深入理解 I/O 模型

海纳

目录

同步I/O模型的基本原理

异步I/O模型原理

I/O 多路复用的基本原理

实现 EventLoop

同步 I/O 模型的基本原理

I/O 模型的基本概念

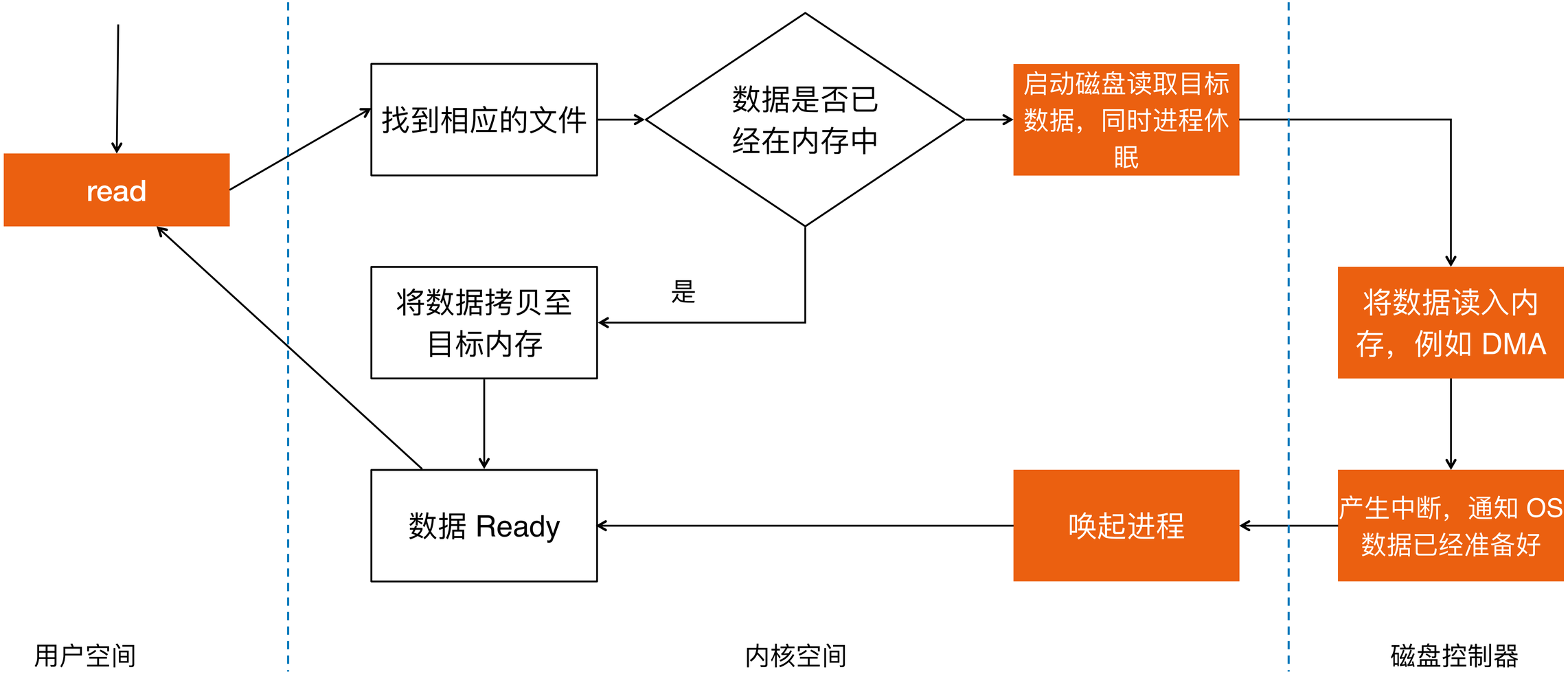
输入输出（Input/Output, I/O ）是以 CPU 为中心视角的概念。当数据是从外设流向 CPU 时，就称为输入，例如键盘，磁盘读，从网卡上接受数据都是输入。当数据是从 CPU 流向外设时，就称为输出，例如屏幕显示，磁盘写，向网卡发送数据都是输出。

显然， I/O 是由计算机硬件和软件协同工作完成的，由于涉及到输入输出设备的驱动，所以输入输出的软件模型主要是由操作系统定义。下面简要介绍两个概念。

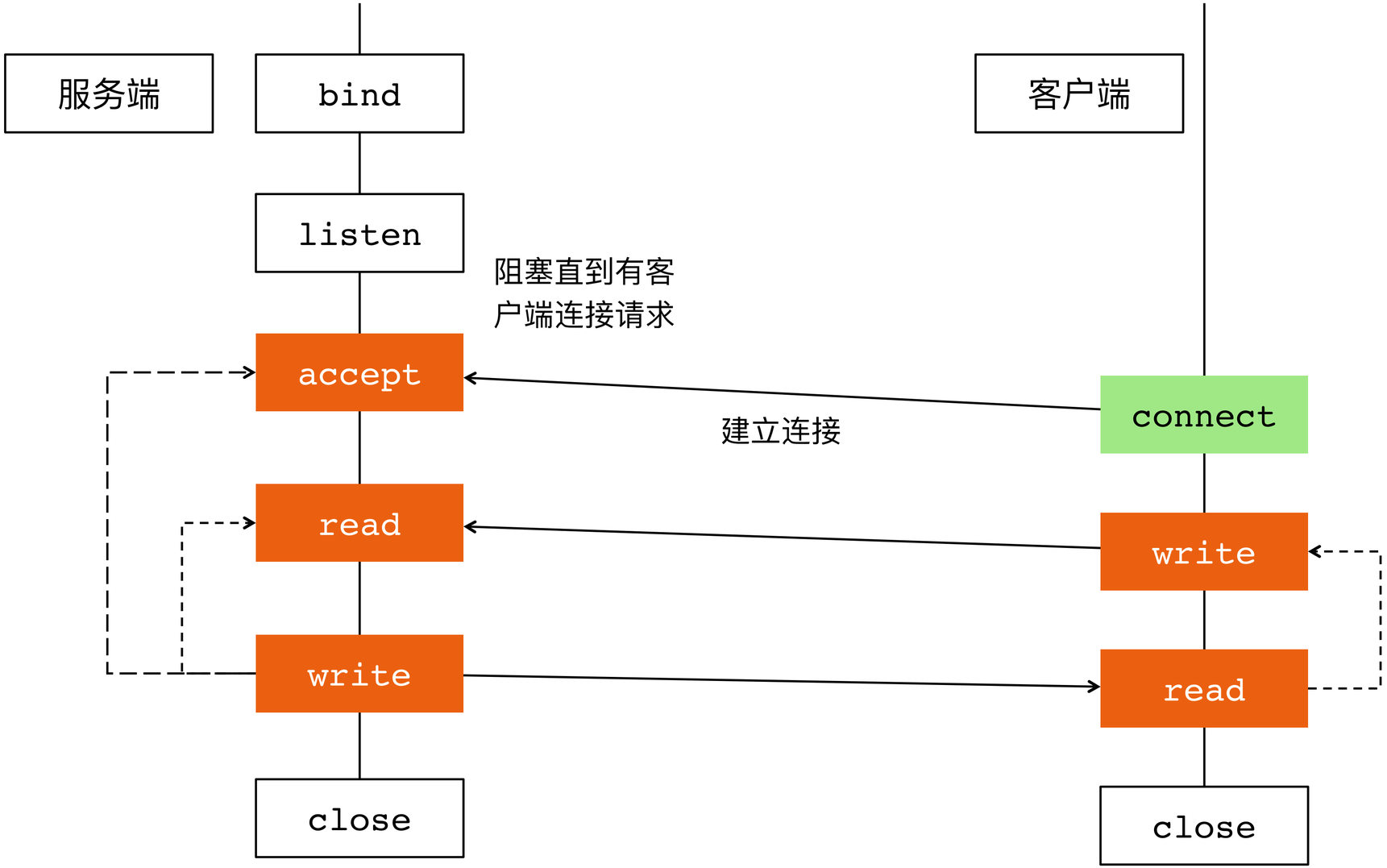
阻塞式 I/O（Blocking ），是指在进行 I/O 的时候会暂停进程，在等待的数据没到达之前进程不会执行。

非阻塞式 I/O（Nonblocking），是指在进行 I/O 的时候进程还可以继续执行，比如可以运行其他的任务。

阻塞产生的原因


read
找到相应的文件
数据是否已
经在内存中
启动磁盘读取目标
数据，同时进程休
眠
将数据读入内
存，例如 DMA
产生中断，通知 OS
数据已经准备好
唤起进程
数据 Ready
将数据拷贝至
目标内存
是
用户空间
内核空间
磁盘控制器

阻塞式 I/O 的例子


服务端
bind
listen
accept
read
write
close
客户端
connect
write
read
close
阻塞直到有客
户端连接请求
建立连接

解决方案一：多线程

服务端可以开启多个线程，每个线程服务一个客户端。这样阻塞的就是子线程，而主线程不再被阻塞。

服务端的单一线程和客户端的逻辑是同步的：即服务端线程在没有接受到客户端消息之前不会去做其他事情。每人一步，协调一致地前进，所以这种方式称为同步 I/O 。从这个例子中可以看到同步 I/O 往往和阻塞式 I/O 是可以混用的，但要注意阻塞的实体是进程还是线程。

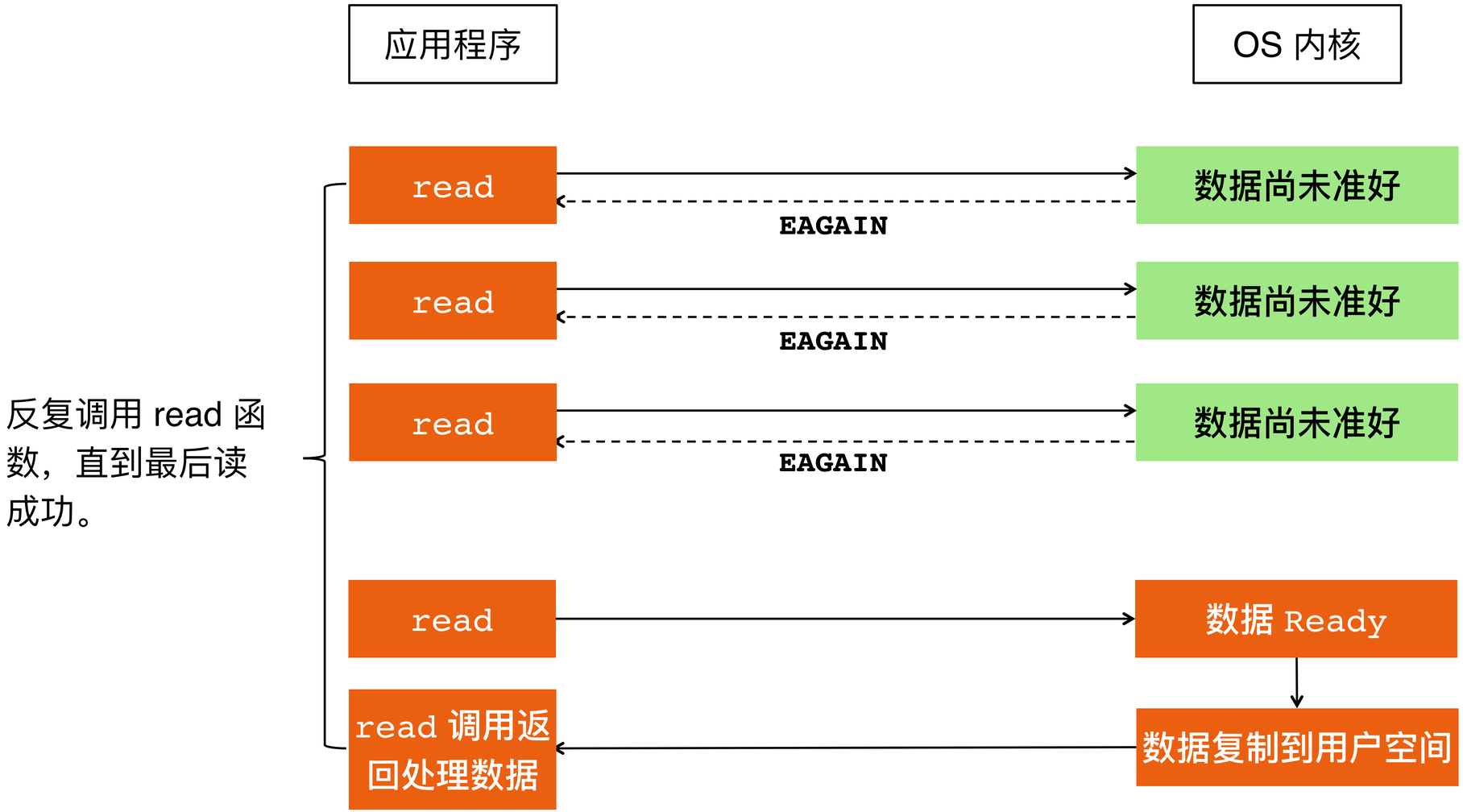
```
while (1) {
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);

    if (!fork()) {
        talk_to_client(clnt_sock);
        exit(0);
    }
}
```

解决方案二：非阻塞 I/O 轮询

可以将 socket 配置成 NONBLOCKING，accept，read, write 等函数就不会阻塞线程了。但是非阻塞式 I/O 接口不能保证一定成功，比如 read 可能返回-1，表示这一次调用没有读到想要的数据。必须通过 errno 的值综合判断。这种轮询所有 socket 的做法虽然不再阻塞线程，但却大量消耗 CPU 时间，其本质还是服务端和客户端以同步的方式进行通讯。

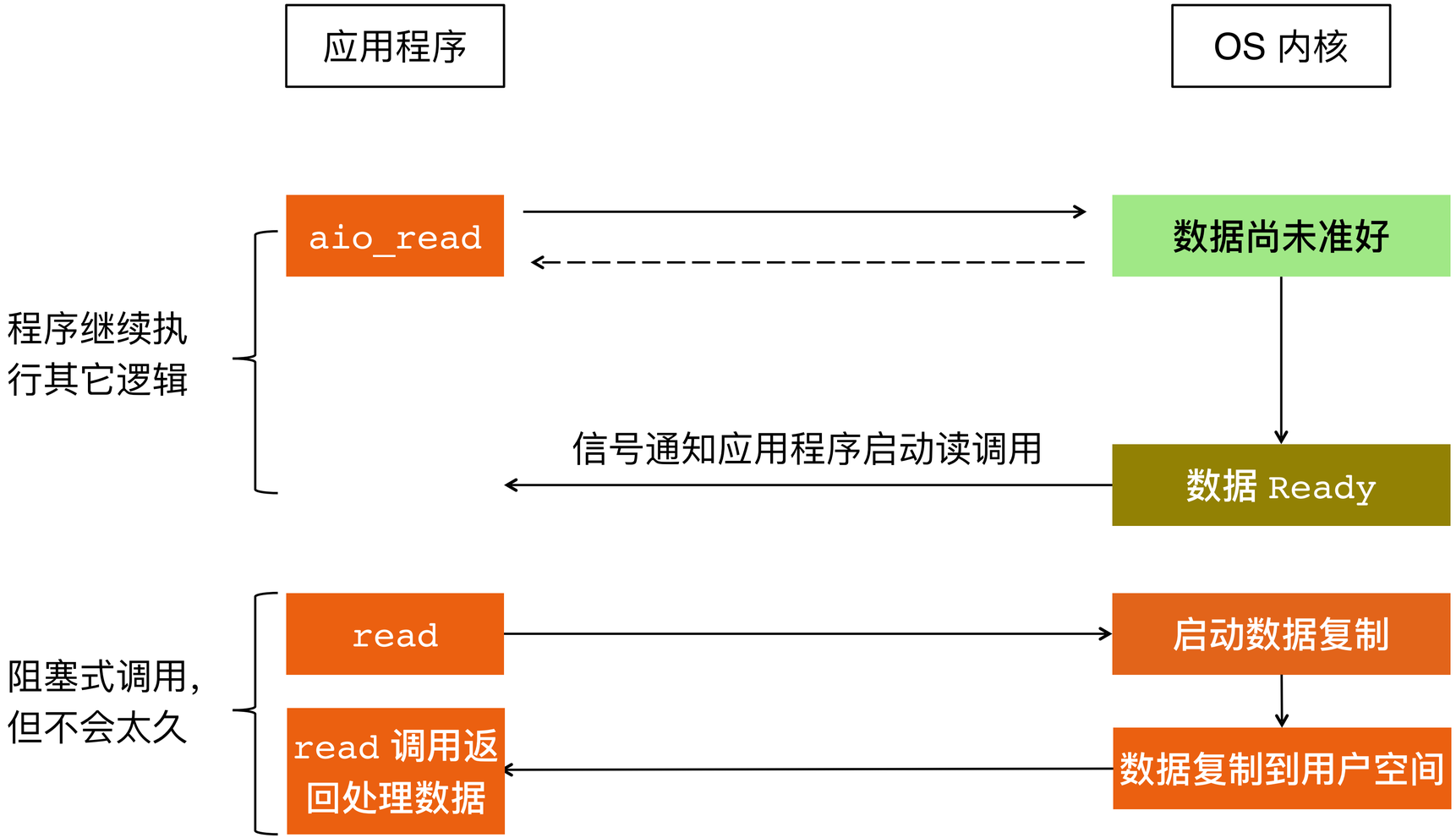
```
while (1) {
    errno = 0;
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
    if (clnt_sock < 0 && errno == 11) {
        continue; // do nothing
    }
    else {
        flags = fcntl(clnt_sock, F_GETFL, 0);
        fcntl(clnt_sock, F_SETFL, flags|O_NONBLOCK);
        socks[index++] = clnt_sock;
    }

    for (i = 0; i < N; i++) {
        if (socks[i]) {
            talk_to_client(socks[i], i);
        }
    }
}
```

应用程序轮询流程图


应用程序
OS 内核
read
数据尚未准好
EAGAIN
read
数据尚未准好
EAGAIN
read
数据尚未准好
EAGAIN
反复调用 read 函数，直到最后读成功。
read
数据 Ready
read 调用返回处理数据
数据复制到用户空间

异步 I/O 的基本概念

基于信号的 I/O


应用程序
OS 内核
aio_read
数据尚未准好
程序继续执
行其它逻辑
信号通知应用程序启动读调用
数据 Ready
read
启动数据复制
阻塞式调用，
但不会太久
read 调用返
回处理数据
数据复制到用户空间

基于信号的 I/O 特点分析

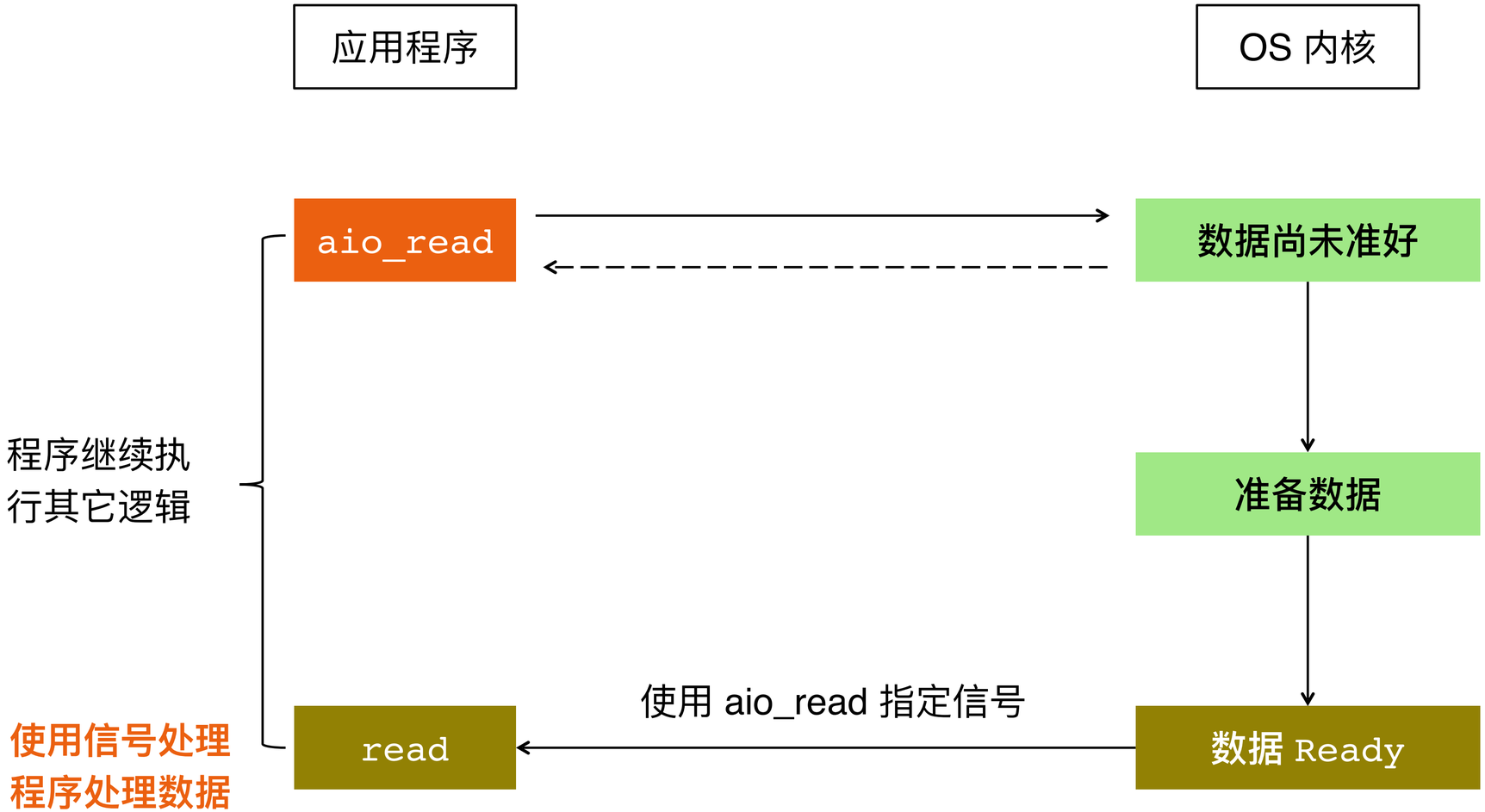
基于性号的 I/O 接口允许线程在发起读请求以后去做其他的事情，这个设计看起来很美好，但是真正实现起来就会发现，**这个线程其实并没有其他的事情可以做**。

可以想象，开发者不太可能把业务线程和 I/O 线程混在一起，让 I/O 线程停下来去做业务，这是信号 I/O 接口设计的初衷，但几乎没有人会这么做。所以，信号 I/O 在 Linux 上一直都没有得到有效的支持（使用 SIGIO 信号当然可以实现，但没人会这么做）。

而真正的异步的 I/O ，例如 POSIX AIO，以及 Windows 系统上的 iocp，是现实中比较常见的。Linux 的 AIO 实现是使用 I/O 多路复用进行封装得到的。所以在 Linux 服务器上， I/O 多路复用是比较重要的。尽管如此，AIO 的原理还是需要掌握的，因为新的 io_uring 是基于 AIO 的思想实现的。

io_uring 我们的课程暂不涉及。有兴趣的同学可以在掌握了 AIO 的原理以后自行学习。

异步 I/O


应用程序
OS 内核
aio_read
数据尚未准好
程序继续执
行其它逻辑
准备数据
使用 aio_read 指定信号
使用信号处理
程序处理数据
read
数据 Ready

I/O 多路复用

操作系统替应用程序高效轮询

在第一节的轮询方案中，我们展示了一个应用程序主动轮询的例子。本节课的核心重点 I/O 多路利用其本质就是操作系统代替应用程序做轮询（polling）。

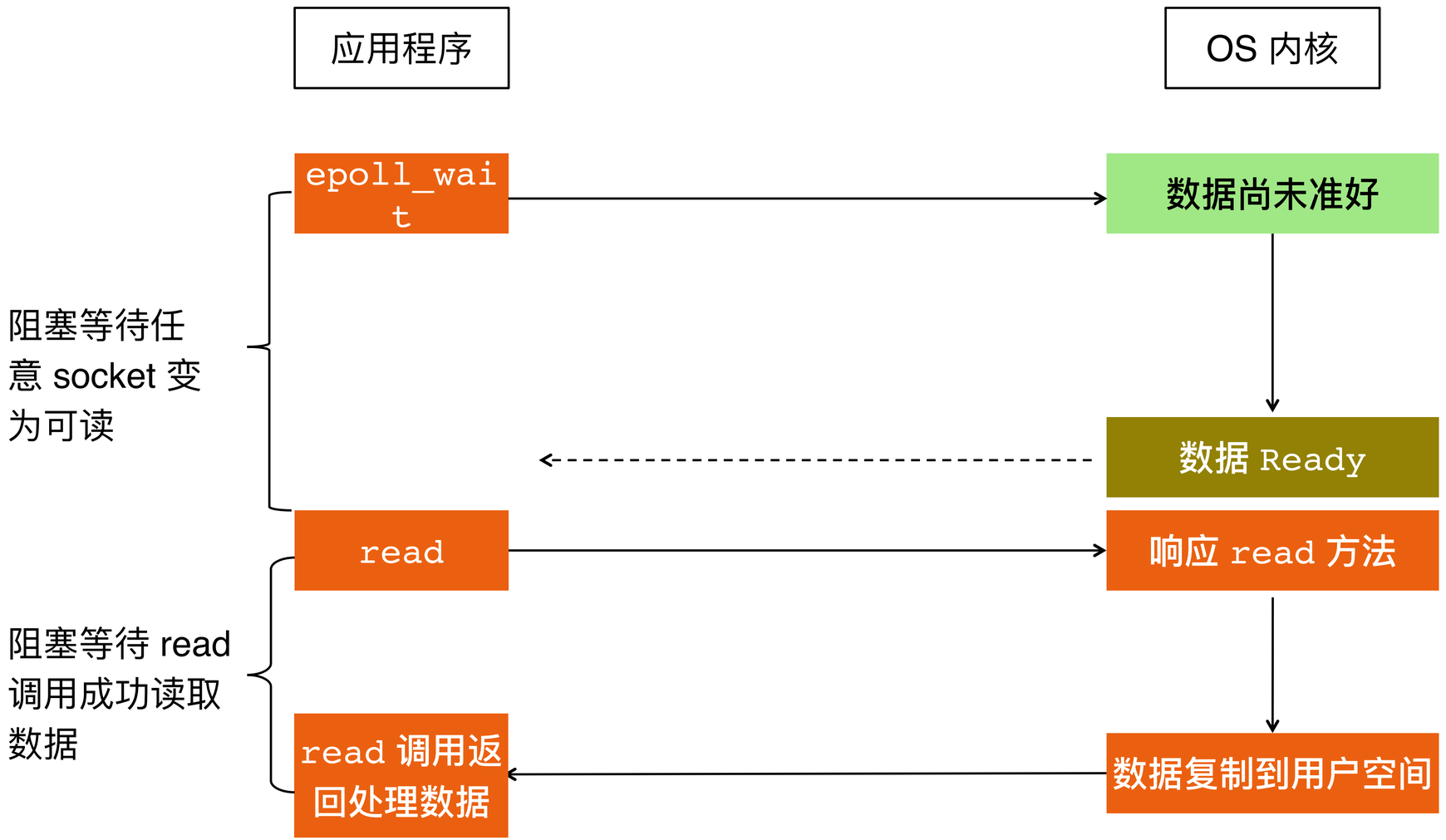
可以这样理解操作系统轮询：应用程序把所有的 socket 都交给操作系统，然后调用 wait 阻塞式等待，只要有一个 socket 可读可写了，那 wait 就会返回，否则就会一直等待。由于操作系统可以通过中断，回调等办法获知哪一个 socket 可读可写，这时，它就会唤醒等待的进程，让 wait 调用返回。这样，应用程序就不会空转从而造成 CPU 时间的浪费。

经过分析，我们需要托管 socket 和等待这两个基本的函数：

- epoll_ctl，用于增加监听的 socket，以及修改 socket 感兴趣的事件。
- epoll_wait，用于应用程序集中等待任一 socket 事件发生。

以前的阻塞式接口现在只保留了 epoll_wait 一个，而不是任何一个 socket 都要阻塞调用。这种多个 I/O 合用一个阻塞式接口的情况就叫做 I/O 多路复用。

epoll 流程


应用程序
OS 内核
epoll_wait
数据尚未准好
阻塞等待任意 socket 变为可读
数据 Ready
read
响应 read 方法
阻塞等待 read 调用成功读取数据
read 调用返回处理数据
数据复制到用户空间

实现 EventLoop

Reactor 模型和 EventLoop

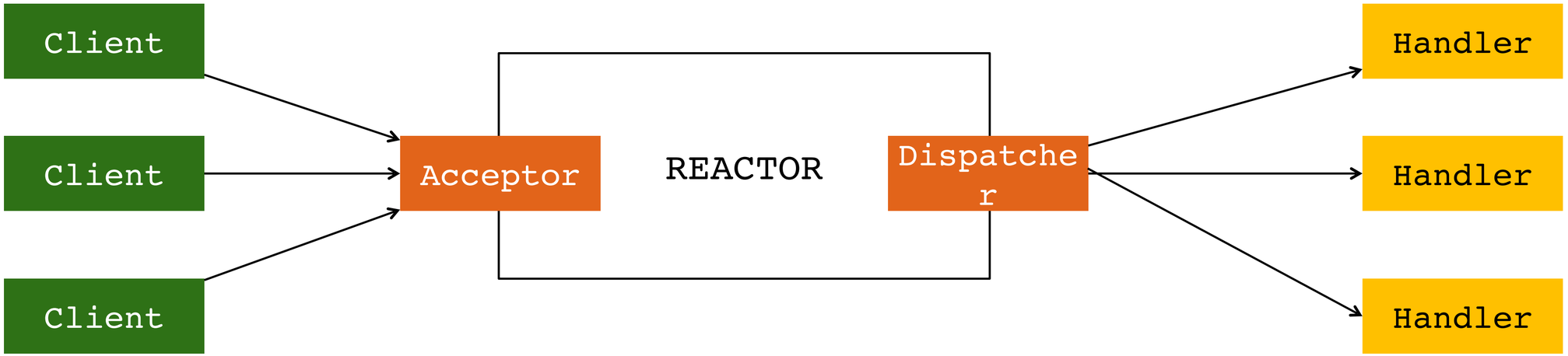
Netty 中的核心数据结构是 EventLoop，Node.js 依赖 libuv 提供异步回调的实现。

EventLoop 的实现，其背后的思想是 Reactor 模型。Reactor 模型一般分为以下几个部分：

1. Client，是请求的发起者。
2. Handler，请求的服务程序。
3. Reactor，响应请求，并将请求分发给不同的服务程序，它又包含以下两个组件：
    1. Acceptor，Reactor 的一个组件，负责接入；
    2. Dispatcher，Reactor 的一个组件，负责分发。

    Handler 的执行是由连接上的事件驱动的，Handler 执行在哪个线程上不是固定的，它的基本形式也是以回调。所以 Reactor 是一种完全的异步模型。

Reactor 模型


Client
Client
Client
Acceptor
REACTOR
Dispatcher
Handler
Handler
Handler

简单类比

假如你在网上下单了一个洗衣机，你在等着洗衣机送货上门。

1. 阻塞式IO：坐在大门口什么也不干，专等着洗衣机送上门你再自己安装。

2. 非阻塞式轮询：打扫一会儿卫生，就跑到门口看看洗衣机送来没有，一直到送到为止。

3. 多线程阻塞式IO：你继续打扫卫生，让弟弟去门口守着。

4. 信号式IO：送到的时候打电话通知你，你再去门口取。

5. IO多路复用：整个小区的快递都送到快递代收点，代收点的人再通知有快递的人去取。

6. 真异步：快递负责上门安装，你打扫完卫生一抬头发现洗衣机装好了，什么人在什么时间做的你不需要关心。

如果按照原始的epoll去写代码就是5，如果再把请求交给线程池，那就相当于在5的基础上又找了一个劳务外包帮你做了安装的工作。

目录

epoll 工作原理

I/O的终极解决方案：协程

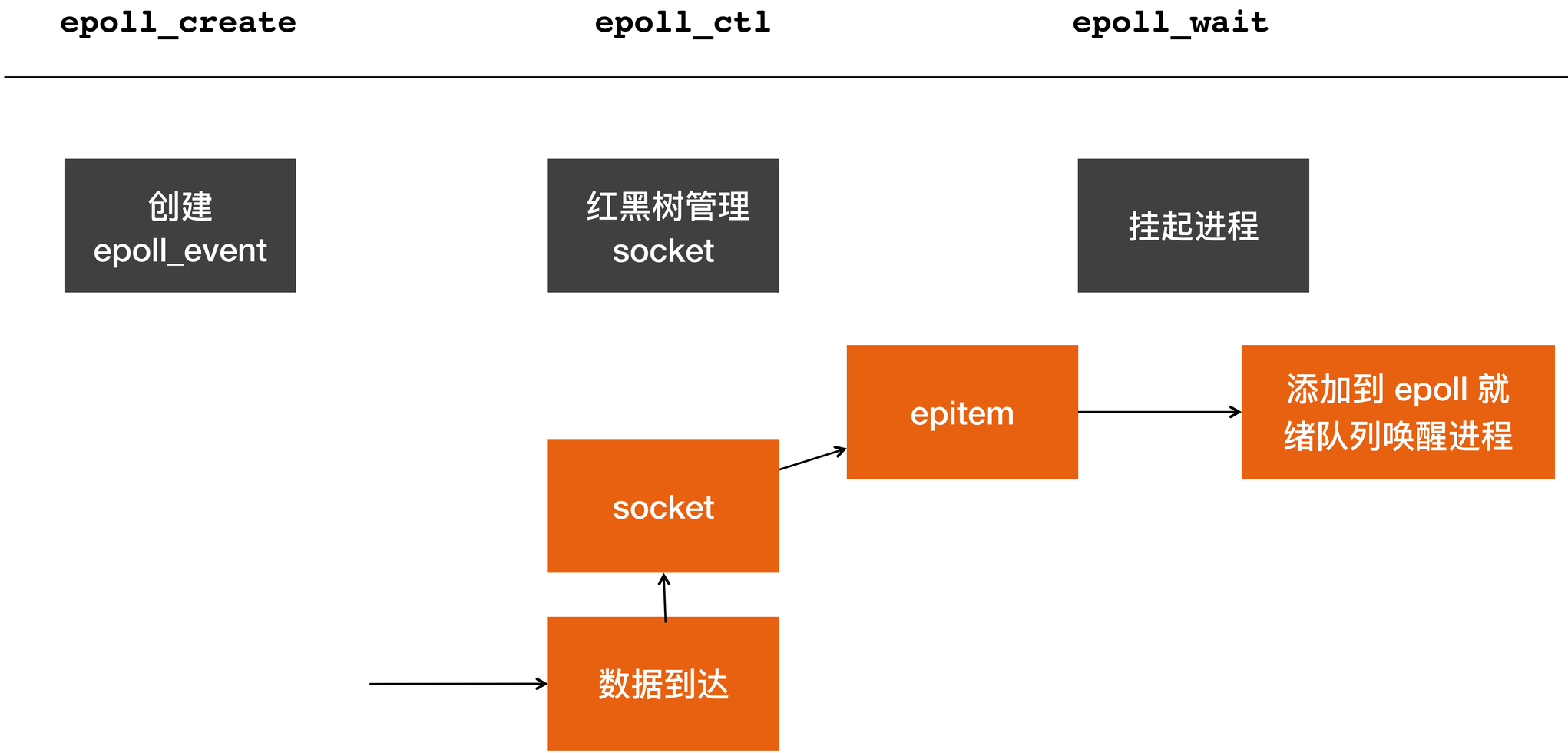
深入理解协程

epoll 工作原理

select 和 epoll对比

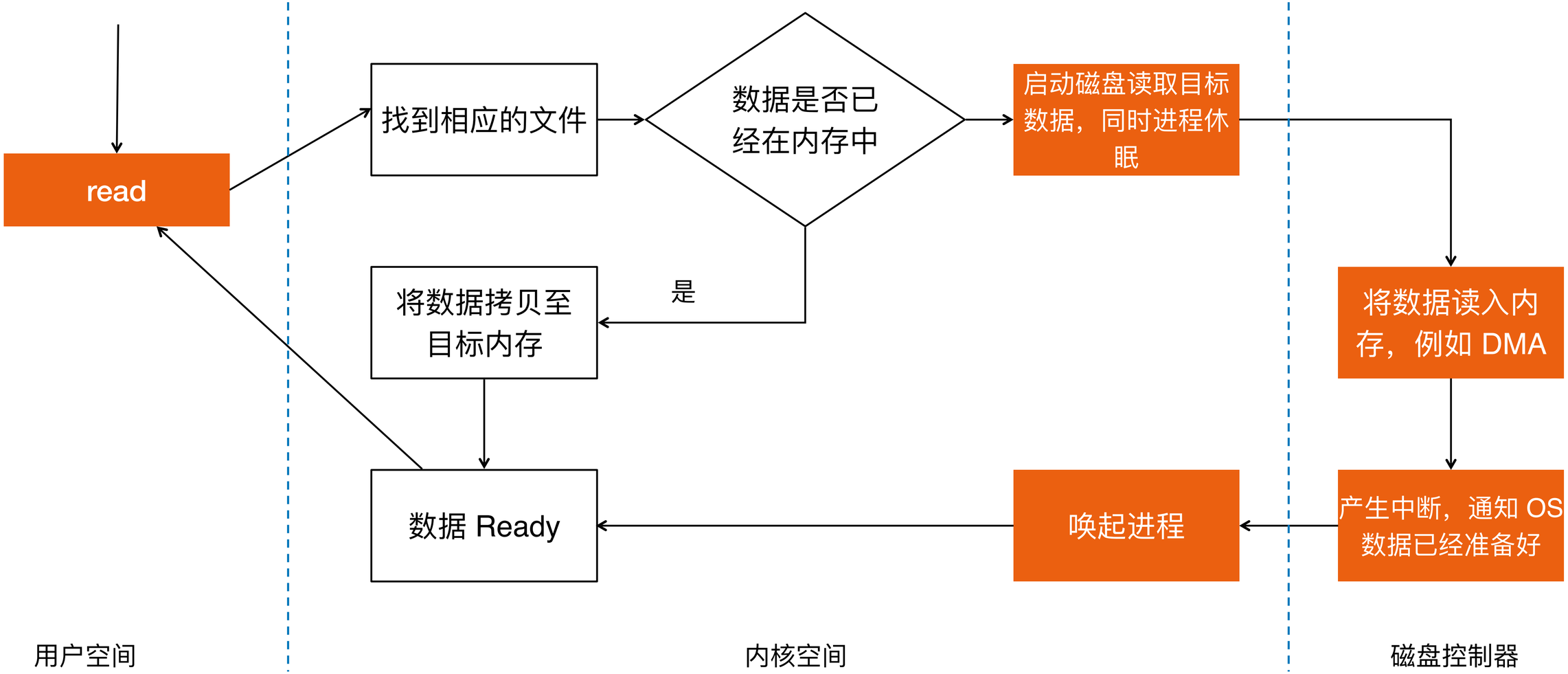
	select	epoll
内核处理机制	轮询	回调
时间复杂度	只要有一个文件描述符的状态发生变化，就需要轮询全部描述符。	只返回状态有变化的描述符集合。
性能	随着连接数的增大，性能下降很快	连接数的增加对 epoll 几乎无影响
连接数	不超过 1024	无限制

epoll的工作原理


epoll_create
epoll_ctl
epoll_wait
创建
epoll_event
红黑树管理
socket
挂起进程
epitem
添加到 epoll 就
绪队列唤醒进程
socket
数据到达

I/O的终极解决方案：协程

复习：同步阻塞式接口


read
找到相应的文件
数据是否已
经在内存中
启动磁盘读取目标
数据，同时进程休
眠
将数据读入内
存，例如 DMA
产生中断，通知 OS
数据已经准备好
唤起进程
数据 Ready
将数据拷贝至
目标内存
是
用户空间
内核空间
磁盘控制器

复习：多线程解决方案

服务端可以开启多个线程，每个线程服务一个客户端。这样阻塞的就是子线程，而主线程不再被阻塞。

服务端的单一线程和客户端的逻辑是同步的：即服务端线程在没有接受到客户端消息之前不会去做其他事情。每人一步，协调一致地前进，所以这种方式称为同步 I/O 。从这个例子中可以看到同步 I/O 往往和阻塞式 I/O 是可以混用的，但要注意阻塞的实体是进程还是线程。

```
while (1) {
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);

    if (!fork()) {
        talk_to_client(clnt_sock);
        exit(0);
    }
}
```

沿着降低执行单元资源消耗方向前进：协程

多线程的情况下阻塞的是线程。

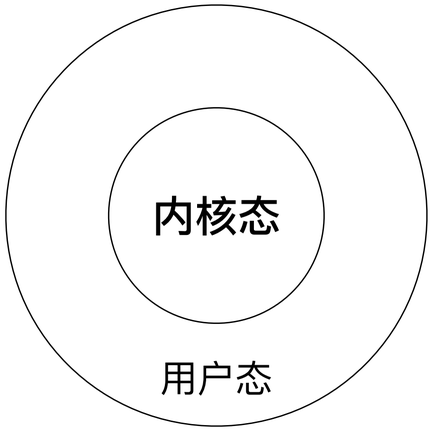
默认情况下，Linux上的线程栈是1M，创建一个线程就要消耗多于1M的内存，而且线程的切换还是要在内核态下进行。所以尽管线程的资源消耗虽然比起进程小了很多，但对于高并发程序而言，还是太大了，切换的时间也比较慢。

而使用协程则给解决高并发场景的资源消耗带来可能。

```
while (1) {
    int clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);

    if (!start_coroutine(talk_to_client, clnt_sock)) {
        printf("Error while create coroutine\n");
        continue;
    }
}
```

一点点操作系统的知识：进程切换

Linux有两种运行模式：用户态和内核态。内核态的运行权限高，用户态的运行权限低。当操作系统要维护各种系统关键状态时就会进行内核态运行。例如进程切换，系统调用，响应硬件中断，管理内存页表等等。而开发者编写的应用程序则在运行在用户态。在用户态，我们无法对操作系统的核心资产进行直接的访问和配置。



深入理解协程

执行单元的切换

不管是进程切换还是线程切换，最重要的就是保存当前执行环境上下文。当操作系统内核试图将进程A切换成进程B时，就要将进程A的上下文（主要是寄存器的值）保存到自己的栈上，然后把栈指针切换成进程B的栈。再从进程B的栈上恢复B的上下文。

这个过程有两个问题，一个切换需要运行在内核态，二时切换的时机不能确定。

进一步，这些问题会导致并发程序的正确性问题。


```
// linux 0.11 借助TSS进行进程切换
#define switch_to(n) {\
    struct {long a,b;} __tmp; \
    __asm__("cmpl %%ecx,current\n\t" \
            "je 1f\n\t" \
            "movw %%dx,%1\n\t" \
            "xchgl %%ecx,current\n\t" \
            "ljmp *%0\n\t" \
            "1:" \
            ::"m" (*&__tmp.a),"m" (*&__tmp.b), \
            "d" (_TSS(n)),"c" ((long) task[n])); \
}
```

协程的切换

在用户态模拟进程的切换，使用内嵌汇编可以做到协程的栈的切换。这样做的好处是：

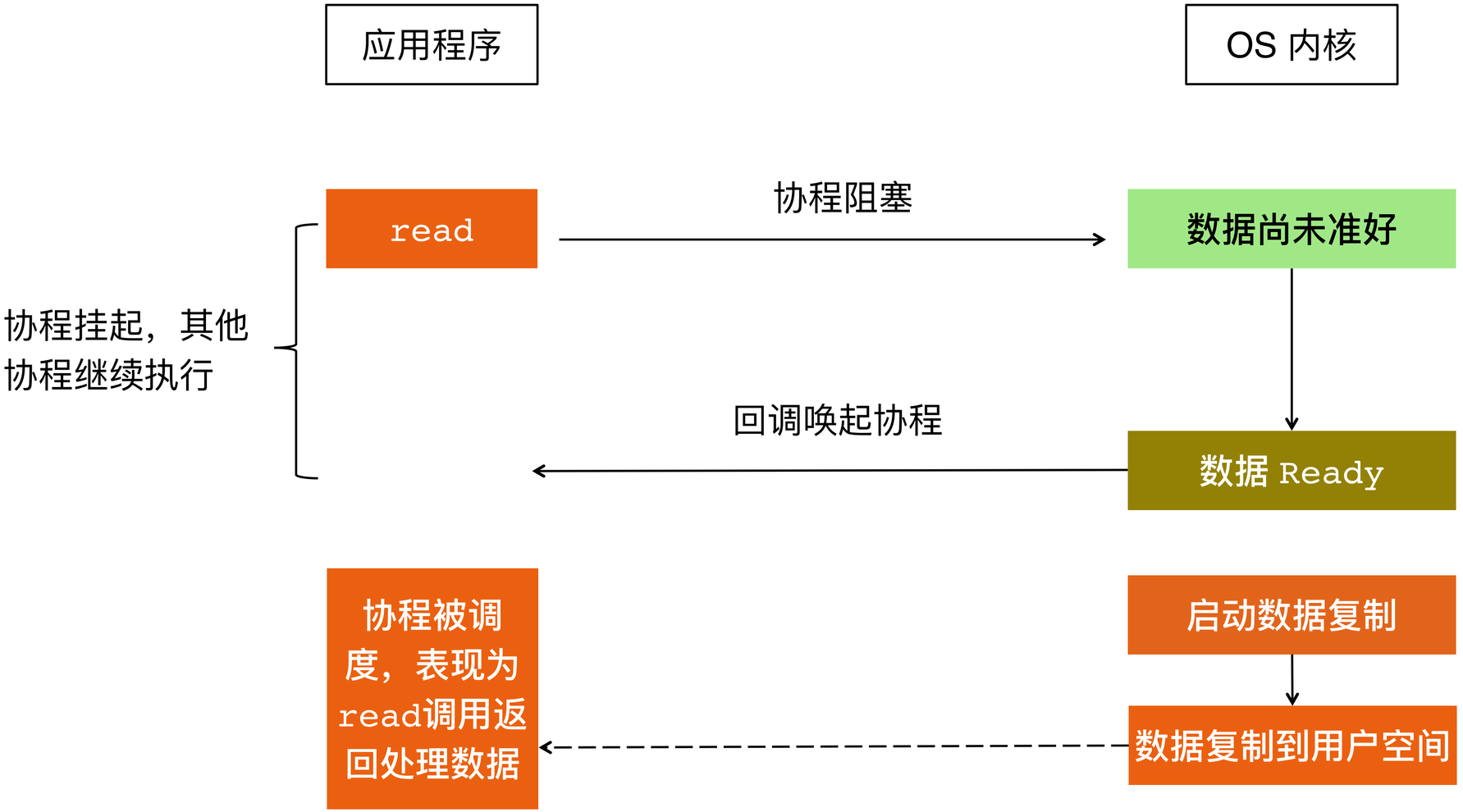
1. 切换的时机可以由用户程序自己掌握，可以更合理地分配CPU时间。

2. 切换不必进入内核态，非常快。

3. 由于切换时机确定，并发程序可以简单地推理证明，甚至可以做到形式化验证其正确性。

使用协程可以模拟进程阻塞的过程，即发出请求后休眠等待，数据返回时回调唤起休眠的执行单元，只是在协程的情况下，休眠的是协程，不是进程。而协程所占用的资源很少，在优秀的实现中几乎可以忽略不计。通过这种方式，我们就可以使用极少的资源实现同步I/O接口。

协程中的同步IO


应用程序
OS 内核
read
协程阻塞
数据尚未准好
协程挂起，其他
协程继续执行
回调唤起协程
数据 Ready
协程被调
度，表现为
read调用返
回处理数据
启动数据复制
数据复制到用户空间

协程的关键技术：栈空间

由前边的分析可知，协程的实现最重要的是如何管理栈空间。主流的协程栈空间的管理方案有以下两种：

1. 独立栈，即每个协程都有自己独立的栈空间，在切换的时候可以通过修改栈指针快速实现切换。但一个协程的栈要开辟多大空间是一个要解决的问题。

    独立栈的空间开辟也有两种主要的解决办法：

    a) 开辟固定空间的栈。优点是实现简单，性能也好，但是不能支持太多协程。

    b) 栈可以按需增长。优点是可以支持大量协程，缺点是需要编译器特殊支持，栈不能保证连续，调试也比较麻烦。

2. 共享栈，多个协程共享同一个线程的栈空间，在切换的时候要先把目标协程的栈上的内容拷贝进来。优点是很容易做到按需分配空间，缺点是切换时会损失一些性能。

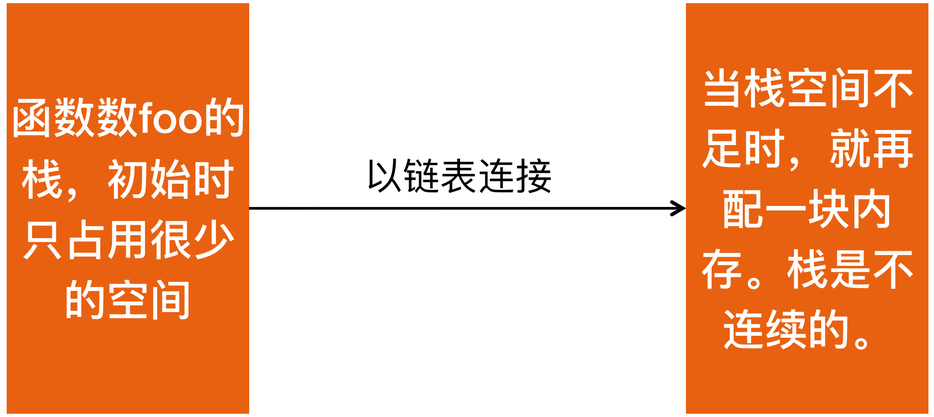
协程的关键技术：栈空间

独立栈，每个协程的栈都是1M

共享栈，只有宿主线程的栈是1M。活跃的协程的栈被拷贝到线程里，不活跃的协程栈会被安置在堆空间。这部分的内存是按需分配的。

动态分配栈空间的两种策略

Go1.2的策略，当栈空间不足时，就再分配一块新的内存。这需要编译器支持。但因为会存在循环中反复申请释放内存的情况，这种方案被废弃



Go1.3以后的策略。类似vector的增长策略。对语言有要求，从栈上出发的引用不能指回栈上，也不能存在指向协程栈的其他引用。

不完整的协程：Generator

```python
# python 3.x

def fib():
    a = 0
    b = 1
    while True:
        yield b
        a, b = b, a + b

o = fib()
for i in range(10):
    print(o.__next__())
```

```javascript
// Javascripit
function* fib() {
    a = 1;
    b = 1;
    while (true) {
        yield a;
        t = a;
        a += b;
        b = t;
    }
}
```