



中国海洋大学
OCEAN UNIVERSITY OF CHINA

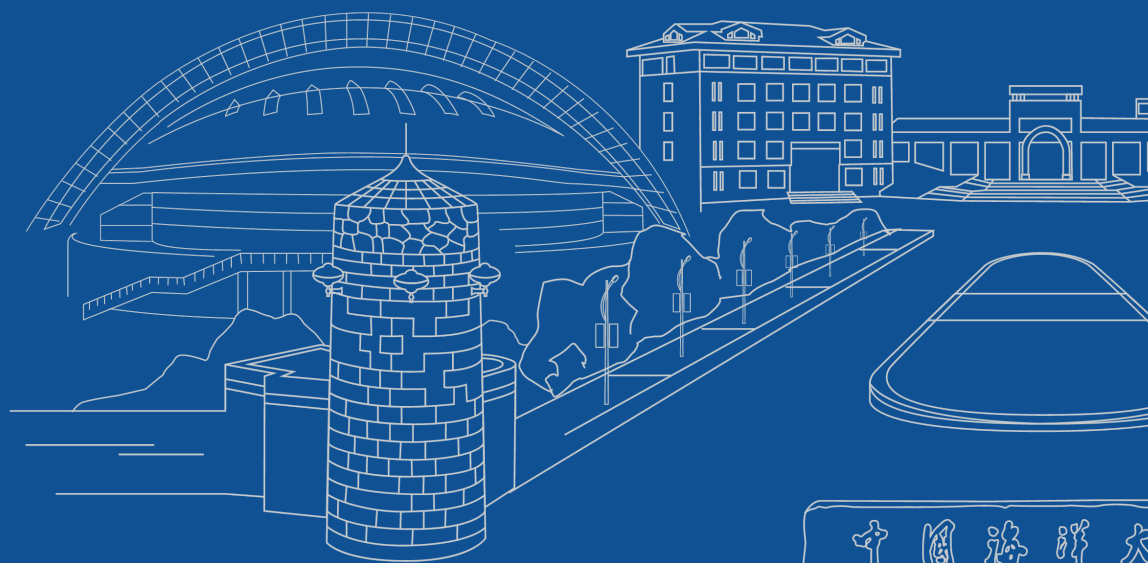
系统开发工具基础第四次实验报告

掌握 Shell 的基本用法 学习数据整理与练习 Vim

于景一

23090032047

信息科学与工程学部
计算机学院
计算机科学与技术专业



中国海洋大学

OUC, September 2024



中国海洋大学
OCEAN UNIVERSITY OF CHINA

系统开发工具基础第四次实验报告

掌握 Shell 的基本用法 学习数据整理与练习 Vim

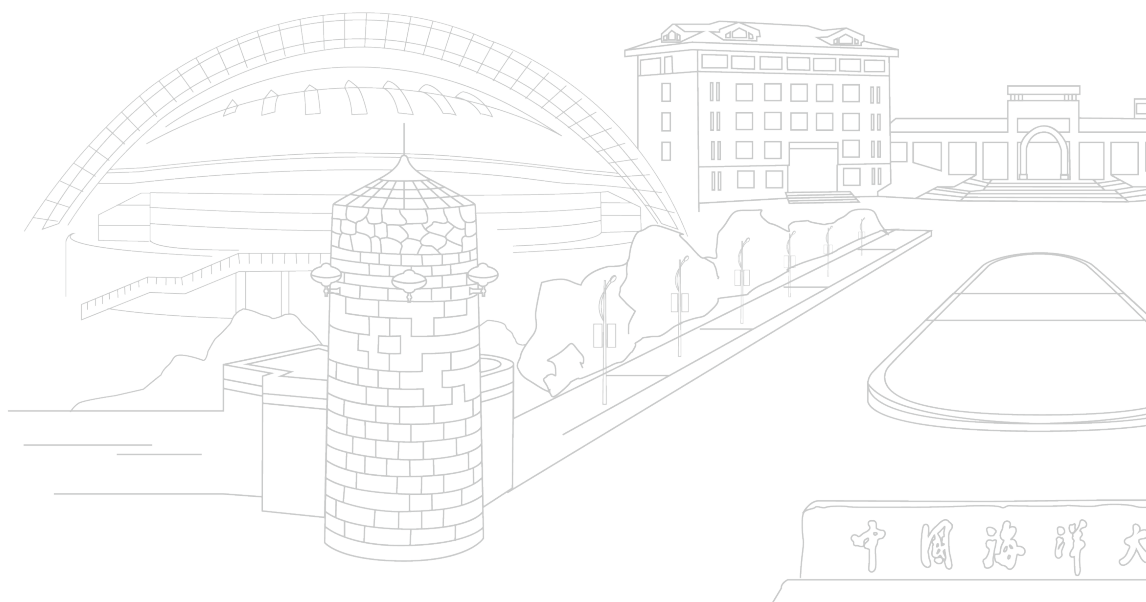
于景一

23090032047

指导老师: 周小伟

系统开发工具基础, 计算机学院

信息科学与工程学部
计算机学院
计算机科学与技术专业



OUC, September 2024

中国海洋大学

概述

本实验报告使用由于景一制作的 \LaTeX 模板完成。关于此模板的信息,您可以前往GitHub模板仓库具体了解。¹² 本文章使用的是“实验报告”模板。³

本实验报告是系统开发工具基础课程的第四次实验报告,主要内容为调试与性能分析、元编程、PyTorch 编程和大杂烩(进程守护、API、Docker 等)。我们以上所述的调试与性能分析、元编程主要面向 Linux 系统,PyTorch 编程则以本机 Windows 为实验环境。本次实验遵循体验为主的原则,主要提高了动手能力。至于其中具体用法,仍需后续系统学习。

本实验报告的仓库地址为 ToolBasics4 by jstar0。

调试与性能分析主要包括,使用专用调试工具和第三方日志系统结合进行开发过程的调试。我们主要使用了 GDB 和 LLDB 进行 C/C++ 调试分析,对于其他不同的程序,使用 log4cpp 和 spdlog 进行日志记录。考虑其他可视化调试工具,如 DTrace, Perfetto, Valgrind, htop。了解性能分析的基本方法。

元编程其实是一个流程,我们主要讨论了构建系统、持续集成系统等两个方面。在构建系统中,我们首先介绍了 Makefile,后面以 CMake 为例介绍使用,在持续集成系统中,我们以 GitHub Action 为例,讲解其中的 CI 流程方法。

PyTorch 是一个深度学习框架,我们主要学习 PyTorch 的基本使用方法。首先介绍 PyTorch 的基本概念,然后讲解 PyTorch 的基本使用方法,包括 Tensor 的使用、Module 的使用、Optimizer 的使用、Dataset 和 DataLoader 的使用。最后,我们讲解了 PyTorch 的 GPU 加速方法(基于 Windows)。

NOTE

知识之海是无边无际的,只是尽力游弋,就已倍感费时费力,然而学习的过程是美好的。本文十分惭愧地呈现了我在实验中对上述领域的浅薄了解,如有谬误还请批评斧正。

¹ 或您可直接搜索 GitHub 账号 @jstar0 了解更多

² 您请注意,本模板基于 LPPL v1.3c 分发,本项目在原模板 Polytechnic University of Leiria: LaTeX Thesis Template 的基础上进行了合法地大量二改,包括但不限于自定义风格、中文化支持、样式重定义、功能增加等。

³ 模板提供两种样式,一种为学术论文样式,另一种为实验报告样式,具体区别请检查 GitHub 仓库上的两个分支。

目录

目录	1
1 调试与性能分析	2
1.1 概述	2
1.2 专用调试工具	2
1.2.1 使用 GNU Debugger	2
1.2.2 使用 LLVM Debugger	3
1.3 第三方日志系统	5
1.3.1 简介	5
1.3.2 spdlog 的简单用例	5
1.4 可视化调试工具	5
1.4.1 简介	5
1.4.2 STrace 的简单用例	6
1.4.3 Perfetto 的简单用例	6
1.5 内存泄漏检测与性能分析	6
1.5.1 概述	6
1.5.2 体验 Valgrind	6
2 元编程	8
2.1 构建系统	8
2.1.1 Makefile	8
2.1.2 CMake	10
2.2 持续集成系统	11
2.2.1 概述	11
2.2.2 GitHub Action	11
3 PyTorch 编程	13
3.1 简介	13
3.2 安装与配置	13
3.3 基本张量并用 GPU 加速	13
3.4 构建一个简单的神经网络	15

调试与性能分析

1.1 概述

在 Linux 开发环境中，调试与性能分析是系统开发和优化过程中至关重要的环节。本实验围绕多种调试工具和性能分析技术展开，练习如何定位程序中的错误、优化性能瓶颈，并深入了解系统资源的使用情况。

Linux 系统的开放，使其拥有丰富的调试和性能分析工具，这些工具各有其适用场景与优缺点，实验中会详细讨论并尝试应用这些工具。

1.2 专用调试工具

1.2.1 使用 GNU Debugger

GDB (GNU Debugger) 一般面向 C/C++ 程序，是一个功能强大的调试工具。通过命令行提供强大的调试能力，允许用户设置断点、单步执行、检查变量值、修改内存状态等。实际上，我们在 C/C++ 的学习中，已经对 GDB 有所涉猎。

一般地，我们通过如1.1所示的方法，在 C 语言程序中启动 GDB，并通过 `run` 命令执行程序。在程序运行过程中，我们可以通过 `break` 命令设置断点，通过 `next` 命令单步执行，通过 `print` 命令查看变量值，通过 `watch` 命令监视变量值等。

```
1 # 有可能要先安装 gcc 和 gdb 工具
2 sudo apt install gcc gdb -y
3
4 # 用 echo 编写一个简单的 C 程序
5 echo '#include <stdio.h>
6 int main() {
7     int a = 1;
8     int b = 2;
9     int c = a + b;
10    printf("a + b = %d\n", c);
11    return 0;
12 }' > test.c
13
```

```
14 # 编译 C 程序
15 # 其中 -g 选项表示生成调试信息, 这样 GDB 才能正确调试
16 gcc -g test.c -o test
17 chmod +x test
18
19 # 启动 GDB 调试
20 gdb ./test
21
22 # 使用 list 命令可以查看源代码
23 (gdb) list
24 ## 测试中的程序源码被打印
25
26 # 在 GDB 中执行 run 命令
27 (gdb) run
28 ## 程序开始运行, 输出 a + b = 3
29
30 # 可以使用 break (b), next (n), step (s), print (p), continue (c) 等命令在 run 后进行调试
31
32 # 设置断点为 main 函数
33 # 或者我们设置断点为 Line 4
34 (gdb) b main
35 ## Breakpoint 1, main () at test.c:3
36 ## 3          int a = 0;
37 (gdb) b 4
38 ## Breakpoint 2 at 0x555555555515c: file test.c, line 4.
39
40 # 执行程序
41 (gdb) run
42 # 打印变量 a
43 (gdb) p a
44 ## $1 = 0
45 (gdb) c
46 ## Continuing.
47 ## Breakpoint 2, main () at test.c:4
48 ## 4          int b = 2;
49 # 此时再打印变量 a
50 (gdb) p a
51 ## $2 = 1
52 (gdb) c
53 ## Continuing.
54 ## a + b = 3
55 ## [Inferior 1 (process 16498) exited normally]
56
57 # 退出 GDB
58 (gdb) quit
```

代码 1.1: 使用 GDB 调试 C 程序的示例

1.2.2 使用 LLVM Debugger

LLDB (LLVM Debugger) 也是一个功能强大的调试工具, 面向 C/C++ 程序。与 GDB 相比, 是更为强大的替代工具, 基于 LLVM 架构, 启动速度更快, 支持现代 C++ 标准, 尤其在调试 C++ 项目时优势明显。

LLDB 的使用方法与 GDB 类似, 如 1.2 所示。我们可以通过 `lldb` 命令启动 LLDB, 通过 `run` 命令执行程序, 通过 `breakpoint set` 命令设置断点, 通过 `next` 命令单步执行, 通过 `print` 命令查看变量值等。

```
1  # 有可能要先安装 clang 和 lldb 工具
2  sudo apt install clang lldb -y
3
4  # 依然用 echo 编写一个简单的 C 程序
5  echo '#include <stdio.h>
6  int main() {
7      int a = 1;
8      int b = 2;
9      int c = a + b;
10     printf("a + b = %d\n", c);
11     return 0;
12 }' > test.c
13
14 # 编译 C 程序
15 clang -g test.c -o test
16 chmod +x test
17
18 # 启动 LLDB 调试
19 lldb ./test
20
21 # 使用 list 命令可以查看源代码
22 (lldb) list
23 ## 测试中的程序源码被打印
24
25 # 在 LLDB 中执行 run 命令
26 (lldb) run
27 ## 程序开始运行, 输出 a + b = 3
28
29 # 类似的, 可以使用 breakpoint set, next, step, print, continue 等命令在 run 后进行调试
30
31 # 设置断点为 main 函数
32 # 或者我们设置断点为 Line 4
33 (lldb) breakpoint set --name main
34 ## Breakpoint 1: where = test`main, address = 0x0000000100000f80
35 (lldb) breakpoint set --file test.c --line 4
36 ## Breakpoint 2: where = test`main + 20 at test.c:4, address = 0x0000000100000f94
37
38 # 执行程序
39 (lldb) run
40
41 # 打印变量 a
42 (lldb) print a
43 ## (int) $0 = 0
44
45 # 继续执行
46 (lldb) continue
47
48 # 此时再打印变量 a
49 (lldb) print a
50 ## (int) $1 = 1
51
52 # 继续执行
53 (lldb) continue
54 ## a + b = 3
55 ## Process 16498 exited with status = 0 (0x00000000)
56
57 # 退出 LLDB
58 (lldb) quit
```

代码 1.2: 使用 LLDB 调试 C 程序的示例

1.3 第三方日志系统

1.3.1 简介

第三方日志系统是一种记录程序运行状态的工具。在实际开发中，对于复杂的应用程序，实时跟踪每个函数的执行情况非常困难，因此集成第三方日志系统成为调试的重要手段。我们经常使用 *log4cpp* 和 *spdlog* 等日志库。

1.3.2 *spdlog* 的简单用例

在 C++ 程序中,我们使用 *spdlog* 库来记录日志。如1.3所示,我们可以通过 `spdlog::info`、`spdlog::warn`、`spdlog::error` 等方法记录不同级别的日志。

```
1  #include <spdlog/spdlog.h>
2
3  int main() {
4      spdlog::info("Program started.");
5      int a = 5;
6      int b = 10;
7      spdlog::debug("a = {}, b = {}", a, b);
8      int c = a + b;
9      spdlog::info("Result: {}", c);
10     spdlog::warn("This is a warning!");
11     spdlog::error("An error occurred!");
12     return 0;
13 }
```

代码 1.3: 使用 *spdlog* 记录日志的示例

如此,可以做到方便地与文件系统或控制台集成,并通过其异步日志记录功能,确保日志输出不会显著影响程序性能。在复杂应用程序中,适当使用日志记录能够大幅度简化问题排查流程。

若要具体地使用 *spdlog*, 我们需要先安装 *spdlog* 库, 然后在 *CMakeLists.txt*¹中添加如下内容:

```
1  find_package(spdlog REQUIRED)
2  target_link_libraries(${PROJECT_NAME} PRIVATE spdlog::spdlog)
```

1.4 可视化调试工具

1.4.1 简介

STrace 是一款强大的动态追踪工具, 允许开发者在程序的各个执行点插入探针, 从而捕捉系统调用、函数调用等底层信息。而 **Perfetto** 则专注于时间线可视化分析, 能够将程序运行时的每个事件绘制成直观的图表, 以帮助理解程序的执行顺序和时长。譬如说, 如

¹ 对 CMake 的具体介绍请见章节2。

果我们要调试一个多线程程序，使用 *Perfetto* 可以非常方便地定位线程调度问题，分析哪部分代码导致了性能瓶颈或死锁。

1.4.2 STrace 的简单用例

STrace 是一个 *Linux* 系统下的系统调用跟踪工具，我们可以通过 `strace` 命令来使用。如1.4所示，我们可以通过 `strace` 命令跟踪程序的系统调用，从而了解程序的执行情况。

```
1 # 使用 strace 跟踪程序
2 strace ./test
3 ## 输出系统调用信息
```

代码 1.4: 使用 STrace 跟踪 C 程序的示例

当然，`strace` 的功能远不止如此，我们可以通过 `strace` 命令的不同选项，来实现更多的功能。例如，我们可以通过 `-c` 选项统计系统调用的次数，通过 `-t` 选项显示时间戳，通过 `-p` 选项跟踪指定进程等。由于学习曲线过于陡峭，我们在此不再详细展开。

1.4.3 Perfetto 的简单用例

Perfetto 是一个 *Linux* 系统下的性能分析工具，我们可以通过 `perfetto` 命令来使用。如??所示，我们可以通过 `perfetto` 命令启动 *Perfetto*，并通过 UI 界面来查看程序的性能分析结果。

perfetto 是基于 **Chrome** 的 **Tracing** 实现，因此应该在具有图形界面的环境下使用。

1.5 内存泄漏检测与性能分析

1.5.1 概述

性能分析是系统开发过程中的重要环节，通过性能分析，我们可以找出程序中的性能瓶颈，优化程序的性能。而内存泄漏检测则是系统开发过程中的常见问题，通过内存泄漏检测，我们可以找出程序中的内存泄漏问题，避免程序的内存泄漏。

Valgrind 和 **htop** 是资源监控与分析的核心工具。*Valgrind* 是 *Linux* 上用于内存分析的知名工具，能够帮助开发者检测内存泄漏、无效内存访问、未初始化变量使用等问题。*htop* 则是实时资源监控工具，提供了一种直观的方式查看系统的 CPU、内存、I/O 等资源使用情况。

1.5.2 体验 Valgrind

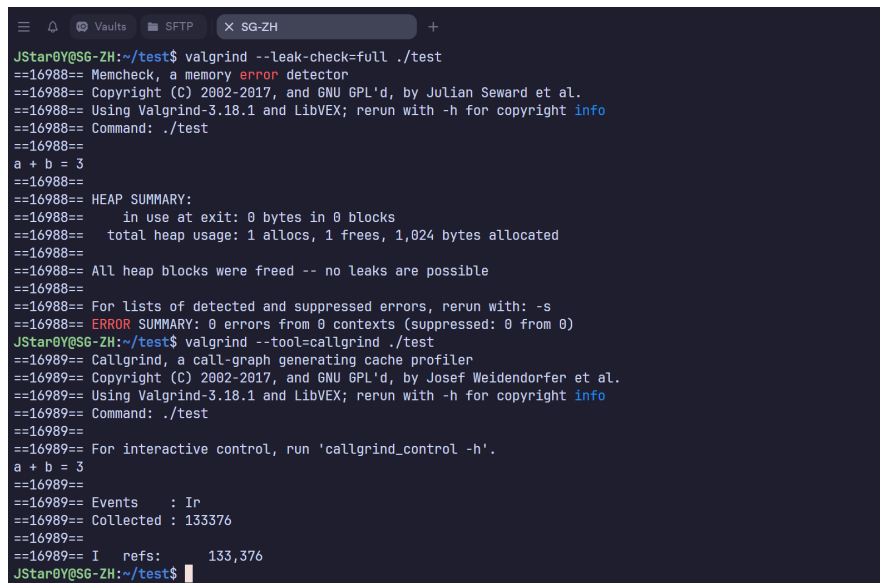
Valgrind 会生成一份详细的报告，指出每次内存分配和释放的情况，并指出潜在的内存问题。对于 C/C++ 程序，内存管理错误常常难以排查，*Valgrind* 通过其动态内存分析能力，为开发者提供了极具价值的诊断信息。除了内存分析，*Valgrind* 还包含了一些其他工具，例如 *callgrind*，用于分析程序的调用图和函数性能。

接下来，我们体验其内存检测能力与函数测试能力。请见代码1.5。

```
1 # 有可能要先安装 valgrind 工具
2 sudo apt install valgrind -y
3
4 # 对我们刚刚的 C 程序测试内存泄漏
5 valgrind --leak-check=full ./test
6
7 # 对我们刚刚的 C 程序测试函数调用链
8 valgrind --tool=callgrind ./test
```

代码 1.5: 使用 Valgrind 检测 C 程序的示例

具体执行结果见图1.1。



```
JStar0Y@SG-ZH:~/test$ valgrind --leak-check=full ./test
==16988== Memcheck, a memory error detector
==16988== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==16988== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==16988== Command: ./test
a + b = 3
==16988== HEAP SUMMARY:
==16988==   in use at exit: 0 bytes in 0 blocks
==16988==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==16988== All heap blocks were freed -- no leaks are possible
==16988== For lists of detected and suppressed errors, rerun with: -s
==16988== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
JStar0Y@SG-ZH:~/test$ valgrind --tool=callgrind ./test
==16989== Callgrind, a call-graph generating cache profiler
==16989== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==16989== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==16989== Command: ./test
a + b = 3
==16989== For interactive control, run 'callgrind_control -h'.
==16989== Events      : Ir
==16989== Collected : 133376
==16989== I    refs:    133,376
JStar0Y@SG-ZH:~/test$
```

图 1.1: Valgrind 的执行结果

元编程

2.1 构建系统

构建系统是开发流程的基础。其作用是通过一系列规则和指令自动化地将源代码编译为可执行文件、库文件或者其他输出目标。在小型项目中，手动编译和链接源文件可能已经足够，但是随着项目规模的增长，源文件之间的依赖关系变得错综复杂，手工编译不仅耗时耗力，还容易出错。这时，构建工具便显得尤为重要。构建工具如 `Makefile` 和 `CMake` 可以自动处理文件依赖，编译、链接源文件，并支持跨平台的构建需求，使项目的开发流程更加高效和可靠。

2.1.1 Makefile

`Makefile` 是 Linux 系统中最早期且广泛使用的构建工具之一。它依赖于 `GNU Make` 工具，通过定义一组目标 (*target*)、依赖关系 (*dependencies*) 和规则 (*rules*)，自动化地完成编译过程。典型的 `Makefile` 文件通常由多个部分组成，描述了从源文件生成目标文件的过程。

我们只能给出一个简单的 `Makefile` 示例，如代码2.1所示。

```
1 CC = gcc
2 CFLAGS = -Wall -g
3
4 SRCS = main.c utils.c
5 OBJS = $(SRCS:.c=.o)
6 TARGET = program
7
8 $(TARGET): $(OBJS)
9     $(CC) $(CFLAGS) -o $(TARGET) $(OBJS)
10
11 %.o: %.c
12     $(CC) $(CFLAGS) -c $< -o $@
13
14 clean:
15     rm -f $(OBJS) $(TARGET)
```

代码 2.1: `Makefile` 编译 C 项目的示例

在这个 `Makefile` 中，`CC` 表示编译器（此处为 `gcc`），`CFLAGS` 定义了编译器的选项，包括显示所有警告（`-Wall`）和生成调试信息（`-g`）。`SRCS` 列出了所有源文件，而 `OBJS` 则是

由源文件生成的目标文件。`$(TARGET)` 是最终的可执行文件。

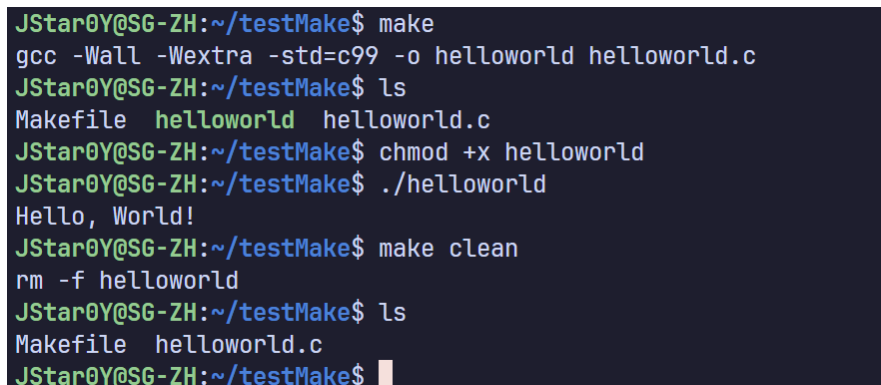
执行 `make` 命令，`Make` 就会自动计算每个目标文件的依赖，并根据规则编译源代码。`clean` 命令用于清理生成的目标文件和可执行文件。

下面我们给出一个使用 `Makefile` 编译一个 Hello World C 程序的示例，如代码2.2所示。

```
1  # 安装 make 工具
2  sudo apt install build-essential -y
3
4  # 假设目录中已有了 helloworld.c 文件
5  # 创建 Makefile 文件
6  echo 'CC=gcc
7  CFLAGS=-Wall -Wextra -std=c99
8
9  all: helloworld
10
11 helloworld: helloworld.c
12 ^I$(CC) $(CFLAGS) -o helloworld helloworld.c
13
14 clean:
15 ^Irm -f helloworld' >> Makefile
16
17 # 执行 make 命令
18 make
19 ## 看到 gcc -Wall -Wextra -std=c99 -o helloworld helloworld.c 说明编译成功
20 chmod +x helloworld
21 ./helloworld
22 ## Hello, World!
23
24 # 清理编译文件
25 make clean
26 ## rm -f helloworld 此时 helloworld 文件已被删除
```

代码 2.2: 使用 `Makefile` 编译 Hello World C 程序

具体运行效果见图2.1。



```
JStar0Y@SG-ZH:~/testMake$ make
gcc -Wall -Wextra -std=c99 -o helloworld helloworld.c
JStar0Y@SG-ZH:~/testMake$ ls
Makefile helloworld helloworld.c
JStar0Y@SG-ZH:~/testMake$ chmod +x helloworld
JStar0Y@SG-ZH:~/testMake$ ./helloworld
Hello, World!
JStar0Y@SG-ZH:~/testMake$ make clean
rm -f helloworld
JStar0Y@SG-ZH:~/testMake$ ls
Makefile helloworld.c
JStar0Y@SG-ZH:~/testMake$
```

图 2.1: 使用 `Makefile` 编译 Hello World C 程序的运行效果

Make 的缺点显而易见。虽然 `Makefile` 的简单、灵活和易于扩展使其在许多项目中被广泛使用。然而，当项目变得更加复杂且跨平台需求增多时，**Makefile** 的局限性开始显现。特别是，当项目需要在不同的操作系统或编译器之间切换时，手动管理 `Makefile` 的复杂性变得不可控。

2.1.2 CMake

CMake 是一款开源的跨平台构建工具，它的设计初衷是为了解决 Makefile 的跨平台问题。在 Linux 系统中，**CMake** 生成 Makefile，而在 Windows 上，它可以生成 Visual Studio 的解决方案文件，或者也生成 Makefile。**CMake** 的核心思想是将构建过程抽象化，通过 **CMake** 脚本定义项目的构建逻辑，使得同一套脚本可以在不同平台上复用。也就是说，**CMake** 是一个构建系统生成器，它可以根据用户提供的配置信息生成适用于不同平台的构建系统。而 **CMake** 本身的配置是更为简单和方便的（至少个人感觉）。

CMakeLists.txt 文件是 **CMake** 的核心配置文件，用于定义项目的构建步骤。我们在夏季小学期的实验中，就用了 **CMake** 来构建我们的项目。其 **CMakeList.txt** 的配置如代码2.3所示。

```

1 cmake_minimum_required(VERSION 3.12)
2 project(Game)
3
4 set(CMAKE_CXX_STANDARD 17)
5 set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fexec-charset=GBK -static-libstdc++ -static-libgcc -g")
6
7 file(GLOB SOURCES "src/*.cpp")
8
9 include_directories(${PROJECT_SOURCE_DIR}/include)
10
11 add_executable(Game ${SOURCES})

```

代码 2.3: CMake 配置文件示例

在这个 **CMake** 配置文件中，**cmake_minimum_required** 指定了 **CMake** 的最低版本要求，**project** 定义了项目的名称。接着，**set** 命令定义了编译器标准和编译选项，**file** 命令用于查找源文件，**include_directories** 命令用于指定头文件目录，**add_executable** 命令用于定义可执行文件。

请看 **CMake** 在我们具体的项目中的使用，如图2.2所示。

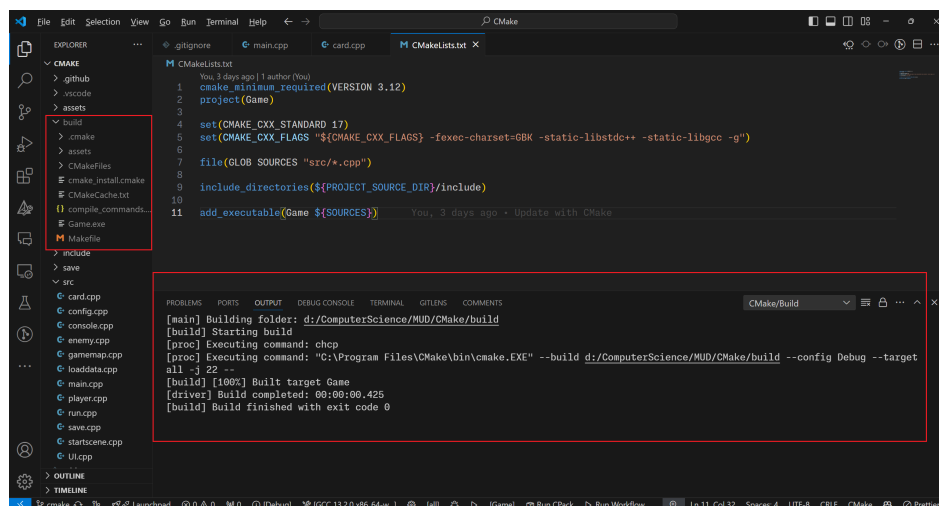


图 2.2: 使用 CMake 构建项目的示例

此外，在大型项目中，**CMake** 允许开发者将不同的构建任务分成多个 **CMakeLists** 文件，方便管理。**CMake** 还支持第三方库的查找和自动化集成，例如通过 **find_package** 命

令自动查找并链接外部依赖库。对于跨平台项目，**CMake** 能够根据不同平台自动调整编译选项和链接方式，使得开发者无需为每个平台单独编写构建脚本。

特别地，我们要介绍 **Ninja**，这款构建工具类似于 **Makefile**，但是与传统的 *Makefile* 相比，*Ninja* 能够更高效地处理并行编译任务，从而显著减少编译时间。它正是一款与 **CMake** 集成良好的工具，专为提高大规模项目的构建速度而设计。在实际开发中，使用 `cmake -G Ninja` 可以生成 *Ninja* 的构建文件。

2.2 持续集成系统

2.2.1 概述

持续集成（**Continuous Integration, CI**）是一种软件开发实践，其核心思想是频繁地将代码集成到主干分支，通过自动化构建和测试，尽早发现和解决代码集成引入的问题。持续集成的目标是减少开发周期，提高软件质量，降低风险。持续集成系统是实现持续集成的关键工具，可以自动化构建、测试和部署过程，提高开发效率和代码质量。

2.2.2 GitHub Action

GitHub Action 是 **GitHub** 提供的一款持续集成工具，它允许开发者在 *GitHub* 仓库中配置自动化构建、测试和部署任务。**GitHub Action** 的核心概念是 *Workflow*，即一组自动化任务的有序执行流程。每个 *Workflow* 由一个或多个 *Job* 组成，每个 *Job* 包含一个或多个 *Step*。每个 *Step* 是一个独立的任务，可以是构建、测试、部署等操作。

我们在夏季小学期的项目中，实际上在完成 **CMake** 构建后，就用了 *GitHub Action* 来自动化构建和测试我们的项目。我们的 `.github/workflows` 目录下的 `cmake-single-platform.yml` 文件如代码2.4所示。

```
1 name: CMake on a single platform
2
3 on:
4   push:
5     branches: [ "main" ]
6   pull_request:
7     branches: [ "main" ]
8
9 env:
10   # Customize the CMake build type here (Release, Debug, RelWithDebInfo, etc.)
11   BUILD_TYPE: Release
12
13 jobs:
14   build:
15     # Runs on Windows platform
16     runs-on: windows-latest
17
18     steps:
19       # Step 1: Check out the code from the repository
20       - uses: actions/checkout@v4
21
22       # Step 2: Install necessary dependencies
23       - name: Install Dependencies
```

```

24     run: |
25         choco install cmake
26         choco install ninja # If using Ninja generator, you can adjust based on your needs
27
28     # Step 3: Configure CMake (use the 'build' directory)
29     - name: Configure CMake
30       run: cmake -B "${github.workspace}"/build -S "${github.workspace}" -G "Ninja"
31           ↪ -DCMAKE_BUILD_TYPE=${env.BUILD_TYPE}
32
33     # Step 4: Build your project using CMake
34     - name: Build
35       run: cmake --build "${github.workspace}"/build --config ${env.BUILD_TYPE}
36
37     # Step 5: Run tests using CTest
38     - name: Test
39       working-directory: "${github.workspace}"/build
40       run: ctest -C ${env.BUILD_TYPE} --output-on-failure
41
42     - name: Prepare Artifacts
43       run: |
44         mkdir "${github.workspace}"/artifacts
45         copy "${github.workspace}"/build/Game.exe "${github.workspace}"/artifacts/Game.exe
46         xcopy /E /I "${github.workspace}"/assets "${github.workspace}"/artifacts/assets
47
48     - name: Upload Artifacts
49       uses: actions/upload-artifact@v4
50       with:
51         name: game-build
52         path: "${github.workspace}"/artifacts

```

代码 2.4: GitHub Action 配置文件示例

在其中包含了自动安装 CMake、Ninja、配置 CMake、构建项目、运行测试、准备构建产物和上传构建产物等步骤。具体运行效果见图2.3。

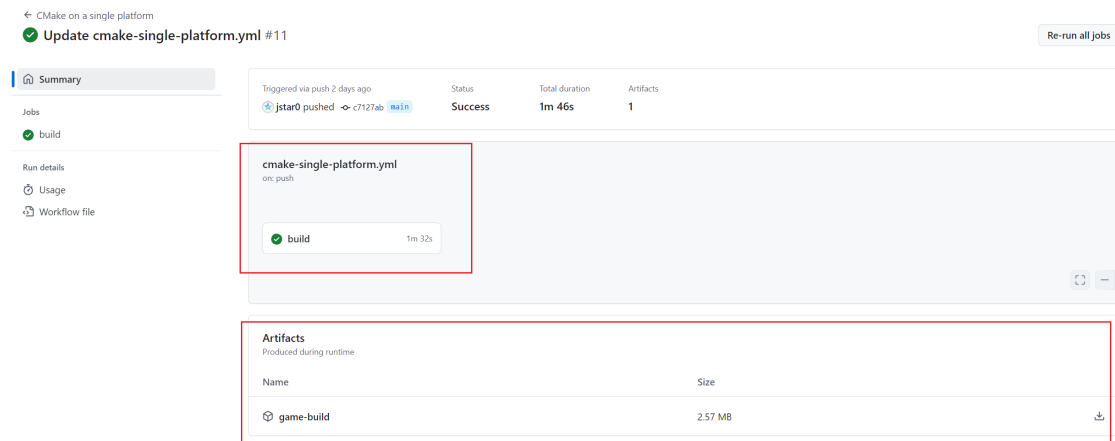


图 2.3: 使用 GitHub Action 构建项目的示例

您也可以前往Repo Workflows 页面查看具体的运行效果。

PyTorch 编程

3.1 简介

PyTorch 是一个以灵活性和易用性著称的深度学习框架，它的设计目标是支持动态计算图。与静态计算图的框架（如 TensorFlow 的早期版本）不同，*PyTorch* 的计算图是在每次前向传播时动态生成的，故而它在进行调试和处理复杂模型时具有更大的灵活性。**PyTorch** 的基本构件是张量（**tensor**），它是一个多维数组，类似于 NumPy 的数组，但不同之处在于 PyTorch 的张量可以使用 **GPU** 进行加速计算。本次实验，我们在 Windows 系统上配备了 NVIDIA 显卡的情况下，就可以通过 **CUDA** 加速神经网络的训练过程。

3.2 安装与配置

需要首先安装支持 **CUDA** 的 **PyTorch** 版本。首先应该确定自己的显卡型号，可以使用命令 `nvidia-smi` 查看，其中包含了 **CUDA** 版本，然后前往 PyTorch 官网查看支持自己的版本，找到对应的安装命令。例如，对于 CUDA 12.4，可以使用如下命令安装：

```
1 pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu124
```

若未安装过 **CUDA**，还需要前往 NVIDIA 官网下载并安装。

安装过程会比较耗时，安装完成后，可以检查是否安装成功：

```
1 python3 -c "import torch; print(torch.__version__)"
```

成功安装的输出如图 3.1 所示。

3.3 基本张量并用 GPU 加速

我们可以创建一个 2 行 3 列的随机张量，如果 `torch.cuda.is_available() == True` 函数满足（判断是否有可用的 GPU），将张量移动到 GPU 上，加速随后的计算操作。示例如代码 3.1 所示。

3.4 构建一个简单的神经网络

这部分内容过于专业与复杂，我们只得借鉴他人的成功实践。

例子中，我们定义了一个包含三层的简单全连接神经网络，输入维度为 10，输出维度为 1。forward 函数定义了前向传播的过程，使用 ReLU 激活函数来引入非线性。模型初始化后，我们同样通过判断是否有可用的 GPU，将整个网络结构移动到 GPU 上。这种情况下，网络的所有计算操作都将在 GPU 上完成，极大地提高了计算速度。示例如代码 3.2 所示。

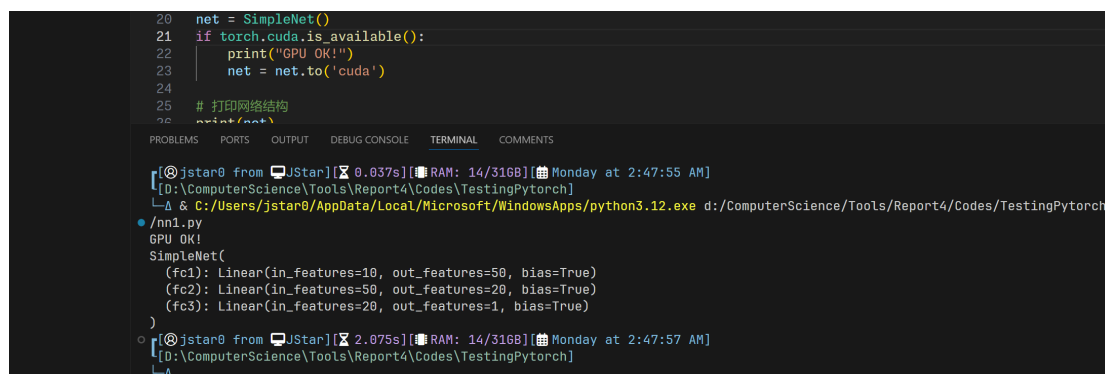
```

1  import torch
2  import torch.nn as nn
3  import torch.optim as optim
4
5  # 定义一个简单的神经网络
6  class SimpleNet(nn.Module):
7      def __init__(self):
8          super(SimpleNet, self).__init__()
9          self.fc1 = nn.Linear(10, 50)
10         self.fc2 = nn.Linear(50, 20)
11         self.fc3 = nn.Linear(20, 1)
12
13     def forward(self, x):
14         x = torch.relu(self.fc1(x))
15         x = torch.relu(self.fc2(x))
16         x = self.fc3(x)
17         return x
18
19 # 初始化网络并将其移动到 GPU
20 net = SimpleNet()
21 if torch.cuda.is_available():
22     net = net.to('cuda')
23
24 # 打印网络结构
25 print(net)

```

代码 3.2: PyTorch 构建简单神经网络

如果一切顺利，则有如图 3.3 所示的输出。



```

20 net = SimpleNet()
21 if torch.cuda.is_available():
22     print("GPU OK!")
23     net = net.to('cuda')
24
25 # 打印网络结构
26 print(net)

```

PROBLEMS PORTS OUTPUT DEBUG CONSOLE TERMINAL COMMENTS

```

[!@jstar0 from JStar][X 0.037s][RAM: 14/316B][Monday at 2:47:55 AM]
[D:\ComputerScience\Tools\Report4\Codes\TestingPytorch]
Δ & C:\Users\jstar0\AppData\Local\Microsoft\WindowsApps\python3.12.exe d:/ComputerScience/Tools/Report4/Codes/TestingPytorch
• /nn1.py
GPU OK!
SimpleNet(
  (fc1): Linear(in_features=10, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=20, bias=True)
  (fc3): Linear(in_features=20, out_features=1, bias=True)
)
[!@jstar0 from JStar][X 2.075s][RAM: 14/316B][Monday at 2:47:57 AM]
[D:\ComputerScience\Tools\Report4\Codes\TestingPytorch]
Δ

```

图 3.3: PyTorch 构建简单神经网络成功

最后，我们尝试如何对神经网络送入数据。PyTorch 提供了强大的数据处理模块 `torch.utils.data`，用于加载、处理数据，并将数据批量送入神经网络进行训练。以下示例展示了如何使用 PyTorch 的 `DataLoader` 加载数据，并将其送入网络进行前向传播，见代码 3.3。

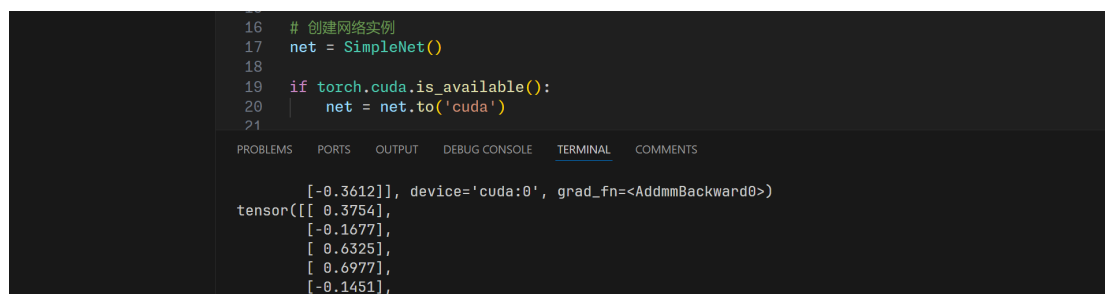
```

1  import torch
2  import torch.nn as nn
3  from torch.utils.data import DataLoader, TensorDataset
4
5  # 定义一个最简单的神经网络
6  class SimpleNet(nn.Module):
7      def __init__(self):
8          super(SimpleNet, self).__init__()
9          # 线性层的输入大小为 10, 输出大小为 1
10         self.fc = nn.Linear(10, 1)
11
12     def forward(self, x):
13         # 前向传播: 直接输出线性层的结果
14         return self.fc(x)
15
16 # 创建网络实例
17 net = SimpleNet()
18
19 if torch.cuda.is_available():
20     net = net.to('cuda')
21
22 # 创建一些随机数据
23 inputs = torch.randn(100, 10) # 100 个样本, 每个样本 10 个特征
24 targets = torch.randn(100, 1) # 对应的目标值
25
26 # 创建数据集和数据加载器
27 dataset = TensorDataset(inputs, targets)
28 dataloader = DataLoader(dataset, batch_size=16, shuffle=True)
29
30 # 在训练循环中加载数据并进行前向传播
31 for batch_inputs, batch_targets in dataloader:
32     if torch.cuda.is_available():
33         batch_inputs, batch_targets = batch_inputs.to('cuda'), batch_targets.to('cuda')
34
35     outputs = net(batch_inputs)
36     print(outputs)

```

代码 3.3: PyTorch 使用 *DataLoader* 加载数据

我在示例代码所做的工作是，创建一个最简单的神经网络 **SimpleNet**，然后将其实例化为 **net**，并移动到 GPU 上。其余部分的训练参考了网上的代码。如果一切顺利，我们就可以看到如图 3.4 所示的输出。



```

16  # 创建网络实例
17  net = SimpleNet()
18
19  if torch.cuda.is_available():
20      net = net.to('cuda')
21
22  # 创建一些随机数据
23  inputs = torch.randn(100, 10) # 100 个样本, 每个样本 10 个特征
24  targets = torch.randn(100, 1) # 对应的目标值
25
26  # 创建数据集和数据加载器
27  dataset = TensorDataset(inputs, targets)
28  dataloader = DataLoader(dataset, batch_size=16, shuffle=True)
29
30  # 在训练循环中加载数据并进行前向传播
31  for batch_inputs, batch_targets in dataloader:
32      if torch.cuda.is_available():
33          batch_inputs, batch_targets = batch_inputs.to('cuda'), batch_targets.to('cuda')
34
35      outputs = net(batch_inputs)
36      print(outputs)

```

[-0.3612]], device='cuda:0', grad_fn=<AddmmBackward0>)

tensor([[0.3754],

[-0.1677],

[0.6325],

[0.6977],

[-0.1451],

图 3.4: PyTorch 使用 *DataLoader* 加载数据成功

