

源码阅读报告（一）：主要功能分析与建模

0>什么是pytorch

wiki百科上对此有描述：PyTorch 是一个基于 Torch 库的机器学习库，用于计算机视觉和自然语言处理等应用，最初由 Meta AI 开发，现在是 Linux 基金会的一部分。它是与 TensorFlow 并列的两个最受欢迎的深度学习库之一，提供根据修改后的 BSD 许可证发布的免费开源软件。尽管 Python 接口更加精致并且是开发的主要焦点，但 PyTorch 也有 C++ 接口。

PyTorch 的主要特点

- 动态计算图**：PyTorch 使用动态计算图（Dynamic Computational Graph），这意味着计算图在运行时是动态构建的。这使得调试和开发更加灵活和直观。
- 强大的 GPU 加速**：PyTorch 支持 GPU 加速，能够利用 CUDA 和 ROCm 后端进行高效的数值计算。
- 丰富的库和工具**：PyTorch 提供了丰富的库和工具，如 torchvision（用于计算机视觉）、torchtext（用于自然语言处理）和 torchaudio（用于音频处理）。
- 社区和生态系统**：PyTorch 拥有一个活跃的社区和广泛的生态系统，提供了大量的教程、示例和预训练模型。
- 与其他工具的集成**：PyTorch 可以与其他深度学习和机器学习工具（如 TensorBoard、ONNX）无缝集成，方便模型的可视化和部署。

PyTorch 的应用领域

- 计算机视觉**：如图像分类、目标检测、图像生成等。
- 自然语言处理**：如文本分类、机器翻译、文本生成等。
- 强化学习**：如策略优化、价值函数估计等。
- 生成对抗网络 (GANs)**：用于生成逼真的图像、视频和音频。
- 时间序列分析**：如预测、异常检测等。

PyTorch 的基本组件

- 张量 (Tensor)**：PyTorch 的核心数据结构，类似于 NumPy 的 ndarray，但可以在 GPU 上进行加速计算。
- 自动微分 (Autograd)**：PyTorch 提供了自动微分功能，能够自动计算梯度，方便实现反向传播算法。
- 神经网络模块 (torch.nn)**：提供了构建神经网络的基础模块和层，如全连接层、卷积层、循环层等。
- 优化器 (torch.optim)**：提供了常用的优化算法，如 SGD、Adam、RMSprop 等。
- 数据加载和预处理 (torch.utils.data)**：提供了数据加载和预处理的工具，如 DataLoader、Dataset 等。

PyTorch 示例代码

在这里给出一个简单的例子,笔者将展示pytorch的具体使用
首先引用torch包

```
import torch
```

我们将创建一个 3x3 的张量, 来进行后面的使用,

```
tensor = torch.rand(3, 3)
print("Original Tensor:")
print(tensor)
```

我们对这个变量做张量加法

```
tensor_add = tensor + tensor
print("\nTensor after addition:")
print(tensor_add)
```

接下来是乘法

```
tensor_mul = tensor * tensor
print("\nTensor after multiplication:")
print(tensor_mul)
```

运行这个代码, 得到:

```
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计> cd code_reading_reports
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计\code_reading_reports> cd code_reading_reports
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计\code_reading_reports\code_reading_reports> python test_codes.py
Original Tensor:
tensor([[0.3344, 0.7163, 0.4869],
        [0.4670, 0.2064, 0.6263],
        [0.6342, 0.9539, 0.0631]])

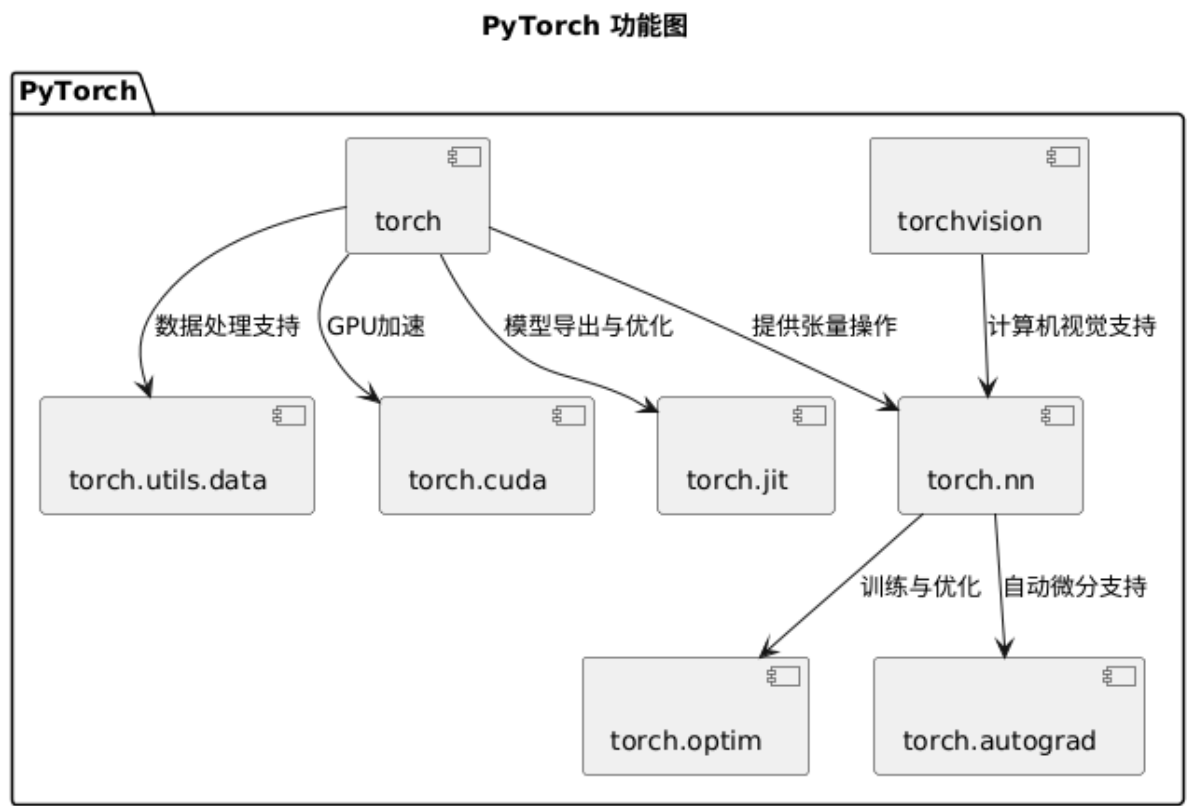
Tensor after addition:
tensor([[0.6687, 1.4327, 0.9738],
        [0.9340, 0.4129, 1.2527],
        [1.2684, 1.9079, 0.1262]])

Tensor after multiplication:
tensor([[0.1118, 0.5132, 0.2371],
        [0.2181, 0.0426, 0.3923],
        [0.4022, 0.9100, 0.0040]])
PS C:\Users\lenovo\Desktop\新建文件夹\学科资料\面向对象的程序设计\code_reading_reports\code_reading_reports> |
```

综上, 我们可以得出结论:PyTorch 是一个灵活且强大的深度学习框架, 支持动态计算图和 GPU 加速, 广泛应用于计算机视觉、自然语言处理等领域

1>主要功能分析与建模

使用在线绘制UML图的平台，绘制pytorch的UML图如下：



1.0>一个典型的pytorch类

在这里，我们将给出一个典型的 PyTorch 类的示例，该类实现了一个简单的神经网络模型，并使用 DataLoader 加载数据集进行训练和评估。

这里给出了一个自定义的数据集类 CustomDataset 和一个简单的神经网络模型 SimpleNN。

CustomDataset 类用于加载数据集，SimpleNN 类用于定义神经网络模型。

```
class CustomDataset(Dataset):
    def __init__(self, data, labels):
        self.data = data
        self.labels = labels

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = self.data[idx]
        label = self.labels[idx]
        return sample, label

class SimpleNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out = self.fc1(x)
```

```
out = self.relu(out)
out = self.fc2(out)
return out
```

1.1>需求建模

需求模型如下：

1.1.1> 系统功能：

这段代码的核心需求是定义自定义数据集和神经网络模型，以便进行机器学习任务。核心功能包括：

自定义数据集加载与索引访问

神经网络的前向传播定义

1.1.2> 角色与需求

角色：

开发者：需要灵活地定义数据集和模型，以便用于不同的机器学习任务。

模型训练框架（如 **PyTorch**）：需要从数据集中提取样本，并在前向传播时调用神经网络。

主要需求：

需求1：开发者能够自定义数据集，以便处理不同格式的数据源。

需求2：开发者能够定义一个简单的神经网络，以适应特定的输入输出需求。

需求3：训练框架能够通过数据索引从数据集中获取样本，并输入到神经网络进行训练和预测。

1.1.3> 用例分析

用例名称：加载自定义数据集

场景：

who: 开发者，`torch.utils.data.DataLoader`

where: 内存中保存的数据集对象

when: 训练开始时，或者需要从数据集中获取样本时

描述: 开发者通过 `CustomDataset` 类将数据和标签传入，然后通过数据加载器 `DataLoader` 进行批量加载。训练过程中，框架会根据索引从数据集中提取样本和标签。

用例名称：神经网络前向传播

场景：

who: 开发者，训练框架

where: 模型内部的各层网络，输入和输出之间

when: 当输入张量传入模型时（训练或推理阶段）

描述: 开发者定义了一个神经网络 `SimplENN`，框架在训练和推理阶段会将输入数据传入该网络，按前向传播流程依次通过各层，最终得到预测结果。

1.1.4>类模型

我们可以将代码中定义的类映射为系统中的实体，列出它们的属性和方法：

CustomDataset:

属性: `data`, `labels`

方法: `__len__()`, `__getitem__(idx)`

SimplENN:

属性: `fc1`, `relu`, `fc2`

方法: `forward(x)`

1.1.5> 交互模型

在系统中，`CustomDataset` 和 `SimplENN` 类通过训练框架进行交互：

`DataLoader` 会调用 `CustomDataset` 的 `__getitem__` 方法来获取训练样本。

训练框架将样本传递给 `SimplENN` 进行前向传播，调用 `forward` 方法得到输出。

1.1.6> 需求模型总结

自定义数据加载：系统允许开发者根据特定需求加载并索引数据集（需求1）。

网络结构灵活定义：系统支持开发者定义前馈神经网络，并可以根据任务灵活设定输入、隐藏层和输出大小（需求2）。

数据与模型的交互：数据集与神经网络通过 `PyTorch` 的 `DataLoader` 和 `forward` 方法进行交互，实现批量数据训练（需求3）。

[限制与缺陷]：

- 1：数据如果没有经过逻辑增强，需要进行额外的预处理，才能使用。
- 2：神经网络是静态的，无法根据集体任务调整网络层数。
- 3:使用上述代码进行训练的时候，没有纠错机制，发生错误，就会直接导致训练失败。

综合上述代码，简答归纳可以得到一个功能表格，展示如下：

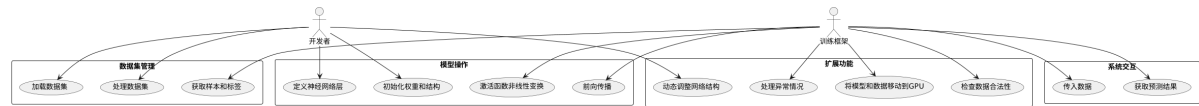
功能分类	动词	功能描述
数据集相关功能	加载	从内存中加载数据集
	处理	对数据集进行预处理（尚未实现）
	索引	通过索引获取样本及其标签
模型相关功能	定义	定义神经网络层
	初始化	初始化神经网络的权重和结构
	激活	通过激活函数（ReLU）进行非线性变换
	前向传播	计算输入数据通过网络的输出结果
系统交互功能	传入	将数据输入模型
	获取	通过索引获取样本和标签，供模型使用
	返回	返回模型的预测结果
限制相关功能	调整	动态调整网络结构（如层数、神经元数量）
	检查	检查输入输出数据的尺寸和合法性
	处理	处理错误或异常情况（如数据超出索引范围等）
	移动	将数据和模型移动到GPU加速设备

提取其中的关键词，我们初步挑选出其中的名词数据集相关功能：

1.2>提取关键词，画出对应的UML图

类别	关键词
数据集相关功能	内存, 数据集, 预处理, 样本, 标签, 索引
模型相关功能	神经网络, 权重, 结构, 激活函数, 非线性变换, 输入数据, 输出结果
系统交互功能	传入, 获取, 返回
限制相关功能	调整, 检查, 处理, 移动

得到的UML图如下:



在这个节点，已经创建一个 pytorch 涉及的 simpleNN 与 CustomDataset 主要类，这些类已经能初步完成神经网络的训练，我们只需要调用 pytorch 的包就可以快速实现我们想要的神经网络，利用我们的数据集来训练。不得不说，项目在屏蔽底层细节、提升用户体验上做的确实十分到位。让人可以专注于神经网络的实现，而不必深入具体的代码实现。

1.3>实际执行和具体细节探究。

由于pytorch代码实在过于庞杂和笔者水平原因，我们仅在这里以点带面，找到一个更加简单的样例来探究pytorch如何实现细节封装，达到“高内聚，低耦合”的效果。

```
import torch
a = torch.rand(5)
b = a + a
print(b)
```

这段代码随机生成一个长度为五的一维张量，然后加倍，打印出来。很容易理解。

我们可以在site-packages下查看torch包包含的内容:

```
ls      contrib      fx      _lowrank.py      package      serialization.py      test
amp     cpu            hub.py      masked          _prims        share                testing
ao      cuda          include     _meta_registrations.py  _six.py       _torch_docs.py
_appdirs.py  _decomp      _init_.py  monitor         _prims_common  _sources.py         torch_version.py
autograd  _deploy.py   jit        multiprocessing  profiler        sparse              types.py
backends  _dispatch   jit_internal.py  nanotensor_internals.py  _pycache_     special             utils.py
bin      distributed  lazy        nested          pytorch_dispatcher.py  quantization        _utils_internal.py
_C       distutils    lib         nn              py_type        storage_docs.py     _utils.py
_C.python-38-x86_64-linux-gnu.so  fft          library.py  onnx            quasirandom.py    _subclassses        version.py
_C.flatbuffer  functional.py  linalg     _ops.py        random.py         _tensor_docs.py     _VF.py
_C.flatbuffer.python-38-x86_64-linux-gnu.so  _future_.py  linalg_utils.py  optim           _refs             _vmap_internals.py
_classes.py  _futures     _lobpcg.py  overrides.py    return_types.py   _tensor_str.py      _weights_only_unpickler.py
_config_.py
```

对比下方源码的torch目录:

```
# ls
abi-check.cpp      _config_.py      distributions     jit_internal.py  masked            package           _refs            storage.py        _utils_internal.py
amp               contrib           extension.h       lazy             _meta_registrations.py  _prims           return_types.py  _subclasses
cpu              cpu              ifconfig         _legacy          monitor           _pyscopes        return_types.py  _tensor_docs.py
_appdirs.py       csrc             functional.py     _functional.py   _multiprocessing  profiler          tensor.py        version.py
autograd          cuda            _future_.py     _lib64           _namedtensor_internals.py  _pycache        serialization    _VF.py
backends         custom_class_detail.h  _futures        library.h        _nested           _python_dispatcher.py  share           _vmem_internals.py
bin             custom_class.h    fx              library.py       nn               np               _six.py         testing
_cdecim          _decim           hub.py           _libaio.py       _onnx              _onnx_ops.py     _torch_docs.py  _torch_docs.py
_C_flatbuffer    _deploy.py       include          _liblinalg.py   _ops.py           _optim            _quasirandom.py  torch_version.py
_classes.py      _dispatch        jit             _init_.py       _lowrank.py       overrides.py      random.py        types.py
CMakelists.txt  distributed       jit             _jit_internal.py  _lowrank.py       overrides.py      README.txt       _utils
```

可以发现torch的wheel包安装的内容基本上就是torch目录下的python内容。而c++内容（csrc目录）并没有被复制过来，而是以编译好的动态库文件（`C.cpython-*.so`）代替。

在torch目录下的init.py文件中，可以看到将C(动态库)导入的地方：

```

4
3 # Appease the type checker; ordinarily this binding is inserted by the
2 # torch._C module initialization code in C
1 if TYPE_CHECKING:
97     import torch._C as C
1

```

简单来说：在pytorch中：PyTorch 的 Python 接口，它为用户提供了一组用于构建、训练和评估深度学习模型的工具。前端接口使用 Python 编写，因此可以轻松与其他 Python 库集成（如numpy等）。看到这里我们初步理解，pytorch代码编写时 `import torch` 的意思。

同时，pytorch为了性能也使用了很多C++的API，这主要是出于性能考虑，python的性能是没有C/C++出色的。这一部分其实也可以正常的C++API接口使用，在使用PyTorch C++时，只需要将项目动态库与头文件正确链接即可。PyTorch官方提供了一个libtorch项目编译cmake例子：

```

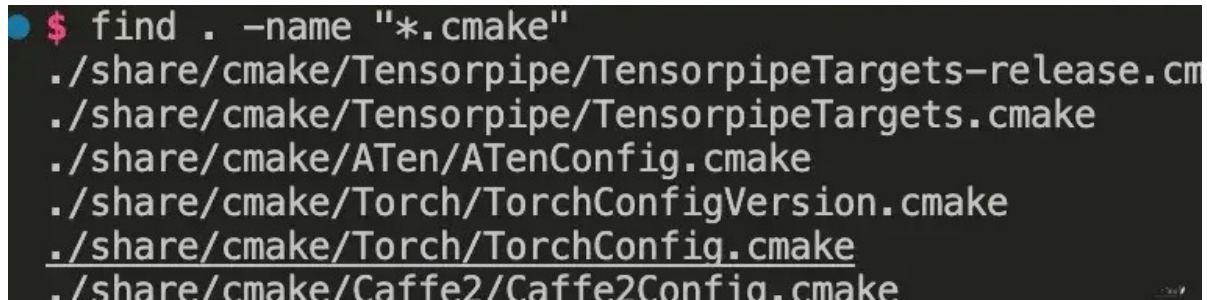
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(dcgan)

find_package(Torch REQUIRED)

add_executable(dcgan dcgan.cpp)
target_link_libraries(dcgan "${TORCH_LIBRARIES}")
set_property(TARGET dcgan PROPERTY CXX_STANDARD 14)

```

这里的find_package(Torch)这句是通过TorchConfig.cmake实现的库与头文件的链接，这是PyTorch提供的cmake文件，可以让我们快速实现c++项目编译。



```

$ find . -name "*.cmake"
./share/cmake/Tensorpipe/TensorpipeTargets-release.cmake
./share/cmake/Tensorpipe/TensorpipeTargets.cmake
./share/cmake/ATen/ATenConfig.cmake
./share/cmake/Torch/TorchConfigVersion.cmake
./share/cmake/Torch/TorchConfig.cmake
./share/cmake/Caffe2/Caffe2Config.cmake

```

torch的c++ API使用和python有一定区别，但在“形式”上大致相似，例如求tensor加法：

```

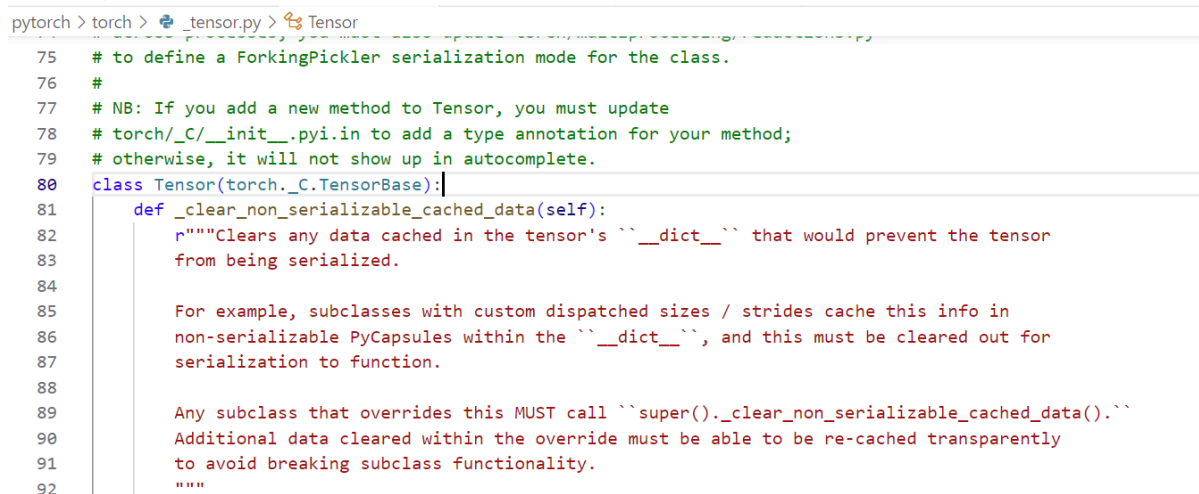
#include <torch/torch.h>
#include <iostream>

int main() {
    torch::Tensor a = torch::rand(10);
    auto b = a.add(a);
    std::cout << b << std::endl;
    return 0;
}

```

那么接下来的问题就自然而然：怎么将这两部分能够“丝滑合作”，达到我们想要的效果。

下面是PyTorch中的Tensor实现，我们可以看到它继承了C中的TensorBase



```

pytorch > torch > _tensor.pyi > Tensor
75 # to define a ForkingPickler serialization mode for the class.
76 #
77 # NB: If you add a new method to Tensor, you must update
78 # torch/_C/__init__.pyi.in to add a type annotation for your method;
79 # otherwise, it will not show up in autocomplete.
80 class Tensor(torch._C.TensorBase):
81     def _clear_non_serializable_cached_data(self):
82         r"""Clears any data cached in the tensor's ``__dict__`` that would prevent the tensor
83         from being serialized.
84
85         For example, subclasses with custom dispatched sizes / strides cache this info in
86         non-serializable PyCapsules within the ``__dict__``, and this must be cleared out for
87         serialization to function.
88
89         Any subclass that overrides this MUST call ``super()._clear_non_serializable_cached_data()``
90         Additional data cleared within the override must be able to be re-cached transparently
91         to avoid breaking subclass functionality.
92         """

```

在_C目录下init.pyi中，我们可以看到TensorBase的类。pyi文件是存根文件(stub file)，表示公共接口。其中有各种成员函数的定义，但是没有具体的实现。

在文件的注释中，我们可以看到其具体实现是在python_variable.cpp中。

pytorch > torch > _C > ≡ __init__.pyi.in

```
1724
1725 # Defined in torch/csrc/autograd/python_variable.cpp
1726 class TensorBase(metaclass=_TensorMeta):
1727     requires_grad: _bool
1728     retains_grad: _bool
1729     shape: Size
1730     data: Tensor
1731     names: List[str]
1732     device: _device
1733     dtype: _dtype
1734     layout: _layout
1735     real: Tensor
1736     imag: Tensor
1737     T: Tensor
1738     H: Tensor
1739     mT: Tensor
1740     mH: Tensor
1741     ndim: _int
1742     output_nr: _int
1743     _version: _int
1744     _base: Optional[Tensor]
1745     _cdata: _int
1746     grad_fn: Optional[_Node]
1747     _grad_fn: Any
1748     _grad: Optional[Tensor]
1749     grad: Optional[Tensor]
1750     _backward_hooks: Optional[Dict[_int, Callable[[Tensor], Optional[Tensor]]]]
1751     nbytes: _int
1752     itemsize: _int
1753
```

在python_variable.cpp文件中，我们可以看到PyTorch实际是用了pybind，将c++和Python进行交互的。


```

1834
1835 PyTypeObject THPVariableType = {
1836     PyVarObject_HEAD_INIT(&THPVariableMetaType, 0)
1837     "torch._C.TensorBase", /* tp_name */
1838     sizeof(THPVariable), /* tp_basicsize */
1839     0, /* tp_itemsize */
1840     // This is unspecified, because it is illegal to create a THPVariableTyp
1841     // directly. Subclasses will have their tp_dealloc set appropriately
1842     // by the metaclass
1843     nullptr, /* tp_dealloc */
1844     0, /* tp_vectorcall_offset */
1845     nullptr, /* tp_getattr */
1846     nullptr, /* tp_setattr */
1847     nullptr, /* tp_reserved */
1848     nullptr, /* tp_repr */
1849     nullptr, /* tp_as_number */
1850     nullptr, /* tp_as_sequence */
1851     &THPVariable_as_mapping, /* tp_as_mapping */
1852     nullptr, /* tp_hash */
1853     nullptr, /* tp_call */
1854     nullptr, /* tp_str */
1855     nullptr, /* tp_getattro */
1856     nullptr, /* tp_setattro */
1857     nullptr, /* tp_as_buffer */
1858     // NOLINTNEXTLINE(misc-redundant-expression)
1859     Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
1860     | Py_TPFLAGS_HAVE_GC, /* tp_flags */
1861     nullptr, /* tp_doc */
1862     // Also set by metaclass
1863     (traverseproc)THPFake_traverse, /* tp_traverse */

```

我们这里以Tensor.dtype为例，它的实现在THPVariable_properties中。

找到这个结构体的实现，可以看到dtype具体绑定了THPVariable_dtype()这个函数。因此在Python中执行tensor.dtype时，实际运行的就是这个c++函数。

pytorch > torch > csrc > autograd > [python_variable.cpp](#)

```
1835 PyTypeObject THPVariableType = {
1866     0, /* tp_weaklistoffset */
1867     nullptr, /* tp_iter */
1868     nullptr, /* tp_iternext */
1869     nullptr, /* tp_methods */
1870     nullptr, /* tp_members */
1871     THPVariable_properties, /* tp_getset */
1872     nullptr, /* tp_base */
1873     nullptr, /* tp_dict */
1874     nullptr, /* tp_descr_get */
1875     nullptr, /* tp_descr_set */
1876     0, /* tp_dictoffset */
1877     nullptr, /* tp_init */
1878     nullptr, /* tp_alloc */
1879     // Although new is provided here, it is illegal to call this with cls ==
1880     // THPVariableMeta. Instead, subclass it first and then construct it
1881     THPVariable_pynew, /* tp_new */
1882 };
1883
1884 {
1885     nullptr,
1886     nullptr,
1887     {"dtype", (getter)THPVariable_dtype, nullptr, nullptr, nullptr},
1888     {"layout", (getter)THPVariable_layout, nullptr, nullptr, nullptr},
1889     {"device", (getter)THPVariable_device, nullptr, nullptr, nullptr},
1890     {"ndim", (getter)THPVariable_get_ndim, nullptr, nullptr, nullptr},
1891     {"nbytes", (getter)THPVariable_get_nbytes, nullptr, nullptr, nullptr},
1892     {"itemsize", (getter)THPVariable_get_itemsize, nullptr, nullptr, nullptr},
1893     {"names",
1894      (getter)THPVariable_get_names,
1895      (setter)THPVariable_set_names,
```

PyTorch主要是python API与C++ API，在设计上采用Pythonic式的编程风格，可以让用户像使用python一样使用PyTorch，其对外也提供的C++接口，也可以一定程度上实现PyTorch的大部分功能。主要是通过pybind调用c++实现，具体是c++被编译成C.[**].so动态库，然后python调用torch.C实现调用c++中的函数。