`Serquential`, `ModuleList`,和 `ModuleDict`-2

`MoudleList`

和命名一样，`ModuleList`是有着`Python List`行为的`Module`类。他和`Sequential`最主要的区别在于需要自己实现`forward`方法，因为`Python List`本身也没有先后顺序，因此`ModuleList`也不存在`Sequential`那样的链式顺序。

构造方法

`ModuleList`的构造方法很简单也很直接，就是直接使用`+=`注册`moudles`：

```
def __init__(self, modules: Optional[Iterable[Module]] = None) -> None:
  super(ModuleList, self).__init__()
  if modules is not None:
    self += modules
```

`__iadd__`和`__add__`

构造方法里涉及到`MoudleList`的自加操作`+=`，也就是实现了`__iadd__`操作：

```
def __iadd__(self, modules: Iterable[Module]) -> 'ModuleList':
  return self.extend(modules)
```

这里调用了自身的`exrtend()`方法，实现如下：

```
def extend(self, modules: Iterable[Module]) -> 'ModuleList':
  # 迭代类型判断
  #  ..
  offset = len(self)
  for i, module in enumerate(modules):
    self.add_module(str(offset + i), module)
    return self
```

逻辑很简单，也是调用`super().add_module()`注册模型，这点在`Sequential`中已经讲过，不再赘述。

接下来是对`__add__`的实现，这里用到了`itertools.chain()`返回`self`和需要被加对象的迭代器，然后返回一个新的`ModuleList()`，`chain()`的说明见：`itertools`

```
def __add__(self, other: Iterable[Module]) -> 'ModuleList':
  combined = ModuleList()
  for i, module in enumerate(chain(self, other)):
    combined.add_module(str(i), module)
    return combined
```

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

和`Sequential`一样，这三个魔术方法的实现也依赖一个处理索引的函数`_get_abs_string_index`,其主要功能就是把用户传入的`idx:int in [-len(self), len(self)]`转换为`idx:str`：

```
def _get_abs_string_index(self, idx):
  """Get the absolute index for the list of modules"""
  idx = operator.index(idx)
  if not (-len(self) <= idx < len(self)):
    raise IndexError('index {} is out of range'.format(idx))
    if idx < 0:
      idx += len(self)
      return str(idx)
```

`__getitem__`, `__setitem__`的逻辑和`Sequential`一样：

```
def __getitem__(self, idx: int) -> Union[Module, 'ModuleList']:
  if isinstance(idx, slice):
    return self.__class__(list(self._modules.values())[idx])
  else:
    return self._modules[self._get_abs_string_index(idx)]

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

`__delitem__`就需要做一个额外操作：重新组织模型以保证子模型的编号正确

```
def __delitem__(self, idx: Union[int, slice]) -> None:
  if isinstance(idx, slice):
    for k in range(len(self._modules))[idx]:
      delattr(self, str(k))
    else:
      delattr(self, self._get_abs_string_index(idx))
      str_indices = [str(i) for i in range(len(self._modules))]
      self._modules = OrderedDict(list(zip(str_indices, self._modules.values())))
```

> 在读到这一段时有一个疑问：为什么`Sequential`和`ModuleList`都是维护字典形式的`_modules`，但一个需要重新组织序号，一个不需要？ 原因在于`Sequential`的逻辑都是以`key-value`进行的，而`ModuleList`的组织方式是`idx:int -> idx:str`。其实`ModuleList`也可以用字典的形式组织，但索引的方式更符合`list`的行为。
>
> 其实在实际生产中，我们可以结合`ModuleList`，`Sequential`和后文的`ModuleDict`组织一种更“动态”的模型，暂且叫它`ModuleChain`。这个后续有空会尝试一下。

其他魔术方法

`__dir__`没什么可说的:

```
def __dir__(self):
  keys = super(ModuleList, self).__dir__()
  keys = [key for key in keys if not key.isdigit()]
  return keys
```

列表行为方法

作为一个具有列表行为的模型，最主要也是最常用的方法`append`和`insert`也很直观，就是在尾部插入和在中间移位插入:

```
def insert(self, index: int, module: Module) -> None:
  for i in range(len(self._modules), index, -1):
    self._modules[str(i)] = self._modules[str(i - 1)]
    self._modules[str(index)] = module
def append(self, module: Module) -> 'ModuleList':
  self.add_module(str(len(self)), module)
  return self
```