




SMART CONTRACTS BLOCKCHAIN



Solidity 語法介紹 **01**

TABLE OF CONTENTS



02 智能合約實作



01

Solidity 語法介紹

程式結構

pragma keyword

合約程式碼的第一行是由 pragma 開頭，用來描述使用哪個版本的編譯器。

版本範圍可以用 >, >=, <, <=, = 等等符號約定。

contract

合約定義，為合約的本體內容。

```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6  * @title Storage
7  * @dev Store & retrieve value in a variable
8  */
9 contract Storage {
10
11     uint256 number;
12
13     /**
14      * @dev Store value in variable
15      * @param num value to store
16      */
17     function store(uint256 num) public {
18         number = num;
19     }
20
21     /**
22      * @dev Return value
23      * @return value of 'number'
24      */
25     function retrieve() public view returns (uint256){
26         return number;
27     }
28 }
```

getter 函式

狀態變數再宣告後，Compiler 將會自動為所有

Public 變數 創建 getter 函式。

getter 函式 的可見度為 “external”。

公開的 mapping 不會自動生成 getter 函式。

```
1  pragma solidity ^0.4.20;  
2  contract C {  
3      uint public data = 42;  
4      function data() public returns(uint){  
5          return data  
6      }  
7  }
```

函式與狀態變數的可見度

external

可以被其他合約呼叫，
但不能被內部呼叫，
除非使用 `this` 語法。

public

可以接受外部、內部呼叫，
若是公開狀態變數，會產生 `getter` 函式。

internal

只能在內部被存取，
且不需要使用 `this` 語法。

private

只能在內部被呼叫，
且不能夠被繼承。

修飾函式

function 語法

```
function name([argument, ...]) [visibility] [view|pure]  
    [payable] [modifier, ...] [returns([argument, ...])];
```

Example

```
function getResult() public view returns(uint product, uint sum){ ... }
```

修飾函式

當函式將不更改任何狀態，
可以在函式中標記 “view”

- 修改狀態變數的值
- 觸發事件
- 創建其他智能合約
- 呼叫 selfdestruct
- 呼叫函式發送 Ether
- 呼叫不是 view、pure 的函式

View

當函式將不更改任何狀態
且不進行讀取時，
可以在函式中標記 “pure”

Pure

修飾函式

若要讓函式可以接收以太幣，
必須在宣告時加入payable。

Payable

```
function fund(string _name) public payable {  
    // ...  
}
```

Function Modifiers

接續回原本的function

```
modifier onlyOwner() {  
    require(msg.sender == owner, "Permission denied.");  
    _;  
}  
function setName(string _name) public onlyOwner returns (string) {  
    name = _name;  
    return name;  
}
```

在function執行前做預先處理

Mapping

```
mapping(_KeyType => _ValueType) _VariableName;
```

- Mapping是動態大小的陣列
- _KeyType 不支援 enum 和 struct 的型態
- _ValueType 則無限制
- Key 對應到的 value 將全部被初始化



錯誤處理

`require(bool condition)`
`require(bool condition, string message)`

- 檢查條件是否符合預期，若否則狀態將被復原。

- 退回剩餘 Gas 費用。

Require

```
require(owner = msg.sender);  
require(owner = msg.sender, "no permission");  
revert();  
revert("Error");
```

`revert()`
`revert(string message)`

- 執行將被終止，並回復回修改前的狀態。

- 退回剩餘 Gas 費用。

Revert



特殊變數、函式

msg.sender

(address)

正在與合約互動的帳戶

msg.value

(uint256)

發送的以太幣數量(wei)

selfdestruct(address recipient)

摧毀目前的合約，並將合約內的錢移轉給參數中的位址。

this

只當前的合約，
可以轉換為合約位址。

<address>.balance

(uint256)
餘額 (wei)

**<address>.transfer
(uint256 amount)**

(wei)
將合約內的錢移轉給指定的位址，



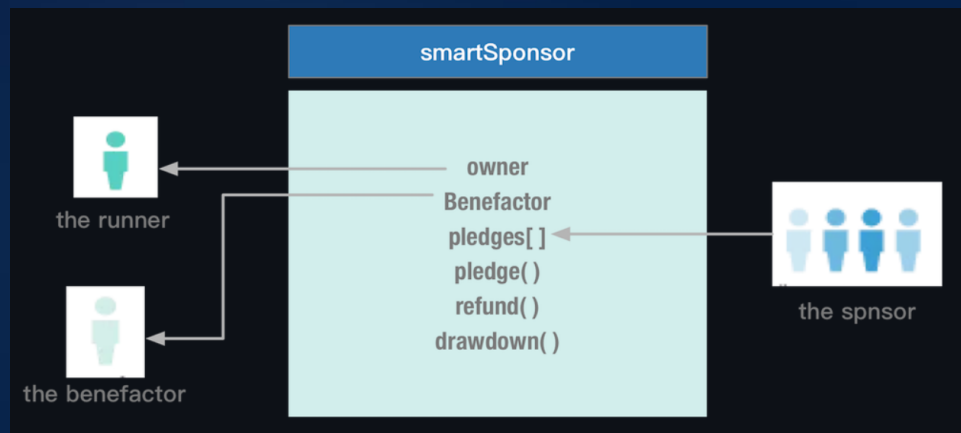
02

實作智能合約範例

<https://reurl.cc/6EWgVk>

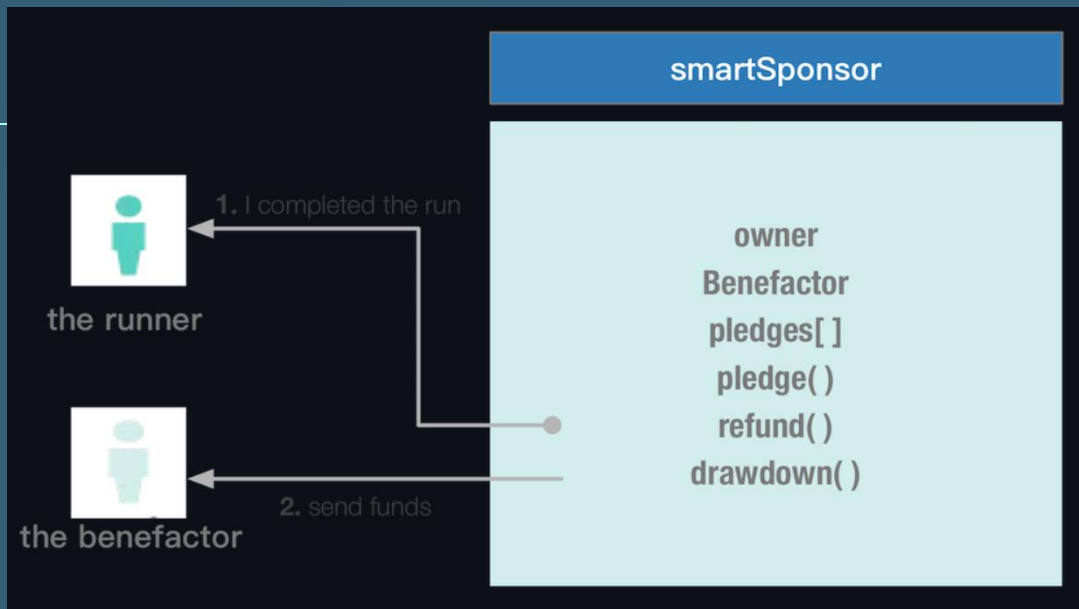
慈善機構募款活動

- 一個慈善機構舉辦募款活動，我們將機構稱為 thebenefactor 。
- 為機構發起募款的人，我們將他稱為 therunner 。
- 對 therunner 贊助的人，我們將它成為 thesponsor 。
- 由 therunner 進行贊助合約的執行。
- 當合約建立時，runner 會任命接收贊助的 thebenefactor 。
- 用戶若要進行贊助，必須呼叫一個在智能合約上的函式，將以太幣從贊助者的帳戶中轉移至“合約”。



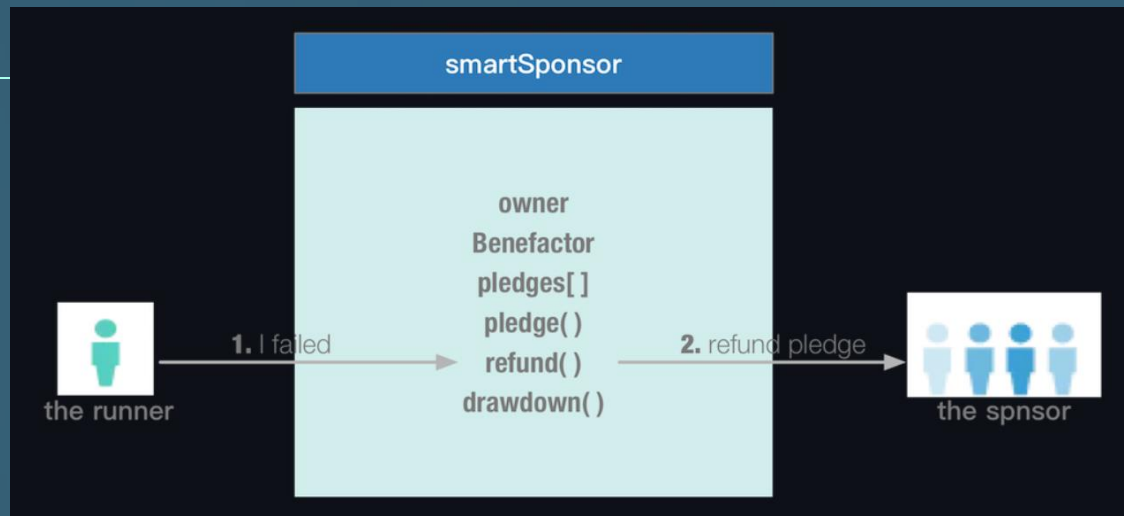
合約可能結果(1)

合約按照計畫進行，
runner將合約內的資金
轉移給受捐助之機構。



合約可能結果(2)

合約因某些原因無法如期執行，
runner 將合約內的資金退還給
贊助者。



合約中的函式

smartSponsor

合約的建構子，
由此初始化合約。

pledge

贊助者呼叫此合約
來捐贈以太幣。

getPot

回傳目前儲存在合約中
的以太幣總數。

refund

將贊助者的錢全數退回，
只有runner可以呼叫此合約。

drawdown

將合約中的資金總數
轉移給受捐助帳戶，
只有runner可以呼叫此合約。



Remix



Thanks

