评测脚本编写与内核制作

一、文件结构

```
cg_contest-grader (省略号表示不用在意，只需要关注列出来的文件)
  |...
  |--compiler (主要修改部分)
     |--backup-kernels (每次导出内核会在这里备份)
     |--coursegrader   (笔者本地测试时的尝试，可以忽略)
     |--docker         (构建镜像时会调用其中的Dockerfile)
        |--Dockerfile  (根据需要修改)
        |--pygrading-1.1.8.dev0-py3-none-any.whl (希冀官方提供的包，需要手动传入镜像)
     |--kernel         (主要修改部分)
        |...
        |--exception.py(应对测评异常的脚本)
        |--prework.py  (测评预处理脚本，兼加载测试数据)
        |--run.py      (测评运行脚本，兼分数计算)
        |--postwork.py (测评收尾脚本，兼生成学生可看到的测评结果)
     |--test           (测试数据存放在这里)
     |--README.md      (本文)
     |--kernel.pyz     (每次导出内核会在这里留下最新版本)
     |--manage.py      (希冀制作镜像和评测内核的主要程序)
     |--make.sh        (便捷制作镜像)
     |--test.json      (本地测试时需要修改此配置)

  |...
  |--README.md (希冀官方的README，包含脚本和内核的介绍)
  |--pygrading-1.1.8.dev0-py3-none-any.whl (希冀官方提供的包，需要手动传入镜像)
```

二、我们需要做什么 - 流程

TLDR: 我们需要（1）修改 `prework.py` `run.py` `postwork.py` 和 `test.json`，来执行正确的测评逻辑；（2）向 `test` 文件夹中添加测试数据和提交文件，以适配不同的题目；（3）修改 `Dockerfile` 以正确生成评测所需的环境镜像；（4）进行本地测试并导出 `kernel.pyz`；（5）在通过本地测试之后，将内核文件 `kernel.pyz` 和对应的 `Dockerfile` 放到希冀平台的通用评测工作台中的镜像（Docker）里，再把测试数据也传到希冀上，使同学们能在希冀上测评。本文后续将分别介绍（1）~（5）部分。

评测原理是什么? 在希冀平台上，当学生点击提交时，希冀系统会自动使用指定的 Docker 镜像运行本评测核心，并将题目数据、学生提交内容及其他相关信息通过目录挂载或环境变量方式传入 Docker 容器中。评测机需要以恰当的方式运行学生所提交的程序，并根据其输出结果与标准答案进行比对，给出得分。

为保证学生使用的流畅性，我们需要先进行本地测试，实际上就是在本地模拟评测机行为。下面是希冀官方给出的一段描述：

> 对于测试数据的管理，可以先在本地测试，随后将文件提交到线上服务器。
>
> 在本地测试中，所有题目的测试数据位于\compiler\test 目录下，其对应的测试结果输出到 \compiler\test_result 目录下。

> 项目中test目录应该包含所有的测试题目，每个题目一个子目录，目录名无限制。**每个题目应包含testdata和submits两个子目录，testdata目录中应包含测试输入和标准输出，与希冀系统中“测试数据编辑器”中“work”目录内容保持一致。submits目录中可有多个子目录，子目录名不限。每个子目录表示一个不同的提交**，由于学生提交程序不仅仅有正确执行一种情况，还可能出现WA、CE、RE、TLE等多种情况，可以在此准备多个提交程序，以测试评测机应对不同情况的能力。
>
> **项目中\compiler\test.json用于声明内核执行方式和需要进行测试的样例。其中submit_dir、testdata_dir、kernel_dir、entrypoint与希冀系统中的“提交文件挂载路径”、“测试数据挂载路径”、“内核扩展挂载路径”和“启动参数”保持一致。**
>
> test.json中的testcases字段用于声明需要进行测试的样例，为列表类型，其中每个元素为一个测试题目。每个题目对象必须拥有一个name字段，为题目名称，与test目录中的每个子目录一致。题目对象中的submits字段表示此题目的提交数据，为列表类型。其元素中name字段与此题目中submits中的子目录名一致，verdict，score等其他字段为此题目在此提交下的期望输出，若此题目仅有一个提交，也可以将submits简写成submit，甚至可以省略submits。
>
> 执行python manage.py test即可自动使用本机Docker程序运行所有的测试点，测试结果存放在test_result目录中。
>
> 在线上测试中，单个题目的测试数据可以通过**题目-编辑-测试数据编辑器**进行修改，**其work目录下的内容应该与相对应的testdata目录下的内容保持一致。**
>
> **PyGrading推荐将评测过程分为prework，run和postwork三个阶段。**
>
> prework为评测前准备，主要完成评测环境的初始化，对学生提交的程序进行编译，准备测试数据等工作。prework阶段必须使用pygrading.create_testcase(100)创建测试样例集，并将本次评测所使用的测试数据使用append函数添加到样例集中，最后使用job.set_testcase函数设置测试数据。
>
> 对于每一个testcase都会执行一次run函数。在run函数中需要正式运行学生程序，并将其结果与标准答案进行比对，以Json格式返回此测试点的评分。PyGrading会将多次运行run函数的结果组织成一个列表，列表的长度与测试样例集的长度相同。
>
> 在postwork中，可以使用job.get_summary()获得执行run函数的结果列表。postwork需要通过对象中的score、verdict、comment、detail、secret、rank等函数设置评测结果。为了获得更好的显示结果，comment、detail、secret三个字段中可以包含HTML标签。PyGrading中内置了jinja2实现HTML模板渲染。

三、脚本编写

1.环境要求

docker，可参考[Ubuntu Docker 安装 | 菜鸟教程]

cython，

```
apt-get install python-pip python3
apt-get install build-essential
pip install cython
```

pygrading

```
cd compiler/docker/
pip install pygrading-1.1.8.dev0-py3-none-any.whl
```

2.测试数据准备

需要准备：`casesonfig.json` 配置文件，`.in` `.out` 文件（视情况决定是否需要标准输入和对比输出）

其中比较重要的是配置文件，其中的内容影响到脚本编写。

```
{
    "phase": "softmax",
    "testcaseList": [
        {
            "name": "softmax",
            "output": "/coursegrader/testdata/testcases/softmax.out",
            "score": 100
        }
    ]
}
```

其中的**“phase”“testcaseList” 关键都不需要修改，只需要在里面加入样例信息。**例如上图是笔者写的softmax测试信息，下图是编译原理课程另一位助教写的测试信息。需要什么就加什么

```
    "phase": "lab4_mem2reg",
    "base_fraction": 40,
    "idx": 4,
    "testcaseList": [
        {
            "name": "/coursegrader/testdata/testcases/0_lab2_lv0_1.cminus",
            "score": 2,
            "input": "/coursegrader/testdata/testcases/0_lab2_lv0_1.in",
            "output": "/coursegrader/testdata/testcases/0_lab2_lv0_1.out",
            "err": "/coursegrader/testdata/testcases/0_lab2_lv0_1.err",
            "min": 88,
            "max": 0
        },
```

3.prework.py

写在前面：

- **编写脚本的时候多用 `loge()` 可以在终端输出，相当于用输出监视、debug**
- **每个Python脚本中都保留了几个编译原理课程实验测评脚本，可以进行参考借鉴**

执行prework时会先执行脚本自带的 `prework()`，加载测试数据的配置信息，如果在对应的路径找不到配置信息 `caseconfig.json`，就会直接报 UnknownError

```python
def prework(job: Job):
    config_file = os.path.join(
        job.get_config().testcase_dir, "caseconfig.json" # 本地测试可以改成testcases/caseconfig.json
    )
    loge(job.get_config().testcase_dir)
    loge("prework config_file " + config_file)
    if not os.path.exists(config_file):
        loge("prework config_file not exist")      You, yesterday • first add
```

`prework()` 中的 `testcase_dir` 实际上是 `/coursegrader/testdata`，是 `test.json` 中默认的存放测试数据的路径，这一点在希冀平台上也是一样的，所以最好不要随意修改 `testcase_dir`

```python
    with open(config_file) as conf:
        raw_data = conf.read()
    files_config = json.loads(raw_data)
    phase = job.get_config()["phase"] = files_config["phase"]
    loge("prework...")
    eval("prework_" + phase + "(job)")
    loge("prework...done")
```

`prework()` 还会根据 `caseconfig.json` 中的 “phase” 来指定接下来要执行的函数，因此编写脚本时函数的名称是固定的，例如："phase" = "softmax", 那么我就需要写 `prework_softmax()`, `run_softmax()`, `postwork_softmax()`

接下来看 `prework_softmax()`：

首先是加载 `caseconfig.json` 中的各种信息，

```python
def prework_softmax(job:Job):
    # 获取测试点和配置信息
    testcase_dir = os.path.join(job.get_config().testcase_dir, "testcases")
    files_config = {}
    config_file = os.path.join(testcase_dir, "caseconfig.json")
    with open(config_file) as conf:
        raw_data = conf.read()
        files_config = json.loads(raw_data)
    testcaseList = files_config.get("testcaseList")
```

之后两句是惯例：

```python
    # 设置满分
    testcases = gg.create_testcase(100)
    # 更新测试点和配置信息到任务
    job.set_testcases(testcases)
```

然后，`prework.py` 还部分承担了环境准备的任务，此时可以进行 `cmake/make/build/run .sh/...` 等操作

```
    # 编译学生提交的程序
    submit_dir = job.get_config().submit_dir
    loge(f"--- submit_dir: {submit_dir}")
    try:
        # 建立 Ascend 环境
        compile_result = gg.exec(f"bash -lc 'cd {submit_dir} && source setup_env.sh'")
        if compile_result.returncode != 0:
            raise CG.CompileError(compile_result.stderr)
```

再加一些异常处理即可

通过环境准备之后将测试数据传递到 `run.py`，这里需要用到设置满分时准备的 `testcases` 对象

```
    # 加载测试点到希冀 testcases
    #gg.exec("sleep 5s")
    for itemCase in testcaseList:
        name = itemCase["name"]
        score = itemCase["score"]
        itemCaseDir = itemCase["output"]
        loge(f"--- expected output file: {itemCaseDir}")
        if os.path.exists(itemCaseDir):
            testcases.append(name = name, output_src = itemCaseDir, score = score, extension = {"detail": "", "submit_dir": submit_di
        else:
            detail = "Error: Cannot find testcase "+ name
            testcases.append(name=name, output_src = itemCaseDir, score=score, extension = {"detail": detail})
```

`testcases` 的 `extension` 中可以添加一些有利于 `run_softmax()` 的变量

4.run.py

创建一个 `result` 作为 `run_softmax()` 的返回值，以便传递给 `postwork_softmax()`。其中的内容需要至少包含 `name, score, verdict, detail`，其他可以自行添加

```
def run_softmax(job: Job, case: TestCases.SingleTestCase) -> dict:
    # 创建一个结果对象       You, yesterday • first add
    result = {
        "name": case.name,
        "score": 0,
        "verdict": Verdict.WrongAnswer,
        "detail": case.extension["detail"],
        "exe_result": []
    }
```

之后从 `prework_softmax()` 传递来的 `testcases(case)` 里获取测试数据，运行学生提交的程序。下图中笔者又执行了一遍 `source setup_env.sh` 是因为这个环境只在同一个“bash”生效

```python
    # 加载测试输入，若没有测试数据可以直接运行
    # 加载测试输出
    loge(f"-> case.output_src: {case.output_src}")
    expected_output = None
    if case.output_src:
        with open(case.output_src) as f:
            expected_output = f.read().strip()
    loge(f"-> set up env and execute python ")
    # 运行提交的程序
    submit_dir = case.extension["submit_dir"]
    try:
        # run_operation = f"cd {case.extension["submit_dir"]} && rm -rf kernel_meta && python softmax.py"
        # exec_result = gg.exec(run_operation, time_out=None)
        run_operation = (
            f"cd {submit_dir} && "
            f"source ./setup_env.sh && "
            f"rm -rf kernel_meta && "
            f"python softmax.py"
        )
```

运行成功后，加载学生的输出 `.out` 文件

```python
        exec_result = gg.exec(run_operation)
        #gg.exec("sleep 5s")
        loge(f"-> executed python and get result {exec_result}")
        # 加载学生的输出
        actual_path = os.path.join(submit_dir, "softmax.out")
        loge(f"-> actual_result_path: {actual_path}")
        with open(actual_path) as f:
            actual_output = f.read().strip()
        # actual_output.append(str(exec_result.returncode))
        # expected_output = str2list(expected_output)
        loge(f"-> expected output: {expected_output}")
        loge(f"-> actual output: {actual_output}")
```

将输出进行对比。理论上无论是以字符串形式还是浮点数形式都可以对比，看需要。笔者用的是浮点数对比

此处还可以添加分数计算公式，根据学生的答案和其他信息赋给 "score"（参考 `run_lab4_mem2reg()` ）

```python
        if expected_output is not None:
            try:
                exp = float(expected_output)
                act = float(actual_output)
                ok = (abs(exp - act) <= 1e-8)
            except ValueError:
                ok = (expected_output == actual_output)
        else:
            ok = False
        if ok:
            result["verdict"] = Verdict.Accept
            result["score"] = case.score
            result["exe_result"] =  actual_output
            loge("--- Accept! ---")
```

后面再添加一些异常处理、verdict 记录。将 `result` 传递给下一阶段

5.postwork.py

这一阶段可以参考编译原理课程实验脚本，主要作用是汇总分数、通过样例个数、渲染HTML。下图是 `postwork_softmax()` 的主体部分

```
    # 遍历每个测试点
    for result in summary:
        show_item = {}
        show_item['name'] = os.path.basename(result['name'])
        show_item['verdict'] = result['verdict']
        show_item['score'] = result['score']

        if result['verdict'] == Verdict.Accept:
            passed_tests += 1
        else:
            show_item['detail'] = result['detail']
            err_type = result['verdict']
        show_summary.append(show_item)
        all_tests += 1
        loge(f"result{all_tests}: {result}")
        score += result['score']
    # 每个测试点所要展示的结果对象
    info = {
        "all": all_tests,
        "passed": passed_tests,
        "score": score,
        "err_type": err_type
    }
```

四、内核制作

放置好 submit、testdata 并且编写好评测脚本之后，在 compiler 目录下执行 `python manage.py test` 可以根据 `test.json` 和 `caseconfig.json` 一键测试 compiler/test 下的样例。但是此操作需要 docker 的支持，因为前面提到过，希冀的测评逻辑就是调用 Docker、挂载测试数据和学生提交的文件。在类脑平台提供的临时镜像中，我们无法再次启动一个镜像。

> manage.py 支持的操作：
>
> 构建Docker镜像
>
> ```
> python3 manage.py docker-build
> ```
>
> 导出Docker镜像
>
> ```
> python3 manage.py docker-save
> ```
>
> 打包内核

```
python3 manage.py package
```

测试

```
python3 manage.py test
```

但是在类脑平台的机器上，我们可以使用 `python3 manage.py package` 进行内核打包，它会把 compiler/kernel 下的脚本打包，而且可以使用 `python/python3 kernel.pyz` 调用我们写好的脚本，因此我们能够进行本地验证（也就是 kernel.pyz 把我们的虚拟机当作测评用的那个 Docker）。

每次打包的内核会存在 compiler/backup-kernels, 而 compiler 目录下会出现 `kernel.pyz` 方便我们调用和下载。

美中不足的是，希冀打包好的内核默认测试数据挂载路径 /coursegrader/testdata, 提交文件挂载路径 /coursegrader/submits （可能会根据 `test.json` 改变，笔者暂未尝试）。有两个解决办法：

（1）直接在自己虚拟机建立 /coursegrader 文件夹操作，这样kernel能找到正确路径

（2）bind 到现有路径，例如：

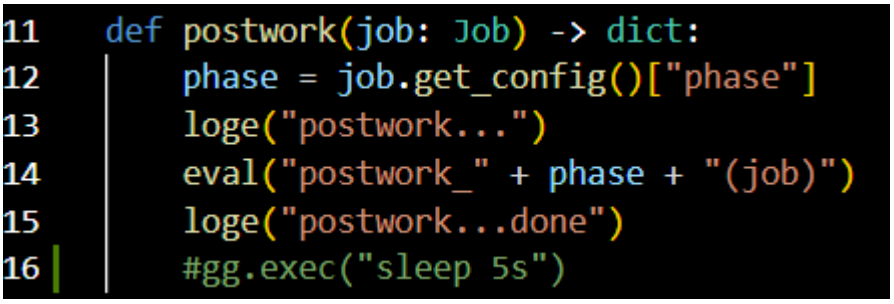
```
docker run --rm -e PYGRADING_DEBUG=true
--mount type=bind,
source="/home/yang/2023_compiler_xiji/compiler/test/4-opt-phase1/testdata",
target="/coursegrader/testdata",
readonly --mount type=bind,
source="/home/yang/2023_compiler_xiji/compiler/test/4-opt-phase1/submits/ta",
target="/coursegrader/submit"
--mount type=bind,
source="/home/yang/2023_compiler_xiji/compiler/kernel.pyz",
target="/coursegrader/dockerext/kernel.pyz"
--entrypoint python3 lxq2lxq/compiler-env:latest /coursegrader/dockerext/kernel.pyz
```

或者：

```
ln -s $(pwd)/test/softmax/testdata /coursegrader/testdata
ln -s $(pwd)/test/softmax/submits/stu /coursegrader/submit
```

五、本地验证

kernel.pyz 能跑通之后，就可以进行本地验证。善于用 loge() 寻找错误。另外需要注意： `postwork.py` 执行结束之后，kernel.pyz 正常结束后，把“评测结果 JSON”原样打印到了 stdout，终端会出现很多 JSON/HTML样的字段，淹没我们需要的信息。所以可以在下图位置加一句 `gg.exec("sleep 5s")` 及时 Ctrl+C



下图是笔者本地测试 softmax 算子的终端输出，在 compiler 目录下运行 `manage.py package` 和 `kernel.pyz`

```
prework config_file /coursegrader/testdata/testcases/caseconfig.json
prework...
--- submit_dir: /coursegrader/submit
--- expected output file: /coursegrader/testdata/testcases/softmax.out
prework...done
run...
-> case.output_src: /coursegrader/testdata/testcases/softmax.out
-> set up env and execute python
-> executed python and get result <pygrading.utils.Exec object at 0xffffaa35f950>
-> actual_result_path: /coursegrader/submit/softmax.out
-> expected output: 1e-8
-> actual output: 0.00000001
--- Accept! ---
run...done
postwork...
result1: {'name': 'softmax', 'score': 100, 'verdict': 'Accept', 'detail': '', 'exe_result': '0.00000001'}
```

六、转移到希冀平台

1.构建通用评测机

本地测试无误后，登录希冀平台教师端，点击**“作业” -- “通用评测工作台” -- “新建评测机”**，写入或复制粘贴正确无误的 Dockerfile 的内容，**上传 pygrading 附件，上传对应的内核 pyz 文件**，然后构建。也可以在现有评测机那一栏选择 "Fork" (注意修改高级选项-架构！)

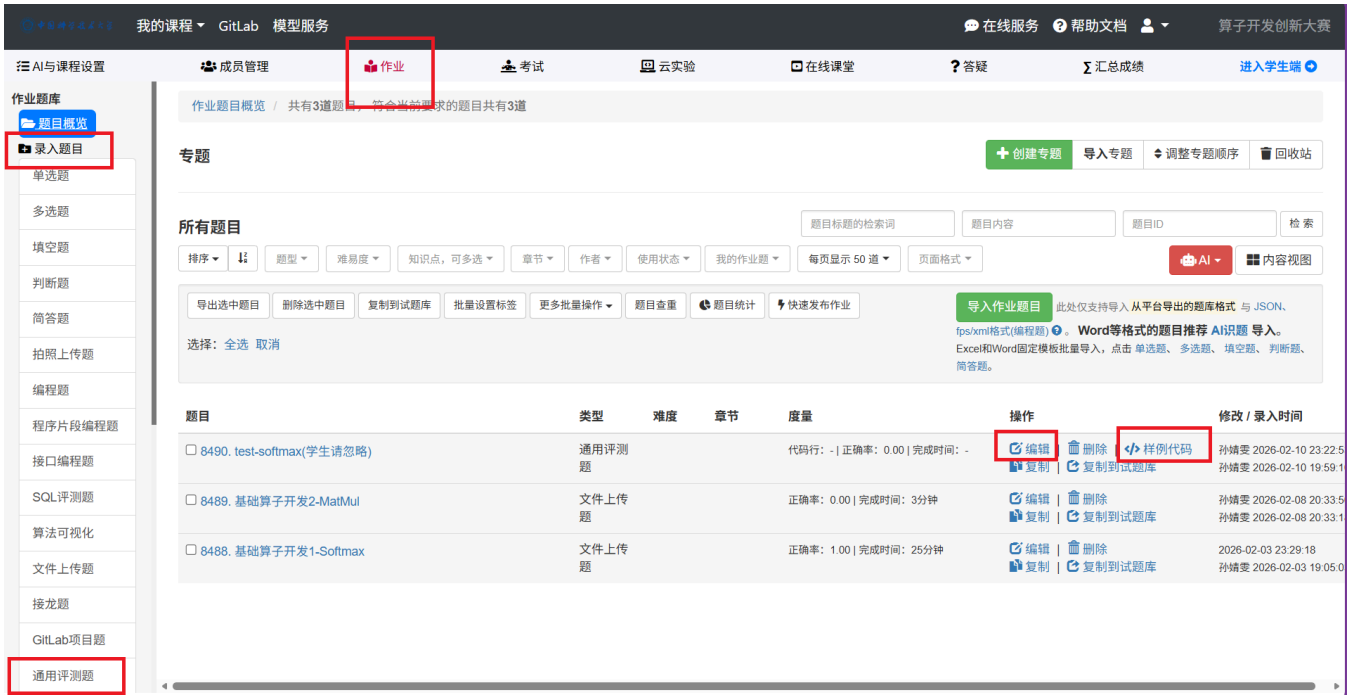
2.导入测试数据

在希冀平台点击**“作业”--“录入题目” -- “通用评测题”**，填写题目信息。 在最下面 “正确性验证” 处点击 **“测试数据编辑器”**，打开 JupyterNotebook，然后点击左上角 ”File“，打开 /work 目录。

这个目录相当于本地测试中的 /coursegrader/testdata。接下来对应地上传 `caseconfig.json` 和 testcases 就可以了。

学生提交后的上传数据会自动挂载到 `submit_dir`，不用管理

题目创建成功后可以上传样例代码，借此机会检测我们本地测试跑出来的内核是否正确、测试数据是否放置到位



其他（前人遗留）

project_utils.py 执行test在对比输出结果能够被json读取时 会正确打印test_result 并将comment detail 以html格式写入相应的html文件
而stdout则会被读取走 加载到平台的控制台上
若cg.is_terminate == true 则会终止run 和 postwork

```
FROM ascendai/cann:8.5.0-910b-ubuntu22.04-py3.11
#FROM osrf/ubuntu_arm64:jammy
USER root
ENV DEBIAN_FRONTEND=noninteractive

RUN sed -i 's|http://ports.ubuntu.com/ubuntu-ports|https://mirrors.ustc.edu.cn/ubuntu-ports|g' /etc/apt/sources.list && \
    apt-get update && \
    apt-get install -y --no-install-recommends \
        python3 \
        python3-pip \
        build-essential \
        llvm-dev \
    && rm -rf /var/lib/apt/lists/*

RUN pip3 config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple

COPY pygrading-1.1.8.dev0-py3-none-any.whl /tmp/
RUN pip3 install /tmp/pygrading-1.1.8.dev0-py3-none-any.whl && \
    pip3 install json5 setuptools wheel

ENTRYPOINT ["python3", "/coursegrader/dockerext/kernel.pyz"]
```