北京理工大學

结课作业报告

《Java 程序设计》结课项目文档

Final Report of Java Programming

学　　院：　计算机学院

专　　业：　数据科学与大数据技术（全英文教学专业）

2025 年 12 月 28 日

目　录

第 1 章 程序的运行环境、安装步骤

本项目使用 JDK 21 进行编译，程序主体为由源码编译、包装得到的 ClinicAssistant.exe，需要配合安装包附带的 ExternalTools、app 和 runtime 文件夹使用。ExternalTools 文件夹包含了程序运行需要的 LibreHardwareMonitorWrapper、Furmark2、cpuburn、CLINIC_OP 工具。

本项目无需安装，在确保上述 ExternalTools、app 和 runtime 文件夹与 ClinicAssistant.exe 在同级目录时，使用管理员权限开启 ClinicAssistant.exe 即可。

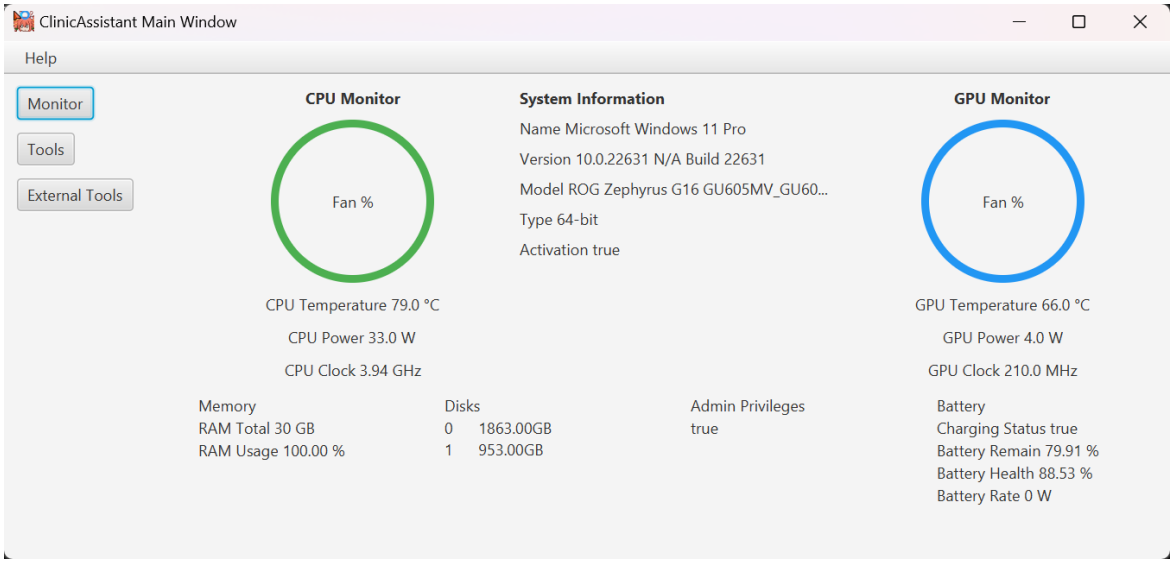

图 1-1 项目预览图

第 2 章 程序开发平台

本项目有 Java 代码 4268 行，Python 代码 84 行，C# 代码 803 行，共计 5156 行代码。程序开发主要使用以下平台和工具：

语言	开发平台	语言版本
Java	IntelliJ IDEA 2025.3	Azul 21.0.9
Python	PyCharm 2025.3	Python 3.13.5
C#	Rider 2025.3	.NET 9.0
LaTeX	VS Code 1.107.1	3.141592653

其中，Java 负责程序主要逻辑和界面的实现，Python 用于生成大量简单重复的样板代码，C# 用于与 LibreHardwareMonitor 通信。

由于使用了持有 GPL v3 的开源项目 fan-control[1]的部分代码，按照规定本项目也采用 GPL v3 开源协议。

[1] https://github.com/wiiznokes/fan-control

第 3 章 程序需求分析

3.1 动因描述

我是我校校级学生组织网络开拓者协会下属电脑诊所的一员。在修电脑的过程中，我们经常需要进行烤机测试、BitLocker 解锁、硬件状态查看等操作。这需要我们在诊所的工具箱 CLINIC_OP 中打开位于不同目录的若干软件、在多个软件之间切换，非常不方便。有时，一些常见的维修需要使用命令行进行，这就要求诊所的同学记住不少 Powershell 命令，无疑增大了维修的精力成本。另外，诊所常做的烤机测试[1]需要人工全程监视硬件状态，以判断电脑的散热能力，这理应被自动化的工具代替，作为一个自动进行的工作。

因此，我开发了 ClinicAssistant 实用功能工具箱，提供硬件信息监视、一键激活 Windows、一键进入 BIOS、解锁 BitLocker、重置代理、修复网卡代码 56 错误、快捷进入 CLINIC_OP 工具等诊所常用的实用功能。另外，借助数值分析的知识，我对诊所判断烤机结果的经验方法进行数学建模，从而实现了自动烤机测试功能。本工具可以降低电脑维修的技术门槛、规范维修的操作流程，从而极大地便利诊所同学维修电脑，使诊所的服务效率更高、更规范。

3.2 竞品分析

目前，最常见的集成了若干维修工具的工具箱程序为“图吧工具箱”。

图吧工具箱类似于一个导航页，收集、罗列出了电脑维修各个方面的若干软件，并为每个软件提供了简单介绍。它允许直接从程序内启动这些软件，而本身并不提供任何维修工具。除此以外，图吧工具箱无其它自动化功能。此外，图吧工具箱缺少常用命令的一键执行脚本。对于电脑维修的新手而言，图吧工具箱并不能起到太大帮助；对于经验丰富的高手来说，又没必要从图吧工具箱去启动工具。

[1]烤机测试：指借助特殊软件使电脑的 CPU、GPU 处于最大负荷运行状态一段时间，通过对电脑温度、功率等指标的观察判断其最大性能和散热能力。

图 3-1 图吧工具箱

诊所最常使用的工具莫过于电脑硬件指标（温度、功率、频率等）的监测和烤机测试工具。指标监测常用软件是 AIDA64 或 HWiNFO；烤机测试常用软件是 cpuburner（集成于 AIDA64）和 Furmark。每次进行烤机操作时，都需要打开 AIDA64 和 Furmark 两个软件并在软件内操作。AIDA64 可以同时进行 CPU 的烤机测试和硬件指标检测，但是其没有指标 – 时间图象的功能，无法直观判断指标变化趋势。因此有时需要额外开启 HWiNFO 以获取图象。这样，为了完成烤机测试就需要开启三个程序，操作上非常不便。

此外，上述的两款检测软件中 CPU 和 GPU 数据间被其它硬件信息分隔开，并且软件给出的信息太多，而大多对于烤机测试无用。因此烤机时必须在二者间上下滑动翻找需要的信息，也带来了不必要的繁琐操作和学习成本。

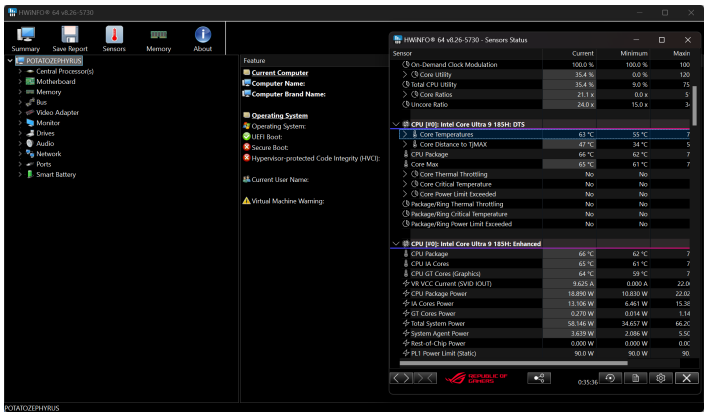

图 3-2 HWiNFO

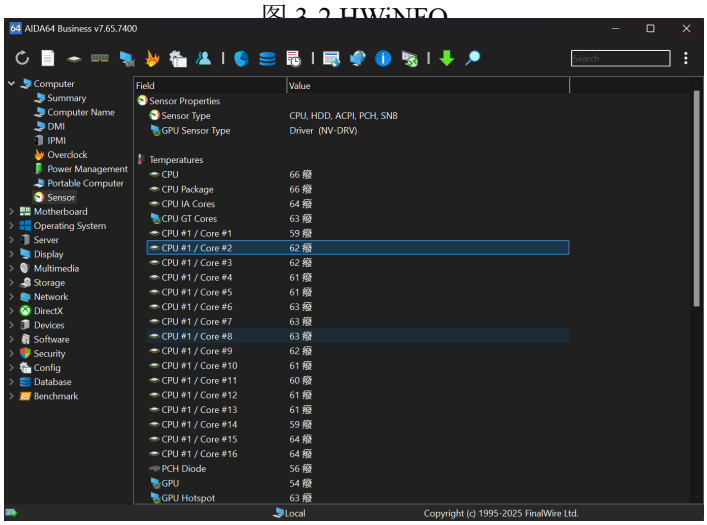

图 3-3 AIDA64

图 3-4 两款常用硬件监测软件

因此，ClinicAssistant 提供的常用指标监视；自动化、一键式操作可以极大地方便电脑维修的各种操作。

3.3 功能描述

本程序作为维修电脑的实用工具箱，主要面向有电脑维护维修需求的同学。通过本程序可以直观地观察到电脑的各项硬件信息和指标、方便地进行各类维护诊断操作。具体功能清单如下：

硬件信息监视：实时监视电脑的 CPU 温度、功率、频率；GPU 温度、功率、频率；内存大小与占用率；物理硬盘数量及大小；电池充电状态、剩余电量、电池健康度和充放电功率。系统信息监视：监视系统的名称、版本号、主板模具、位数和激活

状态。如图3-5所示。

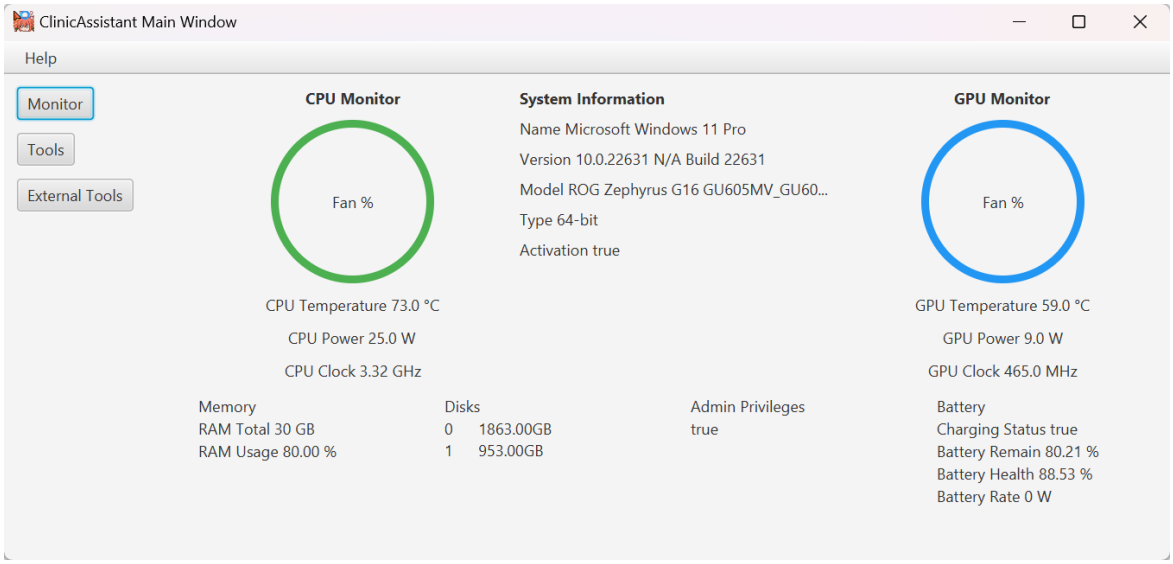

图 3-5 监视器页面

实用功能菜单：提供了一键激活 Windows、一键重启并进入 BIOS、一键解锁 BitLocker、一键代理重置、一键网卡 Code56 修复和自动烤机测试功能。如图3-6所示。

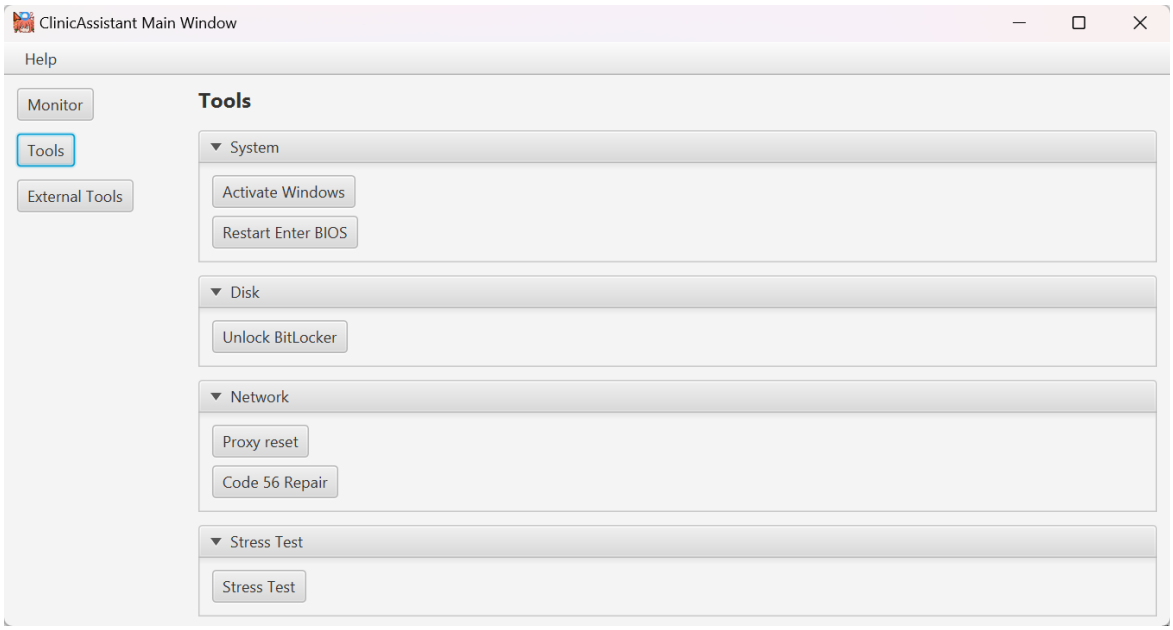

图 3-6 实用工具页面

外部功能导航：提供了快速打开外部工具的导航页，点击按钮即可打开对应工具。如图3-7所示。

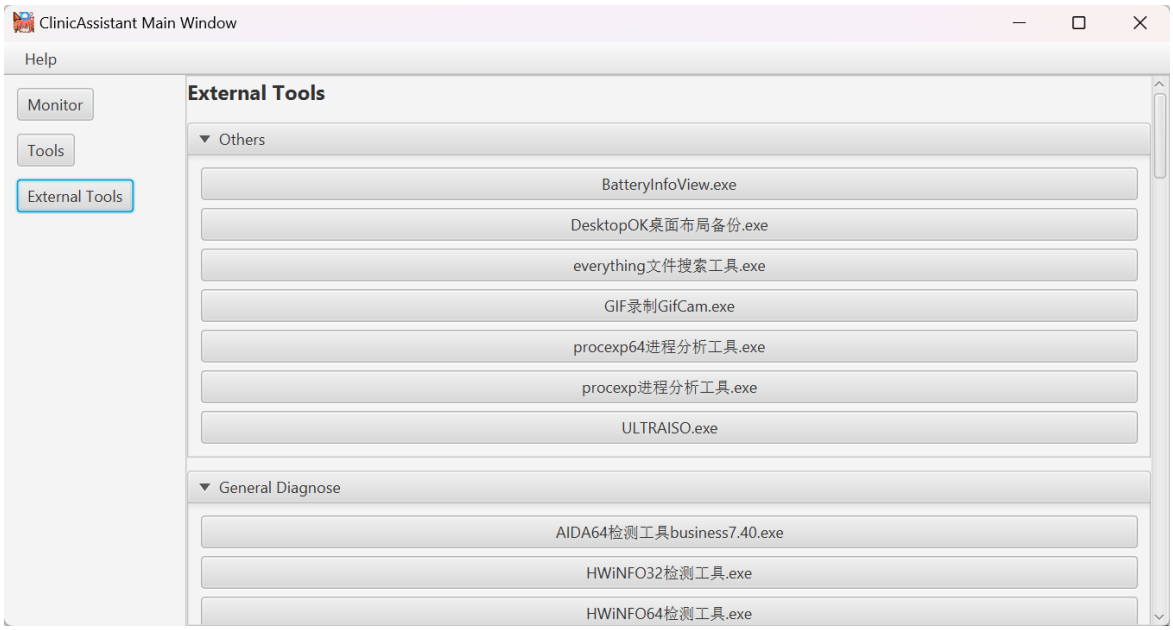

图 3-7 外部工具导航页面

帮助页面：帮助界面附上了网协 wiki 的链接和各咨询群的群号。如图3-8所示。

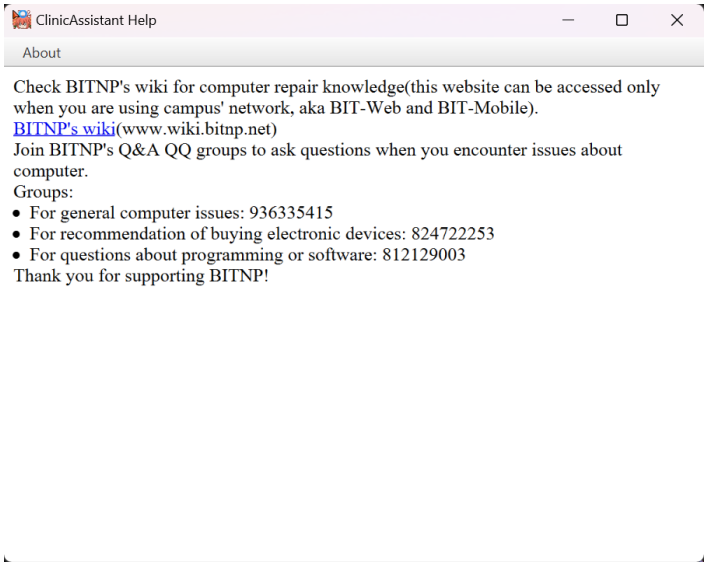

图 3-8 帮助界面

关于页面：关于界面附上了网协首页与本项目 GitHub 仓库的链接。如图3-9所示。

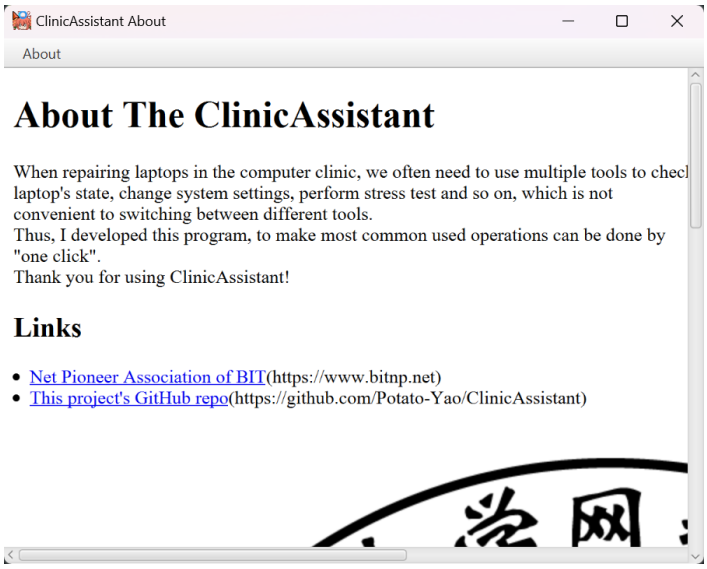

图 3-9 关于界面

版本页面：关于界面附上了当前 ClinicAssistant 的图形化界面和内核版本。如图3-10所示。

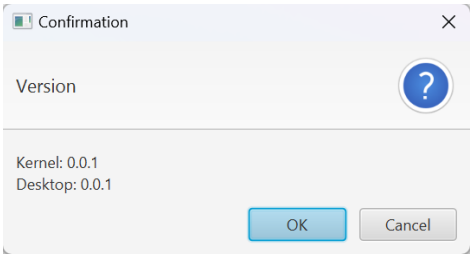

图 3-10 版本界面

一键激活 Windows：使用激活脚本激活 Windows 系统，点击按钮即可激活 Windows 系统，若已激活系统则无法按下激活按钮。如图3-11所示。

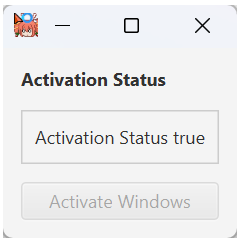

图 3-11 Windows 激活工具

一键进入 BIOS：重启电脑并进入 BIOS 设置界面。如图3-12所示。

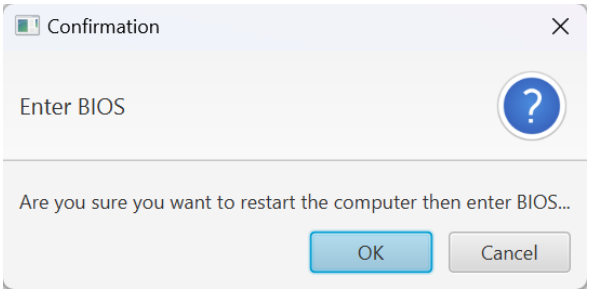

图 3-12 重启并进入 BIOS 工具

一键解锁 BitLocker：实时识别电脑上所有磁盘分区的 BitLocker 加密情况，在没有完全解密的磁盘上按下左键即可招出确认菜单，确认后自动解锁 BitLocker。如图3-13所示。

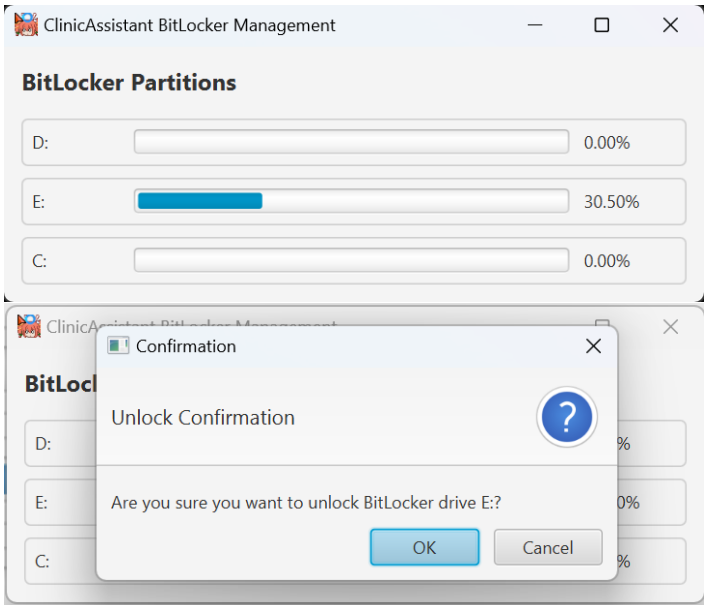

图 3-13 BitLocker 解锁工具

一键重置网络代理：重置网络代理。如图3-14所示。

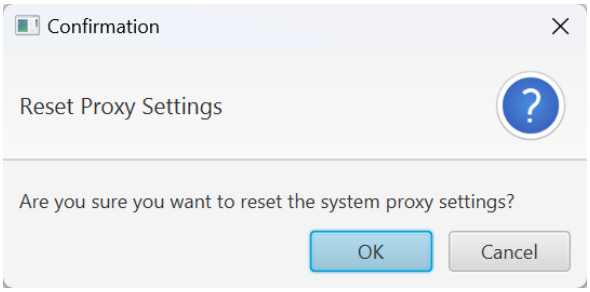

图 3-14 重置网络代理工具

一键修复网卡代码 56：修复网卡驱动报错“代码 56”。如图3-15所示。

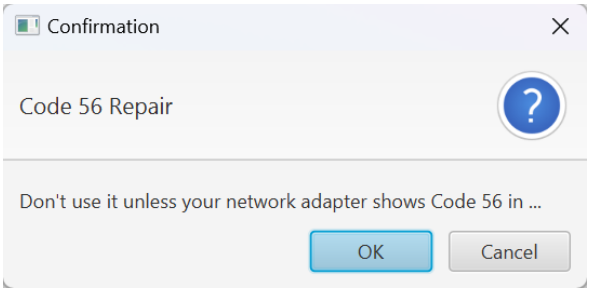

图 3-15 重置网络代理工具

自动烤机：勾选需要烤 CPU 或/和 GPU，输入被烤电脑电源适配器的功率（可选），自动进行烤鸡测试。在测试时可以查看已测试时间、温度 – 时间图象和功率 – 时间图象、算法对指标的实时评估。默认执行自动模式：当算法判定烤机效果已经可以说明散热良好时烤机自动结束。若 CPU 或 GPU 温度超出安全上限则烤机自动结束。如图3-16所示。

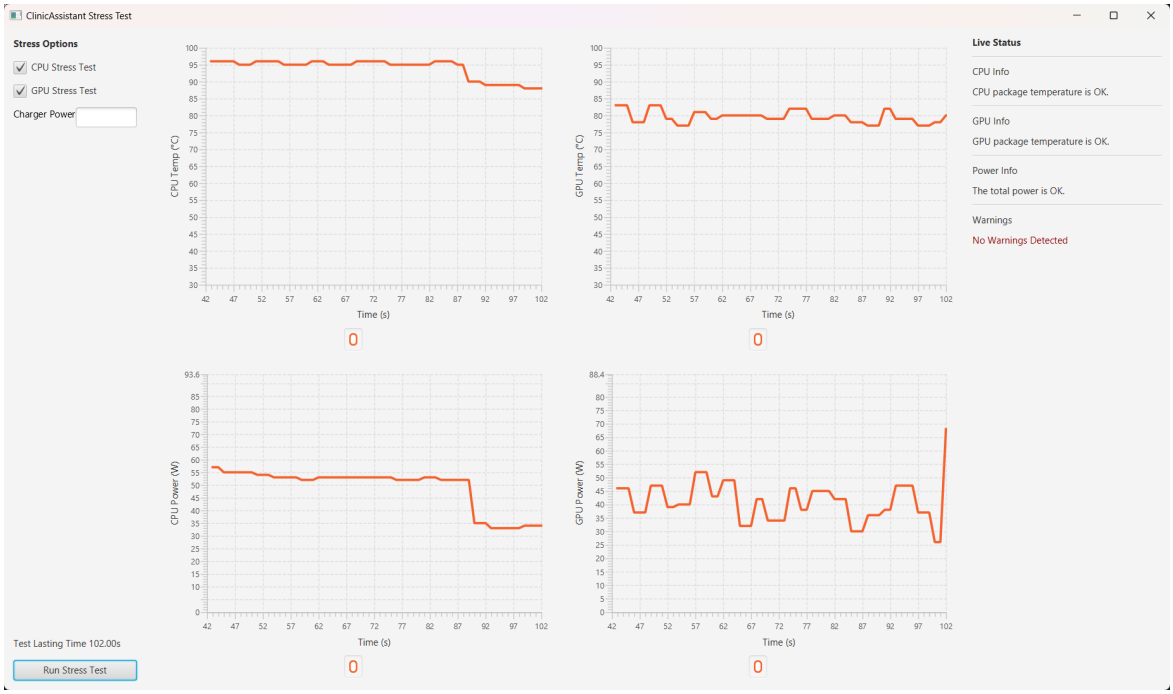

图 3-16 自动烤机测试工具

3.4 UI 界面与交互设计

本项目从诊所工作的实际情况出发进行设计，目标在于使得常用参数指标易于观察、常用操作可被“一键”执行，从而方便诊所的日常工作。因此，我将软件设计成了“监视器”、“工具”和“外部工具”三部分，使用左侧栏进行切换。如图3-5、3-6和3-7所示。

为了这个目标，整个软件都遵循着简单清晰的设计原则：仅使用按钮、文本和图表来设计交互、展示信息，以牺牲部分可定制性和更加详细的信息，换来更快速易于上手的交互逻辑。如 BitLocker 解锁工具（见图3-13）和自动烤机工具（见图3-16）就是这种设计的典型例子：BitLocker 解锁工具只设计了解锁而无加锁、自动烤机工具只设计了自动判断而无定时功能，就是因为这些需求对诊所工作而言相当罕见，因

此无需进行额外设计。

本项目支持国际化，图形化界面的默认语言是英语。这是为了避免系统编码配置有误导致的界面乱码，这常见于使用英文作为显示语言或开启了全局 UTF-8 编码实验功能的系统。

第 4 章 程序架构设计与技术实现方案

4.1 总体技术方案的确定

截至项目立项时，我已经在诊所工作三个学期了。这段期间（特别是第二学年诊所规范要在换硅脂操作后烤机），我发现很多需要使用工具软件的操作都固定且繁琐，所以如果能有一个软件取代人去自动进行这些操作，诊所的工作效率和质量都会得到提升。恰好这学期 Java 课以项目作为作业，Java 也是我熟悉的语言，我有做出项目的信心。

于是，我归纳了诊所常用的各种操作，提取出程序实现的核心需求：硬件指标的读取、外部工具的调用。我必须先明确这两个需求怎么实现，才能继续开发项目。

关于获取硬件指标，我首先去寻找 AIDA64 和 HWiNFO 的文档，发现它们均为闭源软件，没有可供外部调用的 API。因此，我转而寻找开源的替代项目。我在网上搜索得到项目 OpenHardwareMonitor，不过已经停止维护了。取而代之的是 LibreHardwareMonitor，它是使用 C# 语言编写的，我并不熟悉。但是我在仔细搜索、询问 ChatGPT 后都没有得到其它替代选择，便尝试使用 LibreHardwareMonitor。

LibreHardwareMonitor 分为 UI 和 LibreHardwareMonitorLib 两部分，但是没有提供任何文档。我在 GitHub 找到了对其进行封装的项目，但是同样没有提供文档或者代码示例。最终我找到了一个叫做 fan-control 的 Rust 项目，它包含了一个 LibreHardwareMonitorLib 的 wrapper，可以方便地进行通信、获取硬件数据。于是我将 Rust 代码翻译为 Java 代码，并小规模重构了 wrapper，解决了第一个问题。

关于需要调用的外部工具主要分为三类：第一是使用图形化界面的；第二是使用命令行的；第三是文本化界面或者需要在程序的解释器环境执行命令的。第一种很难实现自动操作，我的解决方案是寻找不需要操作图形化界面的替代品。我使用 Furmark2（支持命令行操作）替代常用的 Furmark，使用命令行操作的 cpu burn 替代 cpuburner。第二类只需使用 Process 类进行调用即可。第三类需要在 Process 进行相对复杂的 IO 操作，但仍是可以实现的。

关于总体架构，我选择使用模块化设计，将内核与交互界面分开，从而便于项目的维护和模块间功能的分别、隔离。

4.2 数据存储和处理

本项目处理的数据为各硬件指标（指 CPU、GPU、内存和电池的各项参数）和系统信息（指硬盘分区、BitLocker 状态、系统版本等）。硬件指标是从 LibreHardware-MonitorLib 获取的，系统信息则来自 diskpart、manage-bde 和 systeminfo 等程序。

要想获取硬件指标，首先需要向 LibreHardwareMonitor 发出请求获取 hardware list。hardware list 是一个 JSON 字符串，储存了 LibreHardwareMonitor 能够获取的所有硬件信息以及对应的索引。向 LibreHardwareMonitor 发送索引就能得到对应的值。因此，本程序会遍历 hardware list 并解析并存储需要的项。当需要更新硬件指标时，程序遍历存储的项，依次向 LibreHardwareMonitor 获取对应的值。见章节5.3.3。

关于系统信息的处理则更为复杂，因为 diskpart、manage-bde 和 systeminfo 等程序并不会按照任一数据储存格式返回结果，而是返回便于人类阅读的文本，需要按照具体形式进行特殊处理。大部分程序返回的结果都有固定的形式，因此只需要直接针对某一行的某位置的子字符串进行读取即可。但是如 manage-bde 返回的结果结构上更多样，为了避免信息提取错漏，我为其设计了状态机。见章节5.3.2。

关于信息的存储。除储存烤机测试信息外，本程序并没有其它向本地储存文件的需求。在烤机时，程序会实时将烤机的用时、CPU 温度、CPU 功率、GPU 温度和 GPU 功率储存在程序同级目录的 CAFiles 文件夹下，使用 CSV 格式存储。

使用 CSV 主要出自存储数据的使用场景考虑：导出烤机数据是为了在需要时复盘烤机结果。这需要一种直观简单的格式，以便于人工进行数据阅读和比对，并最好能够导入到如 Excel 等软件进行可视化分析。这就排除了 JSON、XML 和 TOML 等常用格式。因此，紧凑、表格形式组织的 CSV 就成为了合适的选择。另外，需要导出数据的种类是可被提前确定的，因此使用固定列的格式更能够使得数据紧凑直观，这同样是拒绝 TOML、JSON 和 XML 等的因素。

4.3 架构设计、数据结构和算法的面向对象实现技术方案

本项目分为 Kernel 和 Desktop 两个模块。

Kernel 提供了程序需要的一切获取硬件信息、与外部工具交互的方法，分为 External、Hardware、Software 和 Utils 四个包。

包名	功能
External	包含与 LibreHardwareMonitorLib wrapper 等外部工具交互的 helper 类
Hardware	包含用于表示各个硬件的 JavaBean、获取硬件信息的 manager 类
Software	包含用于表示系统信息的 JavaBean、获取系统信息的 manager 类
Utils	包含管理员权限检测等杂项工具

表 4-1 Kernel 模块

Desktop 是程序交互界面的实现，使用 JavaFX 实现，分为 Component、Controller 和 Utils 三个包。

包名	功能
Component	包含对一些 JavaFX 组件的封装
Controller	包含各个 stage 的 controller
Utils	包含 CSV 和对话框等杂项工具

表 4-2 Desktop 模块

本项目在设计上充分采取面向对象设计中的封装思想。为了便于对权限进行控制、避免因修改权限导致下游代码需要修改，本项目中所有成员变量均为 private 标记，使用 public 的 Getter 和 Setter 对外开放。

本项目还使用了若干设计模式，使得代码组织良好、后期改动成本下降。最常用的是 builder 模式和 singleton 模式。builder 模式用于可能加入更多需要赋值、可有默认值的类。如代码4-1所示。

```
/**
 * builder for {@code StressTestUtil}
 */
public class StressTestUtilBuilder {
    private final StressTestUtil util;
    public StressTestUtilBuilder() throws IOException {
        this.util = new StressTestUtil();
    }
    public StressTestUtilBuilder cpuTest(boolean needTest) {
        util.setTestCPU(needTest);
        return this;
    }
    // 省略若干代码
```

```
    public StressTestUtil build() {
        return util;
    }
}
```

代码 4-1 StressTestUtilBuilder

singleton 模式用于因为成员变量一定相同、实例没有功能上区别从而本就不应出现多个实例的类，如各个 manager 类和 helper 类。如代码4-2所示。

```
    private static HardwareInfoManager manager;
    /**
     * the constructor will parse hardware list by LHM, map hardware sensors with index in the list.
     * then it will update sensors value by calling {@code update()} periodically.
     *
     * @throws IOException
     */
    private HardwareInfoManager() throws IOException {
  // 省略若干代码
  }

  public static HardwareInfoManager getHardwareInfoManager() throws IOException {
        if (manager == null) {
            manager = new HardwareInfoManager();
        }
        return manager;
    }
```

代码 4-2 HardwareInfoManager

对于储存硬件信息的各个 Java Bean 和图形化界面的各个 Controller，都定义了父类，以便减少大量重复代码，并便于进行泛型操作。

借助继承关系，在 Desktop 模块中，我使用泛型编写了生成新的窗口（即其对应的 Controller 和 Stage）的方法，使得创建窗口只需要调用函数即可一行解决，极大减少了创建新窗口业务逻辑的重复代码量。如代码4-3所示。

```
    public <Con extends Controller> Pair<Con, Stage> openWindow(String fxml, String title, Class<Con> controllerClass) {
```

```
        try {
            // 国际化相关代码
            // FXML和Stage相关代码
            // Controller相关代码
            // 图标相关代码
            stage.show();
            openedWindows.put(controller, stage);
            return new Pair<>(controller, stage);
        } catch (Exception e) {
            // 省略若干代码
        }
        return null;
    }
```

代码 4-3 openWindow 方法

关于异常。除部分可预料到、可以简单处理的异常外，Kernel 遇到异常不会做任何处理，而是直接向上抛出。这是为了将异常处理置于调用端，增强程序的可定制性，同时降低程序业务逻辑的复杂性。由于时间紧迫，目前版本的 Desktop 遇到异常将会直接退出。

第 5 章 技术亮点、关键点及其解决方案

5.1 技术亮点

本项目提供的各种一键式、自动化操作，极大地方便了业务强度高或技术不熟练的电脑维修者。通过拟合函数的方式判断烤机结果是重要的将经验建模为数学公式的尝试，为更多可能的自动化操作给出了基本思路和方法参考。

5.2 技术关键点

本项目采用多种设计模式，使得代码易于修改，避免了因使用不当造成的错误。见章节4.3。

本项目的图形化界面使用 JavaFX 编写，使用了 MVC 模式组织。

本项目采用了多线程编程。各个软硬件信息获取工具在实例被创建后均会启动自己的 updater 线程从而自动更新数据，避免调用端因忽视数据更新造成错误。

本项目采用 Gradle 作为构建工具，部分核心代码使用了单元测试，代码使用 git 进行版本管理并同步于 GitHub。

本项目使用了若干数值分析算法，使得算法高效、结果准确。

本项目使用了三种编程语言，使用 tcp 协议进行通信，使得开发过程更加高效、与非 Java 程序交流便捷。见章节2。

本项目核心代码均为手写，提升了个人程序开发和资料收集能力，使得代码具有个人风格。见章节6。

5.3 技术难点

本项目开发过程中的难点主要集中在编写 Kernel 时与 Windows“斗智斗勇”上。

5.3.1 操作 diskpart

要想获取硬盘信息，必须要进入 diskpart 软件的解释器环境，才能输入命令、进行操作。因此，我必须编程模拟人工输入命令、等待解释器执行并返回结果的过程。最开始我使用运行 diskpart 的 process 的 outputStream 模拟输入指令，调用 inputStream 的 readLine 方法来读取返回的结果。但是每次返回的都是 diskpart 开启时打印的版本和版权等信息，获得不到命令的执行结果。

经过多次手动模拟，我发现 diskpart 本身执行较慢。因此如果一经启动 diskpart 就输入指令，由于 diskpart 的解释器环境尚未开启，输入的指令就不会执行，自然无法得到正确的结果。只需要在启动 diskpart 和输入指令这两个过程后等待 500 毫秒再去获取输出，就可以得到正常结果了。

另外，在启动 diskpart 后，必须要消耗掉最初的信息。在读取执行结果时也要删去表示可以执行下一次指令的\n DISKPART> 字样。代码如代码5-1和5-2所示。

```
1   /**
2    * connect to the necessary tools for disk management
3    *
4    * @throws IOException
5    */
6   public void connect() throws IOException {
7       // 省略若干行
8       try {
9           Thread.sleep(500);
10      } catch (InterruptedException e) {
11      // 省略若干行
12      // consume initial output
13      while (dpReader.ready()) {
14          dpReader.read();
15      }
16      // wait for diskpart processing
17      try {
18          Thread.sleep(300);
19      } catch (InterruptedException e) {
20      // 省略若干行
21      while (dpReader.ready()) {
22          dpReader.read();
23      }
24  // 省略若干行
25  }
```

代码 5-1 删除 diskpart 启动时的信息

```
1   /**
2    * all diskpart commands can be executed under its interpreter, they can't
    be called externally
3    * <p>
```

```
     * thus, this method is designed for executing command in the interpreter
     *
     * @param command
     * @return
     * @throws IOException the interpreter is not loaded(check if {@code connect
    (()} has been called), or the command just executed failed
     */
    private String executeDPCommand(String command) throws IOException {
        // 省略若干行
        dpWriter.write(command);
        dpWriter.newLine();
        dpWriter.flush();
        // wait for diskpart to process the command
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        // 省略若干行
        StringBuilder output = new StringBuilder();
        while (dpReader.ready()) {
            output.append((char) dpReader.read());
        }
        // remove "\n DISKPART>"
        output.delete(output.length() - 11, output.length());
    // 省略若干行
    }
```

代码 5-2 执行 diskpart 指令

5.3.2 解析 manage-bde 的返回信息

要想获取磁盘分区的信息，就需要使用 manage-bde –status 指令。但是 manage-bde 返回的消息根据硬盘盘符、BitLocker 状态、分区大小、文件系统等会出现不固定的格式，如果使用常规的逐行分析赋值的方法可能会导致错失信息或更严重的信息赋值错位。我设计使用状态机来处理 manage-bde 返回的数据。这样，一旦有任何信息的缺失，在状态机执行完毕时都可以被检测到。状态机实现如代码5-3所示。

```
        // to be aware of missing info, we decide to use state machine
        int state = 0;  // 0: looking for volume, 1: looking for size, 2:
    looking for percentage
```

```
        while ((line = reader.readLine()) != null) {
            String[] lines = line.trim().split("\\s+");
            if (state == 0) {
                if (lines[0].equals("Volume")) {
                    currentPartition = lines[1].substring(0, 1);
                    state = 1;
                }
            } else if (state == 1) {
                if (lines[0].equals("Size:")) {
                    currentSize = Integer.parseInt(lines[1]);
                    state = 2;
                }
            } else if (state == 2) {
                if (lines[0].equals("Percentage")) {
                    currentPercentage = Double.parseDouble(lines[2].substring(0,
     lines[2].length() - 1));
                    state = 0;
                }
            }

            if (currentPartition != null && currentSize != Config.INT_DEFAULT &&
     currentPercentage != Config.INT_DEFAULT) {
                partitionItems.add(new PartitionItem(currentSize,
    currentPartition, currentPercentage));

                currentPartition = null;
                currentSize = Config.INT_DEFAULT;
                currentPercentage = Config.INT_DEFAULT;
            }
        }
        if (currentPartition != null || currentSize != Config.INT_DEFAULT ||
    currentPercentage != Config.INT_DEFAULT) {
        }
```

代码 5-3 处理 manage-bde 返回信息的状态机（省略若干行业务代码）

5.3.3 从 LibreHardwareMonitorLib 获取信息

要想从 LibreHardwareMonitorLib 获取硬件指标，首先需要与其通信获取 hardware list，其具体形式为一个 JSON 字符串，储存 LibreHardwareMonitorLib 能获取到

的各个参数，如代码5-4所示。

```
[
  {
    "Id": "Load0",
    "Name": "CPU Core #1",
    "Index": 0,
    "Type": "Sensor",
    "Info": "CPU Core #1"
  },
  {
    "Id": "Load1",
    "Name": "CPU Core #2",
    "Index": 1,
    "Type": "Sensor",
    "Info": "CPU Core #2"
  },
// 省略若干行
  {
    "Id": "Temperature0",
    "Name": "Core Max",
    "Index": 24,
    "Type": "Sensor",
    "Info": "Core Max"
  },
  {
    "Id": "Temperature1",
    "Name": "Core Average",
    "Index": 25,
    "Type": "Sensor",
    "Info": "Core Average"
  },
  {
    "Id": "Temperature2",
    "Name": "CPU Core #1",
    "Index": 26,
    "Type": "Sensor",
    "Info": "CPU Core #1"
  },
// 省略若干行
```

代码 5-4 LibreHardwareMonitorLib 返回的 hardware list

每一项都有 index，向 LibreHardwareMonitorLib 发送索引即可获取对应项的值。

为了储存本程序需要获取的信息在 hardware list 中对应的索引，我定义了 JavaBean *Sensor* 用来表示 hardware list 中的每一项，如代码5-5所示。

```
package com.potato.kernel.Hardware;

import com.google.gson.annotations.SerializedName;

/**
 * fits the result from lhm, represent a sensor info it gives
 */
public class Sensor {
    @SerializedName("Id")
    private String id;
    @SerializedName("Name")
    private String name;
    @SerializedName("Index")
    private  int index;
    @SerializedName("Info")
    private String info;

    // Getters
}
```

代码 5-5 JavaBean Sensor

随后需要遍历所有项，分析其是否需要被存储、应当归为哪类。我定义了数组 *int[] index*，规定了其中每项用于储存的参数，如代码5-6所示。

```
  /*
    // [0, 32] for cpu
    0 -> cpu load total
    1 -> cpu package temperature
    2 -> cpu core average temperature
    3 -> cpu package power
    4 -> cpu core voltage
    5 -> cpu clock begin
    6 -> cpu clock end
```

```
10
11      // 省略若干行
12       */
13      // stores the sensor index in LHM hardware list
14      private int[] index = new int[INDEX_ARRAY_SIZE];
```

代码 5-6 数组 index

在得到所有需要获取的项后，每次更新数据时需要对其遍历、依次询问 LibreHardwareMonitorLib 获取对应值。

为了使设计简单、运行高效，对每一个 Sensor 的分析和更新数据时获取值的方法是如代码5-7和5-8的若干简单条件判断。

```
1       if (name.equals("CPU Total") && info.equals("Load")) {
2           index[0] = ind;
3       } else if (name.equals("CPU Package") && info.equals("Temperature")) {
4           index[1] = ind;
5       } else if (name.equals("Core Average") && info.equals("Temperature")) {
6           index[2] = ind;
7     // 省略若干行
```

代码 5-7 分析 Sensor 数组

```
1       if (index[0] != -1) {
2           cpu.setLoad(lhmHelper.getValue(index[0]));
3       }
4       if (index[1] != -1) {
5           cpu.setPackageTemperature(lhmHelper.getValue(index[1]));
6       }
7       if (index[2] != -1) {
8           cpu.setAverageTemperature(lhmHelper.getValue(index[2]));
9       }
10    // 省略若干行
```

代码 5-8 获取新值

代码5-7和5-8的代码简单、重复，手动编写效率低下且容易出错。因此我编写了Python 脚本用于生成样板代码，如代码5-9所示。

```
1   import sys
2
3   sensors = [
```

```
    (0, "CPU Total", "equals", "Load", "cpu", "setLoad"),
    (1, "CPU Package", "equals", "Temperature", "cpu", "setPackageTemperature"),
# 省略若干行
]

def warning_message(pos):
    return f"// THE CODE {pos} IS SCRIPT GENERATED, DON'T CHANGE THEM DIRECTLY! CHANGE THE SCRIPT {sys.argv[0]} INSTEAD"

if __name__ == "__main__":
    below_warning = warning_message("BELOW")
    above_warning = warning_message("ABOVE")

    print(below_warning)
    for i in range(0, len(sensors)):
        if i == 0:
            print(
                f'if (name.{sensors[i][2]}("{sensors[i][1]}") && info.equals("{sensors[i][3]}")) {{'
            )
        else:
            print(
                f'}} else if (name.{sensors[i][2]}("{sensors[i][1]}") && info.equals("{sensors[i][3]}")) {{'
            )

        if len(sensors[i]) > 6:
            assert len(sensors[i]) > 6
            print(f"    index[{sensors[i][0]}] = {sensors[i][6]};")
        else:
            print(f"    index[{sensors[i][0]}] = ind;")

    print("}")
    print(above_warning)

    print("\t")

    print(below_warning)
    for i in range(0, len(sensors)):
```

```
        print(f"if (index[{sensors[i][0]}] != -1) {{")
        print(
            f"    {sensors[i][4]}.{sensors[i][5]}(lhmHelper.getValue(index[{sensors[i][0]}]));"
        )
        print("}")

    print(above_warning)
```

代码 5-9 生成样板代码的脚本

5.3.4 子进程管理

在开发中期进行单元测试时，我发现测试结束后程序并没有自行退出。按照设计，程序在开启时会启动 LibreHardwareMonitor，在结束时会向其发送请求要求其关闭。但是借助 Process Explorer，我发现 LibreHardwareMonitor 并没有自行结束。由于子进程没有全部结束，程序本身不会退出。

考虑到强行关闭 LibreHardwareMonitor 并不会造成任何不良影响，我编写了用于强制关闭进程的方法。借助其关闭 LibreHardwareMonitor 即可解决问题。如代码5-10所示。

```
    /**
     * kill a process forcibly
     * @param process
     */
    public static void forceKillProcess(Process process) {
        long pid = process.pid();
        ProcessBuilder killerBuilder = new ProcessBuilder("kill", Long.toString(pid));
        try {
            killerBuilder.start();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        process.destroy();
    }
```

代码 5-10 强制关闭进程

5.3.5 频率显示方法

CPU 频率是判断 CPU 工作状况的重要指标。CPU 有多个核心，每个核心都有其自己的频率，Windows 任务管理器将其通过某种计算合为了一个频率（下面简称为参考频率，记作 r_0），诊所工作经常需要借助任务管理器给出的参考频率进行判断。因此，我需要找出同一将多个核心频率合为一个的办法，得到与任务管理器接近的数值（记作 r）。

首先，简单观察可知参考频率并不是核心频率中的最大值，也不是其简单平均。我取了最大、第二和第三大的三个频率 a、b 和 c，使用加权平均公式5-1得到了近似但偏大的拟合结果。

$$r = 0.5a + 0.25b + 0.25c \tag{5-1}$$

于是，我借助数值分析知识，定义函数 $f(x)$ 使得 $f(r) = r_0$，r 来源于式5-1。对电脑手动限制频率，取得数据点集

$$(0.99, 1), (1.61, 1.24), (2.13, 2.33), (2.55, 2.36), (2.60, 2.44), (2.70, 2.45), (3.10, 2.55)$$

进行牛顿插值，得到多项式5-2。

$$r_0' = 30.79r^6 - 405.53r^5 + 2179.90r^4 - 6102.36r^3 + 9345.26r^2 - 7382.76r + 2334.43 \tag{5-2}$$

但是式子5-2的拟合效果反而更差。推测是因为任务管理器和我的程序获取的数据可能并不是同一时刻，因此数据本身是有误的。最后仍是借助尝试加权平均得到了一个近似的方法：记 a、b、c 和 d 是前四大的频率（单位 MHz），使用加权平均公式5-3。

$$r = \begin{cases} 0.3a + 0.4b + 0.2c + 0.1d, & a - b > 500 \\ 0.35a + 0.35b + 0.2c + 0.1d, & else \end{cases} \tag{5-3}$$

感叹学艺不精，自己的书本知识和现实实验有很大的分隔。

5.4 项目缺陷

由于个人缺乏足够的开发经验、在项目开始前没有进行充分的设计，导致项目实际上有若干设计缺陷，增添项目的维护成本。

第一是 Kernel 中各个用于获取信息的 manager 或 helper 没有统一接口，其 updater 线程没有进行统一管理。前者导致在外部调用时需要查阅方法的具体方法名和参数，增大了使用成本。而后者导致当 manager 或 helper 相互调用时，由于 updater 不同步，会造成获取的信息产生最大 $1.9\times$ 获取信息周期 的延迟。

第二是 Desktop 中所有窗体实际都是声明在 MainApp 中的。因此在添加新的窗体时不仅要编写 controller，还需要在 MainApp 中添加代码。这添加了太大的耦合性，不利于代码的维护和阅读。正确的设计应该是进行一层封装，使用注册的方式添加新的窗体。

第三是开发时单元测试过少。在开发 Kernel 时为了加快进度，我只写了少量的测试，没有覆盖所有代码、所有情况，导致很多 bug 是在编写 Desktop 甚至是开发完成后出现程序行为有误才发现的。

第 6 章 本项目中 AI 技术及工具应用情况的介绍

本项目全部 Java 代码、Python 代码均为我自行设计、编写的，用到了 Github Copilot 的行内补全功能。C# 代码来源于开源项目 fan-control[1]，我进行了少量修改。

[1]https://github.com/wiiznokes/fan-control