前言

1. 说明本项目需要用到x86elf 编译工具链，qemu (其他虚拟机没有添加支持，可以自行完善）目前仅支持在 Macos 上运行（下一轮迭代计划支持 Linux、Windows ）
2. 本项目使用了 git 进行版本管理，每轮迭代都有清楚标注
3. 工具链安装教程

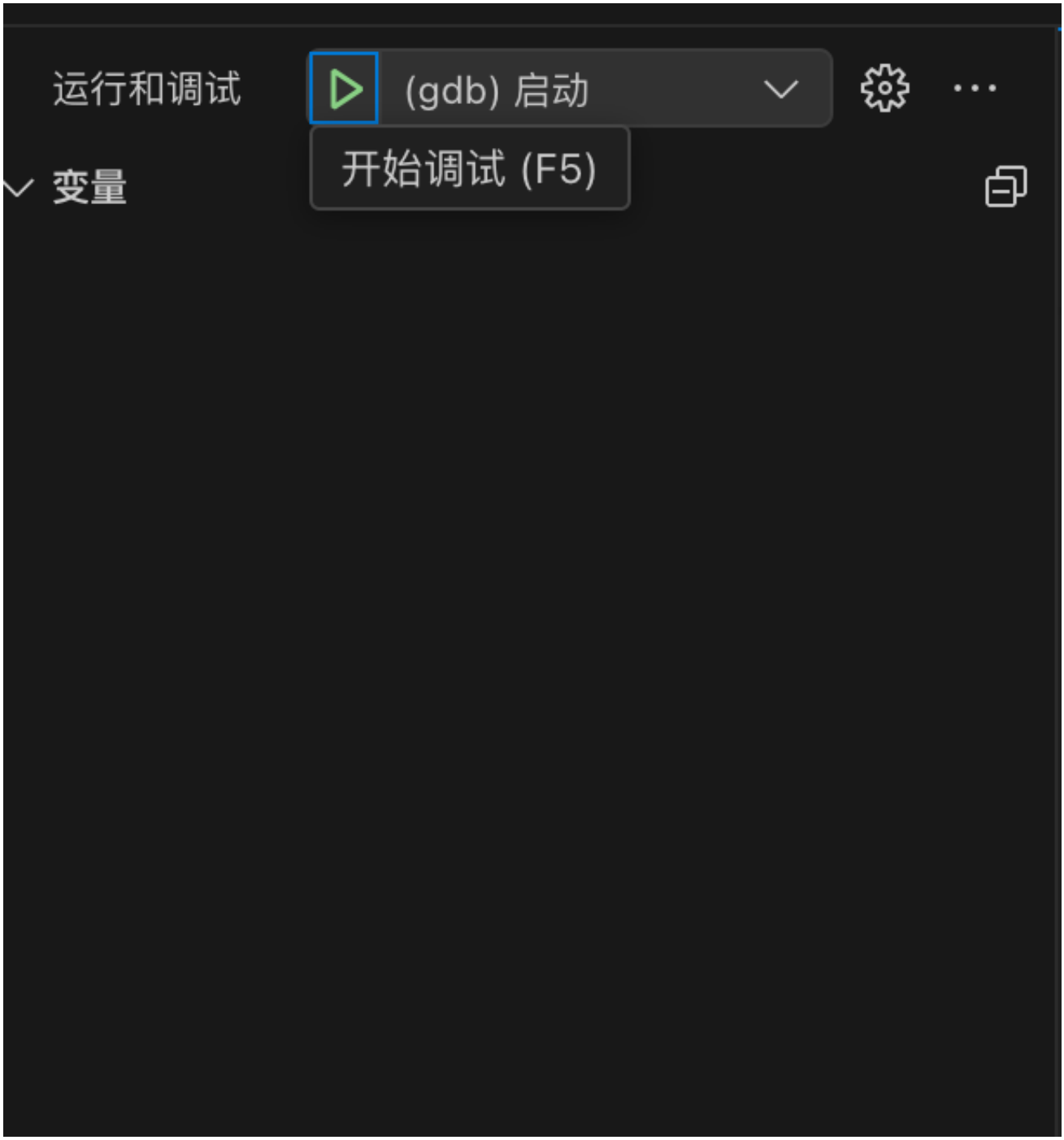
```
brew install x86_64-elf-gcc
brew install x86_64-elf-gdb
brew install qemu-system-i386
```

3. 关于调试：调试建议使用 vscode，相应的配置文件已经写好（下一轮迭代预计支持gdb 一键调试）
4. 关于真机运行：由于目前的较新的电脑都采用了 UEFI 而非 BIOS，故只有能使用 bios 的电脑可以直接裸机运行
5. 关于使用方法具体步骤：

```
make
make start
```

然后在 vscode 里面按 F5

或者点击 gdb 启动即可



更新内容

1. 更加优化的 makefile 文件

   现在的主Makefile：

```
all : build       You, 6分钟前 • Uncommitted changes
    dd if=/dev/zero of=./build/disk.img bs=1024 count=1000
    dd if=./build/boot_all.bin of=./build/disk.img conv=notrunc
    dd if=./build/load_all.bin of=./build/disk.img bs=512 seek=2 conv=notrunc
    dd if=./build/kernel_init.bin of=./build/disk.img bs=512 seek=100 conv=notrunc

build :
    mkdir -p build
    mkdir -p asm

    make -f ./src/boot/makefile Makefile boot
    make -f ./src/loader/makefile Makefile loader
    make -f ./src/kernel/makefile Makefile kernel

start :
    qemu-system-i386  -S -gdb tcp::1234 -m 128M  -drive file=./build/disk.img,index=0,media=

clear :
    rm -rf build
```

现在的 makefile 把编译组件分下去了，而不是写成一坨，阅读起来更加清晰

2. kernel 的框架，完善 loader

   已经完成 bootinfo 的传参：

```
#include "../../../comm/bootinfo.h"
void kernel_init(boot_info_t *boot_info) {
    boot_info->ram_region_count = 100;
    int a = 1;
    a++;
    return;
}
```

在 loader32 里面添加了使用 LBA 方式读取硬盘：

```
static void read_disk(int sector, int sector_count, uint8_t *buf) {
    outb(port: 0x1F6, data: (uint8_t)(0xE0));

    outb(port: 0x1F2, data: (uint8_t)(sector_count >> 8));
    outb(port: 0x1F3, data: (uint8_t)(sector >> 24)); // LBA参数的24~31位
    outb(port: 0x1F4, data: (uint8_t)(0));            // LBA参数的32~39位
    outb(port: 0x1F5, data: (uint8_t)(0));            // LBA参数的40~47位

    outb(port: 0x1F2, data: (uint8_t)(sector_count));
    outb(port: 0x1F3, data: (uint8_t)(sector));       // LBA参数的0~7位
    outb(port: 0x1F4, data: (uint8_t)(sector >> 8));  // LBA参数的8~15位
    outb(port: 0x1F5, data: (uint8_t)(sector >> 16)); // LBA参数的16~23位

    outb(port: 0x1F7, data: (uint8_t)0x24);

    // 读取数据
    uint16_t *data_buf = (uint16_t *)buf;
    while (sector_count-- > 0) {
        // 每次扇区读之前都要检查，等待数据就绪
        while ((inb(port: 0x1F7) & 0x88) != 0x8) {
        }
        int sector_size = 512;
        // 读取并将数据写入到缓存中
        for (int i = 0; i < sector_size / 2; i++) {
            *data_buf++ = inw(port: 0x1F0);
        }
    }
}
```

3. 常见汇编指令写成内联汇编便于 c 语言调用

```
#ifndef INSTRCUT
#define INSTRCUT
#include "type.h"

static inline void outb(uint16_t port, uint8_t data) {
  __asm__ __volatile__("outb %[v], %[p]" : : [p] "d"(port), [v] "a"(data));
}

static inline uint16_t inw(uint16_t port) {
  uint16_t rv;
  __asm__ __volatile__("in %1, %0" : "=a"(rv) : "dN"(port));
  return rv;
}

static inline uint8_t inb(uint16_t port) {
  uint8_t rv;
  __asm__ __volatile__("inb %[p], %[v]" : [v] "=a"(rv) : [p] "d"(port));
  return rv;
}

#endif
```

4. clangd-format 文件便于统一整个项目的代码风格

.clang-format

```
# 使用 LLVM 风格作为基准（你也可以选择其他风格）
BasedOnStyle: LLVM

# 强制使用 Tab 字符进行缩进
UseTab: Always

# 设置 Tab 的宽度为 4 个空格
IndentWidth: 4
TabWidth: 4
AllowShortBlocksOnASingleLine: false
BraceWrapping:
  AfterFunction: true
  AfterStruct: true
  AfterClass: true
  AfterEnum: true
  BeforeElse: true
  AfterControlStatement: true


# 控制函数参数和返回值类型的空格风格
SpacesInParentheses: false
```