

对象存储、Docker、CI

无12 王诗凯

目录

对象存储、Docker、CI

目录

对象存储

需要了解的基本概念

为什么要使用对象存储

存储桶的权限类别

如何使用对象存储

1. 注册账号

2. 创建存储桶

3. 查看存储桶列表

4. 访问存储桶

更多内容

Docker

什么是Docker

安装Docker

Windows

Mac

Docker的基本组成

镜像(image)

容器(container)

仓库(repository)

Docker的基本命令

进入容器内部

容器与主机

数据卷 (Volume)

创建数据卷

挂载主机目录作为数据卷

Docker commit

Dockerfile

Docker执行Dockerfile的过程

Dockerfile的命令

如何运行Dockerfile

CI

什么是CI

Github Actions

如何使用Github Action

作业

对象存储

需要了解的基本概念

- 对象存储（Cloud Object Storage, COS）：是腾讯云提供的一种存储海量文件的分布式存储服务。阿里云也提供对象存储的服务（OSS, Object Storage Service）,我们以腾讯云为例来介绍对象存储。
- 存储桶（Bucket）：是对象的载体，可理解为存放对象的“容器”。一个存储桶可容纳无数个对象。**无文件夹和目录的概念。**
- 对象（Object）：是对象存储的基本单元，可以理解为任何格式类型的数据。
- 地域（Region）：是腾讯云托管机房的分布地区，COS 的数据存放在这些地域的存储桶中。

为什么要使用对象存储

- 可以非常方便地进行对象的上传、查询、下载、复制和删除操作，并对文件进行批处理。
- 有很好的权限控制，数据安全性很强。
- 在THUAI6中，选手代码以及选手包就是利用对象存储进行管理。

存储桶的权限类别

- 公共权限
 - 私有读写：只有该存储桶的创建者及有授权的账号才对该存储桶中的对象有读写权限，其他任何人对该存储桶中的对象都没有读写权限。存储桶访问权限默认为私有读写。
 - 公有读私有写：任何人（包括匿名访问者）都对该存储桶中的对象有读权限，但只有存储桶创建者及有授权的账号才对该存储桶中的对象有写权限。
 - 公有读写：任何人（包括匿名访问者）都对该存储桶中的对象有读权限和写权限，不推荐使用。

- 用户权限

主账号默认拥有存储桶的所有权限（即完全控制）。另外 COS 支持添加子账号有**数据读取、数据写入、权限读取、权限写入，甚至完全控制**的最高权限

如何使用对象存储

1. 注册账号

- 访问网站 <https://cloud.tencent.com/>，右上角有“免费注册”选项，如果有账号可以直接登录。

2. 创建存储桶

- 登陆成功后访问“产品-存储-对象存储-立即使用”，进入存储桶控制台。界面如下：



- 点击“创建存储桶”，然后完善相关信息，点击下一步。（默认权限为私有读写，这里修改为公有读私有写）

1 基本信息

2 高级可选配置

3 确认配置

所属地域

中国

北京

存储桶与相同地域的其他腾讯云服务内网互通；**创建后地域无法修改**，请您谨慎选择。

名称 * ?

bucket123

-1314234950 ✓

还能输入 12 个字符，支持小写字母、数字和 - ；**创建后名称无法修改。**

访问权限

☐ 私有读写

☒ 公有读私有写

☐ 公有读写

可对object进行匿名读操作，写操作需要进行身份验证。

注意：
公有读权限可以通过匿名身份直接读取您存储桶中的数据，存在一定的流量盗刷风险。为确保您的业务安全，不推荐此配置，建议您选择私有读写。
此外，建议您把 COS 接入 CDN 并开启 CDN 的回源鉴权。您还可以使用 [防盗链功能](#)，防止流量被盗刷。

默认告警

☒

当检测到1分钟内外网下行流量大于5000MB时，会进行告警通知。

内容安全

☐

为保障您的数据在公有读权限下的安全合规问题，建议您开启内容安全，开启后将默认对您上传的所有图片、视频、音频、文本数据进行内容审核，内容审核将产生一定的费用，请查看 [计费策略](#)

请求域名

bucket123-1314234950.cos.ap-beijing.myqcloud.com

创建完成后，您可以使用该域名对存储桶进行访问

取消

下一步

- 后续的高级可选配置可以先直接默认下一步，最后确认创建成功。

3. 查看存储桶列表

- 左侧有存储桶列表选项可以进行查看，页面如下：



- 点击对应的存储桶名称即可进入对应的存储桶查看。我们进入刚刚创建的存储桶，发现现在还没有任何数据。

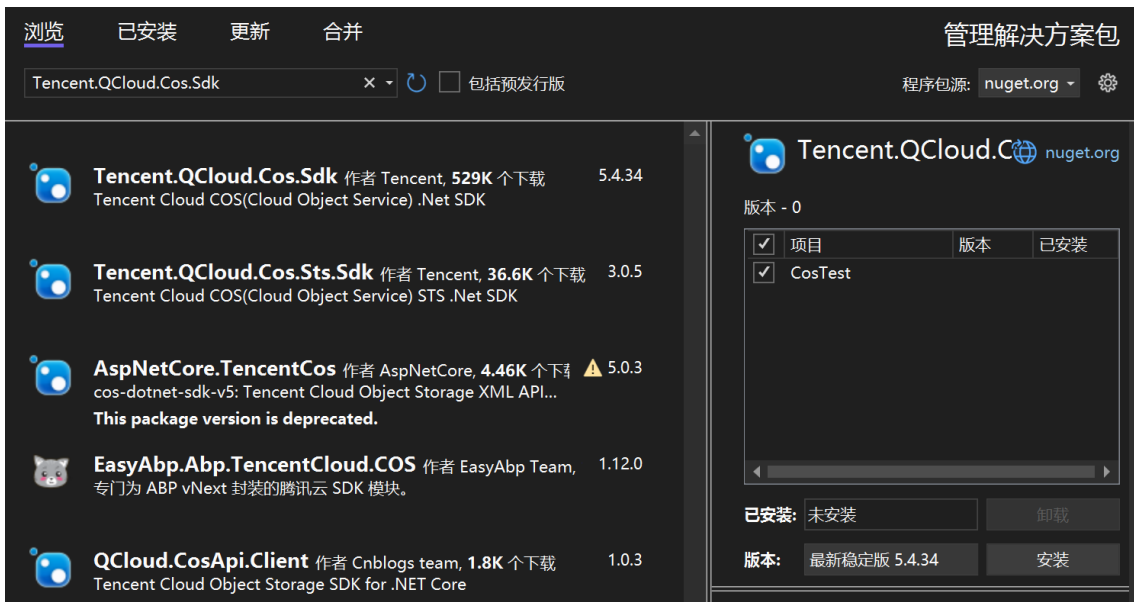


4. 访问存储桶

- 腾讯云提供了多种访问存储桶的方式，分别是：**控制台**、**COSBrowser工具**、**COSCMD工具**、**API方式**、**SDK方式**，其中**控制台**方式就是在腾讯云网页中进行操作。创建存储桶的操作我们已经说过，上传文件、创建文件夹和清空存储桶的操作在上一张图中也可以清晰的看到，同学们可以自行探索。这里主要讲解用SDK方式进行访问的方式。
- 官网提供了很多SDK，我们主要展示C#的SDK。

Android SDK	Android SDK 快速入门
C SDK	C SDK 快速入门
C++ SDK	C++ SDK 快速入门
.NET(C#) SDK	.NET(C#) 快速入门
Flutter SDK	Flutter SDK 快速入门
Go SDK	Go SDK 快速入门
iOS SDK	iOS SDK 快速入门
Java SDK	Java SDK 快速入门
JavaScript SDK	JavaScript SDK 快速入门
Node.js SDK	Node.js SDK 快速入门
PHP SDK	PHP SDK 快速入门
Python SDK	Python SDK 快速入门
React Native SDK	React Native SDK 快速入门
小程序 SDK	小程序 SDK 快速入门

- 首先在Visual Studio中建立C#的控制台工程，在Nuget包管理器中搜索“Tencent.QCloud.Cos.Sdk”，进行安装。



- 添加如下命名空间：

```
1 using COSXML;  
2 using COSXML.Auth;  
3 using COSXML.Model.Object;  
4 using COSXML.Model.Bucket;  
5 using COSXML.CosException;  
6 using COSXML.Model.Service;  
7 using COSXML.Model.Tag;
```

- 初始化服务设置

```
1 //初始化 CosXmlConfig  
2 string appid = "1250000000"; //设置腾讯云账户的账户标识 APPID  
3 string region = "COS_REGION"; //设置一个默认的存储桶地域  
4 CosXmlConfig config = new CosXmlConfig.Builder()  
5     .IsHttps(true) //设置默认 HTTPS 请求  
6     .SetRegion(region) //设置一个默认的存储桶地域  
7     .SetDebugLog(true) //显示日志  
8     .Build(); //创建 CosXmlConfig 对象  
9
```

- APPID是什么？存储桶地域是什么？

存储桶命名规范

存储桶的命名由存储桶名称（BucketName）和 APPID 两部分组成，两者以中划线“-”相连。例如 `examplebucket-1250000000`，其中 `examplebucket` 为用户自定义字符串，`1250000000` 为系统生成数字串（APPID）。在 API、SDK 的示例中，存储桶的命名格式为 `<BucketName-APPID>`。

所属地域已经在存储桶列表中标明，比如北京，就是 `ap-beijing`

- 提供访问凭证

SDK 中提供了3种方式：**持续更新的临时密钥**、**不变的临时密钥**、**永久密钥**。

官方推荐使用持续更新的临时密钥进行访问，但由于此方法代码过于冗长，我们选择较为简单的**永久密钥**进行演示。

- 永久密钥：

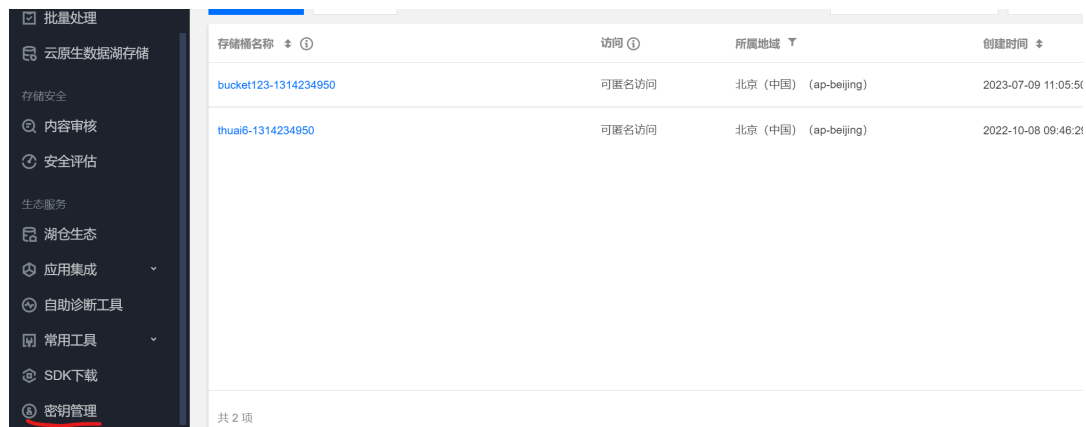
```

1  string secretId = Environment.GetEnvironmentVariable("SECRET_ID");
   //用户的 secretId
2  string secretKey = Environment.GetEnvironmentVariable("SECRET_KEY");
   //用户的 SecretKey,
3  long durationSecond = 600; //每次请求签名有效时长, 单位为秒
4  QCloudCredentialProvider cosCredentialProvider = new
   DefaultQCloudCredentialProvider(
5      secretId, secretKey, durationSecond);
6  //这里Environment.GetEnvironmentVariable()是根据环境变量名获得环境变量值的
   函数
7  //此时在系统的环境变量中需要有SECRET_ID和SECRET_KEY这两个变量, 其值分别对应用
   户的SecretId和SecretKey
8  //这样写是为了保证密钥不会在github等代码托管平台上泄露(通过密钥可直接登录账号)
9  //如果嫌麻烦可以直接像下面这样:
10 string secretId = "***"; //用户的 secretId
11 string secretKey = "***"; //用户的 SecretKey,

```

SecretId和SecretKey是什么?

进入存储桶列表, 左下方进入“密钥管理”选项。



按照提示前往访问密钥进行查看。但这种方法查看得到的密钥为主账号的密钥, 直接在代码中使用不太安全, 按照官方建议, 可以先创建子账号, 再用子账号的密钥进行访问。可以自定义子账号拥有的权限, 使得其对存储桶不具有完全控制权, 保障数据安全。左侧进入用户列表选项, 新建用户创建子账号。



创建结束后, 点击用户名称, 即可在API密钥中查看SecretId和Secretkey。



- 初始化 CosXmlServer

使用 `CosXmlConfig` 与 `QCloudCredentialProvider` 初始化 `CosXmlServer` 服务类。服务类建议在程序中作为单例使用。

```
1 CosXml cosXml = new CosXmlServer(config, cosCredentialProvider);
```

- 访问COS服务

官网给出了如下几个操作，以下给出的代码只是官方给出的框架，具体要根据实际情况进行修改。

- 创建存储桶

```
1 try
2 {
3     string bucket = "examplebucket-1250000000"; //格式: BucketName-APPID
4     PutBucketRequest request = new PutBucketRequest(bucket);
5     //执行请求
6     PutBucketResult result = cosXml.PutBucket(request);
7     //请求成功
8     Console.WriteLine(result.GetResultInfo());
9 }
10 catch (COSXML.CosException.CosClientException clientEx)
11 {
12     //请求失败
13     Console.WriteLine("CosClientException: " + clientEx);
14 }
15 catch (COSXML.CosException.CosServerException serverEx)
16 {
17     //请求失败
18     Console.WriteLine("CosServerException: " + serverEx.GetInfo());
19 }
20
```

- 查询存储桶列表

```
1 try
2 {
3     GetServiceRequest request = new GetServiceRequest();
4     //执行请求
5     GetServiceResult result = cosXml.GetService(request);
6     //得到所有的 buckets
```

```

7      List<ListAllMyBuckets.Bucket> allBuckets =
      result.listAllMyBuckets.buckets;
8  }
9  catch (COSXML.CosException.CosClientException clientEx)
10 {
11     //请求失败
12     Console.WriteLine("CosClientException: " + clientEx);
13 }
14 catch (COSXML.CosException.CosServerException serverEx)
15 {
16     //请求失败
17     Console.WriteLine("CosServerException: " + serverEx.GetInfo());
18 }
19

```

- 这里给出一个实例：

将以下代码添加到第七行之后：

```

1  foreach(var bucket in allBuckets)
2  {
3      Console.WriteLine(bucket.name);      //打印存储桶列表的名称
4  }

```

效果如下：

```

1. 获取存储桶列表
2. 上传对象
3. 查询对象列表
4. 下载对象
5. 删除对象
6. 退出
1
bucket123-1314234950
thuai6-1314234950

```

- 上传对象

```

1  // 初始化 TransferConfig
2  TransferConfig transferConfig = new TransferConfig();
3
4
5  // 初始化 TransferManager
6  TransferManager transferManager = new TransferManager(cosXml,
    transferConfig);
7
8
9  String bucket = "examplebucket-1250000000"; //存储桶，格式：
    BucketName-APPID
10 String cosPath = "exampleobject"; //对象在存储桶中的位置标识符，即称对象键
11 String srcPath = @"temp-source-file"; //本地文件绝对路径
12
13
14 // 上传对象
15 COSXMLUploadTask uploadTask = new COSXMLUploadTask(bucket, cosPath);
16 uploadTask.SetSrcPath(srcPath);
17

```



```

18
19 uploadTask.progressCallback = delegate (long completed, long total)
20 {
21     Console.WriteLine(String.Format("progress = {0:##.##}%",
22                                     completed * 100.0 / total));
23 };
24
25 try {
26     COSXML.Transfer.COSXMLUploadTask.UploadTaskResult result = await
27         transferManager.UploadAsync(uploadTask);
28     Console.WriteLine(result.GetResultInfo());
29     string eTag = result.eTag;
30 } catch (Exception e) {
31     Console.WriteLine("CosException: " + e);
32 }
33

```

■ 下面是上传的一个实例：

```

1. 获取存储桶列表
2. 上传对象
3. 查询对象列表
4. 下载对象
5. 删除对象
6. 退出
2

```

文件本地绝对路径：

E:\player1.cpp

COS路径：

player1.cpp

2

文件本地绝对路径：

E:\player2.cpp

COS路径：

folder/player2.cpp

上传了两个 .cpp 文件，进入控制台效果如下：

[bucket123-1314234950](#) /

<div> <div>上传文件</div> <div>创建文件夹</div> <div>文件碎片</div> <div>清空存储桶</div> <div>更多操作</div> </div>			
<div> <div>前缀搜索</div> <div>只支持搜索当前虚拟目录下的对象</div> <div>刷新</div> <div>共 2 个文件</div> <div>每页</div> </div>			
<input type="checkbox"/> 文件名	大小	存储类型	修改时间
<input type="checkbox"/> folder/	-	-	-
<input type="checkbox"/> player1.cpp	83.00B	标准存储	2023-07-09 16:29:48

注意：上传文件时的cosPath必须带上文件名，且“文件夹”的分隔符必须是/，不能是\\或者\。无文件夹和目录的概念，用对象键来标记每一个文件，只是用文件夹这种人们熟悉的方式来显示。因此在上传时无法直接上传文件夹（srcPath不能是某个文件夹的绝对路径）。但是我们希望实现这样选择一个文件夹，然后将文件夹全部上传的功能。思路是通过递归或者循环的方式遍历一个文件夹中的所有文件，得到每一个文件的绝对路径然后上传。

- 查询对象列表

```
1  try
2  {
3      string bucket = "examplebucket-1250000000"; //格式: BucketName-
APPID
4      GetBucketRequest request = new GetBucketRequest(bucket);
5      //执行请求
6      GetBucketResult result = cosXml.GetBucket(request);
7      //bucket的相关信息
8      ListBucket info = result.listBucket;
9      if (info.isTruncated) {
10         // 数据被截断, 记录下数据下标
11         this.nextMarker = info.nextMarker;
12     }
13 }
14 catch (COSXML.CosException.CosClientException clientEx)
15 {
16     //请求失败
17     Console.WriteLine("CosClientException: " + clientEx);
18 }
19 catch (COSXML.CosException.CosServerException serverEx)
20 {
21     //请求失败
22     Console.WriteLine("CosServerException: " + serverEx.GetInfo());
23 }
24
```

- 下面给出一个实例:

在上面代码的12行后添加如下代码:

```
1  var content = info.contentsList;
2  foreach ( var item in content )
3  {
4      Console.WriteLine(item.key);           //打印对象键
5  }
```

结果如下:

```
3
folder/player2.cpp
player1.cpp
```

- 下载对象

```
1  // 初始化 TransferConfig
2  TransferConfig transferConfig = new TransferConfig();
3
4
5  // 初始化 TransferManager
6  TransferManager transferManager = new TransferManager(cosXml,
transferConfig);
7
8
```

```

9 String bucket = "examplebucket-1250000000"; //存储桶，格式：
  BucketName-APPID
10 String cosPath = "exampleobject"; //对象在存储桶中的位置标识符，即称对象键
11 string localDir = System.IO.Path.GetTempPath();//本地文件夹(这里获取的是
  计算机中保存临时文件的路径)
12 string localFileName = "my-local-temp-file"; //指定本地保存的文件名
13
14
15 // 下载对象
16 COSXMLDownloadTask downloadTask = new COSXMLDownloadTask(bucket,
  cosPath,
17     localDir, localFileName);
18
19
20 downloadTask.progressCallback = delegate (long completed, long
  total)
21 {
22     Console.WriteLine(String.Format("progress = {0:##.##}%",
  completed * 100.0 / total));
23 };
24
25
26 try {
27     COSXML.Transfer.COSXMLDownloadTask.DownloadTaskResult result =
  await
28     transferManager.DownloadAsync(downloadTask);
29     Console.WriteLine(result.GetResultInfo());
30     string eTag = result.eTag;
31 } catch (Exception e) {
32     Console.WriteLine("CosException: " + e);
33 }
34

```

■ 这里给出一个实例：

```

1. 获取存储桶列表
2. 上传对象
3. 查询对象列表
4. 下载对象
5. 删除对象
6. 退出
4
下载的对象键：
folder/player2.cpp
本地路径：
E:\test\folder
要保存的文件名：
test.cpp

```

运行后在指定目录下可以找到对应的文件。

此电脑 > 本地磁盘 (E:) > test > folder				在 folder
名称	修改日期	类型	大小	
test.cpp	2023/7/9 19:17	C++ Source	1 KB	

注意：COS不能直接下载整个“文件夹”，如果尝试把对象键直接写成文件夹的名称，即使不抛出异常，下载得到的文件内容也不是你想要的内容，而是会出现报错信息。因此我们只能一个一个文件进行下载，这有时会导致下载速度非常慢，可以使用多线程下载或者将要下载的文件打包成压缩包上传到存储桶，然后下载压缩包再进行解压的方式来解决。感兴趣的同学可以前往github阅读THUAI6的代码。对象存储中的文件名和下载之后的文件名可以不一样。

- 删除对象

```
1  try
2  {
3      string bucket = "examplebucket-1250000000"; //存储桶，格式：
      BucketName-APPID
4      string key = "exampleobject"; //对象键
5      DeleteObjectRequest request = new DeleteObjectRequest(bucket,
      key);
6      //执行请求
7      DeleteObjectResult result = cosXml.DeleteObject(request);
8      //请求成功
9      Console.WriteLine(result.GetResultInfo());
10 }
11 catch (COSXML.CosException.CosClientException clientEx)
12 {
13     //请求失败
14     Console.WriteLine("CosClientException: " + clientEx);
15 }
16 catch (COSXML.CosException.CosServerException serverEx)
17 {
18     //请求失败
19     Console.WriteLine("CosServerException: " + serverEx.GetInfo());
20 }
21
```

- 给出一个实例：

```
1. 获取存储桶列表
2. 上传对象
3. 查询对象列表
4. 下载对象
5. 删除对象
6. 退出
5
删除的对象键：
folder/player2.cpp
```

之后进入存储桶页面：

bucket123-1314234950 /

上传文件

创建文件夹

文件碎片

清空存储桶

更多操作

前置搜索

只支持搜索当前虚拟目录下的对象

刷新

共 1 个文件

每页

<input type="checkbox"/>	文件名	大小	存储类型	修改时间
<input type="checkbox"/>	player1.cpp	83.00B	标准存储	2023-07-09 16:29:48

发现folder“文件夹”已经被删除。

更多内容

- 请访问 <https://cloud.tencent.com/document/product/436> 进行探索。

Docker

什么是Docker

- docker是基于Go语言的云开源项目，通过将源代码，环境，第三方依赖等打包成一个镜像，然后就可以在任何地方依据镜像生成容器，之后就可以运行了。这样可以有效避免仅提供源码但环境不同，配置不同及第三方依赖的版本不同导致的兼容问题。这样可以轻松实现跨平台，跨服务器的运行，简化了从开发、调试到生成的迁移问题。

安装Docker

Windows

- 一旦安装好Docker Desktop并运行，我们就可以在WSL2中使用Docker并运行Linux容器。但是Windows家庭版无法运行Windows容器，只有Windows企业版或者专业版才可以运行。
- 在官网下载Docker Desktop Installer.exe，打开后根据提示进行安装即可。可参考 <https://docs.docker.com/desktop/install/windows-install/>
- **注意：**如果某天在WSL中使用Docker时发现他告诉你找不到这个命令，然而你已经安装了Docker Desktop，这个时候要先检查Docker Desktop是否处于运行状态。
- 安装成功后，在cmd中输入 `docker` 会出现如下输出

```
C:\Users\lenovo>docker

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default
                        "C:\Users\lenovo\.docker")
  -c, --context string  Name of the context to use to connect to the
                        daemon (overrides DOCKER_HOST env var and
                        default context set with "docker context use")
  -D, --debug           Enable debug mode
  -H, --host list       Daemon socket(s) to connect to
  -l, --log-level string Set the logging level
                        ("debug"|"info"|"warn"|"error"|"fatal")
                        (default "info")
  --tls                Use TLS; implied by --tlsverify
  --tlscacert string    Trust certs signed only by this CA (default
                        "C:\Users\lenovo\.docker\ca.pem")
  --tlscert string      Path to TLS certificate file (default
                        "C:\Users\lenovo\.docker\cert.pem")
  --tlskey string       Path to TLS key file (default
                        "C:\Users\lenovo\.docker\key.pem")
  --tlsverify           Use TLS and verify the remote
  -v, --version         Print version information and quit

Management Commands:
  builder              Manage builds
```

在wsl中输入docker同样会有这样的输出

```
sky@LAPTOP-BHHAQUOA:/mnt/c/Users/lenovo$ docker
Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:
  --config string      Location of client config files (default "/home/sky/.docker")
  -c, --context string  Name of the context to use to connect to the daemon (overrides DOCKER_HOST env var
                        and context use)
  -D, --debug           Enable debug mode
  -H, --host list       Daemon socket(s) to connect to
  -l, --log-level string Set the logging level ("debug"|"info"|"warn"|"error"|"fatal") (default "info")
  --tls                Use TLS; implied by --tlsverify
  --tlscacert string    Trust certs signed only by this CA (default "/home/sky/.docker/ca.pem")
  --tlscert string       Path to TLS certificate file (default "/home/sky/.docker/cert.pem")
  --tlskey string        Path to TLS key file (default "/home/sky/.docker/key.pem")
  --tlsverify           Use TLS and verify the remote
  -v, --version         Print version information and quit

Management Commands:
  builder      Manage builds
  buildx       Docker Buildx (Docker Inc., v0.10.0)
```

即说明安装成功。

Mac

安装过程和Windows类似。注意要区分 Intel 芯片的版本和 Apple 芯片的版本。如果是 Apple 芯片需要先在命令行输入以下内容来安装 Rosetta 2：

```
1 | softwareupdate --install-rosetta
```

虽然 4.3.0 及以后的版本对 Rosetta 2 并没有硬性要求，但是为了良好的体验，官网建议我们进行安装。细节可参考 docker 官网：<https://docs.docker.com/desktop/install/mac-install/>

Docker的基本组成

Docker由三个部分组成，分别是镜像（image），容器（container），仓库（repository）。

镜像(image)

Docker镜像是一个特殊的文件系统，除了提供容器运行时所需的程序、库、资源、配置等文件外，还包含了一些为运行时准备的一些配置参数（比如环境变量）。他相当于是一个只读的模板，有点像C++里的类。

容器(container)

Docker利用容器独立的运行应用，每一个容器都是用镜像创建的运行示例，有点像C++里用类实例化的对象。他是一个虚拟的环境，也算是一个简易版的linux系统（包括root权限，进程空间，用户空间和网络空间）为我们需要运行的程序提供支持，他们可以被启动，开始运行，终止运行，删除等。容器与容器之间相互独立，互不干扰，保证安全。

仓库(repository)

用来集中存放镜像的地方。最大的公开仓库是 docker hub：<https://hub.docker.com/>。

我们可以像仓库上传自己的镜像，也可以在仓库搜索我们需要的镜像，并 pull 下来使用。

Docker的基本命令

```
1 | docker info
```

查看docker的信息，比如资源的占用或者版本。

```
1 | docker --help
```

查看帮助。

```
1 | docker <命令> --help
```

查看具体某条命令的帮助。

```
1 | docker images
```

列出本地的所有镜像及信息。包括仓库名称、标签、镜像ID、创建时间和大小。

参数：

- -q：只显示镜像ID。

```
1 | docker search <镜像名>
```

搜索docker hub中是否有对应的镜像。

参数	说明
NAME	镜像名
DESCRIPTION	镜像说明
STARS	点赞数量
OFFICIAL	是否官方
AUTOMATED	是否是自动构建的

参数：

- --limit N：只列出N个镜像，默认25个（按照stars的顺序从高到低）

```
1 | docker search --limit 5 ubuntu
```

```
1 | docker pull 镜像名[:TAG]
```

拉取指定的镜像。如果不加TAG就默认是 latest。

```
1 | docker system df
```

查看docker中 Images, Container, Local volumes(数据卷)和Build Cache 占用的空间。

```
1 | docker rmi 镜像名1 镜像名2 ...
2 | docker rmi 镜像ID1 镜像ID2 ...
```

删除镜像。

参数：

- -f: 强制删除。如果镜像例化了容器并且容器还存在，那么就无法删除。此时可以用-f来删除。

结合 `docker images -q` 可以全部删除

```
1 docker rmi $(docker images -q)
```

```
1 docker run [Options] IMAGE [CMD]
```

Options	功能
--name	指定容器名字
-d	后台运行容器并返回ID，守护式容器
-i	交互式运行容器（常与-t结合使用）
-t	为容器分配一个伪输入终端（常与-i结合使用）
-P	随机端口映射
-p	指定端口映射

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 `-P` 或 `-p` 参数来指定端口映射，将主机的端口和容器的端口进行映射。

[CMD]是容器创建后，在容器中运行的命令。

```
1 docker run ubuntu:18.04 /bin/echo 'Hello world'
```

在打印一个 `Hello world` 后终止容器。

```
1 docker container start
```

让一个已经终止的容器重新启动。

如果指定了 `-d` 选项，那么执行命令的结果将不会输出在宿主机下，输出结果可用 `docker logs CONTAINER` 查看。

当利用 `docker run` 来创建容器时，Docker 在后台运行的标准操作包括：

- 检查本地是否存在指定的镜像，不存在就从公有仓库下载
- 利用镜像创建并启动一个容器
- 分配一个文件系统，并在只读的镜像层外面挂载一层可读写层
- 从宿主主机配置的网桥接口中桥接一个虚拟接口到容器中去
- 从地址池配置一个 ip 地址给容器
- 执行用户指定的应用程序
- 执行完毕后容器被终止

```
1 docker run -it ubuntu /bin/bash
```

启动一个交互式容器。


```
1 | docker ps [OPTIONS]
```

显示正在运行的容器

参数：

- -a：列出所有当前正在运行的+历史上运行过的容器。
- -l：显示最近创建的容器。
- -n：显示最近n个创建的容器。

```
1 | docker ps -n=3
```

- -q：仅显示容器号。

进入容器后：

```
1 | exit #退出并停止容器
2 | ctrl + p + q #退出但不停止
```

```
1 | docker start [OPTIONS] CONTAINER
```

可以启动已经停止的容器。

```
1 | docker restart [OPTIONS] CONTAINER
```

重新启动容器。这条命令会先停止容器，再启动容器。

```
1 | docker kill [OPTIONS] CONTAINER
2 | docker stop [OPTIONS] CONTAINER
```

停止容器。两者的区别在于docker stop会向Docker容器发送一个停止信息（SIGTERM信号），让容器内的应用程序自行关闭并做好一些收尾工作。但如果应用程序没有正常关闭，那么会等待一段时间（默认为10秒）后再发送一个强制停止信息（SIGKILL信号）来关闭容器。而docker kill则直接向Docker容器发送一个强制停止信息（SIGKILL信号），忽略容器内应用程序的任何信号和状态。因此，dockerkill会导致容器内的应用程序直接被杀死，可能会丢失一些数据。

```
1 | docker rm [OPTIONS] CONTAINER #只能删除已经停止的
2 | docker rm -f [OPTIONS] CONTAINER
```

删除容器。

进入容器内部

```
1 | docker logs CONTAINER #查看容器日志
2 | docker top CONTAINER #查看容器内运行的进程
3 | docker inspect CONTAINER #查看容器内部
```

以上命令可以查看容器内部的信息

```
1 docker exec -it CONTAINER [CMD] #(在容器中打开一个新的终端，exit不会导致容器停止)
2 docker attach CONTAINER #(直接进入容器的启动命令的终端，exit会导致容器停止)
```

上面这两条命令可以进入容器的交互界面，推荐使用exec。

exec后面的[CMD]是进入容器后运行的命令。如果想要打开命令行交互界面，我们可以写bash。

举个例子：

```
1 docker run -d ubuntu:18.04 /bin/sh -c "while true; do echo hello world; sleep 1; done"
```

这条命令会创建一个一直在后台运行的容器，除非通过某种方式停止他。

容器与主机

```
1 docker cp CONTAINER:容器路径 主机路径
```

复制容器中的文件到主机上

```
1 docker export CONTAINER >文件名.tar
```

导出容器内部的内容作为tar归档文件（相当于导出了整个容器，备份了整个容器）

```
1 cat 文件名.tar | docker import - 镜像用户(可选)/镜像名:镜像版本号
```

数据卷 (Volume)

数据卷 是一个可供一个或多个容器使用的特殊目录，可以在容器之间共享和重用。为什么要有数据卷呢？因为如果我们删除了一个容器，那么这个容器中的数据也就会被删除。宿主机与容器之间的通信用文件导出的方法效率很低，而且无法做到实时更新。而数据卷完全独立于容器的生存周期，默认会一直存在，即使容器被删除。对数据卷的修改也会立马生效。这就提高了数据管理的效率。

创建数据卷

```
1 docker volume create NAME
```

创建一个名为 `NAME` 的数据卷。

```
1 docker volume ls
```

查看所有的数据卷

```
1 docker volume inspect NAME
```

查看指定数据卷的信息

```
1 docker run --mount source=VOLUME_NAME,target=CONTAINER_PATH IMAGE [CMD]
```

将名为 `VOLUME_NAME` 的数据卷挂载到容器的 `CONTAINER_PATH` 目录，实质上就是指定 `mount` 的参数。

如果以交互式方式进入容器，就会发现挂载之后的数据卷在容器中以一个文件夹的形式存在。这时如果一个容器向其中存入文件，另一个挂载同样数据卷的文件也会看到同样的更改。

挂载主机目录作为数据卷

同样需要指定 `mount` 的参数。

```
1 docker run --mount type=bind,source=SRC_PATH,target=CONTAINER_PATH,readonly  
  IMAGE [CMD]
```

其中，`SRC_PATH` 为主机的目录，`CONTAINER_PATH` 为容器内的路径，这里指定了 `readonly`，即挂载的目录在容器中是只读的。如果想指定读写，就直接把 `readonly` 去掉。

绑定后，在容器中对 `CONTAINER_PATH` 做的操作就等于对 `SRC_PATH` 做的操作。现在容器可以更方便的使用主机的数据了。

Docker commit

如何构建一个镜像？可以将自己写的容器进行commit，变成镜像。

我们以 `ubuntu:18.04` 为基础，构建一个拥有vim的新镜像。

```
1 docker run -it ubuntu:18.04 /bin/bash  
2 apt-get update  
3 apt-get -y install vim
```

现在已经成功安装vim，按照下面的方法进行提交：

```
1 docker commit -m="add vim" -a="作者" CONTAINER 镜像用户(可选)/镜像名:镜像版本号
```

现在容器成为了一个新的镜像，且该镜像在 `ubuntu` 的基础上有了 `vim` 的功能。我们可以通过继承原有的 `ubuntu` 镜像，为他增加新的镜像层从而实现更高级的功能。有点类似父类与继承。

Dockerfile

Dockerfile是用来构建Docker镜像的文本文件，是由一条条构建镜像所需的指令和参数构成的脚本。我们一次次commit来增强我们的image比较麻烦，Dockerfile可以轻松解决这一问题，在一个file中快速增加镜像的层数，增加镜像的功能。

大致流程：

编写dockerfile-->docker build构建镜像-->docker run实例化容器

Docker执行Dockerfile的过程

- 从基础镜像运行一个容器。
- 执行一条指令并对容器做出修改。
- 执行类似docker commit 的操作，提交生成一个新的镜像层。
- 再基于刚才commit的镜像运行一个新容器。
- 执行下一条指令，直到所有指令都执行完成。

Dockerfile的命令

- FROM: 几乎是所有Dockerfile的第一行, 指定基础镜像。THUAI5编译镜像用的是 `ubuntu:18.04`, THUAI6因为要支持python环境, 因此基础镜像改为了 `python`。
- MAINTAINER: 指定Dockerfile的作者/维护者。(已弃用, 可使用LABEL指令替代)
- LABEL: 添加对象的元数据, 使用键值对的形式。

```
1 LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

- EXPOSE: 声明容器运行时监听的特定网络端口。
- RUN: 在镜像构建过程中执行的命令。

可以是exec形式:

```
1 RUN ["可执行文件", "参数1", "参数2"]
```

也可以是命令行格式:

```
1 RUN apt-get update
```

- WORKDIR: 指定容器的工作目录, 参数为绝对路径。Dockerfile中不能使用cd指令来改变工作目录
- USER: 指定后续指令的用户, 默认为root。
- ENV: 在容器内部设置环境变量。

```
1 ENV RUNNING_PATH /usr/eesast
2 WORKDIR $RUNNING_PATH
```

- COPY: 从宿主机中复制文件或者目录到镜像中。

```
1 COPY src dest
2 COPY ["src", "dest"]
```

其中 `src` 为宿主机的目录, `dest` 为镜像目录

- ADD: 将宿主机下的文件复制到镜像中, 且会自动处理URL和解压TAR压缩包(相对于COPY增加了解压功能)
- ARG: 构建参数。与ENV作用类似, 但是作用域不同。ARG只有在构建镜像时才有用, 构建好的镜像不存在这个变量。

```
1 ARG <参数名>[=<默认值>]
```

在docker build过程中可以使用 `--build-arg <参数名>=<值>` 来覆盖默认值。

- CMD: 类似于RUN指令, 都用来运行程序。但是RUN是在docker build的时候运行, 而CMD是在docker run的时候运行。它的作用是为启动的容器指定默认要运行的程序, 程序运行结束, 容器也就结束。CMD 指令指定的程序可被 docker run 命令行参数中指定要运行的程序所覆盖。如果Dockerfile 中如果存在多个 CMD 指令, 仅最后一个生效。

```
1 CMD <shell 命令>
2 CMD ["<可执行文件或命令>", "<param1>", "<param2>", ...]
3 CMD ["<param1>", "<param2>", ...] # 该写法是为 ENTRYPOINT 指令指定的程序提供默认参数
```

- ENTRYPOINT:

类似于 CMD 指令，但其不会被 docker run 的命令行参数指定的指令所覆盖，而且这些命令行参数会被当作参数送给 ENTRYPOINT 指令指定的程序。但是，如果运行 docker run 时使用了 --entrypoint 选项，将覆盖 ENTRYPOINT 指令指定的程序。如果 Dockerfile 中如果存在多个 ENTRYPOINT 指令，仅最后一个生效。

```
1 ENTRYPOINT ["<EX指令>", "参数1", "参数2"....]
```

可以搭配 CMD 命令使用：一般是变参才会使用 CMD，这里的 CMD 等于是在给 ENTRYPOINT 传参：

假设已通过 Dockerfile 构建了 nginx:test 镜像：

```
1 FROM nginx
2
3 ENTRYPOINT ["nginx", "-c"] # 定参
4 CMD ["/etc/nginx/nginx.conf"] # 变参
```

1、不传参运行

```
1 $ docker run nginx:test
```

容器内会默认运行以下命令，启动主进程。

```
1 nginx -c /etc/nginx/nginx.conf
```

2、传参运行

```
1 $ docker run nginx:test -c /etc/nginx/new.conf
```

容器内会默认运行以下命令，启动主进程(/etc/nginx/new.conf:假设容器内已有此文件)

```
1 nginx -c /etc/nginx/new.conf
```

如何运行Dockerfile

使用 docker build，格式如下：

```
1 docker build [OPTIONS] PATH
```

其中，PATH 是上下文路径。上下文路径，是指 docker 在构建镜像，有时候想要使用到本机的文件（比如复制），docker build 命令得知这个路径后，会将路径下的所有内容打包。因为 docker build 命令构建镜像，其实并非在本地构建，而是在服务端，也就是 Docker 引擎中构建的。那么在这种客户端/服务端的架构中，如何才能让服务端获得本地文件呢？这就引入了上下文的概念。当构建的时候，用户会指定构建镜像上下文的路径，docker build 命令得知这个路径后，会将路径下的所有内容打包，然后

上传给 Docker 引擎。这样 Docker 引擎收到这个上下文包后，展开就会获得构建镜像所需的一切文件。因此在进行COPY和ADD时，宿主机的目录不能超出上下文路径的范围。

一般OPTIONS会用到 `-f`，其参数为Dockerfile的路径。如果不指定，默认为 `PATH/Dockerfile`。

CI

什么是CI

CI就是持续集成（Continuous Integration）的缩写。可以自动化进行代码的构建、测试和部署。举个最简单的例子，我们在进行THUAI6的开发时，每进行一次修改，就要先进行代码风格和编译是否通过的检查，然后将镜像重新上传到docker hub，再将修改后的代码进行打包，形成选手包，上传到存储桶。更复杂的工程可能还涉及到发布一些版本。然而手动进行这些工作过于繁琐，我们就可以通过CI这种自动化工具进行实现。

Github Actions

GitHub Actions 是一种持续集成和持续交付 (CI/CD) 平台，可用于自动执行生成、测试和部署管道。可配置 GitHub Actions 工作流 (workflow)，使其在存储库中发生事件（例如打开拉取请求或创建问题）时触发。工作流包含一个或多个可按顺序或并行运行的Job。每个Job都将在其自己的虚拟机运行器中或在容器中运行，并具有一个或多个step，用于运行定义脚本或运行Action。Action是一个可重用的扩展，可简化工作流。

工作流程在存储库的 `.github/workflows` 目录中定义，存储库可以有多个工作流程。GitHub Actions 使用 YAML 语法来定义工作流程。每个工作流都作为单独的 YAML 文件存储在代码存储库中名为 `.github/workflows` 的目录中。

如何使用Github Action

- 在.github下创建 workflows 文件夹，并在其中创建 .yaml 文件
- 工作流文件的示例如下，他的功能是根据dockerfile创建镜像并上传到docker hub。

```
1  name: "docker"
2  on:
3    push:
4      branches: [main]
5
6  jobs:
7    upload_docker_images:
8      runs-on: ubuntu-latest
9      steps:
10     - uses: actions/checkout@v3
11     - name: Log in to DockerHub
12       run: docker login -u ${ secrets.DOCKER_USERNAME } -p ${ secrets.DOCKER_PASSWORD }
13     - name: Build cpp_compile docker image
14       run: docker build -t ${ secrets.DOCKER_USERNAME }/thuai6_cpp:latest -f ./dependency/Dockerfile/Dockerfile_cpp .
15     - name: Push cpp_compile image to DockerHub
16       run: docker push ${ secrets.DOCKER_USERNAME }/thuai6_cpp:latest
17
18     - name: Build run docker image
19       run: docker build -t ${ secrets.DOCKER_USERNAME }/thuai6_run:latest -f ./dependency/Dockerfile/Dockerfile_run .
```

```

20 - name: Push run image to DockerHub
21     run: docker push ${ secrets.DOCKER_USERNAME }}/thuai6_run:latest
22

```

- `name`: 可选项, 指定workflow的名称, 将出现在github的Action中。
- `on`: 定义workflow的触发条件, 可以写成上面示例那样, 也可以写成


```
1 | on: [push]
```

除此之外, 还有pull_request等触发条件。

- `branches`: 限制触发的分支。上面的workflow只能在main分支中触发。
- `jobs`: 可以理解为工作列表。
- `upload_docker_images`: 定义一个job叫做 `upload_docker_images`。
- `runs-on`: 指定当前任务的运行环境。比如上面的workflow是在 `ubuntu-latest` 上运行。
- `steps`: 指定当前任务的步骤列表。
- `uses`: 指定使用某一写好的action, 可以在github-action的 Marketplace 中搜索现成的action, 也可以自定义。比如 `action/checkout@v3` 就是一个github上提供的将仓库中的代码复制到运行器上的action。可以让仓库中的代码在github的运行器中运行。除此之外, 还有非常丰富的action, 大家可以自行探索。这里举个简单的例子。之前我们讲了对象存储的使用。github上有专门的action可以帮我们将代码上传到存储桶中。

Marketplace
Documentation

Marketplace / Search results / Tencent COS Action



Tencent COS Action

v0.1.0
41

GitHub Action for Tencent COS Command (coscmd)

[View full Marketplace listing](#)

Installation

Copy and paste the following snippet into your .yaml file.

Version: v0.1.0

```

- name: Tencent COS Action
  # You may pin to the exact commit or the
  # uses: zkqiang/tencent-cos-action@0caa1
  uses: zkqiang/tencent-cos-action@v0.1.0
  with:
    # COSCMD args, detail: https://cloud.t
    args:
    # Tencent cloud SecretId, from: https:
    secret_id:
    # Tencent cloud SecretKey, from: https:

```

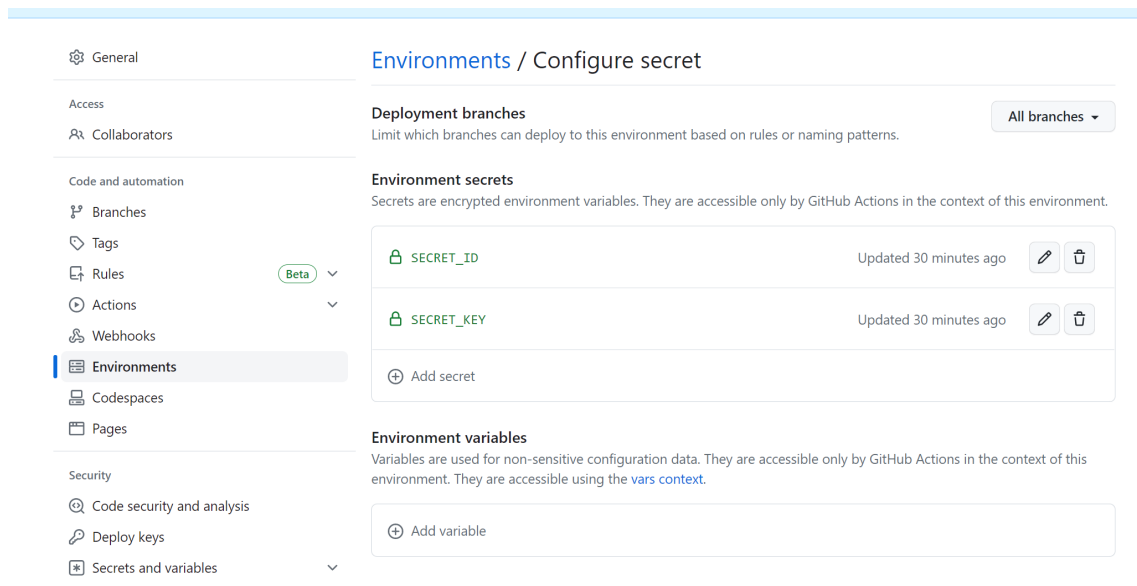
我们只需要将下面的代码复制到我们自己的workflow中, 并按照他的说明填写参数。

```

1 - name: Tencent COS Action
2   # You may pin to the exact commit or the version.
3   # uses: zkqiang/tencent-cos-
  action@0caa1b4f62d8531085abd73ac5c2f62570609752
4   uses: zkqiang/tencent-cos-action@v0.1.0
5   with:
6     # COSCMD args, detail:
    https://cloud.tencent.com/document/product/436/10976
7     args:
8     # Tencent cloud SecretId, from:
    https://console.cloud.tencent.com/cam/capi
9     secret_id:
10    # Tencent cloud SecretKey, from:
    https://console.cloud.tencent.com/cam/capi
11    secret_key:
12    # COS bucket name
13    bucket:
14    # COS bucket region, detail:
    https://cloud.tencent.com/document/product/436/6224
15    region:

```

其中，`secret_id`之类的变量要写在环境变量中，这样不会泄露敏感信息。可以在任意一个 repository 的 Settings 中选择 Environments，添加环境变量。注意一些敏感信息应该要添加到 **Environment secrets** 中，使用的时候用 `secrets.变量名` 进行访问，而非敏感信息可以写在 **Environment variables** 中，使用的时候用 `vars.变量名` 进行访问。



根据表达式的语法规则，如果想要引用表达式的值，需要写成 `${{ express }}`，比如上面示例中的写法。

- `- xxx`：为每一步的开始，注意有空格。
 - `- name`：标注每一个 job 的名字。可选项。
 - `run`：在运行器上运行的命令。
- Github 的 Action 有着非常强大和复杂的功能，我们这里仅仅给大家举一个非常简单的例子供大家了解。有兴趣深入学习的同学可以参考 github 的官方文档。<https://docs.github.com/en/actions>

作业

1. 阅读云盘链接<https://cloud.tsinghua.edu.cn/d/74670cf2f9a545219477/> 的Program.cs中的代码，注册腾讯云账号并创建存储桶，修改 Program.cs 中的代码，为其添加上传文件夹，下载文件夹的功能。
2. 建议使用WSL完成下面的任务：
 - 将下面的add.cpp和mul.cpp复制在宿主机目录下

```
1 //add.cpp
2 #include <iostream>
3
4 int main(int argc, char* argv[])
5 {
6     if(argc < 3){
7         std::cout << "Usage: " << argv[0] << " <num1> <num2>" <<
std::endl;
8         return 1;
9     }
10    else{
11        int num1 = atoi(argv[1]);
12        int num2 = atoi(argv[2]);
13        std::cout << num1 << " + " << num2 << " = " << num1 + num2
<< std::endl;
14    }
15    return 0;
16 }
```

```
1 //mul.cpp
2 #include <iostream>
3
4 int main(int argc, char* argv[])
5 {
6     if(argc < 3){
7         std::cout << "Usage: " << argv[0] << " <num1> <num2>" <<
std::endl;
8         return 1;
9     }
10    else{
11        int num1 = atoi(argv[1]);
12        int num2 = atoi(argv[2]);
13        std::cout << num1 << " * " << num2 << " = " << num1 * num2
<< std::endl;
14    }
15    return 0;
16 }
```

- 这是两个需要命令行参数才能正常运行的程序。请写一个 Dockerfile，用挂载本地目录作为数据卷的方式，用容器编译这两份代码，并将编译好的文件返回给宿主机。不允许使用交互式方式启动容器，也不允许直接将运行结果进行输出，而是通过 docker logs 查看运行结果。要求每次启动容器只编译并运行一个文件，具体编译哪个文件通过 docker run 时传入参数指定，同时传入程序运行所需的参数。（提示：编译c++文件可以直接在ubuntu中安装g++）。

提交时请向云盘<https://cloud.tsinghua.edu.cn/u/d/cbd7c1cdbf084f01a3cd/>上传压缩包，并按照“姓名_学号”的方式命名。DDL为28天。