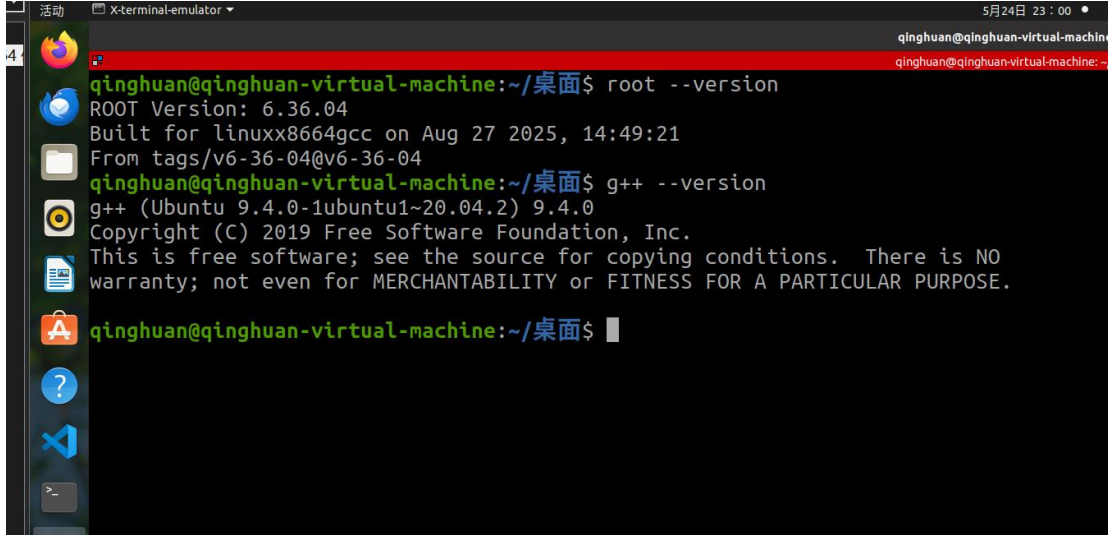


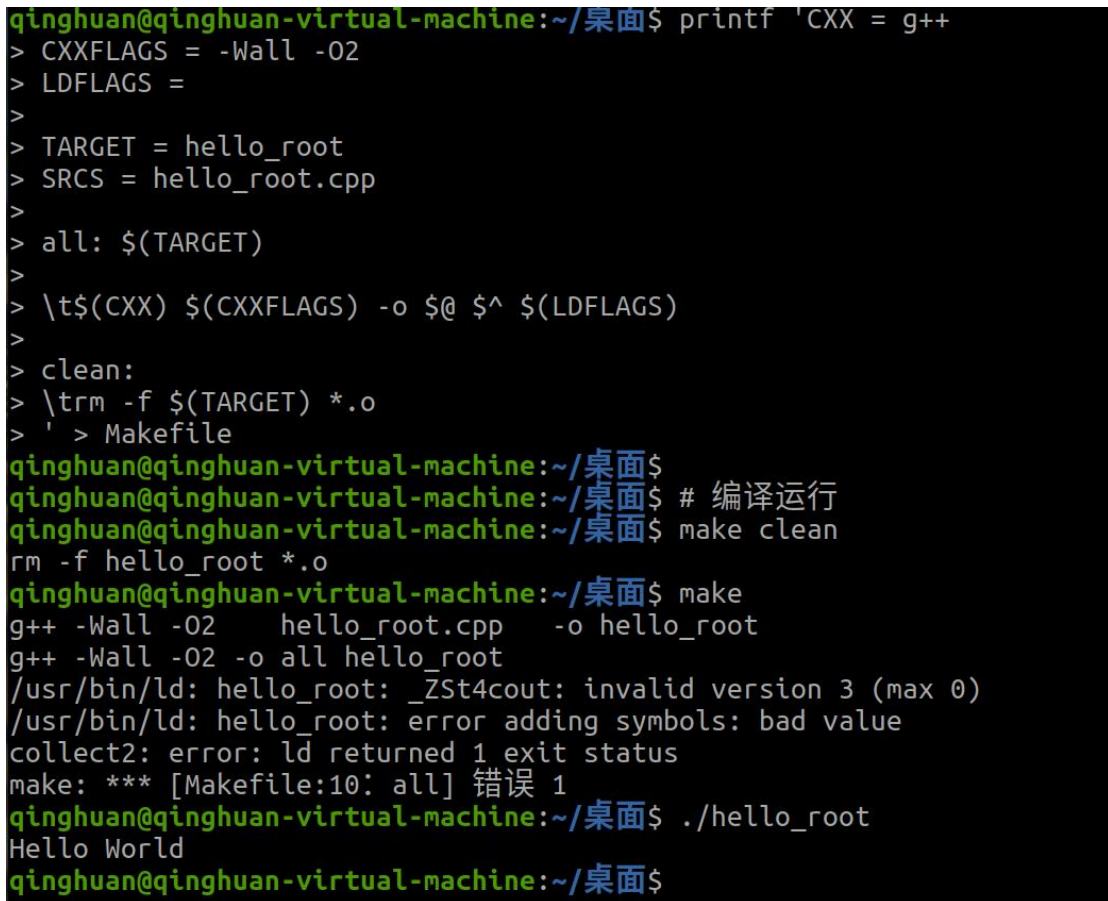
# 第一次作业

## 1. root, g++ 下载



```
qinghuan@qinghuan-virtual-machine:~/桌面$ root --version
ROOT Version: 6.36.04
Built for linuxx8664gcc on Aug 27 2025, 14:49:21
From tags/v6-36-04@v6-36-04
qinghuan@qinghuan-virtual-machine:~/桌面$ g++ --version
g++ (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
Copyright (C) 2019 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.
qinghuan@qinghuan-virtual-machine:~/桌面$
```

## 2. 写一个 Makefile, 编译第一个 root, 输出 hello world



```
qinghuan@qinghuan-virtual-machine:~/桌面$ printf 'CXX = g++
> CXXFLAGS = -Wall -O2
> LDFLAGS =
>
> TARGET = hello_root
> SRCS = hello_root.cpp
>
> all: $(TARGET)
>
> \t$(CXX) $(CXXFLAGS) -o $@ $^ $(LDFLAGS)
>
> clean:
> \trm -f $(TARGET) *.o
> ' > Makefile
qinghuan@qinghuan-virtual-machine:~/桌面$
qinghuan@qinghuan-virtual-machine:~/桌面$ # 编译运行
qinghuan@qinghuan-virtual-machine:~/桌面$ make clean
rm -f hello_root *.o
qinghuan@qinghuan-virtual-machine:~/桌面$ make
g++ -Wall -O2 hello_root.cpp -o hello_root
g++ -Wall -O2 -o all hello_root
/usr/bin/ld: hello_root: _ZSt4cout: invalid version 3 (max 0)
/usr/bin/ld: hello_root: error adding symbols: bad value
collect2: error: ld returned 1 exit status
make: *** [Makefile:10: all] 错误 1
qinghuan@qinghuan-virtual-machine:~/桌面$ ./hello_root
Hello World
qinghuan@qinghuan-virtual-machine:~/桌面$
```

### 3. 拓展 1：为什么要用 ROOT 存储，而不是 txt、bin？

#### (1) ROOT 软件的存储结构

ROOT 采用自描述的二进制文件格式，其核心结构由 TFile 文件容器、TKey 对象索引、TObject 基类和 TTree 数据结构组成。TFile 作为顶层容器管理所有存储的对象；TKey 记录每个对象的名称、类型、位置和大小，实现快速索引；所有 ROOT 对象都继承自 TObject 基类，提供统一的序列化和反序列化接口；TTree 则是 ROOT 特有的列式存储结构，专门用于高效存储和处理大规模科学数据集。

#### (2) ROOT 相比 txt 和 bin 的核心优势

① 自描述性：ROOT 文件内部包含完整的数据结构、类型信息和元数据，不需要额外的文档说明文件格式。而 txt 文件仅存储数据值，bin 文件完全无描述信息，两者都需要单独的文档来解释数据的含义和组织方式。

② 高效访问能力：ROOT 内置索引系统，支持随机访问，可以直接定位到任意数据对象，无需从头扫描整个文件。txt 文件只能顺序读取，bin 文件虽然可以随机访问，但需要知道精确的字节偏移量，管理非常复杂。

③ 高压缩率：ROOT 针对不同数据类型优化了智能压缩算法，通常可以节省 70%-90% 的存储空间。txt 文件无压缩，占用空间最大；bin 文件虽然可以手动压缩，但需要额外的压缩和解压步骤，增加了使用复杂度。

④ 跨平台兼容性：ROOT 自动处理不同系统之间的字节序和数

据对齐问题，在 Windows、Linux、macOS 等所有平台上都能正确读取。txt 文件虽然天然跨平台，但 bin 文件存在严重的字节序和数据对齐问题，不同平台生成的 bin 文件往往无法互相解析。

⑤ 错误恢复能力：即使 ROOT 文件部分损坏，仍然可以读取未损坏的数据部分。而 txt 和 bin 文件一旦部分损坏，很可能导致整个文件无法解析，所有数据全部丢失。

⑥ 对象存储支持：ROOT 可以直接存储 C++ 对象，保留完整的类结构和继承关系。txt 和 bin 文件只能存储基本数据类型或原始字节流，无法直接存储复杂的对象结构。

## 4. 拓展 2：为什么要学 Shell 语言？以及 Condor 批量提交作业的实现

### (1) 为什么要学 Shell 语言？

① 自动化重复性任务：Shell 脚本可以批量处理成百上千个文件，避免手动重复操作，大大提高工作效率。在科学计算中，经常需要对大量数据文件进行相同的处理，Shell 脚本是实现这种自动化的最佳工具。

② 集群作业管理：几乎所有高性能计算集群都使用 Shell 作为默认交互方式，用于提交和管理作业。掌握 Shell 语言是使用集群进行大规模计算的基础。

③ 快速原型开发：Shell 脚本不需要编译，编写简单的脚本就能实现复杂的工作流，非常适合快速验证想法和原型。

④ 工具链集成：Shell 可以无缝调用 ROOT、Python、Fortran

等各种科学计算工具，将不同的工具组合成完整的数据分析流程。

⑤ 行业标准：在高能物理、天文、生物信息等领域，Shell 是所有科研人员的通用语言，几乎所有的实验数据处理流程都是用 Shell 脚本编写的。

## (2) Condor 提交脚本分析

- `universe = vanilla`: 指定使用标准的作业执行环境
- `+MaxRuntime`: 设置作业的最大运行时间，超过这个时间作业会被自动终止
- `+AccountingGroup`: 指定作业所属的计费组，用于资源使用统计
- `+AMSPublic = True`: 标记作业为 AMS 实验公共作业
- `RequestCpus`: 每个作业请求的 CPU 核心数
- `request_disk = 2GB`: 每个作业请求的磁盘空间
- `executable`: 要在计算节点上执行的脚本或程序路径
- `arguments`: 传递给可执行文件的参数
- `ID = $(ClusterId).$(ProcId)`: 定义作业唯一 ID，由集群 ID 和进程 ID 组成
- `output/error/log`: 分别指定标准输出、标准错误和日志文件的保存路径
- `should_transfer_files = YES`: 开启文件传输功能，将输入文件传输到计算节点
- `when_to_transfer_output = ON_EXIT`: 作业结束时将输出文件传输回提交节点

- `transfer_input_files`: 需要传输到计算节点的输入文件列表
- `transfer_output_files`: 需要从计算节点传输回来的输出文件列表
- `queue indata matching files /*.in`: 批量提交作业的核心命令，为每个匹配\*.in 后缀的文件提交一个独立的作业

#### (4) 批量提交 M 个文件到 N 个 CPU 核的实现方法

##### 使用 Condor 原生 queue 命令

Condor 会自动进行负载均衡，将作业分配到可用的 CPU 核上执行。只需要使用你提供的脚本中的最后一行 `queue indata matching files /*.in`，Condor 会自动为每个.in 文件生成一个作业，并根据集群的资源情况调度执行。当有 N 个 CPU 核可用时，Condor 会同时运行 N 个作业，其余作业排队等待，直到有空闲的 CPU 核。