使用 epoll 在 Linux

epoll 是一個設計用於高效監控多個文件描述符事件的 Linux 系統調用，例如當數據可供讀取或套接字準備好寫入時。它在網絡編程中特別有用，因為伺服器可能需要同時處理許多連接。與舊機制如 select 或 poll 相比，epoll 更能適應大量文件描述符，因為它避免了重複掃描所有描述符以檢查其準備狀態。

以下是在 Linux 中使用 epoll 的逐步指南，後面跟隨一個簡單的伺服器示例，該伺服器接受連接並回顯數據。

使用 epoll 的步驟

1. 創建一個 Epoll 實例

要開始使用 epoll，您需要創建一個 epoll 實例，該實例由一個文件描述符表示。使用 epoll_create1 系統調用：

```
int epoll_fd = epoll_create1(0);
```

- **參數**：傳遞 0 以進行基本使用（無特殊標誌）。可以使用 EPOLL_CLOEXEC 等標誌進行高級情況。
- **返回值**：成功時返回一個文件描述符（epoll_fd），錯誤時返回 -1（檢查 errno 以獲取詳細信息）。

舊的 epoll_create 函數類似，但接受一個大小提示（現在被忽略），因此建議使用 epoll_create1。

2. 添加要監控的文件描述符

使用 epoll_ctl 將文件描述符（例如套接字）註冊到 epoll 實例，並指定要監控的事件：

```
struct epoll_event ev;
ev.events = EPOLLIN;  // 監控可讀性
ev.data.fd = some_fd; // 要監控的文件描述符
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, some_fd, &ev);
```

- **參數**：
  - epoll_fd：epoll 實例文件描述符。
  - EPOLL_CTL_ADD：添加文件描述符的操作。
  - some_fd：要監控的文件描述符（例如一個套接字）。
  - &ev：指向定義事件和可選用戶數據的 struct epoll_event。
- **常見事件**：
  - EPOLLIN：數據可供讀取。
  - EPOLLOUT：準備好寫入。

  - EPOLLERR：發生錯誤。
  - EPOLLHUP：掛斷（例如連接關閉）。
- **用戶數據**：struct epoll_event 中的 data 字段可以存儲文件描述符（如上所示）或其他數據（例如指針）以在事件發生時識別來源。

3. 等待事件

使用 epoll_wait 封鎖並等待監控文件描述符上的事件：

```
struct epoll_event events[MAX_EVENTS];
int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
```

- **參數**：
  - epoll_fd：epoll 實例。
  - events：存儲觸發事件的數組。
  - MAX_EVENTS：要返回的最大事件數（數組大小）。
  - -1：超時（毫秒）（-1 表示無限期等待；0 立即返回）。
- **返回值**：具有事件的文件描述符數量（nfds），或錯誤時返回 -1。

4. 處理事件

循環遍歷 epoll_wait 返回的事件並處理它們：

```
for (int i = 0; i < nfds; i++) {
    if (events[i].events & EPOLLIN) {
        // 文件描述符 events[i].data.fd 可讀
    }
}
```

- 使用位運算（例如 events[i].events & EPOLLIN）檢查 events 字段以確定事件類型。
- 使用 events[i].data.fd 來識別觸發事件的文件描述符。

5. 管理文件描述符（可選）

- **移除**：使用 epoll_ctl 並帶有 EPOLL_CTL_DEL 以停止監控文件描述符：

  ```
  epoll_ctl(epoll_fd, EPOLL_CTL_DEL, some_fd, NULL);
  ```

- **修改**：使用 EPOLL_CTL_MOD 調整事件：

  ```
  ev.events = EPOLLOUT; // 更改為監控可寫性
  epoll_ctl(epoll_fd, EPOLL_CTL_MOD, some_fd, &ev);
  ```

關鍵概念

水平觸發 vs. 邊緣觸發

- **水平觸發（默認）**：`epoll` 在條件持續存在時重複通知（例如數據保持未讀）。對於大多數情況來說更簡單。
- **邊緣觸發（`EPOLLET`）**：在狀態變化時只通知一次（例如新數據到達）。需要讀取/寫入所有數據直到 `EAGAIN` 以避免遺漏事件；更高效但更難處理。
- 如果使用邊緣觸發模式，則在 `ev.events` 中設置 `EPOLLET`（例如 `EPOLLIN | EPOLLET`）。

非阻塞 I/O

`epoll` 通常與非阻塞文件描述符配對，以防止在 I/O 操作上阻塞。使用以下方式將套接字設置為非阻塞模式：

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);
```

示例：簡單回顯伺服器

以下是一個基本示例，該伺服器使用 `epoll` 接受連接並將數據回顯給客戶端。它使用水平觸發模式以簡化。

```
#include <sys/epoll.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define MAX_EVENTS 10
#define PORT 8080

int main() {
    // 創建監聽套接字
    int listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_fd == -1) { perror("socket"); exit(1); }

    struct sockaddr_in addr = { .sin_family = AF_INET, .sin_addr.s_addr = INADDR_ANY, .sin_port = htons(PORT)
    if (bind(listen_fd, (struct sockaddr*)&addr, sizeof(addr)) == -1) { perror("bind"); exit(1); }
    if (listen(listen_fd, 5) == -1) { perror("listen"); exit(1); }
```

```
// 將監聽套接字設置為非阻塞
fcntl(listen_fd, F_SETFL, fcntl(listen_fd, F_GETFL) | O_NONBLOCK);

// 創建 epoll 實例
int epoll_fd = epoll_create1(0);
if (epoll_fd == -1) { perror("epoll_create1"); exit(1); }

// 將監聽套接字添加到 epoll
struct epoll_event ev, events[MAX_EVENTS];
ev.events = EPOLLIN; // 水平觸發
ev.data.fd = listen_fd;
if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_fd, &ev) == -1) { perror("epoll_ctl"); exit(1); }

// 事件循環
while (1) {
    int nfds = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
    if (nfds == -1) { perror("epoll_wait"); exit(1); }

    for (int i = 0; i < nfds; i++) {
        int fd = events[i].data.fd;

        if (fd == listen_fd) {
            // 接受新連接
            int client_fd = accept(listen_fd, NULL, NULL);
            if (client_fd == -1) { perror("accept"); continue; }

            // 將客戶端套接字設置為非阻塞
            fcntl(client_fd, F_SETFL, fcntl(client_fd, F_GETFL) | O_NONBLOCK);

            // 將客戶端套接字添加到 epoll
            ev.events = EPOLLIN;
            ev.data.fd = client_fd;
            if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &ev) == -1) {
                perror("epoll_ctl: client_fd");
                close(client_fd);
            }
        } else {
            // 處理客戶端數據
            char buf[1024];
```

```
                ssize_t n = read(fd, buf, sizeof(buf));
                if (n <= 0) {
                    // 錯誤或 EOF：關閉並從 epoll 中移除
                    if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) continue;
                    close(fd);
                    epoll_ctl(epoll_fd, EPOLL_CTL_DEL, fd, NULL);
                } else {
                    // 回顯數據
                    write(fd, buf, n); // 注意：在實際應用中處理部分寫入
                }
            }
        }
    }

    close(epoll_fd);
    close(listen_fd);
    return 0;
}
```

示例說明

- **監聽套接字**：監控 `EPOLLIN` 以檢測新連接。
- **客戶端套接字**：也監控 `EPOLLIN` 以檢測傳入數據。
- **簡化**：假設 `write` 完成。在生產中，緩衝數據並使用 `EPOLLOUT` 進行部分寫入。
- **錯誤處理**：在錯誤或 EOF 時關閉套接字並從 epoll 中移除。

摘要

`epoll` 提供了一種高效的方式來處理 Linux 中的多個文件描述符：1. 使用 `epoll_create1` 創建實例。2. 使用 `epoll_ctl` 註冊文件描述符和事件。3. 使用 `epoll_wait` 等待事件。4. 在循環中處理事件，根據需要調整監控事件或移除描述符。

對於簡單應用程序，建議使用水平觸發模式。對於高性能需求，考慮使用邊緣觸發模式，並仔細處理所有可用數據。始終將 `epoll` 與非阻塞 I/O 配對以獲得最佳結果。