Introducción a los contratos inteligentes

El uso de los contratos inteligentes está aumentando a medida que evolucionan las cadenas de bloques. En la fase actual de las cadenas de bloques, muchos casos de uso giran en torno a la compilación de contratos inteligentes y aplicaciones empresariales. Se trata de un momento interesante en el que empezamos a ver aplicaciones reales de cadenas de bloques en casi todos los sectores.

1. ¿Qué es un contrato inteligente?

Un contrato inteligente es un programa que se almacena dentro de una cadena de bloques. Los contratos inteligentes amplían la cadena de bloques de los datos al código y representan un contrato entre las partes. El contrato se codifica y, cuando se produce una acción, el código se ejecuta y proporciona una respuesta.

Todos los términos y condiciones de los contratos se definen mediante programación. Una definición especifica las reglas, los requisitos y las recompensas de los participantes en la cadena de bloques. También especifica cómo se transfieren los recursos digitales entre las partes. A cada contrato inteligente se le asigna una dirección de 20 bytes que lo identifica de forma única.

Los contratos inteligentes se ejecutan por sí solos, envían eventos que desencadenan transiciones de estado y llaman a funciones. Y, además, son ideales para la tecnología de cadena de bloques porque permiten a la gente que no se conoce hacer negocios de forma segura y sin intermediarios.

Los contratos inteligentes suelen usarse con Ethereum. Ethereum es la primera cadena de bloques programable del mundo. Permite definir contratos inteligentes para facilitar la transferencia de recursos digitales, como la criptomoneda Ether.

El lenguaje que se va a usar para escribir contratos es Solidity. Solidity es un sistema *Turing completo*; es decir, que puede escribir contratos complicados de una manera claramente definida y codificada.

Dado que cada transición de estado se registra y es inmutable, debe probar exhaustivamente el contrato antes de publicarlo en un entorno de producción. Las correcciones de errores pueden ser costosas e incluso provocar daños importantes en el sistema.

Los contratos inteligentes tienen estas propiedades y ventajas clave:

- **Transparencia**: los usuarios de cadenas de bloques pueden leer los contratos inteligentes y acceder a ellos mediante las API.
- **Inmutabilidad**: la ejecución de contratos inteligentes crea registros que no se pueden cambiar.
- **Distribución**: los nodos de la red validan y comprueban la salida del contrato. Los estados del contrato pueden ser visibles públicamente. En algunos casos, incluso las variables "privadas" son visibles.

Casos de uso

Los contratos inteligentes pueden ser realmente útiles para muchos sectores y procesos. Analice los siguientes casos de uso.

Seguros: cuando se producen determinados eventos, los contratos inteligentes desencadenan automáticamente una notificación que simplifica el proceso de notificaciones. Después, para determinar el importe de indemnización que reciben los usuarios, los detalles de la notificación se pueden registrar en la cadena de bloques. Esta funcionalidad puede reducir los tiempos de procesamiento y los errores humanos.

Votaciones: los contratos inteligentes pueden ayudar a que los votos sean automáticos y transparentes. Cada contrato actúa como una votación, que representa la identidad de un votante. Dado que una cadena de bloques es inmutable, lo que significa que no se puede cambiar, los votos no se pueden alterar.

Cadenas de suministro: a medida que los elementos se mueven por la cadena de suministro, los contratos inteligentes pueden registrar la propiedad y confirmar quién es responsable de un producto en un momento dado. En cualquier fase, el contrato inteligente se puede usar para averiguar exactamente dónde deben estar los productos. Si alguna parte de la cadena de suministro no se entrega a tiempo, todos los demás partes sabrán dónde se produjo el problema.

Mantenimiento de registros: muchos sectores pueden usar contratos inteligentes para mejorar la velocidad y la seguridad del mantenimiento de los registros. La tecnología de cadena de bloques se puede usar para digitalizar registros y cifrarlos y almacenarlos de forma segura. Además, el acceso se puede canalizar para que solo los usuarios autorizados puedan acceder a los registros.

Posesión de propiedades: los contratos inteligentes pueden registrar quién es el propietario de la propiedad. Por tanto, son una manera rápida y rentable de registrar la propiedad. Los contratos inteligentes también pueden ayudar a facilitar la transferencia de la propiedad de forma oportuna y segura.

1. ¿Qué es un contrato inteligente?

1.1. Herramientas para trabajar con contratos inteligentes

Muchas herramientas pueden ayudarlo desarrollar eficazmente contratos inteligentes. En las secciones siguientes se sugieren algunos entornos de desarrollo integrado (IDE), extensiones y marcos de trabajo que puede explorar.

IDE

- **Visual Studio Code**: un editor de código que se ha redefinido y optimizado para compilar y depurar aplicaciones web y en la nube modernas. En este módulo, usamos Visual Studio Code para todos los ejercicios.
- **Remix**: un compilador y un IDE basados en explorador que se pueden usar para compilar contratos Ethereum en el lenguaje Solidity y para depurar transacciones. Remix constituye una excelente manera de explorar contratos de ejemplo. Puede usarlo para escribir, probar e implementar sus propios contratos. En este módulo no usamos Remix, pero puede probarlo por su cuenta en los ejercicios de contratos.

Extensiones

- **Extensión de Truffle para VS Code**: esta extensión puede simplificar la creación, compilación e implementación de contratos inteligentes en los libros de contabilidad de Ethereum. Esta extensión incluye la integración con Truffle, Ganache y más herramientas y servicios. En este módulo, usaremos esta extensión para escribir y probar contratos inteligentes.

Marcos de trabajo

- **Truffle Suite**: use el conjunto de herramientas de Truffle para probar los contratos de Ethereum antes de implementarlos en los libros de contabilidad públicos e incurrir en costos reales. Para facilitar su trabajo como programador, desarrolle en un entorno local. El conjunto de herramientas incluye Truffle, Ganache y Drizzle. En este módulo, usaremos Truffle.
- **OpenZeppelin**: use las herramientas de OpenZeppelin para escribir, implementar y usar aplicaciones descentralizadas. OpenZeppelin proporciona dos productos: la biblioteca de contratos y el SDK. En este módulo no usaremos OpenZeppelin, pero más adelante puede probarlo mientras escribe aplicaciones de cadenas de bloques seguras.

2. ¿Qué es Solidity?

Solidity es un lenguaje orientado a objetos para escribir contratos inteligentes.

Los contratos inteligentes son programas almacenados dentro de una cadena de bloques. Especifican las reglas y el comportamiento de cómo se transfieren los recursos digitales. Solidity sirve para programar contratos inteligentes para la plataforma de cadena de bloques Ethereum. Los contratos inteligentes contienen lógica de estado y programable. Las transacciones dan como resultado funciones que se ejecutan en contratos inteligentes, y estos contratos, a su vez, permiten crear un flujo de trabajo empresarial.

2. ¿Qué es Solidity?

2.1. Introduccion

Solidity es el lenguaje de programación más popular de la cadena de bloques Ethereum.

Solidity es un lenguaje de alto nivel que se basa en otros lenguajes de programación, como C++, Python y JavaScript. Si conoce alguno de estos lenguajes, el código de Solidity le será familiar.

Solidity es de tipos estáticos, lo que significa que la comprobación de tipos se realiza en el tiempo de compilación y no en el tiempo de ejecución, como sucede con los lenguajes de tipos dinámicos. En los lenguajes de tipos estáticos hay que especificar el tipo de cada variable. Por ejemplo, Python y JavaScript son lenguajes de tipos dinámicos, mientras que C++ es de tipos estáticos.

Solidity admite la herencia, de modo que las funciones, variables y otras propiedades existentes en un contrato se pueden usar en otro. El lenguaje también admite tipos complejos definidos por el usuario, como estructuras y enumeraciones, que permiten agrupar tipos de datos relacionados.

Solidity es un lenguaje de programación de código abierto que cuenta con una comunidad de colaboradores cada vez más nutrida. Para más información sobre el proyecto Solidity y cómo colaborar en él, vea este repositorio de GitHub.

2. ¿Qué es Solidity?

2.2. ¿Que es Ethereum?

Antes de continuar, conviene familiarizarse también con Ethereum.

Ethereum es una de las plataformas de cadena de bloques más populares, después de Bitcoin. Se trata de una tecnología desarrollada por una comunidad y tiene su propio criptomoneda, denominada Ether (ETH), que se puede comprar y vender.

Lo que singulariza a Ethereum es que es la "cadena de bloques programable del mundo". Si usamos Ethereum, podremos codificar las definiciones de contrato, también conocidas como contratos inteligentes. Los contratos inteligentes sirven para describir el modo en que los participantes de la cadena de bloques transfieren sus activos digitales. Solidity es el

lenguaje de programación principal que se usa para desarrollar en la plataforma Ethereum y lo han creado y mantenido los desarrolladores de Ethereum.

Ethereum Virtual Machine
Los contratos de Solidity se ejecutan en Ethereum Virtual Machine (EVM, en su forma abreviada), que es un entorno de espacio aislado que está completamente aislado. En la red, este entorno no accede a nada más que a los contratos que ejecuta. Por ahora no es preciso que ahondemos demasiado en EVM, solo debemos recordar que los contratos inteligentes de Solidity se van a implementar y ejecutar en un entorno virtual.

2. ¿Qué es Solidity?

2.3. Descripción de los conceptos básicos del lenguaje

Cualquier contrato de Solidity suele incluir lo siguiente:

- Directivas pragma
- Variables de estado
- Functions
- Eventos

Si bien es cierto que hay más elementos que es necesario conocer para programar contratos inteligentes de nivel de producción, con estos de aquí podremos comenzar con buen pie.

Con tener claros estos conceptos, podremos empezar a escribir contratos inteligentes para diferentes casos de uso de forma inmediata.

Directivas pragma

Pragma es la palabra clave que se usa para pedir al compilador que compruebe si su versión de Solidity coincide con la que necesitamos. Si es la misma, nuestro archivo de código fuente funcionará correctamente; si no, el compilador generará un error.

Procure incluir siempre la versión más reciente de Solidity en la definición del contrato. Para saber cuál es la versión actual de Solidity, visite el sitio web de Solidity. Use la versión más reciente en el archivo de código fuente.

Una directiva pragma de versión tiene el siguiente aspecto:

```
pragma solidity ^0.7.0;
```

Esta línea significa que el archivo de código fuente se compilará con un compilador que tiene una versión posterior a `0.7.0`, hasta `0.7.9`. A partir de la versión `0.8.0`, lo más probable es que se introduzcan cambios importantes que el archivo de código fuente no sea capaz de compilar correctamente.

Variables de estado

Las variables de estado son esenciales en cualquier archivo de código fuente de Solidity. Los valores de variable de estado se almacenan permanentemente en el almacenamiento del contrato.

Solidity

```
pragma solidity >0.7.0 <0.8.0;

contract Marketplace {
    uint price; // State variable
```

Nota: ***Los archivos de origen del contrato siempre empiezan con la definición contrato nombreDeContrato.***

En este ejemplo, la variable de estado se denomina `price`, con el tipo **uint**. El tipo entero uint indica que esta variable es un entero sin signo con 256 bits, lo que significa que puede almacenar números positivos dentro del intervalo entre 0 y 2^{256}-1.

En todas las definiciones de variables hay que especificar el tipo y el nombre de la variable.

También podemos especificar la visibilidad de una variable de estado de uno de los siguientes modos:

- **public**: la variable forma parte de la interfaz del contrato y se puede acceder a ella desde otros contratos.
- **internal**: solo se puede acceder a la variable internamente desde el contrato actual.
- **private**: solo es visible en el contrato en el que se definen.

Functions

Dentro de un contrato, las unidades ejecutables de código se conocen como funciones. Las funciones describen una única acción para lograr una tarea. Son reutilizables, y se les puede llamar también desde otros archivos de código fuente, como una biblioteca. Las funciones de Solidity se comportan de forma similar a las funciones de otros lenguajes de programación.

Este es un sencillo ejemplo de cómo definir una función:

Solidity

```
pragma solidity >0.7.0 <0.8.0;

contract Marketplace {
    function buy() public {
        // ...
    }
```

```
}
```
Este código muestra una función con el nombre `buy` que tiene visibilidad pública, lo que significa que otros contratos pueden acceder a ella. Las funciones pueden usar uno de los siguientes especificadores de visibilidad: **public**, **private**, **internal** y **external**.

Las funciones se pueden llamar de forma interna o externa desde otro contrato. Pueden aceptar parámetros, así como devolver variables para pasarse parámetros y valores entre ellas.

Este es un ejemplo de una función que acepta un parámetro (un entero llamado `price`) y devuelve un entero:

Solidity

```
pragma solidity >0.7.0 <0.8.0;

contract Marketplace {
    function buy(uint price) public returns (uint) {
        // ...
    }
}
```

Modificadores de funciones

Podemos usar modificadores de funciones para cambiar el comportamiento de las funciones. Su cometido es comprobar una condición antes de que la función se ejecute. Un ejemplo sería una función que comprobara que solamente los usuarios designados como vendedores pueden incluir un artículo en un catálogo para su venta.

Solidity

```
pragma solidity >0.7.0 <0.8.0;

contract Marketplace {
    address public seller;

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can put an item up for sale."
        );
        _;
    }

    function listItem() public view onlySeller {
        // ...
    }
}
```

Este ejemplo consta de los siguientes elementos:

- Una variable de tipo **address** que almacena la dirección de Ethereum de 20 bytes del usuario vendedor. Profundizaremos en estas variables más adelante en este módulo.
- Un modificador denominado `onlySeller`, que describe que solo un vendedor puede incluir un artículo en el catálogo.
- Un símbolo especial (`_;`) para indicar dónde se inserta el cuerpo de la función.
- Una definición de función que usa el modificador `onlySeller`.

Otros modificadores de función que se pueden usar en la definición de función son:

- **pure**, para describir funciones que no permiten modificaciones o el acceso del estado.
- **view**, para describir funciones que no permiten modificaciones del estado.
- **payable**, para describir las funciones que pueden recibir Ethers.

Eventos

Los eventos describen las acciones que se realizan en el contrato. Al igual que las funciones, los eventos tienen parámetros que se deben especificar cuando se llama al evento.

Para llamar a un evento, debemos usar la palabra clave **emit** junto con el nombre del evento y sus parámetros.

Solidity

```
pragma solidity >0.7.0 <0.8.0;

contract Marketplace {
    event PurchasedItem(address buyer, uint price);

    function buy() public {
        // ...
        emit PurchasedItem(msg.sender, msg.value);
    }
}
```

Cuando se llama a un evento, se captura como una transacción en el registro de transacciones, que es una estructura de datos especial en la cadena de bloques. Estos registros se asocian con la dirección del contrato, se incorporan a la cadena de bloques y permanecen allí para siempre. El registro y sus datos de evento no se pueden modificar, y no se puede acceder a ellos desde los contratos.

2. ¿Qué es Solidity?

2.4. Exploración de los tipos de valores

Los tipos de valor principales que se usan al escribir contratos son **enteros**, **valores booleanos**, **string literal**, **direcciones** y **enumeraciones**.

Enteros

En todos los archivos de código fuente de Solidity se usan enteros. Representan números enteros y pueden llevar o no signo. Los enteros varían en tamaño de almacenamiento de 8 a 256 bits.

- Con signo: incluye números negativos y positivos. Se puede representar como **int**.
- Sin signo: incluyen únicamente números positivos. Se puede representar como **uint**.

Si no se especifica un número de bits, el valor predeterminado es de 256.

En los enteros se pueden usar las siguientes operaciones:

- Comparaciones: `<=`, `<`, `==`, `!=`, `>=`, `>`
- Operadores de bits: `& (and)`, `| (or)`, `^ (bitwise exclusive)`, `~ (bitwise negation)`
- Operadores aritméticos: `+ (addition)`,`- (subtraction)`, `* (multiplication)`, `/ (division)`, `% (modulo)`, `** (exponential)`

Estos son algunos ejemplos de definiciones de enteros:

Enteros

En todos los archivos de código fuente de Solidity se usan enteros. Representan números enteros y pueden llevar o no signo. Los enteros varían en tamaño de almacenamiento de 8 a 256 bits.

- Con signo: incluye números negativos y positivos. Se puede representar como **int**.
- Sin signo: incluyen únicamente números positivos. Se puede representar como **uint**.

Si no se especifica un número de bits, el valor predeterminado es de 256.

En los enteros se pueden usar las siguientes operaciones:

- Comparaciones: `<=`, `<`, `==`, `!=`, `>=`, `>`
- Operadores de bits: `& (and)`, `| (or)`, `^ (bitwise exclusive)`, `~ (bitwise negation)`
- Operadores aritméticos: `+ (addition)`,`- (subtraction)`, `* (multiplication)`, `/ (division)`, `% (modulo)`, `** (exponential)`

Estos son algunos ejemplos de definiciones de enteros:

Solidity

```
int32 price = 25; // signed 32 bit integer
uint256 balance = 1000; // unsigned 256 bit integer

balance - price; // 975
```

```
2 * price; // 50
price % 2; // 1
```

Valores booleanos

Los valores booleanos se definen usando la palabra clave **bool**. Su valor siempre es `true` o `false`.

Así es como se pueden definir:

Solidity

```
bool forSale; //true if an item is for sale
bool purchased; //true if an item has been purchased
```

Los valores booleanos se suelen usar en instrucciones de comparación. Por ejemplo:

Solidity

```
if(balance > 0 & balance > price) {
    return true;
}

if(price > balance) {
    return false;
}
```

También se pueden usar en parámetros de funciones y en tipos de valores devueltos.

Solidity

```
function buy(int price) returns (bool success) {
    // ...
}
```

Literales de cadena

También se usan String literals en la mayoría de los archivos de contrato. que son caracteres o palabras entre comillas simples o dobles.

Solidity

```
    String shipped = "shipped"; // shipped
    String delivered = 'delivered'; // delivered
    String newItem = "newItem"; // newItem
```

Además, con string literals se pueden usar los siguientes caracteres de escape.

- `\<newline>` (escapa a una línea nueva)
- `\n` (línea nueva)
- `\r` (retorno de carro)
- `\t` (pestaña)

Dirección

Una dirección es un tipo con un valor de 20 bytes que representa una cuenta de usuario de Ethereum. Este tipo puede ser **address** a secas o **address payable**.

La diferencia entre ambos es que un tipo **address payable** es una dirección a la que se puede enviar Ethers y que contiene los miembros adicionales `transfer` y `send`.

Solidity

```
address payable public seller; // account for the seller
address payable public buyer; // account for the user

function transfer(address buyer, uint price) {
    buyer.transfer(price); // the transfer member transfers the price of the item
}
```

Enumeraciones

En Solidity, puede usar enumeraciones para crear un tipo definido por el usuario. Decimos "definido por el usuario" porque la persona que crea el contrato decide qué valores se van a incluir. Las enumeraciones se pueden usar para presentar muchas opciones seleccionables, una de las cuales es obligatoria.

Una **enumeración** se puede usar, por ejemplo, para presentar diferentes estados de un artículo. Las enumeraciones pueden considerarse como una presentación de varias opciones de respuesta donde todos los valores están predefinidos y hay que seleccionar uno. Las enumeraciones se pueden declarar en definiciones de contrato o de biblioteca.

Solidity

```
enum Status {
    Pending,
    Shipped,
    Delivered
}

Status public status;

constructor() public {
    status = Status.Pending;
}
```

2. ¿Qué es Solidity?

2.5. Exploración de los tipos de referencia

Al escribir contratos, también debemos saber qué son los tipos de referencia.

A diferencia de los tipos de valor, que siempre pasan una copia independiente del valor, los tipos de referencia proporcionan una **ubicación de datos** al valor. Los tres tipos de referencia son: **structs**, **matrices** y **asignaciones**.

Ubicación de los datos

Cuando usamos un tipo de referencia, debemos proporcionar expresamente la ubicación del almacenamiento de datos de ese tipo. Para especificar la ubicación de los datos donde el tipo se almacena, podemos usar las siguientes opciones:

- `memory`:
  - Ubicación donde se almacenan los argumentos de la función
  - Su duración se limita a lo que dura una llamada a una función externa
- `storage`:
  - Ubicación donde se almacenan las variables de estado
  - Tiene una duración limitada a la duración del contrato
- `calldata`:
  - Ubicación donde se almacenan los argumentos de la función
  - Esta ubicación se requiere en los parámetros de las funciones externas, pero también se puede usar en otras variables
  - Su duración se limita a lo que dura una llamada a una función externa

Los tipos de referencia siempre crean una copia independiente de los datos.

Este es un ejemplo que ilustra cómo usar un tipo de referencia:

Solidity

```
contract C {

  uint[] x;

  // the data location of values is memory
  function buy(uint[] memory values) public {
      x = values; // copies array to storage
      uint[] storage y = x; //data location of y is storage
      g(x); // calls g, handing over reference to x
      h(x); // calls h, and creates a temporary copy in memory
  }

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

Matrices

Las matrices son una forma de almacenar datos similares en una estructura de datos establecida. Las matrices pueden tener un tamaño fijo o dinámico. Sus índices comienzan en 0.

Para crear una matriz de tamaño fijo `k` y con un tipo de elemento `T`, escribiríamos `T[k]`. Para obtener una matriz de tamaño dinámico, escribiría `T[]`.

Los elementos de matriz pueden ser de cualquier tipo. Así pueden contener **uint**, **memory** o **bytes**. Las matrices también pueden incluir **asignaciones** o **structs**.

En los siguientes ejemplos se muestra la creación de una matriz:

Solidity

```
uint[] itemIds; // Declare a dynamically sized array called itemIds
uint[3] prices = [1, 2, 3]; // initialize a fixed size array called prices, with
prices 1, 2, and 3
uint[] prices = [1, 2, 3]; // same as above
```

Miembros de la matriz

Los siguientes miembros pueden manipular y obtener información sobre las matrices:

- **length**: permite obtener la longitud de una matriz.
- **push()**: permite anexar un elemento al final de la matriz.
- **pop**: permite quitar un elemento del final de una matriz.

Estos son algunos ejemplos:

Solidity

```
// Create a dynamic byte array
bytes32[] itemNames;
itemNames.push(bytes32("computer")); // adds "computer" to the array
itemNames.length; // 1
```

Estructuras

Los structs son tipos personalizados que un usuario puede definir para representar objetos del mundo real. Las estructuras se suelen usar a modo de esquema o para representar registros.

Ejemplo de una declaración de estructura:

Solidity

```
struct Items_Schema {
    uint256 _id;
```

```
    uint256 _price;
    string _name;
    string _description;
}
```

Tipos de asignaciones

Las asignaciones son pares clave-valor que se encapsulan o empaquetan juntos. Las asignaciones son más parecidas a los diccionarios u objetos de JavaScript, y se suelen usar para modelar objetos del mundo real y realizar búsquedas de datos más rápidas. Los valores pueden incluir tipos complejos como estructuras, lo que hace que el tipo de asignación sea flexible y legible para el usuario.

En el siguiente ejemplo de código se usa el struct `Items_Schema` y se guarda una lista de artículos representados por `Items_Schema` como un diccionario. La asignación imita en este sentido a una base de datos.

Solidity

```
contract Items {
    uint256 item_id = 0;

    mapping(uint256 => Items_Schema) public items;

    struct Items_Schema {
      uint256 _id:
      uint256 _price:
      string _name;
    }

    function listItem(uint 256 memory _price, string memory _name) public {
      item_id += 1;
      items[vehicle_id] = Items_Schema(item_id, _price, _name);
    }
}
```