并发：基于事件的并发

邰颖
南京大学
智能科学与技术学院

基于事件的并发

- 一种不同风格的**并发编程**
  - 用于**图形用户界面（GUI）应用程序**和一些类型的**互联网服务器**

- 基于事件的并发解决的问题是双重的：
  - **正确管理多线程应用程序中的并发**
    - 可能出现的问题：缺少锁、死锁和其他严重问题。
  - **开发者对在特定时刻安排执行的任务几乎没有控制权。**

基本想法：事件循环

- **方法：**
  - **等待**某些事件发生（即，"事件"发生）
  - 当事件发生时，**检查**它是哪种类型的事件
  - **执行**它所需的少量工作

- **示例：**一个典型的基于事件的服务器，基于一个简单的结构：事件循环

```
while(1){
    events = getEvents();   // 主循环等待某些事件发生
    for(e in events)
        processEvent(e);    // 事件处理程序
}
```

基于事件的服务器如何决定哪个事件发生？

重要API：select()（或poll()）

•检查是否有任何**传入的I/O**需要处理：**select()**，返回所有集合中就绪描述符的总数

```
int select(int nfds,
           fd_set * restrict readfds,
           fd_set * restrict writefds,
           fd_set * restrict errorfds,
           struct timeval * restrict timeout);
```

•**readfds**：让服务器确定是否有**新数据包到达**并需要处理。

•**writefds**：指定**写入的文件描述符**，让服务知道何时可以回复。

•**timeout**：最大等待时间

- **NULL**：使select()调用**无限期阻塞**，直到某个描述符准备好。
- **0**：使select()调用**立即返回**。

使用 select()

- 如何使用 select() 来检查哪些网络描述符有传入的消息。

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    // 打开并设置一些套接字（此部分未展示）
    // 主循环
    while (1) {
        // 将 fd_set 初始化为全零
        fd_set readFDs;
        FD_ZERO(&readFDs);
```

// 使用 FD_ZERO(&readFDs) 清空文件描述符集合

```c
        // 设置描述符集合，表示服务器关心的描述符
        int fd;
        for (fd = minFD; fd < maxFD; fd++) {
            FD_SET(fd, &readFDs);
        }

        // 调用 select 阻塞直到有描述符准备就绪
        int rc = select(maxFD + 1, &readFDs, NULL, NULL, NULL);

        // 使用 FD_ISSET 检查哪些描述符有数据
        for (fd = minFD; fd < maxFD; fd++) {
            if (FD_ISSET(fd, &readFDs)) {
                processFD(fd);  // 处理有数据的描述符
            }
        }
    }
}
```

// 设置需要监视的文件描述符集合

// 判断每个描述符是否准备好

为什么更简单？不需要锁

- 基于事件的服务器**无法被另一个线程中断**
  - 在**单个CPU**和**事件驱动应用程序**的环境下
  - 它是**单线程**
  - 因此，**多线程程序中常见的并发错误**在**基本的事件驱动方法**中不会表现出来。

一个问题：阻塞的系统调用

•如果某个事件要求你发出可能会阻塞的系统调用（例如 read）怎么办？

- 没有其他线程可供执行：**只有主事件循环**
- 整个服务器：**阻塞直到调用完成**
- **可能造成巨大的资源浪费**

在基于事件的系统中：不允许阻塞调用

解决方案：异步I/O

•使得应用程序能够发出I/O请求，并在I/O操作完成之前**立即将控制权返回给调用者**。

```
struct aiocb {
    int aio_fildes;     /* 文件描述符 */
    off_t aio_offset;   /* 文件偏移量 */
    volatile void *aio_buf;    /* 缓冲区位置 */
    size_t aio_nbytes;  /* 传输字节数 */
};
```

- 该接口在 **Mac OS X** 上提供
- 这些API围绕一个基本结构展开，即 **struct aiocb** 或常用术语中的 **AIO控制块（AIO control block）**

解决方案：异步I/O（续）

•异步API：

- 发起一个异步读取文件的操作：

```
int aio_read(struct aiocb *aiocbp);
```

  - 如果成功，立即返回，应用程序可以继续处理其他工作。

- 检查由 aiocbp 指向的请求是否完成：

```
int aio_error(const struct aiocb *aiocbp);
```

  - 应用程序可以通过 aio_error() **定期轮询系统**。
  - 如果请求已完成，返回成功。
  - 如果未完成，返回 EINPROGRESS。

解决方案：异步I/O（续）

- 中断
  - 为了弥补**检查I/O是否完成的开销**
  - 使用 **UNIX 信号** 通知应用程序异步I/O完成时
  - 消除**反复询问系统**的需求

另一个问题：状态管理

- 基于事件的方法通常比**传统的基于线程的代码更复杂**：
  - 它必须为下一个事件处理程序打包一些程序状态，以便在I/O完成时使用。
  - 与使用线程相比，**需要手动管理栈**。

另一个问题：状态管理（续）

- 示例（一个基于线程的服务器）：

```
int rc = read(fd, buffer, size);
rc = write(sd, buffer, size);
```

- 一个基于事件的服务器：无法像基于线程的服务器中连续调用
  - 首先异步地发起读取操作。
  - 然后定期检查读取操作是否完成。
  - 当读取完成时，调用会通知我们读取操作已完成。
  - **基于事件的服务器如何知道接下来应该做什么？**

另一个问题：状态管理（续）

- 一种解决方案：延续（Continuation）机制保存状态
  - •**记录**把后续动作等作为状态信息，记录在某种数据结构
  - •当事件发生时（即当磁盘I/O完成时），**查找**所需的信息并处理该事件。
  - •例子：
    - •sd记录在由fd索引的某种数据结构（如hash表）
    - •当磁盘I/O完成时，事件处理程序将使用文件描述符查找延续信息
    - •服务器完成最后的工作：将数据写入sd

基于事件的并发难点

- 系统从单核CPU迁移到**多核CPU**：
  - 一些事件驱动方式的简单性消失；
  - 由于多个事件处理程序在不同核心上运行，常见的同步问题重新出现。

- 与某些系统活动的**集成不好**：
  - 例如分页：服务器在页错误完成之前无法继续处理（隐式阻塞）。

- 长时间管理困难：不同方法的语义变化：
  - 从非阻塞切换到阻塞的例程需要在事件处理程序中做出更改。

- 异步磁盘I/O**无法简单和统一地与异步网络I/O集成**：
  - 需要将select()与AIO结合使用。

小结

- 介绍了基于事件的并发
  - 基本想法：事件循环
  - select()检查文件描述符集合
  - 异步I/O：处理阻塞的系统调用
  - 状态管理：延续机制
  - 仍存在的一些挑战