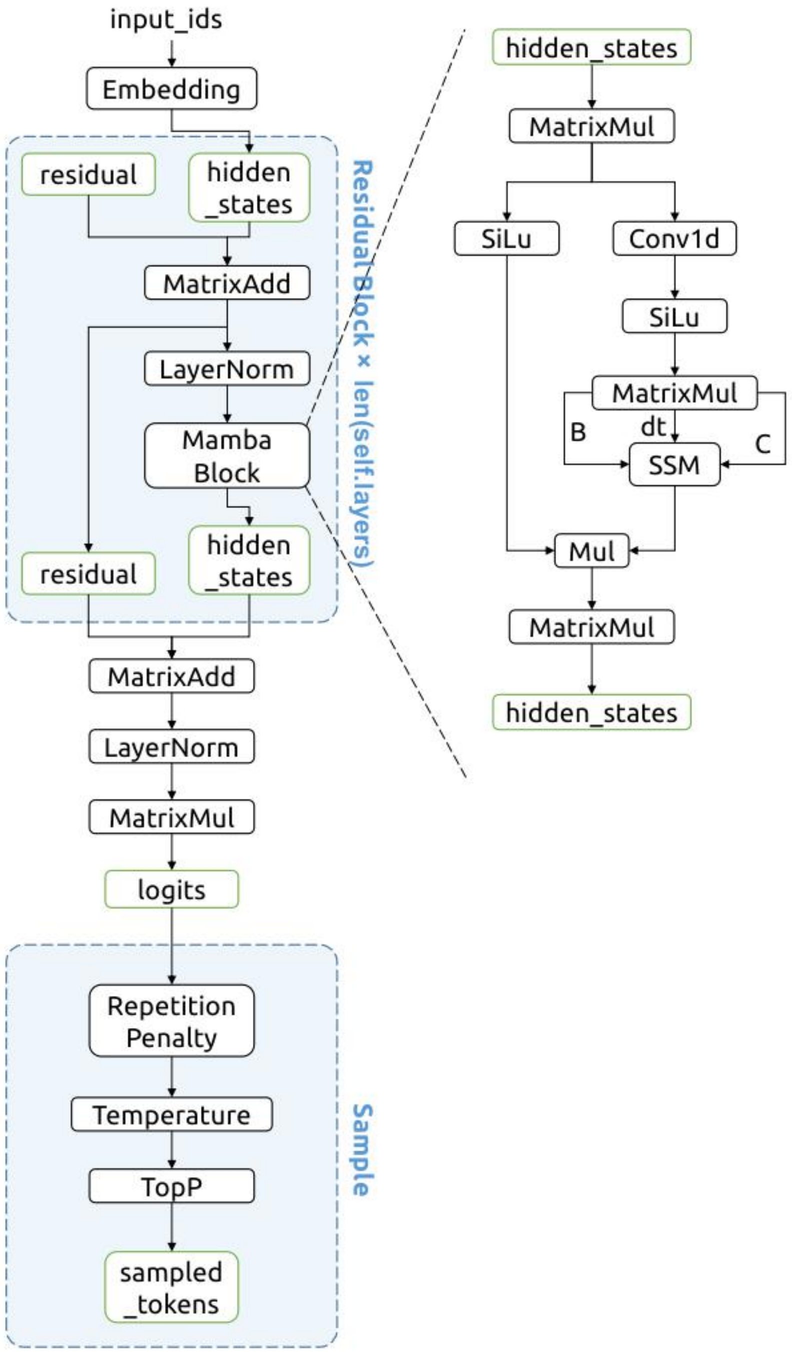
Mamba Source Code

◆ 代码见mamba_ssm/models/mixer_seq_simple.py的MambaLMHeadModel。
◆ input_ids是用户输入的prompt，是一个id（与embedding有关）的序列。
◆ mamba的主体部分是self.backbone，输出hidden_states，其shape是[1,n,k]，这里n指prompt长度，而k指词向量维数（=d_model）。num_last_tokens在目标配置下始终为1，因此会取hidden_states的最后一行，然后送入线性层self.lm_head映为[1,1,m]的向量，这里m指词汇表大小（=vocab_size）。
◆ 输出的lm_logits会经历一系列层作Sampling，最后得到sampled_tokens，同样是一个id。这个id会append到prompt的末尾，然后对新的prompt重新运行这个模型。
◆ 在模型第二次推理时，实际上只截取prompt的最后一个id，然后利用之前计算保留的中间结果推理。

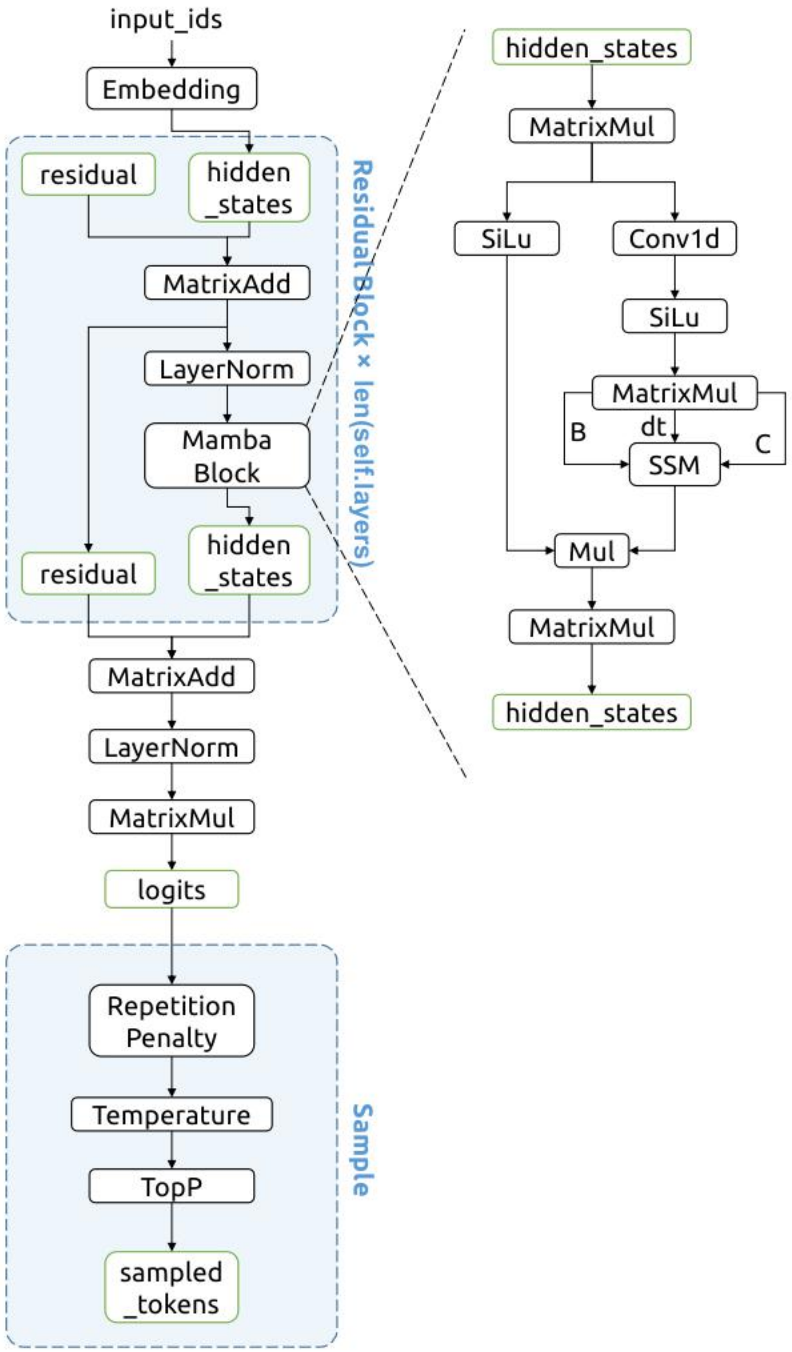



```python
def forward(self, input_ids, position_ids=None, inference_params=None, num_last_tokens=0):
    """
    "position_ids" is just to be compatible with Transformer generation. We don't use it.
    num_last_tokens: if > 0, only return the logits for the last n tokens
    """
    hidden_states = self.backbone(input_ids, inference_params=inference_params)
    if num_last_tokens > 0:
        hidden_states = hidden_states[:, -num_last_tokens:]
    lm_logits = self.lm_head(hidden_states)
    CausalLMOutput = namedtuple("CausalLMOutput", ["logits"])
    return CausalLMOutput(logits=lm_logits)
```

Mamba Source Code

◆ 代码见mamba_ssm/models/mixer_seq_simple.py的MixerModel。
◆ self.embedding做的事情是把input_ids中的id作为索引，在embedding矩阵（可以理解为字典，第j行保存id=j的token的词向量）中取出对应行。最终得到的hidden_states是[1,n,m]（第二次推理开始都是[1,1,m]）。
◆ self.norm_f是nn.LayerNorm，用于作逐行正规化。

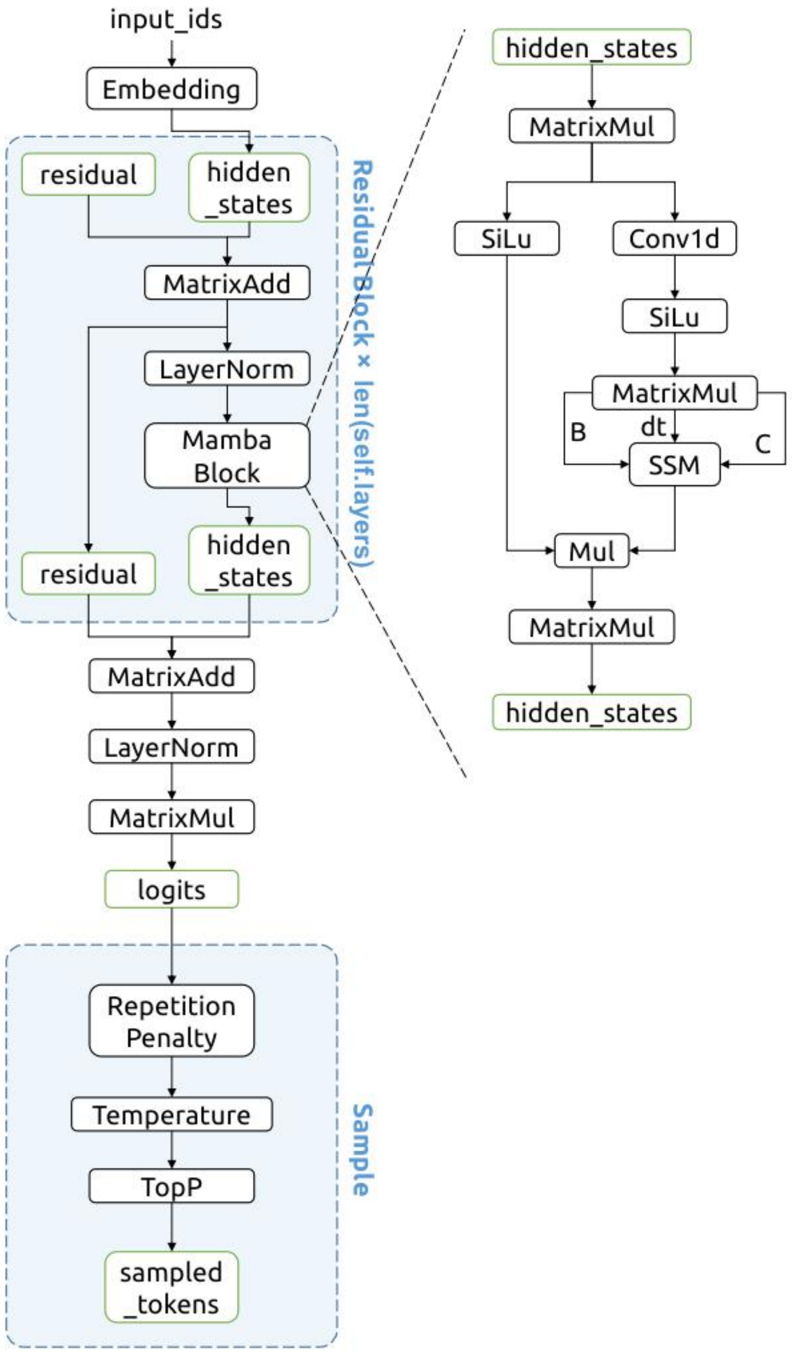
```
def forward(self, input_ids, inference_params=None):
    hidden_states = self.embedding(input_ids)

    residual = None
    for layer in self.layers:
        hidden_states, residual = layer(
            hidden_states, residual, inference_params=inference_params
        )
    if not self.fused_add_norm:
        residual = (hidden_states + residual) if residual is not None else hidden_states
        hidden_states = self.norm_f(residual.to(dtype=self.norm_f.weight.dtype))
    else:
        # Set prenorm=False here since we don't need the residual
        fused_add_norm_fn = rms_norm_fn if isinstance(self.norm_f, RMSNorm) else layer_norm_fn
        hidden_states = fused_add_norm_fn(
            hidden_states,
            self.norm_f.weight,
            self.norm_f.bias,
            eps=self.norm_f.eps,
            residual=residual,
            prenorm=False,
            residual_in_fp32=self.residual_in_fp32,
        )
    return hidden_states
```



Mamba Source Code

- ◆ 代码见mamba_ssm/modules/mamba_simple.py的Block。
- ◆ self.mixer就是图示中的MambaBlock。

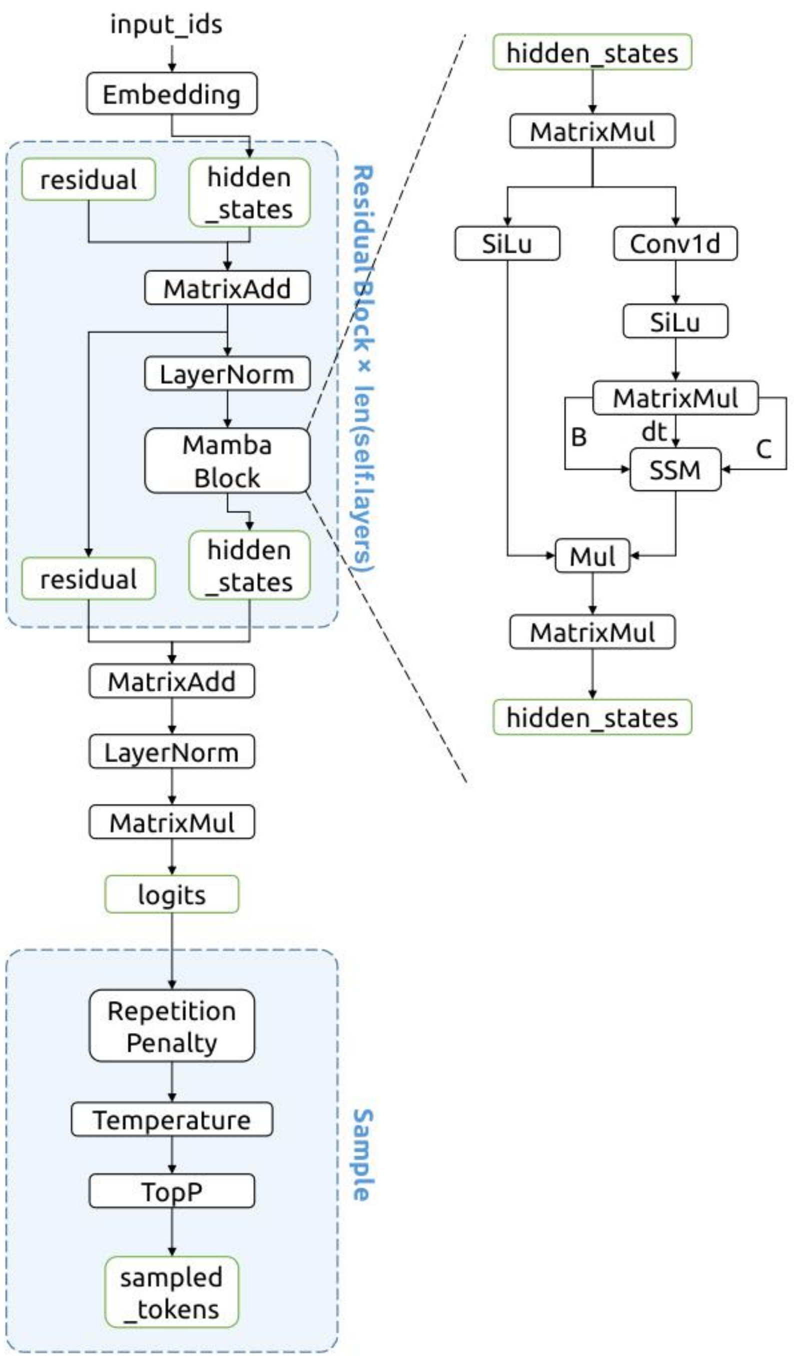
```python
def forward(
    self, hidden_states: Tensor, residual: Optional[Tensor] = None, inference_params=None
):
    r"""Pass the input through the encoder layer.

    Args:
        hidden_states: the sequence to the encoder layer (required).
        residual: hidden_states = Mixer(LN(residual))
    """
    if not self.fused_add_norm:
        residual = (hidden_states + residual) if residual is not None else hidden_states
        hidden_states = self.norm(residual.to(dtype=self.norm.weight.dtype))
        if self.residual_in_fp32:
            residual = residual.to(torch.float32)
    else:
        fused_add_norm_fn = rms_norm_fn if isinstance(self.norm, RMSNorm) else layer_norm_fn
        hidden_states, residual = fused_add_norm_fn(
            hidden_states,
            self.norm.weight,
            self.norm.bias,
            residual=residual,
            prenorm=True,
            residual_in_fp32=self.residual_in_fp32,
            eps=self.norm.eps,
        )
    hidden_states = self.mixer(hidden_states, inference_params=inference_params)
    return hidden_states, residual
```



Mamba Source Code

◆ 代码见mamba_ssm/modules/mamba_simple.py的Mamba。
◆ 第一个if语句块用来判断是不是第一次推理，如果不是第一次推理就只需要把ssm的状态“推进一步”即调用self.step。
◆ xz是hidden_states作用线性层self.in_proj得到的结果，shape是[1,n,2*D]，这里D=d_inner=d_model * expand=2m。xz后续将会等分为[1,n,D]的两份x,z。

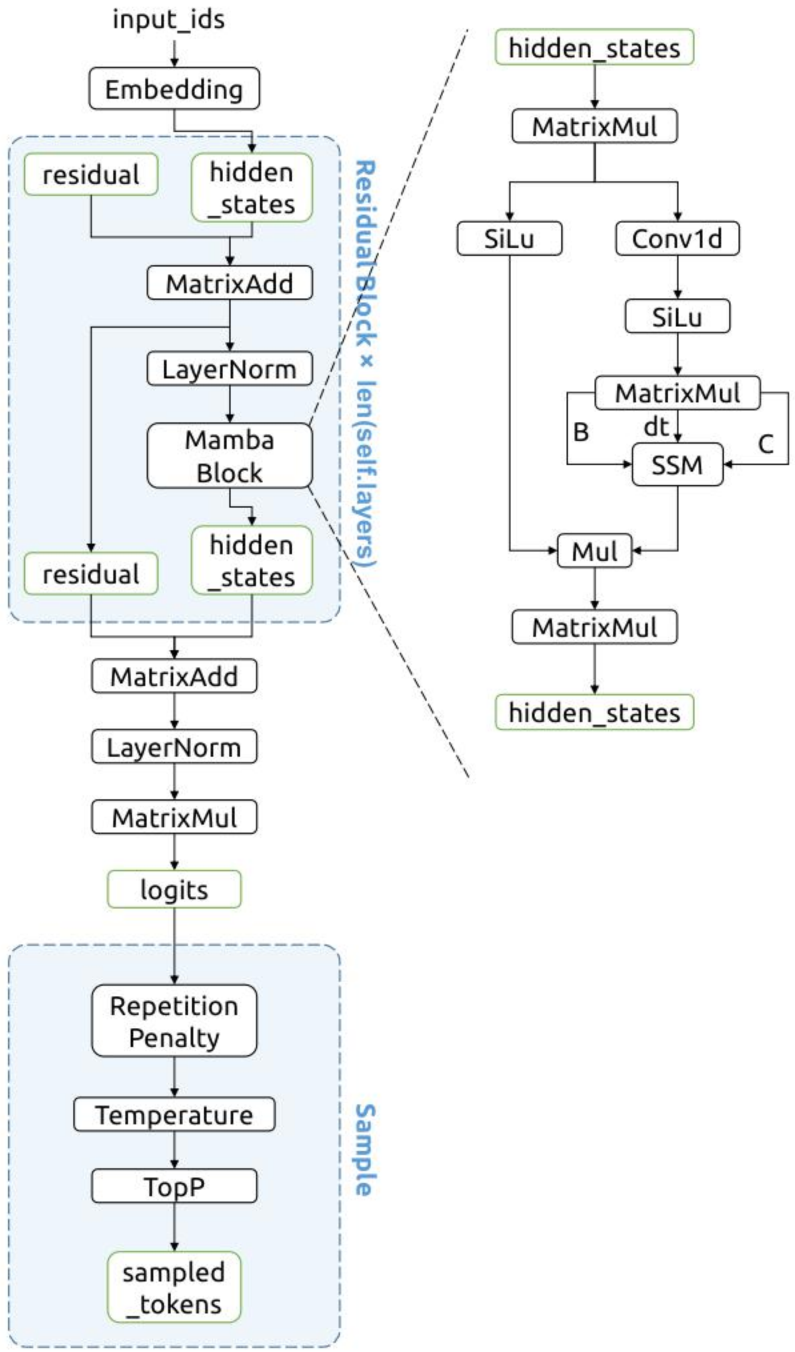
```python
def forward(self, hidden_states, inference_params=None):
    """
    hidden_states: (B, L, D)
    Returns: same shape as hidden_states
    """
    batch, seqlen, dim = hidden_states.shape

    conv_state, ssm_state = None, None
    if inference_params is not None:
        conv_state, ssm_state = self._get_states_from_cache(inference_params, batch)
        if inference_params.seqlen_offset > 0:
            # The states are updated inplace
            out, _, _ = self.step(hidden_states, conv_state, ssm_state)
            return out

    # We do matmul and transpose BLH → HBL at the same time
    xz = rearrange(
        self.in_proj.weight @ rearrange(hidden_states, "b l d → d (b l)"),
        "d (b l) → b d l",
        l=seqlen,
    )
    if self.in_proj.bias is not None:
        xz = xz + rearrange(self.in_proj.bias.to(dtype=xz.dtype), "d → d 1")
```



Mamba Source Code

◆ 代码见mamba_ssm/modules/mamba_simple.py的Mamba。
◆ xz是hidden_states作用线性层self.in_proj得到的结果，shape是[1,n,2*D]，这里D=d_inner=d_model * expand=2m。xz后续将会等分为[1,n,D]的两份x,z。
◆ x被pad以后送入self.conv1d，这是nn.Conv1d层，得到的结果作用激活函数SiLu后预备送入ssm。

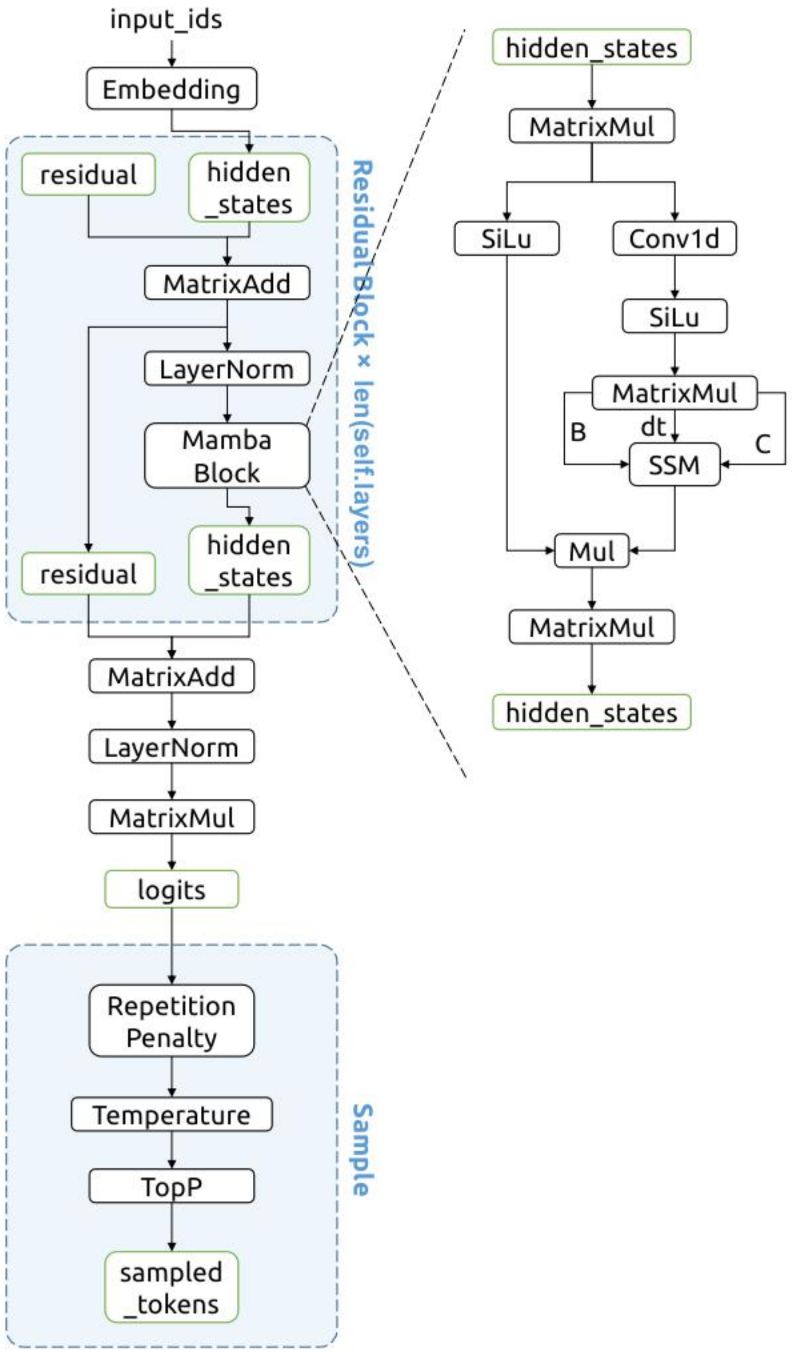
```
x, z = xz.chunk(2, dim=1)
# Compute short convolution
if conv_state is not None:
    # If we just take x[:, :, -self.d_conv :], it will error if seqlen < self.d_conv
    # Instead F.pad will pad with zeros if seqlen < self.d_conv, and truncate otherwise.
    conv_state.copy_(F.pad(x, (self.d_conv - x.shape[-1], 0)))  # Update state (B D W)
if causal_conv1d_fn is None:
    x = self.act(self.conv1d(x)[..., :seqlen])
else:
    assert self.activation in ["silu", "swish"]
    x = causal_conv1d_fn(
        x=x,
        weight=rearrange(self.conv1d.weight, "d 1 w -> d w"),
        bias=self.conv1d.bias,
        activation=self.activation,
    )
```



Mamba Source Code

◆ 代码见mamba_ssm/modules/mamba_simple.py的Mamba。
◆ x被进一步送入线性层self.x_proj，这个线性层把d_inner维的向量转化为dt_rank+d_state * 2维的，这会分解成依赖输入的参数B,C和dt，然后实施ssm。
◆ ssm输出*z得到y同样是[1,n,D]的，经过线性层self.out_proj变回[1,n,m]。

```python
    # We're careful here about the layout, to avoid extra transposes.
    # We want dt to have d as the slowest moving dimension
    # and L as the fastest moving dimension, since those are what the ssm_scan kernel expects.
    x_dbl = self.x_proj(rearrange(x, "b d l → (b l) d"))  # (bl d)
    dt, B, C = torch.split(x_dbl, [self.dt_rank, self.d_state, self.d_state], dim=-1)
    dt = self.dt_proj.weight @ dt.t()
    dt = rearrange(dt, "d (b l) → b d l", l=seqlen)
    B = rearrange(B, "(b l) dstate → b dstate l", l=seqlen).contiguous()
    C = rearrange(C, "(b l) dstate → b dstate l", l=seqlen).contiguous()
    assert self.activation in ["silu", "swish"]
    y = selective_scan_fn(
        x,
        dt,
        A,
        B,
        C,
        self.D.float(),
        z=z,
        delta_bias=self.dt_proj.bias.float(),
        delta_softplus=True,
        return_last_state=ssm_state is not None,
    )
    if ssm_state is not None:
        y, last_state = y
        ssm_state.copy_(last_state)
    y = rearrange(y, "b d l → b l d")
    out = self.out_proj(y)
return out
```



Secret-Sharing

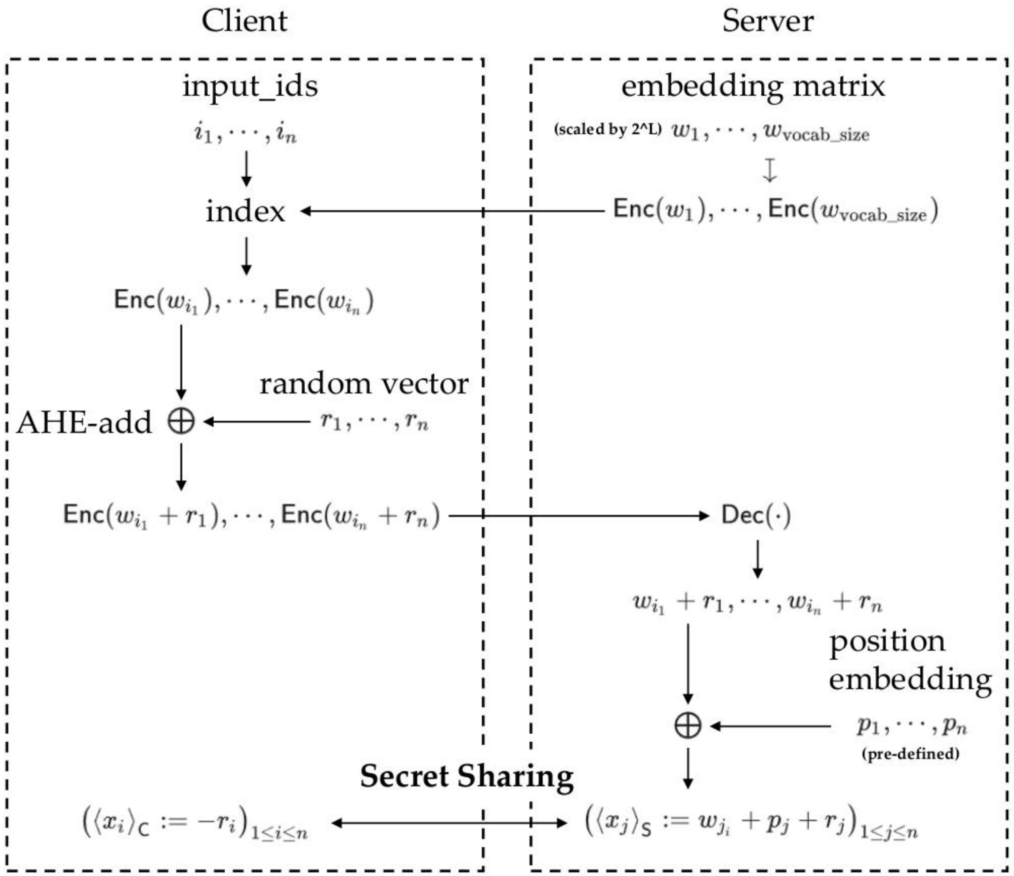
◆ 2-PC的一个安全计算任务是指，有一个可计算函数$f:(x,y)\mapsto(f_0(x,y),f_1(x,y))$，两个参与方0,1分别拥有x,y，他们希望在“保护隐私”的同时计算这个函数，然后使得0拥有f_0(x,y)，1拥有f_1(x,y)。

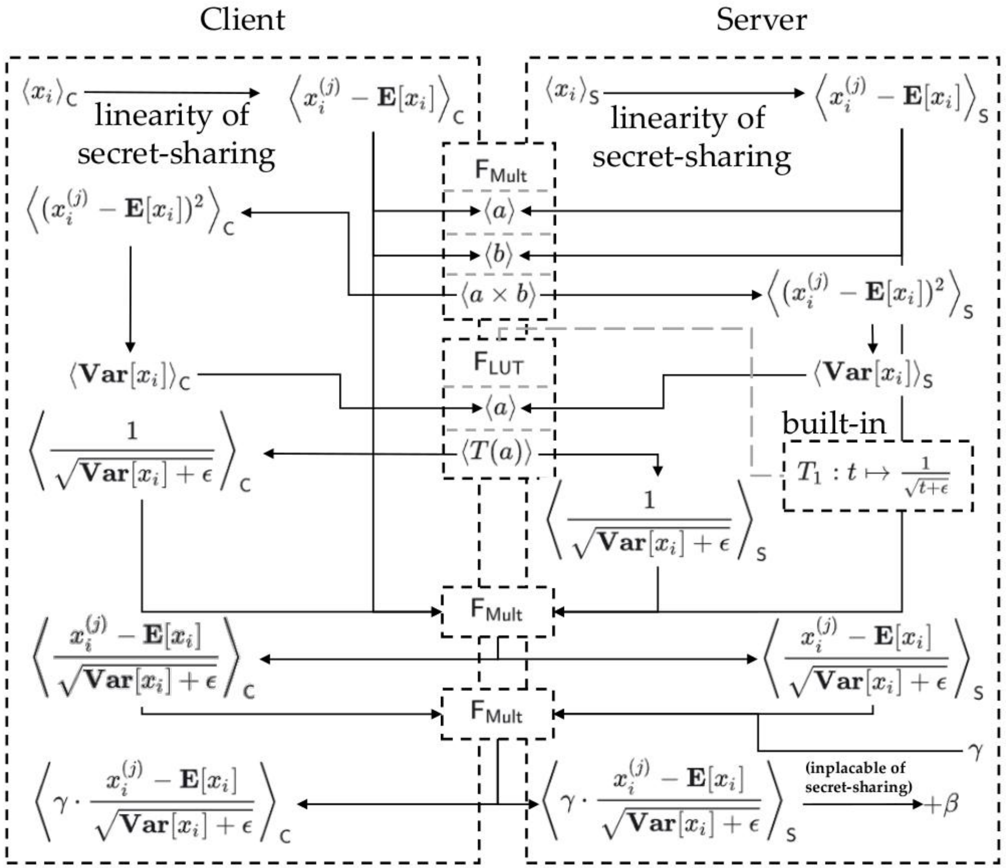
◆ 秘密分享是一种具体的技术。在cipherGPT中采用的是加性秘密分享。一些术语如下：

>>> 第0方和第1方秘密分享了x := 第0方拥有<x>_0，第1方拥有<x>_1，并且x=<x>_0+<x>_1；

>>> 第0方和第1方调用了2-PC协议$\langle z\rangle\leftarrow\pi(\langle x\rangle,\langle y\rangle)$:= 第0方和第1方分别秘密分享了x和y，按照协议制定的操作做运算，最终秘密分享了z；

◆ 实际的2-PC协议往往包括本地计算、调用子协议、消息传递三种操作。





CipherMamba

```python
device = "cuda"
tokenizer = AutoTokenizer.from_pretrained("EleutherAI/gpt-neox-20b")

HOST = "127.0.0.1"
PORT = 43222

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    ss = BetterSocket(s)
    protocol.set_socket(s=ss, role="C")
    print("Successfully connect to the cipher-mamba server")
    while True:
        prompt = input("Input the prompt here: ")
        tokens = tokenizer(prompt, return_tensors="pt")
        input_ids = tokens.input_ids.to(device=device)
        ss.sendall(input_ids) # actually, C should not send input_ids to S

        msg, ret = protocol.synchronize('C', input_ids=input_ids)
        while msg ≠ 'break':
            if msg == 'onemore':
                input_ids = ret
            msg, ret = protocol.synchronize('C', input_ids=input_ids)

        # secure-sharing computation

        out = ss.recv()
        response = tokenizer.batch_decode(out)
        print("Response: ", response)
```

```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    print("Ready to receive the message now.")
    conn, addr = s.accept()
    with conn:
        connn = BetterSocket(conn)
        protocol.set_socket(s=connn, role="S")

        while True:
            input_ids = connn.recv()

            # inference
            max_length = input_ids.shape[1] + args.genlen

            fn = lambda: model.generate(
                input_ids=input_ids,
                max_length=max_length,
                cg=True,
                return_dict_in_generate=True,
                output_scores=True,
                enable_timing=False,
                temperature=args.temperature,
                top_k=args.topk,
                top_p=args.topp,
                min_p=args.minp,
                repetition_penalty=args.repetition_penalty,
            )
            print("Going to generate...")
            out = fn()
            protocol.synchronize('S', message="break")
            out_sequences = out.sequences
            out = out.sequences.tolist()
            connn.sendall(out)
```

CipherMamba

```python
def forward(self, input_ids, inference_params=None):
    # true_hidden_states = self.embedding(input_ids)
    vocab_size = self.embedding.num_embeddings
    W = self.embedding(torch.arange(vocab_size).reshape(1, 1, vocab_size).to('cuda'))
    protocol.synchronize('S', message="embedding")

    hidden_states = protocol.insecure_embedding('S', W=W)
    if inference_params.seqlen_offset > 0:
        hidden_states = hidden_states[0][-1].reshape((1, 1, -1)).to(device='cuda', dtype=torch.float16)
```

```python
def synchronize(self, role, message=None, input_ids=None, token=None):
    if role == 'C':
        s = self.socket_c
        msg = s.recv()
        if msg == 'embedding':
            self.insecure_embedding('C', input_ids=input_ids)
            return msg, None
        elif msg == 'onemore':
            token = s.recv()
            return msg, torch.cat((input_ids, token), 1)
        elif msg == 'break':
            return msg, None
    else:
        s = self.socket_s
        s.sendall(message)
        if message == 'onemore':
            s.sendall(token)
        return None
```

CipherMamba

```
embedding_first_time = True
def insecure_embedding(self, role, input_ids=None, W=None):
    if role == 'C':
        s = self.socket_c
        k = s.recv()
        m = s.recv()

        if self.embedding_first_time == True:
            self.Enc_W = torch.zeros((1, 1, k, m)).to('cuda')
            for i in range(k):
                Enc_W_list = s.recv()
                self.Enc_W[0][0][i] = torch.as_tensor(Enc_W_list)

        n = input_ids.shape[1]
        Enc_w = torch.zeros((1, 1, n, m)).to('cuda')
        Enc_w[0][0] = torch.index_select(self.Enc_W[0][0], 0, input_ids
        r = torch.randn_like(Enc_w).to('cuda')
        Enc_w_plus_r = AHE_add(Enc_w, r)

        s.sendall(n)
        for i in range(n):
            Enc_w_plus_r_list = Enc_w_plus_r[0][0][i].tolist()
            s.sendall(Enc_w_plus_r_list)
        self.x_after_embedding_c = (-1) * r

        # insecure reveal

        s.sendall(self.x_after_embedding_c)
        self.embedding_first_time = False
        return None
    else:
        s = self.socket_s
        k = W.shape[2]
        m = W.shape[3]
        s.sendall(k)
        s.sendall(m)

        if self.embedding_first_time == True:
            for i in range(k):
                Enc_W_list = Enc(W[0][0][i]).tolist()
                s.sendall(Enc_W_list)
        n = s.recv()
        Enc_w_plus_r = torch.zeros((1, 1, n, m)).to('cuda')
        for i in range(n):
            Enc_w_plus_r_list = s.recv()
            Enc_w_plus_r[0][0][i] = torch.as_tensor(Enc_w_plus_r_list)
        w_plus_r = Dec(Enc_w_plus_r)
        self.x_after_embedding_s = w_plus_r

        # insecure reveal

        x = s.recv()
        x = x + self.x_after_embedding_s
        self.embedding_first_time = False
        return x[0].to(torch.float16)
```