构建流程与工具：Make、CMake 等

- 探讨构建流程与软件的自动化方法
- 借助 make 工具及相关技术实现流程自动化
- 提升科研与开发效率

核心主题：

- 探讨构建流程与软件的自动化方法
- 利用 make 工具及相关技术提升效率

学习目标：

- 理解自动化构建的意义
- 掌握 make 工具的使用方法
- 学会设计构建流程，搭建自动化环境

自动化构建的价值

- **效率提升**：替代手动重复操作，如编译、链接等流程，节省时间
- **减少错误**：规避人工操作失误，保证构建过程一致性
- **案例引导**：以科研场景（如论文数据更新、图表生成）为例，说明自动化需求
- 将繁琐步骤自动化，同事更易使用你的工作

建流程基础步骤

1. **配置**：设置环境变量、用户选项，为后续步骤做准备
2. **编译**：将源代码转换为机器可识别的目标代码
3. **链接**：把编译后的代码及依赖库整合，生成可执行文件
4. **安装**：将构建好的软件部署到指定系统位置
5. **运行**：执行软件，验证功能；也可包含分析流程（如用 Python 处理数据生成图表）

TIPS：**编译步骤仅需编译型语言（如C/Fortran）**

make 工具介绍

- **介绍**：make 是一个命令行实用程序，用于确定处理流程中需要执行哪些元素。make 可以用来自动化处理所有具有依赖关系的流程，即各构建文件之间含有依赖关系。
- **作用**：自动处理依赖关系，确定需更新 / 执行的元素，简化构建流程
- **应用场景**：科研论文更新（数据、图表、LaTeX 文档联动）、软件项目构建等
- **优势**：智能追踪依赖，仅更新变动部分，避免重复劳动
- 由于 make 会检测依赖关系树中的哪些文件已更改，所以只会执行必要的处理，而不会从头处理所有内容。这样节省了大量时间，特别是可以省去某些耗时很长、但无需更新的步骤。

尝试运行一次make

在命令行中输入以下语法使用make：

```
make [ -f makefile ] [ options ] ... [ targets ] ...
```

值得注意的是，这里面有许多的参数，参数是默认的时候即可运行

- -f makefile：指定使用的 Makefile 文件名称
- [options]：用于控制make行为，常见的option有
- -n 只显示即将要执行的命令
- -B 强制重构所有目标，忽略时间戳 等等等等

```
~/shell_model $ make
make: *** No targets specified and no makefile found. Stop.
```

那么这个makefile是什么东西呢

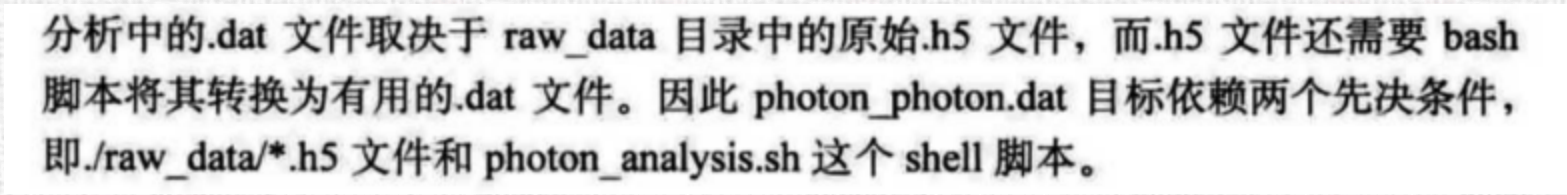
- **定义**：描述文件依赖关系和构建指令的文本，其作用是为make描述文件和人物之间的关系依赖叔，是 make 执行的“剧本”
- **核心语法**： 目标：先决条件 ，搭配执行命令（如生成数据文件、编译代码）
- 如果没有为make命令提供参数,那么make 会尝试在当前目录中寻找一个名为Makefile的文件。上面的例子中报错是因为还没有在分析目录中创建 Makefile。
- 如果makefle文件的名称不是Makefile，那么必须告诉make-f标志就是用于告诉 make 该文件的位置。通常只有在单个目录存在多个Makefile 时才需要使用其他名称。按照惯例，make文件会调用以.mk扩展名结尾的Makefile 文件。

Makefile 进阶 （目标）

- makefile首先会跟刚刚一样定义目标。目标也就是依赖关系树中的节点，主要使用“目标-先决条件-行为”这样的语法来定义目标

```
target : prerequisites (1)
         action (2)
```

- 冒号的目的是将目标名称(target)和先决条件(prerequisites)隔离开,在action前必须要有一个制表符字符



- photon_photon.dat 文件是怎么来的?
  1.它需要两个东西才能生成:
  ①./raw_data/ 文件夹里的所有 .h5 原始数据文件。 （* 代表所有）。
  ②那个叫 photon_analysis.sh 的脚本程序。

- 怎么生成 photon_photon.dat 文件?
  1.运行那个脚本程序 photon_analysis.sh
  运行命令是:
  photon_analysis.sh ./raw_data/*.h5
  ./photon_photon > photon_photon.dat
  (1)其中要创建或更新的目标文件是photon photon.data文件。先决条件(所依赖的文件)是 shell 脚本和.h5 文件。
  (2)根据先决条件更新目标时必须执行这条命令。
  在这个示例中，第一行是描述文件的注释。这种习惯很好，且并不影响makefle的行为。第二行描述了目标和先决条件，第三行描述了make 在检测到其中任何一个先决条件修改后，必须执行这个操作来更新photonphoton.dat。
  如果这个源代码保存在一个名为Makefle的文件中，执行make时就会找到这个文件。
  练习：创建makefile文件

特殊目标与清理操作（Makefile 中的 all 目标

all 是默认目标：

• 当你在命令行直接输入 make（不加参数）时，Makefile 会默认运行它的第一个目标。这个目标习惯上被命名为 all，用来整体更新项目。

1. **all 目标的作用：**
   • 目的： 确保项目的所有主要成果（顶级目标） 都是最新的。
   • 定义方式： 通常没有具体命令（动作）。它只列出它所依赖的所有其他主要目标（比如 photon_photon.dat, fig4.svg 等）。
   • 如何工作： 当你运行 make 或 make all 时：Make 会检查 all 列出的每个依赖项（目标文件）是否需要更新。如果需要，Make 会去执行对应依赖项的规则（命令）来更新那个目标文件。
   • 关键点： all 本身不执行最终的构建操作（比如编译、绘图），它只确保所有依赖项都被更新到了最新状态。
2. **all 的定义特点：**
   • 名称可自定义： 目标的名字 all 是习惯用法，但是你可以取别的名字（例如 build-all），但all是约定俗成的标准。
   • 通配符可用： 在定义 all 的依赖项时，可以使用通配符（如 *）来匹配多个目标文件

- **配置**：适配不同平台（Windows/Linux 等），设置编译、安装参数，使用 cmake 或 scons 等工具生成 Makefile，根据用户选项（如安装路径）和系统特性定制构建规则
- **编译**：将源码转成目标代码，需处理语言特性（如 Python 编译为字节码，C++ 生成机器码），调用编译命令（如 make）将源代码转为二进制文件，并链接依赖库到指定路径
- **链接**：整合依赖库，解决符号引用，生成可执行程序，推荐执行测试（如 make test），验证程序在本地环境是否构建成功，确保功能正常
- **安装**：部署程序到系统路径，配置环境变量（如 PATH、库路径）。
- 以用户的视角来看，相当于以下这四条命令：

```
~ $ .configure [options] (1)
~ $ make [options] (2)
~ $ make test (3)
~ $ [sudo] make install (4)
```

每个步骤所需的命令、标志和自定义内容，都要针对特定的计算机平台、用户和环境。也就是说，由makefle 定义的“动作”需要能够在不同平台或为不同用户执行不同的命令。

例如：对于一个C++程序，有三个用户，他们可能分别使用g++、clang、gcc。每个用户的命令都有所区别。因此必须设置makefile,让其检测机器上存在哪个编译器并相应地调整“操作”

所以接下来介绍的makefile配置工具尤为重要

- **自动配置工具**：CMake、Autotools 等，替代手动编写复杂 Makefile，适配多平台
- **优势**：检测系统环境（编译器、库依赖），生成适配 Makefile；降低跨平台构建难度
- **对比选择**：不同工具适用场景（CMake 对跨平台项目友好，Autotools 擅长 GNU 生态）

自动化工具：1.检测平台和架构特性。2.检测环境变量。3.检测可用的命令、编译器和库。4.接受用户输入。5.生成定制的 makefile。

常用的自动化系统构建工具：Cmake、AutoTools、Scons

不同的计算机平台架构不同，大多数软件就必须针对平台进行单独构建，为了定义makefile的目标，先决条件，行为，配置系统首先就需要确定系统平台。

操作系统有：windows，Linux，UNIX，移动端，嵌入式等等

除了平台架构不同会影响平台配置以外，不同的数据存储方式（32位和64位的不同）也会导致有差异，系统环境，安装的库，安装库的位置的不同也会有差异，影响构建。

因此，构建系统必须基于**不同用户的配置**做出调整，包括：1.使用什么编译器 2.安装的库版本 3.库的位置 4.ATH 和类似的环境变量有哪些目录 5.构建的项目中有哪些可选部分 6.使用什么编译器标志(调试构建、优化构建等)。

依赖配置

1.依赖是什么

如果软件 A 需要使用软件 B 的功能来完成自己的工作，那么软件 B 就是软件 A 的依赖。

举例： 软件 SuperPhysics 依赖 EssentialPhysics 和 ExtraPhysics。

2.安装软件的前提

安装一个软件之前，必须先安装好它所依赖的所有软件（库）。

3.构建失败的主要原因

找不到依赖库： 构建系统在系统里找不到需要的依赖库文件。

版本不对： 找到了依赖库，但库的版本不正确（太高、太低或不兼容）。

4.构建系统如何找库?

构建系统主要通过检查 **PATH、LD_LIBRARY_PATH 这类环境变量**所指定的路径来查找依赖库。

5.常见构建问题的根源

环境变量的库太多： 如果环境变量指定的路径里存在同一个库的多个版本，构建系统可能会链接到错误版本，导致错误。

环境变量的库不足： 如果环境变量指定的路径里缺少某个必需的依赖库，构建一定会失败。

6.保证构建成功的关键:

必须正确安装所有依赖库并确保构建系统能通过环境变量准确找到它们

构建流程挑战

- **依赖问题**：库版本不兼容、路径配置错误，需清晰梳理依赖链（如 Python 包、C++ 库）
- **平台差异**：不同系统（Windows/Linux）编译、安装指令不同，依赖配置工具适配
- **调试方法**：利用 make 日志、逐步执行目标，定位构建失败环节（如编译报错、依赖缺失）

当以上步骤都完成之后，就可以进入编译阶段

1. Makefile作用： 配置好的Makefile用于编译源代码。
2. 核心命令： Makefile中的命令主要是编译器命令（将源代码转换为机器可读的二进制程序）。
3. 默认目标（make）： 构建系统通常生成带默认目标的Makefile。执行make命令会：
   - 将所有源代码编译到本地临时目录。
   - 这相当于实际安装前的测试性构建。
4. 测试（make test）： 编译完成后，通常用make test测试构建结果。
5. 安装： 如果测试通过，构建系统进入安装步骤。

安装

1. **安装的重要性：** 项目能否方便地安装是吸引用户的关键因素。
2. **不同平台安装方式：**
   - **Windows：** 创建 `Setup.exe` 安装程序。
   - **Python：** 使用 `setup.py` 或相关发布工具。
   - **UNIX（源代码）：** 生成带 `install` 目标的 `makefile`，用户执行 `make install`。
3. **为何不用简单脚本安装?**
   - 用户可能需要**升级或卸载**软件。
   - **安装**程序通常由**开发者提供**。
   - **卸载**程序通常由**操作系统（包管理器）处理**。
   - 构建系统能更好地处理这些**平台相关的功能**（安装/卸载）。
4. **Linux打包的实践：** 创建软件包（如 RPM, DEB）时，
   - 不用普通 `make install`。用 `make DESTDIR=<临时目录> install` 将软件“安装”到临时目录。
   - 基于临时目录内容**创建软件包**和**卸载清单**。
5. **受限目录安装：** 如果配置时选择的安装目录（如 `/usr/local/`）需要管理员权限：
   - 必须使用 `sudo` 执行安装命令：`sudo make install`。

本章总结

- **知识沉淀**：掌握 make 工具、Makefile 编写，理解软件构建全流程（配置、编译、链接等）
- **实践方向**：在科研（论文自动化更新）、开发（项目构建）场景应用，尝试 CMake 等工具
- **价值延伸**：自动化构建提升效率，为复杂项目、持续迭代开发奠定基础