Environment

Preparation

- WSL or other Distribution of Linux
- CMake
- cpplint
- *Google Code Style*
- gtest
- clang-format
- git
- vscode

2. VS Code

2.1 Introduction

官方对vscode的描述是这样的

```
Visual Studio Code is a streamlined code editor with support for development
operations like debugging, task running, and version control. It aims to provide
just the tools a developer needs for a quick code-build-debug cycle and leaves
more complex workflows to fuller featured IDEs, such as Visual Studio IDE.
```

个人的使用体验是这样的:

- 简洁: 相比IDE来说，功能没那么冗杂，因此打开十分迅速
- 丰富: 支持许多extension
- 跨平台: 各种平台都可以使用并且，配置可以同步

总而言之，VS code是几乎可以满足所有的和coding相关的需求。

2.2 Configuration

配置令很多刚使用vscode的用户十分头疼，vscode的功能有很多又有很多插件，让刚接触的用户可犯了难。笔者曾经也深受困扰，但经过了一段时间的摸索之后，最开始令我很反感的配置，变得愈发精妙起来。
要想理解vscode中configuration的设计理念，先需要理解一个**工作区**(workspace)的概念

2.2.1 工作区(workspace)

vscode 的配置文件是包含层次关系的，而**工作区**的概念是为了让你配置一个工作环境, 让你更好地针对不同地环境（如JAVA环境，C++环境）设定不同地配置体验更好的VSCode。
vscode 中 configuration的层次关系大概是这样的:

```
系统默认设置（不可修改）-用户设置-工作区设置-文件夹设置
```

后者的设置会覆盖前者的设置，若没有设置某一项，将继续使用前者的设置。其中关于这些设置的理解：

- **用户设置**即全局设置，用户自行设定好后，每次打开VSCode即使用的此设定，若某项无设定即使用默认设置。
- **工作区设置**即工作环境设置，可对不同的工作环境是用不同的工作环境，若某项无设定，即使用上一层设置。
- **文件夹设置**即为项目设置，将一个文件夹当成一个项目，对同一个工作环境下的不同项目，使用不同的设置，若某项无设定，即使用上一层设置。

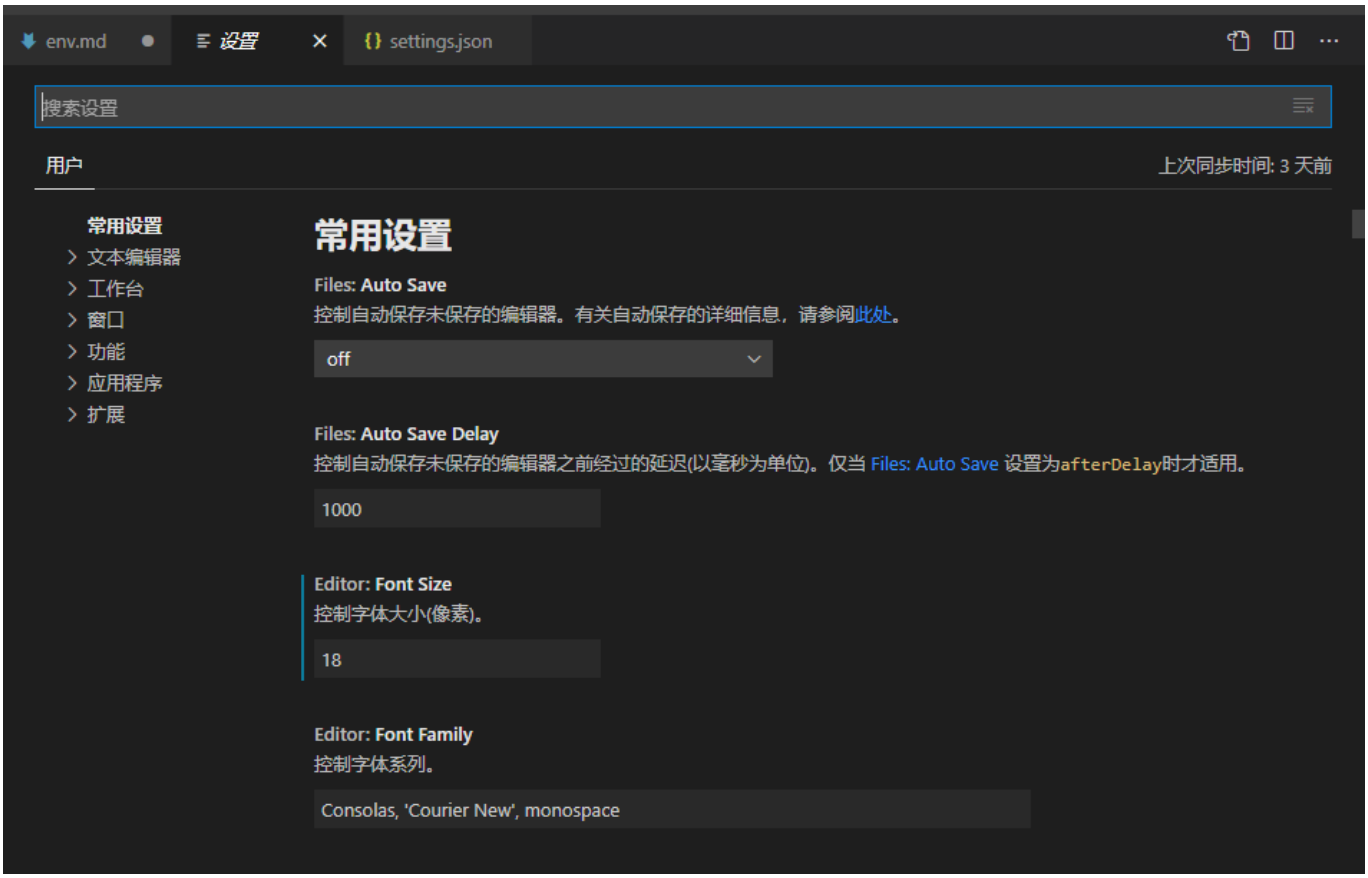
关于如何设置一个workspace，读者可以自行查阅相关资料。

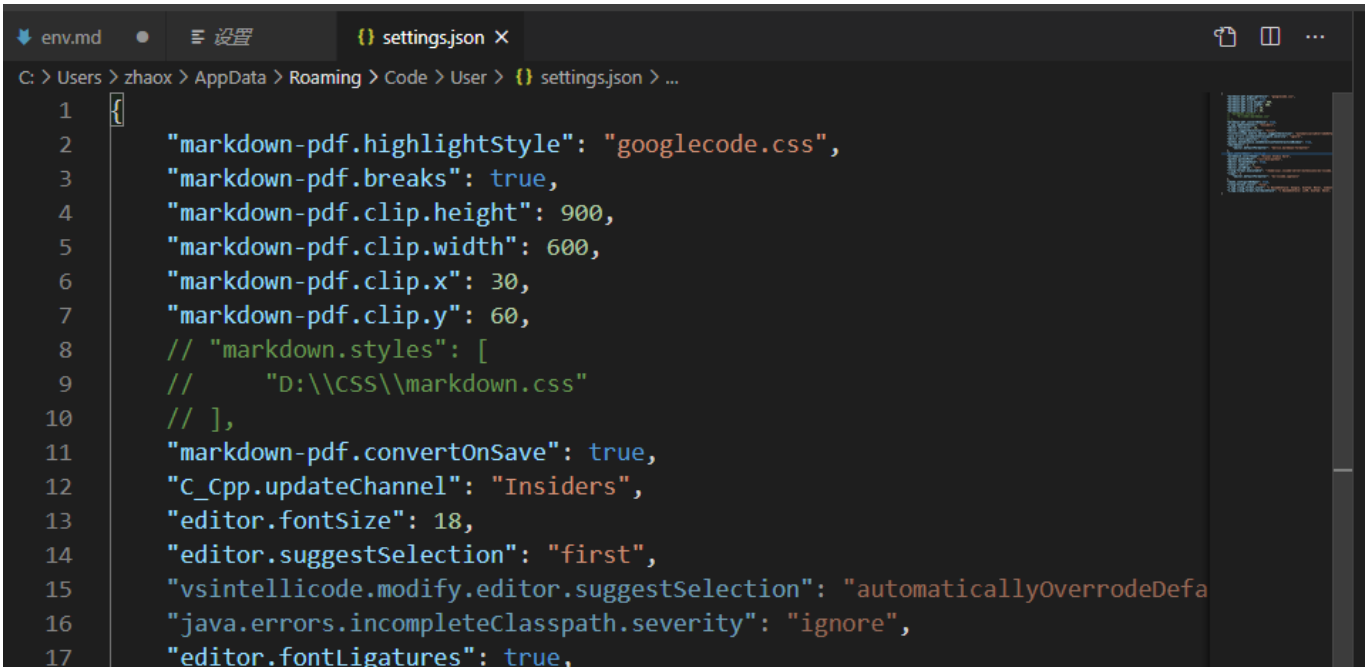
tips: 工作区中不仅仅包含配置文件，还可以包括启动脚本，编译脚本等等。

2.2.2 settings.json

总的来说，vscode的配置分为两种格式：

- GUI形式的配置
- json形式的配置





在GUI界面中，最上方会显示当前文件配置的是哪个工作区，而点击左上角的"打开设置"按钮，可以切换到对应的json文件。两种表达的相同的形式，值得注意的是，只有**默认**设置被改变才会出现在settings.json中，或者**手动点击**, 将设置复制为JSON文本。

可在settings.json中获得对应的条目。而利用settings.json来进行配置，在许多情况下发现是更高效的。

关于vscode的更多技巧，读者可以在实际使用过程中去摸索。

2.3 WSL+VScode

作为ms自家的应用，vscode在利用WSL上有天然的优势，使用vscode+WSL，可以得到甚至近乎原生Linux使用vscode的体验。具体使用可以参考相关资料

3. Format-tool : Clang-format

它是基于clang的一个命令行工具，能够自动化格式C/C++/Obj-C代码，支持多种代码风格：Google, Chromium, LLVM, Mozilla, WebKit，也支持自定义风格（通过编写.clang-format文件）很方便的统一代码格式。
这里介绍通过vscode来集成使用clang-format，vscode中也可以使用ms-vscode.cpptools。

3.1 配置 clang-format extension

推荐看一下这篇博客
https://blog.csdn.net/core571/article/details/82867932

其中关于style的问题，clang-format自动调用workspace的.clang-format文件，也可以使用key-value的形式进行配置，比如我现在使用的是这样

```
"C_Cpp.clang_format_style": "{ BasedOnStyle: Google, UseTab: Never, IndentWidth: 4, TabWidth: 4, BreakBeforeBraces: Attach, AllowShortIfStatementsOnASingleLine: false, IndentCaseLabels: false, ColumnLimit: 0, AccessModifierOffset: -4 }",
```

4. CMake

4.1 Introduction

Make 工具有很多种，例如 GNU Make ， QT 的 qmake ， 微软的 MS nmake，BSD Make（pmake），Makepp，等等。这些 Make 工具遵循着不同的规范和标准，所执行的 Makefile 格式也千差万别。这样就带来了一个严峻的问题：如果软件想跨平台，必须要保证能够在不同平台编译。而如果使用上面的 Make 工具，就得为每一种标准写一次 Makefile ， 这将是一件让人抓狂的工作。
CMake就是针对上面问题所设计的工具：它首先允许开发者编写一种平台无关的 CMakeList.txt 文件来定制整个

编译流程，然后再根据目标用户的平台进一步生成所需的本地化 Makefile 和工程文件，如 Unix 的 Makefile 或 Windows 的 Visual Studio 工程。从而做到“Write once, run everywhere”。显然，CMake 是一个比上述几种 make 更高级的编译配置工具。一些使用 CMake 作为项目架构系统的知名开源项目有 VTK、ITK、KDE、OpenCV、OSG 等。
在 linux 平台下使用 CMake 生成 Makefile 并编译的流程如下：

- 编写 CMake 配置文件 CMakeLists.txt 。
- 执行命令 cmake PATH 或者 ccmake PATH 生成 Makefile 1 1ccmake 和 cmake 的区别在于前者提供了一个交互式的界面。。其中， PATH 是 CMakeLists.txt 所在的目录。
- 使用 make 命令进行编译。

总结起来，Cmake是一个**自动化的项目构建工具**, (autotools) 用来生成对应平台的Makefile文件，再使用make工具进行构建。

Tips: 在以项目为单位构建时，应该为每个目录都编写一份CMakeLists.txt.

4.2 实践

4.2.1 Preparation

- cmake
- make
- g++

4.2.2 第一个尝试

在用户目录下创建目录t1, 并进入t1目录。

```
$ cd ~
$ mkdir t1
$ cd t1
```

并编写main.c和CMakeLists.txt
main.c文件内容：

```
#include <stdio.h>
int main()
{
   printf("Hello World from t1 Main!\n");
   return 0;
}
```

CmakeLists.txt文件内容：

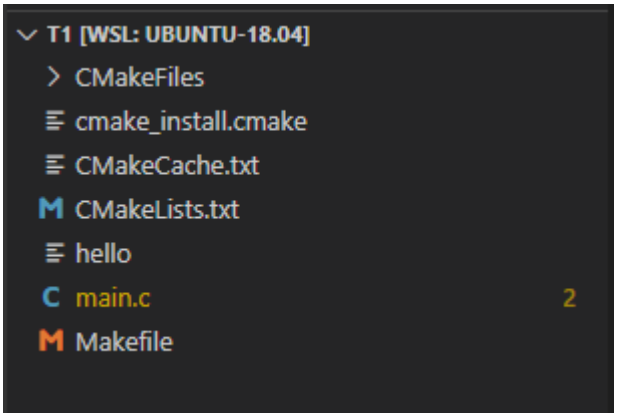
```
PROJECT(HELLO)
SET(SRC_LIST main.c)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
```

```
MESSAGE(STATUS "This is SOURCE dir "${HELLO_SOURCE_DIR})
ADD_EXECUTABLE(hello ${SRC_LIST})
```

执行

```
$ cmake .
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- This is BINARY dir /home/zxy/cmake/t1
-- This is SOURCE dir /home/zxy/cmake/t1
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zxy/cmake/t1
$ make
[ 50%] Building C object CMakeFiles/hello.dir/main.c.o
[100%] Linking C executable hello
[100%] Built target hello
$ ./hello
Hello World from t1 Main!
```

再让我们看一下目录中的内容：



你会发现，系统自动生成了：
CMakeFiles，CMakeCache.txt，cmake_install.cmake等文件，并且生成了Makefile。
到这里，我们的第一个尝试就完成了，他成功的生成了一个**可执行的二进制文件**hello。

4.2.3 CMakeLists.txt

我们来重新看一下CMakeLists.txt，这个文件是cmake的构建定义文件，文件名是大小写相关的，如果工程存在多个目录，需要确保每个要管理的目录都存在一个CMakeLists.txt。(关于多目录构建，后面我们会提到，这里不作过多解释)。

上面例子中的CMakeLists.txt文件内容如下：

```
PROJECT (HELLO)
SET(SRC_LIST main.c)
MESSAGE(STATUS "This is BINARY dir " ${HELLO_BINARY_DIR})
MESSAGE(STATUS "This is SOURCE dir "${HELLO_SOURCE_DIR})
ADD_EXECUTABLE(hello ${SRC_LIST})
```

PROJECT指令的语法是：

```
PROJECT(projectname [CXX] [C] [Java])
```

你可以用这个指令定义工程名称，并可指定工程支持的语言，支持的语言列表是可以忽略的，这个指令隐式的定义了两个cmake变量:
`<projectname>_BINARY_DIR` 以及 `<projectname>_SOURCE_DIR`，这里就是 `HELLO_BINARY_DIR` 和 `HELLO_SOURCE_DIR` (所以CMakeLists.txt中两个MESSAGE指令可以直接使用了这两个变量)，因为采用的是**内部编译**，两个变量目前指的都是工程所在路径`~/cmake/t1`，后面我们会讲到**外部编译**，两者所指代的内容会有所不同。
同时cmake系统也帮助我们预定义了 `PROJECT_BINARY_DIR` 和 `PROJECT_SOURCE_DIR` 变量，他们的值分别跟 `HELLO_BINARY_DIR` 与 `HELLO_SOURCE_DIR` 一致。

为了统一起见，建议以后直接使用 `PROJECT_BINARY_DIR`，`PROJECT_SOURCE_DIR`，即使修改了工程名称，也不会影响这两个变量。如果使用了`<projectname>_SOURCE_DIR`，修改工程名称后，需要同时修改这些变量。
SET指令的语法是：

```
SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
```

现阶段，你只需要了解SET指令可以用来显式的定义变量即可。比如我们用到的是SET(SRC_LIST main.c)，如果有多个源文件，也可以定义成：

```
SET(SRC_LIST main.c t1.c t2.c)
```

MESSAGE指令的语法是：

```
MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] "message to display"...)
```

这个指令用于向终端输出用户定义的信息，包含了三种类型:

- SEND_ERROR，产生错误，生成过程被跳过。
- SATUS，输出前缀为—的信息。
- FATAL_ERROR，立即终止所有cmake过程。

我们在这里使用的是STATUS信息输出，演示了由PROJECT指令定义的两个隐式变量HELLO_BINARY_DIR和HELLO_SOURCE_DIR。

```
ADD_EXECUTABLE(hello ${SRC_LIST})
```

定义了这个工程会生成一个文件名为hello的**可执行文件**，相关的源文件是SRC_LIST中定义的源文件列表， 本例中你也可以直接写成 `ADD_EXECUTABLE(hello main.c)`。

在本例我们使用了${}来**引用变量*，这是cmake的变量应用方式，但是，有一些例外，比如在**IF控制语句**，变量是直接使用**变量名**引用，而不需要${}。如果使用了${}去应用变量，其实IF会去判断名为${}所代表的值的变量，那当然是不存在的了。

将本例改写成一个最简化的CMakeLists.txt：

```
PROJECT(HELLO)
ADD_EXECUTABLE(hello main.c)
```

4.2.4 语法规则

最简单的语法规则是：

1. 变量使用${}方式取值，但是在IF控制语句中是直接使用变量名
2. 指令(参数1 参数2...)参数使用括弧括起，参数之间使用空格或分号分开。*以上面的ADD_EXECUTABLE指令为例，如果存在另外一个func.c源文件，就要写成：ADD_EXECUTABLE(hello main.c func.c)或者 ADD_EXECUTABLE(hello main.c;func.c)*
3. 指令是大小写无关的，参数和变量是大小写相关的。但，推荐你全部使用大写指令。

4.2.5 常用指令

指定 cmake 的最小版本

```
cmake_minimum_required(VERSION 3.4.1)
```

这行命令是可选的，我们可以不写这句话，但在有些情况下，如果 CMakeLists.txt 文件中使用了一些高版本 cmake 特有的一些命令的时候，就需要加上这样一行，提醒用户升级到该版本之后再执行 cmake。

设置项目名称

```
project(demo)
```

这个命令不是强制性的，但最好都加上。它会引入两个变量 demo_BINARY_DIR 和 demo_SOURCE_DIR，同时，cmake 自动定义了两个等价的变量 PROJECT_BINARY_DIR 和 PROJECT_SOURCE_DIR。

设置编译类型

```
add_executable(demo demo.cpp) # 生成可执行文件
add_library(common STATIC util.cpp) # 生成静态库
add_library(common SHARED util.cpp) # 生成动态库或共享库
```

add_library 默认生成是静态库，通过以上命令生成文件名字，
在 Linux 下是：

- demo
- libcommon.a
- libcommon.so

在 Windows 下是：

- demo.exe
- common.lib
- common.dll

4.2.6 指定编译包含的源文件

明确指定包含哪些源文件

```
add_library(demo demo.cpp test.cpp util.cpp)
```

搜索所有的 cpp 文件

`aux_source_directory(dir VAR)` 发现一个目录下所有的源代码文件并将列表存储在一个变量中。

```
aux_source_directory(. SRC_LIST) # 搜索当前目录下的所有.cpp文件
add_library(demo ${SRC_LIST})
```

自定义搜索规则

```
file(GLOB SRC_LIST "*.cpp" "protocol/*.cpp")
add_library(demo ${SRC_LIST})
# 或者
file(GLOB SRC_LIST "*.cpp")
file(GLOB SRC_PROTOCOL_LIST "protocol/*.cpp")
add_library(demo ${SRC_LIST} ${SRC_PROTOCOL_LIST})
# 或者
aux_source_directory(. SRC_LIST)
aux_source_directory(protocol SRC_PROTOCOL_LIST)
add_library(demo ${SRC_LIST} ${SRC_PROTOCOL_LIST})
```

4.2.7 查找指定的库文件

find_library(VAR name path)查找到指定的预编译库，并将它的路径存储在变量中。
默认的搜索路径为 cmake 包含的系统库，因此如果是 NDK 的公共库只需要指定库的 name 即可。

```
find_library( # Sets the name of the path variable.
              log-lib

              # Specifies the name of the NDK library that
              # you want CMake to locate.
              log )
```

类似的命令还有 find_file()、find_path()、find_program()、find_package()。

4.2.8 拓展

到这里关于CMake的一些基础知识就结束了，

更多相关资料可以参考CMake官网，以及
https://github.com/Akagi201/learning-cmake

5. 版本控制系统: Git

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

Git 与常用的版本控制工具 CVS, Subversion 等不同，它采用了分布式版本库的方式，不必服务器端软件支持。

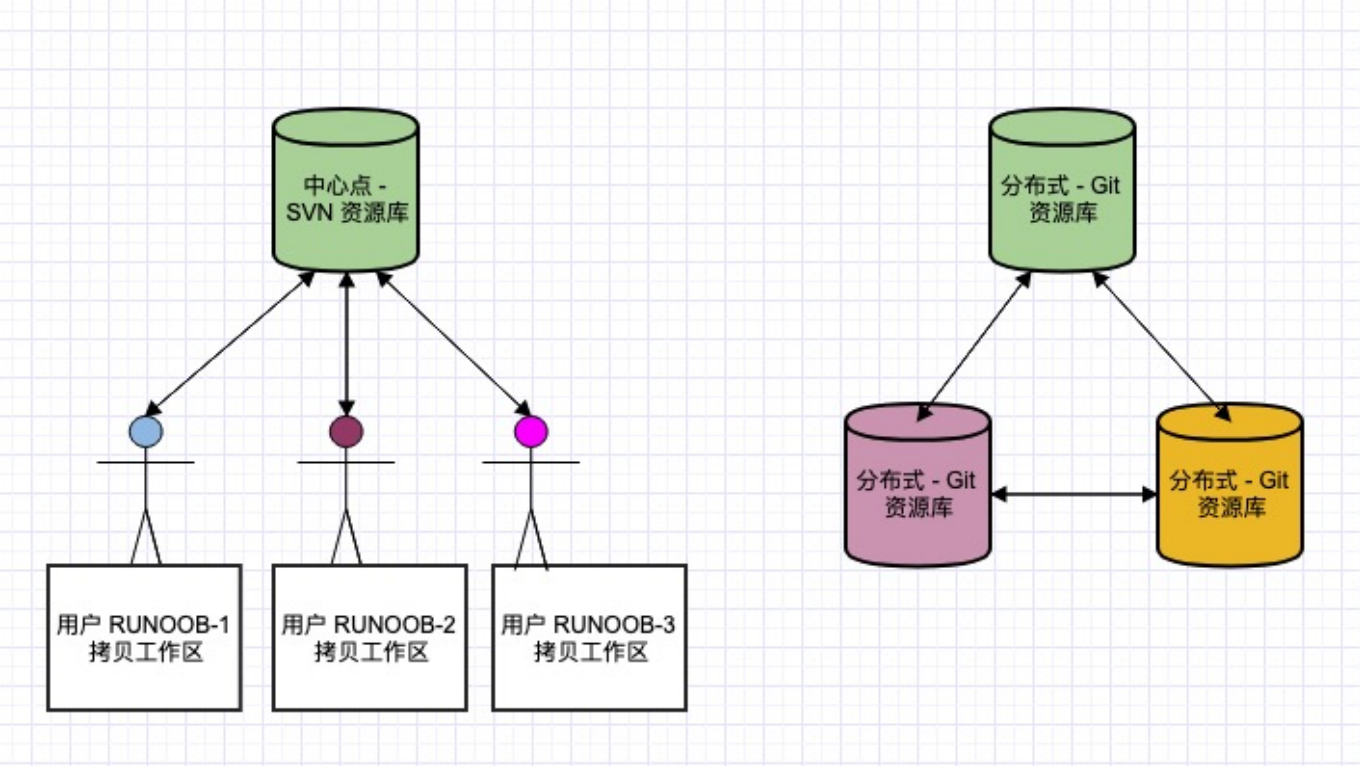
5.1 Git 与 SVN 区别

Git 不仅仅是个版本控制系统，它也是个内容管理系统(CMS)，工作管理系统等。

如果你是一个具有使用 SVN 背景的人，你需要做一定的思想转换，来适应 Git 提供的一些概念和特征。
Git 与 SVN 区别点：

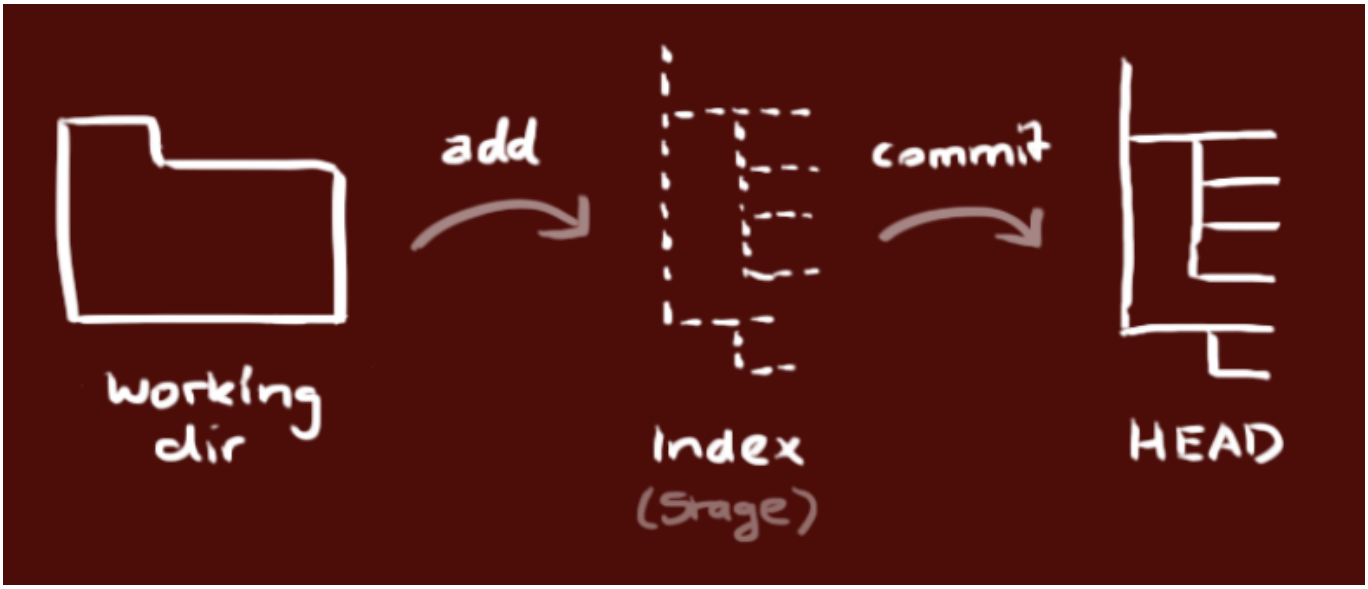
- Git 是分布式的，SVN 不是：这是 Git 和其它非分布式的版本控制系统，例如 SVN，CVS 等，最核心的区别。
- Git 把内容按**元数据**方式存储，而 SVN 是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似 .svn、.cvs 等的文件夹里。
- Git 分支和 SVN 的分支不同：分支在 SVN 中一点都不特别，其实它就是版本库中的另外一个目录。
- Git 没有一个全局的版本号，而 SVN 有：目前为止这是跟 SVN 相比 Git 缺少的最大的一个特征。

- Git 的内容完整性要优于 SVN：Git 的内容存储使用的是 SHA-1 哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。



5.2 Pipeline

你的本地**仓库**(repository)由 git 维护的三棵"树"组成。第一个是你的工作目录，它持有实际文件；第二个是暂存区（Index），它像个缓存区域，临时保存你的改动；最后是 HEAD，它指向你最后一次提交的结果。



5.2.1 init repository

把本地目录初始化为一个**仓库**(repository)

```
git init
```

从远程克隆一个仓库

```
git clone username@host:/path/to/repository
```

5.2.1 add & commit

你可以提出更改（把它们添加到`storge`），使用如下命令：

```
git add <filename>
git add *
```

这是 git 基本工作流程的第一步；使用如下命令以实际提交改动：

```
git commit -m "message"
```

现在，你的改动已经提交到了 HEAD，但是还没到你的远端仓库。

5.2.2 push

你的改动现在已经在本地仓库的 HEAD 中了。执行如下命令以将这些改动提交到远端仓库：

```
git push origin master
```

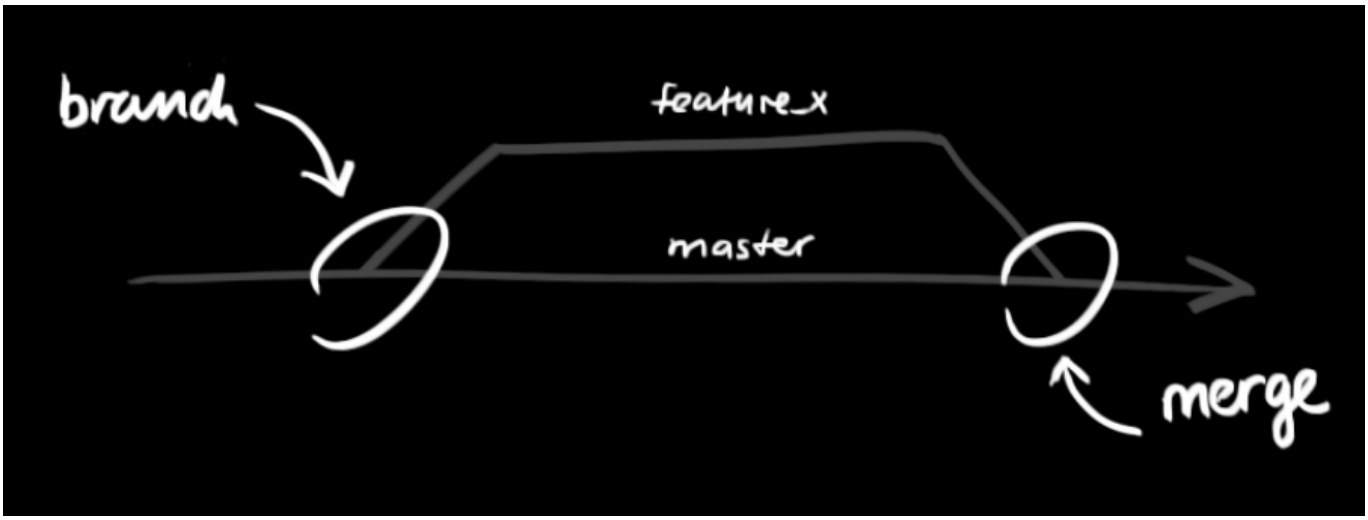
可以把 master 换成你想要推送的任何**分支**(branch)。
如果你还没有克隆现有仓库，并欲将你的仓库连接到某个远程服务器，你可以使用如下命令添加：

```
git remote add origin <server>
```

如此你就能够将你的改动推送到所添加的服务器上去了。

5.2.3 分支：Branch

分支是用来将特性开发绝缘开来的。在你创建仓库的时候，master 是“默认的”分支。在其他分支上进行开发，完成后再将它们合并到主分支上



创建一个叫做“feature_x”的分支，并切换过去：

```
git checkout -b feature_x
```

切换回主分支：

```
git checkout master
```

再把新建的分支删掉：

```
git branch -d feature_x
```

除非你将分支推送到远端仓库，不然该分支就是 不为他人所见的：

```
git push origin <branch>
```

5.2.4 更新与合并

要更新你的本地仓库至最新改动，执行:

```
git pull
```

以在你的工作目录中 **获取**（fetch） 并 **合并**（merge） 远端的改动。
要合并其他分支到你的当前分支（例如 master），执行：

```
git merge <branch>
```

在这两种情况下，git 都会尝试去自动合并改动。遗憾的是，这可能并非每次都成功，并可能出现**冲突**（conflicts）。这时候就需要你修改这些文件来**手动**合并这些**冲突**（conflicts）。改完之后，你需要执行如下命令以将它们标记为合并成功：

```
git add <filename>
```

在合并改动之前，你可以使用如下命令预览差异：

```
git diff <source_branch> <target_branch>
```

tips:git pull 实际上执行的就是git fetch与git merge.

5.2.5 替换本地改动

假如你操作失误（当然，这最好永远不要发生），你可以使用如下命令替换掉本地改动：

```
git checkout -- <filename>
```

此命令会使用 HEAD 中的最新内容替换掉你的工作目录中的文件。已添加到暂存区的改动以及新文件都不会受到影响。
假如你想丢弃你在本地的所有改动与提交，可以到服务器上获取最新的版本历史，并将你本地主分支指向它：

```
git fetch origin
git reset --hard origin/master
```

6. References

1. CMakeLists.txt 语法介绍与实例演练
2. learning-cmake
3. git - 简明指南
4. Git五分钟教程
5. Git 教程