作业2
一 设计加密猫
二数据结构
三存储定义
四可调用函数
五算法伪代码
要满足需求：
1.链上存储加密猫数据
2.遍历所有加密猫
3.每只猫都有自己的dna，为128bit的数据
4.设计如何生成dna（伪代码算法）
5.每个用户可以拥有零到多只猫
6.每只猫只有一个主人
7.遍历用户拥有的所有猫

一 设计加密猫
功能设计:
养猫，且每一只猫都是唯一的不可替代的。用户可以创建猫(唯一 ERC721)。用户可以给猫定价。买卖猫。猫与猫生小猫
二数据结构
哪些数据是需要存储在链上，以及以何种形式存储在链上呢？
Kitty类的设计
所有的猫相关数据结构设计
用户的猫相关数据结构设计
我们的业务中一共需要用到3 substrate的特殊类型：
AccountId
Balance
Hash

然后我们思考该以什么样的形式存储数据？
```
//我们既然是针对猫的业务，那我们需要定义一个猫的数据结构
#[derive(Encode, Decode, Default, Clone, PartialEq)]
#[cfg_attr(feature = "std", derive(Debug))]
pub struct Kitty<Hash, Balance> {
    id: Hash,//业务上的唯一id
    dna: Hash,//3.每只猫都有自己的dna，为128bit的数据
    //Hash的细节看五算法伪代码
    price: Balance,//猫的价格
    gen: u64,//已经是第几代
}
```

存储模块

```
decl_storage! {
    trait Store for Module<T: Trait> as KittyStorage {
```

```
        //  2.遍历所有加密猫
        //所有的猫，map结构，  业务上猫的id => 猫
        Kitties get(kitty): map T::Hash => Kitty<T::Hash, T::Balance>;
        //猫的所有者是谁 map结构 业务上猫的id => 区块链账户地址
        KittyOwner get(owner_of): map T::Hash => Option<T::AccountId>;

        //所有的猫 map结构来替代数组，  下标 => 业务上猫的id
        AllKittiesArray get(kitty_by_index): map u64 => T::Hash;
        //所有的猫的总数
        AllKittiesCount get(all_kitties_count): u64;
        //获得当前猫是第几只猫 业务上猫的id => 猫的总数的位置（下标）
        AllKittiesIndex: map T::Hash => u64;

        // 5.每个用户可以拥有零到多只猫
        // 7.遍历用户拥有的所有猫
        //用户拥有的猫 map结构来代替数组, (账户地址,下标) => 业务上猫的id
        OwnedKittiesArray get(kitty_of_owner_by_index): map (T::AccountId, u64) => T::Hash;
        //用户拥有的猫的总数 map结构, 账户地址 => 总数
        OwnedKittiesCount get(owned_kitty_count): map T::AccountId => u64;
        //用户的这只猫是用户的第几只猫 业务上猫的id => index
        OwnedKittiesIndex: map T::Hash => u64;

        //随机数
        Nonce: u64;
    }
}
```

三存储定义

存储结构定义

开始定义存储模块

1.链上存储加密猫数据

```
//定义需要存储于链上的数据
decl_storage! {
    //我们定义一个存储结构叫KittyStorage
    trait Store for Module<T: Trait> as KittyStorage {
        //6.每只猫只有一个主人
        //用户拥有的猫，map结构，用户地址 => 猫
        OwnedKitty: map T::AccountId => Kitty<T::Hash, T::Balance>;
    }
}
```

四可调用函数

功能考虑

创建一只猫：create_kitty()

给猫定价：set_price()

转让猫的所有权：transfer()

买猫：buy_kitty()

生小猫：breed_kitty()

```
decl_event!(
    pub enum Event<T>
    where
        <T as system::Trait>::AccountId,
        <T as system::Trait>::Hash,
        <T as balances::Trait>::Balance
    {
        Created(AccountId, Hash),//创建猫事件
        PriceSet(AccountId, Hash, Balance),//定价事件
        Transferred(AccountId, AccountId, Hash),//转让事件
        Bought(AccountId, AccountId, Hash, Balance),//购买事件
    }
);
```

五算法伪代码

4.设计如何生成dna（伪代码算法）

3.每只猫都有自己的dna，为128bit的数据

```
//随机hash生成
  let nonce = <Nonce<T>>::get();
  let random_seed = <system::Module<T>>::random_seed();
  let random_hash = (<system::Module<T>>::random_seed(), &sender, nonce)
      .using_encoded(<T as system::Trait>::Hashing::hash);
<Nonce<T>>::mutate(|n| *n += 1);
//random_hash为128bit的数据
```

很多128bit output的 hash function 可以使用

E.g. MD5 hash:

The 128-bit (16-byte) MD5 hashes (also termed *message digests*) are typically represented as a sequence of 32 hexadecimal digits. The following demonstrates a 43-byte ASCII input and the corresponding MD5 hash:

Pseudocode

The MD5 hash is calculated according to this algorithm. All values are in little-endian.

```
//Note: All variables are unsigned 32 bit and wrap modulo 2^32 when
calculating
var int[64] s, K
var int i
```

```
//s specifies the per-round shift amounts
s[ 0..15] := { 7, 12, 17, 22,  7, 12, 17, 22,  7, 12, 17, 22,  7, 12,
17, 22 }
s[16..31] := { 5,  9, 14, 20,  5,  9, 14, 20,  5,  9, 14, 20,  5,  9,
14, 20 }
s[32..47] := { 4, 11, 16, 23,  4, 11, 16, 23,  4, 11, 16, 23,  4, 11,
16, 23 }
s[48..63] := { 6, 10, 15, 21,  6, 10, 15, 21,  6, 10, 15, 21,  6, 10,
15, 21 }

//Use binary integer part of the sines of integers (Radians) as
constants:
for i from 0 to 63
    K[i] := floor(2^32 × abs(sin(i + 1)))
end for
//(Or just use the following precomputed table):
K[ 0.. 3] := { 0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee }
K[ 4.. 7] := { 0xf57c0faf, 0x4787c62a, 0xa8304613, 0xfd469501 }
K[ 8..11] := { 0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be }
K[12..15] := { 0x6b901122, 0xfd987193, 0xa679438e, 0x49b40821 }
K[16..19] := { 0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa }
K[20..23] := { 0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8 }
K[24..27] := { 0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed }
K[28..31] := { 0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a }
K[32..35] := { 0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c }
K[36..39] := { 0xa4beea44, 0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70 }
K[40..43] := { 0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05 }
K[44..47] := { 0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665 }
K[48..51] := { 0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039 }
K[52..55] := { 0x655b59c3, 0x8f0ccc92, 0xffeff47d, 0x85845dd1 }
K[56..59] := { 0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1 }
K[60..63] := { 0xf7537e82, 0xbd3af235, 0x2ad7d2bb, 0xeb86d391 }

//Initialize variables:
var int a0 := 0x67452301   //A
var int b0 := 0xefcdab89   //B
var int c0 := 0x98badcfe   //C
var int d0 := 0x10325476   //D

//Pre-processing: adding a single 1 bit
append "1" bit to message
// Notice: the input bytes are considered as bits strings,
//  where the first bit is the most significant bit of the byte.[50]

//Pre-processing: padding with zeros
append "0" bit until message length in bits ≡ 448 (mod 512)
```

```
append original length in bits mod 2^64 to message

//Process the message in successive 512-bit chunks:
for each 512-bit chunk of padded message
    break chunk into sixteen 32-bit words M[j], 0 ≤ j ≤ 15
    //Initialize hash value for this chunk:
    var int A := a0
    var int B := b0
    var int C := c0
    var int D := d0
    //Main loop:
    for i from 0 to 63
        var int F, g
        if 0 ≤ i ≤ 15 then
            F := (B and C) or ((not B) and D)
            g := i
        else if 16 ≤ i ≤ 31 then
            F := (D and B) or ((not D) and C)
            g := (5×i + 1) mod 16
        else if 32 ≤ i ≤ 47 then
            F := B xor C xor D
            g := (3×i + 5) mod 16
        else if 48 ≤ i ≤ 63 then
            F := C xor (B or (not D))
            g := (7×i) mod 16
        //Be wary of the below definitions of a,b,c,d
        F := F + A + K[i] + M[g] //M[g] must be a 32-bits block
        A := D
        D := C
        C := B
        B := B + leftrotate(F, s[i])
    end for
    //Add this chunk's hash to result so far:
    a0 := a0 + A
    b0 := b0 + B
    c0 := c0 + C
    d0 := d0 + D
end for

var char digest[16] := a0 append b0 append c0 append d0 //(Output is in
little-endian)

//leftrotate function definition
leftrotate (x, c)
    return (x << c) binary or (x >> (32-c));
```

Note: Instead of the formulation from the original RFC 1321 shown, the following may be used for improved efficiency (useful if assembly language is being used – otherwise, the compiler will *generally optimize the above code. Since each computation is dependent on another in these formulations, this is often slower than the above method where the nand/and can be parallelised):*

```
( 0 ≤ i ≤ 15): F := D xor (B and (C xor D))
(16 ≤ i ≤ 31): F := C xor (D and (B xor C))
```