Gas可以说是以太坊生态系统的命脉，任何涉及到以太坊的操作，无论是交易还是智能合约等等，都需要一定量的gas，gas是衡量执行某些操作所需要的计算量单位。随着生态系统的不断发展，网络使用的逐渐增加，对gas的消耗也随之增加，对gas的优化需求也同步增长。今天说的就是优化以太坊gas的消耗。

节能模式（Gas-Saving Patterns）

首先可以在代码中使用下列的模式来减少对gas的消耗。

短路径（Short-circuiting）

这是一种在使用II或者&&时可以使用的策略。该模式通过较低成本执行指令起作用，因此，如果第一个操作的评估结果是true，那么较高成本的那个操作可能会被直接跳过。

堆栈交换示例详情如下：

// f(x) is low cost

// g(y) is expensive

// Ordering should go as follows

f(x) || g(y)

f(x) && g(y)

无用的文库（Unnecessary libraries）

Libraries通常情况下只有对少部分用途进行输入，这意味着对于你的合约来说，其中可能包含大量多余的代码。如果能够安全有效的实现从Libraries到合约的导入功能，就尽可能这么做。

包含冗余的Library：

```
import './SafeMath.sol' as SafeMath;

contract SafeAddition {

  function safeAdd(uint a, uint b) public pure returns(uint)
 {

    return SafeMath.add(a, b);

  }

}
```

不含冗余的Library：

```
contract SafeAddition {

  function safeAdd(uint a, uint b) public pure returns(uint)
 {

    uint c = a + b;

    require(c >= a, "Addition overflow");

    return c;
```

```
  }

}
```

明确的功能可见性

明确的功能可见性通常情况下会为智能合约的安全性和gas的优化方面带来好处。例如，显示标记外部功能会强制将功能参数存储位置设置为calldata，从而在每次功能被执行时节省时间。

正确选择数据类型

在Solidity中，某些数据类型要比其它的更加昂贵，因此，能够选择出正确的数据类型，使其效果最大化是极为重要的。

以下是一些有关数据类型的选择规则：

- 尽可能使用uint类型代替string类型；

- 与uint8相比，uint256需要更少的gas；

- 如果能够限制字节的长度，使用从字节1到字节32的最小数量；

- 字节32类型要比string类型更廉价。

Gas-Costly Patterns

接下来提到的模式都是些反面例子，它们会增加gas的消耗，因此应该尽可能避免使用。

无效代码（Dead Code）

无效代码指的是永远不会运行的代码，因为它的评估基于始终返回false的条件。

例：如果x < 1，则不能 > 2，因此在下面的例子中，永远不会执行第4行。

```
function deadCode(uint x) public pure {

  if(x
```

```
    if(x > 2) {
      return x;
    }
  }
}
```

Opaque Predicate

某些条件的结果无需执行即可知道，因此不需要评估。

例：如果x > 1，则肯定 > 0，因此下面的例子中，第三行是多余的。

```
function opaquePredicate(uint x) public pure {
  if(x > 1) {
    if(x > 0) {
      return x;
    }
  }
}
```

循环中的昂贵操作

由于SLOAD和SSTORE操作码十分昂贵，管理storage里的变量代价要比管理memory中变量的代价高昂的多，因此，不应该在循环中使用storage 变量.

例：循环的每次迭代都需要昂贵的storage管理。

```
uint num = 0;
```

```
function expensiveLoop(uint x) public {

  for(uint i = 0; i

    num += 1;

  }

}
```

对于此模式的解决方法是，创建一个代表全局变量的临时变量，并在完成循环后，将临时变量的值重新分配给全局变量。

如下：

```
uint num = 0;

function lessExpensiveLoop(uint x) public {

  uint temp = num;

  for(uint i = 0; i

    temp += 1;

  }

  num = temp;
```

循环的持续结果

如果循环的结果是可以在编译期间推断的常数，就不要用它。

返回值将会始终相同：

```
function constantOutcome public pure returns(uint) {

  uint num = 0;
```

```
  for(uint i = 0; i

    num += 1;

  }

  return num;

}
```

循环融合

在智能合约中，有时候会存在两个具有相同参数的循环。在循环参数相同的情况下，没有理由使用单独的循环。

这些循环相同，因此可以组合：

```
function loopFusion(uint x, uint y) public pure returns(uint
) {

  for(uint i = 0; i

    x += 1;

  }

  for(uint i = 0; i

    y += 1;

  }

  return x + y;

}
```

循环中重复计算

如果循环中的表达式在每次迭代中产生相同的结果，则可以将其移除循环。尤其是

当表达式中使用的变量被存储在storage中时。

下列中，a*b就可以被移除循环。

```
uint a = 4;

uint b = 5;

function repeatedComputations(uint x) public returns(uint) {

  uint sum = 0;

  for(uint i = 0; i

    sum = sum + a * b;

  }

}
```

循环中的单边结果比较

如果在循环里每次迭代中执行比较的结果都是相同的，就应该将其从循环中移除。

例：下列中，第4行的条件每次结果都相同，因此应该将其从循环中移除。

```
function unilateralOutcome(uint x) public returns(uint) {

  uint sum = 0;

  for(uint i = 0; i

    if(x > 1) {

      sum += 1;

    }

  }
```

```
  return sum;

}
```

以上就是一些在以太坊中会加大gas消耗的模式，以及如何削减以太坊中对gas的消耗方法，只要避开那些代价高昂的模式，选择那些低耗能的模式，相信就可以降低在以太坊上行动时的相应成本。

踢马河：RaTiO Fintech合伙人，曾任某券商自营操盘手，十余年海外对冲基金和国内大型投资机构基金经理，资深交易建模专家，币圈大咖。

请尊重原创！转载请注明出处。