【华为OD机考 统一考试机试C卷】根据IP查找城市（C++ Java JavaScript Python）

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷

2023年11月份，华为官方已经将 华为OD机考：OD统一考试（A卷 / B卷）切换到 OD统一考试（C卷）和 OD统一考试（D卷） 。根据考友反馈：目前抽到的试卷为B卷或C卷/D卷，其中C卷居多 ，按照之前的经验C卷D卷部分考题会复用A卷/B卷题，博主正积极从考过的同学收集C卷和D卷真题，可以查看下面的真题目录。

真题目录：华为OD机考机试 真题目录（C卷 + D卷 + B卷 + A卷） + 考点说明
专栏：2023华为OD机试(B卷+C卷+D卷)（C++JavaJSPy）
华为OD面试真题精选：华为OD面试真题精选
在线OJ：点击立即刷题，模拟真实机考环境 华为OD机考B卷C卷华为OD机考华为OD机考B卷华为OD机试B卷华为OD机试C卷华为OD机考C卷华为OD机考D卷题目华为OD机考C卷/D卷答案华为OD机考C卷/D卷解析华为OD机考C卷和D卷真题华为OD机考C卷和D卷题解

题目描述

某业务需要根据终端的IP地址获取该终端归属的城市，可以根据公开的IP地址池信息查询归属城市。

地址池格式如下：

```
城市名=起始IP,结束IP
```

起始和结束地址按照英文逗号分隔，多个地址段采用英文分号分隔。比如：

```
City1=1.1.1.1,1.1.1.2;City1=1.1.1.11,1.1.1.16;City2=3.3.3.3,4.4.4.4;City3=2.2.2.2,6.6.6.6
```

一个城市可以有多个IP段，比如City1有2个IP段。

城市间也可能存在包含关系，如City3的IP段包含City2的IP段范围。

现在要根据输入的IP列表，返回最佳匹配的城市列表。

注：最佳匹配即包含待查询IP且长度最小的IP段，比如例子中3.4.4.4最佳匹配是City2=3.3.3.3,4.4.4.4，5.5.5.5的最佳匹配是City3=2.2.2.2,6.6.6.6

输入描述

输入共2行。

第一行为城市的IP段列表，多个IP段采用英文分号 ';' 分隔，IP段列表最大不超过500000。城市名称只包含英文字母、数字和下划线。最多不超过100000个。IP段包含关系可能有多层，但不超过100层。

第二行为查询的IP列表，多个IP采用英文逗号 ',' 分隔，最多不超过10000条。

输出描述

最佳匹配的城市名列表，采用英文逗号 ',' 分隔，城市列表长度应该跟查询的IP列表长度一致。

备注

- 无论是否查到匹配正常都要输出分隔符。举例：假如输入IP列表为IPa,IPb，两个IP均未有匹配城市，此时输出为","，即只有一个逗号分隔符，两个城市均为空；
- 可以假定用例中的所有输入均合法，IP地址均为合法的ipv4地址，满足 $(1_{255).(0}255).(0_{255).(0}255)$ 的格式，且可以假定用例中不会出现组播和广播地址；

解题思路

1. 解析输入的IP地址池字符串，将其转换为每个城市对应的IP范围列表的映射。
2. 解析输入的查询IP字符串，将其分割成单独的IP地址。
3. 对于每个查询IP地址，将其转换为长整型数值以便比较。
4. 遍历每个城市的IP范围列表，检查查询IP是否落在某个范围内。
5. 如果查询IP落在某个范围内，计算该范围的大小，并与当前已知的最小范围进行比较。
6. 如果当前范围更小，更新最佳匹配城市和最小范围。
7. 将最佳匹配城市添加到结果字符串中。

将IP地址转换成长整型数值（通常是64位的整数）是为了方便比较和计算。IP地址通常以点分十进制格式呈现，如 `192.168.1.1` ，这种格式不便于直接进行数学运算或比较。转换成长整型数值后，IP地址就变成了一个数值，可以轻松地进行比较和范围检查。

例如，对于IP地址 `192.168.1.1`：

- 将每个十进制块转换为二进制形式： `11000000.10101000.00000001.00000001`
- 将这些二进制块拼接成一个32位的二进制数： `11000000101010000000000100000001`
- 将这个二进制数转换为长整型数值： `3232235777`

这样，IP地址就可以像普通的整数一样参与计算和比较操作，使得IP范围匹配变得简单高效。

C++

```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <climits>

using namespace std;

// IP范围结构体
struct IpRange {
    unsigned long start; // 起始IP的长整型数
    unsigned long end;   // 结束IP的长整型数
    IpRange(unsigned long s, unsigned long e) : start(s), end(e) {} // 构造函数
};

// 将IP地址转换为长整型数的函数
unsigned long ipToLong(const string& ip) {
    unsigned long result = 0;
    istringstream iss(ip);
    string octet;
    while (getline(iss, octet, '.')) {
        result = (result << 8) | (stoul(octet)); // 将每个八位字节转换为无符号长整数，并组合为完整的IP数值
    }
    return result;
}

// 解析IP地址池的函数
unordered_map<string, vector<IpRange>> parseIpPool(const string& ipPool) {
    unordered_map<string, vector<IpRange>> cityIpRanges; // 存储城市与IP范围映射的哈希表
    istringstream iss(ipPool);
    string cityRange;
    while (getline(iss, cityRange, ';')) { // 使用分号分割城市和IP范围
        auto pos = cityRange.find('='); // 找到城市名和IP范围分隔符的位置
        string city = cityRange.substr(0, pos); // 获取城市名
```

```
        string range = cityRange.substr(pos + 1); // 获取IP范围
        auto commaPos = range.find(','); // 找到起始IP和结束IP分隔符的位置
        unsigned long start = ipToLong(range.substr(0, commaPos)); // 获取起始IP的数值
        unsigned long end = ipToLong(range.substr(commaPos + 1)); // 获取结束IP的数值
        cityIpRanges[city].emplace_back(start, end); // 将IP范围添加到对应城市的映射中
    }
    return cityIpRanges;
}

// 匹配城市的函数
string matchCities(const string& ipPool, const string& queryIPs) {
    auto cityIpRanges = parseIpPool(ipPool); // 解析IP池
    istringstream iss(queryIPs);
    string ip;
    ostringstream result; // 用于拼接结果的字符串流
    while (getline(iss, ip, ',')) { // 使用逗号分割查询的IP地址
        unsigned long ipNum = ipToLong(ip); // 将IP地址转换为数值
        string bestMatchCity; // 最佳匹配城市
        unsigned long smallestRange = ULONG_MAX; // 最小IP范围，初始化为最大无符号长整数
        for (const auto& entry : cityIpRanges) { // 遍历城市IP范围映射
            for (const auto& range : entry.second) { // 遍历每个城市的IP范围
                if (ipNum >= range.start && ipNum <= range.end) { // 判断IP是否在当前范围内
                    unsigned long rangeSize = range.end - range.start; // 计算当前范围的大小
                    if (rangeSize < smallestRange) { // 如果当前范围小于已知的最小范围
                        bestMatchCity = entry.first; // 更新最佳匹配城市
                        smallestRange = rangeSize; // 更新最小范围
                    }
                }
            }
        }
        result << bestMatchCity << ","; // 将最佳匹配城市添加到结果字符串流中
    }
    string resultStr = result.str(); // 转换结果字符串流为字符串
    return resultStr.empty() ? resultStr : resultStr.substr(0, resultStr.size() - 1); // 去除最后一个逗号
}

int main() {
    string ipPool;
    getline(cin, ipPool); // 读取IP地址池
    string queryIPs;
```

```
    getline(cin, queryIPs); // 读取查询IP列表

    // 输出匹配的城市列表
    cout << matchCities(ipPool, queryIPs) << endl;

    return 0;
}
```

Java

```
import java.util.*;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String ipPool = scanner.nextLine(); // 读取IP地址池
        String queryIPs = scanner.nextLine(); // 读取查询IP列表

        // 输出匹配的城市列表
        System.out.println(matchCities(ipPool, queryIPs));
    }

    // 匹配城市的方法
    private static String matchCities(String ipPool, String queryIPs) {
        // 解析IP地址池，将其转换为城市与IP范围列表的映射
        Map<String, List<IpRange>> cityIpRanges = parseIpPool(ipPool);
        // 将查询IP列表按逗号分隔
        String[] ipsToQuery = queryIPs.split(",");
        StringBuilder result = new StringBuilder(); // 用于构建结果字符串

        // 遍历每个查询IP
        for (String ip : ipsToQuery) {
            long ipNum = ipToLong(ip); // 将IP地址转换为长整型数
            String bestMatchCity = ""; // 最佳匹配城市
            long smallestRange = Long.MAX_VALUE; // 最小的IP范围

            // 遍历每个城市的IP范围
            for (Map.Entry<String, List<IpRange>> entry : cityIpRanges.entrySet()) {
                for (IpRange range : entry.getValue()) {
```

```
                // 如果IP在当前范围内
                if (ipNum >= range.start && ipNum <= range.end) {
                    long rangeSize = range.end - range.start; // 计算当前IP范围的大小
                    // 如果当前范围更小，则更新最佳匹配城市和最小范围
                    if (rangeSize < smallestRange) {
                        bestMatchCity = entry.getKey();
                        smallestRange = rangeSize;
                    }
                }
            }
        }

        // 将最佳匹配城市添加到结果中
        result.append(bestMatchCity).append(",");
    }

    // 返回结果字符串，去除最后一个逗号
    return result.length() > 0 ? result.substring(0, result.length() - 1) : result.toString();
}

// 解析IP地址池的方法
private static Map<String, List<IpRange>> parseIpPool(String ipPool) {
    Map<String, List<IpRange>> cityIpRanges = new HashMap<>();
    String[] cityRanges = ipPool.split(";"); // 按分号分隔城市IP范围

    // 遍历每个城市的IP范围
    for (String cityRange : cityRanges) {
        String[] parts = cityRange.split("="); // 按等号分隔城市名和IP范围
        String city = parts[0];
        String[] ranges = parts[1].split(","); // 按逗号分隔起始和结束IP

        long start = ipToLong(ranges[0]); // 将起始IP转换为长整型数
        long end = ipToLong(ranges[1]); // 将结束IP转换为长整型数

        // 将城市和对应的IP范围添加到映射中
        cityIpRanges.putIfAbsent(city, new ArrayList<>());
        cityIpRanges.get(city).add(new IpRange(start, end));
    }

    return cityIpRanges;
```

```
    }

    // 将IP地址转换为长整型数的方法
    private static long ipToLong(String ip) {
        String[] octets = ip.split("\\."); // 按点分隔IP地址
        long result = 0;
        for (String octet : octets) {
            result = result << 8; // 每个八位数左移8位
            result |= Integer.parseInt(octet); // 将当前八位数添加到结果中
        }
        return result;
    }

    // 内部类，表示IP范围
    static class IpRange {
        long start; // 起始IP的长整型数
        long end; // 结束IP的长整型数

        // 构造函数
        IpRange(long start, long end) {
            this.start = start;
            this.end = end;
        }
    }
}
```

javaScript

```
const readline = require('readline');

// 创建readline接口实例
const rl = readline.createInterface({
  input: process.stdin, // 设置标准输入
  output: process.stdout // 设置标准输出
});

// 将IP地址转换为长整型数的函数
function ipToLong(ip) {
  // 使用split('.')将IP地址分割为四部分，然后使用reduce方法累加每部分
  return ip.split('.').reduce((acc, octet) => (acc << 8) + parseInt(octet, 10), 0);
```

```
}

// 解析IP地址池的函数
function parseIpPool(ipPool) {
  const cityIpRanges = {}; // 创建一个对象用于存储城市和对应的IP范围
  ipPool.split(';').forEach(cityRange => {
    const [city, range] = cityRange.split('='); // 分割字符串获取城市名和IP范围
    const [startIp, endIp] = range.split(','); // 分割字符串获取起始IP和结束IP
    const start = ipToLong(startIp); // 将起始IP转换为长整型数
    const end = ipToLong(endIp); // 将结束IP转换为长整型数
    if (!cityIpRanges[city]) cityIpRanges[city] = []; // 如果对象中不存在该城市，则初始化一个空数组
    cityIpRanges[city].push({ start, end }); // 将IP范围对象添加到对应城市的数组中
  });
  return cityIpRanges; // 返回解析后的城市IP范围对象
}

// 匹配城市的函数
function matchCities(ipPool, queryIPs) {
  const cityIpRanges = parseIpPool(ipPool); // 调用parseIpPool函数解析IP池
  return queryIPs.split(',').map(ip => {
    const ipNum = ipToLong(ip); // 将查询的IP地址转换为长整型数
    let bestMatchCity = ''; // 初始化最佳匹配城市为空字符串
    let smallestRange = Number.MAX_SAFE_INTEGER; // 初始化最小范围为最大安全整数
    for (const city in cityIpRanges) { // 遍历城市IP范围对象
      cityIpRanges[city].forEach(range => {
        if (ipNum >= range.start && ipNum <= range.end) { // 判断IP是否在当前范围内
          const rangeSize = range.end - range.start; // 计算当前范围的大小
          if (rangeSize < smallestRange) { // 如果当前范围小于已知的最小范围
            bestMatchCity = city; // 更新最佳匹配城市
            smallestRange = rangeSize; // 更新最小范围
          }
        }
      });
    }
    return bestMatchCity; // 返回最佳匹配城市
  }).join(','); // 将匹配结果数组转换为字符串，并用逗号分隔
}

// 读取输入
let lines = []; // 初始化一个数组用于存储输入的每一行
```

```
rl.on('line', line => {
  lines.push(line); // 将每一行输入添加到数组中
}).on('close', () => {
  // 当输入结束时，调用matchCities函数并输出匹配的城市列表
  console.log(matchCities(lines[0], lines[1]));
});
```

Python

```python
import sys

# 将IP地址转换为长整型数的函数
def ip_to_long(ip):
    # 分割IP地址为四个八位字节，并转换为整型
    octets = map(int, ip.split('.'))
    # 将四个八位字节转换为一个长整型数
    return sum(octet << (8 * (3 - index)) for index, octet in enumerate(octets))

# 解析IP地址池的函数
def parse_ip_pool(ip_pool):
    # 创建一个字典，用于存储城市及其对应的IP范围
    city_ip_ranges = {}
    # 分割IP池字符串，获取每个城市及其IP范围
    for city_range in ip_pool.split(';'):
        # 分割得到城市名和IP范围
        city, ranges = city_range.split('=')
        # 分割得到起始IP和结束IP
        start_ip, end_ip = ranges.split(',')
        # 将起始IP和结束IP转换为长整型数
        start = ip_to_long(start_ip)
        end = ip_to_long(end_ip)
        # 如果城市不在字典中，则添加城市并初始化IP范围列表
        if city not in city_ip_ranges:
            city_ip_ranges[city] = []
        # 将IP范围添加到城市的IP范围列表中
        city_ip_ranges[city].append((start, end))
    # 返回城市及其IP范围的字典
    return city_ip_ranges

# 匹配城市的函数
```

```python
def match_cities(ip_pool, query_ips):
    # 解析IP池，获取城市及其IP范围的字典
    city_ip_ranges = parse_ip_pool(ip_pool)
    # 创建一个列表，用于存储查询结果
    result = []
    # 分割查询IP列表，逐个处理
    for ip in query_ips.split(','):
        # 将IP地址转换为长整型数
        ip_num = ip_to_long(ip)
        # 初始化最佳匹配城市为空字符串
        best_match_city = ''
        # 初始化最小范围为系统最大整数
        smallest_range = sys.maxsize
        # 遍历所有城市及其IP范围
        for city, ranges in city_ip_ranges.items():
            # 遍历城市的IP范围
            for start, end in ranges:
                # 判断当前IP是否在范围内
                if start <= ip_num <= end:
                    # 计算当前范围的大小
                    range_size = end - start
                    # 如果当前范围小于已知的最小范围，则更新最佳匹配城市和最小范围
                    if range_size < smallest_range:
                        best_match_city = city
                        smallest_range = range_size
        # 将最佳匹配城市添加到结果列表中
        result.append(best_match_city)
    # 将结果列表转换为字符串，并用逗号分隔
    return ','.join(result)

# 读取输入
ip_pool = input()  # 读取IP地址池
query_ips = input()  # 读取查询IP列表

# 输出匹配的城市列表
print(match_cities(ip_pool, query_ips))
```

文章目录

华为OD机考:统一考试 C卷 + D卷 + B卷 +A卷
题目描述
输入描述
输出描述
解题思路
C++
Java
javaScript
Python