目　录

介绍

data-structures-questions

算法和程序结构是我们学习编程的基础，但是很多的时候，我们很多都是只是在应用，而没有深入的去研究这些，所以自己也在不断的思考和探索，然后分析，学习，总结自己学习的过程，希望可以和大家一起学习和交流下算法！

转载地址

https://github.com/KeKe-Li/data-structures-questions

data-structures-questions

觉得此文章不错，支持我的话可以给我star，:star:！ 如果有问题可以加我的微信 `Sen0676` ,也可以加入我们的交流群一起交流技术！

License

This is free software distributed under the terms of the MIT license

网络协议

网络协议

计算机网络协议技就是网络规则，是各种硬件和软件共同遵循的守则。网络协议融合于其它所有的软件系统中，在网络中协议是无所不在的。网络协议遍及OSI通信模型的各个层次，从比较常见的 `TCP/IP` 、 `HTTP` 、 `FTP` 协议，到 `OSPF` 、 `IGP` 等特殊协议，有上千种之多。
局域网常用 `TCP/IP` 、 `NetBEUI` 、 `IPX/SPX` 这三种通信协议。 `TCP/IP` 协议是最重要、最基础、最麻烦的一个，上网时需要详细设置IP地址、网关、子网掩码、DNS服务器等参数，不过随着技术的进步，现在基本是自动获取了。

`TCP/IP` 协议族中互为关联的协议有上百个之多，且都有不同的功能，分布在不同的协议层.

常用协议如下：

1、UDP：用户数据包协议，位于传输层，和IP协议配合使用，因为不能提供数据包的重传，所以适合传输较短的文件；

2、NFS：网络文件服务器，可使多台计算机透明地访问彼此的目录；

3、FTP：远程文件传输协议，允许用户将远程主机上的文件拷贝到自己的计算机上；

4、SMTP：简单邮政传输协议，用于传输电子邮件；

5、Telnet：提供远程登录功能，一台计算机用户可以登录到远程的另一台计算机上，如同在远程主机上直接操作一样.

6、HTTP协议（HyperText Transfer Protocol，超文本传输协议）是因特网上应用最为广泛的一种网络传输协议，所有的WWW文件都必须遵守这个标准。 HTTP是一个基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等）。

IP 协议

路由器对分组进行转发后，就会把数据包传到网络上，数据包最终是要传递到客户端或者服务器上的，那么数据包怎么知道要发往哪里呢？起到关键作用的就是 IP 协议。

IP 主要分为三个部分，分别是 IP 寻址、路由和分包组包。

IP 地址

一个数据包要在网络上传输，那么肯定需要知道这个数据包到底发往哪里，也就是说需要一个目标地址信息，IP 地址就是连接网络中的所有主机进行通信的目标地址，因此，在网络上的每个主机都需要有自己的 IP 地址。

在 IP 数据报发送的链路中，有可能链路非常长，比如说由中国发往美国的一个数据报，由于网络抖动等一些意外因素可能会导致数据报丢失，这时我们在这条链路中会放入一些 中转站，一方面能够确保数据报是否丢失，另一方面能够控制数据报的转发，这个中转站就是我们前面聊过的路由器，这个转发过程就是 路由控制。

路由控制(Routing) 是指将分组数据发送到最终目标地址的功能，即使网络复杂多变，也能够通过路由控制到达目标地址。因此，一个数据报能否到达目标主机，关键就在于路由器的控制。

这里有一个名词，就是跳，因为在一条链路中可能会布满很多路由器，路由器和路由器之间的数据报传送就是跳，比如你和朋友通信，中间就可能会经过路由器 A-> 路由器 B -> 路由器 C 。

那么这个跳转的范围是多大呢？

通常这个一跳是指从源 MAC 地址到目标 MAC 地址之间传输帧的区间，那么这个MAC 地址又是什么呢？

`MAC` 地址指的就是计算机的物理地址(Physical Address)，它是用来确认网络设备位置的地址。在 OSI 网络模型中，网络层负责 IP 地址的定位，而数据链路层负责 MAC 地址的定位。MAC 地址用于在网络中唯一标示一个网卡，一台设备若有一或多个网卡，则每个网卡都需要并会有一个唯一的 MAC 地址，也就是说 MAC 地址和网卡是紧密联系在一起的。

路由器的每一跳都需要询问当前中转的路由器，下一跳应该跳到哪里，从而跳转到目标地址。而不是数据报刚开始发送后，网络中所有的通路都会显示出来，这种多次跳转也叫做 `多跳路由` 。

IP 地址定义

我们现在又两个版本的 IP 地址， `IPv4` 和 `IPv6` ，我们首先看一下现如今还在广泛使用的 IPv4 地址。

`IPv4` 由 32 位正整数来表示，在计算机内部会转化为二进制来处理，但是二进制不符合人类阅读的习惯，所以我们根据易读性的原则把 32 位的 IP 地址以 8 位为一组，分成四组，每组之间以 `.` 进行分割，再将每组转换为十进制数。

如下图所示:

32 位	2^8	2^8	2^8	2^8
二进制数	10011100	11000101	00000001	00000001
二进制数.	10011100.	11000101.	00000001.	00000001
十进制数	156.	197.	1.	1

面这个 32 位的 IP 地址就会被转换为十进制的 `156.197.1.1` 。

每个这样 8 位位一组的数字，自然是非负数，其取值范围是 [0,255]。

IP 地址的总个数有 `2^32` 次幂个，这个数值算下来是 `4294967296` ，大概能允许 43 亿台设备连接到网络。实际上真的如此吗？

实际上 IP 不会以主机的个数来配置的，而是根据设备上的 网卡(NIC) 进行配置，每一块网卡都会设置一个或者多个 IP 地址，而且通常一台路由器会有至少两块网卡，所以可以设置两个以上的 IP 地址，所以主机的数量远远达不到 43 亿。

IP 地址构造和分类

IP 地址由 网络标识 和 主机标识 两部分组成，网络标识代表着网络地址，主机标识代表着主机地址。网络标识在数据链路的每个段配置不同的值。
网络标识必须保证相互连接的每个段的地址都不重复。而相同段内相连的主机必须有相同的网络地址。IP 地址的 主机标识 则不允许在同一网段内重复出现。

例如：我们所在的小区的某一栋楼就相当于是网络标识，某一栋楼的第几户就相当于是我的 `主机标识` 。这样可以通过xx省xx市xx区xx路xx小区xx栋来定位我的 `网络标识` ，这一栋的第几户就相当于是我的 `主机标识` 。

IP 地址分为四类，分别是 A类、B类、C类、D类、E类，它会根据 IP 地址中的第 1 位到第 4 位的比特对网络标识和主机标识进行分类。

- A 类：(1.0.0.0 - 126.0.0.0)（默认子网掩码：255.0.0.0 或 0xFF000000）第一个字节为网络号，后三个字节为主机号。该类 IP 地址的最前面为 0 ，所以地址的网络号取值于 1~126 之间。一般用于大型网络。
- B 类：(128.0.0.0 - 191.255.0.0)（默认子网掩码：255.255.0.0 或 0xFFFF0000）前两个字节为网络号，后两个字节为主机号。该类 IP 地址的最前面为 10 ，所以地址的网络号取值于 128~191 之间。一般用于中等规模网络。
- C 类：(192.0.0.0 - 223.255.255.0)（子网掩码：255.255.255.0 或 0xFFFFFF00）前三个字节为网络号，最后一个字节为主机号。该类 IP 地址的最前面为 110 ，所以地址的网络号取值于 192~223 之间。一般用于小型网络。
- D 类：是多播地址。该类 IP 地址的最前面为 1110 ，所以地址的网络号取值于 224~239 之间。一般用于多路广播用户。

- E 类：是保留地址。该类 IP 地址的最前面为 1111 ，所以地址的网络号取值于 240~255 之间。

而根据不同的 IP 范围，就有不同的地总空间分类:

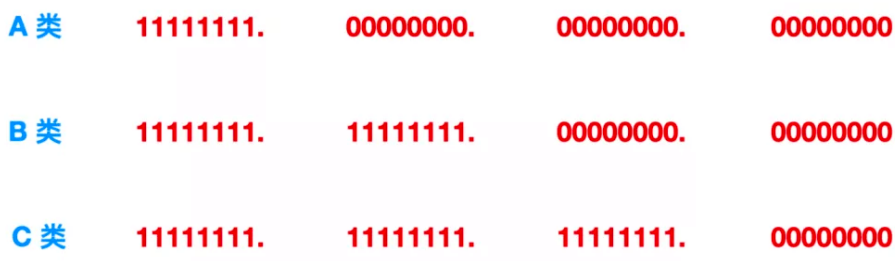
前8位地址范围	类	路由形式	占地址总空间的比例
0–127	A	单播	1/2
128–191	B	单播	1/4
192–223	C	单播	1/8
224–239	D	多播	1/16
240–255	E	–	1/16

子网掩码

子网掩码(subnet mask) 又叫做网络掩码，它是一种用来指明一个 IP 地址的哪些位标识的是主机所在的网络。子网掩码是一个 32位 地址，用于屏蔽 IP 地址的一部分以区别网络标识和主机标识。

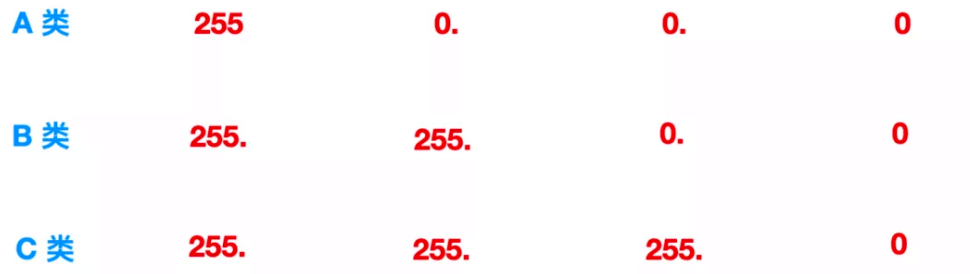
一个 IP 地址只要确定了其分类，也就确定了它的网络标识和主机标识。

由此，各个分类所表示的网络标识范围如下：



用 1 表示 IP 网络地址的比特范围，0 表示 IP 主机地址的范围。

将他们用十进制表示，那么这三类的表示如下:



保留地址

在IPv4 的几类地址中，有几个保留的地址空间不能在互联网上使用。这些地址用于特殊目的，不能在局域网外部路由。

地址块	地址范围	地址数	描述
0.0.0.0/8	0.0.0.0–0.255.255.255	16 777 216	当前网络
10.0.0.0/8	10.0.0.0–10.255.255.255	16 777 216	专用网络的本地通信
127.0.0.0/8	127.0.0.0–127.255.255.255	16 777 216	用于回环的本地地址
172.16.0.0/12	172.16.0.0–172.31.255.255	1 048 576	用于专用网络内的本地通信
192.168.0.0/16	192.168.0.0–192.168.255.255	65 536	用于专用网络内的本地通信
224.0.0.0/4	224.0.0.0–239.255.255.255	268 435 456	用于 IP 多播
192.0.0.0/24	192.0.0.0–192.0.0.255	256	IETF协议分配

IP 协议版本

目前，在全球 Internet 中共存有两个IP版本：IP 版本 4（IPv4）和 IP 版本6（IPv6）。IP 地址由二进制值组成，可驱动 Internet 上所有数据的路由。IPv4 地址的长度为 32 位，而 IPv6 地址的长度为 128 位。

Internet IP 资源由 Internet 分配号码机构（IANA）分配给区域 Internet 注册表（RIR），例如 `APNIC` ，该机构负责根 DNS ，IP 寻址和其他 Internet 协议资源。

然而IP 协议中非常重要的两个版本就是 IPv4 和 IPv6。

IPv4

IPv4 的全称是 `Internet Protocol version 4` ，是 Internet 协议的第四版。IPv4 是一种无连接的协议，这个协议会尽最大努力交付数据包，也就是说它不能保证任何数据包能到达目的地，也不能保证所有的数据包都会按照正确的顺序到达目标主机，这些都是由上层比如传输控制协议控制的。也就是说，单从 IP 看来，这是一个不可靠的协议。

IPv4 的数据报格式如下:

32 比特

0 - 3	4 - 7	8 - 13	14-15	16-18	19 - 31
版本	首部长度	服务类型	拥塞通告	数据报长度(字节)	
标识符				标志	13比特偏移
存活时间		协议		首部校验和	
源 IP 地址					
目标 IP 地址					
选项(如果有)					
数据					

IPv4的数据报包括≈:

- 版本字段(Version) 占用 4 bit，通信双方使用的版本必须一致，对于 IPv4 版本来说，字段值是 4。
- 首部长度(Internet Header Length) 占用 4 bit，首部长度说明首部有多少 32 位(4 字节)。由于 IPv4 首部可能包含不确定的选项，因此这个字段被用来确定数据的偏移量。
  大多数 IP 不包含这个选项，所以一般首部长度设置为 5， 数据报为 20 字节 。
- 服务类型(Differential Services Codepoint，DSCP) 占用 6 bit，以便使用不同的 IP 数据报，比如一些低时延、高吞吐量和可靠性的数据报。

服务类型如下表所示:

比　特	含　义
0 1 2	优先度▼
3	最低延迟
4	最大吞吐
5	最大可靠性
6	最小代价
(3~6)	最大安全
7	未定义

- 拥塞通告(Explicit Congestion Notification，ECN) 占用 2 bit，它允许在不丢弃报文的同时通知对方网络拥塞的发生。ECN 是一种可选的功能，仅当两端都支持并希望使用，且底层网络支持时才被使用。

最开始 `DSCP` 和 `ECN` 统称为 `TOS`，也就是区分服务，但是后来被细化为了 `DSCP` 和 `ECN` 。

- `数据报长度(Total Length)` 占用 16 bit，这 16 位是包括在数据在内的总长度，理论上数据报的总长度为 2 的 16 次幂 - 1，最大长度是 65535 字节，但是实际上数据报很少有超过 1500 字节的。IP 规定所有主机都必须支持最小 576 字节的报文，但大多数现代主机支持更大的报文。
  当下层的数据链路协议的最大传输单元（MTU）字段的值小于 IP 报文长度时，报文就必须被分片。

- `标识符(Identification)` 占用 16 bit，这个字段用来标识所有的分片，因为分片不一定会按序到达，所以到达目标主机的所有分片会进行重组，每产生一个数据报，计数器加1，并赋值给此字段。

- `标志(Flags)` 占用 3 bit，标志用于控制和识别分片，这 3 位分别是

0 位：保留，必须为0；

1 位：禁止分片（Don’t Fragment，DF），当 DF = 0 时才允许分片；

2 位：更多分片（More Fragment，MF），MF = 1 代表后面还有分片，MF = 0 代表已经是最后一个分片。

如果 DF 标志被设置为 1 ，但是路由要求必须进行分片，那么这条数据报回丢弃

- `分片偏移(Fragment Offset)` 占用 13 位，它指明了每个分片相对于原始报文开头的偏移量，以 8 字节作单位。

- `存活时间(Time To Live，TTL)` 占用 8 位，存活时间避免报文在互联网中迷失，比如陷入路由环路。存活时间以秒为单位，但小于一秒的时间均向上取整到一秒。在现实中，这实际上成了一个跳数计数器：报文经过的每个路由器都将此字段减 1，当此字段等于 0 时，报文不再向下一跳传送并被丢弃，这个字段最大值是 255。

- 协议(Protocol) 占用 8 位，这个字段定义了报文数据区使用的协议。协议内容。

- `首部校验和(Header Checksum)` 占用 16 位，首部校验和会对字段进行纠错检查，在每一跳中，路由器都要重新计算出的首部检验和并与此字段进行比对，如果不一致，此报文将会被丢弃。

- `源地址(Source address)` 占用 32 位，它是 IPv4 地址的构成条件，源地址指的是数据报的发送方.

- `目的地址(Destination address)` 占用 32 位，它是 IPv4 地址的构成条件，目标地址指的是数据报的接收方.

- `选项(Options)` 是附加字段，选项字段占用 1 - 40 个字节不等，一般会跟在目的地址之后。如果首部长度 > 5，就应该考虑选项字段。

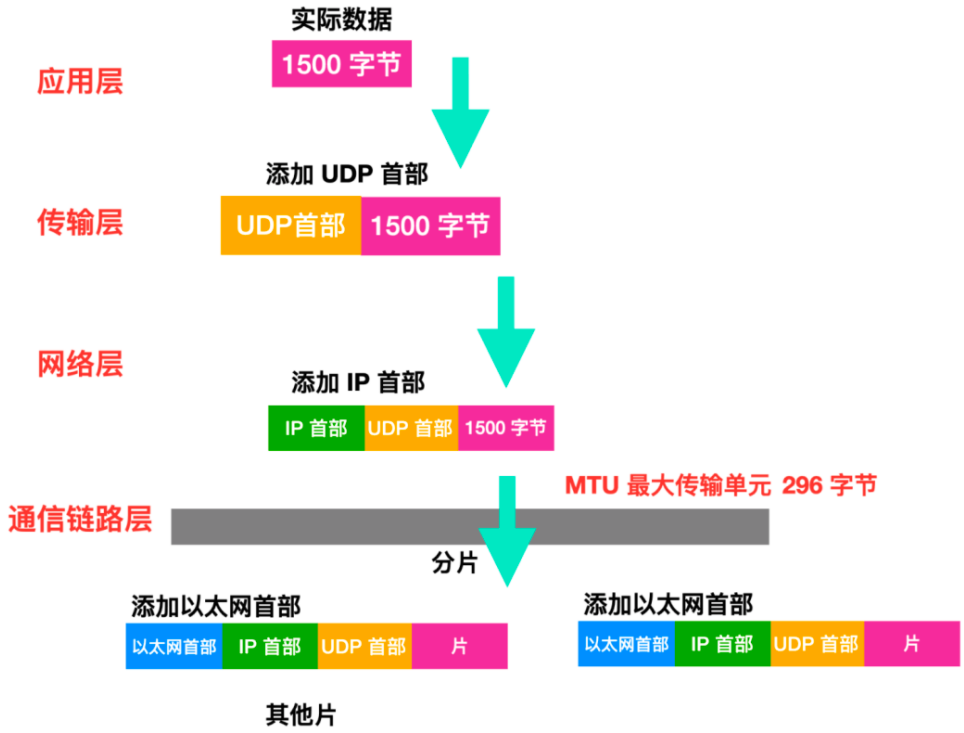
- `数据` 不是首部的一部分，因此并不被包含在首部检验和中。

在 IP 发送的过程中，每个数据报的大小是不同的，每个链路层协议能承载的网络层分组也不一样，有的协议能够承载大数据报，有的却只能承载很小的数据报，不同的链路层能够承载的数据报大小不同。

网络	MTU(字节)
超通道	65535
16Mb/s s令牌环(IBM)	17914
4Mb/s s令牌环	4464
以太网	1500
IEEE 802.3 /802.2	1492
点对点	296

IPv4 分片

一个链路层帧能承载的最大数据量叫做 `最大传输单元(Maximum Transmission Unit, MTU)` ，每个 IP 数据报封装在链路层帧中从一台路由器传到下一台路由器。
因为每个链路层所支持的最大 MTU 不一样，当数据报的大小超过 MTU 后，会在链路层进行分片，每个数据报会在链路层单独封装，每个较小的片都被称为 片(fragement)。



每个片在到达目的地后会进行重组，准确的来说是在运输层之前会进行重组，TCP 和 UDP 都会希望发送完整的、未分片的报文，出于性能的原因，分片重组不会在路由器中进行，而是会在目标主机中进行重组。

当目标主机收到从发送端发送过来的数据报后，它需要确定这些数据报中的分片是否是由源数据报分片传递过来的，如果是的话，还需要确定何时收到了分片中的最后一片，并且这些片会如何拼接一起成为数据报。

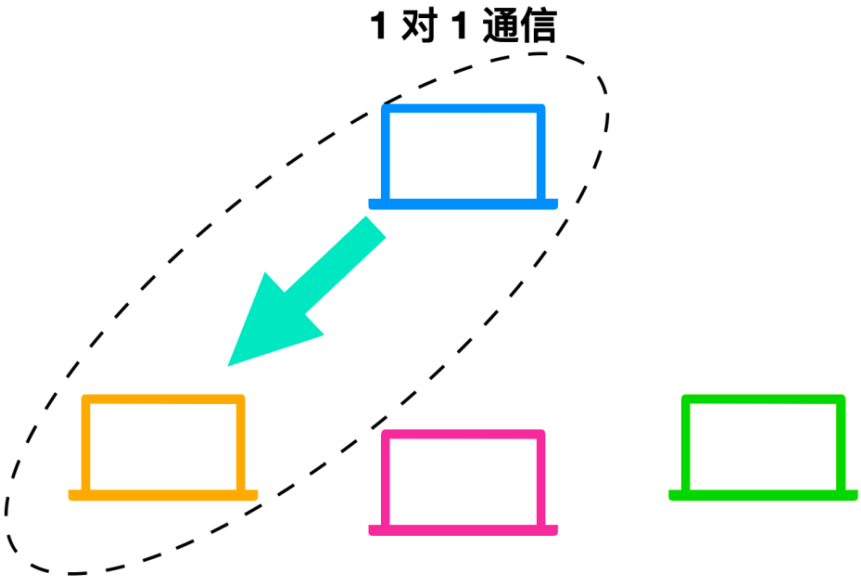
针对这些潜在的问题，IPv4 设计者将 标识、标志和片偏移放在 IP 数据报首部中。当生成一个数据报时，发送主机会为该数据报设置源和目的地址的同时贴上标识号。

发送主机通常将它发送的每个数据报的标识 + 1。当某路由器需要对一个数据报分片时，形成的每个数据报具有初始数据报的源地址、目标地址和标识号。当目的地从同一发送主机收到一系列数据报时，它能够检查数据报的标识号以确定哪些数据是由源数据报发送过来的。由于 IP 是一种不可靠的服务，分片可能会在网路中丢失，鉴于这种情况，通常会把分片的最后一个比特设置为 0 ，其他分片设置为 1，同时使用偏移字段指定分片应该在数据报的哪个位置。

IPv4 寻址

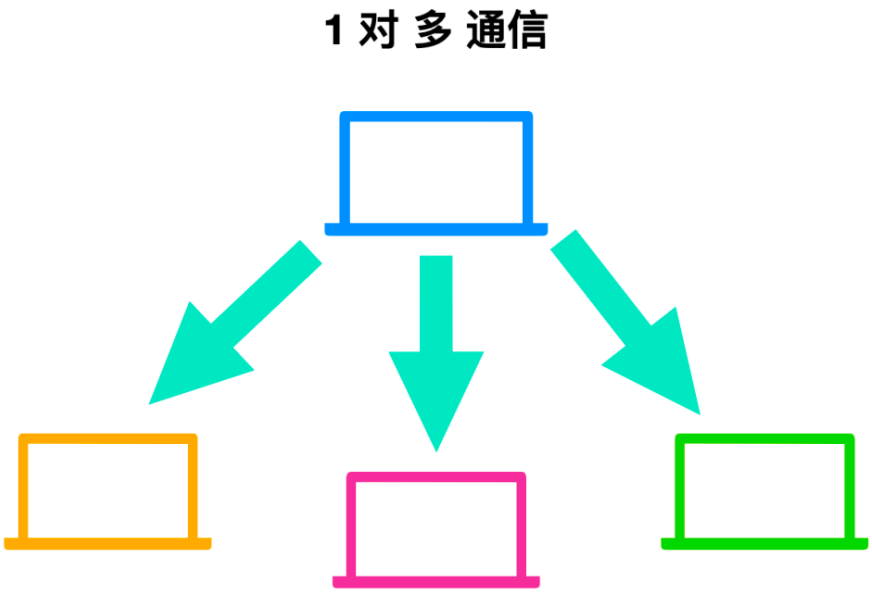
IPv4 支持三种不同类型的寻址模式，分别是:

- 单播寻址模式：在这种模式下，数据只发送到一个目的地的主机。

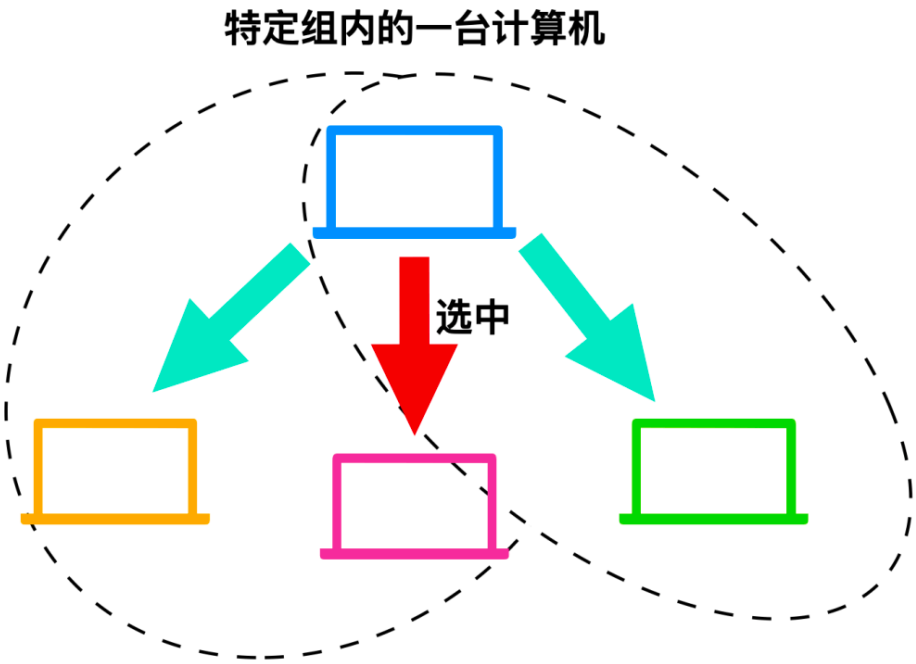


- 广播寻址模式：在此模式下，数据包将被寻址到网段中的所有主机。

  这里客户端发送一个数据包，由所有服务器接收：



- 组播寻址模式：此模式是前两种模式的混合，即发送的数据包既不指向单个主机也不指定段上的所有主机.



IPv6

随着端系统接入的越来越多，IPv4 已经无法满足分配了，所以，IPv6 应运而生，IPv6 就是为了解决 IPv4 的地址耗尽问题而被标准化的网际协议。

IPv4 的地址长度为 4 个 8 字节，即 32 比特， 而 IPv6 的地址长度是原来的四倍，也就是 128 比特，一般写成 8 个 16 位字节。

从 IPv4 切换到 IPv6 及其耗时，需要将网络中所有的主机和路由器的 IP 地址进行设置，在互联网不断普及的今天，替换所有的 IP 是一个工作量及其庞大的任务。我们后面会说。

IPv6 的地址是怎样的:

32 比特

0 - 3	4 - 11	12 - 31	
版本	流量类型	流标签	
0 - 15		16 - 23	24- 31
有效载荷长度		下一个首部	跳限制
源地址(128 比特)			
目的地址(128 比特)			
数据			

- 版本与 IPv4 一样，版本号由 4 bit 构成，IPv6 版本号的值为 6。
- 流量类型(Traffic Class) 占用 8 bit，它就相当于 IPv4 中的服务类型(Type Of Service)。

- 流标签(Flow Label) 占用 20 bit，这 20 比特用于标识一条数据报的流，能够对一条流中的某些数据报给出优先权，或者它能够用来对来自某些应用的数据报给出更高的优先权，只有流标签、源地址和目标地址一致时，才会被认为是一个流。
- 有效载荷长度(Payload Length) 占用 16 bit，这 16 比特值作为一个无符号整数，它给出了在 IPv6 数据报中跟在鼎昌 40 字节数据报首部后面的字节数量。
- 下一个首部(Next Header) 占用 8 bit，它用于标识数据报中的内容需要交付给哪个协议，是 TCP 协议还是 UDP 协议。
- 跳限制(Hop Limit) 占用 8 bit，这个字段与 IPv4 的 TTL 意思相同。数据每经过一次路由就会减 1，减到 0 则会丢弃数据。
- 源地址(Source Address) 占用 128 bit (8 个 16 位)，表示发送端的 IP 地址。
- 目标地址(Destination Address) 占用 128 bit (8 个 16 位)，表示接收端 IP 地址。

可以看到，相较于 IPv4 ，IPv6 取消了下面几个字段:

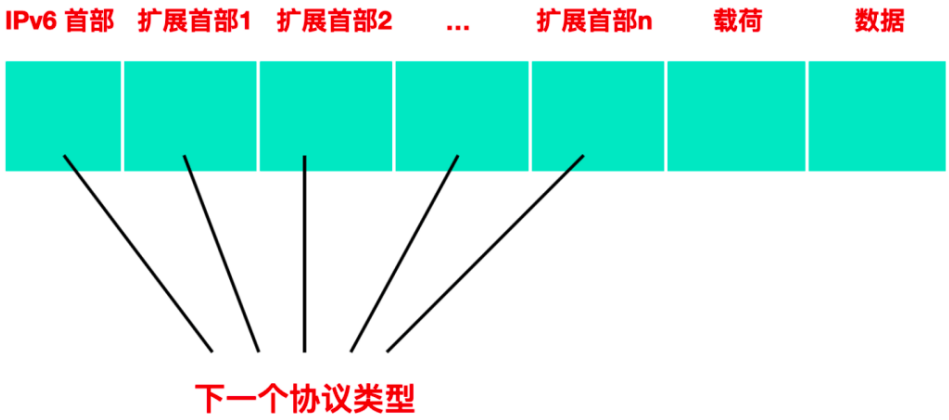
- 标识符、标志和比特偏移：IPv6 不允许在中间路由器上进行分片和重新组装。这种操作只能在端系统上进行，IPv6 将这个功能放在端系统中，加快了网络中的转发速度。
- 首部校验和：因为在运输层和数据链路执行了报文段完整性校验工作，IP 设计者大概觉得在网络层中有首部校验和比较多余，所以去掉了。IP 更多专注的是快速处理分组数据。
- 选项字段：选项字段不再是标准 IP 首部的一部分了，但是它并没有消失，而是可能出现在 IPv6 的扩展首部，也就是下一个首部中。

IPv6 扩展首部

IPv6 首部长度固定，无法将选项字段加入其中，取而代之的是 IPv6 使用了扩展首部

扩展首部通常介于 IPv6 首部与 `TCP/UDP` 首部之间，在 IPv4 中可选长度固定为 40 字节，在 IPv6 中没有这样的限制。IPv6 的扩展首部可以是任意长度。扩展首部中还可以包含扩展首部协议和下一个扩展字段。

IPv6 首部中没有标识和标志字段，对 IP 进行分片时，需要使用到扩展首部。



具体的扩展首部表如下所示:

扩展首部	协议号
IPv6 逐跳选项（HOPOPT）	0
IPv6 路由标头（IPv6-Route）	43
IPv6 片首部（IPv6-Frag）	44
载荷加密（ESP）	50
认证首部（AH）	51
首部终止（IPv6-NoNxt）	59
目标地址选项（IPv6-Opts）	60
移动首部（Mobility Header）	135

IPv6 特点

IPv6 的特点在 IPv4 中得以实现，但是即便实现了 IPv4 的操作系统，也未必实现了 IPv4 的所有功能。

IPv6 却将这些功能大众化了，也就表明这些功能在 IPv6 已经进行了实现，这些功能主要有:

- 地址空间变得更大：这是 IPv6 最主要的一个特点，即支持更大的地址空间。
- 精简报文结构: IPv6 要比 IPv4 精简很多，IPv4 的报文长度不固定，而且有一个不断变化的选项字段；IPv6 报文段固定，并且将选项字段，分片的字段移到了 IPv6 扩展头中，这就极大的精简了 IPv6 的报文结构。
- 实现了自动配置：IPv6 支持其主机设备的状态和无状态自动配置模式。这样，没有 DHCP 服务器不会停止跨段通信。
- 层次化的网络结构：IPv6 不再像 IPv4 一样按照 A、B、C等分类来划分地址，而是通过 IANA -> RIR -> ISP 这样的顺序来分配的。IANA 是国际互联网号码分配机构，RIR 是区域互联网注册管理机构，ISP 是一些运营商（例如电信、移动、联通）。
- IPSec：IPv6 的扩展报头中有一个认证报头、封装安全净载报头，这两个报头是 IPsec 定义的。通过这两个报头网络层自己就可以实现端到端的安全，而无需像 IPv4 协议一样需要其他协议的帮助。
- 支持任播：IPv6 引入了一种新的寻址方式，称为任播寻址。

IPv6 地址

我们知道，IPv6 地址长度为 128 位，他所能表示的范围是 `2 ^ 128` 次幂，这个数字非常庞大，几乎涵盖了你能想到的所有主机和路由器，那么 IPv6 该如何表示呢？

一般我们将 128 比特的 IP 地址以每 16 比特为一组，并用 `:` 号进行分隔，如果出现连续的 0 时还可以将 0 省略，并用 :: 两个冒号隔开，记住，一个 IP 地址只允许出现一次两个连续的冒号。

下面是一些 IPv6 地址的示例:

- 二进制数表示

1111000011010110 : 1111000011010110 : 1111000011010110 :

1111000011010110 : 1111000011010110 : 1111000011010110 :

1111000011010110 : 1111000011010110

- 用十六进制数表示

FEAD : A3D4 : B6EA : 321D :

A34D : 47AE : 732B : 54A6

- 出现两个冒号的情况

A120 : 0 : 0 : 0 : 8 : 4CD : 126B : 32D

A120 :: 8 : 4CD : 126B : 32D

A120 和 4CD 中间的 0 被 `::` 所替换了。

转自：https://github.com/KeKe-Li/data-structures-questions

数据结构

数据结构

数据结构是一门研究非数值计算的程序设计问题中的操作对象，以及它们之间的关系和操作等相关问题的学科。通常我们的程序设计=数据结构+算法,学好数据结构也是我们学习编程的重要一部分.

数据结构，本质上是数据之间的结构关系，或者理解成数据元素相互之间存在的一种或多种特定关系的集合。数据结构中的结构，也就是我们研究的主体对象。在数据结构中我们很少研究数据，因为数据在内存中的表现形式对于我们都是一样的，也就是二进制。传统上，我们把数据结构分为逻辑结构和物理结构。

数据结构，按照视点的不同,我们把数据结构分为逻辑结构和物理结构。

逻辑结构

逻辑结构指反映数据元素之间的逻辑关系的数据结构，其中的逻辑关系是指数据元素之间的前后关系，而与他们在计算机中的存储位置无关。

逻辑结构分为以下四类：

1. 集合结构

集合结构中的数据元素同属于一个集合，他们之间是并列的关系，除此之外没有其他关系。

2. 线性结构

线性结构中的元素存在一对一的相互关系。

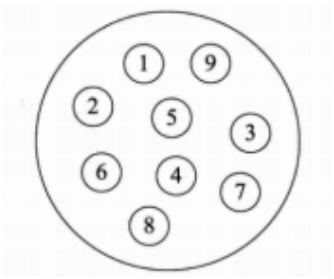
3. 树形结构

树形结构中的元素存在一对多的相互关系。

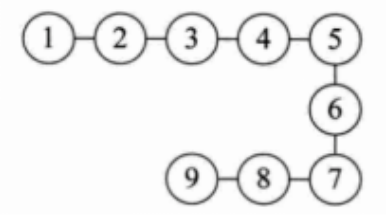
4. 图形结构

图形结构中的元素存在多对多的相互关系。

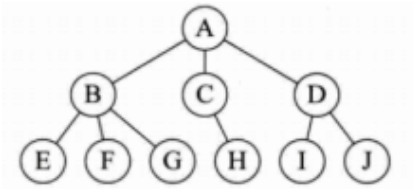
这四种类型之间的图形结构是:



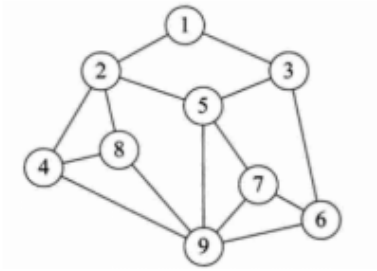
集合结构



线性结构



树形结构



图形结构

物理结构

物理结构又叫存储结构，指数据的逻辑结构在计算机存储空间的存放形式。通俗的讲，物理结构研究的是数据在存储器中存放的形式。
存储器主要针对于内存而言，像硬盘、软盘、光盘等外部存储器的数据组织通常用文件结构来描述。

数据在内存中的存储结构，也就是物理结构，分为两种：顺序存储结构和链式存储结构。

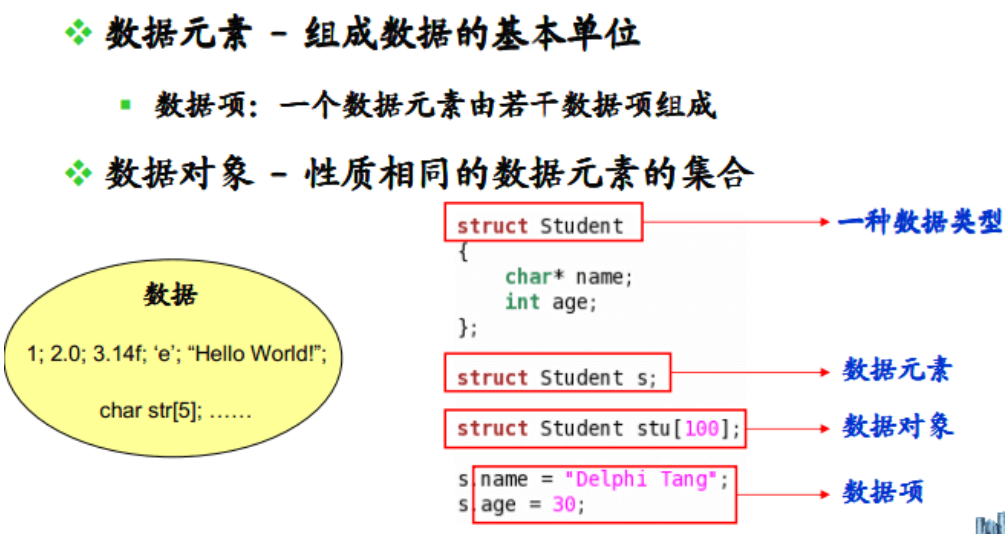
1. 顺序存储结构

顺序存储结构是把数据元素存放在地址连续的存储单元里，其数据间的逻辑关系和物理关系是一致的。数组就是顺序存储结构的典型代表。

2. 链式存储结构

链式存储结构是把数据元素存放在内存中的任意存储单元里，也就是可以把数据存放在内存的各个位置。这些数据在内存中的地址可以是连续的，也可以是不连续的。

和顺序存储结构不同的是，链式存储结构的数据元素之间是通过指针来连接的，我们可以通使用指针来找到某个数据元素的位置，然后对这个数据元素进行一些操作。



因此呢,数据结构是相互之间存在一种或多种特定 关系的数据元素的集合 。 同样是结构，从不同的角度来讨论，会有不同的分类.

逻辑结构分为: 集合结构, 线性结构, 树形结构, 图形结构.

物理结构分为: 顺序存储结构, 链式存储结构.

转自：https://github.com/KeKe-Li/data-structures-questions

算法

算法

什么是算法呢?算法是描述解决问题的方法。算法 (Algo付出m) 这个单词最早出 现在被斯数学家阿勒·花剌子密在公元 825 年(相当于我们中国的唐朝时期)所写的
《印度数字算术》中。

如今普遍认可的对算法的定义是 :算法是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。即算法是描述解决问题的方法.

算法介绍

算法是求解一个问题所需要的步骤所形成的解决方法，每一步包括一个或者多个操作。无论是现实生活中还是计算机中，解决同一个问题的方法可能有很多种，在这N多种算法中，肯定存在一个执行效率最快的方法，那么这个方法就是最优算法。

算法具有五个基本特征：输入、输出、有穷性、确定性和可行性。

1. 输入

一个算法具有零个或者多个输出。以刻画运算对象的初始情况，所谓0个输入是指算法本身定出了初始条件。

2. 输出

算法至少有一个输出。也就是说，算法一定要有输出。输出的形式可以是打印，也可以使返回一个值或者多个值等。也可以是显示某些提示。

3. 有穷性

算法的执行步骤是有限的，算法的执行时间也是有限的。

4. 确定性

算法的每个步骤都有确定的含义，不会出现二义性。

5. 可行性

算法是可用的，也就是能够解决当前问题。

算法的设计要求:

1. 正确性

对于合法输入能够得到满足的结果,算法能够处理非法处理，并得到合理结果.算法对于边界数据和压力数据都能得到满足的结果。

2. 可读性

算法要方便阅读，理解和交流，只有自己能看得懂，其它人都看不懂，谈和好算法。

3. 健壮性

通俗的讲,一个好的算法应该具有捕获异常/处理异常的能力。另外，对于测试人员的压力测试、边界值测试等刁难的测试手段，算法应该能够轻松的扛过去。

4. 高性价比

利用最少的时间和资源得到满足要求的结果，可以通过(时间复杂度和空间复杂度来判定)。

通常判定一种算法的效率可以采用事后统计法和事前分析估算.

事后统计法缺点: 必须编写相应的测试程序，严重依赖硬件和运行时的环境，算法的数据采集相当的困难。

事前分析估算：　主要取决于问题的规模。

这里解释下时间复杂度和空间复杂度.

时间复杂度:

时间复杂度是对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。

公式： `T(n) = O( f(n) )` ,其中f(n)是问题规模n的函数，也就是执行某个操作的次数。

在没有特殊说明的情况下，我们所分析的时间复杂度都是指最坏的时间复杂度。

空间复杂度：

空间复杂度是指算法在计算机内执行时所需存储空间的度量，它也是数据规模n的函数。

公式： S(n) = O(f(n)),其中f(n)是在问题规模为n时所占用的内存空间大小。

大O表示法同样也适合空间复杂度。

常见算法

我们都知道，线性表分为无序线性表和有序线性表。

无序线性表的数据并不是按升序或者降序来排列的，所以在插入和删除时，没有什么必须遵守的规矩而可以插入在数据尾部或者删除在数据尾部（将待删除的数据和最后一个数据交换位置），但是在查找的时候，需要遍历整个数据集，影响了效率。
有序线性表的数据则想法，查找的时候因为数据有序，可以用二分法、插值法、斐波那契查找法来实现，但是，插入和删除需要维护有序的结构，会耗费大量的时间。
为了提高插入和删除的效率，二叉排序树登场了。

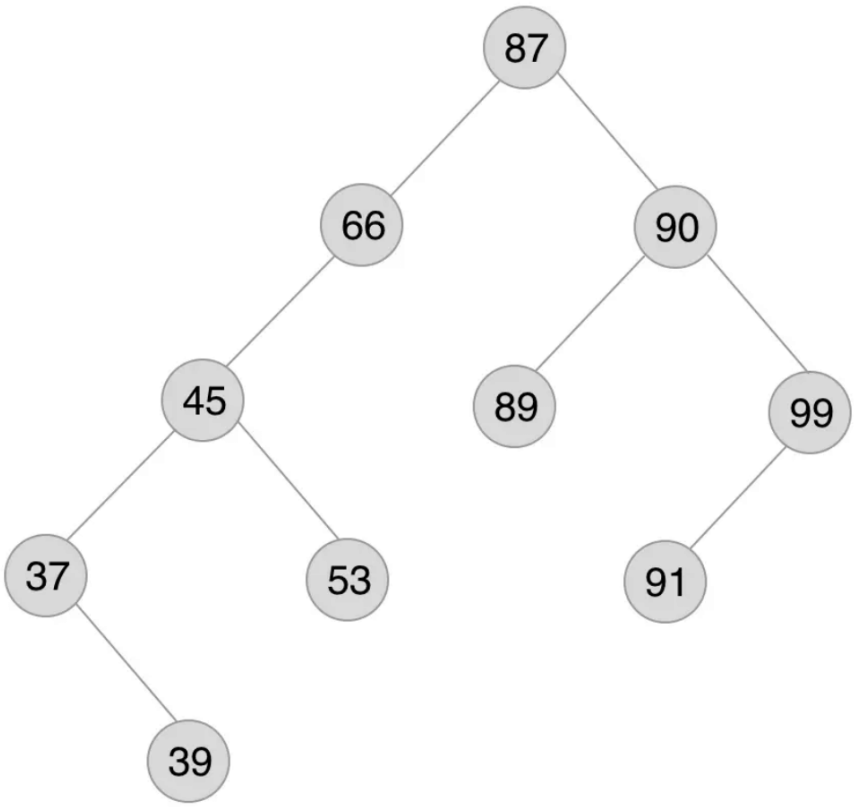
1. 二叉搜索树 (Binary Search Tree)
2. 平衡二叉查找树 (Balanced Binary Search Tree)
3. 红黑树 (Red-Black Tree)
4. B-树和B+树 (B-Tree)

- 二叉搜索树 (Binary Search Tree)

二叉搜索树的特点:

- 所有非叶子结点至多拥有两个树（Left和Right）；
- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左、右子树也分别为二叉搜索树。

二叉搜索树种最关键的是的特点是，左子树结点一定比父结点小，右子树结点一定比父结点大 .



二叉排序树范例

二叉搜索树查找:

通过观察上面的二叉搜索树，可以知道，查找树中一个值，可以从根结点开始查找，和根结点的值做比较，比根结点的值小，就在根结点的左子树中查找，比根结点的值大，就在根结点的右子树中查找。其他结点的行为与根结点的行为也是一样的。

以此出发，可以得到递归算法：

- 如果树是空的，则查找结束，无匹配。
- 如果被查找的值和根结点的值相等，查找成功。否则就在子树中继续查找。如果被查找的值小于根结点的值就选择左子树，大于根结点的值就选择右子树。 在理想情况下，每次比较过后，树会被砍掉一半，近乎折半查找。

遍历打印可以使用 中序遍历 ，打印出来的结果是从小到大的有序数组。

二叉搜索树插入:

新结点插入到树的叶子上，完全不需要改变树中原有结点的组织结构。插入一个结点的代价与查找一个不存在的数据的代价完全相同。

二叉排序的插入是建立在二叉排序的查找之上的，原因很简单，添加一个结点到合适的位置，就是通过查找发现合适位置，把结点直接放进去。
先来说一下插入函数,SearchBST(BiTree T, int key,BiTree f,BiTree *p)中指针p具有非常重要的作用:

- 若查找的key已经有在树中，则p指向该数据结点。
- 若查找的key没有在树中，则p指向查找路径上最后一个结点，而这里的最后一个结点的位置和key应该被放入的位置存在着简单关系（要么当树空时直接插入作为根结点， 要么当树非空时新结点作为查找路径终止结点的左孩子或者右孩子插入 ）。

二叉搜索树删除:

- 被删除的节点是叶子节点，这时候只要把这个节点删除，再把指向这个节点的父节点指针置为空就行
- 被删除的节点有左子树，或者有右子树，而且只有其中一个，那么只要把当前删除节点的父节点指向被删除节点的左子树或者右子树就行。
- 被删除的节点既有左子树而且又有右子树，这时候需要把左子树的最右边的节点或者右子树最左边的节点提到被删除节点的位置，为什么要这样呢，根据二叉查找树的性质，父节点的指针一定比所有左子树的节点值大而且比右子树的节点的值小，为了删除父节点不破坏二叉查找树的平衡性，应当把左子树最大的节点或者右子树最小的节点放在父节点的位置，这样的话才能维护二叉查找树的平衡性。（我是找的右子树的最小节点）

二叉树的删除可以算是二叉树最为复杂的操作，删除的时候要考虑到很多种情况：

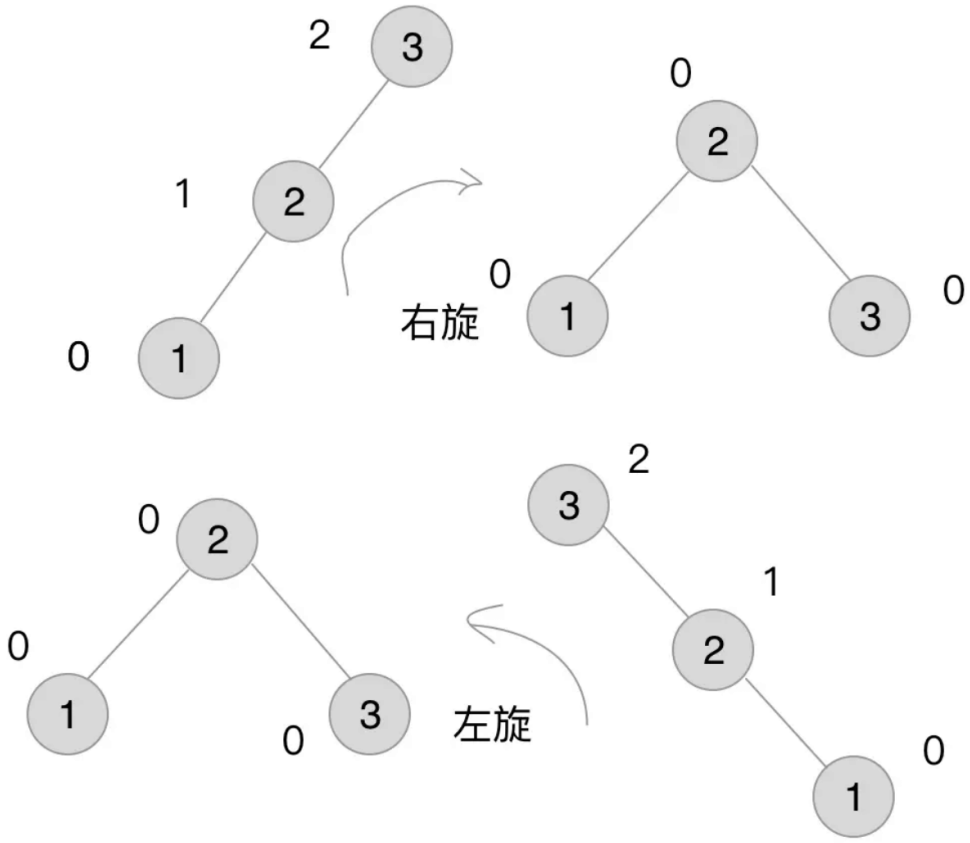
1. 被删除的节点是叶子节点
2. 被删除的节点只有左子节点
3. 被删除的节点只有右子节点
4. 被删除的有两个子节点

二叉搜索树的效率总结: 查找最好时间复杂度O(logN)，最坏时间复杂度O(N)。 插入删除操作算法简单，时间复杂度与查找差不多。

- 平衡二叉查找树 (Balanced Binary Search Tree)

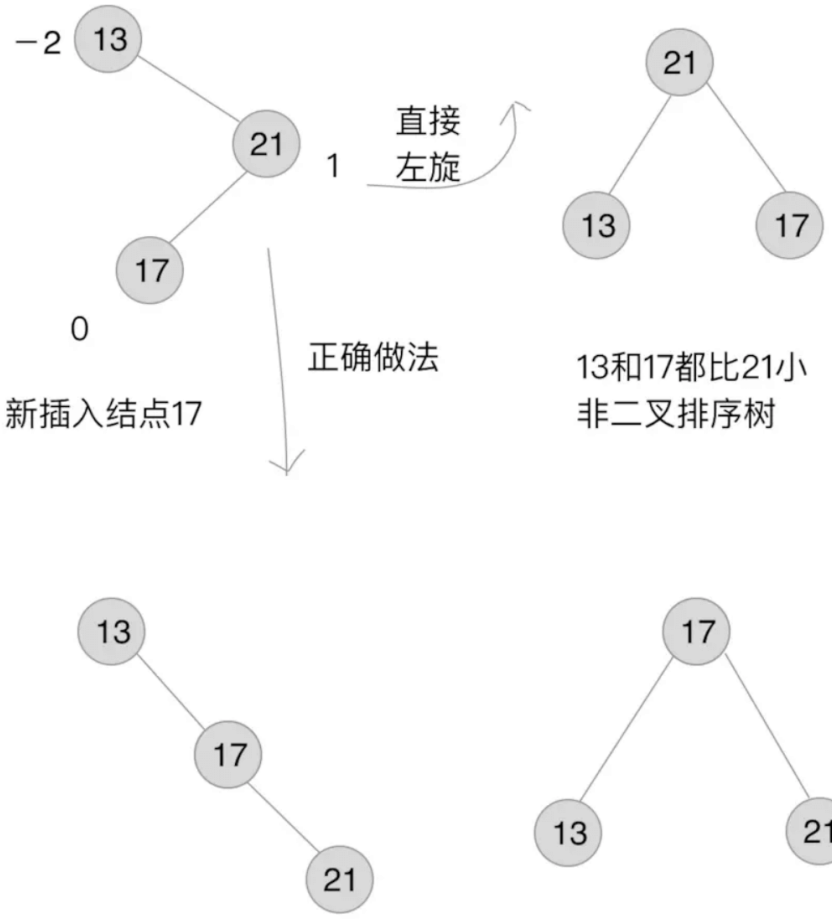
平衡二叉查找树 （Height-Balanced Binary Search Tree） 是一种二叉排序树，其中每一个结点的左子树和右子树的高度差不超过1（小于等于1）。
二叉树的平衡因子 （Balance Factor） 等于该结点的左子树深度减去右子树深度的值称为平衡因子。平衡因子只可能是－1，0，1。
距离插入结点最近的，且平衡因子的绝对值大于1的结点为根的子树，称为最小不平衡子树。

平衡二叉搜索树就是二叉树的构建过程中，每当插入一个结点，看是不是因为树的插入破坏了树的平衡性，若是，则找出最小不平衡树。在保持二叉树特性的前提下，调整最小不平衡子树中各个结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。因此主要是注意：步步调整，步步平衡 。



左旋和右旋的过程我们可以看到平衡因子从（0，1，2）变为（0，0，0），即是一种将非平衡状态转换为平衡状态的过程，这也是AVL树步步调整的核心。

再来观察一种复杂的情况:



新插入一个结点17，使得13的BF（−2）和21的BF（1）符号相反，如果直接左旋，调整后的树就不再是二叉排序树了。因此，正确做法是先在step1中调整符号，然后才能在step2中进行平衡操作。

由此，可以总结出平衡操作中非常必要的符号统一操作：

> 最小不平衡子树的BF和它的子树的BF符号相反时，就需要对结点先进行一次旋转使得符号相同，再 反向旋转一次 才能够完成平衡操作。

- 红黑树（Red–black tree）

红黑树（Red–black tree）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它在1972年由鲁道夫·贝尔发明，被称为"对称二叉B树"，它现代的名字源于Leo J. Guibas和Robert Sedgewick于1978年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 在 O(log n)时间内完成查找，插入和删除，这里的n是树中元素的数目。

B/B+ 树就是N叉（N-ary）平衡树了，每个节点可以有更多的孩子，新的值可以插在已有的节点里，而不需要改变树的高度，从而大量减少重新平衡和数据迁移的次数，这非常适合做数据库索引这种需要持久化在磁盘，同时需要大量查询和插入操作的应用。

红黑树用途和好处:

红黑树和AVL树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。这不只是使它们在时间敏感的应用，如实时应用（real time application）中有价值，而且使它们有在提供最坏情况担保的其他数据结构中作为基础模板的价值；例如，在计算几何中使用的很多数据结构都可以基于红黑树实现。

红黑树在函数式编程中也特别有用，在这里它们是最常用的持久数据结构（persistent data structure）之一，它们用来构造关联数组和集合，每次插入、删除之后它们能保持为以前的版本。除了O(log n)}的时间之外，红黑树的持久版本对每次插入或

删除需要O(log n)的空间。

红黑树是2-3-4树的一种等同。换句话说，对于每个2-3-4树，都存在至少一个数据元素是同样次序的红黑树。在2-3-4树上的插入和删除操作也等同于在红黑树中颜色翻转和旋转。这使得2-3-4树成为理解红黑树背后的逻辑的重要工具，这也是很多介绍算法的教科书在红黑树之前介绍2-3-4树的原因，尽管2-3-4树在实践中不经常使用。

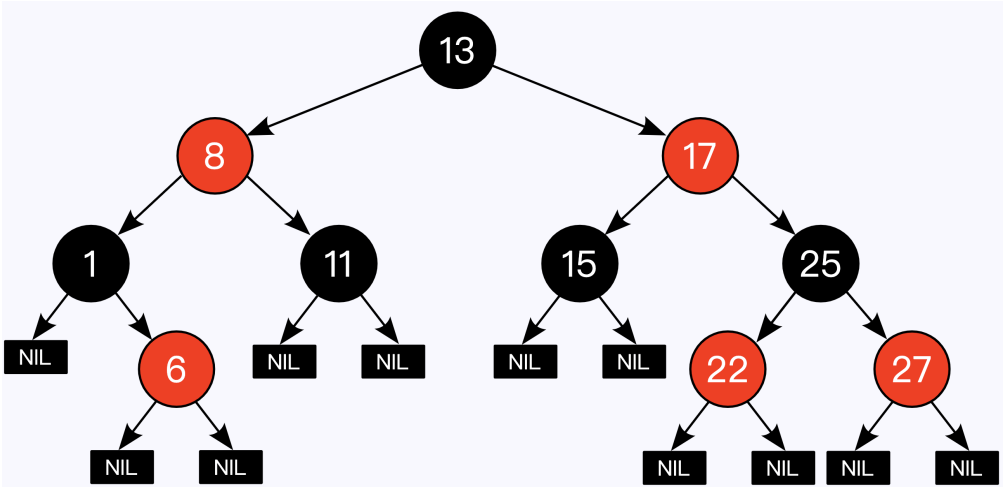
红黑树相对于AVL树来说，牺牲了部分平衡性以换取插入或者删除操作时少量的旋转操作，整体来说性能要优于AVL树。

红黑树性质:

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外.树中的结点包含5个属性：color、key、left、right和p。如果一个结点没有子结点或父结点，则该结点相应指针属性值为NIL。

红黑树要求：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是NIL节点）。
4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。



这些约束确保了红黑树的关键特性：从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些性质确保了这个结果，注意到性质4导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

在很多树数据结构的表示中，一个节点有可能只有一个子节点，而叶子节点包含数据。用这种范例表示红黑树是可能的，但是这会改变一些性质并使算法复杂。为此，本文中我们使用”nil叶子”或”空（null）叶子”，如上图所示，它不包含数据而只充当树在此结束的指示。这些节点在绘图中经常被省略，导致了这些树好像同上述原则相矛盾，而实际上不是这样。与此有关的结论是所有节点都有两个子节点，尽管其中的一个或两个可能是空叶子。

红黑树相比于BST和AVL树有什么优点？

红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。

相比于BST，因为红黑树可以能确保树的最长路径不大于两倍的最短路径的长度，所以可以看出它的查找效果是有最低保证的。在最坏的情况下也可以保证O(logN)的，这是要好于二叉查找树的。因为二叉查找树最坏情况可以让查找达到O(N)。

红黑树的算法时间复杂度和AVL相同，但统计性能比AVL树更高，所以在插入和删除中所做的后期维护操作肯定会比红黑树要耗时好多，但是他们的查找效率都是O(logN)，所以红黑树应用还是高于AVL树的。实际上插入，AVL 树和红黑树的速度取决于你所插入的数据.如果你的数据分布较好,则比较宜于采用 AVL树(例如随机产生系列数)，但是如果你想处理比较杂乱的情况,则红黑树是比较快的。

- AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；
- 而红黑树是弱平衡的，用非严格的平衡来换取增删节点时候旋转次数的降低；
- 所以简单说，查找的次数远远大于插入和删除，那么选择AVL树；如果搜索、插入删除次数几乎差不多，应该选择RB树。

红黑树的应用:

- 在Java中， TreeMap和TreeSet，Java 8中HashMap中TreeNode节点都采用了红黑树实现。
- C++中，STL的map和set也应用了红黑树；
- Linux进程调度Completely Fair Scheduler；
- 用红黑树管理进程控制块epoll在内核中的实现，用红黑树管理事件块；
- Nginx中，用红黑树管理timer等；
- 红黑树的各种操作的时间复杂度是O(lgn)，逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，IO次数多查找慢，效率低。

算法练习

- 数组

1. 实现一个支持动态扩容的数组
2. 实现一个大小固定的有序数组，支持动态增删改操作
3. 实现两个有序数组合并为一个有序数组

- 链表

1. 实现单链表、循环链表、双向链表，支持增删操作
2. 实现单链表反转
3. 实现两个有序的链表合并为一个有序链表
4. 实现求链表的中间结点

- 栈

1. 用数组实现一个顺序栈
2. 用链表实现一个链式栈
3. 编程模拟实现一个浏览器的前进、后退功能

- 队列

1. 用数组实现一个顺序队列
2. 用链表实现一个链式队列
3. 实现一个循环队列

- 递归

1. 编程实现斐波那契数列求值f(n)=f(n-1)+f(n-2)
2. 编程实现求阶乘n!
3. 编程实现一组数据集合的全排列

- 排序

1. 实现归并排序、快速排序、插入排序、冒泡排序、选择排序
2. 编程实现O(n)时间复杂度内找到一组数据的第K大元素

- 二分查找

1. 实现一个有序数组的二分查找算法
2. 实现模糊二分查找算法（比如大于等于给定值的第一个元素）

- 散列表

1. 实现一个基于链表法解决冲突问题的散列表
2. 实现一个LRU缓存淘汰算法

- 字符串

1. 实现一个字符集，只包含a～z这26个英文字母的Trie树
2. 实现朴素的字符串匹配算法

- 二叉树

1. 实现一个二叉查找树，并且支持插入、删除、查找操作
2. 实现查找二叉查找树中某个节点的后继、前驱节点
3. 实现二叉树前、中、后序以及按层遍历

- 堆

1. 实现一个小顶堆、大顶堆、优先级队列
2. 实现堆排序
3. 利用优先级队列合并K个有序数组
4. 求一组动态数据集合的最大Top K

- 图

1. 实现有向图、无向图、有权图、无权图的邻接矩阵和邻接表表示方法
2. 实现图的深度优先搜索、广度优先搜索
3. 实现Dijkstra算法、A*算法
4. 实现拓扑排序的Kahn算法、DFS算法

- 回溯

1. 利用回溯算法求解八皇后问题
2. 利用回溯算法求解0-1背包问题

- 分治

1. 利用分治算法求一组数据的逆序对个数

- 动态规划

1. 0-1背包问题
2. 最小路径和
3. 编程实现莱文斯坦最短编辑距离
4. 编程实现查找两个字符串的最长公共子序列
5. 编程实现一个数据序列的最长递增子序列

参考资料

- Visualizations Algorithms
- Algorithms

数据库

数据库

通常在开发中常用的数据库Mysql开始，MySQL是一个关系型数据库管理系统，由瑞典MySQL AB 公司开发，目前属于 Oracle 旗下公司。MySQL 最流行的关系型数据库管理系统，在应用方面MySQL是最好的 RDBMS (Relational Database Management System，关系数据库管理系统) 应用软件之一。

MySQL基础:

- MySQL的多存储引擎架构
- MySQL的数据类型
- 数据库的事务
- 数据库ACID
- 数据库中的范式
- 并发一致性问题
- 事务隔离级别
- 存储引擎
- MySQL的索引
- 联合索引
- 主键外键和索引的区别
- 聚集索引与非聚集索引
- 数据库中的分页查询
- 数据库中最大的连接数参数意义
- Redis数据库
- 分库分表
- 主从复制与读写分离
- 查询性能优化
- 锁类型

MySQL的多存储引擎架构

MySQL作为一个大型的网络程序、数据管理系统，架构非常复杂。

MySQL的数据类型

1. 整型

类型	存储	存储	最小值	最大值
	byte	bit	signed	signed
TINYINT	1	8	-2^7 = -128	2^7-1 = 127

类型	存储	存储	最小值	最大值
SMALLINT	2	16		
MEDIUMINT	3	24		
INT	4	32	-2^{31} = -2147483648	2^{31}-1 = 2147483647
BIGINT	8	64		

TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT 分别使用 8, 16, 24, 32, 64 位存储空间，一般情况下越小的列越好。

INT(11) 中的数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的。

2. 浮点数

FLOAT 和 DOUBLE 为浮点类型，DECIMAL 为高精度小数类型。CPU 原生支持浮点运算，但是不支持 DECIMAl 类型的计算，因此 DECIMAL 的计算比浮点类型需要更高的代价。

FLOAT、DOUBLE 和 DECIMAL 都可以指定列宽，例如 DECIMAL(18, 9) 表示总共 18 位，取 9 位存储小数部分，剩下 9 位存储整数部分。

3. 字符串

MySQL中主要有 CHAR 和 VARCHAR 两种字符串类型，一种是定长的，一种是变长的。

VARCHAR 这种变长类型能够节省空间，因为只需要存储必要的内容。但是在执行 UPDATE 时可能会使行变得比原来长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM 会将行拆成不同的片段存储，而 InnoDB 则需要分裂页来使行放进页内。

VARCHAR 会保留字符串末尾的空格，而 CHAR 会删除。

4. 时间和日期

MySQL中提供了两种相似的日期时间类型：DATATIME 和 TIMESTAMP。

- DATATIME

能够保存从 1001 年到 9999 年的日期和时间，精度为秒，使用 8 字节的存储空间。

它与时区无关。

默认情况下，MySQL 以一种可排序的、无歧义的格式显示 DATATIME 值，例如“2008-01-16 22:37:08”，这是 ANSI 标准定义的日期和时间表示方法。

- TIMESTAMP

和 UNIX 时间戳相同，保存从 1970 年 1 月 1 日午夜（格林威治时间）以来的秒数，使用 4 个字节，只能表示从 1970 年 到 2038 年。

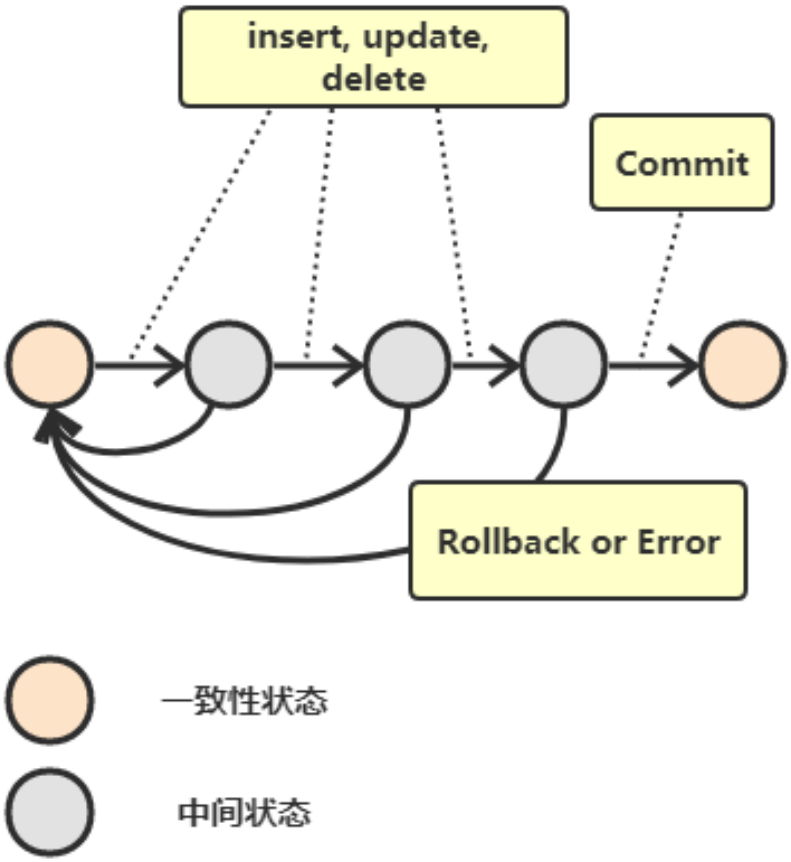
它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL 提供了 `FROM_UNIXTIME()` 函数把 UNIX 时间戳转换为日期，并提供了 `UNIX_TIMESTAMP()` 函数把日期转换为 UNIX 时间戳。

默认情况下，如果插入时没有指定 TIMESTAMP 列的值，会将这个值设置为当前时间。

应该尽量使用 `TIMESTAMP` ，因为它比 `DATETIME` 空间效率更高。

数据库的事务



数据库的事务指的是满足数据库的 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。

MySQL 中默认采用自动提交(AUTOCOMMIT)模式。如果不显式使用 START TRANSACTION 语句来开始一个事务，那么每个查询都会被当做一个事务自动提交。

数据库ACID

1. 原子性（Atomicity）

原子性是指事务是一个不可分割的工作单位，事务中的操作要么全部成功，要么全部失败。比如在同一个事务中的SQL语句，要么全部执行成功，要么全部执行失败。
回滚可以用日志来实现，日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。

2. 一致性（Consistency）

事务必须使数据库从一个一致性状态变换到另外一个一致性状态。以转账为例子，A向B转账，假设转账之前这两个用户的钱加起来总共是100，那么A向B转账之后，不管这两个账户怎么转，A用户的钱和B用户的钱加起来的总额还是100，这个就是事务的一致性。

3. 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

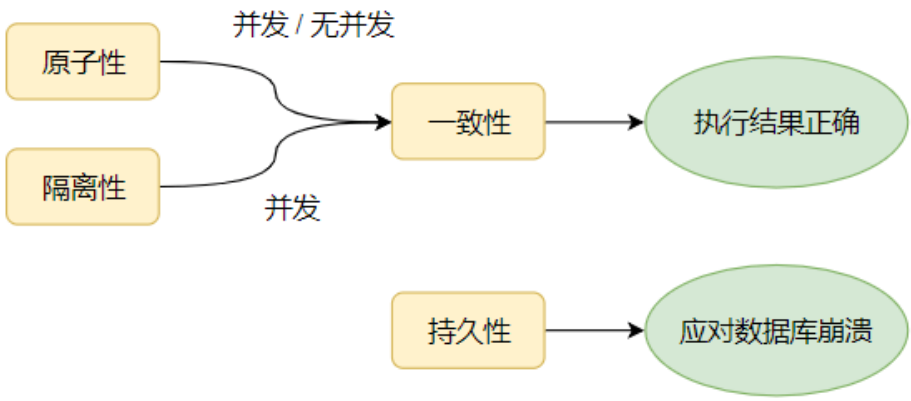
4. 持久性（Durability）

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

可以通过数据库备份和恢复来实现，在系统发生奔溃时，使用备份的数据库进行数据恢复。

事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：

- 只有满足一致性，事务的执行结果才是正确的。
- 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性。
- 在并发的情况下，多个事务并发执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。
- 事务满足持久化是为了能应对数据库奔溃的情况。

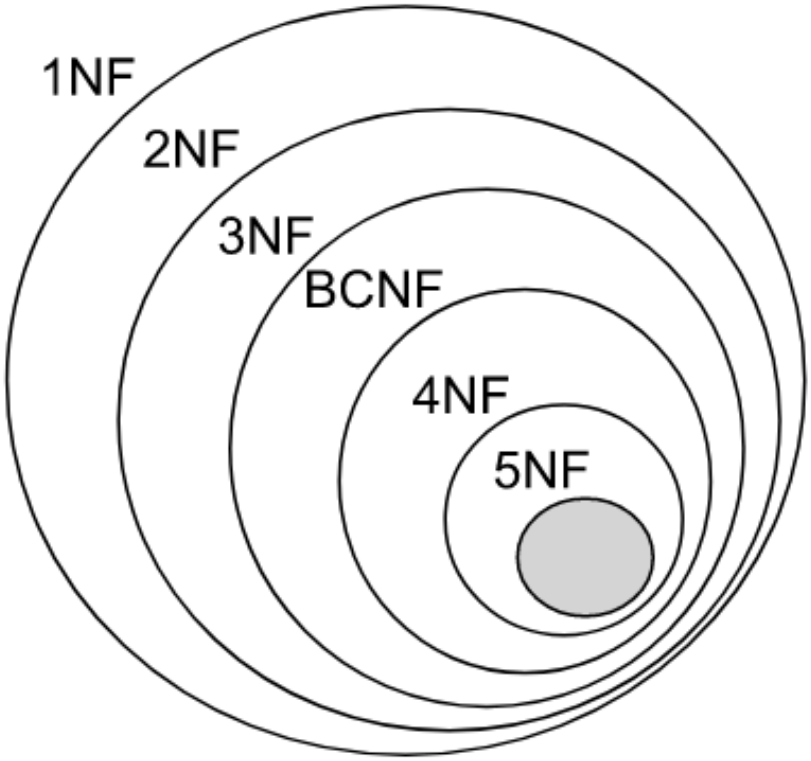


数据库中的范式

满足最低要求的范式是第一范式（1NF）。在第一范式的基础上进一步满足更多规范要求的称为第二范式（2NF），其余范式以次类推。一般说来，数据库只需满足第三范式 (3NF）就可以了。范式的包含关系。一个数据库设计如果符合第二范式，一定也符合第一范式。如果符合第三范式，一定也符合第二范式...

- 1NF：属性不可分.
- 2NF：属性完全依赖于主键 [消除部分子函数依赖].
- 3NF：属性不依赖于其它非主属性 [消除传递依赖].
- BCNF（巴斯-科德范式）：在1NF基础上，任何非主属性不能对主键子集依赖[在3NF基础上消除对主码子集的依赖].
- 4NF：要求把同一表内的多对多关系删除.
- 5NF（完美范式）：从最终结构重新建立原始结构.

范式理论是为了解决以上提到四种异常。高级别范式的依赖于低级别的范式，1NF 是最低级别的范式。



1. 第一范式 (1NF)

属性不可分。

2. 第二范式 (2NF)

每个非主属性完全函数依赖于键码。

可以通过分解来满足。

分解前

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	60
2	学生-2	学院-2	院长-2	课程-2	90
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

以上学生课程关系中，{Sno, Cname} 为键码，有如下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname
- Sno, Cname-> Grade

Grade 完全函数依赖于键码，它没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept 和 Mname 都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次，造成大量冗余数据。

分解后

关系-1

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

有以下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname

关系-2

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

有以下函数依赖：

- Sno, Cname -> Grade

3. 第三范式 (3NF)

非主属性不传递函数依赖于键码。

上面的 关系-1 中存在以下传递函数依赖：

- Sno -> Sdept -> Mname

可以进行以下分解：

关系-11

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

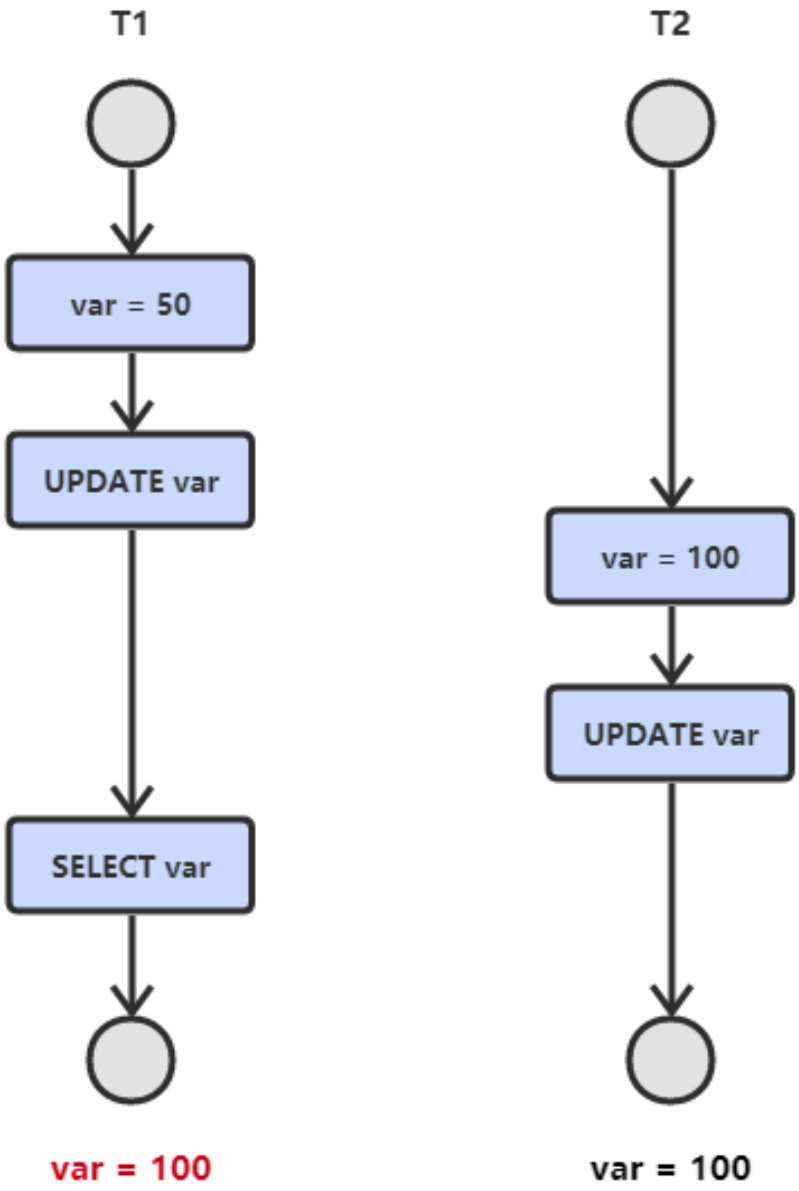
关系-12

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

并发一致性问题

1. 丢失修改

T1 和 T2 两个事务都对一个数据进行修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。



2. 脏数据读取
   （针对未提交数据）如果一个事务中对数据进行了更新，但事务还没有提交，另一个事务可以“看到”该事务没有提交的更新结果，这样造成的问题就是，如果第一个事务回滚，那么，第二个事务在此之前所“看到”的数据就是一笔脏数据。
   （脏读又称无效数据读出。一个事务读取另外一个事务还没有提交的数据叫脏读。）

应用示例：

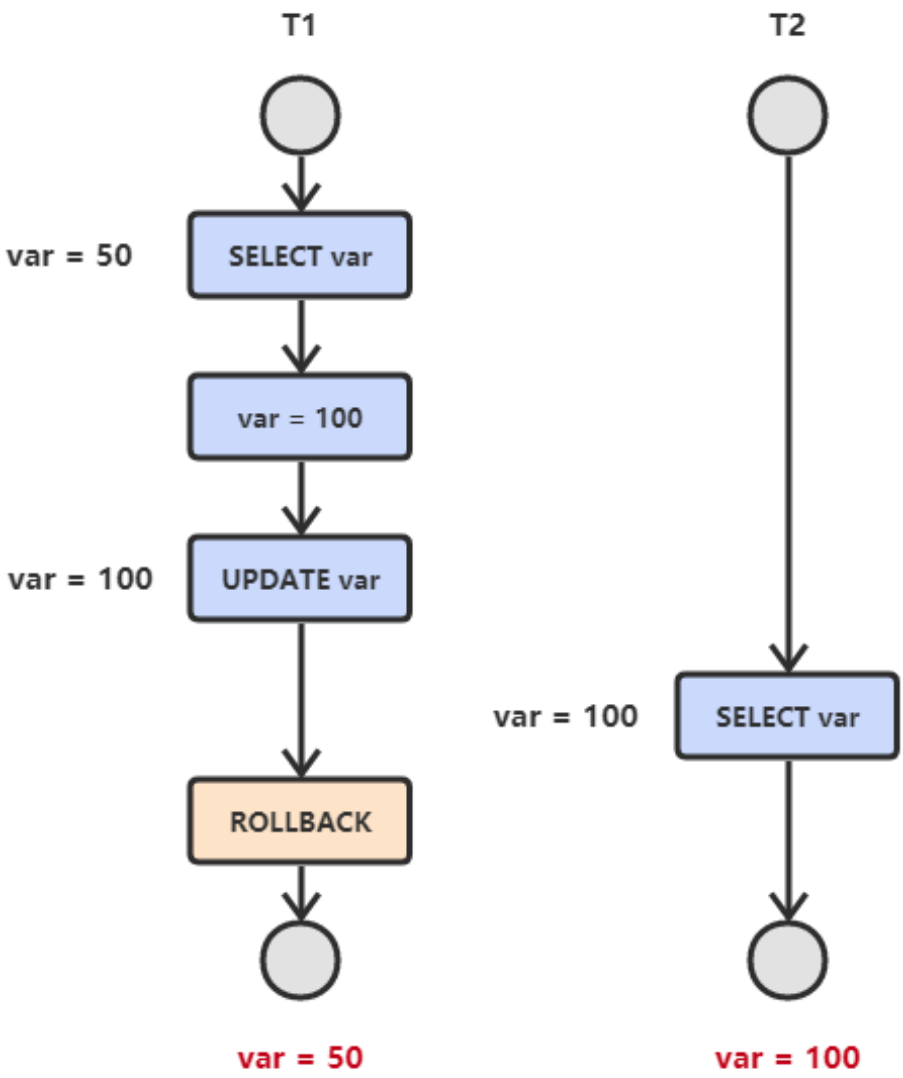
```
Jame的原工资为 1000，财务人员将Jame的工资改为了 8000（但未提交事务）

Jame读取自己的工资，发现自己的工资变为了 8000，欢天喜地！

而财务发现操作有误，回滚了事务，Jame的工资又变为了1000

像这样的事件，Jame记取的工资数8000就是一个脏数据。
```

解决办法：

```
把数据库的事务隔离级别调整到 READ_COMMITTED
```

图解：

```
1 修改一个数据，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。
```



3. 不可重复读

是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样在一个事务内两次读到的数据是不一样的，因此称为是不可重复读。（同时操作，事务1分别读取事务2操作时和提交后的数据，读取的记录内容不一致。不可重复读是指在同一个事务内，两个相同的查询返回了不同的结果。 ）

应用示例：

```
#在事务1中，Jame读取了自己的工资为1000，操作并没有完成

conl = getConnection();
select salary from employee empId ="Mary";
```

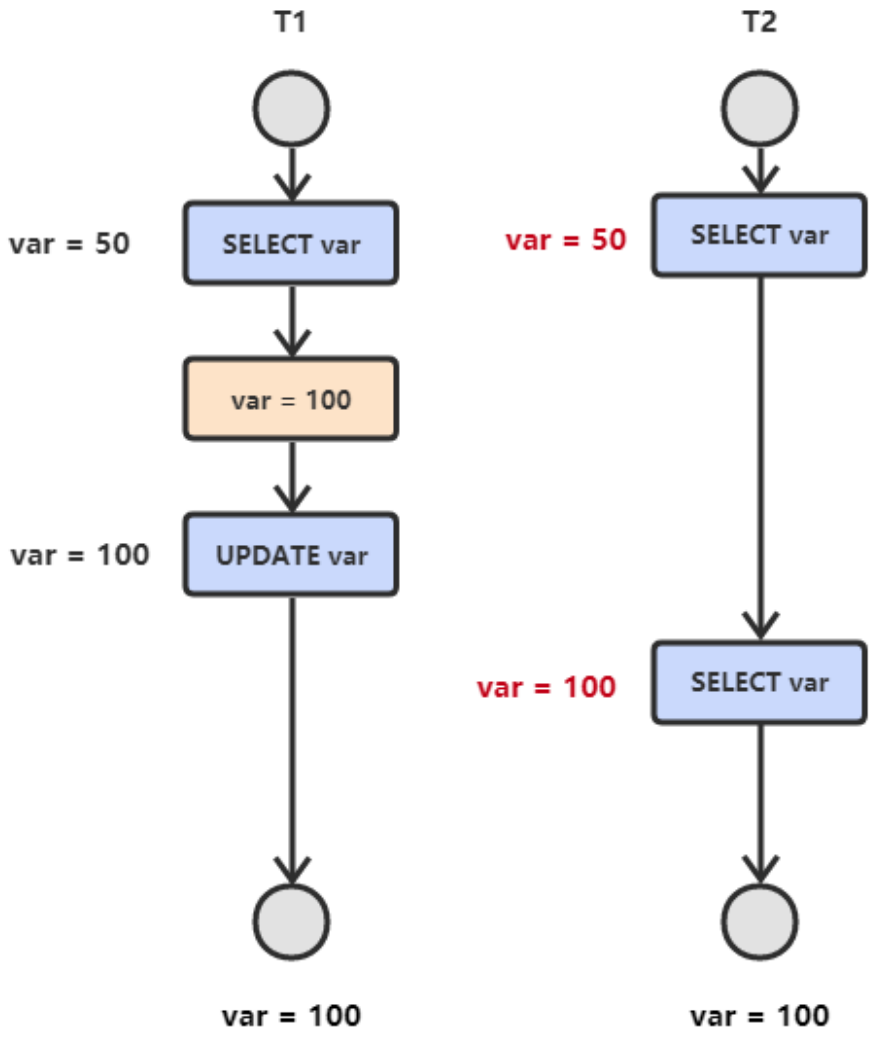
```
#在事务2中，这时财务人员修改了Jame的工资为 2000，并提交了事务.

con2 = getConnection();
update employee set salary = 2000;
con2.commit();

#在事务1中，Jame再次读取自己的工资时，工资变为了2000

#con1
select salary from employee empId ="Mary";
```

在一个事务中前后两次读取的结果并不致，导致了不可重复读。

解决办法：

```
如果只有在修改事务完全提交之后才可以读取数据，则可以避免该问题。把数据库的事务隔离级别调整到REPEATABLE_READ
```

T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。



4. 幻读

事务 T1 读取一条指定的 Where 子句所返回的结果集，然后 T2 事务新插入一行记录，这行记录恰好可以满足T1 所使用的查询条件。然后 T1 再次对表进行检索，但又看到了 T2 插入的数据。（和可重复读类似，但是事务 T2 的数据操作仅仅是插入和删除，不是修改数据，读取的记录数量前后不一致）.

幻读的重点在于新增或者删除 (数据条数变化).

同样的条件，第1次和第2次读出来的记录数不一样.

应用示例:

目前工资为1000的员工有10人

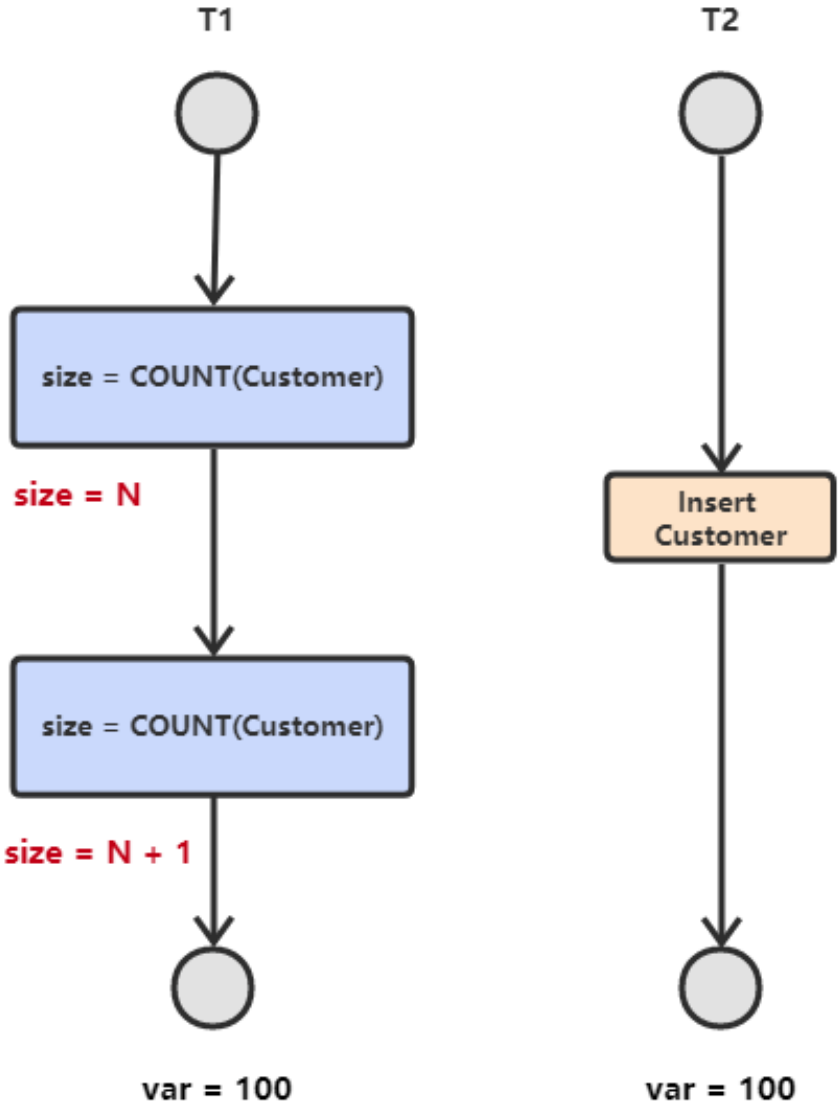
```
#事务1，读取所有工资为 1000 的员工（共读取 10 条记录 ）

con1 = getConnection();
Select * from employee where salary =1000;

#这时另一个事务向 employee 表插入了一条员工记录，工资也为 1000

con2 = getConnection();
Insert into employee(empId,salary) values("Lili",1000);
con2.commit();

#事务1再次读取所有工资为 1000的 员工（共读取到了 11 条记录，这就像产生了幻读）

#con1
select * from employee where salary =1000;
```

解决办法：

```
如果在操作事务完成数据处理之前，任何其他事务都不可以添加新数据，则可避免该问题。把数据库的事务隔离级别调整到 SERIALIZABLE_READ
```

图解：

T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。



事务隔离级别

1. 串行化 (Serializable)

所有事务一个接着一个的执行，这样可以避免幻读 (phantom read)，对于基于锁来实现并发控制的数据库来说，串行化要求在执行范围查询的时候，需要获取范围锁，如果不是基于锁实现并发控制的数据库，则检查到有违反串行操作的事务时，需回滚该事务。

2. 可重复读 (Repeated Read)

所有被 Select 获取的数据都不能被修改，这样就可以避免一个事务前后读取数据不一致的情况。但是却没有办法控制幻读，因为这个时候其他事务不能更改所选的数据，但是可以增加数据，即前一个事务有读锁但是没有范围锁，为什么叫做可重复读等级呢？那是因为该等级解决了下面的不可重复读问题。

注意：现在主流数据库都使用 MVCC 并发控制，使用之后RR（可重复读）隔离级别下是不会出现幻读的现象。

3. 读已提交 (Read Committed)

被读取的数据可以被其他事务修改，这样可能导致不可重复读。也就是说，事务读取的时候获取读锁，但是在读完之后立即释放(不需要等事务结束)，而写锁则是事务提交之后才释放，释放读锁之后，就可能被其他事务修改数据。该等级也是 SQL Server 默认的隔离等级。

4. 读未提交 (Read Uncommitted)

最低的隔离等级，允许其他事务看到没有提交的数据，会导致脏读。

总结

- 四个级别逐渐增强，每个级别解决一个问题，每个级别解决一个问题，事务级别遇到，性能越差，大多数环境(Read committed 就可以用了)

隔离级别	脏读	不可重复读	幻影读
未提交读	√	√	√
提交读	×	√	√
可重复读	×	×	√
可串行化	×	×	×

存储引擎

MySQL中提供了多个存储引擎，包括处理事务安全表的引擎和处理非事务安全表的引擎。在 MySQL 中，不需要在整个服务器中使用同一种存储引擎，针对具体的要求，可以对每一个表使用不同的存储引擎。

MySQL 中的数据用各种不同的技术存储在文件（或者内存）中。这些技术中的每一种技术都使用不同的存储机制、索引技巧、锁定水平并且最终提供广泛的不同的功能和能力。通过选择不同的技术，你能够获得额外的速度或者功能，从而改善你的应用的整体功能。存储引擎说白了就是如何存储数据、如何为存储的数据建立索引和如何更新、查询数据等技术的实现方法。

通常，如果你在研究大量的临时数据，你也许需要使用内存存储引擎。内存存储引擎能够在内存中存储所有的表格数据。又或者，你也许需要一个支持事务处理的数据库（以确保事务处理不成功时数据的回退能力）。

在MySQL中有很多存储引擎，每种存储引擎大相径庭，那么又改如何选择呢？

`MySQL 5.5` 以前的默认存储引擎是 `MyISAM` ， `MySQL 5.5` 之后的默认存储引擎是 `InnoDB`

不同存储引起都有各自的特点，为适应不同的需求，需要选择不同的存储引擎，所以首先考虑这些存储引擎各自的功能和兼容。

1. MyISAM

MySQL 5.5 版本之前的默认存储引擎，在 `5.0` 以前最大表存储空间最大 `4G` ， `5.0` 以后最大 `256TB` 。

Myisam 存储引擎由 `.myd` （数据）和 `.myi` （索引文件）组成， `.frm` 文件存储表结构（所以存储引擎都有）

特性

- 并发性和锁级别 （对于读写混合的操作不好，为表级锁，写入和读互斥）
- 表损坏修复
- Myisam 表支持的索引类型（全文索引）
- Myisam 支持表压缩（压缩后，此表为只读，不可以写入。使用 myisampack 压缩）

应用场景

- 没有事务
- 只读类应用（插入不频繁，查询非常频繁）
- 空间类应用（唯一支持空间函数的引擎）
- 做很多 count 的计算

2. InnoDB

MySQL 5.5 及之后版本的默认存储引擎

特性

- InnoDB为事务性存储引擎
- 完全支持事物的 ACID 特性
- Redo log （实现事务的持久性） 和 Undo log（为了实现事务的原子性，存储未完成事务log，用于回滚）
- InnoDB支持行级锁
- 行级锁可以最大程度的支持并发
- 行级锁是由存储引擎层实现的

应用场景

- 可靠性要求比较高，或者要求事务
- 表更新和查询都相当的频繁，并且行锁定的机会比较大的情况。

3. CSV

文件系统存储特点

- 数据以文本方式存储在文件中
- `.csv` 文件存储表内容
- `.csm` 文件存储表的元数据，如表状态和数据量
- `.frm` 存储表的结构

CSV存储引擎特点

- 以 CSV 格式进行数据存储
- 所有列必须都是不能为 NULL
- 不支持索引
- 可以对数据文件直接编辑（其他引擎是二进制存储，不可编辑）

引用场景

- 作为数据交换的中间表

4. Archive

特性

- 以 zlib 对表数据进行压缩，磁盘 I/O 更少
- 数据存储在ARZ为后缀的文件中（表文件为 `a.arz` ， `a.frm` ）
- 只支持 insert 和 select 操作（不可以 delete 和 update，会提示没有这个功能）
- 只允许在自增ID列上加索引

应用场景

- 日志和数据采集类应用

5. Memory

特性:

- 也称为 HEAP 存储引擎，所以数据保存在内存中（数据库重启后会导致数据丢失）
- 支持 HASH 索引（等值查找应选择 HASH）和 BTree 索引（范围查找应选择）
- 所有字段都为固定长度，varchar(10) == char(10)
- 不支持 BLOG 和 TEXT 等大字段
- Memory 存储使用表级锁（性能可能不如 innodb）
- 最大大小由 `max_heap_table_size` 参数决定
- Memory存储引擎默认表大小只有 `16M` ，可以通过调整 `max_heap_table_size` 参数

应用场景:

- 用于查找或是映射表，例如右边和地区的对应表
- 用于保存数据分析中产生的中间表
- 用于缓存周期性聚合数据的结果表

注意： Memory 数据易丢失，所以要求数据可再生

6. Federated

特性

- 提供了访问远程 MySQL 服务器上表的方法
- 本地不存储数据，数据全部放在远程服务器上

使用 Federated,默认是禁止的。如果需要启用，需要在启动时增加Federated参数

独立表空间和系统表空间应该如何抉择？

可以比较下两者之间的差别进行选择：

- 系统表空间：无法简单的收缩大小（这很恐怖，会导致 ibdata1 一直增大，即使删除了数据也不会变小）
- 独立表空间：可以通过 optimize table 命令收缩系统文件
- 系统表空间：会产生I/O瓶颈（因为只有一个文件）
- 独立表空间：可以向多个文件刷新数据

注意：对于Innodb引擎使用独立表空间（mysql5.6版本以后默认是独立表空间）

系统表转移为独立表的步骤（非常繁琐）

- 使用 mysqldump 导出所有数据库表数据
- 停止 mysql 服务，修改参数，并且删除Innodb相关文件
- 重启 mysql 服务，重建mysql系统表空间

- 重新导入数据
- 问：如何选择存储引擎，需要考虑以下的条件：
- 是否需要事务
- 是否可以热备份
- 崩溃恢复
- 存储引擎的特有特性

注意:不要混合使用存储引擎

推荐：Innodb

- MyISAM和InnoDB引擎的区别?

两者之间的区别：

- MyISAM 不支持外键，而 InnoDB 支持
- MyISAM 是非事务安全型的，而 InnoDB 是事务安全型的。
- MyISAM 锁的粒度是表级，而 InnoDB 支持行级锁定。
- MyISAM 支持全文类型索引，而 InnoDB 不支持全文索引。
- MyISAM 相对简单，所以在效率上要优于 InnoDB，小型应用可以考虑使用 MyISAM。
- MyISAM 表是保存成文件的形式，在跨平台的数据转移中使用 MyISAM 存储会省去不少的麻烦。
- InnoDB 表比 MyISAM 表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（alter table tablename type=innodb）。

应用场景：

- MyISAM 管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的 SELECT 查询，那么 MyISAM 是更好的选择。
- InnoDB 用于事务处理应用程序，具有众多特性，包括 ACID 事务支持。如果应用中需要执行大量的 INSERT或 UPDATE 操作，则应该使用 InnoDB，这样可以提高多用户并发操作的性能。

MySQL的索引

索引使用的场景

索引能够轻易将查询性能提升几个数量级。

- 对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效。
- 对于中到大型的表，索引就非常有效。
- 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

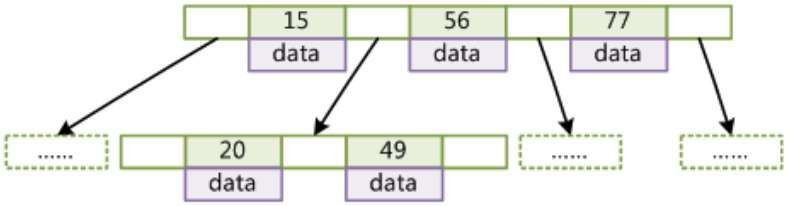
索引分类

特性	说明	InnoDB	MyISAM	MEMORY
B树索引 (B-tree indexes)	自增ID物理连续性更高， 二叉树， 红黑树高度不可控	√	√	√
R树索引 (R-tree indexes)	空间索引		√	
哈希索引 (Hash indexes)	无法做范围查询	√		√
全文索引 (Full-text indexes)		√	√	

B Tree 原理

B-tree（多路搜索树，并不是二叉的）是一种常见的数据结构,也就是我们常说的B树。使用B-tree结构可以显著减少定位记录时所经历的中间过程，从而加快存取速度。按照翻译，B 通常认为是Balance的简称。这个数据结构一般用于数据库的索引，综合效率较高。

B-Tree:



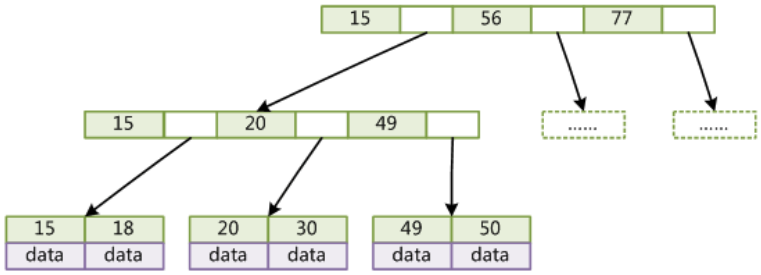
定义一条数据记录为一个二元组 [key, data]，B-Tree 是满足下列条件的数据结构：

- 所有叶节点具有相同的深度，也就是说 B-Tree 是平衡的；
- 一个节点中的 key 从左到右非递减排列；
- 如果某个指针的左右相邻 key 分别是 keyi 和 keyi+1，且不为 null，则该指针指向节点的（所有 key ≥ keyi） 且（key ≤ keyi+1）。

查找算法：首先在根节点进行二分查找，如果找到则返回对应节点的 data，否则在相应区间的指针指向的节点递归进行查找。

由于插入删除新的数据记录会破坏 B-Tree 的性质，因此在插入删除时，需要对树进行一个分裂、合并、旋转等操作以保持 B-Tree 性质。

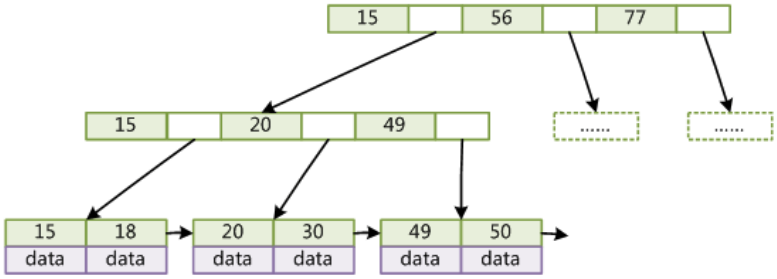
B+Tree:



与 B-Tree 相比，B+Tree 有以下不同点：

- 每个节点的指针上限为 2d 而不是 2d+1（d 为节点的出度）；
- 内节点不存储 data，只存储 key；
- 叶子节点不存储指针。

顺序访问指针：



一般在数据库系统或文件系统中使用的 B+Tree 结构都在经典 B+Tree 基础上进行了优化，在叶子节点增加了顺序访问指针，做这个优化的目的是为了提高区间访问的性能。

优势：

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B Tree 作为索引结构，主要有以下两个原因：

- 更少的检索次数

平衡树检索数据的时间复杂度等于树高 h，而树高大致为 O(h)=O(logdN)，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B Tree 的出度一般都非常大。红黑树的树高 h 很明显比 B Tree 大非常多，因此检索的次数也就更多。

B+Tree 相比于 B-Tree 更适合外存索引，因为 B+Tree 内节点去掉了 data 域，因此可以拥有更大的出度，检索效率会更高。

- 利用计算机预读特性

为了减少磁盘 I/O，磁盘往往不是严格按需读取，而是每次都会预读。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的旋转时间，因此速度会非常快。

操作系统一般将内存和磁盘分割成固态大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点，并且可以利用预读特性，相邻的节点也能够被预先载入。

参考：MySQL 索引背后的数据结构及算法原理

B+Tree 索引

B+Tree 索引是大多数 MySQL 存储引擎的默认索引类型。

因为不再需要进行全表扫描，只需要对树进行搜索即可，因此查找速度快很多。除了用于查找，还可以用于排序和分组。

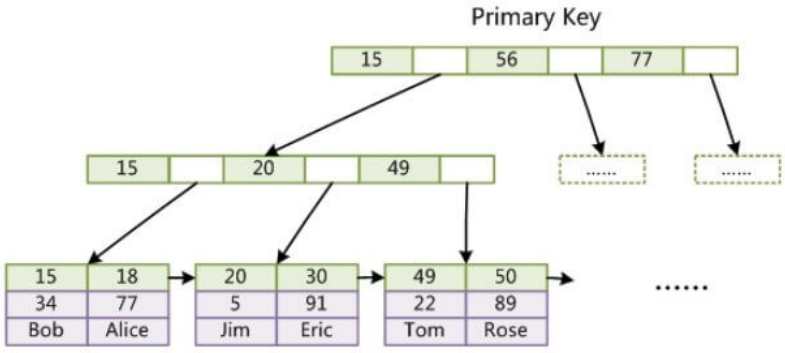
可以指定多个列作为索引列，多个索引列共同组成键。

B+Tree 索引适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。

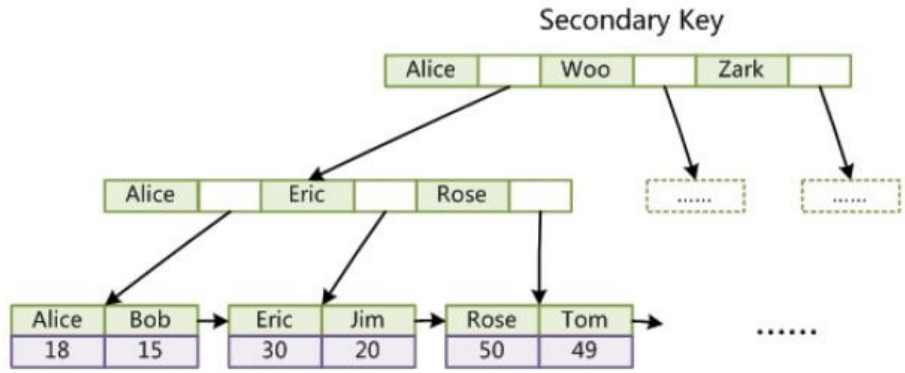
如果不是按照索引列的顺序进行查找，则无法使用索引。

InnoDB 的 B+Tree 索引分为**主索引**和**辅助索引**。

主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引。因为无法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。



辅助索引的叶子节点的 data域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。



哈希索引

InnoDB 引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

哈希索引能以 O(1) 时间进行查找，但是失去了有序性，它具有以下限制：

- 无法用于排序与分组；
- 只支持精确查找，无法用于部分查找和范围查找；

全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引一般使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

空间数据索引（R-Tree）

MyISAM 存储引擎支持空间数据索引，可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

索引的特点

- 可以加快数据库的检索速度
- 降低数据库插入、修改、删除等维护的速度
- 只能创建在表上，不能创建到视图上
- 既可以直接创建又可以间接创建
- 可以在优化隐藏中使用索引
- 使用查询处理器执行SQL语句，在一个表上，一次只能使用一个索引

索引的优点和缺点

索引的优点:

- 可以创建唯一性索引，保证数据库表中每一行数据的唯一性
- 大大加快数据的检索速度，这是创建索引的最主要的原因
- 加速数据库表之间的连接，特别是在实现数据的参考完整性方面特别有意义
- 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间
- 通过使用索引，可以在查询中使用优化隐藏器，提高系统的性能

索引的缺点:

- 可以创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加
- 索引需要占用物理空间，除了数据表占用数据空间之外，每一个索引还要占一定的物理空间，如果建立聚簇索引，那么需要的空间就会更大
- 当对表中的数据进行增加、删除和修改的时候，索引也需要维护，降低数据维护的速度

索引失效的情况

- 如果MySQL估计使用全表扫描比使用索引快，则不适用索引。

如果列key均匀分布在1和100之间，下面的查询使用索引就不是很好：select * from table_name where key>1 and key<90;

- 如果条件中有or，即使其中有条件带索引也不会使用

使用or的查询语句，如果在key1上有索引而在key2上没有索引，则该查询也不会走索引。

```
> select * from table_name where key1='a' or key2='b'
```

- 复合索引，如果索引列不是复合索引的第一部分，则不使用索引（即不符合最左前缀）

复合索引为(key1,key2),则查询而且不会使用索引。

```
> select * from table_name where key2='b'
```

- 如果like是以 % 开始的，则该列上的索引不会被使用。

使用like的查询语句，即使key1上存在索引，也不会被使用如果列类型是字符串，那一定要在条件中使用引号引起来，否则不会使用索引。

```
> select * from table_name where key1 like '%a'
```

- 如果列为字符串，则where条件中必须将字符常量值加引号，否则即使该列上存在索引，也不会被使用。

使用where的查询语句,key1列保存的是字符串，即使key1上有索引，也不会被使用。

```
select * from table_name where key1=1;
```

- 如果使用MEMORY/HEAP表，并且where条件中不使用“=”进行索引列，那么不会用到索引，head表只有在“=”的条件下才会使用索引

适合建立索引的情况

下面的查询适合建立索引：

- 如果SQL查询中出现在关键字order by、group by、distinct后面的字段，建立索引。
- 在union等集合操作的结果集字段上，建立索引。其建立索引的目的同上。
- 经常用作查询选择 where 后的字段，建立索引。
- 经常用作表连接 join 的属性上，建立索引。
- 考虑使用索引覆盖。对数据很少被更新的表，如果用户经常只查询其中的几个字段，可以考虑在这几个字段上建立索引，从而将表的扫描改变为索引的扫描。

为何选择用B+树做索引而不用B-树或红黑树

B+ 树只有叶节点存放数据，其余节点用来索引，而 B- 树是每个索引节点都会有 Data 域。所以从 InooDB 的角度来看，B+ 树是用来充当索引的，一般来说索引非常大，尤其是关系性数据库这种数据量大的索引能达到亿级别，所以为了减少内存的占用，索引也会被存储在磁盘上。

MySQL如何衡量查询效率呢？

主要是通过磁盘 IO 次数。

- B- 树 / B+ 树 的特点就是每层节点数目非常多，层数很少，目的就是为了就少磁盘 IO 次数，但是 B- 树的每个节点都有 data 域（指针），这无疑增大了节点大小，说白了增加了磁盘 IO 次数（磁盘 IO 一次读出的数据量大小是固定的，单个数据变大，每次读出的就少，IO 次数增多，一次 IO 多耗时），而 B+ 树除了叶子节点其它节点并不存储数据，节点小，磁盘 IO 次数就少。

- B+ 树所有的 Data 域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串起来。这样遍历叶子节点就能获得全部数据，这样就能进行区间访问啦。在数据库中基于范围的查询是非常频繁的，而 B 树不支持这样的遍历操作。

B 树和红黑树之间的区别:

- AVL 树和红黑树基本都是存储在内存中才会使用的数据结构。在大规模数据存储的时候，红黑树往往出现由于树的深度过大而造成磁盘 IO 读写过于频繁，进而导致效率低下的情况。之所以会出现这样的情况，是由于我们要获取磁盘上数据，需要先通过磁盘移动臂移动到数据所在的柱面，然后找到指定盘面，接着旋转盘面找到数据所在的磁道，最后对数据进行读写。磁盘IO代价主要花费在查找所需的柱面上，树的深度过大会造成磁盘IO频繁读写。根据磁盘查找存取的次数往往由树的高度所决定，所以，只要我们通过某种较好的树结构减少树的结构尽量减少树的高度，B树可以有多个子女，从几十到上千，可以降低树的高度。

- 数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。为了达到这个目的，在实际实现 B-Tree 还需要在每次新建节点时，申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，这样就实现了一个 node 只需一次 I/O。

联合索引

1. 什么是联合索引

两个或更多个列上的索引被称作联合索引，联合索引又叫复合索引。对于复合索引：Mysql 从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。

例如索引是key index (a,b,c)，可以支持[a]、[a,b]、[a,b,c] 3种组合进行查找，但不支 [b,c] 进行查找。当最左侧字段是常量引用时，索引就十分有效。

2. 命名规则

- 对于需要加索引的字段，要在 where 条件中
- 数据量少的字段不需要加索引
- 如果 where 条件中是OR关系，加索引不起作用
- 符合最左原则

3. 创建索引

在执行 CREATE TABLE 语句时可以创建索引，也可以单独用 CREATE INDEX 或 ALTER TABLE 来为表增加索引。

- ALTER TABLE

ALTER TABLE 用来创建普通索引、UNIQUE 索引或 PRIMARY KEY 索引。

```
ALTER TABLE table_name ADD INDEX index_name (column_list)
ALTER TABLE table_name ADD UNIQUE (column_list)
ALTER TABLE table_name ADD PRIMARY KEY (column_list)
```

其中 table_name 是要增加索引的表名，column_list 指出对哪些列进行索引，多列时各列之间用逗号分隔。索引名 index_name 可选，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE 允许在单个语句中更改多个表，因此可以在同时创建多个索引。

- CREATE INDEX

CREATE INDEX 可对表增加普通索引或 UNIQUE 索引。

```
CREATE INDEX index_name ON table_name (column_list)
CREATE UNIQUE INDEX index_name ON table_name (column_list)
```

table_name、index_name 和 column_list 具有与 ALTER TABLE 语句中相同的含义，索引名不可选。另外，不能用 CREATE INDEX 语句创建 PRIMARY KEY 索引。

4. 索引类型

在创建索引时，可以规定索引能否包含重复值。如果不包含，则索引应该创建为 PRIMARY KEY 或 UNIQUE 索引。对于单列惟一性索引，这保证单列不包含重复的值。对于多列惟一性索引，保证多个值的组合不重复。 PRIMARY KEY 索引和 UNIQUE 索引非常类似。

事实上，PRIMARY KEY 索引仅是一个具有名称 PRIMARY 的 UNIQUE 索引。这表示一个表只能包含一个 PRIMARY KEY，因为一个表中不可能具有两个同名的索引。 下面的SQL语句对 students 表在 sid 上添加 PRIMARY KEY 索引。 ALTER TABLE students ADD PRIMARY KEY (sid)。

5. 删除索引

可以利用 ALTER TABLE 或 DROP INDEX 语句来删除索引。类似于 CREATE INDEX 语句，DROP INDEX 可以在 ALTER TABLE 内部作为一条语句处理。

```
DROP INDEX index_name ON talbe_name
ALTER TABLE table_name DROP INDEX index_name
ALTER TABLE table_name DROP PRIMARY KEY
```

其中，前两条语句是等价的，删除掉 table_name 中的索引 index_name。

第3条语句只在删除 PRIMARY KEY 索引时使用，因为一个表只可能有一个 PRIMARY KEY 索引，因此不需要指定索引名。如果没有创建 PRIMARY KEY 索引，但表具有一个或多个 UNIQUE 索引，则 MySQL 将删除第一个 UNIQUE 索引。

如果从表中删除了某列，则索引会受到影响。对于多列组合的索引，如果删除其中的某列，则该列也会从索引中删除。如果删除组成索引的所有列，则整个索引将被删除。

6. 在什么情况下使用索引

- 通常为了快速查找匹配WHERE条件的行。
- 通常为了从考虑的条件中消除行。
- 如果表有一个multiple-column索引，任何一个索引的最左前缀可以通过使用优化器来查找行。
- 查询中与其它表关联的字，字段常常建立了外键关系
- 查询中统计或分组统计的字段

```
select max(hbs_bh) from zl_yhjbqk
select qc_bh,count(*) from zl_yhjbqk group by qc_bh
```

7. 注意事项

- 创建索引

对于查询占主要的应用来说，索引显得尤为重要。很多时候性能问题很简单的就是因为我们忘了添加索引而造成的，或者说没有添加更为有效的索引导致。如果不加
索引的话，那么查找任何哪怕只是一条特定的数据都会进行一次全表扫描，如果一张表的数据量很大而符合条件的结果又很少，那么不加索引会引起致命的性能下降。但是也不是什么情况都非得建索引不可，比如性别可能就只有两个值，建索引不仅没什么优势，还会影响到更新速度，这被称为过度索引。

- 复合索引

比如有一条语句是这样的：select * from users where area=’shanghai’ and age=17;

如果我们是在area和age上分别创建单个索引的话，由于mysql查询每次只能使用一个索引，所以虽然这样已经相对不做索引时全表扫描提高了很多效率，但是如果在area、age两列上创建复合索引的话将带来更高的效率。如果我们创建了(area, age,salary)的复合索引，那么其实相当于创建了(area,age,salary)、(area,age)、(area)三个索引，这被称为最佳左前缀特性。

因此我们在创建复合索引时应该将最常用作限制条件的列放在最左边，依次递减。

- 索引不会包含有NULL值的列

只要列中包含有NULL值都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此复合索引就是无效的。所以我们在数据库设计时不要让字段的默认值为NULL。

- 使用短索引

对串列进行索引，如果可能应该指定一个前缀长度。例如，如果有一个CHAR(255)的 列，如果在前10 个或20 个字符内，多数值是惟一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

- 排序的索引问题

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引。

- like语句操作

一般情况下不鼓励使用like操作，如果必须要使用，考虑如何使用也是一个问题。like “%name%” 不会使用索引，而like “name%”可以使用索引。

- 不要在列上进行运算

- 不使用NOT IN操作

NOT IN操作不会使用索引将进行全表扫描。NOT IN可以用NOT EXISTS代替

主键外键和索引的区别

	定义	作用	个数
主键	唯一标识一条记录， 不能有重复的，不允许为空	用来保证数据完整性	主键只能有一个
外键	表的外键是另一表的主键， 外键可以有重复的， 可以是空值	用来和其他表建立联系用的	一个表可以有多个外
索引	该字段没有重复值， 但可以有一个空值	是提高查询排序的速度	一个表可以有多个惟

聚集索引与非聚集索引

聚集索引一定是唯一索引。但唯一索引不一定是聚集索引。

聚集索引，在索引页里直接存放数据，而非聚集索引在索引页里存放的是索引，这些索引指向专门的数据页的数据。

数据库中的分页查询

Mysql 的 limit 用法：

```
SELECT * FROM table LIMIT [offset,] rows | rows OFFSET offset
```

LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1) 。

最基本的分页方式：SELECT ... FROM ... WHERE ... ORDER BY ... LIMIT ...

应用示例:

```
SELECT * FROM `video` ORDER BY `createTime`  desc LIMIT  10;
```

根据时间查询最近的10条数据.(降序排列)。

数据库中最大的连接数参数意义

数据库中最大的连接数参数意义:

- SetMaxOpenConns用于设置最大打开的连接数，默认值为0表示不限制。
- SetMaxIdleConns用于设置闲置的连接数。
- SetConnMaxLifetime用于设置连接超时

设置最大的连接数，可以避免并发太高导致连接mysql出现too many connections的错误。设置闲置的连接数则当开启的一个连接使用完成后可以放在池里等候下一次使用。

通常设置连接连接超时是500秒:

```
db.SetConnMaxLifetime(time.Second * 500) //设置连接超时500秒
```

Redis数据库

Redis是一个速度非常快的key-value非关系型数据库，可以存储键(key)与5种不同类型的值（value）之间的映射，可以将存储在内存中的键值对数据持久化到硬盘中。和 Memcached 类似，它支持存储的 value 类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set –有序集合)和 hash（哈希类型）。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。

Redis数据库与 Memcached 相比的不同:

- 两者都可用于存储键值映射，彼此性能也相差无几
- Redis 能够自动以两种不同的方式将数据写入硬盘
- Redis 除了能存储普通的字符串键之外，还可以存储其他4种数据结构，memcached 只能存储字符串键
- Redis 既能用作主数据库，由可以作为其他存储系统的辅助数据库

Redis应用场景:

- 缓存、任务队列、应用排行榜、网站访问统计、数据过期处理、分布式集群架构中的session分离。

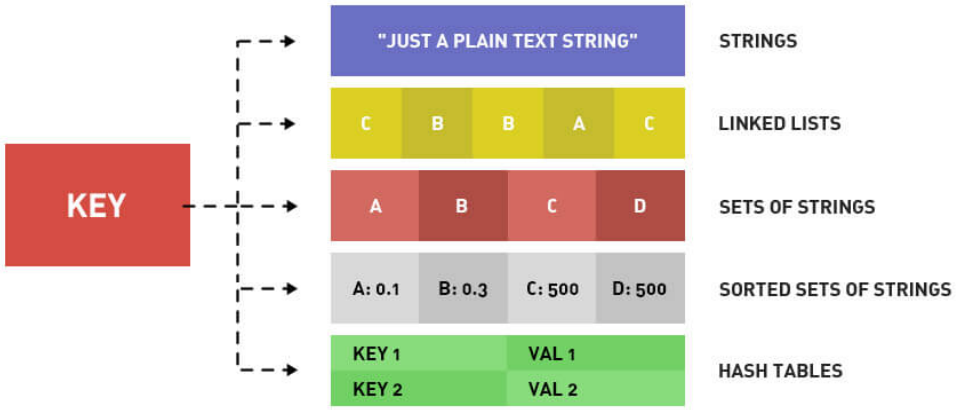
Redis特点：

- 高并发读写,读的速度是 110000次/s（11 W次/s）,写的速度是81000次/s（8.1W 次/s）

- 海量数据的高效存储和访问
- 高可扩展性和高可用性
- Redis命令大全

Redis的数据结构：

- STRING：可以是字符串、整数或者浮点数
- LIST：一个链表，链表上的每个节点都包含了一个字符串
- SET：包含字符串的无序收集器（unordered collection），并且被包含的每个字符串都是独一无二、各不相同的
- HAST：包含键值对的无序散列表
- ZSET：字符串成员（member）与浮点数分值（score）之间的有序映射，元素的排列顺序由分值的大小决定



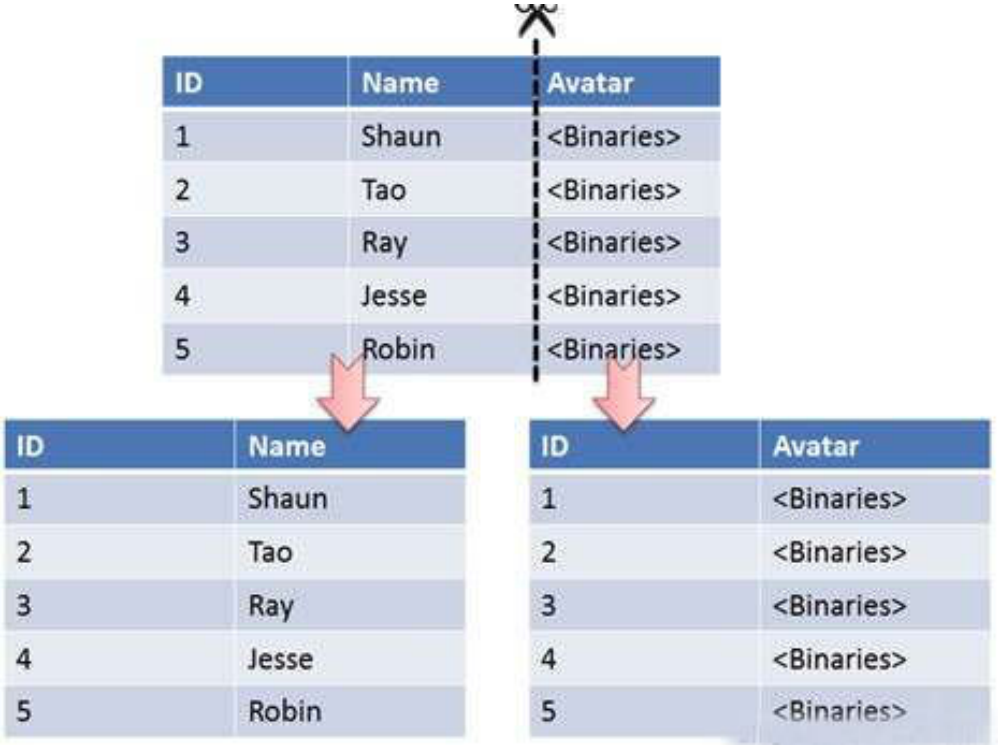
分库分表

简单来说，数据的切分就是通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）中，以达到分散单台设备负载的效果，即分库分表。

数据的切分根据其切分规则的类型，可以分为如下两种切分模式。

- 垂直（纵向）切分：把单一的表拆分成多个表，并分散到不同的数据库（主机）上。
- 水平（横向）切分：根据表中数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上。

1. 垂直切分



垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用垂直切分将按数据库中表的密集程度部署到不同的库中，例如将原来的电商数据库垂直切分成商品数据库 payDB、用户数据库 userBD 等。

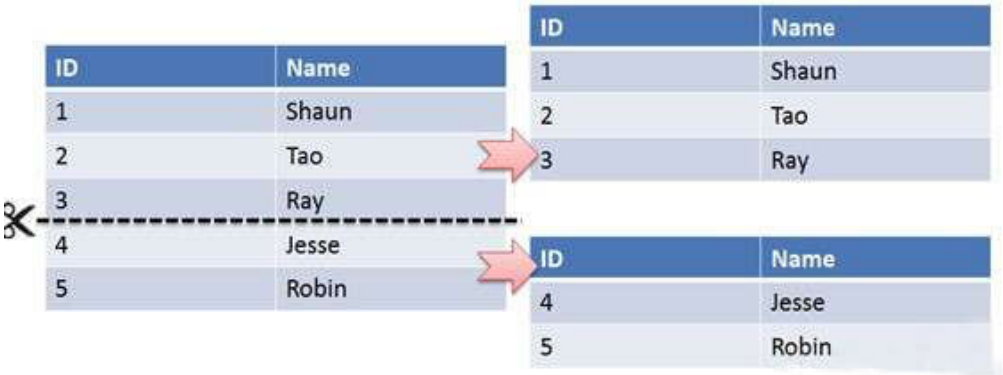
垂直切分的优点:

1. 拆分后业务清晰，拆分规则明确
2. 系统之间进行整合或扩展很容易
3. 按照成本、应用的等级、应用的类型等将表放到不同的机器上，便于管理
4. 便于实现动静分离、冷热分离的数据库表的设计模式
5. 数据维护简单

垂直切分的缺点:

1. 部分业务表无法关联（Join），只能通过接口方式解决，提高了系统的复杂度
2. 受每种业务的不同限制，存在单库性能瓶颈，不易进行数据扩展和提升性能
3. 事务处理复杂

2. 水平切分



水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。

当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

水平切分的优点:

1. 单库单表的数据保持在一定的量级，有助于性能的提高
2. 切分的表的结构相同，应用层改造较少，只需要增加路由规则即可
3. 提高了系统的稳定性和负载能力

水平切分的缺点:

1. 切分后，数据是分散的，很难利用数据库的Join操作，跨库Join性能较差
2. 拆分规则难以抽象
3. 分片事务的一致性难以解决
4. 数据扩容的难度和维护量极大

垂直切分和水平切分的共同点:

1. 存在分布式事务的问题
2. 存在跨节点Join的问题
3. 存在跨节点合并排序、分页的问题
4. 存在多数据源管理的问题

3. Sharding策略和解决方案

Sharding策略为：

- 哈希取模：hash(key) % NUM_DB,可以按照 userId mod 64.将数据分布在64个服务器上。
- 范围：可以是 ID 范围也可以是时间范围，可以根据每台服务器计划存放一个亿的数据,先将数据写入服务器A.一旦服务器A写满,则将数据写入服务器B,以此类推. 这种方式的好处是扩展方便.数据在各个服务器上分布均匀。
- 映射表：使用单独的一个数据库来存储映射关系。

主要的事物问题:

使用分布式事务来解决，比如 XA 接口。

JOIN：

可以将原来的 JOIN 查询分解成多个单表查询，然后在用户程序中进行 JOIN。

ID 唯一性：

- 使用全局唯一 ID：GUID。
- 为每个分片指定一个 ID 范围。
- 分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)。

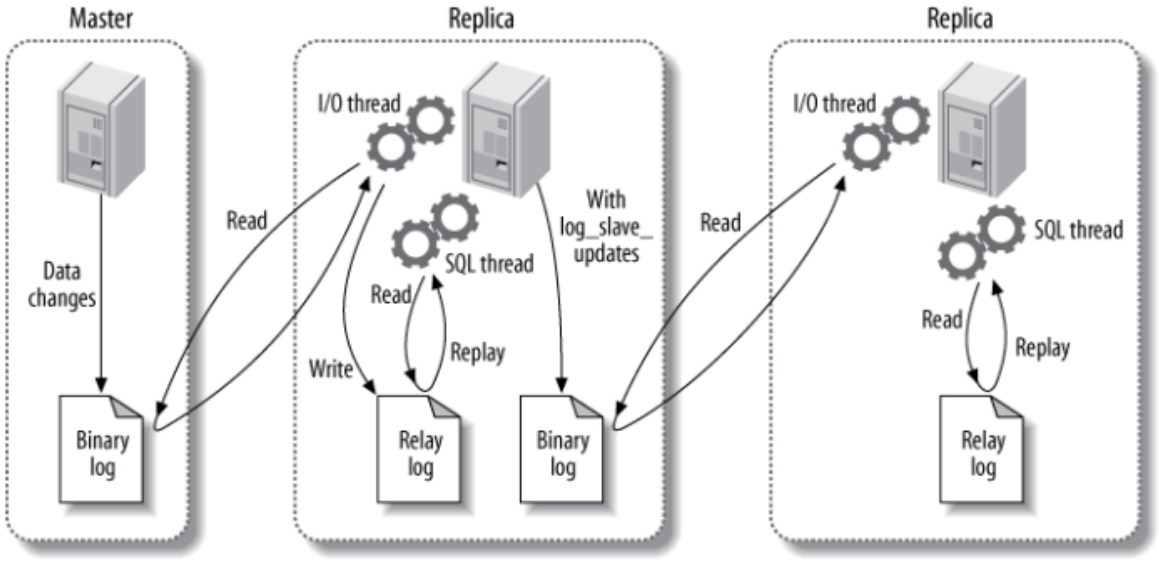
主从复制与读写分离

主从复制:

主从复制主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

- binlog 线程 ：负责将主服务器上的数据更改写入二进制文件（binlog）中。

- I/O 线程 ：负责从主服务器上读取二进制日志文件，并写入从服务器的中继日志中。
- SQL 线程 ：负责读取中继日志并重放其中的 SQL 语句。



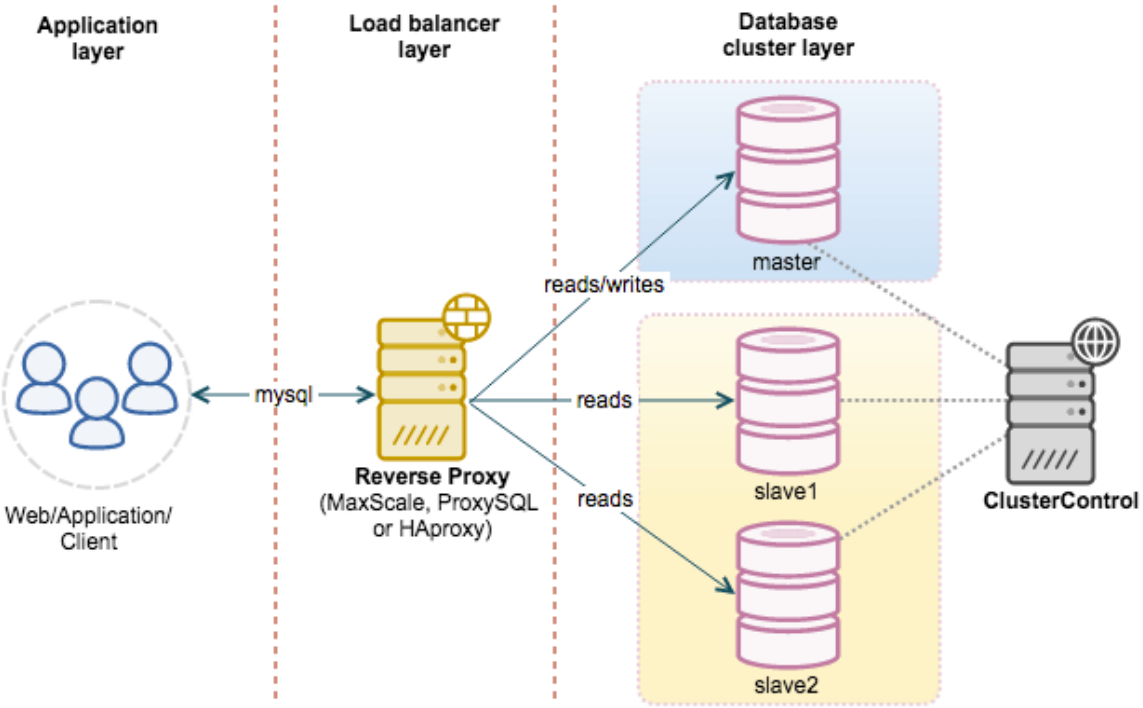
读写分离:

主服务器用来处理写操作以及实时性要求比较高的读操作，而从服务器用来处理读操作。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

MySQL 读写分离能提高性能的原因在于：

- 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
- 从服务器可以配置 MyISAM 引擎，提升查询性能以及节约系统开销；
- 增加冗余，提高可用性。



查询性能优化

1. 使用 Explain 进行分析

Explain 用来分析 SELECT 查询语句，开发人员可以通过分析 Explain 结果来优化查询语句。

字段有：

- select_type：查询类型，有简单查询、联合查询、子查询等
- key：使用的索引
- rows：扫描的行数

```
mysql> explain select * from user_info where id = 3\G
*************************** 1. row ***************************
           id: 3
  select_type: SIMPLE
        table: user_info
   partitions: NULL
         type: const
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 8
          ref: const
         rows: 1
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

2. 优化数据访问

减少请求的数据量:

- 只返回必要的列

最好不要使用 SELECT * 语句。

- 只返回必要的行

使用 WHERE 语句进行查询过滤，有时候也需要使用 LIMIT 语句来限制返回的数据。

- 缓存重复查询的数据

使用缓存可以避免在数据库中进行查询，特别要查询的数据经常被重复查询，缓存可以带来的查询性能提升将会是非常明显的。

- 减少服务器端扫描的行数

对于查询来说，其中最有效的方式是使用索引来覆盖查询。

3. 重构查询方式

- 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

```
DELEFT FROM info WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

```
rows_affected = 0
do {
    rows_affected = do_query(
    "DELETE FROM info WHERE create  < DATE_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000")
} while rows_affected > 0
```

- 分解大连接查询

  将一个大连接查询（JOIN）分解成对每一个表进行一次单表查询，然后将结果在应用程序中进行关联，这样做的好处有：

- 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。

- 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。

- 减少锁竞争；

- 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可扩展。

- 查询本身效率也可能会有所提升。例如下面的例子中，使用 IN() 代替连接查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的连接要更高效。

```
SELECT * FROM vido
JOIN video_post ON video_post.video_id=video.id
JOIN post ON video_post.post_id=post.id
WHERE video.video='mysql';
SELECT * FROM video WHERE video='mysql';
SELECT * FROM video_post WHERE video_id=1234;
SELECT * FROM post WHERE post.id IN (123,456,567,9098,8904);
```

锁类型

MySQL的引擎InnoDB的加锁问题，一直是一个面试中常问的话题。例如，数据库如果有高并发请求，如何保证数据完整性？产生死锁问题如何排查并解决？在工作过程中，也会经常用到，乐观锁，排它锁等。

注意：MySQL 是一个支持插件式存储引擎的数据库系统。接下来我们的分析，都是基于 InnoDB 存储引擎，其他引擎的表现，会有较大的区别。

- 版本查看

  ```
  select version();
  ```

- 存储引擎查看

MySQL 给开发者提供了查询存储引擎的功能,可以通过下面的命令查看：

```
SHOW ENGINES
```

- 乐观锁

通过数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。什么是数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

应用示例:

1. 数据库表设计

```
select id,value,version from TABLE where id=#{id}
```

2. 每次更新表中的value字段时，为了防止发生冲突，需要这样操作

```
update TABLE
set value=2,version=version+1
where id=#{id} and version=#{version};
```

- 悲观锁

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟 Java 中的 synchronized 很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。

说到这里，由悲观锁涉及到的另外两个锁概念就出来了，它们就是共享锁与排它锁。共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。

应用示例:

排它锁:

要使用悲观锁，我们必须关闭 mysql 数据库的自动提交属性，因为 MySQL 默认使用 autocommit 模式，也就是说，当你执行一个更新操作后，MySQL 会立刻将结果进行提交。

我们可以使用命令设置 MySQL 为非 autocommit 模式：

```
set autocommit=0;
# 设置完autocommit后，我们就可以执行我们的正常业务了。具体如下：

# 1. 开始事务（三者选一就可以）
begin; / begin work; / start transaction;

# 2. 查询表信息
select status from TABLE where id=1 for update;

# 3. 插入一条数据
insert into TABLE (id,value) values (2,2);

# 4. 修改数据为
update TABLE set value=2 where id=1;

# 5. 提交事务
commit;/commit work;
```

- 共享锁

共享锁又称读锁（read lock），是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

如果事务 T 对数据 A 加上共享锁后，则其他事务只能对 A 再加共享锁，不能加排他锁。获得共享锁的事务只能读数据，不能修改数据。

```
#三者选一就可以
begin; / begin work; / start transaction;

SELECT * from TABLE where id = 1  lock in share mode;
```

然后在另一个查询窗口中，对 id 为 1 的数据进行更新：

```
update TABLE set name="https://www.youtube.com" where id =1;
```

此时，操作界面进入了卡顿状态，过了超时间，提示错误信息。如果在超时前，执行 commit，此更新语句就会成功。

```
[SQL]update  test_one set name="https://www.youtube.com" where id =1;
[Err] 1205 - Lock wait timeout exceeded; try restarting transaction
```

加上共享锁后，也提示错误信息:

```
update test_one set name="www.souyunku.com" where id =1 lock in share mode;
[SQL]update  test_one set name="https://www.youtube.com" where id =1 lock in share mode;
[Err] 1064 - You have an error in your SQL syntax; check the manual that corresponds to your MySQL server ver
sion for the right syntax to use near 'lock in share mode' at line 1
```

在查询语句后面增加 `lock in share mode，MySQL` 会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

加上共享锁后，对于 `update,insert,delete` 语句会自动加排它锁。

- 排它锁

排他锁(exclusive lock)也叫又称写锁(writer lock)。

排它锁是悲观锁的一种实现，在上面悲观锁也介绍过。

若在一个事务上对数据对象 A 加上 X 锁，该事务可以读 A 也可以修改 A，其他事务不能再对 A 加任何锁，直到该事物释放 A 上的锁。这保证了其他事务在该事物 释放 A 上的锁之前不能再读取和修改 A。排它锁会阻塞所有的排它锁和共享锁。

读取为什么要加读锁呢：防止数据在被读取的时候被别的线程加上写锁

使用方式：在需要执行的语句后面加上 `for update` 就可以了

- 行锁

行锁又分共享锁和排他锁,由字面意思理解，就是给某一行加上锁，也就是一条记录加上锁。

注意：行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。

共享锁：

共享锁又叫做读锁，所有的事务只能对其进行读操作不能写操作，加上共享锁后在事务结束之前其他事务只能再加共享锁，除此之外其他任何类型的锁都不能再加了。

```
#结果集的数据都会加共享锁
SELECT * from TABLE where id = "1"  lock in share mode;
```

排他锁：

若某个事物对某一行加上了排他锁，只能这个事务对其进行读写，在此事务结束之前，其他事务不能对其进行加任何锁，其他进程可以读取，不能进行写操作，需等待其释放。

```
select status from TABLE where id=1 for update;
```

由于对于表中 id 字段为主键，就也相当于索引。执行加锁时，会将 id 这个索引为 1 的记录加上锁，那么这个锁就是行锁。

- 表锁

如何加表锁?

通常在引擎Innodb下的行锁是在有索引的情况下,没有索引的表是锁定全表的。

Innodb中的行锁与表锁：

前在 Innodb引擎中既支持行锁也支持表锁，那么什么时候会锁住整张表，什么时候或只锁住一行呢？ 只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁！

在实际应用中，要特别注意 InnoDB 行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

行级锁都是基于索引的，如果一条 SQL 语句用不到索引是不会使用行级锁的，会使用表级锁。行级锁的缺点是：由于需要请求大量的锁资源，所以速度慢，内存消耗大。

- 死锁

死锁（Deadlock） 所谓死锁：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

解除正在死锁的状态有两种方法：

第一种：

1. 查询是否锁表
   ```
   show OPEN TABLES where In_use > 0;
   ```
2. 查询进程（如果您有SUPER权限，您可以看到所有线程。否则，您只能看到您自己的线程）
   ```
   show processlist
   ```
3. 杀死进程id（就是上面命令的id列）
   ```
   kill id
   ```

第二种：

1. 查看当前的事务
   ```
   SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
   ```

2. 查看当前锁定的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
```

3. 查看当前等锁的事务

```
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
```

杀死进程

```
kill PID
```

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。 产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

虽然不能完全避免死锁，但可以使死锁的数量减至最少。将死锁减至最少可以增加事务的吞吐量并减少系统开销，因为只有很少的事务回滚，而回滚会取消事务执行的所有工作。由于死锁时回滚而由应用程序重新提交。

我们可以通过下面的方法优化最大限度地降低死锁:

- 按同一顺序访问对象
- 避免事务中的用户交互
- 保持事务简短并在一个批处理中
- 使用低隔离级别
- 使用绑定连接

参考资料

- 视频：MySQL 事务的隔离级别与锁

Golang面试问题汇总

Golang面试问题汇总

通常我们去面试肯定会有些不错的Golang的面试题目的，所以总结下，让其他Golang开发者也可以查看到，同时也用来检测自己的能力和提醒自己的不足之处,欢迎大家补充和提交新的面试题目.

Golang面试问题汇总, 这里主要分为 Golang, Mysql , Redis , Network protocol(网络协议), Linux,以及 Algorithm 和 Structrues.

Golang基础

题号	题目
1	Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量
2	无缓冲Chan的发送和接收是否同步
3	Golang并发机制以及它所使用的CSP并发模型
4	Golang中常用的并发模型
5	Go中对nil的Slice和空Slice的处理是一致的吗
6	协程和线程和进程的区别
7	Golang的内存模型中为什么小对象多了会造成GC压力
8	Go中数据竞争问题怎么解决
9	什么是channel，为什么它可以做到线程安全
10	Golang垃圾回收算法
11	GC的触发条件
12	Go的GPM如何调度
13	并发编程概念是什么
14	Go语言的栈空间管理是怎么样的
15	Goroutine和Channel的作用分别是什么
16	怎么查看Goroutine的数量

题号	题目
17	Go中的锁有哪些
18	怎么限制Goroutine的数量
19	Channel是同步的还是异步的
20	Goroutine和线程的区别
21	Go的Struct能不能比较
22	Go的defer原理是什么
23	Go的select可以用于什么
24	Go的Context包的用途是什么
25	Go主协程如何等其余协程完再操作
26	Go的Slice如何扩容
27	Go中的map如何实现顺序读取
28	Go中CAS是怎么回事
29	Go中的逃逸分析是什么
30	Go值接收者和指针接收者的区别
31	Go的对象在内存中是怎样分配的
32	栈的内存是怎么分配的
33	堆内存管理怎么分配的
34	Go中的defer函数使用下面的两种情况下结果是什么
35	在Go函数中为什么会发生内存泄露
36	Go中new和make的区别
37	G0的作用
38	Go中的锁如何实现
39	Go中的channel的实现

题号	题目
40	Go中的map的实现
41	Go中的http包的实现原理
42	Goroutine发生了泄漏如何检测
43	Go函数返回局部变量的指针是否安全
44	Go中两个Nil可能不相等吗
45	Goroutine和KernelThread之间是什么关系

Mysql基础

题号	题目
1	Mysql索引用的是什么算法
2	Mysql事务的基本要素
3	Mysql的存储引擎
4	Mysql事务隔离级别
5	Mysql高可用方案有哪些
6	Mysql中utf8和utf8mb4区别
7	Mysql中乐观锁和悲观锁区别
8	Mysql索引主要是哪些
9	Mysql联合索引最左匹配原则
10	聚簇索引和非聚簇索引区别
11	如何查询一个字段是否命中了索引
12	Mysql中查询数据什么情况下不会命中索引
13	Mysql中的MVCC是什么
14	Mvcc和Redolog和Undolog以及Binlog有什么不同

题号	题目
15	Mysql读写分离以及主从同步
16	InnoDB的关键特性
17	Mysql如何保证一致性和持久性
18	为什么选择B+树作为索引结构
19	InnoDB的行锁模式
20	哈希(hash)比树(tree)更快，索引结构为什么要设计成树型
21	为什么索引的key长度不能太长
22	Mysql的数据如何恢复到任意时间点
23	Mysql为什么加了索引可以加快查询
24	Explain命令有什么用

Redis基础

题号	题目
1	Redis的数据结构及使用场景
2	Redis持久化的几种方式
3	Redis的LRU具体实现
4	单线程的Redis为什么快
5	Redis的数据过期策略
6	如何解决Redis缓存雪崩问题
7	如何解决Redis缓存穿透问题
8	Redis并发竞争key如何解决
9	Redis的主从模式和哨兵模式和集群模式区别
10	Redis事物的了解CheckAndSet操作实现乐观锁

题号	题目
11	Redis有序集合zset底层怎么实现的
12	跳表的查询过程是怎么样的，查询和插入的时间复杂度

网络协议基础

题号	题目
1	TCP和UDP有什么区别
2	TCP中三次握手和四次挥手
3	TCP的LISTEN状态是什么
4	常见的HTTP状态码有哪些
5	301和302有什么区别
6	504和500有什么区别
7	HTTPS和HTTP有什么区别
8	Quic有什么优点相比Http2
9	Grpc的优缺点
10	Get和Post区别
11	Unicode和ASCII以及Utf8的区别
12	Cookie与Session异同
13	Client如何实现长连接
14	Http1和Http2和Grpc之间的区别是什么
15	Tcp中的拆包和粘包是怎么回事
16	TFO的原理是什么
17	TIME_WAIT的作用
18	网络的性能指标有哪些

Linux基础

题号	题目
1	异步和非阻塞的区别
2	虚拟内存作用是什么
3	Linux查看端口占用和cpu负载
4	Linux如何发送信号给一个进程
5	如何避免死锁
6	孤儿进程和僵尸进程区别
7	滑动窗口的概念以及应用
8	Epoll和Select的区别
9	进程之间为什么要进行通信呢
10	输入PingIP后敲回车,发包前会发生什么
11	进程和进程间的通信方式区别和不同

Algorithm和Structrues

题号	题目
1	哪些排序算法是稳定的
2	给定一个二叉树,判断其是否是一个有效的二叉搜索树
3	排序算法
4	如何通过递归反转单链表
5	链表和数组相比有什么优缺点
6	通常一般会用到哪些数据结构

其他基础

题号	题目

题号	题目
1	中间件原理
2	Hash冲突有什么解决办法
3	微服务架构是什么样子的
4	分布式锁实现
5	负载均衡原理是什么
6	互斥锁和读写锁和死锁问题是怎么解决
7	Etcd中的Raft一致性算法原理
8	Git的merge跟rebase的区别
9	如何对一个20GB的文件进行排序
10	LVS原理是什么
11	为什么需要消息队列
12	高并发系统的设计与实现
13	Kafka的文件存储机制
14	Kafka如何保证可靠性
15	Kafka是如何实现高吞吐率的

Golang基础模块信息

1. **Golang中除了加Mutex锁以外还有哪些方式安全读写共享变量**

Golang中Goroutine 可以通过 Channel 进行安全读写共享变量,还可以通过原子性操作进行.

2. **无缓冲Chan的发送和接收是否同步**

```
ch := make(chan int)    无缓冲的channel由于没有缓冲发送和接收需要同步.
ch := make(chan int, 2) 有缓冲channel不要求发送和接收操作同步.
```

- channel无缓冲时，发送阻塞直到数据被接收，接收阻塞直到读到数据。
- channel有缓冲时，当缓冲满时发送阻塞，当缓冲空时接收阻塞。

3. **Golang并发机制以及它所使用的CSP并发模型.**

在计算机科学中，通信顺序过程（communicating sequential processes，CSP）是一种描述并发系统中交互模式的正式语言，它是并发数学理论家族中的一个成员，被称为过程算法（process algebras），或者说过程计算（process calculate），是基于消息的通道传递的数学理论。

CSP模型是上个世纪七十年代提出的,不同于传统的多线程通过共享内存来通信，CSP讲究的是“以通信的方式来共享内存”。用于描述两个独立的并发实体通过共享的通讯 channel(管道)进行通信的并发模型。 CSP中channel是第一类对象，它不关注发送消息的实体，而关注与发送消息时使用的channel。

Golang中channel 是被单独创建并且可以在进程之间传递，它的通信模式类似于 boss-worker 模式的，一个实体通过将消息发送到channel 中，然后又监听这个 channel 的实体处理，两个实体之间是匿名的，这个就实现实体中间的解耦，其中 channel 是同步的一个消息被发送到 channel 中，最终是一定要被另外的实体消费掉的，在实现原理上其实类似一个阻塞的消息队列。

Goroutine 是Golang实际并发执行的实体，它底层是使用协程(coroutine)实现并发，coroutine是一种运行在用户态的用户线程，类似于 greenthread，go底层选择使用coroutine的出发点是因为，它具有以下特点：

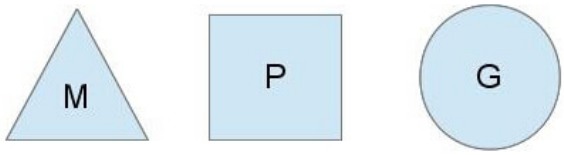
- 用户空间 避免了内核态和用户态的切换导致的成本。
- 可以由语言和框架层进行调度。
- 更小的栈空间允许创建大量的实例。

Golang中的Goroutine的特性:

Golang内部有三个对象： P对象(processor) 代表上下文（或者可以认为是cpu），M(work thread)代表工作线程，G对象（goroutine）.

正常情况下一个cpu对象启一个工作线程对象，线程去检查并执行goroutine对象。碰到goroutine对象阻塞的时候，会启动一个新的工作线程，以充分利用cpu资源。
所有有时候线程对象会比处理器对象多很多.

我们用如下图分别表示P、M、G:



G（Goroutine）：我们所说的协程，为用户级的轻量级线程，每个Goroutine对象中的sched保存着其上下文信息.

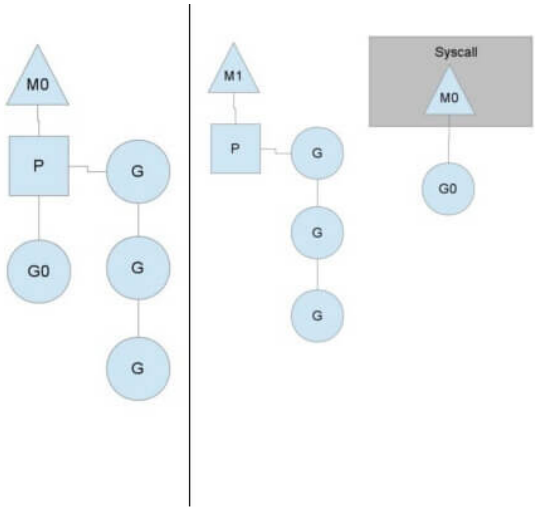
M（Machine）：对Os内核级线程的封装，数量对应真实的CPU数（真正干活的对象）.

P（Processor）：逻辑处理器,即为G和M的调度对象，用来调度G和M之间的关联关系，其数量可通过GOMAXPROCS()来设置，默认为核心数.

在单核情况下，所有Goroutine运行在同一个线程（M0）中，每一个线程维护一个上下文（P），任何时刻，一个上下文中只有一个Goroutine，其他Goroutine在runqueue中等待。

一个Goroutine运行完自己的时间片后，让出上下文，自己回到runqueue中（如下图所示）。

当正在运行的G0阻塞的时候（可以需要IO），会再创建一个线程（M1），P转到新的线程中去运行。



当M0返回时，它会尝试从其他线程中“偷”一个上下文过来，如果没有偷到，会把Goroutine放到Global runqueue中去，然后把自己放入线程缓存中。
上下文会定时检查Global runqueue。

Golang是为并发而生的语言，Go语言是为数不多的在语言层面实现并发的语言；也正是Go语言的并发特性，吸引了全球无数的开发者。

Golang的CSP并发模型，是通过Goroutine和Channel来实现的。

Goroutine 是Go语言中并发的执行单位。有点抽象，其实就是和传统概念上的”线程“类似，可以理解为”线程“。
Channel是Go语言中各个并发结构体(Goroutine)之前的通信机制。通常Channel，是各个Goroutine之间通信的”管道“，有点类似于Linux中的管道。

通信机制channel也很方便，传数据用 `channel <- data` ，取数据用 `<-channel` 。

在通信过程中，传数据 `channel <- data` 和取数据 `<-channel` 必然会成对出现，因为这边传，那边取，两个goroutine之间才会实现通信。

而且不管是传还是取，肯定阻塞，直到另外的goroutine传或者取为止。

因此GPM的简要概括即为: 事件循环,线程池,工作队列.

4. Golang中常用的并发模型

Golang 中常用的并发模型有三种:

- 通过channel通知实现并发控制

无缓冲的通道指的是通道的大小为0，也就是说，这种类型的通道在接收前没有能力保存任何值，它要求发送 goroutine 和接收 goroutine 同时准备好，才可以完成发送和接收操作。

从上面无缓冲的通道定义来看，发送 goroutine 和接收 gouroutine 必须是同步的，同时准备后，如果没有同时准备好的话，先执行的操作就会阻塞等待，直到另一个相对应的操作准备好为止。这种无缓冲的通道我们也称之为同步通道。

```
func main() {
    ch := make(chan struct{})
    go func() {
        fmt.Println("start working")
        time.Sleep(time.Second * 1)
        ch <- struct{}{}
    }()
```

```
    <-ch

    fmt.Println("finished")
}
```

当主 goroutine 运行到 `<-ch` 接受 channel 的值的时候，如果该 channel 中没有数据，就会一直阻塞等待，直到有值。 这样就可以简单实现并发控制

- 通过sync包中的WaitGroup实现并发控制

Goroutine是异步执行的，有的时候为了防止在结束main函数的时候结束掉Goroutine，所以需要同步等待，这个时候就需要用 WaitGroup了，在 sync 包中，提供了 WaitGroup ，它会等待它收集的所有 goroutine 任务全部完成。在WaitGroup里主要有三个方法:

- Add, 可以添加或减少 goroutine的数量.
- Done, 相当于Add(-1).
- Wait, 执行后会堵塞主线程，直到WaitGroup 里的值减至0.

在主 goroutine 中 Add(delta int) 索要等待goroutine 的数量。在每一个 goroutine 完成后 Done() 表示这一个goroutine 已经完成，当所有的 goroutine 都完成后，在主 goroutine 中 WaitGroup 返回。

```
func main(){
    var wg sync.WaitGroup
    var urls = []string{
        "http://www.golang.org/",
        "http://www.google.com/",
    }
    for _, url := range urls {
        wg.Add(1)
        go func(url string) {
            defer wg.Done()
            http.Get(url)
        }(url)
    }
    wg.Wait()
}
```

在Golang官网中对于WaitGroup介绍是 `A WaitGroup must not be copied after first use` ,在 WaitGroup 第一次使用后，不能被拷贝

应用示例:

```
func main(){
 wg := sync.WaitGroup{}
    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(wg sync.WaitGroup, i int) {
            fmt.Printf("i:%d", i)
            wg.Done()
        }(wg, i)
    }
    wg.Wait()
    fmt.Println("exit")
}
```

运行:

```
i:1i:3i:2i:0i:4fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc000094018)
        /home/keke/soft/go/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc000094010)
        /home/keke/soft/go/src/sync/waitgroup.go:130 +0x64
main.main()
        /home/keke/go/Test/wait.go:17 +0xab
exit status 2
```

它提示所有的 `goroutine` 都已经睡眠了，出现了死锁。这是因为 wg 给拷贝传递到了 goroutine 中，导致只有 Add 操作，其实 Done操作是在 wg 的副本执行的。

因此 Wait 就会死锁。

这个第一个修改方式:将匿名函数中 wg 的传入类型改为 `*sync.WaitGroup` ,这样就能引用到正确的 `WaitGroup` 了。

这个第二个修改方式:将匿名函数中的 wg 的传入参数去掉，因为Go支持闭包类型，在匿名函数中可以直接使用外面的 wg 变量.

- 在Go 1.7 以后引进的强大的Context上下文，实现并发控制.

通常,在一些简单场景下使用 channel 和 WaitGroup 已经足够了，但是当面临一些复杂多变的网络并发场景下 channel 和 WaitGroup 显得有些力不从心了。
比如一个网络请求 Request，每个 Request 都需要开启一个 goroutine 做一些事情，这些 goroutine 又可能会开启其他的 goroutine，比如数据库和RPC服务。

所以我们需要一种可以跟踪 goroutine 的方案，才可以达到控制他们的目的，这就是Go语言为我们提供的 Context，称之为上下文非常贴切，它就是goroutine 的上下文。

它是包括一个程序的运行环境、现场和快照等。每个程序要运行时，都需要知道当前程序的运行状态，通常Go 将这些封装在一个 Context 里，再将它传给要执行的 goroutine 。

context 包主要是用来处理多个 goroutine 之间共享数据，及多个 goroutine 的管理。

context 包的核心是 struct Context，接口声明如下：

```
// A Context carries a deadline, cancelation signal, and request-scoped values
// across API boundaries. Its methods are safe for simultaneous use by multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this `Context` is canceled
    // or times out.
    // Done() 返回一个只能接受数据的channel类型，当该context关闭或者超时时间到了的时候，该channel就会有一个取消信号
    Done() <-chan struct{}

    // Err indicates why this Context was canceled, after the Done channel
    // is closed.
    // Err() 在Done() 之后，返回context 取消的原因。
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
    // Deadline() 设置该context cancel的时间点
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    // Value() 方法允许 Context 对象携带request作用域的数据，该数据必须是线程安全的。
```

```
    Value(key interface{}) interface{}
}
```

Context 对象是线程安全的，你可以把一个 Context 对象传递给任意个数的 gorotuine，对它执行 取消 操作时，所有 goroutine 都会接收到取消信号。

一个 Context 不能拥有 Cancel 方法，同时我们也只能 Done channel 接收数据。其中的原因是一致的：接收取消信号的函数和发送信号的函数通常不是一个。

典型的场景是：父操作为子操作操作启动 goroutine，子操作也就不能取消父操作。

5. Go中对nil的Slice和空Slice的处理是一致的吗

首先Go的JSON 标准库对 `nil slice` 和 空 `slice` 的处理是不一致.

通常错误的用法，会报数组越界的错误，因为只是声明了slice，却没有给实例化的对象。

```
var slice []int
slice[1] = 0
```

此时slice的值是nil，这种情况可以用于需要返回slice的函数，当函数出现异常的时候，保证函数依然会有nil的返回值。

empty slice 是指slice不为nil，但是slice没有值，slice的底层的空间是空的，此时的定义如下：

```
slice := make([]int,0)
slice := []int{}
```

当我们查询或者处理一个空的列表的时候，这非常有用，它会告诉我们返回的是一个列表，但是列表内没有任何值。

总之， `nil slice` 和 `empty slice` 是不同的东西,需要我们加以区分的.

6. 协程和线程和进程的区别

- 进程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。
每个进程都有自己的独立内存空间，不同进程通过进程间通信来通信。由于进程比较重量，占据独立的内存，所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。

- 线程

线程是进程的一个实体,线程是内核态,而且是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。
线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。

- 协程

协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。
协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

7. Golang的内存模型中为什么小对象多了会造成GC压力

通常小对象过多会导致GC三色法消耗过多的GPU。优化思路是，减少对象分配.

8. Go中数据竞争问题怎么解决

Data Race问题可以使用互斥锁sync.Mutex, 或者也可以通过CAS无锁并发解决.

其中使用同步访问共享数据或者CAS无锁并发是处理数据竞争的一种有效的方法.

golang在1.1之后引入了竞争检测机制，可以使用 `go run -race` 或者 `go build -race` 来进行静态检测。

其在内部的实现是,开启多个协程执行同一个命令， 并且记录下每个变量的状态.

竞争检测器基于C/C++的 `ThreadSanitizer` 运行时库，该库在Google内部代码基地和Chromium找到许多错误。这个技术在2012年九月集成到Go中，从那时开始，它已经在标准库中检测到42个竞争条件。现在，它已经是我们持续构建过程的一部分，当竞争条件出现时，它会继续捕捉到这些错误。

竞争检测器已经完全集成到Go工具链中，仅仅添加-race标志到命令行就使用了检测器。

```
$ go test -race mypkg    // 测试包
$ go run -race mysrc.go  // 编译和运行程序
$ go build -race mycmd   // 构建程序
$ go install -race mypkg // 安装程序
```

要想解决数据竞争的问题可以使用互斥锁 `sync.Mutex` ,解决数据竞争(Data race),也可以使用管道解决,使用管道的效率要比互斥锁高.

9. 什么是channel，为什么它可以做到线程安全

Channel是Go中的一个核心类型，可以把它看成一个管道，通过它并发核心单元就可以发送或者接收数据进行通讯(communication),Channel也可以理解是一个先进先出的队列，通过管道进行通信。

Golang的Channel,发送一个数据到Channel 和 从Channel接收一个数据 都是 原子性的。

Go的设计思想就是, 不要通过共享内存来通信，而是通过通信来共享内存，前者就是传统的加锁，后者就是Channel。

也就是说，设计Channel的主要目的就是在多任务间传递数据的，本身就是安全的。

10. Golang垃圾回收算法

首先我们先来了解下垃圾回收.

什么是垃圾回收?

内存管理是程序员开发应用的一大难题。传统的系统级编程语言（主要指C/C++）中，程序开发者必须对内存小心的进行管理操作，控制内存的申请及释放。
因为稍有不慎，就可能产生内存泄露问题，这种问题不易发现并且难以定位，一直成为困扰程序开发者的噩梦。

如何解决这个头疼的问题呢?

过去一般采用两种办法：

- 内存泄露检测工具。这种工具的原理一般是静态代码扫描，通过扫描程序检测可能出现内存泄露的代码段。然而检测工具难免有疏漏和不足，只能起到辅助作用。

- 智能指针。这是 c++ 中引入的自动内存管理方法，通过拥有自动内存管理功能的指针对象来引用对象，是程序员不用太关注内存的释放，而达到内存自动释放的目的。这种方法是采用最广泛的做法，但是对程序开发者有一定的学习成本（并非语言层面的原生支持），而且一旦有忘记使用的场景依然无法避免内存泄露。

为了解决这个问题，后来开发出来的几乎所有新语言（java，python，php等等）都引入了语言层面的自动内存管理 – 也就是语言的使用者只用关注内存的申请而不必关心内存的释放，内存释放由虚拟机（virtual machine）或运行时（runtime）来自动进行管理。而这种对不再使用的内存资源进行自动回收的行为就被称为垃圾回收。

常用的垃圾回收的方法:

- 引用计数（reference counting）

这是最简单的一种垃圾回收算法，和之前提到的智能指针异曲同工。对每个对象维护一个引用计数，当引用该对象的对象被销毁或更新时被引用对象的引用计数自动减一，当被引用对象被创建或被赋值给其他对象时引用计数自动加一。当引用计数为0时则立即回收对象。

这种方法的优点是实现简单，并且内存的回收很及时。这种算法在内存比较紧张和实时性比较高的系统中使用的比较广泛，如ios cocoa框架，php，python等。

但是简单引用计数算法也有明显的缺点：

1. 频繁更新引用计数降低了性能。

一种简单的解决方法就是编译器将相邻的引用计数更新操作合并到一次更新；还有一种方法是针对频繁发生的临时变量引用不进行计数，而是在引用达到0时通过扫描堆栈确认是否还有临时对象引用而决定是否释放。等等还有很多其他方法，具体可以参考这里。

2. 循环引用。

当对象间发生循环引用时引用链中的对象都无法得到释放。最明显的解决办法是避免产生循环引用，如cocoa引入了strong指针和weak指针两种指针类型。或者系统检测循环引用并主动打破循环链。当然这也增加了垃圾回收的复杂度。

- 标记-清除（mark and sweep）

标记-清除（mark and sweep）分为两步，标记从根变量开始迭代得遍历所有被引用的对象，对能够通过应用遍历访问到的对象都进行标记为“被引用”；标记完成后进行清除操作，对没有标记过的内存进行回收（回收同时可能伴有碎片整理操作）。
这种方法解决了引用计数的不足，但是也有比较明显的问题：每次启动垃圾回收都会暂停当前所有的正常代码执行，回收时,系统响应能力大大降低 ！当然后续也出现了很多mark&sweep算法的变种（如三色标记法）优化了这个问题。

- 分代搜集（generation）

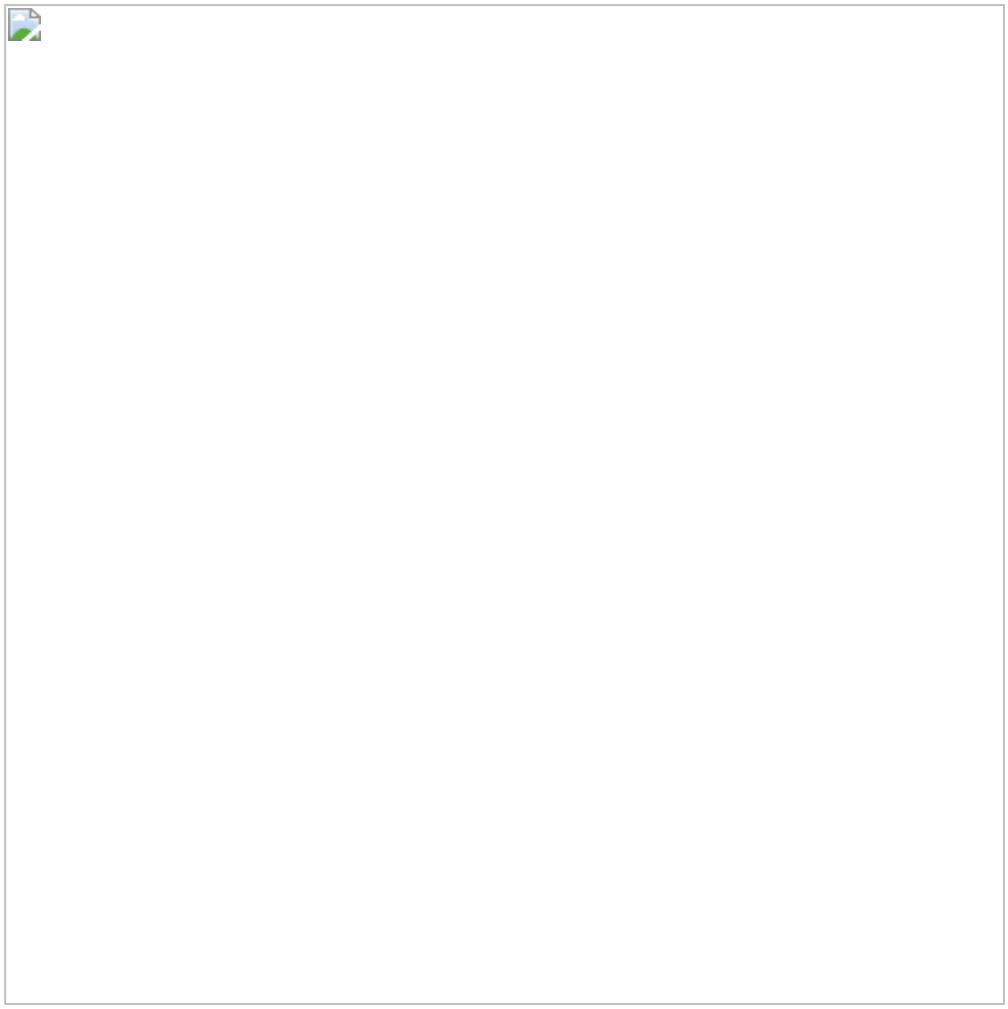
java的jvm 就使用的分代回收的思路。在面向对象编程语言中，绝大多数对象的生命周期都非常短。分代收集的基本思想是，将堆划分为两个或多个称为代（generation）的空间。
新创建的对象存放在称为新生代（young generation）中（一般来说，新生代的大小会比 老年代小很多），随着垃圾回收的重复执行，生命周期较长的对象会被提升（promotion）到老年代中（这里用到了一个分类的思路，这个是也是科学思考的一个基本思路）。

因此，新生代垃圾回收和老年代垃圾回收两种不同的垃圾回收方式应运而生，分别用于对各自空间中的对象执行垃圾回收。新生代垃圾回收的速度非常快，比老年代快几个数量级，即使新生代垃圾回收的频率更高，执行效率也仍然比老年代垃圾回收强，这是因为大多数对象的生命周期都很短，根本无需提升到老年代。

Golang GC 时会发生什么?

Golang 1.5后，采取的是“非分代的、非移动的、并发的、三色的”标记清除垃圾回收算法。

golang 中的 gc 基本上是标记清除的过程：



golang 的垃圾回收是基于标记清扫算法，这种算法需要进行 STW（stop the world)，这个过程就会导致程序是卡顿的，频繁的 GC 会严重影响程序性能.

golang 在此基础上进行了改进，通过三色标记清扫法与写屏障来减少 STW 的时间.

gc的过程一共分为四个阶段：

1. 栈扫描（开始时STW） 所有对象最开始都是白色.
2. 从 root 开始找到所有可达对象，标记为灰色，放入待处理队列。
3. 遍历灰色对象队列，将其引用对象标记为灰色放入待处理队列，自身标记为黑色。
4. 清除（并发） 循环步骤3直到灰色队列为空为止，此时所有引用对象都被标记为黑色，所有不可达的对象依然为白色，白色的就是需要进行回收的对象。
   三色标记法相对于普通标记清扫，减少了 STW 时间. 这主要得益于标记过程是 “on-the-fly” 的，在标记过程中是不需要 STW 的，它与程序是并发执行的，这就大大缩短了STW的时间.

Golang gc 优化的核心就是尽量使得 STW(Stop The World) 的时间越来越短。

详细的Golang的GC介绍可以参看Golang垃圾回收.

写屏障:

当标记和程序是并发执行的，这就会造成一个问题. 在标记过程中，有新的引用产生，可能会导致误清扫.

清扫开始前，标记为黑色的对象引用了一个新申请的对象，它肯定是白色的，而黑色对象不会被再次扫描，那么这个白色对象无法被扫描变成灰色、黑色，它就会最终被清扫，而实际它不应该被清扫.

这就需要用到屏障技术，golang采用了写屏障，其作用就是为了避免这类误清扫问题. 写屏障即在内存写操作前，维护一个约束，从而确保清扫开始前，黑色的对象不能引用白色对象.

11. GC的触发条件

Go中对 GC 的触发时机存在两种形式：

- 主动触发(手动触发)，通过调用 `runtime.GC` 来触发 `GC` ，此调用阻塞式地等待当前 `GC` 运行完毕.
- 被动触发，分为两种方式：

  a. 使用系统监控，当超过两分钟没有产生任何 `GC` 时，强制触发 `GC` .

  b. 使用步调（Pacing）算法，其核心思想是控制内存增长的比例,当前内存分配达到一定比例则触发.

12. Go的GPM如何调度

Goroutine协程:

协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈。
因此，协程能保留上一次调用时的状态（即所有局部状态的一个特定组合），每次过程重入时，就相当于进入上一次调用的状态，换种说法：进入上一次离开时所处逻辑流的位置。

线程和进程的操作是由程序触发系统接口，最后的执行者是系统；协程的操作执行者则是用户自身程序，goroutine也是协程。

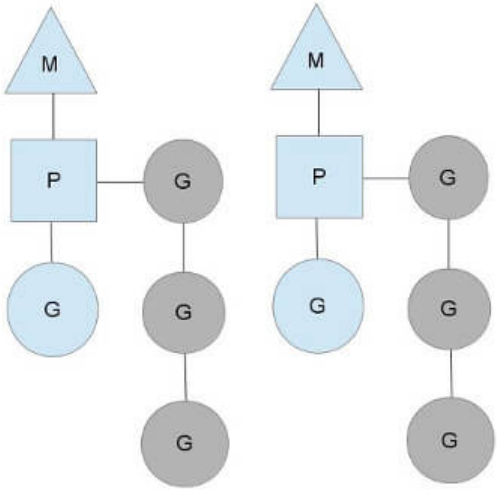
groutine能拥有强大的并发实现是通过GPM调度模型实现.

Go的调度器内部有四个重要的结构：M，P，S，Sched，如上图所示（Sched未给出）.

- M: M代表内核级线程，一个M就是一个线程，goroutine就是跑在M之上的；M是一个很大的结构，里面维护小对象内存cache（mcache）、当前执行的goroutine、随机数发生器等等非常多的信息.
- G: 代表一个goroutine，它有自己的栈，instruction pointer和其他信息（正在等待的channel等等），用于调度.
- P: P全称是Processor，逻辑处理器，它的主要用途就是用来执行goroutine的，所以它也维护了一个goroutine队列，里面存储了所有需要它来执行的goroutine.
- Sched：代表调度器，它维护有存储M和G的队列以及调度器的一些状态信息等.

Go中的GPM调度:

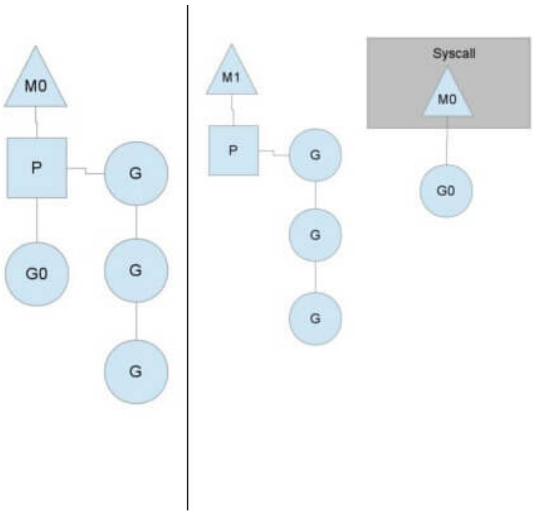
新创建的Goroutine会先存放在Global全局队列中，等待Go调度器进行调度，随后Goroutine被分配给其中的一个逻辑处理器P，并放到这个逻辑处理器对应的Local本地运行队列中，最终等待被逻辑处理器P执行即可。
在M与P绑定后，M会不断从P的Local队列中无锁地取出G，并切换到G的堆栈执行，当P的Local队列中没有G时，再从Global队列中获取一个G，当Global队列中也没有待运行的G时，则尝试从其它的P窃取部分G来执行相当于P之间的负载均衡。



从上图中可以看到，有2个物理线程M，每一个M都拥有一个处理器P，每一个也都有一个正在运行的goroutine。P的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个goroutine可以同时运行。

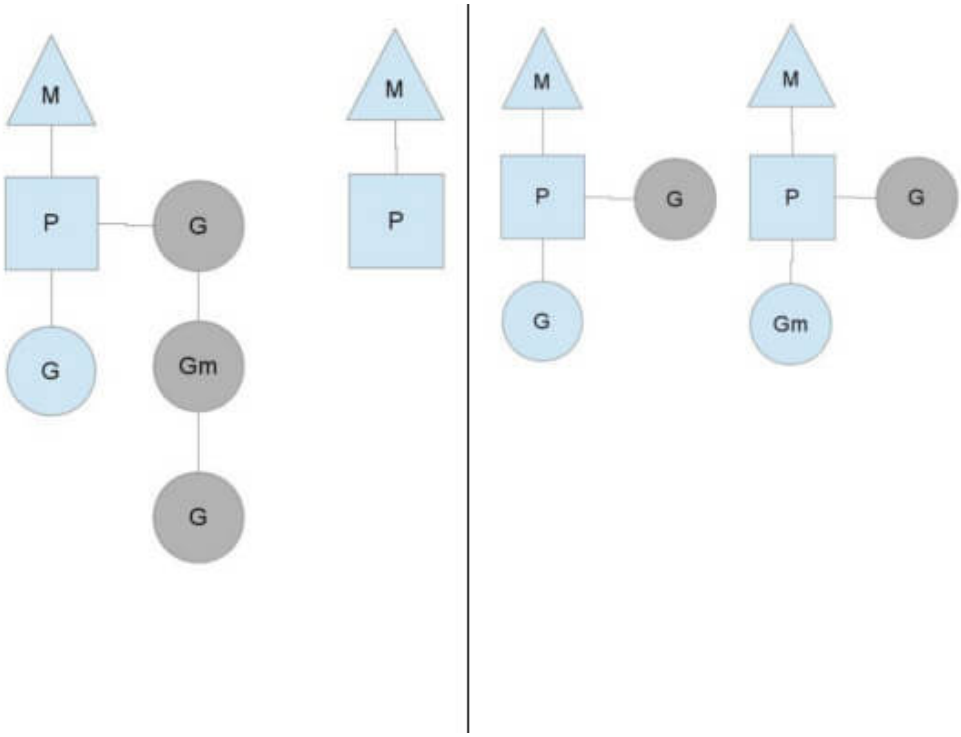
图中灰色的那些goroutine并没有运行，而是出于ready的就绪态，正在等待被调度。P维护着这个队列（称之为runqueue），Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出（如何决定取哪个goroutine？）一个goroutine执行。

当一个OS线程M0陷入阻塞时，P转而在运行M1，图中的M1可能是正被创建，或者从线程缓存中取出。

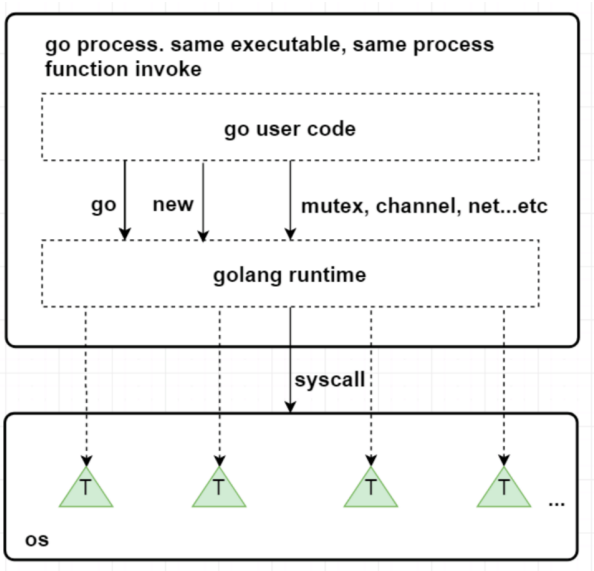


当MO返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS线程那里拿一个P过来，如果没有拿到的话，它就把goroutine放在一个 `global runqueue` 里，然后自己睡眠（放入线程缓存里）。所有的P也会周期性的检查 `global runqueue` 并运行其中的goroutine，否则 `global runqueue` 上的goroutine永远无法执行。

另一种情况是P所分配的任务G很快就执行完了（分配不均），这就导致了这个处理器P处于空闲的状态，但是此时其他的P还有任务，此时如果global runqueue没有任务G了，那么这个P就会从其他的P里偷取一些G来执行。



通常来说，如果P从其他的P那里要拿任务的话，一般就拿 `run queue` 的一半，这就确保了每个OS线程都能充分的使用。



13. 并发编程概念是什么

并行是指两个或者多个事件在同一时刻发生；并发是指两个或多个事件在同一时间间隔发生。

并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的。而并行是真正意义上的“同时执行”。

并发编程是指在一台处理器上“同时”处理多个任务。并发是在同一实体上的多个事件。多个事件在同一时间间隔发生。并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

14. Go语言的栈空间管理是怎么样的

Go语言的运行环境（runtime）会在goroutine需要的时候动态地分配栈空间，而不是给每个goroutine分配固定大小的内存空间。这样就避免了需要程序员来决定栈的大小。

分块式的栈是最初Go语言组织栈的方式。当创建一个goroutine的时候，它会分配一个8KB的内存空间来给goroutine的栈使用。我们可能会考虑当这8KB的栈空间被用完的时候该怎么办?

为了处理这种情况，每个Go函数的开头都有一小段检测代码。这段代码会检查我们是否已经用完了分配的栈空间。如果是的话，它会调用 `morestack` 函数。 `morestack` 函数分配一块新的内存作为栈空间，并且在这块栈空间的底部填入各种信息（包括之前的那块栈地址）。在分配了这块新的栈空间之后，它会重试刚才造成栈空间不足的函数。这个过程叫做栈分裂（stack split）。

在新分配的栈底部，还插入了一个叫做 `lessstack` 的函数指针。这个函数还没有被调用。这样设置是为了从刚才造成栈空间不足的那个函数返回时做准备的。当我们从那个函数返回时，它会跳转到 `lessstack` 。 `lessstack` 函数会查看在栈底部存放的数据结构里的信息，然后调整栈指针（stack pointer）。这样就完成了从新的栈块到老的栈块的跳转。接下来，新分配的这个块栈空间就可以被释放掉了。

`分块式的栈` 让我们能够按照需求来扩展和收缩栈的大小。 Go开发者不需要花精力去估计goroutine会用到多大的栈。创建一个新的goroutine的开销也不大。当 Go开发者不知道栈会扩展到多少大时，它也能很好的处理这种情况。

这一直是之前Go语言管理栈的的方法。但这个方法有一个问题。缩减栈空间是一个开销相对较大的操作。如果在一个循环里有栈分裂，那么它的开销就变得不可忽略了。一个函数会扩展，然后分裂栈。当它返回的时候又会释放之前分配的内存块。如果这些都发生在一个循环里的话，代价是相当大的。
这就是所谓的热分裂问题（hot split problem）。它是Go语言开发者选择新的栈管理方法的主要原因。新的方法叫做 `栈复制法（stack copying）` 。

栈复制法一开始和分块式的栈很像。当goroutine运行并用完栈空间的时候，与之前的方法一样，栈溢出检查会被触发。但是，不像之前的方法那样分配一个新的内存块并链接到老的栈内存块，新的方法会分配一个两倍大的内存块并把老的内存块内容复制到新的内存块里。这样做意味着当栈缩减回之前大小时，我们不需要做任何事情。栈的缩减没有任何代价。而且，当栈再次扩展时，运行环境也不需要再做任何事。它可以重用之前分配的空间。

栈的复制听起来很容易，但实际操作并非那么简单。存储在栈上的变量的地址可能已经被使用到。也就是说程序使用到了一些指向栈的指针。当移动栈的时候，所有指向栈里内容的指针都会变得无效。然而，指向栈内容的指针自身也必定是保存在栈上的。这是为了保证内存安全的必要条件。否则一个程序就有可能访问一段已经无效的栈空间了。

因为垃圾回收的需要，我们必须知道栈的哪些部分是被用作指针了。当我们移动栈的时候，我们可以更新栈里的指针让它们指向新的地址。所有相关的指针都会被更新。我们使用了垃圾回收的信息来复制栈，但并不是任何使用栈的函数都有这些信息。因为很大一部分运行环境是用C语言写的，很多被调用的运行环境里的函数并没有指针的信息，所以也就不能够被复制了。当遇到这种情况时，我们只能退回到分块式的栈并支付相应的开销。

这也是为什么现在运行环境的开发者正在用Go语言重写运行环境的大部分代码。无法用Go语言重写的部分（比如调度器的核心代码和垃圾回收器）会在特殊的栈上运行。这个特殊栈的大小由运行环境的开发者设置。

这些改变除了使栈复制成为可能，它也允许我们在将来实现并行垃圾回收。

另外一种不同的栈处理方式就是在虚拟内存中分配大内存段。由于物理内存只是在真正使用时才会被分配，因此看起来好似你可以分配一个大内存段并让操 作系统处理它。下面是这种方法的一些问题

首先，32位系统只能支持4G字节虚拟内存，并且应用只能用到其中的3G空间。由于同时运行百万goroutines的情况并不少见，因此你很可 能用光虚拟内存，即便我们假设每个goroutine的stack只有8K。

第二，然而我们可以在64位系统中分配大内存，它依赖于过量内存使用。所谓过量使用是指当你分配的内存大小超出物理内存大小时，依赖操作系统保证 在需要时能够分配出物理内存。然而，允许过量使用可能会导致一些风险。由于一些进程分配了超出机器物理内存大小的内存，如果这些进程使用更多内存 时，操作系统将不得不为它们补充分配内存。这会导致操作系统将一些内存段放入磁盘缓存，这常常会增加不可预测的处理延迟。正是考虑到这个原因，一 些新系统关闭了对过量使用的支持。

15. Goroutine和Channel的作用分别是什么

进程是内存资源管理和cpu调度的执行单元。为了有效利用多核处理器的优势，将进程进一步细分，允许一个进程里存在多个线程，这多个线程还是共享同一片内存空间，但cpu调度的最小单元变成了线程。

那协程又是什么呢，以及与线程的差异性?

协程，可以看作是轻量级的线程。但与线程不同的是，线程的切换是由操作系统控制的，而协程的切换则是由用户控制的。

最早支持协程的程序语言应该是lisp方言scheme里的continuation（续延），续延允许scheme保存任意函数调用的现场，保存起来并重新执行。Lua,C#,python等语言也有自己的协程实现。

Go中的goroutinue就是协程,可以实现并行，多个协程可以在多个处理器同时跑。而协程同一时刻只能在一个处理器上跑（可以把宿主语言想象成单线程的就好了）。
然而,多个goroutine之间的通信是通过channel，而协程的通信是通过yield和resume()操作。

goroutine非常简单，只需要在函数的调用前面加关键字go即可，例如:

```
go elegance()
```

我们也可以启动5个goroutines分别打印索引:

```
func main() {
 for i:=1;i<5;i++ {
    go func(i int) {
        fmt.Println(i)
    }(i)
 }
 // 停歇5s，保证打印全部结束
 time.Sleep(5*time.Second)
}
```

在分析goroutine执行的随机性和并发性，启动了5个goroutine，再加上main函数的主goroutine，总共有6个goroutines。由于goroutine类似于”守护线程“，异步执行的,如果主goroutine不等待片刻，可能程序就没有输出打印了。

在Golang中channel则是goroutinues之间进行通信的渠道。

可以把channel形象比喻为工厂里的传送带,一头的生产者goroutine往传输带放东西,另一头的消费者goroutinue则从输送带取东西。channel实际上是一个有类型的消息队列,遵循先进先出的特点。

1. channel的操作符号

`ch <- data` 表示data被发送给 `channel ch` ；

`data <- ch` 表示从 `channel ch` 取一个值，然后赋给data。

2. 阻塞式channel

channel默认是没有缓冲区的，也就是说，通信是阻塞的。send操作必须等到有消费者accept才算完成。

应用示例:

```
func main() {
 ch1 := make(chan int)
 go pump(ch1) // pump hangs
 fmt.Println(<-ch1) // prints only 1
}

func pump(ch chan int) {
 for i:= 1; ; i++ {
    ch <- i
 }
}
```

在函数 `pump()` 里的channel在接受到第一个元素后就被阻塞了，直到主goroutinue取走了数据。最终channel阻塞在接受第二个元素，程序只打印 1。

没有缓冲(buffer)的channel只能容纳一个元素，而带有缓冲(buffer)channel则可以非阻塞容纳N个元素。发送数据到缓冲(buffer) channel不会被阻塞，除非channel已满；同样的，从缓冲(buffer) channel取数据也不会被阻塞，除非channel空了。

16. 怎么查看Goroutine的数量

在Golang中, `GOMAXPROCS` 中控制的是未被阻塞的所有Goroutine,可以被 `Multiplex` 到多少个线程上运行,通过 `GOMAXPROCS` 可以查看Goroutine的数量。

17. Go中的锁有哪些

Go中的三种锁包括:互斥锁,读写锁, `sync.Map` 的安全的锁.

- 互斥锁

Go并发程序对共享资源进行访问控制的主要手段，由标准库代码包中sync中的Mutex结构体表示。

```
// Mutex 是互斥锁， 零值是解锁的互斥锁， 首次使用后不得复制互斥锁。
type Mutex struct {
    state int32
    sema  uint32
}
```

sync.Mutex包中的类型只有两个公开的指针方法Lock和Unlock。

```
// Locker表示可以锁定和解锁的对象。
type Locker interface {
    Lock()
    Unlock()
}

// 锁定当前的互斥量
// 如果锁已被使用，则调用goroutine
// 阻塞直到互斥锁可用。
func (m *Mutex) Lock()

// 对当前互斥量进行解锁
// 如果在进入解锁时未锁定m，则为运行时错误。
// 锁定的互斥锁与特定的goroutine无关。
// 允许一个goroutine锁定Mutex然后安排另一个goroutine来解锁它。
func (m *Mutex) Unlock()
```

声明一个互斥锁：

```
var mutex sync.Mutex
```

不像C或Java的锁类工具，我们可能会犯一个错误：忘记及时解开已被锁住的锁，从而导致流程异常。但Go由于存在defer，所以此类问题出现的概率极低。关于defer解锁的方式如下：

```
var mutex sync.Mutex
func Write()  {
    mutex.Lock()
```

```
    defer mutex.Unlock()
}
```

如果对一个已经上锁的对象再次上锁，那么就会导致该锁定操作被阻塞，直到该互斥锁回到被解锁状态.

```
fpackage main

import (
  "fmt"
  "sync"
  "time"
)

func main() {
    var mutex sync.Mutex
    fmt.Println("begin lock")
    mutex.Lock()
    fmt.Println("get locked")
    for i := 1; i <= 3; i++ {
        go func(i int) {
            fmt.Println("begin lock ", i)
            mutex.Lock()
            fmt.Println("get locked ", i)
        }(i)
    }
    time.Sleep(time.Second)
    fmt.Println("Unlock the lock")
    mutex.Unlock()
    fmt.Println("get unlocked")
    time.Sleep(time.Second)
}
```

我们在for循环之前开始加锁，然后在每一次循环中创建一个协程，并对其加锁，但是由于之前已经加锁了，所以这个for循环中的加锁会陷入阻塞直到main中的锁被解锁， time.Sleep(time.Second) 是为了能让系统有足够的时间运行for循环，输出结果如下：

```
> go run mutex.go
begin lock
get locked
begin lock  3
begin lock  1
begin lock  2
Unlock the lock
get unlocked
get locked  3
```

这里可以看到解锁后，三个协程会重新抢夺互斥锁权，最终协程3获胜。

互斥锁锁定操作的逆操作并不会导致协程阻塞，但是有可能导致引发一个无法恢复的运行时的panic，比如对一个未锁定的互斥锁进行解锁时就会发生panic。避免这种情况的最有效方式就是使用defer。

我们知道如果遇到panic，可以使用recover方法进行恢复，但是如果对重复解锁互斥锁引发的panic却是无用的（Go 1.8及以后）。

```
package main

import (
  "fmt"
```

```
  "sync"
)

func main() {
    defer func() {
        fmt.Println("Try to recover the panic")
        if p := recover(); p != nil {
            fmt.Println("recover the panic : ", p)
        }
    }()
    var mutex sync.Mutex
    fmt.Println("begin lock")
    mutex.Lock()
    fmt.Println("get locked")
    fmt.Println("unlock lock")
    mutex.Unlock()
    fmt.Println("lock is unlocked")
    fmt.Println("unlock lock again")
    mutex.Unlock()
}
```

运行:

```
> go run mutex.go
begin lock
get locked
unlock lock
lock is unlocked
unlock lock again
fatal error: sync: unlock of unlocked mutex

goroutine 1 [running]:
runtime.throw(0x4bc1a8, 0x1e)
     /home/keke/soft/go/src/runtime/panic.go:617 +0x72 fp=0xc000084ea8 sp=0xc000084e78 pc=0x427ba2
sync.throw(0x4bc1a8, 0x1e)
     /home/keke/soft/go/src/runtime/panic.go:603 +0x35 fp=0xc000084ec8 sp=0xc000084ea8 pc=0x427b25
sync.(*Mutex).Unlock(0xc00001a0c8)
     /home/keke/soft/go/src/sync/mutex.go:184 +0xc1 fp=0xc000084ef0 sp=0xc000084ec8 pc=0x45f821
main.main()
     /home/keke/go/Test/mutex.go:25 +0x25f fp=0xc000084f98 sp=0xc000084ef0 pc=0x486c1f
runtime.main()
     /home/keke/soft/go/src/runtime/proc.go:200 +0x20c fp=0xc000084fe0 sp=0xc000084f98 pc=0x4294ec
runtime.goexit()
     /home/keke/soft/go/src/runtime/asm_amd64.s:1337 +0x1 fp=0xc000084fe8 sp=0xc000084fe0 pc=0x450ad1
exit status 2
```

这里试图对重复解锁引发的panic进行recover，但是我们发现操作失败，虽然互斥锁可以被多个协程共享，但还是建议将对同一个互斥锁的加锁解锁操作放在同一个层次的代码中。

- 读写锁

读写锁是针对读写操作的互斥锁，可以分别针对读操作与写操作进行锁定和解锁操作 。

读写锁的访问控制规则如下：

① 多个写操作之间是互斥的
② 写操作与读操作之间也是互斥的
③ 多个读操作之间不是互斥的

在这样的控制规则下，读写锁可以大大降低性能损耗。

在Go的标准库代码包中sync中的RWMutex结构体表示为:

```
// RWMutex是一个读/写互斥锁，可以由任意数量的读操作或单个写操作持有。
// RWMutex的零值是未锁定的互斥锁。
// 首次使用后，不得复制RWMutex。
// 如果goroutine持有RWMutex进行读取而另一个goroutine可能会调用Lock，那么在释放初始读锁之前，goroutine不应该期望能够获取读锁定。
// 特别是，这种禁止递归读锁定。 这是为了确保锁最终变得可用；阻止的锁定会阻止新读操作获取锁定。
type RWMutex struct {
    w           Mutex  //如果有待处理的写操作就持有
    writerSem   uint32 // 写操作等待读操作完成的信号量
    readerSem   uint32 //读操作等待写操作完成的信号量
    readerCount int32  // 待处理的读操作数量
    readerWait  int32  // number of departing readers
}
```

sync中的RWMutex有以下几种方法：

```
//对读操作的锁定
func (rw *RWMutex) RLock()

//对读操作的解锁
func (rw *RWMutex) RUnlock()

//对写操作的锁定
func (rw *RWMutex) Lock()

//对写操作的解锁
func (rw *RWMutex) Unlock()

//返回一个实现了sync.Locker接口类型的值，实际上是回调rw.RLock and rw.RUnlock.
func (rw *RWMutex) RLocker() Locker
```

Unlock方法会试图唤醒所有想进行读锁定而被阻塞的协程，而 RUnlock方法只会在已无任何读锁定的情况下，试图唤醒一个因欲进行写锁定而被阻塞的协程。

若对一个未被写锁定的读写锁进行写解锁，就会引发一个不可恢复的panic，同理对一个未被读锁定的读写锁进行读写锁也会如此。

由于读写锁控制下的多个读操作之间不是互斥的，因此对于读解锁更容易被忽视。对于同一个读写锁，添加多少个读锁定，就必要有等量的读解锁，这样才能其他协程有机会进行操作。

因此Go中读写锁，在多个读线程可以同时访问共享数据,写线程必须等待所有读线程都释放锁以后，才能取得锁.

同样的，读线程必须等待写线程释放锁后，才能取得锁，也就是说读写锁要确保的是如下互斥关系，可以同时读，但是读-写，写-写都是互斥的。

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
```

```
var rwm sync.RWMutex

for i := 0; i < 5; i++ {
    go func(i int) {
        fmt.Println("try to lock read ", i)
        rwm.RLock()
        fmt.Println("get locked ", i)
        time.Sleep(time.Second * 2)
        fmt.Println("try to unlock for reading ", i)
        rwm.RUnlock()
        fmt.Println("unlocked for reading ", i)
    }(i)
}

 time.Sleep(time.Millisecond * 1000)
 fmt.Println("try to lock for writing")

 rwm.Lock()
 fmt.Println("locked for writing")
}
```

运行:

```
> go run rwmutex.go
try to lock read  0
get locked  0
try to lock read  4
get locked  4
try to lock read  3
get locked  3
try to lock read  1
get locked  1
try to lock read  2
get locked  2
try to lock for writing
try to unlock for reading  0
unlocked for reading  0
try to unlock for reading  2
unlocked for reading  2
try to unlock for reading  1
unlocked for reading  1
try to unlock for reading  3
unlocked for reading  3
try to unlock for reading  4
unlocked for reading  4
locked for writing
```

这里可以看到创建了五个协程用于对读写锁的读锁定与读解锁操作。在 rwm.Lock()种会对main中协程进行写锁定，但是for循环中的读解锁尚未完成，因此会造成main中的协程阻塞。当for循环中的读解锁操作都完成后就会试图唤醒main中阻塞的协程，main中的写锁定才会完成。

- sync.Map安全锁

golang中的sync.Map是并发安全的，其实也就是sync包中golang自定义的一个名叫Map的结构体。

应用示例:

```
package main
import (
 "sync"
 "fmt"
)

func main() {
 //开箱即用
 var sm sync.Map

 //store 方法,添加元素
 sm.Store(1,"a")

 //Load 方法，获得value
 if v,ok:=sm.Load(1);ok{
     fmt.Println(v)
 }

 //LoadOrStore方法，获取或者保存
 //参数是一对key：value，如果该key存在且没有被标记删除则返回原先的value（不更新）和true；不存在则store，返回该value 和false
 if vv,ok:=sm.LoadOrStore(1,"c");ok{
     fmt.Println(vv)
 }
 if vv,ok:=sm.LoadOrStore(2,"c");!ok{
     fmt.Println(vv)
 }

 //遍历该map，参数是个函数，该函数参的两个参数是遍历获得的key和value，返回一个bool值，当返回false时，遍历立刻结束。
 sm.Range(func(k,v interface{})bool{
     fmt.Print(k)
     fmt.Print(":")
     fmt.Print(v)
     fmt.Println()
     return true
 })
}
```

运行：

```
a
a
c
1:a
2:c
```

sync.Map的数据结构:

```
type Map struct {
 // 该锁用来保护dirty
 mu Mutex
 // 存读的数据，因为是atomic.value类型，只读类型，所以它的读是并发安全的
 read atomic.Value // readOnly
 //包含最新的写入的数据，并且在写的时候，会把read 中未被删除的数据拷贝到该dirty中，因为是普通的map存在并发安全问题，需要用到上面的mu字段。
 dirty map[interface{}]*entry
 // 从read读数据的时候，会将该字段+1，当等于len（dirty）的时候，会将dirty拷贝到read中（从而提升读的性能）。
```

```
 misses int
}
```

read的数据结构是：

```
type readOnly struct {
  m  map[interface{}]*entry
  // 如果Map.dirty的数据和m 中的数据不一样是为true
  amended bool
}
```

entry的数据结构：

```
type entry struct {
 //可见value是个指针类型，虽然read和dirty存在冗余情况（amended=false），但是由于是指针类型，存储的空间应该不是问题
 p unsafe.Pointer // *interface{}
}
```

Delete 方法:

```
func (m *Map) Delete(key interface{}) {
 read, _ := m.read.Load().(readOnly)
 e, ok := read.m[key]
 //如果read中没有，并且dirty中有新元素，那么就去dirty中去找
 if !ok && read.amended {
     m.mu.Lock()
     //这是双检查（上面的if判断和锁不是一个原子性操作）
     read, _ = m.read.Load().(readOnly)
     e, ok = read.m[key]
     if !ok && read.amended {
         //直接删除
         delete(m.dirty, key)
     }
     m.mu.Unlock()
 }
 if ok {
 //如果read中存在该key，则将该value 赋值nil（采用标记的方式删除！）
     e.delete()
 }
}

func (e *entry) delete() (hadValue bool) {
 for {
     p := atomic.LoadPointer(&e.p)
     if p == nil || p == expunged {
         return false
     }
     if atomic.CompareAndSwapPointer(&e.p, p, nil) {
         return true
     }
 }
}
```

Store 方法:

```
func (m *Map) Store(key, value interface{}) {
 // 如果m.read存在这个key，并且没有被标记删除，则尝试更新。
 read, _ := m.read.Load().(readOnly)
 if e, ok := read.m[key]; ok && e.tryStore(&value) {
     return
 }
 // 如果read不存在或者已经被标记删除
 m.mu.Lock()
 read, _ = m.read.Load().(readOnly)
 if e, ok := read.m[key]; ok {
 //如果entry被标记expunge，则表明dirty没有key，可添加入dirty，并更新entry
     if e.unexpungeLocked() {
         //加入dirty中
         m.dirty[key] = e
     }
     //更新value值
     e.storeLocked(&value)
     //dirty 存在该key，更新
 } else if e, ok := m.dirty[key]; ok {
     e.storeLocked(&value)
     //read 和dirty都没有，新添加一条
 } else {
  //dirty中没有新的数据，往dirty中增加第一个新键
     if !read.amended {
         //将read中未删除的数据加入到dirty中
         m.dirtyLocked()
         m.read.Store(readOnly{m: read.m, amended: true})
     }
     m.dirty[key] = newEntry(value)
 }
 m.mu.Unlock()
}

//将read中未删除的数据加入到dirty中
func (m *Map) dirtyLocked() {
 if m.dirty != nil {
     return
 }
 read, _ := m.read.Load().(readOnly)
 m.dirty = make(map[interface{}]*entry, len(read.m))
 //read如果较大的话，可能影响性能
 for k, e := range read.m {
 //通过此次操作，dirty中的元素都是未被删除的，可见expunge的元素不在dirty中
     if !e.tryExpungeLocked() {
         m.dirty[k] = e
     }
 }
}

//判断entry是否被标记删除，并且将标记为nil的entry更新标记为expunge
func (e *entry) tryExpungeLocked() (isExpunged bool) {
 p := atomic.LoadPointer(&e.p)
 for p == nil {
     // 将已经删除标记为nil的数据标记为expunged
     if atomic.CompareAndSwapPointer(&e.p, nil, expunged) {
         return true
     }
     p = atomic.LoadPointer(&e.p)
 }
 return p == expunged
}
```

```
//对entry 尝试更新
func (e *entry) tryStore(i *interface{}) bool {
 p := atomic.LoadPointer(&e.p)
 if p == expunged {
     return false
 }
 for {
     if atomic.CompareAndSwapPointer(&e.p, p, unsafe.Pointer(i)) {
         return true
     }
     p = atomic.LoadPointer(&e.p)
     if p == expunged {
         return false
     }
 }
}

//read里 将标记为expunge的更新为nil
func (e *entry) unexpungeLocked() (wasExpunged bool) {
 return atomic.CompareAndSwapPointer(&e.p, expunged, nil)
}

//更新entry
func (e *entry) storeLocked(i *interface{}) {
 atomic.StorePointer(&e.p, unsafe.Pointer(i))
}
```

因此，每次操作先检查read，因为read 并发安全，性能好些；read不满足，则加锁检查dirty，一旦是新的键值，dirty会被read更新。

Load方法:

Load方法是一个加载方法，查找key。

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool) {
 //因read只读，线程安全，先查看是否满足条件
 read, _ := m.read.Load().(readOnly)
 e, ok := read.m[key]
 //如果read没有，并且dirty有新数据，那从dirty中查找，由于dirty是普通map，线程不安全，这个时候用到互斥锁了
 if !ok && read.amended {
     m.mu.Lock()
     // 双重检查
     read, _ = m.read.Load().(readOnly)
     e, ok = read.m[key]
     // 如果read中还是不存在，并且dirty中有新数据
     if !ok && read.amended {
         e, ok = m.dirty[key]
         // missLocked()函数是性能是sync.Map 性能得以保证的重要函数，目的讲有锁的dirty数据，替换到只读线程安全的read里
         m.missLocked()
     }
     m.mu.Unlock()
 }
 if !ok {
     return nil, false
 }
 return e.load()
}
```

```
//dirty 提升至read 关键函数，当misses 经过多次因为load之后，大小等于len（dirty）时候，讲dirty替换到read里，以此达到性能提升。
func (m *Map) missLocked() {
  misses++
  if m.misses < len(m.dirty) {
      return
  }
  //原子操作，耗时很小
  m.read.Store(readOnly{m: m.dirty})
  m.dirty = nil
  m.misses = 0
}
```

sync.Map是通过冗余的两个数据结构(read、dirty),实现性能的提升。

为了提升性能，load、delete、store等操作尽量使用只读的read；为了提高read的key击中概率，采用动态调整，将dirty数据提升为read；对于数据的删除，采用延迟标记删除法，只有在提升dirty的时候才删除。

18. 怎么限制Goroutine的数量

在Golang中，Goroutine虽然很好，但是数量太多了，往往会带来很多麻烦，比如耗尽系统资源导致程序崩溃，或者CPU使用率过高导致系统忙不过来。

所以我们可以限制下Goroutine的数量,这样就需要在每一次执行go之前判断goroutine的数量，如果数量超了，就要阻塞go的执行。

所以通常我们第一时间想到的就是使用通道。每次执行的go之前向通道写入值，直到通道满的时候就阻塞了，

```
package main

import "fmt"

var ch chan  int

func elegance(){
    <-ch
    fmt.Println("the ch value receive",ch)
}

func main(){
    ch = make(chan int,5)
    for i:=0;i<10;i++{
        ch <-1
        fmt.Println("the ch value send",ch)
        go elegance()
        fmt.Println("the result i",i)
    }

}
```

运行:

```
> go run goroutine.go
the ch value send 0xc00009c000
the result i 0
the ch value send 0xc00009c000
the result i 1
the ch value send 0xc00009c000
the result i 2
```

```
the ch value send 0xc00009c000
the result i 3
the ch value send 0xc00009c000
the result i 4
the ch value send 0xc00009c000
the result i 5
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the result i 6
the ch value receive 0xc00009c000
the ch value send 0xc00009c000
the result i 7
the ch value send 0xc00009c000
the result i 8
the ch value send 0xc00009c000
the result i 9
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the result i 10
the ch value send 0xc00009c000
the result i 11
the ch value send 0xc00009c000
the result i 12
the ch value send 0xc00009c000
the result i 13
the ch value send 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the ch value receive 0xc00009c000
the result i 14
the ch value receive 0xc00009c000
```

```
> go run goroutine.go
the ch value send 0xc00007e000
the result i 0
the ch value send 0xc00007e000
the result i 1
the ch value send 0xc00007e000
the result i 2
the ch value send 0xc00007e000
the result i 3
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 4
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 5
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the result i 6
the ch value send 0xc00007e000
the result i 7
the ch value send 0xc00007e000
the ch value receive 0xc00007e000
the ch value receive 0xc00007e000
the ch value receive 0xc00007e000
the result i 8
```

```
the ch value send 0xc00007e000
the result i 9
```

这样每次同时运行的goroutine就被限制为5个了。但是新的问题于是就出现了，因为并不是所有的goroutine都执行完了，在main函数退出之后，还有一些goroutine没有执行完就被强制结束了。这个时候我们就需要用到sync.WaitGroup。使用WaitGroup等待所有的goroutine退出。

```
package main

import (
    "fmt"
    "runtime"
    "sync"
    "time"
)

// Pool Goroutine Pool
type Pool struct {
    queue chan int
    wg *sync.WaitGroup
}

// New 新建一个协程池
func NewPool(size int) *Pool{
    if size <=0{
        size = 1
    }
    return &Pool{
        queue:make(chan int,size),
        wg:&sync.WaitGroup{},
    }
}

// Add 新增一个执行
func (p *Pool)Add(delta int){
    // delta为正数就添加
    for i :=0;i<delta;i++{
        p.queue <-1
    }
    // delta为负数就减少
    for i:=0;i>delta;i--{
        <-p.queue
    }
    p.wg.Add(delta)
}

// Done 执行完成减一
func (p *Pool) Done(){
    <-p.queue
    p.wg.Done()
}

// Wait 等待Goroutine执行完毕
func (p *Pool) Wait(){
    p.wg.Wait()
}

func main(){
    // 这里限制5个并发
    pool := NewPool(5)
```

```
    fmt.Println("the NumGoroutine begin is:",runtime.NumGoroutine())
    for i:=0;i<20;i++{
        pool.Add(1)
        go func(i int) {
            time.Sleep(time.Second)
            fmt.Println("the NumGoroutine continue is:",runtime.NumGoroutine())
            pool.Done()
        }(i)
    }
    pool.Wait()
    fmt.Println("the NumGoroutine done is:",runtime.NumGoroutine())
}
```

运行:

```
the NumGoroutine begin is: 1
the NumGoroutine continue is: 6
the NumGoroutine continue is: 7
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 6
the NumGoroutine continue is: 3
the NumGoroutine continue is: 2
the NumGoroutine done is: 1
```

其中，Go的GOMAXPROCS默认值已经设置为CPU的核数， 这里允许我们的Go程序充分使用机器的每一个CPU,最大程度的提高我们程序的并发性能。 `runtime.NumGoroutine` 函数在被调用后，会返回系统中的处于特定状态的Goroutine的数量。这里的特指是指 `Grunnable\Gruning\Gsyscall\Gwaition` 。处于这些状态的Groutine即被看做是活跃的或者说正在被调度。

这里需要注意下：垃圾回收所在Groutine的状态也处于这个范围内的话，也会被纳入该计数器。

19. Channel是同步的还是异步的

Channel是异步进行的, channel存在3种状态：

- nil，未初始化的状态，只进行了声明，或者手动赋值为nil
- active，正常的channel，可读或者可写
- closed，已关闭，千万不要误认为关闭channel后，channel的值是nil

下面我们对channel的三种操作解析:

1. 零值（nil）通道；
2. 非零值但已关闭的通道；
3. 非零值并且尚未关闭的通道。

操作	一个零值nil通道	一个非零值但已关闭的通道	一个非零值且尚未关闭的通
关闭	产生恐慌	产生恐慌	成功关闭

发送数据	永久阻塞	产生恐慌	阻塞或者成功发送
接收数据	永久阻塞	永不阻塞	阻塞或者成功接收

20. **Goroutine和线程的区别**

从调度上看，goroutine的调度开销远远小于线程调度开销。

OS的线程由OS内核调度，每隔几毫秒，一个硬件时钟中断发到CPU，CPU调用一个调度器内核函数。这个函数暂停当前正在运行的线程，把他的寄存器信息保存到内存中，查看线程列表并决定接下来运行哪一个线程，再从内存中恢复线程的注册表信息，最后继续执行选中的线程。这种线程切换需要一个完整的上下文切换：即保存一个线程的状态到内存，再恢复另外一个线程的状态，最后更新调度器的数据结构。某种意义上，这种操作还是很慢的。

Go运行的时候包涵一个自己的调度器，这个调度器使用一个称为一个M:N调度技术，m个goroutine到n个os线程（可以用GOMAXPROCS来控制n的数量），Go的调度器不是由硬件时钟来定期触发的，而是由特定的go语言结构来触发的，他不需要切换到内核语境，所以调度一个goroutine比调度一个线程的成本低很多。

从栈空间上，goroutine的栈空间更加动态灵活。

每个OS的线程都有一个固定大小的栈内存，通常是2MB，栈内存用于保存在其他函数调用期间哪些正在执行或者临时暂停的函数的局部变量。这个固定的栈大小，如果对于goroutine来说，可能是一种巨大的浪费。作为对比goroutine在生命周期开始只有一个很小的栈，典型情况是2KB, 在go程序中，一次创建十万左右的goroutine也不罕见（2KB*100,000=200MB）。而且goroutine的栈不是固定大小，它可以按需增大和缩小，最大限制可以到1GB。

goroutine没有一个特定的标识。

在大部分支持多线程的操作系统和编程语言中，线程有一个独特的标识，通常是一个整数或者指针，这个特性可以让我们构建一个线程的局部存储，本质是一个全局的map，以线程的标识作为键，这样每个线程可以独立使用这个map存储和获取值，不受其他线程干扰。

goroutine中没有可供程序员访问的标识，原因是一种纯函数的理念，不希望滥用线程局部存储导致一个不健康的超距作用，即函数的行为不仅取决于它的参数，还取决于运行它的线程标识。

21. **Go的Struct能不能比较**

- 相同struct类型的可以比较
- 不同struct类型的不可以比较,编译都不过，类型不匹配

```
package main

import "fmt"

func main() {
    type A struct {
        a int
    }
    type B struct {
```

```
        a int
    }
    a := A{1}
    //b := A{1}
    b := B{1}
    if a == b {
        fmt.Println("a == b")
    }else{
        fmt.Println("a != b")
    }
}

output:

> command-line-arguments [command-line-arguments.test]
> ./.go:14:7: invalid operation: a == b (mismatched types A and B)
```

22. Go的defer原理是什么

什么是defer？如何理解 defer 关键字？Go 中使用 defer 的一些坑。

defer 意为延迟，在 golang 中用于延迟执行一个函数。它可以帮助我们处理容易忽略的问题，如资源释放、连接关闭等。但在实际使用过程中，有一些需要注意的地方.

1. 若函数中有多个 defer，其执行顺序为 先进后出，可以理解为栈。

```
package main

import "fmt"

func main() {
  for i := 0; i < 5; i++ {
    defer fmt.Println(i)
  }
}
```

运行:

```
4
3
2
1
0
```

2. return 会做什么呢?

Go 的函数返回值是通过堆栈返回的, return 语句不是原子操作，而是被拆成了两步.

- 给返回值赋值 (rval)
- 调用 defer 表达式
- 返回给调用函数(ret)

```
package main

import "fmt"
```

```
func main() {
    fmt.Println(increase(1))
}

func increase(d int) (ret int) {
  defer func() {
    ret++
  }()

  return d
}
```

运行输出:

```
2
```

3. 若 defer 表达式有返回值，将会被丢弃。

闭包与匿名函数.

- 匿名函数：没有函数名的函数。
- 闭包：可以使用另外一个函数作用域中的变量的函数。

在实际开发中，defer 的使用经常伴随着闭包与匿名函数的使用。

```
package main

import "fmt"

func main() {
    for i := 0; i < 5; i++ {
        defer func() {
            fmt.Println(i)
        }()
    }
}
```

运行输出:

```
5
5
5
5
5
```

之所以这样是因为,defer 表达式中的 i 是对 for 循环中 i 的引用。到最后，i 加到 5，故最后全部打印 5。

如果将 i 作为参数传入 defer 表达式中，在传入最初就会进行求值保存，只是没有执行延迟函数而已。

应用示例:

```
func f1() (result int) {
    defer func() {
        result++
    }()
```

```
    return 0
}
```

```
func f2() (r int) {
    t := 5
   defer func() {
    t = t + 5
   }()
   return t
}
```

```
func f3() (r int) {
    defer func(r int) {
        r = r + 5
    }(r)
    return 1
}
```

```
type Test struct {
    Max int
}

func (t *Test) Println() {
    fmt.Println(t.Max)
}

func deferExec(f func()) {
    f()
}

func call() {
    var t *Test
    defer deferExec(t.Println)

    t = new(Test)
}
```

有没有得出结果？例1的答案不是 0，例2的答案不是 10，例3的答案也不是 6。

defer是在return之前执行的。这个在官方文档中是明确说明了的。要使用defer时不踩坑，最重要的一点就是要明白，`return xxx` 这一条语句并不是一条原子指令!

```
函数返回的过程是这样的：先给返回值赋值，然后调用defer表达式，最后才是返回到调用函数中。

defer表达式可能会在设置函数返回值之后，在返回到调用函数之前，修改返回值，使最终的函数返回值与你想象的不一致。

其实使用defer时，用一个简单的转换规则改写一下，就不会迷糊了。改写规则是将return语句拆成两句写，return xxx会被改写成：

返回值 = xxx
调用defer函数
空的return
```

f1: 比较简单，参考结论2，将 0 赋给 result，defer 延迟函数修改 result，最后返回给调用函数。正确答案是 1。

f1可以修改成长这样的:

```
func f() (result int) {
    result = 0  // return语句不是一条原子调用，return xxx其实是赋值＋ret指令
    func() { // defer被插入到return之前执行，也就是赋返回值和ret指令之间
        result++
    }()
    return
}
```

所以这个返回值是1。

f2: defer 是在 t 赋值给 r 之后执行的，而 defer 延迟函数只改变了 t 的值，r 不变。正确答案 5。

f2可以修改成这样的:

```
func f() (r int) {
    t := 5
    r = t // 赋值指令
    func() {        // defer被插入到赋值与返回之间执行，这个例子中返回值r没被修改过
        t = t + 5
    }
    return        // 空的return指令
}
```

所以这个的结果是5。

f3: 这里将 r 作为参数传入了 defer 表达式。故 func (r int) 中的 r 非 func f() (r int) 中的 r，只是参数命名相同而已。正确答案 1。

f3可以修改成这样的:

```
func f() (r int) {
    r = 1  // 给返回值赋值
    func(r int) {        // 这里改的r是传值传进去的r，不会改变要返回的那个r值
        r = r + 5
    }(r)
    return        // 空的return
}
```

所以这个例子的结果是1。

f4: 这里将发生 panic。将方法传给 deferExec，实际上在传的过程中对方法求了值。而此时的 t 任然为 nil。

因此, defer确实是在return之前调用的。但表现形式上却可能不像。根本原因是 `return xxx` 语句并不是一条原子指令，defer被插入到了赋值 与 ret之间，因此可能有机会改变最终的返回值。

defer关键字的实现跟go关键字很类似，不同的是它调用的是 `runtime.deferproc` 而不是 `runtime.newproc` 。

在defer出现的地方，插入了指令 `call runtime.deferproc` ，然后在函数返回之前的地方，插入指令 `call runtime.deferreturn` 。

普通的函数返回时，汇编代码类似:

```
add xx SP
return
```

如果其中包含了defer语句，则汇编代码是：

```
call runtime.deferreturn,
add xx SP
return
```

goroutine的控制结构中，有一张表记录defer，调用 `runtime.deferproc` 时会将需要defer的表达式记录在表中，而在调用 `runtime.deferreturn` 的时候，则会依次从defer表中出栈并执行。

23. Go的select可以用于什么

Golang 的 select 机制可以理解为是在语言层面实现了和 select, poll, epoll 相似的功能：监听多个描述符的读/写等事件，一旦某个描述符就绪（一般是读或者写事件发生了），就能够将发生的事件通知给关心的应用程序去处理该事件。 golang 的 select 机制是，监听多个channel，每一个 case 是一个事件，可以是读事件也可以是写事件，随机选择一个执行，可以设置 default，它的作用是：当监听的多个事件都阻塞住会执行default的逻辑。

select的源码在 (runtime/select.go)[https://github.com/golang/go/blob/master/src/runtime/select.go] ，看的时候建议是重点关注 pollorder 和 lockorder.

- pollorder保存的是scase的序号，乱序是为了之后执行时的随机性。
- lockorder保存了所有case中channel的地址，这里按照地址大小堆排了一下lockorder对应的这片连续内存。对chan排序是为了去重，保证之后对所有channel上锁时不会重复上锁。

goroutine作为Golang并发的核心，我们不仅要关注它们的创建和管理，当然还要关注如何合理的退出这些协程，不（合理）退出不然可能会造成阻塞、panic、程序行为异常、数据结果不正确等问题。goroutine在退出方面，不像线程和进程，不能通过某种手段强制关闭它们，只能等待goroutine主动退出。

goroutine的优雅退出方法有三种:

1. 使用for-range退出

for-range是使用频率很高的结构，常用它来遍历数据，range能够感知channel的关闭，当channel被发送数据的协程关闭时，range就会结束，接着退出for循环。

它在并发中的使用场景是：当协程只从1个channel读取数据，然后进行处理，处理后协程退出。下面这个示例程序，当in通道被关闭时，协程可自动退出。

```
go func(in <-chan int) {
    // Using for-range to exit goroutine
    // range has the ability to detect the close/end of a channel
    for x := range in {
        fmt.Printf("Process %d\n", x)
    }
}(in)
```

2. 使用select case ,ok退出

for-select也是使用频率很高的结构，select提供了多路复用的能力，所以for-select可以让函数具有持续多路处理多个channel的能力。但select没有感知channel的关闭，这引出了2个问题：

继续在关闭的通道上读，会读到通道传输数据类型的零值，如果是指针类型，读到nil，继续处理还会产生nil。
继续在关闭的通道上写，将会panic。

问题2可以这样解决，通道只由发送方关闭，接收方不可关闭，即某个写通道只由使用该select的协程关闭，select中就不存在继续在关闭的通道上写数据的问题。

问题1可以使用,ok来检测通道的关闭，使用情况有2种。

第一种：如果某个通道关闭后，需要退出协程，直接return即可。示例代码中，该协程需要从in通道读数据，还需要定时打印已经处理的数量，有2件事要做，所有不能使用for-range，需要使用for-select，当in关闭时，ok=false，我们直接返回。

```
go func() {
    // in for-select using ok to exit goroutine
    for {
        select {
        case x, ok := <-in:
            if !ok {
                return
            }
            fmt.Printf("Process %d\n", x)
            processedCnt++
        case <-t.C:
            fmt.Printf("Working, processedCnt = %d\n", processedCnt)
        }
    }
}()
```

第二种：如果某个通道关闭了，不再处理该通道，而是继续处理其他case，退出是等待所有的可读通道关闭。我们需要使用select的一个特征：select不会在nil的通道上进行等待。这种情况，把只读通道设置为nil即可解决。

```
go func() {
    // in for-select using ok to exit goroutine
    for {
        select {
        case x, ok := <-in1:
            if !ok {
                in1 = nil
            }
            // Process
        case y, ok := <-in2:
            if !ok {
                in2 = nil
            }
            // Process
        case <-t.C:
            fmt.Printf("Working, processedCnt = %d\n", processedCnt)
        }

        // If both in channel are closed, goroutine exit
        if in1 == nil && in2 == nil {
            return
        }
    }
}()
```

3. 使用退出通道退出

使用,ok来退出使用for-select协程，解决是当读入数据的通道关闭时，没数据读时程序的正常结束。想想下面这2种场景，,ok还能适用吗？

接收的协程要退出了，如果它直接退出，不告知发送协程，发送协程将阻塞。启动了一个工作协程处理数据，如何通知它退出？

使用一个专门的通道，发送退出的信号，可以解决这类问题。以第2个场景为例，协程入参包含一个停止通道stopCh，当stopCh被关闭，case <-stopCh会执行，直接返回即可。

当我启动了100个worker时，只要main()执行关闭stopCh，每一个worker都会都到信号，进而关闭。如果main()向stopCh发送100个数据，这种就低效了。

```
func worker(stopCh <-chan struct{}) {
    go func() {
        defer fmt.Println("worker exit")
        // Using stop channel explicit exit
        for {
            select {
            case <-stopCh:
                fmt.Println("Recv stop signal")
                return
            case <-t.C:
                fmt.Println("Working .")
            }
        }
    }()
    return
}
```

通过channel控制子goroutine的方法可以总结为：循环监听一个channel，一般来说是for循环里放一个select监听channel以达到通知子goroutine的效果。再借助Waitgroup，主进程可以等待所有协程优雅退出后再结束自己的运行，这就通过channel实现了优雅控制goroutine并发的开始和结束。

因此在退出协程的时候需要注意:

- 发送协程主动关闭通道，接收协程不关闭通道。使用技巧：把接收方的通道入参声明为只读，如果接收协程关闭只读协程，编译时就会报错。
- 协程处理1个通道，并且是读时，协程优先使用for-range，因为range可以关闭通道的关闭自动退出协程。
- ok可以处理多个读通道关闭，需要关闭当前使用for-select的协程。
- 显式关闭通道stopCh可以处理主动通知协程退出的场景。

24. Context包的用途是什么

在 Go http包的Server中，每一个请求在都有一个对应的 goroutine 去处理。请求处理函数通常会启动额外的 goroutine 用来访问后端服务，比如数据库和RPC服务。用来处理一个请求的 goroutine 通常需要访问一些与请求特定的数据，比如终端用户的身份认证信息、验证相关的token、请求的截止时间。 当一个请求被取消或超时时，所有用来处理该请求的 goroutine 都应该迅速退出，然后系统才能释放这些 goroutine 占用的资源。

在Google 内部，我们开发了 Context 包，专门用来简化 对于处理单个请求的多个 goroutine 之间与请求域的数据、取消信号、截止时间等相关操作，这些操作可能涉及多个 API 调用。

context的数据结构是:

```
// A Context carries a deadline, cancelation signal, and request-scoped values
// across API boundaries. Its methods are safe for simultaneous use by multiple
// goroutines.
type Context interface {
    // Done returns a channel that is closed when this `Context` is canceled
    // or times out.
    Done() <-chan struct{}

    // Err indicates why this Context was canceled, after the Done channel
    // is closed.
    Err() error

    // Deadline returns the time when this Context will be canceled, if any.
```

```
    Deadline() (deadline time.Time, ok bool)

    // Value returns the value associated with key or nil if none.
    Value(key interface{}) interface{}
}
```

Context中的方法:

- Done会返回一个channel，当该context被取消的时候，该channel会被关闭，同时对应的使用该context的routine也应该结束并返回。
- Context中的方法是协程安全的，这也就代表了在父routine中创建的context，可以传递给任意数量的routine并让他们同时访问。
- Deadline会返回一个超时时间，routine获得了超时时间后，可以对某些io操作设定超时时间。
- Value可以让routine共享一些数据，当然获得数据是协程安全的。

这里需要注意一点的是在goroutine中使用context包的时候,通常我们需要在goroutine中新创建一个上下文的context,原因是:如果直接传递外部context到协层中,一个请求可能在主函数中已经结束,在goroutine中如果还没有结束的话,会直接导致goroutine中的运行的被取消.

```
go func() {
    _, ctx, _ := log.FromContextOrNew(context.Background(), nil)
}()
```

context.Background函数的返回值是一个空的context，经常作为树的根结点，它一般由接收请求的第一个routine创建，不能被取消、没有值、也没有过期时间。

Background函数的声明如下：

```
// Background returns an empty Context. It is never canceled, has no deadline,
// and has no values. Background is typically used in main, init, and tests,
// and as the top-level `Context` for incoming requests.
func Background() Context
```

WithCancel 和 WithTimeout 函数 会返回继承的 Context 对象， 这些对象可以比它们的父 Context 更早地取消。

当请求处理函数返回时，与该请求关联的 Context 会被取消。 当使用多个副本发送请求时，可以使用 WithCancel取消多余的请求。 WithTimeout 在设置对后端服务器请求截止时间时非常有用。 下面是这三个函数的声明：

```
// WithCancel returns a copy of parent whose Done channel is closed as soon as
// parent.Done is closed or cancel is called.
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

// A CancelFunc cancels a Context.
type CancelFunc func()

// WithTimeout returns a copy of parent whose Done channel is closed as soon as
// parent.Done is closed, cancel is called, or timeout elapses. The new
// Context's Deadline is the sooner of now+timeout and the parent's deadline, if
// any. If the timer is still running, the cancel function releases its
// resources.
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

调用CancelFunc对象将撤销对应的Context对象，这样父结点的所在的环境中，获得了撤销子节点context的权利，当触发某些条件时，可以调用CancelFunc对象来终止子结点树的所有routine。在子节点的routine中，需要判断何时退出routine：

```
select {
    case <-cxt.Done():
        // do some cleaning and return
}
```

根据cxt.Done()判断是否结束。当顶层的Request请求处理结束，或者外部取消了这次请求，就可以cancel掉顶层context，从而使整个请求的routine树得以退出。

WithDeadline和WithTimeout比WithCancel多了一个时间参数，它指示context存活的最长时间。如果超过了过期时间，会自动撤销它的子context。所以context的生命期是由父context的routine和deadline共同决定的。

WithValue 函数能够将请求作用域的数据与 Context 对象建立关系。声明如下：

```
type valueCtx struct {
    Context
    key, val interface{}
}

func WithValue(parent Context, key, val interface{}) Context {
    if key == nil {
        panic("nil key")
    }
    ......
    return &valueCtx{parent, key, val}
}

func (c *valueCtx) Value(key interface{}) interface{} {
    if c.key == key {
        return c.val
    }
    return c.Context.Value(key)
}
```

WithValue返回parent的一个副本，该副本保存了传入的 `key/value` ，而调用Context接口的Value(key)方法就可以得到val。注意在同一个context中设置 `key/value` ，若key相同，值会被覆盖。

Context上下文数据的存储就像一个树，每个结点只存储一个 `key/value` 对。WithValue()保存一个 `key/value` 对，它将父context嵌入到新的子context，并在节点中保存了 `key/value` 数据。Value()查询key对应的value数据，会从当前context中查询，如果查不到，会递归查询父context中的数据。

值得注意的是，context中的上下文数据并不是全局的，它只查询本节点及父节点们的数据，不能查询兄弟节点的数据。

Context 使用原则:

- 不要把Context放在结构体中，要以参数的方式传递。
- 以Context作为参数的函数方法，应该把Context作为第一个参数，放在第一位。
- 给一个函数方法传递Context的时候，不要传递nil，如果不知道传递什么，就使用context.TODO。
- Context的Value相关方法应该传递必须的数据，不要什么数据都使用这个传递。
- Context是线程安全的，可以放心的在多个goroutine中传递。

25. Go主协程如何等其余协程完再操作

Go提供了更简单的方法——使用 `sync.WaitGroup` 。 `WaitGroup` ，就是用来等待一组操作完成的。 `WaitGroup` 内部实现了一个计数器，用来记录未完成的操作个数.

它提供了三个方法， `Add()` 用来添加计数。 `Done()` 用来在操作结束时调用，使计数减一。 `Wait()` 用来等待所有的操作结束，即计数变为0，该函数会在计数不为0时等待，在计数为0时立即返回。

应用示例:

```
package main

import (
    "fmt"
    "sync"
)

func main() {

    var wg sync.WaitGroup

    wg.Add(2) // 因为有两个动作，所以增加2个计数
    go func() {
        fmt.Println("Goroutine 1")
        wg.Done() // 操作完成，减少一个计数
    }()

    go func() {
        fmt.Println("Goroutine 2")
        wg.Done() // 操作完成，减少一个计数
    }()

    wg.Wait() // 等待，直到计数为0
}
```

运行输出:

```
Goroutine 2
Goroutine 1
```

26. Go的Slice如何扩容

slice是 Go 中的一种基本的数据结构，使用这种结构可以用来管理数据集合。但是slice本身并不是动态数据或者数组指针。slice常见的操作有 reslice、append、copy。

slice自身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用底层数组，设定相关属性将数据读写操作限定在指定的区域内。slice本身是一个只读对象，其工作机制类似数组指针的一种封装。

slice是对数组一个连续片段的引用，所以切片是一个引用类型（因此更类似于 C/C++ 中的数组类型，或者 Python 中的 list 类型）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。

这里需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个与指向数组的动态窗口。

slice是可以看做是一个长度可变的数组。

slice数据结构如下:

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

slice的结构体由3部分构成，Pointer 是指向一个数组的指针，len 代表当前切片的长度，cap 是当前切片的容量。cap 总是大于等于 len 的。

通常我们在对slice进行append等操作时，可能会造成slice的自动扩容。

其扩容时的大小增长规则是：

- 如果切片的容量小于1024个元素，那么扩容的时候slice的cap就翻番，乘以2；一旦元素个数超过1024个元素，增长因子就变成1.25，即每次增加原来容量的四分之一。
- 如果扩容之后，还没有触及原数组的容量，那么，切片中的指针指向的位置，就还是原数组，如果扩容之后，超过了原数组的容量，那么，Go就会开辟一块新的内存，把原来的值拷贝过来，这种情况丝毫不会影响到原数组。

通过slice源码可以看到,append的实现只是简单的在内存中将旧slice复制给新slice.

```
newcap := old.cap

if newcap+newcap < cap {
    newcap = cap
} else {
    for {
        if old.len < 1024 {
            newcap += newcap
        } else {
            newcap += newcap / 4
        }
        if newcap >= cap {
            break
        }
    }
}
```

27. Go中的map如何实现顺序读取

Go中map如果要实现顺序读取的话,可以先把map中的key,通过sort包排序.

通过sort中的排序包进行对map中的key进行排序.

```
package main

import (
    "fmt"
    "sort"
)

func main() {
    var m = map[string]int{
        "hello":          0,
        "morning":        1,
        "keke":           2,
        "jame":             3,
    }
    var keys []string
    for k := range m {
        keys = append(keys, k)
    }
    sort.Strings(keys)
    for _, k := range keys {
        fmt.Println("Key:", k, "Value:", m[k])
```

```
    }
}
```

28. Go中CAS是怎么回事

CAS算法（Compare And Swap）,是原子操作的一种, CAS算法是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。可用于在多线程编程中实现不被打断的数据交换操作，从而避免多线程同时改写某一数据时由于执行顺序不确定性以及中断的不可预知性产生的数据不一致问题。

该操作通过将内存中的值与指定数据进行比较，当数值一样时将内存中的数据替换为新的值。

Go中的CAS操作是借用了CPU提供的原子性指令来实现。CAS操作修改共享变量时候不需要对共享变量加锁，而是通过类似乐观锁的方式进行检查，本质还是不断的占用CPU 资源换取加锁带来的开销（比如上下文切换开销）。

```
package main

import (
    "fmt"
    "sync"
    "sync/atomic"
)

var (
    counter int32          //计数器
    wg      sync.WaitGroup //信号量
)

func main() {
    threadNum := 5
    wg.Add(threadNum)
    for i := 0; i < threadNum; i++ {
        go incCounter(i)
    }
    wg.Wait()
}

func incCounter(index int) {
    defer wg.Done()

    spinNum := 0
    for {
        // 原子操作
        old := counter
        ok := atomic.CompareAndSwapInt32(&counter, old, old+1)
        if ok {
            break
        } else {
            spinNum++
        }
    }
    fmt.Printf("thread,%d,spinnum,%d\n", index, spinNum)
}
```

当主函数main首先创建了5个信号量，然后开启五个线程执行incCounter方法,incCounter内部执行，使用cas操作递增counter的值， `atomic.CompareAndSwapInt32` 具有三个参数，第一个是变量的地址，第二个是变量当前值，第三个是要修改变量为多少，该函数如果发现传递的old值等于当前变量的值，则使用第三个变量替换变量的值并返回true，否则返回false。

这里之所以使用无限循环是因为在高并发下每个线程执行CAS并不是每次都成功，失败了的线程需要重写获取变量当前的值，然后重新执行CAS操作。读者可以把线程数改为10000或者更多就会发现输出 `thread,5329,spinnum,1` 其中这个1就说明该线程尝试了两个CAS操作，第二次才成功。

因此呢，go中CAS操作可以有效的减少使用锁所带来的开销，但是需要注意在高并发下这是使用cpu资源做交换的。

29. Go中的逃逸分析是什么

在Go中逃逸分析是一种确定指针动态范围的方法，可以分析在程序的哪些地方可以访问到指针。它涉及到指针分析和形状分析。

当一个变量(或对象)在子程序中被分配时，一个指向变量的指针可能逃逸到其它执行线程中，或者去调用子程序。如果使用尾递归优化（通常在函数编程语言中是需要的），对象也可能逃逸到被调用的子程序中。 如果一个子程序分配一个对象并返回一个该对象的指针，该对象可能在程序中的任何一个地方被访问到——这样指针就成功“逃逸”了。

如果指针存储在全局变量或者其它数据结构中，它们也可能发生逃逸，这种情况是当前程序中的指针逃逸。 逃逸分析需要确定指针所有可以存储的地方，保证指针的生命周期只在当前进程或线程中。

导致内存逃逸的情况比较多，有些可能还是官方未能够实现精确的分析逃逸情况的 bug，通常来讲就是如果变量的作用域不会扩大并且其行为或者大小能够在编译的时候确定，一般情况下都是分配到栈上，否则就可能发生内存逃逸分配到堆上。

内存逃逸的五种情况:

1. 发送指针的指针或值包含了指针到 `channel` 中，由于在编译阶段无法确定其作用域与传递的路径，所以一般都会逃逸到堆上分配。
2. slices 中的值是指针的指针或包含指针字段。一个例子是类似 `[]*string` 的类型。这总是导致 slice 的逃逸。即使切片的底层存储数组仍可能位于堆栈上，数据的引用也会转移到堆中。
3. slice 由于 append 操作超出其容量，因此会导致 slice 重新分配。这种情况下，由于在编译时 slice 的初始大小的已知情况下，将会在栈上分配。如果 slice 的底层存储必须基于仅在运行时数据进行扩展，则它将分配在堆上。
4. 调用接口类型的方法。接口类型的方法调用是动态调度,实际使用的具体实现只能在运行时确定。考虑一个接口类型为 io.Reader 的变量 r。对 r.Read(b) 的调用将导致 r 的值和字节片b的后续转义并因此分配到堆上。
5. 尽管能够符合分配到栈的场景，但是其大小不能够在在编译时候确定的情况，也会分配到堆上.

有效的避免上述的五种逃逸的情况,就可以避免内存逃逸.

30. Go值接收者和指针接收者的区别

Go中的方法能给用户自定义的类型添加新的行为。它和函数的区别在于方法有一个接收者，给一个函数添加一个接收者，那么它就变成了方法。接收者可以是值接收者，也可以是指针接收者。

在调用方法的时候，值类型既可以调用值接收者的方法，也可以调用指针接收者的方法；指针类型既可以调用指针接收者的方法，也可以调用值接收者的方法。

也就是说，不管方法的接收者是什么类型，该类型的值和指针都可以调用，不必严格符合接收者的类型。

```
package main

import "fmt"

type Person struct {
    age int
}

func (p Person) Elegance() int {
    return p.age
}
```

```
func (p *Person) GetAge() {
    p.age += 1
}

func main() {
    // p1 是值类型
    p := Person{age: 18}

    // 值类型 调用接收者也是值类型的方法
    fmt.Println(p.howOld())

    // 值类型 调用接收者是指针类型的方法
    p.GetAge()
    fmt.Println(p.GetAge())

    // ----------------------

    // p2 是指针类型
    p2 := &Person{age: 100}

    // 指针类型 调用接收者是值类型的方法
    fmt.Println(p2.GetAge())

    // 指针类型 调用接收者也是指针类型的方法
    p2.GetAge()
    fmt.Println(p2.GetAge())
}
```

运行

```
18
19
100
101
```

函数和方法	值接收者	指针接收者
值类型调用者	方法会使用调用者的一个副本，类似于“传值”	使用值的引用来调用方法，上例中，p1.GetAge() 实际上是(&p1).GetAge()
指针类型调用者	指针被解引用为值，上例中，p2.GetAge()实际上是(*p1).GetAge()	实际上也是“传值”，方法里的操作会影响到调用者，类似于指针传参，拷贝了一份指针

如果实现了接收者是值类型的方法，会隐含地也实现了接收者是指针类型的方法。

如果方法的接收者是值类型，无论调用者是对象还是对象指针，修改的都是对象的副本，不影响调用者；如果方法的接收者是指针类型，则调用者修改的是指针指向的对象本身。

通常我们使用指针作为方法的接收者的理由：

- 使用指针方法能够修改接收者指向的值。

- 可以避免在每次调用方法时复制该值，在值的类型为大型结构体时，这样做会更加高效。

因而呢,我们是使用值接收者还是指针接收者，不是由该方法是否修改了调用者（也就是接收者）来决定，而是应该基于该类型的本质。

如果类型具备“原始的本质”，也就是说它的成员都是由 Go 语言里内置的原始类型，如字符串，整型值等，那就定义值接收者类型的方法。像内置的引用类型，如 slice，map，interface，channel，这些类型比较特殊，声明他们的时候，实际上是创建了一个 header， 对于他们也是直接定义值接收者类型的方法。这样，调用函数时，是直接 copy 了这些类型的 header，而 header 本身就是为复制设计的。

如果类型具备非原始的本质，不能被安全地复制，这种类型总是应该被共享，那就定义指针接收者的方法。比如 go 源码里的文件结构体（struct File）就不应该被复制，应该只有一份实体。

接口值的零值是指动态类型和动态值都为 nil。当仅且当这两部分的值都为 nil 的情况下，这个接口值就才会被认为 接口值 == nil。

31. Go的对象在内存中是怎样分配的

Go中的内存分类并不像TCMalloc那样分成小、中、大对象，但是它的小对象里又细分了一个Tiny对象，Tiny对象指大小在1Byte到16Byte之间并且不包含指针的对象。

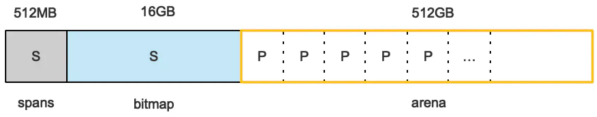
小对象和大对象只用大小划定，无其他区分。

大对象指大小大于32kb.小对象是在mcache中分配的，而大对象是直接从mheap分配的，从小对象的内存分配看起。

Go的内存分配原则:

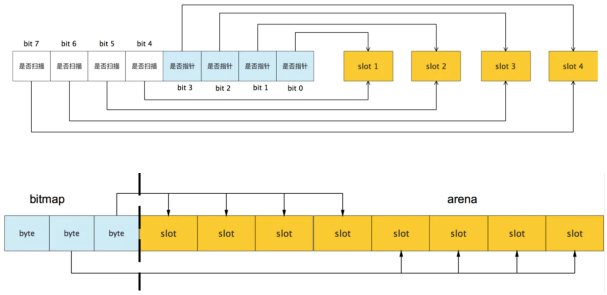
Go在程序启动的时候，会先向操作系统申请一块内存（注意这时还只是一段虚拟的地址空间，并不会真正地分配内存），切成小块后自己进行管理。

申请到的内存块被分配了三个区域，在X64上分别是512MB，16GB，512GB大小。



arena区域就是我们所谓的堆区，Go动态分配的内存都是在这个区域，它把内存分割成8KB大小的页，一些页组合起来称为mspan。

bitmap区域标识arena区域哪些地址保存了对象，并且用4bit标志位表示对象是否包含指针、GC标记信息。bitmap中一个byte大小的内存对应arena区域中4个指针大小（指针大小为 8B ）的内存，所以bitmap区域的大小是 `512GB/(4*8B)=16GB` 。



此外我们还可以看到bitmap的高地址部分指向arena区域的低地址部分，这里bitmap的地址是由高地址向低地址增长的。

spans区域存放mspan（是一些arena分割的页组合起来的内存管理基本单元，后文会再讲）的指针，每个指针对应一页，所以spans区域的大小就是 `512GB/8KB*8B=512MB` 。

除以8KB是计算arena区域的页数，而最后乘以8是计算spans区域所有指针的大小。创建mspan的时候，按页填充对应的spans区域，在回收object时，根据地址很容易就能找到它所属的mspan。

32. 栈的内存是怎么分配的

栈和堆只是虚拟内存上2块不同功能的内存区域：

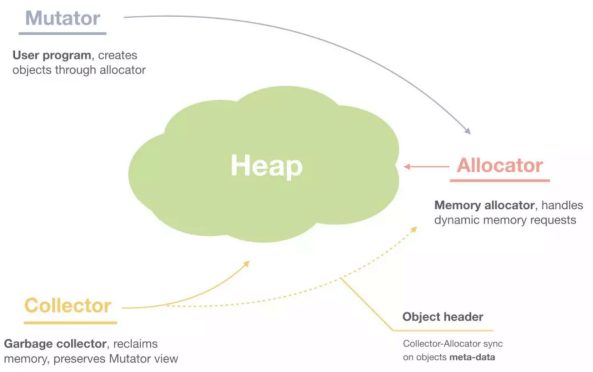
- 栈在高地址，从高地址向低地址增长。
- 堆在低地址，从低地址向高地址增长。

栈和堆相比优势：

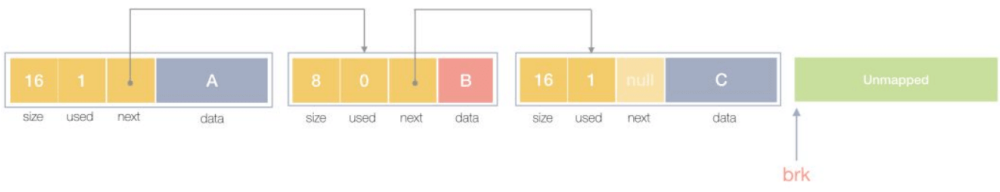
- 栈的内存管理简单，分配比堆上快。
- 栈的内存不需要回收，而堆需要，无论是主动free，还是被动的垃圾回收，这都需要花费额外的CPU。
- 栈上的内存有更好的局部性，堆上内存访问就不那么友好了，CPU访问的2块数据可能在不同的页上，CPU访问数据的时间可能就上去了。

33. 堆内存管理怎么分配的

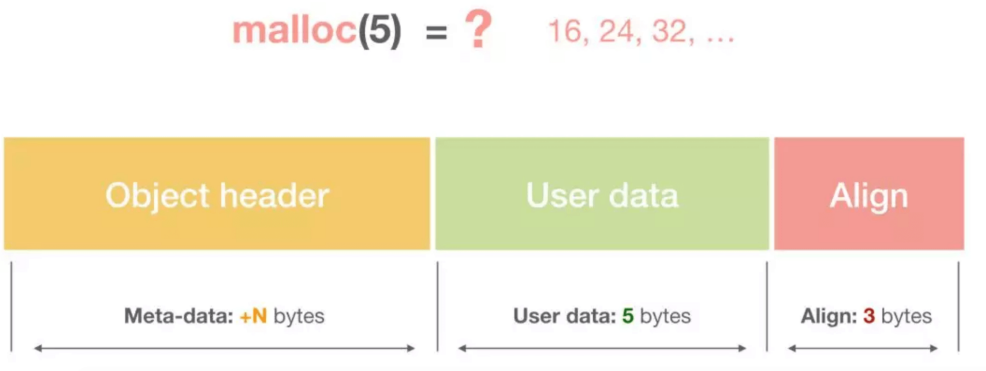
通常在Golang中,当我们谈论内存管理的时候，主要是指堆内存的管理，因为栈的内存管理不需要程序去操心。



堆内存管理中主要是三部分, 1.分配内存块，2.回收内存块, 3.组织内存块。



一个内存块包含了3类信息，如下图所示，元数据、用户数据和对齐字段，内存对齐是为了提高访问效率。下图申请5Byte内存的时候，就需要进行内存对齐。



释放内存实质是把使用的内存块从链表中取出来，然后标记为未使用，当分配内存块的时候，可以从未使用内存块中有先查找大小相近的内存块，如果找不到，再从未分配的内存中分配内存。

上面这个简单的设计中还没考虑内存碎片的问题，因为随着内存不断的申请和释放，内存上会存在大量的碎片，降低内存的使用率。为了解决内存碎片，可以将2个连续的未使用的内存块合并，减少碎片。

想要深入了解可以看下这个文章,《Writing a Memory Allocator》.

34. Go中的defer函数使用下面的两种情况下结果是什么

我们看看下面两种defer函数的返回的是什么:

```
    a := 1
    defer fmt.Println("the value of a1:",a)
    a++

    defer func() {
        fmt.Println("the value of a2:",a)
    }()
```

运行:

```
the value of a1: 1
the value of a1: 2
```

第一种情况：

```
defer fmt.Println("the value of a1:",a)
```

defer延迟函数调用的fmt.Println(a)函数的参数值在defer语句出现时就已经确定了，所以无论后面如何修改a变量都不会影响延迟函数。

第二种情况:

```
defer func() {
      fmt.Println("the value of a2:",a)
}()
```

defer延迟函数调用的函数参数的值在defer定义时候就确定了，而defer延迟函数内部所使用的值需要在这个函数运行时候才确定。

35. 在Go函数中为什么会发生内存泄露

通常内存泄漏，指的是能够预期的能很快被释放的内存由于附着在了长期存活的内存上、或生命期意外地被延长，导致预计能够立即回收的内存而长时间得不到回收。

在 Go 中，由于 goroutine 的存在，因此,内存泄漏除了附着在长期对象上之外，还存在多种不同的形式。

- 预期能被快速释放的内存因被根对象引用而没有得到迅速释放.

当有一个全局对象时，可能不经意间将某个变量附着在其上，且忽略的将其进行释放，则该内存永远不会得到释放。

- goroutine 泄漏

Goroutine 作为一种逻辑上理解的轻量级线程，需要维护执行用户代码的上下文信息。在运行过程中也需要消耗一定的内存来保存这类信息，而这些内存在目前版本的 Go 中是不会被释放的。

因此，如果一个程序持续不断地产生新的 goroutine、且不结束已经创建的 goroutine 并复用这部分内存，就会造成内存泄漏的现象.

例如:

```
func main() {
    for i := 0; i < 10000; i++ {
        go func() {
            select {}
        }()
    }
}
```

36. Go中new和make的区别

在Go中,的值类型和引用类型:

```
值类型：int，float，bool，string，struct和array.
变量直接存储值，分配栈区的内存空间，这些变量所占据的空间在函数被调用完后会自动释放。

引用类型：slice，map，chan和值类型对应的指针.
变量存储的是一个地址（或者理解为指针），指针指向内存中真正存储数据的首地址。内存通常在堆上分配，通过GC回收。
```

这里需要注意的是: 对于引用类型的变量，我们不仅要声明变量，更重要的是，我们得手动为它分配空间.

因此new该方法的参数要求传入一个类型，而不是一个值，它会申请一个该类型大小的内存空间，并会初始化为对应的零值，返回指向该内存空间的一个指针。

```
// The new built-in function allocates memory. The first argument is a type,
// not a value, and the value returned is a pointer to a newly
// allocated zero value of that type.
func new(Type) *Type
```

而make也是用于内存分配，但是和new不同，只用来引用对象slice、map和channel的内存创建，它返回的类型就是类型本身，而不是它们的指针类型。

```
// The make built-in function allocates and initializes an object of type
// slice, map, or chan (only). Like new, the first argument is a type, not a
// value. Unlike new, make's return type is the same as the type of its
// argument, not a pointer to it. The specification of the result depends on
// the type:
//    Slice: The size specifies the length. The capacity of the slice is
```

```
//    equal to its length. A second integer argument may be provided to
//    specify a different capacity; it must be no smaller than the
//    length. For example, make([]int, 0, 10) allocates an underlying array
//    of size 10 and returns a slice of length 0 and capacity 10 that is
//    backed by this underlying array.
//    Map: An empty map is allocated with enough space to hold the
//    specified number of elements. The size may be omitted, in which case
//    a small starting size is allocated.
//    Channel: The channel's buffer is initialized with the specified
//    buffer capacity. If zero, or the size is omitted, the channel is
//    unbuffered.
func make(t Type, size ...IntegerType) Type
```

37. G0的作用

在Go中 g0作为一个特殊的goroutine，为 scheduler 执行调度循环提供了场地（栈）。对于一个线程来说，g0 总是它第一个创建的 goroutine。

之后，它会不断地寻找其他普通的 goroutine 来执行，直到进程退出。

当需要执行一些任务，且不想扩栈时，就可以用到 g0 了，因为 g0 的栈比较大。

g0 其他的一些“职责”有：创建 `goroutine` 、 `deferproc` 函数里新建 `_defer` 、垃圾回收相关的工作（例如 stw、扫描 goroutine 的执行栈、一些标识清扫的工作、栈增长）等等。

38. Go中的锁如何实现

锁是一种同步机制，用于在多任务环境中限制资源的访问，以满足互斥需求。

go源码sync包中经常用于同步操作的方式:

- 原子操作
- 互斥锁
- 读写锁
- waitgroup

我们着重来分析下互斥锁和读写锁.

互斥锁:

下面是互斥锁的数据结构:

```
// A Mutex is a mutual exclusion lock.
// The zero value for a Mutex is an unlocked mutex.
//
// A Mutex must not be copied after first use.
type Mutex struct {
    state int32   // 互斥锁上锁状态枚举值如下所示
    sema  uint32  // 信号量，向处于Gwaitting的G发送信号
}
const (
    mutexLocked = 1 << iota // 值为1，表示在state中由低向高第1位，意义：锁是否可用,0可用，1不可用,锁定中
    mutexWoken   // 值为2，表示在state中由低向高第2位，意义：mutex是否被唤醒
    mutexStarving // 当前的互斥锁进入饥饿状态;
    mutexWaiterShift = iota  //值为2，表示state中统计阻塞在此mutex上goroutine的数目需要位移的偏移量
    starvationThresholdNs = 1e6
```

state和sema两个加起来只占 8 字节空间的结构体表示了 Go 语言中的互斥锁。

互斥锁的状态比较复杂，如下图所示，最低三位分别表示 mutexLocked、mutexWoken 和 mutexStarving，剩下的位置用来表示当前有多少个 Goroutine 等待互斥锁的释放.

在默认情况下，互斥锁的所有状态位都是 0，int32 中的不同位分别表示了不同的状态：

- mutexLocked 表示互斥锁的锁定状态；
- mutexWoken 表示从正常模式被从唤醒；
- mutexStarving 当前的互斥锁进入饥饿状态；
- waitersCount 当前互斥锁上等待的 Goroutine 个数；

sync.Mutex 有两种模式,正常模式和饥饿模式。

在正常模式下，锁的等待者会按照先进先出的顺序获取锁。

但是刚被唤起的 `Goroutine` 与新创建的 `Goroutine` 竞争时，大概率会获取不到锁，为了减少这种情况的出现，一旦 Goroutine 超过 1ms 没有获取到锁，它就会将当前互斥锁切换饥饿模式，防止部分 Goroutine 被饿死。

饥饿模式是在 Go 语言 1.9 版本引入的优化的，引入的目的是保证互斥锁的公平性（Fairness）。

在饥饿模式中，互斥锁会直接交给等待队列最前面的 Goroutine。新的 Goroutine 在该状态下不能获取锁、也不会进入自旋状态，它们只会在队列的末尾等待。

如果一个 Goroutine 获得了互斥锁并且它在队列的末尾或者它等待的时间少于 1ms，那么当前的互斥锁就会被切换回正常模式。

相比于饥饿模式，正常模式下的互斥锁能够提供更好地性能，饥饿模式的能避免 Goroutine 由于陷入等待无法获取锁而造成的高尾延时。

互斥锁的加锁是靠 sync.Mutex.Lock 方法完成的, 当锁的状态是 0 时，将 `mutexLocked` 位置成 1：

```
// Lock locks m.
// If the lock is already in use, the calling goroutine
// blocks until the mutex is available.
func (m *Mutex) Lock() {
    // Fast path: grab unlocked mutex.
    if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
        if race.Enabled {
            race.Acquire(unsafe.Pointer(m))
        }
        return
    }
    // Slow path (outlined so that the fast path can be inlined)
    m.lockSlow()
}
```

如果互斥锁的状态不是 0 时就会调用 `sync.Mutex.lockSlow` 尝试通过自旋（Spinnig）等方式等待锁的释放，

这个方法是一个非常大 for 循环,它获取锁的过程：

1. 判断当前 Goroutine 能否进入自旋；
2. 通过自旋等待互斥锁的释放；
3. 计算互斥锁的最新状态；
4. 更新互斥锁的状态并获取锁；

那么互斥锁是如何判断当前 Goroutine 能否进入自旋等互斥锁的释放,是通过它的lockSlow方法, 由于自旋是一种多线程同步机制，所以呢当前的进程在进入自旋的过程中会一直保持对 CPU 的占用，持续检查某个条件是否为真。
通常在多核的 CPU 上，自旋可以避免 Goroutine 的切换，使用得当会对性能带来很大的增益，但是往往使用的不得当就会拖慢整个程序.

所以 Goroutine 进入自旋的条件非常苛刻：

- 互斥锁只有在普通模式才能进入自旋；
- `runtime.sync_runtime_canSpin` 需要返回 true：

  a. 需要运行在多 CPU 的机器上；

  b. 当前的Goroutine 为了获取该锁进入自旋的次数小于四次；

  c. 当前机器上至少存在一个正在运行的处理器 P 并且处理的运行队列为空；

一旦当前 Goroutine 能够进入自旋就会调用 `runtime.sync_runtime_doSpin` 和 `runtime.procyield` 并执行 30 次的 PAUSE 指令，该指令只会占用 CPU 并消耗 CPU 时间.

处理了自旋相关的特殊逻辑之后，互斥锁会根据上下文计算当前互斥锁最新的状态。

通过几个不同的条件分别会更新 state 字段中存储的不同信息, `mutexLocked` 、 `mutexStarving` 、 `mutexWoken` 和 `mutexWaiterShift` ：

```
    new := old
    if old&mutexStarving == 0 {
        new |= mutexLocked
    }
    if old&(mutexLocked|mutexStarving) != 0 {
        new += 1 << mutexWaiterShift
    }
    if starving && old&mutexLocked != 0 {
        new |= mutexStarving
    }
    if awoke {
        new &^= mutexWoken
    }
```

计算了新的互斥锁状态之后，就会使用 CAS 函数 sync/atomic.CompareAndSwapInt32 更新该状态：

```
if atomic.CompareAndSwapInt32(&m.state, old, new) {
            if old&(mutexLocked|mutexStarving) == 0 {
                break // 通过 CAS 函数获取了锁
            }
            ...
            runtime_SemacquireMutex(&m.sema, queueLifo, 1)
            starving = starving || runtime_nanotime()-waitStartTime > starvationThresholdNs
            old = m.state
            if old&mutexStarving != 0 {
                delta := int32(mutexLocked - 1<<mutexWaiterShift)
                if !starving || old>>mutexWaiterShift == 1 {
                    delta -= mutexStarving
                }
                atomic.AddInt32(&m.state, delta)
                break
            }
            awoke = true
            iter = 0
        } else {
            old = m.state
```

```
        }
    }
}
```

如果我们没有通过 CAS 获得锁，会调用 `runtime.sync_runtime_SemacquireMutex` 使用信号量保证资源不会被两个 Goroutine 获取。

`runtime.sync_runtime_SemacquireMutex` 会在方法中不断调用尝试获取锁并休眠当前 Goroutine 等待信号量的释放，一旦当前 Goroutine 可以获取信号量，它就会立刻返回， `sync.Mutex.Lock` 方法的剩余代码也会继续执行。

在正常模式下，这段代码会设置唤醒和饥饿标记、重置迭代次数并重新执行获取锁的循环.
在饥饿模式下，当前 Goroutine 会获得互斥锁，如果等待队列中只存在当前 Goroutine，互斥锁还会从饥饿模式中退出.

互斥锁的解锁过程 `sync.Mutex.Unlock` 与加锁过程相比就很简单，该过程会先使用 `sync/atomic.AddInt32` 函数快速解锁，这时会发生下面的两种情况：

- 如果该函数返回的新状态等于 0，当前 Goroutine 就成功解锁了互斥锁；
- 如果该函数返回的新状态不等于 0，这段代码会调用 `sync.Mutex.unlockSlow` 方法开始慢速解锁：

```
func (m *Mutex) Unlock() {
    if race.Enabled {
        _ = m.state
        race.Release(unsafe.Pointer(m))
    }

    // Fast path: drop lock bit.
    new := atomic.AddInt32(&m.state, -mutexLocked)
    if new != 0 {
        // Outlined slow path to allow inlining the fast path.
        // To hide unlockSlow during tracing we skip one extra frame when tracing GoUnblock.
        m.unlockSlow(new)
    }
}
```

`sync.Mutex.unlockSlow` 方法首先会校验锁状态的合法性，如果当前互斥锁已经被解锁过了就会直接抛出异常 `sync: unlock of unlocked mutex` 中止当前程序。

在正常情况下会根据当前互斥锁的状态，分别处理正常模式和饥饿模式下的互斥锁.

```
func (m *Mutex) unlockSlow(new int32) {
    if (new+mutexLocked)&mutexLocked == 0 {
        throw("sync: unlock of unlocked mutex")
    }
    if new&mutexStarving == 0 {
        old := new
        for {
            // If there are no waiters or a goroutine has already
            // been woken or grabbed the lock, no need to wake anyone.
            // In starvation mode ownership is directly handed off from unlocking
            // goroutine to the next waiter. We are not part of this chain,
            // since we did not observe mutexStarving when we unlocked the mutex above.
            // So get off the way.
            if old>>mutexWaiterShift == 0 || old&(mutexLocked|mutexWoken|mutexStarving) != 0 {
                return
            }
            // Grab the right to wake someone.
            new = (old - 1<<mutexWaiterShift) | mutexWoken
            if atomic.CompareAndSwapInt32(&m.state, old, new) {
```

```
                runtime_Semrelease(&m.sema, false, 1)
                return
            }
            old = m.state
        }
    } else {
        // Starving mode: handoff mutex ownership to the next waiter, and yield
        // our time slice so that the next waiter can start to run immediately.
        // Note: mutexLocked is not set, the waiter will set it after wakeup.
        // But mutex is still considered locked if mutexStarving is set,
        // so new coming goroutines won't acquire it.
        runtime_Semrelease(&m.sema, true, 1)
    }
}
```

在正常模式下，这段代码会分别处理以下两种情况处理:

- 如果互斥锁不存在等待者或者互斥锁的 `mutexLocked` 、 `mutexStarving` 、 `mutexWoken` 状态不都为 0，那么当前方法就可以直接返回，不需要唤醒其他等待者；
- 如果互斥锁存在等待者，会通过 `sync.runtime_Semrelease` 唤醒等待者并移交锁的所有权；

在饥饿模式下，上述代码会直接调用 `sync.runtime_Semrelease` 方法将当前锁交给下一个正在尝试获取锁的等待者，等待者被唤醒后会得到锁，在这时互斥锁还不会退出饥饿状态；

互斥锁的加锁过程比较复杂，它涉及自旋、信号量以及调度等概念：

- 如果互斥锁处于初始化状态，就会直接通过置位 mutexLocked 加锁；
- 如果互斥锁处于 mutexLocked 并且在普通模式下工作，就会进入自旋，执行 30 次 PAUSE 指令消耗 CPU 时间等待锁的释放；
- 如果当前 Goroutine 等待锁的时间超过了 1ms，互斥锁就会切换到饥饿模式；
- 互斥锁在正常情况下会通过 `runtime.sync_runtime_SemacquireMutex` 函数将尝试获取锁的 Goroutine 切换至休眠状态，等待锁的持有者唤醒当前 Goroutine；
- 如果当前 Goroutine 是互斥锁上的最后一个等待的协程或者等待的时间小于 1ms，当前 Goroutine 会将互斥锁切换回正常模式；

互斥锁的解锁过程与之相比就比较简单，其代码行数不多、逻辑清晰，也比较容易理解：

- 当互斥锁已经被解锁时，那么调用 `sync.Mutex.Unlock` 会直接抛出异常；
- 当互斥锁处于饥饿模式时，会直接将锁的所有权交给队列中的下一个等待者，等待者会负责设置 `mutexLocked` 标志位；
- 当互斥锁处于普通模式时，如果没有 Goroutine 等待锁的释放或者已经有被唤醒的 Goroutine 获得了锁，就会直接返回；在其他情况下会通过 `sync.runtime_Semrelease` 唤醒对应的 Goroutine.

读写锁:

读写互斥锁 `sync.RWMutex` 是细粒度的互斥锁，它不限制资源的并发读，但是读写、写写操作无法并行执行。

sync.RWMutex 中总共包含5 个字段：

```
type RWMutex struct {
    w           Mutex  // 复用互斥锁提供的能力
    writerSem   uint32 // 写等待读
    readerSem   uint32 // 读等待写
```

```
    readerCount int32  // 存储了当前正在执行的读操作的数量
    readerWait  int32  // 当写操作被阻塞时等待的读操作个数
}
```

我们从写锁开始分析:

当我们想要获取写锁时，需要调用 `sync.RWMutex.Lock` 方法:

```
func (rw *RWMutex) Lock() {
    if race.Enabled {
        _ = rw.w.state
        race.Disable()
    }
    // First, resolve competition with other writers.
    rw.w.Lock()
    // Announce to readers there is a pending writer.
    r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxReaders) + rwmutexMaxReaders
    // Wait for active readers.
    if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0 {
        runtime_SemacquireMutex(&rw.writerSem, false, 0)
    }
    if race.Enabled {
        race.Enable()
        race.Acquire(unsafe.Pointer(&rw.readerSem))
        race.Acquire(unsafe.Pointer(&rw.writerSem))
    }
}
```

- 这里调用结构体持有的 `sync.Mutex` 的 `sync.Mutex.Lock` 方法阻塞后续的写操作;

因为互斥锁已经被获取，其他 Goroutine 在获取写锁时就会进入自旋或者休眠:

- 调用 `sync/atomic.AddInt32` 方法阻塞后续的读操作:

如果仍然有其他 Goroutine 持有互斥锁的读锁 `（r != 0）`，该 Goroutine 会调用 `runtime.sync_runtime_SemacquireMutex` 进入休眠状态等待所有读锁所有者执行结束后释放 `writerSem` 信号量将当前协程唤醒。

写锁的释放会调用 `sync.RWMutex.Unlock` 方法:

```
func (rw *RWMutex) Unlock() {
    if race.Enabled {
        _ = rw.w.state
        race.Release(unsafe.Pointer(&rw.readerSem))
        race.Disable()
    }

    // Announce to readers there is no active writer.
    r := atomic.AddInt32(&rw.readerCount, rwmutexMaxReaders)
    if r >= rwmutexMaxReaders {
        race.Enable()
        throw("sync: Unlock of unlocked RWMutex")
    }
    // Unblock blocked readers, if any.
    for i := 0; i < int(r); i++ {
        runtime_Semrelease(&rw.readerSem, false, 0)
    }
    // Allow other writers to proceed.
```

```
    rw.w.Unlock()
    if race.Enabled {
        race.Enable()
    }
}
```

解锁与加锁的过程正好相反,写锁的释放分为以下几个步骤:

1. 调用 `sync/atomic.AddInt32` 函数将 `readerCount` 变回正数，释放读锁;
2. 通过 for 循环触发所有由于获取读锁而陷入等待的 Goroutine;
3. 调用 `sync.Mutex.Unlock` 方法释放写锁;

获取写锁时会先阻塞写锁的获取，后阻塞读锁的获取，这种策略能够保证读操作不会被连续的写操作饿死。

接着是读锁:

读锁的加锁方法 `sync.RWMutex.RLock` 就比较简单了，该方法会通过 `sync/atomic.AddInt32` 将 `readerCount` 加一：

```
func (rw *RWMutex) RLock() {
    if race.Enabled {
        _ = rw.w.state
        race.Disable()
    }
    if atomic.AddInt32(&rw.readerCount, 1) < 0 {
        // A writer is pending, wait for it.
        runtime_SemacquireMutex(&rw.readerSem, false, 0)
    }
    if race.Enabled {
        race.Enable()
        race.Acquire(unsafe.Pointer(&rw.readerSem))
    }
}
```

如果 `RLock` 该方法返回负数,其他 Goroutine 获得了写锁，当前 Goroutine 就会调用 `runtime.sync_runtime_SemacquireMutex` 陷入休眠等待锁的释放;
如果 `RLock` 该方法的结果为非负数,没有 Goroutine 获得写锁，当前方法就会成功返回.

当 Goroutine 想要释放读锁时，会调用如下所示的 `RUnlock` 方法:

```
func (rw *RWMutex) RUnlock() {
    if race.Enabled {
        _ = rw.w.state
        race.ReleaseMerge(unsafe.Pointer(&rw.writerSem))
        race.Disable()
    }
    if r := atomic.AddInt32(&rw.readerCount, -1); r < 0 {
        // Outlined slow-path to allow the fast-path to be inlined
        rw.rUnlockSlow(r)
    }
    if race.Enabled {
        race.Enable()
    }
}
```

该方法会先减少正在读资源的 `readerCount` 整数，根据 `sync/atomic.AddInt32` 的返回值不同会分别进行处理:

- 如果返回值大于等于零,表示读锁直接解锁成功.

- 如果返回值小于零 ,表示有一个正在执行的写操作，在这时会调用 `rUnlockSlow` 方法.

```
func (rw *RWMutex) rUnlockSlow(r int32) {
    if r+1 == 0 || r+1 == -rwmutexMaxReaders {
        race.Enable()
        throw("sync: RUnlock of unlocked RWMutex")
    }
    // A writer is pending.
    if atomic.AddInt32(&rw.readerWait, -1) == 0 {
        // The last reader unblocks the writer.
        runtime_Semrelease(&rw.writerSem, false, 1)
    }
}
```

`rUnlockSlow` 该方法会减少获取锁的写操作等待的读操作数 `readerWait` 并在所有读操作都被释放之后触发写操作的信号量, `writerSem` ，该信号量被触发时，调度器就会唤醒尝试获取写锁的 Goroutine。

其实读写互斥锁(sync.RWMutex),虽然提供的功能非常复杂,不过因为它是在互斥锁(sync.Mutex)的基础上,所以整体的实现上会简单很多。

因此呢:

- 调用 `sync.RWMutex.Lock` 尝试获取写锁时；

每次 `sync.RWMutex.RUnlock` 都会将 `readerCount` 其减一，当它归零时该 Goroutine 就会获得写锁，将 `readerCount` 减少 `rwmutexMaxReaders` 个数以阻塞后续的读操作.

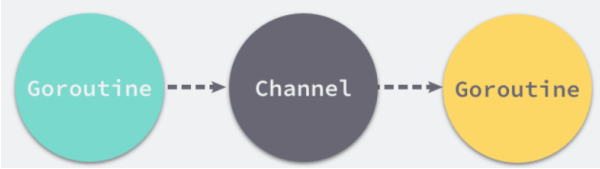
- 调用 `sync.RWMutex.Unlock` 释放写锁时，会先通知所有的读操作，然后才会释放持有的互斥锁；

读写互斥锁在互斥锁之上提供了额外的更细粒度的控制，能够在读操作远远多于写操作时提升性能。

39. Go中的channel的实现

在Go中最常见的就是通信顺序进程（Communicating sequential processes，CSP）的并发模型,通过共享通信,来实现共享内存,这里就提到了channel.

Goroutine 和 Channel 分别对应 CSP 中的实体和传递信息的媒介，Go 语言中的 Goroutine 会通过 Channel 传递数据。



Goroutine通过使用channel传递数据，一个会向 Channel 中发送数据，另一个会从 Channel 中接收数据，它们两者能够独立运行并不存在直接关联，但是能通过 Channel 间接完成通信。

Channel 收发操作均遵循了先入先出（FIFO）的设计，具体规则如下：

- 先从 Channel 读取数据的 Goroutine 会先接收到数据；
- 先向 Channel 发送数据的 Goroutine 会得到先发送数据的权利；

Channel 通常会有以下三种类型：

- 同步 Channel — 不需要缓冲区，发送方会直接将数据交给（Handoff）接收方；
- 异步 Channel — 基于环形缓存的传统生产者消费者模型；

- `chan struct{}` 类型的异步 `Channel` 的 `struct{}` 类型不占用内存空间，不需要实现缓冲区和直接发送（Handoff）的语义；

Channel 在运行时使用 `runtime.hchan` 结构体表示:

```
type hchan struct {
    qcount   uint           // 当前队列里还剩余元素个数
    dataqsiz uint           // 环形队列长度，即缓冲区的大小，即make(chan T,N) 中的N
    buf      unsafe.Pointer // 环形队列指针
    elemsize uint16         // 每个元素的大小
    closed   uint32         // 标识当前通道是否处于关闭状态，创建通道后，该字段设置0，即打开通道；通道调用close将其设置为1，通道关闭
    elemtype *_type         // 元素类型，用于数据传递过程中的赋值
    sendx    uint           // 环形缓冲区的状态字段，它只是缓冲区的当前索引-支持数组，它可以从中发送数据
    recvx    uint           // 环形缓冲区的状态字段，它只是缓冲区当前索引-支持数组，它可以从中接受数据
    recvq    waitq          // 等待读消息的goroutine队列
    sendq    waitq          // 等待写消息的goroutine队列

    // lock protects all fields in hchan, as well as several
    // fields in sudogs blocked on this channel.
    //
    // Do not change another G's status while holding this lock
    // (in particular, do not ready a G), as this can deadlock
    // with stack shrinking.
    lock mutex              // 互斥锁，为每个读写操作锁定通道，因为发送和接受必须是互斥操作
}

type waitq struct {
    first *sudog
    last  *sudog
}
```

其中hchan结构体中有五个字段是构建底层的循环队列:

```
* qcount — Channel 中的元素个数；
* dataqsiz — Channel 中的循环队列的长度；
* buf — Channel 的缓冲区数据指针；
* sendx — Channel 的发送操作处理到的位置；
* recvx — Channel 的接收操作处理到的位置；
```

通常，`elemsize` 和 `elemtype` 分别表示当前 Channel 能够收发的元素类型和大小.

`sendq` 和 `recvq` 存储了当前 Channel 由于缓冲区空间不足而阻塞的 Goroutine 列表，这些等待队列使用双向链表 `runtime.waitq` 表示，链表中所有的元素都是 `runtime.sudog` 结构.

`waitq` 表示一个在等待列表中的 Goroutine，该结构体中存储了阻塞的相关信息以及两个分别指向前后 `runtime.sudog` 的指针。

channel 在Go中是通过make关键字创建,编译器会将make(chan int,10).

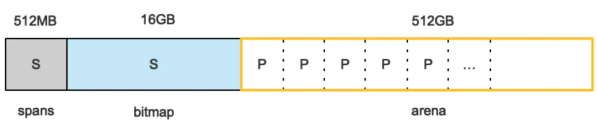
创建管道:

`runtime.makechan` 和 `runtime.makechan64` 会根据传入的参数类型和缓冲区大小创建一个新的 Channel 结构，其中后者用于处理缓冲区大小大于 2 的 32 次方的情况.

这里我们来详细看下 `makechan` 函数:

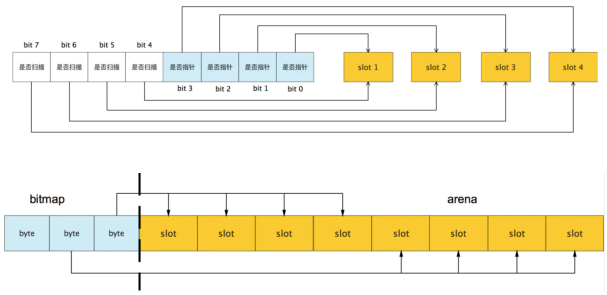
```
func makechan(t *chantype, size int) *hchan {
    elem := t.elem

    // compiler checks this but be safe.
    if elem.size >= 1<<16 {
        throw("makechan: invalid channel element type")
    }
    if hchanSize%maxAlign != 0 || elem.align > maxAlign {
        throw("makechan: bad alignment")
    }

    mem, overflow := math.MulUintptr(elem.size, uintptr(size))
    if overflow || mem > maxAlloc-hchanSize || size < 0 {
        panic(plainError("makechan: size out of range"))
    }

    // Hchan does not contain pointers interesting for GC when elements stored in buf do not contain pointer
s.
    // buf points into the same allocation, elemtype is persistent.
    // SudoG's are referenced from their owning thread so they can't be collected.
    // TODO(dvyukov,rlh): Rethink when collector can move allocated objects.
    var c *hchan
    switch {
    case mem == 0:
        // Queue or element size is zero.
        c = (*hchan)(mallocgc(hchanSize, nil, true))
        // Race detector uses this location for synchronization.
        c.buf = c.raceaddr()
    case elem.ptrdata == 0:
        // Elements do not contain pointers.
        // Allocate hchan and buf in one call.
        c = (*hchan)(mallocgc(hchanSize+mem, nil, true))
        c.buf = add(unsafe.Pointer(c), hchanSize)
    default:
        // Elements contain pointers.
        c = new(hchan)
        c.buf = mallocgc(mem, elem, true)
    }

    c.elemsize = uint16(elem.size)
    c.elemtype = elem
    c.dataqsiz = uint(size)
    lockInit(&c.lock, lockRankHchan)

    if debugChan {
        print("makechan: chan=", c, "; elemsize=", elem.size, "; dataqsiz=", size, "\n")
    }
    return c
}
```

Channel 中根据收发元素的类型和缓冲区的大小初始化 `runtime.hchan` 结构体和缓冲区：



arena区域就是我们所谓的堆区，Go动态分配的内存都是在这个区域，它把内存分割成8KB大小的页，一些页组合起来称为mspan。

bitmap区域标识arena区域哪些地址保存了对象，并且用4bit标志位表示对象是否包含指针、GC标记信息。bitmap中一个byte大小的内存对应arena区域中4个指针大小（指针大小为 8B ）的内存，所以bitmap区域的大小是512GB/(4*8B)=16GB。



此外我们还可以看到bitmap的高地址部分指向arena区域的低地址部分，这里bitmap的地址是由高地址向低地址增长的。

spans区域存放mspan（是一些arena分割的页组合起来的内存管理基本单元，后文会再讲）的指针，每个指针对应一页，所以spans区域的大小就是512GB/8KB*8B=512MB。

除以8KB是计算arena区域的页数，而最后乘以8是计算spans区域所有指针的大小。创建mspan的时候，按页填充对应的spans区域，在回收object时，根据地址很容易就能找到它所属的mspan。

32. 栈的内存是怎么分配的

栈和堆只是虚拟内存上2块不同功能的内存区域：

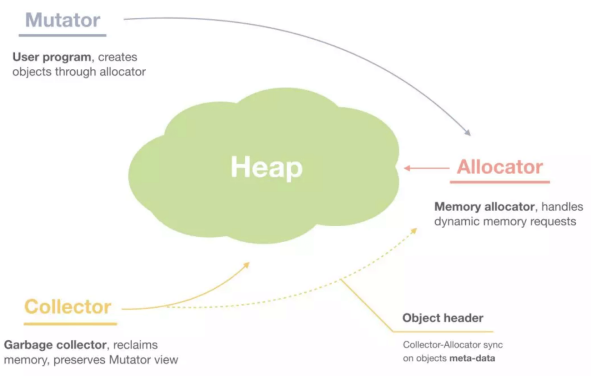
- 栈在高地址，从高地址向低地址增长。
- 堆在低地址，从低地址向高地址增长。

栈和堆相比优势：

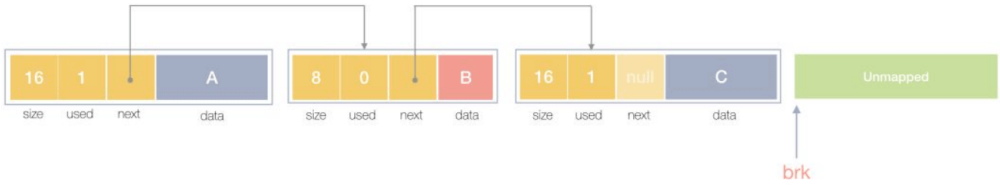
- 栈的内存管理简单，分配比堆上快。
- 栈的内存不需要回收，而堆需要，无论是主动free，还是被动的垃圾回收，这都需要花费额外的CPU。
- 栈上的内存有更好的局部性，堆上内存访问就不那么友好了，CPU访问的2块数据可能在不同的页上，CPU访问数据的时间可能就上去了。

33. 堆内存管理怎么分配的

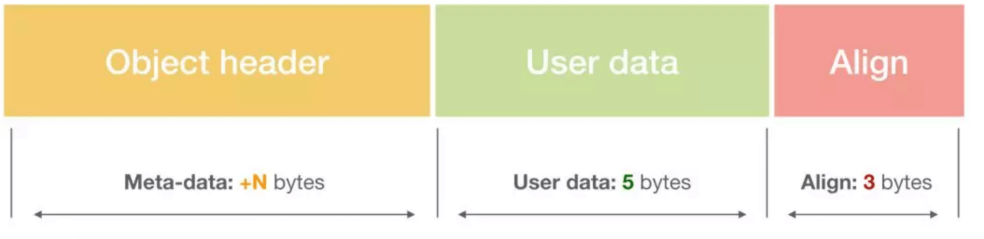
通常在Golang中,当我们谈论内存管理的时候，主要是指堆内存的管理，因为栈的内存管理不需要程序去操心。



堆内存管理中主要是三部分，1.分配内存块，2.回收内存块，3.组织内存块。



一个内存块包含了3类信息，如下图所示，元数据、用户数据和对齐字段，内存对齐是为了提高访问效率。下图申请5Byte内存的时候，就需要进行内存对齐。



释放内存实质是把使用的内存块从链表中取出来，然后标记为未使用，当分配内存块的时候，可以从未使用内存块中有先查找大小相近的内存块，如果找不到，再从未分配的内存中分配内存。

上面这个简单的设计中还没考虑内存碎片的问题，因为随着内存不断的申请和释放，内存上会存在大量的碎片，降低内存的使用率。为了解决内存碎片，可以将2个连续的未使用的内存块合并，减少碎片。

想要深入了解可以看下这个文章,《Writing a Memory Allocator》.

34. Go中的defer函数使用下面的两种情况下结果是什么

我们看看下面两种defer函数的返回的是什么:

```
    a := 1
    defer fmt.Println("the value of a1:",a)
    a++

    defer func() {
        fmt.Println("the value of a2:",a)
    }()
```

运行:

```
the value of a1: 1
the value of a1: 2
```

第一种情况：

```
defer fmt.Println("the value of a1:",a)
```

defer延迟函数调用的fmt.Println(a)函数的参数值在defer语句出现时就已经确定了，所以无论后面如何修改a变量都不会影响延迟函数。

第二种情况:

```
defer func() {
        fmt.Println("the value of a2:",a)
    }()
```

defer延迟函数调用的函数参数的值在defer定义时候就确定了，而defer延迟函数内部所使用的值需要在这个函数运行时候才确定。

35. 在Go函数中为什么会发生内存泄露

通常内存泄漏，指的是能够预期的能很快被释放的内存由于附着在了长期存活的内存上、或生命期意外地被延长，导致预计能够立即回收的内存而长时间得不到回收。

在 Go 中，由于 goroutine 的存在，因此,内存泄漏除了附着在长期对象上之外，还存在多种不同的形式。

- 预期能被快速释放的内存因被根对象引用而没有得到迅速释放.

当有一个全局对象时，可能不经意间将某个变量附着在其上，且忽略的将其进行释放，则该内存永远不会得到释放。

- goroutine 泄漏

Goroutine 作为一种逻辑上理解的轻量级线程，需要维护执行用户代码的上下文信息。在运行过程中也需要消耗一定的内存来保存这类信息，而这些内存在目前版本的 Go 中是不会被释放的。

因此，如果一个程序持续不断地产生新的 goroutine、且不结束已经创建的 goroutine 并复用这部分内存，就会造成内存泄漏的现象.

例如:

```
func main() {
    for i := 0; i < 10000; i++ {
        go func() {
            select {}
        }()
    }
}
```

36. Go中new和make的区别

在Go中,的值类型和引用类型:

```
值类型：int，float，bool，string，struct和array.
变量直接存储值，分配栈区的内存空间，这些变量所占据的空间在函数被调用完后会自动释放。

引用类型：slice，map，chan和值类型对应的指针.
变量存储的是一个地址（或者理解为指针），指针指向内存中真正存储数据的首地址。内存通常在堆上分配，通过GC回收。
```

这里需要注意的是: 对于引用类型的变量，我们不仅要声明变量，更重要的是，我们得手动为它分配空间.

因此new该方法的参数要求传入一个类型，而不是一个值，它会申请一个该类型大小的内存空间，并会初始化为对应的零值，返回指向该内存空间的一个指针。

```
// The new built-in function allocates memory. The first argument is a type,
// not a value, and the value returned is a pointer to a newly
// allocated zero value of that type.
func new(Type) *Type
```

而make也是用于内存分配，但是和new不同，只用来引用对象slice、map和channel的内存创建，它返回的类型就是类型本身，而不是它们的指针类型。

```
// The make built-in function allocates and initializes an object of type
// slice, map, or chan (only). Like new, the first argument is a type, not a
// value. Unlike new, make's return type is the same as the type of its
// argument, not a pointer to it. The specification of the result depends on
// the type:
//    Slice: The size specifies the length. The capacity of the slice is
//    equal to its length. A second integer argument may be provided to
//    specify a different capacity; it must be no smaller than the
//    length. For example, make([]int, 0, 10) allocates an underlying array
//    of size 10 and returns a slice of length 0 and capacity 10 that is
//    backed by this underlying array.
//    Map: An empty map is allocated with enough space to hold the
//    specified number of elements. The size may be omitted, in which case
//    a small starting size is allocated.
//    Channel: The channel's buffer is initialized with the specified
//    buffer capacity. If zero, or the size is omitted, the channel is
//    unbuffered.
func make(t Type, size ...IntegerType) Type
```

- 如果当前 Channel 中不存在缓冲区，那么就只会为 `hchan` 分配一段内存空间.
- 如果当前 Channel 中存储的类型不是指针类型，就会为当前的 Channel 和底层的数组分配一块连续的内存空间.
- 在默认情况下会单独为 `hchan` 和缓冲区分配内存.

发送数据:

当我们想要向 Channel 发送数据时，就需要使用 ch <- i 语句.

`runtime.chansend1` 调用了 `runtime.chansend` 并传入 Channel 和需要发送的数据。

`runtime.chansend` 是向 Channel 中发送数据时最终会调用的函数，这个函数负责了发送数据的全部逻辑，如果我们在调用时将 block 参数设置成 true，那么就表示当前发送操作是一个阻塞操作：

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr) bool {
    ...
    if !block && c.closed == 0 && full(c) {
        return false
    }

    var t0 int64
    if blockprofilerate > 0 {
        t0 = cputicks()
    }

    lock(&c.lock)

    if c.closed != 0 {
        unlock(&c.lock)
        panic(plainError("send on closed channel"))
    }
```

```
    if sg := c.recvq.dequeue(); sg != nil {
        // Found a waiting receiver. We pass the value we want to send
        // directly to the receiver, bypassing the channel buffer (if any).
        send(c, sg, ep, func() { unlock(&c.lock) }, 3)
        return true
    }
    ...
}
```

在发送数据的逻辑执行之前会先为当前 Channel 加锁，防止发生竞争条件。如果 Channel 已经关闭，那么向该 Channel 发送数据时就会报"send on closed channel" 错误并中止程序。

因为 `runtime.chansend` 函数的实现比较复杂，所以我们这里将该函数的执行过程分成以下的三个部分：

- 当存在等待的接收者时，通过 runtime.send 直接将数据发送给阻塞的接收者.
- 当缓冲区存在空余空间时，将发送的数据写入 Channel 的缓冲区.
- 当不存在缓冲区或者缓冲区已满时，等待其他 Goroutine 从 Channel 接收数据.

因此:

当我们使用 `ch <- i` 表达式向 Channel 发送数据时遇到的几种情况：

- 如果当前 Channel 的 `recvq` 上存在已经被阻塞的 Goroutine，那么会直接将数据发送给当前的 Goroutine 并将其设置成下一个运行的 Goroutine；
- 如果 Channel 存在缓冲区并且其中还有空闲的容量，我们就会直接将数据直接存储到当前缓冲区 sendx 所在的位置上；
- 如果不满足上面的两种情况，就会创建一个 `runtime.sudog` 结构并将其加入 Channel 的 `sendq` 队列中，当前 Goroutine 也会陷入阻塞等待其他的协程从 Channel 接收数据；

发送数据的过程中可能包含几个会触发 Goroutine 调度的时机：

1. 发送数据时发现 Channel 上存在等待接收数据的 Goroutine，立刻设置处理器的 `runnext` 属性，但是并不会立刻触发调度.
2. 发送数据时并没有找到接收方并且缓冲区已经满了，这时就会将自己加入 Channel 的 `sendq` 队列并调用 `runtime.goparkunlock` 触发 Goroutine 的调度让出处理器的使用权.

接收数据:

接着我们看看接受数据,Go中可以使用两种不同的方式去接收 Channel 中的数据：

```
* i <- ch
* i, ok <- ch
```

虽然不同的接收方式会被转换成 `runtime.chanrecv1` 和 `runtime.chanrecv2` 两种不同函数的调用，但是这两个函数最终还是会调用 `runtime.chanrecv` 。

当我们从一个空 Channel 接收数据时会直接调用 `runtime.gopark` 直接让出处理器的使用权。

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received bool) {
    ...
    if c == nil {
        if !block {
            return
        }
        gopark(nil, nil, waitReasonChanReceiveNilChan, traceEvGoStop, 2)
```

```
        throw("unreachable")
    }

    lock(&c.lock)

    if c.closed != 0 && c.qcount == 0 {
        if raceenabled {
            raceacquire(c.raceaddr())
        }
        unlock(&c.lock)
        if ep != nil {
            typedmemclr(c.elemtype, ep)
        }
        return true, false
    }
    ...
}
```

如果当前 Channel 已经被关闭并且缓冲区中不存在任何的数据，那么就会清除 `ep` 指针中的数据并立刻返回。

除了上述两种特殊情况，使用 `runtime.chanrecv` 从 Channel 接收数据时还包含以下三种不同情况：

- 当存在等待的发送者时，通过 `runtime.recv` 直接从阻塞的发送者或者缓冲区中获取数据.
- 当缓冲区存在数据时，从 Channel 的缓冲区中接收数据.
- 当缓冲区中不存在数据时，等待其他 Goroutine 向 Channel 发送数据.

因此接受数据的时候,Channel 中接收数据时可能会发生的五种情况：

1. 如果 Channel 为空，那么就会直接调用 `runtime.gopark` 挂起当前 Goroutine；
2. 如果 Channel 已经关闭并且缓冲区没有任何数据，`runtime.chanrecv` 函数会直接返回；
3. 如果 Channel 的 sendq 队列中存在挂起的 Goroutine，就会将 `recvx` 索引所在的数据拷贝到接收变量所在的内存空间上并将 `sendq` 队列中 Goroutine 的数据拷贝到缓冲区；
4. 如果 Channel 的缓冲区中包含数据就会直接读取 recvx 索引对应的数据；
5. 在默认情况下会挂起当前的 Goroutine，将 `runtime.sudog` 结构加入 `recvq` 队列并陷入休眠等待调度器的唤醒；

从 Channel 接收数据时，会触发 Goroutine 调度的两个时机：

- 当 Channel 为空时；
- 当缓冲区中不存在数据并且也不存在数据的发送者时；

最后就是关闭管道:

编译器会将用于关闭管道的 close 关键字调用 `runtime.closechan` 的函数关闭。

当 Channel 是一个空指针或者已经被关闭时，Go 语言运行时都会直接 panic 并抛出异常,处理完了这些异常的情况之后就可以开始执行关闭 Channel 的逻辑.

40. Go中的map的实现

Go中Map是一个KV对集合。底层使用 `hash table` ，用链表来解决冲突 ，出现冲突时，不是每一个Key都申请一个结构通过链表串起来，而是以bmap为最小粒度挂载，一个bmap可以放8个kv。

在哈希函数的选择上，会在程序启动时，检测 cpu 是否支持 `aes` ，如果支持，则使用 `aes hash` ，否则使用 `memhash` 。

hash函数,有加密型和非加密型。加密型的一般用于加密数据、数字摘要等，典型代表就是md5、sha1、sha256、aes256 这种,非加密型的一般就是查找。

在map的应用场景中，用的是查找。

选择hash函数主要考察的是两点：性能、碰撞概率。

每个map的底层结构是hmap，是有若干个结构为bmap的bucket组成的数组。每个bucket底层都采用链表结构。

```
type hmap struct {
    count     int    // 元素个数
    flags     uint8  // 用来标记状态
    B         uint8  // 扩容常量相关字段B是buckets数组的长度的对数 2^B
    noverflow uint16 // noverflow是溢出桶的数量，当B<16时，为精确值,当B>=16时，为估计值
    hash0     uint32 // 是哈希的种子，它能为哈希函数的结果引入随机性，这个值在创建哈希表时确定，并在调用哈希函数时作为参数传入

    buckets    unsafe.Pointer // 桶的地址
    oldbuckets unsafe.Pointer // 旧桶的地址，用于扩容
    nevacuate  uintptr        // 搬迁进度，扩容需要将旧数据搬迁至新数据，这里是利用指针来比较判断有没有迁移

    extra *mapextra // 用于扩容的指针
}

type mapextra struct {
    overflow    *[]*bmap
    oldoverflow *[]*bmap
    // nextOverflow holds a pointer to a free overflow bucket.
    nextOverflow *bmap
}

// A bucket for a Go map.
type bmap struct {
    tophash [bucketCnt]uint8  // tophash用于记录8个key哈希值的高8位，这样在寻找对应key的时候可以更快，不必每次都对key做全等判断
}

//实际上编辑期间会动态生成一个新的结构体
type bmap struct {
    topbits  [8]uint8
    keys     [8]keytype
    values   [8]valuetype
    pad      uintptr
    overflow uintptr
}
```

bmap 就是我们常说的“桶”，桶里面会最多装 8 个 key，这些 key之所以会落入同一个桶，是因为它们经过哈希计算后，哈希结果是“一类”的，关于key的定位我们在map的查询和赋值中详细说明。

在桶内，又会根据key计算出来的hash值的高8位来决定 key到底落入桶内的哪个位置（一个桶内最多有8个位置)。

当map的key和value都不是指针，并且 size都小于128字节的情况下，会把bmap标记为不含指针，这样可以避免gc时扫描整个hmap。

但是，我们看bmap其实有一个overflow的字段，是指针类型的，破坏了 bmap 不含指针的设想，这时会把overflow移动到hmap的extra 字段来。

这样随着哈希表存储的数据逐渐增多，我们会扩容哈希表或者使用额外的桶存储溢出的数据，不会让单个桶中的数据超过 8 个，不过溢出桶只是临时的解决方案，创建过多的溢出桶最终也会导致哈希的扩容。

哈希表作为一种数据结构，我们肯定要分析它的常见操作，首先就是读写操作的原理。哈希表的访问一般都是通过下标或者遍历进行的：

```
_ = hash[key]

for k, v := range hash {
    // k, v
}
```

这两种方式虽然都能读取哈希表的数据，但是使用的函数和底层原理完全不同。

第一个需要知道哈希的键并且一次只能获取单个键对应的值，而第二个可以遍历哈希中的全部键值对，访问数据时也不需要预先知道哈希的键。

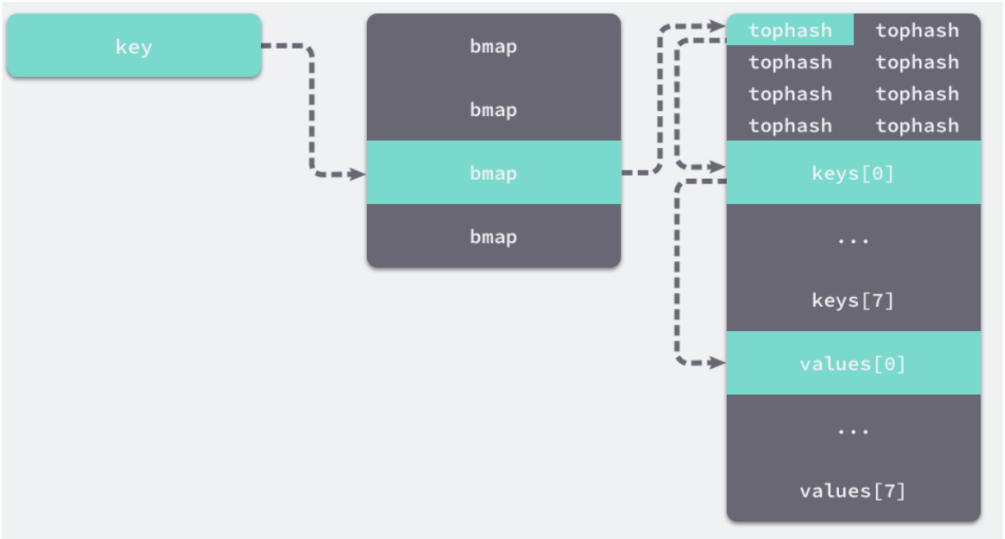
在编译的类型检查期间，`hash[key]` 以及类似的操作都会被转换成哈希的 `OINDEXMAP` 操作，中间代码生成阶段会在 `cmd/compile/internal/gc.walkexpr` 函数中将这些 `OINDEXMAP` 操作转换成如下的代码：

```
v := hash[key] // => v     := *mapaccess1(maptype, hash, &key)
v, ok := hash[key] // => v, ok := mapaccess2(maptype, hash, &key)
```

这里根据赋值语句左侧接受参数的个数会决定使用的运行时方法：

当接受一个参数时，会使用 `runtime.mapaccess1`，该函数仅会返回一个指向目标值的指针；
当接受两个参数时，会使用 `runtime.mapaccess2`，除了返回目标值之外，它还会返回一个用于表示当前键对应的值是否存在的 bool 值：

`mapaccess1` 会先通过哈希表设置的哈希函数、种子获取当前键对应的哈希，再通过 `runtime.bucketMask` 和 `runtime.add` 拿到该键值对所在的桶序号和哈希高位的 8 位数字。



如果在bucket中没有找到，此时如果overflow不为空，那么就沿着overflow继续查找，如果还是没有找到，那就从别的key槽位查找，直到遍历所有bucket。

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    if raceenabled && h != nil {
        callerpc := getcallerpc()
        pc := funcPC(mapaccess1)
```

```
        racereadpc(unsafe.Pointer(h), callerpc, pc)
        raceReadObjectPC(t.key, key, callerpc, pc)
    }
    if msanenabled && h != nil {
        msanread(key, t.key.size)
    }
    //如果h说明都没有，返回零值
    if h == nil || h.count == 0 {
        if t.hashMightPanic() { //如果哈希函数出错
            t.key.alg.hash(key, 0) // see issue 23734
        }
        return unsafe.Pointer(&zeroVal[0])
    }
    //写和读冲突
    if h.flags&hashWriting != 0 {
        throw("concurrent map read and map write")
    }
    //不同类型的key需要不同的hash算法需要在编译期间确定
    alg := t.key.alg
    //利用hash0引入随机性，计算哈希值
    hash := alg.hash(key, uintptr(h.hash0))
    //比如B=5那m就是31二进制是全1，
    //求bucket num时，将hash与m相与，
    //达到bucket num由hash的低8位决定的效果，
    //bucketMask函数掩蔽了移位量，省略了溢出检查。
    m := bucketMask(h.B)
    //b即bucket的地址
    b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
    // oldbuckets 不为 nil，说明发生了扩容
    if c := h.oldbuckets; c != nil {
        if !h.sameSizeGrow() {
            //新的bucket是旧的bucket两倍
            m >>= 1
        }
        //求出key在旧的bucket中的位置
        oldb := (*bmap)(add(c, (hash&m)*uintptr(t.bucketsize)))
        //如果旧的bucket还没有搬迁到新的bucket中，那就在老的bucket中寻找
        if !evacuated(oldb) {
            b = oldb
        }
    }
    //计算tophash高8位
    top := tophash(hash)
bucketloop:
    //遍历所有overflow里面的bucket
    for ; b != nil; b = b.overflow(t) {
        //遍历8个bucket
        for i := uintptr(0); i < bucketCnt; i++ {
            //tophash不匹配，继续
            if b.tophash[i] != top {
                if b.tophash[i] == emptyRest {
                    break bucketloop
                }
                continue
            }
            //tophash匹配，定位到key的位置
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            //若key为指针
            if t.indirectkey() {
                //解引用
                k = *((*unsafe.Pointer)(k))
            }
```

```
            //key相等
            if alg.equal(key, k) {
                //定位value的位置
                e := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                if t.indirectelem() {
                    //value解引用
                    e = *((*unsafe.Pointer)(e))
                }
                return e
            }
        }
    }
    //没有找到，返回0值
    return unsafe.Pointer(&zeroVal[0])
}
```

在 bucketloop 循环中，哈希会依次遍历正常桶和溢出桶中的数据，它先会比较哈希的高 8 位和桶中存储的 tophash，后比较传入的和桶中的值以加速数据的读写。用于选择桶序号的是哈希的最低几位，而用于加速访问的是哈希的高 8 位，这种设计能够减少同一个桶中有大量相等 tophash 的概率影响性能。

因此bucket里key的起始地址就是 `unsafe.Pointer(b)+dataOffset` ；第i个key的地址就要此基础上加i个key大小；value的地址是在key之后，所以第i个value，要加上所有的key的偏移。

另一个同样用于访问哈希表中数据的 `runtime.mapaccess2` 只是在 `runtime.mapaccess1` 的基础上多返回了一个标识键值对是否存在的 bool 值：

```
func mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool) {
    ...
bucketloop:
    for ; b != nil; b = b.overflow(t) {
        for i := uintptr(0); i < bucketCnt; i++ {
            if b.tophash[i] != top {
                if b.tophash[i] == emptyRest {
                    break bucketloop
                }
                continue
            }
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            if t.indirectkey() {
                k = *((*unsafe.Pointer)(k))
            }
            if t.key.equal(key, k) {
                e := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.elemsize))
                if t.indirectelem() {
                    e = *((*unsafe.Pointer)(e))
                }
                return e, true
            }
        }
    }
    return unsafe.Pointer(&zeroVal[0]), false
}
```

使用 `v, ok := hash[k]` 的形式访问哈希表中元素时，我们能够通过这个布尔值更准确地知道当 `v == nil` 时，v 到底是哈希中存储的元素还是表示该键对应的元素不存在，所以在访问哈希时，我们更推荐使用这种方式判断元素是否存在。

写入:

当形如 `hash[k]` 的表达式出现在赋值符号左侧时，该表达式也会在编译期间转换成 `mapassign` 函数的调用，该函数与 `mapaccess1` 比较相似:

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    ...
      hash := t.hasher(key, uintptr(h.hash0))

      // Set hashWriting after calling t.hasher, since t.hasher may panic,
      // in which case we have not actually done a write.
      h.flags ^= hashWriting

      if h.buckets == nil {
          h.buckets = newobject(t.bucket) // newarray(t.bucket, 1)
      }

  again:
      bucket := hash & bucketMask(h.B)
      if h.growing() {
          growWork(t, h, bucket)
      }
      b := (*bmap)(unsafe.Pointer(uintptr(h.buckets) + bucket*uintptr(t.bucketsize)))
      top := tophash(hash)
    ...
}
```

我们可以通过遍历比较桶中存储的 `tophash` 和键的哈希，如果找到了相同结果就会返回目标位置的地址。

如果当前桶已经满了，哈希会调用 `newoverflow` 创建新桶或者使用 `hmap` 预先在 `noverflow` 中创建好的桶来保存数据，新创建的桶不仅会被追加到已有桶的末尾，还会增加哈希表的 `noverflow` 计数器。

如果当前键值对在哈希中不存在，哈希会为新键值对规划存储的内存地址，通过 `typedmemmove` 将键移动到对应的内存空间中并返回键对应值的地址 val。

如果当前键值对在哈希中存在，那么就会直接返回目标区域的内存地址，哈希并不会在 `mapassign` 这个运行时函数中将值拷贝到桶中，该函数只会返回内存地址，真正的赋值操作是在编译期间插入的.

扩容:

随着哈希表中元素的逐渐增加，哈希的性能会逐渐恶化，所以我们需要更多的桶和更大的内存保证哈希的读写性能,这个时候我们就需要用到扩容了.

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    ...
      // Did not find mapping for key. Allocate new cell & add entry.

      // If we hit the max load factor or we have too many overflow buckets,
      // and we're not already in the middle of growing, start growing.
      if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
          hashGrow(t, h)
          goto again // Growing the table invalidates everything, so try again
      }
   ...
}

// 装载因子超过 6.5
func overLoadFactor(count int64, B uint8) bool {
    return count >= bucketCnt && float32(count) >= loadFactor*float32((uint64(1)<<B))
}
```

```
// overflow buckets
func tooManyOverflowBuckets(noverflow uint16, B uint8) bool {
    if B < 16 {
        return noverflow >= uint16(1)<<B
    }
    return noverflow >= 1<<15
}
```

`mapassign` 函数会在以下两种情况发生时触发哈希的扩容：

- 装载因子已经超过 6.5；
- 哈希使用了太多溢出桶；

不过因为 Go 语言哈希的扩容不是一个原子的过程，所以 `mapassign` 还需要判断当前哈希是否已经处于扩容状态，避免二次扩容造成混乱。

根据触发的条件不同扩容的方式分成两种，如果这次扩容是溢出的桶太多导致的，那么这次扩容就是等量扩容 `sameSizeGrow` ，`sameSizeGrow` 是一种特殊情况下发生的扩容，当我们持续向哈希中插入数据并将它们全部删除时，如果哈希表中的数据量没有超过阈值，就会不断积累溢出桶造成缓慢的内存泄漏。

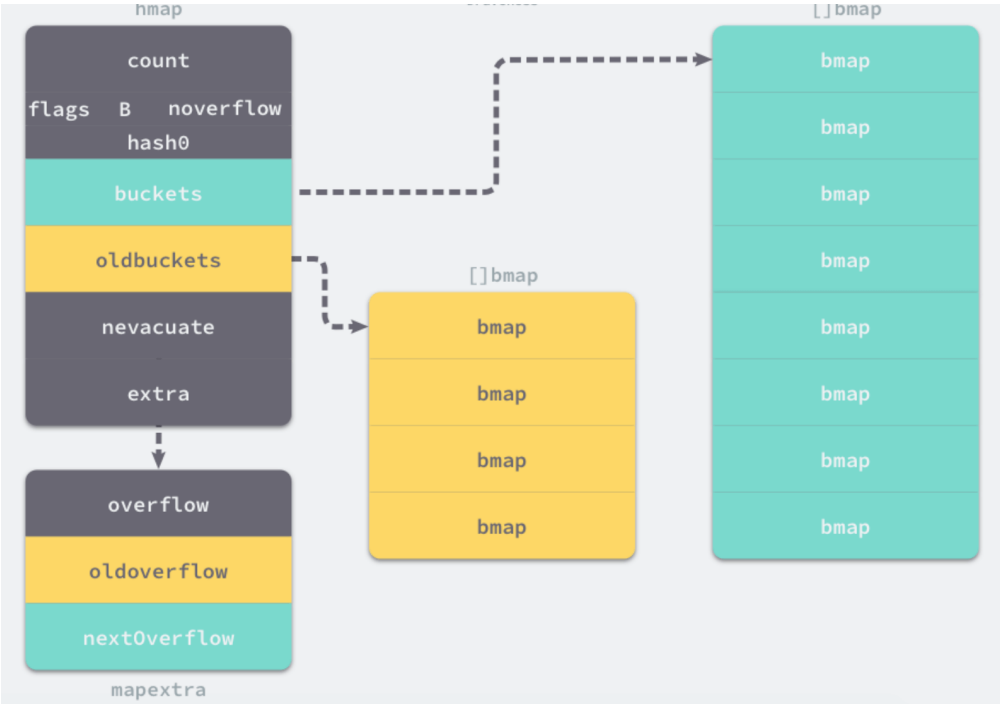
`runtime: limit the number of map overflow buckets` 引入了 `sameSizeGrow` 通过复用已有的哈希扩容机制解决该问题，一旦哈希中出现了过多的溢出桶，它会创建新桶保存数据，垃圾回收会清理老的溢出桶并释放内存\。

扩容的入口是 `hashGrow` ：

```
func hashGrow(t *maptype, h *hmap) {
    // If we've hit the load factor, get bigger.
    // Otherwise, there are too many overflow buckets,
    // so keep the same number of buckets and "grow" laterally.
    // B+1 相当于是原来 2 倍的空间
    bigger := uint8(1)
    if !overLoadFactor(h.count+1, h.B) {
        // 进行等量的内存扩容，所以 B 不变
        bigger = 0
        h.flags |= sameSizeGrow
    }
    // 将老 buckets 挂到 buckets 上
    oldbuckets := h.buckets
    // 申请新的 buckets 空间
    newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)

    flags := h.flags &^ (iterator | oldIterator)
    if h.flags&iterator != 0 {
        flags |= oldIterator
    }
    // commit the grow (atomic wrt gc)
    // 提交 grow 的动作
    h.B += bigger
    h.flags = flags
    h.oldbuckets = oldbuckets
    h.buckets = newbuckets
    // 搬迁进度为 0
    h.nevacuate = 0
    // overflow buckets 数为 0
    h.noverflow = 0

    if h.extra != nil && h.extra.overflow != nil {
        // Promote current overflow buckets to the old generation.
        if h.extra.oldoverflow != nil {
```

```
            throw("oldoverflow is not nil")
        }
        h.extra.oldoverflow = h.extra.overflow
        h.extra.overflow = nil
    }
    if nextOverflow != nil {
        if h.extra == nil {
            h.extra = new(mapextra)
        }
        h.extra.nextOverflow = nextOverflow
    }

    // the actual copying of the hash table data is done incrementally
    // by growWork() and evacuate().
}
```

哈希在扩容的过程中会通过 `makeBucketArray` 创建一组新桶和预创建的溢出桶，随后将原有的桶数组设置到 `oldbuckets` 上并将新的空桶设置到 buckets 上，溢出桶也使用了相同的逻辑更新.
这里会申请到了新的 buckets 空间，把相关的标志位都进行了处理,例如标志 nevacuate 被置为 0， 表示当前搬迁进度为 0。



在 `hashGrow` 中还看不出来等量扩容和翻倍扩容的太多区别，等量扩容创建的新桶数量只是和旧桶一样，该函数中只是创建了新的桶，并没有对数据进行拷贝和转移。

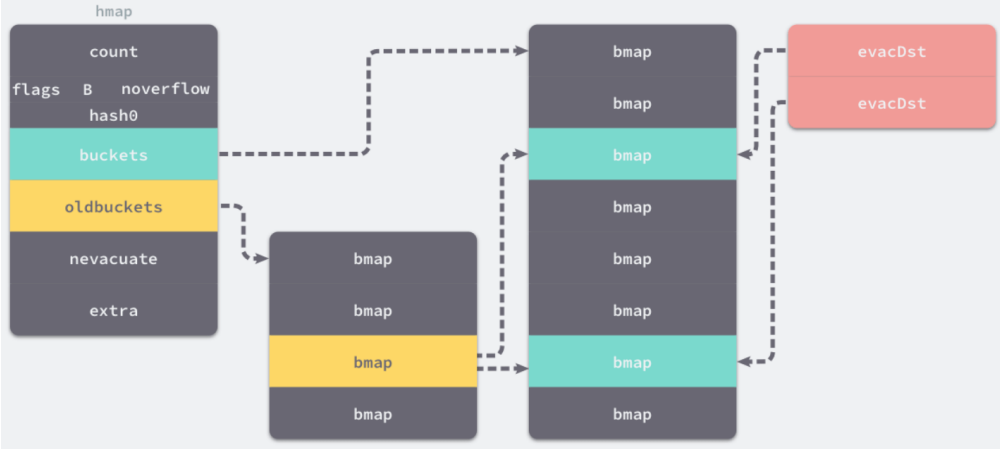
哈希表的数据迁移的过程在是 `evacuate` 中完成的，它会对传入桶中的元素进行再分配。

```
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
    // 这里会定位老的 bucket 地址
    b := (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
    // 结果是 2^B
    newbit := h.noldbuckets()
    // 如果吧没有搬迁过
    if !evacuated(b) {
        // TODO: reuse overflow buckets instead of using new ones, if there
        // is no iterator using the old buckets.  (If !oldIterator.)

        // xy contains the x and y (low and high) evacuation destinations.
```

```
        var xy [2]evacDst
        x := &xy[0]
        x.b = (*bmap)(add(h.buckets, oldbucket*uintptr(t.bucketsize)))
        x.k = add(unsafe.Pointer(x.b), dataOffset)
        x.e = add(x.k, bucketCnt*uintptr(t.keysize))

        // 如果不是等size 扩容,前后bucket序号有变,使用y 进行搬迁
        if !h.sameSizeGrow() {
            // Only calculate y pointers if we're growing bigger.
            // Otherwise GC can see bad pointers.
            y := &xy[1]
            y.b = (*bmap)(add(h.buckets, (oldbucket+newbit)*uintptr(t.bucketsize)))
            y.k = add(unsafe.Pointer(y.b), dataOffset)
            y.e = add(y.k, bucketCnt*uintptr(t.keysize))
        }
        // 遍历所有老的bucket地址
        for ; b != nil; b = b.overflow(t) {
            k := add(unsafe.Pointer(b), dataOffset)
            e := add(k, bucketCnt*uintptr(t.keysize))
            for i := 0; i < bucketCnt; i, k, e = i+1, add(k, uintptr(t.keysize)), add(e, uintptr(t.elemsize))
{
                top := b.tophash[i]
                if isEmpty(top) {
                    b.tophash[i] = evacuatedEmpty
                    continue
                }
                if top < minTopHash {
                    throw("bad map state")
                }
                k2 := k
                if t.indirectkey() {
                    k2 = *((*unsafe.Pointer)(k2))
                }
                var useY uint8
                if !h.sameSizeGrow() {
                    // Compute hash to make our evacuation decision (whether we need
                    // to send this key/elem to bucket x or bucket y).
                    hash := t.hasher(k2, uintptr(h.hash0))
                    if h.flags&iterator != 0 && !t.reflexivekey() && !t.key.equal(k2, k2) {
                        // If key != key (NaNs), then the hash could be (and probably
                        // will be) entirely different from the old hash. Moreover,
                        // it isn't reproducible. Reproducibility is required in the
                        // presence of iterators, as our evacuation decision must
                        // match whatever decision the iterator made.
                        // Fortunately, we have the freedom to send these keys either
                        // way. Also, tophash is meaningless for these kinds of keys.
                        // We let the low bit of tophash drive the evacuation decision.
                        // We recompute a new random tophash for the next level so
                        // these keys will get evenly distributed across all buckets
                        // after multiple grows.
                        useY = top & 1
                        top = tophash(hash)
                    } else {
                        if hash&newbit != 0 {
                            useY = 1
                        }
                    }
                }

                if evacuatedX+1 != evacuatedY || evacuatedX^1 != evacuatedY {
                    throw("bad evacuatedN")
                }
```

```
                b.tophash[i] = evacuatedX + useY // evacuatedX + 1 == evacuatedY
                dst := &xy[useY]                 // evacuation destination

                if dst.i == bucketCnt {
                    dst.b = h.newoverflow(t, dst.b)
                    dst.i = 0
                    dst.k = add(unsafe.Pointer(dst.b), dataOffset)
                    dst.e = add(dst.k, bucketCnt*uintptr(t.keysize))
                }
                dst.b.tophash[dst.i&(bucketCnt-1)] = top // mask dst.i as an optimization, to avoid a bounds
check
                if t.indirectkey() {
                    *(*unsafe.Pointer)(dst.k) = k2 // copy pointer
                } else {
                    typedmemmove(t.key, dst.k, k) // copy elem
                }
                if t.indirectelem() {
                    *(*unsafe.Pointer)(dst.e) = *(*unsafe.Pointer)(e)
                } else {
                    typedmemmove(t.elem, dst.e, e)
                }
                dst.i++
                // These updates might push these pointers past the end of the
                // key or elem arrays.  That's ok, as we have the overflow pointer
                // at the end of the bucket to protect against pointing past the
                // end of the bucket.
                dst.k = add(dst.k, uintptr(t.keysize))
                dst.e = add(dst.e, uintptr(t.elemsize))
            }
        }
        // Unlink the overflow buckets & clear key/elem to help GC.
        if h.flags&oldIterator == 0 && t.bucket.ptrdata != 0 {
            b := add(h.oldbuckets, oldbucket*uintptr(t.bucketsize))
            // Preserve b.tophash because the evacuation
            // state is maintained there.
            ptr := add(b, dataOffset)
            n := uintptr(t.bucketsize) - dataOffset
            memclrHasPointers(ptr, n)
        }
    }

    if oldbucket == h.nevacuate {
        advanceEvacuationMark(h, t, newbit)
    }
}
```

`evacuate` 会将一个旧桶中的数据分流到两个新桶，所以它会创建两个用于保存分配上下文的 `evacDst` 结构体，这两个结构体分别指向了一个新桶：



哈希表扩容目的:

如果这是等量扩容，那么旧桶与新桶之间是一对一的关系，所以两个 `evacDst` 只会初始化一个。而当哈希表的容量翻倍时，每个旧桶的元素会都分流到新创建的两个桶中.

只使用哈希函数是不能定位到具体某一个桶的，哈希函数只会返回很长的哈希,我们还需一些方法将哈希映射到具体的桶上。

那么如何定位key呢?

key 经过哈希计算后得到哈希值，共64个 bit 位（64位机，32位机就不讨论了，现在主流都是64位机），计算它到底要落在哪个桶时，只会用到最后 B 个 bit 位。

如果 B = 5，那么桶的数量，也就是 buckets 数组的长度是 `2^5 = 32` 。

例如，现在有一个 key 经过哈希函数计算后，得到的哈希结果是：

```
10010111 | 0000111101101100100011111001010100010010110010101010 | 01010
```

用最后的 5 个bit 位，也就是 `01010` ，值为 10,那么这个就是10号桶。

再用哈希值的高 8 位，找到此 key 在bucket中的位置，这是在寻找已有的 key。最开始桶内还没有 key，新加入的 key 会找到第一个空位，放入。

buckets 编号就是桶编号，当两个不同的key落在同一个桶中，也就是发生了哈希冲突。

通常哈希冲突的解决手段是用链表法,在 bucket 中，从前往后找到第一个空位。这样，在查找某个 key 时，先找到对应的桶，再去遍历 bucket 中的 key。

因此哈希表扩容的设计和原理，哈希在存储元素过多时会触发扩容操作，每次都会将桶的数量翻倍，扩容过程不是原子的，而是通过 `growWork` 增量触发的，在扩容期间访问哈希表时会使用旧桶，向哈希表写入数据时会触发旧桶元素的分流。

除了这种正常的扩容之外，为了解决大量写入、删除造成的内存泄漏问题，哈希引入了 `sameSizeGrow` 这一机制，在出现较多溢出桶时会整理哈希的内存减少空间的占用。

删除:

如果想要删除哈希中的元素，就需要使用 Go 语言中的 delete 关键字，这个关键字的唯一作用就是将某一个键对应的元素从哈希表中删除，无论是该键对应的值是否存在，这个内建的函数都不会返回任何的结果。

因此呢Go采用拉链法来解决哈希碰撞的问题实现了哈希表，它的访问、写入和删除等操作都在编译期间转换成了运行时的函数或者方法。

哈希在每一个桶中存储键对应哈希的前 8 位，当对哈希进行操作时，这些 `tophash` 就成为可以帮助哈希快速遍历桶中元素的缓存。

哈希表的每个桶都只能存储 8 个键值对，一旦当前哈希的某个桶超出 8 个，新的键值对就会存储到哈希的溢出桶中。

随着键值对数量的增加，溢出桶的数量和哈希的装载因子也会逐渐升高，超过一定范围就会触发扩容，扩容会将桶的数量翻倍，元素再分配的过程也是在调用写操作时增量进行的，不会造成性能的瞬时巨大损耗。

41. Go中的http包的实现原理

Golang中http包中处理 HTTP 请求主要跟两个东西相关：ServeMux 和 Handler。

ServeMux 本质上是一个 HTTP 请求路由器（或者叫多路复用器，Multiplexor）。它把收到的请求与一组预先定义的 URL 路径列表做对比，然后在匹配到路径的时候调用关联的处理器（Handler）。

处理器（Handler）负责输出HTTP响应的头和正文。任何满足了http.Handler接口的对象都可作为一个处理器。通俗的说，对象只要有个如下签名的ServeHTTP方法即可：

```
ServeHTTP(http.ResponseWriter, *http.Request)
```

Go 语言的 HTTP 包自带了几个函数用作常用处理器，比如 `FileServer` ， `NotFoundHandler` 和 `RedirectHandler` 。

应用示例:

```
package main

import (
    "log"
    "net/http"
)

func main() {
    mux := http.NewServeMux()

    rh := http.RedirectHandler("http://www.baidu.com", 307)
    mux.Handle("/foo", rh)

    log.Println("Listening...")
    http.ListenAndServe(":3000", mux)
}
```

在这个应用示例中,首先在 main 函数中我们只用了 `http.NewServeMux` 函数来创建一个空的 `ServeMux` 。 然后我们使用 `http.RedirectHandler` 函数创建了一个新的处理器，这个处理器会对收到的所有请求，都执行307重定向操作到 `http://www.baidu.com` 。

接下来我们使用 `ServeMux.Handle` 函数将处理器注册到新创建的 `ServeMux` ，所以它在 URL 路径 `/foo` 上收到所有的请求都交给这个处理器。 最后我们创建了一个新的服务器，并通过 `http.ListenAndServe` 函数监听所有进入的请求，通过传递刚才创建的 `ServeMux` 来为请求去匹配对应处理器。

在浏览器中访问 `http://localhost:3000/foo` ，你应该能发现请求已经成功的重定向了。

此刻你应该能注意到一些有意思的事情： `ListenAndServer` 的函数签名是 `ListenAndServe(addr string, handler Handler)` ，但是第二个参数我们传递的是个 `ServeMux` 。

通过这个例子我们就可以知道， `net/http` 包在编写golang web应用中有很重要的作用，它主要提供了基于HTTP协议进行工作的client实现和server实现，可用于编写HTTP服务端和客户端。

42. Goroutine发生了泄漏如何检测

通常内存泄漏，指的是能够预期的能很快被释放的内存由于附着在了长期存活的内存上、或生命期意外地被延长，导致预计能够立即回收的内存而长时间得不到回收。

在 Go 中，由于Goroutine的存在，因此,内存泄漏除了附着在长期对象上之外，还存在多种不同的形式。

- 预期能被快速释放的内存因被根对象引用而没有得到迅速释放.

当有一个全局对象时，可能不经意间将某个变量附着在其上，且忽略的将其进行释放，则该内存永远不会得到释放。

- Goroutine 泄漏

Goroutine 作为一种逻辑上理解的轻量级线程，需要维护执行用户代码的上下文信息。在运行过程中也需要消耗一定的内存来保存这类信息，而这些内存在目前版本的 Go 中是不会被释放的。

因此，如果一个程序持续不断地产生新的 goroutine、且不结束已经创建的 goroutine 并复用这部分内存，就会造成内存泄漏的现象.

可以通过Go自带的工具pprof或者使用Gops去检测诊断当前在系统上运行的Go进程的占用的资源.

例如:

```
func main() {
    for i := 0; i < 10000; i++ {
        go func() {
            select {}
        }()
    }
}
```

43. Go函数返回局部变量的指针是否安全

在 Go 中是安全的，Go 编译器将会对每个局部变量进行逃逸分析。如果发现局部变量的作用域超出该函数，则不会将内存分配在栈上，而是分配在堆上

44. Go中两个Nil可能不相等吗

Go中两个Nil可能不相等。

接口(interface) 是对非接口值(例如指针，struct等)的封装，内部实现包含 2 个字段，类型 T 和 值 V。一个接口等于 nil，当且仅当 T 和 V 处于 unset 状态（T=nil，V is unset）。

两个接口值比较时，会先比较 T，再比较 V。
接口值与非接口值比较时，会先将非接口值尝试转换为接口值，再比较。

```
func main() {
    var p *int = nil
    var i interface{} = p
    fmt.Println(i == p) // true
    fmt.Println(p == nil) // true
    fmt.Println(i == nil) // false
}
```

这个例子中，将一个nil非接口值p赋值给接口i，此时,i的内部字段为(T=*int, V=nil)，i与p作比较时，将 p 转换为接口后再比较，因此 `i == p` ，p 与 nil 比较，直接比较值，所以 `p == nil` 。

但是当 i 与nil比较时，会将nil转换为接口(T=nil, V=nil),与i(T=*int, V=nil)不相等，因此 `i != nil` 。因此 V 为 nil，但 T 不为 nil 的接口不等于 nil。

45. Goroutine和KernelThread之间是什么关系

首先我们先看下进程和线程还有协程之间的区别:

- 进程

计算机的操作系统模式是一种多任务系统，操作系统接管了所有的硬件资源，并且本身运行在一个受硬件保护的级别。所有的应用程序都以进程(process)的方式运行在比操作系统权限更低的级别，每个进程都有自己独立的地址空间，使得进程之间的地址空间相互隔离。CPU由操作系统一进行分配，每个进程根据进程的优先级的高低都有机会得到CPU,但是如果允许时间超出了一定的时间，操作系统会暂停该进程，将CPU资源分配给其他等待的进程。这种CPU的分配方式即所谓的抢占式，操作系统可以强制剥夺CPU资源并且分配给它认为目前最需要的进程。如果操作系统分配给每个进程的时间都很短，即CPU在多个进程间快速地切换，从而造成了很多进程都在同时运行的假象。

- 线程

线程有时被称为轻量级进程（Lightweight Process）,是程序执行流的最小单元，一个标准的线程由线程ID,当前指令指针（PC）、寄存器集合和堆栈组成，通常意义上，一个进程□一个到多个线程组成，各个线程之间共享程序的内存空间（包括代码段、数据段、堆等）及一些进程级的资源（如打开文件和信号）。

- 协程

协程（coroutine）是Go语言中的轻量级线程实现，由Go运行时（runtime）管理。

进程、线程、协程的关系和区别：

- 进程拥有自己独立的堆和栈，既不共享堆，亦不共享栈，进程由操作系统调度。
- 线程拥有自己独立的栈和共享的堆，共享堆，不共享栈，线程亦由操作系统调度(标准线程是的)。
- 协程和线程一样共享堆，不共享栈，协程由程序开发者在协程的代码里显示调度。

为什么协程比线程轻量？

a. go协程调用跟切换比线程效率高

线程并发执行流程： 线程是内核对外提供的服务，应用程序可以通过系统调用让内核启动线程，由内核来负责线程调度和切换。线程在等待IO操作时线程变为unrunnable状态会触发上下文切换。现代操作系统一般都采用抢占式调度，上下文切换一般发生在时钟中断和系统调用返回前，调度器计算当前线程的时间片，如果需要切换就从运行队列中选出一个目标线程，保存当前线程的环境，并且恢复目标线程的运行环境，最典型的就是切换ESP指向目标线程内核堆栈，将EIP指向目标线程上次被调度出时的指令地址。

go协程并发执行流程：不依赖操作系统和其提供的线程，golang自己实现的CSP并发模型实现：M, P, G .go协程也叫用户态线程，协程之间的切换发生在用户态。在用户态没有时钟中断，系统调用等机制,因此效率高

b. go协程占用内存少

执行go协程只需要极少的栈内存（大概是4～5KB），默认情况下，线程栈的大小为1MB。goroutine就是一段代码，一个函数入口，以及在堆上为其分配的一个堆栈。所以它非常廉价，我们可以很轻松的创建上万个goroutine，但它们并不是被操作系统所调度执行。

因此协程和线程一样共享堆，不共享栈，协程由用户态下面的轻量级线程。

Mysql基础知识

1. **Mysql索引用的是什么算法**

Mysql 索引选用的是B+树,平衡二叉树的高度太高,查找可能需要较多的磁盘IO。B树索引占用内存较高(非叶子节点存储数据)。

B+树, 主要是查询效率高，O(logN)，可以充分利用磁盘预读的特性，多叉树，深度小，叶子结点有序且存储数据.

2. **Mysql事务的基本要素**

- 原子性：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。
- 一致性：事务开始前和结束后，数据库的完整性约束没有被破坏。
- 隔离性：同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。
- 持久性：事务完成后，事务对数据库的所有更新将被保存到数据库，不能回滚。

3. **Mysql的存储引擎**

- InnoDB存储引擎

InnoDB存储引擎支持事务，其设计目标主要面向在线事务处理（OLTP）的应用。

其特点是行锁设计，支持外键，并支持非锁定锁，即默认读取操作不会产生锁。从Mysql5.5.8版本开始，InnoDB存储引擎是默认的存储引擎。

- MyISAM存储引擎

MyISAM存储引擎不支持事务、表锁设计，支持全文索引，主要面向一些OLAP数据库应用。

InnoDB的数据文件本身就是主索引文件，而MyISAM的主索引和数据是分开的。

- NDB存储引擎

NDB存储引擎是一个集群存储引擎，其结构是share nothing的集群架构，能提供更高的可用性。

NDB的特点是数据全部放在内存中（从MySQL 5.1版本开始，可以将非索引数据放在磁盘上），因此主键查找的速度极快，并且通过添加NDB数据存储节点可以线性地提高数据库性能，是高可用、高性能的集群系统。
NDB存储引擎的连接操作是在MySQL数据库层完成的，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢。如果解决了这个问题，NDB存储引擎的市场应该是非常巨大的。

- Memory存储引擎

Memory存储引擎（之前称HEAP存储引擎）将表中的数据存放在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时表，以及数据仓库中的纬度表。Memory存储引擎默认使用哈希索引，而不是我们熟悉的B+树索引。
虽然Memory存储引擎速度非常快，但在使用上还是有一定的限制。比如，只支持表锁，并发性能较差，并且不支持TEXT和BLOB列类型。最重要的是，存储变长字段时是按照定常字段的方式进行的，因此会浪费内存。

- Archive存储引擎

Archive存储引擎只支持INSERT和SELECT操作，从MySQL 5.1开始支持索引。
Archive存储引擎使用zlib算法将数据行（row）进行压缩后存储，压缩比一般可达1:10。正如其名字所示，Archive存储引擎非常适合存储归档数据，如日志信息。
Archive存储引擎使用行锁来实现高并发的插入操作，但是其本身并不是事务安全的存储引擎，其设计目标主要是提供高速的插入和压缩功能。

- Maria存储引擎

Maria存储引擎是新开发的引擎，设计目标主要是用来取代原有的MyISAM存储引擎，从而成为MySQL的默认存储引擎。它可以看做是MyISAM的后续版本。
Maria存储引擎的特点是：支持缓存数据和索引文件，应用了行锁设计，提供了MVCC功能，支持事务和非事务安全的选项，以及更好的BLOB字符类型的处理性能。

4. **Mysql事务隔离级别**

Mysql有四种事务隔离级别,默认的是可重复读.

事务隔离级别	脏读	不可重复读	幻读
读未提交	是	是	是
读已提交	否	是	是
可重复读	否	否	是
串行	否	否	否

- 读未提交(Read uncommitted)

一个事务可以读取另一个未提交事务的数据，最低级别，任何情况都无法保证。

(1)所有事务都可以看到其他未提交事务的执行结果
(2)本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少
(3)该级别引发的问题是——脏读(Dirty Read)：读取到了未提交的数据

- 读已提交(Read committed)

一个事务要等另一个事务提交后才能读取数据，可避免脏读的发生。

(1)这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）
(2)它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变
(3)这种隔离级别出现的问题是——不可重复读(Nonrepeatable Read),不可重复读意味着我们在同一个事务中执行完全相同的select语句时可能看到不一样的结果。

导致这种情况的原因可能有：

(1)有一个交叉的事务有新的commit，导致了数据的改变;
(2)一个数据库被多个实例操作时,同一事务的其他实例在该实例处理其间可能会有新的commit.

- 可重复读(Repeatable read)

就是在开始读取数据（事务开启）时，不再允许修改操作，可避免脏读、不可重复读的发生。

(1)这是MySQL的默认事务隔离级别
(2)它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行
(3)此级别可能出现的问题——幻读(Phantom Read)：当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行
(4)InnoDB和Falcon存储引擎通过多版本并发控制(MVCC，Multiversion Concurrency Control)机制解决了该问题.InnoDB采用MVCC来支持高并发，实现了四个标准隔离级别。默认基本是可重复读，并且提供间隙锁（next-key locks）策略防止幻读出现。

- 串行(Serializable)

串行(Serializable)，是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。
但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。Mysql的默认隔离级别是Repeatable read。

(1)这是最高的隔离级别.
(2)它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之,它是在每个读的数据行上加上共享锁。
(3)在这个级别，可能导致大量的超时现象和锁竞争.

5. Mysql高可用方案有哪些

Mysql高可用方案包括:

- 主从复制方案

这是MySQL自身提供的一种高可用解决方案，数据同步方法采用的是 `MySQL replication` 技术。 `MySQL replication` 就是从服务器到主服务器拉取二进制日志文件，然后再将日志文件解析成相应的SQL在从服务器上重新执行一遍主服务器的操作，通过这种方式保证数据的一致性。
为了达到更高的可用性，在实际的应用环境中，一般都是采用 `MySQL replication` 技术配合高可用集群软件 `keepalived` 来实现自动 `failover` ，这种方式可以实现95.000%的SLA。

- MMM/MHA高可用方案

MMM提供了MySQL主主复制配置的监控、故障转移和管理的一套可伸缩的脚本套件。在MMM高可用方案中，典型的应用是双主多从架构，通过MySQL replication技术可以实现两个服务器互为主从，且在任何时候只有一个节点可以被写入，避免了多点写入的数据冲突。
同时，当可写的主节点故障时，MMM套件可以立刻监控到，然后将服务自动切换到另一个主节点，继续提供服务，从而实现MySQL的高可用。

- Heartbeat/SAN高可用方案

在这个方案中，处理failover的方式是高可用集群软件Heartbeat，它监控和管理各个节点间连接的网络，并监控集群服务，当节点出现故障或者服务不可用时，自动在其他节点启动集群服务。
在数据共享方面，通过SAN（Storage Area Network）存储来共享数据，这种方案可以实现99.990%的SLA。

- Heartbeat/DRBD高可用方案

这个方案处理failover的方式上依旧采用Heartbeat，不同的是，在数据共享方面，采用了基于块级别的数据同步软件DRBD来实现。
DRBD是一个用软件实现的、无共享的、服务器之间镜像块设备内容的存储复制解决方案。和SAN网络不同，它并不共享存储，而是通过服务器之间的网络复制数据。

- NDB CLUSTER高可用方案

国内用NDB集群的公司非常少，貌似有些银行有用。NDB集群不需要依赖第三方组件，全部都使用官方组件，能保证数据的一致性，某个数据节点挂掉，其他数据节点依然可以提供服务，管理节点需要做冗余以防挂掉。
缺点是：管理和配置都很复杂，而且某些SQL语句例如join语句需要避免。

6. Mysql中utf8和utf8mb4区别

MySQL在5.5.3之后增加了这个 `utf8mb4` 的编码，mb4就是 `most bytes 4` 的意思，专门用来兼容四字节的 `unicode` 。好在 `utf8mb4` 是 `utf8` 的超集，除了将编码改为 `utf8mb4` 外不需要做其他转换。当然，为了节省空间，一般情况下使用utf8也就可以了。

Mysql支持的 `utf8` 编码最大字符长度为 3 字节，如果遇到 4 字节的宽字符就会插入异常了。三个字节的 UTF-8 最大能编码的 `Unicode` 字符是 `0xffff` ，也就是 `Unicode` 中的基本多文种平面(BMP)。任何不在基本多文本平面的 Unicode字符，都无法使用 Mysql 的 `utf8` 字符集存储。

包括 Emoji 表情(Emoji 是一种特殊的 Unicode 编码，常见于 ios 和 android 手机上)，和很多不常用的汉字，以及任何新增的 Unicode 字符等等。

Mysql 中保存 4 字节长度的 UTF-8 字符，需要使用 `utf8mb4` 字符集，但只有5.5.3版本以后的才支持(查看版本：select version();)。因此呢，为了获取更好的兼容性，应该总是使用 `utf8mb4` 而非 utf8.

对于 CHAR 类型数据， `utf8mb4` 会多消耗一些空间，根据 Mysql 官方建议，使用 VARCHAR 替代 CHAR。

7. Mysql中乐观锁和悲观锁区别

- 悲观锁(Pessimistic Lock)

悲观锁顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。

传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

- 乐观锁(Optimistic Lock),

乐观锁顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。

乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于 `write_condition` 机制的其实都是提供的乐观锁。

乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号，或者时间戳，然后按照如下方式实现：

```
1. SELECT data AS old_data, version AS old_version FROM …;
2. 根据获取的数据进行业务操作，得到new_data和new_version
3. UPDATE SET data = new_data, version = new_version WHERE version = old_version
if (updated row > 0) {
    // 乐观锁获取成功，操作完成
} else {
    // 乐观锁获取失败，回滚并重试
}
```

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句时会获取被update行的写锁，直到这一行被成功更新后才释放。

因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这之间没有发生并发的修改。如果更新失败即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。

但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

8. Mysql索引主要是哪些

索引的目的在于提高查询效率.

索引的类型：

- UNIQUE(唯一索引)：不可以出现相同的值，可以有NULL值
- INDEX(普通索引)：允许出现相同的索引内容

- PROMARY KEY(主键索引)：不允许出现相同的值
- fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维
- 组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一

索引虽然好处很多，但过多的使用索引可能带来相反的问题，索引也是有缺点的：

- 虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT,UPDATE和DELETE。因为更新表时，mysql不仅要保存数据，还要保存一下索引文件.
- 建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在要给大表上建了多种组合索引，索引文件会膨胀很宽,索引只是提高效率的一个方式，如果mysql有大数据量的表，就要花时间研究建立最优的索引，或优化查询语句。

使用索引时，有一些技巧：

- 索引不会包含有NULL的列

只要列中包含有NULL值，都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此符合索引就是无效的。

- 使用短索引

对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个char（255）的列，如果在前10个或20个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

- 索引列排序

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。

- like语句操作
  一般情况下不鼓励使用like操作，如果非使用不可，注意正确的使用方式。 `like ‘%aaa%’` 不会使用索引，而 `like ‘aaa%’` 可以使用索引。
- 不要在列上进行运算
- 不使用 `NOT IN` 、 `<>` 、 `!=` 操作，但 `<` , `<=` , `=` , `>` , `>=` ,BETWEEN,IN是可以用到索引的
- 索引要建立在经常进行select操作的字段上。
  这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。
- 索引要建立在值比较唯一的字段上。
- 对于那些定义为text、image和bit数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少。
- 在where和join中出现的列需要建立索引。
- where的查询条件里有不等号(where column != ...),mysql将无法使用索引。

- 如果where字句的查询条件里使用了函数(如：where DAY(column)=...),mysql将无法使用索引。
- 在join操作中(需要从多个数据表提取数据时)，mysql只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用。

组合索引的作用:

1. 减少开销。

建一个组合索引(col1,col2,col3)，实际相当于建了(col1),(col1,col2),(col1,col2,col3)三个索引。每多一个索引，都会增加写操作的开销和磁盘空间的开销。
对于大量数据的表，使用组合索引会大大的减少开销。

2. 覆盖索引。

通常指一个查询语句的执行只用从索引中就能够取得，不必从数据表中读取。也可以称之为实现了索引覆盖。

对组合索引(col1,col2,col3)，如果有如下的 `sql: select col1,col2,col3 from test where col1=1 and col2=2` 。

那么MySQL可以直接通过遍历索引取得数据，而无需回表，这减少了很多的随机io操作。减少io操作，特别的随机io其实是dba主要的优化策略。

所以，在真正的实际应用中，覆盖索引是主要的提升性能的优化手段之一。

3. 效率高。

索引列越多，通过索引筛选出的数据越快。

9. Mysql联合索引最左匹配原则

最左前缀匹配原则：

1. 在MySQL建立联合索引时会遵守最左前缀匹配原则，即最左优先.

在检索数据时从联合索引的最左边开始匹配，Mysql会一直向右匹配直到遇到范围查询（ `>、<、between、like` ）就停止匹配了.

就比如 `a=3 and b=4 and c>5 and d=6` 如果建立(abcd)顺序的索引,d就用不到索引了，如果建立(abdc)的索引则都可以用到索引，其中abd的顺序可以任意调整，因为查询优化器会重新编排（即使是c>5 and b=4 and d=6 and a=3也会全部用到 abdc索引 ）.

2. `=` 和 `in` 可以乱序，比如 `a=1 and b=2 and c=3` 建立(abc)索引可以任意顺序，mysql查询优化器会优化顺序.

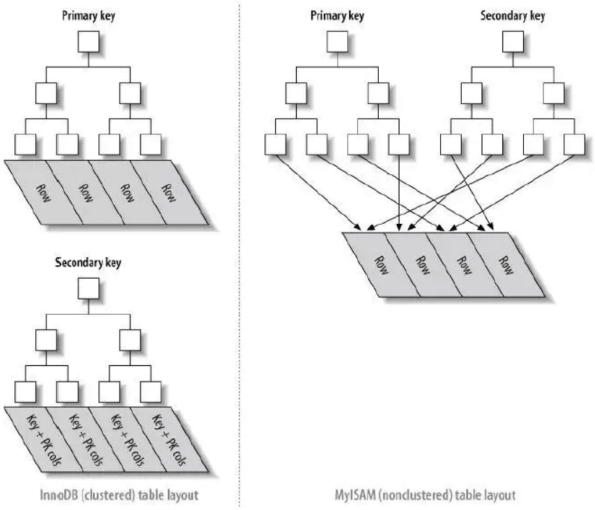
这里需要注意下, 比如abc索引 那么只要查询条件有a即可用到abc索引（如abc ab ac a）,没有a就用不到。

最左前缀匹配成因：Mysql是创建复合索引的规则是根据索引最左边的字段进行排序，在第一个字段排序的基础上再进行第二个字段排序，类似于 `order by col1，col2…` 所以第一个字段是绝对有序的 第二个字段就是无序的了，所以Mysql 强调最左前缀匹配.

10. 聚簇索引和非聚簇索引区别

聚簇索引与非聚簇索引的区别是：叶子节点是否存放一整行记录.

InnoDB 主键使用的是聚簇索引，MyISAM 不管是主键索引，还是二级索引使用的都是非聚簇索引。



1. 对于聚簇索引表来说（左图），表数据是和主键一起存储的，主键索引的叶结点存储行数据(包含了主键值)，二级索引的叶结点存储行的主键值。
   使用的是B+树作为索引的存储结构，非叶子节点都是索引关键字，但非叶子节点中的关键字中不存储对应记录的具体内容或内容地址。叶子节点上的数据是主键与具体记录(数据内容)。

2. 对于非聚簇索引表来说（右图），表数据和索引是分成两部分存储的，主键索引和二级索引存储上没有任何区别。使用的是B+树作为索引的存储结构，所有的节点都是索引，叶子节点存储的是索引+索引对应的记录的数据。

因此, 聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针

11. 如何查询一个字段是否命中了索引

通过explain sql可看下SQL是否走了索引，很快对比出来 .

当一个sql中索引字段为int类型时，例如搜索条件 `where num=″111″` 与 `where num=111` 都可以使用该字段的索引. 当一个中索引字段为varchar类型时，例如搜索条件 `where num=″111″` 可以使用索引， `where num=111` 不可以使用索引.

12. Mysql中查询数据什么情况下不会命中索引

通常不命中索引有接种情况:

- 索引规范不合理,sql解析器不命中索引.
- 表中索引是以表中数据量字段最多的建立的索引,sql解析器不命中索引.(实际就是索引没用,最后全局查找了)
- bool的字段做索引,sql选择器不命中索引.
- 模糊查询 %like
- 索引列参与计算,使用了函数
- 非最左前缀顺序
- where对null判断
- where不等于
- or操作有至少一个字段没有索引
- 需要回表的查询结果集过大（超过配置的范围）

13. Mysql中的MVCC是什么

数据库并发控制——锁:Multiversion (version) concurrency control (MCC or MVCC) 多版本并发控制 ，它是数据库管理系统一种常见的并发控制。

并发控制常用的是锁，当线程要对一个共享资源进行操作的时候，加锁是一种非常简单粗暴的方法(事务开始时给 DQL 加读锁，给 DML 加写锁)，这种锁是一种 悲观 的实现方式，也就是说这会给其他事务造成堵塞，从而影响数据库性能。

其中在数据库中最常见的就是悲观锁和乐观锁:

- 悲观锁

当一个线程需要对共享资源进行操作的时候，首先对共享资源进行加锁，当该线程持有该资源的锁的时候，其他线程对该资源进行操作的时候会被阻塞.

- 乐观锁

当一个线程需要对一个共享资源进行操作的时候，不对它进行加锁，而是在操作完成之后进行判断。
比如乐观锁会通过一个版本号控制，如果操作完成后通过版本号进行判断在该线程操作过程中是否有其他线程已经对该共享资源进行操作了，如果有则通知操作失败，如果没有则操作成功，当然除了版本号还有CAS，如果不了解的可以去学习一下，这里不做过多涉及。

MVCC的两种读形式:

- 快照读

读取的只是当前事务的可见版本，不用加锁。而你只要记住 简单的 `select` 操作就是快照读(select * from table where id = xxx)。

- 当前读

读取的是当前版本，比如 特殊的读操作，更新/插入/删除操作.

比如：

```
select * from table where xxx lock in share mode,
select * from table where xxx for update,
update table set....
insert into table (xxx,xxx) values (xxx,xxx)
delete from table where id = xxx
```

MVCC的实现原理:

MVCC 使用了“三个隐藏字段”来实现版本并发控制，MySQL在创建建表的时候 innoDB 创建的真正的三个隐藏列吧。

RowID	DB_TRX_ID	DB_ROLL_PTR	id	name	password
自动创建的id	事务id	回滚指针	id	name	password

- RowID：隐藏的自增ID，当建表没有指定主键，InnoDB会使用该RowID创建一个聚簇索引。
- DB_TRX_ID：最近修改（更新/删除/插入）该记录的事务ID。
- DB_ROLL_PTR：回滚指针，指向这条记录的上一个版本。

其实还有一个删除的flag字段，用来判断该行记录是否已经被删除。

而 MVCC 使用的是其中的 事务字段，回滚指针字段，是否删除字段。

我们来看一下现在的表格(isDelete是我自己取的，按照官方说法是在一行开头的content里面，这里其实位置无所谓，你只要知道有就行了)。

isDelete	DB_TRX_ID	DB_ROLL_PTR	id	name	password
true/false	事务id	回滚指针	id	name	password

14. **Mvcc和Redolog和Undolog以及Binlog有什么不同**

- Mvcc

MVCC多版本并发控制是MySQL中基于乐观锁理论实现隔离级别的方式，用于读已提交和可重复读取隔离级别的实现。
在MySQL中，会在表中每一条数据后面添加两个字段,最近修改该行数据的事务ID，指向该行（undolog表中）回滚段的指针。
Read View判断行的可见性，创建一个新事务时，copy一份当前系统中的活跃事务列表。意思是，当前不应该被本事务看到的其他事务id列表。

- UndoLog

UndoLog也就是我们常说的回滚日志文件 主要用于事务中执行失败，进行回滚，以及MVCC中对于数据历史版本的查看。
由引擎层的InnoDB引擎实现,是逻辑日志,记录数据修改被修改前的值,比如”把id=’B’ 修改为id = ‘B2’ ，那么undo日志就会用来存放id =’B’的记录”。
当一条数据需要更新前,会先把修改前的记录存储在undolog中,如果这个修改出现异常,,则会使用undo日志来实现回滚操作,保证事务的一致性。
当事务提交之后，undo log并不能立马被删除,而是会被放到待清理链表中,待判断没有事物用到该版本的信息时才可以清理相应undolog。它保存了事务发生之前的数据的一个版本，用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读。

- Redolog

Redolog是重做日志文件是记录数据修改之后的值，用于持久化到磁盘中。

Redolog包括两部分：

一. 是内存中的日志缓冲(redo log buffer)，该部分日志是易失性的；

二. 是磁盘上的重做日志文件(redo log file)，该部分日志是持久的。

由引擎层的InnoDB引擎实现,是物理日志,记录的是物理数据页修改的信息,比如“某个数据页上内容发生了哪些改动”。当一条数据需要更新时,InnoDB会先将数据更新，然后记录redoLog 在内存中，然后找个时间将redoLog的操作执行到磁盘上的文件上。
不管是否提交成功我都记录，你要是回滚了，那我连回滚的修改也记录。它确保了事务的持久性。

- Binlog

Binlog由Mysql的Server层实现,是逻辑日志,记录的是sql语句的原始逻辑，比如”把id=’B’ 修改为id = ‘B2’。

Binlog会写入指定大小的物理文件中,是追加写入的,当前文件写满则会创建新的文件写入。

产生:事务提交的时候,一次性将事务中的sql语句,按照一定的格式记录到binlog中。

用于复制和恢复在主从复制中，从库利用主库上的binlog进行重播(执行日志中记录的修改逻辑),实现主从同步。业务数据不一致或者错了，用binlog恢复。

15. **Mysql读写分离以及主从同步**

16. 原理：主库将变更写binlog日志，然后从库连接到主库后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个中继日志中，接着从库中有一个sql线程会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再执行一遍sql，这样就可以保证自己跟主库的数据一致。

17. 问题：这里有很重要一点，就是从库同步主库数据的过程是串行化的，也就是说主库上并行操作，在从库上会串行化执行，由于从库从主库拷贝日志以及串行化执行sql特点，在高并发情况下，从库数据一定比主库慢一点，是有延时的，所以经常出现，刚写入主库的数据可能读不到了，要过几十毫秒，甚至几百毫秒才能读取到。还有一个问题，如果突然主库宕机了，然后恰巧数据还没有同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。所以mysql实际上有两个机制，一个是半同步复制，用来解决主库数据丢失问题，一个是并行复制，用来解决主从同步延时问题。

18. 半同步复制：semi-sync复制，指的就是主库写入binlog日志后，就会将强制此时立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库ack之后才会认为写完成。

19. 并发复制：指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这样库级别的并行。（将主库分库也可缓解延迟问题）

20. **InnoDB的关键特性**

21. 插入缓冲：对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个Insert Buffer对象中。然后再以一定的频率和情况进行Insert Buffer和辅助索引页子节点的merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。

22. 两次写：两次写带给InnoDB存储引擎的是数据页的可靠性，有经验的DBA也许会想，如果发生写失效，可以通过重做日志进行恢复。这是一个办法。但是必须清楚地认识到，如果这个页本身已经发生了损坏（物理到page页的物理日志成功页内逻辑日志失败），再对其进行重做是没有意义的。这就是说，在应用（apply）重做日志前，用户需要一个页的副本，当写入失效发生时，先通过页的副本来还原该页，再进行重做。在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过memcpy函数将脏页先复制到内存中的doublewrite buffer，之后通过doublewrite buffer再分两次，每次1MB顺序地写入共享表空间的物理磁盘上，这就是doublewrite。

23. 自适应哈希索引：InnoDB存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引。

24. 异步IO：为了提高磁盘操作性能，当前的数据库系统都采用异步IO（AIO）的方式来处理磁盘操作。AIO的另一个优势是可以进行IO Merge操作，也就是将多个IO合并为1个IO，这样可以提高IOPS的性能。

25. 刷新邻接页：当刷新一个脏页时，InnoDB存储引擎会检测该页所在区（extent）的所有页，如果是脏页，那么一起进行刷新。这样做的好处显而易见，通过AIO可以将多个IO写入操作合并为一个IO操作，故该工作机制在传统机械磁盘下有着显著的优势。

26. **Mysql如何保证一致性和持久性**

Mysql为了保证 `ACID` 中的一致性和持久性，使用了WAL(`Write-Ahead Logging` ,先写日志再写磁盘)。 `Redo log` 就是一种WAL的应用。

当数据库忽然掉电，再重新启动时，Mysql可以通过 `Redo log` 还原数据。也就是说，每次事务提交时，不用同步刷新磁盘数据文件，只需要同步刷新 `Redo log` 就足够了。

18. **为什么选择B+树作为索引结构**

- Hash索引：Hash索引底层是哈希表，哈希表是一种以key-value存储数据的结构，所以多个数据在存储关系上是完全没有任何顺序关系的，所以，对于区间查询是无法直接通过索引查询的，就需要全表扫描。所以，哈希索引只适用于等值查询的场景。而B+ 树是一种多路平衡查询树，所以他的节点是天然有序的（左子节点小于父节点、父节点小于右子节点），所以对于范围查询的时候不需要做全表扫描

- 二叉查找树：解决了排序的基本问题，但是由于无法保证平衡，可能退化为链表。

- 平衡二叉树：通过旋转解决了平衡的问题，但是旋转操作效率太低。

- 红黑树：通过舍弃严格的平衡和引入红黑节点，解决了AVL旋转效率过低的问题，但是在磁盘等场景下，树仍然太高，IO次数太多。

- B+树：在B树的基础上，将非叶节点改造为不存储数据纯索引节点，进一步降低了树的高度；此外将叶节点使用指针连接成链表，范围查询更加高效。
  此外, B+树, 主要是查询效率高，O(logN)，可以充分利用磁盘预读的特性，多叉树，深度小，叶子结点有序且存储数据.

19. InnoDB的行锁模式

- 共享锁(S)：用法lock in share mode，又称读锁，允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其他事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁。这保证了其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改。

- 排他锁(X)：用法for update，又称写锁，允许获取排他锁的事务更新数据，阻止其他事务取得相同的数据集共享读锁和排他写锁。
  若事务T对数据对象A加上X锁，事务T可以读A也可以修改A，其他事务不能再对A加任何锁，直到T释放A上的锁。在没有索引的情况下，InnoDB只能使用表锁。

20. 哈希(hash)比树(tree)更快，索引结构为什么要设计成树型

加速查找速度的数据结构，常见的有两类：

(1)哈希，例如HashMap，查询/插入/修改/删除的平均时间复杂度都是O(1)；
(2)树，例如平衡二叉搜索树，查询/插入/修改/删除的平均时间复杂度都是O(lg(n))；

哈希只能满足等值查询, 不满足范围和大小查询, 其次哈希不可以排序.

Mysql是用等值查询,用树的话,等值查询只需要顺序遍历即可.

但是对于排序查询的sql需求：分组：`group by`,排序：`order by`,比较：`<、>`等,哈希型的索引，时间复杂度会退化为O(n)，而树型的“有序”特性，依然能够保持O(log(n)) 的高效率。

21. 为什么索引的key长度不能太长

key 太长会导致一个页当中能够存放的 key 的数目变少，间接导致索引树的页数目变多，索引层次增加，从而影响整体查询变更的效率。

22. Mysql的数据如何恢复到任意时间点

恢复到任意时间点以定时的做全量备份，以及备份增量的 binlog 日志为前提。恢复到任意时间点首先将全量备份恢复之后，再此基础上回放增加的 binlog 直至指定的时间点。

23. Mysql为什么加了索引可以加快查询

在数据十分庞大的时候，索引可以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。

索引的优缺点:

优势：可以快速检索，减少I/O次数，加快检索速度；根据索引分组和排序，可以加快分组和排序；

劣势：索引本身也是表，因此会占用存储空间，一般来说，索引表占用的空间的数据表的1.5倍；索引表的维护和创建需要时间成本，这个成本随着数据量增大而增大；构建索引会降低数据表的修改操作（删除，添加，修改）的效率，因为在修改数据表的同时还需要修改索引表.

24. **Explain命令有什么用**

在开发的过程中,我们有时会用慢查询去记录一些执行时间比较久的Sql语句，找出这些Sql语句并不意味着完事了，这个时候我们就需要用到explain这个命令来查看一个这些Sql语句的执行计划，查看该Sql语句有没有使用上了索引，有没有做全表扫描，这些都可以通过explain命令来查看。
所以我们深入了解Mysql的基于开销的优化器，还可以获得很多可能被优化器考虑到的访问策略的细节，以及当运行SQL语句时哪种策略预计会被优化器采用。

```
> explain select * from server;
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
| id | select_type | table   | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | server  | ALL  | NULL          | NULL | NULL    | NULL |    1 | NULL  |
+----+-------------+---------+------+---------------+------+---------+------+------+-------+
1 row in set (0.03 sec)
```

expain出来的信息有10列，分别是 `id、select_type、table、type、possible_keys、key、key_len、ref、rows、Extra` .

```
id:select选择标识符.
select_type:表示查询的类型.
table:输出结果集的表.
partitions:匹配的分区.
type:表示表的连接类型.
possible_keys:表示查询时，可能使用的索引.
key:表示实际使用的索引.
key_len:索引字段的长度.
ref:列与索引的比较.
rows:扫描出的行数(估算的行数).
filtered:按表条件过滤的行百分比.
Extra:执行情况的描述和说明.
```

- id

id是Sql执行的顺序的标识,Sql从大到小的执行:

1. id相同时，执行顺序由上至下.
2. 如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行.
3. id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行.

- select_type 查询的类型

示查询中每个select子句的类型:

1. SIMPLE(简单SELECT,不使用UNION或子查询等)
2. PRIMARY(查询中若包含任何复杂的子部分,最外层的select被标记为PRIMARY)
3. UNION(UNION中的第二个或后面的SELECT语句)
4. DEPENDENT UNION(UNION中的第二个或后面的SELECT语句，取决于外面的查询)
5. UNION RESULT(UNION的结果)
6. SUBQUERY(子查询中的第一个SELECT)

7. DEPENDENT SUBQUERY(子查询中的第一个SELECT，取决于外面的查询)
8. DERIVED(派生表的SELECT, FROM子句的子查询)
9. UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存，必须重新评估外链接的第一行)

- table

table显示这一行的数据是关于哪张表的，有时不是真实的表名字,看到的是derivedx.

```
> explain select * from (select * from ( select * from t1 where id=2602) a) b;
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
| id | select_type | table      | type   | possible_keys   | key     | key_len | ref  | rows | Extra |
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
|  1 | PRIMARY     | <derived2> | system | NULL            | NULL    | NULL    | NULL |    1 |       |
|  2 | DERIVED     | <derived3> | system | NULL            | NULL    | NULL    | NULL |    1 |       |
|  3 | DERIVED     | t1         | const  | PRIMARY,idx_t1_id | PRIMARY | 4       |      |    1 |       |
+----+-------------+------------+--------+-----------------+---------+---------+------+------+-------+
```

- type 表的连接类型

type表示Mysql在表中找到所需行的方式，又称“访问类型”。

常用的类型有： ALL, index, range, ref, eq_ref, const, system, NULL（从左到右，性能从差到好）.

1. ALL：Full Table Scan， Mysql将遍历全表以找到匹配的行.
2. index: Full Index Scan，index与ALL区别为index类型只遍历索引树.
3. range:只检索给定范围的行，使用一个索引来选择行.
4. 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.
5. eq_ref: 类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件.
6. const、system: 当Mysql对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，Mysql就能将该查询转换为一个常量,system是const类型的特例，当查询的表只有一行的情况下，使用system.
7. NULL: Mysql在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。

- possible_keys

possible_keys指出Mysql能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用.

该列完全独立于 `EXPLAIN` 输出所示的表的次序。这意味着在 `possible_keys` 中的某些键实际上不能按生成的表次序使用。
如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查WHERE子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用EXPLAIN检查查询.

- Key

key列显示MySql实际决定使用的键（索引）.

如果没有选择索引，键是NULL。要想强制Mysql使用或忽视 `possible_keys` 列中的索引，在查询中使用 `FORCE INDEX、USE INDEX` 或者 `IGNORE INDEX` 。

- key_len

key_len表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度（key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的）不损失精确性的情况下，长度越短越好 .

- ref

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

rows表示Mysql根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数.

- Extra执行情况的描述和说明

该列包含Mysql解决查询的详细信息,有以下几种情况：

1. Using where:列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤.
2. Using temporary：表示Mysql需要使用临时表来存储结果集，常见于排序和分组查询.
3. Using filesort：Mysql中无法利用索引完成的排序操作称为“文件排序”
4. Using join buffer：改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。
5. Impossible where：这个值强调了where语句会导致没有符合条件的行。
6. Select tables optimized away：这个值意味着仅通过使用索引，优化器可能仅从聚合函数结果中返回一行.

Redis基础知识

1. **Redis的数据结构及使用场景**

- String字符串

字符串类型是 Redis 最基础的数据结构，首先键都是字符串类型，而且 其他几种数据结构都是在字符串类型基础上构建的，我们常使用的 set key value 命令就是字符串。常用在缓存、计数、共享Session、限速等。

- Hash哈希

在Redis中，哈希类型是指键值本身又是一个键值对结构，哈希可以用来存放用户信息，比如实现购物车。

- List列表（双向链表）

列表（list）类型是用来存储多个有序的字符串。可以做简单的消息队列的功能。

- Set集合

集合（set）类型也是用来保存多个的字符串元素，但和列表类型不一 样的是，集合中不允许有重复元素，并且集合中的元素是无序的，不能通过索引下标获取元素。
利用 Set 的交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

- Sorted Set有序集合（跳表实现）

Sorted Set 多了一个权重参数 Score，集合中的元素能够按 Score 进行排列。可以做排行榜应用，取 TOP N 操作。

2. Redis持久化的几种方式

Redis为了保证效率，数据缓存在了内存中，但是会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件中，以保证数据的持久化。

Redis的持久化策略有两种：

1. RDB：快照形式是直接把内存中的数据保存到一个dump的文件中，定时保存，保存策略。

当Redis需要做持久化时，Redis会fork一个子进程，子进程将数据写到磁盘上一个临时RDB文件中。当子进程完成写临时文件后，将原来的RDB替换掉。

2. AOF：把所有的对Redis的服务器进行修改的命令都存到一个文件里，命令的集合。

使用AOF做持久化，每一个写命令都通过write函数追加到 `appendonly.aof` 中。
aof的默认策略是每秒钟fsync一次，在这种配置下，就算发生故障停机，也最多丢失一秒钟的数据。

缺点是对于相同的数据集来说，AOF的文件体积通常要大于RDB文件的体积。根据所使用的fsync策略，AOF的速度可能会慢于RDB。

Redis默认是快照RDB的持久化方式。对于主从同步来说，主从刚刚连接的时候，进行全量同步（RDB）,全同步结束后，进行增量同步(AOF)。

3. Redis的LRU具体实现

传统的LRU是使用栈的形式，每次都将最新使用的移入栈顶，但是用栈的形式会导致执行 `select *` 的时候大量非热点数据占领头部数据，所以需要改进。
Redis每次按key获取一个值的时候，都会更新value中的lru字段为当前秒级别的时间戳。Redis初始的实现算法很简单，随机从dict中取出五个key,淘汰一个lru字段值最小的。
在3.0的时候，又改进了一版算法，首先第一次随机选取的key都会放入一个pool中(pool的大小为16),pool中的key是按lru大小顺序排列的。
接下来每次随机选取的keylru值必须小于pool中最小的lru才会继续放入，直到将pool放满。放满之后，每次如果有新的key需要放入，需要将pool中lru最大的一个key取出。淘汰的时候，直接从pool中选取一个lru最小的值然后将其淘汰。

4. 单线程的Redis为什么快

- 纯内存操作
- 单线程操作，避免了频繁的上下文切换
- 合理高效的数据结构
- 采用了非阻塞I/O多路复用机制

5. Redis的数据过期策略

Redis 中数据过期策略采用定期删除和惰性删除策略:

- 定期删除策略

Redis 启用一个定时器定时监视所有的 key，判断key是否过期，过期的话就删除。

这种策略可以保证过期的 key 最终都会被删除，但是也存在严重的缺点：每次都遍历内存中所有的数据，非常消耗 CPU资源，并且当 key 已过期，但是定时器还处于未唤起状态，这段时间内 key 仍然可以用。

- 惰性删除策略

在获取 key 时，先判断 key 是否过期，如果过期则删除。

这种方式存在一个缺点：如果这个 key一直未被使用，那么它一直在内存中，其实它已经过期了，会浪费大量的空间。

这两种策略天然的互补，结合起来之后，定时删除策略就发生了一些改变，不在是每次扫描全部的 key 了，而是随机抽取一部分 key 进行检查，这样就降低了对 CPU 资源的损耗，惰性删除策略互补了为检查到的key，基本上满足了所有要求。

但是有时候就是那么的巧，既没有被定时器抽取到，又没有被使用，这些数据又如何从内存中消失？

这个时候就需要用到了,内存淘汰机制.

内存淘汰机制分为：

- 当内存不足以容纳新写入数据时，新写入操作会报错。（Redis 默认策略）
- 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 Key。（LRU推荐使用）
- 当内存不足以容纳新写入数据时，在键空间中，随机移除某个 Key。
- 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 Key。这种情况一般是把 Redis 既当缓存，又做持久化存储的时候才用。
- 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 Key。
- 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 Key 优先移除。

6. 如何解决Redis缓存雪崩问题

- 使用 Redis 高可用架构：使用 Redis 集群来保证 Redis 服务不会挂掉
- 缓存时间不一致，给缓存的失效时间，加上一个随机值，避免集体失效
- 限流降级策略：有一定的备案，比如个性推荐服务不可用了，换成热点数据推荐服务

7. 如何解决Redis缓存穿透问题

- 在接口层做校验
- 存null值（缓存击穿加锁）
- 布隆过滤器拦截：将所有可能的查询key 先映射到布隆过滤器中，查询时先判断key是否存在布隆过滤器中，存在才继续向下执行，如果不存在，则直接返回。
  布隆过滤器将值进行多次哈希bit存储，布隆过滤器说某个元素在，可能会被误判。布隆过滤器说某个元素不在，那么一定不在。

8. Redis并发竞争key如何解决

- 可以利用分布式锁和时间戳来解决.

- 利用消息队列解决.

9. Redis的主从模式和哨兵模式和集群模式区别

Redis集群方式共有三种：主从模式，哨兵模式，集群(cluster)模式

- 主从模式

主从模式是三种集群方式里最简单的。它主要是基于Redis的主从复制特性架构的。通常我们会设置一个主节点，N个从节点;默认情况下，主节点负责处理使用者的IO操作，而从节点则会对主节点的数据进行备份，并且也会对外提供读操作的处理。

主要的特点如下：

1. 主从模式下，当某一节点损坏时，因为其会将数据备份到其它Redis实例上，这样做在很大程度上可以恢复丢失的数据。
2. 主从模式下，可以保证负载均衡.
3. 主从模式下，主节点和从节点是读写分离的。使用者不仅可以从主节点上读取数据，还可以很方便的从从节点上读取到数据，这在一定程度上缓解了主机的压力。

从节点也是能够支持写入数据的，只不过从从节点写入的数据不会同步到主节点以及其它的从节点下。

从以上，我们不难看出Redis在主从模式下，必须保证主节点不会宕机——一旦主节点宕机，其它节点不会竞争称为主节点，此时，Redis将丧失写的能力。这点在生产环境中，是致命的。

- 哨兵模式

哨兵模式是基于主从模式做的一定变化，它能够为Redis提供了高可用性。

在实际生产中，服务器难免不会遇到一些突发状况：服务器宕机，停电，硬件损坏等。这些情况一旦发生，其后果往往是不可估量的。

而哨兵模式在一定程度上能够帮我们规避掉这些意外导致的灾难性后果。其实，哨兵模式的核心还是主从复制。

只不过相对于主从模式在主节点宕机导致不可写的情况下，多了一个竞选机制——从所有的从节点竞选出新的主节点。竞选机制的实现，是依赖于在系统中启动一个sentinel进程。

sentinel特点：

- 监控：它会监听主服务器和从服务器之间是否在正常工作。
- 通知：它能够通过API告诉系统管理员或者程序，集群中某个实例出了问题。
- 故障转移：它在主节点出了问题的情况下，会在所有的从节点中竞选出一个节点，并将其作为新的主节点。
- 提供主服务器地址：它还能够向使用者提供当前主节点的地址。这在故障转移后，使用者不用做任何修改就可以知道当前主节点地址。

sentinel，也可以集群，部署多个哨兵，sentinel可以通过发布与订阅来自动发现Redis集群上的其它sentinel。sentinel在发现其它sentinel进程后，会将其放入一个列表中，这个列表存储了所有已被发现的sentinel。

集群中的所有sentinel不会并发着去对同一个主节点进行故障转移。故障转移只会从第一个sentinel开始，当第一个故障转移失败后，才会尝试下一个。

当选择一个从节点作为新的主节点后，故障转移即成功了(而不会等到所有的从节点配置了新的主节点后)。这过程中，如果重启了旧的主节点，那么就会出现无主节点的情况，这种情况下，只能重启集群。

当竞选出新的主节点后，被选为新的主节点的从节点的配置信息会被sentinel改写为旧的主节点的配置信息。完成改写后，再将新主节点的配置广播给所有的从节点。

- 集群模式

Redis 集群是一个提供在多个Redis间节点间共享数据的程序集, 其中Redis集群分为主节点和从节点。主节点用于处理槽,而从节点用于复制某个主节点，并在被复制的主节点下线时，代替下线的主节点继续处理命令请求。

Redis集群并不支持处理多个keys的命令,因为这需要在不同的节点间移动数据,从而达不到像Redis那样的性能,在高负载的情况下可能会导致不可预料的错误.

Redis 集群通过分区来提供一定程度的可用性,在实际环境中当某个节点宕机或者不可达的情况下继续处理命令. Redis 集群的优势:

自动分割数据到不同的节点上。

整个集群的部分节点失败或者不可达的情况下能够继续处理命令。

Redis集群的数据分片 Redis 集群没有使用一致性hash, 而是引入了哈希槽的概念.

Redis 集群有16384个哈希槽,每个key通过CRC16校验后对16384取模来决定放置哪个槽.集群的每个节点负责一部分hash槽.

例如,当前集群有3个节点,那么:

- 节点 A 包含 0 到 5500号哈希槽.
- 节点 B 包含5501 到 11000 号哈希槽.
- 节点 C 包含11001 到 16384号哈希槽.

这种结构很容易添加或者删除节点. 比如如果我想新添加个节点D, 我需要从节点 A, B, C中得部分槽到D上.

如果我想移除节点A,需要将A中的槽移到B和C节点上,然后将没有任何槽的A节点从集群中移除即可.

由于从一个节点将哈希槽移动到另一个节点并不会停止服务,所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态.

Redis 集群的主从复制模型 为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品.

10. Redis事物的了解CheckAndSet操作实现乐观锁

和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，`MULTI`，`EXEC`，`DISCARD`，`WATCH` 这四个命令是我们实现事务的基石。

相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出 Redis中事务的实现特征:

1.在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

2.和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

3.我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行 `EXEC`，`DISCARD` 命令来提交,回滚该事务内的所有操作。

这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

4.在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

5.当使用 `Append-Only` 模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。

Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。

此时，我们就要充分利用Redis工具包中提供的 `redis-check-aof` 工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

11. Redis有序集合zset底层怎么实现的

Redis中的set数据结构底层用的是跳表实现的.

跳表是一个随机化的数据结构，实质就是一种可以进行二分查找的有序链表。

跳表在原有的有序链表上面增加了多级索引，通过索引来实现快速查找。

跳表不仅能提高搜索性能，同时也可以提高插入和删除操作的性能。

(1)跳表是可以实现二分查找的有序链表；
(2)每个元素插入时随机生成它的level；
(3)最低层包含所有的元素；
(4)如果一个元素出现在level(x)，那么它肯定出现在x以下的level中；
(5)每个索引节点包含两个指针，一个向下，一个向右；
(6)跳表查询、插入、删除的时间复杂度为O(log n)，与平衡二叉树接近；

为什么Redis选择使用跳表而不是红黑树来实现有序集合？(O(logN))

首先，我们来分析下Redis的有序集合支持的操作：

- 插入元素
- 删除元素
- 查找元素
- 有序输出所有元素
- 查找区间内所有元素

其中，前4项红黑树都可以完成，且时间复杂度与跳表一致。但是，最后一项，红黑树的效率就没有跳表高了。
在跳表中，要查找区间的元素，我们只要定位到两个区间端点在最低层级的位置，然后按顺序遍历元素就可以了，非常高效。

而红黑树只能定位到端点后，再从首位置开始每次都要查找后继节点，相对来说是比较耗时的。
此外，跳表实现起来很容易且易读，红黑树实现起来相对困难，所以Redis选择使用跳表来实现有序集合。

12. 跳表的查询过程是怎么样的，查询和插入的时间复杂度

先从第一层查找，不满足就下沉到第二层找，因为每一层都是有序的，写入和插入的时间复杂度都是O(logN)

网络协议基础

1. TCP和UDP有什么区别

TCP与UDP区别总结：

1、TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接.
2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付.

3、TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的.
UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）.
4、每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信.
5、TCP首部开销20字节;UDP的首部开销小，只有8个字节.
6、TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道.

因此UDP不提供复杂的控制机制，利用IP提供面向无连接的通信服务，随时都可以发送数据，处理简单且高效.

经常用于以下场景：

- 包总量较小的通信（DNS、SNMP）
- 视频、音频等多媒体通信（即时通信）
- 广播通信

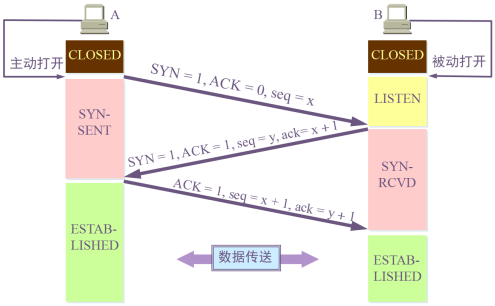
TCP 使用场景:

相对于 UDP，TCP 实现了数据传输过程中的各种控制，可以进行丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

在对可靠性要求较高的情况下，可以使用 TCP，即不考虑 UDP 的时候，都可以选择 TCP。

2. TCP中三次握手和四次挥手

- 三次握手



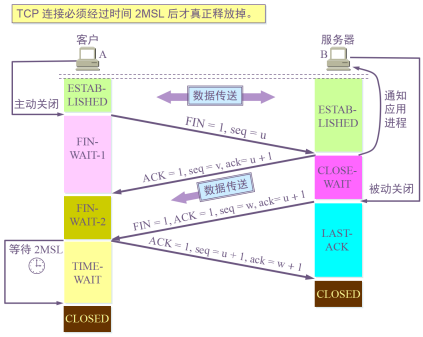
假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文段，`SYN=1`，`ACK=0`，选择一个初始的序号 `seq = x`。
- B 收到连接请求报文段，如果同意建立连接，则向 A 发送连接确认报文段，`SYN=1`，`ACK=1`，确认号为 `x+1`，同时也选择一个初始的序号 `seq = y`。
- A 收到 B 的连接确认报文段后，还要向 B 发出确认，确认号为 `ack = y+1`，序号为 `seq = x+1`。
- A 的 TCP 通知上层应用进程，连接已经建立。
- B 收到 A 的确认后，连接建立。
- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。

为什么TCP连接需要三次握手，两次不可以吗，为什么?

TCP是一个双向通信协议，通信双方都有能力发送信息，并接收响应。如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认

- 四次挥手



数据传输结束后，通信的双方都可释放连接。现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP连接。

- A 把连接释放报文段首部的 `FIN = 1`，其序号 `seq = u`，等待 B 的确认。
- B 发出确认，确认号 `ack = u+1`，而这个报文段自己的序号 `seq = v`。（TCP 服务器进程通知高层应用进程）。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。A 不能向 B 发送数据；B 若发送数据，A 仍要接收。
- 当 B 不再需要连接时，发送连接释放请求报文段，FIN=1。
- A 收到后发出确认，进入 `TIME-WAIT` 状态，等待 `2 MSL（2*2 = 4 mins）` 时间后释放连接。
- B 收到 A 的确认后释放连接。

四次挥手的原因:

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 `CLOSE-WAIT` 状态。

这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

3. **TCP的LISTEN状态是什么**

TCP的LISTEN是服务器处于监听状态:

- CLOSED：初始状态。
- LISTEN：服务器处于监听状态。
- TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。

4. **常见的HTTP状态码有哪些**

状态码	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

5. **301和302有什么区别**

301: Moved Permanently 被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则

这个响应也是可缓存的。

302 Found 请求的资源现在临时从不同的URI响应请求。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。

301是永久重定向，而302是临时重定向。

6. 504和500有什么区别

500的错误通常是由于服务器上代码出错或者是抛出了异常.

502即 Bad Gateway网关(这里的网关是指CGI,即通用网关接口)错误,通常是程序空指针错误。

504即Gateway timeout,即超时错误.

7. HTTPS和HTTP有什么区别

http协议和https协议的区别：传输信息安全性不同、连接方式不同、端口不同、证书专申请方式不同.

一、传输信息安全性不同

1. http协议：是超文本传输协议，信息是明文传输。如果攻击者截取了Web浏览器和网站服务器之间的传输报文，就可以直接读懂其中的信息。

2. https协议：是具有安全性的ssl加密传输协议，为浏览器和服务器之间的通信加密，确保数据传输的安

二,连接方式不同

1. http协议：http的连接很简单，是无状态的。
2. https协议：是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议。

三、端口不同

1. http协议：使用的端口是80。
2. https协议：使用的端口是443.

四、证书申请方式不同

1. http协议：免费申请。

2. https协议：需要到ca申请证书，一般免费证书很少，需要交费。

3. Quic有什么优点相比Http2

- `HTTP1` 有连接无法复用、队头阻塞、协议开销大和安全因素等多个缺陷.
- `HTTP2` 通过多路复用、二进制流、Header 压缩等等技术，极大地提高了性能，但是还是存在着问题的
- `Quic` 基于 UDP 实现，是 HTTP3 中的底层支撑协议，该协议基于 UDP，又取了 TCP 中的精华，实现了即快又可靠的协议.quic中加密认证的报文,(TCP 协议头部没有经过任何加密和认证，所以在传输过程中很容易被中间网络设备篡改，注入和窃听。比如修改序列号、滑动窗口。这些行为有可能是出于性能优化，也有可能是主动攻击。)这样只要对 QUIC 报文任何修改，接收端都能够及时发现，有效地降低了安全风险。

此外quic还有向前纠错的能力,QUIC 协议有一个非常独特的特性，称为向前纠错 (Forward Error Correction，FEC)，每个数据包除了它本身的内容之外，还包括了部分其他数据包的数据，因此少量的丢包可以通过其他包的冗余数据直接组装而无需重传。

向前纠错牺牲了每个数据包可以发送数据的上限，但是减少了因为丢包导致的数据重传，因为数据重传将会消耗更多的时间(包括确认数据包丢失、请求重传、等待新数据包等步骤的时间消耗),

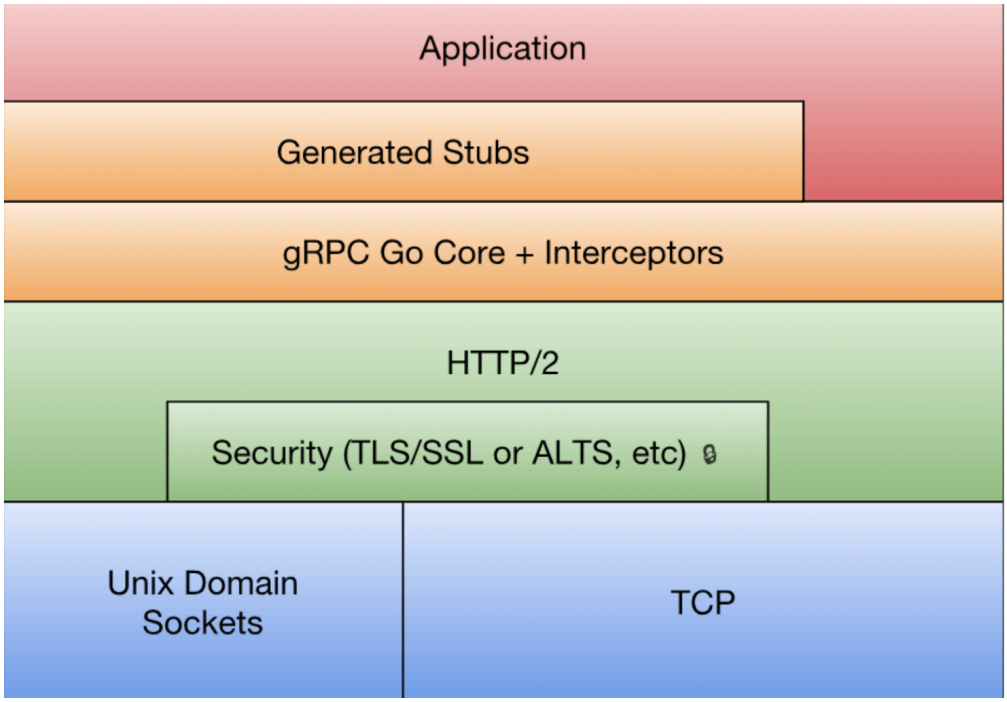
假如说这次我要发送三个包，那么协议会算出这三个包的异或值并单独发出一个校验包，也就是总共发出了四个包。当出现其中的非校验包丢包的情况时，可以通过另外三个包计算出丢失的数据包的内容。
当然这种技术只能使用在丢失一个包的情况下，如果出现丢失多个包就不能使用纠错机制了，只能使用重传的方式了。

Quic 相比现在广泛应用的 http2+tcp+tls 协议有如下优势:

减少了 TCP 三次握手及 TLS 握手时间。改进的拥塞控制。避免队头阻塞的多路复用。连接迁移。前向冗余纠错.

9. **Grpc的优缺点**

gRPC是Google公司基于Protobuf开发的跨语言的开源RPC框架。gRPC基于HTTP/2协议设计，可以基于一个HTTP/2链接提供多个服务，对于移动设备更加友好。



最底层为TCP或Unix Socket协议，在此之上是HTTP/2协议的实现，然后在HTTP/2协议之上又构建了针对Go语言的gRPC核心库。应用程序通过gRPC插件生产的Stub代码和gRPC核心库通信，也可以直接和gRPC核心库通信。

Grpc优缺点：

优点：

- protobuf二进制消息，性能好/效率高（空间和时间效率都很不错）
- proto文件生成目标代码，简单易用
- 序列化反序列化直接对应程序中的数据类，不需要解析后在进行映射(XML,JSON都是这种方式)
- 支持向前兼容（新加字段采用默认值）和向后兼容（忽略新加字段），简化升级
- 支持多种语言（可以把proto文件看做IDL文件）

缺点：

- GRPC尚未提供连接池，需要自行实现
- 尚未提供“服务发现”、“负载均衡”机制
- 因为基于HTTP2，绝大部多数 `HTTP Server、Nginx` 都尚不支持，即Nginx不能将GRPC请求作为HTTP请求来负载均衡，而是作为普通的TCP请求。（nginx1.9版本已支持）

- Protobuf二进制可读性差（貌似提供了Text_Fromat功能）默认不具备动态特性（可以通过动态定义生成消息类型或者动态编译支持）

10. Get和Post区别

Get和Post的区别和不同:

1. Get是不安全的，因为在传输过程，数据被放在请求的URL中；Post的所有操作对用户来说都是不可见的。
2. Get传送的数据量较小，这主要是因为受URL长度限制；Post传送的数据量较大，一般被默认为不受限制。
3. Get限制Form表单的数据集的值必须为ASCII字符；而Post支持整个ISO10646字符集。
4. Get执行效率却比Post方法好。Get是form提交的默认方法。
5. GET产生一个TCP数据包；POST产生两个TCP数据包。（非必然，客户端可灵活决定）

6. Unicode和ASCII以及Utf8的区别

计算机内部，所有信息最终都是一个二进制值。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。

也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从00000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为 ASCII 码，一直沿用至今。

ASCII 码一共规定了128个字符的编码，比如空格SPACE是32（二进制00100000），大写的字母A是65（二进制01000001）。

这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的一位统一规定为0。

- Unicode 是字符集

  如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是 Unicode，就像它的名字都表示的，这是一种所有符号的编码。

  Unicode 当然是一个很大的集合，现在的规模可以容纳100多万个符号。

- UTF-8 是编码规则

Unicode 只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8 就是在互联网上使用最广的一种 Unicode 的实现方式。其他实现方式还包括 UTF-16（字符用两个字节或四个字节表示）和 UTF-32（字符用四个字节表示），不过在互联网上基本不用。
这里需要注意下，这里的关系是，UTF-8 是 Unicode 的实现方式之一。

12. Cookie与Session异同

Cookie 和 Session 都为了用来保存状态信息，都是保存客户端状态的机制，它们都是为了解决HTTP无状态的问题而所做的努力。

- Cookie机制

简单地说，Cookie 就是浏览器储存在用户电脑上的一小段文本文件。Cookie 是纯文本格式，不包含任何可执行的代码。
一个 Web 页面或服务器告知浏览器按照一定规范来储存这些信息，并在随后的请求中将这些信息发送至服务器，Web 服务器就可以使用这些信息来识别不同的用户。

大多数需要登录的网站在用户验证成功之后都会设置一个 Cookie，只要这个 Cookie 存在并可以，用户就可以自由浏览这个网站的任意页面。

Cookie 会被浏览器自动删除，通常存在以下几种原因：

1. 会话 Cooke (Session Cookie) 在会话结束时（浏览器关闭）会被删除
2. 持久化 Cookie（Persistent Cookie）在到达失效日期时会被删除
3. 如果浏览器中的 Cookie 数量达到限制，那么 Cookie 会被删除以为新建的 Cookie 创建空间。

大多数浏览器支持最大为 4096 字节的 Cookie。由于这限制了 Cookie 的大小，最好用 Cookie 来存储少量数据，或者存储用户 ID 之类的标识符。

用户 ID 随后便可用于标识用户，以及从数据库或其他数据源中读取用户信息。 浏览器还限制站点可以在用户计算机上存储的 Cookie 的数量。

大多数浏览器只允许每个站点存储 20 个 Cookie；如果试图存储更多 Cookie，则最旧的 Cookie 便会被丢弃。

有些浏览器还会对它们将接受的来自所有站点的 Cookie 总数作出绝对限制，通常为 300 个。

使用 Cookie 的缺点：

1. 不良站点用 Cookie 收集用户隐私信息；

2. Cookie窃取：黑客以可以通过窃取用户的cookie来模拟用户的请求行为。（跨站脚本攻击XSS）

3. Session 机制

Session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。当程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端的请求里是否已包含了一个Session标识（Session id）.

如果已包含一个SessionID 则说明以前已经为此客户端创建过Session，服务器就按照SessionID把这个 Session检索出来使用（如果检索不到，可能会新建一个）。
如果客户端请求不包含SessionID，则为此客户端创建一个Session并且生成一个与此Session相关联的SessionID，SessionID的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个SessionID将被在本次响应中返回给客户端保存。

具体实现方式：

- Cookie方式

服务器给每个Session分配一个唯一的JSESSIONID，并通过Cookie发送给客户端。

当客户端发起新的请求的时候，将在Cookie头中携带这个JSESSIONID，这样服务器能够找到这个客户端对应的Session。

- URL回写

服务器在发送给浏览器页面的所有链接中都携带JSESSIONID的参数，这样客户端点击任何一个链接都会把JSESSIONID带回服务器。如果直接在浏览器输入服务端资源的url来请求该资源，那么Session是匹配不到的。

Web 缓存:

WEB缓存(cache)位于Web服务器和客户端之间，缓存机制会根据请求保存输出内容的副本，例如html页面，图片，文件，当下一个请求来到的时候：如果是相同的URL，缓存直接使用副本响应访问请求，而不是向源服务器再次发送请求。

主要分三种情况:

1. 未找到缓存(黑色线)：当没有找到缓存时，说明本地并没有这些数据，这种情况一般发生在我们首次访问网站，或者以前访问过，但是清除过缓存后。

浏览器就会先访问服务器，然后把服务器上的内容取回来，内容取回来以后，就要根据情况来决定是否要保留到缓存中了。

2. 缓存未过期(蓝色线)：缓存未过期，指的是本地缓存没有过期，不需要访问服务器了，直接就可以拿本地的缓存作为响应在本地使用了。这样节省了不少网络成本，提高了用户体验过。

3. 缓存已过期(红色线)：当满足过期的条件时，会向服务器发送请求，发送的请求一般都会进行一个验证，目的是虽然缓存文档过期了，但是文档内容不一定会有什么改变，所以服务器返回的也许是一个新的文档，这时候的HTTP状态码是200，或者返回的只是一个最新的时间戳和304状态码。

缓存过期后，有两种方法来判定服务端的文件有没有更新。

第一种在上一次服务端告诉客户端约定的有效期的同时，告诉客户端该文件最后修改的时间，当再次试图从服务端下载该文件的时候，check下该文件有没有更新（对比最后修改时间），如果没有，则读取缓存.

第二种方式是在上一次服务端告诉客户端约定有效期的同时，同时告诉客户端该文件的版本号，当服务端文件更新的时候，改变版本号，再次发送请求的时候check一下版本号是否一致就行了，如一致，则可直接读取缓存。

浏览器是依靠请求和响应中的的头信息来控制缓存的，如下：

- Expires与Cache-Control：服务端用来约定和客户端的有效时间的。

Expires规定了缓存失效时间（Date为当前时间），而 `Cache-Control` 的max-age规定了缓存有效时间（2552s）。

Expires是HTTP1.0的东西，而 `Cache-Control` 是HTTP1.1的，规定如果 `max-age` 和Expires同时存在，前者优先级高于后者。

- Last-Modified/If-Modified-Since缓存过期后，check服务端文件是否更新的第一种方式。

- ETag/If-None-Match：缓存过期时check服务端文件是否更新的第二种方式。

实际上ETag并不是文件的版本号，而是一串可以代表该文件唯一的字符串，当客户端发现和服务器约定的直接读取缓存的时间过了，就在请求中发送If-None-Match选项，值即为上次请求后响应头的ETag值.
该值在服务端和服务端代表该文件唯一的字符串对比（如果服务端该文件改变了，该值就会变），如果相同，则相应HTTP304，客户端直接读取缓存，如果不相同，HTTP200，下载正确的数据，更新ETag值。

当然并不是所有请求都能被缓存。无法被浏览器缓存的请求：

1. HTTP信息头中包含 `Cache-Control:no-cache，pragma:no-cache（HTTP1.0）` ，或 `Cache-Control:max-age=0` 等告诉浏览器不用缓存的请求
2. 需要根据Cookie，认证信息等决定输入内容的动态请求是不能被缓存的
3. POST请求无法被缓存

浏览器缓存过程还和用户行为有关。譬如先打开一个主页有个jquery的请求（假设访问后会缓存下来）。

接着如果直接在地址栏输入 jquery 地址，然后回车，响应HTTP200（from cache），因为有效期还没过直接读取的缓存；如果 `ctrl+r` 进行刷新，则会相应HTTP304（Not Modified），虽然还是读取的本地缓存，但是多了一次服务端的请求；而如果是ctrl+shift+r强刷，则会直接从服务器下载新的文件，响应HTTP200。

13. Client如何实现长连接

TCP协议的 `KeepAlive` 机制与 `HeartBeat` 心跳包

- HeartBeat心跳包

很多应用层协议都有HeartBeat机制，通常是客户端每隔一小段时间向服务器发送一个数据包，通知服务器自己仍然在线，并传输一些可能必要的数据。使用心跳包的典型协议是IM，比如QQ/MSN/飞信等协议。

心跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。事实上这是为了保持长连接，至于这个包的内容，是没有什么特别规定的，不过一般都是很小的包，或者只包含包头的一个空包。

在TCP的机制里面，本身是存在有心跳包的机制的，也就是TCP的选项：SO_KEEPALIVE。系统默认是设置的2小时的心跳频率。但是它检查不到机器断电、网线拔出、防火墙这些断线。而且逻辑层处理断线可能也不是那么好处理。一般，如果只是用于保活还是可以的。

心跳包一般来说都是在逻辑层发送空的echo包来实现的。下一个定时器，在一定时间间隔下发送一个空包给客户端，然后客户端反馈一个同样的空包回来，服务器如果在一定时间内收不到客户端发送过来的反馈包，那就只有认定说掉线了。

其实，要判定掉线，只需要send或者recv一下，如果结果为零，则为掉线。但是，在长连接下，有可能很长一段时间都没有数据往来。

理论上说，这个连接是一直保持连接的，但是实际情况中，如果中间节点出现什么故障是难以知道的。更要命的是，有的节点（防火墙）会自动把一定时间之内没有数据交互的连接给断掉。在这个时候，就需要我们的心跳包了，用于维持长连接，保活。

在获知了断线之后，服务器逻辑可能需要做一些事情，比如断线后的数据清理呀，重新连接呀.....当然，这个自然是要由逻辑层根据需求去做了。

总的来说，心跳包主要也就是用于长连接的保活和断线处理。一般的应用下，判定时间在30-40秒比较不错。如果实在要求高，那就在6-9秒。

- TCP协议的 `KeepAlive` 机制

TCP的IP传输层的两个主要协议是UDP和TCP，其中UDP是无连接的、面向packet的，而TCP协议是有连接、面向流的协议。

TCP的 `KeepAlive` 机制，首先它貌似默认是不打开的，要用 `setsockopt` 将 `SOL_SOCKET.SO_KEEPALIVE` 设置为1才是打开，并且可以设置三个参数 `tcp_keepalive_time/tcp_keepalive_probes/tcp_keepalive_intvl` ，分别表示连接闲置多久开始发 `keepalive` 的ack包、发几个ack包不回复才当对方死了、两个ack包之间间隔多.

在测试的时候用 `Ubuntu Server 10.04` 下面默认值是7200秒（2个小时，要不要这么蛋疼啊！）、9次、75秒。

于是连接就了有一个超时时间窗口，如果连接之间没有通信，这个时间窗口会逐渐减小，当它减小到零的时候，TCP协议会向对方发一个带有ACK标志的空数据包（KeepAlive探针），对方在收到ACK包以后，如果连接一切正常，应该回复一个ACK；如果连接出现错误了（例如对方重启了，连接状态丢失），则应当回复一个RST；如果对方没有回复，服务器每隔intvl的时间再发ACK，如果连续probes个包都被无视了，说明连接被断开了。

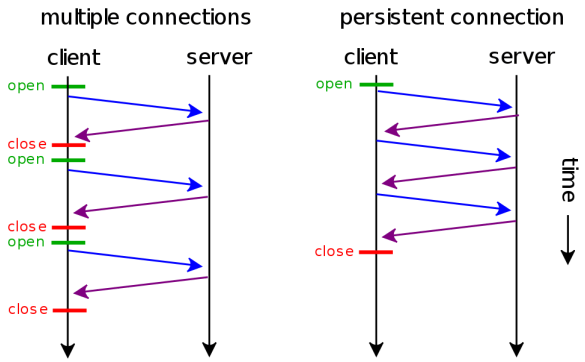
在http早期，每个http请求都要求打开一个tpc socket连接，并且使用一次之后就断开这个tcp连接。

使用 `keep-alive` 可以改善这种状态，即在一次TCP连接中可以持续发送多份数据而不会断开连接。通过使用 `keep-alive` 机制，可以减少tcp连接建立次数，也意味着可以减少 `TIME_WAIT` 状态连接，以此提高性能和提高httpd服务器的吞吐率(更少的tcp连接意味着更少的系统内核调用,socket的 `accept()` 和 `close()` 调用)。

但是， `keep-alive` 并不是免费的午餐,长时间的tcp连接容易导致系统资源无效占用。配置不当的 `keep-alive` ，有时比重复利用连接带来的损失还更大。所以，正确地设置 `keep-alive timeout` 时间非常重要。

使用 `http keep-alvie` ，可以减少服务端 `TIME_WAIT` 数量(因为由服务端httpd守护进程主动关闭连接)。道理很简单，相较而言，启用 `keep-alive` ，建立的tcp连接更少了，自然要被关闭的tcp连接也相应更少了。

使用启用 `keepalive` 的不同。另外， `http keepalive` 是客户端浏览器与服务端httpd守护进程协作的结果，所以，我们另外安排篇幅介绍不同浏览器的各种情况对 `keep-alive` 的利用。



14. Http1和Http2和Grpc之间的区别是什么

在互联网流量传输只使用了几个网络协议。使用 `IPv4` 进行路由，使用 `TCP` 进行连接层面的流量控制，使用 `SSL/TLS` 协议实现传输安全，使用 `DNS` 进行域名解析，使用 `HTTP` 进行应用数据的传输。

但是使用Http进行应用数据的传输,却是在不断的改变,那么Http1和Http2和Grpc之间的区别是什么,我们下面分析下.

通常影响一个 HTTP 网络请求的因素主要有两个：带宽和延迟。

- 带宽

如果说我们还停留在拨号上网的阶段，带宽可能会成为一个比较严重影响请求的问题，但是现在网络基础建设已经使得带宽得到极大的提升，我们不再会担心由带宽而影响网速，那么就只剩下延迟了。

- 延迟

浏览器阻塞（HOL blocking）：浏览器会因为一些原因阻塞请求。浏览器对于同一个域名，同时只能有 4 个连接（这个根据浏览器内核不同可能会有所差异），超过浏览器最大连接数限制，后续请求就会被阻塞。

DNS 查询（DNS Lookup）：浏览器需要知道目标服务器的 IP 才能建立连接。将域名解析为 IP 的这个系统就是 DNS。这个通常可以利用DNS缓存结果来达到减少这个时间的目的。

建立连接（Initial connection）：HTTP 是基于 TCP 协议的，浏览器最快也要在第三次握手时才能捎带 HTTP 请求报文，达到真正的建立连接，但是这些连接无法复用会导致每次请求都经历三次握手和慢启动。三次握手在高延迟的场景下影响较明显，慢启动则对文件类大请求影响较大。

然而,HTTP2并不是对HTTP1协议的重写，相对于HTTP1，HTTP2 的侧重点主要在性能。其中请求方法，状态码和语义和HTTP1都是相同的，可以使用与 HTTP1相同的 API（可能有一些小的添加）来表示协议。

HTTP2主要有两个规范组成:

- `Hypertext Transfer Protocol version 2` (超文本传输协议版本 2)
- `HPACK - HTTP2` 的头压缩 （HPACK 是一种头部压缩算法）

HTTP2和HTTP1相比的新特性包括:

- 新的二进制格式（Binary Format）

HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。

- 多路复用（MultiPlexing）

连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。

- Header压缩

Header压缩，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。

- 服务端推送（server push）

服务端推送（server push），同SPDY一样，HTTP2.0也具有server push功能。

Grpc的设计目标是在任何环境下运行，支持可插拔的负载均衡，跟踪，运行状况检查和身份验证。它不仅支持数据中心内部和跨数据中心的服务调用，它也适用于分布式计算的最后一公里，将设备，移动应用程序和浏览器连接到后端服务，同时，它也是高性能的，而 HTTP2 恰好支持这些。

而Grpc是基于http2的.

- HTTP2天然的通用性满足各种设备，场景.
- HTTP2的性能相对来说也是很好的，除非你需要极致的性能.
- HTTP2的安全性非常好，天然支持 SSL.
- HTTP2的鉴权也非常成熟.
- Grpc基于 HTTP2 多语言实现也更容易.

15. Tcp中的拆包和粘包是怎么回事

拆包和粘包是在socket编程中经常出现的情况，在socket通讯过程中，如果通讯的一端一次性连续发送多条数据包，tcp协议会将多个数据包打包成一个tcp报文发送出去，这就是所谓的粘包。

而如果通讯的一端发送的数据包超过一次tcp报文所能传输的最大值时，就会将一个数据包拆成多个最大tcp长度的tcp报文分开传输，这就叫做拆包。

MTU:

泛指通讯协议中的最大传输单元。一般用来说明TCP/IP四层协议中数据链路层的最大传输单元，不同类型的网络MTU也会不同，我们普遍使用的以太网的MTU是1500，即最大只能传输1500字节的数据帧。可以通过ifconfig命令查看电脑各个网卡的MTU。

MSS:

指TCP建立连接后双方约定的可传输的最大TCP报文长度，是TCP用来限制应用层可发送的最大字节数。如果底层的MTU是1500byte，则 MSS = 1500- 20(IP Header) -20 (TCP Header) = 1460 byte。

16. TFO的原理是什么

TCP快速打开（TCP Fast Open，TFO）是对TCP的一种简化握手手续的拓展，用于提高两端点间连接的打开速度。
简而言之，就是在TCP的三次握手过程中传输实际有用的数据。这个扩展最初在Linux系统实现，Linux服务器，Linux系统上的Chrome浏览器，或运行在Linux上的其他支持的软件。
它通过握手开始时的SYN包中的TFO cookie来验证一个之前连接过的客户端。如果验证成功，它可以在三次握手最终的ACK包收到之前就开始发送数据，这样便跳过了一个绕路的行为，更在传输开始时就降低了延迟。
这个加密的Cookie被存储在客户端，在一开始的连接时被设定好。然后每当客户端连接时，这个Cookie被重复返回。

请求Tcp Fast Open Cookie

- 客户端发送SYN数据包，该数据包包含Fast Open选项，且该选项的Cookie为空，这表明客户端请求Fast Open Cookie；

- 支持TCP Fast Open的服务器生成Cookie，并将其置于SYN-ACK数据包中的Fast Open选项以发回客户端；
- 客户端收到SYN-ACK后，缓存Fast Open选项中的Cookie。

17. TIME_WAIT的作用

主动关闭的Socket端会进入TIME_WAIT状态，并且持续2MSL时间长度，MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。MSL在RFC 1122上建议是2分钟，而源自berkeley的TCP实现传统上使用30秒，因而，TIME_WAIT状态一般维持在1-4分钟。

- 可靠地实现TCP全双工连接的终止

在进行关闭连接四路握手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允 许它重发最终的ACK。

如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在java中会抛出connection reset的SocketException)。

因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭 的客户端必须维持状态信息进入TIME_WAIT状态。

- 允许老的重复分节在网络中消逝

TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个 原来的迷途分节就称为lost duplicate。

在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP连接，后一个连接被称为前一个连接的化身（incarnation)，那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。

为了避免这个情况，TCP不允许处于TIME_WAIT状态的连接启动一个新的化身，因为TIME_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时 候，来自连接先前化身的重复分组已经在网络中消逝。

18. 网络的性能指标有哪些

通常是以4个指标来衡量网络的性能，分别是带宽、延时、吞吐率、PPS（Packet Per Second），它们表示的意义如下：

- 带宽，表示链路的最大传输速率，单位是 `b/s` （比特 / 秒），带宽越大，其传输能力就越强。
- 延时，表示请求数据包发送后，收到对端响应，所需要的时间延迟。不同的场景有着不同的含义，比如可以表示建立 `TCP` 连接所需的时间延迟，或一个数据包往返所需的时间延迟。
- 吞吐率，表示单位时间内成功传输的数据量，单位是 `b/s` （比特 / 秒）或者 `B/s` （字节 / 秒），吞吐受带宽限制，带宽越大，吞吐率的上限才可能越高。
- `PPS` ，全称是 `Packet Per Second` （包 / 秒），表示以网络包为单位的传输速率，一般用来评估系统对于网络的转发能力。

当然，除了以上这四种基本的指标，还有一些其他常用的性能指标，比如：

- 网络的可用性，表示网络能否正常通信；
- 并发连接数，表示 TCP 连接数量；
- 丢包率，表示所丢失数据包数量占所发送数据组的比率；

- 重传率，表示重传网络包的比例；

Linux基础

1. 异步和非阻塞的区别

异步和非阻塞的区别:

- 异步：调用在发出之后，这个调用就直接返回，不管有无结果；异步是过程。
- 非阻塞：关注的是程序在等待调用结果（消息，返回值）时的状态，指在不能立刻得到结果之前，该调用不会阻塞当前线程。

同步和异步的区别：

- 同步：一个服务的完成需要依赖其他服务时，只有等待被依赖的服务完成后，才算完成，这是一种可靠的服务序列。要么成功都成功，失败都失败，服务的状态可以保持一致。
- 异步：一个服务的完成需要依赖其他服务时，只通知其他依赖服务开始执行，而不需要等待被依赖的服务完成，此时该服务就算完成了。被依赖的服务是否最终完成无法确定，一次它是一个不可靠的服务序列。

消息通知中的同步和异步：

- 同步：当一个同步调用发出后，调用者要一直等待返回消息（或者调用结果）通知后，才能进行后续的执行。
- 异步：当一个异步过程调用发出后，调用者不能立刻得到返回消息（结果）。在调用结束之后，通过消息回调来通知调用者是否调用成功。

阻塞与非阻塞的区别：

- 阻塞：阻塞调用是指调用结果返回之前，当前线程会被挂起，一直处于等待消息通知，不能够执行其他业务,函数只有在得到结果之后才会返回。
- 非阻塞：非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

同步与异步是对应的，它们是线程之间的关系，两个线程之间要么是同步的，要么是异步的。

阻塞与非阻塞是对同一个线程来说的，在某个时刻，线程要么处于阻塞，要么处于非阻塞。

阻塞是使用同步机制的结果，非阻塞则是使用异步机制的结果。

2. 虚拟内存作用是什么

我们都知道一个进程是与其他进程共享CPU和内存资源的。正因如此，操作系统需要有一套完善的内存管理机制才能防止进程之间内存泄漏的问题.

为了更加有效地管理内存并减少出错，现代操作系统提供了一种对主存的抽象概念，即是虚拟内存（Virtual Memory）。虚拟内存为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉（每个进程拥有一片连续完整的内存空间）。

虚拟内存的重要意义是它定义了一个连续的虚拟地址空间，使得程序的编写难度降低。并且，把内存扩展到硬盘空间只是使用虚拟内存的必然结果，虚拟内存空间会存在硬盘中，并且会被内存缓存（按需），有的操作系统还会在内存不够的情况下，将某一进程的内存全部放入硬盘空间中，并在切换到该进程时再从硬盘读取.

虚拟内存主要提供了如下三个重要的能力：

- 它把主存看作为一个存储在硬盘上的虚拟地址空间的高速缓存，并且只在主存中缓存活动区域（按需缓存）。

- 它为每个进程提供了一个一致的地址空间，从而降低了程序员对内存管理的复杂性。
- 它还保护了每个进程的地址空间不会被其他进程破坏。

3. Linux查看端口占用和cpu负载

linux ps命令，查看某进程cpu和内存占用率情况:

```
> ps aux
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
admin            72824  17.3  1.4  5518204 118212   ??  R    27 519   54:49.93 /Applications/iTerm.app/Conten
ts/MacOS/iTerm2
_windowserver      179  16.1  0.6  7525352  46552   ??  Rs   21 519  457:09.25 /System/Library/PrivateFramewo
rks/SkyLight.fra
admin              734  12.2  3.3  6095348 273108   ??  R    21 519  635:17.25 /Users/admin/Desktop/Google Ch
rome.app/Content
admin            10718   9.0  2.7  5604388 223604   ??  S    22 519  557:56.89 /Users/admin/Desktop/Google Ch
rome.app/Content
admin              750   6.4  0.6  4633300  52372   ??  S    21 519  147:59.59 /Users/admin/Desktop/Google Ch
rome.app/Content
admin              749   5.6  1.2  5570904  96832   ??  S    21 519  359:56.37 /Users/admin/Desktop/Google Ch
rome.app/Content
admin              818   4.5  0.1  6557980   5508   ??  S    21 519  557:27.52 com.docker.hyperkit -A -u -F v
ms/0/hyperkit.pi
admin            32898   3.5  1.4  4977204 117684   ??  S    10:54上午   0:02.27 /Users/admin/Desktop/Google
 Chrome.app/Content
admin            30591   2.2  3.7  9505844 310584   ??  S     9:47上午  10:49.28 /Applications/GoLand.app/Con
tents/MacOS/goland
root              1300   1.9  0.1  4334916   6212   ??  Ss   21 519  123:53.86 /usr/libexec/taskgated
admin            31232   1.2  1.1 10553808  88860   ??  S    10:24上午   3:28.67 /Applications/WebStorm.app/C
ontents/MacOS/webs
admin            18704   0.7  0.2 19282032  12948   ??  S     3:56下午   4:18.12 /private/var/folders/kp/3yqn
p9cj4f3_9539b06q4
```

- linux 下的ps命令
- USER 进程运行用户
- PID 进程编号
- %CPU 进程的cpu占用率
- %MEM 进程的内存占用率
- VSZ 进程所使用的虚存的大小
- RSS 进程

操作系统解析

操作系统解析

操作系统(operating system)是管理计算机硬件与软件资源的计算机程序，同时也是计算机系统的内核与基石。操作系统需要处理如管理与配置内存、决定系统资源供需的优先次序、控制输入与输出设备、操作网络与管理文件系统等基本事务。操作系统也提供一个让用户与系统交互的操作界面。

操作系统的类型非常多样，不同机器安装的操作系统可从简单到复杂，可从移动电话的嵌入式系统到超级计算机的大型操作系统。许多操作系统制造者对它涵盖范畴的定义也不尽一致，例如有些操作系统集成了图形用户界面，而有些仅使用命令行界面，而将图形用户界面视为一种非必要的应用程序。

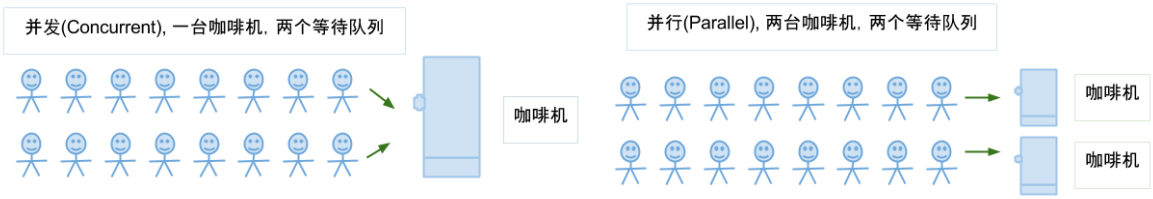
接着我们对操作系统详细解析如下:

操作系统基本特征

1. 并发和并行

并发是指宏观上在一段时间内能同时运行多个程序，而并行则指同一时刻能运行多个指令。
并行需要硬件支持，如多流水线或者多处理器。

操作系统通过引入进程和线程，使得程序能够并发运行。



2. 共享

共享是指系统中的资源可以被多个并发进程共同使用。

有两种共享方式：互斥共享和同时共享。

互斥共享的资源称为临界资源，例如打印机等，在同一时间只允许一个进程访问，需要用同步机制来实现对临界资源的访问。

3. 虚拟

虚拟技术把一个物理实体转换为多个逻辑实体。

利用多道程序设计技术，让每个用户都觉得有一个计算机专门为他服务。

主要有两种虚拟技术：时分复用技术和空分复用技术。例如多个进程能在同一个处理器上并发执行使用了时分复用技术，让每个进程轮流占有处理器，每次只执行一小个时间片并快速切换。

4. 异步

异步指进程不是一次性执行完毕，而是走走停停，以不可知的速度向前推进。

但只要运行环境相同，OS需要保证程序运行的结果也要相同。

操作系统基本功能

1. 进程管理

进程控制、进程同步、进程通信、死锁处理、处理机调度等。

2. 内存管理

内存分配、地址映射、内存保护与共享、虚拟内存等。

3. 文件管理

文件存储空间的管理、目录管理、文件读写管理和保护等。

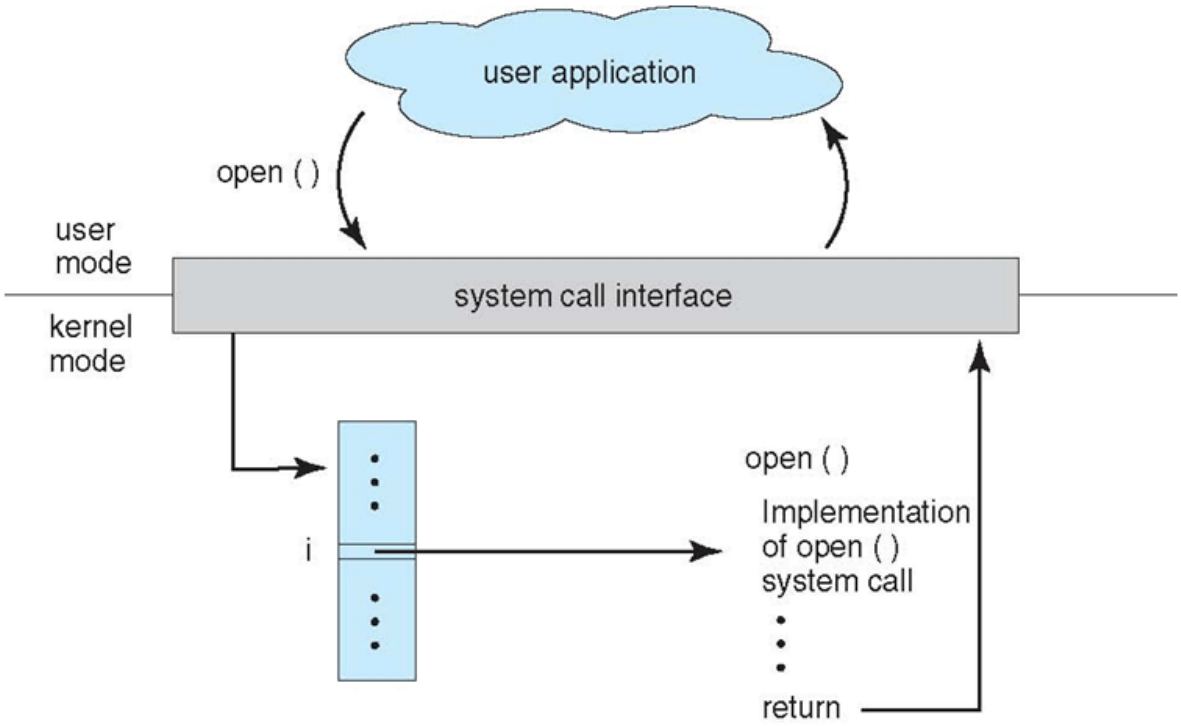
4. 设备管理

完成用户的 I/O 请求，方便用户使用各种设备，并提高设备的利用率。

主要包括缓冲管理、设备分配、设备处理、虚拟设备等。

系统调用

如果一个进程在用户态需要使用内核态的功能，就进行系统调用从而陷入内核，由操作系统代为完成。



Linux 的系统调用主要有以下这些：

Task	Commands
进程控制	fork(); exit(); wait();
进程通信	pipe(); shmget(); mmap();
文件操作	open(); read(); write();
设备操作	ioctl(); read(); write();
信息维护	getpid(); alarm(); sleep();
安全	chmod(); umask(); chown();

大内核和微内核

- 大内核

大内核是将操作系统功能作为一个紧密结合的整体放到内核。

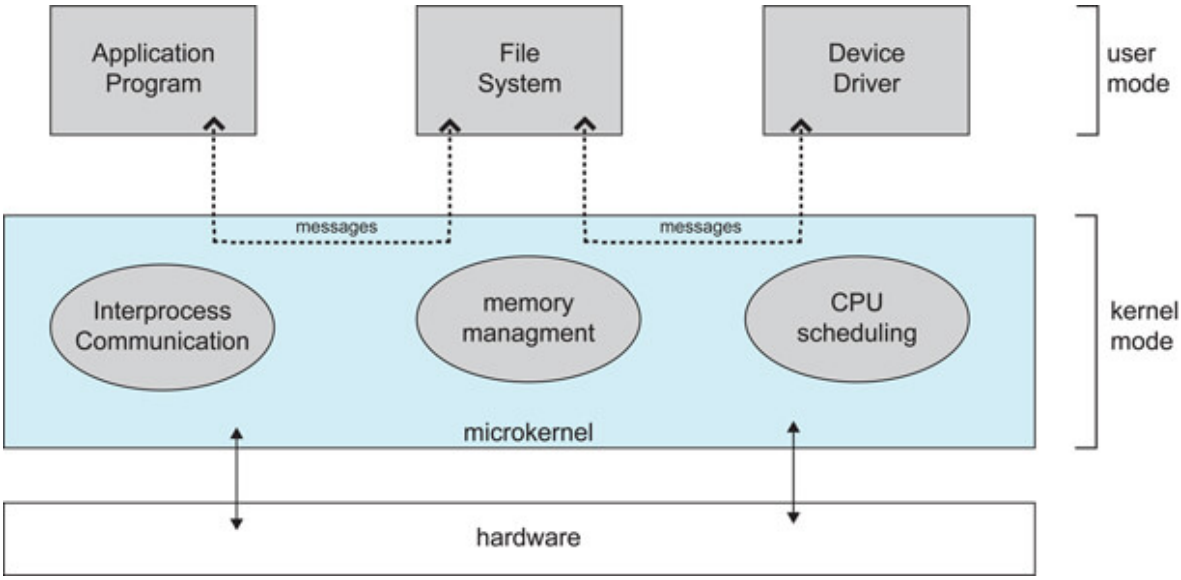
由于各模块共享信息，因此有很高的性能。

- 微内核

由于操作系统不断复杂，因此将一部分操作系统功能移出内核，从而降低内核的复杂性。移出的部分根据分层的原则划分成若干服务，相互独立。

在微内核结构下，操作系统被划分成小的、定义良好的模块，只有微内核这一个模块运行在内核态，其余模块运行在用户态。

因为需要频繁地在用户态和核心态之间进行切换，所以会有一定的性能损失。



中断分类

1. 外中断

由 CPU 执行指令以外的事件引起，如 I/O 完成中断，表示设备输入/输出处理已经完成，处理器能够发送下一个输入/输出请求。此外还有时钟中断、控制台中断等。

2. 异常

由 CPU 执行指令的内部事件引起，如非法操作码、地址越界、算术溢出等。

3. 陷入

在用户程序中使用系统调用。

类型	源头	响应方式	处理机制
中断	外设	异步	持续，对用户应用程序是透明的
异常	应用程序意想不到的行为	同步	杀死或重新执行意想不到的应用
系统调用	应用程序请求操作提供服务	异步或同步	等待和持续

6. 什么是堆和栈？说一下堆栈都存储哪些数据？

栈区（stack）— 由**编译器**自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

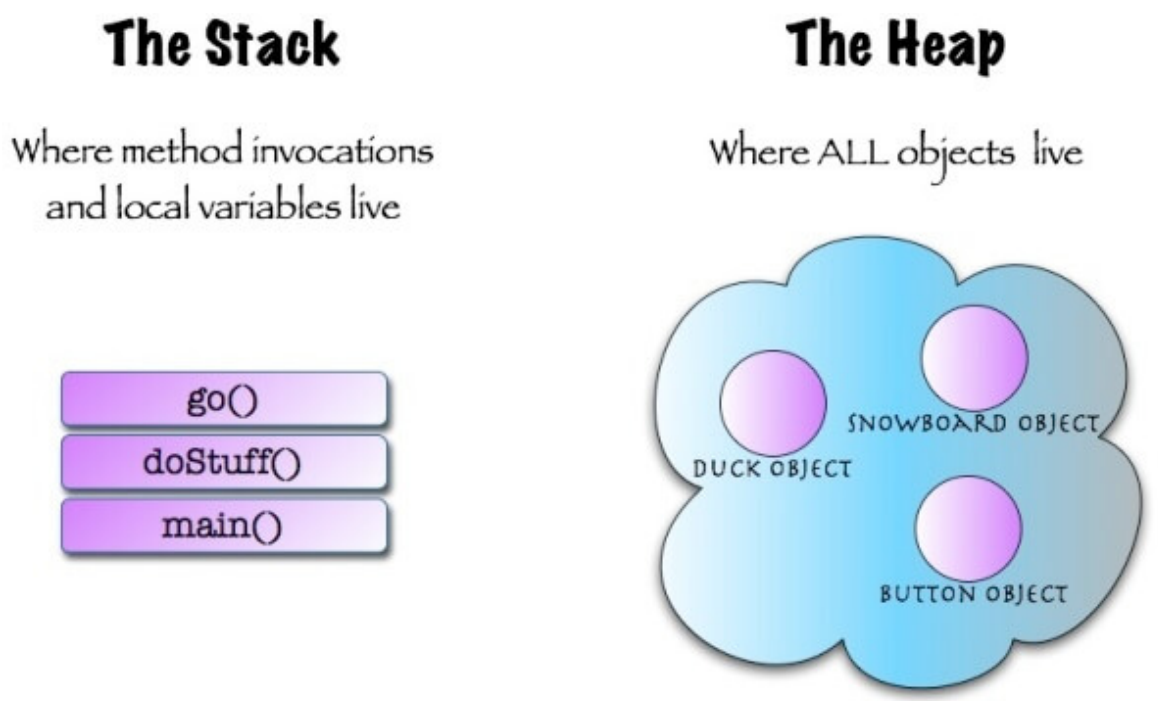
堆区（heap） — 一般由**程序员分配释放**， 若开发者不释放，则会在程序结束时可能由OS回收 。

数据结构中这两个完全就不放一块来讲，数据结构中栈和队列才是好基友，我想新手也很容易区分。

我想需要区分的情况肯定不是在数据结构话题下，而大多是在 OS 关于不同对象的内存分配这块上。

简单讲的话，在 C 语言中：

```
int a[N];   // go on a stack
int* a = (int *)malloc(sizeof(int) * N);  // go on a heap
```

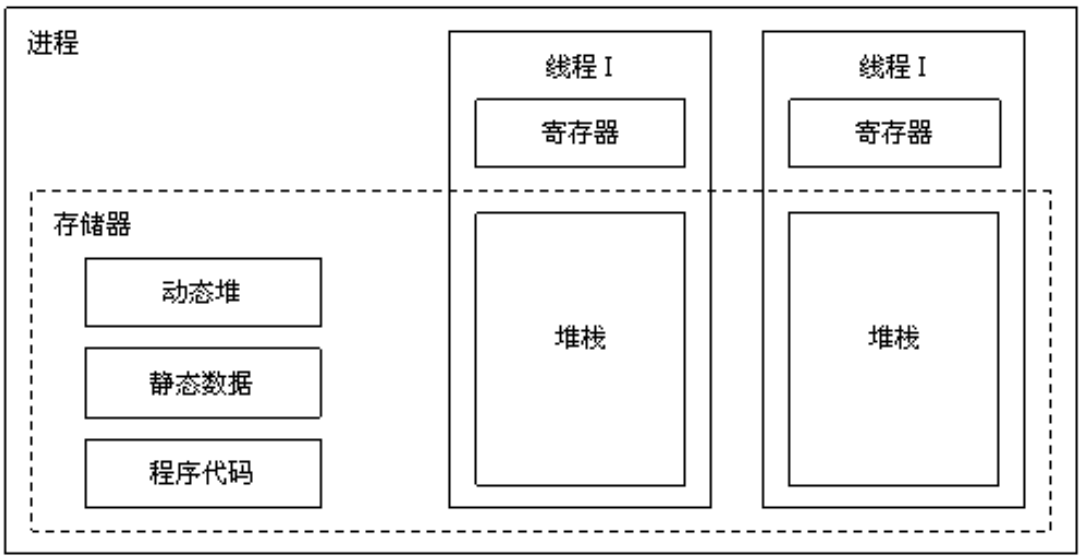


7. 分布式锁

分布式锁，是控制分布式系统之间同步访问共享资源的一种方式。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

进程管理

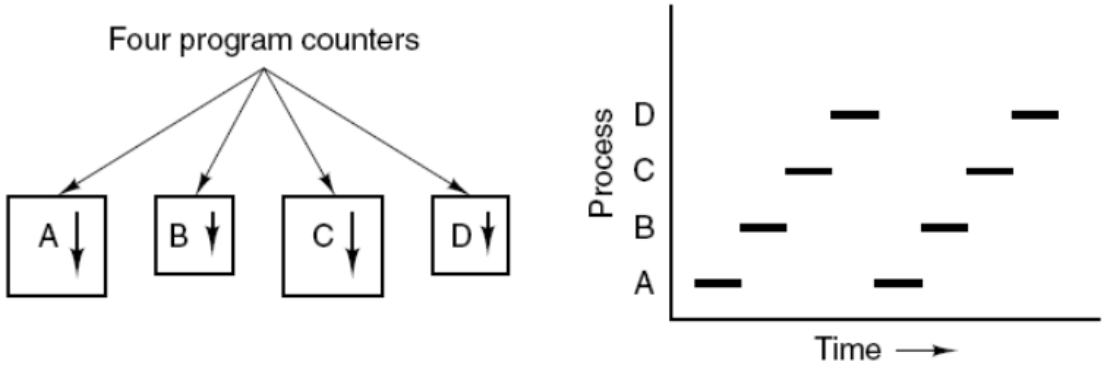
进程与线程:



1. 进程

进程是资源分配的基本单位，用来管理资源（例如：内存，文件，网络等资源）

进程控制块 (Process Control Block, PCB) 描述进程的基本信息和运行状态，所谓的创建进程和撤销进程，都是指对 PCB 的操作。（**PCB是描述进程的数据结构**）



上面是4 个程序创建了 4 个进程，这 4 个进程可以并发地执行。

2. 线程

线程是独立调度的基本单位。

一个进程中可以有多个线程，它们共享进程资源。

QQ 和浏览器是两个进程，浏览器进程里面有很多线程，例如 HTTP 请求线程、事件响应线程、渲染线程等等，线程的并发执行使得在浏览器中点击一个新链接从而发起 HTTP 请求时，浏览器还可以响应用户的其它事件。

3. 区别

（一）拥有资源

进程是资源分配的基本单位，但是线程不拥有资源，线程可以访问隶属进程的资源。

（二）调度

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切换，从一个进程内的线程切换到另一个进程中的线程时，会引起进程切换。

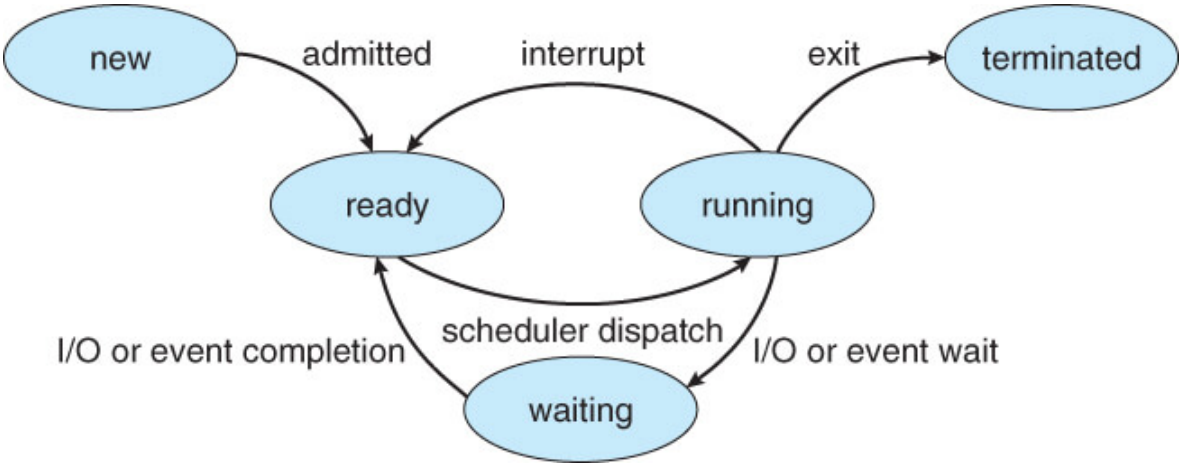
（三）系统开销

由于创建或撤销进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等，所付出的开销远大于创建或撤销线程时的开销。类似地，在进行进程切换时，涉及当前执行进程 CPU 环境的保存及新调度进程 CPU 环境的设置，而线程切换时只需保存和设置少量寄存器内容，开销很小。

（四）通信方面

进程间通信 (IPC) 需要进程同步和互斥手段的辅助，以保证数据的一致性。而线程间可以通过直接读/写同一进程中的数据段（如全局变量）来进行通信。

进程状态的切换



就绪状态（ready）：等待被调度
运行状态（running）
阻塞状态（waiting）：等待资源

需要注意的是:

- 只有就绪态和运行态可以相互转换，其它的都是单向转换。就绪状态的进程通过调度算法从而获得 CPU 时间，转为运行状态；而运行状态的进程，在分配给它的 CPU 时间片用完之后就会转为就绪状态，等待下一次调度。
- 阻塞状态是缺少需要的资源从而由运行状态转换而来，但是该资源不包括 CPU 时间，缺少 CPU 时间会从运行态转换为就绪态。
- 进程只能自己阻塞自己，因为只有进程自身才知道何时需要等待某种事件的发生

进程调度算法

相信很多人都听说过进程调度的算法，但是不同环境的调度算法目标不同，因此需要针对不同环境来讨论调度算法。

1. 批处理系统

批处理系统没有太多的用户操作，在该系统中，调度算法目标是保证吞吐量和周转时间（从提交到终止的时间）。

- 先来先服务
  先来先服务 `first-come first-serverd（FCFS）` ,按照请求的顺序进行调度。这样可以有利于长作业，但不利于短作业，因为短作业必须一直等待前面的长作业执行完毕才能执行，而长作业又需要执行很长时间，造成了短作业等待时间过长。
- 短作业优先

短作业优先 `shortest job first（SJF）` ,按估计运行时间最短的顺序进行调度。长作业有可能会饿死，处于一直等待短作业执行完毕的状态。因为如果一直有短作业到来，那么长作业永远得不到调度。

- 最短剩余时间优先

最短剩余时间优先 `shortest remaining time next（SRTN）` ,按估计剩余时间最短的顺序进行调度。

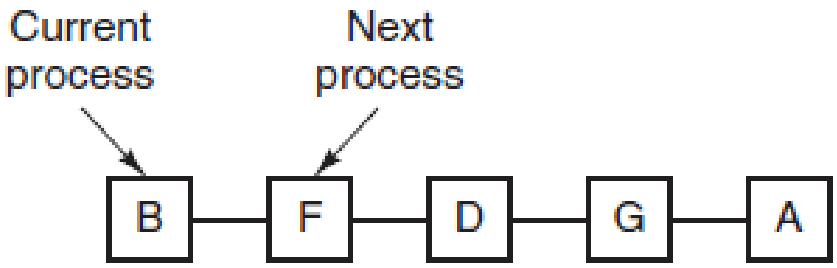
2. 交互式系统

交互式系统有大量的用户交互操作，在该系统中调度算法的目标是快速地进行响应。

- 时间片轮转算法

  将所有进程按FCFS （先来先服务） 的原则排成一个队列，每次调度时，把 CPU 时间分配给队首进程，该进程可以执行一个时间片。当时间片用完时，由计时器发出时钟中断，调度程序便停止该进程的执行，并将它送往就绪队列的末尾，同时继续把 CPU 时间分配给队首的进程。

时间片轮转算法的效率和时间片的大小有很大关系。因为进程切换都要保存进程的信息并且载入新进程的信息，如果时间片太小，会导致进程切换得太频繁，在进程切换上就会花过多时间。



- 优先级调度

为每个进程分配一个优先级，按优先级进行调度。

为了防止低优先级的进程永远等不到调度，可以随着时间的推移增加等待进程的优先级。

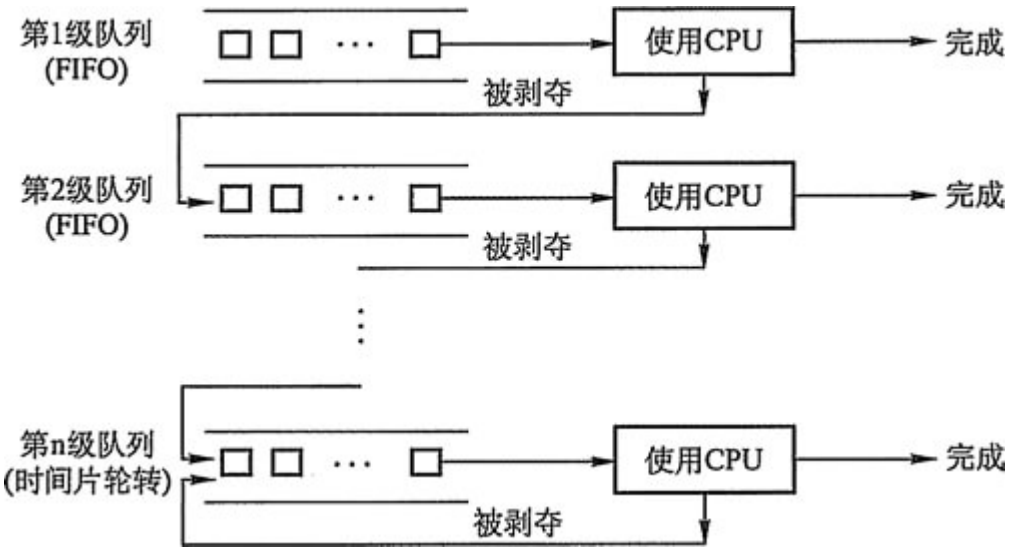
- 多级反馈队列

通常，如果一个进程需要执行 100 个时间片，如果采用时间片轮转调度算法，那么需要交换 100 次。

多级队列是为这种需要连续执行多个时间片的进程考虑，它设置了多个队列，每个队列时间片大小都不同，例如 1,2,4,8,..。进程在第一个队列没执行完，就会被移到下一个队列。这种方式下，之前的进程只需要交换 7 次。

每个队列优先权也不同，最上面的优先权最高。因此只有上一个队列没有进程在排队，才能调度当前队列上的进程。

可以将这种调度算法看成是时间片轮转调度算法和优先级调度算法的结合。



3. 实时系统

实时系统要求一个请求在一个确定时间内得到响应。

分为硬实时和软实时，前者必须满足绝对的截止时间，后者可以容忍一定的超时。

进程同步

- 临界区

对临界资源进行访问的那段代码称为临界区。

为了互斥访问临界资源，每个进程在进入临界区之前，需要先进行检查。

```
entry section
critical section;
exit section
```

- 同步与互斥

同步：多个进程按一定顺序执行；

互斥：多个进程在同一时刻只有一个进程能进入临界区。

- 信号量

> P 和 V 是来源于两个荷兰语词汇，P() ---prolaag （荷兰语，尝试减少的意思），V() ---verhoog（荷兰语，增加的意思）

信号量（Semaphore）是一个整型变量，可以对其执行 down 和 up 操作，也就是常见的 P 和 V 操作。

- down：如果信号量大于 0 ，执行 -1 操作；如果信号量等于 0，进程睡眠，等待信号量大于 0；（阻塞）
- up ：对信号量执行 +1 操作，唤醒睡眠的进程让其完成 down 操作。（唤醒）

down 和 up 操作需要被设计成原语，不可分割，通常的做法是在执行这些操作的时候屏蔽中断。

如果信号量的取值只能为 0 或者 1，那么就成为了 互斥量（Mutex） ，0 表示临界区已经加锁，1 表示临界区解锁。

```
typedef int semaphore;
semaphore mutex = 1;
void P1() {
    down(&mutex);
    // 临界区
    up(&mutex);
}

void P2() {
    down(&mutex);
    // 临界区
    up(&mutex);
}
```

- 使用信号量实现生产者-消费者问题

通常，使用一个缓冲区来保存数据，只有缓冲区没有满，生产者才可以放入数据；只有缓冲区不为空，消费者才可以拿走数据。

因为缓冲区属于临界资源，因此需要使用一个互斥量 mutex 来控制对缓冲区的互斥访问。

为了同步生产者和消费者的行为，需要记录缓冲区中数据的数量。数量可以使用信号量来进行统计，这里需要使用两个信号量：empty 记录空缓冲区的数量，full 记录满缓冲区的数量。其中，empty 信号量是在生产者进程中使用，当 empty 不为 0 时，生产者才可以放入数据；full 信号量是在消费者进程中使用，当 full 信号量不为 0 时，消费者才可以取走数据。

这里需要注意，不能先对缓冲区进行加锁，再测试信号量。也就是说，不能先执行 down(mutex) 再执行 down(empty)。如果这么做了，那么可能会出现这种情况：生产者对缓冲区加锁后，执行 down(empty) 操作，发现 empty = 0，此时生产者睡眠。消费者不能进入临界区，因为生产者对缓冲区加锁了，也就无法执行 up(empty) 操作，empty 永远都为 0，那么生产者和消费者就会一直等待下去，造成死锁。

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer() {
    while(TRUE){
        int item = produce_item(); // 生产一个产品
        // down(&empty) 和 down(&mutex) 不能交换位置，否则造成死锁
        down(&empty); // 记录空缓冲区的数量，这里减少一个产品空间
        down(&mutex); // 互斥锁
        insert_item(item);
        up(&mutex); // 互斥锁
        up(&full); // 记录满缓冲区的数量，这里增加一个产品
    }
}

void consumer() {
    while(TRUE){
        down(&full); // 记录满缓冲区的数量，减少一个产品
        down(&mutex); // 互斥锁
        int item = remove_item();
        up(&mutex); // 互斥锁
        up(&empty); // 记录空缓冲区的数量，这里增加一个产品空间
        consume_item(item);
    }
}
```

- 管程

管程 (Monitors，也称为监视器) 是一种程序结构，结构内的多个子程序（对象或模块）形成的多个工作线程互斥访问共享资源。

使用信号量机制实现的生产者消费者问题需要客户端代码做很多控制，而管程把控制的代码独立出来，不仅不容易出错，也使得客户端代码调用更容易。

管程是为了解决信号量在临界区的 PV 操作上的配对的麻烦，把配对的 PV 操作集中在一起，生成的一种并发编程方法。其中使用了条件变量这种同步机制。

c 语言不支持管程，下面的示例代码使用了类 Pascal 语言来描述管程。示例代码的管程提供了 insert() 和 remove() 方法，客户端代码通过调用这两个方法来解决生产者-消费者问题。

```
monitor ProducerConsumer
    integer i;
```

```
    condition c;

    procedure insert();
    begin
        // ...
    end;

    procedure remove();
    begin
        // ...
    end;
end monitor;
```

管程有一个重要特性：在一个时刻只能有一个进程使用管程。进程在无法继续执行的时候不能一直占用管程，否者其它进程永远不能使用管程。

管程引入了 条件变量 以及相关的操作：wait() 和 signal() 来实现同步操作。对条件变量执行 wait() 操作会导致调用进程阻塞，把管程让出来给另一个进程持有。signal() 操作用于唤醒被阻塞的进程。

- 经典同步问题

1. 读-写问题

允许多个进程同时对数据进行读操作，但是不允许读和写以及写和写操作同时发生。

Rcount：读操作的进程数量（Rcount=0）

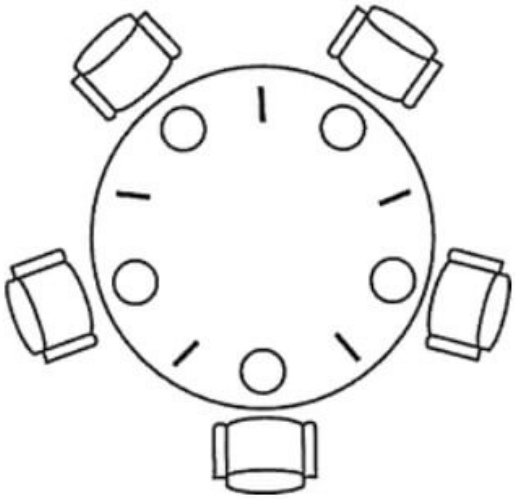
CountMutex：对于Rcount进行加锁（CountMutex=1）

WriteMutex：互斥量对于写操作的加锁（WriteMutex=1）

```
Rcount = 0;
semaphore CountMutex = 1;
semaphore WriteMutex = 1;

void writer(){
    while(true){
        sem_wait(WriteMutex);
        // TO DO write();
        sem_post(WriteMutex);
    }
}

// 读者优先策略
void reader(){
    while(true){
        sem_wait(CountMutex);
        if(Rcount == 0)
            sem_wait(WriteMutex);
        Rcount++;
        sem_post(CountMutex);

        // TO DO read();

        sem_wait(CountMutex);
        Rcount--;
        if(Rcount == 0)
            sem_post(WriteMutex);
        sem_post(CountMutex);
```

```
    }
}
```

2. 哲学家进餐问题



五个哲学家围着一张圆桌，每个哲学家面前放着食物。哲学家的生活有两种交替活动：吃饭以及思考。当一个哲学家吃饭时，需要先拿起自己左右两边的两根筷子，并且一次只能拿起一根筷子。

第一种方案:

下面是一种错误的解法，考虑到如果所有哲学家同时拿起左手边的筷子，那么就无法拿起右手边的筷子，造成死锁。

```
#define N 5           // 哲学家个数
void philosopher(int i)   {// 哲学家编号：0 — 4
    while(TRUE)
    {
        think();            // 哲学家在思考
        take_fork(i);            // 去拿左边的叉子
        take_fork((i + 1) % N);    // 去拿右边的叉子
        eat();               // 吃面条中….
        put_fork(i);            // 放下左边的叉子
        put_fork((i + 1) % N);    // 放下右边的叉子
    }
}
```

第二种方案:

对拿叉子的过程进行了改进，但仍不正确.

```
#define N 5     // 哲学家个数
while(1){
    take_fork(i);            // 去拿左边的叉子
    if(fork((i+1)%N)) {        // 右边叉子还在吗
        take_fork((i + 1) % N);// 去拿右边的叉子
        break;            // 两把叉子均到手
    }
    else {                // 右边叉子已不在
        put_fork(i);        // 放下左边的叉子
        wait_some_time();    // 等待一会儿
    }
}
```

第三种方案:

等待时间随机变化。可行，但非万全之策.

```
#define N 5     // 哲学家个数
while(1) {
    take_fork(i);            // 去拿左边的叉子
    if(fork((i+1)%N)) {        // 右边叉子还在吗
        take_fork((i + 1) % N);// 去拿右边的叉子
        break;              // 两把叉子均到手
    }
    else {                  // 右边叉子已不在
        put_fork(i);         // 放下左边的叉子
        wait_random_time( );    // 等待随机长时间
    }
}
```

第四种方案:

互斥访问。正确，但每次只允许一人进餐

```
semaphore mutex       // 互斥信号量，初值1
void philosopher(int i)  // 哲学家编号i：0－4
{
    while(TRUE){
        think();            // 哲学家在思考
        P(mutex);           // 进入临界区
        take_fork(i);             // 去拿左边的叉子
        take_fork((i + 1) % N);    // 去拿右边的叉子
        eat();                // 吃面条中….
        put_fork(i);            // 放下左边的叉子
        put_fork((i + 1) % N);    // 放下右边的叉子
        V(mutex);           // 退出临界区
    }
}
```

正确方案如下：

为了防止死锁的发生，可以设置两个条件（临界资源）：

- 必须同时拿起左右两根筷子；
- 只有在两个邻居都没有进餐的情况下才允许进餐。

```
//1. 必须由一个数据结构，来描述每个哲学家当前的状态
#define N 5
#define LEFT i // 左邻居
#define RIGHT (i + 1) % N    // 右邻居
#define THINKING 0
#define HUNGRY   1
#define EATING   2
typedef int semaphore;
int state[N];                // 跟踪每个哲学家的状态

//2. 该状态是一个临界资源，对它的访问应该互斥地进行
semaphore mutex = 1;         // 临界区的互斥
```

```
//3. 一个哲学家吃饱后，可能要唤醒邻居，存在着同步关系
semaphore s[N];              // 每个哲学家一个信号量

void philosopher(int i) {
    while(TRUE) {
        think();
        take_two(i);
        eat();
        put_tow(i);
    }
}

void take_two(int i) {
    P(&mutex);  // 进入临界区
    state[i] = HUNGRY; // 我饿了
    test(i); // 试图拿两把叉子
    V(&mutex); // 退出临界区
    P(&s[i]); // 没有叉子便阻塞
}

void put_tow(i) {
    P(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    V(&mutex);
}

void test(i) {          // 尝试拿起两把筷子
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] !=EATING) {
        state[i] = EATING;
        V(&s[i]); // 通知第i个人可以吃饭了
    }
}
```

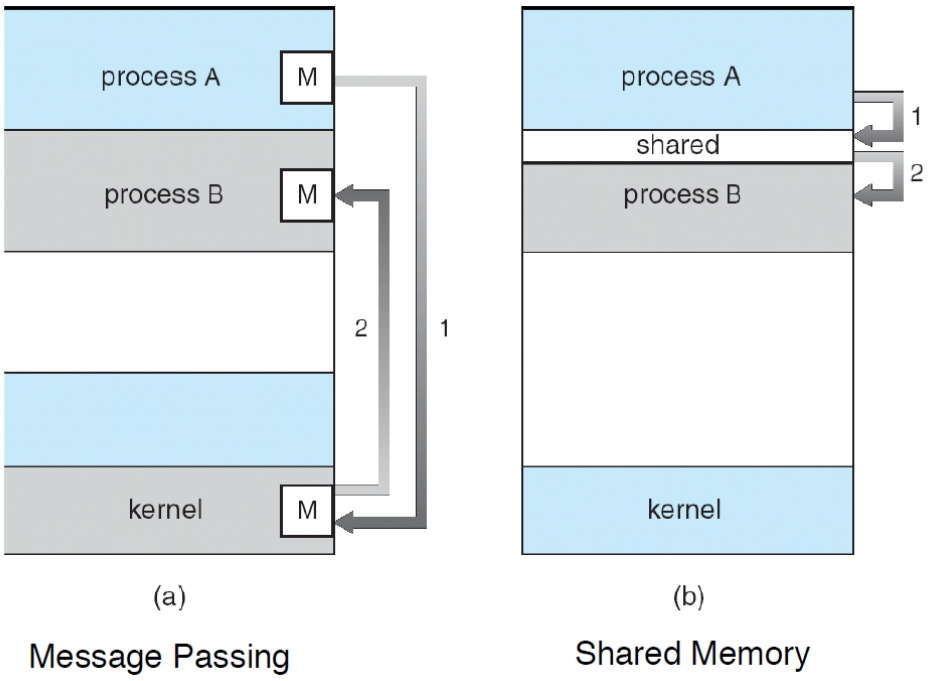
进程通信

进程同步与进程通信很容易混淆，它们的区别在于：

- 进程同步：控制多个进程按一定顺序执行
- 进程通信：进程间传输信息

进程通信是一种手段，而进程同步是一种目的。也可以说，为了能够达到进程同步的目的，需要让进程进行通信，传输一些进程同步所需要的信息。

进程通信方式：



- 直接通信

  发送进程直接把消息发送给接收进程，并将它挂在接收进程的消息缓冲队列上，接收进程从消息缓冲队列中取得消息。

Send 和 Receive 原语的使用格式如下：

```
Send(Receiver,message);//发送一个消息message给接收进程Receiver
Receive(Sender,message);//接收Sender进程发送的消息message
```

- 间接通信

间接通信方式是指进程之间的通信需要通过作为共享数据结构的实体。该实体用来暂存发送进程发给目标进程的消息。

发送进程把消息发送到某个中间实体中，接收进程从中间实体中取得消息。这种中间实体一般称为信箱，这种通信方式又称为信箱通信方式。该通信方式广泛应用于计算机网络中，相应的通信系统称为电子邮件系统。

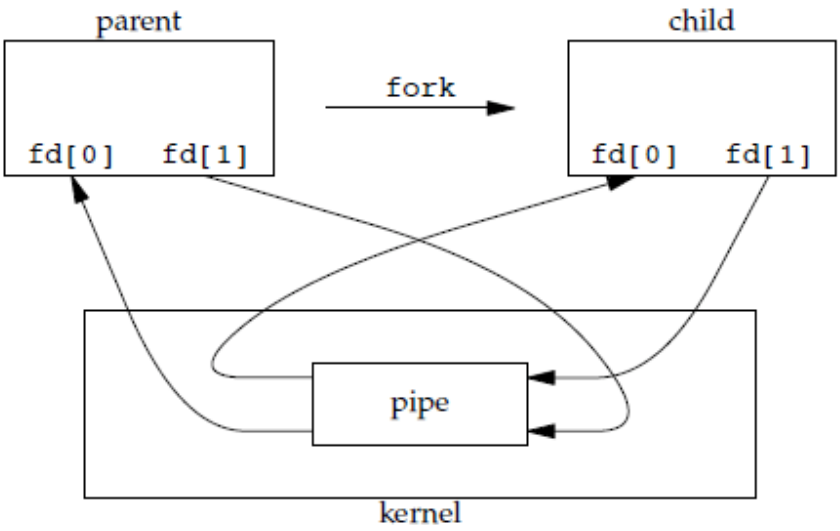
- 管道

管道是通过调用 pipe 函数创建的，fd[0] 用于读，fd[1] 用于写。

```
#include <unistd.h>
int pipe(int fd[2]);
```

它具有以下限制：

- 只支持半双工通信（单向传输）
- 只能在父子进程中使用

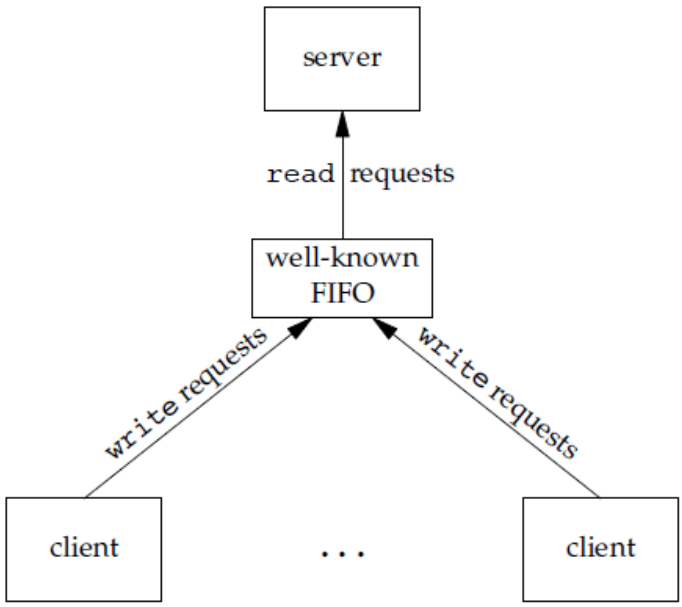


- 命名管道

命名管道的作用是，管道只能在父子进程中使用的限制。

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

FIFO 常用于客户-服务器应用程序中，FIFO 用作汇聚点，在客户进程和服务器进程之间传递数据。



- 消息队列

消息队列相比于 FIFO，消息队列具有以下优点：

消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；

- 避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；
- 读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。
- 信号量

它是一个计数器，用于为多个进程提供对共享数据对象的访问。

- 共享内存

允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。

需要使用信号量用来同步对共享存储的访问。

多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用使用内存的匿名段。

- 套接字

套接字与其它通信机制不同的是，它可用于不同机器间的进程通信。

线程间通信和进程间通信

- 线程间通信

1. synchronized同步

synchronized同步本质上就是“共享内存”式的通信。多个线程需要访问同一个共享变量，谁拿到了锁（获得了访问权限），谁就可以执行。

2. while轮询的方式

while轮询的方式就是，ThreadA 不断地改变条件，ThreadB 不停地通过 while 语句检测这个条件 (list.size()==5) 是否成立 ，从而实现了线程间的通信。但是这种方式会浪费 CPU 资源。这个有点浪费资源。

3. wait/notify机制

当条件未满足时，ThreadA 调用 wait() 放弃 CPU，并进入阻塞状态。（不像 while 轮询那样占用 CPU）

当条件满足时，ThreadB 调用 notify() 通知线程 A，所谓通知线程 A，就是唤醒线程 A，并让它进入可运行状态。

- 进程间通信

1. 管道（Pipe）:管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。
2. 命名管道（named pipe）:命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系 进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。
3. 信号（Signal）:信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送 信号给进程本身；Linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。
4. 消息（Message）队列 :消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺
5. 共享内存 ：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
6. 内存映射（mapped memory）:内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。

7. 信号量（semaphore）:主要作为进程间以及同一进程不同线程之间的同步手段。
8. 套接口（Socket）:更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：linux和System V的变种都支持套接字。

进程操作

Linux进程结构可由三部分组成：

- 代码段（程序）
- 数据段（数据）
- 堆栈段（控制块PCB）

进程控制块是进程存在的惟一标识，系统通过PCB的存在而感知进程的存在。系统通过 PCB 对进程进行管理和调度。PCB 包括创建进程、执行进程、退出进程以及改变进程的优先级等。

一般程序转换为进程分以下几个步骤：

1. 内核将程序读入内存，为程序分配内存空间.
2. 内核为该进程分配进程标识符 PID 和其他所需资源.
3. 内核为进程保存 PID 及相应的状态信息，把进程放到运行队列中等待执行，程序转化为进程后可以被操作系统的调度程序调度执行了.

在 UNIX 里，除了进程 0（即 PID=0 的交换进程，Swapper Process）以外的所有进程都是由其他进程使用系统调用 fork 创建的，这里调用 fork 创建新进程的进程即为父进程，而相对应的为其创建出的进程则为子进程，因而除了进程 0 以外的进程都只有一个父进程，但一个进程可以有多个子进程。操作系统内核以进程标识符（Process Identifier，即 PID ）来识别进程。进程 0 是系统引导时创建的一个特殊进程，在其调用 fork 创建出一个子进程（即 PID=1 的进程 1，又称 init）后，进程 0 就转为交换进程（有时也被称为空闲进程），而进程1（init进程）就是系统里其他所有进程的祖先。

进程0：Linux引导中创建的第一个进程，完成加载系统后，演变为进程调度、交换及存储管理进程。　　进程1：init 进程，由0进程创建，完成系统的初始化. 是系统中所有其它用户进程的祖先进程。

Linux中 1 号进程是由 0 号进程来创建的，因此必须要知道的是如何创建 0 号进程，由于在创建进程时，程序一直运行在内核态，而进程运行在用户态，因此创建 0 号进程涉及到特权级的变化，即从特权级 0 变到特权级 3，Linux 是通过模拟中断返回来实现特权级的变化以及创建 0 号进程，通过将 0 号进程的代码段选择子以及程序计数器EIP直接压入内核态堆栈，然后利用 iret 汇编指令中断返回跳转到 0 号进程运行。

- 创建一个进程

进程是系统中基本的执行单位。Linux 系统允许任何一个用户进程创建一个子进程，创建成功后，子进程存在于系统之中，并且独立于父进程。该子进程可以接受系统调度，可以得到分配的系统资源。系统也可以检测到子进程的存在，并且赋予它与父进程同样的权利。

Linux系统下使用 fork() 函数创建一个子进程，其函数原型如下：

```
#include <unistd.h>
pid_t fork(void);
```

在讨论 fork() 函数之前，有必要先明确父进程和子进程两个概念。除了 0 号进程（该进程是系统自举时由系统创建的）以外，Linux 系统中的任何一个进程都是由其他进程创建的。创建新进程的进程，即调用 fork() 函数的进程就是父进程，而新创建的进程就是子进程。

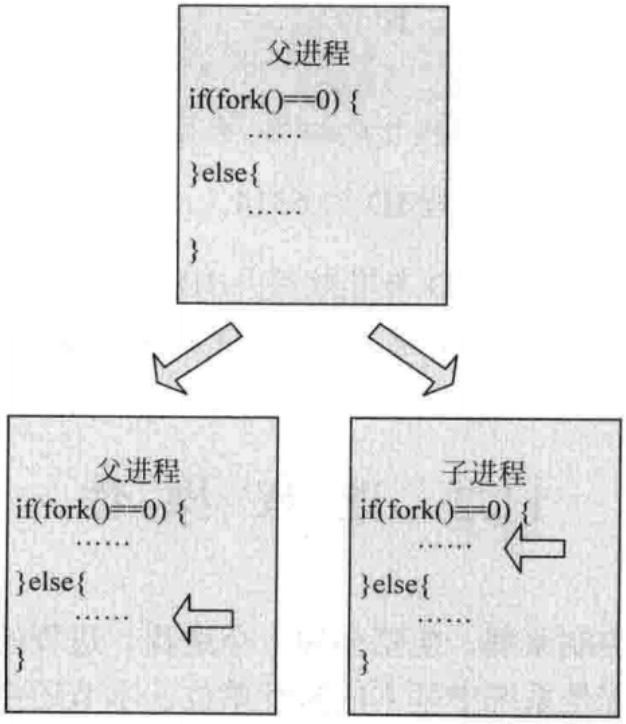
fork() 函数不需要参数，返回值是一个进程标识符 (PID)。对于返回值，有以下 3 种情况：

1. 对于父进程，fork() 函数返回新创建的子进程的 ID。

2. 对于子进程，fork() 函数返回 0。由于系统的 0 号进程是内核进程，所以子进程的进程标识符不会是0，由此可以用来区别父进程和子进程。

3. 如果创建出错，则 fork() 函数返回 -1。

fork() 函数会创建一个新的进程，并从内核中为此进程分配一个新的可用的进程标识符 (PID)，之后，为这个新进程分配进程空间，并将父进程的进程空间中的内容复制到子进程的进程空间中，包括父进程的数据段和堆栈段，并且和父进程共享代码段（写时复制）。这时候，系统中又多了一个进程，这个进程和父进程一模一样，两个进程都要接受系统的调度。



调用 fork()函数之后产生一个子进程

下面给出的示例程序用来创建一个子进程，该程序在父进程和子进程中分别输出不同的内容。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void){
    pid_t pid; // 保存进程ID
    pid = fork(); // 创建一个新进程
    if(pid < 0){ // fork出错
        printf("fail to fork\n");
        exit(1);
    }else if(pid == 0){    // 子进程
        // 打印子进程的进程ID
        printf("this is child, pid is : %u\n", getpid());
    }else{
        // 打印父进程和其子进程的进程ID
        printf("this is parent, pid is : %u, child-pid is : %u\n", getpid(), pid);
    }
    return 0;
}
```

运行：

```
> ./fork
Parent, PID: 2598, Sub-process PID: 2599
```

```
Sub-process, PID: 2599, PPID: 2598
```

由于创建的新进程和父进程在系统看来是地位平等的两个进程，所以运行机会也是一样的，我们不能够对其执行先后顺序进行假设，先执行哪一个进程取决于系统的调度算法。如果想要指定运行的顺序，则需要执行额外的操作。正因为如此，程序在运行时并不能保证输出顺序和上面所描述的一致。

getpid() 是获得当前进程的pid，而 getppid() 则是获得父进程的 id。

- 父子进程的共享资源

子进程完全复制了父进程的地址空间的内容，包括堆栈段和数据段的内容。子进程并没有复制代码段，而是和父进程共用代码段。这样做是存在其合理依据的，因为子进程可能执行不同的流程，那么就会改变数据段和堆栈段，因此需要分开存储父子进程各自的数据段和堆栈段。但是代码段是只读的，不存在被修改的问题，因此这一个段可以让父子进程共享，以节省存储空间.

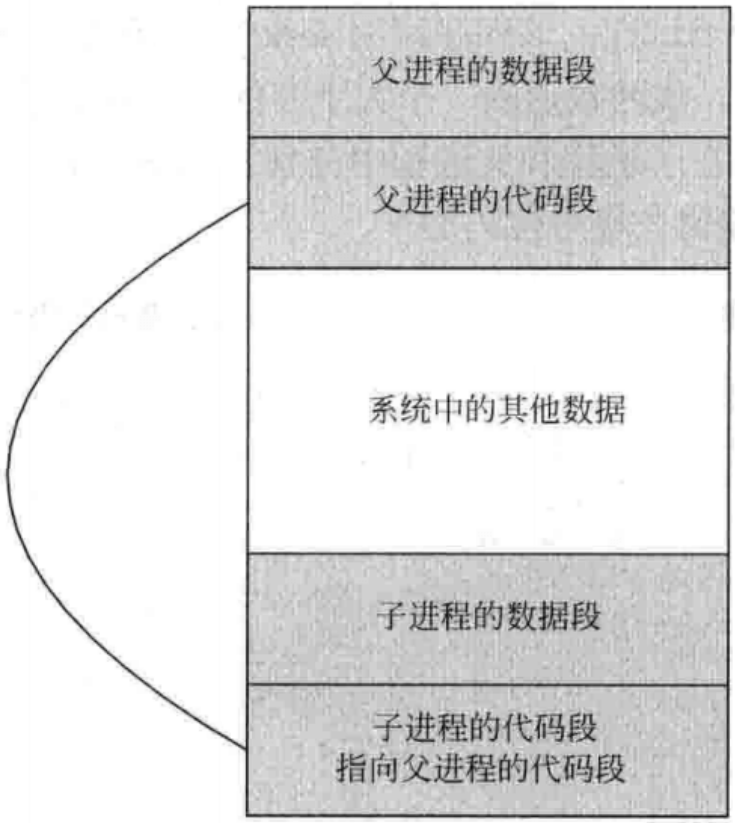


图 11-3　父子进程公用代码段

该程序定义了一个全局变量 global、一个局部变量 stack 和一个指针 heap。该指针用来指向一块动态分配的内存区域。之后，该程序创建一个子进程，在子进程中修改 global、stack 和动态分配的内存中变量的值。然后在父子进程中分别打印出这些变量的值。由于父子进程的运行顺序是不确定的，因此我们先让父进程额外休眠2秒，以保证子进程先运行。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
// 全局变量，在数据段中
int global;
int main(){
    pid_t pid;
    int stack = 1; // 局部变量，在栈中
    int * heap;
```

```
    heap = (int *)malloc(sizeof(int)); // 动态分配的内存，在堆中
    *heap = 2;
    pid = fork(); // 创建一个子进程
    if(pid < 0){ // 创建子进程失败
        printf("fail to fork\n");
        exit(1);
    }else if(pid == 0){ // 子进程，改变各变量的值
        global++; // 修改栈、堆和数据段
        stack++;
        (*heap)++;
        printf("the child, data : %d, stack : %d, heap : %d\n", global, stack, *heap);
        exit(0); // 子进程运行结束
    }
       // 父进程休眠2秒钟，保证子进程先运行
    sleep(2);
       // 输出结果
    printf("the parent, data : %d, stack : %d, heap : %d\n", global, stack, *heap);
    return 0;
}
```

程序运行效果如下：

```
> ./fork
In sub-process, global: 2, stack: 2, heap: 3
In parent-process, global: 1, stack: 1, heap: 2
```

由于父进程休眠了2秒钟，子进程先于父进程运行，因此会先在子进程中修改数据段和堆栈段中的内容。因此不难看出，子进程对这些数据段和堆栈段中内容的修改并不会影响到父进程的进程环境。

- fork()函数的出错情况

有两种情况可能会导致fork()函数出错：

1. 系统中已经有太多的进程存在了.

2. 调用fork()函数的用户进程太多了.

一般情况下，系统都会对一个用户所创建的进程数加以限制。如果操作系统不对其加限制，那么恶意用户可以利用这一缺陷攻击系统。下面是一个利用进程的特性编写的一个病毒程序，该程序是一个死循环，在循环中不断调用fork()函数来创建子进程，直到系统中不能容纳如此多的进程而崩溃为止。

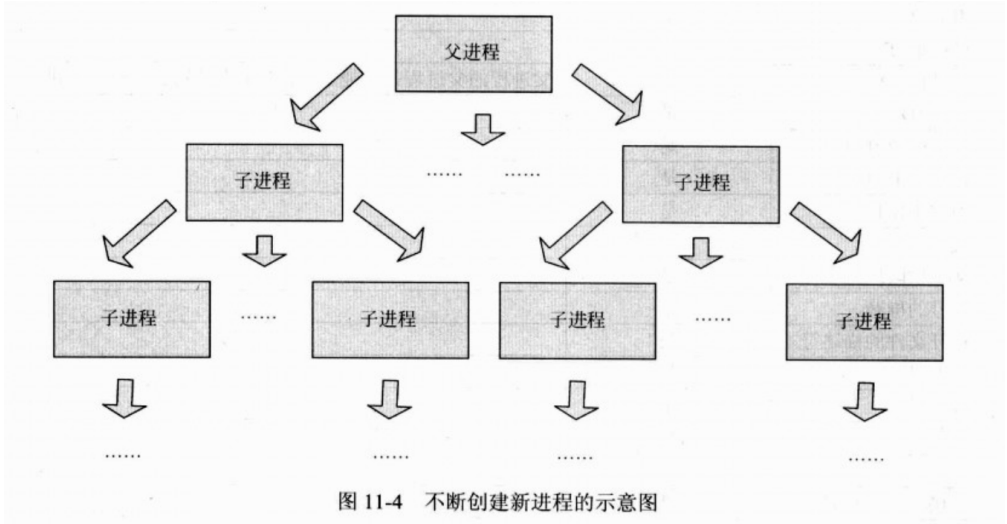


图 11-4　不断创建新进程的示意图

```
#include <unistd.h>
int main(){
    while(1)
        fork(); /* 不断地创建子进程，使系统中进程溢满 */
    return 0;
}
```

- 创建共享空间的子进程

进程在创建一个新的子进程之后，子进程的地址空间完全和父进程分开。父子进程是两个独立的进程，接受系统调度和分配系统资源的机会均等，因此父进程和子进程更像是一对兄弟。如果父子进程共用父进程的地址空间，则子进程就不是独立于父进程的。

Linux环境下提供了一个与 fork() 函数类似的函数，也可以用来创建一个子进程，只不过新进程与父进程共用父进程的地址空间，其函数原型如下：

```
#include <unistd.h>
pid_t vfork(void);
```

vfork() 和 fork() 函数的区别如下：

1. vfork() 函数产生的子进程和父进程完全共享地址空间，包括代码段、数据段和堆栈段，子进程对这些共享资源所做的修改，可以影响到父进程。由此可知，vfork() 函数与其说是产生了一个进程，还不如说是产生了一个线程。

2. vfork() 函数产生的子进程一定比父进程先运行，也就是说父进程调用了 vfork() 函数后会等待子进程运行后再运行。

在子进程中，我们先让其休眠 2 秒以释放 CPU 控制权，在前面的 fork() 示例代码中我们已经知道这样会导致其他线程先运行，也就是说如果休眠后父进程先运行的话，则第 1 点则为假；否则为真。第 2 点为真，则会先执行子进程，那么全局变量便会被修改，如果第 1 点为真，那么后执行的父进程也会输出与子进程相同的内容。

```
//@file vfork.c
//@brief vfork() usage
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int global = 1;

int main(void){
    pid_t pid;
    int   stack = 1;
    int  *heap;

    heap = (int *)malloc(sizeof(int));
    *heap = 1;

    pid = vfork();
    if (pid < 0){
        perror("fail to vfork");
        exit(-1);
    }else if (pid == 0) {
        //sub-process, change values
        sleep(2);//release cpu controlling
        global = 999;
        stack  = 888;
        *heap  = 777;
        //print all values
        printf("In sub-process, global: %d, stack: %d, heap: %d\n",global,stack,*heap);
```

```
        exit(0);
    }else{
        //parent-process
        printf("In parent-process, global: %d, stack: %d, heap: %d\n",global,stack,*heap);
    }

    return 0;
}
```

运行

```
> ./vfork
In sub-process, global: 999, stack: 888, heap: 777
In parent-process, global: 999, stack: 888, heap: 777
```

- 在函数内部调用vfork

在使用 vfork() 函数时应该注意不要在任何函数中调用 vfork() 函数。下面的示例是在一个非 main 函数中调用了 vfork() 函数。该程序定义了一个函数 f1()，该函数内部调用了 vfork() 函数。之后，又定义了一个函数 f2()，这个函数没有实际的意义，只是用来覆盖函数 f1() 调用时的栈帧。main 函数中先调用 f1() 函数，接着调用 f2() 函数。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int f1(void){
    vfork();
    return 0;
}

int f2(int a, int b){
    return a+b;
}

int main(void){
    int c;

    f1();
    c = f2(1,2);
    printf("%d\n",c);

    return 0;
}
```

运行:

```
> ./vfork
3
Segmentation fault (core dumped)
```

通过运行结果可以看出，一个进程运行正常，打印出了预期结果，而另一个进程似乎出了问题，发生了段错误。出现这种情况的原因可以用下图来分析一下：

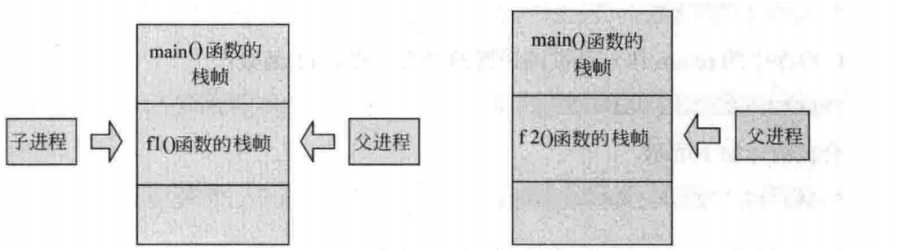


图 11-5　调用 vfork()函数的栈帧示意图

调用 vfork() 之后产生了一个子进程，并且和父进程共享堆栈段，两个进程都要从 f1() 函数返回。由于子进程先于父进程运行，所以子进程先从 f1() 函数中返回，并且调用 f2() 函数，其栈帧覆盖了原来 f1() 函数的栈帧。当子进程运行结束，父进程开始运行时，就出现了右图的情景，父进程需要从 f1() 函数返回，但是 f1() 函数的栈帧已经被 f2() 函数的所替代，因此就会出现父进程返回出错，发生段错误的情况。
由此可知，使用 vfork() 函数之后，子进程对父进程的影响是巨大的，其同步就很必要的了。

- 退出进程

当一个进程需要退出时，需要调用退出函数。Linux 环境下使用 exit() 函数退出进程，其函数原型如下：

```
#include <stdlib.h>
void exit(int status);
```

exit() 函数的参数表示进程的退出状态，这个状态的值是一个整型，保存在全局变量 $?中，在 shell 中可以通过 echo $? 来检查退出状态值。

注意：这里退出函数会深入内核注销掉进程的内核数据结构，并且释放掉进程的资源。

- exit函数与内核函数的关系

exit 函数是一个标准的库函数，其内部封装了 Linux 系统调用 exit() 函数。两者的主要区别在于 exit() 函数会在用户空间做一些善后工作，例如清理用户的 I/O 缓冲区，将其内容写入 磁盘文件等，之后才进入内核释放用户进程的地址空间；而 exit() 函数直接进入内核释放用户进程的地址空间，所有用户空间的缓冲区内容都将丢失。

- 设置进程所有者

每个进程都有两个用户 ID，实际用户 ID 和有效用户 ID。通常这两个 ID 的值是相等的，其取值为进程所有者的用户 ID。但是，在有些场合需要改变进程的有效用户 ID。Linux 环境下使用 setuid() 函数改变一个进程的实际用户ID和有效用户ID，其函数原型如下：

```
#include <unistd.h>
int setuid(uid_t uid);
```

setuid() 函数的参数表示改变后的新用户 ID，如果成功修改当前进程的实际用户 ID 和有效用户 ID，函数返回值为 0；如果失败，则返回 -1。只有两种用户可以修改进程的实际用户 ID 和有效用户 ID：

1. 根用户：根用户可以将进程的实际用户 ID 和有效用户 ID 更换。

2. 其他用户：其该用户的用户 ID 等于进程的实际用户 ID 或者保存的用户 ID。

用户可以将自己的有效用户 ID 改回去。这种情况多出现于下面的情况：一个进程需要具有某种权限，所以将其有效用户 ID 设置为具有这种权限的用户 ID，当进程不需要这种权限时，进程还原自己之前的有效用户 ID，使自己的权限复原。下面给出一个修改的示例：

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

```c
#include <unistd.h>
int main(void){
    FILE *fp;
    uid_t uid;
    uid_t euid;
    uid = getuid();         /* 得到进程的实际用户ID */
    euid = geteuid();    /* 得到进程的有效用户ID */
    printf("the uid is : %d\n", uid);
    printf("the euid is : %d\n", euid);
    if(setuid(8000) == -1){ /* 改变进程的实际用户ID和有效用户ID */
        perror("fail to set uid");
        exit(1);
    }
    printf("after changing\n");
    uid = getuid();         /* 再次得到进程的实际用户ID */
    euid = geteuid();    /* 再次得到进程的有效用户ID */
    printf("the uid is : %d\n", uid);
    printf("the euid is : %d\n", euid);
    return 0;
}
```

运行:

```
> ./setuid
the uid is : 0
the euid is : 0
after changing
the uid is : 8000
the euid is : 8000
```

孤儿进程和僵尸进程

在 Unix/Linux 中，正常情况下，子进程是通过父进程创建的，子进程在创建新的进程。子进程的结束和父进程的运行是一个异步过程，即父进程永远无法预测子进程 到底什么时候结束。当一个进程完成它的工作终止之后，它的父进程需要调用 wait() 或者 waitpid() 系统调用取得子进程的终止状态。

孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程（进程号为1）所收养，并由 init 进程对它们完成状态收集工作.

僵尸进程：一个进程使用 fork 创建子进程，如果子进程退出，而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程。

- 问题和影响

Unix 提供了一种机制可以保证只要父进程想知道子进程结束时的状态信息，就可以得到。这种机制就是：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存等。但是仍然为其保留一定的信息（包括进程号 the process ID，退出状态 the termination status of the process，运行时间 the amount of CPU time taken by the process 等)。直到父进程通过 wait / waitpid 来取时才释放。但这样就导致了问题，如果进程不调用 wait / waitpid 的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程。此即为僵尸进程的危害，应当避免。

孤儿进程是没有父进程的进程，孤儿进程这个重任就落到了 init 进程身上，init 进程就好像是一个民政局，专门负责处理孤儿进程的善后工作。每当出现一个孤儿进程的时候，内核就把孤 儿进程的父进程设置为 init，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。因此孤儿进程并不会有什么危害。

任何一个子进程（init除外）在exit() 之后，并非马上就消失掉，而是留下一个称为僵尸进程 (Zombie) 的数据结构，等待父进程处理。这是每个子进程在结束时都要经过的阶段。如果子进程在exit()之后，父进程没有来得及处理，这时用 ps 命令就能看到子进程的状态是 Z。如果父进程能及时处理，可能用 ps 命令就来不及看到子进程的僵尸状态，但这并不等于子进程不经过僵

尸状态。如果父进程在子进程结束之前退出，则子进程将由 init 接管。 init 将会以父进程的身份对僵尸状态的子进程进行处理。

僵尸进程影响场景：

例如有个进程，它定期的产生一个子进程，这个子进程需要做的事情很少，做完它该做的事情之后就退出了，因此这个子进程的生命周期很短，但是，父进程只管生成新的子进程，至于子进程退出之后的事情，则一概不闻不问，这样，系统运行上一段时间之后，系统中就会存在很多的僵死进程，倘若用 ps 命令查看的话，就会看到很多状态为 Z 的进程。 严格地来说，僵死进程并不是问题的根源，最有危害的是产生出大量僵死进程的那个父进程。因此，当我们寻求如何消灭系统中大量的僵死进程时，答案就是把产生大 量僵死进程的那个元凶枪毙掉（也就是通过 kill 发送 SIGTERM 或者 SIGKILL 信号啦）。kill了真正元凶进程之后，它产生的僵死进程就变成了孤儿进程，这些孤儿进程会被 init 进程接管，init 进程会 wait() 这些孤儿进程，释放它们占用的系统进程表中的资源，这样，这些已经僵死的孤儿进程就能瞑目而去了。

- 孤儿进程应用示例

  ```c
  #include <stdio.h>
  #include <stdlib.h>
  #include <errno.h>
  #include <unistd.h>
  ```

int main(){
pid_t pid;
//创建一个进程
pid = fork();
//创建失败
if (pid < 0){
perror(“fork error:”);
exit(1);
}

```
//子进程
if (pid == 0){
    printf("I am the child process.\n");
    //输出进程ID和父进程ID
    printf("pid: %d\tppid:%d\n",getpid(),getppid());
    printf("I will sleep five seconds.\n");
    //睡眠5s，保证父进程先退出
    sleep(5);
    printf("pid: %d\tppid:%d\n",getpid(),getppid());
    printf("child process is exited.\n");
}else{
    //父进程
    printf("I am father process.\n");
    //父进程睡眠1s，保证子进程输出进程id
    sleep(1);
    printf("father process is  exited.");
}
return 0;
```

}

运行:
```c
> ./test
I am father process.
I am the child process.
pid:3906  ppid:3905
```

```
I will sleep five seconds.
father process is exited.
```

- 僵尸进程应用示例：

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main(){
    pid_t pid;
    pid = fork();
    if (pid < 0){
        perror("fork error:");
        exit(1);
    }else if (pid == 0){
        printf("I am child process.I am exiting.\n");
        exit(0);
    }
    printf("I am father process.I will sleep two seconds\n");
    //等待子进程先退出

    sleep(2);
    //输出进程信息

    system("ps -o pid,ppid,state,tty,command");
    printf("father process is exiting.\n");
    return 0;
}
```

运行:

```
> ./test
I am father process.I will sleep five seconds.
I am child process.I am exiting.
PID   PPID   S   TT          COMMAND
3344  3343   S   PTS/1     -bash
4061  3344   S   PTS/1     ./test
4062  4061   Z   PTS/1     [test] <default>
4063  4061   R   PTS/1     ps -o pid,ppid,state,tty,command
father process is exiting.
```

- 僵尸进程解决办法

- 通过信号机制

子进程退出时向父进程发送SIGCHILD信号，父进程处理SIGCHILD信号。在信号处理函数中调用wait进行处理僵尸进程.

- fork两次

将子进程成为孤儿进程，从而其的父进程变为 init 进程，通过 init 进程可以处理僵尸进程.

守护进程

Linux Daemon（守护进程）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。它不需要用户输入就能运行而且提供某种服务，不是对整个系统就是对某个用户程序提供服务。Linux系统的大多数服务器就是通过守护进程实现的。常见的守护进程包括系统日志进程syslogd、 web服务器httpd、邮件服务器sendmail和数据库服务器mysqld等。

守护进程一般在系统启动时开始运行，除非强行终止，否则直到系统关机都保持运行。守护进程经常以超级用户（root）权限运行，因为它们要使用特殊的端口（1-1024）或访问某些特殊的资源。

一个守护进程的父进程是init进程，因为它真正的父进程在fork出子进程后就先于子进程exit退出了，所以它是一个由init继承的孤儿进程。守护进程是非交互式程序，没有控制终端，所以任何输出，无论是向标准输出设备stdout还是标准出错设备stderr的输出都需要特殊处理。

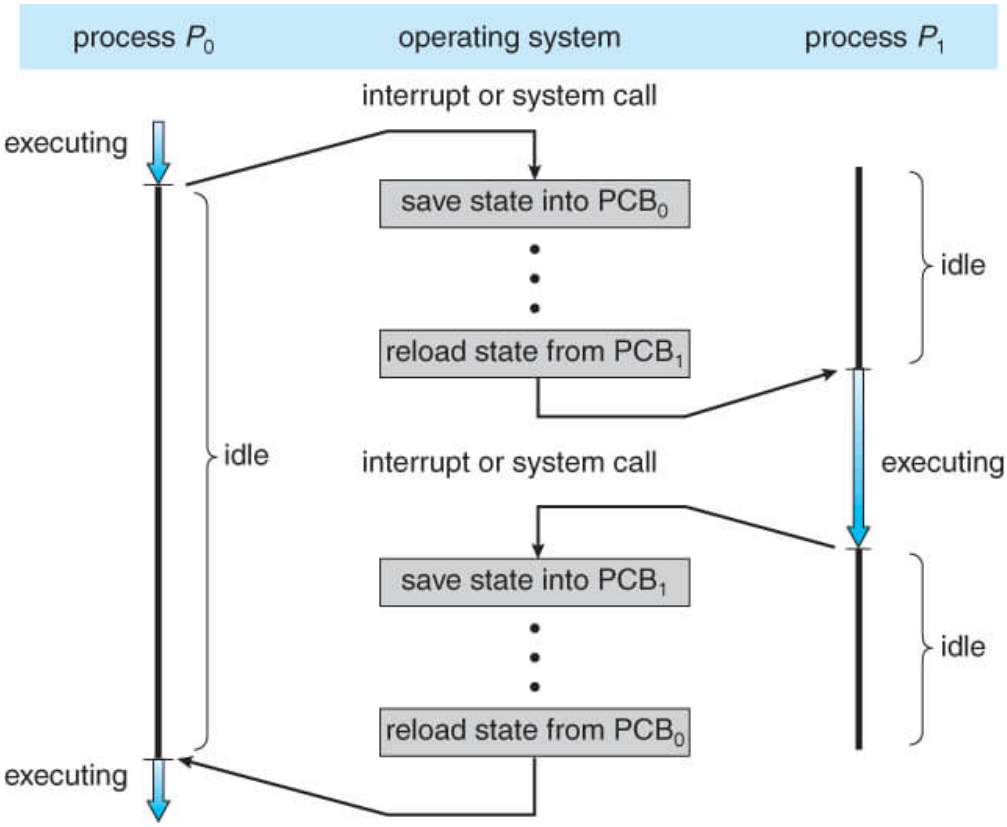
守护进程的名称通常以d结尾，比如sshd、xinetd、crond等

编写守护进程的一般步骤步骤：

- 在父进程中执行 fork 并 exit 推出；
- 在子进程中调用 setsid 函数创建新的会话；
- 在子进程中调用 chdir 函数，让根目录 / 成为子进程的工作目录；
- 在子进程中调用umask函数，设置进程的umask 为 0；
- 在子进程中关闭任何不需要的文件描述符

上下文切换

上下文切换，有时也称做进程切换或任务切换，是指CPU从一个进程或线程切换到另一个进程或线程。 在操作系统中，CPU 切换到另一个进程需要保存当前进程的状态并恢复另一个进程的状态：当前运行任务转为就绪（或者挂起、删除）状态，另一个被选定的就绪任务成为当前任务.



死锁

造成死锁的原因主要是有两种：

1. 可重用资源引起的
2. 消耗资源过多引起的

- 什么是死锁

1. 什么是死锁
   造成死锁的原因就是多个线程或进程对同一个资源的争抢或相互依赖。一个最简单的解释就是你去面试，面试官问你告诉我什么是死锁，我就录用你，你回答面试官你录用我，我告诉你。

如果一个进程集合里面的每个进程都在等待只能由这个集合中的其他一个进程（包括他自身）才能引发的事件，这种情况就是死锁。

我们可以看一个例子，有资源 A、B，进程 C、D 描述如下：

资源 A 和资源 B，都是不可剥夺资源，现在进程 C 已经申请了资源 A，进程 D 也申请了资源 B，进程 C 接下来的操作需要用到资源 B，而进程 D 恰好也在申请资源A，进程 C、D 都得不到接下来的资源，那么就引发了死锁。

然后套用回去定义：如果一个进程集合里面（进程 C 和进程 D）的每个进程（进程 C 和进程 D）都在等待只能由这个集合中的其他一个进程（对于进程 C，他在等进程 D；对于进程 D，他在等进程 C）才能引发的事件（释放相应资源）。

这里的资源包括,软件(程序)和硬件(扫描仪)。 资源一般可以分两种：可剥夺资源（Preemptable）和不可剥夺资源（Nonpreemptable）。一般来说对于由可剥夺资源引起的死锁可以由系统的重新分配资源来解决，所以一般来说大家说的死锁都是由于不可剥夺资源所引起的。

2. 死锁的必要条件

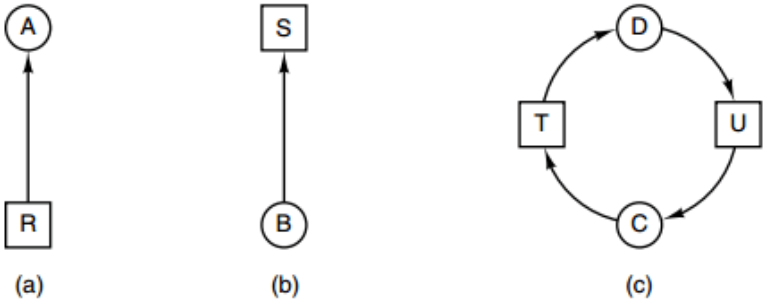


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- 互斥：每个资源要么已经分配给了一个进程，要么就是可用的。
- 占有和等待：已经得到了某个资源的进程可以再请求新的资源。
- 不可抢占：已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。
- 循环等待：有两个或者两个以上的进程组成一条环路，该环路中的每个进程都在等待下一个进程所占有的资源。

3. 死锁的处理方法

处理死锁的策略:

- 鸵鸟策略

把头埋在沙子里，假装根本没发生问题。
因为解决死锁问题的代价很高，因此鸵鸟策略这种不采取任务措施的方案会获得更高的性能。当发生死锁时不会对用户造成多大影响，或发生死锁的概率很低，可以采用鸵鸟策略。
大多数操作系统，包括 Unix，Linux 和 Windows，处理死锁问题的办法仅仅是忽略它。
检测死锁并且恢复。

仔细地对资源进行动态分配，以避免死锁。

通过破除死锁四个必要条件之一，来防止死锁产生。

死锁检测与死锁恢复：

不试图阻止死锁，而是当检测到死锁发生时，采取措施进行恢复。

(一) 每种类型一个资源的死锁检测

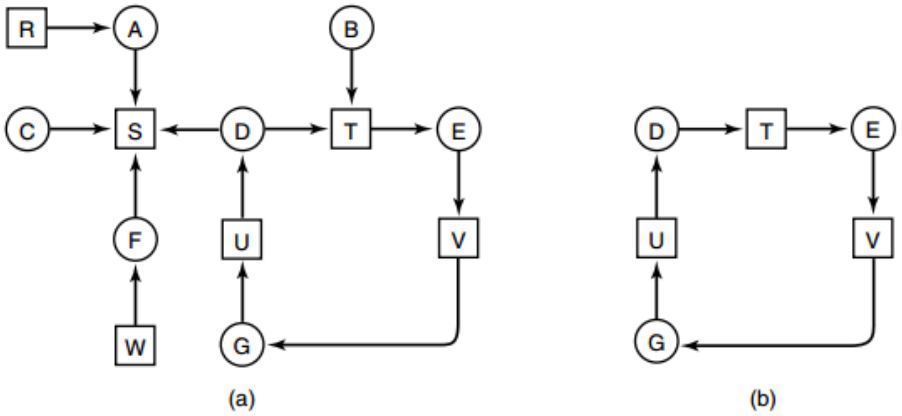


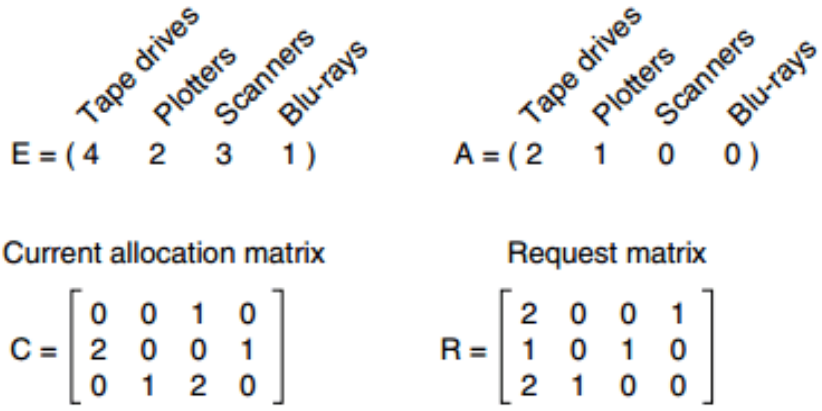
Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

图中是资源分配图，其中方框表示资源，圆圈表示进程。资源指向进程表示该资源已经分配给该进程，进程指向资源表示进程请求获取该资源。

图 a 可以抽取出环，如图 b，它满足了环路等待条件，因此会发生死锁。

每种类型一个资源的死锁检测算法是通过检测有向图是否存在环来实现，从一个节点出发进行深度优先搜索，对访问过的节点进行标记，如果访问了已经标记的节点，就表示有向图存在环，也就是检测到死锁的发生。

(二) 每种类型多个资源的死锁检测



图中有三个进程四个资源，每个数据代表的含义如下：

- E 向量：资源总量.
- A 向量：资源剩余量.
- C 矩阵：每个进程所拥有的资源数量，每一行都代表一个进程拥有资源的数量.
- R 矩阵：每个进程请求的资源数量.

因此就会有:

每个进程最开始时都不被标记，执行过程有可能被标记。当算法结束时，任何没有被标记的进程都是死锁进程。

1. 寻找一个没有标记的进程 Pi，它所请求的资源小于等于 A。
2. 如果找到了这样一个进程，那么将 C 矩阵的第 i 行向量加到 A 中，标记该进程，并转回 1。
3. 如果没有这样一个进程，算法终止。

(三)死锁恢复

- 利用抢占恢复
- 利用回滚恢复
- 通过杀死进程恢复
- 死锁预防

在程序运行之前预防发生死锁，确保系统永远不会进入死锁状态。

(一) 破坏互斥条件

破坏互斥条件最明显的例子就是，脱机打印机技术允许若干个进程同时输出，唯一真正请求物理打印机的进程是打印机守护进程。（把互斥地封装成可以同时访问的，例如：打印机的缓存）.

(二) 破坏占有和等待条件

一种实现方式是规定所有进程在开始执行前请求所需要的全部资源。

但是，这种策略也有如下缺点：

- 在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；
- 资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；
- 降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就因此少了。

(三) 破坏不可抢占条件

允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。

(四) 破坏循环等待

实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在以下缺点：

- 限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；
- 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。
- 死锁避免

在程序运行时避免发生死锁，在使用前进行判断，只允许不会出现死锁的进程请求资源。

（一）安全状态

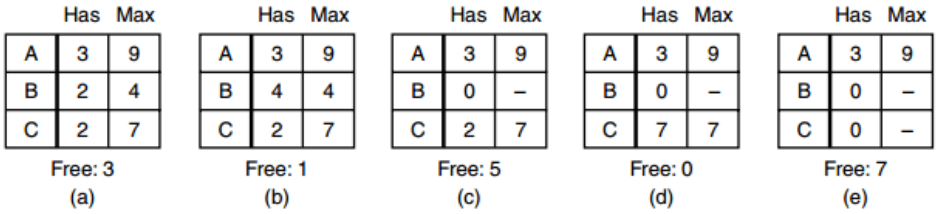


Figure 6-9. Demonstration that the state in (a) is safe.

图中 a 的第二列 Has 表示已拥有的资源数，第三列 Max 表示总共需要的资源数，Free 表示还有可以使用的资源数。从图 a 开始出发，先让 B 拥有所需的所有资源（图 b），运行结束后释放 B，此时 Free 变为 5（图 c）；接着以同样的方式运行 C 和 A，使得所有进程都能成功运行，因此可以称图 a 所示的状态时安全的。

定义：如果没有死锁发生，并且即使所有进程突然请求对资源的最大需求，也仍然存在某种调度次序能够使得每一个进程运行完毕，则称该状态是安全的。

安全状态的检测与死锁的检测类似，因为安全状态必须要求不能发生死锁。下面的银行家算法与死锁检测算法非常类似，可以结合着做参考对比。

（二）单个资源的银行家算法

一个小镇的银行家，银行家向一群客户分别承诺了一定的贷款额度，算法要做的是判断对请求的满足是否会进入不安全状态，如果是，就拒绝请求；否则予以分配。

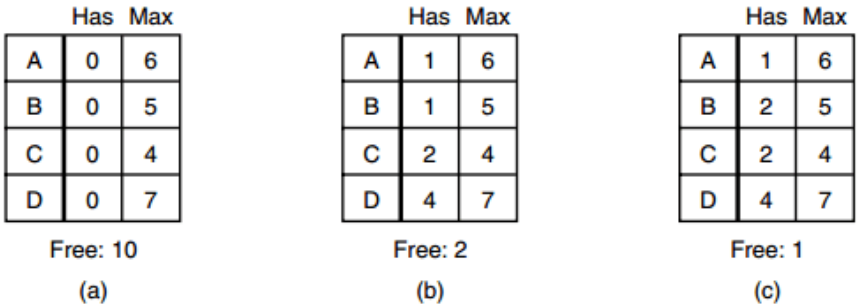


Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

图中c 为不安全状态，因此算法会拒绝之前的请求，从而避免进入图 c 中的状态。

(三) 多个资源的银行家算法

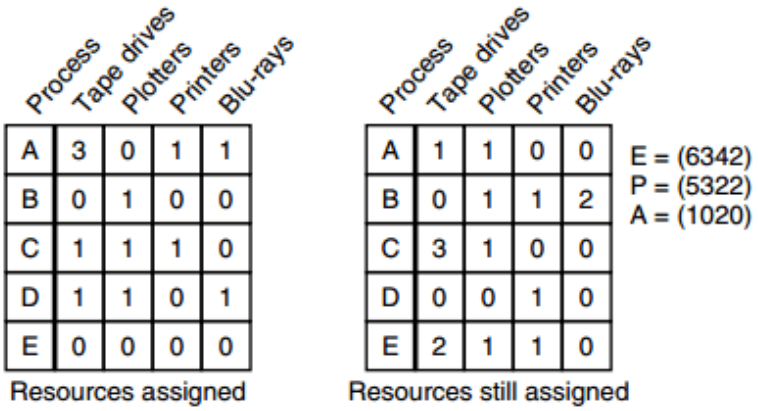


Figure 6-12. The banker's algorithm with multiple resources.

图中有五个进程，四个资源。左边的图表示已经分配的资源，右边的图表示还需要分配的资源。最右边的 E、P 以及 A 分别表示：总资源、已分配资源以及可用资源，注意这三个为向量，而不是具体数值，例如 A=(1020)，表示 4 个资源分别还剩下 1/0/2/0。

检查一个状态是否安全的算法如下：

- 查找右边的矩阵是否存在一行小于等于向量 A。如果不存在这样的行，那么系统将会发生死锁，状态是不安全的。
- 假若找到这样一行，将该进程标记为终止，并将其已分配资源加到 A 中。
- 重复以上两步，直到所有进程都标记为终止，则状态时安全的。

但是，如果一个状态不是安全的，那么就拒绝进入这个状态。

- 如何在程序开发中避免死锁

死锁，发生的主要原因在于了有多个进程去竞争资源，也就是同时去抢占。

可以自己写一个支持多线程的消息管理类，单开一个线程访问独占资源，其它线程用消息交互实现间接访问。 这种机制适应性强、效率高，更适合多核环境。

内存管理

1. 虚拟内存

虚拟内存是当代操作系统必备的一项重要功能了，它向进程屏蔽了底层了RAM和磁盘，并向进程提供了远超物理内存大小的内存空间。

虚拟内存的目的是为了让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。引入虚拟内存后，每个进程都要各自的虚拟内存，内存的并发访问问题的粒度从多进程级别，可以降低到多线程级别。

为了更好的管理内存，操作系统将内存抽象成地址空间。每个程序拥有自己的地址空间，这个地址空间被分割成多个块，每一块称为一页。这些页被映射到物理内存，但不需要映射到连续的物理内存，也不需要所有页都必须在物理内存中。当程序引用到一部分不在物理内存中的地址空间时，由硬件执行必要的映射，将缺失的部分装入物理内存并重新执行失败的指令

由此可以看出，虚拟内存允许程序不用将地址空间中的每一页都映射到物理内存，也就是说一个程序不需要全部调入内存就可以运行，这使得有限的内存运行大程序称为可能。例如有一台计算机可以产生 16 位地址，那么一个程序的地址空间范围是 0~64K。该计算机只有 32KB 的物理内存，虚拟内存技术允许该计算机运行一个 64K 大小的程序。

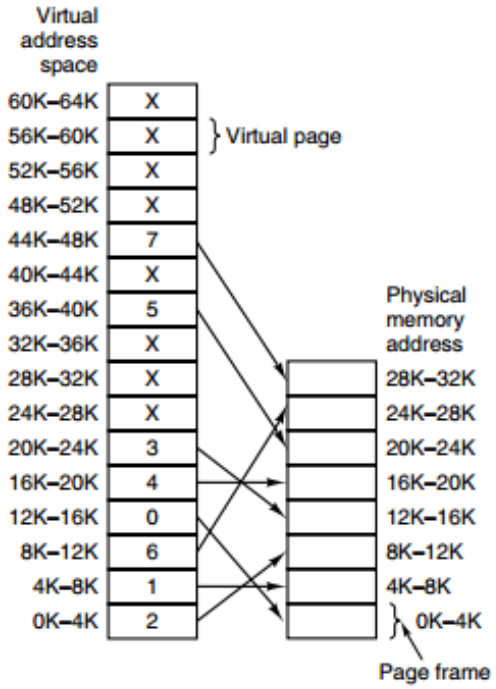


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

2. 分页系统地址映射

- 内存管理单元（MMU）：管理着地址空间和物理内存的转换。
- 页表（Page table）：页（地址空间）和页框（物理内存空间）的映射表。例如下图中，页表的第 0 个表项为 010，表示第 0 个页映射到第 2 个页框。页表项的最后一位用来标记页是否在内存中。

页表存放着 16 个页，这 16 个页需要用 4 个比特位来进行索引定位。因此对于虚拟地址（0010 000000000100），前 4 位是用来存储页面号，而后 12 位存储在页中的偏移量。

（0010 000000000100）根据前 4 位得到页号为 2，读取表项内容为（110 1），它的前 3 为为页框号，最后 1 位表示该页在内存中。最后映射得到物理内存地址为（110 000000000100）。

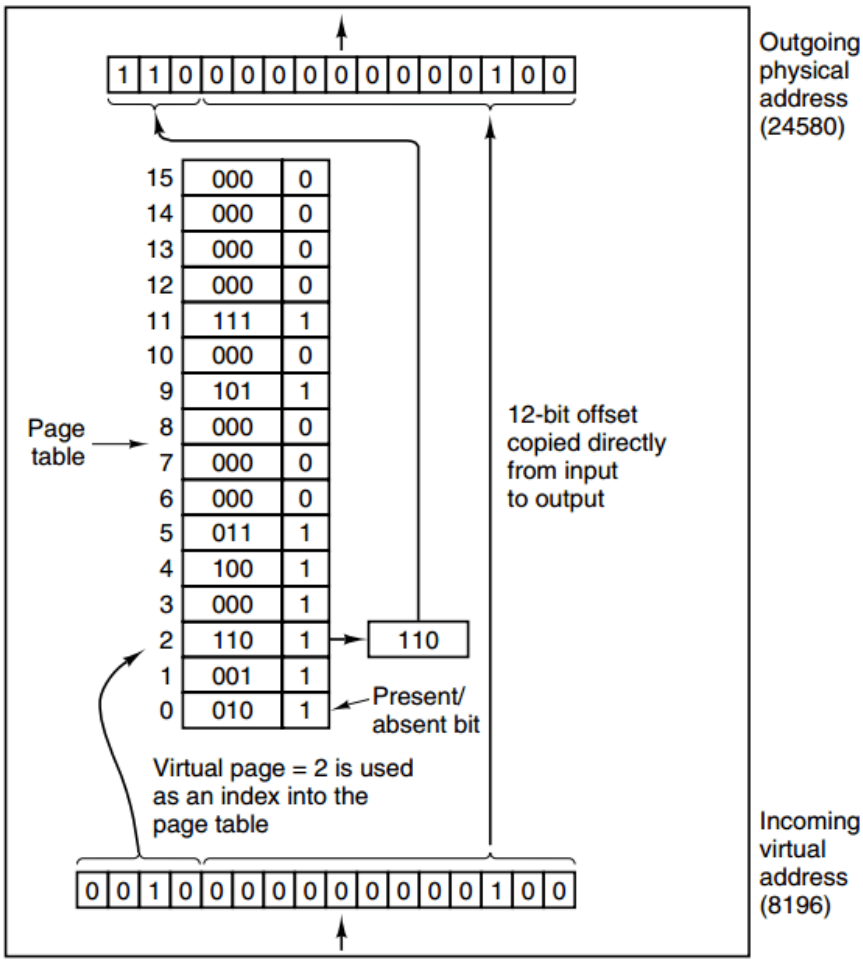


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

3. 页面置换算法

在程序运行过程中，如果要访问的页面不在内存中，就发生缺页中断从而将该页调入内存中。此时如果内存已无空闲空间，系统必须从内存中调出一个页面到磁盘对换区中来腾出空间。

页面置换算法和缓存淘汰策略类似，可以将内存看成磁盘的缓存。在缓存系统中，缓存的大小有限，当有新的缓存到达时，需要淘汰一部分已经存在的缓存，这样才有空间存放新的缓存数据。

页面置换算法的主要目标是使页面置换频率最低（也可以说缺页率最低）。

- Optimal

所选择的被换出的页面将是最长时间内不再被访问，通常可以保证获得最低的缺页率。

是一种理论上的算法，因为无法知道一个页面多长时间不再被访问。

例如：一个系统为某进程分配了三个物理块，并有如下页面引用序列： `70120304230321201701`

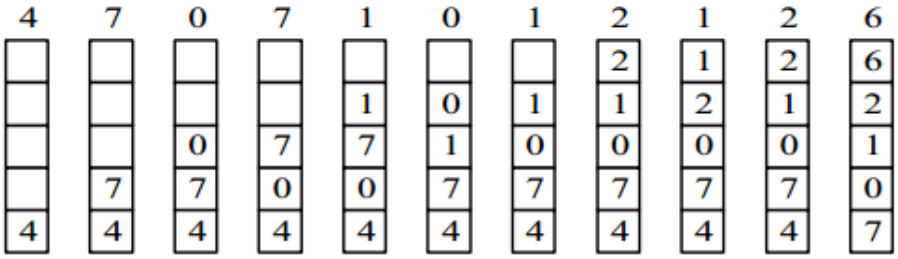
开始运行时，先将 7, 0, 1 三个页面装入内存。当进程要访问页面 2 时，产生缺页中断，会将页面 7 换出，因为页面 7 再次被访问的时间最长。

- Least Recently Used(LRU)

虽然无法知道将来要使用的页面情况，但是可以知道过去使用页面的情况。LRU 将最近最久未使用的页面换出。

为了实现 LRU，需要在内存中维护一个所有页面的链表。当一个页面被访问时，将这个页面移到链表表头。这样就能保证链表表尾的页面时最近最久未访问的。

因为每次访问都需要更新链表，因此这种方式实现的 LRU 代价很高。



- Not Recently Used(NRU)

每个页面都有两个状态位：R 与 M，当页面被访问时设置页面的 R=1，当页面被修改时设置 M=1。其中 R 位会定时被清零。可以将页面分成以下四类：

```
R=0, M=0
R=0, M=1
R=1, M=0
R=1, M=1
```

当发生缺页中断时，NRU 算法随机地从类编号最小的非空类中挑选一个页面将它换出。

NRU 优先换出已经被修改的脏页面（R=0，M=1），而不是被频繁使用的干净页面（R=1，M=0）。

4. First In First Out(FIFO)

选择换出的页面是最先进入的页面。

该算法会将那些经常被访问的页面也被换出，从而使缺页率升高。

5. 第二次机会算法

FIFO 算法可能会把经常使用的页面置换出去，为了避免这一问题，对该算法做一个简单的修改：

当页面被访问 (读或写) 时设置该页面的 R 位为 1。需要替换的时候，检查最老页面的 R 位。如果 R 位是 0，那么这个页面既老又没有被使用，可以立刻置换掉；如果是 1，就将 R 位清 0，并把该页面放到链表的尾端，修改它的装入时间使它就像刚装入的一样，然后继续从链表的头部开始搜索。

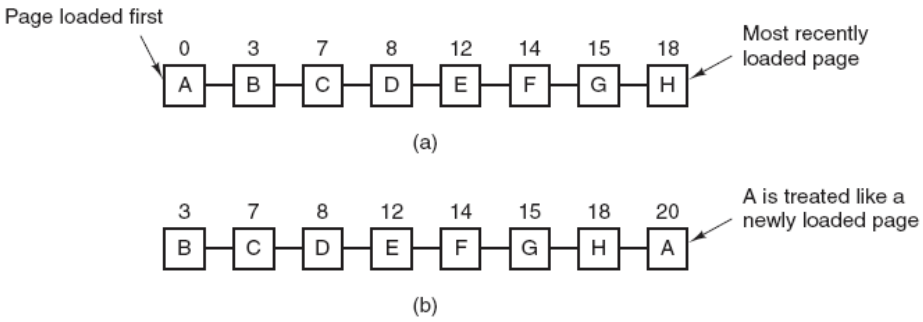


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

6. Clock

第二次机会算法需要在链表中移动页面，降低了效率。时钟算法使用环形链表将页面链接起来，再使用一个指针指向最老的页面。

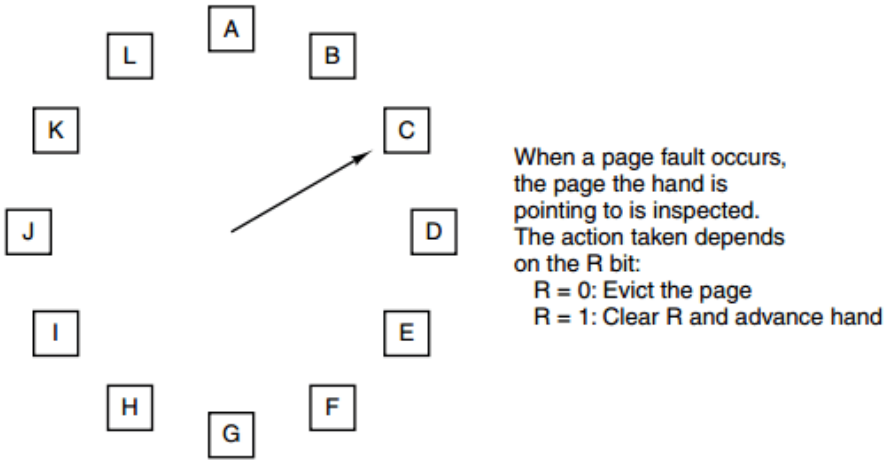
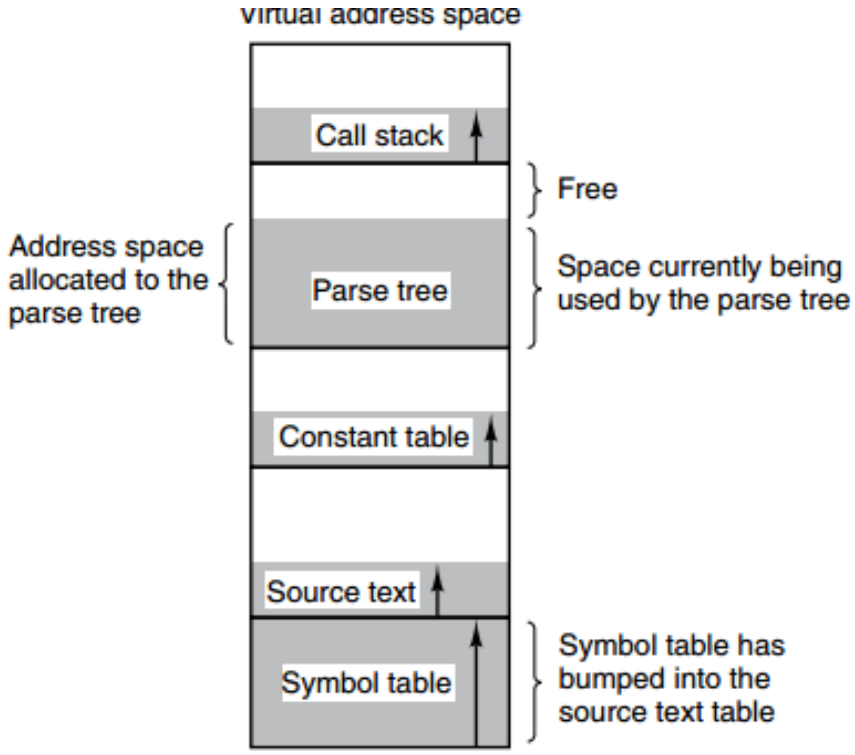


Figure 3-16. The clock page replacement algorithm.

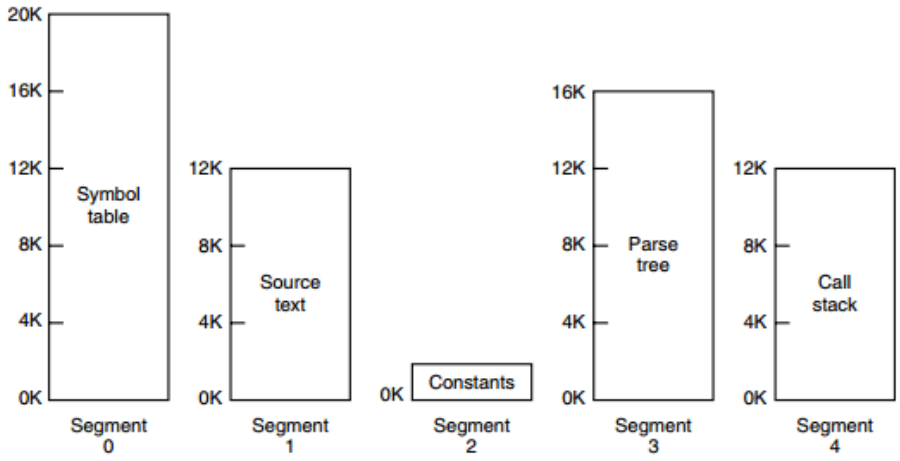
- 分段

虚拟内存采用的是分页技术，也就是将地址空间划分成固定大小的页，每一页再与内存进行映射。

下图为一个编译器在编译过程中建立的多个表，有 4 个表是动态增长的，如果使用分页系统的一维地址空间，动态增长的特点会导致覆盖问题的出现。



分段的做法是把每个表分成段，一个段构成一个独立的地址空间。每个段的长度可以不同，并且可以动态增长。



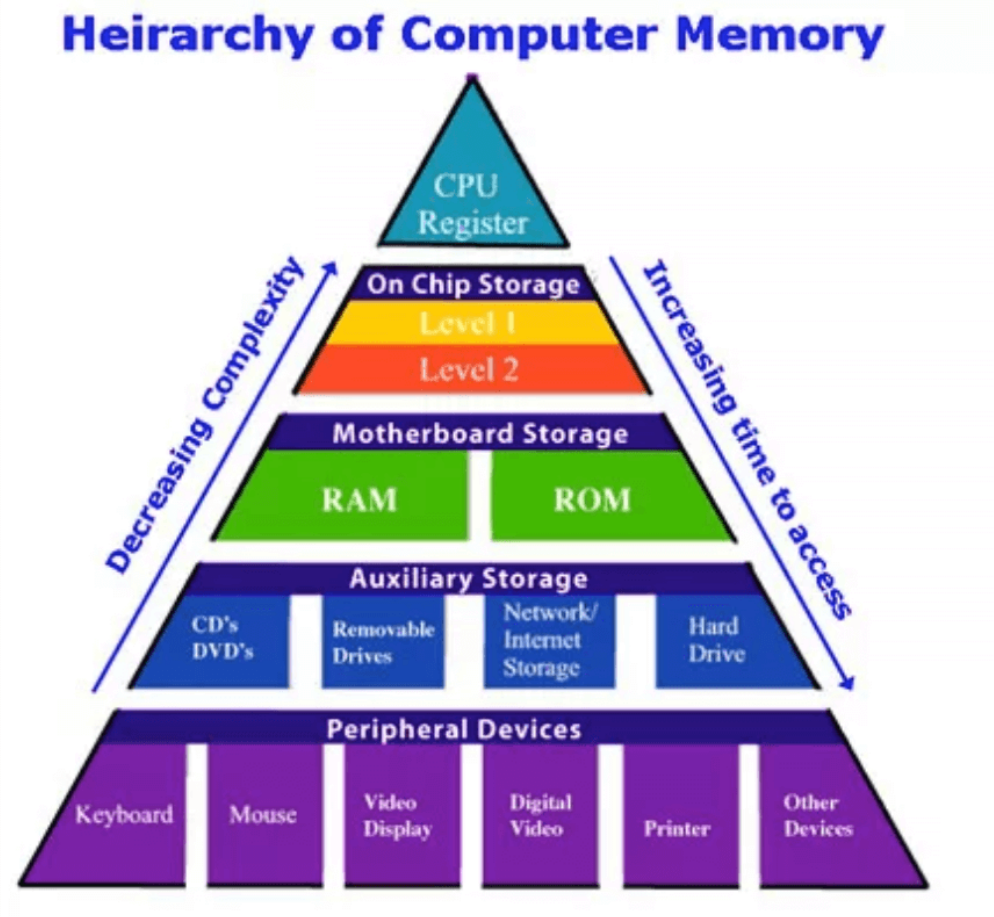
- 段页式

程序的地址空间划分成多个拥有独立地址空间的段，每个段上的地址空间划分成大小相同的页。这样既拥有分段系统的共享和保护，又拥有分页系统的虚拟内存功能。

- 分页与分段的比较

1. 对程序员的透明性：分页透明，但是分段需要程序员显示划分每个段。
2. 地址空间的维度：分页是一维地址空间，分段是二维的。
3. 大小是否可以改变：页的大小不可变，段的大小可以动态改变。
4. 出现的原因：分页主要用于实现虚拟内存，从而获得更大的地址空间；分段主要是为了使程序和数据可以被划分为逻辑上独立的地址空间并且有助于共享和保护。

这里在补充一下对比内存,寄存器、cache、内存、磁盘的各自特点.

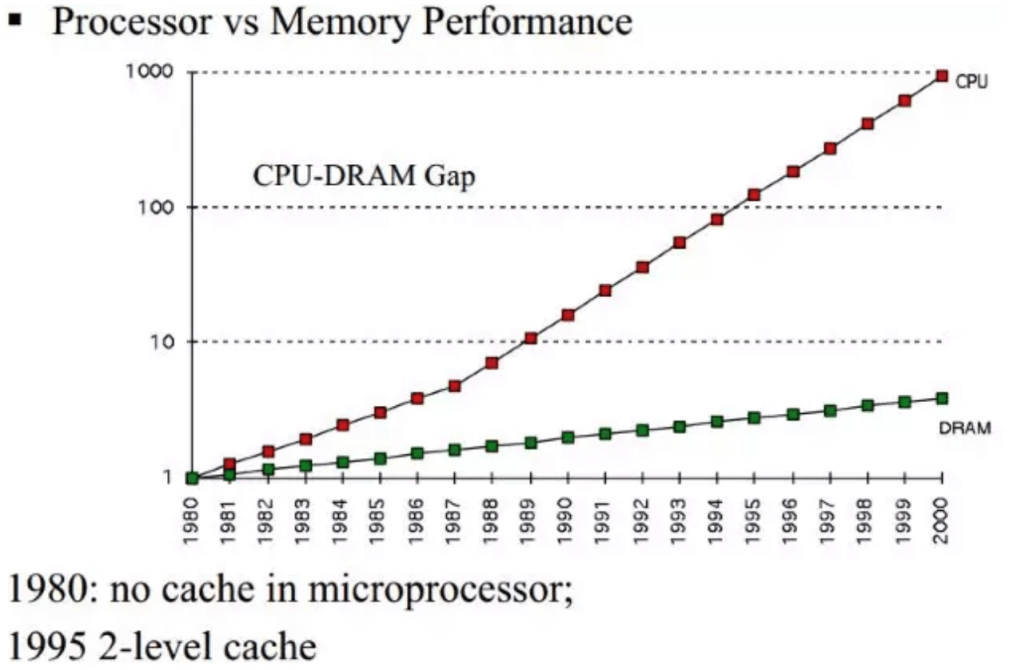


这个图表达了计算机的存储体系，从上至下依次是：

- CPU寄存器
- Cache
- 内存
- 硬盘等辅助存储设备
- 鼠标等外接设备

从上至下，访问速度越来越慢，访问时间越来越长。

CPU速度很快，但硬盘等持久存储很慢，如果CPU直接访问磁盘，磁盘可以拉低CPU的速度，机器整体性能就会低下，为了弥补这2个硬件之间的速率差异，所以在CPU和磁盘之间增加了比磁盘快很多的内存。



然而，CPU跟内存的速率也不是相同的，从上图可以看到，CPU的速率提高的很快（摩尔定律），然而内存速率增长的很慢，虽然CPU的速率现在增加的很慢了，但是内存的速率也没增加多少，速率差距很大，从1980年开始CPU和内存速率差距在不断拉大，为了弥补这2个硬件之间的速率差异，所以在CPU跟内存之间增加了比内存更快的Cache，Cache是内存数据的缓存，可以降低CPU访问内存的时间。

计算机的存储体系介绍:

1. 寄存器: 寄存器是中央处理器的组成部份，可用来暂存指令、数据和位址。通常有通用寄存器，如指令寄存器IR、程序计数器(PC)、累加器(ACC)、堆栈指针寄存器(SP)等，另外还有状态寄存器(标记状态Z、N、V、C)。寄存器最靠近CPU，随取随用，速度最快。

2. Cache:

即高速缓冲存储器，位于CPU与内存之间，容量小但速度快。由于CPU快而内存慢，CPU不存在直接读或者写内存的情况,每次读或者写内存都要访问Cache。

Cache Line是cache与内存同步的最小单位，典型的虚拟内存页面大小为4K，Cache line为32或64字节。Cache中一般保存着CPU刚用过或循环使用的部分数据，当CPU再次使用该部分数据时可从Cache中直接调用,这样就抹平了CPU与内存的速度差。Cache又分为L1、L2、L3（L1、L2一般集成在CPU上）。

理论上L1有着跟寄存器相同的速度，但L1工作在写通过(write-through)模式下时，需要加锁用来同步cache和内存的内容，这段期间L1不能被访问，所以L1就没寄存器快。L2、L3同样需要加锁，并且L2比L1慢，L3比L2慢。因此呢,L1速率最快，与CPU速率最接近，是RAM速率的100倍，L2速率就降到了RAM的25倍，L3的速率更靠近RAM的速率。

Cache下还有一个TLB，TLB是一个内存管理单元用于改进虚拟地址到物理地址转换速度的缓存。

3. RAM(主要针对DRAM):

即内存，其作用是用于暂存CPU的运算数据，以及与硬盘等外部存储器交换的数据。内存的一个存储周期是从存储器收到有效地址(EA)开始，经过地址译码、驱动，直到被访问的存储单元被读出/写入为止。

简单介绍下CPU访问内存的流程：

- 找到数据(一般为操作数)地址(或地址的地址)。（地址一般放在通用寄存器内）
- 将地址送往内存管理单元（MMU），由MMU将虚拟的内存地址翻译成实际的物理地址。
- 将物理地址送往内存控制器，由控制器进行译码找到对应的存储体。

- 从对应的存储体读取数据送回给控制器，最后再送回CPU。

可以看出内存的工作流程比cache多出许多，每一步都会产生延迟。并且当外围设备(比如磁盘)通过DMA控制器与内存进行数据传输时(走数据总线),会与上面的CPU访问内存发生冲突，此时CPU和DMA就会轮流挪用内存周期，这样CPU访问内存的速度就更慢了。

4. HardDisk

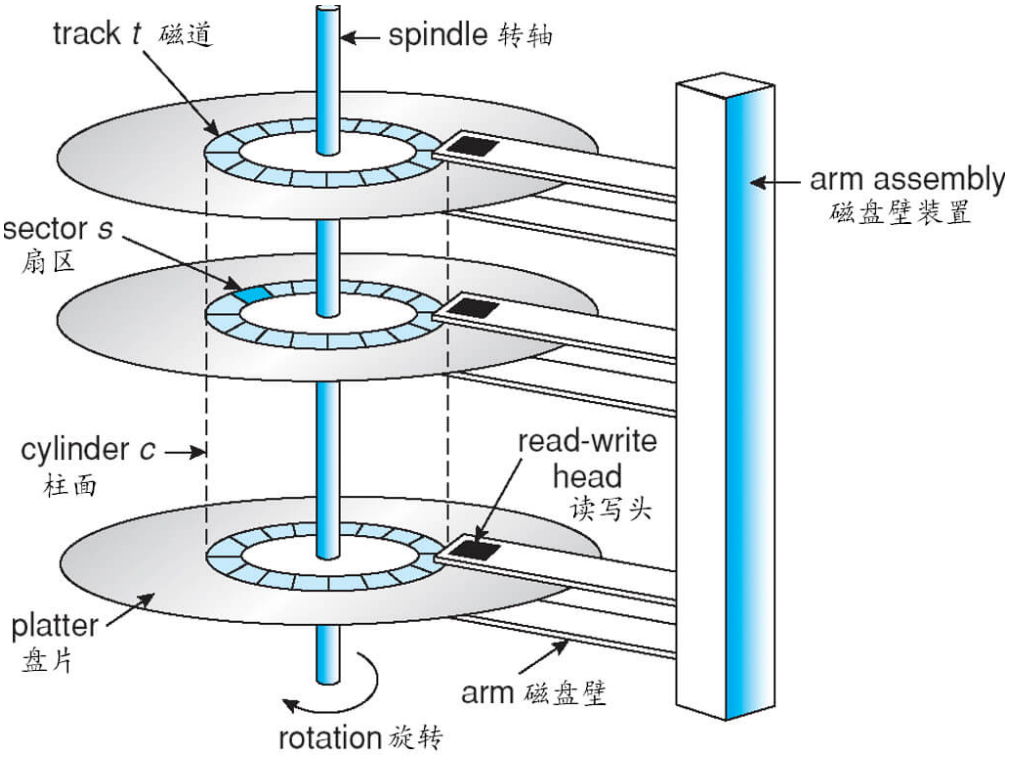
HardDisk又称硬盘驱动器，常见的有磁性旋转机械盘和基于闪存的固态硬盘SSD，这里主要讲机械盘，当进行数据存取时，主要的速度影响来自于磁头的寻道时间和盘片的旋转时间，通常需要花费数毫秒的时间。如果是顺序I/O还好，如果是随机I/O，速度将很慢。虽然磁盘内部、操作系统及应用程序都对磁盘进行了缓存优化，但速度还是远远不及内存。

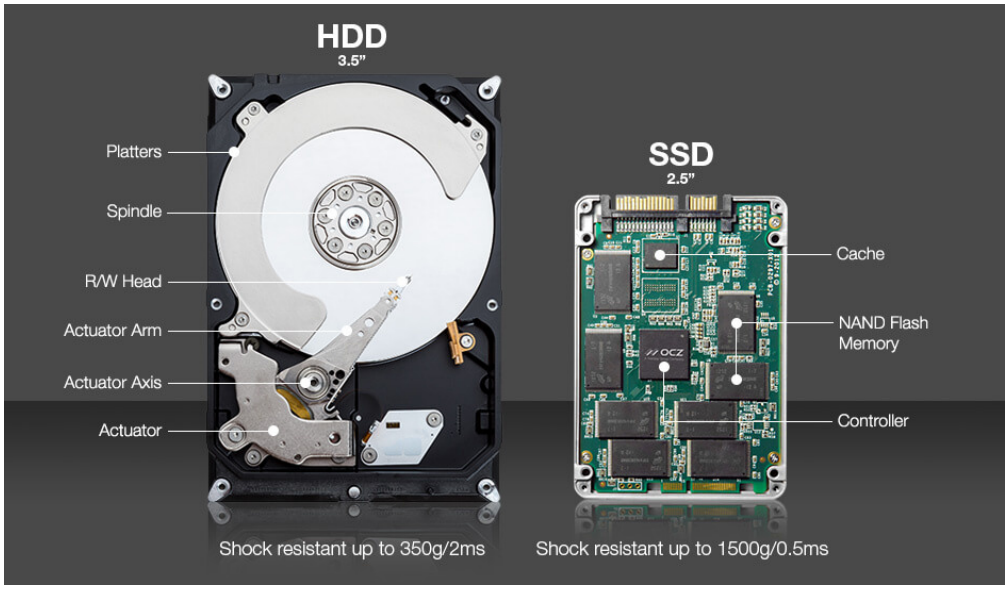
由此可以看到存储体系的分层设计,自顶向下，速率越来越低，访问时间越来越长，从磁盘到CPU寄存器，上一层都可以看做是下一层的缓存。

设备管理

磁盘结构主要包括:

- 盘面（Platter）：一个磁盘有多个盘面；
- 磁道（Track）：盘面上的圆形带状区域，一个盘面可以有多个磁道；
- 扇区（Track Sector）：磁道上的一个弧段，一个磁道可以有多个扇区，它是最小的物理储存单位，目前主要有 512 bytes 与 4 K 两种大小；
- 磁头（Head）：与盘面非常接近，能够将盘面上的磁场转换为电信号（读），或者将电信号转换为盘面的磁场（写）；
- 制动手臂（Actuator arm）：用于在磁道之间移动磁头；
- 主轴（Spindle）：使整个盘面转动。





磁盘调度算法:

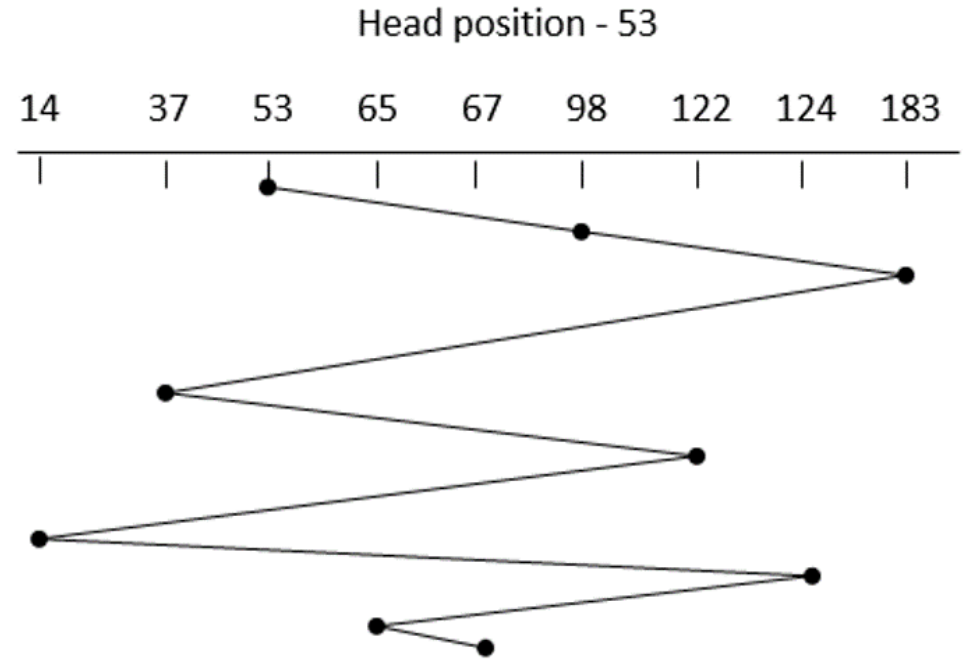
读写一个磁盘块的时间的影响因素有：

- 旋转时间（主轴旋转磁盘，使得磁头移动到适当的扇区上）
- 寻道时间（制动手臂移动，使得磁头移动到适当的磁道上）
- 实际的数据传输时间

  其中，寻道时间最长，因此磁盘调度的主要目标是使磁盘的平均寻道时间最短。

1. 先来先服务(First Come First Served,简称FCFS)

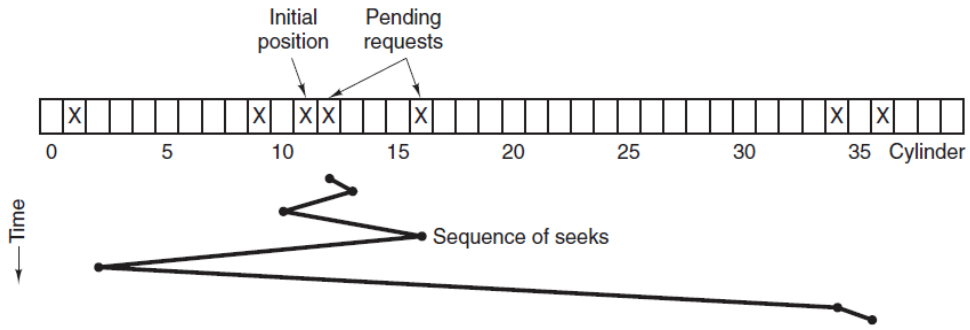
- 按照磁盘请求的顺序进行调度
- 公平对待所有进程
- 在有很多进程的情况下，接近随机调度的性能
- 优点是公平和简单。缺点也很明显，因为未对寻道做任何优化，使平均寻道时间可能较长。



2. 最短寻道时间优先(Shortest Seek Time First,简称SSTF)

优先调度与当前磁头所在磁道距离最近的磁道。

虽然平均寻道时间比较低，但是不够公平。如果新到达的磁道请求总是比一个在等待的磁道请求近，那么在等待的磁道请求会一直等待下去，也就是出现饥饿现象。具体来说，两边的磁道请求更容易出现饥饿现象。

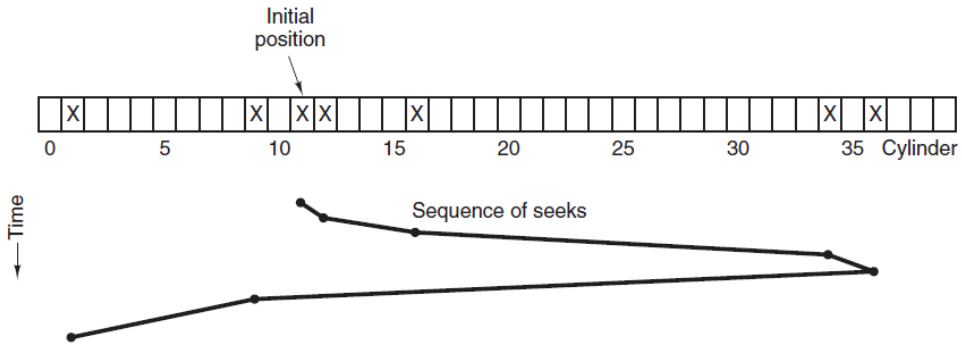


3. 电梯算法(SCAN)

电梯总是保持一个方向运行，直到该方向没有请求为止，然后改变运行方向。

电梯算法（扫描算法）和电梯的运行过程类似，总是按一个方向来进行磁盘调度，直到该方向上没有未完成的磁盘请求，然后改变方向。

因为考虑了移动方向，因此所有的磁盘请求都会被满足，解决了 SSTF 的饥饿问题。



系统处理过程

1. 编译系统

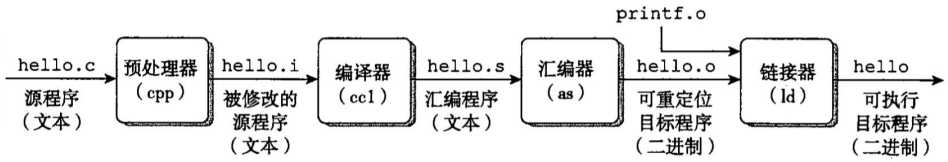
以下是一个 main.c 程序：

```
#include <stdio.h>
int main(){
    printf("Golang is Best Language\n");
    return 0;
}
```

在 Unix 系统上，由编译器把源文件转换为目标文件。

```
gcc -o main main.c
```

整个运行过程:



- 预处理阶段 (Preprocessing phase)

预处理（cpp）根据以字符 # 开头的命令，修改原始的 C 程序，生成扩展名为 .i 的文件。

```
> gcc -E main.c -o main.i
```

- 编译阶段 (Compilation phase)

编译器（cc1）将文本文件 main.i 翻译成文本文件 main.s，它包含一个汇编语言程序。

```
> gcc -S main.i -o main.s
```

- 汇编阶段 (Assembly phase)

编译器（as）将 main.s 翻译成机器语言指令，把这些指令打包成一种叫做可重定位目标程序（relocatable object program）的格式，并将结果保存在目标文件 main.o 中。

```
> as main.s -o main.o
```

- 链接阶段 (Linking phase)
  printf 函数是标准 C 库中的一个函数，在 printf.o 这个单独预编译好的目标文件中。连接器（ld）将 printf.o 和 main.o 合并，结果得到 main 可执行目标文件。
  ```
  > gcc main.o -o main
  ```

2. 静态链接

静态连接器以一组可重定向目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务：

- 符号解析：每个符号对应于一个函数、一个全局变量或一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来。
- 重定位：链接器通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得它们指向这个内存位置。

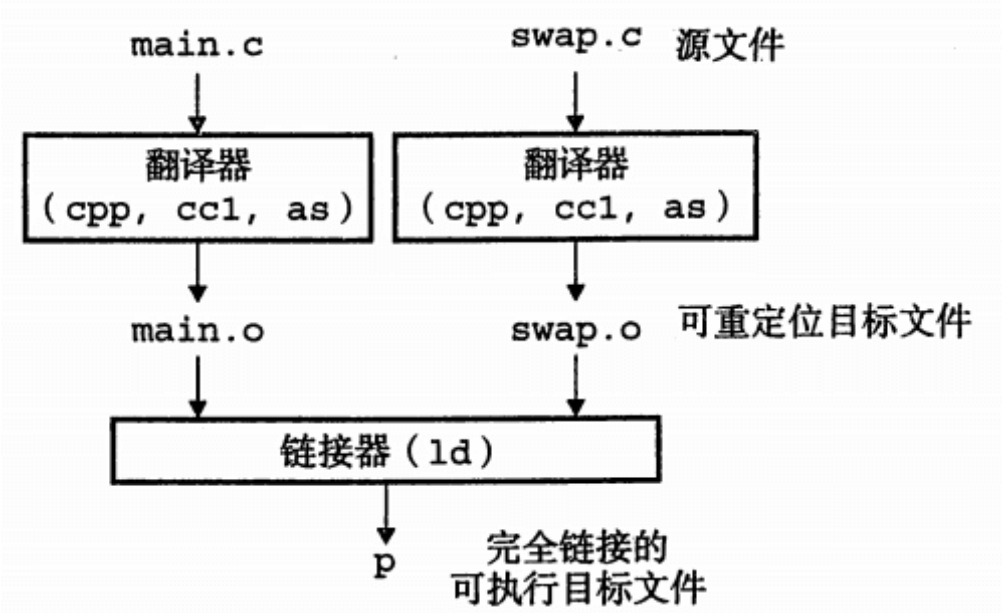


图 7-2 静态链接。链接器将可重定位目标文件组合起来，形成一个可执行目标文件

3. 目标文件

- 可执行目标文件：可以直接在内存中执行；
- 可重定向目标文件：可与其它可重定向目标文件在链接阶段合并，创建一个可执行目标文件；
- 共享目标文件：这是一种特殊的可重定向目标文件，可以在运行时被动态加载进内存并链接；

4. 动态链接

静态库有以下两个问题：

- 当静态库更新时那么整个程序都要重新进行链接；

- 对于 printf 这种标准函数库，如果每个程序都要有代码，这会极大浪费资源。

共享库是为了解决静态库的这两个问题而设计的，在 Linux 系统中通常用 .so 后缀来表示，Windows 系统上它们被称为 DLL。它具有以下特点：

- 在给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中；
- 在内存中，一个共享库的 .text 节（已编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享。

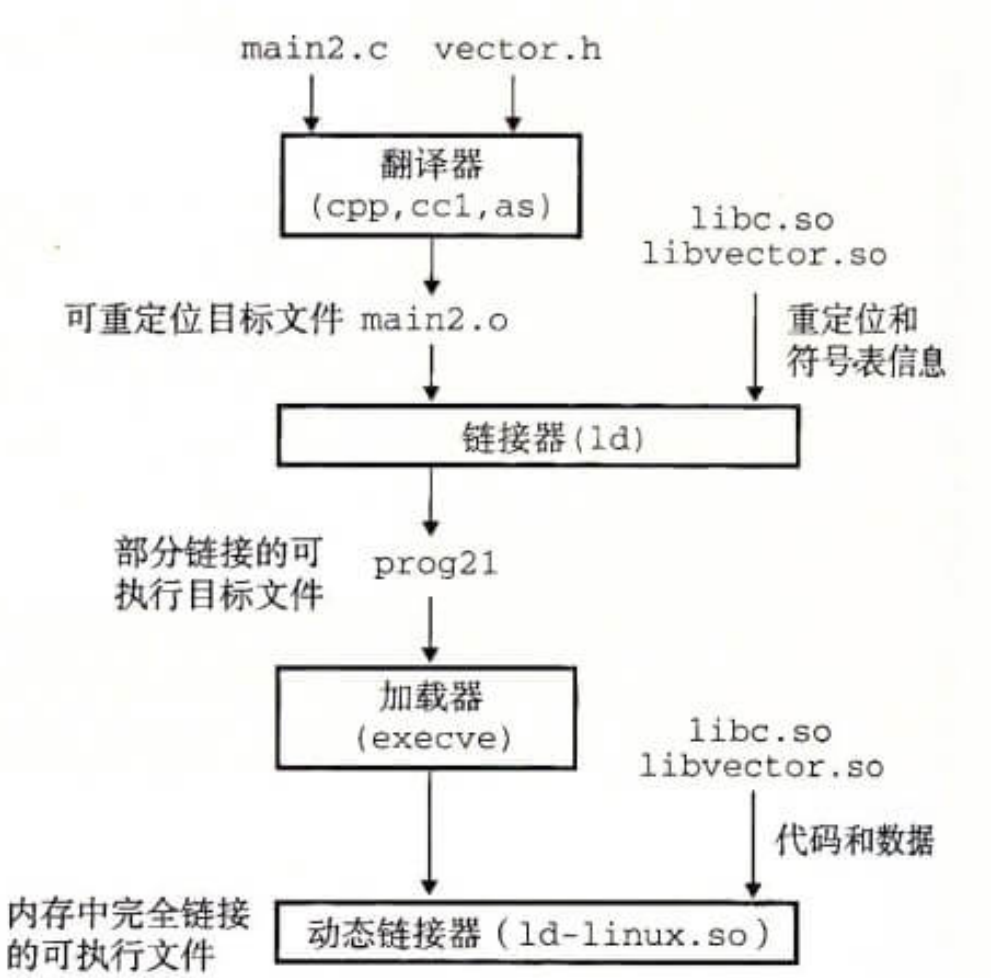


图 7-16　动态链接共享库

Golang的堆栈分配

Go的堆栈

在理解Go的堆栈分配前,我们先理解下什么是堆栈？在计算机中堆栈的概念分为：数据结构的堆栈和内存分配中堆栈。

数据结构的堆栈：

堆: 堆可以被看成是一棵树，如：堆排序。在队列中，调度程序反复提取队列中第一个作业并运行，因为实际情况中某些时间较短的任务将等待很长时间才能结束，或者某些不短小，但具有重要性的作业，同样应当具有优先权。堆即为解决此类问题设计的一种数据结构。

栈: 一种先进后出的数据结构。

在内存分配中的堆和栈:

栈（操作系统）:由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆（操作系统）:一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

堆栈缓存方式

栈使用的是一级缓存, 他们通常都是被调用时处于存储空间中，调用完毕立即释放。

堆则是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

变量是堆（heap）还是堆栈（stack）

官方给出的解释如下:

```
How do I know whether a variable is allocated on the heap or the stack?
From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.

The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.

In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic escape analysis recognizes some cases when such variables will not live past the return from the function and can reside on the stack.
```

从上面可以了解到, 您不需要知道。Go中的每个变量都存在，只要有对它的引用即可。实现选择的存储位置与语言的语义无关。

存储位置确实会影响编写高效的程序。如果可能，Go编译器将为该函数的堆栈帧中的函数分配本地变量。但是，如果编译器在函数返回后无法证明变量未被引用，则编译器必须在垃圾收集堆上分配变量以避免悬空指针错误。此外，如果局部变量非常大，将它存储在堆而不是堆栈上可能更有意义。

在当前的编译器中，如果变量具有其地址，则该变量是堆上分配的候选变量。但是，基础的逃逸分析可以将那些生存不超过函数返回值的变量识别出来，并且因此可以分配在栈上。

Go的编译器会决定在哪(堆or栈)分配内存，保证程序的正确性。

Go的堆栈分配

- 每个goroutine维护着一个栈空间，默认最大为4KB.
- 当goroutine的栈空间不足时，golang会调用 `runtime.morestack` (汇编实现：asm_xxx.s)来进行动态扩容.
- 连续栈是当栈空间不足的时候申请一个2倍于当前大小的新栈，并把所有数据拷贝到新栈,接下来的所有调用执行都发生在新栈上.
- 每个function维护着各自的栈帧(stack frame)，当function退出时会释放栈帧.

Go function内的栈操作

用一段简单的代码来说明Go函数调用及传参时的栈操作：

```
package main

func g(p int) int {
    return p+1;
}

func main() {
    c := g(4) + 1
    _ = c
}
```

执行 `go tool compile -S main.go` 生成汇编，并截取其中的一部分来说明一下程序调用时的栈操作.

```
"".g t=1 size=17 args=0x10 locals=0x0
    // 初始化函数的栈地址
    // 0-16表示函数初始地址为0，数据大小为16字节(input: 8字节，output: 8字节)
    // SB是函数寄存器
    0x0000 00000 (test_stack.go:3)  TEXT    "".g(SB), $0-16
    // 函数的gc收集提示。提示0和1是用于局部函数调用参数，需要进行回收
    0x0000 00000 (test_stack.go:3)  FUNCDATA    $0, gclocals·aef1f7ba6e2630c93a51843d99f5a28a(SB)
    0x0000 00000 (test_stack.go:3)  FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    // FP(frame point)指向栈底
    // 将FP+8位置的数据(参数p)放入寄存器AX
    0x0000 00000 (test_stack.go:4)  MOVQ    "".p+8(FP), AX
    0x0005 00005 (test_stack.go:4)  MOVQ    (AX), AX
    // 寄存器值自增
    0x0008 00008 (test_stack.go:4)  INCQ    AX
    // 从寄存器中取出值，放入FP+16位置(返回值)
    0x000b 00011 (test_stack.go:4)  MOVQ    AX, "".~r1+16(FP)
    // 返回，返回后程序栈的空间会被回收
    0x0010 00016 (test_stack.go:4)  RET
    0x0000 48 8b 44 24 08 48 8b 00 48 ff c0 48 89 44 24 10  H.D$.H..H..H.D$.
    0x0010 c3                                               .
"".main t=1 size=32 args=0x0 locals=0x10
    0x0000 00000 (test_stack.go:7)  TEXT    "".main(SB), $16-0
    0x0000 00000 (test_stack.go:7)  SUBQ    $16, SP
    0x0004 00004 (test_stack.go:7)  MOVQ    BP, 8(SP)
    0x0009 00009 (test_stack.go:7)  LEAQ    8(SP), BP
    0x000e 00014 (test_stack.go:7)  FUNCDATA    $0, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x000e 00014 (test_stack.go:7)  FUNCDATA    $1, gclocals·33cdeccccebe80329f1fdbee7f5874cb(SB)
    // SP(stack point)指向栈顶
    // 把4存入SP的位置
    0x000e 00014 (test_stack.go:8)  MOVQ    $4, "".c(SP)
    // 这里会看到没有第9行`call g()`的调用出现，这是因为go汇编编译器会把一些短函数变成内嵌函数，减少函数调用
    0x0016 00022 (test_stack.go:10) MOVQ    8(SP), BP
```

```
    0x001b 00027 (test_stack.go:10) ADDQ    $16, SP
    0x001f 00031 (test_stack.go:10) RET
```

事实上，即便我定义了指针调用，以上的数据也都是在栈上拷贝的；那么Golang中的数据什么时候会被分配到堆上呢？

Golang逃逸分析

- 在编译程序优化理论中，逃逸分析是一种确定指针动态范围的方法，用于分析在程序的哪些地方可以访问到指针。
- Golang在编译时的逃逸分析可以减少gc的压力，不逃逸的对象分配在栈上，当函数返回时就回收了资源，不需要gc标记清除。
- 如果你定义的对象的方法上有同步锁，但在运行时，却只有一个线程在访问，此时逃逸分析后的机器码，会去掉同步锁运行，提高效率。

还是上面的那段程序代码，我们可以执行 `go build -gcflags '-m -l' test_stack.go` 来进行逃逸分析，输出结果如下

```
# command-line-arguments
./test_stack.go:3: g p does not escape
./test_stack.go:9: main &c does not escape
```

可以看到，对象c是没有逃逸的，还是分配在栈上。

即便在一开始定义的时候直接把c定义为指针：

```
package main

func g(p *int) int {
    return *p + 1
}

func main() {
    c := new(int)
    (*c) = 4
    _ = g(c)
}
```

逃逸分析的结果仍然不会改变:

```
# command-line-arguments
./test_stack.go:3: g p does not escape
./test_stack.go:8: main new(int) does not escape
```

那么，在什么时候指针对象才会逃逸呢？

那就是在按值传递和按址传递时候.

- 按值传递

```
package main

func g(p int) int {
    ret := p + 1
    return ret
}
```

```
func main() {
    c := 4
    _ = g(c)
}
```

返回值ret是按值传递的，执行的是栈拷贝，不存在逃逸.

- 按址传递

```
package main

func g(p *int) *int {
    ret := *p + 1
    return &ret
}

func main() {
    c := new(int)
    *c = 4
    _ = g(c)
}
```

返回值&ret是按址传递，传递的是指针对象，发生了逃逸，将对象存放在堆上以便外部调用.

```
# command-line-arguments
./test_stack.go:5:9: &ret escapes to heap
./test_stack.go:4:14: moved to heap: ret
./test_stack.go:3:17: g p does not escape
./test_stack.go:9:10: main new(int) does not escape
```

golang只有在function内的对象可能被外部访问时，才会把该对象分配在堆上.

- 在g()方法中，ret对象的引用被返回到了方法外，因此会发生逃逸；而p对象只在g()内被引用，不会发生逃逸.
- 在main()方法中，c对象虽然被g()方法引用了，但是由于引用的对象c没有在g()方法中发生逃逸，因此对象c的生命周期还是在main()中的，不会发生逃逸.

```
package main

type Result struct {
    Data *int
}

func g(p *int) *Result {
    var ret Result
    ret.Data = p
    return &ret
}

func main() {
    c := new(int)
    *c = 4
    _ = g(c)
}
```

逃逸分析结果

```
# command-line-arguments
./test_stack.go:10:9: &ret escapes to heap
./test_stack.go:8:6: moved to heap: ret
./test_stack.go:7:17: leaking param: p to result ~r1 level=-1
./test_stack.go:14:10: new(int) escapes to heap
```

- 可以看到，ret和2.2中一样，存在外部引用，发生了逃逸.
- 由于`ret.Data`是一个指针对象，p赋值给`ret.Data`后，也伴随p发生了逃逸.
- main()中的对象c，由于作为参数p传入g()后发生了逃逸，因此c也发生了逃逸.
- 当然，如果定义`ret.Data`为int(instead of *int)的话，对象p也是不会逃逸的(执行了拷贝).

开发建议大对象按址传递，小对象按值传递

- 按址传递更高效，按值传递更安全(from William Kennedy).
- 90%的bug都来自于指针调用.

初始化一个结构体，使用引用的方式来传递指针

```
func r() *Result{
    var ret Result
    ret.Data = ...
    ...
    return &ret
}
```

只有返回ret对象的引用时才会把对象分配在堆上，我们不必要在一开始的时候就显式地把ret定义为指针.

```
ret = &Result{}
...
return ret
```

对阅读代码也会容易产生误导.

参考链接

- Golang汇编快速指南
- Golang汇编
- Golang汇编命令解读
- go语言连续栈
- 为何说Goroutine的栈空间可以无限大
- Goroutine stack

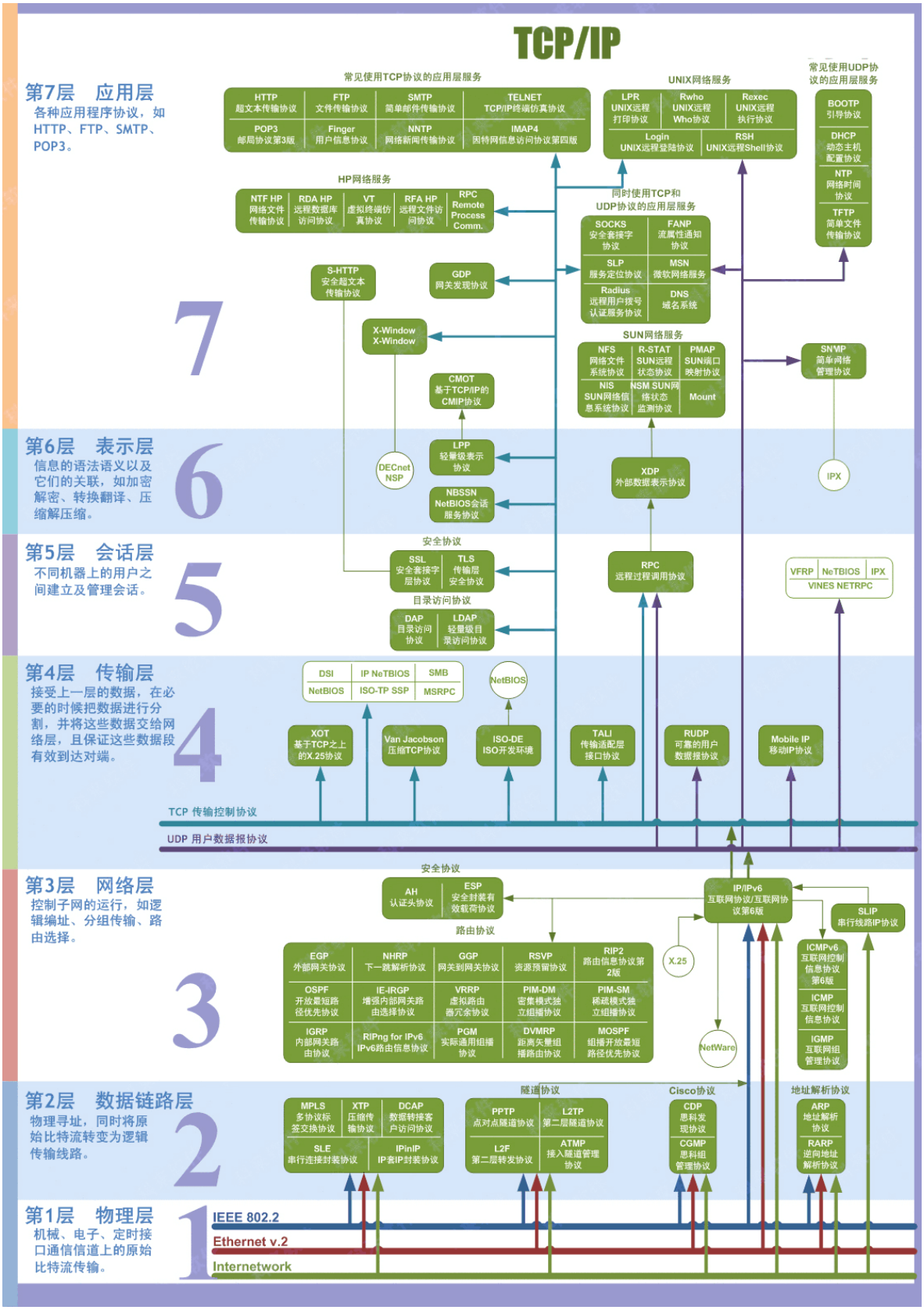
计算机网络基础知识

计算机网络基础知识

计算机网络基础知识,主要围绕网络层、传输层、应用层，核心为 TCP 和 HTTP 两部分。

第一部分：传输层

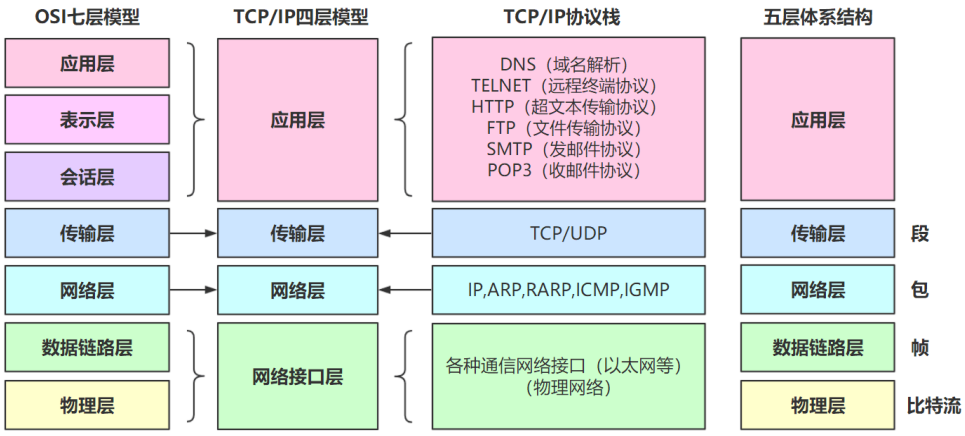
1. OSI七层模型 TCP/IP四层模型 ,五层协议.



（1）五层协议

- **应用层**：提供用户接口，特指能够发起网络流量的程序，比如客户端程序：QQ，MSN，浏览器等；服务器程序：web服务器，邮件服务器，流媒体服务器等等。数据单位为报文。

- **运输层**：提供的是进程间的通用数据传输服务。由于应用层协议很多，定义通用的运输层协议就可以支持不断增多的应用层协议。运输层包括两种协议：
  - 传输控制协议 TCP，提供面向连接、可靠的数据传输服务，数据单位为报文段；
  - 用户数据报协议 UDP，提供无连接、尽最大努力的数据传输服务，数据单位为用户数据报。
  - TCP 主要提供完整性服务，UDP 主要提供及时性服务。
- **网络层**：为主机间提供数据传输服务，而运输层协议是为主机中的进程提供服务。网络层把运输层传递下来的报文段或者用户数据报封装成分组。（负责选择最佳路径 规划IP地址）
  - 路由器查看数据包目标IP地址，根据路由表为数据包选择路径。路由表中的类目可以人工添加（静态路由）也可以动态生成（动态路由）。
- **数据链路层**：不同的网络类型，发送数据的机制不同，数据链路层就是将数据包封装成能够在不同的网络传输的帧。能够进行差错检验，但不纠错，监测处错误丢掉该帧。
  - 帧的开始和结束，透明传输，差错校验
- **物理层**：物理层解决如何在连接各种计算机的传输媒体上传输数据比特流，而不是指具体的传输媒体。物理层的主要任务描述为：确定与传输媒体的接口的一些特性，即：
  - 机械特性：例接口形状，大小，引线数目.
  - 电气特性：例规定电压范围 (-5V 到 +5V)
  - 功能特性：例规定 -5V 表示 0，+5V 表示 1.
  - 过程特性：也称规程特性，规定建立连接时各个相关部件的工作步骤.



(2) ISO七层模型中表示层和会话层功能是什么？

- **表示层**：数据压缩、加密以及数据描述。这使得应用程序不必担心在各台主机中表示/存储的内部格式（二进制、ASCII，比如乱码）不同的问题。

- **会话层**：建立会话，如session认证、断点续传。通信的应用程序之间建立、维护和释放面向用户的连接。通信的应用程序之间建立会话，需要传输层建立1个或多个连接。（...后台运行的木马，netstat -n）

- 说明：五层协议没有表示层和会话层，而是将这些功能留给应用程序开发者处理。

（3）数据在各层之间的传递过程

在向下的过程中，需要添加下层协议所需要的首部或者尾部，而在向上的过程中不断拆开首部和尾部。

1. 路由器只有下面三层协议(三层是路由表,网络的本质其实就是路由)，因为路由器位于网络核心中，不需要为进程或者应用程序提供服务，因此也就不需要运输层和应用层。

2. 交换机只有下面两层协议(局域网就是两层协议通信,两层是转发表).

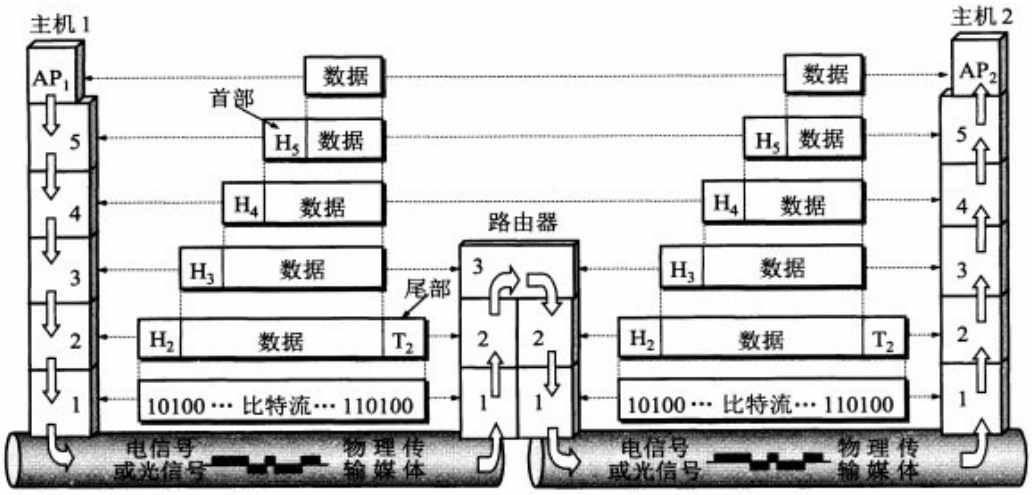


图 1-19　数据在各层之间的传递过程

（4）TCP/IP四层模型

它只有四层，相当于五层协议中**数据链路层和物理层合并为网络接口层**。

现在的 TCP/IP 体系结构不严格遵循 OSI 分层概念，应用层可能会直接使用 IP 层或者网络接口层。

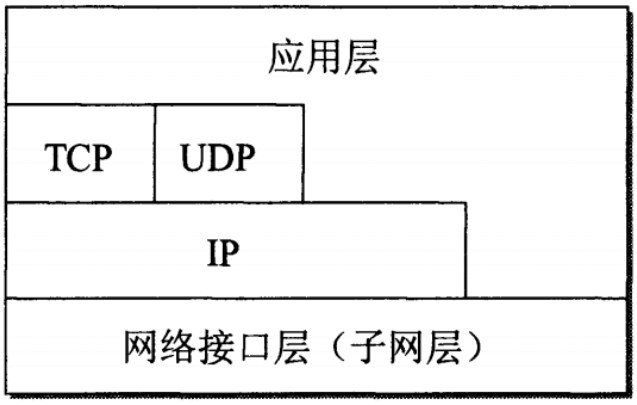


图 1-23　TCP/IP 体系结构的另一种表示方法

TCP/IP 协议族是一种沙漏形状，中间小两边大，IP 协议在其中占用举足轻重的地位。

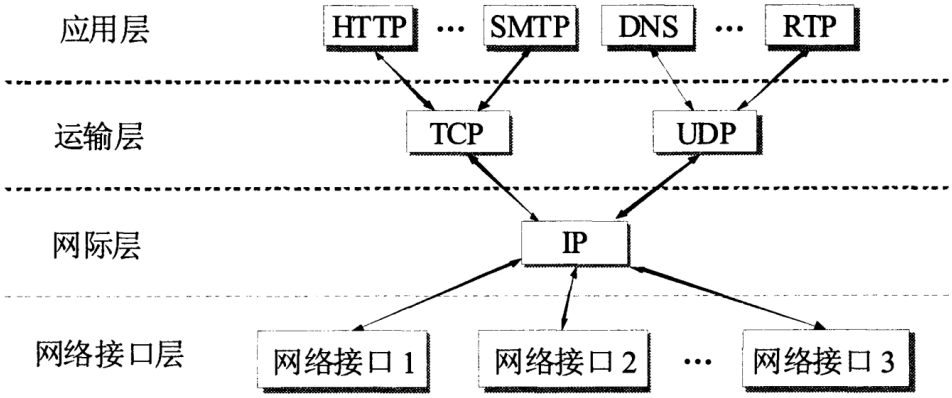


图 1-24　沙漏计时器形状的 TCP/IP 协议族示意

2. TCP报头格式和UDP报头格式

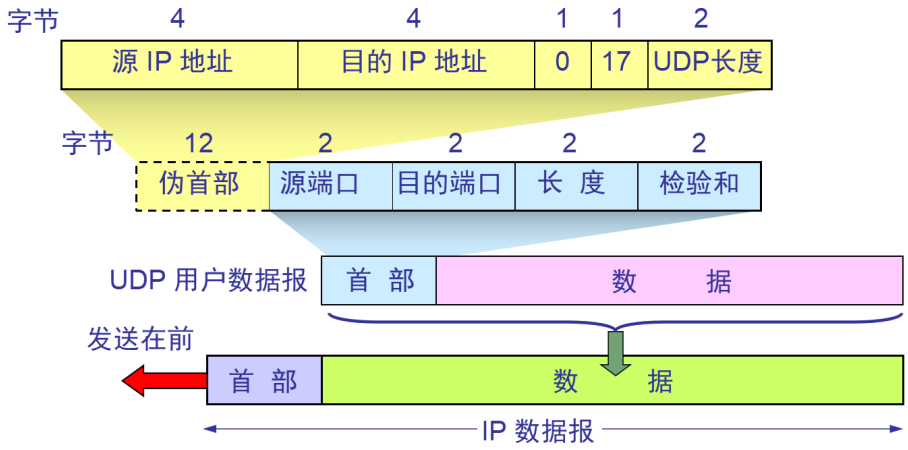
网络层只把分组发送到目的主机，但是真正通信的并不是主机而是主机中的进程。运输层提供了进程间的逻辑通信，运输层向高层用户屏蔽了下面网络层的核心细节，使应用程序看起来像是在两个运输层实体之间有一条端到端的逻辑通信信道。

（1）UDP 和 TCP 的特点

- **用户数据报协议 UDP**（User Datagram Protocol）是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。例如：视频传输、实时通信。

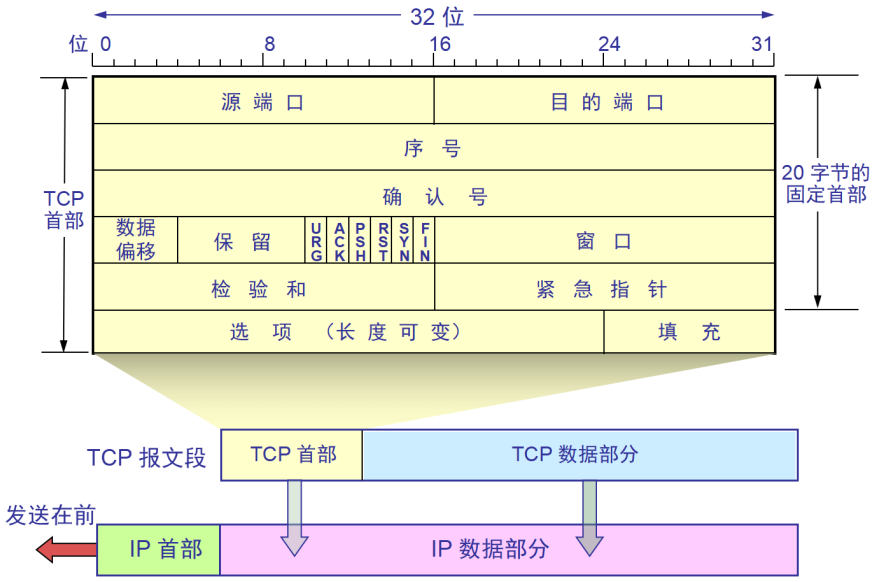
- **传输控制协议 TCP**（Transmission Control Protocol）是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。

（2）UDP 首部格式



首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

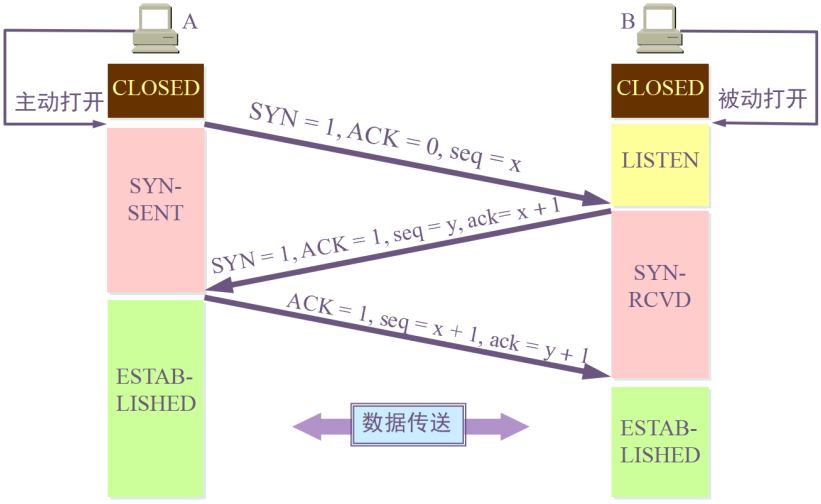
（3）TCP 首部格式



- **序号 seq**：用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。[301,400]为序号301的数据长度，下一个则为401。
- **确认号 ack**：期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
- **数据偏移**：指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
- **确认 ACK**：当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
- **同步 SYN**：在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
- **终止 FIN**：用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
- **窗口**：窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。

3. TCP三次握手？那四次挥手呢？如何保障可靠传输

（1）三次握手



假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文段，SYN=1，ACK=0，选择一个初始的序号 seq = x。
- B 收到连接请求报文段，如果同意建立连接，则向 A 发送连接确认报文段，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 seq = y。
- A 收到 B 的连接确认报文段后，还要向 B 发出确认，确认号为 ack = y+1，序号为 seq = x+1。
- A 的 TCP 通知上层应用进程，连接已经建立。
- B 收到 A 的确认后，连接建立。
- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。

（2）为什么TCP连接需要三次握手，两次不可以吗，为什么

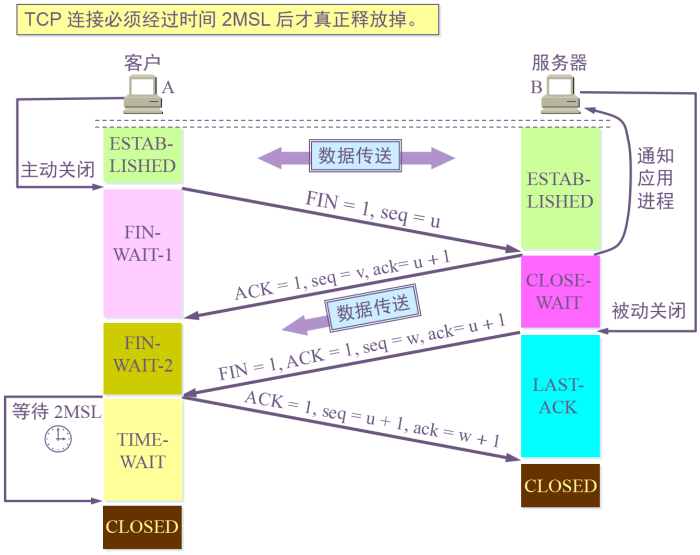
为了防止已失效的连接请求报文段突然又传送到了服务端，占用服务器资源。（假设主机A为客户端，主机B为服务器端）

现假定出现一种异常情况，即A发出的第一个连接请求报文段并没有丢失，而是在某些网络节点长时间滞留了，以致延误到连接释放以后的某个时间才到B。本来这是一个已失效的报文段。但是B收到此失效的连接请求报文段后，就误认为是A有发出一次新的连接请求。于是就向A发出确认报文段，同意建立连接。假定不采用三次握手，那么只要B发出确认，新的连接就建立了。

由于现在A并没有发出建立连接的请求，因此不会理睬B的确认，也不会向B发送数据。但B却以为新的运输连接已经建立了，并一直等待A发来数据。B的许多资源就这样白白浪费了。

采用三次握手的办法可以防止上述现象的发生。例如在刚才的情况下，A不会向B的确认发出确认。B由于收不到确认，就知道A并没有要求建立连接。

（3）四次挥手



数据传输结束后，通信的双方都可释放连接。现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP连接。

- A 把连接释放报文段首部的 FIN = 1，其序号 seq = u，等待 B 的确认。
- B 发出确认，确认号 ack = u+1，而这个报文段自己的序号 seq = v。（TCP 服务器进程通知高层应用进程）。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。A 不能向 B 发送数据；B 若发送数据，A 仍要接收。
- 当 B 不再需要连接时，发送连接释放请求报文段，FIN=1。
- A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（2*2 = 4 mins）时间后释放连接。

- B 收到 A 的确认后释放连接。

（4）四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

（5）TIME_WAIT

> MSL是Maximum Segment Lifetime英文的缩写，中文可以译为“报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。2MSL = 2*2mins = 4mins

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文段能够到达。如果 B 没收到 A 发送来的确认报文段，那么就会重新发送连接释放请求报文段，A 等待一段时间就是为了处理这种情况的发生。

- 等待一段时间是为了让本连接持续时间内所产生的所有报文段都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文段。

（6）如何保证可靠传输

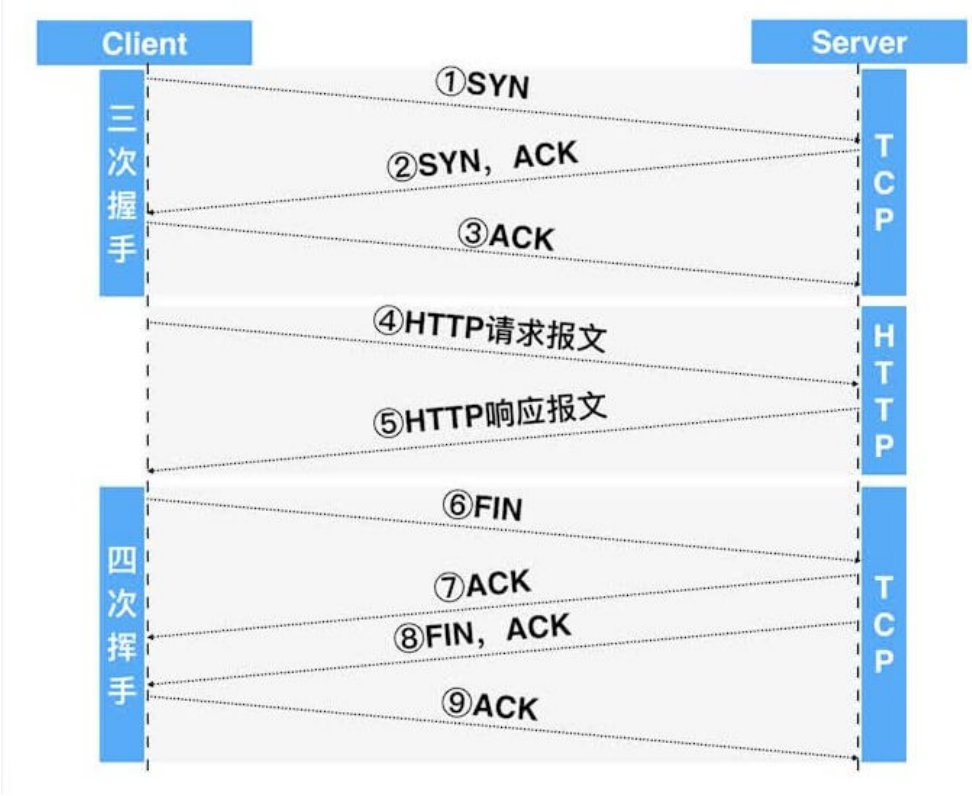
- 应用数据被分割成TCP认为最适合发送的数据块。
- **超时重传**：当TCP发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。
- TCP给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
- **校验和**：TCP将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP将丢弃这个报文段和不确认收到此报文段。
- TCP的接收端会丢弃重复的数据。
- **流量控制**：TCP连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的我数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP使用的流量控制协议是可变大小的滑动窗口协议。
- **拥塞控制**：当网络拥塞时，减少数据的发送。

（7）TCP连接状态？

- CLOSED：初始状态。
- LISTEN：服务器处于监听状态。
- SYN_SEND：客户端socket执行CONNECT连接，发送SYN包，进入此状态。
- SYN_RECV：服务端收到SYN包并发送服务端SYN包，进入此状态。
- ESTABLISH：表示连接建立。客户端发送了最后一个ACK包后进入此状态，服务端接收到ACK包后进入此状态。
- FIN_WAIT_1：终止连接的一方（通常是客户机）发送了FIN报文后进入。等待对方FIN。
- CLOSE_WAIT：（假设服务器）接收到客户机FIN包之后等待关闭的阶段。在接收到对方的FIN包之后，自然是需要立即回复ACK包的，表示已经知道断开请求。但是本方是否立即断开连接（发送FIN包）取决于是否还有数据需要发送给客户端，若有，则在发送FIN包之前均为此状态。
- FIN_WAIT_2：此时是半连接状态，即有一方要求关闭连接，等待另一方关闭。客户端接收到服务器的ACK包，但并没有立即接收到服务端的FIN包，进入FIN_WAIT_2状态。
- LAST_ACK：服务端发动最后的FIN包，等待最后的客户端ACK响应，进入此状态。

- TIME_WAIT：客户端收到服务端的FIN包，并立即发出ACK包做最后的确认，在此之后的2MSL时间称为TIME_WAIT状态。

（8）TCP和HTTP



4. TCP连接中如果断电怎么办

TCP是一种有连接的协议，但是这个连接并不是指有一条实际的电路，而是一种虚拟的电路。TCP的建立连接和断开连接都是通过发送数据实现的，也就是我们常说的三次握手、四次挥手。TCP两端保存了一种数据的状态，就代表这种连接，TCP两端之间的路由设备只是将数据转发到目的地，并不知道这些数据实际代表了什么含义，也并没有在其中保存任何的状态信息，也就是说中间的路由设备没有什么连接的概念，只是将数据转发到目的地，只有数据的发送者和接受者两端真正的知道传输的数据代表着一条连接。

但是这就说明了一点，如果不发送数据那么是无法断开连接的。正常情况下当TCP的一端A调用了SOCKET的close或者进程结束，操作系统就会按照TCP协议发送FIN数据报文。B端收到后就会断开连接。但是当出现了上文所说的异常情况时：被拔掉网线或者断掉电源，总结起来就是没有机会发出断开的FIN数据报文。那么和A直连的路由设备虽然知道A设备已经断开了，但是路由设备并没有保存连接的状态信息，所以路由设备也就不可能去通知B端A端的断开。而B端没有收到断开的数据报文就会依然保持连接。所以A端拔掉网线或者断掉电源后B端是没办法收到断开连接的通知的。

保持连接并不是毫无代价的，如果这种异常断开的连接有很多，那么势必会耗费大量的资源，必须要想办法检测出这种异常连接。
检测的方法很简单，只要让B端主动通过这个连接向A端继续发送数据即可。上文说过，A端异常断开后，和A端直接连接的路由器是知道的。当B端发送的数据经过转发后到达这个路由器后，必然最终会返回B端一个目的不可达。此时B端立刻就会知道这条连接其实已经异常断开了。
但是B端不可能知道什么时候会出现这种异常，所以B端必须定时发送数据来检测连接是否异常断开。数据的内容无关紧要，任何数据都能达到这个效果。这个数据就是我们经常在TCP编程中所说的心跳。

TCP协议本身就提供了一种这样的机制来探测对端的存活。TCP协议有一个KEEP_LIVE开关，只要打开这个开关就会定时发送一些数据长度为零的探测心跳包，发送的频率和次数都可以设置，具体的方法在网上搜索tcp keepalive即可，网上有很多文章，这里不再赘述。

除了使用TCP协议本身的保活开关机制，还可以在应用层主动发送心跳数据包，那么在应用层主动发送心跳数据包的方式和TCP协议本身的保活机制有什么区别呢？

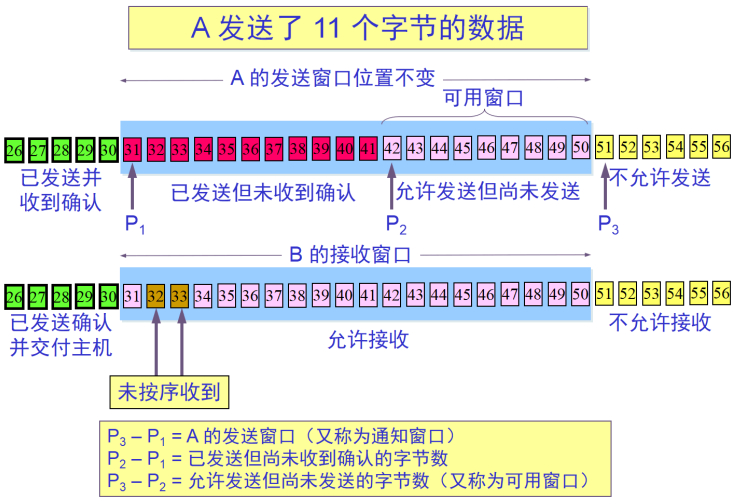
应用层的心跳数据包会耗费更多的带宽，因为TCP协议的保活机制发送的是数据长度为零心跳包，而应用层的心跳数据包长度则必然会大于0。
应用层的心跳数据包可以带一些应用所需要的数据，随应用自己控制，而TCP协议的保活机制则是对于应用层透明的，无法利用心跳携带数据。

如果只是一端向另一端发送心跳就行了呢？显然不行。因为两端都有可能发生异常断开的情况。所以TCP连接的两端必须都向对端发送心跳。
TCP中不使用心跳通常来说并没有什么问题，但是一旦遇到了连接异常断开，那么就会出现问题。所以任何一个完善的TCP应用都应该使用心跳。

5. TCP和UDP区别？如何改进TCP

- TCP和UDP区别
  - UDP 是无连接的，即发送数据之前不需要建立连接。
  - UDP 使用尽最大努力交付，即不保证可靠交付，同时也不使用拥塞控制。
  - UDP 是面向报文的。UDP 没有拥塞控制，很适合多媒体通信的要求。
  - UDP 支持一对一、一对多、多对一和多对多的交互通信。
  - UDP 的首部开销小，只有 8 个字节。
  - TCP 是面向连接的运输层协议。
  - 每一条 TCP 连接只能有两个端点(endpoint)，每一条 TCP 连接只能是点对点的（一对一）。
  - TCP 提供可靠交付的服务。
  - TCP 提供全双工通信。
  - TCP是面向字节流。
  - 首部最低20个字节。
- TCP加快传输效率的方法
  - 采取一块确认的机制

6. TCP滑动窗口



窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，**接收方通过 TCP 报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小**。

发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口左部的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到左部第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向右滑动接收窗口。

接收窗口只会对窗口内最后一个按序到达的字节进行确认，例如接收窗口已经收到的字节为 {31, 34, 35}，其中 {31} 按序到达，而 {32, 33} 就不是，因此只对字节 31 进行确认。发送方得到一个字节的确认之后，就知道这个字节之前的所有字节都已经被接收。

以下进行滑动窗口模拟

在 TCP 中，**滑动窗口是为了实现流量控制**。如果对方发送数据过快，接收方就来不及接收，接收方就需要通告对方，减慢数据的发送。

- **发送方接收到了对方发来的报文 ack = 33, win = 10，知道对方收到了 33 号前的数据**，现在期望接收 [33, 43) 号数据。发送方连续发送了 4 个报文段假设为 A, B, C, D, 分别携带 [33, 35), [35, 36), [36, 38), [38, 41) 号数据。
- 接收方接收到了报文段 A, C，但是没收到 B 和 D，也就是只收到了 [33, 35) 和 [36, 38) 号数据。接收方发送回对报文段 A 的确认：ack = 35, win = 10。
- 发送方收到了 ack = 35, win = 10，对方期望接收 [35, 45) 号数据。接着发送了一个报文段 E，它携带了 [41, 44) 号数据。
- 接收方接收到了报文段 B: [35, 36), D:[38, 41)，接收方发送对 D 的确认：ack = 41, win = 10.（这是一个累积确认）
- 发送方收到了 ack = 41, win = 10，对方期望接收 [41, 51) 号数据。

- ……

- 需要注意的是，接收方接收 tcp 报文的顺序是不确定的，并非是一定先收到 35 再收到 36，也可能是先收到 36，37，再收到 35.

7. TCP流量控制

流量控制是为了控制发送方发送速率，保证接收方来得及接收。

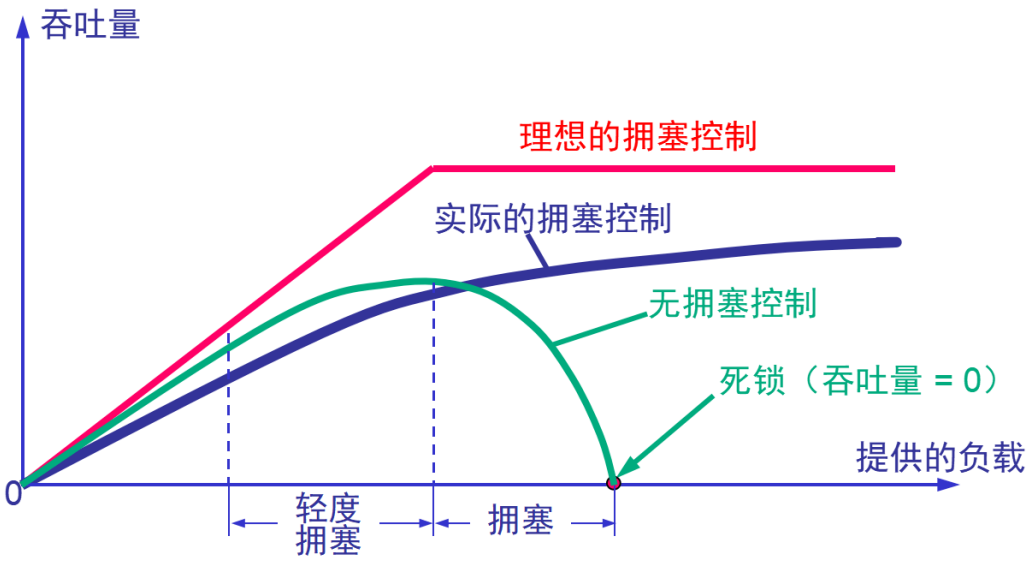
接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

8. TCP拥塞处理（Congestion Handling）

拥塞控制的一般原理

- 在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏——产生拥塞(congestion)。
- 出现资源拥塞的条件：对资源需求的总和 > 可用资源
- 若网络中有许多资源同时产生拥塞，网络的性能就要明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

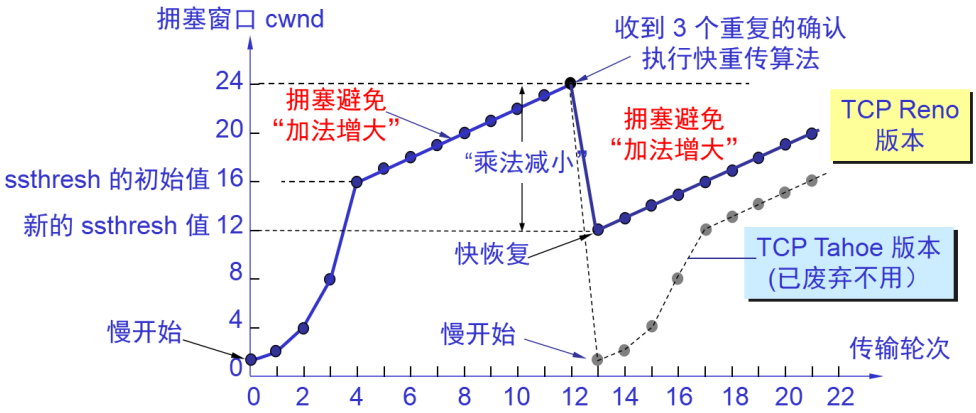


TCP 主要通过四种算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

- 接收方有足够大的接收缓存，因此不会发生流量控制；
- 虽然 TCP 的窗口基于字节，但是这里设窗口的大小单位为报文段。



（1）慢开始与拥塞避免

发送的最初执行慢开始，令 cwnd=1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能也就更高。设置一个慢启动阈值 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。

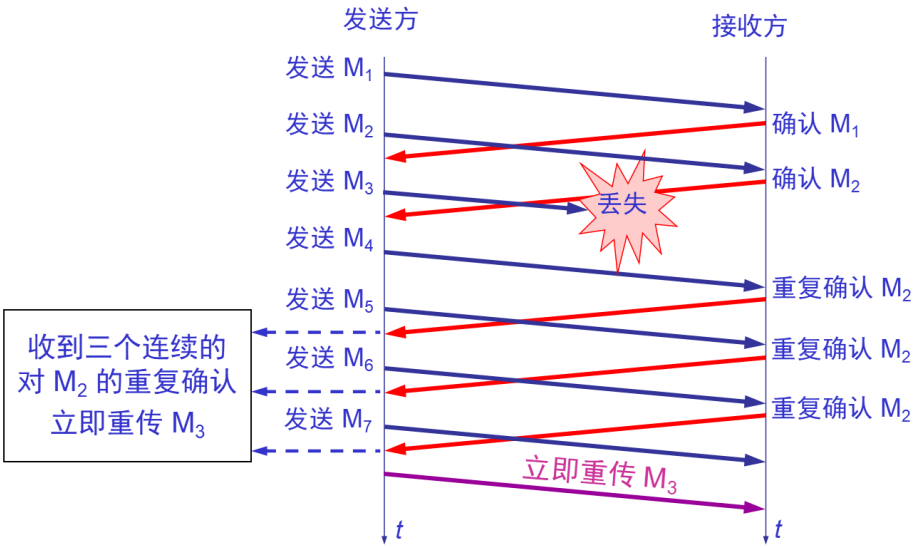
如果出现了超时，则令 ssthresh = cwnd/2，然后重新执行慢开始。

（2）快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M1 和 M2，此时收到 M4，应当发送对 M2 的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M2，则 M3 丢失，立即重传 M3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd/2 ，cwnd = ssthresh，注意到此时直接进入拥塞避免。慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。



（3）发送窗口的上限值

发送方的发送窗口的上限值应当取为接收方窗口 rwnd 和拥塞窗口 cwnd 这两个变量中较小的一个，即应按以下公式确定：

- 发送窗口的上限值 = Min {rwnd, cwnd}
  - 当 rwnd < cwnd 时，是接收方的接收能力限制发送窗口的最大值。
  - 当 cwnd < rwnd 时，则是网络的拥塞限制发送窗口的最大值。

9. 如何区分流量控制和拥塞控制

- 拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。
- 拥塞控制是一个全局性的过程，涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。
- 流量控制往往指在给定的发送端和接收端之间的点对点通信量的控制。
- 流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。
- 流量控制属于通信双方协商；拥塞控制涉及通信链路全局。
- 流量控制需要通信双方各维护一个发送窗、一个接收窗，对任意一方，接收窗大小由自身决定，发送窗大小由接收方响应的TCP报文段中窗口值确定；拥塞控制的拥塞窗口大小变化由试探性发送一定数据量数据探查网络状况后而自适应调整。
- 实际最终发送窗口 = min{流控发送窗口，拥塞窗口}。

10. 解释RTO，RTT和超时重传

- **超时重传**：发送端发送报文后若长时间未收到确认的报文则需要重发该报文。可能有以下几种情况：
  - 发送的数据没能到达接收端，所以对方没有响应。
  - 接收端接收到数据，但是ACK报文在返回过程中丢失。
  - 接收端拒绝或丢弃数据。
- **RTO**：从上一次发送数据，因为长期没有收到ACK响应，到下一次重发之间的时间。就是重传间隔。
  - 通常每次重传RTO是前一次重传间隔的两倍，计量单位通常是RTT。例：1RTT，2RTT，4RTT，8RTT……
  - 重传次数到达上限之后停止重传。
- **RTT**：数据从发送到接收到对方响应之间的时间间隔，即数据报在网络中一个往返用时。大小不稳定。

11. 停止等待和超时重传

(1). 停止等待

通常我们说如果A和B双方建立好tcp连接后就可以相互发送数据了，A为发送方，B为接收方。因为这里讨论可靠传输原理，所以把传输的数据单元称为分组。“停止等待”就是每发送完一个分组就停止发送，等待对方确认后再发送下一个分组。停止等待协议考虑了数据在网络中传输出现的几种情况来提供有效措施保障数据的可靠传输，下面我们就一一来介绍这几种情况。

- 出现差错或丢失的情况

当A在发送M1分组的过程中丢失时，又或者B接收到M1分组检测到差错并丢弃了M1分组时（注意：这里B不会发送M1确认分组，而是什么也不做），可靠传输协议是这样设计的：只要A没有在规定时间内收到B的确认，就认为刚才发送的分组丢失了，并对丢失的分组进行重传，这种方式叫超时重传。要实现超时重传，就要每发送完一个分组就设置一个超时计时器。如果在超时计时器到期之前收到了对方的确认，则撤销该超时计时器。

这里注意几点：

1. A发送完一个分组后，必须暂时存储已发送的分组的副本（发生超时重传时使用），当收到该分组的确认时就清除本地存储的分组的副本。

2. 分组和确认分组都必须进行编号，这样才能明确哪一个已发送的分组被确认，哪一个还没被确认。

3. 超时计时器设置的重传时间比数据分组传输的平均往返时间更长一些，在设置重传时间也是有要求的。因为重传时间设置的过长会导致重传花费的时间长，通信效率慢，但是重传时间设置的过短会导致出现不必要的重传，浪费网络资源。

比如B发送的确认分组发生在网络中发生拥塞，传输的时间过长才到达A，但是A设置的重传时间又很短，就会出现不必要的重传。因此设置重传时间是非常复杂的，因为分组在网络传输的过程中经过哪些网络，是否会出现网络拥塞或其他问题，都是不确定的。

- 确认丢失的情况

当B收到A的M1分组后，B发送的M1确认分组在网络中丢失了，且A在设定的超时重传时间内又没有收到B的M1确认分组，这时A无法知道是自己发送的M1分组出错丢失，还是B发送的M1确认分组丢失了，那么A会在超时计时器到期后重传M1分组。

如果B又收到了重传的分组M1，这时B会丢弃重复的M1分组，并向A发送M1确认分组（很明显，因为A本来就没有收到过确认啊）。

- 确认迟到的情况

这里我们需要假设这么一种情况：A在发送M1分组后，B发送的确认M1分组却迟到了，但是A在超时计时器规定的时间内又没有收到B的确认M1分组，那么A将会重传M1分组，根据前面所知的情况来看，B在收到重复的M1分组后会丢弃并重传确认M1分组。那么在A收到重传的确认分组后，又收到了B迟到的确认M1分组，这是A会丢弃迟到的确认M1分组。

像上面所说的可靠传输协议通常称为自动重传请求，也就是说，重传时自动进行的，只要发送方没收到确认，就会重传。如果A不断重传分组却总是也收不到确认，这说明通信线路太差，不能进行通信。这是上面所有停止等待的情况.

(2). 超时重传

原理是在发送某一个数据以后就开启一个计时器，在一定时间内如果没有得到发送的数据报的ACK报文，那么就重新发送数据，直到发送成功为止。
影响超时重传机制协议效率的一个关键参数是重传超时时间（RTO，Retransmission TimeOut）。

RTO的值被设置过大过小都会对协议造成不利影响。

- RTO设长了，重发就慢，没有效率，性能差。
- RTO设短了，重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

连接往返时间（RTT，Round Trip Time），指发送端从发送TCP包开始到接收它的立即响应所消耗的时间。

12. 从输入网址到获得页面的网络请求过程

- 查询 DNS
    - 浏览器搜索自身的DNS缓存
    - 搜索操作系统的DNS缓存，本地host文件查询
    - 如果 DNS 服务器和我们的主机在同一个子网内，系统会按照下面的 ARP 过程对 DNS 服务器进行 ARP查询
    - 如果 DNS 服务器和我们的主机在不同的子网，系统会按照下面的 ARP 过程对默认网关进行查询
- 浏览器获得域名对应的IP地址后，发起HTTP三次握手
- TCP/IP连接建立起来后，浏览器就可以向服务器发送HTTP请求了
- TLS 握手

    - 客户端发送一个 `ClientHello` 消息到服务器端，消息中同时包含了它的 Transport Layer Security (TLS) 版本，可用的加密算法和压缩算法.
    - 服务器端向客户端返回一个 `ServerHello` 消息，消息中包含了服务器端的TLS版本，服务器所选择的加密和压缩算法，以及数字证书认证机构.（Certificate Authority，缩写 CA）签发的服务器公开证书，证书中包含了公钥。客户端会使用这个公钥加密接下来的握手过程，直到协商生成一个新的对称密钥.
    - 客户端根据自己的信任CA列表，验证服务器端的证书是否可信。如果认为可信，客户端会生成一串伪随机数，使用服务器的公钥加密它。这串随机数会被用于生成新的对称密钥.
    - 服务器端使用自己的私钥解密上面提到的随机数，然后使用这串随机数生成自己的对称主密钥.
    - 客户端发送一个 `Finished` 消息给服务器端，使用对称密钥加密这次通讯的一个散列值.
    - 服务器端生成自己的 hash 值，然后解密客户端发送来的信息，检查这两个值是否对应。如果对应，就向客户端发送一个 `Finished` 消息，也使用协商好的对称密钥加密.
    - 从现在开始，接下来整个 TLS 会话都使用对称秘钥进行加密，传输应用层（HTTP）内容.

- HTTP 服务器请求处理.

  HTTPD(HTTP Daemon)在服务器端处理请求/响应。最常见的 HTTPD 有 Linux 上常用的 Apache 和 nginx，以及 Windows 上的 IIS。

    - HTTPD 接收请求
    -
        - 服务器把请求拆分为以下几个参数：

          HTTP 请求方法(`GET` , `POST` , `HEAD` , `PUT` , `DELETE` , `CONNECT` , `OPTIONS` , 或者 `TRACE`)。直接在地址栏中输入 URL 这种情况下，使用的是 GET 方法域名：google.com请求路径/页面：/ (我们没有请求google.com下的指定的页面，因此 / 是默认的路径)

    - 服务器验证其上已经配置了 google.com 的虚拟主机
    - 服务器验证 google.com 接受 GET 方法
    - 服务器验证该用户可以使用 GET 方法(根据 IP 地址，身份信息等)
    - 如果服务器安装了 URL 重写模块（例如 Apache 的 mod_rewrite 和 IIS 的 URL Rewrite），服务器会尝试匹配重写规则，如果匹配上的话，服务器会按照规则重写这个请求
    - 服务器根据请求信息获取相应的响应内容，这种情况下由于访问路径是 “/“ ,会访问首页文件（你可以重写这个规则，但是这个是最常用的）。
    - 服务器会使用指定的处理程序分析处理这个文件，假如 Google 使用 PHP，服务器会使用 PHP 解析 index 文件，并捕获输出，把 PHP 的输出结果返回给请求者

- 服务器接受到这个请求，根据路径参数，经过后端的一些处理生成HTML页面代码返回给浏览器
- 浏览器拿到完整的HTML页面代码开始解析和渲染，如果遇到引用的外部js，CSS,图片等静态资源，它们同样也是一个个的HTTP请求，都需要经过上面的步骤

- 浏览器根据拿到的资源对页面进行渲染，最终把一个完整的页面呈现给用户

超详细版本请转向阅读：what-happens-when-zh_CN

第二部分：应用层（HTTP）

1. URL、URI、URN区别

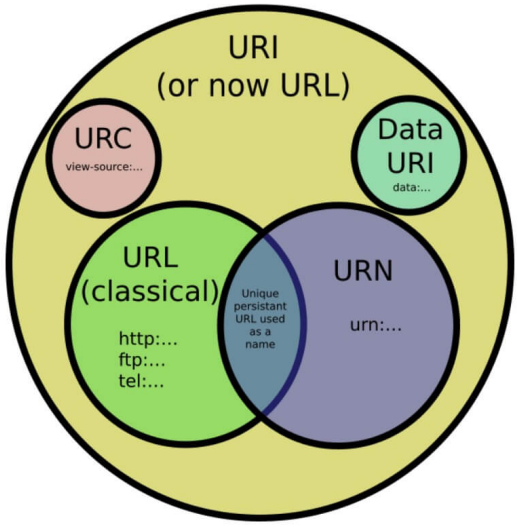
- URI（Uniform Resource Identifier，统一资源标识符）

  web服务器资源的名字，例如： index.html

- URL（Uniform Resource Locator，统一资源定位符）

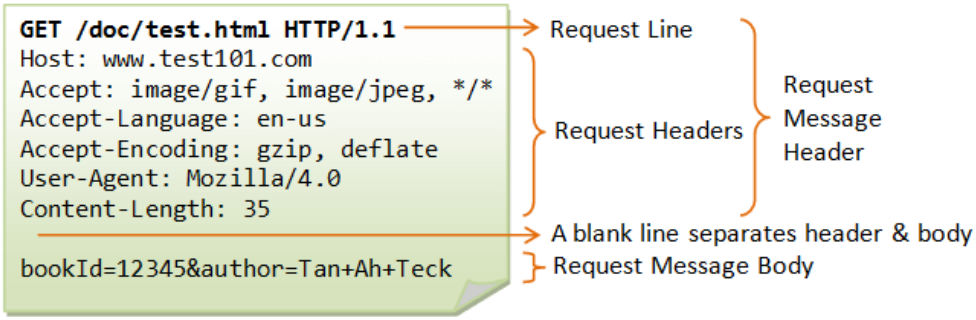
- URN（Uniform Resource Name，统一资源名称），例如 urn:isbn:0-486-27557-4。

URI 包含 URL 和 URN，目前 WEB 只有 URL 比较流行，所以见到的基本都是 URL。

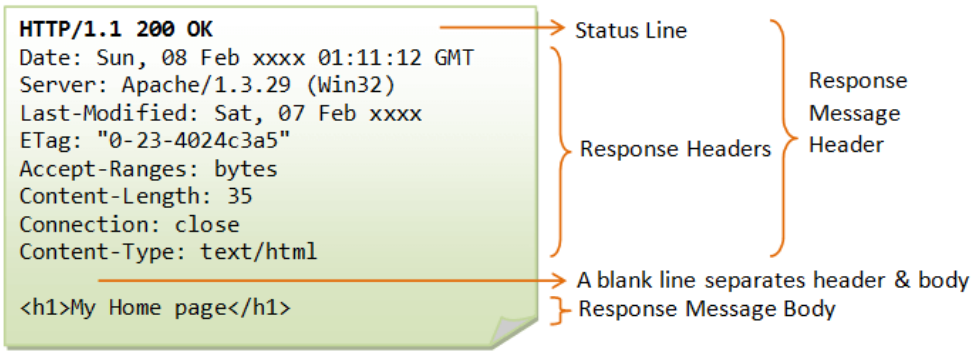


2. HTTP的请求和响应报文

（1）请求报文



（2）响应报文



3. HTTP状态

服务器返回的 **响应报文** 中第一行为状态行，包含了状态码以及原因短语，用来告知客户端请求的结果。

状态码	类别	原因短语
1XX	Informational（信息性状态码）	接收的请求正在处理
2XX	Success（成功状态码）	请求正常处理完毕
3XX	Redirection（重定向状态码）	需要进行附加操作以完成请求
4XX	Client Error（客户端错误状态码）	服务器无法处理请求
5XX	Server Error（服务器错误状态码）	服务器处理请求出错

（1）1XX 信息

- **100 Continue** ：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。

（2）2XX 成功

- **200 OK**
- **204 No Content** ：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
- **206 Partial Content** ：表示客户端进行了范围请求。响应报文包含由 Content-Range 指定范围的实体内容。

（3）3XX 重定向

- **301 Moved Permanently** ：永久性重定向
- **302 Found** ：临时性重定向
- **303 See Other** ：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。
- 注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会在 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。
- **304 Not Modified** ：如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-None-Match，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。

- **307 Temporary Redirect** ：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的 POST 方法改成 GET 方法。

（4）4XX 客户端错误

- **400 Bad Request** ：请求报文中存在语法错误。
- **401 Unauthorized** ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
- **403 Forbidden** ：请求被拒绝，服务器端没有必要给出拒绝的详细理由。
- **404 Not Found**

（5）5XX 服务器错误

- **500 Internal Server Error** ：服务器正在执行请求时发生错误。
- **502 Bad Gateway** : 是用来表示代理或网关在处理请求时发生了错误，并不一定是原始服务器出现了问题
- **503 Service Unavailable** ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。
- **504 Gateway Timeout**：作为网关或者代理工作的服务器尝试执行请求时，未能及时从上游服务器（URI标识出的服务器，例如HTTP、FTP、LDAP）或者辅助服务器（例如DNS）收到响应。

注意：某些代理服务器在DNS查询超时时会返回400或者500错误。

4. HTTP方法

客户端发送的 **请求报文** 第一行为请求行，包含了方法字段。

（1）GET

> 获取资源

当前网络请求中，绝大部分使用的是 GET 方法。

（2）HEAD

> 获取报文首部

和 GET 方法一样，但是不返回报文实体主体部分。

主要用于确认 URL 的有效性以及资源更新的日期时间等。

（3）POST

> 传输实体主体

POST 主要用来传输数据，而 GET 主要用来获取资源。

更多 POST 与 GET 的比较请见第八章。

（4）PUT

> 上传文件

由于自身不带验证机制，任何人都可以上传文件，因此存在安全性问题，一般不使用该方法。

```
PUT /new.html HTTP/1.1
Host: example.com
Content-type: text/html
Content-length: 16

<p>New File</p>
```

（5）PATCH

> 对资源进行部分修改

PUT 也可以用于修改资源，但是只能完全替代原始资源，PATCH 允许部分修改。

```
PATCH /file.txt HTTP/1.1
Host: www.example.com
Content-Type: application/example
If-Match: "e0023aa4e"
Content-Length: 100

[description of changes]
```

（6）DELETE

> 删除文件

与 PUT 功能相反，并且同样不带验证机制。

```
DELETE /file.html HTTP/1.1
```

（7）OPTIONS

> 查询支持的方法

查询指定的 URL 能够支持的方法。

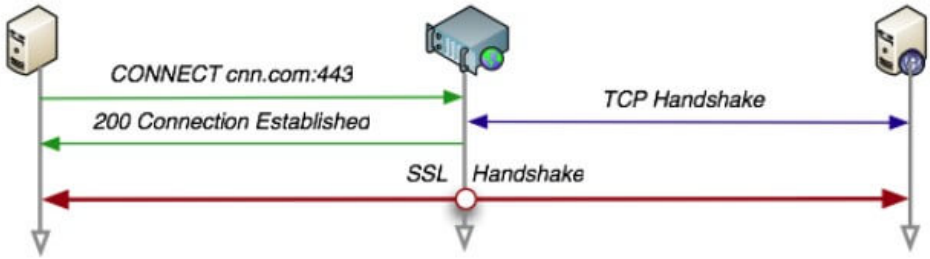
会返回 Allow: GET, POST, HEAD, OPTIONS 这样的内容。

（8）CONNECT

> 要求在与代理服务器通信时建立隧道

使用 SSL（Secure Sockets Layer，安全套接层）和 TLS（Transport Layer Security，传输层安全）协议把通信内容加密后经网络隧道传输。

```
CONNECT www.example.com:443 HTTP/1.1
```



（9）TRACE

> 追踪路径

服务器会将通信路径返回给客户端。

发送请求时，在 Max-Forwards 首部字段中填入数值，每经过一个服务器就会减 1，当数值为 0 时就停止传输。

通常不会使用 TRACE，并且它容易受到 XST 攻击（Cross-Site Tracing，跨站追踪）。

5. GET和POST的区别？

- GET 被强制服务器支持
- 浏览器对URL的长度有限制，所以GET请求不能代替POST请求发送大量数据
- GET请求发送数据更小
- GET请求是不安全的
- GET请求是幂等的
  - 幂等的意味着对同一URL的多个请求应该返回同样的结果
- POST请求不能被缓存
- POST请求相对GET请求是「安全」的
  - 这里安全的含义仅仅是指是非修改信息
- GET用于信息获取，而且是安全的和幂等的.
  - 所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。
- POST是用于修改服务器上的资源的请求
- 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

引申：说完原理性的问题，我们从表面上来看看GET和POST的区别：

- GET是从服务器上获取数据，POST是向服务器传送数据。 GET和 POST只是一种传递数据的方式，GET也可以把数据传到服务器，他们的本质都是发送请求和接收结果。只是组织格式和数据量上面有差别，http协议里面有介绍
- GET是把参数数据队列加到提交表单的ACTION属性所指的URL中，值和表单内各个字段一一对应，在URL中可以看到。POST是通过HTTP POST机制，将表单内各个字段与其内容放置在HTML HEADER内一起传送到ACTION属性所指的URL地址。用户看不到这个过程。 因为GET设计成传输小数据，而且最好是不修改服务器的数据，所以浏览器一般都在地址栏里面可以看到，但POST一般都用来传递大数据，或比较隐私的数据，所以在地址栏看不到，能不能看到不是协议规定，是浏览器规定的。
- 对于GET方式，服务器端用Request.QueryString获取变量的值，对于POST方式，服务器端用Request.Form获取提交的数据。 没明白，怎么获得变量和你的服务器有关，和GET或POST无关，服务器都对这些请求做了封装

- GET传送的数据量较小，不能大于2KB。POST传送的数据量较大，一般被默认为不受限制。但理论上，IIS4中最大量为80KB，IIS5中为100KB。 POST基本没有限制，我想大家都上传过文件，都是用POST方式的。只不过要修改form里面的那个type参数
- GET安全性非常低，POST安全性较高。 如果没有加密，他们安全级别都是一样的，随便一个监听器都可以把所有的数据监听到。

6. 如何理解HTTP协议是无状态的

HTTP协议是无状态的（stateless），指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和上一次打开这个服务器上的网页之间没有任何联系。
HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。

缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

7. 什么是短连接和长连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

8. Cookie

HTTP 协议是无状态的，主要是为了让 HTTP 协议尽可能简单，使得它能够处理大量事务。HTTP/1.1 引入 Cookie 来保存状态信息。

Cookie 是服务器发送到用户浏览器并保存在本地的一小块数据，它会在浏览器下次向同一服务器再发起请求时被携带并发送到服务器上。它用于告知服务端两个请求是否来自同一浏览器，并保持用户的登录状态。

（1）用途

- 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
- 个性化设置（如用户自定义设置、主题等）
- 浏览器行为跟踪（如跟踪分析用户行为等）

Cookie 曾一度用于客户端数据的存储，因为当时并没有其它合适的存储办法而作为唯一的存储手段，但现在随着现代浏览器开始支持各种各样的存储方式，Cookie 渐渐被淘汰。由于服务器指定 Cookie 后，浏览器的每次请求都会携带 Cookie 数据，会带来额外的性能开销（尤其是在移动环境下）。新的浏览器 API 已经允许开发者直接将数据存储到本地，如使用 Web storage API （本地存储和会话存储）或 IndexedDB。

（2）创建过程

服务器发送的响应报文包含Set-Cookie 首部字段，客户端得到响应报文后把 Cookie 内容保存到浏览器中。

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry

[page content]
```

客户端之后对同一个服务器发送请求时，会从浏览器中读出 Cookie 信息通过 Cookie 请求首部字段发送给服务器。

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

（3）分类

- 会话期 Cookie：浏览器关闭之后它会被自动删除，也就是说它仅在会话期内有效。
- 持久性 Cookie：指定一个特定的过期时间（Expires）或有效期（max-age）之后就成为了持久性的 Cookie。

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
```

（4）JavaScript 获取 Cookie

通过 `Document.cookie` 属性可创建新的 Cookie，也可通过该属性访问非 HttpOnly 标记的 Cookie。

```
document.cookie = "yummy_cookie=choco";
document.cookie = "tasty_cookie=strawberry";
console.log(document.cookie);
```

（5）Secure 和 HttpOnly

标记为 Secure 的 Cookie 只应通过被 HTTPS 协议加密过的请求发送给服务端。但即便设置了 Secure 标记，敏感信息也不应该通过 Cookie 传输，因为 Cookie 有其固有的不安全性，Secure 标记也无法提供确实的安全保障。

标记为 HttpOnly 的 Cookie 不能被 JavaScript 脚本调用。因为跨域脚本 (XSS) 攻击常常使用 JavaScript 的 `Document.cookie` API 窃取用户的 Cookie 信息，因此使用 HttpOnly 标记可以在一定程度上避免 XSS 攻击。

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

（6）作用域

Domain 标识指定了哪些主机可以接受 Cookie。如果不指定，默认为当前文档的主机（不包含子域名）。如果指定了 Domain，则一般包含子域名。例如，如果设置 Domain=mozilla.org，则 Cookie 也包含在子域名中（如 developer.mozilla.org）。

Path 标识指定了主机下的哪些路径可以接受 Cookie（该 URL 路径必须存在于请求 URL 中）。以字符 %x2F ("/") 作为路径分隔符，子路径也会被匹配。例如，设置 Path=/docs，则以下地址都会匹配：

- /docs
- /docs/Web/
- /docs/Web/HTTP

9. Session

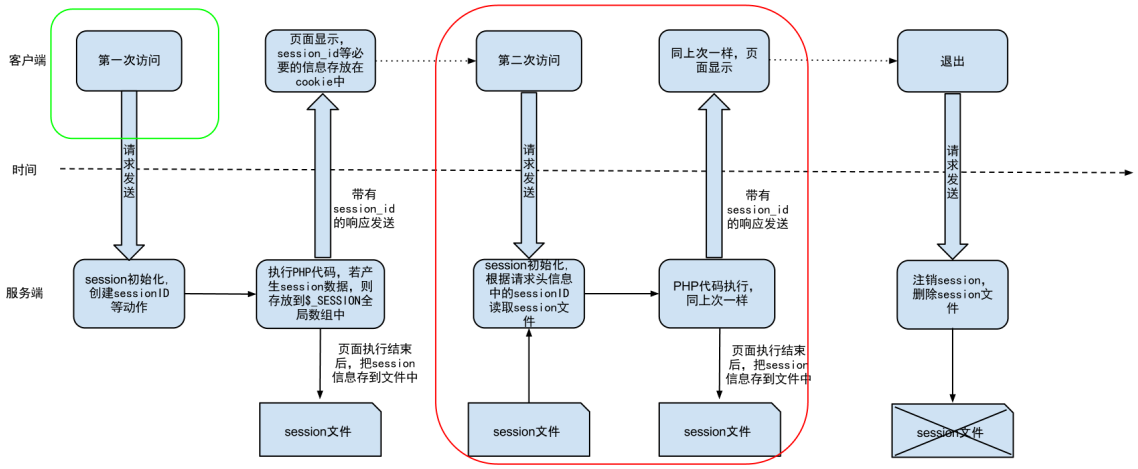
除了可以将用户信息通过 Cookie 存储在用户浏览器中，也可以利用 Session 存储在服务器端，存储在服务器端的信息更加安全。

Session 可以存储在服务器上的文件、数据库或者内存中，现在最常见的是将 Session 存储在内存型数据库中，比如 Redis。

使用 Session 维护用户登录的过程如下：

- 用户进行登录时，用户提交包含用户名和密码的表单，放入 HTTP 请求报文中；
- 服务器验证该用户名和密码；
- 如果正确则把用户信息存储到 Redis 中，它在 Redis 中的 ID 称为 Session ID；
- 服务器返回的响应报文的 Set-Cookie 首部字段包含了这个 Session ID，客户端收到响应报文之后将该 Cookie 值存入浏览器中；
- 客户端之后对同一个服务器进行请求时会包含该 Cookie 值，服务器收到之后提取出 Session ID，从 Redis 中取出用户信息，继续之后的业务操作。

应该注意 Session ID 的安全性问题，不能让它被恶意攻击者轻易获取，那么就不能产生一个容易被猜到的 Session ID 值。此外，还需要经常重新生成 Session ID。在对安全性要求极高的场景下，例如转账等操作，除了使用 Session 管理用户状态之外，还需要对用户进行重新验证，比如重新输入密码，或者使用短信验证码等方式。



10. 浏览器禁用 Cookie

此时无法使用 Cookie 来保存用户信息，只能使用 Session。除此之外，不能再将 Session ID 存放到 Cookie 中，而是使用 URL 重写技术，将 Session ID 作为 URL 的参数进行传递。

11. Cookie 与 Session 选择

- Cookie 只能存储 ASCII 码字符串，而 Session 则可以存取任何类型的数据，因此在考虑数据复杂性时首选 Session；
- Cookie 存储在浏览器中，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；
- 对于大型网站，如果用户所有的信息都存储在 Session 中，那么开销是非常大的，因此不建议将所有的用户信息都存储到 Session 中。

12. HTTPs安全性

HTTP 有以下安全性问题：

- 使用明文进行通信，内容可能会被窃听；

- 不验证通信方的身份，通信方的身份有可能遭遇伪装；
- 无法证明报文的完整性，报文有可能遭篡改。

HTTPs（Hyper Text Transfer Protocol over Secure Socket Layer），是以安全为目标的HTTP通道，简单讲HTTPs是HTTP的安全版。

HTTPs 并不是新协议，而是让 HTTP 先和 SSL（Secure Sockets Layer）通信，再由 SSL 和 TCP 通信。也就是说 HTTPs 使用了隧道进行通信。

通过使用 SSL，HTTPs 具有了加密（防窃听）、认证（防伪装）和完整性保护（防篡改）。

（1）对称密钥加密

对称密钥加密（Symmetric-Key Encryption），加密和解密使用同一密钥。

- 优点：运算速度快；
- 缺点：无法安全地将密钥传输给通信方。

（2）非对称密钥加密

非对称密钥加密，又称公开密钥加密（Public-Key Encryption），加密和解密使用不同的密钥。

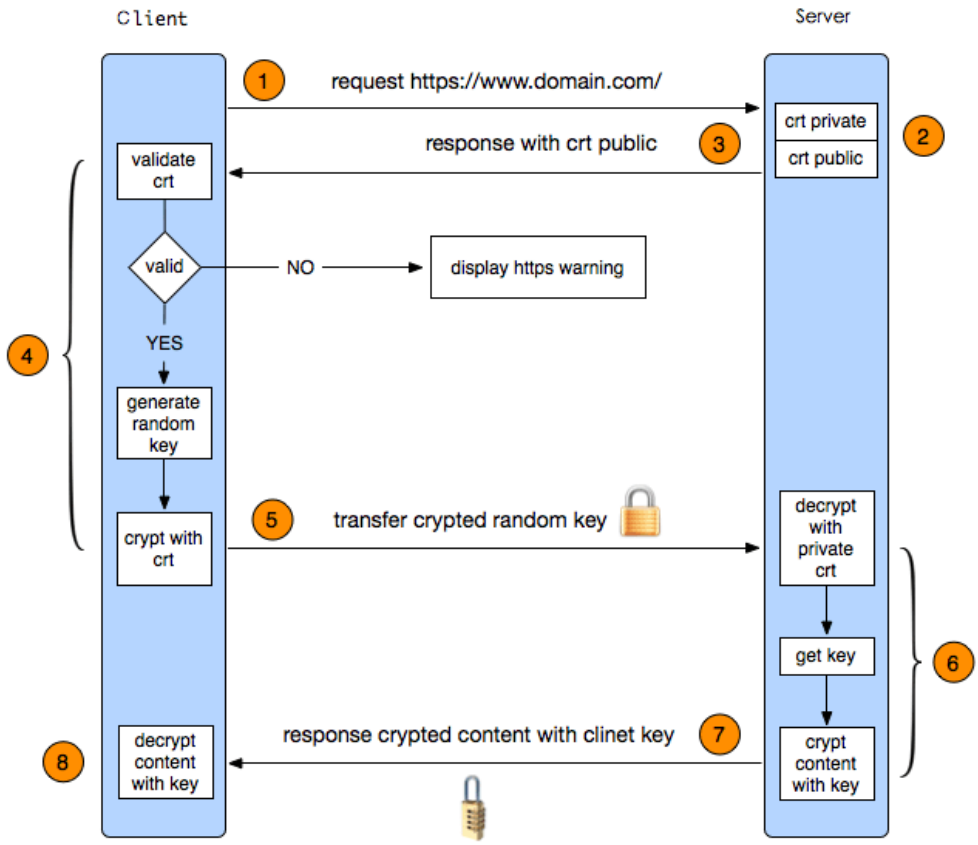
公开密钥所有人都可以获得，通信发送方获得接收方的公开密钥之后，就可以使用公开密钥进行加密，接收方收到通信内容后使用私有密钥解密。

非对称密钥除了用来加密，还可以用来进行签名。因为私有密钥无法被其他人获取，因此通信发送方使用其私有密钥进行签名，通信接收方使用发送方的公开密钥对签名进行解密，就能判断这个签名是否正确。

- 优点：可以更安全地将公开密钥传输给通信发送方；
- 缺点：运算速度慢。

（3）HTTPs 采用的加密方式

HTTPs 采用混合的加密机制，使用非对称密钥加密用于传输对称密钥来保证安全性，之后使用对称密钥加密进行通信来保证效率。



13. SSL/TLS协议的握手过程

我们知道，HTTP 协议都是明文传输内容，在早期只展示静态内容时没有问题。伴随着互联网的快速发展，人们对于网络传输安全性的要求也越来越高，HTTPS 协议因此出现。如上图所示，在 HTTPS 加密中真正起作用的其实是 SSL/TLS 协议。SSL/TLS 协议作用在 HTTP 协议之下，对于上层应用来说，原来的发送接收数据流程不变，这就很好地兼容了老的 HTTP 协议，这也是软件开发中分层实现的体现。

SSL (Secure Socket Layer，安全套接字层)

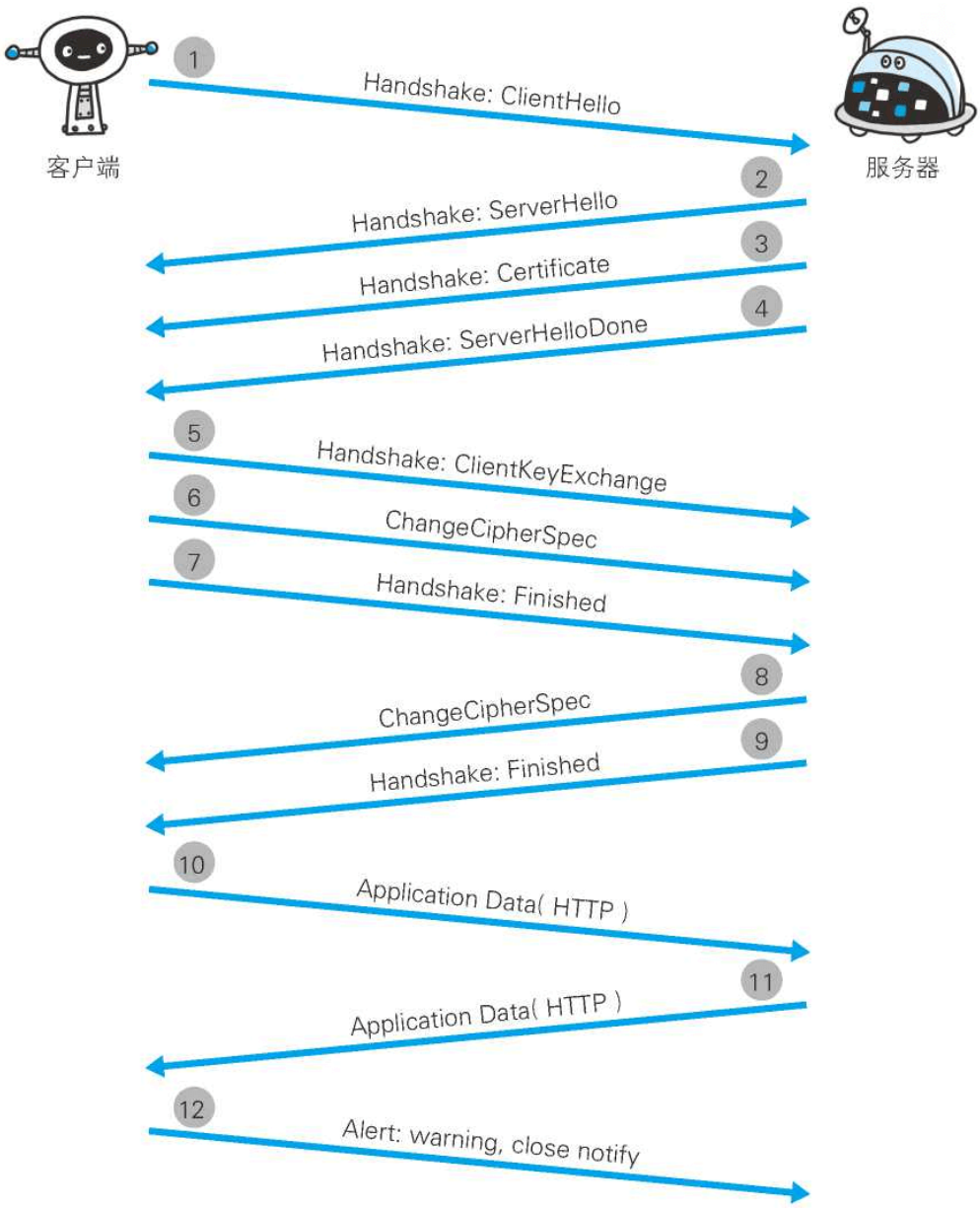
SSL为Netscape所研发，用以保障在Internet上数据传输之安全，利用数据加密(Encryption)技术，可确保数据在网络上之传输过程中不会被截取，当前为3.0版本。

SSL协议可分为两层： SSL记录协议（SSL Record Protocol）：它建立在可靠的传输协议（如TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。 SSL握手协议（SSL Handshake Protocol）：它建立在SSL记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等。

TLS (Transport Layer Security，传输层安全协议)

用于两个应用程序之间提供保密性和数据完整性。
TLS 1.0是IETF（Internet Engineering Task Force，Internet工程任务组）制定的一种新的协议，它建立在SSL 3.0协议规范之上，是SSL 3.0的后续版本，可以理解为SSL 3.1，它是写入了 RFC 的。该协议由两层组成： TLS 记录协议（TLS Record）和 TLS 握手协议（TLS Handshake）。较低的层为 TLS 记录协议，位于某个可靠的传输协议（例如 TCP）上面。

SSL/TLS 握手是为了**安全**地协商出一份**对称加密**的秘钥，这个过程很有意思，下面我们一起来了解一下。



（1）client hello

握手第一步是客户端向服务端发送 Client Hello 消息，这个消息里包含了一个客户端生成的随机数 **Random1**、客户端支持的**加密套件**（Support Ciphers）和 SSL Version 等信息。

（2）server hello

第二步是服务端向客户端发送 Server Hello 消息，这个消息会从 Client Hello 传过来的 Support Ciphers 里确定一份加密套件，这个套件决定了后续加密和生成摘要时具体使用哪些算法，另外还会生成一份随机数 **Random2**。注意，至此客户端和服务端都拥有了两个随机数（Random1+ Random2），这两个随机数会在后续生成对称秘钥时用到。

（3）server certificate

这一步是服务端将自己的证书下发给客户端，让客户端验证自己的身份，客户端验证通过后取出证书中的公钥。

（4）Server Hello Done

Server Hello Done 通知客户端 Server Hello 过程结束。

（5）Client Key Exchange

上面客户端根据服务器传来的公钥生成了 **PreMaster Key**，Client Key Exchange 就是将这个 key 传给服务端，服务端再用自己的私钥解出这个 **PreMaster Key** 得到客户端生成的 **Random3**。至此，客户端和服务端都拥有 **Random1 + Random2 + Random3**，两边再根据同样的算法就可以生成一份秘钥，握手结束后的应用层数据都是使用这个秘钥进行对称加密。

为什么要使用三个随机数呢？这是因为 SSL/TLS 握手过程的数据都是明文传输的，并且多个随机数种子来生成秘钥不容易被暴力破解出来。

（6）Change Cipher Spec(Client)

这一步是客户端通知服务端后面再发送的消息都会使用前面协商出来的秘钥加密了，是一条事件消息。

（7）Finished(Client)

客户端发送Finished报文。该报文包含连接至今全部报文的整理校验值。这次握手协议是否能成功，要以服务器是否能够正确解密该报文作为判定标准。

（8）Change Cipher Spec(Server)

服务器同样发送Change Cipher Spec报文给客户端

（9）Finished(Server)

服务器同样发送Finished报文给客户端

（10-11）Application Data

到这里，双方已安全地协商出了同一份秘钥，所有的应用层数据都会用这个秘钥加密后再通过 TCP 进行可靠传输。

（12）Alert：warning, close notify

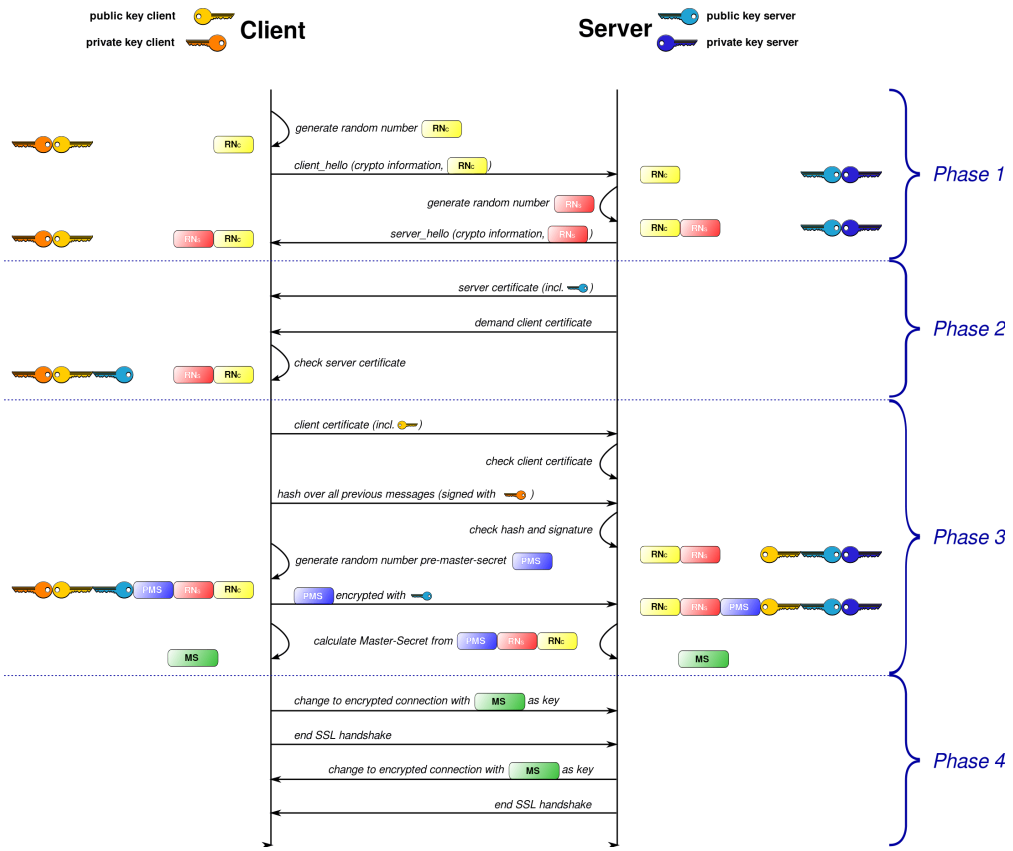
最后由客户端断开连接。断开连接时，发送close_notify报文。上图做了一些省略，在这步之后再发送一种叫做MAC（Message Authentication Code）的报文摘要。MAC能够查知报文是否遭到篡改，从而保护报文的完整性。

（*）demand client certificate

Certificate Request 是服务端要求客户端上报证书，这一步是可选的，对于安全性要求高的场景会用到。

（*）check server certificate

客户端收到服务端传来的证书后，先从 CA 验证该证书的合法性，验证通过后取出证书中的服务端公钥，再生成一个随机数 **Random3**，再用服务端公钥非对称加密 **Random3** 生成 **PreMaster Key**。



14. 数字签名、数字证书、SSL、https是什么关系？

HTTPS 是建立在密码学基础之上的一种安全通信协议，严格来说是基于 HTTP 协议和 SSL/TLS 的组合。理解 HTTPS 之前有必要弄清楚一些密码学的相关基础概念，
比如：明文、密文、密码、密钥、对称加密、非对称加密、信息摘要、数字签名、数字证书。接下来我会逐个解释这些术语，文章里面提到的『数据』、『消息』都是同一个概念，表示用户之间通信的内容载体，此外文章中提到了以下几个角色：

- Alice：消息发送者
- Bob：消息接收者
- Attacker：中间攻击者
- Trent：第三方认证机构

密码

密码学中的“密码”术语与网站登录时用的密码（password）是不一样的概念，password 翻译过来其实是“口令”，它是用于认证用途的一组文本字符串。

而密码学中的密码（cipher）是一套算法(algorithm)，这套算法用于对消息进行加密和解密，从明文到密文的过程称之为加密，密文反过来生成明文称之为解密，加密算法与解密算法合在一起称为密码算法。

密钥

密钥（key）是在使用密码算法过程中输入的一段参数。同一个明文在相同的密码算法和不同的密钥计算下会产生不同的密文。很多知名的密码算法都是公开的，密钥才是决定密文是否安全的重要参数，通常密钥越长，破解的难度越大，比如一个8位的密钥最多有256种情况，使用穷举法，能非常轻易的破解。根据密钥的使用方法，密码可分为对称加密和公钥加密。

对称加密

对称密钥（Symmetric-key algorithm）又称为共享密钥加密，加密和解密使用相同的密钥。常见的对称加密算法有DES、3DES、AES、RC5、RC6。对称密钥的优点是计算速度快，但是它有缺点，接收者需要发送者告知密钥才能解密，因此密钥如

何安全的发送给接收者成为了一个问题。

Alice 给 Bob 发送数据时，把数据用对称加密后发送给 Bob，发送过程中由于对数据进行了加密，因此即使有人窃取了数据也没法破解，因为它不知道密钥是什么。但是同样的问题是 Bob 收到数据后也一筹莫展，因为它也不知道密钥是什么，那么 Alice 是不是可以把数据和密钥一同发给 Bob 呢。当然不行，一旦把密钥和密钥一起发送的话，那就跟发送明文没什么区别了，因为一旦有人把密钥和数据同时获取了，密文就破解了。所以对称加密的密钥配是个问题。如何解决呢，公钥加密是一个办法。

公钥加密（非对称加密）

公开密钥加密（public-key cryptography）简称公钥加密，这套密码算法包含配对的密钥对，分为加密密钥和解密密钥。发送者用加密密钥进行加密，接收者用解密密钥进行解密。加密密钥是公开的，任何人都可以获取，因此加密密钥又称为公钥（public key），解密密钥不能公开，只能自己使用，因此它又称为私钥（private key）。常见的公钥加密算法有 RSA。

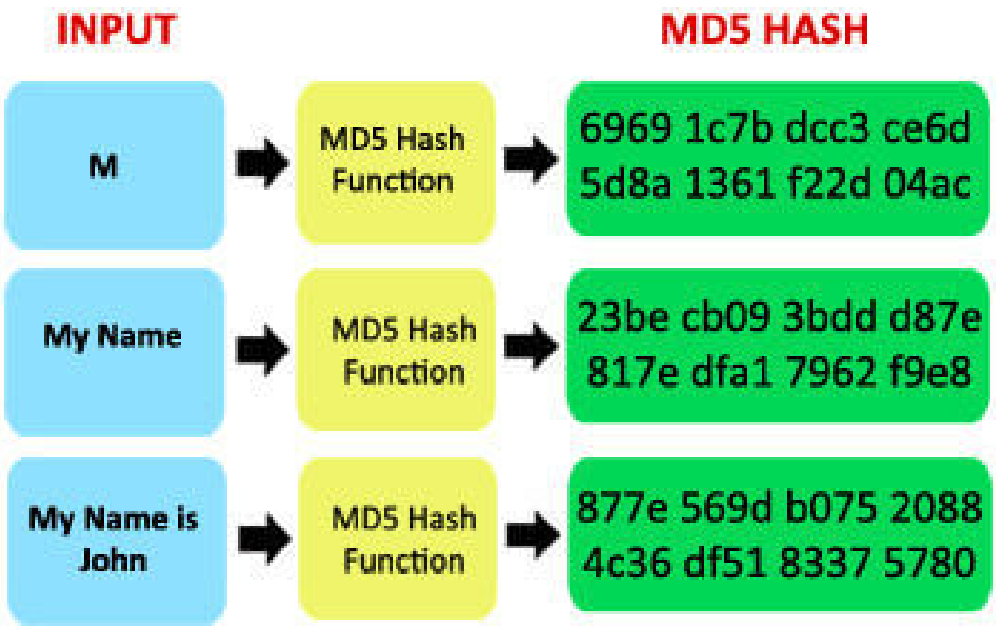
还是以Alice 给 Bob 发送数据为例，公钥加密算法由接收者 Bob 发起

1. Bob 生成公钥和私钥对，私钥自己保存，不能透露给任何人。
2. Bob 把公钥发送给 Alice，发送过程中即使被人窃取也没关系
3. Alice 用公钥把数据进行加密，并发送给 Bob，发送过程中被人窃取了同样没关系，因为没有配对的私钥进行解密是没法破解的
4. Bob 用配对的私钥解密。

虽然公钥加密解决了密钥配送的问题，但是你没法确认公钥是不是合法的，Bob 发送的公钥你不能肯定真的是 Bob 发的，因为也有可能在 Bob 把公钥发送给 Alice 的过程中出现中间人攻击，把真实的公钥掉包替换。
而对于 Alice 来说完全不知。还有一个缺点是它的运行速度比对称加密慢很多。

消息摘要

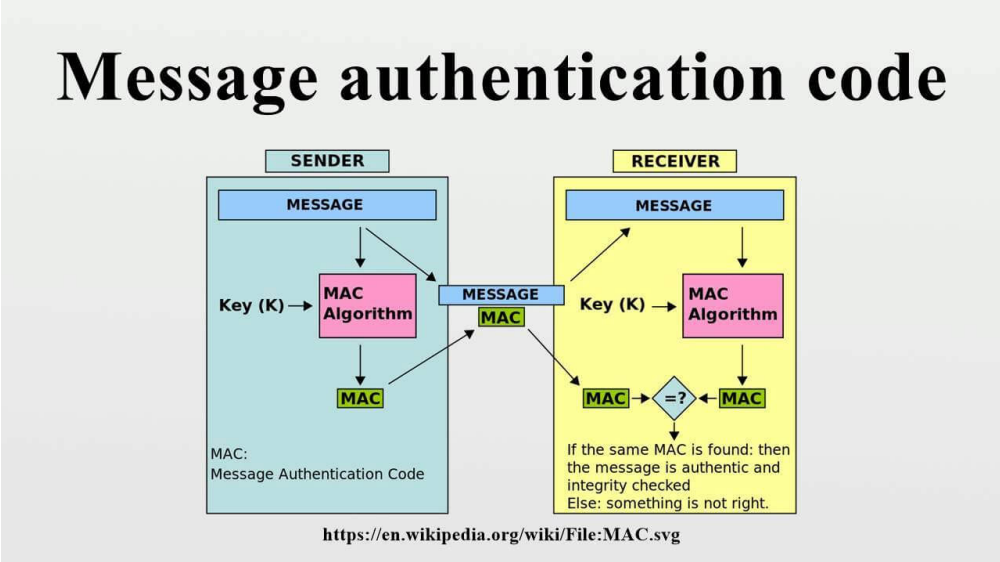
消息摘要（message digest）函数是一种用于判断数据完整性的算法，也称为散列函数或哈希函数，函数返回的值叫散列值，散列值又称为消息摘要或者指纹（fingerprint）。这种算法是一个不可逆的算法，因此你没法通过消息摘要反向推倒出消息是什么。所以它也称为**单向散列函数**。下载软件时如何确定是官方提供的完整版呢，如果有中间人在软件里面嵌入了病毒，你也不得而知。所以我们可以使用散列函数对消息进行运算，生成散列值，通常软件提供方会同时提供软件的下载地址和软件的散列值，用户把软件下载后在本地用相同的散列算法计算出散列值，与官方提供的散列值对比，如果相同，说明该软件是完成的，否则就是被人修改过了。常用的散列算法有MD5、SHA。



散列函数可以保证数据的完整性，识别出数据是否被篡改，但它并不能识别出数据是不是伪装的，因为中间人可以把数据和消息摘要同时替换，数据虽然是完整的，但真实数据被掉包了，接收者收到的并不是发送者发的，而是中间人的。
消息认证是解决数据真实性的办法。认证使用的技术有消息认证码和数字签名。

消息认证码

消息认证码（message authentication code）是一种可以确认消息完整性并进行认证（消息认证是指确认消息来自正确的发送者）的技术，简称 MAC。消息认证码可以简单理解为一种与密钥相关的单向散列函数。



Alice 给 Bob 发送消息前，先把共享密钥（key）发送给 Bob，Alice 把消息计算出 MAC 值，连同消息一起发送给 Bob，Bob 接收到消息和 MAC 值后，与本地计算得到 MAC 值对比，如果两者相同，就说明消息是完整的，而且可以确定是 Alice 发送的，没有中间人伪造。不过，消息认证码同样会遇到对称加密的密钥配送问题，因此解决密钥配送问题还是要采用公钥加密的方式。

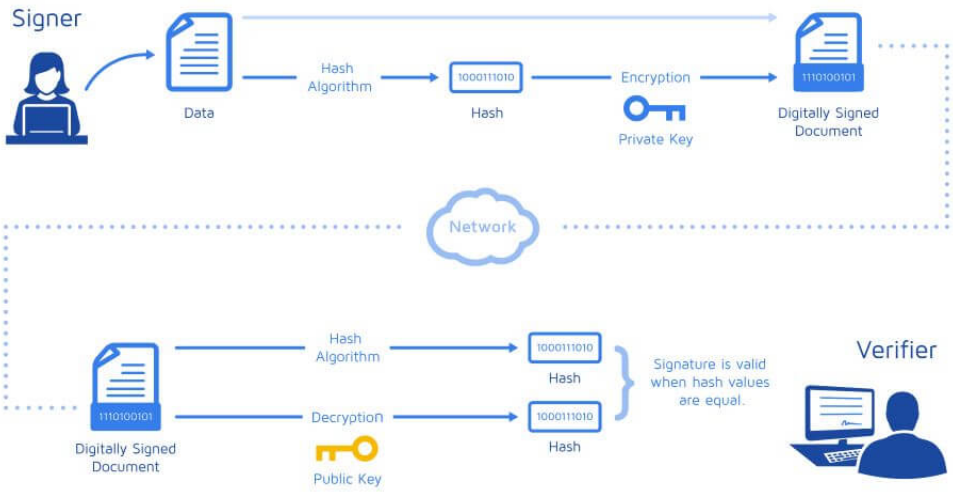
此外，消息认证码还有一个无法解决的问题，Bob 虽然可以识别出消息的篡改和伪装，但是 Alice 可以否认说：“我没发消息，应该是 Bob 的密钥被 Attacker 盗取了，这是 Attacker 发的吧”。Alice 这么说你还真没什么可以反驳的，那么如何防止 Alice 不承认呢，数字签名可以实现。

数字签名

Alice 发邮件找 Bob 借1万钱，因为邮件可以被人篡改（改成10万），也可以被伪造（Alice 根本就没发邮件，而是 Attacker 伪造 Alice 在发邮件），Alice 借了钱之后还可以不承认（不是我借的，我没有签名啊）。

消息认证码可以解决篡改和伪造的问题，Alice 不承认自己借了钱时，Bob 去找第三方机构做公正，即使这样，公正方也没法判断 Alice 有没有真的借钱，因为他们俩共享了密钥，也就是说两个都可以计算出正确的 MAC 值，Bob 说：“明明你发的消息和 MAC 值和我自己生成的 MAC 值一样，肯定是你发的消息”，Alice 说：“你把密钥透露给了其他人，是他发的邮件，你找他去吧”。Alice 矢口否认。

数字签名（Digital Signature）就可以解决否认的问题，发送消息时，Alice 和 Bob 使用不同的密钥，把公钥加密算法反过来使用，发送者 Alice 使用私钥对消息进行签名，而且只能是拥有私钥的 Alice 可以对消息签名，Bob 用配对的公钥去验证签名，第三方机构也可以用公钥验证签名，如果验证通过，说明消息一定是 Alice 发送的，抵赖也不行，因为你只有 Alice 可以生成签名。这就防止了否认的问题。



它的流程是:

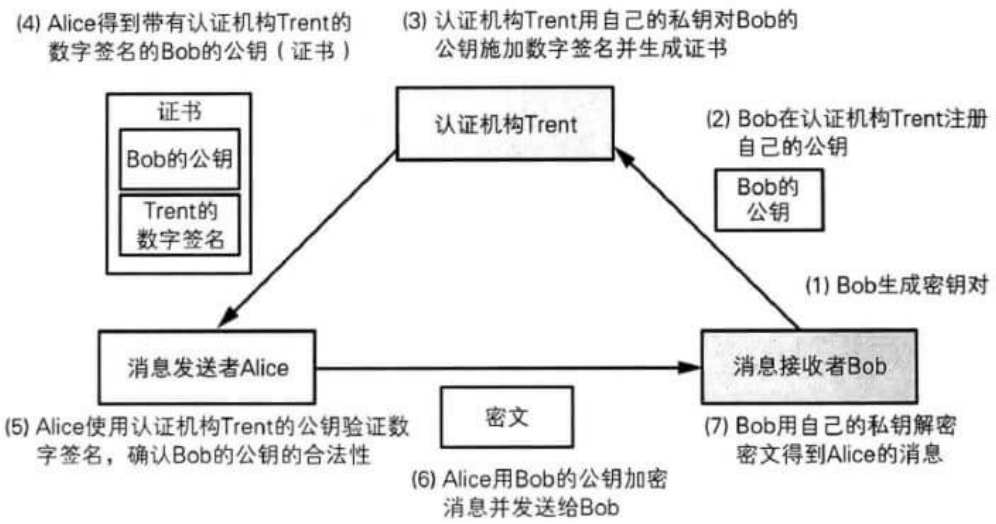
第一步：发送者 Alice **把消息哈希函数处理生成消息摘要，摘要信息使用私钥加密之后生成签名**，连同消息一起发送给接收者 Bob。

第二步：数据经过网络传输，Bob收到数据后，把签名和消息分别提取出来。

第三步：对签名进行验证，验证的过程是先把消息提取出来做同样的Hash处理，得到消息摘要，再与 Alice 传过来的签名用公钥解密，如果两者相等，就表示签名验证成功，否则验证失败，表示不是 Alice发的。

公钥证书

公钥密码在数字签名技术里面扮演举足轻重的角色，但是如何保证公钥是合法的呢，如果是遭到中间人攻击，掉包怎么办？这个时候公钥就应该交给一个第三方权威机构来管理，这个机构就是认证机构（Certification Authority）CA，CA 把用户的姓名、组织、邮箱地址等个人信息收集起来，还有此人的公钥，并由 CA 提供数字签名生成公钥证书（Public-Key Certificate）PKC，简称证书。



Alice 向 Bob 发送消息时，是通过 Bob 提供的公钥加密后的数据，而 Alice 获取的公钥并不是由 Bob 直接给的，而是由委托一个受信任的第三方机构给的。

1. Bob 生成密钥对，私钥自己保管，公钥交给认证机构 Trent。
2. Trent 经过一系列严格的检查确认公钥是 Bob 本人的
3. Trent 事先也生成自己的一套密钥对，用自己的私钥对 Bob 的公钥进行数字签名并生成数字证书。证书中包含了 Bob 的公钥。公钥在这里是不需要加密的，因为任何人获取 Bob 的公钥都没事，只要确定是 Bob 的公钥就行。
4. Alice 获取 Trent 提供的证书。
5. Alice 用 Trent 提供的公钥对证书进行签名验证，签名验证成功就表示证书中的公钥是 Bob 的。
6. 于是 Alice 就可以用 Bob 提供的公钥对消息加密后发送给 Bob。
7. Bob 收到密文后，用与之配对的私钥进行解密。

至此，一套比较完善的数据传输方案就完成了。HTTPS（SSL/TLS）就是在这样一套流程基础之上建立起来的。

15. HTTP和HTTPS的区别

- http是HTTP协议运行在TCP之上。所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。
- https是HTTP运行在SSL/TLS之上，SSL/TLS运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。此外客户端可以验证服务器端的身份，如果配置了客户端验证，服务器方也可以验证客户端的身份。
- https协议需要到ca申请证书，一般免费证书很少，需要交费。
- http是超文本传输协议，信息是明文传输，https 则是具有安全性的ssl加密传输协议
- http和https使用的是完全不同的连接方式用的端口也不一样,前者是80,后者是443。
- http的连接很简单,是无状态的
- HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议 要比http协议安全

16. HTTP2.0特性

HTTP/2的通过支持请求与响应的多路复用来减少延迟，通过压缩HTTP首部字段将协议开销降至最低，同时增加对请求优先级和服务器端推送的支持。

（1）二进制分帧

先来理解几个概念：

帧：HTTP/2 数据通信的最小单位消息：指 HTTP/2 中逻辑上的 HTTP 消息。例如请求和响应等，消息由一个或多个帧组成。

流：存在于连接中的一个虚拟通道。流可以承载双向消息，每个流都有一个唯一的整数ID。

HTTP/2 采用二进制格式传输数据，而非 HTTP 1.x 的文本格式，二进制协议解析起来更高效。 HTTP / 1 的请求和响应报文，都是由起始行，首部和实体正文（可选）组成，各部分之间以文本换行符分隔。HTTP/2 将请求和响应数据分割为更小的帧，并且它们采用二进制编码。

HTTP/2 中，同域名下所有通信都在单个连接上完成，该连接可以承载任意数量的双向数据流。每个数据流都以消息的形式发送，而消息又由一个或多个帧组成。多个帧之间可以乱序发送，根据帧首部的流标识可以重新组装。

（2）多路复用

多路复用，代替原来的序列和阻塞机制。所有就是请求的都是通过一个 TCP连接并发完成。 HTTP 1.x 中，如果想并发多个请求，必须使用多个 TCP 链接，且浏览器为了控制资源，还会对单个域名有 6-8个的TCP链接请求限制，如下图，红色圈出来的请求就因域名链接数已超过限制，而被挂起等待了一段时间。

在 HTTP/2 中，有了二进制分帧之后，HTTP /2 不再依赖 TCP 链接去实现多流并行了，在 HTTP/2中：

- 同域名下所有通信都在单个连接上完成。
- 单个连接可以承载任意数量的双向数据流。
- 数据流以消息的形式发送，而消息又由一个或多个帧组成，多个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装。

这一特性，使性能有了极大提升：

- **同个域名只需要占用一个 TCP 连接**，消除了因多个 TCP 连接而带来的延时和内存消耗。
- 单个连接上可以并行交错的请求和响应，之间互不干扰。
- 在HTTP/2中，每个请求都可以带一个31bit的优先值，0表示最高优先级， 数值越大优先级越低。有了这个优先值，客户端和服务器就可以在处理不同的流时采取不同的策略，以最优的方式发送流、消息和帧。

（3）服务器推送

服务端可以在发送页面HTML时主动推送其它资源，而不用等到浏览器解析到相应位置，发起请求再响应。例如服务端可以主动把JS和CSS文件推送给客户端，而不需要客户端解析HTML时再发送这些请求。

服务端可以主动推送，客户端也有权利选择是否接收。如果服务端推送的资源已经被浏览器缓存过，浏览器可以通过发送RST_STREAM帧来拒收。主动推送也遵守同源策略，服务器不会随便推送第三方资源给客户端。

（4）头部压缩

HTTP 1.1请求的大小变得越来越大，有时甚至会大于TCP窗口的初始大小，因为它们需要等待带着ACK的响应回来以后才能继续被发送。HTTP/2对消息头采用HPACK（专为http/2头部设计的压缩格式）进行压缩传输，能够节省消息头占用的网络的流量。而HTTP/1.x每次请求，都会携带大量冗余头信息，浪费了很多带宽资源。

第三部分：网络层

1. mac和ip怎么转换

ARP协议：

将IP地址通过广播 目标MAC地址是FF-FF-FF-FF-FF-FF 解析目标IP地址的MAC地址
扫描本网段MAC地址。

DHCP协议：

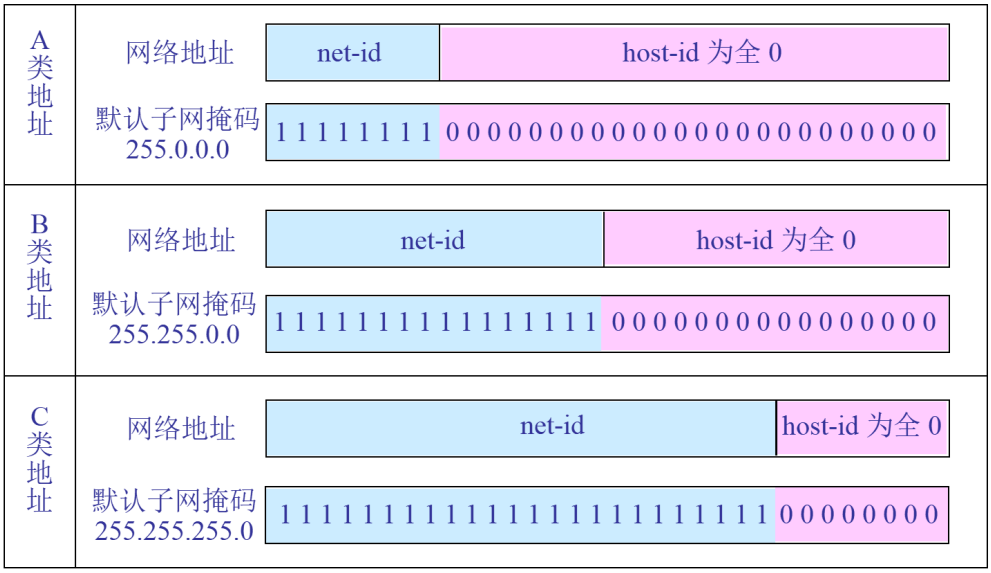
DHCP租约过程就是DHCP客户机动态获取IP地址的过程。

DHCP租约过程分为4步：

1. 客户机请求IP（客户机发DHCPDISCOVER广播包）；
2. 服务器响应（服务器发DHCPOFFER广播包）；
3. 客户机选择IP（客户机发DHCPREQUEST广播包）；
4. 服务器确定租约（服务器发DHCPACK/DHCPNAK广播包）。

2. IP地址子网划分

二进制	十进制
1	1
10	2
100	4
1000	8
10000	16
100000	32
1000000	64
10000000	128
10000000	128
11000000	192
11100000	224
11110000	240
11111000	248
11111100	252
11111110	254
11111111	255




```
IP分类
公有地址：
IP分类          缺省掩码
A 1—127         /8
B 128—191           /16
C 192—223       /24
D 224—239       组播地址
E 240—247       保留地址
私有地址：
A：10.0.0.0 - 10.255.255.255
B:   172.16.0.0 - 172.31.255.255
C:   192.168.0.0 - 192.168.255.255

判断合法的主机（IP）地址：
192.168.10.240/24           合法
192.168.10.0/24             不合法，主机位全为0，网络地址
192.168.10.255/24           不合法，主机位全为1，子网广播地址
255.255.255.255                 不合法，网络和主机位全为1，全网广播地址
127.x.x.x/8                   不合法，本地环回地址
172.16.3.5/24                 合法
192.168.5.240/32            合法
224.10.10.10.1                  不合法，组播地址
300.2.4.200/24                  不合法
```

- IP特殊地址
  - 本地环回地址：127.0.0.0 – 127.255.255.255，测试主机TCP/IP协议栈是否安装正确。
  - 本地链路地址：169.254.0.0 – 169.254.255.255，自动地址无法获取时系统自动配置占位。
  - 受限广播地址：255.255.255.255，发往这个地址的数据不能跨越三层设备，但本地网络内所有的主机都可以接收到数据

3. 地址解析协议ARP

4. 交换机和路由器的区别

1. 路由器可以给你的局域网自动分配IP，虚拟拨号，就像一个交通警察，指挥着你的电脑该往哪走，你自己不用操心那么多了。交换机只是用来分配网络数据的。
2. 路由器在网络层，路由器根据IP地址寻址，路由器可以处理TCP/IP协议，交换机不可以。

3. 交换机在中继层，交换机根据MAC地址寻址。路由器可以把一个IP分配给很多个主机使用，这些主机对外只表现出一个IP。交换机可以把很多主机连起来，这些主机对外各有各的IP。
4. 路由器提供防火墙的服务，交换机不能提供该功能。集线器、交换机都是做端口扩展的，就是扩大局域网(通常都是以太网)的接入点，也就是能让局域网可以连进来更多的电脑。路由器是用来做网间连接，也就是用来连接不同的网络。

交换机是利用**物理地址或者说MAC地址**来确定转发数据的目的地址。而路由器则是利用不同网络的ID号(即IP地址)来确定数据转发的地址。IP地址是在软件中实现的，描述的是设备所在的网络，有时这些第三层的地址也称为协议地址或者网络地址。MAC地址通常是硬件自带的，由网卡生产商来分配的，而且已经固化到了网卡中去，一般来说是不可更改的。而IP地址则通常由网络管理员或系统自动分配。

路由器和交换机的区别一：交换机是一根网线上网，但是大家上网是分别拨号，各自使用自己的宽带，大家上网没有影响。而路由器比交换机多了一个虚拟拨号功能，通过同一台路由器上网的电脑是共用一个宽带账号，大家上网要相互影响。
路由器和交换机的区别二：交换机工作在中继层，交换机根据MAC地址寻址。路由器工作在网络层，根据IP地址寻址，路由器可以处理TCP/IP协议，而交换机不可以。

路由器和交换机的区别三：交换机可以使连接它的多台电脑组成局域网，如果还有代理服务器的话还可以实现同时上网功能而且局域网所有电脑是共享它的带宽速率的，但是交换机没有路由器的自动识别数据包发送和到达地址的功能。路由器可以自动识别数据包发送和到达的地址，路由器相当于马路上的警察，负责交通疏导和指路的。

路由器和交换机的区别四：举几个例子,路由器是小邮局，就一个地址(IP)，负责一个地方的收发(个人电脑，某个服务器，所以你家上网要这个东西)，交换机是省里的大邮政中心，负责由一个地址给各个小地方的联系。简单的说路由器专管入网，交换机只管配送，路由路由就是给你找路让你上网的，交换机只负责开门，交换机上面要没有路由你是上不了网的。

路由器和交换机的区别五：路由器提供了防火墙的服务。路由器仅仅转发特定地址的数据包，不传送不支持路由协议的数据包传送和未知目标网络数据包的传送，从而可以防止广播风暴。

5. 子网掩码的作用

内网中192.168.1.199的前三组是网络号，后一组是主机号，子网掩码就是255.255.255.0

首先要说明的是：不是某个IP的网络号和主机号决定子网掩码是什么，而是子网掩码决定了某个IP地址的网络号与主机号是什么，IP地址是要搭配子网掩码使用的。例如上面的子网掩码决定了192.168.1.199的前三段192.168.1是网络号，最后一段199是主机号。

我们再来理解子网掩码的作用，先举个例子，市面上的两个厂家都生产电子秤，每个厂家都坚称他们的秤最准，那你是怎么知道他们的秤到底准不准？很简单，你去找一个 1KG 的国际千克原器，各放到他们的秤上测量，如果秤的测量值是1KG，那这把秤就是准的，**子网掩码的作用就相当于这个大家公认的国际千克原器，是我们测量两个IP是否属于同一个网段的一个工具（应该说是让你知道某个IP地址的网络号与主机号分别是什么）。**

如果让你判断一个IP地址：192.168.1.199的网络号和主机号分别是什么？

请问你怎么判断？你凭什么说192.168.1是网络号？199是主机号？有什么根据吗？

但是如果我给你一个IP地址是以下（带子网掩码）形式的：

IP：192.168.1.199

子网掩码：255.255.255.0

那么根据大家公认的规则，你就可以得出这个IP的网络号和主机号了，怎么算呢？

子网掩码的长度和IP地址一样也是一串32位的二进制数字，只不过为人类的可读性和记忆性的方便，通常使用十进制数字来表示，例如把上面的IP地址和子网掩码都转换成相应的二进制就是下面这样的：

	十进制	**二进制**

IP 地址：192.168.1.199 **->**11000000.10101000.00000001.11000111

子网掩码：255.255.255.0 **->**11111111.11111111.11111111.00000000

十进制的显示形式是给人看的，二进制的显示形式是给计算机看的。。。

子网掩码的左边是网络位，用二进制数字“1”表示，1的数目等于网络位的长度；右边是主机位，用二进制数字“0”表示，0的数目等于主机位的长度。

例如上面的子网掩码255.255.255.0的 “1”的个数是左边24位，则对应IP地址左边的位数也是24位;

十进制	**二进制**

IP 地址：192.168.1.199 **->11000000.10101000.00000001**.11000111

子网掩码：255.255.255.0 **->11111111.11111111.11111111**.00000000

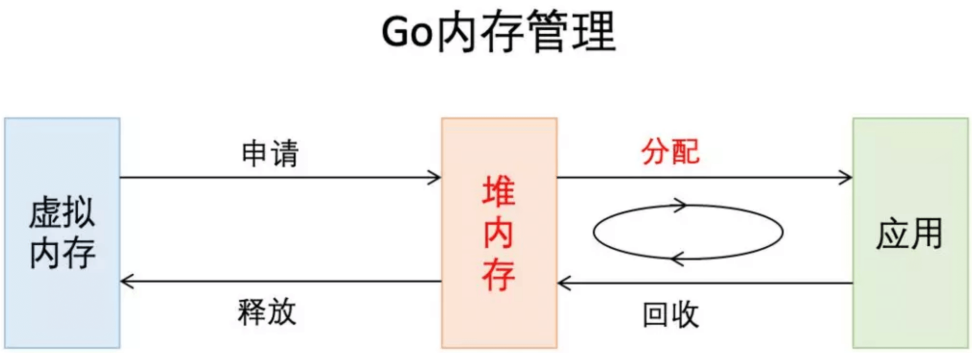
则这个IP地址的网络号就是11000000.10101000.00000001 ，转换成十进制就是 192.168.1，网掩码255.255.255.0的“0”的个数是右边8位，则这个IP地址的主机号就是11000111，转换成十进制就是199.

参考资料

- OSI七层参考模型
- OSI模型,TCP/IP协议栈
- 计算机网络-运输层-笔记
- 4internetLayer
- IP地址和子网划分
- 图解DHCP的4步租约过程
- 一文读懂 HTTP/2 特性
- 数字签名、数字证书、SSL、https是什么关系？
- HTTPS 为什么更安全
- SSL/TLS 握手过程详解
- 图解SSL/TLS协议
- 学习HTTP/2
- 详解 https 是如何确保安全的？
- 20-TCP 协议(滑动窗口——基础)
- 21-TCP 协议(滑动窗口——抓包分析)
- TCP 的那些事儿

Golang内存管理

Go内存管理



Go内存管理基于TCMalloc，使用连续虚拟地址，以页(8k)为单位、多级缓存进行管理：在分配内存时，需要对size进行对齐处理，根据best-fit找到合适的mspan，对未用完的内存还会拆分成其他大小的mspan继续使用。

在new一个object时(忽略逃逸分析)，根据object的size做不同的分配策略：

- 极小对象(size<16byte)直接在当前P的mcache上的tiny缓存上分配;
- 小对象(16byte <= size <= 32k)在当前P的mcache上对应slot的空闲列表中分配，无空闲列表则会继续向mcentral申请(还是没有则向mheap申请)；
- 大对象(size>32k)直接通过mheap申请。

1. 内存分配知识

- 计算机系统的主存被组织成一个由M个连续的字节大小的单元组成的数组，每个字节都有一个唯一的物理地址（PA）
- 现代处理器使用的是一种为虚拟寻址（VA）的寻址形式，最少的寻址单位是字
- 虚拟地址映射物理地址是通过读取页表（page table）进行地址翻译完成的：页表存放在物理存储器中
- MMU（内核吧物理页作为内存管理的基本单位）以页（page）大小为单位来管理系统中的页表
- 在虚拟存储器的习惯说法中，DRAM缓存不命中的成为：缺页
- page的结构与物理页相关，而非与虚拟页相关
- 系统中的每个物理页都要分配一个page结构体

在了解Go的内存分配器原理之前，我们先了解一下“动态存储分配器”。

2. 动态存储分配器

动态存储分配器维护着一个进程的虚拟存储区域，这个区域称为“堆”，堆可以视为一组大小不同的“块”（chunk: 连续的虚拟存储片，无论内存分配器和垃圾回收算法都依赖连续地址）的集合，并交由动态存储器维护。

动态分配器主要分为：

- 显式：常见的malloc.
- 隐式：垃圾回收.

在Go中，分配器将其管理（大块 -> 小块）的内存块分为两种：

- span：由多个连续的页（page）组成的大块内存.
- object：将span按特定大小切分多个小块，每个小块可存储一个对象.

按照其用途，span面向内部管理，object面向对象分配.

分配器按页数来区分不同大小的span。比如，以页数为单位将span存放到管理数组中，需要时就以页数为索引进行查找。当然，span大小并非固定不变。在获取闲置span时，如果没找到大小合适的，那就返回页数更多的，此时会引发裁剪操作，多余部分将构成新的span被放回管理数组。分配器还会尝试将地址相邻的空闲span合并，以构建更大的内存块，减少碎片，提供更灵活的分配策略。

我们在go的malloc.go代码中：

```
const (
    _PageShift = 13
    _PageSize  = 1 << _PageShift
    _PageMask  = _PageSize - 1
)
```

可以看到，用于存储对象的object，按照8字节倍数分为n种。这种方式虽然会造成一些内存浪费，但分配器只须面对有限的几种规格的小块内存，优化了分配和复用管理策略。

```
分配器会尝试将多个微小对象组合到一个object快内，以节约内存。
```

我们看到msize.go的部分：

```
var class_to_size [_NumSizeClasses]int32
var class_to_allocnpages [_NumSizeClasses]int32
var class_to_divmagic [_NumSizeClasses]divMagic

var size_to_class8 [1024/8 + 1]int8
var size_to_class128 [(_MaxSmallSize-1024)/128 + 1]int8

func sizeToClass(size int32) int32 {
    if size > _MaxSmallSize {
        throw("invalid size")
    }
    if size > 1024-8 {
        return int32(size_to_class128[(size-1024+127)>>7])
    }
    return int32(size_to_class8[(size+7)>>3])
}
```

分配器初始化时，会构建对照表存储大小和规格对应关系，包括用来切分的span页数。

```
// Tunable constants.
_MaxSmallSize = 32 << 10
```

从上面的代码段，我们大概可以指定若对象大小超出特定阈值限制，会被当做大对象特别对待。

3. mmap函数

Unix进程可以使用mmap函数来创建新的虚拟存储区域并将对象映射到这些区域中。

mmap函数要求内核创建一个新的虚拟存储区域，最好是从起始地址start开始的一个区域，并将文件描述符fd指定的对象的一个连续的片（chunk）映射到新的区域。

4. 数据频繁分配与回收

对于有效地进行数据频繁分配与回收，减少碎片，一般有两种手段：

- 空闲链表: 提供直接可供使用，已分配的结构块，缺点是不能全局控制.
- slab：linux提供的，可以把不同的对象划分为所谓高速缓存组.

5. Go的内存分配

Go的内存分配器是采用google自家的tcmalloc，tcmalloc是一个带内存池的分配器，底层直接调用mmap函数，并使用bestfit进行动态分配。

Go为每个系统线程分配了一个本地MCache,少量的地址分配就是从MCache分配的，并且定期进行垃圾回收，所以可见go的分配器包含了显式与隐式的调用。

Go定义的小块内存，大小上是指32K或以下的对象，go底层会把这些小块内存按照指定规格（大约100种）进行切割,为了避免随意切割，申请任意字节内存时会向上取整到接近的块，将整块分配（从空闲链表）给到申请者。

Go内存分配主要组件：

- MCache：层次与MHeap类似，对于每个尺寸的类别都有一个空闲链表。每个M都有自己的局部Mcache(小对象从它取，无需加锁)，这就是Go能够在多线程中高效内存管理的重要原因.
- MCentral：在无空闲内存的时候，向Mheap申请一个span,而不是多个，申请的span包含多少个page由central的sizeclass来确定（跨进程复用）
- MHeap：负责将MSpan组织和管理起来。

(1). 分配过程：从free中分配，如果发生切割则将剩余的部分放回到free中.

(2). 回收过程：回收一个Mspan时，首选查找它相邻的地址，再通过map映射得到对应的Mspan，如果Mspan的state是未使用，则可以将 两者进行合并。最后将这页或者合并后的页归还到free分配池或者large中。

6. Go的内存模型

Go的内存模型可以视为两级的内存模型：

第一级：Mheap为主要组件：分配的单位是页，但管理的单位是MSpan,每次分配都是用bestFit的原则分配连续的页，回收是采用位图的方式。

第二级：MCache为主要组件：相当于一个内存池， 回收采用引用计数器.

分配场景:

为对象分配内存须区分是在栈上分配还是在堆上分配。通常情况下，编译器有责任尽可能使用寄存器和栈来存储对象，这有助于提升性能，减少垃圾回收器的压力。

应用示例:

```
package main

func patent() *int {
    x := new(int)
    *x = 1234
    return x
}
```

```
func main() {
    println(*patent())
}
```

我们禁止内联优化来编译上面的代码：

```
> go build -gcflags="-l" -o patent main.go
```

得到的结果是:

```
        main.go:3       0x2040  65488b0c25a0080000      GS MOVQ GS:0x8a0, CX
        main.go:3       0x2049  483b6110              CMPQ 0x10(CX), SP
        main.go:3       0x204d  7639                  JBE 0x2088
        main.go:3       0x204f  4883ec18              SUBQ $0x18, SP
        main.go:3       0x2053  48896c2410            MOVQ BP, 0x10(SP)
        main.go:3       0x2058  488d6c2410            LEAQ 0x10(SP), BP
        main.go:4       0x205d  488d05dc3b0500        LEAQ 0x53bdc(IP), AX
        main.go:4       0x2064  48890424              MOVQ AX, 0(SP)
        main.go:4       0x2068  e823a70000            CALL runtime.newobject(SB)
        main.go:4       0x206d  488b442408            MOVQ 0x8(SP), AX
        main.go:5       0x2072  48c700d2040000        MOVQ $0x4d2, 0(AX)
        main.go:6       0x2079  4889442420            MOVQ AX, 0x20(SP)
        main.go:6       0x207e  488b6c2410            MOVQ 0x10(SP), BP
        main.go:6       0x2083  4883c418              ADDQ $0x18, SP
        main.go:6       0x2087  c3                    RET
        main.go:3       0x2088  e893720400            CALL runtime.morestack_noctxt(SB)
        main.go:3       0x208d  ebb1                  JMP main.patent(SB)
        :-1             0x208f  cc                    INT $0x3
```

从结果的 CALL runtime.newobject(SB)，证明我们的对象在堆上进行分配了。

但当使用默认参数，我们观察下结果：

```
> go build  -o patent main.go
```

当我们跟上面一样分析test的分配情况时：

```
> go tool objdump -s "main\.patent" patent
```

命令执行后，并没有输出,我们分析下main方法:

```
> go tool objdump -s "main\.main" patent
```

得到的结果如下：

```
        main.go:9       0x2040  65488b0c25a0080000      GS MOVQ GS:0x8a0, CX
        main.go:9       0x2049  483b6110              CMPQ 0x10(CX), SP
        main.go:9       0x204d  763d                  JBE 0x208c
        main.go:9       0x204f  4883ec18              SUBQ $0x18, SP
        main.go:9       0x2053  48896c2410            MOVQ BP, 0x10(SP)
        main.go:9       0x2058  488d6c2410            LEAQ 0x10(SP), BP
        main.go:10      0x205d  48c7442408d2040000    MOVQ $0x4d2, 0x8(SP)
        main.go:10      0x2066  e875210200            CALL runtime.printlock(SB)
        main.go:10      0x206b  48c70424d2040000      MOVQ $0x4d2, 0(SP)
```

```
        main.go:10      0x2073  e8b8280200              CALL runtime.printint(SB)
        main.go:10      0x2078  e8e3230200              CALL runtime.printnl(SB)
        main.go:10      0x207d  e8ee210200              CALL runtime.printunlock(SB)
        main.go:11      0x2082  488b6c2410              MOVQ 0x10(SP), BP
        main.go:11      0x2087  4883c418                ADDQ $0x18, SP
        main.go:11      0x208b  c3                      RET
        main.go:9       0x208c  e83f720400              CALL runtime.morestack_noctxt(SB)
        main.go:9       0x2091  ebad                    JMP main.main(SB)
```

这表明内联优化后的代码没有调用newobject在堆上分配内存。

编译器这么做的目的是：没有内联时，需要在两个栈帧间传递对象，因此在堆上分配而不是返回一个失效栈帧的数据。而当内联后，实际上就成看main栈帧内的局部变量，无需到堆上操作。

内存分配流程:

1、将小对象的大小向上取整到一个对应的尺寸类别（大约100种），查找相应的MCache的空闲链表，如果链表不空，直接从上面分配一个对象，这个过程不加锁

2、如果MCache自由链表是空的，通过MCentral的自由链表取一些对象进行补充

3、如果MCentral的自由链表是空的，则往MHeap中取用一些页对MCentral进行补充，然后将这些内存截断成特定规格

4、如果MHeap空或者没有足够大的页的情况下，从操作系统分配一组新的页面，一般在1MB以上

Go分配流程核心源码实现：

```
func mallocgc(size uintptr, typ *_type, flags uint32) unsafe.Pointer {
    if gcphase == _GCmarktermination {
        throw("mallocgc called with gcphase == _GCmarktermination")
    }

    if size == 0 {
        return unsafe.Pointer(&zerobase)
    }

    if flags&flagNoScan == 0 && typ == nil {
        throw("malloc missing type")
    }

    if debug.sbrk != 0 {
        align := uintptr(16)
        if typ != nil {
            align = uintptr(typ.align)
        }
        return persistentalloc(size, align, &memstats.other_sys)
    }

    // assistG is the G to charge for this allocation, or nil if
    // GC is not currently active.
    var assistG *g
    if gcBlackenEnabled != 0 {
        // Charge the current user G for this allocation.
        assistG = getg()
        if assistG.m.curg != nil {
            assistG = assistG.m.curg
        }
        // Charge the allocation against the G. We'll account
        // for internal fragmentation at the end of mallocgc.
        assistG.gcAssistBytes -= int64(size)
```

```
        if assistG.gcAssistBytes < 0 {
            // This G is in debt. Assist the GC to correct
            // this before allocating. This must happen
            // before disabling preemption.
            gcAssistAlloc(assistG)
        }
    }

    // Set mp.mallocing to keep from being preempted by GC.
    mp := acquirem()
    if mp.mallocing != 0 {
        throw("malloc deadlock")
    }
    if mp.gsignal == getg() {
        throw("malloc during signal")
    }
    mp.mallocing = 1

    shouldhelpgc := false
    dataSize := size
    c := gomcache()
    var s *mspan
    var x unsafe.Pointer
    if size <= maxSmallSize {
        if flags&flagNoScan != 0 && size < maxTinySize {
            //小对象分配
            off := c.tinyoffset
            // Align tiny pointer for required (conservative) alignment.
            if size&7 == 0 {
                off = round(off, 8)
            } else if size&3 == 0 {
                off = round(off, 4)
            } else if size&1 == 0 {
                off = round(off, 2)
            }
            if off+size <= maxTinySize && c.tiny != 0 {
                // The object fits into existing tiny block.
                x = unsafe.Pointer(c.tiny + off)
                c.tinyoffset = off + size
                c.local_tinyallocs++
                mp.mallocing = 0
                releasem(mp)
                return x
            }
            // Allocate a new maxTinySize block.
            s = c.alloc[tinySizeClass]
            v := s.freelist
            if v.ptr() == nil {
                systemstack(func() {
                    c.refill(tinySizeClass)
                })
                shouldhelpgc = true
                s = c.alloc[tinySizeClass]
                v = s.freelist
            }
            s.freelist = v.ptr().next
            s.ref++
            // prefetchnta offers best performance, see change list message.
            prefetchnta(uintptr(v.ptr().next))
            x = unsafe.Pointer(v)
            (*[2]uint64)(x)[0] = 0
```

```
            (*[2]uint64)(x)[1] = 0
            // See if we need to replace the existing tiny block with the new one
            // based on amount of remaining free space.
            if size < c.tinyoffset || c.tiny == 0 {
                c.tiny = uintptr(x)
                c.tinyoffset = size
            }
            size = maxTinySize
        } else {
            var sizeclass int8
            if size <= 1024-8 {
                sizeclass = size_to_class8[(size+7)>>3]
            } else {
                sizeclass = size_to_class128[(size-1024+127)>>7]
            }
            size = uintptr(class_to_size[sizeclass])
            s = c.alloc[sizeclass]
            v := s.freelist
            if v.ptr() == nil {
                systemstack(func() {
                    c.refill(int32(sizeclass))
                })
                shouldhelpgc = true
                s = c.alloc[sizeclass]
                v = s.freelist
            }
            s.freelist = v.ptr().next
            s.ref++
            // prefetchnta offers best performance, see change list message.
            prefetchnta(uintptr(v.ptr().next))
            x = unsafe.Pointer(v)
            if flags&flagNoZero == 0 {
                v.ptr().next = 0
                if size > 2*sys.PtrSize && ((*[2]uintptr)(x))[1] != 0 {
                    memclr(unsafe.Pointer(v), size)
                }
            }
        }
    } else {
        var s *mspan
        shouldhelpgc = true
        systemstack(func() {
            s = largeAlloc(size, flags)
        })
        x = unsafe.Pointer(uintptr(s.start << pageShift))
        size = s.elemsize
    }

    if flags&flagNoScan != 0 {
        // All objects are pre-marked as noscan. Nothing to do.
    } else {
        if typ == deferType {
            dataSize = unsafe.Sizeof(_defer{})
        }
        heapBitsSetType(uintptr(x), size, dataSize, typ)
        if dataSize > typ.size {
            // Array allocation. If there are any
            // pointers, GC has to scan to the last
            // element.
            if typ.ptrdata != 0 {
                c.local_scan += dataSize - typ.size + typ.ptrdata
            }
```

```
        } else {
            c.local_scan += typ.ptrdata
        }
        publicationBarrier()
    }
    if gcphase == _GCmarktermination || gcBlackenPromptly {
        systemstack(func() {
            gcmarknewobject_m(uintptr(x), size)
        })
    }

    if raceenabled {
        racemalloc(x, size)
    }
    if msanenabled {
        msanmalloc(x, size)
    }

    mp.mallocing = 0
    releasem(mp)

    if debug.allocfreetrace != 0 {
        tracealloc(x, size, typ)
    }

    if rate := MemProfileRate; rate > 0 {
        if size < uintptr(rate) && int32(size) < c.next_sample {
            c.next_sample -= int32(size)
        } else {
            mp := acquirem()
            profilealloc(mp, x, size)
            releasem(mp)
        }
    }

    if assistG != nil {
        // Account for internal fragmentation in the assist
        // debt now that we know it.
        assistG.gcAssistBytes -= int64(size - dataSize)
    }

    if shouldhelpgc && gcShouldStart(false) {
        gcStart(gcBackgroundMode, false)
    }

    return x
}
```

Go也有happens-before ,go happens-before常用的三原则是：

- 对于不带缓冲区的channel，对其写happens-before对其读.
- 对于带缓冲区的channel,对其读happens-before对其写.
- 对于不带缓冲的channel的接收操作 happens-before 相应channel的发送操作完成.

Golang runtime的调度

Golang runtime的调度

Golang作为一个为并发而产生的语言，从Golang产生的那一刻就注定它具有高并发的特性，而 Go 语言中的并发（并行）编程是经由 goroutine 实现的，goroutine 是 Golang 最重要的特性之一，具有使用成本低、消耗资源低、能效高等特点，官方宣称原生 goroutine 并发成千上万不成问题，于是它也成为 Gopher 们经常使用的特性。

Goroutine，Go 语言基于并发（并行）编程的核心。那么 Goroutine 是什么？

通常 goroutine 会被当做 coroutine（协程）的 golang 实现，从比较粗浅的层面来看，这种认知也算是合理.

但实际上，goroutine 并非传统意义上的协程，现在主流的线程模型分三种： 内核级线程模型、用户级线程模型和两级线程模型（也称混合型线程模型），传统的协程库属于用户级线程模型.

而 goroutine 和它的Go Scheduler在底层实现上其实是属于两级线程模型，

通常 goroutine 会被当做 coroutine（协程）的 golang 实现，从比较粗浅的层面来看，这种认知也算是合理.

但是，goroutine 并非传统意义上的协程，现在主流的线程模型分三种：

1. 内核级线程模型.
2. 用户级线程模型和两级线程模型（也称混合型线程模型）.
3. 传统的协程库属于用户级线程模型.

因此，有时候为了方便理解可以简单把 goroutine 类比成协程，但心里一定要有个清晰的认知 `goroutine` 并不等同于协程。

线程

计算机在是早期的单进程操作系统，这样就面临2个问题：

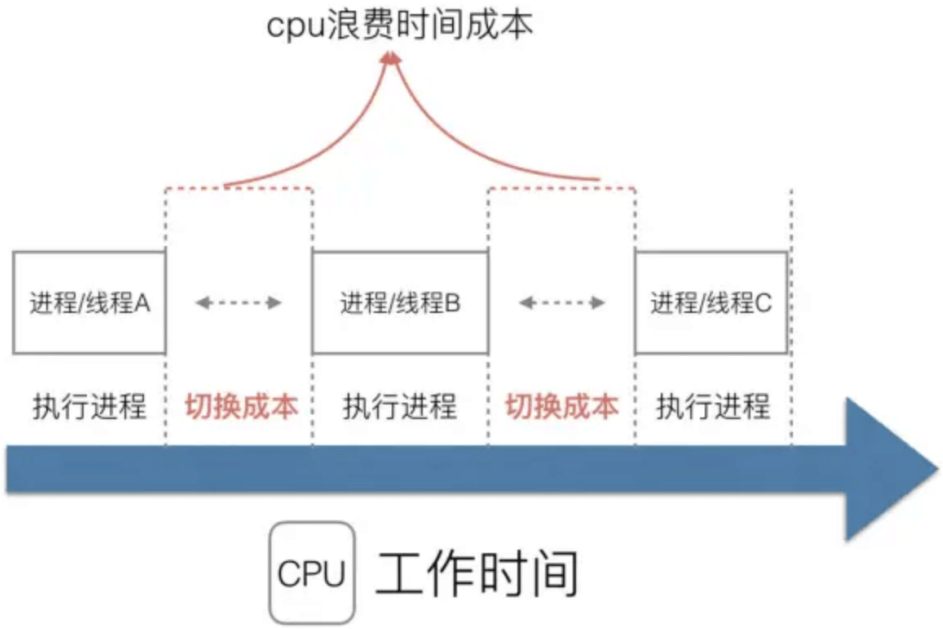
1. 单一的执行流程，计算机只能一个任务一个任务处理。

2. 进程阻塞所带来的CPU时间浪费。

随着技术的发展,后面的操作系统就具有了最早的并发能力：多进程并发，当一个进程阻塞的时候，切换到另外等待执行的进程，这样就能尽量把CPU利用起来，CPU就不浪费了.

在多进程/多线程的操作系统中，就是为了解决在单线程系统中的阻塞的问题，因为一个进程阻塞cpu可以立刻切换到其他进程中去执行，而且调度cpu的算法可以保证在运行的进程都可以被分配到cpu的运行时间片。这样从宏观来看，似乎多个进程是在同时被运行。

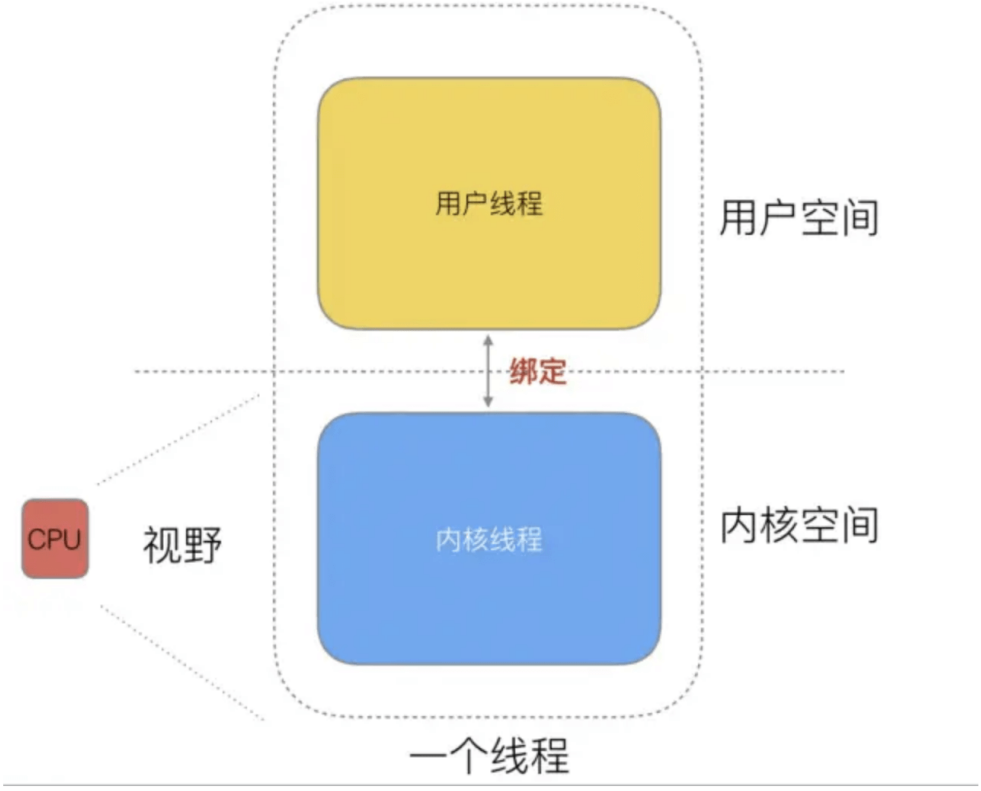
但新的问题就又出现了，进程拥有太多的资源，进程的创建、切换、销毁，都会占用很长的时间，CPU虽然利用起来了，但如果进程过多，CPU有很大的一部分都被用来进行进程调度了。



但是要怎么才能提高CPU的利用率呢？对于Linux操作系统来讲，cpu对进程的态度和线程的态度是一样的。

很明显，CPU调度切换的是进程和线程。尽管线程看起来很美好，但实际上多线程开发设计会变得更加复杂，要考虑很多同步竞争等问题，如锁、竞争冲突等。

随着时间的发展,工程师们发现，其实一个线程分为“内核态“线程和”用户态“线程。 即一个“用户态线程”必须要绑定一个“内核态线程”，但是CPU并不知道有“用户态线程”的存在，它只知道它运行的是一个“内核态线程”(系统的PCB进程控制块)。



这样，我们可以分类一下，内核线程依然叫“线程(thread)”，用户线程就叫“协程(co-routine)”.

线程的实现模型主要有 3 种：内核级线程模型、用户级线程模型和两级线程模型（也称混合型线程模型），它们之间最大的差异就在于用户线程与内核调度实体（KSE，Kernel Scheduling Entity）之间的对应关系上。而所谓的内核调度实体 KSE 就是指可以被操作系统内核调度器调度的对象实体。

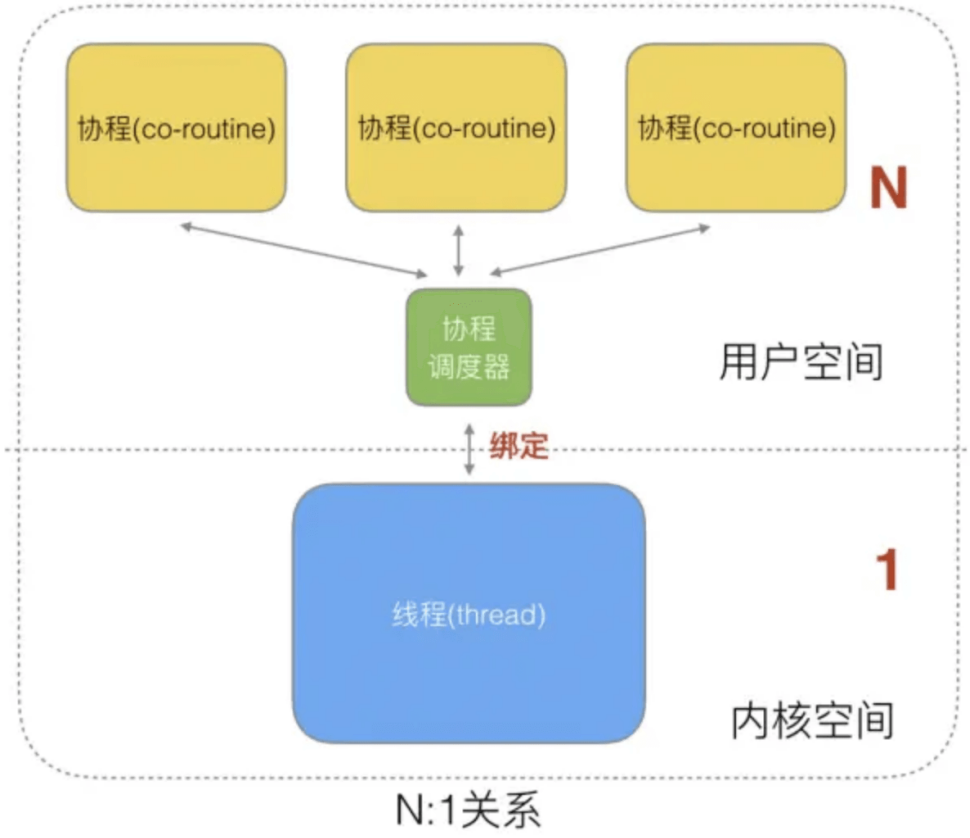
简单来说 KSE 就是内核级线程，是操作系统内核的最小调度单元，也就是我们写代码的时候通俗理解上的线程了。

用户级线程模型

用户线程与内核线程 KSE 是多对一（N：1）的映射模型，多个用户线程的一般从属于单个进程并且多线程的调度是由用户自己的线程库来完成，线程的创建、销毁以及多线程之间的协调等操作都是由用户自己的线程库来负责而无须借助系统调用来实现。一个进程中所有创建的线程都只和同一个 KSE 在运行时动态绑定，也就是说，操作系统只知道用户进程而对其中的线程是无感知的，内核的所有调度都是基于用户进程。许多语言实现的 协程库 基本上都属于这种方式（比如 python 的 gevent）。

由于线程调度是在用户层面完成的，也就是相较于内核调度不需要让 CPU 在用户态和内核态之间切换，这种实现方式相比内核级线程可以做的很轻量级，对系统资源的消耗会小很多，因此可以创建的线程数量与上下文切换所花费的代价也会小得多。但该模型有个原罪：并不能做到真正意义上的并发，假设在某个用户进程上的某个用户线程因为一个阻塞调用（比如 I/O 阻塞）而被 CPU 给中断（抢占式调度）了，那么该进程内的所有线程都被阻塞（因为单个用户进程内的线程自调度是没有 CPU 时钟中断的，从而没有轮转调度），整个进程被挂起。即便是多 CPU 的机器，也无济于事，因为在用户级线程模型下，一个 CPU 关联运行的是整个用户进程，进程内的子线程绑定到 CPU 执行是由用户进程调度的，内部线程对 CPU 是不可见的，此时可以理解为 CPU 的调度单位是用户进程。

所以很多的协程库会把自己一些阻塞的操作重新封装为完全的非阻塞形式，然后在以前要阻塞的点上，主动让出自己，并通过某种方式通知或唤醒其他待执行的用户线程在该 KSE 上运行，从而避免了内核调度器由于 KSE 阻塞而做上下文切换，这样整个进程也不会被阻塞了。



- 特点:

  N个协程绑定1个线程，优点就是协程在用户态线程即完成切换，不会陷入到内核态，这种切换非常的轻量快速。但也有很大的缺点，1个进程的所有协程都绑定在1个线程上.

- 缺点

1. 某个程序用不了硬件的多核加速能力.
2. 一旦某协程阻塞，造成线程阻塞，本进程的其他协程都无法执行了，无并发能力.

内核级线程模型

用户线程与内核线程 KSE 是一对一（1：1）的映射模型，也就是每一个用户线程绑定一个实际的内核线程，而线程的调度则完全交付给操作系统内核去做，应用程序对线程的创建、终止以及同步都基于内核提供的系统调用来完成，大部分编程语言的线程库(比如 Java 的 `java.lang.Thread` 、C++11 的 `std::thread` 等等)都是对操作系统的线程（内核级线程）的一层封装，创建出来的每个线程与一个独立的 KSE 静态绑定，因此其调度完全由操作系统内核调度器去做，也就是说，一个进程里创建出来的多个线程每一个都绑定一个 KSE。这种模型的优势和劣势同样明显：优势是实现简单，直接借助操作系统内核的线程以及调度器，所以 CPU 可以快速切换调度线程，于是多个线程可以同时运行，因此相较于用户级线程模型它真正做到了并行处理；但它的劣势是，由于直接借助了操作系统内核来创建、销毁和以及多个线程之间的上下文切换和调度，因此资源成本大幅上涨，且对性能影响很大。

- 特点:

1个协程绑定1个线程，这种最容易实现。协程的调度都由CPU完成了，不存在N:1缺点。

- 缺点

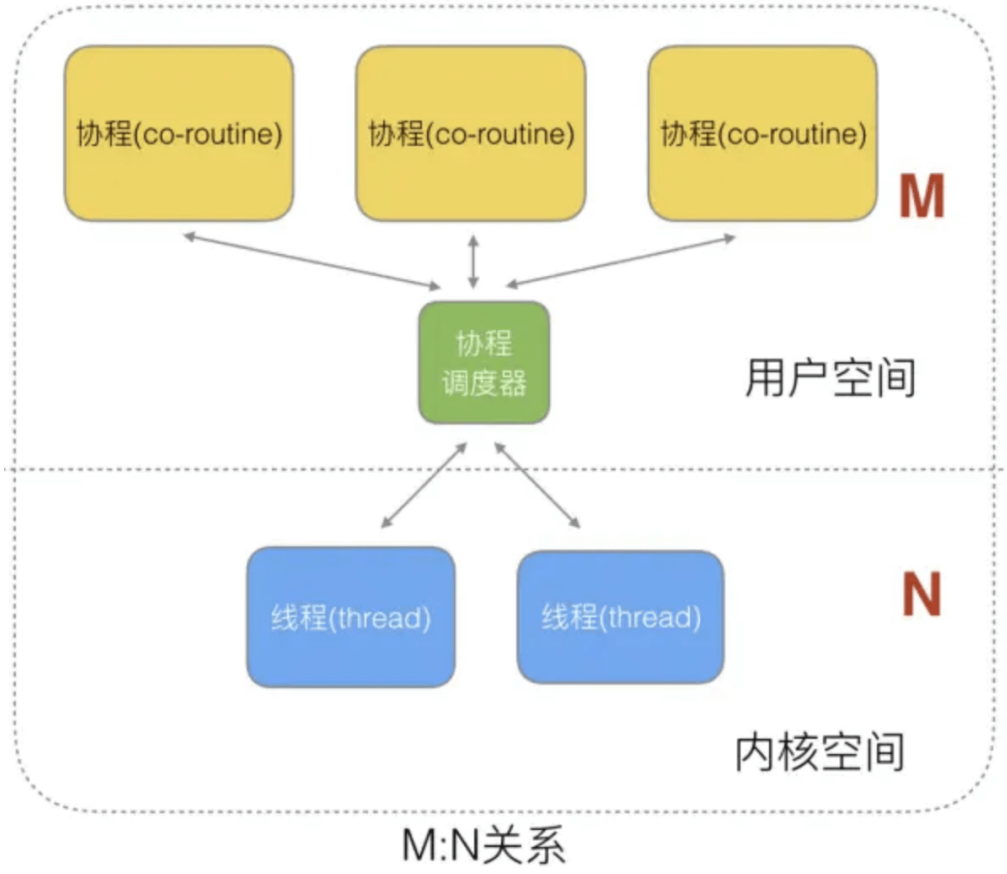
协程的创建、删除和切换的代价都由CPU完成，有点略显昂贵了。

两级线程模型

两级线程模型是博采众长之后的产物，充分吸收前两种线程模型的优点且尽量规避它们的缺点。在此模型下，用户线程与内核 KSE 是多对多（N：M）的映射模型：首先，区别于用户级线程模型，两级线程模型中的一个进程可以与多个内核线程 KSE 关联，也就是说一个进程内的多个线程可以分别绑定一个自己的 KSE，这点和内核级线程模型相似；其次，又区别于内核级线程

模型，它的进程里的线程并不与 KSE 唯一绑定，而是可以多个用户线程映射到同一个 KSE，当某个 KSE 因为其绑定的线程的阻塞操作被内核调度出 CPU 时，其关联的进程中其余用户线程可以重新与其他 KSE 绑定运行。

所以，两级线程模型既不是用户级线程模型那种完全靠自己调度的也不是内核级线程模型完全靠操作系统调度的，而是中间态（自身调度与系统调度协同工作），因为这种模型的高度复杂性，操作系统内核开发者一般不会使用，所以更多时候是作为第三方库的形式出现，而 Go 语言中的 runtime 调度器就是采用的这种实现方案，实现了 Goroutine 与 KSE 之间的动态关联，不过 Go 语言的实现更加高级和优雅；该模型为何被称为两级？即用户调度器实现用户线程到 KSE 的『调度』，内核调度器实现 KSE 到 CPU 上的 `调度` 。



- 特点

G只能运行在M中，一个M必须持有一个P，M与P是1：1的关系。M会从P的本地队列弹出一个可执行状态的G来执行，如果P的本地队列为空，就会向其他的MP组合偷取一个可执行的G来执行，即M个协程绑定1个线程，是N:1和1:1类型的结合，克服了以上2种模型的缺点，但实现起来最为复杂。

协程跟线程是有区别的，线程由CPU调度是抢占式的，协程由用户态调度是协作式的，一个协程让出CPU后，才执行下一个协程。

Go的协程Goroutine

Go为了提供更容易使用的并发方法，使用了goroutine和channel。goroutine来自协程的概念，让一组可复用的函数运行在一组线程之上，即使有协程阻塞，该线程的其他协程也可以被runtime调度，转移到其他可运行的线程上。最关键的是，开发人员是看不到这些底层的细节，这就降低了编程的难度，提供了更容易的并发。

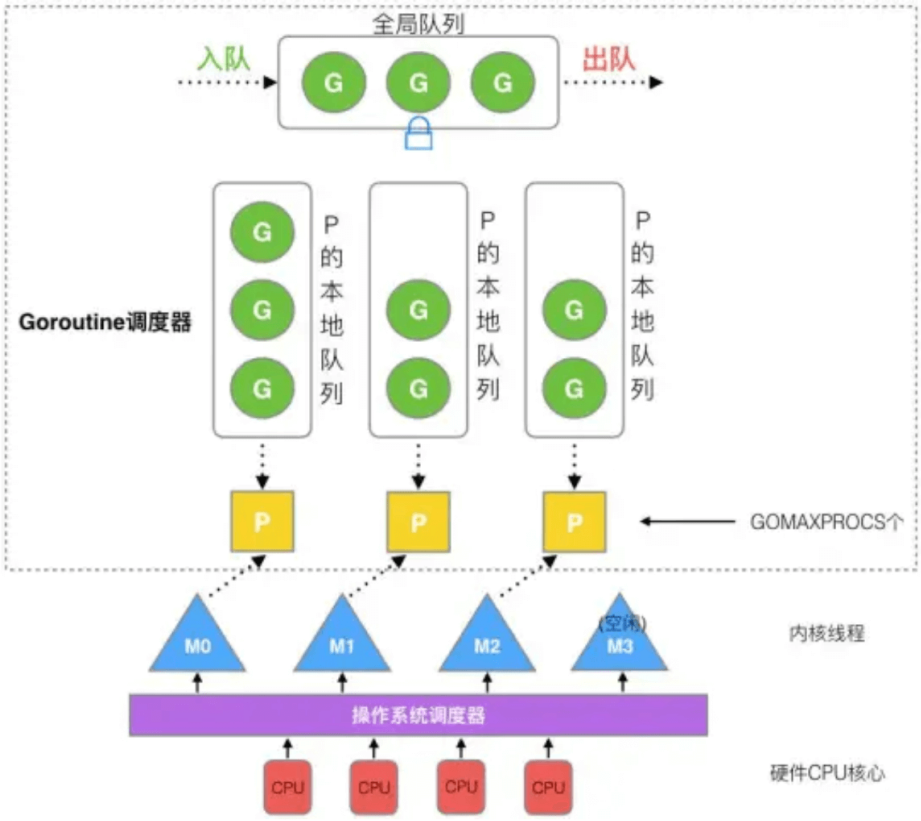
Go中，协程被称为goroutine，它非常轻量，一个goroutine只占几KB，并且这几KB就足够goroutine运行完，这就能在有限的内存空间内支持大量goroutine，支持了更多的并发。虽然一个goroutine的栈只占几KB，但实际是可伸缩的，如果需要更多内容，runtime会自动为goroutine分配。

Goroutine特点:

- 占用内存更小(几Kb).
- 调度更灵活(runtime调度).

GPM模型

在Go中，线程是运行goroutine的实体，调度器的功能是把可运行的goroutine分配到工作线程上。



每一个 OS 线程都有一个固定大小的内存块(一般会是 2MB)来做栈，这个栈会用来存储当前正在被调用或挂起(指在调用其它函数时)的函数的内部变量。这个固定大小的栈同时很大又很小。因为 2MB 的栈对于一个小小的 goroutine 来说是很大的内存浪费，而对于一些复杂的任务（如深度嵌套的递归）来说又显得太小。因此，Go 语言做了它自己的 `线程` 。

在 Go 语言中，每一个 goroutine 是一个独立的执行单元，相较于每个 OS 线程固定分配 2M 内存的模式，goroutine 的栈采取了动态扩容方式， 初始时仅为 2KB，随着任务执行按需增长，最大可达 1GB（64 位机器最大是 1G，32 位机器最大是 256M），且完全由 golang 自己的调度器 Go Scheduler 来调度。

此外，GC 还会周期性地将不再使用的内存回收，收缩栈空间。 因此，Go 程序可以同时并发成千上万个 goroutine 是得益于它强劲的调度器和高效的内存模型。Go 的创造者大概对 goroutine 的定位就是屠龙刀，因为他们不仅让 goroutine 作为 golang 并发编程的最核心组件（开发者的程序都是基于 goroutine 运行的）而且 golang 中的许多标准库的实现也到处能见到 goroutine 的身影，比如 net/http 这个包，甚至语言本身的组件 runtime 运行时和 GC 垃圾回收器都是运行在 goroutine 上的，作者对 goroutine 的厚望可见一斑。

任何用户线程最终肯定都是要交由 OS 线程来执行的，goroutine（称为 G）也不例外，但是 G 并不直接绑定 OS 线程运行，而是由 Goroutine Scheduler 中的 P - Logical Processor （逻辑处理器）来作为两者的传递者，P 可以看作是一个抽象的资源或者一个上下文，一个 P 绑定一个 OS 线程.

在 golang 的实现里把 OS 线程抽象成一个数据结构：M，G 实际上是由 M 通过 P 来进行调度运行的，但是在 G 的层面来看，P 提供了 G 运行所需的一切资源和环境，因此在 G 看来 P 就是运行它的 “CPU”，由 G、P、M 这三种由 Go 抽象出来的实现，最终形成了 Go 调度器的基本结构：

- G: 表示 Goroutine，每个 Goroutine 对应一个 G 结构体，G 存储 Goroutine 的运行堆栈、状态以及任务函数，可重用。G 并非执行体，每个 G 需要绑定到 P 才能被调度执行。

- P: Processor，表示逻辑处理器， 对 G 来说，P 相当于 CPU 核，G 只有绑定到 P(在 P 的 local runq 中)才能被调度。对 M 来说，P 提供了相关的执行环境(Context)，如内存分配状态(mcache)，任务队列(G)等，P 的数量决定了系统内最大可并行的 G 的数量（前提：物理 CPU 核数 >= P 的数量），P 的数量由用户设置的 GOMAXPROCS 决定，但是不论 GOMAXPROCS 设置为多大，P 的数量最大为 256。

- M: Machine，OS 线程抽象，负责调度任务, 代表着真正执行计算的资源，在绑定有效的 P 后，进入 schedule 循环；而 schedule 循环的机制大致是从 Global 队列、P 的 Local 队列以及 wait 队列中获取 G，切换到 G 的执行栈上并执行 G 的函数，调用 goexit 做清理工作并回到 M，如此反复。M 并不保留 G 状态，这是 G 可以跨 M 调度的基础，M 的数量是不定的，由 Go Runtime 调整，为了防止创建过多 OS 线程导致系统调度不过来，目前默认最大限制为 10000 个。

在新的版本1.13.6中Go的GPM的模型的源码位于 `src/runtime/runtime2.go` . 至于为什么M的的最大数量限制在10000, 在这里可以查看

关于 P, 其实在 Go 1.0 发布的时候，它的调度器其实 G-M 模型，也就是没有 P 的，调度过程全由 G 和 M 完成，这个模型暴露出一些问题：

单一全局互斥锁(Sched.Lock)和集中状态存储的存在导致所有 goroutine 相关操作，比如：创建、重新调度等都要上锁；

- goroutine 传递问题：M 经常在 M 之间传递可运行的 goroutine，这导致调度延迟增大以及额外的性能损耗；

- 每个 M 做内存缓存，导致内存占用过高，数据局部性较差；

- 由于 syscall 调用而形成的剧烈的 `worker thread` 阻塞和解除阻塞，导致额外的性能损耗。

这些问题实在太严重了，导致 Go1.0 虽然号称原生支持并发，却在并发性能上一直饱受诟病，于是Dmitry Vyukov在 Scalable Go Scheduler Design Doc提出该模型在并发伸缩性方面的问题，并通过加入P(Processors)来改进该问题。

在重新设计和实现了 Go 调度器（在原有的 G-M 模型中引入了 P）并且实现了一个叫做 work-stealing 的调度算法：

- 每个 P 维护一个 G 的本地队列；

- 当一个 G 被创建出来，或者变为可执行状态时，就把他放到 P 的可执行队列中；

- 当一个 G 在 M 里执行结束后，P 会从队列中把该 G 取出；如果此时 P 的队列为空，即没有其他 G 可以执行， M 就随机选择另外一个 P，从其可执行的 G 队列中取走一半。

该算法避免了在 goroutine 调度时使用全局锁。

GPM调度流程

Go 调度器工作时会维护两种用来保存 G 的任务队列：一种是一个 Global 任务队列，一种是每个 P 维护的 Local 任务队列。

当通过go关键字创建一个新的 goroutine 的时候，它会优先被放入 P 的本地队列。为了运行 goroutine，M 需要持有（绑定）一个 P，接着 M 会启动一个 OS 线程，循环从 P 的本地队列里取出一个 goroutine 并执行。

当然上面提到的 `work-stealing` 调度算法：当 M 执行完了当前 P 的 Local 队列里的所有 G 后，P 也不会就这么在那干等着啥都不干，它会先尝试从 Global 队列寻找 G 来执行，如果 Global 队列为空，它会随机挑选另外一个 P，从它的队列里中拿走一半的 G 到自己的队列中执行。

```
// go1.13.6 src/runtime/proc.go

// 省略了GC检查等其它细节，只保留了主要流程
// g:       G结构体定义
// sched:   Global队列
// 获取一个待执行的G
// 尝试从其他P中steal，从全局队列中获取g，轮询网络。
func findrunnable() (gp *g, inheritTime bool) {
    // 获取当前的G对象
    _g_ := getg()

    // The conditions here and in handoffp must agree: if
    // findrunnable would return a G to run, handoffp must start
    // an M.

top:
    // 获取当前P对象
    _p_ := _g_.m.p.ptr()
    if sched.gcwaiting != 0 {
        gcstopm()
        goto top
    }
    if _p_.runSafePointFn != 0 {
        runSafePointFn()
    }
    if fingwait && fingwake {
        if gp := wakefing(); gp != nil {
            ready(gp, 0, true)
        }
    }
    if *cgo_yield != nil {
        asmcgocall(*cgo_yield, nil)
    }

    // 1. 尝试从P的Local队列中取得G 优先_p_.runnext 然后再从Local队列中取
    if gp, inheritTime := runqget(_p_); gp != nil {
        return gp, inheritTime
    }

    // 2. 尝试从Global队列中取得G
    if sched.runqsize != 0 {
        lock(&sched.lock)
        // globrunqget从Global队列中获取G 并转移一批G到_p_的Local队列
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        if gp != nil {
            return gp, false
        }
    }

    // Poll network.
    // This netpoll is only an optimization before we resort to stealing.
    // We can safely skip it if there are no waiters or a thread is blocked
    // in netpoll already. If there is any kind of logical race with that
    // blocked thread (e.g. it has already returned from netpoll, but does
    // not set lastpoll yet), this thread will do blocking netpoll below
    // anyway.

    // 3. 检查netpoll任务
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 && atomic.Load64(&sched.lastpoll) != 0 {
```

```
        if list := netpoll(false); !list.empty() { // non-blocking
            gp := list.pop()
            // netpoll返回的是G链表，将其它G放回Global队列
            injectglist(&list)
            casgstatus(gp, _Gwaiting, _Grunnable)
            if trace.enabled {
                traceGoUnpark(gp, 0)
            }
            return gp, false
        }
    }

    // 4. 尝试从其它P窃取任务
    procs := uint32(gomaxprocs)
    if atomic.Load(&sched.npidle) == procs-1 {
        // Either GOMAXPROCS=1 or everybody, except for us, is idle already.
        // New work can appear from returning syscall/cgocall, network or timers.
        // Neither of that submits to local run queues, so no point in stealing.
        goto stop
    }
    // If number of spinning M's >= number of busy P's, block.
    // This is necessary to prevent excessive CPU consumption
    // when GOMAXPROCS>>1 but the program parallelism is low.
    if !_g_.m.spinning && 2*atomic.Load(&sched.nmspinning) >= procs-atomic.Load(&sched.npidle) {
        goto stop
    }
    if !_g_.m.spinning {
        _g_.m.spinning = true
        atomic.Xadd(&sched.nmspinning, 1)
    }
    for i := 0; i < 4; i++ {
        // 随机P的遍历顺序
        for enum := stealOrder.start(fastrand()); !enum.done(); enum.next() {
            if sched.gcwaiting != 0 {
                goto top
            }
            stealRunNextG := i > 2 // first look for ready queues with more than 1 g
            // runqsteal执行实际的steal工作，从目标P的Local队列转移一般的G过来
            // stealRunNextG指是否steal目标P的p.runnext G
            if gp := runqsteal(_p_, allp[enum.position()], stealRunNextG); gp != nil {
                return gp, false
            }
        }
    }

stop:

    // 我们没事做如果我们处于GC标记阶段，可以安全地扫描和三色法标记对象为黑色并进行工作，请运行空闲时间标记，
而不是放弃P
    // 当没有G可被执行时，M会与P解绑，然后进入休眠(idle)状态。

    if gcBlackenEnabled != 0 && _p_.gcBgMarkWorker != 0 && gcMarkWorkAvailable(_p_) {
        _p_.gcMarkWorkerMode = gcMarkWorkerIdleMode
        gp := _p_.gcBgMarkWorker.ptr()
        casgstatus(gp, _Gwaiting, _Grunnable)
        if trace.enabled {
            traceGoUnpark(gp, 0)
        }
        return gp, false
    }

    // wasm only:
```

```
    // If a callback returned and no other goroutine is awake,
    // then pause execution until a callback was triggered.
    if beforeIdle() {
        // At least one goroutine got woken.
        goto top
    }

    // Before we drop our P, make a snapshot of the allp slice,
    // which can change underfoot once we no longer block
    // safe-points. We don't need to snapshot the contents because
    // everything up to cap(allp) is immutable.
    allpSnapshot := allp

    // return P and block
    lock(&sched.lock)
    if sched.gcwaiting != 0 || _p_.runSafePointFn != 0 {
        unlock(&sched.lock)
        goto top
    }
    if sched.runqsize != 0 {
        gp := globrunqget(_p_, 0)
        unlock(&sched.lock)
        return gp, false
    }
    if releasep() != _p_ {
        throw("findrunnable: wrong p")
    }
    pidleput(_p_)
    unlock(&sched.lock)

    // Delicate dance: thread transitions from spinning to non-spinning state,
    // potentially concurrently with submission of new goroutines. We must
    // drop nmspinning first and then check all per-P queues again (with
    // #StoreLoad memory barrier in between). If we do it the other way around,
    // another thread can submit a goroutine after we've checked all run queues
    // but before we drop nmspinning; as the result nobody will unpark a thread
    // to run the goroutine.
    // If we discover new work below, we need to restore m.spinning as a signal
    // for resetspinning to unpark a new worker thread (because there can be more
    // than one starving goroutine). However, if after discovering new work
    // we also observe no idle Ps, it is OK to just park the current thread:
    // the system is fully loaded so no spinning threads are required.
    // Also see "Worker thread parking/unparking" comment at the top of the file.
    wasSpinning := _g_.m.spinning
    if _g_.m.spinning {
        _g_.m.spinning = false
        if int32(atomic.Xadd(&sched.nmspinning, -1)) < 0 {
            throw("findrunnable: negative nmspinning")
        }
    }

    // check all runqueues once again
    for _, _p_ := range allpSnapshot {
        if !runqempty(_p_) {
            lock(&sched.lock)
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                acquirep(_p_)
                if wasSpinning {
                    _g_.m.spinning = true
                    atomic.Xadd(&sched.nmspinning, 1)
```

```
                }
                goto top
            }
            break
        }
    }

    // Check for idle-priority GC work again.
    if gcBlackenEnabled != 0 && gcMarkWorkAvailable(nil) {
        lock(&sched.lock)
        _p_ = pidleget()
        if _p_ != nil && _p_.gcBgMarkWorker == 0 {
            pidleput(_p_)
            _p_ = nil
        }
        unlock(&sched.lock)
        if _p_ != nil {
            acquirep(_p_)
            if wasSpinning {
                _g_.m.spinning = true
                atomic.Xadd(&sched.nmspinning, 1)
            }
            // Go back to idle GC check.
            goto stop
        }
    }

    // poll network
    if netpollinited() && atomic.Load(&netpollWaiters) > 0 && atomic.Xchg64(&sched.lastpoll, 0) != 0 {
        if _g_.m.p != 0 {
            throw("findrunnable: netpoll with p")
        }
        if _g_.m.spinning {
            throw("findrunnable: netpoll with spinning")
        }
        list := netpoll(true) // block until new work is available
        atomic.Store64(&sched.lastpoll, uint64(nanotime()))
        if !list.empty() {
            lock(&sched.lock)
            _p_ = pidleget()
            unlock(&sched.lock)
            if _p_ != nil {
                acquirep(_p_)
                gp := list.pop()
                injectglist(&list)
                casgstatus(gp, _Gwaiting, _Grunnable)
                if trace.enabled {
                    traceGoUnpark(gp, 0)
                }
                return gp, false
            }
            injectglist(&list)
        }
    }
    stopm()
    goto top
}
```

GPM模型调度

如果一切正常，调度器会以上述的那种方式顺畅地运行，但总是有特殊的情况存在，下面分析 goroutine 在两种例外情况下的行为。

Go runtime 会在下面的 goroutine 被阻塞的情况下运行另外一个 goroutine：

然而在通常情况下，Go runtime 会在下面的 goroutine 被阻塞的情况下运行另外一个 goroutine：

- blocking syscall (for example opening a file)
- network input
- channel operations
- primitives in the sync package

这里其实可以看做两个情况,即 `用户态阻塞/唤醒` 和 `系统调用阻塞` .

- 用户态阻塞/唤醒

当 goroutine 因为 channel 操作或者 network I/O 而阻塞时（实际上 golang 已经用 netpoller 实现了 goroutine 网络 I/O 阻塞不会导致 M 被阻塞，仅阻塞 G），对应的 G 会被放置到某个 wait 队列(如 channel 的 waitq)，该 G 的状态由 `_Gruning` 变为 `_Gwaitting` ，而 M 会跳过该 G 尝试获取并执行下一个 G.如果此时没有 runnable 的 G 供 M 运行，那么 M 将解绑 P，并进入 sleep 状态.

当阻塞的 G 被另一端的 G2 唤醒时（比如 channel 的可读/写通知），G 被标记为 runnable，尝试加入 G2 所在 P 的 runnext，然后再是 P 的 Local 队列和 Global 队列。

- syscall 系统调用阻塞

当 G 被阻塞在某个系统调用上时，此时 G 会阻塞在_Gsyscall状态，M 也处于 `block on syscall` 状态，此时的 M 可被抢占调度：执行该 G 的 M 会与 P 解绑，而 P 则尝试与其它 idle 的 M 绑定，继续执行其它 G。如果没有其它 idle 的 M，但 P 的 Local 队列中仍然有 G 需要执行，则创建一个新的 M.

当系统调用完成后，G 会重新尝试获取一个 idle 的 P 进入它的 Local 队列恢复执行，如果没有 idle 的 P，G 会被标记为 `runnable` 加入到 Global 队列。

系统调用能被调度的关键有两点:

`runtime/syscall` 包中，将系统调用分为 `SysCall` 和 `RawSysCall` ， `SysCall` 和 `RawSysCall` 的区别是 `SysCall` 会在系统调用前后分别调用 `entersyscall` 和 `exitsyscall` (位于 `src/runtime/proc.go`)，做一些现场保存和恢复操作，这样才能使P安全地与M解绑，并在其它M上继续执行其它G。

某些系统调用本身可以确定会长时间阻塞(比如锁)，会调用 `entersyscallblock` 在发起系统调用前直接让P和M解绑。

这里的关键点是 `sysmon` ，它负责检查所有系统调用的执行时间，判断是否需要解绑。

`sysmon` 是一个由 `runtime` 启动的M，也叫监控线程，它无需P也可以运行，它每20us~10ms唤醒一次，主要执行:

1. 释放闲置超过5分钟的span物理内存；
2. 如果超过2分钟没有垃圾回收，强制执行；
3. 将长时间未处理的netpoll结果添加到任务队列；
4. 向长时间运行的G任务发出抢占调度；
5. 收回因syscall长时间阻塞的P；

`sysmon` 它通过retake实现对syscall和长时间运行的G进行调度:

```
// src/runtime/proc.go:sysmon

type sysmontick struct {
    schedtick   uint32
```

```
    schedwhen   int64
    syscalltick uint32
    syscallwhen int64
}

// forcePreemptNS is the time slice given to a G before it is
// preempted.
const forcePreemptNS = 10 * 1000 * 1000 // 10ms

func retake(now int64) uint32 {
    n := 0
    // Prevent allp slice changes. This lock will be completely
    // uncontended unless we're already stopping the world.
    lock(&allpLock)
    // We can't use a range loop over allp because we may
    // temporarily drop the allpLock. Hence, we need to re-fetch
    // allp each time around the loop.
    for i := 0; i < len(allp); i++ {
        _p_ := allp[i]
        if _p_ == nil {
            // This can happen if procresize has grown
            // allp but not yet created new Ps.
            continue
        }
        pd := &_p_.sysmontick
        s := _p_.status
        sysretake := false
        if s == _Prunning || s == _Psyscall {
            // Preempt G if it's running for too long.
            t := int64(_p_.schedtick)
            if int64(pd.schedtick) != t {
                pd.schedtick = uint32(t)
                pd.schedwhen = now
            } else if pd.schedwhen+forcePreemptNS <= now {
                // 如果当前G执行时间超过10ms，则抢占(preemptone)
                // 执行抢占

                preemptone(_p_)
                // In case of syscall, preemptone() doesn't
                // work, because there is no M wired to P.
                sysretake = true
            }
        }
        if s == _Psyscall {
            // Retake P from syscall if it's there for more than 1 sysmon tick (at least 20us).
            t := int64(_p_.syscalltick)
            if !sysretake && int64(pd.syscalltick) != t {
                pd.syscalltick = uint32(t)
                pd.syscallwhen = now
                continue
            }
            // 如果当前P Local队列没有其它G，当前有其它P处于Idle状态，并且syscall执行事件不超过10ms，则不用解绑当前P(handoffp)
            if runqempty(_p_) && atomic.Load(&sched.nmspinning)+atomic.Load(&sched.npidle) > 0 && pd.syscallwhen+10*1000*1000 > now {
                continue
            }
            // Drop allpLock so we can take sched.lock.
            unlock(&allpLock)
            // Need to decrement number of idle locked M's
            // (pretending that one more is running) before the CAS.
            // Otherwise the M from which we retake can exit the syscall,
```

```
            // increment nmidle and report deadlock.

            incidlelocked(-1)
            if atomic.Cas(&_p_.status, s, _Pidle) {
                if trace.enabled {
                    traceGoSysBlock(_p_)
                    traceProcStop(_p_)
                }
                n++
                _p_.syscalltick++
                handoffp(_p_)
            }
            incidlelocked(1)
            lock(&allpLock)
        }
    }
    unlock(&allpLock)
    return uint32(n)
}
```

抢占式调度

当某个goroutine执行超过10ms， `sysmon` 会向其发起抢占调度请求，由于Go调度不像OS调度那样有时间片的概念，因此实际抢占机制要弱很多: Go中的抢占实际上是为G设置抢占标记(g.stackguard0)，当G调用某函数时(更确切说，在通过newstack分配函数栈时)，被编译器安插的指令会检查这个标记，并且将当前G以 `runtime.Gosched` 的方式暂停，并加入到全局队列。

源代码如下:

```
// Called from runtime·morestack when more stack is needed.
// Allocate larger stack and relocate to new stack.
// Stack growth is multiplicative, for constant amortized cost.
//
// g->atomicstatus will be Grunning or Gscanrunning upon entry.
// If the GC is trying to stop this g then it will set preemptscan to true.
//
// ctxt is the value of the context register on morestack. newstack
// will write it to g.sched.ctxt.

func newstack() {
    thisg := getg()
    // TODO: double check all gp. shouldn't be getg().
    if thisg.m.morebuf.g.ptr().stackguard0 == stackFork {
        throw("stack growth after fork")
    }
    if thisg.m.morebuf.g.ptr() != thisg.m.curg {
        print("runtime: newstack called from g=", hex(thisg.m.morebuf.g), "\n"+"\tm=", thisg.m, " m->curg=",
 thisg.m.curg, " m->g0=", thisg.m.g0, " m->gsignal=", thisg.m.gsignal, "\n")
        morebuf := thisg.m.morebuf
        traceback(morebuf.pc, morebuf.sp, morebuf.lr, morebuf.g.ptr())
        throw("runtime: wrong goroutine in newstack")
    }

    gp := thisg.m.curg

    if thisg.m.curg.throwsplit {
        // Update syscallsp, syscallpc in case traceback uses them.
        morebuf := thisg.m.morebuf
        gp.syscallsp = morebuf.sp
        gp.syscallpc = morebuf.pc
```

```
        pcname, pcoff := "(unknown)", uintptr(0)
        f := findfunc(gp.sched.pc)
        if f.valid() {
            pcname = funcname(f)
            pcoff = gp.sched.pc - f.entry
        }
        print("runtime: newstack at ", pcname, "+", hex(pcoff),
            " sp=", hex(gp.sched.sp), " stack=[", hex(gp.stack.lo), ", ", hex(gp.stack.hi), "]\n",
            "\tmorebuf={pc:", hex(morebuf.pc), " sp:", hex(morebuf.sp), " lr:", hex(morebuf.lr), "}\n",
            "\tsched={pc:", hex(gp.sched.pc), " sp:", hex(gp.sched.sp), " lr:", hex(gp.sched.lr), " ctxt:", g
p.sched.ctxt, "}\n")

        thisg.m.traceback = 2 // Include runtime frames
        traceback(morebuf.pc, morebuf.sp, morebuf.lr, gp)
        throw("runtime: stack split at bad time")
    }

    morebuf := thisg.m.morebuf
    thisg.m.morebuf.pc = 0
    thisg.m.morebuf.lr = 0
    thisg.m.morebuf.sp = 0
    thisg.m.morebuf.g = 0

    // NOTE: stackguard0 may change underfoot, if another thread
    // is about to try to preempt gp. Read it just once and use that same
    // value now and below.
    preempt := atomic.Loaduintptr(&gp.stackguard0) == stackPreempt

    // Be conservative about where we preempt.
    // We are interested in preempting user Go code, not runtime code.
    // If we're holding locks, mallocing, or preemption is disabled, don't
    // preempt.
    // This check is very early in newstack so that even the status change
    // from Grunning to Gwaiting and back doesn't happen in this case.
    // That status change by itself can be viewed as a small preemption,
    // because the GC might change Gwaiting to Gscanwaiting, and then
    // this goroutine has to wait for the GC to finish before continuing.
    // If the GC is in some way dependent on this goroutine (for example,
    // it needs a lock held by the goroutine), that small preemption turns
    // into a real deadlock.
    if preempt {
        if thisg.m.locks != 0 || thisg.m.mallocing != 0 || thisg.m.preemptoff != "" || thisg.m.p.ptr().status
!= _Prunning {
            // Let the goroutine keep running for now.
            // gp->preempt is set, so it will be preempted next time.
            gp.stackguard0 = gp.stack.lo + _StackGuard
            gogo(&gp.sched) // never return
        }
    }

    if gp.stack.lo == 0 {
        throw("missing stack in newstack")
    }
    sp := gp.sched.sp
    if sys.ArchFamily == sys.AMD64 || sys.ArchFamily == sys.I386 || sys.ArchFamily == sys.WASM {
        // The call to morestack cost a word.
        sp -= sys.PtrSize
    }
    if stackDebug >= 1 || sp < gp.stack.lo {
        print("runtime: newstack sp=", hex(sp), " stack=[", hex(gp.stack.lo), ", ", hex(gp.stack.hi), "]\n",
            "\tmorebuf={pc:", hex(morebuf.pc), " sp:", hex(morebuf.sp), " lr:", hex(morebuf.lr), "}\n",
            "\tsched={pc:", hex(gp.sched.pc), " sp:", hex(gp.sched.sp), " lr:", hex(gp.sched.lr), " ctxt:", g
```

```
p.sched.ctxt, "}\n")
    }
    if sp < gp.stack.lo {
        print("runtime: gp=", gp, ", goid=", gp.goid, ", gp->status=", hex(readgstatus(gp)), "\n ")
        print("runtime: split stack overflow: ", hex(sp), " < ", hex(gp.stack.lo), "\n")
        throw("runtime: split stack overflow")
    }

    if preempt {
        if gp == thisg.m.g0 {
            throw("runtime: preempt g0")
        }
        if thisg.m.p == 0 && thisg.m.locks == 0 {
            throw("runtime: g is running but p is not")
        }
        // Synchronize with scang.
        casgstatus(gp, _Grunning, _Gwaiting)
        if gp.preemptscan {
            for !castogscanstatus(gp, _Gwaiting, _Gscanwaiting) {
                // Likely to be racing with the GC as
                // it sees a _Gwaiting and does the
                // stack scan. If so, gcworkdone will
                // be set and gcphasework will simply
                // return.
            }
            if !gp.gcscandone {
                // gcw is safe because we're on the
                // system stack.
                gcw := &gp.m.p.ptr().gcw
                scanstack(gp, gcw)
                gp.gcscandone = true
            }
            gp.preemptscan = false
            gp.preempt = false
            casfrom_Gscanstatus(gp, _Gscanwaiting, _Gwaiting)
            // This clears gcscanvalid.
            casgstatus(gp, _Gwaiting, _Grunning)
            gp.stackguard0 = gp.stack.lo + _StackGuard
            gogo(&gp.sched) // never return
        }

        // Act like goroutine called runtime.Gosched.
        casgstatus(gp, _Gwaiting, _Grunning)
        gopreempt_m(gp) // never return
    }

    // Allocate a bigger segment and move the stack.
    // 扩容至现在的2倍
    oldsize := gp.stack.hi - gp.stack.lo
    newsize := oldsize * 2
    if newsize > maxstacksize {
        print("runtime: goroutine stack exceeds ", maxstacksize, "-byte limit\n")
        throw("stack overflow")
    }

    // The goroutine must be executing in order to call newstack,
    // so it must be Grunning (or Gscanrunning).
    casgstatus(gp, _Grunning, _Gcopystack)

    // The concurrent GC will not scan the stack while we are doing the copy since
    // the gp is in a Gcopystack status.
    // 拷贝栈数据后切换到新栈
```

```
    copystack(gp, newsize, true)
    if stackDebug >= 1 {
        print("stack grow done\n")
    }

    // 恢复执行
    casgstatus(gp, _Gcopystack, _Grunning)
    gogo(&gp.sched)
}

// Copies gp's stack to a new stack of a different size.
// Caller must have changed gp status to Gcopystack.
//
// If sync is true, this is a self-triggered stack growth and, in
// particular, no other G may be writing to gp's stack (e.g., via a
// channel operation). If sync is false, copystack protects against
// concurrent channel operations.
func copystack(gp *g, newsize uintptr, sync bool) {
    if gp.syscallsp != 0 {
        throw("stack growth not allowed in system call")
    }
    old := gp.stack
    if old.lo == 0 {
        throw("nil stackbase")
    }
    used := old.hi - gp.sched.sp

    // allocate new stack
    // 从缓存或堆分配新栈
    new := stackalloc(uint32(newsize))
    if stackPoisonCopy != 0 {
        fillstack(new, 0xfd)
    }
    if stackDebug >= 1 {
        print("copystack gp=", gp, " [", hex(old.lo), " ", hex(old.hi-used), " ", hex(old.hi), "]", " -> [",
 hex(new.lo), " ", hex(new.hi-used), " ", hex(new.hi), "]/", newsize, "\n")
    }

    // Compute adjustment.
    var adjinfo adjustinfo
    adjinfo.old = old
    adjinfo.delta = new.hi - old.hi

    // Adjust sudogs, synchronizing with channel ops if necessary.
    ncopy := used
    if sync {
        adjustsudogs(gp, &adjinfo)
    } else {
        // sudogs can point in to the stack. During concurrent
        // shrinking, these areas may be written to. Find the
        // highest such pointer so we can handle everything
        // there and below carefully. (This shouldn't be far
        // from the bottom of the stack, so there's little
        // cost in handling everything below it carefully.)
        adjinfo.sghi = findsghi(gp, old)

        // Synchronize with channel ops and copy the part of
        // the stack they may interact with.
        ncopy -= syncadjustsudogs(gp, used, &adjinfo)
    }

    // Copy the stack (or the rest of it) to the new location
```

```
    // 拷贝栈到新的位置
    memmove(unsafe.Pointer(new.hi-ncopy), unsafe.Pointer(old.hi-ncopy), ncopy)

    // Adjust remaining structures that have pointers into stacks.
    // We have to do most of these before we traceback the new
    // stack because gentraceback uses them.
    adjustctxt(gp, &adjinfo)
    adjustdefers(gp, &adjinfo)
    adjustpanics(gp, &adjinfo)
    if adjinfo.sghi != 0 {
        adjinfo.sghi += adjinfo.delta
    }

    // Swap out old stack for new one
    // 切换到新栈
    gp.stack = new
    gp.stackguard0 = new.lo + _StackGuard // NOTE: might clobber a preempt request
    gp.sched.sp = new.hi - used
    gp.stktopsp += adjinfo.delta

    // Adjust pointers in the new stack.
    gentraceback(^uintptr(0), ^uintptr(0), 0, gp, 0, nil, 0x7fffffff, adjustframe, noescape(unsafe.Pointer(&a
djinfo)), 0)

    // free old stack
    // 释放旧栈
    if stackPoisonCopy != 0 {
        fillstack(old, 0xfc)
    }
    stackfree(old)
}
```

go在1.3之前栈扩容采用的是分段栈（Segemented Stack），在栈空间不够的时候新申请一个栈空间用于被调用函数的执行， 执行后销毁新申请的栈空间并回到老的栈空间继续执行，当函数出现频繁调用（递归）时可能会引发hot split。

为了避免hot split, 1.3之后采用的是连续栈（Contiguous Stack），栈空间不足的时候申请一个2倍于当前大小的新栈，并把所有数据拷贝到新栈， 接下来的所有调用执行都发生在新栈上。

看完了扩容，我们来看看缩容。一些 `long running` 的goroutine可能由于某次函数调用中引发了栈的扩容， 被调用函数返回后很大部分空间都未被利用，为了解决这样的问题，需要能够对栈进行收缩，以节约内存提高利用率。

栈收缩不是在函数调用时发生的，是由垃圾回收器在垃圾回收时主动触发的。基本过程是计算当前使用的空间，小于栈空间的1/4的话， 执行栈的收缩，将栈收缩为现在的1/2，否则直接返回。

```
// runtime/stack.go
// Maybe shrink the stack being used by gp.
// Called at garbage collection time.
// gp must be stopped, but the world need not be.
func shrinkstack(gp *g) {
    gstatus := readgstatus(gp)
    if gp.stack.lo == 0 {
        throw("missing stack in shrinkstack")
    }
    if gstatus&_Gscan == 0 {
        throw("bad status in shrinkstack")
    }

    if debug.gcshrinkstackoff > 0 {
        return
    }
```

```
    f := findfunc(gp.startpc)
    if f.valid() && f.funcID == funcID_gcBgMarkWorker {
        // We're not allowed to shrink the gcBgMarkWorker
        // stack (see gcBgMarkWorker for explanation).
        return
    }

    // 收缩目标是一半大小
    oldsize := gp.stack.hi - gp.stack.lo
    newsize := oldsize / 2
    // Don't shrink the allocation below the minimum-sized stack
    // allocation.
    if newsize < _FixedStack {
        return
    }
    // Compute how much of the stack is currently in use and only
    // shrink the stack if gp is using less than a quarter of its
    // current stack. The currently used stack includes everything
    // down to the SP plus the stack guard space that ensures
    // there's room for nosplit functions.
    // 如果使用空间超过1/4，则不收缩
    avail := gp.stack.hi - gp.stack.lo
    if used := gp.stack.hi - gp.sched.sp + _StackLimit; used >= avail/4 {
        return
    }

    // We can't copy the stack if we're in a syscall.
    // The syscall might have pointers into the stack.
    if gp.syscallsp != 0 {
        return
    }
    if sys.GoosWindows != 0 && gp.m != nil && gp.m.libcallsp != 0 {
        return
    }

    if stackDebug > 0 {
        print("shrinking stack ", oldsize, "->", newsize, "\n")
    }

    // 用较小的栈替换当前的栈
    copystack(gp, newsize, false)
}
```

这里只是对Go的调度器进行一个分析，当然，Go 的调度中更复杂的抢占式调度、阻塞调度的更多细节，大家可以自行去找相关资料深入理解，这里只讲到 Go 调度器的基本调度过程，所以想了解更多细节的同学可以去看看 Go 调度器 G-P-M 模型的设计者 Dmitry Vyukov 写的该模型的设计文档《Go Preemptive Scheduler Design》 以及直接去看源码，G-P-M 模型的定义放在 `src/runtime/runtime2.go` 里面，而调度过程则放在了 `src/runtime/proc.go` 里。

在Go的最新1.14源码中优化了调度器,后续我们继续分析.

资料参考

- Go调度模型
- The Go scheduler
- work-stealing

Golang的逃逸分析

Golang的逃逸分析

所谓逃逸分析（Escape analysis）是指由编译器决定内存分配的位置，不需要程序员指定。 函数中申请一个新的对象如果分配在栈中，则函数执行结束可自动将内存回收；如果分配在堆中，则函数执行结束可交给GC（垃圾回收）处理.

每当函数中申请新的对象，编译器会跟据该对象是否被函数外部引用来决定是否逃逸： 1. 如果函数外部没有引用，则优先放到栈中； 2. 如果函数外部存在引用，则必定放到堆中.

注意，对于函数外部没有引用的对象，也有可能放到堆中，比如内存过大超过栈的存储能力。

逃逸分析通常有四种情况:

- 指针逃逸
- 栈空间不足逃逸
- 动态类型逃逸
- 闭包引用对象逃逸

逃逸总结

- 栈上分配内存比在堆中分配内存有更高的效率.
- 栈上分配的内存不需要GC处理.
- 堆上分配的内存使用完毕会交给GC处理.
- 逃逸分析目的是决定内分配地址是栈还是堆.
- 逃逸分析在编译阶段完成.

Redis为什么快

Redis为什么快

Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器的内存的大小和网络的带宽，而且单线程的性能已经非常高了，就没有必要使用多线程了，所以 Redis 是单进程单线程的。 提示：

如果我们运行的服务器是多核服务器，为了充分利用多核优势我们可以在单台服务器起多个 Redis 服务，或者架设 主从复制、哨兵模式、集群模式等多机方案。

Redis 服务运行时只是处理客户端请求是单进程单线程的，但是服务运行时会有其他进程或线程处理其他的事，比如RDB的文件的生成就会在子进程中进行等。

Redis为什么这么快?

1. 完全基于内存，绝大部分请求是基于内存的操作，而 Redis 的数据结构是类似于HashMap，而 HashMap 的操作时间复杂度是O(1)
2. Redis 数据结构设计简单，方便操作
3. 使用单线程，避免了进程或线程的上下文切换相关的消耗，不用考虑锁相关问题消耗。
4. 使用多路I/O复用模型，非阻塞IO
5. 使用底层模型不同，底层实现方式以及与客户端之间通信的应用协议不一样，Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求

什么是多路I/O复用

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量。

Redis的数据过期策略

Redis 中数据过期策略采用定期删除和惰性删除策略:

- 定期删除策略：Redis 启用一个定时器定时监视所有的 key，判断key是否过期，过期的话就删除。这种策略可以保证过期的 key 最终都会被删除，但是也存在严重的缺点：每次都遍历内存中所有的数据，非常消耗 CPU 资源，并且当 key 已过期，但是定时器还处于未唤起状态，这段时间内 key 仍然可以用。

- 惰性删除策略：在获取 key 时，先判断 key 是否过期，如果过期则删除。这种方式存在一个缺点：如果这个 key 一直未被使用，那么它一直在内存中，其实它已经过期了，会浪费大量的空间。

这两种策略天然的互补，结合起来之后，定时删除策略就发生了一些改变，不在是每次扫描全部的 key 了，而是随机抽取一部分 key 进行检查，这样就降低了对 CPU 资源的损耗，惰性删除策略互补了为检查到的key，基本上满足了所有要求。但是有时候就是那么的巧，既没有被定时器抽取到，又没有被使用，这些数据又如何从内存中消失？没关系，还有内存淘汰机制，当内存不够用时，内存淘汰机制就会上场。

淘汰策略分为：

1. 当内存不足以容纳新写入数据时，新写入操作会报错。（Redis 默认策略）

2. 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 Key。（LRU推荐使用）

3. 当内存不足以容纳新写入数据时，在键空间中，随机移除某个 Key。

4. 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 Key。这种情况一般是把 Redis 既当缓存，又做持久化存储的时候才用。

5. 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个 Key。

6. 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 Key 优先移除。

注意事项

Redis是基于I/O多路复用的单线程模式，所以 Redis 在处理比较耗时的命令的时候性能会受影响。可以使用 Redis 多机部署方案来应对这样的问题

Golang性能优化

Golang的汇编过程

Golang的汇编过程

在程序编译的时候,汇编的目的是把汇编代码转化为机器指令，因为几乎每一条汇编指令都对应着一条机器指令，所以汇编的过程相对而言非常的简单。
汇编操作所生成的文件叫做目标文件（Object File），目标文件的结构与可执行文件是一致的，它们之间只存在着一些细微的差异。目标文件是无法被执行的，它还需要经过链接这一步操作，目标文件被链接之后才可以产生可执行文件。

Golang原生支持用户级协程，交叉编译，跨平台部署运行, 但是go在编译成机器语言交付给CPU执行的过程中，汇编也只是一个中间状态，汇编指令相对于以上的高级语言而言则显得十分拗口。在大部分强类型的语言中，基本上代码在执行前会经历几个阶段：

```
语法分析--->词法分析--->目标码生成
```

Go的汇编是怎么样的？

Go汇编器所用的指令，一部分与目标机器的指令 一一对应，而另外一部分则不是。这是因为编译器套件不需要汇编器直接参与常规的编译过程。相反，编译器使用了一种半抽象的指令集，并且部分指令是在代码生成后才被选择的。汇编器基于这种半抽象的形式工作，所以虽然你看到的是一条MOV指令，但是工具链针对这条指令实际生成可能完全不是一个移动指令，也许会是清除或者加载。也有可能精确的对应目标平台上同名的指令。
由于这种汇编并不对应某种真实的硬件架构，Go编译器会输出一种抽象可移植的汇编代码。

Golang的defer优化

Golang 中的defer性能提升

在Golang 1.14中新加入了开放编码（Open-coded） `defer` 类型，编译器在ssa过程中会把被延迟的方法直接插入到函数的尾部，避免了运行时的deferproc及 `deferprocStack` 操作。

避免了在没有运行时判断下的 `deferreturn` 调用。如有运行时判断的逻辑，则 `deferreturn` 也进一步优化，开放编码下的 `deferreturn` 不会进行 `jmpdefer` 的尾递归调用，而直接在一个循环里遍历执行。

在1.14中defer的实现原理,共有三种defer模式类型，编译后一个函数里只会一种defer模式。

堆上分配

在 Golang 1.13 之前的版本中，所有 defer 都是在堆上分配 (deferProc)，该机制在编译时会进行两个步骤：

1. 在 defer 语句的位置插入 `runtime.deferproc` ，当被执行时，延迟调用会被保存为一个 `_defer` 记录，并将被延迟调用的入口地址及其参数复制保存，存入 Goroutine 的调用链表中。

2. 在函数返回之前的位置插入 `runtime.deferreturn` ，当被执行时，会将延迟调用从 Goroutine 链表中取出并执行，多个延迟调用则以 `jmpdefer` 尾递归调用方式连续执行。

这种机制的主要性能问题存在于每个 defer 语句产生记录时的内存分配，以及记录参数和完成调用时参数移动的系统调用开销。

栈上分配

在Golang 1.13 版本中新加入 `deferprocStack` 实现了在栈上分配的形式来取代 `deferproc` ，相比后者，栈上分配在函数返回后 `_defer` 便得到释放，省去了内存分配时产生的性能开销，只需适当维护 `_defer` 的链表即可。

编译器可以去选择使用 `deferproc` 还是 `deferprocStack` ，通常情况下都会使用 `deferprocStack` ，性能会提升约 30%。不过在 defer 语句出现在了循环语句里，或者无法执行更高阶的编译器优化时，亦或者同一个函数中使用了过多的 defer 时，依然会使用 `deferproc` 。

栈上分配 (deferprocStack)，基本跟堆上差不多，只是分配方式改为在栈上分配，压入的函数调用栈存有 `_defer` 记录，另外编译器在ssa过程中会预留defer空间。

SSA 代表 `static single-assignment` ，是一种IR(中间表示代码)，要保证每个变量只被赋值一次。这个能帮助简化编译器的优化算法。简单来说,使用ssa可以使二进制文件大小减少了30%，性能提升5%-35%等.

```
// buildssa builds an SSA function for fn.
// worker indicates which of the backend workers is doing the processing.
func buildssa(fn *Node, worker int) *ssa.Func {
    name := fn.funcname()
    printssa := name == ssaDump
    var astBuf *bytes.Buffer
    if printssa {
        astBuf = &bytes.Buffer{}
        fdumplist(astBuf, "buildssa-enter", fn.Func.Enter)
        fdumplist(astBuf, "buildssa-body", fn.Nbody)
        fdumplist(astBuf, "buildssa-exit", fn.Func.Exit)
        if ssaDumpStdout {
            fmt.Println("generating SSA for", name)
            fmt.Print(astBuf.String())
        }
    }
```

```
    var s state
    s.pushLine(fn.Pos)
    defer s.popLine()

    s.hasdefer = fn.Func.HasDefer()
    if fn.Func.Pragma&CgoUnsafeArgs != 0 {
        s.cgoUnsafeArgs = true
    }

    fe := ssafn{
        curfn: fn,
        log:   printssa && ssaDumpStdout,
    }
    s.curfn = fn

    s.f = ssa.NewFunc(&fe)
    s.config = ssaConfig
    s.f.Type = fn.Type
    s.f.Config = ssaConfig
    s.f.Cache = &ssaCaches[worker]
    s.f.Cache.Reset()
    s.f.DebugTest = s.f.DebugHashMatch("GOSSAHASH", name)
    s.f.Name = name
    s.f.PrintOrHtmlSSA = printssa
    if fn.Func.Pragma&Nosplit != 0 {
        s.f.NoSplit = true
    }
    s.panics = map[funcLine]*ssa.Block{}
    s.softFloat = s.config.SoftFloat

    if printssa {
        s.f.HTMLWriter = ssa.NewHTMLWriter(ssaDumpFile, s.f.Frontend(), name, ssaDumpCFG)
        // TODO: generate and print a mapping from nodes to values and blocks
        dumpSourcesColumn(s.f.HTMLWriter, fn)
        s.f.HTMLWriter.WriteAST("AST", astBuf)
    }

    // Allocate starting block
    s.f.Entry = s.f.NewBlock(ssa.BlockPlain)

    // Allocate starting values
    s.labels = map[string]*ssaLabel{}
    s.labeledNodes = map[*Node]*ssaLabel{}
    s.fwdVars = map[*Node]*ssa.Value{}
    s.startmem = s.entryNewValue0(ssa.OpInitMem, types.TypeMem)

    s.hasOpenDefers = Debug['N'] == 0 && s.hasdefer && !s.curfn.Func.OpenCodedDeferDisallowed()
    switch {
    case s.hasOpenDefers && (Ctxt.Flag_shared || Ctxt.Flag_dynlink) && thearch.LinkArch.Name == "386":
        // Don't support open-coded defers for 386 ONLY when using shared
        // libraries, because there is extra code (added by rewriteToUseGot())
        // preceding the deferreturn/ret code that is generated by gencallret()
        // that we don't track correctly.
        s.hasOpenDefers = false
    }
    if s.hasOpenDefers && s.curfn.Func.Exit.Len() > 0 {
        // Skip doing open defers if there is any extra exit code (likely
        // copying heap-allocated return values or race detection), since
        // we will not generate that code in the case of the extra
        // deferreturn/ret segment.
        s.hasOpenDefers = false
    }
```

```
    if s.hasOpenDefers &&
        s.curfn.Func.numReturns*s.curfn.Func.numDefers > 15 {
        // Since we are generating defer calls at every exit for
        // open-coded defers, skip doing open-coded defers if there are
        // too many returns (especially if there are multiple defers).
        // Open-coded defers are most important for improving performance
        // for smaller functions (which don't have many returns).
        s.hasOpenDefers = false
    }

    s.sp = s.entryNewValue0(ssa.OpSP, types.Types[TUINTPTR]) // TODO: use generic pointer type (unsafe.Pointer?) instead
    s.sb = s.entryNewValue0(ssa.OpSB, types.Types[TUINTPTR])

    s.startBlock(s.f.Entry)
    s.vars[&memVar] = s.startmem
    if s.hasOpenDefers {
        // Create the deferBits variable and stack slot.  deferBits is a
        // bitmask showing which of the open-coded defers in this function
        // have been activated.
        deferBitsTemp := tempAt(src.NoXPos, s.curfn, types.Types[TUINT8])
        s.deferBitsTemp = deferBitsTemp
        // For this value, AuxInt is initialized to zero by default
        startDeferBits := s.entryNewValue0(ssa.OpConst8, types.Types[TUINT8])
        s.vars[&deferBitsVar] = startDeferBits
        s.deferBitsAddr = s.addr(deferBitsTemp, false)
        s.store(types.Types[TUINT8], s.deferBitsAddr, startDeferBits)
        // Make sure that the deferBits stack slot is kept alive (for use
        // by panics) and stores to deferBits are not eliminated, even if
        // all checking code on deferBits in the function exit can be
        // eliminated, because the defer statements were all
        // unconditional.
        s.vars[&memVar] = s.newValue1Apos(ssa.OpVarLive, types.TypeMem, deferBitsTemp, s.mem(), false)
    }

    // Generate addresses of local declarations
    s.decladdrs = map[*Node]*ssa.Value{}
    for _, n := range fn.Func.Dcl {
        switch n.Class() {
        case PPARAM, PPARAMOUT:
            s.decladdrs[n] = s.entryNewValue2A(ssa.OpLocalAddr, types.NewPtr(n.Type), n, s.sp, s.startmem)
            if n.Class() == PPARAMOUT && s.canSSA(n) {
                // Save ssa-able PPARAMOUT variables so we can
                // store them back to the stack at the end of
                // the function.
                s.returns = append(s.returns, n)
            }
        case PAUTO:
            // processed at each use, to prevent Addr coming
            // before the decl.
        case PAUTOHEAP:
            // moved to heap - already handled by frontend
        case PFUNC:
            // local function - already handled by frontend
        default:
            s.Fatalf("local variable with class %v unimplemented", n.Class())
        }
    }

    // Populate SSAable arguments.
    for _, n := range fn.Func.Dcl {
        if n.Class() == PPARAM && s.canSSA(n) {
```

```
            v := s.newValue0A(ssa.OpArg, n.Type, n)
            s.vars[n] = v
            s.addNamedValue(n, v) // This helps with debugging information, not needed for compilation itsel
f.
        }
    }

    // Convert the AST-based IR to the SSA-based IR
    s.stmtList(fn.Func.Enter)
    s.stmtList(fn.Nbody)

    // fallthrough to exit
    if s.curBlock != nil {
        s.pushLine(fn.Func.Endlineno)
        s.exit()
        s.popLine()
    }

    for _, b := range s.f.Blocks {
        if b.Pos != src.NoXPos {
            s.updateUnsetPredPos(b)
        }
    }

    s.insertPhis()

    // Main call to ssa package to compile function
    ssa.Compile(s.f)

    if s.hasOpenDefers {
        s.emitOpenDeferInfo()
    }

    return s.f
}
```

如果在构建ssa时如发现 `gcflags` 有N禁止优化的参数 或者 `return数量 * defer数量` 超过了15不适用 `open-coded` 模式。

此外逃逸分析会判断循序的层数，如果有轮询，那么强制使用栈分配模式。

```
// augmentParamHole augments parameter holes as necessary for use in
// go/defer statements.
func (e *Escape) augmentParamHole(k EscHole, call, where *Node) EscHole {
    k = k.note(call, "call parameter")
    if where == nil {
        return k
    }

    // Top level defers arguments don't escape to heap, but they
    // do need to last until end of function. Tee with a
    // non-transient location to avoid arguments from being
    // transiently allocated.
    if where.Op == ODEFER && e.loopDepth == 1 {
        // force stack allocation of defer record, unless open-coded
        // defers are used (see ssa.go)
        where.Esc = EscNever
        return e.later(k)
    }
```

```
    return e.heapHole().note(where, "call parameter")
}
```

开放编码

Golang 1.14 版本继续加入了开发编码（open coded），该机制会将延迟调用直接插入函数返回之前，省去了运行时的 `deferproc` 或 `deferprocStack` 操作，在运行时的 deferreturn 也不会进行尾递归调用，而是直接在一个循环中遍历所有延迟函数执行。

这种机制使得 defer 的开销几乎可以忽略，唯一的运行时成本就是存储参与延迟调用的相关信息，不过使用这个机制还需要三个条件：

1. 没有禁用编译器优化，即没有设置 -gcflags “-N”.
2. 函数内 defer 的数量不超过 8 个，且返回语句与延迟语句个数的乘积不超过 15.
3. defer 不是在循环语句中。

此外该机制还引入了一种元素 —— 延迟比特（defer bit）,用于运行时记录每个 defer 是否被执行（尤其是在条件判断分支中的 defer），从而便于判断最后的延迟调用该执行哪些函数。

延迟比特的原理：

同一个函数内每出现一个 defer 都会为其分配 1个比特，如果被执行到则设为 1，否则设为 0，当到达函数返回之前需要判断延迟调用时，则用掩码判断每个位置的比特，若为 1 则调用延迟函数，否则跳过。

为了轻量，官方将延迟比特限制为 1 个字节，即 8 个比特，这就是为什么不能超过 8 个 defer 的原因，若超过依然会选择堆栈分配，但显然大部分情况不会超过 8 个。

```
// The constant is known to runtime.
const tmpstringbufsize = 32
const zeroValSize = 1024 // must match value of runtime/map.go:maxZero

func walk(fn *Node) {
    Curfn = fn

    if Debug['W'] != 0 {
        s := fmt.Sprintf("\nbefore walk %v", Curfn.Func.Nname.Sym)
        dumplist(s, Curfn.Nbody)
    }

    lno := lineno

    // Final typecheck for any unused variables.
    for i, ln := range fn.Func.Dcl {
        if ln.Op == ONAME && (ln.Class() == PAUTO || ln.Class() == PAUTOHEAP) {
            ln = typecheck(ln, ctxExpr|ctxAssign)
            fn.Func.Dcl[i] = ln
        }
    }

    // Propagate the used flag for typeswitch variables up to the NONAME in its definition.
    for _, ln := range fn.Func.Dcl {
        if ln.Op == ONAME && (ln.Class() == PAUTO || ln.Class() == PAUTOHEAP) && ln.Name.Defn != nil && ln.Na
me.Defn.Op == OTYPESW && ln.Name.Used() {
            ln.Name.Defn.Left.Name.SetUsed(true)
        }
    }

    for _, ln := range fn.Func.Dcl {
        if ln.Op != ONAME || (ln.Class() != PAUTO && ln.Class() != PAUTOHEAP) || ln.Sym.Name[0] == '&' || ln.
Name.Used() {
```

```
            continue
        }
        if defn := ln.Name.Defn; defn != nil && defn.Op == OTYPESW {
            if defn.Left.Name.Used() {
                continue
            }
            yyerrorl(defn.Left.Pos, "%v declared but not used", ln.Sym)
            defn.Left.Name.SetUsed(true) // suppress repeats
        } else {
            yyerrorl(ln.Pos, "%v declared but not used", ln.Sym)
        }
    }

    lineno = lno
    if nerrors != 0 {
        return
    }
    walkstmtlist(Curfn.Nbody.Slice())
    if Debug['W'] != 0 {
        s := fmt.Sprintf("after walk %v", Curfn.Func.Nname.Sym)
        dumplist(s, Curfn.Nbody)
    }

    zeroResults()
    heapmoves()
    if Debug['W'] != 0 && Curfn.Func.Enter.Len() > 0 {
        s := fmt.Sprintf("enter %v", Curfn.Func.Nname.Sym)
        dumplist(s, Curfn.Func.Enter)
    }
}
```

在使用open code的模式的时候,默认open coded最多支持8个defer，超过则取消。

```
const maxOpenDefers = 8

func walkstmt(n *Node) *Node {
    ...
    switch n.Op {
    case ODEFER:
        Curfn.Func.SetHasDefer(true)
        Curfn.Func.numDefers++
        if Curfn.Func.numDefers > maxOpenDefers {
            Curfn.Func.SetOpenCodedDeferDisallowed(true)
        }

        if n.Esc != EscNever {
            Curfn.Func.SetOpenCodedDeferDisallowed(true)
        }
    ...
}
```

因此 `open coded` 的使用条件是,最多8个defer，而且 `return * defer < 15` ，无循环， `gcflags` 无“N”并且取消优化。