第11课：引入事件驱动模型

网络编程里的I/O处理

在Linux网络编程中，I/O（输入/输出）处理主要是指在网络通信过程中，对数据的发送和接收操作。Linux系统提供了多种机制来处理网络I/O，这些机制旨在高效、可靠地从网络接口读取数据并写入数据到网络。

I/O

I/O模型是是操作系统中处理输入输出操作的一种机制。它描述了应用程序如何与操作系统内核交互，以完成对外部设备（如网络、磁盘等）的数据读写操作。I/O 操作通常是阻塞的，可能导致应用程序等待数据的到来或传输完成，进而影响性能。

I/O 操作一般分为两个阶段：

1. **等待数据准备就绪**：例如，从网络上接收数据时，可能需要等待数据包的到来。
2. **将数据从内核空间复制到用户空间**：数据准备好后，需要从操作系统内核缓冲区复制到应用程序的缓冲区。

数据从外部设备（如网络、磁盘等）传输到应用程序内存时，通常需要经过操作系统的内核。I/O 操作通常包含两个阶段：等待数据准备就绪和将数据从内核空间复制到用户空间。以下是详细过程描述：

1. 等待数据准备就绪

- **外部设备与内核交互**：当应用程序请求 I/O 操作时（例如，读取文件或接收网络数据），外部设备会将数据传输到操作系统内核中的缓冲区。
- **等待数据到达**：在网络 I/O 中，内核可能需要等待网络接口控制器（NIC）接收数据包，将数据包放入网络缓冲区中；在磁盘 I/O 中，内核则需要等待磁盘驱动器将数据加载到内核的磁盘缓冲区中。等待时间取决于设备的响应速度。
- **阻塞和非阻塞 I/O**：应用程序发起请求后，如果采用阻塞 I/O，应用程序会暂停执行，等待数据到达；如果采用非阻塞 I/O，应用程序在等待数据期间可以继续执行其他操作。内核会持续检查数据是否到达缓冲区。

2. 将数据从内核空间复制到用户空间

- **内核和用户空间的分离**：现代操作系统中，内存空间通常分为内核空间和用户空间。内核空间具有更高的权限和安全性，而用户空间则为应用程序所用。为了保证安全，应用程序不能直接访问内核空间的数据。
- **数据拷贝**：当数据到达内核缓冲区且准备好后，内核会将数据从内核空间复制到应用程序的用户空间缓冲区中。这个过程称为数据拷贝。
- **效率问题**：这种数据拷贝增加了 I/O 操作的开销，因为数据在用户空间和内核空间之间传输可能会涉及上下文切换和数据复制。一些优化方式（如零拷贝）可以减少这种开销。

I/O 操作的整体流程

结合以上两个阶段，数据进入内核缓冲区并最终到达用户空间的大致过程如下：

1. **应用程序发起 I/O 请求**：应用程序向操作系统发出读取或接收数据的请求（如 `read()` 或 `recv()`）。
2. **数据进入内核缓冲区**：外部设备（网络、磁盘等）将数据传输到内核空间的缓冲区中，此时需要等待数据到达。
3. **检查数据是否就绪**：操作系统通过设备驱动和中断机制通知内核数据已就绪。
4. **数据拷贝到用户空间**：数据到达内核缓冲区后，内核将数据从内核缓冲区复制到应用程序的用户空间缓冲区。
5. **通知应用程序**：数据传输完成后，操作系统通知应用程序可以读取用户空间中的数据，I/O 操作结束。

优化方式：零拷贝

为了减少 I/O 操作中的数据拷贝次数，许多现代系统引入了零拷贝（zero-copy） 技术。这种方式通过直接将数据从内核缓冲区传输到网络接口或磁盘接口，减少了内核空间到用户空间的拷贝操作，提高了传输效率。

不同的 I/O 模型在处理这两个阶段时，采用了不同的策略，以优化应用程序的性能和资源利用率。

硬盘和网络速度慢于内存和 CPU：

磁盘 I/O 慢：硬盘读取和写入数据的速度远远低于内存和 CPU 的处理速度，尤其是机械硬盘（HDD）。即使是速度较快的固态硬盘（SSD），其读写速度也比内存和 CPU 慢很多。

网络 I/O 慢：网络数据的发送和接收涉及路由、传输和协议处理，可能会经过多次中间传输。即使在局域网环境下，网络数据的传输速度也比内存或 CPU 处理速度要慢。

对比分析

CPU 处理：一般是最迅速的，时间在**纳秒级别**（1 纳秒 = 10^-9 秒），可以快速完成请求准备和响应处理。通常只是几微秒或更短时间。

内存操作：内存的读取和写入速度比磁盘快得多，通常在**几十纳秒到几百纳秒**。它的速度大约是 CPU 的几十倍慢，但仍然比 I/O 快很多。

网络 I/O：互联网请求中最耗时的部分通常是**网络 I/O**，包括 DNS 查询、建立 TCP 连接、发送和接收 HTTP 请求与响应。由于涉及网络传输、协议处理等，这些操作通常需要**数十到数百毫秒**（1 毫秒 = 10^-3 秒）。网络延迟和带宽限制是影响网络 I/O 时间的主要因素。

磁盘 I/O：磁盘读取在 SSD 上可能需要**几毫秒**，而在机械硬盘（HDD）上可能需要**几十毫秒**。磁盘速度比内存慢很多，特别是机械硬盘，因为存在机械运动的寻道时间和旋转延迟。

相关概念

套接字 (Socket)： 套接字是进程间通信（IPC）和网络通信的基本抽象。在Linux系统中，创建套接字后，会得到一个文件描述符（file descriptor），它是进程访问网络资源的句柄

句柄（Handle）在计算机科学中是一个抽象的概念，它代表了一个对系统资源（如文件、窗口、数据库连接、网络套接字等）的唯一标识符。句柄不是一个实际的数据值，而是一个指向系统内部分配给特定资源的数据结构或对象的引用。通过句柄，应用程序可以间接地访问和控制这些资源，而不是直接操作底层资源的具体地址。

进程上下文切换： 在不同的I/O模型中，可能涉及进程上下文切换。例如，在阻塞I/O模型中，当进程被I/O操作阻塞时，操作系统可能会挂起当前进程并调度其他进程运行；而在异步I/O或非阻塞I/O模型中，进程能够更高效地利用CPU时间，减少不必要的上下文切换。

> 进程上下文切换（Process Context Switch）是指操作系统在多任务环境下，从一个进程的执行环境（即上下文）切换到另一个进程的执行环境的过程。具体来说：
>
> 当CPU需要暂停当前正在运行的进程转而执行另一个进程时，操作系统必须做以下几件事情：
>
> a. **保存当前进程的状态**：包括程序计数器（PC，指示下一条要执行的指令地址）、寄存器状态、内存管理信息（如页表指针）、打开的文件描述符、信号处理设置等所有相关资源和状态。
> b. **恢复下一个进程的状态：**将被选中要运行的下一个进程的上下文信息从进程控制块（PCB, Process Control Block）中取出，并加载到相应的处理器寄存器和内存管理单元中。
> c. **切换线程调度策略决定的进程**：根据操作系统的调度策略（如时间片轮转、优先级调度等），选择一个新的进程来执行。

这个过程涉及到内核态与用户态之间的转换，如果新进程是从用户态切换过来的，还需要通过系统调用返回到用户态并开始执行新的进程代码。

上下文切换开销较大，因为它涉及到了对硬件状态的保存和恢复以及可能的内存交换（如果涉及到虚拟内存且有页面不在物理内存中）。频繁的上下文切换会消耗大量的CPU时间，降低系统的整体性能。因此，在设计和优化多进程或多线程应用程序时，应尽量减少不必要的上下文切换。

缓冲区管理： 内核维护着用于存储网络数据的缓冲区。当网络数据到达时，先放入内核空间的缓冲区，然后根据I/O模型的不同策略，决定何时以及如何将数据复制到用户空间的缓冲区供进程使用，或者反之，将进程要发送的数据从用户空间复制到内核空间的缓冲区，再由内核发送出去。

事件通知机制： 对于信号驱动I/O和异步I/O，内核通过特定机制通知进程数据已准备就绪。在信号驱动I/O中，是通过发送信号；在异步I/O中，则可能是通过完成队列或回调函数。

常见的五种I/O模型

Unix网络编程中，经典地提出了五种I/O模型，它们分别是：

- **阻塞I/O（Blocking I/O）**
- **非阻塞I/O（Non-blocking I/O）**
- **I/O多路复用（I/O Multiplexing）**
- **信号驱动I/O（Signal-driven I/O）**
- **异步I/O（Asynchronous I/O）**

阻塞I/O (Blocking I/O)：

- **默认模式**：在Unix/Linux系统中，所有的套接字都是阻塞的。

- **工作原理**：调用I/O函数（如`recvfrom`）时，如果数据未准备好，进程将被阻塞，直到数据到达并复制到用户空间。

- **优点**：实现简单，编程方便。

- **缺点**：阻塞模式下，进程在等待数据期间无法执行其他任务，导致资源浪费。

流程：

```
1 应用程序调用recvfrom() --> 阻塞等待数据准备 --> 数据准备好 --> 数据复制到用户空间 --> recvfrom()返回
```

主要特点和工作流程：

a. 同步操作：阻塞I/O是同步操作，这意味着程序会等待I/O操作完成后才能继续执行后续代码。

b. 阻塞等待：当程序执行读取或写入操作时，如果数据不可用或无法立即完成，线程或进程将进入阻塞状态，等待数据就绪或操作完成。

c. 资源浪费：在阻塞等待期间，线程或进程不能执行其他任务，因此可能会浪费系统资源，特别是在大量并发连接的情况下。

d. 适用性：阻塞I/O适用于一些低并发、简单的应用场景，例如单线程的文件操作或简单的网络通信。但在高并发或需要实时响应的情况下，阻塞I/O模型可能效率较低，因为大部分时间都用于等待数据

阻塞I/O的线程生命周期

在阻塞I/O模型中，一个线程的生命周期主要包括以下几个阶段：

e. 创建：线程被创建以处理某个客户端连接或I/O任务。

f. 监听：线程进入一个无限循环，等待接收客户端连接或执行I/O操作。

g. 接收连接：当有客户端连接请求到达时，线程会接受这个连接，创建一个新的套接字，并为该连接创建一个新的线程或处理请求。

h. 执行I/O操作：线程会执行阻塞的I/O操作，如读取数据或等待数据的到达。如果没有数据可读，线程会挂起，等待数据就绪或超时。

i. 完成操作：一旦I/O操作完成（数据可读或写入完成），线程会继续执行相关的任务，如处理接收到的数据或发送响应。

j. 关闭连接：在任务完成后，线程会关闭连接，并结束该客户端的处理。

k. 销毁：线程可能会被销毁，或者回到监听状态，等待下一个连接或I/O任务。

在阻塞I/O模型中，如果一个线程执行I/O操作时没有可读或可写的数据，线程会被挂起等待，直到数据就绪或超时。

非阻塞I/O (Non-blocking I/O)：

- **设置非阻塞模式**：通过设置套接字选项，将其设为非阻塞。

- **工作原理**：I/O函数在数据未准备好时立即返回错误，而不是阻塞进程。

- **优点**：进程不会被阻塞，可以执行其他任务。

- **缺点**：需要不断轮询I/O函数，查询数据是否准备好，导致CPU占用率高。

流程：

```
1 应用程序调用select() --> 阻塞等待任何一个I/O事件准备好 --> I/O事件准备好 --> 应用程序调用recvfrom() --> 数据复制到用户空间 --> recvfrom()返回
```

当进程执行一个非阻塞的读取或写入操作时：

a. 如果数据已经准备好并且可以立即处理，系统调用会成功并返回所需的数据（对于读操作），或者确认数据已发送（对于写操作）。

b. 如果数据没有准备好（例如，对于读操作，网络缓冲区为空；对于写操作，网络连接当前无法接收更多的数据），而非阻塞模式下的系统调用不会等待数据准备就绪，而是立即返回一个错误代码，如在POSIX兼容系统上通常会是 `EAGAIN` 或 `EWOULDBLOCK`，表示这个操作现在不能完成，但稍后可能能够完成。

c. 由于非阻塞I/O不等待数据就绪，应用程序需要采取其他策略来确定何时再次尝试进行I/O操作。常见的方法是通过轮询机制（不断地调用I/O函数检查是否可读/可写）、使用事件驱动编程模型结合如epoll、kqueue等I/O复用技术，或者利用异步I/O回调机制。

信号驱动I/O (Signal-driven I/O)：

概述：进程注册一个信号处理器，当特定的I/O操作准备就绪时，内核通过发送一个信号（通常是SIGIO或SIGPOLL）通知进程。进程接收到信号后，在信号处理函数中执行相应的I/O操作。这种模型允许进程在等待期间执行其他任务，但在实际应用中并不常用。

- **设置信号处理函数**：通过注册信号处理函数，当I/O事件发生时，内核发送信号通知应用程序。

- **工作原理**：应用程序在等待数据期间，不被阻塞，数据准备好时，内核发送 `SIGIO` 信号，通知应用程序进行数据读取。

- **优点**：进程不必轮询I/O状态，提高了CPU利用率。

- **缺点**：信号机制复杂，处理信号的上下文切换开销较大。

流程：

```
1 应用程序设置SIGIO信号处理函数 --> 正常执行其他任务 --> 数据准备好，内核发送SIGIO信号 --> 信号处理函数调用recvfrom() --> 数据复制到用户空间 --> recvfrom()返回
```

过程详述：

a. 设置套接字为信号驱动式I/O： 首先，通过 `fcntl()` 函数将需要进行信号驱动I/O操作的套接字设置为非阻塞，并开启SIGIO或SIGPOLL信号的通知功能。这样，当该套接字上有数据可读或者写入完成时，内核将会发出相应的信号给进程。

b. 注册信号处理器： 进程使用 `signal()` 或 `sigaction()` 系统调用安装一个自定义的信号处理函数。这个函数将在收到SIGIO或SIGPOLL信号时被调用。

c. 等待信号通知： 在完成上述配置后，进程可以继续执行其他任务，而不必忙于检查套接字是否准备好进行I/O操作。

d. 信号触发及处理： 当内核检测到关联的套接字上有数据可读或写入操作已完成时，它会向进程发送已注册的信号（如SIGIO）。进程接收到此信号后，会暂停当前的任务并转而执行预先注册好的信号处理函数。

e. 在信号处理函数中执行I/O操作： 信号处理函数通常会负责执行实际的I/O操作，例如调用 `read()` 或 `write()` 来读取或写入数据。由于在信号处理期间，套接字应当是准备好进行I/O的，因此这些调用应该能够成功完成。

f. 恢复信号处理方式： 为了确保程序的健壮性，有时可能还需要在信号处理函数中重新设置信号处理方式，以防止信号丢失或多次处理同一事件。

虽然信号驱动I/O提供了异步通知的能力，但在现代Linux编程实践中，它不如I/O复用（如epoll）或异步I/O（AIO）常见，因为它们具有更高的效率和更灵活的控制机制。

I/O复用 (I/O Multiplexing)：

- **典型函数**：`select`、`poll`、`epoll`（Linux特有）。

- **工作原理**：使用一个系统调用同时监听多个文件描述符，等待其中的任何一个或多个变为可读或可写。

- **优点**：能够同时监控多个I/O事件，提高了程序的并发性。

- **缺点**：`select`和`poll`在大并发场景下性能较差，需要遍历大量文件描述符。

流程：

```
1 应用程序调用select() --> 阻塞等待任何一个I/O事件准备好 --> I/O事件准备好 --> 应用程序调用recvfrom() --> 数据复制到用户空间 --> recvfrom()返回
```

异步I/O (Asynchronous I/O, AIO)

由操作系统内核完成所有的I/O操作，包括数据的准备和复制，应用程序只需等待通知。

- **工作原理**：应用程序调用aio_read等异步I/O函数，内核立即返回，I/O操作在后台进行，完成后通知应用程序。
- **优点**：无需等待I/O操作完成，进程可以继续执行其他任务，提高并发性能。
- **缺点**：支持异步I/O的系统和库较少，实现复杂。

在Linux中，使用aio_read/aio_write等函数发起异步I/O请求，然后立即返回，不阻塞当前进程。当I/O操作完成后，内核通过回调函数或通过传递给操作系统指定的方式通知进程。这意味着进程可以在发出I/O请求后继续执行其他任务，而无需关心I/O何时完成。极大地提高了系统的并发性和响应速度。然而，Linux下的原生AIO支持相对复杂，实际应用中可能更多采用libaio库或者自行实现类似的逻辑来模拟异步行为。

各种I/O模型的比较

模型	阻塞等待数据准备	数据复制到用户空间	优点	缺点
阻塞I/O	是	是	实现简单	阻塞期间无法处理其他任务
非阻塞I/O	否（轮询）	是	进程不被阻塞	需要轮询，占用CPU资源
I/O多路复用	是	是	同时监控多个I/O事件	`select` / `poll` 性能较差
信号驱动I/O	否	是	不需轮询，由信号通知	信号处理复杂，上下文切换开销大
异步I/O	否	否	真正的异步，效率高	实现复杂，支持有限

事件驱动模型简介

事件驱动模型定义

事件驱动模型是一种编程范式，服务器在这种模型下通过监听事件来驱动程序运行，而不是顺序执行或者等待某个操作完成。这些事件通常包括网络IO事件（如可读、可写），定时器事件等。

上述这五种模型主要描述的是操作系统如何管理和调度进程与I/O设备之间的交互。它们都涉及到网络编程中读写数据时进程状态的变化以及对系统调用的响应方式。

- **与阻塞IO模型的区别**
  - 传统的阻塞IO模型在执行IO操作（如读取网络数据）时，如果没有数据可读，会导致线程挂起等待，直到有数据可读或者超时。这种模型在处理大量并发连接时效率低下，因为大部分时间线程都在等待。
  - 事件驱动模型则不同，它允许程序在一个线程内监听多个IO事件，当某个事件就绪（例如某个socket可读）时，才会执行相应的处理，从而提高效率和减少资源消耗。

1. 阻塞IO线程：
   - 阻塞IO线程通常是多线程模型，每个线程负责处理一个客户端连接或一个IO操作。
   - 每个线程在执行IO操作时会阻塞，等待数据的到达或完成IO操作，这会导致线程挂起。
   - 在阻塞IO模型中，线程通常会一直等待，直到IO操作完成，无论是否有其他任务可以执行。
2. 事件驱动型线程：

- 事件驱动型线程通常是单线程或有限数量的线程。
- 线程会运行事件循环，等待事件的发生，而不会阻塞。
- 当事件发生时，事件循环会调用相应的事件处理函数（回调函数）来处理事件，而不会等待IO操作完成。
- 事件驱动型线程通常采用非阻塞的方式处理事件，因此可以同时处理多个事件或任务，而不会阻塞在单个事件上。

事件驱动下的服务器处理流程

在**事件驱动模型**中，服务器处理客户端请求的过程正是利用事件循环和非阻塞 I/O 来避免阻塞，提高并发和效率。可以具体分解如下：

新连接的可读事件：当服务器主线程监听到有**新的连接**到来时（通常通过 accept 系统调用），这个连接的**可读事件**（表示有数据可读）会被触发。此时，服务器不会立即阻塞等待数据，而是将这个事件**注册到事件循环**中，通常是通过 I/O 多路复用机制（如 epoll、select、kqueue）。

异步读取数据：注册好事件后，服务器主线程会**立即返回**，继续监听其他事件或处理其他任务。与此同时，内核负责从网络设备读取数据（网络数据从网卡进入服务器），直到数据准备好被读取。这种设计确保了主线程**不会阻塞**在等待网络数据上，而是继续执行其他请求或任务。

事件准备好加入就绪列表：当数据从网络设备到达并被读入到**内核缓冲区**时，内核会通知事件循环（通过 epoll_wait 等），将这个连接的事件放入**就绪列表**。这个时候，事件循环会检测到该事件**已经准备好**，可以开始处理。

服务器处理数据：服务器主线程从就绪列表中取出事件，然后通过非阻塞 I/O 函数（如 read 或 recv）**读取内核缓冲区的数据**，并进行处理（例如解析 HTTP 请求、查询数据库、生成响应等）。

核心优势：避免等待 I/O 完成的过程。这个流程的核心优势在于，服务器**不用等待**从网络数据区（网卡）读取到内核缓冲区的过程。主线程在等待期间**可以正常执行其他任务**（如处理其他连接、维护心跳、后台任务等），只有在数据准备好后才开始读取和处理，从而大大提高了服务器的**并发性能**和**资源利用率**。

技术层面解释（重要理解，能讲清楚事件驱动的逻辑）

假设有一个简单的事件驱动型Web服务器，采用单线程模型处理客户端请求。服务器创建一个主循环（事件循环），并设置为监听套接字的读就绪事件。

1. 初始化阶段：
   - 服务器启动时，首先创建一个监听套接字并将其设置为非阻塞模式。
   - 然后将监听套接字注册到事件循环中，以便在有新的客户端连接请求时收到通知。

2. 事件循环：
    - 主线程进入事件循环，调用系统提供的I/O复用函数如 `epoll_wait()`（Linux）或 `kqueue()`（BSD系统）等，这些函数会阻塞等待指定文件描述符集合上的事件发生。
    - 当有新客户端连接请求（即监听套接字变为可读状态），事件循环立即返回，并得到一个表示已就绪的文件描述符列表。
3. 事件处理：
    - **对于每一个就绪的文件描述符，事件循环调用相应的回调函数来处理事件**。
        - 如果是监听套接字，那么事件处理器可能执行 `accept()` 接受新的连接，并为新连接创建一个新的非阻塞套接字，同时将新套接字也注册到事件循环中监听读写事件。
        - 如果是已连接的客户端套接字，事件处理器可能是读取请求数据，解析HTTP请求头和正文，然后调用处理请求的回调函数，该函数生成响应并将其写入相应套接字。
4. 异步IO操作：
    - 在整个过程中，所有网络I/O都是非阻塞的，例如读取客户端数据时，如果当前没有足够的数据可供读取，`read()` 不会阻塞而是立刻返回，事件循环继续检查其他待处理的事件。
    - 当有数据写入客户端时，同样使用非阻塞的 `write()` 调用，若不能一次性写出全部数据，则下次事件循环迭代时再次尝试写入。

通过这种方式，**事件驱动型线程可以在单个线程内高效地并发处理多个客户端连接，每个客户端的请求、响应过程都变成了独立的事件，由事件循环统一调度管理，而不是为每个客户端创建单独的线程或进程，从而降低了资源消耗，提高了系统的并发能力。**

事件驱动模型的关键组件

1. **事件循环（Event Loop）**：负责循环监听事件源，分发事件。
2. **事件源（Event Source）**：触发事件的对象，如网络连接、文件描述符等。
3. **事件处理器（Event Handler）**：处理特定事件的回调函数。

事件驱动模型的工作流程

1. **注册事件**：应用程序向事件循环注册感兴趣的事件和对应的处理器。
2. **事件监听**：事件循环不断监听事件源，等待事件发生。
3. **事件触发**：当事件源有事件发生时，事件循环收到通知。
4. **事件分发**：事件循环调用对应的事件处理器，处理该事件。

5. **重复循环**：处理完当前事件后，继续监听其他事件。

事件驱动模型的优点

- **高并发处理能力**：能同时处理大量的I/O事件，适合高并发场景。
- **资源利用率高**：非阻塞I/O和事件通知机制，避免了线程阻塞，节省系统资源。
- **响应迅速**：事件发生时立即处理，减少了延迟。

事件驱动模型的应用场景

- **网络服务器**：如Nginx、Node.js等，处理高并发的网络请求。
- **GUI程序**：图形用户界面程序，通过事件驱动响应用户的交互。
- **异步编程**：在需要非阻塞、异步操作的场景，如文件I/O、网络I/O等。

epoll原理详细解释

用户态和内核态

1. 用户态（User Mode）：
   - 用户态是指操作系统中的一种运行模式，通常是指用户应用程序运行的环境。
   - 在用户态下，应用程序只能访问有限的资源和执行有限的操作，例如文件读写、网络通信等。
   - 应用程序运行在用户态时，它们不能直接访问底层硬件资源或进行特权操作。
   - 用户态下的程序执行受到严格的权限控制，以确保它们不能干扰操作系统的正常运行。
2. 内核态（Kernel Mode）：
   - 内核态是操作系统的核心运行模式，也被称为特权模式。
   - 在内核态下，操作系统内核具有最高的权限，可以访问所有的硬件资源和执行所有特权操作。
   - 内核态下的代码能够执行关键的系统管理任务，如进程管理、内存管理、设备驱动程序等。
   - 操作系统内核通常运行在内核态下，以便执行系统级任务并提供服务给用户态的应用程序。

epoll工作原理

- **Linux特有的IO多路复用机制**：epoll是专门为Linux系统设计的一种高效的IO多路复用技术。**它通过一个文件描述符来跟踪和管理多个socket。**
- **文件描述符管理**：在epoll模型中，所有需要监视的socket都会被加入到一个由epoll实例管理的内部数据结构中。这个实例由一个文件描述符代表，通过这个描述符可以对这些socket进行各种操作。
- **系统调用**：使用 `epoll_wait` 等系统调用，可以询问epoll实例哪些socket处于就绪状态，即准备好执行读取、写入或其他操作。

epoll的模式

- **LT（水平触发）模式**：
  - 在此模式下，只要满足某个条件（例如，有数据可读），epoll就会不断地通知应用程序，直到该事件被处理。
  - 水平触发模式更容易理解和使用，但在高负载情况下可能会导致性能问题。
- **ET（边缘触发）模式**：
  - 边缘触发模式只在被监视的socket状态发生变化时（例如，从无数据到有数据）通知应用程序一次。
  - 这种模式通常更高效，因为它减少了事件通知的次数，但处理起来更复杂。

与select/poll的比较

工作机制

- IO多路复用机制：select和poll是UNIX和Linux系统中较早提供的IO多路复用机制。它们允许程序同时监控多个文件描述符（通常是网络socket），以检测一个或多个文件描述符是否有IO操作就绪。

1. select()：
   - select系统调用提供了最基本的I/O复用功能。它需要三个集合参数：读就绪集、写就绪集和异常就绪集，用于指定要监视的文件描述符。
   - 调用会阻塞直到至少有一个文件描述符准备好进行I/O操作，或者超时（可选设置）。
   - 返回后，可以通过检查这三个集合得知哪些文件描述符准备好了进行读、写或发生了异常条件。
2. poll()：
   - poll系统调用是select的一个增强版本，同样可以监听多个文件描述符，但没有最大文件描述符数量的限制，并且事件类型定义更为灵活。

- 使用一个pollfd结构体数组来表示待监测的文件描述符及其相应的事件类型。
- 同样会阻塞直到有文件描述符就绪或超时。

性能瓶颈

- 每次调用select或poll时，都需要将所有被监控的文件描述符的集合从用户空间复制到内核空间。这个过程在文件描述符数量较少时开销不大，但在处理数百或数千个文件描述符时，会成为性能瓶颈。
- select和poll需要遍历整个文件描述符集合来检查哪些文件描述符就绪，这在大量并发连接下效率低下

epoll 使用的主要数据结构：

- 红黑树：epoll 内部使用一种称为红黑树的平衡二叉搜索树来存储所有注册的文件描述符（socket）。每个节点代表一个文件描述符及其相关的事件和数据。红黑树保证了插入、删除和查找操作的高效性，即使在管理成千上万的文件描述符时也能保持高效。
- 就绪列表：当某个文件描述符上的事件就绪（如可读或可写）时，它会被添加到一个就绪列表中。这个列表仅包含那些状态发生变化，需要处理的文件描述符。

不需要检查所有文件描述符的原理

- 在 select 或 poll 中，每次调用都需要遍历整个文件描述符集合来检查每个文件描述符的状态。这在文件描述符数量较多时效率低下。
- **在 epoll 中，不需要遍历整个红黑树来检查每个文件描述符。当某个文件描述符上的事件变为就绪状态时，它会被自动添加到就绪列表**。因此，epoll 只需检查这个就绪列表，而不是全部的文件描述符集合。
    - 内核与用户空间共享：epoll通过内核与用户空间之间共享某些数据结构（如就绪列表）的方式来工作。这种共享机制意味着当文件描述符的状态变化时，内核可以直接在这些共享数据结构上操作，而无需进行数据复制

相比 select 更好的扩展性

- 避免重复数据复制：epoll 避免了每次调用时将文件描述符集合从用户空间复制到内核空间的操作，这是 select 和 poll 的一个主要性能瓶颈。
- 高效的事件通知机制：epoll 只通知那些真正发生变化的事件，减少了无效检查和不必要的通知，提高了事件处理的效率。
- 大规模并发连接的管理：得益于其高效的数据结构和算法，epoll 能够管理和处理数千甚至数万个并发网络连接，而不会遇到性能瓶颈，这使得它非常适合构建高性能的网络服务器。

epoll优势：

epoll使用更高效的数据结构和算法来管理大量的socket。它避免了重复的数据复制，且只在socket状态发生变化时通知应用程序，大大提高了处理大规模并发连接的能力。

当一个文件描述符的状态改变（如从不可读变为可读）时，epoll仅通知这个状态变化，而不是每次都检查所有文件描述符。

epoll能够扩展到处理数千甚至数万个并发网络连接，是高性能网络服务器的首选技术。

总结

相比于select和poll，epoll提供了更高的性能和更好的可扩展性。在高并发的网络编程场景中，epoll能够有效地提升资源利用率和响应速度。尽管epoll的API使用起来可能比select和poll复杂，但它的性能优势使得它在现代Linux网络编程中被广泛采用。

epoll实例（理解+能复述）

创建epoll实例

- **创建epoll实例代码**

```
1 int epollfd = epoll_create1(0);
```

  - 这行代码创建一个epoll实例，用于后续管理socket的事件。`epoll_create1(0)`是创建epoll实例的系统调用，其中的`0`表示没有特殊的标志。
  - `epoll_create1`会向操作系统请求创建一个新的epoll句柄。
  - 函数返回的`epollfd`是一个有效的文件描述符，它代表了这个新创建的epoll实例，类似于打开一个文件或创建一个socket时得到的文件描述符。
  - 这个epoll实例就像是一个事件表或者容器，可以添加、删除或查询关注的文件描述符及其发生的事件类型。
  - 参数`0`意味着不设置任何标志，特别是没有指定`EPOLL_CLOEXEC`标记，这意味着该epoll句柄不会在执行exec系列系统调用时自动关闭。

设置非阻塞socket

- **设置非阻塞的代码**

```
1 setNonBlocking(server_fd);
```

- 将socket设置为非阻塞意味着在进行读写操作时，如果没有数据或者不能立即发送数据，操作会立即返回而不是挂起等待，这是实现高效事件驱动模型的关键。
- 这个函数调用将服务器 socket `server_fd` 设置为非阻塞模式。在非阻塞模式下，IO 操作（如读、写）将不会导致线程挂起等待操作完成，而是立即返回，从而使服务器能够在等待数据时继续处理其他任务。

注册事件到epoll

- **注册事件代码**

```
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;
ev.data.fd = server_fd;
epoll_ctl(epollfd, EPOLL_CTL_ADD, server_fd, &ev);
```

- 通过 `epoll_ctl` 将socket注册到epoll实例中，并指定关注的事件类型。这里的 `EPOLLIN` 表示关注可读事件， `EPOLLET` 表示使用边缘触发模式。

`epoll_ctl` 函数是Linux系统中用于操作epoll实例的接口，它可以执行以下三种不同的操作：

1. `EPOLL_CTL_ADD`
   - 作用：向epoll实例中添加一个文件描述符，并注册要监控的事件类型。
   - 语法：

```
C
1int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

   - 在这个操作中，op应设为 `EPOLL_CTL_ADD` ，fd参数是你想要添加到epoll实例中的文件描述符（例如socket），event指向一个已初始化好的 `epoll_event` 结构体，其中包含了你希望在该文件描述符上监听的事件。
2. `EPOLL_CTL_MOD`
   - 作用：修改已经添加到epoll实例中的文件描述符所对应的事件类型。
   - 当需要更改某个已监控的文件描述符上的事件时，可以使用此操作码。这通常用于动态改变对特定文件描述符的事件关注类型，而不需要先删除再重新添加。
3. `EPOLL_CTL_DEL`
   - 作用：从epoll实例中删除一个之前添加过的文件描述符，停止监控其事件。

- 当不再需要监控某个文件描述符上的事件时，可以调用此操作来释放资源。之后对于该文件描述符发生的相应事件，epoll_wait将不会再返回相关信息。

事件循环

- **事件循环代码**

```
while (true) {
    int nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1);
    for (int n = 0; n < nfds; ++n) {
        // 处理事件
      }
  }
```

- 使用`epoll_wait`等待事件的发生，它会阻塞直到有事件发生，然后处理这些事件。这是事件驱动模型中循环监听并处理事件的关键部分。

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int
timeout);
```

参数详解：

- `int epfd`：这是通过调用`epoll_create()`或`epoll_create1()`创建的epoll实例的文件描述符。
- `struct epoll_event *events`：这是一个用户空间预先分配好的数组，用来接收由内核复制过来的已发生的事件结构体。每个结构体包含了触发事件的文件描述符及其相关的事件类型（例如读就绪、写就绪等）。
- `int maxevents`：此参数指定了`events`数组可以接收的最大事件数量。当有多个文件描述符同时准备好时，`epoll_wait()`最多会填充这么多数量的事件到数组中。
- `int timeout`：等待时间（以毫秒为单位）。该值指定`epoll_wait()`函数阻塞的最长时间：
  - 若设置为`-1`，表示将无限期地等待事件发生，即函数直到有至少一个事件发生才会返回。
  - 若设置为`0`，则`epoll_wait()`将立即返回，无论是否有待处理的事件，这通常用于非阻塞模式。
  - 若设置为大于`0`的值，则函数最多会阻塞等待指定的毫秒数，即使在这段时间内没有事件发生也会返回。

处理新的连接请求和IO事件

```
if (events[n].data.fd == server_fd) {
    // 处理新连接
    new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);
    setNonBlocking(new_socket); // 设置新连接为非阻塞模式
    ev.events = EPOLLIN | EPOLLET; // 监听新连接的可读事件和边缘触发
    ev.data.fd = new_socket;
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, new_socket, &ev) == -1) {
        LOG_ERROR("epoll_ctl: new_socket"); // 注册新连接事件失败，记录错误日志
        exit(EXIT_FAILURE);
    }
} else {
    // TODO:处理已连接socket的IO事件

    // ... (读取请求，处理请求，发送响应) ...
}
```

- 事件循环中，根据文件描述符判断事件类型：若为服务器 socket `server_fd`，则表示有新的连接请求；否则，为已建立连接的 socket 上的 IO 事件。对于新连接，使用 `accept` 接受连接，设置为非阻塞，并将其注册到 epoll 实例中。

资源管理和错误处理

- **重要性**
  - 在服务器编程中，正确的资源管理和错误处理是非常关键的。这包括在出错时释放资源，以及确保在连接关闭时，从epoll实例中移除socket并关闭它们。

现在的服务器代码

```
//第九课新增
void setNonBlocking(int sock) {
    int opts;
    opts = fcntl(sock, F_GETFL); // 获取socket的文件描述符当前的状态标志
    if (opts < 0) {
        LOG_ERROR("fcntl(F_GETFL)"); // 获取标志失败，记录错误日志
        exit(EXIT_FAILURE);
    }
    opts = (opts | O_NONBLOCK); // 设置非阻塞标志
    if (fcntl(sock, F_SETFL, opts) < 0) { // 应用新的标志设置到socket
        LOG_ERROR("fcntl(F_SETFL)"); // 设置失败，记录错误日志
        exit(EXIT_FAILURE);
    }
```

```cpp
}

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    struct epoll_event ev, events[MAX_EVENTS];
    int epollfd, nfds;

    // 创建socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0); // 创建TCP socket
    setNonBlocking(server_fd); // 设置为非阻塞模式
    LOG_INFO("Socket created");

    // 定义地址
    address.sin_family = AF_INET; // 指定地址族为IPv4
    address.sin_addr.s_addr = INADDR_ANY; // 绑定到所有可用地址
    address.sin_port = htons(PORT); // 指定端口号

    // 绑定socket
    bind(server_fd, (struct sockaddr *)&address, sizeof(address)); // 绑定socket到指定地址和端口

    // 监听请求
    listen(server_fd, 3); // 开始监听端口，设置最大监听队列长度为3
    LOG_INFO("Server listening on port " + std::to_string(PORT));

     // 设置路由
    setupRoutes();
    LOG_INFO("Server starting");

    // 创建epoll实例
    epollfd = epoll_create1(0); // 创建epoll实例
    if (epollfd == -1) {
        LOG_ERROR("epoll_create -1"); // 创建失败，记录错误日志
        exit(EXIT_FAILURE);
    }

    ev.events = EPOLLIN | EPOLLET; // 设置要监听的事件类型：可读和边缘触发
    ev.data.fd = server_fd;
    if (epoll_ctl(epollfd, EPOLL_CTL_ADD, server_fd, &ev) == -1) {
        LOG_ERROR("epoll_ctl: server_fd"); // 注册事件失败，记录错误日志
        exit(EXIT_FAILURE);
    }

    // 事件循环
    while (true) {
```

```
60          nfds = epoll_wait(epollfd, events, MAX_EVENTS, -1); // 等待事件发生
61          for (int n = 0; n < nfds; ++n) {
62              if (events[n].data.fd == server_fd) {
63                  // 处理新连接
64                  new_socket = accept(server_fd, (struct sockaddr *)&address,
    (socklen_t*)&addrlen);
65                  setNonBlocking(new_socket); // 设置新连接为非阻塞模式
66                  ev.events = EPOLLIN | EPOLLET; // 监听新连接的可读事件和边缘触发
67                  ev.data.fd = new_socket;
68                  if (epoll_ctl(epollfd, EPOLL_CTL_ADD, new_socket, &ev) == -1) {
69                      LOG_ERROR("epoll_ctl: new_socket"); // 注册新连接事件失败，记
    录错误日志
70                      exit(EXIT_FAILURE);
71                  }
72              } else {
73                  // TODO:处理已连接socket的IO事件
74
75                  // ... (读取请求，处理请求，发送响应) ...
76              }
77          }
78      }
79
80      return 0;
81  }
82
```

事件驱动相关的互联网开发面试常问问题

问题1：什么是阻塞I/O和非阻塞I/O？

回答：

- **阻塞I/O**：在进行I/O操作时，如果数据未准备好，调用会阻塞线程，直到数据准备好或操作完成。线程在此期间无法执行其他任务。

- **非阻塞I/O**：I/O操作立即返回，如果数据未准备好，返回错误或特定值，线程可以继续执行其他任务，需轮询或通过其他方式检查数据是否准备好。

问题2：I/O多路复用和多线程有什么区别？

具体我们会在多线程再详细讲解，可以先思考

问题3：什么是水平触发（LT）和边缘触发（ET）？

回答：

- **水平触发（Level Triggered，LT）**：

  - **工作方式**：当I/O事件发生后，只要文件描述符仍然是就绪状态，每次调用 `epoll_wait()` 都会返回该文件描述符。

  - **特点**：应用程序可以不必一次性处理完数据，适合流式数据处理。

- **边缘触发（Edge Triggered，ET）**：

  - **工作方式**：当I/O事件发生且文件描述符从未就绪变为就绪时， `epoll_wait()` 只返回一次。

  - **特点**：要求应用程序一次性将数据处理完，需配合非阻塞I/O使用，提高效率，减少 `epoll_wait()` 的调用次数。

问题4：如何使用epoll实现高并发服务器？

回答：

1. **创建epoll实例**：使用 `epoll_create()` 创建 `epfd` 。

2. **注册事件**：将监听套接字 `listen_fd` 以 `EPOLLIN` 事件注册到 `epfd` 。

3. **事件循环**：在循环中调用 `epoll_wait()` 等待事件发生。

4. **处理事件**：

- **新连接**：如果 `listen_fd` 有 `EPOLLIN` 事件，接受新连接 `accept()` ，并将新连接的套接字 `conn_fd` 以 `EPOLLIN` 事件注册到 `epfd` 。

- **数据读写**：如果 `conn_fd` 有 `EPOLLIN` 事件，读取数据，处理请求，必要时发送响应。

5. **关闭连接**：当连接完成或出错时，关闭 `conn_fd` ，并使用 `epoll_ctl()` 将其从 `epfd` 中删除。

问题5：为什么epoll在高并发下性能更好？

回答：

- **事件通知机制**： `epoll` 采用事件驱动，不需要遍历所有文件描述符，只处理活跃的描述符，降低了CPU开销。

- **内核优化**： `epoll` 在内核中维护红黑树和就绪链表，支持高效的增删改查操作。

- **减少内存拷贝**： `epoll` 通过内核和用户空间共享内存的方式，减少了内存拷贝。

问题6：epoll中红黑树和就绪链表的作用是什么？

回答：

- **红黑树**：存储所有注册到 `epfd` 的文件描述符，支持高效的插入、删除和查找操作，用于管理和维护文件描述符的状态。

- **就绪链表**：存储已准备就绪的文件描述符，当I/O事件发生时，内核将就绪的 `fd` 添加到就绪链表。`epoll_wait()` 直接从就绪链表中获取就绪事件，避免遍历，提高效率。

问题7：在使用epoll时，为什么建议使用非阻塞套接字？

回答：

- **防止阻塞**：在ET模式下，如果套接字是阻塞的，且数据未全部处理完，可能导致应用程序阻塞，影响性能。

- **一次性读取**：非阻塞套接字配合ET模式，确保每次I/O操作都不会阻塞，需要循环读取或写入，直到返回 `EAGAIN`，表示数据处理完毕。

问题8：什么是Reactor和Proactor模型？

回答：

- **Reactor模型**：

- **工作方式**：事件驱动模型的一种，应用程序通过事件循环等待事件发生，当I/O事件准备好时，事件循环分发事件，由应用程序完成实际的I/O读写操作。

- **特点**：事件通知后，应用程序主动进行I/O操作。

- **Proactor模型**：

- **工作方式**：异步I/O模型的一种，应用程序发起异步I/O请求，由操作系统完成I/O操作，完成后通知应用程序处理结果。

- **特点**：操作系统负责完成I/O操作，应用程序只处理业务逻辑。

问题9：在Linux下，为什么epoll比select更高效？

回答：

- **避免描述符集合拷贝**：`select` 每次调用都需要将描述符集合从用户空间拷贝到内核空间，`epoll` 只需在注册时拷贝一次。

- **事件触发机制**：`epoll` 采用事件通知方式，不需要遍历整个描述符集合，`select` 则每次都要遍历整个集合，性能随着描述符数量增加而下降。

- **就绪事件列表**：`epoll` 维护就绪事件列表，应用程序直接获取已准备好的事件，效率更高。

问题10：epoll和异步I/O有什么区别？

回答：

- **epoll**：

  - **性质**：I/O多路复用机制，属于同步非阻塞I/O。

  - **工作方式**：应用程序负责发起I/O操作，`epoll`通知事件发生，应用程序再进行I/O读写操作。

- **异步I/O**：

  - **性质**：由操作系统内核完成I/O操作，应用程序无需关心数据的准备和传输。

  - **工作方式**：应用程序发起异步I/O请求，内核完成I/O操作后，通知应用程序处理结果。

- **区别**：

  - **epoll**需要应用程序参与I/O操作，是同步的。

  - **异步I/O**完全由内核负责I/O操作，是异步的。
