Y86-64 Simulator 说明文档

目录

成员分工

- 高庆麾 19307130062 后端stage1 + 前后端整合stage2
- 孙若诗 19307130296 前端stage2
- 共同完成 功能设计 报告撰写 Presentation

提交说明

- `19307130062&19307130296.md`
  - **Stage2 的报告**，markdown 格式的项目说明文档。
- `19307130062&19307130296.pdf`
  - **Stage2 的报告**，pdf 格式的说明文档，由`19307130062&19307130296.md`导出而来
- `README.md`
  - `markdown` 格式的说明文档
- `README.pdf`
  - `pdf` 格式的后端说明文档，由 `README.md` 导出而来
- `back-end/`
  - `my_pipe/`
  - 初始为空，用于存放 `Y86-64 Simulator` 对 `y86-code/` 下 .yo 文件进行模拟的结果 .pipe 文件
  - `my_pipe_honor/`
  - 初始为空，用于存放 `Y86-64 Simulator` 对 `y86-code-honor/` 下 .yo 文件进行模拟的结果 .pipe 文件

    - `psim/`
    - 存放官方给出的标准模拟器，用于生成每个 .yo 文件的标准模拟结果
    - 目录下有四个文件，分别是 `psim`、`psim.backup`、`psim.bad.backup`、`psim.bad2.backup`
    - 其中 `psim` 是当前使用的标准模拟器，实际是 `psim.bad2.backup` 的拷贝。`psim.backup` 是原始的标准模拟器，`psim.bad.backup` 是经过修改后运行出现错误的标准模拟器，`psim.bad2.backup` 是经过修改后正确运行的模拟器，也是当前正在使用的标准模拟器
    - 当前的 `psim` 标准模拟器的修改为：省去了一些事件输出，如 `Execute: ALU: + 0x0 0x0 --> 0x0`、`Wrote 0x13 to address 0x1f8`、`Writeback: Wrote 0x200 to register %rsp` 等，以便比对模拟结果
    - `std_pipe/`
    - 初始为空，用于存放 标准模拟器 对 `y86-code/` 下 .yo 文件进行模拟的结果 .pipe 文件
    - `std_pipe_honor/`
    - 初始为空，用于存放 标准模拟器 对 `y86-code-honor/` 下 .yo 文件进行模拟的结果 .pipe 文件
    - `y86-code/`
    - 存放用于测试的样例程序文件，其后缀为 .yo
    - `y86-code-honor/`
    - 存放用于测试的样例程序文件，其后缀为 .yo
    - `main`
    - 自行编写的模拟器 `Y86-64 Simulator` 的可执行文件
    - `main.cpp`
    - 自行编写的模拟器 `Y86-64 Simulator` 的源代码
    - `Makefile`
    - 自行编写的 Makefile 文件，支持如下命令：
    - make run，用 `Y86-64 Simulator` 测试所有 `y86-code/` 下的样例程序 .yo，并将模拟结果 .pipe 存放在 `my_pipe/` 下
        - make run s=1，用 标准模拟器 测试所有 `y86-code/` 下的样例程序 .yo，并将模拟结果 .pipe 存放在 `std_pipe/` 下
        - make run h=1，用 `Y86-64 Simulator` 测试所有 `y86-code-honor/` 下的样例程序 .yo，并将模拟结果 .pipe 存放在 `my_pipe_honor/` 下
        - make run s=1 h=1，用 标准模拟器 测试所有 `y86-code-honor/` 下的样例程序 .yo，并将模拟结果 .pipe 存放在 `std_pipe_honor/` 下
        - make clean，清除 **Makefile 所在目录下** 的全部 .yo 和 .pipe 文件。这常用于模拟器运行失败时，相关文件会被保留，需要手动进行清理的情况

- `front-end/`
    - `build/`
    - Vue 工程输出文件夹
    - `config/`
    - Vue 工程相关配置文件夹
    - `node_modules/`
    - npm 模块文件夹，项目需要从这个文件夹中调用模块
    - `semantic/`
    - Semantic-UI 文件夹，用于项目图形界面

- `src/`
- Vue 项目相关文件夹
- `testcases/`
- 存放用于测试的 `.yo` 样例
- `demo.webm`
- 项目演示视频
- `index.html`
- 项目主页面 html
- `install demo.html`
- 准备环境安装的终端输出演示
- `jquery.min.js`
- jquery.js 文件
- `main.cpp`
- Stage2 后端代码，与 Stage1 的后端代码有所不同
- `main.js`
- electron 的主 js 文件
- `main.so`
- `main.cpp` 生成的动态链接库，用于提供前端 js 调用后端主函数运行的接口
- `me.css`
- 自定义样式表
- `package.json`
- 存放一些依赖包信息
- `package-lock.json`
- 存放一些依赖包信息
- `README.md`
- 自述文件，markdown 格式
- `README.pdf`
- 自述文件，pdf 格式
- `semantic.json`
- Semantic-UI 的 json 配置文件
- `test.cpp`
- 文件读写接口源码
- `test.so`
- 由 `test.cpp` 生成的动态链接库，用于提供前端 js 进行文件读写操作的接口
- `work.js`
- 前端 js

除上述文件与目录外，提交目录中不含有任何其他文件或目录

模拟器使用方法

单独测试后端

- 使用上文提到的 make 命令进行测试
- 直接 ./main < [input_file] > [output_file]

前端

- 首先 cd 到前端主目录（front-end）下，然后在终端中执行 `electron .` 即可。electron 会自动弹出一个应用窗口，在窗口中即可看到项目，并看到如下组件：
  - 载入文件按钮：点击后，出现文件选择窗口；打开目标文件，可为模拟器指定输入文件，同时输入内容展示在input模块。测试文件结构需与.yo相同。
  - 运行/暂停按钮：点击后，运行和暂停图标相互切换；进度条未达到最大周期时，点击运行变为运行状态；运行状态下，点击暂停变为静止状态。
  - 单步向前按钮：点击后，若处于运行状态则暂停；若当前未达到最大周期，向前推进一周期。
  - 单步向后按钮：点击后，若处于运行状态则暂停；若当前不处于0周期，向后倒退一周期。
  - 频率进度条：点击任意位置或拖动，左侧频率随进度条变化，运行状态下数据更新频率改变。
  - 周期进度条：点击任意位置或拖动，左侧周期随进度条变化，若处于运行状态则暂停，数据直接更新到指定状态。

实现功能

- 后端支持stage1的各种操作，包括荣誉计划部分。
- 前端常态化展示的数据包括更新频率、当前所处时钟周期、此输入数据下最大时钟周期、条件码、部分寄存器数据、部分内存数据、当前时钟周期各阶段执行语句。
- 前端支持加载输入文件、运行/暂停、单步向前、单步向后、调整更新频率、拖动进度条展示任意时钟周期时刻状态。

实现细节

后端

总体框架和设计原则

- **中心原则**：以硬件设计风格为基调，对硬件单元（寄存器、线路）及其相互作用进行模拟
  - 由此，整个模拟器中，有如下自定义类型：
  - `mem_t` 内存单元类型
  - `reg_t` 寄存器类型
  - `cpu_t` CPU控制层类型

    - `ptr_t` 指针类型
    - `wire_t` 线路类型
    - `stat_t` 状态指示器类型
    - `cc_t` 条件码类型
    - `imm_t` 立即数类型
    - ...
    - 后期设计中，为避免严格类型区分造成的编程复杂度开销和风险，主要仅使用两种类型：
    - `reg_t` 寄存器类型
    - `wire_t` 线路类型
    - 硬件风格编程的优点：
    - 本质上是对电路图的模拟，只要画出详细而正确的电路图，很容易得到正确运行的模拟器
    - 由于本质是对硬件的模拟，能相当容易地解决 所有硬件寄存器**同时**在时钟上升沿更新的问题（这是比较麻烦且很容易出错的模拟部分，但是通过对硬件进行模拟可以很容易地进行处理，且几乎没有任何编程错误风险）
    - 很容易处理硬件电路**并行计算**的特点
- 设计框架
    - 基本硬件组件及工作流程
    - 组合逻辑块、系统硬件逻辑块（包含内存、寄存器文件、条件码等的更新）、阶段寄存器等
    - 标准工作流程：初始化、运算、更新（加载）、运算、更新（加载）、...
    - 建议先阅读标准工作流程中各个阶段的介绍，以便更快理解整体设计思想
    - 组合逻辑块
    - 所有的组合逻辑块，在模拟中等价于无返回值函数，其中输入变量采用**值传递**，输出变量采用**引用传递**，输入变量可以是 `reg_t` 或 `wire_t` 类型，输出变量 **一般为** `wire_t` 类型
    - 系统硬件逻辑块
    - 也可看做组合逻辑块，不同点在于需要真正更新系统硬件（系统硬件作为输出变量），如内存、寄存器文件、条件码等
    - 阶段寄存器（阶段划分）
    - 除标准的五大阶段 **F**（取值），**D**（译码），**E**（执行），**M**（访存），**W**（写回）外，额外添加 **L**（逻辑控制）阶段，该阶段控制各个阶段是否暂停或添加气泡
    - 并不实际存在，由很多阶段内硬件寄存器构成，两个阶段之间存在很多线路，这些线路视作属于下层阶段（如 **D** 阶段和 **E** 阶段之间的线路认为属于 **D** 阶段）
    - 控制该阶段的初始化（init）、运行（run）和加载（flush）
    - 初始化阶段
    - 为各个阶段寄存器添加气泡，以便初始化
    - 添加气泡后，需要对每个阶段寄存器进行一次加载，以将所有阶段寄存器包含的硬件寄存器以及相关系统硬件全部初始化。这与标准工作流程中先运算再加载有所不同
    - 此处的加载顺序需要特别注意，要将逻辑控制阶段寄存器（见上文 “阶段寄存器” 部分）放在最后更新，以避免将气泡洗掉
    - 运算阶段
    - 最重要部分（也是最主要部分）是通过当前阶段的所有 `reg_t` 类型变量（代表硬件寄存器），计算出所有 `wire_t` 类型变量（代表线路），这也是对硬件电路进行模拟的核心（硬件寄存器更新后，其输出线路值随之发生改变）
    - 需要严格注意各个组件的依赖关系，如一个组件 ***A*** 依赖组件 ***B***，则 ***A*** 必须在 ***B*** 之后执行，否则 ***A*** 就相当于使用了 ***B*** 错误的**旧**输出值进行了运算

- 需要注意各个阶段寄存器的运算顺序。正确的顺序为按指令执行的**逆序**，也即 **W**, **M**, **E**, **D**, **F** 的顺序进行运算，最后对 **L** 阶段运算。这是由于各阶段之间存在明显的依赖关系
- 更新（加载）阶段
- 也称 flush 阶段，刷新所有的硬件寄存器，也就是将所有新得到的 `wire_t` 写入对应的 `reg_t` 中。这是对时钟上升沿寄存器更新的模拟
- 需要注意各个阶段寄存器的更新顺序，正确的顺序为先加载 **L**，以明确各个其他阶段是否需要暂停或添加气泡并及时进行干扰。其他阶段顺序理论上可以任意，但实际采用指令执行的顺序，也即 **F**, **D**, **E**, **M**, **W** 进行加载
- 加载时更新所有下一阶段寄存器所包含的硬件寄存器，输入变量为这些寄存器对应的线路，输出变量为这些寄存器。对应线路和寄存器名称完全相同，线路所属阶段是寄存器对应阶段的上一阶段，直接将对应线路值传递给寄存器即可
- 每个阶段寄存器首先判断是否暂停，如需要，直接将气泡指示器和暂停指示器清零，然后退出。其次判断是否添加气泡，若需要，则将包含的各个硬件寄存器复位，再将气泡指示器和暂停指示器清零，然后退出。否则，正常将所有 `wire_t` 传递给对应 `reg_t` 即可
- 有些硬件寄存器会直接传给下一个阶段的对应硬件寄存器，此时我们定义一个 Pass 函数，仍然将这些寄存器值先放入线路中，再在加载时通过线路传给下一阶段寄存器，保证处理的同一性，减少特例以降低编程风险
- 命名规范
- 硬件寄存器：以对应阶段的大写字母开头，加下划线，加该寄存器名称
- 线路：以所属阶段的小写字母开头，加下划线，加该线路名称
- 组合逻辑块：一般以该逻辑块名称命名，若各个阶段之间发生重名，则在名称前加对应阶段大写字母加下划线
- 系统硬件逻辑块：以系统硬件逻辑块本身名称命名
- 系统硬件，以系统硬件本身名称命名

编程细节处理

- 编程风格
  - 几乎所有数值均采用十六进制表示，方便调试且更贴近硬件模拟
  - 采用许多名称空间，将所属变量和功能归类，便于测试和查错
  - 所有硬件都定义为全局变量，便于逻辑控制时引用
  - 由于上一条的存在，所有运算组件（组合逻辑块、系统硬件逻辑块）均定义在上述硬件全局变量**之上**，同时在参数表内列出所有组件相关变量，防止编程错误引用到该组件本不该引用的变量上，避免了编程风险
- （宏）定义部分
  - 最大内存范围、最大寄存器数量、PC 起始位
  - 标准程序状态码以及补充的程序状态：气泡对应的标号
  - 所有指令的字节块、代码部分以及功能部分对应的标号
  - 所有寄存器对应的标号
  - 所有硬件模拟所需的类型定义
- 多层次架构
  - 定义层
  - 宏定义、类型定义

- 工具箱
  - 编写了从标号转化到对应指令、寄存器、状态码并能判断是否合法的函数（不合法时输出 “ERROR” 名称）
- 硬件组件层
  - 定义了所有组合逻辑块、系统硬件逻辑块、阶段寄存器的运算及加载函数
- 硬件定义层
  - 定义了所有硬件寄存器、系统硬件以及线路
  - 定义在全局
- 控制层
- 阶段控制层
  - 定义了每个阶段的初始化、运算和更新方法
  - 定义了每个阶段的相关硬件信息输出方法（高度模拟 标准模拟器 的输出格式）
- CPU 整体控制层
  - 定义了 CPU 初始化、运算和更新方法（也就是把所有阶段合在一起，以便调整并确定每个阶段的初始化、运算和更新顺序等）
- 用户层
- 模拟器初始化层
  - 编写了从 .yo 文件中读取内存字节的函数
- 单元测试层
  - 对特定函数功能进行正确性测试
- Y86 用户层
  - 控制整个 Y86 模拟器的初始化、运行和终止行为
- 主函数调用
- 可以选择进行测试还是进行模拟

前端

设计思路

- 不使用输入框，而是采取按钮、拖动条等方式。
  - 增强应用鲁棒性，规避不合法输入对程序的影响。
  - 优化用户体验，用户可以仅使用鼠标简单地操作模拟器，使用各种功能。
  - 将频率、周期数据图形化，更加直观。
- 采取简洁的设计风格，组内讨论后确定设计草图，并根据草图编写html代码。
- 使用网格化设计，并将页面按展示元素分为三个区域、五个模块。
  - 上侧第一区域含一模块，为上侧导航栏，包括网页标题、按钮和进度条等，方便用户操作。
  - 中间为第二区域，含三模块，从左到右分别为输入数据、条件码和寄存器、内存展示，宽度比例为5：6：5。
  - 下方为第三区域，含一模块，为流水线各阶段展示栏。

- 使用semantic-ui框架，主要用于美化基础元素，以及使用框架内的网格结构。
- 交互流程为：加载输入数据 -> 将数据传输给后端可执行文件 -> 运行可执行文件计算出各周期状态 -> 将数据传输回前端并依照用户操作进行展示。

细节处理

- `html`
  - `head` 部分引入semantic-ui框架、jQuery库、自定义css样式和js函数。
  - `top` 编写了导航栏内容，包括用于展示内容的四个文本框、可操作的四个图标按钮和两个进度条。
  - `function` 内嵌JavaScript代码，用于调用一些不由操作触发，而是长期执行的函数，如进度条行为监控等。
  - `code` 编写了输入数据框，从导航栏选择数据导入后展示在此处。
  - `reg` 编写了条件码和寄存器列表，条件码一行三列，寄存器四行两列。具体实现方式为用 `segment` 生成列表，每行内部再用 `grid` 和 `column` 做划分。
  - `mem` 编写了内存列表，大小为六行两列。
  - `cpu` 编写了流水线各阶段展示板块，大小为五行十一列。
- `css`
  - `size` 部分规定了通适于各类元素的不同等级大小、边界距离和基本样式。
  - `text` 部分规定了文字的宽度、深度、间距等样式。
  - `display` 部分规定了元素展示的样式。
  - `slide` 部分规定了进度条所需的 `box`、`bar`、`line`、`dot` 四个元素的样式。
- `js`
  - `update` 用于在数据变化时更新进度条、文本框等元素。
  - `button` 包括各按钮点击时对应的函数。
  - `slide` 包括进度条监控鼠标按下、拖动操作的函数。

前后端综合

早期

采用了 Semantic-UI，希望结合 html, javascript, css 实现前后端的交互，但是经过实际试验后，才发现有很多想法无法实现，其中很大程度上是因为浏览器严格的保护措施造成的（这些问题都是经过了相当充分的资料搜集和研究后发现无法实现的，可能存在非常高妙的技术，但是在工作中没能了解到这些技术）：

- 由于浏览器的保护，浏览器中的 html 内嵌 js 无法直接对本地文件进行修改，如创建删除本地文件。据说早期版本的 IE 浏览器的一个控件 ActiveX 的对象 ActiveXObject 可以支持这些文件交互功能，但是经过实际试验，在 IE11 和其他许多浏览器上都已经不再支持这个操作
- 由于浏览器的保护，js 可以访问本地文件，但是无法获取它的精确路径（浏览器对路径做了处理，导致显示为 fakepath/XXX）
- 由于浏览器的保护，js 也不能执行本地程序

如此种种，简直和我们早先设计好的交互框架背道而驰。我们的交互框架是通过运行，将输入的指令文件转换为运行输出，然后由前端 js 读入运行输出的数据，再将数据存储好并根据需要渲染到页面上。

探索

考虑过几种解决方法，比如：

- 将 C++ 后端用 js 改写，强行嵌入 html 调用的 js 脚本中
  - 但是这种方法显然工作量很大，之所以有这种考虑，是因为 C++ 和 js 相近的语法能为改写和嵌入提供很大的遍历。但仍然很麻烦。
- 弃用 html + js + css 前端框架
  - 这就意味先前的工作都将废弃，在当时时间非常不充裕的情况下，这也是很大的代价。
- 降低 IE 的版本，在支持相关文件交互的浏览器上进行演示
  - 这显然也不太好，毕竟要跟上时代的潮流
- 放弃前后端交互，手动在本地运行后端程序，输出结果，并通过 js 读取该文件数据渲染前端页面
  - 这也是有些麻烦，而且过于笨拙，缺乏自动控制的美感
- …

偶然间，发现一个比较复杂的解决办法，但是看起来似乎比较可行。这就是使用 ffi 调用本地 dll 。对应在 Ubuntu 系统下，就是我们要将后端封装为一个 .so 动态链接库，里面提供了相关的接口，然后用 js 去调用这些接口。

这种方法显然可以解决几乎所有的问题，我们可以在 .so 中带上与本地文件交互的接口，比如创建文件、向文件写入、读取文件，也可以调用后端运行的接口。

首先通过 npm 将 ffi 安装到本地模块中。这样它就存在于 node_module 文件夹中了。

然而经过尝试，浏览器提示无法识别 require('ffi')，这是由于 require 是 nodejs 支持的语法，然而浏览器并不支持 nodejs 。为了解决这个问题，我们又引入了 electron 框架。这种 electron + ffi 的解决方案其实是非常常用的解决类似问题的方法。这对于第一次尝试前端设计的我们也算是一种宝贵的经验。

后期

首先按照 electron 指示进行安装，为了便于管理工程内容，我们又引入了 Vue 框架。我们也按照提示，建立 Vue 工程。

一切准备好以后，如何解决上面提到的问题？发现直接用 electron 运行项目，仍然会出现先前的问题。因为 electron 浏览器视图的内核是 chrome （从外观看上去也非常相似），所以运行起来的效果确实没什么差异。为了解决问题，我们在 electron 的主驱动 js 中添加一项设置，让它的浏览器能够集成 nodejs ，这样我们就可以使用 ffi 了？

然而并不能，又发生了什么问题？浏览器的控制调试台中显示 bindings 和 ffi 的 node 可执行文件提示版本与当前 nodejs 不匹配，但实际上不匹配的是 electron 的版本，因为 electron 有要求的对应 nodejs

版本，于是我们还需要调整 electron 的版本到 4.2.0 ，再进入 bindings 和 ffi 的模块目录下对他们的可执行文件进行重编译，这样才可以开始执行。

最后就是调整一些数据读取和渲染的方法，以及发现调用接口与文件交互时，必须使用文件指针而非重定向，这可能和文件写入的 buffer 有关。

已修复 bug 清单

后端

- 工具箱未考虑不合法情况，修复后在不合法情况下输出 “ERROR”
- 指令不合法的判断逻辑有误
- 使用 `reg_t` 定义指针并在无符号整数环境下进行判断会出现问题，改为特别定义的有符号整数 `ptr_t` 指针类型
- PC 增加器中至少需增加指令字节对应的一个字节
- 对阶段寄存器中暂停和气泡实现机制的理解有误，正确的做法是在正常加载前进行判断
- **F** 阶段寄存器比较特殊，需要加载下一阶段的硬件寄存器 **以及** 自身包含的 predPC
- 逻辑块中某些情况下未涉及的输出变量必须复位
- 条件码更新逻辑中注意每个条件码仅有一位大小，取与如果大于 **1** 必须移到数值 **1**
- 由于在无符号数环境中操作，需要手动用符号位判断正负而非直接判断正负（因为数值上永远非负）
- ALU 中运算是 ***B*** 操作数在前，***A*** 操作数在后
- 注意数值溢出的判断方法
- 跳转条件运算中，注意不同条件对应的条件码运算式
- 每个逻辑块（函数）中要注意是否有变量没有用到，避免遗漏
- 注意不要将读写内存位搞反
- 注意每个字节长度为 **8** 位
- 注意在读写内存时要判断内存位置是否合法
- 注意每个逻辑块的依赖关系以及相应的执行顺序
- 注意初始化时的特殊加载逻辑（**L** 最后执行）
- 注意从 .yo 文件中 fetch 内存内容时，不要完全以 ‘x’ 作为标志（因为右侧说明部分中也可能含有）

前端

- 自定义css样式不起作用，是由于semantic-ui本身样式的优先级较高，需要在自定义css中增加!important
- 注意点击单步按钮或操作周期进度条时，若当前在运行状态，应变为暂停状态
- 注意导航栏为反色状态时，内部元素也需反色才能正常显示
- 修正了周期进度条跑满后仍保持在运行状态的bug

- 修正了未拖动时初始状态进度条和频率不一致的bug
- 注意周期和频率不能为小数、负数或超过最大值
- 注意进度条数据和文本框数据相互影响，应合理安排计算顺序，避免精度误差导致周期数出错
- 注意`css`的`padding`参数为边距设置，要达到对齐目的，需要内部填充元素也保持一致
- 导航栏文本框的宽度会随数据变化，导致右侧内容反复移动，影响用户观感，需要用css样式规定元素宽度
- 注意`segment`列表需要用`attached`参数保持紧密，而`top`和`bottom`参数可使首尾元素有圆角效果
- 修复了周期数过大时周期文本框和周期进度条重叠的问题
- 在css样式中将周期进度条和频率进度条分开处理，以便独立调整长度、颜色等
- 注意周期进度条不能超过最大长度，否则会影响频率进度条
- 注意频率不能为0，将最小值设为1
- 注意频率与时钟周期的关系并不是线性的，而是反比例关系，因此要设置合适的频率范围和最大时钟周期，以保证数据更新速度在合理范围内
- 优化了初始频率设定，频率为1则现象不明显，且进度条颜色未填充，不利于测试和展示；将频率初始值设为10，较容易观察网页运行状况

Reference

- 使用了前端框架semantic-ui，参考了semantic-ui官方文档
- 使用了JavaScript库jQuery
- 使用了electron框架

总结反思

- html页面基本为手工打造，虽然对学习网页结构和基本元素大有帮助，但编程上有更简单快速的开发方式，使用更多框架性结构也更有利于网页迁移和后续维护。
- 进行全新的开发工作时应尽早开始，因为需要的时间难以估计。不要因为之前有比赛、其他项目等原因完全停滞进度，要学习同时推进多个项目进程，保持学习状态。
- 应当尽早确定设计思路，并在所使用框架基础上对前后端基础交互功能进行测试，再展开前后端分离的开发工作。本次项目先行完成了后端，已经未将前端纳入考量；开发前端时再次忽略了与后端的交互实验，只是从理论上设计了交互流程，随后在此基础上完成了绝大多数工作。最后忽然发现js在浏览器上无法直接与C++交互，直接影响到模拟器的基础功能，导致截止日期前工作量陡增。
- 尽管C/C++对其他高级语言的学习有诸多启发，仍不应想当然地推测其他语言支持的操作，要参考所用语言本身的官方文档，结合语言的使用环境来理解。
- 前端设计在用户行为、按钮操作等方面较为注重鲁棒性，缩放也不会影响页面功能。但对不同浏览器的区别考虑不足，前端开发相关经验较为缺乏。虽然模拟器的基础功能不会受损，但是美观度可能有所下降。
- 本次项目设计需要从前端获取数据，调取后端可执行文件处理，但是未对输入数据做安全性检查，存在一定的安全风险，有较大的优化空间。实际上，我们在js和C++交互上遇到的问题也主要是出

于浏览器的风险控制。尽管由于时间和技术上的不足，我们的课程项目设计目前不得不使用这种方法，但是浏览器的限制确实是合理且必要的。

这就是说明文档的全部内容，感谢观赏！

2020.12.22