزار کامپایلر سالیدیتی[1] روی رایانه شما با خط فرمان را توضیح می‌دهد. اگر در حال کار بر روی قرارداد بزرگتر هستید یا به گزینه‌های کامپایل بیشتری نیاز دارید، یک کامپایلر خط فرمان[2] انتخاب کنید.

npm / Node.js 3.2.3

برای نصب راحت و قابل حمل[3] `solcjs` یک کامپایلر سالیدیتی، از `npm` استفاده کنید. برنامه solcjs دارای ویژگی‌های[4] کمتری نسبت به راه‌های دسترسی به کامپایلر است که در این صفحه توضیح داده شده است. مستندات[5] استفاده از کامپایلر خط فرمان فرض می‌کند که از کامپایلر کامل `solc` استفاده می‌کنید. استفاده از `solcjs` در مخزن خود ثبت شده است.

توجه داشته باشید: پروژه solc-js از C++ solc با استفاده از Emscripten مشتق گرفته شده‌است، به این معنی که هر دو از کد منبع کامپایلر یکسانی استفاده می‌کنند. solc-js را می‌توان مستقیماً در پروژه‌های جاوا اسکریپت (مانند ریمیکس) استفاده کرد. لطفاً به مخزن solc-js برای مشاهده دستورات مراجعه کنید.

```
npm install -g solc
```

توجه داشته باشید
خط فرمان اجرایی `solcjs` نام دارد. گزینه‌های خط فرمان `solcjs` با `solc` سازگار نیستند و ابزارها (مانند `geth`) انتظار دارند رفتار `solc` با `solcjs` کار نکند.

3.2.4 داکر

[1] Solidity compiler software
[2] commandline compiler
[3] portable
[4] features
[5] documentation

تصاویر داکر[1] از نسخه‌های سالیدیتی با استفاده از تصویر `solc` از سازمان `ethereum` در دسترس است. از برچسب `stable` برای آخرین نسخه منتشر شده و `nightly` برای تغییرات احتمالی ناپایدار در شاخه‌ی نسخه استفاده می‌کند.

تصویر داکر کامپایلر اجرایی را اجرا می‌کند، بنابراین می‌توانید تمام آرگومان‌های کامپایلر را به آن منتقل کنید. به عنوان مثال، دستور زیر نسخه پایدار[2] تصویر `solc` را دریافت می‌کند (اگر قبلاً آن را ندارید)، و آن را در یک محفظه جدید اجرا می‌کند و آرگومان `--help` را عبور می‌دهد.

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

همچنین می‌توانید نسخه‌هایِ توسعهِ منتشر شده را با برچسب مشخص کنید، به عنوان مثال، برای نسخه 0.5.4.

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

برای استفاده از تصویر داکر برای کامپایل فایل‌های سالیدیتی در دستگاه میزبان، یک پوشه محلی برای ورودی و خروجی نصب کرده و قرارداد کامپایل را مشخص کنید. برای مثال:

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

همچنین می‌توانید از رابط استاندارد JSON (که هنگام استفاده از کامپایلر با ابزار توصیه می‌شود) استفاده کنید. هنگام استفاده از این رابط لازم نیست هیچ دایرکتوری را نصب کنید.

[1] Docker images

[2] stable version

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

3.2.5 بسته‌های لینوکس[1]

بسته‌های باینری سالیدیتی در مخزن گیتهاب سالیدیتی / انتشارات[2] موجود است.

ما همچنین PPAs برای اوبونتو[3] داریم، می‌توانید آخرین نسخه پایدار[4] را با استفاده از دستورات زیر دریافت کنید:

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

نسخه شبانه را می‌توان با استفاده از این دستورات زیر نصب کرد:

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

ما همچنین یک بسته اسنپ[5] را منتشر می‌کنیم که در همه توزیع‌های پشتیبانی شده لینوکس[6] قابل نصب است.

برای نصب آخرین نسخه پایدار solc:

```
sudo snap install solc
```

اگر می‌خواهید آخرین نسخه توسعه دهنده سالیدیتی را با جدیدترین تغییرات آزمایش کنید، لطفاً از موارد زیر استفاده کنید:

---

[1] Linux Packages
[2] solidity/releases
[3] Ubuntu
[4] stable version
[5] snap package
[6] supported Linux distros

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

توجه داشته باشید
اسنپ `solc` از سختگیری شدید استفاده می‌کند. این حالت امن‌ترین حالت برای بسته‌های فوری است اما با محدودیت‌هایی همراه است، مثلاً شما فقط به فایل‌های موجود در دایرکتوری‌های `/home` و `/media` دسترسی خواهید داشت. برای کسب اطلاعات بیشتر، به Demystifying Snap Confinement مراجعه کنید.

آرک لینوکس[1] همچنین دارای بسته‌هایی است، البته محدود به آخرین نسخه توسعه می‌باشد:

```
pacman -S solidity
```

جنتو لینوکس[2] دارای همپوشانی اتریوم[3] می‌باشد که حاوی بسته سالیدیتی است. پس از راه اندازی این همپوشانی[4]، `solc` را می‌توان در معماری x86_64 توسط فرمان زیر نصب کرد:

```
emerge dev-lang/solidity
```

[1] Arch Linux
[2] Gentoo Linux
[3] Ethereum overlay
[4] overlay

3.2.6 بسته‌های مک‌او اس[1]

ما کامپایلر سالیدیتی را از طریق هوم‌برو[2] به عنوان نسخهِ ساخته شده از منبع[3] توزیع می‌کنیم. مخزن‌های از پیش ساخته شده در حال حاضر پشتیبانی نمی‌شوند.

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

برای نصب جدیدترین نسخه سالیدیتی 0.5.x / 0.4.x می‌توانید به ترتیب از `brew install solidity@4` و `brew install solidity@5` استفاده کنید.

اگر به نسخه خاصی از سالیدیتی نیاز دارید، می‌توانید فرمول هوم‌برو[4] را مستقیماً از گیتهاب نصب کنید.

لینک solidity.rb commits on Github را مشاهده کنید.

هشِ کامیت‌های[5] نسخه موردنظر خود را کپی کرده و در دستگاه خود بررسی کنید.

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

با استفاده از `brew` آن را نصب کنید:

```
brew unlink solidity
# eg. Install 0.4.8
brew install solidity.rb
```

---

[1] macOS
[2] Homebrew
[3] build-from-source version
[4] Homebrew
[5] commit hash

3.2.7 باینری‌های استاتیک[1]

ما یک مخزن[2] حاوی نسخه‌های استاتیک[3] از نسخه‌های کامپایلر قبلی و فعلی را برای همه پلتفرم‌های پشتیبانی شده در solc-bin نگهداری می‌کنیم. این مکان همچنین مکانی است که می‌توانید نسخه‌های شبانه[4] را در آن پیدا کنید.

مخزن نه تنها راهی سریع و آسان برای کاربران نهایی است تا باینری‌ها را برای استفاده در خارج از جعبه [5] آماده کنند، بلکه به معنای مناسب بودن با ابزارهای ثالث است:

- محتوا در https://binaries.soliditylang.org قرارداده شده‌است که در آن می‌توان به راحتی از طریق HTTPS بدون احراز هویت، محدودیت سرعت یا نیاز به استفاده از git بارگیری کرد.
- محتوا با هدرهای صحیح نوع محتوا[6] و با پیکربندی CORS ارائه می‌شود تا بتواند مستقیماً توسط ابزارهایی که در مرورگر اجرا می‌شوند بارگیری شود.
- فایل‌های باینری نیازی به نصب یا باز کردن بسته بندی ندارند (به استثنای نسخه‌های قدیمی ویندوز که همراه با DLLهای ضروری هستند).
- ما برای سطح بالایی از سازگاری با گذشته[7] تلاش می‌کنیم. فایل‌ها، پس از اضافه شدن، بدون ارائه پیوند[8] تغییر مسیر در مکان قدیمی حذف یا منتقل نمی‌شوند. همچنین هرگز در محل خود تغییر داده نمی‌شوند و همیشه باید با چک‌سام[9] اصلی مطابقت داشته باشند. تنها استثناء می‌تواند فایل‌های شکسته یا غیرقابل استفاده باشد که اگر به همین صورت باقی بمانند، می‌توانند بیشتر از فایده باعث آسیب شوند.

---

[1] Static Binaries
[2] repository
[3] static builds
[4] nightly builds
[5] out-of-the-box
[6] correct Content-Type headers
[7] backwards-compatibility
[8] symlink
[9] checksum

- فایل‌ها از طریق HTTP و HTTPS ارائه می‌شوند. تا زمانیکه لیست فایل‌ها را به صورت ایمن (از طریق git، HTTPS، IPFS یا به صورت محلی ذخیره کردید) به دست آوردید. و پس از بارگیری، فایل‌های هش باینری را تأیید کنید. لازم نیست از HTTPS برای خود فایل‌های باینری استفاده کنید.

همین فایل‌های باینری در بیشتر موارد در صفحه انتشارات سالیدیتی[1] در گیت‌هاب موجود است. تفاوت در این است که ما به طور کلی نسخه‌های قدیمی را در صفحه انتشار گیتهاب به روز نمی‌کنیم. این بدان معناست که در صورت تغییر شرایط نامگذاری، نام آن‌ها را تغییر نمی‌دهیم و برای پلتفرم‌هایی که در زمان انتشار پشتیبانی نمی‌شوند، نسخه‌هایی اضافه نمی‌کنیم. این امر فقط در `solc-bin` اتفاق می‌افتد.

مخزن `solc-bin` شامل چندین دایرکتوری سطح بالا است که هر یک نمایانگر یک پلتفرم واحد می‌باشد. هر یک شامل یک فایل `list.json` است که فایل‌های باینری موجود را فهرست می‌کند. برای مثال در `emscripten-wasm32/list.json` اطلاعات زیر را در مورد نسخه 0.7.4 خواهید یافت:

```
{
  "path": "solc-emscripten-wasm32-
v0.7.4+commit.3f05b770.js",
  "version": "0.7.4",
  "build": "commit.3f05b770",
  "longVersion": "0.7.4+commit.3f05b770",
  "keccak256":
"0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99
279af8c3",
  "sha256":
"0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7
e2d55af2",
  "urls": [
```

[1] Solidity release

```
"bzzr://16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8
a97f18c955ab1",

"dweb:/ipfs/QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS"
  ]
}
```

این بدان معناست که:

- شما می‌توانید باینری را در همان فهرست با نام solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js پیدا کنید. توجه داشته باشید که فایل ممکن است یک پیوند[1] باشد و اگر از گیت[2] برای بارگیری آن استفاده نمی‌کنید یا سیستم فایل شما از پیوندها[3] پشتیبانی نمی‌کند، باید خودتان آن را حل کنید.
- باینری نیز در https://binaries.soliditylang.org/emscripten-wasm32/solc-emscripten-wasm32-v0.7.4+commit.3f05b770.js قرار داده شده‌است. در این حالت گیت ضروری نمی‌باشد. و پیوندها، یا با ارائه یک کپی از فایل یا با بازگرداندن یک مسیر HTTP به طور شفاف حل[4] می‌شوند.
- فایل همچنین در IPFS در QmTLs5MuLEWXQkths41HiACoXDiH8zxyqBHGFDRSzVE5CS موجود است.

[1] symlink
[2] git
[3] symlinks
[4] resolved

- ممکن است فایل در آینده در Swarm با شماره 16c5f09109c793db99fe35f037c6092b061bd39260ee7a677c8a97f18c955ab1 موجود باشد.
- با مقایسه هش keccak256 آن با `0x300330ecd127756b824aa13e843cb1f43c473cb22eaf3750d5fb9c99279af8c3` می‌توانید صحت باینری را بررسی کنید. هش را می‌توان در خط فرمان با استفاده از ابزار `keccak256sum` محاسبه کرد که توسط تابع sha3sum یا keccak256() function from ethereumjs-utilدر جاوا اسکریپت ارائه شده‌است.
- همچنین می‌توانید یکپارچگی باینری را با مقایسه هش sha256 آن با `0x2b55ed5fec4d9625b6c7b3ab1abd2b7fb7dd2a9c68543bf0323db2c7e2d55af2` تأیید کنید.

هشدار
به دلیل نیاز به سازگاری زیاد، مخزن حاوی برخی از عناصر قدیمی می‌باشد، اما هنگام نوشتن ابزارهای جدید نباید از آنها استفاده کنید: • اگر می‌خواهید بهترین عملکرد را داشته باشید، از `emscripten-wasm32/` (با جایگزینی برای `emscripten-asmjs/`) به جای `bin/` استفاده کنید. ما تا نسخه 0.6.1 فقط فایل‌های باینری asm.js را ارائه می‌دادیم. با شروع 0.6.2، ما به WebAssembly builds به عملکرد بسیار بهتر روی آوردیم. ما نسخه‌های قدیمی تر را برای wasm بازسازی کرده‌ایم اما فایل‌های اصلی asm.js در `bin/` باقی می‌مانند. موارد جدید باید در یک فهرست جداگانه قرار داده شوند تا از تصادم نامی جلوگیری شود.

- اگر می‌خواهید مطمئن شوید که در حال بارگیری wasm یا باینری asm.js هستید، از `emscripten-asmjs/` و `emscripten-wasm32/` به جای `bin/` و `wasm/` استفاده کنید.
- به جای `list.js` و `list.txt` از `list.json` استفاده کنید. فرمت لیست JSON شامل تمام اطلاعات قدیمی و بیشتر است.
- به جای https://binaries.soliditylang.org از https://solc-bin.ethereum.org استفاده کنید. برای ساده نگه داشتن مسائل، ما تقریباً همه چیز مربوط به کامپایلر را تحت دامنه جدید `soliditylang.org` منتقل کردیم و این امر در مورد `solc-bin` نیز صدق می‌کند. با اینکه دامنه جدید توصیه می‌شود، اما دامنه قبلی هنوز کاملاً پشتیبانی می‌شود و تضمین می‌شود که به همان مکان اشاره می‌کند.

هشدار
فایل‌های باینری نیز در https://ethereum.github.io/solc-bin/ در دسترس هستند، اما این صفحه به روز رسانی خود را پس از انتشار نسخه 0.7.2 متوقف کرد، هیچ نسخه جدید یا نسخه شبانه برای هر پلتفرمی دریافت نمی‌کند و ساختار دایرکتوری جدید، از جمله ساختارهای غیر emscripten را ارائه نمی‌دهد. اگر از آن استفاده می‌کنید، لطفاً به https://binaries.soliditylang.org مراجعه کنید، که یک جایگزینی رها کردن[1] است. این به ما امکان می‌دهد تا به طور شفاف در میزبانی اصلی تغییراتی ایجاد کرده و اختلال را به حداقل برسانیم. بر خلاف دامنه `ethereum.github.io` ، که ما هیچ کنترلی بر آن

[1] drop-in

نداریم، کار کردن `binaries.soliditylang.org` تضمین می‌شود و در دراز مدت همان URL را حفظ کند.

3.2.8 نسخه از منبع

پیش نیازهای – همه‌ی سیستم عامل‌ها

موارد زیر وابستگی‌ها[1] برای همه نسخه‌های سالیدیتی[2] می‌باشند:

نرم افزار	نکات
CMake (version 3.13+)	نسخه مولدِ فایلِ کراس پلتفرم[3]
Boost (version 1.65+)	کتابخانه‌های C++
Git	ابزار خط فرمان برای بازیابی کد منبع.
z3 (version 4.8+, Optional)	برای استفاده با SMT checker.
cvc4 (Optional)	برای استفاده با SMT checker.

توجه داشته باشید

نسخه‌های سالیدیتی قبل از 0.5.10 نمی‌توانند به درستی با نسخه‌های Boost 1.70+ لینک شوند. یک راه حل ممکن این است که قبل از اجرای دستور cmake برای پیکربندی سالیدیتی، نام `<Boost install path>/lib/cmake/Boost-1.70.0` را به طور موقت تغییر نام دهید.

[1] Dependencies
[2] builds of Solidity
[3] Cross-platform build file generator

با شروع از 0.5.10 لینک کردن برخلاف Boost 1.70+ باید بدون دخالت دستی کار کند.

حداقل نسخه‌های کامپایلر

کامپایلرهای C ++ زیر و حداقل نسخه‌های آنها می‌توانند پایگاه کد[1] سالیدیتی را ایجاد کنند:

- GCC ، نسخه 8+
- Clang ، نسخه 7+
- MSVC ، نسخه 2019+

پیش نیازها - مک‌او اس[2]

برای نسخه‌های[3] مک‌او اس، مطمئن شوید که آخرین نسخه Xcode را نصب کرده‌اید. این شامل کامپایلر Clang C++ ، Xcode IDE و سایر ابزارهای توسعه اپل می‌باشد که برای ایجاد برنامه‌های C ++ در OS X مورد نیاز است. اگر برای اولین بار Xcode را نصب می‌کنید یا نسخه جدیدی را نصب کرده‌اید، باید قبل از توسعه، با لایسنس موافقت کنید تا بتوانید با خط فرمان، توسعه‌ها را انجام دهید:

```
sudo xcodebuild -license accept
```

اسکریپت نسخه OS X ما، از مدیریت بسته هوم‌برو[4] برای نصب نیازمندیهای خارجی استفاده می‌کند. اگر می‌خواهید دوباره از ابتدا شروع کنید، در اینجا نحوه حذف نصب هوم‌برو ذکر شده‌است.

پیش نیازها - ویندوز

شما باید وابستگی‌های زیر را برای نسخه‌های ویندوز برای سالیدیتی نصب کنید:

---

[1] codebase
[2] macOS
[3] builds
[4] the Homebrew package manager

نرم افزار	نکات
ابزارهای نسخه 2019 ویژوال استودیو	کامپایلر C++
ویژوال استودیو2019 (اختیاری)	کامپایلر C++ و محیط توسعه

اگر از قبل یک ویرایشگر[1] دارید و فقط به کامپایلر و کتابخانه نیاز دارید، می‌توانید ابزارهای نسخه 2019 ویژوال استودیو را نصب کنید.

ویژوال استودیو 2019 هم ویرایشگر و هم کامپایلر و کتابخانه‌های لازم را ارائه می‌دهد. بنابراین اگر ویرایشگر ندارید و ترجیح می‌دهید سالیدیتی را توسعه دهید، ویژوال استودیو 2019 ممکن است یک انتخاب برای شما باشد تا همه چیز را به راحتی راه اندازی کنید.

در اینجا لیستی از اجزایی که باید در ابزارهای نسخه ویژوال استودیو 2019 نصب شود، آورده شده‌است:

- ویژگی های اصلی Visual Studio C++
- مجموعه ابزارهای VC 2019 v141 toolset (x86,x64)
- CRT SDK ویندوز یونیورسال
- SDK ویندوز 8.1
- پشتیبانی از C ++/CLI

اسکریپت کمکی وابستگی‌ها[2]

ما یک اسکریپت کمکی داریم که می‌توانید از آن برای نصب تمام وابستگی‌های خارجی مورد نیاز در مک‌او اس[3]، ویندوز و چندین توزیع لینوکس استفاده کنید.

```
./scripts/install_deps.sh
```

یا در ویندوز:

```
scripts\install_deps.ps1
```

[1] IDE [2] Dependencies [3] macOS

توجه داشته باشید که دستور دوم `boost` و `cmake` را در زیر شاخه `deps` نصب می‌کند، در حالی که دستور قبلی سعی می‌کند وابستگی‌ها را به صورت جهانی نصب کند.

مخزن را کلون کنید[1]

برای شبیه سازی کد منبع[2]، دستور زیر را اجرا کنید:

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

اگر می‌خواهید به نسخه سالیدیتی کمک کنید، باید سالیدیتی را فورک[3] کنید و فورک شخصی خود را به عنوان ریموت دوم[4] اضافه کنید:

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

توجه داشته باشید
این روش منجر به نسخه پیش انتشار[5] می‌شود که به عنوان مثال یک فَلَگ[6] در هر کد بایتی که توسط چنین کامپایلری تولید می‌شود، تنظیم می‌شود. اگر می‌خواهید کامپایلر سالیدیتی منتشر شده را دوباره توسعه دهید، لطفاً از tarball منبع در صفحه انتشار گیتهاب استفاده کنید: https://github.com/ethereum/solidity/releases/download/v0.X.Y/solidity_0.X.Y.tar.gz (نه "کد منبع" ارائه شده توسط گیتهاب).

نسخه خط فرمان[7]

قبل از توسعه حتماً وابستگی‌های خارجی را نصب کنید (به قسمت بالا مراجعه کنید).

---

[1] Clone the Repository
[2] source code
[3] fork
[4] second remote
[5] prerelease build
[6] flag
[7] Command-Line Build

پروژه سالیدیتی از CMake برای پیکربندی نسخه استفاده می‌کند. ممکن است بخواهید ccache را برای سرعت بخشیدن به نسخه‌های مکرر نصب کنید. CMake آن را به طور خودکار انتخاب می‌کند. نسخه سالیدیتی در لینوکس، مک‌او اس[1] و سایر یونیکس‌ها کاملاً مشابه است:

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

یا حتی در لینوکس و مک‌او اس راحت‌تر، می‌توانید اجرا کنید:

```
#note: this will install binaries solc and soltest at
usr/local/bin
./scripts/build.sh
```

هشدار
توسعه BSD باید کار کند، اما توسط تیم سالیدیتی آزمایش نشده است.

و برای ویندوز:

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

در صورت تمایل به استفاده از نسخه boost نصب شده توسط اسکریپت `scripts\install_deps.ps1`، علاوه بر این باید `-DBoost_DIR="deps\boost\lib\cmake\Boost-*"` و `-DCMAKE_MSVC_RUNTIME_LIBRARY=MultiThreaded` را به عنوان آرگومان برای فراخوانی `cmake` ارسال کنید.

[1] macOS

این عمل باید منجر به ایجاد **solidity.sln** در آن نسخه دایرکتوری شود. دو بار کلیک بر روی آن فایل باعث می‌شود تا ویژوال استودیو روشن شود. ما ساخت پیکربندی انتشار[1] را پیشنهاد می‌کنیم، اما بقیه نیز کار می‌کنند.

از سوی دیگر، می‌توانید برای ویندوز روی خط فرمان[2] توسعه دهید، مانند این:

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

3.2.9 گزینه‌های CMake

اگر علاقه دارید که چه گزینه‌های CMake در دسترس هستند، `cmake .. -LH` را اجرا کنید.

حل کننده‌های[3] SMT

سالیدیتی را می‌توان در کنار حل کننده‌های SMT ایجاد کرد و در صورت یافتن آنها در سیستم به طور پیش فرض این کار را انجام می‌دهد. هر حل کننده را می‌توان با گزینه cmake غیرفعال کرد.

توجه: در برخی موارد ، این نیز می تواند یک راه حل احتمالی برای خرابی نسخه باشد.

در داخل پوشه build می‌توانید آنها را غیرفعال کنید، زیرا به طور پیش فرض فعال هستند:

```
# disables only Z3 SMT Solver.
cmake .. -DUSE_Z3=OFF

# disables only CVC4 SMT Solver.
cmake .. -DUSE_CVC4=OFF

# disables both Z3 and CVC4
cmake .. -DUSE_CVC4=OFF -DUSE_Z3=OFF
```

3.2.10 رشته نسخه[4] با جزئیات

رشته نسخه سالیدیتی شامل چهار قسمت است:

---

[1] Release
[2] command-line
[3] SMT Solvers
[4] Version String

- شماره نسخه
- برچسب پیش از انتشار، معمولاً با `develop.YYYY.MM.DD` یا `nightly.YYYY.MM.DD` تنظیم می‌شود.
- کامیت در قالب `commit.GITHASH`
- پلتفرم، که دارای تعداد دلخواه موارد است، حاوی جزئیات مربوط به پلتفرم و کامپایلر

اگر تغییرات محلی وجود داشته باشد، کامیت‌ها با `.mod` پسوند داده می‌شود.

این قطعات طبق نیاز Semver ترکیب می‌شوند، جایی که برچسب پیش از انتشار سالیدیتی برابر است با پیش از انتشار Semver و کامیت سالیدیتی و پلتفرم ترکیبی فراداده توسعه Semver را تشکیل می‌دهند.

مثال انتشار: `0.4.8+commit.60cc1668.Emscripten.clang`

یک نمونه پیش از انتشار: `0.4.9-nightly.2017.1.17+commit.6ecb4aa3.Emscripten.clang`

3.2.11 اطلاعات مهم در مورد نسخه بندی

پس از انتشار، سطح نسخه پَچ[1] بامپ[2] شده است، زیرا ما فرض می‌کنیم که فقط تغییرات سطح پَچ دنبال می‌شود. وقتی تغییرات ادغام می‌شوند، نسخه باید با توجه به semver و شدت تغییرات بامپ شود. سرانجام، همیشه نسخه‌ای از نسخه فعلی شبانه منتشر می‌شود، اما بدون تعیین `prerelease`.

مثال:

0. نسخه 0.4.0 ساخته شده است.
1. نسخه شبانه[3] از این پس نسخه 0.4.1 دارد.
2. تغییرات بدون تغییرات جدید[4] ارائه می شوند-> بدون تغییر در نسخه.

[1] patch
[2] bump
[3] nightly build
[4] Non-breaking changes

3. یک تغییر جدید[1] معرفی می‌شود -> نسخه به 0.5.0 افزایش می یابد.
4. نسخه 0.5.0 ساخته شده است.

این رفتار با نسخه پراگما[2] به خوبی کار می‌کند.

3.3 سالیدیتی با مثال

3.3.1 رای گیری

قرارداد زیر کاملاً پیچیده است، اما بسیاری از ویژگی‌های سالیدیتی را به نمایش می‌گذارد. قرارداد رای گیری را اجرا می‌کند. البته، مشکلات اصلی رای گیری الکترونیکی نحوه واگذاری حق رای به افراد صحیح و نحوه جلوگیری از دستکاری است. ما همه مشکلات را در اینجا حل نخواهیم کرد، اما حداقل نشان خواهیم داد که چگونه می‌توان رای گیریِ نمایندگان را انجام داد تا شمارش آراء در همان زمان به صورت خودکار و کاملاً شفاف انجام شود.

ایده این است که یک قرارداد برای هر رأی ایجاد شود و نام کوتاهی برای هر گزینه ارائه شود. سپس خالق قرارداد که به عنوان رئیس فعالیت می‌کند به هر آدرس به طور جداگانه حق رای می‌دهد.

افراد پشت آدرس‌ها می‌توانند انتخاب کنند که یا خود رأی دهند یا رای خود را به شخصی که به او اعتماد دارند واگذار کنند.

در پایان زمان رای گیری، `winningProposal()` پیشنهاد را با بیشترین تعداد رای برمیگرداند.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
```

[1] breaking change

[2] version pragma

```
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;

    /// Create a new ballot to choose one of
`proposalNames`.
    constructor(bytes32[] memory proposalNames) {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }

    // Give `voter` the right to vote on this ballot.
    // May only be called by `chairperson`.
```

```
    function giveRightToVote(address voter) public {
        // If the first argument of `require` evaluates
        // to `false`, execution terminates and all
        // changes to the state and to Ether balances
        // are reverted.
        // This used to consume all gas in old EVM
versions, but
        // not anymore.
        // It is often a good idea to use `require` to
check if
        // functions are called correctly.
        // As a second argument, you can also provide an
        // explanation about what went wrong.
        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;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) public {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

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

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
```

```
        // In this case, the delegation will not be
executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not
allowed.
            require(to != msg.sender, "Found loop in
delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount +=
sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) public {
        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;
```

```
        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    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;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and
then
    // returns the name of the winner
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

بهبودهای احتمالی

در حال حاضر، تراکنش‌ها زیادی برای واگذاری حق رأی به همه شرکت کنندگان مورد نیاز است. آیا می‌توانید به راه بهتری فکر کنید؟

3.3.2 مزایده کور[1]

در این بخش، نشان خواهیم داد که ایجاد یک قرارداد حراجِ کاملا کور[2] در اتریوم چقدر آسان است. ما با یک مزایده باز[3] شروع می‌کنیم که در آن همه می‌توانند پیشنهادات[4] ارائه شده را ببینند و سپس این قرارداد را به حراج کور توسعه می‌دهیم که در آن امکان مشاهده پیشنهاد واقعی تا زمان پایان مناقصه وجود ندارد.

مزایده باز ساده

ایده‌ی کلی قرارداد مزایده ساده[5] زیر این است که همه می‌توانند پیشنهادات خود را در طول یک دوره مناقصه[6] ارسال کنند. پیشنهادات در حال حاضر شامل ارسال پول/ اتر[7] به منظور متصل کردن مناقصه‌گرانِ[8] مناقصه به پیشنهاد آنها است. اگر بالاترین پیشنهاد[9] افزایش یابد، مناقصه‌گران قبلی پول خود را پس می‌گیرد. پس از پایان دوره مناقصه، قرارداد باید به صورت دستی فراخوانده شود تا ذینفع پول خود را دریافت کند- قراردادها نمی‌توانند خودشان فعال شوند.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;
contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
```

[1] Blind Auction
[2] completely blind auction
[3] open auction
[4] bids
[5] simple auction contract
[6] bidding period
[7] Ether
[8] bidders
[9] highest bid

```
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // Errors that describe failures.

    // The triple-slash comments are so-called natspec
    // comments. They will be shown when the user
    // is asked to confirm a transaction or
    // when an error is displayed.

    /// The auction has already ended.
    error AuctionAlreadyEnded();
    /// There is already a higher or equal bid.
    error BidNotHighEnough(uint highestBid);
    /// The auction has not ended yet.
    error AuctionNotYetEnded();
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    /// Create a simple auction with `_biddingTime`
    /// seconds bidding time on behalf of the
    /// beneficiary address `_beneficiary`.
    constructor(
        uint _biddingTime,
        address payable _beneficiary
    ) {
        beneficiary = _beneficiary;
        auctionEndTime = block.timestamp + _biddingTime;
    }

    /// Bid on the auction with the value sent
    /// together with this transaction.
    /// The value will only be refunded if the
    /// auction is not won.
    function bid() public payable {
        // No arguments are necessary, all
        // information is already part of
        // the transaction. The keyword payable
        // is required for the function to
```

```
        // be able to receive Ether.

        // Revert the call if the bidding
        // period is over.
        if (block.timestamp > auctionEndTime)
            revert AuctionAlreadyEnded();

        // If the bid is not higher, send the
        // money back (the revert statement
        // will revert all changes in this
        // function execution including
        // it having received the money).
        if (msg.value <= highestBid)
            revert BidNotHighEnough(highestBid);

        if (highestBid != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security
risk
            // because it could execute an untrusted
contract.
            // It is always safer to let the recipients
            // withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because
the recipient
            // can call this function again as part of the
receiving call
            // before `send` returns.
```

```
            pendingReturns[msg.sender] = 0;

            if (!payable(msg.sender).send(amount)) {
                // No need to call throw here, just reset
the amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() public {
        // It is a good guideline to structure functions
that interact
        // with other contracts (i.e. they call functions
or send Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing
conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract
could call
        // back into the current contract and modify the
state or cause
        // effects (ether payout) to be performed multiple
times.
        // If functions called internally include
interaction with external
        // contracts, they also have to be considered
interaction with
        // external contracts.

        // 1. Conditions
        if (block.timestamp < auctionEndTime)
            revert AuctionNotYetEnded();
```

```
        if (ended)
            revert AuctionEndAlreadyCalled();

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

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

مزایده کور

مزایده باز[1] قبلی در ادامه به مزایده کور[2] توسعه یافته‌است. مزیت مزایده کور این است که هیچ گونه فشار زمانی نسبت به پایان دوره مناقصه[3] وجود ندارد. ایجاد مزایده کور بر روی یک پلتفرم محاسباتی شفاف ممکن است متناقض به نظر برسد، اما رمزنگاری به کمک شما می‌آید.

در طول دوره مناقصه، یک پیشنهاد دهنده در واقع پیشنهاد[4] خود را ارسال نمی‌کند، بلکه فقط یک نسخه هش شده از آن را ارسال می‌کند. از آنجا که در حال حاضر یافتن دو مقدار (به اندازه کافی طولانی) که مقادیر هش آنها برابر باشد، عملاً غیرممکن تلقی می‌شود، مناقصه‌گر[5] با این کار متعهد به مناقصه می‌شود. پس از پایان دوره مناقصه، مناقصه‌گران باید پیشنهادات خود را آشکار کنند: آنها مقادیر خود را بدون رمزگذاری ارسال می‌کنند و قرارداد بررسی می‌کند که مقدار هش همان مقدار ارائه شده در دوره مناقصه است.

---

[1] open auction
[2] blind auction
[3] bidding period
[4] bid
[5] bidder

چالش دیگر این است که چگونه مزایده را به طور همزمان اجباری و پنهان یا کور جلوه دهید: تنها راه جلوگیری از ارسال نکردن مبلغ توسط داوطلب پس از برنده شدن در مزایده، ارسال آنها به همراه پیشنهاد است. از آنجا که انتقال مقدار در اتریوم پنهان یا کور نمی‌شود، هر کسی می‌تواند مقدار را ببیند.

قرارداد زیر با قبول هر مقداری که بزرگتر از بالاترین پیشنهاد[1] باشد، این مشکل را حل می‌کند. از آنجایی که این فقط در مرحله آشکار شدن قابل بررسی است، ممکن است برخی از پیشنهادات نامعتبر باشند، و این هدفمند است (حتی یک فلَگ[2] صریح برای قرار دادن پیشنهادات نامعتبر با انتقال مقدار بالا ارائه می‌دهد): مناقصه‌گران با قرار دادن چند پیشنهاد زیاد یا کم اعتبار، می‌توانند رقابت را مختل کنند.

```
// 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;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;
```

[1] highest bid

[2] flag

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

    // Errors that describe failures.

    /// The function has been called too early.
    /// Try again at `time`.
    error TooEarly(uint time);
    /// The function has been called too late.
    /// It cannot be called after `time`.
    error TooLate(uint time);
    /// The function auctionEnd has already been called.
    error AuctionEndAlreadyCalled();

    // Modifiers are a convenient way to validate inputs to
    // functions. `onlyBefore` is applied to `bid` below:
    // The new function body is the modifier's body where
    // `_` is replaced by the old function body.
    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 _beneficiary
    ) {
        beneficiary = _beneficiary;
        biddingEnd = block.timestamp + _biddingTime;
        revealEnd = biddingEnd + _revealTime;
    }

    /// Place a blinded bid with `_blindedBid` =
```

```
    /// keccak256(abi.encodePacked(value, fake, secret)).
    /// The sent ether is only refunded if the bid is
correctly
    /// revealed in the revealing phase. The bid is valid
if the
    /// ether sent together with the bid is at least
"value" and
    /// "fake" is not true. Setting "fake" to true and
sending
    /// not the exact amount are ways to hide the real bid
but
    /// still make the required deposit. The same address
can
    /// place multiple bids.
    function bid(bytes32 _blindedBid)
        public
        payable
        onlyBefore(biddingEnd)
    {
        bids[msg.sender].push(Bid({
            blindedBid: _blindedBid,
            deposit: msg.value
        }));
    }

    /// Reveal your blinded bids. You will get a refund for
all
    /// correctly blinded invalid bids and for all bids
except for
    /// the totally highest.
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
```

```
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.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], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid !=
keccak256(abi.encodePacked(value, fake, secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-
claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        payable(msg.sender).transfer(refund);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because
the recipient
            // can call this function again as part of the
receiving call
            // before `transfer` returns (see the remark
above about
```

```
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

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

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        if (ended) revert AuctionEndAlreadyCalled();
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }
}
```

3.3.3 خرید امن از راه دور

خرید کالا از راه دور در حال حاضر نیاز به چندین طرف دارد که باید به یکدیگر اعتماد کنند. ساده ترین پیکربندی شامل یک فروشنده و یک خریدار است. خریدار مایل است کالایی را از فروشنده دریافت کند و فروشنده مایل است در ازای آن پول (یا معادل آن) دریافت کند. قسمت مشکل ساز محموله در اینجا است: هیچ راهی برای تعیین اطمینان از رسیدن کالا به خریدار وجود ندارد.

روش‌های مختلفی برای حل این مشکل وجود دارد، اما همه آنها در مقابل یک یا بقیه راه‌ها کم می‌آورند. در مثال زیر، هر دو طرف باید مقدار دو برابر یک قلم کالا را به عنوان ضمانت[1] در قرارداد قرار دهند. به محض اینکه این اتفاق افتاد، پول در قرارداد قفل شده[2] خواهد ماند تا زمانی که خریدار تأیید کند که کالا را دریافت کرده‌است. پس از آن، مقدار (نیمی از سپرده خود) به خریدار بازگردانده می‌شود و فروشنده سه برابر مقدار (سپرده خود به علاوه مقدار) دریافت می‌کند. ایده پشت این امر این است که هر دو طرف انگیزه‌ای برای حل اوضاع دارند یا در غیر این صورت پول آنها برای همیشه قفل شده خواهد ماند.

البته این قرارداد مشکلی را حل نمی‌کند، اما یک نمای کلی از چگونگی استفاده از ساختار ماشین حالت مانند[3] در داخل قرارداد ارائه می‌دهد.

```
// 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 }
    // The state variable has a default value of the first
member, `State.created`
    State public state;
```

[1] escrow [2] locked [3] state machine-like

```
    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    /// Only the buyer can call this function.
    error OnlyBuyer();
    /// Only the seller can call this function.
    error OnlySeller();
    /// The function cannot be called at the current state.
    error InvalidState();
    /// The provided value has to be even.
    error ValueNotEven();

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

    // Ensure that `msg.value` is an even number.
```

```
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd
number.
    constructor() payable {
        seller = payable(msg.sender);
        value = msg.value / 2;
        if ((2 * value) != msg.value)
            revert ValueNotEven();
    }

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        // We use transfer here directly. It is
        // reentrancy-safe, because it is the
        // last call in this function and we
        // already changed the state.
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = payable(msg.sender);
```

```
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first
because
        // otherwise, the contracts called using `send`
below
        // can call in again here.
        state = State.Release;

        buyer.transfer(value);
    }

    /// This function refunds the seller, i.e.
    /// pays back the locked funds of the seller.
    function refundSeller()
        public
        onlySeller
        inState(State.Release)
    {
        emit SellerRefunded();
        // It is important to change the state first
because
        // otherwise, the contracts called using `send`
below
        // can call in again here.
        state = State.Inactive;

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

3.3.4 کانال پرداخت خرد[1]

در این بخش نحوه ساختن نمونه پیاده سازی کانال پرداخت[2] را خواهیم آموخت. کانال پرداخت از امضاهای رمزنگاری شده برای انتقال مکرر اتر بین طرف‌های مشابه به صورت ایمن، آنی و بدون کارمزد استفاده می‌کند. برای مثال، باید نحوه امضا و تأیید امضاها و راه اندازی کانال پرداخت را درک کنیم.

ایجاد و تأیید امضا

تصور کنید آلیس می‌خواهد مقداری اتر برای باب ارسال کند، یعنی آلیس فرستنده است و باب گیرنده آن می‌باشد.

آلیس فقط باید پیام های خارج از زنجیرهِ امضا شده با رمزنگاری (مثلاً از طریق ایمیل) را به باب بفرستد و این شبیه چک نوشتن است.

آلیس و باب برای تأیید تراکنش‌ها از امضاها استفاده می‌کنند که با قراردادهای هوشمند در اتریوم امکان پذیر است. آلیس یک قرارداد هوشمند ساده خواهد ساخت که به او امکان می‌دهد اتر را منتقل کند، اما به جای اینکه خودش یک تابع را برای شروع پرداخت فراخوانی کند، به باب اجازه این کار را می‌دهد و بنابراین هزینه تراکنش را پرداخت می‌کند.

قرارداد به شرح زیر کار می‌کند:

1. آلیس قرارداد `ReceiverPays` را دیپلوی می‌کند، به اندازه کافی اتر را برای پوشش پرداخت‌هایی که انجام خواهد شد، پیوست می‌کند.
2. آلیس با امضای پیام با کلید خصوصی خود اجازه پرداخت را می‌دهد.

---

[1] Micropayment Channel [2] payment channel

3. آلیس پیام امضا شده با رمزنگاری را برای باب می‌فرستد. نیازی به مخفی نگه داشتن پیام نیست (بعداً توضیح داده خواهد شد) و سازوکار ارسال آن اهمیتی ندارد.
4. باب با ارائه پیام امضا شده به قرارداد هوشمند، پرداخت خود را مدعی می‌شود. قرارداد صحت پیام را تأیید می‌کند و سپس وجوه را آزاد می‌کند.

ایجاد امضا

آلیس برای امضای تراکنش نیازی به تعامل با شبکه اتریوم ندارد، روند کار کاملا آفلاین است. در این آموزش، ما با استفاده از روش توصیف شده در EIP-762 پیام‌ها را در مرورگر با استفاده از web3.js و متامسک[1] امضا خواهیم کرد، زیرا تعدادی از مزایای امنیتی دیگر را فراهم می‌کند.

```
/// Hashing first makes things easier
var hash = web3.utils.sha3("message to sign");
web3.eth.personal.sign(hash, web3.eth.defaultAccount,
function () { console.log("Signed"); });
```

توجه داشته باشید
`web3.eth.personal.sign` طول پیام را به داده‌های امضا شده اضافه می‌کند. از آنجا که ما ابتدا هش می‌کنیم، پیام همیشه دقیقاً 32 بایت خواهد بود و بنابراین این پیشوند طول[2] همیشه یکسان است.

چه چیزی برای امضا

برای قراردادی که پرداخت‌ها را انجام می‌دهد، پیام امضا شده باید شامل موارد زیر باشد:

1. آدرس گیرنده.
2. مبلغی که باید منتقل شود.

[1] MetaMask

[2] length prefix

3. محافظت در برابر حملات مجدد.[1]

حمله مجدد زمانی رخ می‌دهد که از پیام امضا شده مجدداً برای درخواست مجوز برای اقدام دیگر استفاده می‌شود. برای جلوگیری از حملات مجدد، ما از همان روش تراکنش اتریوم استفاده می‌کنیم، اصطلاحاً نانس[2] نامیده می‌شود، یعنی تعداد تراکنش‌های ارسال شده توسط یک حساب می‌باشد. قرارداد هوشمند بررسی می‌کند که آیا یک نانس چندین بار استفاده شده‌است.

نوع دیگر حمله مجدد می‌تواند هنگامی رخ دهد که مالکِ قرارداد هوشمند `ReceiverPays` را دیپلوی[3] کند، مقداری پرداخت انجام بدهد و سپس قرارداد را از بین ببرد. بعداً، آنها تصمیم می‌گیرند که قرارداد هوشمند `RecipientPays` را دوباره دیپلوی کنند، اما قرارداد جدید، نانس[4] استفاده شده در دیپلوی قبلی را نمی‌شناسد، بنابراین مهاجم می‌تواند دوباره از پیام‌های قدیمی استفاده کند.

آلیس می‌تواند با درج آدرس قرارداد در پیام در برابر این حمله محافظت کند و فقط پیام‌های حاوی آدرس قرارداد خود پذیرفته می‌شوند. نمونه‌ای از این مورد را می‌توانید در دو خط اول تابع `claimPayment()` در قرارداد کامل در انتهای این بخش بیابید.

بسته بندی آرگومان‌ها

حالا که ما مشخص کرده‌ایم که چه اطلاعاتی را باید در پیام امضا شده قرار دهیم، ما آماده هستیم که پیام را کنار هم قرار دهیم و آن را هش و امضا کنیم.

برای سادگی، داده‌ها را بهم پیوند می‌دهیم. کتابخانه ethereumjs-abi تابعی به نام `soliditySHA3` را فراهم می‌کند که رفتار `keccak256` سالیدیتی را که برای آرگومان‌های رمزگذاری شده با استفاده

---

[1] replay attacks
[2] nonce
[3] deploy
4 nonce

از `abi.encodePacked` اعمال می‌شود، را تقلید می‌کند. در اینجا یک تابع جاوا اسکریپت وجود دارد که امضای مناسب را برای مثال `ReceiverPays` ایجاد می‌کند:

```
// recipient is the address that should be paid.
// amount, in wei, specifies how much ether should be sent.
// nonce can be any unique number to prevent replay attacks
// contractAddress is used to prevent cross-contract replay attacks
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);
}
```

بازیابی امضای پیام در سالیدیتی

به طور کلی، امضاهای ECDSA از دو پارامتر `r` و `s` تشکیل شده‌است. امضاها در اتریوم شامل یک پارامتر سوم به نام `v` هستند که می‌توانید برای تایید اینکه کدام کلیدخصوصیِ حساب، برای امضای پیام و تراکنش فرستنده بکار رفته‌است، استفاده کنید. سالیدیتی یک تابع داخلی `ecrecover` را ارائه می‌دهد که با استفاده از پارامترهای `r` ، `s` و `v` یک پیام را می‌پذیرد و آدرسی را که برای امضای پیام استفاده شده بود، برمی‌گرداند.

استخراج پارامترهای امضا

امضاهای تولید شده توسط web3.js بهم پیوستن `r` ، `s` و `v` هستند، بنابراین اولین قدم جدا کردن این پارامترها از یکدیگر است. شما می‌توانید این کار را در سمت کلاینت انجام دهید، اما انجام آن در داخل قرارداد هوشمند به این معنی است که شما فقط باید یک پارامتر امضا را بجای سه پارامتر ارسال کنید. جداسازی آرایه

بایت[1] به قسمت‌های تشکیل دهنده آن یک خرابکاری است، بنابراین ما برای انجام این کار در تابع `splitSignature` از اسمبلی درون خطی[2] استفاده می‌کنیم (سومین تابع در قرارداد کامل در انتهای این بخش).

محاسبه پیام هش

قرارداد هوشمند باید دقیقاً بداند چه پارامترهایی امضا شده‌اند، بنابراین باید پیام را از طریق پارامترها دوباره ایجاد کند و از آن برای تأیید امضا استفاده کند. توابع `prefixed` و `recoverSigner` این کار را در تابع `claimPayment` انجام می‌دهند.

قرارداد کامل

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() payable {}

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

        // this recreates the message that was signed on
the client
        bytes32 message =
prefixed(keccak256(abi.encodePacked(msg.sender, amount,
nonce, this)));
```

[1] byte array

[2] inline assembly

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

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

    /// destroy the contract and reclaim the leftover
funds.
    function shutdown() public {
        require(msg.sender == owner);
        selfdestruct(payable(msg.sender));
    }

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

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32
bytes).
            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);
    }

    /// builds a prefixed hash to mimic the behavior of
eth_sign.
    function prefixed(bytes32 hash) internal pure returns
(bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n32", hash));
    }
}
```

نوشتن یک کانال پرداخت[1] ساده

اکنون آلیس یک پیاده سازی ساده اما کامل از یک کانال پرداخت ایجاد کرده‌است. کانال‌های پرداخت از امضاهای رمزنگاری شده برای انتقال مکرر اتر به صورت ایمن، فوری و تراکنش بدون کارمزد استفاده می‌کنند.

کانال پرداخت چیست؟

کانال‌های پرداخت به شرکت کنندگان این امکان را می‌دهد تا انتقال‌های مکرر اتر را بدون استفاده از تراکنش انجام دهند. این بدان معنی است که شما می‌توانید از تأخیر و هزینه‌های مرتبط با تراکنش‌ها جلوگیری کنید. ما می‌خواهیم یک کانال پرداخت یک طرفه ساده بین دو طرف (آلیس و باب) را بررسی کنیم. شامل سه مرحله است:

1. آلیس قرارداد هوشمند همراه با اتر را تأمین می‌کند. این کانال پرداخت را "باز" می‌کند.

[1] Payment Channel

2. آلیس پیام‌هایی را امضا می‌کند که مشخص می‌کند چه مقدار از آن اتر به گیرنده بدهکار است. این مرحله برای هر پرداخت تکرار می‌شود.
3. باب کانال پرداخت را "می بندد"، سهم خود را از اتر پس می‌گیرد و باقیمانده را به فرستنده ارسال می‌کند.

توجه داشته باشید
فقط مراحل 1 و 3 به تراکنش‌های اتریوم نیاز دارند، مرحله 2 به این معنی است که فرستنده از طریق روش‌های غیر زنجیره‌ای (به عنوان مثال ایمیل) پیام امضا شده رمزنگاری شده را به گیرنده منتقل می‌کند. این بدان معناست که برای پشتیبانی از هر تعداد تراکنش فقط دو تراکنش لازم است.

تضمین شده که باب وجوه[1] خود را دریافت می‌کند زیرا قرارداد هوشمند اتر را اسکو[2] می‌کند (اسکو به معنی اینکه طرفین معامله بر یک سری شرایط توافق میکنند که اگر این شرایط توسط دو طرفین انجام شوند طرف ثالث مانند قرارداد هوشمند امکان برداشت یا پرداخت وجوه را امکان پذیر میکند.) و قول یک پیام معتبر امضا شده را می‌دهد. قرارداد هوشمند همچنین مهلت زمانی را اعمال می‌کند، بنابراین آلیس تضمین می‌کند که سرانجام وجوه خود را بازیابی می‌کند حتی اگر گیرنده از بستن کانال خودداری کند. شرکت کنندگان در یک کانال پرداخت تصمیم میگیرند که چه مدت آن را باز نگه دارند. برای یک معامله کوتاه مدت، مانند پرداخت کافی نت[3] به ازای هر دقیقه دسترسی به شبکه، کانال پرداخت ممکن است به مدت محدودی باز نگه داشته شود. از طرف دیگر، برای پرداخت مکرر، مانند پرداخت دستمزد ساعتی به یک کارمند، کانال پرداخت ممکن است برای چندین ماه یا سال باز نگه داشته شود.

[1] funds [2] escrow [3] internet café

باز کردن کانال پرداخت

برای باز کردن کانال پرداخت، آلیس قرارداد هوشمند را دیپلوی می‌کند، اتر را برای تضمین[1] ضمیمه می‌کند و دریافت کننده مورد نظر و حداکثر مدت زمان[2] وجود کانال را مشخص می‌کند. در انتهای این بخش این تابع `SimplePaymentChannel` در قرارداد است.

ایجاد پرداخت ها

آلیس با ارسال پیام های امضا شده به باب، پرداخت‌ها را انجام می‌دهد. این مرحله کاملاً خارج از شبکه اتریوم انجام می‌شود. پیام‌ها به صورت رمزنگاری شده توسط فرستنده امضا می‌شوند و سپس مستقیماً به گیرنده ارسال می‌شوند.

هر پیام شامل اطلاعات زیر است:

- آدرسِ قراردادِ هوشمند استفاده شده برای جلوگیری از حملات مجدد[3] بین قراردادی.
- مقدارِ کلِ اتری که تاکنون به گیرنده بدهکار است.

در پایان یک سری انتقال‌ها، فقط یک بار کانال پرداخت بسته می‌شود. به همین دلیل، فقط یکی از پیام‌های ارسالی استفاده می‌شود. به همین دلیل است که هر پیام مجموع مقدار کل اتر بدهکار[4] را به جای مقدار جداگانه کانال پرداخت[5] مشخص می‌کند.گیرنده به طور طبیعی آخرین پیام را بخاطر اینکه بالاترین جمع کل[6] را دارد، برای بازخرید[7] انتخاب خواهد کرد. دیگر به نانس[8] برای هر پیام نیاز نمی‌باشد زیرا قرارداد هوشمند فقط به یک پیام پایبنداست. برای جلوگیری از استفاده پیامی که در نظر گرفته شده برای یک کانال پرداخت در کانال دیگر، از آدرس قرارداد هوشمند همچنان استفاده می‌شود.

---

[1] escrow
[2] maximum duration
[3] replay attacks
[4] cumulative total amount of Ether owed
[5] amount of the individual micropayment
[6] highest total
[7] redeem
[8] nonce

در اینجا کد جاوا اسکریپت ویرایش شده برای امضای یک پیام به صورت رمزنگاری از بخش قبلی وجود دارد:

```
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 is used to prevent cross-contract replay
attacks.
// amount, in wei, specifies how much Ether should be sent.

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

بستن کانال پرداخت[1]

هنگامی که باب آماده دریافت وجوه[2] خود باشد، وقت آن است که با فراخوانی تابع `close` در قرارداد هوشمند کانال پرداخت را ببندید. بستن کانال به گیرنده اتری که بدهکار است را پرداخت می‌کند و قرارداد را از بین می‌برد و اتر باقی مانده را برای آلیس می‌فرستد. برای بستن کانال، باب باید پیامی را امضا کند که توسط

[1] Closing the Payment Channel

[2] funds

آلیس امضا شده باشد. قرارداد هوشمند باید تأیید کند که پیام حاوی یک امضای معتبر از طرف فرستنده است. روند انجام این تأیید همان روندی است که گیرنده از آن استفاده می‌کند. توابع سالیدیتی `isValidSignature` و `recoverSigner` درست همانند رونوشت‌های جاوا اسکریپت[1] در بخش قبلی با تابع آخری که از قرارداد `ReceiverPays` گرفته شده کار می‌کند.

فقط گیرنده کانال پرداخت می‌تواند تابع `close` را فراخوانی کند، که به طور طبیعی جدیدترین پیام پرداخت را ارسال می‌کند زیرا این پیام بیشترین مجموع بدهی[2] را دارد. اگر فرستنده اجازه فراخوانی این تابع را داشته باشد، می‌تواند پیامی با مقدار کمتری ارائه دهد و گیرنده را از آنچه طلبکار است، فریب دهد.

تابع تأیید می‌کند که پیام امضا شده با پارامترهای داده شده مطابقت دارد. اگر همه چیز بررسی شود، به گیرنده بخشی از اتر ارسال می شود، و بقیه را از طریق `selfdestruct` برای فرستنده ارسال می‌کند. تابع `close` را می‌توانید در قراردادِ کامل مشاهده کنید.

انقضا کانال[3]

باب می‌تواند در هر زمان کانال پرداخت را ببندد، اما اگر آنها موفق به انجام این کار نشوند، آلیس به راهی برای بازیابی وجوه پس انداز شده[4] خود نیاز دارد. زمان انقضا در زمان استقرار قرارداد تعیین می‌شود. پس از رسیدن به این زمان، آلیس می‌تواند با فراخوانی تابع `claimTimeout` وجوه خود پس بگیرد. تابع `claimTimeout` را می‌توانید در قرارداد کامل مشاهده کنید. بعد از فراخوانی این تابع، باب دیگر نمی‌تواند هیچ اتری دریافت کند، بنابراین مهم است که باب قبل از رسیدن به زمان انقضا، کانال را ببندد.

---

[1] JavaScript counterparts
[2] highest total owed
[3] Channel Expiration
[4] escrowed funds

قرارداد کامل

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract SimplePaymentChannel {
    address payable public sender;      // The account
sending payments.
    address payable public recipient;   // The account
receiving the payments.
    uint256 public expiration;  // Timeout in case the
recipient never closes.

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

    /// the recipient can close the channel at any time by
presenting a
    /// signed amount from the sender. the recipient will
be sent that amount,
    /// and the remainder will go back to the sender
    function close(uint256 amount, bytes memory signature)
public {
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

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

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);
```

```
        expiration = newExpiration;
    }

    /// if the timeout is reached without the recipient
closing the channel,
    /// then the Ether is released back to the sender.
    function claimTimeout() public {
        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)));

        // check that the signature is from the payment
sender
        return recoverSigner(message, signature) == sender;
    }

    /// All functions below this are just taken from the
chapter
    /// 'creating and verifying signatures' chapter.

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

        assembly {
            // first 32 bytes, after the length prefix
            r := mload(add(sig, 32))
```

```
            // second 32 bytes
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes)
            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);
    }

    /// builds a prefixed hash to mimic the behavior of
eth_sign.
    function prefixed(bytes32 hash) internal pure returns
(bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum
Signed Message:\n32", hash));
    }
}
```

توجه داشته باشید
تابع `splitSignature` از همه بررسی‌های امنیتی[1] استفاده نمی‌کند. برای پیاده سازی واقعی باید از یک کتابخانه تست شده با دقت بیشتری استفاده کرد، مانند نسخه openzepplin از این کد.

[1] security checks

تأیید پرداخت‌ها

برخلاف بخش قبلی، پیام‌های موجود در یک کانال پرداخت[1] بلافاصله استفاده نمی‌شوند. گیرنده آخرین پیام را پیگیری می‌کند و هنگامی که زمان بستن کانال پرداخت باشد، آن را استفاده می‌کند. این به این معنی است که بسیار مهم می‌باشد که گیرنده تأیید خود را برای هر پیام انجام دهد. در غیر این صورت هیچ تضمینی وجود ندارد که در پایان گیرنده بتواند وجوه را دریافت کند. گیرنده باید هر پیام را با استفاده از روند زیر تأیید کند:

1- تأیید کند که آدرس قرارداد در پیام با کانال پرداخت مطابقت دارد.

2. تأیید کند که کل جدید[2]، مقدار مورد انتظار است.

3. تأیید کند که کل جدید[3] از مقدار اتر پس انداز شده[4] بیشتر نیست.

4. تأیید کند که امضا معتبر است و از طرف فرستنده کانال پرداخت می‌باشد.

برای نوشتن این تأیید از کتابخانه ethereumjs-util استفاده خواهیم کرد. مرحله آخر را می‌توان به روش‌های مختلفی انجام داد، و ما از جاوا اسکریپت استفاده می‌کنیم. کد زیر تابع `constructMessage` را از امضای کد جاوا اسکریپت در بالا گرفتیم:

```
// this mimics the prefixing behavior of the eth_sign JSON-RPC method.
function prefixed(hash) {
    return ethereumjs.ABI.soliditySHA3(
        ["string", "bytes32"],
        ["\x19Ethereum Signed Message:\n32", hash]
    );
}
```

[1] payment channel
[2] new total
[3] new total
[4] Ether escrowed

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

3.3.5 قراردادهای ماژولی[1]

یک رویکرد ماژولی برای ساخت قراردادها، در کاهش پیچیدگی‌ها و بهبود خوانایی، که به شناسایی باگ‌ها و آسیب پذیری‌ها هنگام توسعه و بررسی کد کمک می‌کند. اگر رفتار یا هر ماژول را جداگانه تعیین و کنترل کنید، فعل و انفعالاتی را باید فقط در روابط بین مشخصات ماژول در نظر بگیرید و نه هر قسمت متحرک دیگر قرارداد.

در مثال زیر، قرارداد از روش `move` کتابخانه `Balances` برای بررسی اینکه بالانس‌های ارسال شده بین آدرس‌ها با آنچه شما انتظار دارید مطابقت دارد، استفاده می‌کند. به این ترتیب، کتابخانه `Balances` یک جز جداگانه است که موجودی حساب‌ها را به درستی ردیابی می‌کند را ارائه می‌دهد. به راحتی می‌توان تأیید کرد

[1] Modular Contracts

که کتابخانه `Balances` هرگز موجودی یا سرریز منفی[1] تولید نمی‌کند و مجموع کل موجودی در طول مدت قرارداد ثابت نیست.

```
// 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);

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

    }
```

[1] negative balances or overflows

```
    function transferFrom(address from, address to, uint
amount) public 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) public
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) public view
returns (uint balance) {
        return balances[tokenOwner];
    }
}
```

3.4 چیدمان یک فایل منبع سالیدیتی[1]

فایل‌های منبع[2] می‌توانند حاوی تعداد دلخواهی از تعاریف قرارداد، دستورهای ایمپورت[3]، دستورهای پراگما، struct، enum، تابع، خطا، تعاریف متغیر ثابت باشند.

3.4.1 مشخص کننده‌ی لایسنس[4] SPDX

[1] Layout of a Solidity Source File
[2] Source files
[3] Import
[4] License Identifier

در صورت در دسترس بودن کد منبع[1]، اعتماد به قرارداد هوشمند می‌تواند بهتر ایجاد شود. از آنجا که در دسترس قرار دادن کد منبع همیشه مشکلات حقوقی مربوط به حق چاپ را تحت تأثیر قرار می دهد، کامپایلر سالیدیتی استفاده از مشخص کنندهِ لایسنس SPDX قابلِ خوانده شدن توسطِ ماشین را ترغیب می‌کند. هر فایل منبع باید با یک کامنت که نشان دهد لایسنس آن است شروع شود:

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

کامپایلر تأیید نمی‌کند که مجوز بخشی از لیستی است که توسط SPDX مجاز است، اما رشته ارائه شده در فراداده بایت کد[2] را شامل می‌شود.

اگر نمی‌خواهید مجوزی را تعیین نکنید یا اگر کد منبع، منبع باز نیست؛ لطفاً از مقدار ویژه `UNLICENSED` استفاده کنید. البته ارائه این کامنت شما را از سایر تعهدات مربوط به صدور لایسنس مانند نیاز به ذکر مجوز خاص در هِدر هر فایل منبع یا دارنده اصلی حق چاپ خلاص نمی کند.

کامنت توسط کامپایلر در هر کجای فایل در سطح فایل شناسایی می‌شود، اما توصیه می‌شود آن را در بالای فایل قرار دهید.

اطلاعات بیشتر در مورد نحوه استفاده از مشخص کننده لایسنس SPDX را می‌توانید در وب سایت SPDX بیابید.

3.4.2 پراگماها[3]

کلمه کلیدی `pragma` یا پراگما برای فعال کردن برخی از ویژگی‌های کامپایلر یا بررسی‌ها استفاده می‌شود. یک دستورالعمل پراگما همیشه محلی برای یک فایل منبع است، بنابراین اگر می‌خواهید آن را در کل پروژه خود

[1] source code
[2] bytecode metadata
[3] Pragmas

فعال کنید، باید پراگما را به تمام فایل‌های خود اضافه کنید. اگر فایل دیگری را ایمپورت[1] کنید، پراگمای آن فایل به طور خودکار در فایل وارد شده اعمال نمی‌شود.

نسخه پراگما

فایل‌های منبع را می‌توان (و باید) با یک نسخه[2]، نسخه بندی کرد تا کامپایل با نسخه‌های کامپایلر آتی را رد کند که ممکن است تغییرات ناسازگار را معرفی کند. ما سعی می‌کنیم این موارد را به حداقل برسانیم و آنها را به گونه ای معرفی کنیم که تغییر در سِمَنتیک‌ها[3] نیاز به تغییرسینتکس[4] داشته باشد، اما این امر همیشه امکان پذیر نیست. به همین دلیل، همیشه ایده خوبی است که حداقل برای نسخه‌هایی که شامل تغییرات خراب کننده هستند، از تغییرات جدید[5] استفاده کنید. این نسخه‌ها همیشه نسخه‌هایی از فرم `0.x.0` یا `x.0.0` دارند.

نسخه پراگما به شرح زیر استفاده می شود: `pragma solidity ^0.5.2;`

یک فایل منبع با خط بالا با یک کامپایلر جدیدتر از نسخه 0.5.2 کامپایل نمی‌شود و همچنین در یک کامپایلر که از نسخه 0.6.0 شروع می‌شود کار نمی‌کند (این شرط دوم با استفاده از `^` اضافه می‌شود). از آنجا که تا قبل از نسخه `0.6.0` هیچ تغییر جدید[6] ایجاد نخواهد شد، می‌توانید مطمئن باشید که کد شما به همان روشی که شما در نظر داشتید، کامپایل می‌شود. نسخه دقیق کامپایلر ثبت نشده است، بنابراین انتشار رفع خطا[7] همچنان امکان پذیر است.

می‌توان قوانین پیچیده‌تری را برای نسخه کامپایلر تعیین کرد، اینها از همان سینتکسی[8] استفاده می‌کنند که توسط npm استفاده می‌شود.

توجه داشته باشید

[1] Import
[2] version
[3] semantics
[4] syntax
[5] breaking changes
[6] breaking changes
[7] bugfix
[8] syntax

با استفاده از نسخه pragma نسخه کامپایلر تغییر نمی‌کند. همچنین ویژگی‌های کامپایلر را فعال یا غیرفعال نمی‌کند. این فقط به کامپایلر دستور می‌دهد که بررسی کند آیا نسخه آن با نسخه مورد نیاز پراگما مطابقت دارد یا خیر. اگر مطابقت نداشته باشد، کامپایلر خطایی را صادر می‌کند.

ABI Coder Pragma

با استفاده از `pragma abicoder v1` یا `pragma abicoder v2` می‌توانید از بین دو پیاده سازی رمزگذار[1] و رمزگشای[2] ABI یکی را انتخاب کنید.

رمزگذار جدید ABI (v2) قادر به رمزگذاری و رمزگشایی آرایه‌ها و structهای دلخواه تو در تو است. ممکن است کد بهینه کمتری تولید کند و به اندازه رمزگذار قدیمی تست نشده باشد، اما از نظر سالیدیتی نسخه 0.6.0 غیر آزمایشی محسوب می‌شود. شما هنوز هم باید با استفاده از `pragma abicoder v2;` آن را صریحاً فعال کنید. در سالیدیتی نسخه 0.8.0 به طور پیش فرض فعال می‌شود، گزینه‌ای برای انتخاب کدگذار قدیمی با استفاده از `pragma abicoder v1;` وجود دارد.

مجموعه نوع‌های[3] پشتیبانی شده توسط رمزگذار جدید یک مجموعه فوق العاده دقیق از نوع‌های پشتیبانی شده توسط رمزگذار قدیمی است. قراردادهایی که از آن استفاده می‌کنند می‌توانند با قراردادهایی که بدون محدودیت نیستند ارتباط برقرار کنند. بازگشت[4] فقط تا زمانی امکان پذیر است که قرارداد غیر `abicoder v2` سعی در فراخوانی‌هایی نداشته باشد که نیاز به انواع رمزگشایی داشته باشند که فقط توسط رمزگذار جدید پشتیبانی می‌شوند. کامپایلر می‌تواند این مورد را تشخیص دهد و خطایی ایجاد کند. فعال کردن `abicoder v2` قرارداد برای اینکه خطا برطرف شود، کافی است.

---

[1] encoder
[2] decoder
[3] types
[4] reverse

توجه داشته باشید
این پراگما برای همه کدهای تعریف شده در فایل در جایی که فعال شده‌است اعمال می‌شود، صرف نظر از اینکه سرانجام این کد به کجا ختم می‌شود. این بدان معناست که قراردادی که فایل منبع آن برای کامپایل با کدگذار ABI v1 انتخاب شده‌است، همچنان می‌تواند حاوی کدی باشد که با به ارث بردن رمزگذار جدید از قرارداد دیگر، از رمزگذار جدید استفاده کند. این درصورتی مجاز است که نوع‌های جدید فقط در داخل استفاده شوند و در امضاهای تابع خارجی نباشند.

توجه داشته باشید
تا سالیدیتی نسخه 0.7.4، می‌توان با استفاده از `pragma experimental ABIEncoderV2`، رمزگذار ABI v2 را انتخاب کرد، اما صریحاً رمزگذار v1 را نمی‌توان انتخاب کرد زیرا پیش فرض بود.

پراگما آزمایشی[1]

پراگما دوم[2]، پراگما آزمایشی است. می‌توانند برای فعال کردن ویژگی‌های کامپایلر یا زبان استفاده شوند که هنوز به طور پیش فرض فعال نشده‌اند. پراگماهای آزمایشی زیر در حال حاضر پشتیبانی می‌شوند:

ABIEncoderV2

از آنجا که رمزگذار ABI v2 دیگر آزمایشی محسوب نمی‌شود، می‌توان از طریق سالیدیتی نسخه 0.7.4 از طریق `pragma abicoder v2` آن را انتخاب کرد (لطفاً به قسمت بالا مراجعه کنید).

[1] Experimental Pragma

[2] second pragma

کنترل کننده SMTC[1]

این مؤلفه[2] باید در هنگام ساخت کامپایلر سالیدیتی فعال شود و بنابراین در تمام باینری‌های سالیدیتی در دسترس نیست. دستورالعمل‌های نسخه، نحوه فعال سازی این گزینه را توضیح می‌دهند. برای نسخه‌های اوبنتو[3] PPA در اکثر نسخه‌ها فعال شده، اما برای تصویرهای داکر[4]، باینری‌های ویندوز[5] یا باینری‌های لینوکس نسخه ایستا[6]، فعال نیست. اگر یک حلال کننده SMT[7] را به صورت محلی نصب کرده باشید و solc-js را از طریق گره (نه از طریق مرورگر) اجرا کنید، می‌تواند از طریق smtCallback برای solc-js فعال شود.

اگر `pragma experimental SMTChecker;` استفاده می‌کنید، هشدارهای ایمنی بیشتری دریافت می‌کنید که با پرس و جو از یک حل کننده SMT بدست می‌آیند. این مولفه هنوز از تمام ویژگی‌های زبان سالیدیتی پشتیبانی نمی‌کند و احتمالاً هشدارهای زیادی را در بر داشته باشد. در صورت گزارش ویژگی‌های پشتیبانی نشده، ممکن است تجزیه و تحلیل کاملاً مناسب نباشد.

3.4.3 ایمپورت کردن سایر فایل‌های منبع

سینتکس و سمنتیک[8]

سالیدیتی برای کمک به ماژولی بودن کد شما که مشابه آنچه در جاوا اسکریپت در دسترس است (از ES6 به بعد)، از دستورات ایمپورت[9] پشتیبانی می‌کند. با این حال، سالیدیتی مفهوم اکسپورت به صورت پیشفرض[10] را پشتیبانی نمی‌کند. در سطح جهانی، می‌توانید از دستورات ایمپور به شکل زیر استفاده کنید:

```
import "filename";
```

[1] SMTChecker
[2] component
[3] Ubuntu
[4] Docker images
[5] Windows binaries
[6] the statically-built Linux binaries
[7] SMT solver
[8] Syntax and Semantics
[9] Import
[10] default export

این عبارت تمام نمادهای جهانی را از "filename " (و نمادهای وارد شده در آنجا) به دامنه جهانی فعلی ایمپورت می‌کند (متفاوت از ES6 اما برای سازگاری سالیدیتی با گذشته[1]). این فرم، برای استفاده توصیه نمی‌شود. زیرا به طور غیر قابل پیش بینی فضای نام را آلوده می‌کند. اگر موارد سطح بالای جدید را به داخل "namename" اضافه کنید، به طور خودکار در همه فایل‌هایی که از این "Filename" ایمپورت می‌شوند ظاهر می‌شوند. بهتر است نمادهای مشخص را به طور صریح وارد کنید.

مثال زیر یک نماد جهانی `symbolName` ایجاد می‌کند که اعضای آن همه نمادهای جهانی از `"filename"` هستند:

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

که منجر به در دسترس بودن همه نمادهای جهانی در قالب `symbolName.symbol` می‌شود.

گونه‌ای از این سینتکس که بخشی از ES6 نیست، اما احتمالاً قابل استفاده باشد:

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

که معادل `import * as symbolName from "filename";` است.

در صورت تصادم نامگذاری، می‌توانید هنگام ایمپورت کردن، نمادها را تغییر نام دهید. به عنوان مثال، کد زیر نمادهای جهانی جدید `alias` و `symbol2` را ایجاد می‌کند که به ترتیب از داخل `"filename"` به `symbol1` و `symbol2` مراجعه می‌کند.

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

[1] backwards-compatible

مسیرها[1]

در موارد فوق، `filename` همیشه به عنوان مسیری که با `/` به عنوان جدا کننده دایرکتوری و `.` به عنوان دایرکتوری فعلی[2] و `..` به عنوان دایرکتوری والد، رفتار می‌کند. زمانی که `.` یا `..` با یک کاراکتر به جز `/` دنبال شود، به عنوان پوشه اصلی یا والدین در نظر گرفته نمی‌شود. همه نام مسیرها به عنوان مسیرهای کامل برخورد می‌شوند مگر اینکه با دایرکتوری فعلی `.` یا دایرکتوری والدین `..` شروع شوند.

برای وارد کردن فایل `filename` به عنوان دایرکتوری فایل فعلی، از `import "./filename" as symbolName;` استفاده کنید. اگر از `import "filename" as symbolName;` استفاده می‌کنید. در عوض، می‌توان به فایل دیگری ارجاع داد (در " شامل دایرکتوری" جهانی).

در واقع نحوه حل مسیرها بستگی به کامپایلر دارد (به "استفاده در کامپایلرهای واقعی" مراجعه کنید). به طور کلی، دایرکتوری سلسله مراتبی نیازی به نگاشت[3] دقیق بر روی سیستم فایل محلی شما ندارد، و مسیر همچنین می تواند به منابعی مانند http، ipfs یا git نگاشت شود.

توجه داشته باشید
همیشه از ایمپورت‌های نسبی مانند `import "./filename.sol";` استفاده کنید. و در استفاده از `..` در مشخص کننده‌های مسیر[4] خودداری کنید. در حالت دوم، احتمالاً بهتر است از مسیرهای جهانی استفاده کنید و تنظیم مجدد را همانطور که در زیر توضیح داده شده است تنظیم کنید.

[1] Paths
[2] current
[3] map
[4] path specifiers

در کامپایلرهای واقعی استفاده کنید

هنگام فراخوانی[1] کامپایلر، می‌توانید اولین عنصر از یک مسیر و همچنین تغییر مجدد پیشوندهای مسیر را مشخص کنید. به عنوان مثال می‌توانید یک نگاشت مجدد[2] تنظیم کنید تا همه آنچه از پوشه مجازی `github.com/ethereum/dapp-bin/library` کتابخانه ایمپورت می‌شود در واقع از دایرکتوری محلی `/usr/local/dapp-bin/library` کتابخانه شما خوانده شود. در صورت اعمال چندین نگاشت مجدد، ابتدا مسیری که طولانی ترین کلید را دارد امتحان می‌شود. پیشوند[3] خالی مجاز نیست. نگاشت مجدد می‌تواند به زمینه‌ای بستگی داشته باشد، که به شما امکان می‌دهد بسته‌ها را برای ایمپورت کردن، به عنوان مثال، نسخه‌های مختلف کتابخانه به همین نام، پیکربندی کنید.

:solc

برای solc (کامپایلر خط فرمان[4])، شما این تغییر مسیرها را به عنوان آرگومان‌های `context:prefix=target` در نظر می‌گیرید، جایی که هم `context:` و هم `=target` قسمت‌های مورد نظر اختیاری هستند (در این حالت پیش فرض‌های `target` به `prefix`). تمام مقادیر نگاشت مجدد که فایل‌های معمولی هستند، ایمپورت می‌شوند (از جمله وابستگی‌های آنها).

این مکانیزم به صورت رو به عقب سازگار[5] است (تا زمانی که هیچ نام فایلی شامل `=` یا `:` نباشد) و بدین ترتیب تغییر جدید[6] نخواهد بود. همه فایل‌های موجود در دایرکتوری یا در زیر دایرکتوری `context` فایلی را که با `prefix` شروع می‌شود، ایمپورت می‌کنند با جایگزینی `prefix` با `target` دوباره مسیردهی می‌شوند.

---

[1] invoke
[2] remapping
[3] prefix
[4] the commandline compiler
[5] backwards-compatible
[6] breaking change

به عنوان مثال، اگر `github.com/ethereum/dapp-bin/` را به صورت محلی به `/usr/local/dapp-bin` کلون کنید، می‌توانید از موارد زیر در فایل منبع خود استفاده کنید:

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

سپس کامپایلر را اجرا کنید:

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

به عنوان یک مثال پیچیده‌تر، فرض کنید به ماژولی اعتماد می‌کنید که از نسخه قدیمی dapp-bin استفاده می‌کند که در `/usr/local/dapp-bin_old` را بررسی کرده‌اید، سپس می‌توانید اجرا کنید:

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

این بدان معنی است که تمام ایمپورت‌ها در `module2` به نسخه قدیمی اشاره دارند اما ایمپورت در `module1` به نسخه جدید اشاره دارد.

توجه داشته باشید

`solc` فقط به شما امکان می‌دهد فایل‌هایی را از دایرکتوری‌های خاص ایمپورت کنید. آنها باید در یکی از فایل‌های منبع صریحاً مشخص شده یا در دایرکتوری (یا زیر دایرکتوری) یک نگاشت مجدد مقصد[1] باشند. اگر می خواهید شامل مطلق مستقیم باشد ، نگاشت مجدد `/=/` را اضافه کنید.

اگر چندین نگاشت مجدد وجود داشته باشد که منجر به یک فایل معتبر می‌شود، نگاشت مجدد با طولانی‌ترین پیشوند مشترک انتخاب می‌شود.

ریمیکس[2]:

ریمیکس یک نگاشتِ مجددِ خودکار را برای گیتهاب فراهم می‌کند و فایل‌ها را به صورت خودکار از طریق شبکه بازیابی می‌کند. می توانید نگاشت قابل تکرار را مانند بالا ایمپورت کنید، به عنوان مثال:

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

ریمیکس ممکن است در آینده سایر ارائه دهندگان کد منبع را اضافه کند.

کامنت‌ها

کامنت‌های تک خطی (`//`) و کامنت‌های چند خطی (`/*...*/`) امکان پذیر است.

```
// This is a single-line comment.

/*
This is a
multi-line comment.
*/
```

[1] remapping target

[2] Remix

توجه داشته باشید
یک کامنت تک خطی توسط هر پایان دهنده خط unicode (LF، VF، FF، CR، NEL، LS یا PS) در رمزگذاری UTF-8 خاتمه می‌یابد. ترمیناتور[1] بعد از کامنت هنوز بخشی از کد منبع است، بنابراین اگر یک نماد ASCII نباشد (اینها NEL، LS و PS هستند)، منجر به خطای تجزیه می‌شود.

علاوه بر این، نوع دیگری از کامنت به نام کامنت NatSpec وجود دارد که در راهنمای استایل[2] به تفصیل آورده شده است. آنها با یک اسلش سه گانه (`///`) یا یک بلوک ستاره دوتایی (`/** ... */`) نوشته می‌شوند و باید مستقیماً بالاتر از دستورات یا دستورات تابع استفاده شوند.

3.5 ساختار قرارداد

قراردادهای سالیدیتی مانند کلاس‌های زبان‌های شی‌گرا هستند. هر قرارداد می‌تواند شامل دستورات متغیرهای حالت[3]، توابع[4]، توابع اصلاح کننده‌ها[5]، رویدادها[6]، خطاها[7]، انواع Structها[8] و انواع Enumها[9] باشد. علاوه بر این، قراردادها می‌توانند از سایر قراردادها ارث ببرند.

قراردادهای دیگری نیز وجود دارد که کتابخانه‌ها[10] و رابط‌ها[11] نامیده می‌شوند.

بخش مربوط به قراردادها شامل جزئیات بیشتری نسبت به این بخش است که یک مرور سریع رو ارائه می‌دهد.

---

[1] terminator
[2] style guide
[3] State Variables
[4] Functions
[5] Function Modifiers
[6] Events
[7] Errors
[8] Struct Types
[9] Enum Types
[10] libraries
[11] interfaces

3.5.1 متغیرهای حالت[1]

متغیرهای حالت متغیرهایی هستند که مقادیر آنها به طور دائمی در فضای ذخیره سازی[2] قرارداد ذخیره می‌شود.

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

contract SimpleStorage {
    uint storedData; // State variable
    // ...
}
```

برای دیدن انواع متغیرهای حالت معتبر[3] و قابلیت مشاهده[4] آن‌ها و گیرنده‌ها[5] برای انتخاب‌های احتمالی برای مشاهده انواع متغیرها، به بخش انواع[6] مراجعه کنید.

3.5.2 توابع

توابع، واحد اجرایی کد هستند. توابع معمولاً در داخل قرارداد تعریف می‌شوند، اما در خارج از قراردادها نیز می‌توانند تعریف شوند.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.7.0 <0.9.0;

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

// Helper function defined outside of a contract
function helper(uint x) pure returns (uint) {
    return x * 2;
```

[1] State Variables
[2] storage
[3] state variable types
[4] Visibility
[5] Getters
[6] Types

```
}
```

فراخوانی توابع می‌توانند به صورت داخلی یا خارجی اتفاق بیفتند و دارای قابلیت مشاهده مختلفی نسبت به سایر قراردادها هستند. توابع پارامترها را می‌پذیرند و متغیرها را برمی‌گردانند تا پارامترها و مقادیر بین آنها منتقل شود.

3.5.3 توابع اصلاح کننده[1]

از تابع اصلاح کننده‌ها می‌توان برای اصلاح سمنتیک[2] توابع به روشی اعلانی استفاده کرد (به توابع اصلاح کننده در بخش قراردادها مراجعه کنید).

اضافه بار[3]، به این معنا که داشتن نام اصلاح کننده یکسان با پارامترهای مختلف، امکان پذیر نیست. مانند توابع، اصلاح کننده‌ها را می‌توان لغو کرد.

مانند توابع، اصلاح کننده‌ها نیز می‌توانند نادیده گرفته شوند.

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

contract Purchase {
    address public seller;

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

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

[1] Function Modifiers [2] semantics [3] Overloading

```
    }
}
```

3.5.4 رویدادها[1]

رویدادها رابط‌های راحتی برای ورود به امکانات EVM هستند.

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

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

    function bid() public payable {
        // ...
        emit HighestBidIncreased(msg.sender, msg.value); //
Triggering event
    }
}
```

برای اطلاع از چگونگی اعلام رویدادها و استفاده از آنها از طریق dapp، به بخش رویدادها در بخش قراردادها مراجعه کنید.

3.5.5 خطاها[2]

خطاها به شما امکان می‌دهند نام‌ها و داده‌های توصیفی را برای شرایط شکست تعریف کنید. از خطاها می‌توان در دستورات revert استفاده کرد. در مقایسه با توضیحات رشته[3]، خطاها بسیار ارزان‌تر هستند و به شما امکان می‌دهند داده‌های اضافی را رمزگذاری کنید. برای توصیف خطا برای کاربر می‌توانید از NatSpec استفاده کنید.

[1] Events [2] Errors [3] String

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

/// Not enough funds for transfer. Requested `requested`,
/// but only `available` available.
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;
        // ...
    }
}
```

برای اطلاعات بیشتر به خطاها و دستورات Revert در قسمت قراردادها مراجعه کنید.

3.5.6 انواع Struct

Structها انواع تعریف شده سفارشی هستند که می‌توانند متغیرهای مختلفی را گروه بندی کنند (به بخش Structها در بخش انواع مراجعه کنید).

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

contract Ballot {
    struct Voter { // Struct
        uint weight;
        bool voted;
        address delegate;
```

```
        uint vote;
    }
}
```

3.5.7 انواع Enum

از Enums می‌توان برای ایجاد انواع سفارشی با مجموعه محدودی از "مقادیر ثابت[1]" استفاده کرد (به قسمت Enumها در بخش انواع مراجعه کنید).

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

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

انواع[2]

سالیدیتی یک زبان نوع استاتیک است، به این معنی که نوع هر متغیر (state و local) باید مشخص شود. سالیدیتی چندین نوع اصلی را ارائه می‌دهد که می‌توانند با هم ترکیب شوند و انواع پیچیده‌ای را تشکیل دهند.

علاوه بر این، انواع می‌توانند در عبارات حاوی عملگر[3] با یکدیگر تعامل داشته باشند. برای مراجعه سریع به عملگرهای مختلف، به بخش ترتیب تقدم عملگرها مراجعه کنید.

مفهوم مقادیر "undefined " یا "null“ " در سالیدیتی وجود ندارد، اما متغیرهای تازه اعلام شده همیشه یک مقدار پیش فرض وابسته به نوع آنها دارند. برای استفاده از مقادیر غیر منتظره، باید از تابع revert استفاده کنید تا کل تراکنش را برگردانید، یا یک تاپل[4] را با مقدار `bool` دوم[5] نشان دهید که موفقیت را نشان می‌دهد.

انواع مقدار[6]

---

[1] constant values
[2] Types
[3] Operators
[4] tuple
[5] second `bool`
[6] Value Types

انواع زیر را نیز انواع مقدار می‌نامند زیرا متغیرهای این نوع‌ها همیشه از نظر مقدار منتقل می‌شوند، یعنی وقتی که به عنوان آرگومان تابع یا در "انتساب‌ها[1]" استفاده میشوند، همیشه کپی می‌شوند.

بولین[2]

`bool`: مقادیر ممکن ثابت‌های `true` و `false` هستند.

عملگرها:

- `!` (logical negation)
- `&&` (logical conjunction, “and”)
- `||` (logical disjunction, “or”)
- `==` (equality)
- `!=` (inequality)

اپراتورها `||` و `&&` قوانین متداول اتصال کوتاه[3] را اعمال می‌کنند. این بدان معنی است که در عبارت `f(x) || g(y)`، اگر `f(x)` به صورت `true` ارزیابی شود، `g(y)` حتی اگر دارای عوارض جانبی باشد نیز ارزیابی نخواهد شد.

عدد صحیح[4]

`uint` / `int`: عددهای صحیح با علامت و بدون علامت در اندازه های مختلف. کلمات کلیدی `uint8` تا `uint256` در گام‌های `8` (بدون علامت 8 تا 256 بیت) و `int8` تا `int256`. `uint` و `int` به ترتیب نام مستعار برای `uint256` و `int256` هستند.

عملگرها:

- مقایسه‌گرها[5]: `<=`، `<`، `==`، `!=`، `>=`، `>` (ارزیابی به `bool`)
- عملگرهای بیت[6]: `&`، `|`، `^` (یا bitwise exclusive)، `~` (bitwise negation)

[1] assignments
[2] Booleans
[3] short-circuiting rules
[4] Integers
[5] Comparisons
[6] Bit operators

- عملگرهای شیفت کردن[1]: `<<` (شیفت چپ) ، `>>` (شیفت راست)
- عملگرهای حسابی[2]: `-` , `+` ، `-` unary (فقط برای اعداد صحیح با علامت) ، `*` ، `/` ، `%` (باقیمانده) ، `**` (توان)

برای یک عدد صحیح نوع `X` ، می‌توانید از `type(X).min` و `type(X).max` برای دستیابی به حداقل و حداکثر مقدار قابل نمایش توسط نوع استفاده کنید.

هشدار
عدد صحیح در سالیدیتی به محدوده خاصی محدود می‌شود. به عنوان مثال، با `uint32` ، این `0` تا `2**32 - 1` است. دو حالت وجود دارد که عملیات حسابی در این نوع انجام می‌شود: حالت "wrapping" یا "unchecked" و حالت "checked". به طور پیش فرض، عملیات حسابی همیشه "checked" می‌شود، به این معنی که اگر نتیجه عملیاتی خارج از محدوده مقدار نوع باشد، فراخوانی با یک ادعای ناموفق برگردانده می‌شود. می‌توانید با استفاده از `unchecked { ... }` به حالت "unchecked" تغییر دهید. جزئیات بیشتر را می‌توانید در بخش "unchecked" مشاهده کنید.

مقایسه‌گرها[3]

مقدار مقایسه‌گر، مقداری است که با مقایسه مقدار عدد صحیح[4] بدست می‌آید.

عملیات بیت (Bit operations)

عملیات بیت بر روی نمایش مکمل دو انجام می‌شود. این بدان معنی است که به عنوان مثال `~int256(0) == int256(-1)` .

[1] Shift operators
[2] Arithmetic operators
[3] Comparisons
[4] integer value

شیفت‌ها[1]

شیفت‌ها نتیجه یک عمل جابجایی دارای نوع عملوند[2] سمت چپ می‌باشند و نتیجه را متناسب با نوع آن کوتاه می‌کنند. عملوند سمت راست باید از نوع بدون علامت[3] باشد. تلاش برای جابجایی با نوع با علامت[4] خطای کامپایل ایجاد می‌کند.

- برای مقادیر `x` مثبت و منفی، `x << y` معادل با `x * 2**y` است.
- برای مقادیر `x` مثبت، `x >> y` معادل با `x / 2**y` است.
- برای مقادیر `x` منفی، `x >> y` معادل با `(x + 1) / 2**y - 1` است (که همان تقسیم `x` به `2**y` در حالی که به سمت بی نهایت منفی گرد می‌شود) است.

هشدار
قبل از نسخه `0.5.0` شیفت راست `x >> y` برای منفی `x` معادل `x / 2**y` بود، یعنی از شیفت‌های راست به جای گرد کردن (به سمت بی نهایت منفی) از گرد کردن (به سمت صفر) استفاده می‌شد.

جمع، تفریق و ضرب

جمع، تفریق و ضرب سمنتیک معمول را دارند، با توجه به دو حالت مختلف از نظر سرریز[5] و زیرریز[6] به طور پیش فرض، تمام محاسبات زیرریز و سریز بررسی می‌شود، اما این می‌تواند با استفاده از unchecked block غیرفعال شود، در نتیجه محاسبات پیچیده می‌شود. جزئیات بیشتر را می‌توان در آن بخش یافت.

عبارت `-x` برابر است با `(T(0) - x)` که `T` نوع `x` است. فقط در انواع امضا شده قابل استفاده است. اگر `x` منفی باشد مقدار `-x` می‌تواند مثبت باشد. اخطار دیگری نیز وجود دارد که ناشی از نمایش مکمل دو[7] است:

---

[1] Shifts
[2] operand
[3] unsigned type
[4] signed type
[5] overflow
[6] underflow
[7] two's complement representation

اگر `int x = type(int).min;` داشته باشید، `-x` با رنج مثبت متناسب نیست. این به این معنی است که `unchecked { assert(-x == x); }` کار می‌کند، و عبارت `-x` هنگامی که در حالت checked استفاده شود، منجر به اعلان شکست[1] می‌شود.

تقسیم[2]

از آنجا که نوع نتیجه یک عملیات همیشه نوع یکی از عملوندها[3] است، تقسیم بر اعداد صحیح همیشه منجر به یک عدد صحیح می‌شود. در سالیدیتی، تقسیم به سمت صفر گرد می‌شود. این بدان معنی است که

`int256(-5) / int256(2) == int256(-2)` .

توجه داشته باشید که در مقابل، تقسیم بر روی لیترال‌ها[4] منجر به مقادیر کسری دلخواه می‌شود.

توجه داشته باشید
تقسیم بر صفر باعث خطای Panic می‌شود. این بررسی از طریق `unchecked { ...}` غیرفعال نمی‌شود.

توجه داشته باشید
عبارت `type(int).min / (-1)` موردی است که تقسیم باعث سرریز[5] می‌شود. در حالت حسابی بررسی شده[6]، باعث اعلان شکست می‌شود، در حالی که در حالت wrapping ، مقدار `type(int).min` خواهد بود.

باقیمانده[7]

عملیات باقیمانده `a % n` پس از تقسیم عملوند[8] `a` توسط عملوند `n` ، باقی مانده `r` را حاصل می‌شود، جایی که `q = int(a / n)` و `r = a - (n * q)` . این بدان معناست که باقیمانده همان

[1] failing assertion
[2] Division
[3] operands
[4] literals
[5] overflow
[6] checked arithmetic mode
[7] Modulo
[8] operand

علامت عملوند سمت چپ (یا صفر) خود را نشان می‌دهد و `a % n == -(-a % n)` برای منفی `a` نگه می‌دارد:

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

توجه داشته باشید
باقیمانده با صفر باعث خطای Panic می‌شود. این بررسی از طریق `unchecked { ... }` غیرفعال نمی‌شود.

به توان رساندن[1]

به توان رساندن فقط برای انواع بدون علامت[2] در توان[3] در دسترس است. نوع توان در نتیجه همیشه با نوع پایه برابر است. لطفاً توجه داشته باشید که به اندازه کافی بزرگ باشد تا بتواند نتیجه را حفظ کند و برای اعلان شکست احتمالی یا رفتار پیچیده آماده شود.

توجه داشته باشید
در حالت بررسی شده[4]، توان فقط از آپکد `exp` نسبتاً ارزان برای پایه‌های کوچک استفاده می‌کند. برای موارد `x**3` ، ممکن است عبارت `x*x*x` ارزان تر باشد. در هر صورت، تست هزینه گاز و استفاده از بهینه ساز توصیه می‌شود.

توجه داشته باشید
توجه داشته باشید که `0**0` توسط EVM به صورت `1` تعریف می‌شود.

[1] Exponentiation
[2] unsigned types
[3] exponent
[4] checked mode

عدد ممیز ثابت [1]

هشدار
عدد ممیز ثابت هنوز توسط سالیدیتی کاملاً پشتیبانی نمی‌شوند. می‌توان آن‌ها را مشخص کرد، اما نمی‌توان آنها را به چیزی یا از چیزی اختصاص داد.

`fixed` / `ufixed` : اعداد ثابت بدون علامت و باعلامت دراندازه‌های مختلف. کلمات کلیدی `ufixedMxN` و `fixedMxN` ، جایی که `M` تعداد بیت‌های گرفته شده توسط نوع را نشان می‌دهد و `N` نشان دهنده تعداد اعشار در دسترس است. `M` باید بر 8 قابل تقسیم باشد و از 8 به 256 بیت تبدیل شود. `N` باید شامل 0 تا 80 باشد. `ufixed` و `fixed` به ترتیب نام‌های مستعار برای `ufixed128x18` و `fixed128x18` هستند.

عملگرها:

- مقایسه ها `<=` ، `<` ، `==` ، `!=` ، `>=` ، `>` (ارزیابی به `bool`)
- عملگرهای حسابی: `+` ، `-` ، unary `-` ، `*` ، `/` ، `%` (باقیمانده)

توجه داشته باشید
تفاوت اصلی بین اعداد float (`float` و `double` در بسیاری از زبانها، به طور دقیق‌تر اعداد IEEE 754) و عدد ممیز ثابت در این است که تعداد بیت‌های مورد استفاده برای عدد صحیح[2] و قسمت کسری[3] (قسمت بعد از نقطه اعشاری[4]) در قبل انعطاف پذیر است، در حالی که در دومی به طور دقیقاً تعریف شده‌است. به طور کلی، در اعداد float تقریباً از کل فضا برای نشان دادن عدد[5] استفاده می‌شود، در حالی که فقط تعداد کمی بیت مکان نقطه اعشار را تعریف می‌کنند.

آدرس[6]

نوع آدرس به دو صورت وجود دارد که تا حد زیادی یکسان هستند:

[1] Fixed Point Numbers
[2] integer
[3] fractional part
[4] Decimal point
[5] number
[6] Address

- `address`: دارای مقدار 20 بایت (اندازه آدرس اتریوم) است.
- `address payable`: همان `address` است، اما با اعضای اضافی `transfer` و `send`.

ایده پشت این تمایز این است که `address payable` آدرسی است که می‌توانید اتر را به آن بفرستید، در حالی که نمی‌توان با یک `address` ساده اتر ارسال کرد.

تبدیل‌های نوع:

تبدیل‌های ضمنی از `address payable` به `address` مجاز است، در حالی که تبدیل از `address` به `address payable` باید از طریق `payable(<address>)` صریح باشد.

تبدیل صریح به و از `address` برای آدرس‌های `uint160`، لیترال‌های عدد صحیح، و `bytes20` انواع قرارداد مجاز است.

فقط عبارات نوع `address` و نوع قرارداد را می توان از طریق تبدیل صریح `payable(...)` به `address payable` تبدیل کرد. برای نوع قرارداد، این تبدیل فقط در صورتی مجاز است که قرارداد بتواند اتر را دریافت کند، به عنوان مثال، قرارداد تابع دریافت[1] یا برگشتی قابل پرداخت[2] داشته باشد. توجه داشته باشید که `payable(0)` معتبر است و از این قاعده مستثنی است.

توجه داشته باشید
اگر به متغیر نوع `address` نیاز دارید و قصد دارید اتر را برای آن ارسال کنید، نوع آن را به عنوان آدرس `address payable` مشخص کنید تا این نیاز قابل مشاهده باشد. همچنین، سعی کنید این تمایز یا تغییر را در اسرع وقت انجام دهید.

عملگرها:

[1] receive

[2] payable fallback function

- `<=`، `<`، `==`، `!=`، `>=` و `>`

هشدار
اگر نوعی را که از اندازه بایت بزرگتری استفاده می‌کند به `address` تبدیل کنید، به عنوان مثال `bytes32` ، سپس به `address` کوتاه می‌شود. برای کاهش ابهام تبدیل ورژن 0.4.24 و بالاتر کامپایلر شما را مجبور به کوتاه کردن صریح در تبدیل می‌کند. به عنوان مثال مقدار 32 بایت `0x1111222233334444555566667777888899999AAAABBBBCCCCDDDDEEEE` `FFFFCCCC` را در نظر بگیرید. می توانید از آدرس `address(uint160(bytes20(b)))` استفاده کنید که نتیجه آن `0x1111222233334444555566667777888899999aAaa` است، یا می‌توانید از آدرس `address(uint160(uint256(b)))` استفاده کنید، که منجر به `0x777788889999AaAAbBbbCcccddDdeeeEfFFfCcCc` می‌شود.

توجه داشته باشید
تمایز بین `address` و `address payable` با ورژن 0.5.0 معرفی شده‌است. همچنین از آن ورژن، قراردادها از نوع آدرس مشتق گرفته نمی‌شوند، اما اگر تابع payable fallback یا receive داشته باشند، هنوز میتوان به صورت صریح به `address` و `address payable` تبدیل شوند.

اعضای آدرس

برای مراجعه سریع به کلیه اعضای آدرس، به اعضای انواع آدرس مراجعه کنید.

- `balance` و `transfer`

می‌توان با استفاده از ویژگی `balance`، بالانس یک آدرس را جستوجو کرد و با استفاده از تابع `transfer` اتر (در واحدهای وی[1]) را به یک آدرس قابل پرداخت[2] ارسال کرد:

[1] wei

[2] payable address

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

اگر بالانس قرارداد فعلی به اندازه کافی بزرگ نباشد یا انتقال اتر توسط حساب دریافت کننده رد شود، تابع `transfer` از کار می‌افتد. تابع `transfer` در صورت شکست برمی‌گردد.

توجه داشته باشید
اگر `x` یک آدرس قرارداد باشد، کد آن (به طور خاص تر: تابع Receive Ether در صورت وجود، یا در غیر این صورت تابع Fallback در صورت وجود) همراه با فراخوانی `transfer` اجرا می‌شود (این ویژگی EVM است و نمی‌توان جلوی آن را گرفت) اگر گاز آن اجرا تمام شود یا به هر صورتی از کار بیفتد، انتقال اتر برگردانده می‌شود و قرارداد جاری با استثنا متوقف می‌شود.

- `send`

Send نقطه مقابل سطح پایین `transfer` است. در صورت عدم اجرا، قرارداد فعلی با استثنا متوقف نخواهد شد، اما `send` مقدار `false` را برمیگرداند.

هشدار
استفاده از `send` خطرات زیادی دارد: اگر فراخوانی پشته عمق 1024 باشد (که همیشه می‌تواند توسط فراخوانی کننده مجبور شود) انتقال شکست میخورد و اگر گاز گیرنده شما تمام شود نیز از کار می‌افتد. بنابراین

برای انجام مطمئن انتقال اتر، همیشه مقدار برگشتی `send`، را با استفاده از `transfer` بررسی کنید یا حتی بهتراست که: از الگویی استفاده کنید که گیرنده پول را برداشت کند.

- `call`، `delegatecall` و `staticcall`

برای برقراری ارتباط با قراردادهایی که به ABI پایبند نیستند، یا برای گرفتن کنترل مستقیم‌تری بر روی رمزگذاری ، توابع `call` ، `delegatecall` و `staticcall` ارائه شده‌اند. همه آنها یک پارامتر `bytes memory` را می‌گیرند و شرایط موفقیت (به عنوان `bool`) و داده های برگشتی (`bytes memory`) را برمی‌گردانند. از توابع `abi.encode`، `abi.encodePacked` ، `abi.encodeWithSelector` و `abi.encodeWithSignature` می‌توان برای رمزگذاری داده‌های ساختار یافته[1] استفاده کرد.

مثال:

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

هشدار
همه این توابع، توابع سطح پایینی هستند و باید با احتیاط استفاده شوند. به طور خاص، هر قرارداد ناشناخته‌ای ممکن است مخرب باشد و در صورت تماس با آن، کنترل آن قرارداد را به شما واگذار می‌کند که می‌تواند به

[1] structured data

نوبه خود به قرارداد شما بازگردد، بنابراین در زمان بازگشت فراخوانی‌ها خود را برای تغییراتی که روی متغیرهای حالت شما اتفاق می‌افتد آماده کنید. روش متداول برای برقراری ارتباط با سایر قراردادها، فراخوانی یک تابع در یک شی قرارداد (`x.f()`) است.

توجه داشته باشید

ورژن های قبلی سالیدیتی به این توابع اجازه می‌دهد آرگومان‌های دلخواه را دریافت کنند و همچنین اولین آرگومان از نوع `bytes4` را به گونه دیگری مدیریت کنند. این موارد در نسخه 0.5.0 حذف شده‌اند.

تنظیم گاز تامین شده با اصلاح کننده[1] `gas` امکان پذیر است:

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

به طور مشابه، مقدار اتر عرضه شده نیز می‌تواند کنترل شود:

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

سرانجام، این اصلاح کننده‌ها می‌توانند ترکیب شوند. ترتیب آنها مهم نیست:

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

1

به روشی مشابه می‌توان از تابع `delegatecall` استفاده کرد: تفاوت در این است که فقط از کد آدرس داده شده استفاده می‌شود، تمام جنبه‌های دیگر (storage ، balance ، ...) از قرارداد فعلی گرفته شده‌اند. هدف از فراخوانی `delegatecall` استفاده از کد کتابخانه است که در قرارداد دیگری ذخیره شده‌است. کاربر باید اطمینان حاصل کند که ساختار storage در هر دو قرارداد برای استفاده از delegatecall مناسب است.

توجه داشته باشید
قبل از homestead، فقط یک نوع محدود به نام `callcode` در دسترس بود که دسترسی به مقادیر اولیه `msg.sender` و `msg.value` را فراهم نمی‌کرد. این تابع در نسخه 0.5.0 حذف شد.

از آنجا که بیزانس `staticcall` نیز می‌تواند مورد استفاده قرار گیرد. این اساساً همان `call` است، اما اگر تابع فراخوانی شده به هر طریقی حالت را تغییر دهد، برمی‌گردد.

هر سه تابع `call` ، `delegatecall` و `staticcall` تابع‌های سطح پایینی هستند و فقط به عنوان آخرین راه حل باید از آنها استفاده شود زیرا باعث از بین رفتن ایمنی بودن نوع سالیدیتی می‌شوند.

گزینه `gas` در هر سه روش موجود است، در حالی که گزینه `value` برای `delegatecall` پشتیبانی نمی‌شود.

توجه داشته باشید

بهتر است بدون توجه به اینکه آیا حالت از آن خوانده می شود یا روی آن نوشته شده است، از تکیه بر مقادیر گاز سخت رمزگذاری شده در کد قرارداد هوشمند خود جلوگیری کنید، زیرا این امر می تواند مشکلات زیادی را به همراه داشته باشد. همچنین، دسترسی به گاز ممکن است در آینده تغییر کند.

توجه داشته باشید
کلیه قراردادها را می‌توان به نوع `address` تبدیل کرد، بنابراین می‌توان بالانس قرارداد فعلی را با استفاده از `address(this).balance` جستوجو کرد.

انواع قرارداد

هر قراردادی نوع خاص خود را مشخص می‌کند. به طور ضمنی می‌توانید قراردادها را به قراردادهایی که از آنها به ارث می‌برند تبدیل کنید. قراردادها را می‌توان به طور صریح به نوع `address` تبدیل و از آنها تغییر داد.

تبدیل صریح به نوع `address payable` فقط از آنجا امکان پذیر است که نوع قرارداد تابع برگشتی قابل دریافت یا پرداخت داشته باشد. تبدیل هنوز با استفاده از `address(x)` انجام می‌شود. اگر نوع قرارداد تابع برگشت پذیر یا قابل پرداخت نباشد، تبدیل به `address payable` را می‌توان با استفاده از `payable(address(x))` انجام داد. در بخش مربوط به نوع آدرس می‌توانید اطلاعات بیشتری کسب کنید.

توجه داشته باشید
قبل از ورژن 0.5.0، قراردادها مستقیماً از نوع آدرس نشأت می‌گرفتند و هیچ تفاوتی بین `address` و `address payable` وجود نداشت.

اگر متغیر محلی را از نوع قرارداد `MyContract c` مشخص کنید، می‌توانید توابع مربوط به آن قرارداد را فراخوانی کنید. مراقب باشید که آن را از جایی اختصاص دهید که همان نوع قرارداد باشد.

شما همچنین می‌توانید قراردادها را فوری (یعنی آنهایی که تازه ایجاد شده‌اند) قرار دهید. جزئیات بیشتر را می‌توانید در بخش "قرارداد از طریق new" پیدا کنید.

نمایش داده‌های یک قرارداد با نوع `address` یکسان است و از این نوع در ABI نیز استفاده می‌شود.

قراردادها از هیچ عملگری پشتیبانی نمی‌کنند.

اعضای انواع قرارداد، توابع خارجی قرارداد شامل هر متغیر حالت است که به عنوان `public` مشخص شده‌است.

برای قرارداد `C` می‌توانید از `type(C)` برای دسترسی به اطلاعات مربوط به قرارداد استفاده کنید.

آرایه‌های بایت با اندازه ثابت[1]

مقدارهای مختلف `bytes1`، `bytes2`، `bytes3`، ...، `bytes32` توالی بایت را از یک تا 32 نگه می‌دارد.

عملگرها:

- مقایسه ها: `<=`، `<`، `==`، `!=`، `>=`، `>` (ارزیابی به `bool`)
- عملگرهای بیت: `&`، `|`، `^` (یا bitwise exclusive)، `~` (bitwise negation)
- عملگرهای شیفت : `<<` (شیفت چپ)، `>>` (شیفت راست)

[1] Fixed-size byte arrays

دسترسی به Index: اگر `x` از نوع `bytesI` باشد، سپس `x[k]` برای `0 <= k < I` بایت `k` را برمی‌گردانم (فقط برای خواندن).

عملگر شیفت با نوع عدد صحیح بدون علامت به عنوان عملوند راست کار می‌کند (اما نوع عملوند سمت چپ را برمی‌گرداند)، که تعداد بیت های شیفت را نشان می‌دهد. جابجایی با نوع با علامت[1]، خطای کامپایل ایجاد می‌کند.

اعضا:

- `.length` طول ثابت آرایه بایت را ارائه می‌دهد (فقط برای خواندن).

توجه داشته باشید
نوع `byte[]` ، آرایه‌ای از بایت است. اما به دلیل قوانین padding ، 31 بایت فضا را برای هر عنصر هدر می‌دهد (به جز در storage). بهتر است به جای آن از نوع `bytes` استفاده کنید.

توجه داشته باشید
قبل از ورژن 0.8.0 ، `bytes` یک نام مستعار برای `bytes1` بود.

آرایه بایت با اندازه پویا[2]

`bytes`:

آرایه بایت در اندازه پویا، به آرایه‌ها مراجعه کنید. از نوع مقدار[3] نیست!

[1] signed type

[2] Dynamically-sized byte array

[3] value-type

`string`:

رشته‌ای با کد UTF-8 به صورت پویا، به آرایه‌ها مراجعه کنید. از نوع مقدار نیست!

لیترال‌های Address

لیترال‌های هگزادسیمال که از تست checksum آدرس استفاده می‌کنند، به عنوان مثال `0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF` از نوع `address` هستند.

لیترال‌های هگزادسیمال که دارای طول 39 تا 41 رقم هستند و از تست checksum عبور نمی‌کنند، خطایی ایجاد می‌کنند. برای حذف خطا می‌توانید (برای انواع عدد صحیح) یا (برای انواع bytesNN) صفرها را ضمیمه کنید.

توجه داشته باشید
قالب checksum آدرس مختلط در EIP-55 تعریف شده است.

لیترال‌های عدد گویا و صحیح

لیترال‌های عدد صحیح[1] از توالی اعداد در محدوده 0-9 تشکیل می شوند. آنها به عنوان دیسیمال تفسیر می شوند. به عنوان مثال `69` به معنای شصت و نه است. لیترال‌های Octal در سالیدیتی وجود ندارند و صفرهای قبل از عدد نامعتبر هستند.

لیترال‌های کسری دیسیمال توسط یک `.` با حداقل یک عدد در یک طرف تشکیل می‌شوند. مثال‌ها شامل `1.` ، `.1` و `1.3` است.

[1] Integer literals

نشانه علمی نیز پشتیبانی می‌شود، جایی که پایه می‌تواند کسر داشته باشد و توان تواند. برای مثال از جمله `2e10` ، `-2e10` ، `2e-10` ، `2.5e1` .

زیرخط می‌تواند برای جدا کردن رقم از لیترال‌های عددی برای کمک به خوانایی استفاده شود. به عنوان مثال، دسیمال `123_000` ، هگزادسیمال `0x2eff_abde` ، نماد علمی دسیمال `1_2e345_678` همه معتبر هستند. زیرخط تنها بین دو رقم مجاز است و تنها یک زیرخط متوالی مجاز است. هیچ معنایی سمنتیک اضافی به لیترال عددی حاوی زیرخط اضافه نشده است.

عبارات لیترال‌های عددی دقت دلخواه را حفظ می‌کنند تا زمانی که به یک نوع غیرلیترالی تبدیل شوند (به عنوان مثال استفاده از آنها همراه با یکدیگر با یک عبارت غیرلیترالی یا با تبدیل صریح). این بدان معناست که محاسبات سرریز نمی‌شود و تقسیمات در عبارات لیترال عددی کوتاه نمی‌شوند.

به عنوان مثال، `(2**800 + 1) - 2**800` منجر به ثابت `1` (از نوع `uint8`) می‌شود گرچند نتایج میانی حتی اندازه کلمه ماشین را فیت نمی‌کند. علاوه بر این، `.5 * 8` منجر به عدد صحیح `4` (گرچند در بین آنها غیر عدد صحیح استفاده می‌شود).

هر عملگری که می‌تواند به عدد صحیح اعمال شود، تا زمانی که عملوندها عدد صحیح باشند می تواند به لیترال‌های عددی نیز اعمال شود. اگر هر یکی از این دو کسری باشند، عملیات بیت امکان پذیر نیست و نیز به توان رساندن اگر توان کسری باشد (زیرا ممکن است منجر به یک عدد غیر گویا شوند) امکان پذیر نیست.

تعویض و به توان رساندن با اعداد لیترال بطوریکه سمت چپ (یا پایه) عملوند و نوع عدد صحیح در سمت راست به عنوان عملوند (توان) همیشه در `uint256` (برای لیترال‌های غیر منفی) یا `int256` (برای لیترال‌های منفی)، بدون توجه به نوع سمت راست عملوند (توان)، عمل می‌کند.

هشدار

تقسیم بر روی لیترال‌های عدد صحیح برای کوتاه کردن در سالیدیتی نسخه‌های قبلتر از نسخه 0.4.0 استفاده میشد، اما اکنون به یک عدد گویا تبدیل می شود، برای مثال `5 / 2` برابر با `2` نیست بلکه برابر با `2.5` می‌باشد.

توجه داشته باشید

سالیدیتی برای هر عدد گویا یک نوع لیترال عددی دارد. لیترال های عدد صحیح و لیترال‌های عدد گویا به انواع لیترال‌های عدد تعلق دارند. علاوه بر این، تمام عبارات لیترال‌های عددی (یعنی عباراتی که فقط شامل لیترال‌های عددی و عملگرها هستند) به انواع لیترال‌های عددی تعلق دارند. بنابراین عبارات لیترال عددی `1 + 2` و `2 + 1` هر دو متعلق به همان نوع لیترال عددی برای عدد گویا سه هستند.

توجه داشته باشید

عبارات لیترال عددی به محض استفاده با عبارات غیر لیترال به نوع غیر لیترال تبدیل می‌شوند. با نادیده گرفتن انواع، مقدار عبارتی که به `b` در زیر اختصاص داده شده به عنوان عدد صحیح ارزیابی می‌شود. از آنجا که `a` از نوع `uint128` است، عبارت `2.5 + a` باید نوع مناسبی داشته باشد. از آنجا که نوع متداولی برای نوع `2.5` و `uint128` وجود ندارد، کامپایلر سالیدیتی این کد را قبول نمی‌کند.

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

لیترال‌های string و انواع

لیترال‌های رشته‌ای با دو نقل قول یا تک نقل قولی نوشته می‌شوند (`"foo"` یا `'bar'`)، و همچنین می‌توانند به چند قسمت متوالی تقسیم شوند (`"foo" "bar"` معادل `"foobar"` است) که می‌تواند هنگام کار با رشته‌های طولانی مفید باشد. آنها به مفهوم صفر انتهایی در C نیست. `"foo"` نشانگر سه بایت می‌باشد، نه چهار بایت.

همانند لیترال‌های عدد صحیح، نوع آنها نیز می‌تواند متفاوت باشد، اما در صورت متناسب بودن آنها به بایت‌های `bytes32`, ..., `bytes1` تبدیل می‌شوند، اگر متناسب باشند، به `bytes` و `string` تبدیل می‌شوند.

به عنوان مثال، با `bytes32 samevar = "stringliteral"` لیترال رشته‌ای وقتی به نوع `bytes32` اختصاص یابد به معنای بایت خام[1] تفسیر می‌شود.

لیترال‌های رشته‌ای فقط می‌توانند حاوی کارکترهای ASCII قابل چاپ باشند، این به معنای کارکترهای شامل و بین 0x1F .. 0x7E می‌باشند.

علاوه بر این، لیترال‌های رشته‌ای از کارکترهای escape زیر نیز پشتیبانی می‌کنند:

- `\<newline>` (escapes an actual newline)
- `\\` (backslash)
- `\'` (single quote)
- `\"` (double quote)
- `\b` (backspace)

[1] raw byte

- `\f` (form feed)
- `\n` (newline)
- `\r` (carriage return)
- `\t` (tab)
- `\v` (vertical tab)
- `\xNN` (hex escape, see below)
- `\uNNNN` (unicode escape, see below)

`\xNN` یک مقدار hex می‌گیرد و بایت مناسب را وارد می‌کند، در حالی که `\uNNNN` یک کد رمز Unicode را می‌گیرد و یک توالی UTF-8 را وارد می‌کند.

رشته در مثال زیر دارای طول ده بایت است. این کار با یک بایت خط جدید و به دنبال آن یک دو نقل قول، یک تک نقل قول یک کاراکتر بک اسلش و سپس (بدون جدا کننده) توالی کاراکتر abcdef شروع می‌شود.

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

هر خاتمه دهنده خط Unicode که یک خط جدید نباشد (به عنوان مثال LF، VF، FF، CR، NEL، LS، PS) برای خاتمه لیترال رشته در نظر گرفته می‌شود. Newline فقط در صورتی لیترال رشته را خاتمه می‌دهد که قبل از آن `\` وجود نداشته باشد.

لیترال‌های Unicode

در حالی که لیترال‌های رشته‌ای منظم فقط می‌توانند حاوی ASCII باشند، لیترال‌های Unicode - با پیشوند کلمه کلیدی `unicode` – می‌توانند حاوی هر توالی معتبر UTF-8 باشند. آنها همچنین از همان توالی‌های escape به عنوان لیترال‌های رشته منظم پشتیبانی می‌کنند.

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

لیترال‌های هگزادسیمال

لیترال‌های هگزادسیمال با پیشوند کلمه کلیدی `hex` هستند،که با دو نقل قول یا تک نقل قولی محصور شده‌اند (`hex"001122FF"`، `hex'0011_22_FF'`). محتوای آنها باید ارقام هگزادسیمال باشد که به صورت اختیاری می‌تواند از یک زیر خط به عنوان جدا کننده بین مرز بایت استفاده کند. مقدار لیترال، نمایش دودویی توالی هگزادسیمال خواهد بود.

لیترال‌های مالتی هگزادسیمال جدا شده توسط فضای خالی به یک لیترال متصل می‌شوند: `hex"00112233" hex"44556677"` معادل با `hex"0011223344556677"` است.

لیترال‌های هگزادسیمال مانند لیترال‌های رشته‌ای رفتار می‌کنند و محدودیت‌های تبدیل پذیری یکسانی دارند.

Enums

Enums یکی از راه‌های ایجاد یک نوع تعریف شده توسط کاربر در سالیدیتی می‌باشد. آنها به طور صریح قابل تبدیل به انواع مختلف عدد صحیح هستند اما تبدیل ضمنی مجاز نیست. تبدیل صریح از عدد صحیح در زمان اجرا بررسی می‌کند که مقدار در محدوده enum باشد و در غیر این صورت باعث ایجاد خطای Panic می‌شود. Enums حداقل به یک عضو نیاز دارد و مقدار پیش فرض آن هنگام اعلام اولین عضو است. Enums نمی‌تواند بیش از 256 عضو داشته باشد.

نمایش داده‌ها همانند enum ها در C است: گزینه‌ها با مقادیر صحیح بدون علامت بعدی ارسال می‌شوند که از `0` شروع می شوند.

```
// SPDX-License-Identifier: GPL-3.0
```

```
pragma solidity >=0.4.16 <0.9.0;

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

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

    // Since enum types are not part of the ABI, the
signature of "getChoice"
    // will automatically be changed to "getChoice()
returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns
(ActionChoices) {
        return choice;
    }

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

توجه داشته باشید
Enums همچنین می‌تواند خارج از تعاریف قرارداد یا کتابخانه، در سطح فایل مشخص شوند.

انواع توابع

انواع توابع انواعی از توابع هستند. متغیرهای نوع تابع را می‌توان از توابع اختصاص داد و پارامترهای تابع نوع تابع را می‌توان برای انتقال توابع به توابع برگشتی و از فراخوانی‌های تابع استفاده کرد. انواع توابع به دو صورت هستند– توابع داخلی و خارجی:

توابع داخلی را فقط می‌توان در داخل قرارداد فعلی فراخوانی کرد (به طور خاص، در داخل واحد کد فعلی، که شامل توابع کتابخانه داخلی و توابع وراثتی نیز می‌شود) زیرا نمی‌توانند خارج از متن قرارداد فعلی اجرا شوند. فراخوانی یک تابع داخلی با پرش به برچسب ورودی آن انجام می‌شود، دقیقاً مانند هنگام فراخوانی داخلی توابع قرارداد فعلی.

توابع خارجی شامل یک آدرس و یک امضای تابع می‌باشند و می‌توان آنها را از طریق فراخوانی‌های تابع خارجی منتقل کرد و از آنها بازگرداند.

انواع توابع به شرح زیر ذکر شده‌است:

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

در مقابل انواع پارامترها، انواع بازگشت نمی‌توانند خالی باشند - اگر نوع عملکرد نباید چیزی را برگرداند، کل قسمت `returns (<return types>)` باید حذف شود. به طور پیش فرض، انواع عملکردها داخلی هستند، بنابراین می‌توان کلمه کلیدی داخلی را حذف کرد. توجه داشته باشید که این فقط در انواع توابع اعمال می‌شود. قابلیت مشاهده به طور صریح برای توابع تعریف شده در قراردادها مشخص می‌شود، آنها پیش فرض ندارند.

به طور پیش فرض، انواع توابع داخلی هستند، بنابراین می‌توان کلمه کلیدی `internal` را حذف کرد. توجه داشته باشید که این فقط در انواع توابع اعمال می‌شود. قابلیت مشاهده به طور صریح برای توابع تعریف شده در قراردادها مشخص می‌شود، آنها پیش فرض ندارند.

تبدیل‌ها:

یک تابع نوع `A` به طور ضمنی قابل تبدیل به یک تابع نوع `B` است اگر و فقط اگر انواع پارامترهای آنها یکسان باشد، انواع بازگشت آنها یکسان، ویژگی `internal/external` آنها یکسان باشد و تغییرپذیری حالت `A` محدود کننده‌تر از تغییر پذیری حالت `B` . به خصوص:

- توابع `pure` را می‌توان به `view` و توابع `non-payable` تبدیل کرد
- توابع `view` را می‌توان به توابع `non-payable` پرداخت تبدیل کرد
- توابع `payable` را می‌توان به توابع `non-payable` پرداخت تبدیل کرد

هیچ تبدیل دیگری بین انواع توابع امکان پذیر نیست.

قانون مربوط به `payable` و `non-payable` ممکن است کمی گیج کننده باشد، اما در اصل، اگر تابعی `payable` باشد، این بدان معناست که پرداخت صفر اتر را نیز می‌پذیرد، بنابراین `non-payable` نیز می‌باشد. از طرف دیگر، یک تابع `non-payable` اتر ارسال شده به آن را رد می‌کند، بنابراین توابع `non-payable` نمی‌توانند به توابع `payable` تبدیل شوند.

اگر یک متغیر از نوع تابع مقداردهی اولیه نشده باشد، فراخوانی آن منجر به خطای Panic می‌شود. اگر پس از استفاده از `delete` تابع آن را فراخوانی کنید، همین اتفاق می‌افتد.

اگر از انواع توابع external خارج از زمینه سالیدیتی استفاده شود، با آنها به عنوان نوع `function` رفتار می‌شود، که آدرس و به دنبال آن شناسه تابع را با هم در یک تک نوع `bytes24` رمزگذاری می‌کند.

توجه داشته باشید که توابع عمومی قرارداد جاری می‌توانند هم به عنوان تابع داخلی و هم به عنوان تابع خارجی استفاده شود. برای استفاده از `f` به عنوان یک تابع داخلی ، فقط از `f` استفاده کنید، اگر می‌خواهید از فرم خارجی آن استفاده کنید، از `this.f` استفاده کنید.

یک تابع از یک نوع داخلی را می‌توان به یک متغیر از یک نوع تابع داخلی بدون در نظر گرفتن مکان تعریف شده اختصاص داد. این شامل توابع خصوصی، داخلی و عمومی قراردادها و کتابخانه‌ها و همچنین توابع رایگان است. از طرف دیگر، انواع توابع خارجی فقط با توابع قرارداد عمومی و خارجی سازگار هستند. کتابخانه‌ها از این مستثنی هستند چونکه به یک `delegatecall` نیاز دارند و از یک کنوانسیون[1] مختلف ABI برای انتخابگرهای خود استفاده می‌کنند. توابع اعلام شده در رابط‌ها تعریفی ندارند، بنابراین اشاره به آنها نیز معنی ندارد.

اعضا:

توابع خارجی (یا عمومی) اعضای زیر را دارند:

- `.address` آدرس قرارداد تابع را برمی‌گرداند.
- `.selector` انتخابگر تابع ABI را برمی‌گرداند.

توجه داشته باشید

[1] convention

توابع خارجی (یا عمومی) برای داشتن اعضای اضافی `.gas(uint)` و `.value(uint)` استفاده می‌شود. اینها در سالیدیتی نسخه 0.6.2 منسوخ شده و در سالیدیتی نسخه 0.7.0 حذف شدند. در عوض از `{gas: ...}` و `{value: ...}` برای تعیین مقدار گاز یا مقدار wei ارسال شده به یک تابع استفاده کنید. برای اطلاعات بیشتر به فراخوانی تابع خارجی مراجعه کنید.

مثالی که نحوه استفاده از اعضا را نشان می‌دهد:

```
// 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}();
    }
}
```

مثالی که نحوه استفاده از انواع توابع داخلی را نشان می‌دهد:

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

library ArrayUtils {
    // internal functions can be used in internal library
functions because
```

```
    // they will be part of the same code context
    function map(uint[] memory self, function (uint) pure
returns (uint) f)
        internal
        pure
        returns (uint[] memory r)
    {
        r = new uint[](self.length);
        for (uint i = 0; i < self.length; i++) {
            r[i] = f(self[i]);
        }
    }

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

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


contract Pyramid {
    using ArrayUtils for *;
```

```
    function pyramid(uint l) public pure returns (uint) {
        return ArrayUtils.range(l).map(square).reduce(sum);
    }

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

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

مثال دیگری که از انواع توابع خارجی استفاده می‌کند:

```
// 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 {
        // Here goes the check that the reply comes from a
trusted source
        requests[requestID].callback(response);
    }
}


contract OracleUser {
    Oracle constant private ORACLE_CONST =
Oracle(address(0x00000000219ab540356cBB839Cbe05303d7705Fa))
; // known contract
    uint private exchangeRate;

    function buySomething() public {
        ORACLE_CONST.query("USD", this.oracleResponse);
    }

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

توجه داشته باشید
توابع Lambda یا inline برنامه ریزی شده‌اند اما هنوز پشتیبانی نمی‌شوند.

انواع مرجع[1]

مقادیر نوع مرجع را می‌توان از طریق چندین نام مختلف اصلاح کرد. هر زمان که متغیر از نوع مقدار[2] استفاده شود، در جایی که یک کپی مستقل دریافت می‌کنید نوع مرجع را با انواع مقدار مقایسه کنید. به همین دلیل، انواع مرجع باید با دقت بیشتری از انواع مقادیر رسیدگی شوند. در حال حاضر، انواع مرجع شامل structها، آرایه‌ها و mappingها است. اگر از یک نوع مرجع استفاده می‌کنید، همیشه باید صریحاً منطقه داده‌ای[3] را که نوع در آن ذخیره شده‌است فراهم کنید: `memory` (طول عمر آن فقط به یک فراخوانی کنند تابع خارجی محدود می‌شود)، `storage` (مکانی که متغیرهای حالت در آن ذخیره می‌شوند، جایی که طول عمر آنها به طول عمر قرارداد محدود می‌شود) یا `calldata` (مکان داده ویژه‌ای که شامل آرگومان‌های تابع است).

یک انتساب یا تبدیل نوع که مکان داده را تغییر می‌دهد، همیشه موجب یک عملیات کپی خودکار خواهد شد، در حالی که انتساب در داخل همان مکان داده فقط در برخی موارد برای انواع storage کپی می‌شوند.

مکان داده[4]

هر نوع مرجع حاوی یادداشت اضافی است، "data location"، در مورد مکانی که ذخیره می‌شود. سه مکان داده وجود دارد: `memory` ، `storage` و `calldata` .

Calldata یک منطقه غیرقابل تغییر و غیرقابل ماندگاری است که آرگومان‌های تابع در آن ذخیره می‌شود و بیشتر مانند مِمُوری رفتار می‌کند. برای پارامترهای توابع خارجی لازم است اما می‌تواند برای سایر متغیرها نیز استفاده شود.

توجه داشته باشید

[1] Reference Types
[2] value type
[3] data area
[4] Data location

قبل از نسخه 0.5.0 ، مکان داده را می‌توان حذف کرد، و بسته به نوع متغیر، نوع تابع و غیره به مکان‌های مختلف پیش فرض می‌رود ، اما اکنون همه انواع پیچیده باید یک مکان داده مشخص داشته باشند.

توجه داشته باشید
اگر می‌توانید، سعی کنید از `calldata` به عنوان مکان داده استفاده کنید زیرا از کپی جلوگیری می‌کند و همچنین مطمئن می‌شوید که داده‌ها قابل اصلاح نیستند. آرایه‌ها و structهای دارای مکان داده `calldata` نیز می‌توانند از توابع برگردانده شوند، اما اختصاص چنین نوع‌هایی امکان پذیر نیست.

مکان داده و رفتار انتساب[1]

مکان داده[2] نه تنها برای ماندگاری داده‌ها بلکه برای معنای انتساب‌ها نیز مهم هستند:

- انتساب‌ها بین `storage` و `memory` (یا از `calldata`) همیشه یک کپی مستقل ایجاد می‌کنند.
- انتساب‌ها از `memory` به `memory` فقط مراجع را ایجاد می‌کند. این بدان معناست که تغییرات در یک متغیر مِمُوری در سایر متغیرهای مِمُوری که به داده‌های مشابه ارجاع می‌کنند نیز قابل مشاهده‌است.
- انتساب‌ها از `storage` به یک متغیر `storage` محلی نیز فقط یک مرجع اختصاص می‌دهند.

[1] assignment behaviour

[2] Data location

- سایر انتسابات به `storage` همیشه کپی می‌شوند. نمونه‌هایی برای این مورد، انتساب به متغیرهای حالت یا اعضای متغیرهای محلی از نوع `storage struct` می‌باشند، حتی اگر متغیر محلی فقط یک مرجع باشد.

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

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array
to storage
        uint[] storage y = x; // works, assigns a pointer,
data location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies
y
        // The following does not work; it would need to
create a new temporary /
        // unnamed array in storage, but storage is
"statically" allocated:
        // y = memoryArray;
        // This does not work either, since it would
"reset" the pointer, but there
        // is no sensible location it could point to.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent,
temporary copy in memory
```

```
    }

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

آرایه‌ها

آرایه‌ها می‌توانند اندازه ثابت زمان کامپایل داشته باشند، یا می‌توانند اندازه پویا داشته باشند.

نوع آرایه‌ای با اندازه ثابت `k` و نوع عنصر `T` به صورت `T[k]` و آرایه‌ای با اندازه پویا به صورت `T[k]` نوشته می‌شود.

به عنوان مثال، آرایه‌ای از 5 آرایه دینامیکی `uint` به صورت `uint[][5]` نوشته می‌شود. علامت گذاری در مقایسه با برخی از زبان‌های دیگر معکوس می‌شود. در سالیدیتی ، `X[3]` همیشه یک آرایه است که شامل سه عنصر از نوع `X` است، حتی اگر `X` خودش یک آرایه باشد. این مورد در زبان‌های دیگر مانند C وجود ندارد.

شاخص‌ها[1] مبتنی بر صفر هستند و دسترسی در خلاف جهت اعلامیه[2] است.

به عنوان مثال، اگر یک متغیر `uint[][5] memory x` داشته باشید، با استفاده از `x[2][1]` به `uint` دوم در آرایه پویای سوم دسترسی پیدا می‌کنید و برای دسترسی به آرایه پویای سوم، از `x[2]` استفاده کنید. باز هم اگر یک آرایه `T[5] a` برای نوع `T` دارید که می‌تواند یک آرایه نیز باشد، `a[2]` همیشه نوع `T` را دارد.

[1] Indices

[2] opposite direction of the declaration

عناصر آرایه می‌توانند از هر نوع شامل mapping یا struct باشند. محدودیت‌های کلی برای انواع اعمال می‌شود، به این دلیل که mappingها فقط در محل داده storage می‌توانند ذخیره شوند و توابع قابل مشاهده به صورت عمومی، نیاز به پارامترهایی دارند که از نوع ABI باشند.

می‌توان آرایه‌های متغیر حالت را به صورت public علامت گذاری کرد و از سالیدیتی برای ایجاد یک getter استفاده کرد. شاخص عددی به یک پارامتر مورد نیاز برای getter تبدیل می‌شود.

دستیابی به آرایه‌ای که از انتهای آن گذشته است، ادعای ناموفقی را ایجاد می‌کند. از روش های .push() و .push(value) می‌توان برای افزودن یک عنصر جدید در انتهای آرایه استفاده کرد، جایی که .push() یک عنصر مقداردهی شده صفر را اضافه می‌کند و مرجعی را به آن برمی‌گرداند.

bytes و string به صورت آرایه‌ها

متغیرهای نوع bytes و string آرایه‌های خاصی هستند. bytes مانند byte[] است، اما در calldata و مِمُوری کاملاً بسته بندی شده است. string برابر با bytes است اما اجازه دسترسی به طول[1] یا index را نمی‌دهد.

سالیدیتی توابع دستکاری[2] string ندارد، اما کتابخانه‌های string طرف سوم وجود دارد. همچنین می‌توانید دو string را توسط keccak256-hash آنها با استفاده از keccak256(abi.encodePacked(s1)) == keccak256(abi.encodePacked(s2)) مقایسه کنید و دو رشته را با استفاده از bytes.concat(bytes(s1), bytes(s2)) بهم پیوست دهید.

[1] length

[2] manipulation functions

از آنجا که `byte[]` سی و یک padding bytes بین عناصر اضافه می‌کند، شما باید از `bytes` بیش از `byte[]` استفاده کنید زیرا ارزان‌تر است. به عنوان یک قاعده کلی، برای داده‌های `bytes` خام با طول دلخواه از `bytes` و برای داده‌های `string` با طول دلخواه (UTF-8) از `string` استفاده کنید. اگر می‌توانید طول را به تعداد مشخصی از بایت محدود کنید، همیشه از یکی از انواع مقدار `bytes1` تا `bytes32` استفاده کنید زیرا بسیار ارزان‌تر هستند.

توجه داشته باشید
اگر می‌خواهید به نمایش بایت[1] یک رشته‌ی `s` دسترسی پیدا کنید، از `bytes(s).length` یا `bytes(s)[7] = 'x';` استفاده کنید. بخاطر داشته باشید که شما به بایت‌های سطح پایین، پیش نمایش UTF-8 و نه به کارکترهای جداگانه دسترسی پیدا می‌کنید.

تابع `bytes.concat`

با استفاده از `bytes.concat` می‌توانید تعدادی متغیر از `bytes` یا `bytes1 ... bytes32` را بهم پیوند دهید. این تابع یک تک آرایه `bytes memory` را برمی‌گرداند که شامل محتویات آرگومان‌ها بدون padding است. اگر می‌خواهید از پارامترهای رشته‌ای یا انواع دیگر استفاده کنید، ابتدا باید آنها را به `bytes` یا `bytes1`/.../`bytes32` تبدیل کنید.

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

contract C {
```

[1] byte-representation

```
    bytes s = "Storage";
    function f(bytes calldata c, string memory m, bytes16
b) public view {
        bytes memory a = bytes.concat(s, c, c[:2],
"Literal", bytes(m), b);
        assert((s.length + c.length + 2 + 7 +
bytes(m).length + 16) == a.length);
    }
}
```

اگر بدون آرگومان `bytes.concat` فراخوانی کنید، آرایه‌ای خالی از `bytes` را برمی‌گرداند.

تخصیص آرایه های مِمُوری

آرایه‌های مِمُوری با طول پویا را می‌توان با استفاده از عملگر `new` ایجاد کرد. در مقایسه با آرایه‌های storage، تغییر اندازه آرایه‌های مِمُوری امکان پذیر نیست (به عنوان مثال توابع عضو `.push` در دسترس نیستند). یا باید اندازه مورد نیاز را از قبل محاسبه کنید یا یک آرایه مِمُوری جدید ایجاد کنید و هر عنصر را کپی کنید.

مثل همهِ متغیرها در سالیدیتی، عناصر آرایه‌های تازه تخصیص یافته همیشه با مقدار پیش فرض مقداردهی می‌شوند.

```
// 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;
    }
```

```
}
```

آرایه‌های لیترال

آرایه لیترال لیستی جدا شده با کاما از یک یا چند عبارت است که در بِراکت مربعی محصور شده است (`[...]`). به عنوان مثال `[1, a, f(3)]` . نوع آرایه به صورت زیر تعیین می‌شود:

همیشه یک آرایه مِمُوری با اندازه ایستا است که طول آن تعداد عبارات است.

نوع پایه‌ی آرایه، نوع اولین عبارت در لیست است به طوری که می‌توان بقیه عبارات را به طور ضمنی به آن تبدیل کرد. اگر این امکان وجود نداشته باشد خطای نوع است.

کافی نیست نوعی وجود داشته باشد که همه عناصر بتوانند به آن تبدیل شوند. یکی از عناصر باید از آن نوع باشد.

در مثال زیر، نوع `[1, 2, 3]` ، `uint8[3] memory` می‌باشد، زیرا نوع هر یک از این ثابت‌ها `uint8` است. اگر می‌خواهید نتیجه از نوع `uint8[3] memory` باشد، باید اولین عنصر را به `uint8` تبدیل کنید.

```
// 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 {
        // ...
    }
```

```
}
```

آرایه لیترال `[1, -1]` نامعتبر است زیرا نوع عبارت اول `int8` است در حالی که نوع دوم `int8` است و نمی‌توان آنها را به طور ضمنی به یکدیگر تبدیل کرد. برای استفاده از آن، می‌توانید از `[int8(1), -1]` استفاده کنید.

از آنجا که آرایه‌های مِمُوری با اندازه ثابت از انواع مختلف قابل تبدیل به یکدیگر نیستند (حتی اگر انواع پایه بتوانند)، اگر می‌خواهید از لیترال‌های دو بعدی استفاده کنید، باید یک نوع پایه مشترک را به طور صریح مشخص کنید:

```
// 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]];
        // The following does not work, because some of the
inner arrays are not of the right type.
        // uint[2][4] memory x = [[0x1, 1], [0xffffff, 2],
[0xff, 3], [0xffff, 4]];
        return x;
    }
}
```

آرایه های مِمُوری با اندازه ثابت را نمی‌توان به آرایه‌های مِمُوری با اندازه پویا اختصاص داد، یعنی موارد زیر امکان پذیر نیست:

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

// This will not compile.
contract C {
    function f() public {
        // The next line creates a type error because
uint[3] memory
        // cannot be converted to uint[] memory.
        uint[] memory x = [uint(1), 3, 4];
    }
}
```

در آینده برنامه ریزی شده‌است که سالیدیتی این محدودیت را برطرف کند، اما به دلیل نحوهِ انتقال آرایه‌ها در ABI، مشکلاتی ایجاد می‌شود.

اگر می‌خواهید آرایه‌هایی با اندازه پویا را شروع کنید، باید عناصر جداگانه را اختصاص دهید:

```
// 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;
    }
}
```

اعضای آرایه

Length:

آرایه‌ها دارای یک عضو `length` هستند که شامل تعداد عناصر آنها است. طول آرایه‌های مِمُوری پس از ایجاد ثابت است (اما پویا، یعنی می‌تواند به پارامترها در زمان اجرا بستگی داشته باشد).

push ():

آرایه‌های storage و `bytes` پویا (نه `string`) دارای یک عضو تابع به نام `push()` هستند که می‌توانید از آن برای افزودن یک عنصر مقداردهی شده صفر در انتهای آرایه استفاده کنید. یک ارجاع به عنصر را برمی‌گرداند، بنابراین می‌توان از آن مانند `x.push().t = 2` یا `x.push() = b` استفاده کرد.

push(x):

آرایه‌های storage و `bytes` پویا (نه `string`) دارای یک عضو تابع به نام `push(x)` هستند که می‌توانید از آن برای افزودن یک عنصر مشخص در انتهای آرایه استفاده کنید. تابع هیچ چیزی بر نمی‌گرداند.

pop:

آرایه‌های storage و `bytes` پویا (نه `string`) دارای یک عضو تابع به نام `pop` هستند که می‌توانید برای حذف یک عنصر از انتهای آرایه استفاده کنید. همچنین به طور ضمنی delete را روی عنصر حذف شده فراخوانی می‌کند.

توجه داشته باشید
افزایش طول یک آرایه storage با فراخوانی `push()` هزینه گاز ثابت را دارد زیرا مقداردهی اولیه storage صفر می‌باشد، در حالی که کاهش طول با فراخوانی `pop()` هزینه‌ای دارد که به "اندازه" عنصر حذف شده بستگی دارد. اگر آن عنصر آرایه‌ای باشد، می‌تواند بسیار پرهزینه باشد، زیرا شامل پاک کردن صریح عناصر حذف شده مشابه با فراخوانی delete روی آنها است.

توجه داشته باشید
برای استفاده از آرایه های توابع خارجی (به جای عملکرد public) ، باید ABI coder v2 را فعال کنید.

توجه داشته باشید
در نسخه‌های EVM قبل از Byzantium ، دسترسی به آرایه‌های پویا از برگشتیِ توابعِ فراخوانی امکان پذیر نبود. اگر توابعی را فراخوانی می‌کنید که آرایه‌های پویا را برمی‌گردانند، حتماً از EVMی استفاده کنید که روی حالت Byzantium تنظیم شده‌است.

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

contract ArrayContract {
    uint[2**20] m_aLotOfIntegers;
    // Note that the following is not a pair of dynamic
arrays but a
    // dynamic array of pairs (i.e. of fixed size arrays of
length two).
    // Because of that, T[] is always a dynamic array of T,
even if T
    // itself is an array.
    // Data location for all state variables is storage.
    bool[2][] m_pairsOfFlags;
```

```
    // newPairs is stored in memory - the only possibility
    // for public contract function arguments
    function setAllFlagPairs(bool[2][] memory newPairs)
public {
        // assignment to a storage array performs a copy of
``newPairs`` and
        // replaces the complete array ``m_pairsOfFlags``.
        m_pairsOfFlags = newPairs;
    }

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

    function f(uint[] memory c) public {
        // stores a reference to ``s`` in ``g``
        StructType storage g = s;
        // also changes ``s.moreInfo``.
        g.moreInfo = 2;
        // assigns a copy because ``g.contents``
        // is not a local variable, but a member of
        // a local variable.
        g.contents = c;
    }

    function setFlagPair(uint index, bool flagA, bool
flagB) public {
        // access to a non-existing index will throw an
exception
        m_pairsOfFlags[index][0] = flagA;
        m_pairsOfFlags[index][1] = flagB;
    }

    function changeFlagArraySize(uint newSize) public {
        // using push and pop is the only way to change the
        // length of an array
        if (newSize < m_pairsOfFlags.length) {
```

```
            while (m_pairsOfFlags.length > newSize)
                m_pairsOfFlags.pop();
        } else if (newSize > m_pairsOfFlags.length) {
            while (m_pairsOfFlags.length < newSize)
                m_pairsOfFlags.push();
        }
    }

    function clear() public {
        // these clear the arrays completely
        delete m_pairsOfFlags;
        delete m_aLotOfIntegers;
        // identical effect here
        m_pairsOfFlags = new bool[2][](0);
    }

    bytes m_byteData;

    function byteArrays(bytes memory data) public {
        // byte arrays ("bytes") are different as they are
stored without padding,
        // but can be treated identical to "uint8[]"
        m_byteData = data;
        for (uint i = 0; i < 7; i++)
            m_byteData.push();
        m_byteData[3] = 0x08;
        delete m_byteData[2];
    }

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

    function createMemoryArray(uint size) public pure
returns (bytes memory) {
        // Dynamic memory arrays are created using `new`:
```

```
        uint[2][] memory arrayOfPairs = new
uint[2][](size);

        // Inline arrays are always statically-sized and if
you only
        // use literals, you have to provide at least one
type.
        arrayOfPairs[0] = [uint(1), 2];

        // Create a dynamic byte array:
        bytes memory b = new bytes(200);
        for (uint i = 0; i < b.length; i++)
            b[i] = bytes1(uint8(i));
        return b;
    }
}
```

برش‌های آرایه[1]

برش‌های آرایه نمایی از قسمت پیوسته آرایه است. آنها به صورت `x[end - 1]` نوشته می‌شوند، جایی که `start` و `end` عباراتی هستند که منجر به نوع uint256 می‌شوند (یا به طور ضمنی قابل تبدیل به آن هستند). اولین عنصر برش `x[start]` و آخرین عنصر `x[end - 1]` می‌باشد.

اگر `start` از `end` بیشتر باشد یا اگر `end` از طول آرایه بیشتر باشد، یک استثنا ایجاد می‌شود.

`start` و `end` هر دو اختیاری هستند: `start` به طور پیشفرض `0` و `end` به طور پیش فرض به طول آرایه می‌باشد.

[1] Array Slices

برش‌های آرایه هیچ عضوی ندارند. آنها به طور ضمنی قابل تبدیل به آرایه‌هایی از نوع اصلی و دسترسی به index را پشتیبانی می‌کنند. دسترسی index در آرایه اصلی قطعی نیست، اما وابسته به شروع برش است.

برش‌های آرایه دارای نام نوع نیستند، به این معنی که هیچ متغیری نمی‌تواند برش‌های آرایه‌ای را به عنوان نوع داشته باشد، آنها فقط در عبارات میانی وجود دارند.

توجه داشته باشید
از هم اکنون، برش‌های آرایه فقط برای آرایه‌های فراخوانی داده پیاده سازی می‌شوند.

برش‌های آرایه برای رمزگشایی با داده‌های ثانویه ABI که در پارامترهای تابع منتقل می‌شوند مفید هستند:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract Proxy {
    /// @dev Address of the client contract managed by
proxy i.e., this contract
    address client;

    constructor(address _client) {
        client = _client;
    }

    /// Forward call to "setOwner(address)" that is
implemented by client
    /// after doing basic validation on the address
argument.
    function forward(bytes calldata _payload) external {
        // Since ABI decoding requires padded data, we
cannot
        // use abi.decode(_payload[:4], (bytes4)).
```

```
        bytes4 sig =
            _payload[0] |
            (bytes4(_payload[1]) >> 8) |
            (bytes4(_payload[2]) >> 16) |
            (bytes4(_payload[3]) >> 24);
        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.");
    }
}
```

Structها

سالیدیتی راهی برای تعریف نوع‌های جدید به صورت Struct فراهم می‌کند، که در مثال زیر نشان داده شده است:

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

// Defines a new type with two fields.
// Declaring a struct outside of a contract allows
// it to be shared by multiple contracts.
// Here, this is not really needed.
struct Funder {
    address addr;
    uint amount;
}

contract CrowdFunding {
```

```
    // Structs can also be defined inside contracts, which
makes them
    // visible only there and in derived contracts.
    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 is
return variable
        // We cannot use "campaigns[campaignID] =
Campaign(beneficiary, goal, 0, 0)"
        // because the RHS creates a memory-struct
"Campaign" that contains a mapping.
        Campaign storage c = campaigns[campaignID];
        c.beneficiary = beneficiary;
        c.fundingGoal = goal;
    }

    function contribute(uint campaignID) public payable {
        Campaign storage c = campaigns[campaignID];
        // Creates a new temporary memory struct,
initialised with the given values
        // and copies it over to storage.
        // Note that you can also use Funder(msg.sender,
msg.value) to initialise.
        c.funders[c.numFunders++] = Funder({addr:
msg.sender, amount: msg.value});
        c.amount += msg.value;
    }
```

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

این قرارداد عملکرد کامل قرارداد سرمایه گذاری جمعی را فراهم نمی‌کند، اما شامل مفاهیم پایه‌ای لازم برای درک structها است. نوع structها را می‌توان در داخل mappingها و آرایه‌ها استفاده کرد و خود آنها می‌توانند شامل mappingها و آرایه‌ها باشند.

ممکن است یک struct از یک نوع خود عضو داشته باشد، اگرچه struct می‌تواند مقدار نوع یک عضو از mapping باشد یا می واند شامل یک آرایه به اندازه پویا از نوع خود باشد. این محدودیت لازم است، زیرا اندازه ساختار باید محدود باشد.

توجه داشته باشید که چگونه در همه توابع، یک نوع struct به یک متغیر محلی با `storage` مکان داده اختصاص داده می‌شود. این کار struct را کپی نمی‌کند بلکه فقط یک مرجع را ذخیره می‌کند، در حقیقت تا انتسابات به اعضای متغیر محلی[1] در حالت[2] نوشته شود.

[1] local variable

[2] state

البته می‌توانید بدون اختصاص دادن به متغیر محلی، مستقیماً به اعضای struct دسترسی پیدا کنید، مانند در `campaigns[campaignID].amount = 0` .

توجه داشته باشید
تا سالیدیتی نسخه 0.7.0 ، memory-structs که شامل اعضای نوع‌های storage-only هستند (به عنوان مثال mappingها)، مجاز بودند و انتساباتی مانند `campaigns[campaignID] = Campaign(beneficiary, goal, 0, 0)` در مثال بالا کار می‌کند و فقط در silently از آن اعضا صرف نظر می‌کند.

3.6.3 انواع نگاشت‌ها[1]

نوع‌های نگاشت از سینتکس `mapping(_KeyType => _ValueType)` استفاده می‌کنند و متغیرهای نوع نگاشت با استفاده از سینتکس `mapping(_KeyType => _ValueType) _VariableName` مشخص می‌شوند. `_KeyType` می‌تواند هر نوع مقدار داخلی، `bytes` ، `string` یا هر نوع قرارداد یا enum باشد. سایر نوع‌های پیچیده یا تعریف شده توسط کاربر، مانند نگاشت یا (mapping)، structها یا انواع آرایه مجاز نیستند. `_ValueType` می‌تواند هر نوعی باشد، از جمله نگاشت‌ها، آرایه‌ها و structها.

می‌توانید نگاشتها را به صورت جداول هش در نظر بگیرید که عملاً مقداردهی اولیه می‌شوند به گونه ای که هر کلید ممکن وجود دارد و به مقداری نگاشت می‌شود که پیش نمایش بایت همه‌ی آن صفر می‌باشند، یک نوع

[1] Mapping Types

مقدار پیش فرض. شباهت در اینجا پایان می‌یابد، داده‌های کلیدی در نگاشت ذخیره نمی‌شوند، فقط از هش `keccak256` برای جستجوی مقدار استفاده می‌شود.

به همین دلیل، نگاشت‌ها طول یا مفهومی از کلید یا مقدار تنظیم شده ندارند و بنابراین بدون اطلاعات اضافی در مورد کلیدهای اختصاص داده شده پاک نمی‌شوند (به پاکسازی نگاشت مراجعه کنید).

نگاشت‌ها فقط می‌توانند یک مکان داده از `storage` را داشته باشند و بنابراین برای متغیرهای حالت، به عنوان نوع‌های مرجع `storage` در توابع، یا به عنوان پارامترهای توابع کتابخانه مجاز هستند. آنها نمی‌توانند به عنوان پارامترها یا پارامترهای بازگشتی توابع قرارداد که در معرض دید عموم قرار دارند، مورد استفاده قرار گیرند. این محدودیت‌ها برای آرایه‌ها و `struct`های حاوی نگاشت نیز صادق است.

شما می‌توانید متغیرهای حالت از نوع نگاشت را به صورت `public` علامت گذاری کنید و سالیدیتی یک گیرنده (getter) برای شما ایجاد می‌کند. `_KeyType` به یک پارامتر برای getter تبدیل می‌شود. اگر `_ValueType` یک مقدار نوع یا یک struct، getter `_ValueType` را برمی‌گرداند. اگر `_ValueType` یک آرایه یا نگاشت باشد، getter به صورت بازگشتی برای هر `_KeyType` یک پارامتر دارد.

در مثال زیر، قرارداد `MappingExample` یک نگاشت از `balances` عمومی را تعریف می‌کند، با نوع کلید یک `address` و یک نوع مقدار یک `uint`، و یک آدرس اتریوم را به یک مقدار صحیح بدون علامت[1] نگاشت می‌کند. از آنجا که `uint` یک نوع مقدار است، گیرنده مقداری را متناسب با نوع آن برمی‌گرداند که می‌توانید آن را در قرارداد `MappingUser` مشاهده کنید که مقدار را در آدرس مشخص شده برمی‌گرداند.

[1] unsigned integer value

```
// 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));
    }
}
```

مثال زیر یک نسخه ساده از توکن ERC20 است. `_allowances` نمونه‌ای از نوع نگاشت در داخل نوع نگاشت دیگر است. مثال زیر از `_allowances` برای ثبت مبلغی که شخص دیگری مجاز به برداشت از حساب شما است استفاده می‌کند.

```
// 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) {
        _transfer(sender, recipient, amount);
        approve(sender, msg.sender, amount);
        return true;
    }

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

        _allowances[owner][spender] = amount;
        emit Approval(owner, 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");

        _balances[sender] -= amount;
        _balances[recipient] += amount;
```

```
        emit Transfer(sender, recipient, amount);
    }
}
```

نگاشت های تکرارپذیر[1]

نمی‌توانید بر روی نگاشت‌ها تکرار کنید، یعنی نمی‌توانید کلیدهای آنها را بشمارید.گرچند امکان اجرای یک ساختار داده در بالای آنها و تکرار آن وجود دارد. به عنوان مثال، کد زیر یک کتابخانه `IterableMapping` را پیاده سازی می‌کند که قرارداد `User` سپس داده‌ها را نیز اضافه می‌کند و تابع `sum` تکرار می‌شود تا تمام مقادیر را جمع کند.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.8 <0.9.0;

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

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

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++;
```

[1] Iterable Mappings

```
            return false;
        }
    }

    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 iterate_start(itmap storage self) internal
view returns (uint keyIndex) {
        return iterate_next(self, type(uint).max);
    }

    function iterate_valid(itmap storage self, uint
keyIndex) internal view returns (bool) {
        return keyIndex < self.keys.length;
    }

    function iterate_next(itmap storage self, uint
keyIndex) internal view returns (uint r_keyIndex) {
        keyIndex++;
        while (keyIndex < self.keys.length &&
self.keys[keyIndex].deleted)
            keyIndex++;
        return keyIndex;
    }
```

```
    function iterate_get(itmap storage self, uint keyIndex)
internal view returns (uint key, uint value) {
        key = self.keys[keyIndex].key;
        value = self.data[key].value;
    }
}

// How to use it
contract User {
    // Just a struct holding our data.
    itmap data;
    // Apply library functions to the data type.
    using IterableMapping for itmap;

    // Insert something
    function insert(uint k, uint v) public returns (uint
size) {
        // This calls IterableMapping.insert(data, k, v)
        data.insert(k, v);
        // We can still access members of the struct,
        // but we should take care not to mess with them.
        return data.size;
    }

    // Computes the sum of all stored data.
    function sum() public view returns (uint s) {
        for (
            uint i = data.iterate_start();
            data.iterate_valid(i);
            i = data.iterate_next(i)
        ) {
            (, uint value) = data.iterate_get(i);
            s += value;
        }
    }
}
```

3.6.4 اپراتورهایی شامل LValues

اگر `a` یک LValue باشد (به عنوان مثال یک متغیر یا چیزی که می‌توان به آن اختصاص داد)، عملگرهای زیر به صورت مختصر در دسترس هستند:

`a += e` معادل `a = a + e` است. عملگرها `-=`، `*=`، `/=`، `%=`، `|=`، `&=` و `^=` بر این اساس تعریف می‌شوند. `a++` و `a--` معادل `a += 1` / `a -= 1` هستند اما این عبارت هنوز مقدار قبلی `a` را دارد. در مقابل ، `--a` و `++a` تأثیر یکسانی در `a` دارند اما مقدار را پس از تغییر برمی‌گردانند.

حذف[1]

`delete a` یک مقدار اولیه را برای نوع، به `a` اختصاص می‌دهد. یعنی برای اعداد صحیح معادل `a = 0` است، اما همچنین می‌تواند در آرایه ها مورد استفاده قرار گیرد، جایی که یک آرایه پویا از طول صفر یا یک آرایه ایستا با همان طول را با تمام عناصر تنظیم شده روی مقدار اولیه خود اختصاص می‌دهد. `delete a[x]` مورد را در شاخص `x` آرایه حذف می‌کند و سایر عناصر و طول آرایه را دست نخورده باقی می‌گذارد. این کار به طور ویژه به این معنی است که در آرایه شکاف ایجاد می‌کند. اگر قصد حذف موارد را دارید، نگاشت احتمالاً انتخاب بهتری است.

برای structها، یک struct را با تنظیم مجدد همه اعضا اختصاص می‌دهد. به عبارت دیگر، مقدار `a` پس از حذف `a` همانی است که اگر `a` بدون انتساب اعلام شود، با توجه به هشدار زیر:

`delete` تأثیری در نگاشت ندارد (زیرا ممکن است کلیدهای نگاشت دلخواه باشند و به طور کلی ناشناخته باشند). بنابراین اگر یک struct را حذف کنید، همه اعضا را که نگاشت نباشند مجدد تنظیم می‌کند و همچنین به عضوها بازگشت می‌یابد مگر اینکه آنها نگاشت باشند. با این حال، کلیدهای جداگانه و به آنچه نگاشت می‌شوند می‌توانند حذف شوند: اگر `a` نگاشت باشد، سپس `delete a[x]` مقدار ذخیره شده در `x` را حذف خواهد کرد.

توجه به این نکته مهم است که `delete a` واقعاً مانند انتساب به `a` رفتار می‌کند، یعنی یک شی جدید را در `a` ذخیره می‌کند. این تمایز زمانی قابل مشاهده‌است که `a` متغیر مرجع باشد: فقط یک `a` را خود مجدد تنظیم می‌کند نه مقداری که قبلاً به آن اشاره کرده بود.

```
// SPDX-License-Identifier: GPL-3.0
```

[1] delete

```
pragma solidity >=0.4.0 <0.9.0;

contract DeleteExample {
    uint data;
    uint[] dataArray;

    function f() public {
        uint x = data;
        delete x; // sets x to 0, does not affect data
        delete data; // sets data to 0, does not affect x
        uint[] storage y = dataArray;
        delete dataArray; // this sets dataArray.length to
zero, but as uint[] is a complex object, also
        // y is affected which is an alias to the storage
object
        // On the other hand: "delete y" is not valid, as
assignments to local variables
        // referencing storage objects can only be made
from existing storage objects.
        assert(y.length == 0);
    }
}
```

3.6.5 تبدیل بین نوع‌های اصلی

تبدیل‌های ضمنی[1]

در برخی موارد هنگام انتساب‌ها، هنگام انتقال آرگومان‌ها به توابع و هنگام اعمال عملگرها، کامپایلر به طور خودکار یک نوع تبدیل ضمنی اعمال می‌کند. به طور کلی، اگر به صورت سمنتیک معنا پیدا کند و هیچ اطلاعاتی از بین نرود، تبدیل ضمنی بین مقدار_نوع‌های مختلف امکان پذیر است.

به عنوان مثال، `uint8` به `uint16` و `int128` به `int256` قابل تبدیل است، اما `int8` به `uint256` قابل تبدیل نیست، زیرا `uint256` نمی‌تواند مقادیری مانند `-1` را نگه دارد.

[1] Implicit Conversions

اگر یک عملگر بر روی نوع‌های مختلف اعمال شود، کامپایلر سعی می‌کند به طور ضمنی یکی از عملوندها را به نوع دیگری تبدیل کند (این امر برای انتساب‌ها نیز صادق است). این بدان معناست که عملیات همیشه در نوع یکی از عملوندها انجام می‌شود.

برای جزئیات بیشتر در مورد اینکه کدام یک از تغییرات ضمنی امکان پذیر است، لطفاً با بخش‌هایی هر نوع مراجعه کنید.

در مثال زیر، `y` و `z` ، عملوندهای جمع، یک نوع ندارند، اما `uint8` را می‌توان به طور ضمنی به `uint16` تبدیل کرد و نه بالعکس. به همین دلیل، `y` قبل از اینکه جمع در نوع `uint16` انجام شود، به نوع `z` تبدیل می‌شود. `uint16` نوع حاصل از عبارت `y + z` است. از آنجا که به یک متغیر از نوع uint32 اختصاص داده شده است، تبدیل ضمنی دیگر پس از جمع انجام می‌شود.

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

تبدیل‌های صریح[1]

اگر کامپایلر اجازه تبدیل ضمنی را ندهد اما اطمینان دارید که یک تبدیل کار می‌کند، تبدیل صریح نوع گاهی اوقات امکان پذیر است. این ممکن است منجر به یک رفتار غیر منتظره شود و به شما امکان می‌دهد برخی از ویژگی‌های امنیتی کامپایلر را دور بزنید، بنابراین مطمئن شوید که نتیجه همان چیزی است که شما می‌خواهید و انتظار دارید! مثال زیر را در نظر بگیرید که `int` منفی را به `uint` تبدیل می‌کند:

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

[1] Explicit Conversions

در انتهای این قطعه کد، `x` دارای مقدار `0xfffff..fd` (64 نویسه hex) است که در نمایش مکمل 256 بیت -3 (منفی سه) است.

اگر یک عدد صحیح صریح به یک نوع کوچکتر تبدیل شود، بیت‌های مرتبه بالاتر بریده می‌شوند:

```
uint32 a = 0x12345678;
uint16 b = uint16(a); // b will be 0x5678 now
```

اگر یک عدد صحیح صریح به یک نوع بزرگتر تبدیل شود، در سمت چپ (یعنی در انتهای مرتبه بالاتر) قرار می‌گیرد. نتیجه تبدیل برابر با عدد صحیح اصلی[1] خواهد بود:

```
uint16 a = 0x1234;
uint32 b = uint32(a); // b will be 0x00001234 now
assert(a == b);
```

نوع‌های بایت‌های با اندازه ثابت[2] در هنگام تبدیل متفاوت عمل می‌کنند. می‌توان آنها را به عنوان توالی بایت‌های فردی در نظر گرفت و تبدیل به نوع کوچکتر، توالی را قطع می‌کند:

```
bytes2 a = 0x1234;
bytes1 b = bytes1(a); // b will be 0x12
```

اگر یک نوع بایت با اندازه ثابت صریحاً به یک نوع بزرگتر تبدیل شود، در سمت راست پر می‌شود. دستیابی به بایت در یک شاخص ثابت منجر به همان مقدار قبل و بعد از تبدیل خواهد شد (اگر شاخص هنوز در محدوده باشد):

```
bytes2 a = 0x1234;
bytes4 b = bytes4(a); // b will be 0x12340000
assert(a[0] == b[0]);
```

[1] original integer

[2] Fixed-size bytes

```
assert(a[1] == b[1]);
```

ازآنجایی که آرایه‌های بایت با اندازه ثابت در هنگام کوتاه کردن یا پر کردن رفتار متفاوتی دارند، درصورتی که هر دو از اندازه یکسانی برخوردار باشند، تبدیل صریح بین اعداد صحیح و آرایه‌های بایت با اندازه ثابت مجاز است. اگر می‌خواهید بین اعداد صحیح و آرایه‌های بایت با اندازه ثابت با اندازه‌های مختلف تبدیل کنید، باید از تبدیلات میانی استفاده کنید که قوانین کوتاه کردن و پر کردن مورد نظر را صریح می‌کند:

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

3.6.6 تبدیل بین لیترال‌ها و نوع‌های اصلی[1]

انواع عدد صحیح[2]

لیترال‌های عدد دسیمال و هگزادسیمال را می‌توان به طور ضمنی به هر نوع عددی صحیح که به اندازه کافی بزرگ باشد و بتوان آن را بدون کوتاه سازی نشان داد، تبدیل کرد:

```
uint8 a = 12; // fine
uint32 b = 1234; // fine
uint16 c = 0x123456; // fails, since it would have to
truncate to 0x3456
```

توجه داشته باشید

[1] Elementary Types

[2] Integer Types

قبل از نسخه 0.8.0، هر عدد تحت اعشاری یا هگزا دسیمال می‌تواند به ضمنی به یک نوع صحیح تبدیل شود. از 0.8.0، چنین تبدیل‌های صریح به اندازه تبدیل‌های ضمنی سختگیرانه هستند، یعنی تنها در صورتی مجاز هستند که کلمه تحت اللفظی در محدوده حاصله مطابقت داشته باشد.

آرایه‌های بایت با اندازه ثابت[1]

اعداد اعشاری لیترال را نمی‌توان به صورت ضمنی به آرایه‌های بایت با اندازه ثابت تبدیل کرد. می‌تواند لیترال‌های عددی هگزادسیمال باشد، اما فقط در صورتی که تعداد ارقام هگز دقیقاً متناسب با اندازه نوع بایت[2] باشد. به عنوان یک استثنا، هر دو لیترال‌های دسیمال و هگزادسیمال که مقدار آنها صفر است، می‌توانند به هر تایپ بایت با اندازه با اندازه ثابت تبدیل شوند:

```
bytes2 a = 54321; // not allowed
bytes2 b = 0x12; // not allowed
bytes2 c = 0x123; // not allowed
bytes2 d = 0x1234; // fine
bytes2 e = 0x0012; // fine
bytes4 f = 0; // fine
bytes4 g = 0x0; // fine
```

اگر تعداد کاراکترهای آنها با اندازه نوع بایت‌ها مطابقت داشته باشد، می‌توان لیترال‌های رشته‌ای و لیترال‌های رشته‌ای هگزی را به طور ضمنی به آرایه‌های بایت با اندازه ثابت تبدیل کرد:

```
bytes2 a = hex"1234"; // fine
bytes2 b = "xy"; // fine
bytes2 c = hex"12"; // not allowed
bytes2 d = hex"123"; // not allowed
bytes2 e = "x"; // not allowed
bytes2 f = "xyz"; // not allowed
```

[1] Fixed-size byte arrays

[2] bytes type

آدرس‌ها

همانطور که در آدرس لیترال‌ها توضیح داده شد، لیترال‌های هگز با اندازه صحیح که از آزمون چک‌سام[1] عبور می‌کنند از نوع `address` هستند. هیچ لیترال دیگری را نمی‌توان به طور ضمنی به نوع `address` تبدیل کرد.

تبدیل صریح از `bytes20` یا هر نوع عدد صحیح به `address` منجر به `address payable` می‌شود.

`address a` را می‌توان به `address payable` از طریق `payable(a)` تبدیل کرد.

[1] checksum