别名

密集矩阵和数组操作

在 Eigen 中，别名是指在赋值运算符的左侧和右侧出现相同矩阵（或数组或向量）的赋值语句。语句类似`mat = 2 * mat;`或`mat = mat.transpose();`表现出别名。第一个示例中的混叠是无害的，但第二个示例中的混叠会导致意外结果。本页解释了混叠是什么、何时有害以及如何处理。

例子

这是一个展示混叠的简单示例：

例子：

```
MatrixXi mat(3,3);
mat << 1, 2, 3,   4, 5, 6,   7, 8, 9;
cout << "Here is the matrix mat:\n" << mat << endl;

// This assignment shows the aliasing problem
mat.bottomRightCorner(2,2) = mat.topLeftCorner(2,2);
cout << "After the assignment, mat = \n" << mat << endl;
```

输出：

```
Here is the matrix mat:
1 2 3
4 5 6
7 8 9
After the assignment, mat =
1 2 3
4 1 2
7 4 1
```

输出不是人们所期望的。问题是赋值

```
mat.bottomRightCorner(2,2) = mat.topLeftCorner(2,2);
```

此分配表现出混叠：系数`mat(1,1)`出现在`mat.bottomRightCorner(2,2)`分配左侧的块`mat.topLeftCorner(2,2)`和右侧的块中。赋值之后，右下角的(2,2)项应该`mat(1,1)`是赋值之前的值，也就是5。但是，输出显示`mat(2,2)`实际上是1。问题是Eigen使用了惰性求值（参见Expression Eigen 中的模板）用于`mat.topLeftCorner(2,2)`. 结果类似于

```
mat(1,1) = mat(0,0);
mat(1,2) = mat(0,1);
mat(2,1) = mat(1,0);
mat(2,2) = mat(1,1);
```

因此，`mat(2,2)`被分配了*新*值`mat(1,1)`而不是旧值。下一节将解释如何通过调用eval()来解决这个问题。

尝试缩小矩阵时，混叠会更自然地发生。例如，表达式`vec = vec.head(n)`和`mat = mat.block(i,j,r,c)`表现出混叠。

一般来说，编译时无法检测到混叠：如果`mat`第一个示例中的块大一点，那么块就不会重叠，也不会有混叠问题。然而，Eigen 确实检测到一些混叠实例，尽管是在运行时。矩阵和向量算法中提到了以下展示混叠的示例：

例子

```
Matrix2i a;
a << 1, 2, 3, 4;
cout << "Here is the matrix a:\n" << a << endl;

a = a.transpose(); // !!! do NOT do this !!!
cout << "and the result of the aliasing effect:\n" << a << endl;
```

输出

```
Here is the matrix a:
1 2
3 4
and the result of the aliasing effect:  // d
1 2
3 4
```

同样，输出显示了混叠问题。但是，默认情况下，Eigen 使用运行时断言来检测这一点，并以类似的消息退出

```
void Eigen::DenseBase<Derived>::checkTransposeAliasing(const OtherDerived&)
const
[with OtherDerived = Eigen::Transpose<Eigen::Matrix<int, 2, 2, 0, 2, 2> >,
Derived = Eigen::Matrix<int, 2, 2, 0, 2, 2>]:
Assertion
`(!internal::check_transpose_aliasing_selector<Scalar,internal::blas_traits<D
erived>::IsTransposed,OtherDerived>::run(internal::extract_data(derived()),
other))
&& "aliasing detected during transposition, use transposeInPlace() or
evaluate the rhs into a temporary using .eval()"' failed.
```

用户可以通过定义 EIGEN_NO_DEBUG 宏来关闭 Eigen 的运行时断言来检测此别名问题，并且上述程序是在关闭此宏的情况下编译的，以说明别名问题。见断言关于本征的运行时断言的更多信息。

解决混叠问题

如果您了解混叠问题的原因，那么很明显必须采取什么措施来解决它：E**igen 必须将右侧完全评估为临时矩阵/数组，然后将其分配给左侧。**函数eval()正是这样做的。

例如，这是上面第一个示例的更正版本：

例子

```
MatrixXi mat(3,3);
mat << 1, 2, 3,   4, 5, 6,   7, 8, 9;
cout << "Here is the matrix mat:\n" << mat << endl;

// The eval() solves the aliasing problem
mat.bottomRightCorner(2,2) = mat.topLeftCorner(2,2).eval();
cout << "After the assignment, mat = \n" << mat << endl;
```

输出

```
Here is the matrix mat:
1 2 3
4 5 6
7 8 9
After the assignment, mat =
1 2 3
4 1 2
7 4 5
```

现在，`mat(2,2)`赋值后等于 5，这是应该的。

相同的解决方案也适用于第二示例中，与转置：只需更换线`a = a.transpose();`用`a = a.transpose().eval();`。但是，在这种常见情况下，有一个更好的解决方案。Eigen 提供了专用函数 transposeInPlace()，它用转置替换矩阵。这如下所示：

例子

```
MatrixXf a(2,3); a << 1, 2, 3, 4, 5, 6;
cout << "Here is the initial matrix a:\n" << a << endl;

a.transposeInPlace();
cout << "and after being transposed:\n" << a << endl;
```

输出

```
Here is the initial matrix a:
1 2 3
4 5 6
and after being transposed:
1 4
2 5
3 6
```

如果 xxxInPlace() 函数可用，那么最好使用它，因为它更清楚地表明您在做什么。这也可能允许 Eigen 更积极地优化。这些是提供的一些 xxxInPlace() 函数：

原创功能	就地功能
MatrixBase::adjoint()	MatrixBase::adjointInPlace()
DenseBase::reverse()	DenseBase::reverseInPlace()
LDLT::solve()	LDLT::solveInPlace()
LLT::solve()	LLT::solveInPlace()
TriangularView::solve()	TriangularView::solveInPlace()
DenseBase::transpose()	DenseBase::transposeInPlace()

在矩阵或向量使用类似的表达式收缩的特殊情况下`vec = vec.head(n)`，您可以使用conservativeResize()。

别名和组件操作

如上所述，如果相同的矩阵或数组同时出现在赋值运算符的左侧和右侧，则可能会很危险，因此通常需要显式评估右侧。但是，应用逐组件操作（例如矩阵加法、标量乘法和数组乘法）是安全的。

以下示例仅包含组件操作。因此，即使赋值的两边出现相同的矩阵，也不需要eval()。

例子

```
MatrixXf mat(2,2);
mat << 1, 2,  4, 7;
cout << "Here is the matrix mat:\n" << mat << endl << endl;

mat = 2 * mat;
cout << "After 'mat = 2 * mat', mat = \n" << mat << endl << endl;


mat = mat - MatrixXf::Identity(2,2);
cout << "After the subtraction, it becomes\n" << mat << endl << endl;


ArrayXXf arr = mat;
arr = arr.square();
cout << "After squaring, it becomes\n" << arr << endl << endl;

// Combining all operations in one statement:
mat << 1, 2,  4, 7;
mat = (2 * mat - MatrixXf::Identity(2,2)).array().square();
cout << "Doing everything at once yields\n" << mat << endl << endl;
```

输出

```
Here is the matrix mat:
1 2
4 7

After 'mat = 2 * mat', mat =
 2  4
 8 14

```

```
After the subtraction, it becomes
 1  4
 8 13

After squaring, it becomes
  1  16
 64 169

Doing everything at once yields
  1  16
 64 169
```

一般来说，**如果右侧表达式的 (i, j) 项仅取决于左侧矩阵或数组的 (i, j) 项，而不取决于任何其他项，则赋值是安全的条目。**在这种情况下，没有必要明确评估右侧。

混叠和矩阵乘法

在目标矩阵未调整大小的情况下，矩阵乘法是 Eigen 中唯一默认假定别名的操作。因此，如果`matA`是**平方**矩阵，则该语句`matA = matA * matA;`是安全的。Eigen 中的所有其他操作都假设没有混叠问题，因为结果被分配给不同的矩阵或因为它是一个组件操作。

例子

```
MatrixXf matA(2,2);
matA << 2, 0, 0, 2;
matA = matA * matA;
cout << matA;
```

输出

```
4 0
0 4
```

然而，这是有代价的。执行表达式时`matA = matA * matA`，Eigen`matA`在计算后分配给的临时矩阵中计算乘积。这可以。但是当产品被分配到不同的矩阵（例如，`matB = matA * matA`）时，Eigen 会做同样的事情。在这种情况下，将乘积直接计算为更有效，`matB`而不是先将其计算为临时矩阵并将该矩阵复制到`matB`。

用户可以与该指示noalias()函数不存在混叠，如下：`matB.noalias() = matA * matA`。这允许 Eigen 将矩阵乘积`matA * matA`直接评估为`matB`。

例子

```
MatrixXf matA(2,2), matB(2,2);
matA << 2, 0,  0, 2;

// Simple but not quite as efficient
matB = matA * matA;
cout << matB << endl << endl;

// More complicated but also more efficient
matB.noalias() = matA * matA;
cout << matB;
```

输出

```
4 0
0 4

4 0
0 4
```

当然，您不应该`noalias()`在实际上发生混叠时使用。如果这样做，那么您可能会得到错误的结果：

例子

```
MatrixXf matA(2,2);
matA << 2, 0,  0, 2;
matA.noalias() = matA * matA;
cout << matA;
```

输出

```
4 0
0 4
```

此外，从Eigen 3.3开始，如果调整了目标矩阵的大小并且产品没有直接分配给目标，则**不会**假设混叠。因此，下面的例子也是错误的：

例子

```
MatrixXf A(2,2), B(3,2);
B << 2, 0,  0, 3, 1, 1;
A << 2, 0, 0, -2;
A = (B * A).cwiseAbs();
cout << A;
```

输出

```
4 0
0 6
2 2
```

至于任何别名问题，您可以通过在赋值之前显式评估表达式来解决它：

例子

```
MatrixXf A(2,2), B(3,2);
B << 2, 0,  0, 3, 1, 1;
A << 2, 0, 0, -2;
A = (B * A).eval().cwiseAbs();
cout << A;
```

输出

```
4 0
0 6
2 2
```

概括

当相同的矩阵或数组系数同时出现在赋值运算符的左侧和右侧时，就会发生混叠。

- 混叠对系数计算是无害的；这包括标量乘法和矩阵或数组加法。
- 当您将两个矩阵相乘时，Eigen 假设发生了混叠。如果您知道没有别名，那么您可以使用 noalias()。
- 在所有其他情况下，Eigen 假设不存在混叠问题，因此如果确实发生混叠，则会给出错误的结果。为了防止这种情况，您必须使用eval()或 xxxInPlace() 函数之一。