祝你如愿以偿。

计算机网络

网络协议分层

国际标准化组织 ISO 提出了 OSI 开放互连的七层计算机网络模型，从上到下分别是应用层、表示层、会话层、运输层、网络层、链路层和物理层。OSI 模型的概念清楚，理论也比较完善，但是既复杂又不实用。还有一种是 TCP/IP 体系结构，它分为四层，从上到下分别是应用层、运输层、网际层和网络接口层，不过从实质上将只有三层，因为最下面的网络接口层并没有什么具体内容。因特网的协议栈使用一种五层的模型结构，从上到下依次是**应用层、运输层、网络层、链路层和物理层**，其中下层是为上层提供服务的，每层执行某些动作或使用下层的服务来提高服务。

应用层

应用层是网络体系结构中的最高层，应用层的任务就是**通过应用进程之间的交互来完成特定网络应用**，这一层的数据单元叫做**报文**。

应用层的协议定义了**应用进程之间通信和交互的规则**，主要包括了域名系统 DNS、支持万维网的 HTTP 协议、支持电子邮件的 SMTP 协议、文件传输协议 FTP 等。

域名解析系统 DNS

DNS 被设计为一个联机分布式数据库系统，并采用客户服务器方式。DNS 使大多数名字都在本地进行解析，仅少量解析需要在互联网上通信，因此 DNS 的效率很高。由于 DNS 是分布式系统，即使单个计算机出现了故障也不会妨碍到整个 DNS 系统的正常运行。

主机向本地域名服务器的查询一般都采用递归查询，递归查询指如果主机所询问的本地域名服务器不知道被查询域名的 IP 地址，那么本地域名服务器就以 DNS 客户的身份向其他根域名服务器继续发出查询请求报文。递归查询的结果是要查询的 IP 地址，或者是报错，表示无法查询到所需的 IP 地址。

本地域名服务器向根域名服务器查询通常采用迭代查询，迭代查询指当根域名服务器收到本地域名服务器发出的迭代查询请求报文时，要么给出所要查询的 IP 地址，要么告诉它该向哪一个域名服务器进行查询。本地域名服务器也可以采用递归查询，这取决于最初的查询请求报文设置的查询方式。

文件传送协议 FTP

FTP 使用 TCP 可靠的运输服务，FTP 使用客户服务器方式，一个 FTP 服务器进程可以同时为多个客户进程提供服务，在进行文件传输时，FTP 的客户和服务器之间要建立两个并行的 TCP 连接：控制连接和数据连接，实际用于传输文件的是数据连接。

电子邮件系统协议 SMTP/POP3/IMAP

一个电子邮件系统有三个主要组成构件，即用户代理、邮件服务器、以及邮件协议。

从用户代理把邮件传送到邮件服务器，以及在邮件服务器之间的传送都要使用 SMTP，但用户代理从邮件服务器读取邮件时则要使用 POP3 或 IMAP 协议。

基于万维网的电子邮件使用户可以利用浏览器收发电子邮件，用户浏览器和邮件服务器之间使用 HTTP 协议，而邮件服务器之间的传送仍然使用 SMTP 协议。

运输层

运输层的任务就是**负责向两台主机中进程之间的通信提供通用的数据传输服务**，应用进程利用该服务来传送应用层报文。由于一台主机同时可以运行多个进程，因此运输层具有复用和分用的功能，复用就是多个应用层进程可以同时使用下面运输层的服务，分用就是把运输层收到的信息分别交付给上面应用层中的对应进程。

运输层主要使用两种协议：① 用户数据报协议 UDP，这是一种提供无连接的、尽最大努力交付的数据传输服务，不保证数据传输的可靠性，数据传输单位是用户数据报。② 传输控制协议 TCP，这是一种面向连接的、可靠的数据传输服务，数据传输单元是报文。

网络层

网络层负责**为分组交换网上的不同主机提供通信服务**，在发生数据时，网络层把数据层产生的报文或用户数据报封装成**分组**进行传送，由于网络层使用 IP 协议，因此分组也叫 **IP 数据报**。网络层的另一个任务就是选择合适的路由，使源主机运输层所传下来的分组能够通过网络中的路由器找到目的主机。

网络层的协议包括了网际协议 IP、地址解析协议 ARP、网际控制报文协议 ICMP 以及路由选择协议 RIP/OSPF/BGP-4 等。

网际协议 IP

网际协议 IP 是 TCP/IP 体系中两个最主要的协议之一，一般指的是 IPv4。与 IP 协议配套使用的协议还有 ARP、ICMP 和 IGMP，IP 使用 ARP，ICMP 和 IGMP 要使用 IP。由于网际协议 IP 是用来使互连起来的许多计算机网络能够进行通信的，因此 TCP/IP 体系中的网络层也称网际层或 IP 层。要解决 IP 地址耗尽的问题，根本方法是采用具有更大地址空间的新版本 IP 协议即 IPv6，向 IPv6 过渡可以使用双协议栈或使用隧道技术。

地址解析协议 ARP

由于 IP 协议使用了 ARP 协议，因此把 ARP 协议归到网络层，但 ARP 的作用是通过一个 ARP 高速缓存存储本地局域网的各主机和路由器的 IP 地址到硬件地址的映射表，以从网络层的 IP 地址解析出在数据链路层使用的硬件地址，因此也可以把 ARP 划归在数据链路层。与 ARP 对应的协议是 RARP，逆地址解析协议，作用是使只知道自己硬件地址的主机能够找出 IP 地址，但被 DHCP 协议取代。

路由选择协议 RIP/OSPF/BGP-4

路由选择协议有两大类：内部网关协议，如 RIP 和 OSPF；外部网关协议，如 BGP-4。

RIP 是分布式的基于距离向量的路由选择协议，只适用于小型互联网。RIP 按照固定的时间间隔与相邻路由器交换信息，交换的信息是当前路由表。OSPF 是分布式的链路状态协议，适用于大型互联网，只在链路状态发生变化时才向本自治系统中的所有路由器用洪泛法发送与本路由器相邻的所有路由器的链路状态信息。

BGP-4 是不同自治系统的路由器之间交换路由信息的协议，是一种路径向量路由选择协议。其目标是寻找一条能够到达目的网络且比较好的路由而不是最佳路由。

网际控制报文协议 ICMP

ICMP 报文作为 IP 数据报的数据，加上首部后组成 IP 数据报发送出去，使用 ICMP 并非为了实现可靠传输，ICMP 允许主机或路由器报告差错情况和提供有关异常情况的报告。ICMP 报文的种类有两种，即 ICMP 差错报告报文和 ICMP 询问报文。

ICMP 的一个重要应用就是分组间探测 PING，用来测试两台主机之间的连通性，PING 使用了 ICMP 回送请求与回送回答报文。

网际组管理协议 IGMP

IP 多播使用 IGMP 协议，IGMP 并非在互联网范围内对所有多播组成员进行管理，它不知道 IP 多播组包含的成员个数也不知道这些成员都分布在哪些网络上。

IGMP 协议是让连接在本地局域网上的多播路由器知道本局域网上是否有主机上的某个进程参加或推出了某个多播组。

链路层

数据链路层的任务是**将网络层交下来的 IP 数据报组装成帧**，在两个相邻结点之间的链路上传输帧，每一帧包括数据和必要的控制信息（同步信息、地址信息、差错控制等）。在接收数据时，控制信息使接收端能够知道一个帧从哪个比特开始到哪个比特结束，这样链路层就可以从帧中提取出数据部分上交给网络层。控制信息还使接收端能够检测到所收到的帧有无差错，如果有差错就简单地丢弃这个帧以免继续传送而浪费网络资源。

数据链路层的协议包括了点对点协议 PPP 和 CSMA/CD 协议等。

点对点协议 PPP

在通信线路质量较差的年代，使用高级数据链路控制 HDLC 作为实现可靠传输的数据链路层协议，但现在 HDLC 已经很少使用了，对于点对点的链路，简单得多的点对点协议 PPP 是目前使用得最广泛的数据链路层协议。PPP 协议的特点是简单、只检测差错而不纠正差错、不使用序号也不进行流量控制、可同时支持多种网络层协议。

CSMA/CD 协议

以太网采用的是具有冲突检测的载波监听多点接入 CSMA/CD 协议，协议的要点是：发送前先监听、边发送边监听，一旦发现总线上出现了碰撞就立即停止发送。然后按照退避算法等待一段随机时间后再次发送，因此每一个站在自己发送数据之后的一小段时间内存在遭遇碰撞的可能性。以太网上各站点都平等地争用以太网信道。

物理层

物理层的任务是尽可能地**屏蔽掉传输媒体和通信手段的差异**，使物理层上面的数据链路层感觉不到这些差异，使其只需考虑本层的协议和服务。

物理层所传输的数据单位是比特，发送方发送 1 或 0，接收方也接收 1 或 0，因此物理层需要考虑用多大的电压代表 1 或 0，以及接收方如何识别出发送方所发送的比特。除此之外，物理层还要确定连接电缆的插头应当有多少根引以及各引脚如何连接等问题。

TCP

TCP 特点

TCP 是面向连接的运输层协议，一个应用进程在向另一个进程发送数据之前，两个进程必须先建立 TCP 连接，发送某些预备报文段，建立确保数据传输的参数。作为 TCP 连接建立的一部分，连接双方都将初始化与 TCP 连接相关的许多状态变量。这种连接不是电路交换网络中的端到端电路这种物理连接，而是一种逻辑连接，TCP 报文要先传送到 IP 层加上 IP 首部后，再传到数据链路层，加上链路层的首部和尾部后才离开主机发送到物理层。

TCP 连接提供全双工服务，允许通信双方的应用进程在任何时候都能发送数据。TCP 连接的两端都有各自的发送缓存和接收缓存，用来临时存放通信数据。在发送时，应用程序把数据传送给 TCP 缓存后就可以做自己的事，而 TCP 在合适的时候会把数据发送出去。在接收时，TCP 把收到的数据放入缓存，上层应用程序会在合适的时候读取缓存数据。

TCP 连接是点对点的，每一条 TCP 连接只能有两个端点，即只能是单个发送方和单个接收方之间的连接。

TCP 提供可靠的交付服务，通过 TCP 连接传送的数据无差错、不丢失、不重复，按序到达。

TCP 是面向字节流的，流是指流入到进程或从进程中流出的字节序列。面向字节流的含义是：虽然应用程序和 TCP 的交互是一次一个数据块，但是 TCP 把应用程序交下来的数据仅仅看成一连串无结构的字节流。TCP 不保证接收方应用程序收到的数据块和发送方应用程序发出的数据块具有对应大小的关系，但是接收方应用程序收到的字节流必须和发送方应用程序发出的字节流完全一样。接收方应用程序必须有能力识别收到的字节流，并把它还原成有意义的应用层数据。

TCP 报文结构

TCP 传送的数据单元是报文段，一个 TCP 报文段分为首部和数据两部分。首部的前 20 个字节是固定的，后面有 4n 字节是根据需要而增加的选项，因此 TCP 首部的最小长度是 20 字节.TCP 首部的重要字段如下：

源端口和目的端口：各占 2 字节，分别写入源端口号和目的端口号，TCP 的分用功能是通过端口实现的，分用就是指运输层从 IP 层收到发送给各应用进程的数据后，把数据交付给正确的套接字的工作。

序号：占 4 字节。TCP 是面向字节流的，在一个 TCP 连接中传送的字节流中的每一个字节都按顺序编号，首部中的序号字段值指的是本报文段所发送的数据的第一个字节的序号。序号使用 mod2^{32} 计算，每增加到 2^{31}-1 后下一个序号就又回到 0。

确认号：占 4 字节，是期望收到对方下一个报文段的第一个数据字节的序号。如果确认号为 N，代表到序号 N-1 为止的所有数据已经正确收到。序号有 32 位长，一般情况下可以保证当序号重复使用时，旧序号的数据早已通过网络到达终点了。

数据偏移：占 4 字节，实际是TCP 报文段的首部长度，指出了 TCP 报文段的数据起始处到 TCP 报文段的起始处的距离。由于首部中有长度不确定的选项字段，因此数据偏移字段是必要的。

标志字段：占 6 位。URG 是紧急标志，URG=1 时告诉系统此报文段中有紧急数据，应尽快传送，而不按照原来的排队顺序传送，和紧急指针配合使用，紧急指针指出了本报文段中紧急数据的字节数和位置。ACK 是确认标志，ACK=1 时表示成功接收了报文段。SYN 是同步标志，在建立连接时用来同步序号，当 SYN=1 而 ACK=0 时，表示一个连接请求报文段，响应时 SYN 和 ACK 都为 1，因此 SYN=1 表示一个连接请求或连接响应报文。FIN 是终止标志，用来释放一个连接，当 FIN=1 时表示报文段发送方的数据已发送完毕，并要求释放连接。PSH 是推送标志，PSH=1 时接收方就不等待整个缓存填满了再向上交付而是尽快交付数据。RST 是复位标志，当 RST=1 时表示 TCP 连接出现了严重错误，必须释放连接再重新建立连接。

接收窗口：占 2 字节，指的是发送本报文段一方的接收窗口，告诉对方从本报文首部的确认号算起允许对方发送的数据量。窗口值是用来限制发送方的发送窗口的，因为接收方的数据缓存空间是有限的。

检验和：占 2 字节，检验范围包括首部和数据两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

可靠传输协议 ARQ

自动重传请求 ARQ 包括了停止等待协议、回退 N 步协议和选择重传协议，后两种结合了窗口机制，属于连续 ARQ 协议。

停止等待协议

停止等待就是每发送完一个分组就停止发送，等待对方的确认，在收到确认之后再发送下一个分组。停止等待协议包括了三种情况：

1. 无差错情况

A 发送分组 M_1，发送完后就暂停并等待 B 的确认。B 收到了 M_1 之后就向 A 发送确认，当 A 收到确认之后就再发送下一个分组 M_2。

2. 出现差错的情况

当 B 收到 M_1 时检测出了差错就丢弃了 M_1，其他什么也不做，也可能是 M_1 在传输过程中丢失了，B 什么都不知道。在这两种情况下 B 都不会发送任何确认信息，解决方法是：A 只要超过一段时间没有收到确认就认为刚才发送的分组丢失了，因而重传前面发过的分组，这就叫**超时重传。**要实现超时重传，就要在每发送完一个分组时设置一个**超时计时器**，如果在超时计时器到期之间收到了对方的确认就进行撤销。

有三点需要注意：① A 在发送完一个分组后必须暂时保留已发送分组的副本在超时重传时使用，只有在收到确认后才清除副本。② 分组和确认分组都必须进行编号，这样才能明确是哪一个发送出去的分组进行了确认。③ 超时计时器设置的时间应当比数据在分组传输的平均往返时间稍长一些，如果设置过短会产生不必要的重传，如果设置过长会降低通信效率。

3. 确认丢失和确认迟到

B 发送的对 M_1 的确认丢失了，A 在设定的超时重传时间内没有收到确认，并不知道是自己发送的问题还是 B 的问题，因此 A 就会超时重传。假设 B 又收到了重传分组 M_1，此时 B 会采取两个行动：① 丢弃这个分组，不向上层交付。② 重新向 A 发送确认。

还有另一种情况就是 B 发送的确认并没有丢失但是迟到了，A 会受到重复的确认，此时 A 会收下并丢弃。通常 A 最终总是可以收到对所有发出的分组的确认，如果 A 不断重传分组但总是收不到确认，就说明通信线路质量太差，不能进行通信。

停止等待协议的优点是简单，但缺点是信道利用率太低。为了提高传输效率，发送方可以不使用停止等待协议，而是采用流水线传输。流水线传输就是发送方可连续发送多个分组，不必每发送完一个分组就停下来等待对方的确认。这样可以使信道上一直有数据不间断地传送，流水线传输中可能会遇到差错，解决差错的两种基本方法是回退 N 步和选择重传。

回退 N 步协议

在回退 N 步即 GBN 协议中，允许发送方发送多个分组而不需要等待确认。在 GBN 中发送方看到的序号可以分为四个范围，已经发送且被确认的的序号、已经发送还未确认的序号、允许发送但还未发送的序号和不允许发送的序号。其中已经发送但还未确认的序号和允许发送但还未发送的序号可以被看作一个长度为 N 的窗口，随着协议的运行该窗口在序号空间向前滑动，因此 GBN 协议也被称为滑动窗口协议。

GBN 发送方必须响应三种类型的事件：

- **上层的调用**

  当上层调用发送方法时，发送方首先检查发送窗口是否已满，即是否有 N 个已发送但未确认的分组。如果窗口未满，则产生一个分组并将其发送并更新相应变量，如果窗口已满，发送方指需将数据返回给上层，隐式说明该窗口已满。实际实现中，发送方更可能缓存而不是立即发送这些数据，或者使用同步机制允许上层在仅当窗口不满时才调用发送方法。

- **收到一个 ACK**

  在 GBN 协议中，对序号为 n 的分组确认采用累积确认的方式，对按序到达的最后一个分组发送确认，表明接收方已经正确接收到序号为 n 的以前且包括 n 在内的所有分组。例如发送了序号为 1~5 的五个分组，除了第三个全部收到了，那么确认序号就是 2。

- **超时事件**

  回退 N 步的名字来源于出现丢失和时延过长时发送方的行为，就像在停止等待协议中那样，如果超时，发送方会重传所有已经发送但还未确认过的分组。如果收到一个 ACK，但仍有已发送但未确认的分组，则计时器也会重新启动。

在 GBN 协议中，接收方丢弃所有失序分组，即使是正确接收的也要丢弃，这样做的理由是接收方必须按序将数据交付给上层。这种做法的优点是接收缓存简单，即接收方不需要缓存任何失序分组。不过丢弃一个正确失序分组的缺点是随后对该分组的重传也许也会错误，而导致更多的重传。

选择重传协议

GBN 协议潜在地允许用多个分组填充流水线，因此避免了停止等待协议中的信道利用问题，但是 GBN 本身也存在性能问题，单个分组的差错就能引起 GBN 重传大量分组，许多分组根本没有重传必要。随着信道差错率的增加，流水线会被不必要重传的分组所充斥。

选择重传即 SR 协议，通过让发送方仅重传那些它怀疑在接收方出错的分组而避免不必要的重传。这种个别的、按需重传，要求接收方逐个确认正确接收的分组，再次用窗口长度 N 来限制流水线中未完成和未被确认的分组数。与 GBN 不同的是，发送方已经收到了窗口中对某些分组的 ACK。

接收方将确认一个正确接收的分组，不管是否按序。失序分组将被缓存直到所有丢失分组都收到，这时才可以将一批分组按序交付上层。

TCP 可靠原理

TCP 的可靠传输包含很多机制，例如使用**检验和**来检测一个传输分组中的比特错误、使用**定时器**来用于超时重传一个分组、使用**序号**来检测丢失的分组和冗余副本、使用**确认**来告诉发送方确认的分组信息、使用**否定确认**来告诉发送方某个分组未被正确接收。

TCP 的发送方仅需维持已发送过但未被确认的字节的最小序号和下一个要发送的字节的序号，从这种角度看 TCP 更像一个 GBN 协议。但是 TCP 和 GBN 有一些显著的区别，许多 TCP 实现会将正确接收但失序的报文段缓存起来。当分组 n 丢失时，GBN 会重传 n 之后的所有分组，但是 TCP 至多只会重传分组 n。对 TCP 提出的一种修改意见是选择确认，它允许 TCP 接收方有选择地确认失序报文段，而不是累积地确认最后一个正确接收的有序报文段，从这个角度看 TCP 又像 SR 协议。**因此 TCP 的差错恢复机制是一种 GBN 和 SR 的结合体。**

除此之外，TCP 还使用**流量控制**和**拥塞控制**来保证可靠性。

滑动窗口

滑动窗口以字节为单位。发送端有一个发送窗口，窗口中的序号是允许发送的序号，窗口的后沿是已经发送并且确认的序号，窗口的前沿是不允许发送的序号。窗口的后沿可能不动（代表没有收到新的确认），也有可能前移（代表收到了新的确认），但是不会后移（不可能撤销已经确认的数据）。窗口的前沿一般是向前的，也有可能不动（表示没有收到新的请求或对方的接收窗口变小），也有可能收缩，但 TCP 强烈不建议这么做，因为发送端在收到通知前可能已经发送了很多数据，此时如果收缩窗口可能会产生错误。

滑动窗口的状态需要3个指针p1，p2 和 p3。p1 之前的序号表示已经发送并且确认的序号，p1~p2 的序号表示已经发送但还没有确认的序号，p2~p3 表示允许发送的序号，也叫可用窗口，p1~p3 表示发送窗口，p3 之后的序号表示不可发送的序号。

发送缓存用来暂时存放发送应用程序传给发送方 TCP 准备发送的数据和已经发送但还没确认的数据。接收缓存用来暂时存放按序到达的但尚未被应用程序读取的数据以及未按序到达的数据。

注意三点：① 发送窗口根据接收窗口设置，但并不总是一样大，还要根据网络的拥塞情况调整。② 对于不按序到达的数据，TCP 通常存放在接收窗口，等到字节流缺少的字节收到后再按序交付上层应用程序。③ 接收方必须有累积确认功能，可以减小传输开销，可以在合适的时候发送确认，也可以在自己有数据需要发送时捎带确认。但是接收方不能过分推迟发送确认，不能超过0.5秒。

流量控制

如果某个应用程序读取数据的速度较慢，而发送方发送得太多、太快，发送的数据就会很容易使连接的接收缓存溢出，TCP 为它的应用程序提供了流量控制以消除发送方使接收方缓存溢出的可能性。流量控制是一个速度匹配服务，即发送方的发送速率与接收方的应用程序读取速率相匹配。

TCP 通过让**发送方维护一个接收窗口的变量**来提供流量控制。通俗地说，接收窗口用于给发送方一个指示，该接收方还有多少可用的缓存空间，因此方法方的发送窗口不能超过接收方给出的接收窗口的数值。因为 TCP 是全双工通信，在连接两端的发送方都各自维护一个接收窗口。

当接收窗口 rwnd 减小到 0 时，就不再允许发送方发送数据了。但是可能存在一种情况，当发生了零窗口报文段不久后，发送方的接收缓存又有了一些存储空间，因此又发生了新的报文说明自己的接收窗口大小，但是这个报文可能会在传输过程中丢失。接收方就会一直等待发送方的非零窗口通知，而发送方也一直在等待接收方发送数组，形成一种死锁的状态。为了解决这个问题，TCP 为每一个连接设有一个持续计时器，只要 TCP 连接的一方收到对方的零窗口通知就启动该计时器，到期后发送一个零窗口探测报文，如果仍为 0 就重新设置计时器的时间，如果对方给出了新的窗口值就可以解决可能出现的死锁问题。

还有一种问题叫做**糊涂窗口综合症**，当接收方处理接收缓冲区数据很慢时，就会使应用进程间传送的有效数据很小， 极端情况下有效数据可能只有 1 字节但传输开销却有 40 字节（20字节的 IP 头以及 20 字节的 TCP 头)，导致网络效率极低。为了解决这个问题，可以让接收方等待一段时间，使得接收缓存有足够的空间容纳一个最长报文段或者等到接收缓存已有一半的空闲空间。发送方也不要发送太小的报文，而是把数据积累成足够大的报文或达到接收方缓存空间的一半时才发送。

拥塞控制

网络中对资源需求超过了资源可用量的情况就叫做拥塞。当吞吐量明显小于理想的吞吐量时就出现了轻度拥塞，当吞吐量随着负载的增加反而下降时，网络就进入了拥塞状态。当吞吐量降为 0 时，网络已无法正常工作并陷入死锁状态。拥塞控制就是尽量减少注入网络的数据，减轻网络中的路由器和链路的负担。**拥塞控制是一个全局性的问题，它涉及网络中的所有路由器和主机，而流量控制只是一个端到端的问题，是两个端点之间通信量的控制。**

根据网络层是否为运输层拥塞控制提供显式帮助可以将拥塞控制的方法区分为两种：端到端拥塞控制和网络辅助的拥塞控制。TCP 使用端到端的拥塞控制，因为 IP 层不会向端系统提供显式的网络拥塞反馈。TCP 所采取的方法是让每一个发送方根据所感知到的网络拥塞程度来限制其向连接发送数据的速率。如果一个 TCP 发送方感知到它到目的地之间的路径上没什么拥塞则会增加发送速率，如果发送方感知到拥塞就会降低其发送速率。限制发送速率是通过拥塞窗口来实现的，它对发送方能向网络中发送流量的速率进行了限制。判断拥塞是通过超时或者连续接收到 3 个冗余 ACK 实现的。

TCP 的拥塞控制算法主要包括了慢启动、拥塞避免和快恢复。慢启动和拥塞避免是 TCP 的强制部分，差异在于对收到的 ACK 做出反应时 cwnd 增加的方式，慢启动比拥塞避免要更快地增加 cwnd 的长度。快恢复是推荐部分，对 TCP 发送方不是必需的。

1. 慢启动

在慢启动状态，拥塞窗口 cwnd 的值以一个 MSS 最大报文段开始并且每当传输的报文段首次被确认就增加一个 MSS。因此每经过一个 RTT 往返时间，拥塞窗口就会翻倍，发送速率也会翻倍。因此 TCP 的发送速率起始很慢，但是在慢启动阶段以指数增长。

结束慢启动有三种情况：① 如果存在一个超时指示的丢包事件，即发生了拥塞，TCP 发送方就会将 cwnd 设置为 1 并重新开始慢启动过程。它还将慢启动阈值设置为 cwnd/2，即检测到拥塞时将慢启动阈值设置为拥塞窗口的一半。② 当拥塞窗口达到慢启动阈值时就会结束慢启动而进入拥塞避免模式。③ 最后一种结束慢启动的方式是，如果检测到三个冗余的 ACK，TCP 就会执行快重传并进入快恢复状态。

2. 拥塞避免

一旦进入拥塞避免状态，cwnd 的值大约是上次遇到拥塞时的值的一半，即距离拥塞可能并不遥远。因此 TCP 无法再每经过一个 RTT 就将 cwnd 的值翻倍，而是采用一种较为保守的方法，每个 RTT 后只将 cwnd 的值增加一个 MSS。这能够以几种方式完成，一种通用的方法是发送方无论何时收到一个新的确认，都将 cwnd 增加一个 MSS。

当出现超时时，TCP 的拥塞避免和慢启动一样，cwnd 的值将被设置为 1，并且将慢启动阈值设置为 cwnd 的一半。

3. 快恢复

有时候个报文段丢失，而网络中并没有出现拥塞，如果使用慢启动算法就会降低传输效率。这时应该使用快重传来让发送方尽早知道出现了个别分组的丢失，快重传要求接收端不要等待自己发送数据时再捎带确认，而是要立即发送确认。即使收到了乱序的报文段后也要立即发出对已收到报文段的重复确认。当发送方连续收到三个冗余 ACK 后就知道出现了报文段丢失的情况，会立即重传并进入快恢复状态。

在快恢复中，会调整慢启动阈值为 cwnd 的一半，并进入拥塞避免状态。

TCP 连接和释放机制

三次握手

TCP 是全双工通信，任何一方都可以发起建立连接的请求，假设 A 是客户端，B 是服务器。

初始 A 和 B 均处于 CLOSED 状态，B 会创建传输进程控制块 TCB 并进入 LISTEND 状态，监听端口是否收到了 TCP 请求以便及时响应。

当 A 要发生数据时就向 B 发送一个连接请求报文，TCP 规定连接请求报文的 SYN=1，ACK=0，SYN 不可以携带数据，但要消耗一个序号，假设此时 A 发送的序号 seq 为 x。发送完之后 A 就进入了 SYN-SENT 同步已发送状态。

当 B 收到 A 的连接请求报文后，如果同意建立连接就会发送给 A 一个确认连接请求报文，其中 SYN=1，ACK=1，ack=x+1，seq=y，ack 的值为 A 发送的序号加 1，ACK 可以携带数据，如果不携带的话则不消耗序号。发送完之后，B进入 SYN-RCVD 同步已接收状态。

当 A 收到 B 的确认连接请求报文后，还要对该确认再进行一次确认，报文的 ACK=1，ack=y+1，seq=x+1，发送后 A 进入 ESTABLISHED 状态，当 B 接收到该报文后也进入 ESTABLISHED 状态，客户端会稍早于服务器端建立连接。

三次握手的原因主要有两个目的，信息对等和防止超时。

从信息对等的角度看，双方只有确定 4 类信息才能建立连接，即 A 和 B 分别确认自己和对方的发送和接收能力正常。在第二次握手后，从 B 的角度看还不能确定自己的发送能力和对方的接收能力，只有在第三次握手后才能确认。

三次握手也是防止失效连接突然到达导致脏连接，网络报文的生存时间往往会超过 TCP 请求超时时间，A 的某个超时连接请求可能会在双方释放连接之后到达 B，B 会误以为是 A 创建了新的连接请求，然后发送确认报文创建连接。因为 A 机器的状态不是 SYN_SENT，所以直接丢弃了 B 的确认数据。如果是两次握手，连接已经建立了，服务器资源被白白浪费。如果是三次握手，B 由于长时间没有收到确认信息，最终超时导致创建连接失败，因此不会出现脏连接。

四次挥手

当 A 已经没有要发送的数据时就会释放连接，会向 B 发送一个终止连接报文，其中 FIN=1，seq=u，u 的值为之前 A 发送的最后一个序号+1。发送完之后进入 FIN-WAIT-1 状态。

B 收到该报文后，发送给 A 一个确认报文，ACK=1，ack=u+1，seq=v，v 的值为 B 之前发送的最后一个序号+1。此时 A 进入了FIN-WAIT-2 状态，B 进入了 CLOSE-WAIT 状态，但连接并未完全释放，B 会通知高层的应用层结束 A 到 B 方向的连接，此时 TCP 处于半关闭状态。

当 B 发送完数据后准备释放连接时，就向 A 发送连接终止报文，FIN=1，同时还要重发ACK=1，ack=u+1，seq=w，seq 不是 v 的原因是在半关闭状态 B 可能又发送了一些数据，之后 B 进入 LAST-ACK 状态。

A 收到连接终止报文后还要再进行一次确认，确认报文中 ACK=1，ack=w+1，seq=u+1。发送完之后进入 TIME-WAIT 状态，等待 2MSL之后进入 CLOSED 状态，B 收到该确认后进入 CLOSED 状态，服务器端会稍早于客户端释放连接。

四次挥手的原因

第一点原因是为了保证被动关闭方可以进入 CLOSED 状态。MSL 是最大报文段寿命，等待 2MSL 可以保证 A 发送的最后一个确认报文能被 B 接收，如果该报文丢失，B 没有收到就会超时重传之前的 FIN+ACK 报文，而如果 A 在发送确认报文后就立即释放连接就无法收到 B 超时重传的报文，因而也不会再一次发送确认报文段，B 就无法正常进入 CLOSED 状态。

第二点原因是 2MSL 时间之后，本连接中的所有报文就都会从网络中消失，可以防止已失效连接的请求数据包与正常连接的请求数据包混淆而发生异常。

除此之外，TCP 还设有一个保活计时器，用于解决客户端主机故障的问题，服务器每收到一次客户的数据就重新设置保活计时器，时间为 2 小时。如果 2 小时内没有收到就间隔 75 秒发送一次探测报文，连续 10 次都没有响应后就关闭连接。

大量 TIME-WAIT 的原因、导致的问题、处理

在高并发短连接的 TCP 服务器上，服务器处理完请求后立刻主动正常关闭连接，这个场景下会出现大量 socket 处于 TIME-WAIT 状态。

TIME-WAIT 状态无法真正释放句柄资源，socket 使用的本地端口在默认情况下不能再被使用，会限制有效连接数量，成为性能瓶颈。

可以调小 tcp_fin_timeout 的值，将 tcp_tw_reuse 设为 1 开启重用，将 tcp_tw_recycle 设为 1 表示开启快速回收。

TCP 和 UDP 的区别

TCP 是面向连接的，而 UDP 是无连接的，发送数据之前不需要建立连接，减少了开销和发送数据之前的时延。

TCP 保证数据的可靠传输，UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的连接状态。

TCP 是面向字节流的，UDP 是面向报文的，发送方的 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文既不拆分也不合并，而是保留这些报文的边界。如果报文太长，IP 层在传送时可能需要分片，如果报文太短，会使 IP 数据报首部的相对长度太大，都会降低 IP 层的效率。

TCP 有拥塞控制，UDP 没有拥塞控制，因此网络中出现的拥塞不会降低源主机的发送速率。这对某些实时应用很重要，很多实时应用如 IP 电话、实时视频会议等要求源主机以恒定的速率发送数据，并且允许在网络发生拥塞时丢失一些数据，但却不允许网络有太大的时延，UDP 正好适合这种要求。

TCP 是点到点之间的一对一通信，UDP 支持一对一、一对多和多对多的交互通信。

UDP 的首部开销很小，只有 8 字节，相比 TCP 的 20 字节要短。

HTTP

HTTP 概况

HTTP 即超文本传输协议，是 Web 的应用层协议。HTTP 由两个程序实现，一个客户程序和一个服务器程序，客户程序和服务器程序运行在不同的端系统中，通过交换 HTTP 报文进行会话。HTTP 定义了这些报文的结构以及客户和服务器进行报文交换的方式，当用户请求一个 Web 页面时，浏览器向服务器发出对该页面中所包含对象的 HTTP 请求报文，服务器接收到请求并用包含这些对象的 HTTP 响应报文进行响应。

HTTP 使用 TCP 作为它的支撑运输协议，HTTP 客户首先发起一个与服务器的 TCP 连接，一旦连接建立，该浏览器和服务器进程就可以通过套接字访问 TCP。客户端的套接字接口是客户进程与 TCP 连接之间的门，服务器端的套接字接口则是服务器进程与 TCP 连接之间的门。客户向它的套接字接口发送 HTTP 请求报文并从它的套接字接口接收 HTTP 响应报文，类似的，服务器从它的套接字接口接收 HTTP 请求报文并向它的套接字接口发送 HTTP 响应报文。一旦客户向它的套接字接口发送一个 HTTP 请求报文，该报文就脱离了客户控制并进入 TCP 的控制，TCP 为 HTTP 提供可靠的数据传输服务，因此一个客户进程发出的每个 HTTP 请求报文最终都能完整地到达服务器，服务器进程发出地每个 HTTP 响应报文最终也可以完整地到达客户。这里体现了分层体系结构的优点，HTTP 协议不需要担心数据丢失，也不需要关注 TCP 从网络的数据丢失和乱序中如何恢复。

HTTP 是一种无状态的协议，服务器向客户发送被请求的文件，而不存储任何关于该客户的状态信息。假如某个客户在短时间内连续两次请求同一个对象，服务器并不会因为刚刚为该客户做出了响应就不再响应，而是重新进行响应。

非持续连接和持续连接

依据每个请求/响应对经过一个单独的 TCP 连接还是相同的 TCP 连接发送，可以将连接划分为非持续连接和持续连接。HTTP 默认使用持续连接，但是也可以配置成使用非持续连接。

非持续连接

使用非持续连接时，从服务器向客户传送一个包含了一个 HTML 文件和 10 个 JPG 对象的 Web 页面步骤：

① HTTP 客户进程在端口号 80 发起一个到服务器的 TCP 连接，并经它的套接字向服务器发送一个 HTTP 请求报文。② HTTP 服务器进程经它的套接字接收请求报文，从其存储器中检索出请求对象，在一个 HTTP 响应报文中封装对象，并通过套接字向客户发送响应报文。③ HTTP 服务器进程通知 TCP 断开该 TCP 连接，直到 TCP 确认客户已经完整接收到响应报文才会实际断开连接。④ HTTP 客户接收到响应报文，客户从报文中提取出 HTML 文件，得到多个 JPG 图像的引用，并对每个引用的 JPG 图像对象重复前三个步骤。

每个 TCP 连接会在服务器发送一个对象后关闭，即该连接并不为其他的对象而持续下来。每个 TCP 连接只传输一个请求报文和一个响应报文，假如请求一个 HTML 文件和其中包括的 10 个 JPG 图像就要产生 11 个 TCP 连接。

在非持续连接中，每次请求文件到接收响应大约需要花费两个 RTT 加上服务器传输文件的时间，RTT 就是往返时间，指一个短分组从客户到服务器然后再返回客户所发送的时间。其中三次握手的前两个部分占用一个 RTT，三次握手的第三个确认部分向服务器发送了请求报文，服务器接收到之后做出响应，这用去了另一个 RTT。

持续连接

非持续连接有很多缺点。第一，必须为每个请求的对象建立和维护一个新的连接，对于每个连接，在客户和服务器中都要分配 TCP 的缓冲区和保持 TCP 变量，这给 Web 服务器带来了很大负担。第二，每一个对象需要消耗两倍的 RTT，一个用于创建 TCP，一个用于请求和接收对象。

在 HTTP1.1 中使用了持续连接，服务器在发送响应后保持该 TCP 连接打开。在相同的客户与服务器之间，后续的请求和响应报文能够通过相同的连接进行传送。在使用持续连接的情况下，请求一个完整的 Web 页面可以使用单个持续的 TCP 连接，例如之前所说的包含一个 HTML 文件和 10 个 JPG 对象的 Web 页面就只需要一个 TCP 连接而不是 11 个。

HTTP 报文格式

HTTP 报文有两种，分为请求报文和响应报文。

请求报文

HTTP 请求报文的第一行叫做请求行，其后继的行叫做首部行。请求行有三个字段，包括方法、URL 和 HTTP 版本。方法包括了 GET、POST、HEAD、PUT 和 DELETE 等。绝大部分的 HTTP 请求报文使用 GET 方法，当使用 GET 方法时，在 URL 字段中会带有请求对象的标识。

首部行指明了对象所在的主机，其实已经存在 TCP 连接了，但是还需要首部行提供主机信息，这时 Web 代理高速缓存所要求的。通过包含 `Connection:close` 的首部行，可以告诉服务器不要麻烦地使用持续连接，它要求在发送完响应后就关闭连接。`User-agent` 可以用来指明用户代理，即向服务器发送请求的浏览器类型，服务器可以有效地为不同类型的用户代理发送实际相同对象的不同版本。

在首部行之后有一个空行，后面跟着的是实体。使用 GET 方法时实体为空，而使用 POST 方法时才会使用实体。当用户提交表单时，HTTP 客户通常使用 POST 方法，使用 POST 方法时用户仍可以向服务器请求一个 Web 页面，但 Web 页面的特定内容依赖于用户在表单字段中输入的内容。如果使用 POST 方法，则实体中包含的就是用户在表单字段的输入值。表单不是必须使用 POST 方法，也可以使用 GET。

HEAD 方法类似于 GET，当服务器收到一个使用 HEAD 方法的请求时，将会用一个 HTTP 报文进行响应，但是并不返回请求对象。通常开发者使用 HEAD 方法进行调试跟踪。PUT 方法常用于上传对象到指定的 Web 服务器上指定的目录，DELETE 方法允许用户或应用程序删除 Web 服务器上的对象。

响应报文

响应报文包括状态行、首部行和实体。状态行有三个字段，协议版本、状态码和对应的状态信息。实体是报文的主要部分，即所请求的对象本身。

服务器通过首部行来告诉浏览器一些信息。`Connection:close` 可以告诉客户发送完报文后将关闭该 TCP 连接。`Date` 是首部行指示服务器发送响应报文的日期和时间，这个时间不是对象创建或修改的时间，而是服务器从它的文件系统中检索到该对象，将该对象插入响应报文并发送的时间。`Server` 指明了服务器的类型，类似于请求报文中的 `User-agent` 。

状态码及其相应的短语指示了请求的结果，一些常见的状态码和相关短语如下：

状态码	短语	含义
200	OK	成功响应
301	Moved Permanently	请求的对象已经被永久转移了，新的 URL 定义在响应报文的 Location 首部行，客户将自动获取新的 URL。
302	Found	与301类似，但资源只是临时被移动，客户端应继续使用原有 URL。
400	Bad Request	一个通用的差错代码，标识该请求不能被服务器理解。
401	Unauthorized	未认证，缺乏相关权限。
402	Payment Required	保留，将来使用
403	Forbidden	服务器理解客户端的请求，但是拒绝执行。
404	Not Found	被请求的文档不在服务器上，有可能因为请求 URL 出错。
405	Method Not Allowed	客户端中请求的方法被禁止，例如限制 POST 方式但使用了 GET 访问。
500	Internal Server Error	服务器内部错误，无法完成请求。
501	Not Implemented	服务器不支持请求的功能，无法完成请求。
502	Bad Gateway	作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应。
503	Service Unavailable	由于超载或系统维护，服务器暂时的无法处理客户端的请求。
504	Gateway Timeout	充当网关或代理的服务器，未及时从远端服务器获取请求。
505	HTTP Version Not Supported	服务器不支持请求报文使用的 HTTP 协议版本。

cookie

HTTP 的无状态性简化了服务器的设计，并且提高了 Web 服务器的性能，使其可以同时处理大量的 TCP 连接。但是一个 Web 站点通常希望能够识别用户，可能是为了限制用户的访问，也可能为了把内容与用户身份关联起来，为此 HTTP 使用了 cookie，cookie 是一种客户端的会话技术，允许站点对用户进行追踪。

cookie 技术有四个组件

① 在 HTTP 响应报文中的一个 cookie 首部行。② 在 HTTP 请求报文中的一个 cookie 首部行。③ 在用户端系统中保留有一个 cookie 文件，并由用户的浏览器关联。④ 位于 Web 站点的一个后端数据库。cookie 的工作流程：当客户通过浏览器第一次访问某个站点时，该 Web 站点将产生一个唯一识别码，并以此作为索引在它的后端数据库中产生的一个表项。接下来服务器会用一个包含 `Set-cookie` 首部的

HTTP 响应报文对浏览器进行相应，当浏览器收到后将其添加到自己管理的 cookie 文件中，在下次访问该站点时，请求报文的首部行中就会包括这个识别码，尽管浏览器不知道客户是谁，但是可以确定是同一个客户进行了访问。

cookie 和 session 的区别

① cookie 只能存储 ASCII 码字符串，而 session 则可以存储任何类型的数据，因此在考虑数据复杂性时首选session。② session 存储在服务器，而 cookie 存储在客户浏览器中，容易被恶意查看。如果非要将一些隐私数据存在 cookie 中，可以将 cookie 值进行加密，然后在服务器进行解密。

Web 缓存

Web 缓存器也叫代理服务器，它是能够代表初始 Web 服务器来满足 HTTP 请求的网络实体。Web 缓存器有自己的磁盘存储空间，并在存储空间中保存最近请求过的对象副本。可以配置用户的浏览器，使得用户所有的 HTTP 请求首先指向 Web 缓存器。客户通过 Web 缓存器请求对象的步骤如下：

① 浏览器创建一个到 Web 缓存器的 TCP 连接，并向 Web 缓存器中的对象发送一个 HTTP 请求。② Web 缓存器进行检查，查看本地是否存储了该对象副本。如果有，Web 缓存器就向客户浏览器用 HTTP 响应报文返回该对象。③ 如果 Web 缓存器中没有该对象，它就打开一个与该对象的初始服务器的 TCP 连接，Web 缓存器在这个连接上发送一个请求并接受响应。④ Web 缓存器接收到响应后，在本地存储空间存储一份副本，并向客户的浏览器用 HTTP 响应报文发送该副本。

Web 缓存器既是服务器又是客户，当它接受浏览器的请求并响应时，它是一个服务器，当他向初始服务器发出请求并接收响应时，它是一个客户。在因特网上部署 Web 缓存器有两个原因，首先，Web 缓存器可以大大减少对客户请求的响应时间，特别是客户与初始服务器之间的带宽远低于客户与 Web 缓存器之间的带宽时更是如此。其次，Web 缓存器可以大大减少一个机构的接入链路到因特网的通信量，通过减少通信量，机构就不必基于增加带宽，可以降低费用。

输入一个 url 发生的事

① 分析 url

判断输入的 url 是否合法，如果不合法浏览器会使用默认的搜索引擎进行搜索。如果输入的是一个域名，默认会加上一个 http 前缀。

② DNS 查询

检查浏览器的 DNS 缓存，检查本地 hosts 文件的缓存，如果没有会向本地 DNS 服务器发送请求。

主机向本地 DNS 服务器发起请求是递归查询，如果找到则返回，否则会向根 DNS 查询。

根 DNS 查询是迭代查询，没有域名和 ip 的对应关系，而是告知可以查询的域名服务器地址。

本地 DNS 向得到的域名服务器发出请求，收到一个域名和 ip 关系，把结果返回给用户，并把结果保存到缓存中。

③ TCP 建立连接

拿到 ip 地址后，通过 TCP 的三次握手建立连接，按照协议规定的格式发送 HTTP 请求报文。

④ 处理请求

服务器收到 HTTP 请求报文后进行响应，主进程进行监听，创建子进程处理，先判断是否是重定向，如果是重定向则返回重定向地址。如果是静态资源则直接返回，否则通过 REST URL 在代码层面处理返回结果，最后返回 HTTP 响应报文。

⑤ 接收响应

浏览器收到 HTTP 响应报文后进行解析，首先查看响应报文在状态行的状态码，根据不同的状态码做不同的事，解析 HTML、CSS、JS 等文件。构建 DOM 树，渲染树，重绘，将像素发送 GPU 进行渲染，最后将渲染结果返回给用户并进行缓存。

⑥ TCP 断开连接

通过 TCP 的四次挥手断开连接，如果是 HTTP1.1 则会将连接保持一小段时间。

HTTPS

HTTP 存在的问题

HTTP 没有密码加密，无法保证通信内容不被窃听，攻击者可以截取客户发送的数据并得到他的信息。

HTTP 没有报文完整性验证，无法确保通信内容在传输过程中不被改变，攻击者可以篡改客户通信内容。

HTTP 没有身份鉴别，无法让通信双方确认对方的身份，攻击者可以伪装成客户或者服务器。

加密原理

HTTPS 即 HTTP over SSL，在 HTTP 传输上增加了 SSL 安全性服务。SSL 是安全套接字层，通过采用机密性、数据完整性、服务器鉴别以及客户鉴别来强化 TCP，主要用于为发生在 HTTP 之上的事务提供安全性。SSL 会对数据进行加密并把加密数据送往 TCP 套接字，在接收方，SSL 读取 TCP 套接字中的数据并解密，然后把数据交给应用层。HTTPS 采用混合加密机制，使用非对称加密传输对称密钥来保证传输过程的安全性，之后使用对称加密进行通信来保证通信过程的效率。

HTTPS 的传输过程主要分为两部分：通过 SSL 握手建立安全的 HTTPS 通道和在安全的通道上进行数据传输，SSL 握手的步骤如下：

① 客户发送它支持的密码算法列表，以及一个客户的不重数，不重数就是在一个协议的生存期只使用一次的数。

② 服务器从该列表中选择一种对称加密算法（例如 AES），一种公钥加密算法（例如 RSA）和一种报文鉴别码算法。服务器把它的选择以及证书和一个服务器不重数返回给客户。

③ 客户通过 CA 提供的公钥验证该证书，验证成功后提取服务器的公钥，生产一个前主密钥 PMS，用服务器的公钥加密该 PMS，并将加密的 PMS 发送给服务器。

④ 客户和服务器独立地从 PMS 和不重数中计算出仅用于当前 SSL 会话的主密钥 MS，然后该 MS 被切片以生成两个密码和两个报文鉴别码密钥。自从以后，客户和服务器之间发送的所有报文均被加密和鉴别（使用报文鉴别码）。

⑤ 客户和服务器分别发送所有握手报文的一个报文鉴别码。这一步是为了使握手免受篡改危害，在第一步中客户提供的算法列表是以明文形式发送的，因此可能被攻击者截获并删除较强的算法。当客户发送一个级联它以及发送和接收的所有握手报文的报文鉴别码，服务器能够比较这个报文鉴别码和它已经接受和发送的握手报文的报文鉴别码，如果不一致就终止连接。类似的，客户也可以通过服务器发送的报文鉴别码来检查一致性。

第一步和第二步中的不重复数用于防止重放攻击，每个 TCP 会话使用不同的不重复数就可以使加密密钥不同，当收到重放的 SSL 记录时，该记录无法通过完整性检查，假冒的电子事务不会成功。

当结束 SSL 会话时，需要在类型字段中指出该记录是否是用于终止 SSL 会话的。通过包含这样一个字段，如果客户或服务器在收到一个关闭 SSL 记录之前突然收到了一个 TCP FIN，就知道遭受了截断攻击。

网络安全

网络安全主要探讨的问题是攻击者如何攻击计算机网络，以及如何防御这些攻击，或者如何事先预防这样的攻击。

网络攻击

计算机网络面临的威胁主要有被动攻击和主动攻击。

被动攻击指攻击者从网络上窃听他人的通信内容，也叫截获。在被动攻击中，攻击者只是观察和分析某一协议数据单元 PDU 而不干扰信息流。攻击者可以通过观察 PDU 的协议控制信息部分，了解正在通信的协议的地址和身份，通过研究 PDU 的长度和发送频度，了解所交换的数据的某种性质。这种攻击又叫做流量分析。

主动攻击包括：

- **篡改**

  攻击者篡改网络上传输的报文，包括彻底中断传送的报文，甚至把完全伪造的报文发给接收端，这种攻击方式也叫做更改报文流。

- **恶意程序**

  ① 计算机病毒，能够传染其他程序的程序，主要通过修改其他程序来把自身或自身的变种复制进去完成。② 计算机蠕虫，通过网络通信能把自己从一个结点发往另一个节点并且自动启动运行的程序。③ 特洛伊木马，它执行的功能并非声称的功能而是恶意程序，例如一个编译程序除了完成编译任务外还偷偷地复制源程序。④ 逻辑炸弹，当运行环境满足某种特殊条件时就会执行特殊功能的程序，例如当日期为 22 号且为周三的时候就会删除所有文件。⑤ 后门入侵，指利用系统实现中的漏洞通过网络入侵系统。⑥ 流氓软件，一种未经用户同意就在用户计算机上安装并损害用户利益的软件。

- **拒绝服务DoS**

  DoS 攻击使得网络、主机或其他基础设施部分不能由合法用户使用。Web 服务器、电子邮件服务器、DNS 服务器和机构网络都能够成为 DoS 攻击的目标。大多数 DoS 攻击属于以下三种情况：① 弱点攻击，指向一台目标主机上运行的易受攻击的应用程序或操作系统发送制作精细的报文，如果适当顺序的多个分组发送给一个易受攻击的应用程序或操作系统，该服务器可能会停止运行甚至崩溃。② 带宽洪泛，指攻击者向目标主机发送大量的分组，分组数量之多使得目标的接入链路变得阻塞，使合法的分组无法到达服务器。③ 连接洪泛，指攻击者在目标主机中创建大量的半开或全开 TCP 连接，主机因这些伪造的连接而陷入困境，并停止接受合法的连接。

- **ARP欺骗**

  在使用以太网交换机的网络中，攻击者向某个以太网交换机发送大量的伪造源 MAC 地址，以太网交换机收到这样的帧就把虚假的 MAC 源地址填入到交换表中，由于伪造的数量很大很快就填满了表，导致以太网交换机无法正常工作。

对于被动攻击可以采用各种数据加密技术，对付主动攻击则需要将加密技术与适当的鉴别技术相结合。

安全的计算机网络具有以下特性：

- **机密性**

  仅有发送方和希望的接收方能够理解报文传输的内容，因为窃听者可以截获报文，这要求报文必须进行加密，使截取而都报文无法被截获者理解。为了使网络具有保密性，需要使用各种密码技术。

- **报文完整性**

  通信内容在传输过程中需要被确保未被恶意篡改或意外改动。

- **端点鉴别**

  发送方和接收方都应该能证实通信过程中的另一方，以确信通信的另一方确实具有其声称的身份。在实际应用中，报文完整性和端点鉴别往往是不可分割的，因为假设通过了端点鉴别保证了通信双方的身份却没有通过报文鉴别保证报文的完整性是没有意义的。

- **运行安全性**

  几乎所有机构都有与互联网相连的网络，这些网络都因此具有潜在的安全问题。需要通过访问控制来确保网络的安全性，防火墙位于机构和公共网络之间，控制接入和来自网络的分组；入侵检测系统指需深度分组检查任务，向网络管理员发出有关可疑活动的警告。

密码技术

密码技术使发送方可以伪装数据，接收方需要从伪装的数据中恢复出初始数据，而入侵者不能从截获到的数据中获得任何信息。报文的最初形式被称为明文，在使用加密算法加密后得到的加密报文被称为密文。密码体制分为两种，对称密钥密码体制和公开密钥密码体制。

对称密钥密码体制使用相同的加密密钥和解密密钥，对称加密的运算速度快，但是安全性差，因为在密钥传输的过程中可能会被截获。数据加密标准 DES 属于对称密码密钥，这种密码的保密性仅仅取决于对密钥的保密，而算法是公开的，之后被更加安全的高级加密标准 AES 所取代。

公开密钥密码体制使用公钥进行加密，私钥进行解密，其中公钥是任何人都可以得知的，而私钥是通信双方所独有的。非对称加密的运算速度慢，但是安全性好。最常见的公钥加密算法是 RSA，它使用两个大素数 p 和 q 来生成密钥，p 和 q 的值越大，破解的难度就越大，但是执行加密和解密的时间也就越长，RSA 实验室推荐 p 和 q 的乘积为 1024 的数量级。

在使用对称密钥时，由于双方使用同样的密钥，因此在通信信道上可以进行**一对一的双向保密通信**，每一方既可以用该密钥加密明文并发送给对方，也可以接收密文用同一密钥解密。这种保密通信仅限于持有此密钥的双方。但在使用公开密钥时，在通信信道上可以是**多对一的单向保密通信**，可以同时有很多客户利用公钥对自己的报文加密后发送给某个服务器，服务器利用其私钥可以对收到的密文一一解密，但如果是反方向则是行不通的，例如在网购时很多客户都向同一个网站发送各自的信用卡信息。

数字签名

数字签名的作用

① **报文鉴别**：接收者能够核实发送者对报文的签名，也就是说接收者可以确认报文发送方的身份。

② **报文完整性**：接收者可以确信收到的数据和发送者发送的完全一样并且没有被篡改过。

③ **不可否认**：发送者事后不能抵赖对报文的签名。

实现原理

有多种实现数字签名的方法，但采用公钥算法要比采用对称密钥算法更容易实现。为了进行签名，会首先用私钥对报文进行 D 运算得到某种不可读的密文，为了核实签名，接收方会利用发送方的公钥进行 E 运算还原出明文。任何人都可以用发送方的公钥还原出明文，因此这种通信方式并不是为了保密，而是为了进行签名和核实签名，即确认发送方的身份。

数字签名实现报文鉴别：除了发送方之外没有人持有其私钥，因此无法产生发送方才能产生的密文。

数字签名保证报文完整性：如果其他人篡改过密文，那么解密出的明文就会不可读，就知道收到的报文被篡改过。

数字签名保证不可否认：如果发送方抵赖发送过报文，接收方可以把初始报文和密文发送给公证的第三者，第三者通过公钥很容易证实发送方确实发送过报文。

公钥认证

攻击者可能会发送使用自己私钥加密的密文和自己的公钥来伪造发送方的身份，该问题通过 CA 解决，发送方在发送数据时也会发送 CA 签署的证书，接收方会利用 CA 的公钥来核对发送方证书的合法性并提取发送方的公钥。

CA 即认证中心，将公钥与特定的实体绑定， 它的职责就是使识别和发行证书合法化。CA 主要有两个作用：① CA 认证一个实体（一个人、一台路由器等）的真实身份。② 一旦 CA 认证了某个实体的身份，CA 会生成一个将其身份和实体的公钥绑定起来的证书，这个证书包含了这个公钥和公钥所有者全局唯一的身份标识信息（例如一个人的名字或一个 IP 地址），由 CA 对这个证书进行数字签名。

报文鉴别

报文鉴别就是鉴别收到的报文确实是发送方发送的，而不是别人伪造或篡改的。数字签名就可以实现报文鉴别，但是数字签名有一个很大的缺点，就是对较长的报文进行数字签名时会使计算机增加非常大的负担，因为需要较长时间的运算。有一种相对简单的报文鉴别方式，就是密码散列函数。在密码学中的散列函数称为密码散列函数，最重要的特点就是：要找到两个不同的报文，它们具有相同的密码散列函数输出，在计算上是不可行的，也就是说密码散列函数实际上是一种单向函数。

使用散列函数进行报文鉴别的原理

通信双方需要共享一个密钥 k，发送方生成报文 m，用 k 级联 m 生成 m+k，并使用 SHA-1 或 MD5 这样的散列函数计算 m+k 的散列值 h，这个散列值就被称为报文鉴别码 MAC。发送方会将 MAC 附加到报文 m 上，生成一个扩展报文，并将该扩展报文发送给接收方。接收方接到扩展报文后，由于知道共享密钥 k，因此可以计算出报文鉴别码，如果计算出的报文鉴别码和 h 相等就可以得出一切正常的结论。

除了共享密钥，还可以使用公钥加密。发送方可以利用自己的私钥进行散列运算，接收方利用发送方的公钥进行还原。这种方法得到的扩展报文是不为伪造且不可否认的，因为攻击者没有发送方的私钥，无法伪造出发送方发出的报文。

端点鉴别

端点鉴别就是一个实体经过计算机网络向另一个实体证明其身份的过程，实体可以是一个人也可以是一个进程。端点鉴别主要通过鉴别协议 ap 来实现，鉴别协议通常在两个通信实体运行其他协议之前运行。鉴别协议首先建立相互满意的各方标识，仅当鉴别完成之后各方才继续下面的工作。

鉴别协议 ap1.0

发送方直接发送一个报文说明自己的身份，缺陷很大，攻击者可以任意伪造。

鉴别协议 ap2.0

发送方有一个总是用于通信的 IP 地址，接收方可以验证携带鉴别报文的 IP 数据报的源地址和发送方经常使用的 IP 地址是否匹配来进行鉴别。但存在 IP 欺骗的可能性，攻击者也可以伪造源 IP 地址。

鉴别协议 ap3.0

进行鉴别的经典方法是使用秘密口令，口令是鉴别者和被鉴别者之间的一个共享秘密。接收方会要求发送方提供口令来进行验证，但这种方式依旧不是安全的，因为攻击者可能会通过嗅探获得发送方的口令。一种改进想法是对口令进行加密，防止攻击者获得口令，但是这不能解决重放攻击，攻击者可以获取加密口令并不断重放。

鉴别协议 ap4.0

重放攻击主要是由于接收方并不知道此时发送方是否还是活跃的，ap 4.0 主要通过一个**不重数**来防止重放攻击。不重数就是在一个协议的生存期中只会使用一次的数，是一个不重复使用的大随机数，一旦某协议使用了一个不重数，就永远不会再使用那个数了。接收方会向发送方发送一个不重数，发送方将其加密后发回给接收方，接收方通过验证这个数字来判断发送方是否是活跃的。

安全协议

网络层

网络层的安全协议是 IPsec，它并不是一个单一的协议，而是能够为两个网络实体之间的 IP 数据报提供通信安全的协议族。IPsec 并没有限定用户必须使用的加密和鉴定算法，允许通信双方选择合适的算法和参数，为保证互操作性而包含了一套加密算法，要求所有 IPsec 的实现都必须使用。许多机构都使用 IPsec 来保证虚拟专用网 VPN 的安全性。

IPsec 协议族中有两个主要协议：**鉴别首部 AH 协议**和**封装安全有效载荷 ESP 协议**。当某源 IPsec 实体向一个目的实体发发送安全数据报时，它可以使用 AH 或 ESP 协议来实现。AH 提供源鉴别和数据完整性服务，而 ESP 除了这两种服务外还可以提供机密性服务，因此使用要比 AH 广泛许多。使用 AH 或 ESP 的 IP 数据报称为 IP 安全数据报，它可以在两台主机、两台路由器或一台主机和一个路由器之间发送。IP 安全数据报有两种工作方式：① 运输方式，在整个运输层报文段的前后分别加上控制信息再加上 IP 首部，构成 IP 安全数据报。② 隧道方式，在原始的 IP 数据报的前后分别加上控制信息，构成 IP 安全数据报，这种方式使用较多。

运输层

运输层的安全协议主要是 SSL 安全套节字层和 TLS 运输层安全，TLS 是 SSL3.0 的修改版本。SSL 主要作用在端系统的 HTTP 和运输层之间，在 TCP 上建立起一个安全通道，为 TCP 传输的应用层数据提供安全保障。应用层使用 SSL 最多的就是 HTTP，但 SSL 并不是只用于 HTTP，而是可以用于任何应用层的协议，例如 SSL 也可以用于邮件存取的鉴别和数据加密。

SSL 提供的安全性服务包括三种：① **服务器鉴别**，允许用户证实服务器的身份，支持 SSL 的客户端通过验证来自服务器的证书来鉴别服务器的身份并取得服务器的公钥。② **客户鉴别**，SSL 可选的安全服务，允许服务器验证用户的身份。③ **加密的 SSL 会话**，对客户和服务器发送的所有报文进行加密，并检测报文是否被篡改。

防火墙和入侵检测

在计算机网络中，当通信流量进入/离开网络时要执行安全检查、做记录、丢弃或转发，这些工作都有防火墙和入侵检测系统来完成。

防火墙

防火墙是一个硬件和软件的结合体，它将一个机构的内部网络与整个因特网隔离开，内部的属于可信网络，外部的属于不可信网络，允许一些数据分组通过而阻止另一些。它属于一种访问控制技术，通过严格控制进出网络的分组来禁止任何不必要的通信，从而减少潜在侵入的发送，从外部到内部和从内部到外部的所有流量都必须经过防火墙，只有被授权的流量才允许通过，授权与否由本地的安全策略定义，防火墙可以限制对授权流量的访问。

防火墙分为三种：① 基于**分组过滤**，分组过滤器独立地检查每个数据报，然后基于管理员特定的规则决定该数据报应当允许通过还是丢弃，过滤因素通常包括 IP 源或目的地址、TCP 或 UDP 地源和目的端口、IP 数据报中的协议字段类型等。② 基于**状态过滤**，利用一张连接表来实际地跟踪 TCP 连接，并使用跟踪信息做出过滤决定。③ **应用程序网关**，这是一个应用程序特定的服务器，所有应用程序数据都必须通过它，每种程序都需要一个不同的应用网关。

入侵检测系统 IDS

防火墙不可能阻止所有入侵行为，入侵检测系统作为第二道防线，通过对进入网络的分组进行深度分析与检测发现异常网络行为，并进行报警以便进一步处理。IDS 可以用于检测多种攻击，包括网络映射、端口扫描、DoS 带宽洪泛攻击、病毒和蠕虫等。一个机构可以在它的机构网络中部署一个或多个 IDS 传感器，由于 IDS 不仅需要做深度分组检查，还必须要把每个过往的分组与数以万计的特征进行比较，因此会导致很大的处理量，所以一般都需要多个 IDS 传感器。IDS 系统可以大致分为基于特征的系统和基于异常的系统。

基于特征的 IDS 维护了一个范围广泛的攻击特征数据库，每个特征是一个与入侵活动相关联的规则集，基于特征的 IDS 嗅探通过它的每个分组，将分组中的数据与数据库中的特征进行比较，如果匹配将产生一个警告，该警告能够发送一个电子邮件报文给网络管理员或者网络管理系统。但是基于特征的 IDS 无法应对新型攻击，并且即使与特征匹配时也可能不是一个攻击而因此产生了一个虚假警告。

基于异常的 IDS 会观察正常运行的流量，并生成一个流量概况文件。它会寻找统计上不寻常的分组流，例如 ICMP 分组不寻常的百分比或端口扫描指数性突然增长。基于异常的 IDS 最大的特点就是不依赖现有攻击的以前只是，另一方面区分正常流量和统计异常流量也是一个挑战。至今大多数部署的 IDS 主要是基于特征的。

操作系统

进程

多道程序环境下允许多个程序并发执行，进程就是为了更好地描述和控制程序的并发执行，实现操作系统的并发性和共享性。

进程就是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。系统资源指的是处理机、存储器和其他设备服务于某个进程的时间，例如把处理机资源理解为处理机的时间片才是准确的。因为进程是这些资源分配和调度的独立单位，这就决定了进程一定是一个动态的、过程性的概念。

结构

①进程控制块PCB：进程实体的一部分，进程存在的唯一标识，包括进程描述信息、控制和管理信息、资源分配清单和处理机相关信息。

②程序段：就被进程调度程序调度到CPU执行的程序代码段。

③数据段：进程对应的程序加工处理的原始数据，也可以是程序执行时产生的中间或最终结果。

特征

①动态性 进程是一次程序的执行，具有一定的生命周期，是动态地产生、变化和消亡的。动态性是进程最基本的特征。

②并发性 指多个进程同时存在于内存中，能在一段时间内同时运行。并发性是进程的重要特征，也是操作系统的重要特征。进入进程的目的就是为了使程序能与其他进程的程序并发执行，提高资源利用率。

③独立性 指进程实体是一个能独立运行、独立获得自由和独立接受调度的基本单位。

④异步性 由于进程的相互制约，会使进程具有执行的间断性，即进程按各自独立的，不可预知的速度向前推进。

⑤结构性 每个进程都配置有一个进程控制块PCB对其进行描述，从结构上看进程实体是由程序段、数据段和PCB组成的。

进程的状态和转换

①运行态 进程正在处理机上运行

②就绪态 进程已处于准备运行的状态，获得了除处理机外的一切资源

③阻塞态 进程正在等待某一事件而暂停运行，如等待某资源可用或等待输入/输出流

④创建态 进程正在被创建，尚未转到就绪态

⑤结束态 进程正从系统中消失，可能是正常结束或其他原因中断退出

就绪->运行：处于就绪状态的进程被调度后，获得处理机资源（分派处理机时间片）

运行->就绪：处于运行态的进程在时间片用完后，不得不让出处理机。在可剥夺的操作系统中，当有更高优先级的进程就绪时，调度程序将正在执行的进程转为就绪态，让更高优先级的进程执行。

运行->阻塞：进程请求某一资源的使用和分配或等待某事件的发生（如IO完成），进程以系统调用的形式请求操作系统提供服务。

阻塞->就绪：进程等待的事件到来时，如IO结束或中断结束时，中断处理程序必须把相应进程的状态由阻塞转为就绪态。

进程控制

进程控制的主要功能是对系统中的所有进程实施有效的管理，它具有创建新进程、撤销已有进程、实现进程状态转换等功能。

进程创建

允许一个进程创建另一个进程，创建者为父进程，被创建者为子进程。子进程可以继承父进程所拥有的资源，当子进程被撤销时，应将父进程的资源归还。撤销父进程时，必须同时撤销所有子进程。

①为新进程分配一个唯一的进程标识号，并申请一个空白PCB。②为进程分配资源，为新进程的程序和数据分配必要内存空间。若资源不足不会创建失败而是进入阻塞态。③初始化PCB，包括标志信息，处理机状态信息，进程优先级等。④若进程就绪队列未满，就将新进程插入就绪队列等待被调度。

进程终止

正常结束，表示进程任务已经完成并准备退出运行。异常结束，表示进程在运行时发生了某种异常，使程序无法继续运行，例如非法指令，IO故障等。外界干预，指进程因为外界请求而终止，例如操作系统干预或父进程请求终止等。

①根据被终止进程的标识符，检索PCB，读出该进程的状态。②若处于执行状态，终止执行，将处理机资源分配给其他进程。③若进程还有子进程，应将所有子进程终止。④将该进程的全部资源归还给父进程或操作系统。⑤将PCB从所在队列删除。

进程阻塞

①找到将要被阻塞进程的PCB。②如果为运行态，保护现场转为阻塞态，停止运行。③把PCB插入相应事件的等待队列。

进程唤醒

①在该事件的等待队列中找到进程对应的PCB。②将其从等待队列中移除，设置状态为就绪态。③将PCB插入就绪队列，等待调度程序调度。

进程切换

①保存处理机上下文，包括程序计数器和其他寄存器。②更新PCB信息。③把进程的PCB移入相应的队列。④选择另一个进程执行并更新其PCB。⑤更新内存管理的数据结构。⑥恢复处理机上下文。

进程通信

①共享存储：在通信的进程之间存在一块可以直接访问的共享空间，共享存储分为两种：低级的共享基于数据结构，高级的共享基于存储区。操作系统只负责为通信进程提供可共享的存储空间和同步互斥工具，数据交换由用户自己安排读写指令完成。

②消息传递：进程间的数据交换以格式化的消息为单位，进程提供系统提供的发送消息和接收消息两个原语进行数据交换。消息传递分为：直接通信方式，把消息挂在接收进程的消息缓存队列上。间接通信方式，发送进程把消息发送到某个中间实体，中间实体一般称作信箱，相应的通信系统为电子邮件系统。

③管道通信：消息传递的一种特殊方式，管道就是连接一个读进程和一个写进程来实现它们通信的一个共享文件。管道可以理解为共享存储的优化和发展，管道通信中存储空间优化为缓冲区，缓冲区只允许一边写入另一边读出，只要缓冲区有数据进程就能从缓冲区读出，只要有数据写进程就不会往缓冲区写数据，因此管道通信是半双工通信。

线程

引入进程的目的是为了多道程序更好的并发执行，提高资源利用率和吞吐量；引入线程的目的是为了减少程序在并发执行时的时空开销，提高操作系统的并发性能。

线程就是一种轻量级的进程，是一个基本的CPU执行单位，也是程序执行流的最小单元，由线程ID、程序计数器、寄存器集合和堆栈组成。线程是进程中的一个实体，是操作系统独立调度和分配的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它与同一进程下的其他线程共享进程的全部资源。

线程和进程的区别

①调度：进程是拥有资源的基本单位，而线程是独立调度的基本单位。在同一进程中，线程的切换不会引起进程的切换。在不同进程中线程的切换会引起进程切换。

②拥有资源：不管是传统操作系统还是有线程的操作系统，进程都是拥有资源的基本单位，而线程不拥有系统资源，只有一点运行中必不可少的资源。如果线程也是拥有资源的单位，那么切换线程就需要较大的时空开销，它的引入就没有意义。

③系统开销：创建和撤销进程涉及资源的分配和回收，操作系统的开销远大于创建或撤销线程的开销。进程切换也需要涉及CPU环境的保存和新调度到进程CPU环境的设置，但线程切换只需要保存和设置少量的寄存器容量，开销很小。

④地址空间：进程的地址空间之间互相独立，同一进程的各个线程共享进程的资源，进程内的线程对其他进程不可见。

⑤通信：进程间通信需要同步和互斥手段的辅助，保证数据一致性。线程可以直接读写进程数据段（全局变量）来进行通信。

线程的实现方式

①用户级线程：有关线程管理的所有工作都由应用程序完成，内核意识不到线程的存在。

②内核级线程：线程管理的所有工作都由内核完成，应用程序没有进行线程管理的代码，只有一个到内核级线程的编程接口。

死锁

死锁就是指多个进程因为互相竞争资源而陷入的一种僵局，如果没有外力的作用，这些进程都无法继续向前推进。

死锁的原因包含了：

①不可剥夺资源数量的不足，如果是可剥夺资源是不会造成死锁的。

②进程的推进顺序非法，进程请求和释放资源的顺序不当，例如进程P1和P2分别占用资源R1和R2，而此时P1和P2又分别申请资源R2和R1。

③信号量的使用不当，彼此等待对方的消息。

死锁有四个必要条件：

①互斥条件，进程对资源的占用具有排他性控制，如果进程请求的资源已被占用，请求就会被阻塞。

②不可剥夺条件，当一个资源没有被使用完成前是不能被其他进程强行获取的，只有占用它的进程主动释放才可以。

③请求和保持条件，一个进程已经占有了某个资源，又要请求其他资源，而该资源被其他进程占用，请求被阻塞，但进程也不会释放自己已经占有的资源。

④循环等待条件，存在一个进程资源的循环等待链，链中每个进程已经占有的资源同时是其他进程请求的资源。

预防

事先预防，实现起来比较简单，但是条件严格，效率很低。

①破坏互斥条件，系统中的所有资源都允许共享，但是有的资源不能同时访问，不太现实。

②破坏不可剥夺条件，允许剥夺其他进程已经占有的资源，可能会造成前段工作的失效，如果频繁发送就会增加系统开销，严重降低系统的吞吐量。

③破坏请求和保持条件，采用预先资源分配法，一次性分配进程需要的所有资源，缺点是会严重浪费系统资源。

④破坏循环等待条件，采用顺序资源分配法， 缺点是会造成编程不便。

避免

同样也是事先预防，不同的是动态地根据情况来避免死锁，性能比较好。

①系统安全状态，不安全的系统可能会导致死锁，安全的系统状态不会导致死锁，如果资源分配不会进入不安全的系统状态就给进程分配资源。

②银行家算法，把操作系统视为银行家，操作系统管理的资源视为资金，进程向操作系统申请资源相当于贷款。采用预先资源分配策略，主要的数据结构是可利用的资源向量，分配矩阵，需求矩阵，最大需求矩阵。

检测

画出资源分配图，圆圈表示进程，框表示一类资源。进程到资源是请求边，资源到进程是分配边。然后利用死锁定理来简化资源分配图，如果S不可被完全简化那么代表是一个死锁。

解除

如果没有采取死锁的预防和避免，就要采用死锁的检测和解除。

①资源剥夺法：挂起某些死锁进程并剥夺其资源。②撤销进程法：撤销一个甚至全部死锁进程并剥夺其资源。③进程回退法：让一个或多个进程回到不至于造成死锁的状态。

Java 基础

语言特性

Java 语言的优点

① 平台无关，摆脱硬件束缚，"一次编写，到处运行"。

② 安全的内存管理和访问机制，避免大部分内存泄漏和指针越界。

③ 热点代码检测和运行时编译优化，程序随运行时长获得更高性能。

④ 完善的应用程序接口，支持第三方类库。

Java 平台无关

JVM： 编译器生成与计算机体系结构无关的字节码，字节码文件不仅可在任何机器解释执行，还可动态转换成本地机器码，转换由 JVM 实现。JVM 是平台相关的，屏蔽了不同操作系统的差异。

语言规范： 基本数据类型大小有明确规定，如 int 永远 32 位，而 C/C++ 可能是 16 位、32 位，或编译器开发商指定的其他大小。数值类型有固定字节数，二进制数据以固定格式存储，字符串用标准 Unicode 格式。

JDK 和 JRE

JDK： Java Development Kit，开发工具包。提供了编译运行 Java 程序的各种工具，包括编译器、JRE 及常用类库，是 JAVA 核心。

JRE： Java Runtime Environment，运行时环境，运行 Java 程序的必要环境，包括 JVM、核心类库、核心配置工具。

值调用和引用调用

按值调用指方法接收调用者提供的值，**按引用调用**指方法接收调用者提供的变量地址。

Java 总是按值调用，方法得到的是所有参数值的副本，传递对象时实际上方法接收的是对象引用的副本。

方法不能修改基本数据类型的参数，如果传递了一个 int 值 ，改变值不会影响实参，因为改变的是值的一个副本。

可以改变对象参数的状态，但不能让对象参数引用一个新的对象。如果传递了一个 int 数组，改变数组的内容会影响实参，而改变这个参数的引用并不会让实参引用新的数组对象。

浅拷贝和深拷贝

浅拷贝：只复制当前对象的基本数据类型及引用变量，没有复制引用变量指向的实际对象。修改克隆对象的引用数据类型属性可能影响原对象，不安全。

深拷贝：完全拷贝基本数据类型和引用数据类型，修改克隆对象不会影响原对象，是安全的。

反射

在运行状态中，对于任意一个类都能知道它的所有属性和方法，对于任意一个对象都能调用它的任意方法和属性，这种动态获取信息及调用对象方法的功能称为反射。

反射的缺点是破坏了封装性以及泛型约束。

Class 类

在程序运行期间，Java 运行时系统为所有对象维护一个运行时类型标识，这个信息会跟踪每个对象所属的类，虚拟机利用运行时类型信息选择要执行的正确方法，保存这些信息的类就是 Class，这是一个泛型类。

获取 Class 对象：① `类名.class` 。② 对象的 `getClass` 方法。③ `Class.forName(类的全限定名)`。

注解

注解是一种标记，使类或接口附加额外信息，帮助编译器和 JVM 完成一些特定功能，例如 `@Override` 标识一个方法是重写方法。

元注解是自定义注解的注解，例如：

`@Target`：约束作用位置，值是 ElementType 枚举常量，包括 METHOD 方法、VARIABLE 变量、TYPE 类/接口、PARAMETER 方法参数、CONSTRUCTORS 构造方法和 LOACL_VARIABLE 局部变量等。

`@Rentention`：约束生命周期，值是 RetentionPolicy 枚举常量，包括 SOURCE 源码、CLASS 字节码和 RUNTIME 运行时。

`@Documented`：表明这个注解应该被 javadoc 记录。

泛型

泛型本质是参数化类型，解决不确定对象具体类型的问题。泛型在定义处只具备执行 Object 方法的能力。

泛型的好处：① 类型安全，放置什么类型，取出来就是什么类型，不存在 ClassCastException 类型转换异常。② 提升可读性，编码阶段就显式知道泛型集合、泛型方法等处理的对象类型。③ 代码重用，合并了同类型的处理代码。

泛型擦除

泛型用于编译阶段，编译后的字节码文件不包含泛型类型信息，因为虚拟机没有泛型类型对象，所有对象都属于普通类。例如定义 `List<Object>` 或 `List<String>`，在编译后都会变成 `List` 。

定义一个泛型类型，会自动提供一个对应原始类型，类型变量会被擦除。如果没有限定类型就会替换为 Object，如果有限定类型就会替换为第一个限定类型，例如 `<T extends A & B>` 会使用 A 类型替换 T。

JDK8 新特性

lambda 表达式：允许把函数作为参数传递到方法，简化匿名内部类代码。

函数式接口：使用 `@FunctionalInterface` 标识，有且仅有一个抽象方法，可被隐式转换为 lambda 表达式。

方法引用：可以引用已有类或对象的方法和构造方法，进一步简化 lambda 表达式。

接口：接口可以定义 `default` 修饰的默认方法，降低了接口升级的复杂性，还可以定义静态方法。

注解：引入重复注解机制，相同注解在同地方可以声明多次。注解作用范围也进行了扩展，可作用于局部变量、泛型、方法异常等。

类型推测：加强了类型推测机制，使代码更加简洁。

Optional 类：处理空指针异常，提高代码可读性。

Stream 类：引入函数式编程风格，提供了很多功能，使代码更加简洁。方法包括 `forEach` 遍历、`count` 统计个数、`filter` 按条件过滤、`limit` 取前 n 个元素、`skip` 跳过前 n 个元素、`map` 映射加工、`concat` 合并 stream 流等。

日期：增强了日期和时间 API，新的 java.time 包主要包含了处理日期、时间、日期/时间、时区、时刻和时钟等操作。

JavaScript：提供了一个新的 JavaScript 引擎，允许在 JVM上运行特定 JavaScript 应用。

异常

所有异常都是 Throwable 的子类，分为 Error 和 Exception。

Error 是 Java 运行时系统的内部错误和资源耗尽错误，例如 StackOverFlowError 和 OutOfMemoryError，这种异常程序无法处理。

Exception 分为受检异常和非受检异常，受检异常需要在代码中显式处理，否则会编译出错，非受检异常是运行时异常，继承自 RuntimeException。

受检异常：① 无能为力型，如字段超长导致的 SQLException。② 力所能及型，如未授权异常 UnAuthorizedException，程序可跳转权限申请页面。常见受检异常还有 FileNotFoundException、ClassNotFoundException、IOException等。

非受检异常：① 可预测异常，例如 IndexOutOfBoundsException、NullPointerException、ClassCastException 等，这类异常应该提前处理。② 需捕捉异常，例如进行 RPC 调用时的远程服务超时，这类异常客户端必须显式处理。③ 可透出异常，指框架或系统产生的且会自行处理的异常，例如 Spring 的 NoSuchRequestHandingMethodException，Spring 会自动完成异常处理，将异常自动映射到合适的状态码。

数据类型

基本数据类型

数据类型	内存大小	默认值	取值范围
byte	1 B	(byte)0	-128 ~ 127
short	2 B	(short)0	-2^{15} ~ $2^{15}-1$
int	4 B	0	-2^{31} ~ $2^{31}-1$
long	8 B	0L	-2^{63} ~ $2^{63}-1$
float	4 B	0.0F	±3.4E+38（有效位数 6~7 位）
double	8 B	0.0D	±1.7E+308（有效位数 15 位）
char	英文 1B，中文 UTF-8 占 3B，GBK 占 2B。	'\u0000'	'\u0000' ~ '\uFFFF'
boolean	单个变量 4B / 数组 1B	false	true、false

JVM 没有 boolean 赋值的专用字节码指令，`boolean f = false` 就是使用 ICONST_0 即常数 0 赋值。单个 boolean 变量用 int 代替，boolean 数组会编码成 byte 数组。

自动装箱/拆箱

每个基本数据类型都对应一个包装类，除了 int 和 char 对应 Integer 和 Character 外，其余基本数据类型的包装类都是首字母大写即可。

自动装箱：将基本数据类型包装为一个包装类对象，例如向一个泛型为 Integer 的集合添加 int 元素。

自动拆箱：将一个包装类对象转换为一个基本数据类型，例如将一个包装类对象赋值给一个基本数据类型的变量。

比较两个包装类数值要用 `equals`，而不能用 `==`。

String 不可变性

String 类和其存储数据的成员变量 value 字节数组都是 final 修饰的。对一个 String 对象的任何修改实际上都是创建一个新 String 对象，再引用该对象。只是修改 String 变量引用的对象，没有修改原 String 对象的内容。

字符串拼接方式

① 直接用 `+`，底层用 StringBuilder 实现。只适用小数量，如果在循环中使用 `+` 拼接，相当于不断创建新的 StringBuilder 对象再转换成 String 对象，效率极差。

② 使用 String 的 concat 方法，该方法中使用 `Arrays.copyOf` 创建一个新的字符数组 buf 并将当前字符串 value 数组的值拷贝到 buf 中，buf 长度 = 当前字符串长度 + 拼接字符串长度。之后调用 `getChars` 方法使用 `System.arraycopy` 将拼接字符串的值也拷贝到 buf 数组，最后用 buf 作为构造参数 new 一个新的 String 对象返回。效率稍高于直接使用 `+`。

③ 使用 StringBuilder 或 StringBuffer，两者的 `append` 方法都继承自 AbstractStringBuilder，该方法首先使用 `Arrays.copyOf` 确定新的字符数组容量，再调用 `getChars` 方法使用 `System.arraycopy` 将新的值追加到数组中。StringBuilder 是 JDK5 引入的，效率高但线程不安全。StringBuffer 使用 synchronized 保证线程安全。

面向对象

面向对象

面向过程让计算机有步骤地顺序做一件事，是过程化思维，使用面向过程语言开发大型项目，软件复用和维护存在很大问题，模块之间耦合严重。面向对象相对面向过程更适合解决规模较大的问题，可以拆解问题复杂度，对现实事物进行抽象并映射为开发对象，更接近人的思维。

例如开门这个动作，面向过程是 `open(Door door)`，动宾结构，door 作为操作对象的参数传入方法，方法内定义开门的具体步骤。面向对象的方式首先会定义一个类 Door，抽象出门的属性（如尺寸、颜色）和行为（如 open 和 close），主谓结构。

面向过程代码松散，强调流程化解决问题。面向对象代码强调高内聚、低耦合，先抽象模型定义共性行为，再解决实际问题。

封装是对象功能内聚的表现形式，在抽象基础上决定信息是否公开及公开等级，核心问题是以什么方式暴漏哪些信息。主要任务是对属性、数据、敏感行为实现隐藏，对属性的访问和修改必须通过公共接口实现。封装使对象关系变得简单，降低了代码耦合度，方便维护。

迪米特原则就是对封装的要求，即 A 模块使用 B 模块的某接口行为，对 B 模块中除此行为外的其他信息知道得应尽可能少。不直接对 public 属性进行读取和修改而使用 getter/setter 方法是因为假设想在修改属性时进行权限控制、日志记录等操作，在直接访问属性的情况下无法实现。如果将 public 的属性和行为修改为 private 一般依赖模块都会报错，因此不知道使用哪种权限时应优先使用 private。

继承用来扩展一个类，子类可继承父类的部分属性和行为使模块具有复用性。继承是"is-a"关系，可使用里氏替换原则判断是否满足"is-a"关系，即任何父类出现的地方子类都可以出现。如果父类引用直接使用子类引用来代替且可以正确编译并执行，输出结果符合子类场景预期，那么说明两个类符合里氏替换原则。

多态以封装和继承为基础，根据运行时对象实际类型使同一行为具有不同表现形式。多态指在编译层面无法确定最终调用的方法体，在运行期由 JVM 动态绑定，调用合适的重写方法。由于重载属于静态绑定，本质上重载结果是完全不同的方法，因此多态一般专指重写。

重载和重写

重载指方法名称相同，但参数类型个数不同，是行为水平方向不同实现。对编译器来说，方法名称和参数列表组成了一个唯一键，称为方法签名，JVM 通过方法签名决定调用哪种重载方法。不管继承关系如何复杂，重载在编译时可以根据规则知道调用哪种目标方法，因此属于静态绑定。

JVM 在重载方法中选择合适方法的顺序：① 精确匹配。② 基本数据类型自动转换成更大表示范围。③ 自动拆箱与装箱。④ 子类向上转型。⑤ 可变参数。

重写指子类实现接口或继承父类时，保持方法签名完全相同，实现不同方法体，是行为垂直方向不同实现。

元空间有一个方法表保存方法信息，如果子类重写了父类的方法，则方法表中的方法引用会指向子类实现。父类引用执行子类方法时无法调用子类存在而父类不存在的方法。

重写方法访问权限不能变小，返回类型和抛出的异常类型不能变大，必须加 `@Override` 。

类之间的关系

类关系	描述	权力强侧	举例
继承	父子类之间的关系：is-a	父类	小狗继承于动物
实现	接口和实现类之间的关系：can-do	接口	小狗实现了狗叫接口
组合	比聚合更强的关系：contains-a	整体	头是身体的一部分
聚合	暂时组装的关系：has-a	组装方	小狗和绳子是暂时的聚合关系
依赖	一个类用到另一个：depends-a	被依赖方	人养小狗，人依赖于小狗
关联	平等的使用关系：links-a	平等	人使用卡消费，卡可以提取人的信息

Object 类

equals：检测对象是否相等，默认使用 `==` 比较对象引用，可以重写 equals 方法自定义比较规则。equals 方法规范：自反性、对称性、传递性、一致性、对于任何非空引用 x，`x.equals(null)` 返回 false。

hashCode：散列码是由对象导出的一个整型值，没有规律，每个对象都有默认散列码，值由对象存储地址得出。字符串散列码由内容导出，值可能相同。为了在集合中正确使用，一般需要同时重写 equals 和 hashCode，要求 equals 相同 hashCode 必须相同，hashCode 相同 equals 未必相同，因此 hashCode 是对象相等的必要不充分条件。

toString：打印对象时默认的方法，如果没有重写打印的是表示对象值的一个字符串。

clone：clone 方法声明为 protected，类只能通过该方法克隆它自己的对象，如果希望其他类也能调用该方法必须定义该方法为 public。如果一个对象的类没有实现 Cloneable 接口，该对象调用 clone 方法会抛出一个 CloneNotSupport 异常。默认的 clone 方法是浅拷贝，一般重写 clone 方法需要实现 Cloneable 接口并指定访问修饰符为 public。

finalize：确定一个对象死亡至少要经过两次标记，如果对象在可达性分析后发现没有与 GC Roots 连接的引用链会被第一次标记，随后进行一次筛选，条件是对象是否有必要执行 finalize 方法。假如对象没有重写该方法或方法已被虚拟机调用，都视为没有必要执行。如果有必要执行，对象会被放置在 F-Queue 队列，由一条低调度优先级的 Finalizer 线程去执行。虚拟机会触发该方法但不保证会结束，这是为了防止某个对象的 finalize 方法执行缓慢或发生死循环。只要对象在 finalize 方法中重新与引用链上的对象建立关联就会在第二次标记时被移出回收集合。由于运行代价高昂且无法保证调用顺序，在 JDK 9 被标记为过时方法，并不适合释放资源。

getClass：返回包含对象信息的类对象。

wait / notify / notifyAll：阻塞或唤醒持有该对象锁的线程。

内部类

内部类可对同一包中其他类隐藏，内部类方法可以访问定义这个内部类的作用域中的数据，包括 private 数据。

内部类是一个编译器现象，与虚拟机无关。编译器会把内部类转换成常规的类文件，用 $ 分隔外部类名与内部类名，其中匿名内部类使用数字编号，虚拟机对此一无所知。

静态内部类： 属于外部类，只加载一次。作用域仅在包内，可通过 `外部类名.内部类名` 直接访问，类内只能访问外部类所有静态属性和方法。HashMap 的 Node 节点，ReentrantLock 中的 Sync 类，ArrayList 的 SubList 都是静态内部类。内部类中还可以定义内部类，如 ThreadLoacl 静态内部类 ThreadLoaclMap 中定义了内部类 Entry。

成员内部类： 属于外部类的每个对象，随对象一起加载。不可以定义静态成员和方法，可访问外部类的所有内容。

局部内部类： 定义在方法内，不能声明访问修饰符，只能定义实例成员变量和实例方法，作用范围仅在声明类的代码块中。

匿名内部类： 只用一次的没有名字的类，可以简化代码，创建的对象类型相当于 new 的类的子类类型。用于实现事件监听和其他回调。

访问权限控制符

访问权限控制符	本类	包内	包外子类	任何地方
public	√	√	√	√
protected	√	√	√	×
无	√	√	×	×
private	√	×	×	×

接口和抽象类

接口和抽象类对实体类进行更高层次的抽象，仅定义公共行为和特征。

语法维度	抽象类	接口
成员变量	无特殊要求	默认 public static final 常量
构造方法	有构造方法，不能实例化	没有构造方法，不能实例化
方法	抽象类可以没有抽象方法，但有抽象方法一定是抽象类。	默认 public abstract，JDK8 支持默认/静态方法，JDK9 支持私有方法。
继承	单继承	多继承

抽象类体现 is-a 关系，接口体现 can-do 关系。与接口相比，抽象类通常是对同类事物相对具体的抽象。

抽象类是模板式设计，包含一组具体特征，例如某汽车，底盘、控制电路等是抽象出来的共同特征，但内饰、显示屏、座椅材质可以根据不同级别配置存在不同实现。

接口是契约式设计，是开放的，定义了方法名、参数、返回值、抛出的异常类型，谁都可以实现它，但必须遵守接口的约定。例如所有车辆都必须实现刹车这种强制规范。

接口是顶级类，抽象类在接口下面的第二层，对接口进行了组合，然后实现部分接口。当纠结定义接口和抽象类时，推荐定义为接口，遵循接口隔离原则，按维度划分成多个接口，再利用抽象类去实现这些，方便后续的扩展和重构。

例如 Plane 和 Bird 都有 fly 方法，应把 fly 定义为接口，而不是抽象类的抽象方法再继承，因为除了 fly 行为外 Plane 和 Bird 间很难再找到其他共同特征。

子类初始化顺序

① 父类静态代码块和静态变量。② 子类静态代码块和静态变量。③ 父类普通代码块和普通变量。④ 父类构造方法。⑤ 子类普通代码块和普通变量。⑥ 子类构造方法。

集合

ArrayList

ArrayList 是容量可变的非线程安全列表，使用数组实现，集合扩容时会创建更大的数组，把原有数组复制到新数组。支持对元素的快速随机访问，但插入与删除速度很慢。ArrayList 实现了 RandomAcess 标记接口，如果一个类实现了该接口，那么表示使用索引遍历比迭代器更快。

elementData是 ArrayList 的数据域，被 transient 修饰，序列化时会调用 writeObject 写入流，反序列化时调用 readObject 重新赋值到新对象的 elementData。原因是 elementData 容量通常大于实际存储元素的数量，所以只需发送真正有实际值的数组元素。

size 是当前实际大小，elementData 大小大于等于 size。

modCount 记录了 ArrayList 结构性变化的次数，继承自 AbstractList。所有涉及结构变化的方法都会增加该值。expectedModCount 是迭代器初始化时记录的 modCount 值，每次访问新元素时都会检查 modCount 和 expectedModCount 是否相等，不相等就会抛出异常。这种机制叫做 fail-fast，所有集合类都有这种机制。

LinkedList

LinkedList 本质是双向链表，与 ArrayList 相比插入和删除速度更快，但随机访问元素很慢。除继承 AbstractList 外还实现了 Deque 接口，这个接口具有队列和栈的性质。成员变量被 transient 修饰，原理和 ArrayList 类似。

LinkedList 包含三个重要的成员：size、first 和 last。size 是双向链表中节点的个数，first 和 last 分别指向首尾节点的引用。

LinkedList 的优点在于可以将零散的内存单元通过附加引用的方式关联起来，形成按链路顺序查找的线性结构，内存利用率较高。

Set

Set 不允许元素重复且无序，常用实现有 HashSet、LinkedHashSet 和 TreeSet。

HashSet 通过 HashMap 实现，HashMap 的 Key 即 HashSet 存储的元素，所有 Key 都使用相同的 Value ，一个名为 PRESENT 的 Object 类型常量。使用 Key 保证元素唯一性，但不保证有序性。由于 HashSet 是 HashMap 实现的，因此线程不安全。

HashSet 判断元素是否相同时，对于包装类型直接按值比较。对于引用类型先比较 hashCode 是否相同，不同则代表不是同一个对象，相同则继续比较 equals，都相同才是同一个对象。

LinkedHashSet 继承自 HashSet，通过 LinkedHashMap 实现，使用双向链表维护元素插入顺序。

TreeSet 通过 TreeMap 实现的，添加元素到集合时按照比较规则将其插入合适的位置，保证插入后的集合仍然有序。

TreeMap

TreeMap 基于红黑树实现，增删改查的平均和最差时间复杂度均为 $O(\log_n)$ ，最大特点是 Key 有序。Key 必须实现 Comparable 接口或提供的 Comparator 比较器，所以 Key 不允许为 null。

HashMap 依靠 `hashCode` 和 `equals` 去重，而 TreeMap 依靠 Comparable 或 Comparator。TreeMap 排序时，如果比较器不为空就会优先使用比较器的 `compare` 方法，否则使用 Key 实现的 Comparable 的 `compareTo` 方法，两者都不满足会抛出异常。

TreeMap 通过 `put` 和 `deleteEntry` 实现增加和删除树节点。插入新节点的规则有三个：① 需要调整的新节点总是红色的。② 如果插入新节点的父节点是黑色的，不需要调整。③ 如果插入新节点的父节点是红色的，由于红黑树不能出现相邻红色，进入循环判断，通过重新着色或左右旋转来调整。
TreeMap 的插入操作就是按照 Key 的对比往下遍历，大于节点值向右查找，小于向左查找，先按照二叉查找树的特性操作，后续会重新着色和旋转，保持红黑树的特性。

HashMap

JDK8 之前底层实现是数组 + 链表，JDK8 改为数组 + 链表/红黑树，节点类型从Entry 变更为 Node。主要成员变量包括存储数据的 table 数组、元素数量 size、加载因子 loadFactor。

table 数组记录 HashMap 的数据，每个下标对应一条链表，所有哈希冲突的数据都会被存放到同一条链表，Node/Entry 节点包含四个成员变量：key、value、next 指针和 hash 值。

HashMap 中数据以键值对的形式存在，键对应的 hash 值用来计算数组下标，如果两个元素 key 的 hash 值一样，就会发生哈希冲突，被放到同一个链表上，为使查询效率尽可能高，键的 hash 值要尽可能分散。

HashMap 默认初始化容量为 16，扩容容量必须是 2 的幂次方、最大容量为 1<< 30 、默认加载因子为 0.75。

JDK8 之前

hash：计算元素 key 的散列值

① 处理 String 类型时，调用 `stringHash32` 方法获取 hash 值。

② 处理其他类型数据时，提供一个相对于 HashMap 实例唯一不变的随机值 hashSeed 作为计算初始量。

③ 执行异或和无符号右移使 hash 值更加离散，减小哈希冲突概率。

indexFor：计算元素下标

将 hash 值和数组长度-1 进行与操作，保证结果不会超过 table 数组范围。

get：获取元素的 value 值

① 如果 key 为 null，调用 `getForNullKey` 方法，如果 size 为 0 表示链表为空，返回 null。如果 size 不为 0 说明存在链表，遍历 table[0] 链表，如果找到了 key 为 null 的节点则返回其 value，否则返回 null。

② 如果 key 为 不为 null，调用 `getEntry` 方法，如果 size 为 0 表示链表为空，返回 null 值。如果 size 不为 0，首先计算 key 的 hash 值，然后遍历该链表的所有节点，如果节点的 key 和 hash 值都和要查找的元素相同则返回其 Entry 节点。

③ 如果找到了对应的 Entry 节点，调用 `getValue` 方法获取其 value 并返回，否则返回 null。

put：添加元素

① 如果 key 为 null，直接存入 table[0]。

② 如果 key 不为 null，计算 key 的 hash 值。

③ 调用 `indexFor` 计算元素存放的下标 i。

④ 遍历 table[i] 对应的链表，如果 key 已存在，就更新 value 然后返回旧 value。

⑤ 如果 key 不存在，将 modCount 值加 1，使用 `addEntry` 方法增加一个节点并返回 null。

resize：扩容数组

① 如果当前容量达到了最大容量，将阈值设置为 Integer 最大值，之后扩容不再触发。

② 否则计算新的容量，将阈值设为 `newCapacity x loadFactor` 和 `最大容量 + 1` 的较小值。

③ 创建一个容量为 newCapacity 的 Entry 数组，调用 `transfer` 方法将旧数组的元素转移到新数组。

transfer：转移元素

① 遍历旧数组的所有元素，调用 `rehash` 方法判断是否需要哈希重构，如果需要就重新计算元素 key 的 hash 值。

② 调用 `indexFor` 方法计算元素存放的下标 i，利用头插法将旧数组的元素转移到新数组。

JDK8

hash：计算元素 key 的散列值

如果 key 为 null 返回 0，否则就将 key 的 `hashCode` 方法返回值高低16位异或，让尽可能多的位参与运算，让结果的 0 和 1 分布更加均匀，降低哈希冲突概率。

put：添加元素

① 调用 `putVal` 方法添加元素。

② 如果 table 为空或长度为 0 就进行扩容，否则计算元素下标位置，不存在就调用 `newNode` 创建一个节点。

③ 如果存在且是链表，如果首节点和待插入元素的 hash 和 key 都一样，更新节点的 value。

④ 如果首节点是 TreeNode 类型，调用 `putTreeVal` 方法增加一个树节点，每一次都比较插入节点和当前节点的大小，待插入节点小就往左子树查找，否则往右子树查找，找到空位后执行两个方法：`balanceInsert` 方法，插入节点并调整平衡、`moveRootToFront` 方法，由于调整平衡后根节点可能变化，需要重置根节点。

⑤ 如果都不满足，遍历链表，根据 hash 和 key 判断是否重复，决定更新 value 还是新增节点。如果遍历到了链表末尾则添加节点，如果达到建树阈值 7，还需要调用 `treeifyBin` 把链表重构为红黑树。

⑥ 存放元素后将 modCount 加 1，如果 `++size > threshold` ，调用 `resize` 扩容。

get ： 获取元素的 value 值

① 调用 `getNode` 方法获取 Node 节点，如果不是 null 就返回其 value 值，否则返回 null。

② `getNode` 方法中如果数组不为空且存在元素，先比较第一个节点和要查找元素的 hash 和 key ，如果都相同则直接返回。

③ 如果第二个节点是 TreeNode 类型则调用 `getTreeNode` 方法进行查找，否则遍历链表根据 hash 和 key 查找，如果没有找到就返回 null。

resize：扩容数组

重新规划长度和阈值，如果长度发生了变化，部分数据节点也要重新排列。

重新规划长度

① 如果当前容量 `oldCap > 0` 且达到最大容量，将阈值设为 Integer 最大值，return 终止扩容。

② 如果未达到最大容量，当 `oldCap << 1` 不超过最大容量就扩大为 2 倍。

③ 如果都不满足且当前扩容阈值 `oldThr > 0` ，使用当前扩容阈值作为新容量。

④ 否则将新容量置为默认初始容量 16，新扩容阈值置为 12。

重新排列数据节点

① 如果节点为 null 不进行处理。

② 如果节点不为 null 且没有next节点，那么通过节点的 hash 值和 `新容量-1` 进行与运算计算下标存入新的 table 数组。

③ 如果节点为 TreeNode 类型，调用 `split` 方法处理，如果节点数 hc 达到6 会调用 `untreeify` 方法转回链表。

④ 如果是链表节点，需要将链表拆分为 hash 值超出旧容量的链表和未超出容量的链表。对于 `hash & oldCap == 0` 的部分不需要做处理，否则需要放到新的下标位置上，新下标 = 旧下标 + 旧容量。

HashMap 线程不安全

JDK7 存在死循环和数据丢失问题。

数据丢失:

- **并发赋值被覆盖：** 在 `createEntry` 方法中，新添加的元素直接放在头部，使元素之后可以被更快访问，但如果两个线程同时执行到此处，会导致其中一个线程的赋值被覆盖。
- **已遍历区间新增元素丢失：** 当某个线程在 `transfer` 方法迁移时，其他线程新增的元素可能落在已遍历过的哈希槽上。遍历完成后，table 数组引用指向了 newTable，新增元素丢失。
- **新表被覆盖：** 如果 `resize` 完成，执行了 `table = newTable`，则后续元素就可以在新表上进行插入。但如果多线程同时 `resize`，每个线程都会 new 一个数组，这是线程内的局部对象，线程之间不可见。迁移完成后 `resize` 的线程会赋值给 table 线程共享变量，可能会覆盖其他线程的操作，在新表中插入的对象都会被丢弃。

死循环： 扩容时 `resize` 调用 `transfer` 使用头插法迁移元素，虽然 newTable 是局部变量，但原先 table 中的 Entry 链表是共享的，问题根源是 Entry 的 next 指针并发修改，某线程还没有将 table 设为 newTable 时用完了 CPU 时间片，导致数据丢失或死循环。

JDK8 在 `resize` 方法中完成扩容，并改用尾插法，不会产生死循环，但并发下仍可能丢失数据。可用 ConcurrentHashMap 或 `Collections.synchronizedMap` 包装成同步集合。

IO 流

同步/异步/阻塞/非阻塞 IO

同步和异步是通信机制，阻塞和非阻塞是调用状态。

同步 IO 是用户线程发起 IO 请求后需要等待或轮询内核 IO 操作完成后才能继续执行。异步 IO 是用户线程发起 IO 请求后可以继续执行，当内核 IO 操作完成后会通知用户线程，或调用用户线程注册的回调函数。

阻塞 IO 是 IO 操作需要彻底完成后才能返回用户空间 。非阻塞 IO 是 IO 操作调用后立即返回一个状态值，无需等 IO 操作彻底完成。

BIO

BIO 是同步阻塞式 IO，JDK1.4 之前的 IO 模型。服务器实现模式为一个连接请求对应一个线程，服务器需要为每一个客户端请求创建一个线程，如果这个连接不做任何事会造成不必要的线程开销。可以通过线程池改善，这种 IO 称为伪异步 IO。适用连接数目少且服务器资源多的场景。

NIO

NIO 是 JDK1.4 引入的同步非阻塞 IO。服务器实现模式为多个连接请求对应一个线程，客户端连接请求会注册到一个多路复用器 Selector ，Selector 轮询到连接有 IO 请求时才启动一个线程处理。适用连接数目多且连接时间短的场景。

同步是指线程还是要不断接收客户端连接并处理数据，非阻塞是指如果一个管道没有数据，不需要等待，可以轮询下一个管道。

核心组件：

- **Selector：** 多路复用器，轮询检查多个 Channel 的状态，判断注册事件是否发生，即判断 Channel 是否处于可读或可写状态。使用前需要将 Channel 注册到 Selector，注册后会得到一个 SelectionKey，通过 SelectionKey 获取 Channel 和 Selector 相关信息。
- **Channel：** 双向通道，替换了 BIO 中的 Stream 流，不能直接访问数据，要通过 Buffer 来读写数据，也可以和其他 Channel 交互。
- **Buffer：** 缓冲区，本质是一块可读写数据的内存，用来简化数据读写。Buffer 三个重要属性：position 下次读写数据的位置，limit 本次读写的极限位置，capacity 最大容量。
  - `flip` 将写转为读，底层实现原理把 position 置 0，并把 limit 设为当前的 position 值。
  - `clear` 将读转为写模式（用于读完全部数据的情况，把 position 置 0，limit 设为 capacity）。
  - `compact` 将读转为写模式（用于存在未读数据的情况，让 position 指向未读数据的下一个）。
  - 通道方向和 Buffer 方向相反，读数据相当于向 Buffer 写，写数据相当于从 Buffer 读。

  使用步骤：向 Buffer 写数据，调用 flip 方法转为读模式，从 Buffer 中读数据，调用 clear 或 compact 方法清空缓冲区。

AIO

AIO 是 JDK7 引入的异步非阻塞 IO。服务器实现模式为一个有效请求对应一个线程，客户端的 IO 请求都是由操作系统先完成 IO 操作后再通知服务器应用来直接使用准备好的数据。适用连接数目多且连接时间长的场景。

异步是指服务端线程接收到客户端管道后就交给底层处理IO通信，自己可以做其他事情，非阻塞是指客户端有数据才会处理，处理好再通知服务器。

实现方式包括通过 Future 的 `get` 方法进行阻塞式调用以及实现 CompletionHandler 接口，重写请求成功的回调方法 `completed` 和请求失败回调方法 `failed`。

---

java.io

主要分为字符流和字节流，字符流一般用于文本文件，字节流一般用于图像或其他文件。

字符流包括了字符输入流 Reader 和字符输出流 Writer，字节流包括了字节输入流 InputStream 和字节输出流 OutputStream。字符流和字节流都有对应的缓冲流，字节流也可以包装为字符流，缓冲流带有一个 8KB 的缓冲数组，可以提高流的读写效率。除了缓冲流外还有过滤流 FilterReader、字符数组流 CharArrayReader、字节数组流 ByteArrayInputStream、文件流 FileInputStream 等。

序列化

Java 对象 JVM 退出时会全部销毁，如果需要将对象及状态持久化，就要通过序列化实现，将内存中的对象保存在二进制流中，需要时再将二进制流反序列化为对象。对象序列化保存的是对象的状态，因此属于类属性的静态变量不会被序列化。

常见的序列化有三种：

- **Java 原生序列化**

  实现 `Serializabale` 标记接口，Java 序列化保留了对象类的元数据（如类、成员变量、继承类信息）以及对象数据，兼容性最好，但不支持跨语言，性能一般。序列化和反序列化必须保持序列化 ID 的一致，一般使用 `private static final long serialVersionUID` 定义序列化 ID，如果不设置编译器会根据类的内部实现自动生成该值。如果是兼容升级不应该修改序列化 ID，防止出错，如果是不兼容升级则需要修改。

- **Hessian 序列化**

  Hessian 序列化是一种支持动态类型、跨语言、基于对象传输的网络协议。Java 对象序列化的二进制流可以被其它语言反序列化。Hessian 协议的特性：① 自描述序列化类型，不依赖外部描述文件，用一个字节表示常用基础类型，极大缩短二进制流。② 语言无关，支持脚本语言。③ 协议简单，比 Java 原生序列化高效。Hessian 会把复杂对象所有属性存储在一个 Map 中序列化，当父类和子类存在同名成员变量时会先序列化子类再序列化父类，因此子类值会被父类覆盖。

- **JSON 序列化**

  JSON 序列化就是将数据对象转换为 JSON 字符串，在序列化过程中抛弃了类型信息，所以反序列化时只有提供类型信息才能准确进行。相比前两种方式可读性更好，方便调试。

序列化通常会使用网络传输对象，而对象中往往有敏感数据，容易遭受攻击，Jackson 和 fastjson 等都出现过反序列化漏洞，因此不需要进行序列化的敏感属性传输时应加上 transient 关键字。transient 的作用就是把变量生命周期仅限于内存而不会写到磁盘里持久化，变量会被设为对应数据类型的零值。

JVM

内存区域划分

程序计数器

程序计数器是一块较小的内存空间，可以看作当前线程所执行字节码的行号指示器。字节码解释器工作时通过改变计数器的值选取下一条执行指令。分支、循环、跳转、线程恢复等功能都需要依赖计数器完成。是唯一在虚拟机规范中没有规定内存溢出情况的区域。

如果线程正在执行 Java 方法，计数器记录正在执行的虚拟机字节码指令地址。如果是本地方法，计数器值为 Undefined。

Java 虚拟机栈

Java 虚拟机栈来描述 Java 方法的内存模型。每当有新线程创建时就会分配一个栈空间，线程结束后栈空间被回收，栈与线程拥有相同的生命周期。栈中元素用于支持虚拟机进行方法调用，每个方法在执行时都会创建一个栈帧存储方法的局部变量表、操作栈、动态链接和方法出口等信息。每个方法从调用到执行完成，就是栈帧从入栈到出栈的过程。

有两类异常：① 线程请求的栈深度大于虚拟机允许的深度抛出 StackOverflowError。② 如果 JVM 栈容量可以动态扩展，栈扩展无法申请足够内存抛出 OutOfMemoryError（HotSpot 不可动态扩展，不存在此问题）。

本地方法栈

本地方法栈与虚拟机栈作用相似，不同的是虚拟机栈为虚拟机执行 Java 方法服务，本地方法栈为虚本地方法服务。调用本地方法时虚拟机栈保持不变，动态链接并直接调用指定本地方法。

虚拟机规范对本地方法栈中方法的语言与数据结构无强制规定，虚拟机可自由实现，例如 HotSpot 将虚拟机栈和本地方法栈合二为一。

本地方法栈在栈深度异常和栈扩展失败时分别抛出 StackOverflowError 和 OutOfMemoryError。

堆

堆是虚拟机所管理的内存中最大的一块，被所有线程共享的，在虚拟机启动时创建。堆用来存放对象实例，Java 里几乎所有对象实例都在堆分配内存。堆可以处于物理上不连续的内存空间，逻辑上应该连续，但对于例如数组这样的大对象，多数虚拟机实现出于简单、存储高效的考虑会要求连续的内存空间。

堆既可以被实现成固定大小，也可以是可扩展的，可通过 `-Xms` 和 `-Xmx` 设置堆的最小和最大容量，当前主流 JVM 都按照可扩展实现。如果堆没有内存完成实例分配也无法扩展，抛出 OutOfMemoryError。

方法区

方法区用于存储被虚拟机加载的类型信息、常量、静态变量、即时编译器编译后的代码缓存等数据。

JDK8 之前使用永久代实现方法区，容易内存溢出，因为永久代有 `-XX:MaxPermSize` 上限，即使不设置也有默认大小。JDK7 把放在永久代的字符串常量池、静态变量等移出，JDK8 中永久代完全废弃，改用在本地内存中实现的元空间代替，把 JDK 7 中永久代剩余内容（主要是类型信息）全部移到元空间。

虚拟机规范对方法区的约束宽松，除和堆一样不需要连续内存和可选择固定大小/可扩展外，还可以不实现垃圾回收。垃圾回收在方法区出现较少，主要目标针对常量池和类型卸载。如果方法区无法满足新的内存分配需求，将抛出 OutOfMemoryError。

运行时常量池

运行时常量池是方法区的一部分，Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池表，用于存放编译器生成的各种字面量与符号引用，这部分内容在类加载后存放到运行时常量池。一般除了保存 Class 文件中描述的符号引用外，还会把符号引用翻译的直接引用也存储在运行时常量池。

运行时常量池相对于 Class 文件常量池的一个重要特征是动态性，Java 不要求常量只有编译期才能产生，运行期间也可以将新的常量放入池中，这种特性利用较多的是 String 的 `intern` 方法。

运行时常量池是方法区的一部分，受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError。

直接内存

直接内存不属于运行时数据区，也不是虚拟机规范定义的内存区域，但这部分内存被频繁使用，而且可能导致内存溢出。

JDK1.4 中新加入了 NIO 这种基于通道与缓冲区的 IO，它可以使用 Native 函数库直接分配堆外内存，通过一个堆里的 DirectByteBuffer 对象作为内存的引用进行操作，避免了在 Java 堆和 Native堆来回复制数据。

直接内存的分配不受 Java 堆大小的限制，但还是会受到本机总内存及处理器寻址空间限制，一般配置虚拟机参数时会根据实际内存设置 `-Xmx` 等参数信息，但经常忽略直接内存，使内存区域总和大于物理内存限制，导致动态扩展时出现 OOM。

由直接内存导致的内存溢出，一个明显的特征是在 Heap Dump 文件中不会看见明显的异常，如果发现内存溢出后产生的 Dump 文件很小，而程序中又直接或间接使用了直接内存（典型的间接使用就是 NIO），那么就可以考虑检查直接内存方面的原因。

内存溢出

内存溢出和内存泄漏

内存溢出 OutOfMemory，指程序在申请内存时，没有足够的内存空间供其使用。

内存泄露 Memory Leak，指程序在申请内存后，无法释放已申请的内存空间，内存泄漏最终将导致内存溢出。

堆溢出

堆用于存储对象实例，只要不断创建对象并保证 GC Roots 到对象有可达路径避免垃圾回收，随着对象数量的增加，总容量触及最大堆容量后就会 OOM，例如在 while 死循环中一直 new 创建实例。

堆 OOM 是实际应用中最常见的 OOM，处理方法是通过内存映像分析工具对 Dump 出的堆转储快照分析，确认内存中导致 OOM 的对象是否必要，分清到底是内存泄漏还是内存溢出。

如果是内存泄漏，通过工具查看泄漏对象到 GC Roots 的引用链，找到泄露对象是通过怎样的引用路径、与哪些 GC Roots 关联才导致无法回收，一般可以准确定位到产生内存泄漏代码的具体位置。

如果不是内存泄漏，即内存中对象都必须存活，应当检查 JVM 堆参数，与机器内存相比是否还有向上调整的空间。再从代码检查是否存在某些对象生命周期过长、持有状态时间过长、存储结构设计不合理等情况，尽量减少程序运行期的内存消耗。

栈溢出

由于 HotSpot 不区分虚拟机和本地方法栈，设置本地方法栈大小的参数没有意义，栈容量只能由 `-Xss` 参数来设定，存在两种异常：

StackOverflowError： 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError，例如一个递归方法不断调用自己。该异常有明确错误堆栈可供分析，容易定位到问题所在。

OutOfMemoryError： 如果 JVM 栈可以动态扩展，当扩展无法申请到足够内存时会抛出 OutOfMemoryError。HotSpot 不支持虚拟机栈扩展，所以除非在创建线程申请内存时就因无法获得足够内存而出现 OOM，否则在线程运行时是不会因为扩展而导致溢出的。

运行时常量池溢出

String 的 `intern` 方法是一个本地方法，作用是如果字符串常量池中已包含一个等于此 String 对象的字符串，则返回池中这个字符串的 String 对象的引用，否则将此 String 对象包含的字符串添加到常量池并返回此 String 对象的引用。

在 JDK6 及之前常量池分配在永久代，因此可以通过 `-XX:PermSize` 和 `-XX:MaxPermSize` 限制永久代大小，间接限制常量池。在 while 死循环中调用 `intern` 方法导致运行时常量池溢出。在 JDK7 后不会出现该问题，因为存放在永久代的字符串常量池已经被移至堆中。

方法区溢出

方法区主要存放类型信息，如类名、访问修饰符、常量池、字段描述、方法描述等。只要不断在运行时产生大量类，方法区就会溢出。例如使用 JDK 反射或 CGLib 直接操作字节码在运行时生成大量的类。很多框架如 Spring、Hibernate 等对类增强时都会使用 CGLib 这类字节码技术，增强的类越多就需要越大的方法区保证动态生成的新类型可以载入内存，也就更容易导致方法区溢出。

JDK8 使用元空间取代永久代，HotSpot 提供了一些参数作为元空间防御措施，例如 `-XX:MetaspaceSize` 指定元空间初始大小，达到该值会触发 GC 进行类型卸载，同时收集器会对该值进行调整，如果释放大量空间就适当降低该值，如果释放很少空间就适当提高。

创建对象

创建对象的过程

字节码角度

- **NEW：** 如果找不到 Class 对象则进行类加载。加载成功后在堆中分配内存，从 Object 到本类路径上的所有属性都要分配。分配完毕后进行零值设置。最后将指向实例对象的引用变量压入虚拟机栈顶。
- **DUP：** 在栈顶复制引用变量，这时栈顶有两个指向堆内实例的引用变量。两个引用变量的目的不同，栈底的引用用于赋值或保存局部变量表，栈顶的引用作为句柄调用相关方法。
- **INVOKESPECIAL：** 通过栈顶的引用变量调用 init 方法。

执行角度

① 当 JVM 遇到字节码 new 指令时，首先将检查该指令的参数能否在常量池中定位到一个类的符号引用，并检查引用代表的类是否已被加载、解析和初始化，如果没有就先执行类加载。

② 在类加载检查通过后虚拟机将为新生对象分配内存。

③ 内存分配完成后虚拟机将成员变量设为零值，保证对象的实例字段可以不赋初值就使用。

④ 设置对象头，包括哈希码、GC 信息、锁信息、对象所属类的类元信息等。

⑤ 执行 init 方法，初始化成员变量，执行实例化代码块，调用类的构造方法，并把堆内对象的首地址赋值给引用变量。

对象分配内存

对象所需内存大小在类加载完成后便可完全确定，分配空间的任务实际上等于把一块确定大小的内存块从 Java 堆中划分出来。

指针碰撞： 假设 Java 堆内存规整，被使用过的内存放在一边，空闲的放在另一边，中间放着一个指针作为分界指示器，分配内存就是把指针向空闲方向挪动一段与对象大小相等的距离。

空闲列表： 如果 Java 堆内存不规整，虚拟机必须维护一个列表记录哪些内存可用，在分配时从列表中找到一块足够大的空间划分给对象并更新列表记录。

选择哪种分配方式由堆是否规整决定，堆是否规整由垃圾收集器是否有空间压缩能力决定。使用 Serial、ParNew 等收集器时，系统采用指针碰撞；使用 CMS 这种基于清除算法的垃圾收集器时，采用空间列表。

对象创建十分频繁，即使修改一个指针的位置在并发下也不是线程安全的，可能正给对象 A 分配内存，指针还没来得及修改，对象 B 又使用了指针来分配内存。

解决方法：① CAS 加失败重试保证更新原子性。② 把内存分配按线程划分在不同空间，即每个线程在 Java 堆中预先分配一小块内存，叫做本地线程分配缓冲 TLAB，哪个线程要分配内存就在对应的 TLAB 分配，TLAB 用完了再进行同步。

对象的内存布局

对象在堆内存的存储布局可分为对象头、实例数据和对齐填充。

对象头占 12B，包括对象标记和类型指针。对象标记存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁标志、偏向线程 ID 等，这部分占 8B，称为 Mark Word。Mark Word 被设计为动态数据结构，以便在极小的空间存储更多数据，根据对象状态复用存储空间。

类型指针是对象指向它的类型元数据的指针，占 4B。JVM 通过该指针来确定对象是哪个类的实例。

实例数据是对象真正存储的有效信息，即本类对象的实例成员变量和所有可见的父类成员变量。存储顺序会受到虚拟机分配策略参数和字段在源码中定义顺序的影响。相同宽度的字段总是被分配到一起存放，在满足该前提条件的情况下父类中定义的变量会出现在子类之前。

对齐填充不是必然存在的，仅起占位符作用。虚拟机的自动内存管理系统要求任何对象的大小必须是 8B 的倍数，对象头已被设为 8B 的 1 或 2 倍，如果对象实例数据部分没有对齐，需要对齐填充补全。

对象的访问方式

Java 程序会通过栈上的 reference 引用操作堆对象，访问方式由虚拟机决定，主流访问方式主要有句柄和直接指针。

句柄： 堆会划分出一块内存作为句柄池，reference 中存储对象的句柄地址，句柄包含对象实例数据与类型数据的地址信息。优点是 reference 中存储的是稳定句柄地址，在 GC 过程中对象被移动时只会改变句柄的实例数据指针，而 reference 本身不需要修改。

直接指针： 堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference 存储对象地址，如果只是访问对象本身就不需要多一次间接访问的开销。优点是速度更快，节省了一次指针定位的时间开销，HotSpot 主要使用直接指针进行对象访问。

垃圾回收

判断对象是否是垃圾

引用计数：在对象中添加一个引用计数器，如果被引用计数器加 1，引用失效时计数器减 1，如果计数器为 0 则被标记为垃圾。原理简单，效率高，但是在 Java 中很少使用，因为存在对象间循环引用的问题，导致计数器无法清零。

可达性分析：主流语言的内存管理都使用可达性分析判断对象是否存活。基本思路是通过一系列称为 GC Roots 的根对象作为起始节点集，从这些节点开始，根据引用关系向下搜索，搜索过程走过的路径称为引用链，如果某个对象到 GC Roots 没有任何引用链相连，则会被标记为垃圾。可作为 GC Roots 的对象包括虚拟机栈和本地方法栈中引用的对象、类静态属性引用的对象、常量引用的对象。

Java 的引用类型

JDK1.2 后对引用进行了扩充，按强度分为四种：

强引用： 最常见的引用，例如 `Object obj = new Object()` 就属于强引用。只要对象有强引用指向且 GC Roots 可达，在内存回收时即使濒临内存耗尽也不会被回收。

软引用： 弱于强引用，描述非必需对象。在系统将发生内存溢出前，会把软引用关联的对象加入回收范围以获得更多内存空间。用来缓存服务器中间计算结果及不需要实时保存的用户行为等。

弱引用： 弱于软引用，描述非必需对象。弱引用关联的对象只能生存到下次 YGC 前，当垃圾收集器开始工作时无论当前内存是否足够都会回收只被弱引用关联的对象。由于 YGC 具有不确定性，因此弱引用何时被回收也不确定。

虚引用： 最弱的引用，定义完成后无法通过该引用获取对象。唯一目的就是为了能在对象被回收时收到一个系统通知。虚引用必须与引用队列联合使用，垃圾回收时如果出现虚引用，就会在回收对象前把这个虚引用加入引用队列。

GC 算法

标记-清除算法

分为标记和清除阶段，首先从每个 GC Roots 出发依次标记有引用关系的对象，最后清除没有标记的对象。

执行效率不稳定，如果堆包含大量对象且大部分需要回收，必须进行大量标记清除，导致效率随对象数量增长而降低。

存在内存空间碎片化问题，会产生大量不连续的内存碎片，导致以后需要分配大对象时容易触发 Full GC。

标记-复制算法

为了解决内存碎片问题，将可用内存按容量划分为大小相等的两块，每次只使用其中一块。当使用的这块空间用完了，就将存活对象复制到另一块，再把已使用过的内存空间一次清理掉。主要用于进行新生代。

实现简单、运行高效，解决了内存碎片问题。 代价是可用内存缩小为原来的一半，浪费空间。

HotSpot 把新生代划分为一块较大的 Eden 和两块较小的 Survivor，每次分配内存只使用 Eden 和其中一块 Survivor。垃圾收集时将 Eden 和 Survivor 中仍然存活的对象一次性复制到另一块 Survivor 上，然后直接清理掉 Eden 和已用过的那块 Survivor。HotSpot 默认Eden 和 Survivor 的大小比例是 8:1，即每次新生代中可用空间为整个新生代的 90%。

标记-整理算法

标记-复制算法在对象存活率高时要进行较多复制操作，效率低。如果不想浪费空间，就需要有额外空间分配担保，应对被使用内存中所有对象都存活的极端情况，所以老年代一般不使用此算法。

老年代使用标记-整理算法，标记过程与标记-清除算法一样，但不直接清理可回收对象，而是让所有存活对象都向内存空间一端移动，然后清理掉边界以外的内存。

标记-清除与标记-整理的差异在于前者是一种非移动式算法而后者是移动式的。如果移动存活对象，尤其是在老年代这种每次回收都有大量对象存活的区域，是一种极为负重的操作，而且移动必须全程暂停用户线程。如果不移动对象就会导致空间碎片问题，只能依赖更复杂的内存分配器和访问器解决。

垃圾收集器

Serial

最基础的收集器，使用复制算法、单线程工作，只用一个处理器或一条线程完成垃圾收集，进行垃圾收集时必须暂停其他所有工作线程。

Serial 是虚拟机在客户端模式的默认新生代收集器，简单高效，对于内存受限的环境它是所有收集器中额外内存消耗最小的，对于处理器核心较少的环境，Serial 由于没有线程交互开销，可获得最高的单线程收集效率。

ParNew

Serial 的多线程版本，除了使用多线程进行垃圾收集外其余行为完全一致。

ParNew 是虚拟机在服务端模式的默认新生代收集器，一个重要原因是除了 Serial 外只有它能与 CMS 配合。自从 JDK 9 开始，ParNew 加 CMS 不再是官方推荐的解决方案，官方希望它被 G1 取代。

Parallel Scavenge

新生代收集器，基于复制算法，是可并行的多线程收集器，与 ParNew 类似。

特点是它的关注点与其他收集器不同，Parallel Scavenge 的目标是达到一个可控制的吞吐量，吞吐量就是处理器用于运行用户代码的时间与处理器消耗总时间的比值。

Serial Old

Serial 的老年代版本，单线程工作，使用标记-整理算法。

Serial Old 是虚拟机在客户端模式的默认老年代收集器，用于服务端有两种用途：① JDK5 及之前与 Parallel Scavenge 搭配。② 作为CMS 失败预案。

Parellel Old

Parallel Scavenge 的老年代版本，支持多线程，基于标记-整理算法。JDK6 提供，注重吞吐量可考虑 Parallel Scavenge 加 Parallel Old。

CMS

以获取最短回收停顿时间为目标，基于标记-清除算法，过程相对复杂，分为四个步骤：初始标记、并发标记、重新标记、并发清除。

初始标记和重新标记需要 STW（Stop The World，系统停顿），初始标记仅是标记 GC Roots 能直接关联的对象，速度很快。并发标记从 GC Roots 的直接关联对象开始遍历整个对象图，耗时较长但不需要停顿用户线程。重新标记则是为了修正并发标记期间因用户程序运作而导致标记产生变动的那部分记录。并发清除清理标记阶段判断的已死亡对象，不需要移动存活对象，该阶段也可与用户线程并发。

缺点：① 对处理器资源敏感，并发阶段虽然不会导致用户线程暂停，但会降低吞吐量。② 无法处理浮动垃圾，有可能出现并发失败而导致 Full GC。③ 基于标记-清除算法，产生空间碎片。

G1

开创了收集器面向局部收集的设计思路和基于 Region 的内存布局，主要面向服务端，最初设计目标是替换 CMS。

G1 之前的收集器，垃圾收集目标要么是整个新生代，要么是整个老年代或整个堆。而 G1 可面向堆任何部分来组成回收集进行回收，衡量标准不再是分代，而是哪块内存中存放的垃圾数量最多，回收受益最大。

跟踪各 Region 里垃圾的价值，价值即回收所获空间大小以及回收所需时间的经验值，在后台维护一个优先级列表，每次根据用户设定允许的收集停顿时间优先处理回收价值最大的 Region。这种方式保证了 G1 在有限时间内获取尽可能高的收集效率。

G1 运作过程：

- **初始标记：**标记 GC Roots 能直接关联到的对象，让下一阶段用户线程并发运行时能正确地在可用 Region 中分配新对象。需要 STW 但耗时很短，在 Minor GC 时同步完成。
- **并发标记：**从 GC Roots 开始对堆中对象进行可达性分析，递归扫描整个堆的对象图。耗时长但可与用户线程并发，扫描完成后要重新处理 SATB 记录的在并发时有变动的对象。
- **最终标记：**对用户线程做短暂暂停，处理并发阶段结束后仍遗留下来的少量 SATB 记录。
- **筛选回收：**对各 Region 的回收价值排序，根据用户期望停顿时间制定回收计划。必须暂停用户线程，由多条收集线程并行完成。

可由用户指定期望停顿时间是 G1 的一个强大功能，但该值不能设得太低，一般设置为100~300 ms。

ZGC

JDK11 中加入的具有实验性质的低延迟垃圾收集器，目标是尽可能在不影响吞吐量的前提下，实现在任意堆内存大小都可以把停顿时间限制在 10ms 以内的低延迟。

基于 Region 内存布局，不设分代，使用了读屏障、染色指针和内存多重映射等技术实现可并发的标记-整理，以低延迟为首要目标。

ZGC 的 Region 具有动态性，是动态创建和销毁的，并且容量大小也是动态变化的。

内存分配与回收策略

对象优先在 Eden 区分配

大多数情况下对象在新生代 Eden 区分配，当 Eden 没有足够空间时将发起一次 Minor GC。

大对象直接进入老年代

大对象指需要大量连续内存空间的对象，典型是很长的字符串或数量庞大的数组。大对象容易导致内存还有不少空间就提前触发垃圾收集以获得足够的连续空间。

HotSpot 提供了 `-XX:PretenureSizeThreshold` 参数，大于该值的对象直接在老年代分配，避免在 Eden 和 Survivor 间来回复制。

长期存活对象进入老年代

虚拟机给每个对象定义了一个对象年龄计数器，存储在对象头。如果经历过第一次 Minor GC 仍然存活且能被 Survivor 容纳，该对象就会被移动到 Survivor 中并将年龄设置为 1。对象在 Survivor 中每熬过一次 Minor GC 年龄就加 1 ，当增加到一定程度（默认15）就会被晋升到老年代。对象晋升老年代的阈值可通过 `-XX:MaxTenuringThreshold` 设置。

动态对象年龄判定

为了适应不同内存状况，虚拟机不要求对象年龄达到阈值才能晋升老年代，如果在 Survivor 中相同年龄所有对象大小的总和大于 Survivor 的一半，年龄不小于该年龄的对象就可以直接进入老年代。

空间分配担保

MinorGC 前虚拟机必须检查老年代最大可用连续空间是否大于新生代对象总空间，如果满足则说明这次 Minor GC 确定安全。

如果不满足，虚拟机会查看 `-XX:HandlePromotionFailure` 参数是否允许担保失败，如果允许会继续检查老年代最大可用连续空间是否大于历次晋升老年代对象的平均大小，如果满足将冒险尝试一次 Minor GC，否则改成一次 FullGC。

冒险是因为新生代使用复制算法，为了内存利用率只使用一个 Survivor，大量对象在 Minor GC 后仍然存活时，需要老年代进行分配担保，接收 Survivor 无法容纳的对象。

故障处理工具

jps：虚拟机进程状况工具

功能和 ps 命令类似：可以列出正在运行的虚拟机进程，显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一 ID（LVMID)。LVMID 与操作系统的进程 ID（PID）一致，使用 Windows 的任务管理器或 UNIX 的 ps 命令也可以查询到虚拟机进程的 LVMID，但如果同时启动了多个虚拟机进程，必须依赖 jps 命令。

jstat：虚拟机统计信息监视工具

用于监视虚拟机各种运行状态信息。可以显示本地或远程虚拟机进程中的类加载、内存、垃圾收集、即时编译器等运行时数据，在没有 GUI 界面的服务器上是运行期定位虚拟机性能问题的常用工具。

参数含义：S0 和 S1 表示两个 Survivor，E 表示新生代，O 表示老年代，YGC 表示 Young GC 次数，YGCT 表示 Young GC 耗时，FGC 表示 Full GC 次数，FGCT 表示 Full GC 耗时，GCT 表示 GC 总耗时。

jinfo：Java 配置信息工具

实时查看和调整虚拟机各项参数，使用 jps 的 -v 参数可以查看虚拟机启动时显式指定的参数，但如果想知道未显式指定的参数值只能使用 jinfo 的 -flag 查询。

jmap：Java 内存映像工具

用于生成堆转储快照，还可以查询 finalize 执行队列、Java 堆和方法区的详细信息，如空间使用率，当前使用的是哪种收集器等。和 jinfo 一样，部分功能在 Windows 受限，除了生成堆转储快照的 -dump 和查看每个类实例的 -histo 外，其余选项只能在 Linux 使用。

jhat：虚拟机堆转储快照分析工具

JDK 提供 jhat 与 jmap 搭配使用分析 jmap 生成的堆转储快照。jhat 内置了一个微型的 HTTP/Web 服务器，生成堆转储快照的分析结果后可以在浏览器查看。

jstack：Java 堆栈跟踪工具

用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的目的通常是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间挂起等。线程出现停顿时通过 jstack 查看各个线程的调用堆栈，可以获知没有响应的线程在后台做什么或等什么资源。

类加载机制

Java 程序运行过程

- 首先通过 Javac 编译器将 `.java` 转为 JVM 可加载的 `.class` 字节码文件。

  Javac 是由 Java 编写的程序，编译过程可以分为： ① 词法解析，通过空格分割出单词、操作符、控制符等信息，形成 token 信息流，传递给语法解析器。② 语法解析，把 token 信息流按照 Java 语法规则组装成语法树。③ 语义分析，检查关键字使用是否合理、类型是否匹配、作用域是否正确等。④ 字节码生成，将前面各个步骤的信息转换为字节码。

  字节码必须通过类加载过程加载到 JVM 后才可以执行，执行有三种模式，解释执行、JIT 编译执行、JIT 编译与解释器混合执行（主流 JVM 默认执行的方式）。混合模式的优势在于解释器在启动时先解释执行，省去编译时间。

- 之后通过即时编译器 JIT 把字节码文件编译成本地机器码。

  Java 程序最初都是通过解释器进行解释执行的，当虚拟机发现某个方法或代码块的运行特别频繁，就会认定其为"热点代码"，热点代码的检测主要有基于采样和基于计数器两种方式，为了提高热点代码的执行效率，虚拟机会把它们编译成本地机器码，尽可能对代码优化，在运行时完成这个任务的后端编译器被称为即时编译器。

- 还可以通过静态的提前编译器 AOT 直接把程序编译成与目标机器指令集相关的二进制代码。

类加载

Class 文件中描述的各类信息都需要加载到虚拟机后才能使用。JVM 把描述类的数据从 Class 文件加载到内存，并对数据进行校验、解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这个过程称为虚拟机的类加载机制。

与编译时需要连接的语言不同，Java 中类型的加载、连接和初始化都是在运行期间完成的，这增加了性能开销，但却提供了极高的扩展性，Java 动态扩展的语言特性就是依赖运行期动态加载和连接实现的。

一个类型从被加载到虚拟机内存开始，到卸载出内存为止，整个生命周期经历加载、验证、准备、解析、初始化、使用和卸载七个阶段，其中验证、解析和初始化三个部分称为连接。加载、验证、准备、初始化阶段的顺序是确定的，解析则不一定：可能在初始化之后再开始，这是为了支持 Java 的动态绑定。

类初始化的情况

① 遇到 `new`、`getstatic`、`putstatic` 或 `invokestatic` 字节码指令时，还未初始化。典型场景包括 new 实例化对象、读取或设置静态字段、调用静态方法。

② 对类反射调用时，还未初始化。

③ 初始化类时，父类还未初始化。

④ 虚拟机启动时，会先初始化包含 main 方法的主类。

⑤ 使用 JDK7 的动态语言支持时，如果 MethodHandle 实例的解析结果为指定类型的方法句柄且句柄对应的类还未初始化。

⑥ 接口定义了默认方法，如果接口的实现类初始化，接口要在其之前初始化。

其余所有引用类型的方式都不会触发初始化，称为被动引用。被动引用实例：① 子类使用父类的静态字段时，只有父类被初始化。② 通过数组定义使用类。③ 常量在编译期会存入调用类的常量池，不会初始化定义常量的类。

接口和类加载过程的区别：初始化类时如果父类没有初始化需要初始化父类，但接口初始化时不要求父接口初始化，只有在真正使用父接口时（如引用接口中定义的常量）才会初始化。

类加载的过程

加载

该阶段虚拟机需要完成三件事：① 通过一个类的全限定类名获取定义类的二进制字节流。② 将字节流所代表的静态存储结构转化为方法区的运行时数据区。③ 在内存中生成对应该类的 Class 实例，作为方法区这个类的数据访问入口。

验证

确保 Class 文件的字节流符合约束。如果虚拟机不检查输入的字节流，可能因为载入有错误或恶意企图的字节流而导致系统受攻击。验证主要包含四个阶段：文件格式验证、元数据验证、字节码验证、符号引用验证。

验证重要但非必需，因为只有通过与否的区别，通过后对程序运行期没有任何影响。如果代码已被反复使用和验证过，在生产环境就可以考虑关闭大部分验证缩短类加载时间。

准备

为类静态变量分配内存并设置零值，该阶段进行的内存分配仅包括类变量，不包括实例变量。如果变量被 final 修饰，编译时 Javac 会为变量生成 ConstantValue 属性，准备阶段虚拟机会将变量值设为代码值。

解析

将常量池内的符号引用替换为直接引用。

符号引用以一组符号描述引用目标，可以是任何形式的字面量，只要使用时能无歧义地定位目标即可。与虚拟机内存布局无关，引用目标不一定已经加载到虚拟机内存。

直接引用是可以直接指向目标的指针、相对偏移量或能间接定位到目标的句柄。和虚拟机的内存布局相关，引用目标必须已在虚拟机的内存中存在。

初始化

直到该阶段 JVM 才开始执行类中编写的代码。准备阶段时变量赋过零值，初始化阶段会根据程序员的编码去初始化类变量和其他资源。初始化阶段就是执行类构造方法中的 `<client>` 方法，该方法是 Javac 自动生成的。

类加载器

自 JDK1.2 起 Java 一直保持三层类加载器：

- **启动类加载器**

  在 JVM 启动时创建，负责加载最核心的类，例如 Object、System 等。无法被程序直接引用，如果需要把加载委派给启动类加载器，直接使用 null 代替即可，因为启动类加载器通常由操作系统实现，并不存在于 JVM 体系。

- **平台类加载器**

  从 JDK9 开始从扩展类加载器更换为平台类加载器，负载加载一些扩展的系统类，比如 XML、加密、压缩相关的功能类等。

- **应用类加载器**

  也称系统类加载器，负责加载用户类路径上的类库，可以直接在代码中使用。如果没有自定义类加载器，一般情况下应用类加载器就是默认的类加载器。自定义类加载器通过继承 ClassLoader 并重写 `findClass` 方法实现。

双亲委派模型

类加载器具有等级制度但非继承关系，以组合的方式复用父加载器的功能。双亲委派模型要求除了顶层的启动类加载器外，其余类加载器都应该有自己的父加载器。

一个类加载器收到了类加载请求，它不会自己去尝试加载，而将该请求委派给父加载器，每层的类加载器都是如此，因此所有加载请求最终都应该传送到启动类加载器，只有当父加载器反馈无法完成请求时，子加载器才会尝试。

类跟随它的加载器一起具备了有优先级的层次关系，确保某个类在各个类加载器环境中都是同一个，保证程序的稳定性。

判断两个类是否相等

任意一个类都必须由类加载器和这个类本身共同确立其在虚拟机中的唯一性。

两个类只有由同一类加载器加载才有比较意义，否则即使两个类来源于同一个 Class 文件，被同一个 JVM 加载，只要类加载器不同，这两个类就必定不相等。

并发

JMM

JMM

Java 线程的通信由 JMM 控制，JMM 的主要目的是定义程序中各种变量的访问规则。变量包括实例字段、静态字段，但不包括局部变量与方法参数，因为它们是线程私有的，不存在多线程竞争。JMM 遵循一个基本原则：只要不改变程序执行结果，编译器和处理器怎么优化都行。例如编译器分析某个锁只会单线程访问就消除锁，某个 volatile 变量只会单线程访问就把它当作普通变量。

JMM 规定所有变量都存储在主内存，每条线程有自己的工作内存，工作内存中保存被该线程使用的变量的主内存副本，线程对变量的所有操作都必须在工作空间进行，不能直接读写主内存数据。不同线程间无法直接访问对方工作内存中的变量，线程通信必须经过主内存。

关于主内存与工作内存的交互，即变量如何从主内存拷贝到工作内存、从工作内存同步回主内存，JMM 定义了 8 种原子操作：

操作	作用变量范围	作用
lock	主内存	把变量标识为线程独占状态
unlock	主内存	释放处于锁定状态的变量
read	主内存	把变量值从主内存传到工作内存
load	工作内存	把 read 得到的值放入工作内存的变量副本
user	工作内存	把工作内存中的变量值传给执行引擎
assign	工作内存	把从执行引擎接收的值赋给工作内存变量
store	工作内存	把工作内存的变量值传到主内存
write	主内存	把 store 取到的变量值放入主内存变量中

---

as-if-serial

不管怎么重排序，单线程程序的执行结果不能改变，编译器和处理器必须遵循 as-if-serial 语义。

为了遵循 as-if-serial，编译器和处理器不会对存在数据依赖关系的操作重排序，因为这种重排序会改变执行结果。但是如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

as-if-serial 把单线程程序保护起来，给程序员一种幻觉：单线程程序是按程序的顺序执行的。

---

happens-before

先行发生原则，JMM 定义的两项操作间的偏序关系，是判断数据是否存在竞争的重要手段。

JMM 将 happens-before 要求禁止的重排序按是否会改变程序执行结果分为两类。对于会改变结果的重排序 JMM 要求编译器和处理器必须禁止，对于不会改变结果的重排序，JMM 不做要求。

JMM 存在一些天然的 happens-before 关系，无需任何同步器协助就已经存在。如果两个操作的关系不在此列，并且无法从这些规则推导出来，它们就没有顺序性保障，虚拟机可以对它们随意进行重排序。

- **程序次序规则**：一个线程内写在前面的操作先行发生于后面的。
- **管程锁定规则**： unlock 操作先行发生于后面对同一个锁的 lock 操作。
- **volatile 规则**：对 volatile 变量的写操作先行发生于后面的读操作。
- **线程启动规则**：线程的 `start` 方法先行发生于线程的每个动作。
- **线程终止规则**：线程中所有操作先行发生于对线程的终止检测。
- **对象终结规则**：对象的初始化先行发生于 `finalize` 方法。

- **传递性**：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C 。

---

as-if-serial 和 happens-before 的区别

as-if-serial 保证单线程程序的执行结果不变，happens-before 保证正确同步的多线程程序的执行结果不变。

这两种语义的目的都是为了在不改变程序执行结果的前提下尽可能提高程序执行并行度。

指令重排序

为了提高性能，编译器和处理器通常会对指令进行重排序，重排序指从源代码到指令序列的重排序，分为三种：① 编译器优化的重排序，编译器在不改变单线程程序语义的前提下可以重排语句的执行顺序。② 指令级并行的重排序，如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。③ 内存系统的重排序。

原子性、可见性、有序性

原子性

基本数据类型的访问都具备原子性，例外就是 long 和 double，虚拟机将没有被 volatile 修饰的 64 位数据操作划分为两次 32 位操作。

如果应用场景需要更大范围的原子性保证，JMM 还提供了 lock 和 unlock 操作满足需求，尽管 JVM 没有把这两种操作直接开放给用户使用，但是提供了更高层次的字节码指令 monitorenter 和 monitorexit，这两个字节码指令反映到 Java 代码中就是 synchronized。

可见性

可见性指当一个线程修改了共享变量时，其他线程能够立即得知修改。JMM 通过在变量修改后将值同步回主内存，在变量读取前从主内存刷新的方式实现可见性，无论普通变量还是 volatile 变量都是如此，区别是 volatile 保证新值能立即同步到主内存以及每次使用前立即从主内存刷新。

除了 volatile 外，synchronized 和 final 也可以保证可见性。同步块可见性由"对一个变量执行 unlock 前必须先把此变量同步回主内存，即先执行 store 和 write"这条规则获得。final 的可见性指：被 final 修饰的字段在构造方法中一旦初始化完成，并且构造方法没有把 this 引用传递出去，那么其他线程就能看到 final 字段的值。

有序性

有序性可以总结为：在本线程内观察所有操作是有序的，在一个线程内观察另一个线程，所有操作都是无序的。前半句指 as-if-serial 语义，后半句指指令重排序和工作内存与主内存延迟现象。

Java 提供 volatile 和 synchronized 保证有序性，volatile 本身就包含禁止指令重排序的语义，而 synchronized 保证一个变量在同一时刻只允许一条线程对其进行 lock 操作，确保持有同一个锁的两个同步块只能串行进入。

---

volatile

JMM 为 volatile 定义了一些特殊访问规则，当变量被定义为 volatile 后具备两种特性：

- **保证变量对所有线程可见**

当一条线程修改了变量值，新值对于其他线程来说是立即可以得知的。volatile 变量在各个线程的工作内存中不存在一致性问题，但 Java 的运算操作符并非原子操作，导致 volatile 变量运算在并发下仍不安全。

- **禁止指令重排序优化**

  使用 volatile 变量进行写操作，汇编指令带有 lock 前缀，相当于一个内存屏障，后面的指令不能重排到内存屏障之前。

  使用 lock 前缀引发两件事：① 将当前处理器缓存行的数据写回系统内存。②使其他处理器的缓存无效。相当于对缓存变量做了一次 store 和 write 操作，让 volatile 变量的修改对其他处理器立即可见。

静态变量 i 执行多线程 i++ 的不安全问题

自增语句由 4 条字节码指令构成的，依次为 `getstatic`、`iconst_1`、`iadd`、`putstatic`，当 `getstatic` 把 i 的值取到操作栈顶时，volatile 保证了 i 值在此刻正确，但在执行 `iconst_1`、`iadd` 时，其他线程可能已经改变了 i 值，操作栈顶的值就变成了过期数据，所以 `putstatic` 执行后就可能把较小的 i 值同步回了主内存。

适用场景

① 运算结果并不依赖变量的当前值。② 一写多读，只有单一的线程修改变量值。

内存语义

写一个 volatile 变量时，把该线程工作内存中的值刷新到主内存。

读一个 volatile 变量时，把该线程工作内存值置为无效，从主内存读取。

指令重排序特点

第二个操作是 volatile 写，不管第一个操作是什么都不能重排序，确保写之前的操作不会被重排序到写之后。

第一个操作是 volatile 读，不管第二个操作是什么都不能重排序，确保读之后的操作不会被重排序到读之前。

第一个操作是 volatile 写，第二个操作是 volatile 读不能重排序。

JSR-133 增强 volatile 语义的原因

在旧的内存模型中，虽然不允许 volatile 变量间重排序，但允许 volatile 变量与普通变量重排序，可能导致内存不可见问题。JSR-133 严格限制编译器和处理器对 volatile 变量与普通变量的重排序，确保 volatile 的写-读和锁的释放-获取具有相同的内存语义。

---

final

final 可以保证可见性，被 final 修饰的字段在构造方法中一旦被初始化完成，并且构造方法没有把 this 引用传递出去，在其他线程中就能看见 final 字段值。

在旧的 JMM 中，一个严重缺陷是线程可能看到 final 值改变。比如一个线程看到一个 int 类型 final 值为 0，此时该值是未初始化前的零值，一段时间后该值被某线程初始化，再去读这个 final 值会发现值变为 1。

为修复该漏洞，JSR-133 为 final 域增加重排序规则：只要对象是正确构造的（被构造对象的引用在构造方法中没有逸出），那么不需要使用同步就可以保证任意线程都能看到这个 final 域初始化后的值。

写 final 域重排序规则

禁止把 final 域的写重排序到构造方法之外，编译器会在 final 域的写后，构造方法的 return 前，插入一个 Store Store 屏障。确保在对象引用为任意线程可见之前，对象的 final 域已经初始化过。

读 final 域重排序规则

在一个线程中，初次读对象引用和初次读该对象包含的 final 域，JMM 禁止处理器重排序这两个操作。编译器在读 final 域操作的前面插入一个 Load Load 屏障，确保在读一个对象的 final 域前一定会先读包含这个 final 域的对象引用。

锁

synchronized

每个 Java 对象都有一个关联的 monitor，使用 synchronized 时 JVM 会根据使用环境找到对象的 monitor，根据 monitor 的状态进行加解锁的判断。如果成功加锁就成为该 monitor 的唯一持有者，monitor 在被释放前不能再被其他线程获取。

同步代码块使用 monitorenter 和 monitorexit 这两个字节码指令获取和释放 monitor。这两个字节码指令都需要一个引用类型的参数指明要锁定和解锁的对象，对于同步普通方法，锁是当前实例对象；对于静态同步方法，锁是当前类的 Class 对象；对于同步方法块，锁是 synchronized 括号里的对象。

执行 monitorenter 指令时，首先尝试获取对象锁。如果这个对象没有被锁定，或当前线程已经持有锁，就把锁的计数器加 1，执行 monitorexit 指令时会将锁计数器减 1。一旦计数器为 0 锁随即就被释放。

例如有两个线程 A、B 竞争 monitor，当 A 竞争到锁时会将 monitor 中的 owner 设置为 A，把 B 阻塞并放到等待资源的 ContentionList 队列。ContentionList 中的部分线程会进入 EntryList，EntryList 中的线程会被指定为 OnDeck 竞争候选者，如果获得了锁资源将进入 Owner 状态，释放锁后进入 !Owner 状态。被阻塞的线程会进入 WaitSet。

被 synchronized 修饰的同步块对一条线程来说是可重入的，并且同步块在持有锁的线程释放锁前会阻塞其他线程进入。从执行成本的角度看，持有锁是一个重量级的操作。Java 线程是映射到操作系统的内核线程上的，如果要阻塞或唤醒一条线程，需要操作系统帮忙完成，不可避免用户态到核心态的转换。

不公平的原因

所有收到锁请求的线程首先自旋，如果通过自旋也没有获取锁将被放入 ContentionList，该做法对于已经进入队列的线程不公平。

为了防止 ContentionList 尾部的元素被大量线程进行 CAS 访问影响性能，Owner 线程会在释放锁时将 ContentionList 的部分线程移动到 EntryList 并指定某个线程为 OnDeck 线程，该行为叫做竞争切换，牺牲了公平性但提高了性能。

锁优化策略

JDK 6 对 synchronized 做了很多优化，引入了自适应自旋、锁消除、锁粗化、偏向锁和轻量级锁等提高锁的效率，锁一共有 4 个状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁，状态会随竞争情况升级。锁可以升级但不能降级，这种只能升级不能降级的锁策略是为了提高锁获得和释放的效率。

自旋锁

同步对性能最大的影响是阻塞，挂起和恢复线程的操作都需要转入内核态完成。许多应用上共享数据的锁定只会持续很短的时间，为了这段时间去挂起和恢复线程并不值得。如果机器有多个处理器核心，我们可以让后面请求锁的线程稍等一会，但不放弃处理器的执行时间，看看持有锁的线程是否很快会释放锁。为了让线程等待只需让线程执行一个忙循环，这项技术就是自旋锁。

自旋锁在 JDK1.4 就已引入，默认关闭，在 JDK6 中改为默认开启。自旋不能代替阻塞，虽然避免了线程切换开销，但要占用处理器时间，如果锁被占用的时间很短，自旋的效果就会非常好，反之只会白白消耗处理器资源。如果自旋超过了限定的次数仍然没有成功获得锁，就应挂起线程，自旋默认限定次数是 10。

自适应自旋?

JDK6 对自旋锁进行了优化，自旋时间不再固定，而是由前一次的自旋时间及锁拥有者的状态决定。

如果在同一个锁上，自旋刚刚成功获得过锁且持有锁的线程正在运行，虚拟机会认为这次自旋也很可能成功，进而允许自旋持续更久。如果自旋很少成功，以后获取锁时将可能直接省略掉自旋，避免浪费处理器资源。

有了自适应自旋，随着程序运行时间的增长，虚拟机对程序锁的状况预测就会越来越精准。

锁消除

锁消除指即时编译器对检测到不可能存在共享数据竞争的锁进行消除。

主要判定依据来源于逃逸分析，如果判断一段代码中堆上的所有数据都只被一个线程访问，就可以当作栈上的数据对待，认为它们是线程私有的而无须同步。

锁粗化

原则需要将同步块的作用范围限制得尽量小，只在共享数据的实际作用域中进行同步，这是为了使等待锁的线程尽快拿到锁。

但如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作是出现在循环体之内的，即使没有线程竞争也会导致不必要的性能消耗。因此如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，将会把同步的范围扩展到整个操作序列的外部。

偏向锁

偏向锁是为了在没有竞争的情况下减少锁开销，锁会偏向于第一个获得它的线程，如果在执行过程中锁一直没有被其他线程获取，则持有偏向锁的线程将不需要进行同步。

当锁对象第一次被线程获取时，虚拟机会将对象头中的偏向模式设为 1，同时使用 CAS 把获取到锁的线程 ID 记录在对象的 Mark Word 中。如果 CAS 成功，持有偏向锁的线程以后每次进入锁相关的同步块都不再进行任何同步操作。

一旦有其他线程尝试获取锁，偏向模式立即结束，根据锁对象是否处于锁定状态决定是否撤销偏向，后续同步按照轻量级锁那样执行。

轻量级锁

轻量级锁是为了在没有竞争的前提下减少重量级锁使用操作系统互斥量产生的性能消耗。

在代码即将进入同步块时，如果同步对象没有被锁定，虚拟机将在当前线程的栈帧中建立一个锁记录空间，存储锁对象目前 Mark Word 的拷贝。然后虚拟机使用 CAS 尝试把对象的 Mark Word 更新为指向锁记录的指针，如果更新成功即代表该线程拥有了锁，锁标志位将转变为 00，表示处于轻量级锁定状态。

如果更新失败就意味着至少存在一条线程与当前线程竞争。虚拟机检查对象的 Mark Word 是否指向当前线程的栈帧，如果是则说明当前线程已经拥有了锁，直接进入同步块继续执行，否则说明锁对象已经被其他线程抢占。如果出现两条以上线程争用同一个锁，轻量级锁就不再有效，将膨胀为重量级锁，锁标志状态变为 10，此时Mark Word 存储的就是指向重量级锁的指针，后面等待锁的线程也必须阻塞。

解锁同样通过 CAS 进行，如果对象 Mark Word 仍然指向线程的锁记录，就用 CAS 把对象当前的 Mark Word 和线程复制的 Mark Word 替换回来。假如替换成功同步过程就顺利完成了，如果失败则说明有其他线程尝试过获取该锁，就要在释放锁的同时唤醒被挂起的线程。

偏向锁、轻量级锁和重量级锁的区别

偏向锁的优点是加解锁不需要额外消耗，和执行非同步方法比仅存在纳秒级差距，缺点是如果存在锁竞争会带来额外锁撤销的消耗，适用只有一个线程访问同步代码块的场景。

轻量级锁的优点是竞争线程不阻塞，程序响应速度快，缺点是如果线程始终得不到锁会自旋消耗 CPU，适用追求响应时间、同步代码块执行快的场景。

重量级锁的优点是线程竞争不使用自旋不消耗CPU，缺点是线程会阻塞，响应时间慢，适应追求吞吐量、同步代码块执行慢的场景。

Lock 和 synchronized 的区别

Lock 接是 juc 包的顶层接口，基于Lock 接口，用户能够以非块结构来实现互斥同步，摆脱了语言特性束缚，在类库层面实现同步。Lock 并未用到 synchronized，而是利用了 volatile 的可见性。

重入锁 ReentrantLock 是 Lock 最常见的实现，与 synchronized 一样可重入，不过它增加了一些高级功能：

- **等待可中断：** 持有锁的线程长期不释放锁时，正在等待的线程可以选择放弃等待而处理其他事情。
- **公平锁：** 公平锁指多个线程在等待同一个锁时，必须按照申请锁的顺序来依次获得锁，而非公平锁不保证这一点，在锁被释放时，任何线程都有机会获得锁。synchronized 是非公平的，ReentrantLock 在默认情况下是非公平的，可以通过构造方法指定公平锁。一旦使用了公平锁，性能会急剧下降，影响吞吐量。
- **锁绑定多个条件：** 一个 ReentrantLock 可以同时绑定多个 Condition。synchronized 中锁对象的 `wait` 跟 `notify` 可以实现一个隐含条件，如果要和多个条件关联就不得不额外添加锁，而 ReentrantLock 可以多次调用 `newCondition` 创建多个条件。

一般优先考虑使用 synchronized：① synchronized 是语法层面的同步，足够简单。② Lock 必须确保在 finally 中释放锁，否则一旦抛出异常有可能永远不会释放锁。使用 synchronized 可以由 JVM 来确保即使出现异常锁也能正常释放。③ 尽管 JDK5 时 ReentrantLock 的性能优于 synchronized，但在 JDK6 进行锁优化后二者的性能基本持平。从长远来看 JVM 更容易针对synchronized 优化，因为 JVM 可以在线程和对象的元数据中记录 synchronized 中锁的相关信息，而使用 Lock 的话 JVM 很难得知具体哪些锁对象是由特定线程持有的。

ReentrantLock 的可重入实现

以非公平锁为例，通过 `nonfairTryAcquire` 方法获取锁，该方法增加了再次获取同步状态的处理逻辑：判断当前线程是否为获取锁的线程来决定获取是否成功，如果是获取锁的线程再次请求则将同步状态值增加并返回 true，表示获取同步状态成功。

成功获取锁的线程再次获取锁将增加同步状态值，释放同步状态时将减少同步状态值。如果锁被获取了 n 次，那么前 n-1 次 `tryRelease` 方法必须都返回 fasle，只有同步状态完全释放才能返回 true，该方法将同步状态是否为 0 作为最终释放条件，释放时将占有线程设置为null 并返回 true。

对于非公平锁只要 CAS 设置同步状态成功则表示当前线程获取了锁，而公平锁则不同。公平锁使用 `tryAcquire` 方法，该方法与`nonfairTryAcquire` 的唯一区别就是判断条件中多了对同步队列中当前节点是否有前驱节点的判断，如果该方法返回 true 表示有线程比当前线程更早请求锁，因此需要等待前驱线程获取并释放锁后才能获取锁。

读写锁

ReentrantLock 是排他锁，同一时刻只允许一个线程访问，读写锁在同一时刻允许多个读线程访问，在写线程访问时，所有的读写线程均阻塞。读写锁维护了一个读锁和一个写锁，通过分离读写锁使并发性相比排他锁有了很大提升。

读写锁依赖 AQS 来实现同步功能，读写状态就是其同步器的同步状态。读写锁的自定义同步器需要在同步状态，即一个 int 变量上维护多个读线程和一个写线程的状态。读写锁将变量切分成了两个部分，高 16 位表示读，低 16 位表示写。

写锁是可重入排他锁，如果当前线程已经获得了写锁则增加写状态，如果当前线程在获取写锁时，读锁已经被获取或者该线程不是已经获得写锁的线程则进入等待。写锁的释放与 ReentrantLock 的释放类似，每次释放减少写状态，当写状态为 0 时表示写锁已被释放。

读锁是可重入共享锁，能够被多个线程同时获取，在没有其他写线程访问时，读锁总会被成功获取。如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已被其他线程获取则进入等待。读锁每次释放会减少读状态，减少的值是（1<<16），读锁的释放是线程安全的。

锁降级指把持住当前拥有的写锁，再获取读锁，随后释放先前拥有的写锁。

锁降级中读锁的获取是必要的，这是为了保证数据可见性，如果当前线程不获取读锁而直接释放写锁，假设此刻另一个线程 A 获取写锁修改了数据，当前线程无法感知线程 A 的数据更新。如果当前线程获取读锁，遵循锁降级的步骤，A 将被阻塞，直到当前线程使用数据并释放读锁之后，线程 A 才能获取写锁进行数据更新。

AQS

AQS 队列同步器是用来构建锁或其他同步组件的基础框架，它使用一个 volatile int state 变量作为共享资源，如果线程获取资源失败，则进入同步队列等待；如果获取成功就执行临界区代码，释放资源时会通知同步队列中的等待线程。

同步器的主要使用方式是继承，子类通过继承同步器并实现它的抽象方法来管理同步状态，对同步状态进行更改需要使用同步器提供的 3个方法 `getState`、`setState` 和 `compareAndSetState`，它们保证状态改变是安全的。子类推荐被定义为自定义同步组件的静态内部类，同步器自身没有实现任何同步接口，它仅仅定义若干同步状态获取和释放的方法，同步器既支持独占式也支持共享式。

同步器是实现锁的关键，在锁的实现中聚合同步器，利用同步器实现锁的语义。锁面向使用者，定义了使用者与锁交互的接口，隐藏实现细节；同步器面向锁的实现者，简化了锁的实现方式，屏蔽了同步状态管理、线程排队、等待与唤醒等底层操作。

每当有新线程请求资源时都会进入一个等待队列，只有当持有锁的线程释放锁资源后该线程才能持有资源。等待队列通过双向链表实现，线程被封装在链表的 Node 节点中，Node 的等待状态包括：CANCELLED（线程已取消）、SIGNAL（线程需要唤醒）、CONDITION （线程正在等待）、PROPAGATE（后继节点会传播唤醒操作，只在共享模式下起作用）。

AQS 两种模式

独占模式表示锁只会被一个线程占用，其他线程必须等到持有锁的线程释放锁后才能获取锁，同一时间只能有一个线程获取到锁。

共享模式表示多个线程获取同一个锁有可能成功，ReadLock 就采用共享模式。

独占模式通过 acquire 和 release 方法获取和释放锁，共享模式通过 acquireShared 和 releaseShared 方法获取和释放锁。

AQS 独占式获取/释放锁的原理

获取同步状态时，调用 `acquire` 方法，维护一个同步队列，使用 `tryAcquire` 方法安全地获取线程同步状态，获取失败的线程会被构造同步节点并通过 `addWaiter` 方法加入到同步队列的尾部，在队列中自旋。之后调用 `acquireQueued` 方法使得该节点以死循环的方式获取同步状态，如果获取不到则阻塞，被阻塞线程的唤醒主要依靠前驱节点的出队或被中断实现，移出队列或停止自旋的条件是前驱节点是头节点且成功获取了同步状态。

头节点是成功获取到同步状态的节点，后继节点的线程被唤醒后需要检查自己的前驱节点是否是头节点。目的是维护同步队列的 FIFO 原则，节点和节点在循环检查的过程中基本不通信，而是简单判断自己的前驱是否为头节点，这样就使节点的释放规则符合 FIFO，并且也便于对过早通知的处理，过早通知指前驱节点不是头节点的线程由于中断被唤醒。

释放同步状态时，同步器调用 `tryRelease` 方法释放同步状态，然后调用 `unparkSuccessor` 方法唤醒头节点的后继节点，使后继节点重新尝试获取同步状态。

AQS 共享式式获取/释放锁的原理

获取同步状态时，调用 `acquireShared` 方法，该方法调用 `tryAcquireShared` 方法尝试获取同步状态，返回值为 int 类型，返回值不小于于 0 表示能获取同步状态。因此在共享式获取锁的自旋过程中，成功获取同步状态并退出自旋的条件就是该方法的返回值不小于0。

释放同步状态时，调用 `releaseShared` 方法，释放后会唤醒后续处于等待状态的节点。它和独占式的区别在于 `tryReleaseShared` 方法必须确保同步状态安全释放，通过循环 CAS 保证，因为释放同步状态的操作会同时来自多个线程。

线程

程的生命周期

NEW：新建状态，线程被创建且未启动，此时还未调用 `start` 方法。

RUNNABLE：Java 将操作系统中的就绪和运行两种状态统称为 RUNNABLE，此时线程有可能在等待时间片，也有可能在执行。

BLOCKED：阻塞状态，可能由于锁被其他线程占用、调用了 `sleep` 或 `join` 方法、执行了 `wait` 方法等。

WAITING：等待状态，该状态线程不会被分配 CPU 时间片，需要其他线程通知或中断。可能由于调用了无参的 `wait` 和 `join` 方法。

TIME_WAITING：限期等待状态，可以在指定时间内自行返回。导可能由于调用了带参的 `wait` 和 `join` 方法。

TERMINATED：终止状态，表示当前线程已执行完毕或异常退出。

线程的创建方式

① 继承 Thread 类并重写 run 方法。实现简单，但不符合里氏替换原则，不可以继承其他类。

② 实现 Runnable 接口并重写 run 方法。避免了单继承局限性，编程更加灵活，实现解耦。

③实现 Callable 接口并重写 call 方法。可以获取线程执行结果的返回值，并且可以抛出异常。

线程方法

① `sleep` 方法会导致当前线程进入休眠状态，与 `wait` 不同的是该方法不会释放锁资源，进入的是 TIMED-WAITING 状态。

② `yiled` 方法使当前线程让出 CPU 时间片给优先级相同或更高的线程，回到 RUNNABLE 状态，与其他线程一起重新竞争CPU时间片。

③ `join` 方法用于等待其他线程运行终止，如果当前线程调用了另一个线程的 join 方法，则当前线程进入阻塞状态，当另一个线程结束时当前线程才能从阻塞状态转为就绪态，等待获取CPU时间片。底层使用的是wait，也会释放锁。

守护线程

守护线程是一种支持型线程，可以通过 `setDaemon(true)` 将线程设置为守护线程，但必须在线程启动前设置。

守护线程被用于完成支持性工作，但在 JVM 退出时守护线程中的 finally 块不一定执行，因为 JVM 中没有非守护线程时需要立即退出，所有守护线程都将立即终止，不能靠在守护线程使用 finally 确保关闭资源。

线程通信的方式

命令式编程中线程的通信机制有两种，共享内存和消息传递。在共享内存的并发模型里线程间共享程序的公共状态，通过写-读内存中的公共状态进行隐式通信。在消息传递的并发模型里线程间没有公共状态，必须通过发送消息来显式通信。Java 并发采用共享内存模型，线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。

volatile 告知程序任何对变量的读需要从主内存中获取，写必须同步刷新回主内存，保证所有线程对变量访问的可见性。

synchronized 确保多个线程在同一时刻只能有一个处于方法或同步块中，保证线程对变量访问的原子性、可见性和有序性。

等待通知机制指一个线程 A 调用了对象的 `wait` 方法进入等待状态，另一线程 B 调用了对象的 `notify/notifyAll` 方法，线程 A 收到通知后结束阻塞并执行后序操作。对象上的 `wait` 和 `notify/notifyAll` 如同开关信号，完成等待方和通知方的交互。

如果一个线程执行了某个线程的 `join` 方法，这个线程就会阻塞等待执行了 `join` 方法的线程终止，这里涉及等待/通知机制。`join` 底层通过 `wait` 实现，线程终止时会调用自身的 `notifyAll` 方法，通知所有等待在该线程对象上的线程。

管道 IO 流用于线程间数据传输，媒介为内存。PipedOutputStream 和 PipedWriter 是输出流，相当于生产者，PipedInputStream 和 PipedReader 是输入流，相当于消费者。管道流使用一个默认大小为 1KB 的循环缓冲数组。输入流从缓冲数组读数据，输出流往缓冲数组中写数据。当数组已满时，输出流所在线程阻塞；当数组首次为空时，输入流所在线程阻塞。

ThreadLocal 是线程共享变量，但它可以为每个线程创建单独的副本，副本值是线程私有的，互相之间不影响。

线程池好处

降低资源消耗，复用已创建的线程，降低开销、控制最大并发数。

隔离线程环境，可以配置独立线程池，将较慢的线程与较快的隔离开，避免相互影响。

实现任务线程队列缓冲策略和拒绝机制。

实现某些与时间相关的功能，如定时执行、周期执行等。

线程池处理任务的流程

① 核心线程池未满，创建一个新的线程执行任务，此时 workCount < corePoolSize。

② 如果核心线程池已满，工作队列未满，将线程存储在工作队列，此时 workCount >= corePoolSize。

③ 如果工作队列已满，线程数小于最大线程数就创建一个新线程处理任务，此时 workCount < maximumPoolSize，这一步也需要获取全局锁。

④ 如果超过大小线程数，按照拒绝策略来处理任务，此时 workCount > maximumPoolSize。

线程池创建线程时，会将线程封装成工作线程 Worker，Worker 在执行完任务后还会循环获取工作队列中的任务来执行。

创建线程池的方法

可以通过 Executors 的静态工厂方法创建线程池：

① `newFixedThreadPool`，固定大小的线程池，核心线程数也是最大线程数，不存在空闲线程，keepAliveTime = 0。该线程池使用的工作队列是无界阻塞队列 LinkedBlockingQueue，适用于负载较重的服务器。

② `newSingleThreadExecutor`，使用单线程，相当于单线程串行执行所有任务，适用于需要保证顺序执行任务的场景。

③ `newCachedThreadPool`，maximumPoolSize 设置为 Integer 最大值，是高度可伸缩的线程池。该线程池使用的工作队列是没有容量的 SynchronousQueue，如果主线程提交任务的速度高于线程处理的速度，线程池会不断创建新线程，极端情况下会创建过多线程而耗尽CPU 和内存资源。适用于执行很多短期异步任务的小程序或负载较轻的服务器。

④ `newScheduledThreadPool`：线程数最大为 Integer 最大值，存在 OOM 风险。支持定期及周期性任务执行，适用需要多个后台线程执行周期任务，同时需要限制线程数量的场景。相比 Timer 更安全，功能更强，与 `newCachedThreadPool` 的区别是不回收工作线程。

⑤ `newWorkStealingPool`：JDK8 引入，创建持有足够线程的线程池支持给定的并行度，通过多个队列减少竞争。

创建线程池的参数

① corePoolSize：常驻核心线程数，如果为 0，当执行完任务没有任何请求时会消耗线程池；如果大于 0，即使本地任务执行完，核心线程也不会被销毁。该值设置过大会浪费资源，过小会导致线程的频繁创建与销毁。

② maximumPoolSize：线程池能够容纳同时执行的线程最大数，必须大于等于 1，如果与核心线程数设置相同代表固定大小线程池。

③ keepAliveTime：线程空闲时间，线程空闲时间达到该值后会被销毁，直到只剩下 corePoolSize 个线程为止，避免浪费内存资源。

④ unit：keepAliveTime 的时间单位。

⑤ workQueue：工作队列，当线程请求数大于等于 corePoolSize 时线程会进入阻塞队列。

⑥ threadFactory：线程工厂，用来生产一组相同任务的线程。可以给线程命名，有利于分析错误。

⑦ handler：拒绝策略，默认使用 AbortPolicy 丢弃任务并抛出异常，CallerRunsPolicy 表示重新尝试提交该任务，DiscardOldestPolicy 表示抛弃队列里等待最久的任务并把当前任务加入队列，DiscardPolicy 表示直接抛弃当前任务但不抛出异常。

关闭线程池

可以调用 `shutdown` 或 `shutdownNow` 方法关闭线程池，原理是遍历线程池中的工作线程，然后逐个调用线程的 `interrupt` 方法中断线程，无法响应中断的任务可能永远无法终止。

区别是 `shutdownNow` 首先将线程池的状态设为 STOP，然后尝试停止正在执行或暂停任务的线程，并返回等待执行任务的列表。而 `shutdown` 只是将线程池的状态设为 SHUTDOWN，然后中断没有正在执行任务的线程。

通常调用 `shutdown` 来关闭线程池，如果任务不一定要执行完可调用 `shutdownNow`。

线程池的选择策略

可以从以下角度分析：①任务性质：CPU 密集型、IO 密集型和混合型。②任务优先级。③任务执行时间。④任务依赖性：是否依赖其他资源，如数据库连接。

性质不同的任务可用不同规模的线程池处理，CPU 密集型任务应配置尽可能小的线程，如配置 N_{cpu}+1 个线程的线程池。由于 IO 密集型任务线程并不是一直在执行任务，应配置尽可能多的线程，如 2*N_{cpu}。混合型的任务，如果可以拆分，将其拆分为一个 CPU 密集型任务和一个 IO 密集型任务，只要两个任务执行的时间相差不大那么分解后的吞吐量将高于串行执行的吞吐量，如果相差太大则没必要分解。

优先级不同的任务可以使用优先级队列 PriorityBlockingQueue 处理。

执行时间不同的任务可以交给不同规模的线程池处理，或者使用优先级队列让执行时间短的任务先执行。

依赖数据库连接池的任务，由于线程提交 SQL 后需要等待数据库返回的结果，等待的时间越长 CPU 空闲的时间就越长，因此线程数应该尽可能地设置大一些，提高 CPU 的利用率。

建议使用有界队列，能增加系统的稳定性和预警能力，可以根据需要设置的稍微大一些。

阻塞队列

阻塞队列支持阻塞插入和移除，当队列满时，阻塞插入元素的线程直到队列不满。当队列为空时，获取元素的线程会被阻塞直到队列非空。阻塞队列常用于生产者和消费者的场景，阻塞队列就是生产者用来存放元素，消费者用来获取元素的容器。

Java 中的阻塞队列

ArrayBlockingQueue，由数组组成的有界阻塞队列，默认情况下不保证线程公平，有可能先阻塞的线程最后才访问队列。

LinkedBlockingQueue，由链表结构组成的有界阻塞队列，队列的默认和最大长度为 Integer 最大值。

PriorityBlockingQueue，支持优先级的无界阻塞队列，默认情况下元素按照升序排序。可自定义 `compareTo` 方法指定排序规则，或者初始化时指定 Comparator 排序，不能保证同优先级元素的顺序。

DelayQueue，支持延时获取元素的无界阻塞队列，使用优先级队列实现。创建元素时可以指定多久才能从队列中获取当前元素，只有延迟期满时才能从队列中获取元素，适用于缓存和定时调度。

SynchronousQueue，不存储元素的阻塞队列，每一个 put 必须等待一个 take。默认使用非公平策略，也支持公平策略，适用于传递性场景，吞吐量高。

LinkedTransferQueue，链表组成的无界阻塞队列，相对于其他阻塞队列多了 `tryTransfer` 和 `transfer` 方法。`transfer`方法：如果当前有消费者正等待接收元素，可以把生产者传入的元素立刻传输给消费者，否则会将元素放在队列的尾节点并等到该元素被消费者消费才返回。`tryTransfer` 方法用来试探生产者传入的元素能否直接传给消费者，如果没有消费者等待接收元素则返回 false，和 `transfer` 的区别是无论消费者是否消费都会立即返回。

LinkedBlockingDeque，链表组成的双向阻塞队列，可从队列的两端插入和移出元素，多线程同时入队时减少了竞争。

实现原理

使用通知模式实现，生产者往满的队列里添加元素时会阻塞，当消费者消费后，会通知生产者当前队列可用。当往队列里插入一个元素，如果队列不可用，阻塞生产者主要通过 LockSupport 的 `park` 方法实现，不同操作系统中实现方式不同，在 Linux 下使用的是系统方法 `pthread_cond_wait` 实现。

ThreadLocal

ThreadLoacl 是线程共享变量，主要用于一个线程内跨类、方法传递数据。ThreadLoacl 有一个静态内部类 ThreadLocalMap，其 Key 是 ThreadLocal 对象，值是 Entry 对象，Entry 中只有一个 Object 类的 vaule 值。ThreadLocal 是线程共享的，但 ThreadLocalMap 是每个线程私有的。ThreadLocal 主要有 set、get 和 remove 三个方法。

set 方法

首先获取当前线程，然后再获取当前线程对应的 ThreadLocalMap 类型的对象 map。如果 map 存在就直接设置值，key 是当前的 ThreadLocal 对象，value 是传入的参数。

如果 map 不存在就通过 `createMap` 方法为当前线程创建一个 ThreadLocalMap 对象再设置值。

get 方法

首先获取当前线程，然后再获取当前线程对应的 ThreadLocalMap 类型的对象 map。如果 map 存在就以当前 ThreadLocal 对象作为 key 获取 Entry 类型的对象 e，如果 e 存在就返回它的 value 属性。

如果 e 不存在或者 map 不存在，就调用 `setInitialValue` 方法先为当前线程创建一个 ThreadLocalMap 对象然后返回默认的初始值 null。

remove 方法

首先通过当前线程获取其对应的 ThreadLocalMap 类型的对象 m，如果 m 不为空，就解除 ThreadLocal 这个 key 及其对应的 value 值的联系。

存在的问题

线程复用会产生脏数据，由于线程池会重用 Thread 对象，因此与 Thread 绑定的 ThreadLocal 也会被重用。如果没有调用 remove 清理与线程相关的 ThreadLocal 信息，那么假如下一个线程没有调用 set 设置初始值就可能 get 到重用的线程信息。

ThreadLocal 还存在内存泄漏的问题，由于 ThreadLocal 是弱引用，但 Entry 的 value 是强引用，因此当 ThreadLocal 被垃圾回收后，value 依旧不会被释放。因此需要及时调用 remove 方法进行清理操作。

JUC

CAS

CAS 表示 Compare And Swap，比较并交换，CAS 需要三个操作数，分别是内存位置 V、旧的预期值 A 和准备设置的新值 B。CAS 指令执行时，当且仅当 V 符合 A 时，处理器才会用 B 更新 V 的值，否则它就不执行更新。但不管是否更新都会返回 V 的旧值，这些处理过程是原子操作，执行期间不会被其他线程打断。

在 JDK 5 后，Java 类库中才开始使用 CAS 操作，该操作由 Unsafe 类里的 `compareAndSwapInt` 等几个方法包装提供。HotSpot 在内部对这些方法做了特殊处理，即时编译的结果是一条平台相关的处理器 CAS 指令。Unsafe 类不是给用户程序调用的类，因此 JDK9 前只有 Java 类库可以使用 CAS，譬如 juc 包里的 AtomicInteger类中 `compareAndSet` 等方法都使用了Unsafe 类的 CAS 操作实现。

CAS 从语义上来说存在一个逻辑漏洞：如果 V 初次读取时是 A，并且在准备赋值时仍为 A，这依旧不能说明它没有被其他线程更改过，因为这段时间内假设它的值先改为 B 又改回 A，那么 CAS 操作就会误认为它从来没有被改变过。

这个漏洞称为 ABA 问题，juc 包提供了一个 AtomicStampedReference，原子更新带有版本号的引用类型，通过控制变量值的版本来解决 ABA 问题。大部分情况下 ABA 不会影响程序并发的正确性，如果需要解决，传统的互斥同步可能会比原子类更高效。

原子类

JDK 5 提供了 java.util.concurrent.atomic 包，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。到 JDK 8 该包共有17个类，依据作用分为四种：原子更新基本类型类、原子更新数组类、原子更新引用类以及原子更新字段类，atomic 包里的类基本都是使用 Unsafe 实现的包装类。

AtomicInteger 原子更新整形、 AtomicLong 原子更新长整型、AtomicBoolean 原子更新布尔类型。

AtomicIntegerArray，原子更新整形数组里的元素、 AtomicLongArray 原子更新长整型数组里的元素、 AtomicReferenceArray 原子更新引用类型数组里的元素。

AtomicReference 原子更新引用类型、AtomicMarkableReference 原子更新带有标记位的引用类型，可以绑定一个 boolean 标记、 AtomicStampedReference 原子更新带有版本号的引用类型，关联一个整数值作为版本号，解决 ABA 问题。

AtomicIntegerFieldUpdater 原子更新整形字段的更新器、 AtomicLongFieldUpdater 原子更新长整形字段的更新器AtomicReferenceFieldUpdater 原子更新引用类型字段的更新器。

AtomicIntger 实现原子更新的原理

AtomicInteger 原子更新整形、 AtomicLong 原子更新长整型、AtomicBoolean 原子更新布尔类型。

`getAndIncrement` 以原子方式将当前的值加 1，首先在 for 死循环中取得 AtomicInteger 里存储的数值，第二步对 AtomicInteger 当前的值加 1 ，第三步调用 `compareAndSet` 方法进行原子更新，先检查当前数值是否等于 expect，如果等于则说明当前值没有被其他线程修改，则将值更新为 next，否则会更新失败返回 false，程序会进入 for 循环重新进行 `compareAndSet` 操作。

atomic 包中只提供了三种基本类型的原子更新，atomic 包里的类基本都是使用 Unsafe 实现的，Unsafe 只提供三种 CAS 方法：`compareAndSwapInt`、`compareAndSwapLong` 和 `compareAndSwapObject`，例如原子更新 Boolean 是先转成整形再使用 `compareAndSwapInt` 。

CountDownLatch

CountDownLatch 是基于执行时间的同步类，允许一个或多个线程等待其他线程完成操作，构造方法接收一个 int 参数作为计数器，如果要等待 n 个点就传入 n。每次调用 `countDown` 方法时计数器减 1，`await` 方法会阻塞当前线程直到计数器变为0，由于 `countDown` 方法可用在任何地方，所以 n 个点既可以是 n 个线程也可以是一个线程里的 n 个执行步骤。

CyclicBarrier

循环屏障是基于同步到达某个点的信号量触发机制，作用是让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障才会解除。构造方法中的参数表示拦截线程数量，每个线程调用 `await` 方法告诉 CyclicBarrier 自己已到达屏障，然后被阻塞。还支持在构造方法中传入一个 Runnable 任务，当线程到达屏障时会优先执行该任务。适用于多线程计算数据，最后合并计算结果的应用场景。

CountDownLacth 的计数器只能用一次，而 CyclicBarrier 的计数器可使用 `reset` 方法重置，所以 CyclicBarrier 能处理更为复杂的业务场景，例如计算错误时可用重置计数器重新计算。

Semaphore

信号量用来控制同时访问特定资源的线程数量，通过协调各个线程以保证合理使用公共资源。信号量可以用于流量控制，特别是公共资源有限的应用场景，比如数据库连接。

Semaphore 的构造方法参数接收一个 int 值，表示可用的许可数量即最大并发数。使用 `acquire` 方法获得一个许可证，使用 `release` 方法归还许可，还可以用 `tryAcquire` 尝试获得许可。

Exchanger

交换者是用于线程间协作的工具类，用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。

两个线程通过 `exchange` 方法交换数据，第一个线程执行 `exchange` 方法后会阻塞等待第二个线程执行该方法，当两个线程都到达同步点时这两个线程就可以交换数据，将本线程生产出的数据传递给对方。应用场景包括遗传算法、校对工作等。

JDK7 的 ConcurrentHashMap 原理

ConcurrentHashMap 用于解决 HashMap 的线程不安全和 HashTable 的并发效率低，HashTable 之所以效率低是因为所有线程都必须竞争同一把锁，假如容器里有多把锁，每一把锁用于锁容器的部分数据，那么多线程访问容器不同数据段的数据时，线程间就不会存在锁竞争，从而有效提高并发效率，这就是 ConcurrentHashMap 的锁分段技术。首先将数据分成 Segment 数据段，然后给每一个数据段配一把锁，当一个线程占用锁访问其中一个段的数据时，其他段的数据也能被其他线程访问。

get 实现简单高效，先经过一次再散列，再用这个散列值通过散列运算定位到 Segment，最后通过散列算法定位到元素。get 的高效在于不需要加锁，除非读到空值才会加锁重读。get 方法中将共享变量定义为 volatile，在 get 操作里只需要读所以不用加锁。

put 必须加锁，首先定位到 Segment，然后进行插入操作，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置，然后将其放入数组。

size 操作用于统计元素的数量，必须统计每个 Segment 的大小然后求和，在统计结果累加的过程中，之前累加过的 count 变化几率很小，因此先尝试两次通过不加锁的方式统计结果，如果统计过程中容器大小发生了变化，再加锁统计所有 Segment 大小。判断容器是否发生变化根据 modCount 确定。

JDK8 的 ConcurrentHashMap 原理

主要对 JDK7 做了三点改造：① 取消分段锁机制，进一步降低冲突概率。② 引入红黑树结构，同一个哈希槽上的元素个数超过一定阈值后，单向链表改为红黑树结构。③ 使用了更加优化的方式统计集合内的元素数量。具体优化表现在：在 put、resize 和 size 方法中设计元素总数的更新和计算都避免了锁，使用 CAS 代替。

get 同样不需要同步，put 操作时如果没有出现哈希冲突，就使用 CAS 添加元素，否则使用 synchronized 加锁添加元素。

当某个槽内的元素个数达到 7 且 table 容量不小于 64 时，链表转为红黑树。当某个槽内的元素减少到 6 时，由红黑树重新转为链表。在转化过程中，使用同步块锁住当前槽的首元素，防止其他线程对当前槽进行增删改操作，转化完成后利用 CAS 替换原有链表。由于 TreeNode 节点也存储了 next 引用，因此红黑树转为链表很简单，只需从 first 元素开始遍历所有节点，并把节点从 TreeNode 转为 Node 类型即可，当构造好新链表后同样用 CAS 替换红黑树。

ArrayList 的线程安全集合

可以使用 CopyOnWriteArrayList 代替 ArrayList，它实现了读写分离。写操作复制一个新的集合，在新集合内添加或删除元素，修改完成后再将原集合的引用指向新集合。这样做的好处是可以高并发地进行读写操作而不需要加锁，因为当前集合不会添加任何元素。使用时注意尽量设置容量初始值，并且可以使用批量添加或删除，避免多次扩容，比如只增加一个元素却复制整个集合。

适合读多写少，单个添加时效率极低。CopyOnWriteArrayList 是 fail-safe 的，并发包的集合都是这种机制，fail-safe 在安全的副本上遍历，集合修改与副本遍历没有任何关系，缺点是无法读取最新数据。这也是 CAP 理论中 C 和 A 的矛盾，即一致性与可用性的矛盾。

框架

Spring IoC

IoC

IoC 即控制反转，简单来说就是把原来代码里需要实现的对象创建、依赖反转给容器来帮忙实现，需要创建一个容器并且需要一种描述让容器知道要创建的对象间的关系，在 Spring 中管理对象及其依赖关系是通过 Spring 的 IoC 容器实现的。

IoC 的实现方式有依赖注入和依赖查找，由于依赖查找使用的很少，因此 IoC 也叫做依赖注入。依赖注入指对象被动地接受依赖类而不用自己主动去找，对象不是从容器中查找它依赖的类，而是在容器实例化对象时主动将它依赖的类注入给它。假设一个 Car 类需要一个 Engine 的对象，那么一般需要需要手动 new 一个 Engine，利用 IoC 就只需要定义一个私有的 Engine 类型的成员变量，容器会在运行时自动创建一个 Engine 的实例对象并将引用自动注入给成员变量。

基于 XML 的容器初始化

当创建一个 ClassPathXmlApplicationContext 时，构造方法做了两件事：① 调用父容器的构造方法为容器设置好 Bean 资源加载器。② 调用父类的 `setConfigLocations` 方法设置 Bean 配置信息的定位路径。

ClassPathXmlApplicationContext 通过调用父类 AbstractApplicationContext 的 `refresh` 方法启动整个 IoC 容器对 Bean 定义的载入过程，`refresh` 是一个模板方法，规定了 IoC 容器的启动流程。在创建 IoC 容器前如果已有容器存在，需要把已有的容器销毁，保证在 `refresh` 方法后使用的是新创建的 IoC 容器。

容器创建后通过 `loadBeanDefinitions` 方法加载 Bean 配置资源，该方法做两件事：① 调用资源加载器的方法获取要加载的资源。② 真正执行加载功能，由子类 XmlBeanDefinitionReader 实现。加载资源时首先解析配置文件路径，读取配置文件的内容，然后通过 XML 解析器将 Bean 配置信息转换成文档对象，之后按照 Spring Bean 的定义规则对文档对象进行解析。

Spring IoC 容器中注册解析的 Bean 信息存放在一个 HashMap 集合中，key 是字符串，值是 BeanDefinition，注册过程中需要使用 synchronized 保证线程安全。当配置信息中配置的 Bean 被解析且被注册到 IoC 容器中后，初始化就算真正完成了，Bean 定义信息已经可以使用且可被检索。Spring IoC 容器的作用就是对这些注册的 Bean 定义信息进行处理和维护，注册的 Bean 定义信息是控制反转和依赖注入的基础。

基于注解的容器初始化

分为两种：① 直接将注解 Bean 注册到容器中，可以在初始化容器时注册，也可以在容器创建之后手动注册，然后刷新容器使其对注册的注解 Bean 进行处理。② 通过扫描指定的包及其子包的所有类处理，在初始化注解容器时指定要自动扫描的路径。

依赖注入的实现方法

构造方法注入： IoC Service Provider 会检查被注入对象的构造方法，取得它所需要的依赖对象列表，进而为其注入相应的对象。这种方法的优点是在对象构造完成后就处于就绪状态，可以马上使用。缺点是当依赖对象较多时，构造方法的参数列表会比较长，构造方法无法被继承，无法设置默认值。对于非必需的依赖处理可能需要引入多个构造方法，参数数量的变动可能会造成维护的困难。

setter 方法注入： 当前对象只需要为其依赖对象对应的属性添加 setter 方法，就可以通过 setter 方法将依赖对象注入到被依赖对象中。setter 方法注入在描述性上要比构造方法注入强，并且可以被继承，允许设置默认值。缺点是无法在对象构造完成后马上进入就绪状态。

接口注入： 必须实现某个接口，接口提供方法来为其注入依赖对象。使用少，因为它强制要求被注入对象实现不必要接口，侵入性强。

依赖注入的相关注解

`@Autowired`：自动按类型注入，如果有多个匹配则按照指定 Bean 的 id 查找，查找不到会报错。

`@Qualifier`：在自动按照类型注入的基础上再按照 Bean 的 id 注入，给变量注入时必须搭配 `@Autowired`，给方法注入时可单独使用。

`@Resource`：直接按照 Bean 的 id 注入，只能注入 Bean 类型。

`@Value`：用于注入基本数据类型和 String 类型。

依赖注入的过程

`getBean` 方法获取 Bean 实例，该方法会调用 `doGetBean` ，`doGetBean` 真正实现从 IoC 容器获取 Bean 的功能，也是触发依赖注入的地方。

具体创建 Bean 对象的过程由 ObjectFactory 的 `createBean` 完成，该方法主要通过 `createBeanInstance` 方法生成 Bean 包含的 Java 对象实例和 `populateBean` 方法对 Bean 属性的依赖注入进行处理。

在 `populateBean`方法中，注入过程主要分为两种情况：① 属性值类型不需要强制转换时，不需要解析属性值，直接进行依赖注入。② 属性值类型需要强制转换时，首先解析属性值，然后对解析后的属性值进行依赖注入。依赖注入的过程就是将 Bean 对象实例设置到它所依赖的 Bean 对象属性上，真正的依赖注入是通过 `setPropertyValues` 方法实现的，该方法使用了委派模式。

BeanWrapperImpl 类负责对完成初始化的 Bean 对象进行依赖注入，对于非集合类型属性，使用 JDK 反射，通过属性的 setter 方法为属性设置注入后的值。对于集合类型的属性，将属性值解析为目标类型的集合后直接赋值给属性。

当容器对 Bean 的定位、载入、解析和依赖注入全部完成后就不再需要手动创建对象，IoC 容器会自动为我们创建对象并且注入依赖。

Bean 的生命周期

在 IoC 容器的初始化过程中会对 Bean 定义完成资源定位，加载读取配置并解析，最后将解析的 Bean 信息放在一个 HashMap 集合中。当 IoC 容器初始化完成后，会进行对 Bean 实例的创建和依赖注入过程，注入对象依赖的各种属性值，在初始化时可以指定自定义的初始化方法。经过这一系列初始化操作后 Bean 达到可用状态，接下来就可以使用 Bean 了，当使用完成后会调用 destroy 方法进行销毁，此时也可以指定自定义的销毁方法，最终 Bean 被销毁且从容器中移除。

XML 方式通过配置 bean 标签中的 init-Method 和 destory-Method 指定自定义初始化和销毁方法。

注解方式通过配置 `@Bean` 注解中的 init-Method 和 destory-Method 指定自定义初始化和销毁方法。

Bean 的作用范围

通过 scope 属性指定 bean 的作用范围，包括：

① singleton：单例模式，是默认作用域，不管收到多少 Bean 请求每个容器中只有一个唯一的 Bean 实例。

② prototype：原型模式，和 singleton 相反，每次 Bean 请求都会创建一个新的实例。

③ request：每次 HTTP 请求都会创建一个新的 Bean 并把它放到 request 域中，在请求完成后 Bean 会失效并被垃圾收集器回收。

④ session：和 request 类似，确保每个 session 中有一个 Bean 实例，session 过期后 bean 会随之失效。

⑤ global session：当应用部署在 Portlet 容器时，如果想让所有 Portlet 共用全局存储变量，那么该变量需要存储在 global session 中。

通过 XML 方式创建 Bean

默认无参构造方法，只需要指明 bean 标签中的 id 和 class 属性，如果没有无参构造方法会报错。

静态工厂方法，通过 bean 标签中的 class 属性指明静态工厂，factory-method 属性指明静态工厂方法。

实例工厂方法，通过 bean 标签中的 factory-bean 属性指明实例工厂，factory-method 属性指明实例工厂方法。

通过注解创建 Bean

`@Component` 把当前类对象存入 Spring 容器中，相当于在 xml 中配置一个 bean 标签。value 属性指定 bean 的 id，默认使用当前类的首字母小写的类名。

`@Controller`，`@Service`，`@Repository` 三个注解都是 `@Component` 的衍生注解，作用及属性都是一模一样的。只是提供了更加明确语义，`@Controller` 用于表现层，`@Service` 用于业务层，`@Repository` 用于持久层。如果注解中有且只有一个 value 属性要赋值时可以省略 value。

如果想将第三方的类变成组件又没有源代码，也就没办法使用 `@Component` 进行自动配置，这种时候就要使用 `@Bean` 注解。被 `@Bean` 注解的方法返回值是一个对象，将会实例化，配置和初始化一个新对象并返回，这个对象由 Spring 的 IoC 容器管理。name 属性用于给当前 `@Bean` 注解方法创建的对象指定一个名称，即 bean 的 id。当使用注解配置方法时，如果方法有参数，Spring 会去容器查找是否有可用 bean对象，查找方式和 `@Autowired` 一样。

注解配置文件

`@Configuration` 用于指定当前类是一个 spring 配置类，当创建容器时会从该类上加载注解，value 属性用于指定配置类的字节码。

`@ComponentScan` 用于指定 Spring 在初始化容器时要扫描的包。basePackages 属性用于指定要扫描的包。

`@PropertySource` 用于加载 `.properties` 文件中的配置。value 属性用于指定文件位置，如果是在类路径下需要加上 classpath。

`@Import` 用于导入其他配置类，在引入其他配置类时可以不用再写 `@Configuration` 注解。有 `@Import` 的是父配置类，引入的是子配置类。value 属性用于指定其他配置类的字节码。

BeanFactory、FactoryBean 和 ApplicationContext 的区别

BeanFactory 是一个 Bean 工厂，使用简单工厂模式，是 Spring IoC 容器顶级接口，可以理解为含有 Bean 集合的工厂类，作用是管理 Bean，包括实例化、定位、配置对象及建立这些对象间的依赖。BeanFactory 实例化后并不会自动实例化 Bean，只有当 Bean 被使用时才实例化与装配依赖关系，属于延迟加载，适合多例模式。

FactoryBean 是一个工厂 Bean，使用了工厂方法模式，作用是生产其他 Bean 实例，可以通过实现该接口，提供一个工厂方法来自定义实例化 Bean 的逻辑。FactoryBean 接口由 BeanFactory 中配置的对象实现，这些对象本身就是用于创建对象的工厂，如果一个 Bean 实现了这个接口，那么它就是创建对象的工厂 Bean，而不是 Bean 实例本身。

ApplicationConext 是 BeanFactory 的子接口，扩展了 BeanFactory 的功能，提供了支持国际化的文本消息，统一的资源文件读取方式，事件传播以及应用层的特别配置等。容器会在初始化时对配置的 Bean 进行预实例化，Bean 的依赖注入在容器初始化时就已经完成，属于立即加载，适合单例模式，一般推荐使用。

Spring AOP

AOP

AOP 即面向切面编程，简单地说就是将代码中重复的部分抽取出来，在需要执行的时候使用动态代理技术，在不修改源码的基础上对方法进行增强。

Spring 根据类是否实现接口来判断动态代理方式，如果实现接口会使用 JDK 的动态代理，核心是 InvocationHandler 接口和 Proxy 类，如果没有实现接口会使用 CGLib 动态代理，CGLib 是在运行时动态生成某个类的子类，如果某个类被标记为 final，不能使用 CGLib 。

JDK 动态代理主要通过重组字节码实现，首先获得被代理对象的引用和所有接口，生成新的类必须实现被代理类的所有接口，动态生成Java 代码后编译新生成的 `.class` 文件并重新加载到 JVM 运行。JDK 代理直接写 Class 字节码，CGLib 是采用 ASM 框架写字节码，生成代理类的效率低。但是 CGLib 调用方法的效率高，因为 JDK 使用反射调用方法，CGLib 使用 FastClass 机制为代理类和被代理类各生成一个类，这个类会为代理类或被代理类的方法生成一个 index，这个 index 可以作为参数直接定位要调用的方法。

常用场景包括权限认证、自动缓存、错误处理、日志、调试和事务等。

相关注解

`@Aspect`：声明被注解的类是一个切面 Bean。

`@Before`：前置通知，指在某个连接点之前执行的通知。

`@After`：后置通知，指某个连接点退出时执行的通知（不论正常返回还是异常退出）。

`@AfterReturning`：返回后通知，指某连接点正常完成之后执行的通知，返回值使用returning属性接收。

`@AfterThrowing`：异常通知，指方法抛出异常导致退出时执行的通知，和`@AfterReturning`只会有一个执行，异常使用throwing属性接收。

相关术语

`Aspect`：切面，一个关注点的模块化，这个关注点可能会横切多个对象。

`Joinpoint`：连接点，程序执行过程中的某一行为，即业务层中的所有方法。。

`Advice`：通知，指切面对于某个连接点所产生的动作，包括前置通知、后置通知、返回后通知、异常通知和环绕通知。

`Pointcut`：切入点，指被拦截的连接点，切入点一定是连接点，但连接点不一定是切入点。

`Proxy`：代理，Spring AOP 中有 JDK 动态代理和 CGLib 代理，目标对象实现了接口时采用 JDK 动态代理，反之采用 CGLib 代理。

`Target`：代理的目标对象，指一个或多个切面所通知的对象。

`Weaving`：织入，指把增强应用到目标对象来创建代理对象的过程。

AOP 的过程

Spring AOP 由 BeanPostProcessor 后置处理器开始，这个后置处理器是一个监听器，可以监听容器触发的 Bean 生命周期事件，向容器注册后置处理器以后，容器中管理的 Bean 就具备了接收 IoC 容器回调事件的能力。BeanPostProcessor 的调用发生在 Spring IoC 容器完成 Bean 实例对象的创建和属性的依赖注入后，为 Bean 对象添加后置处理器的入口是 `initializeBean` 方法。

Spring 中 JDK 动态代理通过 JdkDynamicAopProxy 调用 Proxy 的 `newInstance` 方法来生成代理类，JdkDynamicAopProxy 也实现了 InvocationHandler 接口，`invoke` 方法的具体逻辑是先获取应用到此方法上的拦截器链，如果有拦截器则创建 MethodInvocation 并调用其 `proceed` 方法，否则直接反射调用目标方法。因此 Spring AOP 对目标对象的增强是通过拦截器实现的。

Spring MVC

处理流程

Web 容器启动时会通知 Spring 初始化容器，加载 Bean 的定义信息并初始化所有单例 Bean，然后遍历容器中的 Bean，获取每一个 Controller 中的所有方法访问的 URL，将 URL 和对应的 Controller 保存到一个 Map 集合中。

所有的请求会转发给 DispatcherServlet 前端处理器处理，DispatcherServlet 会请求 HandlerMapping 找出容器中被 `@Controler` 注解修饰的 Bean 以及被 `@RequestMapping` 修饰的方法和类，生成 Handler 和 HandlerInterceptor 并以一个 HandlerExcutionChain 处理器执行链的形式返回。

之后 DispatcherServlet 使用 Handler 找到对应的 HandlerApapter，通过 HandlerApapter 调用 Handler 的方法，将请求参数绑定到方法的形参上，执行方法处理请求并得到 ModelAndView。

最后 DispatcherServlet 根据使用 ViewResolver 试图解析器对得到的 ModelAndView 逻辑视图进行解析得到 View 物理视图，然后对视图渲染，将数据填充到视图中并返回给客户端。

组件

`DispatcherServlet`：SpringMVC 中的前端控制器，是整个流程控制的核心，负责接收请求并转发给对应的处理组件。

`Handler`：处理器，完成具体业务逻辑，相当于 Servlet 或 Action。

`HandlerMapping`：完成 URL 到 Controller 映射，DispatcherServlet 通过 HandlerMapping 将不同请求映射到不同 Handler。

`HandlerInterceptor`：处理器拦截器，是一个接口，如果需要完成一些拦截处理，可以实现该接口。

`HandlerExecutionChain`：处理器执行链，包括两部分内容：Handler 和 HandlerInterceptor。

`HandlerAdapter`：处理器适配器，Handler执行业务方法前需要进行一系列操作，包括表单数据验证、数据类型转换、将表单数据封装到JavaBean等，这些操作都由 HandlerAdapter 完成。DispatcherServlet 通过 HandlerAdapter 来执行不同的 Handler。

`ModelAndView`：装载模型数据和视图信息，作为 Handler 处理结果返回给 DispatcherServlet。

`ViewResolver`：视图解析器，DispatcherServlet 通过它将逻辑视图解析为物理视图，最终将渲染的结果响应给客户端。

相关注解

`@Controller`：在类定义处添加，将类交给IoC容器管理。

`@RequtestMapping`：将URL请求和业务方法映射起来，在类和方法定义上都可以添加该注解。`value` 属性指定URL请求的实际地址，是默认值。`method` 属性限制请求的方法类型，包括GET、POST、PUT、DELETE等。如果没有使用指定的请求方法请求URL，会报405 Method Not Allowed 错误。`params` 属性限制必须提供的参数，如果没有会报错。

`@RequestParam`：如果 Controller 方法的形参和 URL 参数名一致可以不添加注解，如果不一致可以使用该注解绑定。`value` 属性表示HTTP请求中的参数名。`required` 属性设置参数是否必要，默认 false。`defaultValue` 属性指定没有给参数赋值时的默认值。

`@PathVariable`：Spring MVC 支持 RESTful 风格 URL，通过 `@PathVariable` 完成请求参数与形参的绑定。

Spring Data JPA

ORM

ORM 即 Object-Relational Mapping ，表示对象关系映射，映射的不只是对象的值还有对象之间的关系，通过 ORM 就可以把对象映射到关系型数据库中。操作实体类就相当于操作数据库表，可以不再重点关注 SQL 语句。

JPA 的使用

只需要持久层接口继承 JpaRepository 即可，泛型参数列表中第一个参数是实体类类型，第二个参数是主键类型。

运行时通过 `JdkDynamicAopProxy` 的 `invoke` 方法创建了一个动态代理对象 `SimpleJpaRepository`，`SimpleJpaRepository` 中封装了 JPA 的操作，通过 `hibernate`（封装了 JDBC）完成数据库操作。

实体类相关注解

`@Entity`：表明当前类是一个实体类。

`@Table` ： 关联实体类和数据库表。

`@Column` ： 关联实体类属性和数据库表中字段。

`@Id` ： 声明当前属性为数据库表主键对应的属性。

`@GeneratedValue`： 配置主键生成策略。

`@OneToMany` ： 配置一对多关系，mappedBy 属性值为主表实体类在从表实体类中对应的属性名。

`@ManyToOne` ： 配置多对一关系，targetEntity 属性值为主表对应实体类的字节码。

`@JoinColumn`：配置外键关系，name 属性值为外键名称，referencedColumnName 属性值为主表主键名称。

对象导航查询

通过 get 方法查询一个对象的同时，通过此对象可以查询它的关联对象。

对象导航查询一到多默认使用延迟加载的形式， 关联对象是集合，因此使用立即加载可能浪费资源。

对象导航查询多到一默认使用立即加载的形式， 关联对象是一个对象，因此使用立即加载。

如果要改变加载方式，在实体类注解配置加上 fetch 属性即可，LAZY 表示延迟加载，EAGER 表示立即加载。

Mybatis

Mybatis 的优缺点

优点

相比 JDBC 减少了大量代码量，减少冗余代码。

使用灵活，SQL 语句写在 XML 里，从程序代码中彻底分离，降低了耦合度，便于管理。

提供 XML 标签，支持编写动态 SQL 语句。

提供映射标签，支持对象与数据库的 ORM 字段映射关系。

缺点

SQL 语句编写工作量较大，尤其是字段和关联表多时。

SQL 语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

Mybatis 的 XML 文件标签属性

`select`、`insert`、`update`、`delete` 标签分别对应查询、添加、更新、删除操作。

`parameterType` 属性表示参数的数据类型，包括基本数据类型和对应的包装类型、String 和 Java Bean 类型，当有多个参数时可以使用 `#{argn}` 的形式表示第 n 个参数。除了基本数据类型都要以全限定类名的形式指定参数类型。

`resultType` 表示返回的结果类型，包括基本数据类型和对应的包装类型、String 和 Java Bean 类型。还可以使用把返回结果封装为复杂类型的 `resultMap` 。

Mybatis 的一级缓存

一级缓存是 SqlSession 级别，默认开启且不能关闭。

操作数据库时需要创建 SqlSession 对象，对象中有一个 HashMap 存储缓存数据，不同 SqlSession 之间缓存数据区域互不影响。

一级缓存的作用域是 SqlSession 范围的，在同一个 SqlSession 中执行两次相同的 SQL 语句时，第一次执行完毕会将结果保存在缓存中，第二次查询直接从缓存中获取。

如果 SqlSession 执行了 DML 操作（insert、update、delete），Mybatis 必须将缓存清空保证数据有效性。

Mybatis 的二级缓存

二级缓存是Mapper 级别，默认关闭。

使用二级缓存时多个 SqlSession 使用同一个 Mapper 的 SQL 语句操作数据库，得到的数据会存在二级缓存区，同样使用 HashMap 进行数据存储，相比于一级缓存，二级缓存范围更大，多个 SqlSession 可以共用二级缓存，作用域是 Mapper 的同一个 namespace，不同 SqlSession 两次执行相同的 namespace 下的 SQL 语句，参数也相等，则第一次执行成功后会将数据保存在二级缓存中，第二次可直接从二级缓存中取出数据。

要使用二级缓存，需要在全局配置文件中配置 `<setting name="cacheEnabled" value="true"/>`，再在对应的映射文件中配置一个 `<cache/>` 标签。

Mybatis `#{}` 和 `${}` 的区别?

使用 `${}` 相当于使用字符串拼接，存在 SQL 注入的风险。

使用 `#{}` 相当于使用占位符，可以防止 SQL 注入，不支持使用占位符的地方就只能使用 `${}`，典型情况就是动态参数。

数据结构和算法

数据结构

AVL 树

AVL 树是平衡二叉查找树，增加和删除节点后通过树形旋转重新达到平衡。右旋是以某个节点为中心，将它沉入当前右子节点的位置，而让当前的左子节点作为新树的根节点，也称为顺时针旋转。同理左旋是以某个节点为中心，将它沉入当前左子节点的位置，而让当前的右子节点作为新树的根节点，也称为逆时针旋转。

红黑树

红黑树是 1972 年发明的，称为对称二叉 B 树，1978 年正式命名红黑树。主要特征是在每个节点上增加一个属性表示节点颜色，可以红色或黑色。红黑树和 AVL 树类似，都是在进行插入和删除时通过旋转保持自身平衡，从而获得较高的查找性能。与 AVL 树相比，红黑树不追求所有递归子树的高度差不超过 1，保证从根节点到叶尾的最长路径不超过最短路径的 2 倍，所以最差时间复杂度是 $O(\log_n)$。红黑树通过重新着色和左右旋转，更加高效地完成了插入和删除之后的自平衡调整。

红黑树在本质上还是二叉查找树，它额外引入了 5 个约束条件：① 节点只能是红色或黑色。② 根节点必须是黑色。③ 所有 NIL 节点都是黑色的。④ 一条路径上不能出现相邻的两个红色节点。⑤ 在任何递归子树中，根节点到叶子节点的所有路径上包含相同数目的黑色节点。这五个约束条件保证了红黑树的新增、删除、查找的最坏时间复杂度均为 $O(\log_n)$。如果一个树的左子节点或右子节点不存在，则均认定为黑色。红黑树的任何旋转在 3 次之内均可完成。

AVL 树和红黑树的区别

红黑树的平衡性不如 AVL 树，它维持的只是一种大致的平衡，不严格保证左右子树的高度差不超过 1。这导致节点数相同的情况下，红黑树的高度可能更高，也就是说平均查找次数会高于相同情况的 AVL 树。

在插入时，红黑树和 AVL 树都能在至多两次旋转内恢复平衡，在删除时由于红黑树只追求大致平衡，因此红黑树至多三次旋转可以恢复平衡，而 AVL 树最多需要 $O(\log_n)$ 次。AVL 树在插入和删除时，将向上回溯确定是否需要旋转，这个回溯的时间成本最差为 $O(\log_n)$，而红黑树每次向上回溯的步长为 2，回溯成本低。因此面对频繁地插入与删除红黑树更加合适。

B 树和B+ 树的区别

B 树中每个节点同时存储 key 和 data，而 B+ 树中只有叶子节点才存储 data，非叶子节点只存储 key。InnoDB 对 B+ 树进行了优化，在每个叶子节点上增加了一个指向相邻叶子节点的链表指针，形成了带有顺序指针的 B+ 树，提高区间访问的性能。

B+ 树的优点在于：① 由于 B+ 树在非叶子节点上不含数据信息，因此在内存页中能够存放更多的 key，数据存放得更加紧密，具有更好的空间利用率，访问叶子节点上关联的数据也具有更好的缓存命中率。② B+树的叶子结点都是相连的，因此对整棵树的遍历只需要一次线性遍历叶子节点即可。而 B 树则需要进行每一层的递归遍历，相邻的元素可能在内存中不相邻，所以缓存命中性没有 B+树好。但是 B 树也有优点，由于每个节点都包含 key 和 value，因此经常访问的元素可能离根节点更近，访问也更迅速。

排序

排序分类

排序可以分为内部排序和外部排序，在内存中进行的称为内部排序，当数据量很大时无法全部拷贝到内存需要使用外存，称为外部排序。

内部排序包括比较排序和非比较排序，比较排序包括插入/选择/交换/归并排序，非比较排序包括计数/基数/桶排序。

插入排序包括直接插入/希尔排序，选择排序包括直接选择/堆排序，交换排序包括冒泡/快速排序。

直接插入排序

稳定，平均/最差时间复杂度 $O(n^2)$，元素基本有序时最好时间复杂度 O(n)，空间复杂度 O(1)。

每一趟将一个待排序记录按其关键字的大小插入到已排好序的一组记录的适当位置上，直到所有待排序记录全部插入为止。

```
public void insertionSort(int[] nums) {
    for (int i = 1; i < nums.length; i++) {
        int insertNum = nums[i];
        int insertIndex;
        for (insertIndex = i - 1; insertIndex >= 0 && nums[insertIndex] >
insertNum; insertIndex--) {
            nums[insertIndex + 1] = nums[insertIndex];
        }
        nums[insertIndex + 1] = insertNum;
    }
}
```

直接插入没有利用到要插入的序列已有序的特点，插入第 i 个元素时可以通过二分查找找到插入位置 insertIndex，再把 i~insertIndex 之间的所有元素后移一位，把第 i 个元素放在插入位置上。

```
public void binaryInsertionSort(int[] nums) {
    for (int i = 1; i < nums.length; i++) {
        int insertNum = nums[i];
        int insertIndex = -1;
        int start = 0;
        int end = i - 1;
        while (start <= end) {
            int mid = start + (end - start) / 2;
            if (insertNum > nums[mid])
```

```
                start = mid + 1;
            else if (insertNum < nums[mid])
                end = mid - 1;
            else {
                insertIndex = mid + 1;
                break;
            }
        }
        if (insertIndex == -1)
            insertIndex = start;
        if (i - insertIndex >= 0)
            System.arraycopy(nums, insertIndex, nums, insertIndex + 1, i -
insertIndex);
        nums[insertIndex] = insertNum;
    }
}
```

希尔排序

又称缩小增量排序，是对直接插入排序的改进，不稳定，平均时间复杂度 $O(n^{1.3})$，最差时间复杂度 $O(n^2)$，最好时间复杂度 O(n)，空间复杂度 O(1)。

把记录按下标的一定增量分组，对每组进行直接插入排序，每次排序后减小增量，当增量减至 1 时排序完毕。

```
public void shellSort(int[] nums) {
    for (int d = nums.length / 2; d > 0 ; d /= 2) {
        for (int i = d; i < nums.length; i++) {
            int insertNum = nums[i];
            int insertIndex;
            for (insertIndex = i - d; insertIndex >= 0 && nums[insertIndex] >
insertNum; insertIndex -= d) {
                nums[insertIndex + d] = nums[insertIndex];
            }
            nums[insertIndex + d] = insertNum;
        }
    }
}
```

直接选择排序

不稳定，时间复杂度 $O(n^2)$，空间复杂度 O(1)。

每次在未排序序列中找到最小元素，和未排序序列的第一个元素交换位置，再在剩余未排序序列中重复该操作直到所有元素排序完毕。

```
public void selectSort(int[] nums) {
    int minIndex;
    for (int index = 0; index < nums.length - 1; index++){
        minIndex = index;
        for (int i = index + 1;i < nums.length; i++){
            if(nums[i] < nums[minIndex])
                minIndex = i;
        }
        if (index != minIndex){
            swap(nums, index, minIndex);
        }
    }
}
```

堆排序

是对直接选择排序的改进，不稳定，时间复杂度 O(nlogn)，空间复杂度 O(1)。

将待排序记录看作完全二叉树，可以建立大根堆或小根堆，大根堆中每个节点的值都不小于它的子节点值，小根堆中每个节点的值都不大于它的子节点值。

以大根堆为例，在建堆时首先将最后一个节点作为当前节点，如果当前节点存在父节点且值大于父节点，就将当前节点和父节点交换。在移除时首先暂存根节点的值，然后用最后一个节点代替根节点并作为当前节点，如果当前节点存在子节点且值小于子节点，就将其与值较大的子节点进行交换，调整完堆后返回暂存的值。

```
public void add(int[] nums, int i, int num){
    nums[i] = num;
    int curIndex = i;
    while (curIndex > 0) {
        int parentIndex = (curIndex - 1) / 2;
        if (nums[parentIndex] < nums[curIndex])
            swap(nums, parentIndex, curIndex);
        else break;
        curIndex = parentIndex;
    }
}

public int remove(int[] nums, int size){
    int result = nums[0];
    nums[0] = nums[size - 1];
    int curIndex = 0;
    while (true) {
        int leftIndex = curIndex * 2 + 1;
        int rightIndex = curIndex * 2 + 2;
        if (leftIndex >= size) break;
        int maxIndex = leftIndex;
        if (rightIndex < size && nums[maxIndex] < nums[rightIndex])
            maxIndex = rightIndex;
        if (nums[curIndex] < nums[maxIndex])
            swap(nums, curIndex, maxIndex);
        else break;
        curIndex = maxIndex;
    }
    return result;
}
```

冒泡排序

稳定，平均/最坏时间复杂度 O(n^2)，元素基本有序时最好时间复杂度 O(n)，空间复杂度 O(1)。

比较相邻的元素，如果第一个比第二个大就进行交换，对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对，每一轮排序后末尾元素都是有序的，针对 n 个元素重复以上步骤 n -1 次排序完毕。

```
public void bubbleSort(int[] nums) {
    for (int i = 0; i < nums.length - 1; i++) {
        for (int index = 0; index < nums.length - 1 - i; index++) {
            if (nums[index] > nums[index + 1])
                swap(nums, index, index + 1)
        }
    }
}
```

当序列已经有序时仍会进行不必要的比较，可以设置一个标志记录是否有元素交换，如果没有直接结束比较。

```
public void betterBubbleSort(int[] nums) {
    boolean swap;
    for (int i = 0; i < nums.length - 1; i++) {
        swap = true;
        for (int index = 0; index < nums.length - 1 - i; index++) {
            if (nums[index] > nums[index + 1]) {
                swap(nums, index ,index + 1);
                swap = false;
            }
        }
        if (swap) break;
    }
}
```

快速排序

是对冒泡排序的一种改进，不稳定，平均/最好时间复杂度 O(nlogn)，元素基本有序时最坏时间复杂度 O(n^2)，空间复杂度 O(logn)。

首先选择一个基准元素，通过一趟排序将要排序的数据分割成独立的两部分，一部分全部小于等于基准元素，一部分全部大于等于基准元素，再按此方法递归对这两部分数据进行快速排序。

快速排序的一次划分从两头交替搜索，直到 low 和 high 指针重合，一趟时间复杂度 O(n)，整个算法的时间复杂度与划分趟数有关。

最好情况是每次划分选择的中间数恰好将当前序列等分，经过 log(n) 趟划分便可得到长度为 1 的子表，这样时间复杂度 O(nlogn)。

最坏情况是每次所选中间数是当前序列中的最大或最小元素，这使每次划分所得子表其中一个为空表，这样长度为 n 的数据表需要 n 趟划分，整个排序时间复杂度 O(n^2)。

```
public void quickSort(int[] nums, int start, int end) {
    if (start < end) {
```

```
        int pivotIndex = getPivotIndex(nums, start, end);
        quickSort(nums, start, pivotIndex - 1);
        quickSort(nums, pivotIndex + 1, end);
    }
}

public int getPivotIndex(int[] nums, int start, int end) {
    int pivot = nums[start];
    int low = start;
    int high = end;
    while (low < high) {
        while (low <= high && nums[low] <= pivot)
            low++;
        while (low <= high && nums[high] > pivot)
            high--;
        if (low < high)
            swap(nums, low, high);
    }
    swap(nums, start, high);
    return high;
}
```

优化：当规模足够小时，例如 `end - start < 10` 时，采用直接插入排序。

归并排序

归并排序基于归并操作，是一种稳定的排序算法，任何情况时间复杂度都为 O(nlogn)，空间复杂度为 O(n)。

基本原理：应用分治法将待排序序列分成两部分，然后对两部分分别递归排序，最后进行合并，使用一个辅助空间并设定两个指针分别指向两个有序序列的起始元素，将指针对应的较小元素添加到辅助空间，重复该步骤到某一序列到达末尾，然后将另一序列剩余元素合并到辅助空间末尾。

适用场景：数据量大且对稳定性有要求的情况。

```
int[] help;

public void mergeSort(int[] arr) {
    int[] help = new int[arr.length];
    sort(arr, 0, arr.length - 1);
}

public void sort(int[] arr, int start, int end) {
    if (start == end) return;
    int mid = start + (end - start) / 2;
    sort(arr, start, mid);
    sort(arr, mid + 1, end);
    merge(arr, start, mid, end);
}

public void merge(int[] arr, int start, int mid, int end) {
    if (end + 1 - start >= 0) System.arraycopy(arr, start, help, start, end + 1
- start);
    int p = start;
    int q = mid + 1;
    int index = start;
```

```
    while (p <= mid && q <= end) {
        if (help[p] < help[q])
            arr[index++] = help[p++];
        else
            arr[index++] = help[q++];
    }
    while (p <= mid) arr[index++] = help[p++];
    while (q <= end) arr[index++] = help[q++];
}
```

排序算法选择

数据量规模较小，考虑直接插入或直接选择。当元素分布有序时直接插入将大大减少比较和移动记录的次数，如果不要求稳定性，可以使用直接选择，效率略高于直接插入。

数据量规模中等，选择希尔排序。

数据量规模较大，考虑堆排序（元素分布接近正序或逆序）、快速排序（元素分布随机）和归并排序（稳定性）。

一般不使用冒泡。

设计模式

设计模式原则

开闭原则：OOP 中最基础的原则，指一个软件实体（类、模块、方法等）应该对扩展开放，对修改关闭。强调用抽象构建框架，用实现扩展细节，提高代码的可复用性和可维护性。

单一职责原则：一个类、接口或方法只负责一个职责，降低代码复杂度以及变更引起的风险。

依赖倒置原则：程序应该依赖于抽象类或接口，而不是具体的实现类。

接口隔离原则：将不同功能定义在不同接口中实现接口隔离，避免了类依赖它不需要的接口，减少了接口之间依赖的冗余性和复杂性。

里氏替换原则：开闭原则的补充，规定了任何父类可以出现的地方子类都一定可以出现，可以约束继承泛滥，加强程序健壮性。

迪米特原则：也叫最少知道原则，每个模块对其他模块都要尽可能少地了解和依赖，降低代码耦合度。

合成/聚合原则：尽量使用组合(has-a)/聚合(contains-a)而不是继承(is-a)达到软件复用的目的，避免滥用继承带来的方法污染和方法爆炸，方法污染指父类的行为通过继承传递给子类，但子类并不具备执行此行为的能力；方法爆炸指继承树不断扩大，底层类拥有的方法过于繁杂，导致很容易选择错误。

设计模式的分类

创建型： 在创建对象的同时隐藏创建逻辑，不使用 new 直接实例化对象，程序在判断需要创建哪些对象时更灵活。包括工厂/抽象工厂/单例/建造者/原型模式。

结构型： 通过类和接口间的继承和引用实现创建复杂结构的对象。包括适配器/桥接模式/过滤器/组合/装饰器/外观/享元/代理模式。

行为型： 通过类之间不同通信方式实现不同行为。包括责任链/命名/解释器/迭代器/中介者/备忘录/观察者/状态/策略/模板/访问者模式。

简单工厂模式

简单工厂模式指由一个工厂对象来创建实例，客户端不需要关注创建逻辑，只需提供传入工厂的参数。

适用于工厂类负责创建对象较少的情况，缺点是如果要增加新产品，就需要修改工厂类的判断逻辑，违背开闭原则，且产品多的话会使工厂类比较复杂。

Calendar 抽象类的 `getInstance` 方法，调用 `createCalendar` 方法根据不同的地区参数创建不同的日历对象。

Spring 中的 BeanFactory 使用简单工厂模式，根据传入一个唯一的标识来获得 Bean 对象。

工厂方法模式

工厂方法模式指定义一个创建对象的接口，让接口的实现类决定创建哪种对象，让类的实例化推迟到子类中进行。

客户端只需关心对应工厂而无需关心创建细节，主要解决了产品扩展的问题，在简单工厂模式中如果产品种类变多，工厂的职责会越来越多，不便于维护。

Collection 接口这个抽象工厂中定义了一个抽象的 `iterator` 工厂方法，返回一个 Iterator 类的抽象产品。该方法通过 ArrayList 、HashMap 等具体工厂实现，返回 Itr、KeyIterator 等具体产品。

Spring 的 FactoryBean 接口的 `getObject` 方法也是工厂方法。

抽象工厂模式

抽象工厂模式指提供一个创建一系列相关或相互依赖对象的接口，无需指定它们的具体类。

客户端不依赖于产品类实例如何被创建和实现的细节，主要用于系统的产品有多于一个的产品族，而系统只消费其中某一个产品族产品的情况。抽象工厂模式的缺点是不方便扩展产品族，并且增加了系统的抽象性和理解难度。

java.sql.Connection 接口就是一个抽象工厂，其中包括很多抽象产品如 Statement、Blob、Savepoint 等。

单例模式的特点

单例模式属于创建型模式，一个单例类在任何情况下都只存在一个实例，构造方法必须是私有的、由自己创建一个静态变量存储实例，对外提供一个静态公有方法获取实例。

优点是内存中只有一个实例，减少了开销，尤其是频繁创建和销毁实例的情况下并且可以避免对资源的多重占用。缺点是没有抽象层，难以扩展，与单一职责原则冲突。

Spring 的 ApplicationContext 创建的 Bean 实例都是单例对象，还有 ServletContext、数据库连接池等也都是单例模式。

单例模式

饿汉式：在类加载时就初始化创建单例对象，线程安全，但不管是否使用都创建对象可能会浪费内存。

```
public class HungrySingleton {
    private HungrySingleton(){}

    private static HungrySingleton instance = new HungrySingleton();

    public static HungrySingleton getInstance() {
        return instance;
    }
}
```

懒汉式：在外部调用时才会加载，线程不安全，可以加锁保证线程安全但效率低。

```
public class LazySingleton {
    private LazySingleton(){}

    private static LazySingleton instance;

    public static LazySingleton getInstance() {
        if(instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}
```

双重检查锁：使用 volatile 以及多重检查来减小锁范围，提升效率。

```
public class DoubleCheckSingleton {
    private DoubleCheckSingleton(){}

    private volatile static DoubleCheckSingleton instance;

    public static DoubleCheckSingleton getInstance() {
        if(instance == null) {
            synchronized (DoubleCheckSingleton.class) {
                if (instance == null) {
                    instance = new DoubleCheckSingleton();
                }
            }
        }
        return instance;
    }
}
```

静态内部类：同时解决饿汉式的内存浪费问题和懒汉式的线程安全问题。

```
public class StaticSingleton {
    private StaticSingleton(){}

    public static StaticSingleton getInstance() {
        return StaticClass.instance;
    }

    private static class StaticClass {
        private static final StaticSingleton instance = new StaticSingleton();
    }
}
```

枚举：《Effective Java》提倡的方式，不仅能避免线程安全问题，还能防止反序列化重新创建新的对象，绝对防止多次实例化，也能防止反射破解单例的问题。

```
public enum EnumSingleton {
    INSTANCE;
}
```

代理模式

代理模式属于结构型模式，为其他对象提供一种代理以控制对这个对象的访问。优点是可以增强目标对象的功能，降低代码耦合度，扩展性好。缺点是在客户端和目标对象之间增加代理对象会导致请求处理速度变慢，增加系统复杂度。

Spring 利用动态代理实现 AOP，如果 Bean 实现了接口就使用 JDK 代理，否则使用 CGLib 代理。

静态代理：代理对象持有被代理对象的引用，调用代理对象方法时也会调用被代理对象的方法，但是会在被代理对象方法的前后增加其他逻辑。需要手动完成，在程序运行前就已经存在代理类的字节码文件，代理类和被代理类的关系在运行前就已经确定了。 缺点是一个代理类只能为一个目标服务，如果要服务多种类型会增加工作量。

动态代理：动态代理在程序运行时通过反射创建具体的代理类，代理类和被代理类的关系在运行前是不确定的。动态代理的适用性更强，主要分为 JDK 动态代理和 CGLib 动态代理。

- **JDK 动态代理：**通过 `Proxy` 类的 `newInstance` 方法获取一个动态代理对象，需要传入三个参数，被代理对象的类加载器、被代理对象实现的接口，以及一个 `InvocationHandler` 调用处理器来指明具体的逻辑，相比静态代理的优势是接口中声明的所有方法都被转移到 `InvocationHandler` 的 `invoke` 方法集中处理。
- **CGLib 动态代理：**JDK 动态代理要求实现被代理对象的接口，而 CGLib 要求继承被代理对象，如果一个类是 final 类则不能使用 CGLib 代理。两种代理都在运行期生成字节码，JDK 动态代理直接写字节码，而 CGLib 动态代理使用 ASM 框架写字节码，ASM 的目的是生成、转换和分析以字节数组表示的已编译 Java 类。 JDK 动态代理调用代理方法通过反射机制实现，而 GCLib 动态代理通过 FastClass 机制直接调用方法，它为代理类和被代理类各生成一个类，该类为代理类和被代理类的方法分配一个 int 参数，调用方法时可以直接定位，因此调用效率更高。

装饰器模式

装饰器模式属于结构型模式，在不改变原有对象的基础上将功能附加到对象，相比继承可以更加灵活地扩展原有对象的功能。

装饰器模式适合的场景：在不想增加很多子类的前提下扩展一个类的功能。

java.io 包中，InputStream 字节输入流通过装饰器 BufferedInputStream 增强为缓冲字节输入流。

装饰器模式的关注点在于给对象动态添加方法，而动态代理更注重对象的访问控制。动态代理通常会在代理类中创建被代理对象的实例，而装饰器模式会将装饰者作为构造方法的参数。

适配器模式

适配器模式属于结构型模式，它作为两个不兼容接口之间的桥梁，结合了两个独立接口的功能，将一个类的接口转换成另外一个接口使得原本由于接口不兼容而不能一起工作的类可以一起工作。

缺点是过多使用适配器会让系统非常混乱，不易整体把握。

java.io 包中，InputStream 字节输入流通过适配器 InputStreamReader 转换为 Reader 字符输入流。

Spring MVC 中的 HandlerAdapter，由于 handler 有很多种形式，包括 Controller、HttpRequestHandler、Servlet 等，但调用方式又是确定的，因此需要适配器来进行处理，根据适配规则调用 handle 方法。

Arrays.asList 方法，将数组转换为对应的集合（注意不能使用修改集合的方法，因为返回的 ArrayList 是 Arrays 的一个内部类）。

适配器模式没有层级关系，适配器和被适配者没有必然连续，满足 has-a 的关系，解决不兼容的问题，是一种后置考虑。

装饰器模式具有层级关系，装饰器与被装饰者实现同一个接口，满足 is-a 的关系，注重覆盖和扩展，是一种前置考虑。

适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。

策略模式

策略模式属于行为型模式，定义了一系列算法并封装起来，之间可以互相替换。策略模式主要解决在有多种算法相似的情况下，使用 if/else 所带来的难以维护。

优点是算法可以自由切换，可以避免使用多重条件判断并且扩展性良好，缺点是策略类会增多并且所有策略类都需要对外暴露。

在集合框架中，经常需要通过构造方法传入一个比较器 Comparator 进行比较排序。Comparator 就是一个抽象策略，一个类通过实现该接口并重写 compare 方法成为具体策略类。

创建线程池时，需要传入拒绝策略，当创建新线程使当前运行的线程数超过 maximumPoolSize 时会使用相应的拒绝策略处理。

模板模式

模板模式属于行为型模式，使子类可以在不改变算法结构的情况下重新定义算法的某些步骤，适用于抽取子类重复代码到公共父类。

优点是可以封装固定不变的部分，扩展可变的部分。缺点是每一个不同实现都需要一个子类维护，会增加类的数量。

为防止恶意操作，一般模板方法都以 final 修饰。

HttpServlet 定义了一套处理 HTTP 请求的模板，service 方法为模板方法，定义了处理HTTP请求的基本流程，doXXX 等方法为基本方法，根据请求方法的类型做相应的处理，子类可重写这些方法。

观察者模式

观察者模式属于行为型模式，也叫发布订阅模式，定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。主要解决一个对象状态改变给其他对象通知的问题，缺点是如果被观察者对象有很多的直接和间接观察者的话通知很耗时， 如果存在循环依赖的话可能导致系统崩溃，另外观察者无法知道目标对象具体是怎么发生变化的。

ServletContextListener 能够监听 ServletContext 对象的生命周期，实际上就是监听 Web 应用。当 Servlet 容器启动 Web 应用时调用 `contextInitialized` 方法，终止时调用 `contextDestroyed` 方法。

MySQL

逻辑架构

MySQL 的逻辑架构

第一层是服务器层，主要提供连接处理、授权认证、安全等功能。

第二层实现了 MySQL 核心服务功能，包括查询解析、分析、优化、缓存以及日期和时间等所有内置函数，所有跨存储引擎的功能都在这一层实现，例如存储过程、触发器、视图等。

第三层是存储引擎层，存储引擎负责 MySQL 中数据的存储和提取。服务器通过 API 与存储引擎通信，这些接口屏蔽了不同存储引擎的差异，使得差异对上层查询过程透明。除了会解析外键定义的 InnoDB 外，存储引擎不会解析 SQL，不同存储引擎之间也不会相互通信，只是简单响应上层服务器请求。

MySQL 的读写锁

在处理并发读或写时，可以通过实现一个由两种类型组成的锁系统来解决问题。这两种类型的锁通常被称为共享锁和排它锁，也叫读锁和写锁。读锁是共享的，相互不阻塞，多个客户在同一时刻可以同时读取同一个资源而不相互干扰。写锁则是排他的，也就是说一个写锁会阻塞其他的写锁和读锁，确保在给定时间内只有一个用户能执行写入并防止其他用户读取正在写入的同一资源。

在实际的数据库系统中，每时每刻都在发生锁定，当某个用户在修改某一部分数据时，MySQL 会通过锁定防止其他用户读取同一数据。写锁比读锁有更高的优先级，一个写锁请求可能会被插入到读锁队列的前面，但是读锁不能插入到写锁前面。

MySQL 的锁策略

表锁是MySQL中最基本的锁策略，并且是开销最小的策略。表锁会锁定整张表，一个用户在对表进行写操作前需要先获得写锁，这会阻塞其他用户对该表的所有读写操作。只有没有写锁时，其他读取的用户才能获取读锁，读锁之间不相互阻塞。

行锁可以最大程度地支持并发，同时也带来了最大开销。InnoDB 和 XtraDB 以及一些其他存储引擎实现了行锁。行锁只在存储引擎层实现，而服务器层没有实现。

数据库死锁

死锁是指多个事务在同一资源上相互占用并请求锁定对方占用的资源而导致恶性循环的现象。当多个事务试图以不同顺序锁定资源时就可能会产生死锁，多个事务同时锁定同一个资源时也会产生死锁。

为了解决死锁问题，数据库系统实现了各种死锁检测和死锁超时机制。越复杂的系统，例如InnoDB 存储引擎，越能检测到死锁的循环依赖，并立即返回一个错误。这种解决方式很有效，否则死锁会导致出现非常慢的查询。还有一种解决方法，就是当查询的时间达到锁等待超时的设定后放弃锁请求，这种方式通常来说不太好。InnoDB 目前处理死锁的方法是将持有最少行级排它锁的事务进行回滚。

死锁发生之后，只有部分或者完全回滚其中一个事务，才能打破死锁。对于事务型系统这是无法避免的，所以应用程序在设计时必须考虑如何处理死锁。大多数情况下只需要重新执行因死锁回滚的事务即可。

事务

事务是一组原子性的 SQL 查询，或者说一个独立的工作单元。如果数据库引擎能够成功地对数据库应用该组查询的全部语句，那么就执行该组查询。如果其中有任何一条语句因为崩溃或其他原因无法执行，那么所有的语句都不会执行。也就是说事务内的语句要么全部执行成功，要么全部执行失败。

事务特性

原子性 atomicity

一个事务在逻辑上是必须不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说不可能只执行其中的一部分。

一致性 consistency

数据库总是从一个一致性的状态转换到另一个一致性的状态。

隔离性 isolation

针对并发事务而言，隔离性就是要隔离并发运行的多个事务之间的相互影响，一般来说一个事务所做的修改在最终提交以前，对其他事务是不可见的。

持久性 durability

一旦事务提交成功，其修改就会永久保存到数据库中，此时即使系统崩溃，修改的数据也不会丢失。

MySQL 的隔离级别

未提交读 READ UNCOMMITTED

在该级别事务中的修改即使没有被提交，对其他事务也是可见的。事务可以读取其他事务修改完但未提交的数据，这种问题称为脏读。这个级别还会导致不可重复读和幻读，性能没有比其他级别好很多，很少使用。

提交读 READ COMMITTED

多数数据库系统默认的隔离级别。提交读满足了隔离性的简单定义：一个事务开始时只能"看见"已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前的任何修改对其他事务都是不可见的。也叫不可重复读，因为两次执行同样的查询可能会得到不同结果。

可重复读 REPEATABLE READ（MySQL默认的隔离级别）

可重复读解决了不可重复读的问题，保证了在同一个事务中多次读取同样的记录结果一致。但还是无法解决幻读，所谓幻读指的是当某个事务在读取某个范围内的记录时，会产生幻行。InnoDB 存储引擎通过多版本并发控制MVCC 解决幻读的问题。

可串行化 SERIALIZABLE

最高的隔离级别，通过强制事务串行执行避免幻读。可串行化会在读取的每一行数据上都加锁，可能导致大量的超时和锁争用的问题。实际应用中很少用到这个隔离级别，只有非常需要确保数据一致性且可以接受没有并发的情况下才考虑该级别。

MVCC

MVCC 是多版本并发控制，在很多情况下避免加锁，大都实现了非阻塞的读操作，写操作也只锁定必要的行。

InnoDB 的 MVCC 通过在每行记录后面保存两个隐藏列来实现，这两个列分别保存了行的创建时间和过期时间。不过存储的不是实际的时间值而是系统版本号，每开始一个新的事务系统版本号都会自动递增，事务开始时刻的系统版本号会作为事务的版本号，用来和查询到的每行记录的版本号比较。

MVCC 只能在 `READ COMMITTED` 和 `REPEATABLE READ` 两个隔离级别下工作，因为 `READ UNCOMMITTED` 总是读取最新的数据行，而不是符合当前事务版本的数据行，而 `SERIALIZABLE` 则会对所有读取的行都加锁。

InnoDB

InnoDB 是 MySQL 的默认事务型引擎，用来处理大量短期事务。InnoDB 的性能和自动崩溃恢复特性使得它在非事务型存储需求中也很流行，除非有特别原因否则应该优先考虑 InnoDB。

InnoDB 的数据存储在表空间中，表空间由一系列数据文件组成。MySQL4.1 后 InnoDB 可以将每个表的数据和索引放在单独的文件中。

InnoDB 采用 MVCC 来支持高并发，并且实现了四个标准的隔离级别。其默认级别是 `REPEATABLE READ`，并通过间隙锁策略防止幻读，间隙锁使 InnoDB 不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定防止幻行的插入。

InnoDB 表是基于聚簇索引建立的，InnoDB 的索引结构和其他存储引擎有很大不同，聚簇索引对主键查询有很高的性能，不过它的二级索引中必须包含主键列，所以如果主键很大的话其他所有索引都会很大，因此如果表上索引较多的话主键应当尽可能小。

InnoDB 的存储格式是平台独立的，可以将数据和索引文件从一个平台复制到另一个平台。

InnoDB 内部做了很多优化，包括从磁盘读取数据时采用的可预测性预读，能够自动在内存中创建加速读操作的自适应哈希索引，以及能够加速插入操作的插入缓冲区等。

MyISAM

MySQL5.1及之前，MyISAM 是默认存储引擎，MyISAM 提供了大量的特性，包括全文索引、压缩、空间函数等，但不支持事务和行锁，最大的缺陷就是崩溃后无法安全恢复。对于只读的数据或者表比较小、可以忍受修复操作的情况仍然可以使用 MyISAM。

MyISAM 将表存储在数据文件和索引文件中，分别以 `.MYD` 和 `.MYI` 作为扩展名。MyISAM 表可以包含动态或者静态行，MySQL 会根据表的定义决定行格式。MyISAM 表可以存储的行记录数一般受限于可用磁盘空间或者操作系统中单个文件的最大尺寸。

MyISAM 对整张表进行加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但是在表有读取查询的同时，也支持并发往表中插入新的记录。

对于MyISAM 表，MySQL 可以手动或自动执行检查和修复操作，这里的修复和事务恢复以及崩溃恢复的概念不同。执行表的修复可能导致一些数据丢失，而且修复操作很慢。

对于 MyISAM 表，即使是 BLOB 和 TEXT 等长字段，也可以基于其前 500 个字符创建索引。MyISAM 也支持全文索引，这是一种基于分词创建的索引，可以支持复杂的查询。

MyISAM 设计简单，数据以紧密格式存储，所以在某些场景下性能很好。MyISAM 最典型的性能问题还是表锁问题，如果所有的查询长期处于 Locked 状态，那么原因毫无疑问就是表锁。

Memory

如果需要快速访问数据且这些数据不会被修改，重启以后丢失也没有关系，那么使用 Memory 表是非常有用的。Memory 表至少要比 MyISAM 表快一个数量级，因为所有数据都保存在内存，不需要磁盘 IO，Memory 表的结构在重启后会保留，但数据会丢失。

Memory 表适合的场景：查找或者映射表、缓存周期性聚合数据的结果、保存数据分析中产生的中间数据。

Memory 表支持哈希索引，因此查找速度极快。虽然速度很快但还是无法取代传统的基于磁盘的表，Memory 表使用表级锁，因此并发写入的性能较低。它不支持 BLOB 和 TEXT 类型的列，并且每行的长度是固定的，所以即使指定了 VARCHAR 列，实际存储时也会转换成CHAR，这可能导致部分内存的浪费。

如果 MySQL 在执行查询的过程中需要使用临时表来保持中间结果，内部使用的临时表就是 Memory 表。如果中间结果太大超出了Memory 表的限制，或者含有 BLOB 或 TEXT 字段，临时表会转换成 MyISAM 表。

查询执行流程

简单来说分为五步：① 客户端发送一条查询给服务器。② 服务器先检查查询缓存，如果命中了缓存则立刻返回存储在缓存中的结果，否则进入下一阶段。③ 服务器端进行 SQL 解析、预处理，再由优化器生成对应的执行计划。④ MySQL 根据优化器生成的执行计划，调用存储引擎的 API 来执行查询。⑤ 将结果返回给客户端。

数据类型

VARCHAR 和 CHAR 的区别

VARCHAR 用于存储可变字符串，是最常见的字符串数据类型。它比 CHAR 更节省空间，因为它仅使用必要的空间。VARCHAR 需要 1 或 2 个额外字节记录字符串长度，如果列的最大长度不大于 255 字节则只需要 1 字节。VARCHAR 不会删除末尾空格。

VARCHAR 适用场景：字符串列的最大长度比平均长度大很多、列的更新很少、使用了 UTF8 这种复杂字符集，每个字符都使用不同的字节数存储。

CHAR 是定长的，根据定义的字符串长度分配足够的空间。CHAR 会删除末尾空格。

CHAR 适合存储很短的字符串，或所有值都接近同一个长度，例如存储密码的 MD5 值。对于经常变更的数据，CHAR 也比 VARCHAR更好，因为定长的 CHAR 不容易产生碎片。对于非常短的列，CHAR 在存储空间上也更有效率，例如用 CHAR 来存储只有 Y 和 N 的值只需要一个字节，但是 VARCHAR 需要两个字节，因为还有一个记录长度的额外字节。

DATETIME 和 TIMESTAMP 的区别

DATETIME 能保存大范围的值，从 1001~9999 年，精度为秒。把日期和时间封装到了一个整数中，与时区无关，使用 8 字节存储空间。

TIMESTAMP 和 UNIX 时间戳相同，只使用 4 字节的存储空间，范围比 DATETIME 小得多，只能表示 1970 ~2038 年，并且依赖于时区。

数据类型优化策略

更小的通常更好

一般情况下尽量使用可以正确存储数据的最小数据类型，更小的数据类型通常也更快，因为它们占用更少的磁盘、内存和 CPU 缓存。

尽可能简单

简单数据类型的操作通常需要更少的 CPU 周期，例如整数比字符操作代价更低，因为字符集和校对规则使字符相比整形更复杂。应该使用 MySQL 的内建类型 date、time 和 datetime 而不是字符串来存储日期和时间，另一点是应该使用整形存储 IP 地址。

尽量避免 NULL

通常情况下最好指定列为 NOT NULL，除非需要存储 NULL值。因为如果查询中包含可为 NULL 的列对 MySQL 来说更难优化，可为 NULL 的列使索引、索引统计和值比较都更复杂，并且会使用更多存储空间。当可为 NULL 的列被索引时，每个索引记录需要一个额外字节，在MyISAM 中还可能导致固定大小的索引变成可变大小的索引。

如果计划在列上建索引，就应该尽量避免设计成可为 NULL 的列。

索引

索引作用

索引也叫键，是存储引擎用于快速找到记录的一种数据结构。索引对于良好的性能很关键，尤其是当表中数据量越来越大时，索引对性能的影响愈发重要。在数据量较小且负载较低时，不恰当的索引对性能的影响可能还不明显，但数据量逐渐增大时，性能会急剧下降。

索引大大减少了服务器需要扫描的数据量、可以帮助服务器避免排序和临时表、可以将随机 IO 变成顺序 IO。但索引并不总是最好的工具，对于非常小的表，大部分情况下会采用全表扫描。对于中到大型的表，索引就非常有效。但对于特大型的表，建立和使用索引的代价也随之增长，这种情况下应该使用分区技术。

在MySQL中，首先在索引中找到对应的值，然后根据匹配的索引记录找到对应的数据行。索引可以包括一个或多个列的值，如果索引包含多个列，那么列的顺序也十分重要，因为 MySQL 只能使用索引的最左前缀。

B-Tree 索引

大多数 MySQL 引擎都支持这种索引，但底层的存储引擎可能使用不同的存储结构，例如 NDB 使用 T-Tree，而 InnoDB 使用 B+ Tree。

B-Tree 通常意味着所有的值都是按顺序存储的，并且每个叶子页到根的距离相同。B-Tree 索引能够加快访问数据的速度，因为存储引擎不再需要进行全表扫描来获取需要的数据，取而代之的是从索引的根节点开始进行搜索。根节点的槽中存放了指向子节点的指针，存储引擎根据这些指针向下层查找。通过比较节点页的值和要查找的值可以找到合适的指针进入下层子节点，这些指针实际上定义了子节点页中值的上限和下限。最终存储引擎要么找到对应的值，要么该记录不存在。叶子节点的指针指向的是被索引的数据，而不是其他的节点页。

B-Tree索引的限制：

- 如果不是按照索引的最左列开始查找，则无法使用索引。
- 不能跳过索引中的列，例如索引为 (id,name,sex)，不能只使用 id 和 sex 而跳过 name。
- 如果查询中有某个列的范围查询，则其右边的所有列都无法使用索引。

Hash 索引

哈希索引基于哈希表实现，只有精确匹配索引所有列的查询才有效。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码，哈希码是一个较小的值，并且不同键值的行计算出的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。

只有 Memory 引擎显式支持哈希索引，这也是 Memory 引擎的默认索引类型。

因为索引自身只需存储对应的哈希值，所以索引的结构十分紧凑，这让哈希索引的速度非常快，但它也有一些限制：

- 哈希索引数据不是按照索引值顺序存储的，无法用于排序。
- 哈希索引不支持部分索引列匹配查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。例如在数据列(a,b)上建立哈希索引，如果查询的列只有a就无法使用该索引。
- 哈希索引只支持等值比较查询，不支持任何范围查询。

自适应哈希索引

自适应哈希索引是 InnoDB 引擎的一个特殊功能，当它注意到某些索引值被使用的非常频繁时，会在内存中基于 B-Tree 索引之上再创键一个哈希索引，这样就让 B-Tree 索引也具有哈希索引的一些优点，比如快速哈希查找。这是一个完全自动的内部行为，用户无法控制或配置，但如果有必要可以关闭该功能。

空间索引

MyISAM 表支持空间索引，可以用作地理数据存储。和 B-Tree 索引不同，这类索引无需前缀查询。空间索引会从所有维度来索引数据，查询时可以有效地使用任意维度来组合查询。必须使用 MySQL 的 GIS 即地理信息系统的相关函数来维护数据，但 MySQL 对 GIS 的支持并不完善，因此大部分人都不会使用这个特性。

全文索引

通过数值比较、范围过滤等就可以完成绝大多数需要的查询，但如果希望通过关键字匹配进行查询，就需要基于相似度的查询，而不是精确的数值比较，全文索引就是为这种场景设计的。

MyISAM 的全文索引是一种特殊的 B-Tree 索引，一共有两层。第一层是所有关键字，然后对于每一个关键字的第二层，包含的是一组相关的"文档指针"。全文索引不会索引文档对象中的所有词语，它会根据规则过滤掉一些词语，例如停用词列表中的词都不会被索引。

聚簇索引

聚簇索引不是一种索引类型，而是一种数据存储方式。InnoDB 的聚簇索引实际上在同一个结构中保存了 B-Tree 索引和数据行。当表有聚餐索引时，它的行数据实际上存放在索引的叶子页中，因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

优点：① 可以把相关数据保存在一起。② 数据访问更快，聚簇索引将索引和数据保存在同一个 B-Tree 中，因此获取数据比非聚簇索引要更快。③ 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

缺点：① 聚簇索引最大限度提高了 IO 密集型应用的性能，如果数据全部在内存中将会失去优势。② 更新聚簇索引列的代价很高，因为会强制每个被更新的行移动到新位置。③ 基于聚簇索引的表插入新行或主键被更新导致行移动时，可能导致页分裂，表会占用更多磁盘空间。④ 当行稀疏或由于页分裂导致数据存储不连续时，全表扫描可能很慢。

覆盖索引

覆盖索引指一个索引包含或覆盖了所有需要查询的字段的值，不再需要根据索引回表查询数据。覆盖索引必须要存储索引列的值，因此 MySQL 只能使用 B-Tree 索引做覆盖索引。

优点：① 索引条目通常远小于数据行大小，可以极大减少数据访问量。② 因为索引按照列值顺序存储，所以对于 IO 密集型防伪查询回避随机从磁盘读取每一行数据的 IO 少得多。③ 由于 InnoDB 使用聚簇索引，覆盖索引对 InnoDB 很有帮助。InnoDB 的二级索引在叶子节点保存了行的主键值，如果二级主键能覆盖查询那么可以避免对主键索引的二次查询。

索引使用原则

建立索引

对查询频次较高且数据量比较大的表建立索引。索引字段的选择，最佳候选列应当从 WHERE 子句的条件中提取，如果 WHERE 子句中的组合比较多，应当挑选最常用、过滤效果最好的列的组合。业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引。

使用前缀索引

索引列开始的部分字符，索引创建后也是使用硬盘来存储的，因此短索引可以提升索引访问的 IO 效率。对于 BLOB、TEXT 或很长的 VARCHAR 列必须使用前缀索引，MySQL 不允许索引这些列的完整长度。前缀索引是一种能使索引更小更快的有效方法，但缺点是 MySQL 无法使用前缀索引做 ORDER BY 和 GROUP BY，也无法使用前缀索引做覆盖扫描。

选择合适的索引顺序

当不需要考虑排序和分组时，将选择性最高的列放在前面。索引的选择性是指不重复的索引值和数据表的记录总数之比，索引的选择性越高则查询效率越高，唯一索引的选择性是 1，因此也可以使用唯一索引提升查询效率。

删除无用索引

MySQL 允许在相同列上创建多个索引，重复的索引需要单独维护，并且优化器在优化查询时也需要逐个考虑，这会影响性能。重复索引是指在相同的列上按照相同的顺序创建的相同类型的索引，应该避免创建重复索引。如果创建了索引 (A,B) 再创建索引 (A) 就是冗余索引，因为这只是前一个索引的前缀索引，对于 B-Tree 索引来说是冗余的。解决重复索引和冗余索引的方法就是删除这些索引。除了重复索引和冗余索引，可能还会有一些服务器永远不用的索引，也应该考虑删除。

索引失效的情况

如果索引列出现了隐式类型转换，则 MySQL 不会使用索引。常见的情况是在 SQL 的 WHERE 条件中字段类型为字符串，其值为数值，如果没有加引号那么 MySQL 不会使用索引。

如果 WHERE 条件中含有 OR，除非 OR 前使用了索引列而 OR 之后是非索引列，索引会失效。

MySQL 不能在索引中执行 LIKE 操作，这是底层存储引擎 API 的限制，最左匹配的 LIKE 比较会被转换为简单的比较操作，但如果是以通配符开头的 LIKE 查询，存储引擎就无法做比较。这种情况下 MySQL 只能提取数据行的值而不是索引值来做比较。

如果查询中的列不是独立的，则 MySQL 不会使用索引。独立的列是指索引列不能是表达式的一部分，也不能是函数的参数。

对于多个范围条件查询，MySQL 无法使用第一个范围列后面的其他索引列，对于多个等值查询则没有这种限制。

如果 MySQL 判断全表扫描比使用索引查询更快，则不会使用索引。

索引文件具有 B-Tree 的最左前缀匹配特性，如果左边的值未确定，那么无法使用此索引。

优化

定位低效 SQL

可以通过两种方式来定位执行效率较低的 SQL 语句。一种是通过慢查询日志定位，可以通过慢查询日志定位那些已经执行完毕的 SQL 语句。另一种是使用 SHOW PROCESSLIST 查询，慢查询日志在查询结束以后才记录，所以在应用反应执行效率出现问题的时候查询慢查询日志不能定位问题，此时可以使用 SHOW PROCESSLIST 命令查看当前 MySQL 正在进行的线程，包括线程的状态、是否锁表等，可以实时查看 SQL 的执行情况，同时对一些锁表操作进行优化。找到执行效率低的 SQL 语句后，就可以通过 SHOW PROFILE、EXPLAIN 或 trace 等丰富来继续优化语句。

SHOW PROFILE

通过 SHOW PROFILE 可以分析 SQL 语句性能消耗，例如查询到 SQL 会执行多少时间，并显示 CPU、内存使用量，执行过程中系统锁及表锁的花费时间等信息。例如 `SHOW PROFILE CPU/MEMORY/BLOCK IO FOR QUERY N` 分别查询 id 为 N 的 SQL 语句的 CPU、内存以及 IO 的消耗情况。

---

trace

从 MySQL5.6 开始，可以通过 trace 文件进一步获取优化器是是如何选择执行计划的，在使用时需要先打开设置，然后执行一次 SQL，最后查看 information_schema.optimizer_trace 表而都内容，该表为联合i表，只能在当前会话进行查询，每次查询后返回的都是最近一次执行的 SQL 语句。

EXPLAIN 的字段

执行计划是 SQL 调优的一个重要依据，可以通过 EXPLAIN 命令查看 SQL 语句的执行计划，如果作用在表上，那么该命令相当于 DESC。EXPLAIN 的指标及含义如下：

指标名	含义
id	表示 SELECT 子句或操作表的顺序，执行顺序从大到小执行，当 id 一样时，执行顺序从上往下。
select_type	表示查询中每个 SELECT 子句的类型，例如 SIMPLE 表示不包含子查询、表连接或其他复杂语法的简单查询，PRIMARY 表示复杂查询的最外层查询，SUBQUERY 表示在 SELECT 或 WHERE 列表中包含了子查询。
type	表示访问类型，性能由差到好为：ALL 全表扫描、index 索引全扫描、range 索引范围扫描、ref 返回匹配某个单独值得所有行，常见于使用非唯一索引或唯一索引的非唯一前缀进行的查找，也经常出现在 join 操作中、eq_ref 唯一性索引扫描，对于每个索引键只有一条记录与之匹配、const 当 MySQL 对查询某部分进行优化，并转为一个常量时，使用这些访问类型，例如将主键或唯一索引置于 WHERE 列表就能将该查询转为一个 const、system 表中只有一行数据或空表，只能用于 MyISAM 和 Memory 表、NULL 执行时不用访问表或索引就能得到结果。SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是consts 最好。
possible_keys	表示查询时可能用到的索引，但不一定使用。列出大量可能索引时意味着备选索引数量太多了。
key	显示 MySQL 在查询时实际使用的索引，如果没有使用则显示为 NULL。
key_len	表示使用到索引字段的长度，可通过该列计算查询中使用的索引的长度，对于确认索引有效性以及多列索引中用到的列数目很重要。
ref	表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值。
rows	表示 MySQL 根据表统计信息及索引选用情况，估算找到所需记录所需要读取的行数。
Extra	表示额外信息，例如 Using temporary 表示需要使用临时表存储结果集，常见于排序和分组查询。Using filesort 表示无法利用索引完成的文件排序，这是 ORDER BY 的结果，可以通过合适的索引改进性能。Using index 表示只需要使用索引就可以满足查询表得要求，说明表正在使用覆盖索引。

优化 SQL 的策略

优化 COUNT 查询

COUNT 是一个特殊的函数，它可以统计某个列值的数量，在统计列值时要求列值是非空的，不会统计 NULL 值。如果在 COUNT 中指定了列或列的表达式，则统计的就是这个表达式有值的结果数，而不是 NULL。

COUNT 的另一个作用是统计结果集的行数，当 MySQL 确定括号内的表达式不可能为 NULL 时，实际上就是在统计行数。当使用 COUNT(*) 时，* 不会扩展成所有列，它会忽略所有的列而直接统计所有的行数。

某些业务场景并不要求完全精确的 COUNT 值，此时可以使用近似值来代替，EXPLAIN 出来的优化器估算的行数就是一个不错的近似值，因为执行 EXPLAIN 并不需要真正地执行查询。

通常来说 COUNT 都需要扫描大量的行才能获取精确的结果，因此很难优化。在 MySQL 层还能做的就只有覆盖扫描了，如果还不够就需要修改应用的架构，可以增加汇总表或者外部缓存系统。

优化关联查询

确保 ON 或 USING 子句中的列上有索引，在创建索引时就要考虑到关联的顺序。

确保任何 GROUP BY 和 ORDER BY 的表达式只涉及到一个表中的列，这样 MySQL 才有可能使用索引来优化这个过程。

在 MySQL 5.5 及以下版本尽量避免子查询，可以用关联查询代替，因为执行器会先执行外部的 SQL 再执行内部的 SQL。

优化 GROUP BY

如果没有通过 ORDER BY 子句显式指定要排序的列，当查询使用 GROUP BY 时，结果集会自动按照分组的字段进行排序，如果不关心结果集的顺序，可以使用 ORDER BY NULL 禁止排序。

优化 LIMIT 分页

在偏移量非常大的时候，需要查询很多条数据再舍弃，这样的代价非常高。要优化这种查询，要么是在页面中限制分页的数量，要么是优化大偏移量的性能。最简单的办法是尽可能地使用覆盖索引扫描，而不是查询所有的列，然后根据需要做一次关联操作再返回所需的列。

还有一种方法是从上一次取数据的位置开始扫描，这样就可以避免使用 OFFSET。其他优化方法还包括使用预先计算的汇总表，或者关联到一个冗余表，冗余表只包含主键列和需要做排序的数据列。

优化 UNION 查询

MySQL 通过创建并填充临时表的方式来执行 UNION 查询，除非确实需要服务器消除重复的行，否则一定要使用 UNION ALL，如果没有 ALL 关键字，MySQL 会给临时表加上 DISTINCT 选项，这会导致对整个临时表的数据做唯一性检查，这样做的代价非常高。

使用用户自定义变量

在查询中混合使用过程化和关系化逻辑的时候，自定义变量可能会非常有用。用户自定义变量是一个用来存储内容的临时容器，在连接 MySQL 的整个过程中都存在，可以在任何可以使用表达式的地方使用自定义变量。例如可以使用变量来避免重复查询刚刚更新过的数据、统计更新和插入的数量等。

优化 INSERT

需要对一张表插入很多行数据时，应该尽量使用一次性插入多个值的 INSERT 语句，这种方式将缩减客户端与数据库之间的连接、关闭等消耗，效率比多条插入单个值的 INSERT 语句高。也可以关闭事务的自动提交，在插入完数据后提交。当插入的数据是按主键的顺序插入时，效率更高。

复制

MySQL 主从复制的作用

复制解决的基本问题是让一台服务器的数据与其他服务器保持同步，一台主库的数据可以同步到多台备库上，备库本身也可以被配置成另外一台服务器的主库。主库和备库之间可以有多种不同的组合方式。

MySQL 支持两种复制方式：基于行的复制和基于语句的复制，基于语句的复制也称为逻辑复制，从 MySQL 3.23 版本就已存在，基于行的复制方式在 5.1 版本才被加进来。这两种方式都是通过在主库上记录二进制日志、在备库重放日志的方式来实现异步的数据复制。因此同一时刻备库的数据可能与主库存在不一致，并且无法包装主备之间的延迟。

MySQL 复制大部分是向后兼容的，新版本的服务器可以作为老版本服务器的备库，但是老版本不能作为新版本服务器的备库，因为它可能无法解析新版本所用的新特性或语法，另外所使用的二进制文件格式也可能不同。

复制解决的问题：数据分布、负载均衡、备份、高可用性和故障切换、MySQL 升级测试。

MySQL 主从复制的步骤

① 在主库上把数据更改记录到二进制日志中。② 备库将主库的日志复制到自己的中继日志中。 ③ 备库读取中继日志中的事件，将其重放到备库数据之上。

第一步是在主库上记录二进制日志，每次准备提交事务完成数据更新前，主库将数据更新的事件记录到二进制日志中。MySQL 会按事务提交的顺序而非每条语句的执行顺序来记录二进制日志，在记录二进制日志后，主库会告诉存储引擎可以提交事务了。

下一步，备库将主库的二进制日志复制到其本地的中继日志中。备库首先会启动一个工作的 IO 线程，IO 线程跟主库建立一个普通的客户端连接，然后在主库上启动一个特殊的二进制转储线程，这个线程会读取主库上二进制日志中的事件。它不会对事件进行轮询。如果该线程追赶上了主库将进入睡眠状态，直到主库发送信号量通知其有新的事件产生时才会被唤醒，备库 IO 线程会将接收到的事件记录到中继日志中。

备库的 SQL 线程执行最后一步，该线程从中继日志中读取事件并在备库执行，从而实现备库数据的更新。当 SQL 线程追赶上 IO 线程时，中继日志通常已经在系统缓存中，所以中继日志的开销很低。SQL 线程执行的时间也可以通过配置选项来决定是否写入其自己的二进制日志中。

Redis

架构

Redis 特点

基于键值对的数据结构服务器

值不仅可以是字符串，还可以是其他数据结构，不仅能应用于多种场景，也可以提高效率。主要提供五种数据结构：字符串、哈希、列表、集合、有序集合，在字符串基础上演变出 Bitmaps 和 HyperLogLog，Redis 3.2 加入了有关 GEO 地理信息定位的功能。

丰富的功能

① 键过期，可实现缓存。② 发布订阅，可实现消息系统。③ 支持 Lua 脚本，可创造 Redis 命令。④ 简单的事务功能，一定程度上保证事务特性。⑤ 流水线功能，客户端能将一批命令一次性传到 Redis，减少网络开销。

简单稳定

① 源码很少，早期只有 2 万行，在 3.0 版本添加了集群特性，增加到了 5 万行，相对于 NoSQL 数据库代码量少很多。② 单线程模型，服务端处理模型更简单，客户端开发更简单。③ 不依赖底层操作系统的类库，自己实现了事件处理的相关功能。虽然简单，但也稳定。

客户端语言多

Java、PHP、Python、C、C++ 等。

持久化

数据放在内存中不安全，一旦发生断电或故障就可能丢失，Redis 提供了两种持久化方式 RDB 和 AOF 将内存的数据保存到硬盘。

高性能

Redis 使用单线程架构和 IO 多路复用模型实现高性能的内存数据库服务。

每次客户端调用都经历了发送命令、执行命令、返回结果三个过程，因为 Redis 是单线程处理命令的，所以一条命令从客户端到达服务器不会立即执行，所有命令都会进入一个队列中逐个执行。客户端的执行顺序不确定，但确定不会有两条命令同时执行，不存在并发问题。

通常来说单线程处理能力要比多线程差，Redis 快的原因：① 纯内存访问，Redis 将所有数据放在内存。② 非阻塞 IO，Redis 使用 epoll 作为 IO 多路复用技术的实现，此外 Redis 本身的事件处理模型将 epoll 中的连接、读写、关闭都转换为事件，不在网络 IO 上浪费过多时间。③ 单线程避免了线程切换和竞争消耗。单线程对于每个命令的执行时间有要求，如果执行时间过长会造成其他命令阻塞，因此 Redis 是面向快速执行场景的数据库。

Redis 的数据结构

可以使用 type 命令查看当前键的数据类型结构，它们分别是：string、hash、list、set、zset，但这些只是 Redis 对外的数据结构。实际上每种数据结构都有自己底层的内部编码实现，这样 Redis 会在合适的场景选择合适的内部编码，string 包括了 raw、int 和 embstr，hash 包括了 hashtable 和 ziplist，list 包括了 linkedlist 和 ziplist，set 包括了 hashtable 和 intset，zset 包括了 skiplist 和 ziplist。可以使用 `object encoding` 查看内部编码。

Redis 内部编码作用

① 可以改进内部编码，而对外的数据结构和命令没有影响。

② 多种内部编码实现可以在不同场景下发挥各自的优势，例如 ziplist 比较节省内存，但在列表元素较多的情况下性能有所下降，这时 Redis 会根据配置选项将列表类型的内部实现转换为 linkedlist。

---

string

字符串类型是 Redis 最基础的数据结构，键都是字符串类型，而且其他几种数据结构都是在字符串类型的基础上构建的。字符串类型的值可以实际可以是字符串（简单的字符串、复杂的字符串如 JSON、XML）、数字（整形、浮点数）、甚至二进制（图片、音频、视频），但是值最大不能超过 512 MB。

---

string 的命令

设置值

`set key value [ex seconds] [px millseconds] [nx|xx]`

- ex seconds：为键设置秒级过期时间，跟 setex 效果一样
- px millseconds：为键设置毫秒级过期时间
- nx：键必须不存在才可以设置成功，用于添加，跟 setnx 效果一样。由于 Redis 的单线程命令处理机制，如果多个客户端同时执行，则只有一个客户端能设置成功，可以用作分布式锁的一种实现。
- xx：键必须存在才可以设置成功，用于更新

获取值

`get key`，如果不存在返回 nil

批量设置值

`mset key value [key value...]`

批量获取值

`mget key [key...]`

批量操作命令可以有效提高开发效率，假如没有 mget，执行 n 次 get 命令需要 n 次网络时间 + n 次命令时间，使用 mget 只需要 1 次网络时间 + n 次命令时间。Redis 可以支持每秒数万的读写操作，但这指的是 Redis 服务端的处理能力，对于客户端来说一次命令处理命令时间还有网络时间。因为 Redis 的处理能力已足够高，对于开发者来说，网络可能会成为性能瓶颈。

计数

`incr key`

incr 命令用于对值做自增操作，返回结果分为三种：① 值不是整数返回错误。② 值是整数，返回自增后的结果。③ 值不存在，按照值为 0 自增，返回结果 1。除了 incr 命令，还有自减 decr、自增指定数字 incrby、自减指定数组 decrby、自增浮点数 incrbyfloat。

---

string 的内部编码

- int：8 个字节的长整形
- embstr：小于等于 39 个字节的字符串
- raw：大于 39 个字节的字符串

---

string 的应用场景

缓存功能

Redis 作为缓存层，MySQL 作为存储层，首先从 Redis 获取数据，如果失败就从 MySQL 获取并将结果写回 Redis 并添加过期时间。

计数

Redis 可以实现快速计数功能，例如视频每播放一次就用 incy 把播放数加 1。

共享 Session

一个分布式 Web 服务将用户的 Session 信息保存在各自服务器，但会造成一个问题，出于负载均衡的考虑，分布式服务会将用户的访问负载到不同服务器上，用户刷新一次可能会发现需要重新登陆。为解决该问题，可以使用 Redis 将用户的 Session 进行集中管理，在这种模式下只要保证 Redis 是高可用和扩展性的，每次用户更新或查询登录信息都直接从 Redis 集中获取。

限速

例如为了短信接口不被频繁访问会限制用户每分钟获取验证码的次数或者网站限制一个 IP 地址不能在一秒内访问超过 n 次。可以使用键过期策略和自增计数实现。

---

hash

哈希类型指键值本身又是一个键值对结构，哈希类型中的映射关系叫 field-value，这里的 value 是指 field 对于的值而不是键对于的值。

---

hash 的命令

设置值

`hset key field value`，如果设置成功会返回 1，反之会返回 0，此外还提供了 hsetnx 命令，作用和 setnx 类似，只是作用于由键变为 field。

获取值

`hget key field`，如果不存在会返回 nil。

删除 field

`hdel key field [field...]`，会删除一个或多个 field，返回结果为删除成功 field 的个数。

计算 field 个数

`hlen key`

批量设置或获取 field-value

```
hmget key field [field...]
hmset key field value [field value...]
```

判断 field 是否存在

`hexists key field`，存在返回 1，否则返回 0。

获取所有的 field

`hkeys key`，返回指定哈希键的所有 field。

获取所有 value

`hvals key`，获取指定键的所有 value。

获取所有的 field-value

`hgetall key`，获取指定键的所有 field-value。

---

hash 的内部编码

ziplist 压缩列表：当哈希类型元素个数和值小于配置值（默认 512 个和 64 字节）时会使用 ziplist 作为内部实现，使用更紧凑的结构实现多个元素的连续存储，在节省内存方面比 hashtable 更优秀。

hashtable 哈希表：当哈希类型无法满足 ziplist 的条件时会使用 hashtable 作为哈希的内部实现，因为此时 ziplist 的读写效率会下降，而 hashtable 的读写时间复杂度都为 O(1)。

---

hash 的应用场景

缓存用户信息，每个用户属性使用一对 field-value，但只用一个键保存。

优点：简单直观，如果合理使用可以减少内存空间使用。

缺点：要控制哈希在 ziplist 和 hashtable 两种内部编码的转换，hashtable 会消耗更多内存。

---

list

list 是用来存储多个有序的字符串，列表中的每个字符串称为元素，一个列表最多可以存储 2^{32}-1 个元素。可以对列表两端插入（push）和弹出（pop），还可以获取指定范围的元素列表、获取指定索引下标的元素等。列表是一种比较灵活的数据结构，它可以充当栈和队列的角色，在实际开发中有很多应用场景。

list 有两个特点：① 列表中的元素是有序的，可以通过索引下标获取某个元素或者某个范围内的元素列表。② 列表中的元素可以重复。

list 的命令

添加

从右边插入元素：`rpush key value [value...]`

从左到右获取列表的所有元素：`lrange 0 -1`

从左边插入元素：`lpush key value [value...]`

向某个元素前或者后插入元素：`linsert key before|after pivot value`，会在列表中找到等于 pivot 的元素，在其前或后插入一个新的元素 value。

查找

获取指定范围内的元素列表：`lrange key start end`，索引从左到右的范围是 0~N-1，从右到左是 -1~-N，lrange 中的 end 包含了自身。

获取列表指定索引下标的元素：`lindex key index`，获取最后一个元素可以使用 `lindex key -1`。

获取列表长度：`llen key`

删除

从列表左侧弹出元素：`lpop key`

从列表右侧弹出元素：`rpop key`

删除指定元素：`lrem key count value`，如果 count 大于 0，从左到右删除最多 count 个元素，如果 count 小于 0，从右到左删除最多个 count 绝对值个元素，如果 count 等于 0，删除所有。

按照索引范围修剪列表：`ltrim key start end`，只会保留 start ~ end 范围的元素。

修改

修改指定索引下标的元素：`lset key index newValue`。

阻塞操作

阻塞式弹出：`blpop/brpop key [key...] timeout`，timeout 表示阻塞时间。

当列表为空时，如果 timeout = 0，客户端会一直阻塞，如果在此期间添加了元素，客户端会立即返回。

如果是多个键，那么brpop会从左至右遍历键，一旦有一个键能弹出元素，客户端立即返回。

如果多个客户端对同一个键执行 brpop，那么最先执行该命令的客户端可以获取弹出的值。

---

list 的内部编码

ziplist 压缩列表：跟哈希的 zipilist 相同，元素个数和大小小于配置值（默认 512 个和 64 字节）时使用。

linkedlist 链表：当列表类型无法满足 ziplist 的条件时会使用linkedlist。

Redis 3.2 提供了 quicklist 内部编码，它是以一个 ziplist 为节点的 linkedlist，它结合了两者的优势，为列表类提供了一种更为优秀的内部编码实现。

---

list 的应用场景

消息队列

Redis 的 lpush + brpop 即可实现阻塞队列，生产者客户端使用 lpush 从列表左侧插入元素，多个消费者客户端使用 brpop 命令阻塞式地抢列表尾部的元素，多个客户端保证了消费的负载均衡和高可用性。

文章列表

每个用户有属于自己的文章列表，现在需要分页展示文章列表，就可以考虑使用列表。因为列表不但有序，同时支持按照索引范围获取元素。每篇文章使用哈希结构存储。

lpush + lpop = 栈、lpush + rpop = 队列、lpush + ltrim = 优先集合、lpush + brpop = 消息队列。

---

set

集合类型也是用来保存多个字符串元素，和列表不同的是集合不允许有重复元素，并且集合中的元素是无序的，不能通过索引下标获取元素。一个集合最多可以存储 2^{32}-1 个元素。Redis 除了支持集合内的增删改查，还支持多个集合取交集、并集、差集。

---

set 的命令

添加元素

`sadd key element [element...]`，返回结果为添加成功的元素个数。

删除元素

`srem key element [element...]`，返回结果为成功删除的元素个数。

计算元素个数

`scard key`，时间复杂度为 O(1)，会直接使用 Redis 内部的遍历。

判断元素是否在集合中

`sismember key element`，如果存在返回 1，否则返回 0。

随机从集合返回指定个数个元素

`srandmember key [count]`，如果不指定 count 默认为 1。

从集合随机弹出元素

`spop key`，可以从集合中随机弹出一个元素。

获取所有元素

`smembers key`

求多个集合的交集/并集/差集

`sinter key [key...]`

`sunion key [key...]`

`sdiff key [key...]`

保存交集、并集、差集的结果

`sinterstore/sunionstore/sdiffstore destination key [key...]`

集合间运算在元素较多情况下比较耗时，Redis 提供这三个指令将集合间交集、并集、差集的结果保存在 destination key 中。

---

set 的内部编码

intset 整数集合：当集合中的元素个数小于配置值（默认 512 个时），使用 intset。

hashtable 哈希表：当集合类型无法满足 intset 条件时使用 hashtable。当某个元素不为整数时，也会使用 hashtable。

---

set 的应用场景

set 比较典型的使用场景是标签，例如一个用户可能与娱乐、体育比较感兴趣，另一个用户可能对例时、新闻比较感兴趣，这些兴趣点就是标签。这些数据对于用户体验以及增强用户黏度比较重要。

sadd = 标签、spop/srandmember = 生成随机数，比如抽奖、sadd + sinter = 社交需求。

---

zset

有序集合保留了集合不能有重复成员的特性，不同的是可以排序。但是它和列表使用索引下标作为排序依据不同的是，他给每个元素设置一个分数（score）作为排序的依据。有序集合提供了获取指定分数和元素查询范围、计算成员排名等功能。

---

zset 的命令

添加成员

`zadd key score member [score member...]`，返回结果是成功添加成员的个数

Redis 3.2 为 zadd 命令添加了 nx、xx、ch、incr 四个选项：

- nx：member 必须不存在才可以设置成功，用于添加。
- xx：member 必须存在才能设置成功，用于更新。
- ch：返回此次操作后，有序集合元素和分数变化的个数。
- incr：对 score 做增加，相当于 zincrby。

zadd 的时间复杂度为 O($\log_n$)，sadd 的时间复杂度为 O(1)。

计算成员个数

`zcard key`，时间复杂度为 O(1)。

计算某个成员的分数

`zscore key member`，如果不存在则返回 nil。

计算成员排名

`zrank key member`，从低到高返回排名。

`zrevrank key member`，从高到低返回排名。

删除成员

`zrem key member [member...]`，返回结果是成功删除的个数。

增加成员的分数

`zincrby key increment member`

返回指定排名范围的成员

`zrange key start end [withscores]`，从低到高返回

`zrevrange key start end [withscores]`， 从高到底返回

返回指定分数范围的成员

`zrangebyscore key min max [withscores] [limit offset count]`，从低到高返回

`zrevrangebyscore key min max [withscores] [limit offset count]`， 从高到底返回

返回指定分数范围成员个数

`zcount key min max`

删除指定分数范围内的成员

`zremrangebyscore key min max`

交集和并集

`zinterstore/zunionstore destination numkeys key [key...] [weights weight [weight...]] [aggregate sum|min|max]`

- `destination`：交集结果保存到这个键
- `numkeys`：要做交集计算键的个数
- `key`：需要做交集计算的键
- `weight`：每个键的权重，默认 1
- `aggregate sum|min|max`：计算交集后，分值可以按和、最小值、最大值汇总，默认 sum。

---

zset 的内部编码

ziplist 压缩列表：当有序集合元素个数和值小于配置值（默认128 个和 64 字节）时会使用 ziplist 作为内部实现。

skiplist 跳跃表：当 ziplist 不满足条件时使用，因为此时 ziplist 的读写效率会下降。

---

zset 的应用场景

有序集合的典型使用场景就是排行榜系统，例如用户上传了一个视频并获得了赞，可以使用 zadd 和 zincrby。如果需要将用户从榜单删除，可以使用 zrem。如果要展示获取赞数最多的十个用户，可以使用 zrange。

键和数据库管理

键重命名

`rename key newkey`

如果 rename 前键已经存在，那么它的值也会被覆盖。为了防止强行覆盖，Redis 提供了 renamenx 命令，确保只有 newkey 不存在时才被覆盖。由于重命名键期间会执行 del 命令删除旧的键，如果键对应值比较大会存在阻塞的可能。

键过期

`expire key seconds`：键在 seconds 秒后过期。

如果过期时间为负值，键会被立即删除，和 del 命令一样。persist 命令可以将键的过期时间清除。

对于字符串类型键，执行 set 命令会去掉过期时间，set 命令对应的函数 setKey 最后执行了 removeExpire 函数去掉了过期时间。setex 命令作为 set + expire 的组合，不单是原子执行并且减少了一次网络通信的时间。

键迁移

- move

  move 命令用于在 Redis 内部进行数据迁移，`move key db` 把指定的键从源数据库移动到目标数据库中。

- dump + restore

  可以实现在不同的 Redis 实例之间进行数据迁移，分为两步：

  ① `dump key` ，在源 Redis 上，dump 命令会将键值序列化，格式采用 RDB 格式。

  ② `restore key ttl value`，在目标 Redis 上，restore 命令将序列化的值进行复原，ttl 代表过期时间， ttl = 0 则没有过期时间。

  整个迁移并非原子性的，而是通过客户端分步完成，并且需要两个客户端。

- migrate

  实际上 migrate 命令就是将 dump、restore、del 三个命令进行组合，从而简化操作流程。migrate 具有原子性，支持多个键的迁移，有效提高了迁移效率。实现过程和 dump + restore 类似，有三点不同：

  ① 整个过程是原子执行，不需要在多个 Redis 实例开启客户端。

  ② 数据传输直接在源 Redis 和目标 Redis 完成。

  ③ 目标 Redis 完成 restore 后会发送 OK 给源 Redis，源 Redis 接收后根据 migrate 对应选项来决定是否在源 Redis 上删除对应键。

切换数据库

`select dbIndex`，Redis 中默认配置有 16 个数据库，例如 select 0 将切换到第一个数据库，数据库之间的数据是隔离的。

清除数据库

用于清除数据库，flushdb 只清除当前数据库，flushall 会清除所有数据库。如果当前数据库键值数量比较多，flushdb/flushall 存在阻塞 Redis 的可能性。

持久化

RDB 持久化的原理

RDB 持久化是把当前进程数据生成快照保存到硬盘的过程，触发 RDB 持久化过程分为手动触发和自动触发。

手动触发分别对应 save 和 bgsave 命令：

- save：阻塞当前 Redis 服务器，直到 RDB 过程完成为止，对于内存比较大的实例会造成长时间阻塞，线上环境不建议使用。
- bgasve：Redis 进程执行 fork 操作创建子进程，RDB 持久化过程由子进程负责，完成后自动结束。阻塞只发生在 fork 阶段，一般时间很短。bgsave 是针对 save 阻塞问题做的优化，因此 Redis 内部所有涉及 RDB 的操作都采用 bgsave 的方式，而 save 方式已经废弃。

除了手动触发外，Redis 内部还存在自动触发 RDB 的持久化机制，例如：

- 使用 save 相关配置，如 save m n，表示 m 秒内数据集存在 n 次修改时，自动触发 bgsave。
- 如果从节点执行全量复制操作，主节点自动执行 bgsave 生成 RDB 文件并发送给从节点。
- 执行 debug reload 命令重新加载 Redis 时也会自动触发 save 操作。
- 默认情况下执行 shutdown 命令时，如果没有开启 AOF 持久化功能则自动执行 bgsave。

---

bgsave

① 执行 bgsave 命令，Redis 父进程判断当前是否存在正在执行的子进程，如 RDB/AOF 子进程，如果存在 bgsave 命令直接返回。

② 父进程执行 fork 操作创建子进程，fork 操作过程中父进程会阻塞。

③ 父进程 fork 完成后，bgsave 命令返回并不再阻塞父进程，可以继续响应其他命令。

④ 子进程创建 RDB 文件，根据父进程内存生成临时快照文件，完成后对原有文件进行原子替换。

⑤ 进程发送信号给父进程表示完成，父进程更新统计信息。

RDB 持久化的优点

RDB 是一个紧凑压缩的二进制文件，代表 Redis 在某个时间点上的数据快照。非常适合于备份，全量复制等场景。例如每 6 个消时执行 bgsave 备份，并把 RDB 文件拷贝到远程机器或者文件系统中，用于灾难恢复。

Redis 加载 RDB 恢复数据远远快于 AOF 的方式。

RDB 持久化的缺点

RDB 方式数据无法做到实时持久化/秒级持久化，因为 bgsave 每次运行都要执行 fork 操作创建子进程，属于重量级操作，频繁执行成本过高。针对 RDB 不适合实时持久化的问题，Redis 提供了 AOF 持久化方式。

RDB 文件使用特定二进制格式保存，Redis 版本演进过程中有多个格式的 RDB 版本，存在老版本 Redis 服务无法兼容新版 RDB 格式的问题。

AOF 持久化的原理

AOF 持久化以独立日志的方式记录每次写命令，重启时再重新执行 AOF 文件中的命令达到恢复数据的目的。AOF 的主要作用是解决了数据持久化的实时性，目前是 Redis 持久化的主流方式。

开启 AOF 功能需要设置：`appendonly yes`，默认不开启。保存路径同 RDB 方式一致，通过 dir 配置指定。

AOF 的工作流程操作：命令写入 append、文件同步 sync、文件重写 rewrite、重启加载 load：

- 所有的写入命令会追加到 aof_buf 缓冲区中。
- AOF 缓冲区根据对应的策略向硬盘做同步操作。

- 随着 AOF 文件越来越大，需要定期对 AOF 文件进行重写，达到压缩的目的。
- 当服务器重启时，可以加载 AOF 文件进行数据恢复。

AOF 命令写入

AOF 命令写入的内容直接是文本协议格式，采用文本协议格式的原因：

- 文本协议具有很好的兼容性。
- 开启 AOF 后所有写入命令都包含追加操作，直接采用协议格式避免了二次处理开销。
- 文本协议具有可读性，方便直接修改和处理。

AOF 把命令追加到缓冲区的原因：

Redis 使用单线程响应命令，如果每次写 AOF 文件命令都直接追加到硬盘，那么性能完全取决于当前硬盘负载。先写入缓冲区中还有另一个好处，Redis 可以提供多种缓冲区同步硬盘策略，在性能和安全性方面做出平衡。

AOF 文件同步

Redis 提供了多种 AOF 缓冲区文件同步策略，由参数 `appendfsync` 控制，不同值的含义如下：

- always：命令写入缓冲区后调用系统 fsync 操作同步到 AOF 文件，fsync 完成后线程返回。每次写入都要同步 AOF，性能较低，不建议配置。
- everysec：命令写入缓冲区后调用系统 write 操作，write 完成后线程返回。fsync 同步文件操作由专门线程每秒调用一次。是建议的策略，也是默认配置，兼顾性能和数据安全。
- no：命令写入缓冲区后调用系统 write 操作，不对 AOF 文件做 fsync 同步，同步硬盘操作由操作系统负责，周期通常最长 30 秒。由于操作系统每次同步 AOF 文件的周期不可控，而且会加大每次同步硬盘的数据量，虽然提升了性能，但安全性无法保证。

AOF 文件重写

文件重写是把 Redis 进程内的数据转化为写命令同步到新 AOF 文件的过程，可以降低文件占用空间，更小的文件可以更快地被加载。

重写后 AOF 文件变小的原因：

- 进程内已经超时的数据不再写入文件。
- 旧的 AOF 文件含有无效命令，重写使用进程内数据直接生成，这样新的 AOF 文件只保留最终数据写入命令。
- 多条写命令可以合并为一个，为了防止单条命令过大造成客户端缓冲区溢出，对于 list、set、hash、zset 等类型操作，以 64 个元素为界拆分为多条。

AOF 重写分为手动触发和自动触发，手动触发直接调用 bgrewriteaof 命令，自动触发根据 `auto-aof-rewrite-min-size` 和 `auto-aof-rewrite-percentage` 参数确定自动触发时机。

重写流程：

① 执行 AOF 重写请求，如果当前进程正在执行 AOF 重写，请求不执行并返回，如果当前进程正在执行 bgsave 操作，重写命令延迟到 bgsave 完成之后再执行。

② 父进程执行 fork 创建子进程，开销等同于 bgsave 过程。

③ 父进程 fork 操作完成后继续响应其他命令，所有修改命令依然写入 AOF 缓冲区并同步到硬盘，保证原有 AOF 机制正确性。

④ 子进程根据内存快照，按命令合并规则写入到新的 AOF 文件。每次批量写入数据量默认为 32 MB，防止单次刷盘数据过多造成阻塞。

⑤ 新 AOF 文件写入完成后，子进程发送信号给父进程，父进程更新统计信息。

⑥ 父进程把 AOF 重写缓冲区的数据写入到新的 AOF 文件并替换旧文件，完成重写。

重启加载

AOF 和 RDB 文件都可以用于服务器重启时的数据恢复。Redis 持久化文件的加载流程：

① AOF 持久化开启且存在 AOF 文件时，优先加载 AOF 文件。

② AOF 关闭时且存在 RDB 文件时，记载 RDB 文件。

③ 加载 AOF/RDB 文件成功后，Redis 启动成功。

④ AOF/RDB 文件存在错误导致加载失败时，Redis 启动失败并打印错误信息。
