# 致歉

真的很抱歉，上周作业拓展二因为没有完全理解，加上学业繁重，现在才补交。我现在差不多理解了一些关于拓展二的内容，现在可以开始作答了

# 1.关于Condor的认识(虽然是AI生成，但我已经严肃学习)

## 1. 餐厅里的“领班经理”

想象你有一家很大的餐厅（**计算机集群**，有很多台电脑/CPU核心），后厨有很多厨师（**CPU核心**）。
你要做 100 道菜（**M 个作业**），但后厨只有 8 个灶台（**N 个 CPU 核**）。

- 如果没有领班：你自己把 100 道菜同时扔给后厨，灶台不够，厨房炸了（电脑卡死）。
- 如果有领班（Condor）：你把 100 道菜的要求告诉领班，领班负责：
  - **排队**：每次只给厨师上 8 道菜
  - **分配**：哪道菜给哪个厨师做
  - **监控**：谁做完了，把结果收回来，再给下一道菜
  - **记录**：谁做失败了，留下日志

Condor 就是这个“领班经理”。

## 2. 技术上的定义

Condor（现在改名叫 **HTCondor**）是一个专门用于**高吞吐量计算**的作业调度系统。
它的核心能力是：

- **管理大量计算任务**（可以成千上万个）
- **自动分配资源**（哪台机器有空闲 CPU，就去那里跑）
- **保证任务最终完成**（失败了可以重跑，机器坏了自动换一台）
- **记录日志**（每个任务的输出、错误、运行记录）

## 3. 在你作业中的角色

你写的那个脚本（`.sub` 文件），就是给 Condor 的“任务清单”。比如：

```
executable = protest3.sh           # 要做什么菜（程序）
transfer_input_files = $(indata)    # 用什么原料（输入文件）
queue indata matching files *.in    # 一共有多少道菜（100个.in文件）
```

你运行 `condor_submit` 后，Condor 就会：

1. 扫描出所有 `.in` 文件

2. 为每个文件创建一个作业（job）
3. 把作业放入队列
4. 根据集群当前的空闲 CPU 核数，**同时运行 N 个作业**（N 由管理员配置）
5. 每个作业完成后，把输出文件（`.root`）传回来
6. 记录日志到指定目录

## 4. 为什么用 Condor 而不是自己写脚本?

**你可能会想：** 我自己写一个 Shell 循环，一个一个跑不行吗?

- 自己跑：串行执行，100 个任务要等 100 倍的时间。
- 自己开多个后台进程：需要手动管理数量、处理失败重试、考虑不同机器的资源。很麻烦。

**Condor 的好处：**

- **并行**：同时跑 N 个，充分利用所有 CPU。
- **容错**：某个作业因为机器故障失败了，Condor 会自动在其他机器上重跑。
- **跨机器**：你的作业可能会跑到集群里不同的物理机上，你完全不用关心是哪台。
- **优先级、配额**：可以设置作业优先级、限制用户最多用多少资源（避免你一个人占满整个集群）。

## 5. 一个非常简单的理解

| 你做的事 | Condor 做的事 |
| --- | --- |
| 写一个 `submit.sub` 文件 | 阅读它，理解任务内容 |
| 运行 `condor_submit` | 把任务加入全局队列 |
| - | 找空闲的 CPU 核 |
| - | 把程序、输入文件复制到那台机器 |
| - | 运行你的程序 |
| - | 把输出文件复制回来 |
| - | 记录日志，告诉你成功/失败 |
| 运行 `condor_q` | 显示当前队列状态 |

## 6. 你不需要管的技术细节

- Condor 支持多种“宇宙”(universe)，你用的是 `vanilla`，最常用的标准模式。
- 它内部有“中央管理器”“执行节点”“提交节点”等组件，但作为用户你只需提交作业。
- 很多大学、实验室、CERN（欧洲核子研究中心）都在用 Condor 处理海量数据。

# 总结一句话

**Condor 就是一个自动帮你“排队、分配、监控、重试”大量计算任务的软件，让你能轻轻松松同时跑几百上千个作业，而不用担心资源不够或管理混乱。**

# 2.我为什么要学shell语言

# 1. Shell 就是你和电脑之间的“翻译官”

你平时操作电脑是点点鼠标，但服务器（比如你们学校的计算集群）通常**没有图形界面**，只能打字输入命令。
Shell 就是一个**命令行环境**，你打的每一个命令（如 `ls`，`cd`，`mkdir`）其实都是 Shell 在解释执行。

学 Shell 语言，就是学怎么**写一串命令保存成一个文件**，让电脑自动按顺序执行。这个文件就叫 **Shell 脚本**（`.sh` 文件）。

# 2. 没有 Shell 脚本，你的 Condor 作业根本跑不起来

回顾你刚做的作业：

- Condor 提交文件告诉系统：`executable = protest3.sh`
- 这个 `protest3.sh` 就是 Shell 脚本，里面写了真正的处理步骤（比如解压数据、运行程序、保存结果）。

**如果你不会 Shell 语言：**

- 你看不懂 `protest3.sh` 里写了什么，出错了不会改。
- 你没法把“一个输入文件”变成“输出文件”的逻辑写清楚。
- 你不能灵活地给每个作业传递不同的参数。

**学会 Shell 后：**
你就能自己写这样的脚本：

```
#!/bin/bash
# 处理一个 .in 文件，生成 .root 文件
input=$1
output=${input%.in}.root
/my_program $input $output
```

非常简短，但能完成核心工作。

# 3. Shell 可以批量处理文件（比手动快 100 倍）

假设你有 1000 个 `.in` 文件，你想先看看每个文件的前几行内容，不需要提交作业，只想检查。
手动做：累死。
用 Shell 一行搞定：

bash

for f in *.in; do echo "= $f ="; head -n 5 $f; done

学会这种循环、条件判断、变量替换，你就能把重复的工作交给电脑。

## 4. Shell 是连接各种工具的“胶水语言”

你的计算流程可能不止一步：

1. 解压数据（`tar -xzf`）
2. 运行 C++ 程序（`./simulation`）
3. 用 Python 画图（`python plot.py`）
4. 把结果打包（`zip results.zip *.root`）

Shell 脚本可以把这些不同语言的程序**串起来**，一条龙自动完成。

## 5. 在服务器上调试作业，必须懂 Shell

当你用 `condor_q` 查看作业状态，或者用 `condor_ssh` 登录到运行作业的机器，看到的都是命令行。
如果你连 `ls`，`cat`，`grep`，`ps` 这些基本命令都不懂，你就不知道：

- 作业为什么失败（看日志 `cat job.err`）
- 机器资源够不够（`top`，`free -h`）
- 文件在哪里（`find`，`ls -l`）

**Shell 就是你在服务器上的“生存技能”**。

## 6. 学 Shell 并不难，而且很快能见效

你不需要成为专家，只需要掌握 20% 的常用语法（变量、循环、条件、函数、重定向），就能解决 80% 的日常任务。
而且你会发现，很多其他工具（比如 Condor 提交文件、Git 钩子、Docker 启动命令）都带有 Shell 的影子。

## 总结：回到你的问题“为什么要学 Shell 语言？”

> **因为它是你控制计算机（尤其是服务器）最直接、最自动化的方式。没有它，你就只能手动重复劳动，出错了不会改，批量了做不了。你刚完成的 Condor 作业，只是第一次尝到了“自动化”的甜头，而 Shell 就是那把能打开所有自动化的钥匙。**

## 3.完成脚本

## 1. 作业要做什么？

你有一个**文件夹**，里面有 M 个文件（比如 100 个），每个文件的后缀是 `.in`。
你需要对**每一个** `.in` **文件**运行同一个处理程序（`protest3.sh`），但**同时最多只能使用 N 个**

**CPU 核**（比如 8 核）。

这就像你有 100 个任务，但只能同时做 8 个，做完一个再开始下一个。
这种“排队做任务”的工作，就要交给 **Condor 作业调度系统** 来管理。

## 2. Condor 提交文件的作用

你给的那段代码就是一个 **Condor 提交文件**（通常以 `.sub` 结尾）。
它的作用是告诉 Condor：

- 要运行哪个程序（`executable`）
- 每个任务需要的资源（CPU、内存等）
- 输入文件在哪里、输出文件放在哪
- 要处理多少个文件（通过 `queue` 语句）

## 3. 一步一步教你修改提交文件

假设：

- 你的 `.in` 文件放在 `/home/yourname/data/` 目录下
- 你的处理脚本 `protest3.sh` 在 `/home/yourname/bin/` 目录下
- 你想让每个作业用 1 个 CPU 核，最多同时运行 N 个（N 由学校分配给你，比如 8）
- 你把输出日志放在 `/home/yourname/log/` 目录下（需要自己创建）

### 步骤 1：创建必要的目录

打开终端，执行：

```
mkdir -p /home/yourname/log
```

（把 `yourname` 换成你的用户名）

### 步骤 2：修改提交文件（关键！）

把原代码复制到一个新文件，比如 `submit_jobs.sub`，然后按下表修改：

| 原代码行 | 修改为 |
| --- | --- |
| `RequestCpus =` | `request_cpus = 1` |
| `executable = /afs/cern.ch/user/q/qyan/.../protest3.sh` | `executable = /home/yourname/bin/prot` |
| `queue indata matching files //*.in` | `queue indata matching files /home/yourname/data/*.in` |

| 原代码行 | 修改为 |
| --- | --- |
| `output = //log/$Fn(indata).$(ID).out` | `output = /home/yourname/log/job.$(ClusterId)` |
| `error = //log/$Fn(indata).$(ID).err` | `error = /home/yourname/log/job.$(ClusterId)` |
| `log = //log/$Fn(indata).$(ID).log` | `log = /home/yourname/log/job.$(ClusterId)` |
| `arguments = .C /.C $Fnx(indata) ...` | `arguments = "$(indata)"` |
| `transfer_output_files = ""` | `transfer_output_files = *.root` |

> 为什么简化 `arguments`？因为原来的 `$Fnx(indata)` 可能是自定义函数，我们不确定能用。更可靠的方法是：让脚本自己根据输入文件名决定输出文件名。后面我会教你修改脚本。

### 步骤 3：修改你的处理脚本 `protest3.sh`（如果必要）

原来的脚本可能依赖复杂的参数形式。为了保险，我们给它加一个“外包装”，让它接受一个简单参数（输入文件名），然后在内部调用原来的逻辑。

打开 `protest3.sh`，在**开头**加入以下内容（假设原来脚本的参数格式是 `.C /.C 输入文件名 输出文件名 '' ''`）：

bash

#!/bin/bash

# 这个包装脚本会接收一个参数：输入文件路径

input_file="$1"

# 从输入文件名生成输出文件名（比如 a.in → a.root）

base_name=$(basename "input_file" .in)
output_file="${base_name}.root"

# 调用原来的命令（根据原 arguments 的格式调整）

# 原来可能是：.C /.C $Fnx(indata)///$Fn(indata).root " "

# 我们改成直接用生成好的输出文件名

/bin/bash -c ".C /.C $input_file/$output_file " ""

> 如果原来的命令不是你写的，你需要去问一下或者看原脚本的说明。但作为作业，这样简化通常是允许的，因为核心是“批量提交”，而不是脚本逻辑。

## 步骤 4：提交作业

在终端执行：

```
condor_submit submit_jobs.sub
```

然后你会看到类似：`Submitted 100 job(s).` 就成功了。

## 步骤 5：监控作业运行

- 查看你的作业队列：`condor_q`
- 查看所有正在跑的作业：`condor_q -run`
- 如果要限制同时运行的作业数恰好为 N（比如 8），**一般不需要额外设置**，因为 Condor 会根据你申请的 `request_cpus` 和实际可用的 CPU 核心数自动调度。如果 N 是你们组分配的限制，管理员已经配置好了。

如果你确实想**强制**同时跑不超过 N 个作业，可以在提交文件最后加一行：

text

max_running = 8

（把 8 换成你的 N）

## 步骤 6：查看结果

作业完成后，输出文件（`.root` 文件）会出现在你当前目录（因为 `transfer_output_files = *.root` 会把它们传回来）。
日志文件在 `/home/yourname/log/` 下，可以查看每个作业的输出和错误。