课程作业报告

课程名称：系统工具开发基础

学生姓名：张家玮

学号：23020007157

日期：2024 年 9 月 13 日

目录

Chapter 1

调试与性能分析

1.1 shellcheck

由于我们是Windows系统，我们可以从官网上下载对应版本的shellcheck，然后手动添加到环境变量中，然后我们在Git Bash中可以使用对应的shellcheck命令来查找.sh文件中出现的错误。或者我们可以在vscode编辑器中加入shellcheck插件，也可以让.sh脚本文件能够通过高亮等来给我们提示错误。

1.2 查询登录信息

我们这里使用Windows系统来实际操作一下查询：

1. 打开事件查看器：按下Win+r，输入eventvwr.msc，然后按回车打开事件查看器。
2. 导航到安全日志：在左侧面板展开Windows日志。选择安全性。
3. 过滤登录事件：在筛选当前日志中，在事件ID中输入4624来查看成功登录的事件。

1.3 性能分析

1.3.1 快速排序和插入排序的性能比较

我们在这里编写一个程序，来分析一下插入排序和快速排序性能。在程序中使用到$cProfile$和$line_profiler$。
程序运行结果如下

```
Starting insertion sort profiling...
Starting quick sort profiling...
Starting line profiler for insertion sort...
Line profiler stats for insertion sort:
Timer unit: 1e-07 s

Total time: 0.317392 s
File: c:\Users\23724\Desktop\系统工具开发\python\week4\compare.py
Function: profile_insertion_sort at line 29

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    29                                           @profiler
    30                                           def profile_insertion_sort():
    31         2     106056.0  53028.0      3.3      arr = [random.randint(0, 100
0) for _ in range(1000)]
    32         2    3067862.0    2e+06     96.7      insertion_sort(arr.copy())

Total time: 0.0145349 s
File: c:\Users\23724\Desktop\系统工具开发\python\week4\compare.py
Function: profile_quick_sort at line 34

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    34                                           @profiler
    35                                           def profile_quick_sort():
    36         1      56549.0  56549.0     38.9      arr = [random.randint(0, 100
0) for _ in range(1000)]
    37         1      88800.0  88800.0     61.1      quick_sort(arr.copy())

Starting line profiler for quick sort...
Line profiler stats for quick sort:
Timer unit: 1e-07 s
```

再结合我们的分析:

插入排序法中，时间复杂度最坏情况：On,平均情况：On,最好情况：On. 空间复杂度为O1（原地排序，不需要额外的存储空间）

快速排序法中，时间复杂度最坏情况：On,平均情况：$Onlogn$,最好情况：$Onlogn$. 空间复杂度为O$logn$（递归调用栈的空间）. 所以我们可以得出结论：

插入排序：简单易实现，适合小规模或基本有序的数据，但在大规模数据时效率较低。

快速排序：在大多数情况下表现优异，适合大规模数据，但在最坏情况下性能可能下降。

1.3.2 快速排序和插入排序的内存消耗比较

我们编写程序利用，$memory_profiler$ 来检查内存消耗。利用这个库我们可以在终端中看到每个函数的内存使用情况。如下图中展示的一小部分：

```
Filename: c:\Users\23724\Desktop\系统工具开发\python\week4\compare_momery.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    16     32.1 MiB     28.1 MiB         605   @profile
    17                                         def quick_sort(arr):
    18     32.1 MiB     -4.0 MiB         605       if len(arr) <= 1:
    19     32.1 MiB     -2.0 MiB         303           return arr
    20     32.1 MiB     -1.9 MiB         302       pivot = arr[len(arr) // 2]
    21     32.1 MiB    -27.9 MiB        5967       left = [x for x in arr if x <
pivot]
    22     32.1 MiB    -27.9 MiB        5967       middle = [x for x in arr if x
== pivot]
    23     32.1 MiB    -27.9 MiB        5967       right = [x for x in arr if x >
 pivot]
    24     32.1 MiB     -2.0 MiB         302       return quick_sort(left) + midd
le + quick_sort(right)
```

总结：

插入排序：通常使用较少的额外内存，因为它是一个原地排序算法（空间复杂度为 O（1））。

快速排序：由于在递归和分割阶段创建了多个新列表（left, middle, right），通常使用更多的内存，特别是在处理较大数组时。

1.4 查看调用次数

我们可以通过编写程序来在终端中显示出课程所给的斐波那契数列的各个函数调用次数。

如下图

```
Function call counts:
fib9: 1 times
fib8: 1 times
fib7: 2 times
fib6: 3 times
fib5: 5 times
fib4: 8 times
fib3: 13 times
fib2: 21 times
fib1: 34 times
fib0: 21 times
```

```
Function call counts:
fib9: 1 times
fib8: 1 times
fib7: 1 times
fib6: 1 times
fib5: 1 times
fib4: 1 times
fib3: 1 times
fib2: 1 times
fib1: 1 times
fib0: 1 times
```

可以看到左图为原来的程序调用次数，右图为把注释中的函数加上的调用次数。

Chapter 2

元编程

2.1 构建系统

构建系统用于自动化编译、链接和打包代码的过程。元编程在构建系统中主要通过定义构建规则和自动化流程来提高灵活性和效率。我们这里展示使用cmake来构建系统，它使用CMakeLists.txt文件来定义构建规则。

如下图中有一个简单的示例

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.10)

project(MyProject)

set(CMAKE_CXX_STANDARD 11)

# 查找所有的源文件
file(GLOB SOURCES "*.cpp")

# 创建可执行文件
add_executable(my_program ${SOURCES})
```

在这个示例中，file（GLOB SOURCES ”*.cpp”）用于自动查找当前目录下的所有.cpp源文件，并将其包含到构建过程中。

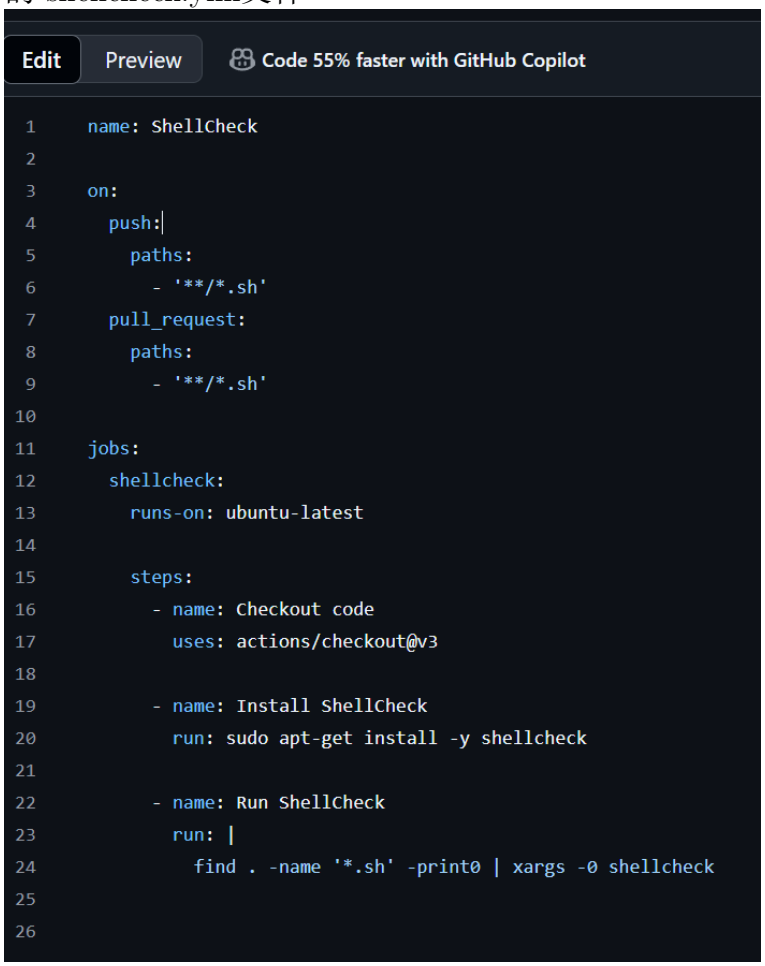
2.2 依赖管理

依赖管理用于自动化处理项目中各种库和工具的版本和兼容性问题。元编程在这里用于自动生成和管理依赖关系。我们这里尝试使用Python的pipreqs自动生成requirements.txt，pipreqs是一个工具，用于根据项目中实际使用的包生成requirements.txt文件。 pipreqs会扫描指定路径下的Python文件，并生成包含实际使用包及其版本的requirements.txt文件。这种方法避免了手动维护requirements.txt的麻烦。我们可以使用如下指令来进行这项操作：

```
pip install pipreqs
pipreqs /path/to/your/project
```

2.3 持续集成系统

持续集成系统用于自动化构建、测试和部署过程，以确保代码质量和稳定性。元编程在CI系统中可以帮助自动化配置和管理CI流程。这里我们添加一个 GitHub Action 到该仓库，对仓库中的所有 shell 文件执行 shellcheck。我们可以再GitHub中添加如下的 shellcheck.yml文件



这样就可以对所有的shell脚本执行shellcheck。

2.4 自动化构建和测试

我们可以设计一个shell脚本来对简单的C项目进行自动化构建和测试：

```
# 自动化构建和测试脚本
echo "Building project..."
gcc -o my_program main.c util.c

if [ $? -ne 0 ]; then
  echo "Build failed"
  exit 1
fi

echo "Running tests..."
./my_program

if [ $? -ne 0 ]; then
  echo "Tests failed"
  exit 1
fi

echo "Build and tests succeeded"
```

2.5 自动生成函数

在python中我们可以利用元编程的特点自动生成函数：

```python
def create_operation_function(operation):
    def operation_function(x, y):
        if operation == 'add':
            return x + y
        elif operation == 'subtract':
            return x - y
        elif operation == 'multiply':
            return x * y
        elif operation == 'divide':
            return x / y
        else:
            raise ValueError("Unsupported operation")
    return operation_function

if __name__ == "__main__":
    # 动态生成不同的操作函数
    add = create_operation_function('add')
    subtract = create_operation_function('subtract')
    multiply = create_operation_function('multiply')
    divide = create_operation_function('divide')

    # 使用这些生成的函数
    print("Addition: 5 + 3 =", add(5, 3))
    print("Subtraction: 5 - 3 =", subtract(5, 3))
    print("Multiplication: 5 * 3 =", multiply(5, 3))
    print("Division: 6 / 3 =", divide(6, 3))
```

2.6 模板

在C++中的模板也体现了元编程的思想。这里我们尝试使用模板来进行阶乘的计算。

如图：

```cpp
template<int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {
    static const int value = 1;
};

int main() {
    int result = Factorial<5>::value;
}
```

Chapter 3

大杂烩

3.1 修改键位映射

作为一名程序员，键盘是我们的主要输入工具。在Windows系统中我们可以下面的操作来进行

1. 打开注册表编辑器
2. 导航到键盘映射设置
3. 添加或修改键盘映射
4. 重启计算机

根据上面的步骤可以让我们来修改适合自己的键盘映射。

3.2 守护进程

守护进程（Daemon）是指在计算机系统中后台运行的进程，它们通常在系统启动时自动启动，并在整个系统运行期间持续运行。
有如下特点：

- 后台运行
- 长时间运行
- 无终端
- 系统服务

在Windows系统中我们可以使用sc命令来管理这些服务，如下：

- sc start #启动服务

- sc stop #停止服务
- sc restart #重启服务
- sc query #查看服务状态

3.3 常见命令行标志参数及模式

1.标志（Flags）,例如：

- -h 或 –help：显示帮助信息。
- -v 或 –version：显示程序的版本信息。
- -f 或 –file：指定文件。
- -r 或 –recursive：递归操作。
- -q 或 –quiet：减少输出信息（安静模式）。

2.参数（Arguments）,例如：

- ls -l /path/to/dir：/path/to/dir 是参数，-l 是标志，表示长格式列出文件。
- python script.py input.txt：input.txt 是参数，script.py 是脚本文件。

3.模式（Modes）,例如：grep：搜索文件内容

- -i：忽略大小写
- -n：显示行号

3.4 Markdown

Markdown 是一种轻量级的标记语言，用于格式化文本。它设计得简洁易读，让文档既可以以纯文本的形式编写，也可以通过渲染工具显示为格式化的文本。
常见的语法如下：

1. 标题:Markdown 支持六级标题，使用 # 符号表示不同的标题级别。
2. 段落和换行:段落之间需要一个空行。要创建换行符，可以在行末添加两个或更多空格，然后按回车。
3. 强调：斜体用单个星号或下划线包裹文本。粗体用双星号或双下划线包裹文本。粗斜体用三个星号或三个下划线包裹文本。
4. 列表：无序列表使用星号、加号或减号。有序列表使用数字加点等。
5. 引用：使用 > 符号表示引用。

3.5 GitHub

GitHub 是一个广泛使用的代码托管和版本控制平台，基于 Git 系统。它提供了许多功能来协作开发、跟踪版本和管理代码库。

1. 账户：在 GitHub 上，你需要一个账户来创建和管理仓库、参与项目、提交代码等。

2. 仓库：仓库是用于存储和管理项目文件的地方。每个仓库包含一个项目的所有文件及其版本历史。

GitHub主要可以完成一下功能：

- Issues：用于追踪 bug、功能请求和任务。每个问题都可以分配给特定的人员，并附带标签和截止日期。

- Pull Requests：代码审查和合并的机制。开发者通过 pull request 提交更改，请求将其合并到主分支或其他分支。其他团队成员可以审查代码、添加评论、要求更改。

- Actions：GitHub Actions 是一个持续集成和持续部署（CI/CD）服务。你可以设置自动化工作流来构建、测试和部署代码。

- Wiki：用于为项目创建和维护文档。每个仓库可以有一个或多个 Wiki 页面，以便记录项目的使用方法、开发进度等信息。

- Projects：项目管理工具，可以创建看板（Kanban board）来组织和跟踪任务、问题和工作进度。

- Releases：管理软件发布的功能，可以创建版本发布，附带发布说明和变更日志。

Chapter 4

pytorch入门

PyTorch 是一个强大的深度学习框架，广泛应用于各种机器学习和人工智能任务。我们可以用其来完成很多任务。下面是一些简单应用。

4.1 向量运算

```python
import torch

# 创建向量
vector1 = torch.tensor([1.0, 2.0, 3.0])
vector2 = torch.tensor([4.0, 5.0, 6.0])

# 向量加法
result_add = vector1 + vector2
print(f"向量加法结果: {result_add}")  # 输出: tensor([5., 7., 9.])

# 向量点积
result_dot = torch.dot(vector1, vector2)
print(f"向量点积结果: {result_dot.item()}")  # 输出: 32.0

# 向量的逐元素乘积
result_elementwise = vector1 * vector2
print(f"逐元素乘积结果: {result_elementwise}")  # 输出: tensor([ 4., 10., 18.])
```

4.2 矩阵乘法

```python
import torch

# 创建两个矩阵
A = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
B = torch.tensor([[5.0, 6.0], [7.0, 8.0]])

# 矩阵乘法
C = torch.matmul(A, B)
print("Matrix Multiplication Result:\n", C)
```

4.3 计算梯度

```python
import torch

x = torch.tensor(2.0, requires_grad=True)

y = x**2 + 2*x + 1

# 计算梯度
y.backward()

# 输出梯度 (dy/dx)
print(f'The gradient of y with respect to x is: {x.grad.item()}')
```

4.4 线性回归

```python
import torch
import torch.nn as nn
import torch.optim as optim

# 生成一些简单的数据
x_train = torch.tensor([[1.0], [2.0], [3.0]], dtype=torch.float32)
y_train = torch.tensor([[2.0], [4.0], [6.0]], dtype=torch.float32)

# 定义线性模型
model = nn.Linear(1, 1)

criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

for epoch in range(100):
    model.train()

    y_pred = model(x_train)
    loss = criterion(y_pred, y_train)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 10 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')

# 测试模型
model.eval()
with torch.no_grad():
    test_input = torch.tensor([[4.0]], dtype=torch.float32)
    prediction = model(test_input)
    print(f"\nInput: 4.0, Predicted Output: {prediction.item():.4f}")
```

Chapter 5

学习心得

5.1 感受

本周的学习内容比较多，调试与性能分析是程序中很重要的部分，学会高效地调试自己地代码要比编写代码更加重要，元编程可以提高代码的灵活性和复用性，但也可能增加代码的复杂性。利用元编程可以实现动态生成代码，分析代码结构和行为等等功能。大杂烩的块中也包含了很多有用的项目，有很多比较实用的技巧。之前学习了LaTex这种比较规范的排版模式，现在又学习了Markdown这种轻量级的文字处理工具，两者结合可以帮助我们处理几乎所有的文字工作，熟练之后效率也会变得很高。同时我们也熟练掌握很多关于GitHub的操作，利用这个强大的工具，能够很好地帮助我们学习。本周的重头戏还是pytorch的学习，PyTorch是一种深度学习框架，以其易用性和灵活性著称，特别适合研究和开发深度学习模型。但是在学习的时候也是很困难的，是一个需要深刻理解的工具。

5.2 遇到的困难

1. 无法准确定位代码出错的位置，这需要我们将问题逐步分解，使用断点调试，逐步检查每个函数的输入输出。通过在关键点打印日志信息，可以快速定位异常发生的位置。但是还是需要不断地练习，来锻炼自己对错误地敏感度。

2. 结合到元编程时还有一个问题，调试复杂的动态生成代码。这个感觉调试起来确实没有好的办法，对于复杂的元编程，需要提前编写测试用例，通过小步迭代开发可以帮助在早期发现和解决问题。

3. pytorch学习中需要深入理解其自动微分机制、复杂的神经网络设计以及优化模型训练的性能。这就需要花费大量地时间来理解学习。

Chapter 6

参考资料

- Missing Semester CN
- Google
- Wikipedia
- B站
- 菜鸟教程
- pytorch
- shellcheck

如果想要查看相关实例练习的源代码可以查询我的GitHub仓库
GitHub Repository