#### 典型回答

1. 套接字

套接字为通信的端点。通过网络通信的每对进程需要使用一对套接字，即每个进程各有一个。每个套接字由一个 IP 地址和一个端口号组成。通常，套接字采用 CS 架构，服务器通过监听指定的端口，来等待特定服务。服务器在收到请求后，接受来自客户端套接字的连接，从而完成连接。

1. 管道

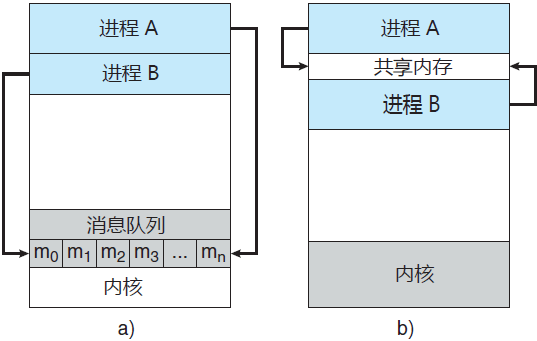
管道提供了一个相对简单的进程间的相互通信，普通管道允许父进程和子进程之间的通信，而命名管道允许不相关进程之间的通信。

#### 知识延伸

进程间通信有两种基本模型：**共享内存**和**消息传递**。

共享内存模型会建立起一块供协作进程共享的内存区域，进程通过向此共享区域读出或写入数据来交换信息。消息传递模型通过在协作进程间交换信息来实现通信。

下图给出了两个模型的对比：



很多系统同时实现了这两种模型。

消息传递对于交换较少数量的数据很有用，因为无需避免冲突。对于分布式系统，消息传递也比共享内存更易实现。共享内存可以快于消息传递，这是因为消息传递的实现经常采用系统调用，因此需要更多的时间以便内核介入。与此相反，共享内存系统仅在建立共享内存区域时需要系统调用；一旦建立共享内存，所有访问都可作为常规内存访问，无需借助内核。

对具有多个处理核的系统上，消息传递的性能要优于共享内存。共享内存会有高速缓存一致性问题，这是由共享数据在多个高速缓存之间迁移而引起的。随着系统处理核的日益增加，可能导致消息传递作为 IPC 的首选机制。

##### 共享内存系统

采用共享内存的进程间通信，需要通信进程建立共享内存区域。通常，这一片共享内存区域驻留在创建共享内存段的进程地址空间内。其它希望使用这个共享内存段进行通信的进程应将其附加到自己的地址空间。回忆一下，通常操作系统试图阻止一个进程访问另一个进程的内存。共享内存需要两个或更多的进程同意取消这一限制；这样它们通过在共享区域内读出或写入来交换信息。数据的类型或位置取决于这些进程，而不是受控于操作系统。另外，进程负责确保，它们不向同一位置同时写入数据。

##### 消息传递模型

消息传递提供一种机制，以便允许进程不必通过共享地址空间来实现通信和同步。对分布式环境（通信进程可能位于通过网络连接的不同计算机），这特别有用。

需要通信的进程应有一个方法，以便互相引用。它们可以使用直接或间接的通信。

对于直接通信，需要通信的每个进程必须明确指定通信的接受者或发送者。采用这种方案，原语 send() 和 receive() 定义如下：

send(P, message): 向进程 P 发送 message  
receive(Q, message): 从进程 Q 接收 message

这种方案展示了寻址的对称性，即发送和接收进程必须指定对方，以便通信。这种方案的一个变形采用寻址的非对称性，即只要发送者指定接受者，而接受者不需要指定发送者。采用这种方案，原语 send() 和 receive() 定义如下：

send(P, message): 向进程 P 发送 message  
receive(id, message): 从任何进程接收 message，这里变量 id 被设置成与其通信进程的名称

在间接通信中，通过邮箱或端口来发送和接收消息。邮箱可以抽象成一个对象，进程可以向其中存放消息，也可从中删除消息，每个邮箱都有一个唯一的标识符。一个进程可以通过多个不同邮箱与另一个进程通信，但是两个进程只有拥有一个共享邮箱时才能通信。原语 send() 和 receive() 定义如下：

send(A, message): 向邮箱 A 发送 message  
receive(A, message): 从邮箱 A 接收 message

#### 客户机 / 服务器通信

客户机 / 服务器通信的两种策略：套接字和管道。

套接字：

套接字为通信的端点。通过网络通信的每对进程需要使用一对套接字，即每个进程各有一个。每个套接字由一个 IP 地址和一个端口号组成。通常，套接字采用 CS 架构，服务器通过监听指定的端口，来等待特定服务。服务器在收到请求后，接受来自客户端套接字的连接，从而完成连接。

Java 提供三种不同类型的套接字。面向连接的 TCP 套接字用 Socket 类实现。无连接的 UDP 套接字使用 DatagramSocket 类。最后，MulticastSocket 类为 DatagramSocket 类的子类，多播套接字允许数据发送到多个接受者。

使用套接字的通信，虽然常用和高效，但是属于分布式进程之间的一种低级形式的通信。一个原因是，套接字只允许在通信线程之间交换无结构的字节流。客户机或服务器需要自己加上数据结构。

管道：

管道允许两个进程进行通信。管道是早期 UNIX 系统最早使用的一种 IPC 机制。管道为进程之间的相互通信提供了一种较为简单的方法，尽管也有一定的局限性。在实现管道时，应该考虑以下四个问题：

1. 管道允许单向通信还是双向通信？
2. 如果允许双向通信，它是半双工（数据在同一时间内只能按一个方向传输）的还是全双工（数据在同一时间内可在两个方向上传输）的？
3. 通信进程之间知否有一定的关系（如父子关系）？
4. 管道通信是否能够通过网络，还是只能在同一机器上进行？

下面就讨论两种常见类型的用于 UNIX 和 Windows 系统的管道：普通管道和命名管道。

**普通管道**

普通管道允许两个进程按标准的生产者和消费者方式进行通信：生产者向管道的一端（写入端）写，消费者从管道的另一端（读出端）读。因此，普通管道是单向的，只允许单向通信。如果需要双向通信，那么就要采用两个管道，而每个管道向不同方向发送数据。

在 UNIX 系统上，普通管道的创建采用函数：

pipe(int fd[])

这个函数创建了一个管道，以便通过文件描述符 int fd[] 来访问：fd[0] 为管道的读出端，而 fd[1] 为管道的写入端。Unix 将管道作为一种特殊类型的文件。因此，访问管道可以采用普通的系统调用 read() 和 write()。

普通管道只能由创建进程所访问。通常情况下，父进程创建了一个管道，并使用它与其子进程通信（该进程由 fork() 来创建）。如前面所述，子进程继承了父进程打开的文件的打开文件。由于管道是一种特殊类型的文件，因此子进程也继承了父进程的管道。

对于 Windows 系统，普通管道被称为匿名管道，它们的行为类似于 Unix 的管道：它们是单向的，通信进程之间具有父子关系。

采用普通管道的进程需要有父子关系，这意味着，这些管道只可用于同一机器的进程间通信。

**命名管道**

普通管道提供了一个简单的机制，允许一对进程通信。然鹅，只有当进程相互通信时，普通管道才存在。对于 Unix 和 Windows 系统，一旦进程已经完成通信并终止了，那么管道就不存在了。

命名管道提供了一个更强大的通信工具。通信可以是双向的，并且父子关系不是必需的。当建立一个命名管道后，多个进程多可用它通信。事实上，在一个典型的场景中，一个管道有多个写者。此外，当通信进程完成后，命名管道继续存在。虽然 Unix 和 Windows 系统都支持命名管道，但是实现细节具有很大不同。

对于 Unix，命名管道为 FIFO。一旦创建，它们表现为文件系统的典型文件。通过系统调用 mkfifo()，可以创建 FIFO；通过系统调用 open()、read()、write() 和 close()，可以操作 FIFO。FIFO 会一直存在，直到它被显式的从文件系统中删除。虽然 FIFO 允许双向通信，只允许半双工传输。如果数据要在两个方向上传输，那么通常使用两个 FIFO。此外，通信进程应位于同一台机器上，如果需要两个不同系统之间的通信，那么应该使用套接字。

与 UNIX 系统相比，Windows 系统的命名管道通信机制更加丰富。允许全双工通信，并且通信进程可以位于同一机器或不同机器。此外，UNIX 的 FIFO 只支持字节流的数据，而 Windows 系统允许字节流或消息流的数据。