# 数据库

## 增删改查基本语句使用

create database mytest;

use mytest;

create table score(

name varchar(20),

id int primary key,

class varchar(10),

math int

);

insert into score values(‘a’, 1, ‘a1’, 90);

insert into score values(‘b’, 2, ‘a1’, 92);

update score set class = ‘a2’ where id = 2;

delete from score where class = ‘a2’;

select math from score where id = 1;

## 建表约束

alter table score drop primary key;

alter table score add primary key(id);

alter table score modify id int primary key;

## 三范式

## 查询语句及其顺序

语句：select distinct … from … join … on… where… group by … having… order by… limit…

执行：from… join on… where… group by … having… select from order by limit

其他：all()大于全部，any()大于其中一个

## 1.5 delete drop truncate区别，如何恢复

1）DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录。在日志中保存以便进行回滚操作。

DELETE FROM 表名 WHERE 条件表达式

2）TRUNCATE TABLE 则一次性地从表中删除所有的数据，并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

TRUNCATE 只能对TABLE；DELETE可以是table和view

3）DROP则删除整个表（结构和数据）。 TRUNCATE 和DELETE只删除数据。

恢复：对于delete，找到日志中误删的起始和终点，反写，将delete用insert替换；对于drop：找操作日志binlog，全部重新执行一遍，drop操作前的操作为结束点。（上次备份的数据库+增量日志来恢复）

**Mysql日志：**

重做日志（redo log）、回滚日志(undo log)、二进制日志(binlog)、错误日志(errorlog)、慢查询日志(slow query log)、一般查询日志(general log)

重做日志：作用：为了确保事务的持久性。redo日志记录事务执行后的状态，用来恢复未写入数据文件的已成功事务更新的数据。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的持久性这一特性。事务开始之前产生redo log, 脏页写入磁盘后被覆盖。

回滚日志：作用：保证数据的原子性，保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC）。事务开始之前，将当前的版本生成undo log，当事务提交之后，undo log并不能立马被删除，而是放入待清理的链表，由purge线程判断是否由其他事务在使用undo段中表的上一个事务之前的版本信息，决定是否可以清理undo log的日志空间。

二进制日志：作用：用于复制，在主从复制中，从库利用主库上的binlog进行重播，实现主从同步。用于数据库的基于时间点的还原。内容：包括了执行的sql语句（增删改）反向的信息，也就意味着delete对应着delete本身和其反向的insert；update对应着update执行前后的版本的信息；insert对应着delete和insert本身的信息。事务提交时记录到binlog中，过期时间配置的天数后自动删除。

错误日志：记录mysql启动和暂停，以及运行中发生的错误相关信息。（默认关闭）

普通查询日志：记录了服务器接收到的每一个查询或是命令，无论这些查询或是命令是否正确甚至是否包含语法错误，general log 都会将其记录下来（开销较大，默认关闭）

慢查询日志：记录执行时间过长和没有使用索引的查询语句

## 1.6 join

## 1.7 事务

事务特征、隔离级别

原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户A和用户B两者的钱加起来一共是5000，那么不管A和B之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是5000，这就是事务的一致性。

## 1.8 引擎：MyISAM、InnoDB

## 1.9 索引

定义、分类、优势、劣势、索引如何更新、哪些需要建索引、哪些不需要、

覆盖索引：包含所有满足查询需要的数据的索引，查询列要被所建的索引覆盖

## 1.10 聚集索引和非聚集索引

聚集索引：聚集索引中键值的逻辑顺序决定了表中相应行的物理存储顺序，检索效率比非聚集索引高，但对数据更新影响较大，一个表中只能拥有一个聚集索引 （理解：字典中的拼音）

非聚集索引：索引的逻辑顺序与磁盘上行的物理存储顺序不同，一个表中可以拥有多个非聚集索引。（理解：字典中的部首）

聚簇索引的叶子节点是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针

## 1.11 查看执行计划

explain+查询语句

执行计划详细信息

## 1.12 慢日志分析，show profiles 数据库优化

## 1.13 索引失效

## 1.14 锁

概念、分类、加锁操作、如何锁定一行、间隙锁、查看冲突情况

可重复读会加什么锁？（next-key锁）

## 1.15 mysql底层：b+树，连接：基于TCP

## 1.16 varchar和char区别

## 1.17 sql注入攻击

通过sql语句攻击数据库

例：

后台认证：strSQL = "SELECT \* FROM users WHERE (name = ' " + userName + " ') and (password = ' "+ passWord +" ');"

用户名填入：'or 1 = 1 --

则后台会判定SELECT \* FROM users WHERE username=''or 1 = 1 -- and password=''；能正确执行，进入系统

更严重的，用户名填入'; DROP DATABASE (DB name) --可直接删库

sql注入攻击防止：使用预编译语句集PreparedStatement，利用setxxx方法传值，只将输入串作为数据处理而不解析sql语句；js检查是否包含非法字符；字符串过滤

## 1.18 Mysql和Redis比较

应用场景，各自优劣

Redis为什么快？（io模型 + 基于内存 + 线程模型）采用了多路复用io阻塞机制；数据结构简单，操作节省时间；运行在内存中

为什么内存比磁盘快：磁盘是通过磁头切换读取，内存时通过电读取。

## 1.19 CAP+BASE

## 1.20 Redis数据类型及其基本操作、使用场景

String：内容可以是任何值，存储字符串、数、图，计数器

k-v：存储有结构信息的数据，比string更为方便

list：双端链表，可做栈也可做队列，如社区应用中点赞

set：去重、抽奖、好友相似度

zset：热搜、附近的人

## 1.21 Redis缓存策略

Redis清除过期键的策略：惰性删除（放任不管，取出来时发现过期了就删掉，返回空）+定期清除（每个一段时间，对expires字典进行检查，删除过期键）

## 1.22 Redis持久化

## 1.23 Redis事务 乐观锁 悲观锁

## 1.24 Redis主从复制

## 1.25 缓存穿透、缓存雪崩、缓存击穿

缓存穿透：key对应的数据在数据库并不存在，每次针对此key的请求从缓存获取不到，请求都会到数据库，从而可能压垮数据库。比如用一个不存在的用户id获取用户信息，不论缓存还是数据库都没有，若黑客利用此漏洞进行攻击可能压垮数据库。解决办法：最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中。

缓存击穿：key对应的数据存在，但在redis中过期，此时若有大量并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。解决方案：使用互斥锁，mutex，缓存失效时，不立即加载数据库，而是通过setnx设置一个mutex key，然后在加载数据库回设缓存，缓存未失效了就从缓存取。

缓存雪崩：当缓存服务器重启或者大量缓存集中在某一个时间段失效（设置了相同过期时间），大并发的请求会给后端系统带来很大压力。解决方案：加锁或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写；将缓存失效时间分割开

## 1.26 MySQL触发器

触发器（trigger）是SQL server 提供给程序员和数据分析员来保证数据完整性的一种方法，它是与表事件相关的特殊的存储过程，它的执行不是由程序调用，也不是手工启动，而是由事件来触发，当对一个表进行操作（insert，delete update）时就会激活它执行。触发器经常用于加强数据的完整性约束和业务规则等。

触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。基于行触发，增删改前后都可以会触发。

比如你现在有两个表【用户表】和【日志表】，当一个用户被创建的时候，就需要在日志表中插入创建的log日志，如果在不使用触发器的情况下，你需要编写程序语言逻辑才能实现，但是如果你定义了一个触发器，触发器的作用就可以是当你在用户表中插入一条数据的之后帮你在日志表中插入一条日志信息。

数据库触发器作用：1. 安全性，可以基于数据库的值使用户具有操作数据库的某种权利，例如不允许股票的价格的升幅一次超过10%。2. 审计。可以跟踪用户对数据库的操作，比如把用户对数据库的操作写入审计表。3. 实现复杂的数据完整性规则。触发器可以对数据库中相关的表进行连环更新。例如，在auths表author\_code列上的删除触发器可导致相应删除在其它表中的与之匹配的行。4. 同步实时地复制表中的数据。5. 自动计算数据值，如果数据的值达到了一定的要求，则进行特定的处理。例如，如果公司的帐号上的资金低于5万元则立即给财务人员发送警告数据。

触发器定义语法：

CREATE TRIGGER trigger\_name trigger\_time trigger\_event ON tb\_name FOR EACH ROW trigger\_stmt

trigger\_name：触发器的名称

tirgger\_time：触发时机，为BEFORE或者AFTER

trigger\_event：触发事件，为INSERT、DELETE或者UPDATE

tb\_name：表示建立触发器的表明，就是在哪张表上建立触发器

trigger\_stmt：触发器的程序体，可以是一条SQL语句或者是用BEGIN和END包含的多条语句

MySQL可以创建以下六种触发器：

BEFORE INSERT, BEFORE DELETE, BEFORE UPDATE

AFTER INSERT, AFTER DELETE, AFTER UPDATE

# 2. 计算机网络

## 2.1 OSI七层协议

## 2.2 ARP协议作用

解决同一局域网上的主机或路由器的IP地址和硬件地址的映射问题

## 2.3 IP地址

分类、每个IP地址均包含两个标识码(ID)，即网络ID和主机ID

## 2.4 路由器和网桥的区别

网桥工作在链路层，基于 MAC 地址转发，路由器工作在网络层，基于 IP 转发；

网桥只能连接两个相同的网络，就是把不同物理位置的机器组成一个大的局域网，而路由器可以连接不同网络；

网桥不隔离广播，而路由器可以隔离广播；

（交换机不能用于不同网段通信，除非是三层交换机）

## 2.5 ICMP协议的应用（ping， traceroute）

## 2.6 TCP、UDP

TCP标志位：SYN（建立连接）、ACK（响应、确认）、PSH（有数据传输）、FIN（关闭连接）、RST（重置）、URG（紧急指针字段值有效）

## 2.7 TCP如何实现可靠连接，如何实现流量控制、拥塞控制

可靠性：接收校验、确认响应、超时重传（接收方收到数据包后进行校验，校验成功会发送确认响应，若校验出错会丢弃数据包，不发送确认响应，一段时间后发送方会超时重传；发送方发送完数据包后，会启动一个计时器，如果没有及时收到确认响应，则会重新发送数据包）

滑动窗口机制：发送方和接收方分别维护着独立的发送和接收缓冲区，需要发送的所有数据都在发送缓冲区，允许发送和已发送未收到确认的数据在发送窗口。

流量控制：接收方传递消息给发送方时，会包含接收窗口大小。如果接收方告诉发送方缓冲区已满，则发送方停止发送数据，接收区出现富余后会通知发送方接收窗口大小。若接收区发送的报文丢失，可能出现发送方等待接收方的通知，同时接收方等待发送方发送数据的死锁状态，TCP引入持续计数器，发送方接收到接收方的零窗口通知时，启动该计数器，时间到则发送探测报文，回应仍为0则继续等待。

拥塞控制：发送窗口=min（接收窗口，拥塞窗口）。慢开始+拥塞避免算法（发送数据时，先探测网络拥塞程度，先发送1字节探测报文，收到确认后再发送2字节的报文，依次指数递增，直到慢开始门限，即从1开始每次指数增长。到了门限后采用线性增长，当出现网络拥塞，比如丢包时，将慢开始门限设为出现拥塞时窗口大小的一半，从1开始执行慢开始算法）。快重传+快恢复算法（接收方收到一个失序报文段后立即发出重复确认，发送方收到3个重复确认后立即重传，比规定的接收确认时间短；快恢复：拥塞时不从1开始，直接从新门限线性增长。）

## 2.8如何实现UDP的可靠

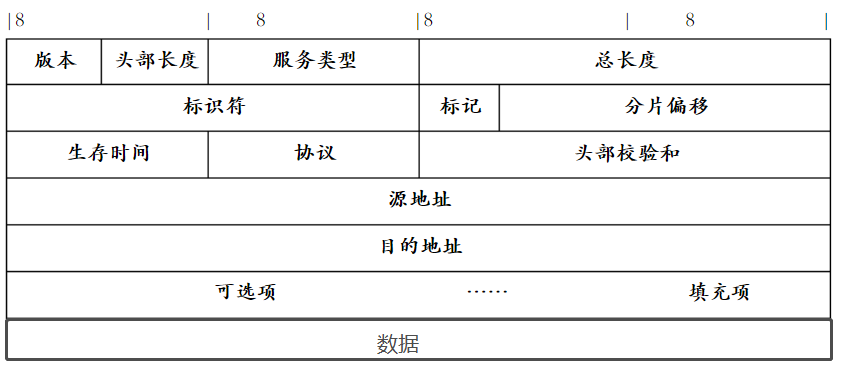
添加seq/ack机制，确保数据发送到对端；添加发送和接收缓冲区，主要为了超时重传；添加超时重传机制

## 2.9 三次握手过程，为什么不能两次

## 2.10 四次挥手过程

## 2.11 各层传输数据及头部结构

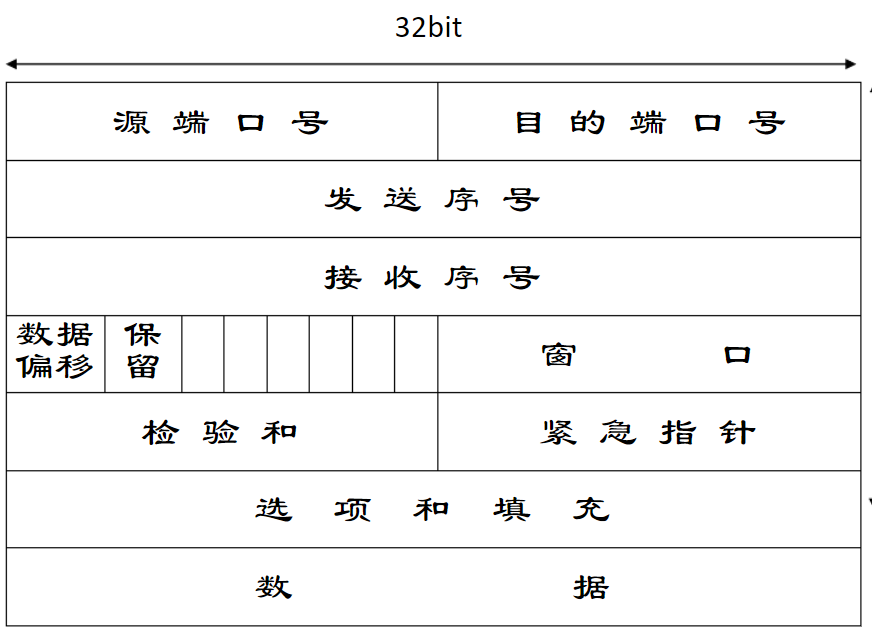
IPv4：



IPv6：



TCP：



UDP：



## 2.12 DNS、过程、DNS劫持

先搜索浏览器自身的DNS缓存，未找到后搜索操作系统、hosts、运营商提供的dns服务器，获取IP。

## 2.13 HTTP 常见状态返回码 GET/POST/PUT/DELETE

GET：通常用于请求url获得资源，也可提交数据，提交的数据显示在url中，不安全，有字符编码限制和传输数据量限制，但速度快；

POST：添加新的资源，通常用于表单提交，请求头多了Content-Type和Content-Length，post提交的表单数据放在请求体中，支持多种编码方式，无大小限制，比get更安全。

PUT通常用于更改资源，两次相同请求时后一次会把前一次覆盖掉。POST不会。

## 2.14 访问一个网页的过程

## 2.15 两台机器如何测试连通

## 2.16 cookie和session区别

会话，指用户登录网站后的一系列动作，cookie和session是会话跟踪技术。

Cookie实际上是一小段文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用response向客户端浏览器颁发一个Cookie。客户端会把Cookie保存起来。当浏览器再请求该网站时，浏览器把请求的网址连同该Cookie一同提交给服务器。服务器检查该Cookie，以此来辨认用户状态。

Session是另一种记录客户状态的机制，保存在服务器上。客户端浏览器访问服务器的时候，服务器会自动为其分配一个SessionId，客户端浏览器再次访问时只需要从Session中查找该客户的状态就可以了。

session保存在服务器上；cookie保存在本地浏览器上；

cookie的存储限制了数据量只允许4KB，而session则没有；

cookie不是很安全，可能会被利用，session是安全的；

cookie不占用服务器资源，但session过多会影响服务器性能

## 2.17 http和https区别

## 2.18 https ssl加密和证书签发过程

https=http+ssl安全传输协议+ca身份认证

①客户端的浏览器向服务器请求传送客户端SSL 协议的版本号，加密算法的种类，以及其他服务器和客户端之间通讯所需要的各种信息。

②服务器向客户端传送SSL 协议的版本号，加密算法的种类以及其他相关信息，同时服务器还将向客户端传送自己的证书。（服务端会向CA确认客户端的信息真实性，CA审核通过会发送证书，内容包括申请者公钥、申请者组织信息、签发机构CA信息、有效时间、证书序列号等信息的明文，同时还会包含一个签名和签名产生的算法：首先使用散列函数计算公开的明文信息的信息摘要，然后采用CA的私钥对信息摘要进行加密，密文即为签名。）

③客户利用服务器传过来的信息验证服务器的合法性，服务器的合法性包括：证书是否过期，发行服务器证书的CA 是否可靠，发行者证书的公钥能否正确解开服务器证书的“发行者的数字签名”，服务器证书上的域名是否和服务器的实际域名相匹配。如果合法性验证没有通过，通讯将断开；如果合法性验证通过，将继续进行第四步。

④用户端随机产生一个用于后面通讯的“对称密码B”，然后用服务器的公钥A（服务器的公钥从步骤②中的服务器的证书中获得）对其加密得到C，然后传给服务器。

⑤服务器用私钥D解密C得到“对称密码B”(此处的公钥和私钥是相互关联的，公钥加密的数据只能用私钥解密，私钥只在服务器端保留。然后用其作为服务器和客户端的“通话密码”加解密通讯。同时在SSL 通讯过程中还要完成数据通讯的完整性，防止数据通讯中的任何变化。

⑥客户端向服务器端发出信息，指明后面的数据通讯将使用的步骤⑤中的主密码B为对称密钥，同时通知服务器客户端的握手过程结束。

⑦服务器向客户端发出信息，指明后面的数据通讯将使用的步骤⑤中的主密码B为对称密钥，同时通知客户端服务器端的握手过程结束。

⑧SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户和服务器开始使用相同的对称密钥进行数据通讯，同时进行通讯完整性的检验。

https传输在建立连接时使用的是非对称加密算法，一旦连接建立完成，后续的通讯则使用了对称加密算法，加快速率

https连接流程：

1. 客户端发起https请求

2. 服务器将证书信息传输给客户端

3. 客户端解析证书，验证公钥是否有效，若发现异常，则弹出警示框，提示证书有问题，若无异常，则生成一个随机数串作为客户端的共享密钥

4. 用公钥对随机数串进行加密，传输到服务端

5. 服务端用私钥解密，得到共享密钥

缺点：通信速率降低（除TCP连接、发送和响应请求外，还需要进行ssl通信，整体通信信息量增加）；加密过程消耗资源；证书开销（需向认证机构购买）

浏览器上抓到的包并未加密

## 2.19 http发展历程

0.9，1991发布，只有一个GET命令，服务器只能回复HTML格式的字符串

1.0， 1996年5月，内容大大增加，任何格式的内容都可以发送，还引入了POST/HEAD命令，新增头信息/状态码/字符集/多部分发送/权限/缓存/内容编码，新增 connnection：keep-alive

1.1 引入了持久连接（persisten connnection）tcp默认不关闭，可以被多个请求复用一个请求，发送Connection:close ,明确要求服务器关闭TCP连接。增加了PUT/PATCH/HEAD/OPTIONS/DELETE方法，缺点，队头阻塞（Head-of-lineblocking）方案：1.减少请求数 2.同时多开持久化

2.0 2015 彻底的二进制协议。客户端可以同时发送多个请求和回应，不用按照顺序一一对应，避免了队头阻塞

3.0

https http协议数据传输是明文的，https使用ssl/tls协议对http进行加密处理

## 2.20 对称加密、非对称加密

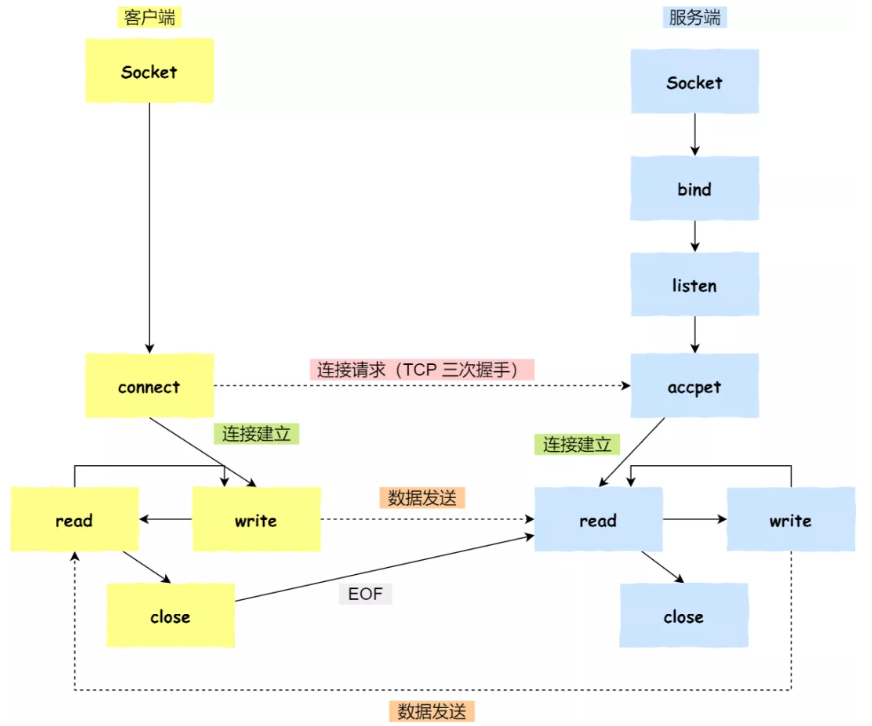
## 2.21 HTTP头部攻击、跨站脚本攻击、DoS、DDoS

跨站脚本攻击(XSS，利用网站漏洞从用户那里恶意盗取信息)、

DoS：拒绝服务，DDoS：分布式拒绝服务，DoS的一种。Dos与DDOS都是攻击目标服务器、网络服务的一种方式，使计算机或网络无法提供正常服务。Dos是利用自己的计算机攻击目标，是一对一的关系，以三次握手为例，如果攻击端发送完第一次握手的数据后，然后就“消失”了，那么服务器就会不断的发送第二次握手的数据，于是，服务器的资源大量被消耗，直到死机为止。DDOS是DoS攻击基础之上产生的一种新的攻击方式,利用控制成百上千台机器，组成一个DDOS攻击群，同一时刻对目标发起攻击。

## 2.22 socket通信

针对TCP协议：



步骤：

1. 服务端和客户端初始化socket，得到文件描述符

2. 服务端调用bind，绑定ID和端口

3. 服务端调用listen，进行监听

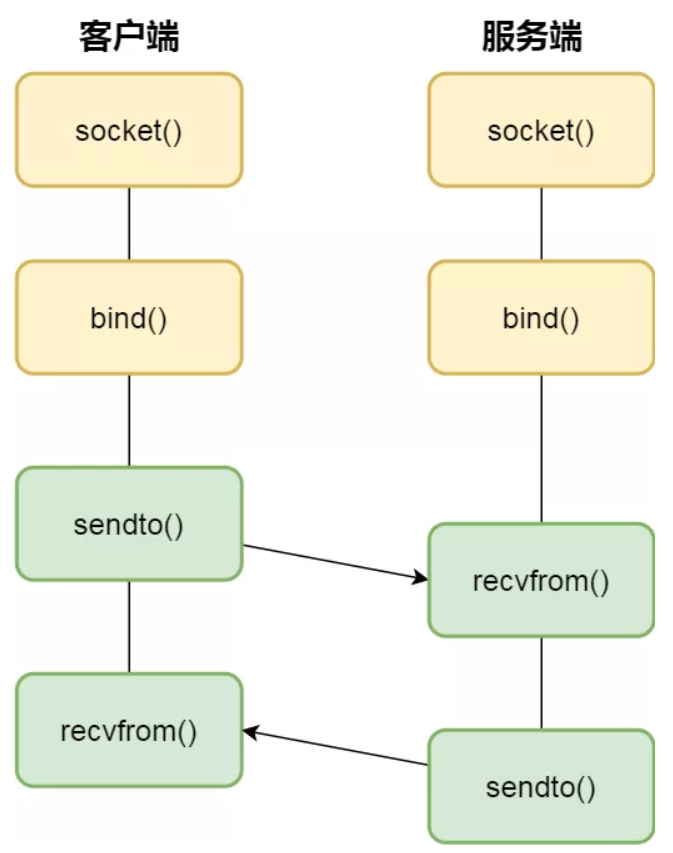
4. 服务端调用accept，等待客户端连接

5. 客户端调用connect，向服务器端的地址和端口发起连接请求

6. 三次握手

服务端调用accept，连接成功了会返回一个已完成连接的 socket，后续用来传输数据。

针对UDP：



UDP 无连接，所以不需要三次握手，也就不需要像 TCP 调用 listen 和 connect，但是 UDP 的交互仍然需要 IP 地址和端口号，因此也需要 bind。

对于 UDP 来说，不需要维护连接，那么也就没有所谓的发送方和接收方，甚至都不存在客户端和服务端的概念，只要有一个 socket ，多台机器就可以任意通信，因此每一个 UDP 的 socket 都需要 bind。另外，每次通信时，调用 sendto 和 recvfrom，传入目标主机的 IP 地址和端口。接收端打开端口进行接收。

针对本地：

本地socket支持字节流和数据报两种协议，本地字节流 socket 和 本地数据报 socket 在 bind 的时候，不像 TCP 和 UDP 要绑定 IP 地址和端口，而是绑定一个本地文件。

# 操作系统

## 进程和线程

线程不拥有系统资源，只拥有一点儿在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。

## 死锁

## 进程状态及其转化

## 进程调度策略

## 进程间的通信方式

每个进程的用户地址空间都是独立的，一般而言是不能互相访问的，但内核空间是每个进程都共享的，所以进程之间要通信必须通过内核。

管道，消息队列，共享内存，信号量，socket，信号

1. 管道：‘|’连接，匿名管道，前一个输入作为后一个输出，只能单向通信，通信范围只能在具有亲属关系的进程间使用；命名管道，用mkfifo创建，用echo和cat写入和读出数据，命名管道允许无亲缘关系进程间的通信。管道简单，但通信方式效率低，不适合进程间频繁地交换数据。

2. 消息队列，是消息的链表，存放在内核中，消息体（节点）是固定大小的存储块，存放着数据报的类型（双方约定好）和内容，进程从消息队列中读取了消息体，内核就会把它删除。消息队列允许一个或多个进程写入或者读取消息。生命周期随内核，进程终止时，消息队列及其内容并不会被删除。可实现双向通信。缺点：可能通信不及时；消息队列不适合比较大数据的传输，有消息长度和队列长度限制；数据写到消息队列及读出时，存在内核态和用户态之间的数据拷贝开销。

3. 共享内存：不需要用户态和内核态间的数据拷贝，将同一块物理内存一块映射到不同的进程的虚拟地址空间中，实现不同进程间对同一资源的共享，无需拷贝，提高通信速度。问题：多个进程同时修改共享内存时会发生冲突。

4. 信号量：是一个整型的计数器，用来控制多个进程对共享资源的访问，主要用于实现进程间的互斥与同步，基于PV 操作。解决共享内存的同步问题。信号量+共享内存通常结合在一起使用。

5. 信号：异常情况下的工作模式，需要信号来通知进程。信号是进程间通信机制中唯一的异步通信机制，因为可以在任何时候发送信号给某一进程。kill –l可以查看所有信号；crtl+c产生sigint信号，终止进程；kill -9 xxx 结束ID为xxx的进程。

6. 套接字：用于不同机器间的进程通信。

使用共享内存的方式如何加锁：写操作用互斥锁，读操作用共享锁。

## 进程的同步和互斥

同步：一个操作结束，才能执行另一个操作，不然等待；

互斥：最多允许一个进程处于临界区

## 僵尸进程和孤儿进程 linux如何处理孤儿进程

子进程先于父进程退出后，子进程的PCB需要其父进程释放，但是父进程并没有释放子进程的PCB，这样的子进程就称为僵尸进程，僵尸进程实际上是一个已经死掉的进程。僵尸进程在进程列表中保留一个位置，记载该进程的退出状态等信息供其他进程收集，占用ID损耗系统运行。

一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。系统会帮助父进程回收处理孤儿进程，所以孤儿进程实际上是不占用资源的。

## 线程状态

创建、就绪、运行run、阻塞sleep suspend、死亡

1. 新建(NEW)：新创建了一个线程对象。

2. 就绪/可运行(RUNNABLE)：线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu 的使用权 。

3. 运行(RUNNING)：可运行状态(runnable)的线程获得了cpu 时间片（timeslice），执行程序代码。

4. 阻塞(BLOCKED)：阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

(一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waiting queue)中。

(二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。

(三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms)或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。

5. 死亡(DEAD)：线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

## 线程间的通信方式

同步、while轮询、wait/notify、管道

<https://www.cnblogs.com/hapjin/p/5492619.html>

## 3.10 IO多路复用

阻塞I/O：IO操作如read和write，当调用read时，如果没有数据收到，那么线程或者进程就会被挂起，直到收到数据。服务器的多个连接需要多个线程或进程来处理，线程切换频繁，消耗内存和CPU。

非阻塞I/O：调用read时，如果有数据收到，就返回数据，如果没有数据收到，就立刻返回一个错误，如EWOULDBLOCK。这样不会阻塞线程，但是要不断的轮询来读取或写入。

IO多路复用是指使用一个线程来检查多个文件描述符（Socket）的就绪状态，减少了内存开销和上下文切换的CPU开销。

select，poll，epoll都是IO多路复用的机制

select本质是通过设置或者检查存放fd（文件描述符）标志位的数据结构来进行下一步处理，只知道IO事件发生，不知道是哪个。缺点是1. 每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大；2. 每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大，O(n)的无差别轮询复杂度；3. select支持的文件描述符数量有限（32位1024,64位2048）

poll和select无本质区别，它没有最大连接数的限制，基于链表存储

epoll，时间复杂度O(1)，相对于select来说，epoll没有描述符个数限制，使用一个文件描述符管理多个描述符，将用户关心的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

epoll相关函数：

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

epoll\_create 函数创建一个epoll句柄，参数size表明内核要监听的描述符数量。调用成功时返回一个epoll句柄描述符，失败时返回-1。

epoll\_ctl 函数注册要监听的事件类型，参数：epfd表示epoll句柄；op表示fd类型（创建/修改/删除）；fd表示要监听的描述符；event表示要监听的时间。

epoll\_wait 函数等待事件的就绪，成功时返回就绪的事件数目，调用失败时返回 -1，等待超时返回 0。

epoll能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。原因是获取事件时，无须遍历整个被侦听的描述符集，只要遍历被内核IO事件异步唤醒而加入Ready队列的描述符集合

工作模式：LT（水平触发）、ET（边缘触发）

LT：默认工作模式，即当epoll\_wait检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用epoll\_wait时，会再次通知此事件

ET：当epoll\_wait检测到某描述符事件就绪并通知应用程序时，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次通知此事件。（边缘触发只在状态由未就绪变为就绪时只通知一次，如接受缓冲区从空到有数据，从满到开始有空数据）

ET模式很大程度上减少了epoll事件的触发次数，因此效率比LT模式高。

## 3.12 页和段

虚拟内存管理

## 3.13 虚拟内存

将小部分代码装入内存，使大程序在小内存上运行。

# 测试

测试时为了发现错误而执行程序的过程，不是为了证明不存在错误或能达到预定的功能。

## 测试方法

黑盒、白盒

黑盒测试：数据驱动的测试，白盒测试：逻辑驱动的测试

白盒：语句覆盖、判定覆盖、条件覆盖、判定/条件覆盖、路径覆盖

黑盒：等价划分（确定等价类、生成测试用例）、边界分析（恰好等于，超过或小于）、因果图（考虑输入条件的组合情况）、错误猜想（增加更多特殊的测试用例）

## 自动化测试工具

Junit，JMockit

单元测试指标：测试用例通过率(100%，黑盒)，代码覆盖率(70-80%，白盒)

代码静态分析工具：eslint（JS）/Pylint(Python)

性能测试工具

## 软件开发过程

制定计划、需求分析、软件设计、程序编写、软件测试和运行维护

## 敏捷开发流程

以用户的需求进化为核心，采用迭代、循序渐进的方法进行软件开发。把一个大项目分为多个相互联系，但也可独立运行的小项目，并分别完成，在此过程中软件一直处于可使用状态

## 4.5 瀑布模式

开发过程是通过设计一系列阶段顺序展开的，将软件生命周期划分为制定计划、需求分析、软件设计、程序编写、软件测试和运行维护等六个基本活动，规定了它们自上而下、相互衔接的固定次序，如同瀑布流水，逐级下落。从系统需求分析开始直到产品发布和维护，每个阶段都会产生循环反馈

# 5. Python

## 5.1 语言比较

## 5.2 Pandas

https://www.cnblogs.com/wobujiaonaoxin/articles/11378281.html

## 5.3 new和init

\_\_new\_\_()和\_\_init\_\_()的区别：new是创建对象的函数，返回的对象是self，new会调用init；init是初始化函数（构造函数）。

## 5.4 编码方法

## 5.5 可变参数、关键字参数

## 5.6 迭代器

## 5.7 生成器

## 5.8 装饰器

## 5.9 高阶函数

## 5.10 浅拷贝、深拷贝

## 5.11 \_x, \_\_x, \_\_x\_\_

\_x: 私有，仅供内部使用

\_\_x: 为了避免子类定义的名词冲突

\_\_x\_\_：特殊用途，比如\_\_init\_\_是对象构造函数

## 5.12 is和==区别

is是比较id，==是比较值

java中：==对于基本数据类型，比较的是值是否相等；对于引用类型，比较的是地址值是否相等（即比较是否为同一对象的引用）；equals()是一个方法，只能比较引用数据类型，重写前比较的是地址值（object类中是==），重写后一般比较对象属性

## 5.13 Python多线程

\_thread和threading两种方式，threading是高级模块，对\_thread进行了封装，通常使用threading。

普通创建线程方式：传入函数并创建Thread实例，调用start执行。

<https://www.cnblogs.com/luyuze95/p/11289143.html>

import threading

import time

def run(n):

print("task", n)

time.sleep(1)

print('2s')

time.sleep(1)

print('1s')

time.sleep(1)

print('0s')

time.sleep(1)

if \_\_name\_\_ == '\_\_main\_\_':

t1 = threading.Thread(target=run, args=("t1",))

t2 = threading.Thread(target=run, args=("t2",))

t1.start()

t2.start()

----------------------------------

>>> task t1

>>> task t2

>>> 2s

>>> 2s

>>> 1s

>>> 1s

>>> 0s

>>> 0s

使用setDaemon(True)把所有的子线程都变成了主线程的守护线程，因此当主进程结束后，子线程也会随之结束，整个程序就退出了。

为了让守护线程执行结束之后，主线程再结束，可以使用join方法，让主线程等待子线程执行。

同一个进程中的多线程是共享资源的。

锁：互斥锁Lock、递归锁RLock(和互斥锁一样，只是支持嵌套)、信号量

例子：

互斥锁

from threading import Thread,Lock

import os,time

def work():

global n

lock.acquire()

temp=n

time.sleep(0.1)

n=temp-1

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

lock=Lock()

n=100

l=[]

for i in range(100):

p=Thread(target=work)

l.append(p)

p.start()

for p in l:

p.join()

递归锁：

import threading

import time

def Func(lock):

global gl\_num

lock.acquire()

gl\_num += 1

time.sleep(1)

print(gl\_num)

lock.release()

if \_\_name\_\_ == '\_\_main\_\_':

gl\_num = 0

lock = threading.RLock()

for i in range(10):

t = threading.Thread(target=Func, args=(lock,))

t.start()

信号量

import threading

import time

def run(n, semaphore):

semaphore.acquire() #加锁

time.sleep(1)

print("run the thread:%s\n" % n)

semaphore.release() #释放

if \_\_name\_\_ == '\_\_main\_\_':

num = 0

semaphore = threading.BoundedSemaphore(5) # 最多允许5个线程同时运行

for i in range(22):

t = threading.Thread(target=run, args=("t-%s" % i, semaphore))

t.start()

while threading.active\_count() != 1:

pass # print threading.active\_count()

else:

print('-----all threads done-----')

Python多线程问题：无论有多少核，同时只能执行一个线程

GIL（Global Interpreter Lock）全局解释器锁，某个线程想要执行，必须先拿到GIL，我们可以把GIL看作是“通行证”，在一个python进程中，GIL只有一个，同一时间只能执行一个线程。

python下想要充分利用多核CPU，就用多进程。每个进程有各自独立的GIL，互不干扰，这样就可以真正意义上的并行执行。

# 6. Java

## 6.1 Java容器

Collections（List/Set/Queue）、Map

## 6.2 覆盖和重载，继承封装和多态 父类静态方法能否被子类重写（不能重写，只能继承）

静态变量和静态方法在编译的时候就将其与类绑定在一起；重写指的是根据运行时对象的类型来决定调用哪个方法，而不是根据编译时的类型

## 6.3 HashMap和HashTable

HashMap基于哈希表，是对Map接口的实现，不是线程安全的，可以存储为null的键值对，不保证映射顺序。底层是数组，每个数组元素Entry是一个key-value对，有指向下一元素的引用，构成链表。put操作时，根据key的哈希码和散列函数得到数组中的元素，该位置没有元素就直接放在该位置上，有元素则对链表所有元素进行遍历（看看是否已存在），将元素以链表形式存放，新Entry放入链头。

HashMap允许多个线程同时操作，比如addEntry操作，两个线程在同一位置add，线程1未结束，线程2开始，线程1执行结束时会把2的值覆盖。所以线程不安全。使用迭代器时有其他线程修改map，会抛ConcurrentModificationException异常（modCount不等于初始化时的modCount）。

resize：当元素个数超过数组大小\*loadFactor（默认0.75）时，会进行扩容，扩大一倍重新计算每个元素在数组中的位置，耗性能。多线程时会产生竞争，最先读出来的头结点，最后会变成尾结点。

区别：

(1) HashTable继承自Dictionary而HashMap继承自AbstractMap, 都实现了Map接口

(2) HashTable的方法是同步的，每个方法加了synchronized关键字，一个线程访问synchronized方法时，其他线程无法访问，相当于加锁。不允许key或value为null。HashMap方法不同步，线程不安全，允许null。1.5以后ConcurrentHashMap是HashTable的代替，解决HashTable加锁后效率低问题。

(3) HashMap效率高于HashTable

(4) HashTable hash数组默认大小是11，增加方式是old\*2+1，HashMap默认是16，一定是2的指数。计算哈希散列码的函数不一样。

## 6.4 HashMap jdk1.7和1.8改进 为什么长度大于8转红黑树，8怎么来的

（1）1.7每个数组元素是个Entry，put的节点作为链表头节点；1.8每个数组原始是Node，新增节点插到链表尾部；（多线程情况下，头插法容易成环，尾插法是发生数据覆盖；扩容时1.7会变顺序，1.8不会）

（2）发生冲突时，1.7 将相同散列值的元素放于同一链表，1.8中，冲突元素个数大于8（默认）时，就转换为红黑树，删除节点后，个数小于6时又转为链表。（选8是根据概率统计选择的，在理想状态下，受随机分布的hashCode影响，链表中的节点遵循泊松分布，通常情况下，链表长度很难达到8，达到8时链表性能已经很差，就采用红黑树提高性能）

（3）初始阈值设置不同，1.7是默认是初始容量，1.8是大于等于初始容量且为2的指数，resize后均为容量\*装载因子。1.7resize后要调换顺序，1.8不用。

（4）计算哈希码的方式不一样

## 6.5 迭代器是做什么的？迭代器的fail-fast机制了解吗？主要为了解决什么问题？

迭代器是遍历集合的一种方式，不同集合均可使用迭代器，不用关心内部存储结构。for (Type x: collections) {}

fail-fast机制：集合在迭代时，如果有其他线程在修改，会抛出ConcurrentModificationException异常。解决线程不安全问题。

## 6.6 ConcurrentHashMap的底层数据结构

（要掌握put，get的全过程，如果能对rehash有深入的理解就更好了）

扩展：为什么1.8放弃了分段锁，分段锁的优缺点以及1.8的改进

HashTable同步时相当于给整个哈希表加一个大锁，只要有一个线程访问或操作该对象，其他的线程阻塞或轮询等待，线程竞争激烈的情况下，这种方法效率非常低。

ConcurrentHashMap1.7采用分段锁，将数据分段存储，每一段分配一个Segment，当线程占用一个Segment时，其他线程可访问其他数据段。ConcurrentHashMap底层是一个Segment数组，一个Segment就是一个子哈希表，每个Segment包含一个HashEntry数组并守护它，修改HashEntry数组数据时，先获取对应的Segment锁。HashEntry数据采用分离链表法处理冲突，每个HashEntry元素是链表结构。

ConcurrentHashMap不允许key或value为空。

执行put时，先定位到Segment数组的索引，再定位到HashEntry数组索引。

ConcurrentHashMap迭代方式中，当iterator创建后，集合发生改变不抛出ConcurrentModificationException，而是改变时创建新的数据，不影响原来的数据迭代，iterator完成后再将头指针替换为新数据。

ConcurrentHashMap1.8：HashEntry是Node，冲突数据超过8时链表变为红黑树。取消了分段锁Segment概念，**使用Synchronized和CAS实现线程安全**，降低了锁的粒度，粒度是HashEntry（或者说是Node，锁住头节点）,1.7粒度是基于Segment的。1.8的数据结构更简单，无Segment，每次编辑时用Synchronized锁住该节点。使用CAS机制，当多个线程并发操作时，只有一个线程可以成功。

## 6.7 String, StringBuilder, StringBuffer的区别

运行速度：StringBuilder > StringBuffer > String

String运行速度慢的原因：String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，在拼接时每次会创建新的字符串，扔掉旧的字符串，浪费内存，影响gc效率；后两者的对象是变量，是可以更改的，可预分配缓冲区，往StringBuilder新增字符时，不会创建新的临时对象

String不变，线程安全

线程安全：StringBuilder和StringBuffer方法和功能完全是等价的，只是StringBuilder是线程不安全的，而StringBuffer是线程安全的，StringBuffer中很多方法可以带有synchronized关键字。单线程时用StringBuilder，无需加锁，速度快。

## 6.8 ArrayList和LinkedList的区别，增删改查效率

## 6.9 Final finally finalize区别

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在初始化后不能被改变。finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个复活的机会，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常

## 6.10接口和抽象类区别

都不能被实例化；

抽象类只能单继承，接口可以多实现；

在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性，这是抽象类的优势；接口中只能有抽象的方法。（java1.8后可以实现default方法）；

抽象类可以定义实例字段，接口不能定义实例字段；

## 6.11 int和Integer，谁占内存多

Integer，Integer 是一个对象，需要存储对象的元数据

## 6.12 Object有哪些公用方法

native方法指本地方法，是用c/c++编写的程序编译后得到的dll文件，java加载dll文件中的函数，局限性是依赖平台，可移植性不高

1） Object()：默认构造函数

2） private static native registerNatives(): 注册本地方法，初始化java方法到c方法的映射

3） public final native Class<?> getClass(): 返回该对象运行时的类

4） public final native hashCode(): 返回对象的哈希码（即内存地址），使用哈希表相关的数据类型时，是根据哈希码来判断是否相等，因此需要重写hashCode和equals

5） public boolean equals(Object obj) {return (this == obj);}: 返回两个对象是否相等，判断的是地址。当需要判断内容是否相等时，需要重写。String已重写。

6） protected native Object clone()：创建并返回该对象的一个拷贝，调用该方法的对象一定要实现Cloneable接口，默认返回浅拷贝。

7）public String toString(): 默认返回类名称和当前对象的16进制哈希码

8） notify() 9)wait()

10）protected void finalize(): gc回收对象之后会调用，Object类中未实现，子类可以重写，用于资源释放，不推荐使用，该方法不保证会被jvm执行，而且性能低。

## 6.13 类加载过程

概念：根据类的全名寻找该类的二进制字节流（.class文件），并将其放在方法区，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构，提供访问方法区内数据结构的接口。

类加载过程包括加载、验证、准备、解析和初始化五个阶段，加载、验证、准备和初始化按顺序开始，而解析不一定，为了支持动态绑定。

1．加载：查找并加载类的二进制数据。JVM在加载阶段需要完成三件事，一是通过类的全名来获取二进制字节流；二是将字节流代码的静态存储结构转化为方法区运行时的数据接口；三是在堆中生成一个代表这个类的Class对象，作为方法区数据结构的访问入口。

2. 连接

验证（非必须）：确保被加载类的正确性，包括文件格式验证、元数据验证、字节码验证、符号引用验证

准备：为类的静态变量分配内存，并将其初始值设为默认值

解析：将类中的符号引用转换为直接引用。

3. 初始化：为类的静态变量赋予正确的初始值。

初始化顺序：父类静态成员变量、子类静态成员变量、父类成员变量、父类构造器、子类成员变量、子类构造器

类加载的三种方式：命令行启动时由JVM初始化加载；通过Class.forName()动态加载；通过ClassLoader.loadClass动态加载

双亲委派模型工作流程：如果一个类加载器收到了类加载的请求，首先不会尝试去加载这个类，而是把请求委托给父加载器，依次向上。因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围内没找到所需的类时，子加载器才会尝试去加载该类。

## 6.14 什么是反射机制，应用场景有哪些

在程序运行过程中，可以得到一个jvm中已加载类的所有构造器、成员变量和方法，还可以得到该类的对象，调用任意一个成员变量和方法。

获取Class对象的三种方式：

1. cla = Person.class

2. Person person = new Person();

cla = person.getClass();

3. cla = class.forName(“com.java.reflection.Person”); //常用

反射机制应用场景：（很少用）

逆向代码 ，例如反编译；

与注解相结合的框架 例如Retrofit；

单纯的反射机制应用框架 例如EventBus 2.x；

动态生成类框架 例如Gson

## 6.15 Java哪些数据结构用了红黑树

TreeMap、TreeSet

## 6.16 Java序列化

序列化就是将java对象按照一定的序列化协议编码成一连串的字节序列；反序列化就是将这些字节重建成java对象。实现了Serializable接口的类对象都可以序列化。

好处：可将对象永久保存在磁盘上，需要的时候进行反序列化；可将java对象以字节流的形式在网络中传输；可以在进程间传递对象

## 6.17 Java内存分配和内存回收

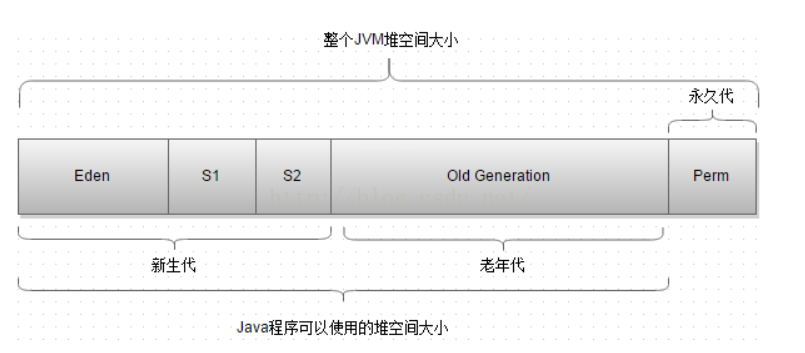
内存分配：栈（存储局部变量）、堆（存储new创建的数组、对象、非静态变量）、方法区（又称静态区，存储类信息、静态变量、final常量）、本地方法区（和系统相关）、寄存器（给CPU使用）

字符串常量存堆

static作用：1. 修饰类中的方法和字段，属于类，不属于具体的对象，被对象所共享；2. 修饰代码块，用来优化程序性能，只会在类加载的时候执行一次；3. 修饰类（只能内部类），静态类使用有很多限制，静态类只能访问外部类的静态成员，非静态内部类能够访问外部类的静态和非静态成员。非静态内部类依赖于外部类的实例，也就是说需要先创建外部类实例，才能用这个实例去创建非静态内部类。而静态内部类不需要。 4. 修饰导包import static ClassName.\* 导入这个类里的静态方法，好处是可以简化代码，不用写类名，如可将System.out.println写入一个静态方法print，可以直接使用print，Integer.MAX\_VALUE可以不用写Integer。如果用的多可以使用静态导包，只使用一两次没必要静态导包。

内存回收主要回收堆中的内存。

堆空间划分：



堆空间分为三个区：新生代、老年代和永久代（永久代是方法区）。新生代也分为三个区域：Eden、From Survivor、To Survivor（比例默认为8:1:1）。堆大小=新生代+老年代。

Java程序中不能显示地分配和注销内存，都是由jvm处理。JVM 每次只会使用 Eden 和其中的一块 Survivor 区域来为对象服务，所以无论什么时候，总是有一块 Survivor 区域是空闲着的。因此，新生代实际可用的内存空间为 9/10 (即90% )的新生代空间。（将对象设置为null或调用System.gc()来试图显式清理内存，调用System.gc()会显著影响系统性能，JVM执行GC会暂停所有应用程度的执行，stop-the-world，应该杜绝使用）

堆内存回收分为两种：Minor GC（新生代）、Major/FULL GC（老年代）

Minor GC采用复制算法，发生频率高，不一定要Eden区满才触发，具体做法：当对象在 Eden (包括一个Survivor区域，假设是 from ) 出生后，在经过一次 Minor GC 后，如果对象还存活，并且能够被另外一块 Survivor 区域所容纳(to区域有足够的内存空间来存储Eden和from区域中存活的对象)，则使用复制算法将这些仍然还存活的对象复制到另外一块 Survivor 区域 (即to区域) 中，然后清理所使用过的 Eden 以及 Survivor 区域 (即from区域)，并且将这些对象的年龄设置为1，交换from和to区域，以后对象在 Survivor 区每熬过一次 Minor GC，就将对象的年龄 + 1，当对象的年龄达到某个值时 ( 默认是 15 岁，可以通过参数 -XX: MaxTenuringThreshold 来设定 )，这些对象就会成为老年代。 但这也不是一定的，对于一些较大的对象 (即需要分配一块较大的连续内存空间) 则是直接进入到老年代。老年代满时触发Full GC。

需要两个Survivor区的原因：1.更明显的复制算法，每次只用一块内存，始终保持另一块为空；2. 如果只有一块Survivor区，一次gc之后，Survivor区就存在了存活对象，第二次gc后，对Survivor进行回收，就会产生内存碎片，达不到复制效果。

Full GC 是发生在老年代的垃圾收集动作，所采用的是标记-清除算法。从根集合开始，对存放的对象进行标记，标记完后扫描整个空间中未被标记的对象，进行回收。由于垃圾对象所处的内存位置可能很离散，这种算法很容易产生内存碎片，需要为大对象分配空间时无法找到足够的空间。

标记整理算法：标记存活对象，将存活对象移到一端，清除掉边界意外的内存。

触发Full GC情况：调用System.gc()，不建议；老年代空间不足；永久代空间不足(1.7及以前)，需要加载的类或反射类较多时，永久代可能会占满。

内存溢出：申请内存无足够空间供其使用

内存泄露：分配的内存不再使用但无法回收，导致本该被回收的对象不能被回收而停留在堆内存中。

内存泄露原因：1. 单例造成的内存泄露；2. 非静态内部类创建静态实例造成的内存泄露；3. Handler造成的 4. 线程造成的 5. 资源为关闭造成的 6. 使用ListView造成的 7. 集合容器造成的 8. WebView造成的

如何判断可回收：无任何变量指向的对象是垃圾对象，但无法解决循环引用问题，jvm采用可达分析法，通过GC root对象作为起点进行搜索，不可达对象一定会被标记为可回收对象。可作为GC root对象的有：虚拟机栈中引用的对象（本地变量表）、本地方法栈中引用的对象（Native 对象），方法区中静态属性引用的对象、方法区中常量引用的对象

## 6.18 Error和Exception区别

Error一般指与虚拟机相关的问题，比如系统奔溃、虚拟机奔溃、内存泄露，仅靠程序本身无法解决 OutofMemoryError，StackOverflowError，Error不应该被捕获。

Exception表示程序可以处理的异常。异常分为受检异常和运行时异常，受检异常是编译时异常，编译器在编译时进行校验，是正常逻辑的补偿处理手段，通过throws或者try catch来处理，如NoSuchFileException,NumberFormatException；运行时异常是运行过程中可能发生的异常，无需使用throws，异常发生时线程应该停止，比如NullPointerException，IndexOutOfBoundException。

## 6.19 创建对象的方式

1.运用new 关键字创建实例，这是最常用的创建对象方法。

2.运用反射，调用Java.lang.Class类当中newInstance方法。只能调用公共的无参构造函数。

3.运用反射，调用java.lang.reflect.Constructor类中的newInstance方法提供无参或有参实例。除了无参构造器，还可以调用有参数的/私有的/受保护的构造函数。事实上Class的newInstance方法内部调用Constructor的newInstance方法。这也是众多框架Spring、Hibernate、Struts等使用后者的原因。

4.调用对象的clone方法。必须先实现java.lang.Cloneable接口。

5.使用序列化和反序列化。必须先实现Serializable接口。

6.使用unsafe.allocateInstance(class)创建对象

## 6.20注解

@restController：springboot相关

## 6.21 Comparable和Comparator比较

## 6.22 引用

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析算法判断对象是否可达，判定对象是否可被回收都与引用有关。

强引用、弱引用、软引用、虚引用

强引用：被强引用关联的对象不会被回收。（使用new创建）

软引用：被软引用关联的对象只有在内存不够的情况下才会被回收。使用 SoftReference 类来创建软引用。

弱引用：被弱引用关联的对象一定会被回收，也就是说它只能存活到下一次垃圾回收发生之前。使用 WeakReference 类来创建弱引用。

虚引用：一个对象是否有虚引用的存在，不会对其生存时间造成影响，也无法通过虚引用得到一个对象。为一个对象设置虚引用的唯一目的是能在这个对象被回收时收到一个系统通知。

**Java多线程：**

## 6.23 创建线程方法

1. 直接继承Thread类，new一个Thread对象，然后调用start()方法；

2. 实现runnable接口，然后调用Thread的构造器，new Thread(runnable target), 得到Thread对象后调用start()方法

3. 实现callable接口，与 Runnable 相比，Callable 可以有返回值，返回值通过 FutureTask 进行封装。

更推荐实现接口的方式，实现接口的同时还可以继承其他的类，而java不能多继承，第一种方法虽然简单，但不能继承其他类。另外，直接继承整个Thread类开销比较大

FutureTask：FutureTask 实现了 RunnableFuture 接口，该接口继承自 Runnable 和 Future 接口，这使得 FutureTask 既可以当做一个任务执行，也可以有返回值。

FutureTask 可用于异步获取执行结果或取消执行任务的场景。当一个计算任务需要执行很长时间，那么就可以用 FutureTask 来封装这个任务，主线程在完成自己的任务之后再去获取结果。

start()和run()方法的区别：

start()创建了一个子线程，真正实现了多线程，无需等待run()方法体中的代码执行完毕，就可以接着执行下面的代码。执行start之后的线程处于就绪状态，得到CPU的时间片后就会执行其中的run方法。run方法执行结束，线程就终止。

run()没有新举措，就是调用类中的方法当作普通的方法的方式调用，未创建线程，程序中只有一个主线程，等到run()方法中的代码执行完毕才继续执行下面的代码。

线程调度：java定义了线程调度器来监控系统中所有处于runnable状态的线程，按照线程优先级决定哪个线程被CPU运行。优先级范围：1-10,1最低，10最高，默认优先级是5.（电脑多核可并发，分开跑多个优先级不同的线程）。设置优先级函数：setPriority().

线程调度器是抢占式的，线程执行过程中，有优先级更高的线程进入runnable状态，则高优先级线程立即被调度执行。

运行的主线程中，创建的子进程优先级被设置为等同于主线程的优先级。

java有两种线程：用户线程和守护线程，通过isDaemon()区分，返回false说明是用户线程，否则是守护线程。用户线程一般用于执行用户任务，守护线程是后台线程，一般用来在后台提供一种通用服务，如垃圾回收。创建子线程的父线程是守护线程时，子线程才是守护线程。

线程常用方法：

Thread.yield(): 线程让步，处于running状态的线程，执行该方法后，会让出对CPU的调度，重新回到runnable状态和其他线程竞争。

new Thread().join()：在当前线程中调用另一个线程的join方法，则当前线程进入阻塞状态，另一线程执行完毕后当前线程变为就绪状态。该方法经常用于两个线程同时执行时，线程1耗时更长，但线程2需要得到线程1的执行结果，就可以使用join方法。

new Thread().interrupt()：设置线程中断状态位为true，可以中断任何阻塞状态，线程会不时检查中断状态位，线程正常执行时不会造成任何影响，但线程进入到wait()/sleep()/join()后，就会抛出InterruptedException，抛出异常后立即将线程的中断状态清除，设为false，使线程退出阻塞状态。

Thread().sleep(): sleep是Thread类方法，表示使当前线程暂停执行，让CPU去执行其他线程，但不会释放对象锁，指定的时候后线程返回可执行状态，CPU再回到该线程继续执行。

Object.wait(): 属于Object类方法，当前线程让自己释放对象的同步锁，以便其他正在等待资源的线程得到该资源进而执行，只有调用了notify()方法，之后调用wait()的线程才会解除wait()状态，参与竞争同步资源锁，进而得到执行。（notify只是让调用wait的进程有权重新参与线程的调度。）wait和notify/notifyAll必须要与synchronized(Obj)一起使用，只能针对已经取得了Obj锁进行操作，语法角度上wait和notify需放在synchronized(Obj){…}语句块中。wait()方法调用时，会释放线程获得的锁，wait()方法返回后，线程又会重新试图获得锁。

new ReentrantLock().newCondition().await(): 达到效果与wait方法一致，都是使线程进入阻塞状态，使用signal方法唤醒，也要在同步代码块中执行。ReentrantLock是一种公平锁，在等待队列上实现先入先出。Condition与重入锁是通过lock.newCondition()方法产生一个与当前重入锁绑定的Condition实例，也就是一个等待队列。

sleep和wait异同：

同：都是多线程环境下，使线程阻塞，并在指定时间后继续执行；都可以通过interrupt()方法打断线程暂停状态，从而使线程抛出InterruptException。

异：最大区别：sleep睡眠时，保持对象锁，wait睡眠时释放对象锁；sleep属于Thread类方法，wait属于object；sleep可以在任何地方使用，wait、notify只能在同步控制方法中使用，该方法调用对象的监听器。

CAS：比较和交换

乐观锁的一种实现方式，当多个线程同时对某个资源进行CAS操作时，只有一个线程操作成功，但不会阻塞其它线程，其他只是返回操作失败信号

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B，当且仅当预期值A和内存值V相同时，将内存值修改为B，否则什么也不做。

java多线程的内存模型：

多线程三大特性：原子性（一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行）、可见性（当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够“立即”看得到修改的值）、有序性（主要用于线程间的通讯，如join、wait等操作）

java内存模型简称jmm，定义了一个线程对另一个线程可见，共享内存模型指的就是Java内存模型。共享变量存放在主内存中，每个线程都有自己的本地内存，线程的本地内存保存(load)了该线程需要用到的变量的拷贝值，线程在CPU上运行都是对自己工作线程中的数据进行读写操作，运行结束后才把数据同步到(save)主内存中。当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题，需要通过线程同步机制达到多线程对同一内存区域的读写控制。

synchronized关键字：

为了避免同一数据对象被多个线程同时访问。

修饰范围：（无论修饰什么范围，同步效果都是作用于对象）

1. 修饰代码块：synchronized(this) {…} 被修饰的代码块称为同步语句块，作用范围是{}中语句，作用的是调用代码块的对象；

2. 修饰方法，被修饰的方法称为同步方法，作用范围是整个方法，作用的是调用方法的对象

3. 修饰静态方法，作用范围是整个静态方法，作用于类的所有对象

4. 修饰类，synchronized(类名.class){},作用范围是{}，作用于类的所有对象

修饰方法和修饰代码块效果相同，都是作用于调用对象，可相互改写

一个线程访问对象的synchronized代码块时，其他线程可访问该对象的非同步代码块而不受阻塞。

synchronized关键字不能被继承，父类中某个方法使用了synchronized关键字，子类覆盖该方法，默认不同步，仍需显示加上synchronized关键字。子类调用父类的同步方法是同步的。

底层实现原理：同步代码块使用monitorenter和monitorexit指令实现，同步方法依靠常量池中的ACC\_SYNCHRONIZED标识符，是隐式的monitor实现，调用方法时将会检查该标识符是否被设置，若设置了，执行线程将先获取monitor，获取成功后才能执行方法体，方法执行完后再释放monitor。

需要注意的是，任何对象都有一个monitor与之相关联，获取该对象的锁指的是占用该对象的monitor，执行monitor enter指令时，将会尝试获取该对象的monitor，每个monitor enter之后都一定会有两个monitorexit指令，第二个monitorexit是为了保证代码异常退出时也能释放对象的monitor。如果monitor的进入数为0，则该线程进入monitor，将进入数设为1，该线程为monitor的所有者。

volatile关键字：

是程度较轻的synchronized，效率更高。synchronized具有互斥性和可见性，volatile只具有可见性：当一个线程修改了某个共享变量的值，其他线程能够立刻看到修改后的值。

在Java虚拟机中，变量的值保存在主内存中，但是，当线程访问变量时，它会先获取一个副本，并保存在自己的工作内存中。如果线程修改了变量的值，虚拟机会在某个时刻把修改后的值回写到主内存，但是，这个时间是不确定的！

使用volatile关键字，使得每次访问变量时，总是获取主内存的最新值；每次修改变量后，立刻回写到主内存。

synchronized：线程加锁时，将清空工作内存共享变量的值，从主内存读取最新值；线程解锁时，将共享变量最新值刷新到主内存中。

transient关键字：

被修饰的变量不再是对象持久化的一部分，该变量内容在序列化后无法获得访问，也就是说生命周期仅存于调用者的内存中，而不会写到磁盘里持久化。transient变量反序列化的结果是null。

transient只能修饰成员变量和类变量，不能修饰局部变量、方法和类。静态变量不管是否被transient修饰，都不能被序列化，静态变量反序列化的值是当前jvm中对应static变量的值，不由jvm反序列化得出。

final关键字

修饰变量：表示数据不可变，是一个常量。修饰引用变量时，只保证地址不变，对象的值可变。

修饰方法：方法不能被覆盖

修饰类：类无法继承

ThreadLocal线程局部变量

维护了一组线程局部变量，对应共享变量，为每个线程维护该变量的一个独立副本，每个线程都可以独立使用，将变量作用范围限制在线程上下文中，不会与其他线程副本冲突，保护线程安全。ThreadLocal实例通常是private static类型，减少同一线程内多个函数或组件间公共变量传递的复杂性。

用途：比如线程池连接，保存每个线程的connection对象，以保证每个线程从连接池中获得的都是自己的connection。也可用于传参。

常用方法：get、set、remove

底层实现：每个线程维护一个ThreadLocalMap映射表，key是ThreadLocal实例本身，value是需存储的线程共享变量。调用get方式时，先获取当前线程的ThreadLocalMap映射表，根据ThreadLocal实例获取对应的值。

Java中的线程通信的方式有如下：

1. volatile关键字 实现共享变量

2. Object类的wait（） notify（）notifyAll（）方法

3. CountDownLatch 并发组件 中的wait（） 和down（）方法

4. ReentrantLock和Condition 结合使用

5. LockSupport 类中的park（）和unpark（）方法

可重入锁：能被同一个线程反复获取的锁。ReentrantLock比直接使用synchronized更安全，可以使用trylock()尝试获取锁，未获取到不会无限等待进入死锁状态。必须先获取到锁，再进入try {...}代码块，最后使用finally保证释放锁；可重入锁使用Condition对象的await和signal/signalAll来实现wait和notify的功能

ReadWriteLock：把读写操作分别用读锁和写锁来加锁，在读取时，多个线程可以同时获得读锁，这样就大大提高了并发读的执行效率，适用于读多写少的场景。StampedLock提供了乐观读锁，可取代ReadWriteLock以进一步提升并发性能。

3.5 java各种锁实现和使用场景

3.7 9.synchronized和lock分别在什么情况下使用，使用的理由？扩展：synchronized锁升级过程

3.27 Java中的thread.join()的效果

3.28 创建线程的两种方式，区别

3.29 Thread类中的start()和run()方法有什么区别

3.30 怎么检测一个线程是否持有对象监视器

3.31 Runnable和Callable的区别

3.32 什么导致线程阻塞 wait(),notify()和suspend(),resume()之间的区别

3.33 怎么唤醒进程

3.34 线程运行异常怎么办

3.35 FutureTask

3.36 ThreadLocal

3.37 java线程安全的机制有哪些。

3.38 JDK的lock类如何实现公平锁和非公平锁的。

3.39 线程池几个重要的参数，底层怎么实现的

3.40 多线程的实现方式/多线程写入一个文件（代码编程）

多线程创建方式，有什么特点

3.42 synchronized和volatile的区别

## ReentrantLock和synchronized有什么区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的:ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word.

1. 锁的实现： synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。

2. 性能：新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等，synchronized 与 ReentrantLock 大致相同。

3. 等待可中断：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。

ReentrantLock 可中断，而 synchronized 不行。

4. 公平锁：公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。

synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的。

5. 锁绑定多个条件：一个 ReentrantLock 可以同时绑定多个 Condition 对象。

## JVM对synchronized的优化

自旋锁：思想是让一个线程在请求一个共享数据的锁时执行忙循环（自旋）一段时间，如果在这段时间内能获得锁，就可以避免进入阻塞状态。自旋锁虽然能避免进入阻塞状态从而减少开销，但是它需要进行忙循环操作占用 CPU 时间，它只适用于共享数据的锁定状态很短的场景。

锁消除：锁消除是指对于被检测出不可能存在竞争的共享数据的锁进行消除。

锁粗化：如果一系列的连续操作都对同一个对象反复加锁和解锁，而频繁的加锁操作就会导致性能损耗，编译器将会把加锁的范围扩展（粗化）到整个操作序列的外部。

轻量级锁：JDK 1.6 引入了偏向锁和轻量级锁，从而让锁拥有了四个状态：无锁状态（unlocked）、偏向锁状态（biasble）、轻量级锁状态（lightweight locked）和重量级锁状态（inflated）。轻量级锁是相对于传统的重量级锁而言，它使用 CAS 操作来避免重量级锁使用互斥量的开销。对于绝大部分的锁，在整个同步周期内都是不存在竞争的，因此也就不需要都使用互斥量进行同步，可以先采用 CAS 操作进行同步，如果 CAS 失败了再改用互斥量进行同步。

偏向锁：思想是偏向于让第一个获取锁对象的线程，这个线程在之后获取该锁就不再需要进行同步操作，甚至连 CAS 操作也不再需要。

3.44 Spring

3.45 Spring boot的优势、配置情况

3.46 springboot和springcloud的差异

3.47 IOC和AOP的理解 AOP底层实现

3.48 Spring中事务传播的各种情况

3.49 Spring如何解决循环依赖

3.50 SpringCloud的各种组建

3.51 Spring启动流程， SpringBoot启动流程

3.52 SpringMVC的工作流程

3.53 SpringBoot的SPI机制是如何实现的

# 7. cpp

# 8. 大数据平台

Hadoop Mapreduce spark hive hbase

# 9. 设计模式

设计模式：一种编程思想，依据前人在处理类似问题时经验

## 9.1 单例模式

一个类只能被实例化一次，只有一个对象。

单例模式的三种实现方式：

1 懒汉式：

public class Singleton{

private static Singleton singleton;

private Singleton(){}

private static Singleton getInstance(){

if (null==singleton) {singleton = new Singleton();}

return singleton;

}

}

优点：延迟加载（类加载了，对象未被实例化）；缺点：无synchronized锁，线程不安全，多线程中容易出现不同步情况。

2 饿汉式：和懒汉式相比，无延迟加载，但线程安全

public class Singleton{

private static Singleton singleton = new Singleton();

private Singleton(){}

private static Singleton getInstance(){

return singleton;

}

}

3 枚举式：最简洁有效

public enum Singleton{

singleton;

}

## 9.2 工厂模式

使用场景：需要频繁创建和销毁的对象，创建耗时过多的对象，资源共享的情况（数据库连接池、日志）

将创建对象时的创建逻辑隐藏，使用共同的类或接口来创建对象

1. 简单工厂模式

抽象产品类（接口）、具体产品类、工厂类

在工厂类中判断需要实例化的具体商品类

缺点：不够灵活，当添加一个具体的产品类是，需要修改工厂类的逻辑

2. 工厂方法模式

是简单工厂模式的进一步抽象化和推广，工厂方法模式例不再只由一个工厂类决定哪一个产品类，交友抽象工厂的子类去做。

抽象产品类（接口）、具体产品类、抽象工厂类（接口）、具体工厂类

3. 抽象工厂模式

工厂方法模式的扩充，工厂方法模式通常是一个抽象产品类，抽象工厂模式存在一系列相关的抽象产品类。

使用场景：基类包含共有操作，多个具体类实现该基类，就可以使用工厂模式得到各基类的对象

## 9.3 观察者模式

也称发布-订阅模式，对象间存在一对多关系时，比如当一个对象被修改时，则会自动通知依赖于它的所有对象。

观察者，可以当作一个监听者，一旦被观察/监听的目标发生变化，就会通知订阅者。比如微博订阅功能。

## 9.4 Builder模式

使用多个简单的对象一步一步构建一个复杂的对象

含有多个可选对象的构造器就可以使用builder模式

## 9.5 策略模式

针对一个类似（相关）算法，每个算法都是一个实现类，并实现同一接口，在使用时根据环境选择需要使用的策略。（比如根据会员等级打不同的折扣）

## 9.6 适配器模式

作为两个不兼容接口之间的桥梁。适配器必须在需要的时候使用，过多的使用会让系统结构变得混乱。

# 10. 数据结构

<https://github.com/CyC2018/CS-Notes>

<https://github.com/Snailclimb/JavaGuide>

<https://www.cnblogs.com/shiyh/p/12913966.html>