在 Pytorch 中一种模型保存和加载的方式如下：

```
# save
torch.save(model.state_dict(), PATH)

# load
model = MyModel(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

`model.state_dict()`其实返回的是一个`OrderDict`，存储了网络结构的名字和对应的参数，下面看看源代码如何实现的。

state_dict

```
# torch.nn.modules.module.py
class Module(object):
    def state_dict(self, destination=None, prefix='', keep_vars=False):
        if destination is None:
            destination = OrderedDict()
            destination._metadata = OrderedDict()
        destination._metadata[prefix[:-1]] = local_metadata =
dict(version=self._version)
        for name, param in self._parameters.items():
            if param is not None:
                destination[prefix + name] = param if keep_vars else
param.data
        for name, buf in self._buffers.items():
            if buf is not None:
                destination[prefix + name] = buf if keep_vars else buf.data
        for name, module in self._modules.items():
            if module is not None:
                module.state_dict(destination, prefix + name + '.',
keep_vars=keep_vars)
        for hook in self._state_dict_hooks.values():
            hook_result = hook(self, destination, prefix, local_metadata)
            if hook_result is not None:
                destination = hook_result
        return destination
```

可以看到state_dict函数中遍历了4中元素，分别是`_paramters`，`_buffers`，`_modules`和`_state_dict_hooks`，前面三者在之前的文章已经介绍区别，最后一种就是在读取`state_dict`时希望执行的操作，一般为空，所以不做考虑。另外有一点需要注意的是，在读取`Module`时采用的递归的读取方式，并且名字间使用`.`做分割，以方便后面`load_state_dict`读取参数。

```
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.my_tensor = torch.randn(1) # 参数直接作为模型类成员变量
        self.register_buffer('my_buffer', torch.randn(1)) # 参数注册为 buffer
        self.my_param = nn.Parameter(torch.randn(1))
        self.fc = nn.Linear(2,2,bias=False)
        self.conv = nn.Conv2d(2,1,1)
        self.fc2 = nn.Linear(2,2,bias=False)
        self.f3 = self.fc
    def forward(self, x):
        return x

model = MyModel()
print(model.state_dict())
>>>OrderedDict([('my_param', tensor([-0.3052])), ('my_buffer', tensor([0.5583])),
('fc.weight', tensor([[ 0.6322, -0.0255],
    [-0.4747, -0.0530]])), ('conv.weight', tensor([[[[ 0.3346]],

     [[-0.2962]]]])), ('conv.bias', tensor([0.5205])), ('fc2.weight', tensor([[-0.4949,
0.2815],
    [ 0.3006,  0.0768]])), ('f3.weight', tensor([[ 0.6322, -0.0255],
    [-0.4747, -0.0530]]))])
```

load_state_dict

下面的代码中我们可以分成两个部分看，

1. `load(self)`

这个函数会递归地对模型进行参数恢复，其中的`_load_from_state_dict`的源码附在文末。

首先我们需要明确`state_dict`这个变量表示你之前保存的模型参数序列，而`_load_from_state_dict`函数中的`local_state`表示你的代码中定义的模型的结构。

那么`_load_from_state_dict`的作用简单理解就是假如我们现在需要对一个名为`conv.weight`的子模块做参数恢复，那么就以递归的方式先判断`conv`是否在`staet__dict`和`local_state`中，如果不在就把`conv`添加到`unexpected_keys`中去，否则递归的判断`conv.weight`是否存在，如果都存在就执行`param.copy_(input_param)`，这样就完成了`conv.weight`的参数拷贝。

1. `if strict:`

这个部分的作用是判断上面参数拷贝过程中是否有`unexpected_keys`或者`missing_keys`，如果有就报错，代码不能继续执行。当然，如果`strict=False`，则会忽略这些细节。

```
def load_state_dict(self, state_dict, strict=True):
    missing_keys = []
```

```
        unexpected_keys = []
        error_msgs = []

        # copy state_dict so _load_from_state_dict can modify it
        metadata = getattr(state_dict, '_metadata', None)
        state_dict = state_dict.copy()
        if metadata is not None:
                state_dict._metadata = metadata

        def load(module, prefix=''):
                local_metadata = {} if metadata is None else metadata.get(prefix[:-1], {})
                module._load_from_state_dict(
                        state_dict, prefix, local_metadata, strict, missing_keys,
unexpected_keys, error_msgs)
                for name, child in module._modules.items():
                        if child is not None:
                                load(child, prefix + name + '.')

        load(self)

        if strict:
                error_msg = ''
                if len(unexpected_keys) > 0:
                        error_msgs.insert(
                                0, 'Unexpected key(s) in state_dict: {}. '.format(
                                        ', '.join('"{}"'.format(k) for k in unexpected_keys)))
                if len(missing_keys) > 0:
                        error_msgs.insert(
                                0, 'Missing key(s) in state_dict: {}. '.format(
                                        ', '.join('"{}"'.format(k) for k in missing_keys)))

        if len(error_msgs) > 0:
                raise RuntimeError('Error(s) in loading state_dict for {}:\n\t{}'.format(
                                           self.__class__.__name__, "\n\t".join(error_msgs)))
```

`2._load_from_state_dict`

```
def _load_from_state_dict(self, state_dict, prefix, local_metadata, strict,
                              missing_keys, unexpected_keys, error_msgs):
        for hook in self._load_state_dict_pre_hooks.values():
                hook(state_dict, prefix, local_metadata, strict, missing_keys,
unexpected_keys, error_msgs)

        local_name_params = itertools.chain(self._parameters.items(),
self._buffers.items())
        local_state = {k: v.data for k, v in local_name_params if v is not None}

        for name, param in local_state.items():
            key = prefix + name
            if key in state_dict:
                  input_param = state_dict[key]

                  # Backward compatibility: loading 1-dim tensor from 0.3.* to version
0.4+
                  if len(param.shape) == 0 and len(input_param.shape) == 1:
                        input_param = input_param[0]

                  if input_param.shape != param.shape:
                        # local shape should match the one in checkpoint
                        error_msgs.append('size mismatch for {}: copying a param with
shape {} from checkpoint, '
                                                      'the shape in current model is {}.'
                                                      .format(key, input_param.shape,
param.shape))
                        continue

                  if isinstance(input_param, Parameter):
                        # backwards compatibility for serialized parameters
                        input_param = input_param.data
                  try:
                        param.copy_(input_param)
                  except Exception:
                        error_msgs.append('While copying the parameter named "{}", '
                                                      'whose dimensions in the model are {} and
'
                                                      'whose dimensions in the checkpoint are
{}.'
                                                      .format(key, param.size(),
input_param.size()))
            elif strict:
                  missing_keys.append(key)

      if strict:
            for key, input_param in state_dict.items():
                  if key.startswith(prefix):
                        input_name = key[len(prefix):]
                        input_name = input_name.split('.', 1)[0]  # get the name of
param/buffer/child
                        if input_name not in self._modules and input_name not in
local_state:
                              unexpected_keys.append(key)
```