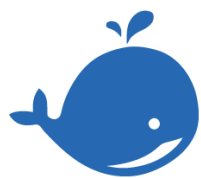


第二课-张量(Tensor)的设计与实现



Datawhale

KuiperInfer

这节课我们将一起学习张量，张量本质上讲就是一个多维数组，用于在算子之间传递数据，而这需要张量不仅仅只有存放数据的功能，我们还需要对张量的操纵方法等进行定义。

本文赞助方：datawhale

本文作者：[傅莘莘](#)

特别感谢：[散步](#)

在上一节课中，我们进行了简单的矩阵计算实践；但在深度学习的计算中，我们往往采用的是多种多样的矩阵进行运算，而这些矩阵的集合体通常被称为张量。我们可以试着构思一个张量应该有哪些成分：

思考题

请你在此暂停阅读，写出你自己对张量类设计的思考，需要有那些数据成员和方法（比如一个张量需要被创建，且能够支持形状的获取），并在课程结束后对比自己的设计和本文的设计，看看区别在哪。

对于一个张量类而言，数据将被设计成依次摆放的三维格式，分别是 `channels` (通道数), `rows` (行数), `cols` (列数)。一个张量类主要由以下部分组成：

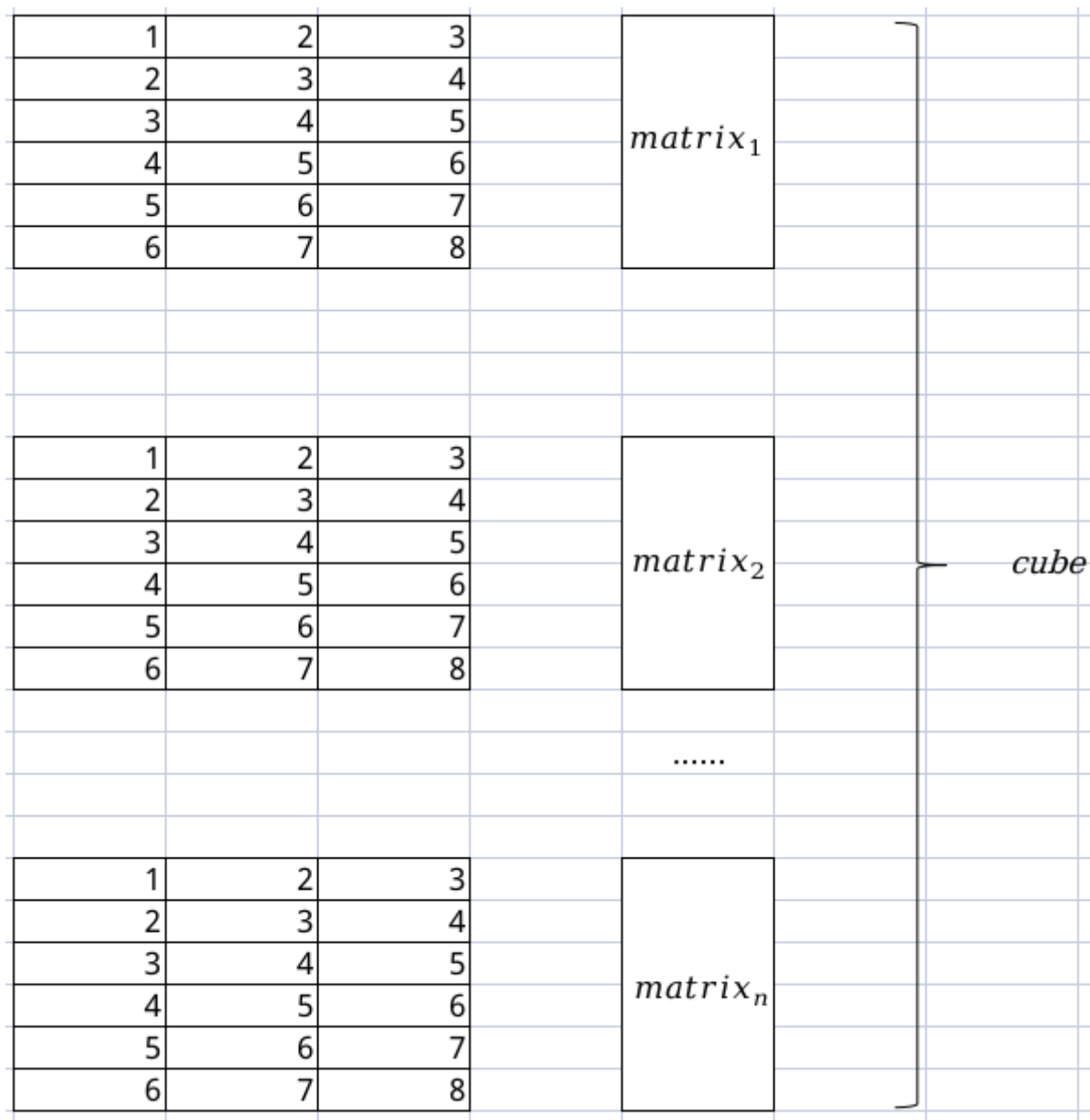
1. 数据本身存储在该类的数据空间中，数据可包括双精度(double)、单精度(float)或整型(int)。
2. 为了处理多维张量数据，需要使用 shape 变量来存储张量的维度信息。例如，对于一个维度为 3，长和宽均为 224 的张量，其维度信息可以表示为 (3, 224, 224)。
3. 张量类中定义了多个类方法，如返回张量的宽度、高度、填充数据和张量变形(reshape)等操作。

张量类的设计

为了更好地满足计算密集型任务的需求，一个张量类不仅需要在软件工程的层面上优化对外接口，还需要提供高效的矩阵相乘等算法实现。尤其是对于深度学习推理等任务来说，高效实现这些算法至关重要。

然而，从头设计一个张量类具有较大的编码难度，因此，在本项目中，我们选择在 `arma::fcube`（三维矩阵）的基础上进行开发。

如下图所示，三维的 `arma::fcube` 是由多个二维矩阵 `matrix`（即上一节课中介绍的 `arma::fmat`）沿通道维度叠加得到。在此基础上，我们的张量类将在叠加而成的三维矩阵 `arma::fcube` 的基础上提供扩充和封装，以使其更适用于我们的推理框架项目。



接下来我们可以一起来看看有关 `fcube` 和张量类（Tensor）的关系，不难看出 `data_` 作为一个 `fcube` 类提供了**数据管理和维护**的功能。

对于这样的一个Tensor类，我们主要做了以下的工作：

1. 提供对外的接口，对外接口由 `Tensor` 类在 `fcube` 类的基础上进行提供，以供用户更好地访问多维数据。
2. 封装矩阵相关的计算功能，这样一来不仅有更友好的数据访问和使用方式，也能有高效的矩阵算法实现。

```
template <>
class Tensor<float> {
public:
    uint32_t rows() const;
    uint32_t cols() const;
    uint32_t channels() const;
```

```

uint32_t size() const;
void set_data(const arma::fcube& data);
...
...
...
private:
    std::vector<uint32_t> raw_shapes_; // 张量数据的实际尺寸大小
    arma::fcube data_;                // 张量数据
};

```

接下来我们可以直观的感受一下 `arma::fmat` 与 `arma::fcube` 的区别。我们从前文中可以知道，矩阵类的维度是二维的，具有行宽等属性，以下是一个矩阵类数据分布的图示：

0	7
5	3
1	8
6	4
2	9

以下的每一个矩阵都是4行3列大小的矩阵，即 `4 x 3` 矩阵。上文提到，一个三维矩阵(`arma::fcube`)是由多个二维矩阵沿着通道维度叠加而成的。因此，以下的三维矩阵 `fcube` 的形状为 `(3, 4, 3)`，数据分布如下图所示：

1	5	9
2	6	10
3	7	11
4	8	12
1	5	9
2	6	10
3	7	11
4	8	12
1	5	9
2	6	10
3	7	11
4	8	12

有了张量的形状认知，下一步就是思考如何将数据存入到各式各样的的张量数据之中，不同的存放顺序很可能会对我们的计算过程造成影响。

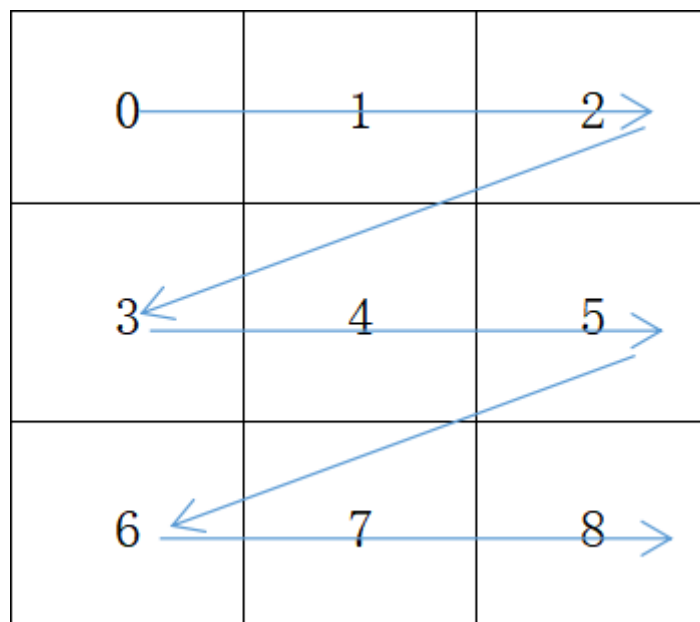
数据的摆放顺序

矩阵存储有两种形式：行主序和列主序。

行主序

对于一组数据，在矩阵中如果按行摆放，直到该行摆满后再到下一行摆放，并以此类推，这种存储数据的方式被称为行主序（Row-major order）

假如我们现在**有一组数据的值是0到8**，这9个数据在一个行主序的 `3 x 3` 矩阵中有如下的排布形式，其中箭头指示了内存地址的增长方向。从图中可以看出，内存地址增长的方向先是横向，然后是纵向，呈Z字形。

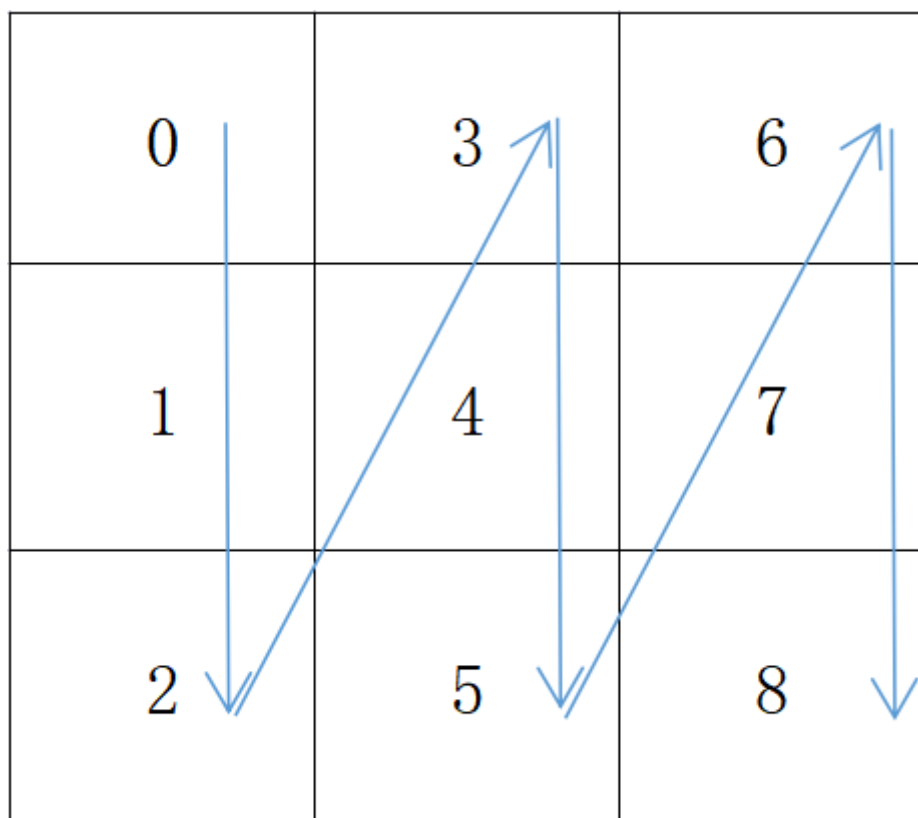


列主序

对于一组数据，在矩阵中如果依次存放顺序是先摆满一列，将剩余的数据依次存放到下一列，并以此类推，按照这种方式摆放的形式被称为列主序（Column-major order）。

同理，我们现在有9个数据，依次为0到8，将它摆放到一个列主序 3×3 的矩阵当中，并有如下的形式，其中箭头指示了内存地址的增长方向。

从图中可以看出，内存地址增长的方向先是纵向，然后是横向，呈倒Z字形。在 `armadillo` 中默认的顺序就是列主序的，而Pytorch张量默认顺序是行主序的，所以我们在程序中需要进行一定适应和调整。



张量类(tensor)方法概述

了解了基础的张量数据格式，接下来便需要思考如何实现一个兼容不同计算需要的张量，需要设计什么方法可以让他更好的被使用。在本节的内容中，我们会对张量类实现的主要方法进行列举和讲解，并会保留几个方法来让同学们亲自动手实现，以此来加深理解的过程。

创建张量

首先，在所有事情开始前，我们需要创建一个张量。在创建张量——这一多维矩阵的过程中，自然而然想到我们需要用一个 `raw_shapes` 变量来存储张量的维度，而不同维度将决定了 `arma::fcube` 的具体结构。我们需要根据输入的维度信息创建相应维度的 `arma::fcube`，且创建一个用于存储维度的变量。

- 如果张量是1维的，则 `raw_shapes` 的长度就等于1；
- 如果张量是2维的，则 `raw_shapes` 的长度就等于2，以此类推；
- 在创建3维张量时，则 `raw_shapes` 的长度为3；

值得注意的是，如果当 `channel` 和 `rows` 同时等于1时，`raw_shapes` 的长度也会是1，表示此时 `Tensor` 是一维的；而当 `channel` 等于1时，`raw_shapes` 的长度等于2，表示此时 `Tensor` 是二维的。

创建1维张量

```
Tensor<float>::Tensor(uint32_t size) {  
    data_ = arma::fcube(1, size, 1); // 传入的参数依次是，rows  
    cols channels  
    this->raw_shapes_ = std::vector<uint32_t>{size};  
}
```

创建2维张量

```
Tensor<float>::Tensor(uint32_t rows, uint32_t cols) {  
    data_ = arma::fcube(rows, cols, 1); // 传入的参数依次是，  
    rows cols channels  
    this->raw_shapes_ = std::vector<uint32_t>{rows, cols};  
}
```

创建3维张量


```

Tensor<float>::Tensor(uint32_t channels, uint32_t rows,
uint32_t cols) {
    data_ = arma::fcube(rows, cols, channels);
    if (channels == 1 && rows == 1) {
        // 当channel和rows同时等于1时，raw_shapes的长度也会是1，表示
        此时Tensor是一维的
        this->raw_shapes_ = std::vector<uint32_t>{cols};
    } else if (channels == 1) {
        // 当channel等于1时，raw_shapes的长度等于2，表示此时Tensor是
        二维的
        this->raw_shapes_ = std::vector<uint32_t>{rows, cols};
    } else {
        // 在创建3维张量时，则raw_shapes的长度为3，表示此时Tensor是三
        维的
        this->raw_shapes_ = std::vector<uint32_t>{channels,
rows, cols};
    }
}

```

返回张量的维度信息

```

uint32_t Tensor<float>::rows() const {
    CHECK(!this->data_.empty());
    return this->data_.n_rows;
}

uint32_t Tensor<float>::cols() const {
    CHECK(!this->data_.empty());
    return this->data_.n_cols;
}

uint32_t Tensor<float>::channels() const {
    CHECK(!this->data_.empty());
    return this->data_.n_slices;
}

uint32_t Tensor<float>::size() const {
    CHECK(!this->data_.empty());
    return this->data_.size();
}

```

```
}
```

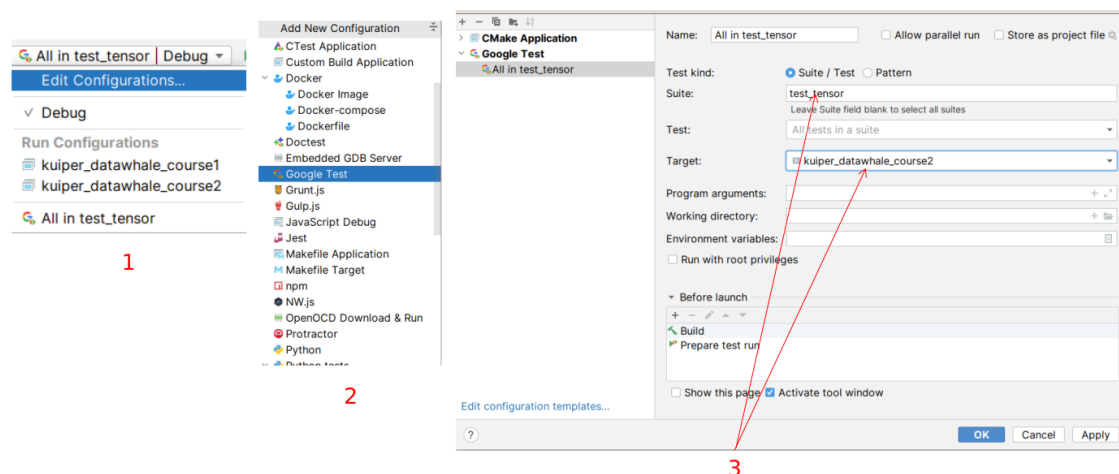
这四个方法分别返回张量的行数(rows)、列数(cols)、维度(channels)以及张量中数据的总数量(size)。假设我们有一个大小为 $(3 \times 3 \times 2)$ 的张量数据。

```
Tensor<float> tensor(3,3,2);
tensor.rows(); // 返回3
tensor.cols(); // 返回3
tensor.channels() // 返回2
tensor.size(); // 返回18
```

善用单元测试进行调试

1. 在开始之前，记得首先使用 `git pull` 命令更新本项目(<https://github.com/zjhelloworld/kuiperdatawhale>)的代码，本节课的代码位于 `course2` 目录中；
2. 随后在 Clion 中使用 `reload cmake project` 重新加载项目；如果发生头文件缺失的问题，可以使用 `resync with remote hosts` 同步头文件。
3. 本小节中与单元测试相关的代码存在于 `test_create_tensor.cpp` 文件中。

配置方法如下，具体操作请看视频操作，其中suite中的值来自于下方代码中的第一个参数。



如果你想要单独运行某个单元测试，也可以点击下方图示中的三角形来启动该操作，具体操作请看视频操作。

```
TEST(test_tensor, tensor_init1D) {
    using namespace kuiper_infer;
    Tensor<float> f1(4);
    f1.Fill(1.f);
    f1.Show();
}
```

Tips: 对于 `Tensor` 类可以使用 `Tensor::Show` 接口，对于 `arma::fmat` 类，可以使用 `LOG(INFO)<< fmat1;`

思考题

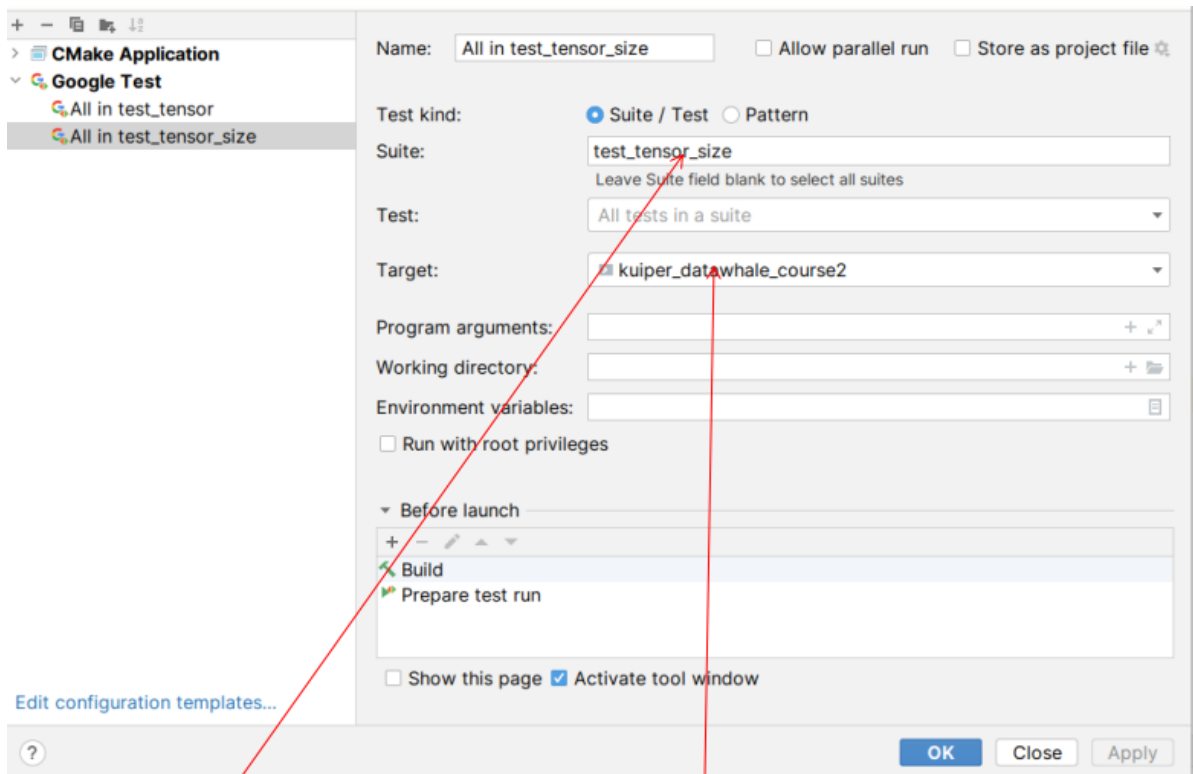
请你运行 `test_create_tensor.cpp` 中对应的所有单元测试，观察张量创建的情况和不同方法获取的结果。（如果有不认识的方法不用担心，接下来马上会介绍。）

我们强烈建议你调试课件中提及的所有单元测试，并仔细观察数据的逐步变化，更欢迎你自己编写单元测试对其他方法进行熟悉。

单元测试的配置

对应 `tensor_get_size.cpp`，配置方法同下，具体操作请看视频，其中 **suite** 中的值来自于下方代码中的第一个参数。

```
TEST(test_tensor_size, tensor_size1) {
    ...
}
```



尝试完简单的张量创建方法，我们来进一步看看我们需要对张量进行哪些更进一步的操作。

获得张量中的数据

```
const arma::fmat& Tensor<float>::slice(uint32_t channel)
const {
    CHECK_LT(channel, this->channels());
    return this->data_.slice(channel);
}
```

以上方法用于返回 `fcube` 变量中的第 `channel` 个矩阵。换句话说，一个 `fcube` 作为数据的实际存储者，由多个矩阵叠加而成。当我们调用 `slice` 方法时，它会返回其中的第 `channel` 个矩阵。

```
float Tensor<float>::at(uint32_t channel, uint32_t row,
uint32_t col) const {
    CHECK_LT(row, this->rows());
    CHECK_LT(col, this->cols());
    CHECK_LT(channel, this->channels());
    return this->data_.at(row, col, channel);
}
```

以上方法用于访问三维张量中第 (channel, row, col) 位置的对应数据。
对于以下的Tensor，访问 (1, 1, 1) 位置的元素，会得到6。

1	5	9
2	6	10
3	7	11
4	8	12
1	5	9
2	6	10
3	7	11
4	8	12
1	5	9
2	6	10
3	7	11
4	8	12

单元测试

对应 tensor_get_values.cpp，配置方法和之前的一样，不做赘述，具体请看视频操作。

```
TEST(test_tensor_values, tensor_values1) {  
    using namespace kuiper_infer;  
    Tensor<float> f1(2, 3, 4);  
    f1.Rand();  
    f1.Show();  
  
    LOG(INFO) << "Data in the first channel: " <<  
f1.slice(0); // 返回第一个通道(channel)中的数据  
    LOG(INFO) << "Data in the (1,1,1): " << f1.at(1, 1, 1);  
    // 返回(1,1,1)位置的数据  
}
```

张量的填充

```
void Tensor<float>::Fill(const std::vector<float>& values,
bool row_major) {
    CHECK(!this->data_.empty());
    const uint32_t total_elems = this->data_.size();
    CHECK_EQ(values.size(), total_elems);
    if (row_major) {
        const uint32_t rows = this->rows();
        const uint32_t cols = this->cols();
        const uint32_t planes = rows * cols;
        const uint32_t channels = this->data_.n_slices;

        for (uint32_t i = 0; i < channels; ++i) {
            auto& channel_data = this->data_.slice(i);
            const arma::fmat& channel_data_t =
                arma::fmat(values.data() + i * planes, this-
>cols(), this->rows());
            channel_data = channel_data_t.t();
        }
    } else {
        std::copy(values.begin(), values.end(), this-
>data_.memptr());
    }
}
```

如果函数中的 `row_major` 参数为 `true`，则表示按照行优先的顺序填充元素；如果该参数为 `false`，则将按照列优先的顺序填充元素。

思考题

请你在 `test_fill_reshape.cpp` 中找到 `Fill` 相关的测试并尝试将其改为列优先填充（欢迎你写出自己的单元测试），随后使用 `Tensor::Show` 接口方法查看这两种结果的区别。

有了直观感受后，接下来，我们将仔细思考行列主序填充的原理与差别之处。

行主序

如果我们采用**行主序**方式填充整个张量的数据区域，我们将把一组数据**按照行的顺序先填充第一行元素，然后按照顺序依次填充到第二行、第三行等位置。**

值得注意的是，**数据组的数量必须与待填充张量中的元素数量相同**，即 `values.size() == total_elems`。（软件工程的习惯很重要）

```
for (uint32_t i = 0; i < channels; ++i) {
    auto& channel_data = this->data_.slice(i);
    const arma::fmat& channel_data_t =
        arma::fmat(values.data() + i * planes, this-
>cols(), this->rows());
    channel_data = channel_data_t.t();
}
```

```
const arma::fmat& channel_data_t =
    arma::fmat(values.data() + i * planes, this->cols(), this-
>rows())
```

它会从 `values` 数组中拷贝一个通道（ $planes = rows \times cols$ ）个数据，并将其存储到 `channel_data_t` 中。但由于 `arma::fmat` 默认采用列主序的方式保存数据，因此我们还需要对它进行转置。

举个例子，假设我们从 `values` 中取得($rows \times cols = 4$) 个数据，排列如下： `[1, 2, 3, 4]`，那么它们放入到 `channel_data_t` 中，就会得到左边矩阵（因为是列主序的关系）。


接着我们需要对这个矩阵进行转置，得到 `channel_data`，以便将它变成了右边正确排布的矩阵。这样就完成了一个通道的数据的存储和转置。

1	3			1	2
2	4		transpose	3	4

```
for (uint32_t i = 0; i < channels; ++i)
```

然后再从 `values` 中再选取 4 个放置到第二个通道的数据中，重复此过程直到 `values` 中的 12 个元素都被放置到张量的存储空间中。第二通道数据填充的具体过程如下：

1	2		1	2	3	4	5	6	7	8
3	4									



列主序

对于列主序，因为 `Armadillo` 数学库默认的数据存储方式就是**列主序**的，所以直接将输入数据数组 `values` 拷贝到张量的存储空间上即可。

对应到代码就是：

```
std::copy(values.begin(), values.end(), this-
>data_.memptr());
```

单元测试

对应 `tensor_fill_reshape.cpp`，配置方法和之前的一样，不做赘述，具体请看视频操作。

```
TEST(test_fill_reshape, fill1) {
    /// 行主序的填充方式
    using namespace kuiper_infer;
    Tensor<float> f1(2, 3, 4);
    std::vector<float> values(2 * 3 * 4);
    // 将1到24填充到values中
    for (int i = 0; i < 24; ++i) {
        values.at(i) = float(i + 1);
    }
    f1.Fill(values);
    f1.Show();
}
```

该过程就是将顺序的数据 `1...24` 以**行主序**填充到一个 `(2, 3, 4)` 大小的张量中。

对张量中的元素依次处理

```
void Tensor<float>::Transform(const
std::function<float(float)>& filter) {
    CHECK(!this->data_.empty());
    this->data_.transform(filter);
}
```

`Transform`方法依次将张量中每个元素进行处理，处理的公式如下：

$$x = function(x)$$

单元测试

我们首先定义一个处理函数，这个处理函数是将值减去1。

```
float MinusOne(float value) { return value - 1.f; }
```

我们将这个处理函数应用于张量中的每一个数据，单元测试定义于 `tensor_transform.cpp` 中。

```
TEST(test_transform, transform1) {
    using namespace kuiper_infer;
    Tensor<float> f1(2, 3, 4);
    f1.Rand();
    f1.Show();
    f1.Transform(MinusOne);
    f1.Show();
}
```

对张量进行变形

```
void Tensor<float>::Reshape(const std::vector<uint32_t>&
shapes,
                                bool row_major) {
    CHECK(!this->data_.empty());
    CHECK(!shapes.empty());
    const uint32_t origin_size = this->size();
    const uint32_t current_size =
```

```

        std::accumulate(shapes.begin(), shapes.end(), 1,
std::multiplies());
    CHECK(shapes.size() <= 3);
    CHECK(current_size == origin_size);

    std::vector<float> values;
    if (row_major) {
        values = this->values(true);
    }
    if (shapes.size() == 3) {
        this->data_.reshape(shapes.at(1), shapes.at(2),
shapes.at(0));
        this->raw_shapes_ = {shapes.at(0), shapes.at(1),
shapes.at(2)};
    } else if (shapes.size() == 2) {
        this->data_.reshape(shapes.at(0), shapes.at(1), 1);
        this->raw_shapes_ = {shapes.at(0), shapes.at(1)};
    } else {
        this->data_.reshape(1, shapes.at(0), 1);
        this->raw_shapes_ = {shapes.at(0)};
    }

    if (row_major) {
        this->Fill(values, true);
    }
}

```

`reshape` 方法用于对张量的维度进行调整，例如张量原先的大小是 `(channel1, row1, col1)`，再进行 `reshape` 之后我们将张量的大小调整为 `(channel2, row2, col2)`。在调整的过程中，我们需要注意，前后的两组维度要满足以下的关系：

$$channel_1 \times rows_1 \times col_1 = channel_2 \times rows_2 \times col_2$$

行主序

同时，我们在 `reshape` 中提供两种不同模式，第一种是行优先的。

0	1	2	3	4
5	6	7	8	9

上图是 `reshape` 之前的张量数据分布，其中 `reshape` 之前的 `raw_shape` 等于 `(2, 5)`，也就是**该张量的通道数为 1，行数为 2，列数为 5**。

按照**行主序**取出后的 `values` 数组后，得到数据 `0...9`，可以使用 `reshape` 函数将张量形状改变为 `(5, 2)`。随后使用 `Fill` 函数**将取出的值按照行主序**填充到 改变过形状的张量中。具体代码如下所示：

```
std::vector<float> values;
if (row_major) {
    values = this->values(true);
}
...
...
改变张量的形状，从(2, 5)到(5, 2)
// 填充数据
if (row_major) {
    this->Fill(values, true);
}
```

0	1
2	3
4	5
6	7
8	9

列主序

列摆放的原则则是将元素按照先列后行的次序取出，取出后依次为 `(0, 5, 1, 6, 2, 7, 3, 8, 4, 9)`，并在取出后按照先列后行的次序摆放到 `(5, 2)` 的张量中，分布图例如下；

0	7
5	3
1	8
6	4
2	9

单元测试

对应 `tensor_fill_reshape.cpp` 中的 `reshape1` 函数，配置方法和之前的一样，不做赘述，具体请看视频操作。

```
TEST(test_fill_reshape, reshape1) {
    using namespace kuiper_infer;
    Tensor<float> f1(2, 3, 4);
    std::vector<float> values(2 * 3 * 4);
    // 将1到12填充到values中
    for (int i = 0; i < 24; ++i) {
        values.at(i) = float(i + 1);
    }
    f1.Fill(values);
    f1.Show();
    /// 将大小调整为(4, 3, 2)
    f1.Reshape({4, 3, 2}, true);
    f1.Show();
}
```

张量类的辅助函数

判断张量符合是否为空

```
bool Tensor<float>::empty() const { return this->data_.empty(); }
```

返回张量数据存储区域的起始地址

```
const float* Tensor<float>::raw_ptr() const {  
    CHECK(!this->data_.empty());  
    return this->data_.memptr();  
}
```

上文说到，张量类中数据存储由三维矩阵类(`fcube`)负责，所以在`raw_ptr`的目的就是返回数据存储的起始位置。

返回张量的shape

```
const std::vector<uint32_t>& Tensor<float>::raw_shapes()  
const {  
    CHECK(!this->raw_shapes_.empty());  
    CHECK_LE(this->raw_shapes_.size(), 3);  
    CHECK_GE(this->raw_shapes_.size(), 1);  
    return this->raw_shapes_;  
}
```

返回张量的三维形状 `[channels, rows, cols]`。

- 如果 `channels = 1` 并且 `rows = 1`，则 `raw_shapes` 返回一维形状 `[cols]`，张量是一个一维张量。
- 如果 `channels = 1`，则 `raw_shapes` 返回二维形状 `[rows, cols]`，张量是一个二维张量。
- 否则表明该 `Tensor` 就是一个三维张量，`raw_shapes` 返回三维形状 `[channels, rows, cols]`。

课堂作业

1. 自行编写 `Tensor::Flatten` 函数，对应测试用例 `test_homework.cpp` 中的 `homework1_flattenx`，这个函数的功能是将多维数据展开为一维，如下图所示：

1	2	3												
4	5	6		1	2	3	4	5	6	7	8	9		
7	8	9												

2. 自行编写 `Tensor::Padding` 函数，这个函数是在多维张量的周围做填充，如下图所示：

0	0	0	0	0		0	0	0	0	0		0	0	0	0	0
0	1	1	1	0		0	1	1	1	0		0	1	1	1	0
0	1	1	1	0		0	1	1	1	0		0	1	1	1	0
0	1	1	1	0		0	1	1	1	0		0	1	1	1	0
0	0	0	0	0		0	0	0	0	0		0	0	0	0	0

完成作业1, 2之后，请自行运行 `test_homework.cpp` 中的测试用例，如果通过测试用例表明编写的代码正确。