

## 5.11 科创

### 1、安装 g++、ROOT

```
one@fedora:~  
(base) one@fedora:~$ conda activate rootenv  
(rootenv) one@fedora:~$ root --version  
ROOT Version: 6.38.04  
Built for linuxx86_64gcc on Apr 08 2026, 08:58:34  
From tags/6-38-04@6-38-04  
(rootenv) one@fedora:~$  
  
one@fedora:~  
(base) one@fedora:~$ g++ --version  
g++ (GCC) 16.1.1 20260515 (Red Hat 16.1.1-2)  
Copyright (C) 2026 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

### 2、写一个 Makefile，编译第一个 root 程序，输出“Hello World”

即可

```
one@fedora:~  
(base) one@fedora:~$ vim hello.cxx  
(base) one@fedora:~$ cat hello.cxx  
#include<iostream>  
using namespace std;  
  
int main(){  
    cout << "Hello World"<<endl;  
    return 0;  
}  
(base) one@fedora:~$ vim Makefile  
(base) one@fedora:~$ cat Makefile  
CXX = g++  
CXXFLAGS = $(shell root-config --cflags)  
LDFLAGS = $(shell root-config --libs)  
hello: hello.cxx  
    $(CXX) -o hello hello.cxx $(CXXFLAGS) $(LDFLAGS)  
clean:  
    rm -f hello  
(base) one@fedora:~$ make  
make: root-config: No such file or directory  
make: root-config: No such file or directory  
g++ -o hello hello.cxx  
(base) one@fedora:~$ ./hello  
Hello World  
(base) one@fedora:~$
```

### 3、拓展 1：（为什么要用 ROOT? ）了解 ROOT 软件的存储结构，为什么用 ROOT 存储，而不是 txt、bin？

ROOT 的储存结构：头文件、数据记录和尾部；调用文件内容时，检索目录，只读取对应内容

#### 核心差异速览

- TXT：流式的行结构，纯文本。
- BIN：块式的字段结构，连续内存镜像。
- ROOT：文件系统式的对象结构，带索引与目录。

#### TXT 适用场景

核心优势：人类可读、通用性强

- 配置文件：如 config.txt，参数少且需人工修改
- 调试输出：程序日志或中间结果，方便肉眼检查
- 数据交换：与不支持二进制的旧系统对接
- 超小数据集：记录数 < 1万，此时解析开销可忽略
- 教学示例：演示基础 I/O，结构一目了然

不适合：GB 级以上数据、需频繁随机访问、追求存储或解析效率的场景

#### BIN 适用场景

核心优势：紧凑、定长、直接映射内存

- 嵌入式系统：资源受限环境，解析逻辑需极简
- 固定结构日志：传感器按固定格式每秒写入一条记录
- 内存映射文件：需将磁盘数据直接当数组操作（如 mmap）
- 游戏存档：直接 dump 内存结构，简单快速
- 作为 ROOT 的中间层：先存 BIN，后续转换

不适合：结构会频繁变化、需跨不同架构移植、要按特定列过滤的场景

#### ROOT 适用场景

核心优势：自描述、列式存储、随机访问

- 高能物理分析：PB 级数据，按列读取（如只读动量）
- 大规模科学模拟：含元数据、运行参数和分析结果的复杂数据集
- 机器学习预处理：需频繁读取部分特征列
- 长期存档：数据结构会随分析需求迭代（ROOT 原生支持模式进化）
- 复杂对象存储：不仅存数字，还要存直方图、图谱等对象

不适合：数据量小于 10MB、只需简单查看几行、协作者都未安装 ROOT 环境的场景

高能物理分析，数据量大，txt、bin 调用不方便。ROOT 列式储存，可快速读取指定数据，并且可压缩，内存小。

## 4、拓展 2：（为什么要学 Shell 语言？）已知有一个文件夹的 M 个文件需要提交，分配得到的 CPU 核数为 N，执行文件为 executable 变量，如何批量实现提交作业？

学习 Shell 语言是为了：

1. 批量化：处理大量（M 个）文件/作业时，循环自动化
2. 智能化：根据实际情况（N 个 CPU 核）动态调整策略
3. 流程化：构建“提交→监控→重试→合并”完整流水线
4. 可复用：写一次脚本，反复使用，避免重复劳动

在 HPC 集群、Condor、Slurm 等环境中，Shell 就是胶水语言，把各种工具粘合成自动化工作流。

**理由：批量处理重复操作，提升效率**

**实现思路：准备文件列表——编写 Condor 提交文件（定义变量，遍历）——控制同时处理数量小于 N**

核心目标：

有 M 个数据文件要处理，分配给用户的 CPU 核数是 N，需要把这些文件全部处理完，但不能让同时运行的作业数超过 N（否则会被系统限制或排队）。

解决思路分三步：

### 1. 准备文件列表

先把所有需要处理的 M 个输入文件的完整路径，全部写到一个文本文件里，每行一个文件。这样 Condor 可以一行一行地读取。

### 2. 编写 Condor 提交描述文件

- 在这个文件里，不要写死具体的文件名，而是定义一个变量（比如叫 `indata`），代表“当前要处理的这个输入文件”。
- 在 `executable`（你要运行的程序或脚本）后面，用 `$(indata)` 把这个文件路径传进去。

- 输入、输出、日志文件的名字也要用 `$(indata)` 或者 Condor 自带的作业 ID (`$(ClusterId).$(ProcId)`) 来区分，防止不同作业意思就是从 `filelist.txt` 里读取每一行，每次把这一行的值赋给 `indata`，然后提交一个作业”。

### 3. 控制并发数量

- 默认 Condor 会尽可能多地同时跑作业，但这样可能超过你分配的 N 个核，导致作业被挂起或挤占别人资源。
- 所以提交时要加限制：  
`condor_submit -maxjobs N` 你的提交文件。这样 Condor 最多同时运行 N 个作业，跑完一个再启动下一个，直到 M 个全部完成。

总结一句话：

把 M 个文件路径写进列表，让 Condor 用 `queue from` 语句为每个文件生成一个作业，再用 `-maxjobs N` 限制同时运行的作业数不超过 CPU 核数 N。