`Serquential`, `ModuleList`,和 `ModuleDict`-1

`Sequential`

简而言之：`nn.Sequential`是Moudle的子类，它的构造方法接受一系列子模型作为输入。同时也定义了`forward()`方法，这是`nn.Sequential`最大的特点，在其构造方法中传递的子类会被"链式"地计算输出。

构造方法

观察`@overload`可以看到`Sequential`的构造方法接受两种形式的输入:

```
    @overload
    def __init__(self, *args: Module) -> None:
        ...

    @overload
    def __init__(self, arg: 'OrderedDict[str, Module]') -> None:
        ...

    def __init__(self, *args):
      # 具体实现
```

虽然有两种形式的构造方法，但其本质还是调用`super.add_module`以`OrderedDict`的形式添加子模型，如果传递的参数是`Module`类型，那就以其传递编号作为`key`:

```
# 具体实现
if len(args) == 1 and isinstance(args[0], OrderedDict):
  for key, module in args[0].items():
    self.add_module(key, module)
  else:
    for idx, module in enumerate(args):
      self.add_module(str(idx), module)
```

紧接着看`add_module()`的实现，也十分简单，就是做一些列的规范化判断，然后按照`key,value`的形式把子模型注册到类属性`_module`中:

```
def add_module(self, name: str, module: Optional['Module']) -> None:
    # 一些规范化的判断，如name不能冲突，不能含有'.'之类的判断
  self._modules[name] = module
```

其他魔术方法

`__getitem__`, `__setitem__`, 和 `__delitem__`

在介绍这三个魔术方法之前，需要先介绍一个关键方法 `_get_item_by_idx(self, iterator, idx) -> T`

```
def _get_item_by_idx(self, iterator, idx) -> T:
  """Get the idx-th item of the iterator"""
  size = len(self)
  idx = operator.index(idx)
  if not -size <= idx < size:
    raise IndexError('index {} is out of range'.format(idx))
    idx %= size
    return next(islice(iterator, idx, None))
```

这个函数接收一个迭代对象和索引，返回 `T`,`T = TypeVar('T', bound=Module)`

第一行先获取迭代对象的长度，第二行把索引转换为整数, 具体见 `operator` 文档。

最后使用 `islice()` 返回索引对应的元素， `islice()` 见 `itertools` 文档

`__getitem__()` 方法返回一个 `Sequential` 或模型对象：

```
def __getitem__(self, idx) -> Union['Sequential', T]:
  if isinstance(idx, slice):
    return self.__class__(OrderedDict(list(self._modules.items())[idx]))
  else:
    return self._get_item_by_idx(self._modules.values(), idx)
```

逻辑也很简单：先判断 `idx` 是不是 `slice` 对象，至于这个 `slice` 对象，其实就是 `[start:stop:step]` 操作，如果是 `slice` 对象，那就直接用 `__class__` 返回一个 `Sequential` 实例。如果不是 `slice` 对象，那就使用 `self._get_item_by_idx()` 返回 `T`。注意 `self._modules.values()` 返回的是一个 `dict_values` 对象，而 `_get_item_by_idx` 中 `next()` 操作的是 `islice` 返回的可迭代对象。

> 个人觉得这么做而不是直接用 `list` 相关操作的原因可能是处于效率的考虑

`__setitem__()` 稍微绕一点：

```
def __setitem__(self, idx: int, module: Module) -> None:
  key: str = self._get_item_by_idx(self._modules.keys(), idx)
  return setattr(self, key, module)
```

最明显的"绕"其实是它也借助了 `_get_item_by_idx` 方法返回 `key`,从上文可知,该方法的声明返回值 `T` 和 `key` 的类型声明 `str` 是不一致的。

> Python并不强行要求变量类型和声明一致，这些声明大多都是给人理解的。这种不匹配的类型声明尽管不会影响代码的正确性，但与其做这种混乱的声明,还不如不做。

继续聊 `__setitem__` 的逻辑，实际上就是把 `idx` 的模型替换为 `module`,但由于在构造方法中注册 `module` 时不只是序号索引作为 `key`，也可能是用户指定的字符串，因此要先找到 `idx` 的位置，再拿到真正的 `key`，再进行替换操作。

`__delitem__` 和 `__setitem__` 逻辑类似。

其他魔术方法

`__dir__`和`__iter__`没什么可说的

forward

`forward`定义了`Sequential`的链式计算过程，这个也是和`ModuleList`最显著的区别

```
def forward(self, input):
  for module in self:
    input = module(input)
    return input
```

> 个人不建议使用`input`，毕竟是关键字

append

`append`就是在当前的模型列表后串联一个模型:

```
def append(self, module: Module) -> 'Sequential':
  self.add_module(str(len(self)), module)
  return self
```

不过这个函数的实现默认`key`为最新的索引顺序，可以根据需求重构为传入指定`key`