请同学们签到

飞桨动静统一开发范式

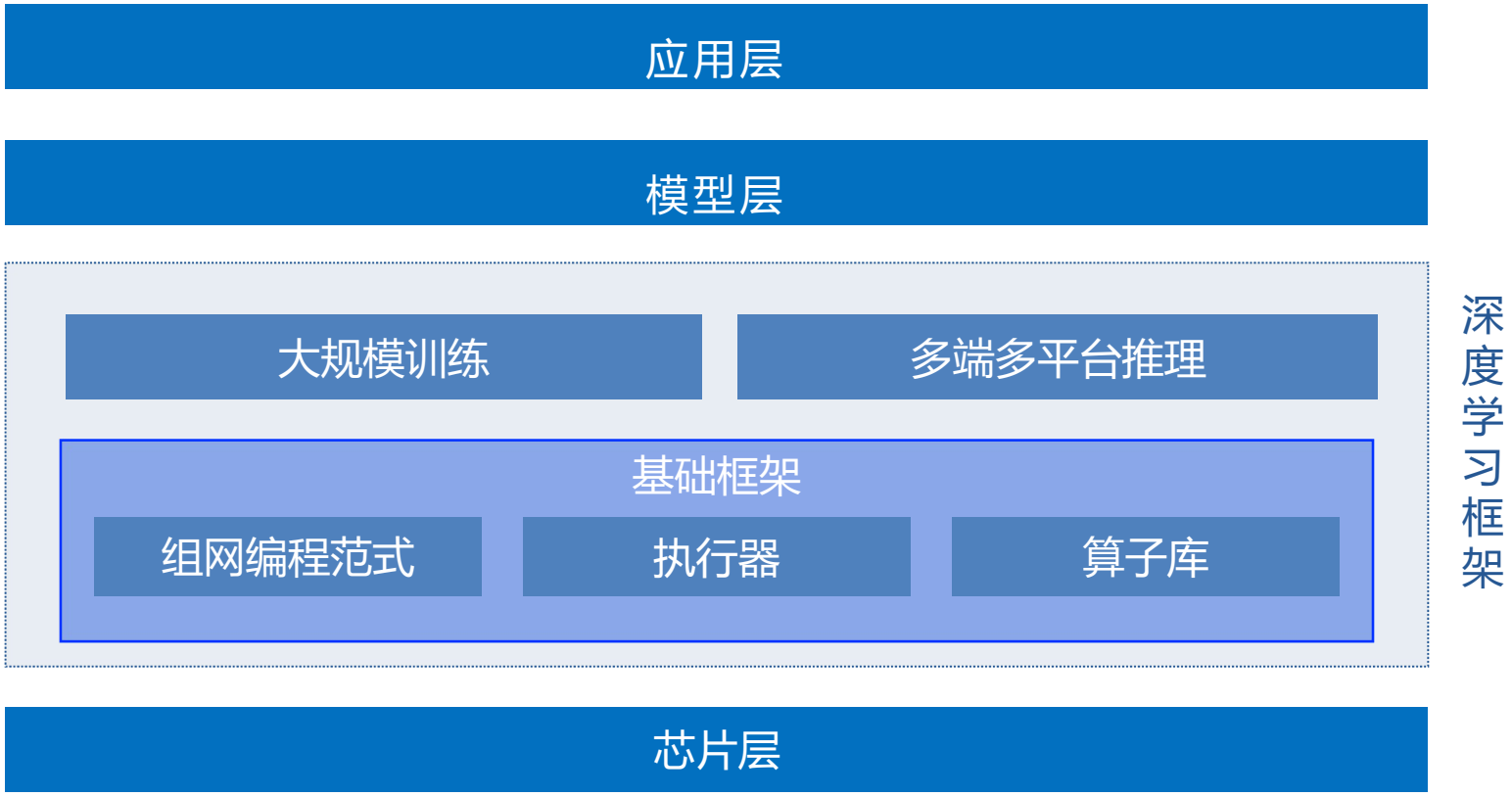
部门：深度学习技术平台部
姓名：刘红雨

飞桨动静统一开发范式

主要内容

1. 深度学习框架的发展历程
2. 动态图的优劣势分析
3. 飞桨动态图实现及动静统一特色
4. 深度学习框架的未来发展趋势

深度学习框架的定位



深度学习任务的特点

```cpp
int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[low];
    int i = low + 1, j = high;
    while (true) {
        while (i < high && arr[i] < pivot) i++;
        while (j > low && arr[j] > pivot) j--;
        if (i >= j) break;
        swap(arr[i], arr[j]);
        i++;
        j--;
    }
    swap(arr[low], arr[j]);
    return j;
}
```

快排程序关注点（处理对象是int，float等）

1. i,j 能否按照正确的逻辑进行更新

```python
def naive_atten(q, k, v, mask):
    qt = paddle.transpose(q, [0, 2, 1, 3])
    kt = paddle.transpose(k, [0, 2, 1, 3])        1
    vt = paddle.transpose(v, [0, 2, 1, 3])

    scale = 1.0 / np.sqrt(q.shape[-1])            2,4
    qt *= scale

    s = paddle.matmul(qt, kt, transpose_y=True)

    p = F.softmax(s + mask)                       3

    o = paddle.matmul(p, vt)
    return paddle.transpose(o, [0, 2, 1, 3])
```

深度学习Attention程序关注点（处理对象是Tensor）

1. 复杂的Tensor变换（不满足矩阵乘要求，进行transpose变换）
2. 保障收敛的特殊处理
3. 变长数据的处理
4. 高性能实现

深度学习任务的特点

1. 复杂的Tensor变换
   - 变换包含shape、data type、place、 value信息的变换
   - 框架本身会进行隐式变换（mask会用bool类型，运算的时候promote到float）
2. 收敛性保障
   - 特殊算子的要求（如softmax需要将输入downscale）
   - 低精度训练场景需要防止数据溢出（用float32做求和等运算）
3. 变长的数据处理
   - 理解padding处理的方式
   - mask信息传递和特殊处理（如softmax前将padding词位置的值设置为inf）
4. 高性能实现（需要大量训练数据）
   - 要了解框架执行的特点（如scale乘在小tensor上性能更好）
   - 要考虑尽可能利用CUDA的并行计算能力

深度学习任务的特点

除了组网层，在模型的训练也存在其特点

1. 大数据基础上的多轮迭代
   - 需要大量的数据才能学习到一个更好模型参数
   - 用户端看到最终效果的周期较长
2. 无严格标准答案
   - 模型收敛的效果没有一个严格的答案
   - 不像数据库场景，能到一个相对确定的结果
3. 超参数众多
   - 参数初始化、学习率、dropout的prob
   - 需要多组实验才能取得一个较好的效果

深度学习框架的要求

- 深度学习的特点要求，框架的**开发效率**和**执行效率**都非常重要
- 两者的目标在设计上存在部分的冲突，因此存在两种主流的开发范式

```
x = paddle.ones(shape=[2, 2])
y = paddle.ones(shape=[2,2])
z = x + y
print(z)
```

命令式编程范式（动态图）

```
x = paddle.ones(shape=[2, 2])
y = paddle.ones(shape=[2,2])
z = x + y

exe = paddle.static.Executor()
main_program =
paddle.static.default_main_program()
real_z = exe.run(main_program, feed={},
        fetch_list=[z.name])

print(real_z)
```

声明式编程范式（静态图）

深度学习框架开发范式对比

开发效率

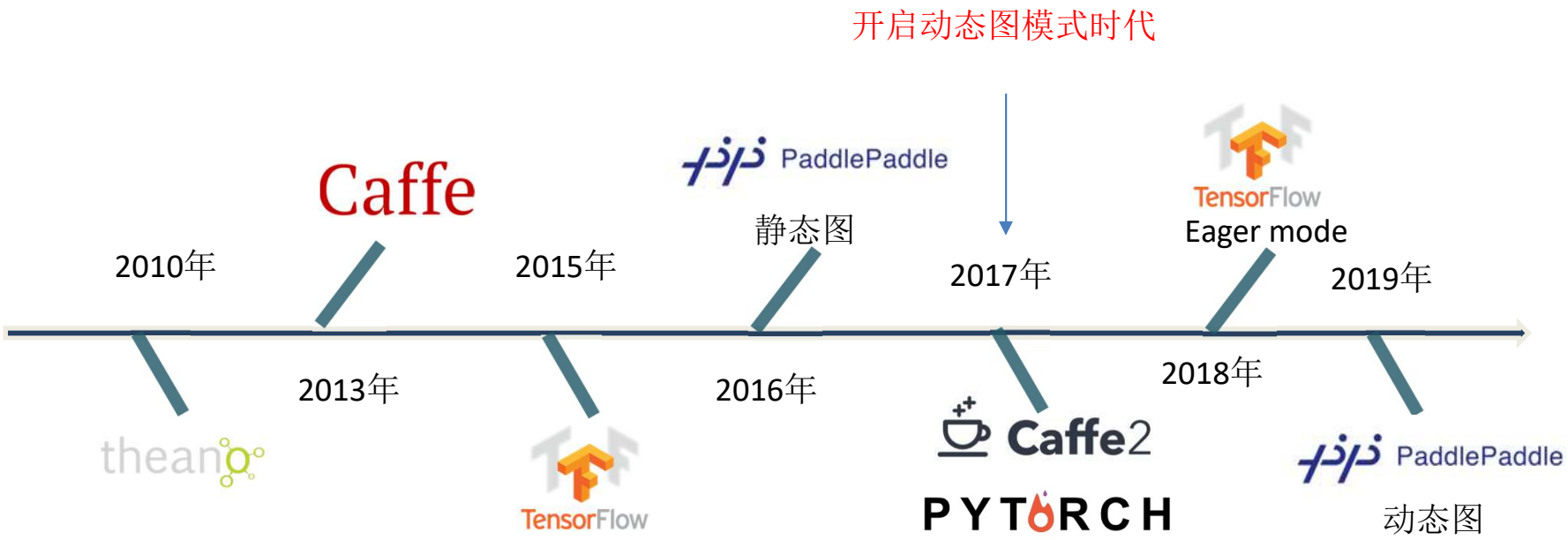
	命令式编程范式（动态图）	声明式编程范式（静态图）
复杂的Tensor变换	能够拿到真实的属性信息	无法拿到真实的属性信息，API使用成本变高
收敛性保障	很方便进行数据的分析	数据分析成本较高
变长的数据处理	复用python控制流成本较低	使用框架提供的控制流，成本较高

执行效率

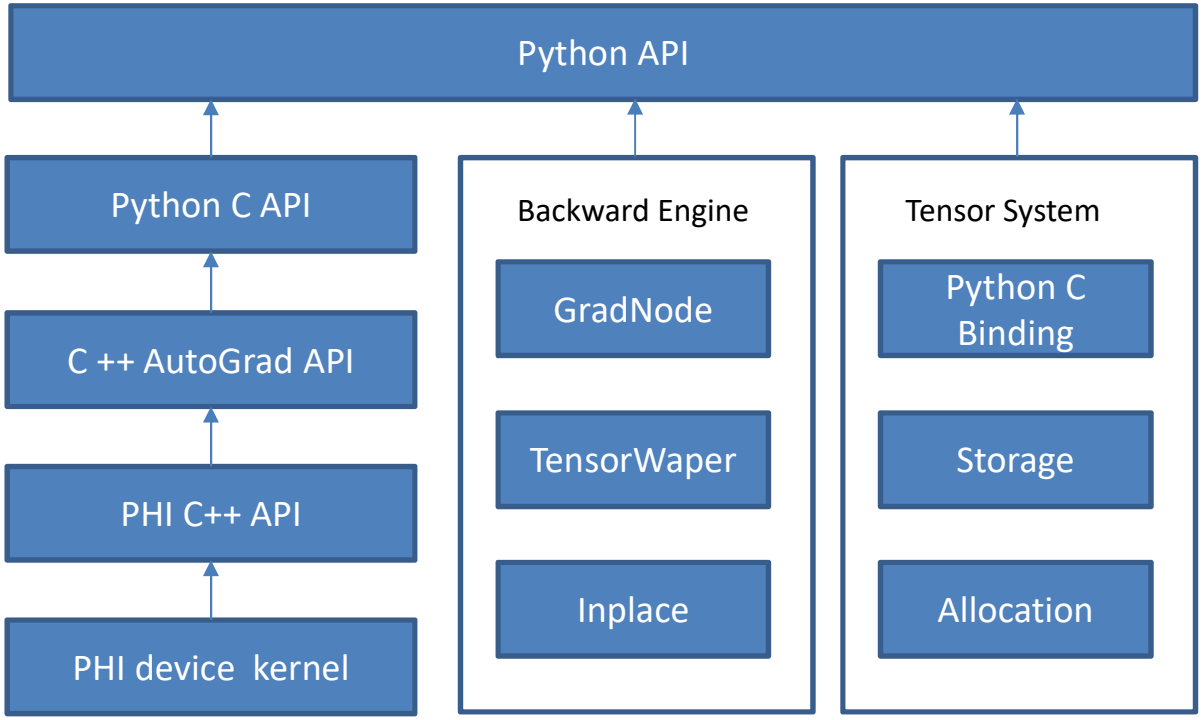
	命令式编程范式（动态图）	声明式编程范式（静态图）
高性能实现	能够更好的理解框架运行模式，写出不错的性能表现	有较多的内部优化，极限性能更好

动态图的编程范式，凭借更优异的编程范式，迅速占据市场

深度学习框架的发展历史

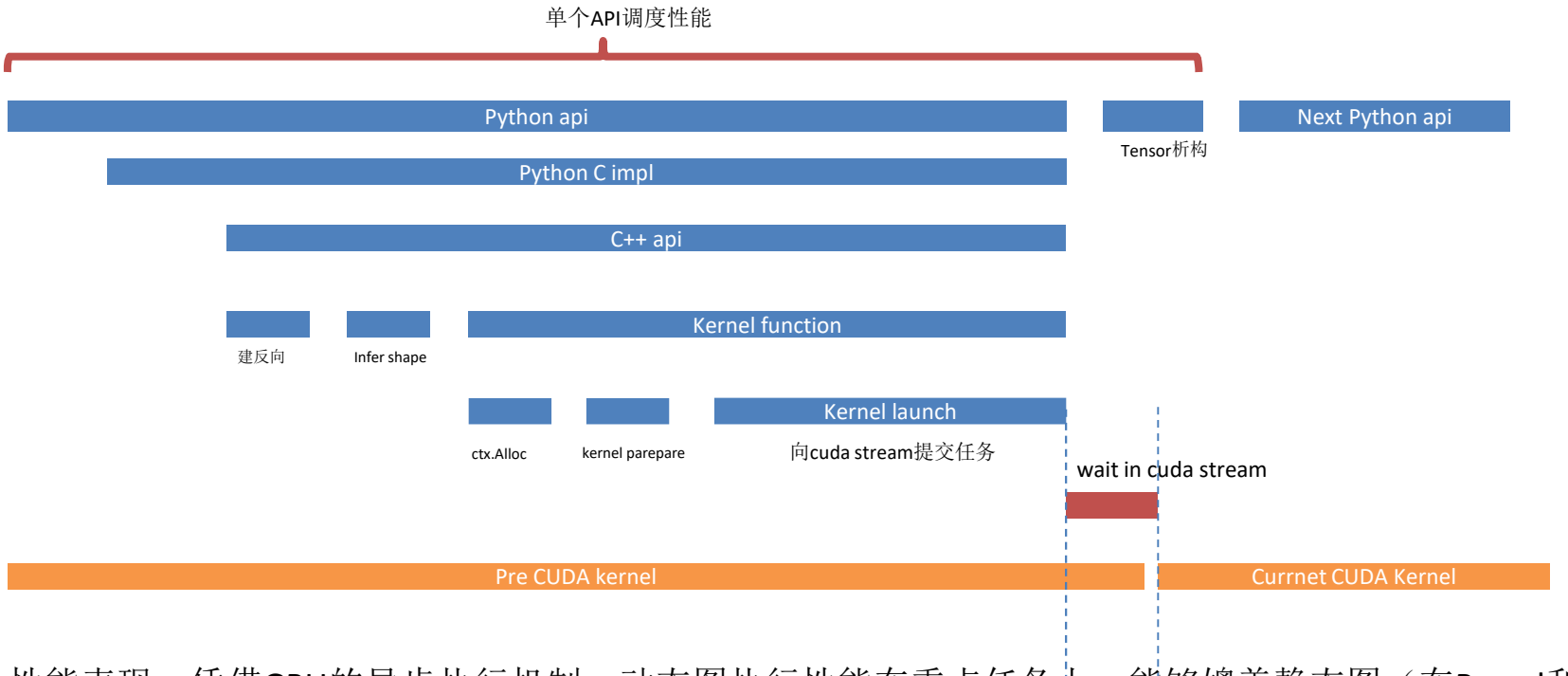


飞桨动态图的架构设计



1. 纵向函数直接调用保证执行效率
2. Backward engine保证“动态”网络的特性
3. Tensor system共享Python 和C++ Tensor，保证Pythonic的开发体验

动态图性能保障



性能表现：凭借GPU的异步执行机制，动态图执行性能在重点任务上，能够媲美静态图（在Pascal和volta架构上调度和计算取得了一个较好的平衡）

备注：GPU场景，当python端返回的时候，GPU kernel并未运行完成

动态图介绍

```
import numpy as np
import paddle
```

环境准备

```
model = paddle.nn.Linear(32, 32)
```

模型，参数初始化

```
x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")
```

准备输入数据

```
res = model(x)
target = paddle.sum(res)
```

模型前向运行和反向图构造

```
target.backward()
```

模型反向运行

```
opt = paddle.optimizer.Momentum(parameters=model.parameters())
```

优化器定义

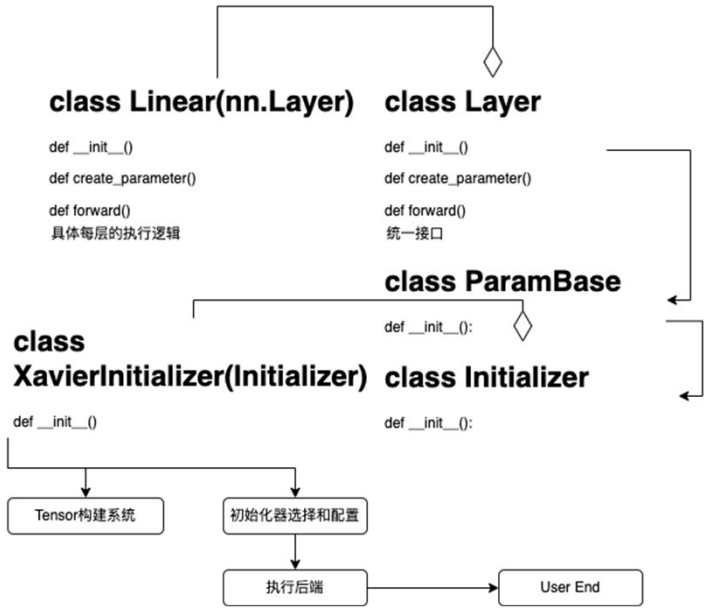
```
opt.step()
```

参数更新

动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer.
Momentum(parameters=model.parameters())

opt.step()
```

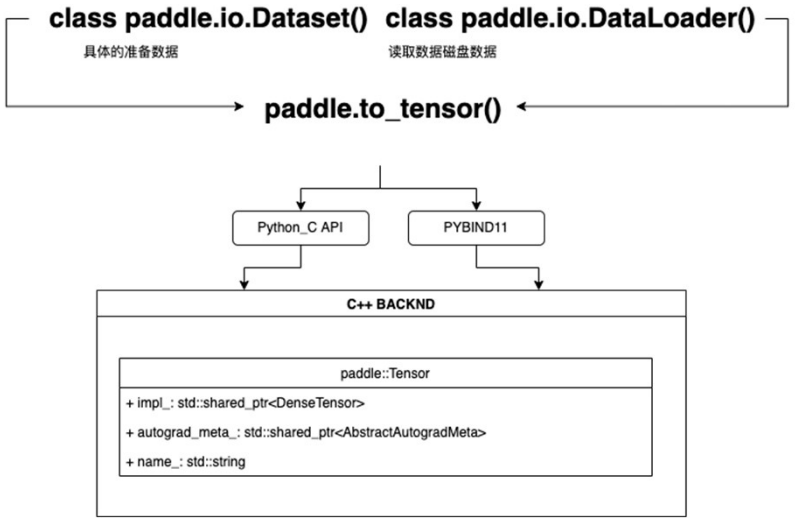
1. 模型和参数初始化



动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer.
Momentum(parameters=model.parameters())

opt.step()
```

2. 输入数据准备



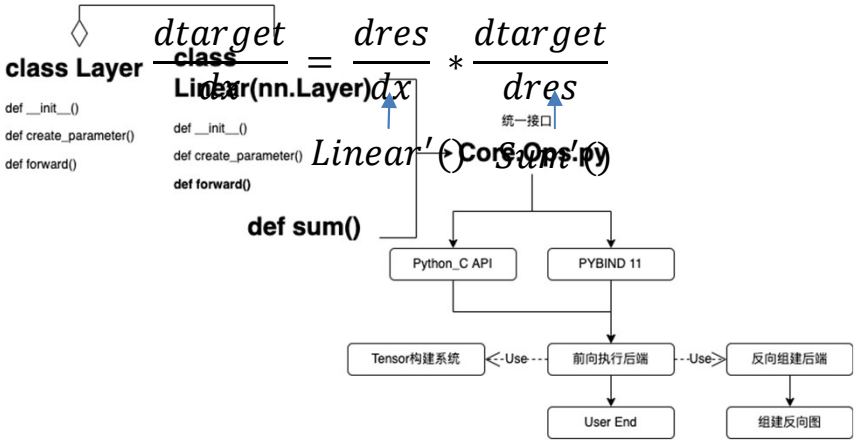
动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer.
Momentum(parameters=model.parameters())

opt.step()
```

3. 前向运行和反向图构造

$$res = Linear(x), target = sum(res)$$

$$\frac{dtarget}{dx} = \frac{dres}{dx} * \frac{dtarget}{dres}$$



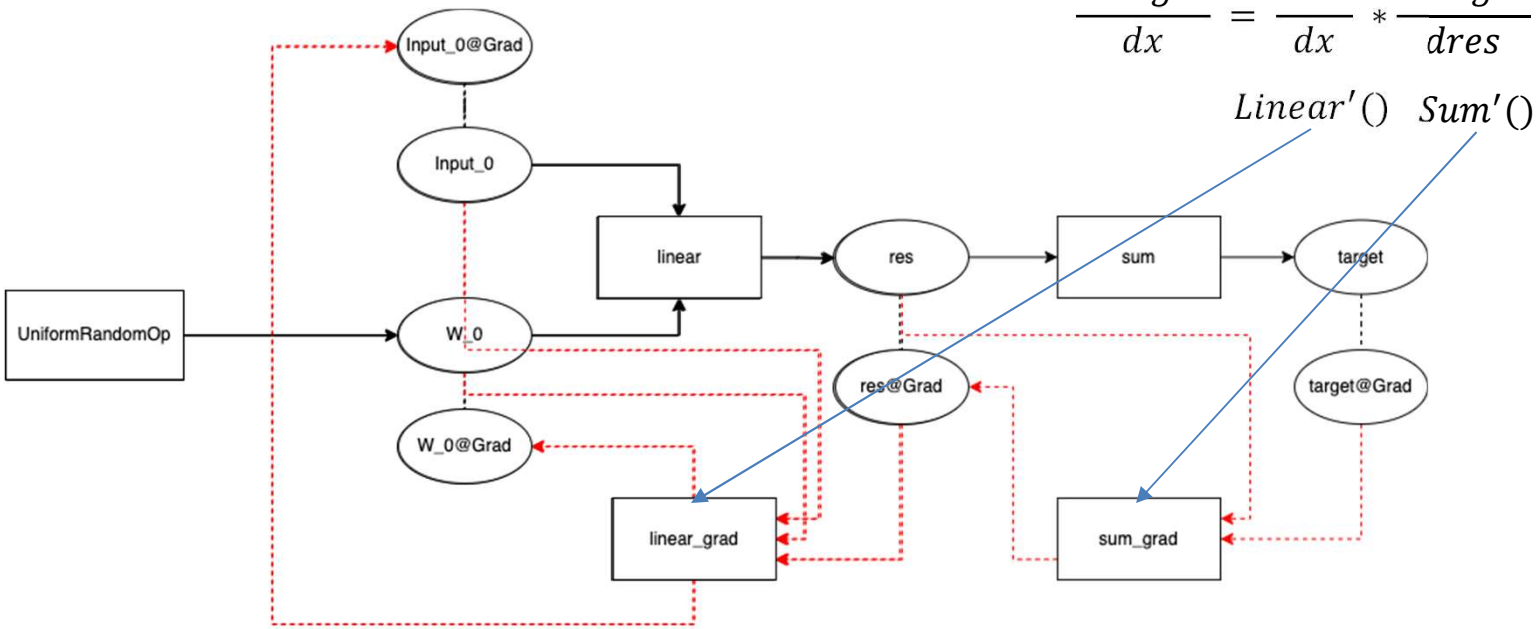
动态图介绍

计算图

$$res = Linear(x), target = sum(res)$$

$$\frac{dtarget}{dx} = \frac{dres}{dx} * \frac{dtarget}{dres}$$

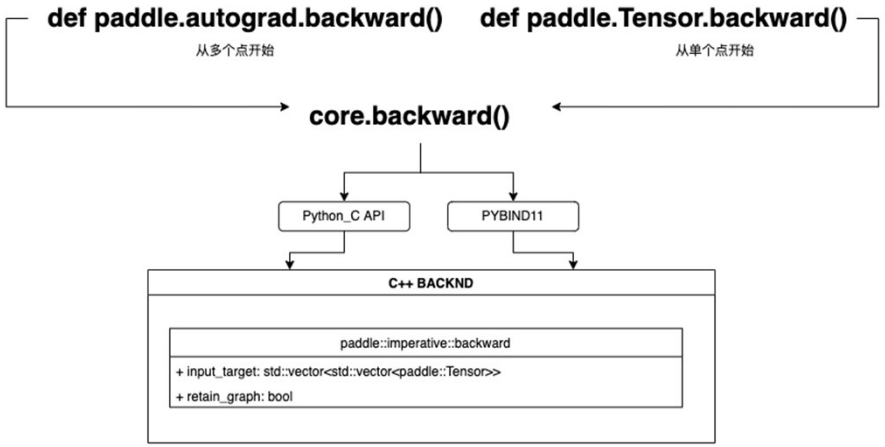
$Linear'()$ $Sum'()$



动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer.
Momentum(parameters=model.parameters())

opt.step()
```

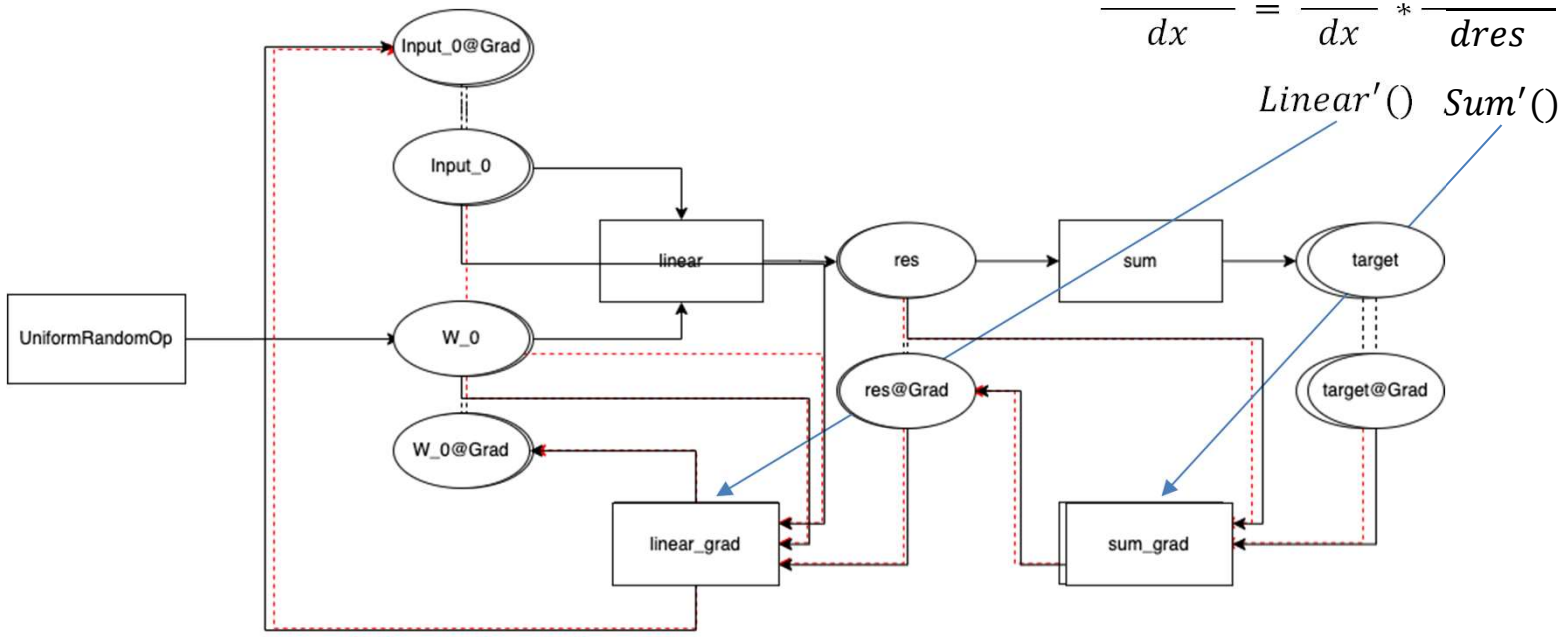
4. 反向运行并组建反向



动态图介绍

$$res = Linear(x), target = sum(res)$$

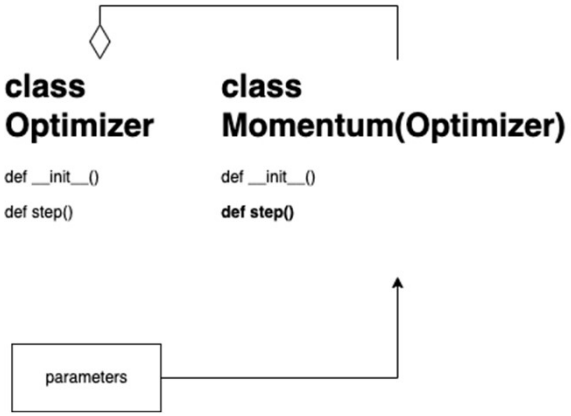
$$\frac{dtarget}{dx} = \frac{dres}{dx} * \frac{dtarget}{dres}$$



动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer.
Momentum(parameters=model.parameters())

opt.step()
```

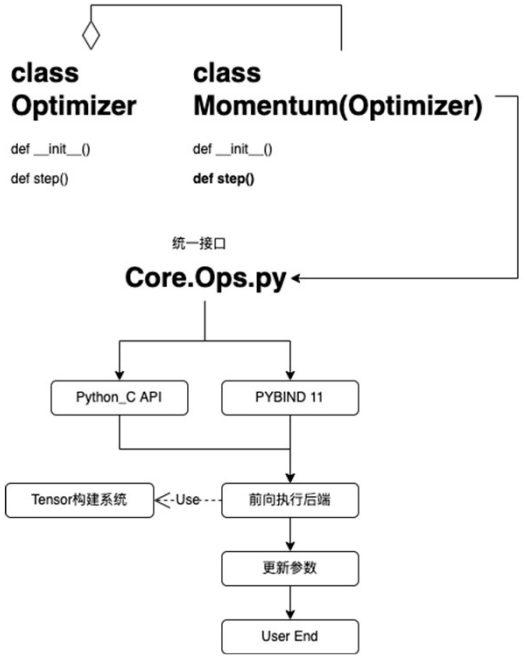
5. 初始化优化器



动态图介绍

```
import numpy as np
import paddle

model = paddle.nn.Linear(32, 32)

x_data = np.random.random([1,1,1,32]).astype("float32")
x = paddle.to_tensor(x_data, name="input_data")

res = model(x)
target = paddle.sum(res)

target.backward()

opt = paddle.optimizer. Momentum(parameters=model.parameters())

opt.step()
```

6. 参数更新



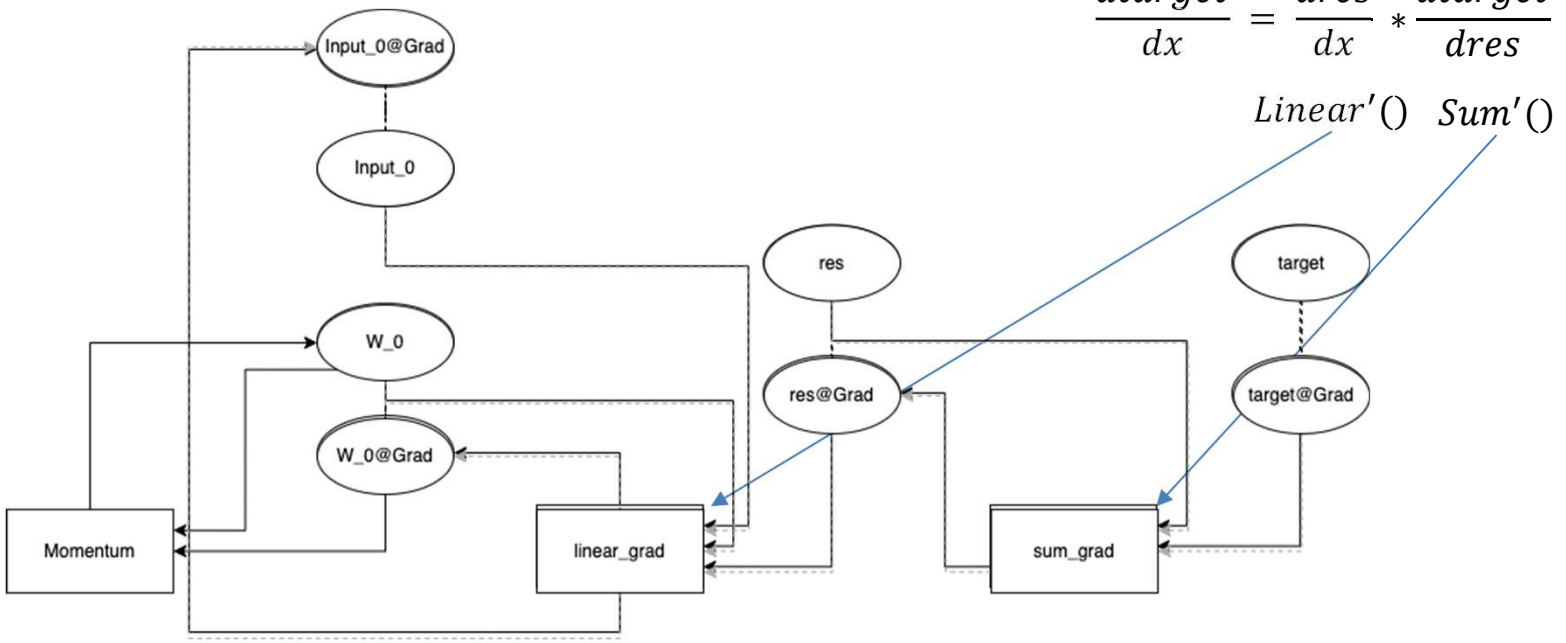
动态图介绍

计算图

$$res = Linear(x), target = sum(res)$$

$$\frac{dtarget}{dx} = \frac{dres}{dx} * \frac{dtarget}{dres}$$

$Linear'()$ $Sum'()$



动态图开发范式的劣势

1. 推理部署
   - 无法嵌入到C++业务环境中部署
   - 部署工具依赖于静态图 IR 表示
   - PY端部署依赖模型源码，分发成本高

2. 性能优化
   - 无法复用静态图的加速策略
   - 无法和编译器优化技术相结合

3. 开发效率
   - 大模型场景（4D并行）动态图的上手成本较高

飞桨动静统一开发范式介绍

为解决动态图开发范式的劣势，飞桨提出了动静统一开发范式

- 前端语言
- 中间表达和优化
- 调度执行和底层数据
- 设备代码

飞桨动静统一开发范式介绍

```
@jit.to_static()
def f(x, y):
    z = x + y

x = paddle.ones(shape=[2, 2])
y = paddle.ones(shape=[2,2])

z = f(x,y)
print(z)

Or

paddle.jit.save(f, path)
```

仅需一个装饰器，即可完成将动态图的代码转成静态图进行执行或者进行保存

飞桨动静统一开发范式介绍

飞桨实现动静态统一的方式：通过将动态图代码自动转换为静态图代码的方案

当任务中存在数据依赖的控制流时，动静态图的表示方式差异较大，转化难度较大

1. 依赖Tensor的值

2. 依赖Tenor的shape信息

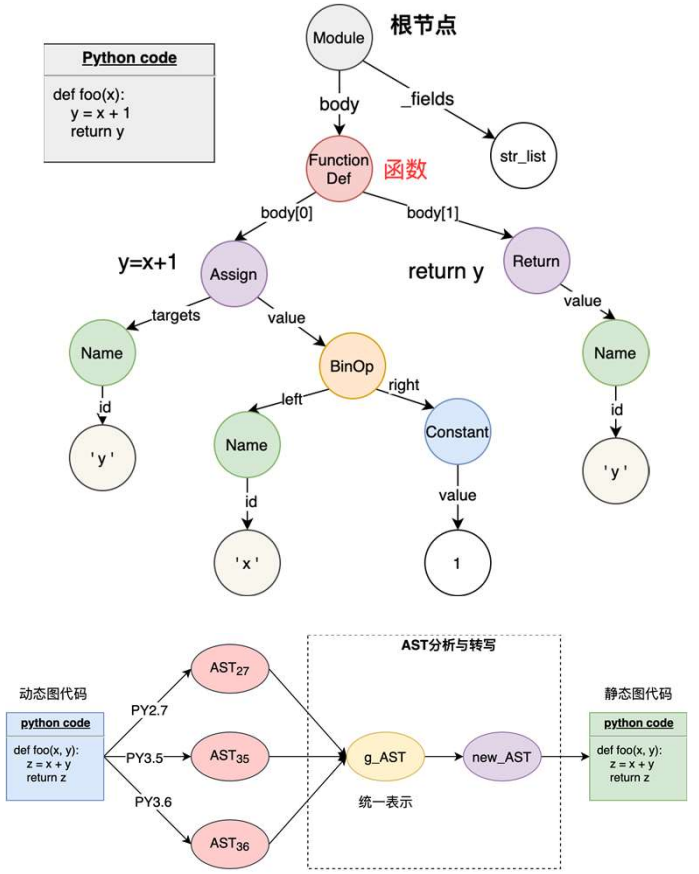
解决方案：提出基于AST的转写技术，解决动静态图差异较大的问题

```
# Case 1:
if x > y:
  return x + y
else:
  return x - y

# Case 2:
bs, seq_len, _ = x.shape

res = []
for i in range(seq_len):
  out = fn( in )
  res.append(out)
  in = out
```

动转静介绍

- **AST：抽象语法树（Abstract Syntax Tree）**
  - 是个「树」，N叉树结构，根节点总是Module，每个节点都是一个Node
  - **有限集合**，Node主要包含两大类型：Operator、Data（文法规则）
  - **语法信息**，涵盖了PY代码所有必要的信息：函数、变量、关键字等

- **工具：GAST 库**
  - **AST 获取**，PY一切皆为Object，把用户函数对象转为AST对象
  - **AST 遍历**，在PY解释器下，从根节点遍历，按需解析和转写
  - 版本兼容，不同PY版本AST的文法规则有差异，GAST屏蔽了此差异性



动转静介绍

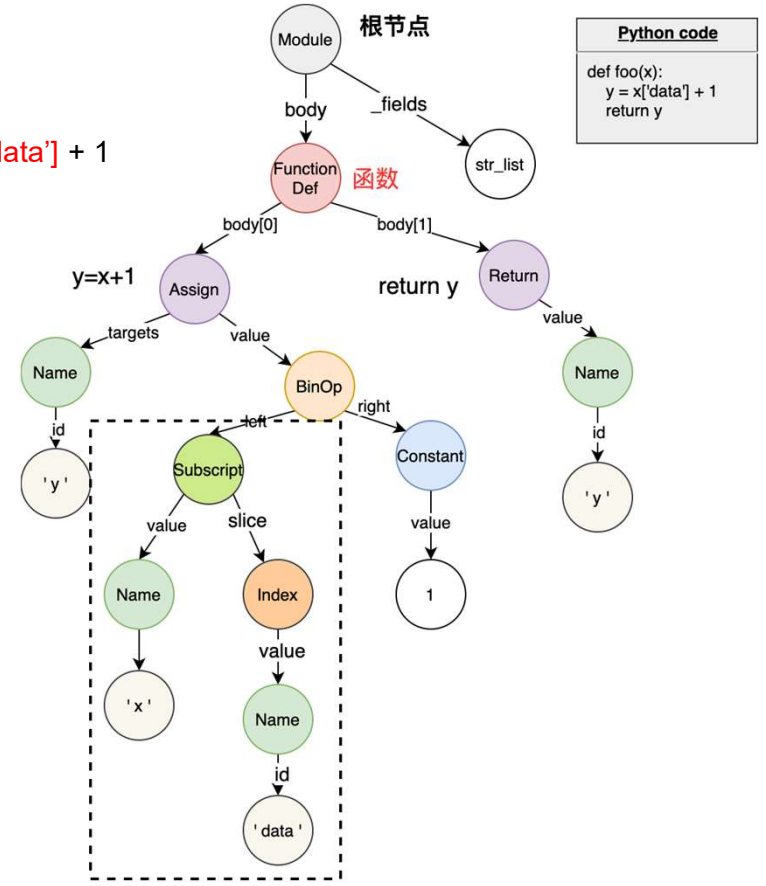
- 设计模式：访问者模式（Visitor）

y = x + 1 改为 y = x[‘data’] + 1

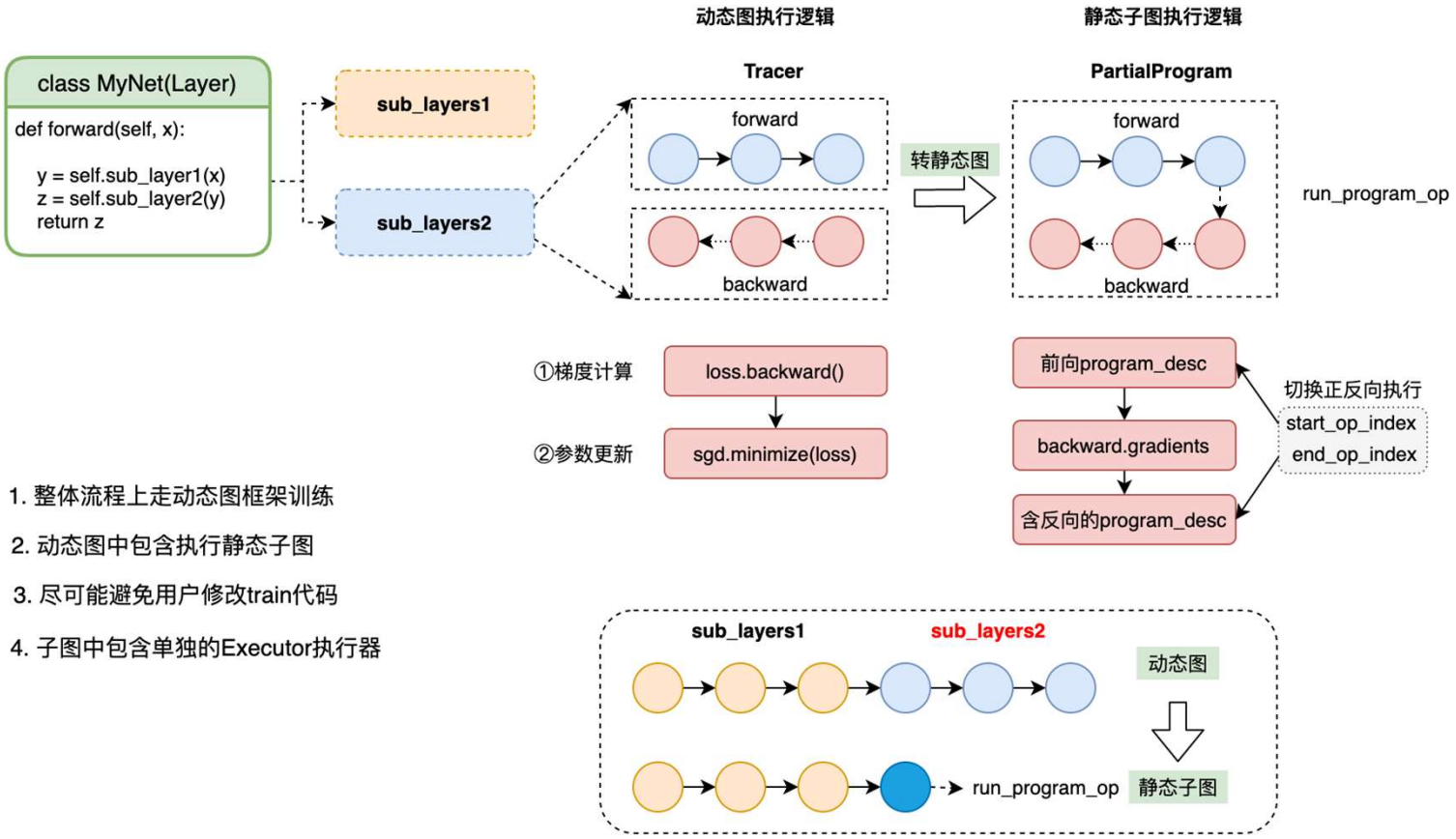
```
class NodeVisitor(object):

    def visit(self, node):
        """Visit a node."""
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)
```

- AST 自定义改写

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):
        if node.id == 'x':
            node = Subscript(
                value=Name(id='data', ctx=Load()),
                slice=Index(value=Constant(value=node.id)),
                ctx=node.ctx)
        return node

foo_node = ast.parse(inspect.getsource(foo))
node = RewriteName().visit(foo_node)
```



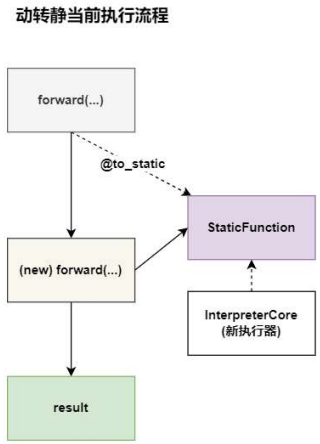
动转静介绍

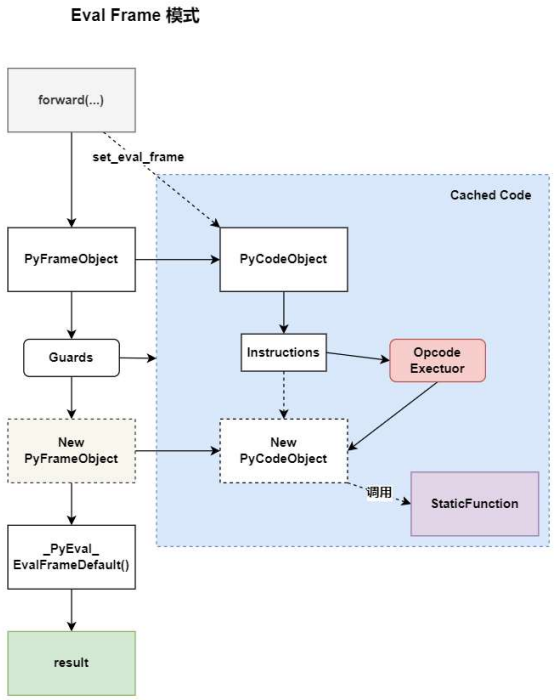

动态图执行逻辑
静态子图执行逻辑
class MyNet(Layer)
def forward(self, x):
y = self.sub_layer1(x)
z = self.sub_layer2(y)
return z
sub_layers1
sub_layers2
Tracer
forward
backward
转静态图
PartialProgram
forward
backward
run_program_op
①梯度计算
loss.backward()
②参数更新
sgd.minimize(loss)
前向program_desc
backward.gradients
含反向的program_desc
切换正反向执行
start_op_index
end_op_index
1. 整体流程上走动态图框架训练
2. 动态图中包含执行静态子图
3. 尽可能避免用户修改train代码
4. 子图中包含单独的Executor执行器
sub_layers1
sub_layers2
动态图
run_program_op
静态子图

动转静新技术探索

当前AST方案存在的局限

1. 难以处理动态和静态相互混合的场景
2. 控制流和容器的混合使用时有边界 case
3. 不支持源码加密场景下使用，完备性存在上限
4. 无法支持组合算子场景未知shape的情况

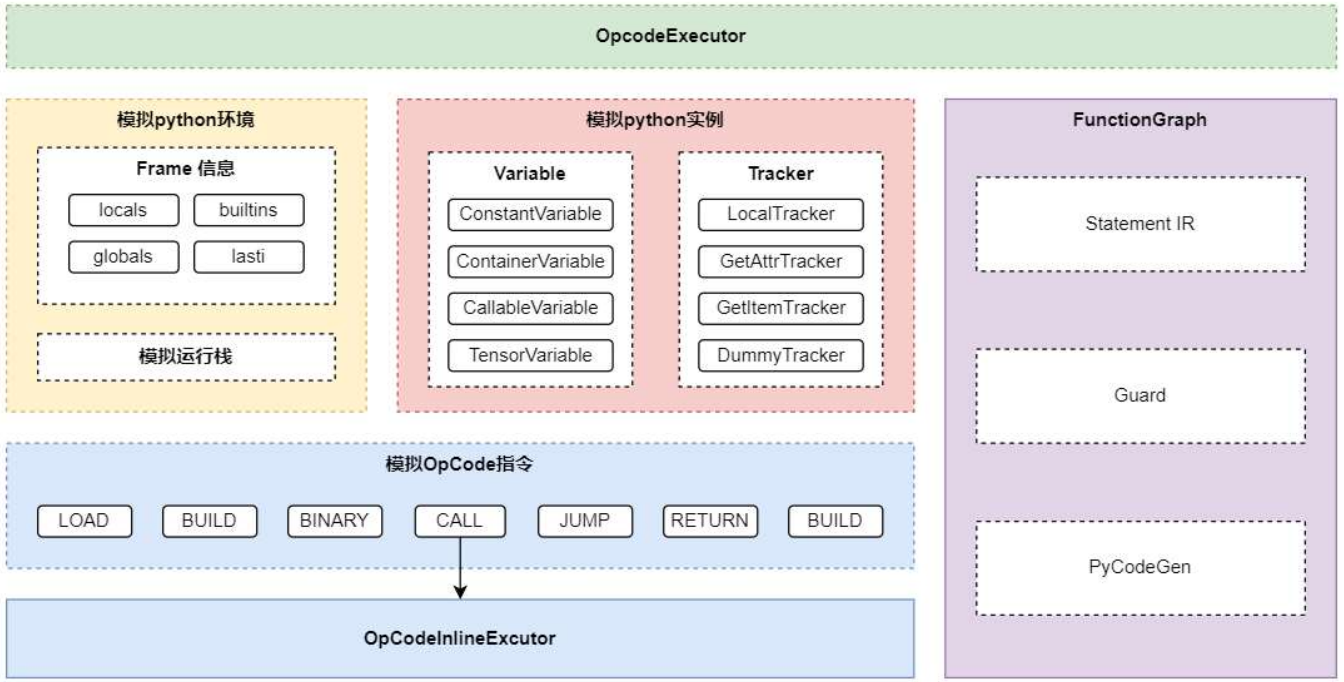




动转静新技术探索

字节码模拟运行

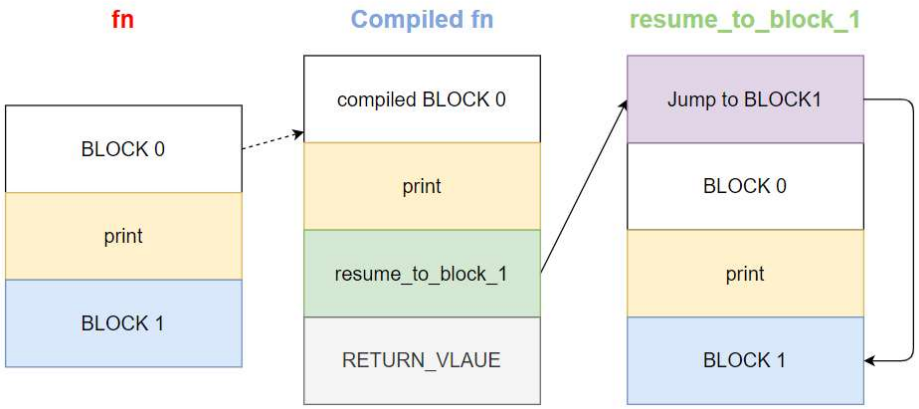
```
def hello()
    print("Hello, World!")



0 LOAD_GLOBAL      0 (print)
2 LOAD_CONST       1 ('Hello, World!')
4 CALL_FUNCTION    1
```



动转静新技术探索

子图fallback

```
def fn(x):
    x = x + 1
    print(x)
    x = x * 2
    return x

x = paddle.to_tensor([1])
out = symbolic_trace(fn)(x)
print(out)
```



深度学习框架未来的发展趋势

问：动态图模式会长期占据主导地位吗？

答：我觉得纯动态图模式不会，一定还是动静结合的路线

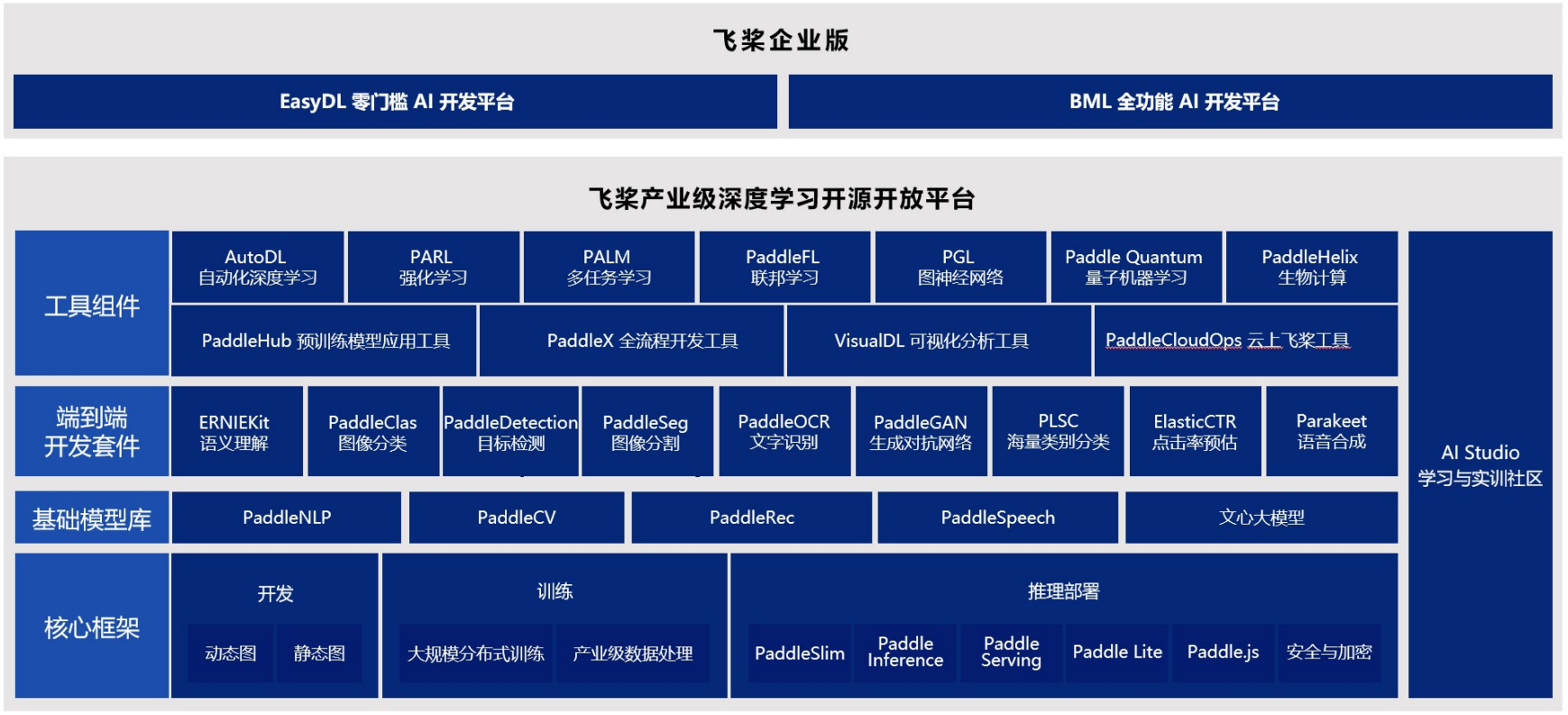
理由：

1. 硬件算力的发展速度是远要大于CPU性能、总线带宽、访存带宽三者的发展速度，这样会导致动态图的速度会存在较大的差距
   - CPU和总线带宽会使得调度会成为任务的瓶颈
   - 访存的瓶颈会使得任务中，访存密集型的算子耗时占比会增加
2. 动态图支持大模型的上手成本太高
   - 用户需要手动处理分布式的并行策略，Tensor需要手工推导单卡的信息

业界动向：

1. 飞桨较早就提出了动静统一的设计模式
2. Pytorch在2.0版本提出了compile的方式，将动态图转成静态图，并使用编译器方式进行优化

飞桨全景图


飞桨企业版
EasyDL 零门槛 AI 开发平台
BML 全功能 AI 开发平台
飞桨产业级深度学习开源开放平台
工具组件
AutoDL 自动化深度学习
PARL 强化学习
PALM 多任务学习
PaddleFL 联邦学习
PGL 图神经网络
Paddle Quantum 量子机器学习
PaddleHelix 生物计算
PaddleHub 预训练模型应用工具
PaddleX 全流程开发工具
VisualDL 可视化分析工具
PaddleCloudOps 云上飞桨工具
端到端开发套件
ERNIEKit 语义理解
PaddleClas 图像分类
PaddleDetection 目标检测
PaddleSeg 图像分割
PaddleOCR 文字识别
PaddleGAN 生成对抗网络
PLSC 海量类别分类
ElasticCTR 点击率预估
Parakeet 语音合成
基础模型库
PaddleNLP
PaddleCV
PaddleRec
PaddleSpeech
文心大模型
核心框架
开发
动态图
静态图
训练
大规模分布式训练
产业级数据处理
推理部署
PaddleSlim
Paddle Inference
Paddle Serving
Paddle Lite
Paddle.js
安全与加密
AI Studio 学习与实训社区

谢谢

问卷调研

扫一扫二维码分享