密码学 zk 系列

第 13 课：zkStark 证明系统

lynndell 博士

新火科技 密码学专家 lynndell2010@gmail.com

目录

1.Merkle 承诺

多项式的值为 a,b,c,d （ $a=f(0),b=f(1),c=f(2),d=f(3)$ ）。

哈希值组成 Merkle 树的叶子节点（图中的 4,5,6,7 节点），其他非叶子节点都是它对应的两个子节点拼接后的哈希（如 $Hash(\#2)=Hash\big(Hash(\#4),Hash(\#5)\big)$ 。

根节点称为 Merkle 根 $root$ 。

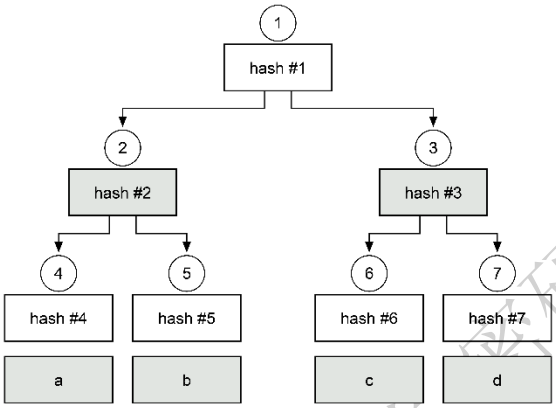


图 1. Merkle 树

Merkle 树的特点是只要有任意一个叶子节点有变化，则 Merkle 根会发生变化。**反之，只要能够证明 Merkle 根保持不变，那么整棵树都保持不变，也就意味着叶子节点承诺的值没有被修改。**

多项式值的 *Merkle* 承诺：

- **承诺：**基于多项式的值（例如上图的 4 个节点）生成一棵 Merkle 树的 root。
- **打开承诺：**基于 $root(f)$ 计算随机点（如节点 c ）。

  发送数据为 $\big(root(f),c,path(c)\big)$ ，其中 $path(c)$ 为 hash#7 和 hash#2，称为节点 c 的**验证路径**；
- **验证：**根据 $\big(c,path(c)\big)$ 计算 Merkle 根 $root'$ ，

$$root' := Merkle\big(c,path(c)\big)$$

  与接收到的 Merkle 根进行一致性校验

$$root' = root$$

确保 c 在 Merkle 树上。

2.FRI 多项式低阶检测

2.1 直接测试

已知常量多项式 $f(x)=c$，在固定点 z_1 和基于 Fait-Shamir 计算随机点 w，验证方检测

$$f(z_1)=f(w)$$

则接受多项式 $f(x)$ 的阶小于 1。

已知线性多项式 $f(x)=bx+c$，在固定点 $(z_1,f(z_1)),(z_2,f(z_2))$ 和基于 Fait-Shamir 计算随机点 w，**验证方**检测第三个点在同一直线上

$$(z_1,f(z_1)),(z_2,f(z_2)),(w,f(w))$$

则接受多项式 $f(x)$ 的阶小于 2.

一般化： d 阶**以内**的多项式 $f(x)$ 需要 d 个固定点和 1 个随机点进行检测。

2.2 组合测试

有两个 d 阶以内的两个多项式 $f(x),g(x)$，使用上述**直接测试**，则需要 $2d+2$ 个点进行进行测试。

组合测试：使用 Fait-Shamir 计算随机数 α，如下合并多项式 $f(x),g(x)$

$$h(x):=f(x)+\alpha\cdot g(x)$$ **线性组合**

证明方基于 $h(x)$ 的值构造 Merkle 树，将 Merkle 根发送给验证方，则验证方验证 Merkle 树并仅需要使用 $d+1$ 个点，则能够验证 $h(x)$ 的阶**小于** d 。

注意： $h(x)$ 的阶为 $\max\{\deg(f),\deg(g)\}$。

如果其中一个多项式的阶更小，则乘以随机多项式，多项式的阶补上去。

2.3 多项式折半

已知阶为 $d=2^3$ 的多项式

$$f(x)=a_0+a_1x+a_2x^2+a_3x^3+a_4x^4+a_5x^5+a_6x^6+a_7x^7+a_8x^8$$

$f(x)$ 的**偶数项**和**奇数项**分别为

$$g(x^2) = a_0 + a_2x^2 + a_4x^4 + a_6x^6 + a_8x^8$$
$$x \cdot h(x^2) = a_1x + a_3x^3 + a_5x^5 + a_7x^7 + 0x^9$$
$$= x \cdot (a_1 + a_3x^2 + a_5x^4 + a_7x^6 + 0x^8)$$

则

$$f(x) = g(x^2) + x \cdot h(x^2)$$

令 $y = x^2$，则

$$g(y) = a_0 + a_2y + a_4y^2 + a_6y^3 + a_8y^4$$
$$h(y) = a_1 + a_3y + a_5y^2 + a_7y^3 + 0y^4$$

符号修改：

$$g(x) = a_0 + a_2x + a_4x^2 + a_6x^3 + a_8x^4$$
$$h(x) = a_1 + a_3x + a_5x^2 + a_7x^3 + 0x^4$$

因此，$g(x), h(x)$ 的阶为 $d/2$。

原 d 阶多项式 $f(x)$ 等于**新** d/2 阶偶数项多项式 $g(y)$ 与奇数项多项式 $h(y)$ 之和。
原 d 阶多项式 $f(x)$ 等于**新** d/2 阶偶数项多项式 $g(x)$ 与奇数项多项式 $h(x)$ 之和。

已知多项式 $f(x)$ 的**阶为** d，其中 $d = 2^n$。

令 $g(x^2)$ 为 $f(x)$ 的偶数项，$h(x^2)$ 为 $f(x)$ 的奇数项，则

$$f(x) = g(x^2) + x \cdot h(x^2)$$ **折半公式**

令 $y = x^2$，则对于 y，多项式 $g(y), h(y)$ 的**阶为** $d/2$，

$$f'(y) := g(y) + \alpha \cdot h(y)$$ **线性组合**

符号修改：

$$f'(x) := g(x) + \alpha \cdot h(x)$$

构造出对应的 d/2 阶多项式，**类似向量内积承诺的**折半响应**。**

2.4 FRI 原理

目标：检测多项式的阶为小于 d。

FRI 协议：快速 Reed-Solomon 近似交互式预言机证明（**F**ast **R**eed-Solomon **I**nteractive Oracle Proof of Proximity）。区块链等应用领域需要非交互式协议，所以基于 Fait-Shamir 计算随机数，而不是验证方提供随机数。

结合上述**直接测试、线性组合**和**折半公式**，仅需要 $O(\log d)$ 次，可实现对 d 阶多项式 $f_0(x)$ 的阶测试，其中 $d=2^n$。

d 阶多项式，折 $\log d$ 次，每次折半均线性组合，最终有一个常量多项式。

第 1 步

证明：将 d 阶多项式 $f_1(x)$ 的值进行 Merkle 承诺得到 Merkle 根 $root(f_1)$；

对 Merkle 根 $root(f_1)$ 计算哈希值，获得随机数 z

$$z := Hash\left(root(f_1)\right)$$

计算 z 的函数值 $f_1(z), f_1(-z)$；

发送数据为： $root(f_1), f_1(z), f_1(-z), path(f_1(z)), path(f_1(-z))$

验证：根据 **Merkle 根** $root(f_1)$ **和验证路径** $path(f_1(z))$，确保 $f_1(z)$ 在 Merkle 树 $root(f_1)$ 上

$$root(f_1) = Merkle\left(f_1(z), path(f_1(z))\right)$$

同理，能够对确保 $f_1(-z)$ 在 Merkle 树上

$$root(f_1) = Merkle\left(f_1(-z), path(f_1(-z))\right)$$

因此，确保 $f_1(z), f_1(-z)$ 在 Merkle 树 $root(f_1)$ 上。

第 2 步

证明：多项式折半， d 阶多项式 $f_1(x)$ 的**折半公式**表达如下

$$f_1(x) = g_1(x^2) + x \cdot h_1(x^2)$$ **折半公式**

根据随机数 z，计算哈希值，获得随机数 α_2

$$\alpha_2 := Hash(z) = Hash\left(Hash\left(root(f_1)\right)\right)$$

构造**线性组合** $d/2$ 多项式，且令 $y = x^2$

$$f_2(y) = g_1(y) + \alpha_2 \cdot h_1(y)$$ **线性组合**

符号修改：

$$f_2(x) = g_1(x) + \alpha_2 \cdot h_1(x)$$

对多项式 $f_2(x)$ 的值 Merkle 承诺得到 Merkle 根 $root(f_2)$；此处 $f_2(x)$ 多项式的阶为 $d/2$。

计算随机数 $-z^2$ 处的函数值 $f_2(-z^2)$；

发送数据为： $root(f_2), f_2(-z^2), path(f_2(z^2)), path(f_2(-z^2))$

注意：因为可以直接计算 $f_2(z^2)$，所以不需要发送多项式值 $f_2(z^2)$。

验证：（Fait-Shamir）对 Merkle 根 $root(f_1)$ 计算哈希值，获得随机数 z

$$z := Hash\big(root(f_1)\big)$$

根据第 1 步数据 $f_1(z), f_1(-z)$、随机数 z 和**折半公式**，可以得到以下方程

$$f_1(z) = g_1(z^2) + z \cdot h_1(z^2)$$
$$f_1(-z) = g_1(z^2) - z \cdot h_1(z^2)$$

所以能够解方程，获得 $g_1(z^2), h_1(z^2)$。

（Fait-Shamir）根据 $z, root(f_1)$，计算哈希值，获得随机数 α_2

$$\alpha_2 := Hash(z, root(f_2))$$

根据 $g_1(z^2), h_1(z^2)$、随机数 α_2 和**线性组合** $f_2(x) = g_1(x) + \alpha_2 \cdot h_1(x)$，如下计算

$$f_2(z^2) := g_1(z^2) + \alpha_2 \cdot h_1(z^2)$$

则得到 $f_2(z^2)$。

根据接收数据 $root(f_2), path(f_2(z^2))$，确保 $f_2(z^2)$ 在 Merkle 树 $root(f_2)$ 上

$$root(f_2) = Merkle\big(f_2(z^2), path(f_2(z^2))\big)$$

同理，确保 $f_2(-z^2)$ 在 Merkle 树 $root(f_2)$ 上

$$root(f_2) = Merkle\big(f_2(-z^2), path(f_2(-z^2))\big)$$

因此，确保 $f_2(z^2), f_2(-z^2)$ 在 Merkle 树 $root(f_2)$ 上。

第 3 步：

证明：根据**多项式折半**，$d/2$ 阶多项式 $f_2(x)$ 的**折半公式**表达如下

$$f_2(x) = g_2(x^2) + x \cdot h_2(x^2) \text{ 折半公式}$$

根据 $z, root(f_2)$，计算哈希值，获得随机数 α_3

$$\alpha_3 := Hash(z, root(f_2))$$

构造**线性组合**多项式，且令 $y = x^2$

$$f_3(y) = g_2(y) + \alpha_3 \cdot h_2(y) \text{ 线性组合}$$

符号修改：

$$f_3(x) = g_2(x) + \alpha_3 \cdot h_2(x)$$

对多项式 $f_3(x)$ 的值 Merkle 承诺得到 Merkle 根 $root(f_3)$；

此处 $f_3(x)$ 多项式的阶为 $d/2^2$。

计算随机数 $-z^4$ 处的函数值 $f_3(-z^4)$；

发送数据： $root(f_3), f_3(-z^4), path(f_3(z^4)), path(f_3(-z^4))$

注意：因为可以直接计算 $f_3(z^4)$，所以不需要发送多项式值 $f_3(z^4)$。

验证：根据第 1 步验证后的函数值 $f_2(z^2)$，随机数 z^2 和**折半公式**，得到方程

$$f_2(z^2) = g_2(z^4) + z^2 \cdot h_2(z^4)$$
$$f_2(-z^2) = g_2(-z^4) - z^2 \cdot h_2(-z^4)$$

所以能够解方程，获得 $g_2(z^4), h_2(z^4)$。

（Fait-Shamir）根据 $z, root(f_2)$，计算哈希值，获得随机数随机数 α_3

$$\alpha_3 := Hash(z, root(f_2))$$

根据 $g_2(z^4), h_2(z^4)$、随机数 α_3 和**线性组合** $f_3(x) = g_2(x) + \alpha_3 \cdot h_2(x)$，如下计算

$$f_3(z^4) := g_2(z^4) + \alpha_3 \cdot h_2(z^4)$$

则得到 $f_3(z^4)$。

根据 $root(f_3), path(f_3(z^4))$，确保 $f_3(z^4)$ 在 Merkle 树 $root(f_3)$ 上

$$root(f_3) = Merkle\left(f_3(z^4), path(f_3(z^4))\right)$$

同理，确保 $f_3(-z^4)$ 在 Merkle 树 $root(f_3)$ 上

$$root(f_3) = Merkle\left(f_3(-z^4), path(f_3(-z^4))\right)$$

因此，确保 $f_3(z^4), f_3(-z^4)$ 在 Merkle 树 $root(f_3)$ 上。

以此类推…

第 log(d)步：证明发送数据为

$root(f_{\log(d)}), f_{\log(d)}(-z^n), path(f_{\log(d)}(z^n)), path(f_{\log(d)}(-z^n))$

因为可以直接计算 $f_{\log_2 d}(z^n)$，所以不需要发送多项式值 $f_{\log_2 d}(z^n)$。

验证：计算出多项式的函数值 $f_{\log(d)}(z^n)$。

基于**多项式折半公式**和**线性组合**，则获得一个**多项式**

$$f_{\log(d)}(x) = g_{\log(d)}(x) + \alpha_{\log(d)} \cdot h_{\log(d)}(x)$$

该多项式的阶为 $d / 2^{\log(d)} = 1$，所以该多项式为**常量多项式**。
验证方使用**直接测试**完成常量多项式阶的测试。

注意：

- **对于上述协议的取值范围，需要确定范围 L 中的每个值 z，$-z$ 也在范围 L 中。**
- **$f_1(x)$ 多项式的承诺不是在范围 L 上，而是在范围 $L^2 = \{x^2 : x \in L\}$ 上。**

2.5 概率分析

目标：线性组合的正确性。

对于多项式的 2 个值 $f_0(z), f_0(-z)$。假设 $f_0(z) = f_0(-z) = 0$ 的概率为 ε，其中 $0 < \varepsilon < 1$；则其他概率为 $1-\varepsilon$，其中 $0 < 1-\varepsilon < 1$。

$$f(x) = g(x^2) + x \cdot h(x^2)$$ **折半公式**

如果进入$1-\varepsilon$概率，即 $f_0(z)=f_0(-z)\neq 0$，则存在唯一α_1，使得线性组合为零

$$f_1(z^2)=g_0(z^2)+\alpha_1\cdot h_0(z^2)=0$$ **线性组合**

但是，α_1是基于 Fait-Shamir 计算的随机数，操控α_1为期望值的概率等于 POW 困难。因此，$f_1(z^2)\neq 0$的概率接近 1。

如果证明方**选择随机数**作为 $f_1(z^2)$，则 Merkle 验证一致性不通过。因此，进入$1-\varepsilon$概率分布，则验证方拒绝。

反之，证明方某一轮作弊成功概率为ε，在$\log(d)$轮中，证明方全都成功的概率为$\varepsilon^{\log(d)}$呈指数小。因此，证明方作弊成功概率可忽略。

验证方如果认可是 d 阶多项式，则认可证明方的保密数据正确。

3.斐波纳契数列 Stark

3.1.零知识证明

对于一个 NP 问题F，证明方的**公开输入**X和**保密输入**Y，**公开输出**Z，证明：知道保密输入Y满足**计算关系**

$$F(X,Y)=Z$$

验证方接受该事实。

Sigma 零知识证明知道秘密ω，公开输入为 G，公开输出为 H，满足运算关系

$$H=\omega\cdot G$$

3.2.斐波纳契数列

证明方需要证明第 1000 个斐波纳契输出值Z的起始为一个公开输入X和一个**保密输入**Y，且$F(X,Y)=Z$。该过程等价于证明

$$F_0:=X,F_1:=Y$$
$$F_i:=F_{i-2}+F_{i-1}$$
$$Z:=F_{1000}$$

NP 问题

该计算需要n步和w个寄存器，迹T是一个$n\times w$的表格，其中$n=1000,w=2$。例如$X=3,Y=4$，则构造出如下表格

n	$T_{n,0}$	$T_{n,1}$
0	PubInput = 3	PrivateInput = 4
1	4	7
2	7	11
3	11	18
4	18	29
…	…	…
999	F_{999}	F_{1000}

上述算法编码为**转换约束***transition_constraints*

$$T_{i+1,0} = T_{i,1}$$
$$T_{i+1,1} = T_{i,0} + T_{i,1}$$

边界约束*boundary_constraints*

$$T_{0,0} = X$$
$$T_{1000,1} = Z$$

因此，得出以下核心结论 1：

- **证明方证明知道秘密值** Y **满足关系** $F(X,Y) = Z$ **，等价于**
- **证明方证明知道迹** T **满足转换约束** *transition_constraints* **和边界约束** *boundary_constraints*。

注释：
zkSTARK 中：基于秘密值 Y，计算出**迹 trace**。
Groth16 中：基于秘密 witness，计算出**向量** $\vec{s}$ 。

3.3.迹多项式

将 $n = 0,...,999$ 当作多项式的横坐标，将寄存器 $T_{n,0}$ 和寄存器 $T_{n,1}$ 中存储的迹值当作多项式的值，则构造出**多项式值表达**

$$P_0(i) = T_{i,0}, i = 0,...,999$$
$$P_1(i) = T_{i,1}, i = 0,...,999$$

则能够构造出阶为 999 的**多项式系数表达** $P_0(x), P_1(x)$。

如果使用快速傅里叶变换 FFT，则 i 的取值为 n 次单位根 ω；如果使用快速数论变换 NTT，则 i 的取值为模 p 原根 g。

将迹的多项式值表达等价转化为多项式的系数表达，则有以下**等价转化关系**：

斐波那契数列（加法三元组）：$a_n=a_{n-1}+a_{n-2}$

- **多项式值表达**：转换约束 $T_{i+1,1}=T_{i,0}+T_{i,1}, i=0,...,999$
- **多项式系数表达**：$P_1(i+1)=P_0(i)+P_1(i), i=0,...,999$
- 多项式系数表达：$P_1(i+1)-(P_0(i)+P_1(i))=0, i=0,...,999$

多项式系数表达：$Q(x):=P_1(x+1)-(P_0(x)+P_1(x))=0, x=0,...,999$，

$Q(x)$ 阶为 1000

构造公开的目标多项式

$$R(x)=(x-0)(x-1)(x-2)...(x-999)$$

如果 $x=0,...,999$，则目标多项式 $R(x)=0$。

多项式 $Q(x)=0$ 存在其他解 x'，因此，存在商多项式 $C(x)$ 是一个常量多项式

$$C(x)=\frac{Q(x)}{R(x)}$$

与 KZG 承诺中的商多项式的效果一样。要求商多项式存在，则整除成功，则 Q(x)多项式正确，则转换约束正确，则计算正确，则保密数据正确。
根据 Schwartz－Zippel 引理，攻击者作恶成功概率可忽略。

斐波那契数列（乘法三元组）：$a_n=a_{n-1}*a_{n-2}$

- **多项式值表达**：转换约束 $T_{i+1,1}=T_{i,0}\cdot T_{i,1}, i=0,...,999$
- **多项式系数表达**：$P_1(i+1)=P_0(i)\cdot P_1(i), i=0,...,999$
- 多项式系数表达：$P_1(i+1)-(P_0(i)\cdot P_1(i))=0, i=0,...,999$

多项式系数表达：$Q(x):=P_1(x+1)-(P_0(x)\cdot P_1(x))=0, x=0,...,999$，

$Q(x)$ 阶为 2000

多项式 $Q(x)=0$ 存在其他解 x'，因此，存在商多项式 $C(x)$ 是一个阶为 **1000** 的多项式

$$C(x)=\frac{Q(x)}{R(x)}$$

斐波那契数列（乘法四元组）： $a_n=a_{n-1}*a_{n-2}*a_{n-3}$

- **多项式值表达：** 转换约束 $T_{i+1,1}=T_{i,0}\cdot T_{i,1}\cdot T_{i,2}, i=0,...,999$
- **多项式系数表达：** $P_1(i+1)=P_0(i)\cdot P_1(i)\cdot P_2(i), i=0,...,999$
- 多项式系数表达： $P_1(i+1)-(P_0(i)\cdot P_1(i)\cdot P_2(i))=0, i=0,...,999$

多项式系数表达： $Q(x):=P_1(x+1)-(P_0(x)\cdot P_1(x)\cdot P_2(x))=0, x=0,...,999$

$Q(x)$ 阶为 3000

多项式 $Q(x)=0$ 存在其他解 x'，因此，存在商多项式 $C(x)$ 是一个阶为 **2000** 的多项式

$$C(x)=\frac{Q(x)}{R(x)}$$

因此，得出以下 **2** 个关键结论：

算法通常会涉及乘法，商多项式的阶通常会高于迹 **Trace** 多项式。
加法、减法、乘法、除法等任意运算状态变化，都能抽象 **Trace** 多项式、**Q(x)** 与目标多项式 **R(x)**满足整除关系，即存在商多项式 **C(x)**。

回到二元加法斐波那契数列，转换约束 *transition_constraints* 满足以下等价关系

- $T_{i+1,1}=T_{i,0}+T_{i,1}\Leftrightarrow C_0=\frac{Q(x)}{R(x)}=\frac{P_1(x+1)-(P_0(x)+P_1(x))}{\prod_{[0,...,998]}^{i}(x-i)}$
- $T_{i+1,0}=T_{i,1}\Leftrightarrow C_1=\frac{P_0(x+1)-P_1(x)}{\prod_{[0,...,998]}^{i}(x-i)}$

边界约束 *boundary_constraints* 满足以下等价关系

- $T_{0,0}=X\Leftrightarrow C_2(x)=\frac{P_0(x)-X}{x-0}$
- $T_{999,1}=Z\Leftrightarrow C_3(x)=\frac{P_1(x)-Z}{x-999}$

因此，得出以下核心结论 2：

- **证明方证明知道迹** T **满足转换约束** *transition_constraints* **和边界约束** *boundary_constraints*，**等价于**
- **证明方证明知道多项式** $P_0(x), P_1(x)$ **使得 4 个商多项式** $C_0(x), C_1(x), C_2(x), C_3(x)$ **存在**

反之，验证方认可 4 个商多项式 $C_0(x), C_1(x), C_2(x), C_3(x)$ **存在，则转换约束和边界约束成立，则认可证明方证明其知道秘密值** Y **满足关系** $F(X,Y)=Z$ **。**

要求商多项式存在，则整除成功，则 Q(x)多项式正确，则约束正确，则计算正确，则保密数据正确。

验证方如果认可是 d 阶多项式，则认可证明方的保密数据正确。

3.4.FRI 原理

使用 Fait-Shamir 变换计算随机数 $\alpha_0, \alpha_1, \alpha_2, \alpha_3$，**线性组合**转换约束多项式 $C_0(x), C_1(x)$ 和边界约束多项式 $C_2(x), C_3(x)$

$$C(x) = \alpha_0 \cdot C_0(x) + \alpha_1 \cdot C_1(x) + \alpha_2 \cdot C_2(x) + \alpha_3 \cdot C_3(x)$$

如果阶不等，需要乘以随机多项式，让阶相等。

利用**多项式折半**，

1. **证明方**给出多项式 $C(x)$ 的系数的 Merkle 根 *Root* 并证明其阶为 2^n；
2. **等价于证明方**通过 Fait-Shamir 变换的随机数 β_0，证明多项式 $C^0(x)$ 的系数的 Merkle 根 $Root_0$ 并证明其阶为 2^{n-1}；
3. **等价于证明方**通过 Fait-Shamir 变换计算的的随机数 β_1，证明多项式 $C^1(x)$ 的系数的 Merkle 根 $Root_1$ 并证明其阶为 2^{n-2}；
4. 以此类推，…
5. **等价于证明方**通过 Fait-Shamir 变换计算的的随机数 β_n，证明多项式 $C^n(x)$ 的系数的 Merkle 根 $Root_n$ 并证明其阶为 2^0；

6. **最后，验证方**验证：**折半过程、验证多项式$C^n(x)$的系数**的 Merkle 根 $Root_n$、验证线性组合多项式$C^n(x)$的阶为 1，**则等价于**验证线性组合多项式$C(x)$的系数的 Merkle 根 *Root* 并证明其阶为2^n，**则等价于**验证迹T满足转换约束 *transition_constraints* 和边界约束 *boundary_constraints*，**则等价于**验证 $F(X,Y)=Z$ 而不知道秘密 Y。

4.zkStark 进阶

4.1 算术化

算术化：将计算问题转换为有限域 F 上的多项式的代数问题。zk-STARK 对计算完整性（Computational Integrity，CI）声明，“输出结果 α 是程序 C 根据输入 x 经过 T 步执行所得”，进行证明的第一步是**算术化**。

zk-STARK 算术化的方法包括 2 **个重要方法，**
首先，构建程序 C 的代数中间表达(Algebraic Intermediate Representation, AIR)，**用 s 个多项式描述当前执行状态与下一步状态的转化约束；**
其次，为降低证明者的时间复杂度和空间复杂度，**将 s 个多项式线性组合为 1 个多项式，该方法称为 ALI（Algebraic Linking Interactive Oracle Proof）。**详细说明如下。

思路一样的：运算关系 转换为 多项式整除关系

4.1.1 AIR 转换

对计算完整性进一步深入理解下，实际上需要通过 AIR 将这一约束表达出来，即从输入 x 到输出 α 的程序执行过程中的**中间计算状态转换**，中间计算状态则是一堆寄存器数值。因此，给出 AIR 的定义，是一个低度多项式的集合

$$P=\{P_1(\vec{X},\vec{Y}),...,P_s(\vec{X},\vec{Y})\}$$

- 低度多项式的系数在域 F 内
- $\vec{X}=(X_1,...,X_w)$ 代表当前计算状态
- $\vec{Y}=(X_1,...,X_w)$ 代表下一步计算状态
- w 是证明系统所中某一个计算状态的变量数量
- P_i 代表转换关系的多项式
- 有一对解 $(\vec{x},\vec{y})$ 使得**转换关系**成立，**当且仅当** $(\vec{x},\vec{y})$ 是 P 的共同的解，即

$$P_1(\vec{x},\vec{y}) = \ldots = P_s(\vec{x},\vec{y})$$

- AIR 的多项式的阶是所有 $P_i(x)$多项式中的最大值；
- s 是所有约束的数量。

AIR 将程序 C 执行过程中寄存器数值前后变化关系转化为约束多项式，以便验证者能信任输入 x 到输出 y 的完整过程，而不是随便得出的结果或伪造输出结果。上述的计算状态具体包括执行轨迹。

4.1.2 ALI 协议

由于多项式的插值计算及验证计算需要耗费大量的计算资源，因此，**ALI 协议将 s 个多项式约束化简为单一约束，可以使用随机线性组合。**

4.2 执行轨迹（Execution trace 或 trace）

执行轨迹是由执行某个计算时，由**状态序列**构成。为执行某个程序 C，申请 W 个寄存器，执行 N 个步骤，则执行轨迹可以用一个 N×W 表描述。以斐波那契数列计算为例

```
fn fib(start: (Felt, Felt), n: usize) -> Felt {
    let mut t0 = start.0;
    let mut t1 = start.1;

    for _ in 0..(n - 1) {
        t1 = t0 + t1;
        core::mem::swap(&mut t0, &mut t1);
    }
    t1
}
```

使用 2 个寄存器，s_0、s_1 分别存放每一个中间状态下的数据。模拟 n =10 生成的执行轨迹是一个数组，包含 5 对中间状态数据(state[0],state[1])。这里的执行轨迹是由 prover 定义并生成的，源码如下：

```
pub fn build_trace(start: (Felt, Felt), n: usize) -> TraceTable<Felt> {

    let mut trace = TraceTable::new(2, n / 2);
    trace.fill(
        |state| {
            state[0] = start.0;
            state[1] = start.1;
        },
        |_, state| {
            state[0] += state[1];
            state[1] += state[0];
        },
    );

    trace
}
```

执行轨迹中的数据如下图所示

Double 斐波纳契数列

s_0	s_1
0	1
1	2

3	5
8	13
21	34
55	89

定义的执行轨迹只有 2 列，状态转换的具体约束条件如下：

- result[0] = next_state[0] - (current_state[0] + current_state[1]);
- result[1] = next_state[1] - (current_state[1] + next_state[0]);

每一次状态转换必须满足这 2 个约束：

- $s_{\{0, i+1\}} = s_{\{0, i\}} + s_{\{1, i\}}$
- $s_{\{1, i+1\}} = s_{\{1, i\}} + s_{\{0, i+1\}}$

相应的约束多项式的阶最大为 1。因此，创建 AIR 时，定义状态转换的约束多项式的度为 1。

```
let degrees = vec![
    TransitionConstraintDegree::new(1),
    TransitionConstraintDegree::new(1),
];
```

AIR 需要对输入、输出进行**边界约束**，参考如下：

```
vec![
    Assertion::single(0, 0, self.start.0),       // input contraint
    Assertion::single(1, 0, self.start.1),       // input contraint
    Assertion::single(0, last_step, self.end),   // output contraint
]
```

因此，构建执行轨迹，并根据执行轨迹构建对应的约束。

4.3 zkStark 证明系统

4.3.1 Stark 中多项式承诺

两种类型的多项式

从**算术化**的约束系统中，会得到两种类型的 witness，

第一是整个待证明程序的**执行轨迹** execution trace，对应**迹多项式**；

第二是在执行过程中需要满足的**约束条件** constraint，对应**商多项式**；

轨迹可以看成是一组寄存器的状态转换过程。以斐波那契数列的计算为例，要求 fib(10)，且用两个寄存器 s0,s1 来保存计算的中间状态，则在程序正确执行 5 步后，s0[5]就是我们需要的计算结果。

Double 斐波纳契数列

s_0	s_1
0	1
1	2
3	5
8	13
21	34
55	89

有了程序的执行轨迹，还需要额外的约束来保证执行轨迹确实是程序按照规定的计算方式来生成的。**约束分为两部分：**

- 在**边界**处，寄存器的状态必须和指定的寄存器状态一致。例如 s_0,s_1 的初始状态要满足 **$s_0[0]=0$ 且 $s_1[0]=1$**。另外，我们得到的计算结果必须是程序执行到第 5 步后 s_0 寄存器的状态，即 **result=$s_0[5]$**。
- 寄存器每一次的**状态转换**必须满足如下计算规则：
  - $s_0[i] + s_1[i] = s_0[i+1]$
  - $s_1[i] + s_0[i+1] = s_1[i+1]$

有了这些约束，就能够信任程序的执行结果，即 fib(10) = result = $s_0[5]$
接下来对执行迹和约束条件做多项式承诺。

4.3.1.1多项式承诺

在讲具体的实现之前，先了解 STARK 中是如何对一个多项式进行承诺的。多项式承诺需要满足如下使用场景：

- **设置：**能够生成特定的代数结构 G，以及⟨PK, VK⟩，用于承诺一个阶数小于 t 的多项式；for 循环对**电路**有影响，则对⟨PK, VK⟩，需要**递归零知识证明**。
- **承诺：**输出多项式 $\phi(x)$的承诺 C；
- **打开承诺：**对于验证者给定的随机数 i，证明者给出多项式在该点处的值 $\phi(i)$，以及关于该求值的一个证明。验证者能够利用之前的承诺 C 验证其求值是正确的。

在 KZG 多项式承诺方案中，证明者首先使用椭圆曲线点乘的方式，对多项式进行承诺，随后在打开时构造一个新的商多项式，使得验证者可以通过椭圆曲线配对，验证该商多项式正是原始的 $\phi(x)$在挑战点处构造出的多项式。

STARK 使用**默克尔树**进行多项式承诺，实现方式如下：

- **设置：**STARK 不需要额外的可信设置，但需要事先确定**哈希函数（为提高效率，会使用特殊的哈希函数，如 poseidon2 等）**；**没有 PK 和 VK，**因此，**for** 循环等约束不会对 **air** 没任何影响。
- **承诺：**首先求出多项式 $\phi(x)$ 在其求值域上所有单位根处的值，$\phi(\omega^0),\phi(\omega^1),\phi(\omega^i),...$，以这些值为叶子节点，组成一颗默克尔树，最后公开得到的默克尔树树根 root，是该多项式的承诺。
- **打开承诺：**验证者选择一些随机挑战点，证明者提供多项式在该点处的值，以及一条默克尔树路径。验证者检查该默克尔树路径合法。

4.3.1.2迹多项式（Trace Polynomial）的承诺

对于一个 STARK 寄存器的**执行轨迹**，可以将它的每一行看作是其对应的轨迹多项式在某个单位根处的取值。因此，对于示例中的 $s_0(x)$, $s_1(x)$：

$s_0(\omega^0)=0$	$s_1(\omega^0)=1$
$s_0(\omega^1)=1$	$s_1(\omega^1)=2$
$s_0(\omega^2)=3$	$s_1(\omega^2)=5$
$s_0(\omega^3)=8$	$s_1(\omega^3)=13$
$s_0(\omega^4)=21$	$s_1(\omega^4)=34$

$s_0(\omega^5)=55$	$s_1(\omega^5)=89$

Merkle 树要求叶子节点个数必须是 2 的幂次，需要补上程序继续按约束条件运行结果：

$s_0(\omega^6)=144$	$s_1(\omega^6)=233$
$s_0(\omega^7)=377$	$s_1(\omega^7)=610$

就可以为 $s_0(x),s_1(x)$分别生成默克尔树承诺。

注意：可以不填充，需要根据经验，每次选择合适数量的交易单，使得多项式的阶为 2^n。例如：首先测出 $k_1=100,k_2=200,k_3=500,...$个交易单生成的多项式的阶为 $2^{n_1},2^{n_2},2^{n_3}$。以后每次都打包 k_1,k_2,k_3 个交易单，则不需要填补数据。

为协议的安全性，还需要进行扩展。

4.3.1.3低度多项式扩展（Low-Degree Extension）

从程序的执行轨迹中，得到数个长度为 N 的轨迹多项式。**出于安全性的考虑，需要在一个更大的求值域上承诺该多项式，一般需要将求值域扩大 2^k 倍，扩大的倍数称为爆炸倍数（blowup factor），该求值域称为 LDE 域。**

将长度为 N 的轨迹多项式扩展到 2^kN，原理如下：

- 利用拉格朗日插值法，**求出该轨迹多项式的系数 $a_0,a_1,a_2,..,a_{n-1}$**（注意其单位根需要使用 2^kN 求值域上的单位根）；
- 然后在 2^kN 的求值域上用系数求出其他单位根处的多项式值。

利用低次多项式生成更大值域上单位根处值的方法称为低度多项式扩展。

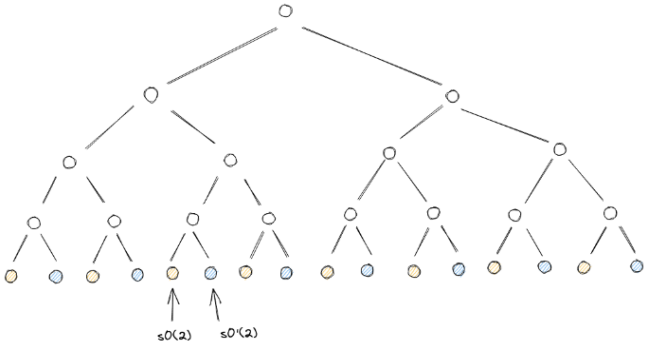
举例：多项式的值在 2000 内 $f_1,...,f_n$，在 1000 内计算**多项式系数表达** $a_1,...,a_n$

$$f(x)=a_0+a_1x+...+a_nx^n,x\in\{0,...,1000\}$$

扩大 x 的取值空间，变为 5000，能够计算更多的**多项式的值** $f_0,...,f_{5000}$，对这些值进行 Merkle 承诺。检测多项式的阶仍然应该小于 1000. **扩大倍数**就是**爆炸因子**。

关键点：扩展后，多项式的阶没发生变化。

下图是 blowup factor=2 时，$s_0(x)$的低度多项式扩展示例，黄色的节点是多项式在原始的求值域上对应单位根处的值（**它们也是在 LDE 域上所有单位根处值的一部分，**因为 FFT 性质 3 相消性：$\omega_{origin}^{i}=\omega_{LDE}^{2^k*i}$），蓝色为 LDE 扩展后新的单位根处的求值：



4.3.1.4约束多项式（ConstraintPolynomial）的承诺

程序在执行过程中需要满足的**约束条件**转换为多项式表示。例如，约束 $s_0(\omega^5)=55$，可以按如下的方式构造约束多项式：

$$c(x)=\frac{s_0(x)-55}{x-\omega^5}$$

根据多项式基本定理，$s_0(x)-55$ 可以被 $x-\omega^5$ 整除**当且仅当** $s_0(\omega^5)=55$。也就是说，当约束条件满足的时候，c(x)的次数小于 $s_0(x)$。利用这种方法，将所有的约束条件写成多项式的形式：

Double 斐波纳契数列

$$c_0(x)=\frac{s_0(x)-0}{x-\omega^0}$$

$$c_1(x)=\frac{s_1(x)-1}{x-\omega^0}$$

$$c_2(x)=\frac{s_0(x)-55}{x-\omega^5}$$

$$c_3(x)=\frac{s_0(\omega x)-s_0(x)-s_1(x)}{\prod_{i\in[0,6]}\left(x-\omega^i\right)}$$

$$c_4(x)=\frac{s_1(\omega x)-s_0(\omega x)-s_1(x)}{\prod_{i\in[0,6]}\left(x-\omega^i\right)}$$

前 2 个是**起始约束**，第 3 个是**输出约束**，最后 2 个是**转换约束**。

4.3.1.5约束多项式线性组合

如果检查每个约束多项式的次数，代价会很大，也没有必要。可以将所有的约束多项式组合到一起，形成一个**线性组合多项式**（Composition Polynomial）。根据 Schwartz–Zippel 引理，对该线性组合多项式的检查等价于对所有约束多项式的检查。

在线性组合之前，需要注意：**上述的约束多项式的次数不是相同的**。需要先将约束多项式补齐到相同的次数，然后线性再组合。

补齐方法：乘上一个随机的 $d-d_j$ 次多项式。该随机多项式的系数是验证者在证明者承诺完轨迹多项式后随机产生的

$$CP(x)=\sum_{j=0}^{4}c_j(x)(\alpha\cdot x^{D-D_j}-\beta_j)$$

最后得到**线性组合多项式** CP(x)，需要在 LDE 域上进行多项式承诺，**因此，像之前的迹多项式那样，使用低度多项式扩展的方式进行多项式承诺。**

4.3.1.6 DEEP-FRI

得到**迹多项式** $s_i(x)$ 和**线性组合多项式** $CP(x)$ 后，其实已经可以进行证明了。**只要把轨迹多项式和线性组合多项式再做一次随机的线性组合，然后证明最终的多项式度数小于 D，就能证明所有的约束条件满足，且程序的执行过程和被承诺的 Trace 一致。**

但是，从安全性角度出发，还需要做一些调整。STARK 使用称为 Domain Extension for Eliminating Pretenders (DEEP) 的方法进行检查。

从基域上随机选一个点，然后检查在这个点上，各多项式的值是否满足约束。**基域是自变量 x 的取值范围 F ，是一个比 LDE 域大得多的域。**

设随机点 z∈F，则 DEEP 多项式可以构造为：

$$DEEP(x)=\alpha_0\frac{s_0(x)-s_0(z)}{x-z}+\alpha_1\frac{s_1(x)-s_1(z)}{x-z}+\alpha_2\frac{CP(x)-CP(z)}{x-z}$$

注意：商多项式 DEEP(x)存在，则多个多项式的随机打开点正确，则多个多项式正确，则 trace 多项式和 CP 多项式正确，则约束正确，则计算正确，则保密数据正确。

在实际应用中，CP(x)的度数一般比迹多项式要高，取决于约束条件。如果约束条件有两列相乘（即二次的约束），则 **CP(x)的度数就为 2d。这种情况下，需要将 CP(x)拆分为多个次数小于 d 的多项式。**

上面构造的 DEEP(x)多项式，正常来说其次数应该小于 d−2（因为分子均为次数小于 d−1 次的多项式，而分母为 1 次多项式）。但是，后面做 FRI 承诺的时候计算方便，仍然通过乘上一个多项式的方式，将其次数升高一次，这样它的次数应该小于 d−1。

构造完 DEEP(x)后，如果能够证明 DEEP(x)是一个次数小于 d−1 的多项式，就可以证明所有的多项式在 z 点处的取值即是所预期的值，进而证明程序的执行结果正确，且所有的约束条件都满足。

注意：

- **先补齐、线性组合。**
- **再乘以爆炸因子。**

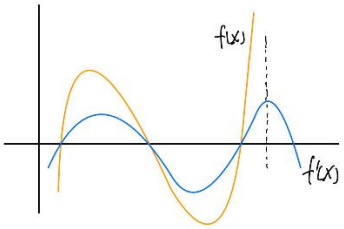
4.3.2 FRI 低度多项式测试

4.3.2.1低度多项式测试的原理

如何证明一个多项式 f(x)的次数小于特定的 d 次。

直接测试

从 f(x)上任意挑选 d+1 个点值对(x_i, $f(x_i)$)，使用其中的 d 个点值对构造一个 d−1 次多项式 f'(x)。若 f'(x)和 f(x)在第 d+1 个点处的求值相等，则有较大概率 f(x)是一个次数小于 d 的多项式。



出于安全性的考虑，可以将这个过程重复 k 次。

如果进行一次测试，证明者作弊成功的概率为 p，

那么挑选 d+k 个点值对，证明者成功的概率是 p^k，可忽略的概率。

4.3.2.2折叠多项式，减小阶

上述低度多项式测试的方法有一个缺点：构造 f'(x)需要 d+k, k≥1 个点值对，在多项式度数非常大的前提下，测试需要的时间和通信量也会变得无法承受。

解决方案举例：

假设 1000 个点，构造 d/N=100 行 N=10 列矩阵。

FRI 通过一种称为“**折叠**”的方法，降低通信开销：

1. 将所有 d=1000 个点值(x_i, $f(x_i)$)进行划分，得到 d/N=100 行与 N=10 列的**矩阵.**
2. 证明者**承诺矩阵的每一行。**承诺的方式是将 d/N=100 **行**中每个多项式的**值** $f(x_i)$计算哈希值，得到共 d/N=100 个**哈希值，组成默克尔树**并输出**树根**。
3. 对应**每一行**，证明者将该**行**的 **d/N=100 个哈希值**使用随机数 α 进行线性组合：

$$f'(x_i') = f(x_i) + \alpha^1 \cdot f(x_{i+1}) + ... + \alpha^j \cdot f(x_{i+j}) + ... + \alpha^{N-1} \cdot f(x_{N-1})$$

4. 选择对应的横坐标 x_i，则得到新多项式可以看作**原多项式的每一项按模 N=10 划分出的多项式的组合，其系数正是原多项式对应项系数的线性组合。因此，**得到一个项数为 d/N 的多项式 f'(x)。
5. 重复 1-4 步，直到新多项式的次数降到可以承受的范围之内。**假设重复 k 次，则只需检查最后的多项式的阶是否小于 d / N^k 。**

   分析：承诺每行，则多项式不可修改；然后使用随机数对低阶多项式线性组合。

定义域发生变化

原多项式的求值域上单位根分别为 $\omega^0,\omega^1,\omega^2,...,\omega^{n-1}$；

新多项式其求值域是原来的 1/N，其单位根变为 $\omega^0, \omega^N, \omega^{2N}, ..., \omega^{\frac{n}{N}*N}$

$$f'(\omega^0) = f(\omega^0) + \alpha \cdot f(w^0 * w^1) + \alpha^2 \cdot f(\omega^0 * \omega^2) + \dots$$
$$f'(\omega^N) = f(\omega^N) + \alpha \cdot f(\omega^N * w^1) + \alpha^2 \cdot f(\omega^N * \omega^2) + \dots$$
$$f'(\omega^{2N}) = f(\omega^{2N}) + \alpha \cdot f(\omega^{2N} * w^1) + \alpha^2 \cdot f(\omega^{2N} * \omega^2) + \dots$$

令 $u(x) = f(x * \omega), v(x) = f(x * \omega^2), w(x) = f(x * \omega^3)$，则

$$f'(x) = f(x) + \alpha \cdot u(x) + \alpha^2 \cdot v(x) + \alpha^3 \cdot w(x) + \dots$$
$$x \in [\omega^0, \omega^N, \dots]$$

因此，**f'(x)是次数为 d/N 的多项式**，降低原多项式 f(x)的阶 N。对多项式 **f'(x)**进行低阶检测，能够降低复杂度。

4.3.3 验证者的检查

4.3.3.1检查FRI步骤的正确性

首先，验证折叠关系。

因为证明者每次 FRI 折叠均承诺折叠后所有新点**值**对的**默克尔树**，验证者只需在同一层上抽取 N 个相关的折叠点，在下一层上抽取一个点，就能验证本层的 N 个点确实按正确的随机数折叠到下一层的同一个点。**只知道降低多少 N^k，而不知道阶是多少。**

其次，验证折叠后的多项式的阶。

验证者检查最后余下的多项式的阶小于 d / N^k，这个使用低度多项式测试进行抽查即可验证。**知道阶低于 d / N^k，就满足转换约束和边界约束。**

4.3.3.2检验多项式是否匹配

目标：确保多项式来源于转换约束、边界约束。

如果证明者伪造一个 $DEEP_{fake}(x)$，它的度数小于 d，那么它总是能够通过低度测试。因此验证者需检查：通过 FRI 低度测试的多项式，确实为证明者构造的 DEEP(x)多项式。

验证者可以向证明者请求 LDE 域中任意一点处的 $DEEP_{fake}(x)$的值，以及之前的轨迹多项式，组合多项式在该点处的值 $s_0(x), s_1(x), CP(x)$。验证者自行计算 $DEEP_{real}(x)$，若它和证明者发送的 $DEEP_{fake}(x)$不匹配，则验证不通过。

总结：验证者自己计算 k 个多项式的值，与证明方计算的值看是否相等，防止证明方承诺一个伪造的低阶多项式。

如果证明者想要伪造 $DEEP_{fake}(x)$，需要让该多项式的阶数小于 d。因此若原始的 $DEEP_{real}(x)$是个高次多项式（次数很高，远高于 LDE 求值域的度数），则二者在 LDE 上最多可以在 d 个求值点处相等（若有多于 d 个求值点相等，则伪造的多项式 $DEEP_{fake}(x)$次数必定大于等于 d）。此时，验证者随机检查一个求值点，两个多项式在点处值相等的概率即为 $1/2^k$（k 为爆炸倍数）。验证者可以 **k 次重复**这一过程，直到证明者作弊成功的概率小于目标安全参数。

2^(-40)

5.3.3.3磨损因子(Grinding Factor)

因为上述的验证手段均为概率的，为防止证明者靠庞大的计算能力进行反复重试，可以要求证明者在提供 STARK 证明的同时也提交一个工作量证明（PoW），使得重试的代价变大。

每个攻击手段，有对应的防范方法，效率才是最高的。

4.4 RS Code

4.4.1 RS Proximity Testing/Verification

之所以简单介绍 RS code，是因为 FRI 论文就是为了解决 RSproximitytesting 问题而提出的。同时，RS code 是一组编码方法，它本质是一种纠错码，最开始是为了解决**通信**中由于通信信道不可信导致的接收方来检测和更正发送方发来的原始数据的。它的基本思想很简单，就是通过发送方按照算法发送**拉格朗日冗余**。

所有的数据编码为多项式的系数 n。发送多项式的值 n，可以多发 50 个值作为**冗余 k=50**。n+50

（1） 可以少 50 个以内 ：n—>n+50.

（2） 可以错 49 个以内 ：n+1。

传输允许有 40 个值是错误的。n

4.4.2 Original RS Code(RS60)

在 FRI 文献中所找到的 RS60，就是指 I.S.REED 和 G.SOLOMON 在 1960 年发表的《POLYNOMIAL CODES OVER CERTAIN FINITE FIELDS》[2]论文所提出的纠错码编码方法，因而得名 RS code。这种编码方法，其实很容易理解，它有如下几个要点。

它的目的是将一个 m 长度的消息(message)编码成一个固定长度 n 的消息(code word)

message 与 codeword 的具体表示都是定义在一个域上的，通常是 GF(2^k)。理解为就是字符串的二进制表示，一般 $n=2^k$。

编码方法很简单，它首先构造了一个(m−1)次多项式，这个多项式的系数就是待编码的消息(message)；而编码的结果(codeword)就是这个多项式在某个乘法子群各点处的值。

正式的过程是：

- message:m-tuple($a_0,a_1,...,a_{m-1}$)
- 构造一个(m−1)次多项式 P(X)，使得 $P(X)=a_0+a_1x+a_2x^2+...+a_{m-1}x^{m-1}$
- 在本原根处求值，形成 code word

$$P(0)=a_0P(\beta)=a_0+a_1\beta+a_2\beta^2+...+a_{m-1}\beta^{m-1}$$

$$P(\beta^2)=a_0+a_1\beta^2+a_2\beta^4+...+a_{m-1}\beta^{2(m-1)}$$

$$...$$

$$P(\beta^{2n-2})=a_0+a_1\beta^{2n-2}+a_2\beta^{2(2n-2)}+...+a_{m-1}\beta^{(m-1)(2n-2)}$$

$$P(1)=a_0+a_1+a_2+...+a_{m-1}$$

所以最终的 code word 就是(P(0), P(β), $P(\beta_2)$,..., $P(\beta_{2n-2}$,P(1)))。
接收者在接到 code word 之后如果发生错误的位置在一定范围内，接收者是可以从接收到的 code word“恢复”出正确的数据的。这是因为上面任意 m 个方程都是线性无关的，所以只要“正确”的值还占主要地位，就可以恢复出正确数据。

多项式的系数编码数据 m，计算 n 个多项式的值。发送多项式的值。
如果 n>m，则存在冗余，能够容错。从 n 中随意取 m 个多项式的值，计算多项式的系数都是可以的。不用担心数据传输导致出错。

基于不同的多项式值的集合构造出来的多项式系数表达，应该是相同的，去掉不同的。

lynndell 博士 新火科技 密码学专家 lynndell2010@gmail.com