第 33 章　基于事件的并发（进阶）

目前为止，我们提到的并发，似乎只能用线程来实现。就像生活中的许多事，这不完全对。具体来说，一些基于图形用户界面（GUI）的应用[O96]，或某些类型的网络服务器[PDZ99]，常常采用另一种并发方式。这种方式称为基于事件的并发（event-based concurrency），在一些现代系统中较为流行，比如 node.js[N13]，但它源自于 C/UNIX 系统，我们下面将讨论。

基于事件的并发针对两方面的问题。一方面是多线程应用中，正确处理并发很有难度。正如我们讨论的，忘加锁、死锁和其他烦人的问题会发生。另一方面，开发者无法控制多线程在某一时刻的调度。程序员只是创建了线程，然后就依赖操作系统能够合理地调度线程。要实现一个在各种不同负载下，都能够良好运行的通用调度程序，是极有难度的。因此，某些时候操作系统的调度并不是最优的。关键问题如下。

关键问题：不用线程，如何构建并发服务器

不用线程，同时保证对并发的控制，避免多线程应用中出现的问题，我们应该如何构建一个并发服务器？

33.1　基本想法：事件循环

我们使用的基本方法就是基于事件的并发（event-based concurrency）。该方法很简单：我们等待某事（即“事件”）发生；当它发生时，检查事件类型，然后做少量的相应工作（可能是 I/O 请求，或者调度其他事件准备后续处理）。这就好了！

在深入细节之前，我们先看一个典型的基于事件的服务器。这种应用都是基于一个简单的结构，称为事件循环（event loop）。事件循环的伪代码如下：

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

它确实如此简单。主循环等待某些事件发生（通过 getEvents()调用），然后依次处理这些发生的事件。处理事件的代码叫作事件处理程序（event handler）。重要的是，处理程序在处理一个事件时，它是系统中发生的唯一活动。因此，调度就是决定接下来处理哪个事件。这种对调度的显式控制，是基于事件方法的一个重要优点。

但这也带来一个更大的问题：基于事件的服务器如何决定哪个事件发生，尤其是对于

网络和磁盘 I/O？具体来说，事件服务器如何确定是否有它的消息已经到达？

33.2 重要 API：select()（或 poll()）

知道了基本的事件循环，我们接下来必须解决如何接收事件的问题。大多数系统提供了基本的 API，即通过 select()或 poll()系统调用。

这些接口对程序的支持很简单：检查是否有任何应该关注的进入 I/O。例如，假定网络应用程序（如 Web 服务器）希望检查是否有网络数据包已到达，以便为它们提供服务。这些系统调用就让你做到这一点。

下面以 select()为例，手册页（在 macOS X 上）以这种方式描述 API：

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

手册页中的实际描述：select()检查 I/O 描述符集合，它们的地址通过 readfds、writefds 和 errorfds 传入，分别查看它们中的某些描述符是否已准备好读取，是否准备好写入，或有异常情况待处理。在每个集合中检查前 nfds 个描述符，即检查描述符集合中从 0 到 nfds-1 的描述符。返回时，select()用给定请求操作准备好的描述符组成的子集替换给定的描述符集合。select()返回所有集合中就绪描述符的总数。

补充：阻塞与非阻塞接口

阻塞（或同步，synchronous）接口在返回给调用者之前完成所有工作。非阻塞（或异步，asynchronous）接口开始一些工作，但立即返回，从而让所有需要完成的工作都在后台完成。

通常阻塞调用的主犯是某种 I/O。例如，如果一个调用必须从磁盘读取才能完成，它可能会阻塞，等待发送到磁盘的 I/O 请求返回。

非阻塞接口可用于任何类型的编程（例如，使用线程），但在基于事件的方法中非常重要，因为阻塞的调用会阻止所有进展。

关于 select()有几点要注意。首先，请注意，它可以让你检查描述符是否可以读取和写入。前者让服务器确定新数据包已到达并且需要处理，而后者则让服务知道何时可以回复（即出站队列未满）。

其次，请注意超时参数。这里的一个常见用法是将超时设置为 NULL，这会导致 select()无限期地阻塞，直到某个描述符准备就绪。但是，更健壮的服务器通常会指定某种超时。一种常见的技术是将超时设置为零，因此让调用 select()立即返回。

poll()系统调用非常相似。有关详细信息，请参阅其手册页或 Stevens 和 Rago 的书[SR05]。

无论哪种方式，这些基本原语为我们提供了一种构建非阻塞事件循环的方法，它可以简单地检查传入数据包，从带有消息的套接字中读取数据，并根据需要进行回复。

33.3 使用 select()

为了让这更具体，我们来看看如何使用 select()来查看哪些网络描述符在它们上面有传入消息。图 33.1 展示了一个简单的例子。

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <sys/time.h>
4    #include <sys/types.h>
5    #include <unistd.h>
6
7    int main(void) {
8        // open and set up a bunch of sockets (not shown)
9        // main loop
10       while (1) {
11           // initialize the fd_set to all zero
12           fd_set readFDs;
13           FD_ZERO(&readFDs);
14
15           // now set the bits for the descriptors
16           // this server is interested in
17           // (for simplicity, all of them from min to max)
18           int fd;
19           for (fd = minFD; fd < maxFD; fd++)
20               FD_SET(fd, &readFDs);
21
22           // do the select
23           int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);
24
25           // check which actually have data using FD_ISSET()
26           int fd;
27           for (fd = minFD; fd < maxFD; fd++)
28               if (FD_ISSET(fd, &readFDs))
29                   processFD(fd);
30       }
31   }
```

图 33.1 使用 select()的简单代码

这段代码实际上很容易理解。初始化完成后，服务器进入无限循环。在循环内部，它使用 FD_ZERO()宏首先清除文件描述符集合，然后使用 FD_SET()将所有从 minFD 到 maxFD 的文件描述符包含到集合中。例如，这组描述符可能表示服务器正在关注的所有网络套接字。最后，服务器调用 select()来查看哪些连接有可用的数据。然后，通过在循环中使用 FD_ISSET()，事件服务器可以查看哪些描述符已准备好数据并处理传入的数据。

当然，真正的服务器会比这更复杂，并且在发送消息、发出磁盘 I/O 和许多其他细节时需要使用逻辑。想了解更多信息，请参阅 Stevens 和 Rago 的书 [SR05]，了解 API 信息，或

Pai 等人的论文、Welsh 等人的论文[PDZ99，WCB01]，以便对基于事件的服务器的一般流程有一个很好的总体了解。

33.4 为何更简单？无须锁

使用单个 CPU 和基于事件的应用程序，并发程序中发现的问题不再存在。具体来说，因为一次只处理一个事件，所以不需要获取或释放锁。基于事件的服务器不能被另一个线程中断，因为它确实是单线程的。因此，线程化程序中常见的并发性错误并没有出现在基本的基于事件的方法中。

提示：请勿阻塞基于事件的服务器

基于事件的服务器可以对任务调度进行细粒度的控制。但是，为了保持这种控制，不可以有阻止调用者执行的调用。如果不遵守这个设计提示，将导致基于事件的服务器阻塞，客户心塞，并严重质疑你是否读过本书的这部分内容。

33.5 一个问题：阻塞系统调用

到目前为止，基于事件的编程听起来很棒，对吧？编写一个简单的循环，然后在事件发生时处理事件。甚至不需要考虑锁！但是有一个问题：如果某个事件要求你发出可能会阻塞的系统调用，该怎么办？

例如，假定一个请求从客户端进入服务器，要从磁盘读取文件并将其内容返回给发出请求的客户端（很像简单的 HTTP 请求）。为了处理这样的请求，某些事件处理程序最终将不得不发出 open()系统调用来打开文件，然后通过一系列 read()调用来读取文件。当文件被读入内存时，服务器可能会开始将结果发送到客户端。

open()和 read()调用都可能向存储系统发出 I/O 请求（当所需的元数据或数据不在内存中时），因此可能需要很长时间才能提供服务。使用基于线程的服务器时，这不是问题：在发出 I/O 请求的线程挂起（等待 I/O 完成）时，其他线程可以运行，从而使服务器能够取得进展。事实上，I/O 和其他计算的自然重叠（overlap）使得基于线程的编程非常自然和直接。

但是，使用基于事件的方法时，没有其他线程可以运行：只是主事件循环。这意味着如果一个事件处理程序发出一个阻塞的调用，整个服务器就会这样做：阻塞直到调用完成。当事件循环阻塞时，系统处于闲置状态，因此是潜在的巨大资源浪费。因此，我们在基于事件的系统中必须遵守一条规则：不允许阻塞调用。

33.6 解决方案：异步 I/O

为了克服这个限制，许多现代操作系统已经引入了新的方法来向磁盘系统发出 I/O 请求，

一般称为异步 I/O（asynchronous I/O）。这些接口使应用程序能够发出 I/O 请求，并在 I/O 完成之前立即将控制权返回给调用者，另外的接口让应用程序能够确定各种 I/O 是否已完成。

例如，让我们来看看在 macOS X 上提供的接口（其他系统有类似的 API）。这些 API 围绕着一个基本的结构，即 struct aiocb 或 AIO 控制块（AIO control block）。该结构的简化版本如下所示（有关详细信息，请参阅手册页）：

```
struct aiocb {
    int              aio_fildes;    /* File descriptor */
    off_t            aio_offset;    /* File offset */
    volatile void    *aio_buf;      /* Location of buffer */
    size_t           aio_nbytes;    /* Length of transfer */
};
```

要向文件发出异步读取，应用程序应首先用相关信息填充此结构：要读取文件的文件描述符（aio_fildes），文件内的偏移量（ai_offset）以及长度的请求（aio_nbytes），最后是应该复制读取结果的目标内存位置（aio_buf）。

在填充此结构后，应用程序必须发出异步调用来读取文件。在 macOS X 上，此 API 就是异步读取（asynchronous read）API：

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

该调用尝试发出 I/O。如果成功，它会立即返回并且应用程序（即基于事件的服务器）可以继续其工作。

然而，我们必须解决最后一个难题。我们如何知道 I/O 何时完成，并且缓冲区（由 aio buf 指向）现在有了请求的数据？

还需要最后一个 API。在 macOS X 上，它被称为 aio_error()（有点令人困惑）。API 看起来像这样：

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

该系统调用检查 aiocbp 引用的请求是否已完成。如果有，则函数返回成功（用零表示）。如果不是，则返回 EINPROGRESS。因此，对于每个未完成的异步 I/O，应用程序可以通过调用 aio_error()来周期性地轮询（poll）系统，以确定所述 I/O 是否尚未完成。

你可能已经注意到，检查一个 I/O 是否已经完成是很痛苦的。如果一个程序在某个特定时间点发出数十或数百个 I/O，是否应该重复检查它们中的每一个，或者先等待一会儿，或者……

为了解决这个问题，一些系统提供了基于中断（interrupt）的方法。此方法使用 UNIX 信号（signal）在异步 I/O 完成时通知应用程序，从而消除了重复询问系统的需要。这种轮询与中断问题也可以在设备中看到，正如你将在 I/O 设备章节中看到的（或已经看到的）。

补充：UNIX 信号

所有现代 UNIX 变体都有一个称为信号（signal）的巨大而迷人的基础设施。最简单的信号提供了一种与进程进行通信的方式。具体来说，可以将信号传递给应用程序。这样做会让应用程序停止当前的任何工作，开始运行信号处理程序（signal handler），即应用程序中某些处理该信号的代码。完成后，该进程就恢复其先前的行为。

每个信号都有一个名称，如 HUP（挂断）、INT（中断）、SEGV（段违规）等。有关详细信息，请参阅手册页。有趣的是，有时是内核本身发出信号。例如，当你的程序遇到段违规时，操作系统发送一个 SIGSEGV（在信号名称之前加上 SIG 是很常见的）。如果你的程序配置为捕获该信号，则实际上可以运行一些代码来响应这种错误的程序行为（这可能对调试有用）。当一个信号被发送到一个没有配置处理该信号的进程时，一些默认行为就会生效。对于 SEGV 来说，这个进程会被杀死。

下面一个进入无限循环的简单程序，但首先设置一个信号处理程序来捕捉 SIGHUP：

```
#include <stdio.h>
#include <signal.h>

void handle(int arg) {
    printf("stop wakin' me up...\n");
}

int main(int argc, char *argv[]) {
    signal(SIGHUP, handle);
    while (1)
        ; // doin' nothin' except catchin' some sigs
    return 0;
}
```

你可以用 kill 命令行工具向其发送信号（是的，这是一个奇怪而富有攻击性的名称）。这样做会中断程序中的主 while 循环并运行处理程序代码 handle()：

```
prompt> ./main &
[3] 36705
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
prompt> kill -HUP 36705
stop wakin' me up...
```

要了解信号还有很多事情要做，以至于单个页面，甚至单独的章节，都远远不够。与往常一样，有一个重要来源：Stevens 和 Rago 的书[SR05]。如果感兴趣，请阅读。

在没有异步 I/O 的系统中，纯基于事件的方法无法实现。然而，聪明的研究人员已经推出了相当适合他们的方法。例如，Pai 等人 [PDZ99]描述了一种使用事件处理网络数据包的混合方法，并且使用线程池来管理未完成的 I/O。详情请阅读他们的论文。

33.7 另一个问题：状态管理

基于事件的方法的另一个问题是，这种代码通常比传统的基于线程的代码更复杂。原因如下：当事件处理程序发出异步 I/O 时，它必须打包一些程序状态，以便下一个事件处理

程序在 I/O 最终完成时使用。这个额外的工作在基于线程的程序中是不需要的，因为程序需要的状态在线程栈中。Adya 等人称之为手工栈管理（manual stack management），这是基于事件编程的基础[A + 02]。

为了使这一点更加具体一些，我们来看一个简单的例子，在这个例子中，一个基于线程的服务器需要从文件描述符（fd）中读取数据，一旦完成，将从文件中读取的数据写入网络套接字描述符 SD）。代码（忽略错误检查）如下所示：

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

如你所见，在一个多线程程序中，做这种工作很容易。当 read()最终返回时，代码立即知道要写入哪个套接字，因为该信息位于线程堆栈中（在变量 sd 中）。

在基于事件的系统中，生活并没有那么容易。为了执行相同的任务，我们首先使用上面描述的 AIO 调用异步地发出读取。假设我们使用 aio_error()调用定期检查读取的完成情况。当该调用告诉我们读取完成时，基于事件的服务器如何知道该怎么做？

解决方案，如 Adya 等人[A+02]所述，是使用一种称为“延续（continuation）”的老编程语言结构[FHK84]。虽然听起来很复杂，但这个想法很简单：基本上，在某些数据结构中，记录完成处理该事件需要的信息。当事件发生时（即磁盘 I/O 完成时），查找所需信息并处理事件。

在这个特定例子中，解决方案是将套接字描述符（sd）记录在由文件描述符（fd）索引的某种数据结构（例如，散列表）中。当磁盘 I/O 完成时，事件处理程序将使用文件描述符来查找延续，这会将套接字描述符的值返回给调用者。此时（最后），服务器可以完成最后的工作将数据写入套接字。

33.8 什么事情仍然很难

基于事件的方法还有其他一些困难，我们应该指出。例如，当系统从单个 CPU 转向多个 CPU 时，基于事件的方法的一些简单性就消失了。具体来说，为了利用多个 CPU，事件服务器必须并行运行多个事件处理程序。发生这种情况时，就会出现常见的同步问题（例如临界区），并且必须采用通常的解决方案（例如锁定）。因此，在现代多核系统上，无锁的简单事件处理已不再可能。

基于事件的方法的另一个问题是，它不能很好地与某些类型的系统活动集成，如分页（paging）。例如，如果事件处理程序发生页错误，它将被阻塞，并且因此服务器在页错误完成之前不会有进展。尽管服务器的结构可以避免显式阻塞，但由于页错误导致的这种隐式阻塞很难避免，因此在频繁发生时可能会导致较大的性能问题。

还有一个问题是随着时间的推移，基于事件的代码可能很难管理，因为各种函数的确切语义发生了变化[A+02]。例如，如果函数从非阻塞变为阻塞，调用该例程的事件处理程序也必须更改以适应其新性质，方法是将其自身分解为两部分。由于阻塞对于基于事件的服务器而言是灾难性的，因此程序员必须始终注意每个事件使用的 API 语义的这种变化。

最后，虽然异步磁盘 I/O 现在可以在大多数平台上使用，但是花了很长时间才做到这一点[PDZ99]，而且与异步网络 I/O 集成不会像你想象的那样有简单和统一的方式。例如，虽然人们只想使用 select()接口来管理所有未完成的 I/O，但通常需要组合用于网络的 select()和用于磁盘 I/O 的 AIO 调用。

33.9 小结

我们已经介绍了不同风格的基于事件的并发。基于事件的服务器为应用程序本身提供了调度控制，但是这样做的代价是复杂性以及与现代系统其他方面（例如分页）的集成难度。由于这些挑战，没有哪一种方法表现最好。因此，线程和事件在未来很多年内可能会持续作为解决同一并发问题的两种不同方法。阅读一些研究论文（例如[A+02，PDZ99，vB+03，WCB01]），或者写一些基于事件的代码，以了解更多信息，这样更好。

参考资料

[A+02]“Cooperative Task Management Without Manual Stack Management”Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, John R. Douceur USENIX ATC ’02, Monterey, CA, June 2002
这篇论文首次明确阐述了基于事件的并发的一些困难，并提出了一些简单的解决方案，同时也探讨了将两种并发管理整合到单个应用程序中的更疯狂的想法！

[FHK84]“Programming With Continuations”
Daniel P. Friedman, Christopher T. Haynes, Eugene E. Kohlbecker
In Program Transformation and Programming Environments, Springer Verlag, 1984
这是来自编程语言世界的这个老思想的经典参考。现在在一些现代语言中越来越流行。

[N13]“Node.js Documentation”By the folks who build node.js
许多很酷的新框架之一，可帮助你轻松构建 Web 服务和应用程序。每个现代系统黑客都应该精通这样的框架（可能还有多个框架）。花时间在这些世界中的一个进行开发，并成为专家。

[O96]“Why Threads Are A Bad Idea (for most purposes)”John Ousterhout
Invited Talk at USENIX ’96, San Diego, CA, January 1996
关于线程如何与基于 GUI 的应用程序不太匹配的一次很好的演讲（但是这些想法更通用）。Ousterhout 在开发 Tcl/Tk 时，形成了这些观点中的大部分，Tcl/Tk 是一种很酷的脚本语言和工具包，与当时最先进的技术相比，开发基于 GUI 的应用程序要容易 100 倍。虽然 Tk GUI 工具箱继续存在（例如在 Python 中），但 Tcl 似乎正在慢慢死去（很遗憾）。

[PDZ99]“Flash: An Efficient and Portable Web Server”Vivek S. Pai, Peter Druschel, Willy Zwaenepoel USENIX ’99, Monterey, CA, June 1999

关于如何在当今新兴的互联网时代构建 Web 服务器的开创性论文。阅读它以了解基础知识，并了解作者在缺乏对异步 I/O 支持时如何构建混合体系的想法。

[SR05]“Advanced Programming in the UNIX Environment”
W. Richard Stevens and Stephen A. Rago Addison-Wesley, 2005
UNIX 系统编程的经典必备图书。如果你需要知道一些细节，就在这里。

[vB+03]“Capriccio: Scalable Threads for Internet Services”
Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer SOSP ’03, Lake George, New York, October 2003
一篇关于如何使线程在极端规模下工作的论文，这是当时正在进行的所有基于事件的工作的反驳。

[WCB01]“SEDA: An Architecture for Well-Conditioned, Scalable Internet Services”Matt Welsh, David Culler, and Eric Brewer
SOSP ’01, Banff, Canada, October 2001
基于事件的服务的一个不错的变通，它将线程、队列和基于事件的处理合并为一个简化的整体。其中一些想法已经进入谷歌、亚马逊等公司的基础设施。