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 提示
200 ok 204 non content 206 partial content
301 moved permanently 302 move temporary 304 not modified
400 bad request 403 forbidden 404 non found
500 internal error 501 not implemented 502 bad gateway 503 service unavailable

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。

python