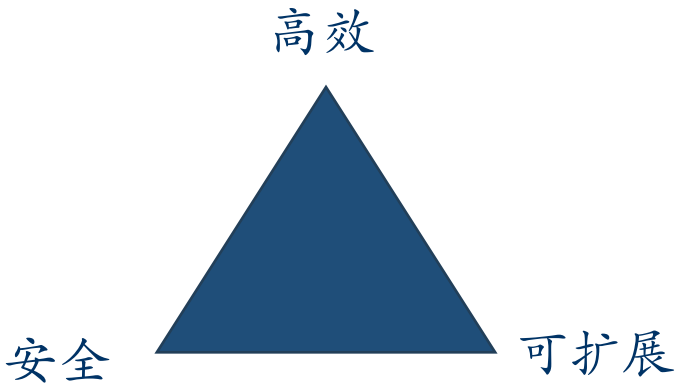
Eigen模板库
项目阅读报告

王子轩 2023011307



- Part1 整体介绍 && 框架分析
  - 功能介绍(Page2) && 使用示例(Page3)
  - 层次结构(Page4-5) && 设计模式(Page6)
- Part2 功能测试 && 拓展尝试
  - 性能测试 && 内存管理测试 (Page7-12)
  - 功能扩展 (Page13-15)
- 注：报告对应的代码为根据Eigen库功能设置的4个测点/扩展 Test 1~4

功能介绍

- 简介：Eigen 是一个开源的、基于 C++ 模板的**高性能线性代数库**。它提供了关于**矩阵、向量、数值分析**以及相关算法的丰富功能.
- 基本功能
  - 基本数据类型：支持**密集和稀疏矩阵、向量、数组**等.
  - 线性代数运算：包括**矩阵的加减乘除、转置、点积、叉积.**
  - 高级算法：提供多种**矩阵分解**（LU，QR，SVD，Cholesky）、**特征值计算、线性方程组求解器.**
- 特点
  - 高效性：通过**表达式模板（Expression Templates）**技术，避免在计算过程中产生不必要的临时对象，将多个运算连接起来进行优化.
  - 易用性：API 设计直观，代码可读性强。**仅由头文件构成**，无需链接外部库，方便集成.

使用示例 (Test 1)

```cpp
#include <iostream>
#include <Eigen/Dense> // 引入头文件
int main() {
    Eigen::Matrix3d m = Eigen::Matrix3d::Random(); // 声明Eigen数据结构
    Eigen::Vector3d v = Eigen::Vector3d::Random();
    Eigen::Vector3d result = m * v; // 直接进行线性代数运算
    std::cout << "Matrix m:\n" << m << std::endl;
    std::cout << "Vector v:\n" << v << std::endl;
    std::cout << "Result of m*v:\n" << result << std::endl;
    return 0;
}// 编译指令，指定Eigen头文件查找位置即可
// g++ -o main main.cpp -O2 -msse2 -I ../.. && ./main
```

- 引入**头文件**后，可以直接用**接近数学表达式的直观方式**编写代码，同时获得较高的计算性能.

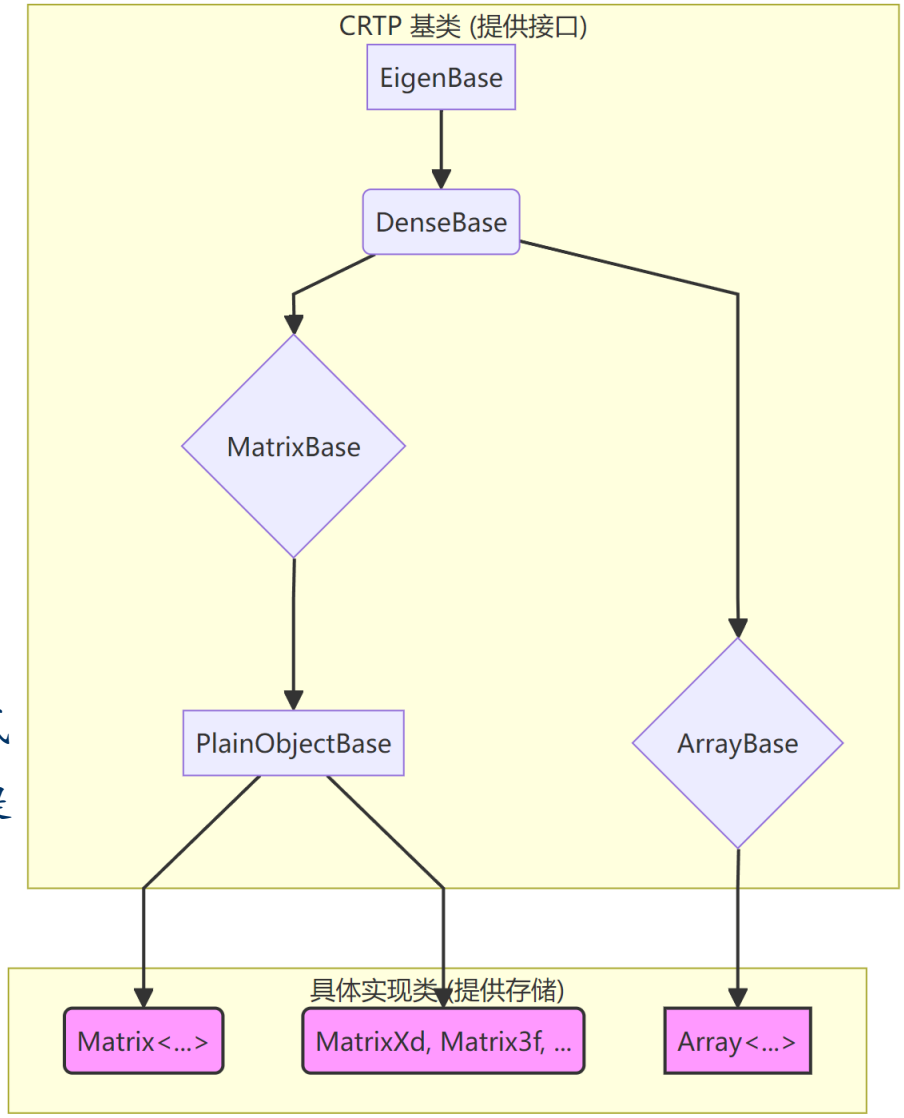
```
root@Alex:/mnt/e/GITHUB/eigen/demos/AB# make
g++ -o main main.cpp -O2 -msse2 -I ../.. && ./main
Matrix m:
 0.696235   0.33421  0.445064
 0.205189 -0.469531 -0.632501
-0.414795  0.927654 0.0241114
Vector v:
0.0722581
 0.432106
-0.046008
Result of m*v:
0.174246
-0.15896
0.369763
```

框架分析-层次结构 (整体布局)

- **主要数据结构**
  - Matrix: 用于密集(Dense)矩阵和向量的核心类. 分为动态大小 (MatrixXd, 大小在运行时确定)和固定大小(Matrix3f, 大小在编译期确定, 通常更快). Vector 是 Matrix 的一种特例(行或列为1).
  - Array: 与 Matrix 共享底层数据, 但提供的是逐元素的运算.
  - SparseMatrix: 用于高效存储和操作稀疏矩阵的核心类. 采用压缩行/列的存储方式节省内存, 为稀疏计算提供算法.
- **视图与映射 (Views and Maps):**
  - 视图: block(), row(), col() 等方法返回的对象, 是对原矩阵一部分数据的引用, 操作视图等价于操作原数据且没有拷贝开销.
  - 映射: Map 类可以将一个已存在的裸指针或C风格数组“包装”成一个Eigen对象, 使其能参与Eigen的运算.
- **主要功能模块**
  - Core: 提供 Matrix, Array 等核心数据结构以及最基础的运算.
  - Dense: 便捷头文件, 包含了Core模块以及所有用于密集矩阵的线性代数算法
  - Sparse: 提供 SparseMatrix 以及针对稀疏矩阵的算法.
  - Geometry: 处理2D和3D几何变换, 旋转: Quaternion, AngleAxis. 变换: Translation, Scaling, Transform. 在图形学、机器人学中很有用途.

框架分析-层次结构 (核心模块：Dense Object)

- 右图展示的是Core模块中密集对象(Dense Object) 的继承关系
- EigenBase<Derived>: **顶层基类**，提供了所有 Eigen 表达式共有的方法，如 rows(), cols() ,size()等
- DenseBase<Derived>: 为所有密集矩阵/数组表达式添加通用功能
- MatrixBase<Derived>/ArrayBase<Derived>: 分别为具有**线性代数语义**（矩阵乘法等）的表达式和具有**逐元素运算语义**（数组乘法等）的表达式提供接口.
- PlainObjectBase<Derived>: 关键的中间层，代表了**真正拥有内存存储的类**. Matrix和Array都继承自它; Matrix<...>/Array<...>: 具体的矩阵和数组类，用户直接使用的类.



框架分析-设计模式

- 表达式模板 (Expression Templates)
  - 对于 Vector c = a + b; 传统实现会先计算 a + b 并将结果存入一个临时对象 tmp，然后再将 tmp 赋值给 c. 这引入了额外的**内存分配和数据拷贝开销.**
  - Eigen利用C++模板元编程，表达式本身会生成一个**表达式对象** (Expression Object)，仅描述操作但**并不执行计算**。只有当这个表达式被赋值给一个具体的 Matrix 或 Vector 时，Eigen 才会遍历每一个元素，**完全避免了临时对象**
- 静态多态与**CRTP** (Curiously Recurring Template Pattern).
  - CRTP: 让**子类将自身作为模板参数传递给基类**的设计模式. 例如 class Matrix : public MatrixBase<Matrix>.
  - 基类 MatrixBase 可以通过模板参数直接调用子类 Matrix 的方法，实现了**静态多态**，避免了**虚函数带来的运行时开销**，并将很多**派发决策提前到编译期**完成.
- 优点
  - 实现了**延迟计算(Lazy Evaluation)**: 通过**消除中间临时对象**、增多编译器的**优化机会**，大大提升了**计算性能**
  - **灵活性 (Flexibility):**无论是具体的矩阵 MatrixXd 还是复杂的表达式如 A.transpose() * B 都继承自相同的基类体系，可以被传递给期望 MatrixBase的函数.

功能测试-链式表达式求值性能

- Test2：链式表达式求值性能测试（高效性）
- Test3：内存管理安全性（鲁棒性）
- Test4：自定义表达式求值（可扩展性）
- Test2
- 任务设定：计算表达式y = a*s1 + b*s2 + c*s3，向量包含200万个元素.量化分析在执行链式代数运算时，由临时对象物化所带来的性能损耗.
- 测试对象：Eigen::VectorXd，Eigen 的动态双精度浮点向量. 其设计采用了表达式模板技术，运算不立即求值
- 测试方法：手动实现NaiveVector作为baseline与Eigen库的实现进行耗时比较.

naive > Naivevector.h > NaiveVector<T>

```cpp
template<typename T>
class NaiveVector {
public:
    std::vector<T> data;
    NaiveVector(size_t size = 0) : data(size) {}
    NaiveVector(const std::vector<T>& vec) : data(vec) {}
    NaiveVector(const NaiveVector<T>& other) : data(other.data) {}
    NaiveVector<T>& operator=(const NaiveVector<T>& other) {
        if (this != &other) {
            data = other.data;
        }
        return *this;
    }
    size_t size() const { return data.size(); }
    NaiveVector<T> operator+(const NaiveVector<T>& other) const {
        if (size() != other.size()) {
            throw std::runtime_error("Vector sizes do not match for addition.");
        }
        NaiveVector<T> result(size());
        for (size_t i = 0; i < size(); ++i) {
            result.data[i] = data[i] + other.data[i];
        }
        return result; // 返回临时对象，触发拷贝/移动
    }
    NaiveVector<T> operator*(T scalar) const {
        NaiveVector<T> result(size());
        for (size_t i = 0; i < size(); ++i) {
            result.data[i] = data[i] * scalar;
        }
        return result; // 返回临时对象
    }

    friend NaiveVector<T> operator*(T scalar, const NaiveVector<T>& vec) {
        return vec * scalar;
    }
    static NaiveVector<T> Random(size_t size) {
        NaiveVector<T> vec(size);
        for(size_t i = 0; i < size; ++i) {
            vec.data[i] = static_cast<T>(rand()) / static_cast<T>(RAND_MAX);
        }
        return vec;
    }
    bool isApprox(const NaiveVector<T>& other, T tolerance = 1e-6) const {
        if (size() != other.size()) return false;
        for (size_t i = 0; i < size(); ++i) {
            if (std::abs(data[i] - other.data[i]) > tolerance) {
                return false;
            }
        }
        return true;
```

功能测试-链式表达式求值性能

- **测试结果:**
  - NaiveVector 实现耗时: 103.717 ms
  - Eigen 实现耗时: 21.913 ms

```cpp
void test_chained_operations(size_t size) {
    std::cout << "\n--- Test Case 2: Chained Operations Performance (Vector size: " << size << ") ---\n";

    double s1 = 1.5, s2 = -2.0, s3 = 0.5;

    // --- Naive 实现 ---
    auto na = NaiveVector<double>::Random(size);
    auto nb = NaiveVector<double>::Random(size);
    auto nc = NaiveVector<double>::Random(size);
    NaiveVector<double> nd_naive(size);
    {
        Timer t("Naive Chained Ops");
        nd_naive = na * s1 + nb * s2 + nc * s3;
    }

    // --- Eigen 实现 ---
    // 使用 Map 将 NaiveVector 的数据包装成 Eigen 对象，无需拷贝，确保数据一致
    Eigen::Map<Eigen::VectorXd> ea(na.data.data(), size);
    Eigen::Map<Eigen::VectorXd> eb(nb.data.data(), size);
    Eigen::Map<Eigen::VectorXd> ec(nc.data.data(), size);
    Eigen::VectorXd nd_eigen(size);
    {
        Timer t("Eigen Chained Ops");
        nd_eigen = ea * s1 + eb * s2 + ec * s3;
    }
  bool passed = true;
    for (size_t i = 0; i < size; ++i) {
        if (std::abs(nd_naive.data[i] - nd_eigen(i)) > 1e-9) {
            passed = false;
            break;
        }
    }
    std::cout << "Verification: Results are " << (passed ? "CONSISTENT." : "INCONSISTENT!") << std::endl;
}
```

- **机制分析:**
- NaiveVector在每一个 + 或 * 操作都立即在堆上分配新内存，执行一次完整的循环，然后返回一个临时对象。这导致多次完整的数据遍历，极大地消耗了内存带宽并阻碍了编译器优化

- **Eigen 的高性能机制**
  - **表达式模板**：将C++的类型系统用作一种元编程工具。执行 a * s1 + ... 时，并不调用任何返回 VectorXd 的运算符. 返回一个独特的、无状态的代理对象. 其类型编码整个运算结构，CwiseBinaryOp<internal::scalar_sum_op<...>, ...>. 代理对象本身是一张计算图的静态表示。

功能测试-链式表达式求值性能

- **Eigen 的高性能机制（续）**
  - **CRTP && 静态派发：**
    - Eigen 的抽象基类 EigenBase<Derived> 采用了**奇异循环模板模式（CRTP）.**
    - 表达式树中的每个节点都继承自 EigenBase，并将**自身类型作为 Derived 参数传入.**
    - 基类通过 static_cast<Derived*>(this) 调用派生类的具体实现，而无需 virtual 关键字。这种静态派发机制**完全消除了动态多态的虚函数表查找开销**，为编译器进行强制内联（Inlining）优化提供便利.
  - **延迟计算与解释器模式：**
    - 赋值操作（operator=）是触发计算的信号.
    - AssignEvaluator 组件开始工作，可以被视为一个解释器，任务是遍历表达式模板.
    - AssignEvaluator 递归地解析表达式树的类型结构，最终为整个表达式生成一个单一、优化的循环使得**多个逻辑操作在一次物理内存遍历**中完成，极大地提升了**数据局部性**，最大限度地减少了对内存总线的访问压力.

```
--- Test Case 2: Chained Operations Performance (Vector size: 2000000) ---
[TIMER] Naive Chained Ops: 103.717 ms
[TIMER] Eigen Chained Ops: 21.913 ms
Verification: Results are CONSISTENT.
```

功能测试-内存管理安全性

- **Test3**
- **任务设定**：执行一个显式的Aliasing操作：`v.head(5) = v.tail(5);`此操作意图用向量的后5个元素覆盖前5个元素。
- **测试目标**：评估 Eigen 的计算框架在处理内存混叠（即赋值操作的源与目标内存区域存在重叠）时的正确性和鲁棒性
- **测试结果**：通过

```
--- Test Case 3: Aliasing Robustness Test ---
Original vector:
 1  2  3  4  5  6  7  8  9 10
Aliased result (Eigen default):
 6  7  8  9 10  6  7  8  9 10
Verification (Eigen default): CORRECT
```

- **测试分析**：采用延迟计算的朴素实现，操作会因数据冒险而失败。求值器可能会生成 `v[0]=v[5]`, `v[1]=v[6]`, ... 的指令.一旦 `v[0]` 被写入，后续计算若依赖于旧的 `v[0]`（例如 `v=v.reverse()`），就会读取到错误的数据。
- **Eigen 的解决方案**：
  - 策略（Strategy）模式

功能测试-内存管理安全性

```
//简化版核心代码 Eigen/src/Core/AssignEvaluator.h
namespace Eigen {
namespace internal {
template<typename DstXprType> struct Assignation;
template<typename DstXprType>
class AssignEvaluator
{
public:
    EIGEN_DEVICE_FUNC
    AssignEvaluator(DstXprType& dst)
      : m_dst(dst)
    {
        // ...
    }
    template<typename SrcXprType>
    EIGEN_DEVICE_FUNC
    void run(const SrcXprType& src)
    {
        // ... (根据类型和标志位进行大量的编译时分派)

#if !defined(EIGEN_NO_ALGORITHMIC_ALIASING)
        // 核心的混叠检测逻辑
        // check_for_aliasing 会在运行时进行指针和大小的比较
        if (check_for_aliasing(m_dst, src))
        {
           //
           // PlainObjectBase::eval() 会触发一次完整的求值
            m_dst.copy(src.eval());
        }
        else
#endif
        {
            // 如果没有混叠，则执行“高性能”策略：
            // 直接调用赋值函数，这将生成融合的循环，直接写入目标内存。
            Assignation<DstXprType>::run(m_dst, src);
        }
    }

  protected:
    typename DstXprType::PlainObject& m_dst;
    // ...
};

} // end namespace internal

} // end namespace Eigen

// 测试代码如下
```

```
void test_aliasing() {
    std::cout << "\n--- Test Case 3: Aliasing Robustness Test ---\n";

    Eigen::VectorXd v_eigen(10);
    v_eigen << 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
    std::cout << "Original vector:\n" << v_eigen.transpose() << std::endl;

    Eigen::VectorXd v_expected(10);
    v_expected << 6, 7, 8, 9, 10, 6, 7, 8, 9, 10;

    v_eigen.head(5) = v_eigen.tail(5);

    std::cout << "Aliased result (Eigen default):\n" << v_eigen.transpose() << std::endl;
    std::cout << "Verification (Eigen default): " << (v_eigen.isApprox(v_expected) ? "CORRECT" : "WRONG") << std::endl;

    // ... (aliasing with noalias part remains the same) ...
}
```

功能测试-内存管理安全性

- **Eigen的策略（Strategy）模式**
- operator= 赋值操作是此模式的调用上下文.
- AssignEvaluator 扮演客户端角色，它负责在运行时决策并执行一个**具体的求值策略.**
  - **高性能策略**：假定无混叠风险，直接生成**融合循环进行原地（in-place）赋值.**
  - **安全策略**：它首先在内部申请一块**临时内存**，将表达式右侧完整结果计算并存入这块“干净”的内存中。随后，再将这个**临时结果拷贝到左侧的目标内存区域.**
- **动态策略选择**：在执行任何求值动作前，AssignEvaluator 会调用一个轻量级的 does_alias(...) 函数. 该函数通过对表达式树中涉及的各个对象的指针和大小进行分析，判断是否存在内存重叠。如果检测到混叠，则自动选择安全策略；否则，采用 DirectEvaluation 策略.
- 这种设计自动规避了潜在的逻辑错误。同时，Eigen 提供了 .noalias() 方法，允许开发者在确信没有混叠风险的情况下，手动禁止此检查，强制使用最高性能的直接求值策略.

功能扩展-自定义表达式求值扩展

- **Test3**
- **任务设定**：实现的自定义函数 Eigen::poly()
- **扩展目标**：在不修改 Eigen 源码的前提下，为其添加新的、可无缝集成到表达式模板系统中的自定义函数
- **测试结果**：通过

```cpp
void test_extension() {
    std::cout << "\n--- Test Case 4: Extensibility Test ---\n";

    Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(5, 1, 5);
    double c0 = 1.0, c1 = 2.0, c2 = 3.0;

    std::cout << "Input vector x:\n" << x.transpose() << std::endl;

    Eigen::VectorXd y_poly;
    {
        Timer t("Custom 'poly' expression");
        // poly 现在在 Eigen 命名空间中
        y_poly = Eigen::poly(x, c0, c1, c2);
    }
    std::cout << "Result of poly(x, 1, 2, 3):\n" << y_poly.transpose() << std::endl;

    // ... (verification and seamless integration part remains the same, but call Eigen::poly) ...
    Eigen::VectorXd z;
    {
        Timer t("Custom expr + Eigen expr");
        z = Eigen::poly(x, c0, c1, c2) + 2.0 * x - x.cwiseInverse();
    }
    std::cout << "Result of 'poly(x, 1, 2, 3) + 2.0 * x - 1.0/x':\n" << z.transpose() << std::endl;

    Eigen::VectorXd y_expected(5);
    for(int i=0; i<5; ++i) { y_expected(i) = c0 + c1 * x(i) + c2 * x(i) * x(i); }
    Eigen::VectorXd z_expected = y_expected + 2.0 * x - x.cwiseInverse();
    std::cout << "Verification (poly): " << (y_poly.isApprox(y_expected) ? "CORRECT" : "WRONG") << std::endl;
    std::cout << "Verification (integrated): " << (z.isApprox(z_expected) ? "CORRECT" : "WRONG") << std::endl;
}
```

```
--- Test Case 4: Extensibility Test ---
Input vector x:
1 2 3 4 5
[TIMER] Custom 'poly' expression: 0 ms
Result of poly(x, 1, 2, 3):
 6 17 34 57 86
[TIMER] Custom expr + Eigen expr: 0 ms
Result of 'poly(x, 1, 2, 3) + 2.0 * x - 1.0/x':
      7    20.5 39.6667   64.75    95.8
Verification (poly): CORRECT
Verification (integrated): CORRECT
```

功能测试-自定义表达式求值扩展

```
template<typename Scalar>
struct PolynomialFunctor {
    const Scalar c0, c1, c2;
    PolynomialFunctor(Scalar c0, Scalar c1, Scalar c2) : c0(c0), c1(c1), c2(c2) {}

    EIGEN_STRONG_INLINE const Scalar operator()(const Scalar& x) const {
        return c0 + x * (c1 + x * c2);
    }
};
namespace Eigen {
template<typename ArgType> class Polynomial;
namespace internal {
template<typename ArgType>
    struct traits<Polynomial<ArgType>> : public traits<ArgType> {};
}
template<typename ArgType>
class Polynomial : public CwiseUnaryOp<PolynomialFunctor<typename internal::traits<ArgType>::Scalar>, const ArgType>
{
public:
    typedef CwiseUnaryOp<PolynomialFunctor<typename internal::traits<ArgType>::Scalar>, const ArgType> Base;
    EIGEN_GENERIC_PUBLIC_INTERFACE(Polynomial)

    Polynomial(const ArgType& arg,
               typename internal::traits<ArgType>::Scalar c0,
               typename internal::traits<ArgType>::Scalar c1,
               typename internal::traits<ArgType>::Scalar c2)
        : Base(arg, PolynomialFunctor<Scalar>(c0, c1, c2)) {}
};
template <typename ArgType>
Polynomial<ArgType> poly(const DenseBase<ArgType>& xpr,
                         typename internal::traits<ArgType>::Scalar c0,
                         typename internal::traits<ArgType>::Scalar c1,
                         typename internal::traits<ArgType>::Scalar c2) {
    return Polynomial<ArgType>(xpr.derived(), c0, c1, c2);} } // namespace Eigen
```

功能测试-自定义表达式求值扩展

- **Eigen 的扩展机制分析**
  - **装饰器模式**：创建 Polynomial<ArgType> 类是一个**装饰器**。它“包裹”了另一个 Eigen 表达式（ArgType），在不改变其核心接口的前提下，为其“装饰”上“多项式函数”的新能力.
  - **CRTP 的接口注入**：该装饰器能够被 Eigen 框架识别和接纳，核心在于它**通过 CRTP 继承了 Eigen 的内部基类,**相当于将 Polynomial “注入”到 Eigen 的类型系统中，使其**表现得如同一个原生表达式**，从而能够响应赋值求值、参与更复杂的表达式链。.
  - **命令模式**：
    - PolynomialFunctor 结构体扮演命令的角色，将一个**具体操作**（多项式计算）及其所需的**所有信息**（三个系数）**封装在一个可传递的对象**中.
    - Polynomial 装饰器持有这个**命令对象**，并在最终求值循环的每个迭代中执行它,实现了计算逻辑与表达式结构的解耦分离.