命名管道 fifo

命名管道（Named Pipe，或 FIFO）是一种特殊的文件类型，用于在无亲缘关系的进程之间实现单向或双向通信。命名管道可以在文件系统中创建，并且不同进程可以通过访问该文件来通信。命名管道和普通管道的区别在于，命名管道具有文件名，可以在进程之间共享，不要求通信的进程有父子关系。

命名管道的原理

1. **文件系统中的特殊文件**：命名管道是文件系统中的一个实体，它的行为类似于文件，但它并不存储数据。它只是一种通道，通过它可以实现进程之间的数据传输。
2. **内核缓冲区**：当进程向命名管道写入数据时，数据会被写入内核的缓冲区中；另一端的进程读取数据时，从该缓冲区读取。这种机制确保了数据可以在两个进程之间传递。
3. **阻塞机制**：管道遵循阻塞的行为。如果管道为空，读操作将被阻塞，直到写入操作完成。同样，如果管道缓冲区满了，写操作也会被阻塞，直到有空间可以写入数据。
4. **半双工或全双工**：命名管道可以实现单向通信，如果需要双向通信，通常会创建两个命名管道。

创建命名管道

可以使用 `mkfifo()` 函数在 Linux 系统中创建一个命名管道，或者使用命令行工具 `mkfifo`。例如：

bash 复制

```
mkfifo /tmp/my_fifo
```

这将在 `/tmp` 目录下创建一个名为 `my_fifo` 的命名管道文件。

使用命名管道的代码示例

以下是一个简单的示例，展示如何使用命名管道实现进程间通信：

1. 创建命名管道

cpp 复制

```cpp
#include <iostream>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    const char* fifoPath = "/tmp/my_fifo";

    // 创建命名管道
    if (mkfifo(fifoPath, 0666) == -1) {
        std::cerr << "Error creating named pipe" << std::endl;
        return 1;
    }

    std::cout << "Named pipe created at " << fifoPath << std::endl;
    return 0;
}
```

2. 写进程（向管道写入数据）

cpp 复制

```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>
#include <cstring>

int main() {
    const char* fifoPath = "/tmp/my_fifo";
    int fd = open(fifoPath, O_WRONLY);  // 打开管道写端

    if (fd == -1) {
        std::cerr << "Error opening named pipe for writing" << std::endl;
        return 1;
    }

    const char* message = "Hello from writer process!";
    write(fd, message, strlen(message));
    close(fd);  // 关闭管道写端
    std::cout << "Message sent to pipe" << std::endl;

    return 0;
}
```

3. 读进程（从管道读取数据）

cpp 复制

```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>

int main() {
    const char* fifoPath = "/tmp/my_fifo";
    int fd = open(fifoPath, O_RDONLY);  // 打开管道读端

    if (fd == -1) {
```

3. 读进程（从管道读取数据）

cpp 复制

```cpp
#include <iostream>
#include <fcntl.h>
#include <unistd.h>

int main() {
    const char* fifoPath = "/tmp/my_fifo";
    int fd = open(fifoPath, O_RDONLY);  // 打开管道读端

    if (fd == -1) {
        std::cerr << "Error opening named pipe for reading" << std::endl;
        return 1;
    }

    char buffer[256];
    ssize_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);
    if (bytesRead >= 0) {
        buffer[bytesRead] = '\0';  // 添加字符串结束符
        std::cout << "Message received: " << buffer << std::endl;
    }

    close(fd);  // 关闭管道读端
    return 0;
}
```

运行步骤

1. **创建管道**：先运行创建命名管道的程序，生成命名管道文件 `/tmp/my_fifo`。
2. **运行读进程**：运行读进程程序，打开管道并等待数据。
3. **运行写进程**：运行写进程程序，将数据写入管道。

读进程将读取并输出写进程发送的数据。

注意事项

- **阻塞特性**：如果读进程未启动，写进程会阻塞，直到读进程启动并读取数据。
- **关闭管道**：当管道的读端和写端都关闭后，管道中的数据会被清除。
- **删除管道**：命名管道在文件系统中表现为一个文件，使用完后可以用 `unlink()` 删除文件，避免占用空间。

适用场景

命名管道适用于不同进程之间的简单通信，尤其适合无需双向通信或数据量不大的场景。例如，父进程与子进程之间的消息传递，或简单的跨进程数据传输任务。

线程同步

在多线程编程中，**线程同步** 是为了防止多个线程在访问共享资源时产生冲突，从而确保数据的一致性和正确性。以下是常见的线程同步方法：

1. 互斥锁（Mutex）

- **特点**：互斥锁是最常用的线程同步方式，用于保证同一时刻只有一个线程能够访问临界区。
- **实现**：通过 `pthread_mutex_t`（POSIX）或 `std::mutex`（C++11）实现。
- **应用场景**：适用于需要确保某一资源或临界区只能被一个线程访问的情况。

cpp 复制

```cpp
std::mutex mtx;
mtx.lock();     // 加锁
// 临界区代码
mtx.unlock();   // 解锁
```

2. 读写锁（Read-Write Lock）

- **特点**：允许多个线程同时读取资源，但写操作是独占的，适合读多写少的场景。
- **实现**：通过 `pthread_rwlock_t`（POSIX）或 `std::shared_mutex`（C++17）实现。
- **应用场景**：适用于需要高效读取但写入较少的场景，如缓存访问。

cpp 复制

```cpp
std::shared_mutex rwlock;
rwlock.lock_shared();    // 加共享锁（读锁）
// 读操作
rwlock.unlock_shared();  // 解锁
```

3. 条件变量（Condition Variable）

- **特点**：条件变量用于线程间的等待和通知机制。一个线程可以等待特定条件满足，其他线程在条件满足时通知它。
- **实现**：通过 `pthread_cond_t`（POSIX）或 `std::condition_variable`（C++11）实现。
- **应用场景**：适合生产者-消费者模型等场景，一个线程等待资源就绪，另一个线程通知资源已就绪。

cpp 复制

```cpp
std::condition_variable cv;
std::mutex mtx;
std::unique_lock<std::mutex> lock(mtx);
cv.wait(lock);          // 等待条件
cv.notify_one();        // 通知一个线程
```

4. 信号量（Semaphore）

- **特点**：信号量用于限制可以访问共享资源的线程数量。计数信号量允许多线程同时进入临界区，二进制信号量相当于互斥锁。
- **实现**：通过 `sem_t`（POSIX）或 `std::counting_semaphore`（C++20）实现。
- **应用场景**：适用于资源有限的情况，如连接池中控制最大并发数。

cpp 复制

```cpp
sem_t semaphore;
sem_init(&semaphore, 0, 2);   // 初始化信号量，初始计数为 2
sem_wait(&semaphore);         // 获取信号量
// 临界区
sem_post(&semaphore);         // 释放信号量
```