

八股文

一、计算机网络

1.1 HTTP、HTTPS、DNS协议

1.1.1 TCP、UDP属于什么层

1. 为什么要分层?

分层设计：不同层实现不同功能 **类比** 函数设计：函数体不能太长



2. OSI七层作用



3. OSI七层与TCP/IP四层对应关系以及所用协议



4. 每一层的数据结构



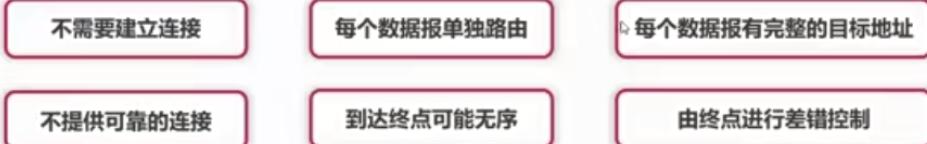
5. 网络层特点

网络层

- ◆ OSI模型：第三层、TCP/IP模型：第二层

- ◆ 关键协议：IP协议、ICMP协议

网络层属于主机之间的通信，它的目的是向上提供简单灵活的、无连接的、尽最大努力交付的数据报服务，网络层不提供服务质量的承诺。



6. 传输层特点

传输层

- ◆ OSI模型：第四层、TCP/IP模型：第三层

- ◆ 关键协议：TCP协议、UDP协议

传输层属于主机间不同进程的通信，传输层向上面的应用层提供通信服务，并屏蔽了下面的核心网络细节，使得面向传输层编程就像是两个主机进程之间有一条端到端的逻辑通信信道一样；当传输层采用TCP协议时，这条逻辑通信信道就是一条可靠的通信信道，而尽管下面的网络是不可靠的。

7. 应用层特点

应用层

- ◆ 关键协议：HTTP协议、FTP协议、SMTP协议、DNS等等
- ◆ 定义了运行在不同端系统上的应用程序进程如何相互传递报文

网络层：提供主机之间的通信

传输层：提供主机不同进程之间的通信

应用层：提供不同应用之间的通信

实际上，应用层定义了进程交换的报文类型、报文的语法、字段的含义、进程如何发生数据、怎么样发送数据等等。

面试题

- 1.TCP、UDP、IP协议分别属于什么层
- 2.网络中传输层有什么作用？它有哪些协议

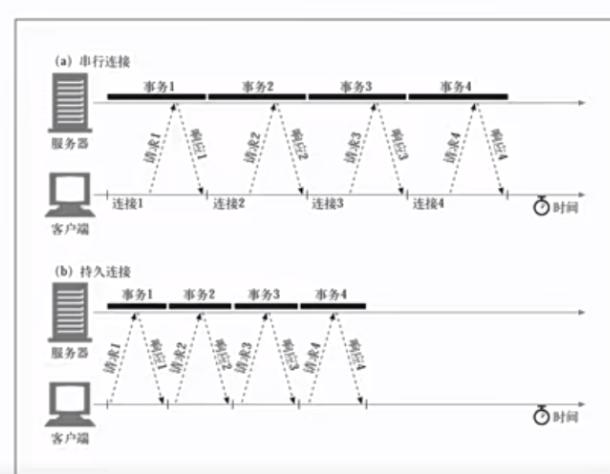
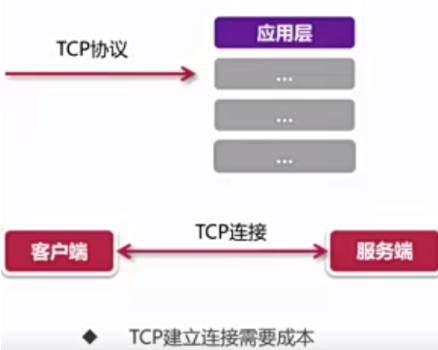
1.1.2 HTTP1.0、1.1、2.0有什么区别

1. 各版本一些区别



2. keep-alive长连接的好处

keep-alive长连接

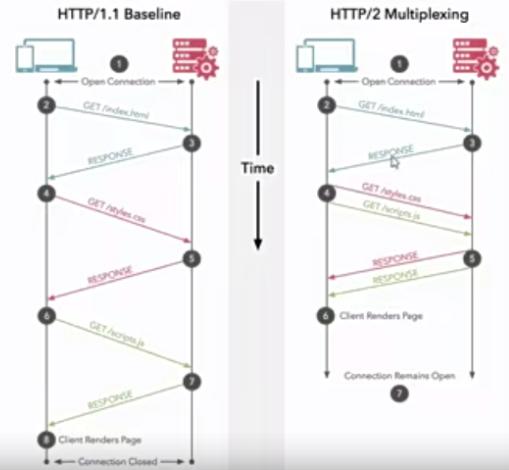


3. HTTP2.0

多路复用

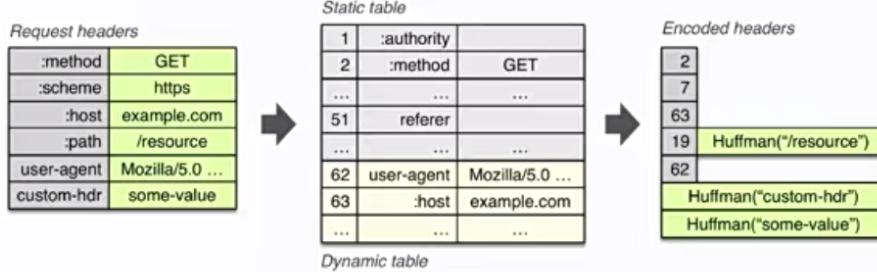
多路复用通常表示在一个信道上传输多路信号或数据流的过程和技术。通过使用多路复用，通信运营商可以避免维护多条线路，从而有效地节约运营成本

- ◆ 二进制分帧是基础，通信单位为帧
- ◆ 多请求并行不依赖多TCP连接
- ◆ 并行在一个TCP连接交互多种类型信息



头部压缩通过哈夫曼编码和字典实现

头部压缩 HPACK header compression



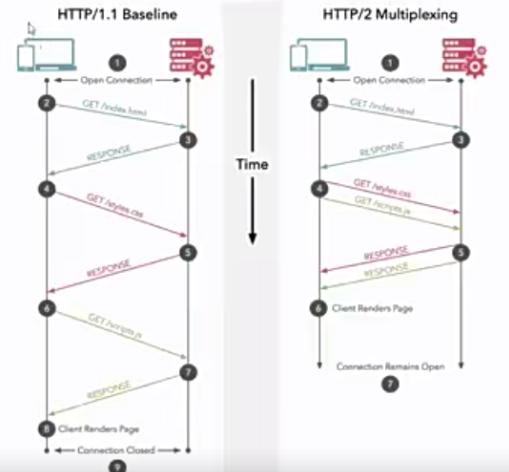
- Literal values are (optionally) encoded with a static Huffman code
- Previously sent values are (optionally) indexed
 - e.g. "2" in above example expands to "method: GET"

服务端推送

- ◆ GET /index.html HTTP/1.1
- ◆ GET /style.css HTTP/1.1
- ◆ GET /scripts.js HTTP/1.1

服务端“未卜先知”

服务端主动推送资源



1. 简述HTTP1.0, 1.1, 2.0的主要区别
2. HTTP头Connection:Keep-alive是什么意思？解决了什么问题？

1.1.3 HTTP状态码

1. HTTP报文结构

HTTP报文结构



比如一个场景

GET <https://www.baidu.com> HTTP/1.1

Accept-Encoding:gzip

Accept-Language:zh-CN

{

 "page":1,

 "pageSize":5

}

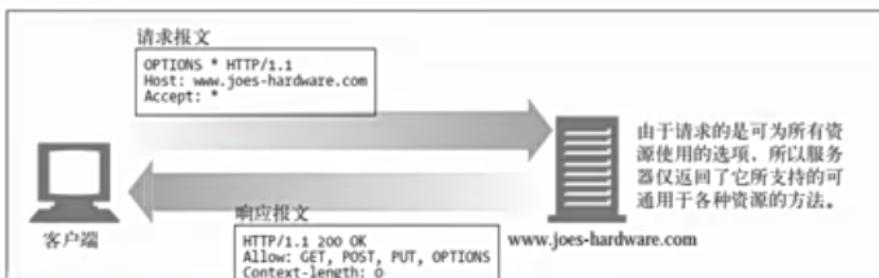
2. HTTP请求方法

HTTP请求方法

请求方法：HTTP请求的本质是对服务器资源进行操作的过程（增删改查+系统功能），

通过定义不同方法实现不同操作是清晰并且是必要的。

- ◆ GET: 最常用的方法，常用于请求服务器发送某个资源
- ◆ HEAD: 和GET类似，但服务器在响应中只返回首部
- ◆ POST: 向服务器写入数据
- ◆ TRACE: 观察请求报文到达服务器的最终样子
- ◆ PUT: 和GET相反，向服务器写入资源
- ◆ DELETE: 请求服务器删除请求 URL 所指定的资源
- ◆ OPTIONS: ...



3. 幂等操作：指对同一资源的多次操作，其结果与执行一次操作的效果相同

幂等函数：指对于相同的输入值，多次调用函数的结果与单次调用的结果相同

4. HTTP状态码

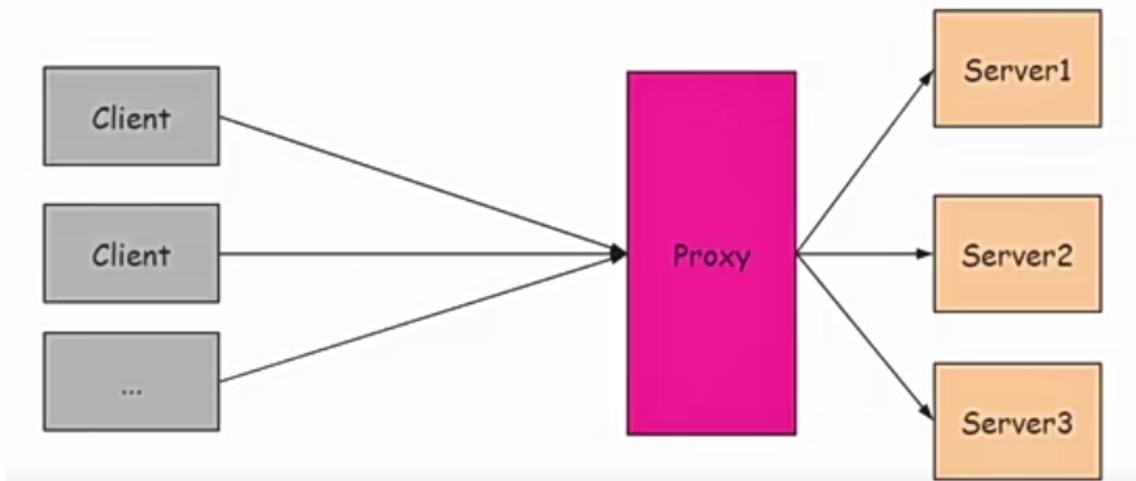
HTTP状态码

状态码	含义
200~299	成功状态码
300~399	重定向状态码
400~499	客户端错误状态码
500~599	服务端错误状态码

5. 一些常见的状态码

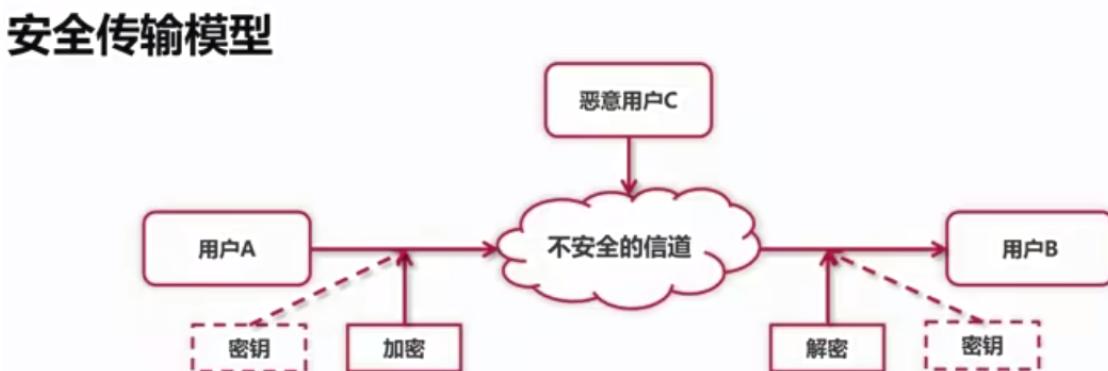
- ◆ 200: OK 请求没问题，实体的主体部分包含了所请求的资源
- ◆ 204: No Content 响应报文中包含若干首部和一个状态行，但没有实体的主体部分
- ◆ 304: Not Modified 所请求的资源未修改，服务器返回此状态码时，不会返回任何资源
- ◆ 400: Bad Request 客户端请求的语法错误，服务器无法理解
- ◆ 401: Unauthorized 请求客户端在获取对资源的访问权之前，对自己进行认证
- ◆ 403: Forbidden 请求被服务器拒绝了
- ◆ 404: Not Found 用于说明服务器无法找到所请求的 URL
- ◆ 500: Internal Server Error 服务器内部错误，无法完成请求
- ◆ 502: Bad Gateway 作为网关或者代理工作的服务器尝试执行请求时，从远程服务器接收到了一个无效的响应
- ◆ 503: Service Unavailable 用来说明服务器现在无法为该请求提供服务
- ◆ 504: Gateway Timeout 网关或代理的服务器，未及时从远端服务器获取请求

6. 网关分配资源



1.1.4 对称加密和非对称加密

1. 一种安全传输模型



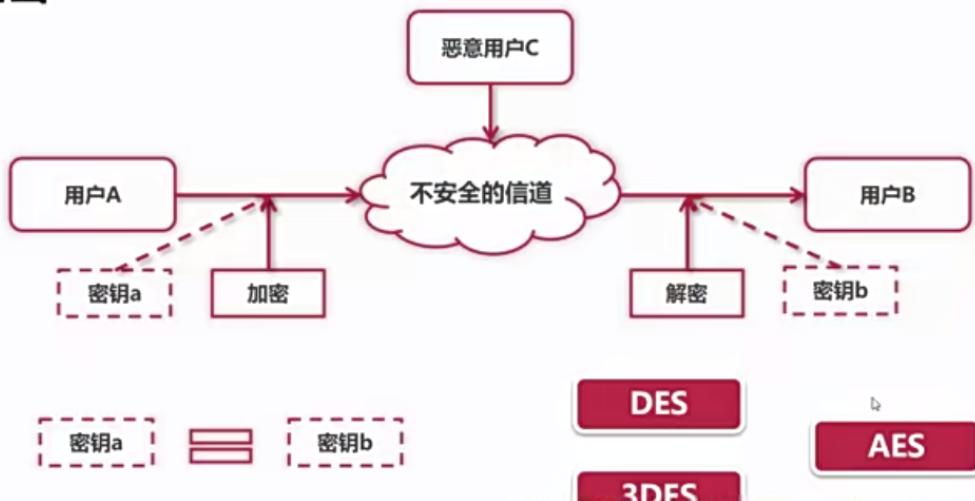
密钥它是在明文转换为密文或将密文转换为明文的算法中输入的参数。

密钥分为对称密钥与非对称密钥。

2. 古典密码学是通过一一映射的，但是可以通过频率统计出来

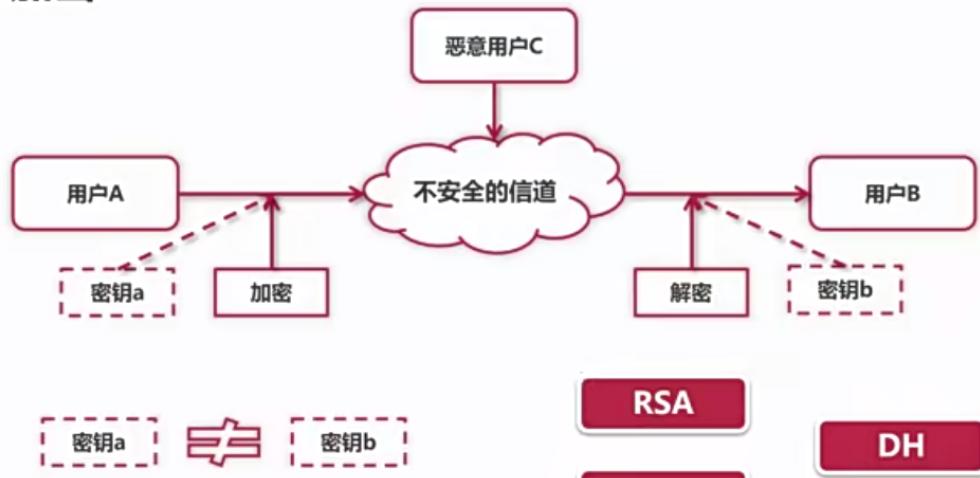
3. 对称加密

对称加密

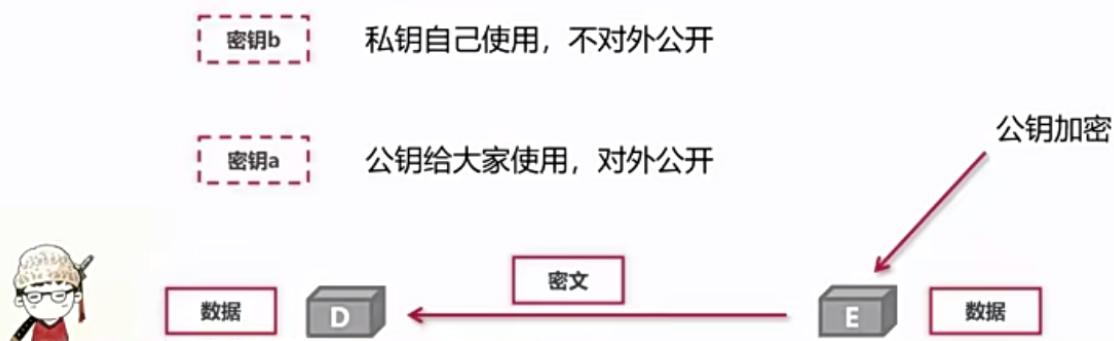


4. 非对称加密

非对称加密



◆ 密钥a、密钥b是具有一定数学关系的一组秘钥



5. 对比

	对称加密	非对称加密
密钥	一串密钥	一组密钥（公钥、私钥）
效率	效率高	效率低
安全性	较高	更高
管理成本	高	低

6. 哈希散列不是加密算法，因为哈希是单向的，不具备解密能力。即使通过哈希函数将用户密码保存进数据库了，也有风险被黑客破解(时间暴力枚举等等)，这时可以在哈希之前加盐，加盐就是在进行哈希散列的时候自定义一个字符串，将该字符串和原密码拼接，最后将这个整体一起哈希散列，加大破译难度。

1.1.5 HTTPS加密认证--TLS技术

1. HTTP与HTTPS对比

HTTP VS HTTPS

- ◆ HTTPS(Secure)是安全的HTTP协议

- ◆ http(s)://<主机>:<端口>/<路径>

	HTTP	HTTPS
安全性	不安全	安全
复杂度	低	高
效率	高	低
端口	一般是80	443

2. TLS

TLS

- ◆ TLS: 传输层安全性协议 (Transport Layer Security)

- ◆ 数据安全和数据完整
- ◆ 对传输层数据进行加密后传输



综合了对称加密、非对称加密技术设计的安全协议

3. 数字证书，类似身份证件，使用非对称加密

数字证书

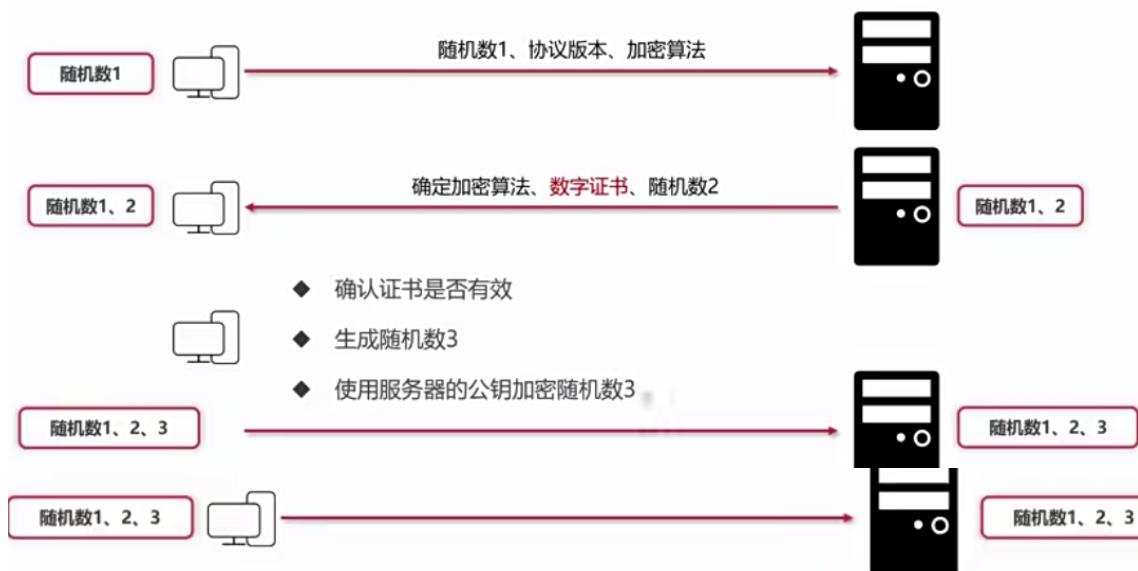
数字证书是指在互联网通讯中标志通讯各方身份信息的一个数字认证，人们可以在网上用它来识别对方的身份。

数字证书是**可信任组织**颁发给**特定对象**的认证。

证书格式、版本号
证书序列号
签名算法
有效期
对象名称
对象公开密钥
...

使用非对称加密算法来生成对称密钥

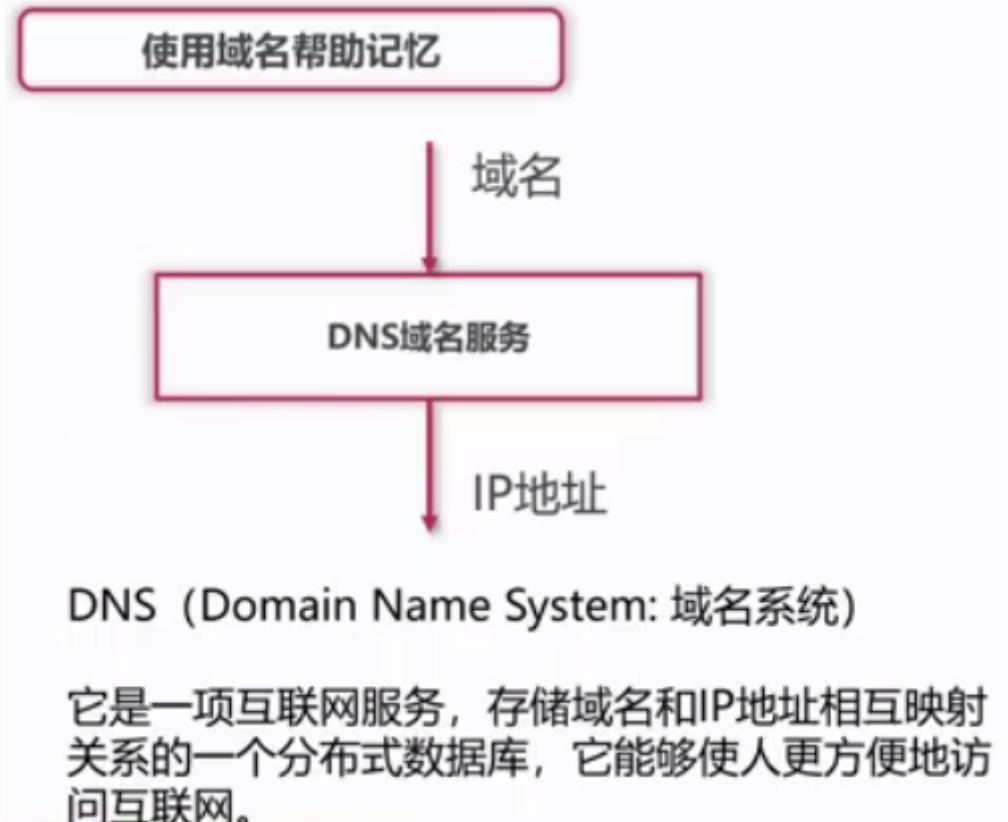
非对称加密



双方分别生成密钥，没有经过传输

1.1.6 域名系统DNS

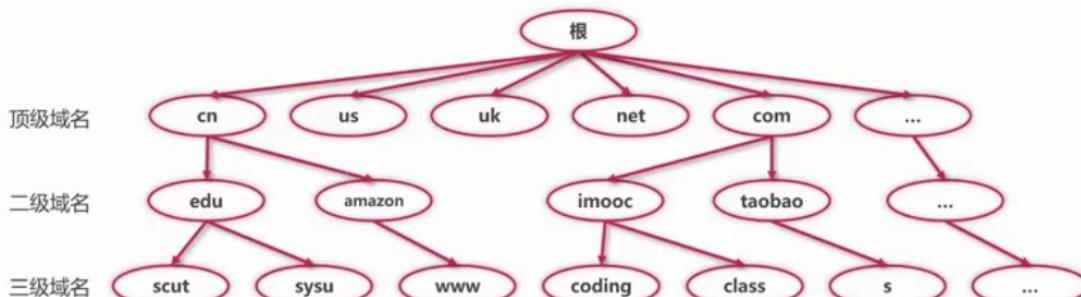
1. 进程服务：IP+端口，比如115.182.41.180:443，但是这样太难记住了，那么



2. 域名工作原理

1. 域名由点、字母、数字组成
 2. 点分隔不同域
 3. 域名分为顶级域、二级域、三级域
 4. 比如 www.baidu.com，www是三级域、baidu是二级域、com是顶级域
3. 那么这些域就组成了一棵树

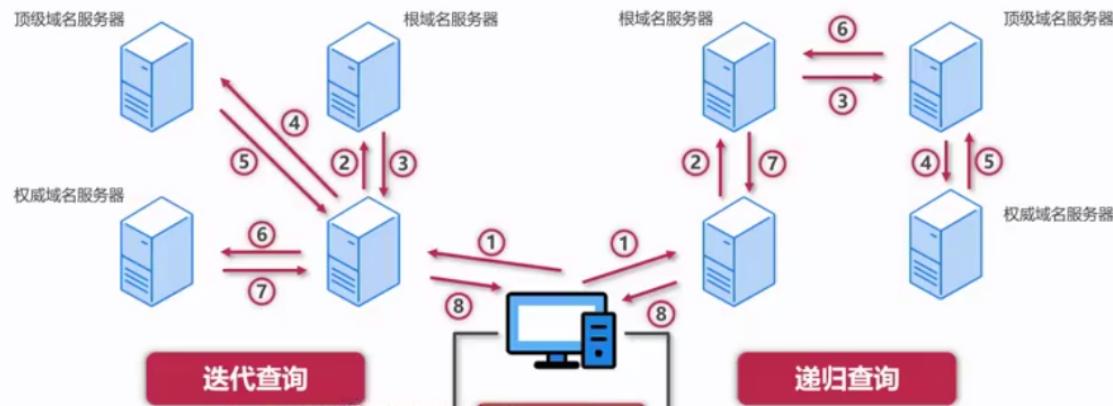
DNS工作原理



树形结构

4. DNS会优先在本机hosts文件里查找是否有映射，然后通过迭代或者递归查询

DNS工作原理

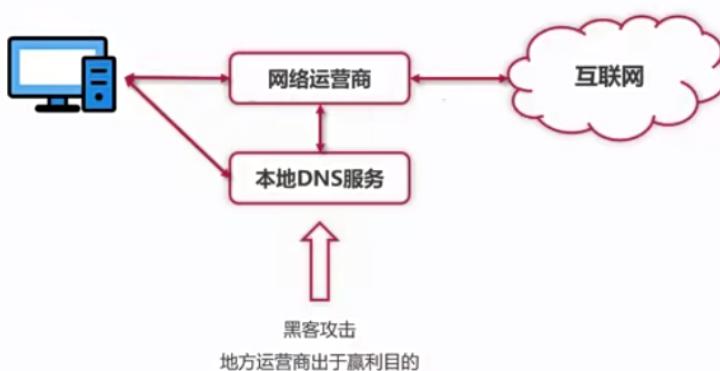


1.1.7 DNS攻击

1. 比如我们请求的url是www.baidu.com，但是经过DNS解析后返回的是www.twly.com，将正常站点解析到恶意页面(赚取游戏推广费等等)

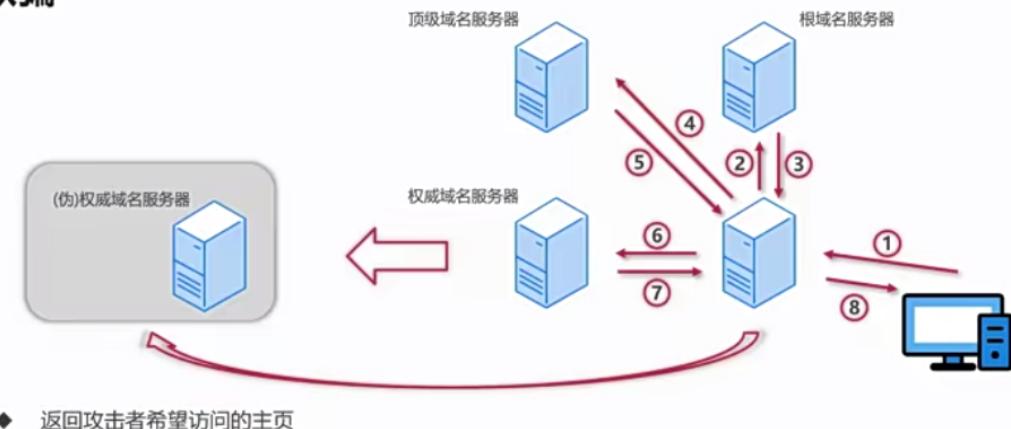
2. DNS劫持

DNS劫持



3. DNS欺骗

DNS欺骗



◆ 返回攻击者希望访问的主页

4. DDoS攻击是大量请求DNS解析服务器，使得网络或者系统资源耗尽，让服务中断或者停止，这样其他用户就无法正常访问了，防范手段：

1. DNS服务商角度：提高本机安全些
2. 个人角度：选择更加权威的服务商

1.2 IP、TCP、UDP协议

1.2.1 TCP、UDP的区别

1. UDP协议

UDP协议

16位源端口号	16位目的端口号
16位UDP长度	16位UDP校验和

2. TCP协议

TCP协议

固定20字节	16位源端口	16位目的端口
	序号	
	确认号	
	数据偏移	保留字段
	控制位	窗口
	校验和	紧急指针
	TCP选项 (可选)	填充

TCP协议-序号

16位源端口	16位目的端口
序号	
确认号	
数据偏移	保留字段
控制位	窗口
校验和	紧急指针
TCP选项 (可选)	填充

- ◆ 4个字节 [0, 4294967295]

◆ TCP数据是字节流——每个字节都有唯一的序号

◆ 起始序号在建立TCP连接的时候设置

◆ 序号表示本报文段数据的第一个字节的序号

TCP协议-确认号

16位源端口	16位目的端口
序号	
确认号	
数据偏移	保留字段
控制位	窗口
校验和	紧急指针
TCP选项 (可选)	填充

- ◆ 和序号一致——四个字节

◆ 期待收到对方下一个报文的第一个数据字节序号

◆ 若确认号=N，则序号N-1为止的所有数据都已经正确收到

TCP协议-控制位

- ◆ 6个比特位

U	A	P	R	S	F
R	C	S	S	Y	I
G	K	H	T	N	N

控制位	含义
URG	Urgent: 紧急位, URG=1, 表示紧急数据
ACK	Acknowledgement: 确认位, ACK=1, 确认号才生效
PSH	Push: 推送位, PSH=1, 尽快地把数据交付给应用层
RST	Reset: 重置位, RST=1, 重新建立连接
SYN	Synchronization: 同步位, SYN=1 表示连接请求报文
FIN	Finish: 终止位, FIN=1 表示释放连接

TCP协议-窗口

- ◆ 2字节——[0, 65535]

- ◆ 窗口指明允许对方发送的数据量

- ◆ 数据缓冲空间有限, 不能无限缓存数据

3. UDP vs TCP

16位源端口	16位目的端口
序号	
确认号	
数据偏移	控制位
保留字段	窗口
校验和	紧急指针
TCP选项 (可选)	填充

UDP vs TCP

- ◆ TCP提供的是可靠的**有连接**服务

- 建立连接
- 通过连接进行通信
- 释放连接

- ◆ UDP提供的是不可靠的**无连接**服务

- 可靠传输: 无差错、不丢失、不重复
- 按**序**到达: 数据有序



网络层: 提供主机之间的通信

UDP协议只管发送, 什么都不保证

- TCP——“打电话”
- UDP——“写信”

UDP vs TCP

	UDP	TCP
性能	性能负载低	性能负载高
速度	速度快	速度慢
实现难度	实现简单	实现复杂
应用场景	简单场景	复杂场景
面向连接	无连接服务	有连接服务
可靠性	不可靠服务	可靠服务

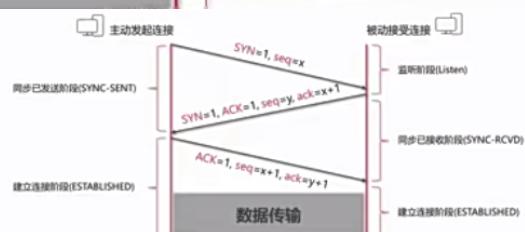
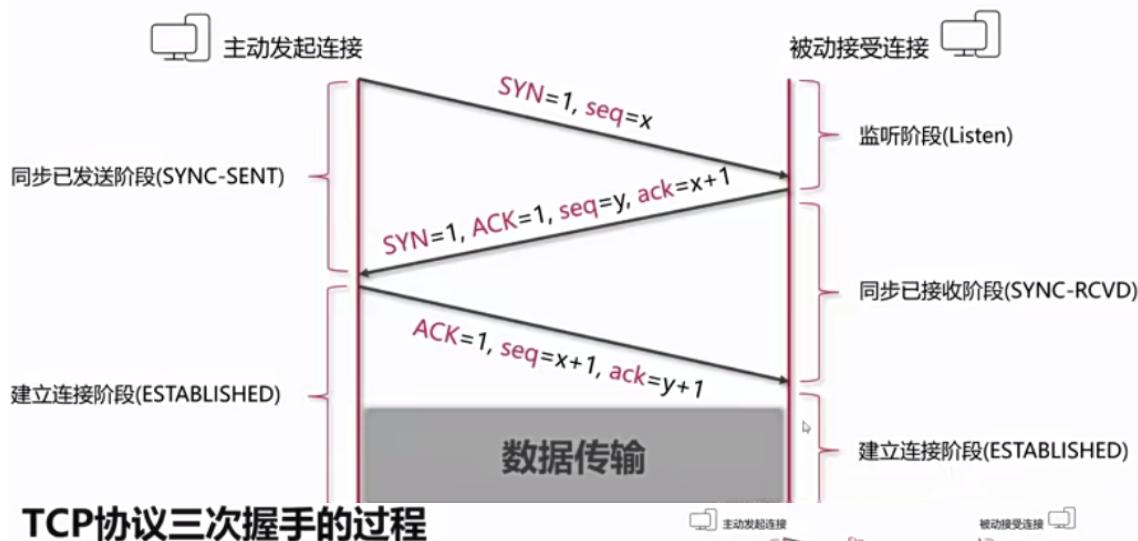
UDP vs TCP

应用	应用层协议	传输层协议
名字转换	DNS(域名系统)	UDP
文件传送	TFTP(简单文件传送协议)	UDP
流式多媒体通信	-	UDP
IP地址配置	DHCP(动态主机配置协议)	UDP
电子邮件	SMTP(简单邮件传送协议)	TCP
文件传送	FTP(文件传送协议)	TCP
远端终端接入	TELNET(远程终端协议)	TCP
WWW	HTTP(超文本传输协议)	TCP

DNS通常处理大量的短期请求，而且这些请求的响应需要在短时间内迅速返回。UDP是无连接的，没有复杂的握手和确认过程，因此在速度和效率方面比TCP更合适。DNS服务器通常面对的是大量的短暂请求，而非大量长连接。UDP在这种情境下更为适用，因为它允许更高的并发连接数量。

1.2.2 TCP三次握手

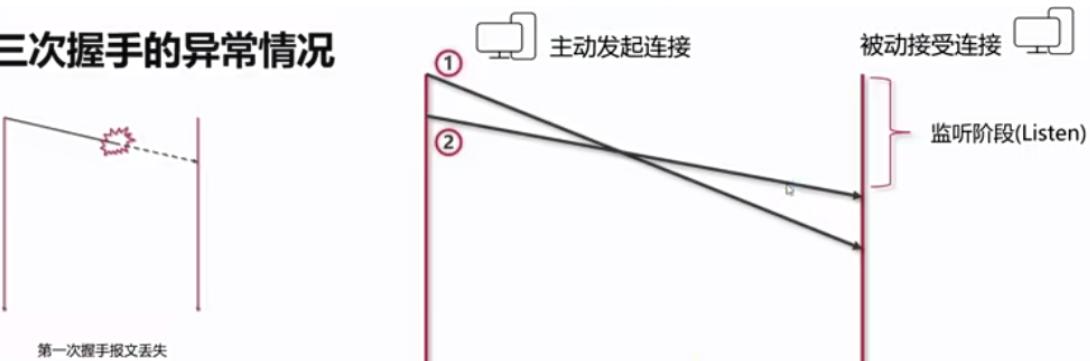
1. 三次握手图解



- ◆ 第一次：SYN=1请求同步并告诉对方自己的数据序列号
- ◆ 第二次：SYN=1、ACK=1，确认对方的数据并告诉对方自己的数据序列号
- ◆ 第三次：ACK=1，确认了对方的数据并开始传输数据

2. 异常情况

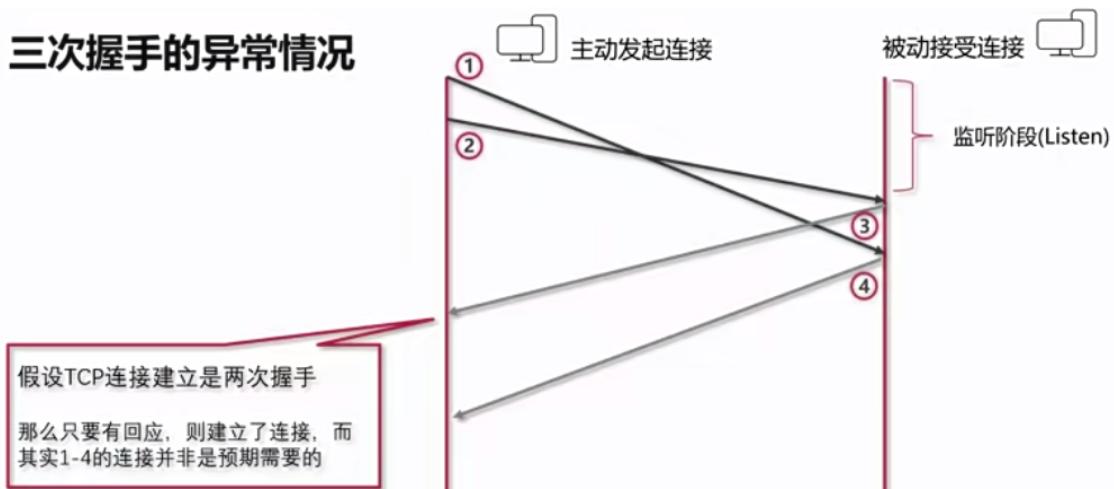
三次握手的异常情况



主动方认为第一次握手报文丢失

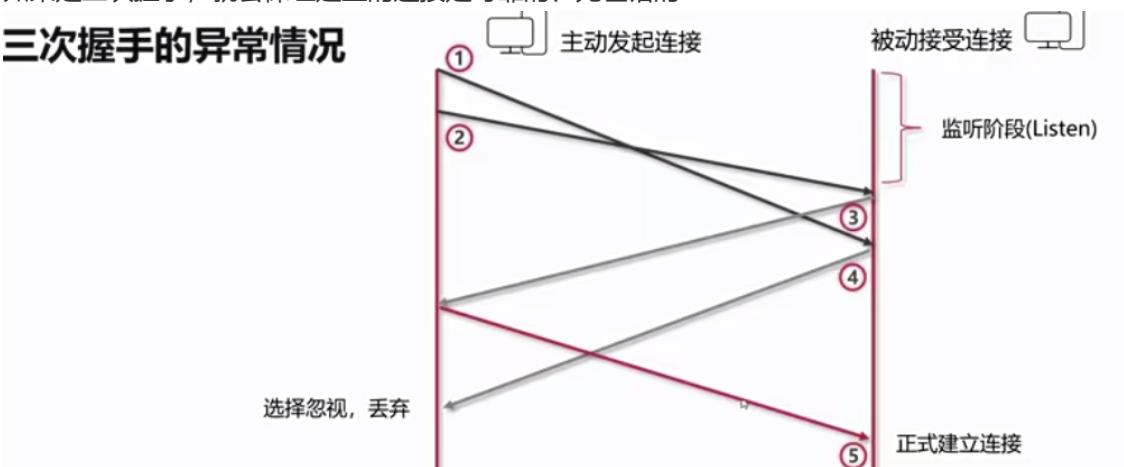
如果是两次握手，那么如果因为滞慢等原因没有及时传达，那就建立起两次连接，造成资源浪费

三次握手的异常情况



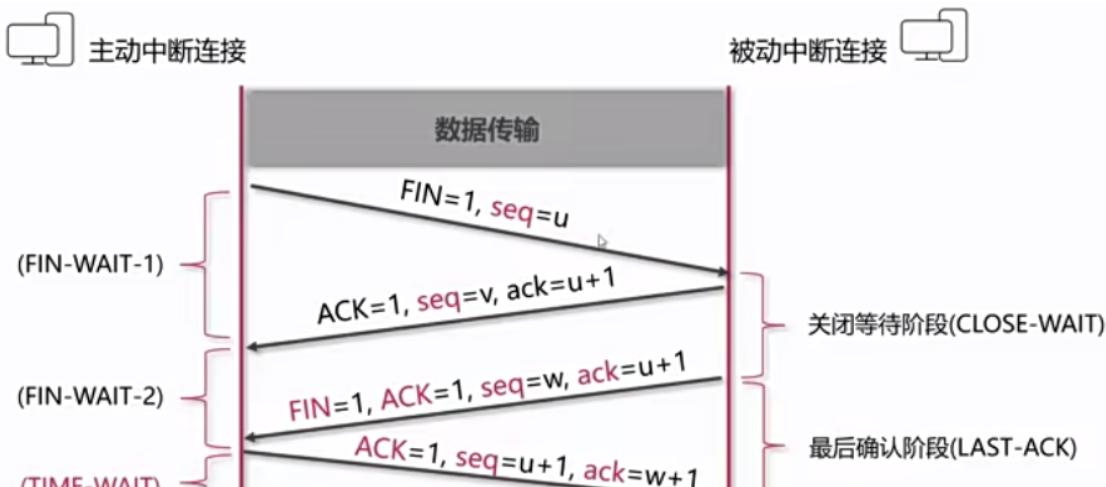
如果是三次握手，就会保证建立的连接是可靠的、无差错的

三次握手的异常情况



1.2.3 TCP四次挥手

1. TCP的释放

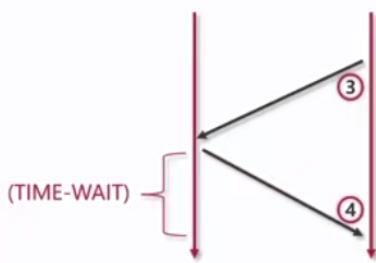


TCP连接释放四次挥手的过程

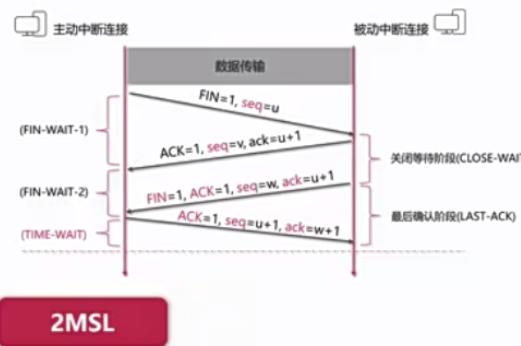
- ◆ 第一次: $FIN=1$, 主动请求中断连接
- ◆ 第二次: $ACK=1$ 、 $ack=u+1$, 表示确认收到中断报文
- ◆ 第三次: $FIN=1$ 、 $ACK=1$ 、 $seq=w$ 、 $ack=u+1$, 请求中断连接
- ◆ 第四次: $ACK=1$ 、 $seq=u+1$ 、 $ack=w+1$, 确认中断连接

2. TIME-WAIT

TIME-WAIT状态



TIME-WAIT状态指的是第四次挥手后，主动中断连接方所处的状态，这个状态下，主动方尚未完全关闭TCP连接，端口不可复用。

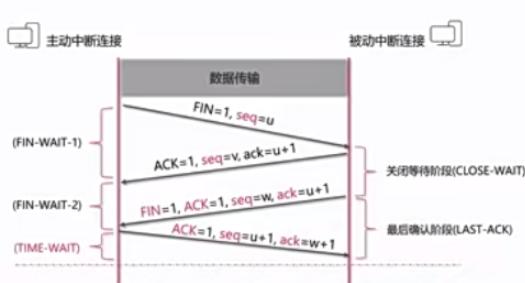


MSL(Max Segment Lifetime): 最长报文段寿命
RFC 793标准建议设置为2分钟

为什么TIME-WAIT状态需要等待2MSL?

1. 最后一个报文没有确认
2. 确认最后一个ACK报文一定能到达对方
3. 2MSL时间内，如果没有到达对方，那么对方会重新进行第三次挥手，确保连接正常释放

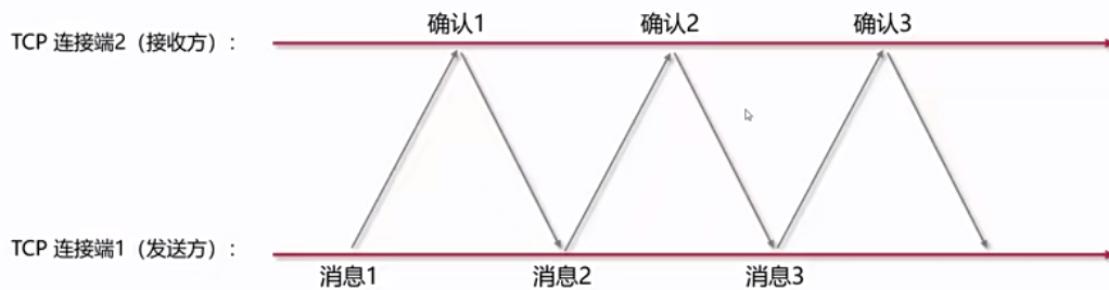
1. 确保当前连接所有的报文都已经过期



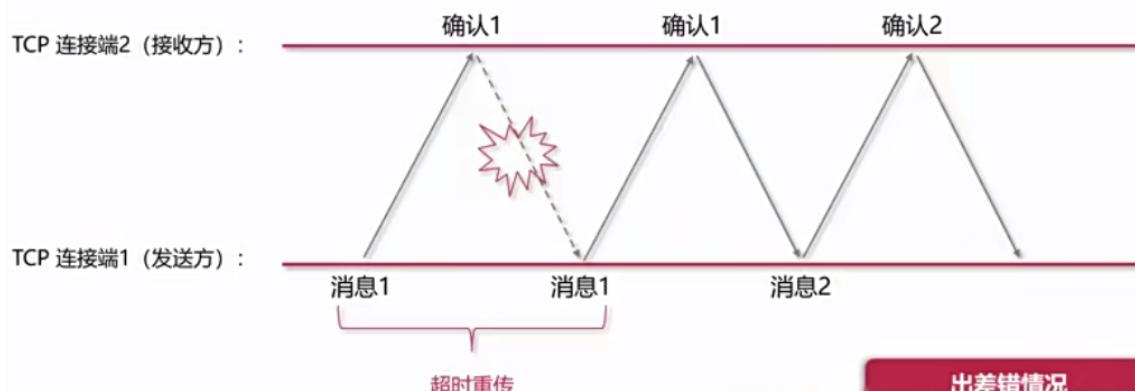
3. TCP可靠运输-滑动窗口

4. 停止-等待协议

停止-等待协议



停止-等待协议



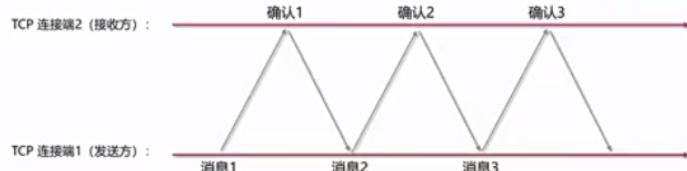
5. ARQ协议

连续ARQ(Automatic Repeat reQuest)协议

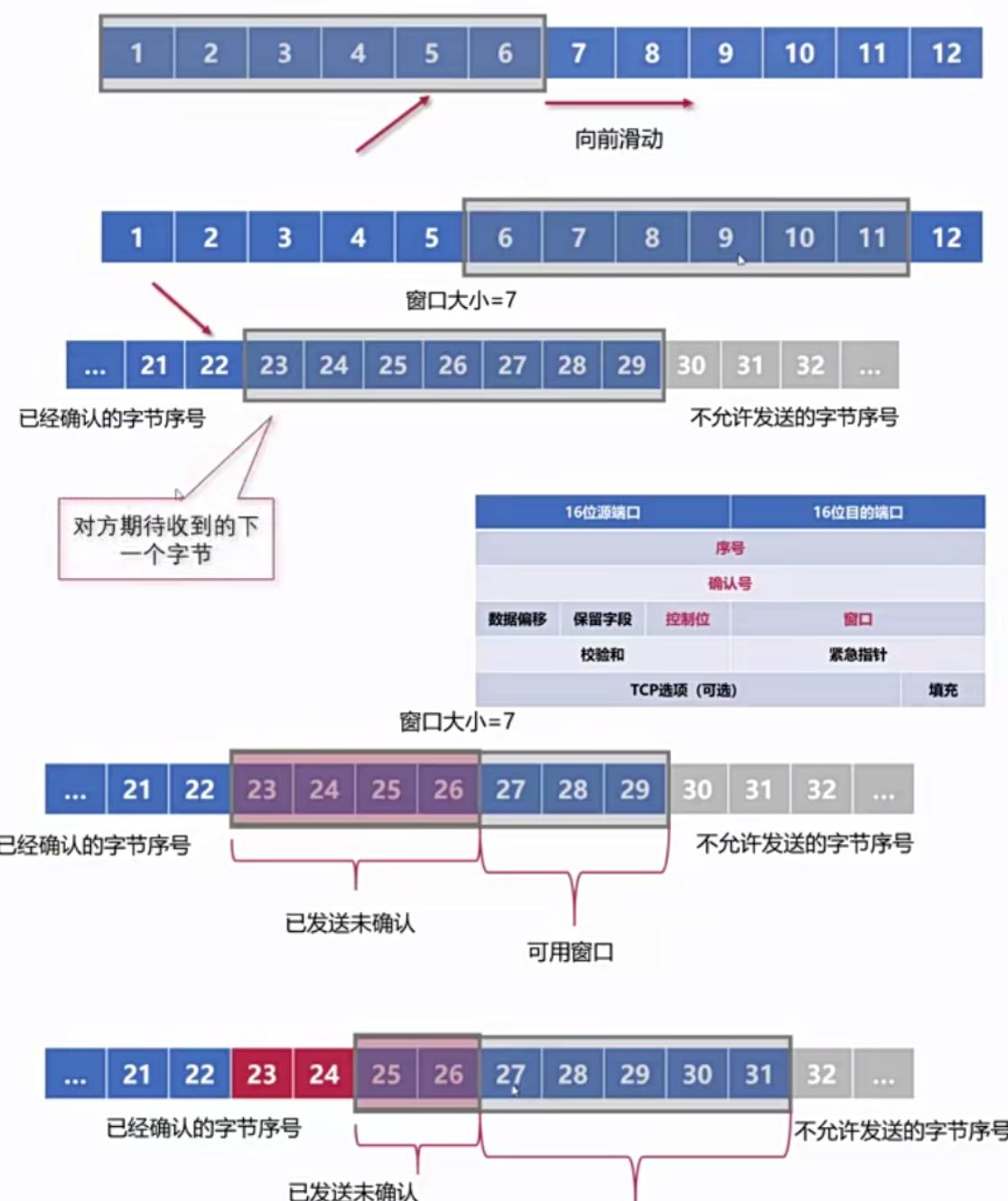
◆ 停止-等待协议是最简单的可靠传输协议

既然单个发送、确认的效率低，是否可以批量发送和确认呢？

◆ 停止-等待协议对信道的利用效率不高



连续ARQ(Automatic Repeat reQuest)协议



6. 滑动窗口使得信息传递不再是串联，提高了速度

滑动窗口

1. 窗口指明允许对方发送的数据量
2. TCP协议是传输数据流的协议，通过TCP协议头部序列号、确认号以及窗口等字段的控制，可以在有限的缓冲资源下，接收几乎无限的数据



1.2.4 TCP拥塞避免算法

1. 定义：

在某段时间内，若对网络中的某一资源（带宽、缓存、处理机等）的需求超过了该资源所能提供的可用部分，网络性能就会变坏，这种情况称为网络拥塞。



拥塞是很多原因引发的，是一个全局问题

2. 慢开始与拥塞避免

慢开始与拥塞避免

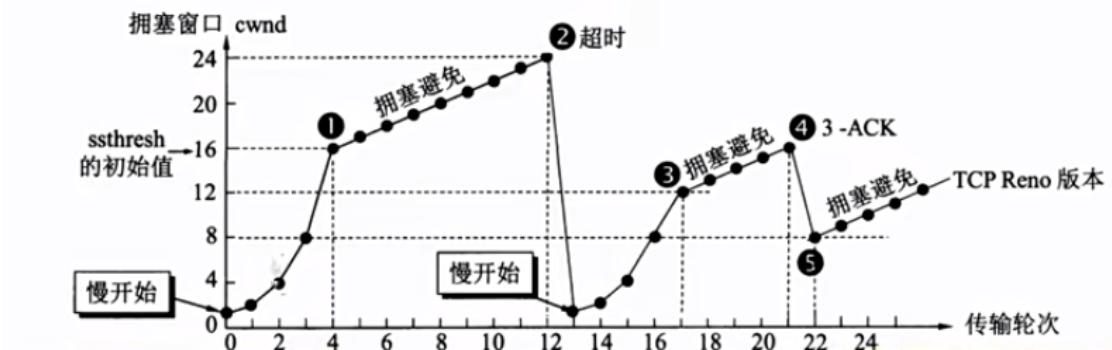
拥塞窗口(cwnd): 拥塞窗口是TCP协议基于窗口的拥塞控制需要的一个变量配置。发送方在发送数据时会维持一个叫拥塞窗口cwnd(congestion window)的状态变量，并且可以动态变化，在TCP报文头部，发送方让自己的发送窗口等于拥塞窗口。

门限值(ssthresh): 拥塞避免算法启动阈值，当拥塞窗口cwnd超过门限值ssthresh时，启动慢启动算法。

传输轮次(Route-Trip): 一次报文发送和确认的时间称为一次传输轮次，RTT(Route-Trip Time)定义的是一次传输轮次的往返时间。

慢开始与拥塞避免

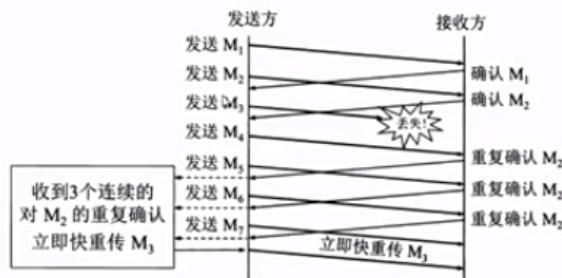
$$\text{门限值(ssthresh)} = \text{拥塞窗口(cwnd)} / 2$$



3. 快重传与快恢复

快重传与快恢复

快重传：让发送方尽早知道个别报文段的丢失，并立即重传，以避免发送方认为网络发生了拥塞，从而因为拥塞避免算法降低发送数据。

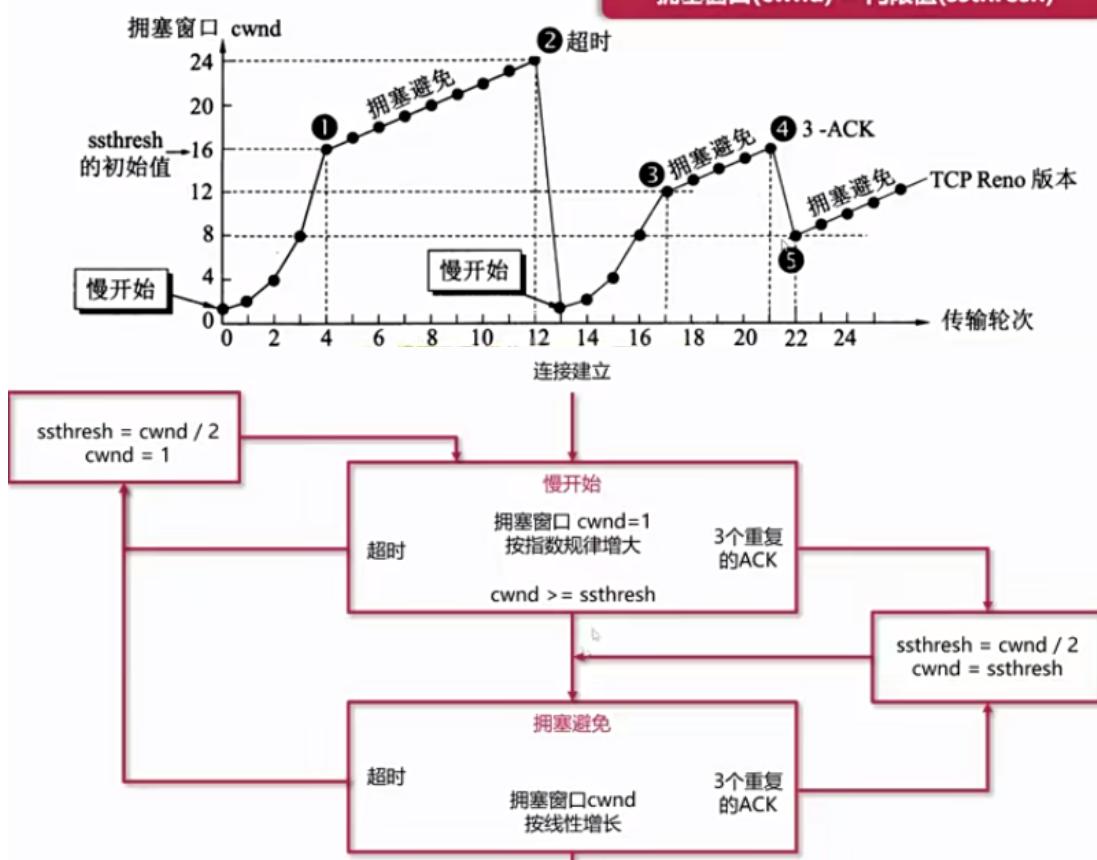


- ◆ 不是捎带确认，而是立即确认
- ◆ 对丢失报文多次重复确认

快重传与快恢复

$$\text{门限值(ssthresh)} = \text{拥塞窗口(cwnd)} / 2$$

$$\text{拥塞窗口(cwnd)} = \text{门限值(ssthresh)}$$



1.2.5 TCP粘包原理

1. 粘包并不是TCP协议造成的，而是应用层协议设计缺陷造成的
2. 应用层数据拆分
 1. 基于长度的标识
 2. 基于特殊分隔符

3. Nagle算法

Nagle 算法

Nagle算法是一种通过减少数据包的方式提高 TCP 传输性能的算法。

因为网络带宽有限，它不会将小的数据块直接发送到目的主机，而是会在本地缓冲区中等待更多待发送的数据，这种批量发送数据的策略虽然会影响实时性和网络延迟，但是能够降低网络拥堵的可能性并减少额外开销。

Telnet协议

Header(40字节) +数据(1字节)

带宽利用率： $1/(40+1) \sim 2.44\%$

- ◆ 缓冲区中数据超过最大数据段 (MSS)

- ▲ 等待一个确认往返时间 (ACK) =

4. 总结

总结

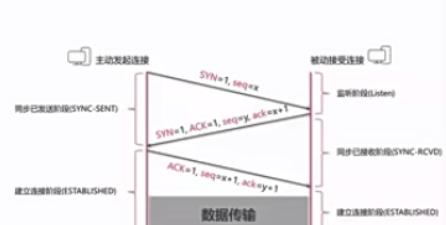
- ◆ TCP 协议数据传输的核心机制 — 基于字节流，不存在消息、数据包的概念
- ◆ TCP粘包是应用层开发者错误设计实现导致的
- ◆ 应用层协议需要自行设计消息边界，以正确分离消息，避免消息粘连
- ◆ 分离消息的两个方法：基于长度的拆分、基于特殊字符的拆分
- ◆ Nagle算法时通过合并字节流数据、减少数据包来提升TCP协议传输能力的协议

5. TCP协议安全性

1. SYN flood攻击，攻击方即是主动发送方

SYN flood攻击原理

- ◆ 利用三次握手的过程漏洞
- ◆ 大量发送第一次握手（假IP）的报文
- ◆ 攻击方忽略第二次握手的报文
- ◆ 被攻击方多个TCP连接处于(SYNC-RCVD)阶段，耗费大量资源
- ◆ 最终因为资源耗尽，拒绝服务(DoS)



2. 资源耗尽类攻击

资源耗尽类攻击

- ◆ 攻击方控制大量 “肉鸡”
- ◆ 向攻击目标发送大量连接握手报文
- ◆ 完成三次握手后保持连接，但不做任何事情，消耗TCP连接资源
- ◆ 或者马上断开连接又重新发起新连接
- ◆ 服务器不堪重负，最终影响正常服务或直接宕机

3. 协议特性漏洞攻击

协议特性漏洞攻击

- ◆ 攻击方控制大量“肉鸡”
- ◆ 攻击方与攻击目标建立正常连接
- ◆ 依据TCP流量控制的特性，马上将TCP窗口设置为0
- ◆ 攻击方断开连接，此时攻击目标还傻傻等待窗口重新打开
- ◆ 服务器不堪重负，最终影响正常服务或直接宕机

4. 防范手段



1.2.6 虚拟专用网VPN

1. 场景

1. 公司内网
2. 校园网
3. 工业专用网

2. 专用IP地址

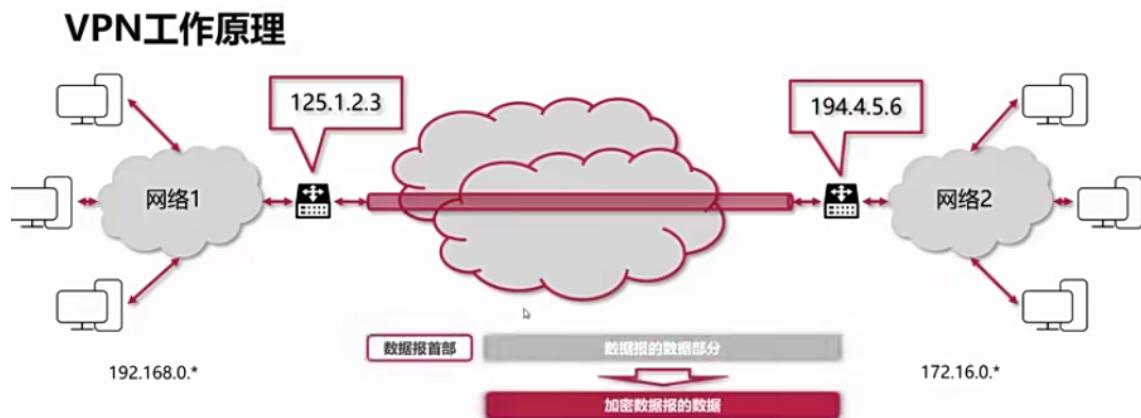
专用IP地址

专用IP地址是指一些只能提供给机构内部通信的IP地址；这类IP地址不能接入到互联网上与其他网络主机进行通信。

即专用IP地址只能用作本地地址而不能用作全球地址，因为专用IP地址在全球并不是唯一的，互联网上的路由器对目的地址为专用IP地址的数据报一律不进行转发。

- ◆ 10.0.0.0 ~ 10.255.255.255
- ◆ 172.16.0.0 ~ 172.31.255.255
- ◆ 192.168.0.0 ~ 192.168.255.255

3. 工作原理



二、操作系统

2.1 进程、线程和协程

2.1.1 进程

1. 演进历史

操作系统的演进历史



2. 多道程序设计

1. 在计算机内存中同时存放多个程序
2. 在计算机的管理程序之下相互穿插运行
3. 用户无需面向硬件接口编程
4. IO设备管理软件，提供读写接口
5. 文件管理软件，提供操作文件接口
6. 操作系统实现了对计算机硬件资源的管理和抽象

3. 怎么隔离资源？调度程序？提高利用率？

为什么需要进程

- ◆ 进程是系统进行资源分配和调度的基本单位
 - ◆ 进程作为程序独立运行的载体保障程序正常执行
 - ◆ 进程的存在使得操作系统资源的利用率大幅提升



2.1.2 同步与异步

1. 五状态模型：创建、就绪、终止、阻塞、运行

- ◆ 当进程被分配到除CPU以外所有必要的资源后
- ◆ 只要再获得CPU的使用权，就可以立即运行
- ◆ 其他资源都准备好、只差CPU资源的状态为**就绪状态**

就绪状态

- ◆ 进程获得CPU，其程序正在执行称为执行状态
- ◆ 在单处理机中，在某个时刻只能有一个进程是处于执行状态

执行状态

- ◆ 进程因某种原因如：其他设备未就绪而无法继续执行
- ◆ 从而放弃CPU的状态称为阻塞状态

阻塞状态



关系图如上，创建状态如下

分配PCB

插入就绪队列

- ◆ 创建进程时拥有PCB但其他资源尚未就绪的状态称为创建状态

操作系统提供fork函数接口创建进程

创建状态

系统清理

PCB归还

- ◆ 进程结束由系统清理或者归还PCB的状态称为终止状态

终止状态

终止状态如上，完整关系图如下

创建

就绪

终止

阻塞

运行

创建完成

IO完成

进程调度

IO请求

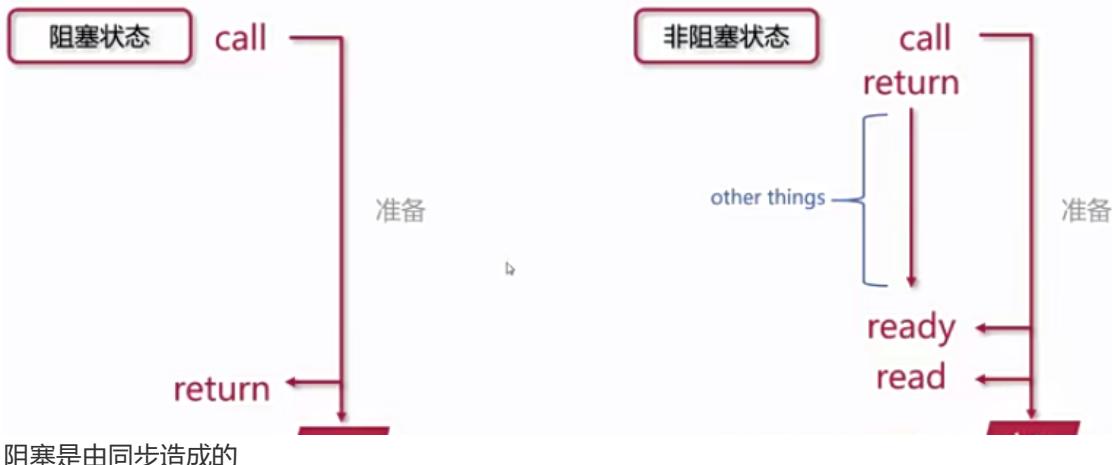
时间片完

执行完成

进程五状态模型

2. 阻塞、非阻塞、同步、异步

阻塞、非阻塞、同步、异步



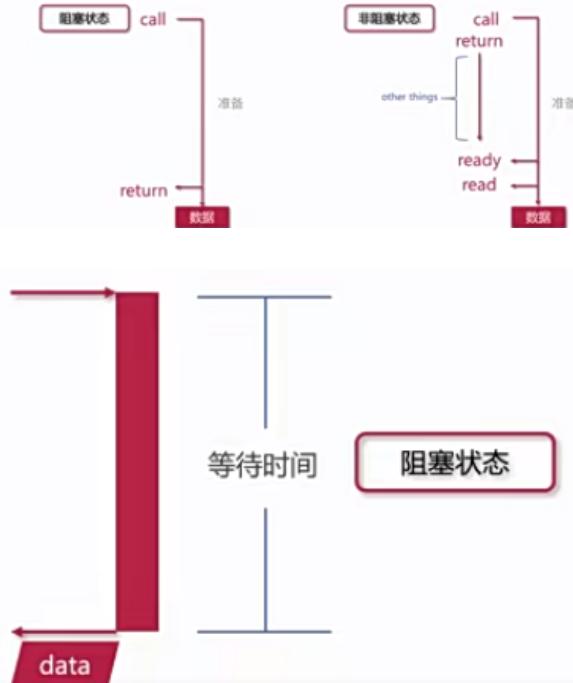
◆ 同步和异步强调的是消息通信机制

◆ 阻塞和非阻塞强调的是程序在等待调用结果时的状态

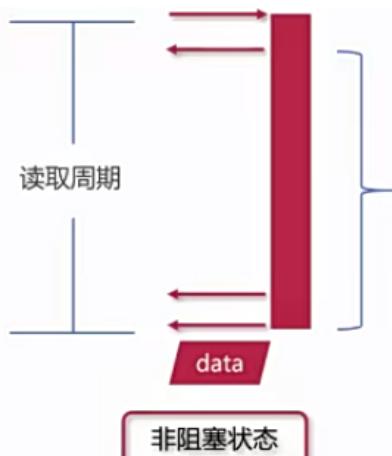
一些常见的阻塞情况

◆ 读写磁盘

◆ 读写网络



非阻塞的情况



非阻塞就是通顺，是多路并行，是高效异步

2.1.3 进程和线程的区别

1. 线程是什么

线程是什么？

- ◆ 20世纪60年代提出进程的概念
- ◆ 比进程更小的独立运行的基本单位——线程(Threads)
- ◆ 20世纪90年代以后，多处理机系统迅速发展

提高系统内程序并发执行的程度

线程是什么？

进程是系统进行资源分配
和调度的基本单位

- ◆ 线程是操作系统进行运行调度的最小单位
- ◆ 包含在进程之中，是进程中实际运行工作的单位
- ◆ 一个进程可以并发多个线程，每个线程执行不同的任务

进程控制块(PCB)



线程控制块(TCB)

线程共享进程资源

进程与线程的关系

◆ 进程 (Process)

◆ 线程 (Thread)

进程的线程共享进程资源

线程(Thread)

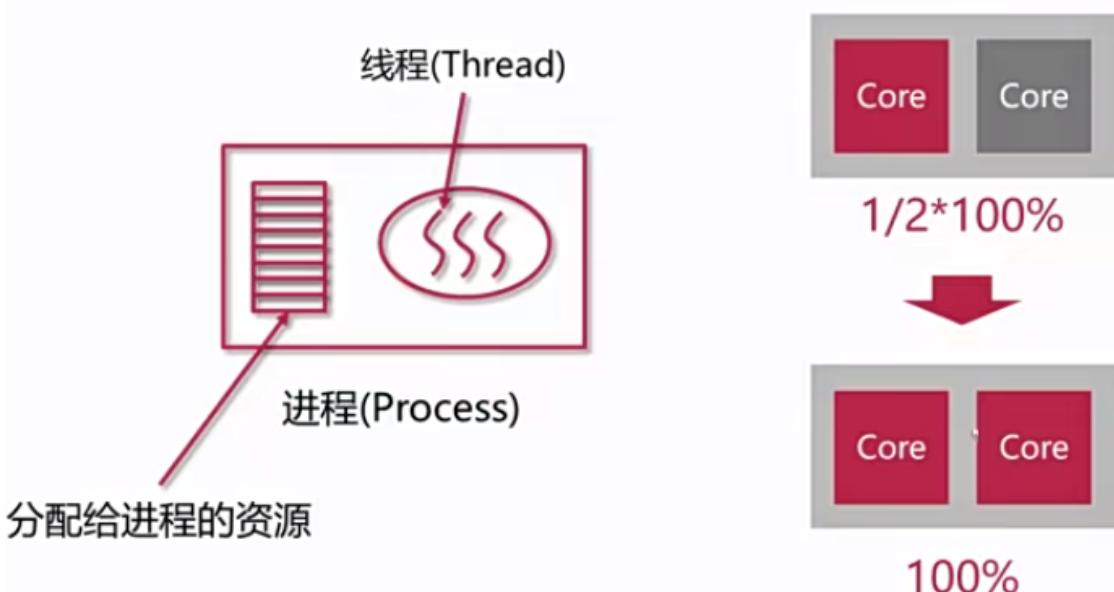


分配给进程的资源

二者对比

	进程	线程
资源	资源分配的基本单位	不拥有资源
调度	调度开销大	调度开销小
系统开销	进程系统开销大	线程系统开销小
并发性	进程间并发	进程间并发 进程内多线程并发

多线程就可以充分利用CPU的资源



2.1.4 用户态和内核态

1. 操作系统资源管理

1. 处理器资源
2. IO设备资源
3. 存储器资源
4. 文件资源

2. Linux设计

1. 对不同操作赋予不同的执行等级
2. 与系统相关的特别操作必须由最高权限完成

3. 内核态

内核态

- ◆ 内核空间：存放的是内核代码和数据
- ◆ 进程执行操作系统内核的代码
- ◆ CPU可以访问内存所有数据，包括外围设备

4. 用户态

用户态

- ◆ 用户空间：存放的是用户程序的代码和数据
- ◆ 进程在执行用户自己的代码（非系统调用之类的函数）
- ◆ CPU只可以访问有限的内存，不允许访问外设

5. 二者切换的时机

内核态/用户态切换



2.1.5 IO密集服务

1. 计算密集型

计算密集型

- ◆ 也称为CPU密集型(CPU bound)
- ◆ 完成一项任务的时间取决于CPU的速度
- ◆ CPU利用率高、其他事情处理慢

2. IO密集型

IO密集型

- ◆ 频繁读写网络、磁盘等任务都属于IO密集型任务
- ◆ 完成一项任务的时间取决于IO设备的速度
- ◆ CPU利用率低、大部分时间在等待设备完成

往磁盘写入1G文件?

3. 对于计算密集型，应该使用多线程去充分利用CPU资源。对于IO密集，提升多线程效率已经没什么用了，但是可以提高硬盘读写速度

4. 服务部署

服务部署

计算密集型

- ◆ 对CPU要求高
- ◆ 对内存要求高
- ◆ 对磁盘要求一般

16核 + 128G + SATA/SSD

56核 + 128G + SATA

72核 + 512G + SSD

IO密集型

- ◆ 对磁盘读写速度要求高
- ◆ 对内存要求高
- ◆ 对CPU要求不高

混合密集型

- ◆ 对CPU要求高
- ◆ 对内存要求高
- ◆ 对磁盘要求高

2.1.6 协程

1. 上下文切换

上下文切换

Context

分时系统

Linux 是一个多任务操作系统，它支持远大于 CPU 数量的任务同时运行。

当然，这些任务实际上并不是真的在同时运行，而是因为系统在很短的时间内，将 CPU 轮流分配给它们，造成多任务同时运行的错觉。

而在每个任务运行前，CPU 都需要知道任务从哪里加载、又从哪里开始运行。

进程控制块
标识符
状态
优先级
程序计数器
内存指针
上下文数据
IO状态信息
记账信息
...

- ◆ 寄存器级上下文
- ◆ 用户级上下文
- ◆ 系统级上下文

上下文切换



上下文切换势必有资源开销，那么进程与进程间的切换肯定是最大的

上下文切换的成本



2. 协程

协程(Coroutine)

- ◆ 比线程更小的粒度

- ◆ 运行效率更高

- ◆ 可以支持更高的并发

协程的本质是用户级线程，一般的线程是内核级线程



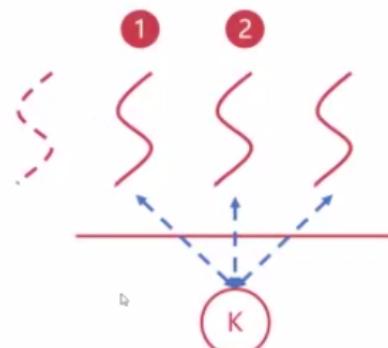
3. 为什么协程叫做协作式线程？因为协程之间是相互让步、相互协作的。由用户自行调度，内核无法干涉。线程对协程是一对多

4. 协程优缺点

协程的优缺点

- ◆ 调度、切换、管理更加轻量

- ◆ 内核无法感知协程的存在

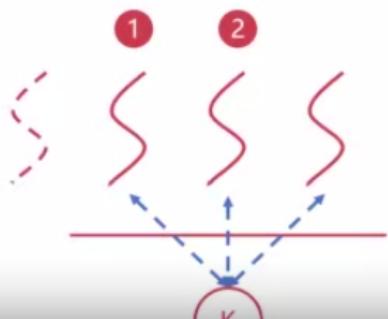


协程的优缺点

- ◆ 可以减少上下文切换的成本

- ◆ 无法发挥CPU的多核优势

- ◆ 协程主要运用在多IO的场景



2.2 存储系统详解

2.2.1 缓存

1. 层次结构

存储器的层次结构

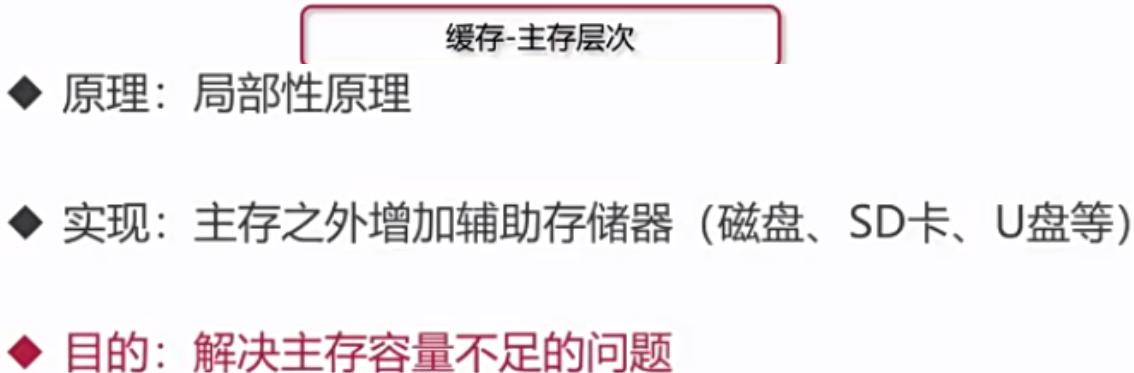


2. 局部性原理

局部性原理是指CPU访问存储器时，无论是存取指令还是存取数据，所访问的存储单元都趋于聚集在一个较小的连续区域中。



- ◆ 原理：局部性原理
- ◆ 实现：在CPU与主存之间增加一层速度快（容量小）的Cache
- ◆ 目的：解决主存速度不足的问题



主存-辅存层次

缓存的设计

- ◆ 原理：局部性原理
- ◆ 分离冷热数据，降低热点数据服务的负载
- ◆ 提升吞吐量、并发量，提升服务质量

2.2.2 虚拟内存

1. 概述

虚拟内存概述

- ◆ 有些进程实际需要的内存很大，超过物理内存的容量
- ◆ 多道程序设计，使得每个进程可用物理内存更加稀缺
- ◆ 不可能无限增加物理内存，物理内存总有不够的时候

虚拟内存概述

- ◆ 虚拟内存是操作系统内存管理的关键技术
- ◆ 使得多道程序运行和大程序运行成为现实
- ◆ 把程序使用内存划分，将部分暂时不使用的内存放置在辅存

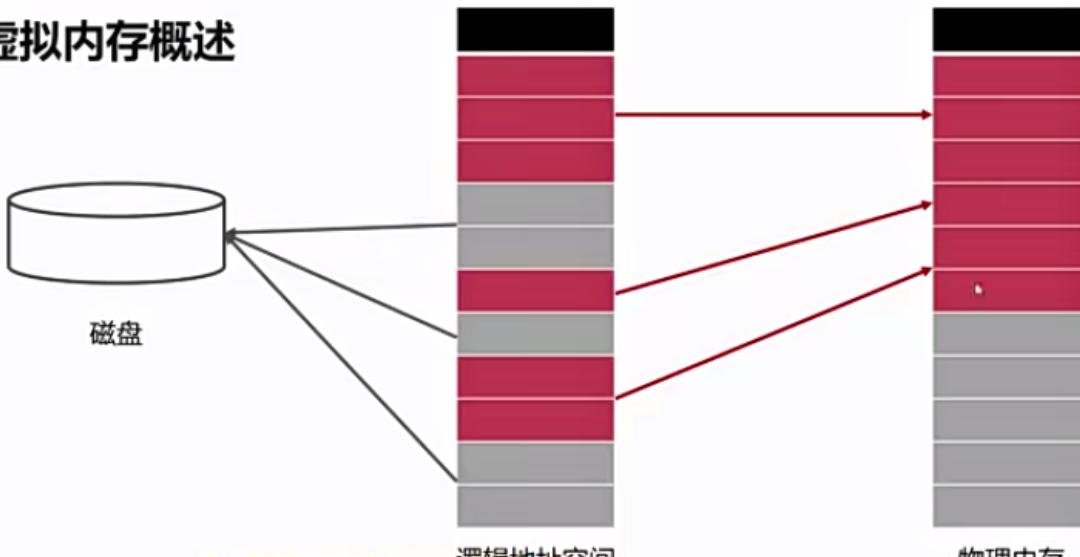
虚拟内存概述

逻辑地址空间

物理地址空间

- | | |
|---|---|
| <ul style="list-style-type: none">◆ 逻辑地址空间是指进程可以使用的内存空间◆ 逻辑地址空间的大小仅受CPU地址长度限制◆ 32位地址最大逻辑空间为4G◆ 逻辑地址空间是一个进程运行时程序指令与程序数据可以用的相对地址空间 | <ul style="list-style-type: none">◆ 物理地址指向物理内存的存储空间◆ 物理地址空间是指程序运行过程在物理内存分配和使用的地址空间 |
|---|---|

虚拟内存概述



虚拟内存概述

- ◆ 程序运行时，无需全部装入内存，装载部分即可
- ◆ 如果访问页不在内存，则发出缺页中断，发起页面置换
- ◆ 从用户层面看，程序拥有很大的空间，即是虚拟内存

虚拟内存实际是对物理内存的补充，速度接近于内存，成本接近于辅存

2. 当缓存没有数据时



高速缓存替换的时机

3. 当主存没有数据时



主存页面的替换时机

4. 总结

- ◆ 替换策略发生在Cache-主存层次、主存-辅存层次
- ◆ Cache-主存层次的替换策略主要是为了解决速度问题
- ◆ 主存-辅存层次主要是为了解决容量问题

2.2.3 缺页中断

1. 操作系统如何管理进程空间

2. 内存管理

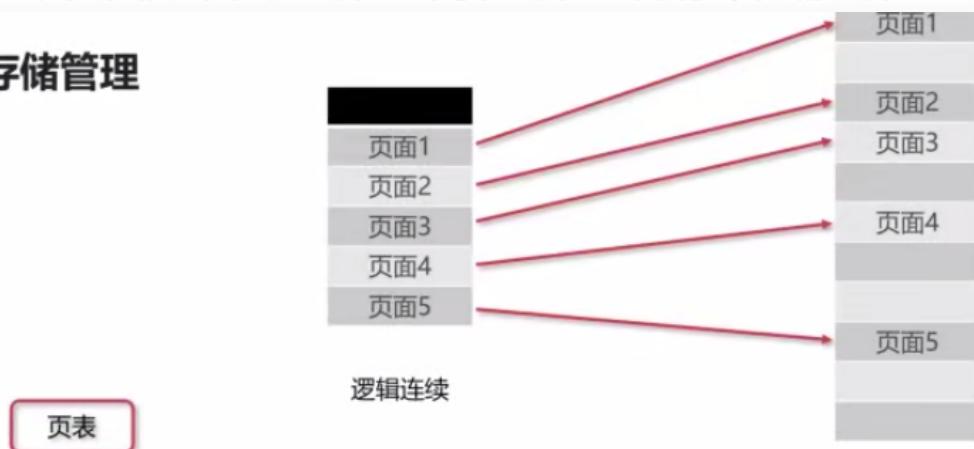
1. 页式存储管理
2. 段式存储管理
3. 段页式存储管理

3. 页式存储管理

页式存储管理

- ◆ 将进程逻辑空间等分成若干大小的页面
- ◆ 相应的把物理内存空间分成与页面大小的物理块
- ◆ 以页面为单位^⑤把进程空间装进物理内存中分散的物理块

页式存储管理



- ◆ **页表**记录进程逻辑空间与物理空间的映射

物理分散

页式存储管理

- ◆ 页面大小应该适中，过大难以分配，过小页表管理空间大

- ◆ 页面大小通常是512B~8K

页式存储管理

现代计算机系统中，可以支持非常大的逻辑地址空间 (2^{32} ~ 2^{64})，这样，页表就变得非常大，要占用非常多的内存空间，如，具有**32位**逻辑地址空间的分页系统，规定页面大小为**4KB**，则在每个进程页表中的页表项可达**1M(2^{20})**个，如果每个页表项占用**1Byte**，故每个进程仅仅页表就要占用**1MB**的内存空间。

32位系统进程的寻址空间为4G

$4G/4KB = 2^{20}$

多级页表

页式存储管理

- ◆ 将进程逻辑空间等分成若干大小的页面
- ◆ 相应的把物理内存空间分成与页面大小的物理块
- ◆ 以页面为单位把进程空间装进物理内存中分散的物理块

有一段连续的逻辑分布在多个页面中，将大大降低执行效率

4. 段式存储管理

段式存储管理

- ◆ 将进程逻辑空间划分成若干段（非等分）
- ◆ 段的长度由连续逻辑的长度决定

◆ 主函数MAIN、子程序段X、子函数Y等

段式存储管理



段式存储管理相比页式存储管理更加灵活

主存

段页式存储管理

- ◆ 分页管理有效提高内存利用率

5. 段页式存储管理

- ◆ 分段管理更加灵活

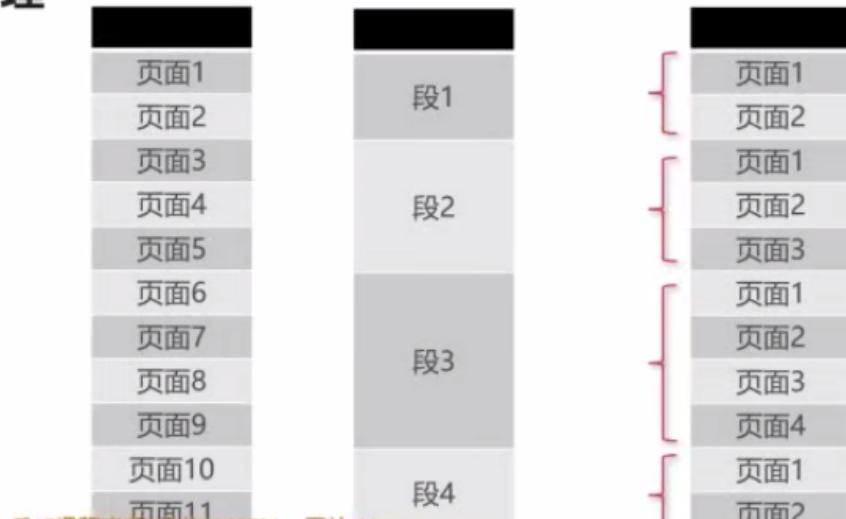
- ◆ 两者结合，形成段页式存储管理

段页式存储管理

- ◆ 先将逻辑空间按段式管理分成若干段
- ◆ 再把段内空间按页式管理等分成若干页



段页式存储管理



6. 缺页中断

缺页中断



在请求分页系统中，可以通过查询页表中的状态位来确定所要访问的页面是否存在于内存中。

每当所要访问的页面不在内存时，会产生一次**缺页中断**，此时操作系统会根据页表中的**外存地址**在外存中找到所缺的一页，将其调入内存。

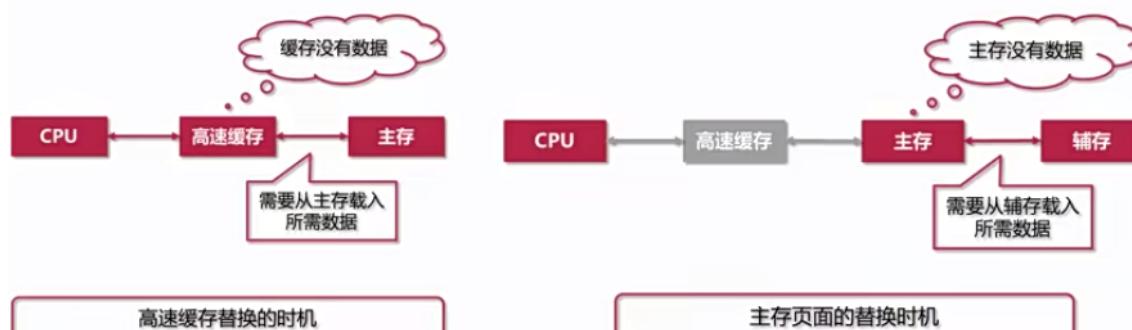
- ◆ 磁盘属于外设，读写磁盘需要**系统调用**
- ◆ 保护CPU现场→分析中断原因→中断处理→恢复CPU环境
- ◆ 在指令执行期间产生和处理中断信号
- ◆ 一条指令执行期间，可能产生多次缺页中断

系统调用属于内核态

2.2.4 页面置换算法

1. 页面置换时机

页面置换的时机



2. 缓存置换算法

1. 先进先出 FIFO
 2. 最近最少使用 LRU
 3. 最不经常使用 LFU
3. 先进先出，类似队列

先进先出算法(FIFO)



4. 最不经常使用

最不经常使用算法(LFU)

◆ 优先淘汰最不经常使用的字块

◆ 需要额外的空间记录字块的使用频率

最不经常使用算法(LFU)

缓存	1	2	3	4	5	6	7	8
频率	0	0	0	0	0	0	0	0

⇒ 访问: 2

缓存	1	2	3	4	5	6	7	8
频率	0	1	0	0	0	0	0	0

⇒ 访问: 6

缓存	1	2	3	4	5	6	7	8
频率	0	1	0	0	0	1	0	0

最不经常使用算法(LFU)

缓存	1	2	3	4	5	6	7	8
频率	7	4	2	1	8	4	2	6

← 9

缓存	1	2	3	4	5	6	7	8
频率	7	4	2	1	8	4	2	6

4	←	缓存	1	2	3	9	5	6	7	8
频率	7	4	2	1	8	1	8	4	2	6

5. 最近最少使用

最近最少使用算法 (LRU)

- ◆ 优先淘汰一段时间内没有使用的字块
- ◆ 有多种实现方法，一般使用双向链表
- ◆ 把当前访问节点置于链表前面（保证链表头部节点是最近使用的）

最近最少使用算法 (LRU)

- | | |
|-----------------|-----------------|
| (1) 1 | (6) 6、4、5、7 [2] |
| (2) 2、1 | (1) 1、6、4、5 [7] |
| (4) 4、2、1 | (6) 6、1、4、5 |
| (7) 7、4、2、1 | (7) 7、6、1、4 [5] |
| (5) 5、7、4、2 [1] | (4) 4、7、6、1 |
| (4) 4、5、7、2 | (1) 1、4、7、6 |

跟FIFO类似，但FIFO是严格的队列。而LRU是会“插队”的

2.2.5 软链接与硬链接

1. 常见文件系统

常见文件系统

FAT

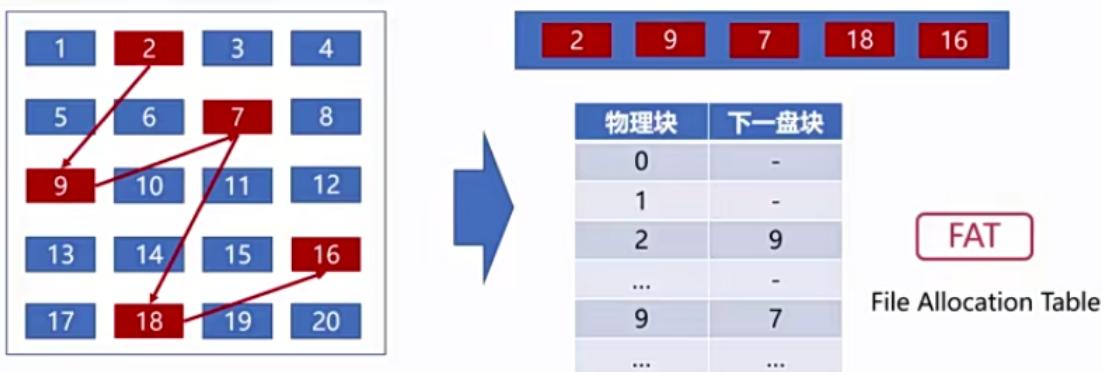
NTFS

ext2/3/4

FAT文件系统

- ◆ FAT(File Allocation Table)
- ◆ FAT16、FAT32等，微软DOS/Windows使用的文件系统
- ◆ 使用一张表保存盘块的信息

FAT文件系统



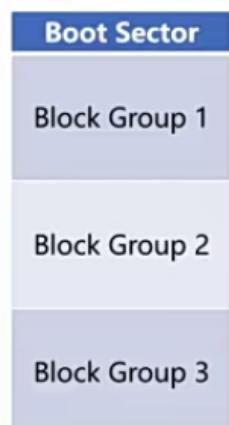
NTFS文件系统

- ◆ NTFS (New Technology File System)
- ◆ Windows NT 环境的文件系统
- ◆ NTFS 对 FAT 进行了改进，取代了旧的文件系统

EXT文件系统

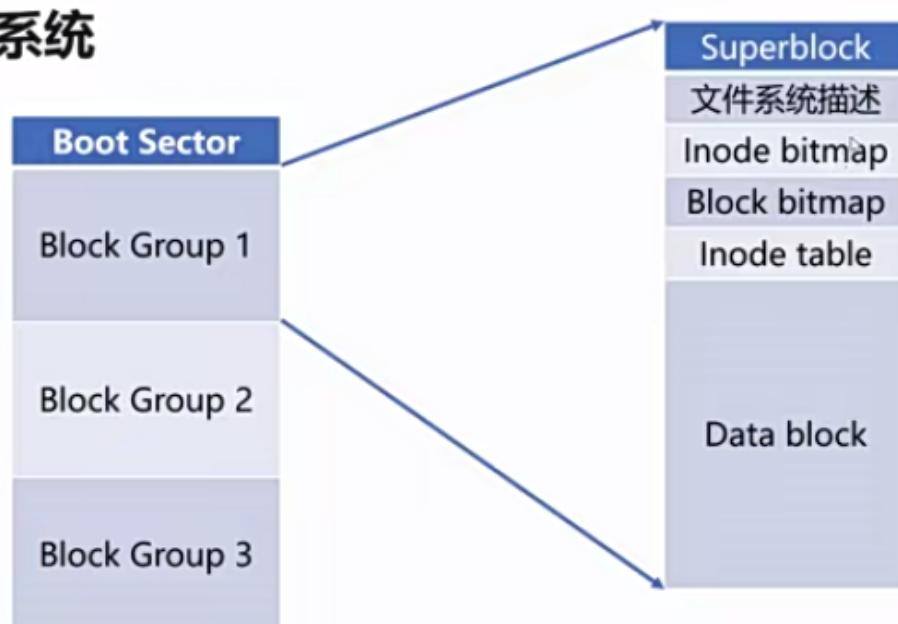
- ◆ EXT(Extended file system): 扩展文件系统
- ◆ Linux的文件系统
- ◆ EXT2/3/4 数字表示第几代

EXT文件系统



- ◆ Boot Sector: 启动扇区，安装开机管理程序
- ◆ Block Group: 块组，存储数据的实际位置

Ext文件系统



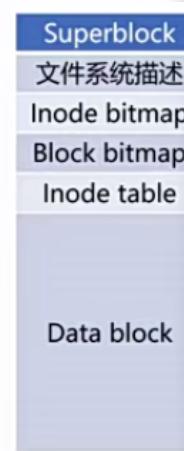
Ext文件系统

- ◆ 存放文件Inode的地方
- ◆ 每一个文件（目录）都有一个Inode
- ◆ 是每一个文件（目录）的索引节点



Inode Table

Ext文件系统



Ext文件系统

- ◆ 文件名不是存放在Inode节点上的，而是存放在目录的Inode节点
- ◆ 列出目录文件的时候无需加载文件的Inode

Inode

Ext文件系统

- ◆ Inode的位示图
- ◆ 记录已分配的Inode和未分配的Inode

Inode表	1	2	3	4	5	6	7	8	...
-	0	0	0	1	1	0	0	0	...

Inode bitmap



Ext文件系统

- ◆ Data block是存放文件内容的地方
- ◆ 每个block都有唯一的编号
- ◆ 文件的block记录在文件的Inode上

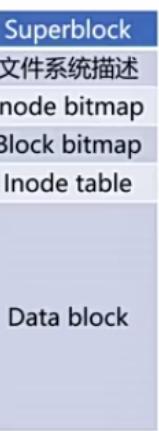
Data block



Ext文件系统

- ◆ 功能与Inode bitmap类似
- ◆ 记录Data block的使用情况

Block bitmap



Ext文件系统

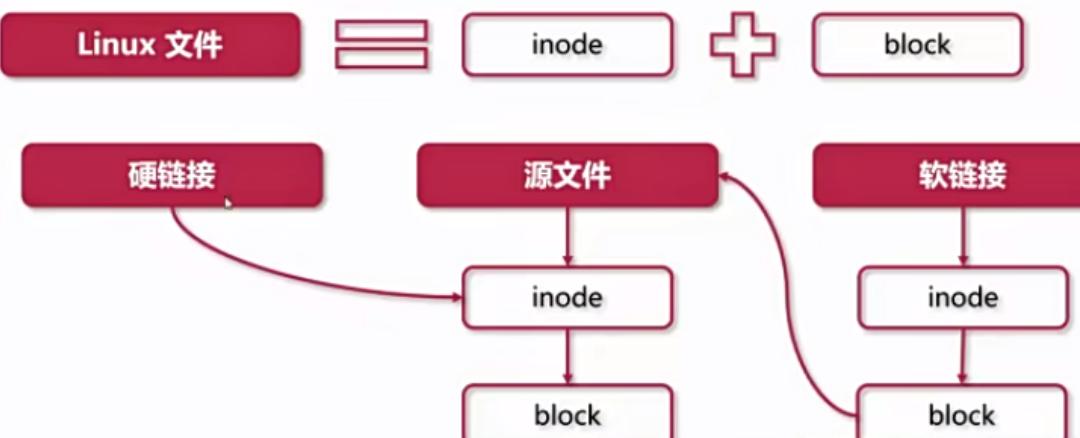
- ◆ 记录整个文件系统相关信息的地方
- ◆ Block和Inode的使用情况
- ◆ 时间信息、控制信息等

Superblock



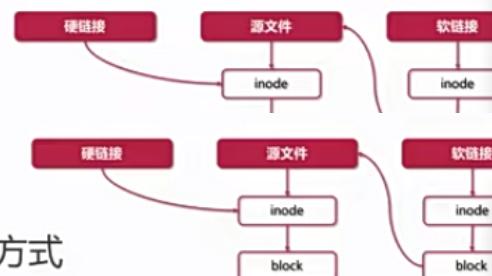
5. 软链接-硬链接

软链接-硬链接



软链接-硬链接

- ◆ 具有相同inode节点号的文件互为硬链接文件
- ◆ 删除硬链接文件或者删除源文件任意之一，文件实体并未被删除
- ◆ 创建硬链接命令 ln 源文件 硬链接文件



软链接-硬链接

- ◆ 软链接类似windows系统的快捷方式
- ◆ 软链接里面存放的是源文件的路径，指向源文件
- ◆ 删除源文件，软链接依然存在，但无法访问源文件内容

2.2.6 磁盘冗余阵列

1. RAID是什么

RAID是什么？

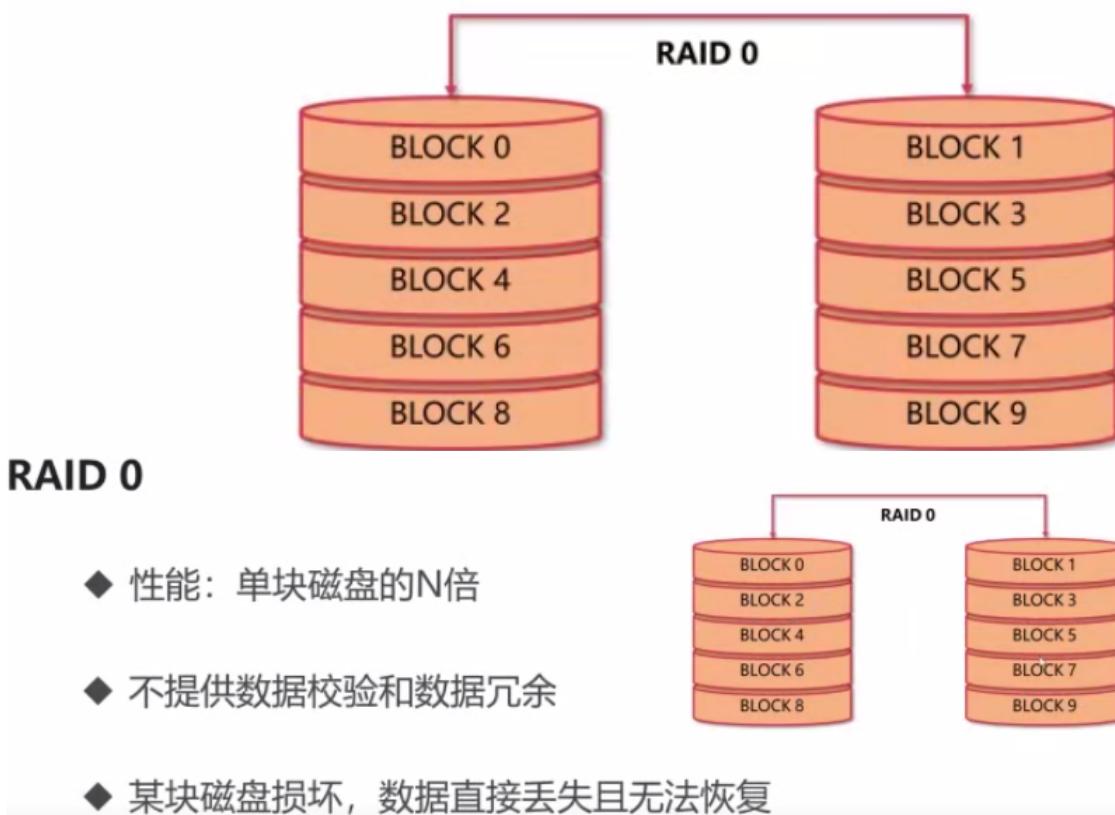
- ◆ RAID(Redundant Array of Independent Disks): 磁盘冗余阵列

磁盘冗余阵列：利用虚拟化存储技术把多个硬盘组合起来，成为一个或多个硬盘阵列组，目的为提升性能或减少冗余，或是两者同时提升。



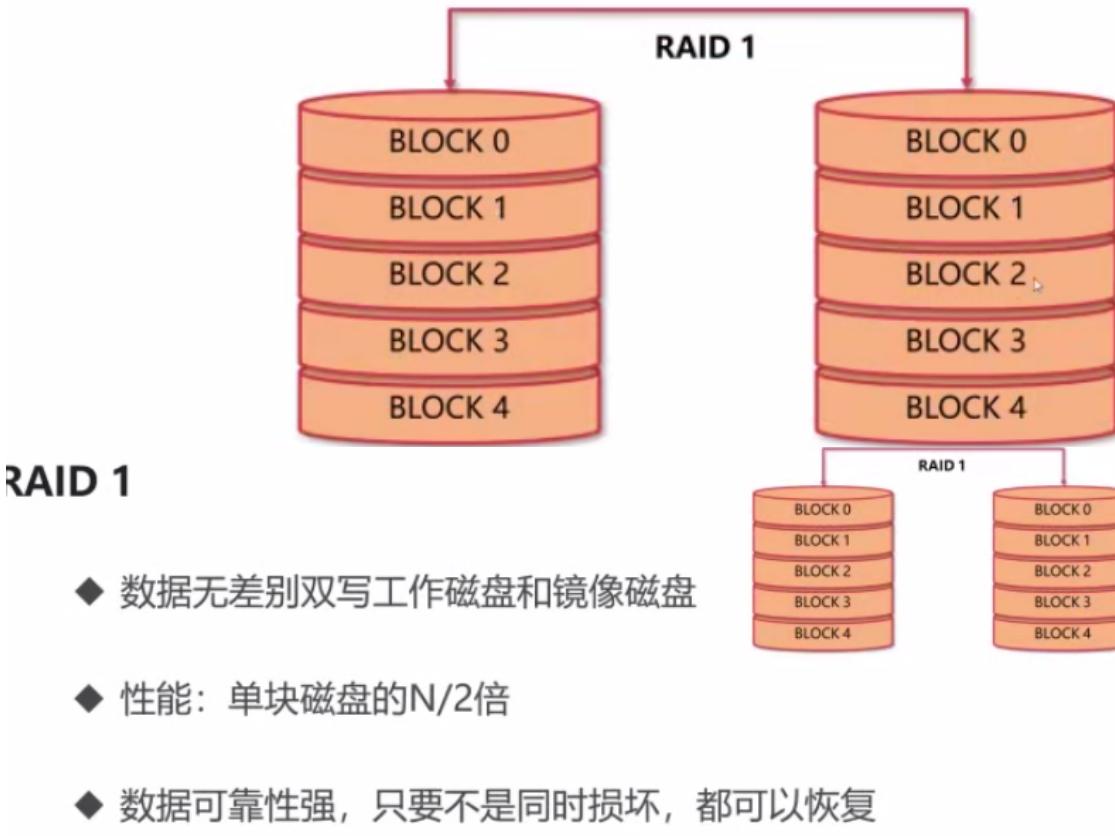
2. RAID 0

RAID 0



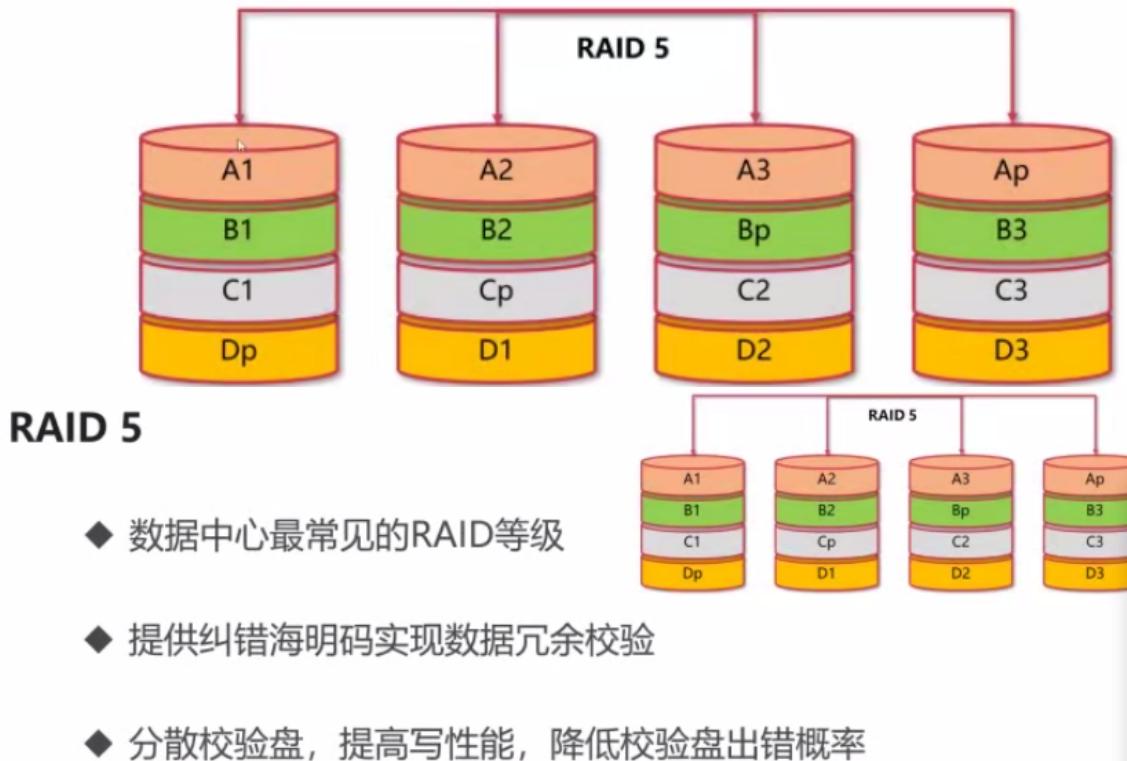
3. RAID 1

RAID 1



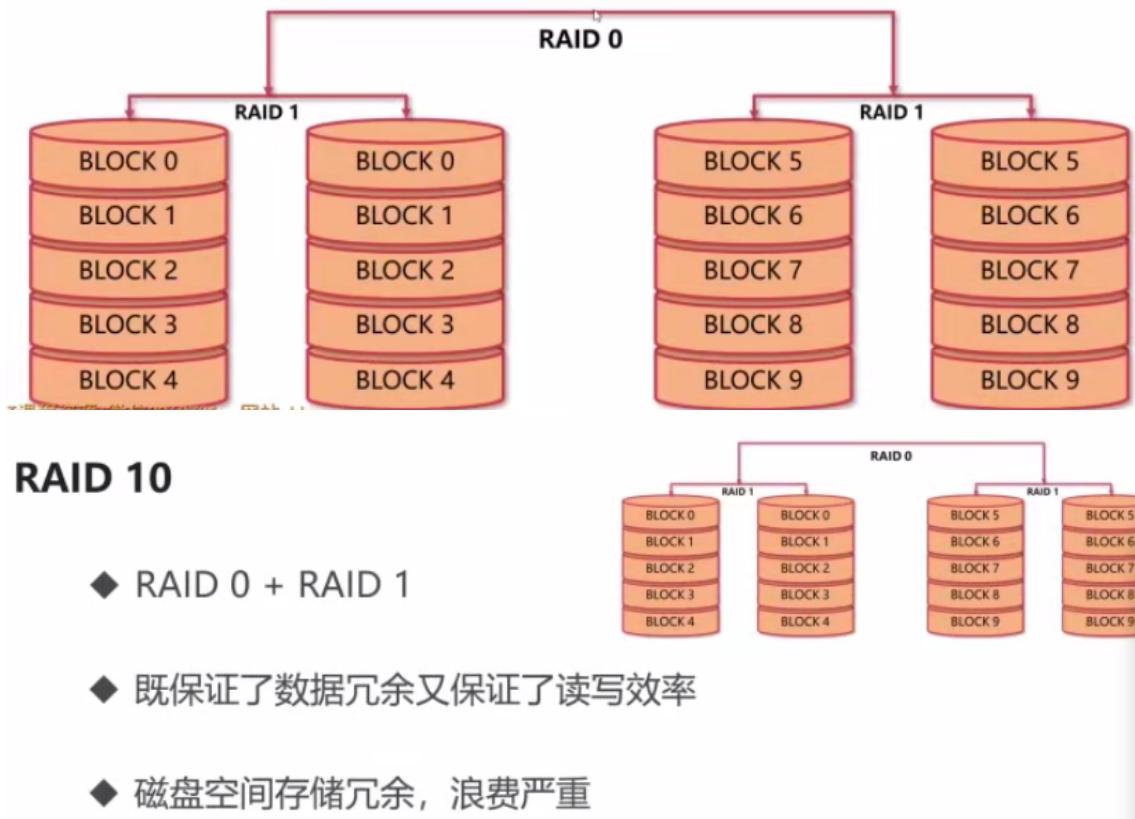
4. RAID 5

RAID 5



5. RAID 10

RAID 10



6. 对比

RAID对比

	RAID 0	RAID 1	RAID 5	RAID 10
数据保护	不提供	提供	提供	提供
写效率	高	高	低	高
读效率	高	中	高	高
容量	100%	50%	67% ~ 94%	50%
应用	读写要求高	数据安全/容易恢复	兼顾经济性/数据安全	数据安全/容易恢复

三、计算机系统

3.1 锁、同步与通信

3.1.1 死锁

1. 定义

死锁

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

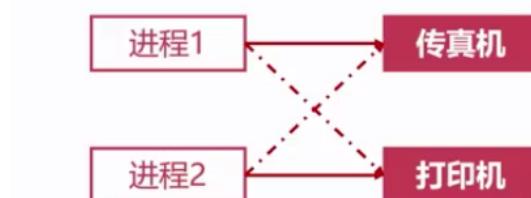


2. 产生原因

1. 竞争资源

- 共享资源资源数量不满足各个进程需求
- 各个进程之间发生资源竞争导致死锁

竞争资源



◆ 等待请求的资源被释放

◆ 自身占用资源不释放

2. 进程调度顺序不当

调度顺序不当



3. 死锁的四个必要条件

1. 互斥条件

1. 进程对资源的使用是**排他性的使用**
2. 某资源只能由一个进程使用，其他进程需要使用只能等待

2. 请求保持条件

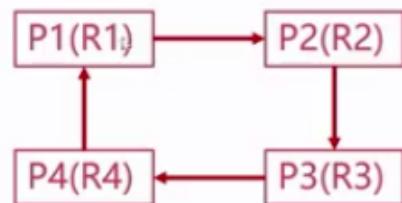
1. 进程至少保持一个资源，又提出新的资源请求
2. 新资源被占用，请求被阻塞
3. 被阻塞的进程不释放自己保持的资源

3. 不可剥夺条件

1. 进程获得的资源在未完成使用前不能被剥夺
2. 获得的资源只能由进程自身释放

4. 环路等待条件

◆ **发生死锁时，必然存在进程-资源环形链**



5. 预防死锁方法，破坏以上原因(除开互斥)

摒弃请求保持条件

- ◆ 系统规定进程运行之前，一次性申请所有需要的资源
- ◆ 进程在运行期间不会提出资源请求，从而摒弃请求保持条件

摒弃不可剥夺条件

- ◆ 当一个进程请求新的资源得不到满足时，必须释放占有的资源
- ◆ 进程运行时占有的资源可以被释放，意味着可以被剥夺

摒弃环路等待条件

- ◆ 可用资源线性排序，申请必须按照需要递增申请
- ◆ 线性申请不再形成环路，从而摒弃了环路等待条件



6. 银行家算法

银行家算法

- ◆ 客户申请的贷款是有限的，每次申请需声明最大资金量
- ◆ 银行家在能够满足贷款时，都应该给用户贷款
- ◆ 客户在使用贷款后，能够及时归还贷款

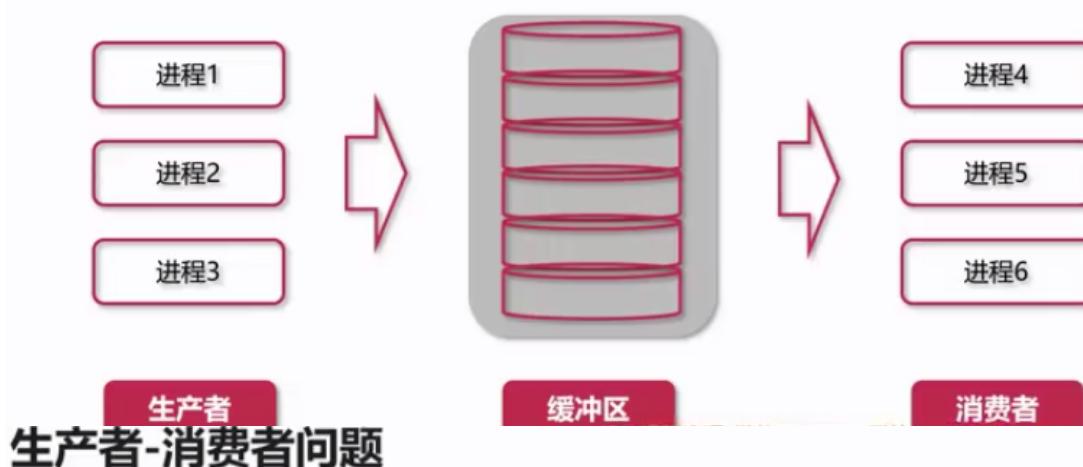
3.1.2 线程-进程同步问题

1. 生产者-消费者问题

生产者-消费者问题

- ◆ 一组生产者进程、一组消费者进程、一个缓冲区
- ◆ 生产者在缓冲区溢出前，不断往缓冲区生产数据
- ◆ 消费者在缓冲区为空前，不断从缓冲区消费数据

生产者-消费者问题

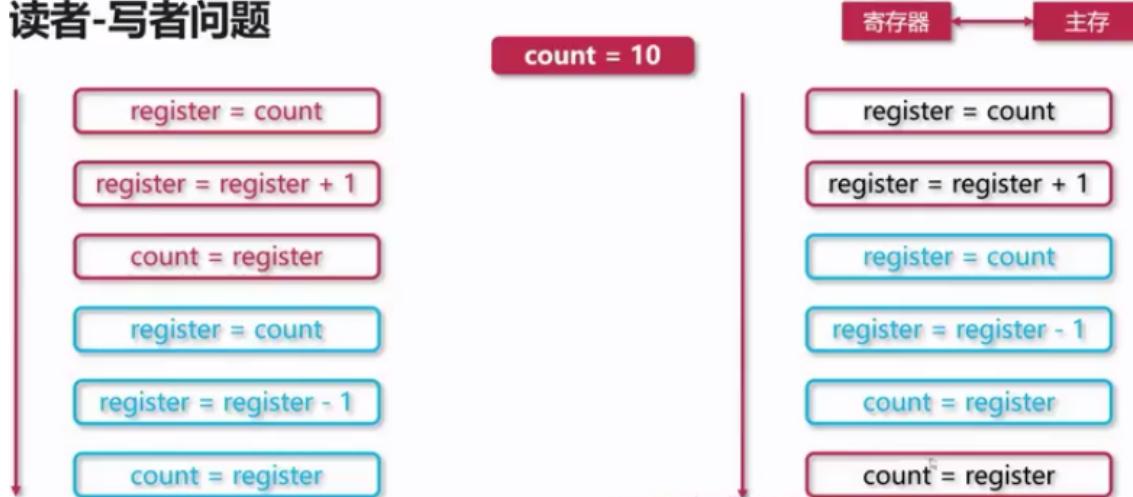


生产者-消费者问题

- ◆ 生产者-消费者通过缓冲区存在同步关系
 - ◆ 当缓冲区满时，生产者必须等待消费者消费数据
 - ◆ 当缓冲区空时，消费者必须等待生产者生产数据
- ◆ 生产者-消费者、生产者之间、消费者之间存在互斥关系
 - ◆ 对缓冲区数据进行存取操作时，必须互斥进行

2. 读者-写者问题

读者-写者问题



读者-写者问题

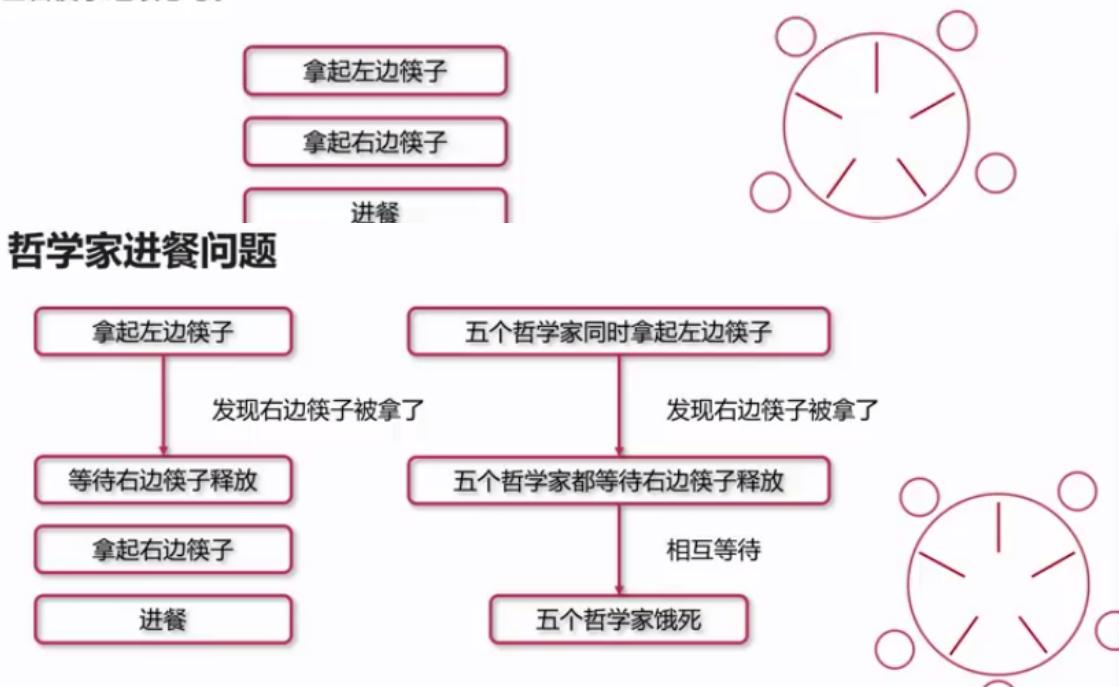
◆ 读-写操作之间存在同步关系

◆ 多个写操作应该串行完成

3. 哲学家进餐问题

哲学家进餐问题

有五个哲学家，他们的生活方式是交替地进行思考和进餐，哲学家们共同使用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子。平时哲学家们只进行思考，饥饿时则试图取靠近他们的左、右两支筷子，只有两支筷子都被他拿到的时候就能进餐，进餐完毕之后，放下左右筷子继续思考。



4. 临界资源

临界资源

临界资源指的是一些虽作为共享资源却又无法同时被多个线程共同访问的共享资源。当有进程在使用临界资源时，其他进程必须依据操作系统的同步机制等待占用进程释放该共享资源才可重新竞争使用共享资源。

缓冲区

读/写数据

筷子

5. 原子性

原子性

- ◆ 原子性是指一系列操作不可被中断的特性
- ◆ 这一系列操作要么全部执行完成，要么全部没有执行
- ◆ 不存在部分执行部分未执行的情况

3.1.3 乐观锁、悲观锁与可重入锁

1. 特点

乐观锁/悲观锁

- ◆ 悲观锁每次操作都加锁、乐观锁默认不添加锁
- ◆ 悲观锁适合写操作的场景
- ◆ 乐观锁适合读操作的场景

无锁/偏向锁/轻量级锁/重量级锁

- ◆ 无锁：不锁资源，多个线程只一个线程修改成功，其他线程会重试
- ◆ 偏向锁：同一个线程执行临界资源会自动获取资源
- ◆ 轻量级锁：多个线程竞争同步资源时，没有获得资源的线程自旋等待锁释放
- ◆ 重量级锁：多个线程竞争同步资源时，没有获得资源的线程阻塞等待唤醒

这四个的重要性依次递增

2. 公平锁/非公平锁

公平锁/非公平锁



公平锁/非公平锁

- ◆ 公平锁的优点：等待锁的线程不会饥饿等待
- ◆ 公平锁的缺点：整体吞吐效率相对非公平锁要低
- ◆ 非公平锁的优缺点：整体的吞吐效率高，CPU不必唤醒所有线程

3. 可重入锁/非可重入锁

可重入锁/非可重入锁

- ◆ 重入：任意线程获取锁以后，这个线程再次获得该锁时不会阻塞

```
Function fun() {
```

```
    ...<acquire lock> {
```

```
        ...<acquire lock>
```

```
    }
```

造成死锁

```
Function fun() {
```

```
    ...<acquire lock> {
```

```
        ...<acquire lock>
```

```
    }
```

不会死锁

可重入锁/非可重入锁

- ◆ 可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁，不会因为之前已经获取过还没释放而阻塞。

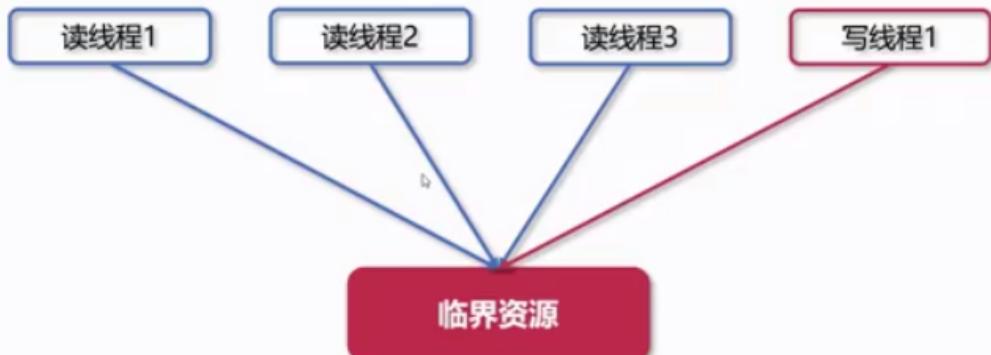
- ◆ 不可重入锁，当前线程再次获取当前线程已经获得的锁时，如果该锁仍被当前线程所持有，未被释放，那么将会出现死锁。

4. 共享锁和排他锁

共享锁/排他锁

- ◆ 排他锁（互斥锁）是指该锁一次只能被一个线程所持有
- ◆ 共享锁是指该锁可被多个线程所持有
- ◆ 获得共享锁的线程只能读数据，不能修改数据

共享锁/排他锁

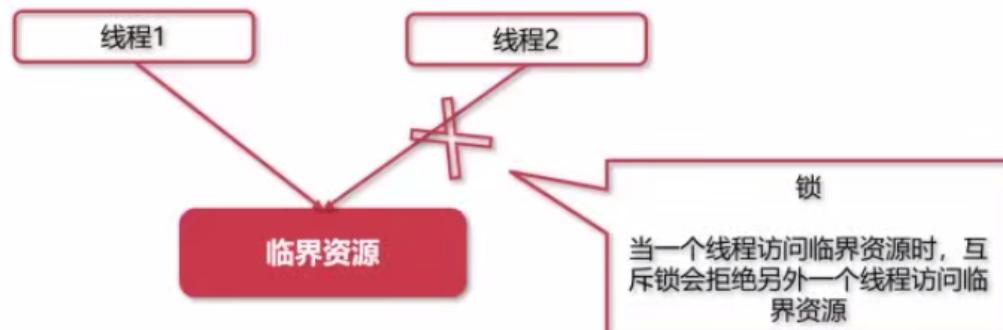


读线程使用共享锁，写线程使用排他锁

3.1.4 线程间通信方式

1. 互斥锁

互斥锁(mutex)



互斥锁(mutex)

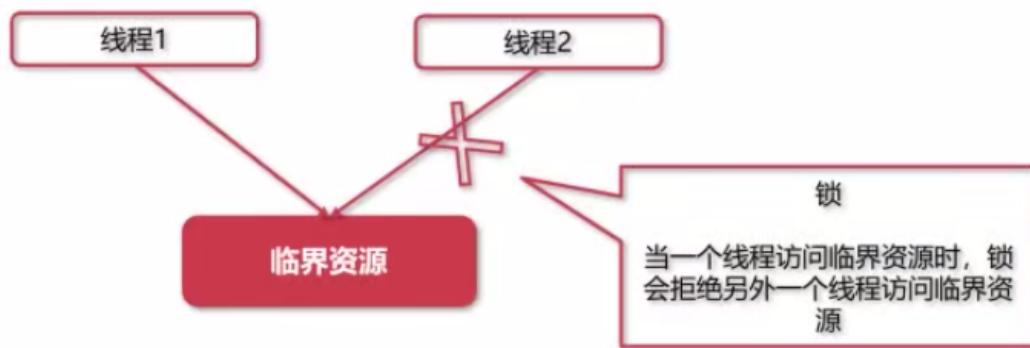


互斥锁(mutex)

- ◆ 互斥量是最简单的线程同步的方法
- ◆ 互斥量（互斥锁），处于两态之一的变量：解锁和加锁
- ◆ 两个状态可以保证资源访问的串行
- ◆ 操作系统层和高级语言都直接提供了接口，可直接使用

2. 自旋锁

自旋锁(spin_lock)



自旋锁实现原理与互斥锁实现原理不一样

自旋锁(spin_lock)

- ◆ 自旋锁也是一种多线程同步的变量
- ◆ 使用自旋锁的线程会反复检查锁变量是否可用
- ◆ 自旋锁不会让出CPU，是一种**忙等待**状态

自旋锁(spin_lock)



会不会让出CPU是自旋锁与互斥锁的区别，这样的好处是

自旋锁(spin_lock)

- ◆ 自旋锁避免了进程或线程上下文切换的开销
- ◆ 操作系统内部很多地方使用的是自旋锁
- ◆ 自旋锁不适合在单核CPU使用

3. 读写锁

读写锁(rwlock)



读写锁(rwlock)

- ◆ Read-write locks are a special type of spin locks
- ◆ Allows multiple readers to access the resource simultaneously to improve read performance
- ◆ For write operations, it is mutual-exclusion

4. 条件变量

条件变量(condition)

- ◆ 条件变量是一种相对复杂的线程同步方法
- ◆ 条件变量允许线程睡眠，直到满足某种条件
- ◆ 当满足条件时，可以向该线程信号，通知唤醒



条件变量(condition)

- ◆ 生产者-消费者通过缓冲区存在同步关系

- ◆ 当缓冲区满时，生产者必须等待消费者消费数据
- ◆ 当缓冲区空时，消费者必须等待生产者生产数据



3.1.5 进程间通信

1. 进程与线程的区别

进程VS线程

- ◆ 线程是系统进行运行调度的最小单位
- ◆ 进程是系统进行资源分配和调度的基本单位
- ◆ 互斥锁(mutex)
- ◆ 读写锁(rwlock)
- ◆ 自旋锁(spin_lock)
- ◆ 条件变量(condition)

因为进程间的资源是相互独立的，所以线程间通信方式不适用

2. 进程间通信的方法

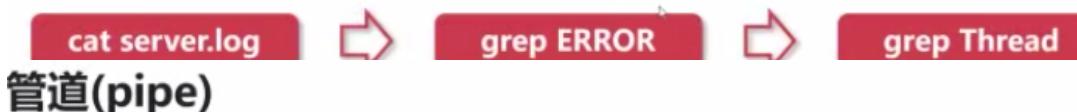


1. 管道

管道(pipe)

```
> netstat -anlp | grep 8080
> cat server.log | grep ERROR | grep Thread
> man netstat | more
```

这里的 “|” 实际上就是管道的意思， “|” 前面部分作为 “|” 后面的输入。



管道(pipe)

```
> netstat -anlp | grep 8080
> cat server.log | grep ERROR | grep Thread
> man netstat | more
```

匿名管道

命名管道

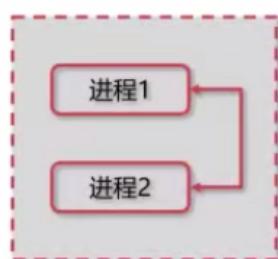
```
> mkfifo test_pipe
> echo "this is a test pipe demo" > test_pipe
> cat test_pipe
```

2. 消息队列

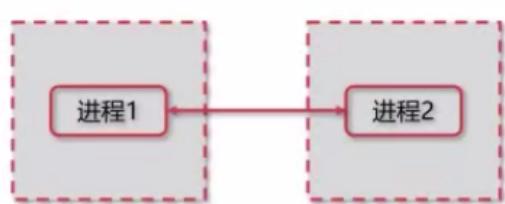
消息队列



消息队列



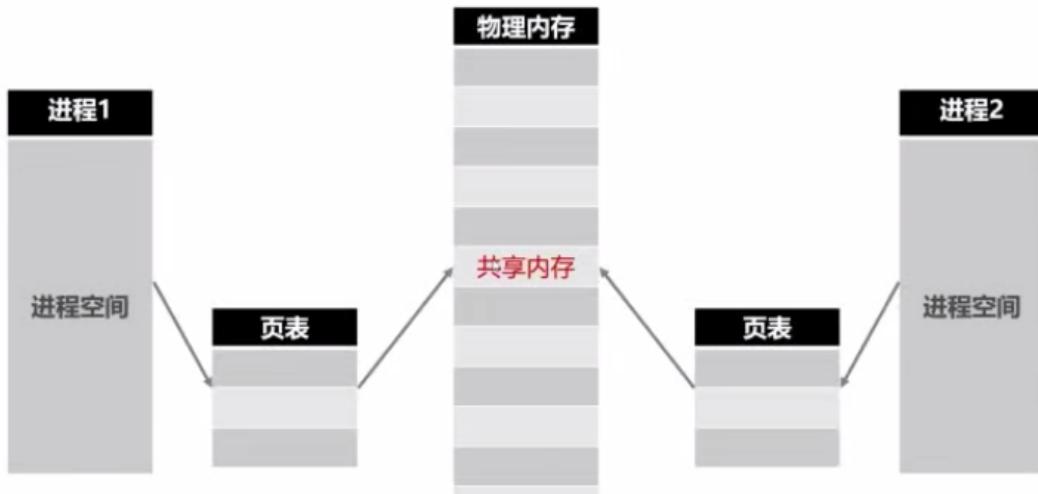
单机进程间通信



跨机进程间通信

> 消息中间件：Kafka、RabbitMQ...

3. 共享内存



共享内存

- ◆ 在某种程度上，多进程是共同使用物理内存的
- ◆ 由于操作系统的进程管理，进程间的内存空间是独立的
- ◆ 共享存储允许不相关的进程访问同一片物理内存
- ◆ 共享内存是两个进程之间共享和传递数据最快的方式
- ◆ 共享内存未提供同步机制，需要借助其他机制管理访问

4. 信号

信号



- ◆ 在操作系统中，不同信号使用不同的值来表示
- ◆ 接收信号的进程需要注册对应的信号处理函数

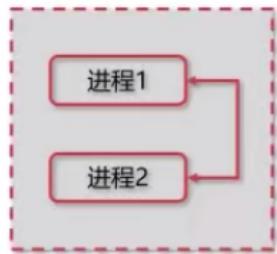
信号

```
> kill -l # 查看支持的信号列表
```

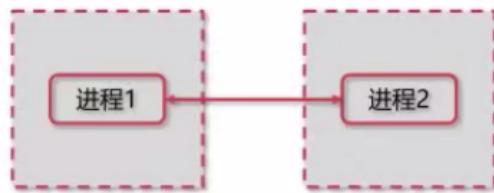
信号名字	信号取值	信号说明
SIGHUP	1	挂起，在用户终端连接(正常或非正常)结束时发出
SIGINT	2	中断，用于通知前台进程组终止进程
SIGQUIT	3	退出，和SIGINT类似，但由QUIT字符(通常是Ctrl-\)来控制
SIGKILL	9	Kill信号
SIGSTKFLT	16	栈溢出

5. 套接字

套接字



单机进程间通信



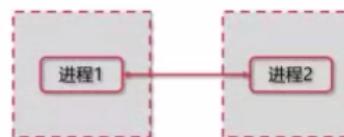
跨机进程间通信

网络套接字

网络层：提供主机之间的通信

传输层：提供主机**不同进程之间的通信**

应用层：提供不同应用之间的通信



跨机进程间通信

主机

IP地址

套接字(Socket)

IP地址

端口

主机网络进程

端口

◆ 16个比特位

◆ 端口范围：0 ~ 65535

网络套接字



域套接字

- ◆ 套接字(socket)原是网络通信中使用的术语
- ◆ 域套接字是一种高级的进程间通信的方法
- ◆ Unix域套接字可以用于同一机器进程间通信
- ◆ Unix系统提供的域套接字提供了网络套接字类似的功能
- ◆ Unix域套接字通信无需经过完整的网络协议栈

套接字



前者是server端，后者是client端

3.1.6 CAS原理与无锁技术

1. 锁的弊端

大量使用锁的弊端

- ◆ 开发难度：并行系统访问临界资源必须考虑加锁
- ◆ 墨菲定律：只要存在的一定会发生，死锁
- ◆ 调度问题：低优先级线程持有锁导致高优先级线程无法执行
- ◆ 性能问题：满足一致性要求的前提下需要串行访问
- ◆ 锁粒度：锁粒度过小/过大，设计不当

2. CAS技术

基石：CAS技术

原子性：原子性是指一系列操作不可被中断的特性；这一系列操作要么全部执行完成，要么全部没有执行，不存在部分执行，部分未执行的情况。



基石：CAS技术

CAS——Compare & Set，或是 Compare & Swap，现在几乎所有的CPU指令都支持**CAS的原子操作**，X86下对应的是 CMPXCHG 汇编指令。

```
int compare_and_swap(int* reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval) {
        *reg = newval;
        return old_reg_val;
    }
}
```

比较旧值

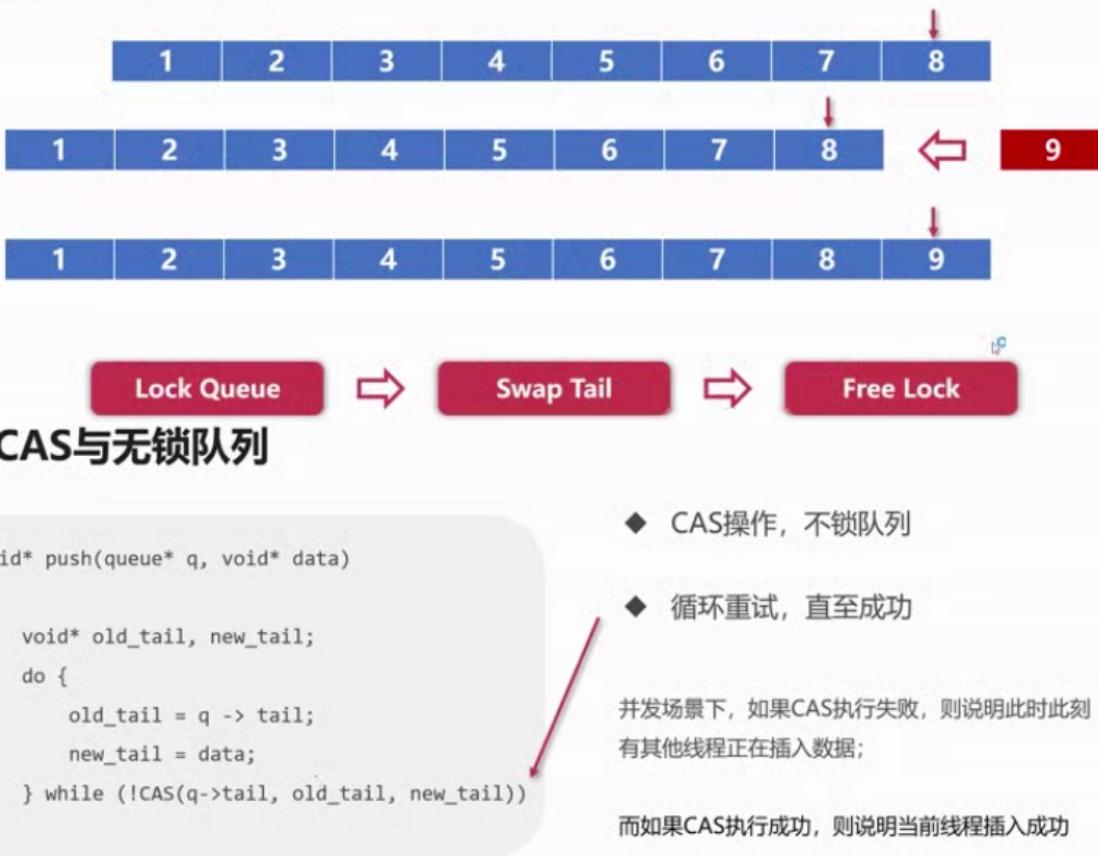
交换新值

基石：CAS技术

- ◆ Fetch And Add：一般用来对变量做 +1 的原子操作
- ◆ Test And Set：写值到某个内存位置并传回其旧值

3. CAS与无锁队列

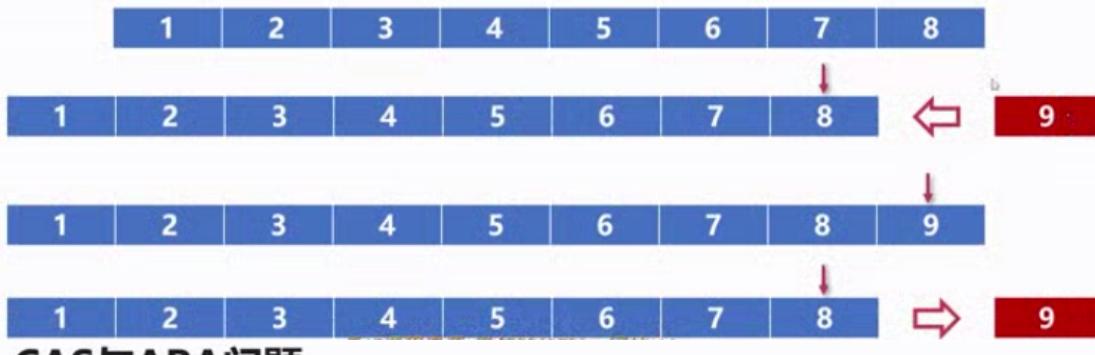
CAS与无锁队列



4. CAS弊端

CAS与ABA问题

ABA问题是指CAS交换数据在多次操作后恢复原值而线程无法感知的问题。



```
int compare_and_swap(int* reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval) {
        *reg = newval;
        return old_reg_val;
    }
}
```

当前线程以为的旧值：tail = 8

实际的旧值：tail = 8 → tail = 9 → tail = 8

当前线程认为CAS成功了，实际上当前tail已经被修改多次

解决方法：像Java一样增加一个版本号，如果发生改变那么就有变化

3.1.7 分布式锁实现

1. 应用场景

1. 订单系统，秒杀系统
2. 积分系统，消费系统
3. 消息中间件，服务中间件，数据发布-订阅
4. 分布式部署：集群、微服务
5. 服务节点之间需要通信
6. 数据强一致要求，性能要求，并发量要求

2. redis

基于Redis

Redis是一个使用ANSI C编写的开源、支持网络、基于内存、分布式、可选持久性的键值对存储数据库。

- ◆ 读写性能优异
- ◆ 内存读写，可持久化数据
- ◆ 数据类型丰富、单线程、数据自动过期、发布-订阅

基于Redis



- | | |
|---|---|
| <ul style="list-style-type: none">◆ Redis：性能优异的k-v数据库◆ setnx <key> <value>◆ del <key>◆ 单点问题、雪崩效应 | <ul style="list-style-type: none">◆ 避免单点问题◆ 节点一致性由集群保证◆ 集群如何保证一致性的？ |
|---|---|

→ Redis集群原理

3. Zookeeper

基于Zookeeper

ZooKeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，是Hadoop和Hbase的重要组件。

它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

基于Zookeeper

- ◆ Zookeeper数据节点：znode
- ◆ 服务1在Zookeeper创建znode1
- ◆ 服务2在Zookeeper创建znode1失败
- ◆ 服务1释放znode，服务2创建成功

临时节点：临时节点由某个客户端创建，当客户端与ZK集群断开连接，则该节点自动被删除

4. MySQL

基于传统数据库：MySQL

- ◆ MySQL提供一致性服务：事务、表级锁、行级锁
- ◆ UNIQUE KEY：表级唯一，不能重复插入
- ◆ 通过MySQL保证同一个KEY只有一个节点能插入成功
- ◆ 通过删除记录释放锁

把锁竞争的压力交给了MySQL，且MySQL同样存在单点问题，需要集群解决

5. 分布式锁框架

分布式锁框架

京东: SharkLock

Netflix: Curator

Google: Chubby

ETCD

Redisson

consul

3.2 编程语言与运行原理

3.2.1 计算机层次结构

1. 一个复杂的系统应该有清洗明确的分层设计

2. 层次

计算机的层次结构



计算机的层次结构

- ◆ 满足用户在各种场景下便捷使用计算机的需求

办公场景

娱乐场景

学习场景

应用层

计算机的层次结构

- ◆ 程序员面向的计算机层次
- ◆ Java、C/C++、Python、PHP、Javascript...
- ◆ 编程实现易用的应用层软件提供用户使用

高级语言层

计算机的层次结构

- ◆ 偏底层软件工程师面向的计算机层次
 - ◆ 嵌入式工程师、二进制安全、机械自动化...
 - ◆ 汇编语言可以翻译成可直接执行的机器语言
- PUSH DS
PUSH AX
MOV AX,0040
MOV DS,AX

汇编语言层

计算机的层次结构

- ◆ 传统硬件与传统软件的分界线
- ◆ 向上提供了简易的操作接口
- ◆ 向下对接了指令系统，管理硬件资源

操作系统层

3.2.2 编译与解释

1. 编译与解释



2. 编译型

1. C/C++
2. Object-C
3. Golang

3. 解释型

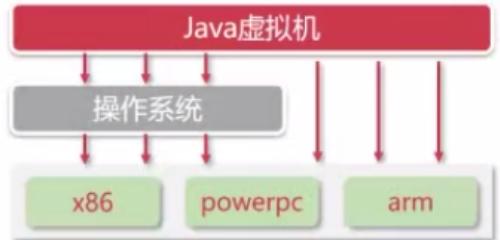
1. Python(像Java, 不是纯粹的解释语言)
2. Php
3. Javascript
4. 虚拟机

高级语言的虚拟机



高级语言的虚拟机

- ◆ 不同平台不同的编译器
- ◆ 不同平台部署需要重新编译
- ◆ 编译结果不复用

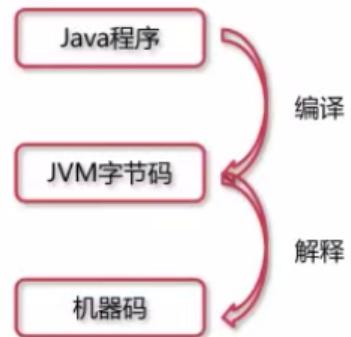


➤ 一次编译 随处运行

Java虚拟机与字节码

编译+解释

字节码 (英语: Bytecode) 通常指的是已经**经过编译**, 但与特定机器代码无关, 需要**解释器转译**后才能成为机器代码的中间代码。字节码通常不像源码一样可以让人阅读, 而是编码后的数值常量、引用、指令等构成的序列。



3.2.3 编译器原理

1. 编译器运行

编译器的运行过程



词法分析

◆ 保留关键字: int、float

◆ 运算符: +、-、*、/

◆ 变量: sum, result

字符流

词法分析

符号流

ID	名字	值
1	result	...
2	initial	...
3	rate	...

result = initial + rate * 60

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

语法分析

result = initial + rate * 60

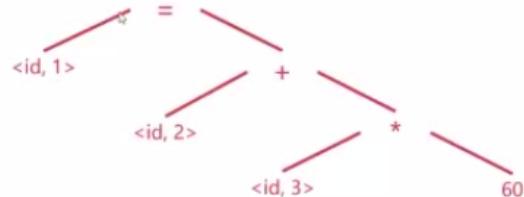
<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

◆ 构建语法树

符号流

词法分析

语法树



语义分析

◆ 构建语义树

◆ 类型检查

◆ 类型转换

中间代码生成

t1 = int(60)

t2 = id3 * t1

三地址代码

t3 = id2 + t2

id1 = t3

优化分析

t1 = int(60)

t2 = id3 * t1

t1 = id3 * int(60)

t3 = id2 + t2

id1 = id2 + t1

id1 = t3

代码生成

```
t1 = id3 * int(60)
```

```
id1 = id2 + t1
```

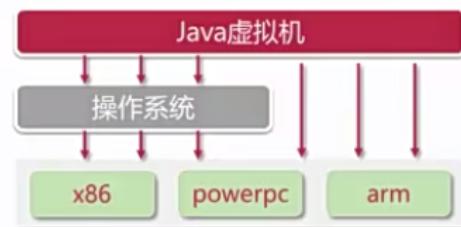
```
LD R2, id3  
MUL R2, R2, #60  
LD R1, id2  
ADD R1, R1, R2  
ST id1, R1
```

3.2.4 程序运行原理

1. CPU体系

CPU体系结构

- ◆ 二进制程序本质是一条一条的CPU 指令
- ◆ 不同CPU体系结构的指令集不同
- ◆ C/C++程序跨平台需要重新编译

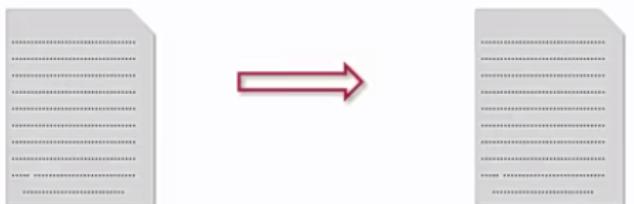


2. 程序运行过程

程序运行过程



- ◆ 以C/C++程序为例



程序运行过程：预编译

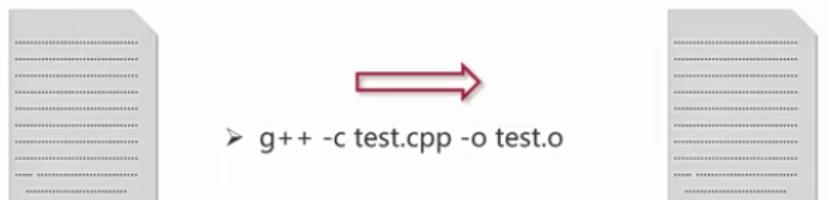


- ◆ 预编译主要是做一些代码文本的替换工作
- ◆ #define、#include、条件编译
- ◆ 代码注释

程序运行过程：编译



- ◆ 以C/C++程序为例



程序运行过程：汇编



将汇编代码转成机器码

编译器

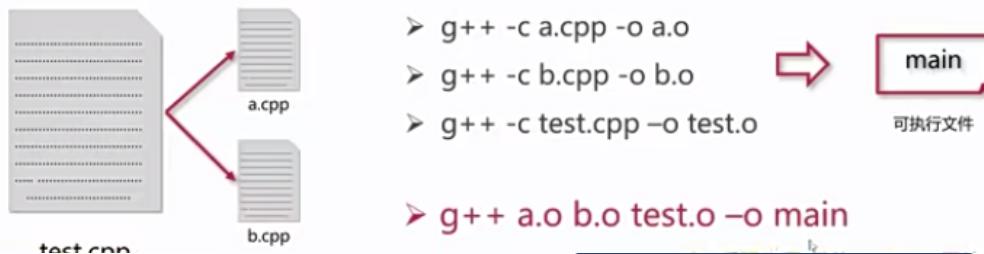
汇编器



编译器

程序运行过程：链接

- ◆ 目标文件仅仅是当前的源码文件编译成的二进制文件
- ◆ 并没有经过链接过程，是不能够执行的



程序运行过程：装载

可执行文件加载到内存运行

- ◆ 确定的进程入口地址
- ◆ 完整的进程空间

虚拟内存空间

段页式存储管理

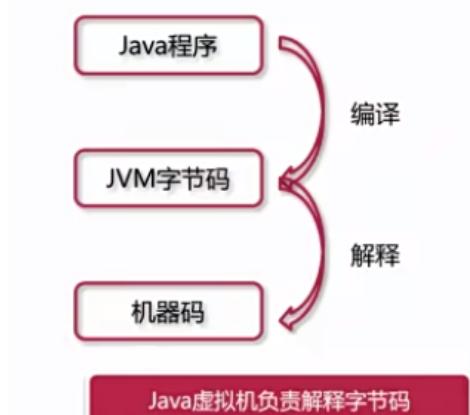


程序运行过程



3. JIT技术

JIT技术



JIT技术(Just In Time): 通常对于存在中间代码的运行系统(Java、Python等)，解释执行过程的效率不如传统本地代码的执行效率，在实现JIT的系统中，JIT可以在运行过程动态将中间字节码编译成本地代码，从而加快运行速度。

性能优化收益



编译消耗

3.2.5 链接方式

1. 动态/静态链接

动态链接/静态链接

库：库是写好的现有的，成熟的，可以复用的代码。现实中每个程序都要依赖很多基础的底层库，不可能每个人的代码都从零开始，因此库的存在意义非同寻常。

链接库：制作成通用格式的共享库，具备标准通用的加载接口，可以提供相同平台下不同程序使用的库。

动态/静态链接库：动态/静态特指链接库提供调用的不同形式，动态链接、静态链接各有优劣。

动态链接/静态链接



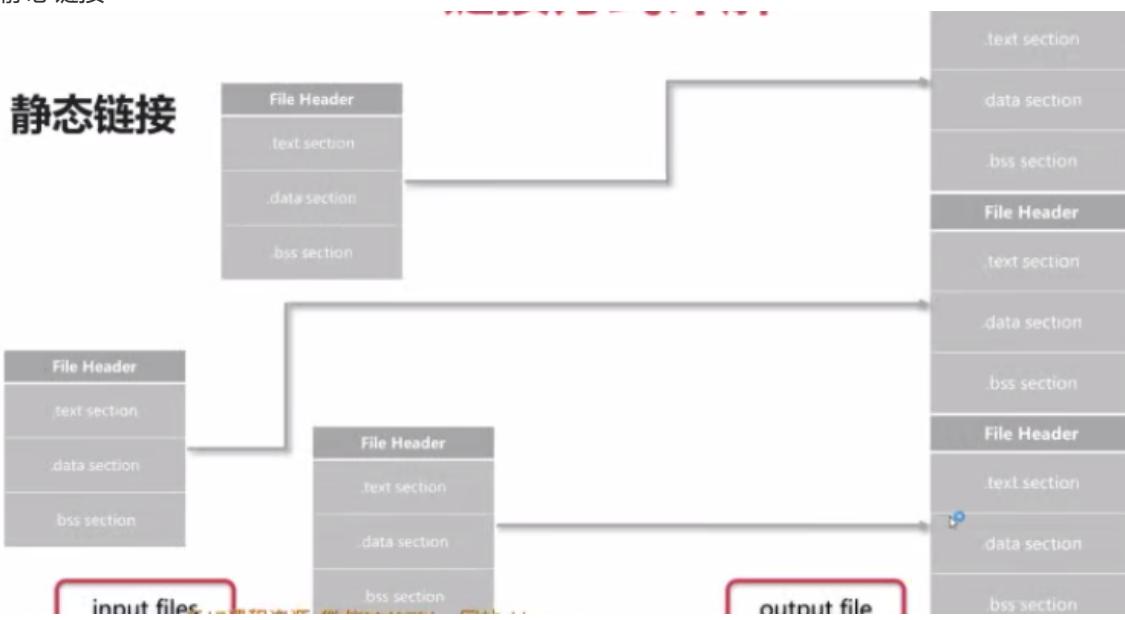
2. 目标文件

目标文件

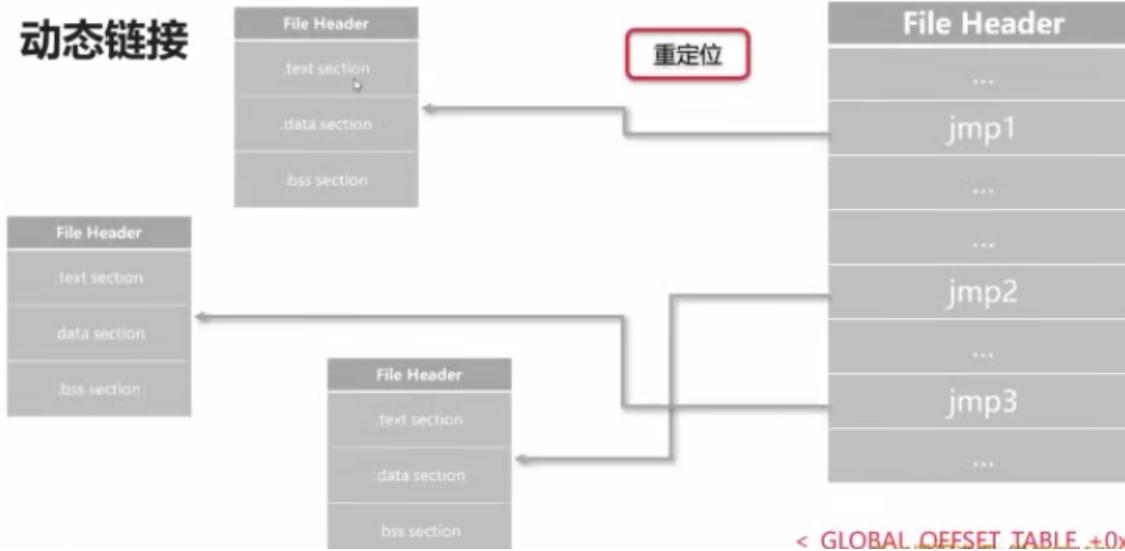


- ◆ .text(代码段)：程序执行代码
- ◆ .data(数据段)：已初始化全局变量
- ◆ .bss(bss段)：未初始化全局变量
- ◆ 堆、栈...

3. 静态链接



4. 动态链接



5. 动态链接与装载

动态链接与装载

可执行文件加载到内存运行

- ◆ 确定的进程入口地址
- ◆ 完整的进程空间

虚拟内存空间

段页式存储管理



- ◆ 静态链接目标：只需要装载执行文件即可
- ◆ 动态链接目标：还需要另外加载动态链接库

6. 两者对比

动态链接/静态链接

	动态链接	静态链接
内存空间	节省内存空间	浪费内存空间
编译时间	较短	较长
共享对象更新	便捷	麻烦
灵活性	较灵活	不灵活
启动速度	较慢	较快
寻址速度	较低	较高

四、算法与数据结构

4.1 链表、栈、队列与二叉树

4.1.1 时间/空间复杂度

1.

时间复杂度

```
num_list = [75, 67, 88, 91, 69]
max = 0
for num1 <- num_list:
    if max < num1:
        max = num1
print(max)
```

操作次数: $1 + n + n + n/2$

$O(n)$

```
num_list = [75, 67, 88, 91, 69]
result = []
for num1 <- num_list:
    for num2 <- num_list:
        if num2 > num1:
            sum <- num1 + num2
            result.add(sum)
print(result)
```

操作次数: $n * (n+n+n/2) = \frac{5}{2}n^2$

$O(n^2)$

2. 取去掉系数的最高阶

时间复杂度

复杂度	非正式术语	操作次数
$O(1)$	常数阶	1
$O(n)$	线性阶	$2n + 3$
$O(n^2)$	平方阶	$3n^2 + 2n + 1$
$O(\log n)$	对数阶	$5\log n + 20$
$O(n \log n)$	对数阶	$n \log n + 5n + 20$
$O(2^n)$	指数阶	2^n

空间复杂度

```
num_list = [75, 67, 88, 91, 69]
sum = 0
for num1 <- num_list:
    sum <= sum + num1
print(sum)
```

3.

```
num_list = [75, 67, 88, 91, 69]
result = []
for num <- num_list:
    r = num * num
    result.add(r)
print(result)
```

分配次数: 1

O(1)

因为前者sum的空间就一个，只不过是经常变值而已

分配次数: n

O(n)

4. 空间跟时间相互驳斥

判断某个数字是否存

在列表内 O(n):

[75, 67, 88, 91, 69, 78, 89, 31, 28, 97, 71, ...]

```
num_list = [75, 67, 88, 91, ...]
valid = 100
for num1 <- num_list:
    if valid == num1:
        return True
```

O(n)

O(1)

```
num_list = [75, 67, 88, 91, ...]
map = {}
map[75] = 1
...
map[69] = 1
if map.get(100) == 1:
    return True
```

O(1)

O(n)

4.1.2 链表

1. 定义

链表是什么？

◆ 链表是一种物理存储单元上非连续、非顺序的存储结构

◆ 链表的逻辑顺序是通过链表中的指针链接次序实现的

2. 链表

1. 数据域

2. 指针域

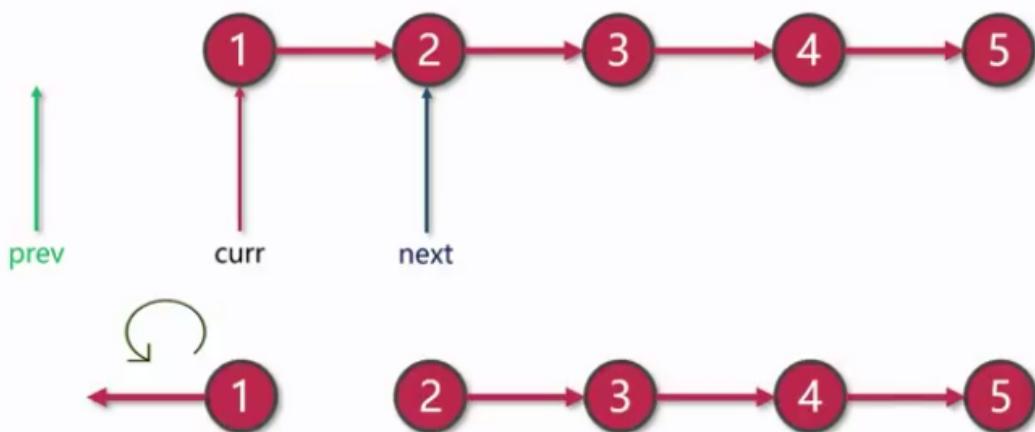
4.1.3 链表算法题

1. LeetCode 206.反转链表

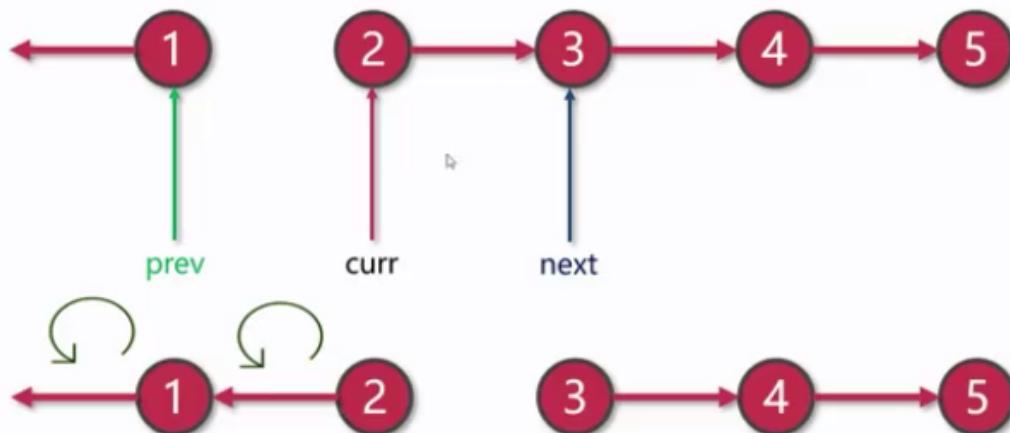
反转链表



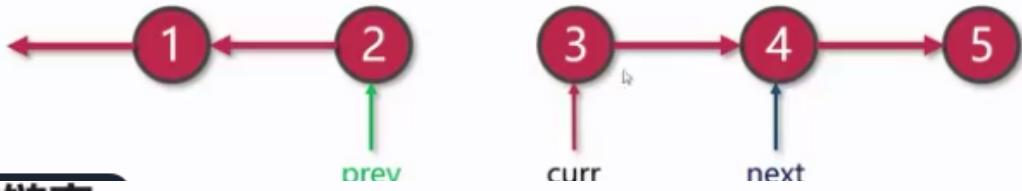
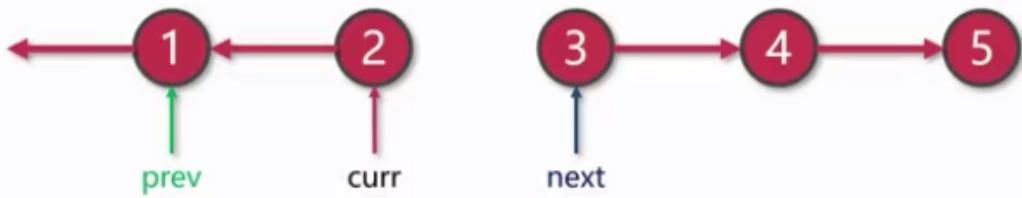
反转链表



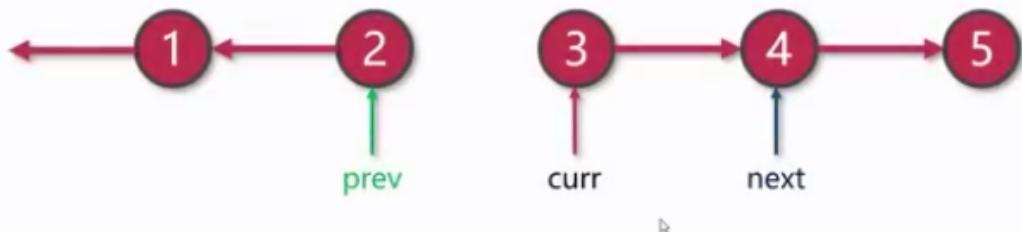
反转链表



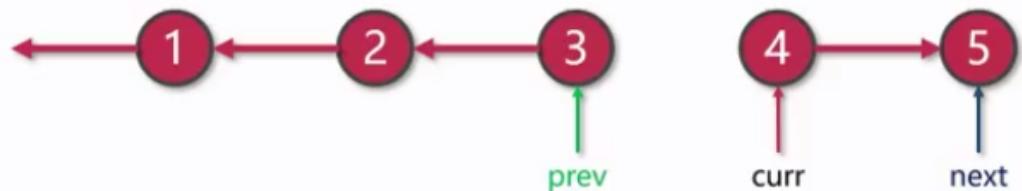
反转链表



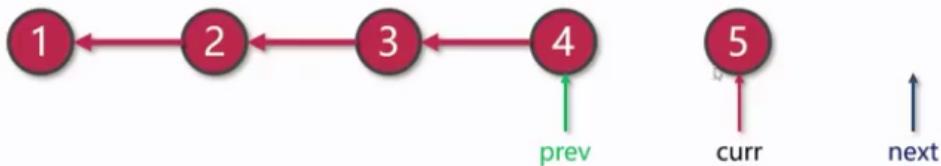
反转链表



反转链表



反转链表



最后一步已经跳出循环了，但是最后一个还没有逆序

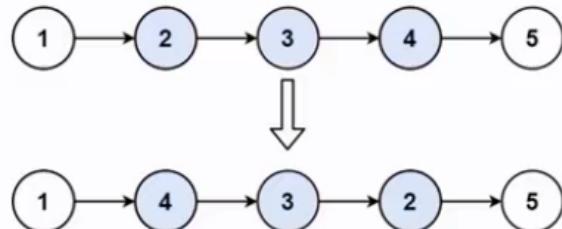
2. LeetCode 92

Leetcode 92 (反转链表 II)

给你单链表的头指针 head 和两个整数 left 和 right，其中 $left \leq right$ 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表。

输入: head = [1,2,3,4,5], left = 2, right = 4

输出: [1,4,3,2,5]



TODO

五、数据库

5.1 DB、表、视图、事务与函数

5.1.1 关系型数据库

关系数据库

关系型数据库模型是把复杂的数据结构归结为简单的二元关系（即二维表格形式）。在关系型数据库中，对数据的操作几乎全部建立在一个或多个关系表格上，通过对这些关联的表格分类、
1. 合并、连接或选取等运算来实现数据库的管理。

MySQL

SQLServer

SQLite

PostgreSQL

关系数据库

科目	老师	学分	课时	学生人数
语文	黄老师	4	60	80
英语	陈老师	3	45	95
数学	刘老师	6	80	70

名字	性别	毕业院校	职称
黄老师	M	华南师范大学	高级教师
陈老师	G	湖南师范大学	特级教师
刘老师	M	北京师范大学	副教授

名字	性别	出生年月	身高
Alice	M	1995-10-01	165
Bob	G	1994-06-05	176
Candy	M	1995-07-12	160

非关系数据库(NoSQL)

- ◆ NoSQL: Not Only SQL
- ◆ NoSQL数据库种类繁多、数据库结构简单
- ◆ 去掉关系数据库的关系型特性

Key-Value数据库

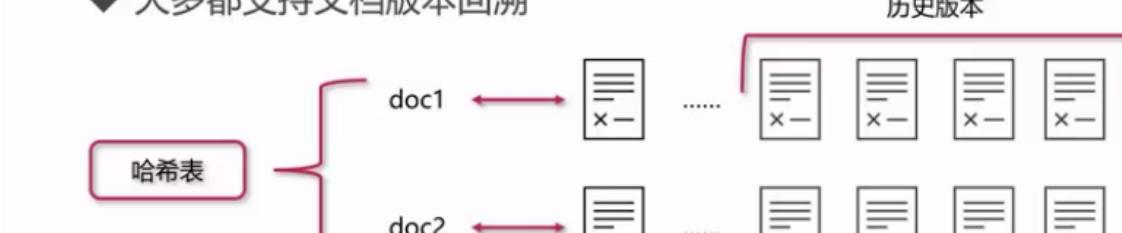
- ◆ 特定的键(key)指向指定的数据(Value)
- ◆ Value一般支持多种数据格式
- ◆ 通过指定Key检索Value速度非常快

Redis

Memcache

文档数据库

- ◆ K-V数据库的升级版，数据类型为结构化的文档
- ◆ 大多都支持文档版本回溯



5. 列存储数据库

列存储数据库

- ◆ 分列数据格式：对一个列的数据进行分组和存储
- ◆ 大多数查询并不会涉及表中的所有列
- ◆ 不适合数据量更新、删除频繁的场景

列存储数据库

科目	老师	学分	课时	学生人数
语文	黄老师	4	60	80
英语	陈老师	3	45	95
数学	刘老师	6	80	70

→

时间	设备1	设备2	设备3	设备4
2021-10-01	-	-	6.1221	10.2219
2021-10-02	9.1345	3	-	7.2238
2021-10-03	7.1231	6	8.1239	8.2491
2021-10-04	-	7	-	9.0

6. 图数据库

图数据库

- ◆ 应用图形理论存储实体之间的关系信息
- ◆ 典型的NoSQL数据库
- ◆ 在智能推荐领域有着非常广泛的应用

5.1.2 数据库设计、创建、维护

1. {DATABASE|SCHEMA}

◆ DATABASE: 数据库

两者同义

◆ SCHEMA: 模式

CREATE DATABASE

2. [create_option]: CHARACTER SET

◆ CHARACTER SET: 字符集

◆ show charset;

utf8mb4

CREATE DATABASE IF NOT EXISTS test_db_for_imoooc
CHARACTER SET = utf8mb4

3. [create_option]: COLLATE

◆ 对字符串类型字段的排序规则

◆ 国内比较常用的是utf8mb4_general_ci

5.1.3 数据表设计、创建、维护

1. 官方实例

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition, ...)
  [table_options]
  [partition_options]
```

[create_definition]:

- ◆ 定义表字段名字、定义表字段属性
- 2.
 - ◆ <column_name> <column_definition>

数据类型

是否为空

默认值

其他属性

[create_definition]:

列名	列定义
FCourseId	INTEGER NOT NULL AUTO_INCREMENT,
FCourseName	VARCHAR(255) NOT NULL DEFAULT "",
FTeacherId	INTEGER NOT NULL,
FCourseCredit	SMALLINT NOT NULL,
FCourseDuration	SMALLINT NOT NULL,
FCreateTime	DATETIME NOT NULL DEFAULT "",
FModifyTime	TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

[create_definition]:

- ◆ 定义主键、外键、其他索引

PRIMARY KEY (FCourseId),
FOREIGN KEY(FTeacherId) REFERENCES teacher(FId)
KEY(FCourseName)

3. [table_options]:

- ◆ 定义表级属性，如引擎、编码集、加密、最大行数等

ENGINE = InnoDB DEFAULT CHARSET = utf8;

4. [partition_options]:

- ◆ 指定表数据的分区规则

分区规则	描述
range分区	行数据基于属于一个给定连续区间的列值被放入分区
list分区	分区面向的是离散的值
key分区	根据MYSQL数据库提供的哈希函数来进行分区
hash分区	根据用户自定义的表达式的返回值来进行分区

实际使用较少

5.1.4 char、varchar和text

1. MySQL类型

数据类型	定义
数字类型(Numeric Data)	TINYINT, INTEGER, FLOAT, DOUBLE, DECIMAL, NUMERIC
日期类型(Date and Time Data)	DATE, TIME, DATETIME, TIMESTAMP, YEAR
字符串类型(String Data)	CHAR, VARCHAR, BINARY, VARBINARY, BLOB, TEXT, ENUM, SET
空间数据类型(Spatial Data)	-
JSON数据类型(JSON Data)	JSON

2. 数字类型

数字类型

数据类型	占用字节	定义
TINYINT	1	[-128, 127] 或者 [0, 255]
SMALLINT	2	[-32768, 32767] 或者 [0, 65535]
INTEGER	4	-
BIGINT	8	-
FLOAT	4	单精度浮点数
DOUBLE	8	双精度浮点数
DECIMAL	-	存储精确数值, DECIMAL(P, D), P: 精度, D: 小数点位数

- ◆ DECIMAL(24, 4): 表示存储小数点后4位，则整数部分最大20位。

DECIMAL: 存储精确数值，如货币数据、消费数据等

日期类型

3. 日期类型

数据类型	数据值
DATE	"0000-00-00"
TIME	"00:00:00"
DATETIME	"0000-00-00 00:00:00"
TIMESTAMP	"0000-00-00 00:00:00"
YEAR	0000

4. 字符串类型

字符串类型

数据类型	数据值
CHAR	固定长度, 最大255字节
VARCHAR	不固定长度, 最大65535字节
BLOB	对象存储, 最大65535字节
TINYTEXT	最大255字节
TEXT	最大65535字节
LONGTEXT	最大4294967295字节

字符串类型

◆ CHAR vs VARCHAR: VARCHAR可变长, CHAR占用固定存储空间

◆ VARCHAR vs TEXT: TEXT不允许默认值

- Text(mediumtext、longtext)支持比varchar更长的存储长度
- 给text字段建索引, 索引的占用空间较大
- 从名字上, 区分存储内容的区别
- 总的来说, 用中长用char, 只是使用varchar, 只是小用text

5. JSON

JSON类型

◆ JSON列中的内容会被自动校验, 不允许错误

◆ 存储在JSON列中的内容会被转换为允许快速读取元素的内部格式

◆ 优化器支持局部读写, 不用覆盖完整内容

5.1.5 数据库ACID

◆ 原子性 (Atomicity)

◆ 一致性 (Consistency)

1. 定义

◆ 隔离性 (Isolation)

◆ 持久性 (Durability)

2. 事务处理

事务处理

数据库事务主要用于维护数据库的完整性，通过事务，可以保证成批的MySQL操作要么完全执行，要么完全不执行。

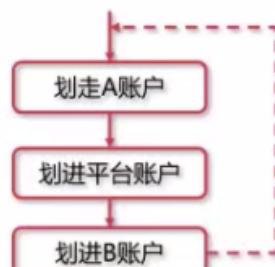
为保证事务 (transaction) 是正确可靠的，所必须具备的四个特性：原子性 (atomicity，或称不可分割性) 、一致性 (consistency) 、隔离性 (isolation，又称独立性) 、持久性 (durability) 。

3. 原子性

原子性 (Atomicity)

一个事务 (transaction) 中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚 (Rollback) 到事务开始前的状态，就像这个事务从来没有执行过一样。即，事务不可分割、不可约简。

交易数据：A账户划走X元，平台抽成20%，B账户增加Y元

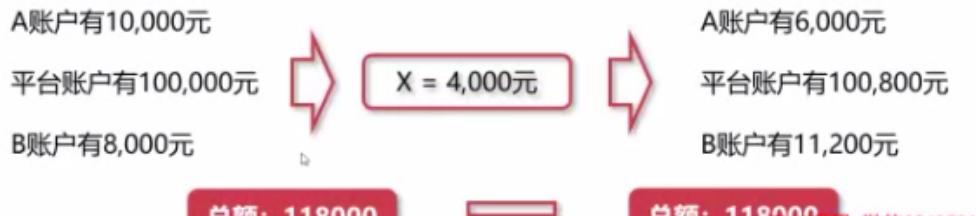


4. 一致性

一致性 (Consistency)

在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设约束、触发器、级联回滚等

交易数据：A账户划走X元，平台抽成20%，B账户增加Y元



5. 隔离性

隔离性 (Isolation)

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。



6. 持久性

持久性 (Durability)

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。



5.1.6 数据事务隔离级别

- ◆ 未提交读 (Read uncommitted)

- ◆ 提交读 (read committed)

1. 级别

- ◆ 可重复读 (repeatable read)

- ◆ 串行化 (Serializable)

2. 脏读、不可重复读、幻读

脏读、不可重复读和幻读

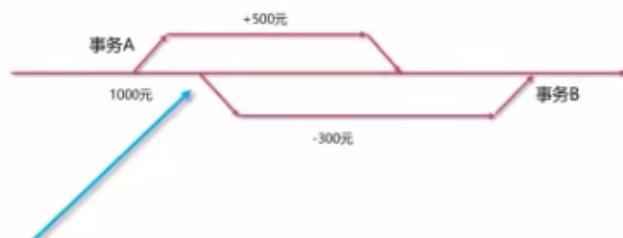
脏读：事务A读取了事务B操作未提交的数据，则事务A读到的是脏数据，此时称为脏读

不可重复读：事务A多次读取到同一份数据，事务B在事务A多次读取的过程中，对数据进行了更新并提交，导致事务A多次读取同一数据时，结果不一致。

幻读：事务A需要操作一份数据，并事先匹配成功，在实际进行操作时，却发现数据被事务B插入/删除了。即在一个事务操作里面发现了未被操作的数据。

3. 未提交读

未提交读



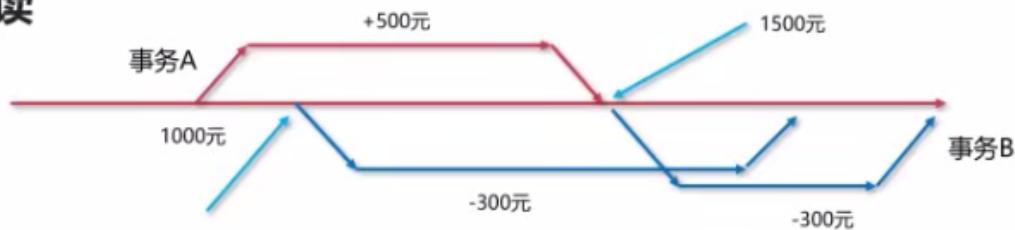
事务B执行时，读取到了事务A尚未提交的值：1500

脏读：事务A读取了事务B操作未提交的数据，则事务A读到的是脏数据，此时称为脏读

未提交读：即能够读取到事务未提交的数据，无法解决脏读、不可重复读、幻读的任何一种。

4. 提交读

提交读



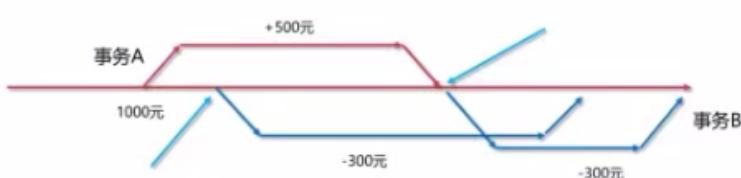
事务B执行时，能够读取那些已提交的数据，可以避免脏读。

不可重复读：事务B多次读取到同一份数据，事务A在事务B多次读取的过程中，对数据进行了更新并提交，导致事务B多次读取同一数据时，结果不一致。

多个事务操作一个数据

5. 可重复读

可重复读



不可重复读：事务A多次读取到同一份数据，事务B在事务A多次读取的过程中，对数据进行了更新并提交，导致事务A多次读取同一数据时，结果不一致。

无法解决幻读问题

在事务B结束之前，事务B对某一份数据多次读取得一致的结果

6. 串行读

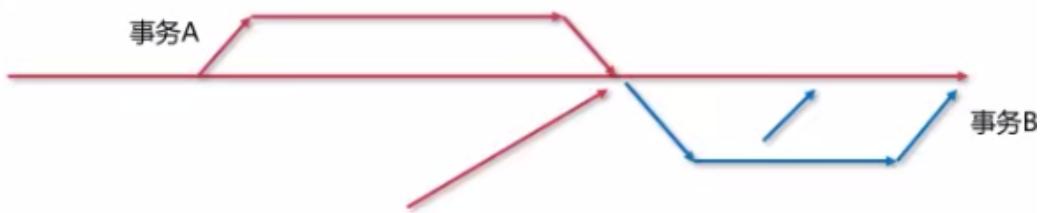
串行读

不可重复读：事务A多次读取到同一份数据，事务B在事务A多次读取的过程中，对数据进行了更新并提交，导致**事务A多次读取同一数据时，结果不一致。**

幻读：事务A需要操作一份数据，并事先匹配成功，在实际进行操作时，却发现数据被事务B插入/删除了。**即在一个事务操作里面发现了未被操作的数据。**

两者区别：不可重复读强调数据被更改、幻读强调数据被插入/删除。

串行读



串行读：不管多少事务，都必须执行完一个事务再执行另外一个事务，不存在并行执行事务的情况

彻底解决脏读、不可重复读、幻读的问题，牺牲了事务性能

7. 总结

	脏读可能性	不可重复读可能性	幻读可能性	加锁读
未提交读	是	是	是	否
提交读	否	是	是	否
可重复读	否	否	是	否
串行化	否	否	否	是

5.2 索引、性能和安全

5.2.1 MySQL索引

1. 数据库的功能

对于一个数据库而言，最普遍的查询就是两种：精确查询和范围查询

查询

精确查询是指通过一个具体的key找到对应的一条或者多条数据

范围查询是指查询key在某个范围内的所有数据

对于一个数据库而言，插入需要考虑两个问题：插入性能和查询性能

插入

插入性能是指插入数据的过程不能消耗太多的时间

查询性能是指插入的数据在被查询的时候，需要快速返回

2. 二叉搜索树、多叉树

二叉搜索树、多叉树

数组

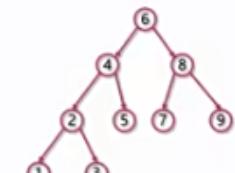
- ◆ 当数据量超过内存时，部分数据需要暂存硬盘

链表

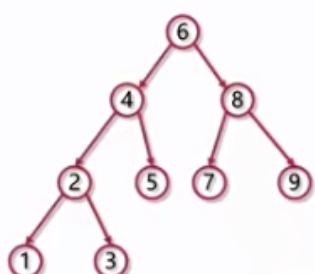
- ◆ 平均操作复杂度为 $O(n)$

树

二叉搜索树



二叉搜索树、多叉树



- ◆ 左子树上所有结点的值均小于它的根结点的值
- ◆ 右子树上所有结点的值均大于它的根结点的值

保证有序

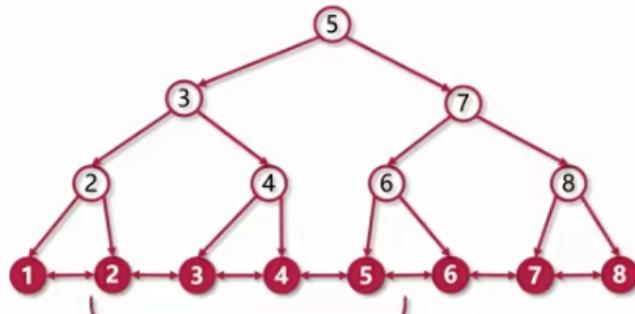
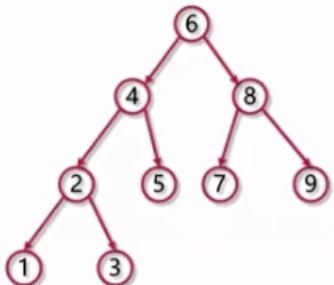
$O(log n)$

- ◆ 根节点、叶子节点都保存有数据
- ◆ 对于范围查询来说，不友好（需要不断从根节点遍历）

根节点不保存数据，只在叶子节点保存数据

叶子节点使用双向链表连接起来，即可快速范围查询

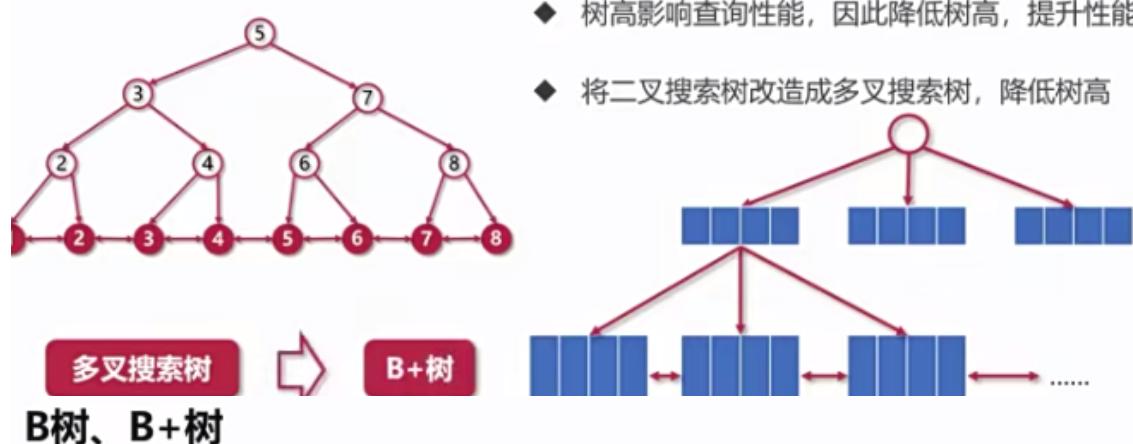
二叉搜索树、多叉树



范围查询

3. B、B+树

B树、B+树



B树、B+树

- ◆ 树高影响查询性能，因此降低树高，提升性能
- ◆ 将二叉搜索树改造成多叉搜索树，降低树高

- ◆ B树在叶子节点、根节点都保存数据，不利于范围查询
- ◆ B+树根节点不存储数据，叶子使用双向链表连接，适合范围查询
- ◆ 在插入数据、精确查询时，两者性能相近

为什么使用B+树作为MySQL索引？

- ◆ 线性数据结构

4. 为什么是B+树

- ◆ 二叉树、二叉搜索树、B树

- ◆ 范围查询：B+树

5.2.2 聚簇/非聚簇索引

1. 索引

索引

在关系数据库中，索引是一种单独的、物理的对数据库表中一列或多列的值进行排序的一种存储结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。索引的作用相当于图书的目录，可以根据目录中的页码快速找到所需的内容。



2. 存储区层次

存储器层次结构



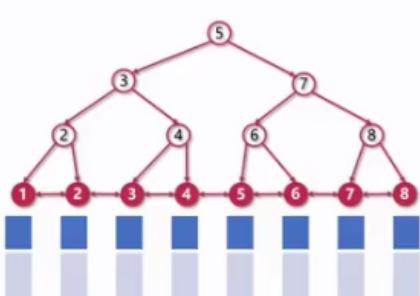
- ◆ 聚簇索引：将数据存储与索引放在一块，找到索引就找到了数据
- ◆ 非聚簇索引：将数据存储与索引分开存储，索引指向数据内存空间
- ◆ 非聚簇索引通常也被称为**二级索引**

3. 聚簇索引

聚簇索引

- ◆ 一个数据表只有一个聚簇索引（想想为什么？）
- ◆ 默认情况下聚簇索引是主键
- ◆ 聚簇索引性能最好且具有唯一性，需要慎重选择字段作为聚簇索引

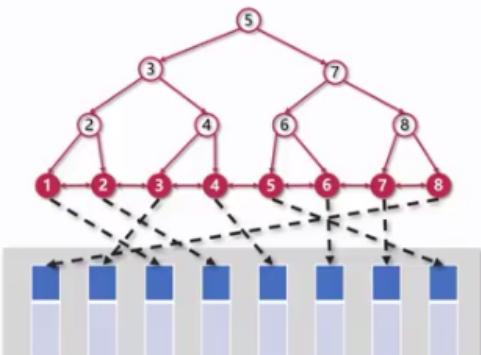
聚簇索引



- ◆ InnoDB使用的是聚簇索引
- ◆ 将主键组织到一棵B+树中，而行数据就存储在叶子节点
- ◆ 最终目的：相同结果集的情况下，减少逻辑IO

4. 非聚簇索引

非聚簇索引



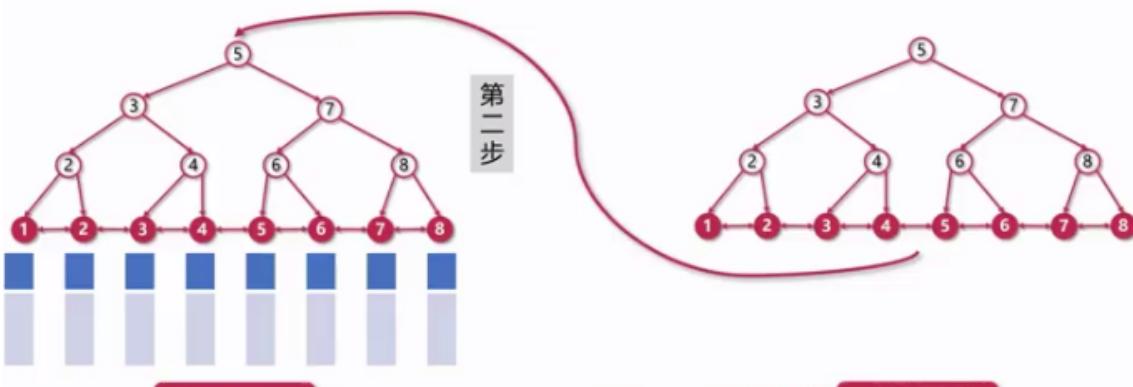
- ◆ 一个数据表可以有多个非聚簇索引
- ◆ MyISAM使用的是非聚簇索引
- ◆ 非聚簇索引和聚簇索引的B+树看上去并无不同
- ◆ 节点完全一致，只是存储的内容不一致

5. InnoDB的聚簇与非聚簇

InnoDB中的聚簇索引与非聚簇索引

- ◆ 一个数据表只有一个聚簇索引
- ◆ 一个数据表可以有多个非聚簇索引
- ◆ 为什么非聚簇索引被称为二级索引？

InnoDB中的聚簇索引与非聚簇索引



InnoDB中的聚簇索引与非聚簇索引

- ◆ 先在非聚簇辅助索引检索到对应的记录，获得相应主键
- ◆ 根据主键在聚簇索引再进行一次检索操作
- ◆ 对非聚簇索引的数据列检索需要两次检索索引

6. 聚簇优点

聚簇索引的优点

- ◆ 行数据与叶子节点一起存储，结合内存页结构，检索速度快
- ◆ 聚簇索引适合排序场合、范围查询场合
- ◆ 维护聚簇索引不需要对行数据进行额外的管理操作

聚簇索引的注意事项

- ◆ 不建议使用长字符串(UUID)作为主键索引
- ◆ 稀疏数据的列不适合建立聚簇索引
- ◆ 频繁更新的列不适合建立索引

7. 注意事项

5.2.3 联合索引原理

联合索引

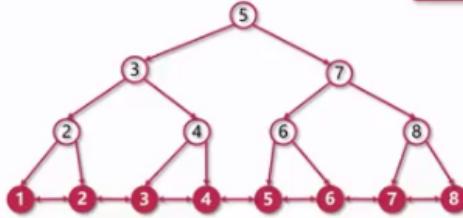
联合索引，也称为复合索引，即是由多个字段组成的一个索引。

```
KEY `idx_y_z`(`f_y`, `f_z`);
```

1. 定义

```
KEY `idx_x_y_z`(`f_x`, `f_y`, `f_z`);
```

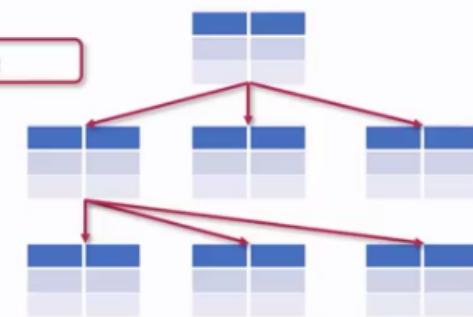
联合索引



联合索引每个节点包含多个字段的值

idx_y_z

idx_x_y_z



- ◆ f_x: 100
- ◆ f_y: 100

100 * 100 * 100

2. 联合索引使用与Where顺序无关

联合索引的作用时机

◆ 联合索引使用遵循**最左匹配原则**

◆ where条件必须使用联合索引的第一个字段

◆ 联合索引与where条件的顺序无关，只和字段有关

5.2.4 MVCC原理

TODO

5.2.5 MySQL日志类型

5.2.6 MySQL权限维护

权限粒度

- ◆ 全局层级(global level)
- 1. 权限粒度 ◆ 数据库层级(database level)
- ◆ 数据库对象层级(table level)
- ◆ 列层级(column level)

权限粒度

全局层级(global level): 作用于整个mysql实例级别，包括操作数据库、服务启停、配置更改等，其权限信息存储在mysql.user表。

数据库层级(database level): 作用于指定的数据库的所有对象中，包括触发器、视图、表、索引等，权限信息存储在mysql.db表。

数据库对象层级(table level): 作用于指定数据库的指定表，包括对表结构的维护、表级锁等，权限信息存储在mysql.tables_priv表。

列层级(column level): 作用于指定数据库指定表的列，包括对表特定列的增删改查、列定义修改等，权限信息存储在mysql.columns_priv表。

2. 创建用户并授权

创建用户并授权

```
CREATE USER <username>@<hostname> IDENTIFIED BY <password>;
```

- ◆ Username: 创建的用户名 e.g: dddq
- ◆ Hostname: 允许登录的主机IP, localhost表示本机, “%” 表示任意主机均可
- ◆ Password: 用户的登录密码

创建用户的用户(root)需要有全局级别的权限

创建用户并授权

```
grant all privileges on *.* to <username>@<hostname> identified by <password>;
```

- ◆ all privileges: 授权所有权限
- ◆ *.*: 第一个 “*” 表示任意数据库, 第二个 “*” 表示任意表

```
grant SELECT, INSERT on test.* to <username>@<hostname> identified by <password>;
```

```
grant SELECT, INSERT on test.t_student to <username>@<hostname> identified by <password>;
```

创建用户并授权

```
grant all privileges on *.* to <username>@<hostname> identified by <password>;
```

- ◆ all privileges: 授权所有权限
- ◆ *.*: 第一个 “*” 表示任意数据库, 第二个 “*” 表示任意表

```
grant SELECT, INSERT on test.* to <username>@<hostname> identified by <password>;
```

```
grant SELECT, INSERT on test.t_student to <username>@<hostname> identified by <password>;
```

刷新权限: ➤ flush privileges;

3. 移除授权

移除授权

使用关键字: revoke, 其余和授权语句grant语法一样。

```
revoke all privileges on *.* to <username>@<hostname>;
```

```
revoke SELECT, INSERT on test.* to <username>@<hostname>;
```

```
revoke SELECT, INSERT on test.t_student to <username>@<hostname>;
```

4. 用户更改

用户更改

删除用户: drop user <username>@<hostname>;

更改用户: rename user <username>@<hostname> to <new username>@<hostname>;

更改用户密码:

set password for <username>@<hostname> = password(<new password>;

alter user <username>@<hostname> identified by <new password>;

mysqladmin -u <username> -h <hostname> password <new password>;

六、编程能力

6.1 面向对象

6.1.1 面向对象思想

1. 概念

对象、属性、行为

对象是对软件系统模拟的真实世界的抽象，所有真实世界的实体和概念都可以抽象为对象，例如：在学校这个系统中，学生、成绩单、教师、课程、教室等；在生物领域，肉食动物、草食动物，鸟类等。

每个对象都是唯一的，对象的唯一性来自他们在真实世界中的唯一性，因为在真实世界中，他们都有特定的属于自己的部分，比如学生，有不同的名字、背景、家庭、年龄，这些描述一个对象都称为属性。

行为是对象唯一性的另外一种呈现，如果属性描述的是对象静态唯一性的话，行为描述的则是对象的动态唯一性，对象通过其行为对外提供功能。

状态、方法、实现

对象具有状态，状态是指对象某一个瞬间对对象各个属性的取值；因为对象的行为可以改变对象的状态，比如学生如果认真学习，那么未来的成绩会提升；所以对象具有状态，有时候，对象的状态在一个系统中能起到决定性的作用，当然，很多bug也是因为状态的变更带来的。

对象的行为包括具有的功能和具体的实现，在建立对象模型的阶段，则仅仅关注对象有什么样的功能；但不需要考虑如何实现，但是在实际实例化时，必须通过具体的方法来实现具体的行为，否则类是无法对外提供服务的。

消息、服务

软件系统的复杂功能是通过各种对象协同工作来共同完成的，类和类之间是一种协作的关系；而在设计的时候，每个类的功能又是相对固定的，类具备的是特定的功能，所以在类之间的协作关系中，必然存在通信行为。

类之间通过消息相互协作，子系统之间通过消息相互服务，系统对外通过消息提供服务。就实现而言，类提供的服务是由对象的方法来实现的，所以类之间的协作，一般通过调用对象方法来实现；子系统、系统之间的通信，则复杂一些，如消息队列、远程调用等，但本质上都是消息的传递。

封装、透明

- 封装：即隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别；将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体（类）。



封装、透明

封装指的是隐藏对象的属性和实现细节，仅仅对外提供公开的接口，封装能够为软件系统带来诸多优点：

- 便于使用者正确、方便地理解和使用系统，防止错误修改属性
- 有助于建立各个系统之间的松耦合关系，提高系统的独立性
- 提高软件的可重用性，每个系统都是一个独立的整体，可以在多个环境得到重用

封装、透明

- 把尽可能多的东西隐藏起来，对外提供简洁的接口
- 把所有属性都隐藏起来，避免外部调用者错误修改属性
- 合理使用public、protect、private
- 最小化的用户窗口，使用者知道的越少越好

继承、扩展、覆盖

- 父类和子类之间都存在继承和扩展关系
- 子类继承父类的属性和方法，同时，子类也扩展父类的功能
- 当父类功能无法满足子类的需求时，子类可以对父类功能进行覆盖
- 扩展、覆盖都发生在继承的生态中，但继承并不是万能的，继承也可能带来很多问题

继承与组合

- 组合是一种用多个简单子系统组装出大型复杂系统的有效手段
- 计算机（键盘、主机、显示器、鼠标）主机（CPU、内存、硬盘、网卡）
- 当一个系统由多个部分、且多个部分并无共性时，更应该使用组合
- 当一个系统需要扩展功能时，为避免继承树过大，应该优先考虑使用组合

6.1.2 接口

1. 概念

接口的概念和基本特征

- 接口的成员变量默认都是public、static、final的，必须被显式初始化
- 接口的方法默认都是public、abstract类型的
- 接口没有构造方法，不能被实例化，在接口中定义构造方法时invalid的
- 接口不能实现(implement)另一个接口，但它可以继承多个其他接口

接口的概念和基本特征

- 当类实现某个接口时，必须实现该接口的所有抽象方法，否则该类应定义为抽象类
- 不允许创建接口的实例，但是允许定义接口类型的引用变量（向上转型）

```
public interface Engine {}
```

```
Engine engine = new Engine()
```

```
public class PetrolEngine {}
```

```
Engine engine = new PetrolEngine()
```

抽象类与接口的区别与联系

```
public abstract class PetrolEngine {}
```

```
public interface Engine {}
```

- 抽象类和接口都位于继承树的上层，都代表着系统的抽象层

- 抽象类和接口都不能被实例化，但是可以当作引用变量的定义

- 都能包含抽象方法，并不必提供具体实现

抽象类与接口的区别与联系

- 接口中的成员变量只能是public、static和final类型的，但在抽象类中可以定义各种类型的实例变量和静态变量；这是抽象类的优势，更加灵活
- 一个类只能继承一个直接的父类，这个父类可能是抽象类，但是一个类可以实现多个接口；这是接口的优势所在，更加符合抽象的定义

```
public class PetrolEngine implement Engine{}  
public class BYDPetrolEngine extends PetrolEngine{}
```

```
public class BYDPetrolEngine implement Engine, BYDTech{}
```

抽象类与接口的区别与联系

- 接口是系统中最高层次的抽象模型，用接口作为系统与外界交互的窗口，站在外界使用者的角度，接口必须向使用者承诺系统能够提供哪些服务；站在系统设计者的角度，接口指定了系统必须实现哪些服务。
- 由于外部使用者依赖系统接口，所以接口一旦指定，就不允许随意修改，否则会对系统内部和外部调用者都造成严重的影响。
- 抽象类作为实现类和接口的中间类，既保有了接口的规范性，也实现了部分接口不能实现的功能，为实现类提供更多的便捷

6.1.3 类的实现与继承

1. 概念

方法重载(@Overload)与方法覆盖(@Override)

- 通常情况下，类的同一个函数（功能）有多种实现方式，到底使用哪种方式取决于给定的参数，而为每种实现方式定义一个函数名字显然不现实，这个时候，就需要进行方法重载
- 重载方法必须满足以下条件：
 - 方法名相同
 - 方法的参数类型、个数、顺序至少有一项不同（即是方法的输入不同）
 - 方法的返回类型可以不同
 - 方法的修饰符可以不同

方法重载(@Overload)与方法覆盖(@Override)

```
imooc  
public class Bird {  
    public void eat() {  
        System.out.println("this is eat1");  
    }  
    public void eat() {  
        System.out.println("this is eat2");  
    }  
}
```

```
public class Animal {  
    public void eat() {  
        ....  
    }  
    public void eat(string food) {  
        ....  
    }  
}
```

方法重载(@Overload)与方法覆盖(@Override)

- 在继承的场景中，如果子类需要实现一个函数，这个函数的名称、返回类型、参数签名等都和父类的函数完全匹配的时候（通常情况下是父类实现的这个函数无法满足子类的需求），这种情况成为方法覆盖。

```
public class Animal {  
    public void eat(String food, int type) {  
        System.out.println(  
            "eat in parent class");  
    }  
}  
  
public class Bird extends Animal{  
    @Override  
    public void eat(String food, int type) {  
        System.out.println(  
            "bird eating food");  
    }  
}
```

方法重写如上，二者区别

方法重载(@Overload)与方法覆盖(@Override)

- 方法覆盖和方法重载都要求方法名字相同
- 方法覆盖要求方法定义（参数签名）完全一致，方法重载要求参数签名不一致
- 方法覆盖要求方法返回类型必须一致，而方法重载对此不做限制
- 方法覆盖只能用于子类覆盖父类的方法，方法重载用于同一个类的所有方法
- 父类的方法只能在子类中被覆盖一次，但是一个方法可以在一个类中被重载多次

2. super、this

super、this

- super、this关键字都可以用来覆盖Java的默认作用域，使得被屏蔽的方法或参数变得在当前可用
- 在一个方法内，当局部变量和类的成员变量同名时，或者局部变量和父类的成员变量同名时，按照变量的作用域规则，只有局部变量在当前方法可见
- 当子类的某一个方法覆盖父类的方法时，在子类的范围内，父类的方法不可见
- 当子类中定义了和父类同名的成员变量时，在子类的范围内，父类的成员变量不可见

super、this

```
public class Animal {  
    protected int type;  
    public void eat(int mode, int type) {}  
}  
  
public class Bird extends Animal{  
    public int type;  
    public void eat(int mode, int type) {  
        super.eat();  
        System.out.println(this.type);  
    }  
}
```

6.1.4 继承的使用原则

1. 概念

继承的使用原则

继承是提高程序代码可重用性、以及提高系统的可扩展性的有效手段。但是如果在使用继承的时候不注意规范，也可能削弱系统的可扩展性以及可维护性。

- ◆ 继承树的层次不要太多，尽量保持在2-3层
 - 如果继承树的层次太多，则系统的对象结构模型过于复杂，难以理解，增加开发的难度，比如对于子类继承过来的属性，难以找出具体的定义位置，函数覆盖和函数重载时，难以确定具体实现的类
 - 影响系统的可扩展性，继承树的层次越多，在继承树上继承一个新类时，需要创建的类就越多

继承的使用原则

- ◆ 继承树的最上层应该为抽象层（接口而非抽象类或者实现类）
 - 当一个系统使用一棵继承树上的类时，可以直接引用继承树的上层类型，提高两个系统之间的松耦合。
 - 抽象层能够作为一整棵继承树对外提供服务



继承的使用原则

- ◆ 继承关系最大的问题：打破了封装
 - 就封装的概念而言，一个类是属性和行为的集成，在一个类中完成所有操作的闭环，继承的最大问题在于打破了这种闭环关系，在子类中使用或操作了父类的属性或者行为，打破了类本来的封装属性
- ◆ 精心设计用于被继承的类
 - 如果一个类或者接口是需要被继承的，那么在进行设计的时候，应该尽可能考虑通用性，避免一些特有的方法或属性

6.2 设计模式

6.2.1 单例模式

1. 为什么需要单例模式

为什么需要单例模式

对于系统来说，某些类并不需要创建多个实例，重复初始化浪费资源。

要不然需

Windows任务管理器

要同步全部内容

为什么需要单例模式

- ◆ 多个Windows任务管理器内容完全一致
- ◆ 如果多个窗口内容不一致，则问题更加严重
- ◆ Windows任务管理器的功能决定了它只能单例呈现

为什么需要单例模式

- ◆ 系统只需要一个实例对象，要求全局唯一
- ◆ 类的初始化资源消耗太大而只允许创建一个对象
- ◆ 实例只允许使用一个公共访问点，不能通过其他途径访问该实例

单例模式的实现方式

- ◆ 静态变量：static

2. 实现方式

- ◆ 静态对象获取方法
- ◆ 私有构造函数

3. 多线程安全问题

单例模式的实现方式——多线程安全问题

- ◆ 多线程调用getInstance函数
- ◆ 私有变量重复初始化
- ◆ 实现单例模式需要注意多线程的安全问题

4. 懒汉模式

单例模式的实现方式——懒汉模式

- ◆ 懒汉模式是指等到需要该对象时才初始化单例对象
- ◆ 使用volatile修饰私有变量
- ◆ 使用synchronized修饰初始化代码

单例模式的实现方式——饿汉模式

5. 饿汉模式

- ◆ 饿汉模式是指提前初始化单例对象
- ◆ 相比懒汉模式，实现过程更加简单

6.2.2 建造者模式

1. 为什么需要建造者模式

为什么需要建造者模式

◆ 可选参数过多的类进行初始化的时候需要很多构造函数

◆ 这种类不管是在实现还是初始化都比较麻烦

➤ 当一个类的构造函数参数个数超过4个，而且这些参数有些是可选的参数，考虑使用建造者模式。

2. 突破了数量、顺序的限制

建造者模式的优劣

◆ 建造者相对独立，易于扩展，不受被建造类的影响

◆ 建造者使用过程控制粒度更细，便于控制风险

◆ 实现过程复杂

6.2.3 适配器模式

1. 为什么需要适配器

为什么需要适配器模式

定义：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。

需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容。

如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决这些问题。

两个类之间不能直接联系在一起，可以做一个适配器承上启下。比如插座跟手机之间不能直接充电，不可能直接给插座或者手机外接一个充电线，充电器就是中间的适配器

2. 适配器优劣

适配器模式的优劣

- ◆ 客户端通过适配器可以透明地调用目标接口
- ◆ 复用了现存的类，最小程度地修改原有代码而重用现有的类
- ◆ 将目标类和适配者类解耦，解决接口不一致的问题

适配器模式的优劣

- ◆ 增加系统的复杂性
- ◆ 增加代码阅读难度，降低代码可读性

6.2.4 装饰器模式

1. 为什么需要装饰器

为什么需要装饰器模式

定义：指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式。

在软件开发过程中，有时想用一些现存的组件。

这些组件可能只是完成了一些核心功能。但在不改变其结构的情况下，可以动态地扩展其功能。所有这些都可以采用装饰器模式来实现。

2. 装饰器优劣

装饰器模式的优劣

- ◆ 装饰器是继承的有力补充，比继承灵活
- ◆ 不改变原有对象的情况下动态的给一个对象扩展功能，即插即用
- ◆ 通过使用不用装饰类及这些装饰类的排列组合，可以实现不同效果

6.2.5 代理模式

1. 为什么需要代理模式

为什么需要代理模式

定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。

一个客户不能或者不想直接访问另一个对象，这时需要找一个中介帮忙完成某项任务，这个中介就是代理对象。

例如，购买火车票不一定要去火车站买，可以通过 12306 网站或者去火车票代售点买。

比如有五台服务器，那么就可以用一台代理控制五台机器，坐负载均衡等等事情

代理模式的优劣

- ◆ 在客户端与目标对象之间起到一个中介作用和保护目标对象的作用
- ◆ 代理对象可以扩展目标对象的功能
- ◆ 在一定程度上降低了系统的耦合度，增加了程序的可扩展性

代理模式的优劣

- ◆ 代理模式会造成系统设计中类的数量增加
- ◆ 增加一个代理对象会造成请求处理速度变慢
- ◆ 增加了系统的复杂度

6.2.6 设计模式原则

1. 开闭原则

开闭原则

当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

- ◆ 对软件测试的影响
- ◆ 提高代码的可复用性
- ◆ 提高代码的可维护性

2. 里氏替换原则

里氏替换原则

里氏替换原则通俗来讲就是：子类可以扩展父类的功能，但不能改变父类原有的功能。

- ◆ 子类可以实现父类的抽象方法，但不要覆盖父类的非抽象方法
- ◆ 子类可以实现子类特有的方法
- ◆ 子类重载父类方法时，方法的输入一定要比父类方法更宽松
- ◆ 子类实现父类方法时，方法的输出一定要比父类方法更加严格

3. 依赖倒置原则

依赖倒置原则

依赖倒置原则的原始定义为：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象，要面向接口编程，不要面向实现编程。

- ◆ 依赖倒置原则可以降低类之间的耦合性
- ◆ 依赖倒置原则可以提高系统的稳定性
- ◆ 依赖倒置原则可以减少并行开发引起的风险
- ◆ 依赖倒置原则可以提高代码的可读性和可维护性

4. 单一职责原则

单一职责原则

单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分。

- ◆ 降低类的复杂度
- ◆ 提高类的可读性
- ◆ 提高系统的可维护性
- ◆ 变更引起的风险降低

5. 接口隔离原则

接口隔离原则

“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法。

- ◆ 提高系统的灵活性和可维护性
- ◆ 提高了系统的内聚性
- ◆ 保证系统的稳定性
- ◆ ↓
◆ 减少项目工程中的代码冗余

6. 迪米特原则

迪米特法则

↓

如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

- ◆ 降低了类之间的耦合度，提高了模块的相对独立性。
- ◆ 提高了类的可复用率和系统的扩展性。
- ◆ 过度使用迪米特法则会使系统产生大量的中介类
- ◆ 增加系统的复杂性，降低模块之间的通信效率

7. 合成复用原则

合成复用原则

合成复用原则要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。

- ◆ 继承复用破坏了类的封装性。
- ◆ 继承使得父类与子类的耦合度过高。
- ◆ 组合复用维持了类的封装性，新旧类之间耦合度低。
- ◆ 组合复用的灵活性高。

七、
