背景

Git 是一个分布式版本控制系统，用于跟踪计算机文件的变化，并协调多个用户之间的协作。它最初由 Linus Torvalds 在 2005 年为 Linux 内核的开发所创建，现在已被广泛应用于各种软件项目的版本控制中。Git 允许开发者记录每次变更的历史，这样可以方便地回溯到早期版本，或者在出现问题时撤销更改。此外，Git 还支持分支(branching)和合并(merging)，使得代码的实验性改动与稳定代码库之间能够有效隔离。

Git 的 staging area（暂存区）是 Git 中的一个重要概念，它是一个临时区域，在提交(commit)之前用于收集改动。当你修改了一个文件后，这些改动最初是存储在工作目录（working directory）中的。为了将这些改动纳入下一次提交中，你需要先将它们添加到暂存区（也称为索引index）。这一步通常通过使用 `git add` 命令来完成。将文件改动添加到暂存区之后，你可以使用 `git commit` 命令来创建一个提交记录，这个提交会包含所有已经暂存的改动。

Staging area 的作用在于它允许开发者更精细地控制哪些改动会被包含在最终的提交中。你可以一次性添加整个文件的所有改动，也可以选择只添加部分改动。这种方式非常适合处理大型项目或复杂功能的开发，因为它能帮助保持提交历史的整洁与有序。

引言

在第一个项目，你将实现Nit，即NJU git。亲自编写代码，实现一个自制git。

虽然实现Nit必不可少地需要操作文件，但是鉴于授课进度，我们**并不**要求掌握C++中的文件操作。相应地，善良（？）的助教将提供一套API，只需要理解这些API的用法。就可以愉快地开始编程。当然，可以选择不使用助教提供的API，这是完全OK的。

Nit参考了UCB CS61b的其中一个project：gitlet。可以自行搜索阅读gitlet的实验文档，也许会有所启发。

实验要求

使用C++，实现Nit的功能。不限制STL的使用。项目的实现代码必须是自己完成的。不要修改助教提供的代码。

Nit没有branch功能。因此只需要实现单分支的维护。简单起见，不需要处理子文件夹；换句话说，总是可以认为Nit的工作目录是一个“平坦的”，没有子目录的文件夹。

属于Nit自身的数据都应当被保存在 `.nit` 文件夹中；如果一个目录中存在 `.nit` 文件夹，那么就可以认为这个目录已经被初始化。除了 `init` 外的Nit命令**只能**在已初始化的目录中执行。如果在未初始化的目录下执行 `init` 之外的命令，程序应该直接退出并打印错误信息。在 `.nit` 文件夹之外的文件是需要处理的文件。

另外，请总是检查用户的命令行输入。如果用户输入的指令不存在/输入的参数个数不正确，你的程序应打印错误信息后退出。

需要实现的命令

init

- 用法：`nit init`
- 描述：在当前目录下创建一个新的Nit版本控制系统。这个系统在创建之初就有一个commit。这个commit不包含任何文件，并且其commit信息为: `initial commit`。

- 错误处理：如果当前目录已经有一个Nit版本控制系统，程序应该直接退出并打印错误信息`A Nit version-control system already exists in the current directory.`，并且**不应该**修改任何文件。

status

- 用法：`nit status`
- 描述：展示当前staging area中的文件，输出文件名。同时输出将从下个commit中移除的文件。
- 错误处理：无
- 示例：

```
=== Staged Files ===
waagh.txt
waaaagh.txt

=== Removed Files ===
goodbye.txt
```

add

- 用法：`nit add [filename]`
- 描述：将工作目录下指定的文件加入到staging area。这个操作也叫做stage一个文件。如果这个文件已经被stage，新的拷贝将会覆盖旧的拷贝。如果文件内容和当前commit的内容一样，不需要将其加入staging area，并且如果其已经存在于staging area，将其移除。staging area的信息应该被存放于`.nit`文件夹里的某个位置。特别的，如果文件在工作目录中不存在但是current commit追踪了对应的文件名，将此文件标记为已删除，并且在下一次commit中将不会包含此文件。
- 错误处理：如果文件不存在且没有被current commit追踪，提示用户`File does not exist.`后退出。

checkout

- 用法：`nit checkout [commit id]`
- 描述：将commit id对应的commit追踪的文件全部复制到工作目录，覆盖同名的文件。删除所有current commit追踪但checkout的目标commit没有追踪的文件。直接清除staging area。命令完成后current commit应该被修改为checkout的目标commit。
- 错误处理：如果找不到commit id对应的commit，提示用户`No commit with that id exists.`后退出；如果一个将要被checkout命令覆盖的文件存在于工作目录，且current commit没有追踪它，打印错误信息`There is an untracked file in the way; delete it, or add and commit it first.`，之后退出程序。对于错误的检查应该在你修改任何文件之前进行。

log

- 用法：`nit log`
- 描述：从current commit开始向最初的commit回溯，并打印每个commit的hash值和msg信息。
- 错误处理：无
- 示例：

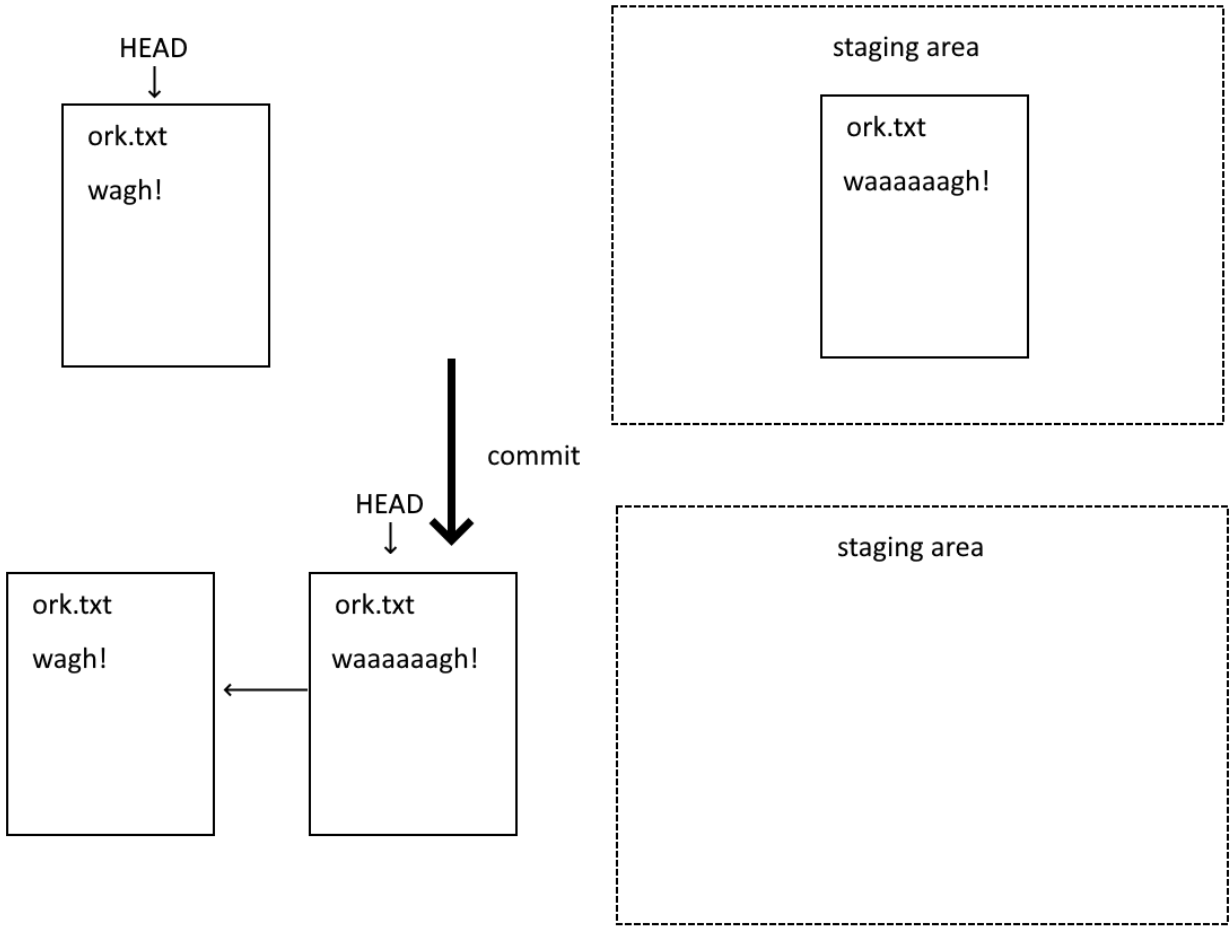
```
===
commit a0da1ea5a15ab613bf9961fd86f010cf74c7ee48
A commit message.
```

```
===
commit 3e8bf1d794ca2e9ef8a4007275acf3751c7170ff
Another commit message.

===
commit e881c9575d180a215d1a636545b8fd9abfb1d2bb
initial commit
```

commit

- 用法：`nit commit [msg]`
- 描述：保存当前commit和staging area中跟踪了的文件的快照，以便以后可以恢复它们，并创建一个新的commit。这个commit被认为是在跟踪这些保存的文件。默认情况下，每个commit所保存的文件快照会与其父commit的文件快照完全相同；它会保持文件的版本不变，而不会更新它们。只有那些在commit时被暂存在staging area的被跟踪文件，commit才会更新其内容，在这种情况下，commit将包含在staging area的文件版本，而不是从其父commit继承的版本。对于那些在staging area但没有被其父commit跟踪的文件，新的commit将会保存并开始跟踪这些文件。
- 提示：
  - staging area在commit后会被清空
  - commit命令只与其父commit和staging area有关，与工作目录状态无关。比如，如果`a.txt`已经位于staging area，但随后在工作目录中被删除，commit仍然会提交`a.txt`位于staging area中的版本。
  - 新提交的commit会作为一个新节点添加到commit链表中，且成为current commit，HEAD指针应当指向这个新commit。新commit的父commit为HEAD指针之前指向的commit。
- 错误处理：如果staging area没有任何文件，程序应该直接退出并打印错误信息`No changes added to the commit.`。如果用户在调用commit命令时没有提供`msg`参数，直接退出并打印错误信息`Please enter a commit message.`。



扩展功能

可以实现更多的Git指令，如：`rm`；也可以增强Nit的指令，如实现：`nit add *`或在`nit status`中输出更多信息；也可以拓展Nit本身的适用范围，如支持Nit处理子文件夹等等。只要合理即可。

实验指南

读取命令行参数

Nit从命令行中获取输入。可以通过`argc`获取用户输入的参数个数，并通过`argv`获取用户实际输入的参数字符串。

检查用户输入的命令是否正确，参数个数是否匹配十分重要。比如，如果用户执行`nit waaaaagh`。你应该提醒用户出了错。

保存你的程序状态

Nit在执行完每条命令后会退出程序（就像Git一样）。也就是说程序每次运行只会执行**一条**命令。需要保证Nit下一次运行时必要的信息（如staging area的文件内容）依然存在。

类设计

这个实验的目标是运用面向对象知识编写cli程序，因此，我们建议至少设计以下三个类：

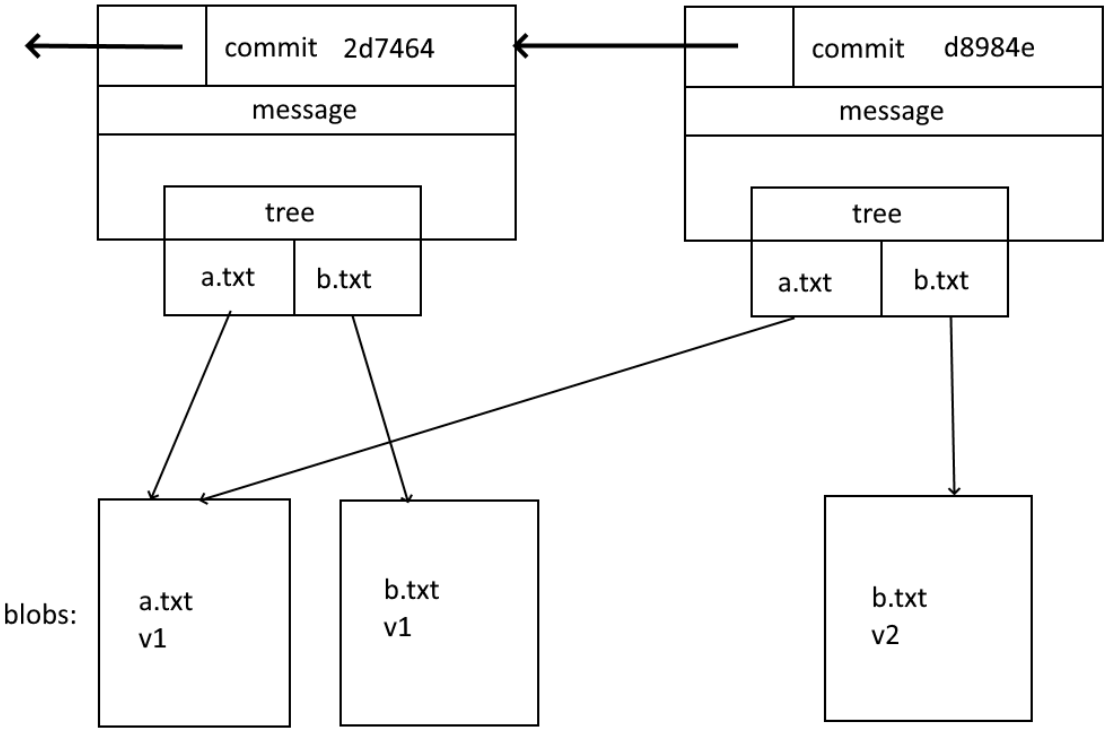
- blobs：blob表示一个文件对象，存储文件的内容。由于文件可能有多个版本，一个文件可能对应多个blob对象，由不同的commit追踪。
- trees：tree是一个将文件名映射到blobs的映射(mapping)。
- commits：commit应当包括一条log信息，一个tree对象，一个指向父commit的引用。

可以自行设计其他类，为类添加其他你觉得有必要的成员。

推荐学习c++ STL中`std::vector`和`std::map`的使用。

Nit数据结构模型

Nit的数据结构模型相比Git而言极其简单，基本上，Nit数据的主体部分就是一个链表。



commits组成一个单链表（在真实的git中，由于可以merge两个分支，commits会组成更复杂的图结构）。commit中的tree对象表示这个commit对应的目录状态，并将文件名映射到blob对象。

保存对象到磁盘

C++本身并没有提供*序列化*机制，因此你可能需要自行考虑如何将一个对象保存到磁盘/从磁盘还原。我们提供了将一个字符串写入文件的API。一种可行的方法是用字符串来保存重建对象需要的信息，并将这个字符串写入文件保存。

我们建议将不同类的对象保存在不同的文件夹中。

引用磁盘里的对象

我们都知道如何引用一个内存里的对象：只需要保存它的指针就可以了。但是如何引用一个保存在磁盘里的对象呢？哈希值可以作为指向对象的"指针"。真实的Git使用sha1作为哈希函数。无论输入数据的大小如何，sha1总是产生一个固定长度的输出，且即使输入数据只产生很小的变化，也会导致完全不同的散列值。这种性质使得你可以方便地将sha1哈希值作为保存对象文件的文件名，并能通过哈希值找到存储对象的文件。另外，你还可以通过比较两个文件的哈希值来判断两个文件内容是否相同，而无需实际比较两个文件。

我们为你提供了`hash()`函数，实现了sha1功能。`hash()`接受一个字符串并返回这个字符串的哈希值，哈希值以字符串表示。你不需要关心sha1的细节。

staging area和HEAD指针

你需要在磁盘中存储staging area和current commit的信息。可以参考Git的实现，尤其是Git的`index`文件和`HEAD`文件。

检查你的实现

你总是可以方便地检查你的实现是否正确。基本上，Nit的功能与Git只有细微差别。你可以使用Git进行和Nit一模一样的操作。除了实验手册规定的部分（如init的行为），在大部分情况下它们的行为都应当一致。

配置框架代码

我们提供了一个可选的框架代码，内含一些文件操作相关的api以及计算sha1的函数。框架代码使用cmake进行管理。在windows和linux下都可以使用。如果希望从零开始，也可以不使用本框架代码。

在linux下直接使用cmake（推荐）

在linux中进行本项目的编程可以获得相当好的体验。如果没有安装了linux系统的电脑。WSL(Windows Subsystem for Linux)可以直接在 windows 上使用 linux 应用程序。

在linux系统中，进入框架代码的文件夹，然后：

```
mkdir build
cd build
cmake ..
```

就能完成项目的配置。

随后，在build文件夹中，执行

```
make
```

进行编译。编译完成后你应该能看见编译好的可执行文件nit。

通过

```
./nit
```

就可以运行程序。

如果不知道如何增加源代码文件，请打开CMakeLists.txt，并在`set(SOURCES ...)`这一条语句中直接加入你新增的源代码名。

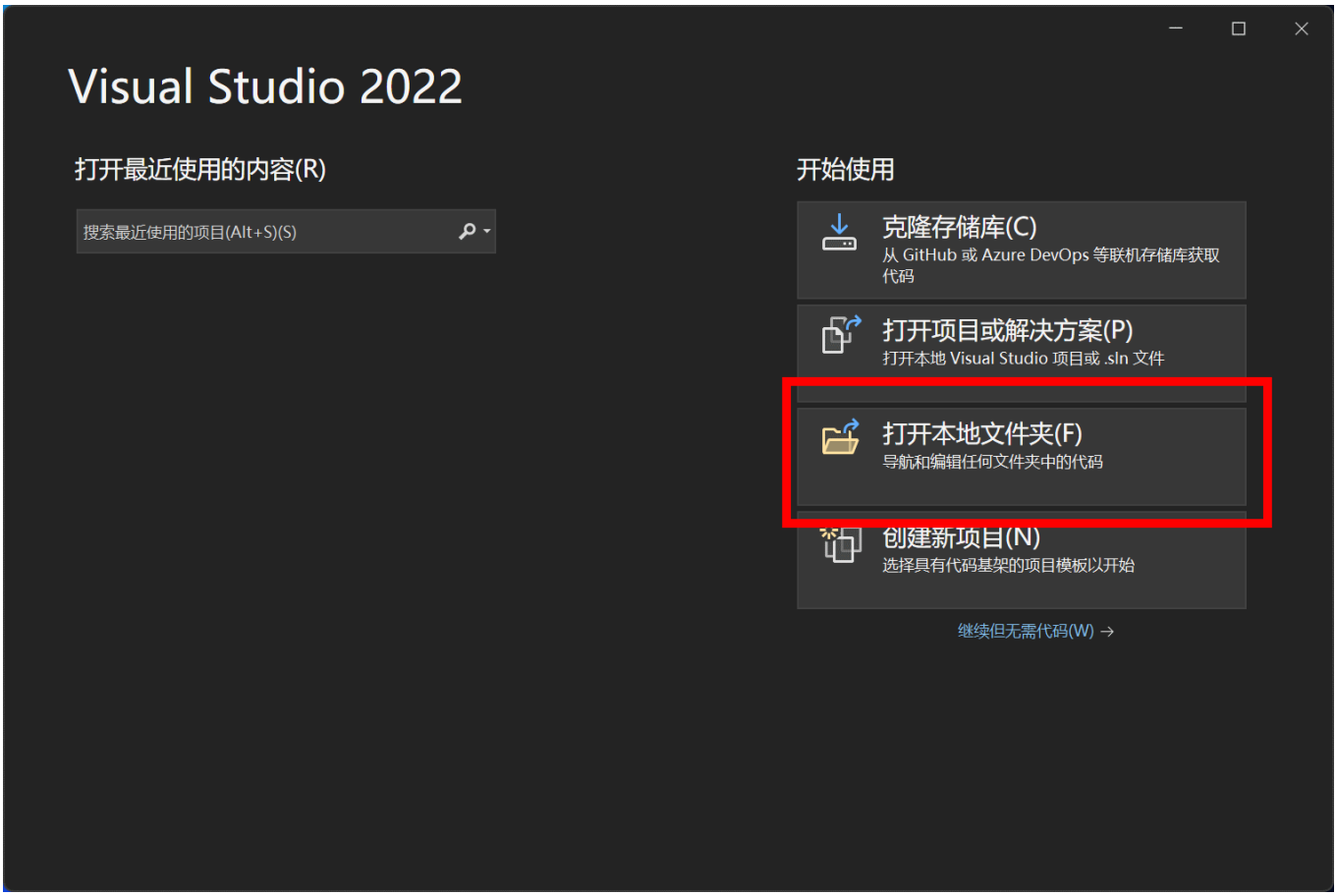
在linux下，可以使用gdb来调试你的程序。

另外，可以通过VSCode的remote功能连接linux虚拟机进行代码编写。

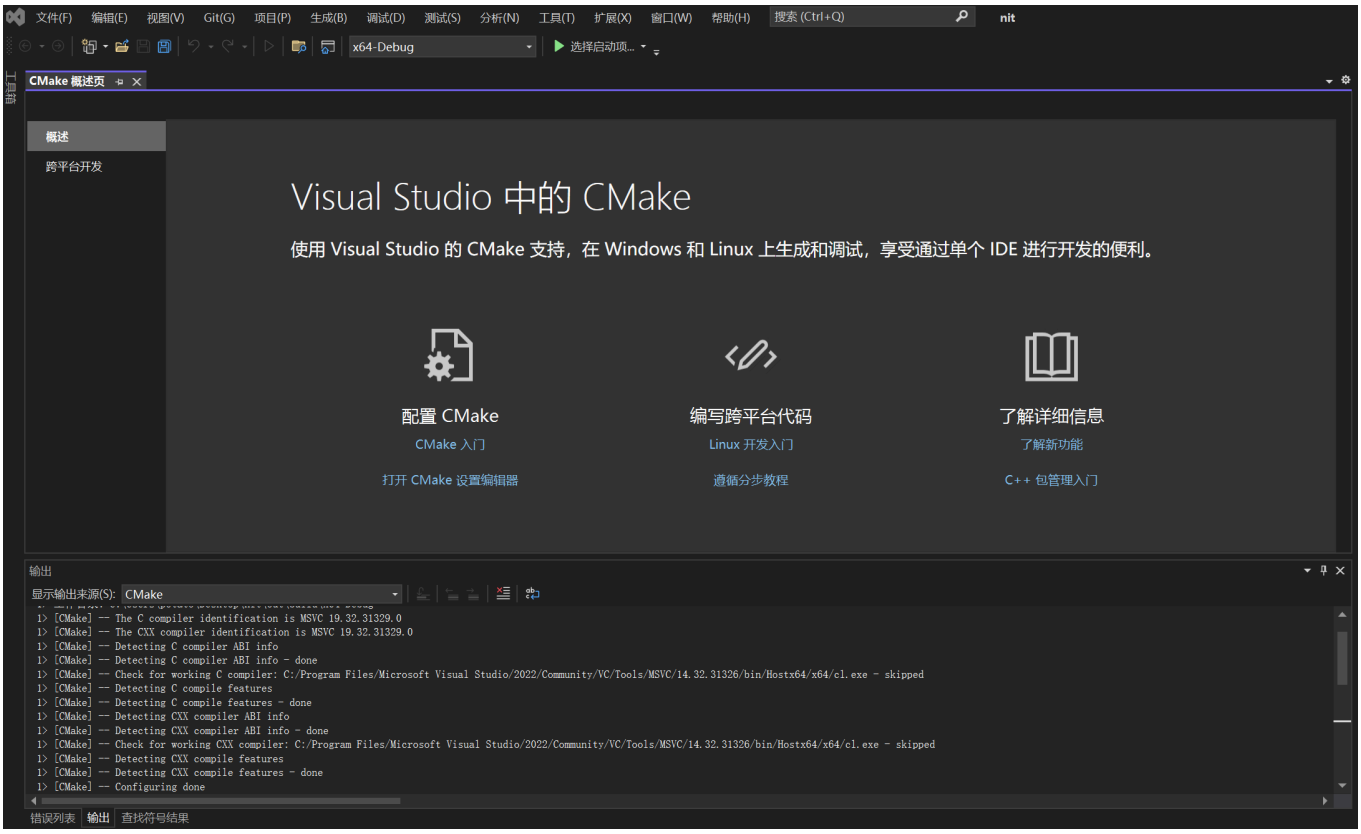
在Windows下使用VS Studio

我们强烈建议至少使用VS Studio 2022及以上版本。框架代码需要的C++版本不低于C++17。

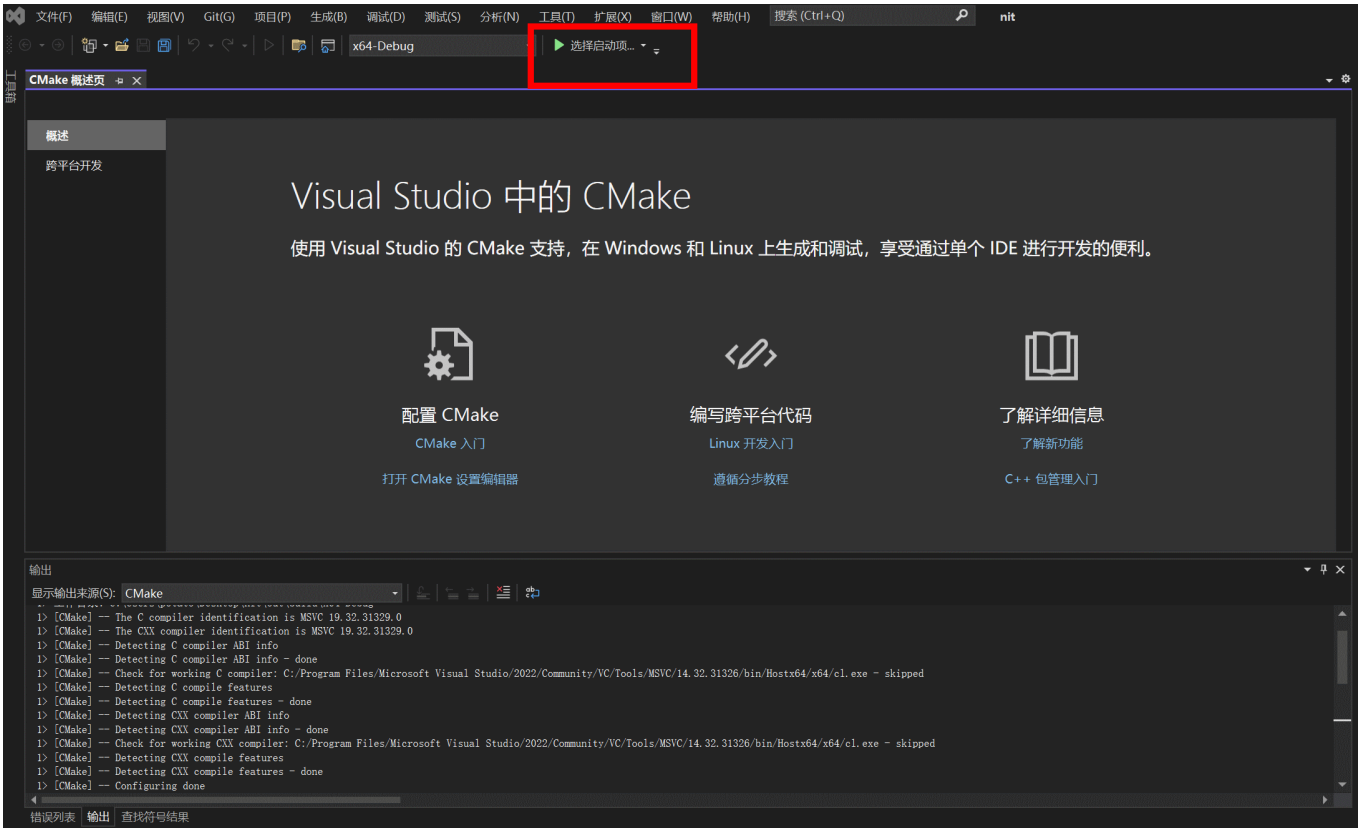
首先，打开VS Studio，选择"打开本地文件夹"。并在打开的文件选择界面中选择本项目框架代码文件夹打开。



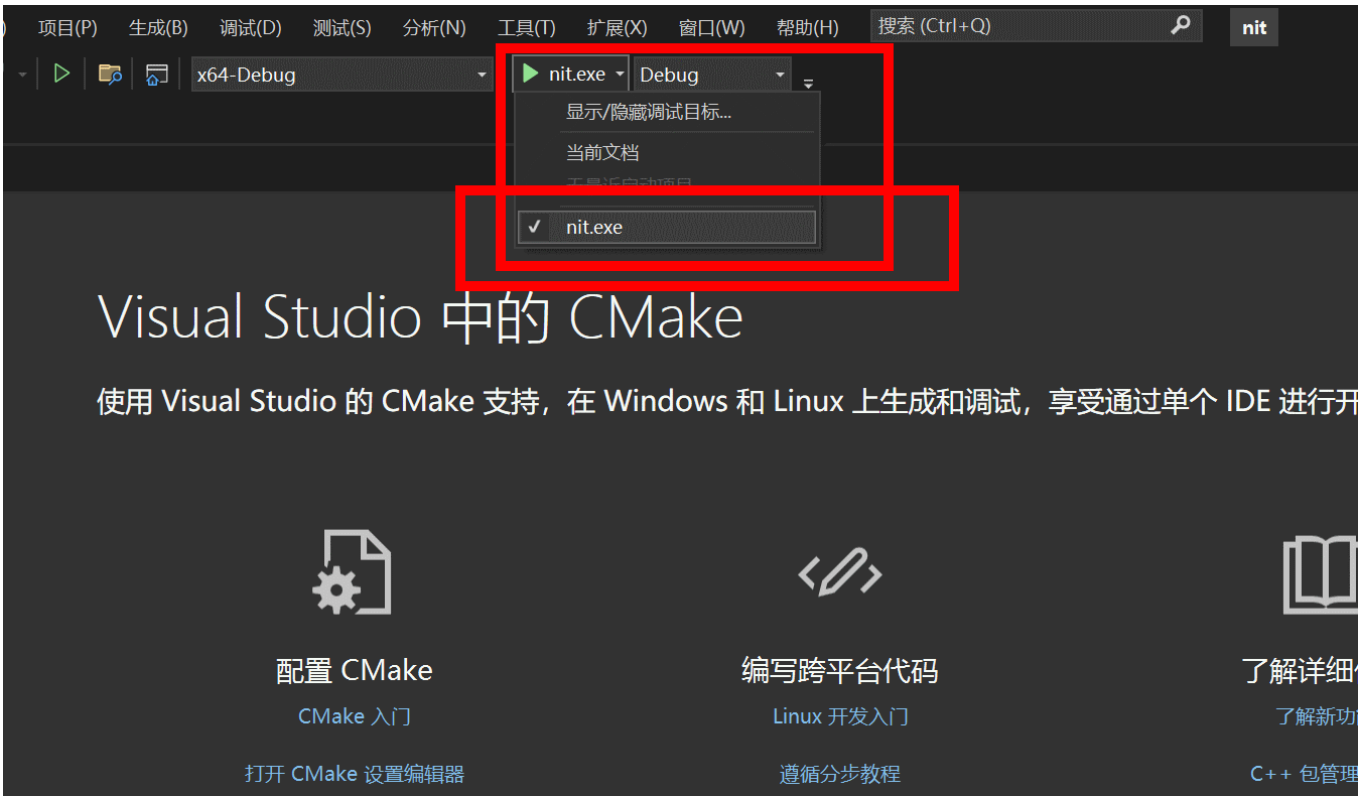
VS Studio会自动解析CMake文件，然后，你应该能看到类似这样的页面：



之后，点击位于页面上方的“选择启动项”：

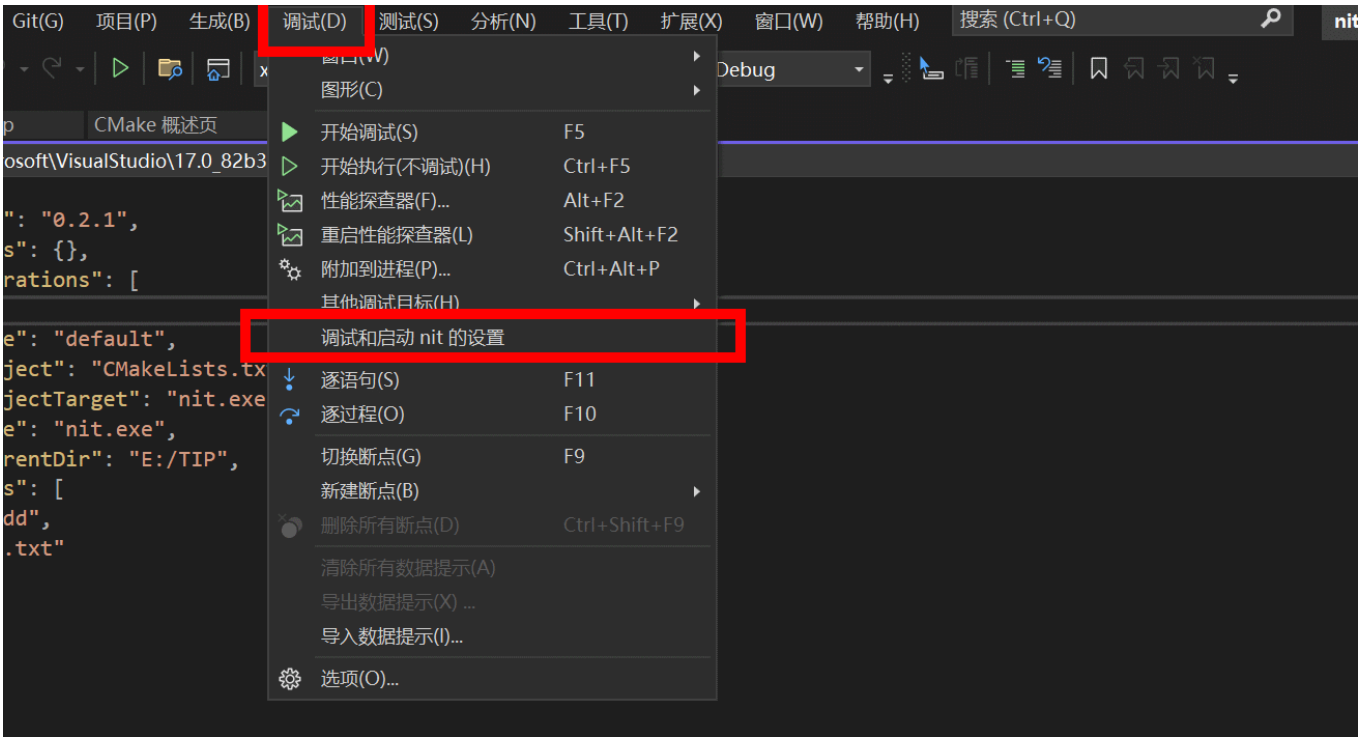


并在下拉的选项卡中选择“nit.exe”：



如果你希望直接点击调试按钮就可以进行调试，你可以进行如下操作：

选择顶部选项卡中的“调试”，并在下拉菜单中选择“调试和启动nit的设置”。



在弹出的launch.vs.json的编辑界面中，添加参数。

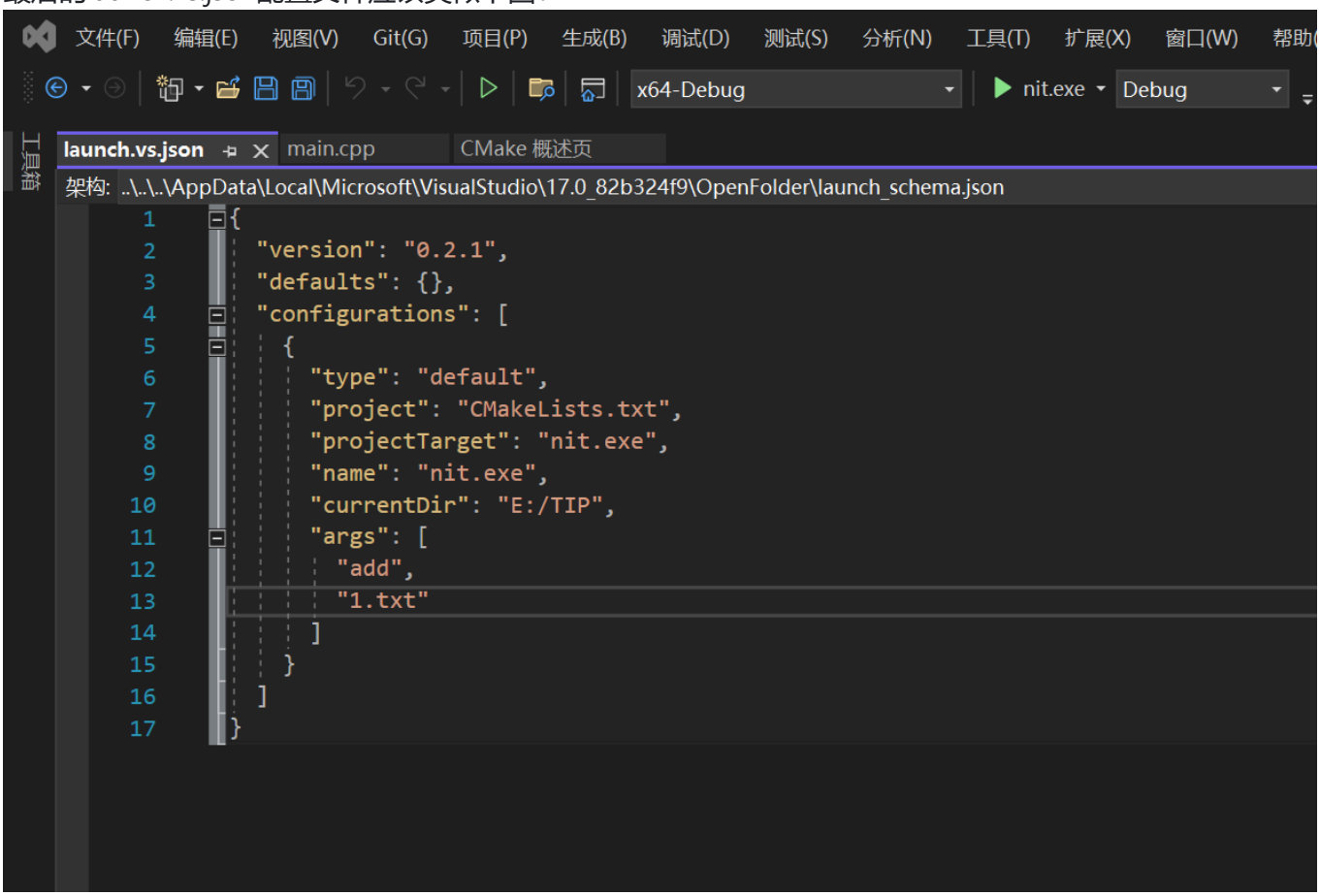
比如，如果你希望Nit将文件夹 `E:/TIP` 作为工作目录，并执行 `add` 命令，命令参数为 `1.txt` ，那么你就应该加入如下配置：

```
"currentDir": "E:/TIP",
"args": [
    "add",
```

```
    "1.txt"
]
```

第一项配置代表将`E:/TIP`设为工作目录（注意：使用`/`或`\\`来作为路径中的斜杠号），第二项配置代表向Nit传递的命令行参数为`add`和`1.txt`

最后的launch.vs.json配置文件应该类似下图：



完成配置后就可以直接点击调试按钮进行调试。

当然，也可以在cmd中直接运行可执行文件。

我不用CMake，也不用VS?

将框架代码中.cpp/.h文件全部拷贝到你的项目目录中，确保能编译成功即可。

使用API

所有的API声明都位于`apis.h`的`UsefulApi`命名空间中。添加代码：

```
#include "apis.h"
using namespace UsefulApi;
```

来使用api。当然，完全不使用我们的api也是ok的。

其中，核心的api为：

- UsefulApi::writeToFile
- UsefulApi::readFromFile

- UsefulApi::cwd
- UsefulApi::hash 我们提供了详细的注释。阅读`apis.h`来了解api的功能。