常见的陷阱

模板方法的编译错误

请参阅此页面。

别名

不要错过这个关于别名的页面，特别是如果你在目标出现在表达式右侧的语句中得到错误的结果。

对齐问题（运行时断言）

Eigen 进行显式矢量化，虽然许多用户对此表示赞赏，但在数据对齐受到损害的特殊情况下，这也会导致一些问题。事实上，在 C++17 之前，C++ 对显式数据对齐的支持不够好。在这种情况下，您的程序遇到断言失败（即"受控崩溃"），并显示一条消息，告诉您查阅此页面：

http: //eigen.tuxfamily.org/dox/group__TopicUnalignedArrayAssert.html

看看它，看看自己是否可以应对。它包含有关如何处理该问题的每个已知原因的详细信息。

现在，如果您不关心矢量化并且不想为这些对齐问题烦恼怎么办？然后阅读如何摆脱它们。

C++11 和 auto 关键字

简而言之：不要在 Eigen 的表达式中使用 auto 关键字，除非您 100% 确定自己在做什么。特别是，不要使用 auto 关键字代替 `Matrix<>` 类型。下面是一个例子：

```
MatrixXd A, B;
auto C = A * B;
for(...) { ... w = C * v;  ...}
```

在这个例子中， C 的类型不是 `MatrixXd` 类型而是一个抽象表达式，表示矩阵乘积并存储对 `A` 和 `B` 的引用。因此，for 循环的每一次迭代都会执行多次 `A * B` 的乘积。此外，如果 `A` 或的系数 `B` 在迭代过程中发生变化， `C` 则将计算为不同的值，如下例所示：

```
MatrixXd A = ..., B = ...;
auto C = A * B;
MatrixXd R1 = C;
A = ...;
MatrixXd R2 = C;
```

我们最终得到 `R1` ≠ `R2` 。

这是导致段错误的另一个示例：

```
auto C = ((A+B).eval()).transpose();
// 用 C 做一些事情
```

问题是`eval()`返回一个临时对象（在本例中为 a `MatrixXd`），然后由`Transpose<>`表达式引用。但是，这个临时文件在第一行之后被删除，然后`C`表达式引用了一个死对象。一种可能的解决方法是应用`eval()`整个表达式：

```
auto C = (A+B).transpose().eval();
```

当 Eigen 自动计算子表达式时，可能会出现相同的问题，如下例所示：

```
VectorXd u, v;
auto C = u + (A*v).normalized();
// 用 C 做一些事情
```

在这里，该`normalized()`方法必须评估昂贵的产品，`A*v`以避免对其进行两次评估。同样，一种可能的解决方法是``在整个表达式上调用.eval()：

```
auto C = (u + (A*v).normalized()).eval();
```

在这种情况下，`C`将是一个常规`VectorXd`对象。请注意，DenseBase::eval()足够智能，可以在底层表达式已经是普通的`Matrix<>`.

头文件问题（编译失败）

对于所有库，必须检查要包含哪些头文件的文档。Eigen 也是如此，但稍差一点：对于 Eigen，类中的方法可能需要`#include`比类本身需要的额外内容！例如，如果您想`cross()`在向量上使用该方法（它计算叉积），那么您需要：

```
#include<Eigen/Geometry>
```

我们试图始终记录这一点，但如果我们忘记了某个事件，请告诉我们。

三元运算符

简而言之：避免`(COND ? THEN : ELSE)`在`THEN`和`ELSE`语句的 Eigen 表达式中使用三元运算符。要了解原因，让我们考虑以下示例：

```
Vector3f A;
A << 1, 2, 3;
Vector3f B = ((1 < 0) ? (A.reverse()) : A);
```

此示例将返回`B = 3, 2, 1`. 你明白为什么吗？原因是在 C++ 中，`ELSE`语句的类型是从`THEN`表达式的类型推断出来的，这样两者都匹配。由于`THEN`是`Reverse<Vector3f>`，`ELSE`语句 A 被转换为 a `Reverse<Vector3f>`，因此编译器生成：

```
Vector3f B = ((1 < 0) ? (A.reverse()) : Reverse<Vector3f>(A));
```

在这种非常特殊的情况下，一种解决方法是为`THEN`语句调用 A.reverse().eval()，但最安全和最快的方法实际上是避免使用 Eigen 表达式使用此三元运算符并使用 if/else 构造。

传值

如果您不知道为什么 Eigen 按值传递是错误的，请先阅读此页面。

虽然您可能非常小心并小心确保所有显式使用 Eigen 类型的代码都是按引用传递的，但您必须注意在编译时定义参数类型的模板。

如果模板有一个函数，它接受按值传递的参数，并且相关的模板参数最终是一个特征类型，那么您当然会遇到与在显式定义的函数中通过引用传递特征类型相同的对齐问题.

将 Eigen 类型与其他第三方库甚至 STL 一起使用可能会出现相同的问题。`boost::bind` 例如使用传值在返回的函子中存储参数。这当然会有问题。

至少有两种方法可以解决这个问题：

- 如果您传递的值保证在函子的整个生命周期内都存在，您可以使用 boost::ref() 来包装您传递给 boost::bind 的值。通常，这不是堆栈上值的解决方案，就好像函子曾经被传递到较低或独立的作用域一样，在尝试使用对象时，对象可能已经消失。
- 另一种选择是让你的函数采用一个引用计数指针，如 boost::shared_ptr 作为参数。这避免了需要担心管理被传递对象的生命周期。

具有布尔系数的矩阵

当前使用`Matrix`布尔系数的行为不一致，并且在Eigen 的未来版本中可能会发生变化，因此请谨慎使用！

这种不一致的一个简单例子是

```
template<int Size>
void foo() {
    Eigen::Matrix<bool, Size, Size> A, B, C;
    A.setOnes();
    B.setOnes();

    C = A * B - A * B;
    std::cout << C << "\n";
}
```

因为调用`foo<3>()`打印零矩阵而调用`foo<10>()`打印单位矩阵。