Solidity源文件结构

Solidity 源文件结构

源文件中可以包含任意数量的 合约定义 , 源文件引入 , pragma 、 using for 指令和 struct , enum , function , error 以及 常量 定义。

SPDX 版权许可标识

如果开源智能合约的源代码，就可以更好地建立对其的信任。由于提供源代码总是涉及到版权方面的法律问题，Solidity编译器鼓励使用机器可读的 标识 。 每个源文件都应该以这样的注释开始以说明其版权许可证。

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

编译器不会验证许可证是否属于 SPDX版权许可列表 ，但它会在 bytecode metadata 中包含提供的字符串。

如果你不想指定一个许可证，或者如果源代码不开源，请使用特殊值 UNLICENSED 。

请注意， UNLICENSED （不存在于SPDX许可证列表中）与 UNLICENSE （授予所有人所有权利）不同。

Solidity 遵循 the npm 建议 。 当然，提供这个注释并不能使你摆脱与许可有关的其他义务，比如必须在每个源文件中声明一个特定的许可标识符或人。

版权注释在文件的任何位置都可以被编译器识别，但建议把它放在文件的顶部。

关于如何使用SPDX许可标识符的更多信息可以在 SPDX website 找到。

Pragmas

关键字 pragma 版本标识指令，用来启用某些编译器检查， 版本 标识pragma 指令通常只对本文件有效，所以我们需要把这个版本 标识pragma 添加有的源文件。 如果使用了 import 导入 其他的文件, 标识pragma 并不会从被导入的文件，加入到导入的文件中。

版本标识

为了避免未来被可能引入不兼容更新的编译器所编译，源文件可以（也应该）使用版本 标识pragma 所注解。 我们力图把这类不兼容变更做到尽可能是，Solidity 本身就处在快速的发展之中，所以我们很难保证不引入修改语法的变更。 因此对含重大变更的版本，通读变更日志永远是好办法，变更有版本号表明更新点。

版本号的形式通常是 0.x.0 或者 x.0.0。

版本标识使用如下:

```
pragma solidity ^0.5.2;
```

这样，源文件将既不允许低于 0.5.2 版本的编译器编译， 也不允许高于（包含） 0.6.0 版本的编译器编译（第二个条件因使用 ^ 被添加）。 这种做是，编译器在 0.6.0 版本之前不会有重大变更，所以可确保源代码始终按预期被编译。 上面例子中不固定编译器的具体版本号，因此编译器的补丁用。

可以使用更复杂的规则来指定编译器的版本，表达式遵循 npm 版本语义。

注解

Pragma 是 pragmatic information 的简称，微软 Visual C++ 文档 中译为标识。 Solidity 中沿用 C ， C++ 等中的编译指令概念，用于告知编译器 如——译者注

注解

使用版本标准不会改变编译器的版本，它不会启用或关闭任何编译器的功能。 他仅仅是告知编译器去检查版本是否匹配， 如果不匹配，编译器就会误。

ABI Coder Pragma

使用pragma abicoder v1或pragma abicoder v2，您可以在ABI编码器和解码器的两种实现之间进行选择。

新的ABI编码器(v2)能够编码和解码任意嵌套的数组和结构。 它可能产生不那么优化的代码，也没有像旧的编码器那样接受那么多的测试，但在solic为是非实验性的。 您仍然需要使用pragma abicoder v2显式地激活它。 因为它将默认从Solidity 0.8.0开始激活，所以有一个选项可以使用pragma al择旧的编码器。

新编码器支持的类型集是旧编码器支持的类型的严格超集。使用它的合约可以与不受限制的合约交互。只有当非abicoder v2合约不尝试调用需要解码器支持的类型时，才有可能发生相反的情况。

注解

这个pragma应用于激活它的文件中定义的所有代码，而不管该代码最终在哪里结束。 这意味着，选择用ABI编码器v1编译源文件的合约仍然可以通过
合约继承使用新编码器的代码。 如果新类型只在内部使用而不在外部函数签名中使用，则允许这样做。

注解

在Solidity 0.7.4之前，可以通过使用pragma experimental ABIEncoderV2来选择ABI编码器v2，但不能显式地选择编码器v1，因为它是默认的。

标注实验性功能

第2个标注是用来标注实验性阶段的功能，它可以用来启用一些新的编译器功能或语法特性。 当前支持下面的一些实验性标注:

ABIEncoderV2

从Solidity 0.7.4开始， ABI coder v2 不在作为实验特性，而是可以通过``pragma abicoder v2`` 启用，查看上面。

SMTChecker

当我们使用自己编译的 Solidity 编译器（参考 build instructions） 的时候，这个模块可以启用。 在 Ubuntu PPA 发布的编译器的版本里 SMT solver
的，但是其他版本如：solc-js, Docker 镜像版本, Windows 二进制版本，静态编译的 Linux 二进制版本 是没有启用的。

注解

译者注： SMT 全称是：Satisfiability modulo theories，用来“验证程序等价性”。

使用 `pragma experimental SMTChecker;`, 就可以获得 SMT solver 额外的安全检查。但是这个模块目前不支持 Solidity 的全部语法特性，因此有可
警告信息。

导入其他源文件

语法与语义

Solidity 支持的导入语句来模块化代码，其语法跟 JavaScript（从 ES6 起）非常类似。 尽管 Solidity 不支持 default export 。

注解

ES6 即 ECMAScript 6.0，ES6是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布。 ——译者注

在全局层面上，可使用如下格式的导入语句:

```
import "filename";
```

`filename` 部分称为导入路径（ *import path*）。 此语句将从 “filename” 中**导入所有的全局符号到当前全局作用域**中（不同于 ES6，Solidity 是向后
的）。

这种形式已经不建议使用，因为它会无法预测地污染当前命名空间。 如果在“filename”中添加新的符号，则会自动添加出现在所有导入 “filename” 的
更好的方式是明确导入的具体 符号。

像下面这样，创建了新的 `symbolName` 全局符号，他的成员都来自与导入的 `″filename″` 文件中的全局符号，如： .. code-block:: solidity

> import * as symbolName from “filename”;

然后所有全局符号都以``symbolName.symbol``格式提供。 此语法的变体不属于ES6，但可能有用:

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

它等价于 `import * as symbolName from ″filename″;`。

如果存在命名冲突，则可以在导入时重命名符号。例如，下面的代码创建了新的全局符号 `alias` 和 `symbol2` ，引用的 `symbol1` 和 `symbol2` 来自 “filer

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

导入路径

为了能够在所有平台上支持可复制的构建，Solidity编译器必须抽象出存储源文件的文件系统细节。 因此，导入路径不直接引用主机文件系统中的文
编译器维护一个内部数据库(简称虚拟文件系统或VFS)，其中每个源单元被分配一个唯一的源单元名称，它是一个不透明的和非结构化的标识符。 ir
指定的导入路径被转换成一个源单元名，并用于

使用标准JSON API，可以直接提供所有源文件的名称和内容，作为编译器输入的一部分。 在这种情况下，源单元名称实际上是任意的。 然而，如果编译器自动地找到源代码并将其加载到VFS中，那么您的源单元名称需要以一种使导入回调能够定位它们的方式进行结构化。 当使用命令行编译器时，导入回调只支持从主机文件系统加载源代码，这意味着您的源单元名称必须是路径。 一些环境提供了更通用的自定义回调。 例如，Remix IDE提供了从HTTP、IPFS和Swarm url导入文件或直接引用NPM注册表中的包的工具。

有关虚拟文件系统和编译器使用的路径解析逻辑的完整描述，请参阅路径解析。

注释

可以使用单行注释（//）和多行注释（/*...*/）

```
// 这是一个单行注释。

/*
这是一个
多行注释。
*/
```

注解

单行注释由任何 unicode 行终止符(如采用UTF-8编码的：LF，VF，FF，CR，NEL，LS或PS) 终止。 在注释之后终止符代码仍然是源码的一部分。ASCII符号（NEL，LS和PS），它会导致解析器错误。

此外，有另一种注释称为 NatSpec 注释，详细可参考 编程风格指南 。

NatSpec注释使用 3 斜杠(///) 或两个星号注释，应该直接在函数声明和语句上方使用。