实验一：本地机上的聊天室

1. 实验内容

在linux平台上使用守护进程实现服务器，用于支持宿主机和虚拟机上的客户端进行聊天。客户端使用图形界面实现。

1. 实验准备

服务器端：运行在腾讯云Linux虚拟机上的守护进程

客户端：运行在本地主机的GUI程序

通信协议：TCP协议，使用8888端口，结合虚拟机IP为139.155.27

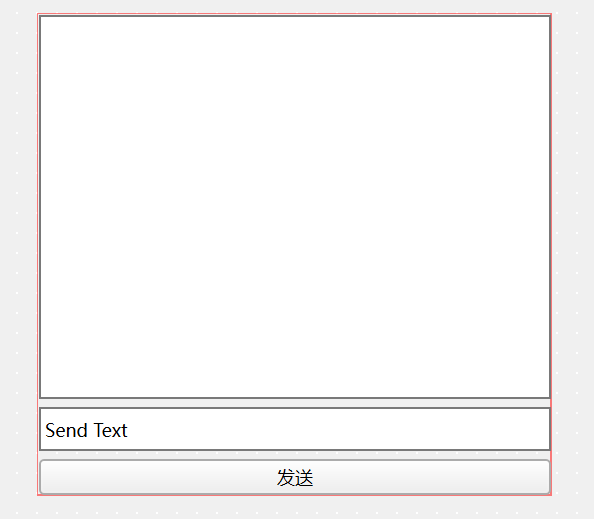
1. 服务器端实现
2. 创建守护进程
3. #include <sys/types.h>
4. #include <sys/stat.h>
5. #include <unistd.h>
6. void daemonize() {
7. pid\_t pid = fork();
8. if (pid < 0) exit(EXIT\_FAILURE);
9. if (pid > 0) exit(EXIT\_SUCCESS); *// 父进程退出*
11. setsid();       *// 创建新会话*
12. chdir("/");     *// 切换工作目录*
13. umask(0);       *// 重置文件权限掩码*
15. *// 关闭标准文件描述符*
16. close(STDIN\_FILENO);
17. close(STDOUT\_FILENO);
18. close(STDERR\_FILENO);
19. }
20. 核心服务器代码
21. server.hpp
22. *// server.hpp*
23. #pragma once
24. #include <sys/stat.h>
25. #include <fcntl.h>
26. #include <unistd.h>
27. #include <cstdlib>
28. #include <sys/socket.h>
29. #include <netinet/in.h>
30. #include <arpa/inet.h>
31. #include <sys/epoll.h>
32. #include <unordered\_map>
33. #include <string>
34. #include <stdexcept>
35. #include <system\_error>
36. class Daemonizer {
37. public:
38. static void daemonize();
39. };
40. class ChatServer {
41. public:
42. static constexpr int MAX\_EVENTS = 100;  *// 定义为公共静态常量*
43. static constexpr int PORT = 8888;
44. private:
45. void initNetwork();
46. void handleEvents();
47. int serverFd = -1;
48. int epollFd = -1;
49. std::unordered\_map<int, std::string> clients;
50. void broadcast(int senderFd, const char\* msg, ssize\_t len);
51. void handleNewConnection();
52. public:
53. void start();
54. };

(2) server.cpp

1. #include "server.hpp"
2. #include <cstring>
3. #include <iostream>
4. #include <stdexcept>
5. #include <system\_error>
6. #include <thread>
7. #include <chrono>
8. *// ==================== Daemonizer 实现 ====================*
9. void Daemonizer::daemonize() {
10. *// 第一次fork*
11. if (pid\_t pid = fork(); pid < 0) {
12. throw std::system\_error(errno, std::generic\_category(), "First fork failed");
13. } else if (pid > 0) {
14. exit(EXIT\_SUCCESS);
15. }
16. *// 创建新会话*
17. if (setsid() < 0) {
18. throw std::system\_error(errno, std::generic\_category(), "setsid failed");
19. }
20. *// 第二次fork*
21. if (pid\_t pid = fork(); pid < 0) {
22. throw std::system\_error(errno, std::generic\_category(), "Second fork failed");
23. } else if (pid > 0) {
24. exit(EXIT\_SUCCESS);
25. }
26. *// 关闭所有文件描述符*
27. for (int fd = sysconf(\_SC\_OPEN\_MAX); fd >= 0; --fd) {
28. close(fd);
29. }
30. *// 设置文件权限掩码*
31. umask(0);
32. *// 设置工作目录*
33. if (chdir("/") < 0) {
34. throw std::system\_error(errno, std::generic\_category(), "chdir failed");
35. }
36. *// 重定向标准IO*
37. int nullfd = open("/dev/null", O\_RDWR);
38. if (nullfd == -1) {
39. throw std::system\_error(errno, std::generic\_category(), "open /dev/null failed");
40. }
41. dup2(nullfd, STDIN\_FILENO);
42. dup2(nullfd, STDOUT\_FILENO);
43. dup2(nullfd, STDERR\_FILENO);
44. close(nullfd);
45. }
46. *// ==================== ChatServer 实现 ====================*
47. void ChatServer::broadcast(int senderFd, const char\* msg, ssize\_t len) {
48. std::cout << "Broadcasting message to " << clients.size() << " clients" << std::endl;
49. for (auto& [fd, \_] : clients) {
50. if (fd != senderFd) {
51. if (write(fd, msg, len) == -1) {
52. perror("Write error");
53. close(fd);
54. clients.erase(fd);
55. }
56. }
57. }
58. }
59. void ChatServer::handleNewConnection() {
60. sockaddr\_in clientAddr{};
61. socklen\_t len = sizeof(clientAddr);
63. int clientFd = accept(serverFd, reinterpret\_cast<sockaddr\*>(&clientAddr), &len);
64. if (clientFd == -1) {
65. perror("Accept error");
66. return;
67. }
68. *// 设置非阻塞模式*
69. int flags = fcntl(clientFd, F\_GETFL, 0);
70. if (flags == -1 || fcntl(clientFd, F\_SETFL, flags | O\_NONBLOCK) == -1) {
71. perror("Set non-block failed");
72. close(clientFd);
73. return;
74. }
75. *// 添加epoll监听*
76. epoll\_event ev{};
77. ev.events = EPOLLIN | EPOLLET;
78. ev.data.fd = clientFd;
79. if (epoll\_ctl(epollFd, EPOLL\_CTL\_ADD, clientFd, &ev) == -1) {
80. perror("Epoll add error");
81. close(clientFd);
82. return;
83. }
84. *// 记录客户端信息*
85. clients[clientFd] = inet\_ntoa(clientAddr.sin\_addr);
86. }
87. void ChatServer::initNetwork() {
88. *// 创建TCP套接字*
89. serverFd = socket(AF\_INET, SOCK\_STREAM | SOCK\_NONBLOCK, 0);
90. if (serverFd == -1) {
91. throw std::system\_error(errno, std::generic\_category(), "Socket creation failed");
92. }
93. *// 绑定地址*
94. sockaddr\_in addr{};
95. addr.sin\_family = AF\_INET;
96. addr.sin\_port = htons(PORT);
97. addr.sin\_addr.s\_addr = INADDR\_ANY;
98. if (bind(serverFd, reinterpret\_cast<sockaddr\*>(&addr), sizeof(addr)) == -1) {
99. throw std::system\_error(errno, std::generic\_category(), "Bind failed");
100. }
101. *// 开始监听*
102. if (listen(serverFd, SOMAXCONN) == -1) {
103. throw std::system\_error(errno, std::generic\_category(), "Listen failed");
104. }
105. }
106. void ChatServer::handleEvents() {
107. epoll\_event events[MAX\_EVENTS];
108. int n = epoll\_wait(epollFd, events, MAX\_EVENTS, -1);
109. if (n == -1) {
110. if (errno == EINTR) return;
111. throw std::system\_error(errno, std::generic\_category(), "Epoll wait error");
112. }
113. for (int i = 0; i < n; ++i) {
114. if (events[i].data.fd == serverFd) {
115. handleNewConnection();
116. } else {
117. *// 处理客户端消息*
118. char buffer[4096];
119. ssize\_t len = read(events[i].data.fd, buffer, sizeof(buffer));
120. if (len <= 0) {
121. close(events[i].data.fd);
122. clients.erase(events[i].data.fd);
123. } else {
124. broadcast(events[i].data.fd, buffer, len);
125. }
126. }
127. }
128. }
129. void ChatServer::start() {
130. try {
131. Daemonizer::daemonize();
132. initNetwork();
134. *// 添加初始化成功日志*
135. syslog(LOG\_INFO, "服务启动成功，监听端口：%d", PORT);
137. while (true) {
138. handleEvents();
139. std::this\_thread::sleep\_for(std::chrono::milliseconds(100));
140. }
141. } catch (const std::exception& e) {
142. *// 将错误记录到系统日志*
143. syslog(LOG\_ERR, "致命错误：%s", e.what());
144. exit(EXIT\_FAILURE);
145. }
146. }
147. Main函数
148. #include "server.hpp"
149. #include <iostream>
150. int main() {
151. try {
152. ChatServer server;
153. server.start();
154. } catch (const std::exception& e) {
155. std::cerr << "Server fatal error: " << e.what() << std::endl;
156. return EXIT\_FAILURE;
157. }
158. return EXIT\_SUCCESS;
159. }
160. 客户端实现
161. 界面设计

QTextBrowser 用于显示消息

QLineEdit 用于输入消息

QPushButton 发送按钮

1. 配置网络通信（CMakeLists.txt）
2. cmake\_minimum\_required(VERSION 3.19)
3. project(untitled1 LANGUAGES CXX)
4. # find\_package(Qt6 6.5 REQUIRED COMPONENTS Core Widgets)
5. find\_package(Qt6 6.5 REQUIRED COMPONENTS Core Widgets Network)
6. qt\_standard\_project\_setup()
7. qt\_add\_executable(untitled1
8. WIN32 MACOSX\_BUNDLE
9. main.cpp
10. mainwindow.cpp
11. mainwindow.h
12. mainwindow.ui
13. )
14. target\_link\_libraries(untitled1
15. PRIVATE
16. Qt::Core
17. Qt::Widgets
18. Qt::Network
19. )
20. include(GNUInstallDirs)
21. install(TARGETS untitled1
22. BUNDLE  DESTINATION .
23. RUNTIME DESTINATION ${CMAKE\_INSTALL\_BINDIR}
24. LIBRARY DESTINATION ${CMAKE\_INSTALL\_LIBDIR}
25. )
26. qt\_generate\_deploy\_app\_script(
27. TARGET untitled1
28. OUTPUT\_SCRIPT deploy\_script
29. NO\_UNSUPPORTED\_PLATFORM\_ERROR
30. )
31. install(SCRIPT ${deploy\_script})
32. mainwindow.h
33. #ifndef MAINWINDOW\_H
34. #define MAINWINDOW\_H
35. #include <QMainWindow>
36. #include <QTcpSocket>
37. #include "ui\_mainwindow.h"
38. class MainWindow : public QMainWindow {
39. Q\_OBJECT
40. public:
41. explicit MainWindow(QWidget \*parent = nullptr);
42. ~MainWindow();
43. *// 组件命名规范（必须与UI文件中的objectName一致）*
44. private:
45. Ui::MainWindow \*ui;          *// UI容器（固定名称）*
46. QTcpSocket \*tcpSocket;       *// 网络套接字组件*
47. *// 信号与槽声明*
48. private slots:
49. void onConnected();          *// 连接成功槽*
50. void onReadyRead();          *// 数据接收槽*
51. void onSendButtonClicked();  *// 发送按钮槽*
52. };
53. #endif *// MAINWINDOW\_H*
54. main.cpp
55. #include <QApplication>
56. #include <QTcpSocket>
57. #include "mainwindow.h"
58. int main(int argc, char \*argv[])
59. {
60. QApplication a(argc, argv);
61. MainWindow w;
62. w.show();
63. return a.exec();
64. }
65. mainwindow.cpp
66. #include "mainwindow.h"
67. #include "ui\_mainwindow.h"
68. *// 必须包含的组件头文件*
69. #include <QDebug>
70. #include <QMessageBox>
71. MainWindow::MainWindow(QWidget \*parent)
72. : QMainWindow(parent)
73. , ui(new Ui::MainWindow)
74. , tcpSocket(new QTcpSocket(this))  *// 初始化网络组件*
75. {
76. ui->setupUi(this);
77. *// 组件信号连接（必须匹配objectName）*
78. connect(tcpSocket, &QTcpSocket::connected, this, &MainWindow::onConnected);
79. connect(tcpSocket, &QTcpSocket::readyRead, this, &MainWindow::onReadyRead);
80. connect(ui->sendButton, &QPushButton::clicked, this, &MainWindow::onSendButtonClicked);
81. *// 连接服务器*
82. tcpSocket->connectToHost("139.155.27.201", 8888);
83. }
84. void MainWindow::onConnected() {
85. ui->textBrowser->append("已连接到服务器");
86. }
87. void MainWindow::onReadyRead() {
88. QByteArray data = tcpSocket->readAll();
89. qDebug() << "Received data:" << data;
90. ui->textBrowser->append(QString::fromUtf8(data));
91. }
92. void MainWindow::onSendButtonClicked() {
93. QString msg = ui->lineEdit->text();
94. if(!msg.isEmpty()) {
95. tcpSocket->write(msg.toUtf8());
96. ui->lineEdit->clear();
97. }
98. }
99. MainWindow::~MainWindow() {
100. delete ui;
101. }
102. 部署步骤
103. 添加安全组，放行端口8888
104. 启动服务
105. *# 进入编译目录*
106. cd ~/chat\_server/bin
107. *# 启动服务端（前台运行，查看实时输出）*
108. ./chatserver
109. *# 后台启动（添加 & 符号）*
110. ./chatserver &
111. 验证端口监听​
112. sudo netstat -tulnp | grep 8888

4．部署生产环境

（1）创建systemd服务文件​

sudo nano /etc/systemd/system/chatserver.service

（2）chatserver.service

[Unit]

Description=Chat Server Daemon

After=network.target

[Service]

Type=simple

User=lighthouse

Group=lighthouse

WorkingDirectory=/home/lighthouse/chat\_server/bin

ExecStart=/home/lighthouse/chat\_server/bin/chatserver

Restart=always

*# 日志配置*

StandardOutput=syslog

StandardError=syslog

SyslogIdentifier=chatserver

[Install]

WantedBy=multi-user.target

（3）启动服务

sudo systemctl daemon-reload

sudo systemctl enable --now chatserver

1. 本地 Qt编译运行
2. 连接服务器进行聊天

telnet 127.0.0.1 8888

1. 测试验证

预备发送：

已发送：

