获取网络上任意计算机节点可通信地址的解决方案及其实现报告

课程：开源技术与应用

组员 1：李富麟　学号：2023302111204

组员 2：萨日娜　学号：2023302101082

组员 3：张喆　　学号：2023302101126

May 23, 2025

Abstract

本研究旨在设计并实现一款基于 Qt 框架的综合性网络扫描与分析工具，以应对复杂局域网环境下的设备管理与安全分析挑战。该工具通过自动检测网络接口、扫描可达计算机、采集设备信息（IP 地址、MAC 地址、主机名、设备类型），并利用 Qt Charts 实现网络拓扑与设备统计的可视化展示。系统核心功能包括：高效的网络设备发现（支持自动子网扫描与自定义 IP 范围扫描）、准确的主机可达性判定（基于 TCP 端口连接尝试）、灵活的端口扫描（支持默认与自定义端口列表），以及线程安全的扫描结果管理。此外，系统还提供设备分析（设备类型与制造商分布统计）、安全风险评估（高风险端口识别与报告生成）、扫描历史管理（会话存储、比较与 JSON 导出）以及用户友好的交互界面（支持明暗主题切换、实时进度显示）。实验结果表明，该工具在典型局域网环境中表现出优异的扫描性能、准确的设备信息展示、直观的网络拓扑可视化以及有效的安全风险评估能力，为网络管理员提供了高效、精准的网络管理与安全分析平台，提升了网络的整体安全性、可靠性与运维效率。

Contents

1 实验背景

在数字化时代，计算机网络已成为现代社会工作与生活不可或缺的核心基础设施。随着网络技术的飞速发展，局域网（Local Area Network, LAN）的规模与复杂性日益增长。一个典型的局域网通常涵盖了多种类型的设备，如路由器、服务器、个人计算机、移动设备、打印机以及各类智能设备。这些设备通过有线或无线方式互联，共同构建了日益复杂的网络生态系统。然而，网络的持续扩展与设备异构性的增加，为网络管理及安全维护带来了前所未有的挑战。网络管理员常需耗费大量时间进行手动故障排查，从逐一检查设备连接到分析网络配置，其过程不仅繁琐且效率低下，更严重制约了网络的稳定运行与用户体验的优化。此外，网络中的设备可能开放了各种端口以提供不同服务，部分非必要的开放端口可能演变为潜在安全脆弱点，易于被恶意行为者利用以渗透网络或发起攻击。传统的网络管理工具通常仅能提供有限的设备列表与简单的状态信息，难以直观地呈现网络整体拓扑结构及设备间的互联关系。这使得网络管理员难以全面洞察网络的整体布局与运行状况，进而难以实时洞察网络运行状态的异常波动或潜在性能瓶颈，从而对网络优化策略与管理决策的制定构成阻碍。鉴于此，研发一款功能强大、操作便捷且具备直观可视化能力的网络扫描工具，对于应对上述挑战具有至关重要的现实意义。本实验旨在充分利用 Qt 框架的技术优势，设计并实现一款综合性网络扫描器，旨在为网络管理员提供一个高效、精准的网络管理与安全分析平台，以有效应对复杂局域网环境下的各项管理挑战，进而提升网络的整体安全性、可靠性与运维效率。

2 实验目的

本实验的核心目的在于设计并实现一款基于 Qt 框架的综合性网络扫描工具，旨在显著提升局域网环境下的设备管理效能与网络安全分析的精准度。通过自动检测本地网络接口并扫描局域网内所有可达计算机，本工具致力于实现对网络设备信息（包括但不限于 IP 地址、MAC 地址、主机名及设备类型）的全面采集，并通过直观的网络拓扑图清晰展示设备间的互联状态与逻辑关系。此外，本实验致力于提供详细的设备分析与安全风险评估功能，包括设备类型与制造商分布统计、开放端口脆弱性分析以及自动化安全报告生成等高级功能，从而辅助网络管理员深入洞察网络结构，并及时识别与响应潜在的安全威胁。

3 系统设计

3.1 功能需求

为实现全面的局域网设备管理与网络安全分析能力，本基于 Qt 的网络扫描工具设计涵盖以下核心功能需求：

- 网络设备扫描：自动检测本地网络接口并扫描局域网内所有可达计算机。支持自定义 IP 地址范围扫描，允许用户根据特定需求指定 IP 地址的起始与结束范围进行精确扫描。同时，通过 TCP 端口连接尝试判定主机是否可达，以适应防火墙限制严格的网络环境，避免依赖系统 `ping` 命令。
- 设备信息展示：系统需全面采集并清晰展示各网络设备的详细信息，涵盖 IP 地址、主机名、MAC 地址、设备类型、制造商信息以及网络可达性状态等关键属性。同时，

工具应具备智能化的设备类型识别能力，能够区分路由器、服务器、个人计算机、移动终端等多样化设备，并以直观的网络拓扑图形式可视化呈现设备间的逻辑连接关系。

- 设备分析与统计：提供详细的设备分析功能，包括设备统计分析和图表展示。通过对扫描结果的分析，生成设备类型分布、制造商分布等统计图表，辅助用户清晰掌握网络中各类设备的构成比例。

- 安全风险评估：对网络中的设备进行安全风险分析，重点识别已知高风险开放端口（例如 21 (FTP)、22 (SSH)、23 (Telnet)、3389 (RDP) 等），并自动生成详尽的安全风险评估报告。该报告应包含设备数量统计、高风险开放端口列表、针对各设备的安全风险等级评估，以及具体的安全加固建议，旨在协助用户迅速定位并处置潜在安全隐患。

- 扫描历史管理：具备扫描历史记录功能，每次扫描任务完成后，其结果应能自动归档至扫描历史记录模块。用户不仅可以查阅过往的扫描结果，亦可通过集成的会话比较功能，对比不同时间节点的扫描数据，从而有效识别网络中新增或移除的设备，动态追踪网络拓扑的演变。此外，系统支持将扫描历史记录导出为 JSON 文件格式，以方便后续的数据查阅、分析或共享。

- 扫描任务配置：允许用户配置自定义端口扫描列表，根据实际需求指定要扫描的端口，提高扫描的针对性。可设置扫描超时时间，以平衡扫描速度和准确性。此外，还需支持计划任务扫描功能，用户可预设扫描任务的执行日期与时间，系统届时将自动执行扫描操作，并根据用户配置决定是否自动保存结果及在任务完成后发出通知。

- 用户界面与体验：支持明暗两种主题模式切换，用户可根据个人喜好和使用环境选择合适的主题模式。系统应实时展示扫描任务的执行进度与阶段性结果，确保用户能够即时掌握扫描工作的当前状态。

3.2 技术选型

- **Qt** 框架：利用 Qt 框架构建跨平台图形用户界面，充分利用其丰富的图形用户界面（GUI）组件库及成熟的信号与槽（signals and slots）机制，构建直观易用的用户交互界面。基于 Qt6 ，结合 QtCharts 模块进行数据可视化，从而将复杂的网络扫描数据转化为直观易懂的图表（例如，设备类型分布饼图、制造商分布柱状图、端口开放情况折线图等），辅助用户更为清晰、高效地理解当前网络状况及设备详细信息。

- **C++** 编程语言：本项目采用 C++17 标准进行开发，旨在充分利用其提供的多线程功能优化网络扫描过程中的并发处理能力，从而显著提升整体扫描性能。C++ 的面向对象特性使得代码结构更加清晰、模块化，便于项目的开发、维护和扩展，能够高效地处理网络扫描任务中的大量数据和复杂逻辑。

- **CMake** 构建工具：运用 CMake 构建项目，确保项目能够在不同操作系统的编译环境中顺利构建。CMake 能够根据项目的源代码和配置文件生成相应的构建文件，支持多种构建系统，为项目的跨平台编译、构建与部署提供了坚实保障，有效简化了开发流程，并提升了团队的开发效率。

3.3 数据结构设计

- **HostInfo 类**：用于存储单个设备的详细信息，包括 IP 地址、主机名、MAC 地址、厂商信息、可连接状态、开放端口列表等。该类对设备的各类属性进行封装，为设备信息的统一管理与高效操作提供了便利。例如，在扫描过程中，每个被扫描设备的信息都将存储于一个 `HostInfo` 对象中，这些对象随后被组织成列表结构，供后续的数据展示与深度分析环节使用。

- 扫描结果列表：采用 `QList<HostInfo>` 容器存储 `HostInfo` 对象，形成扫描结果列表。`QList` 作为 Qt 框架提供的动态数组容器，具备高效便捷的元素增、删、改、查操作能力，适用于存储并管理大规模设备信息集合。扫描过程中，每当发现一个设备，其对应的 `HostInfo` 对象即被添加至此列表。在用户界面展示扫描结果时，遍历该列表即可呈现设备信息。

- 端口扫描结果存储结构：在 `HostInfo` 类中，使用 `QMap<int, bool>` 容器存储设备的开放端口信息，其中键为端口号，值为布尔类型，表示该端口是否开放。`QMap` 作为 Qt 提供的键-值关联数组容器，能够根据端口号（键）快速检索其开放状态（值），极大地方便了端口状态的查询与动态更新。进行端口扫描时，每个设备的已扫描端口及其开放状态均存储于 `QMap` 中，以便后续进行开放端口分布统计和安全风险分析。

- 扫描历史记录结构：扫描历史记录采用 `QList` 容器存储每次扫描的 `HostInfo` 列表以及相关的扫描时间、描述信息等。此数据结构设计旨在方便地对历次扫描结果进行系统化管理与多维度比较。每次扫描任务完成后，其结果列表连同相关的元数据（如扫描时间、配置参数等）将作为一个独立的条目被添加至扫描历史记录列表中。查看历史记录时，可直接从该列表获取相应的扫描结果进行展示或比较。

- 线程池管理结构：使用 QtConcurrent 线程池管理并行扫描任务。该线程池通过 `QFuture` 对象来表征异步计算任务的结果，能够智能化地分配与管理线程资源，有效避免了因线程创建数量失控而导致的系统资源枯竭问题。在扫描执行阶段，针对每个目标 IP 地址的扫描任务均被提交至该线程池。线程池则依据当前系统的 CPU 核心数量，动态调整并行执行的扫描线程数目，以期达到最优的扫描效率。

4 系统组成

4.1 网络扫描模块

4.1.1 核心功能

网络设备发现

自动子网扫描：系统自动获取本地网络接口的 IPv4 地址信息，并对所属子网内的 IP 地址执行扫描操作（默认配置为每个子网扫描前 30 个 IP 地址，全局 IP 地址扫描上限设定为 100 个）。自定义 IP 范围扫描：允许用户手动设定 IP 地址的起始与结束边界，对指定范围内的 IP 地址进行扫描（单次扫描最多支持 50 个连续 IP 地址）。主机可达性判定：通过 TCP 端口连接尝试判定主机是否可达，避免依赖系统 `ping` 命令（适用于防火墙严格的环境）。伪 MAC 地址生成：为满足测试环境的需求，系统能够基于 IP 地址生成特定格式（`CA:FE:xx:xx:xx:xx`）的伪 MAC 地址（在实际应用场景中，应通过 ARP 协议获取真实的 MAC 地址）。

端口扫描

默认端口扫描：预定义常用端口列表（如 80、443、22、3389 等），覆盖 HTTP、SSH、RDP 等常见服务。自定义端口扫描：支持用户设置任意端口列表，满足个性化扫描需求。 TCP 连接测试：利用 QTcpSocket 类尝试与目标主机的指定端口建立 TCP 连接，并依据预设的超时阈值（默认为 500 毫秒）来判定端口的开放状态。

结果管理与扩展功能

扫描结果存储：扫描完成后，系统支持将详细的扫描结果（包括 IP 地址、主机名、MAC 地址、各端口状态等）导出并保存为 CSV (Comma-Separated Values) 文件格式。线程池管理：使用 QtConcurrent 线程池控制并行扫描任务，最大线程数为 CPU 核心数，避免资源耗尽。超时控制：系统采用双重超时控制机制，包括单个端口扫描超时以及全局扫描任务超时（设定为 120 秒），以确保扫描过程的整体效率与响应性。

4.1.2 核心代码解析

构造函数与初始化 构造函数负责初始化扫描器所需的核心参数，包括目标端口列表、连接超时时间以及内部线程池配置，旨在确保扫描器能够稳定、可靠地启动。端口列表可通过 setCustomPortsToScan 动态修改。线程池的运用旨在优化扫描过程的并行性能，通过复用线程资源，有效避免了因创建过多线程而可能导致的系统响应迟滞或卡顿现象。

```
1    /**
2    * @brief NetworkScanner类构造函数
3    * @param parent 父对象
4    * @details 初始化扫描器参数和线程池。默认扫描常用端口，并设置线程池最
     大线程数。
5    *          初始化ping信号量以控制并发ping操作。
6    */
7    NetworkScanner::NetworkScanner(QObject *parent)
8        : QObject(parent), m_isScanning(false), m_scannedHosts(0),
     m_totalHosts(0),
9          m_scanTimeout(500), m_useCustomRange(false), m_scanMode(
     STANDARD),
10         m_debugMode(false), m_randomizeScan(true)
11   {
12       try {
13           // 默认扫描常用端口
14           m_portsToScan.clear();
15           m_portsToScan << 21 << 22 << 23 << 25 << 53 << 80 << 110 <<
     135 << 139 << 143 << 443 << 445 << 993 << 995 << 1723 << 3306 << 3389
      << 5900 << 8080;
16
17           // 设置线程池最大线程数，避免创建过多线程
18           m_threadPool.setMaxThreadCount(QThread::idealThreadCount());
19
20           // 初始化时间记录
21           m_lastProgressUpdate = QDateTime::currentDateTime();
22           m_pingSemaphore = new QSemaphore(m_maxConcurrentPings);
23       } catch (...) {
24           qWarning() << "初始化NetworkScanner时发生异常";
25           // 确保至少添加一些基本端口
```

```
            if (m_portsToScan.isEmpty()) {
                m_portsToScan << 80 << 443 << 22;
            }
        }
    }

```

Listing 1: NetworkScanner 构造函数

扫描启动逻辑 IP 范围处理：该逻辑依据用户是否启用自定义 IP 扫描范围，动态生成待扫描的 IP 地址列表，并通过限制扫描规模来平衡扫描的深度与执行性能。并行扫描：借助 QtConcurrent 框架实现多线程并发扫描，其中每个待扫描的 IP 地址被分配为一个独立的异步任务，从而显著提升整体扫描速度。超时控制：系统通过 processScanResults 方法进行周期性检查，若总体扫描时长超出预设的 120 秒上限，则会强制终止当前的扫描任务。

```
    void NetworkScanner::startScan()
    {
        try {
            if (m_isScanning)
                return;

            m_isScanning = true;
            m_scannedHosts = 0;
            m_scannedHostsList.clear();
            m_scanFutures.clear();  // 清除之前的任务

            // 重置定时器和状态
            m_scanStartTime.start();
            m_lastProgressUpdate = QDateTime::currentDateTime();

            emit scanStarted();
            qDebug() << "开始扫描网络...";

            QList<QHostAddress> addressesToScan;

            if (m_useCustomRange) {
                // 使用自定义IP范围
                quint32 startIP = m_startIPRange.toIPv4Address();
                quint32 endIP = m_endIPRange.toIPv4Address();

                if (startIP <= endIP) {
                    // 限制最大IP数量，根据实际需求调整
                    quint32 maxIPs = 100; // 增加到100个，但添加随机化
                    m_totalHosts = qMin(maxIPs, endIP - startIP + 1);

                    // 创建所有可能的IP地址
                    QList<quint32> allIPs;
                    for (quint32 ip = startIP, count = 0; ip <= endIP &&
count < maxIPs; ++ip, ++count) {
                        allIPs.append(ip);
                    }

```

```
                    // 随机打乱IP顺序，避免顺序扫描
                    std::random_device rd;
                    std::mt19937 g(rd());
                    std::shuffle(allIPs.begin(), allIPs.end(), g);

                    // 转换为QHostAddress
                    for (quint32 ip : allIPs) {
                        addressesToScan << QHostAddress(ip);
                    }
                } else {
                    qDebug() << "无效的IP范围";
                    m_isScanning = false;
                    emit scanError("无效的IP范围");
                    emit scanFinished();
                    return;
                }
            } else {
                // 获取本地网络地址并扫描整个子网
                QList<QHostAddress> networkAddresses =
getLocalNetworkAddresses();

                if (networkAddresses.isEmpty()) {
                    qDebug() << "没有找到可用的网络接口";
                    m_isScanning = false;
                    emit scanError("没有找到可用的网络接口");
                    emit scanFinished();
                    return;
                }

```

Listing 2: NetworkScanner 扫描启动逻辑

主机扫描核心方法 端口扫描：该方法通过 QTcpSocket 的同步连接操作（connectToHost 后调用 waitForConnected）尝试连接目标主机的指定端口，并依据 waitForConnected 的布尔返回值来判定端口的开放状态。线程安全：为确保多线程环境下的数据一致性，代码采用 QMutexLocker 对共享数据（如 m_scannedHostsList 列表、扫描进度计数器等）的访问进行互斥保护，有效避免了潜在的数据竞争问题。

```
    void NetworkScanner::scanHost(const QHostAddress &address)
    {
        try {
            if (!m_isScanning) return;

            HostInfo hostInfo;
            hostInfo.ipAddress = address.toString();
            hostInfo.scanTime = QDateTime::currentDateTime();

            QElapsedTimer timer;
            timer.start();

            // 使用新的pingHost方法进行主机可达性检测
            hostInfo.isReachable = pingHost(address.toString(),
m_scanTimeout + 300); // ping超时设为扫描超时+300ms缓冲

```

```
            int responseTime = timer.elapsed();
            if (m_debugMode || hostInfo.isReachable) { // 仅在调试模式或主机可达时输出
                qDebug() << "Host:" << address.toString() << "Reachable:" << hostInfo.isReachable
                         << "Response time:" << responseTime << "ms";
            }

            hostInfo.hostName = address.toString();
            hostInfo.macAddress = generatePseudoMACFromIP(address.toString());
            hostInfo.macVendor = lookupMacVendor(hostInfo.macAddress);

            if (hostInfo.isReachable) {
                try {
                    scanHostPorts(hostInfo);
                } catch (const std::exception &e) {
                    qDebug() << "端口扫描异常: " << address.toString() << " - " << e.what();
                } catch (...) {
                    qDebug() << "端口扫描未知异常: " << address.toString();
                }
            } else {
                for (int port : m_portsToScan) {
                    hostInfo.openPorts[port] = false;
                }
            }

            {
                QMutexLocker locker(&m_mutex);
                m_scannedHostsList.append(hostInfo);
                emit hostFound(hostInfo);
            }

            {
                QMutexLocker locker(&m_mutex);
                m_scannedHosts++;
                    // 进度更新逻辑移至processScanResults或专用定时器，此处不再直接更新UI
            }
        } catch (const std::exception &e) {
            qWarning() << "扫描主机时发生异常: " << address.toString() << " - " << e.what();
        } catch (...) {
            qWarning() << "扫描主机时发生未知异常: " << address.toString();
        }
    }


```

Listing 3: NetworkScanner **主机扫描核心方法**

结果导出 此方法负责将收集到的扫描结果以 CSV (Comma-Separated Values) 导出，为后续的数据分析、报告生成或外部工具导入提供了便利。CSV 文件的首行为标题行，包含了所有被扫描的端口号；后续的数据行则逐一对应每个被扫描主机及其各个端口的开放状态。时间戳的记录格式统一采用 yyyy-MM-dd hh:mm:ss，该格式符合通用的日志记录与数据交换规范。

```
    /**
    * @brief 保存结果到文件
    * @param filename 文件名
    */
    void NetworkScanner::saveResultsToFile(const QString &filename) const
    {
        QFile file(filename);
        if (file.open(QIODevice::WriteOnly | QIODevice::Text)) {
            QTextStream out(&file);

            // 添加保存时间和扫描信息
            out << "# 网络扫描结果\n";
            out << "# 保存时间: " << QDateTime::currentDateTime().
    toString("yyyy-MM-dd hh:mm:ss") << "\n";
            out << "# 扫描主机数: " << m_scannedHostsList.size() << "\n";

            int reachableHosts = 0;
            for (const HostInfo &host : m_scannedHostsList) {
                if (host.isReachable) {
                    reachableHosts++;
                }
            }
            out << "# 可达主机数: " << reachableHosts << "\n\n";

            // 写入CSV标题
            out << "IP地址,主机名,MAC地址,厂商,状态,扫描时间";

            // 写入端口列表标题
            for (int port : m_portsToScan) {
                out << ",端口" << port;
            }
            out << "\n";

            // 写入每个主机的数据，先写可达主机，再写不可达主机
            // 先将主机分类
            QList<HostInfo> reachableList;
            QList<HostInfo> unreachableList;

            for (const HostInfo &host : m_scannedHostsList) {
                if (host.isReachable) {
                    reachableList.append(host);
                } else {
                    unreachableList.append(host);
                }
            }

            // 对可达主机按IP排序
            std::sort(reachableList.begin(), reachableList.end(),
```

```
                  [](const HostInfo &a, const HostInfo &b) {
                      return a.ipAddress < b.ipAddress;
                  });

        // 先输出可达主机
        for (const HostInfo &host : reachableList) {
            out << host.ipAddress << ","
                << host.hostName << ","
                << host.macAddress << ","
                << host.macVendor << ","
                << "可达" << ","
                << host.scanTime.toString("yyyy-MM-dd hh:mm:ss");

            // 写入端口状态
            for (int port : m_portsToScan) {
                if (host.openPorts.contains(port)) {
                    out << "," << (host.openPorts[port] ? "开放" : "关闭");
                } else {
                    out << ",未扫描";
                }
            }
            out << "\n";
        }

        // 再输出不可达主机
        for (const HostInfo &host : unreachableList) {
            out << host.ipAddress << ","
                << host.hostName << ","
                << host.macAddress << ","
                << host.macVendor << ","
                << "不可达" << ","
                << host.scanTime.toString("yyyy-MM-dd hh:mm:ss");

            // 写入端口状态
            for (int port : m_portsToScan) {
                out << ",未扫描";
            }
            out << "\n";
        }

        file.close();
        qDebug() << "结果已保存到: " << filename;
    } else {
        qDebug() << "无法保存到文件: " << filename << " - " << file.errorString();
    }
}

```

Listing 4: NetworkScanner 结果存储

4.2 网络拓扑可视化模块

4.2.1 核心功能

设备节点图形化：系统依据识别出的设备类型（例如，路由器、服务器、个人计算机、移动终端等），在拓扑图中以不同颜色编码进行差异化显示。当用户鼠标悬停于任一设备节点之上时，系统将即时浮窗显示该设备的详细信息（包括 IP 地址、MAC 地址、开放端口列表等）；同时，被选中的节点将以高亮方式予以突显。拓扑图支持用户对设备节点进行自由拖拽和视图缩放操作，在此过程中，节点间的连接关系将进行动态实时更新。

连接关系渲染：系统能够以不同线型（例如，实线表示直接物理连接、虚线表示无线连接、点划线表示 VPN 连接、波浪线表示路由路径）来区分并展示网络中的多种连接类型。在节点位置发生变化时，连接线能够自动调整其路径，以避免不必要的视觉遮挡或直接穿过节点图形中心区域。连接线上将辅以箭头标识，用以清晰指示数据流向或逻辑连接的主要方向，便于用户理解网络通信路径。

拓扑布局与分析：系统能够依据设备的 TTL (Time-To-Live) 值或其所属子网关系，将网络节点自动进行分层排列（例如，核心路由器通常位于第 0 层，与其直接相连的设备则分布于第 1 层）。用户可通过一键操作，快速生成组织有序、视觉清晰的拓扑结构，系统支持包括环形布局、分层布局在内的多种经典布局算法。系统能够按照 IP 子网对设备节点进行逻辑分组与可视化呈现，极大地方便了用户快速定位并分析特定子网内部的设备情况。

交互与控制：拓扑图界面支持视图大小调整，并提供一键重置功能，可将视图迅速恢复至预设的最佳显示比例。系统提供一个集成的控制面板，内置自动布局、视图重置、缩放调节等常用功能按钮，旨在提升用户的操作便捷性与交互体验。

4.2.2 核心代码解析

图形渲染 此代码片段主要负责定义设备节点在拓扑图中的基础视觉样式，并实现了节点被选中时的高亮效果以及鼠标悬停时的信息提示功能。

```
    /**
    * @brief 绘制设备节点。
    * @param painter QPainter 指针。
    * @param option QStyleOptionGraphicsItem 指针。
    * @param widget QWidget 指针（关联的视图）。
    * @details 根据设备类型绘制不同颜色和图标的圆形节点，并显示IP地址。
    *          选中或高亮时颜色会变化。
    */
void DeviceNode::paint(QPainter *painter, const QStyleOptionGraphicsItem
    *option, QWidget *widget)
{
    Q_UNUSED(widget);

    // 绘制设备节点
    painter->setRenderHint(QPainter::Antialiasing);

    // 设置颜色
    QColor baseColor;
    switch (m_type) {
        case DEVICE_ROUTER: baseColor = Qt::blue; break;
        case DEVICE_SERVER: baseColor = Qt::darkGreen; break;
```

```
        case DEVICE_PC: baseColor = Qt::darkCyan; break;
        case DEVICE_MOBILE: baseColor = Qt::magenta; break;
        case DEVICE_PRINTER: baseColor = Qt::darkYellow; break;
        case DEVICE_IOT: baseColor = Qt::darkMagenta; break;
        default: baseColor = Qt::gray; break;
    }

    // 如果选中或高亮，使用更亮的颜色
    if (isSelected() || m_highlight) {
        baseColor = baseColor.lighter(150);
    }

    // 绘制设备图标背景
    painter->setBrush(baseColor);
    painter->setPen(QPen(baseColor.darker(), 2));
    painter->drawEllipse(QPointF(0, 0), 25, 25);

    // 绘制设备图标
    painter->setPen(Qt::white);
    painter->setFont(QFont("Arial", 12, QFont::Bold));
    QString iconText;
    switch (m_type) {
        case DEVICE_ROUTER: iconText = "R"; break;
        case DEVICE_SERVER: iconText = "S"; break;
        case DEVICE_PC: iconText = "PC"; break;
        case DEVICE_MOBILE: iconText = "M"; break;
        case DEVICE_PRINTER: iconText = "PR"; break;
        case DEVICE_IOT: iconText = "IoT"; break;
        default: iconText = "?"; break;
    }

    // 居中绘制文本
    QFontMetrics fm(painter->font());
    QRect textRect = fm.boundingRect(iconText);
    painter->drawText(QPointF(-textRect.width()/2, textRect.height()/3),
    iconText);

    // 显示IP地址
    painter->setPen(Qt::black);
    painter->setFont(QFont("Arial", 8));
    QString ipText = m_host.ipAddress;
    QFontMetrics fmIp(painter->font());
    QRect ipRect = fmIp.boundingRect(ipText);
    painter->drawText(QPointF(-ipRect.width()/2, 40), ipText);
}
```

Listing 5: DeviceNode 图形渲染

交互逻辑 当用户在拓扑图中拖拽设备节点时，该事件处理逻辑将被触发，进而通知图形场景（scene）进行刷新，以确保连接线能够根据节点位置的变动进行实时的重绘。

```
    /**
```

```
     * @brief 处理鼠标移动事件。
     * @param event 鼠标事件指针。
     * @details 更新场景以重绘连接线。
     */
    void DeviceNode::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
    {
        QGraphicsItem::mouseMoveEvent(event);

        // 通知连接线更新位置
        scene()->update();
    }


```

Listing 6: DeviceNode 交互逻辑

连接渲染 该部分代码根据设备间不同的连接类型（例如，直接物理连接、无线连接、路由连接等），采用差异化的线条样式（如实线、虚线、颜色等）来渲染连接线，以增强拓扑图的可读性。

```
    /**
     * @brief 绘制连接线。
     * @param painter QPainter 指针。
     * @param option QStyleOptionGraphicsItem 指针。
     * @param widget QWidget 指针。
     * @details 根据连接类型绘制不同样式的线条，并在线条末端绘制箭头。
     *          线条会连接到节点的边缘而不是中心。
     */
    void ConnectionLine::paint(QPainter *painter, const
QStyleOptionGraphicsItem *option, QWidget *widget)
    {
        Q_UNUSED(option);
        Q_UNUSED(widget);

        if (!m_source || !m_target) return;

        QPointF sourcePoint = m_source->pos();
        QPointF targetPoint = m_target->pos();

        // 计算源节点和目标节点之间的方向向量
        QLineF line(sourcePoint, targetPoint);

        // 确保不绘制穿过节点的线，而是到节点边缘
        qreal angle = std::atan2(targetPoint.y() - sourcePoint.y(),
targetPoint.x() - sourcePoint.x());
        qreal sourceRadius = 25;  // 节点半径
        qreal targetRadius = 25;  // 节点半径

        QPointF sourceEdge(
            sourcePoint.x() + sourceRadius * std::cos(angle),
            sourcePoint.y() + sourceRadius * std::sin(angle)
        );

        QPointF targetEdge(
```

```
            targetPoint.x() - targetRadius * std::cos(angle),
            targetPoint.y() - targetRadius * std::sin(angle)
        );

        // 根据连接类型设置不同的样式
        switch (m_connectionType) {
            case CONNECTION_DIRECT:
                // 实线表示直接连接
                painter->setPen(QPen(Qt::darkGreen, 1.5, Qt::SolidLine,
Qt::RoundCap, Qt::RoundJoin));
                break;

            case CONNECTION_WIRELESS:
                // 虚线表示无线连接
                painter->setPen(QPen(Qt::blue, 1.5, Qt::DashLine, Qt::
RoundCap, Qt::RoundJoin));
                break;

            case CONNECTION_VPN:
                // 点划线表示VPN连接
                painter->setPen(QPen(Qt::darkCyan, 1.5, Qt::DashDotLine,
Qt::RoundCap, Qt::RoundJoin));
                break;

            case CONNECTION_ROUTED:
                // 波浪线表示通过路由器的连接
                painter->setPen(QPen(Qt::darkGray, 1.5, Qt::
DashDotDotLine, Qt::RoundCap, Qt::RoundJoin));
                break;

            default:
                painter->setPen(QPen(Qt::gray, 1.5, Qt::SolidLine, Qt::
RoundCap, Qt::RoundJoin));
                break;
        }

        // 绘制连接线
        painter->drawLine(sourceEdge, targetEdge);

        // 绘制箭头
        qreal arrowSize = 10;
        double angle_arrow = std::atan2(targetEdge.y() - sourceEdge.y(),
targetEdge.x() - sourceEdge.x());
        QPointF arrowP1 = targetEdge - QPointF(sin(angle_arrow + M_PI /
3) * arrowSize,
                                               cos(angle_arrow + M_PI /
3) * arrowSize);
        QPointF arrowP2 = targetEdge - QPointF(sin(angle_arrow + M_PI -
M_PI / 3) * arrowSize,
                                               cos(angle_arrow + M_PI -
M_PI / 3) * arrowSize);

        QPolygonF arrowHead;
        arrowHead << targetEdge << arrowP1 << arrowP2;
```

```
77          painter->setBrush(Qt::gray);
78          painter->drawPolygon(arrowHead);
79      }
80
```

Listing 7: ConnectionLine 连接渲染

布局算法 该布局算法依据设备的逻辑层级（例如，通过 TTL 值推断）对节点进行分组，并按层次结构进行排列：同层节点在水平方向上均匀分布，不同层级之间则保持一定的垂直间距。为提升用户体验，可选用 QTimeLine 实现节点位置调整时的平滑动画效果。

```
1   /**
2    * @brief 应用层次化布局算法。
3    * @param layers 一个映射表，键为层级，值为该层级下的设备IP列表。
4    * @details 根据层级信息，将节点在垂直方向上分层排列，同层节点水平排
    列。
5    *          节点放置时带有动画效果。
6    */
7   void NetworkTopologyView::hierarchicalLayout(const QMap<int,
    QStringList> &layers)
8   {
9       // 首先确定各个层级的节点数量，以便计算布局
10      int maxNodesInLayer = 0;
11      for (auto it = layers.begin(); it != layers.end(); ++it) {
12          maxNodesInLayer = qMax(maxNodesInLayer, it.value().size());
13      }
14
15      // 计算水平和垂直间距
16      qreal verticalSpacing = 150;  // 层级之间的垂直间距
17      qreal horizontalSpacing = 120;  // 同层节点之间的水平间距
18
19      // 确保最宽的层也能容纳所有节点
20      qreal sceneWidth = qMax(600.0, maxNodesInLayer *
    horizontalSpacing);
21      m_scene->setSceneRect(-sceneWidth/2, -200, sceneWidth, (layers.
    size() + 1) * verticalSpacing);
22
23      // 按层放置节点
24      for (auto it = layers.begin(); it != layers.end(); ++it) {
25          int layer = it.key();
26          const QStringList &ips = it.value();
27
28          // 该层有多少个节点
29          int nodeCount = ips.size();
30
31          // 计算该层第一个节点的水平位置
32          qreal startX = -((nodeCount - 1) * horizontalSpacing) / 2;
33
34          // 放置该层的所有节点
35          for (int i = 0; i < nodeCount; ++i) {
36              QString ip = ips[i];
37              if (m_nodes.contains(ip)) {
```

```
                    QPointF pos(startX + i * horizontalSpacing, layer *
verticalSpacing);

                    // 创建动画效果
                    QTimeLine *timer = new QTimeLine(500, this);
                    timer->setFrameRange(0, 100);

                    QGraphicsItemAnimation *animation = new
QGraphicsItemAnimation;
                    animation->setItem(m_nodes[ip]);
                    animation->setTimeLine(timer);

                    // 设置起始位置和目标位置
                    QPointF startPos = m_nodes[ip]->pos();
                    QPointF endPos = pos;

                    for (int j = 0; j <= 100; ++j) {
                        qreal stepX = startPos.x() + (endPos.x() -
startPos.x()) * j / 100.0;
                        qreal stepY = startPos.y() + (endPos.y() -
startPos.y()) * j / 100.0;
                        animation->setPosAt(j / 100.0, QPointF(stepX,
stepY));
                    }

                    timer->start();
                    connect(timer, &QTimeLine::finished, timer, &QObject
::deleteLater);
                    connect(timer, &QTimeLine::finished, animation, &
QObject::deleteLater);
                }
            }
        }

        // 更新所有连接线的位置
        for (ConnectionLine *line : m_connections) {
            line->updatePosition();
        }

        // 更新视图
        m_scene->update();
    }


```

Listing 8: NetworkTopologyView 布局算法

设备类型推断 此函数综合利用设备的 IP 地址、主机名以及从 MAC 地址解析出的制造商信息，来推断设备的具体类型。推断结果将直接应用于后续的设备图标渲染及拓扑图布局中的节点分组。

```
    /**
    * @brief 根据主机信息判断设备类型
    * @param host 主机信息
```

```
4       * @return 设备类型字符串
5       */
6       QString DeviceAnalyzer::determineDeviceType(const HostInfo &host)
7       {
8           // 根据IP地址、主机名、MAC厂商等信息推断设备类型
9           QString ipLower = host.ipAddress.toLower();
10          QString hostLower = host.hostName.toLower();
11          QString vendorLower = host.macVendor.toLower();
12
13          // 检测路由器
14          if (vendorLower.contains("路由") ||
15              vendorLower.contains("router") ||
16              hostLower.contains("router") ||
17              hostLower.contains("gateway") ||
18              ipLower.endsWith(".1") ||
19              ipLower.endsWith(".254")) {
20              return "路由器";
21          }
22
23          // 检测服务器
24          if (hostLower.contains("server") ||
25              vendorLower.contains("server") ||
26              vendorLower.contains("dell") ||
27              vendorLower.contains("ibm") ||
28              vendorLower.contains("hp") && !vendorLower.contains("printer"
    )) {
29              return "服务器";
30          }
31
32          // 检测打印机
33          if (hostLower.contains("print") ||
34              vendorLower.contains("print") ||
35              vendorLower.contains("canon") ||
36              vendorLower.contains("hp") && vendorLower.contains("printer")
     ||
37              vendorLower.contains("epson")) {
38              return "打印机";
39          }
40
41          // 检测移动设备
42          if (vendorLower.contains("apple") ||
43              vendorLower.contains("iphone") ||
44              vendorLower.contains("ipad") ||
45              vendorLower.contains("android") ||
46              vendorLower.contains("xiaomi") ||
47              vendorLower.contains("huawei") ||
48              vendorLower.contains("oppo") ||
49              vendorLower.contains("vivo")) {
50              return "移动设备";
51          }
52
53          // 检测IoT设备
54          if (hostLower.contains("esp") ||
55              hostLower.contains("arduino") ||
```

```
            hostLower.contains("raspberry") ||
            hostLower.contains("iot") ||
            vendorLower.contains("nest") ||
            vendorLower.contains("smart")) {
            return "智能设备";
        }

        // 默认为PC
        return "个人电脑";
    }

```

Listing 9: NetworkTopologyView 设备类型推断

连接关系推断 该逻辑旨在以网络中的网关设备（Gateway）为核心参照点，推断并构建其他网络设备与主网关或各子网网关之间的连接关系，力求在拓扑图中模拟出更接近真实物理与逻辑结构的网络连接形态。

```
    /**
    * @brief 根据主机信息推断设备之间的连接关系。
    * @param hosts 包含网络中所有主机信息的列表。
    * @return 一个映射表，键为源设备IP，值为目标设备IP列表，表示它们之间的连接。
    * @details 连接关系可能基于网关信息、TTL层级和子网分析。
    *          优先查找网关设备作为连接中心，然后分析TTL层次和子网结构来构建连接。
    */
    QMap<QString, QStringList> TopologyAnalyzer::inferDeviceConnections(const QList<HostInfo> &hosts)
    {
        QMap<QString, QStringList> connections;

        // 寻找网关/路由器设备
        QString gatewayIP;
        for (const HostInfo &host : hosts) {
            if (host.ipAddress.endsWith(".1") || host.ipAddress.endsWith(".254") ||
                host.macVendor.contains("路由", Qt::CaseInsensitive) ||
                host.hostName.contains("router", Qt::CaseInsensitive) ||
                host.hostName.contains("gateway", Qt::CaseInsensitive)) {
                gatewayIP = host.ipAddress;
                break;
            }
        }

        // 分析TTL层次
        QMap<int, QStringList> layers = analyzeTTLLayers(hosts);

        // 分析子网
        QMap<QString, QStringList> subnets = analyzeSubnets(hosts);

        // 基于以上信息构建连接关系
        if (!gatewayIP.isEmpty()) {
            // 路由器是所有连接的中心
```

```
33              QStringList routerConnections;
34
35              // 找到所有第二层设备(与路由器直连)
36              if (layers.contains(1)) {
37                  for (const QString &ip : layers[1]) {
38                      routerConnections.append(ip);
39
40                      // 第二层设备可能是子网的网关
41                      for (auto it = subnets.begin(); it != subnets.end();
    ++it) {
42                          if (it.key() != calculateSubnet(gatewayIP) &&
43                              it.value().contains(ip)) {
44                              // 这个设备可能是子网的网关，连接到该子网中的
    其他设备
45                              QStringList subnetConnections;
46                              for (const QString &subnetIP : it.value()) {
47                                  if (subnetIP != ip) {
48                                      subnetConnections.append(subnetIP);
49                                  }
50                              }
51                              connections[ip] = subnetConnections;
52                          }
53                      }
54                  }
55              }
56
57              connections[gatewayIP] = routerConnections;
58          } else {
59              // 没有明确的路由器，使用基于子网的连接关系
60              for (auto it = subnets.begin(); it != subnets.end(); ++it) {
61                  // 在每个子网中，找到可能的网关
62                  QString subnetGateway;
63                  for (const QString &ip : it.value()) {
64                      if (ip.endsWith(".1") || ip.endsWith(".254")) {
65                          subnetGateway = ip;
66                          break;
67                      }
68                  }
69
70                  if (!subnetGateway.isEmpty()) {
71                      // 将该子网中的其他设备连接到网关
72                      QStringList gatewayConnections;
73                      for (const QString &ip : it.value()) {
74                          if (ip != subnetGateway) {
75                              gatewayConnections.append(ip);
76                          }
77                      }
78                      connections[subnetGateway] = gatewayConnections;
79                  }
80              }
81          }
82
83          return connections;
84      }
```

```
85
```

Listing 10: TopologyAnalyzer 连接关系推断

4.3 主窗口模块

4.3.1 核心功能

扫描控制与配置

多模式扫描配置：提供包括自动扫描本地子网（默认扫描子网内前 100 个活跃 IP）、自定义 IP 地址范围扫描（支持 CIDR 表示法或显式指定起始-结束 IP 地址对）、以及自定义端口列表扫描（用户可选择预设的常用端口组合或手动输入特定端口号）在内的多种扫描模式。精细化参数配置：允许用户对关键扫描参数进行配置，例如网络连接的超时阈值，以及扫描策略的选择（例如，通过调整待扫描端口列表的规模实现" 快速扫描" 或" 深度扫描" 模式的切换）。扫描任务控制与进度监控：主界面提供实时的进度条用以显示当前扫描任务的完成百分比，并允许用户在扫描过程中随时发起停止操作。

扫描结果展示

结构化扫描结果展示：扫描结果以表格形式清晰呈现，包含 IP 地址、主机名、MAC 地址、制造商信息、网络可达性状态（可达/不可达）以及开放端口列表等关键字段。其中，成功探测到的可达设备行将以特定背景色（如绿色）进行标记，表格同时支持按列排序及基于 IP 地址、制造商或设备类型等多维度的数据筛选功能。设备详细信息查阅：用户可通过双击结果表格中的任一行，快速查看对应设备的更为详尽的信息，包括完整的开放端口列表、本次扫描的确切时间等。主界面采用多标签页设计，方便用户在扫描结果、主机详情、网络拓扑、统计分析及扫描历史等不同功能模块间进行切换。

网络拓扑可视化

动态交互式网络拓扑图：集成动态网络拓扑可视化功能，支持多种自动布局算法（如层次化布局、环形布局等），允许用户通过鼠标拖拽调整节点位置，并通过滚轮或控制按钮进行视图缩放。拓扑图能以不同样式的连接线（例如，区分直连、无线、路由、VPN 等）并辅以方向箭头，直观展示设备间的连接关系。视图间交互联动：实现拓扑图节点与结果表格行之间的双向联动选择功能，即点击拓扑图中的任一设备节点，结果表格中对应的条目将被同步高亮选中，反之亦然。此外，用户可将当前生成的网络拓扑图便捷地导出并保存为 PNG 或 JPG 图像文件。

统计分析与安全报告

多维度统计分析图表：系统能够自动生成多种统计分析图表，包括设备类型分布饼图、网络设备制造商分布柱状图，以及常见服务端口开放情况的折线图等，为用户提供宏观的网络画像。自动化安全风险报告：工具能够自动识别并列出那些开放了已知高危端口（例如 RDP 端口 3389、SSH 端口 22 等）的设备，并对潜在的风险设备（如来源不明或配置不当的 IoT 设备）进行标记，初步形成安全态势感知。

扫描历史管理

扫描会话管理与持久化：系统能够自动保存每次扫描任务的完整会话信息，包括扫描执行时间、发现的设备列表、以及当时所使用的扫描配置参数。同时，用户也可以

手动将特定扫描历史记录导出为 JSON 文件格式进行存档，或从已保存的 JSON 文件中加载历史数据。历史会话差异比较：提供强大的会话比较功能，允许用户选取任意两次历史扫描记录进行对比分析，从而清晰识别出网络中新增或消失的设备、设备端口状态的变化等，并能以可视化方式辅助用户理解网络拓扑的动态演变趋势。

系统设置与主题

用户偏好设置持久化：系统能够记忆并持久化用户的常用扫描配置，如默认的 IP 扫描范围、自定义端口列表、连接超时设置等，同时也会保存主窗口最后关闭时的位置与尺寸信息，以便下次启动时恢复。个性化界面主题：为提升用户体验，工具支持浅色与深色两种界面主题模式的一键切换，用户的选择会自动保存。未来可扩展支持更为细致的界面样式自定义功能。

4.3.2 核心代码解析

初始化与布局 主窗口采用 QTabWidget 控件来实现不同功能模块（如扫描配置、结果展示、网络拓扑、统计分析、历史记录等）的清晰分隔与便捷切换，从而提升了用户界面的整体组织性与导航效率。界面布局则主要借助 QVBoxLayout 和 QHBoxLayout 等布局管理器，以实现响应式的用户界面设计，确保窗口在不同尺寸和缩放比例下均能保持良好的视觉效果与可用性。

```
1    /**
2    * @brief MainWindow 构造函数
3    * @param parent 父控件指针
4    */
5    MainWindow::MainWindow(QWidget *parent)
6        : QMainWindow(parent), m_hostsFound(0), m_currentHostIndex(-1),
     m_darkModeEnabled(false),
7        m_deviceTypeChartView(nullptr), m_vendorChartView(nullptr),
     m_portDistributionChartView(nullptr),
8        m_deviceAnalyzer(nullptr), m_scanHistory(nullptr), m_scanner(
     nullptr), m_networkTopology(nullptr)
9    {
10       // 创建UI组件
11       createUI();
12       createMenus();
13
14       // 创建网络扫描器
15       m_scanner = new NetworkScanner(this);
16
17       // 创建设备分析器
18       m_deviceAnalyzer = new DeviceAnalyzer(this);
19
20       // 创建扫描历史管理器
21       m_scanHistory = new ScanHistory(this);
22
23       // 设置图表
24       if (m_deviceTypeChartView && m_deviceAnalyzer) {
25           m_deviceTypeChartView->setChart(m_deviceAnalyzer->
     getDeviceTypeChart());
26       }
27
```

```
28          if (m_vendorChartView && m_deviceAnalyzer) {
29              m_vendorChartView->setChart(m_deviceAnalyzer->
    getVendorDistributionChart());
30          }
31
32          if (m_portDistributionChartView && m_deviceAnalyzer) {
33              m_portDistributionChartView->setChart(m_deviceAnalyzer->
    getPortDistributionChart());
34          }
35
36          // 连接信号和槽
37          setupConnections();
38
39          // 加载保存的设置
40          loadSettings();
41
42          // 设置初始状态
43          m_stopButton->setEnabled(false);
44          m_statusBar->showMessage("网络扫描器就绪");
45      }
46
47      /**
48      * @brief 创建用户界面
49      */
50      void MainWindow::createUI()
51      {
52          // 创建中央部件和标签页
53          m_centralWidget = new QWidget(this);
54          setCentralWidget(m_centralWidget);
55
56          QVBoxLayout *centralLayout = new QVBoxLayout(m_centralWidget);
57          m_tabWidget = new QTabWidget(m_centralWidget);
58          centralLayout->addWidget(m_tabWidget);
59
60          // 创建扫描结果标签页
61          m_scanTab = new QWidget(m_tabWidget);
62          m_mainLayout = new QVBoxLayout(m_scanTab);
63
64          // 创建控制区域
65          m_controlLayout = new QHBoxLayout();
66          m_scanButton = new QPushButton("开始扫描", m_scanTab);
67          m_stopButton = new QPushButton("停止扫描", m_scanTab);
68          m_clearButton = new QPushButton("清除结果", m_scanTab);
69          m_saveButton = new QPushButton("保存结果", m_scanTab);
70          m_progressBar = new QProgressBar(m_scanTab);
71          m_progressBar->setRange(0, 100);
72          m_progressBar->setValue(0);
73          m_statusLabel = new QLabel("就绪", m_scanTab);
74
75          m_controlLayout->addWidget(m_scanButton);
76          m_controlLayout->addWidget(m_stopButton);
77          m_controlLayout->addWidget(m_clearButton);
78          m_controlLayout->addWidget(m_saveButton);
79          m_controlLayout->addWidget(m_progressBar);
```

```
80          m_controlLayout->addWidget(m_statusLabel);
81
82          // 创建过滤控件
83          m_filterWidget = new QWidget(m_scanTab);
84          QHBoxLayout *filterLayout = new QHBoxLayout(m_filterWidget);
85
86          QLabel *filterLabel = new QLabel("过滤:", m_filterWidget);
87          m_filterIPLineEdit = new QLineEdit(m_filterWidget);
88          m_filterIPLineEdit->setPlaceholderText("IP地址");
89
90          m_filterVendorComboBox = new QComboBox(m_filterWidget);
91          m_filterVendorComboBox->addItem("所有厂商");
92
93          m_filterTypeComboBox = new QComboBox(m_filterWidget);
94          m_filterTypeComboBox->addItem("所有设备类型");
95          m_filterTypeComboBox->addItem("路由器");
96          m_filterTypeComboBox->addItem("服务器");
97          m_filterTypeComboBox->addItem("个人电脑");
98          m_filterTypeComboBox->addItem("移动设备");
99          m_filterTypeComboBox->addItem("打印机");
100         m_filterTypeComboBox->addItem("智能设备");
101
102         m_filterButton = new QPushButton("应用过滤", m_filterWidget);
103         m_clearFilterButton = new QPushButton("清除过滤", m_filterWidget)
    ;
104
105         filterLayout->addWidget(filterLabel);
106         filterLayout->addWidget(m_filterIPLineEdit);
107         filterLayout->addWidget(m_filterVendorComboBox);
108         filterLayout->addWidget(m_filterTypeComboBox);
109         filterLayout->addWidget(m_filterButton);
110         filterLayout->addWidget(m_clearFilterButton);
111
112         m_mainLayout->addLayout(m_controlLayout);
113         m_mainLayout->addWidget(m_filterWidget);
114
115         // 创建结果表格
116         m_resultsTable = new QTableWidget(0, 6, m_scanTab);
117         QStringList headers;
118         headers << "IP地址" << "主机名" << "MAC地址" << "厂商" << "状态"
    << "开放端口";
119         m_resultsTable->setHorizontalHeaderLabels(headers);
120         m_resultsTable->horizontalHeader()->setSectionResizeMode(
    QHeaderView::Stretch);
121         m_resultsTable->setSelectionBehavior(QAbstractItemView::
    SelectRows);
122         m_resultsTable->setEditTriggers(QAbstractItemView::NoEditTriggers
    );
123         m_resultsTable->setSortingEnabled(true);
124
125         m_mainLayout->addWidget(m_resultsTable);
126
127         // 创建其他标签页 - 这些方法内部会处理父子关系
128         createSettingsDialog();
```

```
129         createDetailsTab();
130         createTopologyTab();
131         createStatisticsTab();
132         createHistoryTab();
133 
134         // 添加标签页到标签页控件
135         m_tabWidget->addTab(m_scanTab, "扫描结果");
136         if (m_settingsTab) m_tabWidget->addTab(m_settingsTab, "扫描设置")
    ;
137         if (m_detailsTab) m_tabWidget->addTab(m_detailsTab, "主机详情");
138         if (m_topologyTab) m_tabWidget->addTab(m_topologyTab, "网络拓扑")
    ;
139         if (m_statisticsTab) m_tabWidget->addTab(m_statisticsTab, "统计分
    析");
140         if (m_historyTab) m_tabWidget->addTab(m_historyTab, "扫描历史");
141 
142         // 创建状态栏
143         m_statusBar = new QStatusBar(this);
144         setStatusBar(m_statusBar);
145     }
146 
```

Listing 11: MainWindow 初始化与布局

主题切换 主题切换功能主要通过动态修改应用程序的 QPalette（调色板）对象中的颜色值来实现。一旦调色板更新，Qt 框架会自动将新的配色方案应用到界面中的各类标准控件（如表格、按钮、文本框等）以及通过 Qt Charts 模块绘制的图表，从而确保整体视觉风格的一致性。

```
1     /**
2     * @brief 切换暗色/浅色主题
3     * @param enable 是否启用暗色模式
4     */
5     void MainWindow::toggleDarkMode(bool enable)
6     {
7         m_darkModeEnabled = enable;
8         applyTheme(enable);
9     }
10 
11    /**
12    * @brief 应用暗色/浅色主题
13    * @param darkMode 是否应用暗色模式
14    */
15    void MainWindow::applyTheme(bool darkMode)
16    {
17        if (darkMode) {
18            // 暗色主题
19            QPalette darkPalette;
20            QColor darkColor = QColor(45, 45, 45);
21            QColor disabledColor = QColor(127, 127, 127);
22 
23            darkPalette.setColor(QPalette::Window, darkColor);
24            darkPalette.setColor(QPalette::WindowText, Qt::white);
25            darkPalette.setColor(QPalette::Base, QColor(18, 18, 18));
```

```
            darkPalette.setColor(QPalette::AlternateBase, darkColor);
            darkPalette.setColor(QPalette::ToolTipBase, Qt::white);
            darkPalette.setColor(QPalette::ToolTipText, Qt::white);
            darkPalette.setColor(QPalette::Text, Qt::white);
            darkPalette.setColor(QPalette::Disabled, QPalette::Text,
    disabledColor);
            darkPalette.setColor(QPalette::Button, darkColor);
            darkPalette.setColor(QPalette::ButtonText, Qt::white);
            darkPalette.setColor(QPalette::Disabled, QPalette::ButtonText
    , disabledColor);
            darkPalette.setColor(QPalette::BrightText, Qt::red);
            darkPalette.setColor(QPalette::Link, QColor(42, 130, 218));
            darkPalette.setColor(QPalette::Highlight, QColor(42, 130,
    218));
            darkPalette.setColor(QPalette::HighlightedText, Qt::black);
            darkPalette.setColor(QPalette::Disabled, QPalette::
    HighlightedText, disabledColor);

            qApp->setPalette(darkPalette);

            // 设置样式表
            qApp->setStyleSheet("QToolTip { color: #ffffff; background-
    color: #2a82da; border: 1px solid white; }");
        } else {
            // 浅色主题（系统默认）
            qApp->setPalette(QApplication::style()->standardPalette());
            qApp->setStyleSheet("");
        }

        // 通知主题已更改
        emit onThemeChanged();
    }

```

Listing 12: MainWindow 主题切换

扫描启动流程 主窗口通过监听 NetworkScanner 对象发出的信号(如 hostFound、scanProgress、scanFinished 等）来异步更新界面元素，这种机制确保了扫描操作在后台线程执行时不会阻塞用户界面，同时也保障了跨线程更新 UI 的线程安全性。扫描结果表格中，可达设备的对应行会通过设置特定的背景色（例如淡绿色）来直观地区分其状态。

```
    void NetworkScanner::startScan()
    {
        try {
            if (m_isScanning)
                return;

            m_isScanning = true;
            m_scannedHosts = 0;
            m_scannedHostsList.clear();
            m_scanFutures.clear();  // 清除之前的任务

            // 重置定时器和状态
            m_scanStartTime.start();
```

```
14              m_lastProgressUpdate = QDateTime::currentDateTime();
15
16              emit scanStarted();
17              qDebug() << "开始扫描网络...";
18
19              QList<QHostAddress> addressesToScan;
20
21              if (m_useCustomRange) {
22                  // 使用自定义IP范围
23                  quint32 startIP = m_startIPRange.toIPv4Address();
24                  quint32 endIP = m_endIPRange.toIPv4Address();
25
26                  if (startIP <= endIP) {
27                      // 限制最大IP数量，根据实际需求调整
28                      quint32 maxIPs = 100; // 增加到100个，但添加随机化
29                      m_totalHosts = qMin(maxIPs, endIP - startIP + 1);
30
31                      // 创建所有可能的IP地址
32                      QList<quint32> allIPs;
33                      for (quint32 ip = startIP, count = 0; ip <= endIP &&
    count < maxIPs; ++ip, ++count) {
34                          allIPs.append(ip);
35                      }
36
37                      // 随机打乱IP顺序，避免顺序扫描
38                      std::random_device rd;
39                      std::mt19937 g(rd());
40                      std::shuffle(allIPs.begin(), allIPs.end(), g);
41
42                      // 转换为QHostAddress
43                      for (quint32 ip : allIPs) {
44                          addressesToScan << QHostAddress(ip);
45                      }
46                  } else {
47                      qDebug() << "无效的IP范围";
48                      m_isScanning = false;
49                      emit scanError("无效的IP范围");
50                      emit scanFinished();
51                      return;
52                  }
53              } else {
54                  // 获取本地网络地址并扫描整个子网
55                  QList<QHostAddress> networkAddresses =
    getLocalNetworkAddresses();
56
57                  if (networkAddresses.isEmpty()) {
58                      qDebug() << "没有找到可用的网络接口";
59                      m_isScanning = false;
60                      emit scanError("没有找到可用的网络接口");
61                      emit scanFinished();
62                      return;
63                  }
64
65                  // 计算需要扫描的IP地址总数，每个子网最多扫描50个IP
```

```
                const int MAX_IPS_PER_SUBNET = 40; // 降低每个子网扫描的IP数量
                m_totalHosts = 0;

                // 为每个网段准备扫描地址
                for (const QHostAddress &network : networkAddresses) {
                    QHostAddress baseAddress(network.toIPv4Address() & 0xFFFFFF00);

                    // 创建子网内的IP列表
                    QList<QHostAddress> subnetIPs;
                    for (int i = 1; i < 255; ++i) {
                        QHostAddress currentAddress(baseAddress.toIPv4Address() + i);
                        subnetIPs.append(currentAddress);
                    }

                    // 随机打乱子网IP顺序
                    std::random_device rd;
                    std::mt19937 g(rd());
                    std::shuffle(subnetIPs.begin(), subnetIPs.end(), g);

                    // 取前MAX_IPS_PER_SUBNET个IP
                    int count = 0;
                    for (const QHostAddress &ip : subnetIPs) {
                        if (count >= MAX_IPS_PER_SUBNET) break;
                        addressesToScan.append(ip);
                        count++;
                    }

                    m_totalHosts += qMin(MAX_IPS_PER_SUBNET, subnetIPs.size());
                }
            }

            // 确保不扫描过多IP，限制为150个
            if (addressesToScan.size() > 150) {
                qDebug() << "扫描IP过多，限制为前150个";
                while (addressesToScan.size() > 150) {
                    addressesToScan.removeLast();
                }
                m_totalHosts = 150;
            }

            qDebug() << "开始扫描" << addressesToScan.size() << "个IP地址...";

            // 分批次创建扫描任务，减少瞬时资源占用
            const int BATCH_SIZE = 5; // 每批次处理5个IP
            int totalBatches = (addressesToScan.size() + BATCH_SIZE - 1) / BATCH_SIZE;

            for (int batchIndex = 0; batchIndex < totalBatches; batchIndex++) {
```

```
113                 // 计算当前批次的起始和结束索引
114                 int startIdx = batchIndex * BATCH_SIZE;
115                 int endIdx = qMin(startIdx + BATCH_SIZE, addressesToScan.
    size());
116 
117                 // 延迟执行当前批次
118                 QTimer::singleShot(batchIndex * 500, this, [this,
    addressesToScan, startIdx, endIdx]() {
119                     if (!m_isScanning) return;
120 
121                     // 处理该批次中的每个IP
122                     for (int i = startIdx; i < endIdx; i++) {
123                         QHostAddress address = addressesToScan[i];
124 
125                         // 每处理2个IP就处理一次事件，保持UI响应
126                         if ((i - startIdx) % 2 == 0) {
127                             QCoreApplication::processEvents();
128                         }
129 
130                         // 启动扫描任务
131                         QFuture<void> future = QtConcurrent::run([this,
    address]() {
132                             try {
133                                 if (!m_isScanning) return;
134                                 scanHost(address);
135                             } catch (const std::exception &e) {
136                                 qWarning() << "扫描主机异常:" << address.
    toString() << "-" << e.what();
137                             } catch (...) {
138                                 qWarning() << "扫描主机未知异常:" <<
    address.toString();
139                             }
140                         });
141 
142                         QMutexLocker locker(&m_mutex);
143                         m_scanFutures.append(future);
144                     }
145                 });
146             }
147 
148             // 在所有批次启动后启动结果处理
149             QTimer::singleShot(500, this, &NetworkScanner::
    processScanResults);
150 
151             // 添加进度监控定时器，定期更新进度
152             QTimer *progressTimer = new QTimer(this);
153             connect(progressTimer, &QTimer::timeout, this, [this,
    progressTimer]() {
154                 if (!m_isScanning) {
155                     progressTimer->stop();
156                     progressTimer->deleteLater();
157                     return;
158                 }
159 
```

```
                // 更新进度
                if (m_totalHosts > 0) {
                    int progress = (m_scannedHosts * 100) / m_totalHosts;

                    // 处理UI事件，确保界面响应
                    QCoreApplication::processEvents();

                    emit scanProgress(progress);

                    // 记录当前进度
                    static int lastProgress = 0;
                    static int stuckCount = 0;

                    if (progress == lastProgress) {
                        stuckCount++;
                        if (stuckCount > 10) { // 如果进度停滞超过10秒
                            qDebug() << "扫描进度停滞在" << progress << "%超过10秒，考虑强制结束";

                            // 如果进度已经很高，则认为扫描基本完成
                            if (progress > 90) {
                                qDebug() << "进度已超过90%，强制完成扫描";
                                // 手动更新未完成的主机状态为不可达
                                int remaining = m_totalHosts - m_scannedHosts;
                                m_scannedHosts = m_totalHosts;
                                emit scanProgress(100);
                                stopScan();
                            }
                        }
                    } else {
                        lastProgress = progress;
                        stuckCount = 0;
                    }
                }

                // 处理UI事件，确保界面响应
                QCoreApplication::processEvents();
            });
            progressTimer->start(1000); // 每秒更新一次进度

            // 设置总超时保护
            int maxScanTime = 90000; // 1.5分钟，减少原来的2分钟
            QTimer::singleShot(maxScanTime, this, [this]() {
                if (m_isScanning) {
                    qDebug() << "扫描超时，强制结束...";
                    stopScan();
                    emit scanError("扫描超时，已自动结束");
                }
            });
        } catch (const std::exception &e) {
            qWarning() << "启动扫描时发生异常: " << e.what();
            m_isScanning = false;
```

```
211             emit scanError("启动扫描时发生异常: " + QString(e.what()));
212             emit scanFinished();
213         } catch (...) {
214             qWarning() << "启动扫描时发生未知异常";
215             m_isScanning = false;
216             emit scanError("启动扫描时发生未知异常");
217             emit scanFinished();
218         }
219     }
220 
221 
222     /**
223     * @brief 当发现一个主机时调用，更新UI显示
224     * @param host 发现的主机信息
225     */
226     void MainWindow::onHostFound(const HostInfo &host)
227     {
228         int row = m_resultsTable->rowCount();
229         m_resultsTable->insertRow(row);
230 
231         m_resultsTable->setItem(row, 0, new QTableWidgetItem(host.
    ipAddress));
232         m_resultsTable->setItem(row, 1, new QTableWidgetItem(host.
    hostName));
233         m_resultsTable->setItem(row, 2, new QTableWidgetItem(host.
    macAddress));
234         m_resultsTable->setItem(row, 3, new QTableWidgetItem(host.
    macVendor));
235         m_resultsTable->setItem(row, 4, new QTableWidgetItem(host.
    isReachable ? "可达" : "不可达"));
236 
237         QStringList openPortsList;
238         QMapIterator<int, bool> i(host.openPorts);
239         while (i.hasNext()) {
240             i.next();
241             if (i.value()) {
242                 openPortsList << QString::number(i.key());
243             }
244         }
245         QString openPorts = openPortsList.join(", ");
246         m_resultsTable->setItem(row, 5, new QTableWidgetItem(openPorts));
247 
248         if (host.isReachable) {
249             for (int col = 0; col < 6; ++col) {
250                 m_resultsTable->item(row, col)->setBackground(QColor(200,
     255, 200));
251             }
252         }
253 
254         m_hostsFound++;
255         m_statusBar->showMessage(QString("已发现 %1 台主机").arg(
    m_hostsFound));
256     }
257 
```

```
258
```

Listing 13: MainWindow 扫描启动流程

4.4 设备分析模块

4.4.1 核心功能

设备类型识别

多维度智能推断机制：设备类型识别综合运用了多种启发式规则，包括基于 IP 地址特征（例如，以.1 或.254 结尾的 IP 地址通常被假定为路由器）、主机名关键词匹配（例如，主机名中包含"server" 字符串的设备倾向于被识别为服务器）、以及从 MAC 地址解析出的制造商信息（例如，OUI 显示为"HP" 且主机名包含"Printer" 的设备被判定为打印机）。预定义设备类型集：系统内置了一组常见的网络设备类型，包括：路由器 (Router)、服务器 (Server)、个人计算机 (PC)、移动设备 (Mobile Device)、打印机 (Printer) 及各类智能设备 (IoT Device)。

数据统计与图表生成

三大核心图表：

- 设备类型分布图（饼图）：该图表以饼图形式直观展示网络中各类已识别设备的数量占比。不同设备类型将通过独特的颜色进行区分（例如，路由器可能用红色扇区表示，服务器用蓝色扇区表示），并附带图例说明。
- 设备制造商分布图（饼图或柱状图）：此图表旨在揭示网络中设备的主要制造商构成。通常会突出显示排名前 N（例如前 5）的制造商及其设备数量，而其余规模较小的制造商则可能被合并归类为" 其他" 项。
- 开放端口统计图（柱状图）：该图表重点统计网络中常见或关键服务端口（如 SSH 端口 22、HTTP 端口 80、HTTPS 端口 443 等）在所有已发现设备上的开放实例总数，帮助管理员了解哪些服务在网络中最为普遍。

图表动态更新与动画效果：所有统计图表均能在每次扫描任务完成后自动刷新其数据内容。为提升视觉体验，图表在数据更新时可支持平滑的动画过渡效果。

安全风险分析

高风险端口检测：

- 重点监控的高风险端口列表：系统内置一个可配置的高风险服务端口列表，默认包括：21 (FTP), 22 (SSH), 23 (Telnet), 3389 (RDP), 445 (SMB) 等，这些端口若不当暴露，常成为攻击者的首要目标。
- 高风险设备识别与报告：系统能够自动扫描并识别出那些开放了上述高危端口的设备，并将这些发现汇总到安全风险报告中，提请管理员注意。

自动化安全建议：针对已识别的风险，系统能够根据端口类型和潜在威胁，自动生成初步的、具有针对性的安全防护建议，例如提示管理员限制对特定远程端口的访问、建议禁用已知的未加密服务等。

数据管理

数据清理与重置功能：为方便用户管理，分析模块支持一键清除所有历史分析数据和图表缓存，以便为新的扫描结果或分析任务释放显示空间和计算资源。信号驱动的异步更新机制：当设备分析模块完成其数据处理和统计计算后，会发出一个特定的信号（例如 analysisCompleted）。界面层（UI）通过连接到此信号的槽函数来异步接收通知，并据此触发相关图表的更新操作，确保了分析过程与界面响应的解耦。

4.4.2 核心代码解析

图表创建逻辑 图表组件的创建与管理：本模块利用 Qt Charts 提供的核心类（如 QChart 作为图表容器，QPieSeries 用于生成饼图系列，QBarSeries 用于生成柱状图系列等）来实现数据的可视化呈现。在更新图表数据时，为确保数据的准确性和避免状态混淆，部分场景可能采用” 先销毁旧图表对象再创建新图表对象” 的模式，或者更精细地清除系列数据再重新填充，以保证图表始终反映最新的分析结果。

```
    /**
    * @brief 创建设备类型图表
    */
    void DeviceAnalyzer::createDeviceTypeChart()
    {
        if (m_deviceTypeChart) {
            delete m_deviceTypeChart;
        }

        m_deviceTypeChart = new QChart();
        m_deviceTypeChart->setTitle("设备类型分布");
        m_deviceTypeChart->setAnimationOptions(QChart::SeriesAnimations);

        // 创建饼图系列
        if (m_deviceTypeSeries) {
            delete m_deviceTypeSeries;
        }

        m_deviceTypeSeries = new QPieSeries();
        m_deviceTypeChart->addSeries(m_deviceTypeSeries);
        m_deviceTypeChart->legend()->setAlignment(Qt::AlignRight);
    }

```

Listing 14: DeviceAnalyzer 图表创建逻辑

图表数据更新 图表数据填充逻辑：该过程首先遍历所有被识别为可达的网络设备。对每个设备，调用 determineDeviceType 函数（或类似方法）来推断其具体类型。随后，使用一个 QMap<QString, int> 容器来累积统计每种设备类型的数量。统计完成后，这些数据被用于更新饼图（QPieSeries）的各个扇区及其标签。为了在视觉上区分不同的设备类型，系统会为每个类型分配一个预设的（或动态生成的）颜色值，这些颜色值可以直接应用于对应的 QPieSlice 对象。

```
    /**
    * @brief 分析扫描结果并更新统计图表
    * @param hosts 主机信息列表
```

```
4       */
5       void DeviceAnalyzer::analyzeHosts(const QList<HostInfo> &hosts)
6       {
7           // 清除旧数据
8           clear();
9
10          m_totalHosts = hosts.size();
11
12          // 设备类型统计
13          QMap<QString, int> deviceTypes;
14
15          // 端口统计
16          QMap<int, int> portCounts;
17          QList<int> commonPorts = {21, 22, 23, 25, 53, 80, 443, 3306,
    3389, 8080};
18
19          // 厂商统计
20          QMap<QString, int> vendorCounts;
21
22          // 分析每台主机
23          for (const HostInfo &host : hosts) {
24              if (host.isReachable) {
25                  m_reachableHosts++;
26
27                  // 设备类型统计
28                  QString deviceType = determineDeviceType(host);
29                  deviceTypes[deviceType]++;
30
31                  // 端口统计
32                  QMapIterator<int, bool> i(host.openPorts);
33                  while (i.hasNext()) {
34                      i.next();
35                      if (i.value() && commonPorts.contains(i.key())) {
36                          portCounts[i.key()]++;
37                      }
38                  }
39
40                  // 厂商统计
41                  QString vendor = host.macVendor;
42                  if (vendor.isEmpty()) {
43                      vendor = "未知厂商";
44                  }
45                  vendorCounts[vendor]++;
46              }
47          }
48
49          // 更新设备类型图表
50          if (m_deviceTypeSeries) {
51              m_deviceTypeSeries->clear();
52              QMapIterator<QString, int> dt(deviceTypes);
53              while (dt.hasNext()) {
54                  dt.next();
55                  if (dt.value() > 0) {
56                      QString label = QString("%1 (%2)").arg(dt.key()).arg(
```

```
    dt.value());
57                  QPieSlice *slice = m_deviceTypeSeries->append(label,
    dt.value());
58
59                  // 设置不同颜色
60                  if (dt.key() == "路由器") {
61                      slice->setColor(QColor(255, 0, 0));
62                  } else if (dt.key() == "服务器") {
63                      slice->setColor(QColor(0, 0, 255));
64                  } else if (dt.key() == "个人电脑") {
65                      slice->setColor(QColor(0, 128, 0));
66                  } else if (dt.key() == "移动设备") {
67                      slice->setColor(QColor(255, 165, 0));
68                  } else if (dt.key() == "打印机") {
69                      slice->setColor(QColor(128, 0, 128));
70                  } else if (dt.key() == "智能设备") {
71                      slice->setColor(QColor(0, 128, 128));
72                  }
73
74                  slice->setLabelVisible();
75              }
76          }
77      }
78
79      // 更新端口图表
80      if (m_portSeries) {
81          // 先删除旧的数据集
82          while (!m_portSeries->barSets().isEmpty()) {
83              m_portSeries->remove(m_portSeries->barSets().first());
84          }
85
86          // 创建新的数据集
87          QBarSet *portSet = new QBarSet("端口");
88
89          // 设置坐标轴类别
90          QStringList categories;
91
92          // 添加端口数据
93          int maxCount = 1; // 至少为1，避免除零错误
94          for (int port : commonPorts) {
95              int count = portCounts.value(port, 0);
96              portSet->append(count);
97              categories << QString::number(port);
98              if (count > maxCount) {
99                  maxCount = count;
100             }
101         }
102
103         m_portSeries->append(portSet);
104
105         // 更新坐标轴
106         QBarCategoryAxis *axisX = qobject_cast<QBarCategoryAxis*>(
    m_portDistributionChart->axes(Qt::Horizontal).first());
107         if (axisX) {
```

```
108                 axisX->setCategories(categories);
109             }
110 
111             QValueAxis *axisY = qobject_cast<QValueAxis*>(
    m_portDistributionChart->axes(Qt::Vertical).first());
112             if (axisY) {
113                 axisY->setRange(0, maxCount + 1);
114             }
115         }
116 
117         // 更新厂商图表
118         if (m_vendorSeries) {
119             m_vendorSeries->clear();
120             QMapIterator<QString, int> vc(vendorCounts);
121 
122             // 只显示前5个主要厂商，其余归为"其他"
123             int count = 0;
124             int otherCount = 0;
125 
126             while (vc.hasNext()) {
127                 vc.next();
128                 if (count < 5) {
129                     QString label = QString("%1 (%2)").arg(vc.key()).arg(
    vc.value());
130                     QPieSlice *slice = m_vendorSeries->append(label, vc.
    value());
131                     slice->setLabelVisible();
132                     count++;
133                 } else {
134                     otherCount += vc.value();
135                 }
136             }
137 
138             if (otherCount > 0) {
139                 QPieSlice *slice = m_vendorSeries->append(QString("其他
    (%1)").arg(otherCount), otherCount);
140                 slice->setLabelVisible();
141             }
142         }
143 
144         emit analysisCompleted();
145     }
146 
```

Listing 15: DeviceAnalyzer 图表数据更新

设备类型识别 设备类型推断策略：设备类型的初步推断主要依赖于对设备信息（如主机名、MAC 制造商信息）中的特定关键词进行字符串匹配（通常不区分大小写）。为解决一个设备可能同时满足多种类型特征的问题，系统采用了一套预设的优先级规则进行决策，其大致顺序为：路由器具有最高优先级，其次是服务器，接着是打印机、移动设备、其他智能设备，最后是通用个人电脑。这意味着如果一个设备同时具备路由器和服务器的某些特征，它将被优先判定为路由器。

```
    /**
    * @brief 根据主机信息推断设备类型。
    * @param host 主机信息。
    * @return DeviceType 推断出的设备类型。
    * @details 基于IP地址后缀、主机名和MAC厂商中的关键词进行推断。
    */
    DeviceType NetworkTopologyView::determineDeviceType(const HostInfo &
host)
    {
        // 根据IP地址、主机名、MAC厂商等信息推断设备类型
        QString ipLower = host.ipAddress.toLower();
        QString hostLower = host.hostName.toLower();
        QString vendorLower = host.macVendor.toLower();

        // 检测路由器
        if (vendorLower.contains("路由") ||
            vendorLower.contains("router") ||
            hostLower.contains("router") ||
            hostLower.contains("gateway") ||
            ipLower.endsWith(".1") ||
            ipLower.endsWith(".254")) {
            return DEVICE_ROUTER;
        }

        // 检测服务器
        if (hostLower.contains("server") ||
            vendorLower.contains("server") ||
            vendorLower.contains("dell") ||
            vendorLower.contains("ibm") ||
            vendorLower.contains("hp") && !vendorLower.contains("printer"
)) {
            return DEVICE_SERVER;
        }

        // 检测打印机
        if (hostLower.contains("print") ||
            vendorLower.contains("print") ||
            vendorLower.contains("canon") ||
            vendorLower.contains("hp") && vendorLower.contains("printer")
 ||
            vendorLower.contains("epson")) {
            return DEVICE_PRINTER;
        }

        // 检测移动设备
        if (vendorLower.contains("apple") ||
            vendorLower.contains("iphone") ||
            vendorLower.contains("ipad") ||
            vendorLower.contains("android") ||
            vendorLower.contains("xiaomi") ||
            vendorLower.contains("huawei") ||
            vendorLower.contains("oppo") ||
            vendorLower.contains("vivo")) {
            return DEVICE_MOBILE;
```

```
        }

        // 检测IoT设备
        if (hostLower.contains("esp") ||
            hostLower.contains("arduino") ||
            hostLower.contains("raspberry") ||
            hostLower.contains("iot") ||
            vendorLower.contains("nest") ||
            vendorLower.contains("smart")) {
            return DEVICE_IOT;
        }

        // 默认为PC
        return DEVICE_PC;
    }

```

Listing 16: DeviceAnalyzer **设备类型识别** (refined)

安全报告生成 安全报告生成流程：系统内部维护一个 QMap<int, QStringList> 结构（或类似数据结构），其中键为预定义的高风险端口号，值为包含该端口名称及其潜在风险描述的字符串列表。这种设计便于后续对风险端口库的维护与扩展。在生成报告时，程序会执行一个双重遍历：外层遍历所有已发现的主机，内层遍历每个主机的开放端口列表，并与风险端口库进行比对。一旦发现某主机开放了风险端口，该主机及其相关风险信息（IP 地址、开放的风险端口号、端口名称、具体风险描述）将被记录到一个 riskyHosts 映射或列表中。最终生成的安全报告将清晰列出这些存在风险的主机及其详情，并在报告末尾附上一系列通用的安全加固建议。

```
    /**
    * @brief 创建安全风险报告
    * @param hosts 主机信息列表
    * @return 安全报告字符串
    */
    QString DeviceAnalyzer::generateSecurityReport(const QList<HostInfo>
&hosts)
    {
        QString report = "网络安全分析报告\n";
        report += "====================\n\n";

        // 统计主机数量
        report += QString("发现总设备数: %1\n").arg(m_totalHosts);
        report += QString("可访问设备数: %1\n").arg(m_reachableHosts);
        report += QString("不可访问设备数: %1\n\n").arg(m_totalHosts -
m_reachableHosts);

        // 检查高风险端口
        QMap<int, QStringList> riskPorts;
        riskPorts[21] = QStringList() << "FTP" << "文件传输协议，明文传输
凭据，存在安全风险";
        riskPorts[22] = QStringList() << "SSH" << "远程管理端口，应限制访
问";
        riskPorts[23] = QStringList() << "Telnet" << "未加密的远程管理，
极高安全风险";
```

```
21          riskPorts[3389] = QStringList() << "RDP" << "远程桌面协议，存在多种漏洞";
22          riskPorts[445] = QStringList() << "SMB" << "文件共享服务，曾存在多种漏洞";
23
24          // 检查风险端口开放情况
25          QMap<QString, QStringList> hostsWithRiskPorts;
26
27          for (const HostInfo &host : hosts) {
28              if (!host.isReachable) continue;
29
30              for (auto it = riskPorts.begin(); it != riskPorts.end(); ++it) {
31                  int port = it.key();
32
33                  if (host.openPorts.contains(port) && host.openPorts[port]) {
34                      hostsWithRiskPorts[host.ipAddress].append(
35                          QString("%1 (%2)").arg(port).arg(it.value()[0])
36                      );
37                  }
38              }
39          }
40
41          // 生成风险报告
42          if (hostsWithRiskPorts.isEmpty()) {
43              report += "未发现高风险端口开放\n\n";
44          } else {
45              report += "发现以下设备开放了高风险端口：\n";
46
47              for (auto it = hostsWithRiskPorts.begin(); it != hostsWithRiskPorts.end(); ++it) {
48                  QString ip = it.key();
49                  QStringList ports = it.value();
50
51                  // 查找主机名
52                  QString hostname = "未知";
53                  for (const HostInfo &host : hosts) {
54                      if (host.ipAddress == ip) {
55                          hostname = host.hostName;
56                          break;
57                      }
58                  }
59
60                  report += QString("设备：%1 (%2)\n").arg(ip).arg(hostname);
61                  report += QString("开放的高风险端口：%1\n").arg(ports.join(", "));
62                  report += "------------\n";
63              }
64
65              report += "\n";
66          }
67
```

```
68          // 添加建议措施
69          report += "安全建议: \n";
70          report += "1. 限制远程管理端口(22, 23, 3389)的访问范围\n";
71          report += "2. 禁用未加密的服务(如Telnet, FTP)\n";
72          report += "3. 对必要的服务设置强密码和访问控制\n";
73          report += "4. 及时更新系统和应用补丁\n";
74          report += "5. 对重要设备设置防火墙规则\n";
75
76          return report;
77      }
78
```

Listing 17: DeviceAnalyzer 安全报告生成

4.5 扫描历史模块

4.5.1 核心功能

会话管理

扫描会话持久化存储：本模块的核心功能之一是能够完整保存每一次网络扫描任务的结果。这包括捕获到的所有设备信息（如 IP 地址、MAC 地址、主机名、开放端口列表等），同时自动记录下该次扫描的执行时间戳以及用户可能提供的描述性信息，共同构成一个独立的” 扫描会话”。全面的会话管理接口：为方便用户操作和管理历史数据，模块提供了一系列标准接口，包括：`addSession` 用于将一次新的扫描结果添加为历史会话；`getSession` 允许通过索引或其他唯一标识符来检索特定的历史会话数据；`removeSession` 用于删除选定的单个或多个历史会话；以及 `clearHistory` 功能，可以一键清空所有已存储的扫描历史记录。

扫描结果比较

网络设备动态变化检测：模块具备对比分析任意两次历史扫描会话结果的能力。通过比较这两次扫描所获取的设备列表，系统能够准确识别出在两次扫描间隔期间网络中新增的设备以及已离线或被移除的设备。比较结果通常以结构化的形式返回，例如使用 `QPair<QList<HostInfo>, QList<HostInfo>>` 来分别存储新增设备列表和消失设备列表。典型应用场景：此功能对于网络管理员而言具有重要的实用价值，例如，可以用于长期监控网络中设备的增减动态，辅助追踪资产变更；更重要的是，它可以帮助及时发现网络中可能存在的未经授权便接入的新设备，从而增强网络安全态势感知能力。

数据持久化

基于 JSON 的数据持久化：为了实现扫描历史记录的长期存档和跨平台共享，模块支持将完整的扫描历史数据（包括所有会话及其详细设备信息）序列化为 JSON (JavaScript Object Notation) 格式，并通过 `saveToFile` 接口将其保存到本地文件系统。相应地，`loadFromFile` 接口则负责从指定的 JSON 文件中反序列化数据，恢复先前的扫描历史记录。

统计分析

单次扫描会话的概要统计：在 `ScanSession` 类（或类似的数据结构）内部，除了存储原始的设备扫描数据外，通常还会包含一些对该次扫描结果的即时统计信息。例

如，可以计算并存储本次扫描中发现的可达设备总数（如通过 reachableHosts() 方法获取），以及对所有可达设备上开放端口的分布情况进行初步统计（如通过 portDistribution() 方法获取各端口的开放次数）。

4.5.2 核心代码解析

添加新扫描会话 历史会话存储与管理机制：扫描历史记录在内部采用 QList<ScanSession> 数据结构进行存储，其中每个 ScanSession 对象代表一次完整的扫描作业。为方便用户快速访问最近的扫描结果，新添加的会话通常被置于列表的头部（即 prepend 操作）。如果用户在保存会话时未提供自定义描述，系统会自动生成一个基于当前时间戳的默认描述。当历史记录发生任何变更（如添加、删除会话）时，模块会发出相应的信号（例如 historyChanged()），以通知用户界面（UI）或其他相关组件进行响应式更新。

```
1     /**
2     * @brief 添加一次新的扫描会话到历史记录。
3     * @param hosts 本次扫描发现的主机列表。
4     * @param description (可选) 会话的描述，默认为当前日期时间。
5     * @details 创建一个新的 ScanSession 对象并添加到 m_sessions 列表中。
6     *          如果描述为空，则使用当前日期时间作为描述。
7     *          会触发 historyChanged() 信号。
8     */
9     void ScanHistory::addSession(const QList<HostInfo> &hosts, const
    QString &description)
10    {
11        QString sessionDesc = description;
12        if (sessionDesc.isEmpty()) {
13            sessionDesc = QDateTime::currentDateTime().toString("yyyy-MM-
    dd hh:mm:ss 扫描");
14        }
15
16        m_sessions.append(ScanSession(sessionDesc, hosts, hosts.size()));
17        emit historyChanged();
18    }
19
```

Listing 18: ScanHistory 添加新扫描会话

设备变化检测逻辑 设备变化的高效检测算法：为实现两次扫描会话间的设备变化检测，算法首先将两次扫描的设备列表分别转换为以 IP 地址为键、HostInfo 对象为值的 QMap 结构。这样做可以将后续的查找操作优化到接近 $O(1)$ 的平均时间复杂度。随后，通过对这两个 QMap 的键集合进行比较（本质上是集合的差集运算），可以高效地识别出在一个会话中存在但在另一个会话中缺失的 IP 地址，从而确定新增或消失的设备。整个比较过程的时间复杂度大致为 $O(n+m)$，其中 n 和 m 分别代表两次扫描所发现的设备数量，这在实践中是相当高效的。

```
1     /**
2     * @brief 比较两次扫描会话的结果，找出新增和消失的主机。
3     * @param index1 第一个会话的索引（通常是较早的会话）。
4     * @param index2 第二个会话的索引（通常是较新的会话）。
5     * @return QPair<QList<HostInfo>, QList<HostInfo>>
```

```
     *          - first: 在会话2中出现但未在会话1中出现的主机（新增主机）。
     *          - second: 在会话1中出现但未在会话2中出现的主机（消失主机）。
     * @throws std::out_of_range 如果索引无效。
     */
    QPair<QList<HostInfo>, QList<HostInfo>> ScanHistory::compareScans(int
     index1, int index2) const
    {
        if (index1 < 0 || index1 >= m_sessions.size() ||
            index2 < 0 || index2 >= m_sessions.size() || index1 == index2
    ) {
            throw std::out_of_range("无效的扫描会话索引");
        }

        const ScanSession &session1 = m_sessions.at(index1);
        const ScanSession &session2 = m_sessions.at(index2);

        QList<HostInfo> newHosts;     // 在session2中出现，但在session1中
    未出现
        QList<HostInfo> missingHosts; // 在session1中出现，但在session2中
    未出现

        // 将session1中的主机IP存入QSet以便快速查找
        QSet<QString> session1IPs;
        for (const HostInfo &host : session1.hosts) {
            session1IPs.insert(host.ipAddress);
        }

        // 遍历session2，查找新增主机
        for (const HostInfo &host2 : session2.hosts) {
            if (!session1IPs.contains(host2.ipAddress)) {
                newHosts.append(host2);
            }
        }

        // 将session2中的主机IP存入QSet以便快速查找
        QSet<QString> session2IPs;
        for (const HostInfo &host : session2.hosts) {
            session2IPs.insert(host.ipAddress);
        }

        // 遍历session1，查找消失主机
        for (const HostInfo &host1 : session1.hosts) {
            if (!session2IPs.contains(host1.ipAddress)) {
                missingHosts.append(host1);
            }
        }

        return qMakePair(newHosts, missingHosts);
    }

```

Listing 19: ScanHistory 设备变化检测逻辑

5 总结

本项目基于 Qt 框架，设计并实现了一款功能完善的网络设备扫描与分析工具。该系统集成了自动子网发现、自定义 IP 范围扫描、主机可达性判定（基于 TCP 连接）、以及多种模式下的端口扫描等核心功能，能够有效识别局域网中的在线设备及其服务信息。通过采用高效的并行扫描机制与合理的线程池管理策略，系统在不同规模网络环境下均能保持良好的扫描效率和稳定的运行表现。

在实现过程中，项目对多线程任务进行了精细化管理，并结合灵活的超时控制机制，确保了在设备数量较多的网络环境中依然能够快速、准确地完成扫描任务。同时，系统支持对扫描结果的线程安全存取，并提供实时进度更新与阶段性成果展示，使用户能够获得清晰、连续的交互体验。

本工具具备模块化结构、良好的可扩展性与高度的可定制能力，可适应多种复杂网络环境的需求。系统功能完整、运行稳定，具有较强的实用价值和应用前景。

综上所述，本项目不仅完成了一款实用的网络扫描工具的设计与实现，也在系统结构设计、并发处理、网络通信等方面积累了宝贵经验，为后续开展网络安全审计、性能优化分析或资产管理等应用提供了良好的技术基础与实践支撑。