

# 课程作业一

3180105160 赵天辞

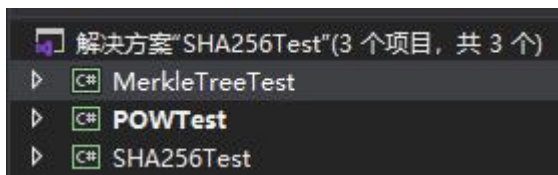
## 代码运行环境：.NET Core 3.1 或更高

如何运行？：

1. 安装完.NET Core SDK（3.1 或更高）后打开 SHA256Test 文件夹，如图

MerkleTreeTest	2020/12/7 19:24	文件夹	
POWTest	2020/12/7 20:04	文件夹	
SHA256Test	2020/12/7 16:51	文件夹	
SHA256Test.sln	2020/12/7 19:31	Microsoft Visual...	3 KB

2. 双击解决方案（VS2019），解决方案中有三个项目



从上至下依次对应：

实验说明 SHA256 如何用于区块数据锁定，  
实验说明 SHA256 在 PoW 中的作用，  
SHA256 代码实现

3. 选择对应项目编译执行，或进入对应项目目录下的 bin/Debug/netcoreapp3.1 执行.exe 程序

## 一、SHA256 实现

### 1.1 SHA256 类使用方法

1. 实例化 SHA256 对象

```
var encoder = new SHA256();
```

2. 设置要加密的内容

```
var content = "哈希函数，又称散列算法，是一种从任何一种数据中创建小的数字“指纹”的方法"
```

3. 获取加密结果

```
var res = encoder.GetSHA256Code(content);
```

### 1.2 样例程序如何运行？

项目 SHA256Test 编译运行后，输入要进行哈希运算的内容（字符串），然后回车  
输出结果有两行：

第一行为本人所写的 SHA256 算法运算结果

第二行为.NET 内置加密库 System.Security.Cryptography 的 SHA256 算法加密结果

### 1.3 运行结果

```
This is a test
C7BE1ED902FB8DD4D48997C6452F5D7E509FBCDBE2808B16BCF4EDCE4C07D14E
C7BE1ED902FB8DD4D48997C6452F5D7E509FBCDBE2808B16BCF4EDCE4C07D14E
```

```
这是一个测试
9F7AC1E7609C31D81D8C3C255798766ABE77F3BF94489E94604BE901CE2A855C
9F7AC1E7609C31D81D8C3C255798766ABE77F3BF94489E94604BE901CE2A855C
```

```
哈希函数，又称散列算法，是一种从任何一种数据中创建小的数字“指纹”的方法
A26744D13125B036239275E8C153AF0474F4212762A20FAB89938379EAE5B55D
A26744D13125B036239275E8C153AF0474F4212762A20FAB89938379EAE5B55D
```

```
alnqalkjalkgjka;nglkafoiuaivoiaslknaw;ltiqoopqijgkjanbfkjbakjfhuiwehoiwqeethewngkjevksnac1kankljgetoipqinNlnzncla;jai;
LEALEKANlnalhgogegikanlknalkfhalkanfkwuiqhotiqnkn, ank jnz; alavjacipuqoi tnqkrnlvjq5093j5p59rheqi 3ohrhoi qnr23hoirwoihqn
fowaioijlkvoiafug83u92joi5h1h1215k112krlkajofiauf0wroqj3oi53oi2h511n2rjkqwnqkfjhgqwfjilamlknafkkhoiqjtlkqflnfmaslioviaow
31k21k25nlnflani,jgklengklqng13n21kn115nlkwnf1gnfk123n4
C49C4182CBAABB73DED2F9AC5B635002F32A5CFCFB2ECE8863911E828477FFD7
C49C4182CBAABB73DED2F9AC5B635002F32A5CFCFB2ECE8863911E828477FFD7
```

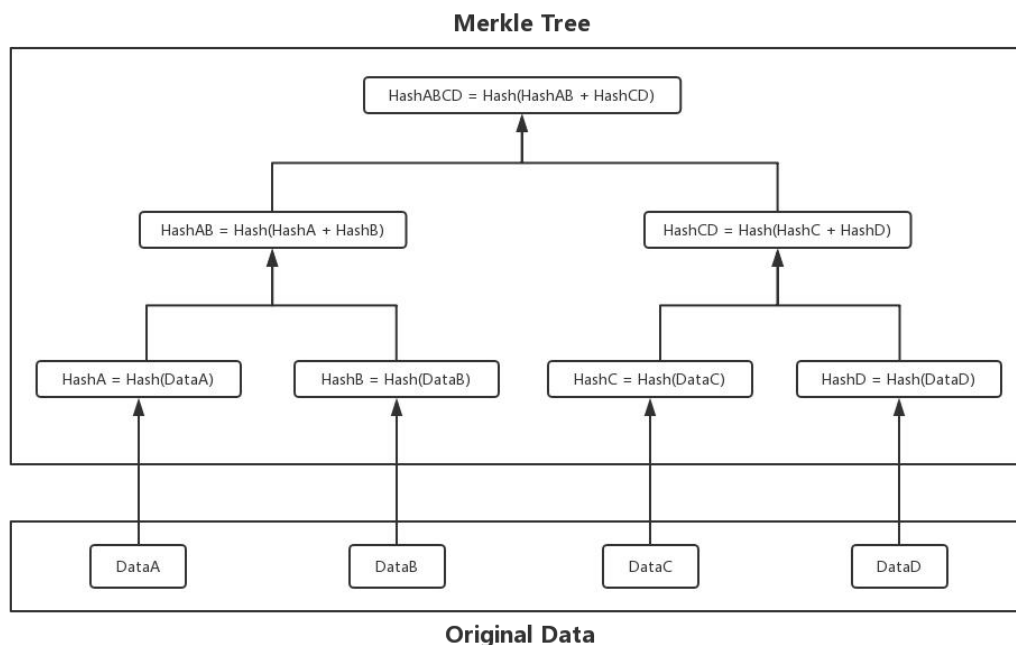
由此可验证，本人所写的 SHA256 算法是基本正确的

## 二、实验说明 SHA256 如何用于区块数据锁定

### 2.1 实验原理

区块链利用 Merkle 树来确保区块数据的正确性和完整性

Merkle 树原理如下图所示，其中的 Hash 算法即为 SHA256 算法



由此图不难看出，一个区块的数据改变，同时需要改变从该区块节点到 Merkle 树根节点的路径上的所有节点的值，由此确保了区块数据的正确性和完整。

同时，利用 Merkle 树的特性，我们可以快速对一个区块的数据进行校验：

例：现有区块 DataB 的值为“Data B”，我们只需要获取 HashA、HashCD 的值，运

算得到 HashABCD 的值与真实值进行比较，即可校验该区块的数据

## 2.2 实验设计

### (1) 构造 Merkle 树

代码：

```
var merkleTree = new MerkleTree(valueList);
```

其中，valueList 为各区块的数据列表（现仅支持 2 的幂次）

为了简化实验，样例程序设置 valueList 的值为{"Data1", "Data2", "Data3", "Data4"}

### (2) 输入区块号和区块数据，验证区块数据

代码：

```
var res = merkleTree.Verify(index, data)
```

Index 为区块号，data 为待校验的区块数据，返回 true/false

## 2.3 样例程序如何运行？

项目 **MerkleTreeTest** 编译运行后，以（区块号 待校验数据）的格式输入，程序输出校验结果

默认的原始数据内容为{"Data1", "Data2", "Data3", "Data4"}，区块号 0-3

## 2.4 运行结果

```
1 Data2
True
```

区块 1 的数据为 Data2，因此输出 True

```
0 Data0
False
```

区块 0 的数据为 Data1，因此输出 False

## 三、实验说明 SHA256 在 PoW 中的作用

### 3.1 实验原理

在区块链中，区块头的值可简单地为：

Header = SHA256(SHA256(上一个 Header + 随机数 nonce))

其中 Header 有效的标准为前 n 位为 0，这就是 POW（Proof of Work）

显然，SHA256 用于计算 POWHash，且 n 越大，计算难度更高；且如果要伪造交易，需要重新进行大量的哈希计算来找到一个有效的 nonce 随机数，因此修改比特币中的交易信息是计算上不可能的

### 3.2 实验设计

#### (1) 设置初始 header

样例程序将初始的 headerData 设置为“Header Data”，初始的 header 由 SHA256 计算而来

#### (2) 设置 n

设置 POWHash 有效标准 n，即前 n 位需要为 0

#### (3) 循环计算有效的 POWHash

样例程序将默认执行 5 次 POW 运算，即依次计算 5 次 POWHash

### 3.3 如何运行样例程序？

项目 **POWTest** 编译运行后，输入 POWHash 有效标准  $n$ ，程序会进行 5 次 POW 运算并输出 POWHash（当  $n$  较大时会非常缓慢）

### 3.4 运行结果

```
5
63D0651D217C16201E45D2DB6EE79503BF331D4DBE959910A8300084F7922573
00000B43C70F5B23561AD79F992C6AAC23ADB210D689FE966026AF22AE796078
00000FBF616DB840F66790F3AF410BFFD3C3B38D9C679ACFE74A4CE55E9B12B2
00000761596FC7E8BCB518B7B310CA799CC68F6156069DB5CBEA8AB26C2A4CDB
00000FBA51E188CC5F00B9B12E961C8588FD341909661AABE67A95D7E08E6659
00000E8F38ECCA4AE8C06D2D9754A7B68A1200FC147506FDCDC30ED02196AF4C
```

$n$  为 5 以内时，程序能较快地计算出结果

当  $n \geq 6$  时，程序计算较为缓慢