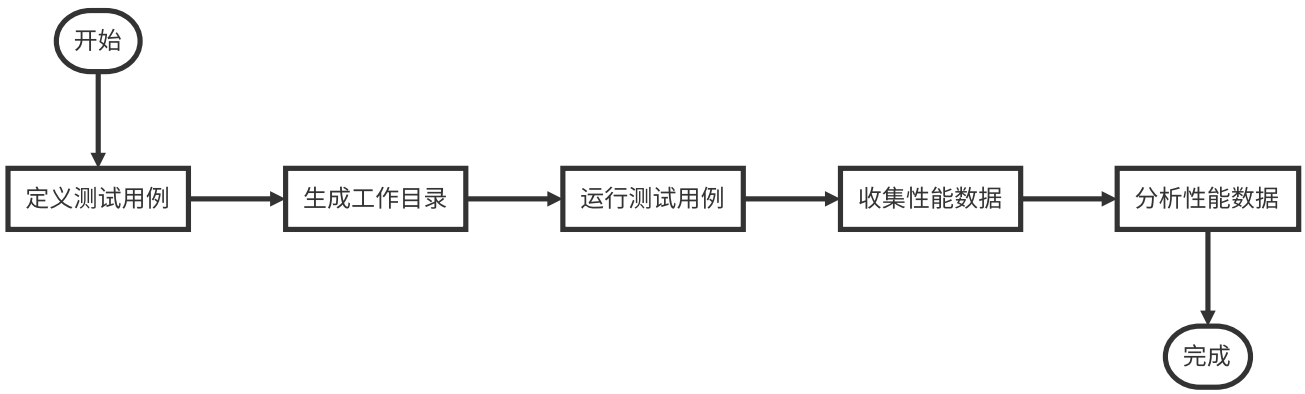
Bentoo 简明教程

Bentoo 是一套结构化性能测试工具，适用于并行算法、性能优化等研究中的性能测试。其特性包括：

1. 确保测试可重复
2. 自动抽取并归档测试结果
3. 适配天河、神威、曙光等多种超级计算机作业调度系统
4. 基本功能只依赖于 Python 标准库

本文对 Bentoo 的使用方法进行介绍。

性能测试基本流程



Bentoo 通过一系列工具，支持上述工作流程：

1. **bentoo-quickstart**: 快速定义测试用例
2. **bentoo-generator**: 自动生成工作目录
3. **bentoo-runner**: 自动运行测试用例，重新运行未完成用例
4. **bentoo-collector**: 自动解析性能数据并归档为 sqlite 数据库
5. **bentoo-analyzer**: 简单分析性能数据

结构化性能测试

Bentoo 的核心是结构化性能测试。结构化性能测试将性能测试定义为N个影响因素构成的N维测试空间。例如：对程序 `Euler` 进行强扩展性测试，研究其在纯进程并行与多线程并行下面的性能和扩展性对比，并确保性能数据在统计意义下的有效性。那么影响因素包括：

1. 并行模式 (mode)：包括纯进程并行 (mpi-core)、结点内纯线程并行 (mpi-node)、处理器内纯线程并行 (mpi-socket) 三种
2. 结点数 (nnodes)：计算结点数目或总核心数，从1到128结点，每结点2块8核处理器，强扩展用结点数倍增方式
3. 测试 ID (test_id)：同一设定的多次测试的测试编号，测试5次，取值为0-4

上述用例的所有影响因素按笛卡尔积的方式构成一个三维的测试空间：$[\text{mpi-core}, \text{mpi-node}, \text{mpi-socket}] \times [1, 2, 4, 8, 16, 32, 64, 128] \times [0, 1, 2, 3, 4]$，包括 120 个测试向量，对应 120 个测试用例。

在 Bentoo 中，**影响因素称为 “test factor”，测试向量称为 “test vector” ，测试用例称为 “test case”**。

定义测试用例

Bentoo 将一次结构化性能测试定义为 **“测试工程 （test project)”**。测试工程是一个包括 `TestProjectConfig.json` 文件的目录。一个典型的测试工程如下：

```
Euler
|-- bin
|   \-- main3d
|-- data
|   \-- Model.stl
|-- templates
|   \-- 3d.input.template
|-- TestProjectConfig.json
\-- make-case.py
```

`TestProjectConfig.json` 是一个 json 或 yaml 格式的数据文件，定义了测试工程的影响因素、测试向量、测试用例等描述信息。`make-case.py` 是用 python 编写的测试向量和测试用例生成器，使用 `3d.input.template` 等辅助文件，在指定的测试用例目录中生成独立运行该测试用例所需的全部文件，并向 Bentoo 返回测试用例的运行环境要求。`bin` 通常用于放置可执行文件，`data` 通常用于放置大型数据文件。

TestProjectConfig.json

上述 Euler 测试对应的 `TestProjectConfig.json` 如下：

```
{
  "version": 1,
```

```
  "project": {
    "name": "Euler",
    "description": "Euler strong scaling study w.r.t. proc-thread combinations",
    "test_factors": ["mode", "nnodes", "test_id"],
    "test_vector_generator": "custom",
    "test_case_generator": "custom",
    "data_files": ["bin", "data"]
  },
  "custom_vector_generator": {
    "import": "make_case.py",
    "func": "make_vectors",
    "args": {}
  },
  "custom_case_generator": {
    "import": "make_case.py",
    "func": "make_case",
    "args": {}
  }
}
```

上述文件包括三个关键字段：`project`、`custom_vector_generator`、和`custom_case_generator`，`version`选择当前测试工程定义文件的版本。

`project`定义测试工程的基本结构，包括：名称`name`、说明`description`、影响因素`test_factors`、测试向量生成器`test_vector_generator`、测试用例生成器`test_case_generator`和辅助文件列表`data_files`。其类型与取值如下：

- `name`: 字符串，测试工程名称
- `description`: 字符串，测试工程描述
- `test_factors`: 字符串列表，影响因素名称
- `test_vector_generator`: 字符串，测试向量生成器的类型，为`simple`、`cart_product`或`custom`
- `test_case_generator`: 字符串，测试用例生成器类型，为`template`或`custom`
- `data_files`: 字符串列表，辅助文件或目录路径列表，每一项为一个绝对路径或相对路径，相对路径代表相对于`TestProjectConfig.json`所在的目录的路径。

`<TYPE>_vector_generator`为与`test_vector_generator`匹配的测试向量生成器定义，`<TYPE>`与`test_vector_generator`取值一致。

`<TYPE>_case_generator`为与`test_case_generator`匹配的测试用例生成器定义，`<TYPE>`与`test_case_generator`取值一致。

自定义测试向量生成器

`custom`类型是最灵活的测试向量生成器类型。它执行一个 python 函数，接收其返回值作为测试向量定义。其在`TestProjectConfig.json`中定义为一个字典，字典项固定为：

- `import`: 字符串，python 函数所在的文件，将通过`import`载入
- `func`: 字符串，待执行的函数名，必须为`import`所指定文件中的函数
- `args`: 字典，表示传递给`func`的 **额外参数**, 将通过`**kwargs`传递给`func`

测试向量生成器所执行的函数原型为：

```python
def make_vectors(conf_root, test_factors, **kwargs):
    result = []
    # fill result with vectors of the same size as `test_factors`
    return result
```

`conf_root` 为用 **绝对路径** 表示的测试工程路径，`test_factors` 为 `project` 字段定义的 `test_factors` 列表，`kwargs` 为在 `custom_vector_generator` 中定义的额外参数。

上述 `Euler` 示例的测试向量生成器函数为：

```python
import itertools

def make_vectors(conf_root, test_factors, **kwargs):
    assert test_factors == ["mode", "nnodes", "test_id"]
    mode = ["mpi-core", "mpi-socket", "mpi-node"]
    nnodes = [1, 2, 4, 8 ,16, 32, 64, 128]
    test_id = range(5)
    return list(itertools.product(mode, nnodes, test_id))
```

自定义测试用例生成器

`custom` 类型是最灵活的测试用例生成器类型，也是最为常用的测试用例生成器类型。它执行一个 python 函数，接收其返回值作为测试用例定义，并由该函数准备测试用例的工作目录。其在 `TestProjectConfig.json` 中定义为一个字典，字典项固定为：

- `import`：字符串，python 函数所在的文件，将通过 `import` 载入
- `func`：字符串，待执行的函数名，必须为 `import` 所指定文件中的函数
- `args`：字典，表示传递给 `func` 的 **额外参数**，将通过 `**kwargs` 传递给 `func`

测试用例生成器所执行的函数原型为：

```python
def make_case(conf_root, output_root, case_path, test_vector, **kwargs):
    # expand test_vector
    # prepare case asserts in `case_path`
    # define test spec
    cmd = []
    envs = {}
    run = {}
    results = []
    validator = {}
    # calculate test spec and return
    return collections.OrderedDict(zip(["cmd", "envs", "run", "results", "validator"],
                                       [cmd, envs, run, results, validator]))
```

`conf_root` 和 `output_root` 为用 **绝对路径** 表示的测试工程路径和工作目录路径，`case_path` 为用 **绝对路径** 表示的测试用例工作目录，`test_vector` 为测试用例所对应的测试向量，用 `OrdredDict` 表示。`kwargs` 为在 `custom_case_generator` 中定义的额外参数。

测试用例生成函数需要完成如下约定功能：

1. 在 `case_path` 中生成测试用例 `test_vector` 对应的辅助文件，包括输入文件、数据文件等

2. 向 Bentoo 返回测试用例的 **执行描述** ，即执行方法的详细描述

执行描述为一个 Python 字典，包括五个字段：

1. `cmd`：字符串列表，表示串行执行用例所执行的命令，如 `["./main3d", "3d.input"]` 等
2. `envs`：字典（以字符串为键，任意类型为值），表示待设置的环境变量
3. `run`：由指定键构成的字典，表示执行测试用例的资源需求和分配方法。键值的类型和定义为：
    1. `nnodes`：整数，表示执行该测试用例所需计算结点数目
    2. `procs_per_node`：整数，表示执行该测试用例时每个结点的进程数目
    3. `tasks_per_proc`：整数，表示执行该测试用例时每个进程的线程数目
    4. `nprocs`：整数，表示执行该测试用例所需的总进程数目
4. `results`：字符串列表，表示测试用例输出结果文件，*必须为相对于* `case_path` *的相对路径*，标准输出和标准错误通过 `STDOUT` 和 `STDERR` 表示
5. `validator`：*可选*，由指定键构成的字典，表示检测测试用例是否成功完成的方法。键值的类型和定义为：
    1. `exists`：字符串列表，表示检测指定的文件是否存在，*必须为相对于* `case_path` *的相对路径*。仅当所有文件均存在才返回真值
    2. `contains`：以字符串为键和值的字典，`key: value` 表示在文件 `key` 中存在匹配正则表达式 `value` 的字符串，`key` *必须为相对于* `case_path` *的相对路径*。仅在所有文件都存在并且各个文件包含相应的字符串时返回真值

上述 `Euler` 示例的测试用例生成器函数为：

```
from collections import OrderedDict
import string
import shutil
import os

NX, NY, NZ = 600, 600, 600
NTHREADS = {
    "mpi-core": 1,
    "mpi-socket": 12,
    "mpi-node": 24
}
CPN = 24

def make_case(conf_root, output_root, case_path, test_vector, **kwargs):
    # Expand test vector
    mode, nnodes, test_id = test_vector.values()

    # Make input file by substitute templates
    content = file(os.path.join(conf_root, "templates", "3d.input.template")).read()
    output = os.path.join(case_path, "3d.input")
    var_values = dict(zip(["nx", "ny", "nz"], [NX, NY, NZ]))
    file(output, "w").write(string.Template(content).safe_substitute(var_values))
    # Link data file to case dir since main3d requires it in current working dir
    data_fn = os.path.join(conf_root, "data", "Model.stl")
    link_dir = os.path.join(case_path, "data")
    link_target = os.path.join(link_dir, "Model.stl")
    if os.path.exists(link_dir):
      shutil.rmtree(link_dir)
```

```
    os.makedirs(link_dir)
    os.symlink(os.path.relpath(data_fn, case_path), link_target)

    # Build case descriptions
    bin_path = os.path.join(output_root, "bin", "main3d")
    bin_path = os.path.relpath(bin_path, case_path)
    cmd = [bin_path, "3d.input"]
    envs = {
        "OMP_NUM_THREADS": NTHREADS[mode],
        "KMP_AFFINITY": "compact"
    }
    run = {
        "nnodes": nnodes,
        "procs_per_node": CPN / NTHREADS[mode],
        "tasks_per_proc": NTHREADS[mode],
        "nprocs": nnodes * CPN / NTHREADS[mode]
    }
    results = ["Euler.log"]
    validator = {
      "contains": {
        "Euler.log": "TIME STATISTICS"
      }
    }
    return OrderedDict(zip(["cmd", "envs", "run", "results", "validator"],
                           [cmd, envs, run, results, validator]))
```

内置测试向量生成器

在 `project` 中设置 `test_vector_generator` 为 `simple` 或 `cart_product` 时，可在 `TestProjectConfig.json` 中直接设定测试向量并通过 Bentoo 生成测试向量，无需编写 python 函数。

cart_product_vector_generator

当设置 `test_vector_generator` 为 `cart_product` 时，需在 `TestProjectConfig.json` 中定义 `cart_product_vector_generator` 字典。该字典仅包含一个键 `test_factor_values`，表示各个影响因素的取值。`test_factor_values` 是一个字典，以 `test_factors` 定义的影响因素名称为键，以列表为值，表示按集合的笛卡尔积方式生成所有测试向量。例如，上述 Euler 示例的测试向量生成器可定义为：

```
{
  "cart_product_vector_generator": {
    "test_factor_values": {
      "mode": ["mpi-core", "mpi-socket", "mpi-node"],
      "nnodes": [1, 2, 4, 8, 16, 32, 64, 128],
      "test_id": [0, 1, 2, 3, 4]
    }
  }
}
```

simple_vector_generator

当设置 `test_vector_generator` 为 `simple` 时，需在 `TestProjectConfig.json` 中定义 `simple_vector_generator` 字典。该字典仅包含一个键 `test_vectors`，表示所有的测试向量。`test_vectors` 是一个有列表组成的列表，每个列表项表示一个或一组测试向量。规则为：当该列表项的元素都是基本类型时，表示一个测试向量；当某个元素为列表类型时，表示由各个元素笛卡尔积张成的一组测试向量。例如，上述 Euler 示例的测试向量生成器可定义为：

```
{
  "simple_vector_generator": {
    "test_vectors": [
      ["mpi-core", [1, 2, 4, 8, 16, 32, 64, 128], [0, 1, 2, 3, 4]],
      ["mpi-socket", [1, 2, 4, 8, 16, 32, 64, 128], [0, 1, 2, 3, 4]],
      ["mpi-node", [1, 2, 4, 8, 16, 32, 64, 128], [0, 1, 2, 3, 4]]
    ]
  }
}
```

内置测试用例生成器

在 `project` 中设置 `test_case_generator` 为 `template` 时，可在 `TestProjectConfig.json` 中直接设定测试用例并通过 Bentoo 生成测试用例，无需编写 python 函数。

template_case_generator

当设置 `test_case_generator` 为 `template` 时，需在 `TestProjectConfig.json` 中定义 `template_case_generator` 字典。该字典包括如下键值对：

- `copy_files`: 字典，表示将指定文件复制到测试用例工作目录
- `link_files`: 字典，表示将指定文件符号链接到测试用例工作目录
- `inst_templates`: 字典，表示将指定文件中的进行模板替换，生成指定的文件并存放到测试用例工作目录
- `case_spec`: 字典，表示测试用例的定义，其结构与自定义测试用例生成函数返回值相同

`copy_files` 和 `link_files` 所定义的字典为如下格式：`key: value` 表示将相对与 `conf_root` 的文件或目录 `key` 复制为相对于 `case_path` 的文件或目录 `value`（`link_files` 为相对于 `output_root`，创建符号链接而非复制)。路径必须为相对路径。路径中可使用 `test_factors` 中定义的名称作为模板变量，这些变量将被替换为测试用例所对应测试向量的取值。

`inst_templates` 所定义的字典包含两个键值对：

- `templates`: 字典，由多个 `A: B` 键值对构成，表示模板替换指定文件 `A` 并生成指定文件 `B`。`A` 必须为相对于 `conf_root` 的相对路径，`B` 必须为相对于 `case_path` 的相对路径
- `variables`: 字典，由多个 `A: B` 键值对构成，表示木板变量 `A` 的值定义为表达式 `B`。`B` 是一个模板字符串，表示一个 python 表达式，可使用 `test_factors` 中定义的名称作为模板变量，这些变量将被替换为测试用例所对应测试向量的取值。Bentoo 将首先进行模板替换，然后对替换后的表达式求值，将求值结果作为 `A` 的取值

`case_spec` 的结构与自定义测试用例生成函数返回值相同，不同的是：所有元素的值均可以是一个模板字符串，表示一个 python 表达式，可使用 `test_factors` 中定义的名称作为模板变量，这些变量将被替换为测试用例所对应测试向量的取值。Bentoo 将首先进行模板替换，然后对替换后的表达式求值，将求值结果作为元素的值。在适合使用路径的地方，还可使用 `output_root` 作为模板变量，它将被替换为相对于 `case_path` 的相对路径。

上述 Euler 示例的 `template_case_generator` 可定义为：

```
{
  "template_case_generator": {
    "copy_files": {},
    "link_files": {
      "data/Model.stl": "data/Model.stl"
    },
    "inst_templates": {
      "templates": {
        "templates/3d.input.template": "3d.input"
      },
      "variables": {
        "nx": 600,
        "ny": 600,
        "nz": 600
      }
    },
    "case_spec": {
      "cmd": ["${output_root}/bin/main3d", "3d.input"],
      "envs": {
        "OMP_NUM_THREADS": "1 if '$mode' == 'mpi-core' else 12 if '$mode' == 'mpi-
socket' else 24",
        "KMP_AFFINITY": "compact"
      },
      "run": {
        "nnodes": "$nnodes",
        "procs_per_node": "24 if '$mode' == 'mpi-core' else 2 if '$mode' == 'mpi-
socket' else 1",
        "tasks_per_proc": "1 if '$mode' == 'mpi-core' else 12 if '$mode' == 'mpi-
socket' else 24",
        "nprocs": "$nnodes * 24 if '$mode' == 'mpi-core' else $nnodes * 2 if '$mode' ==
'mpi-socket' else $nnodes"
      },
      "results": ["Euler.log"],
      "validator": {
        "contains": {
          "Euler.log": "TIME STATISTICS"
        }
      }
    }
  }
}
```

使用 bentoo-quickstart

手工编写测试工程费时费力，使用 `bentoo-quickstart` 工具可生成一个测试工程的大部分文件，包括 `TestProjectConfig.json` 和 `make-case.py`。`bentoo-quickstart` 采用问答模式，在咨询若干问题后，生成所需的测试工程目录。需要注意的是：**生成的目录中的 `TestProjectConfig.json` 和 `make-case.py` 需要通过手工编辑，方能形成最终的工程文件**。

生成工作目录

在定义好测试工程后，可通过 `bentoo-generator` 生成工作目录。该工具用法如下 (可通过 `bentoo-generator --help` 获得):

```
usage: bentoo-generator.py [-h] [--link-files] conf_root output_root

positional arguments:
  conf_root     Project configuration directory
  output_root   Output directory

optional arguments:
  -h, --help    show this help message and exit
  --link-files  Sympolic link data files instead of copy
```

标准的测试工程所生成的工作目录是自包含的，即无需任何外界目录，即可执行每一个测试。此外，若测试工程中的路径全部采用相对路径，所生成的工作目录还是可移动或改名的。

运行测试用例

在生成测试工作目录后，可通过 `bentoo-runer` 执行性能测试。该工具用法如下 (可通过 `bentoo-runner --help` 获得):

```
usage: bentoo-runner.py [-h] [--skip-finished] [--rerun-failed] [-e EXCLUDE]
                        [-i INCLUDE]
                        [--launcher {yhrun,bsub,slurm,pbs,mpirun,auto}]
                        [-t TIMEOUT] [--sleep SLEEP] [--make-script]
                        [--dryrun] [--verbose] [--yhrun-p PARTITION]
                        [--yhrun-x NODELIST] [--yhrun-w NODELIST]
                        [--yhrun-yhbatch] [--yhrun-fix-glex {none,v0,v1}]
                        [--bsub-queue QUEUE] [--bsub-b] [--bsub-cgsp CGSP]
                        [--bsub-share_size SIZE] [--bsub-host_stack SIZE]
                        [--slurm-partition PARTITION] [--slurm-sbatch]
                        [--pbs-queue QUEUE] [--pbs-iface IFACE]
                        [--mpirun-hosts HOSTS] [--mpirun-ppn PPN]
                        project_root

bentoo-runner - Testcase runner bentoo-runner runs the test cases generated by
bentoo-generator on different high performance computing platforms. It
features test case filters, results validator, timeout etc. It supports slurm,
pbs, yhrun (tianhe), bsub (sunway), and plain mpirun at the moment. More
backends will be added overtime.

optional arguments:
  -h, --help            show this help message and exit

Global options:
  project_root          Root directory of the test project
  --skip-finished       Skip already finished cases
  --rerun-failed        Rerun failed jobs (using validator to determine)

Filter options:
  -e EXCLUDE, --exclude EXCLUDE
```

```
                        Excluded case paths, support shell wildcards
  -i INCLUDE, --include INCLUDE
                        Included case paths, support shell wildcards

Launcher options:
  --launcher {yhrun,bsub,slurm,pbs,mpirun,auto}
                        Job launcher (default: auto)
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout for each case, in minites
  --sleep SLEEP         Sleep specified seconds between jobs
  --make-script         Generate job script for each case
  --dryrun              Don't actually run cases
  --verbose             Be verbose (print jobs output currently)

yhrun options:
  --yhrun-p PARTITION, --yhrun-partition PARTITION
                        Select job partition to use
  --yhrun-x NODELIST    Exclude nodes from job allocation
  --yhrun-w NODELIST    Use only selected nodes
  --yhrun-yhbatch       Use yhbatch instead of yhrun
  --yhrun-fix-glex {none,v0,v1}
                        Fix GLEX settings (default: none)

bsub options:
  --bsub-queue QUEUE    Select job queue to use
  --bsub-b              Use large segment support
  --bsub-cgsp CGSP      Number of slave cores per core group
  --bsub-share_size SIZE
                        Share region size
  --bsub-host_stack SIZE
                        Host stack size

slurm options:
  --slurm-partition PARTITION
                        Select job partition to use
  --slurm-sbatch        Use sbatch instead of srun

pbs options:
  --pbs-queue QUEUE     Select job queue to use
  --pbs-iface IFACE     Network interface to use

mpirun options:
  --mpirun-hosts HOSTS  Comma seperated host list
  --mpirun-ppn PPN      Processes per node
```

`bentoo-runner` 通过不同的执行后端对接到不同的作业调度系统。一般情况下，执行命令如下:

```
bentoo-runner --launcher=<LAUNCHER> --make-script --verbose <OUTPUT_ROOT>
```

这里，`--make-script` 选项在每个测试用例目录中生成一个测试执行脚本 `run.sh`，可用于手工执行测试用例。该测试执行脚本处理了环境变量设置、作业系统调用等执行细节，最为常用。`--verbose` 选项用于输出作业的标准输出和标准错误信息。若使用 `--dryrun` 选项，测试用例将不会被调度到作业系统执行，但其他功能将正常执行，是非常常用的选项。

监控失败的测试用例

通过 `--rerun-failed` 可重新运行失败的测试用例。Bentoo 将使用测试用例的 `validator` 字段定义的检测器，检测测试用例是否完成，并重新执行未完成的测试用例。执行示例如：

```
bentoo-runner --launcher=mpirun --verbose --rerun-failed <OUTPUT_ROOT>
```

和 `--dryrun` 联合使用，可检测（但不执行）失败或未完成的用例：

```
bentoo-runner --launcher=mpirun --dryrun --rerun-failed <OUTPUT_ROOT>
```

选择执行测试用例

通过 `--include` 和 `--exclude` 选项可选择或排除相应的测试用例。这两个选项均可多次使用，其结果将被叠加。`--include` 是白名单机制，而 `--exclude` 为黑名单机制。其参数为一个用 shell 通配符表示的路径，表示 `case_path` 相对于 `output_root` 的路径。Bentoo 采用 `<test_factor_name>-<test_factor_value>/...` 方式命名 `case_path`。一个白名单的示例为：

```
bentoo-runner --launcher=mpirun --include mode-mpi-core/*/0 <OUTPUT_ROOT>
```

收集性能数据

在所有测试用例均成功完成后，可通过 `bentoo-collector` 收集性能数据。该工具用法如下 (可通过 `bentoo-collector --help` 获得)：

```
usage: bentoo-collector.py [-h] [-i CASE_PATH [CASE_PATH ...] | -e CASE_PATH
                           [CASE_PATH ...]]
                           [--use-result RESULT_ID [RESULT_ID ...]]
                           [-p {yaml,pipetable,jasmin,jasmin3,jasmin4,likwid,udc,dsv}]
                           [--use-table TABLE_ID [TABLE_ID ...]]
                           [--dsv-seperator CHAR]
                           [-d COLUMN_NAME [COLUMN_NAME ...] | -k COLUMN_NAME
                           [COLUMN_NAME ...]] [-s {sqlite3,pandas}]
                           [--pandas-format {xls,xlsx,csv}] [-a FILE]
                           project_root data_file

Collector - Test results collector

Collector scans a test project directory, parses all result files found and
saves all parsed results as a self-described data sheet in a file. One can then
use Analyser or other tools to investigate the resultant data sheet.

To use collector, simply following the argument document. To use the generated
data sheet, keep in mind that the data sheet is ralational database table alike
```

```
and the concrete format is backend specific. For sqlite3 backend, the result is
stored in a table named 'result'. The data sheet is designed to be easily
parsable by pandas, so the recommendation is to use pandas to investigate the
data.

positional arguments:
  project_root          Test project root directory
  data_file             Data file to save results

optional arguments:
  -h, --help            show this help message and exit

Scanner Arguments:
  -i CASE_PATH [CASE_PATH ...], --include CASE_PATH [CASE_PATH ...]
                        Include only matched cases (shell wildcards)
  -e CASE_PATH [CASE_PATH ...], --exclude CASE_PATH [CASE_PATH ...]
                        Excluded matched cases (shell wildcards)
  --use-result RESULT_ID [RESULT_ID ...]
                        Choose result files to use (as index)

Parser Arguments:
  -p {yaml,pipetable,jasmin,jasmin3,jasmin4,likwid,udc,dsv}, --parser
{yaml,pipetable,jasmin,jasmin3,jasmin4,likwid,udc,dsv}
                        Parser for raw result files (default: jasmin)
  --use-table TABLE_ID [TABLE_ID ...]
                        Choose which data table to use (as index)

dsv parser arguments:
  --dsv-seperator CHAR  regex seperator for dsv values (default: ',')

Aggregator Arguments:
  -d COLUMN_NAME [COLUMN_NAME ...], --drop-columns COLUMN_NAME [COLUMN_NAME ...]
                        Drop un-wanted table columns
  -k COLUMN_NAME [COLUMN_NAME ...], --keep-columns COLUMN_NAME [COLUMN_NAME ...]
                        Keep only speciied table columns

Serializer Arguments:
  -s {sqlite3,pandas}, --serializer {sqlite3,pandas}
                        Serializer to dump results (default: sqlite3)

pandas serializer arguments:
  --pandas-format {xls,xlsx,csv}
                        Output file format

Archiver Arguments:
  -a FILE, --archive FILE
                        Archive output to a zip file
```

`bentoo-collector` 将解析定义于 `results` 字段中的文件中的性能数据表格，并将所有的测试用例的性能数据合并为一张大表，存放于指定的数据库中。目前，默认的数据库后端 为 `sqlite`，不依赖于第三方 python 包。

`bentoo-collector` 的使用范例如下：

```
bentoo-collector -p jasmin4 --use-table -1 --use-result 0 <OUTPUT_ROOT> --archive
results.tar.gz data.sqlite
```

这里 `--use-result` 选项选择在 `results` 字段中定义的结果文件，为 python 语法接受的列表索引类型。`--use-table` 选项选择结果文件中的性能表格（一些测试输出多个性能表格）。`-p` 选项选择性能表格解析器。`--archive` 选项设置将原始数据文件打包为指定压缩包。

过滤测试用例

有时需要将部分测试用例的测试结果排除在性能数据之外，或仅解析指定的测试用例。此时，可通过 `--include` 或 `--exclude` 选项进行。该选项接受一系列字符串，每个字符串代表一个 `case_path` 相对于 `output_root` 的相对路径，可采用 shell 通配符。一个示例如下：

```
bentoo-collector --use-table -1 --use-result 0 -i mode-mpi-core/*/* <OUTPUT_ROOT>
data.sqlite
```

过滤性能数据

有时需要过滤掉性能表格中的部分数据，或仅保留指定的表格列。此时，可通过 `--keep-columns` 和 `--drop-columns` 选项进行。该选项接受一系列字符串，每个字符串代表一个表格列名称，可采用 shell 通配符。`--keep-columns` 为白名单机制，`--drop-columns` 为黑名单机制，两者而选一。一个示例如下：

```
bentoo-collector --use-table -1 --use-result 0 -d *_percent <OUTPUT_ROOT> data.sqlite
```

分析性能数据

收集到的性能数据以表的方式存放在 sqlite 数据库或 excel 文件中，可通过 pandas 等数据分析工具进行数据分析。Bentoo 提供了一个快速数据提取分析工具 `bentoo-analyzer`。其使用方法如下（可通过 `bentoo-analyzer --help` 获得：

```
usage: bentoo-analyser.py [-h] [-r {sqlite,pandas}] [-m MATCHES] [-c COLUMNS]
                          [-p PIVOT] [-s SAVE]
                          [--sqlite-glob-syntax {fnmatch,regex}]
                          [--pandas-backend {excel,sqlite3,auto}]
                          data_file

Analyser - Test project result analyser

Analyser provides command line interface to extract and display test result
data collected by Collector. It provided options to filter result, to choose
display table fields and to pivot resultant table. It provides a simple and
intuitive syntax.

To use the analyser, one invokes bentoo-analyser.py with -m for matcher/filter,
-c for fields selection and -p for pivoting. For example, one can display how
all timers of algorithms scales w.r.t. number of nodes when using 1 threads per
process using the following command line:
```

```
    bentoo-analyser.py result.sqlite -m nthreads=1         -m
timer_name~algs::Numerical*,algs::Copy*         -c timer_name,nnodes,max,summed -p
timer_name,nnodes

Analyser tries to provide a simple CLI interface of pandas for simple use
cases, namely tasks need to be done quick and often in command line. More
sphosticated analysis need to be done directly in python using pandas etc.

positional arguments:
  data_file             Database file

optional arguments:
  -h, --help            show this help message and exit
  -r {sqlite,pandas}, --reader {sqlite,pandas}
                        Database reader (default: sqlite)
  -m MATCHES, --matches MATCHES, --filter MATCHES
                        Value filter, name[~=]value
  -c COLUMNS, --columns COLUMNS
                        Columns to display, value or list of values
  -p PIVOT, --pivot PIVOT
                        Pivoting fields, 2 or 3 element list
  -s SAVE, --save SAVE  Save result to a CSV file

Sqlite Reader Options:
  --sqlite-glob-syntax {fnmatch,regex}
                        Globbing operator syntax (default: fnmatch)

Pandas Reader Options:
  --pandas-backend {excel,sqlite3,auto}
                        Pandas IO backend (default: auto)
```

`bentoo-analyzer` 将抽取性能数据库中的数据，并将其打印到标准输出。也可将抽取的性能数据保存到指定的 csv 文件中。使用 sqlite 后端时，不依赖于第三方 python 包。一个典型的调用如下：

```
bentoo-analyzer data.sqlite -r sqlite -m mode=mpi-core -m test_id=0 -c
timer_name,nnodes,summed
```

这里，`-m` 选项选择指定字段符合符合条件的数据行，`-c` 选项选择进入最终结果的数据列，得到的数据表格将输出到标准输出。可通过 `--save` 选项输出到指定文件。

`bentoo-analyzer` 仅提供基础的数据抽取功能。复杂的数据分析建议通过 pandas 等专业数据分析软件包进行。