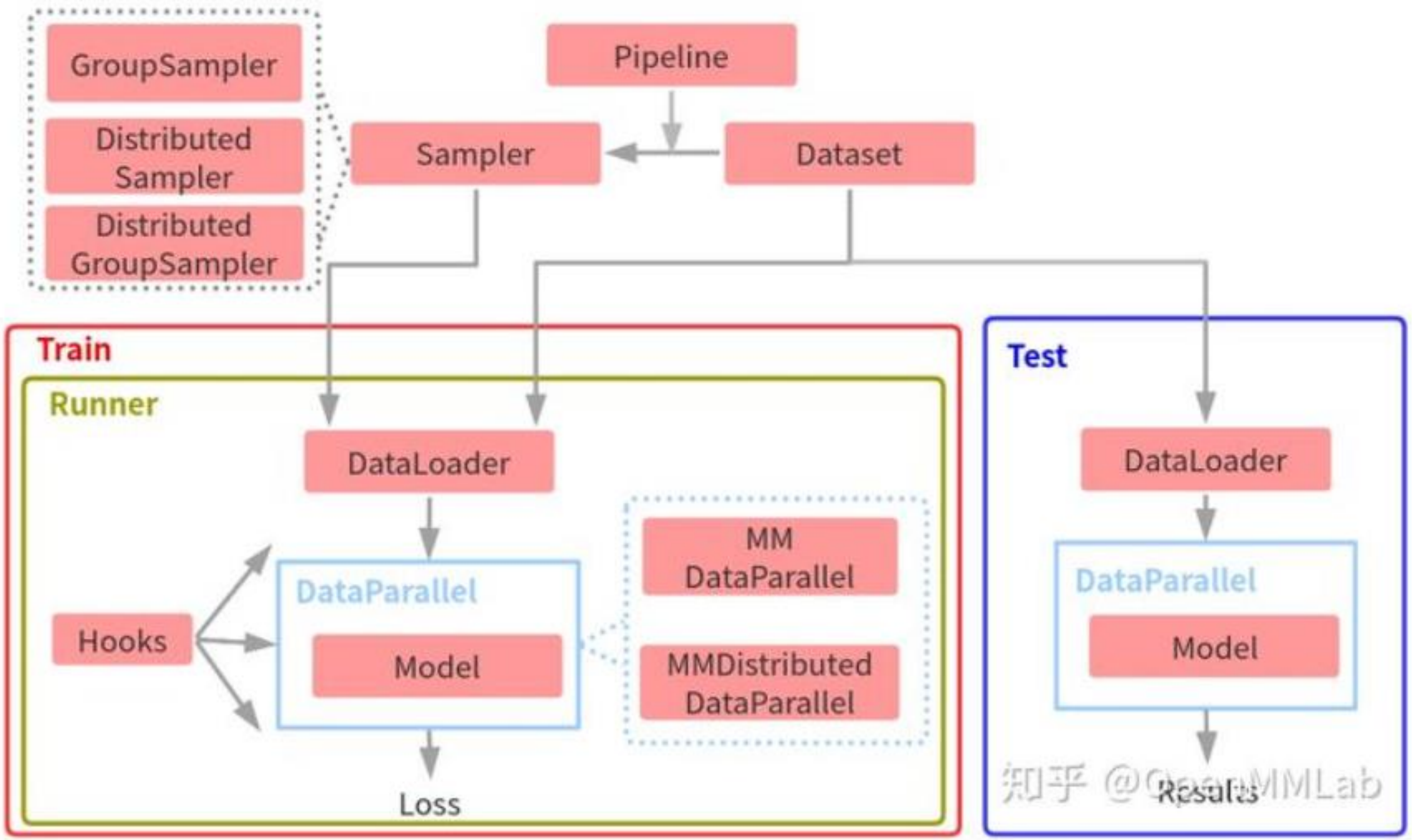
MMDetection2

概述

1. MMDetection 是一个基于 PyTorch 的目标检测开源工具箱。提供了各种检测算法，数据集构建方法等
2. MMCV 是一个面向计算机视觉的基础库，为Mmdet提供底层支持，包括通用IO接口，配置文件解析功能，注册器机制，hook机制，以及统筹全局训练测试流程的runner机制

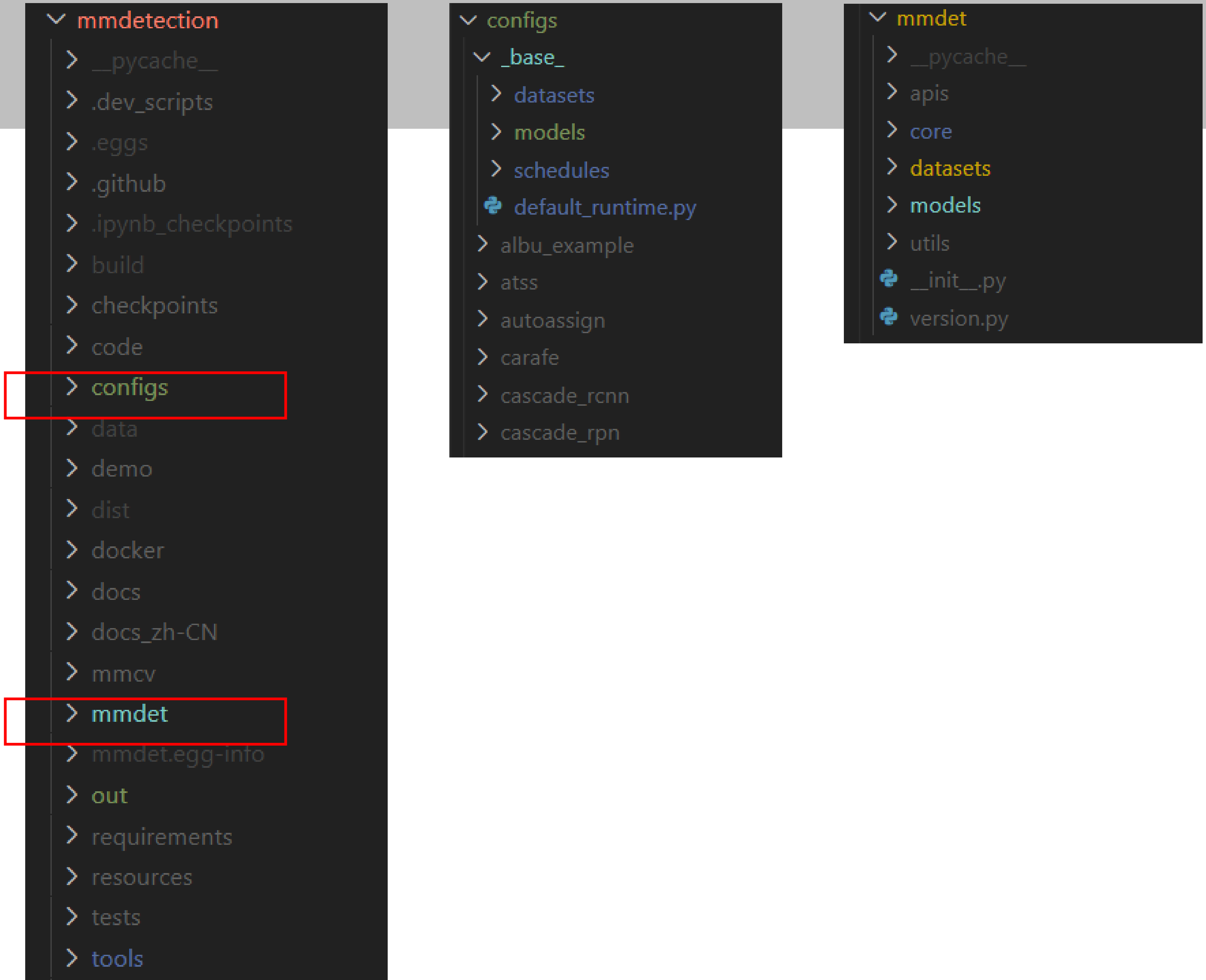
整体框架

Overall framework



配置相关初始化	cfg = Config.fromfile(args.config) 其他一些设置配置，例如 work_dir、gpu_id、logger 等等
相关类初始化	Model 初始化 Dataset、DataLoader 初始化 DataParallel 初始化
runner 初始化	EpochBasedRunner 初始化 注册 train/val 相关 hook 恢复权重等其余操作
runner 运行	判断是否训练完成

train()	val()
for i, data_batch in enumerate(data_loader)	
model.train_step()	model.val_step()


mmdetection
__pycache__
.dev_scripts
.eggs
.github
.ipynb_checkpoints
build
checkpoints
code
configs
data
demo
dist
docker
docs
docs_zh-CN
mmcv
mmdet
mmdet.egg-info
out
requirements
resources
tests
tools
configs
base
datasets
models
schedules
default_runtime.py
albu_example
atss
autoassign
carafe
cascade_rcnn
cascade_rpn
mmdet
__pycache__
apis
core
datasets
models
utils
__init__.py
version.py

配置相关初始化

Mmdetection2训练流程 —— 读取Config

训练的入口文件为tools/train.py　　一条简单启动命令如下：

Python tools/train.py configs/mask_rcnn/mask_rcnn_r50_fpn_1x_coco.py

这个python文件里面的内容为：

```
_base_ = [
    '../_base_/models/mask_rcnn_r50_fpn.py',
    '../_base_/datasets/coco_instance.py',
    '../_base_/schedules/schedule_1x.py', '../_base_/default_runtime.py'
]
```

而train.py定义了parse_args()函数读取命令行参数，保存进参数args中。

使用mmdetection的config类，读取_base_列表中的每个配置文件，依次涉及model，dataset，评估方法、优化器、lr以及runner，ckpt等的设置

读取Config

其他配置文件之外的参数还包括是否开启cudnn加速，workdir等

如果是分布式训练，要调用mmcv.runner封装的init_dist函数，和pytorch的分布式训练机制很像。命令行参数中还要加入GPU数量

```
init_dist(args.launcher, **cfg.dist_params)
_, world_size = get_dist_info()
cfg.gpu_ids = range(world_size)
```

大致逻辑即

设节点数为n，gpu数为g，进程总数为word_size = n*g，设置用于同步所有进程的主进程

```
os.environ['MASTER_ADDR'] = '10.57.23.164'
os.environ['MASTER_PORT'] = '8888'
mp.spawn(train, nprocs=args.gpus, args=(args,))  #用于生成每个进程
```

读取Config

在*init_dist*函数中运行*_distributed.init_process_group*

这个函数需要知道如何找到进程0（process 0），一边所有的进程都可以同步，也知道了一共要同步多少进程。每个独立的进程也要知道总共的进程数，以及自己在所有进程中的阶序（rank）,当然也要知道自己要用那张GPU

在之后定义dataLoader时，要用到分布式Sampler以实现每块GPU分到数据集的单独一部分

相关类初始化

Build model

```
model = build_detector(
        cfg.model,
        train_cfg=cfg.get('train_cfg'),
        test_cfg=cfg.get('test_cfg’))
```

调用了mmdet.model.build_detector,实际上是调用了Registry类的build函数以返回一个Detector类的实例

Registry类可以理解为一个字典，key是类名，value是类对象

用法示例：

```
  CATS = mmcv.Registry('cat')
  # 通过装饰器方式作用在想要加入注册器的具体类中
  @CATS.register_module()
  class BritishShorthair:
      pass
  # 类实例化
  CATS.get('BritishShorthair')(**args)
```

所有的backbone，neck，head，roi_extractor，detector，各类型dataset，优化器，取样器等，全部由registry管理

Init model weight

Model.init_weight()
所有model都是BaseModule的子类，而BaseModule则是nn.Module的子类，在其基础上加入了init_weight方法
逻辑为for i in module.children():
    initialize(i，init_cfg)

Initialize方法中会根据init_cfg的内容，实例化一个initializer，如：

```python
module = nn.Linear(2, 3, bias=True)
init_cfg = dict(type='Constant', layer='Linear', val =1 , bias =2)
initialize(module, init_cfg)
```

Build dataset

```
datasets = [build_dataset(cfg.data.train)]
```

build_dataset同样是调用Registry的build函数，返回一个数据集类实例，maskrcnn即CocoDataset。

所有的Dataset类的基类为CustomDataset，而它又继承自torch.utils.data.Dataset
根据cfg中第路径将数据加载进来后，很重要的一部分是将数据传入pipeline

```
cfg.data.train
```

✓ 0.1s

```
{'type': 'CocoDataset',
 'ann_file': '/root/mmdetection/data/coco/annotations/mytrain.json',
 'img_prefix': '/root/mmdetection/data/coco/mytrain/',
 'pipeline': [{'type': 'LoadImageFromFile'},
  {'type': 'LoadAnnotations', 'with_bbox': True, 'with_mask': True},
  {'type': 'Resize', 'img_scale': (1333, 800), 'keep_ratio': True},
  {'type': 'RandomFlip', 'flip_ratio': 0.5},
  {'type': 'Normalize',
   'mean': [123.675, 116.28, 103.53],
   'std': [58.395, 57.12, 57.375],
   'to_rgb': True},
  {'type': 'Pad', 'size_divisor': 32},
  {'type': 'DefaultFormatBundle'},
```

Dataset

CoCo：

```
{
    "info": info, // dict
     "licenses": [license], // list ，内部是dict
     "images": [image], // list ，内部是dict
     "annotations": [annotation], // list ，内部是dict
     "categories": // list ，内部是dict
}
```

```
ges']

: 3,
e': '000000391895.jpg',
': 'http://images.cocodataset.org/train2017/000000391895.jpg',
 360,
640,
tured': '2013-11-14 11:18:45',
rl': 'http://farm9.staticflickr.com/8186/8119368305_4e622c8349_z.jpg',
895},
```

```
coco['annotations']
✓ 3.5s

[{'segmentation': [[239.97,
    260.24,
    222.04,
    270.49,
    199.84,
    253.41,
    213.5,
    227.79,
    259.62,
    200.46,
    274.13,
    202.17,
    277.55,
    210.71,
    249.37,
    253.41,
    237.41,
    264.51,
    242.54,
    261.95,
    228.87,
    271.34]],
  'area': 2765.1486500000005,
  'iscrowd': 0,
  'image_id': 558840,
  'bbox': [199.84, 200.46, 77.71, 70.88],
  'category_id': 58,
  'id': 156},
```

Build dataset

Datasets.data_infos = load_annotations(ann_file)

```
datasets.data_infos
```

✓ 1.1s


```
[{'license': 3,
  'file_name': '000000391895.jpg',
  'coco_url': 'http://images.cocodataset.org/train2017/000000391895.jpg',
  'height': 360,
  'width': 640,
  'date_captured': '2013-11-14 11:18:45',
  'flickr_url': 'http://farm9.staticflickr.com/8186/8119368305_4e622c8349_z.jpg',
  'id': 391895,
  'filename': '000000391895.jpg'},
```

Build dataset

```
self._set_group_flag()
```

```
datasets.flag
```

✓ 0.3s

```
array([1, 1, 1, ..., 1, 1, 1], dtype=uint8)
```

```
self.pipeline = Compose(pipeline)
```

```
cfg.data.train.pipeline
```

✓ 0.1s

```
[{'type': 'LoadImageFromFile'},
 {'type': 'LoadAnnotations', 'with_bbox': True, 'with_mask': True},
 {'type': 'Resize', 'img_scale': (1333, 800), 'keep_ratio': True},
 {'type': 'RandomFlip', 'flip_ratio': 0.5},
 {'type': 'Normalize',
  'mean': [123.675, 116.28, 103.53],
  'std': [58.395, 57.12, 57.375],
  'to_rgb': True},
 {'type': 'Pad', 'size_divisor': 32},
 {'type': 'DefaultFormatBundle'},
```

Build dataset

```
def __getitem__(self, idx):
        data = self.prepare_train_img(idx)
        return data
```

```
def prepare_train_img(self, idx):

        img_info = self.data_infos[idx]
        ann_info = self.get_ann_info(idx)
        results = dict(img_info=img_info, ann_info=ann_info)
        if self.proposals is not None:
            results['proposals'] = self.proposals[idx]
        self.pre_pipeline(results)
        return self.pipeline(results)
```

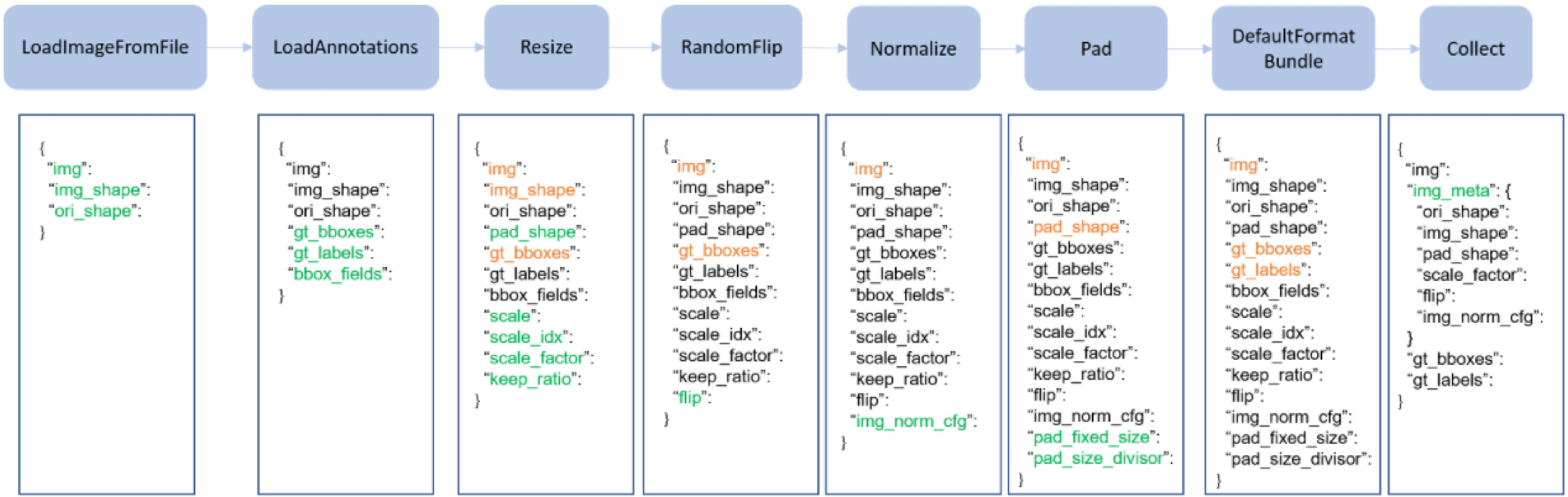
Build dataset

```
for key in datasets.get_ann_info(0):
    print(key)
print(datasets.get_ann_info(0))
```

0.2s

```
bboxes
labels
bboxes_ignore
masks
seg_map
{'bboxes': array([[359.17, 146.17, 471.62, 359.74],
       [339.88,  22.16, 493.76, 322.89],
       [471.64, 172.82, 507.56, 220.92],
       [486.01, 183.31, 516.64, 218.29]], dtype=float32), 'labels': array([3, 0, 0, 1]), 'bboxes_ignor
array([], shape=(0, 4), dtype=float32), 'masks': [[[376.97, 176.91, 398.81, 176.91, 396.38, 147.78, 44
146.17, 448.16, 172.05, 448.16, 178.53, 464.34, 186.62, 464.34, 192.28, 448.97, 195.51, 447.35, 235.96
258.62, 454.63, 268.32, 462.72, 276.41, 471.62, 290.98, 456.25, 298.26, 439.26, 292.59, 431.98, 308.77
313.63, 436.02, 316.86, 429.55, 322.53, 419.84, 354.89, 402.04, 359.74, 401.24, 312.82, 370.49, 303.92
299.87, 391.53, 280.46, 385.06, 278.84, 381.01, 278.84, 359.17, 269.13, 373.73, 261.85, 374.54, 256.19
231.11, 383.44, 205.22, 385.87, 192.28, 373.73, 184.19]], [[352.55, 146.82, 353.61, 137.66, 356.07, 11
357.13, 94.7, 357.13, 84.49, 363.12, 73.92, 370.16, 68.64, 370.16, 66.53, 368.4, 63.71, 368.05, 54.56,
```

Build dataset



Build dataset

```
datasets.pipeline
```

✓ 0.3s Pytho

```
Compose(
    LoadImageFromFile(to_float32=False, color_type='color', file_client_args={'backend': 'disk'})
    LoadAnnotations(with_bbox=True, with_label=True, with_mask=True, with_seg=False, poly2mask=True, poly2mask=
{'backend': 'disk'})
    Resize(img_scale=[(1333, 800)], multiscale_mode=range, ratio_range=None, keep_ratio=True,
bbox_clip_border=True)
    RandomFlip(flip_ratio=0.5)
    Normalize(mean=[123.675 116.28  103.53 ], std=[58.395 57.12  57.375], to_rgb=True)
    Pad(size=None, size_divisor=32, pad_val=0)
    DefaultFormatBundle
    Collect(keys=['img', 'gt_bboxes', 'gt_labels', 'gt_masks'], meta_keys=('filename', 'ori_filename',
'ori_shape', 'img_shape', 'pad_shape', 'scale_factor', 'flip', 'flip_direction', 'img_norm_cfg'))
)
```

Build dataset

Mmdetection定义了DataContainer类用来包装Tensor变量，原因为：
通常情况下，为了组成batch，要把Tensor叠加起来，局限性是：
1.所有张量的大小必须相同。2.类型有限（numpy数组或张量）。
在detection任务中，一个图片具有的实例，bbox数量都不相同，而训练batch中是以图片为单位的，利用DataContainer包装Tensor可以克服这样的局限性

```
gt_bboxes : DataContainer(tensor([[  10.7553,  319.4726,  551.0556,  763.1846],
        [ 624.6176,   17.7485, 1018.6393,  669.3712]]))
gt_labels : DataContainer(tensor([0, 1]))
gt_masks : DataContainer(BitmapMasks(num_masks=2, height=800, width=1056))
```

Build dataset

datasets[0]

```
{
    "img_meta": DataContainer({
        'filename': '/opt/data/private/qmx/data/coco/train2017/000000391895.jpg',
        'ori_filename': '000000391895.jpg',
        'ori_shape': (360, 640, 3),
        'img_shape': (750, 1333, 3),
        'pad_shape': (768, 1344, 3),
        'scale_factor': array([2.0828125, 2.0833333, 2.0828125, 2.0833333], dtype=float32),
        'flip': True, 'flip_direction': 'horizontal',
        'img_norm_cfg': {'mean': array([123.675, 116.28 , 103.53 ], dtype=float32),
                         'std': array([58.395, 57.12 , 57.375], dtype=float32), 'to_rgb': True}
        }),
    "img": DataContainer(Tensor()),
    "gt_bboxes": DataContainer(Tensor(gt_bbox数量, 4)),
    "gt_labels": DataContainer(tensor(gt_bbox数量,)),
    "gt_masks": DataContainer(BitmapMasks(num_masks= gt_bbox数量, height=768, width=1344))}

}
```

```
train_detector(
        model,
        datasets,
        cfg,
        distributed=distributed,
        validate=(not False),
        timestamp=timestamp,
        meta=meta)
```

接下来就要进入到mmdet.apis.train_detector方法中，以下的内容就不在tools/trian.py中

Build DataLoader

```
data_loaders = [
        build_dataloader(
            ds,
            cfg.data.samples_per_gpu,  #每个batch每GPU分配多少数据
            cfg.data.workers_per_gpu,  #加载数据的进程数
            # cfg.gpus will be ignored if distributed
            len(cfg.gpu_ids),
            dist=distributed,
            seed=cfg.seed) for ds in dataset
    ]
```

有几个workflow就定义几个data_loader，workflow可以单有train，也可以train，val。
build_dataloader方法在mmdet.dataset.builder.py中定义
逻辑即根据GPU的标号RANK，定义DistributedGroupSampler来分组采样数据，最后调用torch.utils.data.DataLoader

Build DataLoader

```
data_loader = DataLoader(
        dataset,
        batch_size=batch_size,
        sampler=sampler,
        num_workers=num_workers,
        collate_fn=partial(collate, samples_per_gpu=samples_per_gpu), #用来将数据组成Batch的函数，collate为mmcv中针对dataContainer进行组batch的函数
        pin_memory=False,
        worker_init_fn=init_fn,
        **kwargs)
```

```
gt_bboxes : DataContainer([[tensor([[360.0358, 316.2523, 784.5424, 791.4764],
        [695.7512,   1.6912, 935.9104, 769.4909]]), tensor([[ 496.7310,   42.1968, 1095.8934,  621.1364],
        [  90.8195,   21.0984,  562.5544,  638.0151]])]])
gt_labels : DataContainer([[tensor([0, 1]), tensor([0, 1])]])
gt_masks : DataContainer([[BitmapMasks(num_masks=2, height=800, width=1184), BitmapMasks(num_masks=2, height=800,
width=1216)]])
```

DDP

```
model = MMDistributedDataParallel(
            model.cuda(),
            device_ids=[torch.cuda.current_device()],
            broadcast_buffers=False,
            find_unused_parameters=find_unused_parameters)
```

将模型封装为一个 DistributedDataParallel 模型。这将把模型复制到GPU上进行处理。
MMDistributedDataParallel是pytorch中DDP的子类，可以支持dataContainer并且定义了train_step
Val_step等方法。

Build optimizer

```
optimizer = build_optimizer(model, cfg.optimizer)
```

```
cfg.optimizer
✓ 0.1s
{'type': 'SGD', 'lr': 0.02, 'momentum': 0.9, 'weight_decay': 0.0001}
```

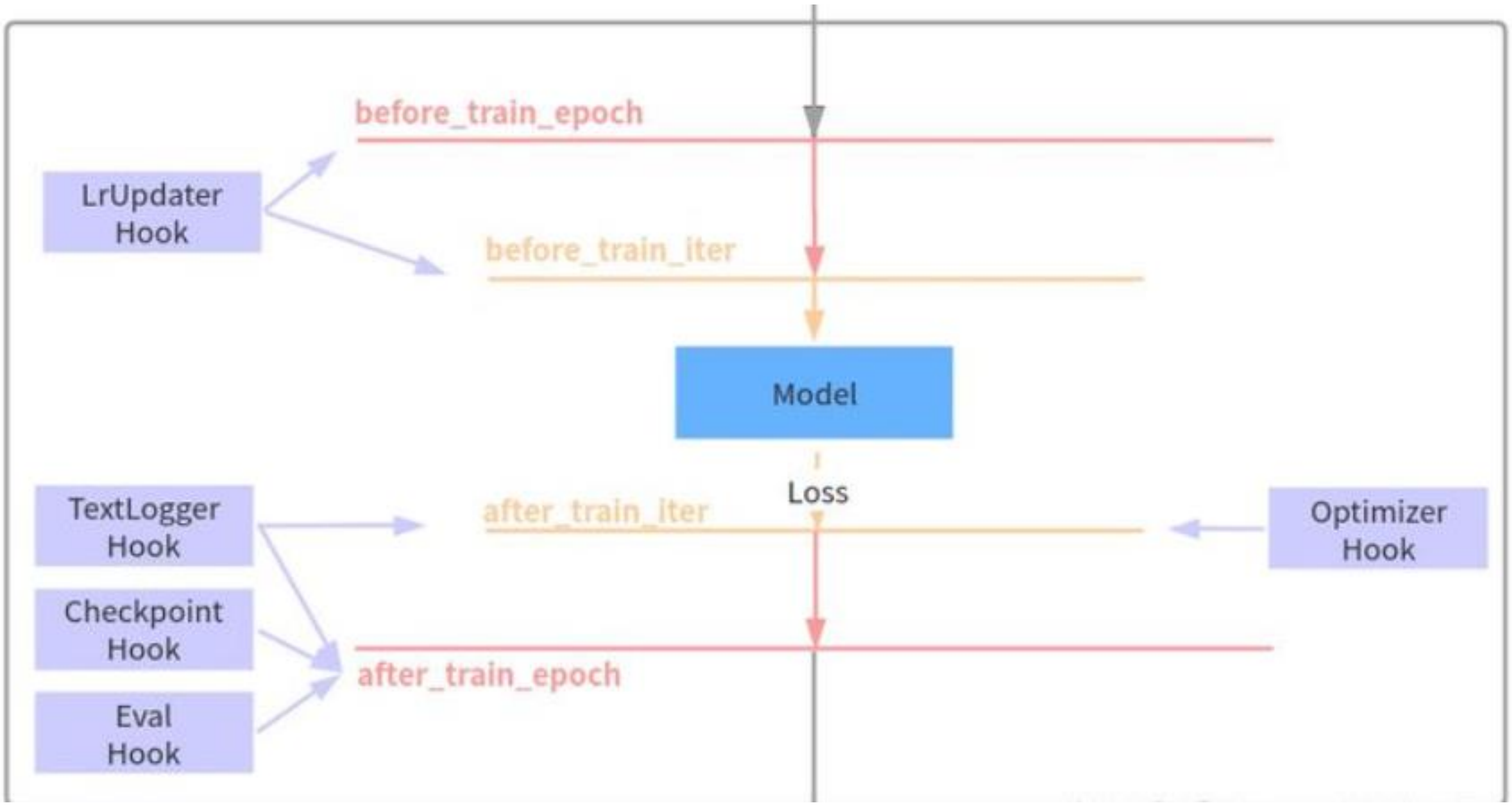
runner初始化

Build Runner

```
runner = build_runner(
        cfg.runner,
        default_args=dict(
            model=model,
            optimizer=optimizer,
            work_dir=cfg.work_dir,
            logger=logger,
            meta=meta))
```

Runner分为EpochBasedRunner和IterBasedRunner，他们都继承自BaseRuner，初始化并没有具体的操作
只是填充一些属性值，其中很关键的有self._hook= []，之后会向其中填充各hook实例

Build Runner



Registry hook

```
runner.register_training_hooks(cfg.lr_config, cfg.optimizer_config,
                               cfg.checkpoint_config, cfg.log_config,
                               cfg.get('momentum_config', None))
```

```
cfg.lr_config: {'policy': 'step', 'warmup': 'linear', 'warmup_iters': 500, 'warmup_ratio': 0.001, 'step': [8, 11]}
cfg.optimizer_config: {'grad_clip': None}
cfg.checkpoint_config: {'interval': 1}
cfg.log_config: {'interval': 5, 'hooks': [{'type': 'TextLoggerHook'}]}
```

```
runner._hooks
```

✓ 0.7s

```
[<mmcv.runner.hooks.lr_updater.StepLrUpdaterHook at 0x7f02c65f6e50>,
 <mmcv.runner.hooks.optimizer.OptimizerHook at 0x7f02c65f6e90>,
 <mmcv.runner.hooks.checkpoint.CheckpointHook at 0x7f02c65f6fd0>,
 <mmcv.runner.hooks.iter_timer.IterTimerHook at 0x7f02c65f6ad0>,
 <mmcv.runner.hooks.logger.text.TextLoggerHook at 0x7f02c6a99b50>]
```

Registry hook

MMdetection中的HOOK可以理解为一种触发器，它规定了在算法训练过程中的种种操作
每个继承自HOOK基类的hook子类，都要实现

```
def before_run(self, runner)
def after_run (self, runner)
def before_epoch (self, runner)
def after_epoch (self, runner)
def before_iter(self, runner)
def after_iter(self, runner)
```

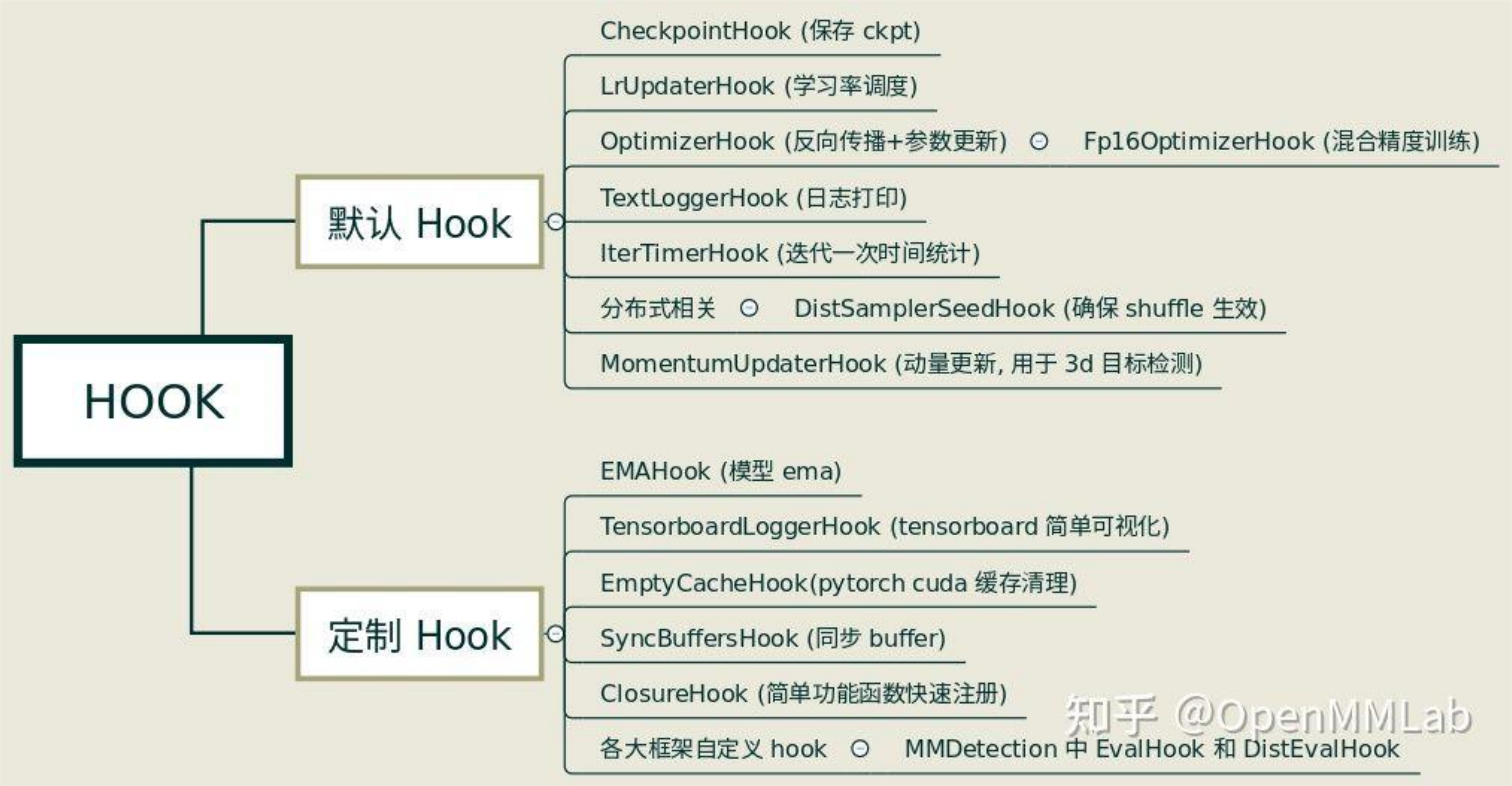
等一系列方法
Mmdet内置的hook类通过runner中的register_training_hooks方法，按照事先定好的hook优先级填充进runner._hook列表中
自定义的hook类通过register_hook方法注册
在训练过程中的特定位置，只需要调用runner.call_hook(“before_run”)，便可以按照优先级，调用_hook列表中所有hook实例的before_run方法

Registry hook

举一个hook运行的具体实例：
After_train_iter的optimizerHook，他会进行反向传播，参数更新

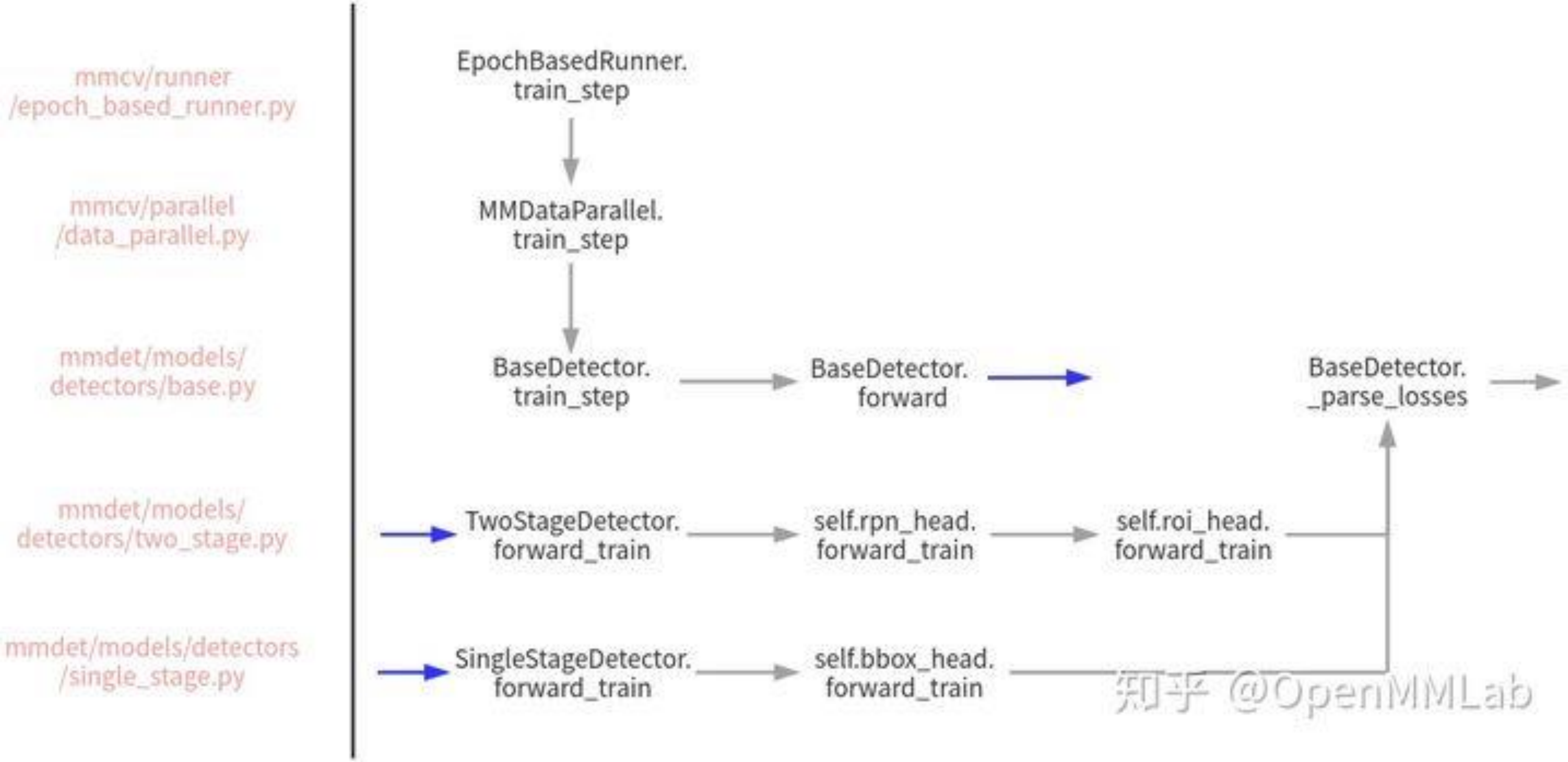
```
@HOOKS.register_module()
class OptimizerHook(Hook):

    def __init__(self, grad_clip=None):
        self.grad_clip = grad_clip

    def after_train_iter(self, runner):
        runner.optimizer.zero_grad()
        runner.outputs['loss'].backward()
        if self.grad_clip is not None:
            grad_norm = self.clip_grads(runner.model.parameters())
        runner.optimizer.step()
```

Registry hook



Runner运行

Run


mmcv/runner
/epoch_based_runner.py
mmcv/parallel
/data_parallel.py
mmdet/models/
detectors/base.py
mmdet/models/
detectors/two_stage.py
mmdet/models/detectors
/single_stage.py
EpochBasedRunner.
train_step
MMDataParallel.
train_step
BaseDetector.
train_step
BaseDetector.
forward
BaseDetector.
_parse_losses
TwoStageDetector.
forward_train
self.rpn_head.
forward_train
self.roi_head.
forward_train
SingleStageDetector.
forward_train
self.bbox_head.
forward_train
知乎 @OpenMMLab

Run

```
runner.run(data_loaders, cfg.workflow)
```

```
cfg.workflow
✓ 0.8s
[('train', 1)]
```

run 方法调用后才是真正开启工作流。设置 workflow = [('train', 3), ('val',1)]， 表示先训练 3 个 epoch ， 然后切换到 val 工作流， 运行 1 个 epoch， 然后循环， 直到训练 epoch 次数达到指定值

对于 EpochBasedRunner， train模式调用runner.train方法，

```
def train(self, data_loader, **kwargs):
        self.data_loader = data_loader
        self._max_iters = self._max_epochs * len(self.data_loader)
        self.call_hook('before_train_epoch')
        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, **kwargs)
            self.call_hook('after_train_iter')
            self._iter += 1
        self.call_hook('after_train_epoch')
        self._epoch += 1
```

Runner.run_iter

```
def run_iter(self, data_batch, train_mode, **kwargs):
    if train_mode:
        # 对于每次迭代，最终是调用如下函数
        outputs = self.model.train_step(data_batch,...)
    else:
        # 对于每次迭代，最终是调用如下函数
        outputs = self.model.val_step(data_batch,...)

    if 'log_vars' in outputs:
        self.log_buffer.update(outputs['log_vars'],...)
    self.outputs = outputs
```

此处`model.train_step`，是指的 MMDataParallel 或者 MMDistributedDataParallel包装后的model