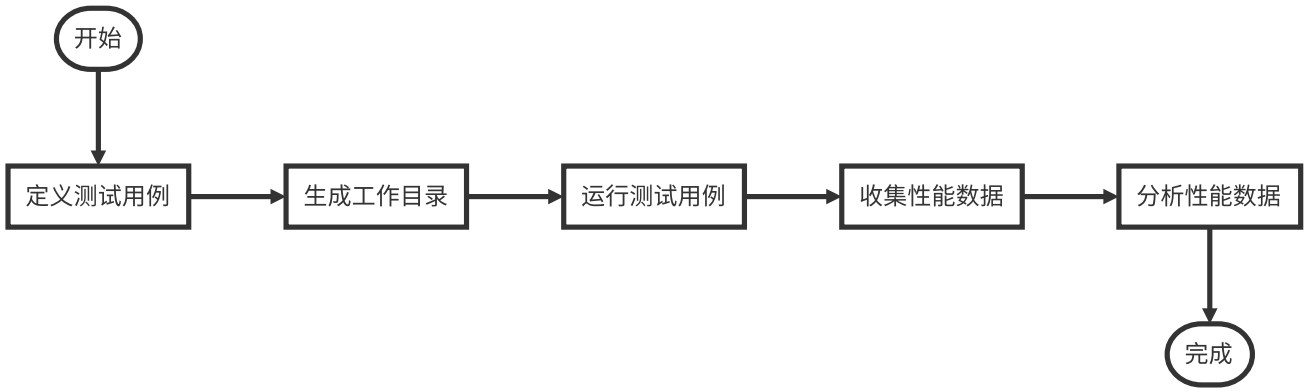
Bentoo 简明教程

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 的使用方法进行简单介绍。

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", "database"]
  },
  "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：

```
def make_vector(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_vector(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` 为相对于 `output_root` 的测试用例目录路径，`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。

[TODO] 测试用例函数功能与返回值说明

[TODO] 上述 `Euler` 示例的测试向量生成器函数为：


```python
import itertools

def make_vector(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))
```