

# 怒肝万字！详解 PyTorch 2.0 Dynamo 字节码，自顶向下，由浅入深

极市平台 2023-04-28 22:00:48 发表于广东 手机阅读 眼

以下文章来源于OpenMMLab，作者带来新知识的

 **OpenMMLab**  
构建国际领先的人工智能开源算法平台

↑ 点击蓝字 关注极市平台



作者 | HAOCHENYE (GitHub ID)

来源 | OpenMMLab

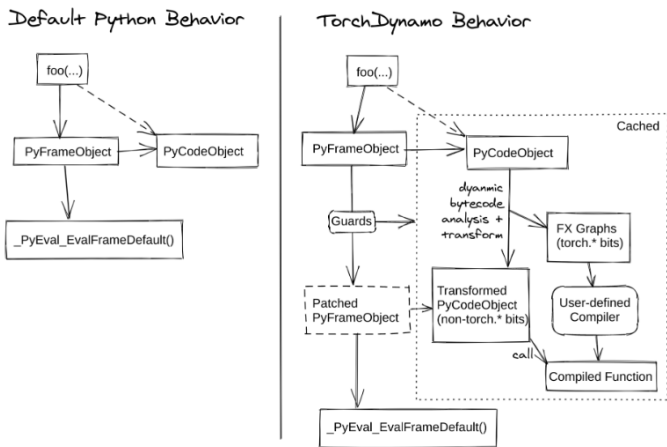
编辑 | 极市平台

### 极市导读

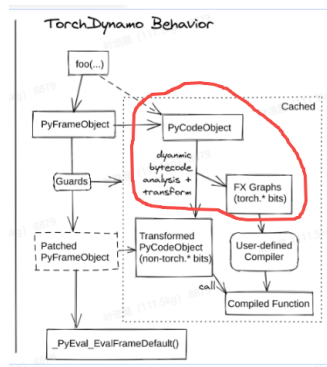
浅入深地好好聊一聊，PyTorch 2.0 中的 Dynamo，是如何完成 Graph trace 的。>>加入极市 CV技术交流群，走在计算机视觉的最前沿

随着 PyTorch 2.0 的正式发布，相信很多小伙伴已经使用过 PyTorch 2.0 的 compile 功能，也尝试写过自己的编译后端，对模型做一些定制化的优化。得益于 Dynamo 强大的字节码解析能力，我们能够在不关心代码解析过程的情况下，随心所欲地写编译优化后端。然而，由于字节码解析部分实现的复杂性，目前并没有比较完整的资料介绍其工作原理。今天我们就来由浅入深地好好聊一聊，**PyTorch 2.0 中的 Dynamo，是如何完成 Graph trace 的。**

上一篇文章我们提到，Dynamo 是如何通过 PEP 523 改变 Python 默认的函数（帧评估）执行流程，将它从下图的 Default Python Behavior 转变为 TorchDynamo Behavior：



在了解 Dynamo 设计的基石后，我们就可以一步一步地理解上图右侧栏各个流程框图的含义：



1. 在第一次执行被 `torch.compile` 编译的函数时，会走上图右侧的分支，从 `PythonFrameObject`（帧的定义可以见上篇文章）中解析出 `PyCodeObject`
2. 基于 `PyCodeObject` 中的字节码解析出 `fx graph`，同时生成守卫（Guard），并在解析过程中使用指定后端对代码进行编译
3. 将编译后的代码替换原有的代码，获得 `Transformed PyCodeObject`，函数实际运行时会调用编译后的代码
4. 第二次执行时，守卫会判断是否需要重新编译，如果不需要则会从缓存中直接读取上次编译的代码，否则会触发重新编译

好好的，一口气抛出这么多概念，相信不少小伙伴会有一种说了等于没说的感觉。没关系，今天我们由浅入深，详细介绍每一个步骤的内容。

## 第一章：Dynamo 的帧执行流程

上篇文章我们提到，Dynamo 基于 PEP 523，设计了一个自定义的帧执行函数，而今天我们就来看看，这个函数具体做了哪些事（只保留了代码的主体逻辑，且不考虑 `subgraph` 等更复杂的情况）：

1. 调用 `torch.compile` 编译函数时，编译返回的函数实际为 `_TorchDynamoContext` 里定义的 `_fn` 函数（[https://github.com/pytorch/pytorch/blob/38da54e9c9471565812d2be123ee4e9fd6bfbdbc0/torch/\\_dynamo/eval\\_frame.py#L215](https://github.com/pytorch/pytorch/blob/38da54e9c9471565812d2be123ee4e9fd6bfbdbc0/torch/_dynamo/eval_frame.py#L215)）
2. `_fn` 会把 Python 默认的帧执行函数替换为 Dynamo 自定义的帧执行函数 `_custom_eval_frame`（[https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/csrc/dynamo/eval\\_frame.c#L636](https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/csrc/dynamo/eval_frame.c#L636)）
3. 执行目标函数时，会进入 `_custom_eval_frame`，并调用 `callback` 函数（关于 `callback` 函数的功能可以见上篇文章）对帧进行解析，并返回编译结果 **result**

- `callback`：用于解析字节码，进行 Graph trace，最后返回编译结果
- `result`：即 `GuardedCode` 实例（[https://github.com/pytorch/pytorch/blob/a9b9fd90a273e3430b2c58632b890ba095db5369/torch/\\_dynamo/convert\\_frame.py#L391](https://github.com/pytorch/pytorch/blob/a9b9fd90a273e3430b2c58632b890ba095db5369/torch/_dynamo/convert_frame.py#L391)），其中 `code` 属性为编译优化后的代码，`check_fn` 为检查代码，用于检查当前是否需要重新编译函数。

调用 `callback` 函数时还需要传入 `cache_size` 参数，表示当前是第几次编译该函数，第一次调用时其值为 0。当 `cache_size` 大于阈值时，不再编译该函数，按照原有逻辑执行。

4. 将 `result` 缓存到 `extra`，其中 `extra` 是一个链表，每执行一次编译链表都会新增一个元素。往后每次执行函数时都会根据当前帧的状态和 `extra` 中的往期编译结果来判断是否需要重新

新编译

5. 执行编译后的代码，返回结果
6. 第 2 次执行时，加载上次生成的 extra，进行查表操作（lookup）。遍历 extra 中的每个元素，执行 **GuardedCode.check\_fn**
- 如果 extra 中某个元素的 check\_fn 返回 True，则将该元素放到链表的最前端，方便下一次检查时优先遍历。同时终止遍历，运行之前编译好的代码。

• 如果所有的 check\_fn 均返回 False，则重复执行 2~4 步骤。需要注意的是，每执行一轮 2-4 步骤。

如果你觉得上述流程说得通，继续按照文章顺序阅读即可，如果你觉得上述流程存在逻辑缺陷，可以直接移步编译子图一节。> 如果你对 C 代码不是很熟，也可以跳过这部分的理解，只需要记住：字节码解析最终会返回 GuardedCode 实例，该实例含有两属性，其中 check\_fn 用来判断代码是否需要重新编译的，code 部分则存放编译好的代码。

## 第二章：字节码解析与图生成

第一章提到的编译好的代码（**GuardedCode.code**）其实已经是 Dynamo 编译器**前端解析+后端编译**的最终产物了，而现在我们要介绍的字节码解析，正是**前端解析**的具体流程。本章我们会深入 callback 函数，理解如何从帧中解析字节码，获取模型图结构，最终生成 GuardedCode。

在 CPython 中，Python 代码是在 CPython 的虚拟机中执行的，而执行的过程，正是上篇文章我们提到的 \_PyEval\_EvalFrameDefault 函数，它会将帧中函数的代码，解析成一系列的字节码，并在一大串的 switch-case 中逐条执行字节码，CPython 支持的所有字节码见 opcode.h（<https://github.com/python/cpython/blob/e6b0bd59481b9bc4570736c1f5ef291dbbe06b8e/Include/opcode.h>）。

我们可以在 Python 代码中，通过使用 dis.dis 函数，来查看任意一个函数在 CPython 虚拟机中执行时的字节码：

```
import dis

def add(x, y):
    res = x + y
    return res

# 以表格的形式输出字节码信息
dis.dis(add)
# 逐条输出字节码详细信息
for inst in dis.Bytecode(add):
    print(inst)
```

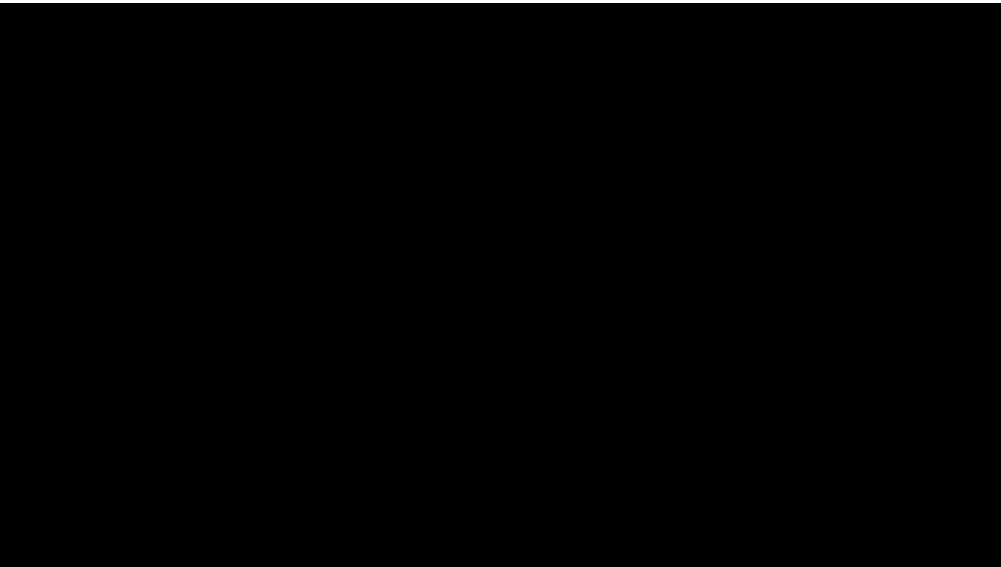
输出：

```
# 代码行数      # 代码对应的字节码      # 变量名
#    5          0 LOAD_FAST          0 (x)
#              2 LOAD_FAST          1 (y)
#              4 BINARY_ADD
```

```
#          6 STORE_FAST          2 (z)
#
#  6          8 LOAD_FAST          2 (z)
#          10 RETURN_VALUE

# Instruction(opname='LOAD_FAST', opcode=124, arg=0, argval='x', argrepr='x', offset=0, st
# Instruction(opname='LOAD_FAST', opcode=124, arg=1, argval='y', argrepr='y', offset=2, st
# Instruction(opname='BINARY_ADD', opcode=23, arg=None, argval=None, argrepr='', offset=4,
# Instruction(opname='STORE_FAST', opcode=125, arg=2, argval='res', argrepr='res', offset=
# Instruction(opname='LOAD_FAST', opcode=124, arg=2, argval='res', argrepr='res', offset=
# Instruction(opname='RETURN_VALUE', opcode=83, arg=None, argval=None, argrepr='', offset=
```

这些字节码到底做了什么事呢，CPython 用非常复杂的 C 代码来解析每个字节码，而 Dynamo 则在 Python 层面对字节码进行解析，并 trace 模型的图结构的。



如上图所示，字节码的解析可以大体分成以下 7 个步骤：

- 1. 解析输入参数 x, y，将其存储到局部变量 (local\_var)
- 2. LOAD\_FAST x: 将变量 x push 入栈
- 3. LOAD\_FAST y: 将变量 y push入栈
- 4. BINARY\_ADD: 从 stack 中 pop 出 x, y，计算出结果后将其 push 入栈
- 5. STORE\_FAST res: 从 stack 中 pop 出栈顶元素（即上一步的结果），并将其存储到局部变量 res
- 6. LOAD\_FAST res: 将变量 res push入栈
- 7. RETURN\_VALUE: pop 出栈中的 res 并返回

Dynamo 实现了 InstructionTranslator ([https://github.com/pytorch/pytorch/blob/bda9d7ba73fa6f8f1aa01189cc2b03d81601cb9e/torch/\\_dynamo/symbolic\\_convert.py#L424](https://github.com/pytorch/pytorch/blob/bda9d7ba73fa6f8f1aa01189cc2b03d81601cb9e/torch/_dynamo/symbolic_convert.py#L424) ) 来解析字节码，为了方便理解其核心内容，这边实现了简易版的 SimpleInstructionTranslator:

在看样例代码之前，我们先介绍几个概念：

1. `torch.fx.Graph` 是 Dynamo 解析字节码后生成的 intermediate representation (IR)，换句话说 `torch.fx.Graph` 是解析字节码后，`trace` 得到的图结构。在示例代码中把它理解成简单的图数据结构就可以。
2. `Graph.create_node` 可以为图生成一系列的节点，传入的 `op` 为节点类型，其中输入节点的类型为 "placeholder"，输出节点的类型为 "output"，函数调用的节点类型为 "call\_function"。
3. `Graph.python_code` 可以从 `trace` 得到的图生成代码，这边我们用它来校验图的正确性。

```
import dis
import operator
from dis import Instruction

from torch.fx import Graph

class SimpleInstructionTranslator:
    def __init__(self, instructions, inputs) -> None:
        self.graph = Graph()
        self.instructions = instructions
        self.stack = []
        self.locals = {}
        for input in inputs:
            node = self.graph.create_node(op='placeholder', target=input, args=(), kwargs=
            self.locals[input] = node

    def run(self):
        for inst in self.instructions:
            print(f'parse the bytecode op {inst.opname}')
            getattr(self, inst.opname)(inst)
        return self.graph

    def LOAD_FAST(self, inst: Instruction):
        argval = inst.argval
        self.push(self.locals[argval])
        return

    def STORE_FAST(self, inst: Instruction):
        argval = inst.argval
        self.locals[argval] = self.pop()
        return

    def BINARY_ADD(self, inst: Instruction):
        add = operator.add
        node = self.graph.create_node(op='call_function', target=add, args=(self.pop(), se
        self.push(node)
        return

    def push(self, val):
        self.stack.append(val)
        return

    def pop(self):
        return self.stack.pop()

    def RETURN_VALUE(self, inst):
        output = self.pop()
        return self.graph.create_node(op='output', target='output', args=(output, ), name=
```

```
def add(x, y):
    res = x + y
    return res

if __name__ == '__main__':
    instructions = dis.Bytecode(add)
    translator = SimpleInstructionTranslator(instructions, ('x', 'y'))
    translator.run()
    translator.graph.print_tabular()
    print(translator.graph.python_code('root').src)

# parse the bytecode op LOAD_FAST
# parse the bytecode op LOAD_FAST
# parse the bytecode op BINARY_ADD
# parse the bytecode op STORE_FAST
# parse the bytecode op LOAD_FAST
# parse the bytecode op RETURN_VALUE

# opcode      name      target      args      kwargs
# -----
# placeholder  x        x           ()        {}
# placeholder  y        y           ()        {}
# call_function add      <built-in function add> (y, x)    {}
# output       output   output      (add,)    {}

# Generated code:
# def forward(self, x, y):
#     add = y + x; y = x = None
#     return add
```

SimpleInstructionTranslator 会在 run 函数中，对每条字节码进行解析。每个同名的字节码函数，都在模拟相应字节码解析的流程。SimpleInstructionTranslator 只实现了LOAD\_FAST LOAD\_FAST BINARY\_ADD RETURN\_VALUE 三个字节码解析函数，因此它只能解析简单加法操作的函数，这里再给一个稍微复杂一点点的例子：

```
def add_three(x, y, z):
    res1 = x + y
    res2 = res1 + z
    return res2

if __name__ == '__main__':
    instructions = dis.Bytecode(add_three)
    translator = SimpleInstructionTranslator(instructions, ('x', 'y', 'z'))
    translator.run()
    translator.graph.print_tabular()
    print(translator.graph.python_code('root').src)

# parse the bytecode op LOAD_FAST
# parse the bytecode op LOAD_FAST
# parse the bytecode op BINARY_ADD
# parse the bytecode op STORE_FAST
# parse the bytecode op LOAD_FAST
# parse the bytecode op LOAD_FAST
# parse the bytecode op BINARY_ADD
# parse the bytecode op STORE_FAST
# parse the bytecode op LOAD_FAST
```

```
# parse the bytecode op RETURN_VALUE

# opcode      name      target      args      kwargs
# -----
# placeholder  x        x           ()        {}
# placeholder  y        y           ()        {}
# placeholder  z        z           ()        {}
# call_function add      <built-in function add> (y, x)    {}
# call_function add_1    <built-in function add> (z, add)  {}
# output       output   output      (add_1,)  {}

# def forward(self, x, y, z):
#     add = y + x; y = x = None
#     add_1 = z + add; z = add = None
#     return add_1 world!";
```

显然，SimpleInstructionTranslator 依旧很好地完成了 add\_three 字节码解析和图 trace 的工作。

由于实际解析的代码会更加的复杂，官方的 InstructionTranslator 实现了更多的字节码解析函数，处理各种各样的 corner case。

事实上，PyTorch 并没有往 stack 里 push GraphNode 而选择往里面 push 一个新的抽象 VariableTracker，并在此基础上引入了 Guard 的概念。后续我们将会从原理和源码层面分析，为什么需要 VariableTracker 和 Guard，以及它们又是如何实现的。

为什么需要 VariableTracker

字节码信息的不完整性

字节码不会包含程序的运行时信息，如果光从字节码去 trace 模型，那和从抽象语法树（AST）去 trace 模型没有太大区别，trace 得到的图也不具备动态特性（没有输入自然没有办法根据输入做动态判断）。

再举个最简单的例子，对于这样一行代码：self.layer1(x)，字节码解析的过程中会触发：

- LOAD\_FAST：加载 self，push 入栈
- LOAD\_ATTR：pop 出 self，加载 layer1，将 layer1 push 入栈
- LOAD\_FAST：将 x push 入栈
- CALL\_FUNCTION pop 出 layer1 和 x，并执行 layer1(x)

然而问题在于此时程序没有运行，没法获取到 self.layer1 这个函数，自然也没法进一步解析这个函数的字节码了。Dynamo 的顶层设计决定了 trace 的过程会从 frame evaluation ([https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/csrc/dynamo/eval\\_frame.c#L636](https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/csrc/dynamo/eval_frame.c#L636)) 入手，在运行阶段完成图的追踪。程序运行时我们可以获得输入信息，因此我们需要一个数据结构去承载字节码以外的信息，那就是 VariableTracker。

Graph 的动态特性

Dynamo trace 出来图的动态特性，是由守卫（guard）所赋予的，而守卫的载体就是 VariableTracker，这部分我们后续会进行详细介绍。

字节码信息对于模型图结构是“冗余”的

Dynamo 基于字节码的 graph trace，其目的不是 trace 出一个完整 Python 的图表示，否则这和基于字节码重构抽象语法树也没有太大区别。这里给出一个简单的例子：

```
import torch
import torch.nn as nn
from torch._dynamo.bytecode_transformation import cleaned_instructions

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(1, 1)
        self.linear2 = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear1(x) + self.linear2(x)

def custom_backend(gm, example_inputs):
    gm.graph.print_tabular()
    return gm.forward

if __name__ == '__main__':
    model = Model()
    instructions = cleaned_instructions(model.forward.__code__)
    for i in instructions:
        print(i)

    compiled_model = torch.compile(model, backend=custom_backend)
    compiled_model(torch.rand(1, 1))

# Instruction(opcode=124, opname='LOAD_FAST', arg=0, argval='self', offset=0, starts_line=
# Instruction(opcode=106, opname='LOAD_ATTR', arg=0, argval='linear1', offset=2, starts_li
# Instruction(opcode=124, opname='LOAD_FAST', arg=1, argval='x', offset=4, starts_line=Nor
# Instruction(opcode=131, opname='CALL_FUNCTION', arg=1, argval=1, offset=6, starts_line=
# Instruction(opcode=124, opname='LOAD_FAST', arg=0, argval='self', offset=8, starts_line=
# Instruction(opcode=106, opname='LOAD_ATTR', arg=1, argval='linear2', offset=10, starts_l
# Instruction(opcode=124, opname='LOAD_FAST', arg=1, argval='x', offset=12, starts_line=Nc
# Instruction(opcode=131, opname='CALL_FUNCTION', arg=1, argval=1, offset=14, starts_line=
# Instruction(opcode=23, opname='BINARY_ADD', arg=None, argval=None, offset=16, starts_lir
# Instruction(opcode=83, opname='RETURN_VALUE', arg=None, argval=None, offset=18, starts_l

# opcode      name      target      args      kwar
# -----
# placeholder  x        x          ()        {}
# call_module  self_linear1  self_linear1  (x,)      {}
# call_module  self_linear2  self_linear2  (x,)      {}
# call_function  add        <built-in function add>  (self_linear1, self_linear2)  {}
# output      output      output      ((add,),)  {}
```

可以看到，torch.compile 解析出来的图结构，只包含了部分字节码信息，LOAD\_ATTR 并没有体现在 trace 出来的 graph 上。这事实上也归功于 stack push pop 的不是 Node 实例，而是 VariableTracker 实例。因此在执行 LOAD\_ATTR 的字节码解析函数时，不需要往 graph 中新增一个 node，再将其 push 入栈，为 Graph 更新一个 get\_attr 节点了。

VariableTracker



既然 Node 不适合直接作为字节码解析过程中，push pop 操作的载体，Dynamo 就设计了一个新的数据类，VariableTracker。其功能顾名思义，就是用来追踪字节码解析过程中产生的变量。VariableTracker 能够接受函数运行时的信息，并控制 Graph 的生成。

设想一下，如果我们把样例代码中的所有 Node，都替换成 VariableTracker，直接面临的问题就有两个：

1. Node 是有 op type 的，不同类型 op type 的 Node 相互组合才可以生成 PythonCode 的 Graph，那么 VariableTracker 应该如何体现节点类型的不同呢？
2. VariableTracker 又应该如何和 Node 关联，以生成最终的 Graph 呢？

### 不同类型的 VariableTrackers

正如问题里提到的，解析不同类型的字节码需要生成不同类型的 VariableTracker，例如在执行 CALL\_FUNCTION 之前，我们需往先 stack 里 push 一个 UserFunctionVariable，再往 stack 里 push 一个 TensorVariable（假设函数的输入是 Tensor 类型）。最后在 CALL\_FUNCTION 里将二者 pop 出来，调用 UserFunctionVariable 的方法模拟函数执行。

Dynamo 在 variables 文件夹 ([https://github.com/pytorch/pytorch/tree/main/torch/\\_dynamo/variables](https://github.com/pytorch/pytorch/tree/main/torch/_dynamo/variables)) 中定义了所有的 VariableTracker 类型，感兴趣的话可以看看每个 VariableTracker 的功能。

### 基于 VariableTracker 生成 Graph

上一节我们提到，LOAD\_ATTR 之类的字节码是不会生成相应的 Node 的（绝大部分情况），因此 LOAD\_ATTR 就不应该调用 Graph.create\_node 来生成相应的节点。而像 CALL\_FUNCTION 之类的字节码，是否生成新的节点，会视 VariableTracker 的具体值而定，例如：

```
def foo(x, y):  
    return x + y  
  
def foo1(x, y):  
    return
```

foo 中的 BINARY\_ADD 字节码作为内置函数（BuiltinVariable），解析时会生成新的节点，而 foo1 作为是一个空函数，则不会生成新的节点。因此是否生成新的节点，是和 VariableTracker 实例本身相关，而如果要在 InstructionTranslator 这一层处理这些逻辑，这部分代码的可读性将是一个灾难。

因此 Dynamo 新增了一层抽象 VariableBuilder ([https://github.com/pytorch/pytorch/blob/38b687ed4de5d74423ef0d0a60a4aa007d0c4ec9/torch/\\_dynamo/variables/builder.py#L149](https://github.com/pytorch/pytorch/blob/38b687ed4de5d74423ef0d0a60a4aa007d0c4ec9/torch/_dynamo/variables/builder.py#L149)) 来负责 VariableTracker 的构建，并控制过程中是否生成新的 Node 等操作（包括生成 Guard，下一节会介绍）。

这边贴一段 VariableBuilder 生成 TensorVariable 代码片段，大家自行感受一下（冰山一角）：

```
def wrap_tensor(self, value: torch.Tensor):  
    if self.get_source().guard_source().is_nn_module():  
        return self.tx.output.register_attr_or_module(  
            value,  
            self.name,  
            source=self.get_source(),
```

```

# Guards are done inside register_attr_or_module
# guards=self.make_guards(GuardBuilder.TENSOR_MATCH),
)

if is_constant_source(self.get_source()):
    return self.tx.output.register_attr_or_module(
        value,
        re.sub(r"^[^a-zA-Z0-9]+", "_", self.name),
        source=self.get_source(),
        # Guards are added inside register_attr_or_module
    )
...

```

这边给大家简单翻译一下（可以简单把 source 理解成数据源，用于帮助 Guard 生成检查代码）：

- 如果这个 Tensor 是来自于一个 nn.Module 的（类似 register\_buffer），那么他就会往 graph 里注册一个节点，并返回一个 TensorVariable
- 如果这个 Tensor 的数据来源是一个常量（torch.Tensor(1)），操作同上，只不过名字会有所不同
- ...

不要急，走进 register\_attr\_or\_module 这个函数，你会看到更多的 if-else。不得不说，Dynamo 为了处理代码的各种情况，可以说全是 hardcode，让人看了痛苦不堪。

回到这个问题本身，如果 VariableBuilder 全权负责全部 VariableTracker 的构建和 Graph 节点的更新，那么从层次上来讲好像也还算清晰，InstructionTranslator 也可以免于判断什么时候需要新增节点。例如对于输入参数，InstructionTranslator 直接使用 VariableBuilder 构建一系列的 VariableTracker，完全不需要关心 Graph 相关的逻辑，做到模块之间的功能解耦。

```

self.symbolic_locals = collections.OrderedDict(
    (
        k,
        VariableBuilder(
            self,
            LocalInputSource(k, code_options["co_varnames"].index(k))
            if k in code_options["co_varnames"]
            else LocalSource((k)),
        )(f_locals[k]),
    )
    for k in vars
    if k in f_locals
)

```

## 守卫 (Guard)

前面介绍的种种只是在描述 Dynamo 是如何通过字节码生 trace graph，而为了让 trace 出来的 graph 保持动态特性，就离不开**核心组件：Guard**。在构建 VariableTracker 时，可能会绑定一个或多个 guard，**用于生成监视变量的检查代码**，也就是我们最初提到的 check\_fn。需要注意的是，Graph trace 阶段可能会生成非常多的 guard，但是最后只有部分 guard 会被用于生成 check\_fn，这其实也很好理解，因为只有部分变量都会造成模型的动态结构。

Guard 功能的实现主要依赖两个模块：Guard 和 GuardBuilder。

Guard： [https://github.com/pytorch/pytorch/blob/542fb0b1fad6bf61929df16e2133e9a296820f08/torch/\\_guards.py#L82](https://github.com/pytorch/pytorch/blob/542fb0b1fad6bf61929df16e2133e9a296820f08/torch/_guards.py#L82)

GuardBuilder： [https://github.com/pytorch/pytorch/blob/542fb0b1fad6bf61929df16e2133e9a296820f08/torch/\\_dynamo/guards.py#L85](https://github.com/pytorch/pytorch/blob/542fb0b1fad6bf61929df16e2133e9a296820f08/torch/_dynamo/guards.py#L85)

**Guard**：Graph trace 过程中生成，记录最后生成检查代码阶段所需的额外信息，并最后存储生成后的代码。这边最主要介绍初始化阶段的两个核心参数：

- source：记录守护的变量名 name，例如 "self.layer1.state"，变量名用于生成检查代码
- create\_fn：用于生成检查代码的函数，其值通常为 GuardBuilder 的 method，在 Guard Builder 部分展开介绍

**GuardBuilder**：Graph trace 完成后，基于 trace 过程中生成的 Guards，生成最终的检查代码。

我们通过一些代码示例来理解 Guard 和 GuardBuilder 是如何起作用的。首先修改 Dynamo 的配置，以输出 Guard 相关的日志：

```
import logging

import torch
import torch._dynamo.config
import torch.nn as nn

torch._dynamo.config.log_level = logging.INFO
torch._dynamo.config.output_code = True

class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(1, 1)
        self.linear2 = nn.Linear(1, 1)
        self.x = 1

    def forward(self, x):
        return self.linear1(x) + self.linear2(x)

if __name__ == '__main__':
    model = Model()
    compiled_model = torch.compile(model)
    compiled_model(torch.rand(1, 1)).ardded_class': None
}
```

Guard 相关的输出日志：

```
local 'x' TENSOR_MATCH
{
  'guard_types': ['TENSOR_MATCH'],
  'code': None,
  'obj_weakref': <weakref at 0x7f9035c8bce0; to 'Tensor' at 0x7f8f95bddd00>
  'guarded_class': <weakref at 0x7f8f98cf9440; to 'torch._C._TensorMeta' at 0x57f3e10 (1
}
```

```

-
local 'self' NN_MODULE
{
    'guard_types': ['ID_MATCH'],
    'code': ['__check_obj_id(self, 140260021010384)'],
    'obj_weakref': <weakref at 0x7f8f9864a110; to 'Model' at 0x7f90d4ba7fd0>
    'guarded_class': <weakref at 0x7f90d4bc71a0; to 'type' at 0x705e0f0 (Model)>
}

-
local_nn_module 'self.linear1' NN_MODULE
{
    'guard_types': None,
    'code': None,
    'obj_weakref': None
    'guarded_class': None
}

-
local_nn_module 'self.linear2' NN_MODULE
{
    'guard_types': None,
    'code': None,
    'obj_weakref': None
    'guarded_class': None
}

```

默认配置下，Dynamo 不会对 `nn.Module` 进行检查，即假设训练过程中，`nn.Module` 不会发生 inplace 的替换，因此此处 `self.linear1` 和 `self.linear2` 的 `guard` 均为 `None`，代码执行时不会对其进行检查。上述代码实际起作用的 `Guard` 只有输入 `self` 和 `x`，这边重点介绍 **guard\_types** 和 **code** 属性。

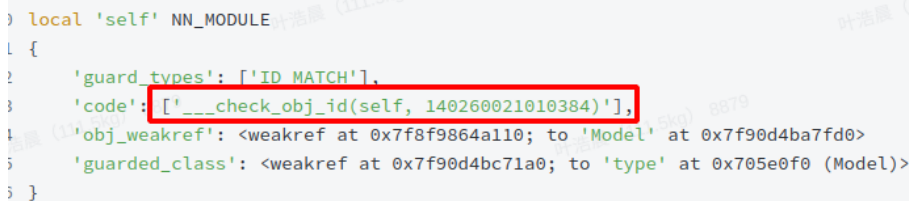
对于不同类型的变量，守卫生成代码的方式也会有所不同。`Guard` 借助 `GuardBuilder` ([https://github.com/pytorch/pytorch/blob/573b2deb4b9a056d25c4e969bdc1e0230c508650/torch/\\_dynamo/guards.py#L85](https://github.com/pytorch/pytorch/blob/573b2deb4b9a056d25c4e969bdc1e0230c508650/torch/_dynamo/guards.py#L85)) 定义了一系列不同类型变量的守护方式（感觉这个类名容易让人产生误解，认为 `Guard` 是通过 `GuardBuider` 构建而成的。然而事实上，`GuardBuilder` 是用于构建 **check\_fn**，即检查代码的）。例如 `CONSTANT_MATCH`，`TENSOR_MATCH` 等等。Graph trace 完成之后，生成的 `Guards` 会调用这些方法以生成 **check\_fn**。

出于性能方面的考虑，**check\_fn** 会调用 `guards.cpp` 里实现 C 函数以实现状态检查（尤其是 `Tensor` 类型，在 Python 里做检查性能损耗严重），这边给出两个例子。

1. 检查变量 `id` 是否相等 (`ID_MATCH`)，`check_fn` 会调用以下函数

```
static PyObject* check_obj_id(PyObject* dummy, PyObject* args) {
    // faster `lambda obj, expected: id(obj) == expected`
    PyObject* obj;
    unsigned long long expected;
    if (!PyArg_ParseTuple(args, "OK", &obj, &expected)) {
        return NULL;
    }
    if (obj == (void*)expected) {
        Py_RETURN_TRUE;
    } else {
        Py_RETURN_FALSE;
    }
}
```

此处 C++ 层面实现的 check\_obj\_id 对应 Guard 信息中的



```
{
  'guard_types': ['ID_MATCH'],
  'code': ['__check_obj_id(self, 140260021010384)'],
  'obj_weakref': <weakref at 0x7f8f9864a110; to 'Model' at 0x7f90d4ba7fd0>
  'guarded_class': <weakref at 0x7f90d4bc71a0; to 'type' at 0x705e0f0 (Model)>
}
```

检查 self 参数时，check\_obj\_id 会根据其 id 是否匹配，来决定是否需要重复编译

## 2. 检查 Tensor 是否匹配 (TENSOR\_MATCH)，check\_fn 会调用以下函数

对于 Tensor 类型数据的检查，出于效率方面的考虑，检查代码同样在 C++ 代码里实现：

```
bool check(const LocalState& state, const at::Tensor& v) {
    if (dispatch_key_ != state.apply(v.key_set()).raw_repr() ||
        dtype_ != v.dtype().toScalarType() ||
        device_index_ != v.device().index() ||
        requires_grad_ != (state.grad_mode_enabled && v.requires_grad())) {
        return false;
    }
    auto ndim = static_cast<size_t>(v.ndimension());
    if (ndim != sizes_.size()) {
        return false;
    }
    if (!dynamic_shapes_) {
        const auto& sizes = v.sizes();
        const auto& strides = v.strides();
        for (auto i : c10::irange(ndim)) {
            if (sizes_[i] != sizes[i] || strides_[i] != strides[i]) {
                return false;
            }
        }
    }
    return true;
}
```

简单来说会检查以下几个内容：

1. 数据类型是否发生变化，例如原来数据类型为 float32，第二次输入时类型变成 float16，返回 False

2. 数据所在设备是否发生变化，例如原来是在 GPU 0 上的，第二次输入变成在 GPU 1 上了，返回 False
3. 数据的梯度属性是否发生变化，例如原来是需要计算梯度的，第二次却不再要求计算梯度，返回 False
4. (Dynamic shape=False 时) 数据的形状以及内存排布是否发生变化

此外，Tensor 以外的变量通常采取一个变量，一个 Guard 的检查策略，而 Tensor 类型的数  
据则会进行集中检查，即所有 Tensor 变量只会生成一个检查函数：\_\_check\_tensors，该函  
数会遍历并检查所有 Tensor。

对于上例来说，其最终生成的检查代码 check\_fn 的过程等价于：

```
import torch

TensorGuards = torch._C._dynamo.guards.TensorGuards
check_obj_id = torch._C._dynamo.guards.check_obj_id

def gen_check_fn(self, x):
    tensor_guards = TensorGuards(x, dynamic_shapes=False)
    id_self = id(self)

    def check_fn(self, x):
        # 返回 True 表示不需要重新编译，反之则需要重新编译
        # tensor_guards.check 允许同时检查多个 tensor
        return (check_obj_id(self, id_self) and
                tensor_guards.check(*(x, )))
    return check_fn

if __name__ == '__main__':
    self = nn.Linear(1, 1)
    x = torch.rand(1, 1)
    func = gen_check_fn(self, x)
    print(f'Should recompiled: {not func(self, x)}')
    print(f'Should recompiled: {not func(nn.Linear(1, 1), x)}')
```

回到第一节编译与执行流程的第四步，其中提到的 check\_fn 等价于上例中返回的 check\_fn，  
如果 self 的 id 发生变化，亦或是 x 无法通过 TensorGuards.check，均会触发重新编译。

## 编译子图

Guard 一节提到，check\_fn 只会检查模型的输入，而不是实际运行一遍代码后，再判断是否  
应该重新编译一遍函数。这也是合情合理的，因为执行一遍代码才能完成代码检查，这样的开  
销是不可接受的。然而这样也会引入其他问题，真的能够仅仅根据输入去判断是否需要重新编  
译模型么？

对于比较简单的函数：

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(1, 1)
        self.linear2 = nn.Linear(1, 1)
        self.x = 1
```

```
def forward(self, x):
    a = self.linear1(x)
    b = self.linear2(x)
    if len(a.shape) == 2:
        return a + b
    else:
        return a - b
```

Dynamo 并不需要为 `a` 和 `b` 生成 `Guard` 和 `check_fn`，因为只要 `x` 的形状不变，`a.shape` 就不会发生变化（假设 `len` 是 builtin func，且 `linear1` 保持不变），因此只需要对 `x` 构建 `guard` 并生成 `check_fn` 就足够了。

那如果换一种写法：

```
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(1, 1)
        self.linear2 = nn.Linear(1, 1)
        self.x = 1

    def forward(self, x):
        a = self.linear1(x)
        b = self.linear2(x)
        if x.sum() >= 1:
            return a + b
        else:
            return a - b
```

这里的 `x.sum()` 会返回一个 `Tensor`，此时无论如何都没有办法仅凭输入去判断会走哪个分支。对于这种情况，Dynamo 的做法是：**编译子图**。

细心的同学可能会发现，编译与执行流程一节提到的执行顺序，是有**漏洞**的。因为在执行完第一步，将默认的执行函数替换成 `_custom_eval_frame` 后，这意味着 `callback` 执行过程中产生的函数栈，也会触发 `_custom_eval_frame`，这是不符合期望的。我们只希望执行被编译的函数时，能够触发 `_custom_eval_frame`，因此完整的执行流程如下：

1. 在 `eval_frame.py` 中，将帧执行函数替换 Dynamo 自定义的执行函数 `_custom_eval_frame`
2. 进入 `_custom_eval_frame` 后，将帧执行函数替换回默认的执行函数
3. 第一次执行待编译的函数时 调用 `callback` 函数，对帧进行解析
4. 将 `result` 缓存到 `extra`，其中 `extra` 是一个链表，每执行一次编译链表都会新增一个元素。往后每次执行函数时都会根据当前帧的状态和 `extra` 中的往期编译结果来判断是否需要重新编译
5. 将默认的帧执行函数重新替换成 `_custom_eval_frame`
6. 用默认的帧执行函数执行编译后的字节码，并返回结果
7. 第 2 次执行时，加载上次生成的 `extra`，进行查表操作（lookup）。遍历 `extra` 中的每个元素，执行 `GuardedCode.check_fn`

- 如果某个元素的 `check_fn` 返回 `True`，则把该元素放到链表的最前端，执行该元素之前编译好的代码。
- 如果所有的 `check_fn` 均返回 `False`，则重复执行 1~3 步骤。需要注意的是，每执行一轮 1~3 步骤，`callback` 传入的 `cache_size` 参数就会递增，当其值大于 `torch._dynamo.config.cache_size_limit` 时，就会认为该函数过于动态，不再对其进行编译，而以函数原有的逻辑去执行代码。

8. 编译完整个函数后，在 `eval_frame.py` 中，将帧执行函数替换回默认的执行函数

第六步，划重点！在第五步我们将帧执行函数替换成 `_custom_eval_frame` 后，如果我们直接执行编译后的字节码，这就意味着会触发无限递归，因此需要调用默认的帧执行函数执行字节码。那既然如此，为什么还需要在第五步把帧执行函数替换成 `_custom_eval_rame` 呢？答案是，**编译子图**。

编译后的字节码中还会存在 `CALL_FUNCTION` 字节码，在执行时会进入 `_custom_eval_frame`，进而触发对子图的编译。

回到上面的例子，函数在第一次编译 `Model.forward` 时，会生成这样的字节码（生成过程详见 `generic_jump`（[https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/\\_dynamo/symbolic\\_convert.py#L234](https://github.com/pytorch/pytorch/blob/141a2ebcf199c3f20b08e090b7e2a0527c5d9da5/torch/_dynamo/symbolic_convert.py#L234)）：

```

22      0 LOAD_GLOBAL          3 (__compiled_fn_0)
      2 LOAD_FAST              1 (x)
      4 CALL_FUNCTION          1
      6 UNPACK_SEQUENCE        3
      8 STORE_FAST             3 (b)
     10 STORE_FAST             2 (a)
     12 POP_JUMP_IF_FALSE     12 (to 24)
     14 LOAD_GLOBAL            4 (__resume_at_32_1)
     16 LOAD_FAST              2 (a)
     18 LOAD_FAST              3 (b)
     20 CALL_FUNCTION          2
     22 RETURN_VALUE

>> 24 LOAD_GLOBAL            5 (__resume_at_40_2)
     26 LOAD_FAST              2 (a)
     28 LOAD_FAST              3 (b)
     30 CALL_FUNCTION          2
     32 RETURN_VALUE

```

等价 Python 代码如下：

```

def compiled_fn(x):
    a, b, is_true = __compiled_fn_0(x)
    if is is_true:
        return __resume_at_32_1(a, b)
    else:
        return __resume_at_40_2(a, b)

```

Dynamo 在解析到 `x.sum() >= 1` 时发现，该函数无法通过 `Guard` 来判断是否需要重新编译，于是就退而求次的把一个函数编译成三个子图，编译的结果如等价 Python 代码所示，相信大



家一看就懂。

这里提到的字节码也正是第一次编译走到第六步时，其执行的字节码。字节码中三次 `CALL_FUNCTION`，对应示例 Python 代码中的 `__compiled_fn_0`，`__resume_at_32_1` 和 `__resume_at_40_2`。其中在执行 `__resume_at_32_1` 和 `__resume_at_40_2` 时，会再次触发 `_custom_eval_frame`，对二者进行编译。因此，在执行上述代码时会显示生成了两次 Guard，第一次发生在编译原始函数，生成 `compiled_fn`，第二次发生在编译 `compiled_fn`，分别对 `__resume_at_32_1` 和 `__resume_at_40_2` 进行编译。

细心的你可能会发现，这样 `__compiled_fn_0` 不是也会触发二次编译么。Dynamo 自然也考虑到了这一点，编译后的函数会经过 `disable` ([https://github.com/pytorch/pytorch/blob/e9050ef74e9facb4a5464756a7b6b187dedab89d/torch/\\_dynamo/output\\_graph.py#L652](https://github.com/pytorch/pytorch/blob/e9050ef74e9facb4a5464756a7b6b187dedab89d/torch/_dynamo/output_graph.py#L652)) 处理，保证后续的调用不会再去 `_custom_eval_frame` 的逻辑。

第一次，解析 forward 时生成的 guard：

```

local 'x' TENSOR_MATCH
{
    'guard_types': ['TENSOR_MATCH'],
    'code': None,
    'obj_weakref': <weakref at 0x7f24b16ad6c0; to 'Tensor' at 0x7f2412371670>
    'guarded_class': <weakref at 0x7f24147b41d0; to 'torch._C._TensorMeta' at
}

-

local 'self' NN_MODULE
{
    'guard_types': ['ID_MATCH'],
    'code': ['__check_obj_id(self, 139798240864976)'],
    'obj_weakref': <weakref at 0x7f24b16afc40; to 'Model' at 0x7f25507caad0>
    'guarded_class': <weakref at 0x7f241223e3e0; to 'type' at 0x6a2d470 (Model)
}

-

local_nn_module 'self.linear1' NN_MODULE
{
    'guard_types': None,
    'code': None,
    'obj_weakref': None
    'guarded_class': None
}

-

local_nn_module 'self.linear2' NN_MODULE
{
    'guard_types': None,
    'code': None,
    'obj_weakref': None
    'guarded_class': None
}

```

第二次，执行 forward 编译后的函数 `compiled_fn`，生成的字节码：

```

local 'a' TENSOR_MATCH
{
    'guard_types': ['TENSOR_MATCH'],

```

```

'code': None,
'obj_weakref': <weakref at 0x7f24b1567c40; to 'Tensor' at 0x7f24114e1490>
'guarded_class': <weakref at 0x7f24147b41d0; to 'torch._C._TensorMeta' at
}

-

local 'b' TENSOR_MATCH
{
'guard_types': ['TENSOR_MATCH'],
'code': None,
'obj_weakref': <weakref at 0x7f24b1567dd0; to 'Tensor' at 0x7f24114e1a80>
'guarded_class': <weakref at 0x7f24147b41d0; to 'torch._C._TensorMeta' at
}

```

动手试一试，相信你会理解的更加深刻，对于更加复杂的情况，子图中还会递归地执行 2-7 步，生成更细粒度的子图。

## InliningInstructionTranslator

如果编译的函数涉及比较复杂的函数调用，例如：

```

import logging

import torch
import torch._dynamo.config
import torch.nn as nn

TensorGuards = torch._C._dynamo.guards.TensorGuards
check_obj_id = torch._C._dynamo.guards.check_obj_id

torch._dynamo.config.log_level = logging.INFO
torch._dynamo.config.output_code = True

def add1(x, y):
    return x + y

def add2(x, y):
    return x + y

def add(x, y, z):
    return add1(add2(x, y), z)

if __name__ == '__main__':
    compiled_model = torch.compile(add)
    # 使用 Tensor 输入作为输入方便输出 Guard 和字节码
    compiled_model(torch.Tensor(1), torch.Tensor(1), torch.Tensor(1))

```

InstructionTranslator 会在解析 CALL\_FUNCTIONS 时，构建一个 InliningInstructionTranslator，获取函数的字节码，在解析字节码的过程中继续完成 graph trace。与编译子图不同的是，InliningInstructionTranslator 会进入函数，“连续”的解析字节码。函数中的字节码可以和之前解析的字节码一起进行编译优化，而编译子图意则是函数内外分开编译。此外，InliningInstructionTranslator 解析的函数也可以触发编译子图的逻辑。

至此我们梳理完了 Dynamo trace graph 的主体逻辑，Dynamo 从字节码入手，首先实现了 Python 版的虚拟机，用于解析函数的字节码，以实现 Graph trace 的功能；在此基础上，为了能够根据输入信息实现动态的 Graph trace，Dynamo 引入了 VariableTracker 以及 Guard 的概念，能够根据模型输入信息去判断是否需要触发重新编译；最后，Dynamo 通过动态地调整帧评估函数，递归地去编译在上一次编译中，重新划分的子图，实现更加灵活地 Graph trace。



公众号后台回复“CVPR2023”获取最新论文分类整理资源



极市平台

为计算机视觉开发者提供全流程算法开发训练平台，以及大咖技术分享、社区交流、竞...  
848篇原创内容

公众号

极市干货

极视角动态：推进智能矿山建设，极视角「皮带传输系列算法」保障皮带安全稳定运行！

CVPR2023：CVPR 2023 | 21 篇数据集工作总结（附打包下载链接）

数据集：垃圾分类、水下垃圾/口罩垃圾/烟头垃圾检测等相关开源数据集汇总 | 异常检测开源数据集汇总 | 语义分割方向开源数据集资源汇总

## • 极市原创作者激励计划 •



极市平台深耕CV开发者领域近6年，拥有一大批优质CV开发者受众，覆盖微信、知乎、B站、微博等多个渠道。通过极市平台，您的文章的观点和看法能分享至更多CV开发者，既能体现文章的价值，又能让文章在视觉圈内得到更大程度上的推广，并且极市还将给予优质的作者可观的稿酬！

我们欢迎领域内的各位来进行投稿或者是宣传自己/团队的工作，让知识成为最为流通的干货！

### 投稿须知：

- 1.作者保证投稿作品为自己的原创作品。
- 2.极市平台尊重原作者署名权，并支付相应稿费。文章发布后，版权仍属于原作者。
- 3.原作者可以将文章发在其他平台的个人账号，但需要在文章顶部标明首发于极市平台

### 投稿方式：

添加小编微信Fengcall（微信号：fengcall19），备注：姓名-投稿



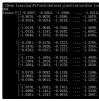
[点击阅读原文进入CV社区](#)

[收获更多技术干货](#)

[阅读原文](#)

喜欢此内容的人还喜欢

实践教程 | PyTorch数据导入机制与标准化代码模板  
极市平台



YOLOv5帮助母猪产仔？南京农业大学研发母猪产仔检测模型并部署到 Jetson Nano开发板  
极市平台



实践教程 | 使用 OpenCV 进行特征提取（颜色、形状和纹理）  
极市平台

