mmdetection2.6 框架源码流程梳理

> 最近毕业设计课题想要基于这个框架来做，最近这段时间要研究一下mmdetection的使用方式以及源码组织方式，学会自己扩展模型以及一些特定的函数， 这里记录学习笔记，网上也没有合适的资料，希望也能帮助初学者，大家一起学习。

> 完整笔记获取地址: https://github.com/lxztju/notes

正文分割线

(一) mmdetection总体结构

mmdetection依赖于mmcv.

1.1 mmdetection的整体代码结构:

configs: 网络组件结构的配置信息,在mmdetection2.0中采用继承结构.

tools: 训练与测试的最终包装以及一些非常实用的程序脚本文件

mmdet:

    apis: 推理训练测试的基础代码,这里就存在三个程序脚本文件train.py, test.py, inference.py.

    core: anchor 生成,bbox,mask 编解码, 变换, 标签锚定, 采样等, 模型

                评估, 加速, 优化器，后处理等

    datasets: coco,voc 等数据类, 数据 pipelines 的统一格式, 数据增强，数

                        据采样

    models: 模型组件(backbone, dense_heads, roi_heads, necks, detectors, losses) ops: 从mmcv的ops中导入,包括nms, roi_align, roi_pooling等操作.

1.2 训练过程逻辑:

训练入口从tools/train.py进入, 从其中代码可以看到整体可分为如下的几个步骤:

1. `mmcv.utils.config.Config.fromfile`从配置文件来解析配置信息
2. **mmdet中的builder.py**来构建模型数据类以及对应的训练train_pipeline.

   `builder.py`依据**mmcv.utils中的build_from_cfg**定义了`build_backbone,build_detector, build_head, build_loss, build_neck,build_roi_extractor, build_shared_head`的函数
3. 在**mmdet.models**中利用上边定义的`build_detector`来一句config文件的配置信息来构建模型.

   3.1. build 系列函数调用 build_from_cfg 函数, 按 type 关键字从注册表中获取相应的对象, 对象的具名参数在注册文件中赋值
   3.2. 然后依据`mmcv.utils.registry.py`中的模型组件注册器.其中注册器的 register_module成员函数是一个装饰器功能函数，在具体的类对象 A 头上装饰 @X.register_module并同时在 A 对象所在包的初始化文件中调用 A，即可将 A 保存到`registry.module_dic`t 中, 完成注册.
   3.3. 在`mmdet.builder.py`中定义了`BACKBONES, DETECTORS, HEADS, LOSSES, NECKS,ROI_EXTRACTORS, SHARED_HEADS`的模型注册器.
4. 在**mmdet.datasets中builder.py**中定义了DATASETS, PIPELINES, build_dataloader, build_dataset
   4.1. 然后依据对应的config文件构建数据集以及pipline

5. 调用**mmdet.apis中的train_detector**函数进行模型训练.
   5.1 这里采用了mmcv中的hook方式进入runner的训练流程.

这里需要研究一下register, runner与hook的工作方式以及在register中采用的python装饰器的功能作用.

- **Runner** 是整个训练过程的流程控制
- **HOOK** 定义在每个迭代周期(epoch)或者每个迭代步骤(iter)之前之后的一些操作
- **Register** 完成了mmdetection模块化设计(**为模块化服务的字符串->类的字典**)

这部分内容主要来自知乎大佬文章,大佬知乎主页:https://www.zhihu.com/people/mo-dao-90/columns

1.3 HOOK(钩子)

hook机制定义在`mmcv.runner.hooks`中.

hook的基类定义在 `mmcv.runner.hooks.hook.py`中.

```
# Copyright (c) Open-MMLab. All rights reserved.
from mmcv.utils import Registry

HOOKS = Registry('hook')


class Hook:

    def before_run(self, runner):
        pass

    def after_run(self, runner):
        pass

    def before_epoch(self, runner):
        pass

    def after_epoch(self, runner):
        pass字符串到

    def before_iter(self, runner):
        pass

    def after_iter(self, runner):
        pass

    def before_train_epoch(self, runner):
        self.before_epoch(runner)

    def before_val_epoch(self, runner):
        self.before_epoch(runner)

    def after_train_epoch(self, runner):
        self.after_epoch(runner)

    def after_val_epoch(self, runner):
        self.after_epoch(runner)

    def before_train_iter(self, runner):
        self.before_iter(runner)

    def before_val_iter(self, runner):
        self.before_iter(runner)

    def after_train_iter(self, runner):
        self.after_iter(runner)

    def after_val_iter(self, runner):
        self.after_iter(runner)
    def every_n_epochs(self, runner, n):
        return (runner.epoch + 1) % n == 0 if n > 0 else False

    def every_n_inner_iters(self, runner, n):
        return (runner.inner_iter + 1) % n == 0 if n > 0 else False

    def every_n_iters(self, runner, n):
        return (runner.iter + 1) % n == 0 if n > 0 else False

    def end_of_epoch(self, runner):
        return runner.inner_iter + 1 == len(runner.data_loader)
```

这个基类函数定义了在模型训练过程中需要用到的功能,**传入的参数均为runner**. Runner是一个模型训练的工厂，在其中我们可以加载数据、训练、验证以及梯度backward等等全套流程. MMdetection在设计的时候也为runner传入丰富的参数，定义了一个非常好的训练范式。在你的每一个hook函数中，都可以对runner进行你想要的操作。

而HOOK是怎么嵌套进runner中的呢？其实是在Runner中定义了一个hook的list，**list中的每一个元素就是一个实例化的HOOK对象**。其中提供了**两种注册hook的方法**，`register_hook`**是传入一个实例化的HOOK对象**，并将它插入到一个列表中，`register_hook_from_cfg`**是传入一个配置项**，根据配置项来实例化HOOK对象并插入到列表中。当然第二种方法又是MMLab的开源生态中定义的一种基础方法`mmcv.build_from_cfg`(存在于`mmcv.utils.register.py`中)了，无论在MMdetection还是其他MMLab开源的算法框架中，都遵循着MMCV的这套基于配置项实例化对象的方法。毕竟MMCV是提供了一个基础的功能，服务于各个算法框架，这也是为什么MMLab的代码高质量的原因。不仅仅是算法的复现，更是架构、编程范式的一种体现，*真·代码如诗*。

下边的代码存在于`mmcv.runner.base_runner.py`中.

```
# 将一个hook对象插入到hook_list中.
def register_hook(self, hook, priority='NORMAL'):
    """Register a hook into the hook list.

    hook将会插入优先级队列中, 对于同样优先级的hook, 将会按照注册
的顺序进行触发.

    The hook will be inserted into a priority queue, with the
specified
    priority (See :class:`Priority` for details of priorities).
    For hooks with the same priority, they will be triggered in the
same
    order as they are registered.

    Args:
      hook (:obj:`Hook`): The hook to be registered.
      priority (int or str or :obj:`Priority`): Hook priority.
        Lower value means higher priority.
    """
    assert isinstance(hook, Hook)
    if hasattr(hook, 'priority'):
      raise ValueError('"priority" is a reserved attribute for hooks')
    # 得到hook对应的优先级.
    priority = get_priority(priority)
    hook.priority = priority
    # 按照制定额的优先级插入list中.
    # insert the hook to a sorted list
    inserted = False
    for i in range(len(self._hooks) - 1, -1, -1):
      if priority >= self._hooks[i].priority:
        self._hooks.insert(i + 1, hook)
        inserted = True
        break
    if not inserted:
      self._hooks.insert(0, hook)

  # 利用cfg配置项来讲hook对象插入hook_list中.
  def register_hook_from_cfg(self, hook_cfg):
    """Register a hook from its cfg.

    Args:
      hook_cfg (dict): Hook config. It should have at least keys 'type'
       and 'priority' indicating its type and priority.

    Notes:
      The specific hook class to register should not use 'type' and
      'priority' arguments during initialization.
    """
    hook_cfg = hook_cfg.copy()
    priority = hook_cfg.pop('priority', 'NORMAL')
    hook = mmcv.build_from_cfg(hook_cfg, HOOKS)
    self.register_hook(hook, priority=priority)
```

调用hook函数

```
  def call_hook(self, fn_name):
    """Call all hooks.

    Args:
      fn_name (str): The function name in each hook to be called,
such as
        "before_train_epoch".
    """
    for hook in self._hooks:
      getattr(hook, fn_name)(self)
```

可以看到HOOK是调用的时候是**遍历hook_List**，然后根据HOOK的名字来调用。这也是为什么要区分优先级的原因，优先级越高的放在List的前面，这样就能更快地被调用。当你想用*before_run_epoch*来做A和B两件事情的时候，在runner里面就是调用一次`self.before_run_epoch`，但是先做A还是先做B，就是通过不同的HOOK的优先级来决定了。比如在evaluation的时候对需要做测

试，但是测试前对参数做滑动平均。比如emaHOOK中的72行，也写明了要在测试之前做指数滑动平均。

```
  def after_train_epoch(self, runner):
      """We load parameter values from ema backup to model before the
      EvalHook."""
      self._swap_ema_parameters()
```

同样在`checkpoint.py`中也定义了`after_train_epoch`这个函数.

```
@master_only
  def after_train_epoch(self, runner):
      if not self.by_epoch or not self.every_n_epochs(runner, self.interval):
          return

      runner.logger.info(f'Saving checkpoint at {runner.epoch + 1} epochs')
      if not self.out_dir:
          self.out_dir = runner.work_dir
      runner.save_checkpoint(
          self.out_dir, save_optimizer=self.save_optimizer, **self.args)

      # remove other checkpoints
      if self.max_keep_ckpts > 0:
          filename_tmpl = self.args.get('filename_tmpl', 'epoch_{}.pth')
          current_epoch = runner.epoch + 1
          for epoch in range(current_epoch - self.max_keep_ckpts, 0, -1):
              ckpt_path = os.path.join(self.out_dir,
                                       filename_tmpl.format(epoch))
              if os.path.exists(ckpt_path):
                  os.remove(ckpt_path)
              else:
                  break
```

从测试代码中可以看到不同的HOOK虽然都是重写了`after_train_epoch`函数，但是调用的顺序还是先调用`ema.py`中的，然后再调用`checkpoint.py`中的`after_train_epoch`。

```
resume_ema_hook = EMAHook(
      momentum=0.5, warm_up=0,
resume_from=f'{work_dir}/epoch_1.pth')
   runner = _build_demo_runner()
   runner.model = demo_model
   # 设置了HIGHREST的优先级
   runner.register_hook(resume_ema_hook, priority='HIGHEST')
   checkpointhook = CheckpointHook(interval=1, by_epoch=True)
   runner.register_hook(checkpointhook)
   runner.run([loader, loader], [('train', 1), ('val', 1)], 2)
```

在`mmcv.runner.priority.py`中一共定义了**7种优先级**.

```
class Priority(Enum):
   """Hook priority levels.
   +------------+------------+
   | Level      | Value      |
   +============+============+
   | HIGHEST    | 0          |
   +------------+------------+
   | VERY_HIGH  | 10         |
   +------------+------------+
   | HIGH       | 30         |
   +------------+------------+
   | NORMAL     | 50         |
   +------------+------------+
   | LOW        | 70         |
   +------------+------------+
   | VERY_LOW   | 90         |
   +------------+------------+
   | LOWEST     | 100        |
   +------------+------------+
   """

   HIGHEST = 0
   VERY_HIGH = 10
   HIGH = 30
   NORMAL = 50
   LOW = 70
   VERY_LOW = 90
   LOWEST = 100


def get_priority(priority):
   """Get priority value.
   Args:
       priority (int or str or :obj:`Priority`): Priority.
   Returns:
       int: The priority value.
   """
   if isinstance(priority, int):
       if priority < 0 or priority > 100:
           raise ValueError('priority must be between 0 and 100')
       return priority
   elif isinstance(priority, Priority):
       return priority.value
   elif isinstance(priority, str):
       return Priority[priority.upper()].value
   else:
       raise TypeError('priority must be an integer or Priority enum
value')
```

1.4 Register

在MMDection中所有功能都是基于注册器来完成模块化操作的. 其中最经典的就是在MMdetection构件模型的builder.py(代码在`mmdetection.mmdet.builder.py`)中就通过注册器完成模型的模块化。

首先，**要明确注册器的使用目的就是为了在算法训练、调参中通过直接更改配置文件**，你想从Faster RCNN切换到Retina，或者是更改学习率等，只需要更改配置即可。因为**注册器完成了字符串->类的映射**，代码会自动解析你config中的内容。下面这个就是在builder.py中的实例化的注册器。

```
from mmcv.utils import Registry, build_from_cfg
from torch import nn

BACKBONES = Registry('backbone')
NECKS = Registry('neck')
ROI_EXTRACTORS = Registry('roi_extractor')
SHARED_HEADS = Registry('shared_head')
HEADS = Registry('head')
LOSSES = Registry('loss')
DETECTORS = Registry('detector')
```

其中，模型被拆解成了Backbones、Necks、Roi_extractors、Shared_Heads、Heads、Losses、Detectors几个部分，当时如果我们需要添加更多的模块也是可以的，只需要实例化我们各自的注册器就行。

从代码中也可以看到**注册器只是通过一个类完成了string类型->类名的一个映射**而已。在`mmcv.utils.registry.py`中，注册类的源码如下。

```
def _register_module(self, module_class, module_name=None, force=False):
  # 首先判断判断参数是否是Class类别
    if not inspect.isclass(module_class):
      raise TypeError('module must be a class, '
              f'but got {type(module_class)}')

    if module_name is None:
     # 获取类名
      module_name = module_class.__name__
    if not force and module_name in self._module_dict:
      raise KeyError(f'{module_name} is already registered '
              f'in {self.name}')
    # 核心就这一句话，是不是超级简单？就是一个dict，key值是string, value是类
    self._module_dict[module_name] = module_class
```

咋一看将string和Class之间的映射链接起来也不过如此啊，不过就是一个dict罢了，python初学者都能理解的东西。但是高就高在大道至简，看似平凡普通的东西，在大师的手里就是一个利器。

首先，**注册类的操作，在MMDetection中使用了装饰器**解决了。不懂装饰器的可以先行百度，简单来说。装饰器的功能就是有A和B两个事情要做。但是做B事情的时候你希望能用到A的一些功能来达成你的需求。在`mmcv.utils.registry.py`中，有一个用于装饰器函数来实现这个功能，并且注释也十分浅显易懂。

```
def register_module(self, name=None, force=False, module=None):
    """Register a module.
```

```
        A record will be added to `self._module_dict`, whose key is the
class
        name or the specified name, and value is the class itself.
        It can be used as a decorator or a normal function.

        Example:
        # 这里写了三种将类注册到backbones注册器中的方法。


            >>> backbones = Registry('backbone')
            >>> @backbones.register_module()
            >>> class ResNet:
            >>>     pass

                # 一个小扩展，如果你不想每个string的名字都是和类名完全一
样，你也可以自定义你记得住的名字，但是这样的话你在config中写你
的Backbone名字就是'mnet'了
            >>> backbones = Registry('backbone')
            >>> @backbones.register_module(name='mnet')
            >>> class MobileNet:
            >>>     pass



                # 这是不用装饰器的时候，我们的正常操作，当然这显然没够上
python优雅简洁的特点。
            >>> backbones = Registry('backbone')
            >>> class ResNet:
            >>>     pass
            >>> backbones.register_module(ResNet)

        Args:
            name (str | None): The module name to be registered. If not
                specified, the class name will be used.
            force (bool, optional): Whether to override an existing class
with
                the same name. Default: False.
            module (type): Module class to be registered.
        """
        if not isinstance(force, bool):
            raise TypeError(f'force must be a boolean, but got 
{type(force)}')
        # NOTE: This is a walkaround to be compatible with the old api,
        # while it may introduce unexpected bugs.
        if isinstance(name, type):
            return self.deprecated_register_module(name, force=force)

        # use it as a normal method: 
x.register_module(module=SomeClass)
        if module is not None:
            self._register_module(
                module_class=module, module_name=name, force=force)
            return module

        # raise the error ahead of time
        if not (name is None or isinstance(name, str)):
            raise TypeError(f'name must be a str, but got {type(name)}')
                # 以下方式是用的最多的一种
        # use it as a decorator: @x.register_module()
        def _register(cls):
            self._register_module(
                module_class=cls, module_name=name, force=force)
            return cls

        return _register
```

理解了注册器功能、实现代码和使用方式之后，好像对这玩意已经掌握了。但是有一点需要说明的是，当你自己使用了一个注册器用装饰器添加了之后就可以直接在外部调用了吗？这里有一个小细节，**如果你写了一个类并通过注册了，那么需要导入过这个类才会生效**，具体来说就是在某个地方`import`这个类之后才会有用。

 当然注册器只是完成一个模块化功能，具体怎么把这些模块化的东西组织起来其实就是一些逻辑控制了。需要注意的是在`mmdetection.mmdet.builder.py`中的第30行`torch.nn.Sequential`的操作并不是把backbone-neck-head组装起来的地方，如下所示，这只是把一些更细粒度的模块组成一起而已。比如`configs/_base_/models/cascade_mask_rcnn_r50_fpn.py`的第44行。

```
def build(cfg, registry, default_args=None):
    """Build a module.

    Args:
        cfg (dict, list[dict]): The config of modules, is is either a dict
            or a list of configs.
        registry (:obj:`Registry`): A registry the module belongs to.
        default_args (dict, optional): Default arguments to build the
module.
            Defaults to None.

    Returns:
        nn.Module: A built nn module.
    """
    if isinstance(cfg, list):
        modules = [
            build_from_cfg(cfg_, registry, default_args) for cfg_ in cfg
        ]
        # 注意这只是把一些细节的模块拼在一起
        return nn.Sequential(*modules)
    else:
        return build_from_cfg(cfg, registry, default_args)
```

其实从设计原则上来说，backbone neck head loss等等应该是要在代码中手动操作的。比如在 `mmdetection.mmdet.models.detectors.two_stage.py` 中(faster_rcnn继承这个基类)，**forward的流程**是这样的。

```
def forward_train(self,
                  img,
                  img_metas,
                  gt_bboxes,
                  gt_labels,
                  gt_bboxes_ignore=None,
                  gt_masks=None,
                  proposals=None,
                  **kwargs):
    """
    Args:
        img (Tensor): of shape (N, C, H, W) encoding input images.
            Typically these should be mean centered and std scaled.
        img_metas (list[dict]): list of image info dict where each dict
            has: 'img_shape', 'scale_factor', 'flip', and may also contain
            'filename', 'ori_shape', 'pad_shape', and 'img_norm_cfg'.
            For details on the values of these keys see
            `mmdet/datasets/pipelines/formatting.py:Collect`.
        gt_bboxes (list[Tensor]): Ground truth bboxes for each image
with
            shape (num_gts, 4) in [tl_x, tl_y, br_x, br_y] format.
        gt_labels (list[Tensor]): class indices corresponding to each
box
        gt_bboxes_ignore (None | list[Tensor]): specify which
bounding
            boxes can be ignored when computing the loss.
        gt_masks (None | Tensor) : true segmentation masks for each
box
            used if the architecture supports a segmentation task.
        proposals : override rpn proposals with custom proposals. Use
when
```

```
            `with_rpn` is False.
        Returns:
            dict[str, Tensor]: a dictionary of loss components
        """
        # 利用backbone+neck的输出feature map
        x = self.extract_feat(img)

        losses = dict()

        # RPN forward and loss
        if self.with_rpn:
            proposal_cfg = self.train_cfg.get('rpn_proposal',
                                              self.test_cfg.rpn)
            rpn_losses, proposal_list = self.rpn_head.forward_train(
                x,
                img_metas,
                gt_bboxes,
                gt_labels=None,
                gt_bboxes_ignore=gt_bboxes_ignore,
                proposal_cfg=proposal_cfg)
            losses.update(rpn_losses)
        else:
            proposal_list = proposals
        # roi的前向传播与loss
        roi_losses = self.roi_head.forward_train(x, img_metas,
proposal_list,
                                                 gt_bboxes, gt_labels,
                                                 gt_bboxes_ignore, gt_masks,
                                                 **kwargs)
        losses.update(roi_losses)

        return losses
```

```
def extract_feat(self, img):
        """Directly extract features from the backbone+neck."""
        x = self.backbone(img)
        if self.with_neck:
            x = self.neck(x)
        return x
```

总而言之，注册器只是提供从配置文件生成实例对象的一种方式，最重要的还是在各个`detector`中对这些模块的调用，以及深度学习算法训练、测试、inference整个流程的处理。

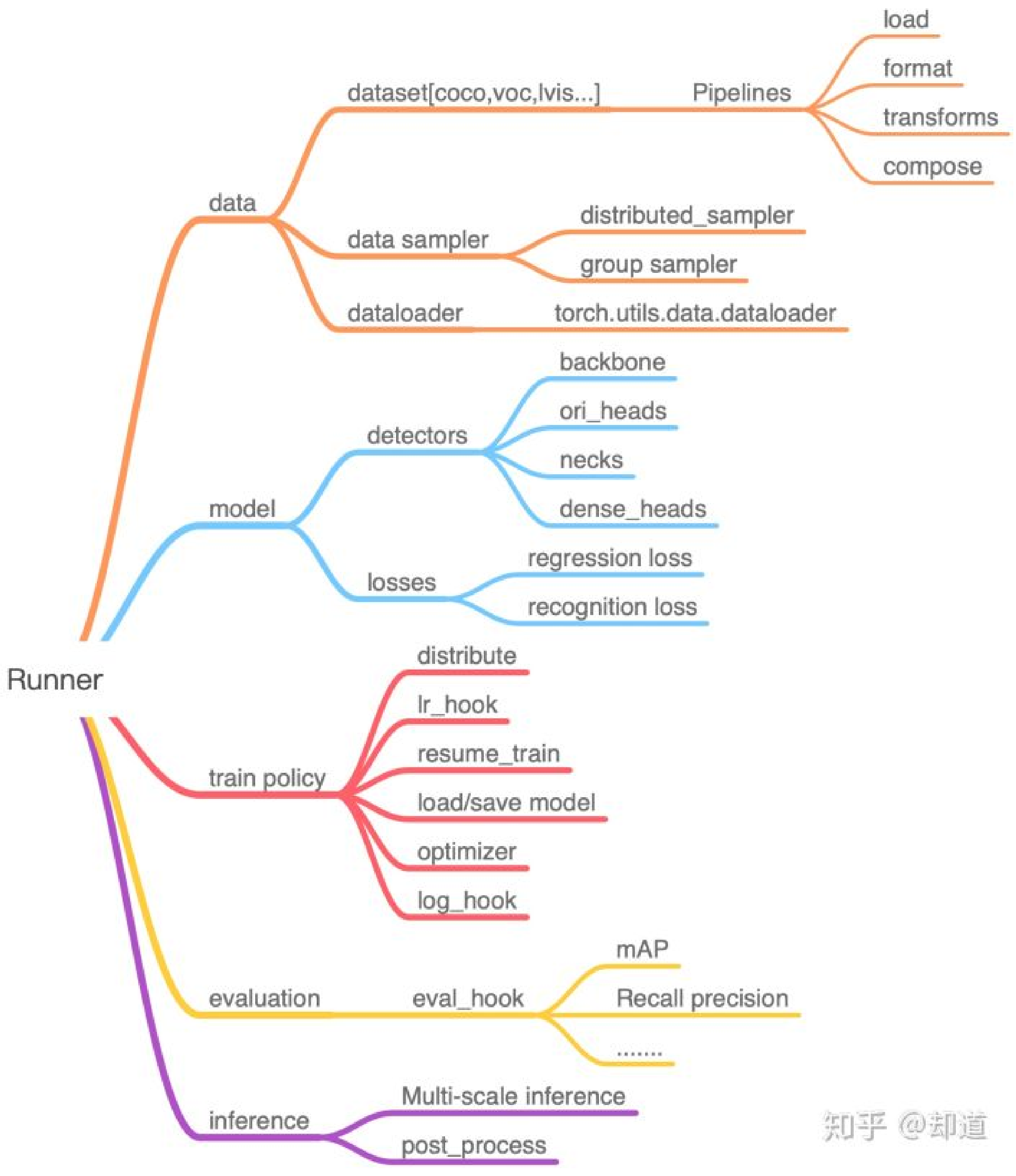
总结: 为模块化服务的字符串->类的字典

1.5 Runner

Runner是MMdetection中的一种深度学习算法“工厂”，是对深度学习算法各个组件的“容器”。简单来说，所有的**机器学习算法所包含的无非就是数据、模型、训练策略、评估、推理这五个部分**。Runner就是将这五个部分组合在一起的工具。

其实光是Runner，可以说的东西不多，但是其背后的设计思路，以及对深度学习算法的概括是值的学习的。所以这篇文章不仅是提炼一下Runner中的操作，也是对一个算法“**数据、模型、训练策略、评估、推理**”这五个部分的一种总结。

Runner的整体结构:



Runner的源码封装在MMCV库当中，目前主要是有**epoch_runner**(`mmcv.runner.epoch_based_runner.py`)和**iter_runner**(`mmcv.runner.iter_based_runner.py`)两种，本质上差不多，只是大家的习惯不同罢了，其实在`mmcv.runner.epoch_based_runner.py`中也有内置计算了iter数。

常用的是epoch_runner，那么就以epoch_runner为例理一理各个模块的使用。**由于代码过多，所以有些类型判断、warning，参数注释我都删去了，只要理清楚整个代码运行逻辑就行。**

```
@RUNNERS.register_module()
class EpochBasedRunner(BaseRunner):
    def run_iter(self, data_batch, train_mode, **kwargs):
        if train_mode:
            outputs = self.model.train_step(data_batch,
self.optimizer,**kwargs)
        else:
            outputs = self.model.val_step(data_batch, self.optimizer,
**kwargs)
        if not isinstance(outputs, dict):
            raise TypeError('"batch_processor()" or "model.train_step()"'
                            'and "model.val_step()" must return a dict')
        if 'log_vars' in outputs:
            self.log_buffer.update(outputs['log_vars'],
outputs['num_samples'])
        self.outputs = outputs

    def train(self, data_loader, **kwargs):
        self.model.train()
        self.mode = 'train'
        self.data_loader = data_loader
        self._max_iters = self._max_epochs * len(self.data_loader)
        self.call_hook('before_train_epoch')
        time.sleep(2)  # Prevent possible deadlock during epoch
transition
        for i, data_batch in enumerate(self.data_loader):
            self._inner_iter = i
            self.call_hook('before_train_iter')
            self.run_iter(data_batch, train_mode=True)
            self.call_hook('after_train_iter')
            self._iter += 1
        self.call_hook('after_train_epoch')
        self._epoch += 1

    def val(self, data_loader, **kwargs):
        self.model.eval()
        self.mode = 'val'
        self.data_loader = data_loader
        # 以下几个call_hook就是几个典型的应用，HOOK的用法可以详细
看下这个专栏里对于HOOK机制剖析的文章
        self.call_hook('before_val_epoch')
        time.sleep(2)  # Prevent possible deadlock during epoch
transition
        for i, data_batch in enumerate(self.data_loader):
            self._inner_iter = i
            self.call_hook('before_val_iter')
            with torch.no_grad():
                self.run_iter(data_batch, train_mode=False)
            self.call_hook('after_val_iter')

        self.call_hook('after_val_epoch')

    def run(self, data_loaders, workflow, max_epochs=None,
**kwargs):
        """Start running.
        Args:
            data_loaders (list[:obj:`DataLoader`]): Dataloaders for
training
                and validation.
            workflow (list[tuple]): A list of (phase, epochs) to specify the
                running order and epochs. E.g, [('train', 2), ('val', 1)] means
                running 2 epochs for training and 1 epoch for validation,
                iteratively.
        """
        for i, flow in enumerate(workflow):
            mode, epochs = flow
            if mode == 'train':
              # epoch runner内部还是
```

```python
                self._max_iters = self._max_epochs * len(data_loaders[i])
                break

        work_dir = self.work_dir if self.work_dir is not None else 'NONE'
        self.call_hook('before_run')

        while self.epoch < self._max_epochs:
            for i, flow in enumerate(workflow):
                mode, epochs = flow
                # 训练则mode='train'
                # 评估则mode='val'
                epoch_runner = getattr(self, mode)
                for _ in range(epochs):
                    if mode == 'train' and self.epoch >= self._max_epochs:
                        break
                    epoch_runner(data_loaders[i], **kwargs)

        time.sleep(1)  # wait for some hooks like loggers to finish
        self.call_hook('after_run')

    def save_checkpoint(self,
                        out_dir,
                        filename_tmpl='epoch_{}.pth',
                        save_optimizer=True,
                        meta=None,
                        create_symlink=True):
        # 保存模型的相关代码
        if meta is None:
            meta = dict(epoch=self.epoch + 1, iter=self.iter)
        elif isinstance(meta, dict):
            meta.update(epoch=self.epoch + 1, iter=self.iter)
        else:
            raise TypeError(
                f'meta should be a dict or None, but got {type(meta)}')
        if self.meta is not None:
            meta.update(self.meta)
    filename = filename_tmpl.format(self.epoch + 1)
    filepath = osp.join(out_dir, filename)
    optimizer = self.optimizer if save_optimizer else None
    save_checkpoint(self.model, filepath, optimizer=optimizer,
meta=meta)
    # in some environments, `os.symlink` is not supported, you
may need to
    # set `create_symlink` to False
    if create_symlink:
      dst_file = osp.join(out_dir, 'latest.pth')
      if platform.system() != 'Windows':
        mmcv.symlink(filename, dst_file)
      else:
        shutil.copy(filename, dst_file)
```

- **数据**：对于数据的处理，主要看`mmdetection.mmdet.datasets`中的代码，数据处理这块主要是逻辑上要严谨，代码上要细致，功能上要全面。比如对开源的不同数据集的处理方式，输出格式的统一，在做transform的时候要考虑一些边界问题，多数据集训练多卡训练。不同sampler的方式，对训练IO通信的优化（预加载数据到内存），样本均衡。
- **模型**：算法框架被分为SingleStage和TwoStage两种，当然每一种都可以细分为AnchorFree或者是AnchorBased。算法这块内容有点多，但是主要还是LOSS和网络结构两个部分的创新。
- **训练策略**：这部分也算是框架内容上，比如warm_up，lr的下降，早停，打印log(支持终端输出和tensorboard)，可配置optimizer，保存模型、加载预训练模型、继续训练模型等。
- **评估策略**：学术集的话一般是有提供评估代码，比如COCO和cityscapes，我们要做的只是把输出的格式处理成相应的格式就行。但是实际工程当中的话评估策略需要根据业务要求来定。
- **推理**：目标检测inference最常用的是多尺度预测，当然还有一些细节，比如检测框和网络predict之间是有相应的变换的，推理的时候需要反变换为原图尺度。其实推理这部分和评估很多地

方是重复的，只是在代码结构上他们是分开的，而且在有些任务中推理和评估还有一些不同之处，因此才把他们区分开来。

(二) mmdetection2.6的整体流程分析

2.1 入口函数

直接从训练入口处入手`tools/train.py`, 伴随着上文中介绍的HOOK, Register, Runner机制来理解代码.

```
# 这是main函数.
def main():
    args = parse_args()

    # mmcv.Config.fromfile 从配置文件解析配置信息, 并做适当更新, 包括环
    # 境搜集，预加载模型文件, 分布式设置，日志记录等
    cfg = Config.fromfile(args.config)
    if args.cfg_options is not None:
        cfg.merge_from_dict(args.cfg_options)
    # import modules from string list.
    if cfg.get('custom_imports', None):
        from mmcv.utils import import_modules_from_strings
        import_modules_from_strings(**cfg['custom_imports'])
    # set cudnn_benchmark
    if cfg.get('cudnn_benchmark', False):
        torch.backends.cudnn.benchmark = True

    # work_dir is determined in this priority: CLI > segment in file >
filename
    if args.work_dir is not None:
        # update configs according to CLI args if args.work_dir is not
None
        cfg.work_dir = args.work_dir
    elif cfg.get('work_dir', None) is None:
        # use config filename as default work_dir if cfg.work_dir is None
        cfg.work_dir = osp.join('./work_dirs',
                                osp.splitext(osp.basename(args.config))[0])
    if args.resume_from is not None:
        cfg.resume_from = args.resume_from
    if args.gpu_ids is not None:
        cfg.gpu_ids = args.gpu_ids
    else:
        cfg.gpu_ids = range(1) if args.gpus is None else range(args.gpus)

    # init distributed env first, since logger depends on the dist info.
    if args.launcher == 'none':
        distributed = False
    else:
        distributed = True
        init_dist(args.launcher, **cfg.dist_params)

    # create work_dir
    mmcv.mkdir_or_exist(osp.abspath(cfg.work_dir))
    # dump config
    cfg.dump(osp.join(cfg.work_dir, osp.basename(args.config)))
    # init the logger before other steps
    timestamp = time.strftime('%Y%m%d_%H%M%S',
time.localtime())
    log_file = osp.join(cfg.work_dir, f'{timestamp}.log')
    logger = get_root_logger(log_file=log_file, log_level=cfg.log_level)

    # init the meta dict to record some important information such as
    # environment info and seed, which will be logged
    meta = dict()
    # log env info
    env_info_dict = collect_env()
    env_info = '\n'.join([(f'{k}: {v}') for k, v in env_info_dict.items()])
    dash_line = '-' * 60 + '\n'
    logger.info('Environment info:\n' + dash_line + env_info + '\n' +
                dash_line)
    meta['env_info'] = env_info
    meta['config'] = cfg.pretty_text
    # log some basic info
    logger.info(f'Distributed training: {distributed}')
    logger.info(f'Config:\n{cfg.pretty_text}')

    # set random seeds
    if args.seed is not None:
        logger.info(f'Set random seed to {args.seed}, '
                    f'deterministic: {args.deterministic}')
        set_random_seed(args.seed, deterministic=args.deterministic)
    cfg.seed = args.seed
    meta['seed'] = args.seed
    meta['exp_name'] = osp.basename(args.config)

######################################################################
    ## 构建检测器mmdet/builder.py
    # 这里依据config文件中的model这个字典,来构建模型,返回一个类对
象
    model = build_detector(
        cfg.model, train_cfg=cfg.train_cfg, test_cfg=cfg.test_cfg)
```

```
  # 构建数据集mmdet/builder.py
  # 依据给定的config文件中的data参数
  datasets = [build_dataset(cfg.data.train)]

  ## 在configs/__base__/default_runtime.py中存在的参数,是否需要
添加验证集.
  if len(cfg.workflow) == 2:
    val_dataset = copy.deepcopy(cfg.data.val)
    val_dataset.pipeline = cfg.data.train.pipeline
    datasets.append(build_dataset(val_dataset))
  if cfg.checkpoint_config is not None:
    # save mmdet version, config file content and class names in
    # checkpoints as meta data
    cfg.checkpoint_config.meta = dict(
      mmdet_version=__version__ + get_git_hash()[:7],
      CLASSES=datasets[0].CLASSES)
  # add an attribute for visualization convenience
  model.CLASSES = datasets[0].CLASSES

  ## 训练模型,函数存在mmdet/apis/train.py中
  train_detector(
    model,
    datasets,
    cfg,
    distributed=distributed,
    validate=(not args.no_validate),
    timestamp=timestamp,
    meta=meta)
```

2.2 模型构建 `mmdet/builder.py`

```
# 在tools/train.py中的模型构建过程.
# model = build_detector(
#     cfg.model, train_cfg=cfg.train_cfg, test_cfg=cfg.test_cfg)
```

```
def build_detector(cfg, train_cfg=None, test_cfg=None):
    """Build detector."""
    return build(cfg, DETECTORS, dict(train_cfg=train_cfg,
test_cfg=test_cfg))

def build(cfg, registry, default_args=None):
    """Build a module.

    Args:
        cfg (dict, list[dict]): The config of modules, is is either a dict
            or a list of configs.
        registry (:obj:`Registry`): A registry the module belongs to.
        default_args (dict, optional): Default arguments to build the
module.
            Defaults to None.

    Returns:
        nn.Module: A built nn module.
    """
    if isinstance(cfg, list):
        modules = [
            build_from_cfg(cfg_, registry, default_args) for cfg_ in cfg
        ]
        return nn.Sequential(*modules)
    else:
        return build_from_cfg(cfg, registry, default_args)


# 代码来自: mmcv/utils/registry.py
#
# def build_from_cfg(cfg, registry, default_args=None):
#     """Build a module from config dict.
#
#     Args:
#         cfg (dict): Config dict. It should at least contain the key "type".
config文件中至少要包含"type"这个键
#         registry (:obj:`Registry`): The registry to search the type from.
#       default_args (dict, optional): Default initialization arguments.
#
#     Returns:
#       object: The constructed object.
#     """
#    config必须是dict格式
#     if not isinstance(cfg, dict):
#       raise TypeError(f'cfg must be a dict, but got {type(cfg)}')
#     if 'type' not in cfg:
#       if default_args is None or 'type' not in default_args:
#           raise KeyError(
#               '`cfg` or `default_args` must contain the key "type", '
#               f'but got {cfg}\n{default_args}')
#     if not isinstance(registry, Registry):
#       raise TypeError('registry must be an mmcv.Registry object, '
#                   f'but got {type(registry)}')
#     if not (isinstance(default_args, dict) or default_args is None):
#       raise TypeError('default_args must be a dict or None, '
#                   f'but got {type(default_args)}')
#
#     args = cfg.copy()
#
#     if default_args is not None:
#       for name, value in default_args.items():
#       如果键不存在于字典中，将会添加键并将值设为默认值。
#           args.setdefault(name, value)
#
#     obj_type = args.pop('type')
#     if is_str(obj_type):
#       obj_cls = registry.get(obj_type)   # 从注册器中提取出type对应的
类,注册器的作用就是将字符串与类进行对应
#       if obj_cls is None:
#           raise KeyError#     return obj_cls(**args)   # 返回类的对象.(
#               f'{obj_type} is not in the {registry.name} registry')
#     elif inspect.isclass(obj_type):
#       obj_cls = obj_type
#     else:
```

```
#       raise TypeError(
#           f'type must be a str or valid type, but got {type(obj_type)}')
#
#    return obj_cls(**args)   # 返回类的对象.
```

2.3 数据集的构建`datasets/builder.py`

利用`datasets/builder.py`中的build_dataset函数构建数据集.

这个构建过程包含pipeline的构建, 由于数据集均继承自`CustomeDataset`, 在其中使用Compose类对pipeline进行处理.

然后在`mmdet/datasets/pipelines/compose.py`中完成pipeline的build. (这个pipeline的构建,我找到吐血)

2.4 `mmdet/apis/train.py`中的训练过程

```
# 构建dataloader
data_loaders = [
    build_dataloader(
        ds,
        cfg.data.samples_per_gpu,
        cfg.data.workers_per_gpu,
        # cfg.gpus will be ignored if distributed
        len(cfg.gpu_ids),
        dist=distributed,
        seed=cfg.seed) for ds in dataset
]

    # 是否进行分布式训练
# put model on gpus
if distributed:
    find_unused_parameters = cfg.get('find_unused_parameters',
False)
    # Sets the `find_unused_parameters` parameter in
    # torch.nn.parallel.DistributedDataParallel
    model = MMDistributedDataParallel(
      model.cuda(),
      device_ids=[torch.cuda.current_device()],
      broadcast_buffers=False,
      find_unused_parameters=find_unused_parameters)
  else:
    model = MMDataParallel(
      model.cuda(cfg.gpu_ids[0]), device_ids=cfg.gpu_ids)


      # build runner
  optimizer = build_optimizer(model, cfg.optimizer)
  # 代码来自mmcv/runner/epoch_based_runner.py 与
mmcv/runner/base_runner.py
  runner = EpochBasedRunner(
    model,
    optimizer=optimizer,
    work_dir=cfg.work_dir,
    logger=logger,
    meta=meta)
  # an ugly workaround to make .log and .log.json filenames the
same
  runner.timestamp = timestamp


    # register hooks
  # 注册hook的过程实际就是按照优先级将对应的操作,添加到优先级队
列中
  runner.register_training_hooks(cfg.lr_config, optimizer_config,
                cfg.checkpoint_config, cfg.log_config,
                cfg.get('momentum_config', None))
    # 最后开始运行,代码在mmcv/runner/epoch_based_runner.py
    # 执行runner对象的运行操作
    runner.run(data_loaders, cfg.workflow, cfg.total_epochs)
```

2.5 Runner中的处理

最后分析runner的处理,主要代码存在`mmcv/runner`这个文件夹下.

主要分析:

- runner类的build过程
- runner类的hook的注册过程
- runner的成员函数run的执行过程

2.5.1 runner类的构建

代码存在于`mmcv/runner/based_runner.py`中.

`EpochBaseRunner`,继承`BaseRunner`,主要的runner初始化构建过程存在于`BaseRunner`类中.

```

class BaseRunner(metaclass=ABCMeta):
    """The base class of Runner, a training helper for PyTorch.

    All subclasses should implement the following APIs:
    @property
    def hooks(self):
        """list[:obj:`Hook`]: A list of registered hooks."""
        return self._hooks
    - ``run()``
    - ``train()``
    - ``val()``
    - ``save_checkpoint()``
    Args:
        model (:obj:`torch.nn.Module`): The model to be run.
        batch_processor (callable): A callable method that process a
data
            batch. The interface of this method should be
            `batch_processor(model, data, train_mode) -> dict`
        optimizer (dict or :obj:`torch.optim.Optimizer`): It can be either
an
            optimizer (in most cases) or a dict of optimizers (in models
that
            requires more than one optimizer, e.g., GAN).
        work_dir (str, optional): The working directory to save
checkpoints
            and logs. Defaults to None.
        logger (:obj:`logging.Logger`): Logger used during training.
             Defaults to None. (The default value is just for backward
             compatibility)
        meta (dict | None): A dict records some import information such
as
            environment info and seed, which will be logged in logger
hook.
            Defaults to None.
        max_epochs (int, optional): Total training epochs.
        max_iters (int, optional): Total training iterations.
    """

    def __init__(self,
                 model,
                 batch_processor=None,
                 optimizer=None,
                 work_dir=None,
                 logger=None,
                 meta=None,
                 max_iters=None,
                 max_epochs=None):
```

2.5.2 runner hook的注册过程

两种注册的方式代码存在于`mmcv/runner/base_runner.py`中

```
  #runner中的hooks是一个列表优先级高的放置在列表的前边,优先级低的放置在后边.
   @property
   def hooks(self):
       """list[:obj:`Hook`]: A list of registered hooks."""
       return self._hooks

   ##########################下边的这两种hook的注册方式,在前文hook机制的介绍中已经给出.

   def register_hook(self, hook, priority='NORMAL'):
       """Register a hook into the hook list.

       The hook will be inserted into a priority queue, with the specified
       priority (See :class:`Priority` for details of priorities).
       For hooks with the same priority, they will be triggered in the same
       order as they are registered.

       Args:
           hook (:obj:`Hook`): The hook to be registered.
           priority (int or str or :obj:`Priority`): Hook priority.
               Lower value means higher priority.
       """
       assert isinstance(hook, Hook)
       if hasattr(hook, 'priority'):
           raise ValueError('"priority" is a reserved attribute for hooks')
       priority = get_priority(priority)
```

```python
        hook.priority = priority
        # insert the hook to a sorted list
        inserted = False
        for i in range(len(self._hooks) - 1, -1, -1):
            if priority >= self._hooks[i].priority:
                self._hooks.insert(i + 1, hook)
                inserted = True
                break
        if not inserted:
            self._hooks.insert(0, hook)

    def register_hook_from_cfg(self, hook_cfg):
        """Register a hook from its cfg.

        Args:
            hook_cfg (dict): Hook config. It should have at least keys 'type'
              and 'priority' indicating its type and priority.

        Notes:
            The specific hook class to register should not use 'type' and
            'priority' arguments during initialization.
        """
        hook_cfg = hook_cfg.copy()
        priority = hook_cfg.pop('priority', 'NORMAL')
        hook = mmcv.build_from_cfg(hook_cfg, HOOKS)
        self.register_hook(hook, priority=priority)



        def register_training_hooks(self,
                                lr_config,
                                optimizer_config=None,
                                checkpoint_config=None,
                                log_config=None,
                                momentum_config=None):
        """Register default hooks for training.
    Default hooks include:

    - LrUpdaterHook
    - MomentumUpdaterHook
    - OptimizerStepperHook
    - CheckpointSaverHook
    - IterTimerHook
    - LoggerHook(s)
    """
    self.register_lr_hook(lr_config)
    self.register_momentum_hook(momentum_config)
    self.register_optimizer_hook(optimizer_config)
    self.register_checkpoint_hook(checkpoint_config)
    self.register_hook(IterTimerHook())
    self.register_logger_hooks(log_config)
```

2.5.3 runner中的run函数运行机制

代码存在于`mmcv/runner/epoch_base_runner.py`中这个类重写的run函数.

```
def run_iter(self, data_batch, train_mode, **kwargs):
    if self.batch_processor is not None:
        outputs = self.batch_processor(
            self.model, data_batch, train_mode=train_mode, **kwargs)
    elif train_mode:
        # 执行每次的迭代
        # 以faster rcnn为例: FasterRCNN类  ==> TwoStageDetector 类
==> BaseDetector类
        # 代码存在mmdet/models/detectors/base.py
        # 因此这个操作执行一次前向传播,这里边不包含反向传播与迭代
器的更新,这些操作存在于optimizer的hook中,
        outputs = self.model.train_step(data_batch, self.optimizer,
                                   **kwargs)
        else:
            outputs = self.model.val_step(data_batch, self.optimizer,
**kwargs)
        if not isinstance(outputs, dict):
            raise TypeError('"batch_processor()" or "model.train_step()"'
                            'and "model.val_step()" must return a dict')
        if 'log_vars' in outputs:
            self.log_buffer.update(outputs['log_vars'],
outputs['num_samples'])
        self.outputs = outputs

    def train(self, data_loader, **kwargs):
        self.model.train()
        self.mode = 'train'
        self.data_loader = data_loader
        # self._max_iters  一共需要迭代运行的次数.
        self._max_iters = self._max_epochs * len(self.data_loader)
        self.call_hook('before_train_epoch')
        time.sleep(2)  # Prevent possible deadlock during epoch
transition

        # 每个epoch的迭代过程
        for i, data_batch in enumerate(self.data_loader):
            # _inner_iter表示在每个epoch迭代的过程的步骤.
            self._inner_iter = i
            self.call_hook('before_train_iter')
            # 每个迭代步骤
            self.run_iter(data_batch, train_mode=True)
            self.call_hook('after_train_iter')
            # 所有的迭代步骤的逐步叠加过程记录.
            self._iter += 1

        self.call_hook('after_train_epoch')
        self._epoch += 1
```

```
  def run(self, data_loaders, workflow, max_epochs=None,
**kwargs):
    """Start running.

    Args:
      data_loaders (list[:obj:`DataLoader`]): Dataloaders for
training
        and validation.
      workflow (list[tuple]): A list of (phase, epochs) to specify the
        running order and epochs. E.g, [('train', 2), ('val', 1)] means
        running 2 epochs for training and 1 epoch for validation,
        iteratively.
    """
    assert isinstance(data_loaders, list)
    assert mmcv.is_list_of(workflow, tuple)
    assert len(data_loaders) == len(workflow)
    if max_epochs is not None:
      warnings.warn(
        'setting max_epochs in run is deprecated, '
        'please set max_epochs in runner_config',
DeprecationWarning)
      self._max_epochs = max_epochs

    assert self._max_epochs is not None, (
      'max_epochs must be specified during instantiation')

    for i, flow in enumerate(workflow):
      mode, epochs = flow
      if mode == 'train':
        self._max_iters = self._max_epochs * len(data_loaders[i])
        break

    work_dir = self.work_dir if self.work_dir is not None else 'NONE'
    self.logger.info('Start running, host: %s, work_dir: %s',
             get_host_info(), work_dir)
    self.logger.info('workflow: %s, max: %d epochs', workflow,
                 self._max_epochs)
        self.call_hook('before_run')

        # 执行所有训练过程的迭代.
        while self.epoch < self._max_epochs:
            for i, flow in enumerate(workflow):
                mode, epochs = flow
                if isinstance(mode, str):  # self.train()
                    if not hasattr(self, mode):
                        raise ValueError(
                            f'runner has no method named "{mode}" to run an '
                            'epoch')
                    # 获得一个函数,train()还是val()
                    epoch_runner = getattr(self, mode)
                else:
                    raise TypeError(
                        'mode in workflow must be a str, but got {}'.format(
                            type(mode)))

                for _ in range(epochs):
                    if mode == 'train' and self.epoch >= self._max_epochs:
                        break
                        # 执行train()或者Val()函数.
                    epoch_runner(data_loaders[i], **kwargs)

        time.sleep(1)  # wait for some hooks like loggers to finish
        self.call_hook('after_run')
```

至此,基本的框架的执行流程就已经大致有了一个了解,大致框架的基本组成有了基本的了解.

更深一部就需要分析模型的细节整体的代码.

分析框架中包含的模型源码