Table of Contents

CONFIG

MMEngine implements an abstract configuration class (`Config`) to provide a unified configuration access interface for users. `Config` supports different types of configuration file, including `python`, `json` and `yaml`, and you can choose the type according to your preference. `Config` overrides some magic method, which could help you access the data stored in `Config` just like getting values from `dict`, or getting attributes from instances. Besides, `Config` also provides an inheritance mechanism, which could help you better organize and manage the configuration files.

Before starting the tutorial, let's download the configuration files needed in the tutorial (it is recommended to execute them in a temporary directory to facilitate deleting these files latter.):

```
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/config_sgd.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/cross_repo.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/custom_imports.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/demo_train.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/example.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/learn_read_config.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/my_module.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/optimizer_cfg.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/predefined_var.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/replace_data_root.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/replace_num_classes.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/refer_base_var.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/resnet50_delete_key.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/resnet50_lr0.01.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/resnet50_runtime.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/resnet50.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/runtime_cfg.py
wget https://raw.githubusercontent.com/open-mmlab/mmengine/main/docs/resources/config/modify_base_var.py
```

- NOTE

  The `Config` supports two styles of configuration files: text style and pure Python style (introduced in v0.8.0). Each has its own characteristics while maintaining a unified interface for calling. For users who are not familiar with the basic usage of the `Config`, it is recommended to start reading from the section on Read the configuration file to understand the functionality of the `Config` and the syntax of text style configuration files. In some cases, the syntax of text style configuration files is more concise and compatible with different formats such as `json` and `yaml`. If you prefer a more flexible syntax for configuration files, it is recommended to use the Pure Python Style Configuration Files (beta).

Read the configuration file

`Config` provides a uniform interface `Config.fromfile()` to read and parse configuration files.

A valid configuration file should define a set of key-value pairs, and here are a few examples:

Python:

```
test_int = 1
test_list = [1, 2, 3]
test_dict = dict(key1='value1', key2=0.1)
```

Json:

```
{
  "test_int": 1,
  "test_list": [1, 2, 3],
  "test_dict": {"key1": "value1", "key2": 0.1}
}
```

YAML:

```
test_int: 1
test_list: [1, 2, 3]
test_dict:
  key1: "value1"
  key2: 0.1
```

For the above three formats, assuming the file names are `config.py`, `config.json`, and `config.yml`. Loading these files with `Config.fromfile('config.xxx')` will return the same result, which contain `test_int`, `test_list` and `test_dict` 3 variables.

Let's take `config.py` as an example:

```
from mmengine.config import Config

cfg = Config.fromfile('learn_read_config.py')
print(cfg)
```

```
Config (path: learn_read_config.py): {'test_int': 1, 'test_list': [1, 2, 3], 'test_dict': {'key1': 'value1', 'key2': 0.1}}
```

How to use `Config`

After loading the configuration file, we can access the data stored in `Config` instance just like getting/setting values from `dict`, or getting/setting attributes from instances.

```
print(cfg.test_int)
print(cfg.test_list)
print(cfg.test_dict)
cfg.test_int = 2

print(cfg['test_int'])
print(cfg['test_list'])
print(cfg['test_dict'])
cfg['test_list'][1] = 3
print(cfg['test_list'])
```

```
1
[1, 2, 3]
{'key1': 'value1', 'key2': 0.1}
2
[1, 2, 3]
{'key1': 'value1', 'key2': 0.1}
[1, 3, 3]
```

- NOTE

  The `dict` object parsed by `Config` will be converted to `ConfigDict`, and then we can access the value of the `dict` the same as accessing the attribute of an instance.

We can use the `Config` combination with the Registry to build registered instance easily.

Here is an example of defining optimizers in a configuration file.

`config_sgd.py`:

```
optimizer = dict(type='SGD', lr=0.1, momentum=0.9, weight_decay=0.0001)
```

Suppose we have defined a registry `OPTIMIZERS` which includes various optimizers. Then we can build the optimizer as below

```
from mmengine import Config, optim
from mmengine.registry import OPTIMIZERS

import torch.nn as nn

cfg = Config.fromfile('config_sgd.py')

model = nn.Conv2d(1, 1, 1)
cfg.optimizer.params = model.parameters()
optimizer = OPTIMIZERS.build(cfg.optimizer)
print(optimizer)
```

```
SGD (
Parameter Group 0
    dampening: 0
    foreach: None
    lr: 0.1
    maximize: False
    momentum: 0.9
    nesterov: False
    weight_decay: 0.0001
)
```

Inheritance between configuration files

Sometimes, the difference between two different configuration files is so small that only one field may be changed. Therefore, it's unwise to copy and paste everything only to modify one line, which makes it hard for us to locate the specific difference after a long time.

In another case, multiple configuration files may have the same batch of fields, and we have to copy and paste them in different configuration files. It will also be hard to maintain these fields in a long time.

We address these issues with inheritance mechanism, detailed as below.

Overview of inheritance mechanism

Here is an example to illustrate the inheritance mechanism.

`optimizer_cfg.py`:

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
```

`resnet50.py`:

```
_base_ = ['optimizer_cfg.py']
model = dict(type='ResNet', depth=50)
```

Although we don't define `optimizer` in `resnet50.py`, since we wrote `_base_ = ['optimizer_cfg.py']`, it will inherit the fields defined in `optimizer_cfg.py`.

```
cfg = Config.fromfile('resnet50.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.02, 'momentum': 0.9, 'weight_decay': 0.0001}
```

`_base_` is a reserved field for the configuration file. It specifies the inherited base files for the current file. Inheriting multiple files will get all the fields at the same time, but it requires that there are no repeated fields defined in all base files.

`runtime_cfg.py`:

```
gpu_ids = [0, 1]
```

`resnet50_runtime.py`:

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
```

In this case, reading the `resnet50_runtime.py` will give you 3 fields `model`, `optimizer`, and `gpu_ids`.

```
cfg = Config.fromfile('resnet50_runtime.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.02, 'momentum': 0.9, 'weight_decay': 0.0001}
```

By this way, we can disassemble the configuration file, define some general configuration files, and inherit them in the specific configuration file. This could avoid defining a lot of duplicated contents in multiple configuration files.

Modify the inherited fields

Sometimes, we want to modify some of the fields in the inherited files. For example we want to modify the learning rate from 0.02 to 0.01 after inheriting `optimizer_cfg.py`.

In this case, you can simply redefine the fields in the new configuration file. Note that since the optimizer field is a dictionary, we only need to redefine the modified fields. This rule also applies to adding fields.

`resnet50_lr0.01.py`:

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
optimizer = dict(lr=0.01)
```

After reading this configuration file, you can get the desired result.

```
cfg = Config.fromfile('resnet50_lr0.01.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.01, 'momentum': 0.9, 'weight_decay': 0.0001}
```

For non-dictionary fields, such as integers, strings, lists, etc., they can be completely overwritten by redefining them. For example, the code block below will change the value of the `gpu_ids` to `[0]`.

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
gpu_ids = [0]
```

Delete key in `dict`

Sometimes we not only want to modify or add the keys, but also want to delete them. In this case, we need to set `_delete_=True` in the target field(`dict`) to delete all the keys that do not appear in the newly defined dictionary.

`resnet50_delete_key.py`:

```
_base_ = ['optimizer_cfg.py', 'runtime_cfg.py']
model = dict(type='ResNet', depth=50)
optimizer = dict(_delete_=True, type='SGD', lr=0.01)
```

At this point, `optimizer` will only have the keys `type` and `lr`. `momentum` and `weight_decay` will no longer exist.

```
cfg = Config.fromfile('resnet50_delete_key.py')
print(cfg.optimizer)
```

```
{'type': 'SGD', 'lr': 0.01}
```

Reference of the inherited file

Sometimes we want to reuse the field defined in `_base_`, we can get a copy of the corresponding variable by using `{{_base_.xxxx}}`:

`refer_base_var.py`

```
_base_ = ['resnet50.py']
a = {{_base_.model}}
```

```
cfg = Config.fromfile('refer_base_var.py')
print(cfg.a)
```

```
{'type': 'ResNet', 'depth': 50}
```

We can use this way to get the variables defined in `_base_` in the `json`, `yaml`, and `python` configuration files.

Although this way is general for all types of files, there are some syntactic limitations that prevent us from taking full advantage of the dynamic nature of the `python` configuration file. For example, if we want to modify a variable defined in `_base_`:

```
_base_ = ['resnet50.py']
a = {{_base_.model}}
a['type'] = 'MobileNet'
```

The `Config` is not able to parse such a configuration file (it will raise an error when parsing). The `Config` provides a more `pythonic` way to modify base variables for `python` configuration files.

`modify_base_var.py`:

```
_base_ = ['resnet50.py']
a = _base_.model
a.type = 'MobileNet'
```

```
cfg = Config.fromfile('modify_base_var.py')
print(cfg.a)
```

```
{'type': 'MobileNet', 'depth': 50}
```

Dump the configuration file

The user may pass some parameters to modify some fields of the configuration file at the entry point of the training script. Therefore, we provide the `dump` method to export the changed configuration file.

Similar to reading the configuration file, the user can choose the format of the dumped file by using `cfg.dump('config.xxx')`. `dump` can also export configuration files with inheritance relationships, and the dumped files can be used independently without the files defined in `_base_`.

Based on the `resnet50.py` defined above, we can load and dump it like this:

```
cfg = Config.fromfile('resnet50.py')
cfg.dump('resnet50_dump.py')
```

`resnet50_dump.py`

```
optimizer = dict(type='SGD', lr=0.02, momentum=0.9, weight_decay=0.0001)
model = dict(type='ResNet', depth=50)
```

Similarly, we can dump configuration files in `json`, `yaml` format:

`resnet50_dump.yaml`

```
model:
  depth: 50
  type: ResNet
optimizer:
  lr: 0.02
  momentum: 0.9
  type: SGD
  weight_decay: 0.0001
```

`resnet50_dump.json`

```
{"optimizer": {"type": "SGD", "lr": 0.02, "momentum": 0.9, "weight_decay": 0.0001}, "model": {"type": "ResNet", "depth": 50}}
```

```
cfg = Config(dict(a=1, b=2))
cfg.dump('dump_dict.py')
```

dump_dict.py

```
a=1
b=2
```

Advanced usage

In this section, we'll introduce some advanced usage of the `Config`, and some tips that could make it easier for users to develop and use downstream repositories.

- NOTE

  If you use pure Python style configuration file. Advanced usage should not be used except for the function described in "Modify the fields in command line"

Predefined fields

Sometimes we need some fields in the configuration file, which are related to the path to the workspace. For example, we define a working directory in the configuration file that holds the models and logs for this set of experimental configurations. We expect to have different working directories for different configuration files. A common choice is to use the configuration file name directly as part of the working directory name. Taking `predefined_var.py` as an example:

```
work_dir = './work_dir/{{fileBasenameNoExtension}}'
```

Here `{{fileBasenameNoExtension}}` means the filename without suffix `.py` of the config file, and the variable in `{{}}` will be interpreted as `predefined_var`

```
cfg = Config.fromfile('./predefined_var.py')
print(cfg.work_dir)
```

```
./work_dir/predefined_var
```

Currently, there are 4 predefined fields referenced from the relevant fields defined in VS Code.

- `{{fileDirname}}` - the directory name of the current file, e.g. `/home/your-username/your-project/folder`
- `{{fileBasename}}` - the filename of the current file, e.g. `file.py`
- `{{fileBasenameNoExtension}}` - the filename of the current file without the extension, e.g. `file`
- `{{fileExtname}}` - the extension of the current file, e.g. `.py`

Modify the fields in command line

Sometimes we only want to modify part of the configuration and do not want to modify the configuration file itself. For example, if we want to change the learning rate during the experiment but do not want to write a new configuration file, the common practice is to pass the parameters at the command line to override the relevant configuration.

If we want to modify some internal parameters, such as the learning rate of the optimizer, the number of channels in the convolution layer etc., `Config` provides a standard procedure that allows us to modify the parameters at any level easily from the command line.

Training script:

`demo_train.py`

```python
import argparse

from mmengine.config import Config, DictAction


def parse_args():
    parser = argparse.ArgumentParser(description='Train a model')
    parser.add_argument('config', help='train config file path')
    parser.add_argument(
        '--cfg-options',
        nargs='+',
        action=DictAction,
        help='override some settings in the used config, the key-value pair '
        'in xxx=yyy format will be merged into config file. If the value to '
        'be overwritten is a list, it should be like key="[a,b]" or key=a,b '
        'It also allows nested list/tuple values, e.g. key="[(a,b),(c,d)]" '
        'Note that the quotation marks are necessary and that no white space '
        'is allowed.')

    args = parser.parse_args()
    return args


def main():
    args = parse_args()
    cfg = Config.fromfile(args.config)
    if args.cfg_options is not None:
        cfg.merge_from_dict(args.cfg_options)
    print(cfg)


if __name__ == '__main__':
    main()
```

The sample configuration file is as follows.

`example.py`

```python
model = dict(type='CustomModel', in_channels=[1, 2, 3])
optimizer = dict(type='SGD', lr=0.01)
```

We can modify the internal fields from the command line by `.` For example, if we want to modify the learning rate, we only need to execute the script like this:

```
python demo_train.py ./example.py --cfg-options optimizer.lr=0.1
```

```
Config (path: ./example.py): {'model': {'type': 'CustomModel', 'in_channels': [1, 2, 3]}, 'optimizer': {'type': 'SGD', 'lr': 0.1}}
```

We successfully modified the learning rate from 0.01 to 0.1. If we want to change a list or a tuple, such as `in_channels` in the above example. We need to put double quotes around `()`, `[]` when assigning the value on the command line.

```
python demo_train.py ./example.py --cfg-options model.in_channels="[1, 1, 1]"
```

```
Config (path: ./example.py): {'model': {'type': 'CustomModel', 'in_channels': [1, 1, 1]}, 'optimizer': {'type': 'SGD', 'lr': 0.01}}
```

- NOTE

  The standard procedure only supports modifying String, Integer, Floating Point, Boolean, None, List, and Tuple fields from the command line. For the elements of list and tuple instance, each of them must be one of the above seven types.

- NOTE

  The behavior of `DictAction` is similar with `"extend"`. It stores a list, and extends each argument value to the list, like:

  ```
  python demo_train.py ./example.py --cfg-options optimizer.type="Adam" --cfg-options model.in_channels="[1, 1, 1]"
  ```

  ```
  Config (path: ./example.py): {'model': {'type': 'CustomModel', 'in_channels': [1, 1, 1]}, 'optimizer': {'type': 'Adam', 'lr': 0.01}}
  ```

Replace fields with environment variables

When a field is deeply nested, we need to add a long prefix at the command line to locate it. To alleviate this problem, MMEngine allows users to substitute fields in configuration with environment variables.

Before parsing the configuration file, the program will search all `{{$ENV_VAR:DEF_VAL}}` fields and substitute those sections with environment variables. Here, `ENV_VAR` is the name of the environment variable used to replace this section, `DEF_VAL` is the default value if `ENV_VAR` is not set.

When we want to modify the dataset path at the command line, we can take `replace_data_root.py` as an example:

```
dataset_type = 'CocoDataset'
data_root = '{{$DATASET:/data/coco/}}'
dataset=dict(ann_file= data_root + 'train.json')
```

If we run `demo_train.py` to parse this configuration file.

```
python demo_train.py replace_data_root.py
```

```
Config (path: replace_data_root.py): {'dataset_type': 'CocoDataset', 'data_root': '/data/coco/', 'dataset': {'ann_file': '/data
/coco/train.json'}}
```

Here, we don't set the environment variable `DATASET`. Thus, the program directly replaces `{{$DATASET:/data/coco/}}` with the default value `/data/coco/`. If we set `DATASET` at the command line:

```
DATASET=/new/dataset/path/ python demo_train.py replace_data_root.py
```

```
Config (path: replace_data_root.py): {'dataset_type': 'CocoDataset', 'data_root': '/new/dataset/path/', 'dataset': {'ann_file':
'/new/dataset/path/train.json'}}
```

The value of `data_root` has been substituted with the value of `DATASET` as `/new/dataset/path`.

It is noteworthy that both `--cfg-options` and `{{$ENV_VAR:DEF_VAL}}` allow users to modify fields in command line. But there is a small difference between those two methods. Environment variable substitution occurs before the configuration parsing. If the replaced field is also involved in other fields assignment, the environment variable substitution will also affect the other fields.

We take `demo_train.py` and `replace_data_root.py` for example. If we replace `data_root` by setting `--cfg-options data_root='/new/dataset/path'`:

```
python demo_train.py replace_data_root.py --cfg-options data_root='/new/dataset/path/'
```

```
Config (path: replace_data_root.py): {'dataset_type': 'CocoDataset', 'data_root': '/new/dataset/path/', 'dataset': {'ann_file': '/data
/coco/train.json'}}
```

As we can see, only `data_root` has been modified. `dataset.ann_file` is still the default value.

In contrast, if we replace `data_root` by setting `DATASET=/new/dataset/path`:

```
DATASET=/new/dataset/path/ python demo_train.py replace_data_root.py
```

```
Config (path: replace_data_root.py): {'dataset_type': 'CocoDataset', 'data_root': '/new/dataset/path/', 'dataset': {'ann_file':
'/new/dataset/path/train.json'}}
```

Both `data_root` and `dataset.ann_file` have been modified.

Environment variables can also be used to replace other types of fields. We can use `{{'$ENV_VAR:DEF_VAL'}}` or `{{"$ENV_VAR:DEF_VAL"}}` format to ensure the configuration file conforms to python syntax.

We can take `replace_num_classes.py` as an example:

```
model=dict(
    bbox_head=dict(
        num_classes={{'$NUM_CLASSES:80'}}))
```

If we run `demo_train.py` to parse this configuration file.

```
lasses.py
```

```
Config (path: replace_num_classes.py): {'model': {'bbox_head': {'num_classes': 80}}}
```

Let us set the environment variable `NUM_CLASSES`

```
NUM_CLASSES=20 python demo_train.py replace_num_classes.py
```

```
Config (path: replace_num_classes.py): {'model': {'bbox_head': {'num_classes': 20}}}
```

import the custom module

If we customize a module and register it into the corresponding registry, could we directly build it from the configuration file as the previous section does? The answer is "I don't know" since I'm not sure the registration process has been triggered. To solve this "unknown" case, `Config` provides the `custom_imports` function, to make sure your module could be registered as expected.

For example, we customize an optimizer:

```
from mmengine.registry import OPTIMIZERS

@OPTIMIZERS.register_module()
class CustomOptim:
    pass
```

A matched config file:

`my_module.py`

```
optimizer = dict(type='CustomOptim')
```

To make sure `CustomOptim` will be registered, we should set the `custom_imports` field like this:

`custom_imports.py`

```
custom_imports = dict(imports=['my_module'], allow_failed_imports=False)
optimizer = dict(type='CustomOptim')
```

And then, once the `custom_imports` can be loaded successfully, we can build the `CustomOptim` from the `custom_imports.py`.

```
cfg = Config.fromfile('custom_imports.py')

from mmengine.registry import OPTIMIZERS

custom_optim = OPTIMIZERS.build(cfg.optimizer)
print(custom_optim)
```

```
<my_module.CustomOptim object at 0x7f6983a87970>
```

Inherit configuration files across repository

It is annoying to copy a large number of configuration files when developing a new repository based on some existing repositories. To address this issue, `Config` support inherit configuration files from other repositories. For example, based on MMDetection, we want to develop a repository, we can use the MMDetection configuration file like this:

`cross_repo.py`

```
_base_ = [
    'mmdet::_base_/schedules/schedule_1x.py',
    'mmdet::_base_/datasets/coco_instance.py',
    'mmdet::_base_/default_runtime.py',
    'mmdet::_base_/models/faster_rcnn_r50_fpn.py',
]
```

```
cfg = Config.fromfile('cross_repo.py')
print(cfg.train_cfg)
```

```
{'type': 'EpochBasedTrainLoop', 'max_epochs': 12, 'val_interval': 1, '_scope_': 'mmdet'}
```

`Config` will parse `mmdet::` to find mmdet package and inherits the specified configuration file. Actually, as long as the `setup.py` of the repository(package) conforms to MMEngine Installation specification, `Config` can use `{package_name}::` to inherit the specific configuration file.

Get configuration files across repository

`Config` also provides `get_config` and `get_model` to get the configuration file and the trained model from the downstream repositories.

The usage of `get_config` and `get_model` are similar to the previous section:

An example of `get_config`:

```
from mmengine.hub import get_config

cfg = get_config(
    'mmdet::faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py', pretrained=True)
print(cfg.model_path)
```

```
https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_rcnn_r50_fpn_1x_coco/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth
```

An example of `get_model`:

```
from mmengine.hub import get_model

model = get_model(
    'mmdet::faster_rcnn/faster_rcnn_r50_fpn_1x_coco.py', pretrained=True)
print(type(model))
```

```
http loads checkpoint from path: https://download.openmmlab.com/mmdetection/v2.0/faster_rcnn/faster_rcnn_r50_fpn_1x_coco
/faster_rcnn_r50_fpn_1x_coco_20200130-047c8118.pth
<class 'mmdet.models.detectors.faster_rcnn.FasterRCNN'>
```

A Pure Python style Configuration File (Beta)

In the previous tutorial, we introduced how to use configuration files to build modules with registry and how to use `_base_` to inherit configuration files. These pure text style configuration files can satisfy most of our development needs and some module aliases can greatly simplify the configuration files (e.g. `ResNet` can refer to `mmcls.models.ResNet`). However, there are also some disadvantages:

1. In the configuration file, the `type` field is specified by a string, and IDE cannot directly jump to the corresponding class definition, which is not conducive to code reading and jumping.
2. The inheritance of configuration files is also specified by a string, and IDE cannot directly jump to the inherited file. When the inheritance structure of the configuration file is complex, it is not conducive to reading and jumping of the configuration file.
3. The inheritance rules are relatively implicit, and beginners find it difficult to understand how the configuration file merges variables with the same fields and derives special syntax such as `_delete_`, resulting in a higher learning cost.
4. It is easy for users to forget to register the module and cause `module not found` errors.
5. In the yet-to-be-mentioned cross-codebase inheritance, the introduction of the scope makes the inheritance rules of the configuration file more complicated, and beginners find it difficult to understand.

In summary, although pure text style configuration files can provide the same syntax rules for `python`, `json`, and `yaml` format configurations, when the configuration files become complex, pure text style configuration files will appear inadequate. Therefore, we provide a pure Python style configuration file, i.e., the `lazy import` mode, which can fully utilize Python's syntax rules to solve the above problems. At the same time, the pure Python style configuration file also supports exporting to `json` and `yaml` formats.

Basic Syntax

In the previous tutorial, we introduced module construction, inheritance, and export based on pure text style configuration files. This section will introduce pure Python style configuration files based on these three aspects.

Module Construction

We use a simple example to compare pure Python style and pure text style configuration files:

Pure Python style | Pure text style

```
# No need for registration
```

Pure Python style | Pure text style

```python
# Configuration file writing
from torch.optim import SGD


optimizer = dict(type=SGD, lr=0.1)
```

Pure Python style | Pure text style

```python
# The construction process is exactly the same
import torch.nn as nn
from mmengine.registry import OPTIMIZERS


cfg = Config.fromfile('optimizer.py')
model = nn.Conv2d(1, 1, 1)
cfg.optimizer.params = model.parameters()
optimizer = OPTIMIZERS.build(cfg.optimizer)
```

From the above example, we can see that the difference between pure Python style and pure text style configuration files is:

1. Pure Python style configuration files do not require module registration.
2. In pure Python style configuration files, the `type` field is no longer a string but directly refers to the module. Correspondingly, import syntax needs to be added in the configuration file.

It should be noted that the OpenMMLab series algorithm library still retains the registration process when adding modules. When users build their own projects based on MMEngine, if they use pure Python style configuration files, registration is not required. You may wonder that if you are not in an environment with torch installed, you cannot parse the sample configuration file. Can this configuration file still be called a configuration file? Don't worry, we will explain this part later.

Inheritance

The inheritance syntax of pure Python style configuration files is slightly different:

Pure Python style Inheritance | Pure text style Inheritance

```python
from mmengine.config import read_base


with read_base():
    from .optimizer import *
```

Pure Python style configuration files use import syntax to achieve inheritance. The advantage of doing this is that we can directly jump to the inherited configuration file for easy reading and jumping. The variable inheritance rule (add, delete, change, and search) is completely aligned with Python syntax. For example, if I want to modify the learning rate of the optimizer in the base configuration file:

```python
from mmengine.config import read_base


with read_base():
    from .optimizer import *

# optimizer is a variable defined in the base configuration file
optimizer.update(
    lr=0.01,
)
```

Of course, if you are already accustomed to the inheritance rules of pure text style configuration files, you can also use merge syntax to achieve the same inheritance rule as pure text style configuration files:

```python
from mmengine.config import read_base


with read_base():
    from .optimizer import *

# optimizer is a variable defined in the base configuration file
optimizer.merge(
    _delete_=True,
    lr=0.01,
    type='SGD'
)

# The equivalent Python style writing is as follows, completely consistent with Python's import rules
# optimizer = dict(
#     lr=0.01,
#     type='SGD'
# )
```

- NOTE

  It should be noted that the `update` method of the dictionary in pure Python style configuration files is slightly different from `dict.update`. Pure Python style update will recursively update the content in the dictionary, for example:

```python
x = dict(a=1, b=dict(c=2, d=3))

x.update(dict(b=dict(d=4)))
# Update rules in the configuration file:
# {a: 1, b: {c: 2, d: 4}}
# Update rules in the normal dict:
# {a: 1, b: {d: 4}}
```

  It can be seen that using the update method in the configuration file will recursively update the fields, rather than simply covering them.

Compared with pure text style configuration files, the inheritance rule of pure Python style configuration files is completely aligned with the import syntax of Python, which is easier to understand and supports jumping between configuration files. You may wonder since both inheritance and module imports use import syntax, why do we need an `with read_base()'` statement for inheriting configuration files? On the one hand, this can improve the readability of configuration files, making inherited configuration files more prominent. On the other hand, it is also restricted by the rules of lazy_import, which will be explained later.

Dump the Configuration File

The pure Python style configuration files can also be exported via the `dump` interface, and there is no difference in usage. However, the exported contents will be different:

Export in pure Python style | Export in pure text style

```python
optimizer = dict(type='torch.optim.SGD', lr=0.1)
```

Export in pure Python style | Export in pure text style

```yaml
optimizer:
    type: torch.optim.SGD
    lr: 0.1
```

Export in pure Python style | Export in pure text style

```json
{"optimizer": "torch.optim.SGD", "lr": 0.1}
```

As can be seen, the type field exported in pure Python style contains the full module information. The exported configuration file can also be directly loaded to construct an instance through the registry.

What is Lazy Import

You may find that pure Python style configuration files seem to organize configuration files using pure Python syntax. Then, I do not need configuration classes, and I could just import configuration files using Python syntax. If you have such a feeling, then it is worth celebrating because this is exactly the effect we want.

As mentioned earlier, parsing configuration files requires dependencies on third-party libraries referenced in the configuration files. This is actually a very unreasonable thing. For example, if I trained a model based on MMagic and wanted to deploy it with the onnxruntime backend of MMDeploy. Due to the lack of torch in the deployment environment, and torch is needed in the configuration file parsing process, this makes it inconvenient for me to directly use the configuration file of MMagic as the deployment configuration. To solve this problem, we introduced the concept of lazy_import.

It is a complex task to discuss the specific implementation of lazy_import, so here we only briefly introduce its function. The core idea of lazy_import is to delay the execution of the import statement in the configuration file until the configuration file is parsed, so that the dependency problem caused by the import statement in the configuration file can be avoided. During the configuration file parsing process, the equivalent code executed by the Python interpreter is as follows:

Original configuration file | Code actually executed by the python interpreter through the configuration class

```python
from torch.optim import SGD


optimizer = dict(type=SGD)
```

As an internal type of the `Config` module, the `LazyObject` cannot be accessed directly by users. When accessing the type field, it will undergo a series of conversions to convert `LazyObject` into the actual `torch.optim.SGD` type. In this way, parsing the configuration file will not trigger the import of third-party libraries, while users can still access the types of third-party libraries normally when using the configuration file.

To access the internal type of `LazyObject`, you can use the `Config.to_dict` interface:

```
cfg = Config.fromfile('optimizer.py').to_dict()
print(type(cfg['optimizer']['type']))
# mmengine.config.lazy.LazyObject
```

At this point, the type accessed is the `LazyObject` type.

However, we cannot adopt the lazy import strategy for the inheritance (import) of base files since we need the configuration file parsed to include the fields defined in the base configuration file, and we need to trigger the import really. Therefore, we have added a restriction on importing base files, which must be imported in the `with read_base'` context manager.

Limitations

1. Functions and classes cannot be defined in the configuration file.
2. The configuration file name must comply with the naming convention of Python modules, which can only contain letters, numbers, and underscores, and cannot start with a number.
3. When importing variables from the base configuration file, such as `from ._base_.alpha import beta`, the `alpha` here must be the module (module) name, i.e., a Python file, rather than the package (package) name containing `__init__.py`.
4. Importing multiple variables simultaneously in an absolute import statement, such as `import torch, numpy, os`, is not supported. Multiple import statements need to be used instead, such as `import torch; import numpy; import os`.

Migration Guide

To migrate from a pure text style configuration file to a pure Python style configuration file, the following rules must be followed:

1. Replace the string type with the specific class:
   - If the code does not depend on the type field being a string, and no special processing is done on the type field, the string type of the type field can be replaced with the specific class, and the class should be imported at the beginning of the configuration file.
   - If the code depends on the type field being a string, the code needs to be modified, or the original string format of the type should be retained.
2. Rename the configuration file. The configuration file name must comply with the naming convention of Python modules, which can only contain letters, numbers, and underscores, and cannot start with a number.
3. Remove scope-related configurations. Pure Python style configuration files no longer need to use scope to get modules across libraries, and modules can be directly imported. For compatibility reasons, we still set the `default_scope` parameter of the Runner to `mmengine`, and users need to manually set it to `None`.
4. For modules that have aliases in the registry, replace their aliases with their corresponding real modules. The following is a table of commonly used alias replacements:

Module	Alias	Notes
nearest	**torch.nn.modules.upsampling.Upsample**	**When replacing 'type' with 'Upsample', the 'mode' parameter needs to be specified as 'nearest'.**
bilinear	**torch.nn.modules.upsampling.Upsample**	**When replacing 'type' with 'Upsample', the 'mode' parameter needs to be specified as 'bilinear'.**
Clip	**mmcv.cnn.bricks.activation.Clamp**	**None**
Conv	**mmcv.cnn.bricks.wrappers.Conv2d**	**None**
BN	**torch.nn.modules.batchnorm.BatchNorm2d**	**None**
BN1d	**torch.nn.modules.batchnorm.BatchNorm1d**	**None**
BN2d	**torch.nn.modules.batchnorm.BatchNorm2d**	**None**
BN3d	**torch.nn.modules.batchnorm.BatchNorm3d**	**None**
SyncBN	**torch.nn.SyncBatchNorm**	**None**
GN	**torch.nn.modules.normalization.GroupNorm**	**None**
LN	**torch.nn.modules.normalization.LayerNorm**	**None**
IN	**torch.nn.modules.instancenorm.InstanceNorm2d**	**None**
IN1d	**torch.nn.modules.instancenorm.InstanceNorm1d**	**None**
IN2d	**torch.nn.modules.instancenorm.InstanceNorm2d**	**None**
IN3d	**torch.nn.modules.instancenorm.InstanceNorm3d**	**None**
zero	**torch.nn.modules.padding.ZeroPad2d**	**None**
reflect	**torch.nn.modules.padding.ReflectionPad2d**	**None**
replicate	**torch.nn.modules.padding.ReplicationPad2d**	**None**
ConvWS	**mmcv.cnn.bricks.conv_ws.ConvWS2d**	**None**
ConvAWS	**mmcv.cnn.bricks.conv_ws.ConvAWS2d**	**None**
[illegible]	[illegible]	**None**

Module	Alias	Notes
deconv	mmcv.cnn.bricks.wrappers.ConvTranspose2d	None
deconv3d	mmcv.cnn.bricks.wrappers.ConvTranspose3d	None
Constant	mmengine.model.weight_init.ConstantInit	None
Xavier	mmengine.model.weight_init.XavierInit	None
Normal	mmengine.model.weight_init.NormalInit	None
TruncNormal	mmengine.model.weight_init.TruncNormalInit	None
Uniform	mmengine.model.weight_init.UniformInit	None
Kaiming	mmengine.model.weight_init.KaimingInit	None
Caffe2Xavier	mmengine.model.weight_init.Caffe2XavierInit	None
Pretrained	mmengine.model.weight_init.PretrainedInit	None

‹ Previous Next ›