# 数据库三范式

一范式：每列或者说每个字段都不可分解，具有原子性

二范式：非主键列要完全依赖主键列，不能只是依赖主键的一部分。即表必须有主键，且没有包含在主键中的列必须完全依赖于主键

（防止出现行冗余，避免出现多行重复数据）

三范式：非主键列要直接依赖主键列，而不是通过依赖非主键列间接依赖主键列。

（防止出现列冗余，避免出现无关的列，无关的多列信息应放在其他表中）

索引数据结构：

B+-树：所有叶子节点在同一层上

B-树：

节点即存储索引也存储数据

叶子节点之间没有指针指向

B+树：

非叶子节点只存储索引，不存储数据，可有更多索引被加载到内存中，减少磁盘IO，非叶子节点也可称作索引节点

叶子节点存储数据或者数据指针，且叶子节点之间有指针指向，方便区间查询

MYSQL：

索引B+树（使用），Hash(对区间查询不友好)

聚簇索引：索引和数据存放在一起，同一个文件中。

非聚簇索引：索引和数据存分开存放，存放在不同的文件中。

MYSQL中的两种存储引擎：

InnoDB属于聚簇索引，MYISAM属于非聚簇索引

InnoDB存储引擎存储数据时数据文件本身就是按B+树索引结构组织的，所以必须要有主键作为索引key值依据，若没有手动指定主键，数据库后台会自动选择一个唯一标识的字段建立主键；若表中没有唯一标识的列，则在表中默认添加一列作为主键字段，维护索引+数据文件。

SQL慢查询：

开启慢查询日志slow-log-log=1并设置慢查询时间阈值long qwuery time=xxx，设置慢查询日志文件slow-query-log-file=xxx。当SQL语句执行时间超过指定的阈值时记录查询日志。

慢查询优化：优化在某些查询速度特别慢（慢查询日志中执行时间大于一定程度的SQL查询语句）

SQL语句优化

索引优化

事务特性ACID

1. 原子性 ：事务里面的操作单元不可切割,要么全部成功,要么全部失败
2. 一致性 ：事务执行前后,业务状态和其他业务状态保持一致.
3. 隔离性 ：一个事务执行的时候最好不要受到其他事务的影响
4. 持久性 ：一旦事务提交或者回滚.这个状态都要持久化到数据库中

脏读:在一个事务中读取到另一个事务没有提交的数据  
不可重复读:在一个事务中,两次查询的结果不一致(针对的update操作)  
虚读(幻读):在一个事务中,两次查询的结果不一致(针对的insert操作)

read uncommitted 读未提交 上面的三个问题都会出现  
read committed 读已提交 可以避免脏读的发生  
repeatable read 可重复读 可以避免脏读和不可重复读的发生  
serializable 串行化 可以避免所有的问题

MVCC-多版本并发控制

MVCC-多版本并发控制，是一种并发控制方式，实现对数据库的并发访问控制，主要是为了提高数据库并发性能，实现读-写冲突不加锁，这里的读指的是快照读。维持一个数据的多个版本，使得读写操作没有冲突。

MVCC实现的关键是 ：

1. 表的隐藏字段 事ID 回滚指针
2. undo log 记录历史版本链
3. readView

**执行更新操作时，**会将旧数据行记录在undo log中并将数据表中当前记录更新，事务ID更新为当前操作的事务ID，回滚指针指向undo log中的旧记录。

**执行查询操作时，**当前事务会生成一个ReadVew，这个ReadView相当于是一个事物快照，即当前系统内活跃的事物列表，即系统内所有已开始但还未提交的事务，当前事务会根据这个ReadView去判断哪些数据可见，哪些数据不可见。

查询一条数据时，事务根据这个ReadView，到undo log中判断，

1. 先查看undo log中当前记录最新数据行，若该数据行事务版本号小于readView中的事务最小ID，说明该事务已提交，对于当前数据库是可见的，可直接作为结果返回
2. 若最新数据行的事务版本号大于readView中事务最大ID，说明该事务是新事务，对当前事务不可见，要继续顺着undo log版本链继续往下寻找直到满足条件的
3. 若当前数据行的事务版本号 存在 readView列表中，则需要遍历整个readView，若数据行的版本号等于readView列表中某个事务ID，说明当前数据行还处于活动状态，对该事务不可见，遍历undo log版本链，直到找到数据行的事务ID小于readView中最小事务ID的记录。

读已提交

事务中每次读操作都会生成一个readView，若事务执行期间某个事务提交了，该事务ID将会从readView中移除，这样确保事务每次读操作都能读到已提交的新数据

可重复读

事务对数据行执行第一次读操作时生成ReadView，后续的读操作重复使用这个readView，计算在这个期间有别的事务提交更新，该更新也是不可见的。

**也就是说已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView,而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView。**

MYSQL的MVCC，通过版本链，实现多版本控制，可并发读-写，写-读，通过ReadView生成策略的不同实现不同的隔离级别。

SQL注入

网络七层/五层模型各层的作用

应用层，应用服务的协议，文件传输、文件服务、HTTP等

表示层，数据格式化、数据加密、转换等

会话层，解除或建立与其他结点的联系

传输层，提供端对端的传输端口或接口

网络层，为数据包的传输选择合适的路由，最短路径优先

数据链路层，传输有地址的帧并检测差错

物理层，以二进制数据形式在物理媒介上传输数据

**SQL语句GROUP BY：**

“Group By”从字面意义上理解就是根据“By”指定的规则对数据进行分组，所谓的分组就是将一个“数据集”划分成若干个“小区域”，然后针对若干个“小区域”进行数据处理

select 类别, sum(数量) as 数量之和

from A

group by 类别

select 类别, sum(数量) AS 数量之和

from A

group by 类别

order by sum(数量) desc

**Group By中Select指定的字段限制**

在select指定的字段要么就要包含在Group By语句的后面，作为分组的依据；要么就要被包含在聚合函数中。

**Group By与聚合函数**

group by语句中select指定的字段必须是“分组依据字段”，其他字段若想出现在select中则必须包含在聚合函数中，常见的聚合函数如下表：

| **函数** | **作用** | **支持性** |
| --- | --- | --- |
| sum(列名) | 求和 |  |
| max(列名) | 最大值 |  |
| min(列名) | 最小值 |  |
| avg(列名) | 平均值 |  |
| first(列名) | 第一条记录 | 仅Access支持 |
| last(列名) | 最后一条记录 | 仅Access支持 |
| count(列名) | 统计记录数 | 注意和count(\*)的区别 |

* where 子句的作用是在对查询结果进行分组前，将不符合where条件的行去掉，即在分组之前过滤数据，where条件中不能包含聚组函数，使用where条件过滤出特定的行。
* having 子句的作用是筛选满足条件的组，即在分组之后过滤数据，条件中经常包含聚组函数，使用having 条件过滤出特定的组，也可以使用多个分组标准进行分组。

微服务架构：将一个单一的应用程序开发为一组小型服务的方法，每个服务都独立运行在自己的进程中，服务间通信采用“轻量级“通信机制，通常使用http资源的API来实现。这些服务围绕业务能力构建并且可通过全自动部署机制独立部署，这些服务共用一个最小的集中式的管理，服务可用不同的语言开发，使用不同的数据存储技术，即不受技术栈的限制。

微服务特征：

1. 每个服务独立运行在自己的进程中
2. 一系列运行的微服务共同构建起整个系统
3. 每个微服务可独立开发，开发过程中只要关注一个当前服务业务模块的特定功能，如支付服务、订单服务、用户管理服务等。
4. 可使用不同的技术栈开发不同的服务
5. 微服务之间直接通过轻量级的通信机制进行通信，如Rest API
6. 全自动的部署机制

微服务适用场景：

1. 大型项目
2. 有快速迭代的要求
3. 访问压力过大的网站

微服务拆分：

1.依据技术栈驱动设计

2.面向对象驱动设计

按职责划分（订单服务、用户服务、商品服务等）

按通用性划分（把一些通用功能做成微服务，如用户中心、支付在中心、消息中心，一个中台由若干个微服务构成）

Zookper，本质就是一个封装了很多算法的软件，投票推选leader，

一个分布式协调框架。

1. 发布订阅管理，作为注册中心
2. 分布式/集群管理
3. 安装 -> 启动zkServer -> 防火墙设置zkServer 端口通行

# 网络IO模型

## BIO：阻塞IO

Server端在accept和read或write时都是阻塞模式，主线程负责accept客户端发来的连接请求，若没有连接则一直阻塞在accept处，针对某个客户端发来的请求，起线程读取相应socket的数据作处理，若一直没有数据发来，则该线程阻塞在这里一直等待，直到socket中有数据传来。

1. 主线程accpet客户端的连接，每发来一个客户端连接，则创建一个线程来处理，这样客户端发来大量连接的情况下，会有非常多的线程，消耗系统资源；
2. 使用线程池进行优化，每当客户端发来一个连接时，将请求封装成任务发送给线程池调度，线程池管理线程，实现线程的复用，用少量的线程处理很多的客户端请求
3. 但是，用线程池能处理的客户端请求也是有限的

## NIO：非阻塞IO

Server端的线程或进行在处理socket时不再阻塞，而是想方法使用事件通知的机制，当有socket中接收到数据可被读取时通知和处理该socket的程序（线程或进程）去读取。

Select 模型，当有socket中传来数据时，返回一个大于等于1的整数，但是此时只是知道有socket中有数据，并不知道是哪个socket中传来的数据，程序要去遍历所有的socket，找出有数据的socket读取数据。Select模型通过一个数组管理被监控的socket。

Epoll模型，维护一个双端队列-就绪list，链接已经有数据可被处理的socket引用，此时不需要再像select那样遍历所有的socket；管理所有的socket使用一个红黑树结构。

等待队列 和 工作队列 中存放的是进程或线程的引用

## 额外：

网络数据传输到网卡后，网卡会发送一个中断信号给CPU，CPU会去优先处理将这些数据拷贝到内存和socket中~ 硬件有数据产生时会发送一个中断信号给CPU，CPU会优先处理，此种优先级比较高。

# HTTP

## HTTP1.0

无状态连接，每一个请求-响应都要三次握手建立连接->请求响应->断开连接

## HTTP1.1

Keep-Alive可以建立长连接，同一个连接被多个请求响应复用

HTTP1.0和HTTP1.1数据都是串行传输的，文本传输

## HTTP2

多路复用，数据可并行传输，使用帧为数据标记顺序，以二进制流传输，提高资源传输响应。

博客： https://www.cnblogs.com/imteck4713/p/12016313.html

HTTP: 数据明文传输，没有身份认证和加密 80

HTTPS:443

0 服务端Server会在第三方机构CA处提交公钥、组织信息、个人信息等信息并申请认证；CA通过线上、线下等方式验证申请后会对申请者签发认证文件，即证书，证书中包含了申请者公钥、申请者各种其他信息和签发机构的CA信息、有效期等，同时包含签名

1 客户端向服务端发起HTTPS请求时，server会先将自己的CA证书返回给client，client读取证书验证证书的合法性（客户端使用散列函数计算公开的明文信息的摘要，并利用CA的公钥对签名进行解密，对比证书的信息摘要若一致，则说明证书合法，即公钥合法），同时客户端还会验证证书相关的域名信息、有效期等等。客户端会内置信任的CA的证书信息，包含公钥，若CA不被信任，则找不到对应CA的证书，CA证书会被认为非法。

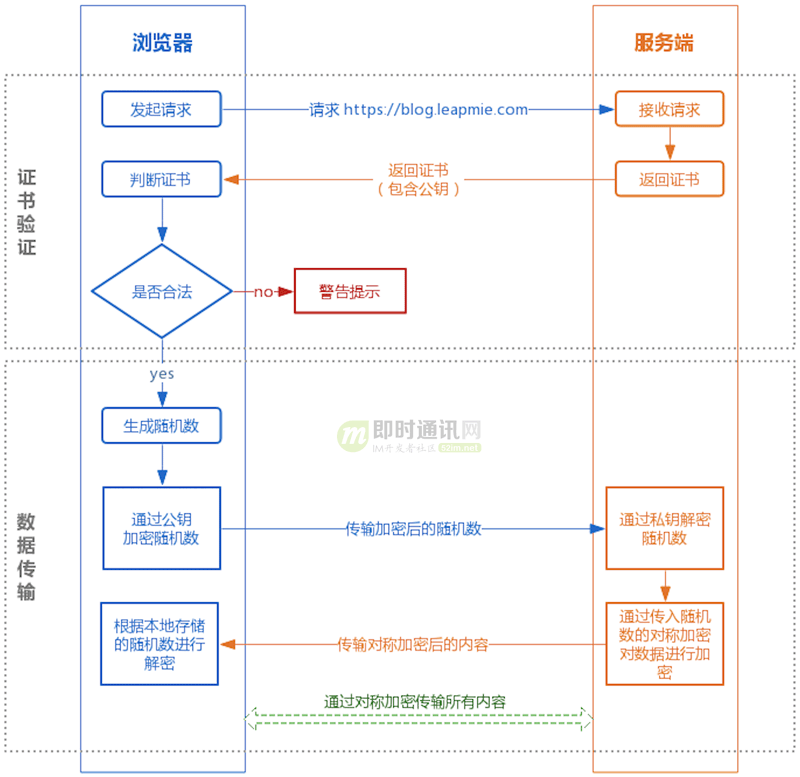
签名产生算法：使用散列函数（哈希）计算公开的明文信息的摘要，并采用CA的私钥对信息摘要进行加密，密文即签名；

2 客户端使用非对称加密的方式与服务器进行通信，生成随机数，协商对称加密传输数据使用的密钥。

客户端使用服务端的公钥加密 对称密钥 的协商数据，服务端用私钥解密；服务端使用解密后随机数生成对称加密算法，对响应内容对称加密，传输给客户端，

1. 双方使用对称加密传输所有内容。

数据传输阶段使用对称加密而不使用非对称加密的原因是，非对称加密的加解密效率非常低，http 的应用场景中通常端与端之间存在大量的交互，非对称加密的效率是无法接受的。另外：在 HTTPS 的场景中只有服务端保存了私钥，一对公私钥只能实现单向的加解密，所以 HTTPS 中内容传输加密采取的是对称加密，而不是非对称加密。



Linux启动过程

Nginx负载均衡策略

从upstream模块定义的后端服务器列表中选取一台服务器接收用户的请求。#动态服务器组

upstream dynamic\_zuoyu {

server localhost:8080; #tomcat 7.0

server localhost:8081; #tomcat 8.0

server localhost:8082; #tomcat 8.5

server localhost:8083; #tomcat 9.0

}

有6中方式的分配策略：

1. 轮询，将请求均匀分配给不同的服务器
2. 权重，按服务器性能不同赋予不同的权重，权重越高分配到处理的请求越多
3. Ip\_hash，根据请求ip即客户端IP的Hash值分配处理的服务器，即相同的客户端的请求会被发送到同一个服务器处理
4. Least\_conn，最少连接方式，即处理请求最少的服务器
5. url\_hash，请求的url的Hash值分配处理服务器
6. fair， 响应时间方式

心跳机制

**UCP与TCP区别**

1.基于连接与无连接；  
2.对系统资源的要求（TCP较多，UDP少）；  
3.UDP程序结构较简单；  
4.流模式与数据报模式 ；

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

**TCP 拥塞控制**

如何辨别网络出现拥塞：超时或出现3个冗余ACK

拥塞控制算法：加性增乘性减、慢启动

慢启动：TCP连接建立初始阶段，拥塞窗口congWin大小为1，TCP发送方在初始阶段不是线性增加发送速率的，而是以指数的速度增加，没经过一个RTT将congWin的值翻倍，直到发生一个丢包事件为止，此时congWin将被降为一半，然后开始线性增长。

乘性减：每发生一次丢包事件将当前CongWin值减半

加性增：每收到一个ACK确认包后将CongWin增大一点

对因超时和收到3个冗余ACK而检测到的丢包事件作出的反应是不同的，

收到3个冗余ACK后，拥塞窗口减半，开始线性增长

超时事件发生，TCP发送方进入慢启动阶段，congWin被设为1，然后窗口指数增长，直到congWin达到超时事件前窗口值得一半为止，

阈值：每当发生一个丢包事件，Threshold值会被设置为当前CongWin的一半。

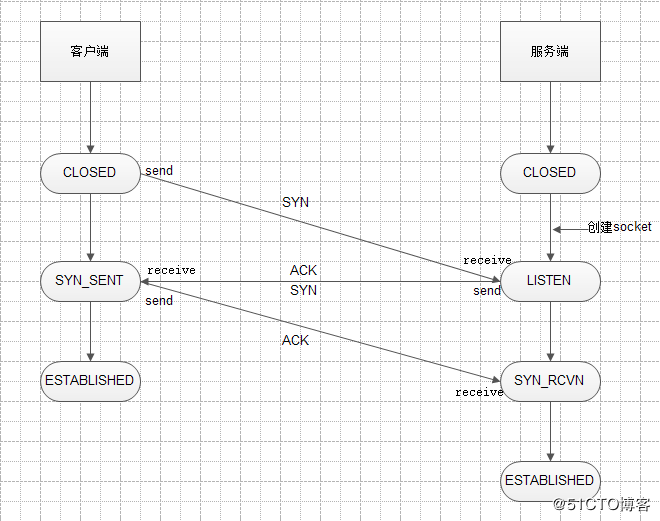
因此，TCP发送方在发生超时事件后进入慢启动阶段，慢启动阶段congWin以指数快速增长，直至congWin达到阈值Threshold为止，当congWin达到阈值时，TCP就进入拥塞避免阶段，在该阶段中，congWin线性增长。

**SYN泛洪攻击**

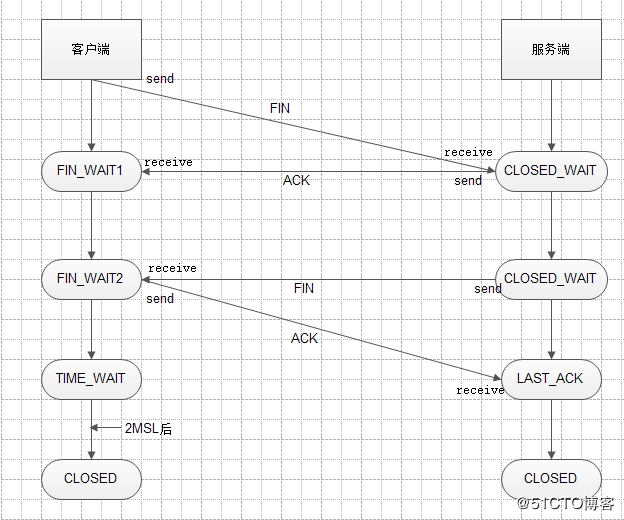
攻击者发送大量的TCP SYN报文段，而不完成三次握手的第三步，通过从多个源发送SYN能够加大供给力度，从而形成DDos分布式拒绝服务SYN泛洪攻击。随着这种SYN报文的到来，服务器要不断为这些半开连接分配资源，结果导致服务器连接资源迅速消耗殆尽。

**三次握手和四次挥手状态**

1、一开始，建立连接之前服务器和客户端的状态都为CLOSED；  
2、服务器创建socket后开始监听，变为LISTEN状态；  
3、客户端请求建立连接，向服务器发送SYN报文，客户端的状态变味SYN\_SENT；  
4、服务器收到客户端的报文后向客户端发送ACK和SYN报文，此时服务器的状态变为SYN\_RCVD；  
5、然后，客户端收到ACK、SYN，就向服务器发送ACK，客户端状态变为ESTABLISHED；  
6、服务器端收到客户端的ACK后变为ESTABLISHED。此时3次握手完成，连接建立！



由于TCP连接是全双工的，断开连接会比建立连接麻烦一点点。  
1、客户端先向服务器发送FIN报文，请求断开连接，其状态变为FIN\_WAIT1；  
2、服务器收到FIN后向客户端发送ACK，服务器的状态围边CLOSE\_WAIT；  
3、客户端收到ACK后就进入FIN\_WAIT2状态，此时连接已经断开了一半了。如果服务器还有数据要发送给客户端，就会继续发送；  
4、直到发完数据，就会发送FIN报文，此时服务器进入LAST\_ACK状态；  
5、客户端收到服务器的FIN后，马上发送ACK给服务器，此时客户端进入TIME\_WAIT状态；  
6、再过了2MSL长的时间后进入CLOSED状态。服务器收到客户端的ACK就进入CLOSED状态。  
至此，还有一个状态没有出来：CLOSING状态。  
CLOSING状态表示：  
客户端发送了FIN，但是没有收到服务器的ACK，却收到了服务器的FIN，这种情况发生在服务器发送的ACK丢包的时候，因为网络传输有时会有意外。



LISTEN：等待从任何远端TCP 和端口的连接请求。

SYN\_SENT：发送完一个连接请求后等待一个匹配的连接请求。

SYN\_RECEIVED：发送连接请求并且接收到匹配的连接请求以后等待连接请求确认。

ESTABLISHED：表示一个打开的连接，接收到的数据可以被投递给用户。连接的数据传输阶段的正常状态。

FIN\_WAIT\_1：等待远端TCP的连接终止请求，或者等待之前发送的连接终止请求的确认。

FIN\_WAIT\_2：等待远端TCP的连接终止请求。

CLOSE\_WAIT：等待本地用户的连接终止请求。

CLOSING：等待远端TCP 的连接终止请求确认。

LAST\_ACK：等待先前发送给远端TCP 的连接终止请求的确认（包括它字节的连接终止请求的确认）

TIME\_WAIT：等待足够的时间过去以确保远端TCP 接收到它的连接终止请求的确认。

TIME\_WAIT 两个存在的理由：

1.可靠的实现tcp全双工连接的终止；

2.允许老的重复分节在网络中消逝。

CLOSED：不在连接状态（这是为方便描述假想的状态，实际不存在）

**JVM内存区域划分：**

线程共享：方法区、堆

方法区：存储类的信息、常量池、类方法、类变量等

JDK1.6、1.7时方法区就是永久代，JDK1.8方法区升级成为元空间

元空间和永久代最大的区别在于：元空间并不再虚拟机中，而是使用本地内存。

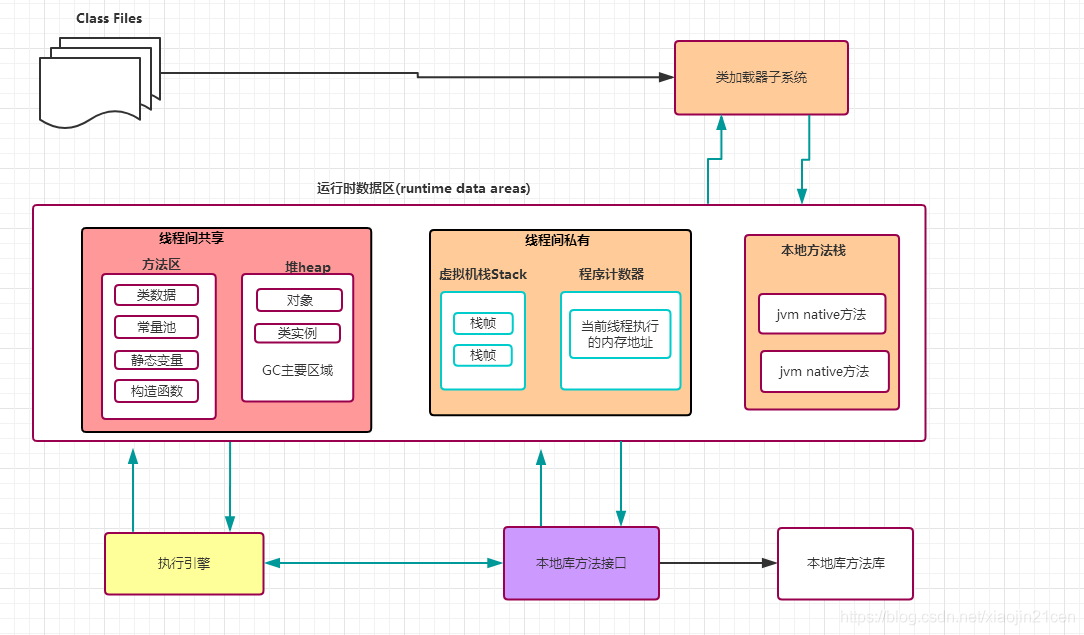
堆：存储对象实例信息、GC回收、OOM溢出

线程私有：程序计数器、虚拟机栈、本地方法栈

程序计数器：记录当前线程执行的字节码的行号指示器。

虚拟机栈：线程执行函数的栈帧，记录了局部变量表、操作数、方法动态链接及方法返回等信息

本地方法栈：线程执行native本地方法时的栈帧



项目：

1. 秒杀

描述：千万级用户秒杀商品，奖参与秒杀的商品提前放入redis队列中，秒杀的时候每有一个用户发起渺少请求，将对应商品的list左pop一个元素，pop成功则表示抢到，并将此用户id存入set集合中，防止超买，否则pop出为空表示没有秒杀到，秒杀失败了。

商品减库存

**静态数据优化**对于稳定数据，在一段事件内保持稳定不变的数据，通常使用两种方式处理：利用缓存或者静态化技术。如商品信息等，可使用Hash数据类型存储

**RabbitMQ异步消息队列流量削峰**

生成订单，使用了rabbitMQ异步消息队列，用户秒杀成功以后使用UUID生成一个订单号并返回，将用户订单信息发送到消息队列中，Order消费者即订单操作程序会将订单信息异步的持久化到数据库中，这里可以对消费者一次性处理的请求进行限流，即设置prefetch可以限制。

秒杀页面要定时（3秒）checkoutOrder是否处理好，若没有生成Order则依然在wait…页面，若Order处理成功则进入订单详情页。

订单若在一定时间内没有被支付，那该订单关闭成为历史订单。

主动说 ，redis缓存穿透、缓存击穿、雪崩及热键

**缓存穿透：**查询一条在redis中不存在且在后端数据库中也不存在的数据，高并发下可使用布隆过滤器，判断，系统启动时将后端数据库中要被访问的信息ID根据多个HASH函数映射到布隆过滤器的相应位bitmap，当有请求过来时先经过布隆过滤器的hash判断，若所有hash位均位1，请求往后传，否则返回，说明数据库中不存在请求的记录。

**缓存击穿：**查询一条在redis中不存在但在后端数据库中存在的数据，这样使用redis互斥锁控制只能有一个线程处理段程序，若如果不再redis里边，上锁访问后端数据库，访问到返回并存储到redis中，这样下一个请求就无需再去访问数据库，没有获得锁就执行从redis中直接取数据的操作即可。

**缓存雪崩：**设置缓存时采用了相同的过期时间，导致缓存在某一时刻同时失效，请求全部转发到DB，DB瞬时压力过重雪崩。缓存失效时间分散开，比如我们可以在原有的失效时间基础上增加一个随机值，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

**Nginx环境下存在session不同步的问题**

每个session默认存放在不同的后端服务器上，Nginx将同一个客户端的请求负载均衡分发到不同的后端服务器，导致session不同步，为了session信息共享，将session存放在一个独立的redis服务器中，将session转存到redis中，即全局只存储一份，所有后端服务器共享这一份，实现session的共享。直接使用的spring-session，导入依赖包，并在程序入口处注释@EnableRedisHttpSession，这样会框架会自动监听将session存储到redis中，这样不同后端服务器共享了同一个session，实现了session同步问题。

**防重和幂等性：**

幂等性：**用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。**

**如何保证幂等性：**

**insert前先select （不适用于并发场景）、状态机，**

**更新时的where条件中包含了该条记录的状态码**

1. **生成订单时的幂等性，根据用户ID和商品ID查看是否有这条记录，有则不操作，没有生成新订单。**

**客户端可使用一个全局唯一的流水号ID，用来表示是不是同一个请求或交易，让客户端生成这个ID，每个抢购或生成订单的请求生成一个唯一ID,并将ID保存到一个流水表中，设为唯一约束UNIQUE KEY，若插入出现冲突，说明这个创建订单的请求已经被处理过了，直接返回之前的操作结果即可。**

**当调用方携带流水号ID调用创建订单的接口，如果出现超时了，调用方不知道订单到底创建成功还是失败，这个时候，用同一个流水号进行重试，订单系统虽然收到了两个请求，但是由于流水号ID是同一个，可以根据流水表来做幂等操作。并告知对方订单创建成功与否。**

1. **支付订单时的幂等性，支付订单时先查询当前订单的支付状态，若状态为待支付那直接支付，若已支付那就不要再支付了，这样保证幂等性，不会多支付。**

**支付的时候产生一个唯一的支付订单号，支付接口保证幂等性，**

电商

1. 动态页面静态化

（1）影片信息是存储在数据库中的，准备首页和详情页模板，提前读取模板和数据库影片信息，使用io流生成html静态文件存储，用户访问的时候直接返回html页面，而无需动态加载，

（2）当然，这里的影片信息也可以在系统启动时存放在redis中，请求来时直接从redis中取渲染页面，这样也可，无需到后端数据库读取

图片存储在一个独立的虚拟图片服务器上

（3）这些静态资源也可以选择在CDN内容分发，CDN会根据用户线路及位置，为用户选择靠近的位置和相同的运营商，提升用户体验。即将用户的请求重新导向到离用户最近的服务结点上，使用户就近取得所访问的内容，提高用户访问网站的响应速度。

1. 多线程抢购同一张票或秒杀同一件商品

使用事务，select ticket … where state = .. for update，这会在数据库中对这行数据加上行锁，此时只能有一条线程访问，-付款-口库存-生成订单使用事务控制。

这里可以根据场次售票时间提前将该场次对应的票加入redis缓存，这样用户买票的时候直接从redis中取，由于每张票只有一张，我觉得可以将场次的票放入一个set中，场次ID为key，ticket的id为set中value，当用户购票请求发来时，会携带场次和ticket的id信息，查询对应的场次set中是否存在该id，存在则秒啥成功，从set中移除，不存在则表示该票已经售卖，这个售卖也要减库存-扣款等，

1. 跨域问题，安全

由于浏览器的安全限制，JS只能访问与所在页面同一域（相同协议、域名、端口）的内容，通过ajax请求后端接口并返回数据，有时会受到浏览器的安全限制，产生跨域问题

JSONP

服务端设置add-allow-origin- 为\*号

1. 自动化测试平台

任务