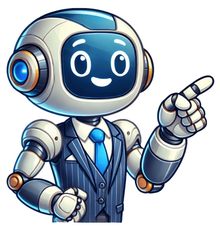
I'm not a bot



张量是一个多维数组，可以是标量、向量、矩阵或更高维度的数据结构。 在 PyTorch 中，张量（Tensor）是数据的核心表示形式，类似于 NumPy 的多维数组，但具有更强大的功能，例如支持 GPU 加速和自动梯度计算。 张量支持多种数据类型（整型、浮点型、布尔型等）。 张量可以存储在 CPU 或 GPU 中，GPU 张量可显著加速计算。 下图展示了不同维度的张量（Tensor）在 PyTorch 中的表示方法： 说明： 1D Tensor / Vector（一维张量/向量）：最基本的张量形式，可以看作是一个数组，图中的例子是一个包含 10 个元素的向量。 2D Tensor / Matrix（二维张量/矩阵）：二维数组，通常用于表示矩阵，图中的例子是一个 4x5 的矩阵，包含了 20 个元素。 3D Tensor / Cube（三维张量/立方体）：三维数组，可以看作是由多个矩阵堆叠而成的立方体，图中的例子展示了一个 3x4x5 的立方体，其中每个 5x5 的矩阵代表立方体的一个"层"。 4D Tensor / Vector of Cubes（四维张量/立方体向量）：四维数组，可以看作是由多个立方体组成的向量，图中的例子没有具体数值，但可以理解为一个包含多个 3D 张量的集合。 5D Tensor / Matrix of Cubes（五维张量/立方体矩阵）：五维数组，可以看作是由多个4D张量组成的矩阵，图中的例子同样没有具体数值，但可以理解为一个包含多个 4D 张量的集合。 创建张量 张量创建的方式有： 方法说明示例代码torch.tensor(data)从 Python 列表或 NumPy 数组创建张量。x = torch.tensor([[1, 2], [3, 4]])torch.zeros(size)创建一个全为零的张量。x = torch.zeros((2, 3))torch.ones(size)创建一个全为 1 的张量。x = torch.ones((2, 3))torch.empty(size)创建一个未初始化的张量。x = torch.empty((2, 3))torch.rand(size)创建一个服从均匀分布的随机张量，值在 [0, 1)。x = torch.rand((2, 3))torch.randn(size)创建一个服从正态分布的随机张量，均值为 0，标准差为 1。x = torch.randn((2, 3))torch.arange(start, end, step)创建一个一维序列张量，类似于 Python 的 range。x = torch.arange(0, 10, 2)torch.linspace(start, end, steps)创建一个在指定范围内等间隔的序列张量。x = torch.linspace(0, 1, 5)torch.eye(size)创建一个单位矩阵（对角线为 1，其他为 0）。x = torch.eye(3)torch.from_numpy(ndarray)将 NumPy 数组转换为张量。x = torch.from_numpy(np.array([1, 2, 3]))使用 torch.tensor() 函数，你可以将一个列表或数组转换为张量： import torch tensor = torch.tensor([1, 2, 3]) print(tensor) 输出如下： tensor([1, 2, 3]) 如果你有一个 NumPy 数组，可以使用 torch.from_numpy() 将其转换为张量： import numpy as np np_array = np.array([1, 2, 3]) tensor = torch.from_numpy(np_array) print(tensor) 输出如下： tensor([1, 2, 3]) 创建 2D 张量（矩阵）： import torch tensor_2d = torch.tensor([[-9, 4, 2, 5, 7], [3, 0, 12, 8, 6], [1, 23, -6, 45, 2], [22, 3, -1, 72, 6]]) print("2D Tensor (Matrix):", tensor_2d) print("Shape:", tensor_2d.shape) # 形状 输出如下： 2D Tensor (Matrix): tensor([[-9, 4, 2, 5, 7], [3, 0, 12, 8, 6], [1, 23, -6, 45, 2], [22, 3, -1, 72, 6]]) Shape: torch.Size([4, 5]) 其他维度的创建： # 创建 3D 张量（立方体） tensor_3d = torch.stack([tensor_2d, tensor_2d + 10, tensor_2d - 5]) # 堆叠 3 个 2D 张量 print("3D Tensor (Cube):", tensor_3d) print("Shape:", tensor_3d.shape) # 形状 # 创建 4D 张量（向量的立方体） tensor_4d = torch.stack([tensor_3d, tensor_3d + 100]) # 堆叠 2 个 3D 张量 print("4D Tensor (Vector of Cubes):", tensor_4d) print("Shape:", tensor_4d.shape) # 形状 # 创建 5D 张量（矩阵的立方体） tensor_5d = torch.stack([tensor_4d, tensor_4d + 1000]) # 堆叠 2 个 4D 张量 print("5D Tensor (Matrix of Cubes):", tensor_5d) print("Shape:", tensor_5d.shape) # 形状 张量的属性 张量的属性如下表： 属性说明示例.shape获取张量的形状tensor.shape.size()获取张量的形状tensor.size().dtype获取张量的数据类型tensor.dtype.device查看张量所在的设备 (CPU/GPU)tensor.device.dim()获取张量的维度数tensor.dim().requires_grad是否启用梯度计算tensor.requires_grad.numel()获取张量中的元素总数tensor.numel().is_cuda检查张量是否在 GPU 上tensor.is_cuda.T获取张量的转置（适用于 2D 张量）tensor.T.item()获取单元素张量的值tensor.item().is_contiguous()检查张量是否连续存储tensor.is_contiguous() import torch # 创建一个 2D 张量 tensor = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32) # 张量的属性 print("Tensor:", tensor) print("Shape:", tensor.shape) # 获取形状 print("Size:", tensor.size()) # 获取形状（另一种方法） print("Data Type:", tensor.dtype) # 数据类型 print("Device:", tensor.device) # 设备 print("Dimensions:", tensor.dim()) # 维度数 print("Total Elements:", tensor.numel()) # 元素总数 print("Requires Grad:", tensor.requires_grad) # 是否启用梯度 print("Is CUDA:", tensor.is_cuda) # 是否在 GPU 上 print("Is Contiguous:", tensor.is_contiguous()) # 是否连续存储 # 获取单元素值 single_value = torch.tensor(42) print("Single Element Value:", single_value.item()) # 转置张量 tensor_T = tensor.T print("Transposed Tensor:", tensor_T) 输出结果： Tensor: tensor([[1., 2., 3.], [4., 5., 6.]]) Shape: torch.Size([2, 3]) Size: torch.Size([2, 3]) Data Type: torch.float32 Device: cpu Dimensions: 2 Total Elements: 6 Requires Grad: False Is CUDA: False Is Contiguous: True Single Element Value: 42 Transposed Tensor: tensor([[1., 4.], [2., 5.], [3., 6.]]) 张量的操作 张量操作方法说明如下。 基础操作： 操作 说明 示例代码 +, -, *, / 元素级加法、减法、乘法、除法。 z = x + y torch.matmul(x, y) 矩阵乘法。 z = torch.matmul(x, y) torch.dot(x, y) 向量点积（仅适用于 1D 张量）。 z = torch.dot(x, y) torch.sum(x) 求和。 z = torch.sum(x) torch.mean(x) 求均值。 z = torch.mean(x) torch.max(x) 求最大值。 z = torch.max(x) torch.min(x) 求最小值。 z = torch.min(x) torch.argmax(x, dim) 返回最大值的索引（指定维度）。 z = torch.argmax(x, dim=1) torch.softmax(x, dim) 计算 softmax（指定维度）。 z = torch.softmax(x, dim=1) 形状操作 操作 说明 示例代码 x.view(shape) 改变张量的形状（不改变数据）。 z = x.view(3, 4) x.reshape(shape) 类似于 view，但更灵活。 z = x.reshape(3, 4) x.t() 转置矩阵。 z = x.t() x.unsqueeze(dim) 在指定维度添加一个维度。 z = x.unsqueeze(0) x.squeeze(dim) 去掉指定维度为 1 的维度。 z = x.squeeze(0) torch.cat((x, y), dim) 按指定维度连接多个张量。 z = torch.cat((x, y), dim=1) import torch # 创建一个 2D 张量 tensor = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32) print("原始张量:", tensor) # 1. **索引和切片操作** print("【索引和切片】") print("获取第一行:", tensor[0]) # 获取第一行 print("获取第一行第一列的元素:", tensor[0, 0]) # 获取特定元素 print("获取第二列的所有元素:", tensor[:, 1]) # 获取第二列所有元素 # 2. **形状变换操作** print("【形状变换】") reshaped = tensor.view(3, 2) # 改变张量形状为 3x2 print("改变形状后的张量:", reshaped) flattened = tensor.flatten() # 将张量展平成一维 print("展平后的张量:", flattened) # 3. **数学运算操作** print("【数学运算】") tensor_add = tensor + 10 # 张量加法 print("张量加 10:", tensor_add) tensor_mul = tensor * 2 # 张量乘法 print("张量乘 2:", tensor_mul) tensor_sum = tensor.sum() # 计算所有元素的和 print("张量元素的和:", tensor_sum.item()) # 4. **与其他张量的操作** print("【与其他张量操作】") tensor2 = torch.tensor([[1, 1, 1], [1, 1, 1]], dtype=torch.float32) print("另一个张量:", tensor2) tensor_dot = torch.matmul(tensor, tensor2.T) # 张量矩阵乘法 print("矩阵乘法结果:", tensor_dot) # 5. **条件判断和筛选** print("【条件判断和筛选】") mask = tensor > 3 # 创建一个布尔掩码 print("大于 3 的元素的布尔掩码:", mask) filtered_tensor = tensor[tensor > 3] # 筛选出符合条件的元素 print("大于 3 的元素:", filtered_tensor) 输出结果： 原始张量: tensor([[1., 2., 3.], [4., 5., 6.]]) 【索引和切片】 获取第一行: tensor([1., 2., 3.]) 获取第一行第一列的元素: tensor(1.) 获取第二列的所有元素: tensor([2., 5.]) 【形状变换】 改变形状后的张量: tensor([[1., 2.], [3., 4.], [5., 6.]]) 展平后的张量: tensor([1., 2., 3., 4., 5., 6.]) 【数学运算】 张量加 10: tensor([[11., 12., 13.], [14., 15., 16.]]) 张量乘 2: tensor([[2., 4., 6.], [8., 10., 12.]]) 张量元素的和: 21.0 【与其他张量操作】 另一个张量: tensor([[1., 1., 1.], [1., 1., 1.]]) 矩阵乘法结果: tensor([[6., 6.], [15., 15.]]) 【条件判断和筛选】 大于 3 的元素的布尔掩码: tensor([[False, False, False], [True, True, True]]) 大于 3 的元素: tensor([4., 5., 6.]) 张量的 GPU 加速 将张量转移到 GPU： device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') x = torch.tensor([1.0, 2.0, 3.0], device=device) 检查 GPU 是否可用： torch.cuda.is_available() # 返回 True 或 False 张量与 NumPy 的互操作 张量与 NumPy 的互操作如下表所示： 操作 说明 示例代码 torch.from_numpy(ndarray) 将 NumPy 数组转换为张量。 x = torch.from_numpy(np_array) x.numpy() 将张量转换为 NumPy 数组（仅限 CPU 张量）。 np_array = x.numpy() import torch import numpy as np # 1. NumPy 数组转换为 PyTorch 张量 print("1. NumPy 转为 PyTorch 张量") numpy_array = np.array([1, 2, 3], [4, 5, 6]]) print("NumPy 数组:", numpy_array) # 使用 torch.from_numpy() 将 NumPy 数组转换为张量 tensor_from_numpy = torch.from_numpy(numpy_array) print("转换后的 PyTorch 张量:", tensor_from_numpy) # 修改 NumPy 数组，观察张量的变化（共享内存） numpy_array[0, 0] = 100 print("修改后的 NumPy 数组:", numpy_array) print("PyTorch 张量也会同步变化:", tensor_from_numpy) # 2. PyTorch 张量转换为 NumPy 数组 print("2. PyTorch 张量转为 NumPy 数组") tensor = torch.tensor([[7, 8, 9], [10, 11, 12]], dtype=torch.float32) print("PyTorch 张量:", tensor) # 使用 tensor.numpy() 将张量转换为 NumPy 数组 numpy_from_tensor = tensor.numpy() print("转换后的 NumPy 数组:", numpy_from_tensor) # 修改张量，观察 NumPy 数组的变化（共享内存） tensor[0, 0] = 77 print("修改后的 PyTorch 张量:", tensor) print("NumPy 数组也会同步变化:", numpy_from_tensor) # 3. 注意：不共享内存的情况（需要复制数据） print("3. 使用 clone() 保证独立数据") tensor_independent = torch.tensor([[13, 14, 15], [16, 17, 18]], dtype=torch.float32) numpy_independent = tensor_independent.clone().numpy() # 使用 clone 复制数据 print("原始张量:", tensor_independent) tensor_independent[0, 0] = 0 # 修改张量数据 print("修改后的张量:", tensor_independent) print("NumPy 数组（不会同步变化）:", numpy_independent) 输出结果： 1. NumPy 转为 PyTorch 张量 NumPy 数组: [[1 2 3] [4 5 6]] 转换后的 PyTorch 张量: tensor([[1, 2, 3], [4, 5, 6]]) 修改后的 NumPy 数组: [[100 2 3] [4 5 6]] PyTorch 张量也会同步变化: tensor([[100, 2, 3], [4, 5, 6]]) 2. PyTorch 张量转为 NumPy 数组 PyTorch 张量: tensor([[7., 8., 9.], [10., 11., 12.]]) 转换后的 NumPy 数组: [[7. 8. 9.] [10. 11. 12.]] 修改后的 PyTorch 张量: tensor([[77., 8., 9.], [10., 11., 12.]]) NumPy 数组也会同步变化: [[77. 8. 9.] [10. 11. 12.]] 3. 使用 clone() 保证独立数据 原始张量: tensor([[13., 14., 15.], [16., 17., 18.]]) 修改后的张量: tensor([[0., 14., 15.], [16., 17., 18.]]) NumPy 数组（不会同步变化）: [[13. 14. 15.] [16. 17. 18.]] Linux 命令大全 Linux ls（英文全拼： list directory contents）命令用于显示指定工作目录下之内容（列出目前工作目录所含的文件及子目录）。 语法 ls [-alrtAFR] [name...] 参数：参数说明-a 或 --all显示所有文件（包括以 . 开头的隐藏文件）。-A 或 --almost-all显示除 . 和 .. 外的所有文件（包括隐藏文件）。-l以长格式（详细信息）列出文件（权限、所有者、大小、修改时间等）。-h 或 --human-readable与 -l 一起使用时，以人类可读的格式显示文件大小（如 KB、MB）。-t按修改时间排序（最新优先）。-r 或 --reverse反向排序（配合 -t、-S 等使用）。-S按文件大小排序（大文件优先）。-R 或 --recursive递归列出子目录内容。-F 或 --classify在文件名后附加标识符（如 / 表示目录，* 表示可执行文件）。--color彩色输出（通常默认启用，--color=auto）。-i 或 --inode显示文件的 inode 编号。-n 或 --numeric-uid-gid以数字形式显示 UID 和 GID（替代用户名和组名）。-d 或 --directory仅显示目录本身，而非其内容（常用于配合 -l）。-1每行只显示一个文件（默认在终端宽度不足时自动启用）。-m以逗号分隔的列表形式显示文件。-Q 或 --quote-name用引号括住文件名（适用于含空格的文件名）。--group-directories-first先显示目录，后显示文件。--time-style=自定义时间显示格式（如 +%Y-%m-%d）。 ls -l # 以长格式显示当前目录中的文件和目录 ls -a # 显示当前目录中的所有文件和目录，包括隐藏文件 ls -lh # 以人类可读的方式显示当前目录中的文件和目录大小 ls -t # 按照修改时间排序显示当前目录中的文件和目录 ls -R # 递归显示当前目录中的所有文件和子目录 ls -l /etc/passwd # 显示/etc/passwd文件的详细信息 实例 详细列出当前目录所有文件（含隐藏文件）： ls -la 按大小反向排序文件（大文件优先）： ls -lShr 递归列出 /var/log 目录内容，并显示人类可读的文件大小： ls -lhR /var/log 仅显示目录的详细信息（不递归）： ls -ld /etc 按修改时间排序（最新文件最后显示）： ls -ltr 列出根目录下的所有目录： # ls / bin dev lib media net root srv upload www boot etc lib64 misc opt sbin sys usr home lost+found mnt proc selinux tmp var 将 /bin 目录以下所有目录及文件详细资料列出: ls -lR /bin 当文件名包含空格、特殊字符或者开始字符为破折号时，可以使用反斜杠（\）进行转义，或者使用引号将文件名括起来。例如： ls "my file.txt" # 列出文件名为"my file.txt"的文件 ls my\ file.txt # 列出文件名为"my file.txt"的文件 ls -- -filename # 列出文件名为"-filename"的文件 ls 命令还可以使用通配符进行模式匹配，例如 * 表示匹配任意字符，? 表示匹配一个字符，[...] 表示匹配指定范围内的字符。例如： ls *.txt # 列出所有扩展名为.txt的文件 ls file?.txt # 列出文件名为file?.txt的文件，其中?表示任意一个字符 ls [abc]*.txt # 列出以a、b或c开头、扩展名为.txt的文件 列出目前工作目录下所有名称是 s 开头的文件，越新的排越后面: ls -ltr s* 在使用 ls -l 命令时，第一列的字符表示文件或目录的类型和权限。其中第一个字符表示文件类型，例如： - 表示普通文件 d 表示目录 l 表示符号链接 c 表示字符设备文件 b 表示块设备文件 s 表示套接字文件 p 表示管道文件 在使用 ls -l 命令时，第一列的其余 9 个字符表示文件或目录的访问权限，分别对应三个字符一组的 rwx 权限。例如： r 表示读取权限 w 表示写入权限 x 表示执行权限 - 表示没有对应权限 前三个字符表示所有者的权限，中间三个字符表示所属组的权限，后三个字符表示其他用户的权限。例如： -rw-r--r-- 1 user group 4096 Feb 21 12:00 file.txt 表示文件名为file.txt的文件，所有者具有读写权限，所属组和其他用户只有读取权限。 查找最近修改的文件： ls -lt | head -5 显示最近修改的 5 个文件。 统计文件数量： ls | wc -l 统计当前目录下的文件数量(不包括隐藏文件)。 注意事项 ls 命令的输出颜色可以通过 --color 选项控制： 蓝色：目录 绿色：可执行文件 红色：压缩文件 青色：链接文件 黄色：设备文件 在脚本中使用 ls 时要注意，直接解析 ls 的输出可能不可靠，建议使用其他方法。 不同 Linux 发行版的 ls 命令可能有细微差别，可以通过 man ls 查看具体帮助文档。 Linux 命令大全 Java 集合框架 ArrayList 类是一个可以动态修改的数组，与普通数组的区别就是它是没有固定大小的限制，我们可以添加或删除元素。 ArrayList 继承了 AbstractList，并实现了 List 接口。 ArrayList 类位于 java.util 包中，使用前需要引入它，语法格式如下： import java.util.ArrayList; // 引入 ArrayList 类 ArrayList objectName =new ArrayList(); // 初始化 E: 泛型数据类型，用于设置 objectName 的数据类型，只能为引用数据类型。 objectName: 对象名。 ArrayList 是一个数组队列，提供了相关的添加、删除、修改、遍历等功能。 添加元素 ArrayList 类提供了很多有用的方法，添加元素到 ArrayList 可以使用 add() 方法: import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); System.out.println(sites); } } 以上实例，执行输出结果为： [Google, Runoob, Taobao, Weibo] 访问元素 访问 ArrayList 中的元素可以使用 get() 方法： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); System.out.println(sites.get(1)); // 访问第二个元素 } } 注意：数组的索引值从 0 开始。 以上实例，执行输出结果为： Runoob 修改元素 如果要修改 ArrayList 中的元素可以使用 set() 方法， set(int index, E element) 方法的第一个参数是索引（index），表示要替换的元素的位置，第二个参数是新元素（element），表示要设置的新值： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); sites.set(2, "Wiki"); // 第一个参数为索引位置，第二个为要修改的值 System.out.println(sites); } } 以上实例，执行输出结果为： [Google, Runoob, Wiki, Weibo] 删除元素 如果要删除 ArrayList 中的元素可以使用 remove() 方法： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); sites.remove(3); // 删除第四个元素 System.out.println(sites); } } 以上实例，执行输出结果为： [Google, Runoob, Taobao] 计算大小 如果要计算 ArrayList 中的元素数量可以使用 size() 方法： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); System.out.println(sites.size()); } } 以上实例，执行输出结果为： 4 迭代数组列表 我们可以使用 for 来迭代数组列表中的元素： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); for (int i = 0; i < sites.size(); i++) { System.out.println(sites.get(i)); } } } 以上实例，执行输出结果为： Google Runoob Taobao Weibo 也可以使用 for-each 来迭代元素： import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Google"); sites.add("Runoob"); sites.add("Taobao"); sites.add("Weibo"); for (String i : sites) { System.out.println(i); } } } 以上实例，执行输出结果为： Google Runoob Taobao Weibo 其他的引用类型 ArrayList 中的元素实际上是对象，在以上实例中，数组列表元素都是字符串 String 类型。 如果我们要存储其他类型，而 只能为引用数据类型，这时我们就需要使用到基本类型的包装类。 基本类型对应的包装类表如下： 基本类型引用类型 booleanBoolean byteByte shortShort intInteger longLong floatFloat doubleDouble charCharacter 此外，BigInteger、BigDecimal 用于高精度的运算，BigInteger 支持任意精度的整数，也是引用类型，但它们没有相对应的基本类型。 ArrayList li=new ArrayList(); // 存放整数元素 ArrayList li=new ArrayList(); // 存放字符元素 以下实例使用 ArrayList 存储数字(使用 Integer 类型): import java.util.ArrayList; public class RunoobTest { public static void main(String[] args) { ArrayList myNumbers = new ArrayList(); myNumbers.add(10); myNumbers.add(15); myNumbers.add(20); myNumbers.add(25); for (int i : myNumbers) { System.out.println(i); } } } 以上实例，执行输出结果为： 10 15 20 25 ArrayList 排序 Collections 类也是一个非常有用的类，位于 java.util 包中，提供的 sort() 方法可以对字符或数字列表进行排序。 以下实例对字母进行排序： import java.util.ArrayList; import java.util.Collections; // 引入 Collections 类 public class RunoobTest { public static void main(String[] args) { ArrayList sites = new ArrayList(); sites.add("Taobao"); sites.add("Wiki"); sites.add("Runoob"); sites.add("Weibo"); sites.add("Google"); Collections.sort(sites); // 字母排序 for (String i : sites) { System.out.println(i); } } } 以上实例，执行输出结果为： Google Runoob Taobao Weibo Wiki 以下实例对数字进行排序： import java.util.ArrayList; import java.util.Collections; // 引入 Collections 类 public class RunoobTest { public static void main(String[] args) { ArrayList myNumbers = new ArrayList(); myNumbers.add(33); myNumbers.add(15); myNumbers.add(20); myNumbers.add(34); myNumbers.add(8); myNumbers.add(12); Collections.sort(myNumbers); // 数字排序 for (int i : myNumbers) { System.out.println(i); } } } 以上实例，执行输出结果为： 8 12 15 20 33 34 Java ArrayList 方法 Java ArrayList 常用方法列表如下： 更多 API 方法可以查看： Java 集合框架 C++ 标准库提供了丰富的功能，其中 是一个非常重要的容器类，用于存储元素集合，支持双向迭代器。 是 C++ 标准模板库（STL）中的一个序列容器，它允许在容器的任意位置快速插入和删除元素。 与数组或向量（）不同， 不需要在创建时指定大小，并且可以在任何位置添加或删除元素，而不需要重新分配内存。 语法 以下是 容器的一些基本操作： 包含头文件： #include 声明列表： std::list mylist; ; 其中 T 是存储在列表中的元素类型。 插入元素： mylist.push_back(value); 删除元素： mylist.pop_back(); 或 mylist.erase(iterator); 访问元素： mylist.front(); 和 mylist.back(); 遍历列表：使用迭代器 for (auto it = mylist.begin(); it != mylist.end(); ++it) 特点 双向迭代： 提供了双向迭代器，可以向前和向后遍历元素。 动态大小：与数组不同，的大小可以动态变化，不需要预先分配固定大小的内存。 快速插入和删除：可以在列表的任何位置快速插入或删除元素，而不需要像向量那样移动大量元素。 声明与初始化 的声明和初始化与其他容器类似： #include #include int main() { std::list list1; // 空的list std::list list2(5); // 包含5个默认初始化元素的list std::list list3(5, 10); // 包含5个元素，每个元素为10 std::list list4 = {1, 2, 3, 4}; // 使用初始化列表 return 0; } 实例 下面是一个使用 的简单示例，包括创建列表、添加元素、遍历列表和输出结果。 #include #include int main() { // 创建一个整数类型的列表 std::list numbers; // 向列表中添加元素 numbers.push_back(10); numbers.push_back(20); numbers.push_back(30); // 访问并打印列表的第一个元素 std::cout a = ['a', 'b', 'c'] >>> n = [1, 2, 3] >>> x = [a, n] >>> x [['a', 'b', 'c'], [1, 2, 3]] >>> x[0] ['a', 'b', 'c'] >>> x[0][1] 'b' 列表比较 列表比较需要引入 operator 模块的 eq 方法（详见：Python operator 模块）： # 导入 operator 模块 import operator a = [1, 2] b = [2, 3] c = [2, 3] print("operator.eq(a,b): ", operator.eq(a,b)) print("operator.eq(c,b): ", operator.eq(c,b)) 以上代码输出结果为： operator.eq(a,b): False operator.eq(c,b): True Python列表函数&方法 Python包含以下函数: Python包含以下方法: Ollama 提供了多种命令行工具（CLI）供用户与本地运行的模型进行交互。 我们可以用 ollama --help 查看包含有哪些命令： Large language model runner Usage: ollama [flags] ollama [command] Available Commands: serve Start ollama create Create a model from a Modelfile show Show information for a model run A model stop Stop a running model pull Pull a model from a registry push Push a model to a registry list List models ps List running models cp Copy a model rm Remove a model help Help about any command Flags: -h, --help help for ollama -v, --version Show version information 1、使用方法 ollama [flags]：使用标志（flags）运行 ollama。 ollama [command]：运行 ollama 的某个具体命令。 2、可用命令 serve：启动 ollama 服务。 create：根据一个 Modelfile 创建一个模型。 show：显示某个模型的详细信息。 run：运行一个模型。 stop：停止一个正在运行的模型。 pull：从一个模型仓库（registry）拉取一个模型。 push：将一个模型推送到一个模型仓库。 list：列出所有模型。 ps：列出所有正在运行的模型。 cp：复制一个模型。 rm：删除一个模型。 help：获取关于任何命令的帮助信息。 3、标志（Flags） -h, --help：显示 ollama 的帮助信息。 -v, --version：显示版本信息。 1. 模型管理 拉取模型 从模型库中下载模型： ollama pull 例如： ollama pull llama2 运行模型 运行已下载的模型： ollama run 例如： ollama run llama2 列出本地模型 查看已下载的模型列表： ollama list 删除模型 删除本地模型： ollama rm 例如： ollama rm llama2 2. 自定义模型 创建自定义模型 基于现有模型创建自定义模型： ollama create -f 例如： ollama create my-llama2 -f ./Modelfile 复制模型 复制一个已存在的模型： ollama cp 例如： ollama cp llama2 my-llama2-copy 推送自定义模型 将自定义模型推送到模型库： ollama push 例如： ollama push my-llama2 3. 服务管理 启动 Ollama 服务 启动 Ollama 服务以在后台运行： ollama serve 停止 Ollama 服务 停止正在运行的 Ollama 服务： ollama stop 重启 Ollama 服务 重启 Ollama 服务： ollama restart 4. 其他常用命令 查看帮助 查看所有可用命令： ollama --help 查看版本信息 查看当前安装的 Ollama 版本： ollama version 更新 Ollama 到最新版本： ollama update 查看日志 查看 Ollama 的日志信息： ollama logs 清理缓存 清理 Ollama 的缓存： ollama clean 5. 模型信息 查看模型详细信息 查看指定模型的详细信息： ollama show 例如： ollama show llama2 查看模型依赖 查看模型的依赖关系： ollama deps 例如： ollama deps llama2 查看模型配置 查看模型的配置文件： ollama config 例如： ollama config llama2 6. 导入与导出 导出模型 将模型导出为文件： ollama export 例如： ollama export llama2 llama2.tar 导入模型 从文件导入模型： ollama import 例如： ollama import llama2.tar 7. 系统信息 查看系统信息 查看 Ollama 的系统信息： ollama system 查看资源使用情况 查看模型的资源使用情况： ollama resources 例如： ollama resources llama2 8. 模型性能 查看模型性能 查看模型性能： ollama perf 例如： ollama perf llama2 9. 模型历史 查看模型历史记录 查看模型的历史记录： ollama history 例如： ollama history llama2 10. 模型状态 检查模型状态 检查指定模型的状态： ollama status 例如： ollama status llama2 序列是Python中最基本的数据结构。 序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。 Python有6个序列的内置类型，但最常见的是列表和元组。 序列都可以进行的操作包括索引，切片，加，乘，检查成员。 此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法。 列表是最常用的Python数据类型，它可以作为一个方括号内的逗号分隔值出现。 列表的数据项不需要具有相同的类型 创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。 如下所示： list1 = ['physics', 'chemistry', 1997, 2000] list2 = [1, 2, 3, 4, 5] list3 = ["a", "b", "c", "d"] 与字符串的索引一样，列表索引从0开始。 列表可以进行截取、组合等。 访问列表中的值 使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示： list1 = ['physics', 'chemistry', 1997, 2000] list2 = [1, 2, 3, 4, 5, 6, 7] print "list1[0]: ", list1[0] print "list2[1:5]: ", list2[1:5] 以上实例输出结果： list1[0]: physics list2[1:5]: [2, 3, 4, 5] 更新列表 你可以对列表的数据项进行修改或更新，你也可以使用append()方法来添加列表项，如下所示： list = [] list.append('Google') list.append('Runoob') print list 注意：我们会在接下来的章节讨论append()方法的使用 以上实例输出结果： ['Google', 'Runoob'] 删除列表元素 可以使用 del 语句来删除列表的元素，如下实例： list1 = ['physics', 'chemistry', 1997, 2000] print list1 del list1[2] print "After deleting value at index 2 : " print list1 以上实例输出结果： ['physics', 'chemistry', 1997, 2000] After deleting value at index 2 : ['physics', 'chemistry', 2000] 注意：我们会在接下来的章节讨论remove()方法的使用 Python列表脚本操作符 列表对 + 和 * 的操作符与字符串相似。 + 号用于组合列表，* 号用于重复列表。 如下所示： Python 表达式结果 描述 len([1, 2, 3])3长度 [1, 2, 3] + [4, 5, 6][1, 2, 3, 4, 5, 6]组合 ['Hi!'] * 4['Hi!', 'Hi!', 'Hi!', 'Hi!']重复 3 in [1, 2, 3]True元素是否存在于列表中 for x in [1, 2, 3]: print x,1 2 3迭代 Python列表截取 Python 的列表截取实例如下： >>>L = ['Google', 'Runoob', 'Taobao'] >>> L[2] 'Taobao' >>> L[-2] 'Runoob' >>> L[1:] ['Runoob', 'Taobao'] >>> 描述： Python 表达式结果 描述 L[2]'Taobao'读取列表中第三个元素 L[-2]'Runoob'读取列表中倒数第二个元素 L[1:]['Runoob', 'Taobao']从第二个元素开始截取列表 Python列表函数&方法 Python包含以下函数: Python包含以下方法: Python 列表 描述 sort() 函数用于对原列表进行排序，如果指定参数，则使用比较函数指定的比较函数。 语法 sort()方法语法： list.sort(cmp=None, key=None, reverse=False) 参数 cmp -- 可选参数, 如果指定了该参数会使用该参数的方法进行排序。 key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。 reverse -- 排序规则，reverse = True 降序， reverse = False 升序（默认）。 返回值 该方法没有返回值，但是会对列表的对象进行排序。 实例 以下实例展示了 sort() 函数的使用方法： aList = ['123', 'Google', 'Runoob', 'Taobao', 'Facebook']; aList.sort(); print("List : ") print(aList) 以上实例输出结果如下： List : ['123', 'Facebook', 'Google', 'Runoob', 'Taobao'] 以下实例降序输出列表： vowels = ['e', 'a', 'u', 'o', 'i'] vowels.sort(reverse=True) print('降序输出:') print(vowels) 以上实例输出结果如下： 降序输出: ['u', 'o', 'i', 'e', 'a'] 以下实例演示了通过指定列表中的元素排序来输出列表： def takeSecond(elem): return elem[1] random = [(2, 2), (3, 4), (4, 1), (1, 3)] random.sort(key=takeSecond) print('排序列表：') print(random) 以上实例输出结果如下： 排序列表： [(4, 1), (2, 2), (1, 3), (3, 4)] Python 列表 Markdown 支持有序列表和无序列表。 无序列表 无序列表使用星号(*)、加号(+)或是减号(-)作为列表标记，这些标记后面要添加一个空格，然后再填写内容： * 第一项 * 第二项 * 第三项 + 第一项 + 第二项 + 第三项 - 第一项 - 第二项 - 第三项 显示结果如下： 选择建议： 建议统一使用减号 - ，因为它在视觉上更清晰 在同一文档中保持一致的标记方式 标记符号后必须有一个空格 有序列表 有序列表用于展示有顺序要求的步骤或项目。 有序列表使用数字并加上 . 号来表示，如： 1. 第一项 2. 第二项 3. 第三项 显示结果如下： 数字可以不连续 Markdown 会自动修正数字顺序： 1. 第一项 3. 第二项（实际显示为2） 7. 第三项（实际显示为3） 从指定数字开始： 5. 第五项 6. 第六项 7. 第七项 列表嵌套 列表嵌套技巧 列表可以嵌套使用，创建多层次的结构： 无序列表嵌套： - 水果 - 苹果 - 红苹果 - 绿苹果 - 香蕉 - 橙子 - 蔬菜 - 胡萝卜 - 白菜 有序列表嵌套： 1. 准备阶段 1. 收集资料 2. 制定计划 2. 执行阶段 1. 开始实施 2. 监控进度 3. 总结阶段 混合嵌套： 1. 主要任务 - 子任务A - 子任务B 1. 详细步骤1 2. 详细步骤2 - 子任务C 2. 次要任务 列表混合嵌套只需在子列表中的选项前面添加两个或四个空格即可： 1. 第一项： - 第一项嵌套的第一个元素 - 第一项嵌套的第二个元素 2. 第二项： - 第二项嵌套的第一个元素 - 第二项嵌套的第二个元素 显示结果如下： 嵌套规则： 子列表需要缩进 2-4 个空格（推荐 2 个） 保持一致的缩进长度 可以无限层嵌套，但实际使用中建议不超过 3 层 任务列表（复选框列表） 任务列表是 GitHub 风格 Markdown 的扩展功能，现在被广泛支持： 基本语法： - [ ] 未完成的任务 - [x] 已完成的任务 - [ ] 另一个未完成的任务 渲染效果： 实际应用示例： ## 项目待办清单 ### 设计阶段 - [x] 需求分析 - [x] 原型设计 - [ ] UI 设计 ### 开发阶段 - [ ] 前端开发 - [x] 页面布局 - [ ] 交互功能 - [ ] 响应式适配 - [ ] 后端开发 - [ ] 数据库设计 - [ ] API 开发 - [ ] 性能优化 ### 测试阶段 - [ ] 单元测试 - [ ] 集成测试 - [ ] 用户验收测试 使用技巧： 方括号内的空格和 x 很重要：[ ] 和 [x] 可以与嵌套列表结合使用 在项目管理、学习计划、生活清单中特别有用 某些编辑器支持点击复选框来切换状态 高级列表技巧： 列表项中包含多段内容： 1. 第一项 这是第一项的详细说明，需要与列表项对齐缩进。 还可以包含第二段内容。 2. 第二项 > 可以在列表项中使用引用 列表项中的换行： - 这是一个很长的列表项，需要换行显示，注意第二行需要与第一行对齐 - 另一个列表项 通过掌握这些基础语法，你已经能够创建结构清晰、格式规范的 Markdown 文档了。这些语法是日常写作中最常用的，熟练掌握它们将大大提高你的文档编写效率。 Ollama 是一个开源的本地大语言模型运行框架，专为在本地机器上便捷部署和运行大型语言模型（LLM）而设计。 Ollama 支持多种操作系统，包括 macOS、Windows、Linux 以及通过 Docker 容器运行。 Ollama 提供对模型量化的支持，可以显著降低显存要求，使得在普通家用计算机上运行大型模型成