TCP：
报文格式
为何保留部首长度：因为TCP部首长度可变，序列号：计算ISN
M+F(SIP,DIP,SPORT,DPORT)
收到窗口大小为0的应答引发死锁(窗口不为0丢失)：隔一段时间后再询问窗口大小
为何udp需要包长：部首长度需要是4的整数倍
TCP
三次握手：
第三次握手可以携带数据，
为何需要三次握手：
交换序列号
防止历史连接初始化
避免资源浪费（两次握手）
三次握手失败：
服务端：
第三次ACK丢失：那服务端该TCP连接的状态为SYN_RECV,并且会根据 TCP的超时重传机制，会等待3秒、6秒、12秒后重新发送SYN+ACK包，以便重新发送ACK包。 如果重发指定次数之后，仍然未收到客户端的ACK应答，那么一段时间后，服务端自动关闭这个连接
SYN阶梯5次重试
客户端：
那么如果 第三次握手中的ACK包丢失的情况下，Client 向 server端发送数据，Server端将以RST包响应，方能感知到Server的错误。

TCP粘包，拆包：
拆包：大于mss，
粘包：多次写入缓冲区的数据一次发送

消息定长：设置MSS
设置消息边界：（换行符）
消息头中添加包长

拆包：IP分片丢失，整个ip都需要重传，需要等待超时才重传，没有效率， 需要协商MSS

快速重传，重传SACK,选择发送丢失的数据，sack回传已接受到的数据

sillywindow 问题：
接收方：当「窗口大小」小于 min(MSS，缓存空间/2)，也就是小于 MSS 与 1/2 缓存大小中的最小值时，就会向发送方通告窗口为 0，也就阻止了发送方再发数据过来。

等到接收方处理了一些数据后，窗口大小 >= MSS，或者接收方缓存空间有一半可以使用，就可以把窗口打开让发送方发送数据过来。

发送方：
- 要等到窗口大小 >= MSS 或是 数据大小 >= MSS
- 收到之前发送数据的 ack 回包

SYN 攻击：

1.修改内核参数，控制队列大小，超出连接限制后直接发送RST包

2.设置syn cookie 不进入syn队列，计算cookie，直接连接

四次挥手：

2MSL 最大报文生存时间，确保所有报文消亡，

1.防止旧的数据包被收到，引发错乱

2.确保连接正确关闭，（如果不是，再次发起连接会被回RST）

优化time_wait

1.打开timestamp，客户端与服务端时间不匹配回被丢弃

客户端故障：TCP保活，客户端崩溃重启后，服务端的探测报文会被回应RST，得知TCP被重置

TCP参数：
控制SYN重发次数
服务端优化:
是否重发RST accept队列满
绕过三次握手：TCP fast open 无需在第三次握手时发送数据，在第一次三次握手后，直接第一次握手发送数据

四次挥手：
异常退出：发送RST
调整孤儿连接的最大数目
timewait过多直接关闭连接，跳过4次挥手
双方同时关闭连接

传输优化：
tcp_window_scaling

滑动窗口：指针：发送窗口，已发送但未被接受，未发送但在处理能力之内，未发送但大小超过接受范围

tcp拥塞：
慢启动
拥塞避免
超时：慢启动 ssthresh = cwnd/2, cwnd = 1
快速重传：cwnd = cwnd/2 ssthredh = cwnd cwnd+=3

无线网络，丢包率高，性能下降
BBR算法：
StartUp 慢启动: 2ln2的增益加速，过程中即使发生丢包也不会引起速率的降低，而是依据返回的确认数据包来判断带宽增长，直到带宽不再增长时就停止慢启动而进入下一个阶段
Drain 排空:排空阶段是为了把慢启动结束时多余的2BDP的数据量清空，此阶段发送速率开始下降，也就是单位时间发送的数据包数量在下降，直到未确认的数据包数量<BDP时认为已经排空
ProbeBW:排空阶段是为了把慢启动结束时多余的2BDP的数据量清空，此阶段发送速率开始下降，也就是单位时间发送的数据包数量在下降，直到未确认的数据包数量<BDP时认为已经排空

Probe RTT: 每过十秒，估计延迟不变，进入延迟探测阶段，只发送极少数量的包估计新的延迟

HTTP：
状态码：
1XX 信息，服务器收到请求，需要请求者继续执行操作
2XX 成功，操作被成功接收并处理
3XX 重定向，需要进一步的操作以完成请求
4XX 客户端错误，请求包含语法错误或无法完成请求
5XX 服务器错误，服务器在处理请求的过程中发生了错误

Http 常见请求头：
通用头，实体头，请求头，相应头
通用头：
Date , cache control, connection(keep alive)

实体头：
content-length 实体长度
content-language 接受语言
content encoding 压缩属性

请求：
Host, referer, Accept, Accept language
响应：
set-cookie, keep-alive
Get 和post的区别：
GET请求提交的数据会在地址栏显示出来，而POST请求不会再地址栏显示出来。GET提交，请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，多个参数用&连接；
POST提交：把提交的数据放置在是HTTP包的包体中。因此，GET提交的数据会在地址栏中显示出来，而POST提交，地址栏不会改变。
HTTP GET请求由于浏览器对地址长度的限制而导致传输的数据有限制。而POST请求不会因为地址长度限制而导致传
POST的安全性要比GET的安全性高。由于数据是会在地址中呈现，所以可以通过历史记录找到密码等关键信息
GET在浏览器回退时是无害的，而POST会再次提交请求。
GET产生的URL地址可以被Bookmark，而POST不可以。
GET请求只能进行url编码，而POST支持多种编码方式。
GET请求在URL中传送的参数是有长度限制的，而POST么有。PUT 和POST方法的区别是，PUT方法是幂等的：连续调用一次或者多次的效果相同（无副作用），而POST方法是非幂等的。除此之外还有一个区别，通常情况下，PUT的URI指向是具体单一资源，而POST可以指向资源集合。
PUT和PATCH都是更新资源，而PATCH用来对已知资源进行局部更新。

分类	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

- GET 用于获取信息，是无副作用的，是幂等的，且可缓存
- POST 用于修改服务器上的数据，有副作用，非幂等，不可缓存

Http 版本：1.0 not keep alive 1.1 keep alive ,管道传输(一次发多个请求)， 缺点：一个点阻塞后面的都会阻塞 Http2: 压缩头：多个相似头会被压缩，采用二进制格式，多路复用 一个连接中并发多个请求或回应，不一定按照顺序， 同域名下的通信都在单个连接上完成， 服务器推送 主动向客户端发送消息 Http3 如果丢包会发生阻塞，http3改为udp传输
Https:
Https请求全过程
Https 验证整数合法性的流程
Token：(header+Uid) sha256+ pkey ==>sig (header+uid+sig ==>token
无状态，可扩展，多平台跨域
Cookie：setcookie(name,value,expire,path,domain,secure) path cookie 的有效范围， secure 是否仅通过https,HTTPonly 禁止js操控cookie
第三方cookie 在网页中植入
跨站脚本xss：js窃取cookie，httponly
跨站请求伪造CSRF
跨域请求cookie CORS

用户态和核心态
为了保护核心态的高级别指令，把用户的指令和机器的指令分开， 用户进程使用调用时，会由用户态转为核心态，使得用户态不会操作系统的内核地址空间
硬件可以中断用户态进入核心态
阻塞I/O
非阻塞I/O 轮询 直到有数据返还，内核拷贝到用户是阻塞的
I/O多路复用：同时监听多个描述符
select 1024个 32bit 2048 64bit 利用fd数组,有限
poll fd链表，最高监听的描述符提升
epoll 红黑树：

传统的select/poll另一个致命弱点就是当你拥有一个很大的socket集合，不过由于网络延时，任一时间只有部分的socket是"活跃"的，但是select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数，其他idle状态socket则不会，在这点上，epoll实现了一个"伪"AIO，因为这时候推动力在os内核。在一些 benchmark中，如果所有的socket基本上都是活跃的---比如一个高速LAN环境，epoll并不比select/poll有什么效率，

- 当我们使用epoll_fd增加一个fd的时候，内核会为我们创建一个epitem实例，讲这个实例作为红黑树的节点，此时你就可以BB一些红黑树的性质，当然你如果遇到让你手撕红黑树的大哥，在最后的提问环节就让他写写吧
- 随后查找的每一个fd是否有事件发生就是通过红黑树的epitem来操作
- epoll维护一个链表来记录就绪事件，内核会当每个文件有事件发生的时候将自己登记到这个就绪列表，然后通过内核自身的文件file-eventpoll之间的回调和唤醒机制，减少对内核描述字的遍历，大俗事件通知和检测的效率

数据： 脏读 读到了未提交事务修改过的数据 不可重复读：事务中的数据发生了改变， 幻读:读到了插入的数据
事务的隔离级别：读未提交,读已提交，可重复读，可串行化
事务的四大特征：
A原子性：全成功、全失败
C一致性：开始和结束后完整性不会破坏
I隔离性：不同事物不相互影响
D持久性 ：修改是永久性的
MVCC 多版本并发控制：
InnoDB MVCC 保存事务ID，保存回滚指针，每开启一个新的事务，递增一个新的事务ID
MVCC只在可重读和读提交两个隔离级别工作 串行化是通过加锁实现
MVCC实现依靠undo log 和read view 读提交每次查询会生成一个readview, 可重复度是只维护当前的readview
MVCC实现的读写不阻塞正如其名：多版本并发控制--->通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot)，并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好像是数据库可以提供同一数据的多个版本。
MVCC使得数据库读不会对数据加锁，普通的SELECT请求不会加锁，提高了数据库的并发处理能力。借助MVCC，数据库可以实现READ COMMITTED，REPEATABLE READ等隔离级别，用户可以查看当前数据的前一个或者前几个历史版本，保证了ACID中的I特性（隔离性)。
Example: read commit 事务A读取了记录(生成版本号)

- 事务B修改了记录(此时加了写锁)
- 事务A再读取的时候，是依据最新的版本号来读取的(当事务B执行commit了之后，会生成一个新的版本号)，如果事务B还没有commit，那事务A读取的还是之前版本号的数据。

可重复读：
串行化加锁： 读读不加锁，其他都加锁
修改操作先写内存中的buffer，同步到redo log中，
binlog:记录了数据库的变更 用来复制和恢复数据 一主多从结构数据库 通过binlog实现
redo log:内存中修改的数据同步到磁盘中 redo log顺序I/O redo log记录的是物理变化
binlog 记录sql语句，redo log 记录的是物理变化 redolog 用作持久化，binlog用作恢复数据库
redo log事务开始的时候开始记录，binlog提交的时候记录 写redo log失败 回滚 写redo log成功 binlog失败 回滚 redo log binlog都成功才会成功
undo log 回滚和MVCC
主从复制：主--》binlog binlog-->从relay log 从--》make change 异步且串行化 master 写binlog 一个线程，slave 读binlog 一个线程，slave 执行relay log 一个线程

长事务运行时间长，长时间未提交的事务，会造成阻塞超时，主从延迟
为何避免长事务：监控长事务并告警

Mysql锁
表锁， 行锁
通过索引加锁，不通过索引检索则对所有行都加锁
表锁 开销小，加锁快 粒度大，适合查询 行级锁：开销大，加锁慢，会出现死锁
InnoDB加锁 自动加锁对UPDATE,INSERT,DELETE SELECT不会自动加锁 in share mode S lock for update 排他锁 锁在commit 或者rollback 的时候释放
共享锁 slock : T 对A加锁， 其他事务可以对A加读锁
排他锁 X lock ： T对X加锁 不许其他事务加锁
意向共享锁: IS lock 事务在请求S前要获得IS锁 可有多个
意向排他锁： IX lock: 请求X前获得IX锁 只能有一个
意向锁：实现行锁和表锁共存并且满足隔离的重要性

锁加的是索引，如果是非聚簇索引，则对应的聚簇索引也要加锁

死锁：在同一资源上互相占用，并请求锁定对方占用的资源 死锁恢复：持有最少行级排他锁回滚 外部通过检测超时时间
加锁算法：
Record lock 单个记录加锁 通过索引 如果没有索引则是给整个表加锁
Gap lock 锁定范围，不包括索引 间隙锁顾名思义锁间隙，不锁记录。锁间隙的意思就是锁定某一个范围，间隙锁又叫 gap 锁，其不会阻塞其他的 gap 锁，但是会阻塞插入间隙锁，这也是用来防止幻读的关键。当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁
Next-key lock: Gap lock + record lock 默认隔离级别REPEATABLE-READ下，InnoDB中行锁默认使用算法Next-Key Lock，只有当查询的索引是唯一索引或主键时，InnoDB会对Next-Key Lock进行优化，将其降级为Record Lock，即仅锁住索引本身，而不是范围。当查询的索引为辅助索引时，InnoDB则会使用Next-Key Lock进行加锁 这个锁本质是记录锁加上gap 锁。在 RR 隔离级别下(InnoDB 默认)，InnoDB 对于行的扫描锁定都是使用此算法，但是如果查询扫描中有唯一索引会退化成只使用记录锁。
在可重复读隔离级别下，innodb默认使用的是next-key lock算法，当查询的索引是主键或者唯一索引的情况下，才会退化为record lock，在使用next-key lock算法时，不仅仅会锁住范围，还会给范围最后的一个键值加一个gap lock。

索引
聚簇索引 叶子节点记录完整的data记录， 聚簇索引
辅助索引 叶子节点记录的是主键的值
哈希索引 key value 仅支持精确查找
全文索引 查找关键词

使用多列索引比多个单列索引性能好，选择性最强的索引放在前面
前缀索引 text,varchar

explain 语句
select_type simple,union,subquery
table
possible key
key
rows

B+ 树 平衡树相比较与B树所有的叶子节点都可以顺序实现，适合范围查找，多路查找树，磁盘I/O小 B+树查询更为稳定 单位节点的数量适合用页的整数倍

红黑树
确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。
红黑树是一种比较宽泛化的平衡树，没AVL的平衡要求高，同时他的插入删除都能在O（lgN）的时间内完成，而且对于其性质的维护，插入至多只需要进行2次旋转就可以完成，对于删除，至多只需要三次就可以完成，所以其统计性能要比AVL树好
旋转操作的次数越少，意味着整棵树的拓扑结构不会频繁的做出大的改变。这一点是AVL树做不到的。
Mysql 页 各个数据页组成双向链表，每个记录都是单项链表 主键查找可通过二分法查找页目录

覆盖索引： 索引上包含字段（主键）
索引下推：在遍历过程中对辅助索引包含字段先判断，减少回表次数

引擎：

InnoDB 和 MyISAM 的比较

- 事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 外键：InnoDB 支持外键。
- 备份：InnoDB 支持在线热备份。
- 崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

Json Contains:

Json contain 和Json contain path

范式：
1. 第一范式（1NF）无重复的列

所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。
2.第二范式（2NF）就是非主属性完全依赖于主关键字。
3.第三范式（3NF）属性不依赖于其它非主属性 [消除传递依赖]

在做范围查找的时候，平衡树比skiplist操作要复杂。在平衡树上，我们找到指定范围的小值之后，还需要以中序遍历的顺序继续寻找其它不超过大值的节点。如果不对平衡树进行一定的改造，这里的中序遍历并不容易实现。而在skiplist上进行范围查找就非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。

平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。

线程和进程调度算法：

1.先来先服务
2.最短作业优先
3.高相应比 (等待时间+要求服务时间)/要求服务时间
4.时间轮片调度 每个任务被分配一个时间轮片，未执行完到队尾
5.最高优先级 （静态，动态（等待时间）） 抢占，非抢占（更高优先级插入需不需要挂起）
6.多级反馈队列调度算法

线程与进程的比较如下：

- 进程是资源（包括内存、打开的文件等）分配的单位，线程是 CPU 调度的单位；
- 进程拥有一个完整的资源平台，而线程只独享必不可少的资源，如寄存器和栈；
- 线程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系；
- 线程能减少并发执行的时间和空间开销；

对于，线程相比进程能减少开销，体现在：

- 线程的创建时间比进程快，因为进程在创建的过程中，还需要资源管理信息，比如内存管理信息、文件管理信息，而线程在创建的过程中，不会涉及这些资源管理信息，而是共享它们；
- 线程的终止时间比进程快，因为线程释放的资源相比进程少很多；
- 同一个进程内的线程切换比进程切换快，因为线程具有相同的地址空间（虚拟内存共享），这意味着同一个进程的线程都具有同一个页表，那么在切换的时候不需要切换页表。而对于进程之间的切换，切换的时候要把页表给切换掉，而页表的切换过程开销是比较大的；
- 由于同一进程的各线程间共享内存和文件资源，那么在线程之间数据传递的时候，就不需要经过内核了，这就使得线程之间的数据交互效率更高了；

所以，线程比进程不管是时间效率，还是空间效率都要高。

进程间的通信：管道 父进程fork子进程，在管道的两端，可以进行通信，不适合频繁的交互数据，消息队列可解决此问题， 共享内存

- 一个是 P 操作，这个操作会把信号量减去 -1，相减后如果信号量 < 0，则表明资源已被占用，进程需阻塞等待；相减后如果信号量 >= 0，则表明还有资源可使用，进程可正常继续执行。
- 另一个是 V 操作，这个操作会把信号量加上 1，相加后如果信号量 <= 0，则表明当前有阻塞中的进程，于是会将该进程唤醒运行；相加后如果信号量 > 0，则表明当前没有阻塞中的进程；

信号：
kill -l 查看所有信号

- Ctrl+C 产生 SIGINT 信号，表示终止该进程；
- Ctrl+Z 产生 SIGTSTP 信号，表示停止该进程，但还未结束；

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，一旦有信号产生，我们就有下面这几种，用户进程对信号的处理方式。

Redis

- 完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。它的，数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)；

- 数据结构简单，对数据操作也简单，Redis中的数据结构是专门进行设计的；
- 采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；
- 使用多路I/O复用模型，非阻塞IO；

RDB做镜像全量持久化，AOF做增量持久化。因为RDB会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要AOF来配合使用。在redis实例重启时，会使用RDB持久化文件重新构建内存，再使用AOF重放近期的操作指令来实现完整恢复重启之前的状态。
取决于AOF日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据
RDB
操作系统多进程 COW(Copy On Write) 机制 拯救了我们。Redis 在持久化时会调用 glibc 的函数 fork 产生一个子进程，简单理解也就是基于当前进程 复制 了一个进程，主进程和子进程会共享内存里面的代码块和数据段：所以 快照持久化 可以完全交给 子进程 来处理，父进程 则继续 处理客户端请求。子进程 做数据持久化，它 不会修改现有的内存数据结构，它只是对数据结构进行遍历读取，然后序列化写到磁盘中。但是 父进程 不一样，它必须持续服务客户端请求，然后对 内存数据结构进行不间断的修改。
Redis数据结构：
String
List
Hash
Set
Zset
跳跃表
首先，因为 zset 要支持随机的插入和删除，所以它 不宜使用数组来实现，关于排序问题，我们也很容易就想到 红黑树/ 平衡树 这样的树形结构，为什么 Redis 不使用这样一些结构呢？

1. 性能考虑： 在高并发的情况下，树形结构需要执行一些类似于 rebalance 这样的可能涉及整棵树的操作，相对来说跳跃表的变化只涉及局部 (下面详细说)；
2. 实现考虑： 在复杂度与红黑树相同的情况下，跳跃表实现起来更简单，看起来也更加直观；

为了避免这一问题，它不要求上下相邻两层链表之间的节点个数有严格的对应关系，而是 为每个节点随机出一个层数(level)。比如，一个节点随机出的层数是 3，那么就把它链入到第 1 层到第 3 层这三层链表中。
Bloom Filter:
布隆过滤器的原理是，当一个元素被加入集合时，通过K个散列函数将这个元素映射成一个位数组中的K个点，把它们置为1。检索时，我们只要看看这些点是不是都是1
缓存穿透
缓存穿透。产生这个问题的原因可能是外部的恶意攻击，例如，对用户信息进行了缓存，但恶意攻击者使用不存在的用户id频繁请求接口，导致查询缓存不命中，然后穿透 DB 查询依然不命中。这时会有大量请求穿透缓存访问到 DB。
解决的办法如下。

1. 对不存在的用户，在缓存中保存一个空对象进行标记，防止相同 ID 再次访问 DB。不过有时这个方法并不能很好解决问题，可能导致缓存中存储大量无用数据。
2. 使用 BloomFilter 过滤器，BloomFilter 的特点是存在性检测，如果 BloomFilter 中不存在，那么数据一定不存在；如果 BloomFilter 中存在，实际数据也有可能会不存在。非常适合解决这类的问题。

删除困难。一个放入容器的元素映射到bit数组的k个位置上是1，删除的时候不能简单的直接置为0，可能会影响其他元素的判断。可以采用

缓存击穿

缓存击穿，就是某个热点数据失效时，大量针对这个数据的请求会穿透到数据源。
解决这个问题有如下办法。

1. 可以使用互斥锁更新，保证同一个进程中针对同一个数据不会并发请求到 DB，减小 DB 压力。
2. 使用随机退避方式，失效时随机 sleep 一个很短的时间，再次查询，如果失败再执行更新。
3. 针对多个热点 key 同时失效的问题，可以在缓存时使用固定时间加上一个小的随机数，避免大量热点 key 同一时刻失效。

缓存雪崩

缓存雪崩，产生的原因是缓存挂掉，这时所有的请求都会穿透到 DB。
解决方法：

1. 使用快速失败的熔断策略，减少 DB 瞬间压力；
2. 使用主从模式和集群模式来尽量保证缓存服务的高可用。

实际场景中，这两种方法会结合使用。

Java 提纲

8大基础数据类型 int,short,long,float,double,byte,char,boolean

equals 默认判断引用是否相等， 在string里有重写
== 判断值，再判断引用
hashcode 对对象进行hash

int 和Integer 的区别：
Integer是int的包装类；int是基本数据类型；
Integer变量必须实例化后才能使用；int变量不需要；
Integer实际是对象的引用，指向此new的Integer对象；int是直接存储数据值 ；
Integer的默认值是null；int的默认值是0。
泛型不支持int，但是支持Integer
int 存储在栈中，Integer 对象的引用存储在栈空间中，对象的数据存储在堆空间中

重写equals必须重写hashcode, 否则违反了equal 相等hashcode必须相等的原则

string 为final 类型，不可以被继承
stringbulder 线程不安全，效率高
stringbuffer 线程安全

创建对象的几种方式：
new, clone，反序列化，使用反射 class.newinstance / 构造器， UNsafe，跳过构造器创建对象

抽象类和接口的区别
抽象类可以有普通方法，接口必须抽象
抽象类可以有普通成员变量，接口类中只能有static final 的变量

java 内部类：内部类可以访问该类域内的所有数据，可以对同一包中其他类隐藏起来，内部类可以解决单继承的缺陷

深拷贝浅拷贝：
浅拷贝拷贝指针
深拷贝拷贝对象
通过序列化和反序列化拷贝对象

Java 反射机制
在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。
反射的核心是JVM在运行时才动态加载类或调用方法/访问属性，它不需要事先知道运行对象是谁
wait() 和 sleep()的区别
wait()是对象的方法，释放锁，sleep()是thread的方法，释放锁

有序的set：linkedset treeset

ArrayList 和LinkList的区别

Java 三大特征：
封装，继承，多态
重写 对父类方法重写
重载：有多个名字相同变量不同的函数
多态的条件：1）继承；2）重写；3）向上转型
用处：可以屏蔽不同子类对象之间的实现差异，从而写出通用的代码达到通用编程

类加载
加载 连接 初始化
连接：》验证 准备 解析

加载：
类的装载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，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。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。
类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误。
验证的目的是为了确保Class文件中的字节流包含的信息符合当前虚拟机的要求，而且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：文件格式的验证、元数据的验证、字节码验证和符号引用验证。

准备：1）这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。
2）这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。
4. 解析
解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。
5. 初始化
①声明类变量时指定初始值
②使用静态代码块为类变量指定初始值

JVM
1、堆是线程共享的内存区域，栈是线程独享的内存区域。
2、堆中主要存放对象实例，栈中主要存放各种基本数据类型、对象的引用。

jvm包括两种数据类型，基本类型和引用类型。
引用类型包括三种，类类型，数组类型，和接口类型。
堆

jvm有一个堆，在所有jvm线程间共享，堆是一个运行时数据区域，所有为类实例和数组分配的内存都来自于它。

堆在jvm启动时创建，堆中对象不用显式释放，gc会帮我们释放并回收内存。
方法区

jvm有一个方法区，在所有jvm线程间共享，它存储每一个类的结构。

像运行时常量池，字段和方法数据，方法和构造函数的代码，还有特殊的方法用于类和实例的初始化，以及接口的初始化。

方法区在jvm启动时创建，虽然方法区在逻辑上是堆的一部分。

但简单实现时可以选择不进行gc和压缩，本规范没有强制要求方法区的位置，也没有要求管理已编译代码的策略。
jvm栈

每一个jvm线程都有一个私有的jvm栈，随着线程的创建而创建，栈中存储的是帧。

jvm栈和传统语言如C的栈相似，保存局部变量和部分计算结果，参与方法的调用和返回。jvm栈主要用于帧的出栈和入栈，除此之外没有其它操作，

帧可能是在堆上分配的，所以jvm栈使用的内存不必是连续的。

native方法栈

native方法不是用Java语言写的，为了支持它需要使用传统栈，如C语言栈。不过jvm不能加载native方法，所以也不需要提供native方法需要的栈。

GC
垃圾识别：引用计数法
可达性算法(GC Root 为起点)对象的 finalize 方法给了对象一次垂死挣扎的机会，当对象不可达（可回收）时，当发生GC时，会先判断对象是否执行了 finalize 方法，如果未执行，则会先执行 finalize 方法，我们可以在此方法里将当前对象与 GC Roots 关联，这样执行 finalize 方法之后，GC 会再次判断对象是否可达，如果不可达，则会被回收，如果可达，则不回收！

垃圾回收：
分代回收：

新生代 Eden, S0, S1
Eden区即将满的时候触发Minor gc
存活的对象移动到S0区，对象的年龄+1，

下一次Minor GC 的时候会把S0 和S1存活的对象一起到S1区，如果再下一次则是 S1,S0功能互换

老年代
对象的年龄超过阈值的时候，会从新生代晋升到老年代
永久代

Thread的状态：
新建、就绪、运行、阻塞、死亡

HashMap
1.7 头插法，不安全，可能造成循环链表
1.8 尾插法，不会造成循环链表，更加安全，增加的红黑树
扩容：0.75，rehash
初始值为2的n次幂 方便进行位运算，比取模运算快

HashTable：效率低下，所有操作都会上锁，key不可位null
这样会有一个问题，当你通过get(k)获取对应的value时，如果获取到的是null时，你无法判 它是put（k,v）的时候value为null

ConcurrentHashMap
由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。
1.7 分为segment分块上锁
1.8 给数组的头加上锁，是synchroized 锁
1.8 put 为cas+synchronized 形式
自旋锁与互斥锁的区别：

- 互斥锁加锁失败后，线程会释放 CPU ，给其他线程；
- 自旋锁加锁失败后，线程会忙等待，直到它拿到锁；

可重入锁：

“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

Synchronized
synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

Synchronized 锁升级：无锁，偏向锁，轻量级锁，重量级锁
偏向锁：没有额外的性能消耗，相同线程获取效率高，多线程竞争会进行锁升级，采用CAS操作
轻量级锁：默认锁持有时间不长，用自旋获取锁

Synchronized 与 Reentrantlock都是可重入锁，Reentrantlock 可实现公平锁，可等待中断

synchronized 与 lock相比不够灵活

- 效率低：锁的释放情况少，只有代码执行完毕或者异常结束才会释放锁；试图获取锁的时候不能设定超时，不能中断一个正在使用锁的线程，相对而言，Lock可以中断和设置超时
- 不够灵活：加锁和释放的时机单一，每个锁仅有一个单一的条件(某个对象)，相对而言，读写锁更加灵活
- 无法知道是否成功获得锁，相对而言，Lock可以拿到状态，如果成功获取锁，....，如果获取失败，.....

Volatile
Volatile强行指定每次读写都到主存中而不是工作内存中，同时防止指令重排序，写happens before 读

- volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。
- 多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞
- volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。
- volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。

Cas 乐观锁：

线程在读取数据时不进行加锁，在准备写回数据时，先去查询原值，操作的时候比较原值是否修改，若未被其他线程修改则写回，若已被修改，则重新执行读取流程。
是因为CAS操作长时间不成功的话，会导致一直自旋，相当于死循环了，CPU的压力会很大。
ABA问题：

因为CAS需要在操作值的时候，检查值有没有发生变化，比如没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时则会发现它的值没有发生变化，但是实际上却变化了。
ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A->B->A就会变成1A->2B->3A。
自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令，那么效率会有一定的提升。

Thread local:
如果你创建了一个ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是ThreadLocal变量名的由来。他们可以使用 get（） 和 set（） 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。 线程ThreadlocalMap存放值， 内存泄露： weakreference的问题 key为弱引用，value是强引用，key会被GC清理掉，value则不会，最后最好调用remove方法

线程池：

- 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

FixedThreadPool

固定大小的线程池，linkedBlockingQueue 无界队列

SingleThreadExecutor

linkedBlockingQueue 无界队列
只创建一条线程处理任务

CachedThreadPool

无线扩大的线程池，适合处理时间比较小的任务，采用SynchronizedQueuec,没有储存空间，只要请求到来就会执行

ScheduledThreadPool

处理延时任务或者定时任务、

Executors 返回线程池对象的弊端如下：

- FixedThreadPool 和 SingleThreadExecutor ：允许请求的队列长度为 Integer.MAX_VALUE ，可能堆积大量的请求，从而导致OOM。
- CachedThreadPool 和 ScheduledThreadPool ：允许创建的线程数量为 Integer.MAX_VALUE ，可能会创建大量线程，从而导致OOM。

ThreadPoolExecutor:

- corePoolSize : 核心线程数线程数定义了最小可以同时运行的线程数量。
- maximumPoolSize : 当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。

- workQueue: 当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，信任就会被存放在队列中。
- keepAliveTime:当线程池中的线程数量大于 corePoolSize 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了keepAliveTime才会被回收销毁；
- unit : keepAliveTime 参数的时间单位。
- threadFactory :executor 创建新线程的时候会用到。
- handler :饱和策略。关于饱和策略下面单独介绍一下。

- ThreadPoolExecutor.AbortPolicy：抛出 RejectedExecutionException来拒绝新任务的处理。
- ThreadPoolExecutor.CallerRunsPolicy：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- ThreadPoolExecutor.DiscardPolicy： 不处理新任务，直接丢弃掉。
- ThreadPoolExecutor.DiscardOldestPolicy： 此策略将丢弃最早的未处理的任务请求。

阻塞队列：

ArrayBlockingQueue

ArrayBlockingQueue是一个用数组实现的有界阻塞队列。此队列按照先进先出的原则对元素进行排序
参数fair用于设置线程是否公平访问队列。所谓公平访问是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列。非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可以争夺访问队列的资格，有可能先阻塞的线程最后才访问队列。为了保证公平性，通常会降低吞吐量。

LinkedBlockingQueue

LinkedBlockingQueue是一个用链表实现的有界阻塞队列。此队列的默认和最大长度为Integer.MAX_VALUE。此队列按照先进先出的原则对元素进行排序。

PriorityBlockingQueue

PriorityBlockingQueue是一个支持优先级的无界阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来进行排序。需要注意的是不能保证同优先级元素的顺序。

DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityBlockingQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。
DelayQueue运用在以下应用场景：

- 缓存系统的设计：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。

- 任务超时处理：比如下单后15分钟内未付款，自动关闭订单。

JUC Atomic原子类：
public final int get() //获取当前的值
public final int getAndSet(int newValue)//获取当前的值，并设置新的值
public final int getAndIncrement()//获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输入值（update）
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。
CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。