**dandyhuang**

# 前端

前端怎么入门https://www.zhihu.com/question/32314049/answer/713711753

## HTML

### vscode先安装插件吧

### w3school撸点基础知识

* 基础

• 标题

• <h1>大标题单词head<h1>

• 段落

• <p> paragraph 段落</p>

• 链接

• <a href="http://www.baidu.com">dandyhuang link</a>

• 图像

• <img src="图片路径" width="304">

• 关闭标签

• <br />

* 元素

• HTML 元素以开始标签起始  
HTML 元素以结束标签终止  
元素的内容是开始标签与结束标签之间的内容  
某些 HTML 元素具有空内容（empty content）  
空元素在开始标签中进行关闭（以开始标签的结束而结束）  
大多数 HTML 元素可拥有属性

* 属性

• 1属性总是在 HTML 元素的开始标签中规定  
2属性总是以名称/值对的形式出现，比如：name="value"

• 居中：<h1 align="center">

• 背景颜色:<body bgcolor="yellow">

# 后端

## 设计模式

## 基础语言go、c++

* + 1. 字节对齐的作用

不仅是便于cpu快速访问，同时合理的利用字节对齐可以有效地节省存储空间。

为了使CPU能够对变量进行快速的访问,变量的起始地址应该具有某些特性,即所谓的”对齐”. 比如4字节的int型,其起始地址应该位于4字节的边界上,即起始地址能够被4整除.

* + 1. C和c++的区别

设计思想上：C++是面向对象的语言，而C是面向过程的结构化编程语言语法上：  
C++具有封装、继承和多态三种特性C++相比C，增加多许多类型安全的功能，比如强制类型转换、C++支持范式编程，比如模板类、函数模板等

* + 1. 基础

int a[5]={1,2,3,4,5};    int \*p=(int \*)(&a+1);//a本来就是指针的，指针在取地址  
    cout<<\*(p-1)<<endl; //输出为5 int a[5]={1,2,3,4,5};    int \*p=(int \*)&(a+1);    cout<<\*(p-1)<<endl;//输出为1  
那么对于(a[row]) [col]，a代表什么呢？让我们变换一下：(a[row]) [col] = (\*(a+row) )[col] =\*(\*(a+row) + col) 

2.1extern的作用  
1 cpp里头，全局变量、函数等，其他文件需要使用的时候，需要加上extern，静态变量不能extern  
2 当需要使用到c文件里头定位的变量函数时候，需要使用extern C来处理，  
  
2.2 请你回答一下C++中拷贝赋值函数的形参能否进行值传递？不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环，无法完成拷贝，栈也会满。

2.3 请你来介绍一下STL的allocaotrSTL的分配器用于封装STL容器在内存管理上的底层细节。在C++中，其内存配置和释放如下： new运算分两个阶段：(1)调用::operator new配置内存;(2)调用对象构造函数构造对象内容delete运算分两个阶段：(1)调用对象希构函数；(2)掉员工::operator delete释放内存 为了精密分工，STL allocator将两个阶段操作区分开来：内存配置有alloc::allocate()负责，内存释放由alloc::deallocate()负责；对象构造由::construct()负责，对象析构由::destroy()负责。 同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL采用了两级配置器，当分配的空间大小超过128B时，会使用第一级空间配置器；当分配的空间大小小于128B时，将使用第二级空间配置器。第一级空间配置器直接使用malloc()、realloc()、free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。  
2.4  请你来说一下STL中迭代器的作用，有指针为何还要迭代器 1、迭代器Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。由于Iterator模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及ostream\_iterator等扩展iterator。 2、迭代器和指针的区别迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、\*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身。 3、迭代器产生原因Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

* + 1. 迭代器失效  
       数组型数据结构(deque、vector): 因为连续，所以删除后，当前迭代器都失效，所以只能用erase返回值是下一个有效迭代器的值链表型数据结构(list): 因为不连续，删除后，不会失效其他迭代器，所以可以erase返回值或者erase(iter++)处理  
       解析：dataMap.erase(iter++);这句话分三步走，先把iter传值到erase里面，然后iter自增，然后执行erase,所以iter在失效前已经自增了。  
       树形结构(map set)： 只能用erase(iter++), 因为erase返回值为void
    2. C++内存模型  
       在C++中，虚拟内存分为代码段、数据段、BSS段、堆区、文件映射区以及栈区六部分。代码段:包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。数据段：存储程序中已初始化的全局变量和静态变量bss 段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量。堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存。映射区:存储动态链接库以及调用mmap函数进行的文件映射栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值  
       https://blog.csdn.net/jirryzhang/article/details/79518408
    3. 内存泄漏处理  
       使用valgrind、mtrace  
       常发生在: 内存分配、句柄泄漏、虚函数析构函数等 。
    4. 虚函数的作用和实现原理，什么是虚函数,有什么作用?  
        这种数组成为虚函数表（virtual function table, vtbl），即，每个类使用一个虚函数表，每个类对象用一个虚表指针  
        虚函数表里只能存放虚函数，不能存放普通函数。如果一个函数不是虚函数，那么对它的调用（即该函数的地址）在编译阶段就会确定。调用虚函数的话（它的地址）要运行时才能确定。虚函数的函数入口是动态绑定的。在运行时，程序根据基类指针指向的实际对象，来调用该对象对应版本的函数。（用该对象的虚函数表指针找到其虚函数表，进而调用不同的函数。）（只有是虚函数的情况下才会这么做（用虚函数表指针去查虚函数表）。非虚函数直接就调用自己的。）
* 纯虚函数，为什么需要纯虚函数？
* 为什么需要虚析构函数,什么时候不需要?父类的析构函数为什么要定义为虚函数?  
  以上代码会产生内存泄露，因为new出来的是Derived类资源，采用一个基类的指针来接收，析构的时候，编译器因为只是知道这个指针是基类的，所以只将基类部分的内存析构了，而不会析构子类的，就造成了内存泄露，如果将基类的析构函数改成虚函数，就可以避免这种情况，因为虚函数是后绑定，其实就是在虚函数列表中，析构函数将基类的析构函数用实际对象的一组析构函数替换掉了，也就是先执行子类的虚函数再执行父类的虚函数，这样子类的内存析构了，父类的内存也释放了，就不会产生内存泄露。  
  注：  
  1.析构函数其实是一个函数，不论子类还是父类，虽然可能看起来名字不一样。而且析构函数执行过程都是执行子类再到父类。  
  2.多态的时候一定要将析构函数写成虚函数，防止内存泄露，各个子类维护自己内部数据释放。
* 内联函数、构造函数、静态成员函数可以是虚函数吗?  
  都不可以。

1.内联函数需要在编译阶段展开，而虚函数是运行时动态绑定的，编译时无法展开；

2.构造函数在进行调用时还不存在父类和子类的概念，父类只会调用父类的构造函数，子类调用子类 的，因此不存在动态绑定的概念；

3.静态成员函数是以类为单位的函数，与具体对象无关，虚函数是 与对象动态绑定的，因此是两个不冲突的概念；

* 构造函数中可以调用虚函数吗?
* 为什么需要虚继承?虚继承实现原理解析，
* c++编译链接的过程是怎么样的  
  编译就是把文本形式源代码翻译为机器语言形式的目标文件的过程。链接是把目标文件、操作系统的启动代码和用到的库文件进行组织，形成最终生成可执行代码的过程  
  gcc编译过程  
  ：预编译处理(.c) －－> 编译、优化程序（.s、.asm）－－> 汇编程序(.obj、.o、.a、.ko) －－> 链接程序（.exe、.elf、.axf等）  
  
* 模板有哪几种类型

类模板，模板函数、类模板特化  
  
c++动态库和静态库

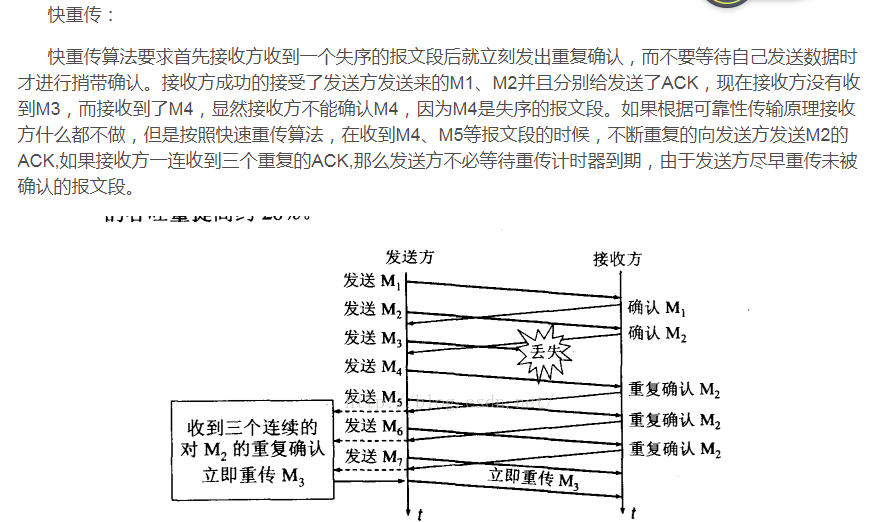
## 网络

### 为什么是三次握手和四次挥手 因为如果是两次握手，a发送给服务器b，这时候因为连接请求，报文丢失而未收到确认，a又重试发起建立连接。这样a发起了两个请求，其中一个正常处理，另外一个因为网络原因，延迟到连接释放以后的某个时间点才到达b，b误以为a又发起新的连接请求。只要b确认就发起建连，此时b一直等待a发送数据，造成资源浪费。 sync攻击就是：client不存在的连接请求。导致服务一直等待client发起确认建连接，并且不断的重试的半连接状态。 尽快释放半连接的占用。 2mL原因： 1 可靠的关闭TCP连接 确保最后一次挥手，a发出的最后一个的ack报文能到达b， 否则B无法进入正常的closed状态。 2 防止 已失效的连接请求报文出现在本连接中 close\_wait越多，连接未释放，一旦达到句柄数上限，新的请求就无法被处理了，接着就是大量Too Many Open Files异常

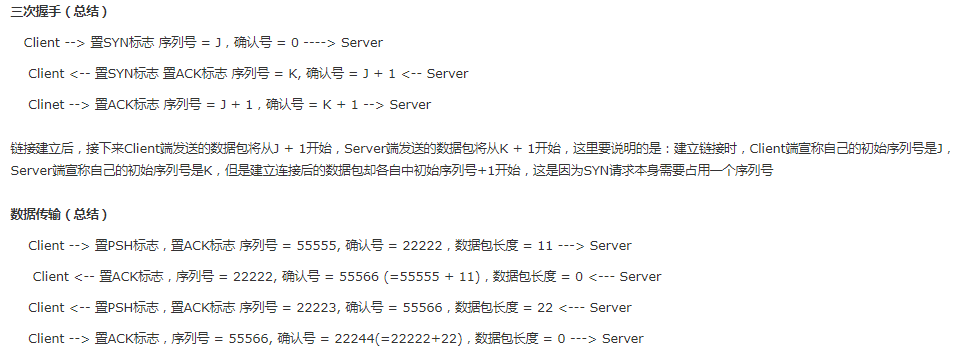
### TCP/IP



* 流量控制:滑动窗口

流量控制，tcp是全双工的，所以任意一方都维护这发送窗口和接收窗口，发送窗口只会收到对端接收的ack确认，才会移动发送窗左边界；接收窗口只有在前面所有的段都确认只会，才会移动左边界。并且每次回复剩余的win窗口大小。 举例：如一个数据被分为pkg1、2、3、4，接收端收到1、2、4，因为1、2是连续的，所以可以直接确认，4不能确认，放到缓存中（为了保证数据顺序、等3到了，才会顺序写入）  
拥塞控制：慢启动、拥塞避免、快速重传和快速恢复  
先探测网络的拥塞程度，由小到大，呈指数增长，若出现丢包，则将拥塞窗口减半，进入拥塞避免阶段。进入拥塞避免阶段，窗口每轮呈线性增长；当收到对一个保温的三个重复ack，认为这个报文的下一个报文丢失，进入快重传阶段，理解下滑动窗口。 收到的是连续对失序的报文发出重复确认。  


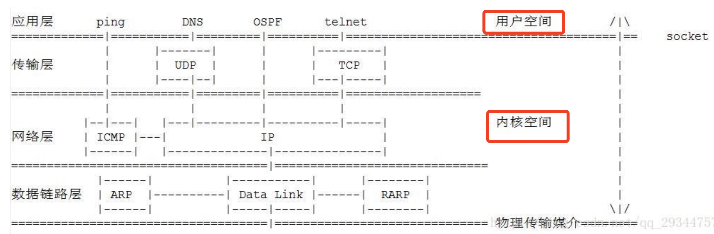
* Tpc首部长度有哪些字段
* tpc如何保证可靠性  
  校验和、序列号、确认应答、超时重传、连接管理、流量控制、拥塞控制  
  (1）序列号、确认应答、超时重传数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明了它下一次需要接收的数据序列号。如果发送发迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是2\*RTT(报文段往返时间）+一个偏差值。（2）窗口控制与高速重发控制/快速重传（重复确认应答）TCP会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发。使用窗口控制，如果数据段1001-2000丢失，后面数据每次传输，确认应答都会不停地发送序号为1001的应答，表示我要接收1001开始的数据，发送端如果收到3次相同应答，就会立刻进行重发；但还有种情况有可能是数据都收到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过它的，会疯狂向它提醒......（3）拥塞控制  
  1、慢开始最开始发送方的拥塞窗口为1，由小到大逐渐增大发送窗口和拥塞窗口。每经过一个传输轮次，拥塞窗口cwnd加倍。当cwnd超过慢开始门限，则使用拥塞避免算法，避免cwnd增长过大。2、拥塞避免每经过一个往返时间RTT，cwnd就增长1。在慢开始和拥塞避免的过程中，一旦发现网络拥塞，就把慢开始门限设为当前值的一半，并且重新设置cwnd为1，重新慢启动。（乘法减小，加法增大）3、快速重传接收方每次收到一个失序的报文段后就立即发出重复确认，发送方只要连续收到三个重复确认就立即重传（尽早重传未被确认的报文段）。4、快速恢复当发送方连续收到了三个重复确认，就乘法减半（慢开始门限减半），将当前的cwnd设置为慢开始门限，并且采用拥塞避免算法（连续收到了三个重复请求，说明当前网络可能没有拥塞）。  
  如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵（大家都在用网，你在这狂发，吞吐量就那么大，当然会堵），甚至造成网络的瘫痪。所以TCP在为了防止这种情况而进行了拥塞控制。慢启动：定义拥塞窗口，一开始将该窗口大小设为1，之后每次收到确认应答（经过一个rtt），将拥塞窗口大小\*2。拥塞避免：设置慢启动阈值，一般开始都设为65536。拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法增加（每次确认应答/每个rtt，拥塞窗口大小+1），以此来避免拥塞。将报文段的超时重传看做拥塞，则一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一半，并且将窗口大小设为初值1，然后重新进入慢启动过程。快速重传：在遇到3次重复确认应答（高速重发控制）时，代表收到了3个报文段，但是这之前的1个段丢失了，便对它进行立即重传。然后，先将阈值设为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3的大小。这样可以达到：在TCP通信时，网络吞吐量呈现逐渐的上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪  
  （<https://blog.csdn.net/wodewutai17quiet/article/details/82252454>）

序列号：  
初始序列号（ISN）是客户端随机产生的一个值2712239078， 确认号是0  
序列号是服务器随机产生的一个值1288781508  ，回确认号都是发送方值+1  


* tcp的序列号为什么不能从0开始

1 防止外部tcp序列 预测攻击  
2 cs断开后,会有可能某些数据仍然滞留在网络中.当cs再次连接,这些延迟数据在建立新连接之后才到达Server,如果都是从0开始会使得新连接与旧连接数据包重叠几率大大增加,造成数据混淆

https://blog.csdn.net/qq\_29344757/article/details/78377500

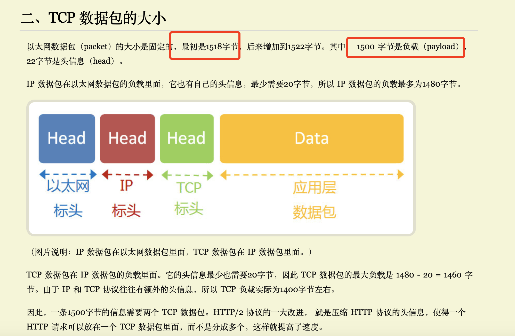


• 请看链接tcp、udp区别等

* 应用层(dns,http,ftp)-传输层(udp,tcp)-网络层-数据链路层  
  1、网络包的组成：  
   报头/起始帧分界符——MAC头部——IP头部——TCP头部——数据——FCS（帧校验序列）
* TCP

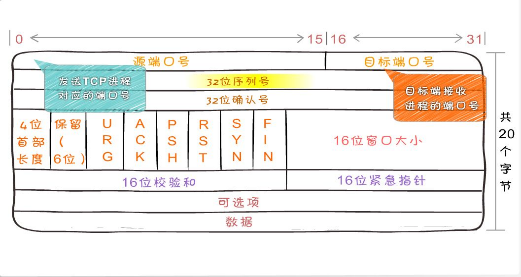
https://zh.wikipedia.org/wiki/%E4%BC%A0%E8%BE%93%E6%8E%A7%E5%88%B6%E5%8D%8F%E8%AE%AE

•



•

https://zhuanlan.zhihu.com/p/33889997



• TCP首部最小为20字节，这20字节分为5行，每行4个字节也就是32个位。

• CWR：用于 IP 首部的 ECN 字段。ECE 为 1 时，则通知对方已将拥塞窗口缩小。  
ECE：在收到数据包的 IP 首部中 ECN 为 1 时将 TCP 首部中的 ECE 设置为 1，表示从对方到这边的网络有拥塞。  
URG：紧急模式  
ACK：确认  
PSH：推送，接收方应尽快给应用程序传送这个数据。没用到  
RST：该位为 1 表示 TCP 连接中出现异常必须强制断开连接。  
SYN：初始化一个连接的同步序列号  
FIN：该位为 1 表示今后不会有数据发送，希望断开连接。

发送方端口号                接收方端口号                序号(发送数据的顺序编号)：发送方告诉接收方该网络包发送的数据相当于所有发送数据的第几个字节。                ACK号(接收数据的顺序编号)                数据偏移量                保留                控制位：                     URG：表示紧急指针字段有效                     ACK：表示接收数据序号字段有效                     PSH：表示通过flush操作发送的数据                     RST：该位为 1 表示 TCP 连接中出现异常必须强制断开连接                     SYN：发送方和接受方相互确认序号，表示连接操作                     FIN：该位为 1 表示今后不会有数据发送，希望断开连接                窗口：接收方告知发送方窗口的大小(即无需等待确认认可一起发送的数据量)                校验和                紧急指针                可选字段

• 可选项：一般为tlv格式，SACK 选项也在里头

• 重传机制

• 超时重传

• RTT包往返时间

• RTO包重传时间

• 快速重传

• 重传之前没有收到的数据

• SACK(Selective  
Acknowledgment)

• 在快速重传，不清楚要重传哪一段的前提下，TCP 头部带上SACK

• D-SACK(  
Duplicate)

• 1 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了;  
  
2 可以知道是不是「发送方」的数据包被网络延迟了;  
  
3 可以知道网络中是不是把「发送方」的数据包给复制了;

• 一般RTO会大于RTT时间

• 每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送

• 滑动窗口

• 发送窗口

• tcp包头有win字段，就是告诉对方缓冲区大小

• 接收窗口

• 流量控制

• 操作系统缓存区与滑动窗口的关系

• 窗口关闭

• TCP 连接一方收到对方的零窗口通知，就启动持续计时器，如果没有这个，将会照成死锁，如果 3 次过后接收窗口还是 0 的话，有的 TCP 实现就会发 RST 报文来中断连接

• 糊涂窗口综合症  
当发送端应用进程产生数据很慢、或接收端应用进程处理接收缓冲区数据很慢, 就会使应用进程间传送的报文段很小，特别是有效载荷很小。极端情况下，有效载荷可能只有1个字节；而传输开销有40字节(20字节的IP头+20字节的TCP头) 这种现象就叫糊涂窗口综合症

• TCP\_NODELAY

• 对于一些需要小数据包交互的场景的程序，比如，telnet 或 ssh 这样的交互性比较强的程序，则需要关闭 Nagle 算法

• 拥塞控制

• 慢启动

• 按照指数形式回可用窗口1、2、4、8

• 拥塞避免

• 拥塞窗口 cwnd 「超过」慢启动门限 ssthresh 就会进入拥塞避免算法，将原先慢启动指数增长变成了线性增长

• 一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。  
当触发了重传机制，也就进入了「拥塞发生算法」

• 拥塞发生

• 超时重传

• 直接又进入满启动模式

• 快速重传

• 快速恢复

• tcp和udp是否可以共用同一个端口

• 三次握手，四次挥手

• 为什么客户端需要等待2MSL

为了这种情况： B向A发送 FIN = 1 的释放连接请求，但这个报文丢失了， A没有接到不会发送确认信息， B 超时会重传，这时A在 WAIT\_TIME 还能够接收到这个请求，这时再回复一个确认就行了。（A收到 FIN = 1 的请求后 WAIT\_TIME会重新记时） 另外服务器B存在一个保活状态，即如果A突然故障死机了，那B那边的连接资源什么时候能释放呢？ 就是保活时间到了后，B会发送探测信息， 以决定是否释放连接

• 为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当Server端收到Client端的SYN连接请求报文后，可以直接发送SYN+ACK报文。其中ACK报文是用来应答的，SYN报文是用来同步的。 但是关闭连接时，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，"你发的FIN报文我收到了"。只有等到我Server端所有的报文都发送完了，我才能发送FIN报文，因此不能一起发送。故需要四步握手。

• Server端易受到SYN攻击？

服务器端的资源分配是在二次握手时分配的，而客户端的资源是在完成三次握手时分配的，所以服务器容易受到SYN洪泛攻击，SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server则回复确认包，并等待Client确认，由于源地址不存在，因此Server需要不断重发直至超时，这些伪造的SYN包将长时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络拥塞甚至系统瘫痪。 防范SYN攻击措施：降低主机的等待时间使主机尽快的释放半连接的占用，短时间受到某IP的重复SYN则丢弃后续请求。

• 四次挥手过程

起初A和B处于ESTABLISHED状态——A发出连接释放报文段并处于FIN-WAIT-1状态——B发出确认报文段且进入CLOSE-WAIT状态——A收到确认后，进入FIN-WAIT-2状态，等待B的连接释放报文段——B没有要向A发出的数据，B发出连接释放报文段且进入LAST-ACK状态——A发出确认报文段且进入TIME-WAIT状态——B收到确认报文段后进入CLOSED状态——A经过等待计时器时间2MSL后，进入CLOSED状态。

• tcp上述应用解析

https://blog.csdn.net/dog250/article/details/6612496

### Linux 五种IO模型

* 操作系统

• 用户空间和内核空间

• 现在操作系统都是采用虚拟存储器，对32位操作系统而言，它的寻址空间（虚拟存储空间）为4G（2的32次方）。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核（kernel），保证内核的安全，操作系统将虚拟空间划分为两部分，一部分为内核空间，一部分为用户空间。针对linux操作系统而言，将最高的1G字节（从虚拟地址0xC0000000到0xFFFFFFFF），供内核使用，称为内核空间，而将较低的3G字节（从虚拟地址0x00000000到0xBFFFFFFF），供各个进程使用，称为用户空间

• 进程切换

• 内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为被称为进程切换

• 1 保存处理机上下文，包括程序计数器和其他寄存器。

• 2 更新PCB信息。

• 计算机进程（PCB）有四个去向：  
1）留在就绪队列  
2）进入CPU  
3）等待队列  
4）I/O操作

• 3 把进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列

• 4 选择另一个进程执行，并更新其PCB

• 5 更新内存管理的数据结构。

• 6 恢复处理机上下文。

• 进程的阻塞

• 正在执行的进程，由于期待的某些事件未发生，如请求系统资源失败、等待某种操作的完成、新数据尚未到达或无新工作做等，则由系统自动执行阻塞原语(Block)，使自己由运行状态变为阻塞状态。可见，进程的阻塞是进程自身的一种主动行为，也因此只有处于运行态的进程（获得CPU），才可能将其转为阻塞状态。当进程进入阻塞状态，是不占用CPU资源的

• 文件描述符fd

• 缓存io

• 缓存 IO 又被称作标准 IO，大多数文件系统的默认 IO 操作都是缓存 IO。在 Linux 的缓存 IO 机制中，操作系统会将 IO 的数据缓存在文件系统的页缓存（ page cache ）中，也就是说，数据会先被拷贝到操作系统内核的缓冲区中，然后才会从操作系统内核的缓冲区拷贝到应用程序的地址空间。  
  
缓存 IO 的缺点：  
  
数据在传输过程中需要在应用程序地址空间和内核进行多次数据拷贝操作，这些数据拷贝操作所带来的 CPU 以及内存开销是非常大的

* 同步模型  
  阻塞io  
  非阻塞IO  
  多路复用IO  
  信号驱动式IO  
  异步IO

• IO多路复用之select、poll、epoll

https://www.jianshu.com/p/dfd940e7fca2

### 常见问题

* • TCP/IP协议栈层次与三次握手、四次挥手需要知道的细节  
  • TCP与UDP适用场景  
  • linux网络模型  
  • epoll\_event结构中epoll\_data\_t的fd与ptr使用场景  
  •Windows网络模型  
  •异步connect  
  •select可以检测网络异常吗  
  •epoll的水平模式和边缘模式  
  •阻塞与非阻塞socket的设置与区别  
  •send/recv返回值问题  
  •如何编写正确的收与发数据代码  
  •收发缓冲区如何设计  
  •SO\_SNDTIMEO、SO\_RCVTIMEO、TCP\_NODELAY、SO\_REUSEADDR和SO\_REUSEPORT、SO\_LINGER选项  
  •shutdown与优雅关闭  
  •错误码EINTR  
  •tcp粘包问题  
  •信号SIGPIPE与EPIPE错误码  
  •gethostbyname阻塞与错误码  
  •SO\_KEEPALIVE选项与心跳包设计技巧  
  •如何设计断线重连机制  
  •如何清除无效的死链  
  •网络框架中定时器不同实现  
  •http协议格式、head、get与post方法细节  
  •http、socks4与socks5代理编码实现

## Redis

https://www.cnblogs.com/kismetv/p/9236731.html

### Redis常问问题

* + - 1. 怎么保证扩缩容
      2. 跳表实现原理
      3. **skiplist与平衡树、哈希表的比较**

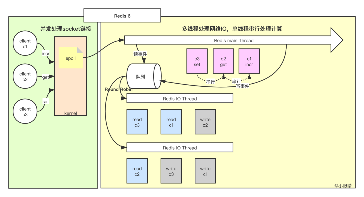
## 1 平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。

2 哈希不支持范围查找

* + - 1. **redis主从同步**

·1发送 SLAVEOF 命令可以进行主从同步，比如：SLAVEOF 127.0.0.6379·2主从同步有同步和命令传播 2 个步骤。 同步：将从服务器的数据库状态更新成主服务器当前的数据库状态（一个消耗资源的操作） 命令传播：当主服务器数据库状态被修改后，导致主从服务器数据库状态不一致，此时需要让主从数据同步到一致的过程·3 主从同步分初次复制和断线后重复制两种情况 从 2.8 版本开始，在出现断线后重复制情况时，主服务器会根据复制偏移量、复制积压缓冲区和 run id，来确定执行完整重同步还是部分重同步4 2.8 版本使用 psync 命令来代替 sync 命令去执行同步操作。目的是为了解决同步（sync 命令）的低效操作。

### redis非关系型数据库，worker线程是单线程(处理计算，逻辑等是单线程) io线程是多线程，并发连接高(连接池、线程池)、 使用io多路复用连接，串行化，每个命令都是原子 操作。 本地方法：计算向数据移动？



• 一致性哈希和HashSlot对比

https://juejin.im/post/5b8f93576fb9a05d11175b8d

• redis 本身有持久化，为什么还要写进 mysql 呢

• redis非关系型数据库  
kv结构，数据关系组建  
困难

• redis+mysql双写一致性问题

http://km.oa.com/group/30346/articles/show/425999

### redis常见问题

* redis持久化
* 删除策略、淘汰策略
* 主从复制原理
* 缓存的三大问题
* 缓存和数据库数据一致性问题

### 数据类型

https://juejin.im/post/5d71d3bee51d453b5f1a04f1

* string

• int:8个字节的长整型。

• 二进制安全的

• redis  
kafka  
zk  
hbase

• embstr:小于等于39个字节的字符串。

• raw:大于39个字节的字符串

• 支持二进制位、字符串、数值、bitmap

• 场景

• session共享，kv缓存、fs文件系统(小文件)、数值计算计数器

• 底层sds

• 记录大小

* list

• 双端链表

• 同向lpush lpop 栈  
异向 队列  
lindex 数组  
ltrim 优化redis内存量

• 场景

• 消息队列  
评论列表

* hash

• rehash重现散列

• 场景

• 聚集列表  
详情页

* set

• 集合  
无序不重复  
  
不推荐  
集合交并差

• 场景

• 抽奖  
验证码  
扑克牌游戏  
共同的好友，

* sorted set  
  zset

• 跳跃表

• 1 每层是一个有序的链表  
2 最底层包含所有元素  
3 每个元素都是随机决定元素

• 场景

• 排行榜

### redis持久化

https://www.cnblogs.com/wdliu/p/9377278.html

* 配置文件设置

• bind: 0.0.0.0 任意ip都能链接  
protected-mode:保护模式，需要设置no，外网ip才能连接，如果设置了密码连接，改设置不生效  
daemonize：守护进程模式  
dir: 配置持久化文件

* 检测防火墙是否关闭:  
  firewall-cmd --state  
  关闭防火墙：  
  systemctl stop firewalld.service
* 分为RDB和AOF持久化方式

• RDB

• 是什么

• 会生成一个快照文件，写文件的时候会先把内存中的全量数据写到临时文件，然后在替换原先的RDB文件

• 配置

• # 时间策略  
save 900 1  
save 300 10  
save 60 10000  
  
# 文件名称  
dbfilename dump.rdb  
  
# 文件保存路径  
dir /home/work/app/redis/data/  
  
# 如果持久化出错，主进程是否停止写入  
stop-writes-on-bgsave-error yes  
  
# 是否压缩  
rdbcompression yes  
  
# 导入时是否检查  
rdbchecksum yes

• save 900 1 表示900s内如果有1条是写入命令，就触发产生一次快照，可以理解为就进行一次备份  
 禁用RDB配置，也是非常容易的，只需要在save的最后一行写上：save ""

• 什么时候触发rdb持久化机制

• 手动触发

• save

• 会阻塞当前redis服务，直到RDB文件写完为止。

• bgsave

• 对上述过程描述：  
  
1 客户端执行bgsave命令，redis主进程收到指令并判断此时是否在执行bgrewriteaof(AOF文件重新过程，后续会讲解)，如果此时正好在执行则bgsave直接返回，不fork子进程，如果没有执行bgrewriteaof重写AOF文件，则进入下一个阶段；  
2 主进程调用fork方法创建子进程，在创建过程中redis主进程阻塞，所以不能响应客户端请求；  
3 子进程创建完成以后，bgsave命令返回“Background saving started”，此时标志着redis可以响应客户端请求了；  
4 子经常根据主进程的内存副本创建临时快照文件，当快照文件完成以后对原快照文件进行替换；  
5 子进程发送信号给redis主进程完成快照操作，主进程更新统计信息（info Persistence可查看）,子进程退出；

• 自动触发

• 1 根据我们的 save m n 配置规则自动触发；  
2 从节点全量复制时，主节点发送rdb文件给从节点完成复制操作，主节点会触发 bgsave；  
3 执行 debug reload 时；  
4 执行 shutdown时，如果没有开启aof，也会触发  
5 slave请求full sync引起的rdb持久化

• kill进程是不会触发的

• 什么时候fork子进程

• 1 通过 fork 创建的子进程能够获得和父进程完全相同的内存空间，父进程对内存的修改对于子进程是不可见的，两者不会相互影响；  
2 通过 fork 创建子进程时不会立刻触发大量内存的拷贝，内存在被修改时会以页为单位进行拷贝，这也就避免了大量拷贝内存而带来的性能问题；

• AOF

• 是什么

• AOF 则以协议文本的方式，将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件

• 配置

• #是否打开aof持久化  
appendonly no  
  
# always: fsync after every write to the append only log. Slow, Safest.  
appendfsync always   
  
# everysec: fsync only one time every second. Compromise.  
appendfsync everysec  
  
# no: don't fsync, just let the OS flush the data when it wants. Faster  
appendfsync no

• 什么时候触发的

• 重写机制(瘦身-触发时机)

• 优化aof的存储命令

• 当子进程在执行 AOF 重写时， 主进程需要执行以下三个工作：  
  
处理命令请求。  
将写命令追加到现有的 AOF 文件中。  
将写命令追加到 AOF 重写缓存中。

• BGREWRITEAOF 命令

• 当子进程完成 AOF 重写之后， 它会向父进程发送一个完成信号， 父进程在接到完成信号之后， 会调用一个信号处理函数， 并完成以下工作：

• 将 AOF 重写缓存中的内容全部写入到新 AOF 文件中。  
对新的 AOF 文件进行改名，覆盖原有的 AOF 文件。  
当步骤 1 执行完毕之后， 现有 AOF 文件、新 AOF 文件和数据库三者的状态就完全一致了。  
  
当步骤 2 执行完毕之后， 程序就完成了新旧两个 AOF 文件的交替。  
  
这个信号处理函数执行完毕之后， 主进程就可以继续像往常一样接受命令请求了。 在整个 AOF 后台重写过程中， 只有最后的写入缓存和改名操作会造成主进程阻塞， 在其他时候， AOF 后台重写都不会对主进程造成阻塞， 这将 AOF 重写对性能造成的影响降到了最低

• 概要

• 配置

• 动态重写相关的配置有这么几项：auto-aof-rewrite-percentage用来指定AOF文件需要比旧AOF文件增大多少时才进行AOF重写。auto-aof-rewrite-min-size 用来指定AOF到达多大体积时才进行AOF重写，Redis 就是用这两个配置来控制AOF重写的（同时达到时才进行重写），整个重写的过程和BGSAVE比较相似，都是通过fork子进程来对AOF文件进行重写的

* rdb和aof混合持久化

• RDB 持久化能够快速地储存和恢复数据， 但是在服务器停机时却会丢失大量数据；  
AOF 持久化能够有效地提高数据的安全性， 但是在储存和恢复数据方面却要耗费大量的时间。

• 配置

• aof-use-rdb-preamble yes  
手动执行bgrewriteaof 命令

•

desc

• 原理

• 混合持久化是通过bgrewriteaof完成的，不同的是当开启混合持久化时，fork出的子进程先将共享的内存副本全量的以RDB方式写入aof文件，然后在将重写缓冲区的增量命令以AOF方式写入到文件，写入完成后通知主进程更新统计信息，并将新的含有RDB格式和AOF格式的AOF文件替换旧的的AOF文件。简单的说：新的AOF文件前半段是RDB格式的全量数据后半段是AOF格式的增量数据

* 查看文件命令od -c appendonly.aof
* 持久化选择和带来的问题

https://mp.weixin.qq.com/s?\_\_biz=MzI4NTA1MDEwNg==&mid=2650769300&idx=1&sn=49a11efa1a6ee605fceaddf240a55c40&chksm=f3f93201c48ebb175fa76053d95e315b621485b0e65e42d8b41fe91b8f859c9278f3adec7ca9&mpshare=1&scene=23&srcid=0731SR4C94CRM0Mljym0oEI3%23rd

* 通过分析：  
  1降低fork的频率，比如可以手动来触发RDB生成快照、与AOF重写；  
  2 控制Redis最大使用内存，防止fork耗时过长；  
  3使用高配置的硬件；  
  4合理配置Linux的内存分配策略，避免因为物理内存不足导致fork失败。

### 主从复制同步

* 开启主从复制，有3种方式：

• （1）配置文件  
在从服务器的配置文件中加入：slaveof <masterip> <masterport>  
（2）启动命令  
redis-server启动命令后加入 --slaveof <masterip> <masterport>  
（3）客户端命令  
Redis服务器启动后，直接通过客户端执行命令：slaveof <masterip> <masterport>，则该Redis实例成为从节点。

* 原理

• 初始化阶段

• 从库向主库发送socket连接

• 从库发送ping命令，主库返回pong

• 如果主库设置了密码，从库会发送auth命令校验

• 从节点发送自己的监听端口，主库保存其端口信息

• 数据同步

• 从节点向主节点发送psync命令  
（Redis2.8以前是sync命令）

• sync

• 向主库发送sync命令，主库收到sync命令后执行bgsave后台保存RDB快照。当快照完成以后，主库将快照文件已经缓存的所有命令发送给从库，从库接受到快照文件并载入，再将执行主库发送的命令，也就是上面我们介绍的复制初始化阶段和数据同步阶段，其后就是命令增量同步，最终主库与从库保持数据一直

• 根据主从节点当前状态的不同，可以分为全量复制和部分复制

• 在数据同步阶段之前，从节点是主节点的客户端，主节点不是从节点的客户端；而到了这一阶段及以后，主从节点互为客户端。原因在于：在此之前，主节点只需要响应从节点的请求即可，不需要主动发请求，而在数据同步阶段和后面的命令传播阶段，主节点需要主动向从节点发送请求（如推送缓冲区中的写命令），才能完成复制

• 命令传播

• 在命令传播阶段，除了发送写命令，主从节点还维持着心跳机制：PING和REPLCONF ACK {offset} 其中offset指从节点保存的复制偏移量

• 乐观复制

### 哨兵sentinel

* 作用

•

desc

• 监控

• 通知

• 当监控的服务器出问题了，向其他(服务器，哨兵)发送通知

• 自动故障转移

* 说明

• 哨兵是一个分布式系统，类似zookeeper，用于对主从结构中的每台服务器进行监控，当发生故障时，通过投票机制选择新的master并将所欲salve连接到新的master。  
哨兵也是一台服务器，只是对外不提供数据，通常  
配置数量为1、3、5、7等单数，选举问题

* 启动方式

• redis-sentinel sentinel-26379.conf  
redis-server sentinel-26379.conf --sentinel

* 原理

• 定时任务

• 每个哨兵节点维护了3个定时任务。定时任务的功能分别如下：每10s通过向主从节点发送info命令获取最新的主从结构；每2s通过发布订阅功能获取其他哨兵节点的信息；每1s通过向其他节点发送ping命令进行心跳检测，判断是否下线。

• 主观下线(SDOWN)

• Subjectively Down在心跳检测的定时任务中，如果其他节点超过一定时间没有回复，哨兵节点就会将其进行主观下线

• 客观下线(ODOWN)

• 增对主节点才会有的，如果判断主节点下线的哨兵数量达到一定数值，则对该主节点进行客观下线。 slaves和sentinel不会有ODOWN状态

• 选举领导者哨兵节点

• 1 跟master断开连接的时长   
2 slave的优先级 （由slave-priority配置指定）  
3 复制偏移量offset   
4 实例运行的id（run id

• 选举算法采用Raft算法，类似与zookpeer  
一般来说，哨兵选择的过程很快，谁先  
完成客观下线，一般就能成为领导者

• 故障转移

• 修改主节点、从节点、哨兵等各个信息配置。。

### 集群cluster

* Redis hash tag进行分槽导致的问题
* 搭建集群

• 启动节点配置

• cluster-enabled yes  
cluster-config-file "node-7000.conf"

• 节点握手

• 节点握手使用cluster meet {ip} {port}命令实现

• 分配槽

• 集群有16384个槽  
(可能需要清楚脏数据flushall, cluster reset )

• redis-cli -h 9.134.21.131 -p 7000 cluster addslots {0..5461}

• CRC16算法产生的hash值有16bit,  
2^16-=65536个值,  
最占空间的是myslots[CLUSTER\_SLOTS/8]。  
当槽位为65536时，这块的大小是:  
65536÷8÷1024=8kb  
而如果是：16384÷8÷1024=2kb

• 指定主从关系

• 每个从节点指定主节点

• redis-cli -p 8000 cluster replicate be816eba968bc16c884b963d768c945e86ac51ae

• 登录加redis-cli -p 7000 -h 9.134.21.131 -c

• 集群的扩容缩容

• 原理

• 通讯协议

• 使用二进制协议进行通讯。 节点之间使用 Gossip 协议 来进行以下工作： (a).传播（propagate）关于集群的信息，以此来发现新的节点; (b).向其他节点发送 PING 数据包，以此来检查目标节点是否正常运作; (c).在特定事件发生时，发送集群信息. 除此之外， 集群连接还用于在集群中发布或订阅信息

• GOSSIP协议:Gossip消息有ping消息、pong消息、meet消息、fail消息. meet消息：用于通知新节点加入，消息发送者通知接收者加入到当前集群，meet消息通信完后，接收节点会加入到集群中，并进行周期性ping pong交换

• 故障转移

• 主要环节包括了:主观下线和客观下线；

• 故障恢复

• 1，资格检查(从新选择主节点)

• 2 准备选举时间：

• 3，发起选举

• 4，选举投票

• 数据迁移

• 只需要迁移槽，状态标记就行

### client

* RESP协议

### redis需要掌握的知识点

https://youzhixueyuan.com/detailed-explanation-of-redis.html

### redis数据一致性

<https://coolshell.cn/articles/17416.html>

* 缓存使用方式

• 先更新数据库，在更新缓存

• 当更新缓存成功，更新数据库失败时，数据库中就是旧数据

• 先更新数据库，再更新缓存

• 1 在多线程并发时，会出现不一致行为。假设请求 A 先操作数据库，请求 B 后操作数据库，但是可能存在请求 B 先写缓存，请求 A 后写缓存的情况，从而导致数据库与缓存之间的数据不一致

• 先删缓存， 再写数据库

• 请求a先删除缓存，请求b获取旧的数据库的数据，  
请求a存数据到db，请求b将旧值写到缓存

• 先写数据库，再删除缓存

• 1 a查询旧值，b更新数据库，b更新新数据缓存，a在更新旧数据缓存  
2 主从读写问题。  
3 缓存删除失败问题

* 缓存问题

• 缓存穿透

• 查询数据库和缓存不存在的key

• bloom filter

• 缓存击穿

• 热key缓存失效，大量的请求  
到达系统

• 1 热点数据进行二级缓存，不同  
级别的缓存设置不同的过期时间，  
2 缓存延迟失效，回源中数据

• 缓存雪崩

• 1 redis挂了  
2 redis设置的key，同时过期了

• 1redis高可用哨兵+集群模式  
2 设置随机的key  
3 本地二级缓存+限流  
4 redis持久化，快速恢复服务

## Mysql

### 事务 正常都是活动乐观锁，都是用的read commited， 因为需要获取实时的版本号，金融级别一般都设置为

* ACID

https://www.toutiao.com/i6837304877706641923/

• 原子性(atomicity)

• 原子性是指事务的原子性操作，对数据的修改要么全部执行成功，要么全部失败，实现事务的原子性，是基于日志的Redo/Undo机制

• Redo log用来记录某数据块被修改后的值，可以用来恢复未写入 data file 的已成功事务更新的数据；Undo log是用来记录数据更新前的值，保证数据更新失败能够回滚

• example：假如某个时刻数据库崩溃，在崩溃之前有事务A和事务B在执行，事务A已经提交，而事务B还未提交。当数据库重启进行 crash-recovery 时，就会通过Redo log将已经提交事务的更改写到数据文件，而还没有提交的就通过Undo log进行roll back

• 一致性Consistent

• 一致性是指执行事务前后的状态要一致，可以理解为数据一致性

• 隔离性Isalotion

读未提交是没有加任何锁的 串行化加的是一把大锁，读的时候加共享锁，不能写，写的时候，加的是排它锁，阻塞其它事务的写入和读取，若是其它的事务长时间不能写入就会直接报超时，所以它的性能也是最差的 读已提交和可重复读底层采用的是MVCC（多版本并发控制）方式实现

• 隔离性侧重指事务之间相互隔离，不受影响，这个与事务设置的隔离级别有密切的关系。

• 读未提交（READ UNCOMMITTED

• 该级别下的事务允许脏读，但不允许更新丢失：当一个事务开始还未提交，修改某个值时，其他的事务可以读这个值，但是无法修改这个值

• 读提交 （READ COMMITTED）

• 事务提交，多次读取会造成不一样的结果,在该级别下允许不可重复读，但不允许脏读。同样的无法修改这个值

• 目前活动开发使用的cdb默认数据隔离级别为Read Committed，因为不可重复读，所以可以利用此特点在更新时加上检查条件防止并发（和数据版本类似）；但是如果选择了更高一级的Repeatable Read，那么在更新时加上条件也有可能因为这个值是之前被获取过的旧值而导致防并发失败。

• 可重复读 （REPEATABLE READ）  
MySQL 默认的隔离级别

• 读取不到别的已提交的事务，读取不到后，但是插入报错等问题，mysql称之为幻读。禁止不可重复读，但可能出现幻读。

• 串行化 （SERIALIZABLE）

• 隐式加一把读共享锁，从而避免了脏读、不可重读复读和幻读问题

• 因为隔离级别产生的：

• 更新丢失

• 两个事务同时修改一行数据，其中一个事务的更新被另外一个事务的更新覆盖了

• 脏读

• 一个事务在修改（未提交）数据时，被另一个事务读到了修改后的数据

• 不可重复读

• 一个事务先读取了某个数据，然后另外一个事务修改（并提交）了数据后，这个事务再一次读取之前的数据时，得到的是不一样的值（即不能读到相同的数据内容）

• 幻读

• 一个事务预先读取到或者未读取到某条数据，然后另外一个事务删除或新增这条数据（并提交）后，这个事务再一次读取之前的数据时未读到或读到数据。

• 锁的问题：因为四大等级从上到下，隔离的效果是逐渐增强，但是性能却是越来越差，所以引入了锁的问题

• 共享锁/读锁（Shared Locks）

• 读加锁，不能写并且可并行读。  
GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。

• 排他锁/写锁（Exclusive Locks）

• 写加锁，其它读写都阻塞

• 间隙锁(for update)

• Gap Locks会锁住两个索引之间的区间，比如select \* from User where id>3 and id<5 for update，就会在区间（3，5）之间加上Gap Locks

• Next-Key Locks是Gap Locks+Record Locks形成闭区间锁select \* from User where id>=3 and id=<5 for update，就会在区间[3,5]之间加上Next-Key Locks

• 行锁和表锁，是从锁的粒度上进行划分的

• 表锁

• 表锁则锁的粒度大，加锁快，开销小，但是锁冲突的概率大，并发度低

• 行锁（Record Locks）

• 行锁锁定当前数据行，锁的粒度小，加锁慢，发生锁冲突的概率小，并发度高，行锁也是MyISAM和InnoDB的区别之一，InnoDB支持行锁并且支持事务

• MVCC，Multi-Version Concurrency Control，多版本并发控制

• MVCC时用到了一致性视图，用于支持读提交和可重复读的实现

• 只需要在事务开始的时候创建一致性视图，也叫做快照，之后的查询里都共用这个一致性视图，后续的事务对数据的更改是对当前事务是不可见的，这样就实现了可重复读

• 而读提交，每一个语句执行前都会重新计算出一个新的视图，这个也是可重复读和读提交在MVCC实现层面上的区别

• 快照需要遵循的规则

• 一个事务视图来说除了对自己更新的总是可见，另外还有三种情况：版本未提交的，都是不可见的；版本已经提交，但是是在创建视图之后提交的也是不可见的；版本已经提交，若是在创建视图之前提交的是可见的

• mvcc底层工作

• 系统版本号：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。

• 事务版本号：事务开始时的系统版本号

• Mysql中的每一个数据行都有可能存在多个版本，在每次事务更新数据的时候，都会生成一个新的数据版本，并且把自己的数据id赋值给当前版本的row trx\_id，如果

•

desc

• 实际上版本1、版本2并非实际物理存在的，而图中的U1和U2实际就是undo log，这v1和v2版本是根据当前v3和undo log计算出来的

• 持久性Durable

• 持久性则是指在一个事务提交后，这个事务的状态会被持久化到数据库中，也就是事务提交，对数据的新增、更新将会持久化到书库中

* 「面试官：」 假如两个事务执行写操作，又怎么保证并发呢？  
    
  「我：」 假如事务1和事务2都要执行update操作，事务1先update数据行的时候，先回获取行锁，锁定数据，当事务2要进行update操作的时候，也会去获取该数据行的行锁，但是已经被事务1占有，事务2只能wait。  
    
  「我：」 若是事务1长时间没有释放锁，事务2就会出现超时异常 。  
    
  「面试官：」 这个是在update的where后的条件是在有索引的情况下吧？  
    
  「我：」 嗯，是的 。  
    
  「面试官：」 那没有索引的条件下呢？没办法快速定位到数据行呢？  
    
  「我：」 若是没有索引的条件下，就获取所有行，都加上行锁，然后Mysql会再次过滤符合条件的的行并释放锁，只有符合条件的行才会继续持有锁。  
    
  「我：」 这样的性能消耗也会比较大。  
    
  「面试官：」 嗯嗯

### 索引问题 - 索引失效、联合索引，索引失效问题 sql优化

* 聚集索引 将数据存储与索引放到了一块，找到索引也就找到了数据 mysql就是主键,叶子节点存放了所有数据
* 非聚集索引， 将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行 就是联合索引、唯一索引、普通索引等，二次查询等问题，只有 MyISAM 存储引擎支持全文索引
* 覆盖索引：SQL只需要通过索引就可以返回查询所需要的数据，而不必通过非聚簇索引查到主键之后再去查询数据，避免二次查找(也叫做回表查询)
* 回表查询  
  先通过普通索引的值定位聚簇索引值，再通过聚簇索引的值定位行记录数据，需要扫描两次索引B+树，它的性能较扫一遍索引树更低。

为什么联合索引索引失效

因为B+树是按照从左到右的顺序来建立收索树的，建立收索树，第一个元素就是比较因子。

### Mysql索引为什么用B+树(https://tech.meituan.com/2014/06/30/mysql-index.html)

### 为什么使用B+树 hash:hash冲突问题、hash内存空间占用、无法进行范围查找 二叉树：蜕变成链表 avl树：二叉查找树、树的平衡插入需要调整等问题

红黑树：数据存储问题，所以需要n叉树

内存跟磁盘进行交换时候，有最小的逻辑单元，称之为页，datapage，一般大小由操作系统(4k)决定，一般为4k或者8k,进行页的整数倍读取，mysql读的是16k。

b树：n叉树，每次读取磁盘块，读取16k，非叶子节点存的是磁盘块指针+索引+data，这样暂用数据，而你只有16k的数据。如果每个data占用1k数据，拿3叉树就只能存16\*16\*16个4096个数据。  
b+树：每个磁盘块不存data数据了，索引和指针大小就很小了，最底层叶子节点才存的真正的数据，并且顺序存储，这样就能存很多数据了。树的高度就更低了。

非叶子节点只存key，大大减少了非叶子的大小，这样每个节点就可以存放更多的记录，树更矮了，io操作更少。叶子节点都是相链的。对整棵树值需要一次遍历即可。数据顺序排列且相连，利于区间查找和收索。B树没有。

### 聊聊MySQL索引的发展过程？是一来就是B+Tree的么？从 没有索引、hash、二叉排序树、AVL树、B树、B+树 聊。

### Mysql脏页是什么 内存数据，和硬盘数据不一致的时候，称为脏页 如何刷新脏页 (1)redo log写满时，没有看见了，此时需要将checkpoint向前推进，推进的这部分日志对应的脏页刷入到磁盘，此时所有的更新全部阻塞，此时写的性能变为0，必须待刷一部分脏页后才能更新。(2)系统内存不足时，需要将一部分数据页淘汰掉，如果淘汰的是脏页，需要先将脏页同步到磁盘。

### Mysql脏读，不可重复读，幻读 脏读：指一个线程中的事务读取到了另外一个线程中未提交的数据。 不可重复读（虚读）：指一个线程中的事务读取到了另外一个线程中提交的update的数据。 C:\Users\ADMINI~1\AppData\Local\Temp\1601110372(1).jpg 幻读：指一个线程中的事务读取到了另外一个线程中提交的insert的数据。 read uncommitted——不作任何隔离，具有脏读、不可重复读、幻读问题read committed——可防止脏读，不能防止不可重复读和幻读问题repeatable read——可以防止脏读、不可重复读，不能防止幻读问题（mysql默认是这个隔离级别）serializable——数据库运行在串行化，上述问题都可以防止，只是性能非常低

### Mysql中redo log undo log binlog 因为innodb为了提供性能，所以在读写的时候先执行缓冲池 redolog: 确保事务的持久性。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。 undolog:事物回滚使用，系统错误或者rollback操作 .数据库中，数据在内存中叫data buffer，数据在磁盘上叫data file。事务的日志也一样，在内存中叫log buffer，在磁盘上叫log file。 undolog: 用于记录事务开始前的状态，用于事务失败时的回滚操作。 二阶段提交： 先进入commit prepare 阶段： 事务中新生成的redo log 会被刷到磁盘，并将回滚段置为prepared状态。binlog不作任何操作。存储引擎写redo log commit阶段： innodb释放锁，释放回滚段，设置redo log提交状态，binlog持久化到磁盘，然后存储引擎层提交。Server层写binlog

### Mysql ACID完美总结

1 事物的原子性是通过undo log来实现的  
2 事物的持久性是通过redo log来实现的  
3 事物的隔离性是通过(锁+mvcc) 来实现的  
4 一致性是通过原子性、持久性、隔离性来实现的  
一致性：在事务开始和完成时，数据都必须保持一致状态  
ACID只是个概念，事务最终目的是要保障数据的可靠性，一致性。

### Innodb和myisam的区别和实现

<https://blog.csdn.net/qq_41706670/article/details/92836395>  
读：MyISAM 默认会把索引读入内存，直接在内存中操作；InnoDB 则是 I/O 操作。（参考stackoverflow，具体url忘记了）写：MyISAM 是表锁；InnoDB 是行锁。

### Myisam为什么查询速度快

1）数据块，INNODB要缓存，MYISAM只缓存索引块，  这中间还有换进换出的减少；2）innodb寻址要映射到块，再到行，MYISAM记录的直接是文件的OFFSET，定位比INNODB要快3）INNODB还需要维护MVCC一致；虽然你的场景没有，但他还是需要去检查和维护

myisam没有事务支持，它的连续的插入和查询速度都比Innodb快很多，但是如果需要插入和查询穿插着来，那么myisam是表锁，innodb是行锁，innodb的并发性好，并且innodb是支持事务的innodb在插入数据的时候需要维护表级缓存，myisam只需要维护索引(文件级offset定位数据行，不需要缓存表)innodb在插入和查询的时候需要维护mvccinnodb在插入时维护主外键关系等innodb因为提供了事务支持和表级锁的支持，维护成本比myisam高很多也就是说，如果读写，一半一半，平均，且考虑并发和事务的要求使用innodb如果读多写少，或者写多读少，比如题库表(读多写少)，或者日志表(写多读少)，而且不是需要事务的情况下(题库修改很少，极少出现事务并发问题，新增也不多，但是会有大量的读操作，新增的时候加入程序级别的锁，防止并发插入，读的时候，直接读，myisam的表读锁之间是共享的)，这个时候就可以考虑使用myisam来增强数据库的读写性能还有就是目前的缓存技术的支持，innodb利用缓存之后，读写能力都有了很大的提升，如果不是非要走数据库查询的话，优化缓存，并利用好二级缓存也是个不错的策略

### Mysql行锁和表锁的区别

InnoDB的行锁是针对索引加的锁，不是针对记录加的锁。并且该索引不能失效，否则都会从行锁升级为表锁。  
默认update、insert、delete都会加上排他锁

间隙锁，范围查找的时候  
1 InnoDB 支持表锁和行锁，使用索引作为检索条件修改数据时采用行锁，否则采用表锁。

2 InnoDB 自动给修改操作加锁，给查询操作不自动加锁

3 行锁可能因为未使用索引而升级为表锁，所以除了检查索引是否创建的同时，也需要通过explain执行计划查询索引是否被实际使用。

4 行锁相对于表锁来说，优势在于高并发场景下表现更突出，毕竟锁的粒度小。

5 当表的大部分数据需要被修改，或者是多表复杂关联查询时，建议使用表锁优于行锁。

6 为了保证数据的一致完整性，任何一个数据库都存在锁定机制。锁定机制的优劣直接影响到一个数据库的并发处理能力和性能。

### Mysql间隙锁

必须在rr级别，并且检索条件必须有所以。 这种程度上，可以防止一定的幻读  
当开启事物的时候，在rc级别下，当更新的数据中有索引，那就是行锁锁定，如果没有，退化成表锁了  
行锁加Gap(间隙锁)，形成的next-key锁，解决RR级别下的幻读。间隙锁是左闭右开区间。如索引id，里头数据有2-5-10条数据，如果第一条update更新5数据，另外一个事物开启的时候 ，这时候间隙锁会锁住[2,5)和[5,10)区别，左闭右开记住。解决幻读插入问题。正常情况一下，一般是用悲观锁加rr级别来解决幻读问题。 Id >5 for update

### 索引优缺点

优点：

1.通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

2.可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

3.可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

4.在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

5.通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

缺点：

1.创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

2.索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

3.当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。

哪些字段适合建索引

1.在经常需要搜索的列上，可以加快搜索的速度；

2.在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；

3.在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；

4.在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

5.在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

6.在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

哪些字段不适合建索引

1.对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

2.对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

3.对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

4.当修改性能远远大于检索性能时，不应该创建索引。这是因为，修改性能和检索性能是互相矛盾的。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

### 子账号跨机房容灾怎么做的

### Mysql主从同步 mysql 主从同步怎么搞的？分哪几个过程？如果有一台新机器要加到从机里，怎么个过程

方法一：

原理：主从复制涉及三个线程：log dump thread ，i/o thread ,SQL thread .

当从节点连接到主节点时**，**主节点会创建一个log dump 线程，用于发送bin-log的内容。在读取bin-log中的操作时，此线程会对主节点上的bin-log加锁，当读取完成，甚至在发动给从节点之前，锁会被释放。

l 从节点I/O线程

当从节点上执行`start slave`命令之后，从节点会创建一个I/O线程用来连接主节点，请求主库中更新的bin-log。I/O线程接收到主节点binlog dump 进程发来的更新之后，保存在本地relay-log中。

l 从节点SQL线程

SQL线程负责读取relay log中的内容，解析成具体的操作并执行，最终保证主从数据的一致性

方法二：

1 主库执行新的事务，commit时，更新 show master status\G ，触发一个信号给2 binlog dump 接收到主库的 show master status\G信息，通知从库日志更新了3 从库IO线程请求新的二进制日志事件4 主库会通过dump线程传送新的日志事件，给从库IO线程5 从库IO线程接收到binlog日志，当日志写入到磁盘上的relaylog文件时,给主库ACK\_receiver线程6 ACK\_receiver线程触发一个事件，告诉主库commit可以成功了7如果ACK达到了我们预设值的超时时间，半同步复制会切换为原始的异步复制。

金融级别的mysql，3个或者5个，通过选举方式，重新获取leader，并且选择binglog最多的作为leader，如果开始主maser写binlog还没同步给slave后挂了，根据半同步，其实业务测也没有感知到有没有成功，及时切换了slave，然后选举出leader也没有问题，因为这个事物还没提交，只要报错或者重试就可以了

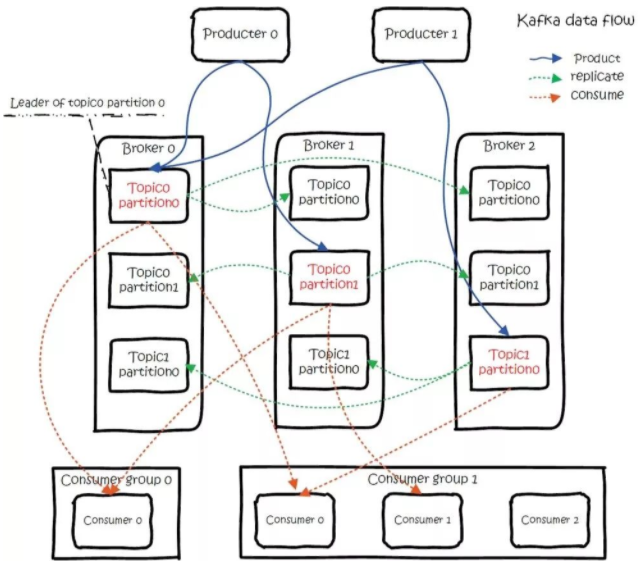
## Zookeeper和kafka

* + 1. 一致性理论祖先  
       https://www.jianshu.com/p/40dbe406d2f4
    2. Cap原理  
       (1) CA: 优先保证一致性和可用性，放弃分区容错。 这也意味着放弃系统的扩展性，系统不再是分布式的，有违设计的初衷。(2) CP: 优先保证一致性和分区容错性，放弃可用性。在数据一致性要求比较高的场合(譬如:zookeeper,Hbase) 是比较常见的做法，一旦发生网络故障或者消息丢失，就会牺牲用户体验，等恢复之后用户才逐渐能访问。(3) AP: 优先保证可用性和分区容错性，放弃一致性。NoSQL中的Cassandra 就是这种架构。跟CP一样，放弃一致性不是说一致性就不保证了，而是逐渐的变得一致。 BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的简写。
    3. 为什么使用消息队列

异步处理 - 相比于传统的串行、并行方式，提高了系统吞吐量。应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。流量削锋 - 可以通过消息队列长度控制请求量；可以缓解短时间内的高并发请求。日志处理 - 解决大量日志传输。消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯的 消息通讯。比如实现点对点消息队列，或者聊天室等。

* + 1. kafka topic怎么创建

kafka对外使用topic的概念，生产者往topic里写消息，消费者从读消息。为了做到水平扩展，一个topic实际是由多个partition组成的，遇到瓶颈时，可以通过增加partition的数量来进行横向扩容。单个parition内是保证消息有序。

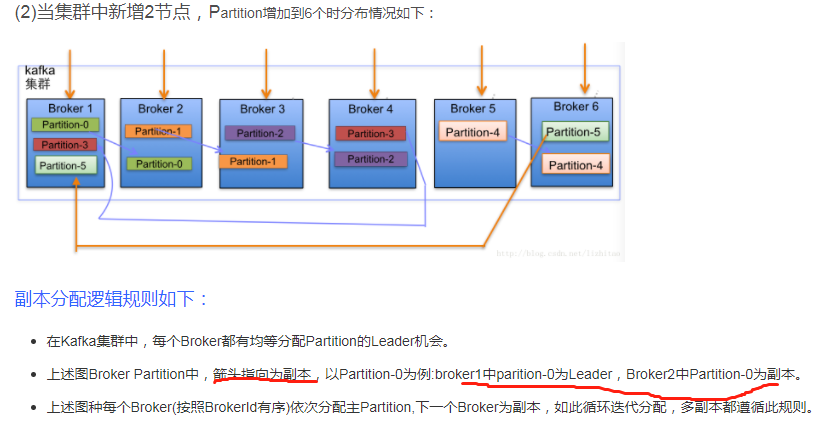


* + 1. raft算法和zab(zk)选主算法
    2. Kafka 选主怎么做的、leader容灾？  
       controller会在Zookeeper的/brokers/ids节点上注册Watch，一旦有broker宕机，它就能知道。当broker宕机后，controller就会给受到影响的partition选出新leader。controller从zk的/brokers/topics/[topic]/partitions/[partition]/state中，读取对应partition的ISR（in-sync replica已同步的副本）列表，选一个出来做leader。选出leader后，更新zk，然后发送LeaderAndISRRequest给受影响的broker，让它们改变知道这事。为什么这里不是使用zk通知，而是直接给broker发送rpc请求，我的理解可能是这样做zk有性能问题吧。如果ISR列表是空，那么会根据配置，随便选一个replica做leader，或者干脆这个partition就是歇菜。如果ISR列表的有机器，但是也歇菜了，那么还可以等ISR的机器活过来。
    3. Kafka 为什么这么快？  
       1 Cache Filesystem Cache PageCache缓存2 顺序写 由于现代的操作系统提供了预读和写技术，磁盘的顺序写大多数情况下比随机写内存还要快。3 Zero-copy 零拷技术减少拷贝次数4 Batching of Messages 批量量处理。合并小的请求，然后以流的方式进行交互，直顶网络上限。5 Pull 拉模式 使用拉模式进行消息的获取消费，与消费端处理能力相符。  
       1 顺序写，但没有办法删除，每个消费者（Consumer）对每个Topic都有一个offset用来表示读取到了第几条数据  
       2 mmap内存映射  
       使用这种方式可以获取很大的I/O提升，省去了用户空间到内核空间复制的开销（调用文件的read会把数据先放到内核空间的内存中，然后再复制到用户空间的内存中。）但也有一个很明显的缺陷——不可靠，写到mmap中的数据并没有被真正的写到硬盘，操作系统会在程序主动调用flush的时候才把数据真正的写到硬盘。Kafka提供了一个参数——producer.type来控制是不是主动flush，如果Kafka写入到mmap之后就立即flush然后再返回Producer叫 同步 (sync)；写入mmap之后立即返回Producer不调用flush叫异步 (async)  
       3 读取数据使用zerocopy sendfile实现  
       传统的是操作：硬盘->内核buf->用户buf->socket相关缓冲区->协议引擎  
       4 批量压缩数据  
       总结：  
       Kafka速度的秘诀在于，它把所有的消息都变成一个批量的文件，并且进行合理的批量压缩，减少网络IO损耗，通过mmap提高I/O速度，写入数据的时候由于单个Partion是末尾添加所以速度最优；读取数据的时候配合sendfile直接暴力输出。
    4. kafka 与 rabbitmq区别
    5. kafka 分区怎么同步的
    6. kafka 怎么保证不丢消息的，消息不被重复消费。  
       0 不等broker确认消息  
       1 leader已经确认消息，replica异步拉取  
       -1 所有都返回确认  
       kafka有个offset的概念，每个消息写入都会有一个offset，每个一段时间会提交给offset\_topic，并且也会提交到zk，服务重启的时候，去zk上面拉取  
       消费自己提交了offset，让kafka认为你消费完了。自动提交，这时候关闭自动提交，业务侧自己手动提交即可。  
       Kafka 弄丢了数据　　这块比较常见的一个场景，就是 Kafka 某个 broker 宕机，然后重新选举 partition 的 leader。大家想想，要是此时其他的 follower 刚好还有些数据没有同步，结果此时 leader 挂了，然后选举某个 follower 成 leader 之后，不就少了一些数据？这就丢了一些数据啊。　　生产环境也遇到过，我们也是，之前 Kafka 的 leader 机器宕机了，将 follower 切换为 leader 之后，就会发现说这个数据就丢了。　　所以此时一般是要求起码设置如下 4 个参数：给 topic 设置 replication.factor 参数：这个值必须大于 1，要求每个 partition 必须有至少 2 个副本。在 Kafka 服务端设置 min.insync.replicas 参数：这个值必须大于 1，这个是要求一个 leader 至少感知到有至少一个 follower 还跟自己保持联系，没掉队，这样才能确保 leader 挂了还有一个 follower 吧。在 producer 端设置 acks=all：这个是要求每条数据，必须是写入所有 replica 之后，才能认为是写成功了。在 producer 端设置 retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。　　我们生产环境就是按照上述要求配置的，这样配置之后，至少在 Kafka broker 端就可以保证在 leader 所在 broker 发生故障，进行 leader 切换时，数据不会丢失。　　生产者会不会弄丢数据？　　如果按照上述的思路设置了acks=all，一定不会丢，要求是，你的 leader 接收到消息，所有的 follower 都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。
    7. kafka 为什么可以扛住这么高的qps
    8. kafka如何扩容
    9. **.Kafka中是怎么体现消息顺序性的？**

kafka每个partition中的消息在写入时都是有序的，消费时，每个partition只能被每一个group中的一个消费者消费，保证了消费时也是有序的。整个topic不保证有序。如果为了保证topic整个有序，那么将partition调整为1.

* + 1. **kafka follower如何与leader同步数据**

Kafka的复制机制既不是完全的同步复制，也不是单纯的异步复制。完全同步复制要求All Alive Follower都复制完，这条消息才会被认为commit，这种复制方式极大的影响了吞吐率。而异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下，如果leader挂掉，会丢失数据，kafka使用ISR的方式很好的均衡了确保数据不丢失以及吞吐率。Follower可以批量的从Leader复制数据，而且Leader充分利用磁盘顺序读以及send file(zero copy)机制，这样极大的提高复制性能，内部批量写磁盘，大幅减少了Follower与Leader的消息量差。

* + 1. kafka partition broker consumer producer group topic 等都是啥关系？  
       Partition: Topic物理上的分组，一个topic可以分为多个partion,每个partion是一个有序的队列。partion中每条消息都会被分配一个 有序的Id(offset)   
       broker:  
       是消息的代理，producer往broker里面指定的topic写消息，consumer从broker里面拉取指定topic的消息 Kafka集群中的一台或多台服务器统称broker.  
       message:  
       消息，是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息   
       总结：  
       每个机器一般只有一个broker，多台机器组成集群，broker可以有多个topic，并且partition也分多个。在个个broker上。做好容灾
    2. Kafka可以脱离zookeeper单独使用吗，为什么
    3. Kafka有几种数据保留的策略
    4. Kafka设置了7天和10G清楚数据，第五天消息到达了10G,这时候kafka将如何处理
    5. 什么情况下回导致kafka变慢
    6. Zookeeper怎么保证主从节点的状态同步
    7. 说一下zookeeper的通知机制
    8. Kafka如何保证消息的顺序性  
       partion中的数据本身就有序，如果多线程处理，处理将相同的key的数据存到同一个queue中，每个线程分别消费一个queue的，保证有序
    9. Kafka或者mq消息过期失效、消息积压。  
       消息积压，需要新建topic，写过临时的分发数据partion，partition也增加10倍的机器，等消费处理完后，在恢复原来的数据。  
       过期失效，本身kafka认为就是不可靠的，写临时程序，补单。或者本身自身也有队长，补单逻辑等。

## rpc框架

1 统一协议。开发支持多平台、可扩展、协议代码自动生成统一协议。通常有protobuf，thirft、jce等。开发人员只需要关系通信协议的字段，不需要关注其细节。  
2 公共主键，中间件的集成。  
3 框架自身做到稳定，高可用，高性能  
4 平台运维问题，服务管理，做到负载均衡，容量管理、容错保护、过载保护、消息染色、idc分组、set分组、数据监控、配置集中、灰度发布，就近接入等。

## 进程线程间通信

互斥锁、条件变量、读写锁、信号量

共享内存、消息队列，管道、socket、文件

共享内存：最快的ipc，同一块物理内存被映射到进程A、B各自的进程地址空间，其他需要四次，共享内存则只拷贝两次数据[1]：一次从输入文件到共享内存区，另一次从共享内存区到输出文件

## 锁的实现

## 操作系统

* + 1. Linux内存分配机制  
       缺页中断：  
       malloc和mmap调用后，并没有实际分配内存，这时候当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个缺页中断  
       fork命令  
       当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作：   
       1、检查要访问的虚拟地址是否合法   
       2、查找/分配一个物理页   
       3、填充物理页内容（读取磁盘，或者直接置0，或者啥也不干）   
       4、建立映射关系（虚拟地址到物理地址）   
       mmap:(内存映射)  
        磁盘文件和内存中某块地址相关联。  
       malloc分配释放内存也是通过brk和mmap这些系统调用，分配虚拟内存，小于128k用brk，反之。Brk都是往高地址分配内存  
         
       malloc分配释放内存也是通过brk和mmap这些系统调用，分配虚拟内存，小于128k用brk，反之。brk分配的内存需要等高地址的内存释放以后才能释放，这就是内存碎片产生的原因。而mmap分配的内存可以单独释放，可以避免内存碎片的产生。brk指针只有一个，如果先开辟的内存空间释放了，那么那块内存会空在那里，如果此时有一个申请同样大小内存的请求，就可以复用，不然就是碎片一直在那里。内存紧缩操作，当brk指针最高地址有超过128k的空闲内存，brk指针就会回退，执行紧缩操作。  
       既然堆内内存brk和sbrk不能直接释放，为什么不全部使用 mmap 来分配，munmap直接释放呢？         既然堆内碎片不能直接释放，导致疑似“内存泄露”问题，为什么 malloc 不全部使用 mmap 来实现呢(mmap分配的内存可以会通过 munmap 进行 free ，实现真正释放)？而是仅仅对于大于 128k 的大块内存才使用 mmap ？         其实，进程向 OS 申请和释放地址空间的接口 sbrk/mmap/munmap 都是系统调用，频繁调用系统调用都比较消耗系统资源的。并且， mmap 申请的内存被 munmap 后，重新申请会产生更多的缺页中断。例如使用 mmap 分配 1M 空间，第一次调用产生了大量缺页中断 (1M/4K 次 ) ，当munmap 后再次分配 1M 空间，会再次产生大量缺页中断。缺页中断是内核行为，会导致内核态CPU消耗较大。另外，如果使用 mmap 分配小内存，会导致地址空间的分片更多，内核的管理负担更大。        同时堆是一个连续空间，并且堆内碎片由于没有归还 OS ，如果可重用碎片，再次访问该内存很可能不需产生任何系统调用和缺页中断，这将大大降低 CPU 的消耗
    2. 进程调度算法、哪些是抢占式的
    3. 操作系统中的页表寻址
    4. 虚拟内存置换方式

参考回答：比较常见的内存替换算法有：FIFO，LRU，LFU，LRU-K，2Q。1、FIFO（先进先出淘汰算法）思想：最近刚访问的，将来访问的可能性比较大。实现：使用一个队列，新加入的页面放入队尾，每次淘汰队首的页面，即最先进入的数据，最先被淘汰。弊端：无法体现页面冷热信息2、LFU（最不经常访问淘汰算法）思想：如果数据过去被访问多次，那么将来被访问的频率也更高。实现：每个数据块一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序。每次淘汰队尾数据块。开销：排序开销。弊端：缓存颠簸。3、LRU（最近最少使用替换算法）思想：如果数据最近被访问过，那么将来被访问的几率也更高。实现：使用一个栈，新页面或者命中的页面则将该页面移动到栈底，每次替换栈顶的缓存页面。优点：LRU算法对热点数据命中率是很高的。缺陷：1）缓存颠簸，当缓存（1，2，3）满了，之后数据访问（0，3，2，1，0，3，2，1。。。）。2）缓存污染，突然大量偶发性的数据访问，会让内存中存放大量冷数据。4、LRU-K（LRU-2、LRU-3）思想：最久未使用K次淘汰算法。LRU-K中的K代表最近使用的次数，因此LRU可以认为是LRU-1。LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。相比LRU，LRU-K需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到K次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K会淘汰第K次访问时间距当前时间最大的数据。实现：1）数据第一次被访问，加入到访问历史列表；2）如果数据在访问历史列表里后没有达到K次访问，则按照一定规则（FIFO，LRU）淘汰；3）当访问历史队列中的数据访问次数达到K次后，将数据索引从历史队列删除，将数据移到缓存队列中，并缓存此数据，缓存队列重新按照时间排序；4）缓存数据队列中被再次访问后，重新排序；5）需要淘汰数据时，淘汰缓存队列中排在末尾的数据，即：淘汰“倒数第K次访问离现在最久”的数据。针对问题：LRU-K的主要目的是为了解决LRU算法“缓存污染”的问题，其核心思想是将“最近使用过1次”的判断标准扩展为“最近使用过K次”。

* + 1. 请问线程需要保存哪些上下文，SP、PC、EAX这些寄存器是干嘛用的线程在切换的过程中需要保存当前线程Id、线程状态、堆栈、寄存器状态等信息。其中寄存器主要包括SP PC EAX等寄存器，其主要功能如下：SP:堆栈指针，指向当前栈的栈顶地址PC:程序计数器，存储下一条将要执行的指令EAX:累加寄存器，用于加法乘法的缺省寄存器
    2. 守护进程、孤儿进程、僵尸进程  
       僵尸进程：子进程退出后，父进程没有调用wait函数等待，子进程虽然释放了资源，但是还有一部分资源等待父进程来释放，子进程的进程号一直会被占用。僵尸进程kill是无法杀除的，一般是通过找到僵尸进程父进程id并kill  
       孤儿进程： 父进程退出，他的子进程还在运行，子进程将成为孤儿进程，将被init进程1管理。  
       守护进程：fork(父进程终止)->setsid(子进程成为会话组长和进程组长，脱离终端控制)->忽略signal信号->在fork(由于子子进程不再是会话组长，从而禁止进程重新打开控制终端)->chdir(改变当前工作目录为根目录)->关闭打开的文件描述符->重设文件创建掩码，清楚父进程继承的文件掩码
    3. 请你回答一下为什么要有page cache，操作系统怎么设计的page cache加快从磁盘读取文件的速率。page cache中有一部分磁盘文件的缓存，因为从磁盘中读取文件比较慢，所以读取文件先去page cache中去查找，如果命中，则不需要去磁盘中读取，大大加快读取速度。在 Linux 内核中，文件的每个数据块最多只能对应一个 Page Cache 项，它通过两个数据结构来管理这些 Cache项，一个是radix tree，另一个是双向链表。Radix tree 是一种搜索树，Linux内核利用这个数据结构来通过文件内偏移快速定位Cache 项
    4. 栈效率高的原因  
       栈是操作系统提供的数据结构，计算机底层对栈提供了一系列支持：分配专门的寄存器存储栈的地址，压栈和入栈有专门的指令执行；而堆是由C/C++函数库提供的，机制复杂，需要一些列分配内存、合并内存和释放内存的算法，因此效率较低

## 性能调优

Strace –p pstack查看进程函数堆栈 pstree以树状图显示进程间的关系 lsof valgrind perf  
磁盘io  
iostat、vmstat、iotop  
cpu  
top、perf、demsg、vmstat

## 负载均衡算法 七层负载：一般就是通过url或者主机名 四层负载: 基于ip+port进行负载

1 轮询、哈希、最小连接数、随机、加权轮询、加权随机

2加权轮询算法实现

## rpc微服务问题 1 服务熔断和服务降级 熔断就是过载保护

## 进程、线程、协程区别

<https://blog.csdn.net/daaikuaichuan/article/details/82951084>  
进程是资源分配的最小单位  
是操作系统调度（CPU调度）执行的最小单位。

## Redis的hash实现 小于一定量使用压缩表，使用数组加链表存储的。Hash函数算出来的大部分都是数字，作为下标。 所有有两个hash表。rehash使用渐进式的扩容方式。查询时，在把相应的keycopy进去，写的时候，重新分配。

## Redis的zset实现

***typedef******struct******zset*** *{*

*dict* ***\*****dict;*

*zskiplist* ***\*****zsl;*

*} zset;*

有哈希表和跳表、压缩表组成

跳表底部是双向链表，顶部是单向有序的链表，所以查找如果当前节点大于目标，保证范围查找  
(1) 由很多层结构组成(2) 每一层都是一个有序的链表(3) 最底层(Level 1)的链表包含所有元素(4) 如果一个元素出现在 Level i 的链表中，则它在 Level i 之下的链表也都会出现。(5) 每个节点包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素。

跳表，红黑树、avl、B+tree区别  
https://blog.csdn.net/congce9794/article/details/100304508?utm\_medium=distribute.pc\_relevant\_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel\_param&depth\_1-utm\_source=distribute.pc\_relevant\_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel\_param  
跳表：  
(1). 跳跃表的每一层都是一条有序的链表.(2). 跳跃表的查找次数近似于层数，时间复杂度为O(logn)，插入、删除也为 O(logn)。(3). 最底层是双向，链表包含所有元素。(4). 跳跃表是一种随机化的数据结构(通过抛硬币来决定层数)。(5) 支持区间查询  
  
AVL树：  
具有二叉查找树的特点(左子树任一节点小于父节点，右子树任一节点大于父节点)，任何一个节点的左子树与右子树都是平衡二叉树任一节点的左右子树高度差小于1，即平衡因子为范围为[-1,1] 如上左图根节点平衡因子=1，为AVL树；右图根节点平衡因子=2，固非AVL树，只是BST。  
  
红黑树：  
每个节点只能是红色或黑色根节点总是黑色的红色节点的父或子节点都必然是黑色的(两个红色的节点不会相连)任一节点到其所有后代NULL节点的每条路径都具有相同数量的黑色节点每个Null节点都是黑色的  
b+树

m 叉树只存储索引，并不真正存储数据，只有最后一行的叶子节点存储行数据  
非叶子节点只存key，大大减少了非叶子的大小，这样每个节点就可以存放更多的记录，树更矮了，io操作更少。叶子节点都是相链的。对整棵树值需要一次遍历即可。数据顺序排列且相连，利于区间查找和收索。B树没有。

## Redis的过期策略和内存淘汰策略

定时过期：创建定时器，到过期时间就删除，cpu消耗大  
惰性过期：只有访问key的时候，才会判断是否删除

定期过期：每隔一段时间检测

**内存淘汰策略：**

* noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。
* allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。
* allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。
* volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。
* volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。
* volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。

## redis分布式锁有什么缺陷

是因为都分开来使用了，如果直接使用setnx ex去设置，就没有问题。我们项目都是这样用的。

## redis如何做内存优化 <https://baijiahao.baidu.com/s?id=1654440869302170492&wfr=spider&for=pc> 详情：https://cloud.tencent.com/developer/article/1162213

## cas概念、原子类实现原理

无锁算法，无所编程在不适用锁的情况下实现多线程之间的变量同步，也叫非阻塞同步，当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

## 多线程问题

* + 1. 生产者消费者问题编写
    2. 多线程同步和互斥问题

c++多线程四种方式：互斥锁、条件变量、读写锁、信号量

互斥锁  
1. **原子性**：把一个互斥量锁定为一个原子操作，这意味着操作系统（或pthread函数库）保证了如果一个线程锁定了一个互斥量，没有其他线程在同一时间可以成功锁定这个互斥量；

2. **唯一性**：如果一个线程锁定了一个互斥量，在它解除锁定之前，没有其他线程可以锁定这个互斥量；

3. **非繁忙等待**：如果一个线程已经锁定了一个互斥量，第二个线程又试图去锁定这个互斥量，则第二个线程将被挂起（不占用任何cpu资源），直到第一个线程解除对这个互斥量的锁定为止，第二个线程则被唤醒并继续执行，同时锁定这个互斥量。

互斥量不是万能的，比如某个线程正在等待共享数据内某个条件出现，可可能需要重复对数据 对象加锁和解锁（轮询），但是这样轮询非常耗费时间和资源，而且效率非常低，所以互斥锁 不太适合这种情况

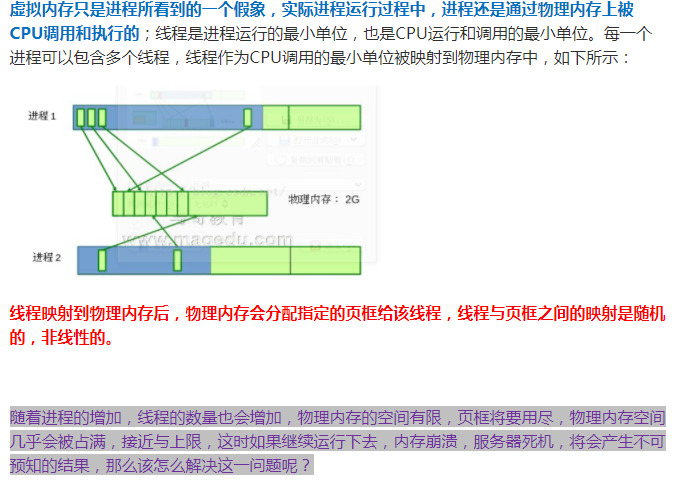
我们需要这样一种方法：当线程在等待满足某些条件时使线程进入睡眠状态，一旦条件满足， 就换线因等待满足特定条件而睡眠的线程

总结：  
1 互斥锁也会等待，但是条件变量里头参数，是需要互斥锁的，所以条件变量一定跟互斥锁搭配着用。

2 互斥锁在pthread\_cond\_wait的时候, wait前要加锁，wait里会把锁释放了，wait出来后又会加锁，所以后续还是需要mutex\_unlock。  
分析上面的java代码：在调用mutex.wait()之前的若干步我们锁住了mutex；调用mutex.wait()让出CPU的使用权，并且释放了锁；调用mutex.notify()之前先获得mutex上的锁，调用notify之后退出了synchronized块，即立刻释放了锁；这时wait()被唤醒，它首先获得mutex上的锁，再执行下面的语句。  
https://www.jianshu.com/p/58a3ab060f89

## 操作系统

* + - 1. 进程调度算法、那些事抢占式的  
         1先来先服务和短作业进程有限调度算法  
         2 优先权、高响应比  
         3 基于时间片的轮询
      2. 虚拟内存和交换空间机制:  
         安装Linux时需要两个分区，一个是根目录，另外一个就是swap(内存交换空间)
      3. 虚拟内存在于不同进程看到自己独占系统的4g内存，3g的用户进程和1g的内核。当然内核会将当前进程所暂用的内存转换为实际的物理内存。



这时候就需要交换空间。内核会在硬盘上一段存取速度较高的存储空间，把他做完交换空间，当实际的物理内存被占满时，内核会先将最近一段时间使用最不频繁的线程，取消和物理内存的映射关系，并且移到swap分区。 当这个线程再次被唤醒的时候，挪出最近使用不频繁的线程到swap分区。 并且将刚才的线程与实际的内存做好映射关系，获取系统内存的资源

## 父进程创建了子进程，它们所指的空间地址一样吗 子进程“继承”父进程的变量，其地址总是一样的，因为在fork时整个虚拟地址空间被复制，但是虚拟地址空间所对应的物理内存却没有复制（这个时候父子进 程中变量 x对应的虚拟地址和物理地址都相同）。等到虚拟地址空间被写时，对应的物理内存空间被复制（这个时候父子进程中变量 i 对应的虚拟地址还是相同的，但是物理地址不同），这就是"写时复制"。一样的是线性地址。

## 大小端、字节序问题 bind和conn里头，需要注意字节序问题，ip和port

## Redis热key问题处理

1 对业务的熟悉，凭借经验，进行评估  
2 在客户端进行收集  
3 在redisproxy层做收集，一般都是这样

4 Redis自带命令  
5 自己抓包评估， 通信是RESP协议，解析维护  
压垮db  
先更新数据库，在更新缓存•      当更新缓存成功，更新数据库失败时，数据库中就是旧数据先更新数据库，再更新缓存•      1 在多线程并发时，会出现不一致行为。假设请求 A 先操作数据库，请求 B 后操作数据库，但是可能存在请求 B 先写缓存，请求 A 后写缓存的情况，从而导致数据库与缓存之间的数据不一致先删缓存， 再写数据库•      请求a先删除缓存，请求b获取旧的数据库的数据，请求a存数据到db，请求b将旧值写到缓存先写数据库，再删除缓存1 a查询旧值，b更新数据库，b更新新数据缓存，a在更新旧数据缓存2 主从读写问题。3 缓存删除失败问题  
缓存穿透: 查询数据库和缓存不存在的key•      bloom filter

缓存击穿：热key缓存失效，大量的请求到达系统  
解决：  
1 热点数据进行二级缓存，不同级别的缓存设置不同的过期时间，2 缓存延迟失效，回源中数据

缓存雪崩1 redis挂了2 redis设置的key，同时过期了  
解决：1 redis高可用哨兵+集群模式2 设置随机的key3 本地二级缓存+限流4 redis持久化，快速恢复服务  
如何解决  
1 分散热key，把key存储是按照某些维度hash处理，打散2 分散热key，把key存储是按照某些维度hash处理，打散  
2 利用二级缓存

如何保证数据一致性问题

一致性hash问题

长度为2^32的整数环,将节点名称的hash值分布到这个环上,(一般是ip地址),之后存放数据时，根据数据的key的hash值，在环上顺时针寻找一个大于或者等于他的节点，如果没有找到，就用node0节点作为存放。虚拟节点会和真实节点做映射关系，

## C++11新特性

[1、关键字及新语法](https://www.cnblogs.com/chengjundu/p/10893702.html#title1)

[1.1、auto关键字及用法](https://www.cnblogs.com/chengjundu/p/10893702.html#title11)

[1.2、nullptr关键字及用法](https://www.cnblogs.com/chengjundu/p/10893702.html#title12)

[1.3、for循环语法](https://www.cnblogs.com/chengjundu/p/10893702.html#title13)

[2、STL容器](https://www.cnblogs.com/chengjundu/p/10893702.html#title2)

[2.1、std::array](https://www.cnblogs.com/chengjundu/p/10893702.html#title21)

[2.2、std::forward\_list](https://www.cnblogs.com/chengjundu/p/10893702.html#title22)

[2.3、std::unordered\_map](https://www.cnblogs.com/chengjundu/p/10893702.html#title23)

[2.4、std::unordered\_set](https://www.cnblogs.com/chengjundu/p/10893702.html#title24)

[3、多线程](https://www.cnblogs.com/chengjundu/p/10893702.html#title3)

[3.1、std::thread](https://www.cnblogs.com/chengjundu/p/10893702.html#title31)

[3.2、st::atomic](https://www.cnblogs.com/chengjundu/p/10893702.html#title32)

[3.3、std::condition\_variable](https://www.cnblogs.com/chengjundu/p/10893702.html#title33)

[4、智能指针内存管理](https://www.cnblogs.com/chengjundu/p/10893702.html#title4)  
智能指针是一个类，这个类的构造函数中传入一个普通指针，析构函数中释放传入的指针。智能指针的类都是栈上的对象，所以当函数（或程序）结束时会自动被释放

[4.1、std::shared\_ptr](https://www.cnblogs.com/chengjundu/p/10893702.html#title41) 引用计数、同时存在循环引用的问题，用下面weak\_ptr解决

[4.2、std::weak\_ptr](https://www.cnblogs.com/chengjundu/p/10893702.html#title42)

4.3、auto\_ptr 两个智能指针不能赋值

4.4、unique\_ptr   
5、std::move()和std::forward()区别  
左值转右值作用

因为右值引用属性不能被转发，所以需要forward达到完美转发。

[5、其他](https://www.cnblogs.com/chengjundu/p/10893702.html#title5)

[5.1、std::function、std::bind封装可执行对象](https://www.cnblogs.com/chengjundu/p/10893702.html#title51)

Fundction可以看做是函数指针、对象等都可以，是一个类模板

1. 普通函数
2. lambda表达式
3. 函数指针
4. 仿函数(functor 重载括号运算符实现)  
   仿函数(functor)，就是使一个类的使用看上去像一个函数。其实现就是类中实现一个operator()，这个类就有了类似函数的行为，就是一个仿函数类了
5. 类成员函数
6. 静态成员函数

std::bind将可调用对象与其参数一起进行绑定，绑定后的结果可以使用std::function保存。std::bind主要有以下两个作用：

1 将可调用对象和其参数绑定成一个防函数()；

2 只绑定部分参数，减少可调用对象传入的参数。

[5.2、lamda表达式](https://www.cnblogs.com/chengjundu/p/10893702.html#title52)

**6、虚函数的override和final指示符**

## redis集群问题(https://juejin.im/post/6844903901599105037)\_

* + 1. redis如何选主。  
       常用的Gossip消息可分为：ping消息、pong消息、meet消息、fail消息  
       选举使用的是raft协议  
       1 接收ping消息的节点，没有在规定时间内，回复pong消息，那么发送ping节点会会将接收ping节点标记为（pfail）  
       2 半数以上的节点都标记为疑似下线，那么这个主节点将被标记为已下线(fail)，并向集群广播。所以主节点的数量一般为奇数，没有的情况，可能会脑裂  
       3 当一个节点发现自己的主节点进入fail状态，将会从这个节点的从节点当中，选出一台，执行slaveof no one命令，变身为主节点。新节点完成槽指派后，并集群广播，告知其他节点。  
       4 每个集群，都会维护一个路由表，进行通讯
    2. redis集群如何持久化  
       1 master关闭持久化  
       2 slave开启rdb同步，必要的时候AOF和RDB都开启  
       问题：  
       redis的主从复制是异步的，master执行完客户端请求的命令后会立即返回结果给客户端，然后异步的方式把命令同步给slave。因此master可能还未来得及将命令传输给slave，就宕机了，此时slave变为master，数据就丢了。
    3. redis主从同步  
       1 节点通过ping-pong机制互联，  
       2 客服端发起请求，接收到moved\asked错误，会自动重定向到其他节点。3 cluster节点采用异步复制，
    4. redis线上使用注意  
       1 数据做好冷热分离，低频数据可存储在 Mysql、 ElasticSearch中  
       2 业务数据分离。多set等  
       3 消息大小限制，大key问题  
       4 链接限制，做好连接池等。  
       5 缓冲key做好过期时间  
       6 严禁使用keys，flush，消息队列，不设置范围的批量操作，(zset,hash,mget)  
       7 禁用lua，事物、moniter  
       8 大流量，不要开aof，开rdb即可
    5. 常见问题

问题1:懂Redis事务么？  
  
高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key是有可能分配在不同的Redis节点上的，在这种情况下Redis的事务机制是不生效的。其次，Redis事务不支持回滚操作，简直是鸡肋！所以基本不用！

问题2:Redis的多数据库机制，了解多少？  
正常版：Redis支持多个数据库，并且每个数据库的数据是隔离的不能共享，单机下的redis可以支持16个数据库（db0 ~ db15）  
高调版: 在Redis Cluster集群架构下只有一个数据库空间，即db0。因此，我们没有使用Redis的多数据库功能！

问题3:Redis集群机制中，你觉得有什么不足的地方吗？  
高调版: 假设我有一个key，对应的value是Hash类型的。如果Hash对象非常大，是不支持映射到不同节点的！只能映射到集群中的一个节点上！还有就是做批量操作比较麻烦！

问题4:懂Redis的批量操作么？  
正常版: 懂一点。比如mset、mget操作等，blabla  
高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key会划分到不同的slot中，因此直接使用mset或者mget等操作是行不通的。

问题5:那在Redis集群模式下，如何进行批量操作？  
高调版:这个问题其实可以写一篇文章了，改天写。这里说一种有一个很简单的答法，足够面试用。即:  
如果执行的key数量比较少，就不用mget了，就用串行get操作。如果真的需要执行的key很多，就使用Hashtag保证这些key映射到同一台redis节点上。简单来说语法如下

**对于key为{foo}.student1、{foo}.student2，{foo}student3，这类key一定是在同一个redis节点上。因为key中“{}”之间的字符串就是当前key的hash tags， 只有key中{ }中的部分才被用来做hash，因此计算出来的redis节点一定是同一个!**

ps:如果你用的是Proxy分片集群架构，例如Codis这种，会将mget/mset的多个key拆分成多个命令发往不同得redis实例，这里不多说。我推荐答的还是redis cluster。

问题6:你们有对Redis做读写分离么？  
高调版:不做读写分离。我们用的是Redis Cluster的架构，是属于分片集群的架构。而redis本身在内存上操作，不会涉及IO吞吐，即使读写分离也不会提升太多性能，Redis在生产上的主要问题是考虑容量，单机最多10-20G，key太多降低redis性能.因此采用分片集群结构，已经能保证了我们的性能。其次，用上了读写分离后，还要考虑主从一致性，主从延迟等问题，徒增业务复杂度。

* + 1. 数据迁移问题(扩缩容)  
       redis-08 搞定 redis 所有面试(非常好写的) <https://www.cnblogs.com/liang1101/archive/2004/01/13/12920590.html>
    2. 数据迁移问题(扩缩容)

1 数据迁移状态：MIGRATING状态、IMPORTING状态，当槽x从Node A向Node B迁移时，Node A和Node B都会有这个槽x，Node A上槽x的状态设置为MIGRATING，Node B上槽x的状态被设置为IMPORTING。

MIGRATING状态

1)如果key存在则成功处理

2)如果key不存在，则返回客户端ASK，客户端根据ASK首先发送ASKING命令到目标节点，然后发送请求的命令到目标节点

3)当key包含多个命令，

    a)如果都存在则成功处理

    b)如果都不存在，则返回客户端ASK

    c)如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的        key都迁移完毕的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就          可以获取这批键

4)此时不刷新客户端中node的映射关系

IMPORTING状态

1)如果key不在该节点上，会被MOVED重定向，刷新客户端中node的映射关系

2)如果是ASKING命令则命令会被执行，key不在迁移的节点已经被迁移到目标的节点

3)Key不存在则新建

请求重定向

a)MOVED错误

1.请求的key对应的槽不在该节点上，节点将查看自身内部所保存的哈希槽到节点ID的映射记录，节点回复一个MOVED错误。

2.需要客户端进行再次重试。

b)ASK错误

1.请求的key对应的槽目前的状态属于MIGRATING状态，并且当前节点找不到这个key了，节点回复ASK错误。ASK会把对应槽的IMPORTING节点返回给你，告诉你去IMPORTING的节点尝试找找。

2.客户端进行重试首先发送ASKING命令，节点将为客户端设置一个一次性的标志（flag），使得客户端可以执行一次针对IMPORTING状态的槽的命令请求，然后再发送真正的命令请求。

3.不必更新客户端所记录的槽至节点的映射。

链接：https://www.jianshu.com/p/614c5e2e6ba1

* + - 1. Redis扩容和一致性hash

槽计算使用CRC16(key) % 16384

## c++异常实现机制

## linux常用命令

top、w、uptime中，第一行，表示(1、10、15分钟)平均负载，正常在0-1区间，多核不能高于cpu核总数

## 在浏览器输入url

url解析  
dns查询:浏览器缓存-操作系统缓存-路由器缓存-dns缓存-根域名服务器  
tcp连接：应用层(http数据)->传输层(tcp首部+http)->网络层(ip首部+tcp首部+http)->数据链路层(以太网首部+ip首部+tcp首部+http ===mac地址，网卡)

## 输入 ping IP 后敲回车，发包前会发生什么

首先根据目的IP和路由表决定走哪个网卡，再根据网卡的子网掩码地址判断目的IP是否在子网内。如果不在则会通过arp缓存查询IP的网卡地址，不存在的话会通过广播询问目的IP的mac地址，得到后就开始发包了，同时mac地址也会被arp缓存起来。

## 原子指令和锁有什么区别

互斥锁使用的是链表维护、自旋锁while一直获取。  
原子操作：不会被线程的调度机制所打断。Inter指令集有个lock，inter会让总线锁住，当这个核把指令执行完了，在开启总线

## 优化锁性能有哪些原则

## 用户态和内核态的切换

1 用户态调用系统函数，软中断  
2 异常  
3 外围设备中断， 硬中断

## C++volatile作用

1 并发编程中三个问题：原子性，可见性，有序性  
2 cpu工作可以分为5个阶段：取指令、指令译码、执行指令、访存取数、结果写回。  
3 在内存中取数，回写与cpu的执行指令速度相比差距大，因此cpu中就有了高速缓存。 所以cpu在执行指令的时候，会将主存的数据，复制到高速缓存，这时候多线程，每个线程是不知道高速缓存中的值得。当运算完毕后，将结果刷新到主存中。这样就存在缓存不一致的问题。并且这也是数据可见性的问题。解决办法就是

（1）通过总线加LOCK锁的方式（2）通过缓存一致协议  
4 有序性问题。处理器为了提供程序运行效率，会对代码进行优化，  
volatile会保证代码不可优化，顺序性，并且会保证去内存中取，而不是从高速缓存中取

## 什么是死锁，什么是僵尸进程，守护进程

## 布隆过滤器，10亿数据问题

10亿int数据，40亿字节(暂用4G)、

位图法：  
一个int整型数占4字节（Byte），也就是32位（bit）。那么把所有int整型数字表示出来需要2^32 bit的空间，换算成字节单位也就是2^32/8 = 2^29 Byte，大约等于512MB  
如何确定数字放在32个位中的哪个位：将数字mod32取模(x%32)。上例中我们如何确定8在tmp[0]中的32个位中的哪个位，这种情况直接mod上32就ok，又如整数8，在tmp[0]中的第8 mod上32等于8，那么整数8就在tmp[0]中的第八个bit位（从右边数起）。

布隆过滤器：  
核心实现是一个超大的位数组和几个哈希函数，  
1 将要查询的元素给k个哈希函数2 得到对应于位数组上的k个位置3 如果k个位置有一个为0，则肯定不在集合中4 如果k个位置全部为1，则可能在集合中

## 4次拷贝

CPU发指令给I/O设备的DMA，由DMA将我们磁盘中的数据传输到内核空间的内核buffer。第二阶段触发我们的CPU中断，CPU开始将将数据从kernel buffer拷贝至我们的应用缓存CPU将数据从应用缓存拷贝到内核中的socket buffer.DMA将数据从socket buffer中的数据拷贝到网卡缓存。总共需要经历四个阶段，2次DMA，2次CPU中断，总共四次拷贝，有四次上下文切换，并且会占用两次CPU。

## Protobuf是如何实现协议兼容性的

## 服务器秒杀，超卖等诸多问题

https://blog.csdn.net/aochijing0046/article/details/101493618

## 如何设计一个高性能网络服务器

1 合理的评估和分配服务资源  
2 避免不必要的浪费  
3 在延迟和吞吐量上衡量  
4 负载问题

## Fork和epoll带来的问题

1 流程是，父进程create->子进程epoll\_ctl add->父进程wait  
 fork之前创建epoll\_create，尽管父子进程拥有自己的空间，但是底层引用的同一个内核结构，如果是父进程epoll\_wait，那么父子进程都会收到epoll触发的事件。 父进程调用accept失败，打印出Bad file descriptor错误，是因为在父进程中，根本没有监听套接字。所以，只要子进程没有调用accept成功，则该连接事件就会一直触发，从而父进程一直打印accept错误信息，直到子进程调用accept成功，打印出connect from  
2 在linux2.6内核上，accept系统调用已经不存在惊群了（至少我在2.6.18内核版本上已经不存在）。大家可以写个简单的程序试下，在父进程中bind,listen，然后fork出子进程，所有的子进程都accept这个监听句柄。这样，当新连接过来时，大家会发现，仅有一个子进程返回新建的连接，其他子进程继续休眠在accept调用上，没有被唤醒。但是很不幸，通常我们的程序没那么简单，不会愿意阻塞在accept调用上，我们还有许多其他网络读写事件要处理，linux下我们爱用epoll解决非阻塞socket。所以，即使accept调用没有惊群了，我们也还得处理惊群这事，因为epoll有这问题。上面说的测试程序，如果我们在子进程内不是阻塞调用accept，而是用epoll\_wait，就会发现，新连接过来时，多个子进程都会在epoll\_wait后被唤醒！  
多进程服务器程序都是在fork()之后，再对epoll\_wait(listen\_fd,...)的事件，这样子当listen\_fd有新的accept请求时，进程们还是会被唤醒。论文的改进主要是在内核级别让accept()成为原子操作，避免被多个进程都调用了  
accept后fork，因为listend\_fd fork时候是引用计数加1，所以需要父进程中关闭accept的描述符，并且在fork子进程中关闭listen的描述符。 fork后，accept，子进程中关闭listen描述符，父进程中不做处理。

## Qps计算优化问题

https://blog.csdn.net/weixin\_34346099/article/details/88679411?utm\_medium=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel\_param&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel\_param

## 系统设计思考

1 如何设计实现中国象棋？2 如何设计一个排队系统？

3 假设客户端有一个大文件与服务端同步。已知文件中某处被修改，请设计一个算法更新被修改的部分

4 敏感词过滤

## Topk问题

求最大前10000个数：

先拿10000个数建堆，然后一次添加剩余元素，如果大于堆顶的数（10000中最小的），将这个数替换堆顶，并调整结构使之仍然是一个最小堆，这样，遍历完后，堆中的10000个数就是所需的最大的10000个。建堆时间复杂度是O（mlogm），算法的时间复杂度为O（nmlogm）（n为10亿，m为10000）。       优化的方法：可以把所有10亿个数据分组存放，比如分别放在1000个文件中。这样处理就可以分别在每个文件的10^6个数据中找出最大的10000个数，合并到一起在再找出最终的结果。

## 开放性问题

1 代码常用的检测方法

## Cpu使用率低负载高，原因

什么是负载：负载就是cpu在一段时间内正在处理以及等待cpu处理的进程数之和的统计信息，也就是cpu使用队列的长度统计信息，这个数字越小越好（如果超过CPU核心\*0.7就是不正常）负载分为两大部分：CPU负载、IO负载例如，假设有一个进行大规模科学计算的程序，虽然该程序不会频繁地从磁盘输入输出，但是处理完成需要相当长的时间。因为该程序主要被用来做计算、逻辑判断等处理，所以程序的处理速度主要依赖于cpu的计算速度。此类cpu负载的程序称为“计算密集型程序”。还有一类程序，主要从磁盘保存的大量数据中搜索找出任意文件。这个搜索程序的处理速度并不依赖于cpu，而是依赖于磁盘的读取速度，也就是输入输出（input/output,I/O）.磁盘越快，检索花费的时间就越短。此类I/O负载的程序，称为“I/O密集型程序”。

进程等待：  
1 等待分配cpu资源的状态。2 等待磁盘输入输出完毕的状态。

场景：  
场景一：磁盘读写请求过多就会导致大量I/O等待场景二：MySQL中存在没有索引的语句或存在死锁等情况场景三：外接硬盘故障，常见有挂了NFS，但是NFS server故障

## Cpu负载高如何定位

Cpu低，负载高，可能磁盘不行

先top、ps看完后，然后定位看 strace pstack看某个进程的当前线程栈运行情况  
strace 用法  
strace -p 17553  
strace -tt -T -v -f -e trace=file -o /data/log/strace.log -s 1024 -p 23489  
<https://www.cnblogs.com/machangwei-8/p/10388883.html>

## Cpu密集型和io密集型

创建多少个线程合适  
1 单核情况下，创建4个线程，和一个线程情况是一样的，但是还要考虑四个线程上下午切换等问题  
2 如果是多核，那起4个线程，效率总体提升4倍  
3