通过空值表达式进行矩阵操作

CwiseNullaryOp类的主要目的是定义*程序*矩阵，例如由 Ones()、Zero()、Constant()、Identity() 和 Random() 方法返回的常量或随机矩阵。然而，通过一些想象力，可以用最少的努力完成非常复杂的矩阵操作，因此很少需要实现新的表达式。

示例 1：循环矩阵

为了探索这些可能性，让我们从实现新表达式主题的*循环*示例开始。让我们回想一下，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。

对于本练习， 的返回类型`makeCirculant`将是我们需要实例化的CwiseNullaryOp： 1 - 正确`circulant_functor`存储输入向量并实现适当的系数访问器`operator(i,j)`2 - 类Matrix的模板实例化，传达编译时信息，例如标量类型、大小和首选存储布局。

调用`ArgType`输入向量的类型，我们可以构造等价的平方矩阵类型如下：

```
template<class ArgType>
struct circulant_helper {
    typedef Matrix<typename ArgType::Scalar,
                   ArgType::SizeAtCompileTime,
                   ArgType::SizeAtCompileTime,
                   ColMajor,
                   ArgType::MaxSizeAtCompileTime,
                   ArgType::MaxSizeAtCompileTime> MatrixType;
};
```

这个小助手结构将帮助我们实现我们的`makeCirculant`功能如下：

```
template <class ArgType>
CwiseNullaryOp<circulant_functor<ArgType>, typename
circulant_helper<ArgType>::MatrixType>
makeCirculant(const Eigen::MatrixBase<ArgType>& arg)
{
    typedef typename circulant_helper<ArgType>::MatrixType MatrixType;
    return MatrixType::NullaryExpr(arg.size(), arg.size(),
circulant_functor<ArgType>(arg.derived()));
}
```

像往常一样，我们的函数将参数 a 作为参数`MatrixBase`（有关更多详细信息，请参阅此页面）。然后，通过具有足够运行时大小的DenseBase::NullaryExpr静态方法构造CwiseNullaryOp对象。

然后，我们需要实现我们的`circulant_functor`，这是一个简单的练习：

```cpp
template<class ArgType>
class circulant_functor {
    const ArgType &m_vec;
public:
    circulant_functor(const ArgType& arg) : m_vec(arg) {}

    const typename ArgType::Scalar& operator() (Index row, Index col) const
{
        Index index = row - col;
        if (index < 0) index += m_vec.size();
        return m_vec(index);
    }
};
```

我们现在都准备尝试我们的新功能：

```cpp
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
```

这种实现 `makeCirculant` 比从头开始定义新表达式要简单得多。

示例 2：索引行和列

这里的目标是模拟 MatLab 通过两个索引向量分别引用要选取的行和列来索引矩阵的能力，如下所示：

```
A =
 7  9 -5 -3
-2 -6  1  0
 6 -3  0  9
 6  6  3  9

A([1 2 1], [3 2 1 0 0 2]) =
 0  1 -6 -2 -2  1
 9  0 -3  6  6  0
 0  1 -6 -2 -2  1
```

为此，让我们首先编写一个空函子来存储对输入矩阵和两个索引数组的引用，并实现所需的 `operator()(i,j)`：

```cpp
template<class ArgType, class RowIndexType, class ColIndexType>
class indexing_functor {
```

```
    const ArgType &m_arg;
    const RowIndexType &m_rowIndices;
    const ColIndexType &m_colIndices;
    public:
    typedef Matrix<typename ArgType::Scalar,
    RowIndexType::SizeAtCompileTime,
    ColIndexType::SizeAtCompileTime,
    ArgType::Flags&RowMajorBit?RowMajor:ColMajor,
    RowIndexType::MaxSizeAtCompileTime,
    ColIndexType::MaxSizeAtCompileTime> MatrixType;

    indexing_functor(const ArgType& arg, const RowIndexType& row_indices,
const ColIndexType& col_indices)
        : m_arg(arg), m_rowIndices(row_indices), m_colIndices(col_indices)
        {}

    const typename ArgType::Scalar& operator() (Index row, Index col) const
{
        return m_arg(m_rowIndices[row], m_colIndices[col]);
    }
};
```

然后，让我们创建一个`indexing(A,rows,cols)`创建空表达式的函数：

```
template <class ArgType, class RowIndexType, class ColIndexType>
    CwiseNullaryOp<indexing_functor<ArgType,RowIndexType,ColIndexType>,
typename indexing_functor<ArgType,RowIndexType,ColIndexType>::MatrixType>
    mat_indexing(const Eigen::MatrixBase<ArgType>& arg, const RowIndexType&
row_indices, const ColIndexType& col_indices)
    {
        typedef indexing_functor<ArgType,RowIndexType,ColIndexType> Func;
        typedef typename Func::MatrixType MatrixType;
        return MatrixType::NullaryExpr(row_indices.size(),
col_indices.size(), Func(arg.derived(), row_indices, col_indices));
    }
```

最后，这是一个如何使用此函数的示例：

```
Eigen::MatrixXi A = Eigen::MatrixXi::Random(4,4);
Array3i ri(1,2,1);
ArrayXi ci(6); ci << 3,2,1,0,0,2;
Eigen::MatrixXi B = mat_indexing(A, ri, ci);
std::cout << "A =" << std::endl;
std::cout << A << std::endl << std::endl;
std::cout << "A([" << ri.transpose() << "], [" << ci.transpose() << "]) =" <<
std::endl;
std::cout << B << std::endl;
```

这个简单的实现已经非常强大，因为行或列索引数组也可以是执行偏移、取模、跨步、反向等的表达式。

```
B =  mat_indexing(A, ri+1, ci);
std::cout << "A(ri+1,ci) =" << std::endl;
std::cout << B << std::endl << std::endl;
#if EIGEN_COMP_CXXVER >= 11
B =  mat_indexing(A, ArrayXi::LinSpaced(13,0,12).unaryExpr([](int x){return x%4;}), ArrayXi::LinSpaced(4,0,3));
std::cout << "A(ArrayXi::LinSpaced(13,0,12).unaryExpr([](int x){return x%4;}), ArrayXi::LinSpaced(4,0,3)) =" << std::endl;
std::cout << B << std::endl << std::endl;
#endif
```

输出是:

```
A(ri+1,ci) =
 9  0 -3  6  6  0
 9  3  6  6  6  3
 9  0 -3  6  6  0

A(ArrayXi::LinSpaced(13,0,12).unaryExpr([](int x){return x%4;}), ArrayXi::LinSpaced(4,0,3)) =
 7  9 -5 -3
-2 -6  1  0
 6 -3  0  9
 6  6  3  9
 7  9 -5 -3
-2 -6  1  0
 6 -3  0  9
 6  6  3  9
 7  9 -5 -3
-2 -6  1  0
 6 -3  0  9
 6  6  3  9
 7  9 -5 -3
```