Cairo简介（不是Cairo 1.0）

一、Field 元素（ Field elements ）

现代的 CPU 的基本数据类型是 *64* 位整数，也就是说我们的数据都要用 *2^64* 取模。

例如：

```
-17 代表 2^64 -17 即 -17 === 2^64 -17
```

```
2^63 * 2 = 2^64 = 0
```

而在 Cairo 里，基本的数据类型也是整数，但它的范围是 *0 <= x <= P* , 其中 *P* 是一个质数（通常情况下，*P = 2^251 + 17 * 2^192 + 1* ）, 同理，所有 Cairo 的计算都需要用 *P* 取模。

这意味着 Cairo 和传统规则还是有一定差异的，但通常情况下，这个差异并没有影响：如果你只是想写一个程序来 "**循环计算 1 到 1000 的和**" ，这并不会导致数值溢出，也就是和超过 *P* ，所以结果一般是正常的。

但在特殊情况下，我们还是要避免 Cairo 的计算导致数值溢出。有几个场景我们要加以小心：

1. 除法：常用 CPU 计算中，整数除法通常是这样的:

   ```
   7 / 2 = 3 结果自动取整
   这会导致 3 * 2 != 7
   也就是乘回来不相等
   ```

   但在 Cairo 中，*(x / y) * y == x*

   也就是说，*x* 如果不能被 *y* 整除，那么 *x / y* 会等于一个让人觉得奇怪的数 *k*（可能有几十位那么长！），这个数 *k* 满足 *k * y == x* 。

   举个较难理解的例子：*2 * (P + 1) / 2 = P + 1 === 1 mod P*

   也就是说，*1 / 2* 在 Cairo 中，不是 *0.5* 也不是取整的 *0* 或者 *1* ，而是 *(P+1)/2*，其中 *P* 的取值在上面有提到，是一个非常大的数。

2. 偶数：通常情况下，整数乘法 *x * 2 = y* 则 *y* 一般为偶数，但在上一点中，

   *2 * (P + 1) / 2 = P + 1 === 1 mod P*

   乘 *2* 得到的结果并非偶数，所以这也是 Cairo 编程中需要注意的点。

二、非确定性计算（ Nondeterministic computation ）

假设我们想证明 Cairo 的某个计算结果是正确的，我们有一些捷径可以使用。例如，我们要证明 *x = 961* 时，*x* 的平方根 *y* 在 ***1到 100*** 之间。比较直接方法是写一个复杂的代码先把 *x* 平方根 *y* 算出来，然后验证 *y* 的范围是否在 ***1到100*** 之间。但在 Cairo 中我们有更便捷的方式，我们可以假设 *y = 31* ，然后求平方得到 *961* ，刚好符合 *x* 的值，然后再验证 y 的范围在 ***1到100*** 之间。

这两种方式的差异在于，前一种确定性计算，只要算平方根的代码写的正确，我们就能在确定的复杂度下面算出 *x* 的平方根，进而验证范围。而后一种为非确定性计算，需要去做假设，反方向求解。

这里顶多是概念上的引入，具体的非确定性计算案例还需后续章节进一步解释，下面是非确定性计算的伪代码：

```
对于求解上述问题：

1. 魔术般地猜出 y 的值（怎么个”魔术“法，有待后续章节讲解）
2. 计算 y^2 且确保结果等于 x
3. 验证 y 的值在 1到100 的范围内
```

练习：

```
1. 写一个非确定性伪代码证明 x^7 + x + 18 = 0 有解
2. 再写一个非确定性伪代码证明上述等式有两种以上的解法
```

三、内存模型

Cairo 支持 **只读的非确定性内存** （真拗口）。

例如，我们用 *[x]* 代表位于 *x* 位置的内存值。如果我们在程序的开头 *assert [0] == 7*，则后续 *[0]* 的值将一直为 *7* 。*assert* 有点像 *solidity* 中的 *assert*，但仍有区别，下面会讲。

简单点理解就是：Cairo 中的内存只能写一次，写完就不能改了。所以对于 *assert [0] == 7*，会有两种解释：

```
1. 如果 [0] 还是空内存，没有写过，就写入 [0] == 7，并自动通过 assert
2. 如果 [0] 已经被写过，则读出值，判断是否等于 7 ，不是的话就回退
```

四、寄存器

可能发生变化的值都存在指定的寄存器里：

1. *ap (allocation pointer)*：指向一个尚未使用的内存单元。
2. *fp (frame pointer)*：指向当前函数的起始点。函数里的参数或者本地变量的地址都是参照 *fp* 这个起始点来算的。当一个函数刚开始运行，*fp == ap* ，但 *ap* 是可能变化的，而 *fp* 在函数的整个作用域内都是不变的。
3. *pc (program counter)*：程序计数器，指向当前要运行的指令。

五、基础指令

```
[ ap ] = [ ap - 1 ] * [ fp ], ap++;
```

以上可以拗口地表示：“*[ap - 1]* **和** *[fp]* **相乘的值必须和这个未使用的内存** *[ap]* **一致**“

但说人话就是：我们把 *[ap - 1]* **和** *[fp]* **相乘的值写到** *[ap]* 里。

其中 *ap++* 代表 *ap* 指向的内存前进一个单位，*ap++* 不能单独拿出来写，它是整个指令的一部分。

下面是我们在 Cairo 中能见到的 **断言式指令** ：

```
[fp - 1] = [ap - 2] + [fp + 4];
[ap - 1] = [fp + 10] * [ap], ap++;
[ap - 1] = [fp + 10] + 12345, ap++;  // See (1) below.
[fp + 2] = [ap + 5];
[fp + 2] = 12345;
[ap + 2] = [[ap + 5]];  // See (2) below.
[ap] = [fp - 3] - [ap + 4];  // See (3) below.
[ap] = [fp - 3] / [ap + 4];  // See (3) below.
```

1. 有两种整数会出现在指令中：
    - 立即数（Immediates）：可以作为第二个操作数使用，例如 *[ap - 1] = [fp + 10] + 12345, ap++;* 中的 *12345*。当然也可以用于单独的赋值，如 *[fp + 2] = 12345;*
    - 偏移量（Offsets）：出现在方括号中，如 *[ap + 5]*

    立即数可以是任何的 ***field*** 变量，但偏移量只能在 *-2^15<=x<=2^15* 之内
2. *[ap + 2] = [[ap + 5]];* 解了两次引用，第一次拿出 *[ap + 5]* 的值，然后用这个值作为地址再取出该地址的值。
3. 这两个指令是语法糖，它们将分别被以下指令替代：

    ```
    1. [fp - 3] = [ap] + [ap + 4];
    2. [fp - 3] = [ap] * [ap + 4];
    ```

练习：

写一个 Cairo 程序，计算如下式子：

```
x^3 + 23*x^2 + 45*x + 67, x = 100
```

1. 程序结束时，结果应该在 *[ap -1]* 这个位置
2. 在这个练习中，你需要假设 ***fp*** 寄存器是不变的，且初始化为 ***ap***

可以使用下面的模板：

```
func main() {
    [ap] = 100, ap++;
    // << Your code here >>

    ret;
}
```

3. ***x*** 的值不能写死

4. 最好能在 ***5*** 条指令内完成（模板内的指令不算）

5. 编译到 ***poly_compiled.json*** 文件中

6. 用如下命令运行：

```
cairo-run \
--program=poly_compiled.json --print_memory --print_info \
--trace_file=poly_trace.bin --memory_file=poly_memory.bin \
--relocate_prints
```

7. 查看结果（最后一个内存单元值应为：1234567）

六、连续内存

Cairo 中，程序访问的内存单元必须是连续的。例如，如果 7 和 9 位置上的内存都被访问了，那么 8 位置上的也必须要具备可访问性（至于先访问谁，这个不需要按顺序）。如果出现空隙，验证器会自动给空出的地址填上随机值，确保访问地址的连续性。但这并不是一个高效的做法，尽管它不会影响安全性。

练习：

1. 运行下述代码：

```
func main() {
    [ap] = 100;
    [ap + 2] = 200;
    ret;
}
```

解释为何上面会出现不高效的情况？验证器是如何填充空隙内存的？ **ret** 前写上一个0指令，使代码变成一个高效的代码。

2. 下面的代码有什么毛病？

```
func main() {
    [ap] = 300;
    [ap + 10000000000] = 400;
    ret;
}
```