        - 6. 获取大小和容量
        - 7. 预留和调整容量
    - hash
        - 删除元素
        - 查看大小
    - deque
        - 删除元素
- c++容器适配器
    - stack
    - queue
    - priority_queue
- c++好用的函数
    - 字符判断函数
        - 📚 常用字符判断函数（来自 <cctype>）：
        - ✅ 示例代码：
    - 字符串转数字
        - 1. 使用 std::stoi 函数
        - 2. 使用 std::atoi 函数
        - 3. 使用 std::atof 函数
        - 4. 使用 std::stringstream 类
        - 5.toupper && tolower
        - rand
- c++算法
- python迭代器用法
    - set
        - 基本用法
            - 1. 创建集合
            - 2. 添加元素
            - 3. 删除元素
            - 4. 集合的基本操作
            - 5. 检查元素是否在集合中
            - 6. 集合的其他常用方法
        - 示例：集合的常见操作
        - 注意：
    - deque
    - defaultdict
    - 优先队列
        - 1. 使用 queue.PriorityQueue

c++容器用法

set

基本用法

1. 包含头文件

要使用 `std::set`，首先需要包含头文件：

```
#include <set>
#include <iostream>
```

2. 声明和初始化

`std::set` 默认按升序排序元素，也可以指定排序的规则。

```
std::set<int> mySet;  // 创建一个整型集合，默认按升序排序
```

3. 插入元素

可以使用 `insert()` 方法将元素插入集合。`set` 会自动忽略重复的元素。

```
mySet.insert(10);
mySet.insert(20);
mySet.insert(10);  // 插入重复元素，集合中不会存储重复值
```

4. 遍历集合

使用范围 `for` 循环可以遍历集合中的元素。

```
for (const auto& elem : mySet) {
    std::cout << elem << " ";
}
std::cout << std::endl;
```

5. 查找元素

使用 `find()` 方法查找一个元素。如果元素存在，返回指向该元素的迭代器；如果不存在，返回 `end()` 迭代器。

```
auto it = mySet.find(10);
if (it != mySet.end()) {
    std::cout << "Found: " << *it << std::endl;
} else {
    std::cout << "Not Found" << std::endl;
}
```

6. 删除元素

使用 `erase()` 方法删除指定元素或指定位置的元素。

```
mySet.erase(10);  // 删除元素 10
```

7. 其他常用操作

- **`size()`**：获取集合中元素的个数。
- **`empty()`**：检查集合是否为空。

```
std::cout << "Size: " << mySet.size() << std::endl;
std::cout << "Is empty: " << (mySet.empty() ? "Yes" : "No") << std::endl;
```

示例代码：

```
#include <iostream>
#include <set>

int main() {
```

```
    std::set<int> mySet;

    // 插入元素
    mySet.insert(5);
    mySet.insert(2);
    mySet.insert(8);
    mySet.insert(2);  // 这个元素不会被插入，重复元素会被忽略

    // 遍历并打印元素
    std::cout << "Set elements: ";
    for (const auto& elem : mySet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // 查找元素
    auto it = mySet.find(5);
    if (it != mySet.end()) {
        std::cout << "Element 5 found." << std::endl;
    } else {
        std::cout << "Element 5 not found." << std::endl;
    }

    // 删除元素
    mySet.erase(2);  // 删除元素 2

    // 打印删除后的集合
    std::cout << "Set after deletion: ";
    for (const auto& elem : mySet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // 检查集合大小
    std::cout << "Size of set: " << mySet.size() << std::endl;

    return 0;
}
```

输出：

```
Set elements: 2 5 8
Element 5 found.
Set after deletion: 5 8
Size of set: 2
```

注意：

1. `std::set` 中的元素是 **唯一的**，即不会存储重复元素。

2. 默认按 **升序** 排序元素，可以通过指定自定义比较函数来改变排序方式。
3. `std::set` 提供了 **高效的查找、插入和删除操作** ，复杂度通常为 `O(log n)`。

使用自定义排序：

如果你希望集合按照降序或其他特定顺序进行排序，可以指定比较函数。例如：

```
std::set<int, std::greater<int>> mySet;  // 降序排列

mySet.insert(10);
mySet.insert(20);
mySet.insert(5);

for (const auto& elem : mySet) {
    std::cout << elem << " ";  // 输出：20 10 5
}
```

通过这些基本操作，你可以在 C++ 中有效地使用集合（`std::set`）。

在 C++ 中，`std::set` 提供了一些非常方便的成员函数和算法来执行集合操作（如求交集、并集等）。这些操作通常可以通过标准库的算法 `std::set_intersection`、`std::set_union`、`std::set_difference` 等来实现。

常用集合操作：

- **交集（Intersection）**：找出两个集合中共同存在的元素。
- **并集（Union）**：合并两个集合，去除重复元素。
- **差集（Difference）**：找出第一个集合中存在而第二个集合中不存在的元素。

示例代码

```
#include <iostream>
#include <set>
#include <algorithm>  // std::set_intersection, std::set_union,
std::set_difference

int main() {
    // 创建两个集合
    std::set<int> set1 = {1, 2, 3, 4, 5};
    std::set<int> set2 = {3, 4, 5, 6, 7};
```

```
    // 求交集
    std::set<int> intersectionSet;
    std::set_intersection(set1.begin(), set1.end(), set2.begin(),
set2.end(),
                          std::inserter(intersectionSet,
intersectionSet.begin()));

    std::cout << "Intersection: ";
    for (const auto& elem : intersectionSet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // 求并集
    std::set<int> unionSet;
    std::set_union(set1.begin(), set1.end(), set2.begin(), set2.end(),
                   std::inserter(unionSet, unionSet.begin()));

    std::cout << "Union: ";
    for (const auto& elem : unionSet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    // 求差集
    std::set<int> differenceSet;
    std::set_difference(set1.begin(), set1.end(), set2.begin(), set2.end(),
                        std::inserter(differenceSet,
differenceSet.begin()));

    std::cout << "Difference (set1 - set2): ";
    for (const auto& elem : differenceSet) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

输出：

```
Intersection: 3 4 5
Union: 1 2 3 4 5 6 7
Difference (set1 - set2): 1 2
```

解释：

1. **交集（Intersection）**：

- 使用 `std::set_intersection` 来找到 `set1` 和 `set2` 中共同的元素。这个算法的结果是 `set1` 和 `set2` 的交集，存储在 `intersectionSet` 中。

```
    std::set_intersection(set1.begin(), set1.end(), set2.begin(), set2.end(),
                          std::inserter(intersectionSet,
intersectionSet.begin()));
```

1. **并集（Union）**：

- 使用 `std::set_union` 来计算 `set1` 和 `set2` 的并集。该算法会将两个集合的所有元素合并，并且去除重复的元素（因为 `set` 本身只允许唯一元素）。

```
    std::set_union(set1.begin(), set1.end(), set2.begin(), set2.end(),
                   std::inserter(unionSet, unionSet.begin()));
```

1. **差集（Difference）**：

- 使用 `std::set_difference` 来找出 `set1` 中有而 `set2` 中没有的元素（`set1 - set2`）。结果存储在 `differenceSet` 中。

```
    std::set_difference(set1.begin(), set1.end(), set2.begin(), set2.end(),
                        std::inserter(differenceSet, differenceSet.begin()));
```

说明：

- **`std::set_intersection`**：计算两个集合的交集。
- **`std::set_union`**：计算两个集合的并集。
- **`std::set_difference`**：计算两个集合的差集（即一个集合中有而另一个集合中没有的元素）。

这些算法要求输入的集合是已经排序的，这也是 `std::set` 容器天然支持这些操作的原因。

其他集合操作：

- **对称差集（Symmetric Difference）**：计算两个集合的对称差集（即两个集合的元素中，只在一个集合中出现的元素）。使用 `std::set_symmetric_difference` 可以实现。

```
  std::set<int> symmetricDifferenceSet;
  std::set_symmetric_difference(set1.begin(), set1.end(), set2.begin(),
set2.end(),
                                std::inserter(symmetricDifferenceSet,
symmetricDifferenceSet.begin()));
```

这些集合操作非常高效，能够在 `O(n log n)` 的时间复杂度下完成，适用于需要处理大量数据的场景。

string

1. 包含头文件

在使用 `std::string` 类之前，需要包含 `<string>` 头文件：

```
#include <string>
```

2. 创建和初始化字符串

可以通过多种方式创建和初始化 `std::string` 对象：

```
#include <iostream>
#include <string>

int main() {
    std::string str1;                    // 默认构造函数，创建一个空字符串
    std::string str2("Hello, World!");   // 使用C风格字符串初始化
    std::string str3(str2);              // 拷贝构造函数
    std::string str4(5, 'A');            // 创建一个包含5个字符 'A' 的字符串

    std::cout << str1 << std::endl;      // 输出：（空行）
    std::cout << str2 << std::endl;      // 输出: Hello, World!
    std::cout << str3 << std::endl;      // 输出: Hello, World!
    std::cout << str4 << std::endl;      // 输出: AAAAA
```

```
    return 0;
}
```

3. 获取字符串长度

使用 `length()` 或 `size()` 方法获取字符串的长度，这两个方法功能相同：

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello";
    std::cout << "Length: " << str.length() << std::endl; // 输出: Length: 5
    std::cout << "Size: " << str.size() << std::endl;     // 输出: Size: 5

    return 0;
}
```

4. 字符串拼接

可以使用 `+` 运算符或 `append()` 方法进行字符串拼接：

```
#include <iostream>
#include <string>

int main() {
    std::string str1 = "Hello";
    std::string str2 = " World";

    // 使用 + 运算符拼接
    std::string str3 = str1 + str2;
    std::cout << str3 << std::endl; // 输出: Hello World

    // 使用 append() 方法拼接
    str1.append(str2);
    std::cout << str1 << std::endl; // 输出: Hello World

    return 0;
}
```

5. 访问和修改字符串中的字符

可以使用下标运算符 `[]` 或 `at()` 方法访问和修改字符串中的字符：

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello";

    // 访问字符
    char ch1 = str[0];       // 'H'
    char ch2 = str.at(1);    // 'e'

    std::cout << ch1 << std::endl; // 输出: H
    std::cout << ch2 << std::endl; // 输出: e

    // 修改字符
    str[0] = 'h';
    str.at(1) = 'a';

    std::cout << str << std::endl; // 输出: hallo

    return 0;
}
```

6. 子字符串提取

使用 `substr()` 方法提取子字符串：

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";
    std::string subStr = str.substr(7, 5); // 从位置7开始，提取5个字符

    std::cout << subStr << std::endl; // 输出: World

    return 0;
}
```

7. 查找子字符串

使用 `find()` 方法查找子字符串的位置：

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";
    size_t pos = str.find("World");

    if (pos != std::string::npos) {
        std::cout << "Found at position: " << pos << std::endl; // 输出:
Found at position: 7
    } else {
        std::cout << "Not found" << std::endl;
    }

    return 0;
}
```

8. 替换子字符串

使用 `replace()` 方法替换子字符串：

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";
    str.replace(7, 5, "C++"); // 从位置7开始，替换5个字符为 "C++"

    std::cout << str << std::endl; // 输出: Hello, C++!

    return 0;
}
```

9. 字符串比较

可以使用比较运算符（如 `==`、`!=`、`<`、`>` 等）或 `compare()` 方法进行字符串比较：

```cpp
#include <iostream>
#include <string>

int main() {
    std::string str1 = "Hello";
    std::string str2 = "World";

    // 使用比较运算符
```

```
    if (str1 == str2) {
        std::cout << "str1 equals str2" << std::endl;
    } else {
        std::cout << "str1 does not equal str2" << std::endl; // 输出: str1
does not equal str2
    }

    // 使用 compare() 方法
    if (str1.compare(str2) == 0) {
        std::cout << "str1 equals str2" << std::endl;
    } else {
        std::cout << "str1 does not equal str2" << std::endl; // 输出: str1
does not equal str2
    }

    return 0;
}
```

10. 转换为 C 风格字符串

使用 `c_str()` 方法将 `std::string` 转换为 C 风格字符串（以 null 终止的字符数组）：

```
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";
    const char* cStr = str.c_str();

    std::cout << cStr << std::endl; // 输出: Hello, World!

    return 0;
}
```

vector

在 C++ 中，`std::vector` 是标准模板库（STL）提供的一个动态数组容器，能够自动管理内存，支持在运行时动态地插入和删除元素。 □cite□turn0search4□

1. 引入头文件

在使用 `std::vector` 之前，需要包含 `<vector>` 头文件：

```
#include <vector>
```

2. 创建和初始化 vector

可以通过多种方式创建和初始化 vector：

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec1;                  // 创建一个空的 vector
    std::vector<int> vec2(10);              // 创建一个包含 10 个默认初始化元素的
vector
    std::vector<int> vec3(5, 100);          // 创建一个包含 5 个值为 100 的元素的
vector
    std::vector<int> vec4 = {1, 2, 3, 4};  // 使用列表初始化 vector
 // 创建一个 n 行 m 列的二维 vector，所有元素初始化为 0
    std::vector<std::vector<int>> vec(n, std::vector<int>(m, 0));

    return 0;
}
```

3. 访问和修改元素

可以使用下标运算符 [] 或 at() 方法访问和修改 vector 中的元素：

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec = {10, 20, 30, 40};

    // 使用下标访问元素
    std::cout << vec[0] << std::endl;  // 输出: 10

    // 使用 at() 方法访问元素
    std::cout << vec.at(1) << std::endl;  // 输出: 20

    // 修改元素
    vec[2] = 100;
    std::cout << vec[2] << std::endl;  // 输出: 100

    return 0;
}
```

注意： `at()` 方法在访问越界时会抛出 `std::out_of_range` 异常，而使用 `[]` 运算符时，行为未定义。

4. 添加和删除元素

`std::vector` 提供了多种方法来添加和删除元素：

- **`push_back()`**：在末尾添加元素。
- **`pop_back()`**：移除末尾元素。
- **`insert()`**：在指定位置插入元素。
- **`erase()`**：移除指定位置的元素或范围内的元素。
- **`clear()`**：移除所有元素。

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec = {1, 2, 3};

    // 在末尾添加元素
    vec.push_back(4);  // vec: {1, 2, 3, 4}

    // 移除末尾元素
    vec.pop_back();    // vec: {1, 2, 3}

    // 在指定位置插入元素
    vec.insert(vec.begin() + 1, 10);  // vec: {1, 10, 2, 3}

    // 移除指定位置的元素
    vec.erase(vec.begin() + 2);       // vec: {1, 10, 3}

    // 清空所有元素
    vec.clear();                      // vec: {}

    return 0;
}
```

5. 遍历 `vector`

可以使用迭代器或范围 `for` 循环遍历 `vector`：

```
#include <vector>
#include <iostream>
```

```cpp
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用迭代器遍历
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
{
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 使用范围 for 循环遍历
    for (const auto& elem : vec) {
        std::cout << elem << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

6. 获取大小和容量

`std::vector` 提供了以下方法来获取其大小和容量：

- **`size()`**：返回当前元素的数量。
- **`capacity()`**：返回在不重新分配内存的情况下，`vector` 可以容纳的元素数量。
- **`empty()`**：如果 `vector` 为空，返回 `true`，否则返回 `false`。

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec = {1, 2, 3};

    std::cout << "Size: " << vec.size() << std::endl;        // 输出: Size:
3
    std::cout << "Capacity: " << vec.capacity() << std::endl; // 输出:
Capacity: 3 或更大
    std::cout << "Is empty: " << std::boolalpha << vec.empty() << std::endl;
// 输出: Is empty: false

    return 0;
}
```

7. 预留和调整容量

可以使用 `reserve()` 方法预留内存，以减少多次重新分配的开销：

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vec;
    vec.reserve(100);  // 预留空间以容纳 100 个元素

    for (int i = 0; i < 100; ++i) {
        vec.push_back(i);
    }

    std::cout << "Size: " << vec.size() << std::endl;        // 输出: Size:
100
    std::cout << "Capacity: " << vec.capacity() << std::endl; // 输出:
Capacity: 100 或更大

    return 0;
}
```

注意： `reserve()` 只增加容量，不改变实际元素的数量；而 `resize()` 可以改变 `vector` 的大小，并初始化新增的元素。

hash

在C++中，哈希表（Hash Table）是一种通过哈希函数将关键字映射到表中位置，以实现高效数据存取的数据结构。C++标准库提供了无序关联容器（unordered associative containers），如 `std::unordered_map` 和 `std::unordered_set`，它们是基于哈希表实现的。

使用 `std::unordered_map`：

`std::unordered_map` 是一个关联容器，用于存储键值对（key-value pairs），其中键是唯一的。其主要特点包括：

- **无序性**： 元素在容器中的存储顺序是无序的。
- **快速查找**： 平均情况下，查找、插入和删除操作的时间复杂度为 O(1)。

示例：

```cpp
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
```

```
    // 创建一个 unordered_map，键类型为 std::string，值类型为 int
    std::unordered_map<std::string, int> fruitMap;

    // 插入元素
    fruitMap["apple"] = 1;
    fruitMap["banana"] = 2;
    fruitMap["cherry"] = 3;

    // 查找并输出元素
    std::string key = "banana";
    auto it = fruitMap.find(key);
    if (it != fruitMap.end()) {
        std::cout << "Key: " << key << ", Value: " << it->second <<
std::endl;
    } else {
        std::cout << "Key not found." << std::endl;
    }

    // 遍历并输出所有元素
    for (const auto& pair : fruitMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

输出：

```
Key: banana, Value: 2
apple: 1
banana: 2
cherry: 3
```

注意事项：

- **头文件**： 使用 `std::unordered_map` 需要包含头文件 `<unordered_map>`。
- **哈希函数**： 对于内置类型（如 `int`、`std::string`）的键，C++标准库提供了默认的哈希函数。如果使用自定义类型作为键，需要为该类型提供哈希函数。例如：

```
struct CustomType {
    int id;
    std::string name;
};

// 自定义哈希函数
struct CustomHash {
    std::size_t operator()(const CustomType& obj) const {
        std::size_t h1 = std::hash<int>{}(obj.id);
        std::size_t h2 = std::hash<std::string>{}(obj.name);
```

```
        return h1 ^ (h2 << 1); // 合并哈希值
    }
};
```

然后，在定义 `std::unordered_map` 时，指定自定义哈希函数：

```
std::unordered_map<CustomType, ValueType, CustomHash> customMap;
```

- **冲突处理**： 当不同的键通过哈希函数映射到相同的位置时，会发生哈希冲突。`std::unordered_map` 内部采用链地址法（separate chaining）来处理冲突，即在同一桶（bucket）中使用链表或其他结构存储多个元素。
- **性能考虑**： 虽然平均情况下操作时间复杂度为 O(1)，但在最坏情况下（例如大量哈希冲突），时间复杂度可能退化为 O(n)。因此，选择合适的哈希函数和合理的桶数量对于性能至关重要。

删除元素

```
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> myMap;
    myMap["apple"] = 1;
    myMap["banana"] = 2;
    myMap["cherry"] = 3;

    // 删除键为 "banana" 的元素
    myMap.erase("banana");

    // 输出剩余元素
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

查看大小

```cpp
#include <iostream>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> myMap;
    myMap["apple"] = 1;
    myMap["banana"] = 2;
    myMap["cherry"] = 3;

    // 获取哈希表中元素的数量
    std::cout << "Size of myMap: " << myMap.size() << std::endl;

    return 0;
}
```

```
Size of myMap: 3
```

关于“大小”的含义：

在哈希表中，“大小”通常有两个含义：

1. **元素数量（size）：** 哈希表中当前存储的键值对（元素）数量。
2. **桶数量（bucket count）：** 哈希表内部用于存储元素的桶（bucket）数量。桶的数量影响哈希表的性能，过少的桶可能导致更多的冲突，过多的桶可能浪费空间。

您可以使用以下代码获取桶数量：

```cpp
std::cout << "Bucket count: " << myMap.bucket_count() << std::endl;
```

deque

在 C++ 中，`deque`（双端队列）是标准模板库（STL）提供的容器，支持在序列的两端高效地插入和删除元素，同时允许随机访问。

1. 引入头文件

使用 `deque` 需要包含头文件：

```cpp
#include <deque>
```

2. 定义双端队列

可以定义存储特定类型元素的双端队列，例如：

```
    std::size_t n = 10; // 指定大小
      int initial_value = 5; // 指定初始值
      std::deque<int> dq(n, initial_value); // 创建一个包含 10 个元素的 deque，所
有元素初始化为 5
```

3. 常用成员函数

`deque` 提供以下常用操作：

- **元素访问：**
  - `at(size_type pos)`：返回指定位置 `pos` 处元素的引用，并进行范围检查。
  - `operator[](size_type pos)`：返回指定位置 `pos` 处元素的引用，不进行范围检查。
  - `front()`：返回首元素的引用。
  - `back()`：返回尾元素的引用。
- **修改容器：**
  - `push_back(const T& value)`：在末尾添加元素。
  - `push_front(const T& value)`：在头部添加元素。
  - `pop_back()`：移除末尾元素。
  - `pop_front()`：移除头部元素。
  - `insert(iterator pos, const T& value)`：在迭代器 `pos` 指定的位置前插入元素。
  - `erase(iterator pos)`：移除迭代器 `pos` 指定位置的元素。
  - `clear()`：清空所有元素。
- **容量相关：**
  - `empty()`：检查是否为空，若为空返回 `true`。
  - `size()`：返回元素个数。
  - `resize(size_type count)`：调整容器大小为 `count`，多出部分用默认值填充。

4. 示例代码

以下示例演示了 `deque` 的基本用法：

```cpp
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq;

    // 在末尾添加元素
    dq.push_back(10);
    dq.push_back(20);
    dq.push_back(30);

    // 在头部添加元素
    dq.push_front(5);
    dq.push_front(0);

    // 输出双端队列的元素
    std::cout << "双端队列中的元素: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // 访问首尾元素
    std::cout << "首元素: " << dq.front() << std::endl;
    std::cout << "尾元素: " << dq.back() << std::endl;

    // 删除首尾元素
    dq.pop_front();
    dq.pop_back();

    // 输出修改后的双端队列
    std::cout << "修改后的双端队列: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5. 注意事项

- `deque` 在两端插入和删除操作的时间复杂度为常数级别 O(1)，但在中间位置插入或删除的效率可能不如 `list`。
- 与 `vector` 相比，`deque` 在两端操作上更为高效，但由于其内存分配方式，可能会占用更多的内存。

通过以上方式，您可以在 C++ 中有效地使用 `deque` 来处理需要在序列两端频繁插入和删除的场景。

删除元素

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq = {10, 20, 30, 40, 50};

    // 删除第三个元素（值为30）
    auto it = dq.begin() + 2;
    dq.erase(it);

    // 输出删除后的双端队列
    std::cout << "删除特定位置元素后的双端队列: ";
    for (int num : dq) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

c++容器适配器

stack

在C++中，`std::stack`是标准模板库（STL）提供的一个容器适配器，用于实现栈（Stack）数据结构。栈是一种遵循后进先出（LIFO，Last In First Out）原则的线性数据结构，即最后添加的元素最先被移除。

`std::stack`的常用方法：

1. **`empty()`**：检查栈是否为空。

- **返回值：**如果栈为空，返回 `true`；否则，返回 `false`。
- **示例：**

```
std::stack<int> s;
if (s.empty()) {
    // 栈为空
}
```

1. **`size()`**：返回栈中元素的数量。

- **返回值：**栈中元素的个数。
- **示例：**

```
std::stack<int> s;
s.push(10);
s.push(20);
std::cout << "栈的大小: " << s.size() << std::endl; // 输出: 栈的大小: 2
```

1. **`top()`**：访问栈顶元素。

- **返回值：**栈顶元素的引用。
- **注意：**在调用 `top()`之前，建议先使用 `empty()`检查栈是否为空，以避免访问非法内存。
- **示例：**

```
std::stack<int> s;
s.push(10);
s.push(20);
if (!s.empty()) {
    std::cout << "栈顶元素: " << s.top() << std::endl; // 输出: 栈顶元素:
20
}
```

1. **`push(const value_type& val)`**：将元素添加到栈顶。

- **参数：**要添加到栈顶的元素值。
- **示例：**

```
std::stack<int> s;
s.push(10); // 将10压入栈顶
s.push(20); // 将20压入栈顶
```

1. **`pop()`**：移除栈顶元素。

- **注意：**在调用 `pop()`之前，建议先使用 `empty()`检查栈是否为空，以避免未定义行为。
- **示例：**

```
std::stack<int> s;
s.push(10);
s.push(20);
if (!s.empty()) {
    s.pop(); // 移除栈顶元素20
}
```

示例代码：

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);

    std::cout << "栈的大小: " << s.size() << std::endl; // 输出: 栈的大小: 3
    std::cout << "栈顶元素: " << s.top() << std::endl;   // 输出: 栈顶元素: 30

    s.pop(); // 移除栈顶元素

    std::cout << "移除栈顶元素后，栈的大小: " << s.size() << std::endl; // 输出: 栈的大小: 2
    std::cout << "移除栈顶元素后，栈顶元素: " << s.top() << std::endl;   // 输出: 栈顶元素: 20

    return 0;
}
```

注意事项：

- **底层容器：** `std::stack`默认使用 `std::deque`作为底层容器，也可以通过模板参数指定其他容器，如 `std::vector`或 `std::list`。
- **功能限制：** `std::stack`仅提供栈的基本操作，不支持直接访问栈中间的元素，也不提供迭代器支持。如果需要更灵活的操作，可以考虑使用其他容器，如 `std::vector`或 `std::list`。

queue

```
#include <iostream>
#include <queue>
```

```cpp
int main() {
    // 创建一个整数类型的队列
    std::queue<int> q;

    // 向队列尾部添加元素
    q.push(10);
    q.push(20);
    q.push(30);

    // 输出队列中的元素数量
    std::cout << "队列中的元素数量: " << q.size() << std::endl;

    // 输出队首元素
    std::cout << "队首元素: " << q.front() << std::endl;

    // 输出队尾元素
    std::cout << "队尾元素: " << q.back() << std::endl;

    // 移除队首元素
    q.pop();
    std::cout << "移除队首元素后，队首元素: " << q.front() << std::endl;

    // 再次输出队列中的元素数量
    std::cout << "队列中的元素数量: " << q.size() << std::endl;

    return 0;
}
```

输出结果：

```
队列中的元素数量: 3
队首元素: 10
队尾元素: 30
移除队首元素后，队首元素: 20
队列中的元素数量: 2
```

常用成员函数：

- `push(const value_type& val)`：将元素 `val` 添加到队列尾部。
- `pop()`：移除队列头部的元素。
- `front()`：返回对队列头部元素的引用。
- `back()`：返回对队列尾部元素的引用。
- `empty()`：检查队列是否为空。
- `size()`：返回队列中元素的数量。

注意事项： `std::queue` 不提供迭代器，因此无法像其他 STL 容器那样使用范围-based for 循环或迭代器进行遍历。要遍历队列中的元素，需要在移除元素的同时访问它们，例如：

- ```
  while (!q.empty()) {
      std::cout << q.front() << " ";
      q.pop();
  }
  std::cout << std::endl;
  ```

priority_queue

`std::priority_queue` 的基本用法：

1. 包含头文件：

   ```
   #include <queue>
   ```

2. 定义优先队列：

   ```
   std::priority_queue<Type> pq;
   ```

3. 常用操作：

   - 插入元素：

     ```
     pq.push(value);
     ```

     将元素 `value` 插入到优先队列中。

   - 访问队头元素：

     ```
     Type topElement = pq.top();
     ```

     获取队头元素的值。注意，`top()` 返回的是对队头元素的引用，直接修改可能影响队列的结构。

   - 移除队头元素：
     ```

```
pq.pop();
```

移除优先队列中的队头元素，即优先级最高的元素。

- **检查队列是否为空：**

```
bool isEmpty = pq.empty();
```

如果队列为空，`empty()` 返回 `true`，否则返回 `false`。

- **获取队列大小：**

```
size_t size = pq.size();
```

返回队列中元素的数量。

自定义优先级：

默认情况下，`std::priority_queue` 是一个最大堆，即优先级高的元素在队头。如果需要实现最小堆（即优先级低的元素在队头），可以通过自定义比较函数来实现：

```
#include <iostream>
#include <queue>
#include <vector>
#include <functional> // std::greater

int main() {
    // 定义最小堆的比较函数
    auto compare = [](int left, int right) {
        return left > right; // 返回 true 表示左边元素优先级低于右边元素
    };

    // 创建优先队列，使用 std::vector 作为底层容器，std::greater 实现最小堆
    std::priority_queue<int, std::vector<int>, decltype(compare)>
minHeap(compare);

    // 插入元素
    minHeap.push(10);
    minHeap.push(30);
    minHeap.push(20);

    // 访问并移除元素
    while (!minHeap.empty()) {
        std::cout << minHeap.top() << ' '; // 输出当前最小元素
```

```
            minHeap.pop(); // 移除当前最小元素
        }

        return 0;
    }
```

注意事项：

- **底层容器：** `std::priority_queue` 默认使用 `std::vector` 作为底层容器，也可以通过模板参数指定其他容器类型，如 `std::deque`。
- **不支持迭代器：** 由于其内部结构是堆，`std::priority_queue` 不提供迭代器，因此无法像其他 STL 容器那样进行范围遍历。
- **元素唯一性：** `std::priority_queue` 允许元素重复，如果需要元素唯一性，可以在插入前进行检查。

通过使用 `std::priority_queue`，您可以在 C++ 中方便地实现优先级队列，广泛应用于任务调度、图算法等需要按照优先级处理元素的场景。

c++好用的函数

字符判断函数

C++ 标准库中提供了一些非常实用的函数，用来判断字符是否是字母、数字、空格等等。这些函数都在头文件 `<cctype>`（C 风格）中。

📚 常用字符判断函数（来自 `<cctype>`）：

函数名	作用说明
`isalpha(c)`	判断是否为字母（a~z 或 A~Z）
`isdigit(c)`	判断是否为数字（0~9）
`isalnum(c)`	判断是否为字母或数字（字母+数字）
`islower(c)`	判断是否为小写字母
`isupper(c)`	判断是否为大写字母
`isspace(c)`	判断是否为空白字符（空格、换行等）

函数名	作用说明
`ispunct(c)`	判断是否为标点符号
`isxdigit(c)`	判断是否为十六进制字符（0~9, a~f, A~F）
`isprint(c)`	判断是否为可打印字符
`iscntrl(c)`	判断是否为控制字符

✅ 示例代码：

```cpp
#include <iostream>
#include <cctype>  // 包含字符处理函数

int main() {
    char ch = 'A';

    if (isalpha(ch)) {
        std::cout << ch << " 是字母" << std::endl;
    }

    if (isdigit(ch)) {
        std::cout << ch << " 是数字" << std::endl;
    }

    if (isalnum(ch)) {
        std::cout << ch << " 是字母或数字" << std::endl;
    }

    if (isupper(ch)) {
        std::cout << ch << " 是大写字母" << std::endl;
    }

    ch = ' ';
    if (isspace(ch)) {
        std::cout << "这是一个空格" << std::endl;
    }

    return 0;
}
```

字符串转数字

在 C++ 中，可以使用多种方法将字符串转换为数字，主要包括 `std::stoi`、`std::atof`、`std::atoi` 等函数。以下是这些函数的详细介绍和使用示例：

1. 使用 `std::stoi` 函数

`std::stoi`（string to integer）是 C++11 引入的标准库函数，用于将 `std::string` 转换为整数。其函数签名为：

```
int stoi(const std::string& str, std::size_t* pos = nullptr, int base = 10);
```

- `str`：要转换的字符串。
- `pos`：指向 `size_t` 的指针，用于存储转换结束的位置索引，可选参数，默认为 `nullptr`。
- `base`：转换所使用的进制，默认为 10。

示例：

```
#include <iostream>
#include <string>

int main() {
    std::string str = "12345";
    try {
        int num = std::stoi(str);
        std::cout << "转换后的数字为：" << num << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cerr << "无效的输入：" << e.what() << std::endl;
    } catch (const std::out_of_range& e) {
        std::cerr << "输入超出范围：" << e.what() << std::endl;
    }
    return 0;
}
```

注意：

- 如果输入字符串不是有效的数字表示，`std::stoi` 会抛出 `std::invalid_argument` 异常。
- 如果转换结果超出了 `int` 类型的表示范围，会抛出 `std::out_of_range` 异常。 □cite□turn0search6□

2. 使用 `std::atoi` 函数

`std::atoi`（ASCII to integer）是 C 标准库函数，可在 C++ 中使用，用于将 C 风格的字符串（即以空字符 `'\0'` 结尾的字符数组）转换为整数。其函数签名为：

```
int atoi(const char* str);
```

示例：

```
#include <iostream>
#include <cstdlib>

int main() {
    const char* str = "6789";
    int num = std::atoi(str);
    std::cout << "转换后的数字为: " << num << std::endl;
    return 0;
}
```

注意：

- `std::atoi` 不会进行错误检查，如果输入字符串不是有效的数字，返回值未定义。
- 不建议在现代 C++ 中使用 `std::atoi`，因为它缺乏错误处理机制，建议使用 `std::stoi` 替代。 cite turn0search4

3. 使用 `std::atof` 函数

`std::atof`（ASCII to float）用于将 C 风格的字符串转换为浮点数。其函数签名为：

```
double atof(const char* str);
```

示例：

```
#include <iostream>
#include <cstdlib>

int main() {
    const char* str = "3.14159";
    double num = std::atof(str);
    std::cout << "转换后的数字为: " << num << std::endl;
    return 0;
}
```

注意：

- 与 `std::atoi` 类似，`std::atof` 也缺乏错误处理机制，建议在现代 C++ 中使用 `std::stod`（string to double）替代。 □cite□turn0search9□

4. 使用 `std::stringstream` 类

`std::stringstream` 是 C++ 标准库中的类，可用于将字符串转换为各种数值类型。

示例：

```
#include <iostream>
#include <sstream>
#include <string>

int main() {
    std::string str = "42";
    int num;
    std::stringstream ss(str);
    ss >> num;
    if (ss.fail()) {
        std::cerr << "转换失败" << std::endl;
    } else {
        std::cout << "转换后的数字为：" << num << std::endl;
    }
    return 0;
}
```

注意：

- 使用 `std::stringstream` 进行转换时，需要检查流的状态以确保转换成功。

5.toupper && tolower

```
#include <iostream>
#include <cctype>
#include <string>

int main() {
    std::string str = "Hello, World!";

    // 转换为大写
    for (char& c : str) {
        c = std::toupper(static_cast<unsigned char>(c));
    }
    std::cout << "大写转换结果: " << str << std::endl;

    // 转换为小写
```

```
    for (char& c : str) {
        c = std::tolower(static_cast<unsigned char>(c));
    }
    std::cout << "小写转换结果: " << str << std::endl;

    return 0;
}
```

rand

在C++中生成随机数，主要有两种方法：使用传统的 `rand()`函数和C++11引入的 `<random>`库。

1. 使用 `rand()`和 `srand()`函数：

`rand()`函数返回一个范围在 `0`到 `RAND_MAX`之间的伪随机整数。`RAND_MAX`的值通常为 `32767`，定义在 `<cstdlib>`头文件中。为了使每次程序运行时生成不同的随机数序列，需要使用 `srand()`函数设置随机数种子，通常以当前时间作为种子。

示例代码：

```
#include <iostream>
#include <cstdlib>
#include <ctime>

int main() {
    std::srand(static_cast<unsigned int>(std::time(nullptr))); // 设置随机数种
子
    for (int i = 0; i < 5; ++i) {
        int random_number = std::rand();
        std::cout << random_number << std::endl;
    }
    return 0;
}
```

上述代码中，`std::srand()`使用当前时间初始化随机数种子，`std::rand()`生成随机数。需要注意的是，`rand()`生成的随机数质量有限，且不同平台上的实现可能有所差异。此外，`rand()`的随机性较弱，可能不适用于对随机性要求较高的场景。

2. 使用C++11的 `<random>`库：

C++11引入了 `<random>`头文件，提供了更为强大和灵活的随机数生成功能，包括随机数引擎和分布。推荐使用 `std::mt19937`（梅森旋转算法）作为随机数引擎，配合

`std::uniform_int_distribution`或 `std::uniform_real_distribution`等分布来生成随机数。

示例代码：

```
#include <iostream>
#include <random>

int main() {
    std::random_device rd;  // 用于获得一个真正的随机数种子
    std::mt19937 gen(rd()); // 以该种子初始化梅森旋转算法引擎

    // 生成范围在[0, 100]之间的整数随机数
    std::uniform_int_distribution<> distrib(0, 100);

    for (int i = 0; i < 5; ++i) {
        int random_number = distrib(gen);
        std::cout << random_number << std::endl;
    }
    return 0;
}
```

在上述代码中，`std::random_device`用于生成一个随机种子，`std::mt19937`是随机数引擎，`std::uniform_int_distribution<>`定义了一个均匀分布，用于生成指定范围内的整数随机数。这种方法生成的随机数质量更高，适用于对随机性要求较高的应用场景。

综上所述，虽然传统的 `rand()`和 `srand()`方法简单易用，但在需要高质量随机数的情况下，建议使用C++11提供的 `<random>`库。它提供了更丰富的功能和更好的随机性，适用于更广泛的应用场景。

c++算法

在C++中，`binary_search()`、`reverse()`和 `count()`是常用的算法函数，分别用于在已排序范围内查找元素、反转元素顺序以及统计元素出现次数。

1. `binary_search()`

- **功能：** 在已排序的范围内检查是否存在指定值的元素。
- **头文件：** `<algorithm>`
- **函数原型：**

```
  template< class ForwardIterator, class T >
  bool binary_search( ForwardIterator first, ForwardIterator last, const T&
value );
```

- **参数：**
  - `first`：指向范围起始位置的迭代器。
  - `last`：指向范围结束位置的迭代器。
  - `value`：要查找的值。
- **返回值：** 如果范围内存在等于 `value`的元素，返回 `true`；否则，返回 `false`。
- **注意：** `binary_search()`只能用于已排序的范围。

示例：

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 3, 5, 7, 9};
    int value = 5;
    if (std::binary_search(numbers.begin(), numbers.end(), value)) {
        std::cout << value << " found in the range." << std::endl;
    } else {
        std::cout << value << " not found in the range." << std::endl;
    }
    return 0;
}
```

2. `reverse()`

- **功能：** 反转指定范围内元素的顺序。
- **头文件：** `<algorithm>`
- **函数原型：**

```
  template< class BidirectionalIterator >
  void reverse( BidirectionalIterator first, BidirectionalIterator last );
```

- **参数：**
  - `first`：指向范围起始位置的双向迭代器。
  - `last`：指向范围结束位置的双向迭代器。
- **返回值：** 无。

示例：

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    std::reverse(numbers.begin(), numbers.end());
    for (const auto& num : numbers) {
        std::cout << num << ' ';
    }
    return 0;
}
```

3. `count()`

- **功能：** 统计指定范围内某个元素出现的次数。
- **头文件：** `<algorithm>`
- **函数原型：**

```
template< class InputIterator, class T >
typename std::iterator_traits<InputIterator>::difference_type
count( InputIterator first, InputIterator last, const T& value );
```

- **参数：**
  - `first`：指向范围起始位置的输入迭代器。
  - `last`：指向范围结束位置的输入迭代器。
  - `value`：要统计的值。
- **返回值：** 指定元素在范围内出现的次数。

示例：

```
#include <iostream>
#include <algorithm>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 2, 3, 4, 2, 5};
    int value = 2;
    int occurrences = std::count(numbers.begin(), numbers.end(), value);
    std::cout << value << " appears " << occurrences << " times in the
range." << std::endl;
```

```
    return 0;
}
```

这些算法函数在处理容器数据时非常有用，能够简洁高效地完成常见操作。

python迭代器用法

set

在 Python 中，`set` 是一个非常常用的数据类型，它表示一个无序的集合，且集合中的元素是 **唯一** 的，不能重复。`set` 是 Python 中的一种内建数据结构，它与数学中的集合类似，可以进行集合的基本操作，如交集、并集、差集等。

基本用法

1. 创建集合

你可以使用 `{}` 或 `set()` 来创建一个集合。

```
# 使用花括号创建集合
my_set = {1, 2, 3, 4, 5}
print(my_set)  # 输出: {1, 2, 3, 4, 5}

# 使用 set() 函数创建集合
my_set2 = set([1, 2, 2, 3, 4])  # 重复的元素会被去除
print(my_set2)  # 输出: {1, 2, 3, 4}
```

2. 添加元素

使用 `add()` 方法可以向集合中添加单个元素。如果该元素已经存在，集合不会发生变化。

```
my_set.add(6)
print(my_set)  # 输出: {1, 2, 3, 4, 5, 6}
```

3. 删除元素

使用 `remove()` 或 `discard()` 方法删除元素。`remove()` 在删除元素时，如果该元素不存在，会抛出 `KeyError` 异常；`discard()` 如果元素不存在不会报错。

```
my_set.remove(3)  # 删除元素 3
print(my_set)  # 输出: {1, 2, 4, 5, 6}

my_set.discard(7)  # 该元素不存在，不会报错
print(my_set)  # 输出: {1, 2, 4, 5, 6}
```

4. 集合的基本操作

- **交集** (`&` 或 `intersection()`)
- **并集** (`|` 或 `union()`)
- **差集** (`-` 或 `difference()`)
- **对称差集** (`^` 或 `symmetric_difference()`)

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# 交集
intersection = set1 & set2
print("Intersection:", intersection)  # 输出: {3, 4}

# 并集
union = set1 | set2
print("Union:", union)  # 输出: {1, 2, 3, 4, 5, 6}

# 差集
difference = set1 - set2
print("Difference (set1 - set2):", difference)  # 输出: {1, 2}

# 对称差集
symmetric_difference = set1 ^ set2
print("Symmetric Difference:", symmetric_difference)  # 输出: {1, 2, 5, 6}
```

5. 检查元素是否在集合中

你可以使用 `in` 关键字来检查元素是否在集合中。

```
print(3 in my_set)  # 输出: True
print(7 in my_set)  # 输出: False
```

6. 集合的其他常用方法

- `len()`：获取集合的大小。
- `clear()`：清空集合。
- `copy()`：复制集合。

```
# 获取集合的大小
print(len(my_set))  # 输出: 5

# 清空集合
my_set.clear()
print(my_set)  # 输出: set()

# 复制集合
set_copy = my_set2.copy()
print(set_copy)  # 输出: {1, 2, 3, 4}
```

示例：集合的常见操作

```
# 创建集合
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# 交集
print("Intersection:", set1 & set2)  # {4, 5}

# 并集
print("Union:", set1 | set2)  # {1, 2, 3, 4, 5, 6, 7, 8}

# 差集
print("Difference:", set1 - set2)  # {1, 2, 3}

# 对称差集
print("Symmetric Difference:", set1 ^ set2)  # {1, 2, 3, 6, 7, 8}
```

注意：

- 集合中的元素是 **无序的**，意味着你不能通过索引访问集合中的元素。
- 集合中的元素是 **唯一的**，重复元素会被自动去除。
- 集合支持快速的 **查找、插入、删除** 操作，通常复杂度为 `O(1)`。

deque

在Python中，`deque`（双端队列）是`collections`模块提供的一个类，支持在序列的两端高效地添加和删除元素。与列表（`list`）相比，`deque`在两端操作时具有更优的性能，特别适用于需要频繁进行首尾插入和删除操作的场景。

1. 导入 `deque`类：

```
from collections import deque
```

2. 创建 `deque`对象：

- **无固定长度的 `deque`：**

```
d = deque()
```

这将创建一个空的双端队列，没有长度限制。

- **具有固定长度的 `deque`：**

```
d = deque(maxlen=3)
```

此队列最多容纳3个元素。当队列已满且有新元素加入时，最旧的元素会被自动移除。
□cite□turn0search9□

3. 向 `deque`添加元素：

- **在右侧（尾部）添加单个元素：**

```
d.append('a')
```

将元素 `'a'`添加到队列的右端。

- **在左侧（头部）添加单个元素：**

```
d.appendleft('b')
```

将元素 `'b'`添加到队列的左端。

- **在右侧添加多个元素：**

```
d.extend(['c', 'd', 'e'])
```

依次将 `'c'`、`'d'`和 `'e'`添加到队列的右端。

- **在左侧添加多个元素：**

```
d.extendleft(['x', 'y', 'z'])
```

依次将 `'x'`、`'y'`和 `'z'`添加到队列的左端。注意，`extendleft`会依照可迭代对象的顺序依次将元素添加到左端，因此最终队列的顺序可能与预期不同。

4. 从 `deque`中删除元素：

- **移除右侧（尾部）元素：**

```
right_elem = d.pop()
```

移除并返回队列右端的元素。如果队列为空，调用此方法会引发 `IndexError`异常。

- **移除左侧（头部）元素：**

```
left_elem = d.popleft()
```

移除并返回队列左端的元素。如果队列为空，调用此方法会引发 `IndexError`异常。

- **删除指定元素：**

```
d.remove('a')
```

移除队列中第一个值为 `'a'`的元素。如果该元素不存在，会引发 `ValueError`异常。

5. 其他常用操作：

- **旋转队列：**

```
d.rotate(2)
```

将队列向右旋转2个位置。若参数为负数，则向左旋转相应的步数。

- **获取队列长度：**

```
length = len(d)
```

返回队列中的元素数量。

- **访问特定位置的元素：**

```
elem = d[1]
```

获取索引为1的元素。需要注意的是，直接访问 deque中的元素在性能上可能不如列表高效。

- **限制队列最大长度：**

```
d = deque(maxlen=5)
```

创建一个最大长度为5的 deque。当队列已满且有新元素加入时，最旧的元素会被自动移除。

6. 示例代码：

```
from collections import deque

# 创建一个无固定长度的双端队列
d = deque()

# 从右侧添加元素
d.append(1)
d.append(2)

# 从左侧添加元素
d.appendleft(3)

print(d)  # 输出: deque([3, 1, 2])
```

```
# 从右侧移除元素
right = d.pop()
print(right)  # 输出: 2

# 从左侧移除元素
left = d.popleft()
print(left)  # 输出: 3

print(d)  # 输出: deque([1])
```

deque在需要高效地对序列两端进行操作的场景下非常有用，如实现队列、栈、滑动窗口等功能。与列表相比，deque在两端插入和删除操作上的性能更优，适合处理需要频繁进行此类操作的数据结构。

defaultdict

在Python中，defaultdict是 collections模块提供的一个字典子类，功能与普通字典类似，但具有为不存在的键提供默认值的特性。使用 defaultdict可以避免在访问不存在的键时引发 KeyError异常。

1. 导入 defaultdict：

```
from collections import defaultdict
```

2. 创建 defaultdict对象： defaultdict需要一个工厂函数作为参数，该函数用于生成默认值。例如：

- **默认值为整数0：**

```
d = defaultdict(int)
```

当访问不存在的键时，int()返回0。

- **默认值为空列表：**

```
d = defaultdict(list)
```

当访问不存在的键时，list()返回一个空列表。

- **默认值为空字符串：**

```
d = defaultdict(str)
```

当访问不存在的键时，`str()`返回一个空字符串。

3. 使用 `defaultdict`： 与普通字典相似，可以使用 `[]`操作符添加和访问元素。

```
d = defaultdict(int)
d['a'] += 1  # 相当于 d['a'] = d['a'] + 1
print(d['a'])  # 输出: 1
print(d['b'])  # 输出: 0, 因为默认值为int(), 即0
```

4. 示例：统计元素出现次数 使用 `defaultdict`统计字符串中每个字符的出现次数：

```
from collections import defaultdict

s = 'mississippi'
d = defaultdict(int)
for char in s:
    d[char] += 1

print(d)
# 输出: defaultdict(<class 'int'>, {'m': 1, 'i': 4, 's': 4, 'p': 2})
```

在此示例中，`defaultdict(int)`创建了一个默认值为0的字典，遍历字符串 `s`，统计每个字符的出现次数。

5. 示例：分组数据 使用 `defaultdict`将一组数据按键分组：

```
from collections import defaultdict

data = [('A', 1), ('B', 2), ('A', 3), ('B', 4), ('A', 5)]
d = defaultdict(list)
for key, value in data:
    d[key].append(value)

print(d)
# 输出: defaultdict(<class 'list'>, {'A': [1, 3, 5], 'B': [2, 4]})
```

在此示例中，`defaultdict(list)`创建了一个默认值为空列表的字典，将相同键的值分组到一个列表中。

6. 注意事项：

- `defaultdict`的默认值是通过工厂函数动态生成的，而非预先定义的静态值。
- 如果访问不存在的键，`defaultdict`会调用工厂函数生成默认值，而普通字典则会引发 `KeyError`。
- 使用 `defaultdict`时，如果需要获取默认值类型，可以使用 `type(d.default_factory)`。

通过使用 `defaultdict`，可以简化代码，避免手动检查键是否存在，提高代码的可读性和效率。

优先队列

在 Python 中，优先队列（Priority Queue）是一种特殊的队列，它允许在队列中插入具有不同优先级的元素，元素会根据优先级进行排序，并且总是优先取出优先级最高的元素。Python 标准库提供了一个模块 `queue` 和 `heapq` 来实现优先队列。

1. 使用 `queue.PriorityQueue`

`queue.PriorityQueue` 是 `queue` 模块中的一个类，它实现了一个线程安全的优先队列。该队列按照元素的优先级进行排序，优先级最低的元素会最先被取出。它是基于最小堆（min-heap）实现的。

示例：

```
import queue

# 创建一个优先队列
pq = queue.PriorityQueue()

# 插入元素（优先级，元素）
pq.put((3, 'apple'))
pq.put((1, 'banana'))
pq.put((2, 'cherry'))

# 取出优先级最高的元素（优先级最低的值会优先被取出）
print(pq.get())  # 输出: (1, 'banana')
print(pq.get())  # 输出: (2, 'cherry')
print(pq.get())  # 输出: (3, 'apple')
```

说明：

- `put()` 用来向队列中插入元素。优先级和元素一起插入。
- `get()` 用来从队列中取出优先级最高（即值最小）的元素。

2. 使用 `heapq` 模块

`heapq` 模块提供了一种基于堆（heap）实现的优先队列。堆是一种完全二叉树，最小堆的父节点总是小于其子节点，这使得最小元素可以在 O(1) 的时间复杂度内访问，并且插入和删除元素的操作是 O(log n) 时间复杂度。

`heapq` 本身并没有专门的优先队列类，它提供了对列表进行堆操作的函数。为了实现优先队列，我们通常用一个元组（优先级，元素）来存储数据，因为元组会首先比较优先级。

示例：

```
import heapq

# 创建一个空的堆（即列表）
pq = []

# 插入元素（优先级，元素）
heapq.heappush(pq, (3, 'apple'))
heapq.heappush(pq, (1, 'banana'))
heapq.heappush(pq, (2, 'cherry'))

# 取出优先级最高的元素（优先级最低的元素）
print(heapq.heappop(pq))  # 输出: (1, 'banana')
print(heapq.heappop(pq))  # 输出: (2, 'cherry')
print(heapq.heappop(pq))  # 输出: (3, 'apple')
```

说明：

- `heappush()` 用来向堆中插入元素，并保持堆的性质。
- `heappop()` 用来从堆中弹出最小元素（即优先级最高的元素）。

3. `heapq` 其他函数

- **`heapify()`**：将一个列表转化为堆。

```
data = [3, 1, 4, 1, 5, 9, 2, 6, 5]
heapq.heapify(data)
```

```
print(data)  # 输出: [1, 1, 2, 3, 5, 9, 4, 6, 5]
```

- **heappushpop()**：将元素插入堆中，并弹出堆中最小的元素。

```
heapq.heappushpop(pq, (0, 'orange'))  # 插入新元素并弹出最小元素
```

random

```
import random
print(random.random())  # 输出类似 0.37444887175646646
import random
print(random.randint(1, 5))  # 输出 1 至 5 之间的随机整数，例如 3
import random
print(random.randrange(0, 101, 2))  # 输出 0 至 100 之间的随机偶数，例如 40
import random
items = ['apple', 'banana', 'cherry']
print(random.choice(items))  # 输出 'apple'、'banana' 或 'cherry'
```