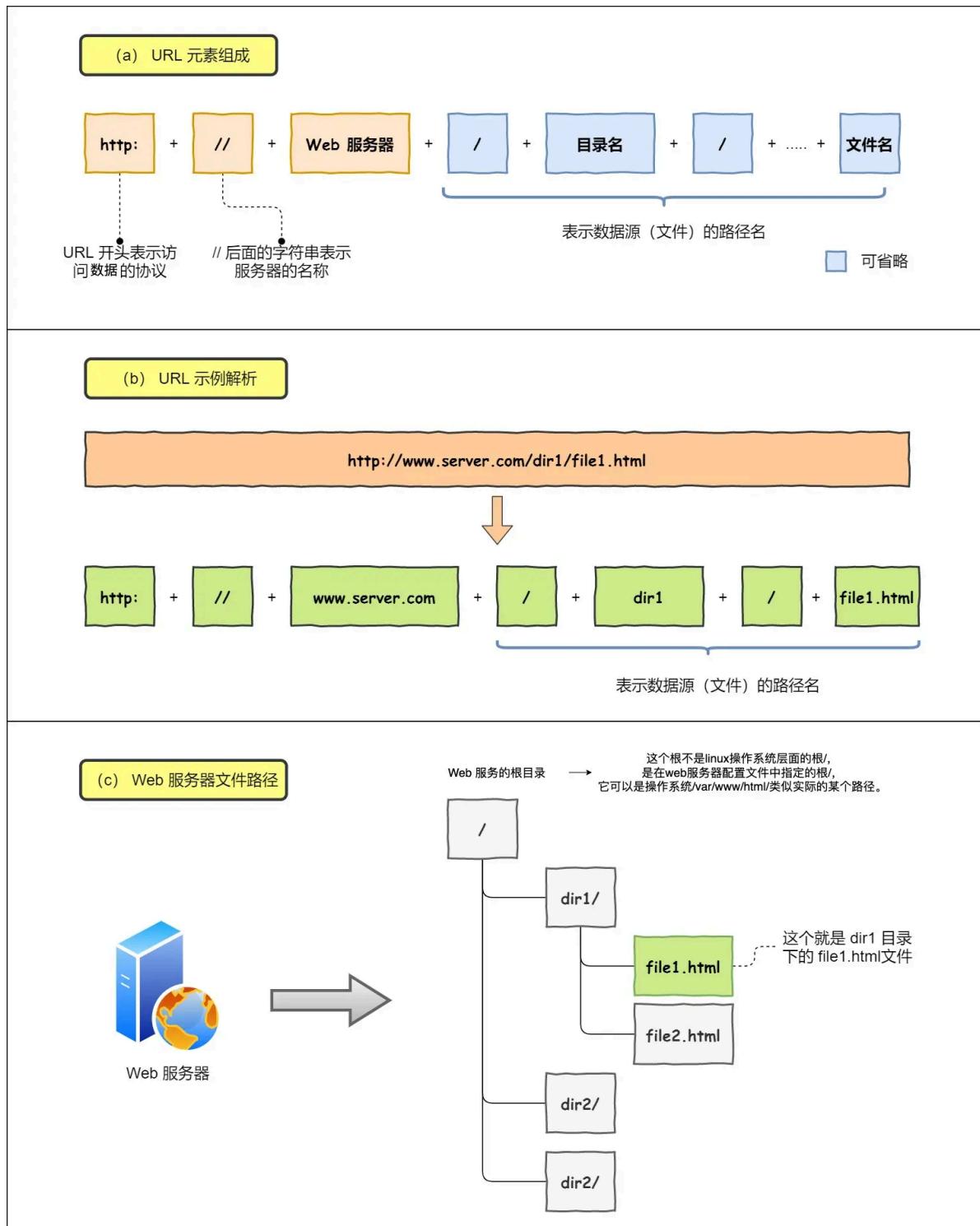


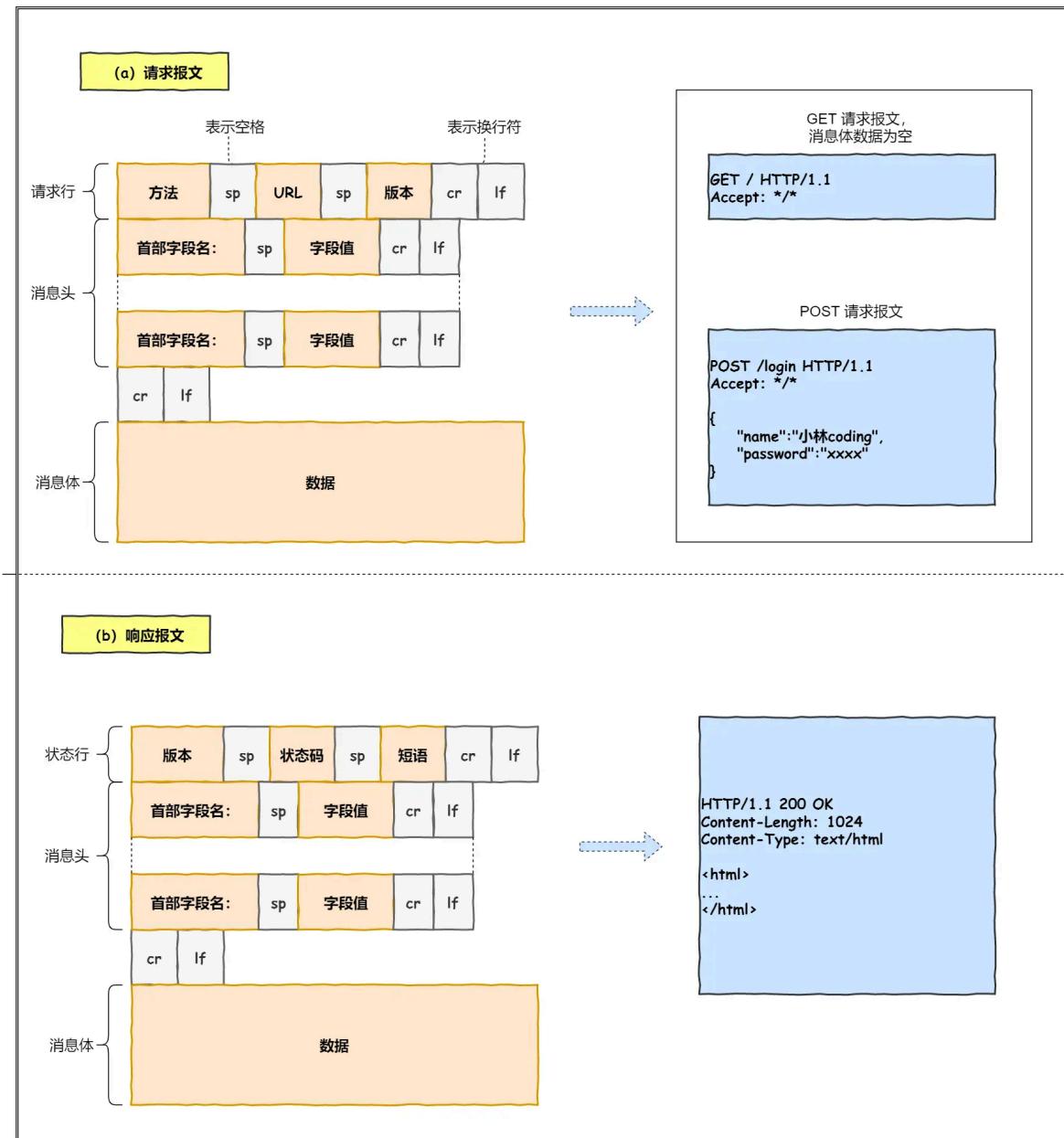
键入网址到网页显示，期间发生了什么？

解析URL



如果请求的文件名省略，默认访问根目录下的默认文件，例如 `index.html`。

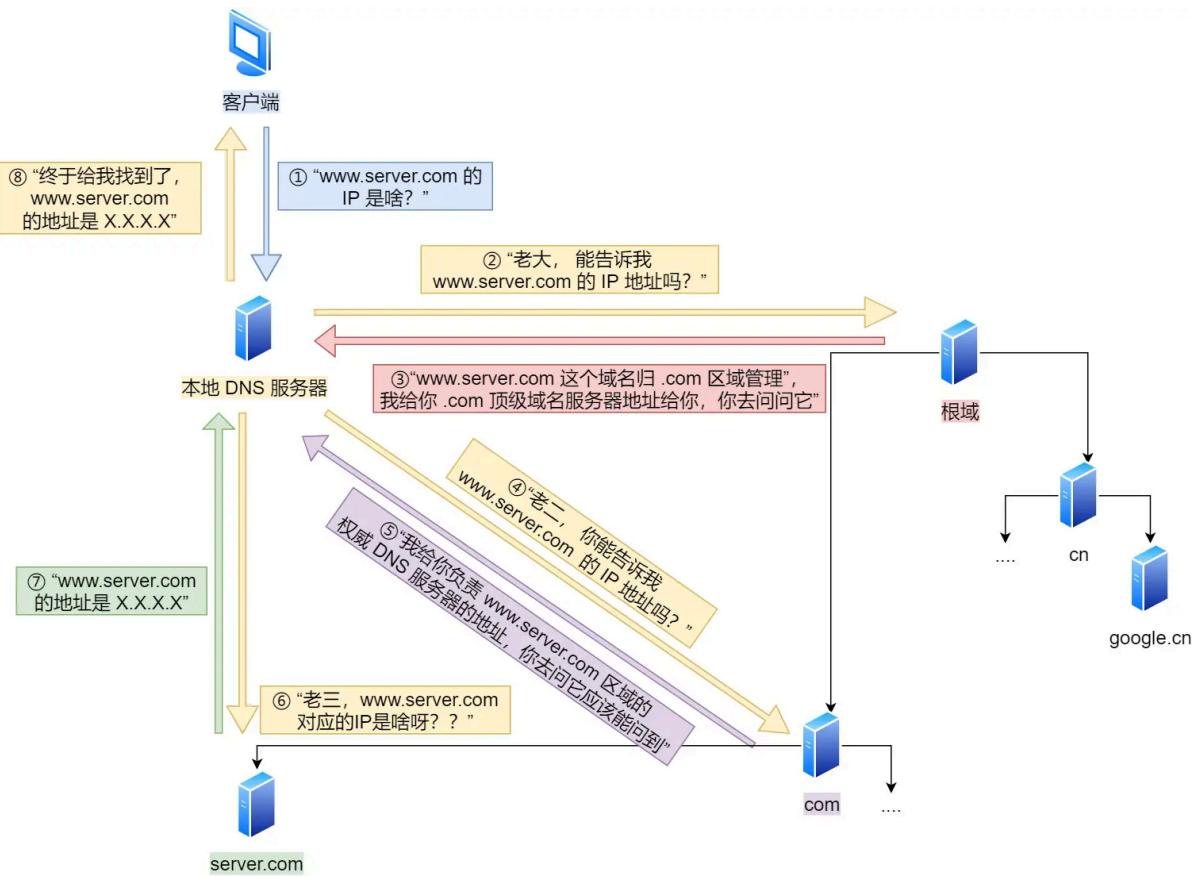
解析 URL 后，浏览器会根据信息构造 `HTTP Request`，请求行中包括请求的资源路径，消息头中包含请求的服务器名，消息头（比如 `POST`）中包含具体的数据。



DNS域名解析

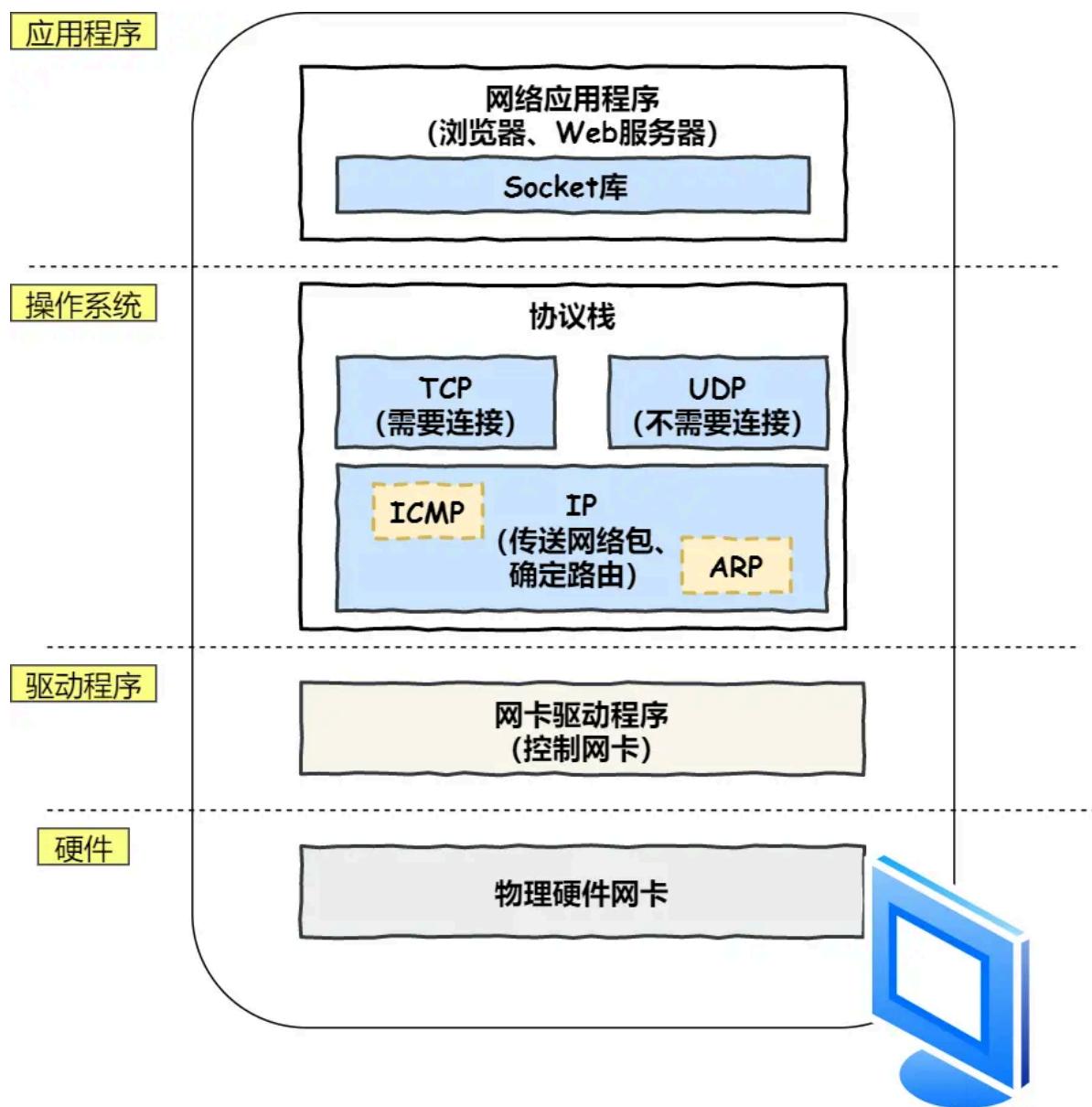
将 `HTTP Request` 发送至目标服务器前，需要对目标服务器的域名进行域名解析。对于域名，越靠右的层级越高，例如 `www.server.com`，完整的是 `www.server.com.`，`.` 是根域名，`com` 是顶级域名。

每个 `dns` 服务器都保存有根域名服务器的地址，域名解析的过程是一层一层去解析的。



当然为了节约时间，会先去查 DNS 缓存（先浏览器查看存储的缓存，找不到 os 再去查，找不到再去查 hosts 文件），不行再去查设置的 dns 服务器。

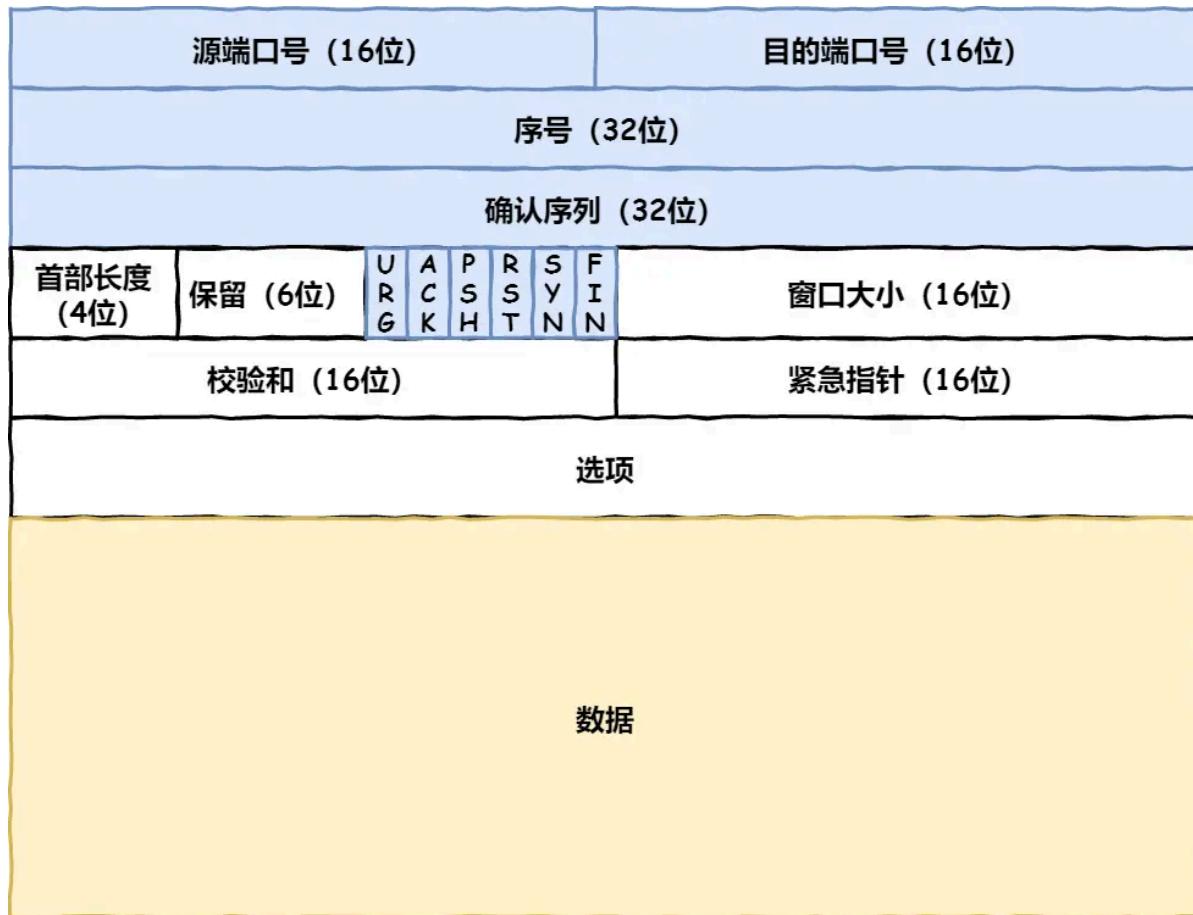
协议栈



完成 dns 解析后，浏览器会通过 socket 库向目标服务器监听套接字发送 connect 请求。HTTP/HTTPS 的传输层协议是 TCP，再通过 TCP 三次握手后，与服务器完成连接建立，并在连接套接字中完成数据的发送和读取。由 IP 协议控制网络包的收发操作，网卡驱动程序和网卡完成网络包的实际发送。

IP 协议包括 ICMP 协议和 ARP 协议，前者用于传送网络通信产生的错误和各种控制信息；后者用于将 IP 地址转换为 MAC 地址。

TCP

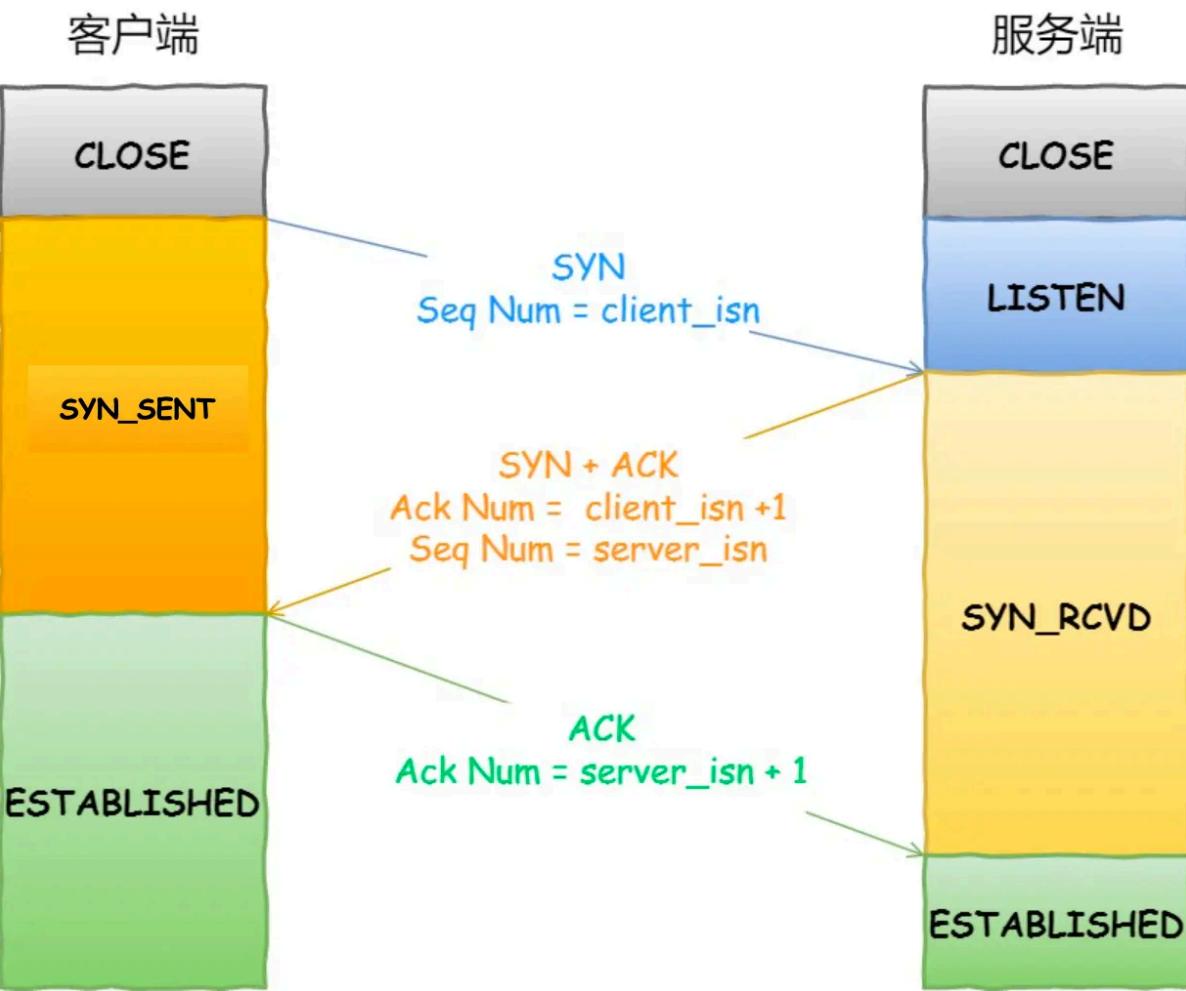


TCP 报文中的一些重要内容：

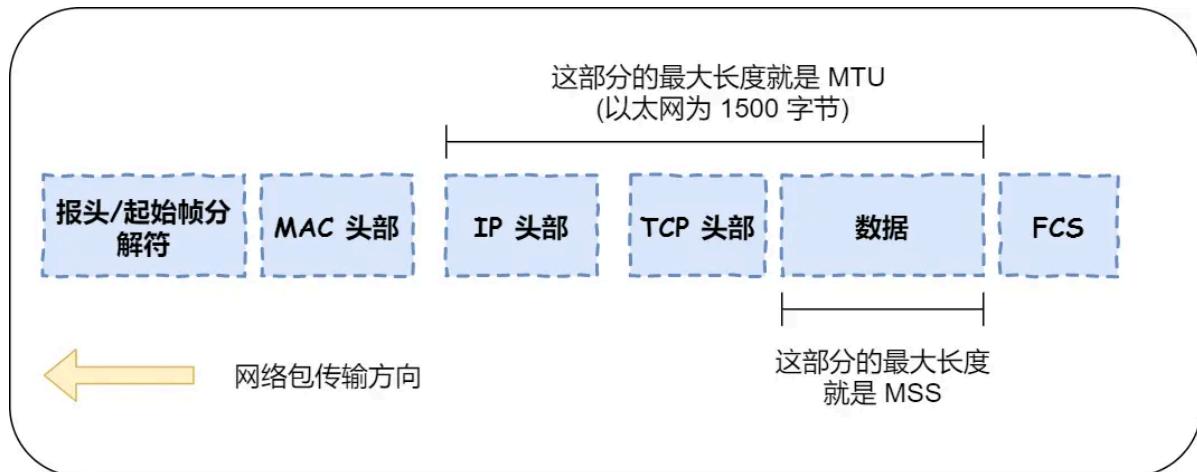
- 源端口号和目的端口号
- 序号：确保包的发送有序
- 确认序号：确保包发送去对方有收到，没收到就继续发送，解决丢包问题
- 状态位：`SYN` 建立连接，`ACK` 为回复，`RST` 为重新连接，`FIN` 为结束连接
- 窗口大小：用于流量控制，标识当前能够处理的能力也就是缓存大小
- 拥塞控制，控制发送速度

TCP 三次握手：

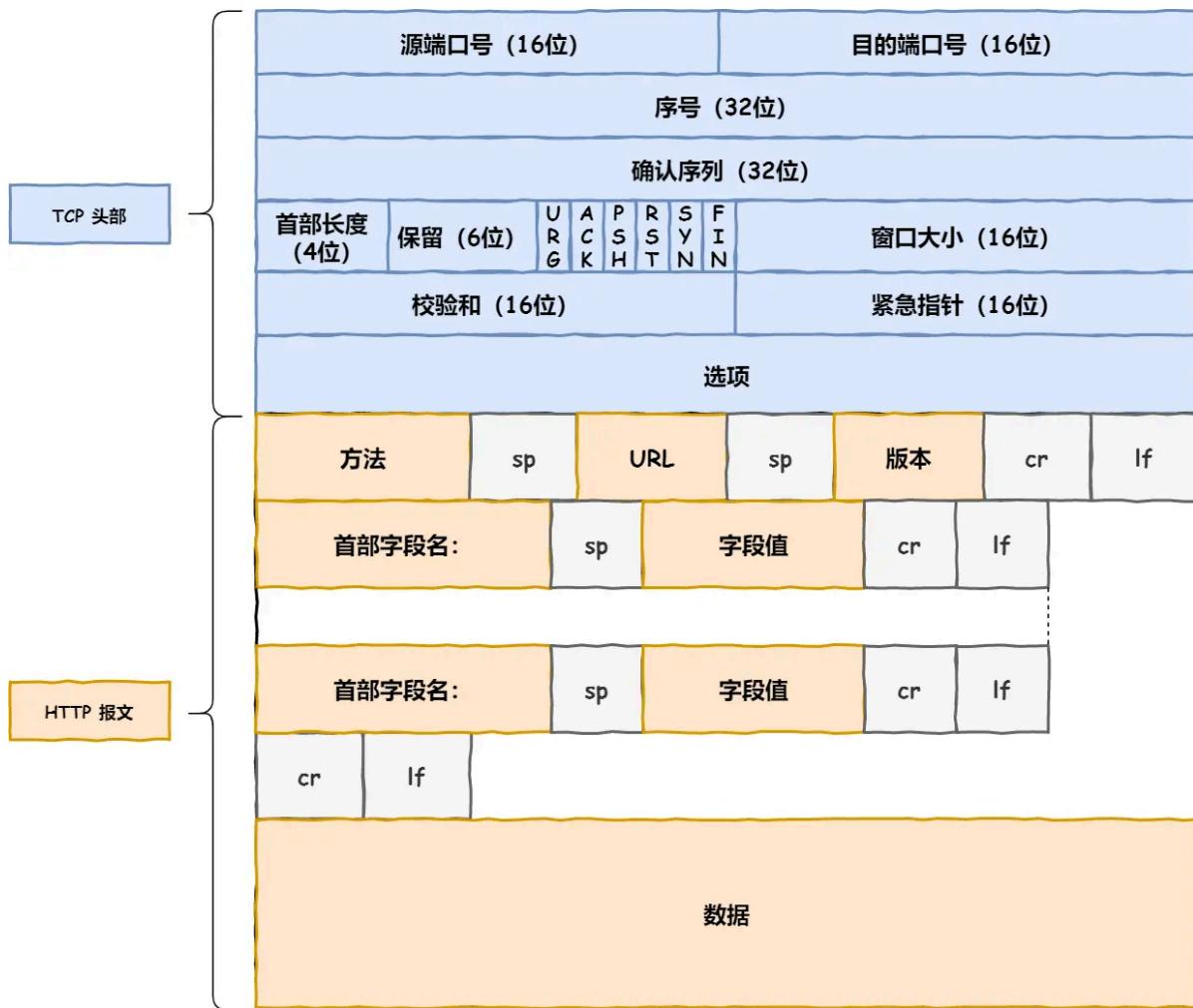
- 如果只有两次握手：客户端发送 `SYN`，服务端回复 `SYN+ACK`，此时服务端准备好接受和发送数据，但是客户端已经崩溃，就会造成资源浪费；另外加入客户端的 `SYN` 因为网络延时延迟发送到了服务端，如果只有两次握手就会建立不必要的连接
- 三次握手可以解决上述问题，并保证了双方都有收发数据的能力
- 注意握手过程中的序列号变化，`ACK` 是 `SYN` 的序列号 +1



如果 TCP 传输的数据太大（超过了 MSS ），会将其进行划分，并加上 TCP 头；而 IP 包划分是 IP 头 + TCP 头 + 数据超过了 MTU 会进行划分。



TCP 报文：



IP

IP包中主要包括：

- 源 IP 地址：如果有多个网卡和 IP 地址，会根据路由表选择一个网卡作为源 IP 地址
- 目的 IP 地址：DNS 解析出的服务器的 IP 地址
- 协议号：这里是 TCP，为 16 进制

MAC

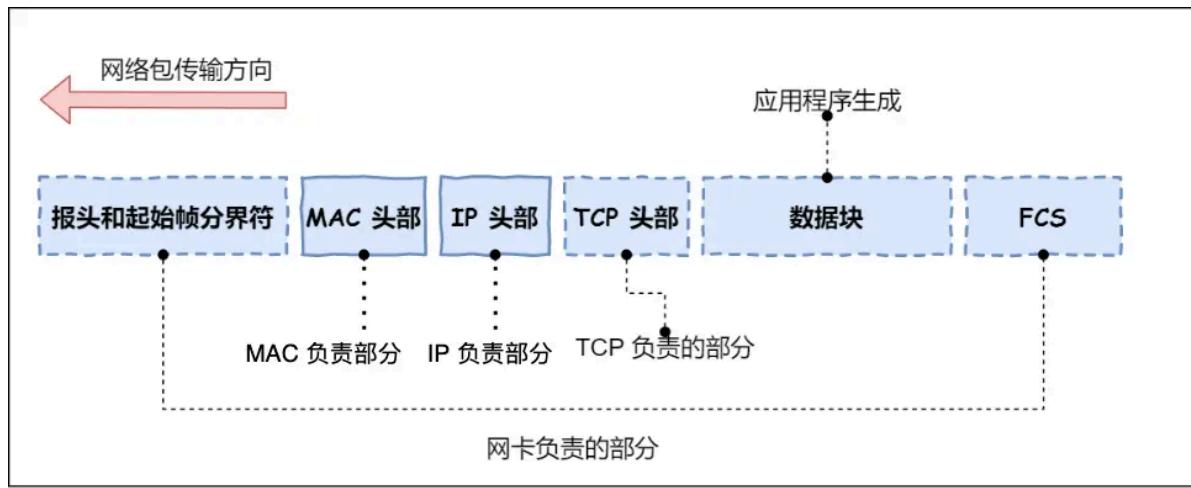
MAC 包头中包括：

- 发送方 MAC 地址，发送方 MAC 地址是网卡生产时烧录到 ROM 中，读出即可
- 源 MAC 地址是下一跳的 MAC 地址，比如下一跳是路由器，源 MAC 地址就是路由器的 MAC 地址
- 协议：TCP/IP 通信中，MAC 包头的协议类型只使用 IP 协议和 ARP 协议

MAC 地址的查找会先去找 ARP 缓存，查找不到再使用 ARP 协议进行广播。

网卡

封装成 IP 包后，网卡驱动程序会将内存中的 IP 包复制到其网卡内的缓冲区中，并加上报头和起始帧分界符，末尾加上校验错误的帧检测序列。再转换为电信号，发送出去。



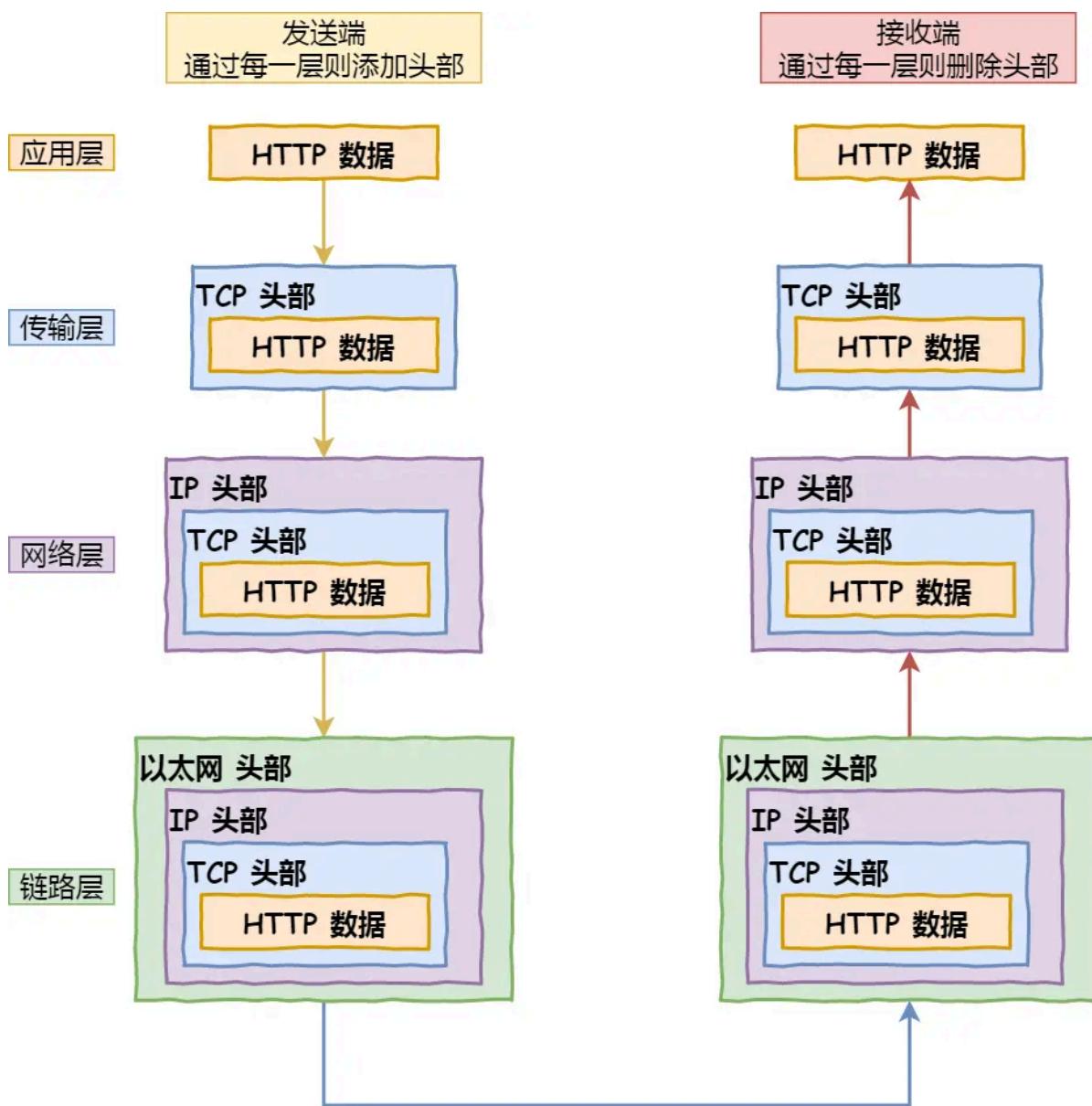
交换机

交换机是二层网络设备，工作在 MAC 层。交换机本身没有 MAC 地址，网络包到达交换机后，交换机会查 MAC 地址表将网络包转发到对应的端口。如果 MAC 地址表中没有，则会广播到除源端口外的其他端口。

路由器

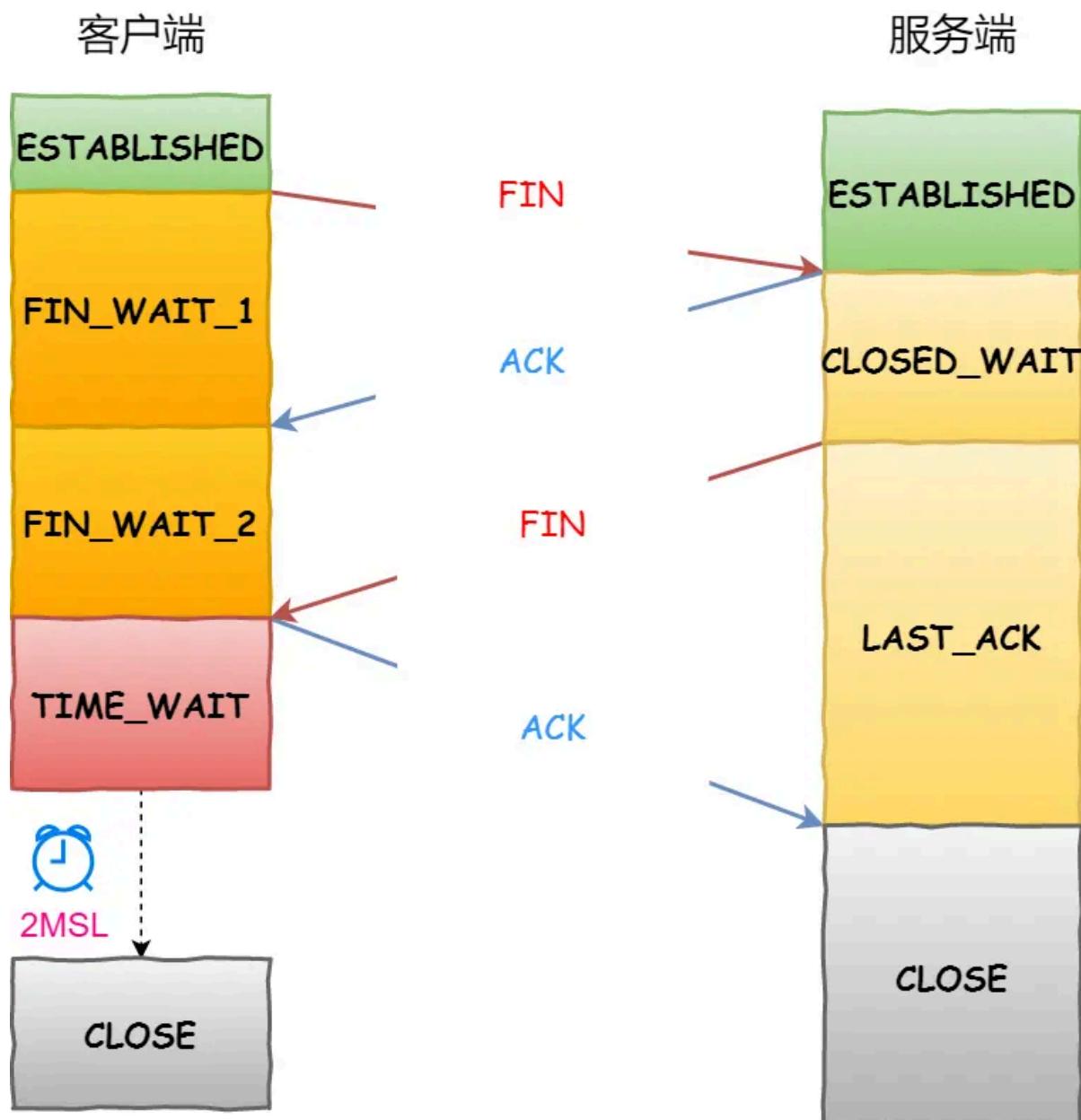
路由器是三层网络设备，每个端口都有 MAC 地址和 IP 地址。路由器会检查网络包中的 MAC 地址是否发送给自己的，不是则丢弃。路由器根据路由表确定需要转发的端口。查询路由表的过程中，如果网关是 IP 地址，说明目标地址不在同一个子网，则需要进一步转发；如果为空，说明目标地址处于同一个子网，通过 ARP 协议得到 MAC 地址，直接发送。

服务器解包



一层层的去掉头部，实际是写入到连接套接字的缓冲区中，由服务器进行读取处理。

四次挥手



TCP/IP网络模型

同一设备上的进程间通信可以通过管道、消息队列、共享内存、信号等，而不同设备上的进程通信就需要通过网络通信，为了兼容不同设备的多样性，需要一套通用的网络协议。

OSI七层模型

请不要把暗号告诉任何人 (Please Do Not Tell the Secret Password to Anyone) 。

- Please | 物理层 (Physical Layer)
- Do | 数据链路层 (Data Link Layer)
- Not | 网络层 (Network Layer)
- Tell | 传输层 (Transport Layer)
- Secret | 会话层 (Session Layer)
- Password | 表示层 (Presentation Layer)
- Anyone | 应用层 (Application Layer)

TCP/IP模型

相较于OSI模型，TCP/IP模型只有5层，更简洁，在实际应用中更为广泛，更符合实际网络通信的需求，是互联网的基础协议。

- 物理层 (Physical Layer)
- 数据链路层 (Data Link Layer)
- 网络层 (Network Layer)
- 传输层 (Transport Layer)
- 应用层 (Application Layer)

也可以把物理层和数据链路层并在一起称为网络接口层，一共4层。

应用层

常见的应用层协议和对应的传输层协议以及端口号。

常用服务	协议	端口号
POP3	TCP	110
IMAP	TCP	143
SMTP	TCP	25
Telnet	TCP	23
终端服务	TCP	3389
PPTP	TCP	1723
HTTP	TCP	80
FTP(控制)	TCP	21
FTP(数据)	TCP	20
HTTPS	TCP	443
NTP	UDP	123
RADIUS	UDP	1645
DHCP	UDP	67
DNS	UDP	53
DNS	TCP	53
SNMP	UDP	161
ipsec	UDP	500
TFTP	UDP	69
L2TP	UDP	1701

大部分应用层协议都是基于TCP的，DNS既有TCP，又有UDP，而DHCP是基于UDP的。

应用层无需关注数据如何传输，只需关注为用户提供具体应用功能。应用层工作在操作系统中的用户态，传输层及以下则工作在内核态。

传输层

传输层为应用层提供网络支持，主要包括2个传输协议**TCP/UDP**。

TCP（传输控制协议）是**面向连接的、可靠的字节流服务，支持流量控制、超时重传、拥塞控制等功能**。能保证连接传送的数据，无差错，不丢失，不重复，且按序到达。

UDP协议相对简单，**只负责发送数据包，不保证数据包是否抵达，实时性更好，传输效率也更高**。当然UDP协议也可以实现可靠传输，只需把TCP的特性在应用层上实现。

对于视频流服务，为了保证视频的完整性和可靠性，视频播放更多的使用TCP（我们可以接受缓存再播放，但画面有马赛克等是不接受的）；而语音视频聊天强调实时性，使用UDP。

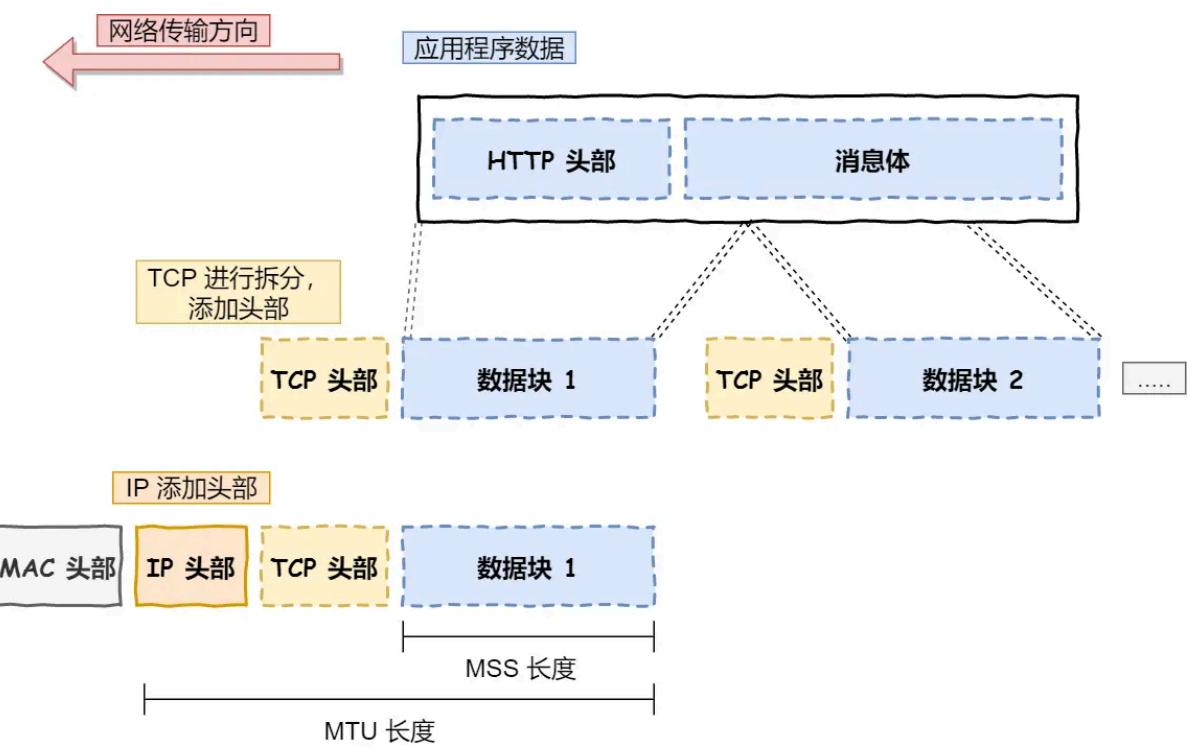
此外，对于实时多媒体数据流，还有**RTP/RTCP**两种传输层协议，前者用于传输流媒体数据、RTCP对RTP进行控制、同步。

如果传输的数据包大小超过了**MSS (TCP最大报文段长度)**，就需要将数据包进行分块，这样即使途中有一个数据块丢失了，只需发送这一个数据块而不是整个数据包。对于TCP协议，每个分块称为**TCP段**。

传输层的报文中会携带端口号，接收方可以识别出报文是发送给哪个进程的。浏览器客户端中的每个标签栏都是一个独立的进程，操作系统会为这些进程分配临时的端口号。

网络层

传输层作为应用间数据传输的媒介，服务好应用到应用间的通信即可，而实际的传输功能交给网络层。



当IP报文超过了**MTU (最大传输单元, 1500字节)**，会进行分片。注意TCP分块时是对应用层数据进行划分，不包括TCP头部；而IP报文划分是对整个IP报文进行划分，包括IP头部。

相较于传输层通过端口号去区分不同的应用，网络层则使用IP地址去区分不同的网络设备。

IP地址分为网络号和主机号，IPV4一共 $4 \times 8 = 32$ 位，IPV6一共 $8 \times 16 = 128$ 位。IP地址需要配合网络掩码才能算出IP地址的网络号和主机号，从而确定唯一的网络设备。

除了寻址能力，IP协议还支持路由。当IP报文到达一个网络节点时，需要通过路由算法决定下一步走哪条路径，具体来说就是去匹配子网，并交给对应的网络，再去匹配主机。

局域网地址

- A类地址：10.0.0.0 - 10.255.255.255
- B类地址：172.16.0.0 - 172.31.255.255
- C类地址：192.168.0.0 - 192.168.255.255

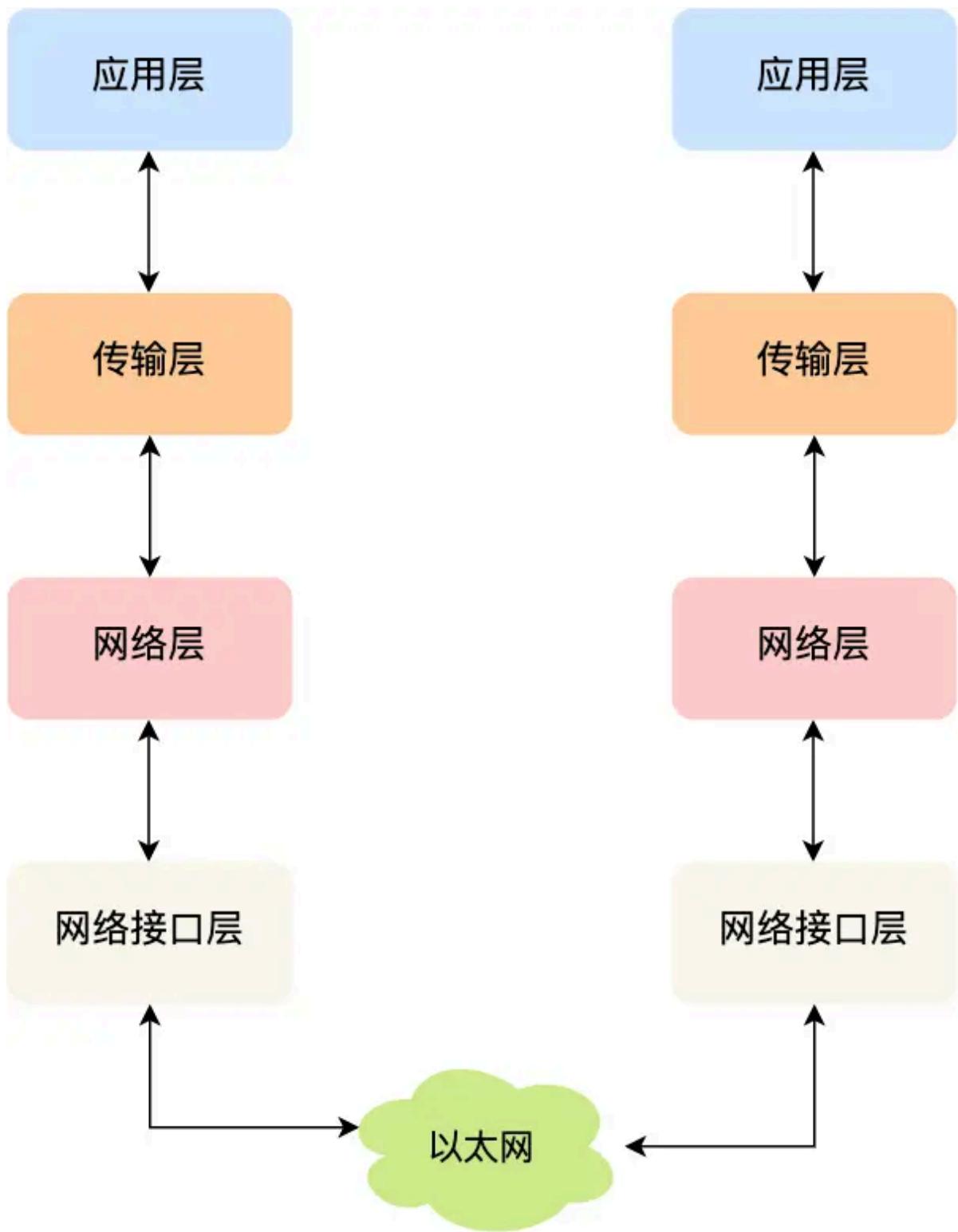
网络接口层

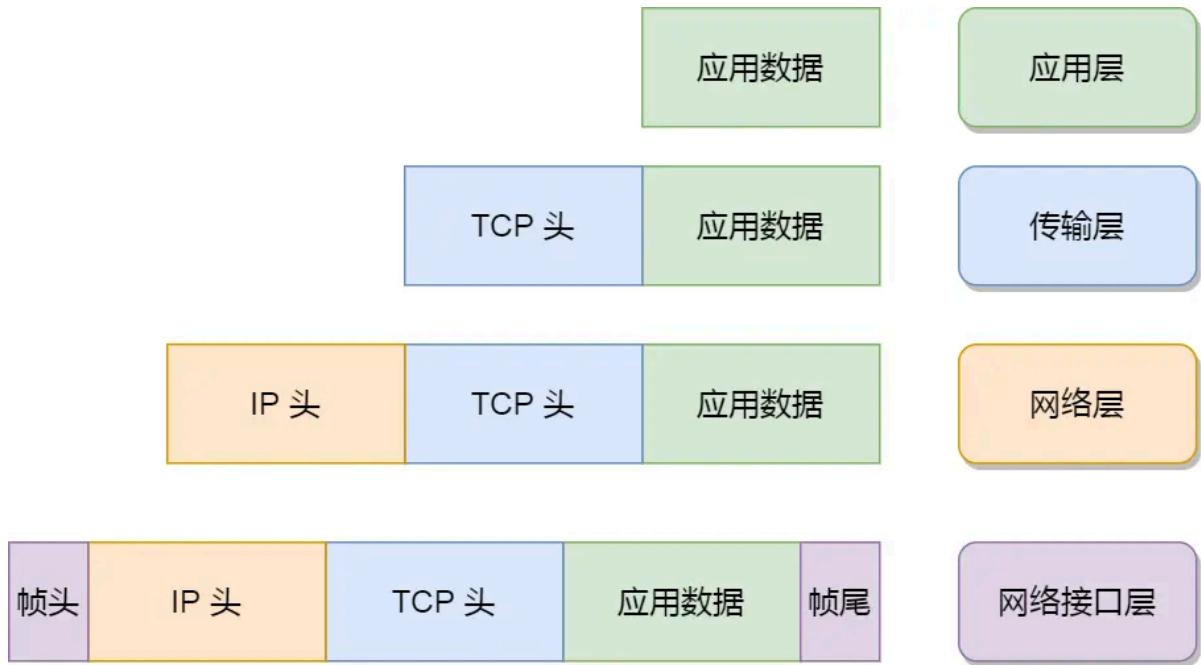
网络接口层在**IP报文前面加上MAC头部，封装成数据帧**。电脑上的以太网接口，Wi-Fi接口，以太网交换机、路由器上的千兆，万兆以太网口，还有网线，它们都是以太网的组成部分。以太网就是一种在「**局域网**」内，把附近的设备连接起来，使它们之间可以进行通讯的技术。

以太网在判断网络包目的地时和IP的方式不同，因此必须采用相匹配的方式才能在以太网中将包发往目的地，而MAC头部就是干这个用的，它包含了接收方和发送方的MAC地址等信息，我们可以通过ARP协议获取对方的MAC地址。

所以说，网络接口层主要为网络层提供「链路级别」传输的服务，负责在以太网、WiFi这样的底层网络上发送原始数据包，工作在网卡这个层次，使用MAC地址来标识网络上的设备。

NAT将私有IP地址转换为公有IP地址；ARP则将IP地址映射为MAC地址。





网络接口层的传输单位是帧 (frame) , IP 层的传输单位是包 (packet) , TCP 层的传输单位是段 (segment) , HTTP 的传输单位则是消息或报文 (message) 。但这些名词并没有什么本质的区别，可以统称为数据包。

有了IP地址，为什么还需要MAC地址

1. 先有的MAC地址，才有的IP地址
2. 集线器是物理层，交换机是数据链路层；前者无脑全部转发全部端口，非接收者需要丢弃数据包，浪费资源并且不安全；后者维护MAC地址表，直接通过特定的端口进行转发。
3. MAC地址设备唯一，最早网络规模小使用MAC地址进行通信和寻址；网络规模大了，比如需要将其他局域网的数据包通过交换机的某个端口转发到路由器，到网络层路由器则需要根据路由表将一个数据包转发到对应的子网，再转发到具体的主机。
4. 正因为交换机需要维护MAC表，端口数量也是有限的，因此对于其他局域网的数据包只需转发到特定的端口（这个端口对应的MAC地址是路由器的MAC地址），路由器再做个中转。如何确定数据包是某个局域网呢？就需要IP地址，而MAC地址的表示中前面是烧录厂商的编号，在一个局域网中使用同一种厂商的设备是不现实的。
5. 数据包传输的过程中变化的是MAC地址，源IP地址和目的IP地址是不会变化的。
6. 理论上是可以只使用IP地址。

socket 编程核心代码

server

```
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>

int main() {
    int server_fd, new_socket;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    const char* hello = "Hello from server";

    // 创建 socket 文件描述符
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // 绑定 socket 到端口 8080
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(8080);

    if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    // 监听这个端口
    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    // 接受连接
    if ((new_socket = accept(server_fd, (struct sockaddr*)&address,
    (socklen_t*)&addrlen)) < 0) {
        perror("accept");
        exit(EXIT_FAILURE);
    }

    // 读取数据
    read(new_socket, buffer, 1024);
    std::cout << "Message from client: " << buffer << std::endl;

    // 发送数据
    send(new_socket, hello, strlen(hello), 0);
    std::cout << "Hello message sent\n";
```

```
// 关闭 socket
close(new_socket);
close(server_fd);

return 0;
}
```

client

```
#include <iostream>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>

int main() {
    int sock = 0;
    struct sockaddr_in serv_addr;
    char buffer[1024] = {0};
    const char* hello = "Hello from client";

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cout << "Socket creation error" << std::endl;
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(8080);

    // 将 IPv4 地址从文本转换为二进制形式
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr) <= 0) {
        std::cout << "Invalid address/ Address not supported" << std::endl;
        return -1;
    }

    // 连接到服务器
    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) < 0) {
        std::cout << "Connection Failed" << std::endl;
        return -1;
    }

    // 发送数据
    send(sock, hello, strlen(hello), 0);
    std::cout << "Hello message sent\n";

    // 接收数据
    int valread = read(sock, buffer, 1024);
    std::cout << "Message from server: " << buffer << std::endl;

    // 关闭 socket
    close(sock);

    return 0;
}
```

TCP中的全连接队列和半连接队列

半连接队列 (SYN队列)

半连接队列，也称为 SYN 队列，用于存储已经接收到 SYN 包但还没有完成三次握手的连接请求。当一个客户端向服务器发送一个 SYN 包（第一次握手，请求建立连接）时，服务器回应一个 SYN-ACK 包（第二次握手，确认客户端的请求并附带自己的连接请求），然后这个连接就会被放入半连接队列中。此时连接还未完全建立，服务器正在等待客户端发送 ACK 包（第三次握手，确认服务器的连接请求）。

全连接队列

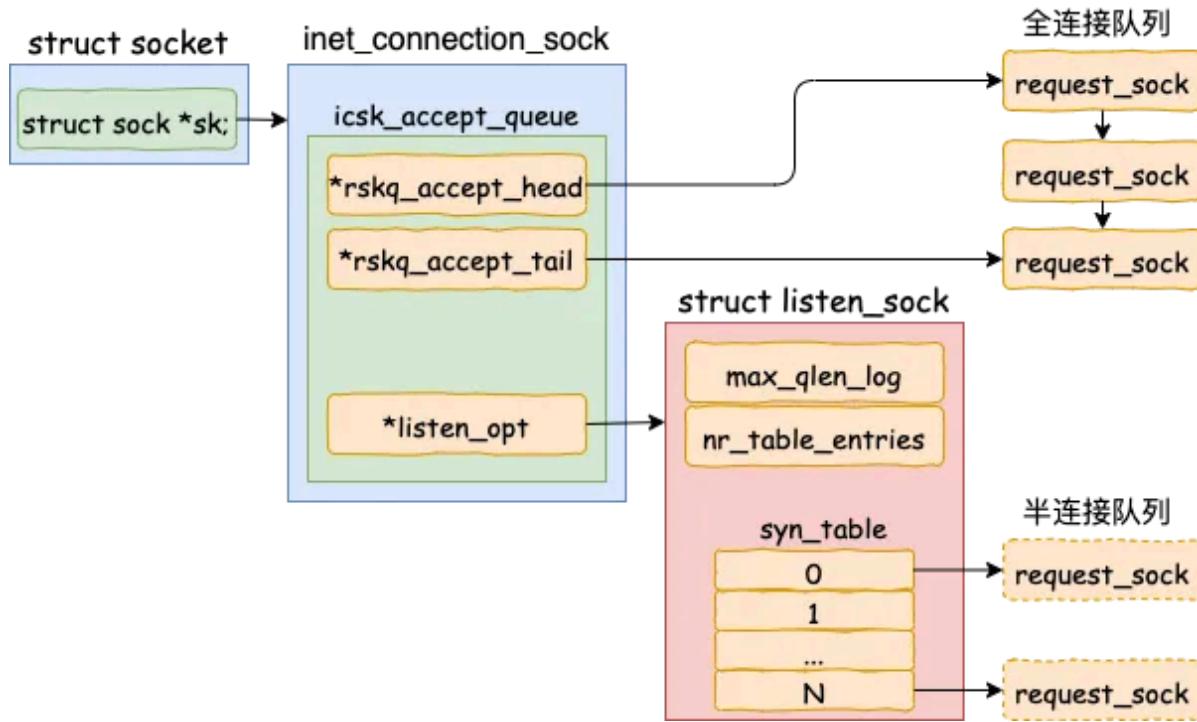
全连接队列用于存储已经完成三次握手的连接请求。当服务器收到客户端的 ACK 包后，三次握手完成，连接从半连接队列移动到全连接队列。在全连接队列中的连接是已经完全建立的，服务器可以开始接收和发送数据。

如果半连接队列太小，那么在高流量条件下，新的连接请求可能会因为队列已满而被丢弃。类似地，全连接队列的大小也需要根据预期的并发连接数来合理配置。

在安全方面，半连接队列可能会成为 SYN 洪水攻击的目标，攻击者通过发送大量的 SYN 包来尝试耗尽服务器的半连接队列资源，导致正常用户无法建立连接。因此，操作系统和网络设备通常提供了一些机制（如 SYN cookies）来防护此类攻击，确保即使在攻击下也能有效管理连接请求。

服务器为什么需要先listen

listen 最主要的工作就是申请和初始化接收队列（计算长度，申请内存），包括全连接队列和半连接队列。其中全连接队列是一个链表，而半连接队列由于需要快速的查找，所以使用的是一个哈希表（其实半连接队列更准确的叫法应该叫半连接哈希表）。



全/半两个队列是三次握手中很重要的两个数据结构，有了它们服务器才能正常响应来自客户端的三次握手。所以服务器端都需要 listen 一下才行。

除此之外我们还有额外收获，我们还知道了内核是如何确定全/半连接队列的长度的。

1.全连接队列的长度

对于全连接队列来说，其最大长度是 listen 时传入的 **backlog**（传入的参数）和 **net.core.somaxconn**（系统级内核参数）之间较小的那个值。如果需要加大全连接队列长度，那么就是调整 **backlog** 和 **somaxconn**。

2.半连接队列的长度

在 listen 的过程中，内核我们也看到了对于半连接队列来说，其最大长度是 `min(backlog, somaxconn, tcp_max_syn_backlog) + 1` 再上取整到 2 的幂次，但最小不能小于 16。如果需要加大半连接队列长度，那么需要一并考虑 `backlog`, `somaxconn` 和 `tcp_max_syn_backlog` 这三个参数。网上任何告诉你修改某一个参数就能提高半连接队列长度的文章都是错的。

客户端connect

客户端在 `connect` 的时候，把本地 `socket` 状态设置成了 `TCP_SYN_SENT`，选了一个可用的端口，接着发出 `SYN` 握手请求并启动重传定时器，该定时器的作用是等到一定时间后收不到服务器的反馈的时候来开启重传。在 3.10 版本中首次超时时间是 1 s，一些老版本中是 3 s。

客户端建立连接前需要确定一个端口，该端口会在两个位置进行确定。

第一个位置，也是最主要的确定时机是 `connect` 系统调用执行过程。在 `connect` 的时候，会随机地从 `ip_local_port_range` 选择一个位置开始循环判断。找到可用端口后，发出 `syn` 握手包。如果端口查找失败，会报错 “`Cannot assign requested address`”。这个时候你应该首先想到去检查一下服务器上的 `net.ipv4.ip_local_port_range` 参数，是不是可以再放的多一些。

如果你因为某种原因不希望某些端口被使用到，那么就把它们写到 `ip_local_reserved_ports` 这个内核参数中就行了，内核在选择的时候会跳过这些端口。

另外注意即使是一个端口是可以被用于多条 TCP 连接的（四元组）。所以一台客户端机最大能建立的连接数并不是 65535。只要 server 足够多，单机发出百万条连接没有任何问题。



ESTAB	0	0	.192:10000	.119:8109
ESTAB	0	0	.192:10000	.119:8101
ESTAB	0	0	.192:10000	.119:8107
ESTAB	0	0	.192:10000	.119:8102
ESTAB	0	0	.192:10000	.119:8106
ESTAB	0	0	.192:10000	.119:8113
ESTAB	0	0	.192:10000	.119:8110
ESTAB	0	0	.192:10000	.119:8114
ESTAB	0	0	.192:10000	.119:8108
ESTAB	0	0	.192:10000	.119:8116
ESTAB	0	0	.192:10000	.119:8103
ESTAB	0	0	.192:10000	.119:8112

截图中左边的 192 是客户端，右边的 119 是服务器的 ip。可以看到客户端的 10000 这个端口号是用在了多条连接上了的。

第二个位置，如果在 `connect` 之前使用了 `bind`，将会使得 `connect` 时的端口选择方式无效。转而使用 `bind` 时确定的端口。`bind` 时如果传入了端口号，会尝试首先使用该端口号，如果传入了 0，也会自动选择一个。但默认情况下一个端口只会被使用一次。所以对于客户端角色的 `socket`，不建议使用 `bind`！

最后我再想多说一句，上面的选择端口的都是从 `ip_local_port_range` 范围中的某一个随机位置开始循环的。如果可用端口很充足，则能快一点找到可用端口，那循环很快就能退出。假设实际上 `ip_local_port_range` 中的端口快被用光了，这时候内核就大概率得把循环多执行很多轮才能找到，这会导致 `connect` 系统调用的 CPU 开销的上涨。所以，最好不要等到端口不够用了才加大 `ip_local_port_range` 的范围，而是事先就应该保持一个充足的范围。

服务器响应SYN

服务器响应 `ack` 是主要工作是判断下半连接队列是否满了，满的话可能会丢弃该请求，否则发出 `synack`。申请 `request_sock` 添加到半连接队列中，同时启动定时器（通常是重传定时器），确保如果客户端没有响应，服务器会在一定时间后重发 `SYN-ACK`。这也帮助服务器管理半连接队列中的条目，防止资源长时间被半开放连接占用。

客户端响应SYN ACK

客户端响应来自服务器端的 synack 时清除了 connect 时设置的重传定时器（也就是SYN发送时设置的），把当前 socket 状态设置为 ESTABLISHED，开启保活计时器（如果配置了 TCP keepalive 选项，这个计时器用于定期发送保活包，以确保连接在长时间无数据交换时仍然保持活跃，并检测任何潜在的连接断开或故障）后发出第三次握手的 ack 确认。

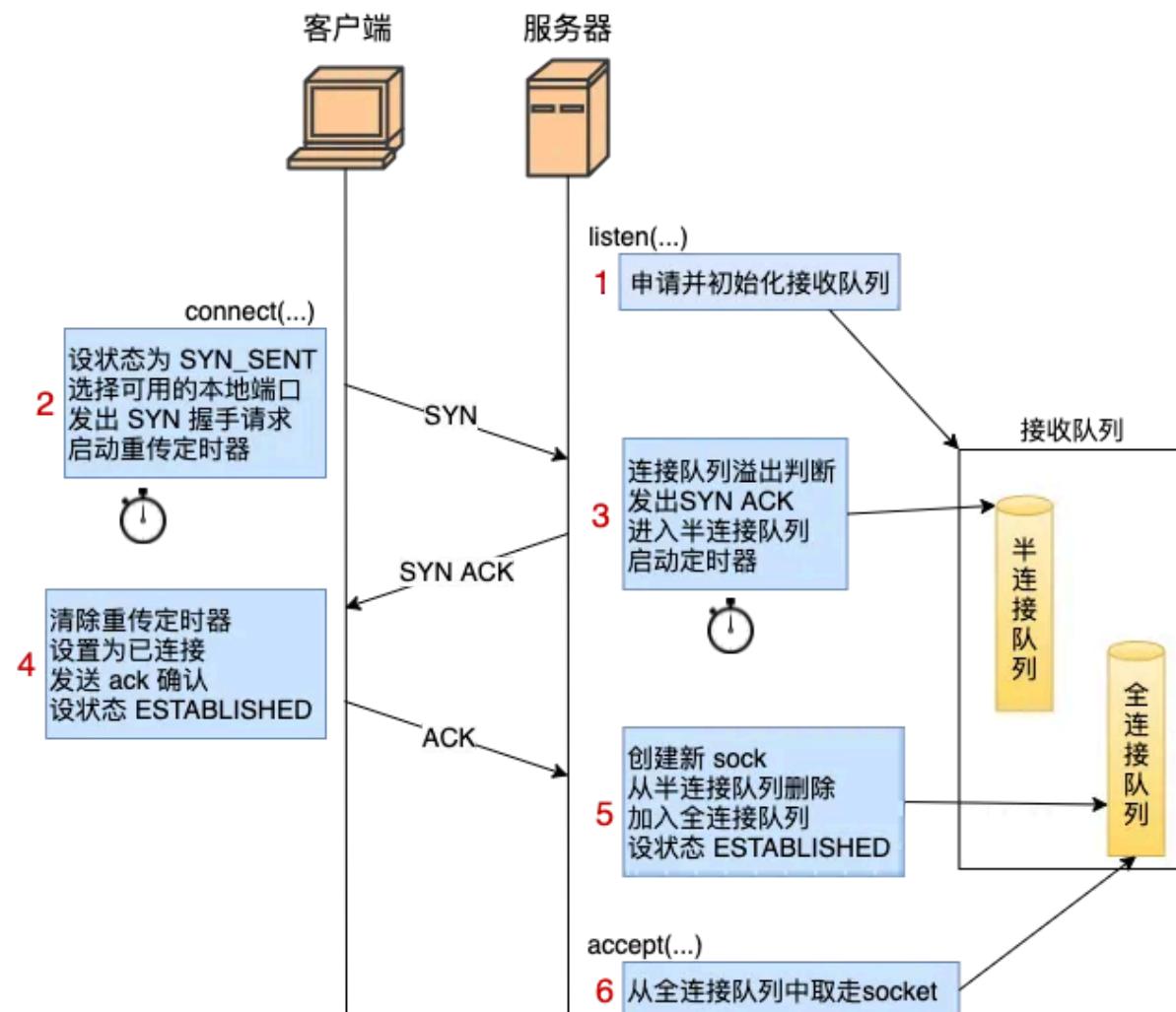
服务器响应ACK

服务器响应第三次握手 ack 所做的工作是把当前半连接对象删除，创建了新的 socket 后加入到全连接队列中，最后将新连接状态设置为 ESTABLISHED。

服务器accept

accept 的重点工作就是从已经建立好的全连接队列中取出一个返回给用户进程。

总结



- 服务器 listen 时，计算了全/半连接队列的长度，还申请了相关内存并初始化。
- 客户端 connect 时，把本地 socket 状态设置成了 TCP_SYN_SENT，选则一个可用的端口，发出 SYN 握手请求并启动重传定时器。
- 服务器响应 ack 时，会判断下接收队列是否满了，满的话可能会丢弃该请求。否则发出 synack，申请 request_sock 添加到半连接队列中，同时启动定时器。
- 客户端响应 synack 时，清除了 connect 时设置的重传定时器，把当前 socket 状态设置为 ESTABLISHED，开启保活计时器后发出第三次握手的 ack 确认。

- 服务器响应 ack 时，把对应半连接对象删除，创建了新的 sock 后加入到全连接队列中，最后将新连接状态设置为 ESTABLISHED。
- accept 从已经建立好的全连接队列中取出一个返回给用户进程。

另外要注意的是，如果握手过程中发生丢包（网络问题，或者是连接队列溢出），内核会等待定时器到期后重试，重试时间间隔在 3.10 版本里分别是 1s 2s 4s ...。在一些老版本里，比如 2.6 里，第一次重试时间是 3 秒。最大重试次数分别由 `tcp_syn_retries` 和 `tcp_synack_retries` 控制。

如果你的线上接口正常都是几十毫秒内返回，但偶尔出现了 1 s、或者 3 s 等这种偶发的响应耗时变长的问题，那么你就要去定位一下看看是不是出现了握手包的超时重传了。

IP 基本认识

IP 在 TCP/IP 参考模型中处于第三层，也就是**网络层**。网络层的主要作用是：**实现主机与主机之间的通信，也叫点对点（end to end）通信。**

IP 的作用是主机之间通信用的，而 **MAC** 的作用则是实现「直连」的两个设备之间通信，而 IP 则负责在「没有直连」的两个网络之间进行通信传输。

计算机网络中也需要「数据链路层」和「网络层」这个分层才能实现向最终目标地址的通信。**源IP地址**和**目标IP地址**在传输过程中是不会变化的（前提：没有使用 NAT 网络），只有**源 MAC 地址**和**目标 MAC**一直在变化。

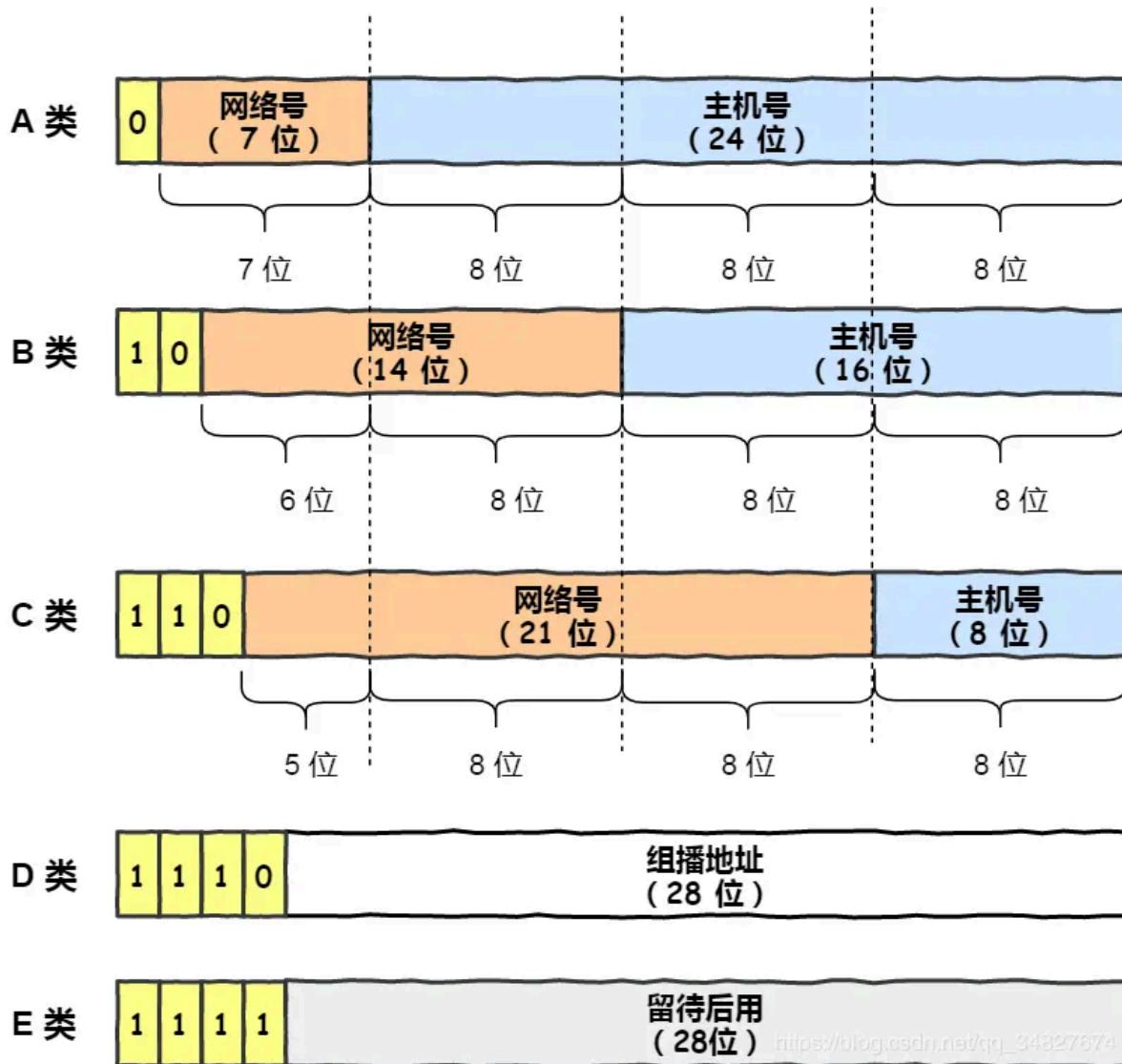
IP地址

在 TCP/IP 网络通信时，为了保证能正常通信，每个设备都需要配置正确的 IP 地址，否则无法实现正常的通信。IP 地址（IPv4 地址）由 32 位正整数来表示，IP 地址在计算机是以二进制的方式处理的。而人类为了方便记忆采用了**点分十进制**的标记方式，也就是将 32 位 IP 地址以每 8 位为组，共分为 4 组，每组以「.」隔开，再将每组转换成十进制。

IP 地址并不是根据主机台数来配置的，而是以网卡。像服务器、路由器等设备都是有 2 个以上的网卡（一块网卡也可以设置2个以上的IP地址），也就是它们会有 2 个以上的 IP 地址。

IP 地址的分类

对于 A、B、C 类主要分为两个部分，分别是**网络号**和**主机号**。



类别	IP 地址范围	最大主机数
A	0.0.0.0 ~ 127.255.255.255	16777214
B	128.0.0.0 ~ 191.255.255.255	65534
C	192.0.0.0 ~ 223.255.255.255	254

最大主机个数，就是要看主机号的位数，如C类地址的主机号占8位，那么C类地址的最大主机个数： $2^8 - 2 = 254$ ，为什么要减2呢？因为在IP地址中，有两个IP是特殊的，分别是主机号全为1和全为0地址。

- 主机号全为1指定某个网络下的所有主机，用于广播。广播地址用于在同一个链路中相互连接的主机之间发送数据包。
- 主机号全为0指定某个网络

广播地址可以分为本地广播和直接广播两种。

- 在本网络内广播的叫做本地广播。例如网络地址为 192.168.0.0/24 的情况下，广播地址是 192.168.0.255。因为这个广播地址的 IP 包会被路由器屏蔽，所以不会到达 192.168.0.0/24 以外的其他链路上。
- 在不同网络之间的广播叫做直接广播。例如网络地址为 192.168.0.0/24 的主机向 192.168.1.255/24 的目标地址发送 IP 包。收到这个包的路由器，将数据转发给 192.168.1.0/24，从而使得所有 192.168.1.1~192.168.1.254 的主机都能收到这个包（由于直接广播有一定的安全问题，多数情况下会在路由器上设置为不转发。）。

D 类和 E 类地址是没有主机号的，所以不可用于主机 IP，D 类常被用于多播，E 类是预留的分类，暂时未使用。多播用于**将包发送给特定组内的所有主机**。由于广播无法穿透路由，若想给其他网段发送同样的包，就可以使用可以穿透路由的多播。

IP分类的优点：不管是路由器还是主机解析到一个 IP 地址时候，我们判断其 IP 地址的首位是否为 0，为 0 则为 A 类地址，那么就能很快的找出网络地址和主机地址。

IP分类的缺点：

1. 同一网络下没有地址层次，比如一个公司里用了 B 类地址，但是可能需要根据生产环境、测试环境、开发环境来划分地址层次，而这种 IP 分类是没有地址层次划分的功能，所以这就**缺少地址的灵活性**。
2. A、B、C类有个尴尬处境，就是**不能很好的与现实网络匹配**。
 - C 类地址能包含的最大主机数量实在太少了，只有 254 个，估计一个网吧都不够用。
 - 而 B 类地址能包含的最大主机数量又太多了，6 万多台机器放在一个网络下面，一般的企业基本达不到这个规模，闲着的地址就是浪费。

无分类地址 CIDR

这种方式不再有分类地址的概念，32 比特的 IP 地址被划分为两部分，前面是**网络号**，后面是**主机号**。比如 10.100.122.2/24，这种地址表示形式就是 CIDR，/24 表示前 24 位是网络号，剩余的 8 位是主机号。

还有另一种划分网络号与主机号形式，那就是**子网掩码**，掩码的意思就是掩盖掉主机号，剩余的就是网络号。**将子网掩码和 IP 地址按位计算 AND，就可得到网络号**。实际上子网掩码还有一个作用，那就是**划分子网**。

子网划分实际上是将主机地址分为两个部分：子网网络地址和子网主机地址。形式如下：

- 未做子网划分的 ip 地址：网络地址 + 主机地址
- 做子网划分后的 ip 地址：网络地址 + （子网网络地址 + 子网主机地址）

假设对 C 类地址进行子网划分，网络地址 192.168.1.0，使用子网掩码 255.255.255.192 对其进行子网划分。C 类地址中前 24 位是网络号，最后 8 位是主机号，根据子网掩码可知**从 8 位主机号中借用 2 位作为子网号**。

	网络号	主机号
C 类网络地址	192.168.1	.0
子网掩码 255.255.255.192	255.255.255	11 000000
子网划分	网络地址 24 位	子网网络地址 2 位 子网主机地址 6 位

由于子网网络地址被划分成 2 位，那么子网地址就有 4 个，分别是 00、01、10、11，划分后的 4 个子网如下表格：

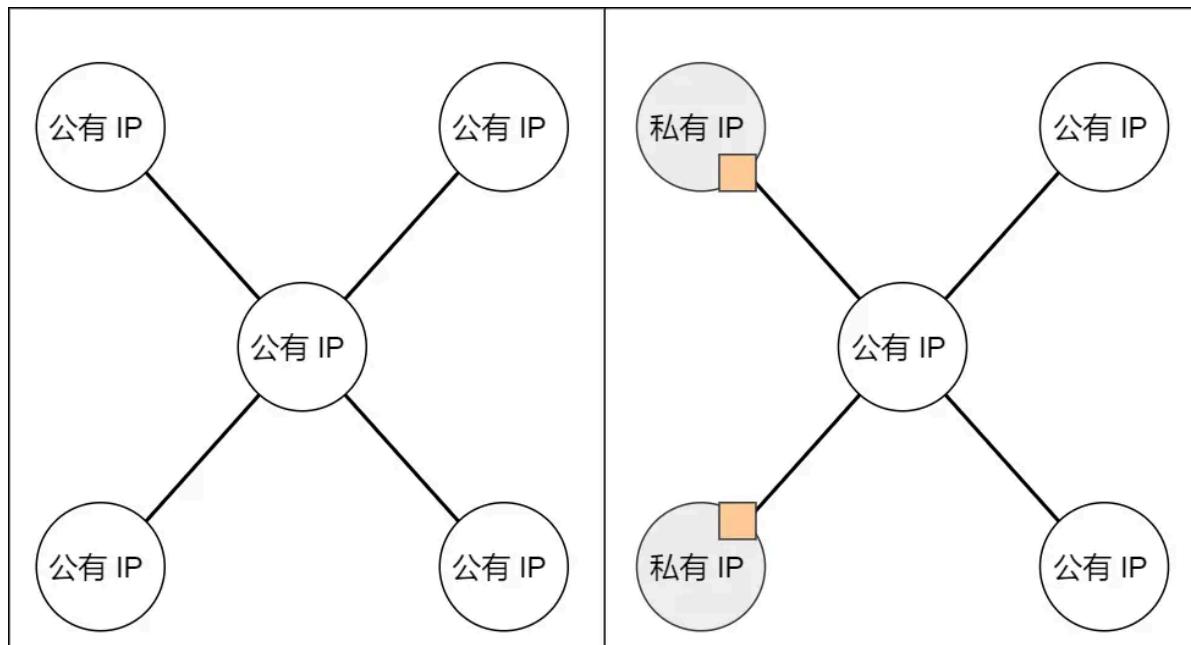
子网号	网络地址	主机地址范围	广播地址
0	192.168.1.0	192.168.1.1 ~ 192.168.1.62	192.168.1.63
1	192.168.1.64	192.168.1.65 ~ 192.168.1.126	192.168.1.127
2	192.168.1.128	192.168.1.129 ~ 192.168.1.190	192.168.1.191
3	192.168.1.192	192.168.1.193 ~ 192.168.1.254	192.168.1.255

为什么要分离网络号和主机号？因为两台计算机要通讯，首先要判断是否处于同一个广播域内，即网络地址是否相同。如果网络地址相同，表明接受方在本网络上，那么可以把数据包直接发送到目标主机。路由器寻址工作中，也就是通过这样的方式来找到对应的网络号的，进而把数据包转发给对应的网络内。

公有 IP 地址与私有 IP 地址

在 A、B、C 分类地址，实际上有分公有 IP 地址和私有 IP 地址。

类别	IP 地址范围	最大主机数	私有 IP 地址范围
A	0.0.0.0 ~ 127.255.255.255	16777214	10.0.0.0 ~ 10.255.255.255
B	128.0.0.0 ~ 191.255.255.255	65534	172.16.0.0 ~ 172.31.255.255
C	192.0.0.0 ~ 223.255.255.255	254	192.168.0.0 ~ 192.168.255.255



每个公有 IP 是不能重复的

■ 表示 NAT 转换的 IP 地址

私有 IP 地址通常是内部的 IT 人员管理，公有 IP 地址是由 ICANN 组织管理，中文叫「互联网名称与数字地址分配机构」。

IP 地址与路由控制

IP 地址的**网络地址**这一部分是用于进行路由控制。路由控制表中记录着网络地址与下一步应该发送至路由器的地址。在主机和路由器上都会有各自的路由器控制表。在发送 IP 包时，首先要确定 IP 包首部中的**目标地址**，再从路由控制表中找到与该地址具有相同网络地址的记录，根据该记录将 IP 包转发给相应的下一个路由器。如果路由控制表中存在多条相同网络地址的记录，就选择相同位数最多的网络地址，也就是**最长匹配**。

环回地址是在同一台计算机上的程序之间进行网络通信时所使用的一个默认地址。计算机使用一个特殊的 IP 地址 **127.0.0.1 作为环回地址**。与该地址具有相同意义的是一个叫做 **localhost** 的主机名。使用这个 IP 或主机名时，数据包不会流向网络。

IP 分片与重组

每种数据链路的最大传输单元 MTU 都是不相同的，如 FDDI 数据链路 MTU 4352、以太网的 MTU 是 1500 字节等。那么当 IP 数据包大小大于 MTU 时，IP 数据包就会被分片。

经过分片之后的 IP 数据报在被重组的时候，只能由目标主机进行，路由器是不会进行重组的（注意分片是可以在中间路由器进行的，IPv6 则不允许）。假设发送方发送一个 4000 字节的大数据报，若要传输在以太网链路，则需要把数据报分片成 3 个小数据报进行传输，再交由接收方重组为大数据报。在分片传输中，一旦某个分片丢失，则会造成整个 IP 数据报作废，所以 TCP 引入了 **MSS** 也就是在 TCP 层进行分片不由 IP 层分片，那么对于 UDP 我们尽量不要发送一个大于 **MTU** 的数据报文。

IPv6 基本认识

IPv4 的地址是 32 位的，大约可以提供 42 亿个地址，但是早在 2011 年 IPv4 地址就已经被分配完了。但是 IPv6 的地址是 128 位的，是以每 16 位作为一组，每组用冒号「:」隔开。如果出现连续的 0 时还可以将这些 0 省略，并用两个冒号「::」隔开。但是，一个 IP 地址中只允许出现一次两个连续的冒号。说个段子 IPv6 可以保证地球上的每粒沙子都能被分配到一个 IP 地址。

IPv6 用二进制数表示

```
111111011011100 : 1011101010011000 : 0111011001010100 : 0011001000010000 : 111111011011100 : 1011101010011000 : 0111011001010100 : 0011001000010000
```

IPv6 十六进制数表示

```
FEDC: BA98: 7654: 3210: FEDC: BA98: 7654: 3210
```

但 IPv6 除了有更多的地址之外，还有更好的安全性和扩展性。但是因为 **IPv4 和 IPv6 不能相互兼容**（要解决，可以通过 Nginx 配置了同时监听 IPv4 和 IPv6 的端口，并将所有请求转发到 IPv4 后端服务器。），所以不但要我们电脑、手机之类的设备支持，还需要网络运营商对现有的设备进行升级，所以这可能是 IPv6 普及率比较慢的一个原因。

IPv6 不仅仅只是可分配的地址变多了，它还有非常多的亮点。

- **IPv6 可自动配置，即使没有 DHCP 服务器也可以实现自动分配IP地址**
- **IPv6 包头包首部长度采用固定的值 40 字节，去掉了包头校验和，简化了首部结构，减轻了路由器负荷，大大提高了传输的性能**
- **IPv6 有应对伪造 IP 地址的网络安全功能以及防止线路窃听的功能，大大提升了安全性。**

DNS

DNS 域名解析，DNS 可以将域名网址自动转换为具体的 IP 地址。DNS 中的域名都是用**句点**来分隔的，比如 `www.server.com`，这里的句点代表了不同层次之间的**界限**。根域是在最顶层，它的下一层就是 com 顶级域，再下面是 server.com。

根域的 DNS 服务器信息保存在互联网中所有的 DNS 服务器中。这样一来，任何 DNS 服务器就都可以找到并访问根域 DNS 服务器了。因此，**客户端只要能够找到任意一台 DNS 服务器，就可以通过它找到根域 DNS 服务器，然后再一路顺藤摸瓜找到位于下层的某台目标 DNS 服务器。**

域名解析的流程：浏览器首先看一下自己的缓存里有没有，如果没有就向操作系统的缓存要，还没有就检查本机域名解析文件 `hosts`，如果还是没有，就会 DNS 服务器进行查询，查询的过程如下：

1. 客户端首先会发出一个 DNS 请求，问 `www.server.com` 的 IP 是啥，并发给本地 DNS 服务器（也就是客户端的 TCP/IP 设置中填写的 DNS 服务器地址）。
2. 本地域名服务器收到客户端的请求后，**如果缓存里的表格能找到 `www.server.com`，则它直接返回 IP 地址**。如果没有，本地 DNS 会去问它的根域名服务器：“老大，能告诉我 `www.server.com` 的 IP 地址吗？”**根域名服务器是最高层次的，它不直接用于域名解析，但能指明一条道路。**
3. 根 DNS 收到来自本地 DNS 的请求后，发现后置是 .com，说：“`www.server.com` 这个域名归 .com 区域管理，我给你 .com 顶级域名服务器地址给你，你去问问它吧。”
4. 本地 DNS 收到顶级域名服务器的地址后，发起请求问“老二，你能告诉我 `www.server.com` 的 IP 地址吗？”
5. 顶级域名服务器说：“我给你负责 `www.server.com` 区域的权威 DNS 服务器的地址，你去问它应该能问到”。
6. 本地 DNS 于是转向问权威 DNS 服务器：“老三，`www.server.com` 对应的 IP 是啥呀？”server.com 的权威 DNS 服务器，它是域名解析结果的原出处。为啥叫权威呢？就是我的域名我做主。
7. **权威 DNS 服务器查询后将对应的 IP 地址 X.X.X.X 告诉本地 DNS。**
8. **本地 DNS 再将 IP 地址返回客户端，客户端和目标建立连接。**

ARP

在传输一个 IP 数据报的时候，确定了源 IP 地址和目标 IP 地址后，就会通过主机「路由表」确定 IP 数据包下一跳。然而，网络层的下一层是数据链路层，所以我们还要知道「下一跳」的 MAC 地址。由于主机的路由表中可以找到下一跳的 IP 地址，所以可以通过 ARP 协议，求得下一跳的 MAC 地址。ARP 是借助 ARP 请求与 ARP 响应两种类型的包确定 MAC 地址的。

- 主机会通过广播发送 ARP 请求，这个包中包含了想要知道的 MAC 地址的主机 IP 地址。
- 当同个链路中的所有设备收到 ARP 请求时，会去拆开 ARP 请求包里的内容，如果 ARP 请求包中的目标 IP 地址与自己的 IP 地址一致，那么这个设备就将自己的 MAC 地址塞入 ARP 响应包返回给主机。

操作系统通常会把第一次通过 ARP 获取的 MAC 地址缓存起来，以便下次直接从缓存中找到对应 IP 地址的 MAC 地址。不过，MAC 地址的缓存是有一定期限的，超过这个期限，缓存的内容将被清除。

RARP 协议正好相反，它是已知 MAC 地址求 IP 地址。例如将打印机服务器等小型嵌入式设备接入到网络时就经常会用得到。通常这需要架设一台 RARP 服务器，在这个服务器上注册设备的 MAC 地址及其 IP 地址。然后再将这个设备接入到网络，接着：

- 该设备会发送一条「我的 MAC 地址是XXXX，请告诉我，我的 IP 地址应该是什么」的请求信息。
- RARP 服务器接到这个消息后返回「MAC 地址为 XXXX 的设备，IP 地址为 XXXX」的信息给这个设备。

DHCP

我们的电脑通常都是通过 DHCP 动态获取 IP 地址，大大省去了配 IP 信息繁琐的过程。**DHCP 客户端进程监听的是 68 端口号，DHCP 服务端进程监听的是 67 端口号。**

- 客户端首先发起 **DHCP 发现报文（DHCP DISCOVER）** 的 IP 数据报，由于客户端没有 IP 地址，也不知道 DHCP 服务器的地址，所以使用的是 UDP 广播通信，其使用的广播目的地址是 255.255.255.255（端口 67）并且使用 0.0.0.0（端口 68）作为源 IP 地址。**DHCP 客户端将该 IP 数据报传递给链路层，链路层然后将帧广播到所有的网络中设备。**
- DHCP 服务器收到 DHCP 发现报文时，用 **DHCP 提供报文（DHCP OFFER）** 向客户端做出响应。该报文仍然使用 IP 广播地址 255.255.255.255，该报文信息携带服务器提供可租约的 IP 地址、子网掩码、默认网关、DNS 服务器以及 IP 地址租用期。
- 客户端收到一个或多个服务器的 DHCP 提供报文后，从中选择一个服务器，并向选中的服务器发送 **DHCP 请求报文（DHCP REQUEST）** 进行响应，回显配置的参数。
- 最后，服务端用 **DHCP ACK 报文** 对 DHCP 请求报文进行响应，应答所要求的参数。

一旦客户端收到 DHCP ACK 后，交互便完成了，并且客户端能够在租用期内使用 DHCP 服务器分配的 IP 地址。如果租约的 DHCP IP 地址快到期后，客户端会向服务器发送 DHCP 请求报文：

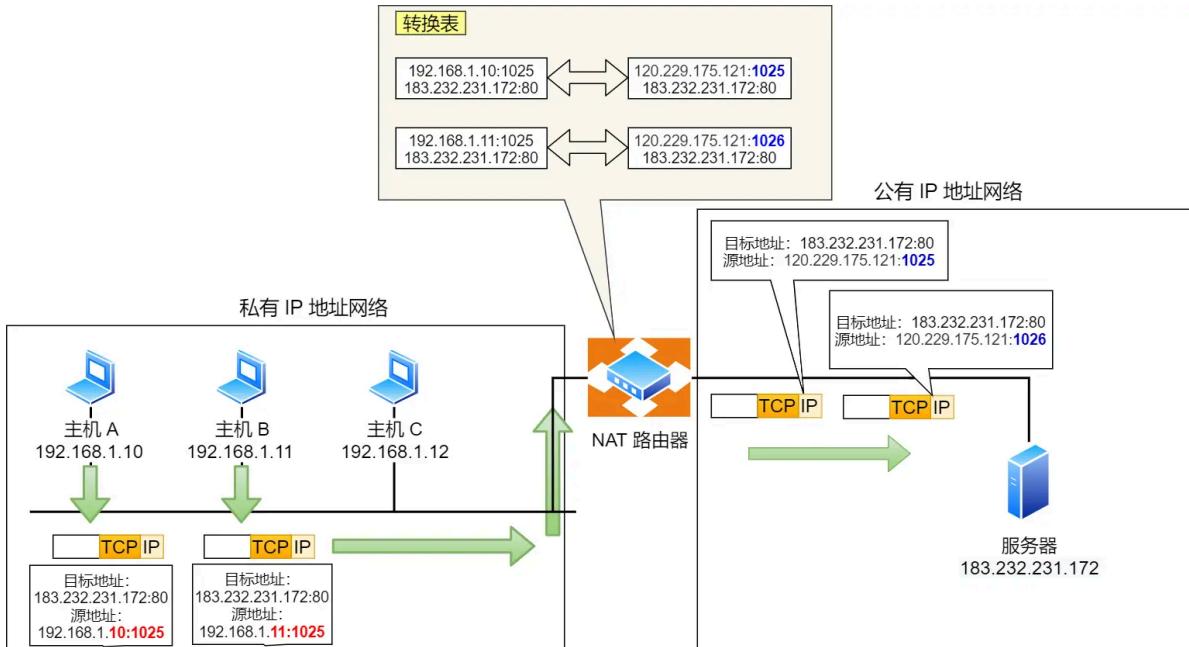
- 服务器如果同意继续租用，则用 **DHCP ACK 报文** 进行应答，客户端就会延长租期。
- 服务器如果不同意继续租用，则用 **DHCP NACK 报文**，客户端就要停止使用租约的 IP 地址。

可以发现，DHCP 交互中，**全程都是使用 UDP 广播通信**。

如果 DHCP 服务器和客户端不是在同一个局域网内，路由器又不会转发广播包，可以**使用 DHCP 中继代理，对不同网段的 IP 地址分配也可以由一个 DHCP 服务器统一进行管理**。

NAT

网络地址与端口转换 NAPT:



图中有两个客户端 192.168.1.10 和 192.168.1.11 同时与服务器 183.232.231.172 进行通信，并且这两个客户端的本地端口都是 1025。此时，**两个私有 IP 地址都转换为公有地址 120.229.175.121，但是以不同的端口号作为区分。**于是，生成一个 NAPT 路由器的转换表，就可以正确地转换地址跟端口的组合，令客户端 A、B 能同时与服务器之间进行通信。

这种转换表在 NAT 路由器上自动生成。例如，在 TCP 的情况下，建立 TCP 连接首次握手时的 SYN 包一经发出，就会生成这个表。而后又随着收到关闭连接时发出 FIN 包的确认应答从表中被删除。由于 NAT/NAPT 都依赖于自己的转换表，因此会有以下的问题：

- 外部无法主动与 NAT 内部服务器建立连接，因为 NAPT 转换表没有转换记录。
- 转换表的生成与转换操作都会产生性能开销。
- 通信过程中，如果 NAT 路由器重启了，所有的 TCP 连接都将被重置。

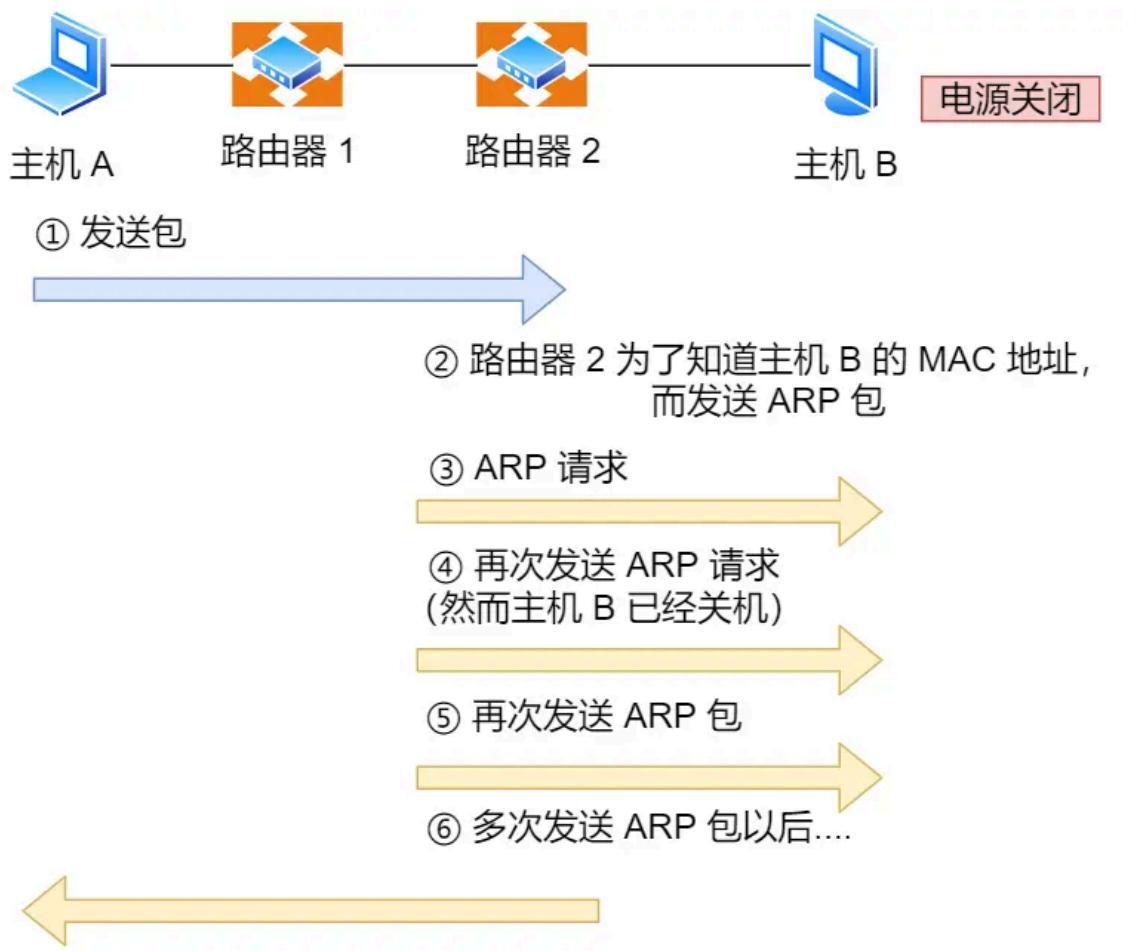
如何解决 NAT 潜在的问题呢？

解决的方法主要有两种方法。

1. 第一种就是改用 IPv6
2. 第二种 NAT 穿透技术：客户端主动从 NAT 设备获取公有 IP 地址，然后自己建立端口映射条目，然后用这个条目对外通信，就不需要 NAT 设备来进行转换了

ICMP

ICMP 报文是封装在 IP 包里面，它工作在网络层。ICMP 主要的功能包括：确认 IP 包是否成功送达目标地址、报告发送过程中 IP 包被废弃的原因和改善网络设置等。在 IP 通信中如果某个 IP 包因为某种原因未能达到目标地址，那么这个具体的原因将由 ICMP 负责通知。



⑦ 由于始终无法到达主机 B，路由器 2 返回一个
ICMP 目标不可达的包给主机 A

如上图例子，主机 A 向主机 B 发送了数据包，由于某种原因，途中的路由器 2 未能发现主机 B 的存在，这时，路由器 2 就会向主机 A 发送一个 ICMP 目标不可达数据包，说明发往主机 B 的包未能成功。

ICMP 的这种通知消息会使用 **IP** 进行发送。因此，从路由器 2 返回的 ICMP 包会按照往常的路由控制先经过路由器 1 再转发给主机 A。收到该 ICMP 包的主机 A 则分解 ICMP 的头部和数据域以后得知具体发生问题的原因。

ICMP 大致可以分为两大类：

- 一类是用于诊断的查询消息，也就是「**查询报文类型**」，也就是 ping
- 另一类是通知出错原因的错误消息，也就是「**差错报文类型**」

ICMP 类型

内容	种类
0 回送应答 (Echo Reply)	查询报文类型
3 目标不可达 (Destination Unreachable)	差错报文类型
4 原点抑制 (Source Quench)	差错报文类型
5 重定向或改变路由 (Redirect)	差错报文类型
8 回送请求 (Echo Request)	查询报文类型
11 超时 (Time Exceeded)	差错报文类型

6 种常见的目标不可达类型的代码：

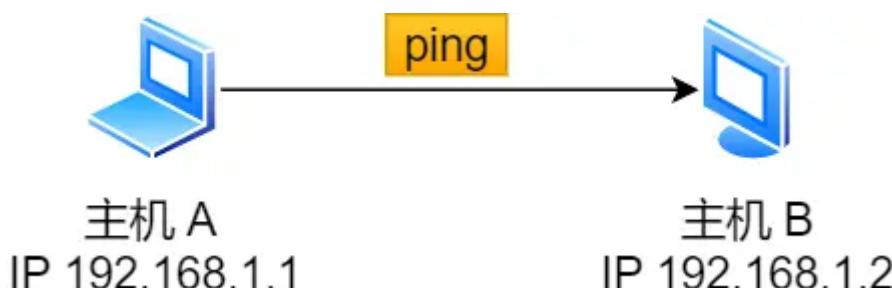
- 网络不可达代码为 0
- 主机不可达代码为 1
- 协议不可达代码为 2
- 端口不可达代码为 3
- 需要进行分片但设置了不分片位代码为 4

IGMP

IGMP 是因特网组管理协议，工作在主机（组播成员）和最后一跳路由之间。

ping —— 查询报文类型的使用

接下来，我们重点来看 `ping` 的发送和接收过程。同个子网下的主机 A 和主机 B，主机 A 执行 `ping` 主机 B 后，我们来看看其间发送了什么？



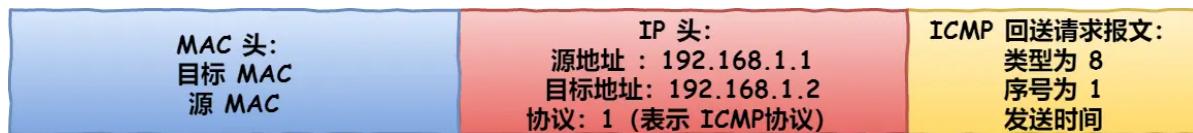
ping 命令执行的时候，源主机首先会构建一个 ICMP 回送请求消息数据包。ICMP 数据包内包含多个字段，最重要的是两个：

- 第一个是**类型**，对于回送请求消息而言该字段为 8；
- 另外一个是**序号**，主要用于区分连续 ping 的时候发出的多个数据包。

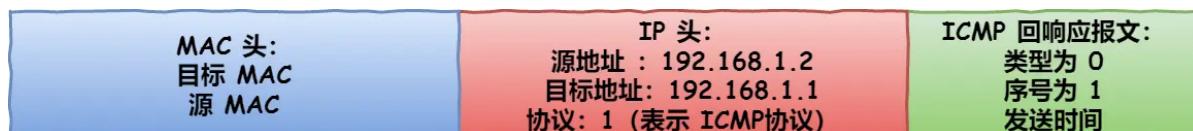
每发出一个请求数据包，序号会自动加 1。为了能够计算往返时间 RTT，它会在报文的数据部分插入发送时间。然后，由 ICMP 协议将这个数据包连同地址 192.168.1.2 一起交给 IP 层。IP 层将以 192.168.1.2 作为**目的地址**，本机 IP 地址作为**源地址**，**协议字段设置为 1 表示是 ICMP 协议**，再加上一些其他控制信息，构建一个 IP 数据包。



接下来，需要加入 MAC 头。如果在本地 ARP 映射表中查找出 IP 地址 192.168.1.2 所对应的 MAC 地址，则可以直接使用；如果没有，则需要发送 ARP 协议查询 MAC 地址，获得 MAC 地址后，由数据链路层构建一个数据帧，目的地址是 IP 层传过来的 MAC 地址，源地址则是本机的 MAC 地址；还要附加上一些控制信息，依据以太网的介质访问规则，将它们传送出去。



主机 B 收到这个数据帧后，先检查它的目的 MAC 地址，并和本机的 MAC 地址对比，如符合，则接收，否则就丢弃。接收后检查该数据帧，将 IP 数据包从帧中提取出来，交给本机的 IP 层。同样，IP 层检查后，将有用的信息提取后交给 ICMP 协议。主机 B 会构建一个**ICMP 回送响应消息**数据包，回送响应数据包的**类型**字段为 0，**序号**为接收到的请求数据包中的序号，然后再发送出去给主机 A。



在规定的时候内，源主机如果没有接到 ICMP 的应答包，则说明目标主机不可达；如果接收到了 ICMP 回声响应消息，则说明目标主机可达。此时，源主机将用当前时刻减去该数据包最初从源主机上发出的时刻，就是 ICMP 数据包的时间延迟。ping 这个程序是**使用了 ICMP 里面的 ECHO REQUEST (类型为 8) 和 ECHO REPLY (类型为 0)**。

traceroute —— 差错报文类型的使用

有一款充分利用 ICMP 差错报文类型的应用叫做 traceroute。

1. traceroute 作用一：traceroute 的第一个作用就是**故意设置特殊的 TTL，来追踪去往目的地时沿途经过的路由器**。traceroute 的参数指向某个**目的 IP 地址**：

```
traceroute 192.168.1.100
```

它的原理就是利用 IP 包的**生存期限**从 1 开始按照顺序递增的同时发送 UDP 包，强制接收 ICMP 超时消息的一种方法。比如，将 TTL 设置为 1，则遇到第一个路由器，就牺牲了，接着返回 ICMP 差错报文网络包，类型是**时间超时**。接下来将 TTL 设置为 2，第一个路由器过了，遇到第二个路由器也牺牲了，也同时返回了 ICMP 差错报文数据包，如此往复，直到到达目的主机。

这样的过程，traceroute 就可以拿到了所有的路由器 IP。

当然有的路由器根本就不会返回这个 ICMP，所以对于有的公网地址，是看不到中间经过的路由的。

发送方如何知道发出的 UDP 包是否到达了目的主机呢？

traceroute 在发送 `UDP` 包时，会填入一个**不可能的端口号**值作为 `UDP` 目标端口号：33434。然后对于每个下一个探针，它都会增加一个，这些端口都是通常认为不会被使用，不过，没有人知道当某些应用程序监听此类端口时会发生什么。当目的主机，收到 `UDP` 包后，会返回 `ICMP` 差错报文消息，但这个差错报文消息的类型是「**端口不可达**」。所以，**当差错报文类型是端口不可达时，说明发送方发出的 `UDP` 包到达了目的主机。**

2. traceroute 作用二

traceroute 还有一个作用是**故意设置不分片，从而确定路径的 MTU**。这样做的目的是为了**路径MTU发现**。因为有的时候我们并不知道路由器的 `MTU` 大小，以太网的数据链路上的 `MTU` 通常是 1500 字节，但是非以太网的 `MTU` 值就不一样了，所以我们要知道 `MTU` 的大小，从而控制发送的包大小。

它的工作原理如下：首先在发送端主机发送 `IP` 数据报时，将 `IP` 包首部的**分片禁止标志位设置为 1**。根据这个标志位，途中的路由器不会对大数据包进行分片，而是将包丢弃。随后，通过一个 `ICMP` 的**不可达消息将数据链路上 MTU 的值一起给发送主机**，不可达消息的类型为「**需要进行分片但设置了不分片位**」。发送主机端每次收到 `ICMP` 差错报文时就**减少包的大小**，以此来定位一个合适的 `MTU` 值，以便能到达目标主机。

TCP发数据和ping的区别

在 TCP 传输中创建的方式是 `socket(AF_INET, SOCK_STREAM, 0)`，其中 `AF_INET` 表示将使用 IPV4 里 `host:port` 的方式去解析待会你输入的网络地址。`SOCK_STREAM` 是指使用面向字节流的 TCP 协议，**工作在传输层**。创建好了 `socket` 之后，就可以愉快的把要传输的数据写到这个文件里。调用 `socket` 的 `sendto` 接口的过程中进程会从**用户态进入到内核态**，最后会调用到 `sock_sendmsg` 方法。

`ping`，整个过程也基本跟 `TCP` 发数据类似，差异的地方主要在于，创建 `socket` 的时候用的是 `socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)`，`SOCK_RAW` 是原始套接字，**工作在网络层**，所以构建 `ICMP`（网络层协议）的数据，是再合适不过了。`ping` 在进入内核态后最后也是调用的 `sock_sendmsg` 方法，进入到网络层后加上**ICMP和IP头**后，数据链路层加上**MAC头**，也是顺着网卡发出。

为什么断网了还能 ping 通 127.0.0.1

当发现目标IP是回环地址时，就会选择本地网卡。本地网卡，其实就是一个“假网卡”，它不像“真网卡”那样有个 `ring buffer` 什么的，“假网卡”会把数据推到一个叫 `input_pkt_queue` 的链表中。这个链表，其实是所有网卡共享的，上面挂着发给本机的各种消息。消息被发送到这个链表后，会再触发一个软中断。专门处理软中断的工具人“`ksoftirqd`”（这是个内核线程），它在收到软中断后就会立马去链表里把消息取出，然后顺着数据链路层、网络层等层层往上传递最后给到应用程序。

`ping` 回环地址和**通过TCP等各种协议发送数据到回环地址**都是走这条路径。整条路径从发到收，都没有经过“真网卡”。之所以127.0.0.1叫本地回环地址，可以理解为，消息发出到这个地址上的话，就不会出网络，在本机打个转就又回来了。所以断网，依然能 `ping` 通 127.0.0.1。

ping回环地址和ping本机地址有什么区别

`ping` 本机IP 跟 `ping` 回环地址一样，相关的网络数据，都是走的 `lo0`，本地回环接口，也就是前面提到的“**假网卡**”。只要走了本地回环接口，那数据都不会发送到网络中，在本机网络协议栈中兜一圈，就发回来了。因此 `ping`回环地址和`ping`本机地址没有区别。

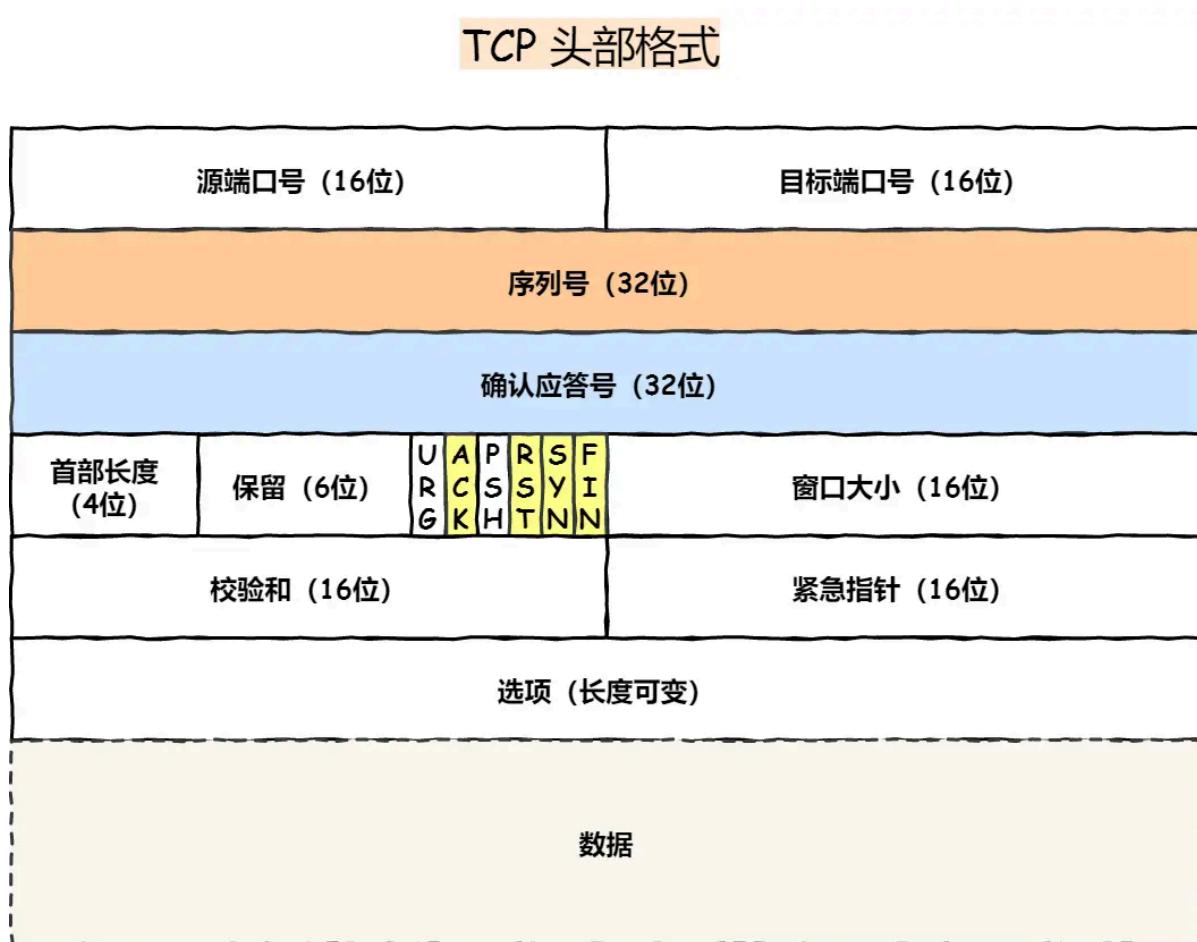
127.0.0.1 和 localhost 以及 0.0.0.0 有区别吗

`localhost` 就不叫 `IP`，它是一个域名，就跟 `"baidu.com"` 是一个形式的东西，只不过默认会把它解析为 `127.0.0.1`，当然这可以在 `/etc/hosts` 文件下进行修改。所以默认情况下，使用 `localhost` 跟使用 `127.0.0.1` 确实是没区别的。

其次就是 `0.0.0.0`，执行 `ping 0.0.0.0`，是会失败的，因为它在 `IPV4` 中表示的是无效的**目标地址**。但它还是很有用处的，回想下，我们启动服务器的时候，一般会 `listen` 一个 IP 和端口，等待客户端的连接。如果此时 `listen` 的是本机的 `0.0.0.0`，那么它表示本机上的**所有IPV4地址**。举个例子。刚刚提到的 `127.0.0.1` 和 `192.168.31.6`，都是本机的IPV4地址，如果监听 `0.0.0.0`，那么用上面两个地址，都能访问到这个服务器。当然，客户端 `connect` 时，不能使用 `0.0.0.0`。必须指明要连接哪个服务器IP。

TCP 头格式有哪些？

我们先来看看 TCP 头的格式，标注颜色的表示与本文关联比较大的字段，其他字段不做详细阐述。



序列号：在建立连接时由计算机生成的随机数作为其初始值，通过 SYN 包传给接收端主机，每发送一次数据，就「累加」一次该「数据字节数」的大小。用来解决网络包乱序问题。**

确认应答号：指下一次「期望」收到的数据的序列号，发送端收到这个确认应答以后可以认为在这个序号以前的数据都已经被正常接收。用来解决丢包的问题。**

控制位：

- **ACK**: 该位为 1 时，「确认应答」的字段变为有效，TCP 规定除了最初建立连接时的 SYN 包之外该位必须设置为 1。
- **RST**: 该位为 1 时，表示 TCP 连接中出现异常必须强制断开连接。
- **SYN**: 该位为 1 时，表示希望建立连接，并在其「序列号」的字段进行序列号初始值的设定。
- **FIN**: 该位为 1 时，表示今后不会再有数据发送，希望断开连接。当通信结束希望断开连接时，通信双方的主机之间就可以相互交换 FIN 位为 1 的 TCP 段。

为什么需要 TCP 协议？TCP 工作在哪一层？

IP 层是「不可靠」的，它不保证网络包的交付、不保证网络包的按序交付、也不保证网络包中的数据的完整性。如果需要保障网络数据包的可靠性，那么就需要由上层（传输层）的 TCP 协议来负责。因为 TCP 是一个工作在传输层的可靠数据传输的服务，它能确保接收端接收的网络包是无损坏、无间隔、非冗余和按序的。

什么是 TCP ?

TCP 是面向连接的、可靠的、基于字节流的传输层通信协议。



- **面向连接**: 一定是「一对一」才能连接，不能像 UDP 协议可以一个主机同时向多个主机发送消息，也就是一对多是无法做到的；
- **可靠的**: 无论的网络链路中出现了怎样的链路变化，TCP 都可以保证一个报文一定能够到达接收端；
- **字节流**: 用户消息通过 TCP 协议传输时，消息可能会被操作系统「分组」成多个的 TCP 报文，如果接收方的程序如果不知道「消息的边界」，是无法读出一个有效的用户消息的。并且 TCP 报文是「有序的」，当「前一个」TCP 报文没有收到的时候，即使它先收到了后面的 TCP 报文，那么也不能扔给应用层去处理，同时对「重复」的 TCP 报文会自动丢弃。

什么是 TCP 连接？

「连接」简单来说就是，用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，包括 **Socket**、**序列号** 和 **窗口大小** 称为连接。所以我们可以知道，建立一个 TCP 连接是需要客户端与服务端达成上述三个信息的共识。

- **Socket**: 由 IP 地址和端口号组成
- **序列号**: 用来解决乱序问题等
- **窗口大小**: 用来做流量控制

如何唯一确定一个 TCP 连接呢？

TCP 四元组可以唯一的确定一个连接，四元组包括如下：

- 源地址
- 源端口
- 目的地址
- 目的端口

源地址和目的地址的字段（32 位）是在 IP 头部中，作用是通过 IP 协议发送报文给对方主机。

源端口和目的端口的字段（16 位）是在 TCP 头部中，作用是告诉 TCP 协议应该把报文发给哪个进程。

有一个 IP 的服务端监听了一个端口，它的 TCP 的最大连接数是多少？

服务端通常固定在某个本地端口上监听，等待客户端的连接请求。因此，客户端 IP 和端口是可变的，其理论值计算公式如下：

最大 TCP 连接数 = 客户端的IP 数 × 客户端的端口数

当然，服务端最大并发 TCP 连接数远不能达到理论上限，会受以下因素影响：

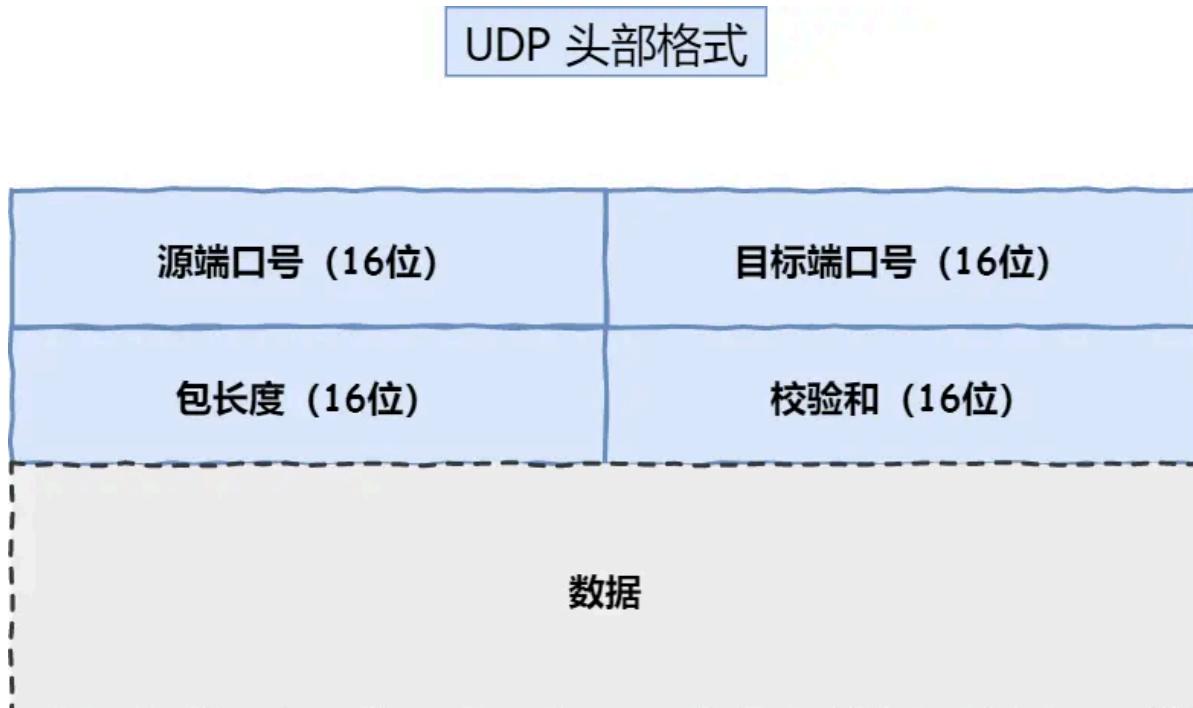
- 文件描述符限制

每个 TCP 连接都是一个文件，如果文件描述符被占满了，会发生 Too many open files。Linux 对可打开的文件描述符的数量分别作了三个方面的限制：

- **系统级**: 当前系统可打开的最大数量，通过 `cat /proc/sys/fs/file-max` 查看；
- **用户级**: 指定用户可打开的最大数量，通过 `cat /etc/security/limits.conf` 查看；
- **进程级**: 单个进程可打开的最大数量，通过 `cat /proc/sys/fs/nr_open` 查看；
- **内存限制**，每个 TCP 连接都要占用一定内存，操作系统的内存是有限的，如果内存资源被占满后，会发生 OOM。

UDP 和 TCP 有什么区别呢？分别的应用场景是？

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。UDP 协议真的非常简单，头部只有 8 个字节（64 位）， UDP 的头部格式如下：



- 目标和源端口：主要是告诉 UDP 协议应该把报文发给哪个进程。
- 包长度：该字段保存了 UDP 首部的长度跟数据的长度之和。
- 校验和：校验和是为了提供可靠的 UDP 首部和数据而设计，防止收到在网络传输中受损的 UDP 包。

TCP 和 UDP 区别：

1. 连接

- TCP 是面向连接的传输层协议，传输数据前先要建立连接。
- UDP 是不需要连接，即刻传输数据。

2. 服务对象

- TCP 是一对一的两点服务，即一条连接只有两个端点。
- **UDP 支持一对一、一对多、多对多的交互通信**

3. 可靠性

- TCP 是可靠交付数据的，数据可以无差错、不丢失、不重复、按序到达。

- UDP 是尽最大努力交付，不保证可靠交付数据。但是我们可以基于 UDP 传输协议实现一个可靠的传输协议，比如 QUIC 协议

4. 拥塞控制、流量控制

- TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。
- UDP 则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率。

5. 首部开销

- **TCP 首部长度较长，会有一定的开销**，首部在没有使用「选项」字段时是 20 个字节，如果使用了「选项」字段则会变长的。
- UDP 首部只有 8 个字节，并且是固定不变的，**开销较小**。

6. 传输方式

- **TCP 是流式传输，没有边界，但保证顺序和可靠**。
- **UDP 是一个包一个包的发送，是有边界的，但可能会丢包和乱序**。

7. 分片不同

- **TCP 的数据大小如果大于 MSS 大小，则会在传输层进行分片**，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这个分片。
- **UDP 的数据大小如果大于 MTU 大小，则会在 IP 层进行分片**，目标主机收到后，在 IP 层组装完数据，接着再传给传输层。

TCP 和 UDP 应用场景：

由于 TCP 是面向连接，能保证数据的可靠性交付，因此经常用于：

- **FTP** 文件传输；
- **HTTP / HTTPS**；

由于 UDP 面向无连接，它可以随时发送数据，再加上 UDP 本身的处理既简单又高效，因此经常用于：

- 包总量较少的通信，如 **DNS**、**SNMP** 等；
- 视频、音频等多媒体通信；
- 广播通信；

为什么 UDP 头部没有「首部长度」字段，而 TCP 头部有「首部长度」字段呢？

原因是 **TCP 有可变长的「选项」字段**，而 **UDP 头部长度则是不会变化的**，无需多一个字段去记录 UDP 的首部长度。

为什么 UDP 头部有「包长度」字段，而 TCP 头部则没有「包长度」字段呢？

先说说 TCP 是如何计算负载数据长度：

TCP 数据的长度 = IP 总长度 - IP 首部长度 - TCP 首部长度

其中 IP 总长度和 IP 首部长度，在 IP 头部格式是已知的。TCP 首部长度，则是在 TCP 头部格式已知的，所以就可以求得 TCP 数据的长度。

大家这时就奇怪了问：“UDP 也是基于 IP 层的呀，那 UDP 的数据长度也可以通过这个公式计算呀？为何还要有「包长度」呢？”这么一问，确实感觉 UDP 的「包长度」是冗余的。我查阅了很多资料，我觉得有两个比较靠谱的说法：

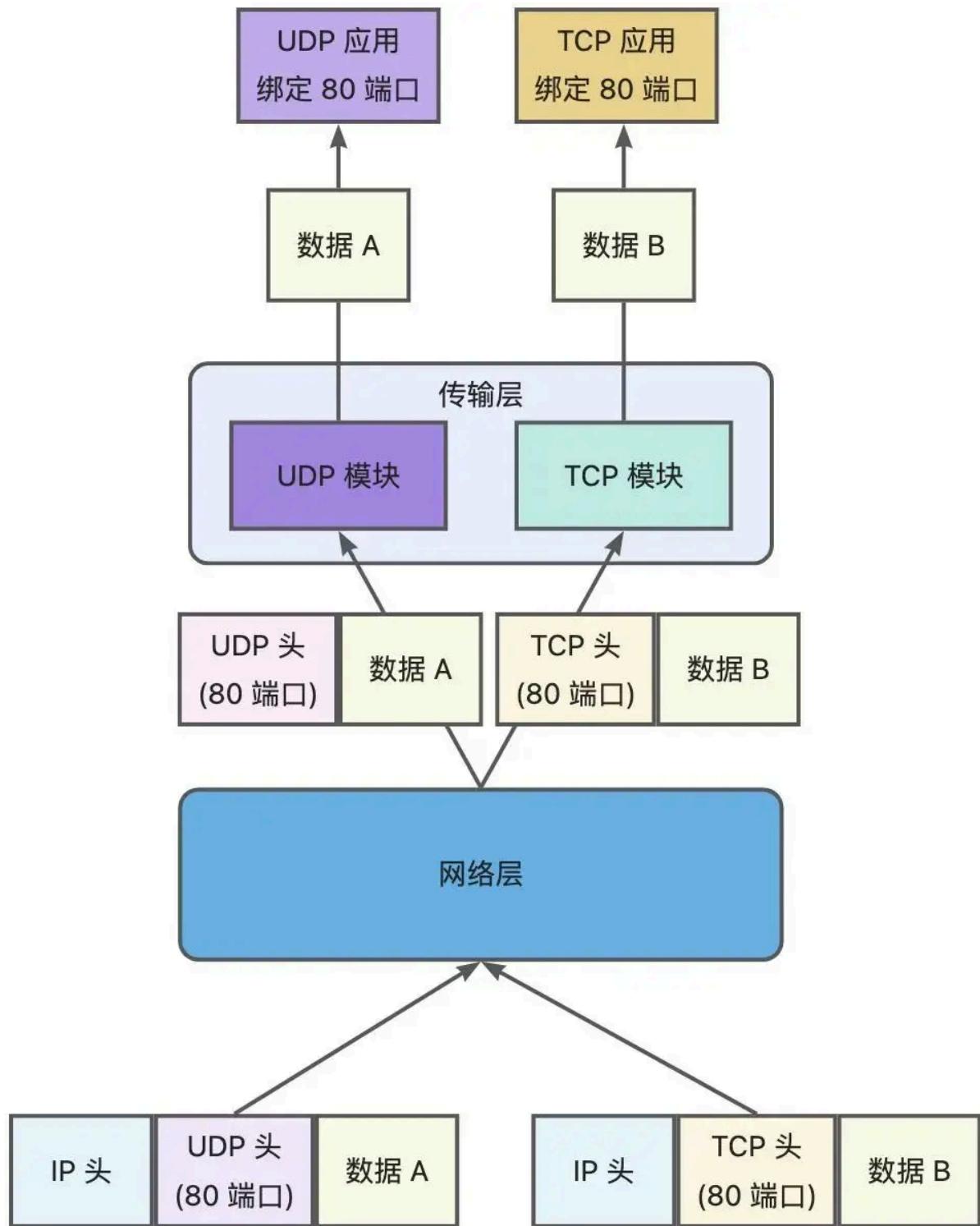
- 第一种说法：因为为了网络设备硬件设计和处理方便，首部长度需要是 4 字节的整数倍。如果去掉 UDP 的「包长度」字段，那 UDP 首部长度就不是 4 字节的整数倍了，所以我觉得这可能是为了补充 UDP 首部长度是 4 字节的整数倍，才补充了「包长度」字段。
- 第二种说法：如今的 UDP 协议是基于 IP 协议发展的，而当年可能并非如此，依赖的可能是别的不提供自身报文长度或首部长度的网络层协议，因此 UDP 报文首部需要有长度字段以供计算。

TCP 和 UDP 可以使用同一个端口吗？

答案：可以的。

在数据链路层中，通过 MAC 地址来寻找局域网中的主机。在网际层中，通过 IP 地址来寻找网络中互连的主机或路由器。在传输层中，需要通过端口进行寻址，来识别同一计算机中同时通信的不同应用程序。

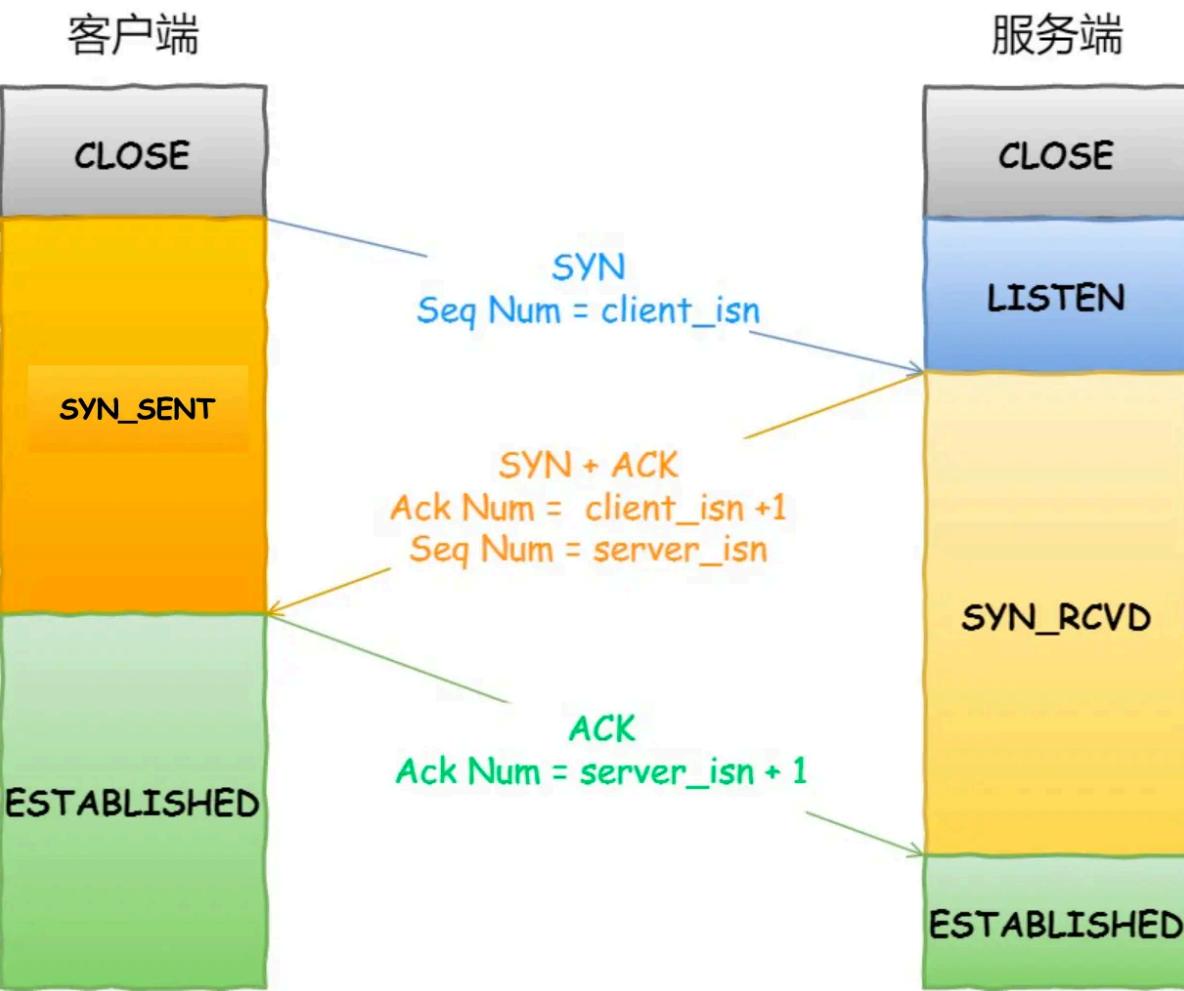
所以，传输层的「端口号」的作用，是为了区分同一个主机上不同应用程序的数据包。传输层有两个传输协议分别是 TCP 和 UDP，在内核中是两个完全独立的软件模块。当主机收到数据包后，可以在 IP 包头的「协议号」字段知道该数据包是 TCP/UDP，所以可以根据这个信息确定送给哪个模块（TCP/UDP）处理，送给 TCP/UDP 模块的报文根据「端口号」确定送给哪个应用程序处理。



因此，TCP/UDP各自的端口号也相互独立，如TCP有一个80号端口，UDP也可以有一个80号端口，二者并不冲突。

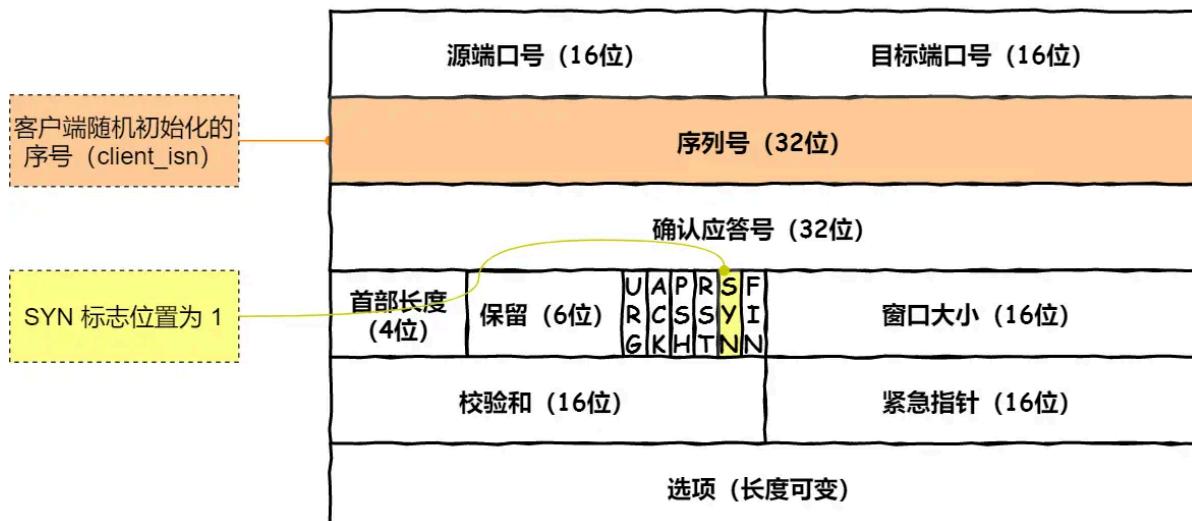
TCP三次握手过程是怎样的？

TCP是面向连接的协议，所以使用TCP前必须先建立连接，而建立连接是通过三次握手来进行的。三次握手的过程如下图：



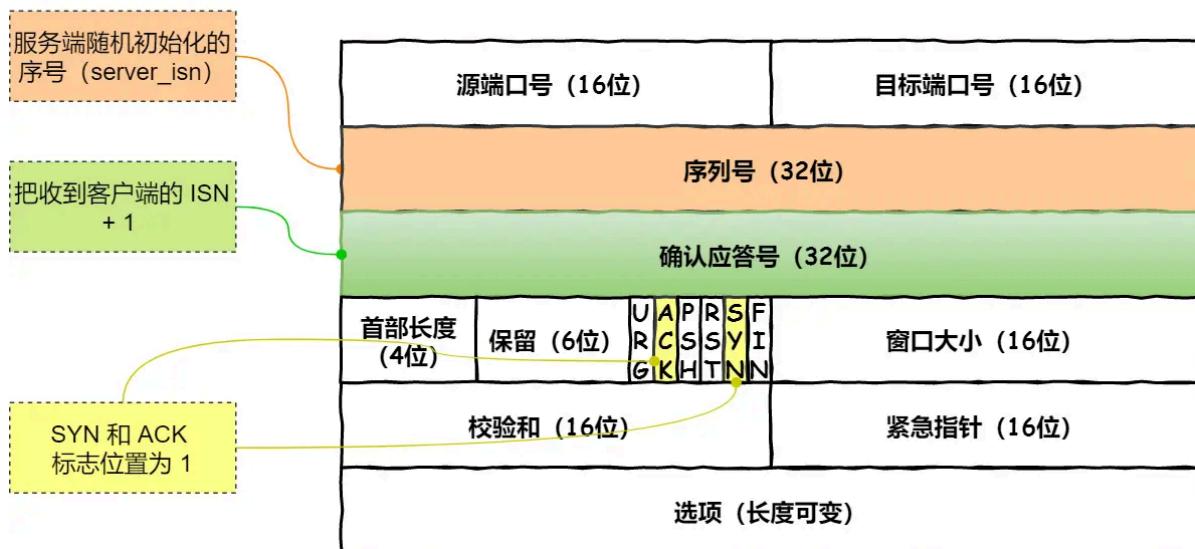
- 一开始，客户端和服务端都处于 **CLOSE** 状态。先是服务端主动监听某个端口，处于 **LISTEN** 状态

三次握手的第一个报文： SYN 报文



- 客户端会随机初始化序号 (`client_isn`)，将此序号置于 TCP 首部的「序号」字段中，同时把 **SYN** 标志位置为 **1**，表示 **SYN** 报文。接着把第一个 **SYN** 报文发送给服务端，表示向服务端发起连接，该报文不包含应用层数据，之后客户端处于 **SYN-SENT** 状态。

三次握手的第二个报文： SYN + ACK 报文



- 服务端收到客户端的 `SYN` 报文后，首先服务端也随机初始化自己的序号 (`server_isn`)，将此序号填入 TCP 首部的「序号」字段中，其次把 TCP 首部的「确认应答号」字段填入 `client_isn + 1`，接着把 `SYN` 和 `ACK` 标志位置为 `1`。最后把该报文发给客户端，**该报文也不包含应用层数据**，之后服务端处于 `SYN-RCVD` 状态。

三次握手的第三个报文： ACK 报文



- 客户端收到服务端报文后，还要向服务端回应最后一个应答报文，首先该应答报文 TCP 首部 `ACK` 标志位置为 `1`，其次「确认应答号」字段填入 `server_isn + 1`，最后把报文发送给服务端，**这次报文可以携带客户到服务端的数据，之后客户端处于 ESTABLISHED 状态**。
- 服务端收到客户端的应答报文后，也进入 `ESTABLISHED` 状态。

从上面的过程可以发现**第三次握手是可以携带数据的，前两次握手是不可以携带数据的**，这也是面试常问的题。一旦完成三次握手，双方都处于 `ESTABLISHED` 状态，此时连接就已建立完成，客户端和服务端就可以相互发送数据了。

如何在 Linux 系统中查看 TCP 状态？

TCP 的连接状态查看，在 Linux 可以通过 `netstat -napt` 命令查看。

Active Internet connections (servers and established)					
Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	::ffff:192.168.3.100:80	::ffff:192.168.3.20:55288	ESTABLISHED

图中注释：TCP 协议、源地址 + 端口、目标地址 + 端口、连接状态、Web 服务的进程 PID 和进程名称

为什么是三次握手？不是两次、四次？

相信大家比较常回答的是：“因为三次握手才能保证双方具有接收和发送的能力。”这回答是没问题，但这回答是片面的，并没有说出主要的原因。**为什么三次握手才可以初始化 Socket、序列号和窗口大小并建立 TCP 连接。**

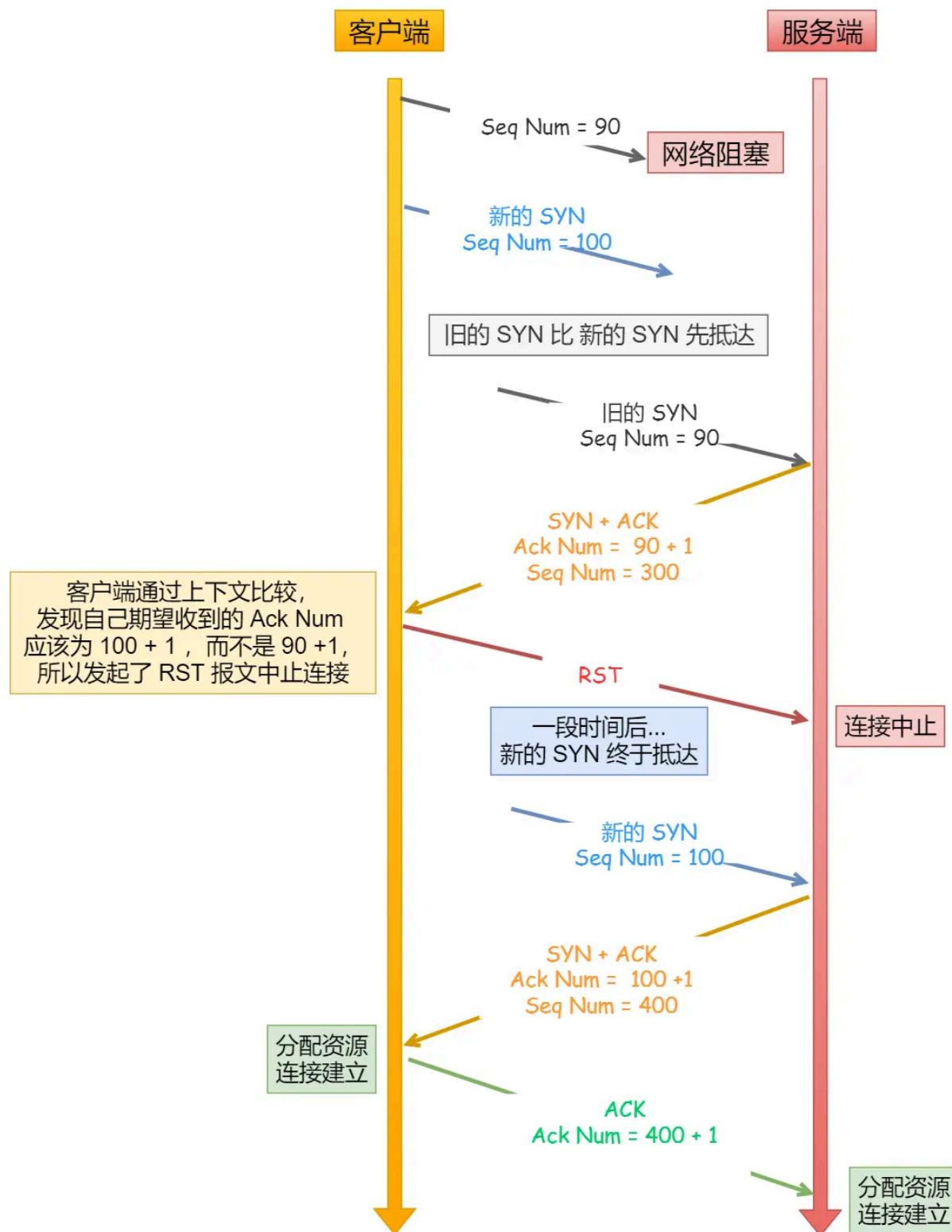
接下来，以三个方面分析三次握手的原因：

- 三次握手才可以阻止重复历史连接的初始化（主要原因）
- 三次握手才可以同步双方的初始序列号
- 三次握手才可以避免资源浪费

原因一：避免历史连接，三次握手的首要原因是防止旧的重复连接初始化造成混乱。

我们考虑一个场景，客户端先发送了 SYN ($seq = 90$) 报文，然后客户端宕机了，而且这个 SYN 报文还被网络阻塞了，服务端并没有收到，接着客户端重启后，又重新向服务端建立连接，发送了 SYN ($seq = 100$) 报文（注意！不是重传 SYN，重传的 SYN 的序列号是一样的）。

三次握手 避免历史连接



客户端连续发送多次 SYN (都是同一个四元组) 建立连接的报文，在**网络拥堵**情况下：

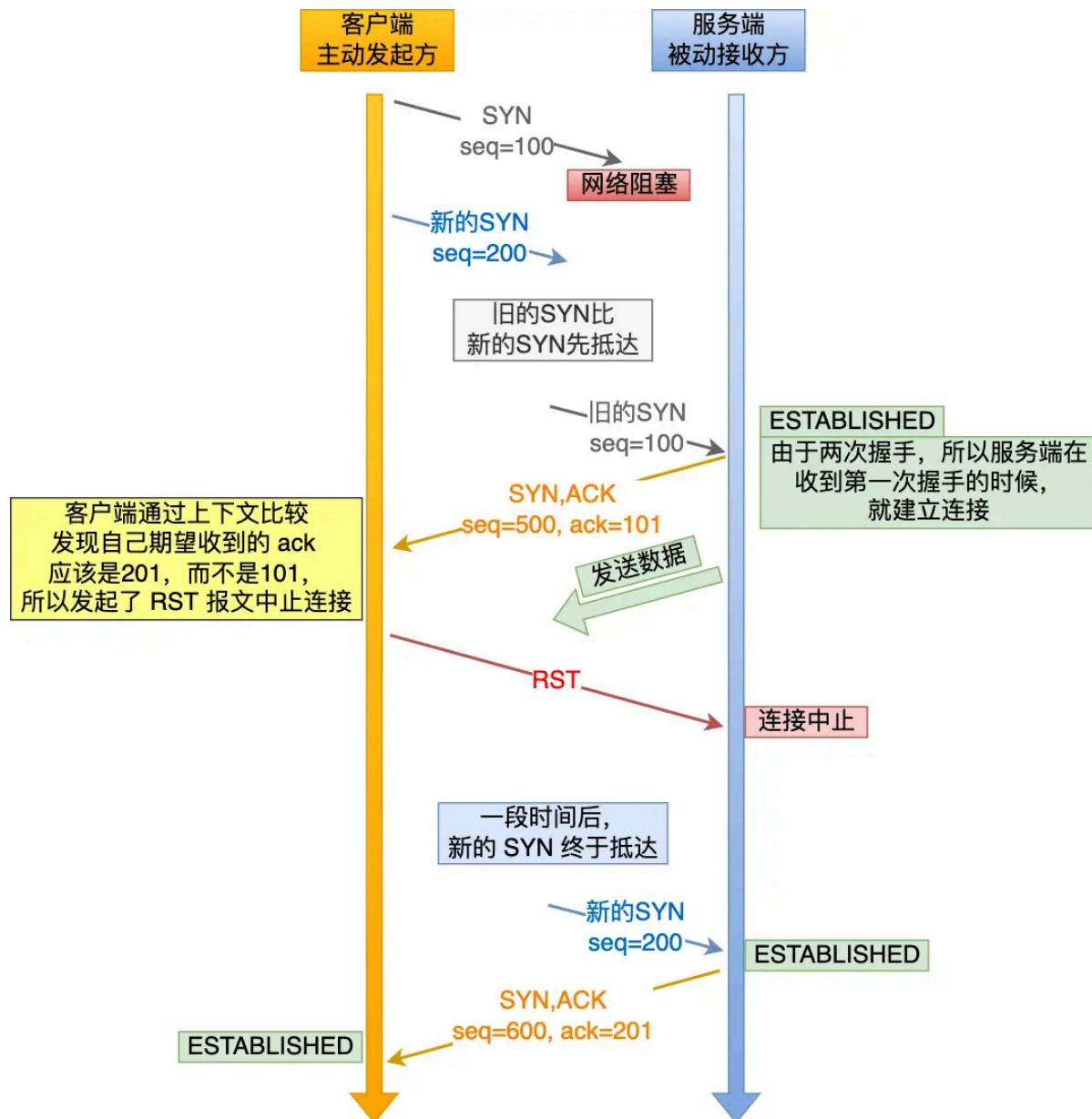
- 一个「旧 SYN 报文」比「最新的 SYN」报文早到达了服务端，那么此时服务端就会回一个 SYN + ACK 报文给客户端，此报文中的确认号是 91 (90+1)。
- 客户端收到后，发现自己期望收到的确认号应该是 100 + 1，而不是 90 + 1，于是就会回 RST 报文（如果只有2次连接的话，那么就会直接建立连接了）。
- 服务端收到 RST 报文后，就会释放连接。

- 后续最新的 SYN 抵达了服务端后，客户端与服务端就可以正常的完成三次握手了。

上述中的「旧 SYN 报文」称为历史连接，TCP 使用三次握手建立连接的最主要原因是防止「历史连接」初始化了连接。

有很多人问，如果服务端在收到 RST 报文之前，先收到了「新 SYN 报文」，也就是服务端收到客户端报文的顺序是：「旧 SYN 报文」->「新 SYN 报文」，此时会发生什么？当服务端第一次收到 SYN 报文，也就是收到「旧 SYN 报文」时，就会回复 **SYN + ACK** 报文给客户端，此报文中的确认号是 $91 + (90+1)$ 。然后这时再收到「新 SYN 报文」时，就会回 [Challenge Ack \(opens new window\)](#) 报文给客户端，这个 ack 报文并不是确认收到「新 SYN 报文」的，而是上一次的 ack 确认号，也就是 $91 + (90+1)$ 。所以客户端收到此 ACK 报文时，发现自己期望收到的确认号应该是 101，而不是 91，于是就会回 RST 报文。

如果是两次握手连接，就无法阻止历史连接，那为什么 TCP 两次握手为什么无法阻止历史连接呢？我先直接说结论，主要是因为在两次握手的情况下，服务端没有中间状态给客户端来阻止历史连接，导致服务端可能建立一个历史连接，造成资源浪费。



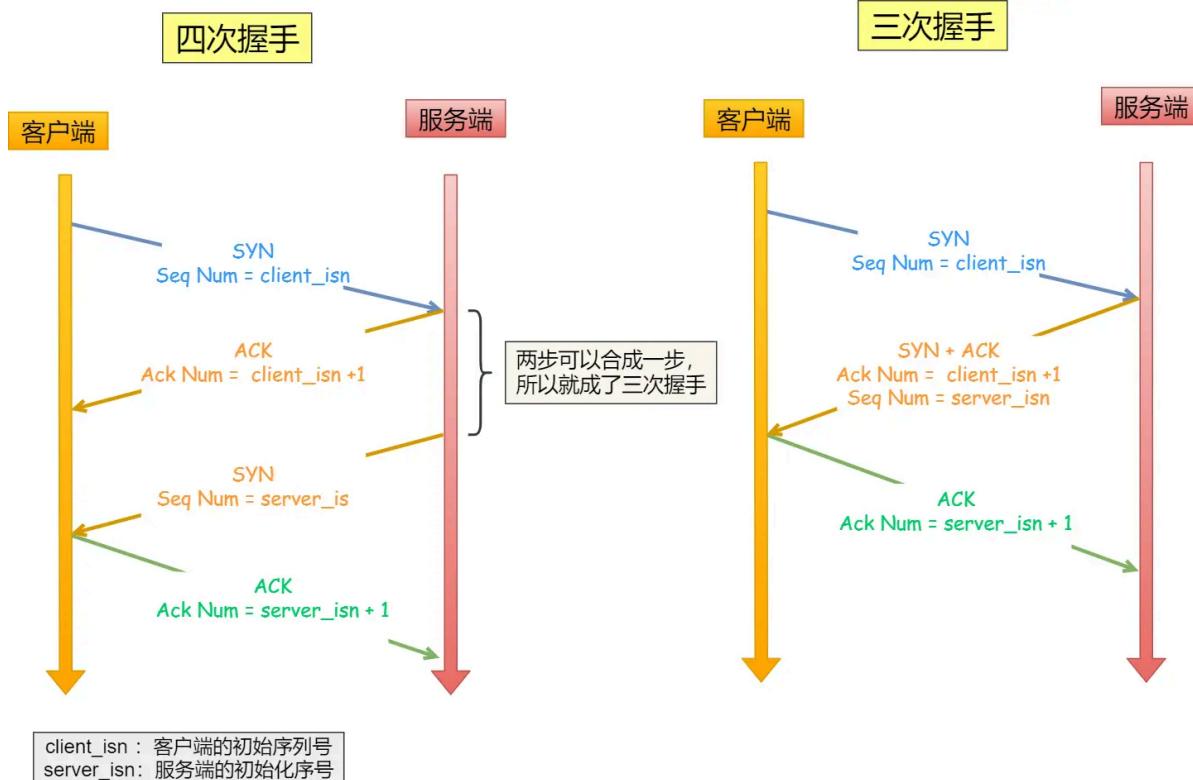
可以看到，如果采用两次握手建立 TCP 连接的场景下，服务端在向客户端发送数据前，并没有阻止掉历史连接，导致服务端建立了一个历史连接，又白白发送了数据，妥妥地浪费了服务端的资源。要解决这种现象，最好就是在服务端发送数据前，也就是建立连接之前，要阻止掉历史连接，这样就不会造成资源浪费，而要实现这个功能，就需要三次握手。所以，TCP 使用三次握手建立连接的最主要原因是防止「历史连接」初始化了连接。

原因二：同步双方初始序列号

TCP 协议的通信双方，都必须维护一个「序列号」，序列号是可靠传输的一个关键因素，它的作用：

- 接收方可以去除重复的数据；
- 接收方可以根据数据包的序列号按序接收；
- 可以标识发送出去的数据包中，哪些是已经被对方收到的（通过 ACK 报文中的序列号知道）；

可见，序列号在 TCP 连接中占据着非常重要的作用，所以当客户端发送携带「初始序列号」的 SYN 报文的时候，需要服务端回一个 ACK 应答报文，表示客户端的 SYN 报文已被服务端成功接收，那当服务端发送「初始序列号」给客户端的时候，依然也要得到客户端的应答回应，**这样来回一回，才能确保双方的初始序列号能被可靠的同步。**



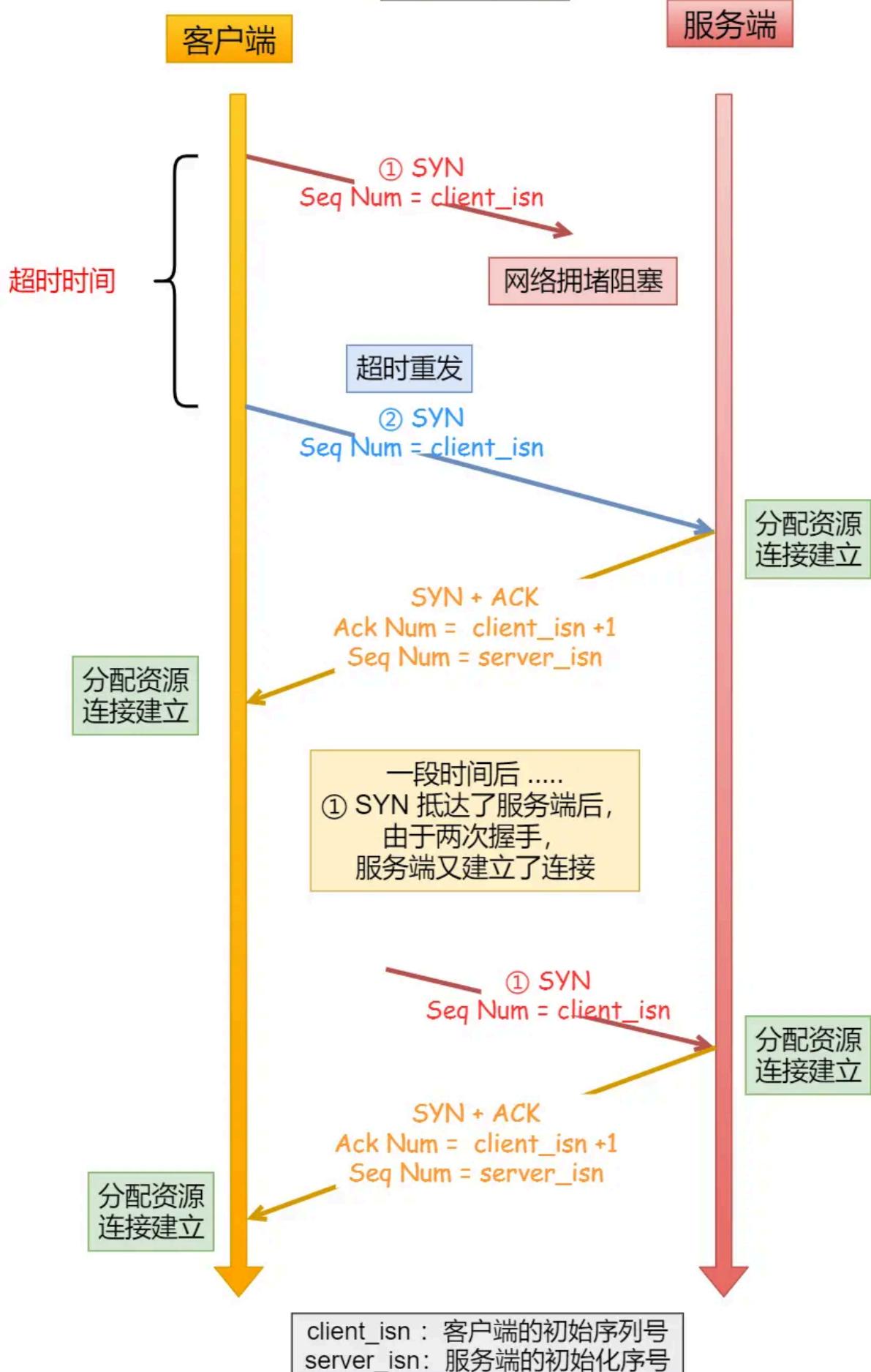
四次握手其实也能够可靠的同步双方的初始化序号，但由于**第二步和第三步可以优化成一步**，所以就成为了「三次握手」。而两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。

原因三：避免资源浪费

如果只有「两次握手」，当客户端发送的 SYN 报文在网络中阻塞，客户端没有接收到 ACK 报文，就会重新发送 SYN，由于没有第三次握手，服务端不清楚客户端是否收到了自己回复的 ACK 报文，所以服务端每收到一个 SYN 就只能先主动建立一个连接，这会造成什么情况呢？

如果客户端发送的 SYN 报文在网络中阻塞了，重复发送多次 SYN 报文，那么服务端在收到请求后就会建立多个冗余的无效链接，造成不必要的资源浪费。

两次握手



即两次握手会造成消息滞留情况下，服务端重复接受无用的连接请求 SYN 报文，而造成重复分配资源。

TCP 建立连接时，通过三次握手能防止历史连接的建立，能减少双方不必要的资源开销，能帮助双方同步初始化序列号。序列号能够保证数据包不重复、不丢弃和按序传输。

不使用「两次握手」和「四次握手」的原因：

- 「两次握手」：无法防止历史连接的建立，会造成双方资源的浪费，也无法可靠的同步双方序列号；
- 「四次握手」：三次握手就已经理论上最少可靠连接建立，所以不需要使用更多的通信次数。

为什么每次建立 TCP 连接时，初始化的序列号都要求不一样呢？

主要原因有两个方面：

- 为了防止历史报文被下一个相同四元组的连接接收（主要方面）；如果每次建立连接，客户端和服务端的初始化序列号都是一样的话，很容易出现历史报文被下一个相同四元组的连接接收的问题。
- 为了安全性，防止黑客伪造的相同序列号的 TCP 报文被对方接收；

初始序列号 ISN 是如何随机产生的？

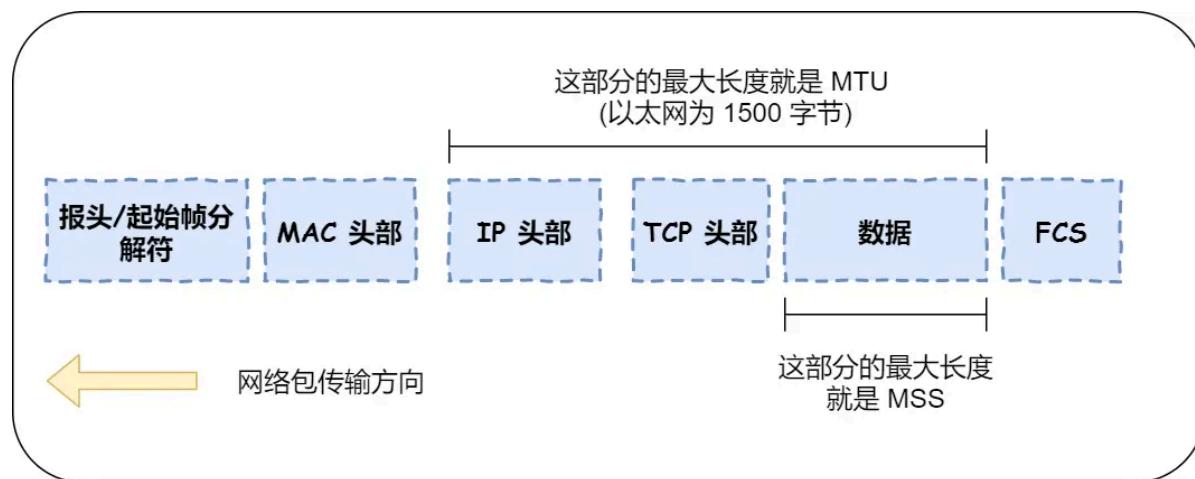
起始 **ISN** 是基于时钟的，每 4 微秒 + 1，转一圈要 4.55 个小时。RFC793 提到初始化序列号 ISN 随机生成算法： $ISN = M + F(\text{localhost}, \text{localport}, \text{remotehost}, \text{remoteport})$ 。

- **M** 是一个计时器，这个计时器每隔 4 微秒加 1。
- **F** 是一个 Hash 算法，根据源 IP、目的 IP、源端口、目的端口生成一个随机数值。要保证 Hash 算法不能被外部轻易推算得出，用 MD5 算法是一个比较好的选择。

可以看到，随机数是会基于时钟计时器递增的，基本不可能会随机成一样的初始化序列号。

既然 IP 层会分片，为什么 TCP 层还需要 MSS 呢？

我们先来认识下 MTU 和 MSS



- **MTU**：一个网络包的最大长度（包括IP头+TCP头+数据），以太网中一般为 1500 字节；
- **MSS**：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度；

如果在 TCP 的整个报文（头部 + 数据）交给 IP 层进行分片，会有什么异常呢？

当 IP 层有一个超过 **MTU** 大小的数据（TCP 头部 + TCP 数据）要发送，那么 IP 层就要进行分片，把数据分片成若干片，保证每一个分片都小于 MTU。把一份 IP 数据报进行分片以后，由目标主机的 IP 层来进行重新组装后，再交给上一层 TCP 传输层。

这看起来井然有序，但这存在隐患的，那么当如果一个 IP 分片丢失，整个 IP 报文的所有分片都得重传。因为 IP 层本身没有超时重传机制，它由传输层的 TCP 来负责超时和重传。当某一个 IP 分片丢失后，接收方的 IP 层就无法组装成一个完整的 TCP 报文（头部 + 数据），也就无法将数据报文送到 TCP 层，所以接收方不会响应 ACK 给发送方，因为发送方迟迟收不到 ACK 确认报文，所以会触发超时重传，就会重发「整个 TCP 报文（头部 + 数据）」。

因此，可以得知由 IP 层进行分片传输，是非常没有效率的。所以，为了达到最佳的传输效能 TCP 协议在建立连接的时候通常要协商双方的 MSS 值，当 TCP 层发现数据超过 MSS 时，则会先进行分片，当然由它形成的 IP 包的长度也就不会大于 MTU，自然也就不用 IP 分片了。

```
[SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
[SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412 SACK_PERM=1 WS=16384
[ACK] Seq=1 Ack=1 Win=66304 Len=0
```

经过 TCP 层分片后，如果一个 TCP 分片丢失后，进行重发时也是以 MSS 为单位，而不用重传所有的分片，大大增加了重传的效率。

第一次握手丢失了，会发生什么？

当客户端想和服务端建立 TCP 连接的时候，首先第一个发的就是 SYN 报文，然后进入到 `SYN_SENT` 状态。在这之后，如果客户端迟迟收不到服务端的 SYN-ACK 报文（第二次握手），就会触发「超时重传」机制，重传 SYN 报文，而且重传的 SYN 报文的序列号都是一样的。

不同版本的操作系统可能超时时间不同，有的 1 秒的，也有 3 秒的，这个超时时间是写死在内核里的，如果想要更改则需要重新编译内核，比较麻烦。当客户端在 1 秒后没收到服务端的 SYN-ACK 报文后，客户端就会重发 SYN 报文，那到底重发几次呢？

在 Linux 里，客户端的 SYN 报文最大重传次数由 `tcp_syn_retries` 内核参数控制，这个参数是可以自定义的，默认值一般是 5。通常，第一次超时重传是在 1 秒后，第二次超时重传是在 2 秒，第三次超时重传是在 4 秒后，第四次超时重传是在 8 秒后，第五次是在超时重传 16 秒后。没错，每次超时的时间是上一次的 2 倍。当第五次超时重传后，会继续等待 32 秒，如果服务端仍然没有回应 ACK，客户端就不再发送 SYN 包，然后断开 TCP 连接。所以，总耗时是 $1+2+4+8+16+32=63$ 秒，大约 1 分钟左右。

第二次握手丢失了，会发生什么？

当服务端收到客户端的第一次握手后，就会回 SYN-ACK 报文给客户端，这个就是第二次握手，此时服务端会进入 `SYN_RECV` 状态。第二次握手的 SYN-ACK 报文其实有两个目的：

- 第二次握手里的 ACK，是对第一次握手的确认报文；
- 第二次握手里的 SYN，是服务端发起建立 TCP 连接的报文；

所以，如果第二次握手丢了，就会发生比较有意思的事情，具体会怎么样呢？

因为第二次握手报文里是包含对客户端的第一次握手的 ACK 确认报文，所以，如果客户端迟迟没有收到第二次握手，那么客户端就觉得可能自己的 SYN 报文（第一次握手）丢失了，于是客户端就会触发超时重传机制，重传 SYN 报文。

然后，因为第二次握手中包含服务端的 SYN 报文，所以当客户端收到后，需要给服务端发送 ACK 确认报文（第三次握手），服务端才会认为该 SYN 报文被客户端收到了。那么，如果第二次握手丢失了，服务端就收不到第三次握手，于是服务端这边会触发超时重传机制，重传 SYN-ACK 报文。

在 Linux 下，SYN-ACK 报文的最大重传次数由 `tcp_synack_retries` 内核参数决定，默认值是 5。

因此，当第二次握手丢失了，客户端和服务端都会重传：

- 客户端会重传 SYN 报文，也就是第一次握手，最大重传次数由 `tcp_syn_retries` 内核参数决定；

- 服务端会重传 SYN-ACK 报文，也就是第二次握手，最大重传次数由 `tcp_synack_retries` 内核参数决定。

第三次握手丢失了，会发生什么？

客户端收到服务端的 SYN-ACK 报文后，就会给服务端回一个 ACK 报文，也就是第三次握手，此时客户端状态进入到 `ESTABLISHED` 状态。

因为这个第三次握手的 ACK 是对第二次握手的 SYN 的确认报文，所以当第三次握手丢失了，如果服务端那一方迟迟收不到这个确认报文，就会触发超时重传机制，重传 SYN-ACK 报文，直到收到第三次握手，或者达到最大重传次数。注意，ACK 报文是不会有重传的，当 ACK 丢失了，就由对方重传对应的报文。

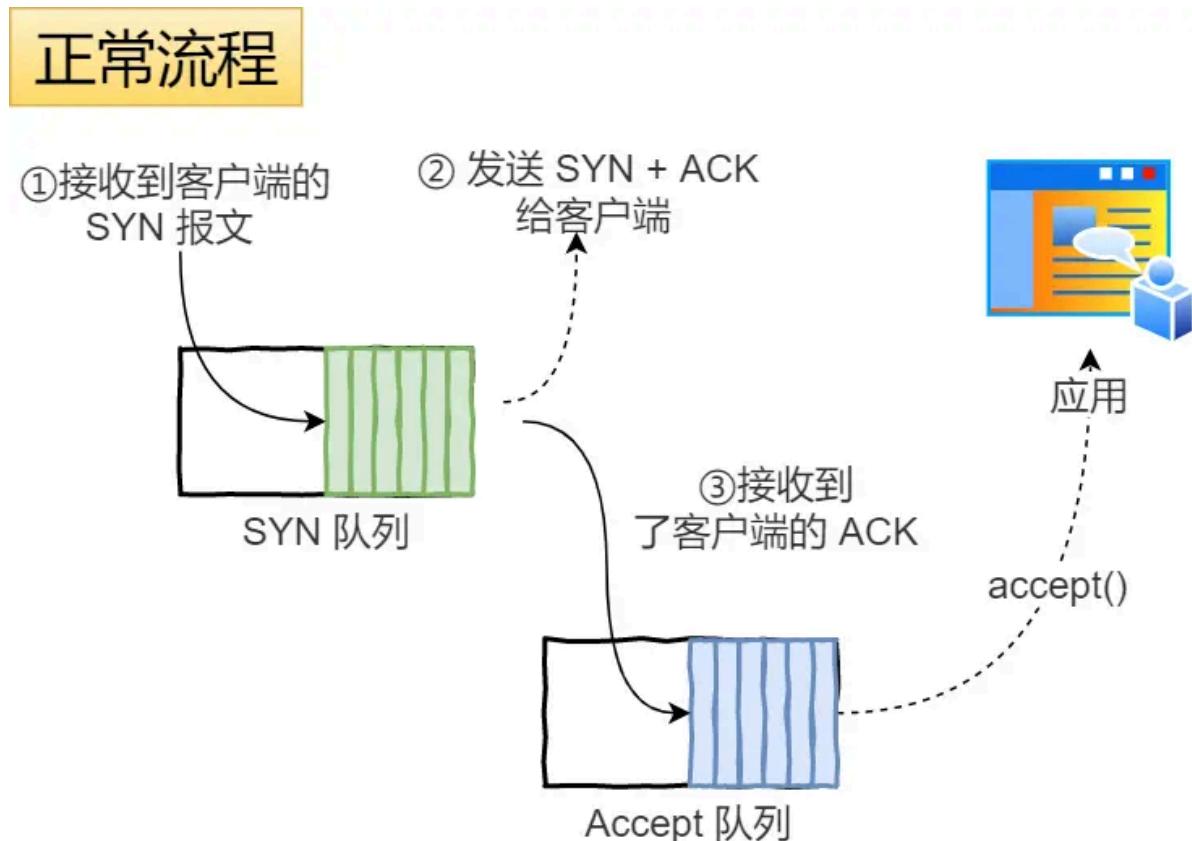
什么是 SYN 攻击？如何避免 SYN 攻击？

我们都知道 TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 `SYN` 报文，服务端每接收到一个 `SYN` 报文，就进入 `SYN_RECV` 状态，但服务端发送出去的 `ACK + SYN` 报文，无法得到未知 IP 主机的 `ACK` 应答，久而久之就会占满服务端的半连接队列，使得服务端不能为正常用户提供服务。

在 TCP 三次握手的时候，Linux 内核会维护两个队列，分别是：

- 半连接队列，也称 `SYN` 队列；
- 全连接队列，也称 `accept` 队列；

我们先来看下 Linux 内核的 `SYN` 队列（半连接队列）与 `Accept` 队列（全连接队列）是如何工作的？



正常流程：

- 当服务端接收到客户端的 SYN 报文时，会创建一个半连接的对象，然后将其加入到内核的「SYN 队列」；
- 接着发送 SYN + ACK 给客户端，等待客户端回应 ACK 报文；
- 服务端接收到 ACK 报文后，从「SYN 队列」取出一个半连接对象，然后创建一个新的连接对象放入到「Accept 队列」；

- 应用通过调用 `accept()` socket 接口，从「Accept 队列」取出连接对象。

不管是半连接队列还是全连接队列，都有最大长度限制，超过限制时，默认情况都会丢弃报文。**SYN 攻击方式最直接的表现就会把 TCP 半连接队列打满，这样当 TCP 半连接队列满了，后续再在收到 SYN 报文就会丢弃，导致客户端无法和服务端建立连接。**

避免 SYN 攻击方式，可以有以下四种方法：

- 调大 `netdev_max_backlog`；
- 增大 TCP 半连接队列；
- 开启 `tcp_syncookies`；
- 减少 SYN+ACK 重传次数

方式一：调大 `netdev_max_backlog`

当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包。控制该队列的最大值如下参数，默认值是 1000，我们要适当调大该参数的值，比如设置为 10000：

```
net.core.netdev_max_backlog = 10000
```

方式二：增大 TCP 半连接队列

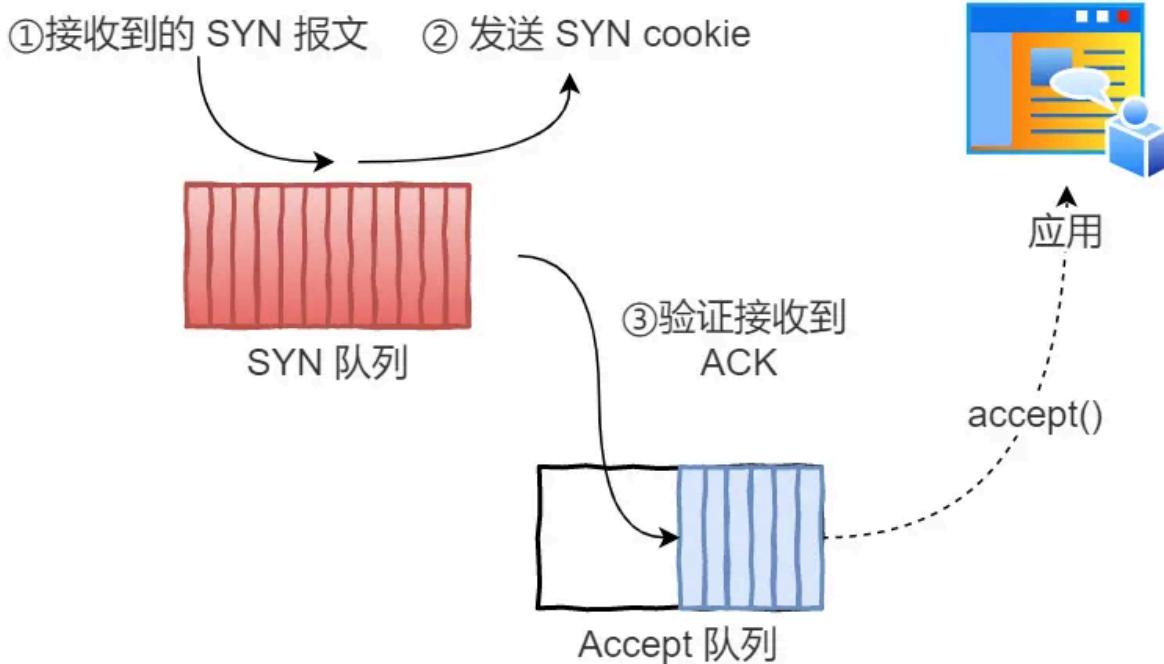
增大 TCP 半连接队列，要同时增大下面这三个参数：

- 增大 `net.ipv4.tcp_max_syn_backlog`
- 增大 `listen()` 函数中的 `backlog`
- 增大 `net.core.somaxconn`

方式三：开启 `net.ipv4.tcp_syncookies`

开启 `syncookies` 功能就可以在不使用 SYN 半连接队列的情况下成功建立连接，相当于绕过了 SYN 半连接来建立连接。

SYN 队列占满，启动 cookie



具体过程：

- 当「SYN 队列」满之后，后续服务端收到 SYN 包，不会丢弃，而是根据算法，计算出一个 `cookie` 值；
- 将 `cookie` 值放到第二次握手报文的「序列号」里，然后服务端回第二次握手给客户端；
- 服务端接收到客户端的应答报文时，服务端会检查这个 ACK 包的合法性。如果合法，将该连接对象放入到「Accept 队列」。
- 最后应用程序通过调用 `accept()` 接口，从「Accept 队列」取出的连接。

`net.ipv4.tcp_syncookies` 参数主要有以下三个值：

- 0 值，表示关闭该功能；
- 1 值，表示仅当 SYN 半连接队列放不下时，再启用它；
- 2 值，表示无条件开启功能；

那么在应对 SYN 攻击时，只需要设置为 1 即可。

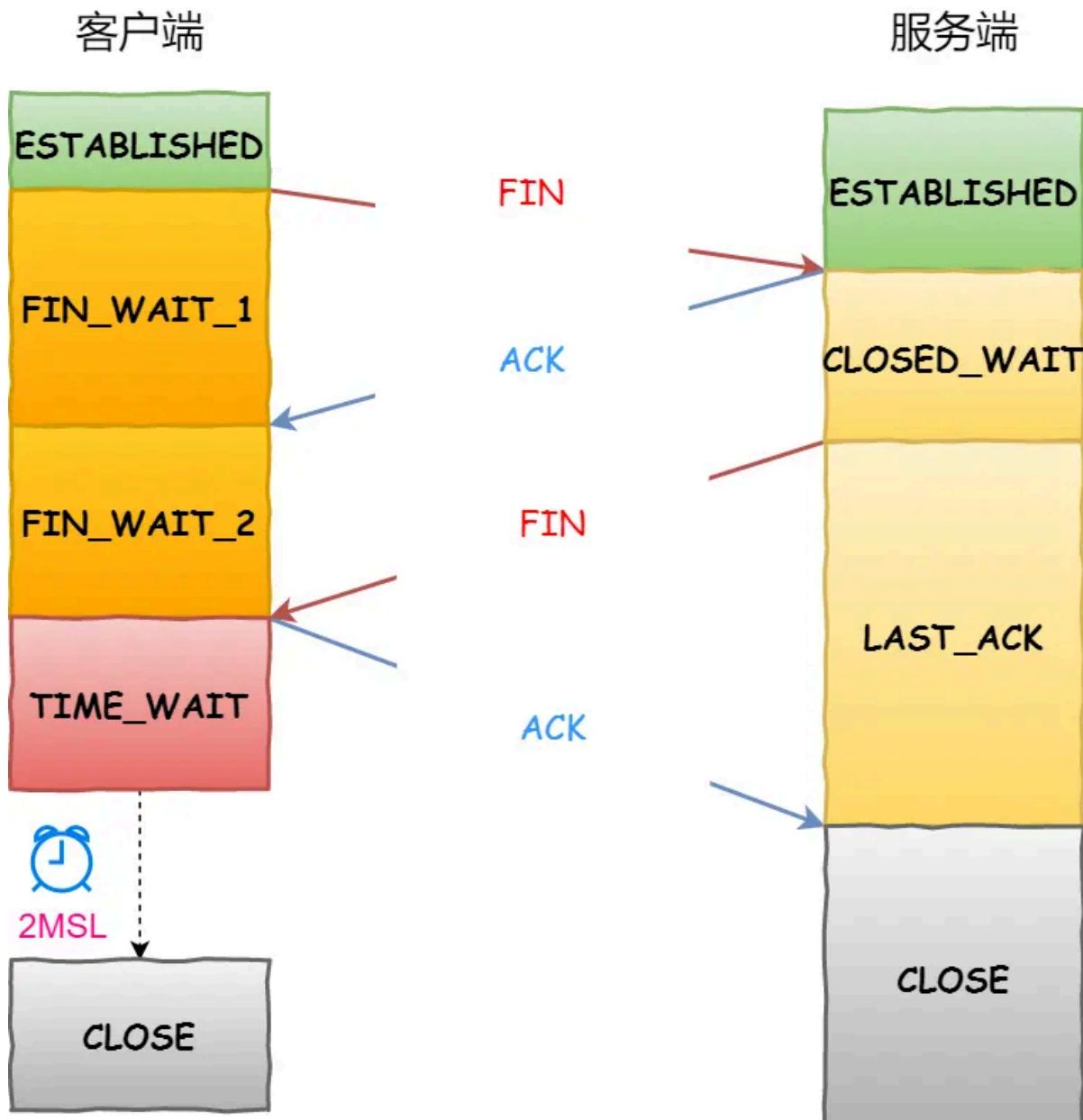
```
$ echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

方式四：减少 SYN+ACK 重传次数

当服务端受到 SYN 攻击时，就会有大量处于 SYN_RECV 状态的 TCP 连接，处于这个状态的 TCP 会重传 SYN+ACK，当重传超过次数达到上限后，就会断开连接。那么针对 SYN 攻击的场景，我们可以减少 SYN-ACK 的重传次数，以加快处于 SYN_RECV 状态的 TCP 连接断开。SYN-ACK 报文的最大重传次数由 `tcp_synack_retries` 内核参数决定（默认值是 5 次），比如将 `tcp_synack_retries` 减少到 2 次：

TCP 四次挥手过程是怎样的？

TCP 断开连接是通过四次挥手方式。双方都可以主动断开连接，断开连接后主机中的「资源」将被释放，四次挥手的过程如下图：



- 客户端主动调用关闭连接的函数，于是就会发送 FIN 报文，这个 FIN 报文代表客户端不会再发送数据了，进入 FIN_WAIT_1 状态；
- 服务端收到了 FIN 报文，然后马上回复一个 ACK 确认报文，此时服务端进入 CLOSE_WAIT 状态。在收到 FIN 报文的时候，TCP 协议栈会为 FIN 包插入一个文件结束符 EOF 到接收缓冲区中，服务端应用程序可以通过 read 调用来感知这个 FIN 包，这个 EOF 会被放在已排队等候的其他已接收的数据之后，所以必须要得继续 read 接收缓冲区已接收的数据；
- 接着，当服务端在 read 数据的时候，最后自然就会读到 EOF，接着 read() 就会返回 0，这时服务端应用程序如果有数据要发送的话，就发完数据后才调用关闭连接的函数，如果服务端应用程序没有数据要发送的话，可以直接调用关闭连接的函数，这时服务端就会发一个 FIN 包，这个 FIN 报文代表服务端不会再发送数据了，之后处于 LAST_ACK 状态；
- 客户端接收到服务端的 FIN 包，并发送 ACK 确认包给服务端，此时客户端将进入 TIME_WAIT 状态；
- 服务端收到 ACK 确认包后，就进入了最后的 CLOSE 状态；

- 客户端经过 2MSL 时间之后，也进入 CLOSE 状态；

你可以看到，每个方向都需要一个 FIN 和一个 ACK，因此通常被称为**四次挥手**。这里一点需要注意是：**主动关闭连接的，才有 TIME_WAIT 状态。**

为什么挥手需要四次？

再来看看四次挥手双方发 FIN 包的过程，就能理解为什么需要四次了。

- 关闭连接时，客户端向服务端发送 FIN 时，仅仅表示客户端不再发送数据了但是还能接收数据。
- 服务端收到客户端的 FIN 报文时，先回一个 ACK 应答报文，而服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 FIN 报文给客户端来表示同意现在关闭连接。

从上面过程可知，服务端通常需要等待完成数据的发送和处理，所以服务端的 ACK 和 FIN 一般都会分开发送，因此是需要四次挥手。但是在特定情况下，四次挥手是可以变成三次挥手的。

第一次挥手丢失了，会发生什么？

当客户端（主动关闭方）调用 close 函数后，就会向服务端发送 FIN 报文，试图与服务端断开连接，此时客户端的连接进入到 FIN_WAIT_1 状态。正常情况下，如果能及时收到服务端（被动关闭方）的 ACK，则会很快变为 FIN_WAIT2 状态。**如果第一次挥手丢失了，那么客户端迟迟收不到被动方的 ACK 的话，也就触发超时重传机制，重传 FIN 报文，重发次数由 tcp_orphan_retries 参数控制。**当客户端重传 FIN 报文的次数超过 tcp_orphan_retries 后，就不再发送 FIN 报文，则会在等待一段时间（时间为上一次超时时间的 2 倍），如果还是没能收到第二次挥手，那么直接进入到 close 状态。

第二次挥手丢失了，会发生什么？

当服务端收到客户端的第一手挥手后，就会先回一个 ACK 确认报文，此时服务端的连接进入到 CLOSE_WAIT 状态。在前面我们也提了，ACK 报文是不会重传的，所以**如果服务端的第二次挥手丢失了，客户端就会触发超时重传机制，重传 FIN 报文，直到收到服务端的第二次挥手，或者达到最大的重传次数。**

当客户端收到第二次挥手，也就是收到服务端发送的 ACK 报文后，客户端就会处于 FIN_WAIT2 状态，在这个状态需要等服务端发送第三次挥手，也就是服务端的 FIN 报文。

1. 对于 close 函数关闭的连接，由于无法再发送和接收数据，所以 FIN_WAIT2 状态不可以持续太久，而 tcp_fin_timeout 控制了这个状态下连接的持续时长，默认值是 60 秒。**这意味着对于调用 close 关闭的连接，如果在 60 秒后还没有收到 FIN 报文，客户端（主动关闭方）的连接就会直接关闭。**
2. 如果主动关闭方使用 shutdown 函数关闭连接，指定了只关闭发送方向，而接收方向并没有关闭，那么意味着主动关闭方还是可以接收数据的。此时，**如果主动关闭方一直没收到第三次挥手，那么主动关闭方的连接将会一直处于 FIN_WAIT2 状态（tcp_fin_timeout 无法控制 shutdown 关闭的连接）。**

第三次挥手丢失了，会发生什么？

当服务端（被动关闭方）收到客户端（主动关闭方）的 FIN 报文后，内核会自动回复 ACK，同时连接处于 CLOSE_WAIT 状态，顾名思义，它表示等待应用进程调用 close 函数关闭连接。此时，内核是没有权利替代进程关闭连接，必须由进程主动调用 close 函数来触发服务端发送 FIN 报文（因为可能服务端进程可能需要处理数据）。

服务端处于 CLOSE_WAIT 状态时，调用了 close 函数，内核就会发出 FIN 报文，同时连接进入 LAST_ACK 状态，等待客户端返回 ACK 来确认连接关闭。如果迟迟收不到这个 ACK，服务端就会重发 FIN 报文，重发次数仍然由 `tcp_orphan_retries` 参数控制，这与客户端重发 FIN 报文的重传次数控制方式是一样的。

第四次挥手丢失了，会发生什么？

当客户端收到服务端的第三次挥手的 FIN 报文后，就会回 ACK 报文，也就是第四次挥手，此时客户端连接进入 TIME_WAIT 状态。在 Linux 系统，TIME_WAIT 状态会持续 2MSL 后才会进入关闭状态，如果途中再次收到第三次挥手（FIN 报文）后，就会重置定时器，当等待 2MSL 时长后，客户端就会断开连接。然后，服务端（被动关闭方）没有收到 ACK 报文前，还是处于 LAST_ACK 状态。如果第四次挥手的 ACK 报文没有到达服务端，服务端就会重发 FIN 报文，重发次数仍然由前面介绍过的 `tcp_orphan_retries` 参数控制。

为什么 TIME_WAIT 等待的时间是 2MSL？

`MSL` 是 Maximum Segment Lifetime，报文最大生存时间，它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。因为 TCP 报文基于是 IP 协议的，而 IP 头中有一个 `TTL` 字段，是 IP 数据报可以经过的最大路由数，每经过一个处理他的路由器此值就减 1，当此值为 0 则数据报将被丢弃，同时发送 ICMP 报文通知源主机。MSL 与 TTL 的区别：MSL 的单位是时间，而 TTL 是经过路由跳数。所以 MSL 应该要大于等于 TTL 消耗为 0 的时间，以确保报文已被自然消亡。TTL 的值一般是 64，Linux 将 MSL 设置为 30 秒，意味着 Linux 认为数据报文经过 64 个路由器的时间不会超过 30 秒，如果超过了，就认为报文已经消失在网络中了。

TIME_WAIT 等待 2 倍的 MSL，如果被动关闭方没有收到断开连接的最后的 ACK 报文，就会触发超时重发 `FIN` 报文，另一方接收到 `FIN` 后，会重发 `ACK` 给被动关闭方，一来一去正好 2 个 MSL。可以看到 **2MSL时长** 这其实是相当于**至少允许报文丢失一次**。比如，若 `ACK` 在一个 MSL 内丢失，这样被动方重发的 `FIN` 会在第 2 个 MSL 内到达，TIME_WAIT 状态的连接可以应对。

为什么不是 4 或者 8 MSL 的时长呢？你可以想象一个丢包率达到百分之一的糟糕网络，连续两次丢包的概率只有万分之一，这个概率实在是太小了，忽略它比解决它更具性价比。

`2MSL` 的时间是从**客户端接收到 FIN 后发送 ACK 开始计时的**。如果在 TIME-WAIT 时间内，因为客户端的 `ACK` 没有传输到服务端，客户端又接收到了服务端重发的 `FIN` 报文，那么 **2MSL 时间将重新计时**。在 Linux 系统里 `2MSL` 默认是 `60` 秒，那么一个 `MSL` 也就是 `30` 秒。**Linux 系统停留在 TIME_WAIT 的时间为固定的 60 秒**。其定义在 Linux 内核代码里的名称为 `TCP_TIMEWAIT_LEN`。

为什么需要 TIME_WAIT 状态？

主动发起关闭连接的一方，才会有 TIME-WAIT 状态。需要 TIME-WAIT 状态，主要是两个原因：

- 防止历史连接中的数据，被后面相同四元组的连接错误的接收；
- 保证「被动关闭连接」的一方，能被正确的关闭；

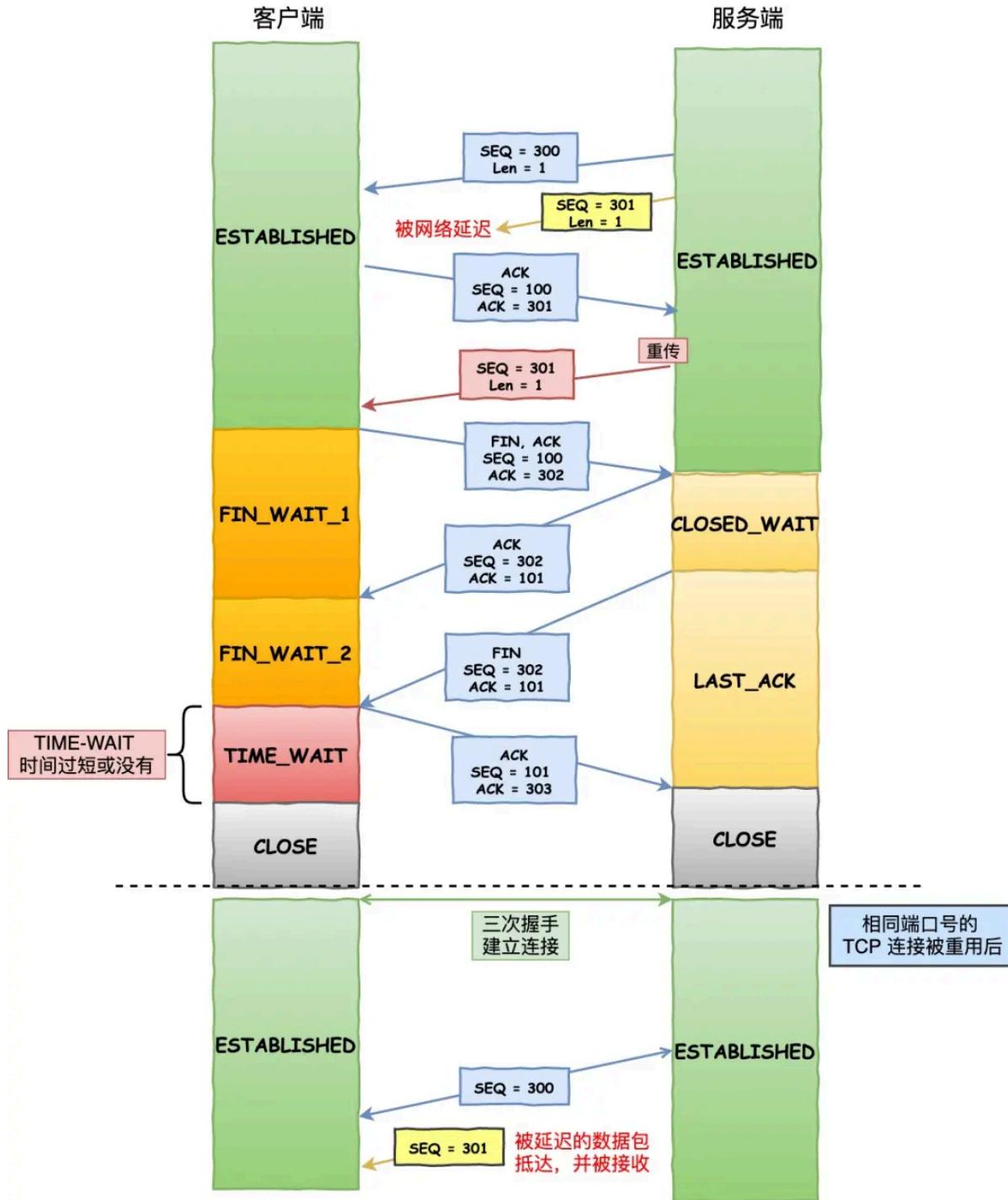
原因一：防止历史连接中的数据，被后面相同四元组的连接错误的接收

为了能更好的理解这个原因，我们先来了解序列号（SEQ）和初始序列号（ISN）。

- **序列号**，是 TCP 一个头部字段，标识了 TCP 发送端到 TCP 接收端的数据流的一个字节，因为 TCP 是面向字节流的可靠协议，为了保证消息的顺序性和可靠性，TCP 为每个传输方向上的每个字节都赋予了一个编号，以便于传输成功后确认、丢失后重传以及在接收端保证不会乱序。**序列号是一个 32 位的无符号数，因此在到达 4G 之后再循环回到 0。**

- **初始序列号**，在 TCP 建立连接的时候，客户端和服务端都会各自生成一个初始序列号，它是基于时钟生成的一个随机数，来保证每个连接都拥有不同的初始序列号。初始化序列号可被视为一个 32 位的计数器，该计数器的数值每 4 微秒加 1，循环一次需要 4.55 小时。

通过前面我们知道，序列号和初始化序列号并不是无限递增的，会发生回绕为初始值的情况，这意味着无法根据序列号来判断新老数据。假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会发生什么呢？



如上图：

- 服务端在关闭连接之前发送的 **SEQ = 301** 报文，被网络延迟了。
- 接着，服务端以相同的四元组重新打开了新连接，前面被延迟的 **SEQ = 301** 这时抵达了客户端，而且该数据报文的序列号刚好在客户端接收窗口内，因此客户端会正常接收这个数据报文，但是这个数据报文是上一个连接残留下来的，这样就产生数据错乱等严重的问题。

为了防止历史连接中的数据，被后面相同四元组的连接错误的接收，因此 TCP 设计了 TIME_WAIT 状态，状态会持续 2MSL 时长，这个时间足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。

原因二：保证「被动关闭连接」的一方，能被正确的关闭

也就是说，TIME-WAIT 作用是等待足够的时间以确保最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭。如果客户端（主动关闭方）最后一次 ACK 报文（第四次挥手）在网络中丢失了，那么按照 TCP 可靠性原则，服务端（被动关闭方）会重发 FIN 报文。假设客户端没有 TIME_WAIT 状态，而是在发完最后一次回 ACK 报文就直接进入 CLOSE 状态，如果该 ACK 报文丢失了，服务端则重传的 FIN 报文，而这时客户端已经进入到关闭状态了，在收到服务端重传的 FIN 报文后，就会回 RST 报文。

服务端收到这个 RST 并将其解释为一个错误（Connection reset by peer），这对于一个可靠的协议来说不是一个优雅的终止方式。为了防止这种情况出现，客户端必须等待足够长的时间，确保服务端能够收到 ACK，如果服务端没有收到 ACK，那么就会触发 TCP 重传机制，服务端会重新发送一个 FIN，这样一去一来刚好两个 MSL 的时间。客户端在收到服务端重传的 FIN 报文时，TIME_WAIT 状态的等待时间，会重置回 2MSL。

TIME_WAIT 过多有什么危害？

过多的 TIME-WAIT 状态主要的危害有两种：

- 第一是占用系统资源，比如文件描述符、内存资源、CPU 资源、线程资源等；
- 第二是占用端口资源，端口资源也是有限的，一般可以开启的端口为 32768~61000，也可以通过 `net.ipv4.ip_local_port_range` 参数指定范围。

客户端和服务端 TIME_WAIT 过多，造成的影响是不同的。

如果客户端（主动发起关闭连接方）的 TIME_WAIT 状态过多，占满了所有端口资源，那么就无法对「目的 IP+ 目的 PORT」都一样的服务端发起连接了，但是被使用的端口，还是可以继续对另外一个服务端发起连接的。

如果服务端（主动发起关闭连接方）的 TIME_WAIT 状态过多，并不会导致端口资源受限，因为服务端只监听一个端口，而且由于一个四元组唯一确定一个 TCP 连接，因此理论上服务端可以建立很多连接，但是 TCP 连接过多，会占用系统资源，比如文件描述符、内存资源、CPU 资源、线程资源等。

如何优化 TIME_WAIT？

这里给出优化 TIME-WAIT 的几个方式，都是有利有弊：

- 打开 `net.ipv4.tcp_tw_reuse` 和 `net.ipv4.tcp_timestamps` 选项：可以复用处于 TIME_WAIT 的 socket 为新的连接所用。有一点需要注意的是，`tcp_tw_reuse` 功能只能用客户端（连接发起方），因为开启了该功能，在调用 `connect()` 函数时，内核会随机找一个 time_wait 状态超过 1 秒的连接给新的连接复用。
- `net.ipv4.tcp_max_tw_buckets`：这个值默认为 18000，当系统中处于 TIME_WAIT 的连接一旦超过这个值时，系统就会将后面的 TIME_WAIT 连接状态重置，这个方法比较暴力。
- 程序中使用 `SO_LINGER`，应用强制使用 RST 关闭。

方式三：程序中使用 SO_LINGER

我们可以通过设置 socket 选项，来设置调用 close 关闭连接行为。

```
struct linger so_linger;
so_linger.l_onoff = 1;
so_linger.l_linger = 0;
setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger, sizeof(so_linger));
```

如果 `l_onoff` 为非 0，且 `l_linger` 值为 0，那么调用 `close` 后，会立即将一个 `RST` 标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了 `TIME_WAIT` 状态，直接关闭。但这为跨越 `TIME_WAIT` 状态提供了一个可能，不过是一个非常危险的行为，不值得提倡。

如果服务端要避免过多的 `TIME_WAIT` 状态的连接，就永远不要主动断开连接，让客户端去断开，由分布在各处的客户端去承受 `TIME_WAIT`。

服务器出现大量 `TIME_WAIT` 状态的原因有哪些？

首先要知道 `TIME_WAIT` 状态是主动关闭连接才会出现的状态，所以如果服务器出现大量的 `TIME_WAIT` 状态的 TCP 连接，就是说明服务器主动断开了很多 TCP 连接。

问题来了，什么场景下服务端会主动断开连接呢？

- 第一个场景：HTTP 没有使用长连接（两边都需要开启）
- 第二个场景：HTTP 长连接超时
- 第三个场景：HTTP 长连接的请求数量达到上限

根据大多数 Web 服务的实现，不管哪一方禁用了 HTTP Keep-Alive，都是由服务端主动关闭连接，那么此时服务端上就会出现 `TIME_WAIT` 状态的连接。

客户端禁用了 HTTP Keep-Alive，服务端开启 HTTP Keep-Alive，谁是主动关闭方？

当客户端禁用了 HTTP Keep-Alive，这时候 HTTP 请求的 header 就会有 `Connection:close` 信息，这时服务端在发完 HTTP 响应后，就会主动关闭连接。

为什么要这么设计呢？HTTP 是请求-响应模型，发起方一直是客户端，HTTP Keep-Alive 的初衷是为客户端后续的请求重用连接，如果我们在某次 HTTP 请求-响应模型中，请求的 header 定义了 `connection: close` 信息，那不再重用这个连接的时机就只有在服务端了，所以我们在 HTTP 请求-响应这个周期的「末端」关闭连接是合理的。

客户端开启了 HTTP Keep-Alive，服务端禁用了 HTTP Keep-Alive，谁是主动关闭方？

当客户端开启了 HTTP Keep-Alive，而服务端禁用了 HTTP Keep-Alive，这时服务端在发完 HTTP 响应后，服务端也会主动关闭连接。

因此，当服务端出现大量的 `TIME_WAIT` 状态连接的时候，可以排查下是否客户端和服务端都开启了 HTTP Keep-Alive，因为任意一方没有开启 HTTP Keep-Alive，都会导致服务端在处理完一个 HTTP 请求后，就主动关闭连接，此时服务端上就会出现大量的 `TIME_WAIT` 状态的连接。

针对这个场景下，解决的方式也很简单，让客户端和服务端都开启 HTTP Keep-Alive 机制。

第二个场景：HTTP 长连接超时

HTTP 长连接的特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。对没错，所以为了避免资源浪费的情况，web 服务软件一般都会提供一个参数，用来指定 HTTP 长连接的超时时间，比如 nginx 提供的 `keepalive_timeout` 参数。假设设置了 HTTP 长连接的超时时间是 60 秒，nginx 就会启动一个「定时器」，如果客户端在完后一个 HTTP 请求后，在 60 秒内都没有再发起新的请求，定时器的时间一到，nginx 就会触发回调函数来关闭该连接，那么此时服务端上就会出现 `TIME_WAIT` 状态的连接。

可以往网络问题的方向排查，比如是否是因为网络问题，导致客户端发送的数据一直没有被服务端接收到，以至于 HTTP 长连接超时。

第三个场景：HTTP 长连接的请求数量达到上限

Web 服务端通常会有个参数，来定义一条 HTTP 长连接上最大能处理的请求数量，当超过最大限制时，就会主动关闭连接。比如 nginx 的 `keepalive_requests` 这个参数，这个参数是指一个 HTTP 长连接建立之后，nginx 就会为这个连接设置一个计数器，记录这个 HTTP 长连接上已经接收并处理的客户端请求的数量。**如果达到这个参数设置的最大值时，则 nginx 会主动关闭这个长连接**，那么此时服务端上就会出现 `TIME_WAIT` 状态的连接。

`keepalive_requests` 参数的默认值是 100，意味着每个 HTTP 长连接最多只能跑 100 次请求，这个参数往往被大多数人忽略，因为当 QPS (每秒请求数) 不是很高时，默认值 100 凑合够用。

但是，**对于一些 QPS 比较高的场景，比如超过 10000 QPS，甚至达到 30000, 50000 甚至更高，如果 `keepalive_requests` 参数值是 100，这时候就 nginx 就会很频繁地关闭连接，那么此时服务端上就会出大量的 `TIME_WAIT` 状态**。针对这个场景下，解决的方式也很简单，调大 nginx 的 `keepalive_requests` 参数就行。

服务器出现大量 `CLOSE_WAIT` 状态的原因有哪些？

`CLOSE_WAIT` 状态是「被动关闭方」才有的状态，而且如果「被动关闭方」没有调用 `close` 函数关闭连接，那么就无法发出 `FIN` 报文，从而无法使得 `CLOSE_WAIT` 状态的连接转变为 `LAST_ACK` 状态。

所以，**当服务端出现大量 `CLOSE_WAIT` 状态的连接的时候，说明服务端的程序没有调用 `close` 函数关闭连接**。可以发现，**当服务端出现大量 `CLOSE_WAIT` 状态的连接的时候，通常都是代码的问题，这时候我们需要针对具体的代码一步一步的进行排查和定位，主要分析的方向就是服务端为什么没有调用 `close`**。

如果已经建立了连接，但是客户端突然出现故障了怎么办？

客户端出现故障指的是客户端的主机发生了宕机，或者断电的场景。发生这种情况的时候，如果服务端一直不会发送数据给客户端，那么服务端是永远无法感知到客户端宕机这个事件的，也就是服务端的 TCP 连接将一直处于 `ESTABLISH` 状态，占用着系统资源。

为了避免这种情况，TCP 搞了个保活机制。这个机制的原理是这样的：**定义一个时间段，在这个时间段内，如果有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个探测报文，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。**

TCP 保活的这个机制检测的时间是有点长，我们可以自己在应用层实现一个心跳机制。比如，web 服务软件一般都会提供 `keepalive_timeout` 参数，用来指定 HTTP 长连接的超时时间。如果设置了 HTTP 长连接的超时时间是 60 秒，web 服务软件就会启动一个定时器，如果客户端在完成一个 HTTP 请求后，在 60 秒内都没有再发起新的请求，定时器的时间一到，就会触发回调函数来释放该连接。**

如果已经建立了连接，但是服务端的进程崩溃会发生什么？

TCP 的连接信息是由内核维护的，所以当服务端的进程崩溃后，内核需要回收该进程的所有 TCP 连接资源，于是内核会发送第一次挥手 `FIN` 报文，后续的挥手过程也都是在内核完成，并不需要进程的参与，所以即使服务端的进程退出了，还是能与客户端完成 TCP 四次挥手的过程。

我自己做了个实验，使用 `kill -9` 来模拟进程崩溃的情况，发现在 `kill` 掉进程后，服务端会发送 `FIN` 报文，与客户端进行四次挥手。

重传机制

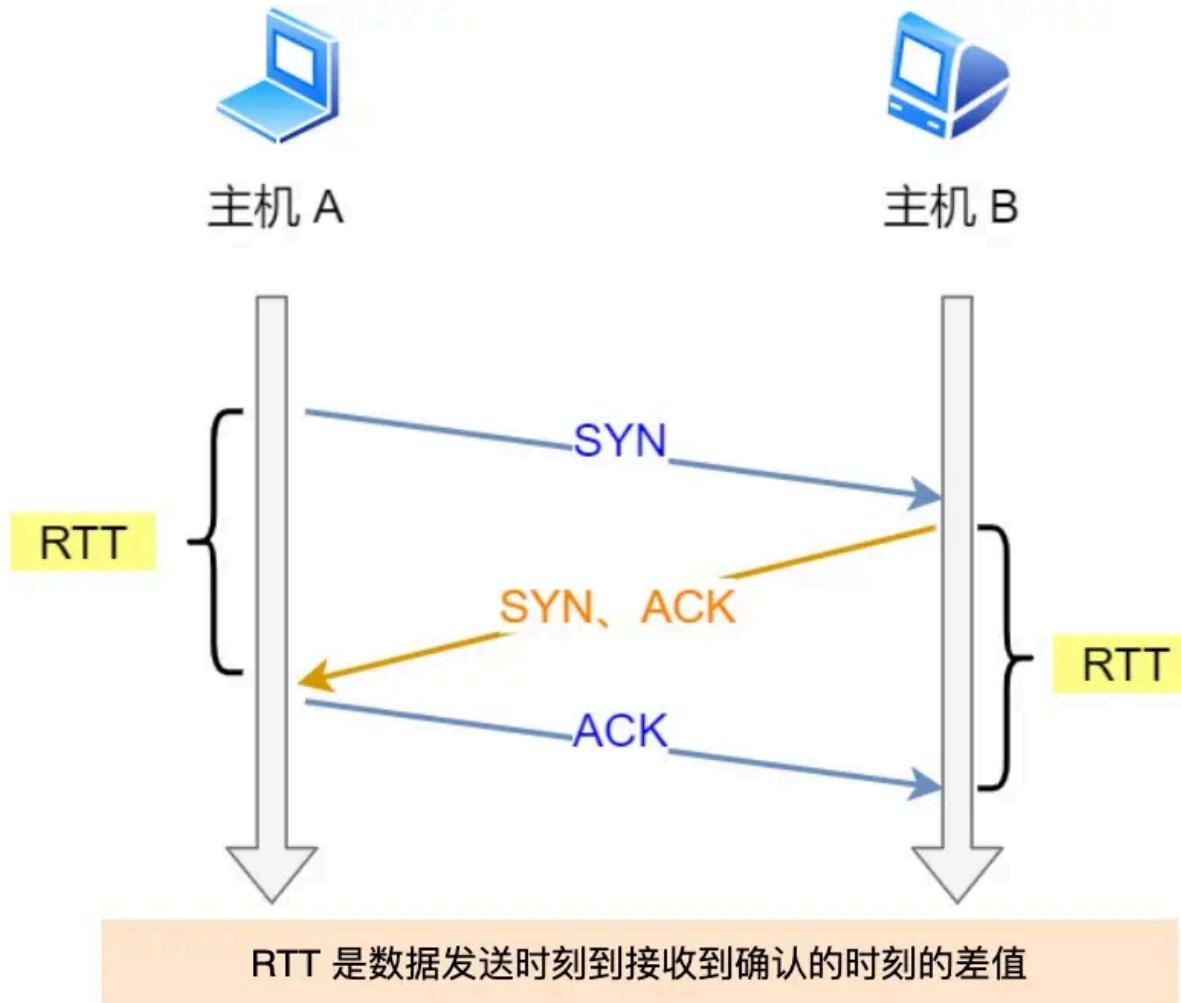
超时重传

发送数据时，设定一个定时器，当超过指定的时间后，没有收到对方的 `ACK` 确认应答报文，就会重发该数据，也就是我们常说的超时重传。TCP 会在以下两种情况发生超时重传：

- 数据包丢失
- 确认应答丢失

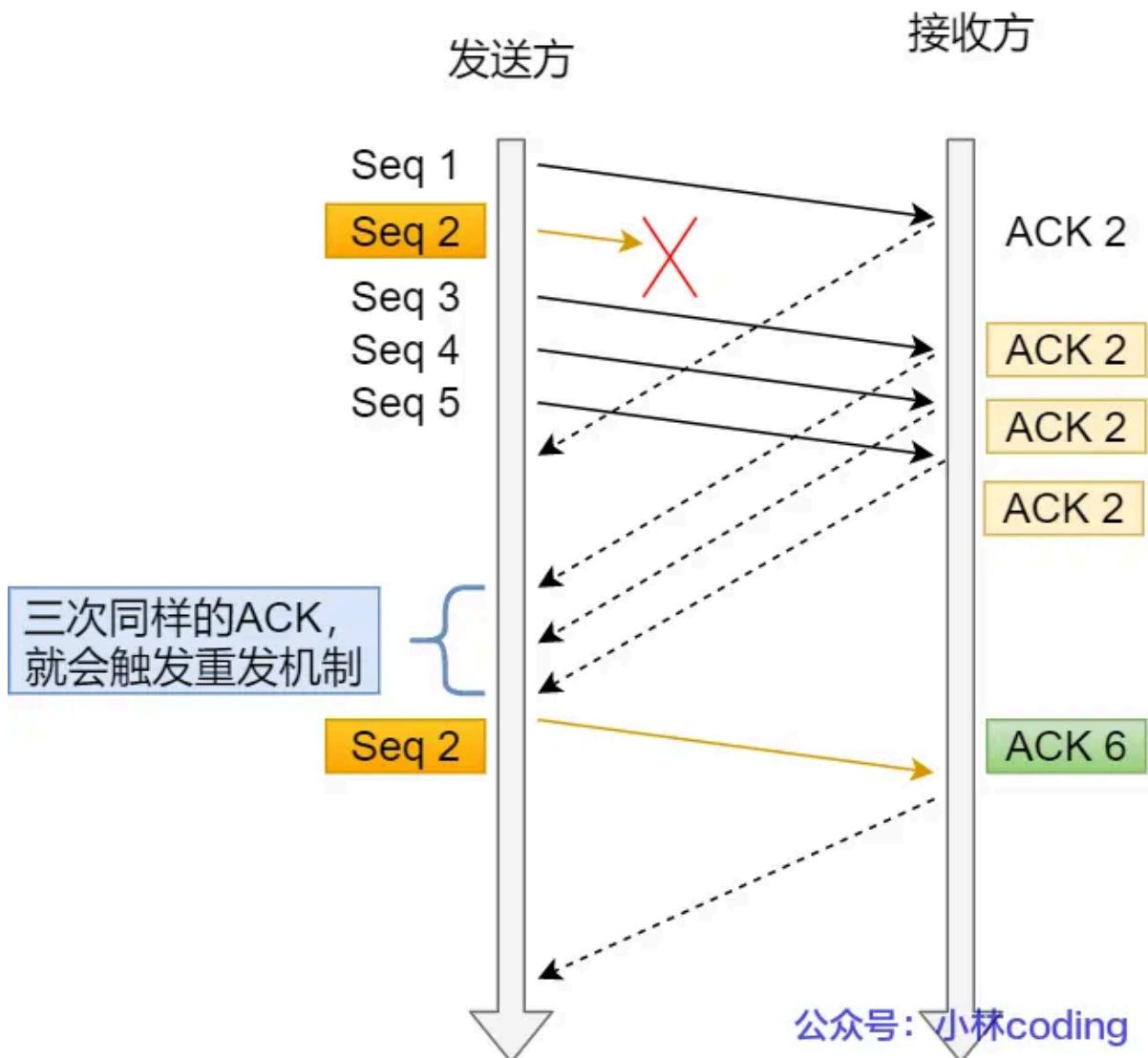
`RTT` (Round-Trip Time 往返时延) 指的是数据发送时刻到接收到确认的时刻的差值，也就是包的往返时间。超时重传时间是以 `RTO` (Retransmission Timeout 超时重传时间) 表示。超时重传时间 `RTO` 的值应该略大于报文往返 `RTT` 的值，`RTO` 是动态变化的值，每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍：

- 当超时时间 `RTO` 较大时，重发就慢，丢了老半天才重发，没有效率，性能差；
- 当超时时间 `RTO` 较小时，会导致可能并没有丢就重发，于是重发的就快，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。



快速重传

发送端收到了三个 `Ack = 2` 的确认，知道了 `Seq2` 还没有收到，就会在定时器过期之前，重传丢失的 `Seq2`。但重传的时候，是重传一个，还是重传所有的问题。为了解决不知道该重传哪些 TCP 报文，于是就有 `SACK` 方法。



SACK 方法

`SACK` (Selective Acknowledgment)，**选择性确认**。这种方式需要在 TCP 头部「选项」字段里加一个 `SACK` 的东西，它可以将已收到的数据的信息发送给「发送方」，这样发送方就可以知道哪些数据收到了，哪些数据没收到，知道了这些信息，就可以只重传丢失的数据。在 Linux 下，可以通过 `net.ipv4.tcp_sack` 参数打开这个功能（Linux 2.4 后默认打开）。**PS：一般此时的ACK比SACK小。**

Duplicate SACK

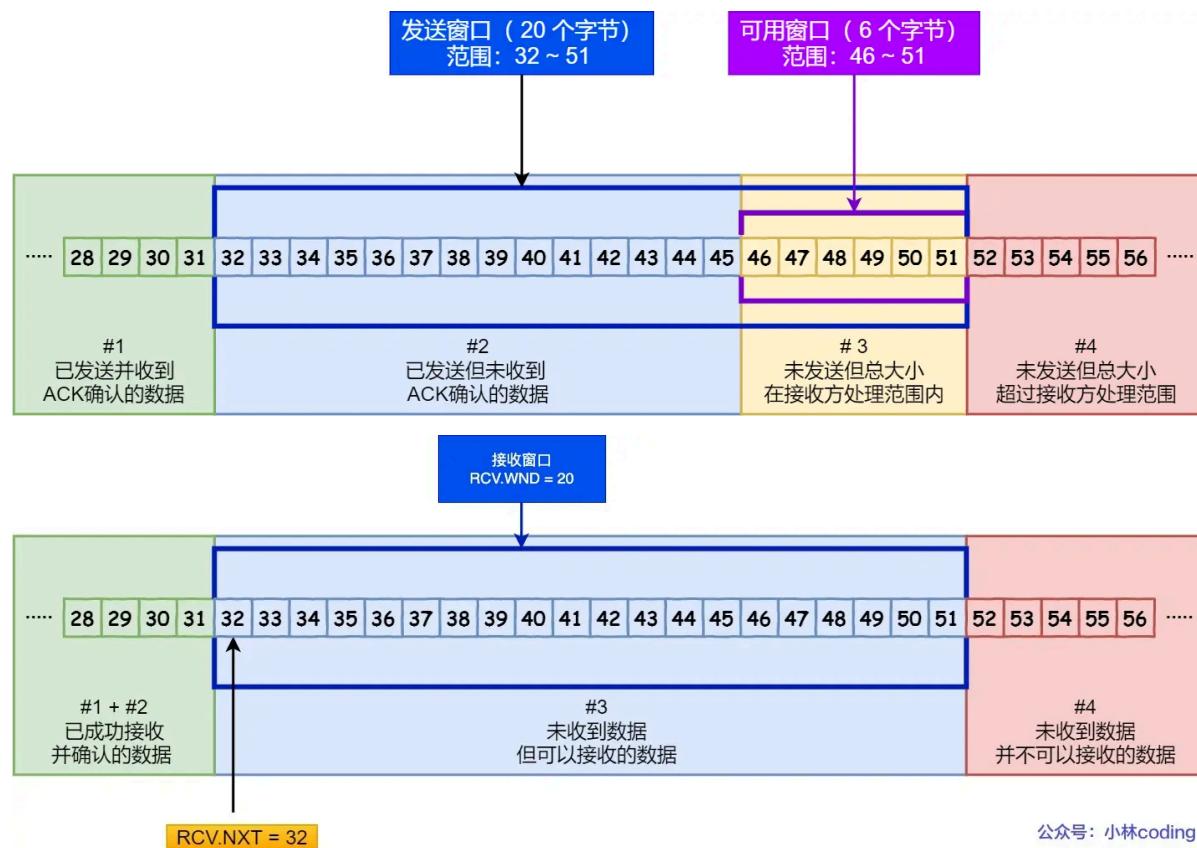
Duplicate SACK 又称 `D-SACK`，其主要使用了 `SACK` 来告诉「发送方」有哪些数据被重复接收了。在 Linux 下可以通过 `net.ipv4.tcp_dsack` 参数开启/关闭这个功能（Linux 2.4 后默认打开）。**PS：一般此时的ACK比SACK大。**

滑动窗口

如果没有窗口这个概念的话，我们每次都得等收到上次发送数据的 `ACK` 才能发送下一次数据，这样数据包的往返时间越长，通信的效率越低。有了窗口，就可以指定窗口大小，窗口大小就是指无需等待确认应答，而可以继续发送数据的最大值。窗口的实现实际上是操作系统开辟的一个缓存空间，发送方主机在等到确认应答返回之前，必须在缓冲区中保留已发送的数据。如果按期收到确认应答，此时数据就可以从缓存区清除。假设窗口大小为 3 个 TCP 段，那么发送方就可以「连续发送」3 个 TCP 段，并且中途若有 `ACK` 丢失，可以通过「下一个确认应答进行确认」。这个模式就叫累计确认或者累计应答。

TCP 头里有一个字段叫 `window`，也就是窗口大小。这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。通常窗口的大小是由接收方的窗口大小来决定的。发送方发送的数据大小不能超过接收方的窗口大小，否则接收方就无法正常接收到数据。

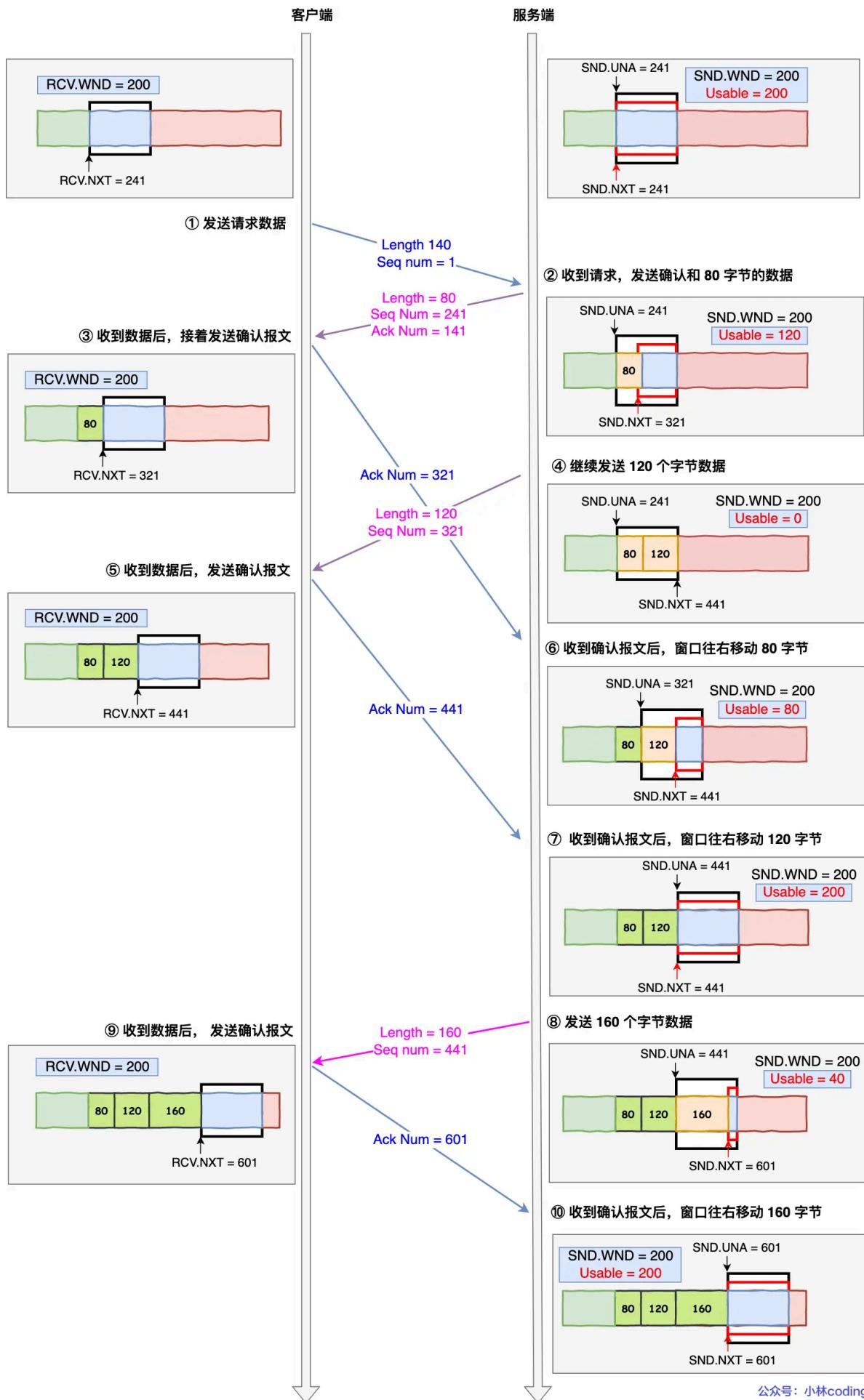
如下图所示，窗口由已发送但未收到ACK确认的数据和未发送但总大小在接收方处理范围内的数据组成。TCP 滑动窗口方案使用三个指针来跟踪在四个传输类别中的每一个类别中的字节。其中两个指针是绝对指针（指特定的序列号），一个是相对指针（需要做偏移）。



接收窗口的大小是**约等于**发送窗口的大小的。因为滑动窗口并不是一成不变的。比如，当接收方的应用进程读取数据的速度非常快的话，这样的话接收窗口可以很快的就空缺出来。那么新的接收窗口大小，是通过 TCP 报文中的 Windows 字段来告诉发送方。那么这个传输过程是存在时延的，所以接收窗口和发送窗口是**约等于**的关系。

流量控制

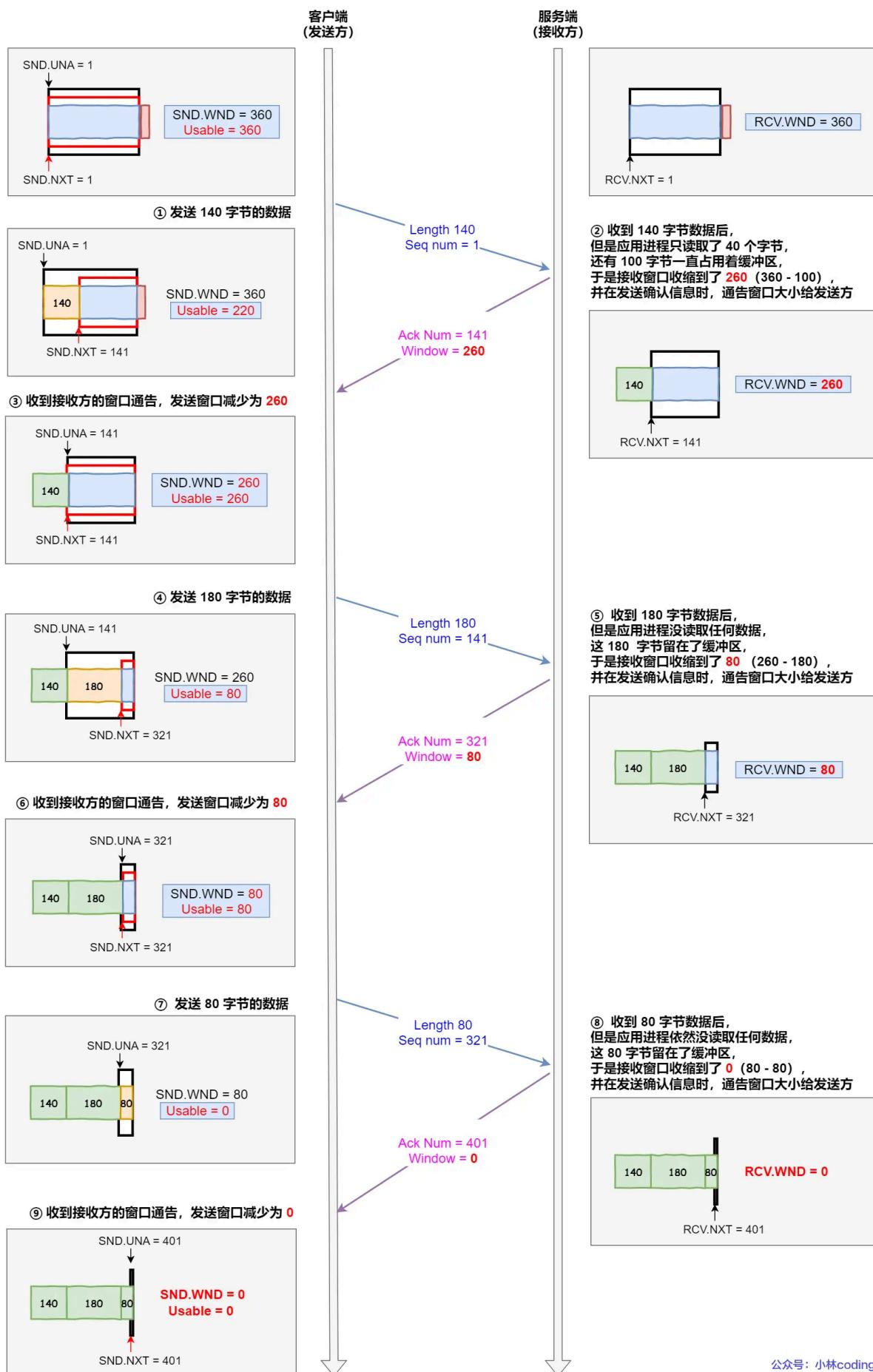
TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是所谓的**流量控制**。



公众号：小林coding

操作系统缓冲区与滑动窗口的关系

发送窗口和接收窗口中所存放的字节数，都是放在操作内存缓冲区中的，而操作系统的缓冲区，会被操作系统调整。



另外一种情况是，当服务端系统资源非常紧张的时候，操作系统可能会直接减少了接收缓冲区大小，这时应用程序又无法及时读取缓存数据，那么这时候就有严重的事情发生了，会出现数据包丢失的现象。为了防止这种情况发生，TCP 规定是不允许同时减少缓存又收缩窗口的，而是采用先收缩窗口，过段时间再减少缓存，这样就可以避免了丢包情况。

窗口关闭

如果窗口大小为 0 时，就会阻止发送方给接收方传递数据，直到窗口变为非 0 为止，这就是窗口关闭。

接收方向发送方通告窗口大小时，是通过 ACK 报文来通告的。那么，当发生窗口关闭时，接收方处理完数据后，会向发送方通告一个窗口非 0 的 ACK 报文，如果这个通告窗口的 ACK 报文在网络中丢失了，那麻烦就大了。这会导致发送方一直等待接收方的非 0 窗口通知，接收方也一直等待发送方的数据，如不采取措施，这种相互等待的过程，会造成了死锁的现象。

为了解决这个问题，TCP 为每个连接设有一个持续定时器，只要 TCP 连接一方收到对方的零窗口通知，就启动持续计时器。如果持续计时器超时，就会发送窗口探测（Window probe）报文，而对方在确认这个探测报文时，给出自己现在的接收窗口大小。

- 如果接收窗口仍然为 0，那么收到这个报文的一方就会重新启动持续计时器；
- 如果接收窗口不是 0，那么死锁的局面就可以被打破了。

窗口探测的次数一般为 3 次，每次大约 30-60 秒（不同的实现可能会不一样）。如果 3 次过后接收窗口还是 0 的话，有的 TCP 实现就会发 RST 报文来中断连接。其实也就是 TCP 的 keep_alive 机制。应用程序若想使用 TCP 保活机制需要通过 socket 接口设置 SO_KEEPALIVE 选项才能够生效，如果没有设置，那么就无法使用 TCP 保活机制。

糊涂窗口综合症

如果接收方太忙了，来不及取走接收窗口里的数据，那么就会导致发送方的发送窗口越来越小。到最后，如果接收方腾出几个字节并告诉发送方现在有几个字节的窗口，而发送方会义无反顾地发送这几个字节，这就是糊涂窗口综合症。要知道，我们的 TCP + IP 头有 40 个字节，为了传输那几个字节的数据，要搭上这么大的开销，这太不经济了。

要解决糊涂窗口综合症，就要同时解决上面两个问题就可以了：

- 让接收方不通告小窗口给发送方
- 让发送方避免发送小数据

怎么让接收方不通告小窗口呢？

接收方通常的策略如下：当「窗口大小」小于 $\min(\text{MSS}, \text{缓存空间}/2)$ ，也就是小于 MSS 与 1/2 缓存大小中的最小值时，就会向发送方通告窗口为 0，也就阻止了发送方再发数据过来。等到接收方处理了一些数据后，窗口大小 $>= \text{MSS}$ ，或者接收方缓存空间有一半可以使用，就可以把窗口打开让发送方发送数据过来。

怎么让发送方避免发送小数据呢？

发送方通常的策略如下：使用 Nagle 算法（拉革勒算法），该算法的思路是延时处理，只有满足下面两个条件中的任意一个条件，才可以发送数据：

- 条件一：要等到窗口大小 $>= \text{MSS}$ 并且 数据大小 $>= \text{MSS}$ ；
- 条件二：收到之前发送数据的 ack 回包；

只要上面两个条件都不满足，发送方一直在囤积数据，直到满足上面的发送条件。

所以，接收方得满足「不通告小窗口给发送方」+ 发送方开启 Nagle 算法，才能避免糊涂窗口综合症。Nagle 算法默认是打开的，如果对于一些需要小数据包交互的程序，比如，telnet 或 ssh 这样的交互性比较强的程序，则需要关闭 Nagle 算法。可以在 Socket 设置 `TCP_NODELAY` 选项来关闭这个算法（关闭 Nagle 算法没有全局参数，需要根据每个应用自己的特点来关闭）

拥塞控制

流量控制是避免「发送方」的数据填满「接收方」的缓存，但是并不知道网络中发生了什么。计算机网络都处在一个共享的环境。因此也有可能会因为其他主机之间的通信使得网络拥堵。在在网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，这时 TCP 就会重传数据，但是一重传就会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这个情况就会进入恶性循环被不断地放大....于是，就有了拥塞控制，控制的目的就是避免「发送方」的数据填满整个网络。为了在「发送方」调节所要发送数据的量，定义了一个叫做「拥塞窗口」的概念。

拥塞窗口 `cwnd` 是发送方维护的一个的状态变量，它会根据网络的拥塞程度动态变化的。我们在前面提到过发送窗口 `swnd` 和接收窗口 `rwnd` 是约等于的关系，那么由于加入了拥塞窗口的概念后，此时发送窗口的值是 $\text{swnd} = \min(\text{cwnd}, \text{rwnd})$ ，也就是拥塞窗口和接收窗口中的最小值。

拥塞窗口 `cwnd` 变化的规则：

- 只要网络中没有出现拥塞，`cwnd` 就会增大；
- 但网络中出现了拥塞，`cwnd` 就减少；

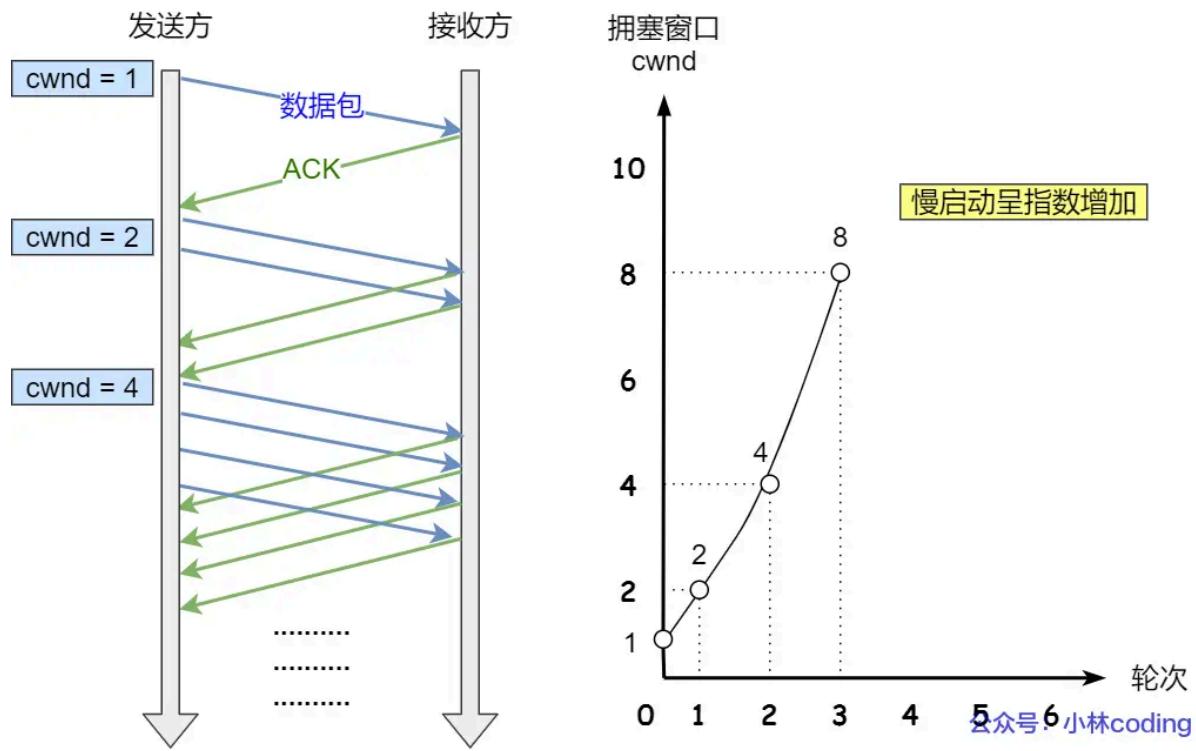
只要「发送方」没有在规定时间内接收到 ACK 应答报文，也就是发生了超时重传，就会认为网络出现了拥塞。

慢启动

慢启动的算法记住一个规则就行：当发送方每收到一个 ACK，拥塞窗口 `cwnd` 的大小就会加 1。这里假定拥塞窗口 `cwnd` 和发送窗口 `swnd` 相等，下面举个栗子：

- 连接建立完成后，一开始初始化 `cwnd = 1`，表示可以传一个 `MSS` 大小的数据。
- 当收到一个 ACK 确认应答后，`cwnd` 增加 1，于是一次能够发送 2 个
- 当收到 2 个的 ACK 确认应答后，`cwnd` 增加 2，于是就可以比之前多发 2 个，所以这一次能够发送 4 个
- 当这 4 个的 ACK 确认到来的时候，每个确认 `cwnd` 增加 1，4 个确认 `cwnd` 增加 4，于是就可以比之前多发 4 个，所以这一次能够发送 8 个。

慢启动算法的变化过程如下图：



可以看出慢启动算法，发包的个数是**指数性的增长**。有一个叫慢启动门限 `ssthresh` (slow start threshold) 状态变量。

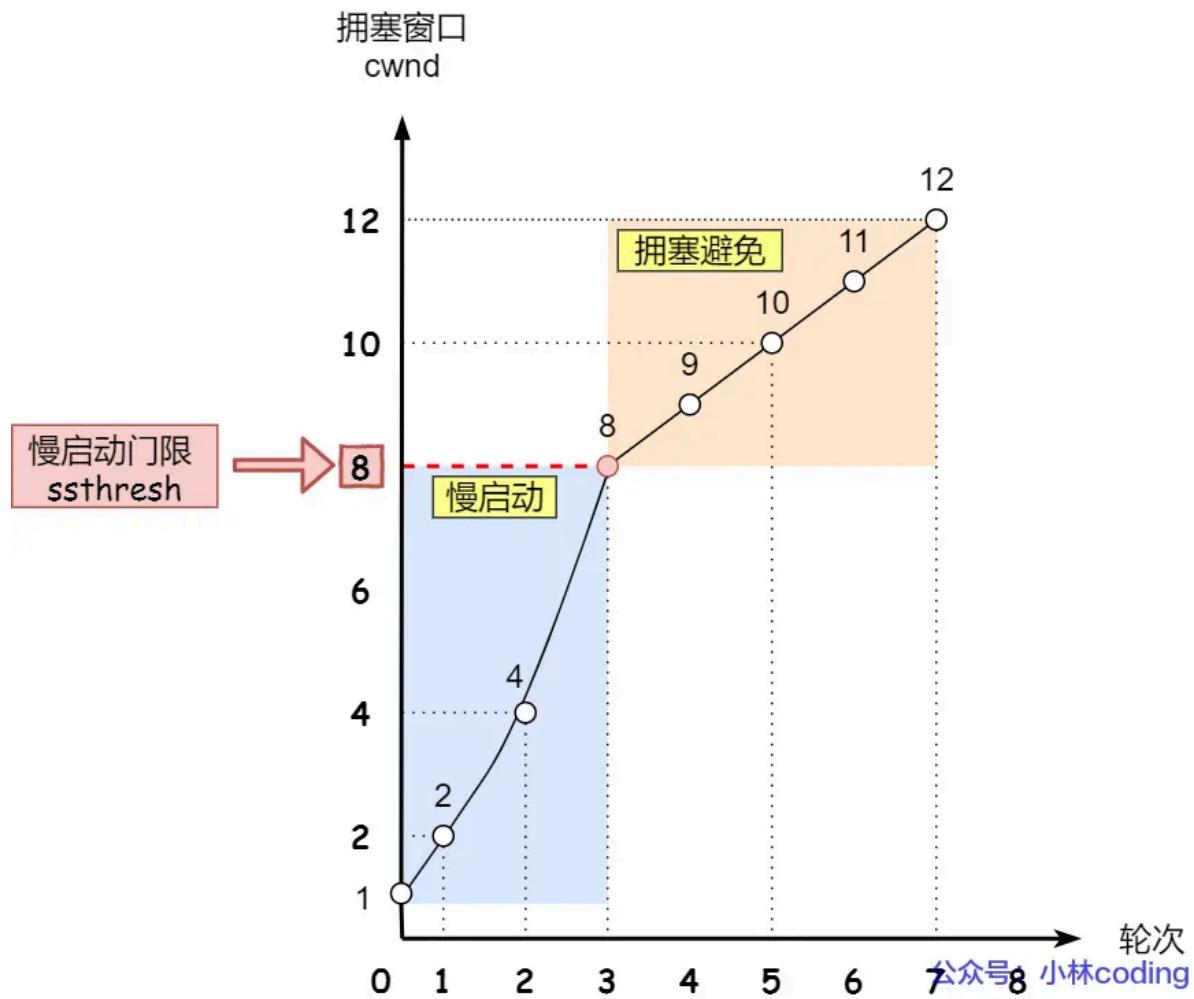
- 当 `cwnd < ssthresh` 时，使用慢启动算法。
- 当 `cwnd >= ssthresh` 时，就会使用「拥塞避免算法」。

拥塞避免算法

一般来说 `ssthresh` 的大小是 65535 字节。那么进入拥塞避免算法后，它的规则是：**每当收到一个 ACK 时，`cwnd` 增加 $1/cwnd$** 。接上前面的慢启动的栗子，现假定 `ssthresh` 为 8：

- 当 8 个 ACK 应答确认到来时，每个确认增加 $1/8$ ，8 个 ACK 确认 `cwnd` 一共增加 1，于是这一次能够发送 9 个 `MSS` 大小的数据，变成了**线性增长**。

拥塞避免算法的变化过程如下图：



所以，拥塞避免算法就是将原本慢启动算法的指数增长变成了线性增长，还是增长阶段，但是增长速度缓慢了一些。就这么一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。当触发了重传机制，也就进入了「拥塞发生算法」。

拥塞发生

重传机制主要有两种：

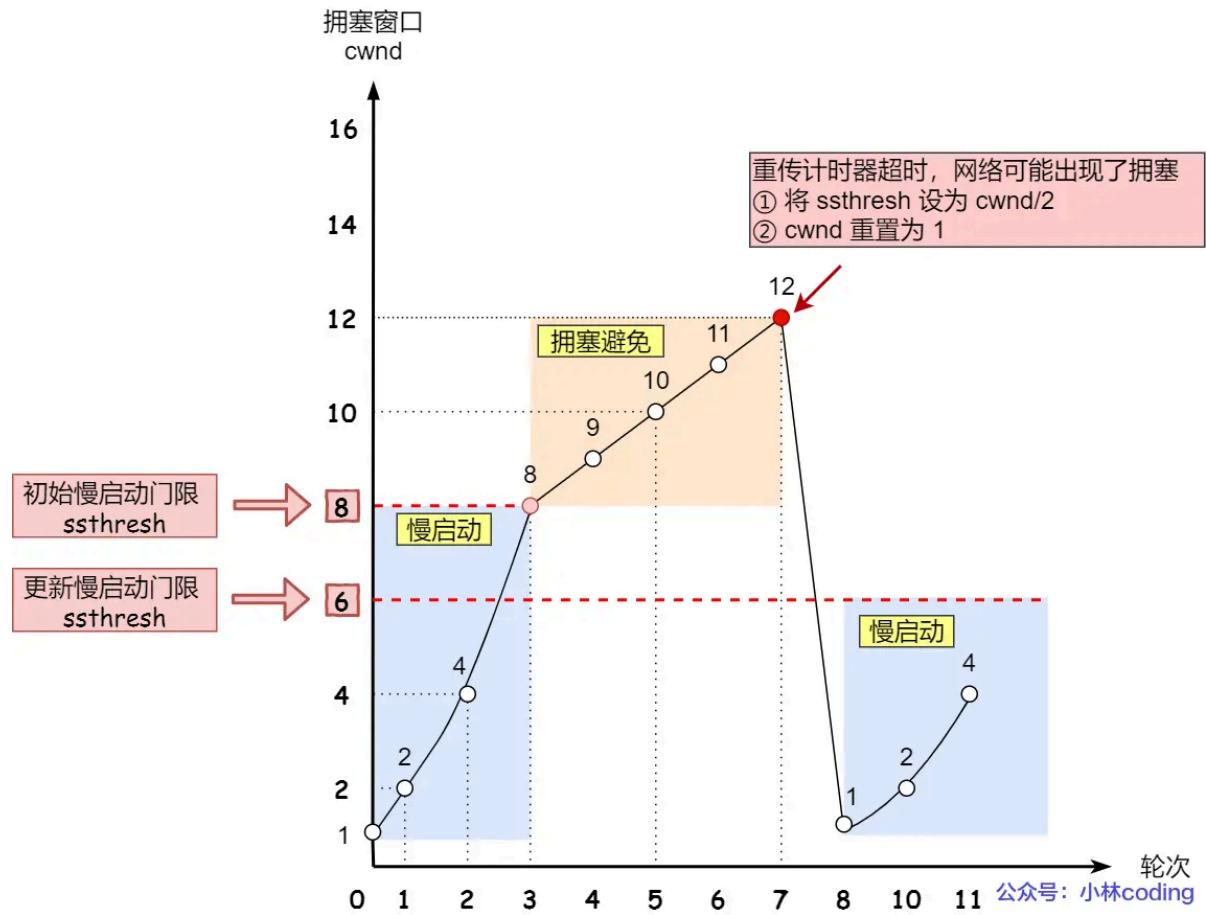
- 超时重传
- 快速重传

这两种使用的拥塞发送算法是不同的，接下来分别来说说。

当发生了「超时重传」，则就会使用拥塞发生算法。这个时候，`ssthresh` 和 `cwnd` 的值会发生变化：

- `ssthresh` 设为 `cwnd/2`，
- `cwnd` 重置为 1（是恢复为 `cwnd` 初始值，我这里假定 `cwnd` 初始值 1）

接着，就重新开始慢启动，慢启动是会突然减少数据流的。这真是一旦「超时重传」，马上回到解放前。但是这种方式太激进了，反应也很强烈，会造成网络卡顿。



当接收方发现丢了一个中间包的时候，发送三次前一个包的 ACK，于是发送端就会快速地重传，不必等待超时再重传。TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分，则 `ssthresh` 和 `cwnd` 变化如下：

- `cwnd = cwnd/2`，也就是设置为原来的一半；
- `ssthresh = cwnd`；
- 进入快速恢复算法

快速恢复

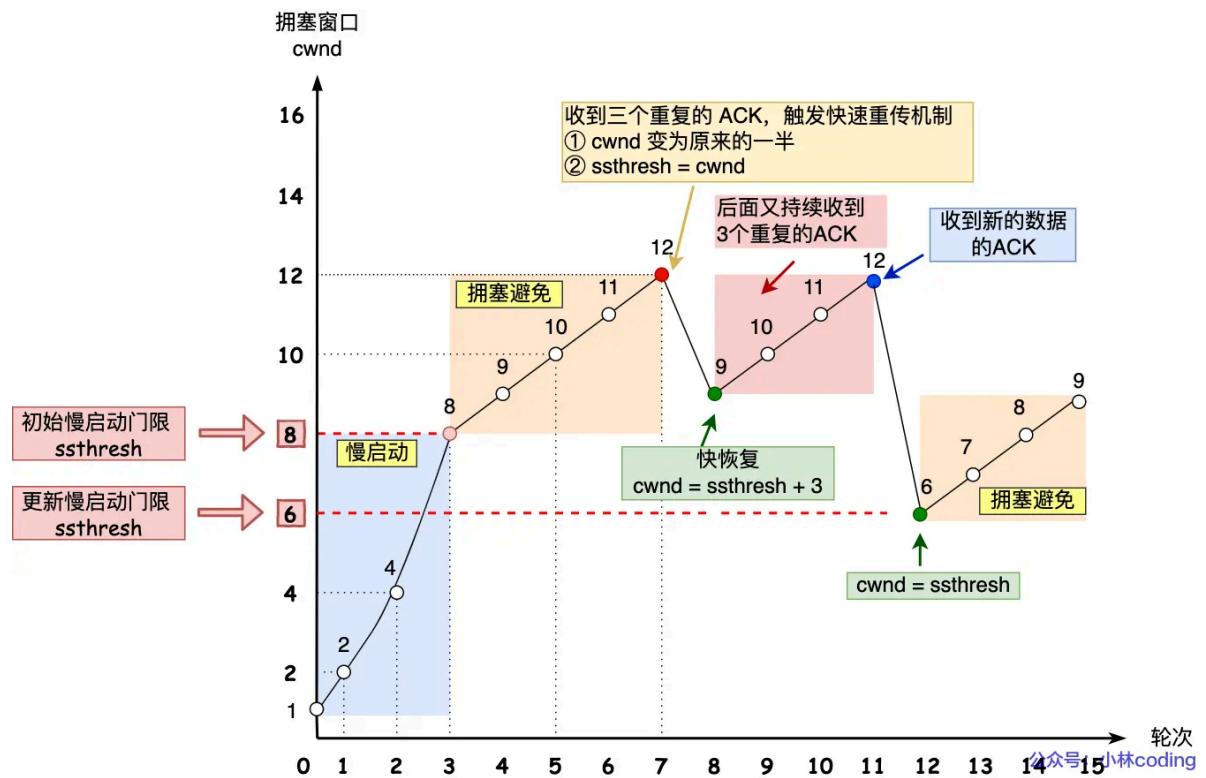
快速重传和快速恢复算法一般同时使用，正如前面所说，进入快速恢复之前，`cwnd` 和 `ssthresh` 已被更新了：

- `cwnd = cwnd/2`，也就是设置为原来的一半；
- `ssthresh = cwnd`；

然后，进入快速恢复算法如下：

- 拥塞窗口 `cwnd = ssthresh + 3`（3 的意思是确认有 3 个数据包被收到了）；
- 重传丢失的数据包；
- 如果再收到重复的 ACK，那么 `cwnd` 增加 1；
- 如果收到新数据的 ACK 后，把 `cwnd` 设置为第一步中的 `ssthresh` 的值，原因是该 ACK 确认了新的数据，说明从 `duplicated ACK` 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态；

快速恢复算法的变化过程如下图：



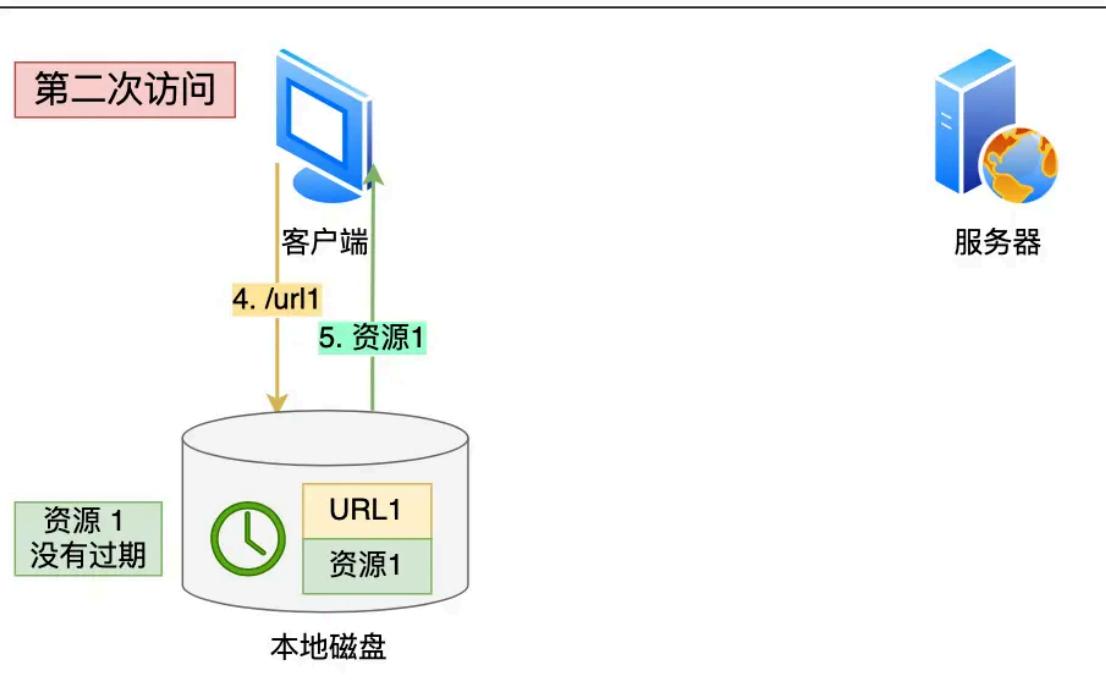
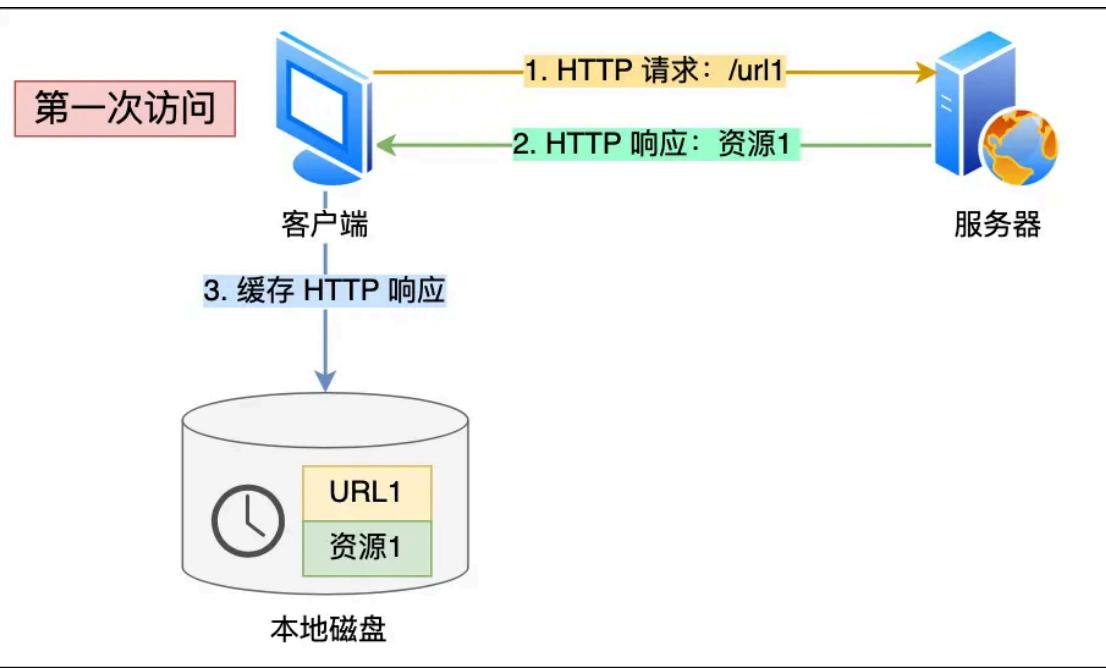
也就是没有像「超时重传」一夜回到解放前，而是还在比较高的值，后续呈线性增长。

HTTP1.1如何优化

1. 尽可能避免发送HTTP请求：利用缓存技术，使用url做为key，响应内容做为value，响应头部会有一个过期时间；etag字段包含响应的摘要，客户端二次请求时，服务端校验这个摘要，如果没变化则返回304 Not modified响应。
2. 减少重定向请求次数：由代理服务器完成重定向
3. 合并请求（HTTP1.1 pipeline技术默认不启用，造成队头阻塞），这里的合并请求还是用一个TCP连接，只是我们把资源进行合并，比如把小内容打包成大文件。
4. 延迟获取内容，只要用户不请求就不需要全部获取过来。
5. 减少HTTP响应大小
 1. gzip无损压缩，http请求头部说明压缩算法
 2. 有损压缩，http请求头部说明质量因子

如何避免发送HTTP请求

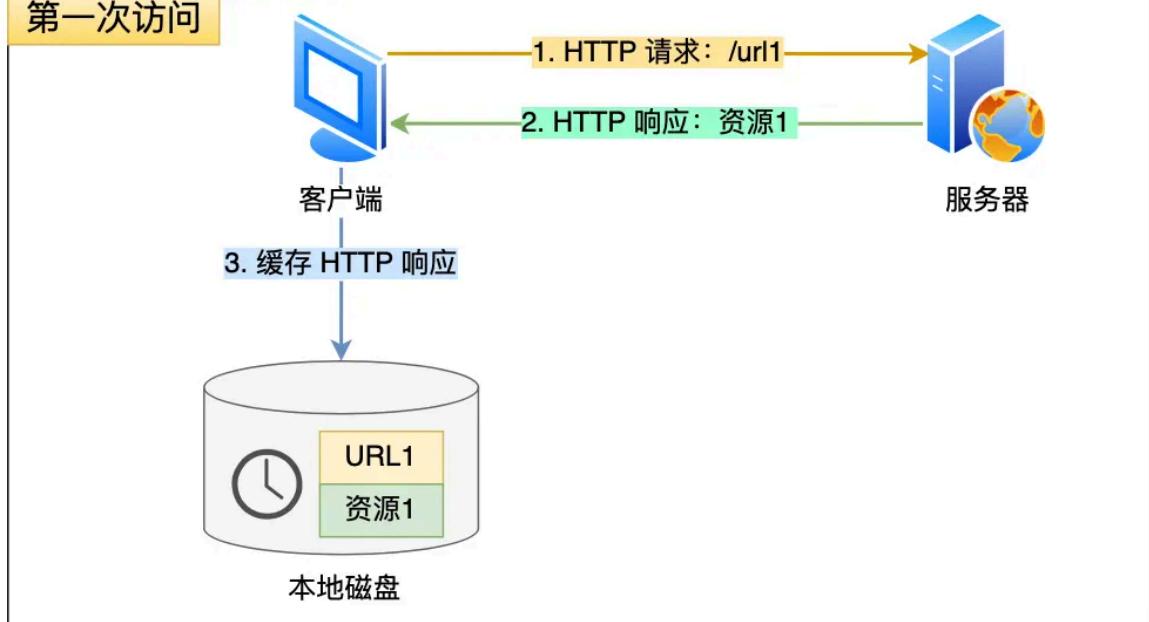
避免发送 HTTP 请求的方法就是通过**缓存技术**，HTTP 协议的头部有不少是针对缓存的字段。客户端会把第一次请求以及响应的数据保存在本地磁盘上，其中将请求的 URL 作为 key，而响应作为 value，两者形成映射关系。这样当后续发起相同的请求时，就可以先在本地磁盘上通过 key 查到对应的 value，也就是响应，如果找到了，就直接从本地读取该响应。毋庸置疑，读取本地磁盘的速度肯定比网络请求快得多，如下图：



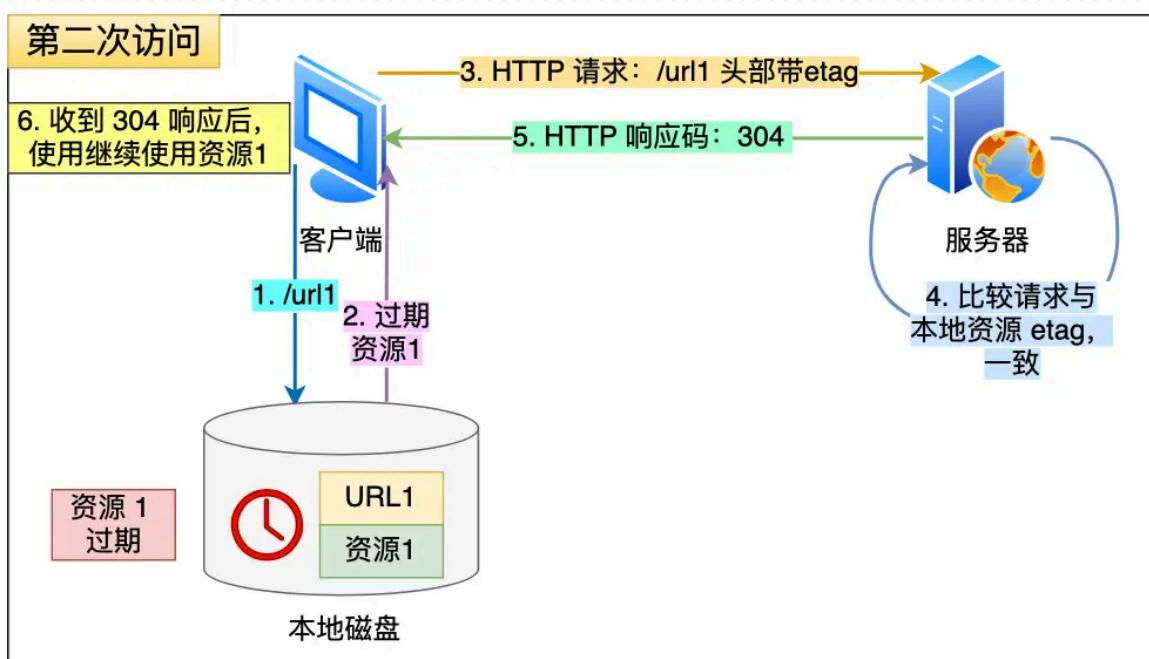
万一缓存的响应不是最新的，而客户端并不知情，那么该怎么办呢？**服务器在发送 HTTP 响应时，会估算一个过期的时间，并把这个信息放到响应头部中，这样客户端在查看响应头部的信息时，一旦发现缓存的响应是过期的，则就会重新发送网络请求。**如果客户端从第一次请求得到的响应头部中发现该响应过期了，客户端重新发送请求，假设服务器上的资源并没有变更，还是老样子，那么你觉得还要在服务器的响应带上这个资源吗？

很显然不带的话，可以提高 HTTP 协议的性能，那具体如何做到呢？**只需要客户端在重新发送请求时，在请求的 Etag 头部带上第一次请求的响应头部中的摘要，这个摘要是唯一标识响应的资源，当服务器收到请求后，会将本地资源的摘要与请求中的摘要做个比较。**如果不同，那么说明客户端的缓存已经没有价值，服务器在响应中带上最新的资源。如果相同，说明客户端的缓存还是可以继续使用的，那么服务器仅返回不含有包体的 304 Not Modified 响应，告诉客户端仍然有效，这样就可以减少响应资源在网络中传输的延时，如下图：

第一次访问



第二次访问

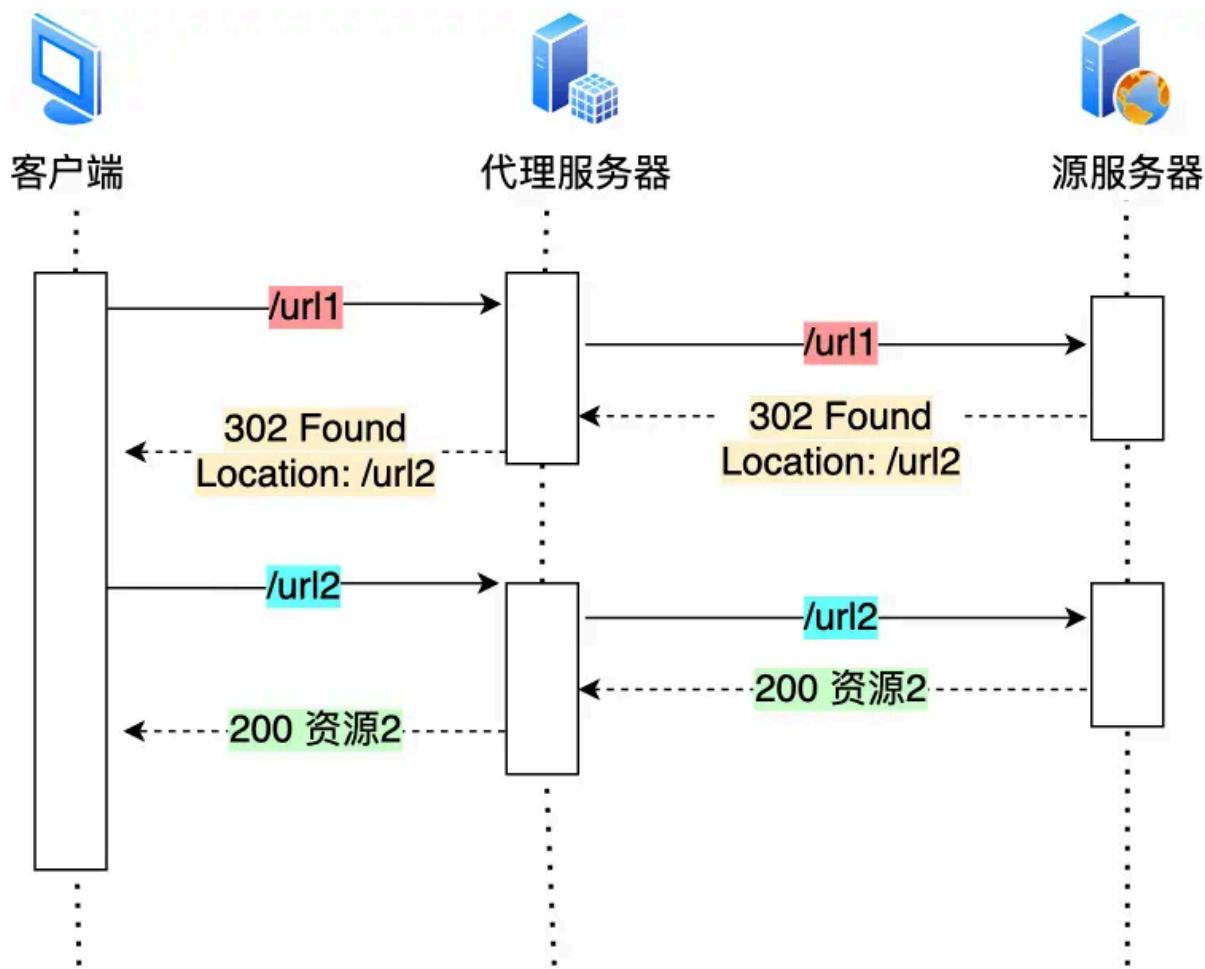


如何减少 HTTP 请求次数?

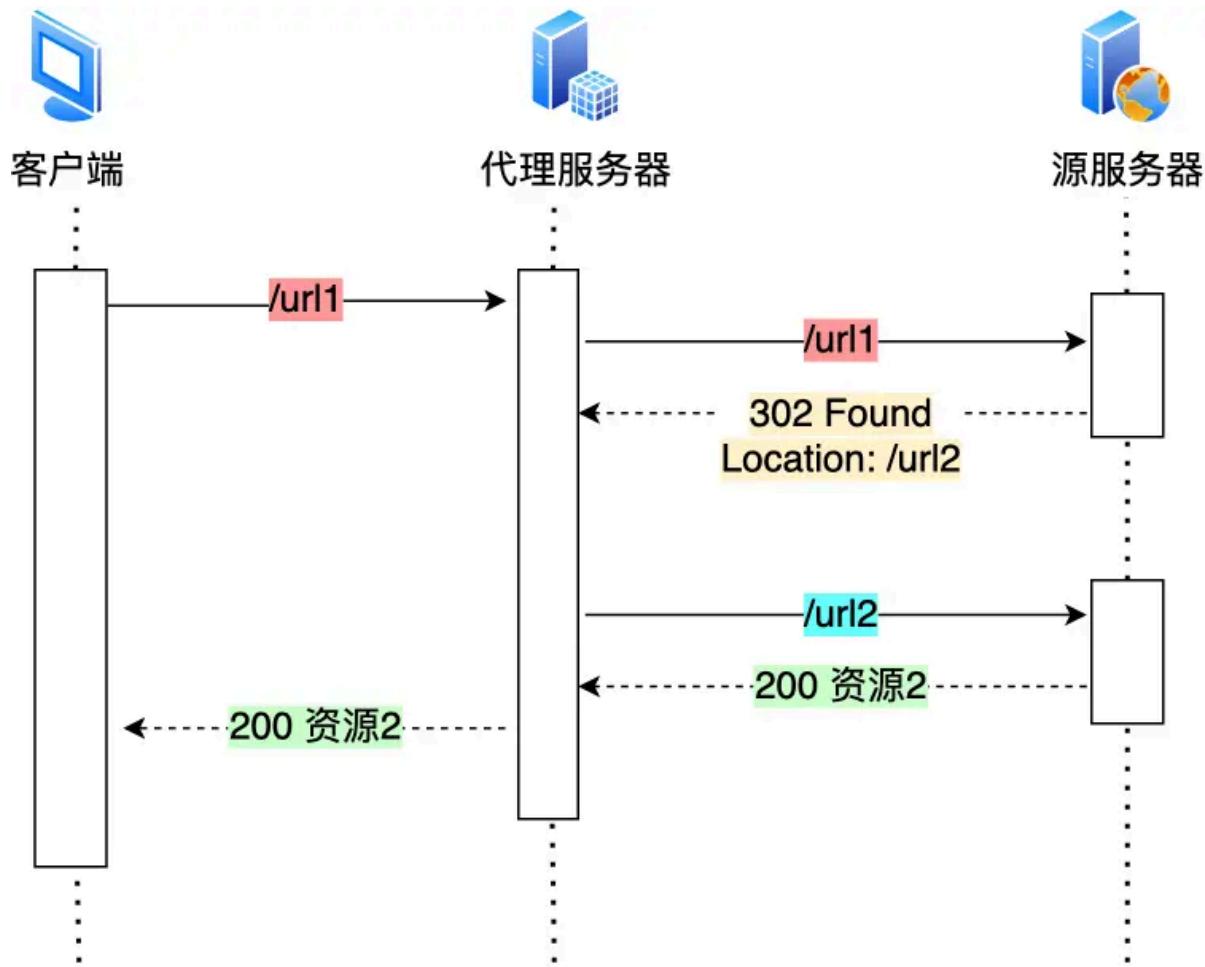
减少重定向请求次数

重定向请求：服务器上的一个资源可能由于迁移、维护等原因从 url1 移至 url2 后，而客户端不知情，它还是继续请求 url1，这时服务器不能粗暴地返回错误，而是通过 302 响应码和 Location 头部，告诉客户端该资源已经迁移至 url2 了，于是客户端需要再发送 url2 请求以获得服务器的资源。

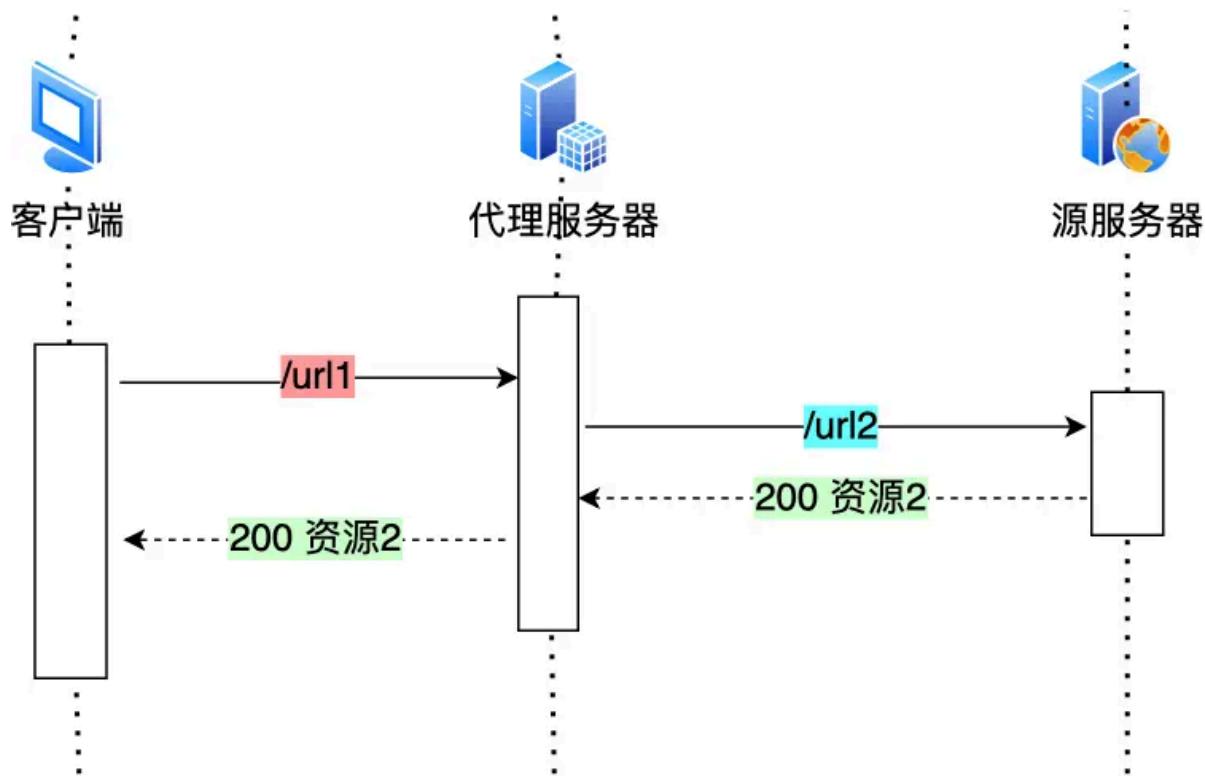
那么，如果重定向请求越多，那么客户端就要多次发起 HTTP 请求，每一次的 HTTP 请求都得经过网络，这无疑会降低网络性能。另外，服务端这一方往往不只有一台服务器，比如源服务器上一级是代理服务器，然后代理服务器才与客户端通信，这时客户端重定向就会导致客户端与代理服务器之间需要 2 次消息传递，如下图：



如果重定向的工作交由代理服务器完成，就能减少 HTTP 请求次数了，如下图：



而且当代理服务器知晓了重定向规则后，可以进一步减少消息传递次数，如下图：



除了 302 重定向响应码，还有其他一些重定向的响应码，你可以从下图看到：

响应状态码	描述	意义
301	Moved Permanently	资源永久的重定向到另一个 URI 中
302	Found	资源临时的重定向到另一个 URI 中
303	See Other	重定向到其他资源，常用于 POST/PUT 方法的响应中
307	Temporary Redirect	类似 302 的临时重定向，但请求方法不得改变
308	Permanent Redirect	类似 301 的永久重定向，但请求方法不得改变

合并请求

如果把多个访问小文件的请求合并成一个大的请求，虽然传输的总资源还是一样，但是减少请求，也就意味着减少了重复发送的 HTTP 头部。另外由于 HTTP/1.1 是请求响应模型，如果第一个发送的请求，未收到对应的响应，那么后续的请求就不会发送（PS：HTTP/1.1 管道模式是默认不使用的，所以讨论 HTTP/1.1 的队头阻塞问题，是不考虑管道模式的），于是为了防止单个请求的阻塞，所以一般浏览器会同时发起 5-6 个请求，每一个请求都是不同的 TCP 连接，那么如果合并了请求，也就会减少 TCP 连接的数量，因而省去了 TCP 握手和慢启动过程耗费的时间。

这种方式就是通过将多个小图片合并成一个大图片（再根据 CSS 数据把大图片切割成多张小图片）来减少 HTTP 请求的次数，以减少 HTTP 请求的次数，从而减少网络的开销。

除了将小图片合并成大图片的方式，还有服务端使用 webpack 等打包工具将 js、css 等资源合并打包成大文件，也是能达到类似的效果。

另外，还可以将图片的二进制数据用 `base64` 编码后，以 URL 的形式嵌入到 HTML 文件，跟随 HTML 文件一并发送。

可以看到，**合并请求的方式就是合并资源，以一个大资源的请求替换多个小资源的请求**。但是这样的合并请求会带来新的问题，**当大资源中的某一个小资源发生变化后，客户端必须重新下载整个完整的大资源文件**，这显然带来了额外的网络消耗。

延迟发送请求

不要一口气吃成大胖子，一般 HTML 里会含有很多 HTTP 的 URL，当前不需要的资源，我们没必要也获取过来，于是可以通过「**按需获取**」的方式，来减少第一时间的 HTTP 请求次数。请求网页的时候，没必要把全部资源都获取到，而是只获取当前用户所看到的页面资源，当用户向下滑动页面的时候，再向服务器获取接下来的资源，这样就达到了延迟发送请求的效果。

如何减少 HTTP 响应的数据大小？

对于 HTTP 的请求和响应，通常 HTTP 的响应的数据大小会比较大。我们可以考虑对响应的资源进行**压缩**，这样就可以减少响应的数据大小，从而提高网络传输的效率。

压缩的方式一般分为 2 种，分别是：

- 无损压缩；
- 有损压缩；

无损压缩

`gzip` 就是比较常见的无损压缩。客户端支持的压缩算法，会在 HTTP 请求中通过头部中的 `Accept-Encoding` 字段告诉服务器：

```
Accept-Encoding: gzip, deflate, br
```

服务器收到后，会从中选择一个服务器支持的或者合适的压缩算法，然后使用此压缩算法对响应资源进行压缩，最后通过响应头部中的 `Content-Encoding` 字段告诉客户端该资源使用的压缩算法。

```
Content-Encoding: gzip
```

`gzip` 的压缩效率相比 Google 推出的 Brotli 算法还是差点意思，也就是上文中的 `br`，所以如果可以，服务器应该选择压缩效率更高的 `br` 压缩算法。

有损压缩

有损压缩主要将次要的数据舍弃，**牺牲一些质量来减少数据量、提高压缩比，这种方法经常用于压缩多媒体数据，比如音频、视频、图片。可以通过 HTTP 请求头部中的 `Accept` 字段里的「`q` 质量因子」，告诉服务器期望的资源质量。**

```
Accept: audio/*; q=0.2, audio/basic
```

HTTP 与 HTTPS 有哪些区别？

- HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
- HTTP 连接建立相对简单，TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
- 两者的默认端口不一样，HTTP 默认口号是 80，HTTPS 默认口号是 443。
- HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

HTTPS 解决了 HTTP 的哪些问题？

HTTP 由于是明文传输，所以安全上存在以下三个风险：

- **窃听风险**，比如通信链路上可以获取通信内容，用户号容易没。
- **篡改风险**，比如强制植入垃圾广告，视觉污染，用户眼容易瞎。
- **冒充风险**，比如冒充淘宝网站，用户钱容易没。

HTTPS 在 HTTP 与 TCP 层之间加入了 SSL/TLS 协议，可以很好的解决了上述的风险：

- **混合加密的方式实现信息的机密性**，解决了窃听的风险。
- **摘要算法的方式来实现完整性**，它能够为数据生成独一无二的「指纹」，指纹用于校验数据的完整性，解决了篡改的风险。
- 将服务器公钥放入到**数字证书**中，解决了冒充的风险。

1. 混合加密

通过**混合加密**的方式可以保证信息的**机密性**，解决了窃听的风险：

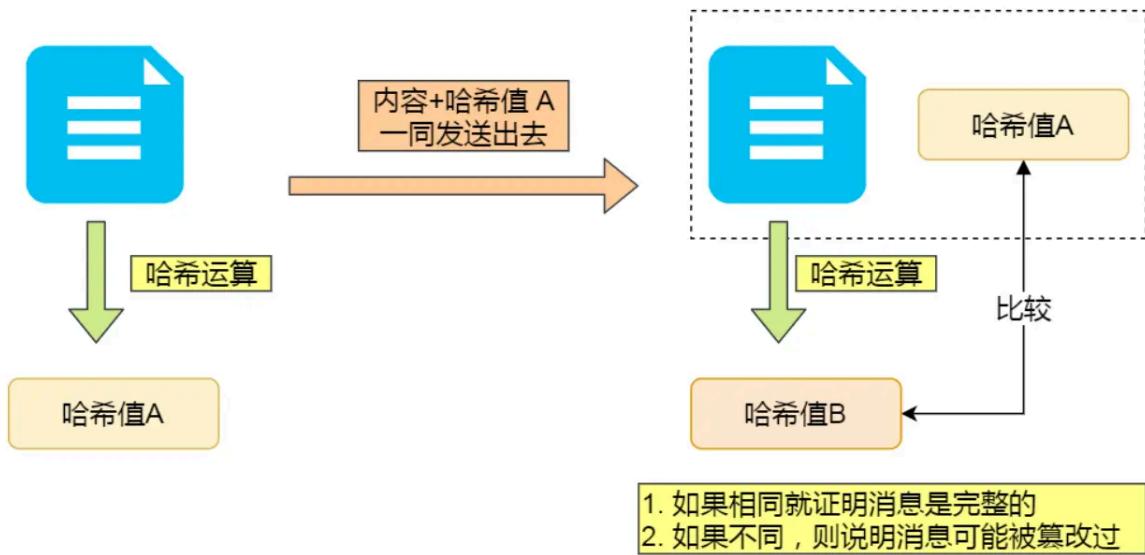
- 在通信建立前采用**非对称加密**的方式交换「会话秘钥」，后续就不再使用非对称加密。
- 在通信过程中全部使用**对称加密**的「会话秘钥」的方式加密明文数据。

采用「混合加密」的方式的原因：

- **对称加密**只使用一个密钥，运算速度快，密钥必须保密，无法做到安全的密钥交换。
- **非对称加密**使用两个密钥：公钥和私钥，公钥可以任意分发而私钥保密，解决了密钥交换问题但速度慢。

2. 摘要算法 + 数字签名

在计算机里会用**摘要算法（哈希函数）**来计算出内容的哈希值，也就是内容的「指纹」，这个**哈希值**是唯一的，且无法通过哈希值推导出内容。



通过哈希算法可以确保内容不会被篡改，但是并不能保证「内容 + 哈希值」不会被中间人替换，因为这里缺少对客户端收到的消息是否来源于服务端的证明。

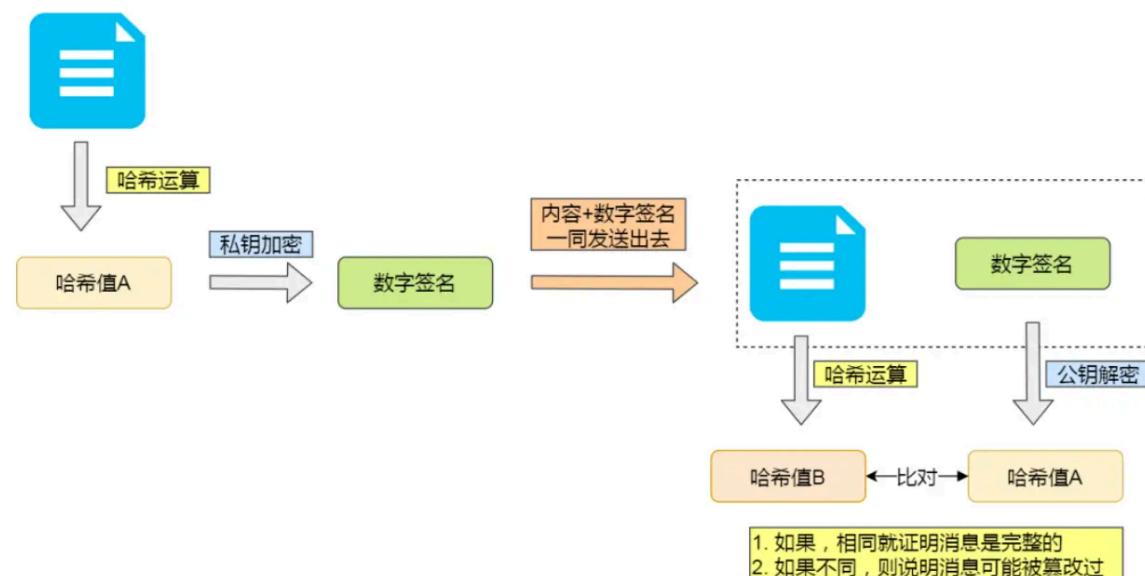
那为了避免这种情况，计算机里会用**非对称加密算法**来解决，共有两个密钥：

- 一个是公钥，这个是可以公开给所有人的；
- 一个是私钥，这个必须由本人管理，不可泄露。

这两个密钥可以**双向加解密**的，流程的不同，意味着目的也不相同：

- 公钥加密，私钥解密。**这个目的是为了**保证内容传输的安全**，因为被公钥加密的内容，其他人是无法解密的，只有持有私钥的人，才能解密出实际的内容；
- 私钥加密，公钥解密。**这个目的是为了**保证消息不会被冒充**，因为私钥是不可泄露的，如果公钥能正常解密出私钥加密的内容，就能证明这个消息是来源于持有私钥身份的人发送的。

所以非对称加密的用途主要在于通过「**私钥加密，公钥解密**」的方式，来确认消息的身份，我们常说的**数字签名算法**，就是用的是这种方式，不过私钥加密内容不是内容本身，而是**对内容的哈希值加密**。



3. 数字证书

前面我们知道：

- 可以通过哈希算法来保证消息的完整性；
- 可以通过数字签名来保证消息的来源可靠性（能确认消息是由持有私钥的一方发送的）；

但是这还远远不够，还缺少身份验证的环节，万一公钥是被伪造的呢？在计算机里，这个权威的机构就是 CA（数字证书认证机构），将服务器公钥放在数字证书（由数字证书认证机构颁发）中，只要证书是可信的，公钥就是可信的。

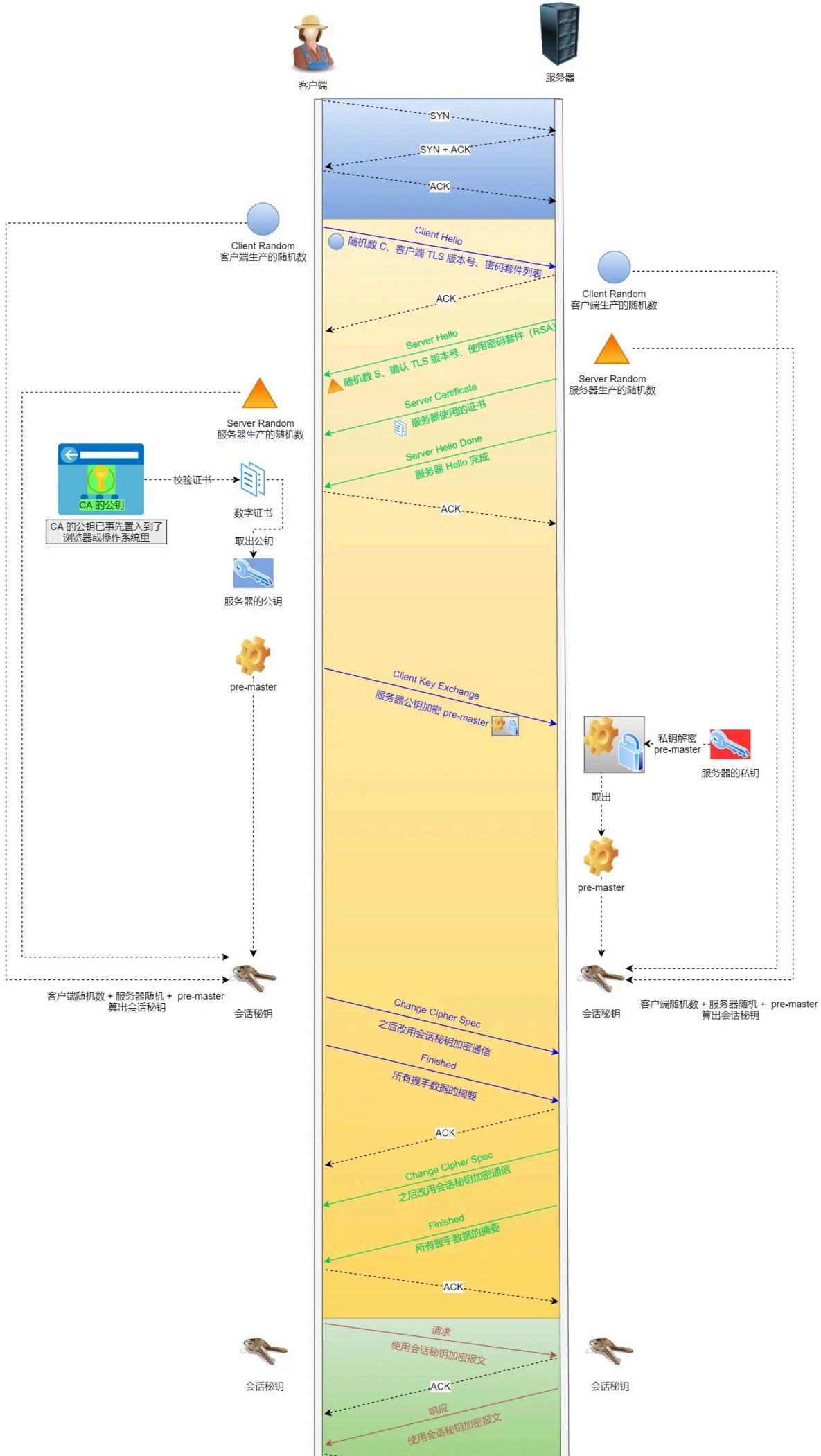


HTTPS 是如何建立连接的？其间交互了什么？

SSL/TLS 协议基本流程：

- 客户端向服务器索要并验证服务器的公钥。
- 双方协商生产「会话秘钥」。
- 双方采用「会话秘钥」进行加密通信。

前两步也就是 SSL/TLS 的建立过程，也就是 TLS 握手阶段。TLS 的「握手阶段」涉及四次通信，使用不同的密钥交换算法，TLS 握手流程也会不一样的，现在常用的密钥交换算法有两种：[RSA 算法 \(opens new window\)](#) 和 [ECDHE 算法 \(opens new window\)](#)。基于 RSA 算法的 TLS 握手过程比较容易理解，所以这里先用这个给大家展示 TLS 握手过程，如下图：





TLS 协议建立的详细流程：

1. ClientHello

首先，由客户端向服务器发起加密通信请求，也就是 `ClientHello` 请求。在这一步，客户端主要向服务器发送以下信息：

- (1) 客户端支持的 TLS 协议版本，如 TLS 1.2 版本。
- (2) 客户端生产的随机数 (`client Random`)，后面用于生成「会话秘钥」条件之一。
- (3) 客户端支持的密码套件列表，如 RSA 加密算法。

2. ServerHello

服务器收到客户端请求后，向客户端发出响应，也就是 `ServerHello`。服务器回应的内容有如下内容：

- (1) 确认 TLS 协议版本，如果浏览器不支持，则关闭加密通信。
- (2) 服务器生产的随机数 (`server Random`)，也是后面用于生产「会话秘钥」条件之一。
- (3) 确认的密码套件列表，如 RSA 加密算法。
- (4) 服务器的数字证书。

3. 客户端回应

客户端收到服务器的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务器的数字证书的真实性。如果证书没有问题，客户端会从数字证书中取出服务器的公钥，然后使用它加密报文，向服务器发送如下信息：

- (1) 一个随机数 (`pre-master key`)。该随机数会被服务器公钥加密。
- (2) 加密通信算法改变通知，表示随后的信息都将用「会话秘钥」加密通信。
- (3) **客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时把之前所有内容的数据做个摘要，用来供服务端校验。**

上面第一项的随机数是整个握手阶段的第三个随机数，会发给服务端，所以这个随机数客户端和服务端都是一样的。**服务器和客户端有了这三个随机数 (Client Random、Server Random、pre-master key)**，接着就用双方协商的加密算法，各自生成本次通信的「会话秘钥」。

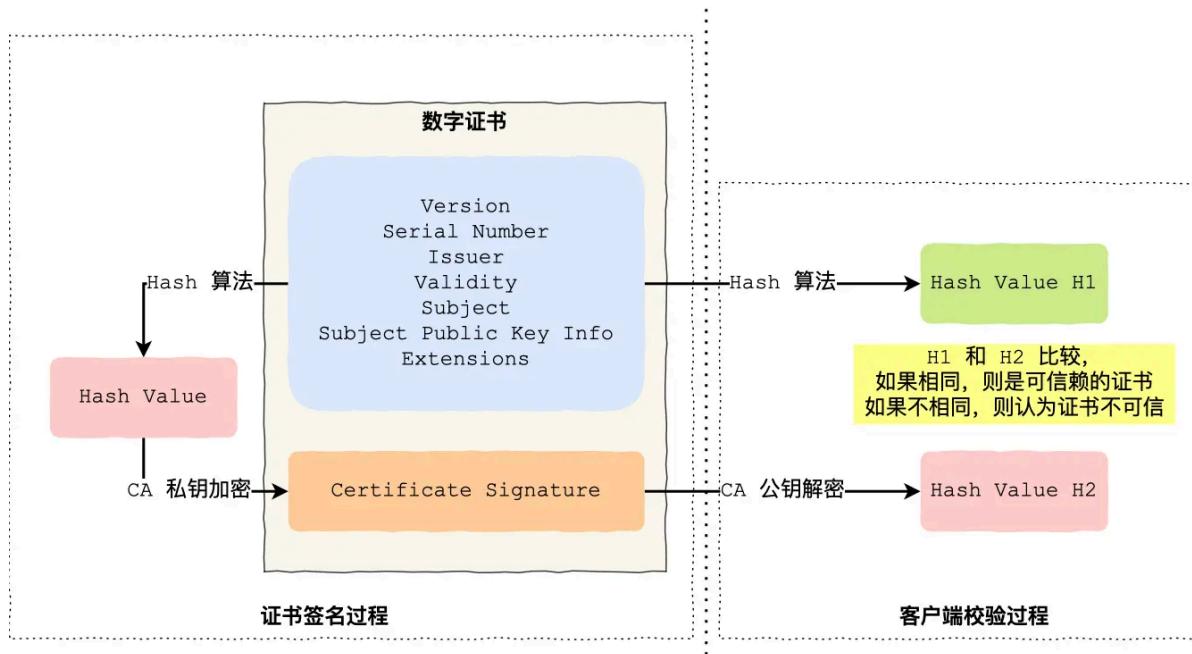
4. 服务器的最后回应

服务器收到客户端的第三个随机数 (`pre-master key`) 之后，通过协商的加密算法，计算出本次通信的「会话秘钥」。然后，向客户端发送最后的信息：

- (1) 加密通信算法改变通知，表示随后的信息都将用「会话秘钥」加密通信。
- (2) **服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时把之前所有内容的数据做个摘要，用来供客户端校验。**

至此，整个 TLS 的握手阶段全部结束。接下来，客户端与服务器进入加密通信，就完全是使用普通的 HTTP 协议，只不过用「会话秘钥」加密内容。

如下图所示，为数字证书签发和验证流程：



CA 签发证书的过程，如上图左边部分：

- 首先 CA 会把持有者的公钥、用途、颁发者、有效时间等信息打成一个包，然后对这些信息进行 Hash 计算，得到一个 Hash 值；
- 然后 CA 会使用自己的私钥将该 Hash 值加密，生成 Certificate Signature，也就是 CA 对证书做了签名；
- 最后将 Certificate Signature 添加在文件证书上，形成数字证书；

客户端校验服务端的数字证书的过程，如上图右边部分：

- 首先客户端会使用同样的 Hash 算法获取该证书的 Hash 值 H1；
- 通常浏览器和操作系统中集成了 CA 的公钥信息，浏览器收到证书后可以使用 CA 的公钥解密 Certificate Signature 内容，得到一个 Hash 值 H2；
- 最后比较 H1 和 H2，如果值相同，则为可信赖的证书，否则则认为证书不可信。

但事实上，证书的验证过程中还存在一个证书信任链的问题，因为我们向 CA 申请的证书一般不是根证书签发的，而是由中间证书签发的，比如百度的证书，从下图你可以看到，证书的层级有三级：



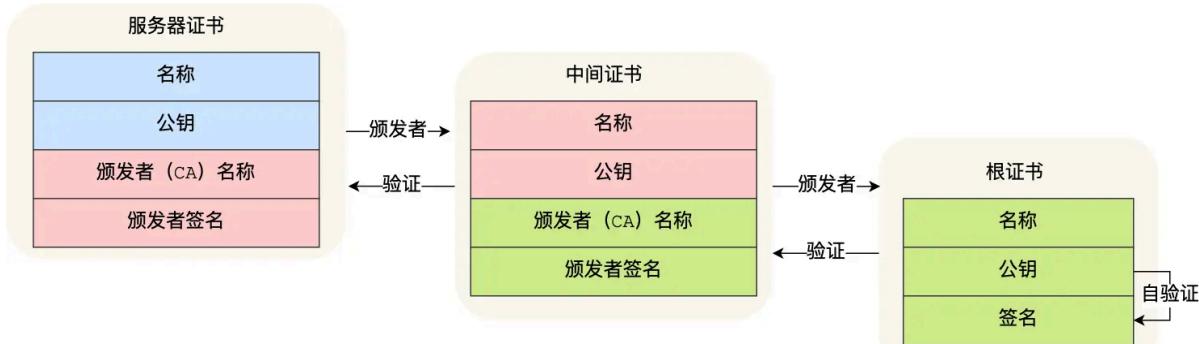
对于这种三级层级关系的证书的验证过程如下：

- 客户端收到 baidu.com 的证书后，发现这个证书的签发者不是根证书，就无法根据本地已有的根证书中的公钥去验证 baidu.com 证书是否可信。于是，客户端根据 baidu.com 证书中的签发者，找到该证书的颁发机构是“GlobalSign Organization Validation CA - SHA256 - G2”，然后向 CA 请求该中间证书。

- 请求到证书后发现“GlobalSign Organization Validation CA - SHA256 - G2”证书是由“GlobalSign Root CA”签发的，由于“GlobalSign Root CA”没有再上级签发机构，说明它是根证书，也就是自签证书。应用软件会检查此证书是否已预载于根证书清单上，如果有，则可以利用根证书中的公钥去验证“GlobalSign Organization Validation CA - SHA256 - G2”证书，如果发现验证通过，就认为该中间证书是可信的。
- “GlobalSign Organization Validation CA - SHA256 - G2”证书被信任后，可以使用“GlobalSign Organization Validation CA - SHA256 - G2”证书中的公钥去验证 baidu.com 证书的可信性，如果验证通过，就可以信任 baidu.com 证书。

在这四个步骤中，最开始客户端只信任根证书 GlobalSign Root CA 证书的，然后“GlobalSign Root CA”证书信任“GlobalSign Organization Validation CA - SHA256 - G2”证书，而“GlobalSign Organization Validation CA - SHA256 - G2”证书又信任 baidu.com 证书，于是客户端也信任 baidu.com 证书。

这样的一层层地验证就构成了一条信任链路，整个证书信任链验证流程如下图所示：



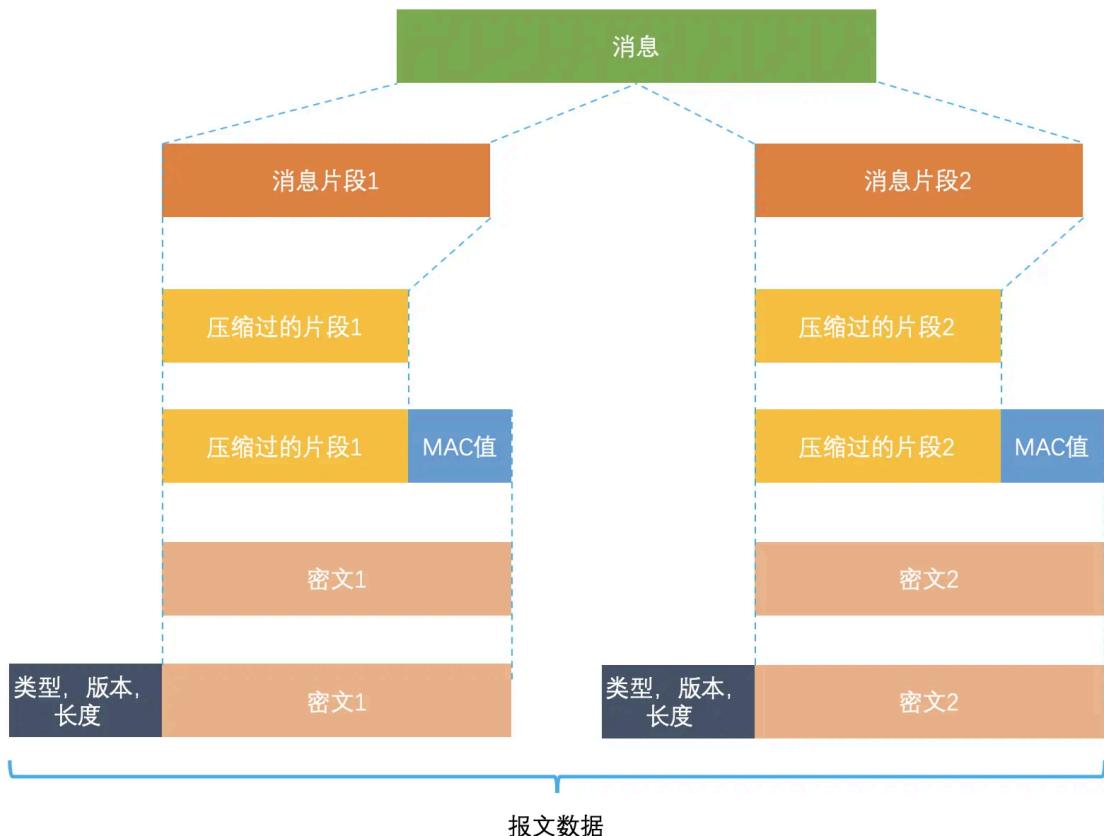
最后一个问题是，为什么需要证书链这么麻烦的流程？Root CA 为什么不直接颁发证书，而是要搞那么多中间层级呢？**这是为了确保根证书的绝对安全性，将根证书隔离地越严格越好，不然根证书如果失守了，那么整个信任链都会有问题。**

HTTPS 的应用数据是如何保证完整性的？

TLS 在实现上分为**握手协议**和**记录协议**两层：

- TLS 握手协议就是我们前面说的 TLS 四次握手的过程，负责协商加密算法和生成对称密钥，后续用此密钥来保护应用程序数据（即 HTTP 数据）；
- **TLS 记录协议负责保护应用程序数据并验证其完整性和来源，所以对 HTTP 数据加密是使用记录协议；**

TLS 记录协议主要负责消息（HTTP 数据）的压缩，加密及数据的认证，过程如下图：



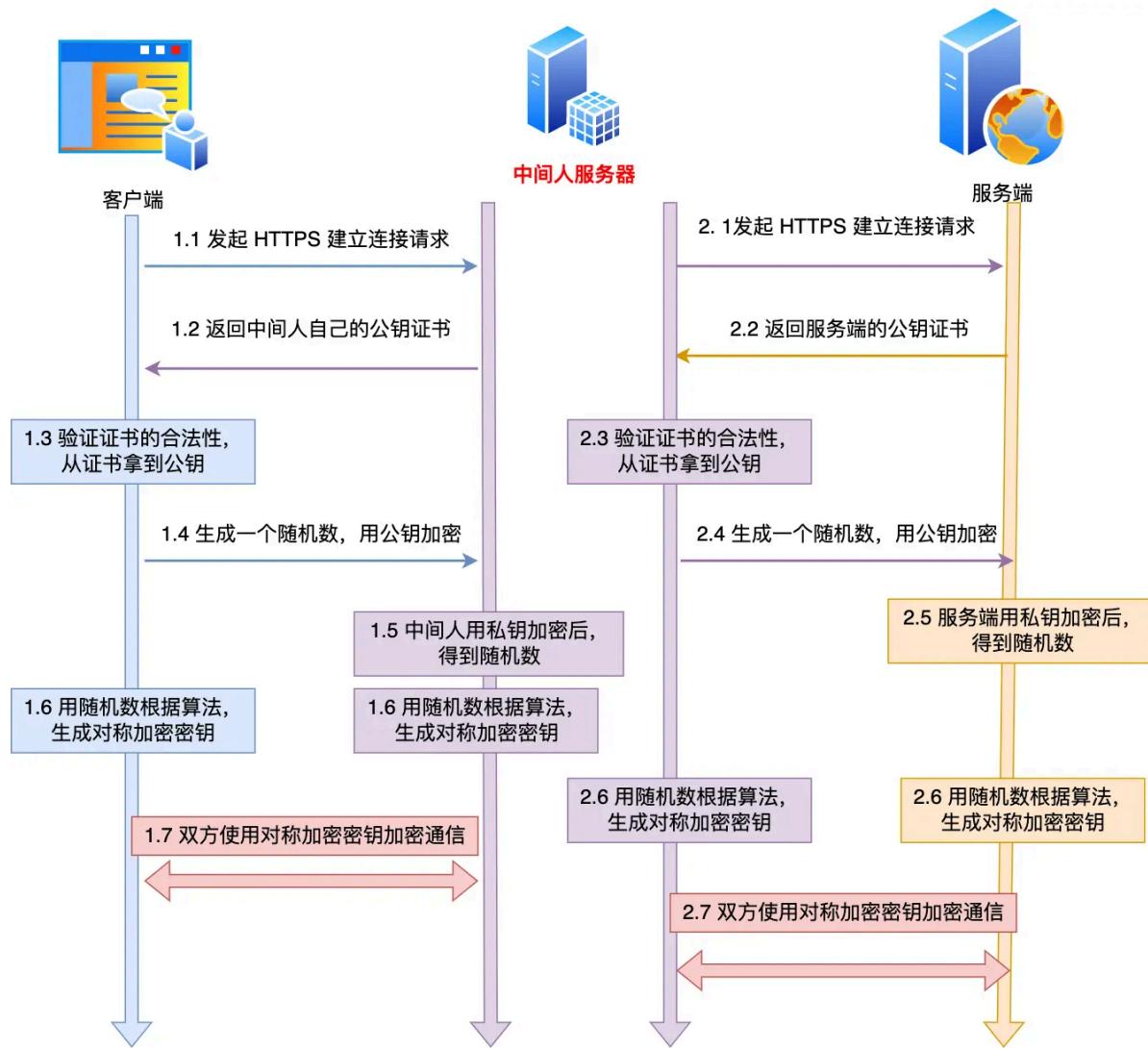
具体过程如下：

- 首先，消息被分割成多个较短的片段,然后分别对每个片段进行压缩。
- 接下来，经过压缩的片段会被**加上消息认证码（MAC 值，这个是通过哈希算法生成的）**，这是为了**保证完整性，并进行数据的认证**。通过附加消息认证码的 MAC 值，可以识别出篡改。与此同时，为了防止重放攻击，在计算消息认证码时，还加上了片段的编码。
- 再接下来，经过压缩的片段再加上消息认证码会一起通过对称密码进行加密。
- 最后，上述经过加密的数据再加上由数据类型、版本号、压缩后的长度组成的报头就是最终的报文数据。

记录协议完成后，最终的报文数据将传递到传输控制协议 (TCP) 层进行传输。

HTTPS 一定安全可靠吗？

这个问题的场景是这样的：客户端通过浏览器向服务端发起 HTTPS 请求时，被「假基站」转发到了一个「中间人服务器」，于是客户端是和「中间人服务器」完成了 TLS 握手，然后这个「中间人服务器」再与真正的服务端完成 TLS 握手。



具体过程如下：

- 客户端向服务端发起 HTTPS 建立连接请求时，然后被「假基站」转发到了一个「中间人服务器」，接着中间人向服务端发起 HTTPS 建立连接请求，此时客户端与中间人进行 TLS 握手，中间人与服务端进行 TLS 握手；
- 在客户端与中间人进行 TLS 握手过程中，中间人会发送自己的公钥证书给客户端，**客户端验证证书的真伪**，然后从证书拿到公钥，并生成一个随机数，用公钥加密随机数发送给中间人，中间人使用私钥解密，得到随机数，此时双方都有随机数，然后通过算法生成对称加密密钥（A），后续客户端与中间人通信就用这个对称加密密钥来加密数据了。
- 在中间人与服务端进行 TLS 握手过程中，服务端会发送从 CA 机构签发的公钥证书给中间人，从证书拿到公钥，并生成一个随机数，用公钥加密随机数发送给服务端，服务端使用私钥解密，得到随机数，此时双方都有随机数，然后通过算法生成对称加密密钥（B），后续中间人与服务端通信就用这个对称加密密钥来加密数据了。
- 后续的通信过程中，中间人用对称加密密钥（A）解密客户端的 HTTPS 请求的数据，然后用对称加密密钥（B）加密 HTTPS 请求后，转发给服务端，接着服务端发送 HTTPS 响应数据给中间人，中间人用对称加密密钥（B）解密 HTTPS 响应数据，然后再用对称加密密钥（A）加密后，转发给客户端。

从客户端的角度看，其实并不知道网络中存在中间人服务器这个角色。那么中间人就可以解开浏览器发起的 HTTPS 请求里的数据，也可以解开服务端响应给浏览器的 HTTPS 响应数据。相当于，中间人能够“偷看”浏览器与服务端之间的 HTTPS 请求和响应的数据。

但是要发生这种场景是有前提的，前提是用户点击接受了中间人服务器的证书。中间人服务器与客户端在 TLS 握手过程中，实际上发送了自己伪造的证书给浏览器，而这个伪造的证书是能被浏览器（客户端）识别出是非法的，于是就会提醒用户该证书存在问题。如果用户执意点击「继续浏览此网站」，相当于用户接受了中间人伪造的证书，那么后续整个 HTTPS 通信都能被中间人监听了。所以，这其实并不能说 HTTPS 不够安全，毕竟浏览器都已经提示证书有问题了，如果用户坚决要访问，那不能怪 HTTPS，得怪自己手贱。

另外，如果你的电脑中毒了，被恶意导入了中间人的根证书，那么在验证中间人的证书的时候，由于你操作系统信任了中间人的根证书，那么等同于中间人的证书是合法的，这种情况下，浏览器是不会弹出证书存在问题的风险提醒的。

这其实也不关 HTTPS 的事情，是你电脑中毒了才导致 HTTPS 数据被中间人劫持的。

所以，**HTTPS 协议本身到目前为止还是没有任何漏洞的，即使你成功进行中间人攻击，本质上是利用了客户端的漏洞（用户点击继续访问或者被恶意导入伪造的根证书），并不是 HTTPS 不够安全。**

为什么抓包工具能截取 HTTPS 数据？

很多抓包工具之所以可以明文看到 HTTPS 数据，工作原理与中间人一致的。

对于 HTTPS 连接来说，中间人要满足以下两点，才能实现真正的明文代理：

1. 中间人，作为客户端与真实服务端建立连接这一步不会有问題，因为服务端不会校验客户端的身份；
2. **中间人，作为服务端与真实客户端建立连接，这里会有客户端信任服务端的问题，也就是服务端必须有对应域名的私钥；**

中间人要拿到私钥只能通过如下方式：

1. 去网站服务端拿到私钥；
2. 去CA处拿域名签发私钥；
3. 自己签发证书，切要被浏览器信任；

不用解释，抓包工具只能使用第三种方式取得中间人的身份。

如何避免被中间人抓取数据？

我们要保证自己电脑的安全，不要被病毒乘虚而入，而且也不要点击任何证书非法的网站，这样 HTTPS 数据就不会被中间人截取到了。

当然，我们还可以通过 **HTTPS 双向认证** 来避免这种问题。

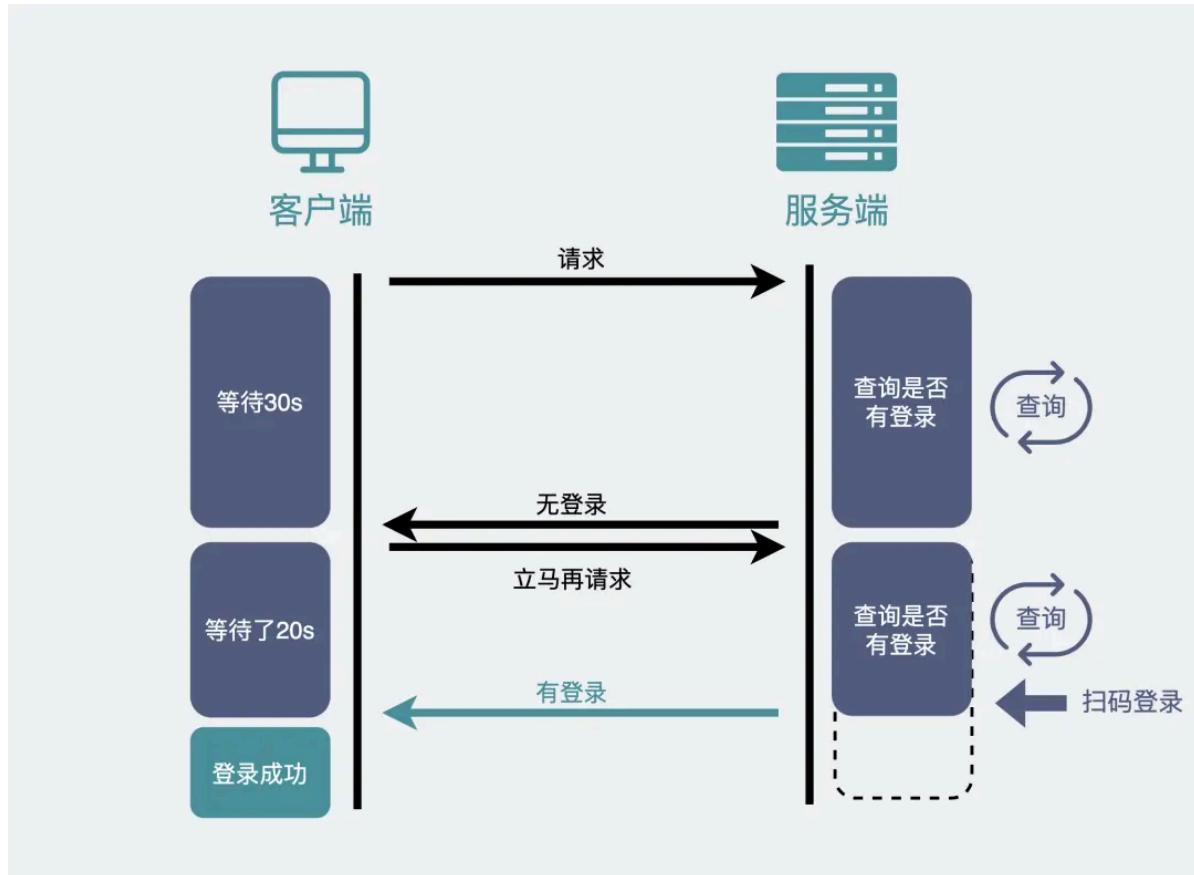
一般我们的 HTTPS 是单向认证，客户端只会验证了服务端的身份，但是服务端并不会验证客户端的身份。



如果用了双向认证方式，不仅客户端会验证服务端的身份，而且服务端也会验证客户端的身份。服务端一旦验证到请求自己的客户端为不可信任的，服务端就拒绝继续通信，客户端如果发现服务端为不可信任的，那么也中止通信。

长轮询

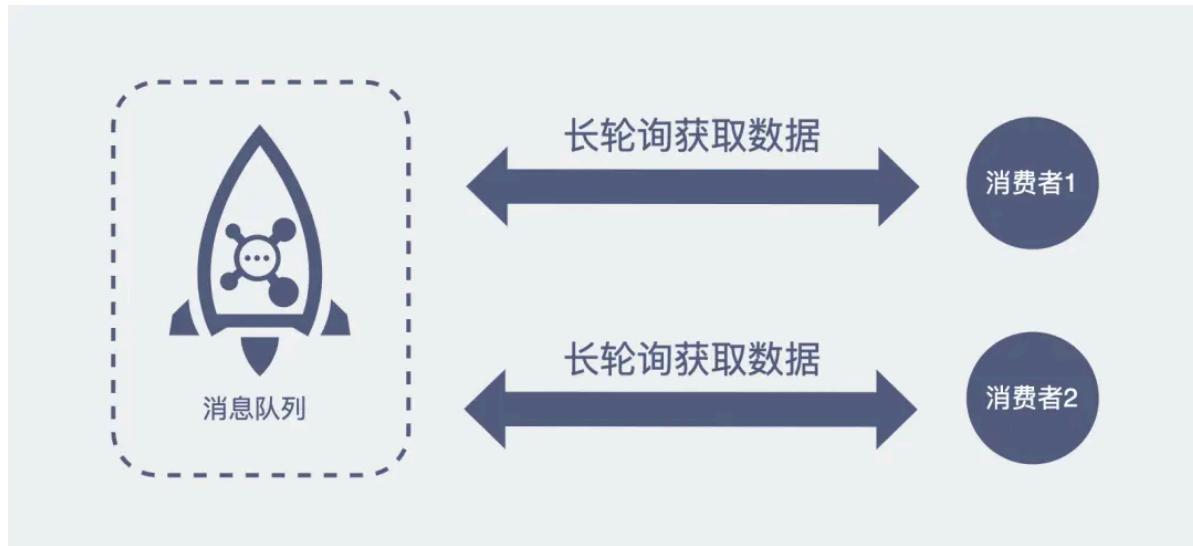
我们知道，HTTP 请求发出后，一般会给服务器留一定的时间做响应，比如 3 秒，规定时间内没返回，就认为是超时。如果我们的 HTTP 请求将超时设置的很大，比如 30 秒，在这 30 秒内只要服务器收到了扫码请求，就立马返回给客户端网页。如果超时，那就立马发起下一次请求。这样就减少了 HTTP 请求的个数，并且由于大部分情况下，用户都会在某个 30 秒的区间内做扫码操作，所以响应也是及时的。



比如，某度云网盘就是这么干的。所以你会发现一扫码，手机上点个确认，电脑端网页就秒跳转，体验很好。



像这种发起一个请求，在较长时间内等待服务器响应的机制，就是所谓的**长轮询机制**。我们常用的消息队列 RocketMQ 中，消费者去取数据时，也用到了这种方式。



像这种，在用户不感知的情况下，服务器将数据推送给浏览器的技术，就是所谓的**服务器推送技术**，它还有个毫不沾边的英文名，**comet** 技术，大家听过就好。

上面提到的两种解决方案（不断轮询（不断的去请求后端，这样体验不好，浪费资源）和长轮询），本质上，其实还是客户端主动去取数据。

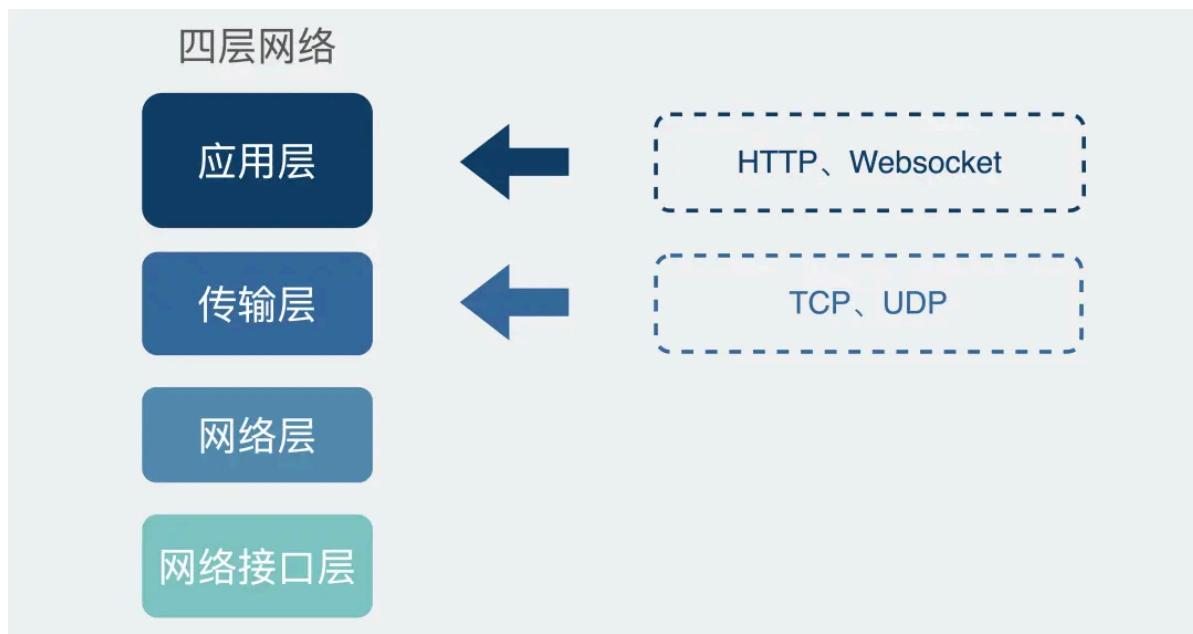
对于像扫码登录这样的**简单场景**还能用用。但如果是网页游戏呢，游戏一般会有大量的数据需要从服务器主动推送到客户端。

WebSocket是什么

我们知道 TCP 连接的两端，**同一时间里，双方都可以主动向对方发送数据**。这就是所谓的**全双工**。而现在使用最广泛的 **HTTP/1.1**，也是基于TCP协议的，**同一时间里，客户端和服务器只能有一方主动发数据**，这就是所谓的**半双工**。

这是由于 **HTTP 协议设计之初，考虑的是看看网页文本的场景，能做到客户端发起请求再由服务器响应，就够了，根本就没考虑网页游戏这种，客户端和服务器之间都要互相主动发大量数据的场景**。

所以，为了更好的支持这样的场景，我们需要另外一个**基于TCP的新协议**。于是新的应用层协议 **WebSocket** 就被设计出来了。虽然名字带了个socket，但其实 **socket 和 WebSocket 之间，就跟雷锋和雷锋塔一样，二者接近毫无关系**。



怎么建立WebSocket连接

我们平时刷网页，一般都是在浏览器上刷的，一会刷刷图文，这时候用的是 **HTTP 协议**，一会打开网页游戏，这时候就得切换成我们新介绍的 **WebSocket 协议**。为了兼容这些使用场景。浏览器在 **TCP 三次握手** 建立连接之后，都统一使用 **HTTP 协议** 先进行一次通信。

- 如果此时是**普通的 HTTP 请求**，那后续双方就还是老样子继续用普通 HTTP 协议进行交互，这点没啥疑问。
- 如果这时候是**想建立 WebSocket 连接**，就会在 HTTP 请求里带上一些**特殊的header 头**，如下：

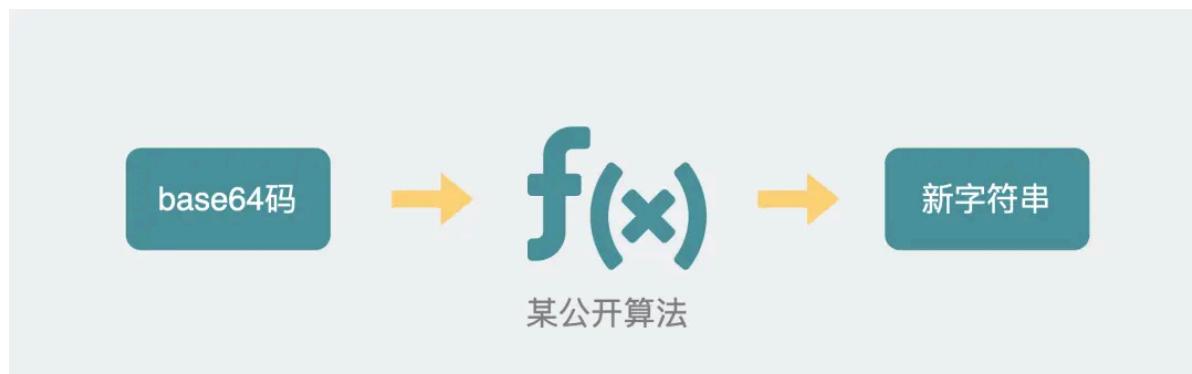
```
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: T2a6wZlAwhgQNqruZ2YUyg==\r\n
```

这些 header 头的意思是，浏览器想**升级协议 (Connection: Upgrade)**，并且**想升级成 WebSocket 协议 (Upgrade: WebSocket)**。同时带上一段随机生成的 **base64 码 (Sec-WebSocket-Key)**，发给服务器。

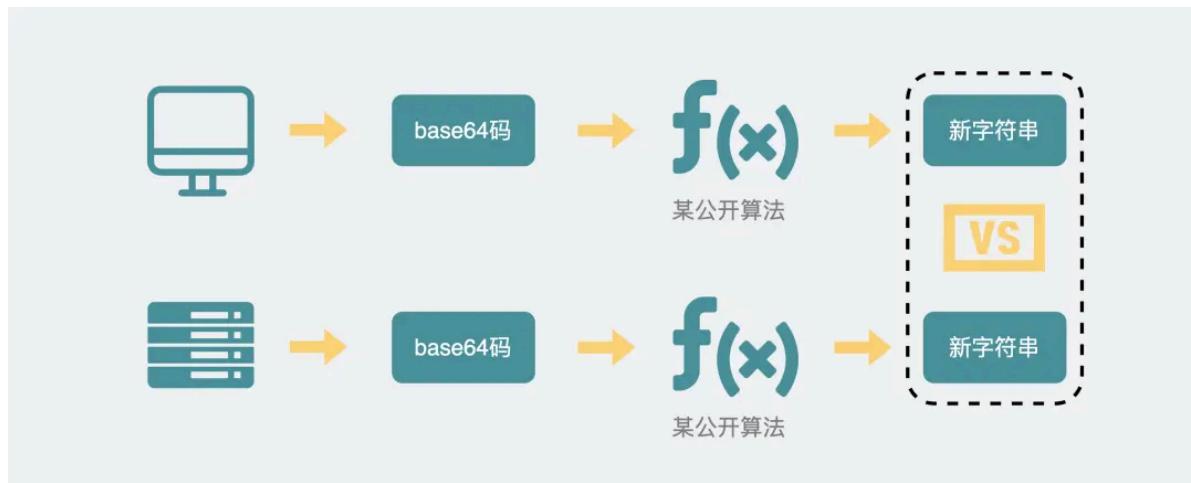
如果服务器正好支持升级成 WebSocket 协议。就会走 WebSocket 握手流程，同时根据客户端生成的 base64 码，用某个**公开的算法**变成另一段字符串，放在 HTTP 响应的 **Sec-WebSocket-Accept** 头里，同时带上 **101状态码**，发回给浏览器。HTTP 的响应如下：

```
HTTP/1.1 101 Switching Protocols\r\n
Sec-WebSocket-Accept: iBjKv/ALIW2DobfoA4dmr3JHBCY=\r\n
Upgrade: websocket\r\n
Connection: Upgrade\r\n
```

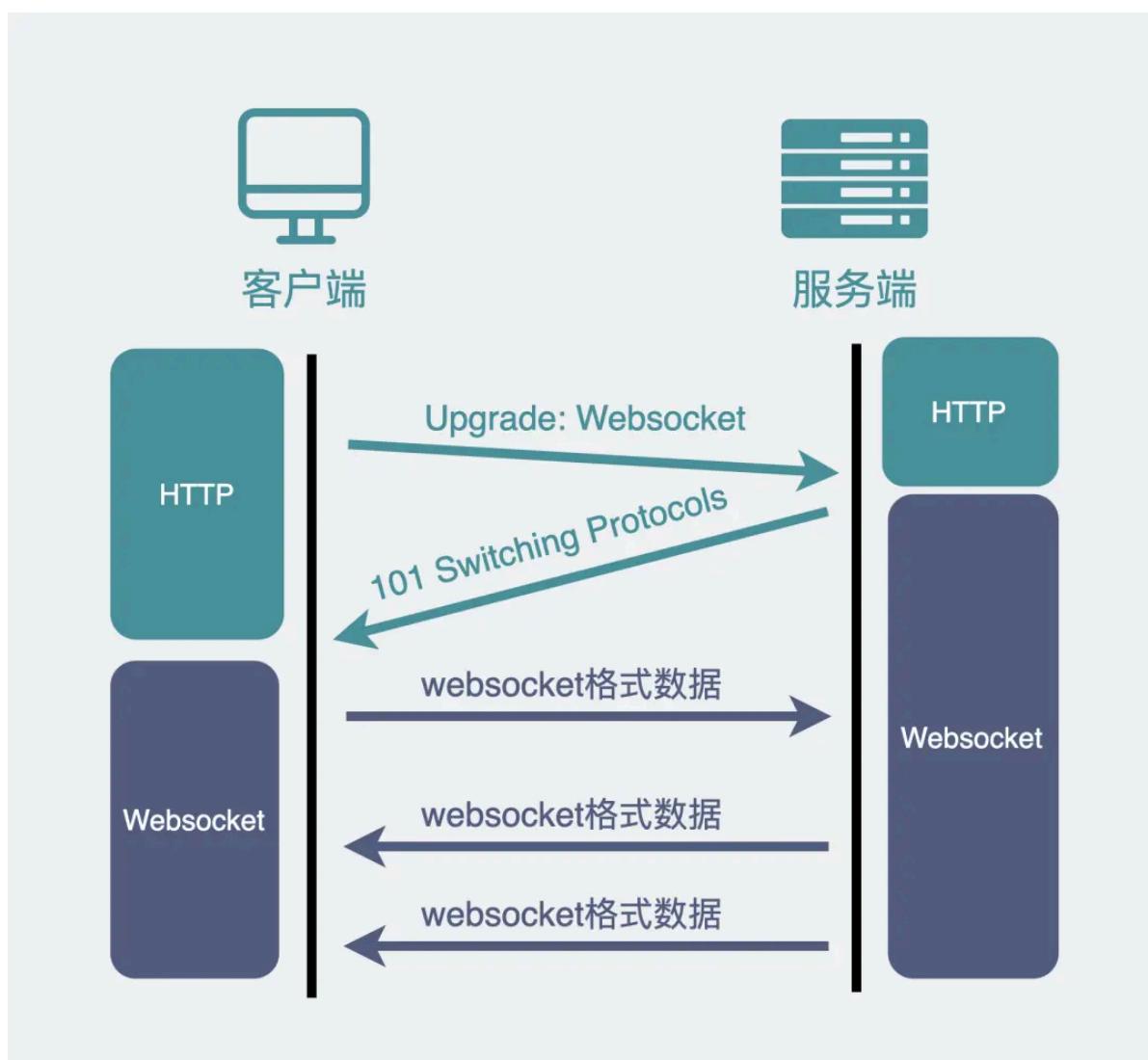
HTTP 状态码=200（正常响应）的情况，大家见得多了。101 确实不常见，它其实是指**协议切换**。



之后，浏览器也用同样的**公开算法**将 **base64码** 转成另一段字符串，如果这段字符串跟服务器传回来的**字符串一致**，那验证通过。



就这样经历了一来一回两次 HTTP 握手，WebSocket 就建立完成了，后续双方就可以使用 websocket 的数据格式进行通信了。你在网上可能会看到一种说法：“WebSocket 是基于HTTP的新协议”，**其实这并不对**，因为 WebSocket 只有在建立连接时才用到了HTTP，**升级完成之后就跟HTTP没有任何关系了。**



WebSocket的消息格式

WebSocket的数据格式也是**数据头（内含payload长度） + payload data**的形式。这是因为 TCP 协议本身就是全双工，但直接使用**纯裸TCP**去传输数据，会有**粘包的“问题”**。为了解决这个问题，上层协议一般会用**消息头+消息体**的格式去重新包装要发的数据。而**消息头里一般含有消息体的长度**，通过这个长度可以去截取真正的消息体。HTTP 协议和大部分 RPC 协议，以及我们今天介绍的WebSocket协议，都是这样设计的。

消息头

消息体

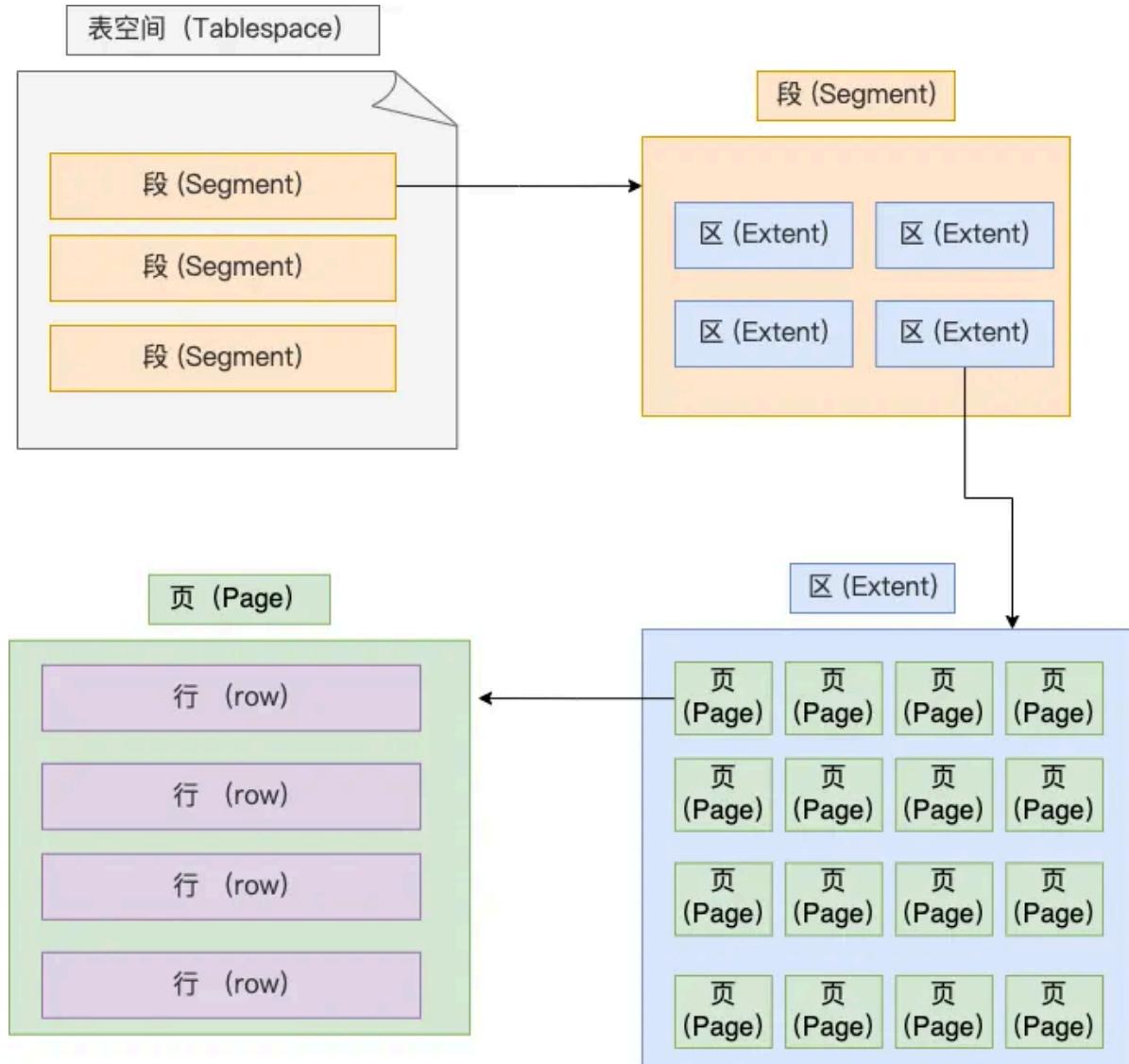
WebSocket的使用场景

WebSocket完美继承了 TCP 协议的**全双工能力**，并且还贴心的提供了解决粘包的方案。它适用于**需要服务器和客户端（浏览器）频繁交互**的大部分场景，比如网页/小程序游戏，网页聊天室，以及一些类似飞书这样的网页协同办公软件。

回到文章开头的问题，在使用 WebSocket 协议的网页游戏里，怪物移动以及攻击玩家的行为是**服务器逻辑**产生的，对玩家产生的伤害等数据，都需要由**服务器主动发送给客户端**，客户端获得数据后展示对应的效果。

InnoDB 是如何存储数据的？

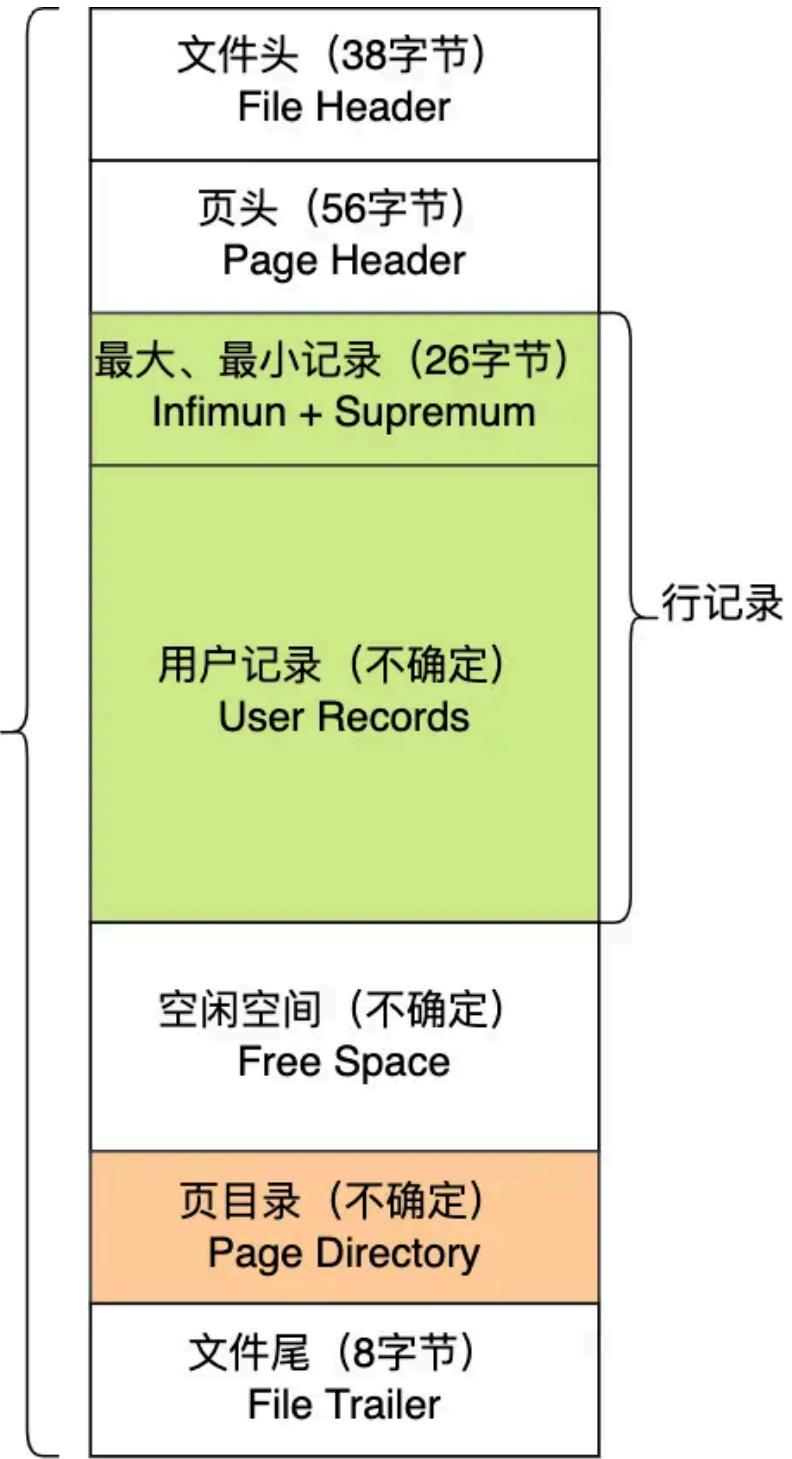
表空间由段 (segment)、区 (extent)、页 (page)、行 (row) 组成，InnoDB 存储引擎的逻辑存储结构大致如下图：



记录是按照行来存储的，但是数据库的读取并不以「行」为单位，否则一次读取（也就是一次 I/O 操作）只能处理一行数据，效率会非常低。因此，**InnoDB 的数据是按「数据页」为单位来读写的**，也就是说，当需要读一条记录的时候，并不是将这个记录本身从磁盘读出来，而是以页为单位，将其整体读入内存。数据库的 I/O 操作的最小单位是页，**InnoDB 数据页的默认大小是 16KB**，意味着数据库每次读写都是以 16KB 为单位的，一次最少从磁盘中读取 16K 的内容到内存中，一次最少把内存中的 16K 内容刷新到磁盘中。

数据页包括七个部分，结构如下图：

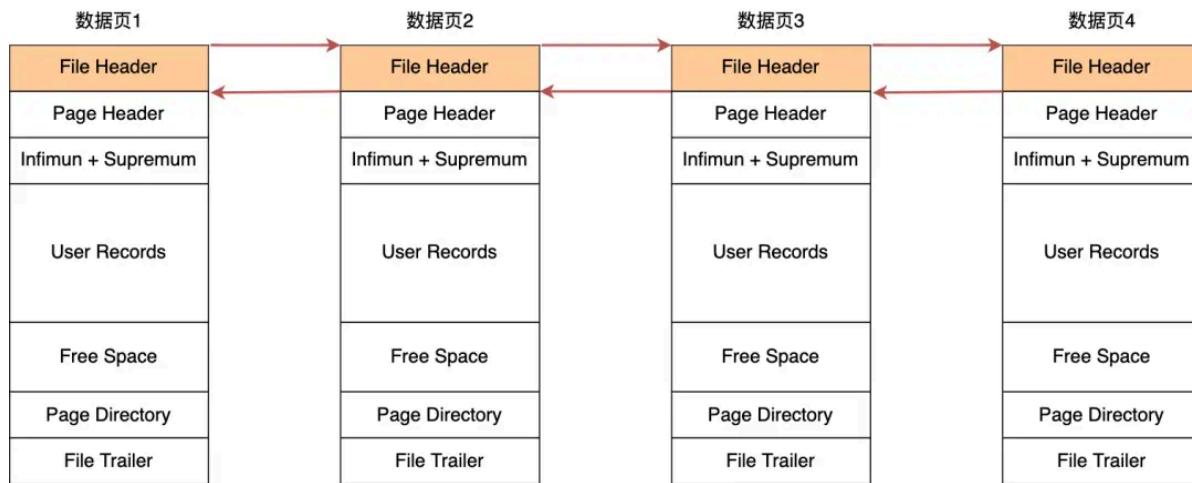
数据页
默认大小 16KB



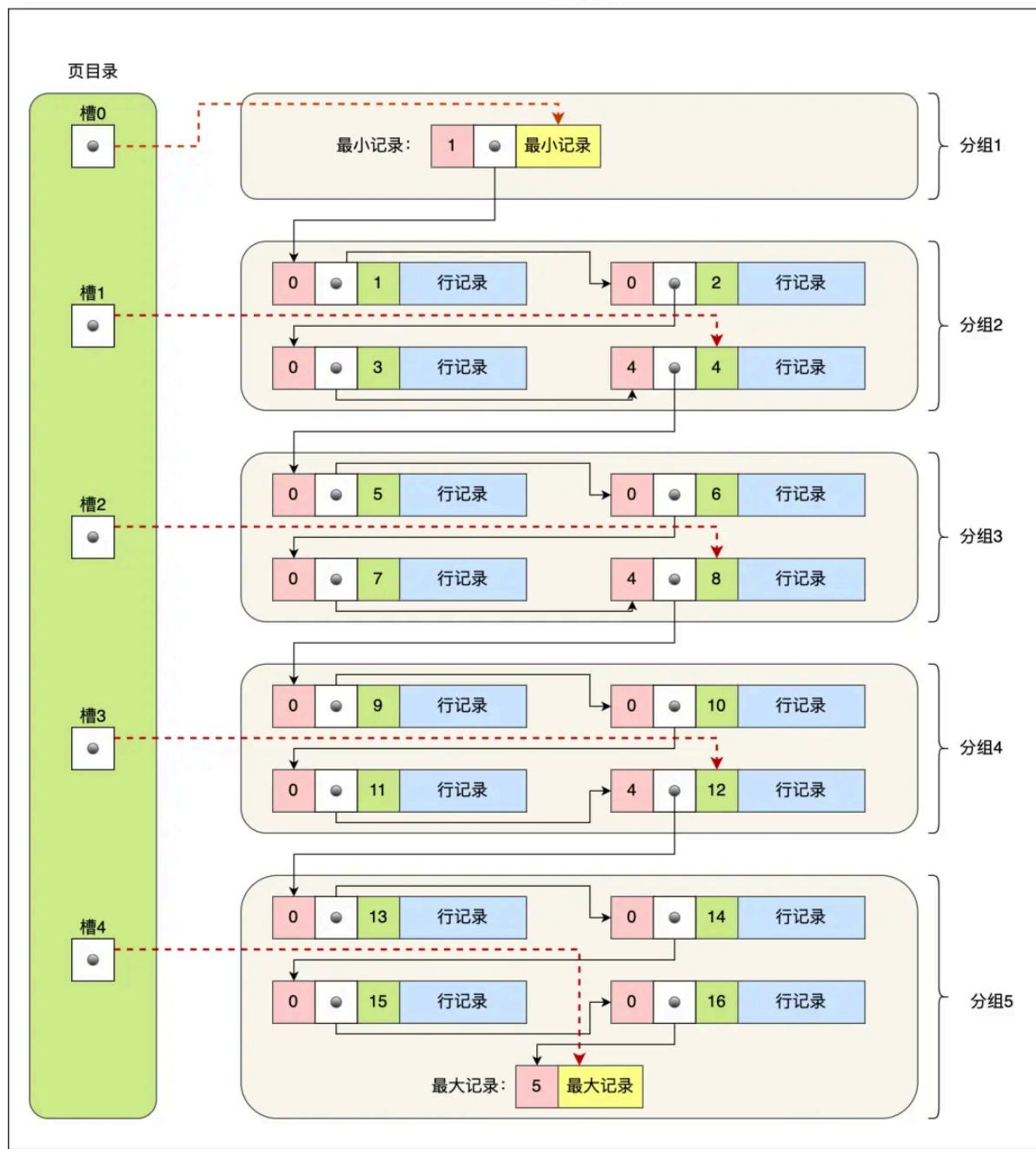
这 7 个部分的作用如下图：

名称	说明
文件头 File Header	文件头，表示页的信息
页头 Page Header	页头，表示页的状态信息
最小和最大记录 Infimum+supremum	两个虚拟的伪记录，分别表示页中的最小记录和最大记录
用户记录 User Records	存储行记录内容
空闲空间 Free Space	页中还没被使用的空间
页目录 Page Directory	存储用户记录的相对位置，对记录起到索引作用
文件尾 File Tailer	校验页是否完整

在 File Header 中有两个指针，分别指向下一个数据页和下一个数据页，连接起来的页相当于一个双向的链表，采用链表的结构是让数据页之间不需要是物理上的连续的，而是逻辑上的连续。如下图所示：



数据页的主要作用是存储记录，也就是数据库的数据。数据页中的记录按照「主键」顺序组成单向链表，单向链表的特点就是插入、删除非常方便，但是检索效率不高，最差的情况下需要遍历链表上的所有节点才能完成检索。因此，数据页中有一个页目录，起到记录的索引作用。那 InnoDB 是如何给记录创建页目录的呢？页目录与记录的关系如下图：



页目录创建的过程如下：

1. 将所有的记录划分成几个组，这些记录包括最小记录和最大记录，但不包括标记为“已删除”的记录；
2. **每个记录组的最后一条记录就是组内最大的那条记录，并且最后一条记录的头信息中会存储该组一共有多少条记录，作为 n_owned 字段（上图中粉红色字段）**
3. 页目录用来存储每组最后一条记录的地址偏移量，这些地址偏移量会按照先后顺序存储起来，每组的地址偏移量也被称之为槽（slot），**每个槽相当于指针指向了不同组的最后一个记录（也就是组里面最大的数据）。**

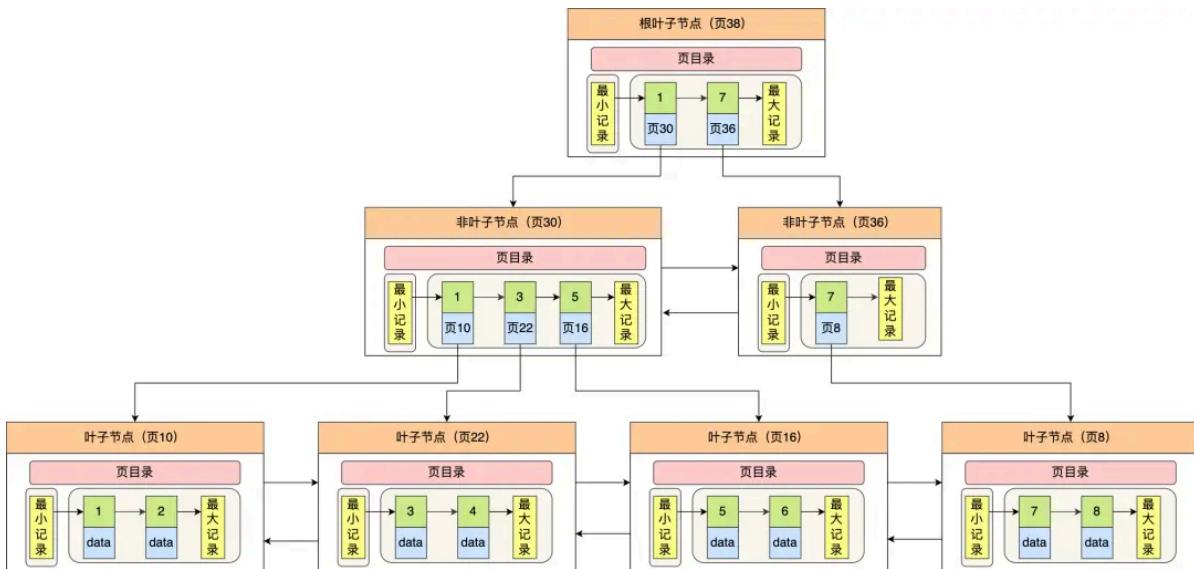
从图可以看到，**页目录就是由多个槽组成的，槽相当于分组记录的索引**。然后，因为记录是按照「主键值」从小到大排序的，所以我们通过槽查找记录时，可以使用**二分法快速定位要查询的记录在哪个槽（哪个记录分组）**，定位到槽后，再遍历槽内的所有记录，找到对应的记录，无需从最小记录开始遍历整个页中的记录链表。

InnoDB 对每个分组中的记录条数都是有规定的，避免查找时间复杂度退化为 $O(n)$ ，槽内的记录就只有几条：

- 第一个分组中的记录只能有 1 条记录；
- 最后一个分组中的记录条数范围只能在 1-8 条之间；
- 剩下的分组中记录条数范围只能在 4-8 条之间。

B+ 树是如何进行查询的？

InnoDB 里的 B+ 树中的每个节点都是一个数据页，结构示意图如下：



通过上图，我们看出 B+ 树的特点：

- 只有叶子节点（最底层的节点）才存放了数据，非叶子节点（其他上层节）仅用来存放目录项作为索引。
- 非叶子节点分为不同层次，通过分层来降低每一层的搜索量；
- 所有节点按照索引键大小排序，构成一个双向链表，便于范围查询；

我们再看看 B+ 树如何实现快速查找主键为 6 的记录，以上图为例子：

- 从根节点开始，通过二分法快速定位到符合页内范围包含查询值的页，因为查询的主键值为 6，在 [1, 7] 范围之间，所以到页 30 中查找更详细的目录项；
- 在非叶子节点（页 30）中，继续通过二分法快速定位到符合页内范围包含查询值的页，主键值大于 5，所以就到叶子节点（页 16）查找记录；
- 接着，在叶子节点（页 16）中，通过槽查找记录时，使用二分法快速定位要查询的记录在哪个槽（哪个记录分组），定位到槽后，再遍历槽内的所有记录，找到主键为 6 的记录。

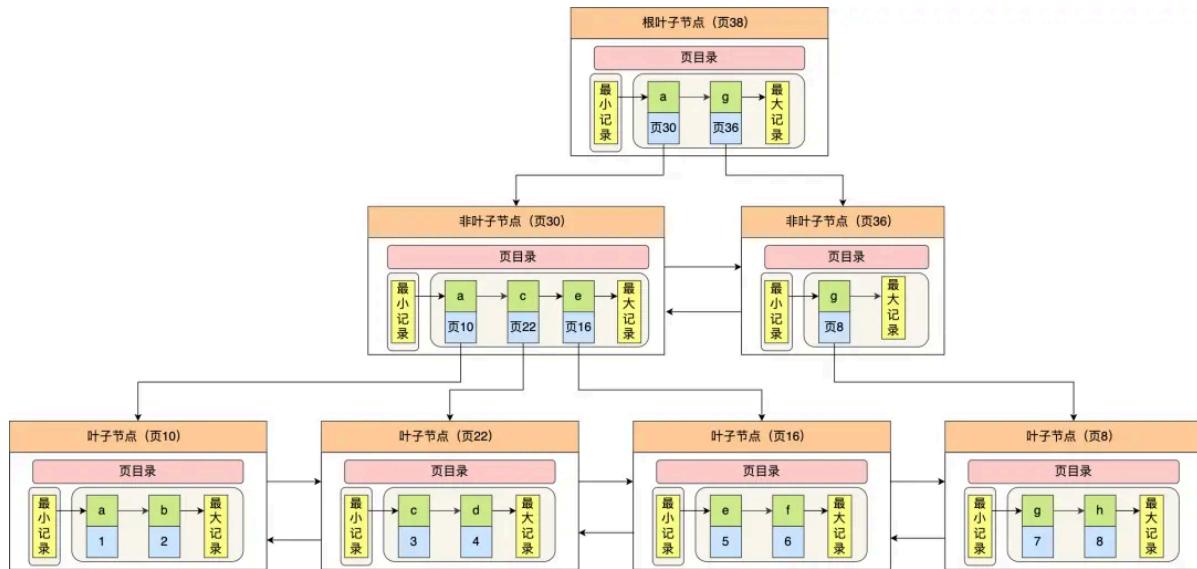
可以看到，在定位记录所在哪一个页时，也是通过二分法快速定位到包含该记录的页。定位到该页后，又会在该页内进行二分法快速定位记录所在的分组（槽号），最后在分组内进行遍历查找。

聚簇索引和二级索引

索引又可以分成聚簇索引和非聚簇索引（二级索引），它们区别就在于叶子节点存放的是什么数据。因为表的数据都是存放在聚簇索引的叶子节点里，所以 InnoDB 存储引擎一定会为表创建一个聚簇索引，且由于数据在物理上只会保存一份，所以聚簇索引只能有一个：

- 聚簇索引的叶子节点存放的是实际数据，所有完整的用户记录都存放在聚簇索引的叶子节点；
- 二级索引的叶子节点存放的是主键值，而不是实际数据。

二级索引的 B+ 树如下图，数据部分为主键值：



怎样的索引的数据结构是好的？

MySQL 的数据是持久化的，意味着数据（索引+记录）是保存到磁盘上的，因为这样即使设备断电了，数据也不会丢失。磁盘读写的最小单位是扇区，扇区的大小只有 512B 大小，操作系统一次会读写多个扇区，所以操作系统的最小读写单位是块（Block）。Linux 中的块大小为 4KB，也就是一次磁盘 I/O 操作会直接读写 8 个扇区。

由于数据库的索引是保存到磁盘上的，因此当我们通过索引查找某行数据的时候，就需要先从磁盘读取索引到内存，再通过索引从磁盘中找到某行数据，然后读入到内存，也就是说查询过程中会发生多次磁盘 I/O，而磁盘 I/O 次数越多，所消耗的时间也就越大。

1. 所以，我们希望索引的数据结构能在尽可能少的磁盘的 I/O 操作中完成查询工作，因为磁盘 I/O 操作越少，所消耗的时间也就越小。
2. 另外，MySQL 是支持范围查找的，所以索引的数据结构不仅要能高效地查询某一个记录，而且也要能高效地执行范围查找。

什么是二分查找树？

二叉查找树的特点是一个节点的左子树的所有节点都小于这个节点，右子树的所有节点都大于这个节点，这样我们在查询数据时，不需要计算中间节点的位置了，只需将查找的数据与节点的数据进行比较。假设，我们查找索引值为 key 的节点：

1. 如果 key 大于根节点，则在右子树中进行查找；
2. 如果 key 小于根节点，则在左子树中进行查找；
3. 如果 key 等于根节点，也就是找到了这个节点，返回根节点即可。

另外，二叉查找树解决了插入新节点的问题，因为二叉查找树是一个跳跃结构，不必连续排列。这样在插入的时候，新节点可以放在任何位置，不会像线性结构那样插入一个元素，所有元素都需要向后排列。但二叉查找树也是有缺点的：

1. **当每次插入的元素都是二叉查找树中最大的元素或最小的元素，二叉查找树就会退化成了一条链表，查找数据的时间复杂度变成了 O(n)**
2. 树是存储在磁盘中的，访问每个节点，都对应一次磁盘 I/O 操作（假设一个节点的大小「小于」操作系统的最小读写单位块的大小），也就是说树的高度就等于每次查询数据时磁盘 I/O 操作的次数，所以树的高度越高，就会影响查询性能。

此外，二叉查找树不能范围查询，因此不适合做为数据库的索引结构。

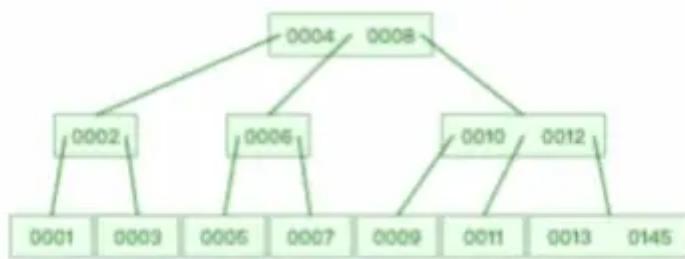
什么是自平衡二叉树？

为了解决二叉查找树会在极端情况下退化成链表的问题，后面就有人提出**平衡二叉查找树（AVL 树）**。主要是在二叉查找树的基础上增加了一些条件约束：**在插入或删除元素时，通过左旋右旋操作，让每个节点的左子树和右子树的高度差不能超过 1**。也就是说节点的左子树和右子树仍然为平衡二叉树，这样查询操作的时间复杂度就会一直维持在 $O(\log n)$ 。

红黑树就是一种自平衡二叉树，旋转次数比AVL树少，可以用在插入次数频繁的情况，C++的map和set就是用红黑树实现的。但不管平衡二叉查找树还是红黑树，都会随着插入的元素增多，而导致树的高度变高，这就意味着磁盘 I/O 操作次数多，会影响整体数据查询的效率。那当树的节点越多的时候，并且树的分叉数 M 越大的时候，M 叉树的高度会远小于二叉树的高度。

什么是 B 树

B 树，它不再限制一个节点就只能有 2 个子节点，而是允许 M 个子节点 ($M > 2$)，从而降低树的高度。B 树的每一个节点最多可以包括 M 个子节点，M 称为 B 树的阶，所以 B 树就是一个多叉树。假设 M = 3，那么就是一棵 3 阶的 B 树，特点就是每个节点最多有 2 个 ($M-1$ 个) 数据和最多有 3 个 (M 个) 子节点，超过这些要求的话，就会分裂节点。我们来看看一棵 3 阶的 B 树的查询过程是怎样的？



假设我们在上图一棵 3 阶的 B 树中要查找的索引值是 9 的记录那么步骤可以分为以下几步：

1. 与根节点的索引(4, 8) 进行比较，9 大于 8，那么往右边的子节点走；
2. 然后该子节点的索引为 (10, 12)，因为 9 小于 10，所以会往该节点的左边子节点走；
3. 走到索引为9的节点，然后我们找到了索引值 9 的节点。

可以看到，一棵 3 阶的 B 树在查询叶子节点中的数据时，由于树的高度是 3，所以在查询过程中会发生 3 次磁盘 I/O 操作。而如果同样的节点数量在平衡二叉树的场景下，树的高度就会很高，意味着磁盘 I/O 操作会更多。所以，B 树在数据查询中比平衡二叉树效率要高。

但是 B 树的每个节点都包含数据（索引+记录），而用户的记录数据的大小很有可能远远超过了索引数据，这就需要花费更多的磁盘 I/O 操作次数来读到「有用的索引数据」。而且，在我们查询位于底层的某个节点（比如 A 记录）过程中，「非 A 记录节点」里的记录数据会从磁盘加载到内存，但是这些记录数据是没用的，我们只是想读取这些节点的索引数据来做比较查询，而「非 A 记录节点」里的记录数据对我们是没用的，这样不仅增多磁盘 I/O 操作次数，也占用内存资源。

另外，如果使用 B 树来做范围查询的话，需要使用中序遍历，这会涉及多个节点的磁盘 I/O 问题，从而导致整体速度下降。

什么是 B+ 树？

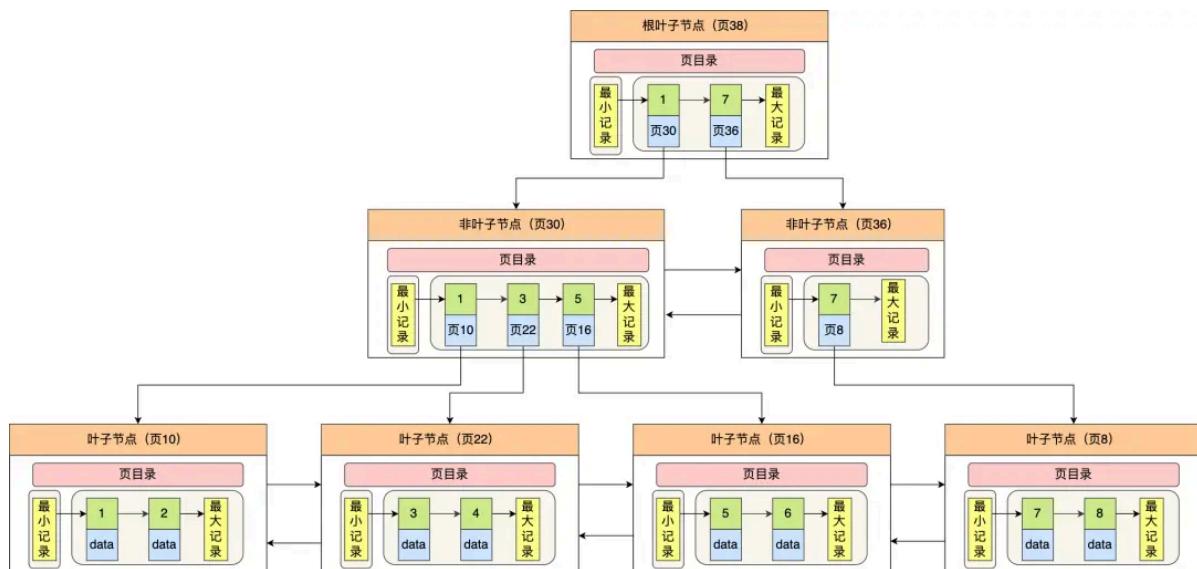
B+ 树就是对 B 树做了一个升级，B+ 树与 B 树差异的点，主要是以下这几点：

- 叶子节点（最底部的节点）才会存放实际数据（索引+记录），非叶子节点只会存放索引，相比存储即存索引又存记录的 B 树，B+树的非叶子节点可以存放更多的索引，因此 B+ 树可以比 B 树更「矮胖」，查询底层节点的磁盘 I/O 次数会更少；
- 所有索引都会在叶子节点出现，叶子节点之间构成一个有序链表（双链表），这种设计对范围查询非常有帮助；
- 非叶子节点的索引也会同时存在在子节点中，并且是在子节点中所有索引的最大（或最小）。
- 非叶子节点中有多少个子节点，就有多少个索引；

B+ 树有大量的冗余节点，这样使得删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不动非叶子节点，这样删除非常快，**插入和删除效率更高**。

MySQL 中的 B+ 树

MySQL 的存储方式根据存储引擎的不同而不同，我们最常用的就是 Innodb 存储引擎，它就是采用了 B+ 树作为索引的数据结构。下图就是 Innodb 里的 B+ 树：



但是 Innodb 使用的 B+ 树有一些特别的点，比如：

- B+ 树的叶子节点之间是用「双向链表」进行连接，这样的好处是既能向右遍历，也能向左遍历。
- B+ 树节点内容是数据页，数据页里存放了用户的记录以及各种信息，每个数据页默认大小是 16 KB。

为什么 MySQL InnoDB 选择 B+tree 作为索引的数据结构?	<p>B+Tree vs B Tree:</p> <ul style="list-style-type: none"> 存储相同数据量级别的情况下，B+Tree 树高比 B Tree 低，磁盘 I/O 次数更少。 B+Tree 叶子节点用双向链表串起来，适合范围查询，B Tree 无法做到这点 <p>B+Tree vs 二叉树:</p> <ul style="list-style-type: none"> 随着数据量的增加，二叉树的树高会越来越高，磁盘 I/O 次数也会更多，B+Tree 在千万级别的数据量下，高度依然维持在 3~4 层左右，也就是说一次数据查询操作只需要做 3~4 次的磁盘 I/O 操作就能查询到目标数据。 <p>B+Tree vs Hash:</p> <ul style="list-style-type: none"> 虽然 Hash 的等值查询效率很高，但是无法做范围查询
什么时候适用索引?	<ul style="list-style-type: none"> 字段有唯一性限制的，比如商品编码； 经常用于 WHERE 查询条件的字段； 经常用于 GROUP BY 和 ORDER BY 的字段；
什么时候不需要创建索引?	<ul style="list-style-type: none"> WHERE 条件，GROUP BY，ORDER BY 里用不到的字段； 字段中存在大量重复数据，不需要创建索引； 表数据太少的时候，不需要创建索引； 经常更新的字段不用创建索引；
什么时候索引会失效?	<ul style="list-style-type: none"> 当我们使用左或者右模糊匹配的时候，也就是 `like %xx` 或者 `like %xx%` 这两种方式都会造成索引失效； 当我们在查询条件中对索引列做了计算、函数、类型转换操作，会导致索引失效 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。 为了更好的利用索引，索引列要设置为 NOT NULL 约束。
有什么优化索引的方法?	<ul style="list-style-type: none"> 前缀索引优化； 覆盖索引优化； 主键索引最好是自增的； 防止索引失效；

什么是索引?

索引的定义就是帮助存储引擎快速获取数据的一种数据结构，形象的说就是索引是数据的目录。

索引的分类

我们可以按照四个角度来分类索引。

- 按「数据结构」分类：B+tree索引、Hash索引、Full-text索引。
- 按「物理存储」分类：聚簇索引（主键索引）、二级索引（辅助索引）。
- 按「字段特性」分类：主键索引、唯一索引、普通索引、前缀索引。
- 按「字段个数」分类：单列索引、联合索引。

按数据结构分类的索引

InnoDB 是在 MySQL 5.5 之后成为默认的 MySQL 存储引擎，B+Tree 索引类型也是 MySQL 存储引擎采用最多的索引类型。创建的主键索引和二级索引默认使用的是 B+Tree 索引。在创建表时，InnoDB 存储引擎会根据不同的场景选择不同的列作为聚簇索引的索引键：

- 如果有主键，**默认会使用主键作为聚簇索引的索引键（key）**；
- 如果没有主键，**就选择第一个不包含 NULL 值的唯一列作为聚簇索引的索引键（key）**；
- 在上面两个都没有的情况下，InnoDB 将**自动生成一个隐式自增 id 列作为聚簇索引的索引键（key）**；

索引类型	InnoDB 引擎	MyISAM 引擎	Memory 引擎
B+Tree 索引	Yes	Yes	Yes
HASH 索引	No (不支持hash索引，但是在内存结构中有一个自适应 hash索引)	No	Yes
Full-Text 索引	Yes (MySQL 5.6 版本后支持)	Yes	No

存储在B+树是怎么索引的

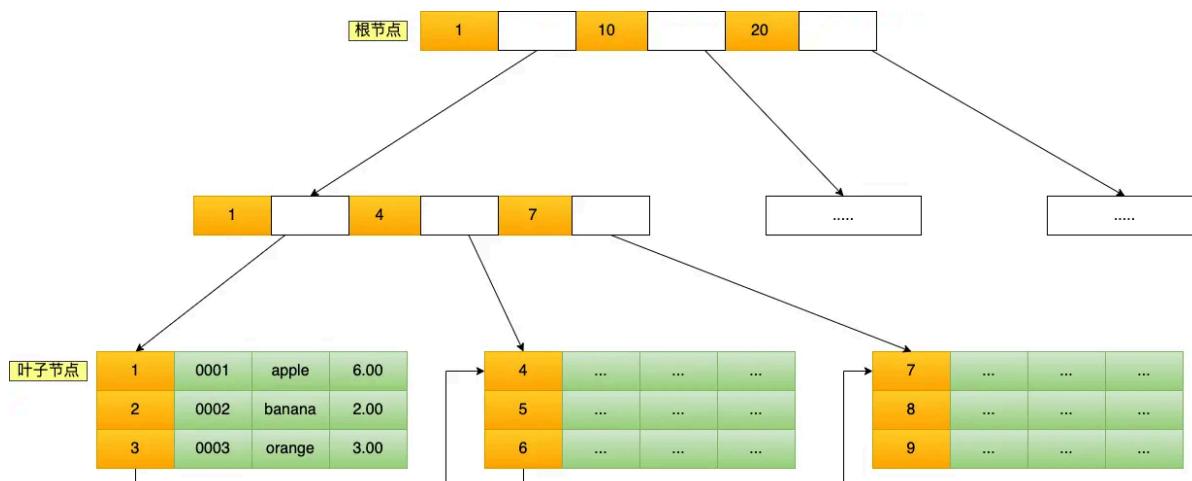
先创建一张商品表，id 为主键，如下：

```
CREATE TABLE `product` (
  `id` int(11) NOT NULL,
  `product_no` varchar(20) DEFAULT NULL,
  `name` varchar(255) DEFAULT NULL,
  `price` decimal(10, 2) DEFAULT NULL,
  PRIMARY KEY (`id`) USING BTREE
) CHARACTER SET = utf8 COLLATE = utf8_general_ci ROW_FORMAT = Dynamic;
```

商品表里，有这些行数据：

id	product_no	name	price
1	0001	apple	6.00
2	0002	banana	2.00
3	0003	orange	3.00
4	0004	iphone13	5000.00
5	0005	ipad8	3500.00
6	0006	macbookpro	10000.00
7	0007	ps5	4000.00
8	0008	grape	10.00
9	0009	watermelon	40.00
10	0010	mango	8.00

这些行数据，存储在 B+Tree 索引时是长什么样子的？B+Tree 是一种多叉树，叶子节点才存放数据，非叶子节点只存放索引，而且每个节点里的数据是按主键顺序存放的。每一层父节点的索引值都会出现在下层子节点的索引值中，因此在叶子节点中，包括了所有的索引值信息，并且每一个叶子节点都有两个指针，分别指向下一个叶子节点和上一个叶子节点，形成一个双向链表。主键索引的 B+Tree 如图所示（图中叶子节点之间我画了单向链表，但是实际上是双向链表，原图我找不到了，修改不了，偷个懒我不重画了，大家脑补成双向链表就行）：



通过主键查询商品数据的过程

比如，我们执行了下面这条查询语句：

```
select * from product where id= 5;
```

这条语句使用了主键索引查询 id 号为 5 的商品。查询过程是这样的，B+Tree 会自顶向下逐层进行查找：

- 将 5 与根节点的索引数据 (1, 10, 20) 比较，5 在 1 和 10 之间，所以根据 B+Tree 的搜索逻辑，找到第二层的索引数据 (1, 4, 7)；
- 在第二层的索引数据 (1, 4, 7) 中进行查找，因为 5 在 4 和 7 之间，所以找到第三层的索引数据 (4, 5, 6)；
- 在叶子节点的索引数据 (4, 5, 6) 中进行查找，然后我们找到了索引值为 5 的行数据。

数据库的索引和数据都是存储在硬盘的，我们可以把读取一个节点当作一次磁盘 I/O 操作。那么上面的整个查询过程一共经历了 3 个节点，也就是进行了 3 次 I/O 操作。**B+Tree** 存储千万级的数据只需要 3-4 层高度就可以满足，这意味着从千万级的表查询目标数据最多需要 3-4 次磁盘 I/O，所以**B+Tree** 相比于 B 树和二叉树来说，最大的优势在于查询效率很高，因为即使在数据量很大的情况，查询一个数据的磁盘 I/O 依然维持在 3-4 次。

通过二级索引查询商品数据的过程

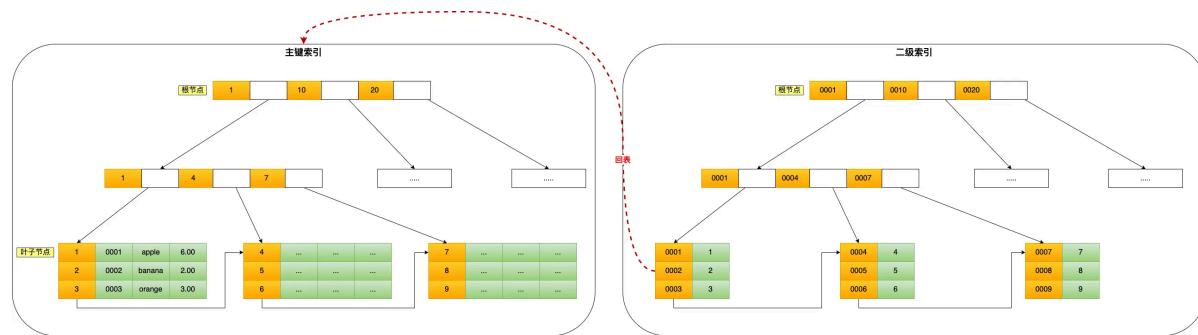
主键索引的 B+Tree 和二级索引的 B+Tree 区别如下：

- 主键索引的 B+Tree 的叶子节点存放的是实际数据，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里；
- 二级索引的 B+Tree 的叶子节点存放的是主键值，而不是实际数据。

我这里将前面的商品表中的 product_no (商品编码) 字段设置为二级索引。如果我用 product_no 二级索引查询商品，如下查询语句：

```
select * from product where product_no = '0002';
```

会先检二级索引中的 B+Tree 的索引值 (商品编码, product_no)，找到对应的叶子节点，然后获取主键值，然后再通过主键索引中的 B+Tree 树查询到对应的叶子节点，然后获取整行数据。这个过程叫「回表」，也就是说要查两个 B+Tree 才能查到数据。如下图 (图中叶子节点之间我画了单向链表，但是实际上是双向链表，原图我找不到了，修改不了，偷个懒我不重画了，大家脑补成双向链表就行)：



不过，当查询的数据是能在二级索引的 B+Tree 的叶子节点里查询到，这时就不用再查主键索引查，比如下面这条查询语句：

```
select id from product where product_no = '0002';
```

这种在二级索引的 B+Tree 就能查询到结果的过程就叫作「覆盖索引」，也就是只需要查一个 B+Tree 就能找到数据。

为什么 MySQL InnoDB 选择 B+tree 作为索引的数据结构？

前面已经讲了 B+Tree 的索引原理，现在就来回答一下 B+Tree 相比于 B 树、二叉树或 Hash 索引结构的优势在哪儿？

1. B+Tree vs B Tree

1. B+Tree 只在叶子节点存储数据，而 B 树的非叶子节点也要存储数据，所以 B+Tree 的单个节点的数据量更小，在相同的磁盘 I/O 次数下，就能查询更多的节点。
2. B+Tree 叶子节点采用的是双链表连接，适合 MySQL 中常见的基于范围的顺序查找，而 B 树无法做到这一点。

2. B+Tree vs 二叉树

对于有 N 个叶子节点的 B+Tree，其搜索复杂度为 $O(\log dN)$ ，其中 d 表示节点允许的最大子节点个数为 d 个。在实际的应用当中，d 值是大于 100 的，这样就保证了，即使数据达到千万级别时，B+Tree 的高度依然维持在 3~4 层左右，也就是说一次数据查询操作只需要做 3~4 次的磁盘 I/O 操作就能查询到目标数据。

而二叉树的每个父节点的儿子节点个数只能是 2 个，意味着其搜索复杂度为 $O(\log N)$ ，这已经比 B+Tree 高出不少，因此二叉树检索到目标数据所经历的磁盘 I/O 次数要更多。

3、B+Tree vs Hash

Hash 在做等值查询的时候效率贼快，搜索复杂度为 $O(1)$ 。

但是 Hash 表不适合做范围查询，它更适合做等值的查询，这也是 B+Tree 索引要比 Hash 表索引有着更广泛的适用场景的原因。

按字段特性分类的索引

从字段特性的角度来看，索引分为主键索引、唯一索引、普通索引、前缀索引。

主键索引

主键索引就是建立在主键字段上的索引，通常在创建表的时候一起创建，一张表最多只有一个主键索引，索引列的值不允许有空值且唯一。在创建表时，创建主键索引的方式如下：

```
CREATE TABLE table_name (
    ...
    PRIMARY KEY (index_column_1) USING BTREE
);
```

唯一索引

唯一索引建立在 UNIQUE 字段上的索引，一张表可以有多个唯一索引，索引列的值必须唯一，但是允许有空值。在创建表时，创建唯一索引的方式如下：

```
CREATE TABLE table_name (
    ...
    UNIQUE KEY(index_column_1, index_column_2, ...)
);
```

建表后，如果要创建唯一索引，可以使用这条命令：

```
CREATE UNIQUE INDEX index_name
ON table_name(index_column_1, index_column_2, ...);
```

普通索引

普通索引就是建立在普通字段上的索引，既不要求字段为主键，也不要要求字段为 UNIQUE。在创建表时，创建普通索引的方式如下：

```
CREATE TABLE table_name (
    ...
    INDEX(index_column_1, index_column_2, ...)
);
```

建表后，如果要创建普通索引，可以使用这面这条命令：

```
CREATE INDEX index_name  
ON table_name(index_column_1, index_column_2, ...);
```

前缀索引

前缀索引是指对字符类型字段的前几个字符建立的索引，而不是在整个字段上建立的索引，前缀索引可以建立在字段类型为 `char`、`varchar`、`binary`、`varbinary` 的列上。使用前缀索引的目的是为了减少索引占用的存储空间，提升查询效率。在创建表时，创建前缀索引的方式如下：

```
CREATE TABLE table_name(  
    column_list,  
    INDEX(column_name(length))  
);
```

建表后，如果要创建前缀索引，可以使用这面这条命令：

```
CREATE INDEX index_name  
ON table_name(column_name(length));
```

按字段个数分类

从字段个数的角度来看，索引分为单列索引、联合索引（复合索引）。

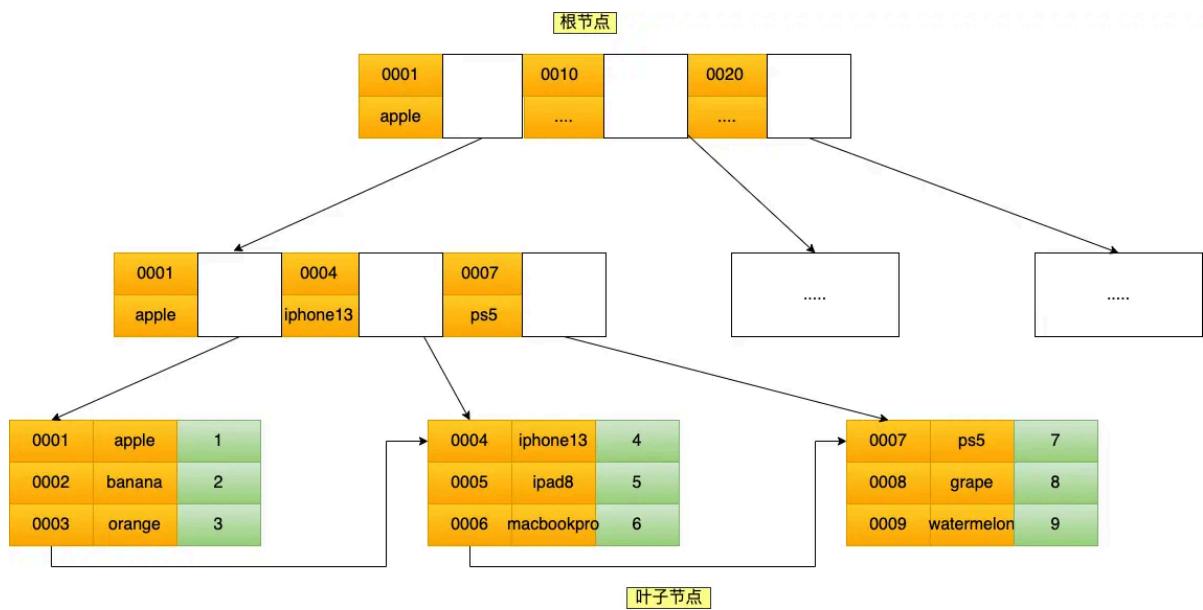
- 建立在单列上的索引称为单列索引，比如主键索引；
- 建立在多列上的索引称为联合索引；

联合索引

通过将多个字段组合成一个索引，该索引就被称为联合索引。比如，将商品表中的 `product_no` 和 `name` 字段组合成联合索引 `(product_no, name)`，创建联合索引的方式如下：

```
CREATE INDEX index_product_no_name ON product(product_no, name);
```

联合索引 `(product_no, name)` 的 B+Tree 示意图如下（图中叶子节点之间我画了单向链表，但是实际上是双向链表，原图我找不到了，修改不了，偷个懒我不重画了，大家脑补成双向链表就行）。



可以看到，联合索引的非叶子节点用两个字段的值作为 B+Tree 的 key 值。当在联合索引查询数据时，先按 `product_no` 字段比较，在 `product_no` 相同的情况下再按 `name` 字段比较。也就是说，联合索引查询的 B+Tree 是先按 `product_no` 进行排序，然后再 `product_no` 相同的情况下再按 `name` 字段排序。

联合索引的最左匹配原则

使用联合索引时，存在**最左匹配原则**，也就是按照最左优先的方式进行索引的匹配。在使用联合索引进行查询的时候，如果不遵循「最左匹配原则」，联合索引会失效，这样就无法利用到索引快速查询的特性了。

比如，如果创建了一个 `(a, b, c)` 联合索引，如果查询条件是以下这几种，就可以匹配上联合索引：

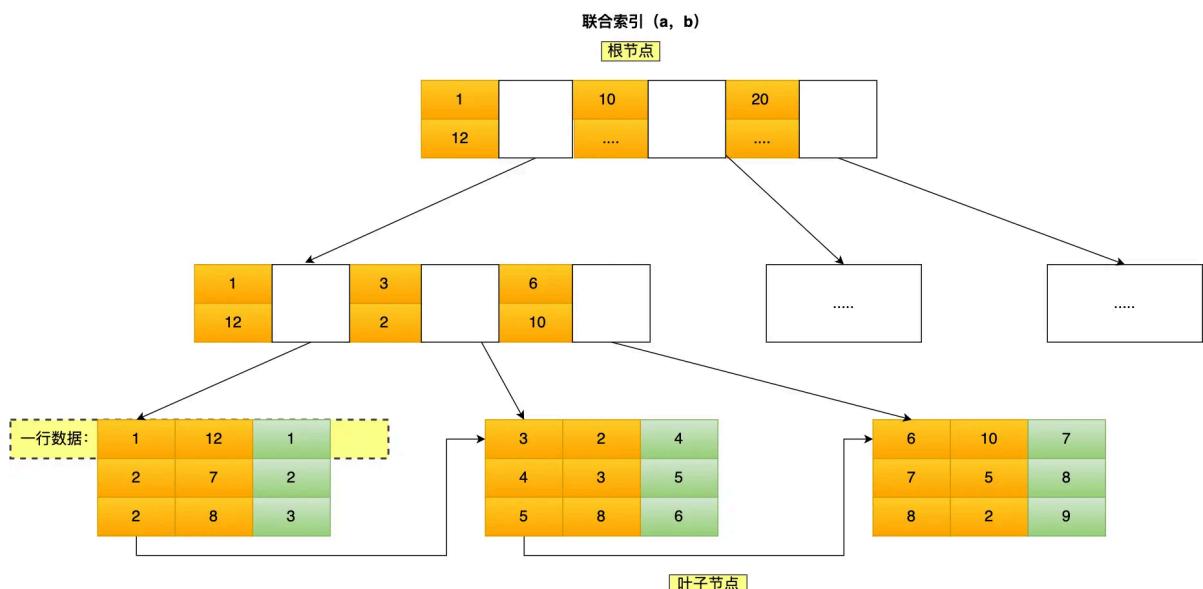
- `where a=1;`
- `where a=1 and b=2 and c=3;`
- `where a=1 and b=2;`

需要注意的是，因为**有查询优化器**，所以 `a` 字段在 `where` 子句的顺序并不重要。但是，如果查询条件是以下这几种，因为不符合最左匹配原则，所以就无法匹配上联合索引，联合索引就会失效：

- `where b=2;`
- `where c=3;`
- `where b=2 and c=3;`

上面这些查询条件之所以会失效，是因为 `(a, b, c)` 联合索引，是先按 `a` 排序，在 `a` 相同的情况下再按 `b` 排序，在 `b` 相同的情况下再按 `c` 排序。所以，**`b` 和 `c` 是全局无序，局部相对有序的**，这样在没有遵循最左匹配原则的情况下，是无法利用到索引的。

我这里举联合索引 `(a, b)` 的例子，该联合索引的 B+ Tree 如下（图中叶子节点之间我画了单向链表，但是实际上是双向链表，原图我找不到了，修改不了，偷个懒我不重画了，大家脑补成双向链表就行）。



可以看到，a 是全局有序的 (1, 2, 2, 3, 4, 5, 6, 7, 8)，而 b 是全局是无序的 (12, 7, 8, 2, 3, 8, 10, 5, 2)。因此，直接执行 `where b = 2` 这种查询条件没有办法利用联合索引的，**利用索引的前提是索引里的 key 是有序的**。只有在 a 相同的情况下，b 才是有序的，比如 a 等于 2 的时候，b 的值为 (7, 8)，这时就是有序的，这个有序状态是局部的，因此，执行 `where a = 2 and b = 7` 是 a 和 b 字段能用到联合索引的，也就是联合索引生效了。

联合索引范围查询

联合索引有一些特殊情况，并不是查询过程使用了联合索引查询，就代表联合索引中的所有字段都用到了联合索引进行索引查询，也就是可能存在部分字段用到联合索引的 B+Tree，部分字段没有用到联合索引的 B+Tree 的情况。这种特殊情况就发生在范围查询。联合索引的最左匹配原则会一直向右匹配直到遇到「范围查询」就会停止匹配。也就是范围查询的字段可以用到联合索引，但是在范围查询字段的后面的字段无法用到联合索引。

范围查询有很多种，那到底是哪些范围查询会导致联合索引的最左匹配原则会停止匹配呢？

`select * from t_table where a > 1 and b = 2 , 联合索引 (a, b)`
哪一个字段用到了联合索引的 B+Tree?

由于联合索引（二级索引）是先按照 a 字段的值排序的，所以符合 `a > 1` 条件的二级索引记录肯定是相邻，于是在进行索引扫描的时候，可以定位到符合 `a > 1` 条件的第一条记录，然后沿着记录所在的链表向后扫描，直到某条记录不符合 `a > 1` 条件位置。所以 a 字段可以在联合索引的 B+Tree 中进行索引查询。

但是在符合 `a > 1` 条件的二级索引记录的范围里，b 字段的值是无序的。Q1 这条查询语句只有 a 字段用到了联合索引进行索引查询，而 b 字段并没有使用到联合索引。我们也可以在执行计划中的 `key_len` 知道这一点，在使用联合索引进行查询的时候，通过 `key_len` 我们可以知道优化器具体使用了多少个字段的搜索条件来形成扫描区间的边界条件。

举例个例子，a 和 b 都是 int 类型且不为 NULL 的字段，那么 Q1 这条查询语句执行计划如下，可以看到 `key_len` 为 4 字节（如果字段允许为 NULL，就在字段类型占用的字节数上加 1，也就是 5 字节），说明只有 a 字段用到了联合索引进行索引查询，而且可以看到，即使 b 字段没用到联合索引，key 为 `idx_a_b`，说明 Q1 查询语句使用了 `idx_a_b` 联合索引。

1 EXPLAIN select * from t_table where a > 1 and b = 2 |

• 只有 a 字段用到了联合索引

Result 1											
Message	Result 1	Profile	Status								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_table	(NULL)	range	idx_a_b	idx_a_b	4	(NULL)	8	11.11	Using where; Using index

```
select * from t_table where a >= 1 and b = 2, 联合索引 (a, b)  
哪一个字段用到了联合索引的 B+Tree?
```

由于联合索引（二级索引）是先按照 a 字段的值排序的，所以符合 $a \geq 1$ 条件的二级索引记录肯定是相邻的，于是在进行索引扫描的时候，可以定位到符合 $a \geq 1$ 条件的第一条记录，然后沿着记录所在的链表向后扫描，直到某条记录不符合 $a \geq 1$ 条件位置。所以 a 字段可以在联合索引的 B+Tree 中进行索引查询。虽然在符合 $a \geq 1$ 条件的二级索引记录的范围里，b 字段的值是「无序」的，但是对于符合 $a = 1$ 的二级索引记录的范围里，b 字段的值是「有序」的（因为对于联合索引，是先按照 a 字段的值排序，然后在 a 字段的值相同的情况下，再按照 b 字段的值进行排序）。

于是，在确定需要扫描的二级索引的范围时，当二级索引记录的 a 字段值为 1 时，可以通过 b = 2 条件减少需要扫描的二级索引记录范围（b 字段可以利用联合索引进行索引查询的意思）。也就是说，从符合 $a = 1 \text{ and } b = 2$ 条件的第一条记录开始扫描，而不需要从第一个 a 字段值为 1 的记录开始扫描。

所以，Q2 这条查询语句 a 和 b 字段都用到了联合索引进行索引查询。

我们也可以在执行计划中的 key_len 知道这一点。执行计划如下，可以看到 key_len 为 8 字节，说明优化器使用了 2 个字段的查询条件来形成扫描区间的边界条件，也就是 a 和 b 字段都用到了联合索引进行索引查询。

1 EXPLAIN select * from t_table where a >= 1 and b = 2

The execution plan shows a simple query (id 1) with a joint index scan (select_type SIMPLE) on table t_table. The key used is idx_a_b, which has a key_len of 8 bytes. The result set contains 9 rows, with a filtered count of 11.11 and the note "Using where; Using index". A callout box highlights the 'a and b fields both used a joint index'.

Result 1									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	t_table	(NULL)	index	idx_a_b	idx_a_b	8	(NULL)	9

11.11 Using where; Using index

```
SELECT * FROM t_table WHERE a BETWEEN 2 AND 8 AND b = 2, 联合索引 (a, b) 哪一个字段用到了联合索引的 B+Tree?
```

在 MySQL 中，BETWEEN 包含了 value1 和 value2 边界值，类似于 \geq and \leq 。而有的数据库则不包含 value1 和 value2 边界值（类似于 $>$ and $<$ ）。由于 MySQL 的 BETWEEN 包含 value1 和 value2 边界值，所以类似于 Q2 查询语句，因此 Q3 这条查询语句 a 和 b 字段都用到了联合索引进行索引查询。

我们也可以在执行计划中的 key_len 知道这一点。执行计划如下，可以看到 key_len 为 8 字节，说明优化器使用了 2 个字段的查询条件来形成扫描区间的边界条件，也就是 a 和 b 字段都用到了联合索引进行索引查询。

1 EXPLAIN select * from t_table where a BETWEEN 2 and 8 and b = 2;

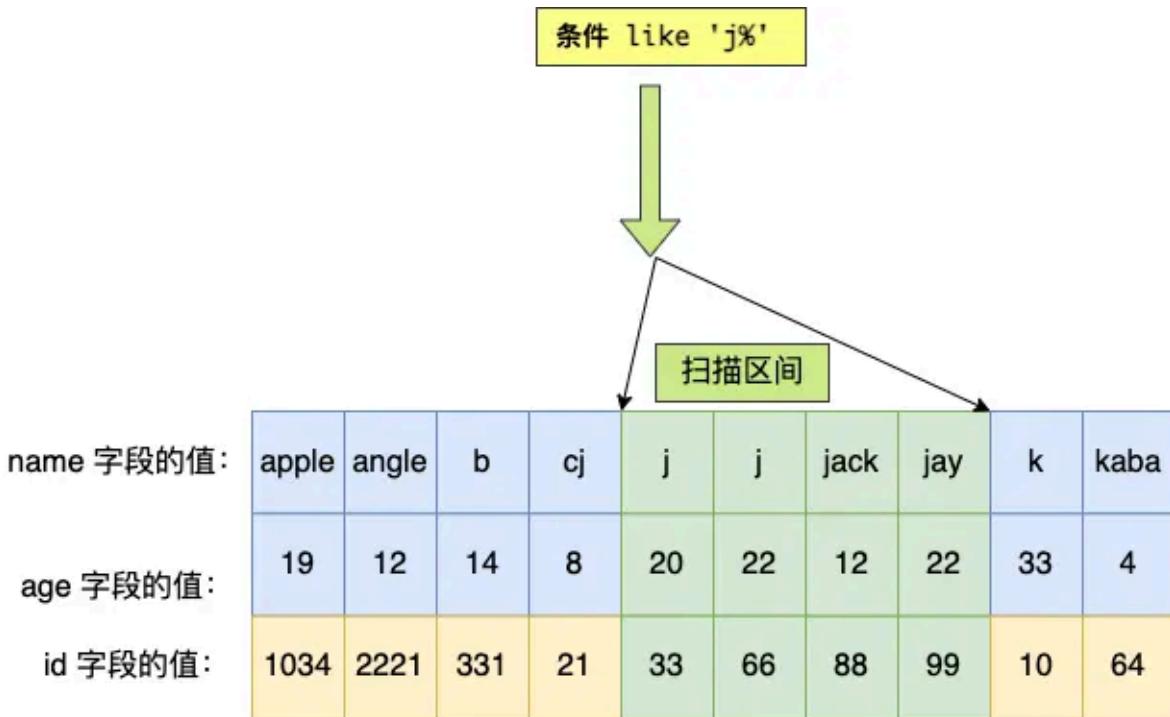
The execution plan shows a simple query (id 1) with a joint index range scan (select_type SIMPLE) on table t_table. The key used is idx_a_b, which has a key_len of 8 bytes. The result set contains 8 rows, with a filtered count of 11.11 and the note "Using where; Using index". A callout box highlights the 'a field and b field both used a joint index'.

Result 1									
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	t_table	(NULL)	range	idx_a_b	idx_a_b	8	(NULL)	8

11.11 Using where; Using index

```
SELECT * FROM t_user WHERE name like 'j%' and age = 22, 联合索引 (name, age) 哪一个字段用到了联合索引的 B+Tree?
```

由于联合索引（二级索引）是先按照 name 字段的值排序的，所以前缀为 'j' 的 name 字段的二级索引记录都是相邻的，于是在进行索引扫描的时候，可以定位到符合前缀为 'j' 的 name 字段的第一条记录，然后沿着记录所在的链表向后扫描，直到某条记录的 name 前缀不为 'j' 为止。所以 a 字段可以在联合索引的 B+Tree 中进行索引查询，形成的扫描区间是['j', 'k')。注意，j 是闭区间。如下图：



联合索引 index_name_age 示意图
(先按照 name 排序, name 相同的情况下, 再按照 age 排序)

虽然在符合前缀为 'j' 的 name 字段的二级索引记录的范围里, age 字段的值是「无序」的, 但是对于符合 `name = j` 的二级索引记录的范围里, age 字段的值是「有序」的。所以, Q4 这条查询语句 a 和 b 字段都用到了联合索引进行索引查询。

我们也可以在执行计划中的 `key_len` 知道这一点。本次例子中:

- name 字段的类型是 `varchar(30)` 且不为 `NULL`, 数据库表使用了 `utf8mb4` 字符集, 一个字符集为 `utf8mb4` 的字符是 4 个字节, 因此 name 字段的实际数据最多占用的存储空间长度是 120 字节 (30×4), 然后因为 name 是变长类型的字段, 需要再加 2 字节 (用于存储该字段实际数据的长度值), 也就是 name 的 `key_len` 为 122。
- age 字段的类型是 `int` 且不为 `NULL`, `key_len` 为 4。

可能有的同学对于「因为 name 是变长类型的字段, 需要再加 2 字节」这句话有疑问。之前这篇文章 ([opens new window](#)) 说「如果变长字段允许存储的最大字节数小于等于 255 字节, 就会用 1 字节表示变长字段的长度」, 而这里为什么是 2 字节? `key_len` 的显示比较特殊, 行格式是由 innodb 存储引擎实现的, 而执行计划是在 server 层生成的, 所以它不会去问 innodb 存储引擎可变字段的长度占用多少字节, 而是不管三七二十一都使用 2 字节表示可变字段的长度。毕竟 `key_len` 的目的只是为了告诉你索引查询中用了哪些索引字段, 而不是为了准确告诉这个字段占用多少字节空间。

Q4 查询语句的执行计划如下, 可以看到 `key_len` 为 126 字节, name 的 `key_len` 为 122, age 的 `key_len` 为 4, 说明优化器使用了 2 个字段的查询条件来形成扫描区间的边界条件, 也就是 name 和 age 字段都用到了联合索引进行索引查询。

1	EXPLAIN	select * from t_user where name like "j%" and age = 22;	Message	Result 1	Profile	Status			
				key	key_len	ref	rows	filtered	Extra
	1 SIMPLE	t_user	(NULL)	range	index_name_age	index_name_age	126	(NULL)	1 10.00 Using index condition

综上所示, 联合索引的最左匹配原则, 在遇到范围查询 (如 `>`、`<`) 的时候, 就会停止匹配, 也就是范围查询的字段可以用到联合索引, 但是在范围查询字段的后面的字段无法用到联合索引。注意, 对于 `>=`、`<=`、`BETWEEN`、`like 前缀匹配` 的范围查询, 并不会停止匹配, 前面我也用了四个例子说明了。

索引下推

现在我们知道，对于联合索引 (a, b)，在执行 `select * from table where a > 1 and b = 2` 语句的时候，只有 a 字段能用到索引，那在联合索引的 B+Tree 找到第一个满足条件的主键值 (ID 为 2) 后，还需要判断其他条件是否满足 (看 b 是否等于 2)，那是在联合索引里判断？还是回主键索引去判断呢？

- 在 MySQL 5.6 之前，只能从 ID2 (主键值) 开始一个个回表，到「主键索引」上找出数据行，再对比 b 字段值。
- 而 MySQL 5.6 引入的索引下推优化 (index condition pushdown)，可以在联合索引遍历过程中，对联合索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

当你的查询语句的执行计划里，出现了 Extra 为 `Using index condition`，那么说明使用了索引下推的优化。

索引区分度

另外，建立联合索引时的字段顺序，对索引效率也有很大影响。越靠前的字段被用于索引过滤的概率越高，实际开发工作中建立联合索引时，要把区分度大的字段排在前面，这样区分度大的字段越有可能被更多的 SQL 使用到。区分度就是某个字段 column 不同值的个数「除以」表的总行数，计算公式如下：

$$\text{区分度} = \frac{\text{distinct}(column)}{\text{count}(*)}$$

比如，性别的区分度就很小，不适合建立索引或不适合排在联合索引列的靠前的位置，而 UUID 这类字段就比较适合做索引或排在联合索引列的靠前的位置。因为如果索引的区分度很小，假设字段的值分布均匀，那么无论搜索哪个值都可能得到一半的数据。在这些情况下，还不如不要索引，因为 MySQL 还有一个查询优化器，查询优化器发现某个值出现在表的数据行中的百分比（惯用的百分比界线是“30%”）很高的时候，它一般会忽略索引，进行全表扫描。

联合索引进行排序

这里出一个题目，针对下面这条 SQL，你怎么通过索引来提高查询效率呢？

```
select * from order where status = 1 order by create_time asc
```

有的同学会认为，单独给 status 建立一个索引就可以了。但是更好的方式给 status 和 create_time 列建立一个联合索引，因为这样可以避免 MySQL 数据库发生文件排序。因为在查询时，如果只用到 status 的索引，但是这条语句还要对 create_time 排序，这时就要用文件排序 filesort，也就是在 SQL 执行计划中，Extra 列会出现 `Using filesort`。所以，要利用索引的有序性，在 status 和 create_time 列建立联合索引，这样根据 status 筛选后的数据就是按照 create_time 排好序的，避免在文件排序，提高了查询效率。

创建联合索引的顺序

1. 索引区分度高的在前面
2. 最左匹配原则，考虑where子句和order by、group by子句的顺序

索引的缺点

- 需要占用物理空间，数量越大，占用空间越大；
- 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大；
- 会降低表的增删改的效率，因为每次增删改索引，B+树为了维护索引有序性，都需要进行动态维护。

什么时候适用索引？

- 字段有唯一性限制的，比如商品编码；
- 经常用于 WHERE 查询条件的字段，这样能够提高整个表的查询速度，如果查询条件不是一个字段，可以建立联合索引。
- 经常用于 GROUP BY 和 ORDER BY 的字段，这样在查询的时候就不需要再去做一次排序了，因为我们都已经知道了建立索引之后在 B+Tree 中的记录都是排序好的。

什么时候不需要创建索引？

- WHERE 条件，GROUP BY，ORDER BY 里用不到的字段，索引的价值是快速定位，如果起不到定位的字段通常是不需要创建索引的，因为索引是会占用物理空间的。
- 字段中存在大量重复数据，不需要创建索引
- 表数据太少的时候，不需要创建索引；
- 经常更新的字段不用创建索引，比如不要对电商项目的用户余额建立索引，因为索引字段频繁修改，由于要维护 B+Tree 的有序性，那么就需要频繁的重建索引，这个过程是会影响数据库性能的。

有什么优化索引的方法？

- 前缀索引优化：使用前缀索引是为了减小索引字段大小，可以增加一个索引页中存储的索引值，有效提高索引的查询速度。在一些大字符串的字段作为索引时，使用前缀索引可以帮助我们减小索引项的大小。不过，前缀索引有一定的局限性，例如：
 - order by 就无法使用前缀索引；
 - 无法把前缀索引用作覆盖索引；
- 覆盖索引优化：建立联合索引，避免回表，减少大量的I/O操作
- 主键索引最好是自增的：
 - 如果我们使用自增主键，那么每次插入的新数据就会按顺序添加到当前索引节点的位置，不需要移动已有的数据，当页面写满，就会自动开辟一个新页面。因为每次插入一条新记录，都是追加操作，不需要重新移动数据，因此这种插入数据的方法效率非常高。
 - 如果我们使用非自增主键，由于每次插入主键的索引值都是随机的，因此每次插入新的数据时，就可能会插入到现有数据页中间的某个位置，这将不得不移动其它数据来满足新数据的插入，甚至需要从一个页面复制数据到另外一个页面，我们通常将这种情况称为页分裂。页分裂还可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率。

- 主键字段的长度不要太大，因为主键字段长度越小，意味着二级索引的叶子节点越小（二级索引的叶子节点存放的数据是主键值），这样二级索引占用的空间也就越小。
- 索引最好设置为 NOT NULL：
 - 索引列存在 NULL 就会导致优化器在做索引选择的时候更加复杂
 - NULL 值是一个没意义的值，但是它会占用物理空间，所以会带来的存储空间的问题
- 防止索引失效：避免写出索引失效的查询语句
 - 当我们使用左或者左右模糊匹配的时候，也就是 `like %xx` 或者 `like %xx%` 这两种方式都会造成索引失效；
 - 当我们在查询条件中对索引列做了计算、函数、类型转换操作，这些情况下都会造成索引失效；
 - 联合索引要能正确使用需要遵循最左匹配原则，也就是按照最左优先的方式进行索引的匹配，否则就会导致索引失效。
 - 在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。这是因为数据库优化器在处理 OR 操作符时需要确保查询的每个部分都能够高效执行。如果 OR 的一侧不能使用索引，那么整个查询可能会退化为全表扫描。

explain

`EXPLAIN` 语句用于显示查询的执行计划。这有助于了解查询的执行方式，并找出可能的性能瓶颈。

`EXPLAIN` 会提供关于查询执行的详细信息，包括使用的索引、扫描的行数、连接类型等。对于执行计划，参数有：

- `possible_keys` 字段表示可能用到的索引；
- `key` 字段表示实际用的索引，如果这一项为 NULL，说明没有使用索引；
- `key_len` 表示索引的长度；
- `rows` 表示扫描的数据行数。
- `type` 表示数据扫描类型，我们需要重点看这个。

`type` 字段就是描述了找到所需数据时使用的扫描方式是什么，常见扫描类型的执行效率从低到高的顺序为：

- All (全表扫描)；
- index (全索引扫描)：index 和 all 差不多，只不过 index 对索引表进行全扫描，这样做的好处是不再需要对数据进行排序，但是开销依然很大。所以，要尽量避免全表扫描和全索引扫描。
- range (索引范围扫描)：一般在 where 子句中使用 <、>、in、between 等关键词，只检索给定范围的行，属于范围查找。从这一级别开始，索引的作用会越来越明显，因此我们需要尽量让 SQL 查询可以使用到 range 这一级别及以上的 type 访问方式。
- ref (非唯一索引扫描)：采用了非唯一索引，或者是唯一索引的非唯一性前缀，返回数据返回可能是多条。因为虽然使用了索引，但该索引列的值并不唯一，有重复。这样即使使用索引快速查找到了第一条数据，仍然不能停止，要进行目标值附近的小范围扫描。但它的好处是它并不需要扫全表，因为索引是有序的，即便有重复值，也是在一个非常小的范围内扫描。
- eq_ref (唯一索引扫描)：使用主键或唯一索引时产生的访问方式，通常使用在多表联查中。比如，对两张表进行联查，关联条件是两张表的 user_id 相等，且 user_id 是唯一索引，那么使用 EXPLAIN 进行执行计划查看的时候，type 就会显示 eq_ref。
- const (结果只有一条的主键或唯一索引扫描)。

需要说明的是 const 类型和 eq_ref 都使用了主键或唯一索引，不过这两个类型有所区别，**const 是与常量进行比较，查询效率会更快，而 eq_ref 通常用于多表联查中。**

除了关注 type，我们也要关注 extra 显示的结果。这里说几个重要的参考指标：

- Using filesort：当查询语句中包含 group by 操作，而且无法利用索引完成排序操作的时候，这时不得不选择相应的排序算法进行，甚至可能会通过文件排序，效率是很低的，所以要避免这种问题的出现。
- Using temporary：用了用临时表保存中间结果，**MySQL 在对查询结果排序时使用临时表，常见于排序 order by 和分组查询 group by。效率低，要避免这种问题的出现。**
- Using index：所需数据只需在索引即可全部获得，不须要再到表中取数据，也就是**使用了覆盖索引**，避免了回表操作，效率不错。

InnoDB 和 MyISAM 的B+树区别

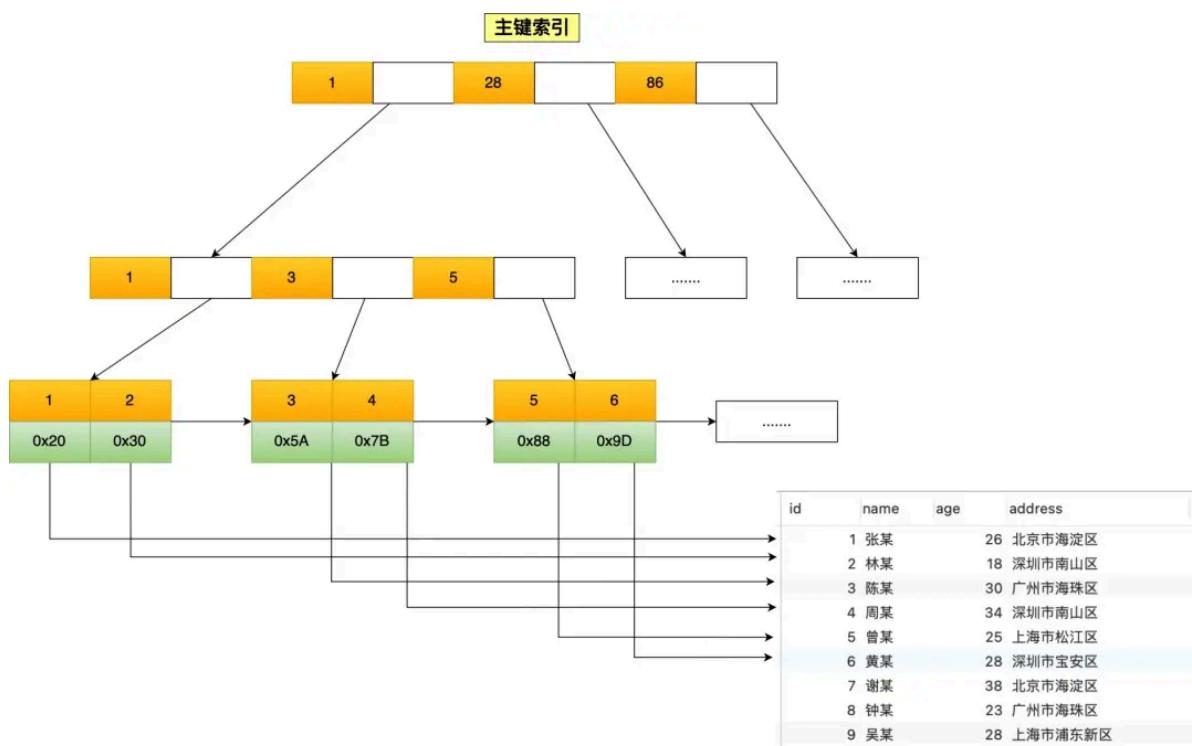
InnoDB 和 MyISAM 都支持 B+ 树索引，但是它们数据的存储结构实现方式不同。不同之处在于：

- InnoDB 存储引擎：B+ 树索引的叶子节点保存数据本身；
- MyISAM 存储引擎：B+ 树索引的叶子节点保存数据的物理地址；

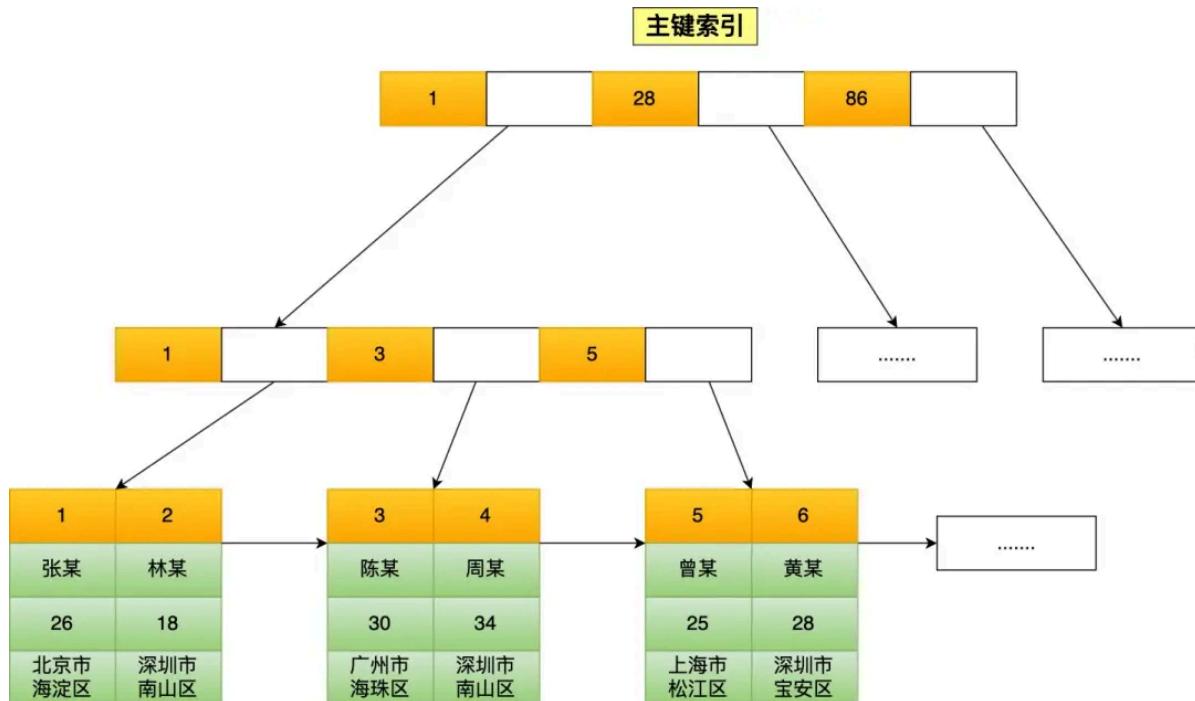
这里有一张 t_user 表，其中 id 字段为主键索引，其他都是普通字段。

id	name	age	address
1	张某	26	北京市海淀区
2	林某	18	深圳市南山区
3	陈某	30	广州市海珠区
4	周某	34	深圳市南山区
5	曾某	25	上海市松江区
6	黄某	28	深圳市宝安区
7	谢某	38	北京市海淀区
8	钟某	23	广州市海珠区
9	吴某	28	上海市浦东新区

如果使用的是 MyISAM 存储引擎，B+ 树索引的叶子节点保存数据的物理地址，即用户数据的指针，如下图：



如果使用的是 InnoDB 存储引擎，B+ 树索引的叶子节点保存数据本身，如下图所示（图中叶子节点之间我画了单向链表，但是实际上是双向链表，原图我找不到了，修改不了，偷个懒我不重画了，大家脑补成双向链表就行）。



对索引使用左或者左右模糊匹配

当我们使用左或者左右模糊匹配的时候，也就是 `like %xx` 或者 `like %xx%` 这两种方式都会造成索引失效。因为索引 B+ 树是按照「索引值」有序排列存储的，只能根据前缀进行比较。

对索引使用函数

有时候我们会用一些 MySQL 自带的函数来得到我们想要的结果，这时候要注意了，如果查询条件中对索引字段使用函数，就会导致索引失效。比如下面这条语句查询条件中对 `name` 字段使用了 `LENGTH` 函数，执行计划中的 `type=ALL`，代表了全表扫描：

```
// name 为二级索引
select * from t_user where length(name)=6;
```

```
1 explain select * from t_user where length(name)=6;
```

Message												Result 1	Profile	Status
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra			
1	SIMPLE	t_user	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	9	100.00	Using where			

因为索引保存的是索引字段的原始值，而不是经过函数计算后的值，自然就没办法走索引了。不过，从 MySQL 8.0 开始，索引特性增加了函数索引，即可以针对函数计算后的值建立一个索引，也就是说该索引的值是函数计算后的值，所以就可以通过扫描索引来查询数据。举个例子，我通过下面这条语句，对 `length(name)` 的计算结果建立一个名为 `idx_name_length` 的索引。

```
alter table t_user add key idx_name_length ((length(name)));
```

然后我再用下面这条查询语句，这时候就会走索引了。

```
1 explain select * from t_user where length(name)=6;
```

Message										Result 1	Profile	Status
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	t_user	(NULL)	ref	idx_name_length	idx_name_length	5	const	5	100.00	(NULL)	

对索引进行表达式计算

在查询条件中对索引进行表达式计算，也是无法走索引的。比如，下面这条查询语句，执行计划中 type = ALL，说明是通过全表扫描的方式查询数据的：

```
explain select * from t_user where id + 1 = 10;
```

```
1 explain select * from t_user where id + 1 = 10;
```

Message										Result 1	Profile	Status
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	t_user	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	9	100.00	Using where	

但是，如果把查询语句的条件改成 where id = 10 - 1，这样就不是在索引字段进行表达式计算了，于是就可以走索引查询了。

```
1 explain select * from t_user where id = 10 - 1;
```

Message										Result 1	Profile	Status
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra	
1	SIMPLE	t_user	(NULL)	const	PRIMARY	PRIMARY	8	const	1	100.00	(NULL)	

因为索引保存的是索引字段的原始值，而不是 $id + 1$ 表达式计算后的值，所以无法走索引，只能通过把索引字段的取值都取出来，然后依次进行表达式的计算来进行条件判断，因此采用的就是全表扫描的方式。有的同学可能会说，这种对索引进行简单的表达式计算，在代码特殊处理下，应该是可以做到索引扫描的，比方将 $id + 1 = 10$ 变成 $id = 10 - 1$ 。是的，是能够实现，但是 MySQL 还是偷了这个懒，没有实现。我的想法是，可能也是因为，表达式计算的情况多种多样，每种都要考虑的话，代码可能会很臃肿，所以干脆将这种索引失效的场景告诉程序员，让程序员自己保证在查询条件中不要对索引进行表达式计算。

对索引隐式类型转换

如果索引字段是字符串类型，但是在条件查询中，输入的参数是整型的话，你会在执行计划的结果发现这条语句会走全表扫描。我在原本的 t_user 表增加了 phone 字段，是二级索引且类型是 varchar。

Name	Type	Length	Dec
id	bigint	8	
name	varchar	30	
age	int	4	
address	varchar	255	
phone	varchar	30	

然后我在条件查询中，用整型作为输入参数，此时执行计划中 type = ALL，所以是通过全表扫描来查询数据的。

```
select * from t_user where phone = 1300000001;
```

```
1 explain select * from t_user where phone = 1300000001;
```

Message Result 1 Profile Status											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user	(NULL)	ALL	index_phone	(NULL)	(NULL)	(NULL)	9	11.11	Using where

但是如果索引字段是整型类型，查询条件中的输入参数即使字符串，是不会导致索引失效，还是可以走索引扫描。我们再看第二个例子，id 是整型，但是下面这条语句还是走了索引扫描的。

```
explain select * from t_user where id = '1';
```

```
1 explain select * from t_user where id = '1';
```

Message Result 1 Profile Status											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user	(NULL)	const	PRIMARY	PRIMARY	8	const	1	100.00	(NULL)

为什么第一个例子会导致索引失效，而第二例子不会呢？

要明白这个原因，首先我们要知道 MySQL 的数据类型转换规则是什么？就是看 MySQL 是会将字符串转成数字处理，还是将数字转换成字符串处理。我在看《mysql45讲的时候》看到一个简单的测试方式，就是通过 select "10" > 9 的结果来知道 MySQL 的数据类型转换规则是什么：

- 如果规则是 MySQL 会将自动「字符串」转换成「数字」，就相当于 select 10 > 9，这个就是数字比较，所以结果应该是 1；
- 如果规则是 MySQL 会将自动「数字」转换成「字符串」，就相当于 select "10" > "9"，这个是字符串比较，字符串比较大小是逐位从高位到低位逐个比较（按ascii码），那么"10"字符串相当于 "1" 和 "0" 字符的组合，所以先是拿 "1" 字符和 "9" 字符比较，因为 "1" 字符比 "9" 字符小，所以结果应该是 0。

在 MySQL 中，执行的结果如下图：

```
1 select "10" > 9
```

```
"10" > 9
```

```
1
```

上面的结果为 1，说明 MySQL 在遇到字符串和数字比较的时候，会自动把字符串转为数字，然后再进行比较。前面的例子一中的查询语句，我也跟大家说了是会走全表扫描：

```
//例子一的查询语句
select * from t_user where phone = 1300000001;
```

这是因为 phone 字段为字符串，所以 MySQL 要会自动把字符串转为数字，所以这条语句相当于：

```
select * from t_user where CAST(phone AS signed int) = 1300000001;
```

可以看到，**CAST** 函数是作用在了 phone 字段，而 phone 字段是索引，也就是对索引使用了函数！而前面我们也说了，对索引使用函数是会导致索引失效的。

例子二中的查询语句，我跟大家说了是会走索引扫描：

```
//例子二的查询语句
select * from t_user where id = "1";
```

这时因为字符串部分是输入参数，也就需要将字符串转为数字，所以这条语句相当于：

```
select * from t_user where id = CAST("1" AS signed int);
```

可以看到，索引字段并没有用任何函数，**CAST** 函数是用在了输入参数，因此是可以走索引扫描的。

联合索引非最左匹配

WHERE 子句中的 OR

在 WHERE 子句中，如果在 OR 前的条件列是索引列，而在 OR 后的条件列不是索引列，那么索引会失效。

MySQL 使用 like "%x"，索引一定会失效吗？

使用左模糊匹配 (like "%xx") 并不一定会走全表扫描，关键还是看数据表中的字段。如果数据库表中的字段只有主键+二级索引，那么即使使用了左模糊匹配，也不会走全表扫描 (type=all)，而是走全扫描二级索引树(type=index)。因为表的字段没有「非索引」字段，所以 `select *` 相当于 `select id, name`，然后这个查询的数据都在二级索引的 B+ 树，因为二级索引的 B+ 树的叶子节点包含「索引值 + 主键值」，所以查二级索引的 B+ 树就能查到全部结果了，这个就是覆盖索引。

再说一个相似，我们都知道联合索引要遵循最左匹配才能走索引，但是如果数据库表中的字段都是索引的话，即使查询过程中，没有遵循最左匹配原则，也是走全扫描二级索引树(type=index)，比如下图：

```
1 show CREATE TABLE t;
2
3 CREATE TABLE `t` (
4     `a` int NOT NULL,
5     `b` int NOT NULL,
6     `c` int NOT NULL,
7     `id` int NOT NULL AUTO_INCREMENT,
8     PRIMARY KEY (`id`) USING BTREE,
9     KEY `abc` (`a`,`b`,`c`)
10 ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
11
12 explain select * from t where c = 1;
```

Message Result 1 Profile Status											
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t	(NULL)	index	abc	abc	12	(NULL)	3	33.33	Using where; Using index

MySQL中有哪些锁

在 MySQL 里，根据加锁的范围，可以分为

1. 全局锁
2. 表级锁
3. 行锁

全局锁

要使用全局锁，则要执行这条命令：

```
flush tables with read lock
```

执行后，整个数据库就处于只读状态了，这时其他线程执行以下操作，都会被阻塞：

- 对数据的增删改操作，比如 insert、delete、update 等语句；
- 对表结构的更改操作，比如 alter table、drop table 等语句。

如果要释放全局锁，则要执行这条命令：

```
unlock tables
```

当然，当会话断开了，全局锁会被自动释放。

全局锁应用场景是什么？

全局锁主要应用于做全库逻辑备份，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期的不一样。加上全局锁，意味着整个数据库都是只读状态。那么如果数据库里有很多数据，备份就会花费很多的时间，关键是备份期间，业务只能读数据，而不能更新数据，这样会造成业务停滞。

但如果数据库的引擎支持的事务支持可重复读的隔离级别，那么在备份数据库之前先开启事务，会先创建 Read View，然后整个事务执行期间都在用这个 Read View，而且由于 MVCC 的支持，备份期间业务依然可以对数据进行更新操作。备份数据库的工具是 mysqldump，在使用 mysqldump 时加上 `-single-transaction` 参数的时候，就会在备份数据库之前先开启事务。这种方法只适用于支持「可重复读隔离级别的事务」的存储引擎。InnoDB 存储引擎默认的事务隔离级别正是可重复读，因此可以采用这种方式来备份数据库。但是，对于 MyISAM 这种不支持事务的引擎，在备份数据库时就要使用全局锁的方法。

表级锁

MySQL 里面表级别的锁有这几种：

- 表锁；
- 元数据锁（MDL）；
- 意向锁；
- AUTO-INC 锁；

表锁

如果我们想对学生表 (t_student) 加表锁，可以使用下面的命令：

```
//表级别的共享锁，也就是读锁；其它会话和当前会话只能对表进行读操作，不能进行写操作  
lock tables t_student read;
```

```
//表级别的独占锁，也就是写锁；其它会话不能对表进行读操作和写操作，当前会话可以进行读写操作  
lock tables t_stuent write;
```

要释放表锁，可以使用下面这条命令，会释放当前会话的所有表锁：

```
unlock tables
```

另外，当会话退出后，也会释放所有表锁。不过尽量避免在使用 InnoDB 引擎的表使用表锁，因为表锁的颗粒度太大，会影响并发性能，**InnoDB 牛逼的地方在于实现了颗粒度更细的行级锁。**

元数据锁 (MDL)

我们不需要显式的使用 MDL，因为当我们对数据库表进行操作时，会自动给这个表加上 MDL：

- 对一张表进行 CRUD 操作时，加的是 **MDL 读锁**；当有线程在执行 select 语句（加 MDL 读锁）的期间，如果有其他线程要更改该表的结构（申请 MDL 写锁），那么将会被阻塞，直到执行完 select 语句（释放 MDL 读锁）。
- 对一张表做结构变更操作的时候，加的是 **MDL 写锁**；当有线程对表结构进行变更（加 MDL 写锁）的期间，如果有其他线程执行了 CRUD 操作（申请 MDL 读锁），那么就会被阻塞，直到表结构变更完成（释放 MDL 写锁）。

MDL 是为了保证当用户对表执行 CRUD 操作时，**防止其他线程对这个表结构做了变更**。MDL 是在事务提交后才会释放，这意味着**事务执行期间，MDL 是一直持有的**。

那如果数据库有一个**长事务（所谓的长事务，就是开启了事务，但是一直还没提交）**，那在对表结构做变更操作的时候，可能会发生意想不到的事情，比如下面这个顺序的场景：

1. 首先，线程 A 先启用了事务（但是一直不提交），然后执行一条 select 语句，此时就先对该表加上 MDL 读锁；
2. 然后，线程 B 也执行了同样的 select 语句，此时并不会阻塞，因为「读读」并不冲突；
3. 接着，线程 C 修改了表字段，此时由于线程 A 的事务并没有提交，也就是 MDL 读锁还在占用着，这时线程 C 就无法申请到 MDL 写锁，就会被阻塞，

那么在线程 C 阻塞后，后续有对该表的 select 语句，就都会被阻塞，如果此时有大量该表的 select 语句的请求到来，就会有大量的线程被阻塞住，这时数据库的线程很快就会爆满了。

为什么线程 C 因为申请不到 MDL 写锁，而导致后续的申请读锁的查询操作也会被阻塞？

这是因为申请 MDL 锁的操作会形成一个队列，队列中写锁获取优先级高于读锁，一旦出现 MDL 写锁等待，会阻塞后续该表的所有 CRUD 操作。所以为了能安全的对表结构进行变更，在对表结构变更前，先要看数据库中的长事务，是否有事务已经对表加上了 MDL 读锁，如果可以考虑 kill 掉这个长事务，然后再做表结构的变更。

意向锁

- 在使用 InnoDB 引擎的表里对某些记录加上「共享锁」之前，需要先在表级别加上一个「意向共享锁」；
- 在使用 InnoDB 引擎的表里对某些纪录加上「独占锁」之前，需要先在表级别加上一个「意向独占锁」；

也就是说，当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。而普通的 select 是不会加行级锁的，普通的 select 语句是利用 MVCC 实现一致性读，是无锁的。不过，select 也是可以对记录加共享锁和独占锁的，具体方式如下：

```
//先在表上加上意向共享锁，然后对读取的记录加共享锁  
select ... lock in share mode;  
  
//先表上加上意向独占锁，然后对读取的记录加独占锁  
select ... for update;
```

意向共享锁和意向独占锁是表级锁，不会和行级的共享锁和独占锁发生冲突，而且意向锁之间也不会发生冲突，只会和共享表锁 (*lock tables ... read*) 和独占表锁 (*lock tables ... write*) 发生冲突。

表锁和行锁是满足读读共享、读写互斥、写写互斥的。如果没有「意向锁」，那么加「独占表锁」时，就需要遍历表里所有记录，查看是否有记录存在独占锁，这样效率会很慢。那么有了「意向锁」，由于在对记录加独占锁前，先会加上表级别的意向独占锁，那么在加「独占表锁」时，直接查该表是否有意向独占锁，如果有就意味着表里已经有记录被加了独占锁，这样就不用去遍历表里的记录。所以，意向锁的目的是为了快速判断表里是否有记录被加锁。

AUTO-INC 锁

表里的主键通常都会设置成自增的，这是通过对主键字段声明 `AUTO_INCREMENT` 属性实现的。之后可以在插入数据时，可以不指定主键的值，数据库会自动给主键赋值递增的值，这主要是通过 AUTO-INC 锁实现的。

AUTO-INC 锁是特殊的表锁机制，锁不是再一个事务提交后才释放，而是再执行完插入语句后就会立即释放。在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 `AUTO_INCREMENT` 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。那么，一个事务在持有 AUTO-INC 锁的过程中，其他事务的如果要向该表插入语句都会被阻塞，从而保证插入数据时，被 `AUTO_INCREMENT` 修饰的字段的值是连续递增的。

但是，AUTO-INC 锁再对大量数据进行插入的时候，会影响插入性能，因为另一个事务中的插入会被阻塞。因此，在 MySQL 5.1.22 版本开始，InnoDB 存储引擎提供了一种轻量级的锁来实现自增。一样也是在插入数据的时候，会为被 `AUTO_INCREMENT` 修饰的字段加上轻量级锁，然后给该字段赋值一个自增的值，就把这个轻量级锁释放了，而不需要等待整个插入语句执行完后才释放锁。

一般设置 `innodb_autoinc_lock_mode = 2`（只对自增主键加轻量级锁），并且 `binlog_format = row`，既能提升并发性，又不会出现数据一致性问题。

行级锁

InnoDB 引擎是支持行级锁的，而 MyISAM 引擎并不支持行级锁。普通的 select 语句是不会对记录加锁的，因为它属于快照读。如果要在查询时对记录加行锁，可以使用下面这两个方式，这种查询会加锁的语句称为锁定读。

```
//对读取的记录加共享锁  
select ... lock in share mode;  
  
//对读取的记录加独占锁  
select ... for update;
```

上面这两条语句必须在一个事务中，**因为当事务提交了，锁就会被释放**，所以在使用这两条语句的时候，要加上 begin、start transaction 或者 set autocommit = 0。共享锁（S锁）满足读读共享，读写互斥。独占锁（X锁）满足写写互斥、读写互斥。

行级锁的类型主要有三类：

- **Record Lock**, 记录锁，也就是仅仅把一条记录锁上；
- **Gap Lock**, 间隙锁，锁定一个范围，但是不包含记录本身；
- **Next-Key Lock: Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。**

Record Lock

Record Lock 称为记录锁，锁住的是一条记录。而且记录锁是有 S 锁和 X 锁之分的：

- 当一个事务对一条记录加了 S 型记录锁后，其他事务也可以继续对该记录加 S 型记录锁（S 型与 S 锁兼容），但是不可以对该记录加 X 型记录锁（S 型与 X 锁不兼容）；
- 当一个事务对一条记录加了 X 型记录锁后，其他事务既不可以对该记录加 S 型记录锁（S 型与 X 锁不兼容），也不可以对该记录加 X 型记录锁（X 型与 X 锁不兼容）。

举个例子，当一个事务执行了下面这条语句：

```
mysql > begin;  
mysql > select * from t_test where id = 1 for update;
```

就是对 t_test 表中主键 id 为 1 的这条记录加上 X 型的记录锁，这样其他事务就无法对这条记录进行修改了。**当事务执行 commit 后，事务过程中生成的锁都会被释放。**

Gap Lock

Gap Lock 称为间隙锁，只存在于可重复读隔离级别，目的是为了解决可重复读隔离级别下幻读的现象。假设，表中有一个范围 id 为 (3, 5) 间隙锁，那么其他事务就无法插入 id = 4 这条记录了，这样就有效的防止幻读现象的发生。间隙锁虽然存在 X 型间隙锁和 S 型间隙锁，但是并没有什么区别，**间隙锁之间是兼容的，即两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系，因为间隙锁的目的是防止插入幻影记录而提出的。**

Next-Key Lock

Next-Key Lock 称为临键锁，是 Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。假设，表中有一个范围 id 为 (3, 5] 的 next-key lock，那么其他事务即不能插入 id = 4 记录，也不能修改 id = 5 这条记录。

临键锁: (3, 5]

The diagram shows a table with two columns: 'id' and 'name'. The 'id' column has values 1, 3, 5, and 6. The 'name' column has values 路飞, 索隆, 乌索普, and 山治. A light blue box at the top contains the text '临键锁: (3, 5]'. Two lines point from this box to the cell containing '5' in the 'id' column, indicating that this range is locked.

	id 列:	1	3	5	6
	name 列:	路飞	索隆	乌索普	山治

所以，next-key lock 即能保护该记录，又能阻止其他事务将新纪录插入到被保护记录前面的间隙中。
next-key lock 是包含间隙锁+记录锁的，如果一个事务获取了 X 型的 next-key lock，那么另外一个事务在获取相同范围的 X 型的 next-key lock 时，是会被阻塞的。虽然相同范围的间隙锁是多个事务相互兼容的，但对于记录锁，我们是要考虑 X 型与 S 型关系，X 型的记录锁与 X 型的记录锁是冲突的。

插入意向锁

一个事务在插入一条记录的时候，需要判断插入位置是否已被其他事务加了间隙锁（next-key lock 也包含间隙锁）。如果有的话，插入操作就会发生阻塞，直到拥有间隙锁的那个事务提交为止（释放间隙锁的时刻），在此期间会生成一个插入意向锁，表明有事务想在某个区间插入新记录，但是现在处于等待状态。

举个例子，假设事务 A 已经对表加了一个范围 id 为 (3, 5) 间隙锁。

间隙锁: (3, 5]

The diagram shows a table with two columns: 'id' and 'name'. The 'id' column has values 1, 3, 5, and 6. The 'name' column has values 路飞, 索隆, 乌索普, and 山治. A light blue box at the top contains the text '间隙锁: (3, 5]'. Two lines point from this box to the cell containing '3' in the 'id' column, indicating that this point is locked by an intent lock.

	id 列:	1	3	5	6
	name 列:	路飞	索隆	乌索普	山治

当事务 A 还没提交的时候，事务 B 向该表插入一条 id = 4 的新记录，这时会判断到插入的位置已经被事务 A 加了间隙锁，于是事务 B 会生成一个插入意向锁，然后将锁的状态设置为等待状态（PS: MySQL 加锁时，是先生成锁结构，然后设置锁的状态，如果锁状态是等待状态，并不是意味着事务成功获取到了锁，只有当锁状态为正常状态时，才代表事务成功获取到了锁），此时事务 B 就会发生阻塞，直到事务 A 提交了事务。

插入意向锁名字虽然有意向锁，但是它并不是意向锁，它是一种特殊的间隙锁，属于行级别锁。如果说间隙锁锁住的是一个区间，那么「插入意向锁」锁住的就是一个点。因而从这个角度来说，插入意向锁确实是一种特殊的间隙锁。

插入意向锁允许多个事务在同一个间隙上同时设置插入意向锁，但这些事务最终插入的记录必须是互不冲突的。插入意向锁与间隙锁的另一个非常重要的差别是：尽管「插入意向锁」也属于间隙锁，但两个事务却不能在同一时间内，一个拥有间隙锁，另一个拥有该间隙区间内的插入意向锁（当然，插入意向锁如果不在间隙锁区间内则是可以的）。

什么 SQL 语句会加行级锁？

注意一般使用sql语句除了加行级锁外，还会加表级的意向锁。

InnoDB 引擎是支持行级锁的，而 MyISAM 引擎并不支持行级锁。所以，在说 MySQL 是怎么加行级锁的时候，其实是在说 InnoDB 引擎是怎么加行级锁的。

普通的 select 语句是不会对记录加锁的（除了串行化隔离级别），因为它属于快照读，是通过 MVCC（多版本并发控制）实现的。如果要在查询时对记录加行级锁，可以使用下面这两个方式，这两种查询会加锁的语句称为**锁定读**。

```
//对读取的记录加共享锁(S型锁)
select ... lock in share mode;

//对读取的记录加独占锁(X型锁)
select ... for update;
```

上面这两条语句必须在一个事务中，**因为当事务提交了，锁就会被释放**，所以在使用这两条语句的时候，要加上 begin 或者 start transaction 开启事务的语句。**除了上面这两条锁定读语句会加行级锁之外，update 和 delete 操作都会加行级锁，且锁的类型都是独占锁(X型锁)。**

```
//对操作的记录加独占锁(X型锁)
update table .... where id = 1;

//对操作的记录加独占锁(X型锁)
delete from table where id = 1;
```

MySQL 是怎么加行级锁的？

加锁的对象是索引，加锁的基本单位是 **next-key lock**，它是由记录锁和间隙锁组合而成的，**next-key lock** 是前开后闭区间，而间隙锁是前开后开区间。

但是，next-key lock 在一些场景下会退化成记录锁或间隙锁。那到底是什么场景呢？总结一句，**在能使用记录锁或者间隙锁就能避免幻读现象的场景下，next-key lock 就会退化成记录锁或间隙锁。**

唯一索引等值查询：

- 当查询的记录是「存在」的，在索引树上定位到这一条记录后，将该记录的索引中的 next-key lock 会退化成「记录锁」。
- 当查询的记录是「不存在」的，在索引树找到第一条大于该查询记录的记录后，将该记录的索引中的 next-key lock 会退化成「间隙锁」。

非唯一索引等值查询：

- 当查询的记录「存在」时，由于不是唯一索引，所以肯定存在索引值相同的记录，于是非唯一索引等值查询的过程是一个扫描的过程，直到扫描到第一个不符合条件的二级索引记录就停止扫描，然后在扫描的过程中，对扫描到的二级索引记录加的是 next-key 锁，而对于第一个不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。同时，在符合查询条件的记录的主键索引上加记录锁。
- 当查询的记录「不存在」时，扫描到第一条不符合条件的二级索引记录，该二级索引的 next-key 锁会退化成间隙锁。因为不存在满足查询条件的记录，所以不会对主键索引加锁。

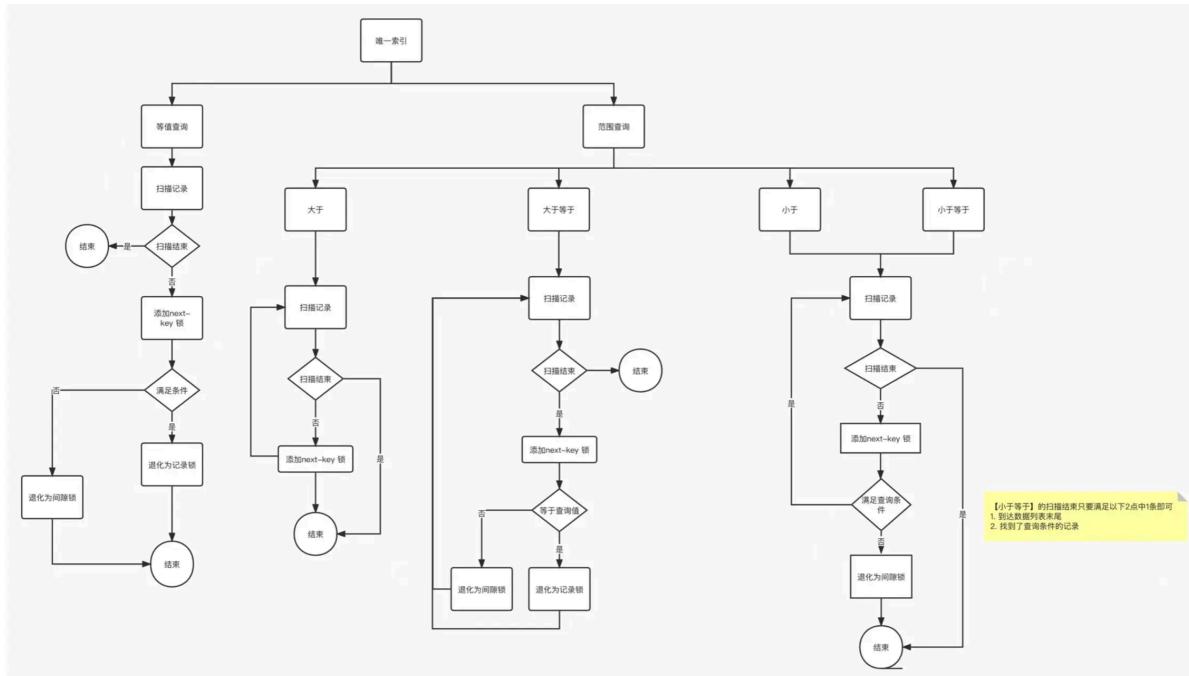
非唯一索引和主键索引的范围查询的加锁规则不同之处在于：

- 唯一索引在满足一些条件的时候，索引的 next-key lock 退化为间隙锁或者记录锁。
- 非唯一索引范围查询，索引的 next-key lock 不会退化为间隙锁和记录锁。

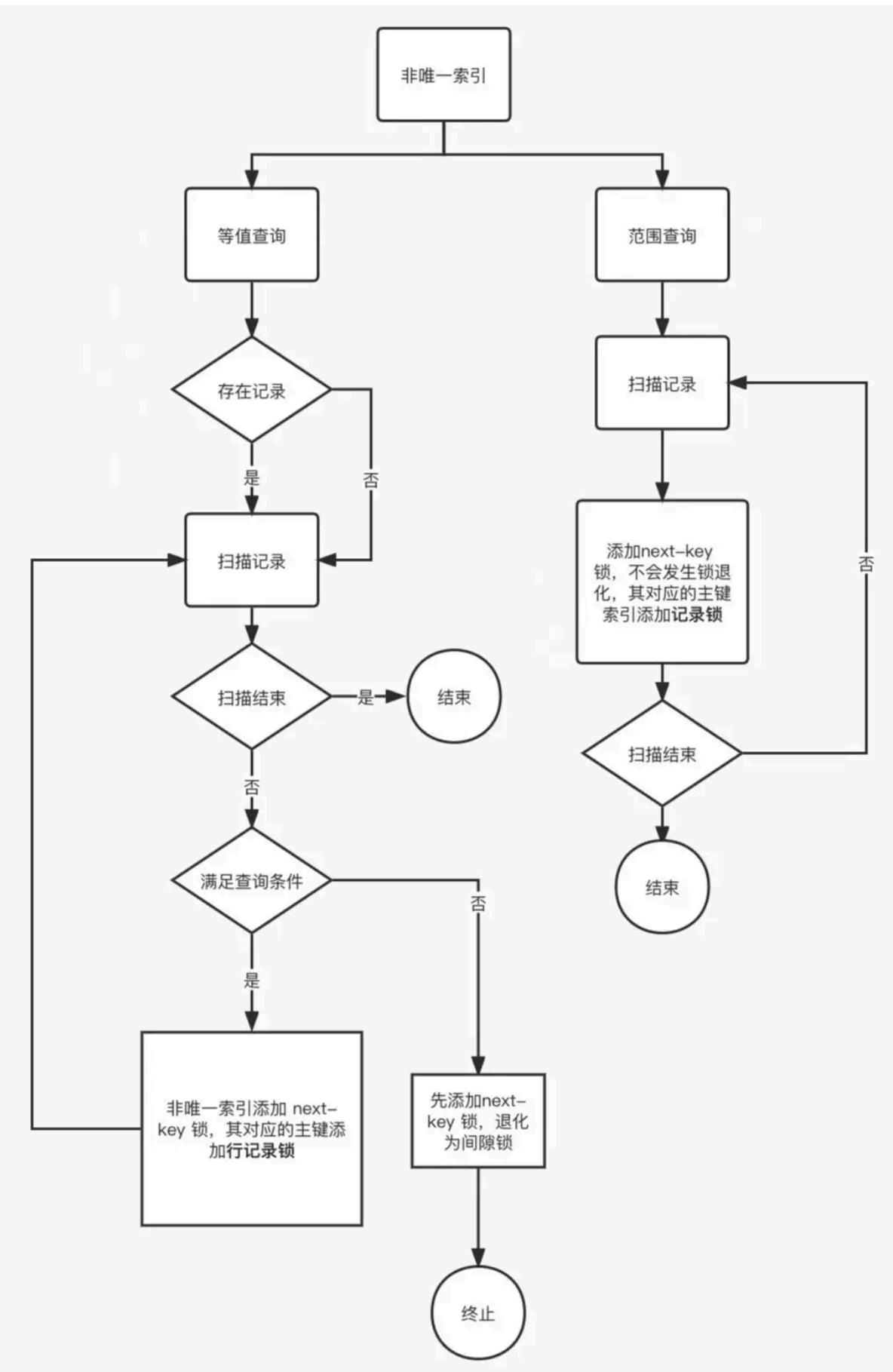
其实理解 MySQL 为什么要这样加锁，主要要以避免幻读角度去分析，这样就很容易理解这些加锁的规则了。

还有一件很重要的事情，在线上在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，**如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了，这是挺严重的问题。**

唯一索引（主键索引）加锁的流程图如下。（注意这个流程图是针对「主键索引」的，如果是二级索引的唯一索引，除了流程图中对二级索引的加锁规则之外，还会对查询到的记录的主键索引项加「记录锁」，流程图没有提示这一个点，所以在这里用文字补充说明下）



非唯一索引加锁的流程图：



MySQL 记录锁+间隙锁可以防止删除操作而导致的幻读吗？

在 MySQL 的可重复读隔离级别下，针对当前读的语句会对索引加记录锁+间隙锁，这样可以避免其他事务执行增、删、改时导致幻读的问题。有一点要注意的是，在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了，这是挺严重的问题。

MySQL死锁

场景

本次案例使用存储引擎 InnoDB，隔离级别为可重复读（RR）。我建了一张订单表，其中 id 字段为主键索引，order_no 字段普通索引，也就是非唯一索引：

```
CREATE TABLE `t_order` (
  `id` int NOT NULL AUTO_INCREMENT,
  `order_no` int DEFAULT NULL,
  `create_date` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `index_order` (`order_no`) USING BTREE
) ENGINE=InnoDB ;
```

然后，先 t_order 表里现在已经有了 6 条记录：

id	order_no	create_date
1	1001	2021-12-28 13:59:07
2	1002	2021-12-28 13:59:14
3	1003	2021-12-28 13:59:21
4	1004	2021-12-28 13:59:30
5	1005	2021-12-28 13:59:36
6	1006	2021-12-28 14:24:33

假设这时有两事务，一个事务要插入订单 1007，另外一个事务要插入订单 1008，因为需要对订单做幂等性校验（即使一个操作被执行多次，其结果也应保持不变），所以两个事务先要查询该订单是否存在，不存在才插入记录，过程如下：

事务 A	事务 B
begin;	begin;
//检查 1007 订单是否存在 select id from t_order where order_no = 1007 for update;	
	//检查 1008 订单是否存在 select id from t_order where order_no = 1008 for update;
//如果没有，则插入订单记录 insert into t_order (order_no, create_date) values (1007, now()); 阻塞等待....	
	//如果没有，则插入订单记录 insert into t_order (order_no, create_date) values (1008, now()); 阻塞等待....

可以看到，两个事务都陷入了等待状态（前提没有打开死锁检测），也就是发生了死锁，因为都在相互等待对方释放锁。这里在查询记录是否存在时候，使用了 `select ... for update` 语句，目的为了防止事务执行的过程中，有其他事务插入了记录，而出现幻读的问题。如果没有使用 `select ... for update` 语句，而使用了单纯的 `select` 语句，如果是两个订单号一样的请求同时进来，就会出现两个重复的订单，有可能出现幻读，如下图：

事务 A	事务 B
<code>begin;</code>	<code>begin;</code>
//检查 1007 订单是否存在 <code>select id from t_order where order_no = 1007</code>	
	//检查 1007 订单是否存在 <code>select id from t_order where order_no = 1007</code>
//如果没有，则插入订单记录 <code>insert into t_order (order_no, create_date) values (1007, now());</code>	
	//如果没有，则插入订单记录 <code>insert into t_order (order_no, create_date) values (1007, now());</code>
<code>commit;</code>	<code>commit;</code>
存在两条订单为 1007 的记录（幻读）	

死锁是怎么产生的

事务 A 在执行下面这条语句的时候，我们可以通过 `select * from performance_schema.data_locks\G;` 这条语句，查看事务执行 SQL 过程中加了什么锁。

```
select id from t_order where order_no = 1007 for update;
```

```
*****
1. row *****
    ENGINE: INNODB
    ENGINE_LOCK_ID: 140515090899640:1093:140515095894464
ENGINE_TRANSACTION_ID: 23968397
    THREAD_ID: 94
    EVENT_ID: 19
    OBJECT_SCHEMA: test
    OBJECT_NAME: t_order
    PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
    INDEX_NAME: NULL
OBJECT_INSTANCE_BEGIN: 140515095894464
    LOCK_TYPE: TABLE
    LOCK_MODE: IX
    LOCK_STATUS: GRANTED
    LOCK_DATA: NULL
*****
2. row *****
    ENGINE: INNODB
    ENGINE_LOCK_ID: 140515090899640:32:6:1:140515098031136
ENGINE_TRANSACTION_ID: 23968397
    THREAD_ID: 94
    EVENT_ID: 19
    OBJECT_SCHEMA: test
    OBJECT_NAME: t_order
    PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
    INDEX_NAME: index_order
OBJECT_INSTANCE_BEGIN: 140515098031136
    LOCK_TYPE: RECORD
    LOCK_MODE: X
    LOCK_STATUS: GRANTED
    LOCK_DATA: supremum pseudo-record
```

从上图可以看到，共加了两个锁，分别是：

- 表锁：X类型的意向锁；
- 行锁：X类型的next-key锁；

这里我们重点关注行锁，图中 LOCK_TYPE 中的 RECORD 表示行级锁，而不是记录锁的意思，通过 LOCK_MODE 可以确认是 next-key 锁，还是间隙锁，还是记录锁：

- 如果 LOCK_MODE 为 `X`，说明是 X 型的 next-key 锁；
- 如果 LOCK_MODE 为 `X, REC_NOT_GAP`，说明是 X 型的记录锁；
- 如果 LOCK_MODE 为 `X, GAP`，说明是 X 型的间隙锁；

因此，此时事务 A 在二级索引（INDEX_NAME : index_order）上加的是 X 型的 next-key 锁，锁范围是 `(1006, +∞]`。

当事务 B 往事务 A next-key 锁的范围 `(1006, +∞]` 里插入 id = 1008 的记录就会被锁住：

```
Insert into t_order (order_no, create_date) values (1008, now());
```

因为当我们执行以下插入语句时，会在插入间隙上获取插入意向锁，而插入意向锁与间隙锁是冲突的，所以当其它事务持有该间隙的间隙锁时，需要等待其它事务释放间隙锁之后，才能获取到插入意向锁。而间隙锁与间隙锁之间是兼容的（间隙锁意义只是为了阻止区间被插入），所以两个事务中 `select ... for update` 语句并不会相互影响。

案例中的事务 A 和事务 B 在执行完后 `select ... for update` 语句后都持有范围为 `(1006, +∞]` 的 next-key 锁，而接下来的插入操作为了获取到插入意向锁，都在等待对方事务的间隙锁释放，于是就造成了循环等待，导致死锁。

Insert 语句是怎么加行级锁的？

Insert 语句在正常执行时是不会生成锁结构的，它是靠聚簇索引记录自带的 `trx_id` 隐藏列来作为隐式锁来保护记录的。当事务需要加锁时，如果这个锁不可能发生冲突，InnoDB 会跳过加锁环节，这种机制称为隐式锁。隐式锁是 InnoDB 实现的一种延迟加锁机制，其特点是只有在可能发生冲突时才加锁，从而减少了锁的数量，提高了系统整体性能。

隐式锁就是在 Insert 过程中不加锁，只有在特殊情况下，才会将隐式锁转换为显示锁，这里我们列举两个场景。

- 如果记录之间加有间隙锁，为了避免幻读，此时是不能插入记录的；
- 如果 Insert 的记录和已有记录存在唯一键冲突，此时也不能插入记录；

如何避免死锁？

死锁的四个必要条件：**互斥、占有且等待、不可强占用、循环等待**。只要系统发生死锁，这些条件必然成立，但是只要破坏任意一个条件就死锁就不会成立。

在数据库层面，有两种策略通过「打破循环等待条件」来解除死锁状态：

- **设置事务等待锁的超时时间。** 当一个事务的等待时间超过该值后，就对这个事务进行回滚，于是锁就释放了，另一个事务就可以继续执行了。在 InnoDB 中，参数 `innodb_lock_wait_timeout` 是用来设置超时时间的，默认值时 50 秒。

当发生超时后，就出现下面这个提示：

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

- **开启主动死锁检测。** 主动死锁检测在发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。将参数 `innodb_deadlock_detect` 设置为 `on`，表示开启这个逻辑，默认就开启。

当检测到死锁后，就会出现下面这个提示：

```
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
```

上面这个两种策略是「当有死锁发生时」的避免方式。我们可以回归业务的角度来预防死锁，对订单做幂等性校验的目的是为了保证不会出现重复的订单，那我们可以直接将 `order_no` 字段设置为唯一索引列，利用它的唯一性来保证订单表不会出现重复的订单，不过有一点不好的地方就是在我们插入一个已经存在的订单记录时就会抛出异常。

MySQL中的日志：

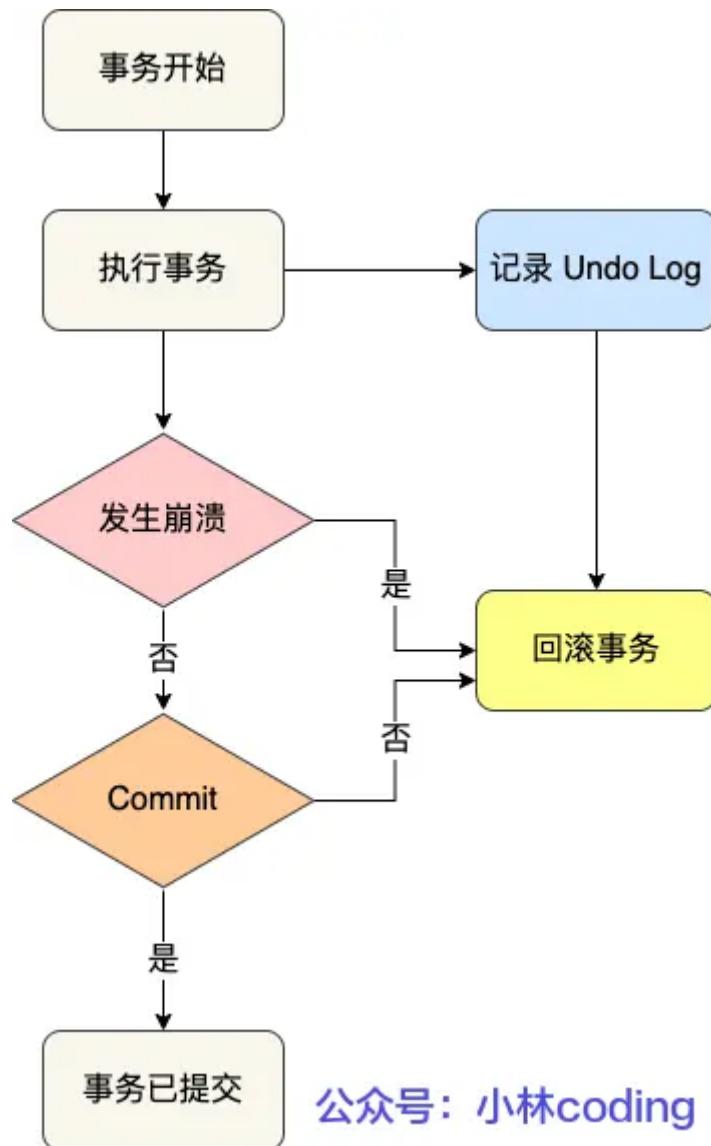
- **undo log (回滚日志)**：是 Innodb 存储引擎层生成的日志，实现了事务中的**原子性**，主要**用于事务回滚和 MVCC**。
- **redo log (重做日志)**：是 Innodb 存储引擎层生成的日志，实现了事务中的**持久性**，主要**用于掉电等故障恢复**；
- **binlog (归档日志)**：是 Server 层生成的日志，主要**用于数据备份和主从复制**；

为什么需要 undo log？

我们在执行执行一条“增删改”语句的时候，虽然没有输入 begin 开启事务和 commit 提交事务，但是 MySQL 会**隐式开启事务**来执行“增删改”语句的，执行完就自动提交事务的，这样就保证了执行完“增删改”语句后，我们可以及时在数据库表看到“增删改”的结果了。

我们每次在事务执行过程中，都记录下回滚时需要的信息到一个日志里，那么在事务执行中途发生了 MySQL 崩溃后，就不用担心无法回滚到事务之前的数据，我们可以通过这个日志回滚到事务之前的数据。实现这一机制就是 **undo log (回滚日志)**，它保证了事务的**ACID 特性中的原子性 (Atomicity)**。

undo log 是一种用于撤销回退的日志。在**事务没提交之前**，MySQL 会先记录更新前的数据到 undo log 日志文件里面，当事务回滚时，可以利用 undo log 来进行回滚。



每当 InnoDB 引擎对一条记录进行操作（修改、删除、新增）时，要把回滚时需要的信息都记录到 undo log 里，比如：

公众号：小林coding

- 在插入一条记录时，要把这条记录的主键值记下来，这样之后回滚时只需要把这个主键值对应的记录删掉就好了；
- 在删除一条记录时，要把这条记录中的内容都记下来，这样之后回滚时再把由这些内容组成的记录插入到表中就好了；
- 在更新一条记录时，要把被更新的列的旧值记下来，这样之后回滚时再把这些列更新为旧值就好了。

在发生回滚时，就读取 undo log 里的数据，然后做原先相反操作。比如当 delete 一条记录时，undo log 中会把记录中的内容都记下来，然后执行回滚操作的时候，就读取 undo log 里的数据，然后进行 insert 操作。

版本链



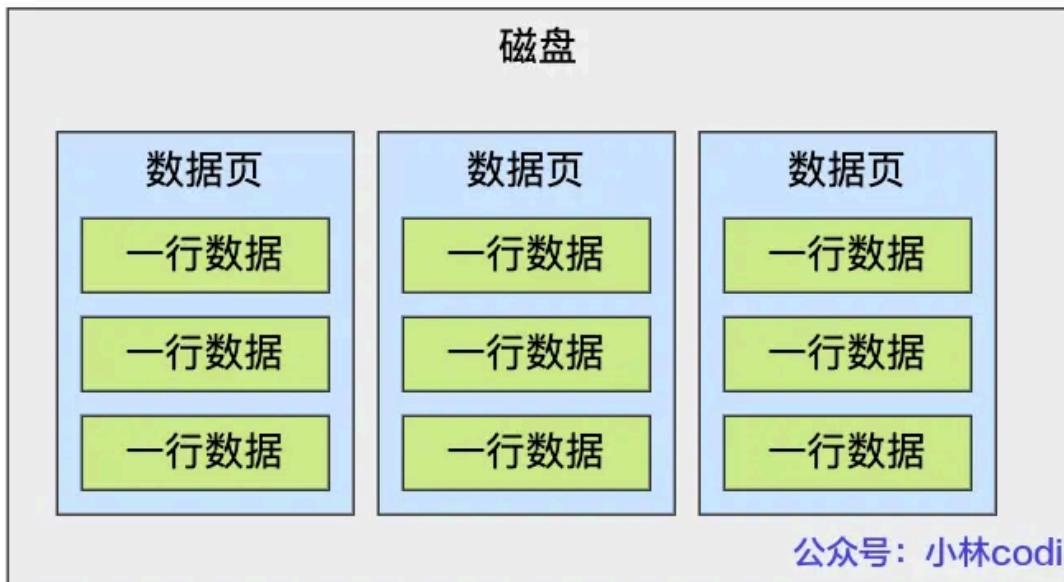
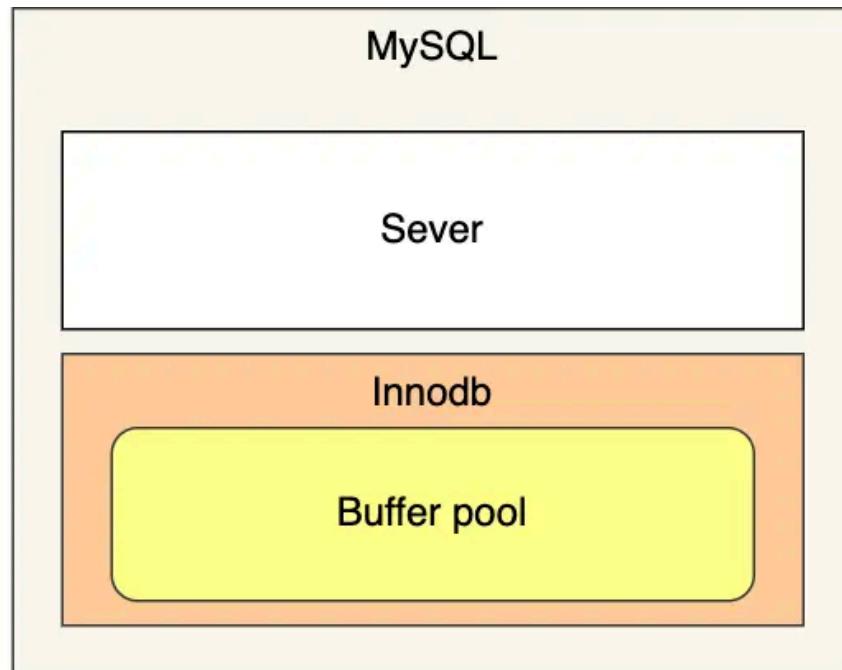
undo log 还有一个作用，通过 ReadView + undo log 实现 MVCC（多版本并发控制）。

undo log 是如何刷盘（持久化到磁盘）的？

undo log 和数据页的刷盘策略是一样的，都需要通过 redo log 保证持久化。buffer pool 中有 undo 页，对 undo 页的修改也都会记录到 redo log。redo log 会每秒刷盘，提交事务时也会刷盘，数据页和 undo 页都是靠这个机制保证持久化的。

为什么需要 Buffer Pool？

MySQL 的数据都是存在磁盘中的，那么我们要更新一条记录的时候，得先要从磁盘读取该记录，然后在内存中修改这条记录。那修改完这条记录是选择直接写回到磁盘，还是选择缓存起来呢？当然是缓存起来好，这样下次有查询语句命中了这条记录，直接读取缓存中的记录，就不需要从磁盘获取数据了。为此，Innodb 存储引擎设计了一个缓冲池（Buffer Pool），来提高数据库的读写性能。



有了 Buffer Pool 后：

- 当读取数据时，如果数据存在于 Buffer Pool 中，客户端就会直接读取 Buffer Pool 中的数据，否则再去磁盘中读取。
- 当修改数据时，如果数据存在于 Buffer Pool 中，那直接修改 Buffer Pool 中数据所在的页，然后将其页设置为脏页（该页的内存数据和磁盘上的数据已经不一致），为了减少磁盘I/O，不会立即将脏页写入磁盘，后续由后台线程选择一个合适的时机将脏页写入到磁盘。

Buffer Pool 缓存什么？

InnoDB 会把存储的数据划分为若干个「页」，以页作为磁盘和内存交互的基本单位，一个页的默认大小为 16KB。因此，Buffer Pool 同样需要按「页」来划分。在 MySQL 启动的时候，**InnoDB 会为 Buffer Pool 申请一片连续的内存空间，然后按照默认的 16KB 的大小划分出一个个的页，Buffer Pool 中的页就叫做缓存页**。此时这些缓存页都是空闲的，之后随着程序的运行，才会有磁盘上的页被缓存到 Buffer Pool 中。

所以，MySQL 刚启动的时候，你会观察到使用的虚拟内存空间很大，而使用到的物理内存空间却很小，这是因为只有这些虚拟内存被访问后，操作系统才会触发缺页中断，申请物理内存，接着将虚拟地址和物理地址建立映射关系。Buffer Pool 除了缓存「索引页」和「数据页」，还包括了 Undo 页，插入缓存、自适应哈希索引、锁信息等等。



Undo 页是记录什么？

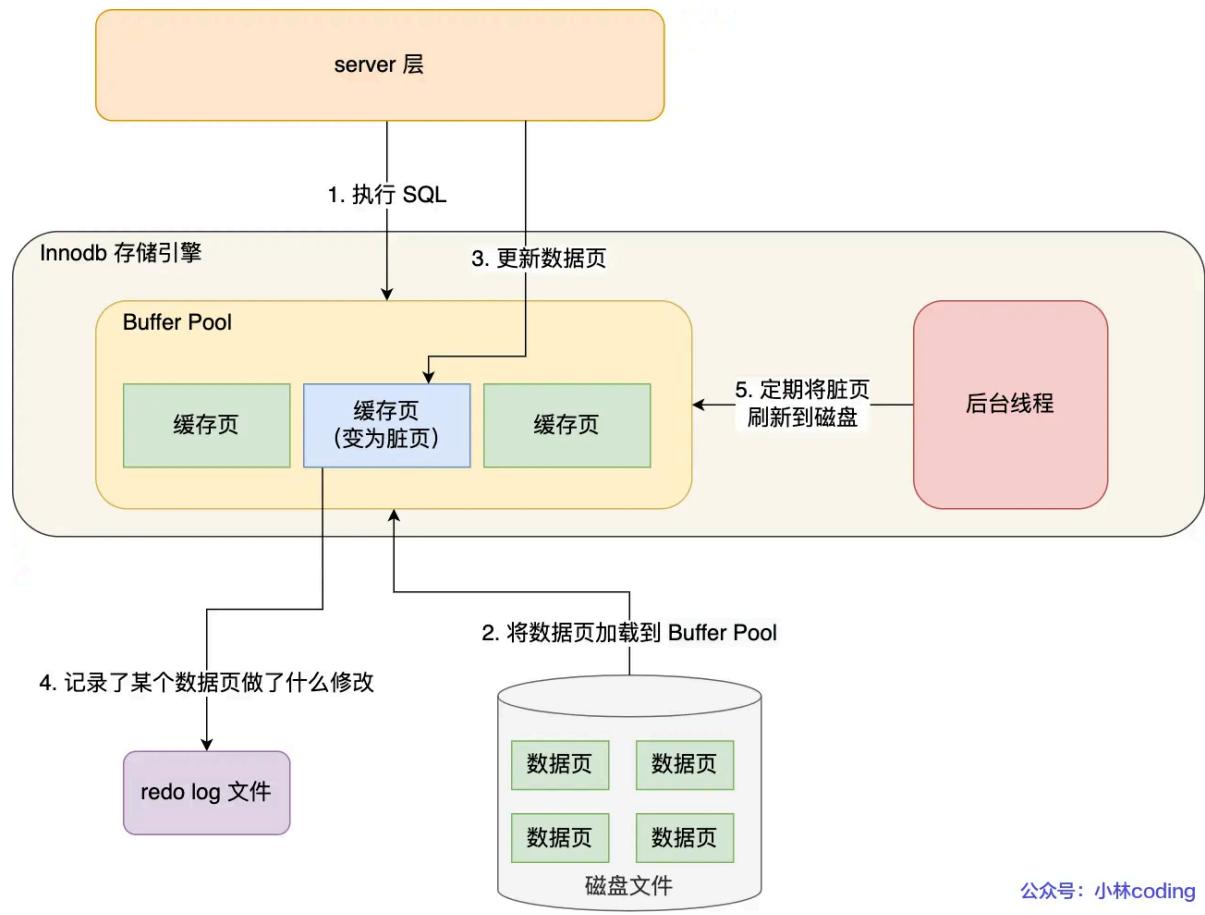
开启事务后，InnoDB 在更新记录前，首先要记录相应的 undo log，如果是更新操作，需要把被更新的列的旧值记下来，也就是要生成一条 undo log，undo log 会写入 Buffer Pool 中的 Undo 页面，在内存修改该 Undo 页面后，需要记录对应的 redo log。

查询一条记录，就只需要缓存一条记录吗？

不是的。当我们查询一条记录时，InnoDB 是会把整个页的数据加载到 Buffer Pool 中，将页加载到 Buffer Pool 后，再通过页里的「页目录」去定位到某条具体的记录。

为什么需要 redo log？

Buffer Pool 是基于内存的，为了防止断电导致数据丢失的问题，当有一条记录需要更新的时候，InnoDB 引擎就会先更新内存（同时标记为脏页），然后将本次对这个页的修改以 redo log 的形式记录下来，这个时候更新就算完成了。后续，InnoDB 引擎会在适当的时候，由后台线程将缓存在 Buffer Pool 的脏页刷新到磁盘里，这就是 WAL (Write-Ahead Logging) 技术。WAL 技术指的是，MySQL 的写操作并不是立刻写到磁盘上，而是先写日志，然后在合适的时间再写到磁盘上。



redo log 是物理日志，记录了某个数据页做了什么修改，比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了 AAA 更新，每当执行一个事务就会产生这样的一条或者多条物理日志。在事务提交时，只要先将 redo log 持久化到磁盘即可，可以不需要等到将缓存在 Buffer Pool 里的脏页数据持久化到磁盘。当系统崩溃时，虽然脏页数据没有持久化，但是 redo log 已经持久化，接着 MySQL 重启后，可以根据 redo log 的内容，将所有数据恢复到最新的状态。

redo log 和 undo log 的区别

- redo log 记录了此次事务「完成后」的数据状态，记录的是更新之后的值；
- undo log 记录了此次事务「开始前」的数据状态，记录的是更新之前的值；

redo log 要写到磁盘，数据也要写磁盘，为什么要多此一举？

写入 redo log 的方式使用了追加操作，所以磁盘操作是顺序写，而写入数据需要先找到写入位置，然后才写到磁盘，所以磁盘操作是随机写。磁盘的「顺序写」比「随机写」高效的多，因此 redo log 写入磁盘的开销更小。

- 实现事务的持久性，让 MySQL 有 crash-safe 的能力，能够保证 MySQL 在任何时间段突然崩溃，重启后之前已提交的记录都不会丢失；
- 将写操作从「随机写」变成了「顺序写」，提升 MySQL 写入磁盘的性能。

产生的 redo log 是直接写入磁盘的吗？

redo log 也有自己的缓存——redo log buffer，每当产生一条 redo log 时，会先写入到 redo log buffer，后续在持久化到磁盘如下图：

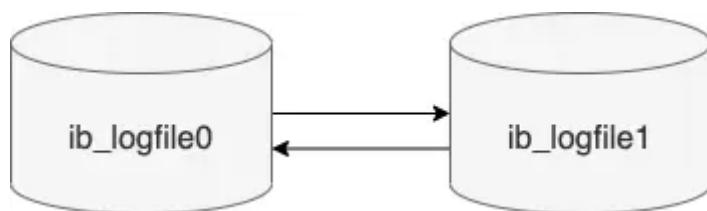
redo log 什么时候刷盘?

主要有以下几个时机:

- MySQL 正常关闭时;
- 当 redo log buffer 中记录的写入量大于 redo log buffer 内存空间的一半时, 会触发落盘;
- InnoDB 的后台线程每隔 1 秒, 将 redo log buffer 持久化到磁盘。
- 每次事务提交时都将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘

redo log 文件写满了怎么办?

默认情况下, InnoDB 存储引擎有 1 个重做日志文件组(redo log Group) , 「重做日志文件组」由有 2 个 redo log 文件组成, 这两个 redo 日志的文件名叫 : `ib_logfile0` 和 `ib_logfile1` 。在重做日志组中, 每个 redo log File 的大小是固定且一致的, 假设每个 redo log File 设置的上限是 1 GB, 那么总共就可以记录 2GB 的操作。重做日志文件组是以**循环写**的方式工作的, 从头开始写, 写到末尾就又回到开头, 相当于一个环形。



我们知道 redo log 是为了防止 Buffer Pool 中的脏页丢失而设计的, 那么如果随着系统运行, Buffer Pool 的脏页刷新到了磁盘中, 那么 redo log 对应的记录也就没用了, 这时候我们擦除这些旧记录, 以腾出空间记录新的更新操作。

redo log 文件满了, 这时 MySQL 不能再执行新的更新操作, 也就是说 MySQL 会被阻塞 (因此所以针对并发量大的系统, 适当设置 redo log 的文件大小非常重要), 此时会停下来将 Buffer Pool 中的脏页刷新到磁盘中, 然后标记 redo log 哪些记录可以被擦除, 接着对旧的 redo log 记录进行擦除, 等擦除完旧记录腾出了空间, **checkpoint** 就会往后移动 (图中顺时针) , 然后 MySQL 恢复正常运行, 继续执行新的更新操作。

为什么需要 binlog ?

undo log 和 redo log 这两个日志都是 Innodb 存储引擎生成的。 MySQL 在完成一条更新操作后, Server 层还会生成一条 binlog, 等之后事务提交的时候, 会将该事物执行过程中产生的所有 binlog 统一写入 binlog 文件。binlog 文件是记录了所有数据库表结构变更和表数据修改的日志, 不会记录查询类的操作, 比如 SELECT 和 SHOW 操作。

只依靠 binlog 是没有 crash-safe 能力的, 所以 InnoDB 使用 redo log 来实现 crash-safe 能力。

redo log 和 binlog 有什么区别?

1、**适用对象不同:**

- binlog 是 MySQL 的 Server 层实现的日志, 所有存储引擎都可以使用;
- redo log 是 Innodb 存储引擎实现的日志;

2、**文件格式不同:**

- binlog 有 3 种格式类型, 分别是 STATEMENT (默认格式) 、 ROW、 MIXED, 区别如下:
- redo log 是物理日志, 记录的是在某个数据页做了什么修改, 比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了 AAA 更新;

3、写入方式不同：

- binlog 是追加写，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志。
- redo log 是循环写，日志空间大小是固定，全部写满就从头开始，保存未被刷入磁盘的脏页日志。

4、用途不同：

- binlog 用于备份恢复、主从复制；
- redo log 用于掉电等故障恢复。

如果不小心整个数据库的数据被删除了，能使用 redo log 文件恢复数据吗？

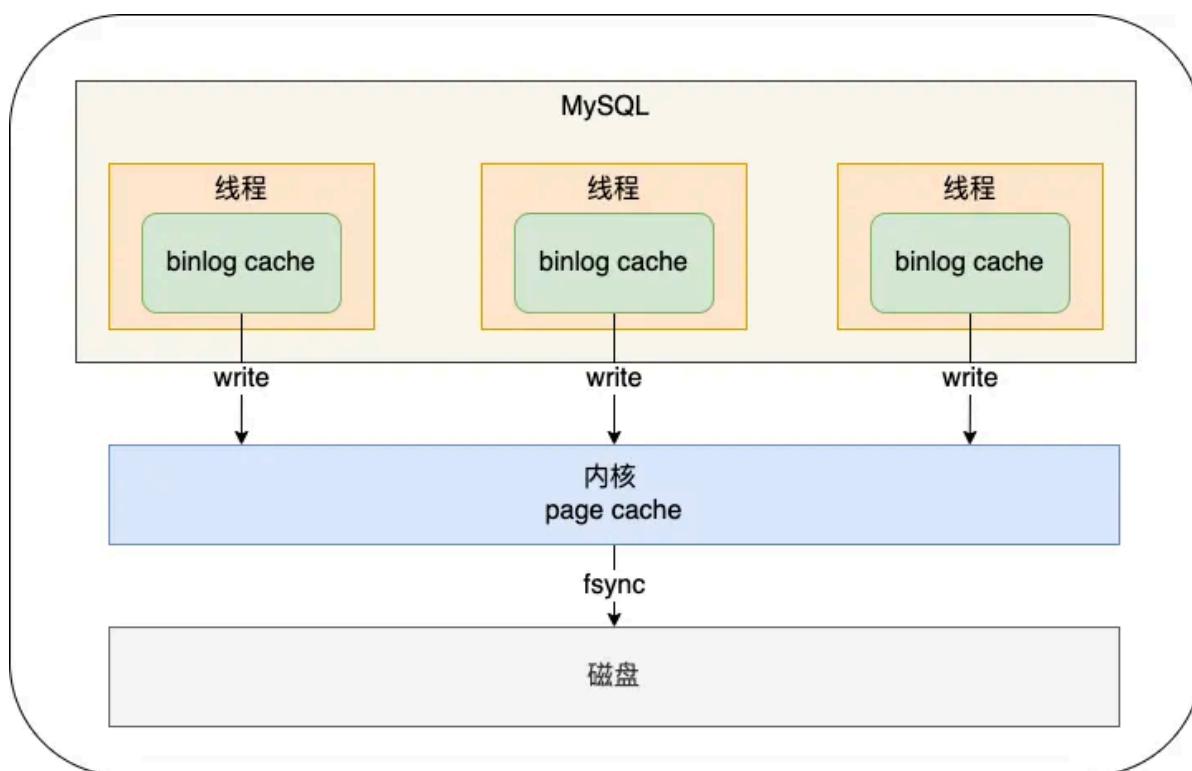
不可以使用 redo log 文件恢复，只能使用 binlog 文件恢复。因为 redo log 文件是循环写，是会边写边擦除日志的，只记录未被刷入磁盘的数据的物理日志，已经刷入磁盘的数据都会从 redo log 文件里擦除。

binlog 文件保存的是全量的日志，也就是保存了所有数据变更的情况，理论上只要记录在 binlog 上的数据，都可以恢复，所以如果不小心整个数据库的数据被删除了，得用 binlog 文件恢复数据。

binlog 什么时候刷盘？

事务执行过程中，先把日志写到 binlog cache (Server 层的 cache)，事务提交的时候，再把 binlog cache 写到 binlog 文件中。一个事务的 binlog 是不能被拆开的，因此无论这个事务有多大（比如有很多条语句），也要保证一次性写入。

在事务提交的时候，执行器把 binlog cache 里的完整事务写入到 binlog 文件中，并清空 binlog cache。如下图：



虽然每个线程有自己 binlog cache，但是最终都写到同一个 binlog 文件：

- 图中的 write，就是指把日志写入到 binlog 文件，但是并没有把数据持久化到磁盘，因为数据还缓存在文件系统的 page cache 里，write 的写入速度还是比较快的，因为不涉及磁盘 I/O。
- 图中的 fsync，才是将数据持久化到磁盘的操作，这里就会涉及磁盘 I/O，所以频繁的 fsync 会导致磁盘的 I/O 升高。

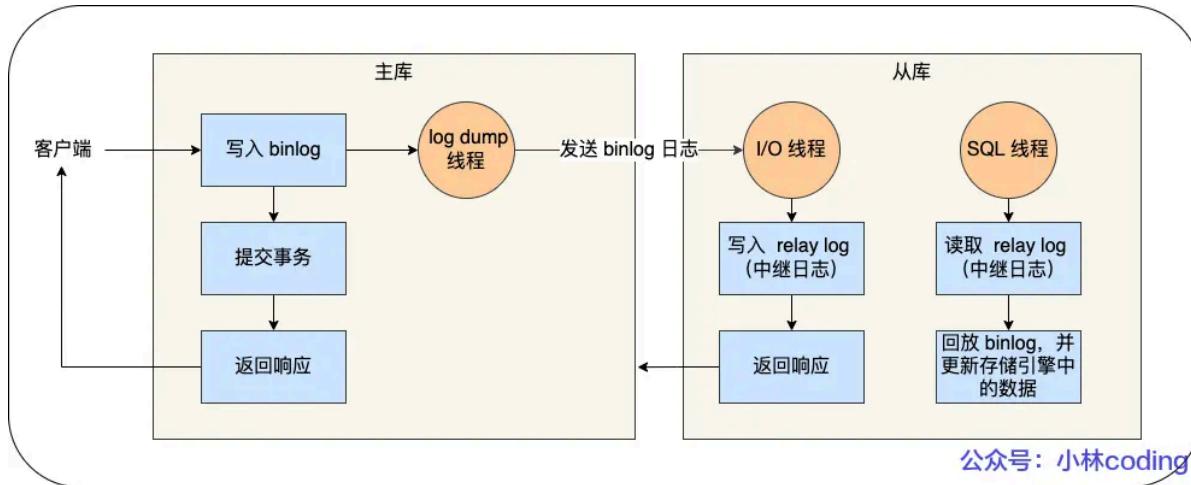
MySQL提供一个 sync_binlog 参数来控制数据库的 binlog 刷到磁盘上的频率：

- sync_binlog = 0 的时候，表示每次提交事务都只 write，不 fsync，后续交由操作系统决定何时将数据持久化到磁盘；
- sync_binlog = 1 的时候，表示每次提交事务都会 write，然后马上执行 fsync；
- sync_binlog =N(N>1) 的时候，表示每次提交事务都 write，但累积 N 个事务后才 fsync。

在MySQL中系统默认的设置是 sync_binlog = 0，也就是不做任何强制性的磁盘刷新指令，这时候的性能是最好的，但是风险也是最大的。因为一旦主机发生异常重启，还没持久化到磁盘的数据就会丢失。

MySQL主从复制是怎么实现？

MySQL 的主从复制依赖于 binlog，也就是记录 MySQL 上的所有变化并以二进制形式保存在磁盘上。复制的过程就是将 binlog 中的数据从主库传输到从库上。这个过程一般是异步的，也就是主库上执行事务操作的线程不会等待复制 binlog 的线程同步完成。



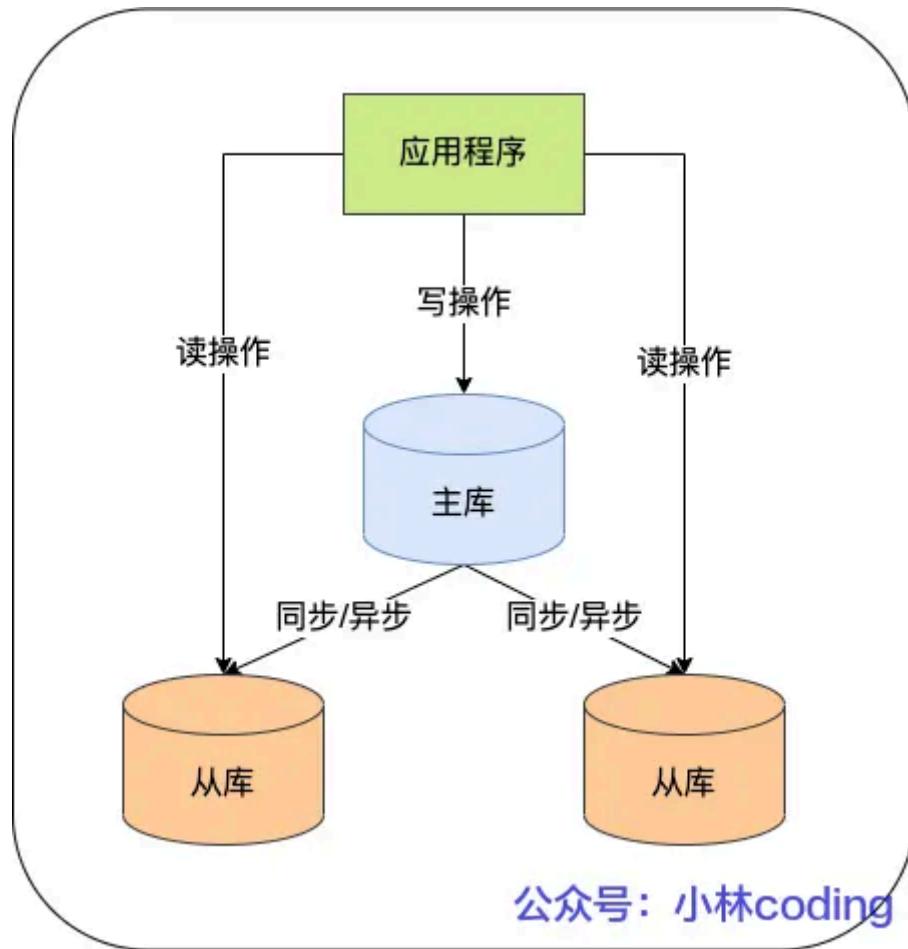
MySQL 集群的主从复制过程梳理成 3 个阶段：

- **写入 Binlog**: 主库写 binlog 日志，提交事务，并更新本地存储数据。
- **同步 Binlog**: 把 binlog 复制到所有从库上，每个从库把 binlog 写到暂存日志中。
- **回放 Binlog**: 回放 binlog，并更新存储引擎中的数据。

具体详细过程如下：

- MySQL 主库在收到客户端提交事务的请求之后，会先写入 binlog，再提交事务，更新存储引擎中的数据，事务提交完成后，返回给客户端“操作成功”的响应。
- 从库会创建一个专门的 I/O 线程，连接主库的 log dump 线程，来接收主库的 binlog 日志，**再把 binlog 信息写入 relay log 的中继日志里，再返回给主库“复制成功”的响应。**
- **从库会创建一个用于回放 binlog 的线程，去读 relay log 中继日志，然后回放 binlog 更新存储引擎中的数据，最终实现主从的数据一致性。**

在完成主从复制之后，你就可以在写数据时只写主库，在读数据时只读从库，这样即使写请求会锁表或者锁记录，也不会影响读请求的执行。



公众号：小林coding

从库是不是越多越好？

不是的。因为从库数量增加，从库连接上来的 I/O 线程也比较多，**主库也要创建同样多的 log dump 线程来处理复制的请求，对主库资源消耗比较高，同时还受限于主库的网络带宽**。所以在实际使用中，一个主库一般跟 2~3 个从库（1 套数据库，1 主 2 从 1 备主），这就是一主多从的 MySQL 集群结构。

MySQL 主从复制还有哪些模型？

主要有三种：

- **同步复制**：MySQL 主库提交事务的线程要等待所有从库的复制成功响应，才返回客户端结果。这种方式在实际项目中，基本上没法用，原因有两个：一是性能很差，因为要复制到所有节点才返回响应；二是可用性也很差，主库和所有从库任何一个数据库出问题，都会影响业务。
- **异步复制**（默认模型）：MySQL 主库提交事务的线程并不会等待 binlog 同步到各从库，就返回客户端结果。这种模式一旦主库宕机，数据就会发生丢失。
- **半同步复制**：MySQL 5.7 版本之后增加的一种复制方式，介于两者之间，**事务线程不用等待所有的从库复制成功响应，只要一部分复制成功响应回来就行**，比如一主二从的集群，只要数据成功复制到任意一个从库上，主库的事务线程就可以返回给客户端。这种半同步复制的方式，兼顾了异步复制和同步复制的优点，即使出现主库宕机，至少还有一个从库有最新的数据，不存在数据丢失的风险。

update 语句的执行过程

当优化器分析出成本最小的执行计划后，执行器就按照执行计划开始进行更新操作。具体更新一条记录 `UPDATE t_user SET name = 'xiaolin' WHERE id = 1;` 的流程如下：

1. 执行器负责具体执行，会调用存储引擎的接口，通过主键索引树搜索获取 `id = 1` 这一行记录：
 - 如果 `id=1` 这一行所在的数据页本来就在 buffer pool 中，就直接返回给执行器更新；

- 如果记录不在 buffer pool，将数据页从磁盘读入到 buffer pool，返回记录给执行器。
2. 执行器得到聚簇索引记录后，会看一下更新前的记录和更新后的记录是否一样：
 - 如果一样的话就不进行后续更新流程；
 - 如果不一样的话就把更新前的记录和更新后的记录都当作参数传给 InnoDB 层，让 InnoDB 真正的执行更新记录的操作；
 3. **开启事务，InnoDB 层更新记录前，首先要记录相应的 undo log，因为这是更新操作，需要把被更新的列的旧值记下来，也就是要生成一条 undo log，undo log 会写入 Buffer Pool 中的 Undo 页面，不过在内存修改该 Undo 页面后，需要记录对应的 redo log。**
 4. **InnoDB 层开始更新记录，会先更新内存（同时标记为脏页），然后将记录写到 redo log 里面，这个时候更新就算完成了。为了减少磁盘I/O，不会立即将脏页写入磁盘，后续由后台线程选择一个合适的时机将脏页写入到磁盘。这就是 WAL 技术，MySQL 的写操作并不是立刻写到磁盘上，而是先写 redo 日志，然后在合适的时间再将修改的行数据写到磁盘上。**
 5. 至此，一条记录更新完了。
 6. **在一条更新语句执行完成后，然后开始记录该语句对应的 binlog，此时记录的 binlog 会被保存到 binlog cache，并没有刷新到硬盘上的 binlog 文件，在事务提交时才会统一将该事务运行过程中所有 binlog 刷新到硬盘。**
 7. 事务提交，剩下的就是「两阶段提交」的事情了，接下来就讲这个。

为什么需要两阶段提交？

事务提交后，redo log 和 binlog 都要持久化到磁盘，但是这两个是独立的逻辑，可能出现半成功的状态（也就是一个成功，一个不成功），这样就造成两份日志之间的逻辑不一致，尤其在主从复制中导致不一致性。MySQL 为了避免出现两份日志之间的逻辑不一致的问题，使用了「两阶段提交」来解决，两阶段提交其实是分布式事务一致性协议，它可以保证多个逻辑操作要不全部成功，要不全部失败，不会出现半成功的状态。

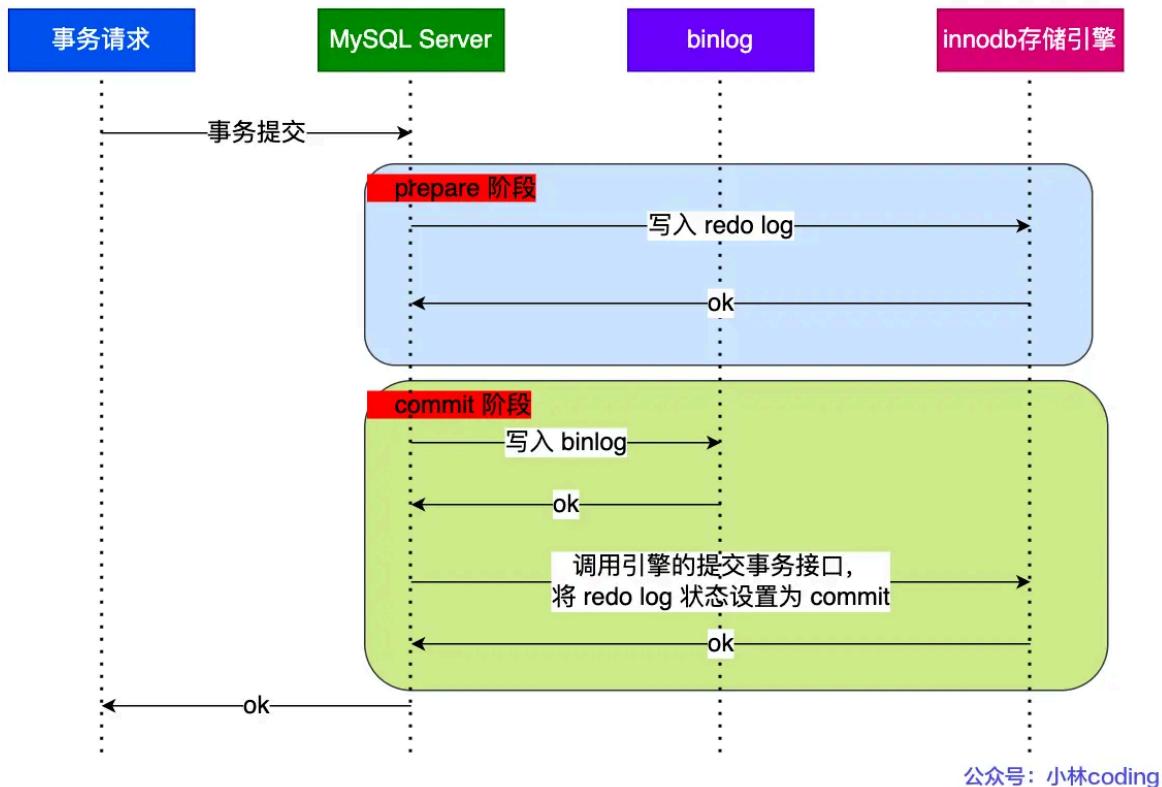
举个例子，假设 id = 1 这行数据的字段 name 的值原本是 'jay'，然后执行 `UPDATE t_user SET name = 'xiaolin' WHERE id = 1;` 如果在持久化 redo log 和 binlog 两个日志的过程中，出现了半成功状态，那么就有两种情况：

- **如果在将 redo log 刷入到磁盘之后，MySQL 突然宕机了，而 binlog 还没有来得及写入。MySQL 重启后，通过 redo log 能将 Buffer Pool 中 id = 1 这行数据的 name 字段恢复到新值 xiaolin，但是 binlog 里面没有记录这条更新语句，在主从架构中，binlog 会被复制到从库，由于 binlog 丢失了这条更新语句，从库的这一行 name 字段是旧值 jay，与主库的值不一致性；**
- **如果在将 binlog 刷入到磁盘之后，MySQL 突然宕机了，而 redo log 还没有来得及写入。由于 redo log 还没写，崩溃恢复以后这个事务无效，所以 id = 1 这行数据的 name 字段还是旧值 jay，而 binlog 里面记录了这条更新语句，在主从架构中，binlog 会被复制到从库，从库执行了这条更新语句，那么这一行 name 字段是新值 xiaolin，与主库的值不一致性；**

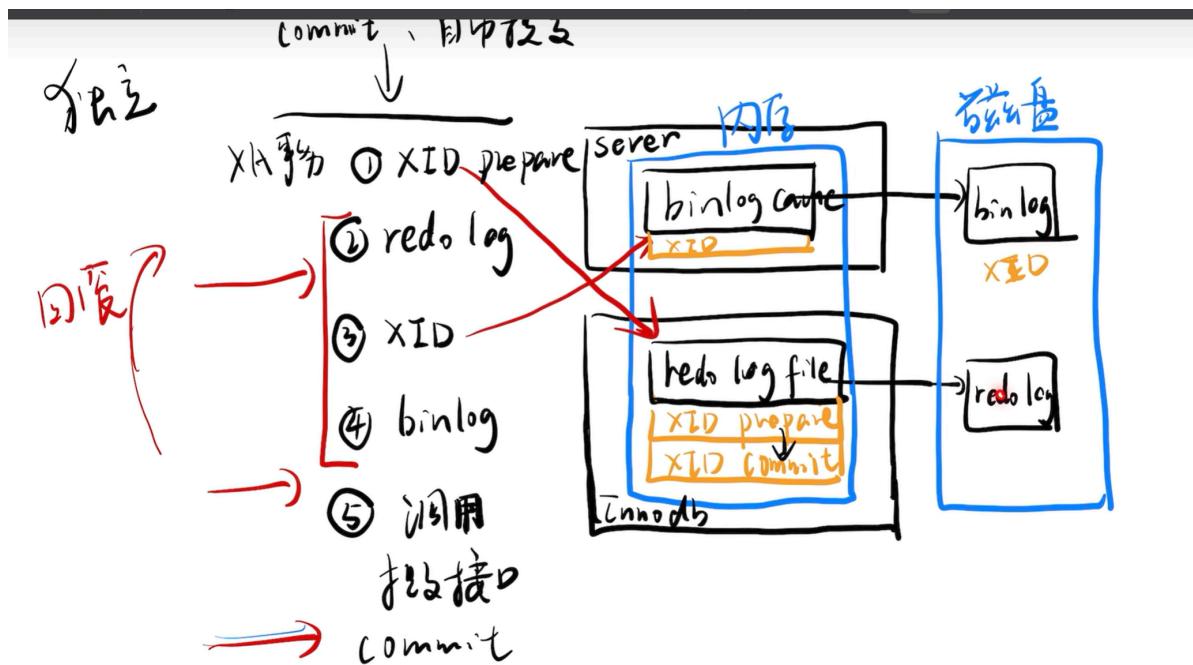
两阶段提交把单个事务的提交拆分成了 2 个阶段，分别是「准备（Prepare）阶段」和「提交（Commit）阶段」（注意这里的提交非事务的commit语句，commit语句执行的时候包含提交的阶段），每个阶段都由协调者（Coordinator）和参与者（Participant）共同完成。注意，不要把提交（Commit）阶段和 commit 语句混淆了，commit 语句执行的时候，会包含提交（Commit）阶段。

MySQL两阶段提交的过程是怎样的？

MySQL 使用了内部 XA 事务（是的，也有外部 XA 事务），内部 XA 事务由 binlog 作为协调者，存储引擎是参与者。当客户端执行 commit 语句或者在自动提交的情况下，MySQL 内部开启一个 XA 事务，分两阶段来完成 XA 事务的提交，如下图：



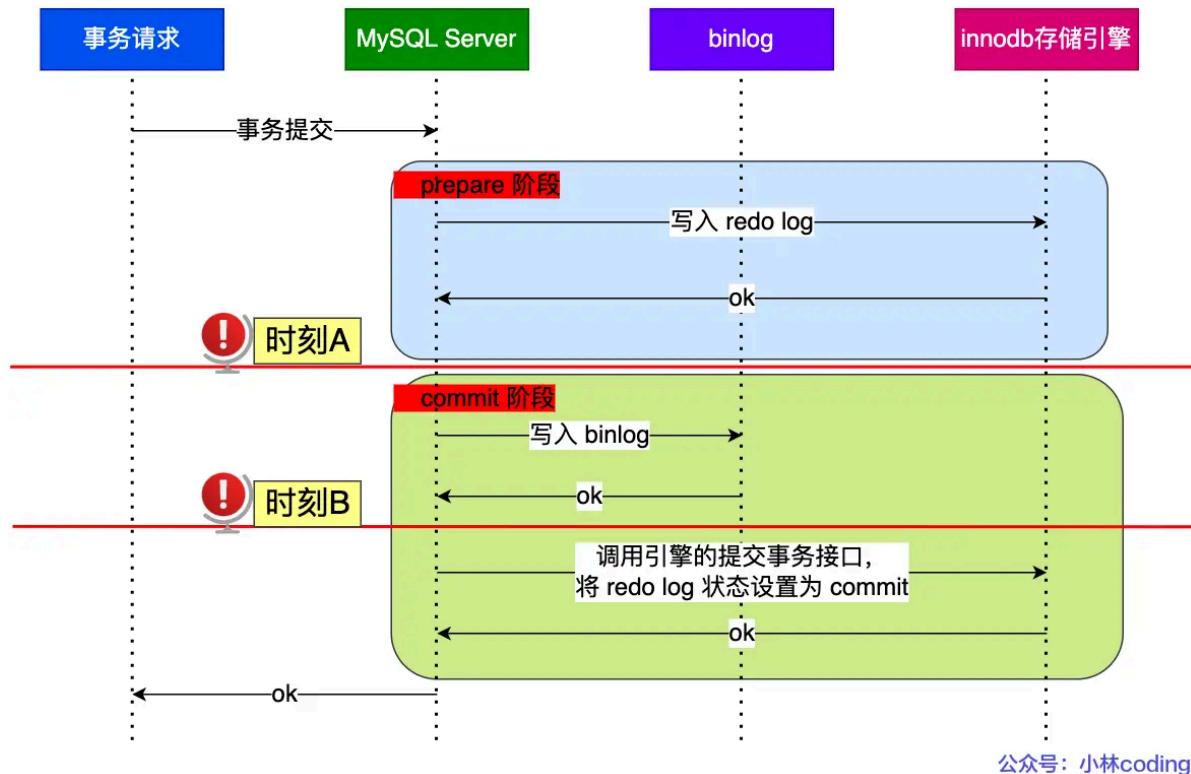
公众号：小林coding



从图中可看出，事务的提交过程有两个阶段，就是将 redo log 的写入拆成了两个步骤：prepare 和 commit，中间再穿插写入binlog，具体如下：

- **prepare 阶段:** 将 XID (内部 XA 事务的 ID) 写入到 redo log，同时将 redo log 对应的事务状态设置为 prepare，然后将 redo log 持久化到磁盘 (innodb_flush_log_at_trx_commit = 1 的作用)；
- **commit 阶段:** 把 XID 写入到 binlog，然后将 binlog 持久化到磁盘 (sync_binlog = 1 的作用)，接着调用引擎的提交事务接口，将 redo log 状态设置为 commit，此时该状态并不需要持久化到磁盘，只需要 write 到文件系统的 page cache 中就够了，因为只要 binlog 写磁盘成功 (也就是将 xid 写入了 binlog)，就算 redo log 的状态还是 prepare 也没有关系，一样会被认为事务已经执行成功；

两阶段提交的不同时刻，MySQL 异常重启会出现什么现象？



不管是时刻 A (redo log 已经写入磁盘, binlog 还没写入磁盘), 还是时刻 B (redo log 和 binlog 都已经写入磁盘, 还没写入 commit 标识) 崩溃, 此时的 redo log 都处于 prepare 状态。在 MySQL 重启后会按顺序扫描 redo log 文件, 碰到处于 prepare 状态的 redo log, 就拿着 redo log 中的 XID 去 binlog 查看是否存在此 XID:

- 如果 binlog 中没有当前内部 XA 事务的 XID, 说明 redolog 完成刷盘, 但是 binlog 还没有刷盘, 则回滚事务。对应时刻 A 崩溃恢复的情况。
- 如果 binlog 中有当前内部 XA 事务的 XID, 说明 redolog 和 binlog 都已经完成了刷盘, 则提交事务。对应时刻 B 崩溃恢复的情况。

可以看到, 对于处于 prepare 阶段的 redo log, 即可以提交事务, 也可以回滚事务, 这取决于是否能在 binlog 中查找到与 redo log 相同的 XID, 如果有就提交事务, 如果没有就回滚事务。这样就可以保证 redo log 和 binlog 这两份日志的一致性了。

所以说, 两阶段提交是以 binlog 写成功为事务提交成功的标识, 因为 binlog 写成功了, 就意味着能在 binlog 中查找到与 redo log 相同的 XID。

处于 prepare 阶段的 redo log 加上完整 binlog, 重启就提交事务, MySQL 为什么要这么设计?

binlog 已经写入了, 之后就会被从库 (或者用这个 binlog 恢复出来的库) 使用。所以, 在主库上也要提交这个事务。采用这个策略, 主库和备库的数据就保证了一致性。

事务没提交的时候, redo log 会被持久化到磁盘吗?

会的。事务执行中间过程的 redo log 也是直接写在 redo log buffer 中的, 这些缓存在 redo log buffer 里的 redo log 也会被「后台线程」每隔一秒一起持久化到磁盘。也就是说, 事务没提交的时候, redo log 也是可能被持久化到磁盘的。有的同学可能会问, 如果 mysql 崩溃了, 还没提交事务的 redo log 已经被持久化磁盘了, mysql 重启后, 数据不就不一致了? 放心, 这种情况 mysql 重启会进行回滚操作,

因为事务没提交的时候，binlog 是还没持久化到磁盘的。所以，redo log 可以在事务没提交之前持久化到磁盘，但是 binlog 必须在事务提交之后，才可以持久化到磁盘。

两阶段提交有什么问题？

两阶段提交虽然保证了两个日志文件的数据一致性，但是性能很差，主要有两个方面的影响：

- **磁盘 I/O 次数高**：对于“双1”配置，每个事务提交都会进行两次 fsync（刷盘），一次是 redo log 刷盘，另一次是 binlog 刷盘。
- **锁竞争激烈**：两阶段提交虽然能够保证「单事务」两个日志的内容一致，但在「多事务」的情况下，却不能保证两者的提交顺序一致，因此，在两阶段提交的流程基础上，还需要加一个锁来保证提交的原子性，从而保证多事务的情况下，两个日志的提交顺序一致。

为什么两阶段提交的磁盘 I/O 次数会很高？

binlog 和 redo log 在内存中都对应的缓存空间，binlog 会缓存在 binlog cache，redo log 会缓存在 redo log buffer，它们持久化到磁盘的时机分别由下面这两个参数控制。一般我们为了避免日志丢失的风险，会将这两个参数设置为 1：

- 当 sync_binlog = 1 的时候，表示每次提交事务都会将 binlog cache 里的 binlog 直接持久到磁盘；
- 当 innodb_flush_log_at_trx_commit = 1 时，表示每次事务提交时，都将缓存在 redo log buffer 里的 redo log 直接持久化到磁盘；

可以看到，如果 sync_binlog 和当 innodb_flush_log_at_trx_commit 都设置为 1，那么在每个事务提交过程中，都会至少调用 2 次刷盘操作，一次是 redo log 刷盘，一次是 binlog 落盘，所以这会成为性能瓶颈。

为什么锁竞争激烈？

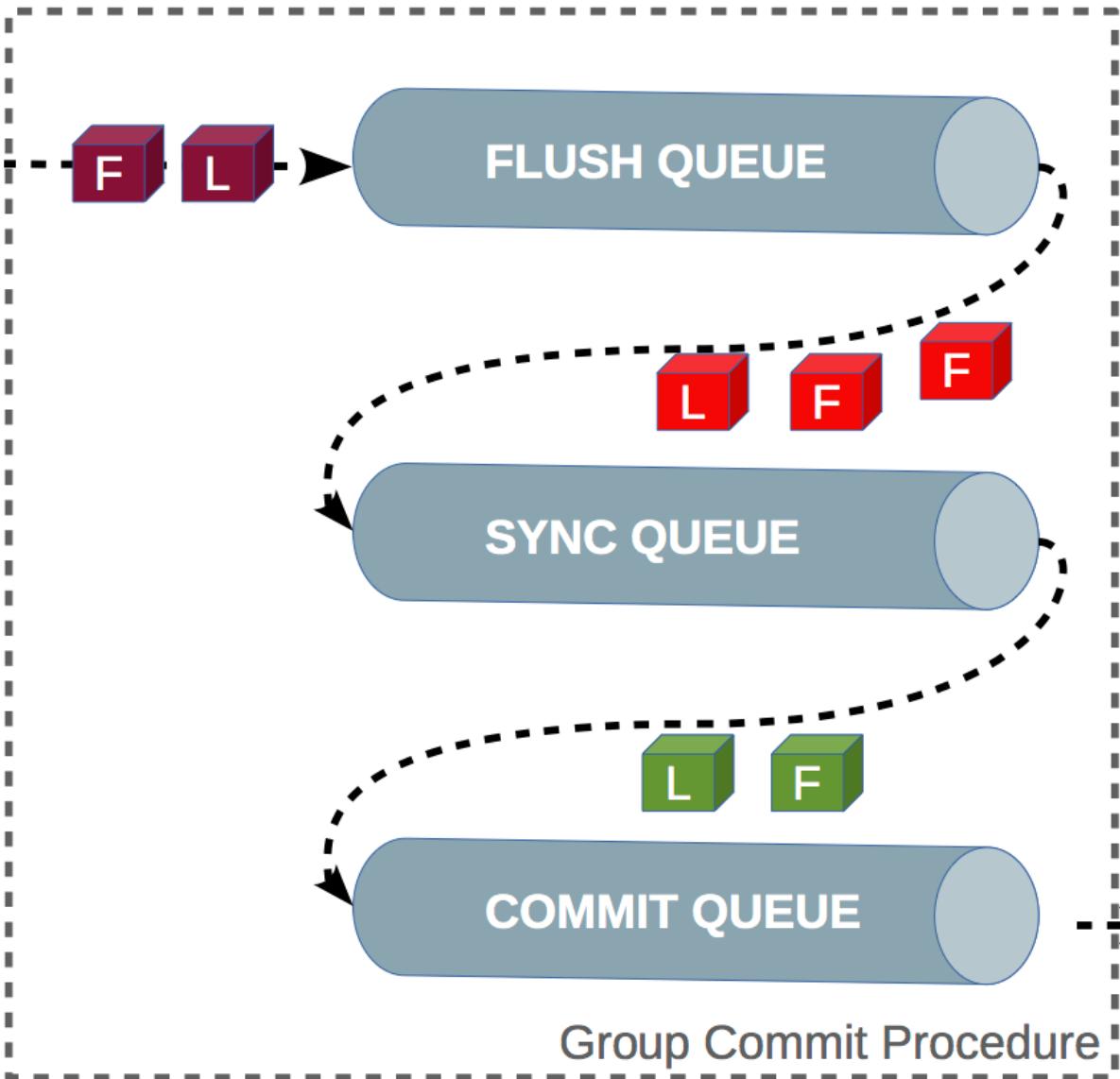
在早期的 MySQL 版本中，通过使用 prepare_commit_mutex 锁来保证事务提交的顺序，在一个事务获取到锁时才能进入 prepare 阶段，一直到 commit 阶段结束才能释放锁，下个事务才可以继续进行 prepare 操作。通过加锁虽然完美地解决了顺序一致性的问题，但在并发量较大的时候，就会导致对锁的争用，性能不佳。

组提交

MySQL 引入了 binlog 组提交（group commit）机制，当有多个事务提交的时候，会将多个 binlog 刷盘操作合并成一个，从而减少磁盘 I/O 的次数，如果说 10 个事务依次排队刷盘的时间成本是 10，那么将这 10 个事务一次性一起刷盘的时间成本则近似于 1。引入了组提交机制后，prepare 阶段不变，只针对 commit 阶段，将 commit 阶段拆分为三个过程：

- **flush 阶段**：多个事务按进入的顺序将 binlog 从 cache 写入文件（不刷盘）；
- **sync 阶段**：对 binlog 文件做 fsync 操作（多个事务的 binlog 合并一次刷盘）；
- **commit 阶段**：各个事务按顺序做 InnoDB commit 操作；

上面的每个阶段都有一个队列，每个阶段有锁进行保护，因此保证了事务写入的顺序，第一个进入队列的事务会成为 leader，leader 领导所在队列的所有事务，全权负责整队的操作，完成后通知队内其他事务操作结束。



对每个阶段引入了队列后，锁就只针对每个队列进行保护，不再锁住提交事务的整个过程，可以看的出来，**锁粒度减小了，这样就使得多个阶段可以并发执行，从而提升效率。**

有 binlog 组提交，那有 redo log 组提交吗？

这个要看 MySQL 版本，MySQL 5.6 没有 redo log 组提交，MySQL 5.7 有 redo log 组提交。

在 MySQL 5.6 的组提交逻辑中，每个事务各自执行 prepare 阶段，也就是各自将 redo log 刷盘，这样就没办法对 redo log 进行组提交。

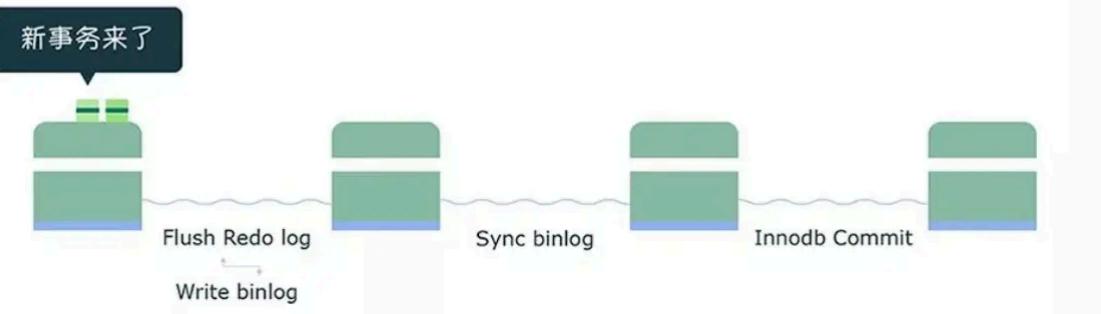
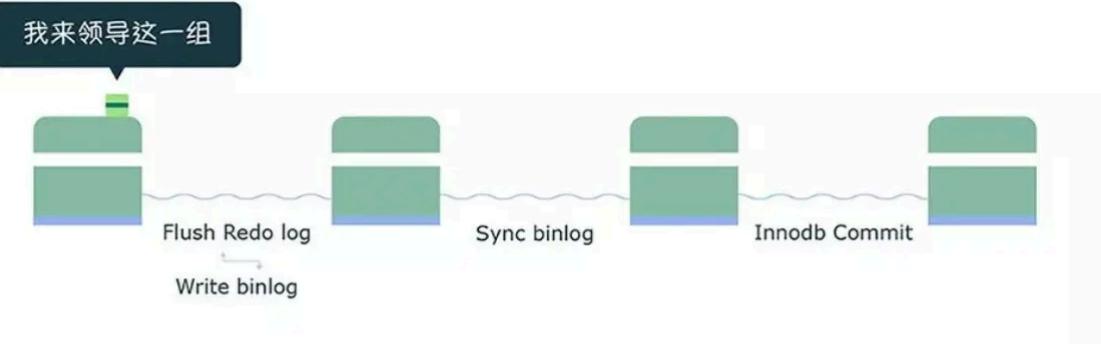
所以在 MySQL 5.7 版本中，做了个改进，在 prepare 阶段不再让事务各自执行 redo log 刷盘操作，而是推迟到组提交的 flush 阶段，也就是说 prepare 阶段融合在了 flush 阶段。

这个优化是将 redo log 的刷盘延迟到了 flush 阶段之中，sync 阶段之前。通过延迟写 redo log 的方式，为 redolog 做了一次组写入，这样 binlog 和 redo log 都进行了优化。

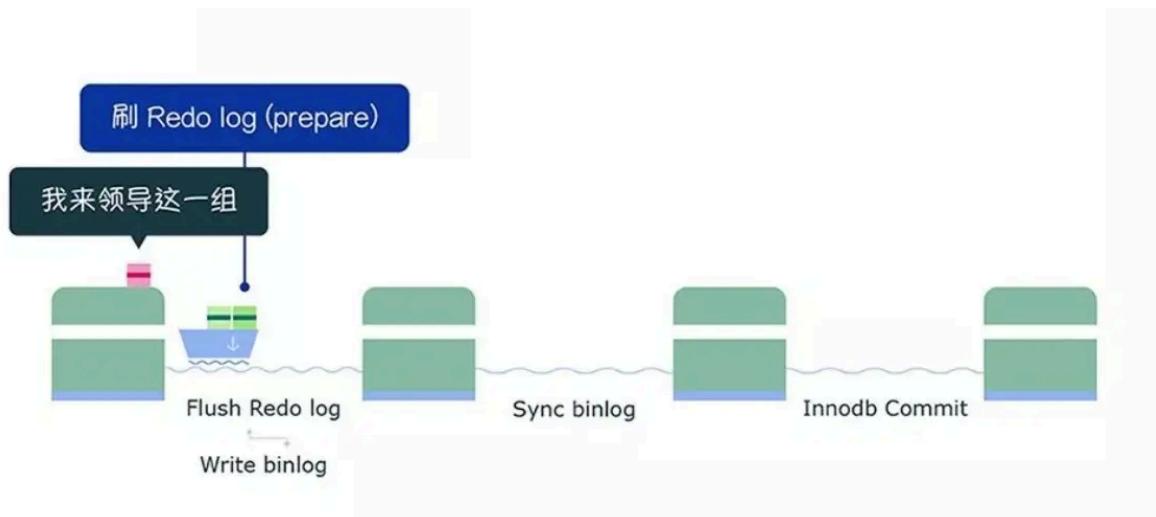
接下来介绍每个阶段的过程，注意下面的过程针对的是“双 1”配置（`sync_binlog` 和 `innodb_flush_log_at_trx_commit` 都配置为 1）。

flush 阶段

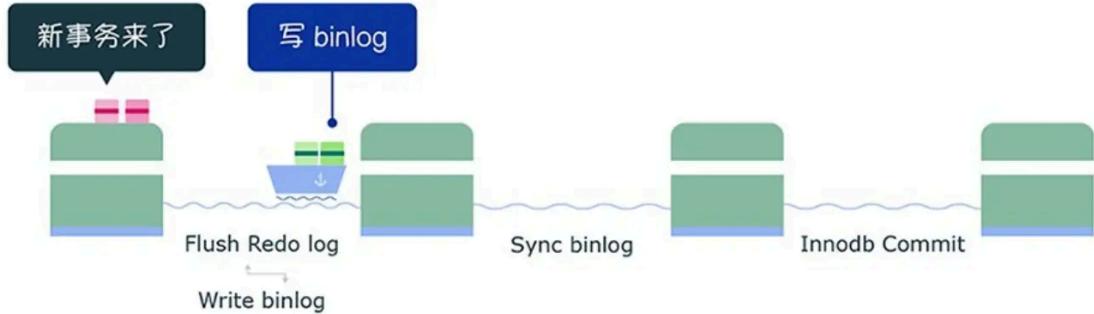
第一个事务会成为 flush 阶段的 Leader，此时后面到来的事务都是 Follower：



接着，获取队列中的事务组，由绿色事务组的 Leader 对 redo log 做一次 write + fsync，即一次将同组事务的 redolog 刷盘：



完成了 prepare 阶段后，将绿色这一组事务执行过程中产生的 binlog 写入 binlog 文件（调用 write，不会调用 fsync，所以不会刷盘，binlog 缓存在操作系统的文件系统中）。



从上面这个过程，可以知道 flush 阶段队列的作用是用于支撑 redo log 的组提交。

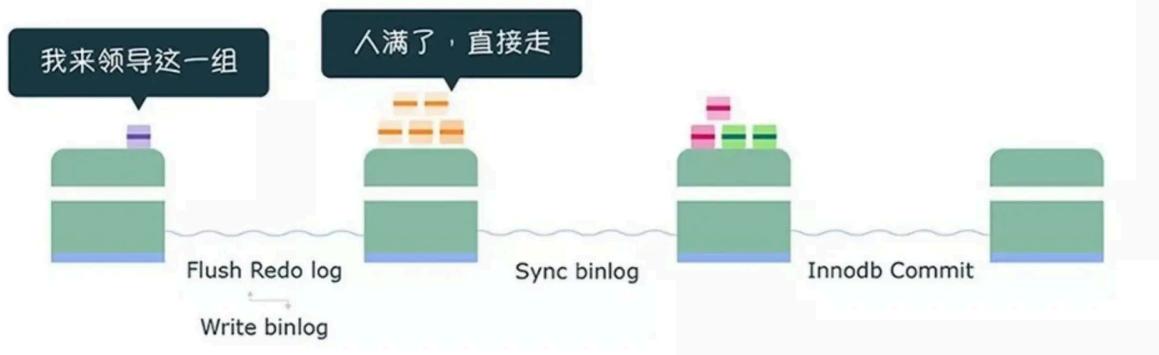
如果在这一步完成后数据库崩溃，由于 binlog 中没有该组事务的记录，所以 MySQL 会在重启后回滚该组事务。

sync 阶段

绿色这一组事务的 binlog 写入到 binlog 文件后，并不会马上执行刷盘的操作，而是会等待一段时间，这个等待的时长由 `Binlog_group_commit_sync_delay` 参数控制，目的是为了组合更多事务的 binlog，然后再一起刷盘，如下过程：



不过，在等待的过程中，如果事务的数量提前达到了 `binlog_group_commit_sync_no_delay_count` 参数设置的值，就不用继续等待了，就马上将 binlog 刷盘，如下图：



从上面的过程，可以知道 sync 阶段队列的作用是**用于支持 binlog 的组提交**。

如果想提升 binlog 组提交的效果，可以通过设置下面这两个参数来实现：

- `binlog_group_commit_sync_delay= N`，表示在等待 N 微妙后，直接调用 fsync，将处于文件系统中 page cache 中的 binlog 刷盘，也就是将「binlog 文件」持久化到磁盘。
- `binlog_group_commit_sync_no_delay_count = N`，表示如果队列中的事务数达到 N 个，就忽视`binlog_group_commit_sync_delay`的设置，直接调用 fsync，将处于文件系统中 page cache 中的 binlog 刷盘。

如果在这一步完成后数据库崩溃，由于 binlog 中已经有了事务记录，MySQL会在重启后通过 redo log 刷盘的数据继续进行事务的提交。

commit 阶段

最后进入 commit 阶段，调用引擎的提交事务接口，将 redo log 状态设置为 commit。



commit 阶段队列的作用是承接 sync 阶段的事务，完成最后的引擎提交，使得 sync 可以尽早的处理下一组事务，最大化组提交的效率。

三阶段提交 (3PC)

三阶段提交协议在两阶段提交的基础上增加了一个阶段，以减少阻塞问题。三阶段提交分为三个阶段：准备阶段、预提交阶段和提交阶段。

阶段 1：准备阶段 (CanCommit Phase)

1. **协调者发送准备请求：**协调者向所有参与者发送一个准备请求，询问他们是否可以准备提交事务。
2. **参与者响应：**每个参与者在接收到准备请求后，检查是否能够提交事务，并返回准备好的响应或失败响应给协调者。

阶段 2：预提交阶段 (PreCommit Phase)

1. **协调者发送预提交请求：**如果所有参与者都返回准备好的响应，协调者发送预提交请求；否则，发送中止请求。

2. **参与者响应**: 参与者在接收到预提交请求后, 执行事务操作但不提交, 并将操作结果记录到日志中。然后, 参与者返回预提交响应给协调者。

阶段 3: 提交阶段 (DoCommit Phase)

1. **协调者决定**: 协调者根据所有参与者的预提交响应做出最终决定。如果所有参与者都返回预提交响应, 协调者发送提交请求; 否则, 发送回滚请求。
2. **参与者执行**: 参与者根据协调者的指示, 执行提交或回滚操作, 并将结果通知协调者。
3. **协调者完成**: 协调者在收到所有参与者的结果后, 完成事务。

优点和缺点

- **优点**: 减少了阻塞问题, 通过增加预提交阶段, 使得在协调者崩溃时, 参与者可以根据日志独立决定事务的提交或回滚。
- **缺点**: 增加了协议的复杂性和开销。

总结

- **两阶段提交 (2PC)** : 通过准备阶段和提交阶段确保分布式事务的一致性, 但存在阻塞问题。
- **三阶段提交 (3PC)** : 在两阶段提交的基础上增加预提交阶段, 减少阻塞问题, 但协议更加复杂。

MySQL 磁盘 I/O 很高, 有什么优化的方法?

现在我们知道事务在提交的时候, 需要将 binlog 和 redo log 持久化到磁盘, 那么如果出现 MySQL 磁盘 I/O 很高的现象, 我们可以控制“延迟” binlog 和 redo log 刷盘的时机, 从而降低磁盘 I/O 的频率。

事务及事务的特性

mysql中事务的操作语句：

1. 开始： `START TRANSACTION;` 或 `BEGIN;`
2. 执行事务的操作
3. 提交： `COMMIT;`
4. 回滚： `ROLLBACK;`

事务：要么全部执行成功，要么全部执行失败。有数据库操作执行完成后，才提交事务，对于已经提交的事务来说，该事务对数据库所做的修改将永久生效，如果中途发生发生中断或错误，那么该事务期间对数据库所做的修改将会被回滚到没执行该事务之前的状态。

事务是由 MySQL 的引擎来实现的，InnoDB 引擎是支持事务的，MyISAM 引擎就不支持事务。

原子性 (Atomicity)

通过undo log (回滚日志) 保证。

一个事务中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节，而且事务在执行过程中发生错误，会被回滚到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性 (Consistency)

通过原子性、隔离性和持久性保证。

是指**事务操作前和操作后，数据满足完整性约束，数据库保持一致性状态**。比如，用户 A 和用户 B 在银行分别有 800 元和 600 元，总共 1400 元，用户 A 给用户 B 转账 200 元。一致性就是要求上述步骤操作后，最后的结果是用户 A 还有 600 元，用户 B 有 800 元，总共 1400 元。

隔离性 (Isolation)

通过**MVCC (多版本并发控制)** 和锁保证。

数据库允许多个并发事务同时对其数据进行读写和修改的能力，**隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致**，因为多个事务同时使用相同的数据时，不会相互干扰，每个事务都有一个完整的数据空间，对其他并发事务是隔离的。

持久性 (Durability)

通过**redo log**保证。

事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

Redis事务的原子性

通过 `MULTI` 命令显式地表示开启一个事务，随后的命令将排队缓存，并不会实际执行。也可通过 `DISCARD` 丢弃第二步中保存在队列中的命令。调用 `EXEC` 时，即可安排队列命令执行。

- 命令入队时就报错，会放弃事务执行，保证原子性；
- 命令入队时没报错，实际执行时报错，不保证原子性；
- `EXEC` 命令执行时实例故障，如果开启了 AOF 日志，可以保证原子性。

并行事务会引发什么问题？

在同时处理多个事务的时候，就可能出现脏读（dirty read）、不可重复读（non-repeatable read）、幻读（phantom read）的问题，破坏事务一致性。

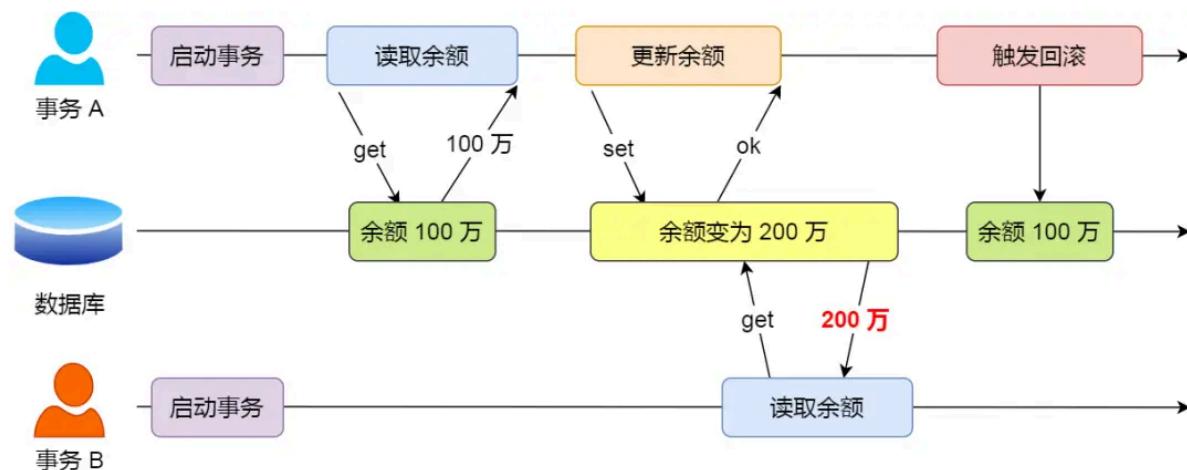
脏读和不可重复读被认为比幻读更严重，主要因为它直接导致数据不一致性和业务逻辑错误，而幻读主要影响范围查询的结果完整性。

这三个现象的严重性排序如下：



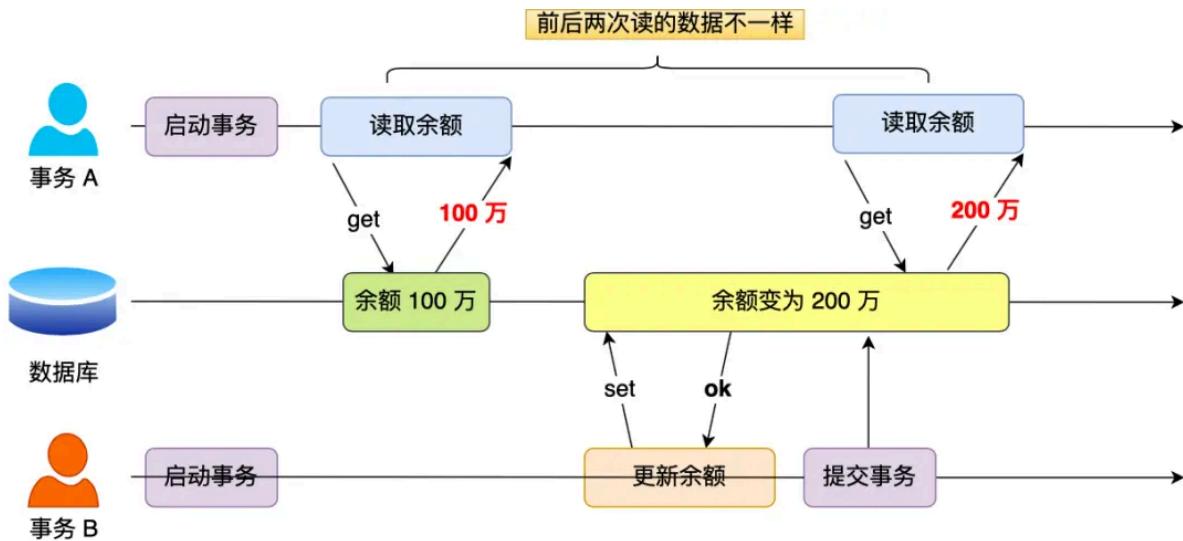
脏读

如果一个事务「读到」了另一个「未提交事务修改过的数据」，就意味着发生了「脏读」现象。下图中，因为事务 A 是还没提交事务的，也就是它随时可能发生回滚操作，如果在上面这种情况事务 A 发生了回滚，那么事务 B 刚才得到的数据就是过期的数据，这种现象就被称为脏读。



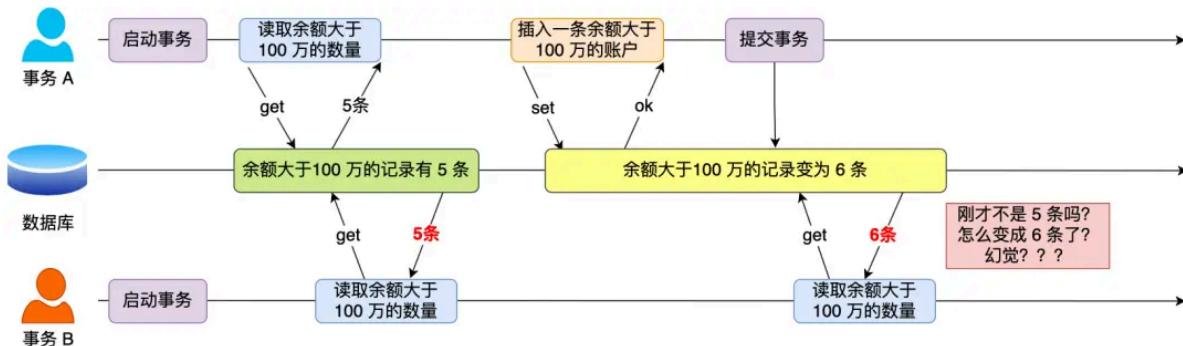
不可重复读

在一个事务内多次读取同一个数据，如果出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象（也就是在事务过程中，其它事务对数据进行了修改提交）。在这过程中如果事务 B 更新了这条数据，并提交了事务，那么当事务 A 再次读取该数据时，就会发现前后两次读到的数据是不一致的，这种现象就被称为不可重复读。



幻读

在一个事务内多次查询某个符合查询条件的「记录数量」，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象（也就是在事务执行过程中，其它事务对数据进行了增加或删除）。



事务的隔离级别有哪些？

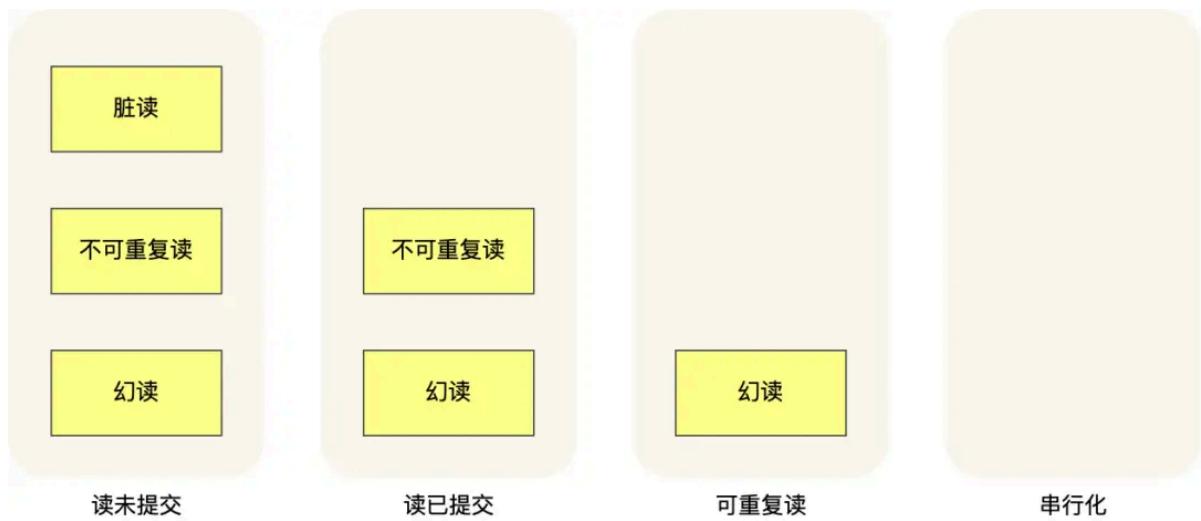
SQL 标准提出了四种隔离级别来规避这些现象，隔离级别越高，性能效率就越低，这四个隔离级别如下（有的数据库只实现了其中几种）：

- **未提交读 (read uncommitted)**，指一个事务还没提交时，它做的变更就能被其他事务看到；直接读取最新的数据
- **提交读 (read committed)**，指一个事务提交之后，它做的变更才能被其他事务看到；通过在每个语句执行前重新生成一个Read View（快照）
- **可重复读 (repeatable read)**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，MySQL InnoDB 引擎的默认隔离级别；通过在启动事务时生成一个Read View，整个事务期间都使用这个Read View。
- **串行化 (serializable)**；会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行；

按隔离水平高低排序如下：



针对不同的隔离级别，并发事务时可能发生的现象也会不同。

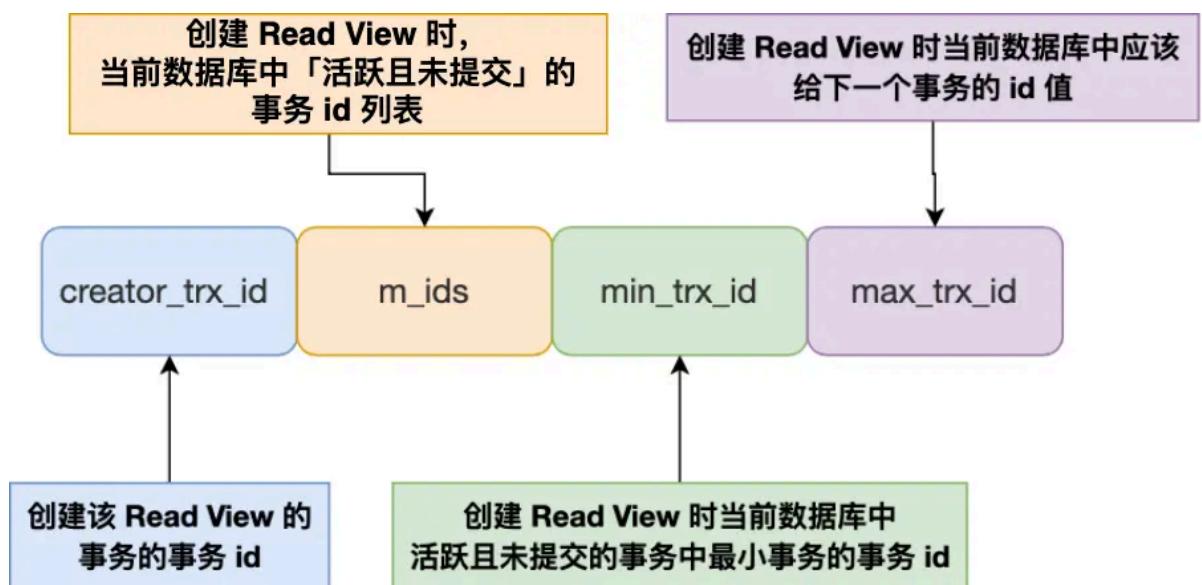


MySQL 在「可重复读」隔离级别下，可以很大程度上避免幻读现象的发生（注意是很大程度避免，并不是彻底避免），所以 MySQL 并不会使用「串行化」隔离级别来避免幻读现象的发生，因为使用「串行化」隔离级别会影响性能：

- 针对快照读（普通 select 语句），是通过 MVCC 方式解决了幻读，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好了避免幻读问题。
- 针对当前读（select ... for update 等语句），是通过 next-key lock（记录锁+间隙锁）方式解决了幻读，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好了避免幻读问题。

Read View 在 MVCC 里如何工作的？

那 Read View 到底是个什么东西？

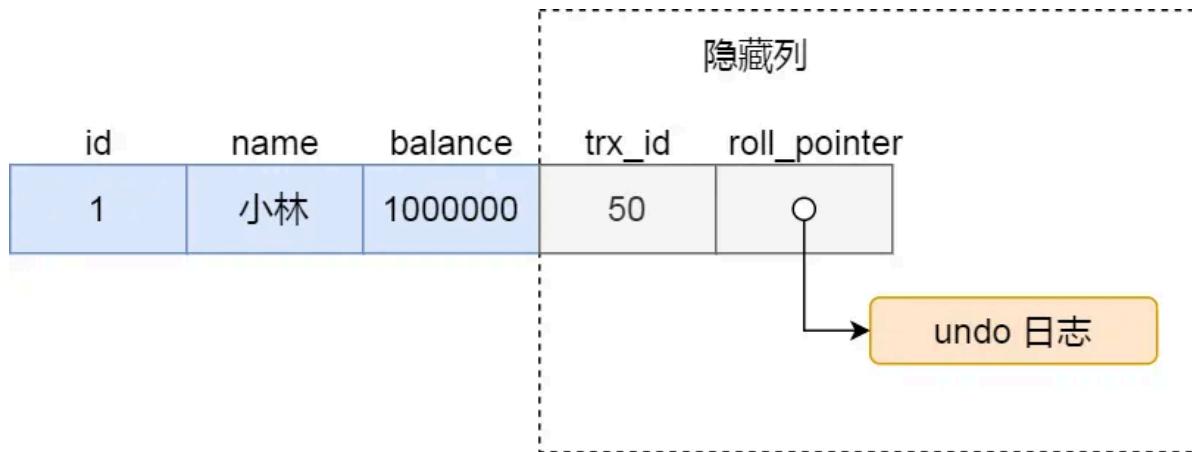


Read View 有四个重要的字段：

- m_ids：指的是在创建 Read View 时，当前数据库中「活跃事务」的事务 id 列表，注意是一个列表，“活跃事务”指的就是，启动了但还没提交的事务。
- min_trx_id：指的是在创建 Read View 时，当前数据库中「活跃事务」中事务 id 最小的事务，也就是 m_ids 的最小值。
- max_trx_id：这个并不是 m_ids 的最大值，而是创建 Read View 时当前数据库中应该给下一个事务的 id 值，也就是全局事务中最大的事务 id 值 + 1；

- `creator_trx_id`：指的是创建该 Read View 的事务的事务 id。

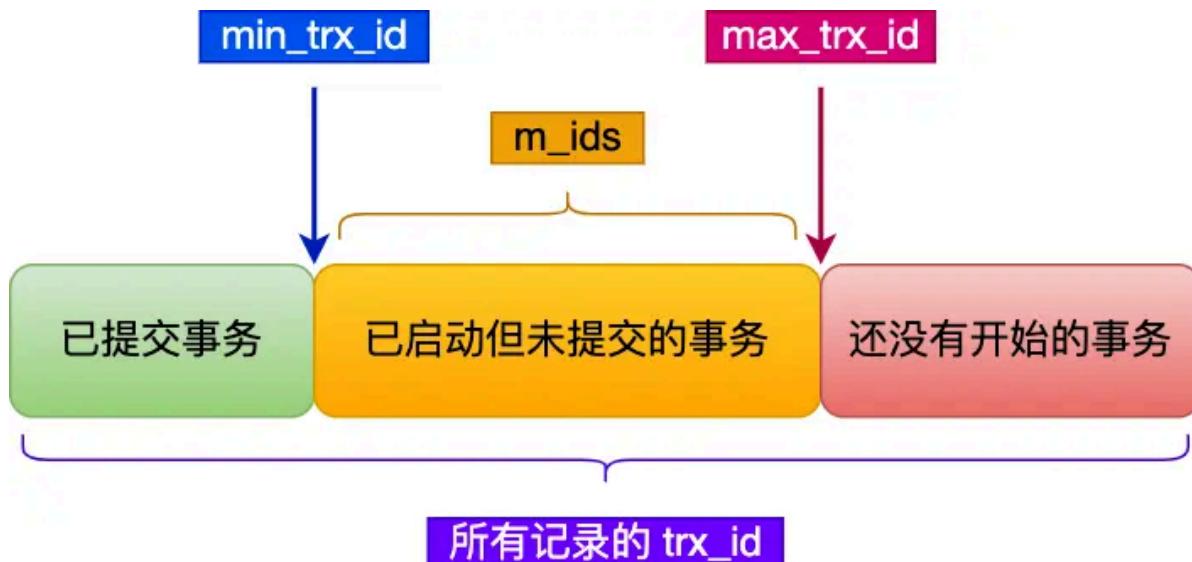
知道了 Read View 的字段，我们还需要了解聚簇索引记录中的两个隐藏列。假设在账户余额表插入一条小林余额为 100 万的记录，然后我把这两个隐藏列也画出来，该记录的整个示意图如下：



对于使用 InnoDB 存储引擎的数据库表，它的聚簇索引记录中都包含下面两个隐藏列：

- `trx_id`，当一个事务对某条聚簇索引记录进行改动时，就会把该事务的事务 id 记录在 `trx_id` 隐藏列里；
- `roll_pointer`，每次对某条聚簇索引记录进行改动时，都会把旧版本的记录写入到 undo 日志中，然后这个隐藏列是个指针，指向每一个旧版本记录，于是就可以通过它找到修改前的记录。

在创建 Read View 后，我们可以将记录中的 `trx_id` 划分这三种情况：



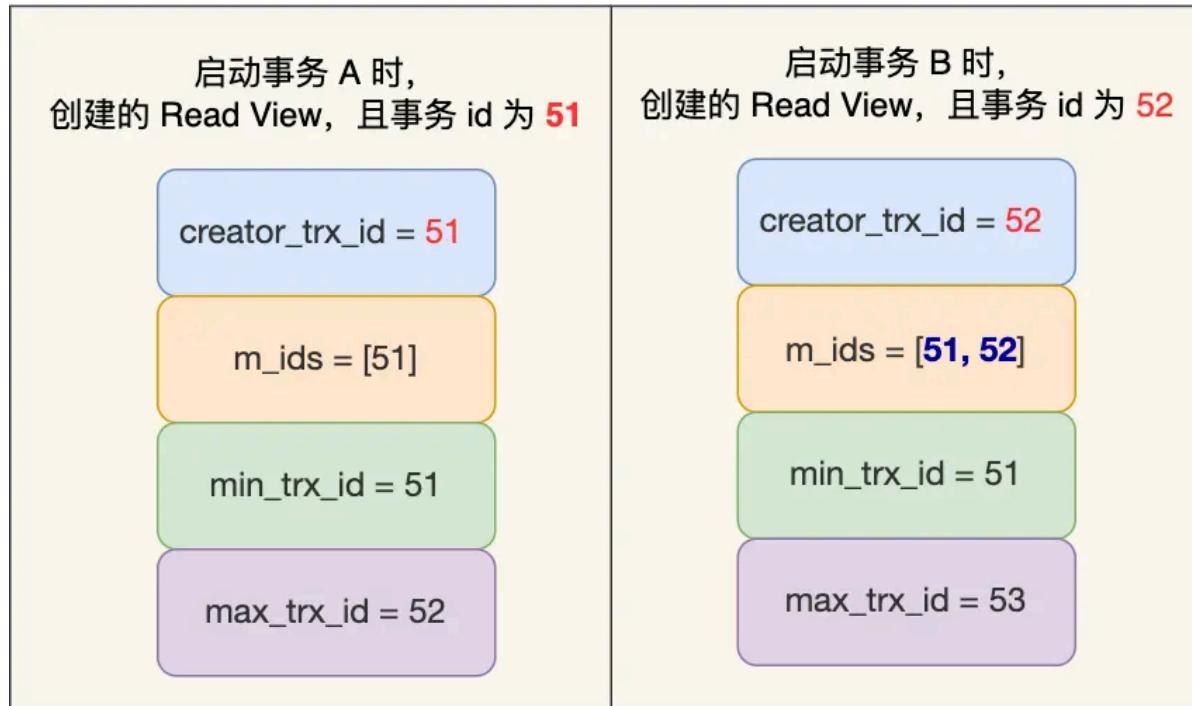
一个事务去访问记录的时候，除了自己的更新记录总是可见之外，还有这几种情况：

- 如果记录的 `trx_id` 值小于 Read View 中的 `min_trx_id` 值，表示这个版本的记录是在创建 Read View 前已经提交的事务生成的，所以该版本的记录对当前事务可见。
- 如果记录的 `trx_id` 值大于等于 Read View 中的 `max_trx_id` 值，表示这个版本的记录是在创建 Read View 后才启动的事务生成的，所以该版本的记录对当前事务不可见。
- 如果记录的 `trx_id` 值在 Read View 的 `min_trx_id` 和 `max_trx_id` 之间，需要判断 `trx_id` 是否在 `m_ids` 列表中：
 - 如果记录的 `trx_id` 在 `m_ids` 列表中，表示生成该版本记录的活跃事务依然活跃着（还没提交事务），所以该版本的记录对当前事务不可见。
 - 如果记录的 `trx_id` 不在 `m_ids` 列表中，表示生成该版本记录的活跃事务已经被提交，所以该版本的记录对当前事务可见。

这种通过「版本链」来控制并发事务访问同一个记录时的行为就叫 MVCC（多版本并发控制）。

可重复读是如何工作的？

可重复读隔离级别是启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View。假设事务 A（事务 id 为 51）启动后，紧接着事务 B（事务 id 为 52）也启动了，那这两个事务创建的 Read View 如下：



记录的字段				
id	name	balance	trx_id	roll_pointer
1	小林	1000000	50	○

事务 A 和事务 B 的 Read View 具体内容如下：

- 在事务 A 的 Read View 中，它的事务 id 是 51，由于它是第一个启动的事务，所以此时活跃事务的事务 id 列表就只有 51，活跃事务的事务 id 列表中最小的事务 id 是事务 A 本身，下一个事务 id 则是 52。
- 在事务 B 的 Read View 中，它的事务 id 是 52，由于事务 A 是活跃的，所以此时活跃事务的事务 id 列表是 51 和 52，**活跃的事务 id 中最小的事务 id 是事务 A**，下一个事务 id 应该是 53。

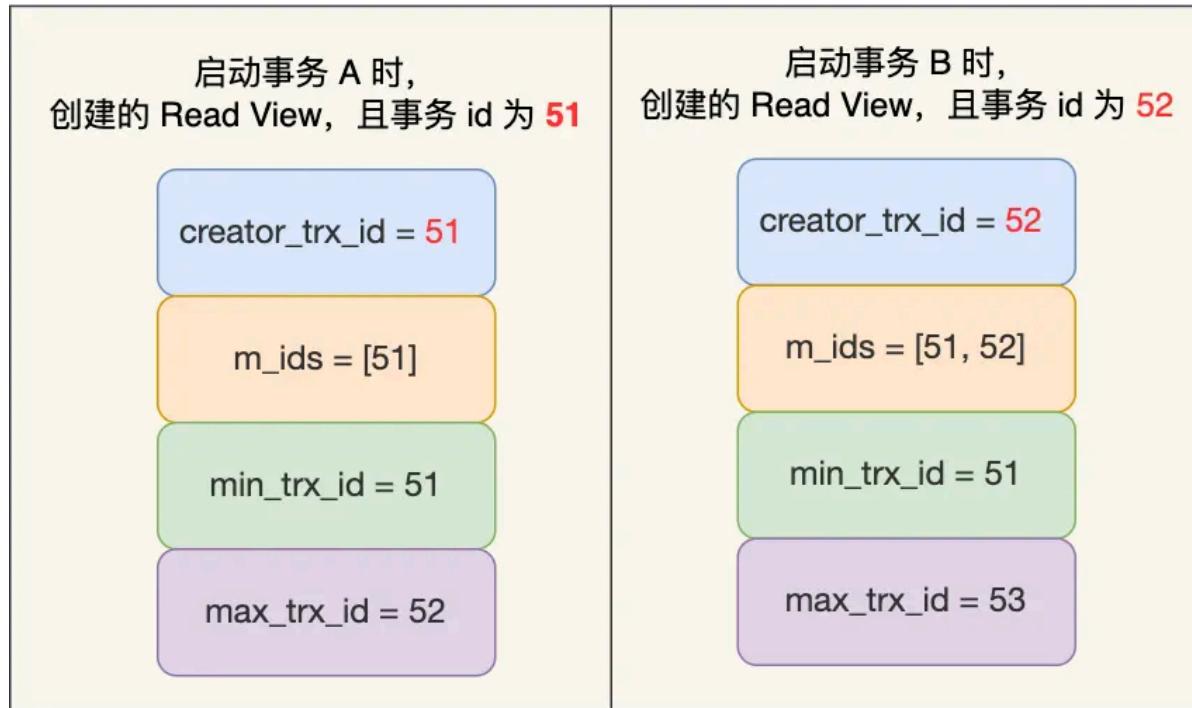
接着，在可重复读隔离级别下，事务 A 和事务 B 按顺序执行了以下操作：

- 事务 B 读取小林的账户余额记录，读到余额是 100 万；
- 事务 A 将小林的账户余额记录修改成 200 万，并没有提交事务；
- 事务 B 读取小林的账户余额记录，读到余额还是 100 万；
- 事务 A 提交事务；

- 事务 B 读取小林的账户余额记录，读到余额依然还是 100 万；

事务 B 第一次读小林的账户余额记录，在找到记录后，它会先看这条记录的 `trx_id`，此时发现 `trx_id` 为 50，比事务 B 的 Read View 中的 `min_trx_id` 值（51）还小，这意味着修改这条记录的事务早就在事务 B 启动前提交过了，所以该版本的记录对事务 B 可见的，也就是事务 B 可以获取到这条记录。

接着，事务 A 通过 update 语句将这条记录修改了（还未提交事务），将小林的余额改成 200 万，这时 MySQL 会记录相应的 undo log，并以链表的方式串联起来，形成版本链，如下图：



你可以在上图的「记录的字段」看到，由于事务 A 修改了该记录，以前的记录就变成旧版本记录了，于是最新记录和旧版本记录通过链表的方式串起来，而且最新记录的 `trx_id` 是事务 A 的事务 id (`trx_id = 51`)。

然后事务 B 第二次去读取该记录，发现这条记录的 `trx_id` 值为 51，在事务 B 的 Read View 的 `min_trx_id` 和 `max_trx_id` 之间，则需要判断 `trx_id` 值是否在 `m_ids` 范围内，判断的结果是在的，那么说明这条记录是被还未提交的事务修改的，这时事务 B 并不会读取这个版本的记录。而是沿着 undo log 链条往下找旧版本的记录，直到找到 `trx_id` 「小于」 事务 B 的 Read View 中的 `min_trx_id` 值的第一条记录，所以事务 B 能读取到的是 `trx_id` 为 50 的记录，也就是小林余额是 100 万的这条记录。

最后，当事物 A 提交事务后，由于隔离级别时「可重复读」，所以事务 B 再次读取记录时，还是基于启动事务时创建的 Read View 来判断当前版本的记录是否可见。所以，即使事物 A 将小林余额修改为 200 万并提交了事务，事务 B 第三次读取记录时，读到的记录都是小林余额是 100 万的这条记录。

就是通过这样的方式实现了，「可重复读」隔离级别下在事务期间读到的记录都是事务启动前的记录。

读提交是如何工作的？

读提交隔离级别是在每次读取数据时，都会生成一个新的 Read View。也意味着，事务期间的多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间另外一个事务修改了该记录，并提交了事务。那读提交隔离级别是怎么工作呢？我们还是以前面的例子来聊聊。

假设事物 A（事务 id 为 51）启动后，紧接着事物 B（事务 id 为 52）也启动了，接着按顺序执行了以下操作：

- 事物 B 读取数据（创建 Read View），小林的账户余额为 100 万；
- 事物 A 修改数据（还没提交事务），将小林的账户余额从 100 万修改成了 200 万；
- 事物 B 读取数据（创建 Read View），小林的账户余额为 100 万；
- 事物 A 提交事务；
- 事物 B 读取数据（创建 Read View），小林的账户余额为 200 万；

那具体怎么做到的呢？我们重点看事物 B 每次读取数据时创建的 Read View。前两次 事物 B 读取数据时创建的 Read View 如下图：

第一步，事务 B 读取数据：

启动事务 B 后，第一次读取数据
创建的 Read View

creator_trx_id = 52

m_ids = [51, 52]

min_trx_id = 51

max_trx_id = 53

第二步，事务 A 修改数据（还未提交）：

记录的字段

id	name	balance	trx_id	roll_pointer
1	小林	2000000	51	○

id	name	balance	trx_id	roll_pointer
1	小林	1000000	50	○

第三步，事务 B 读取数据：

启动事务 B 后，第二次读取数据
创建的 Read View

creator_trx_id = 52

m_ids = [51, 52]

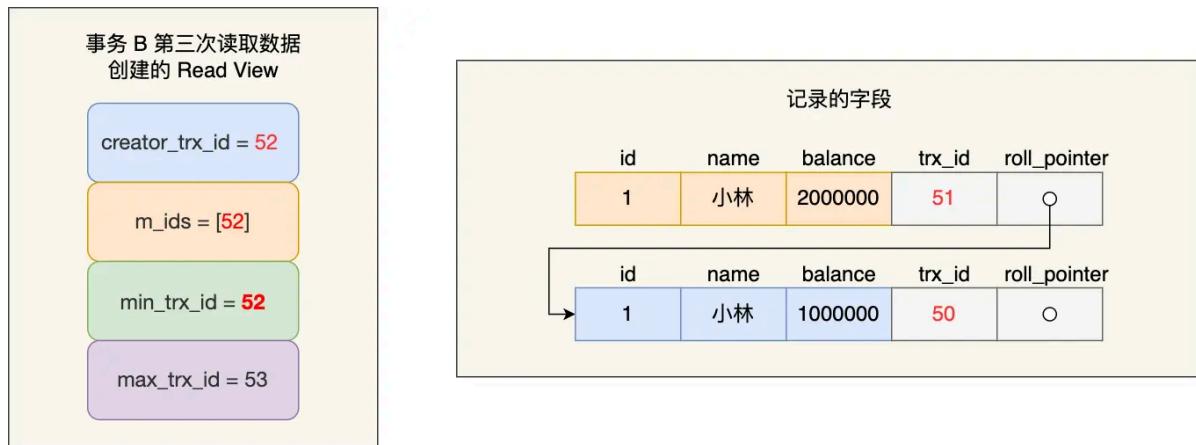
min_trx_id = 51

max_trx_id = 53

我们来分析下为什么事务 B 第二次读数据时，读不到事务 A（还未提交事务）修改的数据？

事务 B 在找到小林这条记录时，会看这条记录的 trx_id 是 51，在事务 B 的 Read View 的 min_trx_id 和 max_trx_id 之间，接下来需要判断 trx_id 值是否在 m_ids 范围内，判断的结果是在的，那么说明**这条记录是被还未提交的事务修改的，这时事务 B 并不会读取这个版本的记录**。而是，沿着 undo log 链条往下找旧版本的记录，直到找到 trx_id 「小于」 事务 B 的 Read View 中的 min_trx_id 值的第一条记录，所以事务 B 能读取到的是 trx_id 为 50 的记录，也就是小林余额是 100 万的这条记录。

在事务 A 提交后，**由于隔离级别是「读提交」，所以事务 B 在每次读数据的时候，会重新创建 Read View**，此时事务 B 第三次读取数据时创建的 Read View 如下：



事务 B 在找到小林这条记录时，会发现这条记录的 trx_id 是 51，比事务 B 的 Read View 中的 min_trx_id 值（52）还小，这意味着修改这条记录的事务早就在创建 Read View 前提交过了，所以该版本的记录对事务 B 是可见的。

正是因为在读提交隔离级别下，事务每次读数据时都重新创建 Read View，那么在事务期间的多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间另外一个事务修改了该记录，并提交了事务。

MySQL当前读是如何避免幻读的？

MySQL 里除了普通查询是快照读，其他都是**当前读**，比如 update、insert、delete，这些语句执行前都会查询最新版本的数据，然后再做进一步的操作。这很好理解，假设你要 update 一个记录，另一个事务已经 delete 这条记录并且提交事务了，这样不是会产生冲突吗，所以 update 的时候肯定要知道最新的数据。

Innodb 引擎为了解决「可重复读」隔离级别使用「当前读」而造成的幻读问题，就引出了间隙锁。假设，表中有一个范围 id 为 (3, 5) 间隙锁，那么其他事务就无法插入 id = 4 这条记录了，这样就有效的防止幻读现象的发生。

间隙锁: (3, 5)

id 列:	1	3	5	6
name 列:	路飞	索隆	乌索普	山治

举个具体例子，场景如下：

事务A	事务B
<pre>begin; select name from t_stu where id > 2 for update</pre>	
	<pre>begin; insert into t_stu values(5,"小飞", 100); 阻塞!</pre>

事务 A 执行了这面这条锁定读语句后，就在对表中的记录加上 id 范围为 $(2, +\infty]$ 的 next-key lock (next-key lock 是间隙锁+记录锁的组合)。然后，事务 B 在执行插入语句的时候，判断到插入的位置被事务 A 加了 next-key lock，于是事务 B 会生成一个插入意向锁，同时进入等待状态，直到事务 A 提交了事务。这就避免了由于事务 B 插入新记录而导致事务 A 发生幻读的现象。

幻读被完全解决了吗？

可重复读隔离级别下虽然很大程度上避免了幻读，但是还是没有能完全解决幻读。

第一个发生幻读现象的场景

id	name	score
1	小林	50
2	小明	60
3	小红	70
4	小蓝	80

事务 A 执行查询 $id = 5$ 的记录，此时表中是没有该记录的，所以查询不出来。

```
# 事务 A
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from t_stu where id = 5;
Empty set (0.01 sec)
```

然后事务 B 插入一条 id = 5 的记录，并且提交了事务。

```
# 事务 B
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> insert into t_stu values(5, '小美', 18);
Query OK, 1 row affected (0.00 sec)

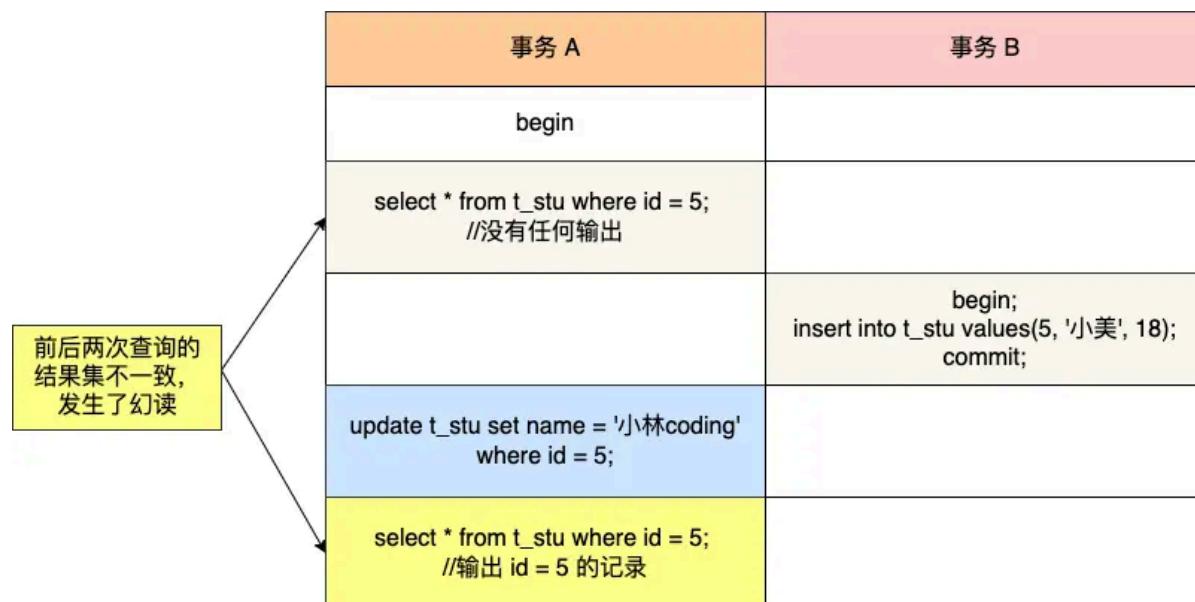
mysql> commit;
Query OK, 0 rows affected (0.00 sec)
```

此时，事务 A 更新 id = 5 这条记录，对没错，事务 A 看不到 id = 5 这条记录，但是他去更新了这条记录，这场景确实很违和，然后再次查询 id = 5 的记录，事务 A 就能看到事务 B 插入的纪录了，幻读就是发生在这种违和的场景。

```
# 事务 A
mysql> update t_stu set name = '小林coding' where id = 5;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from t_stu where id = 5;
+----+-----+-----+
| id | name      | age   |
+----+-----+-----+
| 5  | 小林coding |    18 |
+----+-----+-----+
1 row in set (0.00 sec)
```

整个发生幻读的时序图如下：



在可重复读隔离级别下，事务 A 第一次执行普通的 select 语句时生成了一个 ReadView，之后事务 B 向表中新插入了一条 id = 5 的记录并提交。接着，事务 A 对 id = 5 这条记录进行了更新操作，在这个时刻，这条新记录的 trx_id 隐藏列的值就变成了事务 A 的事务 id，之后事务 A 再使用普通 select 语句去查询这条记录时就可以看到这条记录了，于是就发生了幻读。

因为这种特殊现象的存在，所以我们认为 MySQL InnoDB 中的 MVCC 并不能完全避免幻读现象。

第二个发生幻读现象的场景

除了上面这一种场景会发生幻读现象之外，还有下面这个场景也会发生幻读现象。

- T1 时刻：事务 A 先执行「快照读语句」：select * from t_test where id > 100 得到了 3 条记录。
- T2 时刻：事务 B 往插入一个 id= 200 的记录并提交；
- T3 时刻：事务 A 再执行「当前读语句」 select * from t_test where id > 100 for update 就会得到 4 条记录，此时也发生了幻读现象。

要避免这类特殊场景下发生幻读的现象的话，就是尽量在开启事务之后，马上执行 select ... for update 这类当前读的语句，因为它会对记录加 next-key lock，从而避免其他事务插入一条新记录。

为什么使用主从复制、读写分离

主从复制、读写分离一般是一起使用的，为了提高数据库的并发性能。比如我们使用一台mysql作为master负责写操作，另外两台slave负责读操作。如果是单机部署的mysql，随着并发量的增多，I/O频率过高，会导致性能下降。

主从复制的原理

SQL语句的分类

1. **DDL (Data Definition Languages) 语句**: 数据定义语言，这些语句定义了不同的数据段、数据库、表、列、索引等数据库对象的定义。常用的语句关键字主要包括 create、drop、alter 等。
2. **DML (Data Manipulation Language) 语句**: 数据操纵语句，用于添加、删除、更新和查询数据库记录，并检查数据完整性，常用的语句关键字主要包括 insert、delete、update 和 select 等。
3. **DCL (Data Control Language) 语句**: 数据控制语句，用于控制不同数据段直接的许可和访问级别的语句。这些语句定义了数据库、表、字段、用户的访问权限和安全级别。主要的语句关键字包括 grant、revoke 等。

Bin Log、Relay Log、Redo Log和Undo Log

Bin Log: 二进制日志，记录了所有修改数据库数据的语句（INSERT、UPDATE、DELETE）等，以及 DDL 语句，Bin Log 是顺序写入的，可以配置自动过期删除旧的日志文件。

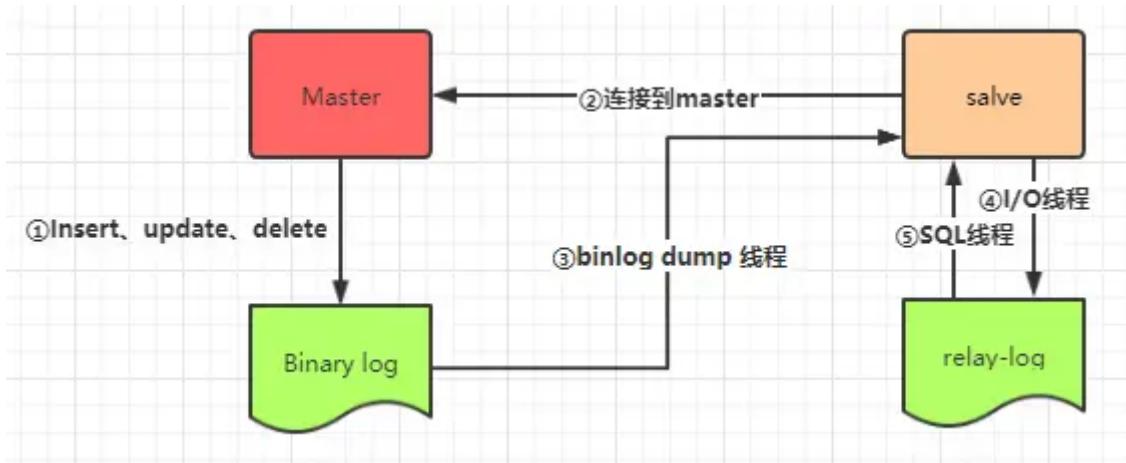
Relay Log: 中继日志，是从服务器（slave）上的一个关键组件，它在主从复制过程中起到了桥梁的作用。中继日志主要用于记录从主服务器接收到的所有二进制日志（binlog）事件，然后从服务器的SQL线程会读取这些记录，应用到本地数据库中以保持与主服务器的数据一致性。

Redo Log: 重做日志，InnoDB 的 Redo Log 用于发生故障时确保事务的持久性，通过先写 redo log，然后异步写入数据到磁盘，InnoDB 可以提高写入性能。redo log 是固定大小的，通常配置为一组文件，循环写入。

Undo Log: 撤销日志，InnoDB 的撤销日志用于存储在事务执行过程中如何撤销已执行的操作，这是事务原子性的关键。如果事务失败或被回滚，撤销日志可以用来撤销进行中的修改。撤销日志不是预设固定大小，会根据需要动态增长，并在事务完成后进行清理。

原理

1. master 节点进行写操作时，会按顺序写入 binlog 中
2. slave 节点连接 master 节点，有多少个 slave，master 就会创建多少个 binlog dump（转存、导出、保存）线程
3. master 节点上的 binlog 发生变化时，binlog dump 线程会通知所有的 slave 节点，并将对应的 binlog 内容推送给 slave 节点
4. I/O 线程收到 binlog 内容后，将内容写入到本地的 relay log
5. SQL 线程读取 relay log，并根据内容对数据库进行操作



读写分离

如何实现master写入数据，slave读取数据呢，可以通过AOP方式，根据方法名去判断连接master还是slave。尽管主从复制、读写分离能很大程度保证MySQL服务的高可用和提高整体性能，但是问题也不少：

- 从机是通过binlog日志从master同步数据的，如果在网络延迟的情况下，从机就会出现数据延迟。那么就有可能出现master写入数据后，slave读取数据不一定能马上读出来。

这个问题可以让同一线程且同一数据库连接，如果有写入操作，读操作均从主库读取，保证数据一致性。

mysql聚合函数

MySQL 提供了一系列强大的聚合函数，用于在查询中对数据进行汇总和统计。聚合函数在 GROUP BY 子句中尤为常用，但也可以在没有 GROUP BY 的情况下使用，来对整个结果集进行汇总。

1. COUNT() 函数用于计算行数。
2. SUM() 函数用于计算列值的总和。
3. AVG() 函数用于计算列值的平均值。
4. MAX() 函数用于查找列中的最大值。
5. MIN() 函数用于查找列中的最小值。
6. GROUP_CONCAT() 函数用于将组中的值连接成一个字符串。

count() 是什么？

count() 是一个聚合函数，函数的参数不仅可以是字段名，也可以是其他任意表达式，该函数作用是统计符合查询条件的记录中，函数指定的参数不为 NULL 的记录有多少个。假设 count() 函数的参数是字段名，如下：

```
select count(name) from t_order;
```

这条语句是统计「t_order 表中，name 字段不为 NULL 的记录」有多少个。也就是说，如果某一条记录中的 name 字段的值为 NULL，则就不会被统计进去。

再来假设 count() 函数的参数是数字 1 这个表达式，如下：

```
select count(1) from t_order;
```

这条语句是统计「t_order 表中，1 这个表达式不为 NULL 的记录」有多少个。1 这个表达式就是单纯数字，它永远都不是 NULL，所以上面这条语句，其实是在统计 t_order 表中有多少个记录。

在通过 count 函数统计有多少个记录时，MySQL 的 server 层会维护一个名叫 count 的变量。server 层会循环向 InnoDB 读取一条记录，如果 count 函数指定的参数不为 NULL，那么就会将变量 count 加 1，直到符合查询的全部记录被读完，就退出循环。最后将 count 变量的值发给客户端。

1. 如果表里只有主键索引，没有二级索引时，那么，InnoDB 循环遍历聚簇索引，将读取到的记录返回给 server 层，然后读取记录中的 id 值，就会 id 值判断是否为 NULL，如果不为 NULL，就将 count 变量加 1。
2. 但是，如果表里有二级索引时，InnoDB 循环遍历的对象就不是聚簇索引，而是二级索引。因为二级索引会占用更少的空间

count(1) 执行过程是怎样的？

如果表里只有主键索引，没有二级索引时。InnoDB 循环遍历聚簇索引（主键索引），将读取到的记录返回给 server 层，但是不会读取记录中的任何字段的值，因为 count 函数的参数是 1，不是字段，所以不需要读取记录中的字段值。

如果表里有二级索引时，InnoDB 循环遍历的对象就二级索引了。

count(*) 执行过程是怎样的？

count(*) 其实等于 count(0)，也就是说，当你使用 count(*) 时，MySQL 会将 * 参数转化为参数 0 来处理。

count(字段) 执行过程是怎样的？

count(字段) 的执行效率相比前面的 count(1)、count(*)、count(主键字段) 执行效率是最差的。

```
// name不是索引，普通字段  
select count(name) from t_order;
```

对于这个查询来说，会采用全表扫描的方式来计数，所以它的执行效率是比较差的。

Result 1								
id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	t_order	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)

InnoDB和MyISAM对于count的区别

对于InnoDB，count 函数需要通过遍历的方式来统计记录个数。但是在 MyISAM 存储引擎里，执行 count 函数的方式是不一样的，通常在没有任何查询条件下的 count(*)，MyISAM 的查询速度要明显快于 InnoDB。使用 MyISAM 引擎时，执行 count 函数只需要 O(1) 复杂度，这是因为每张 MyISAM 的数据表都有一个 meta 信息有存储了 row_count 值，由表级锁保证一致性，所以直接读取 row_count 值就是 count 函数的执行结果。而 InnoDB 存储引擎是支持事务的，同一个时刻的多个查询，由于多版本并发控制（MVCC）的原因，InnoDB 表“应该返回多少行”也是不确定的，所以无法像 MyISAM 一样，只维护一个 row_count 变量。而当带上 where 条件语句之后，MyISAM 跟 InnoDB 就没有区别了，它们都需要扫描表来进行记录个数的统计。

如何优化 count(*)？

如果对一张大表经常用 count(*) 来做统计，其实是很不好的。

1. 近似值：如果你的业务对于统计个数不需要很精确，可以使用 show table status 或者 explain 命令来表进行估算。

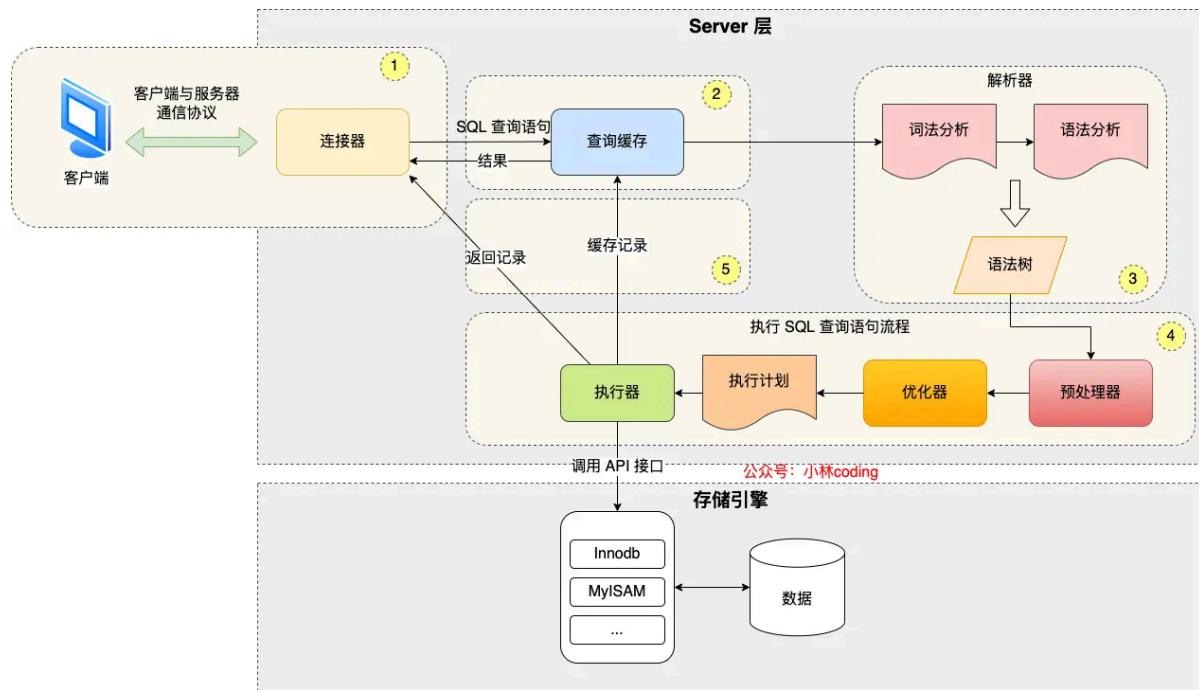
The screenshot shows the MySQL Workbench interface with the following details:

- Connection: 127.0.0.1
- Schema: test
- Query: `explain select count(*) from t_order;`
- Result Type: Result 1
- Table Headers:
 - able
 - partitions
 - type
 - possible_keys
 - key
 - key_len
 - ref
 - rows
 - filtered
 - Ex
- Table Data:

t_order	(NULL)	index	(NULL)	PRIMARY	4	(NULL)	12707308	100.00	Us
---------	--------	-------	--------	---------	---	--------	----------	--------	----
- Bottom Status Bar:
 - explain select c...
 - 0.004s elapsed

2. 额外表保存计数值:当我们在数据表插入一条记录的同时, 将计数表中的计数字段 + 1。也就是说, 在新增和删除操作时, 我们需要额外维护这个计数表。

MySQL 执行一条语句的大概流程



可以看到，MySQL 的架构共分为两层：**Server 层**和**存储引擎层**，

- **Server 层负责建立连接、分析和执行 SQL。** MySQL 大多数的核心功能模块都在这实现，主要包括**连接器、查询缓存、解析器、预处理器、优化器、执行器等**。另外，所有的内置函数（如日期、时间、数学和加密函数等）和所有跨存储引擎的功能（如存储过程、触发器、视图等。）都在 Server 层实现。
- **存储引擎层负责数据的存储和提取。支持 InnoDB、MyISAM、Memory 等多个存储引擎，不同的存储引擎共用一个 Server 层。** 现在最常用的存储引擎是 InnoDB，从 MySQL 5.5 版本开始，InnoDB 成为了 MySQL 的默认存储引擎。我们常说的**索引数据结构，就是由存储引擎层实现的**，不同的存储引擎支持的索引类型也不相同，比如 InnoDB 支持索引类型是 B+树，且是默认使用，也就是说在数据表中创建的主键索引和二级索引默认使用的是 B+ 树索引。

执行一条 SQL 查询语句，期间发生了什么？

- 连接器：建立连接，管理连接、校验用户身份；
- 查询缓存：查询语句如果命中查询缓存则直接返回，否则继续往下执行。MySQL 8.0 已删除该模块；
- 解析 SQL，通过解析器对 SQL 查询语句进行词法分析、语法分析，然后构建语法树，方便后续模块读取表名、字段、语句类型；
- 执行 SQL：执行 SQL 共有三个阶段：
 - 预处理阶段：检查表或字段是否存在；将 `select *` 中的 `*` 符号扩展为表上的所有列。
 - 优化阶段：基于查询成本的考虑，选择查询成本最小的执行计划；
 - 执行阶段：根据执行计划执行 SQL 查询语句，从存储引擎读取记录，返回给客户端；

第一步：连接器

```
# -h 指定 MySQL 服务得 IP 地址，如果是连接本地的 MySQL 服务，可以不用这个参数；  
# -u 指定用户名，管理员角色名为 root；  
# -p 指定密码，如果命令行中不填写密码（为了密码安全，建议不要在命令行写密码），就需要在交互对话里面输入密码  
mysql -h$ip -u$user -p
```

连接的过程需要先经过 TCP 三次握手，因为 MySQL 是基于 TCP 协议进行传输的。如果 MySQL 服务正常运行，完成 TCP 连接的建立后，连接器就要开始验证你的用户名和密码。如果用户密码都没有问题，连接器就会获取该用户的权限，然后保存起来，后续该用户在此连接里的任何操作，都会基于连接开始时读到的权限进行权限逻辑的判断。所以，如果一个用户已经建立了连接，即使管理员中途修改了该用户的权限，也不会影响已经存在连接的权限。修改完成后，只有再新建的连接才会使用新的权限设置。

如果你想知道当前 MySQL 服务被多少个客户端连接了，你可以执行 `show processlist` 命令进行查看。

```
mysql> show processlist;  
+----+----+----+----+----+----+----+----+  
| Id | User | Host      | db   | Command | Time | State    | Info          |  
+----+----+----+----+----+----+----+----+  
| 6  | root | localhost | NULL | Sleep   | 736  |          | NULL          |  
| 7  | root | localhost | NULL | Query   | 0    | init     | show processlist |  
+----+----+----+----+----+----+----+----+  
2 rows in set (0.00 sec)
```

其中 id 为 6 的用户的 Command 列的状态为 `Sleep`，这意味着该用户连接完 MySQL 服务就没有再执行过任何命令，也就是说这是一个空闲的连接，并且空闲的时长是 736 秒（Time 列）。MySQL 定义了 **空闲连接的最大空闲时长，由 `wait_timeout` 参数控制的，默认值是 8 小时（2880秒）**，如果空闲连接超过了这个时间，连接器就会自动将它断开。使用的是 `kill connection + id` 的命令。一个处于空闲状态的连接被服务端主动断开后，这个客户端并不会马上知道，等到客户端在发起下一个请求的时候，才会收到这样的报错“`ERROR 2013 (HY000): Lost connection to MySQL server during query`”。

MySQL 服务支持的最大连接数由 `max_connections` 参数控制，比如我的 MySQL 服务默认是 151 个，超过这个值，系统就会拒绝接下来的连接请求，并报错提示“`Too many connections`”。MySQL 的连接也跟 HTTP 一样，有短连接和长连接的概念，使用长连接的好处就是可以减少建立连接和断开连接的过程，所以一般是推荐使用长连接。但是，如果长连接累计很多，将导致 MySQL 服务占用内存太大，有可能会被系统强制杀掉，这样会发生 MySQL 服务异常重启的现象。

- 第一种，**定期断开长连接**。既然断开连接后就会释放连接占用的内存资源，那么我们可以定期断开长连接。
- 第二种，**客户端主动重置连接**。MySQL 5.7 版本实现了 `mysql_reset_connection()` 函数的接口，注意这是接口函数不是命令，那么当客户端执行了一个很大的操作后，在代码里调用 `mysql_reset_connection` 函数来重置连接，达到释放内存的效果。这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

第二步：查询缓存

这里说的查询缓存是 server 层的，并不是 InnoDB 存储引擎中的 buffer pool。

如果 SQL 是查询语句（select 语句），MySQL 就会先去查询缓存（Query Cache）里查找缓存数据，看看之前有没有执行过这一条命令，这个查询缓存是以 key-value 形式保存在内存中的，key 为 SQL 查询语句，value 为 SQL 语句查询的结果。如果查询的语句命中查询缓存，那么就会直接返回 value 给客户端。如果查询的语句没有命中查询缓存中，那么就要往下继续执行，等执行完后，查询的结果就会被存入查询缓存中。

但对于更新比较频繁的表，查询缓存的命中率很低的，因为只要一个表有更新操作，那么这个表的查询缓存就会被清空。所以，MySQL 8.0 版本直接将查询缓存删掉了。

第三步：解析 SQL（解析器）

1. 第一件事情，**词法分析**。MySQL 会根据你输入的字符串识别出关键字出来，例如，SQL语句
`select username from userinfo`，在分析之后，会得到4个Token，其中有2个Keyword，分别为 `select` 和 `from`：
2. 第二件事情，**语法分析**。根据词法分析的结果，语法解析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法，如果没问题就会构建出 SQL 语法树，这样方便后面模块获取 SQL 类型、表名、字段名、where 条件等等。

如果我们输入的 SQL 语句语法不对，就会在解析器这个阶段报错。比如，我下面这条查询语句，把 `from` 写成了 `form`，这时 MySQL 解析器就会给报错。但是注意，表不存在或者字段不存在，并不是在解析器里做的，而是在执行SQL的预处理阶段。

第四步：执行 SQL

预处理器

- 检查 SQL 查询语句中的表或者字段是否存在；
- 将 `select *` 中的 `*` 符号，扩展为表上的所有列；

优化器

优化器主要负责将 SQL 查询语句的执行方案确定下来，比如在表里面有多个索引的时候，优化器会基于查询成本的考虑，来决定选择使用哪个索引。比如如果是覆盖索引直接使用二级索引。

执行器

经历完优化器后，就确定了执行方案，接下来 MySQL 就真正开始执行语句了，这个工作是由「执行器」完成的。在执行的过程中，执行器就会和存储引擎交互了，交互是以记录（比如一行）为单位的，也就是说执行器的查询过程是个while循环，直到执行器收到存储引擎报告查询完毕的信息，退出循环。

索引下推

索引下推能够减少二级索引在查询时的回表操作，提高查询的效率，因为它将 Server 层部分负责的事情，交给存储引擎层去处理了。举一个具体的例子，方便大家理解，这里一张用户表如下，我对 `age` 和 `reward` 字段建立了联合索引 `(age, reward)`：



id	name	age	reward
1	路飞	20	100000
2	索隆	22	100000
3	香吉士	24	100000
4	法尔科	30	50000
5	凯多	49	1999999
6	娜美	18	50000
7	盖特	39	50000
8	弗兰克	36	2000
9	布鲁克	100	2000

现在有下面这条查询语句：

```
select * from t_user where age > 20 and reward = 100000;
```

联合索引当遇到范围查询 (>、<) 就会停止匹配，也就是 **age 字段能用到联合索引，但是 reward 字段则无法利用到索引**。那么，不使用索引下推（MySQL 5.6 之前的版本）时，执行器与存储引擎的执行流程是这样的：

- Server 层首先调用存储引擎的接口定位到满足查询条件的第一条二级索引记录，也就是定位到 age > 20 的第一条记录；
- 存储引擎根据二级索引的 B+ 树快速定位到这条记录后，获取主键值，然后进行回表操作，将完整的记录返回给 Server 层；
- Server 层在判断该记录的 reward 是否等于 100000，如果成立则将其发送给客户端；否则跳过该记录；
- 接着，继续向存储引擎索要下一条记录，存储引擎在二级索引定位到记录后，获取主键值，然后回表操作，将完整的记录返回给 Server 层；
- 如此往复，直到存储引擎把表中的所有记录读完。

可以看到，**没有索引下推的时候，每查询到一条二级索引记录，都要进行回表操作，然后将记录返回给 Server，接着 Server 再判断该记录的 reward 是否等于 100000**。而使用索引下推后，判断记录的 reward 是否等于 100000 的工作交给了存储引擎层，过程如下：

- Server 层首先调用存储引擎的接口定位到满足查询条件的第一条二级索引记录，也就是定位到 age > 20 的第一条记录；
- **存储引擎定位到二级索引后，先不执行回表操作，而是先判断一下该索引中包含的列（reward 列）的条件（reward 是否等于 100000）是否成立。如果条件不成立，则直接跳过该二级索引。如果成立，则执行回表操作，将完成记录返回给 Server 层。**
- Server 层再判断其他的查询条件（本次查询没有其他条件）是否成立，如果成立则将其发送给客户端；否则跳过该记录，然后向存储引擎索要下一条记录。
- 如此往复，直到存储引擎把表中的所有记录读完。

可以看到，使用了索引下推后，虽然 reward 列无法使用到联合索引，但是因为它包含在联合索引 (age, reward) 里，所以直接在存储引擎过滤出满足 reward = 100000 的记录后，才去执行回表操作获取整个记录。相比于没有使用索引下推，节省了很多回表操作。

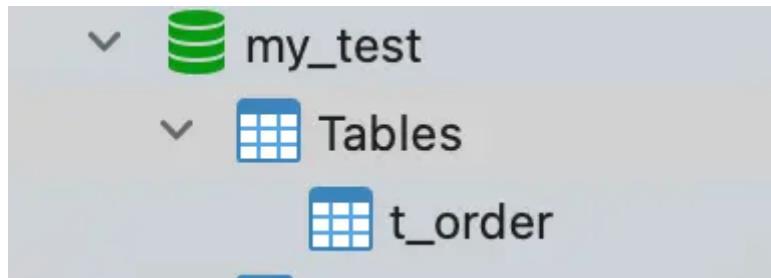
当你发现执行计划里的 `Extr` 部分显示了“Using index condition”，说明使用了索引下推。

The screenshot shows the MySQL EXPLAIN command output for a query. The output includes columns for id, select_type, table, partitions, type, possible_keys, key, key_len, ref, rows, filtered, and Extra. A red box highlights the 'key' column showing 'idx_age_reward' and the 'Extra' column showing 'Using index condition'. Another red box highlights the 'key' column showing 'idx_age_reward' and the note '使用了联合索引查询' (Used joint index query).

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t_user	(NULL)	range	idx_age_reward	idx_age_reward	4	(NULL)	7	11.11	Using index condition 使用了联合索引查询

MySQL 的数据存放在哪个文件？

我们每创建一个 database（数据库）都会在 `/var/lib/mysql/` 目录里面创建一个以 database 为名的目录，然后保存表结构和表数据的文件都会存放在这个目录里。比如，我这里有一个名为 `my_test` 的 database，该 database 里有一张名为 `t_order` 数据库表。



然后，我们进入 `/var/lib/mysql/my_test` 目录，看看里面有什么文件？

```
[root@xiaolin ~]#ls /var/lib/mysql/my_test
db.opt
t_order.frm
t_order.ibd
```

可以看到，共有三个文件，这三个文件分别代表着：

- `db.opt`，用来存储当前数据库的默认字符集和字符校验规则。
- `t_order.frm`，`t_order` 的表结构会保存在这个文件。在 MySQL 中建立一张表都会生成一个`.frm` 文件，该文件是用来保存每个表的元数据信息的，主要包含表结构定义。
- `t_order.ibd`，`t_order` 的表数据会保存在这个文件。表数据既可以存在共享表空间文件（文件名：`ibdata1`）里，也可以存放在独占表空间文件（文件名：表名字`.ibd`）。这个行为是由参数 `innodb_file_per_table` 控制的，若设置了参数 `innodb_file_per_table` 为 1，则会将存储的数据、索引等信息单独存储在一个独占表空间，从 MySQL 5.6.6 版本开始，它的默认值就是 1 了，因此从这个版本之后，MySQL 中每一张表的数据都存放在一个独立的`.ibd` 文件。

COMPACT 行格式长什么样？

先跟 Compact 行格式混个脸熟，它长这样：



可以看到，一条完整的记录分为「记录的额外信息」和「记录的真实数据」两个部分。

MySQL 的 NULL 值是怎么存放的？

MySQL 的 Compact 行格式中会用「NULL值列表」来标记值为 NULL 的列，NULL 值并不会存储在行格式中的真实数据部分。NULL值列表会占用 1 字节空间，当表中所有字段都定义成 NOT NULL，行格式中就不会有 NULL值列表，这样可节省 1 字节的空间。

MySQL 怎么知道 varchar(n) 实际占用数据的大小？

MySQL 的 Compact 行格式中会用「变长字段长度列表」存储变长字段实际占用的数据大小。

VARCHAR(n) 中的 n 表示可以存储的字符数，而不是字节数。存储的实际字符数可以是 0 到 n 个字符。不同的字符集使用不同的字节数来存储字符。例如，utf8 字符集使用 1 到 3 个字节存储一个字符，而 utf8mb4 使用 1 到 4 个字节。VARCHAR 字段还需要额外的 1 或 2 个字节来存储字符串的长度信息。

- 如果最大字符长度 n 小于或等于 255，则使用 1 个字节来存储长度信息。
- 如果最大字符长度 n 大于 255，则使用 2 个字节来存储长度信息。

假设我们有一个 VARCHAR(10) 字段，并使用 utf8 字符集：

- 如果存储的字符串是 "hello"（5 个字符，每个字符 1 个字节），实际存储大小为 5（字符） + 1（长度字节）= 6 个字节。
- 如果存储的字符串是 "你好"（2 个字符，每个字符 3 个字节），实际存储大小为 6（字符） + 1（长度字节）= 7 个字节。

varchar(n) 中 n 最大取值为多少？

一行记录最大能存储 65535 字节的数据（utf8 字符集），但是这个是包含「变长字段字节数列表所占用的字节数」和「NULL值列表所占用的字节数」。所以，我们在算 varchar(n) 中 n 最大值时，需要减去这两个列表所占用的字节数。如果一张表只有一个 varchar(n) 字段，且允许为 NULL，字符集为 ascii。varchar(n) 中 n 最大取值为 65532。计算公式：65535 - 变长字段字节数列表所占用的字节数 - NULL值列表所占用的字节数 = 65535 - 2 - 1 = 65532。如果有多个字段的话，要保证所有字段的长度 + 变长字段字节数列表所占用的字节数 + NULL值列表所占用的字节数 <= 65535。

行溢出后，MySQL 是怎么处理的？

如果一个数据页存不了一条记录，InnoDB 存储引擎会自动将溢出的数据存放到「溢出页」中。

Compact 行格式针对行溢出的处理是这样的：当发生行溢出时，在记录的真实数据处只会保存该列的一部分数据，而把剩余的数据放在「溢出页」中，然后真实数据处用 20 字节存储指向溢出页的地址，从而可以找到剩余数据所在的页。

Compressed 和 Dynamic 这两种格式采用完全的行溢出方式，记录的真实数据处不会存储该列的一部分数据，只存储 20 个字节的指针来指向溢出页。而实际的数据都存储在溢出页中。

关系型和非关系型数据库

关系型数据库 (RDBMS) 的取名来源于所使用的关系模型。关系模型：使用表格形式组织数据，每个表都有行（记录）和列（属性），可以通过SQL进行高效查询。比如MySQL。

非关系数据库 (NoSQL)：不仅仅使用SQL作为查询语言，不遵循传统的关系模型，采用更灵活的数据模型。比如Redis、MongoDB，适合数据量大、高可用、日志系统、地理位置存储的场景。

主要区别

1. **数据模型：**
 - 关系型：表格模型，严格的数据结构。
 - 非关系型：多样化的数据模型，包括键值对、文档等。
2. **查询语言：**
 - 关系型：使用SQL，一个强大且标准化的查询语言。
 - 非关系型：查询方式因类型而异，没有统一标准。
3. **事务处理：**
 - 关系型：支持复杂的事务（一组操作，要么全部成功，要么全部不执行，特点是ACID）管理
和ACID原则（原子性、一致性、隔离性、持久性）。
 - 非关系型：某些类型支持事务，但通常不如关系型数据库强大。
4. **扩展性：**
 - 关系型：通常支持垂直扩展（增加单个服务器的资源）。
 - 非关系型：设计用于水平扩展（增加更多服务器）。
5. **一致性和可靠性：**
 - 关系型：高一致性和可靠性。
 - 非关系型：侧重于可用性和分区容错性，可能采用最终一致性。

关系型数据库适合需要严格数据完整性和复杂查询的应用，而非关系型数据库则适合需要高扩展性和灵活数据模型的大规模应用。

为什么使用索引

1. 唯一性索引保证数据库表中每一行数据的唯一性
2. 大大加快数据检索的速度
3. 加快数据排序，索引本身就是按照一定的顺序存储
4. 在进行连接 (JOIN) 操作或复杂的查询时，索引可以显著提高处理速度，减少数据库的负载

主键和索引

总结：

1. 主键逻辑存在，相当于一本书的页码，不允许重复和NULL
2. 索引物理存在，相当于一本书的目录，可以重复

主键是数据库表中的一个列（或一组列），用于唯一标识表中的每一行记录。主键的要求是：

- 每个表只能有一个主键。
- 主键的值必须是唯一的，不允许重复。

- 主键列不能有空值 (NULL)。

主键的主要目的是确保数据的完整性和唯一性，它是实体完整性的一部分。

索引是数据库表的一个结构，可以帮助快速检索表中的数据。索引可以创建在一个或多个列上，目的是加快查询速度和数据的访问。索引本身并不强制数据的唯一性或完整性（除非特别指定为唯一索引），索引可以有多个。

主键和索引的关系

- **自动索引：**在大多数数据库系统中，当你定义一个列为主键时，系统会自动在该列上创建一个唯一索引。因此**主键一定是唯一性索引，唯一性索引不一定是主键**。
- **功能区别：**主键的主要功能是数据的唯一标识和完整性保障，而索引的主要功能是提高查询效率。

总结来说，**主键确保记录的唯一性和完整性，而索引则主要用于提高数据访问的速度（减少磁盘I/O的次数）**。

Innodb为什么使用自增id作为主键

MySQL是一个关系型数据库管理系统，提供了多种存储引擎，**InnoDB是MySQL中的一种存储引擎，它负责数据的存储、索引和检索等操作**。

如果使用自增id作为主键，每次插入新的记录就会在当前索引节点的后续位置，一页写完就开辟新的页。

如果使用非自增主键（学号等），每次插入新纪录类似随机，会产生大量碎片，索引结构不够紧凑。

MyISAM和InnoDB实现B树索引方式的区别是什么

MyISAM也是MySQL的存储引擎，**MyISAM类型的表强调的是性能，其执行速度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能**。

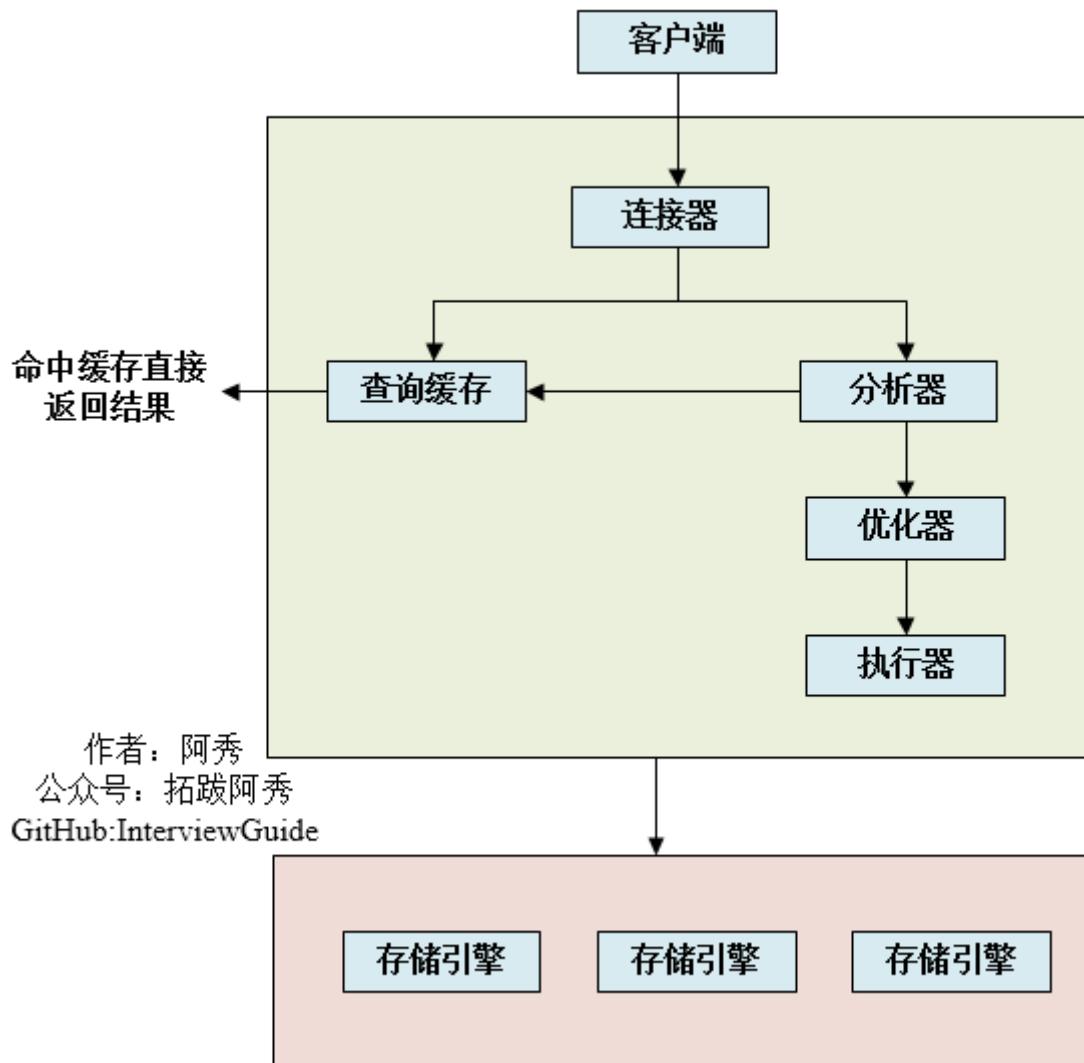
MyISAM

- **非聚集索引：**MyISAM 使用**非聚集索引**。在这种索引结构中，索引和数据是分开存储的。**索引项包含键值和一个指向数据记录的物理指针**。因此，即使找到了索引，还需要通过指针去数据文件中检索实际的数据。
- **全表扫描：**如果没有使用索引，MyISAM 需要进行全表扫描来查找数据。

InnoDB

- **聚集索引：**InnoDB 使用**聚集索引**。在这种结构中，**表数据实际上存储在索引的叶子页中**。每个表都有一个主键索引，且数据按主键的顺序存储。如果表没有显式定义主键，InnoDB 会自动选择一个唯一列作为主键，如果没有唯一列，则会自动生成一个隐藏的行ID作为主键。
- **辅助索引：**对于非主键索引，InnoDB 创建的是**辅助索引**，其叶节点包含相应行的主键值而不是指针。这意味着使用辅助索引时，InnoDB 首先通过辅助索引查找到主键，然后再通过主键索引来访问实际数据。

MySQL如何执行一条SQL的



1. 客户端请求
2. 连接器验证用户身份，基于权限
3. 查询缓存，存在缓存则直接返回
4. 对sql进行词法分析和语法分析
5. 对执行的sql优化选择最优的执行方案
6. 操作引擎，返回结果
7. 存储引擎存储数据，提供读写接口

MySQL的内部构造

可以分为服务层和存储引擎：

1. 服务层：
 1. 连接器
 2. 查询缓存
 3. 分析器
 4. 优化器
 5. 执行器

2. 存储引擎层：插件式

1. innodb：默认
2. myisam
3. memory

Drop、Delete和Truncate的异同点

Drop：

1. 删整个表及其结构，不可逆不可回滚

Delete：

1. 用于删除一行或多行数据，可以搭配where
2. 只删除数据，不删除表结构
3. 事务安全，可以回滚
4. 相对较慢，没删除一行会记录日志

Truncate：

1. 快速清空表中所有数据，保留表结构
2. 比delete快，因为不逐行删除
3. 不可回滚

速度：drop>truncate>delete

MySQL优化，性能优化

1. 为搜索字段创建索引，索引可以提高搜索速度
2. 避免使用select *
3. 垂直分割表，将表中的列拆分为多个表，比如可以将频繁访问的列和不常访问的分开，提高性能
4. 选择正确的存储引擎：MyISAM适合读密集型场景，但不支持事务；InnoDB适合高并发和事务的应用

脏读、幻读、丢弃修改和不可重复读

1. 脏读：一个事务读取了另一个事务未提交的数据，如果那个事务回滚，读取的数据是无效的。
2. 不可重复读：一个事务两次读取同一数据时，数据发生变化，这是因为另一个并发事务在两次读取之间更新了数据。
3. 幻读：和不可重复读类似，但涉及新插入或删除的记录，事务 A 根据某个条件查询数据，事务 B 在此条件下插入或删除了一些记录并提交，当事务 A 重新执行相同的查询时，会发现有额外的“幻影”记录出现或消失。
4. 丢弃修改：比如两个写事务同时某个数据递增，其中一个事务覆盖另一个事务导致结果不正确

脏读关注的是未提交数据的读取，不可重复读关注的是已提交的更新操作，而幻读则是关注于已提交的插入或删除操作。

数据库隔离级别

1. 未提交读：事务发生了修改，即使没有提交，其它事务也可见。可能会导致脏读、幻读或不可重复读。
2. 提交读：一个事务的修改在提交之前都是不可见的。可以阻止脏读、幻读或不可重复读仍可能发生。
3. 重复读：对于一个记录读取多次记录是相同的，可以阻止脏读、不可重复读，幻读仍可能发生。
4. 可串行化读：并发情况下，和串行化的读取结果一致，不会发生脏读、不可重复读以及幻读。

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED 未提交读	√	√	√
READ-COMMITTED 提交读	✗	√	√
REPEATABLE-READ 重复读	✗	✗	√
SERIALIZABLE 可串行化读	✗	✗	✗

InnoDB默认支持的隔离级别是重复读，但这个级别其采用**next-key lock**算法，也可以避免幻读的发生，完全可以达到可串行化的隔离级别。

一般来说隔离级别越低，事务请求的锁就越少，大部分数据库系统的隔离级别是提交读，但InnoDB默认使用重复读，并不会有性能损失。InnoDB在分布式事务的情况下一般会用到可串行化隔离级别。

数据库索引采用B+树的原因

B+树是一种自平衡的树数据结构，主要用于数据库和操作系统的数据存储和索引，是B树的一个变种。

1. B+树由一系列节点构成，分为内部节点和叶子节点，内部节点存储键（key）用于导航，叶子节点存储实际的数据
2. 叶子节点包含全部数据键及其对应的记录指针，节点通过指针串联成一个有序链表。

数据库使用B+树的优点：

1. 高效的搜索性能，B+树是一种多级索引，可以快速定位数据，搜索时间复杂度为O(logn)，即使大规模，查找效率也很高
2. 插入，删除节点，B+树自平衡
3. 数据都在叶子节点，叶子节点通过指针链接，进行范围查找十分高效
4. 减少磁盘I/O操作，叶子节点保持在同一层，存储在磁盘的同一块区域，减少磁盘寻道时间和页的读取次数

视图和游标

视图：是一种虚拟的表，是一个表或多个表的行或列的子集，具有和物理表相同的功能

游标：用于在数据集中逐行遍历数据，使用场景包括处理复杂的逐行逻辑，或者在需要逐条处理或分析数据时进行操作。

MySQL中的事务回滚

回滚机制是通过回滚日志实现的，事务的所有操作都会记录到这个日志中，然后在对应的数据库中进行写入，事务被提交就无法回滚了，因此需要先写日志再写数据库。

InnoDB和MyISAM的区别

InnoDB：MySQL默认存储引擎，聚集索引、辅助索引，支持事务，支持在线热备份，支持外键和行级锁

MyISAM：非聚集型索引，适合读密集型场景，不支持事务，支持压缩表和空间数据索引，只支持表级锁

数据库并发事务的问题

1. 脏读
2. 不可重复读
3. 幻读
4. 丢弃修改

数据库悲观锁和乐观锁的原理和应用场景

1. 悲观锁：先获取锁，再进行业务操作，适合写操作多，冲突频繁的环境，比如银行账户处理，对数据的一致性要求很高
2. 乐观锁：先进行业务操作，实际更新数据时检查数据是否更新过，基于CAS机制通过版本号和时间戳实现，数据版本未变，事务提交成功，数据版本改变，事务回滚。适合读操作多，写操作少的场景，例如社交网络的数据展示等。

MySQL索引主要使用的两种数据结构

1. 哈希索引：使用哈希表，如果绝大多数需求为单条数据查询，选择哈希索引，查询性能最快，其余场景选择B+树索引
2. B+树索引：

数据库为什么要分库、分表

1. 减轻单表负担、优化查询性能，
 1. 垂直分表：将不常用或大型数据字段分离出来，将频繁访问的放在一起
 2. 水平分表：按照某个规则（如ID范围、时间戳、哈希值等）将表中的行分配到多个相同结构的表中。适用于数据行数极大的表，如用户信息表、交易记录表等。

2. 分库：将数据分布到多个数据库，每个数据库可以部署到不同的服务器，通过负载均衡减小单一数据库压力，高可用和容错性

MySQL中的四种索引类型

1. B-Tree：也就是B+树，是MySQL里默认和最常用的索引类型，分为内部节点和叶子节点，内部节点只存储键，从而减小树的高度，减少磁盘I/O操作；叶子节点存储值，叶子节点通过链表链接，可以通过链表顺序访问范围内的所有顺序。B树只能中序遍历所有节点，效率低。
2. HASH：基于哈希表实现，查询速度非常快，但不支持范围查找，只适合单条数据查询。
3. FULLTEXT：全文索引对文本内容进行索引，支持包含某个词语的查询，而不是B-TREE那样基于整个列的比较。适用于CHAR、VARCHAR或TEXT类型的列。
4. R-TREE：空间索引，用的比较少，专门用于处理地理空间数据

视图的作用，可以更改吗

1. 虚拟表，使用和物理表一样
2. 是一个或多个表的列的子集
3. 本身不包含数据，数据仍然存储在基础表中

作用：

1. 安全性：限制用户访问特定数据
2. 简化复杂的SQL操作
3. 隐藏具体的细节

视图主要用于简化检索，保护数据，一般不对视图进行更新；修改视图的定义

```
CREATE VIEW IT_Department AS
SELECT EmployeeID, Name, Salary
FROM Employees
WHERE Department = 'IT';
```

选择MySQL作为数据库存储数据，一天五万条以上的增量，预计运维三年，怎么优化

1. 设计良好的数据库结构，避免join查询，提高效率
2. 选择合适的表字段数据类型，适当添加索引
3. 对表进行划分，垂直分表，水平分表
4. 增加缓存机制
5. 书写高效率的SQL，比如减少使用SELECT*

什么时候需要建立数据库索引

索引主要用于提高查询性能：

1. 频繁查询的列：经常使用where语句、join语句的列
2. 经常需要排序和分组的列
3. 唯一性验证：在需要保证值唯一性的列上创建唯一索引，可以在插入或修改数据时自动检查重复值
4. 外键：作为外键的列上创建索引可以加速联表查询的速度
5. 一些特殊数据类型比如文本字段不宜建立索引

覆盖索引

查询的所有列都包含在索引中，就叫覆盖索引，可以直接从索引本身返回需要的数据，无需访问数据表中的数据行，性能很好。

主键、超键、候选键、外键

1. 主键：能唯一标识一行记录的列或列的组合，主键不能为空，一个表只能有一个主键
2. 超键：能够唯一标识一行记录的列或列的组合，主键也是超键，主键和其它可以唯一标识一行记录的列也可以成为超键
3. 候选键：超键的子集，没有多余的属性，也就是从候选键中移除任何属性，它将不是超键，候选键可以作为主键
4. 外键：一个表中的列是另一个表的主键，用于建立和维护两个表中的关系

5. 班级表 (Classes)

- ClassID (班级编号，主键)
- ClassName (班级名称)

6. 学生表 (Students)

- StudentID (学生编号，主键)
- StudentName (学生姓名)
- ClassID (班级编号，外键)
- Email (学生邮箱):

7. 主键 (Primary Key)

- **班级表**中的 ClassID 是主键，因为它可以唯一标识表中的每一行。
- **学生表**中的 StudentID 是主键，同样用于唯一标识表中的每一行。

8. 超键 (Super Key)

- **班级表**中的任何包含 ClassID 的列集合都是超键，例如{ClassID}, {ClassID, ClassName}。
- **学生表**中的超键包括{StudentID}, {StudentID, StudentName}, {StudentID, ClassID}, {StudentID, Email}，等等，因为这些组合都能唯一标识表中的记录。

9. 候选键 (Candidate Key)

- **班级表**中的候选键只有 ClassID，因为它是最小的超键。
- **学生表**中的候选键为 StudentID 和 Email (假设每个学生的邮箱也是唯一的)。这两个都是最小的超键，即移除任何属性都将无法保持唯一性。

10. 外键 (Foreign Key)

- **学生表**中的 ClassID 是外键，它引用了**班级表**的 ClassID。这样的设计用于表明每个学生属于哪个班级，建立两个表之间的关联。

数据库三大范式

第一范式：

1. 对关系模型的基本要求，不满足第一范式的数据库就不是关系数据库。
2. **每一列都不可再分，无重复的列**
3. 原子性

第二范式：

1. 满足第二范式必须先满足第一范式
2. 要求每一行都能被唯一标识，也就是需要有主键，并且非主键部分依赖于主键
3. 唯一性

第三范式：

1. 满足第三范式必须先满足第二范式
2. 要求一个数据库表中不包含已在其它表中已包含的非主关键字信息
3. 比如班级表中班级编号是主键，还有班级名称列；那么学生表中就只能出现班级编号，不能有班级名称
4. 避免数据冗余

事务的四大特性：A（原子性）C（一致性）I（隔离性）D（持久性）

1. 原子性：事务要么全部成功，要么全部回滚
2. 一致性：事务的执行结果必须从一个正确状态转换到另一个正确状态，事务在执行过程中，即使发生错误或者系统故障，最终数据库的数据和状态必须保持正确和合法
3. 隔离性：多个用户并发访问数据库时，不能被其它事务所干扰，多个并发事务需要相互隔离
4. 持久性：事务一旦被提交了，对数据库中的数据的改变就是永久性的

sql中的NOW()和CURRENT_DATE()两个函数

1. NOW(): 显示年、月、日、小时、分、秒
2. CURRENT_DATE(): 年、月、日

聚集索引、非聚集索引及其区别

通过聚集索引可以直接查到所需要的数据

通过非聚集索引可以查到记录对应的主键值，再使用主键的值查找到需要的数据

聚集索引直接按索引顺序存储数据行，而非聚集索引则是存储键和指向数据行的指针。

InnoDB 使用聚集索引来组织数据。

MyISAM 使用非聚集索引。

MySQL中CHAR和VARCHAR

char长度不可变，用空格填充到指定长度大小，存取速度快，对英文字符（ASCII）占用1个字节，对汉字占用2个字节

varchar长度可变，对英文字符和汉字都占用2个字节

使用索引的注意事项

1. 不要在列上使用函数或运算，这会让索引失效，并进行全表扫描

```
# 使用函数
select * from news where year(publish_time) < 2017
select * from news where publish_time < '2017-01-01'
# 使用运算
select * from news where id / 100 = 1
select * from news where id = 1 * 100
```

2. 避免在where子句使用!=或not in或<>等否定操作符，使用or，这会让索引失效，并进行全表扫描
3. 使用覆盖索引
4. 使用组合索引代替多个单列索引

mysql中有哪些索引

1. 普通索引：仅加速查询
2. 唯一索引：加速查询+列值唯一（可以为null）
3. 主键索引：加速查询+列值唯一+全表只有1个+不可为空
4. 组合索引：多列值组合成一个索引
5. 全文索引：对文本内容进行分词进行搜索
6. 覆盖索引：查询的列都是索引，可以直接从索引中获得数据，不用去访问数据表

为什么不对表中每一列创建一个索引

创建索引和维护索引需要空间和时间，数据量增大会消耗很多性能

索引如何提高查询速度

将无序的数据变成相对有序的数据

使用索引的注意事项

1. 频繁查询的列上使用索引，加快查询速度
2. 在经常排序分组的列上使用索引
3. 避免在where中进行函数和运算
4. 在打算加索引的列设置为NOT NULL，否则将导致引擎放弃使用索引而进行全表扫描
5. 在外键上使用索引
6. 避免过多的使用索引，浪费资源

增加B+树的路数可以降低树的高度，那么无限增加树的路数是不是可以有最优的查找效率

增加B+树的路数（节点的分支树）可以降低树的高度，从理论上减少查找需要经过的节点数量，提高查找效率。

1. 管理更加复杂
2. 节点过大可能导致内存使用效率降低，因为每次节点加载到内存中时，都可能加载大量当前查询不需要的数据。
3. B+树通常用于数据库和文件系统中，其设计考虑了磁盘I/O操作的成本。每次读取节点可能涉及一次磁盘I/O操作。如果节点太大，即使树的高度很低，每次读取节点的成本也会增加，因为每个节点占用的磁盘空间更大。

数据库的表锁和行锁

表锁：不会出现死锁，出现锁冲突几率高，因此并发低。表锁分为表共享读锁和表独占写锁：

1. MyISAM在执行查询语句（SELECT）前，会自动给涉及到的所有表加读锁，读锁不会阻塞其它进程对同一表的读操作，但会阻塞写操作
2. MyISAM在执行增删查改时，会自动给涉及到的所有表加写锁，写锁会阻塞其它进程对同一表的读和写操作

因此MyISAM只适合读密集型应用，因为添加写锁后就不能进行任何操作，如果更新时间很长，会造成其它线程长时间阻塞。

行锁：会出现死锁，发生锁冲突几率低，并发高。MySQL的InnoDB引擎支持行锁，MySQL的行锁通过索引加载，行锁是加在索引响应的行上的，如果sql语句不涉及索引，则会全表扫描，行锁无法实现，实现的是表锁。

1. 行锁必须依靠索引实现
2. 共享锁中，两个事务可以锁同一个索引，排它锁则不能
3. insert、delete和update在事务中默认自动加上排它锁

SQL语法中内连接、自连接、外连接（左、右、全）和交叉连接的区别

内连接 (inner join)：只返回两个表中匹配的行

自连接 (self join)：将表和自身进行连接，通常用于表内部的层次或结构化查询

外连接：

1. 左外连接：返回左表所有行，即使右表没有匹配
2. 右外连接：返回右表所有行，即使左表没有匹配
3. 全外连接：结合左外连接和右外连接的结果，返回两个表中的所有行，匹配行显示匹配值，不匹配的行显示NULL

交叉连接：返回两个表的笛卡尔积，第一个表的每一行和第二个表的每一行的组合

好的，让我们通过具体的例子来解释不同类型的SQL连接。假设我们有两个简单的表：`Employees` 和 `Departments`。

Employees 表：

EmployeeID	Name	DepartmentID
1	Alice	1
2	Bob	1
3	Carol	2
4	David	3

Departments 表：

DepartmentID	DepartmentName
1	HR
2	IT
5	Marketing

1. 内连接 (INNER JOIN)

查询：找出每个员工所属的部门名称。

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

结果：

Name	DepartmentName
Alice	HR
Bob	HR

Name	DepartmentName
Carol	IT

这里只返回匹配的行，David所在的部门（DepartmentID = 3）在 Departments 表中不存在，因此不在结果中。

2. 自连接 (SELF JOIN)

查询：给定一个员工表，找出同一个部门中的员工对。

```
SELECT A.Name AS Employee1, B.Name AS Employee2
FROM Employees A, Employees B
WHERE A.DepartmentID = B.DepartmentID AND A.EmployeeID != B.EmployeeID;
```

结果：

Employee1	Employee2
Alice	Bob
Bob	Alice

3. 外连接 (LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN)

LEFT JOIN:

查询：列出所有员工及其部门名称，包括没有对应部门的员工。

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
LEFT JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

结果：

Name	DepartmentName
Alice	HR
Bob	HR
Carol	IT
David	NULL

RIGHT JOIN 和 FULL OUTER JOIN 在 SQLite 中不直接支持，但可以通过其他方式模拟。在其他数据库（如 SQL Server, PostgreSQL）中，可以直接使用。

4. 交叉连接 (CROSS JOIN)

查询：列出所有可能的员工与部门组合。

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
CROSS JOIN Departments;
```

结果 (部分显示) :

Name	DepartmentName
Alice	HR
Alice	IT
Alice	Marketing
Bob	HR
Bob	IT
Bob	Marketing
Carol	HR
Carol	IT
Carol	Marketing
David	HR
David	IT
David	Marketing

数据库结构优化的手段

1. 拆分表
2. 拆分数据库，读写分离，主库负责写，从库负责读
3. 限定数据范围
4. 范式优化：消除冗余
5. 反范式优化：增加适当冗余（减少join）

拆分数据表

垂直拆分：把表中的列分为多个较小的表，每个表包含原始表中的一部分列，一般将不常用的数据或非常大的数据（文本、图片等）从频繁访问的列中分离出来。

水平拆分：将一个表分成多个较小的表，每个表包含原始表的子集，但结构相同。这种拆分通常基于某个属性，如地理位置、时间范围或其他业务规则。

MySQL为什么不用哈希索引和红黑树

哈希：

1. 不支持范围查询
2. 要处理哈希冲突
3. 只适用于精确匹配

红黑树：

1. **磁盘I/O效率：**红黑树的节点通常只包含极少数键值对（通常是一个）。这意味着在处理大量数据时，树的高度可能较高，导致多次磁盘I/O操作。

- 2. **不适合磁盘存储**: 由于每个节点较小, 红黑树不适合直接映射到磁盘上的存储结构, 这可能导致频繁的磁盘读写操作。

如何保证一致性

1. 通过事务的AID特性保证
2. 通过代码判断是否提交还是回滚

如何保证原子性

任务执行失败, 通过undo log (回滚日志) 回滚

如何保证持久性

1. Redo Log (重做日志)

Redo log 是数据库恢复机制的核心组成部分, 用于记录事务对数据库所做的所有修改。这些日志记录了足够的信息, 以便在数据库重新启动后重放这些操作, 从而恢复数据库到最后一次提交的状态。以下是其工作流程:

- **日志记录**: 当事务进行数据修改时, 数据库不仅在内存中修改数据, 还在 redo log buffer (重做日志缓冲区) 中记录下修改操作。
- **日志写入**: 当事务提交时, 相关的 redo log 必须先被写入到磁盘上的 redo log file 中。这个过程称为“write-ahead logging”(预写日志), 确保即使数据库在事务提交后立即崩溃, 所有的修改也已经安全记录在磁盘上。
- **恢复过程**: 数据库启动时, 会检查 redo log, 根据日志内容重做 (redo) 所有已提交事务的操作, 以确保这些更改不会丢失。

2. Undo Log (回滚日志)

Undo log 记录了事务执行前的数据状态, 使得在事务执行过程中如果需要回滚, 可以利用 undo log 恢复到事务开始前的状态。这不仅对于处理事务失败情况下的数据回滚至关重要, 也在系统崩溃后的恢复过程中发挥作用:

- **记录反操作**: 在执行数据修改操作时, 系统也会在 undo log 中记录相应的“反操作”。
- **事务回滚**: 如果事务需要回滚 (例如, 由于错误或事务逻辑要求), 数据库可以利用 undo log 来撤销已进行的修改, 恢复到事务开始前的状态。
- **崩溃恢复**: 在系统崩溃后, 除了需要 redo log 来重做已提交的事务外, 还可能需要 undo log 来撤销那些未完成的事务。

3. Checkpoint (检查点)

Checkpoint 是另一种确保数据持久性的机制, 通过定期将内存中的数据页刷新到磁盘, 减少了系统恢复时间。检查点不仅减少了在崩溃恢复时需要重做的日志量, 也保证了即使在大量数据在内存中但未写入磁盘的情况下, 数据也能得到恢复。

数据库高并发

1. 缓存
2. 索引
3. 主从读写分离
4. 数据库拆分
5. 分布式架构

怎么判断 SQL 走了索引 (EXPLAIN 关键字)

在 MySQL 中，可以使用 `EXPLAIN` 关键字来分析 SQL 语句的执行计划，从而判断查询是否使用了索引。执行计划中包含了关于如何访问表中数据的详细信息，包括是否使用索引，使用了哪种类型的索引，以及索引的效率等。

例如：

```
EXPLAIN SELECT * FROM users WHERE username = 'example';
```

在返回的结果中，`type` 字段显示了查询的类型，`key` 字段显示了使用的索引。如果 `key` 字段非空，表示查询使用了索引。

MySQL 的基础数据类型

MySQL 支持多种数据类型，主要可以分为以下几类：

- **数值类型**：包括整数类型（如 `INT`, `BIGINT`）、浮点数类型（如 `FLOAT`, `DOUBLE`）和定点数（如 `DECIMAL`）。
- **字符串类型**：包括 `CHAR`, `VARCHAR`, `TEXT`, `BLOB` 等。
- **日期和时间类型**：如 `DATE`, `TIME`, `DATETIME`, `TIMESTAMP` 等。
- **逻辑类型**：`BOOLEAN` 或 `BOOL`（实际上是 `TINYINT` 的别名）。
- **枚举类型**：`ENUM`。
- **集合类型**：`SET`。

MySQL 中的 CHAR 介绍

`CHAR` 是一种固定长度的字符串数据类型，在定义时需要指定长度（例如 `CHAR(5)`）。MySQL 根据定义的长度存储数据，如果存储的字符串短于定义的长度，MySQL 会在字符串后面填充空格以达到指定长度。

使用场景：

- **存储固定长度的数据**：如性别字段（男、女）、状态码（如 "OK", "FAIL"）、邮政编码等。这些字段的长度通常是固定的，使用 `CHAR` 可以获得更好的性能。
- **频繁更新的字段**：`CHAR` 类型由于是固定长度，更新时不会引起数据页的重新整理，这在某些高频更新场景下有性能优势。

总的来说，当数据项的长度固定并且长度较短时，使用 `CHAR` 类型可以提高效率。对于长度变化较大的字段，使用 `VARCHAR` 类型更为合适，因为它会节省空间。

假如我有1000个学生，10门课，学生要选课，我应该怎么设计表；现在每个学生要在教室上课，增加了教室信息应该怎么设计表

初始设计（不含教室信息）

为了设计一个学生选课系统的数据库，可以使用以下三个表：

1. Students (学生表)

- `StudentID` (主键)
- `StudentName`
- `OtherStudentDetails` (如年级、联系信息等)

2. Courses (课程表)

- CourseID (主键)
- CourseName
- OtherCourseDetails (如课程描述、学分等)

3. Enrollments (选课表)

- StudentID (外键, 引用 Students)
- CourseID (外键, 引用 Courses)
- EnrollmentDate

这种设计允许每个学生选择多门课程, 每门课程也可以被多个学生选择, 实现多对多的关系。

增加教室信息

当涉及到在特定教室上课的需求时, 可以增加一个教室表, 并更新选课表以包含教室信息:

1. Classrooms (教室表)

- ClassroomID (主键)
- ClassroomName
- Location
- Capacity

更新 Enrollments 表以包括教室信息:

- ClassroomID (外键, 引用 Classrooms)

更新后的 Enrollments 表:

- StudentID (外键, 引用 Students)
- CourseID (外键, 引用 Courses)
- ClassroomID (外键, 引用 Classrooms)
- EnrollmentDate

作为开发者, 微信中有很多动态图片, 保存起来太大了, 有什么好的解决方案吗?

1. 使用视频格式代替GIF

- **转换为MP4/WebM:** 动态图片可以转换为视频格式, 如MP4或WebM。这些格式通常比GIF有更好的压缩率, 且质量更高。例如, 许多社交平台已经使用视频格式来代替传统的GIF以减少数据量。

2. 使用高效的图片格式

- **WebP:** Google开发的WebP格式提供了比GIF更高的压缩效率, 并支持动画。WebP通常可以在保持相同图片质量的情况下, 大幅减小文件大小。
- **AVIF:** 作为较新的格式, AVIF提供了比WebP更优秀的压缩效果, 但兼容性和支持度可能还不如WebP广泛。

3. 压缩和优化GIF

- **优化工具:** 使用工具如 gifsicle, ImageOptim 或在线服务如 Ezgif.com 来优化GIF文件。这些工具可以减少颜色深度、删除重复帧、应用更高效的压缩算法等。

- **减少帧数和分辨率:** 减少GIF的帧数和调整其分辨率可以显著减小文件大小，但可能会影响动画的流畅性和清晰度。

4. 懒加载和按需加载

- **懒加载:** 只有当动态图片进入视口时才加载，可以减少初次页面加载时的数据传输。
- **按需加载:** 提供低分辨率的预览版，当用户交互时再加载完整的动态图片。

5. CDN和缓存策略

- **使用CDN:** 利用内容分发网络（CDN）可以加速图片的加载速度，减少服务器的压力。
- **缓存策略:** 适当设置HTTP缓存头，使浏览器或客户端缓存图片，减少重复加载的需求。

6. 数据库和文件系统的存储策略

- **分散存储:** 将图片文件存储在不同的服务器或存储系统中，可以根据地理位置或用户活跃度进行智能分配。
- **数据库BLOB存储:** 对于小型的动态图片，可以考虑使用数据库的BLOB字段直接存储。这种方法便于管理，但可能影响数据库的性能。

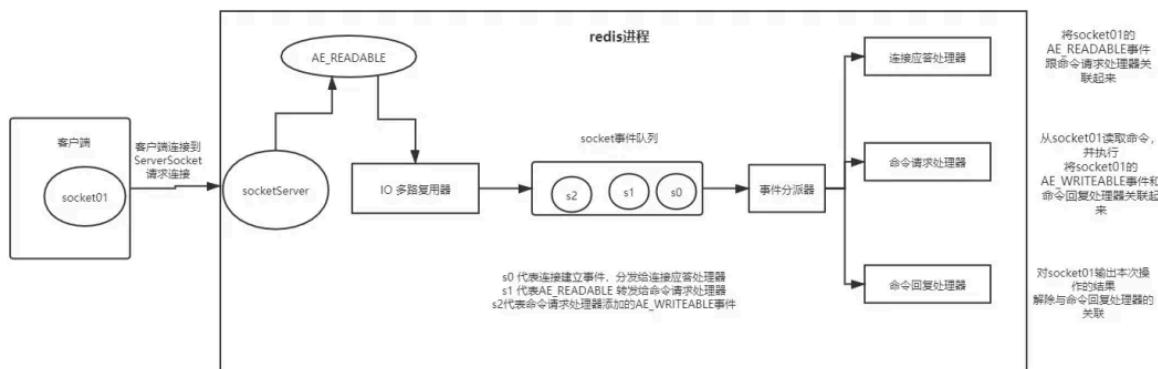
Redis和Memcache的区别

1. redis支持丰富的数据结构（基础数据结构：string、hash、list、set和zset；高级数据结构：位图、位域、消息队列、地理索引、超级日志）
2. memcache只支持key-value的存储
3. redis应用场景更丰富，数据库、消息队列、缓存等；memcache应用场景更局限，用于缓存
4. redis原生支持集群，memcache没有原生的集群模式

redis单线程处理请求

redis采用IO多路复用机制来处理请求，采用reactor IO模型，处理流程如下：

1. 首先接收到客户端的socket请求，多路复用器将socket转给连接应答处理器；
2. 连接应答处理器将AE_READABLE事件与命令请求处理器关联(这里是把socket事件放入一个队列)；
3. 命令请求处理器从socket中读到指令，再内存中执行，并将AE_WRITEABLE事件与命令回复处理器关联；
4. 命令回复处理器将结果返回给socket，并解除关联。



redis 单线程效率高的原因

1. I/O多路复用
2. 纯内存操作效率高
3. 单线程避免了多线程的切换和互斥同步开销

redis key过期策略

对于设置了过期时间的key，redis的删除策略：定期删除+惰性删除。

1. 定期删除：redis默认每100ms随机抽取了一些设置了过期时间的key，如果过期就进行删除。如果redis设置了10万个key都设置了过期时间，每隔几百毫秒就要检查10万个key那CPU负载就很高了，所以redis并不会每隔100ms就检查所有的key，而是随机抽取一些key来检查。
2. 惰性删除：定期删除会导致有些key过期了并没有被删除，因此在获取某个key的时候发现过期了，如果key过期了就删除掉不会返回。

这两个策略结合起来保证过期的key一定会被删除。

redis内存淘汰

redis内存占用过大，就会进行内存淘汰：

1. 新写入操作直接报错
2. LRU (最常用)
3. 随机移除几个key
4. 在设置了过期时间的key中，LRU
5. 在设置了过期时间的key中，随机移除几个

redis主从模式保证高并发和高可用

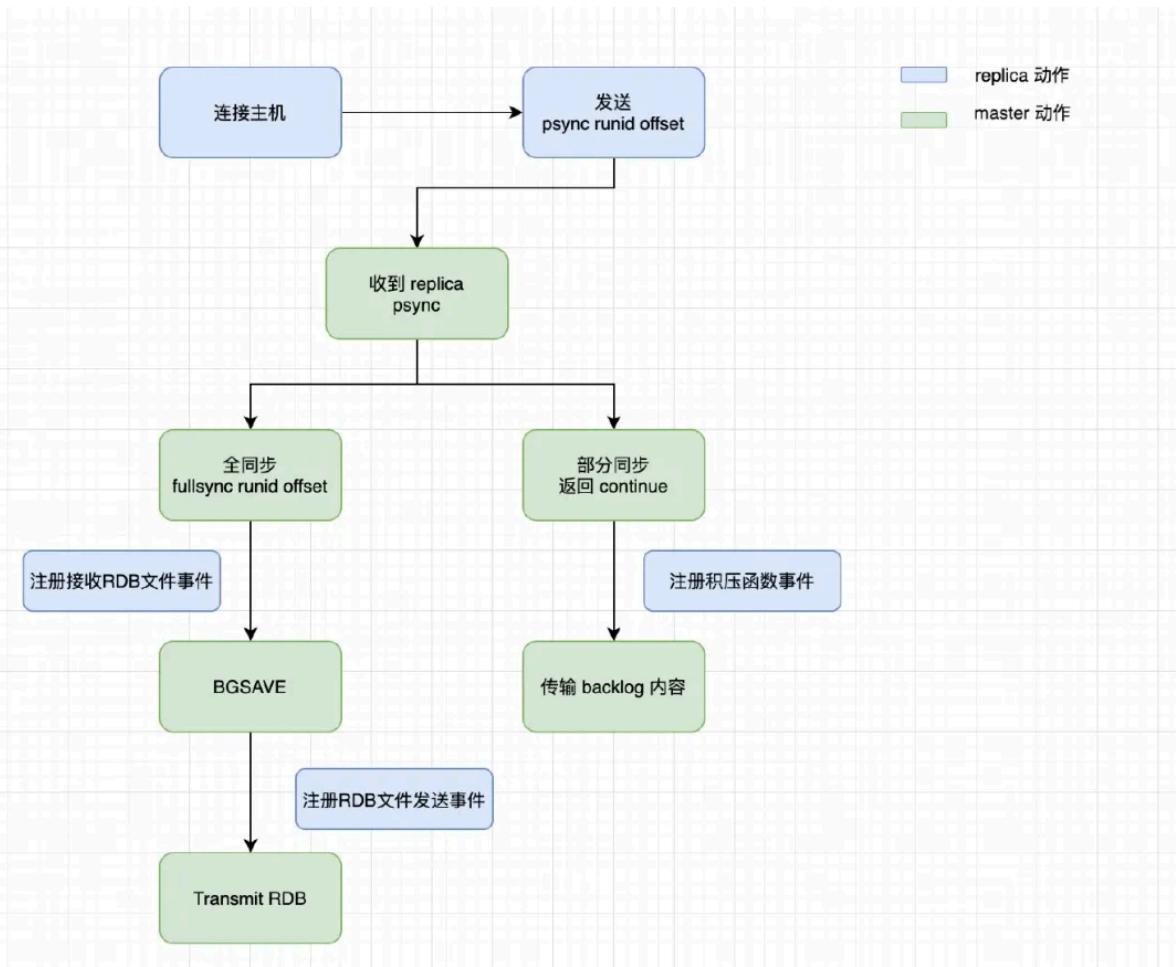
读写分离：读写分离保证高并发(10W+ QPS)：对于缓存来说一般都是支撑高并发读，写请求都是比较少的。采用读写分离的架构(一主多从)，master 负责接收写请求，数据同步到 slave 上提供读服务，如果遇到瓶颈只需要增加 slave 机器就可以水平扩容。

主从复制：

1. redis异步复制到slave节点
2. slave节点做复制时不会阻塞，而是使用旧的数据集提供服务。复制完成后，删除旧的数据集，加载新的数据集（加载过程中会暂停服务）
3. 开启主从复制，master必须开启**持久化 (rdb和aof)**，否则master宕机重启后数据是空的，slave一同步就把数据抹除了

同步流程：

1. slave启动给master发送psync命令
2. 如果是重新连接，master会复制给slave缺少的那部分数据
3. 如果是第一次连接master，会触发全量复制，master生成一份rdb快照，期间客户端的命令都缓存在内存中。rdb生成完成后，发送给slave，**slave先写入磁盘再加载到内存**，然后master将缓存的命令发送给slave。



Redis哨兵模式

哨兵功能：

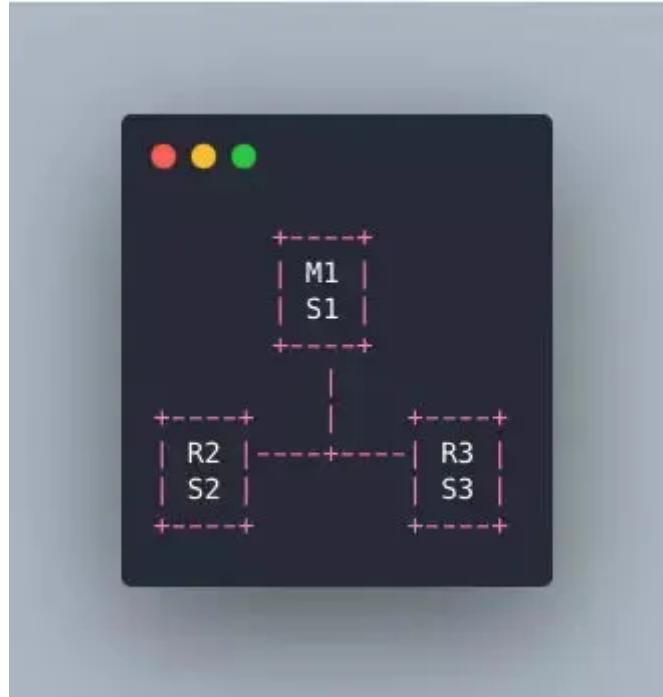
1. 集群监控：监控master和slave是否正常工作
2. 消息通知：如果某个redis实例有问题，哨兵将发消息通知管理员
3. 故障转移：如果master实例发生故障，会选举出新的master
4. 配置中心：如果故障转移完成，会通知客户端新的master地址

哨兵的注意事项：

1. 哨兵至少三个，保证自己的高可用；
2. 哨兵+主从的部署架构是用来保证 redis 集群高可用的，并非保证数据不丢失；
3. 哨兵(Sentinel)需要通过不断的测试和观察才能保证高可用。
4. 哨兵检测master宕机分为主观宕机和客观宕机：
 1. 主观：一个哨兵觉得 master 宕机了，达成条件是如果一个哨兵 ping master 超过了 is-master-down-after-milliseconds 指定的毫秒数后就认为主观宕机；
 2. 客观：一个哨兵在指定时间内收到了 majority(大多数) 数量的哨兵也认为那个 master 宕机了，就是客观宕机。
5. 哨兵之间的互相发现：哨兵是通过 redis 的 pub/sub 实现的。

为什么2个哨兵不能执行：因为要满足大多数哨兵存活，2的大多数是2，此时master宕机，其节点上的哨兵也宕机，就不能满足大多数，从而无法实现故障转移。

三哨兵集群：



Redis持久化——RDB

RDB (Redis DataBase) 是将某一个时刻的内存快照 (Snapshot)，以二进制的方式写入磁盘的过程。RDB 有两种方式 save 和 bgsave:

- save: 执行就会触发 Redis 的持久化，但同时也是使 Redis 处于阻塞状态，直到 RDB 持久化完成，才会响应其他客户端发来的命令；
- bgsave: bgsave 会 fork() 一个子进程来执行持久化，整个过程中只有在 fork() 子进程时有短暂的阻塞，当子进程被创建之后，Redis 的主进程就可以响应其他客户端的请求了。

除了使用 save 和 bgsave 命令触发之外，RDB 支持自动触发。自动触发策略可配置 Redis 在指定的时间内，数据发生了多少次变化时，会自动执行 bgsave 命令。在 redis 配置文件中配置：

在时间 `m` 秒内，如果 `Redis` 数据至少发生了 `n` 次变化，那么就自动执行`BGSAVE`命令。
`save m n`

RDB 的优点:

- RDB 会定时生成多个数据文件，**每个数据文件都代表了某个时刻的 redis 全量数据，适合做冷备**，可以将这个文件上传到一个远程的安全存储中，以预定好的策略来定期备份 redis 中的数据；
- RDB 对 redis **对外提供读写服务的影响非常小**，redis 是通过 `fork` 主进程的一个子进程操作磁盘 IO 来进行持久化的；
- **相对于 AOF，直接基于 RDB 来恢复 reids 数据更快。**

RDB 的缺点:

- 如果使用 RDB 来恢复数据，**会丢失一部分数据，因为 RDB 是定时生成的快照文件**；
- RDB 每次来 `fork` 出子进程的时候，**如果数据文件特别大，可能会影响对外提供服务，暂停数秒**(主进程需要拷贝自己的内存表给子进程，实例很大的时候这个拷贝过程会很长)。`latest_fork_usec` 代表 fork 导致的延时；Redis 上执行 `INFO` 命令查看 `latest_fork_usec`；**当 RDB 比较大的时候，应该在 slave 节点执行备份，并在低峰期执行。**

Redis持久化——AOF

redis 对每条写入命令进行日志记录，以 append-only 的方式写入一个日志文件（文本），redis 重启的时候通过重放日志文件来恢复数据集。（由于运行久了 AOF 文件会越来越大，redis 提供一种 rewrite 机制，基于当前内存中的数据集，来构建一个更小的 AOF 文件，将旧的庞大的 AOF 文件删除）。rewrite 即把日志文件压缩，通过 bgrewriteaof 触发重写。AOF rewrite 后台执行的方式和 RDB 有类似的地方，fork 一个子进程，主进程仍进行服务，子进程执行 AOF 持久化，数据被 dump 到磁盘上。与 RDB 不同的是，后台子进程持久化过程中，主进程会记录期间的所有数据变更（主进程还在服务），并存储在 server.aof_rewrite_buf_blocks 中；后台子进程结束后，Redis 更新缓存追加到 AOF 文件中，是 RDB 持久化所不具备的。

AOF 的工作流程如下：

1. Redis 执行写命令后，把这个命令写入到 AOF 文件内存中（write 系统调用）；
2. Redis 根据配置的 AOF 刷盘策略，把 AOF 内存数据刷到磁盘上（fsync 系统调用）；
3. 根据 rewrite 相关的配置触发 rewrite 流程。

appendfsync: 刷盘的机制：

- always: 主线程每次执行写操作后立即刷盘，此方案会占用比较大的磁盘 IO 资源，但数据安全性最高；
- everysec: 主线程每次写操作只写内存就返回，然后由后台线程每隔 1 秒执行一次刷盘操作（触发 fsync 系统调用），此方案对性能影响相对较小，但当 Redis 崩机时会丢失 1 秒的数据（常用）；
- no: 主线程每次写操作只写内存就返回，内存数据什么时候刷到磁盘，交由操作系统决定，此方案对性能影响最小，但数据安全性也最低，Redis 崩机时丢失的数据取决于操作系统刷盘时机。

AOF还有rewrite策略：

1. 相较于上一版aof文件大小百分比达到多少时重写
2. 最小容忍aof文件大小

AOF 的优点：

- 可以更好的保证数据不丢失，一般 AOF 每隔 1s 通过一个后台线程来执行 fsync(强制刷新磁盘页缓存)，**最多丢失 1s 的数据**；
- AOF 以 append-only 的方式写入(顺序追加)，没有磁盘寻址开销，性能很高；
- AOF 即使文件很大，触发后台 rewrite 的操作的时候一般也不会影响客户端的读写，(rewrite 的时候会对其中指令进行压缩，创建出一份恢复需要的最小日志出来)。
- 通过文本记录，适合做灾难性的误操作的紧急恢复，比如不小心使用 flushall 清空了所有数据，只要 rewrite 没有发生，就可以立即拷贝 AOF，将最后一条 flushall 命令删除，再回放 AOF 恢复数据。

AOF 的缺点：

- 同一份数据，因为 AOF 记录的命令会比 RDB 快照文件更大；
- AOF 开启后，支持写的 QPS 会比 RDB 支持写的 QPS 要低，毕竟 AOF 有写磁盘的操作。

结合使用AOF和RDB

两者综合使用，将 AOF 配置成每秒 fsync 一次。RDB 作为冷备，**AOF 用来保证数据不丢失的恢复第一选择，当 AOF 文件损坏或不可用的时候还可以使用 RDB 来快速恢复。**

数据恢复过程

1. 启动Redis服务：当Redis启动时，它会检查配置文件中的持久化设置。

2. 检测AOF文件：

- 如果 `appendonly` 配置设置为 `yes`，Redis会首先查找AOF文件。
- 如果AOF文件存在且有效，Redis将使用AOF文件来恢复数据。这是因为AOF文件通常包含自最后一次RDB快照以来的所有写操作，能够重构出最完整的数据状态。

3. AOF文件缺失或损坏时使用RDB：

- 如果AOF文件不存在或损坏，Redis将回退到使用RDB文件。
- Redis将加载RDB文件中的数据快照，这通常代表了在某个时间点的完整数据状态。

4. 处理AOF中的新增记录：

- 如果在RDB文件被加载后，还存在有效的AOF文件，Redis将处理AOF文件中自最后一次RDB快照之后记录的所有命令。
- 这意味着Redis首先通过RDB文件快速恢复到一个较早的状态，然后通过执行AOF文件中记录的命令来更新这些数据，从而达到最新状态。

5. 数据完整性验证：

- 一旦数据通过RDB和AOF文件恢复完成，建议进行数据的完整性验证，确保所有数据都已正确加载且无损坏。

Redis集群

redis主从复制、读写分离实现了一定程度的高并发、并保证高可用，但有如下限制：

1. master 数据和 slave 数据一模一样，master 的数据量就是集群的限制瓶颈；
2. redis 集群的写能力也受到了 master 节点的单机限制。

Redis Cluster 支持 N 个 master node，每个 master node 可以挂载多个 slave node：

1. 自动将数据切片，每个 master 上放一部分数据；
2. 提供内置的高可用支持，部分 master 不可用时还是能够工作；
3. redis cluster 模式下，每个 redis 要开放两个端口：6379 和 10000+以后的端口(如 16379)。
16379 是用来节点之间通信的，使用的是 cluster bus 集群总线。cluster bus 用来做故障检测，配置更新，故障转移授权。

redis集群负载均衡：redis cluster 采用一致性 hash+虚拟节点 来负载均衡。redis cluster 有固定的 16384 个 slot (2^{14})，对每个 key 做 CRC16 值计算，然后对 16384 mod。可以获取每个 key 的 slot。redis cluster 每个 master 都会持有部分 slot，比如三个 master 那么每个 master 就会持有 5000 多个 slot。hash slot 让 node 的添加和删除变得很简单，增加一个 master，就将其他 master 的 slot 移动部分过去，减少一个就分给其他 master，这样让集群扩容的成本变得很低。

redis集群通信：与集中式不同(如使用 zookeeper 进行分布式协调注册)，redis cluster 使用的是 gossip 协议进行通信。并不是将集群元数据存储在某个节点上，而是不断的互相通信，保持整个集群的元数据是完整的。gossip 协议所有节点都持有一份元数据，不同节点的元数据发生了变更，就不断的将元数据发送给其他节点，让其他节点也进行元数据的变更。

redis cluster 主备切换与高可用

1. 判断节点宕机：如果有一个节点认为另外一个节点宕机，那就是主观宕机。如果多个节点认为一个节点宕机，那就是客观宕机。跟哨兵的原理一样；

2. 对宕机的 master，从其所有的 slave 中选取一个切换成 master node，再此之前会进行一次过滤，检查每个 slave 与 master 的断开时间，如果超过了 cluster-node-timeout * cluster-slave-validity-factor 就没有资格切换成 master；
3. 从节点选取：每个从节点都会根据从 master 复制数据的 offset，来设置一个选举时间，offset 越大的从节点，选举时间越靠前，master node 开始给 slave 选举投票，如果大部分 master($n/2+1$)都投给了某个 slave，那么选举通过(与 zk 有点像，选举时间类似于 epochid)；
4. 整个流程与哨兵类似，可以说 redis cluster 集成了哨兵的功能，更加的强大；
5. Redis 集群部署相关问题 redis 机器的配置，多少台机器，能达到多少 qps?
 - 机器标准:8 核+32G
 - 集群: 5 主+5 从(每个 master 都挂一个 slave)
 - 效果: 每台机器最高峰每秒大概 5W，5 台机器最多就是 25W，每个 master 都有一个从节点，任何一个节点挂了都有备份可切换成主节点进行故障转移

脑裂问题哨兵模式下

- master 下挂载了 3 个 slave，如果 master 由于网络抖动被哨兵认为宕机了，执行了故障转移，从 slave 里面选取了一个作为新的 master，这个时候老的 master 又恢复了，刚好又有 client 连的还是老的 master，就会产生脑裂，数据也会不一致，比如 incr 全局 id 也会重复。

1. 解决脑裂问题的配置策略

在Redis中，`min-slaves-to-write` 和 `min-slaves-max-lag` 这两个配置参数可以帮助防止脑裂问题，尤其是在主从复制的环境中。这两个参数的作用如下：

- `min-slaves-to-write`: 这个参数设置了一个阈值，指定了必须有至少多少个从节点与主节点保持在连接状态，主节点才接受写操作。例如，如果设置为1，那么至少需要有一个从节点连接到主节点，主节点才会接受写请求。
- `min-slaves-max-lag`: 这个参数用来设置从节点与主节点之间的最大允许复制延迟时间（以秒为单位）。如果从节点复制主节点的数据延迟超过了这个时间，即使从节点的数量满足 `min-slaves-to-write` 的要求，主节点也会拒绝写操作。

这两个参数联合使用时，可以防止在主节点数据未能及时同步到从节点的情况下发生数据写入，从而减少因脑裂导致的数据不一致问题。

2. 客户端连接到Redis的TCP连接考量

Redis虽然是单线程处理命令请求，但它使用IO多路复用技术来同时处理多个网络连接。关于使用单个TCP连接还是多个TCP连接的性能考量，以下是一些关键点：

- **多个TCP连接**: 使用多个TCP连接可以在一个IO周期内处理更多的就绪IO事件，这通常会提高处理效率，因为Redis可以并行地从多个连接读取数据并响应。
- **单个TCP连接与Pipeline**: 如果使用单个TCP连接，并配合pipeline技术（即一次发送多个命令并批量接收响应），这种方式可以减少TCP连接的建立和关闭的开销，同时减少网络延迟。然而，如果单次pipeline的请求过多，也可能会导致在单个IO周期内处理的命令数量过多，从而影响性能。
- **连接数的选择**: 在Redis Cluster环境中，通常建议控制每个节点的连接数在100个以内。这是因为过多的连接可能会增加Redis服务器的负担，尤其是在处理大量小请求时。

Redis的底层数据结构

Redis是一个开源的、基于内存的、使用C语言编写的key-value数据库。

Redis的基础数据类型及其对应的底层数据库如下：

string	list	hash	set	sorted set
简单动态字符串	双向链表、压缩链表	哈希表、压缩链表	整数数组、字典	跳表、压缩链表

底层实现的时间复杂度：

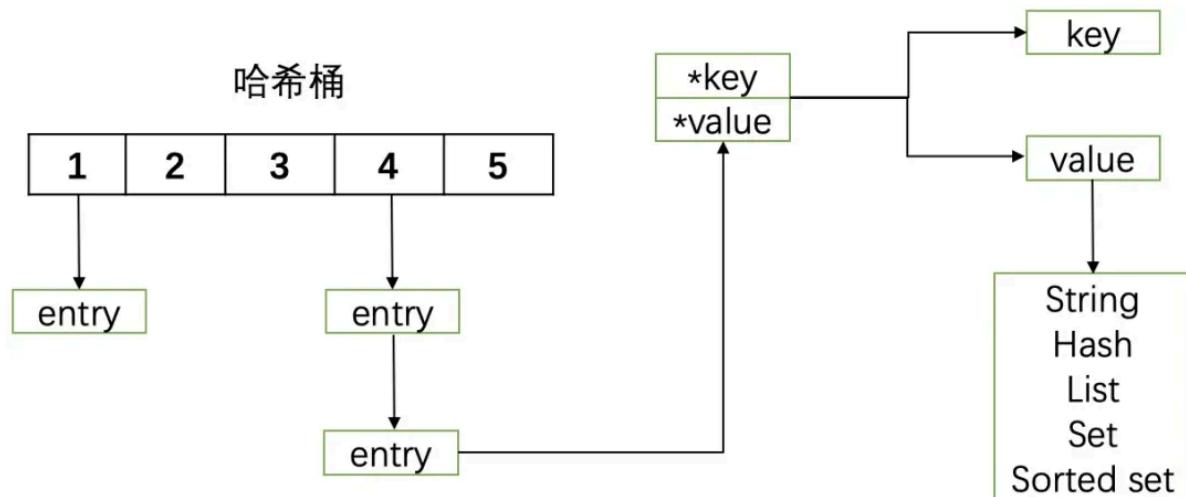
双向链表	双向链表	哈希表	整数数组	跳表
O(n)	O(n)	O(1)	O(n)	O(logn)

可以看出除了 string 类型的底层实现只有一种数据结构，其他四种均有两种底层实现，这四种类型为集合类型，其中一个键对应了一个集合的数据；

Redis键值如何存储

Redis 为了快速访问键值对，采用了哈希表来保存所有的键值对，一个哈希表对应了多个哈希桶，所谓的哈希桶是指哈希表数组中的每一个元素，当然哈希表中保存的不是值本身，是指向值的指针，如下图。

其中哈希桶中的 entry 元素中保存了 key 和 value 指针，分别指向了实际的键和值。通过 Redis 可以在 O(1) 的时间内找到键值对，只需要计算 key 的哈希值就可以定位位置，但从下图可以看出，在 4 号位置出现了冲突，两个 key 映射到了同一个位置，这就产生了哈希冲突，会导致哈希表的操作变慢。虽然 Redis 通过链式冲突解决该问题，但如果数据持续增多，产生的哈希冲突也会越来越多，会加重 Redis 的查询时间；



为了解决上述的哈希冲突问题，Redis 会对哈希表进行rehash操作，也就是增加目前的哈希桶数量，使得 key 更加分散，进而减少哈希冲突的问题，主要流程如下：

1. 采用两个 hash 表进行操作，当哈希表 A 需要进行扩容时，给哈希表 B 分配两倍的空间；
2. 将哈希表 A 的数据重新映射并拷贝给哈希表 B；
3. 释放 A 的空间。

上述的步骤可能会存在一个问题，当哈希表 A 向 B 复制的时候，是需要一定的时间的，可能会造成 Redis 的线程阻塞，就无法服务其他的请求了。

针对上述问题，Redis 采用了渐进式 rehash（过程类似redis删除key的定期删除和惰性删除），主要的流程是：Redis 还是继续处理客户端的请求，每次处理一个请求的时候，就会将该位置所有的 entry 都拷贝到哈希表 B 中，当然也会存在某个位置一直没有被请求。Redis 也考虑了这个问题，通过设置一个定时任务进行 rehash，在一些键值对一直没有操作的时候，会周期性的搬移一些数据到哈希表 B 中，进而缩短 rehash 的过程。

Redis为什么使用单线程

Redis的单线程指网络I/O和键值对读写由一个线程来完成， bgsave、bgwriteaof等持久化操作由额外的线程执行。

单线程避免了线程切换和互斥同步的开销，通常情况下，使用多线程可以增加系统吞吐率或者可以增加系统扩展性。但其实对于多线程并发访问的控制一直是一个难点问题，如果没有精细的设计，比如说，只是简单地采用一个粗粒度互斥锁，就会出现不理想的结果。即使增加了线程，大部分线程也在等待获取访问共享资源的互斥锁，并行变串行，系统吞吐率并没有随着线程的增加而增加。

值得注意的是在 Redis6.0 中引入了多线程。在 Redis6.0 之前，从网络 IO 处理到实际的读写命令处理都是由单个线程完成的，但随着网络硬件的性能提升，Redis 的性能瓶颈有可能会出现在网络 IO 的处理上，也就是说**单个主线程处理网络请求的速度跟不上底层网络硬件的速度**。针对此问题，Redis 采用多个 IO 线程来处理网络请求，提高网络请求处理的并行度，**但多 IO 线程只用于处理网络请求，对于读写命令，Redis 仍然使用单线程处理！**

Redis为什么单线程还这么快

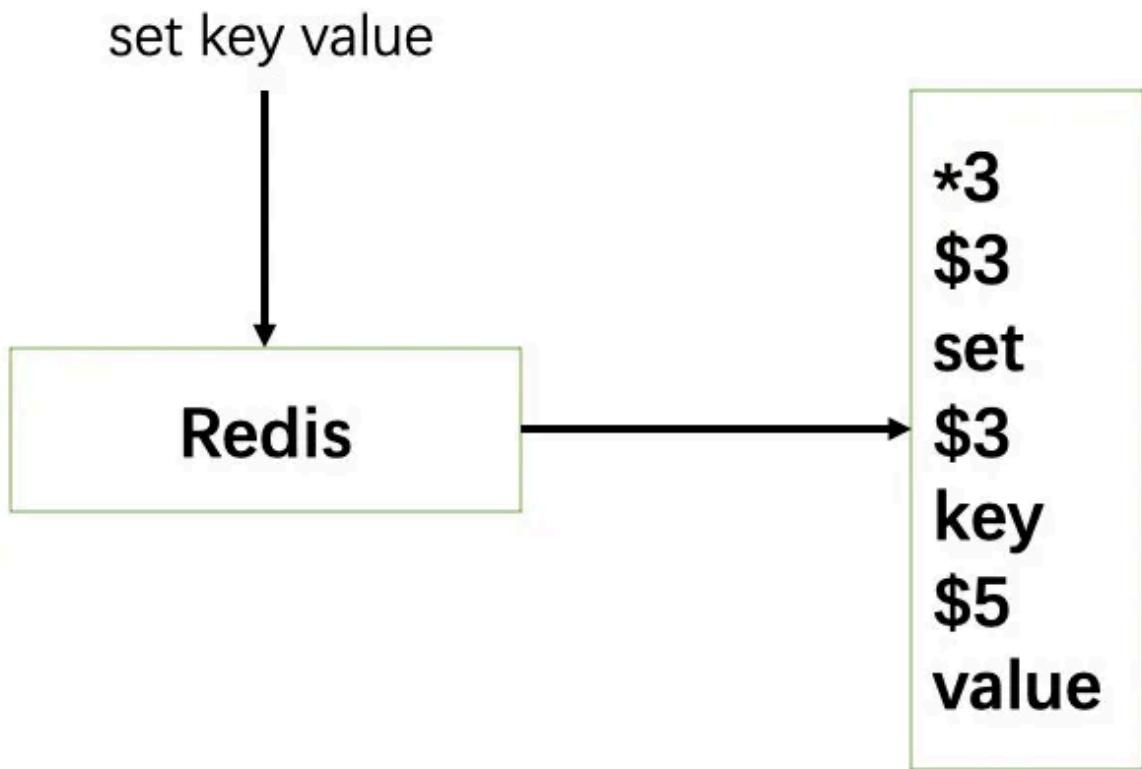
1. Redis 使用 I/O 多路复用机制，Reactor 模型，I/O 多路复用机制允许内核中同时存在多个监听套接字和已连接套接字，内核会一直监听这些套接字上的连接请求或数据请求。一旦有请求到达，就会交给 Redis 线程处理，这就实现了一个 Redis 线程处理多个 IO 流的效果，进而提升并发性。通过 Reactor 模型，Redis 将不同的事件分派给对应的事件处理器进行处理。
2. Redis 基于内存，十分迅速。
3. Redis 具有高效的底层数据结构，为优化内存，基本数据结构中除了 string 都有两种底层实现方式。
4. Redis 采用单线程，**避免了不必要的上下文切换和资源竞争，不存在多线程导致的 CPU 切换和锁的问题；**

AOF

AOF日志是先执行命令再写日志：

1. AOF 并不检查命令的正确性，因此先执行命令再写日志，避免错误的命令被记录，从而影响数据恢复，同时也不会阻塞当前的写操作。

AOF 中的命令格式：



*3表示命令分为3部分，\$3表示该部分命令、键、值多少字节。

AOF重写机制

AOF 重写就是根据所有的键值对创建一个新的 **AOF 文件**，可以减少大量的文件空间，减少的原因是：
AOF 对于命令的添加是追加的方式，逐一记录命令，但有可能存在某个键值被反复更改，产生了一些冗余数据，这样在重写的时候就可以过滤掉这些指令，从而更新当前的最新状态。

AOF 重写的过程是通过主线程 fork 后台的 bgrewriteaof 子进程来实现的，可以避免阻塞主进程导致性能下降，整个过程如下：

- AOF 每次重写，fork 过程会把主线程的内存拷贝一份 bgrewriteaof 子进程，里面包含了数据库的数据，拷贝的是父进程的页表，可以在不影响主进程的情况下逐一把拷贝的数据记入重写日志；
- 因为主线程没有阻塞，仍然可以处理新来的操作，如果这时候存在写操作，会先把操作先放入缓冲区，对于正在使用的日志，如果宕机了这个日志也是齐全的，可以用于恢复；对于正在更新的日志，也不会丢失新的操作，等到数据拷贝完成，就可以将缓冲区的数据写入到新的文件中，保证数据库的最新状态。

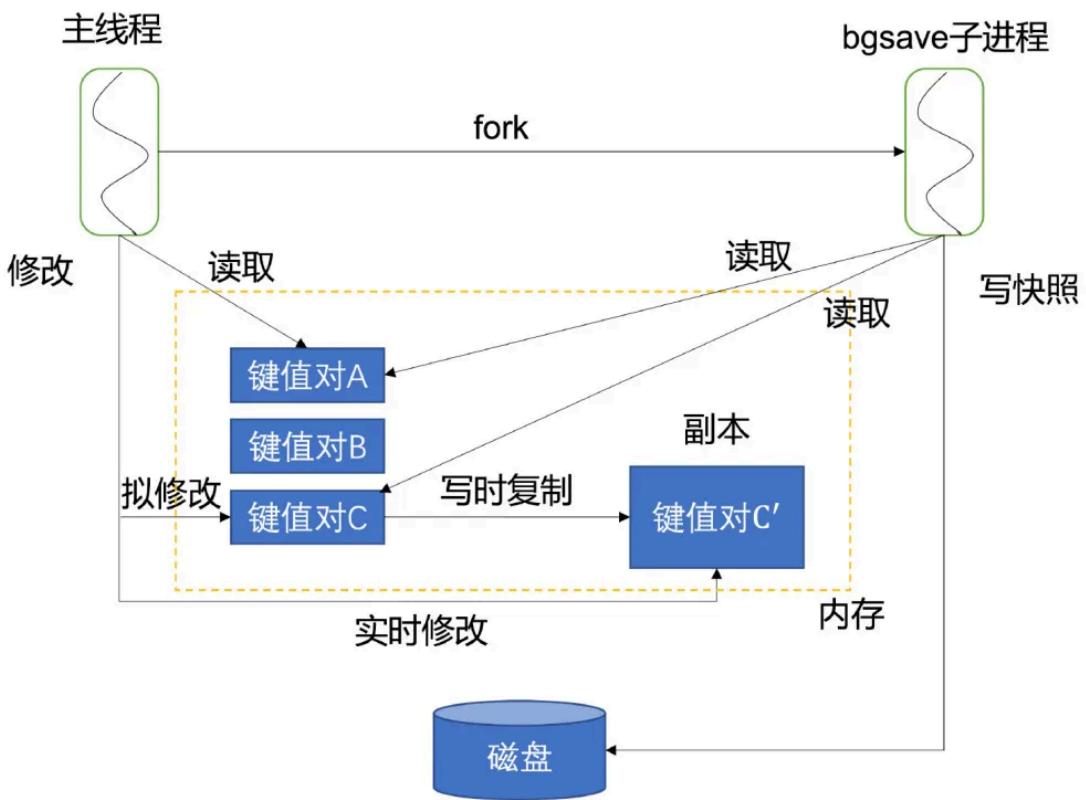
注意这里区别于RDB，RDB对于要修改的key是COW；AOF是写到缓冲区

RDB

`bgsave` 的命令来执行全量快照，提供了数据的可靠性保证，也避免了对 Redis 的性能影响。执行快照期间数据能不能修改呢？如果不能修改，快照过程中如果有新的写操作，数据就会不一致，这肯定是不符合预期的。Redis 借用了操作系统的**写时复制**，在执行快照的期间，正常处理写操作。

主要流程为：

- `bgsave` 子进程是由主线程 `fork` 出来的，可以共享主线程的所有内存数据；
- `bgsave` 子进程运行后，开始读取主线程的内存数据，并把它们写入 RDB 文件中；
- 如果主线程对这些数据都是读操作，例如 A，那么主线程和 `bgsave` 子进程互不影响；
- 如果主线程需要修改一块数据，如 C，这块数据会被复制一份，生成数据的副本，然主线程在这个副本上进行修改；`bgsave` 子进程可以把原来的数据 C 写入 RDB 文件；



理论上来说快照时间间隔越短越好，可以减少数据的丢失，毕竟 fork 的子进程不会阻塞主线程，但是频繁的将数据写入磁盘，会给磁盘带来很多压力，也可能会存在多个快照竞争磁盘带宽（当前快照没结束，下一个就开始了）。另一方面，虽然 fork 出的子进程不会阻塞，但 fork 这个创建过程是会阻塞主线程的，当主线程需要的内存越大，阻塞时间越长；针对上面的问题，Redis 采用了增量快照，在做一次全量快照后，后续的快照只对修改的数据进行记录，需要记住哪些数据被修改了，可以避免全量快照带来的开销。

混合使用AOF日志和RDB快照

在 Redis4.0 提出了混合使用 AOF 和 RDB 快照的方法，也就是两次 RDB 快照期间的所有命令操作由 AOF 日志文件进行记录。这样的好处是 RDB 快照不需要很频繁的执行，可以避免频繁 fork 对主线程的影响，而且 AOF 日志也只记录两次快照期间的操作，不用记录所有操作，也不会出现文件过大的情况，避免了重写开销。

通过上述方法既可以享受 RDB 快速恢复的好处，也可以享受 AOF 记录简单命令的优势。

对于 AOF 和 RDB 的选择问题：

- 数据不能丢失时，内存快照和 AOF 的混合使用是一个很好的选择；
- 如果允许分钟级别的数据丢失，可以只使用 RDB；
- 如果只用 AOF，优先使用 everysec 的配置选项，因为它在可靠性和性能之间取了一个平衡。

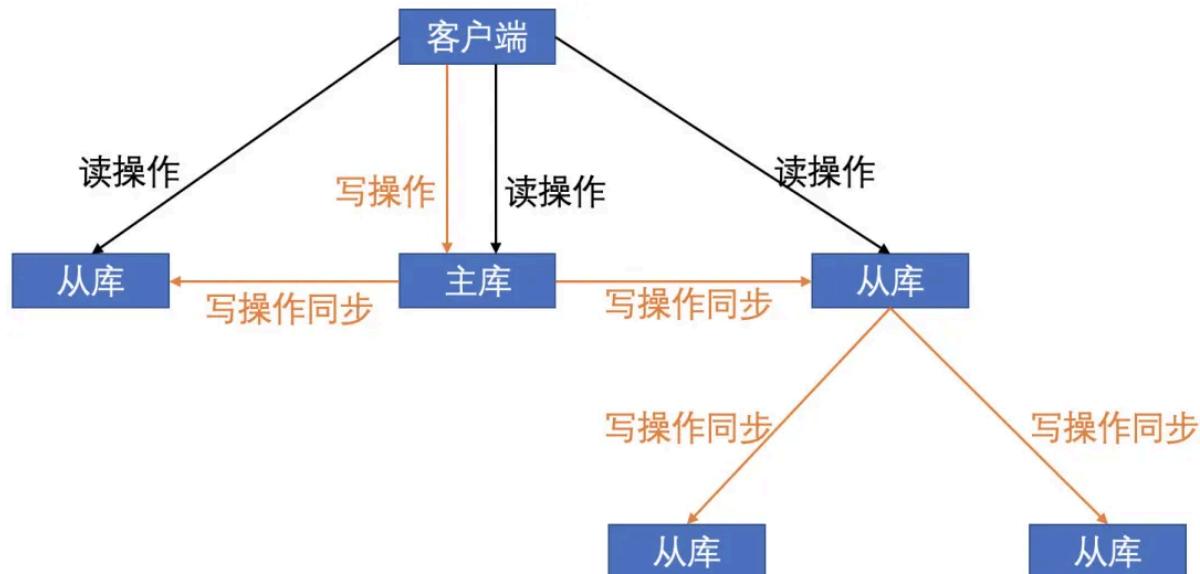
Redis数据同步

当存在多个 Redis 实例的时候，可以通过 replicaof 命令形成主库和从库的关系，在从库中输入：replicaof 主库 ip 6379 就可以在主库中复制数据，具体有三个阶段：

- 首先是主从库建立连接、协商同步的过程：从库向主库发送 psync 命令，代表要进行数据同步；psync 中包含了主库的 runID（Redis 启动时生成的随机 ID，初始值为：?）和复制进度 offset（设为-1，代表第一次复制）两个参数，主库接收到 psync 命令后，会用 FULLRESYNC 命令返回给从库，包含两个参数：主库 runID 和复制进度 offset；其中 FULLRESYNC 代表的全量复制，会将主库所有的数据都复制给从库；

- 待从库接收到数据后，在本地完成数据加载：主库执行 `bgsave` 命令，生成 RDB 文件，然后将文件发给从库，从库接收到 RDB 文件后，首先清空当前数据，然后再加载 RDB 文件；这个过程主库不会被阻塞，仍然可以接受请求，如果存在写操作，刚刚生成的 RDB 文件中是不包含这些新数据的，此时主库会在内存中用专门的 replication buffer 记录 RDB 文件生成后所有的写操作；
- 最后，主库会把 replication buffer 中的修改操作发给从库，从库重新执行这些操作，就可以实现主从库同步了。

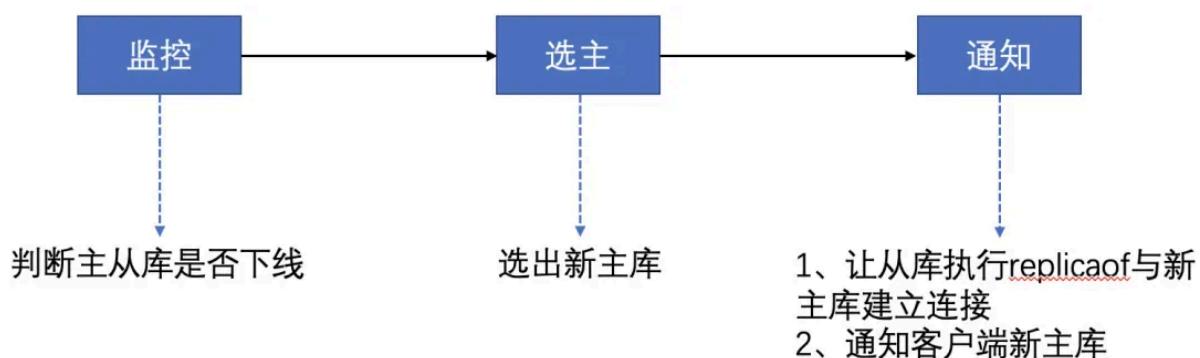
当然如果从库过多，主库就会频繁 fork 来生成 RDB 文件，fork 这个过程是会阻塞主进程的，因此一般使用主-从-从的模式，或者设置从库从其它从库来复制同步。



主从切换

Redis 采用了 **哨兵机制** 应对这些问题，哨兵机制是实现主从库自动切换的关键机制，在主从库运行的同时，它也在进行 **监控、选择主库和通知** 的操作；

- 监控。** 哨兵在运行时，周期性的给所有的主从库发送 PING 命令，检测是否仍在运行。如果从库没有响应哨兵的 PING 命令，哨兵就会将它标记为下线状态；如果主库没有在规定时间内响应哨兵的 PING 命令，哨兵也会判断主库下限，然后开始自动切换主库的流程。
- 选主。** 主库挂了之后，哨兵需要按照一定的规则选择一个从库，并将他作为新的主库。
- 通知。** 选取了新的主库后，哨兵会把新主库的连接信息发给其他从库，让它们执行 `replicaof` 命令和新主库建立连接，并进行数据复制；同时哨兵也会将新主库的消息发给客户端；



哨兵至少3个，必须多数哨兵认为主库下线才是客观下线。

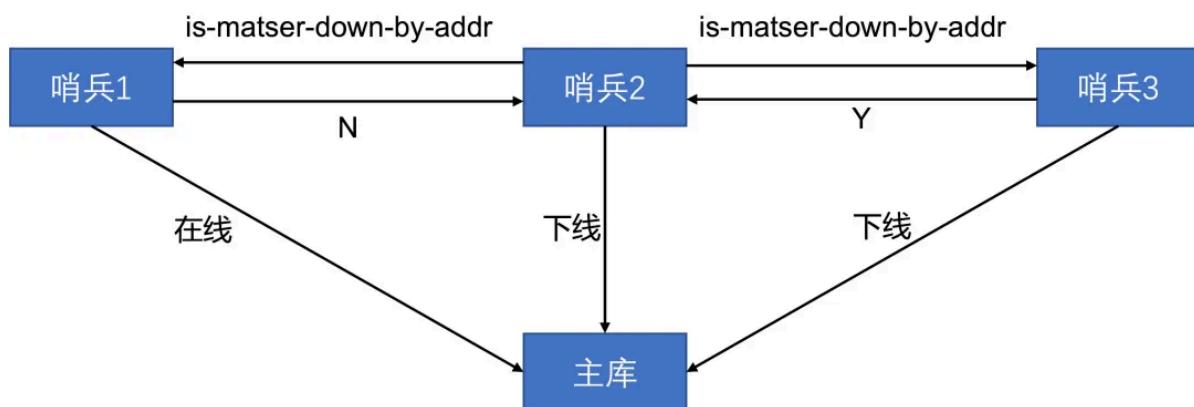
哨兵如何选举新的主库

选取主库的过程分为“**筛选 和 打分**”。主要是按照一定的规则过滤掉不符合的从库，再按照一定的规则给其余的从库打分，将最高分的从库作为新的主库。

- **筛选**。首先从库一定是正在运行的，还要判断从库之前的网络连接状态，如果总是断连并且超过了一定的阈值，哨兵会认为该从库的网络不好，也会将其筛掉。
- **打分**。哨兵机制根据三个规则依次进行打分：**从库优先级、从库复制进度以及从库 ID 号**。在某一轮有从库得分最高，那么它就是新的主库了，选主过程结束。如果该轮没有出现最高的，继续下一轮。
 1. 优先级最高的从库。用户可以通过 **slave-priority** 配置项，给不同的从库设置优先级。选主库的时候哨兵会给优先级高的从库打高分，如果一个从库优先级高，那么就是新主库；
 2. 从库复制进度最接近。主库的 `slave_repl_offset` 和从库 `master_repl_offset` 越接近，得分越高；
 3. ID 小的从库得分高。如果上面两轮也没有选出新主库，就会根据从库实例的 ID 来判断，ID 越小的从库得分越高。

哪个哨兵来执行主从库切换

这个过程和判断主库“客观下线”类似，也是一个投票的过程。如果某个哨兵判断了主库为下线状态，就会给其他的哨兵实例发送 **is-master-down-by-addr** 的命令，其他实例会根据自己和主库的连接状态作出 Y 或 N 的响应，Y 相当于赞成票，N 为反对票。一个哨兵获得一定的票数后，就可以标记主库为“客观下线”，这个票数是由参数 `quorum` 设置的。如下图：



例如：现在有 3 个哨兵，`quorum` 配置的是 2，那么，一个哨兵需要 2 张赞成票，就可以标记主库为“客观下线”了。这 2 张赞成票包括哨兵自己的一张赞成票和另外两个哨兵的赞成票。

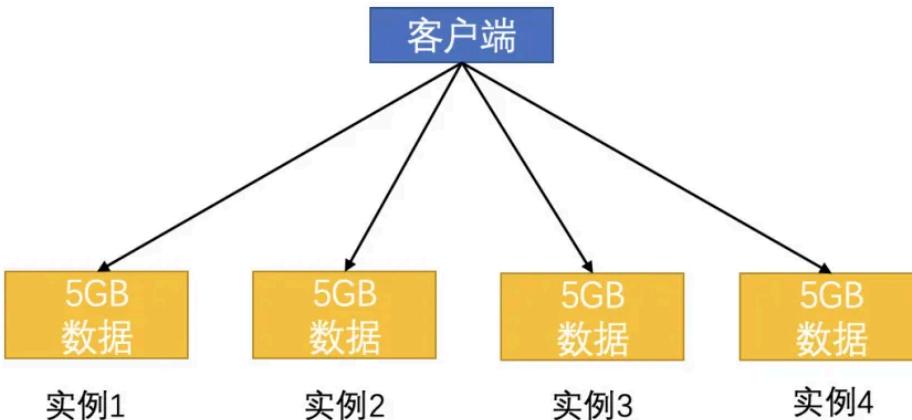
这个时候哨兵就可以给其他哨兵发送消息，表示希望自己来执行主从切换，并让所有的哨兵进行投票，这个过程称为“Leader 选举”，进行主从切换的哨兵称为 Leader。任何一个想成为 Leader 的哨兵都需要满足两个条件：

- 拿到半数以上的哨兵赞成票；
- 拿到的票数需要大于等于 `quorum` 的值。

以上就可以选出 Leader 然后进行主从库切换了。

Redis 集群

Redis 的切片集群可以解决这个问题，也就是启动多个 Redis 实例来组成一个集群，再按照一定的规则把数据划分为多份，每一份用一个实例来保存，这样客户端只需要访问对应的实例就可以获取数据。在这种情况下 fork 子进程一般不会给主线程带来较长时间的阻塞



选择切片集群也是需要解决一些问题的：

- 数据切片后，在多个实例之间怎么分布？
- 客户端怎么确定想要访问的实例是哪一个？

Redis 采用了 Redis Cluster 的方案来实现切片集群，具体的 Redis Cluster 采用了哈希槽（Hash Slot）来处理数据和实例之间的映射关系。在 Redis Cluster 中，一个切片集群共有 16384 个哈希槽（[为什么 Hash Slot 的个数是 16384](#)），这些哈希槽类似于数据的分区，每个键值对都会根据自己的 key 被映射到一个哈希槽中，映射步骤如下：

- 根据键值对 key，按照 CRC16 算法计算一个 16bit 的值；
- 用计算的值对 16384 取模，得到 0 ~ 16383 范围内的模数，每个模数对应一个哈希槽。

这时候可以得到一个 key 对应的哈希槽了，哈希槽又是如何找到对应的实例的呢？

在部署 Redis Cluster 的时候，可以通过 cluster create 命令创建集群，此时 Redis 会自动把这些槽分布在集群实例上，例如一共有 N 个实例，那么每个实例包含的槽个数就为 $16384/N$ 。当然可能存在 Redis 实例中内存大小配置不一的问题，内存大的实例具有更大的容量。这种情况下可以通过 cluster addslots 命令手动分配哈希槽。

客户端定位集群中的数据

客户端请求的 key 可以通过 CRC16 算法计算得到，但客户端还需要知道哈希槽分布在哪个实例上。在最开始客户端和集群实例建立连接后，实例就会把哈希槽的分配信息发给客户端，实例之间会把自己的哈希槽信息发给和它相连的实例，完成哈希槽的扩散。这样客户端访问任何一个实例的时候，都能获取所有的哈希槽信息。当客户端收到哈希槽的信息后会把哈希槽对应的信息缓存在本地，当客户端发送请求的时候，会先找到 key 对应的哈希槽，然后就可以给对应的实例发送请求了。

但是，哈希槽和实例的对应关系不是一成不变的，可能会存在新增或者删除的情况，这时候就需要重新分配哈希槽；也可能为了负载均衡，Redis 需要把所有的实例重新分布。虽然实例之间可以互相传递消息以获取最新的哈希槽分配信息，但是客户端无法感知这个变化，就会导致客户端访问的实例可能不是自己所需要的了。

Redis Cluster 提供了重定向的机制，当客户端给实例发送数据读写操作的时候，如果这个实例上没有找到对应的数据，此时这个实例就会给客户端返回 MOVED 命令的相应结果，这个结果中包含了新实例的访问地址，此时客户端需要再给新实例发送操作命令以进行读写操作，MOVED 命令如下：

```
GET hello:key
(error) MOVED 3333 33.33.33.33:6379
```

返回的信息代表客户端请求的 key 所在的哈希槽为 3333，实际是在 33.33.33.33 这个实例上，此时客户端只需要向 33.33.33.33 这个实例发送请求就可以了。

此时也存在一个小问题，哈希槽中对应的数据过多，导致还没有迁移到其他实例，此时客户端就发起了请求，在这种情况下，客户端就对实例发起了请求，**如果数据还在对应的实例中，会给客户端返回数据；如果请求的数据已经被转移到其他实例上，客户端就会收到实例返回的 ASK 命令，该命令表示：哈希槽中数据还在前一种、ASK 命令把客户端需要访问的新实例返回了。此时客户端需要给新实例发送 ASKING 命令以进行请求操作；**

值得注意的是 ASK 信息和 MOVED 信息不一样，**ASK 信息并不会更新客户端本地的缓存的哈希槽分配信息**，也就是说如果客户端再次访问该哈希槽还是会请求之前的实例，直到数据迁移完成。

为什么需要分布式锁

与分布式锁对应的是单机锁：写多线程程序时，为了避免同时操作进程中的全局变量，通常会使用一把锁来「互斥」，以保证全局变量的正确性。又或者，当我们在同一台机器的不同进程，想要同时操作一个共享资源（例如修改同一个文件），我们可以使用操作系统提供的「文件锁」或「信号量」来做互斥。

高并发业务场景下，部署在不同机器上的业务进程，如果需要同时操作共享资源，为了避免「时序性」问题，通常会借助分布式锁来做互斥，以保证业务的正确性。

可以实现分布式锁的中间件有很多，redis、zookeeper和etcd，高并发业务场景下，为了追求更好的性能，通常选择redis。

基于redis的分布式锁如何实现

最开始最基础的分布式锁

redis提供了`SETNX (SET if Not exists)`命令实现互斥，即只有当这个key不存在，才进行设置。

客户端1进行加锁，加锁成功：

```
127.0.0.1:6379> SETNX lock 1
(integer) 1      // 客户端1, 加锁成功
```

客户端2进行加锁，加锁失败：

```
127.0.0.1:6379> SETNX lock 1
(integer) 0      // 客户端2, 加锁失败
```

客户端1操作完毕，使用`DEL`释放锁：

```
127.0.0.1:6379> DEL lock // 释放锁
(integer) 1
```

存在的问题

上述方式存在明显的问题：

1. 客户端1业务逻辑异常，没有及时释放锁
2. 进程挂了，没机会释放锁

针对问题1，在写业务的时候可以提高代码的健壮性，比如go可以用`defer`关键字确保锁被释放。

针对问题2，我们可以为锁设置过期时间，到期立即释放。

```
127.0.0.1:6379> SETNX lock 1      // 加锁
(integer) 1
127.0.0.1:6379> EXPIRE lock 10   // 10s后自动过期
(integer) 1
```

但上述操作依旧存在问题，`EXPIRE`操作可能因为客户端宕机、网络原因或者redis宕机未执行，导致锁不能被释放。

因此我们需要将加锁和设置过期时间原子性执行，Redis 2.6.12 之后，Redis 扩展了 `SET` 命令的参数，把 `NX/EX` 集成到了 `SET` 命令中，用这一条命令就可以了：

```
// 一条命令保证原子性执行
127.0.0.1:6379> SET lock 1 EX 10 NX
OK
```

这样是否就没问题了呢？试想这样一种场景：

1. 客户端1加锁成功，开始操作资源
2. 客户端锁过期，但客户端1资源操作还未完成，锁被自动释放
3. 客户端2成功获得锁
4. 客户端1资源操作完成，释放锁，但却释放了客户端2的锁

这里出现了两个问题：

1. 资源未操作完，锁便过期，我们加锁只是为了多个进程操作共享资源的互斥性，如果资源未操作完，锁释放，那就不能保证资源的安全
2. 释放了不属于自己的锁

针对问题1，我们可以先设置1个过期时间，再通过守护线程，定期去检测这个锁的过期时间，如果快过期了，资源操作还没完成，就重新设置过期时间。

针对问题2，我们需要再释放自己锁的时候进行判断当前锁是否是自己的，也就是在加锁的时候加入自己的唯一标识，比如UUID。

```
// 锁的VALUE设置为UUID
127.0.0.1:6379> SET lock $uuid EX 20 NX
OK

// 释放锁的伪代码
// 锁是自己的，才释放
if redis.get("lock") == $uuid:
    redis.del("lock")
```

但同样需要保证释放操作是原子性的，不然可能出现以下情况：

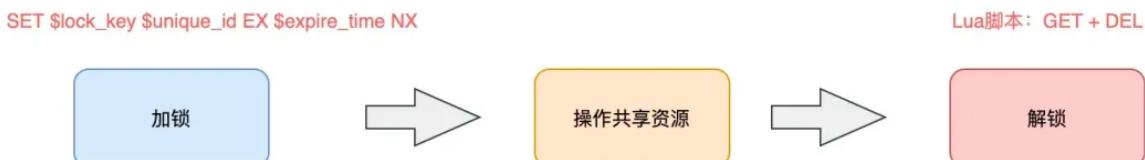
1. 客户端1检查到当前锁是自己的
2. 锁刚好过期
3. 客户端2获得锁
4. 客户端1释放了客户端2的锁

由于redis是单线程的，会顺序执行所有操作，因此可以通过lua脚本保证原子性。虽然 Lua 脚本内部可以执行多个命令，但是从 Redis 的角度看，整个脚本的执行就像一个事务一样。这意味着脚本中的所有操作要么全部执行，要么因为错误而全部不执行。并且由于是单线程，在 Lua 脚本执行期间，不会发生任何形式的上下文切换，这避免了并发执行的问题。

```
// 判断锁是自己的，才释放
if redis.call("GET",KEYS[1]) == ARGV[1]
then
    return redis.call("DEL",KEYS[1])
else
    return 0
end
```

基于redis的分布式锁

1. **加锁**: SET lock unique_id EX \$expire_time NX
2. **操作共享资源**: 没操作完之前，开启守护线程，定期给锁续期
3. **释放锁**: Lua 脚本，先 GET 判断锁是否归属自己，再 DEL 释放锁



以上场景是针对单个redis实例，但在实际生产环境中，通常是采用主从集群+哨兵的模式部署。当发生主从切换时，这个分布式锁可能不安全：

1. 客户端1加锁成功
2. 主库异常宕机，同步命令还没同步到从库上
3. 从库被哨兵提升为主库，锁在新的主库上丢失了

Redlock方案

针对上述问题，redis作者提出了一种名为redlock的解决方案，redlock的前提是：

1. 不部署从库和哨兵实例，只部署主库
2. 主库部署多个，至少5个实例

redlock流程如下：

1. 获取当前时间戳T1
2. 依次向5个主库发送加锁请求，每个请求都会设置1个超时时间（远小于锁的有效时间），如果加锁失败（网络超时、锁被它人持有），就立即向下一个redis实例申请
3. 如果大于等于3个（大多数）个redis实例加锁成功，则再次获取当前时间戳T2，如果T2-T1<锁的过期时间，则认为客户端加锁成功，否则失败。
4. 加锁成功，则操作资源
5. 加锁失败，对每个redis实例尝试释放锁

Redlock为什么这样做

1) 为什么要在多个实例上加锁？

为了容错，即使部分实例宕机，剩余实例加锁成功，整个锁服务依旧可用

2) 为什么大多数加锁成功，才算成功？

这是分布式系统的容错问题，这个问题的结论是：**如果只存在「故障」节点，只要大多数节点正常，那么整个系统依旧是可以提供正确服务的。**

参考拜占庭将军模型

3) 为什么步骤 3 加锁成功后，还要计算加锁的累计耗时？

确保返回加锁成功的所有实例的锁没有过期

4) 为什么释放锁，要操作所有节点？

可能有些redis实例已经加上锁了，但由于网络问题，导致我们没收到请求，以为这个实例没加上锁，因此需要对每个实例释放锁，从而确保每个锁被正确释放。

Redlock的问题

分布式专家 Martin 对于 Redlock 的质疑，在他的文章中，主要阐述了 4 个论点：

1) 分布式锁的目的是什么？

第一，效率。

使用分布式锁的互斥能力，是避免不必要的做同样的两次工作（例如一些昂贵的计算任务）。如果锁失效，并不会带来「恶性」的后果，例如发了 2 次邮件等，无伤大雅。

第二，正确性。

使用锁用来防止并发进程互相干扰。如果锁失效，会造成多个进程同时操作同一条数据，产生的后果是**数据严重错误、永久性不一致、数据丢失**等恶性问题，就像给患者服用重复剂量的药物一样，后果严重。

他认为，如果你是为了前者——效率，那么使用单机版 Redis 就可以了，即使偶尔发生锁失效（宕机、主从切换），都不会产生严重的后果。而使用 Redlock 太重了，没必要。

而如果是为了正确性，Martin 认为 Redlock 根本达不到安全性的要求，也依旧存在锁失效的问题！

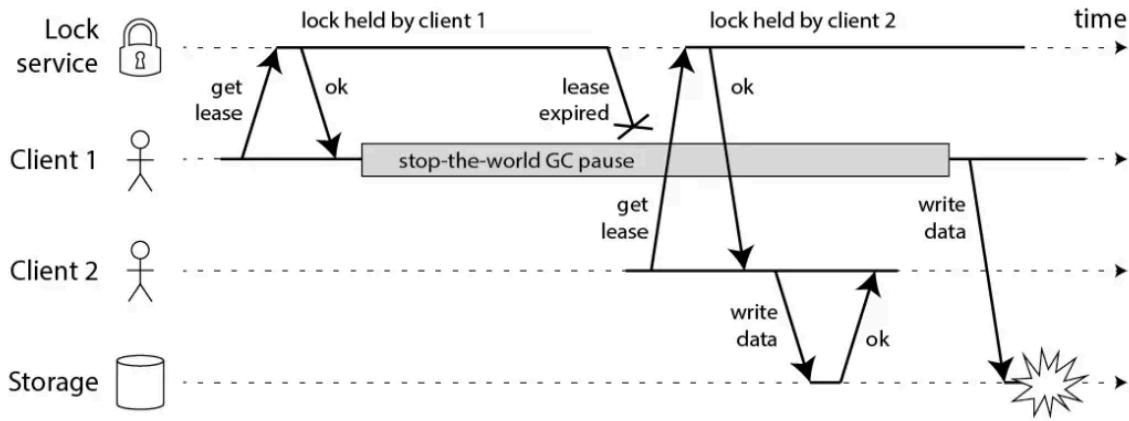
2) 锁在分布式系统中会遇到的问题

Martin 表示，一个分布式系统，存在着你想不到的各种异常情况。这些异常场景主要包括三大块，这也是分布式系统会遇到的三座大山：**NPC**。

- N: Network Delay，网络延迟
- P: Process Pause，进程暂停（GC）
- C: Clock Drift，时钟漂移

Martin 用一个进程暂停（GC）的例子，指出了 Redlock 安全性问题：

1. 客户端 1 请求锁定节点 A、B、C、D、E
2. 客户端 1 的拿到锁后，进入 GC（时间比较久）
3. 所有 Redis 节点上的锁都过期了
4. 客户端 2 获取到了 A、B、C、D、E 上的锁
5. 客户端 1 GC 结束，认为成功获取锁
6. 客户端 2 也认为获取到了锁，发生「冲突」



Martin 认为，GC 可能发生在程序的任意时刻，而且执行时间是不可控的。

注：当然，即使是使用没有 GC 的编程语言，在发生网络延迟时，也都有可能导致 Redlock 出现问题，这里 Martin 只是拿 GC 举例。

3) 假设时钟正确的是不合理的

又或者，当多个 Redis 节点「时钟」发生问题时，也会导致 Redlock 锁失效。

1. 客户端 1 获取节点 A、B、C 上的锁，但由于网络问题，无法访问 D 和 E
2. 节点 C 上的时钟「向前跳跃」，导致锁到期
3. 客户端 2 获取节点 C、D、E 上的锁，由于网络问题，无法访问 A 和 B
4. 客户端 1 和 2 现在都相信它们持有了锁（冲突）

Martin 觉得，Redlock 必须「强依赖」多个节点的时钟是保持同步的，一旦有节点时钟发生错误，那这个算法模型就失效了。

即使 C 不是时钟跳跃，而是「崩溃后立即重启」，也会发生类似的问题。

Martin 继续阐述，机器的时钟发生错误，是很有可能发生的：

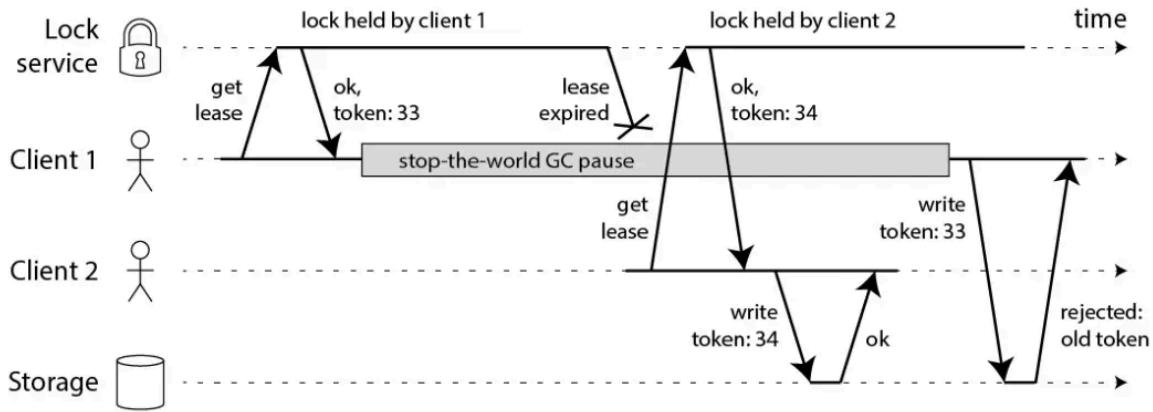
- 系统管理员「手动修改」了机器时钟
- 机器时钟在同步 NTP 时间时，发生了大的「跳跃」

总之，Martin 认为，Redlock 的算法是建立在「同步模型」基础上的，有大量资料研究表明，同步模型的假设，在分布式系统中是有问题的。在混乱的分布式系统的中，你不能假设系统时钟就是对的，所以，你必须非常小心你的假设。

4) 提出 fencing token 的方案，保证正确性

相对应的，Martin 提出一种被叫作 fencing token 的方案，保证分布式锁的正确性。这个模型流程如下：

1. 客户端在获取锁时，锁服务可以提供一个「递增」的 token
2. 客户端拿着这个 token 去操作共享资源
3. 共享资源可以根据 token 拒绝「后来者」的请求



这样一来，无论 NPC 哪种异常情况发生，都可以保证分布式锁的安全性，因为它是建立在「异步模型」上的。而 Redlock 无法提供类似 fencing token 的方案，所以它无法保证安全性。他还表示，**一个好的分布式锁，无论 NPC 怎么发生，可以不在规定时间内给出结果，但并不会给出一个错误的结果。也就是只会影响到锁的「性能」（或称之为活性），而不会影响它的「正确性」。**

Martin 的结论：

- 1、**Redlock 不伦不类**：它对于效率来讲，Redlock 比较重，没必要这么做，而对于正确性来说，Redlock 是不够安全的。
- 2、**时钟假设不合理**：该算法对系统时钟做出了危险的假设（假设多个节点机器时钟都是一致的），如果不满足这些假设，锁就会失效。
- 3、**无法保证正确性**：Redlock 不能提供类似 fencing token 的方案，所以解决不了正确性的问题。为了正确性，请使用有「共识系统」的软件，例如 Zookeeper。

Redlock对于问题的解释

1) 解释时钟问题

Redis 作者表示，Redlock 并不需要完全一致的时钟，只需要大体一致就可以了，允许有「误差」。例如要计时 5s，但实际可能记了 4.5s，之后又记了 5.5s，有一定误差，但只要不超过「误差范围」锁失效时间即可，这种对于时钟的精度的要求并不是很高，而且这也符合现实环境。

对于对方提到的「时钟修改」问题，Redis 作者反驳到：

1. **手动修改时钟**：不要这么做就好了，否则你直接修改 Raft 日志，那 Raft 也会无法工作...
2. **时钟跳跃**：通过「恰当的运维」，保证机器时钟不会大幅度跳跃（每次通过微小的调整来完成），实际上这是可以做到的

2) 解释网络延迟、GC 问题

之后，Redis 作者对于对方提出的，网络延迟 wan、进程 GC 可能导致 Redlock 失效的问题，也做了反驳：

在 Redlock 步骤 3，加锁成功后为什么要重新获取「当前时间戳 T2」？还用 $T2 - T1$ 的时间，与锁的过期时间做比较？

Redis 作者强调：如果在 1-3 发生了网络延迟、进程 GC 等耗时长的异常情况，那在第 3 步 $T2 - T1$ ，是可以检测出来的，如果超出了锁设置的过期时间，那这时就认为加锁会失败，之后释放所有节点的锁就好了！

Redis 作者继续论述，如果对方认为，发生网络延迟、进程 GC 是在步骤 3 之后，也就是客户端确认拿到了锁，去操作共享资源的途中发生了问题，导致锁失效，那这不止是 Redlock 的问题，任何其它锁服务例如 Zookeeper，都有类似的问题，这不在讨论范畴内。

这里我举个例子解释一下这个问题：

1. 客户端通过 Redlock 成功获取到锁（通过了大多数节点加锁成功、加锁耗时检查逻辑）
2. 客户端开始操作共享资源，此时发生网络延迟、进程 GC 等耗时很长的情况
3. 此时，锁过期自动释放
4. 客户端开始操作 MySQL（此时的锁可能会被别人拿到，锁失效）

Redis 作者这里的结论就是：

- 客户端在拿到锁之前，无论经历什么耗时长问题，Redlock 都能够在第 3 步检测出来
- 客户端在拿到锁之后，发生 NPC，那 Redlock、Zookeeper 都无能为力

所以，Redis 作者认为 Redlock 在保证时钟正确的基础之上，是可以保证正确性的。

3) 质疑 fencing token 机制

Redis 作者对于对方提出的 fencing token 机制，也提出了质疑，主要分为 2 个问题：

第一，这个方案必须要求要操作的「共享资源服务器」有拒绝「旧 token」的能力。

例如，要操作 MySQL，从锁服务拿到一个递增数字的 token，然后客户端要带着这个 token 去改 MySQL 的某一行，这就需要利用 MySQL 的「事务隔离性」来做。

```
// 两个客户端必须利用事务和隔离性达到目的
// 注意 token 的判断条件
UPDATE
    table T
SET
    val = $new_val, current_token = $token
WHERE
    id = $id AND current_token < $token
```

但如果操作的不是 MySQL 呢？例如向磁盘上写一个文件，或发起一个 HTTP 请求，那这个方案就无能为力了，这对要操作的资源服务器，提出了更高的要求。也就是说，大部分要操作的资源服务器，都是没有这种互斥能力的。**再者，既然资源服务器都有了「互斥」能力，那还要分布式锁干什么？**

第二，退一步讲，即使 Redlock 没有提供 fencing token 的能力，但 Redlock 已经提供了随机值（就是前面讲的 UUID），利用这个随机值，也可以达到与 fencing token 同样的效果。

1. 客户端使用 Redlock 拿到锁
2. 客户端在操作共享资源之前，先把这个锁的 VALUE，在要操作的共享资源上做标记
3. 客户端处理业务逻辑，最后，在修改共享资源时，判断这个标记是否与之前一样，一样才修改（类似 CAS 的思路）

还是以 MySQL 为例，举个例子就是这样的：

1. 客户端使用 Redlock 拿到锁
2. 客户端要修改 MySQL 表中的某一行数据之前，先把锁的 VALUE 更新到这一行的某个字段中（这里假设为 current_token 字段）
3. 客户端处理业务逻辑

4. 客户端修改 MySQL 的这一行数据，把 VALUE 当做 WHERE 条件，再修改

```
UPDATE
  table T
SET
  val = $new_val
WHERE
  id = $id AND current_token = $redlock_value
```

可见，这种方案依赖 MySQL 的事务机制，也达到对方提到的 fecing token 一样的效果。

但这里还有个小问题，是网友参与问题讨论时提出的：**两个客户端通过这种方案，先「标记」再「检查+修改」共享资源，那这两个客户端的操作顺序无法保证啊？** 而用 Martin 提到的 fecing token，因为这个 token 是单调递增的数字，资源服务器可以拒绝小的 token 请求，保证了操作的「顺序性」！Redis 作者对于这个问题做了不同的解释，我觉得很有道理，他解释道：**分布式锁的本质，是为了「互斥」，只要能保证两个客户端在并发时，一个成功，一个失败就好了，不需要关心「顺序性」。**

综上，Redis 作者的结论：

- 1、作者同意对方关于「时钟跳跃」对 Redlock 的影响，但认为时钟跳跃是可以避免的，取决于基础设施和运维。
- 2、Redlock 在设计时，充分考虑了 NPC 问题，在 Redlock 步骤 3 之前出现 NPC，可以保证锁的正确性，但在步骤 3 之后发生 NPC，不止是 Redlock 有问题，其它分布式锁服务同样也有问题，所以不在讨论范畴内。

基于zookeeper的分布式锁

zookeeper实现的分布式锁如下：

1. 客户端1和2都尝试创建临时节点，比如/lock
2. 客户端1先到达，则加锁成功
3. 客户端1操作共享资源
4. 客户端1删除/lock节点，释放锁

通过采用临时节点，只要客户端连接不断，就会一直持有锁，不会过期。并且如果客户端崩溃宕机了，这个临时节点也会自动删除，保证锁被释放。

由于客户端需要保持连接，才能持有锁。因此客户端会和zookeeper服务器维护一个session，这个 session 依赖客户端“定时心跳”来维持连接，如果 Zookeeper 长时间收不到客户端的心跳，就认为这个 Session 过期了，也会把这个临时节点删除。

同样地，基于此问题，我们也讨论一下 GC 问题对 Zookeeper 的锁有何影响：

1. 客户端 1 创建临时节点 /lock 成功，拿到了锁
2. 客户端 1 发生长时间 GC
3. 客户端 1 无法给 Zookeeper 发送心跳，Zookeeper 把临时节点「删除」
4. 客户端 2 创建临时节点 /lock 成功，拿到了锁
5. 客户端 1 GC 结束，它仍然认为自己持有锁（冲突）

可见，即使是使用 Zookeeper，也无法保证进程 GC、网络延迟异常场景下的安全性。

基于Etcd的分布式锁

基于Etcd的分布式锁如下：

1. 客户端1创建1个租约（设置过期时间）
2. 客户端1携带这个租约，创建/lock节点
3. 客户端1发现节点不存在，获得锁
4. 客户端2尝试创建节点，获取锁失败
5. 客户端1定时给租约续期，保持自己一直持有锁
6. 客户端1操作共享资源
7. 客户端1删除/lock节点，释放锁

定时给租约续期的步骤，和上面 Zookeeper 客户端定时给 Server 发心跳类似，其目的都是让服务端保持这个 Session 或 KV 持续有效。

所以，它依旧存在和 Zookeeper 相同的问题：

1. 客户端 1 创建节点 /lock 成功，拿到了锁
2. 客户端 1 发生长时间 GC
3. 客户端 1 无法向 Etcd 发请求给租约「续期」
4. 租约到期，Etcd 「删除」锁节点
5. 客户端 2 创建临时节点 /lock 成功，拿到了锁
6. 客户端 1 GC 结束，它仍然认为自己持有锁（冲突）

可见，基于 Etcd 实现的分布锁，当拿到锁发生 GC、网络延迟问题，依旧可能失效。

至此，这里我们可以得出结论：**一个分布式锁，无论是基于 Redis 还是 Zookeeper、Etcd 实现，在极端情况下，都无法保证 100% 安全，都存在失效的可能。**如果你的业务数据非常敏感，在使用分布式锁时，一定要注意这个问题，不能假设分布式锁 100% 安全。

到底要不要用Redlock

1) 到底要不要用 Redlock？

前面也分析了，Redlock 只有建立在「时钟正确」的前提下，才能正常工作，如果你可以保证这个前提，那么可以拿来使用。

但保证时钟正确，并不是你想的那么简单就能做到的。

第一，从硬件角度来说，时钟发生偏移是时有发生，无法避免的。例如，CPU 温度、机器负载、芯片材料都是有可能导致时钟发生偏移。

第二，人为错误也是很难完全避免的。

对于 Redlock，尽量不用它，而且它的性能不如单机版 Redis，部署成本也高，我还是会优先考虑使用 Redis 「主从+哨兵」的模式，实现分布式锁。

那正确性如何保证呢？第二点给你答案。

2) 如何正确使用分布式锁？

在分析 Martin 观点时，它提到了 fencing token 的方案，给我了很大的启发，虽然这种方案有很大的局限性，但对于保证「正确性」的场景，是一个非常好的思路。

所以，我们可以把这两者结合起来用：

- 1、使用分布式锁，在上层完成「互斥」目的，虽然极端情况下锁会失效，但它可以最大程度把并发请求阻挡在最上层，减轻操作资源层的压力。
- 2、但对于要求数据绝对正确的业务，在资源层一定要做好「兜底」，设计思路可以借鉴 `fencing token` 的方案来做，即在 DB 层通过版本号的方式来更新数据，避免并发冲突。

Redis数据结构及其底层数据结构详解

简单动态字符串

简单动态字符串是redis基础数据结构string的底层实现，相较于C语言传统的字符串char *（不能包含空字符），不以'\0'来标识字符串的结束，可以存储任何二进制数据，包括空字符。

```
struct sdshdr {
    // 记录buf数组中已使用字节的数量，等于sds所保存字符串的长度
    int len;

    // 记录buf数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    char buf[];
}
```

优点：

1. 获取字符串长度的时间复杂度为O(1)
2. 避免缓冲区溢出
3. 二进制安全，传统的 C 字符串依赖于空字符 '\0' 来标记字符串的结束，这限制了字符串中不能包含空字符。与之相对，SDS 由于存储了字符串的实际长度，因此可以安全地包含任何二进制数据，包括空字符
4. 字符串扩展时预分配内存，减少内存分配的次数
5. C字符串函数兼容

双向链表

双向链表是Redis基础数据结构列表list的底层实现，当列表元素较多或元素长度较长时，列表使用双向链表做为其底层实现。

```
// 节点
typedef struct ListNode {
    // 前置节点
    struct ListNode *prev;
    // 后置节点
    struct ListNode *next;
    // 节点的值
    void *value;
} ListNode;

// list
typedef struct List {
    // 表头节点
    ListNode *head;
    // 表尾节点
    ListNode *tail;
    // 链表所包含的节点数量
    unsigned long len;
    // 节点值复制函数
    void *(*dup)(void *ptr);
```

```

// 节点值释放函数
void (*free)(void *ptr);
// 节点值对比函数
int (*match) (void *ptr, void *key);
} list;

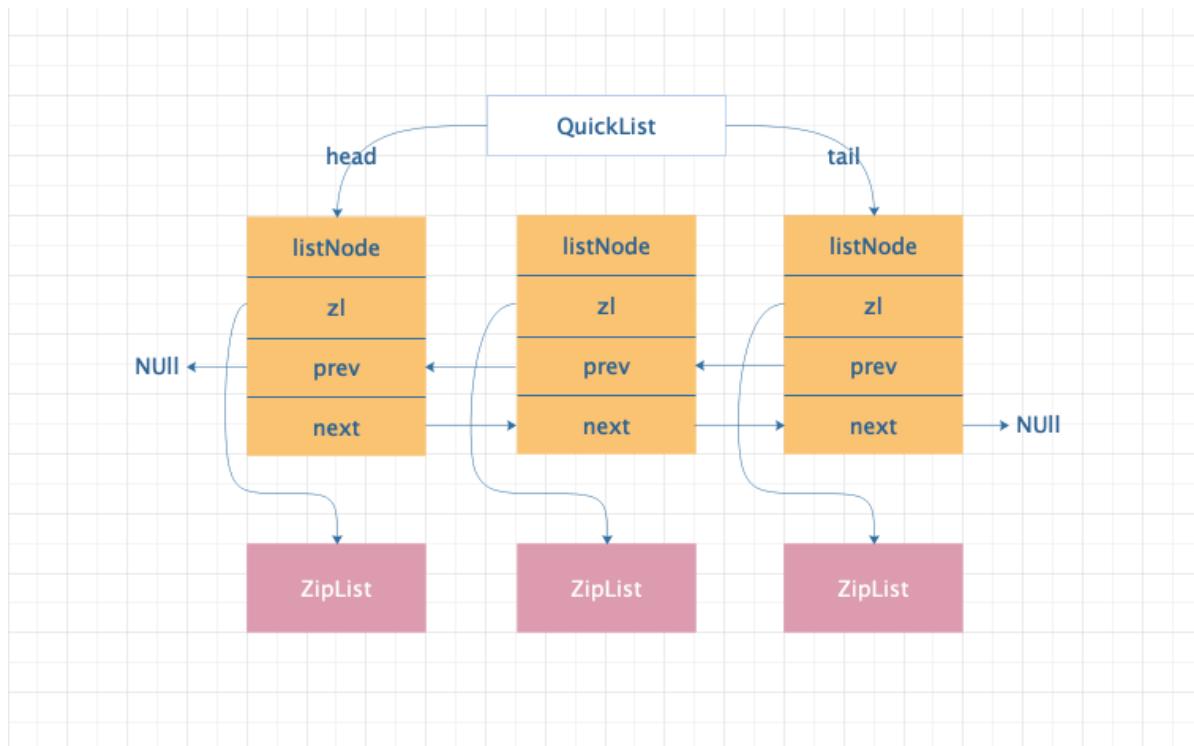
```

1. list的表头节点和表尾节点都指向null，为无环链表
2. 通过函数指针，链表可以对不同类型的值进行操作

QuickList

在 Redis3.2 之前，List 底层采用了 ZipList 和 LinkedList 实现的，在 3.2 之后，List 底层采用了 QuickList。Redis3.2 之前，初始化的 List 使用的 ZipList，List 满足以下两个条件时则一直使用 ZipList 作为底层实现，当以下两个条件任一个不满足时，则会被转换成 LinkedList。

- List 中存储的每个元素的长度小于 64byte
- 元素个数小于 512



Redis 集合采用了 QuickList 作为 List 的底层实现，QuickList 其实就是结合了 ZipList 和 LinkedList 的优点设计出来的。各部分作用说明：

- 每个 listNode 存储一个指向 ZipList 的指针，ZipList 用来真正存储元素的数据。
- ZipList 中存储的元素数据总大小超过 8kb（默认大小，通过 list-max-ziplist-size 参数可以进行配置）的时候，就会重新创建出来一个 ListNode 和 ZipList，然后将其通过指针关联起来。

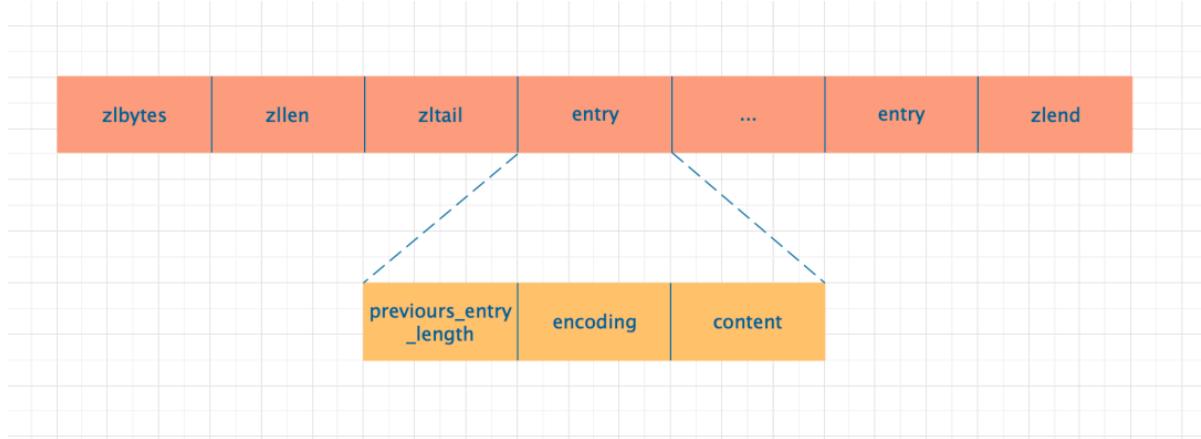
哈希表

哈希表是Redis基础数据结构列表hash的底层实现，当哈希结构的大小或字段值长度超过一定阈值时，使用哈希表做为其底层实现。

1. redis的哈希表使用链地址法来解决哈希冲突，每个哈希表节点都有个next指针，多个哈希表节点可以用这个单向链表连接起来

压缩链表

当列表、哈希、有序集合的长度较小并且元素长度较小时，使用压缩链表做为其底层实现。



ZipList 是由一块连续的存储空间组成，从图中可以看出 ZipList 没有前后指针。各部分作用说明：

- **`zbytes`**: 表示当前 list 的存储元素的总长度。
- **`zllen`**: 表示当前 list 存储的元素的个数。
- **`zltail`**: 表示当前 list 的头结点的地址，通过 `zltail` 就是可以实现 list 的遍历。
- **`zlend`**: 表示当前 list 的结束标识。
- **`entry`**: 表示存储实际数据的节点，每个 `entry` 代表一个元素。
 - **`previous_entry_length`**: 表示当前节点元素的长度，通过其长度可以计算出下一个元素的位置。
 - **`encoding`**: 表示元素的编码格式。
 - **`content`**: 表示实际存储的元素内容。

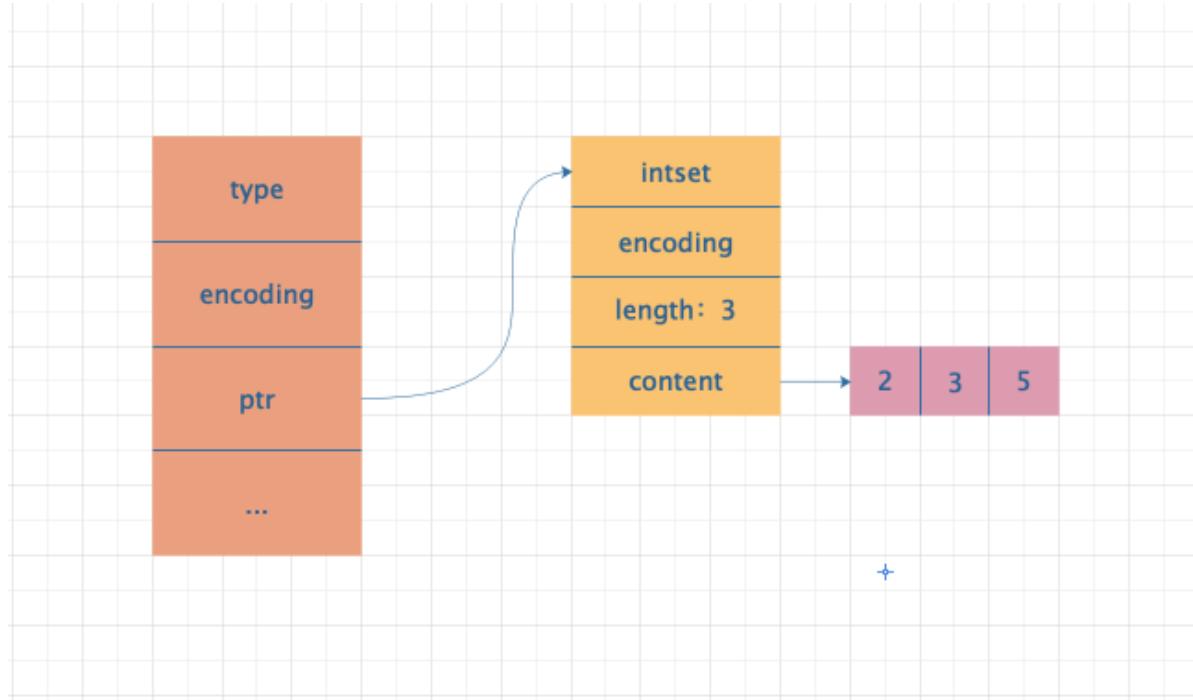
ZipList的优缺点：

- **优点：**内存地址连续，省去了每个元素的头尾节点指针占用的内存。
- **缺点：**对于删除和插入操作比较可能会触发连锁更新反应，比如在 list 中间插入删除一个元素时，在插入或删除位置后面的元素可能都需要发生相应的移动操作。

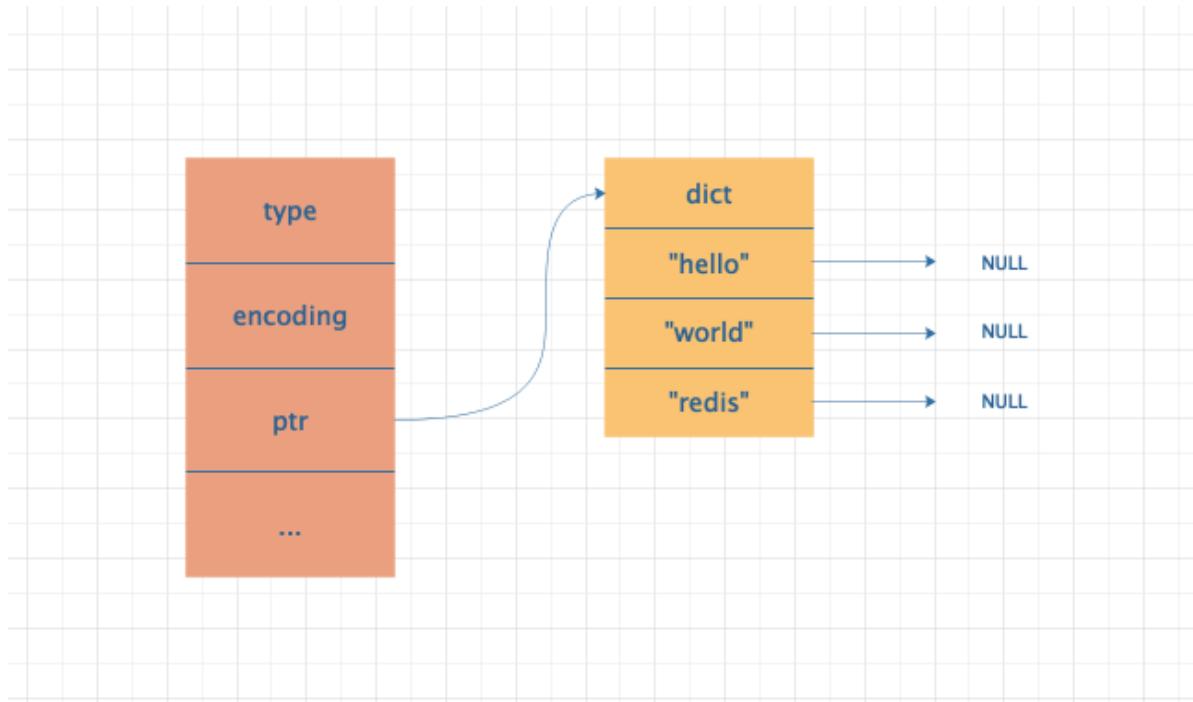
Set 的实现原理

Set 集合采用了整数集合和字典两种方式来实现的，当满足如下两个条件的时候，采用整数集合实现；一旦有一个条件不满足时则采用字典来实现。

- **Set 集合中的所有元素都为整数**
- **Set 集合中的元素个数不大于 512 (默认 512，可以通过修改 `set-max-intset-entries` 配置调整集合大小)**



整数集合实现原理图



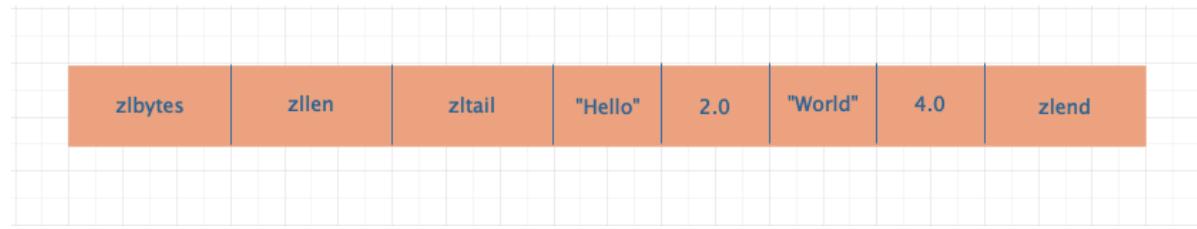
字典实现原理图

Zset 的实现原理

Zset 底层同样采用了两种方式来实现，分别是 ZipList 和 SkipList。当同时满足以下两个条件时，采用 ZipList 实现；反之采用 SkipList 实现。

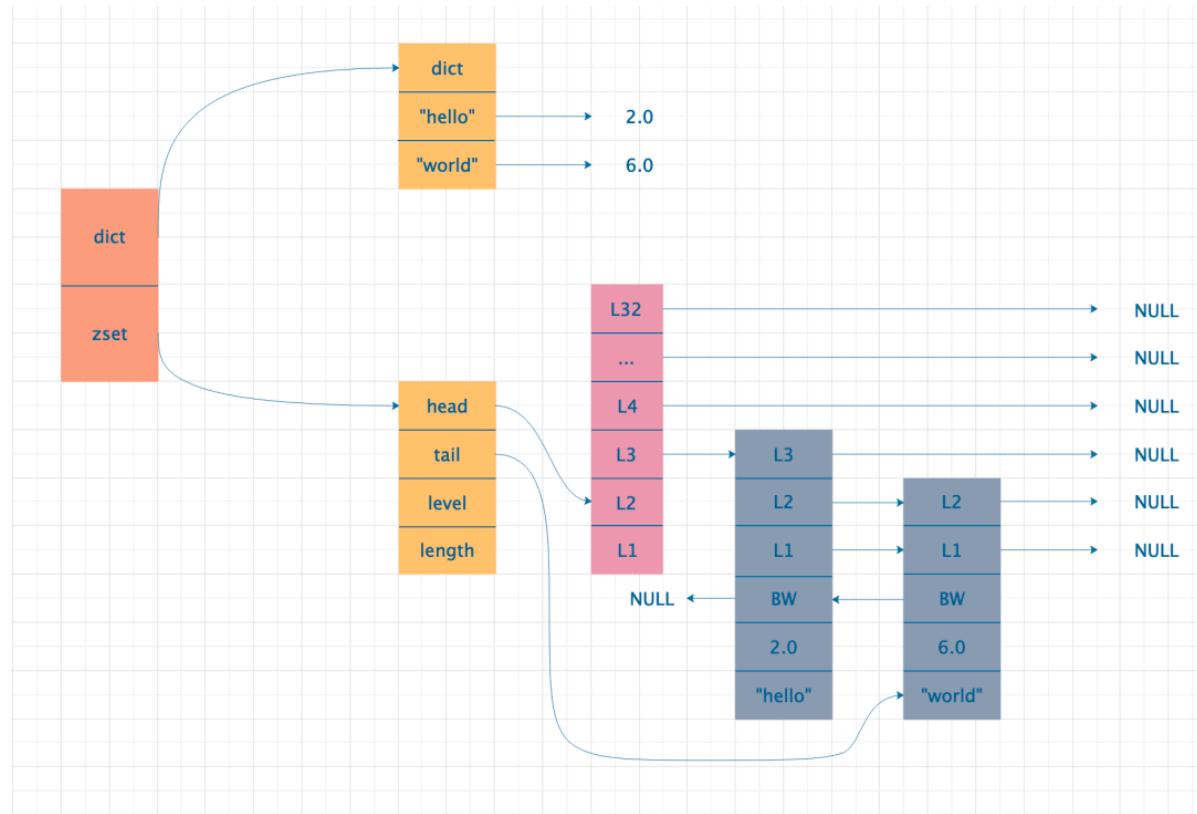
- Zset 中保存的元素个数小于 128。 (通过修改 `zset-max-ziplist-entries` 配置来修改)
- Zset 中保存的所有元素长度小于 64byte。 (通过修改 `zset-max-ziplist-values` 配置来修改)

采用 ZipList 的实现原理



和 List 的底层实现有些相似，对于 Zset 不同的是，其存储是以键值对的方式依次排列，键存储的是实际 value，值存储的是 value 对应的分值。

采用 SkipList 的实现原理



SkipList 分为两部分，dict 部分是由字典实现，Zset 部分使用跳跃表实现，从图中可以看出，dict 和跳跃表都存储的数据，实际上 dict 和跳跃表最终使用指针都指向了同一份数据，即数据是被两部分共享的，为了方便表达将同一份数据展示在两个地方。

C++的Map也是一种缓存型数据结构，为什么不用Map，而选择Redis做缓存

1. **分布式访问**: Redis 是一个独立的服务器，可以被网络中的多个应用同时访问，支持分布式系统，而 C++ 的 Map 是进程内的数据结构，不支持网络分布式访问。
2. **持久性**: Redis 支持数据持久化，可以将内存中的数据保存到硬盘，而 C++ 的 Map 默认是内存中的结构，重启应用数据会丢失。
3. **内置功能**: Redis 提供了丰富的数据结构和操作，如过期策略、发布订阅、事务等，而 C++ 的 Map 功能相对基础。
4. **性能优化**: Redis 作为专门的缓存服务，进行了大量的性能优化，能够处理高并发请求，而 C++ 的 Map 性能受限于单个应用的处理能力。

缓存中的热数据和冷数据

1. **热数据**指的是经常被访问的数据，这部分数据的访问频率高，因此保持在缓存中可以显著提高系统的响应速度和效率。
2. **冷数据**则是相对较少被访问的数据，访问频率低，对性能的直接影响较小。

Redis数据类型使用场景

Redis 五种数据类型的应用场景：

- String 类型的应用场景：缓存对象、常规计数、分布式锁、共享 session 信息等。
- List 类型的应用场景：消息队列（但是有两个问题：1. 生产者需要自行实现全局唯一 ID；2. 不能以消费组形式消费数据）等。
- Hash 类型：缓存对象、购物车等。
- Set 类型：聚合计算（并集、交集、差集）场景，比如点赞、共同关注、抽奖活动等。
- Zset 类型：排序场景，比如排行榜、电话和姓名排序等。

Redis 后续版本又支持四种数据类型，它们的应用场景如下：

- BitMap (2.2 版新增)：二值状态统计的场景，比如签到、判断用户登陆状态、连续签到用户总数等；
- HyperLogLog (2.8 版新增)：海量数据基数统计的场景，比如百万级网页 UV 计数等；
- GEO (3.2 版新增)：存储地理位置信息的场景，比如滴滴叫车；
- Stream (5.0 版新增)：消息队列，相比于基于 List 类型实现的消息队列，有这两个特有的特性：自动生成全局唯一消息ID，支持以消费组形式消费数据。

Redis对设置了过期时间的数据采用定期删除和惰性删除，一定能保证删除数据吗

不能，如果定期删除没有删除掉，并且也没有去访问这个数据，就不会被删除，并且一直占着内存。

但redis有内存淘汰机制：

1. 新写入操作直接报错
2. LRU (最常用)
3. 随机移除几个key
4. 在所有的key中LRU
5. 在所有的key中LFU
6. 在设置了过期时间的key中， LRU

7. 在设置了过期时间的key中，LFU
8. 在设置了过期时间的key中，随机移除几个
9. 移除快过期的key

Redis 是单线程吗？

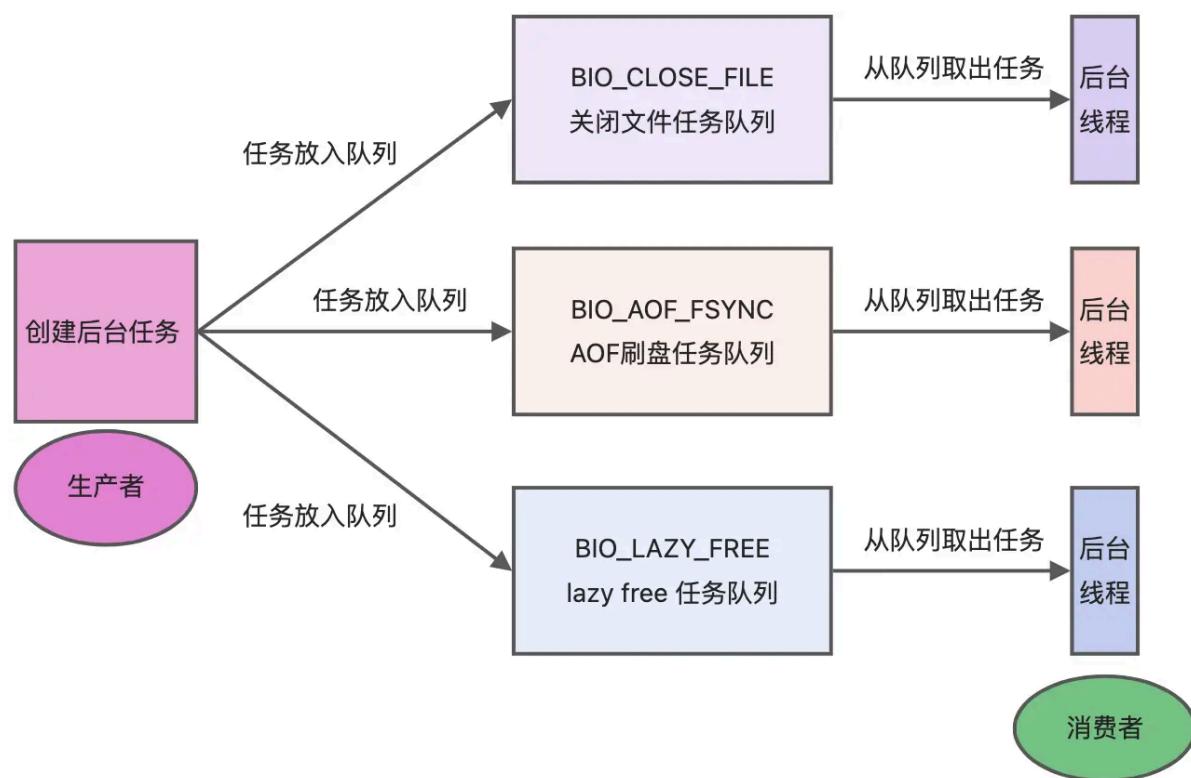
Redis 单线程指的是「接收客户端请求->解析请求->进行数据读写等操作->发送数据给客户端」这个过程是由一个线程（主线程）来完成的，这也是我们常说 Redis 是单线程的原因。

但是，Redis 程序并不是单线程的，Redis 在启动的时候，是会启动后台线程（BIO）的：

- Redis 在 2.6 版本，会启动 2 个后台线程，分别处理关闭文件、AOF 刷盘这两个任务；
- Redis 在 4.0 版本之后，新增了一个新的后台线程，用来异步释放 Redis 内存，也就是 lazyfree 线程。例如执行 unlink key / flushdb async / flushall async 等命令，会把这些删除操作交给后台线程来执行，好处是不会导致 Redis 主线程卡顿。因此，当我们要删除一个大 key 的时候，不要使用 del 命令删除，因为 del 是在主线程处理的，这样会导致 Redis 主线程卡顿，因此我们应该使用 unlink 命令来异步删除大key。

之所以 Redis 为「关闭文件、AOF 刷盘、释放内存」这些任务创建单独的线程来处理，是因为这些任务的操作都是很耗时的，如果把这些任务都放在主线程来处理，那么 Redis 主线程就很容易发生阻塞，这样就无法处理后续的请求了。

后台线程相当于一个消费者，生产者把耗时任务丢到任务队列中，消费者（BIO）不停轮询这个队列，拿出任务就去执行对应的方法即可。



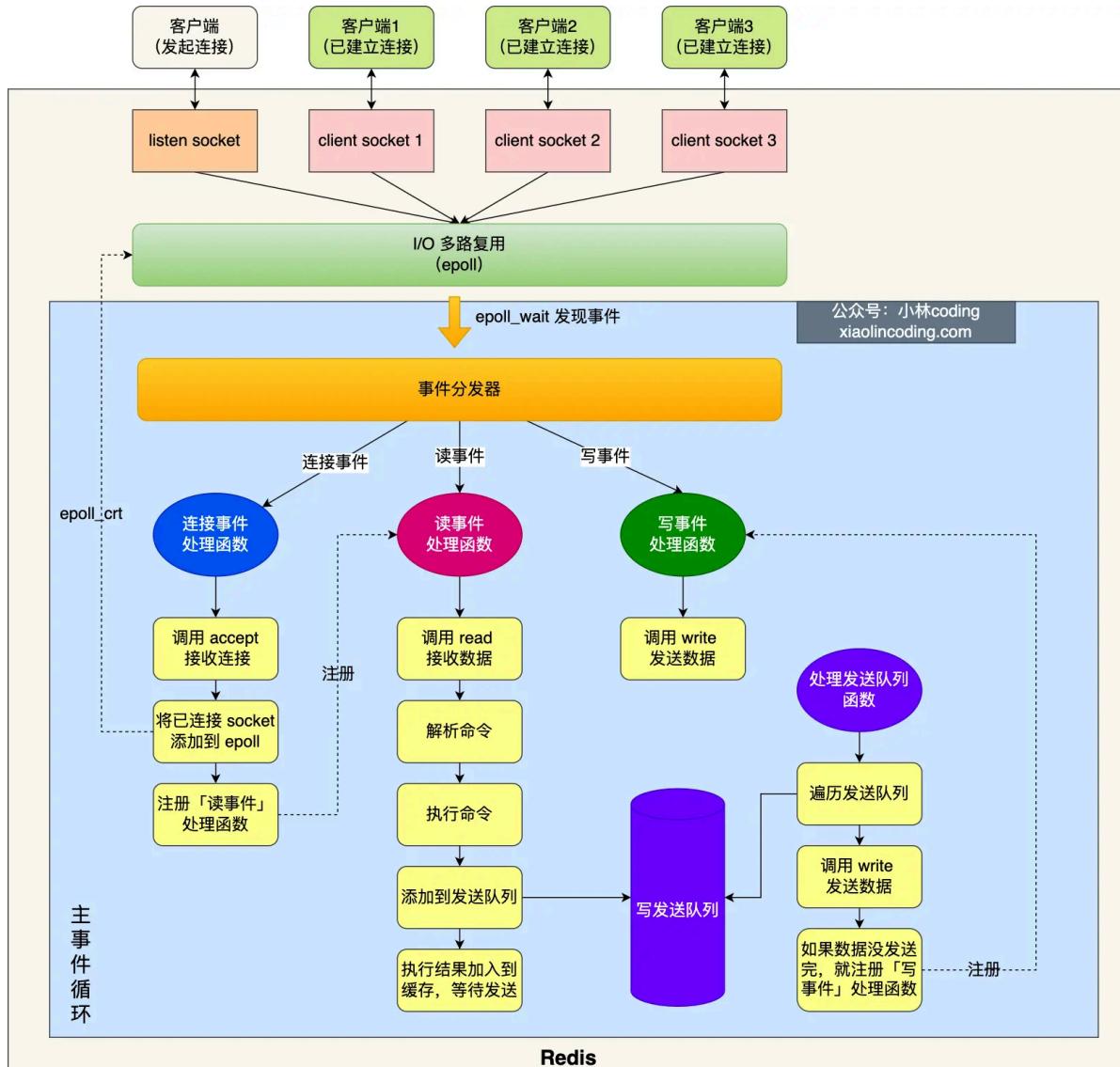
关闭文件、AOF 刷盘、释放内存这三个任务都有各自的任务队列：

- **BIO_CLOSE_FILE**，关闭文件任务队列：当队列有任务后，后台线程会调用 close(fd)，将文件关闭；
- **BIO_AOF_FSYNC**，AOF刷盘任务队列：当 AOF 日志配置成 everysec 选项后，主线程会把 AOF 写日志操作封装成一个任务，也放到队列中。当发现队列有任务后，后台线程会调用 fsync(fd)，将 AOF 文件刷盘，

- BIO_LAZY_FREE, lazy free 任务队列：当队列有任务后，后台线程会 free(obj) 释放对象 / free(dict) 删除数据库所有对象 / free(skiplist) 释放跳表对象；

Redis 单线程模式是怎样的？

Redis 6.0 版本之前的单线程模式如下图：



图中的蓝色部分是一个事件循环，是由主线程负责的，可以看到网络 I/O 和命令处理都是单线程。Redis 初始化的时候，会做下面这几件事情：

- 首先，调用 `epoll_create()` 创建一个 epoll 对象和调用 `socket()` 创建一个服务端 socket
- 然后，调用 `bind()` 绑定端口和调用 `listen()` 监听该 socket；
- 然后，将调用 `epoll_ctl()` 将 listen socket 加入到 epoll，同时注册「连接事件」处理函数。

初始化完后，主线程就进入到一个**事件循环函数**，主要会做以下事情：

- 首先，先调用**处理发送队列函数**，看是发送队列里是否有任务，如果有发送任务，则通过 `write` 函数将客户端发送缓存区里的数据发送出去，如果这一轮数据没有发送完，就会注册写事件处理函数，等待 `epoll_wait` 触发可写后再处理。
- 接着，调用 `epoll_wait` 函数等待事件的到来：
 - 如果是**连接事件**到来，则会调用**连接事件处理函数**，该函数会做这些事情：调用 `accpet` 获取已连接的 socket -> 调用 `epoll_ctl` 将已连接的 socket 加入到 epoll -> 注册「读事件」处理函数；

- 如果是读事件到来，则会调用**读事件处理函数**，该函数会做这些事情：调用 read 获取客户端发送的数据 -> 解析命令 -> 处理命令 -> 将客户端对象添加到发送队列 -> 将执行结果写到发送缓存区等待发送；
- 如果是写事件到来，则会调用**写事件处理函数**，该函数会做这些事情：通过 write 函数将客户端发送缓存区里的数据发送出去，如果这一轮数据没有发送完，就会继续注册写事件处理函数，等待 epoll_wait 发现可写后再处理。

Redis 6.0 之后为什么引入了多线程？

虽然 Redis 的主要工作（网络 I/O 和执行命令）一直是单线程模型，但是在 Redis 6.0 版本之后，也采用了多个 I/O 线程来处理网络请求，这是因为随着网络硬件的性能提升，Redis 的性能瓶颈有时会出现在网络 I/O 的处理上。

所以为了提高网络 I/O 的并行度，Redis 6.0 对于网络 I/O 采用多线程来处理。**但是对于命令的执行，Redis 仍然使用单线程来处理，所以大家不要误解 Redis 有多线程同时执行命令。**

Redis 官方表示，Redis 6.0 版本引入的多线程 I/O 特性对性能提升至少是一倍以上。

Redis 6.0 版本支持的 I/O 多线程特性，默认情况下 I/O 多线程只针对发送响应数据（write client socket），并不会以多线程的方式处理读请求（read client socket）。要想开启多线程处理客户端读请求，就需要把 Redis.conf 配置文件中的 io-threads-do-reads 配置项设为 yes。

```
// 读请求也使用io多线程
io-threads-do-reads yes
```

同时，Redis.conf 配置文件中提供了 IO 多线程个数的配置项。

```
// io-threads N, 表示启用 N-1 个 I/O 多线程（主线程也算一个 I/O 线程）
io-threads 4
```

关于线程数的设置，官方的建议是如果为 4 核的 CPU，建议线程数设置为 2 或 3，如果为 8 核 CPU 建议线程数设置为 6，线程数一定要小于机器核数，线程数并不是越大越好。因此，Redis 6.0 版本之后，Redis 在启动的时候，默认情况下会额外创建 6 个线程（这里的线程数不包括主线程）：

- **Redis-server**：Redis 的主线程，主要负责执行命令；
- **bio_close_file、bio_aof_fsync、bio_lazy_free**：三个后台线程，分别异步处理关闭文件任务、AOF 刷盘任务、释放内存任务；
- **io_thd_1、io_thd_2、io_thd_3**：三个 I/O 线程，io-threads 默认是 4，所以会启动 3 (4-1) 个 I/O 多线程，用来分担 Redis 网络 I/O 的压力。

集群脑裂导致数据丢失怎么办？

什么是脑裂？

先来理解集群的脑裂现象，这就好比一个人有两个大脑，那么到底受谁控制呢？

那么在 Redis 中，集群脑裂产生数据丢失的现象是怎样的呢？

在 Redis 主从架构中，部署方式一般是「一主多从」，主节点提供写操作，从节点提供读操作。**如果主节点的网络突然发生了问题，它与所有的从节点都失联了，但是此时的主节点和客户端的网络是正常的，这个客户端并不知道 Redis 内部已经出现了问题，还在照常的向这个失联的主节点写数据（过程 A），此时这些数据被旧主节点缓存到了缓冲区里，因为主从节点之间的网络问题，这些数据都是无法同步给从节点的。**

这时，哨兵也发现主节点失联了，它就认为主节点挂了（但实际上主节点正常运行，只是网络出问题了），于是哨兵就会在「从节点」中选举出一个 leader 作为主节点，这时集群就有两个主节点了——脑裂出现了。

然后，网络突然好了，哨兵因为之前已经选举出一个新主节点了，它就会把旧主节点降级为从节点 (A)，然后从节点 (A) 会向新主节点请求数据同步，因为第一次同步是全量同步的方式，此时的从节点 (A) 会清空掉自己本地的数据，然后再做全量同步。所以，之前客户端在过程 A 写入的数据就会丢失了，也就是集群产生脑裂数据丢失的问题。

总结一句话就是：由于网络问题，集群节点之间失去联系。主从数据不同步；重新平衡选举，产生两个主服务。等网络恢复，旧主节点会降级为从节点，再与新主节点进行同步复制的时候，由于从节点会清空自己的缓冲区，所以导致之前客户端写入的数据丢失了。

解决方案

当主节点发现从节点下线或者通信超时的总数量小于阈值时，那么禁止主节点进行写数据，直接把错误返回给客户端。

在 Redis 的配置文件中有两个参数我们可以设置：

- min-slaves-to-write x，主节点必须要有至少 x 个从节点连接，如果小于这个数，主节点会禁止写数据。
- min-slaves-max-lag x，主从数据复制和同步的延迟不能超过 x 秒，如果超过，主节点会禁止写数据。

我们可以把 min-slaves-to-write 和 min-slaves-max-lag 这两个配置项搭配起来使用，分别给它们设置一定的阈值，假设为 N 和 T。

这两个配置项组合后的前提是，主库连接的从库中至少有 N 个从库，和主库进行数据复制时的 ACK 消息延迟不能超过 T 秒，否则，主库就不会再接收客户端的写请求了。即使原主库是假故障，它在假故障期间也无法响应哨兵心跳，也不能和从库进行同步，自然也就无法和从库进行 ACK 确认了。这样一来，min-slaves-to-write 和 min-slaves-max-lag 的组合要求就无法得到满足，**原主库就会被限制接收客户端写请求，客户端也就不能在原主库中写入新数据了。**

等到新主库上线时，就只有新主库能接收和处理客户端请求，此时，新写的数据会被直接写到新主库中。而原主库会被哨兵降为从库，即使它的数据被清空了，也不会有新数据丢失。

再来举个例子。假设我们将 min-slaves-to-write 设置为 1，把 min-slaves-max-lag 设置为 12s，把哨兵的 down-after-milliseconds 设置为 10s，主库因为某些原因卡住了 15s，导致哨兵判断主库客观下线，开始进行主从切换。同时，因为原主库卡住了 15s，没有一个从库能和原主库在 12s 内进行数据复制，原主库也无法接收客户端请求了。这样一来，主从切换完成后，也只有新主库能接收请求，不会发生脑裂，也就不会发生数据丢失的问题了。

Redis 对过期 key 定期删除的详细流程

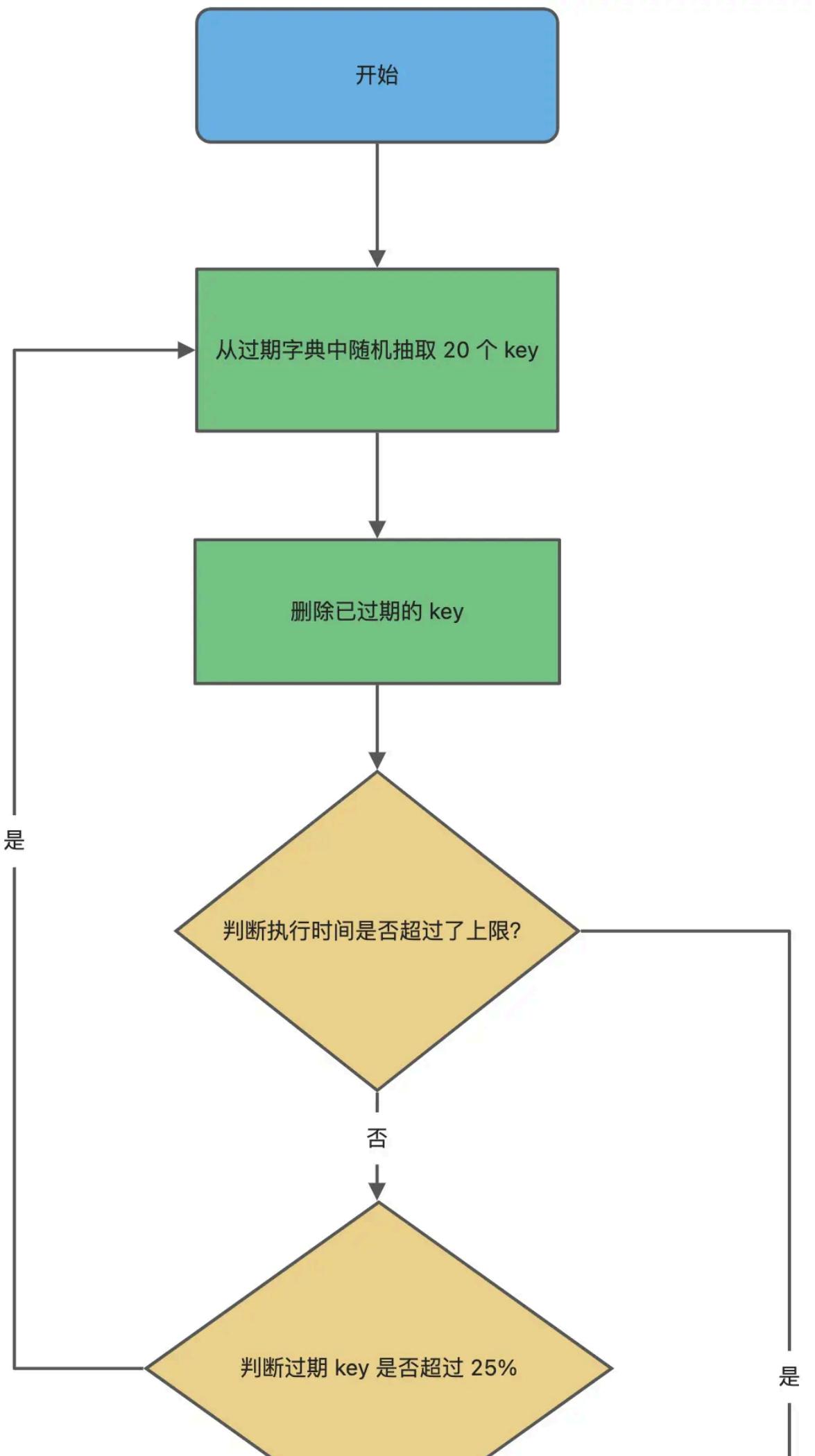
定期删除策略的做法是，每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期 key。

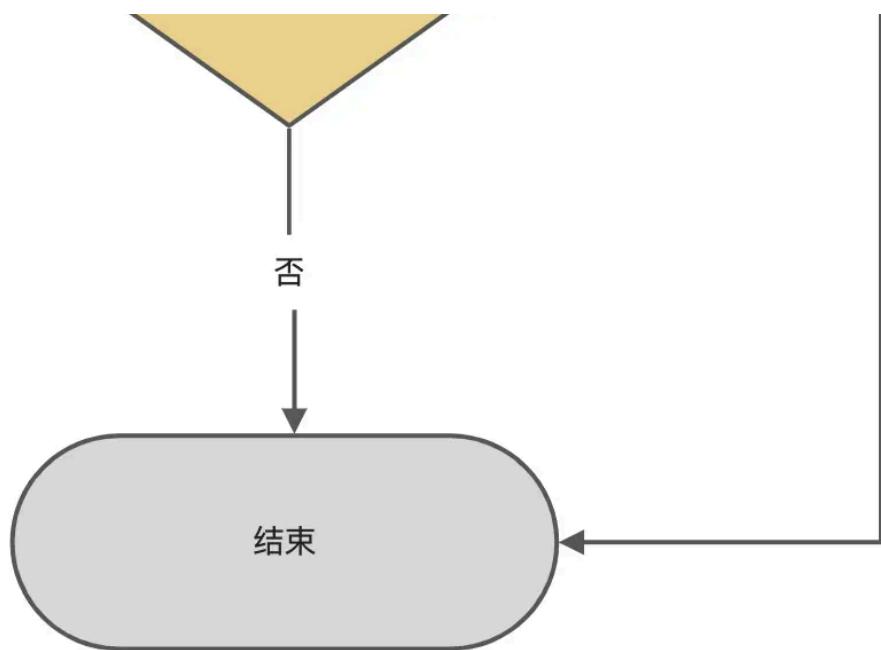
Redis 的定期删除的流程：

1. 从过期字典中随机抽取 20 个 key；
2. 检查这 20 个 key 是否过期，并删除已过期的 key；
3. 如果本轮检查的已过期 key 的数量，超过 5 个 (20/4)，也就是「已过期 key 的数量」占比「随机抽取 key 的数量」大于 25%，则继续重复步骤 1；如果已过期的 key 比例小于 25%，则停止继续删除过期 key，然后等待下一轮再检查。

可以看到，定期删除是一个循环的流程。那 Redis 为了保证定期删除不会出现循环过度，导致线程卡死现象，为此增加了定期删除循环流程的时间上限，默认不会超过 25ms。

定期删除的流程如下：





定期删除策略的优点：

- 通过限制删除操作执行的时长和频率，来减少删除操作对 CPU 的影响，同时也能删除一部分过期的数据减少了过期键对空间的无效占用。

定期删除策略的缺点：

- 难以确定删除操作执行的时长和频率。如果执行的太频繁，就会对 CPU 不友好；如果执行的太少，那又和惰性删除一样了，过期 key 占用的内存不会及时得到释放。

Redis 持久化时，对过期键会如何处理的？

Redis 持久化文件有两种格式：RDB（Redis Database）和 AOF（Append Only File），下面我们分别来看过期键在这两种格式中的呈现状态。

RDB 文件分为两个阶段，RDB 文件生成阶段和加载阶段。

- **RDB 文件生成阶段：**从内存状态持久化成 RDB（文件）的时候，会对 key 进行过期检查，**过期的键「不会」被保存到新的 RDB 文件中**，因此 Redis 中的过期键不会对生成新 RDB 文件产生任何影响。
- RDB 加载阶段，要看服务器是主服务器还是从服务器，分别对应以下两种情况：
 - 如果 Redis 是「主服务器」运行模式的话，在载入 RDB 文件时，程序会对文件中保存的键进行检查，**过期键「不会」被载入到数据库中**。所以过期键不会对载入 RDB 文件的主服务器造成影响；
 - 如果 Redis 是「从服务器」运行模式的话，在载入 RDB 文件时，**不论键是否过期都会被载入到数据库中**。但由于主从服务器在进行数据同步时，从服务器的数据会被清空。所以一般来说，过期键对载入 RDB 文件的从服务器也不会造成影响。

AOF 文件分为两个阶段，AOF 文件写入阶段和 AOF 重写阶段。

- **AOF 文件写入阶段：**当 Redis 以 AOF 模式持久化时，**如果数据库某个过期键还没被删除，那么 AOF 文件会保留此过期键，当此过期键被删除后，Redis 会向 AOF 文件追加一条 DEL 命令来显式地删除该键值。**
- **AOF 重写阶段：**执行 AOF 重写时，会对 Redis 中的键值对进行检查，**已过期的键不会被保存到重写后的 AOF 文件中**，因此不会对 AOF 重写造成任何影响。

Redis 主从模式中，对过期键会如何处理？

当 Redis 运行在主从模式下时，**从库不会进行过期扫描，从库对过期的处理是被动的。**也就是即使从库中的 key 过期了，如果有客户端访问从库时，依然可以得到 key 对应的值，像未过期的键值对一样返回。

从库的过期键处理依靠主服务器控制，**主库在 key 到期时，会在 AOF 文件里增加一条 del 指令，同步到所有的从库**，从库通过执行这条 del 指令来删除过期的 key。

Redis中的LRU和LFU

LRU 全称是 Least Recently Used 翻译为**最近最少使用**，会选择淘汰最近最少使用的数据。传统 LRU 算法的实现是基于「链表」结构，链表中的元素按照操作顺序从前往后排列，最新操作的键会被移动到表头，当需要内存淘汰时，只需要删除链表尾部的元素即可，因为链表尾部的元素就代表最久未被使用的元素。

Redis 并没有使用这样的方式实现 LRU 算法，因为传统的 LRU 算法存在两个问题：

- 需要用链表管理所有的缓存数据，这会带来额外的空间开销；
- 当有数据被访问时，需要在链表上把该数据移动到头端，如果有大量数据被访问，就会带来很多链表移动操作，会很耗时，进而会降低 Redis 缓存性能。

Redis 是如何实现 LRU 算法的？

Redis 实现的是一种**近似 LRU 算法**，目的是为了更好的节约内存，它的**实现方式是在 Redis 的对象结构体中添加一个额外的字段，用于记录此数据的最后一次访问时间**。

当 Redis 进行内存淘汰时，会使用**随机采样的方式来淘汰数据**，它是随机取 5 个值（此值可配置），然后淘汰最久没有使用那个。

Redis 实现的 LRU 算法的优点：

- 不用为所有的数据维护一个大链表，节省了空间占用；
- 不用在每次数据访问时都移动链表项，提升了缓存的性能；

但是 LRU 算法有一个问题，**无法解决缓存污染问题**，比如应用一次读取了大量的数据，而这些数据只会被读取这一次，那么这些数据会留存在 Redis 缓存中很长一段时间，造成缓存污染。

因此，在 Redis 4.0 之后引入了 LFU 算法来解决这个问题。

什么是 LFU 算法？

LFU 全称是 Least Frequently Used 翻译为**最近最不常用的**，LFU 算法是根据数据访问次数来淘汰数据的，它的核心思想是“如果数据过去被访问多次，那么将来被访问的频率也更高”。

所以，LFU 算法会记录每个数据的访问次数。当一个数据被再次访问时，就会增加该数据的访问次数。这样就解决了偶尔被访问一次之后，数据留存在缓存中很长一段时间的问题，相比于 LRU 算法也更合理一些。

Redis 是如何实现 LFU 算法的？

LFU 算法相比于 LRU 算法的实现，多记录了「数据的访问频次」的信息。Redis 对象的结构如下：

```
typedef struct redisObject {  
    ...  
  
    // 24 bits, 用于记录对象的访问信息  
    unsigned lru:24;  
    ...  
} robj;
```

Redis 对象头中的 lru 字段，在 LRU 算法下和 LFU 算法下使用方式并不相同。

在 LRU 算法中，Redis 对象头的 24 bits 的 lru 字段是用来记录 key 的访问时间戳，因此在 LRU 模式下，Redis 可以根据对象头中的 lru 字段记录的值，来比较最后一次 key 的访问时间长，从而淘汰最久未被使用的 key。

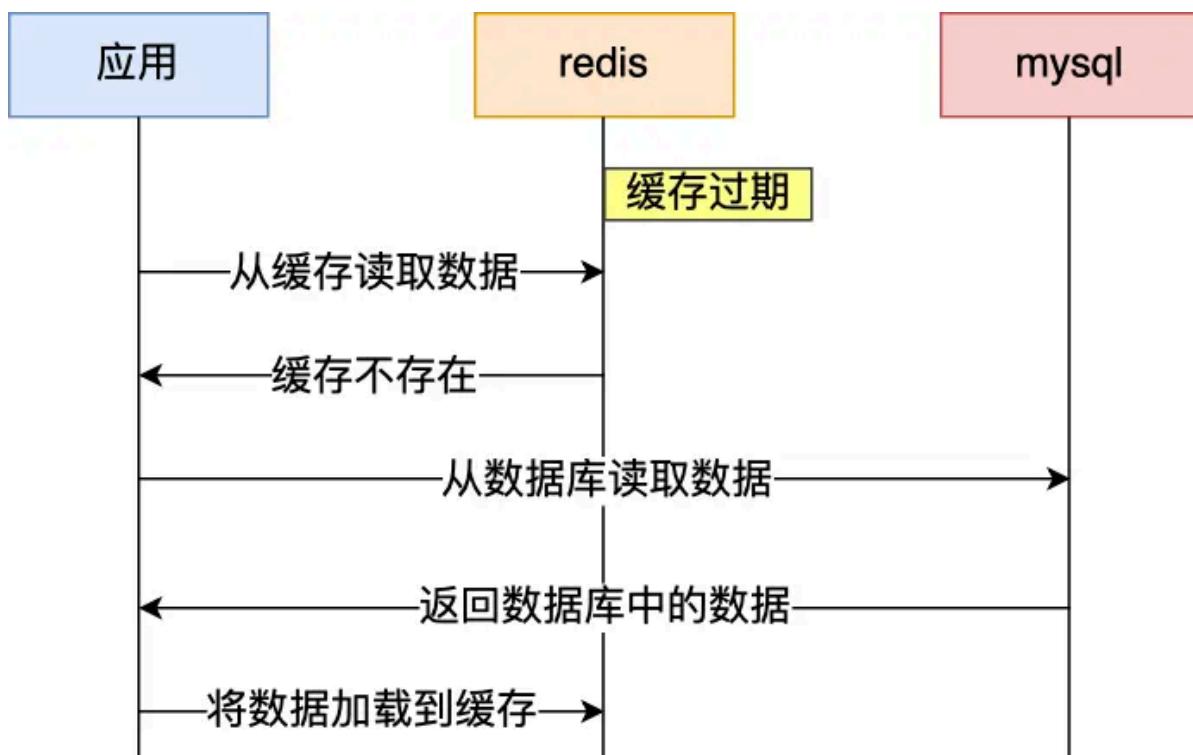
在 LFU 算法中，Redis 对象头的 24 bits 的 lru 字段被分成两段来存储，高 16bit 存储 ldt(Last Decrement Time)，用来记录 key 的访问时间戳；低 8bit 存储 logc(Logistic Counter)，用来记录 key 的访问频次。



如何避免缓存雪崩、缓存击穿、缓存穿透？

如何避免缓存雪崩？

通常我们为了保证缓存中的数据与数据库中的数据一致性，会给 Redis 里的数据设置过期时间，当缓存数据过期后，用户访问的数据如果不在缓存里，业务系统需要重新生成缓存，因此就会访问数据库，并将数据更新到 Redis 里，这样后续请求都可以直接命中缓存。



那么，当大量缓存数据在同一时间过期（失效）^{*}时，如果此时有大量的用户请求，都无法在 Redis 中处理，于是全部请求都直接访问数据库，从而导致数据库的压力骤增，严重的会造成数据库宕机，从而形成一系列连锁反应，造成整个系统崩溃，这就是^{*}缓存雪崩的问题。

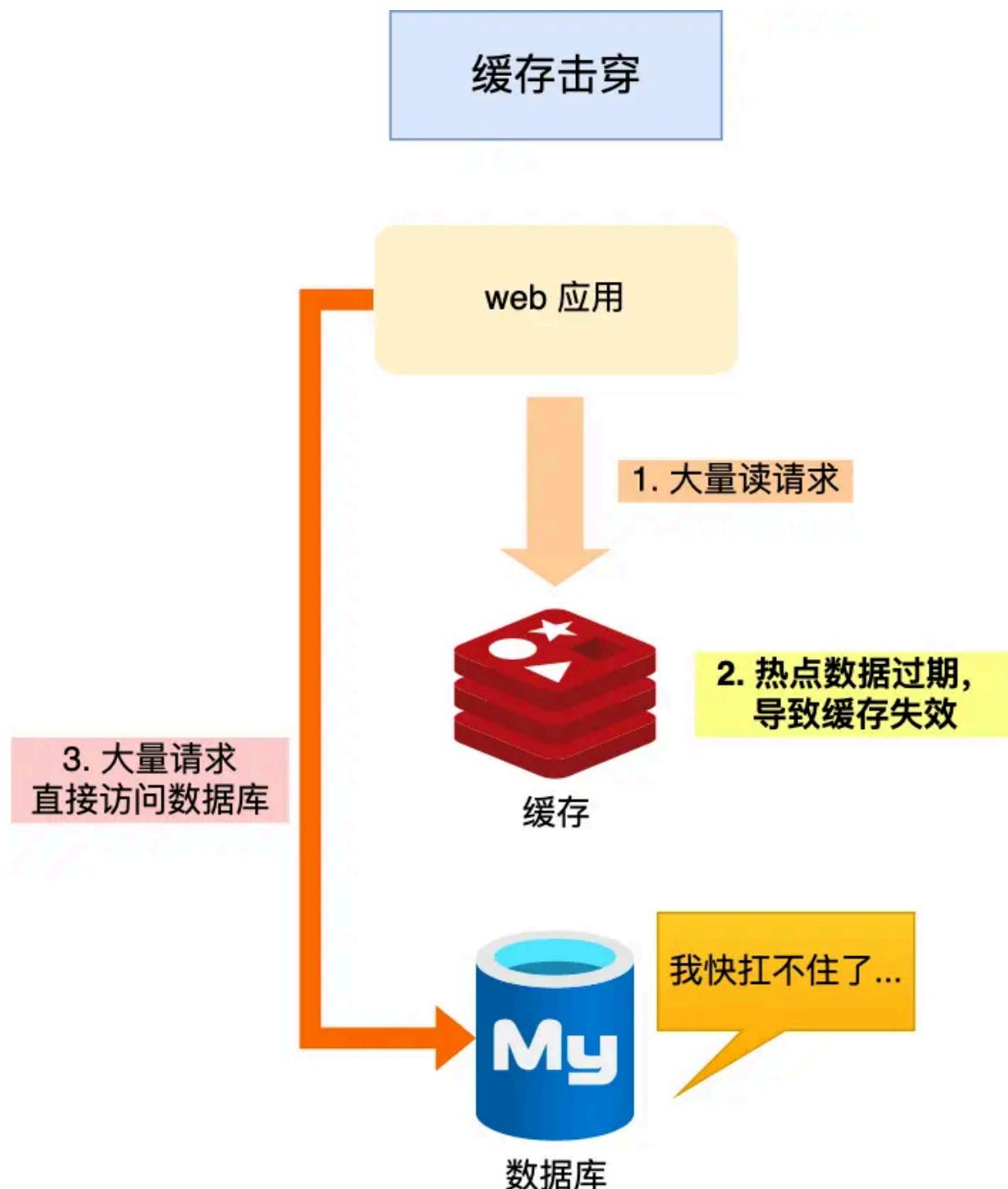
对于缓存雪崩问题，我们可以采用两种方案解决。

- **将缓存失效时间随机打散：** 我们可以在原有的失效时间基础上增加一个随机值（比如 1 到 10 分钟）这样每个缓存的过期时间都不重复了，也就降低了缓存集体失效的概率。
- **设置缓存不过期：** 我们可以通过后台服务来更新缓存数据，从而避免因为缓存失效造成的缓存雪崩，也可以在一定程度上避免缓存并发问题。

如何避免缓存击穿？

我们的业务通常会有几个数据会被频繁地访问，比如秒杀活动，这类被频地访问的数据被称为热点数据。

如果缓存中的某个热点数据过期了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，数据库很容易就被高并发的请求冲垮，这就是**缓存击穿**的问题。



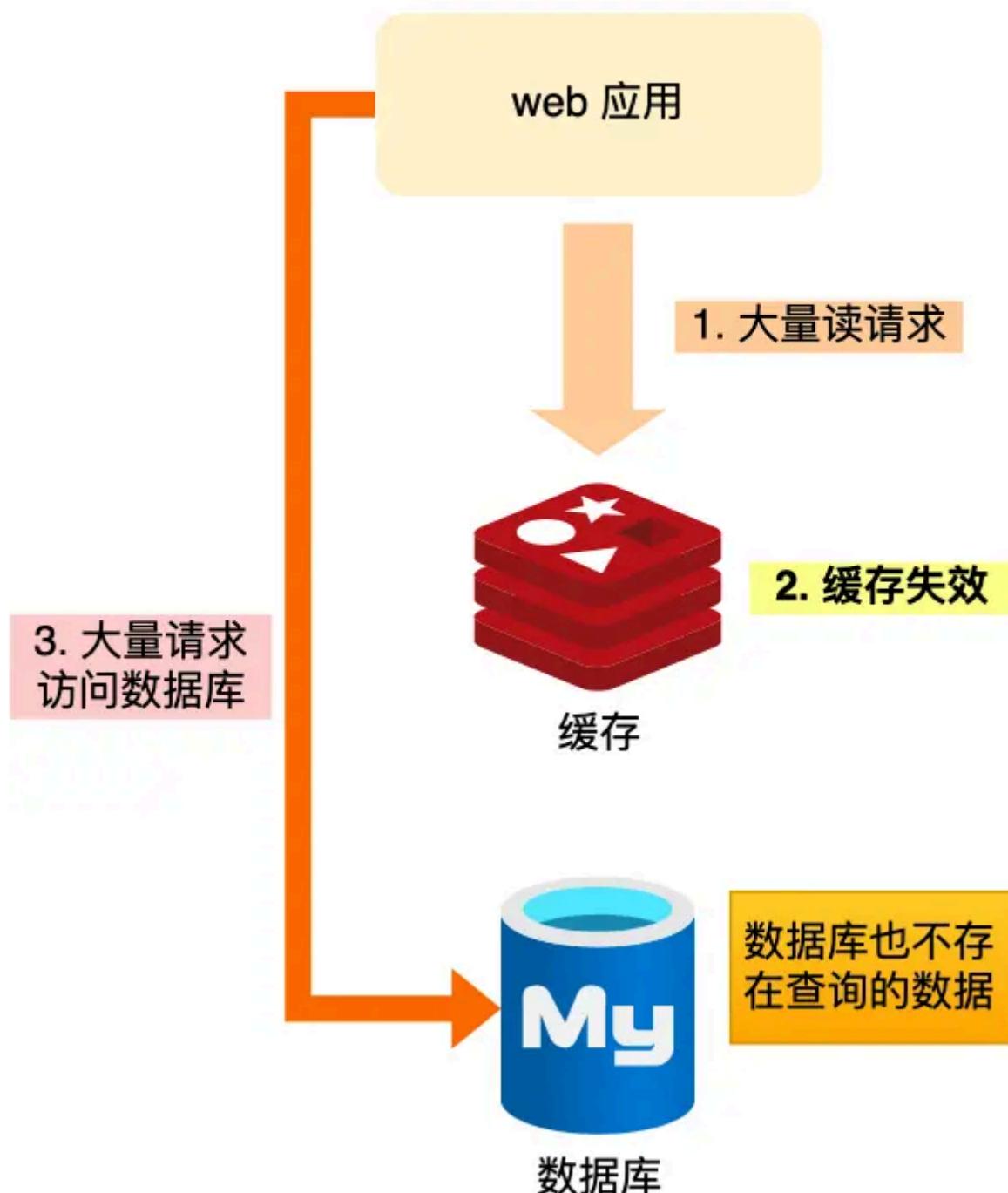
可以发现缓存击穿跟缓存雪崩很相似，你可以认为缓存击穿是缓存雪崩的一个子集。应对缓存击穿可以采取前面说到两种方案：

- **互斥锁方案**（Redis 中使用 setNX 方法设置一个状态位，表示这是一种锁定状态），保证同一时间只有一个业务线程请求缓存，未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。
- **不给热点数据设置过期时间，由后台异步更新缓存，或者在热点数据准备要过期前，提前通知后台线程更新缓存以及重新设置过期时间；**

如何避免缓存穿透？

当发生缓存雪崩或击穿时，数据库中还是保存了应用要访问的数据，一旦缓存恢复相对应的数据，就可以减轻数据库的压力，而缓存穿透就不一样了。当用户访问的数据，**既不在缓存中，也不在数据库中**，导致请求在访问缓存时，发现缓存缺失，再去访问数据库时，发现数据库中也没有要访问的数据，没办法构建缓存数据，来服务后续的请求。那么当有大量这样的请求到来时，数据库的压力骤增，这就是**缓存穿透**的问题。

缓存穿透



缓存穿透的发生一般有这两种情况：

- 业务误操作，缓存中的数据和数据库中的数据都被误删除了，所以导致缓存和数据库中都没有数据；
- 黑客恶意攻击，故意大量访问某些读取不存在数据的业务；

应对缓存穿透的方案，常见的方案有三种。

- 非法请求的限制：**当有大量恶意请求访问不存在的数据的时候，也会发生缓存穿透，因此在 API 入口处我们要判断请求参数是否合理，请求参数是否含有非法值、请求字段是否存在，如果判断出是恶意请求就直接返回错误，避免进一步访问缓存和数据库。
- 设置空值或者默认值：**当我们线上业务发现缓存穿透的现象时，可以针对查询的数据，在缓存中设置一个空值或者默认值，这样后续请求就可以从缓存中读取到空值或者默认值，返回给应用，而不会继续查询数据库。
- 使用布隆过滤器快速判断数据是否存在，避免通过查询数据库来判断数据是否存在：**我们可以在写入数据库数据时，使用布隆过滤器做个标记，然后在用户请求到来时，业务线程确认缓存失效后，可以通过查询布隆过滤器快速判断数据是否存在，如果不存在，就不用通过查询数据库来判断数据是否存在，即使发生了缓存穿透，大量请求只会查询 Redis 和布隆过滤器，而不会查询数据库，保证了数据库能正常运行，Redis 自身也是支持布隆过滤器的。

缓存异常	产生原因	应对方案
缓存雪崩	大量数据同时过期	<ul style="list-style-type: none"> - 均匀设置过期时间，避免同一时间过期 - 互斥锁，保证同一时间只有一个应用在构建缓存 - 双 key 策略，主 key 设置过期时间，备 key 永久，主 key 过期时，返回备 key 的内容 - 后台更新缓存，定时更新、消息队列通知更新
	Redis 故障宕机	<ul style="list-style-type: none"> - 服务熔断 - 请求限流 - 构建 Redis 缓存高可靠集群
缓存击穿	频繁访问的热点数据过期	<ul style="list-style-type: none"> - 互斥锁 - 不给热点数据设置过期时间，由后台更新缓存
缓存穿透	访问的数据既不在缓存，也不在数据库	<ul style="list-style-type: none"> - 非法请求的限制； - 缓存空值或者默认值； - 使用布隆过滤器快速判断数据是否存在；

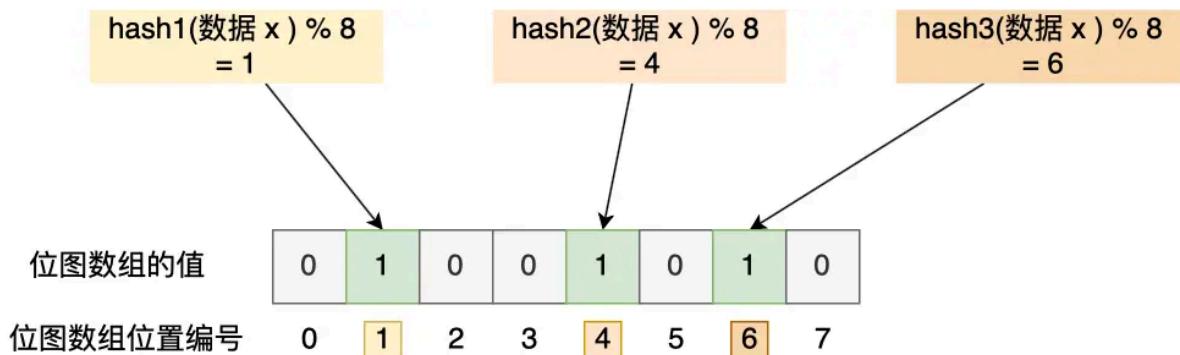
布隆过滤器

布隆过滤器由「初始值都为 0 的位图数组」和「N 个哈希函数」两部分组成。当我们在写入数据库数据时，在布隆过滤器里做个标记，这样下次查询数据是否在数据库时，只需要查询布隆过滤器，如果查询到数据没有被标记，说明不在数据库中。

布隆过滤器会通过 3 个操作完成标记：

- 第一步，使用 N 个哈希函数分别对数据做哈希计算，得到 N 个哈希值；**
- 第二步，将第一步得到的 N 个哈希值对位图数组的长度取模，得到每个哈希值在位图数组的对应位置。**
- 第三步，将每个哈希值在位图数组的对应位置的值设置为 1；**

举个例子，假设有一个位图数组长度为 8，哈希函数 3 个的布隆过滤器。



在数据库写入数据 x 后，把数据 x 标记在布隆过滤器时，数据 x 会被 3 个哈希函数分别计算出 3 个哈希值，然后对这 3 个哈希值对 8 取模，假设取模的结果为 1、4、6，然后把位图数组的第 1、4、6 位置的值设置为 1。当应用要查询数据 x 是否数据库时，通过布隆过滤器只要查到位图数组的第 1、4、6 位置的值是否全为 1，只要有一个为 0，就认为数据 x 不在数据库中。

布隆过滤器由于是基于哈希函数实现查找的，高效查找的同时存在哈希冲突的可能性，比如数据 x 和数据 y 可能都落在第 1、4、6 位置，而事实上，可能数据库中并不存在数据 y ，存在误判的情况。

所以，查询布隆过滤器说数据存在，并不一定证明数据库中存在这个数据，但是查询到数据不存在，数据库中一定就不存在这个数据。

如何设计一个缓存策略，可以动态缓存热点数据呢？

由于数据存储受限，系统并不是将所有数据都需要存放到缓存中的，而只是将其中一部分热点数据缓存起来，所以我们要设计一个热点数据动态缓存的策略。

热点数据动态缓存的策略总体思路：通过数据最新访问时间来做排名，并过滤掉不常访问的数据，只留下经常访问的数据。

以电商平台场景中的例子，现在要求只缓存用户经常访问的 Top 1000 的商品。具体细节如下：

- 先通过缓存系统做一个排序队列（比如存放 1000 个商品），系统会根据商品的访问时间，更新队列信息，越是最近访问的商品排名越靠前；
- 同时系统会定期过滤掉队列中排名最后的 200 个商品，然后再从数据库中随机读取出 200 个商品加入队列中；
- 这样当请求每次到达的时候，会先从队列中获取商品 ID，如果命中，就根据 ID 再从另一个缓存数据结构中读取实际的商品信息，并返回。

在 Redis 中可以用 `zadd` 方法和 `zrange` 方法来完成排序队列和获取 200 个商品的操作。

说说常见的缓存更新策略？

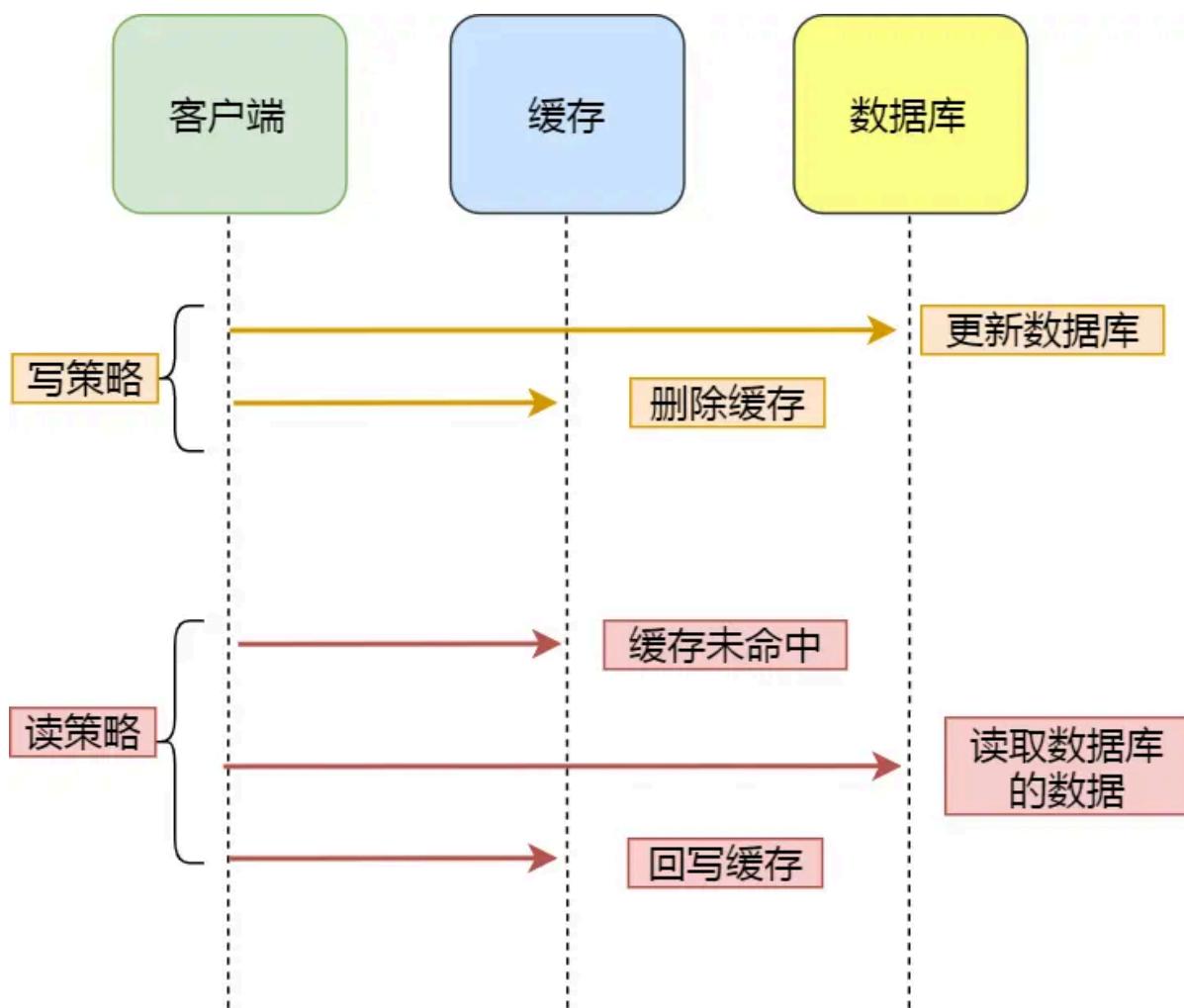
常见的缓存更新策略共有3种：

- Cache Aside (旁路缓存) 策略；**
- Read/Write Through (读穿 / 写穿) 策略；**
- Write Back (写回) 策略；**

实际开发中，Redis 和 MySQL 的更新策略用的是 Cache Aside，另外两种策略应用不了。

Cache Aside (旁路缓存) 策略

Cache Aside (旁路缓存) 策略是最常用的，应用程序直接与「数据库、缓存」交互，并负责对缓存的维护，该策略又可以细分为「读策略」和「写策略」。



写策略的步骤:

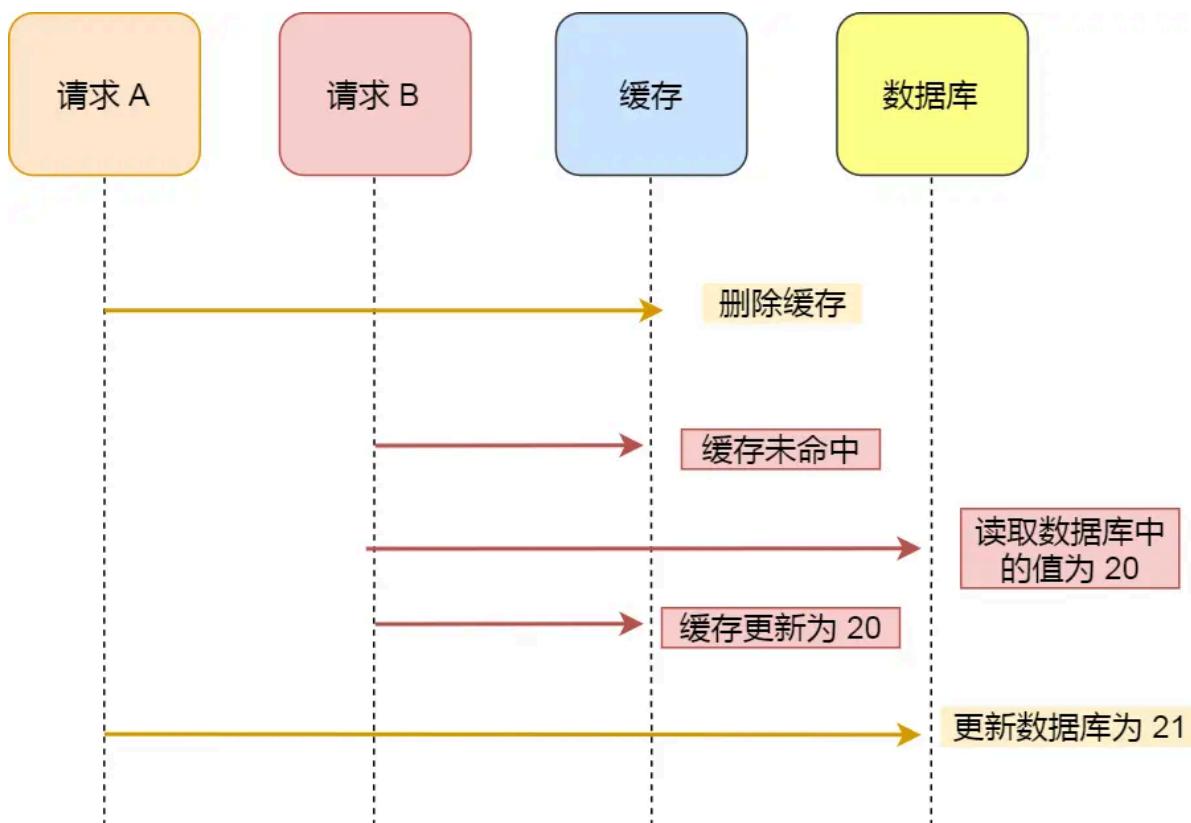
- 先更新数据库中的数据，再删除缓存中的数据。

读策略的步骤:

- 如果读取的数据命中了缓存，则直接返回数据；
- 如果读取的数据没有命中缓存，则从数据库中读取数据，然后将数据写入到缓存，并且返回给用户。

注意，写策略的步骤的顺序不能倒过来，即不能先删除缓存再更新数据库，原因是在「读+写」并发的时候，会出现缓存和数据库的数据不一致性的问题。

举个例子，假设某个用户的年龄是 20，请求 A 要更新用户年龄为 21，所以它会删除缓存中的内容。这时，另一个请求 B 要读取这个用户的年龄，它查询缓存发现未命中后，会从数据库中读取到年龄为 20，并且写入到缓存中，然后请求 A 继续更改数据库，将用户的年龄更新为 21。

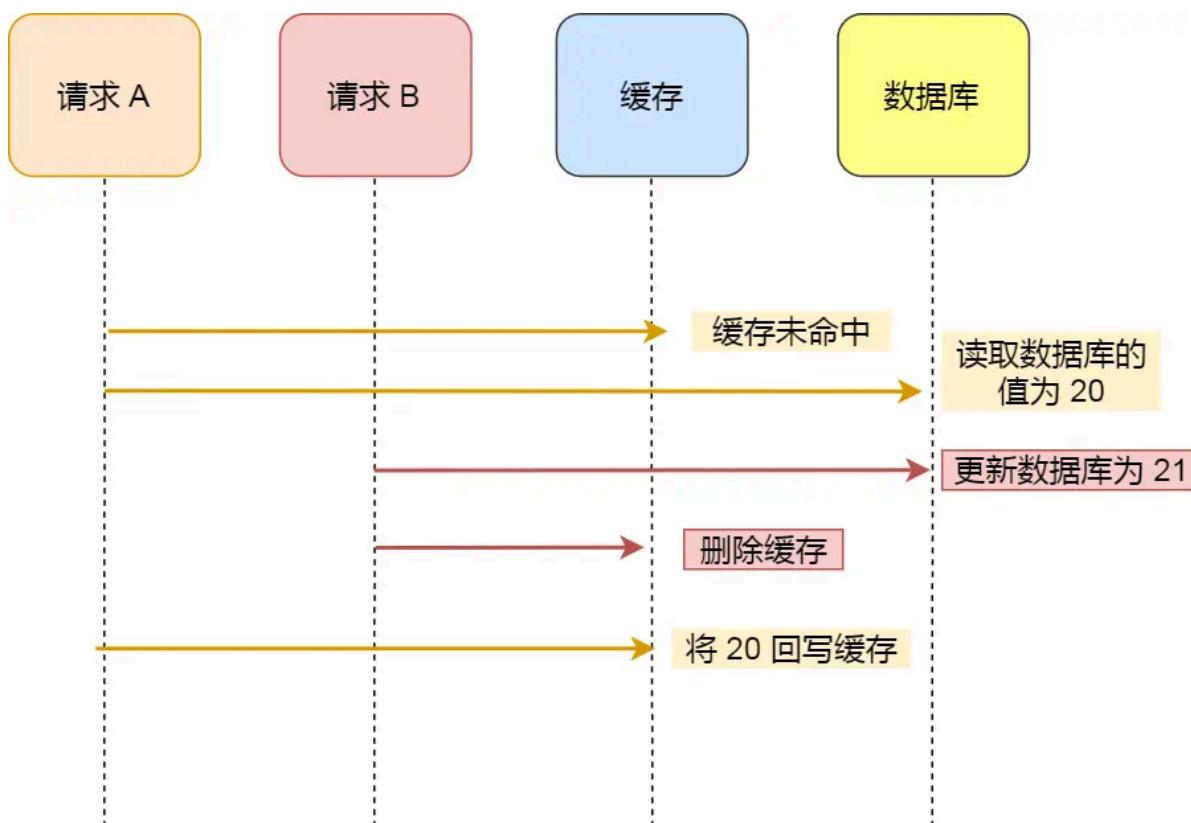


最终，该用户年龄在缓存中是 20（旧值），在数据库中是 21（新值），缓存和数据库的数据不一致。

为什么「先更新数据库再删除缓存」不会有数据不一致的问题？

继续用「读 + 写」请求的并发的场景来分析。

假如某个用户数据在缓存中不存在，请求 A 读取数据时从数据库中查询到年龄为 20，在未写入缓存中时另一个请求 B 更新数据。它更新数据库中的年龄为 21，并且清空缓存。这时请求 A 把从数据库中读到的年龄为 20 的数据写入到缓存中。



最终，该用户年龄在缓存中是 20（旧值），在数据库中是 21（新值），缓存和数据库数据不一致。从上面的理论上分析，先更新数据库，再删除缓存也是会出现数据一致性的问题，**但是在实际中，这个问题出现的概率并不高。**

因为缓存的写入通常要远远快于数据库的写入，所以在实际中很难出现请求 B 已经更新了数据库并且删除了缓存，请求 A 才更新完缓存的情况。而一旦请求 A 早于请求 B 删除缓存之前更新了缓存，那么接下来的请求就会因为缓存不命中而从数据库中重新读取数据，所以不会出现这种不一致的情况。

Cache Aside 策略适合读多写少的场景，不适合写多的场景，因为当写入比较频繁时，缓存中的数据会被频繁地清理，这样会对缓存的命中率有一些影响。如果业务对缓存命中率有严格的要求，那么可以考虑两种解决方案：

- 一种做法是在更新数据时也更新缓存，只是在更新缓存前先加一个分布式锁，因为这样在同一时间只允许一个线程更新缓存，就不会产生并发问题了。当然这么做对于写入的性能会有一些影响；
- 另一种做法同样也是在更新数据时更新缓存，只是给缓存加一个较短的过期时间，这样即使出现缓存不一致的情况，缓存的数据也会很快过期，对业务的影响也是可以接受。

Read/Write Through（读穿 / 写穿）策略

Read/Write Through（读穿 / 写穿）策略原则是应用程序只和缓存交互，不再和数据库交互，而是由缓存和数据库交互，相当于更新数据库的操作由缓存自己代理了。

1. Read Through 策略

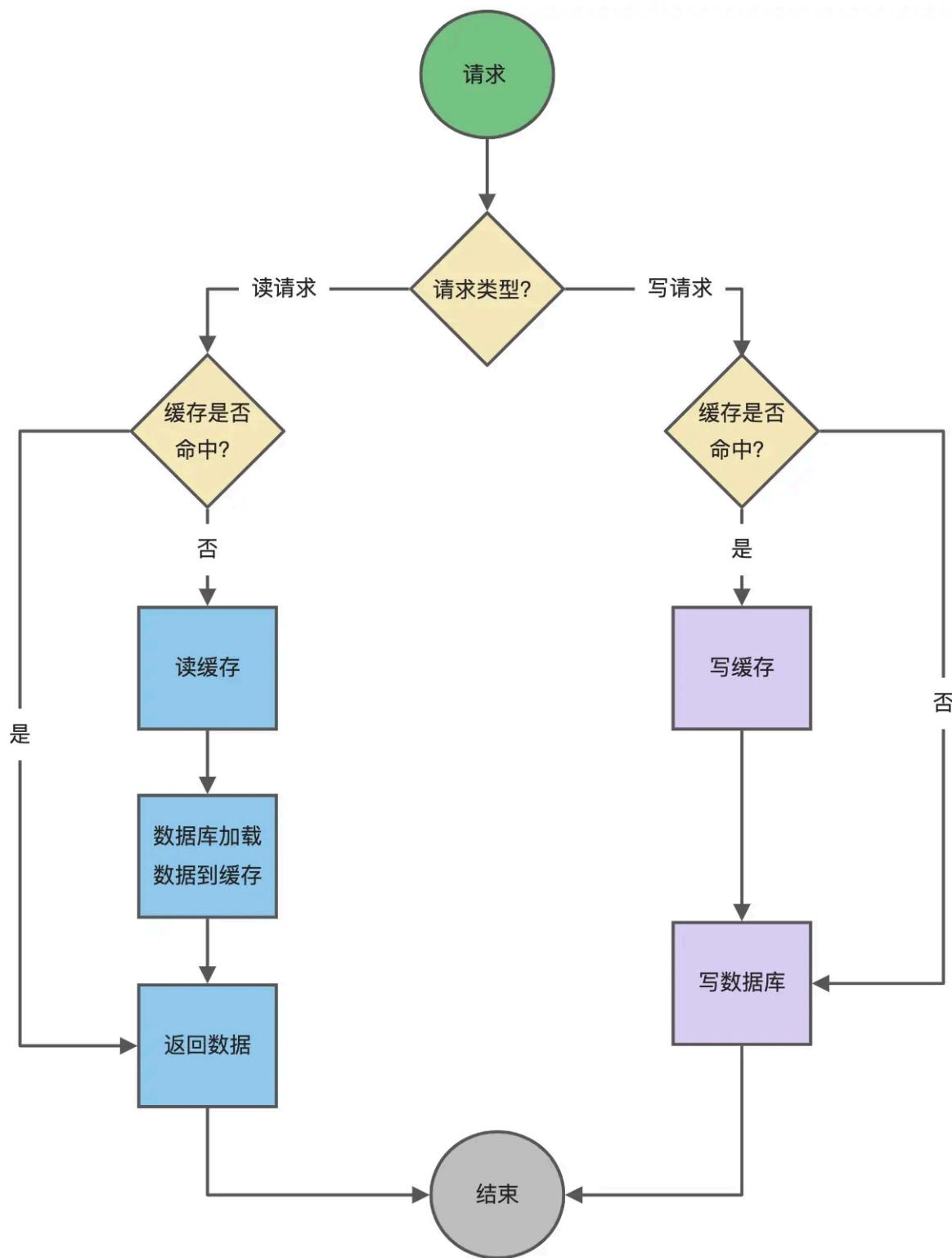
先查询缓存中数据是否存在，如果存在则直接返回，如果不存在，则由缓存组件负责从数据库查询数据，并将结果写入到缓存组件，最后缓存组件将数据返回给应用。

2. Write Through 策略

当有数据更新的时候，先查询要写入的数据在缓存中是否已经存在：

- 如果缓存中数据已经存在，则更新缓存中的数据，并且由缓存组件同步更新到数据库中，然后缓存组件告知应用程序更新完成。
- 如果缓存中数据不存在，直接更新数据库，然后返回；

下面是 Read Through/Write Through 策略的示意图：



Read Through/Write Through 策略的特点是由缓存节点而非应用程序来和数据库打交道，在我们开发过程中相比 Cache Aside 策略要少见一些，原因是我们在经常使用的分布式缓存组件，无论是 Memcached 还是 Redis 都不提供写入数据库和自动加载数据库中的数据的功能。而我们在使用本地缓存的时候可以考虑使用这种策略。

Write Back (写回) 策略

Write Back (写回) 策略在更新数据的时候，只更新缓存，同时将缓存数据设置为脏的，然后立马返回，并不会更新数据库。对于数据库的更新，会通过批量异步更新的方式进行。

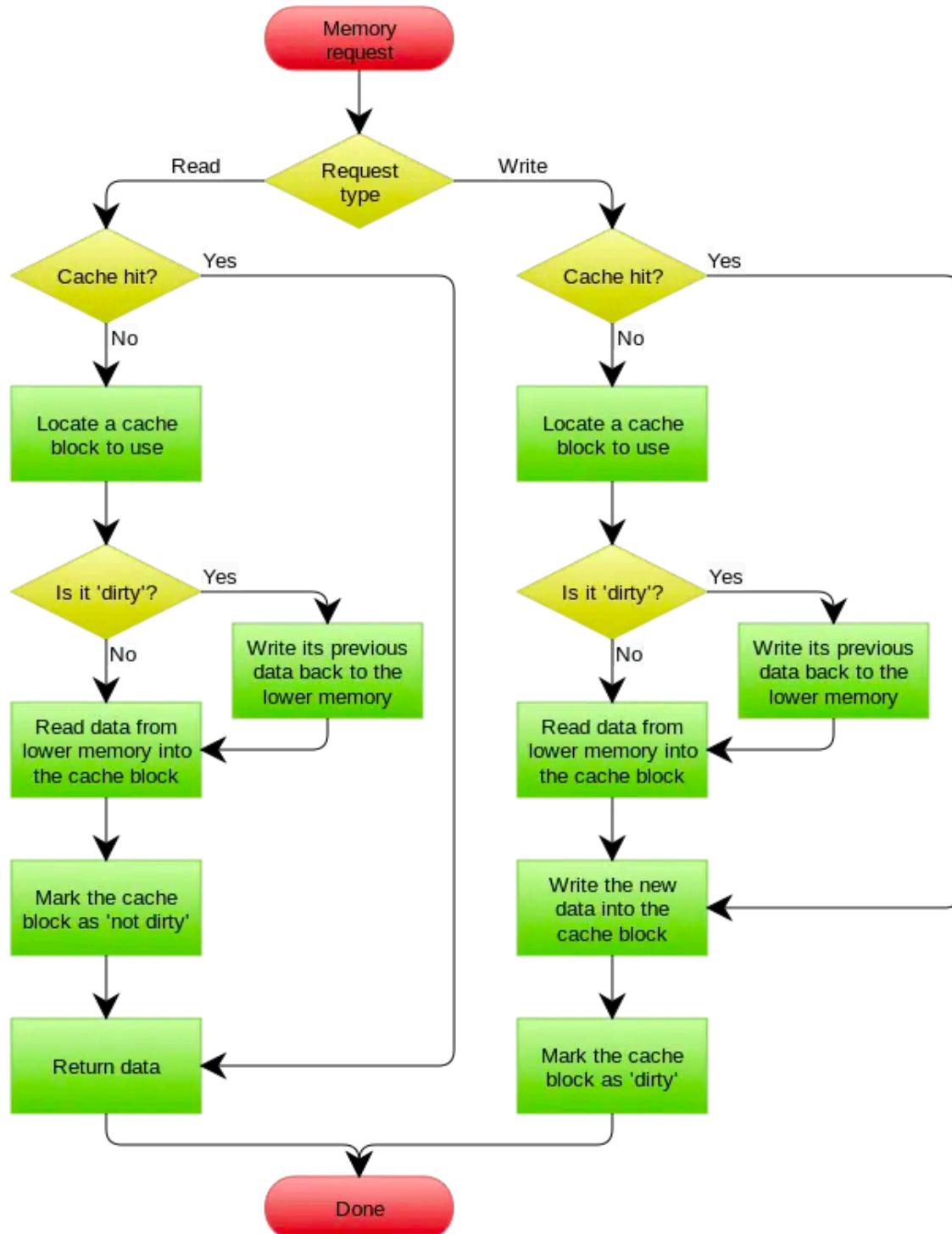
实际上，Write Back (写回) 策略也不能应用到我们常用的数据库和缓存的场景中，因为 Redis 并没有异步更新数据库的功能。

Write Back 是计算机体系结构中的设计，比如 CPU 的缓存、操作系统中文件系统的缓存都采用了 Write Back（写回）策略。

Write Back 策略特别适合写多的场景，因为发生写操作的时候，只需要更新缓存，就立马返回了。比如，写文件的时候，实际上是写入到文件系统的缓存就返回了，并不会写磁盘。

但是带来的问题是，数据不是强一致性的，而且会有数据丢失的风险，因为缓存一般使用内存，而内存是非持久化的，所以一旦缓存机器掉电，就会造成原本缓存中的脏数据丢失。所以你会发现系统在掉电之后，之前写入的文件会有部分丢失，就是因为 Page Cache 还没有来得及刷盘造成的。

这里贴一张 CPU 缓存与内存使用 Write Back 策略的流程图：

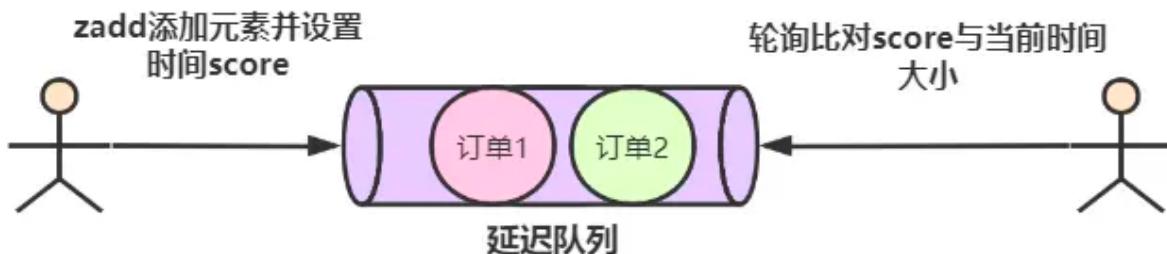


Redis 如何实现延迟队列？

延迟队列是指把当前要做的事情，往后推迟一段时间再做。延迟队列的常见使用场景有以下几种：

- 在淘宝、京东等购物平台上下单，超过一定时间未付款，订单会自动取消；
- 打车的时候，在规定时间没有车主接单，平台会取消你的单并提醒你暂时没有车主接单；
- 点外卖的时候，如果商家在10分钟还没接单，就会自动取消订单；

在 Redis 可以使用有序集合（ZSet）的方式来实现延迟消息队列的，ZSet 有一个 Score 属性可以用来存储延迟执行的时间。使用 zadd score1 value1 命令就可以一直往内存中生产消息。再利用 zrangebyscore 查询符合条件的所有待处理的任务，通过循环执行队列任务即可。



Redis 的大 key 如何处理？

什么是 Redis 大 key？

大 key 并不是指 key 的值很大，而是 key 对应的 value 很大。一般而言，下面这两种情况被称为大 key：

- String 类型的值大于 10 KB；
- Hash、List、Set、ZSet 类型的元素的个数超过 5000 个；

大 key 会造成什么问题？

大 key 会带来以下四种影响：

- **客户端超时阻塞。**由于 Redis 执行命令是单线程处理，然后在操作大 key 时会比较耗时，那么就会阻塞 Redis，从客户端这一视角看，就是很久很久都没有响应。
- **引发网络阻塞。**每次获取大 key 产生的网络流量较大，如果一个 key 的大小是 1 MB，每秒访问量为 1000，那么每秒会产生 1000MB 的流量，这对于普通千兆网卡的服务器来说是灾难性的。
- **阻塞工作线程。**如果使用 del 删除大 key 时，会阻塞工作线程，这样就没办法处理后续的命令。
- **内存分布不均。**集群模型在 slot 分片均匀情况下，会出现数据和查询倾斜情况，部分有大 key 的 Redis 节点占用内存多，QPS 也会比较大。

如何找到大 key？

1. redis-cli --bigkeys 查找大key

可以通过 redis-cli --bigkeys 命令查找大 key：

```
redis-cli -h 127.0.0.1 -p6379 -a "password" -- bigkeys
```

使用的时候注意事项：

- **最好选择在从节点上执行该命令。因为主节点上执行时，会阻塞主节点；**
- **如果没有从节点，那么可以选择在 Redis 实例业务压力的低峰阶段进行扫描查询，以免影响到实例的正常运行；或者可以使用 -i 参数控制扫描间隔，避免长时间扫描降低 Redis 实例的性能。**

该方式的不足之处：

- 这个方法只能返回每种类型中最大的那个 bigkey，无法得到大小排在前 N 位的 bigkey；
- 对于集合类型来说，这个方法只统计集合元素个数的多少，而不是实际占用的内存量。但是，一个集合中的元素个数多，并不一定占用的内存就多。因为，有可能每个元素占用的内存很小，这样的话，即使元素个数有很多，总内存开销也不大；

2. 使用 `SCAN` 命令查找大 key

使用 `SCAN` 命令对数据库扫描，然后用 `TYPE` 命令获取返回的每一个 key 的类型。

对于 String 类型，可以直接使用 `STRLEN` 命令获取字符串的长度，也就是占用的内存空间字节数。

对于集合类型来说，有两种方法可以获得它占用的内存大小：

- 如果能够预先从业务层知道集合元素的平均大小，那么，可以使用下面的命令获取集合元素的个数，然后乘以集合元素的平均大小，这样就能获得集合占用的内存大小了。List 类型：`LLEN` 命令；Hash 类型：`HLEN` 命令；Set 类型：`SCARD` 命令；Sorted Set 类型：`ZCARD` 命令；
- 如果不能提前知道写入集合的元素大小，可以使用 `MEMORY USAGE` 命令（需要 Redis 4.0 及以上版本），查询一个键值对占用的内存空间。

3. 使用 `RdbTools` 工具查找大 key

使用 `RdbTools` 第三方开源工具，可以用来解析 Redis 快照（RDB）文件，找到其中的大 key。

比如，下面这条命令，将大于 10 kb 的 key 输出到一个表格文件。

```
rdb dump.rdb -c memory --bytes 10240 -f redis.csv
```

如何删除大 key？

删除操作的本质是要释放键值对占用的内存空间，不要小瞧内存的释放过程。

释放内存只是第一步，为了更加高效地管理内存空间，在应用程序释放内存时，操作系统需要把释放掉的内存块插入一个空闲内存块的链表，以便后续进行管理和再分配。这个过程本身需要一定时间，而且会阻塞当前释放内存的应用程序。

所以，如果一下子释放了大量内存，空闲内存块链表操作时间就会增加，相应地就会造成 Redis 主线程的阻塞，如果主线程发生了阻塞，其他所有请求可能都会超时，超时越来越多，会造成 Redis 连接耗尽，产生各种异常。

因此，删除大 key 这一个动作，我们要小心。具体要怎么做呢？这里给出两种方法：

- 分批次删除
- 异步删除（Redis 4.0 版本以上）

1. 分批次删除

对于删除大 Hash，使用 `hscan` 命令，每次获取 100 个字段，再用 `hdel` 命令，每次删除 1 个字段。

Python 代码：

```

def del_large_hash():
    r = redis.StrictRedis(host='redis-host1', port=6379)
    large_hash_key = "xxx" #要删除的大hash键名
    cursor = '0'
    while cursor != 0:
        # 使用 hscan 命令，每次获取 100 个字段
        cursor, data = r.hscan(large_hash_key, cursor=cursor, count=100)
        for item in data.items():
            # 再用 hdel 命令，每次删除1个字段
            r.hdel(large_hash_key, item[0])

```

对于删除大 List，通过 `ltrim` 命令，每次删除少量元素。

Python代码：

```

def del_large_list():
    r = redis.StrictRedis(host='redis-host1', port=6379)
    large_list_key = 'xxx' #要删除的大list的键名
    while r.llen(large_list_key)>0:
        #每次只删除最右100个元素
        r.ltrim(large_list_key, 0, -101)

```

对于删除大 Set，使用 `sscan` 命令，每次扫描集合中 100 个元素，再用 `srem` 命令每次删除一个键。

Python代码：

```

def del_large_set():
    r = redis.StrictRedis(host='redis-host1', port=6379)
    large_set_key = 'xxx' # 要删除的大set的键名
    cursor = '0'
    while cursor != 0:
        # 使用 sscan 命令，每次扫描集合中 100 个元素
        cursor, data = r.sscan(large_set_key, cursor=cursor, count=100)
        for item in data:
            # 再用 srem 命令每次删除一个键
            r.srem(large_size_key, item)

```

对于删除大 ZSet，使用 `zremrangebyrank` 命令，每次删除 top 100个元素。

Python代码：

```

def del_large_sortedset():
    r = redis.StrictRedis(host='large_sortedset_key', port=6379)
    large_sortedset_key='xxx'
    while r.zcard(large_sortedset_key)>0:
        # 使用 zremrangebyrank 命令，每次删除 top 100个元素
        r.zremrangebyrank(large_sortedset_key,0,99)

```

2、异步删除

从 Redis 4.0 版本开始，可以采用异步删除法，用 `unlink` 命令代替 `del` 来删除。

这样 Redis 会将这个 key 放入到一个异步线程中进行删除，这样不会阻塞主线程。

除了主动调用 unlink 命令实现异步删除之外，我们还可以通过配置参数，达到某些条件的时候自动进行异步删除。

主要有 4 种场景， 默认都是关闭的：

```
lazyfree-lazy-eviction no  
lazyfree-lazy-expire no  
lazyfree-lazy-server-del  
noslave-lazy-flush no
```

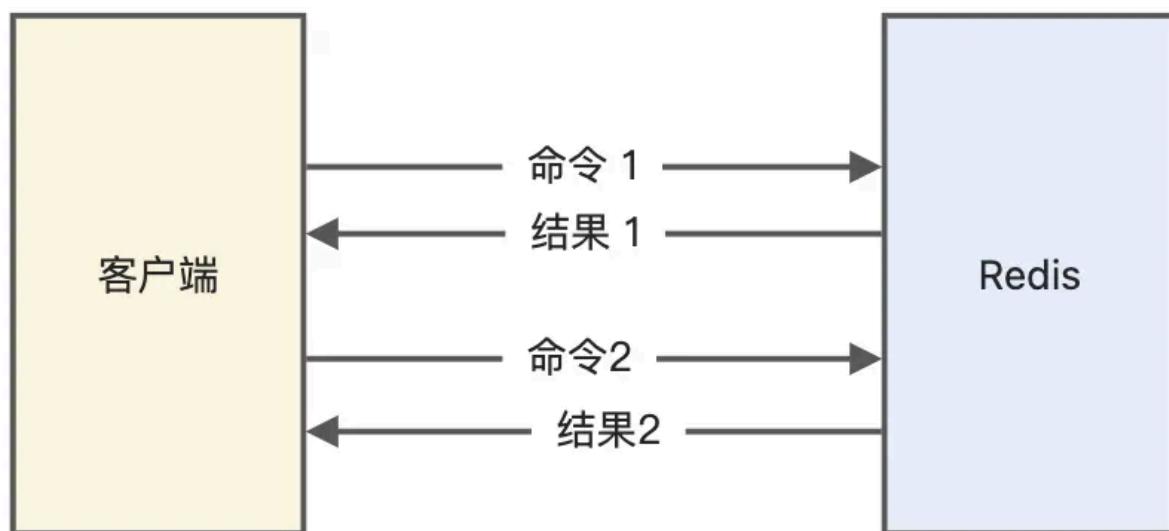
它们代表的含义如下：

- lazyfree-lazy-eviction：表示当 Redis 运行内存超过 maxmemory 时，是否开启 lazy free 机制删除；
- lazyfree-lazy-expire：表示设置了过期时间的键值，当过期之后是否开启 lazy free 机制删除；
- lazyfree-lazy-server-del：有些指令在处理已存在的键时，会带有一个隐式的 del 键的操作，比如 rename 命令，当目标键已存在，Redis 会先删除目标键，如果这些目标键是一个 big key，就会造成阻塞删除的问题，此配置表示在这种场景中是否开启 lazy free 机制删除；
- slave-lazy-flush：针对 slave (从节点) 进行全量数据同步，slave 在加载 master 的 RDB 文件前，会运行 flushall 来清理自己的数据，它表示此时是否开启 lazy free 机制删除。

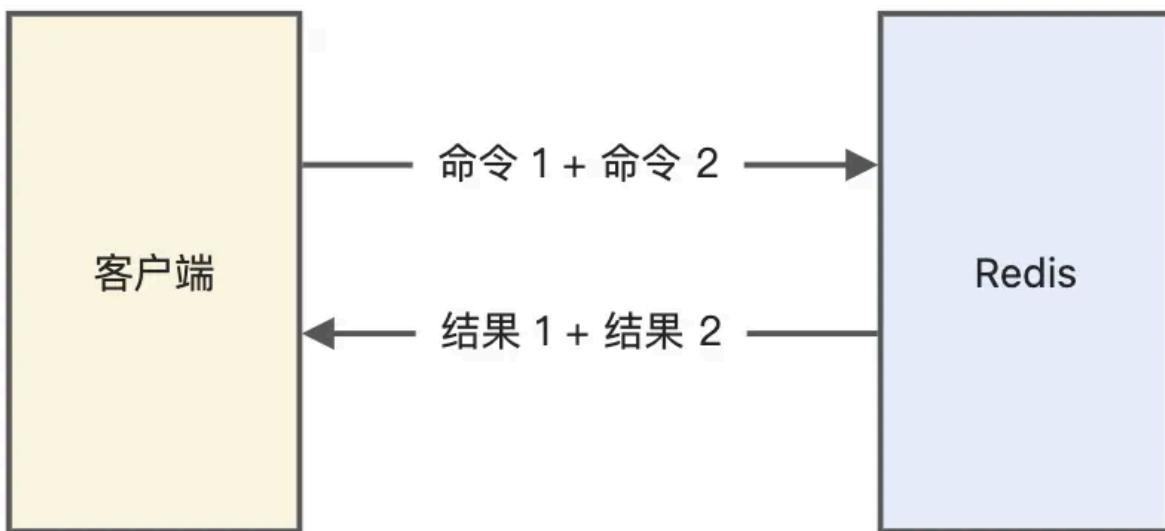
建议开启其中的 lazyfree-lazy-eviction、lazyfree-lazy-expire、lazyfree-lazy-server-del 等配置，这样就可以有效的提高主线程的执行效率。

Redis 管道有什么用？

管道技术 (Pipeline) 是客户端提供的一种批处理技术，用于一次处理多个 Redis 命令，从而提高整个交互的性能。普通命令模式，如下图所示：



管道模式，如下图所示：



使用管道技术可以解决多个命令执行时的网络等待，它是把多个命令整合到一起发送给服务器端处理之后统一返回给客户端，这样就免去了每条命令执行后都要等待的情况，从而有效地提高了程序的执行效率。但使用管道技术也要注意避免发送的命令过大，或管道内的数据太多而导致的网络阻塞。要注意的是，**管道技术本质上是客户端提供的功能，而非 Redis 服务器端的功能**。

Redis 事务支持回滚吗？

MySQL 在执行事务时，会提供回滚机制，当事务执行发生错误时，事务中的所有操作都会撤销，已经修改的数据也会被恢复到事务执行前的状态。**Redis 中并没有提供回滚机制**，虽然 Redis 提供了 DISCARD 命令，但是这个命令只能用来主动放弃事务执行，把暂存的命令队列清空，起不到回滚的效果。

下面是 DISCARD 命令用法：

```
#读取 count 的值
127.0.0.1:6379> GET count
"1"
#开启事务
127.0.0.1:6379> MULTI
OK
#发送事务的第一个操作，对count减1
127.0.0.1:6379> DECR count
QUEUED
#执行DISCARD命令，主动放弃事务
127.0.0.1:6379> DISCARD
OK
#再次读取a:stock的值，值没有被修改
127.0.0.1:6379> GET count
"1"
```

事务执行过程中，**如果命令入队时没报错，而事务提交后，实际执行时报错了，正确的命令依然可以正常执行**，所以这可以看出 Redis 并不一定保证原子性（原子性：事务中的命令要不全部成功，要不全部失败）。

比如下面这个例子：

```
#获取name原本的值
127.0.0.1:6379> GET name
"xiaolin"
#开启事务
127.0.0.1:6379> MULTI
```

```
OK
#设置新值
127.0.0.1:6379(TX)> SET name xialincoding
QUEUED
#注意，这条命令是错误的
# expire 过期时间正确来说是数字，并不是‘10s’字符串，但是还是入队成功了
127.0.0.1:6379(TX)> EXPIRE name 10s
QUEUED
#提交事务，执行报错
#可以看到 set 执行成功，而 expire 执行错误。
127.0.0.1:6379(TX)> EXEC
1) OK
2) (error) ERR value is not an integer or out of range
#可以看到，name 还是被设置为新值了
127.0.0.1:6379> GET name
"xialincoding"
```

为什么Redis不支持事务回滚？

Redis [官方文档](#)(*opens new window*)的解释如下：

大概的意思是，作者不支持事务回滚的原因有以下两个：

- 他认为 Redis 事务的执行时，错误通常都是编程错误造成的，这种错误通常只会出现在开发环境中，而很少会在实际的生产环境中出现，所以他认为没有必要为 Redis 开发事务回滚功能；
- 不支持事务回滚是因为这种复杂的功能和 Redis 追求的简单高效的设计主旨不符合。

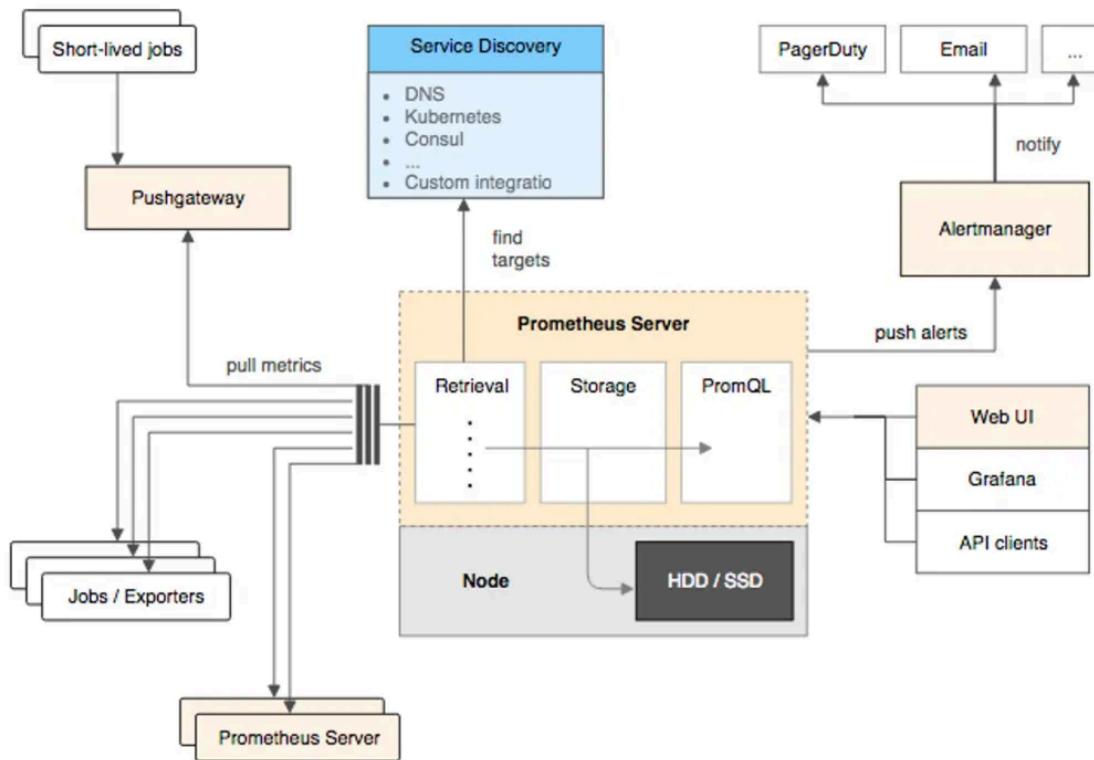
这里不支持事务回滚，指的是不支持事务运行时错误的事务回滚。

Prometheus

Prometheus 是一个数据监控的解决方案，为我们的系统提供指标收集和监控，让我们能随时掌握系统运行的状态，快速定位问题和排除故障。Prometheus 发展速度很快，使用go进行开发，12 年开发完成，16 年加入 CNCF，成为继 K8s 之后第二个 CNCF 托管的项目。

整体生态

Prometheus 提供了从指标暴露，到指标抓取、存储和可视化，以及最后的监控告警等一系列组件。



指标暴露

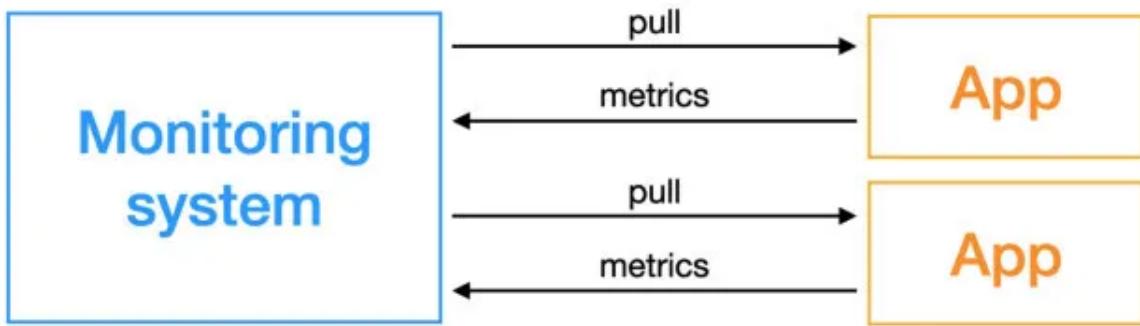
在Prometheus中每个被监控的服务都被称为**job**，Prometheus 为这些 Job 提供了官方的 **SDK**，利用这个 SDK 可以自定义并导出自己的业务指标，也可以使用 Prometheus 官方提供的各种常用组件和中间件的 **Exporter**（比如常用的 MySQL, Consul 等等）。对于短时间执行的脚本任务或者不好直接 Pull 指标的任务，Prometheus 提供了 **PushGateWay** 网关给这些任务将服务指标主动推 Push 到网关，Prometheus 再从这个网关里 Pull 指标。

指标抓取

Prometheus中指标抓取有**Push**和**Pull**两种抓取模型。

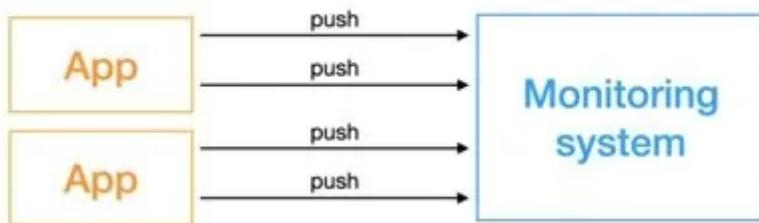
1. **Pull**: 监控服务主动拉取被监控服务的指标。被监控服务一般通过主动暴露 metrics 端口或者通过 **Exporter** 的方式暴露指标，监控服务依赖服务发现模块发现被监控服务，从而去定期的抓取指标。默认是一分钟拉取一次指标。

Pull-based system



2. Push: 被监控服务主动将指标推送到监控服务，可能需要对指标做协议适配，必须得符合监控服务要求的指标格式。

Push-based monitoring system



Prometheus 对外都是用的 Pull 模型，一个是 Pull Exporter 的暴露的指标，一个是 Pull PushGateway 暴露的指标。

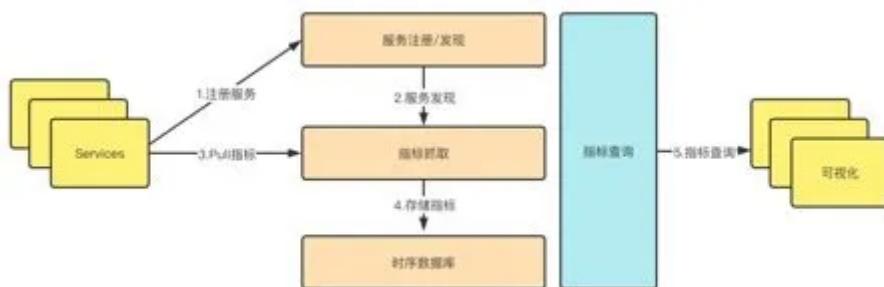
指标存储和查询

指标抓取后会存储在内置的时序数据库中，Prometheus 也提供了 PromQL 查询语言给我们做指标的查询，我们可以在 Prometheus 的 WebUI 上通过 PromQL，可视化查询我们的指标，也可以很方便的接入第三方的可视化工具，例如 grafana。

监控告警

Prometheus 提供了 alertmanager 基于 promql 来做系统的监控告警，当 promql 查询出来的指标超过我们定义的阈值时，Prometheus 会发送一条告警信息到 alertmanager，manager 会将告警下发到配置好的邮箱。

工作原理



服务注册

首先明确2个概念：

1. job：被监控服务
2. target：被监控服务的单个实例

对于服务注册而言，就是在prometheus中对job和target进行注册。注册分为静态注册和动态注册。

静态注册：静态的将服务的 IP 和抓取指标的端口号配置在 Prometheus yaml 文件的 scrape_configs 配置下：

```
scrape_configs:  
  - job_name: "prometheus"  
    static_configs:  
      - targets: ["localhost:9090"]
```

以上就是注册了一个名为 prometheus 的服务，这个服务下有一个实例，暴露的抓取地址是 localhost:9090。

动态注册：动态注册就是在 Prometheus yaml 文件的 scrape_configs 配置下配置服务发现的地址和服务名，Prometheus 会去该地址，根据你提供的服务名动态发现实例列表，在 Prometheus 中，支持 consul, DNS, 文件, K8s 等多种服务发现机制。

基于 consul 的服务发现：

```
- job_name: "node_export_consul"  
  metrics_path: /node_metrics  
  scheme: http  
  consul_sd_configs:  
    - server: localhost:8500  
      services:  
        - node_exporter
```

我们 consul 的地址就是：localhost:8500，服务名是 node_exporter，在这个服务下有一个 exporter 实例：localhost:9600。

我的理解简单来说，静态注册需要我们指定job和target的地址和端口；而动态注册只需要我们指定job的地址和端口，而具体的target实例的会通过发现机制去发现。

配置更新

修改完Prometheus的配置后，需要进行配置更新加载。一种是通过重启，一种是通过动态更新。

第一步：首先要保证启动 Prometheus 的时候带上启动参数：--web.enable-lifecycle

```
prometheus --config.file=/usr/local/etc/prometheus.yaml --web.enable-lifecycle
```

第二步：去更新我们的 Prometheus 配置：

第三步：更新完配置后，我们可以通过 Post 请求的方式，动态更新配置：

```
curl -v --request POST 'http://localhost:9090/-/reload'
```

原理：**Prometheus 在 web 模块中，注册了一个 handler：**

```

if o.EnableLifecycle {
    router.Post("/-/quit", h.quit)
    router.Put("/-/quit", h.quit)
    router.Post("/-/reload", h.reload) // reload配置
    router.Put("/-/reload", h.reload)
}

```

通过 `h.reload` 这个 handler 方法实现：这个 handler 就是往一个 channel 中发送一个信号：

```

func (h *Handler) reload(w http.ResponseWriter, r *http.Request) {
    rc := make(chan error)
    h.reloadCh <- rc // 发送一个信号到channel中
    if err := <-rc; err != nil {
        http.Error(w, fmt.Sprintf("failed to reload config: %s", err),
        http.StatusInternalServerError)
    }
}

```

在 main 函数中会去监听这个 channel，只要有监听到信号，就会做配置的 reload，重新将新配置加载到内存中：

```

case rc := <-webHandler.Reload():
    if err := reloadConfig(cfg.configFile, cfg.enableExpandExternalLabels,
    cfg.tsdb.EnableExemplarStorage, logger, noStepSubqueryInterval, reloaders...);
    err != nil {
        level.Error(logger).Log("msg", "Error reloading config", "err", err)
        rc <- err
    } else {
        rc <- nil
    }
}

```

指标抓取和存储

rometheus 对指标的抓取采取主动 Pull 的方式，即周期性的请求被监控服务暴露的 metrics 接口或者是 PushGateway，从而获取到 Metrics 指标，默认时间是 15s 抓取一次，配置项如下：

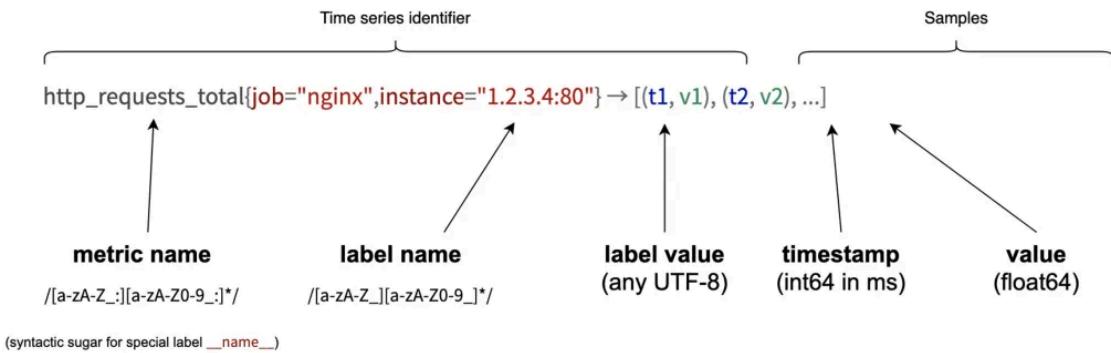
```

global:
  scrape_interval: 15s

```

抓取到的指标会被以时间序列的形式保存在内存中，并且定时刷到磁盘上，默认是两个小时回刷一次。并且为了防止 Prometheus 发生崩溃或重启时能够恢复数据，Prometheus 也提供了类似 MySQL 中 binlog 一样的预写日志，当 Prometheus 崩溃重启时，会读这个预写日志来恢复数据。

Metric



Prometheus 采集的所有指标都是以时间序列的形式进行存储，每一个时间序列有三部分组成：

- **指标名和指标标签集合**: metric_name{<label1=v1>,<label2=v2>,...},
 - 指标名：表示这个指标是监控哪一方面的状态，比如 http_request_total 表示：请求数量；
 - 指标标签，描述这个指标有哪些维度，比如 http_request_total 这个指标，有请求状态码 code = 200/400/500，请求方式：method = get/post 等，实际上指标名称实际上是以标签的形式保存，这个标签是**name**，即：**name=**。
- **时间戳**：描述当前时间序列的时间，单位：毫秒。
- **样本值**：当前监控指标的具体数值，比如 http_request_total 的值就是请求数是多少。

Prometheus指标类型

Prometheus 底层存储上其实并没有对指标做类型的区分，都是以时间序列的形式存储，但是为了方便用户的使用和理解不同监控指标之间的差异，Prometheus 定义了 4 种不同的指标类型：**计数器 counter**，**仪表盘 gauge**，**直方图 histogram**，**摘要 summary**。

1. Counter 计数器：**Counter** 类型和 redis 的自增命令一样，**只增不减**，通过 Counter 指标可以统计 Http 请求数量，请求错误数，接口调用次数等单调递增的数据。同时可以结合 increase 和 rate 等函数统计变化速率，后续我们会提到这些内置函数。
2. Gauge 仪表盘：和 Counter 不同，Gauge 是可增可减的，可以反映一些动态变化的数据，例如当前内存占用，CPU 利用，Gc 次数等动态可上升可下降的数据，在 Prometheus 上通过 Gauge，可以不用经过内置函数直观的反映数据的变化情况。
3. Histogram 直方图：Histogram 是一种直方图类型，可以观察到指标在各个不同的区间范围的分布情况，如下图所示：可以观察到请求耗时在各个桶的分布。
4. Summary 摘要：Summary 也是用来做统计分析的，和 Histogram 区别在于，Summary 直接存储的就是百分位数，如下所示：可以直观的观察到样本的中位数，P90 和 P99。

Prometheus指标导出

指标导出有两种方式：

1. 一种是使用 **Prometheus 社区提供的定制好的 Exporter** 对一些组件诸如 MySQL，Kafka 等的指标作导出，
2. 也可以利用**社区提供的 Client** 来自定义指标导出。

自定义指标，大体流程为：

1. 定义指标（类型、名字、帮助信息）
2. 注册指标

3. 定义标签

4. 添加值

```
package main

import (
    "fmt"
    "net/http"
    "github.com/prometheus/client_golang/prometheus"
)

var (
    MyCounter prometheus.Counter
)

// init 注册指标
func init() {
    // 1. 定义指标 (类型, 名字, 帮助信息)
    MyCounter = prometheus.NewCounter(prometheus.CounterOpts{
        Name: "my_counter_total",
        Help: "自定义counter",
    })
}

// 2. 注册指标
prometheus.MustRegister(MyCounter)
}

// SayHello
func SayHello(w http.ResponseWriter, r *http.Request) {
    // 接口请求量递增
    MyCounter.Inc()
    fmt.Fprintf(w, "Hello wrold!")
}
```

PromQL

通过prometheus提供的PromQL可以查询导出的指标，**PromQL 是 Prometheus 为我们提供的函数式的查询语言**，查询表达式有四种类型：

- **字符串**：只作为某些内置函数的参数出现
- **标量**：单一的数字值，可以是函数参数，也可以是函数的返回结果
- **瞬时向量**：某一时刻的时序数据。直接通过指标名即可进行查询，查询结果是当前指标最新的时间序列，比如查询 Gc 累积消耗的时间，并设置指定标签进行过滤（也支持正则表达式）：
`go_gc_duration_seconds_count{instance="127.0.0.1:9600"}`
- **区间向量**：某一时间区间内的时序数据集合。范围查询的结果集就是区间向量，可以通过[]指定时间来做范围查询，查询 5 分钟内的 Gc 累积消耗时间：
`go_gc_duration_seconds_count{}[5m]`

PromQL提供的常用内置函数

1. **rate** 函数可以用来求指标的平均变化速率：**rate函数=时间区间前后两个点的差 / 时间范围**，一般 **rate** 函数可以用来求某个时间区间内的请求速率，也就是我们常说的 QPS。
2. 但是 **rate** 函数只是算出来了某个时间区间内的平均速率，没办法反映突发变化，假设在一分钟的时间区间里，前 50 秒的请求数量都是 0 到 10 左右，但是最后 10 秒的请求数量暴增到 100 以上，这时候算出来的值可能无法很好的反映这个峰值变化。这个问题可以通过 **irate** 函数解决，**irate 函数求出来的是瞬时变化率：时间区间内最后两个样本点的差 / 最后两个样本点的时间差。**
3. **聚合函数：Sum() by() without()**：sum 将指标值相加，by 进行分组，without 不通过 xx 分组

Grafana可视化

除了可以利用 Prometheus 提供的 webUI 可视化我们的指标外，还可以接入 Grafana 来做指标的可视化。

1. 对接数据源，配置 Prometheus 的地址
2. 创建仪表盘，编辑仪表盘，编写 PromQL 完成查询和可视化

监控告警

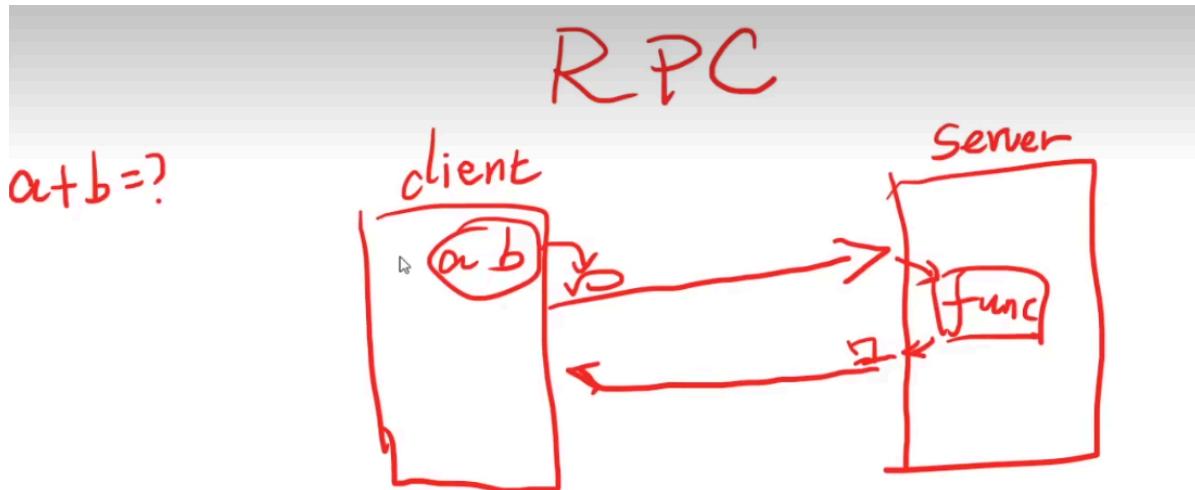
AlertManager 是 prometheus 提供的告警信息下发组件，包含了对告警信息的分组，下发，静默等策略。配置完成后可以在 webui 上看到对应的告警策略信息。告警规则也是基于 PromQL 进行定制的。

在 `prometheus.yml` 中配置告警配置文件，需要配置上 `alertmanager` 的地址和告警文件的地址。配置告警信息，例如告警发送地址，告警内容模版，分组策略等都在 `alertmanager` 的配置文件中配置。

RPC

分布式计算中，**远程过程调用**（英语：Remote Procedure Call，RPC）是一个**计算机通信协议**（也可以说是一种思想，一种框架）。该协议允许运行于一台计算机的程序调用另一个地址空间（通常为一个开放网络的一台计算机）的子程序，而程序员就像调用本地程序一样，无需额外地为这个交互作用编程（**无需关注细节**）。RPC是一种**服务器-客户端**（Client/Server）模式，经典实现是一个通过发送请求-接受回应进行信息交互的系统。

RPC是一种进程间通信的模式，程序分布在不同的地址空间里。如果在同一主机里，RPC可以通过不同的虚拟地址空间（即便使用相同的物理地址）进行通讯，而在不同的主机间，则通过不同的物理地址进行交互。许多技术（通常是不兼容）都是基于这种概念而实现的。



为什么要RPC

解耦。本机主要运行的是A功能，但是A功能的过程中会需要B功能，但是B功能不太属于A功能范围之类的事情，那么就会借助RPC去请求实现B功能的机器。

HTTP和RPC

RPC是一种思想、框架。HTTP可以是实现RPC的一种方式，比如gRPC就是基于HTTP2实现的。

HTTP与RPC的区别

1. 通信模型：

- **HTTP**：是一种基于请求-响应模型的协议，主要用于客户端与服务器之间的通信。工作在应用层，是一种通用的协议，可以用于各种类型的通信（如浏览网页、RESTful API等）。**HTTP请求通常用于获取或提交资源，如网页、文件等。**
- **RPC**：工作在更高的抽象层次，直接支持函数调用的语义，通常用于构建分布式系统和微服务架构。是一种**通过网络调用远程服务的方法，旨在使远程服务调用看起来就像本地调用一样。RPC隐藏了网络通信的细节，使开发者可以专注于业务逻辑。**

2. 数据格式：

- **HTTP**：通常使用文本格式（如HTML、XML、JSON）进行数据传输。**HTTP本身不规定数据格式，但在RESTful API中，JSON是最常用的数据格式。**
- **RPC**：通常使用二进制格式（如Protocol Buffers、Thrift）进行数据传输，以提高效率和性能。

3. 性能：

- **HTTP**：由于其通用性和文本格式，可能会有较高的开销，特别是在频繁的短消息通信中。

- **RPC**: 通常使用高效的二进制序列化格式，减少了网络传输的开销，性能更高。

4. 协议复杂度：

- **HTTP**: 相对简单，广泛使用，有成熟的生态系统和工具支持。
- **RPC**: 实现较为复杂，需要处理序列化、反序列化、网络传输、错误处理等多个方面。

为什么在有了HTTP之后我们还需要RPC

1. **效率和性能**: RPC通常使用高效的二进制序列化格式，传输效率更高，适合高性能和低延迟的场景。
2. **开发便利性**: RPC使得远程调用像本地调用一样简单，隐藏了网络通信的复杂性，缩短了开发周期，减少了错误。
3. **协议和平台独立性**: RPC框架（如gRPC、Thrift）通常支持多种编程语言和平台，使得跨语言、跨平台的系统集成更加容易。
4. **功能丰富**: 现代RPC框架提供了很多高级功能，如负载均衡、服务发现、错误重试、流式传输等，这些功能在HTTP协议中需要额外实现。
5. **微服务架构**: 在微服务架构中，服务之间的通信频繁且复杂，RPC提供了更高效和灵活的方式来实现服务间的调用和协作。

提高rpc的性能

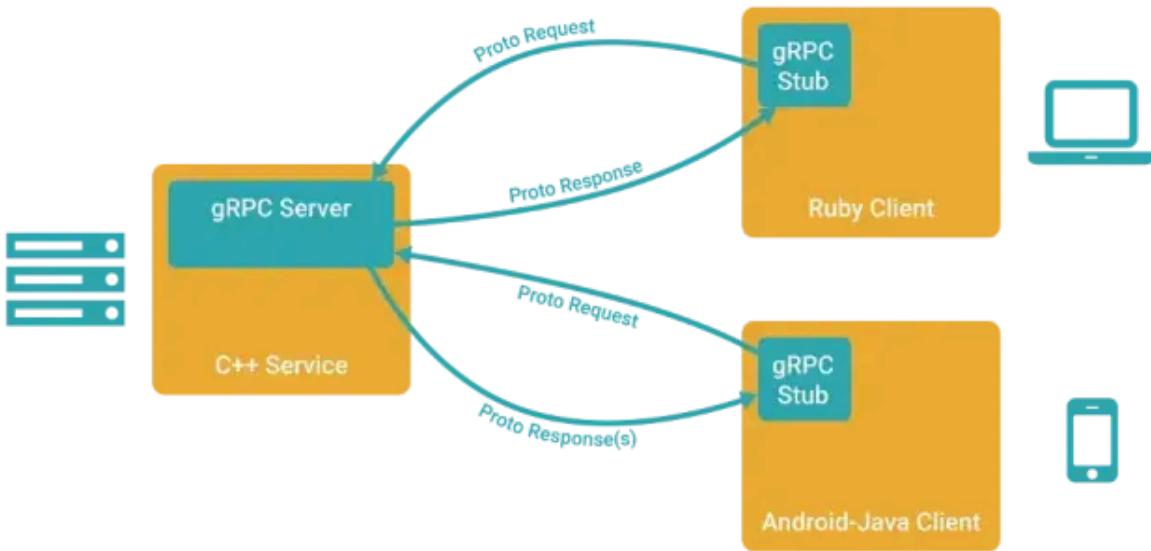
1. 参数和结果传输，序列化和反序列化
2. 网络I/O传输

gRPC

概述

gRPC 是谷歌推出的一个开源、高性能的 RPC 框架。默认情况下使用 protoBuf 进行序列化和反序列化，并基于 HTTP/2 传输报文，带来诸如多请求复用一个 TCP 连接(所谓的多路复用)、双向流、流控、头部压缩等特性。gRPC 目前提供 C、Go 和 JAVA 等语言版本，对应 gRPC、gRPC-Go 和 gRPC-JAVA 等开发框架。

在 gRPC 中，开发者可以像调用本地方法一样，通过 gRPC 的客户端调用远程机器上 gRPC 服务的方法，gRPC 客户端封装了 HTTP/2 协议数据帧的打包、以及网络层的通信细节，把复杂留给框架自己，把便捷提供给用户。gRPC 基于这样的一个设计理念：定义一个服务，及其被远程调用的方法(方法名称、入参、出参)。在 gRPC 服务端实现这个方法的业务逻辑，并在 gRPC 服务端处理远程客户端对这个 RPC 方法的调用。在 gRPC 客户端也拥有这个 RPC 方法的存根(stub)。gRPC 的客户端和服务端都可以用任何支持 gRPC 的语言来实现，例如一个 gRPC 服务端可以是 C++语言编写的，以供 Ruby 语言的 gRPC 客户端和 JAVA 语言的 gRPC 客户端调用，如下图所示：



gRPC 默认使用 ProtoBuf 对请求/响应进行序列化和反序列化，这使得传输的请求体和响应体比 JSON 等序列化方式包体更小、更轻量。gRPC 基于 HTTP/2 协议传输报文，HTTP/2 具有多路复用、头部压缩等特性，基于 HTTP/2 的帧设计，实现了多个请求复用一个 TCP 连接，基本解决了 HTTP/1.1 的队头阻塞问题，相对 HTTP/1.1 带来了巨大的性能提升。

HTTP2

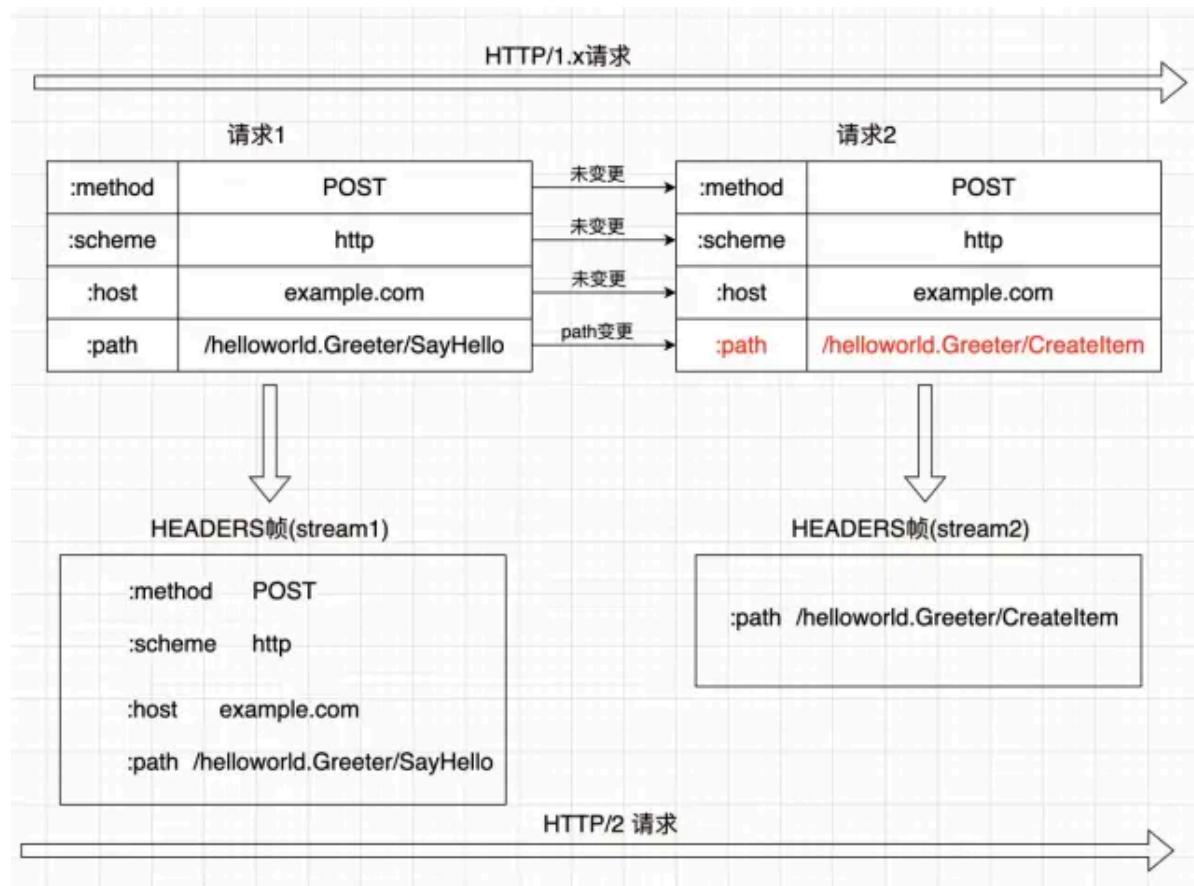
HTTP1.1存在的问题

- 队头阻塞：**HTTP 的队头阻塞等特性导致基于 HTTP 的应用程序性能有较大影响。**队头阻塞是指顺序请求的一个请求必须处理完才能处理后续的其他请求，当一个请求被阻塞时会给应用程序带来延迟。**虽然 HTTP/1.1 提供了流水线(request pipeline)的请求操作，但是由于受到 HTTP 自身协议的限制，无法消除 HTTP 的队头阻塞带来的延迟。**为了减少延迟，需要 HTTP 的客户端与服务器建立多个连接实现并发处理请求，降低延迟。**然而，在高并发情况下，大量的网络连接可能耗尽系统资源，可以使用连接池模式只维持固定的连接数可以防止服务的资源耗尽。连接池连接数的设置在对性能要求极高的应用程序也是一个挑战，需要根据实际机器配置的压测情况确定。
- 头字段重复并且冗余：**HTTP 头字段重复且冗长，导致网络传输不必要的冗余报文，以及初始 TCP 拥塞窗口很快被填满。一个 TCP 连接处理大量请求是会导致较大的延迟。

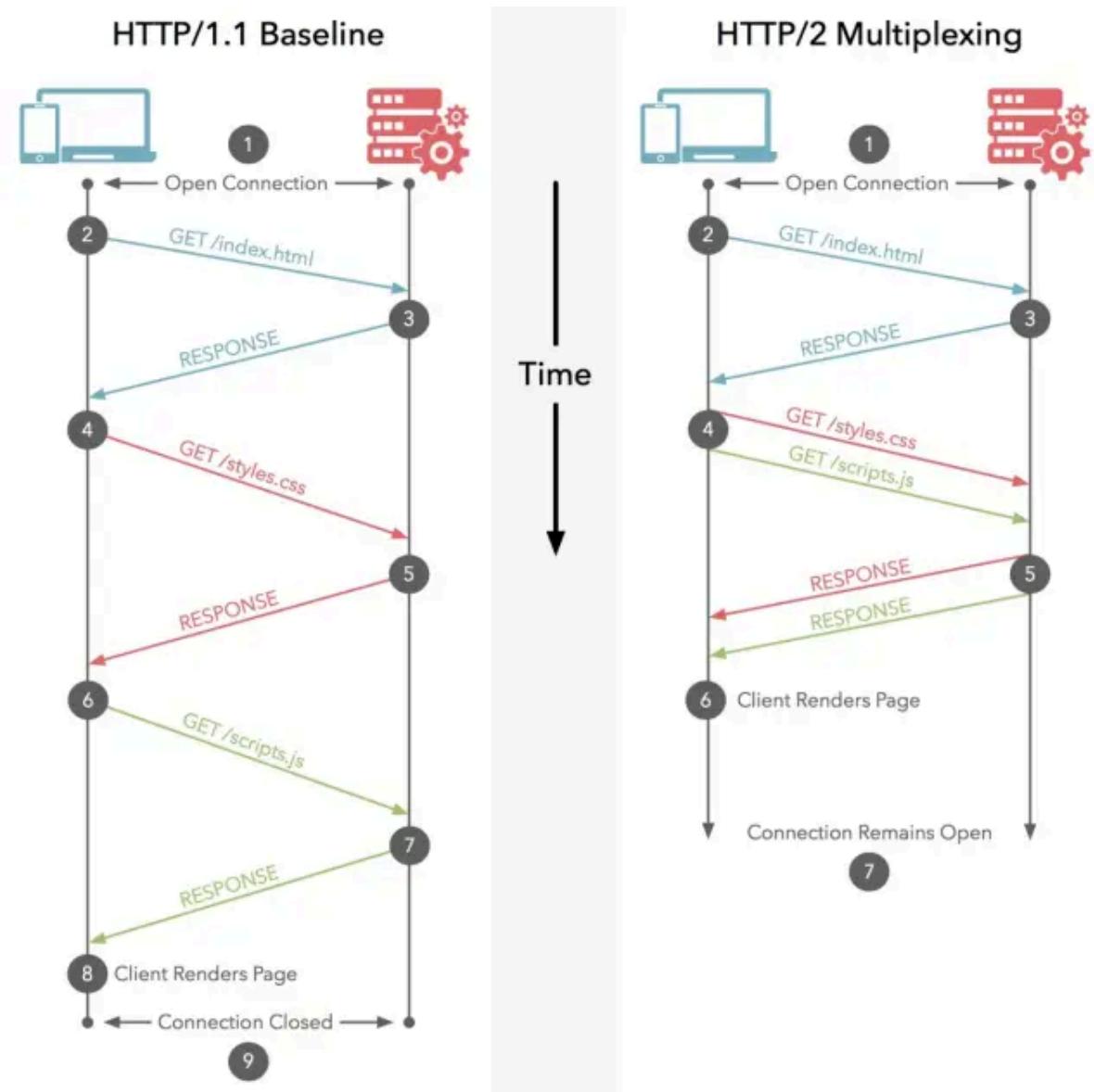
HTTP/2 通过优化 HTTP 的报文定义，**允许同一个网络连接上并发交错的处理请求和响应，并通过减少 HTTP 头字段的重复传输、压缩 HTTP 头，提高了处理性能。**HTTP 每次网络传输会携带通信的资源、浏览器属性等大量冗余头信息，为了减少这些重复传输的开销，HTTP/2 会压缩这些头部字段：

1. 基于 HTTP/2 协议的客户端和服务器使用“头部表”来跟踪与存储发送的键值对，对于相同的键值对数据，不用每次请求和响应都发送；
2. 头部表在 HTTP/2 的连接有效期内一直存在，由客户端和服务器共同维护更新；
3. 每个新的 HTTP 头键值对要么追加，要么替换头部表原来的值。

举个例子，有两个请求，在 HTTP/1.x 中，请求 1 和请求 2 都要发送全部的头数据；在 HTTP/2 中，请求 1 发送全部的头数据，请求 2 仅仅发送变更的头数据，这样就可以减少冗余的数据，降低网络开销。如下图所示：



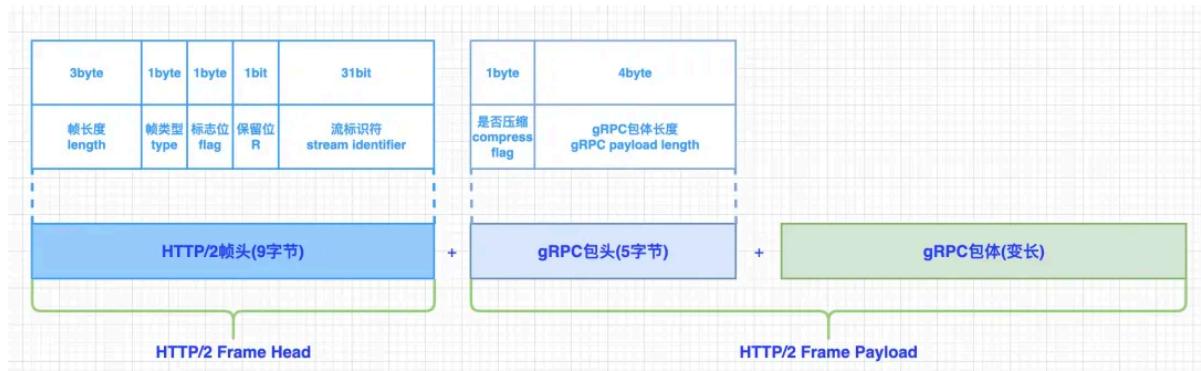
这里再举个例子说明 HTTP/1.x 和 HTTP/2 处理请求的差异，浏览器打开网络要请求/index.html、styles.css 和/scripts.js 三个文件，**基于 HTTP/1.x 建立的连接只能先请求/index.html,得到响应后再请求 styles.css，得到响应后再获取/scripts.js。而基于 HTTP/2 一个网络连接在获取/index.html 后,可以同时获取 styles.css 和/scripts.js 文件，如下图所示：**



HTTP/2 对服务资源和网络更友好，相对与 HTTP/1.x，处理同样量级的请求，HTTP/2 的需要建立的 TCP 连接数更少。这主要得益于 HTTP/2 使用二进制数据帧来传输数据，使得一个 TCP 连接可以同时处理多个请求而不用等待一个请求处理完成再处理下一个。从而充分发掘了 TCP 的并发能力。

gRPC协议

gRPC 基于 HTTP/2/协议进行通信，使用 ProtoBuf 组织二进制数据流，gRPC 的协议格式如下图：



从以上图可知，gRPC 协议在 HTTP 协议的基础上，对 HTTP/2 的帧的有效包体(Frame Payload)做了进一步编码：gRPC 包头(5 字节)+gRPC 变长包头，其中：

1. 5 字节的 gRPC 包头由：1 字节的压缩标志(compress flag)和 4 字节的 gRPC 包头长度组成；

2. gRPC 包体长度是变长的，其是这样的一串二进制流：使用指定序列化方式(通常是 ProtoBuf)序列化成字节流，再使用指定的压缩算法对序列化的字节流压缩而成的。如果对序列化字节流进行了压缩，gRPC 包头的压缩标志为 1。

ProtoBuf

Proto Buffer 是一种语言中立的、平台中立的、可扩展的序列化结构数据的方法。序列化指的是将一个数据结构或者对象转换为某种能被跨平台识别的字节格式，以便进行跨平台存储或者网络传输。常见的序列化结果有JSON、Protobuf、XML、YAML等。

Protobuf 所具备的特性：

- **语言中立与平台中立**：Protobuf 不依赖于某一种特殊的编程语言或者操作系统的机制，意味着我们可以在多种编程环境中直接使用。（大部分序列化机制其实都具有这个特性，但是某些编程语言提供了内置的序列化机制，这些机制可能只在该语言的生态系统内有效，例如 Python 的 pickle 模块）
- **可拓展**：Protobuf 可以在不破坏现有代码的情况下，更新消息类型。具体表现为**向后兼容与向前兼容**。
- **更小更快**：序列化的目的之一是进行网络传输，在传输过程中数据流越小传输速度自然越快，可以整体提升系统性能。**Protobuf 利用字段编号与特殊的编码方法巧妙地减少了要传递的信息量，并且使用二进制格式，相比于 JSON 的文本格式，更节省空间。**

使用ProtoBuf的基本流程：

1. 定义数据结构：首先，开发者使用.proto文件来定义数据结构。这个文件是一种领域特定语言(DSL)，用来描述数据消息的结构，包括字段名称、类型（如整数、字符串、布尔值等）、字段标识号等等。

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    string email = 3;
}
```

为什么需要额外定义 proto 文件呢？Proto Buffer 能够利用该文件中的定义，去做很多方面的事情，例如生成多种编程语言的代码方便跨语言服务通信，例如借助字段编码与类型来压缩数据获得更小的字节流，再例如提供一个更加准确类型系统，为数据提供强类型保证。听上去或许比较抽象，这里先用一个简单的例子来说明 proto 文件的好处之一：如果我们采用 JSON 进行序列化，由于 JSON 的类型比较宽松，比如数字类型不区分整数和浮点数，这可能会导致在不同的语言间交换数据时出现歧义，而 proto 中我们可以定义 float int32 等等更加具体的类型。

2. 生成工具函数代码：接下来，我们需要使用 protobuf 编译器 (protoc) 处理.proto文件，生成对应目标语言（如C++、Java、Python等）的源代码。这些代码包含了数据结构的类定义（称为消息类）以及用于序列化和反序列化的函数。

3. 使用生成的代码进行网络传输：当需要发送数据或者接收到消息对象时，我们就可以利用生成代码中所提供的序列化与反序列化函数对数据进行处理了。

值得注意的是，在利用 Protobuf 进行网络数据传输时，**确保通信双方拥有一致的 .proto 文件至关重要**。缺少了相应的 .proto 文件，通信任何一方都无法生成必要的工具函数代码，进而无法解析接收到的消息数据。与 JSON 这种文本格式不同，后者即便在没有 JSON.parse 反序列化函数的情况下，人们仍能大致推断出消息内容。相比之下，Protobuf 序列化后的数据是**二进制字节流**，它并不适合人类阅读，且必须通过特定的反序列化函数才能正确解读数据。Protobuf 的这种设计在提高数据安全性方面具

有优势，因为缺少 `.proto` 文件就无法解读数据内容。然而，这也意味着在通信双方之间需要维护一致的 `.proto` 文件，随着项目的扩展，这可能会带来额外的维护成本。

ProtoBuf的语法

```
syntax = "proto3"; // 指定使用的语法版本，默认情况下是 proto2

// 定义包含四个字段的消息 User
message User {
    uint32 id = 1; // 字段 id 的类型为 uint32，编号 1
    string name = 2; // 字段 name 的类型为 string，编号 2
    string email = 3; // ...
    string password = 4;
}
```

- 需要注意的是 `syntax = "proto3";` 必须是文件的第一个非空的非注释行。

在声明 protobuf 文件的语法版本之后，我们就可以开始定义消息结构。在定义字段时，必须指明**字段的类型，名称以及一个唯一的字段编号**。

- 类型**: proto 提供了丰富的类型系统，包括无符号整数 `uint32`、有符号整数 `sint32`、浮点数 `float`、字符串、布尔等等，你可以在[这个链接](#)中查看完整的类型描述。当然，除了为字段指定基本的类型意外，你还可以为其指定 `enum` 或是自定义的消息类型。
- 字段编号**: 每个字段都需要一个唯一的数字标识符，也就是**字段编号**。这些编号在序列化和反序列化过程中至关重要，因为他们将替代字段名称出现在序列化后二进制数据流中。在使用 JSON 序列化数据时，其结果中往往包含人类可读的字段名称，例如 `{"id": "123456"}`，但是在 protobuf 中，序列化后的结果中只会包含字段编号而非字段名称，例如在本例中，`id` 的编号为 1，那我序列化后的结果只会包含 1 而非 `id`。这种方法有点类似于 HTTP 的头部压缩，可以显著减少传输过程中的数据流大小。事实上**字段编号的使用是 proto 中非常重要的一环，在使用中务必遵循以下原则**：
 - 字段编号一旦被分配后就不应更改，这是为了保持向后兼容性。
 - 编号在 `[1,15]` 范围内的字段编号在序列化时只占用一个字节。因此，为了优化性能，对于频繁使用的字段，尽可能使用该范围内的数字。同时也要为未来可能添加的常用字段预留一些编号（不要一股脑把 15 之内的编号都用了！）
 - 字段编号从 1 开始，最大值是 29 位，字段号 `19000,19999` 是为 Protocol Buffers 实现保留的。如果在消息定义中使用这些保留字段号之一，协议缓冲区编译器将报错提示。
- (可选) 字段标签**: 除了上述三个必须设置的元素外，你还可以选择性设置**字段标签**：
 - `optional`: 之后字段被显式指定时，才会参与序列化的过程，否则该字段将保持默认值，并且不会参与序列化。在 proto3 中所有字段默认都是可选的，并不需要使用这个关键字来声明字段，除非在某些情况下我们需要区分字段是否被设置过。在 proto3 中，如果字段未被设置，它将不会包含在序列化的消息之中。在 JavaScript 中，如果一个字段被指定为 `optional` 并且没有设置值，在解析后的对象将不会包含该字段（如果没有指定 `optional` 将会包含该字段的默认值）。
 - `repeated`: 以重复任意次数（包括零次）的字段。它们本质上是对应数据类型列表的动态数组。
 - `map`: 成对的键/值字段类型，语法类似 Typescript 中的 `Record`。

- **保留字段**: 如果你通过完全删除字段或将其注释来更新消息类型，则未来其他开发者对类型进行自己的更新时就有可能重用字段编号。当旧版本的代码遇到新版本生成的消息时，由于字段编号的重新分配，可能会引发解析错误或不预期的行为。为了避免这种潜在的兼容性问题，protobuf 提供 `reserved` 关键字来明确标记不再使用的字段编号或标识符，如果将来的开发者尝试使用这些标识符，proto 编译器将会报错提醒。

```
message Foo {
    reserved 2, 15, 9 to 11, 40 to max;
    // 9 to 11 表示区间 [9,11]，40 to max 表示区间 [40, 编号的最大值]
    reserved "foo", "bar";
}
```

- **默认值**: 在解析消息时，如果编码的消息中并不包含某个**不具有字段标签**的字段，那么解析后对象中的响应字段将设置为该字段的默认值。默认值的规则如下：
 - 对于 `string`，默认值为空字符串
 - 对于 `byte`，默认值为空字节
 - 对于 `bool`，默认值为 `false`
 - 对于数字类型，默认值为 0
 - 对于 `enum` 类型，默认值为第一个定义的枚举值（编号为 0）
- 假设某个字段具有 `optional` 字段标签（或是其他什么的标签），那么在解析后的对象中将不会存在这些字段。

编码原理

对于 protobuf 它的编码是很紧凑的，我们先看一下 message 的结构，举一个简单的例子：

```
message Student {
    string name = 1;
    int32 age = 2;
}
```

message 是一系列键值对，编码过之后实际上只有 tag 序列号和对应的值，这一点相比我们熟悉的 json 很不一样，所以对于 protobuf 来说没有 `.proto` 文件是无法解出来的：

Message



luozhiyun

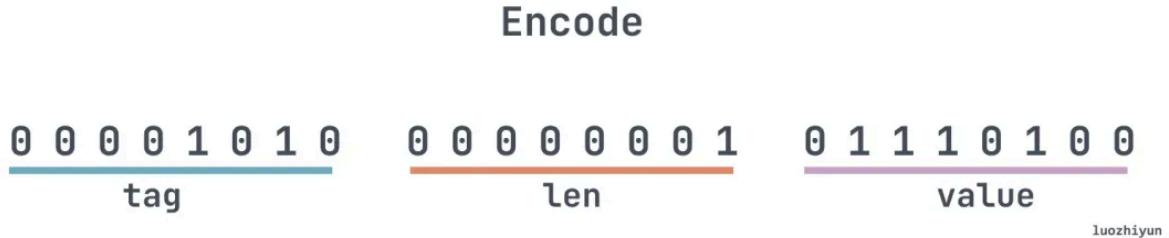
对于 tag 来说，它保存了 message 字段的编号以及类型信息，我们可以做个实验，把 name 这个 tag 编码后的二进制打印出来：

```

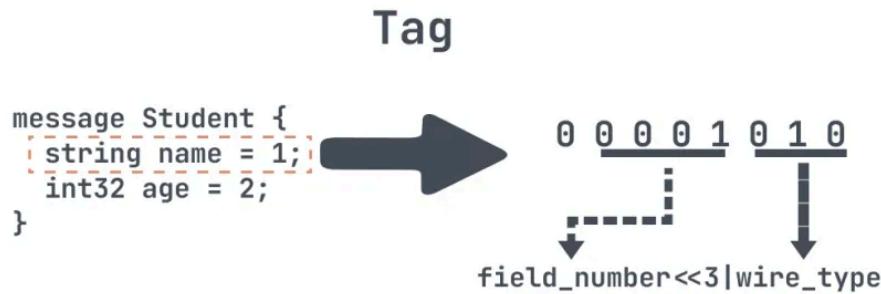
func main() {
    student := student.Student{}
    student.Name = "t"
    marshal, _ := proto.Marshal(&student)
    fmt.Println(fmt.Sprintf("%08b", marshal)) // 00001010 00000001 01110100
}

```

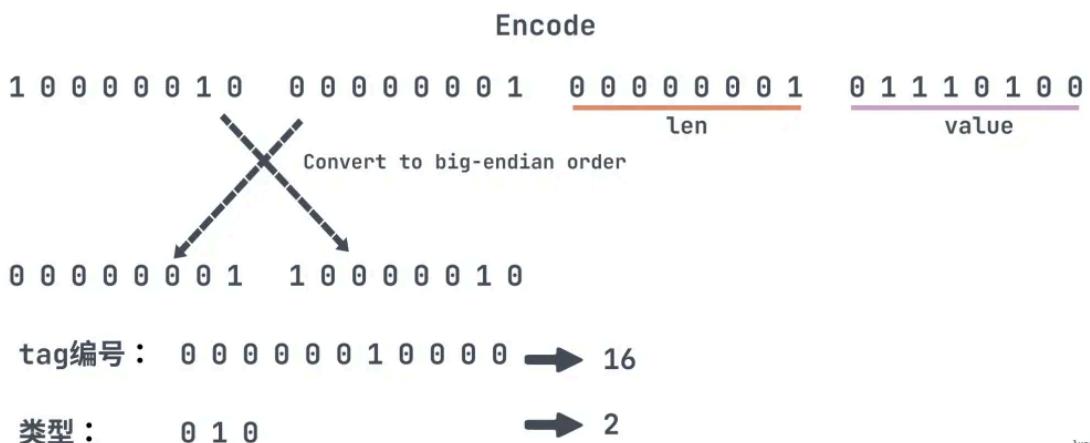
打印出来的结果是这样：



上图中，由于 name 是 string 类型，所以第一个 byte 是 tag，第二个 byte 是 string 的长度，第三个 byte 是值，也就是我们上面设置的 “t”。我们下面先看看 tag：



tag 里面会包含两部分信息：字段序号，字段类型，计算方式就是上图的公式。上图中将 name 这个字段序列化成二进制我们可以看到，第一个 bit 是标记位，表示是否字段结尾，这里是 0 表示 tag 已结尾，tag 占用 1byte；接下来 4 个 bit 表示的是字段序号，所以范围 1 到 15 中的字段编号只需要 1 bit 进行编码，我们可以做个实验看看，将 tag 改成 16：



由上图所示，每个 byte 第一个 bit 表示是否结束，0 表示结束，所以上面 tag 用两个 byte 表示，并且 protobuf 是小端编码的，需要转成大端方便阅读，所以我们可以知道 tag 去掉每个 byte 第一个 bit 之后，后三位表示类型，是 3，其余位是编号表示 16。

所以从上面编码规则我们也可以知道，字段尽可能精简一些，字段尽量不要超过 16 个，这样就可以用一个 byte 表示了。

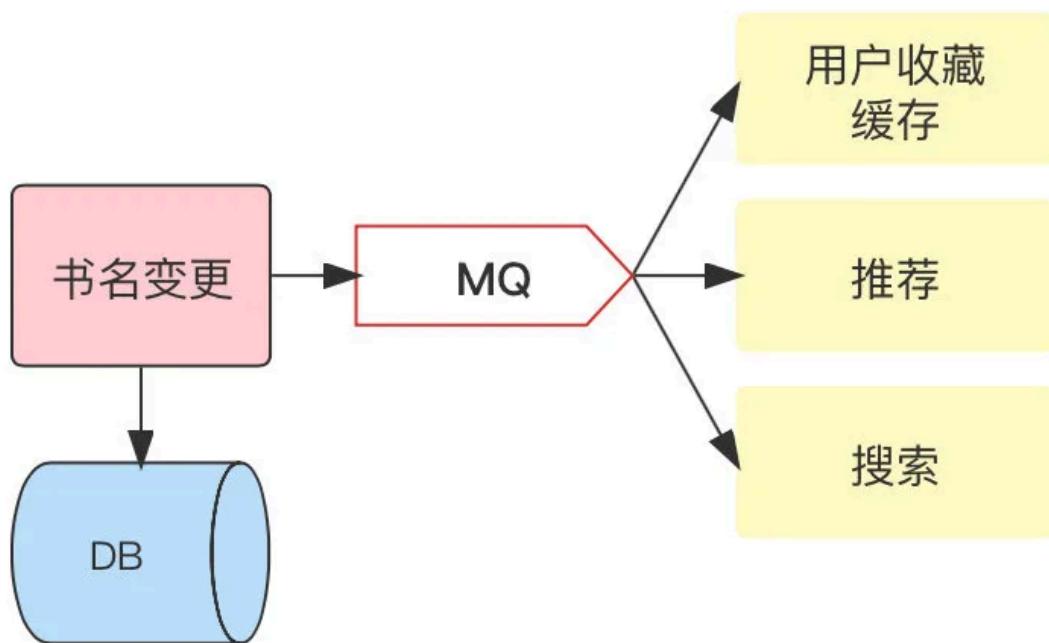
同时我们也可以知道，protobuf 序列化是不带字段名的，所以如果客户端的 proto 文件只修改了字段名，请求服务端是安全的，服务端继续用根据序列编号还是解出来原来的字段。但是需要注意的是不要修改字段类型。

接下来我们看看类型，protobuf 共定义了 6 种类型，其中两种是废弃的：

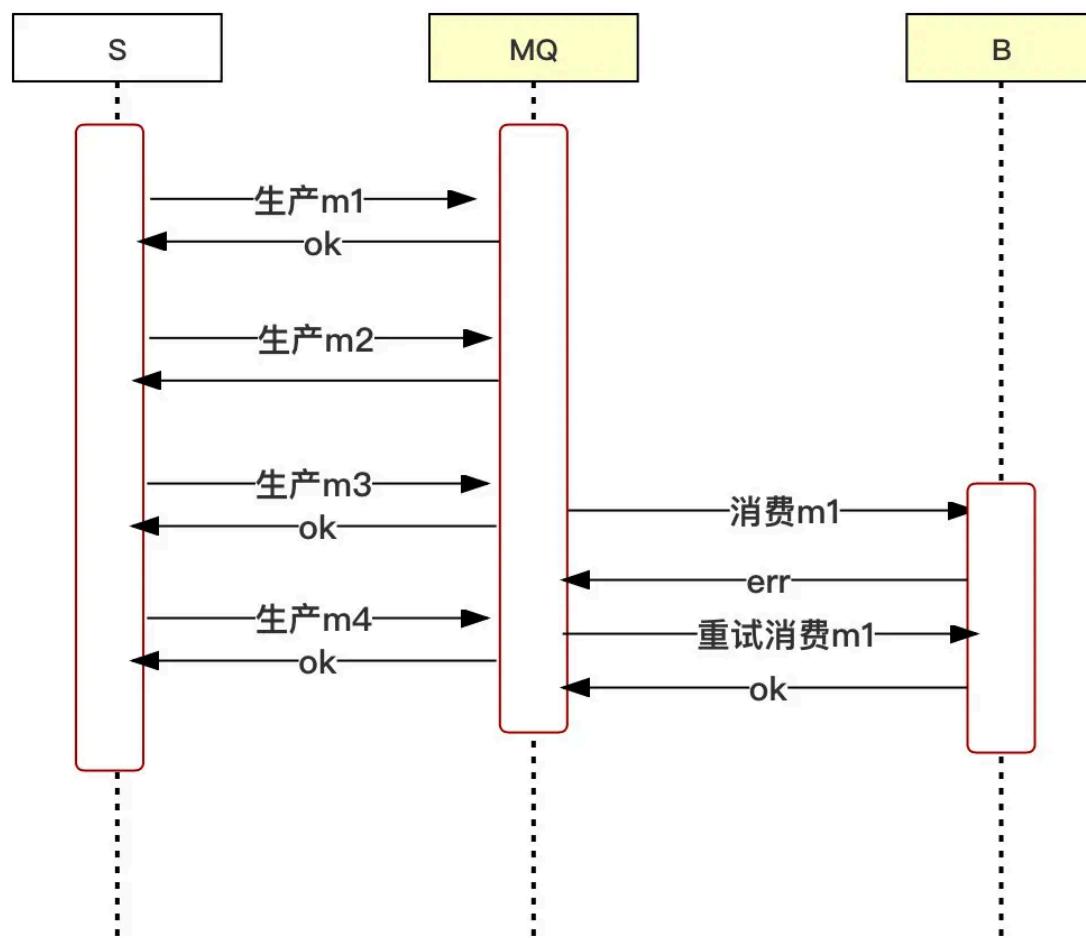
ID	Name	Used For
0	VARINT	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	I64	fixed64, sfixed64, double
2	LEN	string, bytes, embedded messages, packed repeated fields
3	SGROUP	group start (deprecated)
4	EGROUP	group end (deprecated)
5	I32	fixed32, sfixed32, float

上面的例子中，Name 是 string 类型所以上面 tag 类型解出来是 010，也就是 2。

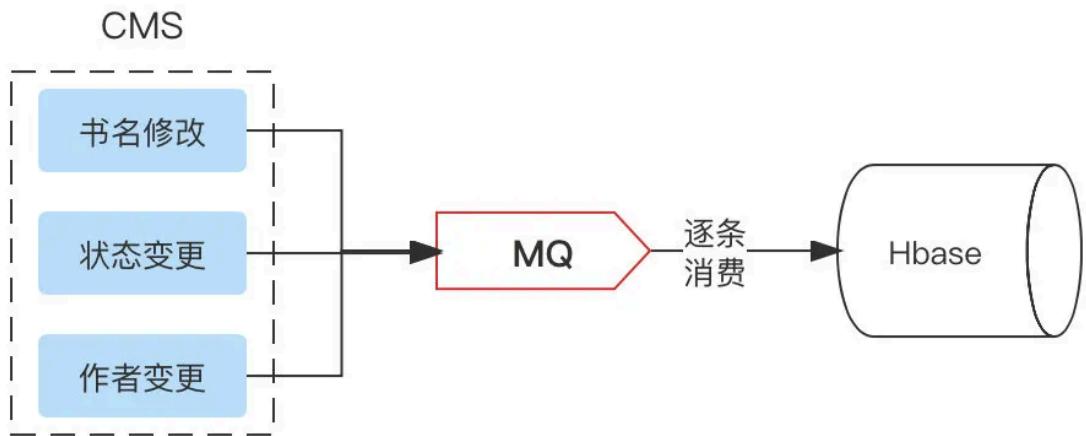
消息队列的常用场景



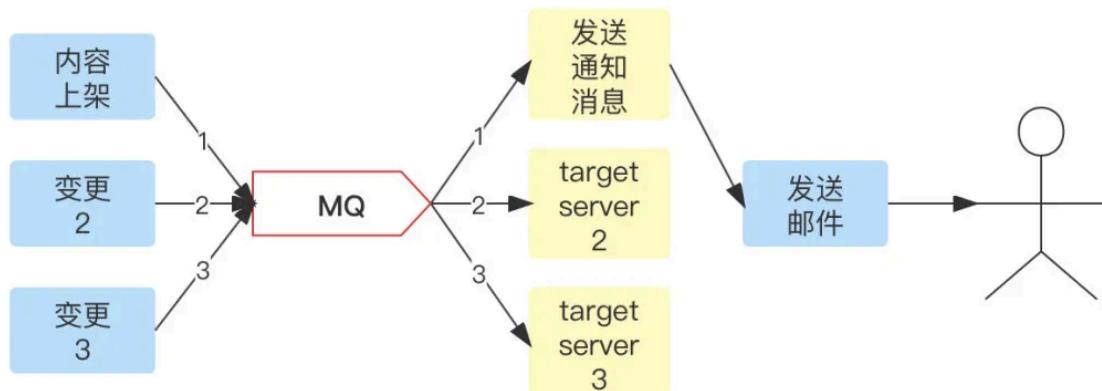
解耦：多个服务监听、处理同一条消息，避免多次 rpc 调用。



异步消息：消息发布者不用等待消息处理的结果。



削峰填谷：较大流量、写入场景，为下游 I/O 服务抗流量。当然大流量下就需要使用其他方案了。



消息驱动框架：在事件总线中，服务通过监听事件消息驱动服务完成相应动作。

消息队列模式

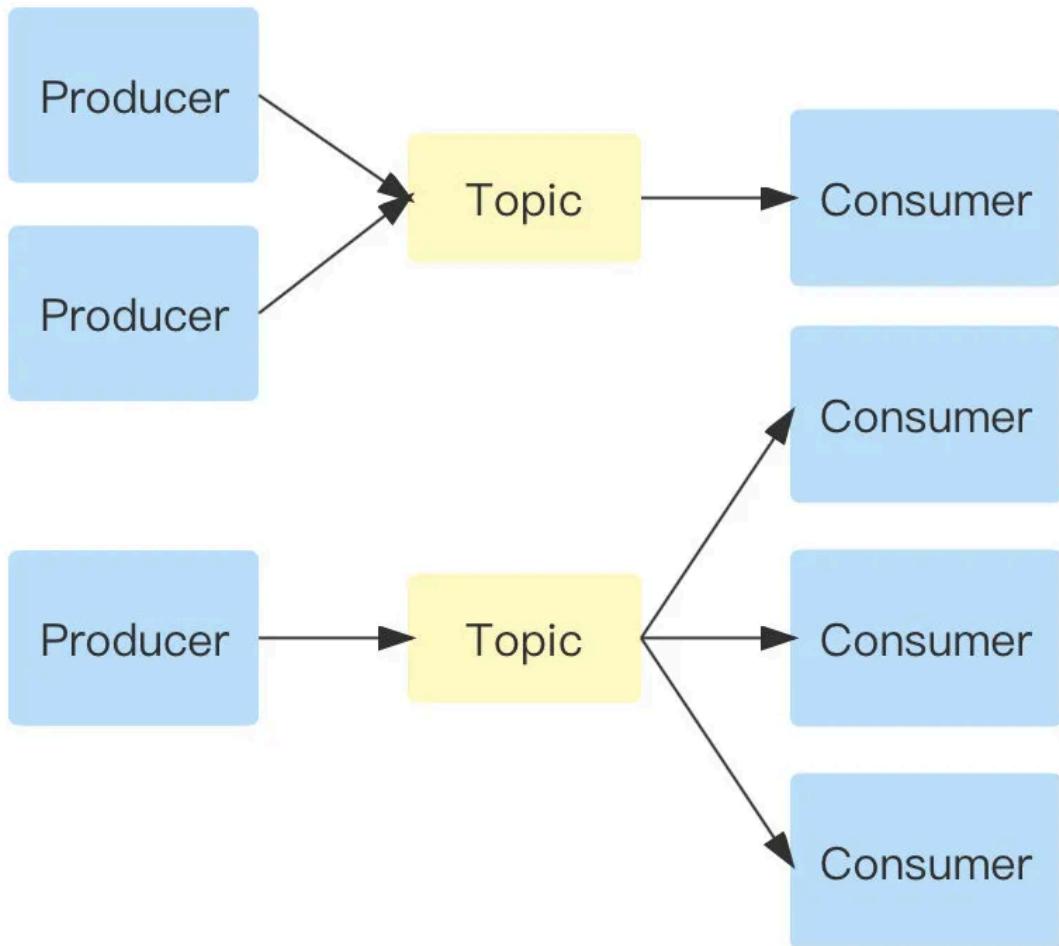
点对点模式，不可重复消费

多个生产者可以向同一个消息队列发送消息，一个消息在被一个消息者消费成功后，这条消息会被移除，其他消费者无法处理该消息。如果消费者处理一个消息失败了，那么这条消息会重新被消费。



发布/订阅模式

发布订阅模式需要进行注册、订阅，根据注册消费对应的消息。多个生产者可以将消息写到同一个 Topic 中，多种消息可以被同一个消费者消费。一个生产者生产的消息，同样也可以被多个消费者消费，只要他们进行过消息订阅。



选型参考

- 消息顺序：发送到队列的消息，消费时是否可以保证消费的顺序；
- 伸缩：当消息队列性能有问题，比如消费太慢，是否可以快速支持扩容；当消费队列过多，浪费系统资源，是否可以支持缩容。
- 消息留存：消息消费成功后，是否还会继续保留在消息队列；
- 容错性：当一条消息消费失败后，是否有一些机制，保证这条消息一定能成功，比如异步第三方退款消息，需要保证这条消息消费掉，才能确定给用户退款成功，所以必须保证这条消息消费成功的准确性；
- 消息可靠性：是否会存在丢消息的情况，比如有 A/B 两个消息，最后只有 B 消息能消费，A 消息丢失；
- 消息时序：主要包括“消息存活时间”和“延迟消息”；
- 吞吐量：支持的最高并发数；
- 消息路由：根据路由规则，只订阅匹配路由规则的消息，比如有 A/B 两者规则的消息，消费者可以只订阅 A 消息，B 消息不会消费。

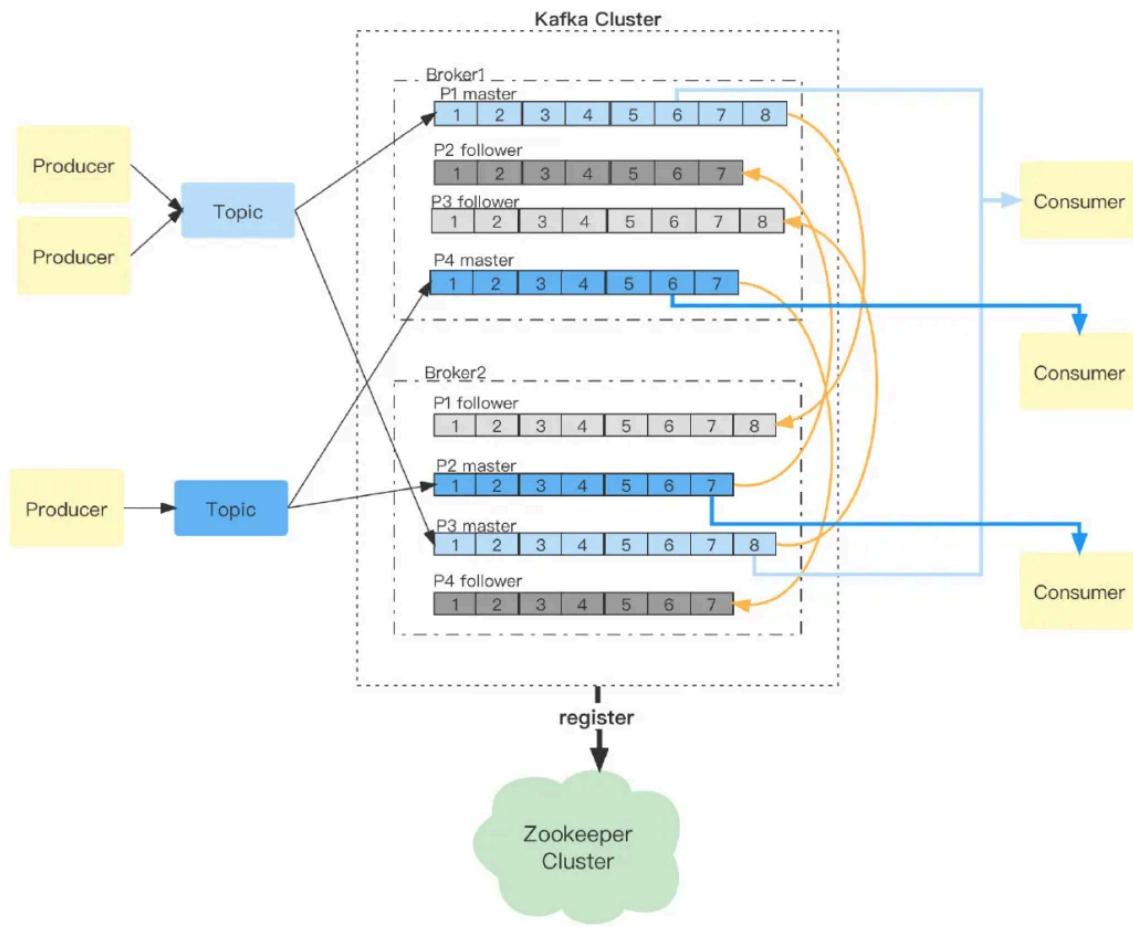
Kafka

Kafka 是由 Apache 软件基金会开发的一个开源流处理平台，由 Scala 和 Java 编写。该项目的目标是为处理实时数据提供一个统一、高吞吐、低延迟的平台。其持久化层本质上是一个“按照分布式事务日志架构的大规模发布/订阅消息队列”，这使它作为企业级基础设施来处理流式数据非常有价值。

基本术语

1. **Producer**: 消息生产者。一般情况下，一条消息会被发送到特定的主题上。通常情况下，写入的消息会通过轮询将消息写入各分区。生产者也可以通过设定消息 key 值将消息写入指定分区。写入分区的数据越均匀 Kafka 的性能才能更好发挥。
2. **Topic**: Topic 是个抽象的虚拟概念，一个集群可以有多个 Topic，作为一类消息的标识。一个生产者将消息发送到 topic，消费者通过订阅 Topic 获取分区消息。
3. **Partition**: Partition 是个物理概念，一个 Topic 对应一个或多个 Partition。新消息会以追加的方式写入分区里，在同一个 Partition 里消息是有序的。Kafka 通过分区，实现消息的冗余和伸缩性，以及支持物理上的并发读、写，大大提高了吞吐量。
4. **Replicas**: 一个 Partition 有多个 Replicas 副本。这些副本保存在 broker，每个 broker 存储着成百上千个不同主题和分区的副本，存储的内容分为两种：master 副本，每个 Partition 都有一个 master 副本，所有内容的写入和消费都会经过 master 副本；follower 副本不处理任何客户端的请求，只同步 master 的内容进行复制。如果 master 发生了异常，很快会有一个 follower 成为新的 master。
5. **Consumer**: 消息读取者。消费者订阅主题，并按照一定顺序读取消息。Kafka 保证每个分区只能被一个消费者使用。
6. **Offset**: 偏移量是一种元数据，是不断递增的整数。在消息写入时 Kafka 会把它添加到消息里。在分区内的偏移量是唯一的。消费过程中，会将最后读取的偏移量存储在 Kafka 中，消费者关闭偏移量不会丢失，重启会继续从上次位置开始消费。
7. **Broker**: 独立的 Kafka 服务器。一个 Topic 有 N 个 Partition，一个集群有 N 个 Broker，那么每个 Broker 都会存储一个这个 Topic 的 Partition。如果某 topic 有 N 个 partition，集群有(N+M)个 broker，那么其中有 N 个 broker 存储该 topic 的一个 partition，剩下的 M 个 broker 不存储该 topic 的 partition 数据。如果某 topic 有 N 个 partition，集群中 broker 数目少于 N 个，那么一个 broker 存储该 topic 的一个或多个 partition。在实际生产环境中，尽量避免这种情况的发生，这种情况容易导致 Kafka 集群数据不均衡。

系统框架



工作流程

1. 第一个 topic 有两个生产者，新消息被写入到 partition 1 或者 partition 3，两个分区在 broker1、broker2 都有备份。
2. 有新消息写入后，两个 follower 分区会从两个 master 分区同步变更。
3. 对应的 consumer 会从两个 master 分区根据现在 offset 获取消息，并更新 offset。
4. 第二个 topic 只有一个生产者，同样对应两个 partition，分散在 Kafka 集群的两个 broker 上。
5. 有新消息写入，两个 follower 分区会同步 master 变更。
6. 两个 Consumer 分别从不同的 master 分区获取消息。

优点

1. **高吞吐量、低延迟**: kafka 每秒可以处理几十万条消息，它的延迟最低只有几毫秒；
2. **可扩展性**: kafka 集群支持热扩展；
3. **持久性、可靠性**: 消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；
4. **容错性**: 允许集群中节点故障，一个数据多个副本，少数机器宕机，不会丢失数据；
5. **高并发**: 支持数千个客户端同时读写。

缺点

1. **分区有序**: 仅在同一分区保证有序，无法实现全局有序；
2. **无延时消息**: 消费顺序是按照写入时的顺序，不支持延时消息；
3. **重复消费**: 消费系统宕机、重启导致 offset 未提交；
4. **Rebalance**: Rebalance 的过程中 consumer group 下的所有消费者实例都会停止工作，等待 Rebalance 过程完成。

注意kafka本身不支持延迟消息，消费顺序按照写入时的顺序，从偏移量位置开始消费。

6.如果要让 Kafka 支持延迟消息你会怎么做？你有几种方案？各有什么优缺点？

Apache Kafka 本身并不直接支持延迟消息队列的功能，但我们可以通过一些策略来实现类似的效果。以下是一些可能的方案：

1. 使用定时任务

在生产者端，将消息和预定的发送时间一同存储在数据库或其他存储系统中。然后，使用定时任务（比如 Quartz 或者 Spring Scheduler）定期扫描数据库，将达到预定发送时间的消息发送到 Kafka。

优点：实现简单，不需要修改 Kafka 或消费者的代码。

缺点：可能会产生大量的数据库操作，对数据库性能有一定影响。并且，消息的发送时间可能不够精确，比如如果定时任务每分钟运行一次，那么消息的发送延迟可能会达到一分钟。

1. 在消费者端实现延迟

在消费者端，首先立即消费消息，然后检查消息的预定处理时间。如果预定处理时间还未到，那么将消息存储起来，并在预定处理时间到达时再处理消息。

优点：不需要修改 Kafka 或生产者的代码。

缺点：增加了消费者的复杂性。并且，如果消费者重启，那么可能会丢失那些还未处理的消息。

使用场景

1. **日志收集**: 大量的日志消息先写入 kafka，数据服务通过消费 kafka 消息将数据落地；
2. **消息系统**: 解耦生产者和消费者、缓存消息等；
3. **用户活动跟踪**: kafka 经常被用来记录 web 用户或者 app 用户的各种活动，如浏览网页、搜索、点击等活动，这些活动信息被各个服务器发布到 kafka 的 topic 中，然后消费者通过订阅这些 topic 来做实时的监控分析，亦可保存到数据库；
4. **运营指标**: 记录运营、监控数据，包括收集各种分布式应用的数据，生产各种操作的集中反馈，比如报警和报告；
5. **流式处理**: 比如 spark streaming。

其它的消息队列中间件

RabbitMQ

优点

- 基于 AMQP 协议：除了 Qpid，RabbitMQ 是唯一一个实现了 AMQP 标准的消息服务器；
- 健壮、稳定、易用；
- 社区活跃，文档完善；
- 支持定时消息；

- 可插入的身份验证，授权，支持 TLS 和 LDAP；
- 支持根据消息标识查询消息，也支持根据消息内容查询消息。

缺点

- erlang 开发源码难懂，不利于做二次开发和维护；
- 接口和协议复杂，学习和维护成本较高。

总结

- erlang 有并发优势，性能较好。虽然源码复杂，但是社区活跃度高，可以解决开发中遇到的问题；
- 业务流量不大的话可以选择功能比较完备的 RabbitMQ。

Pulsar

优点

- 灵活扩容
- 无缝故障恢复
- 支持延时消息
- 内置的复制功能，用于跨地域复制如灾备
- 支持两种消费模型：流（独享模式）、队列（共享模式）

RocketMQ

优点

支持发布/订阅（Pub/Sub）和点对点（P2P）消息模型：

- 顺序队列：在一个队列中可靠的先进先出（FIFO）和严格的顺序传递；支持拉（pull）和推（push）两种消息模式；
- 单一队列百万消息的堆积能力；
- 支持多种消息协议，如 JMS、MQTT 等；
- 分布式横向扩展架构；
- 满足至少一次消息传递语义；
- 提供丰富的 Dashboard，包含配置、指标和监控等；
- 支持的客户端，目前是 java、c++ 及 golang

缺点

- 社区活跃度一般；
- 延时消息：开源版不支持任意时间精度，仅支持特定的 level。

使用场景

- 为金融互联网领域而生，对于可靠性要求很高的场景。

什么是Git

Git是一个分布式版本控制系统，跟踪目录里的修改。它的工作流是非线性的（不同电脑上的平行分支形成了一个graph）。和主从式的系统不一样的是，每台电脑上的每个git目录都是一个完整的repo，包含全部历史和完整的版本跟踪能力。

关于工作流是非线性的，这意味着它支持多种并行开发路径，并能够处理复杂的分支和合并操作。建立分支、rebase、修订commit、强制推送、cherry-pick、分支复位，在git都是很正常的使用方式。

关于合并和rebase：这两种操作都是用来整合来自不同分支的改动。合并操作会保留分支的历史，而rebase则是将一系列的提交“移植”到另一个分支的新基点上，这有助于创建一个更线性的历史。

什么不是Git

GitHub/GitLab

只是提供Git服务的社区和网站，Git本身并不需要网络。

Fork

Pull Request/Merge Request

Code Review和合并的流程。

分支策略

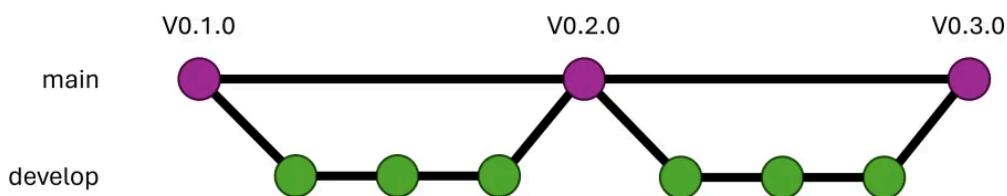
Git的工作流是基于分支的。不但每个repo是平等的，每个分支也是。Master/main、develop这些只是为了简化管理而人工指定的有特殊含义的分支。这里的分支策略是为了更好地协作而产生的习惯规范，不是git的工作流本身必须定义的。分支可以分为几个层次。

main

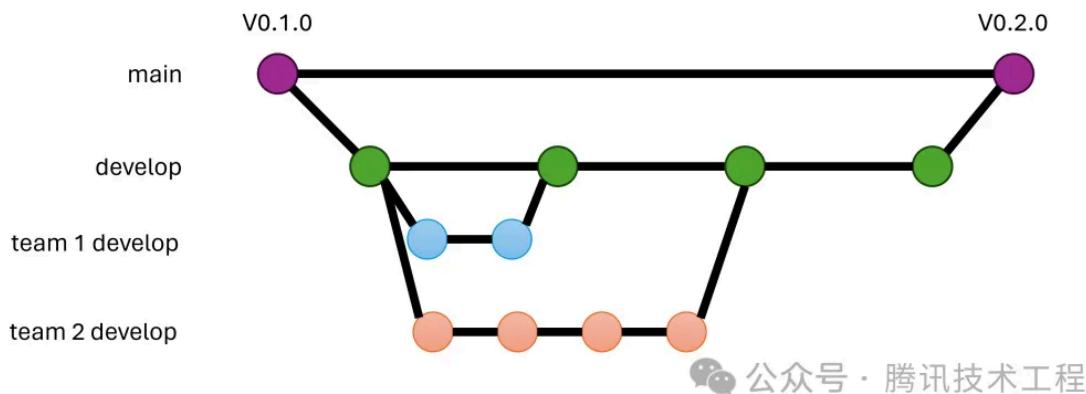
整个项目的稳定分支，里面的内容可能相对较老，但是这个分支里的内容都是经过测试和验证的。原先都叫master，因为政治正确的要求，最近越来越多新项目开始用main。有些快速开发的项目甚至不采用main分支。

Develop

开发主要发生在develop分支。新特性先放到这个分支，再去优化和增强稳定性。



对于跨团队的大项目，每个团队都有自己的兴趣点和发布周期。很常见的做法是，每个团队有自己的develop分支。每过一段时间合并到总的develop分支。一般来说，中等大小的团队，专注于repo的某一部分，可以采取这样的分支形式。小团队或者个人没有必要有自己的develop分支。那样反而会浪费时间和增加合并过程中的风险。



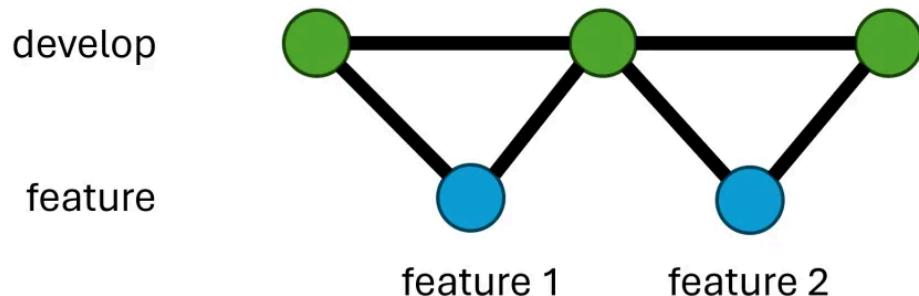
公众号 · 腾讯技术工程

Feature

Feature分支是生命期很短的分支，专注于单个特性的开发。

- 从develop分支上新建一个feature分支
- 提交一些关于这个feature的代码
- 合并回去
- 删掉这个feature分支

对于本地repo里的feature分支，你可以做任何事。常见的用法是在开发过程中非常频繁地提交，走一小步就提交一次。在发出MR之前，先合并成一个commit，把这个分支变整洁，方便后续操作。



公众号 · 腾讯技术工程

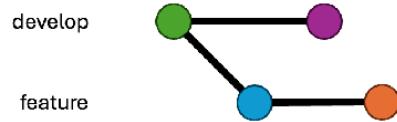
Release分支群

就好像一个目录，包含了不同版本给不同产品线的release分支。

Merge还是Rebase

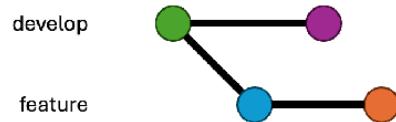
Merge

把一个分支合并到目标分支，在顶上建立一个commit用来合并，两个分支里已有的commit不会有变化。



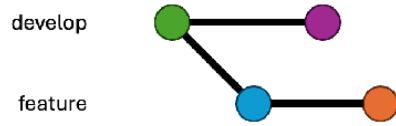
Rebase

从分支分出来的地方切开，嫁接到目标分支的顶端上。



squash

选择一个分支里的一些commit，压扁成一个commit。



对于Feature分支，最好是使用Rebase和Squash：

历史简洁性：通过使用 squash 将一个 feature 分支的多个 commits 压缩成一个单一的 commit，可以使主分支的历史更加清晰和易于管理。这样做的结果是，每个 feature 都对应一个单独的 commit，使得历史线性化，便于理解和回溯。

避免无意义的合并 commit：使用 rebase 而不是 merge 来将 feature 分支整合到 develop 或 main 分支可以避免产生那些“合并 commit”，这些 commit 在功能上通常没有实际内容，只是表明了一个合并行为。

减少冲突解决的复杂性：在合并之前进行 rebase，可以先解决与基底分支的冲突，这使得合并操作更为简单，因为 rebase 会将 feature 分支的起点移动到目标分支的最新提交上。