c13n #39

c13n

2025 年 11 月 19 日

第 I 部

深入理解并实现基本的异步 I/O 机制

黄京

Oct 30, 2025

1 导言：为什么我们需要异步 I/O?

想象一个简单的网络服务器场景，它使用同步阻塞模型来处理客户端连接。每个新连接到来时，服务器会创建一个新线程来服务该客户端；线程在执行读或写操作时会被阻塞，直到数据就绪或传输完成。这种模式在面对大量并发连接时，会迅速暴露其局限性，即著名的「C10K 问题」—— 如何让单台服务器同时处理成千上万个客户端连接。同步阻塞模型的痛点在于资源消耗巨大，每个线程都需要分配独立的内存栈空间，并且操作系统频繁的线程上下文切换会消耗大量 CPU 时间；更严重的是，线程在等待 I/O 操作完成时处于空闲状态，导致 CPU 利用率低下。异步 I/O 的承诺正是为了解决这些问题，它允许使用更少的资源（例如单个线程）来管理海量连接，从而实现高吞吐量和低延迟的网络服务。本文的目标不仅是阐述异步 I/O 的核心原理，还将带领读者亲手实现一个基于 Reactor 模式的简单异步服务器，以揭示现代框架如 Node.js 或 Netty 的底层机制。

2 第一部分：基石概念 —— 同步 vs. 异步，阻塞 vs. 非阻塞

在深入异步 I/O 之前，我们必须厘清几个容易混淆的核心概念。同步 I/O 指的是用户线程发起 I/O 请求后，需要主动等待或轮询直到操作完成；例如，调用 read 函数时，线程会一直阻塞，直到数据可用。异步 I/O 则不同，用户线程发起请求后立即返回，操作系统负责处理整个 I/O 过程，并在完成后主动通知用户线程，这类似于委托任务后无需等待结果。阻塞 I/O 意味着在调用结果返回前，当前线程会被挂起，无法执行其他任务；非阻塞 I/O 则不会阻塞线程，即使调用无法立即完成，也会返回一个错误码如 EWOULDBLOCK，允许线程继续处理其他工作。

这些概念可以组合成多种 I/O 模型。同步阻塞 I/O 是最传统的模式，例如在标准 read 调用中线程被阻塞；同步非阻塞 I/O 允许线程通过轮询方式检查 I/O 状态，但会消耗大量 CPU 资源在空转上。I/O 多路复用是一种关键机制，它本质上是同步的，但能非阻塞地管理多个连接；通过系统调用如 select 或 epoll，线程可以同时监视多个文件描述符，并在任一就绪时进行处理。真正的异步 I/O 如 Linux 的 AIO 理论上更高效，但实际应用中往往受限，因此现代高性能系统多依赖于 I/O 多路复用来模拟异步行为。一个重要结论是：我们常说的「异步编程」如 Node.js，其底层通常基于 I/O 多路复用和非阻塞 I/O，通过事件循环和回调在用户态实现异步效果。

3 第二部分：演进之路 —— I/O 多路复用技术

I/O 多路复用技术的发展历程反映了对高性能的不断追求。select 系统调用是最初的解决方案，它允许进程将一组文件描述符传递给内核，内核通过线性扫描检查哪些描述符就绪，然后返回就绪集合。然而，select 有显著缺点：文件描述符数量受限于 FD_SETSIZE（通常为 1024）；每次调用都需要在用户态和内核态之间拷贝整个 fd_set 结构；并且扫描效率随描述符数量增加而线性下降，这在海量连接下成为瓶颈。

poll 系统调用对 select 进行了改进，它使用 pollfd 结构体数组来避免描述符数量限制，但本质上仍需遍历整个数组，在大量空闲连接时性能依然不佳。epoll 是 Linux 提供的高效替代方案，成为现代异步框架的基石。epoll 通过三个核心接口工作：epoll_create 用于

创建 epoll 实例；epoll_ctl 用于注册或修改监控的文件描述符及其事件；epoll_wait 则等待事件发生并返回就绪列表。epoll 的核心优势在于无需重复拷贝描述符集合，内核通过回调机制维护就绪列表，使得 epoll_wait 的时间复杂度接近 $O(1)$；同时，它支持水平触发和边缘触发模式，水平触发会在数据可读时持续通知，而边缘触发仅在状态变化时通知一次，这对编程模型有重要影响。

4　第三部分：动手实现 —— 构建一个简单的 Reactor 模式

为了将理论付诸实践，我们将用 C 语言和 epoll 实现一个单线程的 Reactor 模式 Echo 服务器。Reactor 模式是一种事件驱动架构，其核心组件包括：Handle（如 socket 描述符）、Synchronous Event Demultiplexer（即 epoll_wait）、Initiation Dispatcher（事件循环）和 Event Handler（事件处理接口）。工作流程遵循「等待事件-分发事件-处理事件」的循环，从而高效处理多个连接。

首先，我们创建监听 socket。代码中，调用 socket 函数创建 TCP socket，设置其为非阻塞模式，然后绑定地址并开始监听。这里的关键是将 socket 设置为非阻塞，以避免 accept 调用阻塞整个线程。

```
int listen_sock = socket(AF_INET, SOCK_STREAM, 0);
int flags = fcntl(listen_sock, F_GETFL, 0);
fcntl(listen_sock, F_SETFL, flags | O_NONBLOCK);
struct sockaddr_in addr = {0};
addr.sin_family = AF_INET;
addr.sin_port = htons(8080);
addr.sin_addr.s_addr = INADDR_ANY;
bind(listen_sock, (struct sockaddr*)&addr, sizeof(addr));
listen(listen_sock, SOMAXCONN);
```

这段代码首先创建 socket，然后使用 fcntl 设置非阻塞标志，确保后续操作不会阻塞。接着，绑定到本地地址和端口，并开始监听连接。非阻塞设置是必须的，因为它允许事件循环在等待连接时继续处理其他事件。

接下来，我们创建 epoll 实例并注册监听 socket。调用 epoll_create1 创建 epoll 实例，然后使用 epoll_ctl 将监听 socket 的 EPOLLIN 事件（即可读事件）添加到监控中。

```
int epoll_fd = epoll_create1(0);
struct epoll_event ev;
ev.events = EPOLLIN;
ev.data.fd = listen_sock;
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, listen_sock, &ev);
```

这里，epoll_create1 初始化 epoll 实例，epoll_ctl 用于注册事件。EPOLLIN 表示我们关心 socket 的可读事件，这样当新连接到来时，epoll_wait 会返回通知。

然后，我们进入事件循环。在一个无限循环中，调用 epoll_wait 等待事件发生，并遍历返回的就绪事件列表进行处理。

```
struct epoll_event events[MAX_EVENTS];
```

```
while (1) {
   int n = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
   for (int i = 0; i < n; i++) {
      if (events[i].data.fd == listen_sock) {
         // 处理新连接
      } else {
         // 处理客户端事件
      }
   }
}
```

epoll_wait 会阻塞直到有事件发生，返回就绪事件数量。我们遍历这些事件，如果是监听 socket 就绪，表示有新连接；否则处理客户端 socket 事件。

对于新连接，我们调用 accept 接受连接，设置新 socket 为非阻塞，并注册到 epoll 实例中，使用边缘触发模式。

```
int client_sock = accept(listen_sock, NULL, NULL);
fcntl(client_sock, F_SETFL, O_NONBLOCK);
struct epoll_event client_ev;
client_ev.events = EPOLLIN | EPOLLET;
client_ev.data.fd = client_sock;
epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_sock, &client_ev);
```

accept 返回新客户端 socket，我们立即设置其为非阻塞，并注册 EPOLLIN 事件和 EPOLLET（边缘触发）。边缘触发模式下，epoll 只在 socket 状态变化时通知一次，因此我们必须循环读取直到数据读完。

当客户端 socket 可读时，我们循环读取数据，直到返回 EAGAIN 表示暂时无数据。

```
char buffer[1024];
ssize_t bytes_read;
while ((bytes_read = read(events[i].data.fd, buffer, sizeof(buffer)))
    ↪ > 0) {
   // 处理数据，例如缓存起来
}
if (bytes_read == -1 && errno != EAGAIN) {
   // 处理错误
}
```

在边缘触发模式下，必须循环 read 直到返回 EAGAIN，否则可能丢失数据。读取的数据可以缓存起来，然后修改 epoll 事件为 EPOLLOUT 准备写入。

对于可写事件，我们将缓存的数据写回客户端，同样循环写入直到返回 EAGAIN。

```
ssize_t bytes_written;
while (cached_data_len > 0) {
```

```
    bytes_written = write(events[i].data.fd, cached_data,
        ↪ cached_data_len);
    if (bytes_written < 0) {
        if (errno == EAGAIN) break;
        // 处理错误
    }
    cached_data_len -= bytes_written;
}
```

如果写缓冲区满，write 返回 EAGAIN，我们等待下次可写事件；否则，数据写完后可关闭连接或改回监听读事件。整个实现中，非阻塞 I/O 和状态管理至关重要，每个连接需要维护自己的缓冲区和状态机，以避免阻塞事件循环。

5 第四部分：从底层到上层 —— 现代异步编程的演进

尽管我们实现的 Reactor 模式高效，但直接使用回调会导致代码嵌套深、难以维护，这就是所谓的「回调地狱」。例如，如果每个 I/O 操作都需嵌套回调，代码会变得复杂且易错。为了解决这个问题，现代异步编程引入了 Promise/Future 和 Async/Await 等抽象。Promise 代表一个未来可能完成的操作，它允许链式调用而非嵌套回调；Async/Await 语法则让异步代码看起来像同步代码，提高可读性。这些抽象的底层仍然基于事件循环和状态机，本质上是对 Reactor 模式的高级封装。

协程是另一种演进，它作为用户态线程，可以在单个线程内实现多任务切换。例如，Go 语言的 goroutine 利用异步 I/O 的事件循环来调度大量协程，每个协程在等待 I/O 时主动让出 CPU，从而高效处理高并发。协程与异步 I/O 结合，进一步简化了编程模型，同时保持了高性能。这些演进表明，异步 I/O 的核心思想是将 I/O 等待任务卸载到操作系统内核，最大化线程利用率。

通过本文的探讨，我们从同步阻塞的问题出发，回顾了 I/O 多路复用技术的演进，并亲手实现了一个基于 epoll 的 Reactor 模式服务器。异步 I/O 的本质在于将等待 I/O 的任务从应用程序线程卸载到操作系统内核，从而提升资源利用率和系统吞吐量。理解这些底层机制有助于在不同场景下做出技术选型，例如在 I/O 密集型应用中选择 Node.js 或 Go，而在需要更细粒度控制时使用原生 epoll。鼓励读者进一步阅读相关源码如 libevent 或 libuv，以深化对异步编程的理解，并在实践中不断探索。

第 II 部

深入理解并实现基本的信号量（Semaphore）机制

李睿远

Oct 31, 2025

从并发问题出发，到原理剖析，最后用代码亲手实现一个信号量。

想象一个现实世界中的场景：一个停车场只有十个车位，却有十五辆车需要停放。如果没有有效的管理机制，车辆可能会超停、争抢车位，导致混乱和冲突。在计算机科学中，这种场景对应于多线程或进程争抢有限的共享资源，例如数据库连接池、线程池或打印队列。当多个线程同时访问和修改共享数据时，会出现竞争条件，导致结果不可预测。这种问题的核心在于并发编程的挑战，包括竞争条件、临界区问题以及线程间的协调与同步。临界区是指需要互斥访问的代码段，而线程间不仅需要互斥，有时还需要协作，例如在生产者－消费者模型中。

为了解决这些问题，荷兰计算机科学家 Dijkstra 提出了一种经典的同步工具 —— 信号量。简单来说，信号量是一个计数器，用于管理对多个进程或线程共享的资源池的访问。它通过简单的操作来实现线程间的同步，避免资源冲突和数据不一致。

6　信号量核心原理解析

信号量的核心数据结构包括一个整数值和一个等待队列。整数值通常初始化为非负数，代表可用资源的数量；等待队列用于存放因资源不足而阻塞的线程。信号量的基本操作是 P 操作和 V 操作，它们分别对应于等待和发送信号。P 操作源自荷兰语 Proberen，意为尝试；V 操作源自 Verhogen，意为增加。在英语中，它们常被称为 Wait 和 Signal，或 Down 和 Up。

P 操作用于申请资源。其伪代码如下所示：

```
P(Semaphore S) {
   S.value--;
   if (S.value < 0) {
      将当前线程加入 S 的等待队列 ;
      阻塞当前线程 ;
   }
}
```

这段代码首先减少信号量的值，如果值小于零，表示资源已耗尽，当前线程会被加入等待队列并阻塞。形象地说，这就像进入停车场前查看剩余车位：如果车位充足，直接进入；否则，必须排队等待。

V 操作用于释放资源。其伪代码如下：

```
V(Semaphore S) {
   S.value++;
   if (S.value <= 0) {
      从 S 的等待队列中移除一个线程 T;
      唤醒线程 T;
   }
}
```

这里，信号量的值增加，如果值小于或等于零，说明有线程在等待，于是从队列中唤醒一个线程。这类似于离开停车场时归还车位，并通知等待的车辆进入。

信号量分为两种类型：计数信号量和二进制信号量。计数信号量的值可以大于一，用于控制对多个实例资源的访问，如连接池。二进制信号量的值只能为零或一，主要用于实现互斥，保护临界区。值得注意的是，互斥锁是二进制信号量的一种特例，它强调所有权概念，即加锁和解锁通常由同一线程执行。

7 经典案例：用信号量解决生产者－消费者问题

生产者－消费者问题是一个经典的同步问题，其中生产者线程生产数据并放入缓冲区，消费者线程从缓冲区取出数据。需要确保缓冲区满时生产者等待，缓冲区空时消费者等待，同时保证对缓冲区的操作是互斥的。

解决方案使用三个信号量：empty 代表空槽位数量，初始值为缓冲区大小；full 代表已占用槽位数量，初始值为零；mutex 是二进制信号量，用于互斥访问缓冲区，初始值为一。

生产者线程的伪代码如下：

```
while (true) {
   生产一个数据项 ;
   P(empty);
   P(mutex);
   将数据放入缓冲区 ;
   V(mutex);
   V(full);
}
```

首先，生产者生产数据后，通过 P(empty) 申请空位；如果无空位，则阻塞。然后，通过 P(mutex) 进入临界区，确保只有一线程操作缓冲区。数据放入后，释放互斥锁并通过 V(full) 通知消费者。

消费者线程的伪代码如下：

```
while (true) {
   P(full);
   P(mutex);
   从缓冲区取出一个数据项 ;
   V(mutex);
   V(empty);
   消费数据项 ;
}
```

消费者通过 P(full) 申请数据项；如果无数据，则阻塞。然后获取互斥锁，取出数据后释放锁，并通过 V(empty) 通知生产者。关键点在于，P(empty) 和 P(mutex) 的顺序不能颠倒，否则可能造成死锁。例如，如果生产者先获取互斥锁再申请空位，而缓冲区已满，它可能阻塞并持有锁，导致消费者无法释放资源。

8 动手实现：用代码构建我们自己的信号量

为了深入理解信号量，我们可以用代码实现一个简单的 Semaphore 类。这里以 Python 风格伪代码为例，设计一个包含计数器和等待队列的类。

首先，定义 Semaphore 类的成员变量：count 表示信号量的计数值，waitingQueue 是一个队列用于存放等待线程，lock 是一个锁用于保护对 count 和 waitingQueue 的修改，确保 P 和 V 操作的原子性。

实现 P 操作（wait 方法）的代码如下：

```
def wait(self):
   with self.lock:
      self.count -= 1
      if self.count < 0:
         current_thread = get_current_thread()
         self.waitingQueue.enqueue(current_thread)
         sleep(current_thread)
```

这段代码使用 with 语句获取锁，确保操作原子性。首先减少 count 值，如果值小于零，将当前线程加入等待队列并使其睡眠。睡眠操作会释放锁，允许其他线程执行。

实现 V 操作（signal 方法）的代码如下：

```
def signal(self):
   with self.lock:
      self.count += 1
      if self.count <= 0:
         thread_to_wakeup = self.waitingQueue.dequeue()
         wakeup(thread_to_wakeup)
```

这里，同样先获取锁，增加 count 值。如果值小于或等于零，说明有线程在等待，于是从队列中取出一个线程并唤醒。被唤醒的线程会重新尝试获取锁，并从睡眠点继续执行。

这种实现采用了阻塞等待，而非忙等待，从而避免 CPU 资源浪费。原子性保证是关键，因为 P 和 V 操作本身必须是不可分割的，否则可能导致竞争条件。

9 现代编程语言中的信号量

在实际开发中，我们通常使用现代编程语言提供的成熟信号量实现，而不是自行构建。例如，在 Java 中，可以使用 java.util.concurrent.Semaphore 类。初始化时指定许可数量，如 Semaphore sem = new Semaphore(5); 然后通过 sem.acquire() 执行 P 操作，sem.release() 执行 V 操作。这些方法经过优化和测试，能有效处理高并发场景。

在 Python 中，threading 模块提供了 Semaphore 类。创建实例如 sem = threading.Semaphore(5)，然后使用 sem.acquire() 和 sem.release() 进行操作。C++ 从 C++20 标准开始，在 <semaphore> 头文件中提供了信号量支持。使用这些库可以避免常见错误，提高代码可靠性和性能。

建议开发者在项目中优先使用语言内置的信号量实现，因为它们集成了底层系统优化，并能处理边缘情况，如超时和中断。

信号量是一种强大的同步原语，通过计数器和等待队列管理共享资源访问。其核心操作 P 和 V 实现了线程间的协调，计数信号量适用于资源池管理，而二进制信号量常用于互斥。与互斥锁相比，信号量更灵活，P 和 V 操作可由不同线程执行，但也更容易出错，例如如果 V 操作多于 P 操作，可能导致信号量值异常。

进阶挑战包括读者－写者问题和哲学家就餐问题，这些经典问题需要更复杂的信号量应用来避免死锁和确保公平性。例如，在读者－写者问题中，需要平衡读写线程的访问权限，防止写者饥饿。信号量使用不当可能引入风险，因此在实际应用中需仔细设计测试。

10 互动与参考资料

读者可以尝试用自实现的 Semaphore 类重写生产者－消费者问题，以加深理解。另外，考虑如何扩展实现，添加超时等待功能，例如 wait(timeout) 方法，这能提高程序的响应性。参考资料包括 Dijkstra 的原始论文、经典教材如《现代操作系统》和《操作系统概念》，以及 Java 和 Python 的官方文档，这些资源提供了更深入的理论和实践指导。

第 III 部

深入理解并实现基本的哈希表数据结构

黄京

Nov 01, 2025

在日常生活中，我们经常需要快速查找信息，例如在字典中查询单词、在电话簿中寻找号码，或者从缓存中获取数据。这些场景都要求一种能实现快速查找、插入和删除操作的数据结构。如果使用数组，虽然插入操作快速，但查找需要遍历整个数组，时间复杂度为 O(n)；链表同样存在查找效率低下的问题。因此，我们迫切需要一种能接近 O(1) 时间复杂度的数据结构。哈希表就是这样一种“魔法”数据结构，它通过巧妙的设计，将平均时间复杂度降至 O(1)。本文的目标是彻底解析哈希表的工作原理，掌握处理哈希冲突的核心方法，从头开始用代码实现一个功能完整的哈希表，并对其性能进行分析和优化。

11 哈希表的核心思想 —— 为何它能如此之快?

哈希表的快速性能源于数组的随机访问能力。数组通过下标索引，可以在 O(1) 时间内访问任何元素，这为哈希表的高效性奠定了基石。然而，问题在于我们如何将任意类型的键，例如字符串或对象，转换为一个数组下标？这就需要引入哈希函数的概念。

哈希函数是一座从“键”到“地址”的魔法桥梁。它接受一个键作为输入，返回一个整数哈希值，然后将这个整数映射到数组的固定范围内，即 $index = \text{hash}(key) \text{ mod array\_size}$。一个理想的哈希函数应具有确定性，即相同的键必须始终产生相同的哈希值；高效性，计算速度要快；以及均匀性，哈希值应尽可能均匀分布，以减少冲突。例如，假设我们有一个存储员工信息的哈希表，键是员工 ID。如果哈希函数计算 hash(101) 返回 1，hash(102) 返回 2，那么我们可以直接将数据存入数组的对应位置，实现近乎即时的访问。

12 无法避免的挑战 —— 哈希冲突

哈希冲突是指两个不同的键经过哈希函数计算后，得到了相同的数组索引。例如，hash(John) 和 hash(Jane) 都计算出索引 5。由于键空间远大于数组空间，冲突是必然发生的。因此，一个好的哈希表设计不在于避免冲突，而在于高效地解决冲突。解决哈希冲突有两种主流方法：链地址法和开放地址法。

链地址法的思想是让数组的每个位置都指向一个链表，所有哈希到同一索引的键值对都存储在这个链表中。操作时，插入需要计算索引并将键值对添加到对应链表的末尾；查找需要遍历链表比对键是否相等；删除则需要找到并移除对应节点。链地址法的优点是简单有效，链表可以无限扩展，适合不知道数据量的情况。开放地址法则是在发生冲突时，按照某种探测序列在数组中寻找下一个空闲的位置。常见的探测方法包括线性探测，即 $index = (\text{original\_index} + i) \text{ mod size}$，其中 i 从 1 开始递增；二次探测，即 $index = (\text{original\_index} + i^2) \text{ mod size}$；以及双重哈希，使用第二个哈希函数来计算步长。开放地址法的优点是所有数据都存储在数组中，缓存友好，但删除操作复杂，需要标记删除，且容易产生聚集现象。

13 动手实现 —— 构建我们自己的哈希表（采用链地址法）

我们将使用链地址法来实现一个基本的哈希表。首先，设计数据结构。定义键值对节点类 HashNode，它包含 key、value 和 next 指针，用于构建链表结构。然后定义哈希表类 MyHashMap，核心字段包括一个存储链表的数组 table、当前键值对数量 size、数组容量 capacity 和负载因子 loadFactor。负载因子定义为 $loadFactor = size/capacity$，

用于在后续操作中触发动态扩容。

接下来实现核心方法。构造函数初始化一个固定容量的数组，例如 16，并设置默认负载因子为 0.75。哈希函数 _hash 是一个私有方法，对于整数键，直接取模，例如 return key % capacity；对于字符串键，使用多项式哈希码并取模以确保均匀性，例如通过遍历字符串字符计算哈希值：$hash = s[0] \times 31^{(n-1)} + s[1] \times 31^{(n-2)} + \cdots + s[n-1]$，其中 n 是字符串长度，然后对 capacity 取模。

put 方法用于插入键值对。首先计算索引 $index = \_hash(key)$，然后遍历 table[index] 对应的链表。如果找到相同的 key，则更新其 value；否则，在链表头部插入新节点，以保持 O(1) 的插入时间。插入后增加 size，并检查负载因子是否超过阈值（例如 0.75），如果超过，则调用 _resize 方法进行扩容。这段代码的关键在于处理链表遍历和节点插入，确保在冲突时正确维护数据结构。

get 方法用于查找键对应的值。计算索引后，遍历链表，查找并返回对应 key 的 value，若未找到则返回 null。这体现了链地址法的查找逻辑，依赖于链表的线性搜索，但在平均情况下，链表长度短，性能接近 O(1)。

remove 方法用于删除键值对。计算索引，遍历链表找到节点并删除，同时减少 size。删除操作需要小心处理链表指针，例如如果删除的是头节点，需要更新数组引用；否则，调整前驱节点的 next 指针。

动态扩容是保证性能的关键。当元素过多时，链表会变长，性能从 O(1) 退化为 O(n)。_resize 方法创建一个新数组，容量通常是原容量的两倍，然后遍历旧表中的每一个节点，根据新的容量重新哈希所有键，并将它们放入新数组的正确位置。最后将 table 引用指向新数组。尽管扩容是一个耗时操作，时间复杂度为 O(n)，但通过摊还分析，其平均成本依然是 O(1)，这确保了哈希表在长期运行中的高效性。

14　分析与优化

哈希表的时间复杂度在最佳情况下，当哈希函数均匀且无冲突时，所有操作均为 O(1)。平均情况下，在合理负载因子下，通过链表平均长度分析，操作依然是 O(1)。最坏情况下，所有键都冲突，退化为一个链表，操作 O(n)。因此，设计一个好的哈希函数至关重要，例如 Java 中 String.hashCode() 的实现使用多项式哈希码，充分利用键的所有信息，让结果的每一位都影响最终哈希值，以确保均匀分布。

进阶优化思路包括将链表转换为红黑树，当链表过长时，将查找性能从 O(n) 提升至 O(log n)，如 Java 8+ 的 HashMap 所做的那样。此外，选择优质的初始容量和负载因子也能优化性能，例如根据预期数据量设置初始容量，避免频繁扩容。

本文全面回顾了哈希表的核心思想：通过哈希函数和数组实现快速访问，使用链地址法或开放地址法解决冲突，并通过动态扩容维持性能。哈希表广泛应用于数据库索引、缓存如 Redis、集合以及对象表示等场景。鼓励读者动手实现一遍，并尝试不同的哈希函数或冲突解决策略，以加深理解。下期可能介绍更高级的数据结构，如 ConcurrentHashMap，或深入探讨红黑树的原理与应用。

附录：完整代码实现可参考 GitHub 仓库（例如，使用 Java 编写），其中包含了上述所有方法的详细实现和测试用例。

第 IV 部

深入理解并实现基本的信号量（Semaphore）机制

杨岢瑞

Nov 02, 2025

信号量在并发编程和操作系统中扮演着至关重要的角色，用于解决多线程或进程环境下的资源竞争和数据不一致问题。本文旨在帮助读者从理论到实践全面掌握信号量机制，包括核心概念、类型、操作、应用场景以及代码实现细节，并通过示例和解读加深理解。
在并发编程中，多个线程或进程同时访问共享资源时，常会导致资源竞争和数据不一致等风险。例如，多个线程同时修改一个共享变量，可能产生不可预测的结果，甚至引发程序崩溃。信号量作为一种同步机制，由 Edsger Dijkstra 在 1960 年代提出，是解决这些问题的基石工具。本文将逐步解析信号量的原理、类型、操作、应用场景，并展示如何实现基本的信号量机制，最后讨论常见问题和最佳实践，以帮助读者构建系统的知识框架。

15　什么是信号量?

信号量是一种用于控制多线程或进程访问共享资源的同步机制，其核心思想是通过一个计数器来管理资源访问权限。当线程需要访问资源时，它执行「P 操作」来申请资源；如果计数器大于零，则减一并继续执行；否则，线程进入阻塞等待状态。当线程释放资源时，它执行「V 操作」，将计数器加一并唤醒等待的线程。这种机制确保了资源访问的有序性和安全性。
信号量的工作原理可以通过一个比喻来直观理解：想象一个停车场，信号量就像车位计数器。当有车辆进入时，计数器减一；当车位已满时，新来的车辆必须等待。当有车辆离开时，计数器加一，允许等待的车辆进入。这种类比帮助读者理解信号量如何通过计数来协调资源分配。
与互斥锁和条件变量相比，信号量更具灵活性。互斥锁通常用于实现互斥，确保同一时间只有一个线程访问资源，而信号量可以用于计数场景，控制多个资源的访问。条件变量则用于在特定条件下等待和通知，但信号量通过计数器直接管理资源可用性，适用于更广泛的同步需求。

16　信号量的类型与操作

信号量主要有两种类型：二进制信号量和计数信号量。二进制信号量的值仅为 0 或 1，常用于实现互斥锁，保护临界区，确保同一时间只有一个线程访问资源。例如，在访问共享变量时，使用二进制信号量可以防止并发修改导致的数据错误。计数信号量的值为非负整数，用于控制多个资源的访问，例如在连接池或缓冲区管理中，限制同时使用的连接数或缓冲槽数。
信号量的核心操作包括「P 操作」和「V 操作」。P 操作，也称为等待或向下操作，用于申请资源。其伪代码如下：

```
P(semaphore S) {
    while (S <= 0) ; // 忙等待或阻塞
    S = S - 1;
}
```

在实际实现中，为了避免忙等待带来的性能损耗，通常使用阻塞机制。P 操作首先检查信号量值，如果大于零则减一，否则线程进入等待状态，直到被唤醒。V 操作，也称为信号或向上操作，用于释放资源。其伪代码如下：

```
V(semaphore S) {
```

```
    S = S + 1;
    // 唤醒一个等待的线程
}
```

V 操作将信号量值加一，并唤醒一个等待的线程（如果有）。这些操作必须是原子的，以防止竞态条件。原子性意味着在操作执行期间，不会被其他线程中断，底层实现通常依赖硬件指令（如测试并设置）或操作系统提供的同步原语。

17 信号量的应用场景

信号量常用于解决经典同步问题，例如生产者-消费者问题和读者-写者问题。在生产者-消费者问题中，生产者线程生产数据并放入共享缓冲区，消费者线程从缓冲区取出数据消费。需要使用信号量来同步访问，防止缓冲区溢出或下溢。通常，使用两个信号量：一个表示空槽数量，另一个表示满槽数量。生产者执行 P 操作在空槽信号量上，如果空槽不足则等待；消费者执行 P 操作在满槽信号量上，如果满槽不足则等待。生产者和消费者分别执行 V 操作在对方信号量上，以通知状态变化。

在读者-写者问题中，多个读者线程可以同时读取共享资源，但写者线程需要独占访问。可以使用信号量来实现读者优先或写者优先的策略。例如，使用一个信号量来控制写者访问，另一个信号量来保护读者计数，确保写者不会在读者活跃时修改资源。

在实际应用中，信号量用于操作系统中的资源管理，如限制文件句柄或网络连接的数量。在多线程编程中，信号量可以用于任务调度，例如在线程池中限制并发线程数，防止资源耗尽。这些场景展示了信号量在现实系统中的广泛适用性。

18 实现基本的信号量机制

在实现信号量之前，需要考虑编程语言和依赖工具。本文以 C 语言和 Java 为例，因为它们广泛用于并发编程。在 C 语言中，可以使用 pthread 库的互斥锁和条件变量来实现信号量。以下是一个基于互斥锁和条件变量的信号量实现示例：

```
#include <pthread.h>

typedef struct {
    int value;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} semaphore_t;

void sem_init(semaphore_t *sem, int value) {
    sem->value = value;
    pthread_mutex_init(&sem->mutex, NULL);
    pthread_cond_init(&sem->cond, NULL);
}

```

```
void P(semaphore_t *sem) {
   pthread_mutex_lock(&sem->mutex);
   while (sem->value <= 0) {
      pthread_cond_wait(&sem->cond, &sem->mutex);
   }
   sem->value--;
   pthread_mutex_unlock(&sem->mutex);
}

void V(semaphore_t *sem) {
   pthread_mutex_lock(&sem->mutex);
   sem->value++;
   pthread_cond_signal(&sem->cond);
   pthread_mutex_unlock(&sem->mutex);
}
```

在这个实现中，信号量结构包含一个整数值、一个互斥锁和一个条件变量。初始化函数 sem_init 设置初始值并初始化互斥锁和条件变量。P 操作首先获取互斥锁，然后检查信号量值；如果值小于等于零，线程等待在条件变量上；否则，值减一并释放锁。V 操作获取互斥锁，值加一，然后通知条件变量唤醒一个等待线程。这种实现确保了操作的原子性和线程安全性。

在 Java 中，可以使用 synchronized 关键字或 ReentrantLock 来实现信号量。以下是一个使用 synchronized 的简单实现：

```
public class Semaphore {
   private int value;

   public Semaphore(int value) {
      this.value = value;
   }

   public synchronized void P() throws InterruptedException {
      while (value <= 0) {
         wait();
      }
      value--;
   }

   public synchronized void V() {
      value++;
      notify();
   }
}
```

这个实现使用对象的内置锁和 wait/notify 机制。P 方法在值小于等于零时等待，V 方法增加值并通知一个等待线程。这种实现简洁易用，但需要注意线程中断处理，例如 `InterruptedException`。

测试与验证时，可以创建一个简单测试用例，例如多个线程访问共享计数器。使用信号量来确保线程安全，避免数据竞争。调试时，注意死锁和资源泄漏问题，例如通过日志输出或调试工具监控线程状态。

19 常见问题与最佳实践

在使用信号量时，常见陷阱包括死锁和竞态条件。死锁发生在多个线程循环等待资源时，例如线程 A 持有信号量 S1 并等待 S2，线程 B 持有 S2 并等待 S1。避免死锁的方法包括按固定顺序申请信号量或使用超时机制，例如在 P 操作中设置等待时间限制。竞态条件由于操作非原子性导致，在信号量实现中，必须确保 P 和 V 操作是原子的，否则多个线程可能同时修改值，导致不一致。使用互斥锁或原子指令可以解决这个问题。

在高并发场景下，信号量可能带来性能开销，因为线程可能频繁阻塞和唤醒。根据具体场景，可以选择轻量级同步机制，如自旋锁，但自旋锁在等待时消耗 CPU，适用于短时间等待的情况。最佳实践包括初始化信号量时设置合理的初始值，避免过度使用信号量。在可能的情况下，优先使用更高级的抽象，如阻塞队列，它们内部可能使用信号量，但提供更简单的接口，减少出错概率。

信号量是强大的同步工具，适用于资源计数和互斥场景。理解 P 和 V 操作以及信号量类型的选择是关键。通过实现经典同步问题，如生产者-消费者，可以加深对信号量机制的理解。进一步学习方向包括阅读操作系统教材，如《现代操作系统》，或 Java 并发编程资源。鼓励读者实践更复杂的问题，如哲学家就餐问题，以巩固知识。实践是掌握信号量的最佳方式，通过编码实现可以更好地应对实际开发中的挑战。

20 参考资料

参考资料包括《操作系统概念》、《Java 并发编程实战》等书籍，以及在线文档如 Linux man 页面和 Java API 文档。这些资源提供了更深入的理论背景和实践指导，帮助读者扩展知识。

第 V 部

深入理解并实现基本的基数排序（Radix Sort）算法

杨其臻

Nov 03, 2025

在计算机科学中，排序算法是基础且关键的主题。我们熟知的基于比较的排序算法，如快速排序和归并排序，其时间复杂度下界为 $O(n \log n)$，这意味着它们无法在比较模型下突破这一限制。然而，是否存在一种算法能够超越这一界限，在某些场景下实现更高效的排序呢？答案是肯定的 —— 基数排序作为一种非比较型整数排序算法，能够在处理特定数据时达到 $O(k \times n)$ 的时间复杂度，其中 k 表示数字的最大位数。本文旨在深入解析基数排序的原理、实现方式及其性能，帮助读者从理论到实践全面掌握这一算法。

21 理解基数排序的核心思想

基数排序的核心在于“基数”这一概念。基数指的是进制的基数，例如十进制数的基数为 10，二进制数的基数为 2。算法通过按位排序来实现整体有序，排序顺序可以从最低有效位到最高有效位（LSD 方式），或反之（MSD 方式）。这种方法的巧妙之处在于，它避免了直接比较元素大小，而是通过分配和收集操作来逐步排序。

为了直观理解基数排序，让我们以一个简单的十进制数字数组为例，如 `[170, 45, 75, 90, 802, 24, 2, 66]`。排序过程从最低位（个位）开始，将数字分配到对应的桶中（0 到 9），然后按桶顺序收集回数组。接着处理十位数，重复分配和收集操作，最后处理百位数。每一轮排序都必须保持稳定性，即相等元素的相对顺序在排序后不变。稳定性是基数排序正确性的关键，因为如果某一轮排序不稳定，之前位数的排序信息可能会丢失，导致最终结果错误。

22 基数排序的两种实现方式

基数排序主要有两种实现方式：最低位优先（LSD）和最高位优先（MSD）。LSD 方式从数字的最低位开始排序，逐步向最高位推进。这种方式直观且易于实现，因为每一轮排序都基于前一轮的结果，通过稳定排序确保高位权重更大时能够覆盖低位的顺序。LSD 的正确性依赖于稳定排序的累积效应，最终实现整体有序。

相比之下，MSD 方式从最高位开始排序，并递归地对每个桶中的数字处理次高位。MSD 更类似于分治策略，可能在早期就将数据分割成小块，从而减少后续处理次数。然而，MSD 的实现较为复杂，需要处理递归和边界情况，通常更适用于字符串字典序排序等场景。尽管 LSD 和 MSD 都基于位的分配和收集，但它们在处理顺序和效率上各有侧重，读者可以根据具体需求选择适合的方式。

23 手把手实现 LSD 基数排序

LSD 基数排序的实现可以分为几个清晰步骤。首先，需要找到数组中的最大数字，以确定排序的轮数（即最大位数）。其次，初始化十个桶，对应数字 0 到 9。然后，从最低位开始，遍历每一位进行分配和收集操作：分配阶段将每个元素根据当前位数字放入对应桶中，收集阶段按桶顺序将元素取回数组。重复这一过程，直到处理完最高位。

以下是一个 Python 实现示例，我们将逐步解读代码关键部分。

```
def radix_sort(arr):
    # 步骤 1：找到最大值，确定最大位数
    max_num = max(arr)
```

```
    exp = 1 # 起始位：个位

    while max_num // exp > 0:
        # 步骤 2：初始化 10 个桶
        buckets = [[] for _ in range(10)]

        # 步骤 3a：分配
        for num in arr:
            digit = (num // exp) % 10
            buckets[digit].append(num)

        # 步骤 3b：收集
        arr_index = 0
        for bucket in buckets:
            for num in bucket:
                arr[arr_index] = num
                arr_index += 1

        # 移动到下一位
        exp *= 10

# 测试代码
if __name__ == "__main__":
    data = [170, 45, 75, 90, 802, 24, 2, 66]
    radix_sort(data)
    print("排序后的数组：", data)
```

在这段代码中，首先通过 max(arr) 获取数组最大值，从而确定需要处理的位数。变量 exp 初始化为 1，表示从个位开始。循环条件 max_num // exp > 0 确保在所有位数处理完毕前持续迭代。在每一轮中，初始化十个空桶用于存储数字。分配阶段使用 (num // exp) % 10 计算当前位数字，这通过整数除法和取模操作实现，例如当 exp 为 1 时，(num // 1) % 10 得到个位数；当 exp 为 10 时，得到十位数。数字被添加到对应桶中，这里使用列表作为桶，天然保证了稳定性，因为元素按添加顺序存储。收集阶段遍历所有桶，按顺序将元素放回原数组，索引 arr_index 用于跟踪数组位置。最后，exp 乘以 10 移动到下一位。测试部分演示了算法对示例数组的排序效果，输出应为有序数组。

24　深入分析与探讨

基数排序的时间复杂度为 $O(k \times n)$，其中 k 是最大数字的位数，n 是数组长度。每一轮分配和收集操作各需 $O(n)$ 时间，总共进行 k 轮。与基于比较的排序算法如快速排序的 $O(n \log n)$ 相比，当 k 小于 $\log n$ 时，基数排序可能更高效；但如果数字范围极大（k 很大），效率可能下降。空间复杂度为 $O(n + r)$，其中 r 是基数（这里为 10），

因为需要额外空间存储桶和元素。

基数排序的优点包括线性时间复杂度和稳定性，适用于固定位数的整数或字符串排序。然而，它并非原地排序，需要额外内存，且仅适用于可分割为位的数据类型。优化方面，可以选择不同基数（如 256 进制）以利用位运算加速，或使用链表优化桶操作减少数据搬移。

基数排序以其非比较的独特思想，在排序算法家族中占据重要地位。通过按位分配和收集，它实现了高效排序，特别适合处理手机号、身份证号等固定位数数据。理解并实现基数排序，不仅能扩展算法知识，还能在特定场景下提升应用性能。读者可以尝试用其他语言实现，或探索 MSD 版本以加深理解。

25 互动与思考题

动手实现基数排序是巩固知识的好方法，读者可以尝试用 Java 或 C++ 重写代码，或实现 MSD 版本。思考题包括：如何修改算法以处理负数？对于长度不一的字符串排序，该如何调整？为什么在现实应用中快速排序更常见？这些问题有助于进一步探索算法的边界和优化方向。