块操作

此页面解释了块操作的要点。块是矩阵或阵列的矩形部分。块表达式既可以用作右值，也可以用作左值。与Eigen表达式一样，只要您让编译器优化，这种抽象的运行时成本为零。

使用块操作

Eigen 中最通用的块操作称为.block()。有两个版本，其语法如下：

块 操作	构建 动态大小块表达式的版本	构建 固定大小块表达式的版本
块大小`(p,q)`，从`(i,j)`	matrix.block(i, j, p, q);	matrix.block<p, q>(i, j);

和Eigen 一样，索引从 0 开始。

两个版本都可用于固定大小和动态大小的矩阵和数组。这两个表达式在语义上是等价的。唯一的区别是，如果块大小很小，固定大小的版本通常会给你更快的代码，但需要在编译时知道这个大小。

以下程序使用动态大小和固定大小版本来打印矩阵内多个块的值。

例子：

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

using namespace std;

int main()
{
    Eigen::MatrixXf m(4,4);
    m <<  1, 2, 3, 4,
        5, 6, 7, 8,
        9,10,11,12,
        13,14,15,16;
    cout << "Block in the middle" << endl;
    cout << m.block<2,2>(1,1) << endl << endl;
    for (int i = 1; i <= 3; ++i)
    {
        cout << "Block of size " << i << "x" << i << endl;
        cout << m.block(0,0,i,i) << endl << endl;
    }
}
```

输出：

```
Block in the middle
 6  7
10 11

Block of size 1x1
1

Block of size 2x2
```

```
1 2
5 6

Block of size 3x3
 1  2  3
 5  6  7
 9 10 11
```

在上面的例子中，.block()函数被用作*右值*，即它只被读取。但是，块也可以用作*左值*，这意味着您可以分配给块。

这在以下示例中进行了说明。此示例还演示了数组中的块，其工作方式与上面演示的矩阵中的块完全相同。

例子：

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

using namespace std;
using namespace Eigen;

int main()
{
    Array22f m;
    m << 1,2,
         3,4;
    Array44f a = Array44f::Constant(0.6);
    cout << "Here is the array a:" << endl << a << endl << endl;
    a.block<2,2>(1,1) = m;
    cout << "Here is now a with m copied into its central 2x2 block:" <<
endl << a << endl << endl;
    a.block(0,0,2,3) = a.block(2,1,2,3);
    cout << "Here is now a with bottom-right 2x3 block copied into top-left
2x3 block:" << endl << a << endl << endl;
}
```

输出：

```
Here is the array a:
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6
0.6 0.6 0.6 0.6

Here is now a with m copied into its central 2x2 block:
0.6 0.6 0.6 0.6
0.6   1   2 0.6
0.6   3   4 0.6
0.6 0.6 0.6 0.6

Here is now a with bottom-right 2x3 block copied into top-left 2x3 block:
  3   4 0.6 0.6
0.6 0.6 0.6 0.6
0.6   3   4 0.6
0.6 0.6 0.6 0.6
```

虽然.block()方法可用于任何块操作，但对于特殊情况还有其他方法，可提供更专业的 API 和/或更好的性能。在性能方面，重要的是在编译时提供尽可能多的Eigen信息。例如，如果您的块是矩阵中的单个整列，则使用下面描述的专用.col()函数让Eigen知道这一点，这可以为其提供优化机会。

本页的其余部分描述了这些专门的方法。

列和行

单独的列和行是块的特殊情况。Eigen提供了轻松解决它们的方法：.col()和.row()。

块操作	方法
第i行*	matrix.row(i);
第j列*	matrix.col(j);

对自变量 `col()` 和 `row()` 将被访问的列或行的索引。和Eigen 一样，索引从 0 开始。

例子：

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

using namespace std;

int main()
{
    Eigen::MatrixXf m(3,3);
    m << 1,2,3,
        4,5,6,
        7,8,9;
    cout << "Here is the matrix m:" << endl << m << endl;
    cout << "2nd Row: " << m.row(1) << endl;
    m.col(2) += 3 * m.col(0);
    cout << "After adding 3 times the first column into the third column, the matrix m is:\n";
    cout << m << endl;
}
```

输出：

```
Here is the matrix m:
1 2 3
4 5 6
7 8 9
2nd Row: 4 5 6
After adding 3 times the first column into the third column, the matrix m is:
 1  2  6
 4  5 18
 7  8 30
```

该示例还演示了块表达式（此处为列）可以像任何其他表达式一样用于算术。

角相关操作

Eigen还为与矩阵或阵列的角或边之一齐平的块提供特殊方法。例如，.topLeftCorner()可用于引用矩阵左上角的块。

下表总结了不同的可能性：

块操作	构建 动态大小块表达式的版本	构建 固定大小块表达式的版本
左上角 p × q 块*	matrix.topLeftCorner(p,q);	matrix.topLeftCorner<p,q>();
左下 p × q 块*	matrix.bottomLeftCorner(p,q);	matrix.bottomLeftCorner<p,q>();
右上角 p 乘 q 块*	matrix.topRightCorner(p,q);	matrix.topRightCorner<p,q>();
右下 p by q 块*	matrix.bottomRightCorner(p,q);	matrix.bottomRightCorner<p,q>();
包含前 q 行的块*	matrix.topRows(q);	matrix.topRows<q>();
包含最后 q 行的块*	matrix.bottomRows(q);	matrix.bottomRows<q>();
包含前 p 列的块*	matrix.leftCols(p);	matrix.leftCols<p>();
包含最后 q 列的块*	matrix.rightCols(q);	matrix.rightCols<q>();
包含从 i *开始的 q 列的块	matrix.middleCols(i,q);	matrix.middleCols<q>(i);
包含从 i *开始的 q 行的块	matrix.middleRows(i,q);	matrix.middleRows"(i);"

下面是一个简单的例子，说明了上述操作的使用：

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

using namespace std;

int main()
{
    Eigen::Matrix4f m;
    m << 1, 2, 3, 4,
         5, 6, 7, 8,
         9, 10,11,12,
         13,14,15,16;
    cout << "m.leftCols(2) =" << endl << m.leftCols(2) << endl << endl;
    cout << "m.bottomRows<2>() =" << endl << m.bottomRows<2>() << endl << endl;
    m.topLeftCorner(1,3) = m.bottomRightCorner(3,1).transpose();
    cout << "After assignment, m = " << endl << m << endl;
}
```

输出：

```
m.leftCols(2) =
 1  2
 5  6
```

```
 9 10
13 14

m.bottomRows<2>() =
 9 10 11 12
13 14 15 16

After assignment, m =
 8 12 16  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

向量的块操作

Eigen还提供了一组专门为向量和一维数组的特殊情况设计的块操作：

块操作	构建 动态大小块表达式的版本	构建 固定大小块表达式的版本
包含第一个`n`元素的块*	vector.head(n);	vector.head<n>();
包含最后一个`n`元素的块*	vector.tail(n);	vector.tail<n>();
包含`n`元素的块，从位置*开始`i`	vector.segment(i,n);	vector.segment<n>(i);

下面给出了一个例子：

```cpp
#include <Eigen/Dense>
#include <iostream>

using namespace std;

int main()
{
    Eigen::ArrayXf v(6);
    v << 1, 2, 3, 4, 5, 6;
    cout << "v.head(3) =" << endl << v.head(3) << endl << endl;
    cout << "v.tail<3>() = " << endl << v.tail<3>() << endl << endl;
    v.segment(1,4) *= 2;
    cout << "after 'v.segment(1,4) *= 2', v =" << endl << v << endl;
}
```

输出：

```
v.head(3) =
1
2
3

v.tail<3>() =
4
5
6
```

```
10
11 after 'v.segment(1,4) *= 2', v =
12  1
13  4
14  6
15  8
16 10
17  6
```