添加新的表达式类型

> 警告
>
> 免责声明：此页面专为不惧怕处理某些 Eigen 内部方面的高级用户量身定制。在大多数情况下，自定义表达式可以通过使用自定义一元或二元函子来避免，而极其复杂的矩阵操作可以通过上一页中所述的零元函子来实现。

本页通过示例描述了如何在 Eigen 中实现新的轻量级表达式类型。这由三部分组成：表达式类型本身，一个包含有关表达式的编译时信息的特征类，以及用于将表达式计算为矩阵的评估器类。

TO DO：写一个页面来解释设计，详细介绍矢量化等，并在此处参考该页面。

那个设定

循环矩阵是这样一种矩阵，其中每一列都与左边的列相同，只是它向下循环移位。例如，这是一个 4×4 循环矩阵：

$$v = \begin{bmatrix} 1 & 8 & 4 & 2 \\ 2 & 1 & 8 & 4 \\ 4 & 2 & 1 & 8 \\ 8 & 4 & 2 & 1 \end{bmatrix} \qquad (1)$$

循环矩阵由其第一列唯一确定。我们希望编写一个函数`makeCirculant`，在给定第一列的情况下，返回一个表示循环矩阵的表达式。

为简单起见，我们将`makeCirculant`函数限制为密集矩阵。也允许数组或稀疏矩阵可能是有意义的，但我们不会在这里这样做。我们也不想支持矢量化。

入门

我们将部分介绍实现该`makeCirculant`功能的文件。我们首先包含适当的头文件并转发声明表达式类，我们将称之为`Circulant`. 该`makeCirculant`函数将返回此类型的对象。这个类`Circulant`实际上是一个类模板；模板参数`ArgType`是指传递给`makeCirculant`函数的向量的类型。

```
#include <Eigen/Core>
#include <iostream>

template <class ArgType> class Circulant;
```

特质类

对于每个表达式类`X`，命名空间中都应该有一个特征类`Traits<X>`，其中`Eigen::internal`包含有关`X`称为编译时间的信息。

如设置中所述，我们设计了`Circulant`表达式类来引用密集矩阵。循环矩阵的条目与传递给`makeCirculant`函数的向量的条目具有相同的类型。用于索引条目的类型也是相同的。再次为简单起见，我们将只返回列主矩阵。最后，循环矩阵是一个方阵（行数等于列数），行数等于传递给`makeCirculant`函数的列向量的行数。如果这是一个动态大小的向量，则循环矩阵的大小在编译时是未知的。

这导致以下代码：

```
namespace Eigen {
    namespace internal {
        template <class ArgType>
            struct traits<Circulant<ArgType> >
            {
                typedef Eigen::Dense StorageKind;
                typedef Eigen::MatrixXpr XprKind;
                typedef typename ArgType::StorageIndex StorageIndex;
                typedef typename ArgType::Scalar Scalar;
                enum {
                    Flags = Eigen::ColMajor,
                    RowsAtCompileTime = ArgType::RowsAtCompileTime,
                    ColsAtCompileTime = ArgType::RowsAtCompileTime,
                    MaxRowsAtCompileTime = ArgType::MaxRowsAtCompileTime,
                    MaxColsAtCompileTime = ArgType::MaxRowsAtCompileTime
                };
            };
    }
}
```

表达式类

下一步是定义表达式类本身。在我们的例子中，我们想要继承 from `MatrixBase` 以公开密集矩阵的接口。在构造函数中，我们检查是否向我们传递了一个列向量（参见断言），并将我们将要从中构建循环矩阵的向量存储在成员变量中 `m_arg`。最后，表达式类应该计算相应循环矩阵的大小。如上所述，这是一个方阵，其列数与用于构造矩阵的向量一样多。

TO DO：怎么样 `Nested` 的typedef？似乎有必要；这只是暂时的吗？

```
template <class ArgType>
class Circulant : public Eigen::MatrixBase<Circulant<ArgType> >
{
public:
    Circulant(const ArgType& arg)
        : m_arg(arg)
        {
            EIGEN_STATIC_ASSERT(ArgType::ColsAtCompileTime == 1,

 YOU_TRIED_CALLING_A_VECTOR_METHOD_ON_A_MATRIX);
        }

    typedef typename Eigen::internal::ref_selector<Circulant>::type Nested;

    typedef Eigen::Index Index;
    Index rows() const { return m_arg.rows(); }
    Index cols() const { return m_arg.rows(); }

    typedef typename Eigen::internal::ref_selector<ArgType>::type
ArgTypeNested;
    ArgTypeNested m_arg;
};
```

评估者

最后一个大片段实现了`Circulant`表达式的求值器。评估器计算循环矩阵的条目；这是在``.coeff() 成员函数中完成的。这些条目是通过找到构造循环矩阵的向量的对应条目来计算的。当循环矩阵由由复杂表达式给出的向量构成时，获得此条目实际上可能并非易事，因此我们使用与向量相对应的求值器。

该`CoeffReadCost`常量记录了计算循环矩阵的一个条目的成本；我们忽略索引计算，并说这与计算向量的条目的成本相同，循环矩阵是从该向量构建的。

在构造函数中，我们保存了定义循环矩阵的列向量的求值器。我们还保存了该向量的大小；请记住，我们可以查询表达式对象来查找大小而不是求值器。

```
namespace Eigen {
    namespace internal {
    template<typename ArgType>
    struct evaluator<Circulant<ArgType> >
        : evaluator_base<Circulant<ArgType> >
        {
            typedef Circulant<ArgType> XprType;
            typedef typename nested_eval<ArgType,
XprType::ColsAtCompileTime>::type ArgTypeNested;
            typedef typename remove_all<ArgTypeNested>::type
ArgTypeNestedCleaned;
            typedef typename XprType::CoeffReturnType CoeffReturnType;

            enum {
                CoeffReadCost =
evaluator<ArgTypeNestedCleaned>::CoeffReadCost,
                Flags = Eigen::ColMajor
            };

            evaluator(const XprType& xpr)
                : m_argImpl(xpr.m_arg), m_rows(xpr.rows())
                { }

            CoeffReturnType coeff(Index row, Index col) const
            {
                Index index = row - col;
                if (index < 0) index += m_rows;
                return m_argImpl.coeff(index);
            }

            evaluator<ArgTypeNestedCleaned> m_argImpl;
            const Index m_rows;
        };
    }
}
```

入口点

毕竟，`makeCirculant`功能非常简单。它只是创建一个表达式对象并返回它。

```
template <class ArgType>
Circulant<ArgType> makeCirculant(const Eigen::MatrixBase<ArgType>& arg)
{
    return Circulant<ArgType>(arg.derived());
}
```

一个简单的主要测试功能

最后，一个简短的`main`函数展示了如何`makeCirculant`调用该函数。

```
int main()
{
    Eigen::VectorXd vec(4);
    vec << 1, 2, 4, 8;
    Eigen::MatrixXd mat;
    mat = makeCirculant(vec);
    std::cout << mat << std::endl;
}
```

如果将所有片段组合在一起，则会产生以下输出，表明程序按预期工作：

```
1 8 4 2
2 1 8 4
4 2 1 8
8 4 2 1
```