PyTorch 分布式(2) --- 数据加载之 DataLoader

0x01 前情回顾

关于数据加载，上回书我们说到了 DistributedSampler，本文接下来就进行 DataLoader的分析。

为了更好说明，我们首先给出上文的流水线图，本文会对这个图进行细化。

其次，我们再看看数据加载总体逻辑，具体如下图，简要说就是：

1. DataSet 把数据集数目发给DistributedSampler。
2. Sampler 按照某种规则生成数据indices并发送给DataLoader。
3. DataLoader 依据indices来从DataSet之中加载数据（其内部的DataLoaderIter对象负责协调单进程/多进程加载Dataset）。
4. DataLoader 把数据发给模型，进行训练。

```
+--------------------------+                     +------------+
|DistributedSampler        |                     |DataLoader  |
|                          |     2 indices       |            |
|    Some strategy         +-------------------> |            |
|                          |                     |            |
|-------------+-----------|                     |            |
              ^                                 |            |  4 data  +-------+
              |                                 |       -------------->+ train |
            1 | length                          |            |          +-------+
              |                                 |            |
+-------------+-----------+                     |            |
```

```
|DataSet                  |                      |            |
|        +---------+      |       3 Load         |            |
|        |  Data   +-------------------------->  |            |
|        +---------+      |                      |            |
|                         |                      |            |
+-------------------------+                      +------------+
```

接下来，我们就正式进入 DataLoader。

0x02 DataLoader

DataLoader的作用是：结合Dataset和Sampler之后，在数据集上提供了一个迭代器。

可以这么理解：

DataSet 是原始数据，Sampler 提供了如何切分数据的策略（或者说是提供了切分数据的维度），DataLoader就是依据策略来具体打工干活的，其中单进程加载就是一个人干活，多进程加载就是多拉几个人一起干活。

2.1 初始化

初始化的主要参数如下：

- dataset (Dataset) ： 所加载的数据集。
- batch_size (int, optional) ： 每个批次加载多少个样本。
- shuffle (bool, optional) ： 如果为 True，则每个epoch 都会再打乱数据。
- sampler (Sampler or Iterable, optional) ： 定义了如何从样本采样的策略。可以是任何实现了 `__len__` 的迭代器。
- batch_sampler (Sampler or Iterable, optional) ： 与`sampler`类似，但是每次返回一个批次的数据索引。
- num_workers (int, optional) ： 数据加载的子进程数目。如果是 0，表示从主进程加载数据。
- collate_fn (callable, optional)：从一个小批次（ mini-batch）张量中合并出一个样本列表。当从 map-style 数据集做批量加载时候使用。
- pin_memory (bool, optional) : 如果为true，则在返回张量之前把张量拷贝到CUDA固定内存之中。
- drop_last (bool, optional) ： 当数据集不能被均匀分割时，如果为true，丢掉最后一个不完整的批次。如果为False，那么最后一个批次的数据较小。
- timeout (numeric, optional): 如果是整数，则是worker收集批次数据的超时值。
- worker_init_fn (callable, optional)：如果非空，则会在seeding和数据加载之前被每个子进程调用，以Iworker id (`[0, num_workers - 1]`)作为输入参数。
- generator (torch.Generator, optional)：如果非空，则被RandomSampler 用来产生随机索引，也被多进程用来产生 `base_seed` 。
- prefetch_factor (int, optional, keyword-only arg)：每个 worker 提前加载 的 sample 数量。
- persistent_workers (bool, optional)：如果为 `True`, 则在消费一次之后，data loader也 不会关掉worker进程。这允许worker`Dataset`实例维持活动状态。

具体初始化代码如下，主要就是各种设置，为了更好的说明，去除了异常处理代码：

```
class DataLoader(Generic[T_co]):

    dataset: Dataset[T_co]
    batch_size: Optional[int]
    num_workers: int
    pin_memory: bool
    drop_last: bool
```

```python
    timeout: float
    sampler: Sampler
    prefetch_factor: int
    _iterator : Optional['_BaseDataLoaderIter']
    __initialized = False

    def __init__(self, dataset: Dataset[T_co], batch_size: Optional[int] = 1,
                 shuffle: bool = False, sampler: Optional[Sampler[int]] = None,
                 batch_sampler: Optional[Sampler[Sequence[int]]] = None,
                 num_workers: int = 0, collate_fn: Optional[_collate_fn_t] =
None,
                 pin_memory: bool = False, drop_last: bool = False,
                 timeout: float = 0, worker_init_fn: Optional[_worker_init_fn_t]
= None,
                 multiprocessing_context=None, generator=None,
                 *, prefetch_factor: int = 2,
                 persistent_workers: bool = False):
        torch._C._log_api_usage_once("python.data_loader")

        self.dataset = dataset
        self.num_workers = num_workers
        self.prefetch_factor = prefetch_factor
        self.pin_memory = pin_memory
        self.timeout = timeout
        self.worker_init_fn = worker_init_fn
        self.multiprocessing_context = multiprocessing_context

        if isinstance(dataset, IterableDataset):
            self._dataset_kind = _DatasetKind.Iterable
            # 省略异常处理
        else:
            self._dataset_kind = _DatasetKind.Map

        if batch_sampler is not None:
            # auto_collation with custom batch_sampler
            # 省略异常处理
            batch_size = None
            drop_last = False
        elif batch_size is None:
            # no auto_collation
            if drop_last:
                raise ValueError('batch_size=None option disables auto-batching
'
                                 'and is mutually exclusive with drop_last')

        if sampler is None:  # give default samplers
            if self._dataset_kind == _DatasetKind.Iterable:
                # See NOTE [ Custom Samplers and IterableDataset ]
                sampler = _InfiniteConstantSampler()
            else:  # map-style
                if shuffle:
                    sampler = RandomSampler(dataset, generator=generator)
                else:
                    sampler = SequentialSampler(dataset)

        if batch_size is not None and batch_sampler is None:
            # auto_collation without custom batch_sampler
            batch_sampler = BatchSampler(sampler, batch_size, drop_last)
```

```
        self.batch_size = batch_size
        self.drop_last = drop_last
        self.sampler = sampler
        self.batch_sampler = batch_sampler
        self.generator = generator

        if collate_fn is None:
            if self._auto_collation:
                collate_fn = _utils.collate.default_collate
            else:
                collate_fn = _utils.collate.default_convert

        self.collate_fn = collate_fn
        self.persistent_workers = persistent_workers
        self.__initialized = True
        self._IterableDataset_len_called = None
        self._iterator = None
        self.check_worker_number_rationality()
```

2.2 关键函数

这里关键函数之一就是_index_sampler，用来让迭代器调用sampler，我们接下来就会讲到

```
    @property
    def _index_sampler(self):
        # The actual sampler used for generating indices for `_DatasetFetcher`
        # (see _utils/fetch.py) to read data at each time. This would be
        # `.batch_sampler` if in auto-collation mode, and `.sampler` otherwise.
        # We can't change `.sampler` and `.batch_sampler` attributes for BC
        # reasons.
        if self._auto_collation:
            return self.batch_sampler
        else:
            return self.sampler
```

2.3 单进程加载

单进程模式下，Data Loader会在计算进程内加载数据，所以加载过程中可能会阻塞计算。

for 语句会调用enumerate 会返回一个迭代器，以此来遍历数据集。在eumerate之中，dataloader 的 `__next__(self)` 方法会被调用，逐一获取下一个对象，从而遍历数据集。

```
    cuda0 = torch.device('cuda:0')  # CUDA GPU 0
    for i, x in enumerate(train_loader):
        x = x.to(cuda0)
```

2.3.1 区分生成

当多进程加载时候，在DataLoader声明周期之中，迭代器只被建立一次，这样worker可以重用迭代器。

在单进程加载时候，应该每次生成，以避免重置状态。

```python
    def __iter__(self) -> '_BaseDataLoaderIter':
        if self.persistent_workers and self.num_workers > 0: # 如果是多进程或者设置了持久化
            if self._iterator is None: # 如果没有，才会新生成
                self._iterator = self._get_iterator()
            else:
                self._iterator._reset(self)
            return self._iterator
        else: # 单进程
            return self._get_iterator() # 每次都直接生成新的
```

具体会依据是否是多进程来区别生成。

```python
    def _get_iterator(self) -> '_BaseDataLoaderIter':
        if self.num_workers == 0:
            return _SingleProcessDataLoaderIter(self)
        else:
            self.check_worker_number_rationality()
            return _MultiProcessingDataLoaderIter(self)
```

2.3.2 迭代器基类

_BaseDataLoaderIter 是迭代器基类，我们挑选关键函数看看。

这里关键成员变量就是：

- _index_sampler：这里设置了loader 的 sampler，所以迭代器可以据此获取采样策略。
- _sampler_iter：得到 sampler 的迭代器。

```python
class _BaseDataLoaderIter(object):
    def __init__(self, loader: DataLoader) -> None:
        # 初始化参数
        self._dataset = loader.dataset
        self._dataset_kind = loader._dataset_kind
        self._IterableDataset_len_called = loader._IterableDataset_len_called
        self._auto_collation = loader._auto_collation
        self._drop_last = loader.drop_last
        self._index_sampler = loader._index_sampler # 得到采样策略
        self._num_workers = loader.num_workers
        self._prefetch_factor = loader.prefetch_factor
        self._pin_memory = loader.pin_memory and torch.cuda.is_available()
        self._timeout = loader.timeout
        self._collate_fn = loader.collate_fn
        self._sampler_iter = iter(self._index_sampler) # 得到sampler的迭代器
        self._base_seed = torch.empty((), dtype=torch.int64).random_(generator=loader.generator).item()
        self._persistent_workers = loader.persistent_workers
        self._num_yielded = 0
        self._profile_name = "enumerate(DataLoader)#{}.__next__".format(self.__class__.__name__)


    def __next__(self) -> Any:
        with torch.autograd.profiler.record_function(self._profile_name):
            if self._sampler_iter is None:
                self._reset()
```

```
            data = self._next_data() # 获取数据
            self._num_yielded += 1
            if self._dataset_kind == _DatasetKind.Iterable and \
                    self._IterableDataset_len_called is not None and \
                    self._num_yielded > self._IterableDataset_len_called:
                    # 忽略错误提示处理
                warnings.warn(warn_msg)
            return data
```

2.3.3 单进程迭代器

`_SingleProcessDataLoaderIter` 继承了 `_BaseDataLoaderIter`，可以看到，其增加了 `_dataset_fetcher`，在构造时候传入了 `_collate_fn` 等各种参数。

回忆下，`__next__` 会调用 `self._next_data()` 获取数据，而在这里，`_next_data` 就会：

- 使用 `self._next_index()`，其又会使用 `_sampler_iter`（采样器的迭代器）来获取indices 。
- 使用 `self._dataset_fetcher.fetch(index)` 来依据indices获取数据。

```
class _SingleProcessDataLoaderIter(_BaseDataLoaderIter):
    def __init__(self, loader):
        super(_SingleProcessDataLoaderIter, self).__init__(loader)
        assert self._timeout == 0
        assert self._num_workers == 0

        # 获取样本方法
        self._dataset_fetcher = _DatasetKind.create_fetcher(
            self._dataset_kind, self._dataset, self._auto_collation,
self._collate_fn, self._drop_last)

    def _next_data(self):
        index = self._next_index()  # may raise StopIteration
        # 获取样本
        data = self._dataset_fetcher.fetch(index)  # may raise StopIteration
        if self._pin_memory:
            data = _utils.pin_memory.pin_memory(data)
        return data

    def _next_index(self): # 得到indices
        return next(self._sampler_iter)  # may raise StopIteration
```

2.3.4 获取样本

我们接下来看看如何获取样本。就是通过索引传入 fetcher，从而获取想要的样本。

fetcher生成如下，这是在_SingleProcessDataLoaderIter初始化时候生成的：

```
class _DatasetKind(object):
    Map = 0
    Iterable = 1

    @staticmethod
    def create_fetcher(kind, dataset, auto_collation, collate_fn, drop_last):
        if kind == _DatasetKind.Map:
            return _utils.fetch._MapDatasetFetcher(dataset, auto_collation,
collate_fn, drop_last)
        else:
            return _utils.fetch._IterableDatasetFetcher(dataset, auto_collation,
collate_fn, drop_last)
```

对于Map-style，就使用 _MapDatasetFetcher 处理，就是使用 possibly_batched_index 从数据集之中提取数据，possibly_batched_index 是key。

如果有batch sampler，就使用 batch sampler。

如果需要从一个小批次（ mini-batch）张量中合并出一个样本列表。就使用 collate_fn后处理。

```
class _MapDatasetFetcher(_BaseDatasetFetcher):
    def __init__(self, dataset, auto_collation, collate_fn, drop_last):
        super(_MapDatasetFetcher, self).__init__(dataset, auto_collation,
collate_fn, drop_last)

    def fetch(self, possibly_batched_index):
        if self.auto_collation:
            # 如果配置了batch_sampler，_auto_collation就为True，
            # 那么就优先使用batch_sampler，此时fetcher中传入的就是一个batch的索引
            data = [self.dataset[idx] for idx in possibly_batched_index]
        else:
            data = self.dataset[possibly_batched_index]
        return self.collate_fn(data)
```

对于 Iterable-style，因为 `__init__` 方法内设置了 dataset 初始的迭代器，所以在fetch 方法内获取元素的时候，如果是常规 sampler，index 其实已经不起作用，直接从dataset迭代器获取。如果是batch sampler，则index有效果。

```
class _IterableDatasetFetcher(_BaseDatasetFetcher):
    def __init__(self, dataset, auto_collation, collate_fn, drop_last):
        super(_IterableDatasetFetcher, self).__init__(dataset, auto_collation,
collate_fn, drop_last)
        self.dataset_iter = iter(dataset)

    def fetch(self, possibly_batched_index):
        if self.auto_collation:
            # 即auto_collation为True，表示使用batch_sampler。
            # 则使用possibly_batched_index，获取1个batch大小的样本
            data = []
            for _ in possibly_batched_index:
                try:
                    data.append(next(self.dataset_iter))
                except StopIteration:
                    break
            if len(data) == 0 or (self.drop_last and len(data) <
len(possibly_batched_index)):
```

```
                raise StopIteration
        else:
            # sampler则直接往后遍历，提取1个样本
            data = next(self.dataset_iter)
        return self.collate_fn(data)
```

此时总逻辑如下：

```
      +--------------------------+              +-------------------------------+
      | DataLoader               |              | _SingleProcessDataLoaderIter  |
      |                          |              |                               |
      |                          |              |               __next__        |
+---------------+ Sampler        |              |                               |
|     |                          |              |             _next_data +---------
---+
|     |            Dataset       |              |                               |
  |
|     |                          |              |             _next_index       |
  |
|     |           __iter__       |              |                               |
  |
|     |                          |              |            _index_sampler     |
  |
|     |       _get_iterator  +--------------> |                    +          |
  |
|     |                          |              |                    |          |
  |
|     +--------------------------+              +-------------------------------+
  |
|                                                                    |
   |
|                                                                    |
   |
|                                                                    |
   |
|                             +-----------------------------+        |
   |
|                             |Sampler                      |        |
   |
+-------------------------> |                             | <------+
   |
                              |                             |
  |
                              |                             |
  |
                              |                             |
  |
                              +-----------------------------+
  |
  |
  |
                              +-----------------------------+
  |
                              |_BaseDatasetFetcher          |
  |
```

```
                            |                            |
  |
                            |          dataset           |
  |
                            |                            |  <-------------------
---+
                            |          collate_fn        |
                            |                            |
                            +----------------------------+
```

动态流程如下：

```
  User              DataLoader    _SingleProcessDataLoaderIter _DatasetKind
Sampler

    +                   +                    +                        +
  +
    |                   |                    |                        |
  |
    |         1         |                    |                        |
  |
 enumerate--------> __iter__                 |                        |
  |
    |                   +                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |          2         v          3             v
  |
    |             _get_iterator--------> __init__  +----------> create_fetcher
  |
    |         4         |                    +                        +
  |
    | <-----------------+                    |                        |
  |
    |      iterator     |                    |                        |
  |
    |                   |          5         |                        |
  |
for loop +-------------------------------> __next__                   |
  |
    |                   |                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |                 _next_data                  |
  |
    |                   |                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |                    |        6  next         |
  |
```

```
    |                   |                    _next_index  +-------------------------
> |
    |                   |                    |                        |
  |
    |                   |                    |  <-------------------------------
--+
    |                   |                    |           7  index     |
  |
    |                   |                    |                        |
  |
    |                   |                    |                        |
  |
    |                   |                    |        8 fetch(index)  |
  |
    |                   |                    | +--------------------> |
  |
    |                   |                    |                        |
  |
    |                   |                    |  <---------------------+
  |
    |                   |                    |         9  data        |
  |
    |  <-------------------------------------+                        |
  |
    |   10  data        |                    |                        |
  |
    |                   |                    |                        |
  |
    v                   v                    v                        v
  v
```

2.4 多进程加载

为了加速，PyTorch提供了多进程下载，只要把将参数 `num_workers` 设置为正整数，系统就会相应生成多进程处理，在这种模式下，每个worker都是一个独立进程。

由上节我们可以知道，_SingleProcessDataLoaderIter 是单进程加载数据的核心，loader通过它来与sampler，dataset交互。在多进程中，这个核心对应的就是 _MultiProcessingDataLoaderIter。

```python
    def _get_iterator(self) -> '_BaseDataLoaderIter':
        if self.num_workers == 0:
            return _SingleProcessDataLoaderIter(self)
        else:
            self.check_worker_number_rationality()
            return _MultiProcessingDataLoaderIter(self)
```

我们接下来就从 _MultiProcessingDataLoaderIter 开始分析。

2.4.1 总体逻辑

_MultiProcessingDataLoaderIter 中的注释十分详尽，值得大家深读，而且给出了逻辑流程图如下，其基本流程是围绕着三个queue进行的：

- 主进程把需要获取的数据 index 放入index_queue，这是

  指定子进程需要获取哪些数据的队列

  。同时也给子进程传入结果队列，关于结果队列，有两个分支：

    - 如果设置了pin memory，则传入的是 worker_result_queue。
    - 否则传入 data_queue。
- 子进程从 index_queue 之中读取 index，进行数据读取，然后把读取数据的index放入 worker_result_queue，这是向主进程返回结果的队列。
- 主进程进行处理，这里有两个分支：
    - 如果设置了pin memory，则主进程的 pin_memory_thread 会从 worker_result_queue 读取数据index，依据这个index进行读取数据，进行处理，把结果放入 data_queue，这是处理结果的队列。
    - 如果不需要pin memory，则结果已经存在 data_queue 之中，不做新操作。

可以看到，每个进程的输入是一个队列index_queue ，输出也是一个队列worker_result_queue。主进程和子进程通过这2~3个 queue 联系了起来，从而达到解耦合和加速的作用。

```
    # NOTE [ Data Loader Multiprocessing Shutdown Logic ]
    #
    # Preliminary:
    #
    # Our data model looks like this (queues are indicated with curly brackets):
    #
    #               main process                              ||
    #                     |                                   ||
    #               {index_queue}                             ||
    #                     |                                   ||
    #              worker processes                           ||     DATA
    #                     |                                   ||
    #            {worker_result_queue}                        ||     FLOW
    #                     |                                   ||
    #      pin_memory_thread of main process                  ||   DIRECTION
    #                     |                                   ||
    #               {data_queue}                              ||
    #                     |                                   ||
    #                data output                              \/
    #
    # P.S. `worker_result_queue` and `pin_memory_thread` part may be omitted if
    #      `pin_memory=False`.
```

具体如下图所示，如果不需要 pin memory，则为：

```
                                              +-----------+
                indices  -------------+ indices | Worker    | Data
              +--------->+index queue +--------->+ Process   +------+
              |          |            |          |           |      |
              |          -------------+          +-----------+      |
              |                                                     |   +---------
---+
              |                                                     |   |
   |
+---------+   |                                                     +--->
   |
| Main    |   | indices  -------------+ indices +-----------+           |
   |
| Process +------------>+index queue +--------->+ Worker    | Data      | Data
Queue |
|         |   |          |            |          | Process   +---------->
   |
```

```
+----------+ |          -------------+         |           |          |
   |
             |                                 +-----------+      +--->
   |
             |                                                     |  +---------
---+
             |                                                     |
             | indices  -------------+ indices +-----------+       |
             +--------->+index queue +-------->+ Worker    | Data  |
                        |            |         | Process   +------+
                        -------------+         |           |
                                               +-----------+
```

当有pin memory时候，则是先进入 result queue，然后 pin_memory_thread 处理之后会转入到 data queue：

```
                                               +-----------+
               indices  -------------+ indices | Worker    | Data
             +--------->+index queue +-------->+ Process   +------+
             |          |            |         |           |       |
             |          -------------+         +-----------+       |
             |                                                     |  ----------
----+
             |                                                     |  |
    |
+---------+  |                                                    +--->
    |
| Main    |  | indices  -------------+ indices +-----------+          |
    |
| Process +------------->+index queue +-------->+ Worker    | Data      |
result_queue|
|         |  |          |            |         | Process   +---------->
    |
+---------+  |          -------------+         |           |          |
    |
             |                                 +-----------+      +--->
    |
             |                                                     |  ---------
+----+
             |                                                     |            |
             | indices  -------------+ indices +-----------+       |            |
             +--------->+index queue +-------->+ Worker    | Data  | +---------
+---------+
                        |            |         | Process   +------+  |
pin_memory_thread|
                        -------------+         |           |         |          |
       |
                                               +-----------+         |          |
       |
                                                                     |          |
       |
                                                                     +----------
--------+
                                                                                |
                                                                                |
                                                                                |
                                                                                v
```

```
                                                                                  +------
+-------+
                                                                                  | Data
Queue  |
                                                                                  |
      |
                                                                                  +------
------+
```

2.4.2 初始化

初始化函数如下，主要是：

- 配置，生成各种成员变量，配置各种queue。
- 启动各个子进程。
- 启动主进程中的pin_memory的线程。

主要成员变量为：

- `_index_queues`：这是一个queue 列表，列表的每一个元素是一个 queue，就是每个子进程的队列需要处理的数据index，每个子进程对应一个 queue。
- `_worker_result_queue`：子进程处理完的 (idx, data)。
- `data_queue`：经过主进程 pin_memory 线程处理之后的数据队列，如果不需要pin，则直接会使用 `_worker_result_queue`。
- `_worker_queue_idx_cycle` 用以找出下一个工作的worker。

具体代码如下：

```
class _MultiProcessingDataLoaderIter(_BaseDataLoaderIter):
    r"""Iterates once over the DataLoader's dataset, as specified by the
sampler"""

    def __init__(self, loader):
        super(_MultiProcessingDataLoaderIter, self).__init__(loader)

        assert self._num_workers > 0
        assert self._prefetch_factor > 0

        if loader.multiprocessing_context is None:
            multiprocessing_context = multiprocessing
        else:
            multiprocessing_context = loader.multiprocessing_context

        self._worker_init_fn = loader.worker_init_fn
        self._worker_queue_idx_cycle = itertools.cycle(range(self._num_workers))
        # No certainty which module multiprocessing_context is
        self._worker_result_queue = multiprocessing_context.Queue()  # 子进程输出，
读取完数据的index
        self._worker_pids_set = False
        self._shutdown = False
        self._workers_done_event = multiprocessing_context.Event()

        self._index_queues = [] # 子进程输入，需读取数据的index
        self._workers = []
        for i in range(self._num_workers):
            # No certainty which module multiprocessing_context is
```

```python
            index_queue = multiprocessing_context.Queue()  # type: ignore[var-annotated]
            # Need to `cancel_join_thread` here!
            # See sections (2) and (3b) above.
            index_queue.cancel_join_thread()
            w = multiprocessing_context.Process(
                target=_utils.worker._worker_loop, # worker进程主函数，把各种queue和函数传进去
                args=(self._dataset_kind, self._dataset, index_queue,
                      self._worker_result_queue, self._workers_done_event,
                      self._auto_collation, self._collate_fn, self._drop_last,
                      self._base_seed, self._worker_init_fn, i,
self._num_workers,
                      self._persistent_workers))
            w.daemon = True
            w.start()
            self._index_queues.append(index_queue) # 把这个worker对应的index_queue放到主进程这里存起来，以后就可以交互了
            self._workers.append(w)

        if self._pin_memory:
            self._pin_memory_thread_done_event = threading.Event()

            # Queue is not type-annotated
            self._data_queue = queue.Queue()  # pin 处理之后的数据结果
            pin_memory_thread = threading.Thread(
                target=_utils.pin_memory._pin_memory_loop,
                args=(self._worker_result_queue, self._data_queue,
                      torch.cuda.current_device(),
                      self._pin_memory_thread_done_event))
            pin_memory_thread.daemon = True
            pin_memory_thread.start()
            # Similar to workers (see comment above), we only register
            # pin_memory_thread once it is started.
            self._pin_memory_thread = pin_memory_thread
        else:
            self._data_queue = self._worker_result_queue # 如果不需要pin，则直接使用_worker_result_queue

        # .pid can be None only before process is spawned (not the case, so ignore)
        _utils.signal_handling._set_worker_pids(id(self), tuple(w.pid for w in self._workers))  # type: ignore[misc]
        _utils.signal_handling._set_SIGCHLD_handler()
        self._worker_pids_set = True

        self._reset(loader, first_iter=True) # 继续完善业务
```

2.4.3 业务重置

`__init__` 函数最后会调用 _reset 函数，这是进一步完善业务初始化，也用来重置环境。

上小节函数中，已经启动了worker子进程，但是没有分配任务，所以_reset函数会进行任务分配，预取。

_MultiProcessingDataLoaderIter有如下 flag 参数来协调各个 worker （包括各种queue）之间的工作：

- `_send_idx`: 发送索引，用来记录这次要放 index_queue 中 batch 的 idx
- `_rcvd_idx`: 接受索引，记录要从 data_queue 中取出的 batch 的 idx
- `_task_info`: 存储将要产生的 data 信息的 dict，key为 task idx（由 0 开始的整型索引），value 为 `(worker_id,)` 或 `(worker_id, data)`，分别对应数据 未取 和 已取 的情况
- `_tasks_outstanding`: 整型，代表已经准备好的 task/batch 的数量（可能有些正在准备中）
- `_send_idx`: 发送索引，记录下一次要放 index_queue 中 task batch 的 idx。
- `_rcvd_idx`: 接受索引，记录下一次要从 data_queue 中取出的 task batch 的 idx。`_send_idx` 和 `_rcvd_idx` 主要用来进行流量控制和确保接受索引有意义。
- `_task_info`: 存储将要产生的 data 信息的 dict，key为 task batch idx（由 0 开始的整型索引），value 为 `(worker_id,)` 或 `(worker_id, data)`，分别对应数据 未取 和 已取 的情况。`_task_info` 的作用是依据 task batch idx 获取对应的 worker id 和暂存乱序数据。
- `_tasks_outstanding`: 整型，正在准备的 task/batch 的数量，实际上就是进行一些确认工作，没有太实际的意义。

对于加载数据，每个 worker 一次产生一个 batch 的数据，返回 batch 数据前，会放入下一个批次要处理的数据下标，所以 reset 函数会把 `_send_idx`，`_rcvd_idx` 都恢复成0，这样下次迭代就可以重新处理。

在 reset 方法最后，有一个预取数据操作。我们会在后面结合乱序处理进行讲解。

```
    def _reset(self, loader, first_iter=False):
        super()._reset(loader, first_iter)
        self._send_idx = 0  # idx of the next task to be sent to workers
        self._rcvd_idx = 0  # idx of the next task to be returned in __next__
        # information about data not yet yielded, i.e., tasks w/ indices in
range [rcvd_idx, send_idx).
        # map: task idx => - (worker_id,)        if data isn't fetched
(outstanding)
        #                  \ (worker_id, data)   if data is already fetched
(out-of-order)
        self._task_info = {}
        self._tasks_outstanding = 0  # always equal to count(v for v in
task_info.values() if len(v) == 1)
        # A list of booleans representing whether each worker still has work to
        # do, i.e., not having exhausted its iterable dataset object. It always
        # contains all `True`s if not using an iterable-style dataset
        # (i.e., if kind != Iterable).
        # Not that this indicates that a worker still has work to do *for this
epoch*.
        # It does not mean that a worker is dead. In case of
`_persistent_workers`,
        # the worker will be reset to available in the next epoch.
        # 每个worker的状态
        self._workers_status = [True for i in range(self._num_workers)]
        # We resume the prefetching in case it was enabled
        if not first_iter:
            for idx in range(self._num_workers):
                self._index_queues[idx].put(_utils.worker._ResumeIteration())
            resume_iteration_cnt = self._num_workers
            while resume_iteration_cnt > 0:
                return_idx, return_data = self._get_data()
                if isinstance(return_idx, _utils.worker._ResumeIteration):
                    assert return_data is None
                    resume_iteration_cnt -= 1
        # prime the prefetch loop
```

```
        # 预取若干index，目的是为了配合后续的乱序处理。
        for _ in range(self._prefetch_factor * self._num_workers):
            self._try_put_index()
```

2.4.4 获取 index

_try_put_index 函数就是使用sampler获取下一批次的数据index。这里 _prefetch_factor 缺省值是 2，主要逻辑如下。

- 从sampler获取下一批次的index。
- 通过 _worker_queue_idx_cycle 找出下一个可用的工作worker，然后把index分给它。
- 并且调整主进程的信息。

```
    def _next_index(self): # 定义在基类 _BaseDataLoaderIter 之中，就是获取下一批index
        return next(self._sampler_iter)  # may raise StopIteration

    def _try_put_index(self):

        assert self._tasks_outstanding < self._prefetch_factor *
self._num_workers

        try:
            index = self._next_index() # 获取下一批index
        except StopIteration:
            return
        for _ in range(self._num_workers):  # find the next active worker, if
any
            worker_queue_idx = next(self._worker_queue_idx_cycle)
            if self._workers_status[worker_queue_idx]: # 如果已经工作，就继续找
                break
        else:
            # not found (i.e., didn't break)
            return

        # 以下是主进程进行相关记录
        # 给下一个工作worker放入 (任务index, 数据index), 就是给queue放入数据，所以worker
loop之中就立刻会从queue中得到index，从而开始获取数据。
        self._index_queues[worker_queue_idx].put((self._send_idx, index))
        # 记录 将要产生的 data 信息
        self._task_info[self._send_idx] = (worker_queue_idx,)
        # 正在处理的batch个数+1
        self._tasks_outstanding += 1
        # send_idx 记录从sample_iter中发送索引到index_queue的次数
        self._send_idx += 1 # 递增下一批发送的task index
```

2.4.5 worker主函数

_worker_loop 是 worker进程的主函数，主要逻辑如其注释所示：

```
    # [ worker processes ]
    #   While loader process is alive:
    #     Get from `index_queue`.
    #       If get anything else,
    #          Check `workers_done_event`.
    #            If set, continue to next iteration
    #                    i.e., keep getting until see the `None`, then exit.
```

```
    #           Otherwise, process data:
    #               If is fetching from an `IterableDataset` and the iterator
    #                   is exhausted, send an `_IterableDatasetStopIteration`
    #                   object to signal iteration end. The main process, upon
    #                   receiving such an object, will send `None` to this
    #                   worker and not use the corresponding `index_queue`
    #                   anymore.
    #      If timed out,
    #         No matter `workers_done_event` is set (still need to see `None`)
    #         or not, must continue to next iteration.
    #   (outside loop)
    #   If `workers_done_event` is set,  (this can be False with
`IterableDataset`)
    #     `data_queue.cancel_join_thread()`.  (Everything is ending here:
    #                                          main process won't read from it;
    #                                          other workers will also call
    #                                          `cancel_join_thread`.)
```

就是通过index_queue, data_queue与主进程交互。

- 从 index_queue 获取新的数据index;
- 如果没有设置本worker结束，就使用 fetcher获取数据。
- 然后把数据放入data_queue，并且通知主进程，这里需要注意，**data_queue是传入的参数，如果设置了pin memory，则传入的是 worker_result_queue, 否则传入 data_queue。**

```
def _worker_loop(dataset_kind, dataset, index_queue, data_queue, done_event,
                 auto_collation, collate_fn, drop_last, base_seed, init_fn,
worker_id,
                 num_workers, persistent_workers):
    # See NOTE [ Data Loader Multiprocessing Shutdown Logic ] for details on the
    # logic of this function.

    try:
        # Initialize C side signal handlers for SIGBUS and SIGSEGV. Python
signal
        # module's handlers are executed after Python returns from C low-level
        # handlers, likely when the same fatal signal had already happened
        # again.
        # https://docs.python.org/3/library/signal.html#execution-of-python-
signal-handlers
        signal_handling._set_worker_signal_handlers()

        torch.set_num_threads(1)
        seed = base_seed + worker_id
        random.seed(seed)
        torch.manual_seed(seed)
        if HAS_NUMPY:
            np_seed = _generate_state(base_seed, worker_id)
            import numpy as np
            np.random.seed(np_seed)

        global _worker_info
        _worker_info = WorkerInfo(id=worker_id, num_workers=num_workers,
                                  seed=seed, dataset=dataset)

        from torch.utils.data import _DatasetKind
```

```
        init_exception = None

        try:
            if init_fn is not None:
                init_fn(worker_id)

            fetcher = _DatasetKind.create_fetcher(dataset_kind, dataset,
auto_collation, collate_fn, drop_last)
        except Exception:
            init_exception = ExceptionWrapper(
                where="in DataLoader worker process {}".format(worker_id))

        iteration_end = False
        watchdog = ManagerWatchdog()

        while watchdog.is_alive(): # 等待在这里
            try:
                # _try_put_index 如果放入了数据index，这里就被激活，开始工作
                r = index_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
            except queue.Empty:
                continue
            if isinstance(r, _ResumeIteration):
                # Acknowledge the main process
                data_queue.put((r, None))
                iteration_end = False
                # Recreate the fetcher for worker-reuse policy
                fetcher = _DatasetKind.create_fetcher(
                    dataset_kind, dataset, auto_collation, collate_fn,
drop_last)
                continue
            elif r is None:
                # Received the final signal
                assert done_event.is_set() or iteration_end
                break
            elif done_event.is_set() or iteration_end:
                # `done_event` is set. But I haven't received the final signal
                # (None) yet. I will keep continuing until get it, and skip the
                # processing steps.
                continue
            idx, index = r
            data: Union[_IterableDatasetStopIteration, ExceptionWrapper]
            if init_exception is not None:
                data = init_exception
                init_exception = None
            else:
                try:
                    data = fetcher.fetch(index)
                except Exception as e:
                    # 省略处理代码

            data_queue.put((idx, data)) # 放入数据，通知主进程
            del data, idx, index, r  # save memory
    except KeyboardInterrupt:
        # Main process will raise KeyboardInterrupt anyways.
        pass
    if done_event.is_set():
        data_queue.cancel_join_thread()
        data_queue.close()
```

2.4.6 Pin memory thread

在主进程之中，如果设置了需要pin memory，主进程的 pin_memory_thread 会从 worker_result_queue 读取数据，进行处理（加速CPU和GPU的数据拷贝），把结果放入 data_queue。

```
    # [ pin_memory_thread ]
    #   # No need to check main thread. If this thread is alive, the main loader
    #   # thread must be alive, because this thread is set as daemonic.
    #   While `pin_memory_thread_done_event` is not set:
    #     Get from `index_queue`.
    #       If timed out, continue to get in the next iteration.
    #       Otherwise, process data.
    #       While `pin_memory_thread_done_event` is not set:
    #         Put processed data to `data_queue` (a `queue.Queue` with blocking
put)
    #         If timed out, continue to put in the next iteration.
    #         Otherwise, break, i.e., continuing to the out loop.
    #
    #   NOTE: we don't check the status of the main thread because
    #           1. if the process is killed by fatal signal, `pin_memory_thread`
    #              ends.
    #           2. in other cases, either the cleaning-up in __del__ or the
    #              automatic exit of daemonic thread will take care of it.
    #              This won't busy-wait either because `.get(timeout)` does not
    #              busy-wait.
```

具体代码如下：

```
def _pin_memory_loop(in_queue, out_queue, device_id, done_event):
    # This setting is thread local, and prevents the copy in pin_memory from
    # consuming all CPU cores.
    torch.set_num_threads(1)

    torch.cuda.set_device(device_id)

    # See NOTE [ Data Loader Multiprocessing Shutdown Logic ] for details on the
    # logic of this function.
    while not done_event.is_set():
        try:
            r = in_queue.get(timeout=MP_STATUS_CHECK_INTERVAL)
        except queue.Empty:
            continue
        idx, data = r
        if not done_event.is_set() and not isinstance(data, ExceptionWrapper):
            data = pin_memory(data)
            # 省略异常处理代码
            r = (idx, data)
        while not done_event.is_set():
            try:
                out_queue.put(r, timeout=MP_STATUS_CHECK_INTERVAL)
                break
            except queue.Full:
                continue
        del r  # save memory
```

```python
def pin_memory(data):
    if isinstance(data, torch.Tensor):
        return data.pin_memory()
    elif isinstance(data, string_classes):
        return data
    elif isinstance(data, collections.abc.Mapping):
        return {k: pin_memory(sample) for k, sample in data.items()}
    elif isinstance(data, tuple) and hasattr(data, '_fields'):  # namedtuple
        return type(data)(*(pin_memory(sample) for sample in data))
    elif isinstance(data, collections.abc.Sequence):
        return [pin_memory(sample) for sample in data]
    elif hasattr(data, "pin_memory"):
        return data.pin_memory()
    else:
        return data
```

2.4.7 用户获取data

现在数据已经加载完毕，我们接下来看用户如何从DataLoader之中获取数据。

这里有一个很关键的地方：如何保持在不同实验之中数据读取顺序的一致性。为了让多次实验之间可以比对，就需要尽量保证在这些实验中，每次读取数据的顺序都是一致的，这样才不会因为数据原因造成结果的误差。

打破顺序一致性的最大可能就是乱序数据。而造成乱序问题的原因就是：多进程读取，可能某个进程快，某个进程慢。比如，用户这次需要读取6-19，16-26，37-46。但是某一个worker慢，6-19不能即时返回，另一个worker 的 16-26 先返回了，于是就会造成乱序。

如何处理乱序数据？PyTorch的具体做法就是：DataLoader严格按照Sampler的顺序返回数据。如果某一个数据是乱序的，则会把它暂存起来，转而去获取下一个数据，见下面代码中 "store out-of-order samples" 注释处。等到应该返回时候（这个数据顺序到了）才返回。

但是其风险就是数据返回会比当前请求慢，比如应该获取 6，但是Data queue里面没有这个数据，只有 16，27，于是用户只能等待 6 加载完成。

解决慢的方法是：预取（prefetch）。就是在reset方法最后，提前提取若干index，让DataLoader提前去取，这虽然不能保证任意两次训练的数据返回顺序完全一致，但是可以最大限度保证。

具体代码如下，首先，回忆基类的 `__next__` 函数，可以看到其调用了 _next_data 获取数据。

```python
class _BaseDataLoaderIter(object):
    def __next__(self) -> Any:
        with torch.autograd.profiler.record_function(self._profile_name):
            if self._sampler_iter is None:
                self._reset()
            data = self._next_data() # 获取数据
            self._num_yielded += 1
            if self._dataset_kind == _DatasetKind.Iterable and \
                    self._IterableDataset_len_called is not None and \
                    self._num_yielded > self._IterableDataset_len_called:
                # 忽略错误提示处理
                warnings.warn(warn_msg)
            return data
```

所以，我们要看 `_MultiProcessingDataLoaderIter` 的 `_next_data`。

- 因为之前有预取了index，worker进程已经开始获取数据，所以主进程此时可以得到数据，如果没有数据，就继续while True等待。
- 如果获取成功，则使用 `_process_data` 设定下一次的indx，准备下一次迭代。
- 通过 `_task_info` 来记录乱序数据，如果暂时无法处理，就在这里保存。

```
    def _next_data(self):
        while True:
            # If the worker responsible for `self._rcvd_idx` has already ended
            # and was unable to fulfill this task (due to exhausting an
`IterableDataset`),
            # we try to advance `self._rcvd_idx` to find the next valid index.
            #
            # This part needs to run in the loop because both the
`self._get_data()`
            # call and `_IterableDatasetStopIteration` check below can mark
            # extra worker(s) as dead.

            # 找到待取idx
            while self._rcvd_idx < self._send_idx: # 如果 待取batch idx < 已取batch
idx
                info = self._task_info[self._rcvd_idx]
                worker_id = info[0]
                if len(info) == 2 or self._workers_status[worker_id]:  # has
data or is still active
                    break # 有数据或者正在工作，就跳出内部这个while
                del self._task_info[self._rcvd_idx]
                self._rcvd_idx += 1
            else:
                # no valid `self._rcvd_idx` is found (i.e., didn't break)
                if not self._persistent_workers:
                    self._shutdown_workers()
                raise StopIteration

            # Now `self._rcvd_idx` is the batch index we want to fetch

            # Check if the next sample has already been generated
            if len(self._task_info[self._rcvd_idx]) == 2:
                data = self._task_info.pop(self._rcvd_idx)[1]
                return self._process_data(data) # 设定下一次的indx，进行下一次迭代

            assert not self._shutdown and self._tasks_outstanding > 0
            idx, data = self._get_data() # 从 self._data_queue 中取数据
            self._tasks_outstanding -= 1 # 正在准备的batch个数需要减1

            if self._dataset_kind == _DatasetKind.Iterable:
                # Check for _IterableDatasetStopIteration
                if isinstance(data,
_utils.worker._IterableDatasetStopIteration):
                    if self._persistent_workers:
                        self._workers_status[data.worker_id] = False
                    else:
                        self._mark_worker_as_unavailable(data.worker_id)
                    self._try_put_index()
                    continue

            if idx != self._rcvd_idx: # 乱序数据
                # store out-of-order samples
```

```
                self._task_info[idx] += (data,)
            else:
                del self._task_info[idx] # 正常数据
                return self._process_data(data) # 设定下一次的indx，进行下一次迭代
```

其次，我们看看 `_get_data` 如何从 `self._data_queue` 中取数据。具体是使用 _try_get_data 来提取。

- 如果有超时配置，就按照超时读取。
- 如果设置了pin memory，则从pin 线程处理之后的数据读取。
- 否则循环读取worker处理的数据，直至获取到数据为止。

```
    def _get_data(self):
        # Fetches data from `self._data_queue`.
        #
        # We check workers' status every `MP_STATUS_CHECK_INTERVAL` seconds,
        # which we achieve by running
`self._try_get_data(timeout=MP_STATUS_CHECK_INTERVAL)`
        # in a loop. This is the only mechanism to detect worker failures for
        # Windows. For other platforms, a SIGCHLD handler is also used for
        # worker failure detection.
        #
        # If `pin_memory=True`, we also need check if `pin_memory_thread` had
        # died at timeouts.
        if self._timeout > 0: # 如果有超时配置，就按照超时读取
            success, data = self._try_get_data(self._timeout)
            if success:
                return data
            else:
                raise RuntimeError('DataLoader timed out after {}
seconds'.format(self._timeout))
        elif self._pin_memory: # 从pin 线程处理之后的数据读取
            while self._pin_memory_thread.is_alive():
                success, data = self._try_get_data()
                if success:
                    return data
            else:
                # while condition is false, i.e., pin_memory_thread died.
                raise RuntimeError('Pin memory thread exited unexpectedly')
            # In this case, `self._data_queue` is a `queue.Queue`,. But we don't
            # need to call `.task_done()` because we don't use `.join()`.
        else:
            while True:
                success, data = self._try_get_data() # 读取worker处理的数据
                if success:
                    return data
```

`_try_get_data` 就是从 `_data_queue` 读取。主进程和worker进程通过queue上的put, get进行通讯交互。

```
    def _try_get_data(self, timeout=_utils.MP_STATUS_CHECK_INTERVAL):
        # Tries to fetch data from `self._data_queue` once for a given timeout.
        # This can also be used as inner loop of fetching without timeout, with
        # the sender status as the loop condition.
        #
        # This raises a `RuntimeError` if any worker died expectedly. This error
```

```
        # can come from either the SIGCHLD handler in
`_utils/signal_handling.py`
        # (only for non-Windows platforms), or the manual check below on errors
        # and timeouts.
        #
        # Returns a 2-tuple:
        #   (bool: whether successfully get data, any: data if successful else
None)
        try:
            data = self._data_queue.get(timeout=timeout)
            return (True, data)
        except Exception as e:
            # At timeout and error, we manually check whether any worker has
            # failed. Note that this is the only mechanism for Windows to detect
            # worker failures.
            failed_workers = []
            for worker_id, w in enumerate(self._workers):
                if self._workers_status[worker_id] and not w.is_alive():
                    failed_workers.append(w)
                    self._mark_worker_as_unavailable(worker_id)
            # 省略异常处理代码
            import tempfile
            import errno
            try:
                # Raise an exception if we are this close to the FDs limit.
                # Apparently, trying to open only one file is not a sufficient
                # test.
                # See NOTE [ DataLoader on Linux and open files limit ]
                fds_limit_margin = 10
                fs = [tempfile.NamedTemporaryFile() for i in
range(fds_limit_margin)]
            except OSError as e:
                # 省略异常处理代码
            raise
```

设置下一次迭代是使用`_process_data`。

```
    def _process_data(self, data):
        self._rcvd_idx += 1
        self._try_put_index() # 设定下一次的indx，进行下一次迭代
        if isinstance(data, ExceptionWrapper):
            data.reraise()
        return data # 返回数据
```

2.4.8 小结

我们小结一下多进程逻辑。

总体逻辑如下：

- 主进程把需要获取的数据 index 放入index_queue。
- 子进程从 index_queue 之中读取 index，进行数据读取，然后把读取数据的index放入 worker_result_queue。
- 主进程的 pin_memory_thread 会从 worker_result_queue 读取数据index，依据这个index进行读取数据，进行处理，把结果放入 data_queue。

具体流程如下图：

1. 在 _MultiProcessingDataLoaderIter 的初始化函数

   ```
   __init__
   ```

   之中会进行初始化：

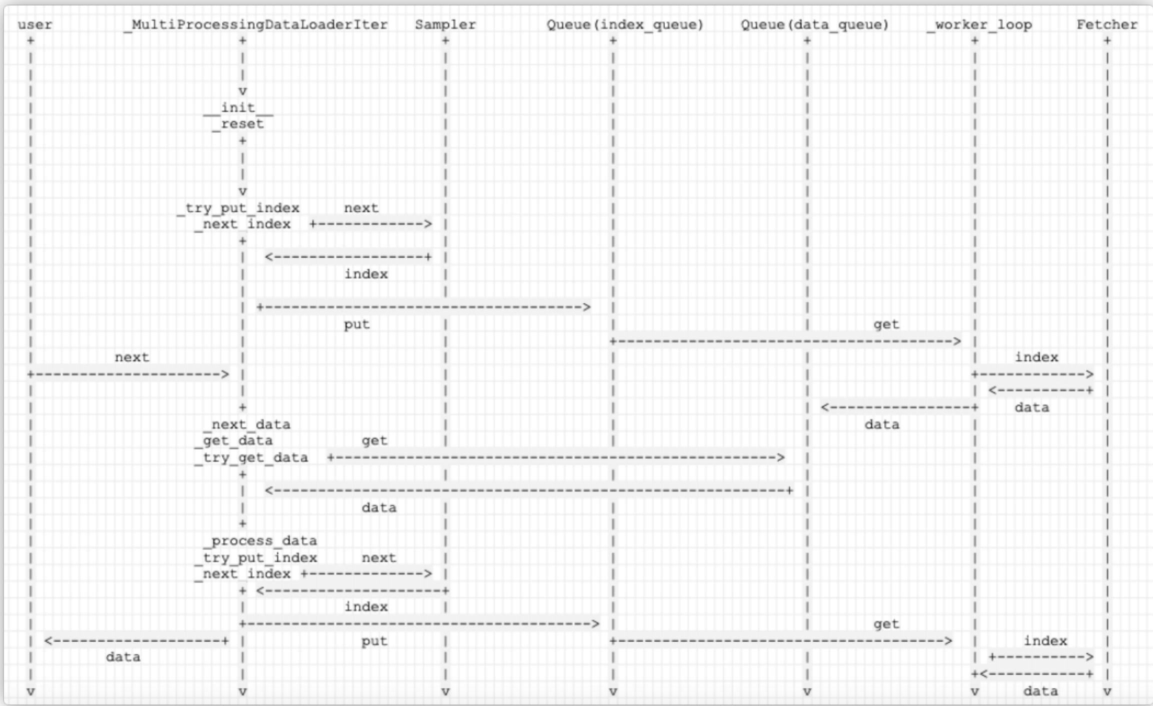
   - 配置，生成各种成员变量，配置各种queue。
   - 启动各个子进程。
   - 启动主进程中的pin_memory的线程。
   - 调用 _reset 函数，这是进一步完善业务初始化，也用来重置环境。上面已经启动了worker子进程，但是没有分配任务，所以reset函数会进行任务分配，**预取。**
2. 接下来是一个预取操作（在看下图中一定要留意）。
   - _try_put_index 函数就是使用sampler获取下一批次的数据index。这里 _prefetch_factor 缺省值是 2，主要逻辑如下。
     - 使用 _next_index 从sampler获取下一批次的index。
     - 通过 _worker_queue_idx_cycle 找出下一个可用的工作worker，然后把index分给它。
     - 并且调整主进程的信息。
   - 拿到index之后，回到主线程。这里会进行数据提取。就是通过index_queue, data_queue与主进程交互。
     - 从 index_queue 获取新的数据index；
     - 如果没有设置本worker结束，就使用 fetcher获取数据。
     - 然后把数据放入data_queue，并且通知主进程，这里需要注意，data_queue是传入的参数，如果设置了pin memory，则传入的是 worker_result_queue，否则传入 data_queue。
3. 当用户迭代时，调用了Loader基类的

   ```
   __next__
   ```

   函数 ，其调用 _next_data 从 DataLoader 之中获取数据。

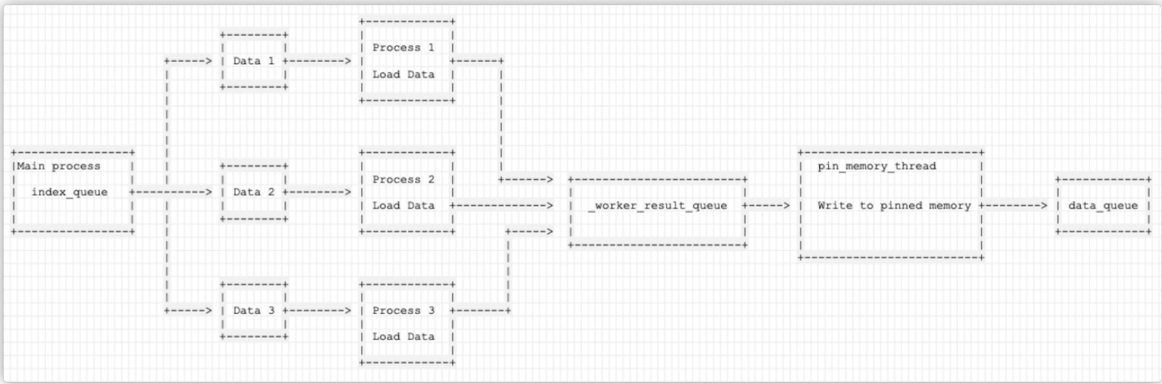
   - 使用 `_get_data` 如何从 `self._data_queue` 中取数据。
   - 使用`_process_data` 设置下一次迭代的 index，即使用 `_try_put_index`，`_next_index` 来进行下一轮设置。

具体如下图：



2.5 Pipleline

至此，我们把之前的pipeline图进一步细化，具体如下：



至此，PyTorch 分布式的数据加载部分分析完毕，下一篇我们回归到 Paracel 如何处理数据加载。