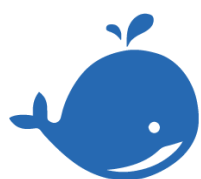


自制深度学习推理框架-第一课

本课程赞助方：Datawhale

作者：[傅莘莘](#)

特别感谢：[散步](#)，[mlmz](#)，[沉迷单车的追风少年](#)



Datawhale

KuiperInfer

项目介绍

项目地址

<https://github.com/zjhelloworld/KuiperInfer> , 欢迎大家点赞和PR.

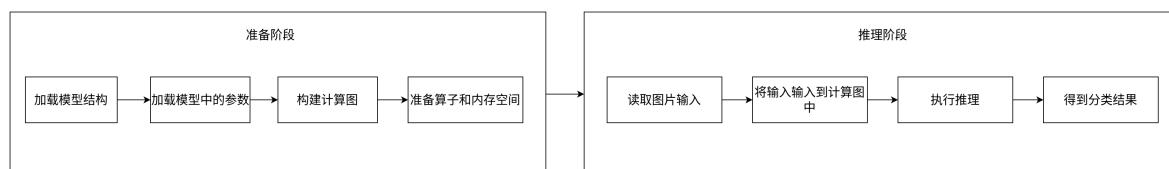
本次课的代码地址

<https://github.com/zjhelloworld/kuiperdatawhale>

什么是推理框架

深度学习推理框架用于对**已经训练完成的神经网络模型文件进行加载**，并根据模型文件中的**网络结构和权重参数**对输入图像进行预测。换句话说，深度学习推理框架就是将深度学习训练框架Pytorch和TensorFlow中训练完成的模型，移植到中心侧和端侧并且在运行时**高效执行**。另外，与深度学习训练框架不同的是，推理框架没有梯度后向传播的过程，因为在**推理阶段模型的权重已经固定**，不需要利用后向传播技术进一步进行调整。

例如对于一个 Resnet 分类网络的模型，深度学习推理框架先对模型文件中的网络结构进行读取和载入，再读取模型文件中的权重参数和其他参数、属性信息填入到 Resnet 网络结构中，随后推理框架**将不同的图像放入到计算图的输入中，并执行预测过程**，从而得到其归属的类别。以下的图示是我对如上内容的总结：



关于KuiperInfer的技术全景概述

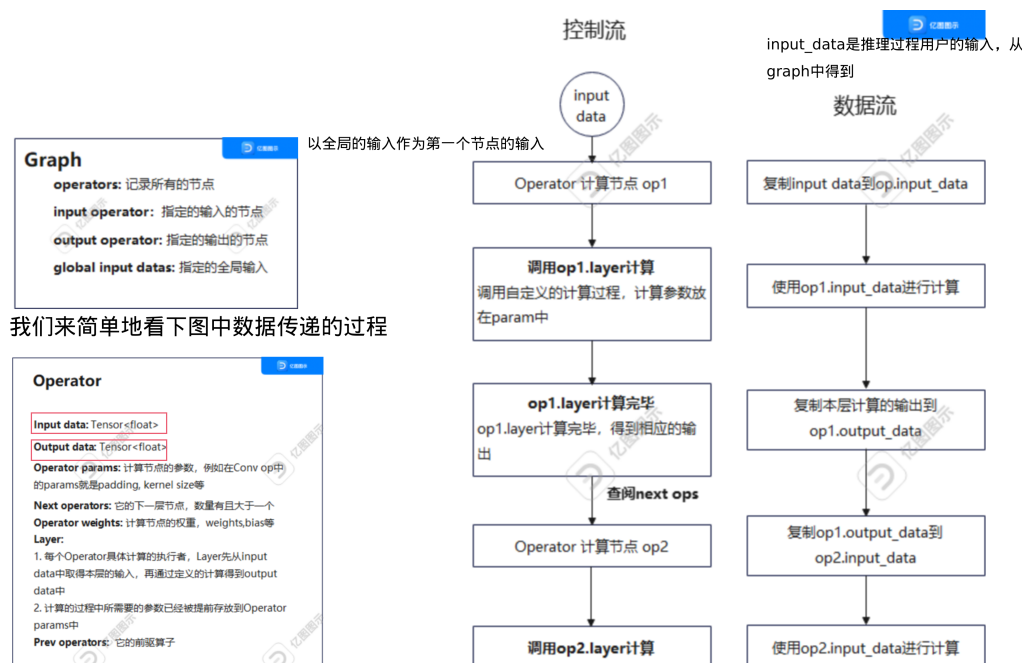
我们在这里对本课程的成品项目 KuiperInfer 进行分模块介绍，**给同学们留一个初步的整体印象**。顺便说一句名字的来源，Kuiper 行星带是太阳系外围的一个区域，行星带中包含着大量的冰冻小行星、彗星和其他冰质天体。之所以取这个名字，是因为我希望这个框架具有一定“边缘”属性，**另外也是希望更多的人像“小行星”一样加入到星带一样，加入到这个开源项目中。**

KuiperInfer 可以分为以下的几个模块：

1. **Operator**: 深度学习计算图中的计算节点，包含以下的几个部分：
 - **存储输入输出的张量**，用于存放深度学习中各层的输入输出。比如对于一个 Convolution 层，需要一部分空间来保存计算的输入和输出。
 - **计算节点的类型和名称**，计算节点类型可以有 Convolution, Relu, Maxpooling 等，计算节点的**名称是唯一的**，用来区分任意一个节点，可以是 Convolution_1, Convolution_2 等。
 - **计算节点参数信息**，例如卷积中的步长、卷积核的大小等。
 - **计算节点的权重信息**，例如卷积节点中的 weight, bias 权重。
2. **Graph**: 有多个 Operator 串联得到的有向无环图，规定了各个计算节点（Operator）执行的流程和顺序。
3. **Layer**: 计算节点中运算的具体执行者，Layer 类先读取输入张量中的数据，然后对输入张量进行计算，得到的结果存放到计算节点的输出张量中，**当然，不同的算子中 Layer 的计算过程会不一致。**

4. **Tensor**: 用于存放**多维数据**的数据结构，方便数据在计算节点之间传递，同时该结构也封装矩阵乘、点积等与矩阵相关的基本操作。

以下的图示是对如上的模块的总结，每个节点都从输入张量 **input_data** 中读取数据，并调用该节点对应的 **Layer** 计算对应的结果，最后再将结果放入到 **output_data** 中。整个计算图第一个节点的输入也是计算图全局的输入，同时，最后一个节点的输出也是整个计算图的全局输出。



这是一个概览，同学们在目前阶段留一个印象即可，随着课程的进行我们会逐步分析其中的模块，直到掌握全局。

常见的问题

1. 对C++基础的要求是什么？如果C++水平不高怎么办？
 - 需要学过C++或者C语言，如果对自己的C++水平不够自信，可以自行阅读《C++ Primer》一书。另外在课程中，我也会穿插着去讲C++的高级语法，只要有一定基础的同学，我觉得这方面不用太担心。
2. 对AI基础的要求，会很高吗？
 - AI基础不做很高要求，但是需要知道卷积、池化、激活等基本概念，对常见模型有一定的了解，例如 **Resnet**, **Yolo**, **U-net** 等。
3. **KuiperInfer** 和本门课程有什么关系？
 - **KuiperInfer** 是本课程的上游项目，该项目的作者和本门课的作者都是同一人，而且 **KuiperInfer** 是**专门为了课程讲解而开发**的一个

教学性质框架，以在设计上标准化，编码上简单化作为目标。

- 另外在课程结束后，每位跟学跟练的同学都会具有**开发同等级推理框架的技术实力**。同时，同学们也会**逐步写出一个属于自己的深度学习AI推理框架**。在全民AI的时代，这将是一个亮点不小的个人项目，对大家以后的求职可能也会非常有帮助。

功能演示

首先要说明的是，进行功能演示的项目是我们课程结束后的成品，有基础的同学也可以自行克隆（[项目地址](#)）项目编译并调试，**但是不推荐大家现在花费过多的时间在编译项目和环境配置上**。参加的同学如果能从头到位跟听跟练课程，都能获得一个功能接近的深度学习推理框架，**具体请看视频演示**。

从零开始的环境搭建

根据第一次开课的经验教训，我们本次课程直接使用 Docker 来搭建项目的环境，请同学们跟着视频和以下步骤来一步步做。

Docker 的安装

Windows 下的 Docker 快速配置

TODO By sanbu

Linux 或 WSL 下的 Docker 快速配置

TODO By sanbu

环境准备

1. 拉取Docker镜像：

```
sudo docker pull registry.cn-hangzhou.aliyuncs.com/hellofss/kuiperinfer:datawhale
```

2. 创建本地文件夹，并将课程代码克隆到该文件夹中，这里我们用 `~/code/kuipercourse` 作为本地文件夹。

```
mkdir ~/code/kuiperdatawhale
cd ~/code/kuiperdatawhale
git clone
https://github.com/zjhellofss/kuiperdatawhale.git
```

3. 创建并运行一个镜像的容器。

```
sudo docker run -it registry.cn-
hangzhou.aliyuncs.com/hellofss/kuiperinfer:datawhale
/bin/bash
```

4. 在容器中输入 `ifconfig` 命令查看 `ip` 地址。

```
eth0: flags=4163<UP, BROADCAST, RUNNING, MULTICAST> mtu
1500
        inet 172.17.0.4 netmask 255.255.0.0 broadcast
172.17.255.255
        ether 02:42:ac:11:00:04 txqueuelen 0
(Ethernet)
        RX packets 55 bytes 8479 (8.4 KB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 0 bytes 0 (0.0 B)
        TX errors 0 dropped 0 overruns 0 carrier 0
collisions 0
```

5. 尝试使用 `ssh` 命令连接容器，这里的用户名固定是 `me`，`ip` 地址是上方 `ipconfig` 输出中的 `inet`，登录密码是 1。

```
ssh me@172.17.0.4
```

如果出现连接超时问题，可以尝试替换映射端口或自行查阅资料解决。

6. **(可选)** `me` 用户的权限或者对文件夹的读写权限，可以用 `root` 用户进行再次分配，`root` 用户也就是在步骤3中运行容器时登录的用户。

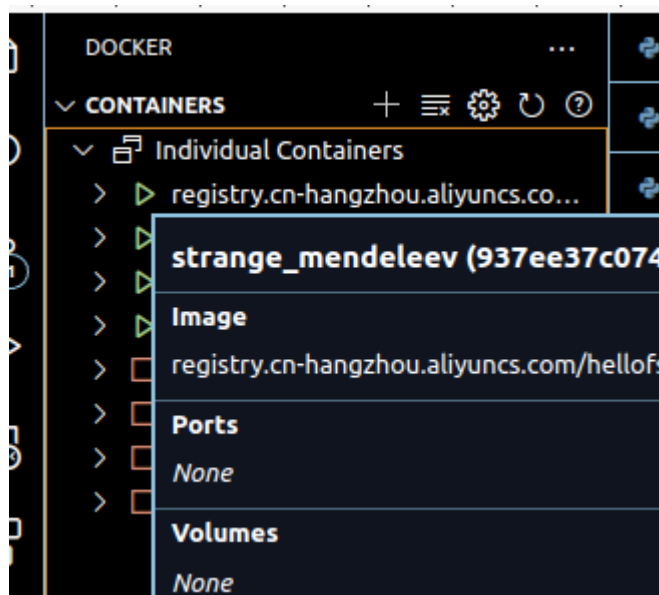
至此我们已经顺利完成了环境的准备，接下来是时候选择一款心仪的 IDE 作为自己的开发工具，考虑到每个人喜好不同，在这里提供了 VSCode 和 Clion 的配置指南，你只需要选择最喜欢的即可。

用 VSCode 连接容器

1. 在 VSCode 中，连接容器是一件很简单的事情，首先你需要安装 Docker 插件，点击左侧的扩展栏（或是按下 `ctrl+shift+X`），键入 Docker，随后选择安装：



安装后，你将会看到左侧出现一个相同长相的小鲸鱼，左键点击它你将会看到我们启动的所有容器：



2. 接下来，你需要右键之前启动的容器 `registry.cn-hangzhou.aliyuncs.com/hellofss/kuiperinfer:datawhale`，选择 **附加 Visual Studio Code** 进入 Docker 环境：

在进入 Docker 环境后，我们还需要手动再次 clone 课程代码，你可以执行 `ctrl + J` 调出控制台终端，在终端中输入：

```
cd /home
git clone
https://github.com/zjhelloworldss/kuiperdatawhale.git
```

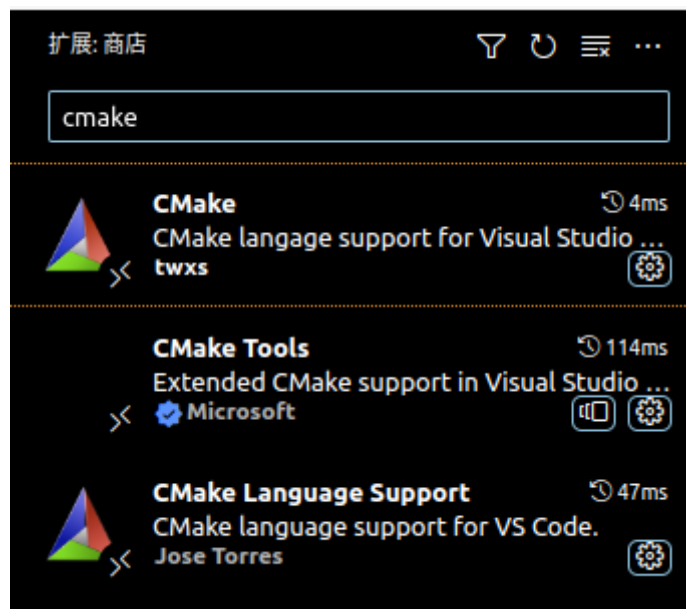
然后在左上角选择**文件 —— 打开文件夹 ——**

/home/kuiperdatawhale —— 确定即可进入课程主界面。

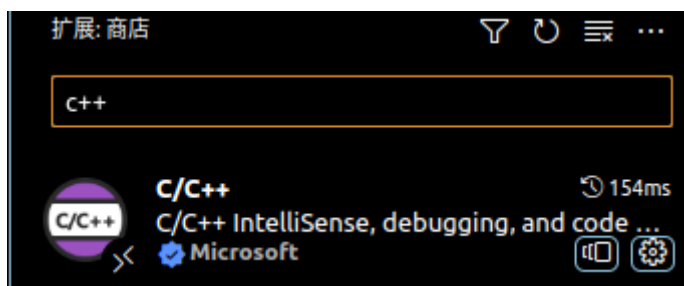
3. 进入 Docker 环境后的 VSCode 拥有独立的插件环境，所以我们需要重新安装有关插件：

你需要在此时的主界面找到扩展，且根据安装 Docker 插件的步骤自行搜索完成以下几款插件的安装：

CMake 相关插件的安装（共三种）



C++ 相关插件的安装（共一种）

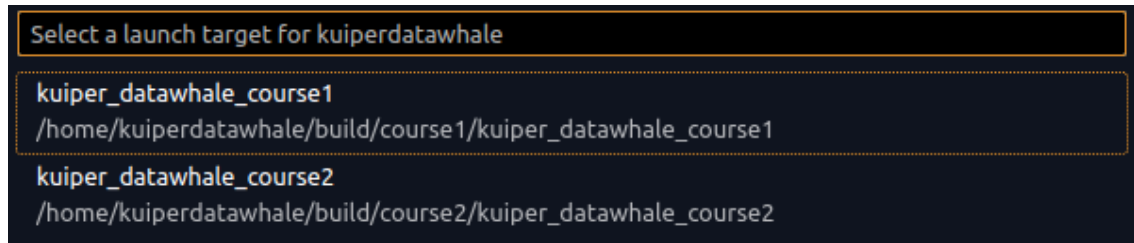


4. 在安装成功后，我们就可以开始编译调试了，你将会在下方看到如下界面：



首先，你需要点击工具图表，选择 `GCC 9.4.0 x86_64-linux-gnu` 版本的编译工具，确保编译工具一致。

接下来让我们尝试编译运行主程序，只需要在 ▶ 键右侧选择对应的编译目标（比如图中我选择了 `kuiper_datawhale_course1`），再点击 ▶ 按钮 即可开始编译运行。



如果配置成功，程序会出现如下的运行结果，出现 `Failed` 这是因为本节课作业需要完成一定的代码，若作业都正确完成，`Failed` 标签就会在运行结果中消失。

```
[=====] Running 7 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 7 tests from test_arma
[ RUN      ] test_arma.Axby
/tmp/tmp.dvfnZylz2q/course1/test/axby.cpp:32: Failure
Expected equality of these values:
  approx_equal(y, answer, "absdiff", 1e-5f)
    Which is: false
  true
[  FAILED  ] test_arma.Axby (0 ms)
[ RUN      ] test_arma.e_power_minus
```

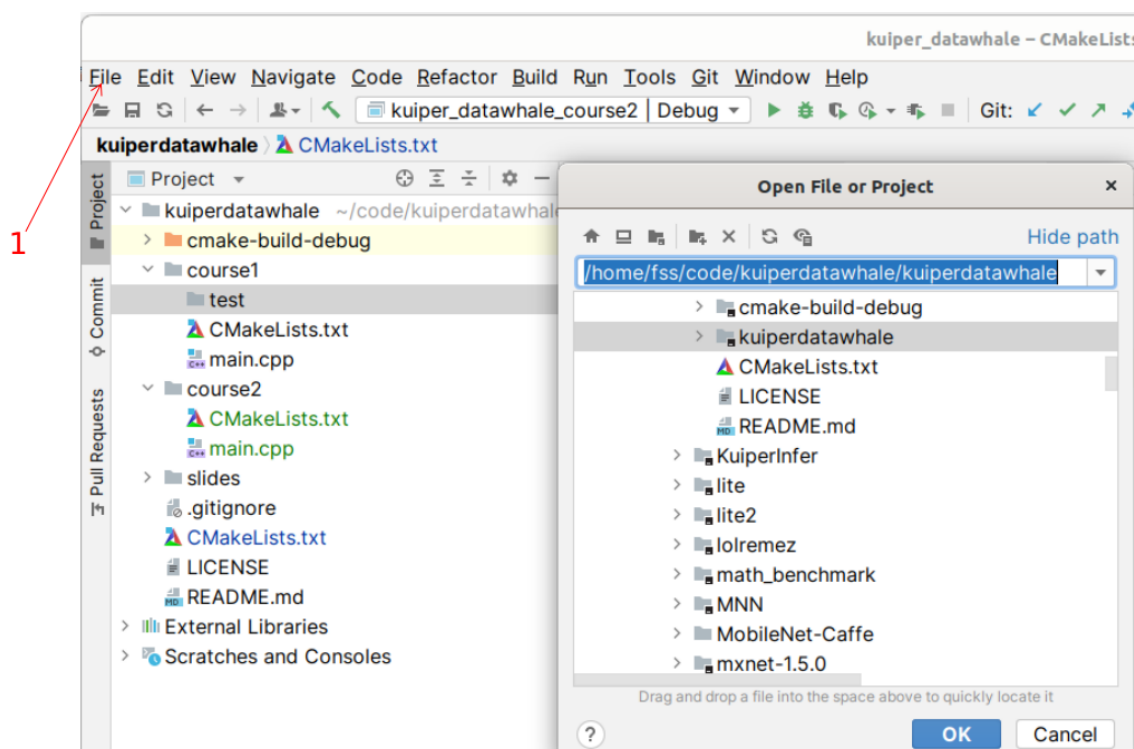
除了编译程序，调试程序也是很重要的一环，在这里你可以试试看点击 ▶ 键旁边的小虫子键，然后观察会发生什么现象，也可以自己尝试打断点看看程序的执行流是否会停留在预期位置，甚至在调试控制台输入变量看看会出现什么结果，这里的探索就交给你。

另外，如果你想了解更多有关 VSCode 的使用方法以及调试说明，请参考[资料](#)。

用 Clion 连接容器

容器在运行的时候已经打开了一个 `ssh` 服务，所以我们只需要用 `Clion` 进行连接即可，在连接成功之后项目的代码就可以直接在容器中进行编译运行。

1. 在 `Clion` 的左上角点击 `File-->Open`，打开项目所在的文件夹，也就是刚才的 `~/code/kuiperdatawhale/kuiperdatawhale` 文件夹。



2. 在容器内输入 `ifconfig`，注意是容器内，不是你宿主机上！我们可以看到 `inet` 对应的值为 `172.17.0.4`。

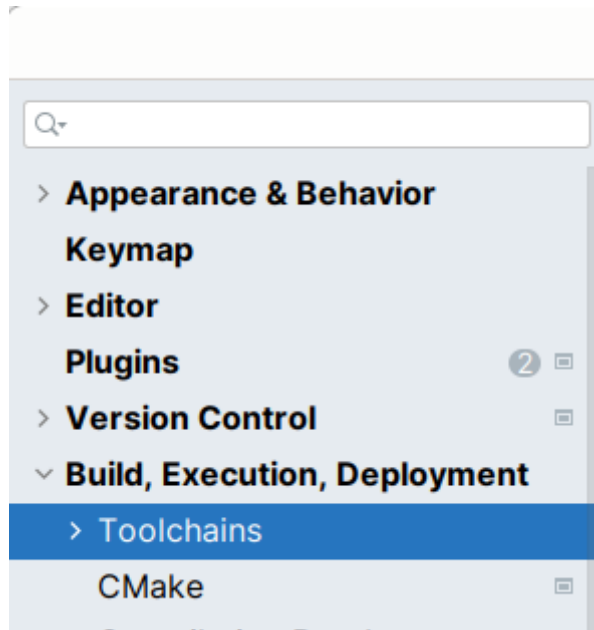
注意这里容器的 `ip` 地址和你实际运行情况有关，需要自行输入

`ifconfig` 命令查看，不一定是 `172.17.0.4`

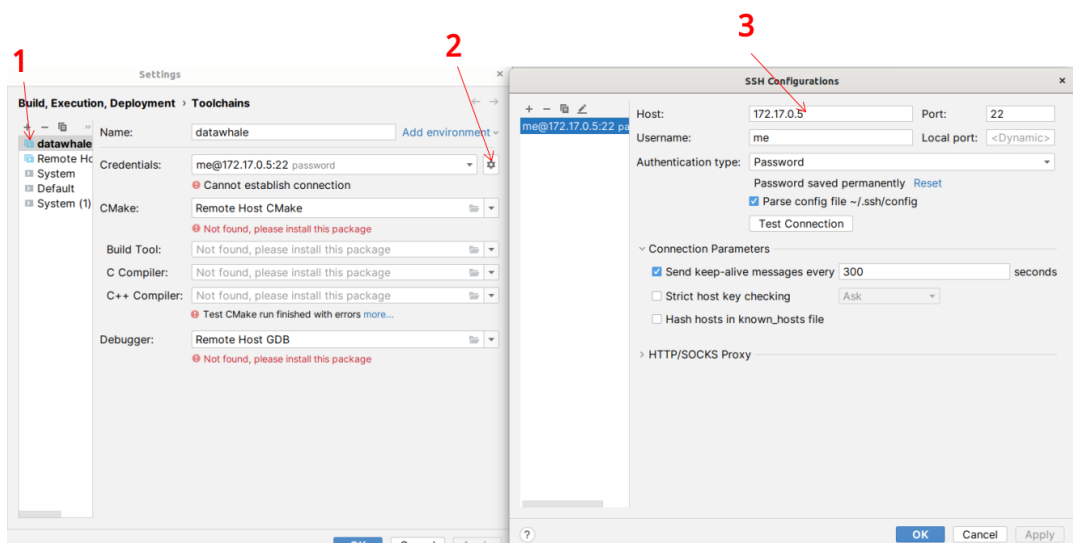
```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.4 netmask 255.255.0.0 broadcast
172.17.255.255
    ether 02:42:ac:11:00:04 txqueuelen 0 (Ethernet)
    RX packets 55 bytes 8479 (8.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0
collisions 0
```

3. 在确保容器已经在运行后，再在Clion的Toolchain设置为容器内的编程环境

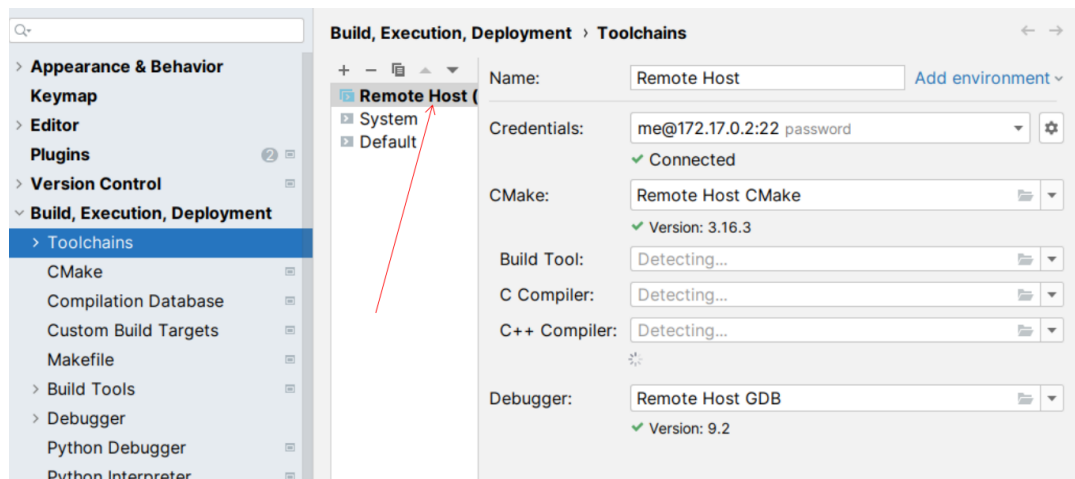
- 点击Clion软件左上角的File->Settings, 并在左侧对应位置找到Toolchains



- 请按图示的顺序进行操作，在3处将host修改为容器中输入ifconfig命令得到的ip地址，并将username设置为me, 登录方式为密码登录，密码为1.
- 所有信息填好之后点击Test Connection. 如果连接成功，则会显示Connected Successfully, 如果此处连接失败，可以尝试替换映射端口或自行查阅资料解决。

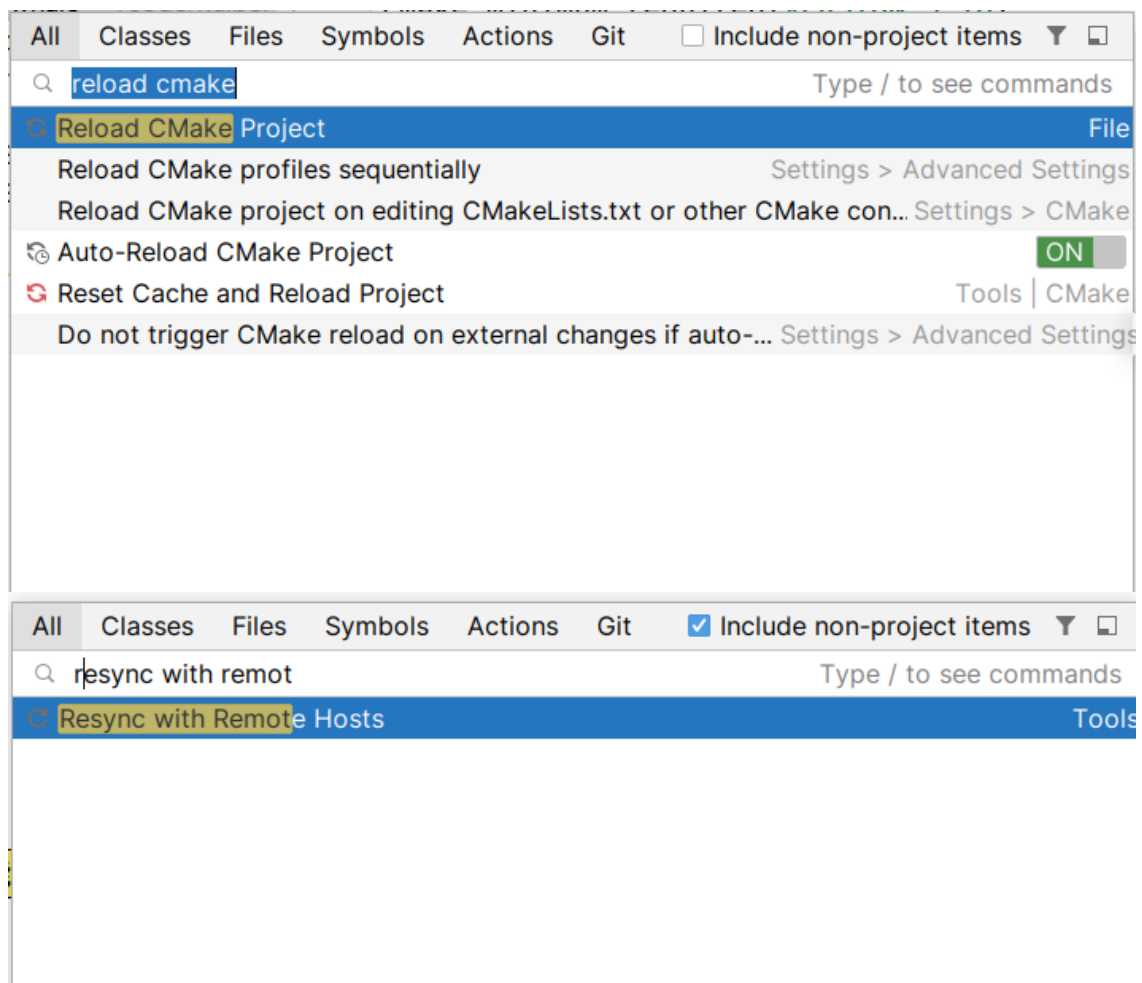


- 随后将datawhale一栏拖动到最上方（鼠标左键拖动），表示软件默认使用容器内的编译工具链。

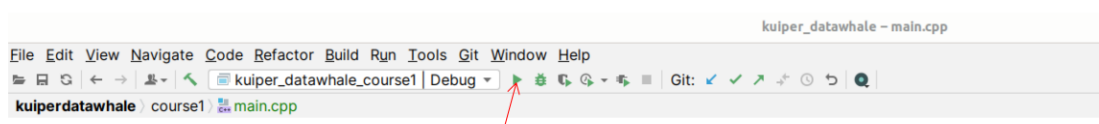


4. 现在编程环境已经配置到了 Docker 容器中，随后点击菜单栏的 `Helps->Find actions`，再输入 `Reload Cmake Project` 对项目进行重新加载，包括如果你之后修改了 `Cmake` 文件，最好都用这个命令重新加载下项目。

- 如果发现头文件报错或者不能智能提示，可以在 `Find actions` 中输入 `resync with remote hosts` 来同步远程的头文件。



5. 随后就可以选择 `course1` 并点击三角形按钮对项目代码进行执行。



如果配置成功，程序会出现如下的运行结果，出现 `Failed` 这是因为本节课作业需要完成一定的代码，若作业都正确完成，`Failed` 标签就会在运行结果中消失。

```
[=====] Running 7 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 7 tests from test_arma
[ RUN      ] test_arma.Axby
/tmp/tmp.dvfnZylz2q/course1/test/axby.cpp:32: Failure
Expected equality of these values:
  approx_equal(y, answer, "absdiff", 1e-5f)
    Which is: false
true
[  FAILED  ] test_arma.Axby (0 ms)
[ RUN      ] test_arma.e_power_minus
```

如果对以上配置远程开发的过程还有疑问，可以参考[资料](#)。

如何写单元测试

搭建好了环境，我们来写一些单元测试看看运行情况。同时，在这个过程中，我们也试着去写项目的单元测试。单元测试框架使用了 `Google` 的 `Google Test`，该框架已经在 `Docker` 镜像中提供。我们以矩阵库 `armadillo` 的计算接口作为我们的测试目标，在测试的过程中也去熟悉这个矩阵库的基本用法，以下是 `armadillo` 矩阵库的文档地址，以供我们在学习的过程中查阅：[armadillo documentation](#)。

在 `test/test1.cpp` 中有以下几个函数：

1. `test_add` 函数用来测试 `armadillo` 的矩阵加法接口

```
TEST(test_arma, add) {
    using namespace arma;
    fmat in_matrix1 = "1,2,3;"
                      "4,5,6;"
                      "7,8,9";

    fmat in_matrix2 = "1,2,3;"
                      "4,5,6;"
```

```

        "7,8,9";

    const fmat &out_matrix1 = "2,4,6;"
        "8,10,12;"
        "14,16,18";

    const fmat &out_matrix2 = in_matrix1 + in_matrix2;
    ASSERT_EQ(approx_equal(out_matrix1, out_matrix2,
        "absdiff", 1e-5), true);
}

```

2. test_sub 函数用来测试 armadillo 的矩阵减法接口

```

TEST(test_arma, sub) {
    using namespace arma;
    fmat in_matrix1 = "1,2,3;"
        "4,5,6;"
        "7,8,9";

    fmat in_matrix2 = "1,2,3;"
        "4,5,6;"
        "7,8,9";

    const fmat &out_matrix1 = "0,0,0;"
        "0,0,0;"
        "0,0,0;";

    const fmat &out_matrix2 = in_matrix1 - in_matrix2;
    ASSERT_EQ(approx_equal(out_matrix1, out_matrix2,
        "absdiff", 1e-5), true);
}

```

3. test_matmul 用来测试 armadillo 的矩阵乘法接口

```

TEST(test_arma, matmul) {
    using namespace arma;
    fmat in_matrix1 = "1,2,3;"
        "4,5,6;"
        "7,8,9";

```

```

    fmat in_matrix2 = "1,2,3;"
                      "4,5,6;"
                      "7,8,9";

    const fmat &out_matrix1 = "30,36,42;"
                              "66,81,96;"
                              "102,126,150;";

    const fmat &out_matrix2 = in_matrix1 * in_matrix2;
    ASSERT_EQ(approx_equal(out_matrix1, out_matrix2,
        "absdiff", 1e-5), true);
}

```

4. `test_pointwise` 用来测试 `armadillo` 的矩阵点积接口

```

TEST(test_arma, pointwise) {
    using namespace arma;
    fmat in_matrix1 = "1,2,3;"
                      "4,5,6;"
                      "7,8,9";

    fmat in_matrix2 = "1,2,3;"
                      "4,5,6;"
                      "7,8,9";

    const fmat &out_matrix1 = "1,4,9;"
                              "16,25,36;"
                              "49,64,81;";

    const fmat &out_matrix2 = in_matrix1 % in_matrix2;
    ASSERT_EQ(approx_equal(out_matrix1, out_matrix2,
        "absdiff", 1e-5), true);
}

```

我们将所有的测试代码都放在了 `test` 文件夹下，`Cmake` 构建系统会自动读取该文件夹下的单元测试文件，如果新添加了 `.cpp` 文件或者新的单元测试函数，需要使用 `Reload Cmake Project` 重新进行载入。

本节作业

1. 在 `axby.cpp` 中编写 $Y = w \times x + b$ 的代码，其中 w 和 b 是一个向量，并通过其中的单元测试，请自行查阅[资料](#)。
2. 在 `axby.cpp` 中编写 $Y = e^{-x}$ 的代码，并通过其中的单元测试，请自行查阅[资料](#)。
3. 在 `axby.cpp` 中编写 $Y = a \times x + y$ 的代码，其中 a 和 y 是一个标量，同样需要通过单元测试，请自行查阅[资料](#)。