操作系统

        - Spring是如何解决循环依赖的?
        - 为什么要使用三级缓存呢？二级缓存能解决循环依赖吗?
    - 什么是AOP
    - AOP的相关注解有哪些
    - AOP的相关术语
    - AOP的过程
  - 3. Spring中JDK动态代理通过JdkDynamicAopProxy调用Proxy的newInstance方法来生成代理类，JdkDynamicAopProxy也实现了InvocationHandler接口，invoke方法的具体逻辑是先获取应用到此方法上的执行器链，如果有执行器则创建MethodInvocation并调用proceed方法，否则直接反射调用目标方法。因此Spring AOP对目标对象的增强是通过拦截器实现的。
    - 什么是事务?
    - Spring支持两种方式的事务管理
    - PlatFormTransactionManager
    - TransactionDefinition
    - TransactionStatus
    - @Tranactional注解
    - Spring框架中用到了哪些设计模式?
    - Spring涉及到的几种设计模式
    - MyBatis
    - #{} 和 ${} 的区别
    - 一级缓存是什么
    - 二级缓存是什么
    - SpringMVC
    - SpringMVC的处理流程
    - DispatcherServlet的作用
    - DispatcherServlet初始化顺序
    - ContextLoaderListener初始化的上下文和DispatcherServlet初始化的上下的关系
    - SpringMVC有哪些组件
    - SpringMVC有哪些注解
    - 处理器拦截器
    - SpringBoot
    - Springboot的优点
    - Springboot的自动配置原理
    - 什么是CSRF攻击
    - 什么是WebSockets
    - SpringBoot达成的jar和普通的jar有什么区别?
- Dubbo
  - Dubbo的相关介绍
  - 服务暴露
  - 服务引用
  - 服务调用
  - SPI
    - 为什么Dubbo不用java的SPI，而要自己实现?
    - Adaptive注解：自适应扩展
  - 如何设计一个RPC
- 分布式

概述

什么是操作系统

1. 操作系统是管理计算机硬件和软件程序的程序，是计算机的基石。
2. 操作系统本质上是一个运行在计算机上的软件程序，用于管理计算机硬件和软件资源。
3. 操作系统的存在屏蔽了硬件层的复杂性。
4. 操作系统的内核是操作系统的核心部分，它负责系统的内存管理，设备管理，文件系统管理和应用程序的管理。

操作系统的基本功能

1. 进程管理：进程控制、进程同步、进程通信、死锁处理、处理机调度等
2. 内存管理：内存分配、地址映射、内存保护与共享、虚拟内存等
3. 设备管理：文件存储空间的管理、目录管理、文件读写管理和保护等
4. 文件管理：完成用户I/O请求，方便用户使用各种设备，并提高设备的利用率。主要包括：缓冲管理、设备分配、设备处理、虚拟设备等。

操作系统的特点

- 共享：操作系统中的资源可以被多个并发进程使用。互斥共享和同时共享。
- 虚拟：虚拟技术把一个物理实体转换成多个逻辑实体。
- 并发：在一段时间内同时运行多个程序。
- 异步：进程不是一次性执行完毕，而是走走停停，以不可知的速度向前推进。

用户态切换到内核态的三种方式

- 中断：当外围设备完成用户请求后，会向CPU发出相应的中断信号，这时CPU会暂停执行当前进程的指令，转而执行与中断信号对应的处理程序。
- 异常：当CPU执行运行在用户态下的程序时，发生了某些实现不可知的异常，这时会触发由当前运行的进程切换到处理此异常的内核的相关的程序中，比如缺页异常。
- 系统调用：进程通信，进程控制，文件操作，设备操作。

什么是系统调用

- 用户态和内核态
- 用户态可以直接读取应用程序中的数据。内核态：操作系统运行的进程或程序几乎可以访问系统的任何资源。
- 当用户态的应用程序想要调用内核态的子功能时需要用到系统调用。也就是说应用程序凡是需要用到与内核态级别的资源有关的操作，都需要通过系统调用向操作系统发出服务请求，由操作系统代为完成。
- 系统调用：进程控制、进程通信、文件操作、设备操作等。

1. 用户态把一些数据放到寄存器，或者创建对应的堆栈，表明需要操作系统提供的服务。
2. 用户态执行系统调用（系统调用是操作系统的最小功能单位）。
3. CPU切换到内核态，跳到对应的内存指定的位置执行指令。
4. 系统调用处理器去读取我们先前放到内存的数据参数，执行程序的请求。
5. 调用完成，操作系统重置CPU为用户态返回结果，并执行下个指令。

linux进程内存空间分为哪几个段

1. Text，代码区：存放可执行的指令操作，只读不可写。
2. Bss，静态区/全局区：存在未初始化的全局变量和静态变量。
3. Data，数据区：存在初始化的全局变量和静态变量。
4. Stack：存放临时变量和函数参数。
5. Heap：存放new/malloc等动态申请的变量，用户必须手动进行delete/free操作。 其中Stack和Heap的内存增长方向是相反的。

进程

进程和线程的区别

- 进程是资源分配的基本单位。线程不拥有资源，但是线程可以访问隶属进程的资源
- 线程是独立调度的基本单位。在同一个进程内的线程之间切换不会引起进程切换，但是从一个进程中的线程切换到另一个进程中的线程时，会引起进程的切换。
- 线程的开销比进程的开销要小。进程的创建或撤销需要分配或者回收资源，如内存空间、I/O设备等，所付出的开销远大于创建或撤销时的开销。
- 线程间可以通过直接读写同一进程内的数据进行通信。

同一进程中不同线程共享哪些资源?

- 堆
- 静态变量
- 全局变量
- 文件等公共资源 线程共享的环境包括：进程的代码段(方法区)、进程打开的文件描述符、进程的公共数据(方法区的运行时常量池)、信号的处理器、进程当前目录和进程用户

进程有哪几种状态

- 创建状态
- 就绪状态
- 运行状态
- 阻塞状态
- 结束状态

进程间通信的方式

- 匿名管道/管道：创建一个匿名管道，返回两个文件描述符，一个读取端文件描述符fd[0]，一个写入端文件描述符fd[1]。匿名管道是特殊的文件，只存在于内存，不存在于文件系统。所谓管道是内核中的一段缓存，从管道的一端写入的数据，是存在内核中的，另一端读取也就是在内核中读取数据。管道的传输的数据是无格式的且大小受限的。fork一个子进程，子进程会复制父进程的文件描述符，`ps -ef | grep java`
- 有名管道 **管道的数据传输是单向的，**
- 共享内存：拿出一块虚拟地址空间表，映射到相同的物理内存上。
- 消息队列：管道的传输效率是很低的，不适合频繁的进程间的交换数据。消息队列是存在在内核中的消息链表，在传输数据的时候会分成一个个的消息体，传输的消息体是用户自定义的数据类型，在传输开始前两个进程约定好数据类型，不像管道只能传输无格式的字节数据。消息队列会随着内核，如果没有释放消息队列或者没有关闭操作系统，消息队列是一直存在的。不像管道是随着进程的创建而创建，销毁而销毁。消息队列是双向的。消息队列不适合比较大数据的传输，每个消息体有一个最大长度的限制，同时所有队列包含的全部消息体的总长度也有上限。消息队列数据传输过程中，存在用户态和内核态之间的数据拷贝。当进程写入到进程的消息队列中时，会将用户态的数据拷贝到内核态中。当进程读取内核态中的消息队列时，会发生内核态拷贝数据到用户态的过程。
- 信号：信号是进程间通信唯一的异步通信机制。任何时候发给信号给某一个进程，一旦有信号产生，就会执行相应的信号处理方式。
- 信号量：当多个进程访问一块共享资源时，会发生数据错乱，所以需要保护机制。信号量实际是一个整形的计数器，主要用于实现进程间的互斥和同步，不用于缓存进程间通信的数据。
- 套接字

线程间同步的方式

- 互斥量：只允许一个线程访问当前资源。
- 信号量：允许多个线程访问同一资源，但是有数量限制。
- 事件：wait/notify
- 临界区

进程的调度算法

- 先到先服务调度算法

- 优先级调度算法
- 时间片轮转调度算法
- 短作业优先调度算法
- 多级反馈队列调度算法

孤儿进程

一个父进程退出，一个或多个子进程还在运行，这些进程被称为孤儿进程。 孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。由于孤儿进程会被init进程所收养，所以孤儿进程不会对系统造成危害。

僵尸进程

一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过wait()或者waitpid()获取了子进程的信息后才会释放。如果子进程退出，父进程并没有调用wait()或者waitpid()，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵尸进程。 系统所能使用的进程号是有限的，如果产生大量僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。 要消灭系统中大量的僵尸进程，只需要将父进程杀死，这些子进程就会变成孤儿进程，被init进程收养，这样init进程就会释放所有的僵尸进程所占有的资源，从而结束僵尸进程。

经典同步问题

1. 哲学家进餐问题
2. 读者-写者问题

死锁

死锁的必要条件

1. 互斥：每个资源要么分配给一个进程，要么就是可用的
2. 占有且等待：已经得到了某个资源的进程可以再请求新的资源
3. 不可抢占：已经分配给一个进程的资源不能强制性的被抢占，它只能被占有它的进程显示的释放
4. 循环等待：有两个或者两个以上的进程组成一条环路，该环路中每个进程都在等到下一个进程占有的资源。

死锁的处理方法

1. 鸵鸟策略
2. 死锁的检测和恢复
3. 死锁的预防
4. 死锁的避免

死锁的检测与恢复

1. 死锁的检测

2. 死锁的恢复

- 利用抢占恢复
- 利用回滚恢复

- 通过杀死进程恢复

死锁的预防

1. 破坏互斥条件
2. 破坏占有且等待条件：规定所有进程在开始执行前请求所需要的全部资源。
3. 破坏不可抢占条件
4. 破坏环路等待条件：给资源统一编号，进程只能按编号顺序来请求资源。

死锁的避免

1. 安全状态
2. 单个资源的银行家算法
3. 多个资源的银行家算法

内存管理

什么是内存管理

- 逻辑地址到物理地址的映射
- 内存的分配与回收

常见的几种内存管理机制

- 连续的：块式内存管理
- 离散的：页式内存管理、段式内存管理、段页式内存管理

1. 段页式内存管理：既有了分段系统的共享和保护，又有了分页系统的虚拟内存功能。

内存分段

1. 段号：段基地址，段偏移量
2. 堆段、栈段、数据段、代码段
3. 分段的好处是能产生连续的内存空间
4. 分段的不足：内存碎片(内碎片和外碎片)、内存交换效率低(访问硬盘的速度比内存的速度低，每一次内存交换，都需要把一大段连续的内存数据写到硬盘上。)

内存分页

1. 页号(虚拟页号、物理页号)和页偏移量
2. 通过页表来进行地址转换，页表存在MMU中。页表包括物理页号每页所在的物理内存的基地址。

MMU(内存管理单元)

1. 页表：虚拟地址和物理地址的映射
2. 地址转换和TLB（快表）的访问与交互

段页式内存管理

1. 段号，段内页号，页内偏移

分页和分段分别是解决什么问题

- 分页是为了实现虚拟内存，从而获得更大的地址空间
- 分段是为了使程序和数据在逻辑上可以被划分为独立的地址空间，并且有利于共享和保护。

快表和多级页表

- 分页内存管理中：虚拟地址到物理地址的转换要快。解决虚拟地址空间大，页表也会很大的问题。
- 快表：局部性原理的概念，将最近访问到的页面或段放到内存中的快表中，当访问的页或者段不在内存中时，首先先去快表中查找，如果快表中存在则直接使用，如果不存在在快表中，则调用相应的页面调度算法，将需要的页或段放入快表中。
- 多级页表：**对局部性原理的充分利用**。为了避免全部页表一直存放到内存中，占用过多的空间，将那些不需要的页表存放到磁盘中。
- 二级页表：每个页表占4k内存，一个页表项占4字节。例如一个进程的虚拟内存空间占4G，进程真正使用的内存空间占4M，那么使用一级页表需要用4M内存来存放虚拟内存空间对应的页表，然后再找到占用的4M的内存空间。如果使用二级页表的话，使用4K的内存来定位页表项，然后真正的内存空间4M对应的页表项的大小为(4M = 1024 * 4K)4K，则最终只需要4K + 4K = 8K的内存空间。
- 使用一级页表需要访问2次内存，使用二级页表需要访问3次内存。

快表为什么比页表快?

1. 快表是一种特殊的高速缓冲存储器(Cache)，内容是页表的一部分或者全部内容。引入快表是为了加快地址映射速度，在虚拟页存储管理中设置了快表，作为当前页表的Cache。通常快表存放在MMU中。

- 快表和页表的区别和联系

1. 页表指出逻辑地址中的页号和所占主存物理块号的对应关系。页式存储管理在用动态重定位方式装入作业时，要用页表做地址转换工作。
2. 快表是存放在高速缓冲存储器中的部分页表。作为页表的Cache，它的作用与页表相似，但提高了访问内存效率。在页式存储管理中，利用页表做地址转换，读写数据需要访问内存两次（去内存读页表，利用得到的绝对地址去读内存），但是快表是存放在MMU中的，利用快表做地址转换，只需要读一次高速缓冲存储器，读一次内存，这样可以加速数据的查找和提高指令的执行速度。

分页和分段的比较

- 分页和分段都是为了提高内存利用率，减少内存碎片。
- 段和页都是离散存储的，段和页中的内存都是连续的。
- 分页对程序员来说是透明的，分段需要程序员显式的划分每个段。
- 页的大小是不可变得，段的大小是可变的。
- 地址空间的维度：页的地址空间是一维的，段的地址空间是二维的。
- **出现的原因：** 分页是主要是用于实现虚拟内存，从而可以获得更大的地址空间。分段是为了使程序和数据在逻辑上可以被划分为独立的地址空间，并且有利于共享和保护。段是逻辑信息的单位，在程序中可以体现为代码段和数据段，能够更好满足用户的需要。

逻辑(虚拟)地址和物理地址

为什么需要虚拟地址空间

1. 如果没有虚拟地址，程序都是直接访问物理地址，程序可以访问任意内存，很容易破坏操作系统，造成操作系统崩溃。然后无法运行多个应用程序，多个应用程序读取同一个物理内存地址，数据容易被覆盖，造成应用程序的崩溃。
2. 通过虚拟地址访问可以有以下优势：(1)程序可以使用虚拟地址访问内存中不相邻的大内存缓冲区。(2)使用虚拟地址访问大于可用物理内存的内存缓冲区，当物理内存可用量变小时，可以根据需要将代码或数据在磁盘和内存中移动。(3)不同进程使用的虚拟地址彼此隔离，一个进程无法更改由另一进程正在使用的物理内存。

虚拟内存

什么是虚拟内存(内存管理的技术，定义了一个连续的虚拟地址空间，并把内存扩展到硬盘空间)

- 让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。
- 为了更好的管理内存，将内存抽象成地址空间，每个程序都拥有自己的地址空间。每个地址空间分割成多个块，每个块成为一页，每一页映射到相应的物理内存上，但不需要映射到连续的物理内存上，也不需要所有的页必须在物理内存中。当程序引用到不在物理内存中的页时，将缺失的部分装入物理内存。
- 使应用程序错觉的认为自己拥有一段连续物理内存的错觉，实际上虚拟内存为应用程序分配了一段封闭的，连续的地址空间，地址空间分割成多个块，每个块为一页，分别映射到了不同的物理碎片上，有的还部分暂时的保存到了物理磁盘上。当应用程序需要的页不在内存空间中时，将缺失的部分装入物理内存中。

局部性原理

- 空间局部性：一旦应用程序访问到某个存储单元，在不久之后，附近的存储单元也可能被访问到。因为程序所访问的地址，可能集中在一定范围内，指令通常是顺序存放，顺序执行，数据一般以向量、数组等形式簇聚存储。
- 时间局部性：程序中某个指令一旦执行，不久之后，该指令可能被再次执行。该数据被访问过，一段时间之后可能会被再次访问。(程序中存在大量的循环)

虚拟存储器

- 时间换空间的概念
- 一部分程序被装入内存，当访问的数据或代码不在内存中时，将物理磁盘中的部分数据调入内存。

虚拟内存的技术实现

- 在离散分配的内存管理方式的基础上
- 请求分页存储管理：建立在分页管理之上，为了支持虚拟存储功能而增加的请求调页功能和页面置换功能。在程序运行之前，将部分页调入内存，使程序能够正常运行。当遇到程序请求的页不在内存中时，系统调用页面置换算法，将部分页调入内存，同时，也将部分用不到的页调出内存。
- 请求分段存储管理：建立在分段管理之上，提供了请求调段功能和分段置换功能。程序运行之前调入部分段使程序能够运行起来。可以使用请求调入中断，将程序即将使用到的段而又不在内存中的调入内存。当内存空间快满时，又可以将部分段调出内存。
- 请求段页式存储管理

页面置换算法

- OPT最佳页面置换算法
- FIFO先进先出页面置换算法
- LRU最近最久未使用页面置换算法
- LFU最少使用页面置换算法

文件管理

三种控制IO设备的方法

1. 使用程序控制IO
2. 使用中断驱动IO
3. 使用DMA的IO

DMA技术(直接内存访问)

在没有DMA技术之前，IO的过程是这样的：

- CPU发出对应的指令给磁盘控制器，然后返回。
- 磁盘控制器收到指令后就开始准备数据，会把数据放入到磁盘控制器的内部缓冲区中，然后产生一个中断。
- CPU收到中断信号，停下手头的工作，接着把磁盘控制器的缓冲区的数据一次一个字节的读进自己的寄存器，然后再把寄存器里的数据写入到内存，而在数据传输的期间CPU是无法执行其他任务的。

DMA技术：在进行IO设备(磁盘控制器)和内存(内核缓冲区)的数据传输的时候，数据搬运的工作全部交给DMA控制器，而CPU不再参与任何与数据搬运相关的事情，这样CPU就可以去处理别的事务。大致过程：

- 用户进程调用read方法，向操作系统发出IO请求请求读取数据到自己的内存缓冲区，进程进入阻塞状态。
- 操作系统收到请求后，进一步将IO请求发送DMA，然后让CPU执行其他任务。
- DMA进一步将IO请求发送给磁盘。
- 磁盘收到DMA的IO请求，把数据从磁盘读取到磁盘控制器的缓冲区中，当磁盘控制器的缓冲区被读满后，向DMA发出中断信号，告知自己的缓冲区已满。
- DMA收到磁盘信号，将磁盘控制器缓冲区中的数据拷贝到内核缓冲区中，此时不占用CPU，CPU可以执行其他任务。
- 当DMA读取足够多的数据，就会发出中断信号给CPU。
- CPU收到DMA的信号，知道数据已经准备好，于是将数据从内核拷贝到用户空间，系统调用返回。

计算机网络面试

计算机网络的五层架构

- 应用层
- 传输层
- 网络层
- 数据链路层
- 物理层

计算机网络为什么要分层

1. 各层之间相互独立：高层是不需要知道底层的功能是采取硬件技术来实现的，它只需要知道通过与底层的接口就可以获得所需要的服务；
2. 灵活性好：各层都可以采用最适当的技术来实现，例如某一层的实现技术发生了变化，用硬件代替了软件，只要这一层的功能与接口保持不变，实现技术的变化都并不会对其他各层以及整个系统的工作产生影响；
3. 易于实现和标准化：由于采取了规范的层次结构去组织网络功能与协议，因此可以将计算机网络复杂的通信过程，划分为有序的连续动作与有序的交互过程，有利于将网络复杂的通信工作过程化解为一系列可以控制和实现的功能模块，使得复杂的计算机网络系统变得易于设计，实现和标准化

应用层

- 常见使用TCP协议的应用服务：**HTTP、SMTP、POP3、FTP文本传送协议**
- 常见使用UDP协议的应用服务：**DHCP、NTP、TFTP**
- 同时使用TCP、UDP的应用服务：**SOCKS安全套接字协议、DNS地址解析协议**

DNS域名系统

- 分布式数据库，提供了IP地址和主机名之间相互转换的服务。
- 根域名服务器、顶级域名服务器、权威DNS域名服务器
- 本地域名服务器：当主机和ISP(互联网服务提供商)服务器进行连接时，该ISP会提供一台主体的IP地址，该主机会具有一台或者多台其本地DNS服务器的IP地址。通过访问网络连接，用户能够容易的确定DNS服务器的IP地址。当主机发出DNS请求后，该请求被发往本地DNS服务器，它起着代理的作用，并将该请求转发到DNS服务器层次系统中。
- 首先请求会先找到本地DNS服务器来查询是否包含IP地址，如果本地DNS无法查询到目标IP地址，就会向根域名服务器发出一个DNS查询。DNS涉及两种查询方式：递归查询和迭代查询。如果根域名服务器无法告知本地DNS服务器下一步需要访问哪个顶级域名服务器，就会使用递归查询。如果根域名服务器能够告知DNS服务器下一步需要访问的顶级域名服务器，就会使用迭代查询。在由根域名服务器 -> 顶级域名服务器 -> 权威DNS服务器后，由权威服务器告诉本地服务器目标 IP 地址，再有本地 DNS 服务器告诉用户需要访问的 IP 地址。
- 可以使用TCP或UDP协议，默认端口为53，大部分情况下使用UDP进行传输，当主机域名服务器向辅助域名服务器传送变化的那部分数据或者返回的响应超过512字节时，使用TCP传输。

FTP(文件传输协议)

DHCP(动态主机配置协议)

- 配置IP地址，DNS服务器的IP地址，子网掩码、网关IP地址

Web页面请求过程

1. **DHCP配置主机协议** 当一台主机还没有分配IP地址时，主机生成一个DHCP请求报文，并将这个报文放入具有目的端口67和源端口68的**UDP报文段**中。该报文段被放入一个具有广播IP目的地址和源IP目的地址的IP数据报中，该数据报被放置到MAC帧中，并广播到与交换机连接的所有设备。当与交换机相连的DHCP服务器收到广播帧后，将广播帧一层一层向上剖析，得到IP数据报、UDP报文段、DHCP请求报文，之后服务器生成DHCP ACK报文，**该报文包括IP地址、DNS服务器的IP地址、默认网关路由器的IP地址和子网掩码**。然后被放到UDP报文段中，接着放到IP数据报中，最后放入到MAC帧中。因为交换机具有自学习能力，交换机记录了MAC地址到其转发接口的交换机表项，因此这时候交换机知道应

该向哪个端口进行转发。主机收到后，不断剖析得到DHCP报文，配置它的IP地址、子网掩码和DNS服务器的IP地址，并在IP转发表中安装默认网关。

2. **ARP解析MAC地址** 主机通过浏览器生成一个TCP套接字，套接字向HTTP服务器发送HTTP请求。为了生成套接字，需要知道网站的域名对应的IP地址。主机生成一个DNS查询报文，该DNS查询报文被放入目的地址为DNS服务器IP地址的IP数据报中。IP数据报被放入以太网帧中，该帧将被发送到网关路由器。DHCP过程中，只知道网关路由器的IP地址，不知道网关路由器的MAC地址，则主机生成一个包含网关路由器IP地址的ARP查询报文，被放入到一个具有广播目的地址的以太网帧中，并向交换机发送以太网帧，交换机收到以太网帧后转发给所有连接的设备，包括网关路由器。网关路由器在收到以太网帧后不断分解得到ARP查询报文，发现其中的IP地址与其接口的IP地址匹配，则生成ARP回答报文，包含了它的MAC地址，发送回主机。
3. **DNS域名解析** 主机在收到网关路由器发送回来的MAC地址后，就可以继续DNS解析过程了。当网关路由器收到主机的发送的包含DNS查询报文的以太网帧后，抽取出IP数据报，并决定该数据报应该发送给的路由器。到达DNS服务器之后，DNS解析得到DNS查询报文，在DNS数据库中查找出相应的记录后，发送DNS回答报文，把DNS查询报文放入UDP报文段中，接着放入IP数据报中，通过路由器反向转发回网关路由器，并通过以太网交换机达到主机。
4. **HTTP请求页面** 主机在得到IP地址后，生成TCP套接字，并向Web服务器发送Http Get报文。在生成套接字之前必须经过三次握手，在连接建立之后，浏览器生成HTTP GET 报文，并交付给HTTP服务器，HTTP服务器通过TCP套接字读取HTTP GET报文，生成一个HTTP响应报文，将Web页面放入报文主体中，发回给主机。浏览器收到HTTP响应报文，抽取出Web页面，之后进行渲染，显示Web页面。

HTTP / HTTPS

1. 长连接和短连接：Connection：keep-alive，1.1之前默认是短连接，之后默认是长连接
2. 状态码
    - 1xx：正在处理
    - 2xx：成功
    - 3xx：重定向码，需要进行附加操作以完成请求
    - 4xx：服务器无法处理请求
    - 5xx：服务器处理请求出错
3. SSL/TSL四次握手
    1. ClientHello：客户端生成随机数Client random，并且携带着支持的TLS版本号以及加密套件的方式发送给服务器，服务器判断是否可用支持的加密方式，如果版本号+加密方式可用，继续进行
    2. ServerHello：服务端生成第二个随机数Server random，携带着SSL证书和服务器选择的密码套件发送给客户端，客户端判断是否可用，可用继续进行
    3. 认证：客户端的证书颁发机构会认证SSL证书，然后发送认证报文，报文中包含公开密钥证书。最后服务器发送`ServerHelloDone`作为hello请求的响应。第一部分握手阶段结束。
    4. 加密阶段：客户端收到服务端的回复后，发送Premaster secret的密钥字符串，这个字符串就是利用服务端的公钥进行加密的字符串，告诉服务端使用私钥解密这个字符串。然后客户端发送`Finished`告诉服务端自己发送完成了
    5. 服务端收到第三个随机数，并利用私钥进行解密，同时利用Client random、Server random和Premaster secret通过一定的算法生成HTTP链路数据传输的一个对话密钥。
4. Cookie、Session
    - HTTP是不保存状态的协议，不对请求和响应之间的通信状态进行保存

- Session
  - 通过服务端记录用户的状态
  - 服务端通过创建一个特定的Session之后就可以标示这个用户，并跟踪这个用户
  - 服务端一般将Session保存在内存或数据库中(Redis)
  - 通过Cookie中附加一个Session ID来进行跟踪用户
- Cookie
  - 都是用来跟踪浏览器用户身份的会话方式。
  - 一般用来保存用户的信息
  - 会话状态管理、个性化设置、浏览器行为跟踪

5. URI、URL

   - URL：统一资源定位符：是URI的子集
   - URI：统一资源标示符

6. HTTPS

   1. HTTP有安全问题：使用明文通信，内容可能被窃听。不验证通信方的身份，通信方的身份可能遭到伪装。无法证明报文的完整性，报文可能被篡改
   2. HTTPS：先让HTTP和SSL(Secure Sockets Layer)通信，SSL再和TCP通信，这样，HTTPS有了加密，认证和完整性保护
      - 加密：非对称加密方式进行通信：通信发送方在收到接收方的公开密钥之后，使用公开密钥对通信内容进行加密，接收方在收到通信内容后，用自己的私有密钥进行解密，得到通信内容
      - 认证：数字证书认证机构 (CA)，服务器运行人员向CA提出公开密钥的申请，CA在判明身份后，会对已签名的公开密钥对数字签名，然后分配这个已签名的公开密钥，与公开密钥证书绑定在一起。服务器会把证书发送给客户端，客户端在收到公开密钥后，先使用数字签名进行验证，如果验证成功就可以通信了。
      - 完整性保护：有了加密和认证这两个机制，就可以形成完整性保护 非对称加密的签名过程是，私钥将一段消息进行加签，然后将签名部分和消息本身一起发送给对方，收到消息后对签名部分利用公钥验签，如果验签出来的内容和消息本身一致，表明消息没有被篡改。

7. HTTP的消息结构

- 请求消息的结构：一个请求消息是由**请求行、请求头字段、一个空行和消息主体**构成。
- 响应消息的结构：状态行，消息报头，空行，响应正文

8. HTTP状态码 200：请求成功。500：程序错误，请求的网页程序本身就报错了。404：服务器上没有该资源，或者服务器上没有找到客户端请求的资源。 301：永久性的重定向。302：临时跳转。304：被请求的资源内容没有发生更改。 400：包含语法错误，无法被服务器解析。403：服务器已经接受请求，但是被拒绝执行。404：请求失败 500：服务器内部错误，无法处理请求。

9. HTTP1.0 与 HTTP1.1的区别

传输层

TCP的三次握手

A为客户端、B为服务端

- 首先B处于监听状态，等待客户端的连接
- A发送SYN=1，ACK=0的连接请求报文，选择一个初始序号x
- B收到请求连接报文，向A发送连接确认报文，SYN=1，ACK=1，确认号为x+1，同时也选择一个初始序号y
- A收到B的连接确认报文后，向B发送确认报文，确认号为y+1，序号为x+1

延迟ACK的原因

ACK是可以合并的，如果连续收到2个TCP包，只需要回复最终的ACK就可以了，可以降低网络流量。 如果接收方有数据要发送，就会放到发送数据的TCP包里，带上ACK信息，这样就避免了大量重复的ACK以单独的包发送，减少了网络流量。

三次握手的原因

第三次握手是为了防止丢失的连接请求到达服务器，让服务器错误的打开连接。

客户端发送的连接请求如果因为网络问题在网络中滞留，这时客户端等待一个超时重传时间之后，会向服务器重新发送连接请求。如果这个滞留的请求最终到达服务器，如果没有第三次握手，服务器就会再次建立一个连接，存在第三次握手客户端就会忽略服务器之后发送的确认连接报文，不进行第二次第三次握手，这样连接就不会建立。

进行三次握手的主要作用是为了确认双方的接收能力正常和发送能力正常。

什么是半连接队列

当服务器第一次收到客户端的SYN时，处于SYN_RCVD状态，此时双方还没有完全建立链接，服务器会把这种状态的请求放在一个队列里。

全连接队列

已经完成三次握手，建立起连接的就是放在全连接队列中，如果队列满了会有出现丢包的现象。

ISN(初始序号)

当一端为建立连接而发送它的SYN时，它为连接选择一个初始序号。ISN随时间而变化，因此每个连接都将具有不同ISN。ISN可以看作是一个32bit的计数器，每4ms加1。这样选择序号的目的在于防止在网络中被延迟的分组在以后又被传送，而导致某个连接的另一方对它作出错误的解释。 三次握手最重要的一个功能就是客户端和服务端交换ISN，以便让对方知道接下来接收数据的时候如何按照序号组装数据。

三次握手可以携带数据么

第三次可以携带数据。

SYN攻击是什么?

服务端的资源分配是在第二次握手的时候，而客户端的资源分配是在完成三次握手时分配的，所以服务器更容易受到SYN泛洪攻击。 在短时间内伪造大量的不存在的IP地址，并向服务器发送SYN包，服务器则回复确认包，并等待客户端确认，由于源地址不存在，因此服务器不断的重发直到超时，这些伪造的SYN包长时间的存放在半连接队列中，导致半连接队列满，最后导致正常的SYN包被丢弃。从而引起网络拥塞甚至系统瘫痪。

TCP三次握手的优化

参考链接

客户端的优化 当客户端在等待服务端的ACK回复的时候，如果等待超时，则会重发SYN，重发次数默认为6次，第一次重发发生在1秒后，接下来的重发时间间隔以翻倍的方式增加，一共经历127秒后，才会终止三次握手。这时可以根据网络的稳定性和服务器的繁忙程度修改重试次数，调整三次握手的时间上限。比如在内网中通讯时，可以减少重发次数，尽快把错误暴露给客户端。 **服务端的优化** 在回复SYN+ACK时，服务器端会将未完成的握手信息放到半连接队列中，如果队列溢出SYN报文会丢失，导致连接失败，我们可以根据netstat -s 给出的统计信息判断队列长度是否合适，进而调整队列的长度。在收到ack时，服务器端会连接移入accept队列中，如果accept队列溢出，系统会丢弃ACK，可以通过netstat -s 给出的统计信息查看accept队列长度是否合适，可以适当调节队列的上限。

在服务器回复SYN+ACK时，若超时将重发SYN+ACK，网络稳定时SYN+ACK的重试次数可以降低。另外为了应付SYN泛洪攻击，应将tcp_syncookies的参数设为1，它在半连接队列满时，将开启syncookies功能，服务器根据状态计算出一个值，放在SYN+ACK的报文中，客户端在返回ACK时也会将该值返回，如果合法则认证建立成功。 **TFO绕过三次握手** 在首次连接建立时，客户端的SYN会明确告诉服务端自己想使用TFO功能，服务端在收到该请求后，会把客户端的ip地址用自己知道的密钥加密，作为Cookie携带在SYN+ACK中，客户端收到后会将Cookie保存到本地。当再次向服务端发送建立连接请求时，就可以在第一次SYN报文中携带数据，并附带Cookie，当服务端验证Cookie合法时，会直接建立成功，并把请求发给进程处理。

TCP四次挥手

- A发送释放报文，FIN=1，**A进入FIN-WAIT1状态**
- B收到释放报文后，发出确认，此时TCP处于半关闭状态，B能向A发送数据，但是A不能向B发送数据，**A进入FIN-WAIT2状态，B进入CLOSE-WAIT状态**
- 当B不再需要连接的时候，B发送释放报文，FIN=1。**B进入LAST-ACK状态**
- A收到后发出确认，并进入TIME-WAIT状态，等待2MSL(最大报文存活时间)后释放连接。**A进入TIME-WAIT状态**
- B收到A的确认后释放连接。

四次挥手的原因

客户端发出连接释放报文之后就入了CLOSED-WAIT状态，这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕后，服务器会发送FIN连接释放报文。

第二次和第三次为什么不能合并?

当服务器执行完第二次挥手后，服务端可能还需要向客户端发送数据，所以此时服务端会等待把之前未传完的数据传输完毕后再发送关闭请求。

为什么会有TIME-WAIT状态(MSL 报文段最大生存时间)

客户端在收到服务端的FIN连接释放报文后，不会立刻进入CLOSED状态，而是会等待2MSL后再释放连接，理由如下：

- 确保客户端最后发出的确认报文能够到达服务端，如果没有到达服务端，服务端会重新发送连接释放请求报文，A等待时间就是为了防止这种事情发生。
- 为了让本次连接持续时间内产生的所有报文都在网络中消失，使得下一次新的连接不会产生旧的连接请求报文。(比如服务端在发送连接释放请求报文前，还发送了一次数据，结果这次数据在网络中滞留

了，连接释放请求报文提前到达，为了不让这次滞留的数据出现在下一次新的连接中，要么等待接收它，要么等待这次请求数据死亡)。

TCP可靠传输

1. 应用程序被分割成TCP认为最合适发送的报文段
2. TCP对每个包进行编号，接收端对包进行排序，然后将有序的数据发送应用层
3. 校验和：TCP保持它的首部与所携带的数据的校验和，如果传输过程中数据发生变化，校验和会有差错，TCP将会丢弃该报文段
4. TCP丢弃重复的报文段
5. 流量控制：TCP连接的双方都会有一定的缓冲区，当接收方来不及处理发送方发来的数据时，会在返回确认的报文中携带能够容纳的数据的缓冲区大小，提示发送方降低发送的速率，防止包丢失。采用了滑动窗口协议。
6. 拥塞控制(慢开始、快速避免、快重传、快恢复)
7. ARQ协议：发完一个分组就停止发送，等待对方确认再接着发送。
8. 超时重传

TCP滑动窗口

TCP连接的双方都会有一定的缓冲区，发送方和接收方各有一个窗口，接收方通过TCP报文段中的窗口大小来告诉发送方自己的窗口大小，发送方根据这个值和其他信息来设计自己的窗口大小。

发送方窗口内的字节都允许被发送，接收方窗口内的字节都允许被接收。如果发送方的左部字节已经发送并得到了确认，发送方的窗口就向右滑动一段距离。接收方的左部字节已经发送确认并交付主机，就向右滑动一段距离。

接收方的滑动窗口只会对窗口内的最后一个按序到达的字节进行确认，比如收到字节为{31,33,34}，只会对31进行确认，接收方发送ack=32，发送方得到一个字节确认之后，**就会知道该字节之前的字节都已经被接收。**

TCP流量控制

流量控制是为了控制对方发送的速率，保证接收方来得及接收。

TCP拥塞控制

1. 拥塞窗口(cwnd)：状态变量
2. 慢开始和拥塞避免 发送的最初执行慢开始，cwnd=1，发送方只能发送一个报文段，当收到确认后，将cwnd加倍。 由于慢开始到后面会使得cwnd增长的越来越快，会使得网络拥塞的可能性更高，所以设置一个ssthresh，当增长的cwnd>=ssthresh时，cwnd之后每次增长的次数变为1，当出现超时时，重新执行慢开始，ssthresh设为上次超时cwnd的一半。
3. 快重传和快恢复 快重传和快恢复是指cwnd的设定值，而不是cwnd的增长率，慢开始cwnd设定为1，快恢复的cwnd设定为ssthresh。 接收方只对窗口内最后一个有序到达的报文段进行确认，例如收到{31,33,34}，只会对31进行确认。当发送方连续收到三个重复确认，则判断下一个报文段丢失，立刻进行重传。在这种情况下，只是丢失个别报文段，不是出现拥塞，则会将ssthresh设置为cwnd/2，cwnd=ssthresh，直接进入拥塞避免。

ARQ协议(自动重传请求)

停止等待ARQ协议

- 每发完一个分组就停止发送，等到确认。如果过了一段时间还是没有收到ACK，则重新发送该分组。
- 在停止等待协议中，接收方收到了重复分组，会丢弃重复分组，但还是要发送ACK。

优点： 简单。 **缺点：** 信道利用率低，等待时间长。 **分三种情况：** (1)无差错情况 (2)出现差错情况 (3)确认丢失和确认迟到

连续ARQ协议

- 发送方维护一个发送窗口，不需要等待接收方的ACK，发送方中发送窗口中的分组可以连续发送出去，采用累计确认的方式。

优点 信道利用率高，容易实现，即使确认丢失，也不必重传 **缺点** 接收方不能正确反映出已经正确接收的所有分组的信息。比如当第三个分组丢失时，接收方只返回前两个分组接收成功，这时发送方要发送第三个分组及之后的所有分组。

TCP与UDP的比较

1. TCP是面向连接的，可靠的的传输协议，有流量控制、拥塞控制，提供全双工通信、面向字节流(把应用层传下来的报文看成字节流，把字节流组织成大小不一的数据块)，点对点交互通信
2. UDP是无连接的，尽最大可能交付的，没有拥塞控制、面向报文(对应用程序传下来的报文不合并也不拆分，只添加UDP首部)，支持一对一、一对多、多对多的交互通信。
3. UDP首部8字节，包括源端口、目的端口、长度、检验和。12字节的伪首部：源IP地址、目的IP地址、0、17、UDP长度。
4. TCP首部20～60个字节，包括源端口，目的端口，序号，确认号，数据偏移，确认ACK，同步SYN，终止FIN，窗口

UDP实现可靠连接

1. 添加seq/ack机制，保证数据发送到对端
2. 添加发送和接收缓冲区，主要是用户超时重传
3. 添加超时重传机制

- 发送端发送数据时，随机生成一个seq=x.然后将每一片的按照数据大小分配seq。数据达到接收端后接收端发入缓存，并发送一个ack=x的包，当发送端收到ack后，删除缓冲区对应的数据。时间到后，定期检查任务是否需要超时重传数据。

TCP粘包拆包问题

什么是粘包?

客户端可以不断的向服务端发送数据，服务端在接收数据的时候就会出现两个数据报粘在一起的情况。

1. TCP是基于字节流的，虽然应用层和TCP传输层之间的数据交互是大小不一的数据块，但是TCP把这些数据块仅仅看成一连串无结构的字节流，没有边界。
2. 从TCP的帧结构也能看出，在TCP的首部没有表示数据长度的字段。 基于这样的情况，才有可能出现粘包或者拆包现象的可能。一个数据包中包含了发送端发送的两个数据包的信息。接收端接收到了两个数据包，这两个数据包要么是不完整的，要么就是多出来一块。

TCP粘包是怎么产生的?

- 发送方产生粘包：采用TCP协议传输数据的客户端和服务端经常保持一个长连接的状态，双方在连接不断开的情况下，可以一直传输数据。但当发送的数据包过于小时，TCP协议默认的会启用Nagle算法，将这些较小的数据包进行合并发送。这个合并的过程就是发生在发送缓冲区中进行的，也就是说数据发送出来的时候就已经是粘包的状态了。

- 接收端产生粘包：当拿数据的速度小于放数据的速度时，我们在程序中调用的读取数据函数不能及时对缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入缓冲区的末尾，等待我们读取数据时就是一个粘包。

怎么解决粘包

1. 特殊字符控制
2. 在包头首部添加数据包的长度。

网络层

- 网络层是整个互联网的核心，因此应当让网络层尽可能简单。网络层向上只能提供简单灵活的、无连接的、尽最大努力交互的数据报服务。
- 与IP协议配套使用的还有三个协议：地址解析协议ARP、网际控制报文协议ICMP、网际组管理协议IGMP

IP协议

- IP数据报的格式：版本、首部长度、区分服务、总长度，标识、标志、片偏移，生存时间、协议、首部检验和，源地址，目的地址

ARP协议：地址解析协议

ARP实现由IP地址得到MAC地址 每个主机都有一个 ARP 高速缓存，里面有本局域网上的各主机和路由器的 IP 地址到 MAC 地址的映射表。 如果主机 A 知道主机 B 的 IP 地址，但是 ARP 高速缓存中没有该 IP 地址到 MAC 地址的映射，此时主机 A 通过广播的方式发送 ARP 请求分组，主机 B 收到该请求后会发送 ARP 响应分组给主机 A 告知其 MAC 地址，随后主机 A 向其高速缓存中写入主机 B 的 IP 地址到 MAC 地址的映射。

ARP欺骗

RARP反向地址解析协议

- 主机向RARP服务器获取自己的IP地址。(必须处于客户端同一子网中)
- MAC地址转换成IP地址
- 在网络上发送一个RARP请求的广播数据包，请求任何收到次请求的RARP服务器分配一个IP地址。RARP服务器在收到请求数据包后，查找RARP表项，查到该MAC地址对应的IP地址，如果存在RARP服务器就给主机发送一个响应数据包，并将次IP地址提供给对方使用。如果不存在，RARP服务器对此不做任何的响应。发送主机利用得到的IP地址进行通信，如果一直没有收到RARP服务器的响应消息，表示初始化失败。
- RARP服务器一般要为多个主机提供硬件地址到IP地址的映射，该映射包含在一个磁盘文件中，而内核一般不读取和分析磁盘文件(ARP服务器在内核中)，所以RARP的功能就由用户进程来提供。更为复杂的是，RARP请求是作为一个特殊类型的以太网数据帧来传送的，其请求是在硬件层面进行广播的，不经过路由器进行转发。RARP请求数据包中没有IP地址，自然就无法通过路由器进行转发。因此路由器

是工作在网络层的，网络层的协议是IP协议，ARP请求能够通过路由器进行转发，是因为ARP请求数据包中有IP字段，而RARP中没有该字段。

- RARP和DHCP的区别：RARP是数据链路层实现的，DHCP是应用层实现的，RARP只能实现MAC到IP地址的查询工作，RARP服务器上的MAC和IP地址必须是事先静态配置好的，但DHCP可以实现除静态分配外的动态IP地址分配以及IP地址租期管理等相对复杂的功能。

ICMP协议

Ping

- Ping 的原理是通过向目的主机发送 ICMP Echo 请求报文（类型8），目的主机收到之后会发送 Echo 回答报文（类型0）。Ping 会根据时间和成功响应的次数估算出数据包往返时间以及丢包率。

1. ping命令执行的时候，首先会创建一个ICMP回送请求报文，类型8，然后ICMP协议将这个数据报连同ip地址一起交给IP层，IP层的协议字段设为 1 表示ICMP协议，然后加入MAC头，如果本地映射表中不知道目的IP地址的MAC地址，会先发送一个ARP请求报文，得到目的IP地址的MAC地址，然后将在数据链路层构建一个数据帧。另一台主机收到后会构建一个回送响应消息包，类型为0，在发送给源主机。
2. 如果在一段时间内没有收到ICMP回送响应消息，则说明主机不可达，否则主机可达。

traceroute

1. 故意设置特殊的TTL，来跟踪去往目的地时沿途经过的路由器。

- 利用IP数据报的生存期限，从1开始按照递增顺序的同时发送UDP包，强制接收ICMP超时消息的一种方法。
- 比如，将 TTL 设置 为 1，则遇到第一个路由器，就牺牲了，接着返回 ICMP 差错报文网络包，类型是时间超时。接下来将 TTL 设置为 2，第一个路由器过了，遇到第二个路由器也牺牲了，也同意返回了 ICMP 差错报文数据包，如此往复，直到到达目的主机。这样的过程，traceroute 就可以拿到了所有的路由器 IP。当然有的路由器根本就不会返回这个 ICMP，所以对于有的公网地址，是看不到中间经过的路由的。
- 发送方如何知道UDP包到达了目的主机：traceroute在发送UDP包时，会填入一个不可能的端口号作为目的地址的端口号，当目的主机接收到ICMP报文后，会回发一个差错报文消息，但这个差错报文的类型是端口不可达。当差错报文的类型是端口不可达时，则说明UDP数据包达到了目的主机。

2. 故意设置不分片，从而确定路径的MTU。

- 为了路径MTU发现。
- 首先在发送端主机发送IP数据报时，将IP包首部的分片禁止标识位设为1。根据这个标识位，途中的路由器不会对大数据报进行分片，而是将包丢弃。随后通过一个ICMP的不可达消息将数据链路上MTU的值一起发送给主机，不可达消息的类型为需要进行分片但设置了不分片位。发送主机端每次收到ICMP差错报文时就减少包的大小，以此来定位一个合适的MTU的值，以便能达到目的主机。

数据链路层

物理层

数据库

事务

什么是事务

事务是指满足ACID特性的一组操作，可以通过Commit提交一个事务，也可以使用Rollback进行回滚。

事务的四大特性

1. 原子性.事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。回滚可以用回滚日志来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些操作即可。
2. 一致性.数据库在事务执行前后都保持一致性状态。在一致性状态下，所有事务对同一数据的读取结果都是相同的。
3. 隔离性.一个事务所做的修改在最终提交之前，对其他事务是不可见的。
4. 持久性.一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统崩溃，事务的执行的结果也不能丢失。系统发生崩溃，可以用重做日志进行恢复，从而实现持久性。

ACID的理解

- 只有满足一致性，事务的执行结果才是正确的。
- 在无并发的情况下，事务串行执行，隔离性一定能满足。此时只要能满足原子性，就一定能满足一致性。
- 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还要满足隔离性才能满足一致性。
- 事务持久化是为了能应对系统崩溃的情况。

事务是怎么实现的?

ACID

1. 原子性：实现要么全部失败，要不全部成功。通过回滚实现的，用到undo log。每次更新操作，都会将修改之前的数据存入undo log。回滚时只需要对undo log做逆向操作。delete逆向操作为insert，insert逆向操作为delete，update逆向操作为update。
2. 持久性：通过redo log，将每次操作完成的数据存入redo log。从而达到故障后恢复。
3. 隔离性：四种隔离级别，通过读写锁+MVCC实现。
4. 一致性：通过回滚、恢复和在并发环境下的隔离做到一致性。

并发一致性问题

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性的问题。

- 丢失修改。一个事务的更新操作被另一个事务的更新操作替代。比如A修改数据的时候，事务未结束，B就来修改数据，结果导致第一个事务修改的数据丢失
- 读脏数据。事务A在进行修改后，事务还未提交，事务B就读到了事务A修改的数据，结果事务A又撤销了修改，事务B读到的数据就是脏数据
- 不可重复读。在一个事务A内多次读同一个数据，在这个事务还没有结束的时候，另一个事务对该数据进行了修改操作，导致第一个事务两次读取的结果不太一样。
- 幻读。幻读也是不可重复读的一种情况，当一个事务读取了几行数据，接着另一个事务插入了几行数据，在随后的查询中，第一个事务就会发现多了一些原本不存在的记录。比如学生数量原来为5，另一个事务又插入了一名学生，导致再次查询时学生数量变成了6.

封锁

封锁的粒度：表级锁和行级锁

- 表级锁，粒度最大的一种锁，对当前操作的整张表进行加锁，资源消耗比较少，加锁快，不会出现死锁，发生锁的冲突概率大，并发度低
- 行级锁，粒度最小的一种锁，对当前操作的行进行加锁，资源消耗比较大，加锁慢，会出现死锁，但是并发度高。
- 锁分类：共享锁S（读锁），排他锁X（写锁）意向锁（表锁）（IS,IX） 一个事务对数据进行读取和更新的时候，需要对这个对象加X锁。 一个事务对数据进行读取的时候，需要对这个对象加S锁，加锁期间，其他事务想要对这个对象进行操作的时候，只能加S锁，不能加X锁。 使用意向锁更容易实现多粒度封锁。IX表示希望对这个对象加X锁，IS表示希望对这个对象加S锁。S锁只与S锁和IS锁兼容，也就是说事务A想对数据加S锁时，其它事务可以对表或表中的行加S锁。

封锁协议

三级封锁协议

1. 一级封锁协议，事务A想要修改数据，必须加X锁，事务结束才能释放锁。（解决丢失修改问题）
2. 二级封锁协议，在一级封锁协议的基础上，事务A想要读数据必须加S锁，读取完马上释放锁。（解决读脏数据问题）
3. 三级封锁协议，在二级封锁协议的基础上，事务A想要读数据必须加S锁，事务结束才能释放锁。（解决不可重复读的问题）

两段封锁协议

加锁和解锁分为两个阶段进行。 可串行调度：通过并发控制，使得并发执行的事务与某个串行执行的事务结果相同。串行执行的事务互不干扰，不会出现并发一致性问题。

隔离级别

- 未提交读。最低的隔离级别，可能会导致脏读，幻读和不可重复读
- 提交读。允许读取已经提交的事务，可以阻止脏读，但不能阻止幻读和不可重复读
- 可重复读（MySQL默认级别）。对同一字段的多次读取结果是一样的，除非数据被本身事务自己所修改，可以阻止脏读和不可重复读，可能会导致幻读。
- 可串性化。最高的隔离级别，完全服从ACID的隔离级别。 MySQL的存储引擎InnoDB在可重复读的隔离级别下，使用的是Next-key Lock锁算法，可以避免幻读的产生，已经完全保证事务的隔离性要求。

多版本控制

多版本并发控制（MVCC）

MVCC是InnoDB存储引擎实现隔离级别的一种方式，实现了提交读和可重复读两种隔离级别。可串性化隔离级别需要对所有读取的行都加锁，单独使用MVCC不能实现。写操作去读最新的版本快照，读操作去读旧的版本快照，没有互斥关系。

版本号

- 系统版本号：SYS_ID：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：TRX_ID：事务开始时的系统版本号。
- MVCC的多版本是指多个版本的快照，快照存放在Undo日志中，该日志通过回滚指针将一个数据行的所有快照连接起来。

隐藏的列

MVCC在每行记录后面都保存着两个隐藏列，用于保存两个系统版本号。

- 创建版本号：创建一个数据行的快照时系统的版本号
- 删除版本号：如果该快照的删除版本号大于当前事务版本号表示事务有效，否则表示已经被删除。

实现过程

当开始一个事务时，当前事务的版本号肯定会大于当前所有行快照的创建版本号。

1. select
2. insert 当前系统版本号作为当前快照的创建版本号。
3. update 当前系统版本号作为更新前的删除版本号，当前系统版本号作为更新后的数据行快照的创建版本号。
4. delete 当前系统版本号作为当前快照的删除版本号。

Next-Key Locks

InnoDB引擎中的锁的算法：Record lock，Gap lock，next-key lock

- Record lock：锁定一个记录上的索引，而不是记录本身。如果表没有设置索引，InnoDB存储引擎会自动在主键上创建隐藏的聚簇索引，所以Record lock依然可以用。
- Gap lock：锁定索引之间的间隙，但不包含索引本身。
- next-key lock：是Record lock和Gap lock的结合，不仅锁定记录上的索引，也锁定索引之间的间隙。它锁定一个前开后闭区间。

数据库的性能调优

MySQL

数据类型

字段类型

- 整型
- 浮点数
- 字符串 CHAR，VARCHAR VARCHAR这种变长类型能够节省空间，因为只需要存储必要的内容。但是在执行UPDATE时可能会使行变得比原来更长，当超过一个页所能容纳的大小时，就要执行额外的操作，MyISAM会将行拆分成不同的片段存储，而InnoDB则需要分裂页来使行放进页内。
- 时间和日期

选择优化的数据类型

1. 更小的更好；更小的数据类型通常更快，因为它们占用更少的磁盘、内存和CPU缓存，并且处理时需要的CPU周期也更少。
2. 简单最好；例如，整型比字符串操作代价更低；使用内建类型而不是字符串来存储日期和时间；用整形存储IP地址。
3. 尽量避免NULL；如果查询中包含可为NULL的列，对于MySQL来说更难优化，因为可为NULL的列使得索引、索引统计和值都比较都更复杂。

VARCHAR和CHAR

- 字符串的最大长度比平均长度大很多；列更新的很少，所以碎片不是问题适合用VARCHAR
- CHAR适合存储很短的字符串，或者所有值都接近同一个长度，如密码的MD5值。对于经常变更的数据，CHAR比VARCHAR更好，因为CHAR不容易产生碎片。

存储引擎

InnoDB

1. 是MySQL的默认存储引擎，只有在需要它不支持的特性时，才考虑使用其他的存储引擎。
2. 实现了四个标准的隔离级别，默认级别是可重复读(RR)。在可重复读的隔离级别上，通过多版本并发控制(MVCC)+间隙锁(Next-key Locking)防止幻读。
3. 主索引是聚簇索引，在索引中保存了数据，从而避免了直接读取磁盘，因此对查询性能有很大的提升。
4. 内部做了很多优化，包括从磁盘读取数据是采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。
5. 支持真正的在线热备份。其他存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合的场景中，停止写入可能也意味着停止读取。

MyISAM

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，提供了大量的特性，包括压缩表、空间数据索引等。

比较

1. 事务：InnoDB是事务型的，可以使用commit和rollback语句。
2. 并发：MyISAM只支持表级锁，而InnoDB还支持行级锁。
3. 外键：InnoDB支持外键。
4. 备份：InnoDB支持在线热备份。
5. 崩溃恢复：MyISAM崩溃后发生损坏的概率比InnoDB高很多，而且恢复的速度更慢。
6. MVCC：InnoDB支持。应对高并发事务, MVCC比单纯的加锁更高效;MVCC只在 RC 和 RR 两个隔离级别下工作;MVCC可以使用乐观(optimistic)锁和悲观(pessimistic)锁来实现；各数据库中MVCC实现并不统一。
7. 其他：MyISAM支持压缩表和空间数据索引。

索引

索引是帮助数据库高效获取数据的数据结构。索引是在存储引擎层面实现的，不是在服务器层实现的。 常见的索引类型有：hash、b树、b+树

B+树

B+树是基于B树和叶子节点顺序访问指针进行实现的，它具有B树的平衡性，而且通过顺序访问指针来提高区间查询的性能。 进行查找操作时，首先在根结点进行二分查找，找到一个key对应的指针，直到找到叶子结点，然后在叶子结点上进行二分查找，找出key对应的data。插入删除操作记录会破坏平衡树的平衡性，因此在插入删除操作后，都需要对树进行一次分裂、合并、旋转等操作来维护平衡性。

- 为什么不使用红黑树

1. 更少的查找次数。平衡树查找操作的时间复杂度等于树高，红黑树的树高明显比B+树的树高大很多，检索次数也就更多。
2. 利用计算机预读的特性。为了减少磁盘I/O，磁盘往往不是严格按需读取的，而是每次都会预读。预读过程中，磁盘进行顺序读取。操作系统一般将内存和磁盘分割成固态大小的块，每一块称为一页，内存与磁盘以页为单位进行交换数据。数据库会将索引的一个节点大小设置为页的大小，使得一个I/O就能完全存入一个节点，并且可以利用预读特性，相邻节点也能够被预先读入。

B+树一个节点到底多大合适?

- 一个节点一页大小或页的倍数最为合适。
- 在MySQL中，B+树一个节点的大小为一页，16k
- 为什么一页就够了，对于叶子结点，假设一行的数据大小为1K，则可以存储16条记录。对于非叶子节点，key8字节，指针6字节，一共14字节，则16k可以存储1170个。那么高度为3的B+树可以存储1170 * 1170 * 16 = 21902400个数据。B+树高度为3树就能满足千万级的数据存储。通常通过主键索引1-3次I/O就可以找到相应的数据。

MySQL索引

- B+树索引 可以指定多个列作为索引列，多个索引列共同组成键。适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找，如果不是按照索引列的顺序进行查找，则无法使用索引。分为主索引和辅助索引。

1. 主索引的叶子结点data域记录着完整的数据记录(聚簇索引)。因为无法把数据行存放在两个不同的地方，所以一个表只有一个索引。
2. 辅助索引的叶子结点的data域记录着主键的值。使用辅助索引时，先找到主键值，然后再到主键值中进行查找。

- 哈希索引 哈希索引能够以O(1)时间进行查找，但是失去了有序性。无法用于排序分组；只支持精确查找，无法用于部分查找和范围查找。InnoDB中有一个自适应哈希索引，当某个索引值被使用的非常频繁时，会在B+树索引之上在创建一个哈希索引，使之具有快速的哈希索引查找的优点。
- 为什么最常用B+树索引
    1. 很适合磁盘存储，能够充分利用局部性原理，磁盘预读。
    2. 很低的树高度，能够存储大量数据。
    3. 索引本身占用的内存很小。
    4. 能够很好的支持单点查询，范围查询，有序性查询。

索引优化

1. 独立的列：在进行查询的时候，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。
2. 多列索引：在需要使用多个列为条件进行查询时，使用多列索引比使用多个单列索引性能更好。

3. 索引列的顺序：让选择性最强的索引列放在前面。索引的选择性是指不重复的索引值和记录总数的比值，也就是说索引的唯一性越强，重复的越少，越应该放在前面。
4. 前缀索引：对于BLOB、TEXT和VARCHAR类型的列，必须使用前缀索引，只索引开始的部分字符。
5. 覆盖索引：索引包含所有需要查询的字段的值。
6. 索引下推：(index condition push ICP) 是在非主索引上做优化，可以有效减少回表的次数，大大提升查询的效率。在不使用ICP的情况下，在使用非主索引进行查询时，存储引擎通过索引检索到数据，然后返回给MySQL服务器，服务器判断数据是否符合条件。在使用ICP的情况下，如果存在某些被索引的列的判断条件时，MySQL服务器会将这部分判断条件传输给存储引擎，然后存储引擎通过判断索引是否符合MySQL服务器传递的条件。只有当索引符合条件时才会将数据检索出来返回给MySQL服务器。

索引的优点

- 大大减少了服务器需要扫描的数据行数
- 帮助服务器避免进行排序和分组，也就是不需要创建临时表。(B+树是有序的，可以用于order by和group by操作。临时表主要用于排序和分组时创建。因为不需要排序和分组，所以不需要临时表)
- 将随机I/O变为顺序I/O。(B+树索引是有序的，也就将相邻的数据都存储在一起。)

索引使用的场景

- 对于非常小的表，大部分的简单的扫描比建索引更有效。
- 对于中到大型的表，索引就很有效。
- 对于特大型的表，建立和维护索引的代价随之增加，一般会使用分区技术。

MySQL中的索引叶子节点存放的是什么?

- MyISAM：主键索引和辅助索引的叶子节点存放的都是key和key对应数据行的地址。
- InnoDB：主键索引存放的是key和对应的数据行，辅助索引存放的是key和key对应的主键值。因此在使用辅助索引的的时候通常会检索两次索引，首先检索辅助索引的主键值，然后用主键值到主键索引中获得记录。

为什么要为InnoDB表设置自增列做主键

1. 使用自增列做主键，写入顺序是自增的，和B+树叶子节点分裂顺序一致。
2. 表不指定自增列做主键，同时也没有可以被选为主键的唯一索引，InnoDB就会选择内置的rowid作为主键，写入顺序和rowid的增加顺序一致。InnoDB表的顺序写入顺序能和B+树索引的叶子节点顺序一致的话，这时候存取效率最高。

应用场景，最佳左前缀原则细化

聚簇索引、非聚簇索引、覆盖索引、复合索引

- 聚簇索引：一种数据存储方式，聚簇索引把索引和数据行放到一起，找到索引也就找到了数据，无需进行回表操作。InnoDB必然会有一个聚簇索引。
- 非聚簇索引：索引和数据行是分开的，找到索引后，需要通过对应的数据行的地址找到对应的数据行。
- 回表查询：InnoDB中，对于普通索引，索引和数据行的存放是分开的，因此在找到索引之后还需要通过主键值再走一遍主键索引，才能找到相应的数据。
- 走普通索引一定会回表操作么？不一定，如果查询语句的字段恰好命中了索引，也就是说，查询的字段恰好包含了普通索引和主键索引，就不需要回表操作，直接查询出来就行。

- 覆盖索引：当索引上包含了查询语句中的所有字段时，无需进行回表操作就能拿到所有请求的数据，因此速度很快。
- 复合索引(联合索引)
    - 联合索引底层使用的是B+树索引，并且还是只有一棵树，只是此时的排序会：首先按照第一个索引排序，然后再按照第二个索引排序，以此类推。
    - 最佳左前缀原则：因此后面的索引是在前边索引排序的基础上进行的，如果没有左边的索引，右边的索引看起来是无序的。

主键索引和普通索引的区别

1. 普通索引是最基本的索引类型，没有任何限制，值可以为空，仅加速查询。普通索引是可以重复的，一个表中可以有多个普通索引。
2. 主键索引是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。索引列的所有值都只能出现一次，即必须唯一。
3. 唯一索引和普通索引在查询能力上是没有差别的，主要考虑的是对更新性能的影响。唯一索引在更新时会进行唯一性检查，不会用到change buffer，而普通索引则会用到change buffer。

一条SQL的执行过程详解

1. 首先系统与MySQL进行交互之前，MySQL驱动会帮我们建立好连接，然后我们将语句通过数据库连接池发送将一次请求发送到MySQL数据库中。
2. MySQL中处理请求的线程在获取到请求以后获取SQL语句然后交给SQL接口去处理。
3. 解析器将SQL语句解析成相应的语句，之后查询优化器根据成本(IO成本和CPU成本)最小原则来选择使用对应的索引，之后优化器调用存储引擎的接口去执行SQL。
4. 执行器根据MySQL的查询计划，先是从缓存池中查询数据，如果没有就去数据库中查询，如果查询到了就将其放入到缓存池中。
5. 在数据被缓存到缓存池的同时，会写入undo log日志文件。(原子性，回滚的时候用到undo log)
6. 更新的动作是在BufferPool中完成的，同时会将更新后的数据添加到redo log buffer中
7. 完成以后就可以提交事务，在提交的同时(1)将redo log buffer中的数据刷入到redo log文件中(持久性，写入redo log)。(2)将本次操作记录写入到bin log文件中。(3)将bin log文件名字和更新内容在bin log中的位置记录到redo log中，同时在redo log最后添加commit标记。

BufferPool(缓冲池)

- 用来缓存数据和索引在内存中，主要用来加速数据的读写。InnoDB会把那些热点数据和认为即将访问到的数据放到BufferPool中，提升读取能力。
- InnoDB在修改数据时，如果数据的页存在BufferPool中，会修改缓存池中的数据，会产生脏页，InnoDB定期会将这些脏页刷入磁盘，这样可以尽量减少I/O操作，提升性能。
- 通过LRU算法来管理这些缓冲页。为了管理这些数据，innodb使用了一些链表。lru链表：用来存放内存中的缓存数据。free链表：用来存放所有的空闲页，每次需要数据页存储数据的时候，就首先检测free中有没有空闲的页来分配。flush链表：在内存中被修改但还没有刷新到磁盘的数据页列表，也就是所谓的脏页列表。

怎么实现的缓存

- 预读：线性预读和随机预读

缓冲池污染

- 当某一个sql语句，要扫描大量数据时，可能导致把缓冲池的所有页都替换出去，导致大量热点数据被换出，MySQL性能急剧下降，这种情况叫缓存池污染。

解决方案

- 基于对LRU方法的优化，mysql设计了冷热数据分离的处理方案，将lru链表分为冷数据区和热数据区两部分。
- 当数据页第一次被加载到缓冲池中的时候，先将其放到冷数据区的链表头部，1s(参数可调)后该缓存页被再次访问了再将其移至热数据区域的链表头部。
- 当数据页已经在热缓冲区中，当热数据区的后 3/4 部分被访问到才将其移动到链表头部，对于前 1/4 部分的缓存页被访问了不会进行移动。

缓存池刷新策略

1. redo log满时
2. 内存不足需要淘汰数据页
3. 系统空闲时后台会定期flush适量的脏页到磁盘中。
4. MySQL正常关闭时会把所有的脏页都flush到磁盘。

日志文件

undo日志文件

记录数据修改前的样子。(原子性)

redo日志文件

记录数据被修改后的样子。(持久性) redo日志文件是InnoDB特有的，他是存储引擎级别的，不是MySQL级别的。

bin log日志文件

记录整个操作过程 bin log属于MySQL级别的日志，redo log记录的东西偏向于物理性质。

bin log与redo log的比较

1. redo log大小是固定的，bin log可通过参数max_bin_log_size来设置每个bin log文件的大小。
2. redo log属于InnoDB特有的，记录的是在具体某个数据页上做了什么修改。而bin log是MySQL层实现的，任何的引擎都可以使用bin log文件。记录的是这个语句的原始逻辑。
3. redo log采用循环写的方式，当写到结尾的时候，会回到开头循环写日志。bin log采用追加的方式，超过文件大小，后续的日志会记录到新的文件上。
4. redo log适合来做崩溃恢复。bin log适用于主从复制和数据恢复。
5. bin log存储修改的数据，同时本次修改的bin log文件名和修改的内容在bin log中的位置记录到redo log中。在redo log最后写入commit标记。

故障情况

- 如果在数据被写入bin log文件的时候，系统宕机了，首先可以确定的是只要redo log最后没有commit标记，MySQL就会认为事务是失败的，但是数据没有丢失，因为已经记录到redo log磁盘文件中了。下次MySQL重启的时候将redo log中的数据恢复到BufferPool中。

- 如果在将更新的数据记录到redo log buffer中的时候，服务器宕机了，缓存池中的数据丢失了，MySQL会认为本次事务是失败的，数据恢复到更新前的样子。
- 如果redo log buffer刷入磁盘后，数据库服务器宕机了，此时redo log buffer中的数据已经被写入到磁盘，被持久化，在下次重启MySQL也会将redo日志文件中的内容恢复到Buffer pool中。

优化

大表优化

大表带来的问题

1. 慢查询，很难在短时间内过滤出需要的数据。查询的数据区分度低，很难在大量的数据中筛选出来，筛选过程中会产生大量的磁盘IO，降低磁盘效率。
2. 对DDL的影响：建立索引需要很长时间，修改表结构需要长时间的锁表。

解决方案

1. 限定数据的范围
2. 读写分离
3. 垂直拆分
4. 水平拆分(水平拆分涉及的逻辑比较复杂，两类解决方案，客户端结构(中小型)，代理结构(大型))

大事务

运行时间长，操作数据比较多的事务

风险：

1. 锁定数据太多。会造成大量的阻塞和锁超时
2. 回滚时间长。
3. 执行时间长。将造成主从延迟，只有当服务器全部执行完写入日志时，从服务器才开始进行同步，造成延时。

解决思路：

4. 避免一次处理太多数据，分批次处理。
5. 移除不必要的select操作，保证事务中只有必要的写操作。

一个网页打开缓慢，说说优化思路

- 大多数情况很正常，偶尔这样，可能是数据库在刷脏页(redo log写满了需要同步到磁盘)，执行的时候遇到锁，行锁，表锁
- 没有用上索引，例如字段没有索引；由于对字段进行运算、函数操作导致无法用索引。数据库选错了索引。

MySQL存储过程

MySQL的可重复读怎么实现的

- 使用MVCC实现，InnoDB在每行记录后面保存两个隐藏的列，分别保存了数据行的创建版本号和删除版本号。每开启一个事务，系统版本号都会递增。事务开始的时刻的系统版本号作为事务的版本号。

1. SELECT：必须满足两个条件才能查询。(1)版本号大于当前版本号的数据行。(2)行的删除版本号要么未定义，要么大于当前事务版本号。
2. INSERT：当前系统版本号作为创建版本号。
3. DELETE：删除的数据行当前系统版本号作为删除版本号。
4. UPDATE：插入新一行的数据，保存当前版本号作为创建版本号，当前版本号作为原来数据行删除版本号。

MVCC(多版本并发控制)

- InnoDB会在每行记录后面增加三个隐藏字段：DB_ROW_ID，DB_TRX_ID，DB_ROLL_PTR

1. DB_ROW_ID：行ID，随着行的加入而递增，如果有主键，则不包含该列
2. DB_TRX_ID：记录插入或更新该行的事务ID
3. DB_ROLL_PTR：回滚指针，指向undo log记录。每次对某条记录进行修改，该列后面都会增加一个指针，通过这个指针可以找到该记录被修改之前的信息。当某条记录被多次修改的时候，该行记录会存在多个版本，通过DB_ROLL_PTR链接会形成一个类似版本链的概念。

- 以RR级别为标准，每开启一个事务时，系统给这个事务分配一个事务ID，**当该事务执行第一个select语句的时候，会生成当前时间点的事务快照ReadView**，主要包含：
    - trx_ids：生成ReadView的时候当前活跃的事务id列表，也就是未执行提交事务的；
    - up_limit_id：低水位，取当前事务id列表中最小的id；trx_id小于该值都能看到
    - low_limit_id：高水位，生成ReadView时系统将要分配给下一个事务的ID，trx_id大于等于该值都不能看到。
    - creator_trx_id：创建该ReadView的事务的事务id。

- 在这种情况下：
    1. 如果一个事务的id等于creator_trx_id，说明当前事务正在访问自己修改过的记录，所以该版本可以被当前事务访问。
    2. 如果被访问版本的trx_id小于ReadView中的up_limit_id的值，意味着访问该版本时，生成该版本的事务在创建ReadView前已经提交，所以该版本可以被访问到。
    3. 如果被访问的版本的trx_id大于ReadView中的low_limit_id的值，说明生成该版本的事务在创建ReadView之后才开启，所以该版本不能被当前事务访问。
    4. 如果被访问版本的trx_id在ReadView的up_limit_id和low_limit_id之间的话，则判断trx_id是否在trx_ids中。如果在，说明创建该版本的事务还是活跃的，该版本不能被访问；如果不在，说明创建该版本的事务已经提交，可以访问。
    - 在进行判断的时候，总是会拿最新的版本来比较，如果该版本无法被当前事务看到，则通过记录的回滚指针找到上一个版本，重新进行比较，直到找到一个能被当前事务看到的版本。
    - 对于删除，其实是一种特殊的更新，InnoDB会使用delete_bit这个标记为表示当前版本是否删除，在进行判断的时候，会检查delete_bit是否已经被删除，如果已经删除了，则跳过该版本，寻找上一个版本。

MVCC解决幻读了没有?

- 对于快照读来说，也就是select，MVCC是从ReadView中读取的，不会看到新插入的行，所以就解决了幻读。
- 对于当前读来说，也就是update/insert/delete等，是无法解决的，需要通过引入Gap锁或者Next—Key lock来解决幻读。
- SQL规定中的RR并不能解决幻读，但是MySQL的RR是可以解决幻读的，因为MySQL的RR级别下，Gap锁默认是开启的，在RC级别下，默认是关闭的。

union和union all的区别

- union all：对两个结果直接进行并集操作，记录可能会有重复，不会进行排序。
- union：对两个结果集进行并集操作，会去重，按照字段的默认规则进行排序。

InnoDB的四大特性

- 插入缓冲(change buffer)：索引是存储在磁盘上的。对于插入主键索引来说，不需要磁盘的随机I/O，只需要不断的追加即可。但是对于辅助索引来说，辅助索引，大概率是无序的，这时候需要用到磁盘的随机I/O，而随机I/O的性能会很差。InnoDB涉及插入缓冲来减少随机I/O的次数。对于非聚集索引的插入或更新操作，不是每一次操作都直接插入到索引页中，而是先判断插入的非聚集索引是否在缓冲池中，如果在直接插入，如果不在，则先放到一个Insert Buffer中，然后再按照一定的频率将Insert Buffer和辅助索引叶子节点进行合并操作，通常是将插入操作合并在一起，大大提高了对于非聚集索引的插入性能。
- 二次写：脏数据刷盘风险，**InnoDB的page size一般是16k，而操作系统写文件通常以4KB为单位**。如果在刷盘过程中服务器宕机，那么只有一部分是成功的，这就是部分页写入问题，会出现数据不完整问题。这时候是不能够使用redo log来进行恢复的，因为redo log记录的是对页的物理修改，如果页本身已经损坏，重做日志也是不行的。为了解决这个问题，设计了doublewrite，doublewrite分为两部分，一部分是内存中的doublewrite buffer，大小为2MB，另一部分是磁盘上共享表空间中连续的128页，也就是两个分区，也是2MB。具体操作是：先将脏数据复制到doublewrite buffer中，然后通过doublewrite buffer再分两次，每次1MB的方式将数据写入磁盘上共享表空间中。完成写入后，调用fsync操作，将doublewrite buffer中的数据写入实际的各个表空间中。如果在写入磁盘的过程中，发生宕机，可以在共享表空间中找到最近写入磁盘页的副本，用来进行数据恢复。
- 自适应哈希索引：InnoDB本身是不支持哈希索引的，但是如果观察到某些索引被频繁的访问到，索引成为热数据，通过建立哈希索引是可以提升查询速度的。
- 预读：则是利用空间局部性原理，当某些页很快要被用到时，会异步的将这些页提前读取到缓冲池中。InnoDB提供两种预读算法来提升I/O性能：**线性预读和随机预读**。线性预读关注的是extent(64个page，分区)，如果extent中被顺序读取的page超过或等于设定的阈值，则会将下一个extent预读到缓冲区中。随机预读关注的是page，如果一个extent中的某些page在缓冲区中被发现时，随机预读会将extent中的剩余page预读到缓冲区中。

change buffer

- 读多写少用change buffer，反之不要使用。
- 为什么写缓存优化仅适用于非唯一普通索引页呢？：如果索引设置了唯一属性，在进行修改操作的时候，必须进行唯一性检查，就必须将页读入内存中才能判断，也就没有必要用change buffer。
- 当需要更新一个数据页的时候，如果数据页在内存中就直接更新，如果不在内存中，在不影响数据一致性的前提下，InnoDB会将这些操作缓存到change buffer中，这样就不需要在磁盘中读取这个页。当下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行change buffer中与这个页相关的操作。通过这种方式就能保证这个数据逻辑的正确性。

- 将change buffer中的操作应用到原始数据页上，得到最新结果的过程叫做`merge`，除了这个数据页会定期触发merge外，系统有后台线程会定期merge。在数据库正常关闭的过程中，也会执行merge操作。
- 如果能够将更新操作先记录到change buffer，减少读磁盘，语句的执行速度会得到明显的提升。而且数据读入内存是需要占buffer pool的，所以这种方式还能避免占用内存，提高内存利用率。

MySQL调优

1. sql语句的调优
    1. 不要使用隐式类型转换
    2. 字段不要默认值为NULL
    3. 负向查询不能使用索引
    4. 前导模糊查询不能使用索引
    5. 索引字段不要作为表达式或者函数的参数
    6. 最左前缀匹配原则
    7. 如果明确只有一条记录返回，limit 1
    8. join两表字段要相同
2. MySQL的优化
    1. 使用Explain进行分析。
        1. type: 表示访问类型，性能由差到好为：ALL 全表扫描、index 索引全扫描、range 索引范围扫描、ref 返回匹配某个单独值得所有行，常见于使用非唯一索引或唯一索引的非唯一前缀进行的查找，也经常出现在 join 操作中、eq_ref 唯一性索引扫描，对于每个索引键只有一条记录与之匹配、const 当 MySQL 对查询某部分进行优化，并转为一个常量时，使用这些访问类型，例如将主键或唯一索引置于 WHERE 列表就能将该查询转为一个 const、system 表中只有一行数据或空表，只能用于 MyISAM 和 Memory 表、NULL 执行时不用访问表或索引就能得到结果。SQL 性能优化的目标：至少要达到 range 级别，要求是 ref 级别，如果可以是consts最好。
        2. key: 显示 MySQL 在查询时实际使用的索引，如果没有使用则显示为 NULL。
        3. ref: 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值。
        4. rows: 表示 MySQL 根据表统计信息及索引选用情况，估算找到所需记录所需要读取的行数。
        5. Extra: 表示额外信息，例如 Using temporary 表示需要使用临时表存储结果集，常见于排序和分组查询。Using filesort 表示无法利用索引完成的文件排序，这是 ORDER BY 的结果，可以通过合适的索引改进性能。Using index 表示只需要使用索引就可以满足查询表得要求，说明表正在使用覆盖索引。

分表分库

水平切分 Sharding

当一个表的数据不断增加时，Sharding是必然的结果，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

垂直切分

将一张表按列切分成多个表，通常是按照列的关系密集程度来切分，也可以将经常使用到的列和不经常使用到的列切分到不同的表中。主键出现冗余，需要管理冗余列，并会引起JOIN操作。垂直分区让事务变得更加复杂。

Sharding策略

- 哈希取模
- 范围：可以使ID范围也可以是时间范围
- 映射表：使用单独的一个数据库来存储映射关系。

Sharding存在的问题及解决方案

- 事务问题，分片事务一致性难以解决：使用分布式事务来解决，如XA接口。
- 跨节点JOIN性能差，逻辑复杂：可以将原来的JOIN分解成多个单表查询，然后在用户程序中进行JOIN。
- ID唯一性：(1)使用全局唯一ID：UUID。(2)为每个分片指定一个ID范围。(3)分布式ID生成器

分库分表后，id主键如何处理

- UUID：不适合做主键，太长了，无序不可读，查询效率低，适合用于生成唯一名字的标示，比如文件名字。
- 数据库自增ID：两台数据库设置不同步长，生成的id有序。
- 利用redis生成id。
- 美团的Leaf分布式ID生成系统。

主从复制与读写分离

主从复制

主要涉及三个线程：binlog线程、I/O线程、SQL线程

- binlog线程：负责将主服务器上的数据更改写入二进制日志中。
- I/O线程：负责从主服务器上读取二进制日志，并写入从服务器的中继日志中。
- SQL线程：负责读取中继日志并重放其中的SQL语句。

主从复制的用途

- 实时灾备，用于故障切换。
- 读写分离，提供查询服务。
- 备份，避免影响业务。

主从复制采用异步复制，主机宕机后，数据可能丢失?

- 采用半同步复制或全同步复制。
- 半同步复制：修改语句写入bin log后，不会立即给客户端返回结果，而是首先通过log dump线程将bin log发送给从节点，从节点的I/O线程在将bin log写入relay log后，返回ACK给主节点，主节点然后返回给客户端成功。
- 全同步复制：主节点和所有从节点全部执行了该事务并确认才会向客户端返回成功。

主库写压力大，从库复制可能出现延迟?

主从延迟的原因

- 一个服务器开放N个链接给客户端来链接，这样会有大并发的更新操作，但是从服务器的里面读取binlog的线程仅有一个，当某个SQL在从服务器上执行的时间稍长或者由于某个SQL要进行锁表就会导致，主服务器的SQL大量积压，未被同步到从服务器里。这就导致了主从不一致，也就是主从延迟。
- 可以使用并行复制(并行是指从库多个SQL线程并行执行relay log)，解决从库复制延迟的问题

读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

- 读写分离能提高性能的原因在于
    - 主从服务器负责各自的读和写，极大程度缓解了锁的争用。
    - 从服务器可以使用MyISAM，提升查询性能以及节约系统开销。
    - 增加冗余，提高可靠性。 读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

Redis

Redis数据结构

为什么使用缓存

- 使用缓存是为了提高用户体验以及应对更多的用户
- 高性能、高并发

Redis是什么

- Redis是C语言开发的开源的高性能键值对的内存数据库，可以用作数据库、缓存、消息中间件等。端口号6379
- 它是一种非关系型数据库，性能优秀，单进程单线程，是线程安全的，采用IO多路复用机制，丰富的数据类型，支持string、list、hash、set、sorted sets等。支持数据持久化，可以将内存中数据保存到磁盘中，重启时加载。主从复制、哨兵、高可用。可以用作分布式锁，可以作为消息中间件使用，支持发布订阅。

Redis是单线程还是多线程

- Redis4.0之前，redis是完全的单线程，这里的单线程指的是与客户端交互完成命令请求和回复的工作线程。
- Redis4.0引入了多线程，但是额外的线程只能用于后台处理，记录删除对象，异步删除、集群数据同步，核心流程还是完全单线程的。(核心线程指的是Redis正常处理客户端请求的流程，包括接受命令，解析命令，执行命令，返回结果等。**Redis的网络IO和键值对读写是由一个线程来完成的**，这也是Redis对外提供键值存储服务的主要流程)
- Redis6.0 这次引入的多线程的概念会涉及到核心线程。**多线程主要用于网络I/O阶段，也就是接收命令和写回结果阶段，而在执行命令阶段，还是由单线程串行执行。**(Redis中的多线程组不会同时存在读和写，这个多线程组只会同时读或者同时写)

Redis6.0加入多线程I/O之后，处理命令的核心流程如下：

- 当有读事件到来时，主线程将客户端连接放到全局等待读队列

- 读取数据：1）主线程将等待读队列的客户端连接通过轮询调度算法分配给I/O线程处理；2）同时主线程也会自己负责处理一个客户端连接的读事件。3）当主线程处理完该连接的读事件后，会自旋等到所有I/O线程处理完毕
- 命令执行：**主线程按照事件被加入全局等待读队列的顺序，串行执行客户端命令，然后将客户端连接放到全局等待写队列。**这里是单线程的，也就是主线程的。
- 写回结果：跟等待读队列处理类似，将等待写队列的客户端连接放入到全局等待写队列中，然后将等待读队列的客户端连接通过轮询调度算法分配给I/O线程处理，同时自己也会处理一个。当主线程处理完毕之后，会自旋等待I/O线程处理了完毕，然后清空队列。

为什么Redis是单线程

- redis是完全基于内存操作的，通常情况下CPU不会成为redis的瓶颈，redis的瓶颈最可能是机器内存的大小或网络带宽。既然CPU不会成为瓶颈，就使用单线程就好了，使用多线程会更复杂，同时需要引入上下文切换、加锁等等，会带来额外的性能消耗。单线程编程容易并且更容易维护。
- 6.0版本对核心流程引入了多线程，主要用于解决redis在网络I/O上的性能瓶颈，而对核心命令的执行，还是单线程。

Redis为什么使用单线程、单线程也很快

- 基于内存的操作
- 使用I/O多路复用模型，select，epoll等，基于reactor模式开发了自己的网络事件处理器
- 单线程避免了不必要的上下文切换和竞争条件，减少了这方面的性能消耗。
- 以上三点是主要原因，还有一些小优化，对数据结构进行了优化，简单动态字符串、压缩列表等。

Redis在项目中的使用场景

- 缓存、分布式锁、排行榜(zset)、计数(incrby)、消息队列(stream)、地理位置(geo)、访客统计(hyperloglog)等。

Redis常见的数据结构

- String(简单动态数组SDS)、List、Hash、Set、Sorted Set：有序集合，Set的基础上加了个分值
- Hyperloglog，通常用于基数统计。使用少量固定大小的内存，来统计集合中唯一的元素的数量。统计结果不是精确值，适用于对于统计结果精度要求不是特别高的场景，例如网站的UV统计。
- Geo：可以将用户给定的地理位置信息存储起来，并对这些信息进行操作：获取2个位置的距离，根据给定地理位置坐标获取指定范围内的地理位置集合。
- Bitmap：位图
- Stream：主要用于消息队列，提供消息的持久化和主备复制功能，可以让任何客户端访问任何时刻的数据，并且能记住每一个客户端的访问位置，还能保证消息不丢失。

为什么会设计redisObject对象

- 类型的命令检查和多态
- 对象共享

SDS的底层实现结构

raw 和 embstr 的区别

- embstr编码是专门保存短字符串的一种优化编码。
- embstr和raw都使用redisObject和sds保存数据，区别在于，embstr的使用只分配一次内存空间(embstr和redisObject是连续的)，而raw需要分配两次内存空间(分别为redisObject和sds分配空间)。因此与raw相比，embstr的好处在于创建时少分配一次内存空间，删除时少释放一次内存空间，以及对象的所有数据连在一起，寻找方便。而embstr的坏处也很明显，如果字符串的长度增加需要分配内存时，整个redisObject和sds都需要重新分配空间，因此redis中的embstr实现为只读。

Redis的字符串(SDS)和C语言的字符串的区别

- 获取字符串长度的复杂度不同 C语言获取长度的复杂度是O(n)，redis是O(1)
- API安全级别不同 C语言API是不安全的，可能会造成缓冲区溢出，SDS的API是安全的，不会造成缓冲区溢出
- 修改字符串需要的内存重分配次数不同 C语言修改字符串N次必须需要N次内存重分配，SDS最多需要N次内存重分配，SDS采用空间预分配和惰性空间释放两种策略。
- 保存数据的格式不同 C语言只能保存文本数据，SDS可以保存文本数据和二进制数据
- 对于string.h库的使用不同 C语言可以使用所有的库函数，SDS只能使用一部分库中的函数

Sorted Set底层数据结构

- Sorted Set(有序集合)：ziplist/(skiplist+ht)
- ziplist：使用压缩列表实现，当保存的元素都小于64字节，同时数量小于128时，使用该编码方式，否则使用(skiplist+ht)。这两个参数可以通过zset-max-ziplist-enties、zset-max-ziplist-value来自定义修改。
- skiplist：zset实现，一个zset同时包含一个字典和一个跳跃表。

Sorted Set为什么同时使用字典和跳跃表

- 主要是为了性能。
- 单独使用字典：在执行范围内操作，如zrank、zrange，字典需要排序，至少需要O(nlogn)的时间复杂度和额外的O(N)的内存空间。
- 单独使用跳跃表：根据成员查找分值操作的复杂度从O(1)上升为O(nlogn)。

Sorted Set为什么使用跳跃表而不是红黑树

- 跳跃表更容易实现和调试
- 在范围查找时，平衡树比跳表操作要复杂。平衡树上，在找到指定范围的小值之后，还需要中序遍历继续寻找不超过最大值的节点。
- 平衡树的插入和删除操作可能引起子树的调整，逻辑复杂，而跳表的插入和删除操作只需要修改相邻节点的指针。
- 在内存占用上，平衡树每个节点至少需要2个指针，而跳表每个节点包含的指针数目平均为1/(1-p)，像Redis里实现的一样，p=1/4，平均每个节点包含1.33个指针

Hash的底层实现结构

Hash对象底层结构

- Hash对象有两种编码：ziplist、hashtable
- ziplist：使用压缩列表实现，每当有新的键值对加入到hash对象中时，会先将键的节点先压入压缩列表的表尾，接着将键的值压入压缩列表的表尾。这样的话保存了同一键值对的两个节点总是紧挨在一

起，保存键的节点在前，保存值的节点在后。先进的键值对被放到压缩列表的表头，后进的被放到表尾。
- hashtable：使用字典作为底层实现，哈希对象中的每个键值对都使用一个字典键值来保存。

Hash对象的扩容流程

- Hash对象在扩容的时候使用了一种叫渐进式rehash方式

1. 计算新表size，掩码，为新表ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表。
2. 将rehash索引计数器变量rehashidx的值设为0，表示rehash开始。
3. 在rehash期间，每次对字典执行添加、删除、查找、更新操作时，程序除了执行指定的操作以外，还会出发额外的rehash操作。该方法会从ht[0]表的rehashidx索引位置上开始向后查找，找到第一个不为空的索引位置，然后将该索引位置上的所有节点rehash到ht[1]，当本次rehash工作完成之后，将ht[0]的rehashidx位置清空，同时将rehashidx属性的值加一。
4. 将rehash分摊到每个操作上，redis除了文件事件外，还有时间事件，redis会定期触发时间事件，这些时间事件用于执行一些后台操作，其中就包括rehash操作；当redis发现有字典正在进行rehash操作时，会花费1毫秒的时候，帮助进行rehash。
5. 随着操作的进行，当ht[0]的所有键值对都被rehash到ht[1]，此时rehash流程完成，会执行最后的清理工作：释放ht[0]的空间、将ht[0]指向ht[1]、重置ht[1]、重置rehashidx的值为-1。

渐进式rehash的优点

- 采取分而治之的方式，将rehash键值对所需的计算工作均摊到对字典的每个添加、删除、查找和更新操作上，从而避免了集中式rehash而带来的庞大计算量。
- 在渐进式rehash过程中，字典会同时使用ht[0]和ht[1]两个哈希表，所以在渐进式rehash进行期间，字典的删除、更新、查找等操作会在两个哈希表中进行。
- 在渐进式rehash期间，新增的键值对会被直接保存到ht[1]，ht[0]不再进行任何添加操作，这样就保证了ht[0]包含的键值对数量只减不增，并随着rehash操作的执行最终变成空表。

rehash流程在数据量大的时候会有什么问题

1. 扩容开始期间，会先给ht[1]分配空间，所以在整个扩容期间，会同时存在ht[0]和ht[1]，会占用额外的空间。
2. 扩容期间同时存在ht[0]和ht[1]，查找、删除、更新操作等有概率同时操作两张表，耗时会增加。
3. redis在内存使用接近maxmemory并且有设置驱逐策略的情况下，出现rehash会使得内存占用超过maxmemory，触发驱逐淘汰操作，导致maste/slave均有大量的key被驱逐淘汰，从而出现master/slave主从不一致。

Redis持久化

Redis的核心主流程

- 最重要的两个事件：文件事件和时间事件。Redis在服务器初始化后，会无限循环，处理产生的文件事件和时间事件。
- 文件事件最常见的：接收连接、读取、写入、关闭连接。
- 时间事件常见的是serverCon，Redis默认设置100ms会被触发一次，在该时间事件中，会执行很多操作：清理过期键、AOF后台重写、RDB的save point的检查、将aof_buf内容写入磁盘，rehash

Redis的持久化机制有哪几种

- RDB、AOF、混合持久化

RDB触发方式

手动触发

- save命令和bgsave命令

自动触发

RDB的实现原理、优缺点

- 实现原理：类似于快照，在某个时间点上，**将Redis在内存中的数据库状态(数据库的键值对等信息)保存到磁盘里面**。RDB持久化生成的RDB文件是经过压缩的二进制文件。
- 有两个命令来用于生成RDB文件，SAVE和BGSAVE。使用save point来配置，当满足save point的时候就会触发BGSAVE来生成RDB文件。
- 注释掉所有的save point配置可以关闭RDB持久化。在save point配置后增加save “”，可以删除所有之前的save point。
- SAVE：生成RDB快照文件，但是会阻塞主线程，服务器无法处理客户端发来的命令请求。BGSAVE：fork子线程来生成RDB快照文件，阻塞之后在fork子线程的时候，之后主线程可以正常处理请求。
- RDB优点：
    - RDB文件是经过压缩的二进制文件，占用内存空间很小，它保存了Redis某个时间点的数据集，很适合用于做备份。
    - RDB非常适用于灾后恢复：它只有一个文件，并且内容十分紧凑，可以将它传送到别的数据中心。
    - RDB最大化redis的性能。父进程在保存RDB文件时唯一要做的就是fork一个子进程，然后这个子进程处理接下来的保存工作，父进程无需执行任何的磁盘I/O操作。
    - RDB在恢复大数据集时的速度比AOF的恢复速度要快。
- RDB的缺点：
    - RDB保存的是整个数据集的状态，它是一个比较重的操作，如果操作太频繁，可能会对redis的性能产生很大的影响。
    - RDB保存时使用fork子进程进行数据的持久化，如果数据比较大时，fork可能会非常耗时，造成redis停止处理服务N毫秒。
    - linux fork采用的时候copy-on-write的方式。Redis在执行RDB持久化期间，如果client写入数据频繁，将会增加Redis占用的内存。刚fork的时候，父进程和子进程共享内存，随着父进程处理写操作，主进程需要将修改的页面copy一份出来进行修改操作。极端情况下，如果所有的页面都需要修改，则此时的内存占用是原来的2倍。
    - RDB文件是二进制的，没有可读性，AOF在了解其结构的情况下可以手动修改或者补全。
- **由于生产环境中我们为Redis开辟的内存区域都比较大（例如6GB），那么将内存中的数据同步到硬盘的过程可能就会持续比较长的时间，而实际情况是这段时间Redis服务一般都会收到数据写操作请求。那么如何保证数据一致性呢？** RDB的核心思路是Copy-on-Write，保证在进行快照操作的这段时间，需

要压缩写入磁盘的数据在内存不会发生变化。在正常的快照操作中，主线程会fork一个子线程进行持久化操作，另一方面，在这段时间内发生的数据变化会存放到另一个新的内存区域，待快照结束后才会同步到原来的空间区域中。

- **频繁做全量快照，会带来两方面的消耗：**
    - 给磁盘带来很大的压力。频繁的进行全量快照，多个快照竞争磁盘资源，另一个快照还没做完，另一个又来了。
    - 会频繁阻塞主线程。虽然fork出子线程后不会阻塞主线程，但是fork这个过程本身需要阻塞主线程，而且主线程的内存越大，阻塞的时间越长。

AOF的实现原理、优缺点

- **保存Redis服务器所执行的所有写操作命令来记录数据库状态**。并在数据库重启时，通过重新执行这些命令来还原数据集。
- AOF持久化功能的实现分为三个步骤：**命令追加、文件写入、文件同步。**
- 命令追加：当AOF持久化功能打开时，服务器执行完一个命令后，会将执行的命令追加到服务器状态的缓冲区的末尾。
- 文件写入和文件同步：linux为了提升性能，使用了页缓存(page cache)。当我们将aof缓冲区中的数据写入磁盘时，此时数据并没有真正的落盘，而是在page cache中，为了将页缓存中的数据真正的落盘，需要执行某些命令来执行强制落盘。文件同步就是文件刷盘操作。
- AOF的优点：
    - AOF比RDB可靠。我们在设置刷盘指令时，默认是everysec，在这种配置下，即使服务器宕机，也只是丢失了一秒钟的数据。
    - AOF是纯追加的日志文件。即使日志因为某些原因而包含了未写入完整的命令，也能通过工具轻易的修复这种问题。
    - 当AOF文件太大时，Redis会进行AOF重写，重写后的AOF文件包含了恢复当前数据的最小命令集合。整个重写是安全的，重写是在新文件上进行的，重写完后，Redis会把新旧文件进行替换，开始把数据写到新文件上。
    - AOF文件有序的保存了对数据库执行的所有写入操作以Redis协议的格式保存，十分容易被人读懂和分析。
- AOF的缺点：
    - 对于相同的数据集，AOF文件比RDB文件大。
    - 根据使用的刷盘策略，AOF的速度可能比RDB慢。
    - 因为个别命令的原因，导致AOF文件在重新载入后，无法将数据集恢复到之前的样子。
- **为什么采用写后日志**
    - 不需要检查命令是否语法错误
    - 不会阻塞当前写操作
    - 在命令执行完成，写入AOF之前系统宕机了，会丢失这段时间的数据。
    - 主线程写磁盘压力大，导致写盘慢，阻塞后续操作。

混合持久化的实现原理、优缺点

- 混合持久化并不是一种全新的持久化方式，而是对已有方式的优化。混合持久化只发生于AOF重写过程。使用了混合持久化，重写后的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。
- 优点：结合了RDB和AOF的优点，更快的重写和恢复。
- 缺点：AOF文件里面的RDB不是不再是AOF格式，可读性差。

AOF重写

为什么需要AOF重写

- AOF是通过保存被执行的写命令来记录数据库状态的，随着被执行的命令越来越多，文件体积越来越大，如果不加以控制，体积过大的AOF文件会对Redis服务器、甚至整个宿主机造成影响。并且随着AOF体积的越来越大，恢复数据的时间也会越来越长。

AOF重写

- Redis生成新的AOF文件来替换旧的文件，这个新的AOF文件包含了恢复数据的最少命令集合。具体的过程就是遍历数据库的所有键，从数据库读取现在的值，然后用一条命令去记录键值对，代替之前记录这个键值对的多条命令。(BGREWRITEOF，REWRITEOF)

AOF后台重写存在的问题

- 子进程在进行AOF重写期间，主进程还需要继续处理命令请求，新的命令可能会对现有的数据库状态进行修改，导致当前的数据库状态和重写后的AOF的数据库状态不一致。

如何解决AOF后台重写存在的数据不一致问题

- 引入AOF重写缓冲区，这个缓冲区在创建子线程之后开始使用，当Redis服务器执行读命令后，它会将这个读命令同时追加到**AOF缓冲区和AOF重写缓冲区**。当子线程完成AOF重写后，父进程会将AOF重写缓冲区的内容写入到新的AOF文件中，并对新的AOF文件进行改名，完成新旧文件的替换。这样主进程就能继续接收命令请求了。

AOF重写缓冲区内容过多怎么办

- 因为AOF重写缓冲区的内容追加到新的AOF文件是由主线程完成的，所以当重写缓冲区文件太大时，会造成一段时间的阻塞，这显然是不能接受的。
- 解决方案：在进行AOF后台重写时，**Redis会创建一组用于父子进程间通信的通道，同时会新增一个文件事件**，该文件事件会将写入AOF重写缓冲区的内容通过该管道发送到子进程。在重写结束后，子进程会通过该管道尽量从父进程读取更多的数据。如果连续20次没有读取到则结束这个过程。

主线程fork出子进程是如何复制内存数据的?

- 拷贝父进程的页表，即虚实映射关系(虚拟内存到物理内存的映射索引表)

在重写日志整个过程中，主线程有哪些地方会被阻塞?

- fork子进程的时候，需要拷贝虚拟页表，会对主线程阻塞。
- 主进程有bigkey写入时，操作系统会创建页面的副本，并拷贝原有的数据，会对主线程阻塞。
- 子进程重写完日志后，主进程追加AOF重写缓冲区内容时可能会对主进程阻塞。

为什么AOF重写不复用原AOF日志?

- 父子进程间写同一文件会产生竞争问题，影响父进程的性能。
- 如果AOF重写过程中失败了，相当于污染了原本的AOF文件，无法做恢复数据使用。

RDB、AOF、混合持久，我应该用哪一个

- 如果想尽量保证数据安全性，应该同时使用RDB和AOF持久化功能。如果能够接受分钟内的丢失，可以使用RDB，如果你的数据可以丢失的，可以关闭持久化功能。

从持久化中恢复数据

- redis重启时判断是否开启aof，如果开启了aof，那么优先加载aof文件
- 如果aof文件存在，就去加载aof文件。如果aof文件加载失败，就会打印日志，此时可以去修复aof文件后重新启动。如果加载成功则重启成功。
- 如果aof文件不存在，则转去加载rdb文件，如果rdb文件不存在，redis直接启动成功。
- 如果rdb文件存在就去加载rdb文件恢复数据，如果加载失败提示启动失败。如果加载成功，则重启成功，并使用rdb文件恢复数据。

性能与实践

消息传递

事件机制

aeEventLoop

事件的调度与执行

文件事件

Redis的事件处理器

- 由4部分组成：套接字、I/O多路复用程序、文件事件分派器以及事件处理器
- 套接字：socket连接，也就是客户端连接。当一个套接字准备好执行连接、写入、读取、关闭等操作时，就会产生一个相应的文件事件。因为一个服务器可能连接多个套接字，所以多个文件事件有可能并发地出现。
- I/O多路复用程序：提供select、epoll、evport、kqueue的实现，会根据当前系统自动选择最佳的方式。负责监听多个套接字，当套接字产生事件后，会向文件事件分派器传送那些产生了事件的套接字。
- 当多个文件事件并发的出现时，I/O多路复用程序会将所有产生事件的套接字都放到一个队列里，然后通过这个队列有序、同步、每次以一个套接字的方式向文件事件分派器传送套接字：当上一个套接字产生的事件被处理完毕后，才会继续传送下一个套接字。
- 文件事件分派器：接收I/O多路复用程序传来的套接字，并根据套接字产生的事件类型，调用相应的事件处理器。
- 事件处理器：就是一个个函数，定一个某个事件发生时，服务器应该执行的动作。例如：建立连接、命令查询、命令写入、连接关闭等。

时间事件

Reactor模式

Redis事务

什么是Redis事务

Redis事务本质上是一组命令的集合。事务支持一次执行多个命令，一个事物中所有命令都会被序列化。在事务执行过程中，会按照顺序串行执行队列中的命令，其他客户端提交的命令请求不会插入到事务执行的命令序列中。

CAS操作实现乐观锁

- WATCH可以为Redis事务提供CAS行为。
- Redis使用WATCH命令来决定事务是继续执行或者是回滚。在MULTI命令之前WATCH某些键值对，然后使用MULTI开启事务，执行对数据结构的各种命令，将这些命令入队。当执行EXEC时，会比较WATCH监控的键值对是否变化，没有发生变化执行队列中的命令，发生变化不执行队列中的命令，事务回滚。无论是否回滚，Redis都会取消事务执行前的WATCH命令。

Redis事务执行步骤

- 开启：使用MULTI命令开启事务。
- 入队：将多个命令入队到事务中，这些命令不会立即执行，而是加入到等待执行的事务队列里面。
- 执行：有EXEC命令触发事务。

Redis为什么不支持回滚

- Redis命令只会对错误的语法导致的失败或者是命令用到了错误类型的键上，这些错误是变成错误造成的，应该在开发过程中被发现，而不是应该出现在生产环境中。
- 不需要对回滚进行支持，Redis内部保持简单和快捷。

如何理解Redis与事务的ACID?

- ACID：原子性、一致性、隔离性、持久性

Redis主从复制

主从复制的作用

- 数据冗余
- 故障恢复
- 负载均衡：主服务器提供写服务，从服务器提供读服务，用于读多写少的环境。
- 高可用基石

主从库之间采用读写分离的方式：

- 读操作：主服务器和从服务器都可以接收
- 写操作：首先在主服务器执行，接着主库将写操作同步给从服务器。

主从复制实现原理

1. 开启主从复制
    - 命令：replicaof <masterip> <masterport> 或者 配置文件 replicaof <masterip> <masterport>

2. 建立套接字连接
    - slave根据ip和端口向master发送套接字连接，master在接受套接字连接后，创建相应的客户端状态。
3. 发送PING命令
    - slave发送ping命令，以检查套接字的读写状态是否正常。
4. 身份验证
    - 如果master和slave都设置密码，则验证，都没有设置密码则不需要验证，一个设置一个不设置返回错误。
5. 发送端口信息
    - slave向master发送自己的监听端口，master收到后记录在slave所对应的客户端状态的slave_listening_port属性中。
6. 发送IP地址
    - slave向master发送slave_announce_ip配置的ip地址，master收到记录后记录在slave所对应的客户端状态的slave_ip属性。
7. 发送CAPA
    - slave发送capa告诉master自己的同步复制能力，master收到后记录在slave对应的客户端状态的slave_capa属性。
8. 数据同步
    - slave向master发送PSYNC命令，master收到该命令后判断是进行部分重同步还是完整重同步，然后根据策略进行数据的同步。
    - slave如果是第一次执行复制，会发送PSYNC？ -1，master返回+FULLRESYNC <replid> <offset>执行完整重同步。
    - 如果不是第一次执行，slave则发送PSYNC replid offset，其中replid是master的复制ID，offset是slave当前的复制偏移量。master根据replid和offset来判断应该执行哪种同步操作。如果是完整重同步，返回+FULLRESYNC <replid> <offset>；如果是部分重同步，则返回+CONTINUE <replid>，此时slave只需要等待master将自己缺少的数据发送过来就可以了。
9. 命令传播
    - 当完成了同步之后，就进入命令传播阶段，master会将自己执行的写命令一直发送给slave，slave负责一直接收命令。这样就能保证master和slave的一致性了。
    - 在命令传播阶段，slave默认每秒一次的频率向master发送命令：REPLCONF ACK <reploff>。
    - 发送REPLCONF有三个作用：1. 检测master与slave之间的网络状态。2. 汇报自己的偏移量，检测命令丢失。3. 辅助实现min-slaves配置，用于防止master在不安全的情况下执行命令。

旧版同步：SYNC

- Redis2.8之前的数据同步通过SYNC命令来实现
- slave向master发送SYNC命令，master收到命令后执行BGSAVE命令，fork子进程生成RDB文件，同时会有一个缓冲区记录当前开始所有的写命令。当BGSAVE执行完成后，master将生成的RDB文件发送给slave，slave接收RDB文件并载入到内存，将数据库状态更新到master执行BGSAVE之前的状态。当master进行命令传播时，将缓冲区中的写命令发送给slave，还会同时写进复制积压缓冲区。当slave重连上master时，会将自己的复制偏移量通过PSYNC发送给master，master通过与复制积压缓冲区进行比较，如果发现部分未同步的命令还在复制积压缓冲区，则将这部分命令进行部分同步，如果重连时间太久，这部分命令已经不在复制积压缓冲区，则将进行全同步。

运行ID(runid)

- 每个Redis Server都会有自己的运行ID，当slave初次复制master时，master会将自己的runid发送给slave进行保存，之后slave再次进行复制的时候就会讲该runid发送给master，master通过比较runid来判断是不是同一个master。
- 引入runid后，数据同步过程变为slave通过PSYNC runid offset命令，将正在复制的runid和offset发送给master，master判断runid与自己的runid是否相等，如果相等，并且offset还在复制积压缓冲区，进行部分重同步。否则，如果runid不想等或者offset已经不在复制积压缓冲区，执行完全重同步。

PSYNC存在的问题

- slave重启，runid和offset都会丢失，需要进行完全重同步。
- redis发生故障切换，故障切换后master runid发生了变化，slave需要进行完全重同步。

PSYNC2

- 引入两组replid和offset替换原来的runid和offset

1. 第一组replid和offset，对于master来说，表示为自己的复制ID和复制偏移量，对于slave来说，表示为自己正在同步的master的复制ID和复制偏移量。
2. 第二组replid和offset，对于master和slave来说，都表示为自己上一个master的复制ID来复制偏移量；主要用于切换时支持部分重同步。

- slave也会开启复制积压缓冲区，主要用于故障切换后，slave代替master，该slave仍可以通过复制积压缓冲区来继续支持部分重同步，否则无法支持部分重同步。

PSYNC2优化场景

1. slave重启后导致完整同步，原因是重启后复制ID和复制偏移量都丢失了，解决方法在关闭服务器之前将这两个变量存下来。
2. master故障切换后导致完整重同步：原因是master发生故障后，出现了新的master，而新的master的复制ID也发生了变化，导致无法进行部分重同步。解决方法，将新的复制ID和复制偏移量与老的复制ID和复制偏移量串联起来。slave在晋升为master后，将自己保存的第一组复制ID和偏移量移动到第二组，第一组生成属于自己的复制ID。这样新master通过第二组id判断slave是否是自己之前的master，如果是尝试进行部分重同步。

主从复制会存在哪些问题

- 一旦主节点宕机，从节点晋升为主节点，同时需要修改应用方的主节点地址，还需要命令所有从节点去复制新的主节点，整个过程需要人工干预。
- 主节点的写能力/存储能力受到单机的限制。

当主服务器不进行持久化时复制的安全性

- 主从复制的时候，主服务器应该开启持久化，当不能这么做时，必须考虑到延迟的问题，应该将实例配置为避免自动重启。

为什么主从全量复制使用RDB而不使用AOF?

1. RDB文件是经过压缩的二进制数据，文件很小。而AOF文件记录的是每一个写操作的命令，写操作越多文件就会变得越大，其中还包括多次对同一个key的冗余操作。在主从全量数据同步的时候，传输RDB

可以尽量降低对主库网络带宽的消耗。二是因为RDB文件是二进制数据，从库读取数据会很快，而AOF需要每次重放写命令，恢复速度相比RDB慢很多。

2. 假如使用AOF做全量复制，意味着必须打开AOF功能，如果使用不当可能会对Redis服务器的性能造成影响。而RDB只需要在需要定时备份和主从全量复制数据时才会触发一次快照，而在很多丢失数据不敏感的场景中，不需要打开AOF。

为什么还有无磁盘复制模式?

主服务器的子进程直接将RDB通过网络发送给从服务器，不通过磁盘作为中间存储。

为什么还有从库的从库设计?

- 发生一次主从复制服务器需要fork子进程，生成RDB文件，发送RDB文件，十分消耗性能和网络资源，如果很多从服务器进行主从复制会给主服务器带来很大压力。采用主-从-从的设计，在部署主从集群的时候，可以手动选择一个从库，用于级联其他的从库。然后可以再选择一些从库让他们和之前选的从库建立起主从关系。

读写分离及其中的问题

- 读写不一致的问题
- 数据过期问题：惰性删除和定期删除
- 故障切换问题

Redis哨兵机制

哨兵

- 主从复制存在的问题可以用哨兵机制来解决。使用Sentinel来完成节点选举工作。
- 哨兵用于监控Redis集群中Master主服务器的工作状态，可以完成Master和Slave的主从转换。

系统可以执行四个任务：

1. 监控：不断检查主服务器和从服务器是否正常运行。
2. 通知：当被监控的某个Redis服务器出现问题，Sentinel通过API脚本向管理员或其他应用发出通知。
3. 自动故障转移：当主节点不能正常工作时，Sentinel会开始一次自动的故障转移。会将失效主节点的从节点中选择一个成为主节点，并将其他从节点指向新的主节点。
4. 配置提供者：在Redis Sentinel模式下，客户端应用在初始化的时候是与Sentinel节点集合连接，从而获取主节点信息。

哨兵集群的组建

哨兵监控Redis库

哨兵的选举机制

工作原理：

1. 每个Sentinel节点每秒一次的频率向它所知的主服务器，从服务器以及其他Sentinel节点发送PING命令

2. 当一个实例距离有效恢复PING命令的有效时间超过down-after-milliseconds所指定的值时，该Sentinel节点标记为主观下线。
3. 如果一个主服务器被标记为主观下线后，那么正在监视的其他Sentinel节点也要以每秒一次的频率确认主服务进入了主观下线状态。
4. 这时有足够数量的Sentinel节点在指定的时间范围内确认该服务器处于主观下线状态，主服务器被标记为客观下线状态。
5. 一般情况下每个Sentinel以每十秒一次的频率向主服务器和从服务器发送INFO命令，当一个主服务器被指定为客观下线时，频率会改为一秒一次。
6. Sentinel协商客观下线的主节点的状态，如果处于SDOWN状态，则投票出新的主节点，并将从节点指向新的主节点进行数据复制。
7. 如果没有足够数量的Sentinel同意主节点下线，则主节点的客观下线状态被移除。当主服务器重新向Sentinel命令有效回复时，主服务器的主观下线状态被移除。

缓存

缓存穿透

- 大量请求的数据根本不在缓存和数据库中，导致大量请求不经过缓存，直接到了数据库，失去了缓存的意义。
- 解决方案：
    1. 参数校验：一些不合法的请求直接抛出异常返回给服务器。
    2. 缓存无效key，当缓存和数据库都不存在该key的数据时，在Redis中写入该key，并设置一定的过期时间。这种方式适合变化不频繁的key的情况。如果key变化频繁会导致Redis中存在大量无效的key。
    3. 布隆过滤器：可以非常方便的判断一个给定的数据是否存在于海量数据中。具体做法是把所有可能存在的请求的值放到过滤器中，用户请求过来的事情，首先判断用户发来的请求的值是否存在于过滤器中，如果不存在直接返回异常结果给客户端。

缓存击穿

- 缓存中没有但数据库中有的数据(一般是缓存时间到期)，这时由于并发用户特别多，同时读缓存没有读到数据，又同时去数据库取数据，引起数据库压力瞬间增大，造成过大压力。
- 解决方案：
    1. 设置热点数据永不过期
    2. **接口限流与熔断，降级**。重要的接口一定要做好限流策略，防止用户恶意刷接口，同时要降级准备，当接口中的某些服务不可用时，进行熔断，失败快速返回机制。
    3. 加互斥锁。

缓存雪崩

- 缓存在同一时间内大面积的失效，导致大量的用户请求直接到了数据库，造成数据库短时间内承受大量的请求。
- 解决方案：
    - 针对Redis服务不可用的情况：(1)设置Redis集群，避免单机出现问题导致整个缓存服务都没办法使用。(2)限流，避免同时处理大量请求。
    - 针对热点缓存失效问题：(1)随机设置热点缓存失效时间.(2)热点数据永不失效。

缓存污染

- 缓存中一些只被访问一次或者几次的数据，被访问后再也不会被访问到，但这部分数据依然在缓存中，消耗缓存空间。
- 缓存空间是有限的，如果缓存空间满了，再往缓存里写数据时就会有额外的开销，影响Redis的性能。这部分额外性能消耗主要是指写的时候判断淘汰策略，根据淘汰策略去选择要淘汰的数据，然后进行删除操作。

最大缓存设置多大

- 设置为总数据量的15%到30%，兼顾访问性能和内存空间开销。

如何保证redis中的数据都是热点数据?

redis内存数据集达到一定大小的时候，就会实行数据淘汰策略。

1. redis的过期策略：定期删除+惰性删除

- 定期删除：每隔100ms就随机抽取一些设置过期时间的数据，检查是否过期，如果过期就删除。注意：这里不是每隔100ms就遍历所有设置过期时间的key，那样就是一场性能上的灾难。实际上redis是每隔100ms就随机抽取一些key进行检查和删除的。可能会存在过多过期key到了过期时间还没有被删除，这时候就需要惰性删除。
- 惰性删除：当获取某个key的时候，redis会检查一下，如果该key过期就删除，不会返回任何结果。 但是定期删除漏掉了很多的过期key，然后也没有定期去做查询，也就没走惰性删除，这样就导致大量过期的key存放在内存中，这种情况就需要内存淘汰策略。

2. redis的内存淘汰策略

- 默认是不删除，当内存不够时，所有引起申请内存的命令都会报错。
- 对于最近未使用或者使用比较少的键尝试回收。(对于所有键)
- 对于最近未使用或者使用比较少的键尝试回收。(对于设置了过期时间的键)
- 随机删除某些键(对于所有键)
- 随机删除某些键(对于设置了过期时间的键)
- 回收时间较短的键(对于设置了过期时间的键)

Redis删除过期键的策略(缓存失效策略、数据过期策略)

- 定时删除：在设置键的过期事件的同时，创建一个定时器，让定时器在键的过期时间来临时，立即执行对键的删除操作。对内存最友好，对cpu时间最不友好。
- 惰性删除：每次获取键时，都会检查键是否过期，如果过期就删除键；如果没有过期就返回该键。对CPU时间最优化，对内存最不友好。
- 定期删除：每个一段时间，默认100ms，程序就会对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少数据库由算法决定。前两种策略的折中，对cpu时间和内存的友好程度较平衡。
- Redis使用惰性删除和定期删除。

Redis的内存驱逐(淘汰)策略

- 当redis的内存空间已经用满时，就会根据配置的驱逐策略，进行相应的动作

- 有以下8中：
    1. 默认策略，不淘汰任何key，返回错误
    2. 在所有key中使用LRU算法淘汰部分key
    3. 在所有key中使用LFU算法淘汰部分key
    4. 在所有key中随机淘汰部分key
    5. 在设置了过期时间的key中，使用LRU算法淘汰部分key
    6. 在设置了过期时间的key中，使用LFU算法淘汰部分key
    7. 在设置了过期时间的key中，随机淘汰部分key
    8. 在设置了过期时间的key中，挑选TTL短的key淘汰

数据库和缓存的一致性问题

不管先写数据库再删缓存还是先删缓存再写数据库，都可能会造成数据不一致的问题。

1. 当写数据库再删缓存时，当写完数据库，写进程的服务器突然宕机了，没有删掉缓存。
2. 当先删缓存时，一个线程还没来得及写数据库，另一个线程来读缓存，发现没有，去读数据库，并更新缓存，这时候写数据库的进程已经把缓存更新了，又更新回了久的，读到了脏数据。

缓存的三种策略

- Cache Aside Pattern 旁路缓存模式
- Read/Write through Pattern 读写穿透模式
- Write behind Pattern 异步缓存写入

旁路缓存模式(Cached Aside Pattern)

- 写：先更新DB，然后直接**失效缓存**
- 读：查询缓存，如果缓存存在，返回结果。如果缓存中不存在，读取DB，返回结果，并把返回的结果写入Cache
- 无论先操作数据库还是先操作缓存，都会存在脏数据的情况。

读写穿透模式(Read/Write Through Pattern)

- 在旁路缓存模式下，应用层去和缓存和数据库打交道，增加了应用层的复杂度，在读写穿透模式下，应用层之和缓存打交道，由缓存去操作和维护数据库。
- 写：读取缓存，(1)缓存未命中，缓存去更新数据库，数据库返回结果，缓存返回更新成功。(2)缓存命中，更新缓存，缓存更新数据库，数据库返回结果，缓存返回更新成功。
- 读：读取缓存，(1)缓存命中，返回结果。(2)缓存未命中，缓存去查询数据库，写入缓存，缓存返回结果。

异步缓存写入模式(Write Behind Caching Pattern)

- 读写穿透模式每次更新数据都回去同时写入数据库，数据写入速度会比较慢。
- 异步缓存写入模式会在一段时间之后异步的将数据批量写入数据库。优点：(1)应用层只写缓存，速度快。(2)缓存在异步的写入数据库的时候会将多个I/O操作合并为一个，减少I/O次数。
- 缺点：复杂度高，更新后的数据如果没写入数据库，遇到断电问题会导致数据丢失。

如何保证数据库与缓存的一致性

- 可以引入分布式事务来解决，常见的有：2PC、TCC、MQ事务消息等。但是引入分布式事务，会带来性能上的影响。
- 在实际的使用中，通常不会去保证cache和DB的强一致性，而是允许两者在短暂时间内的不一致，保证两者的最终一致性。
- 最终一致性的常用方案如下：(1)更新数据库，数据库产生binlog，监听和消费binlog，执行实现缓存操作。如果失效缓存失败，引入重试机制，将失败的数据通过MQ的方式进行重试。

异步更新缓存(基于订阅binlog的同步机制)

1. 读Redis：热数据基本都在Redis
2. 写数据库：增删改都在操作MySQL
3. 更新Redis数据：MySQL的数据更新binlog，来更新Redis
4. Redis增量更新。订阅binlog，读取分析后，对Redis进行更新。

Memcached VS Redis

共同点

1. 都是基于内存的数据库，一般都用来当作缓存使用
2. 都有过期策略
3. 两个性能都很高

区别

1. Redis支持更丰富的数据结构
2. Redis支持数据的持久化
3. Redis有灾难恢复机制
4. Redis在服务器内存用完之后，可以将不用的数据放到磁盘上
5. Memcached没有原生的集群模式
6. Redis使用单线程的多路IO复用模型
7. Redis支持订阅模型
8. Redis过期数据使用了惰性删除和定期删除

消息队列

什么是消息队列

消息队列的作用

1. 异步 有一个支付系统，支付完还有积分系统，优惠券系统，可以让积分系统和优惠券系统异步去做。
2. 削峰 大量的流量进来，可能通过降级的手段返回一个友好页面，当流量过去之后再进行操作。
3. 解耦 积分系统和优惠券系统只需要订阅支付消息，就可以完成操作，不需要因为增加一个系统功能，重新发布整个系统。

消息队列的基本概念

如何保证消息不丢失

1. 生产者生产消息确保消息不丢失 生产者发送消息至Broker，需要处理Broker的响应，不论是同步或者异步发送消息，同步和异步发送消息，同步和异步回调都要做好try-catch，妥善的处理响应，如果Broker返回写入失败等错误消息，需要重试发送。当多次发送失败消息需要作报警，日志记录。
2. Broker存储消息 消息存储阶段，要在写入磁盘之后再返回确认消息，如果在写入缓存中就返回响应，那么机器突然断电，消息就没了，生产者以为已经发送成功了。如果是集群部署，有多副本机制，应该是至少几台机器成功写入之后再返回确认消息。
3. 消费者消息消息 当消息者真正执行完逻辑再返回确认消息。

如何处理重复消息(幂等性)

场景：例如有一个支付系统，支付完成之后，接着有积分系统，优惠券系统，扣库存系统。 当支付完成后，积分系统进行积分的累加，优惠券系统发送优惠券，扣库存系统扣库存，但是积分系统处理失败了，这个系统要求重发一次这个消息，但是优惠券系统和扣库存系统也监听了这个消息，所以又收到了一次，所以出现了错误。这个时候应该考虑**接口幂等，即执行多次和执行一次的结果是一样的**。强校验：比如在插入数据库的时候，首先判断主键或者订单编号是否存在。存在就不插入。弱校验：比如发送短信，再发一次影响也不是很大，那就再发一次。

如何保证消息的有序性

场景：binlog的同步。从数据或主数据库数据同步到备库，这种涉及到数据量很大的时候都是放到消息队列中慢慢消费的。 问题：比如在数据库同时对一个ID的数据进行了增改删的操作，但是消息发过去的时候变成了改增删，这样数据就不对了。

- 全局有序：首先只能由一个生产者往Topic发送消息，并且一个Topic内只能有一个队列，消费者也必须是单线程消费这个消息。这样的消息就是全局有序。
- 部分有序：把Topic划分成我们需要的队列数，把消息通过特定的策略发往固定的队列中，然后每个队列对应一个单线程消费处理的消费者，这样就完成了部分有序。比如可以按照订单号进行取模的方式，将一个订单的全部操作同步的放到一个队列中，只有同个订单创建消息成功，再发送支付消息。然后让消费者消费这些消息。不同的订单号可能会放到不同的队列中。

如何处理消息堆积

- 消息的堆积是因为生产者的速度和消费者的速度不匹配。有可能是消息消费失败反复重试造成的，也有可能是消费者消费能力弱，渐渐地消息就积压了。
- 有bug就处理bug，如果因为本身消费能力比较弱，可以优化一下消费逻辑，如果之前是一条一条消费的，就批量处理。
- 水平扩容，增加Topic的队列数和消费者数量。

消息的不一致问题

1. 查询不一致
2. leader未发送proposal宕机
3. leader成功发送proposal，但是发送commit宕机

ZooKeeper

ZooKeeper概念

- ZooKeeper是一个分布式协调服务的开源框架。主要用来解决分布式集群中应用系统的一致性问题，例如怎样避免同时操作同一数据造成脏读的问题。ZooKeeper本质上是一个分布式的小文件存储系统。提供基于类似文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理，从而来维护和监控你存储的数据的状态变化。将通过监控这些数据的状态的变化，从而可以达到基于数据的集群管理。例如：统一命名服务(dubbo)、分布式配置管理、分布式消息队列、分布式锁、分布式协调等功能。
- Zookeeper是一个典型的分布式数据一致性解决方案，分布式应用程序可以基于ZooKeeper实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master选举、分布式锁和分布式队列等功能。ZooKeeper一个最常用的功能就是用于担任服务生产者和服务消费者的注册中心。

ZooKeeper架构图

1. Leader：ZooKeeper集群工作的核心事务请求的唯一调度和处理者，保证集群事务处理的顺序性；集群内部各个服务的调度者。对于create、setData、delete等有写操作的请求，需要统一转发给leader处理，leader需要决定编号、执行操作、这个过程称为一个事务。
2. Follower：处理客户端非事务，请求转发事务请求给Leader，参与集群leader选举投票2n-1台可以做集群投票。此外对于访问量比较大的zooKeeper集群，还可以新增观察者角色。
3. Observer：观察者角色。观察ZooKeeper集群的最新状态变化并将这些变化同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则转发给Leader服务器处理，不参与任何形式的投票，只提供服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。(增加并发的请求)

ZooKeeper的特点

1. 全局一致性：每个server都保存一份相同的副本，不论client访问哪台server，访问到的数据都是一致的。
2. 可靠性：如果消息被一台服务器接收，那么将被所有服务器接受
3. 顺序性：全局有序性和偏序：全局有序：如果一台服务器上消息a在消息b前发布，那么所有server上消息a在消息b前发布。偏序性：如果一个消息b在消息a后被同一个发送者发布，则a必须排在b前面。通过单长连接通道交互的，中间通过队列缓存。
4. 数据操作的原子性：一次数据更新要么成功、要么失败。
5. 实时性：ZooKeeper保证客户端将在一段时间间隔内收到服务端的更新信息，或者服务器失效的信息。

ZAB协议

- ZAB协议是为ZooKeeper设计的崩溃恢复原子广播协议，它保证zooKeeper集群数据的一致性和命令的全局有序性。

概念介绍

1. 集群角色：Leader(同一时间内只允许有一个leader存在，提供对客户端的读写功能，负责将数据同步到各个节点)、Follower(提供对客户端的读功能，写请求转发给Leader，当leader失效后参与投票)、Observer(与Follower不同的是不参与投票)。
2. 服务状态：LOOKING，FOLLOWING，LEADING，OBSERVERING。**ZAB的状态分为**:ELECTION，DISCOVERY(连上leader，响应leader心态，并检测leader的角色是否更改，通过此步骤之后选举出来的leader才能执行真正的职务)，SYNCHRONIZATION，BROADCAST。
3. ZXID，是一个long型的整数，分为两部分，epoch和计数器(counter)部分，是有一个全局有效的数字。epoch代表当前集群所属哪个leader。epoch代表当前命令的有效性。counter是一个递增的数字。

选举

- 时机：(1)集群当开始启动的时候，还没有leader，需要进行选举。(2)当因为各种意外情况，leader服务器宕机，需要进行leader选举。
- 投票规则：按照epoch(纪元)，zxid和serverId来进行依次选择较大的。由此可以看出尽量把服务性能更大的集群的serverId配置大一些。
- 过程：首先会将票投给自己，然后将自己的投票信息进行广播，当收到其他的投票信息后，决定是否更改投票信息。投票完成后，进行统计投票信息，如果集群中过半的机器都选择了某一台机器，则该机器成为新的leader，选举结束。

广播(集群对外提供服务，如何保证各个节点数据的一致性)

- zab协议在广播状态保证以下特征：

1. 可靠传递：如果消息m被一台服务器传递，那么它将被全部服务器传递。
2. 全局有序：如果一个消息a在消息b之前被一台服务器交付，所有服务器都交付了a和b，并且a先于b。
3. 因果有序：如果消息a因果上先于消息b，并且两者都被交付，那么a必须排在b前面。

写请求

- 整个写请求类似一个二阶段的提交。首先leader收到客户端的写请求后，会生成一个事务(proposal)，其中包含了zxid。leader开始广播该事务，**需要注意的是所有节点的通讯都是由一个FIFO队列维护的**。follower收到事务后，将事务写入本地磁盘，写入成功后并返回一个ack。当leader收到过半的ack后，会进行事务的提交，并广播事务提交信息。从节点开始提交事务。
- ZooKeeper通过二阶段提交来保证集群中数据的一致性，因为只要收到过半的ack就可以提交事务，所以ZooKeeper不是强一致性的。
- zab协议的有序性保证是通过几个方面来体现的：(1)服务之间用的是TCP通讯，保证在网络中的有序性。(2)通讯使用的是FIFO队列，保证消息的全局一致性。(3)通过全局递增zxid来保证因果有序性。

分布式锁

Watcher监听机制和它的原理

ZooKeeper可以提供分布式数据的发布订阅功能，依赖的就是Watcher监听机制。 客户端可以向服务器端注册Watcher监听，服务端的指定事件触发之后，就会向客户端发送一个事件通知。 有几个特性：

1. 一次性：一旦一个Watcher被触发，Zookeeper就会将它从存储中移除。
2. 客户端串行：客户端的Watcher回调处理是串行同步的过程，不要因为一个Watcher的逻辑阻塞整个客户端。
3. 轻量 主要流程如下：
4. 客户端向服务端注册Watcher监听
5. 保存Watcher对象到客户端本地的WatcherManager中。
6. 服务端Watcher事件触发后，客户端收到服务端通知，从WatcherManager中取出响应Watcher对象执行回调逻辑。

如何保持数据一致性

使用ZAB协议来保证数据的最终一致性，类型一个2PC两阶段提交的过程。主要流程与写请求类似。

如何进行leader选举

时机：分为启动时的leader选举与运行期间的leader选举

数据同步

在选举完成之后，Follower和Observer就会去向leader注册，然后就会开始数据同步工作。 数据同步包含3个主要值和4种形式。 PeerLastZxid：learner(follower和observer)服务器最后处理的ZXID。 minCommitedLog：Leader提议缓存队列中最小的zxid maxCommitedLog：Leader提议缓存队列中最大的zxid

1. 直接差异化同步，DIFF同步：PeerLastZxid处于min与max之间
2. 回滚再差异化同步：对于只存在于learner中的提议，先回滚至离max最近的位置，再进行同步
3. 仅回滚同步：PeerLastZxid大于max
4. 全量同步：PeerLastZxid小于min

数据不一致性问题

Kafka

Kafka的简介

- Kafka是一个分布式的基于发布/订阅模式的消息队列，主要应用于大数据的实时处理领域。

使用消息队列的好处

- 解耦
- 可恢复性
- 缓冲
- 灵活性和峰值处理能力
- 异步通信

消息队列的两种模式

- 点对点模式
- 发布/订阅模式：推/拉 Kafka基于拉取的发布/订阅模式，消费者的消费速度可以根据自己来决定。一直维护着长轮询，可能会造成资源的浪费。

Kafka架构

- Topic：Producer将消息发送到特定的Topic，consumer通过订阅这个Topic来消费消息。
- Partition：Partition是Topic的一部分，一个Topic可以有多个Partition，同一Topic下的Partition可以分布在不同的Broker中。
- Producer：消费消息的一方
- Consumer：生产消息的一方
- ConsumerGroup
- Broker：可以看做是一个kafka实例，多个kafka broker组成一个kafka cluster。
- Offset：记录Consumer对Partition中消息的消费进度。

Kafka和其他消息队列的区别

Kafka中Zookeeper的作用

主要为Kafka提供元数据的管理功能。

1. Broker注册：Kafka会将该Broker信息存入其中，Broker在Zookeeper中创建的是临时节点，一旦Broker故障下线，Zookeeper就会将该节点删除。同时可以基于Watcher机制监听该节点，当节点删除后做出相应反应。
2. Topic注册：所有Broker和Topic的对应关系都由Zookeeper来维护。`/brokers/topics/{topicname}`。还完成Topic中leader的选举。
3. Consumer的注册和负载均衡：①Consumer Group的注册`/consumers/{group_id}`。在其目录下有三个子目录。ids：一个Consumer Group有多个Consumer，ids用来记录这些Consumer。owners：记录该用户组可消费的Topic信息。offsets：记录owners中每个Topic的所有Partition的所有OffSet。②Consumer的注册：注册的是临时节点(`/consumers/{group_id}/ids/{consumer_id}`)。③负载均衡：一个Consumer Group下有多个Consumer，怎么去均匀的消费订阅消息。由Zookeeper来维护。
4. Producer的负载均衡：
5. 维护Partition和Consumer的关系：同一个Consumer Group订阅的任一个Partition都只能分配给一个Consumer，Partition和Consumer的对应关系路径：`/consumer/{group_id}/owners/{topic}/{broker_id-partition_id}`，该路径下的内容是该消息分区消费者的Consumer ID。这个路径也是一个临时节点，在Rebalance时会被删除。
6. 记录消息消费的进度：在2.0版本中不再记录在Zookeeper中，而是记录在Kafka的Topic中。

Kafka分区的目的

实现负载均衡，分区对于消费者来说，可以提高并发度，提高效率

Kafka如何做到消息的有序性?

kafka中每个partition的写入是有序的，而且单个partition只能由一个消费者消费，可以保证里面的消息的顺序性，但是分区之间的消息是不保证有序的。

Kafka中leader的选举机制

OffSet的作用

Kafka的高可靠性是怎么保证的?

- Topic分区副本 Kafka可以保证单个分区的消息是有序的，分区可以分为在线和离线，众多的分区中只有一个是leader，其他的是follower。所有的读写操作都是通过leader进行的，同时follower会定期去leader上复制数据。当leader挂了之后，follower称为leader。通过分区副本，引入了数据冗余，同时提供了Kafka的数据可靠性。
- Producer向Broker发送消息 为了让用户设置数据可靠性，Kafka在Producer中提供了消息确认机制，可以通过配置来决定消息发送到对应分区的几个副本才算发送成功。三个级别：发送出去就算成功。发送给leader，并把它写入分区数据文件，返回确认或失败。发送出去，并等到同步副本都收到消息才算成功。

Kafka的数据一致性原理

- ISR：副本能够跟的上leader的进度
- OSR：副本没有跟上leader的进度
- AR：所有的副本

- LEO：当前日志文件的下一条
- HW：高水位
- LSE：对未完成的事务而言，LSO的值等于事务中第一条消息的位置
- 因为网络原因，副本的复制速度可能有所不同，所以kafka只支持读取HW之上的所有数据。

Kafka在什么情况下会出现消息丢失?

如果leader崩溃，另一个副本称为新的leader，那么leader新写的那些消息就可能丢失了。如果我们允许消费者去读取这些消息，可能就破坏了消息的一致性。试想：一个消费者从当前leader读取并处理了message4，这个时候leader挂掉了，选举了新的leader，这个时候另一个消费者在新的leader读取消息，发现这个消息其实并不存在，就造成了数据不一致性。

Kafka数据传输的事务有几种?

- 最多一次：消息不会被重复发送，最多被传输一次，但也有可能一次不传输
- 最少一次：消息不会被漏发送，消息至少被传输一次，但也有可能被重复发送。
- 精确一次：不会被漏发送也不会被重复发送，消息正好被传输一次。

Kakfa高效文件存储设计特点

- Kafka把topic中一个partition大文件分成多个小文件，通过多个小文件段，就容易定期清除或删除已经消费完文件，减少磁盘占用
- 通过索引信息可以快速定位message和确定response大小
- 通过index元数据全部映射到memory，可以避免segment file的io操作。
- 通过索引文件稀疏存储，可以大幅度降低index文件元数据占用空间大小。

Kafka的rebalance

在kafka中，当有新的消费者加入或者订阅的topic数发生变化，会触发rebalance。重新均衡消费者消费。

1. 所有成员向coordinator发送请求，请求入组。一旦所有成员都发送了请求，coordinator会从中选择一个作为leader，并把组成员信息和订阅信息发给leader。
2. leader开始分配消费方案，指定哪个consumer负责消费哪些topic的partition。一旦分配完成，leader将这个方案发送给coordinator。coordinator收到方案后，发送给每个consumer，这样组内每个消费者都能知道自己消费哪个topic的哪个分区了。

Kafka为什么这么快？（高吞吐量）

1. 顺序写入：不断追加到文件中，这个特性可以让kafka充分利用磁盘的顺序读写性能。顺序读写不需要硬盘磁头的寻道时间，只需要很少的磁盘旋转时间，所以速度远快于随机读写。
2. Memory Mapped Files：直接利用操作系统的page来完成文件到物理内存的映射，完成之后对物理内存的操作会直接同步到磁盘。通过内存映射的方式会大大提高IO效率，省去了用户空间到内核空间的复制。
3. 零拷贝sendfile
4. 分区：kafka中的topic中的内容可以被分为多个partition，每个partition又分为多个segment，每次操作都是对一小部分做操作，很轻便，同时增加了并行操作的能力。
5. 批量发送：producer发送消息的时候，可以将消息缓存到本地，等到固定条件发送到kafka中。
6. 数据压缩：减少传输的数据量，减轻对网络传输的压力。
7. 高效的网络模型，Reactor

Spring

Spring框架概述

- Spring框架是一个轻量级的、开放源代码的J2EE应用程序框架。
- Spring有两个核心部分：IOC和Aop

1. IOC：控制反转，把创建对象的过程交给Spring进行管理。
2. Aop：面向切面，不修改源代码的情况下，进行功能的增加和增强。

- Spring框架的相关的特点

1. 方便解耦，简化开发。
2. Aop编程支持
3. 方便程序的测试
4. 方便整合各种优秀框架
5. 降低Java EE API的使用难度
6. 方便进行事务操作

IOC是什么

IOC控制反转，实现两种方式，依赖查找和依赖注入，主要实现方式是依赖注入。之前创建对象需要new，现在将创建对象的权限交给IOC容器来完成。当容器创建对象的时候主动将它需要的依赖注入给它。

IOC容器初始化过程

- 基于XML的初始化方法

1. ClassPathXmlApplicationContext初始化父类，调用父类方法初始化资源加载器。通过setConfigLocation方法加载相应的xml配置文件。
2. refresh方法规定了IOC容器初始化的过程，如果之前存在IOC容器则销毁IOC容器，保证每次创建的IOC容器都是新的。
3. 容器创建后通过loadBeanDefinition方法加载Bean的配置信息。主要做两件事：调用资源加载的方法加载相应的资源。通过XmlBeanDefinitonReader方法加载真正的bean的信息，并将xml文件中的信息转换成对应的文件对象。按照Spring Bean的规则对文档对象进行解析。
4. IOC容器中解析的Bean会放到一个HashMap中，key是string，value是BeanDefinition。注册过程中需要使用Synchronized来保证线程安全。当配置信息中配置的Bean被加载到IOC容器后，初始化就算完成了。Bean的定义信息已经可以被定位和检索，IOC容器的作用就是为Bean定义信息进行处理和维护，注册的Bean的定义信息是控制反转和依赖注入的基础。

- 基于注解的初始化方法

1. 直接将所有的注解Bean注册到容器中，可以在容器初始化时进行注册，也可以在容器初始化完成后进行注册，注册完成之后刷新容器，让容器对Bean的注册信息进行处理和维护。
2. 通过扫描指定的包和子包加载所有的类，在初始化注解容器时，指定要扫描的路径。

依赖注入的实现方式有哪些

1. setter

2. 构造函数
3. 接口

依赖注入的相关注解有哪些

1. @AutoWired 先按照类型，类型相同按照Bean的id
2. @Qualifier 按照类型，Bean的id进行注入
3. @Resource 按照Bean的id进行注入，只能注入Bean类型
4. @Value 只能注入基本数据类型和String类型

依赖注入的过程

1. 当IOC容器被初始化完之后，调用doGetBean方法来实现依赖注入。具体方法是通过BeanFactory的createBean完成，通过触发createBeanInstance方法来创建对象实例和populateBean对其Bean属性依赖进行注入。
2. 依赖注入的过程就是将创建的Bean对象实例设置到所依赖的Bean对象的属性上。真正的依赖注入通过setPropertyValues方法实现的。
3. BeanWrapperImpl方法实现对初始化的Bean对象进行依赖注入，对于非集合类型属性，通过JDK反射，通过属性的setter方法设置所需要的值。对于集合类型的属性，将属性值解析为目标类型的集合后直接赋值给属性。

Bean的生命周期

1. 当完成Bean对象的初始化和依赖注入后，通过调用后置处理器BeanPostProcessor的PostProcessBeforeInitialization方法，在调用初始化方法init-method之前添加我们的实现逻辑，调用init-method方法，之后调用BeanPostProcessor的PostProcessAfterInitialization方法，添加我们的实现逻辑。当Bean调用完成之后，调用destory-method方法，销毁bean。

Spring Bean的生命周期

- 实例化BeanFactoryPostProcessor实现类
- 执行BeanFactoryPostProcessor的postProcessBeanFactory方法
- 实例化BeanPostProcessor实现类
- 实例化InstantiationAwareBeanPostProcessor实现类
- 执行InstantiationAwareBeanPostProcessor的postProcessBeforeInstantiation方法
- **执行Bean的构造器**
- 执行InstantiationAwareBeanPostProcessor的postProcessPropertyValues方法
- **为Bean注入属性**
- 调用BeanNameAware的setBeanName方法
- 调用BeanFactoryAware的setBeanFactory方法
- 执行BeanPostProcessor的postProcessBeforeInitialization方法
- 调用InitializingBean的afterPropertiesSet方法
- **调用<bean>的init-method属性指定的初始化方法**
- 执行BeanPostProcessor的postProcessAfterInitialization方法
- 执行InstantiationAwareBeanPostProcessor的postProcessAfterInstantiation方法
- 容器初始化成功，执行正常调用后，下面销毁容器
- 调用DisposibleBean的destory方法
- 调用<bean>的destroy-method属性指定的初始化方法

Bean的作用范围

1. Singleton
2. Prototype
3. Session
4. global Session
5. request

如何通过XML创建Bean

如何通过注解创建Bean

1. @Component
2. @Controller
3. @Service
4. @Repository
5. @Bean 被Bean注解的方法返回值是一个对象，将会被实例化，配置和初始化一个对象返回，这个对象只能由Spring IOC容器来管理。

如何通过注解配置文件

BeanFactory，FactoryBean和ApplicationContext的区别

- BeanFactory是一个Bean工厂，使用了简单工厂模式，是Spring IOC容器的顶级接口，可以理解为含有Bean集合的工厂类，负责Bean的实例化，依赖注入，BeanFactory实例化后并不会实例化这些Bean，只有当用到的时候才去实例化，采用懒汉式，适合多例模式。
- FactoryBean是一个工厂Bean，使用了工厂方法模式，作用是生产其他的Bean实例，通过实现接口，通过一个工厂方法来实现自定义实例化Bean的逻辑。
- ApplicationContext是BeanFactory的子接口，扩展了BeanFactory的功能。在容器初始化时对Bean进行预实例化，容器初始化时，配置依赖关系就已经完成，属于立即加载，饿汉式，适合单例模式。

Spring 扩展接口

- BeanFactoryPostProcessor：Spring允许在Bean创建之前，读取Bean的元属性，并根据自己的需求对元属性进行修改，比如将Bean的scope从singleton改为prototype
- BeanPostProcessor：在每个bean初始化前后做操作
- InstantiationAwareBeanPostProcessor：在bean实例化前做操作
- BeanNameAware、ApplicationContextAware和BeanFactoryAware：针对bean工厂，可以获取上下文，可以获取当前bean的id
- InitialingBean：在属性设置完毕后做一些自定义操作
- DisposibleBean：在关闭容器之前做一些操作

循环依赖

- 缓存分为三级：1. `singletonObjects`，一级缓存，存储的是所有创建好了的单例Bean。2. `earlySingletonObjects`，完成实例化，但是还未进行属性注入及初始化对象。3. `singletonFactories`，提前暴露的一个单例工厂，二级缓存中存储的就是这个从工厂中获取的对象。

- 每个缓存的作用：1. 一级缓存中存放的是已经完全创建好的单例Bean。2. 三级缓存中存放的是在完成Bean的实例化后，属性注入之前Spring将Bean包装成一个工厂。

Spring是如何解决循环依赖的?

Spring通过三级缓存解决了循环依赖，其中一级缓存为单例池`singletonObjects`，二级缓存为早期曝光对象`earlySingletonObjects`，三级缓存为早期曝光对象工厂`singletonFactories`。

当A、B两个类发生循环引用时，在A完成实例化后，就使用实例化后的对象去创建一个对象工厂，并添加到三级缓存中，如果A被AOP代理，那么通过这个工厂获取到的就是A代理后的对象，如果A没有被AOP代理，那么这个工厂获取到的就是A实例化后的对象。

当A进行属性注入时，会去创建B，同时B又依赖了A，所以创建B的同时又会去调用getBean(a)来获取需要的依赖，此时的getBean(a)会从缓存中获取：

第一步，先获取到三级缓存中的工厂；

第二步，调用对象工厂的getObject方法来获取到对应的对象，得到这个对象后将其注入到B中。紧接着B会走完它的生命周期，包括初始化、后置处理器等。

当B创建完后，会将B再注入到A中，此时A再完成它的整个生命周期。至此，循环依赖结束。

为什么要使用三级缓存呢？二级缓存能解决循环依赖吗?

如果要使用二级缓存来解决循环依赖，意味着所有Bean在实例化后就要完成AOP代理，这样违背了Spring设计的原则，Spring在设计之初就是通过`AnnotationAwareAspectJAutoProxyCreator`这个后置处理器来在Bean生命周期的最后一步来完成AOP代理，而不是在实例化后就立马进行AOP代理。

什么是AOP

AOP的相关注解有哪些

1. @Aspect：声明被注解的类是一个切面Bean
2. @Before：前置通知。指在某个连接点之前执行的通知。
3. @After：后置通知：在某个连接点退出时执行的通知(不论正常返回还是异常退出)
4. @AfterReturning：返回后通知。
5. @AfterThrowing：异常通知，方法抛出异常导致退出时执行的通知。

AOP的相关术语

1. Aspect：切面，一个关注点的模块化，这个关注点可能会横切多个对象。
2. Joinpoint：连接点，程序执行过程中的某一行为，即业务层中的所有方法。
3. Advice：通知，指切面对于某个连接点所产生的的动作，包括前置通知，后置通知，返回后通知，异常通知和环绕通知。
4. Pointcut：切入点，指被拦截的连接点，切入点一定是连接点，但连接点不一定是切入点。
5. Proxy：代理
6. Target：代理的目标对象，指一个或多个切面所通知的对象。
7. Weaving：织入，指把增强应用到目标对象来创建代理对象的过程。

AOP的过程

1. AOP是从BeanPostProcessor后置处理器开始，后置处理器可以监听容器触发的Bean生命周期时间，向容器注册后置处理器以后，容器中管理的Bean就具备了接收IOC容器回调事件的能力。
2. BeanPostProcessor的调用发生在SpringIOC容器完成Bean实例对象的创建和属性的依赖注入后，为Bean对象添加后置处理器的入口是initialization方法。
3. Spring中JDK动态代理通过JdkDynamicAopProxy调用Proxy的newInstance方法来生成代理类，JdkDynamicAopProxy也实现了InvocationHandler接口，invoke方法的具体逻辑是先获取应用到此方法上的执行器链，如果有执行器则创建MethodInvocation并调用proceed方法，否则直接反射调用目标方法。因此Spring AOP对目标对象的增强是通过拦截器实现的。

什么是事务?

Spring支持两种方式的事务管理

声明式事务管理和编程式事务管理

PlatFormTransactionManager

事务管理器，Spring事务策略的核心 Spring并不直接管理事务，而是提供了多种事务管理器。Spring事务管理器的接口是PlatFormTransactionManager

TransactionDefinition

事务定义信息(事务隔离级别、传播行为、超时、只读、回滚规则)

```
public interface TransactionDefinition {

    // ------------  事务传播行为 ------------
    int PROPAGATION_REQUIRED = 0; // 默认的，如果当前存在事务，就加入该事务，如果当前没有事务，则创建一个新的事务。
    int PROPAGATION_SUPPORTS = 1; // 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
    int PROPAGATION_MANDATORY = 2; // 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）
    int PROPAGATION_REQUIRES_NEW = 3; // 创建一个新的事务，如果当前存在事务，则把当前事务刮起。也就是说不管外部方法是否开启事务，修饰的内部方法会新开启自己的事务，且开启的事务互相独立，互不干扰。
    int PROPAGATION_NOT_SUPPORTED = 4; // 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
    int PROPAGATION_NEVER = 5; // 以非事务方式运行，如果当前存在事务，则抛出异常。
    int PROPAGATION_NESTED = 6; // 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION_REQUIRED。也就是说：
    // 1. 在外部方法未开启事务的情况下Propagation.NESTED和Propagation.REQUIRED作用相同，修饰的内部方法都会新开启自己的事务，且开启的事务相互独立，互不干扰。
    // 2. 如果外部方法开启事务的话，Propagation.NESTED修饰的内部方法属于外部事务的子事务，外部主事务回滚的话，子事务也会回滚，而内部子事务可以单独回滚而不影响外部主事务和其他子事务。
```

```
    // -------------  事务隔离级别  ----------
    int ISOLATION_DEFAULT = -1;
    int ISOLATION_READ_UNCOMMITTED = 1;
    int ISOLATION_READ_COMMITTED = 2;
    int ISOLATION_REPEATABLE_READ = 4;
    int ISOLATION_SERIALIZABLE = 8;
    int TIMEOUT_DEFAULT = -1;
    // 返回事务的传播行为，默认值为 REQUIRED。
    int getPropagationBehavior();
    //返回事务的隔离级别，默认值是 DEFAULT
    int getIsolationLevel();
    // 返回事务的超时时间，默认值为-1。如果超过该时间限制但事务还没有完成，则自动回滚事
务。
    int getTimeout();
    // 返回是否为只读事务，默认值为 false
    boolean isReadOnly();

    @Nullable
    String getName();
}
```

- 事务传播行为是为了解决业务层方法之间互相调用的事务问题。当事务方法被另一事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。

TransactionStatus

事务运行状态

@Tranactional注解

Spring框架中用到了哪些设计模式?

- 工厂设计模式：Spring使用工厂模式通过BeanFactory、ApplicationContext创建bean对象
- 代理设计模式：Spring AOP功能的实现
- 单例设计模式：Spring中的Bean默认都是单例的
- 包装器设计模式：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要访问不同的数据库，这种模式会让我们根据客户的需求能够动态切换不同的数据源。
- 观察者模式：Spring事件驱动模型
- 适配器模式：Spring AOP的增强或通知使用到了适配器模式，Spring MVC中也使用到了适配器模式适配Controller

Spring涉及到的几种设计模式

MyBatis

#{} 和 ${} 的区别

${} 相当于使用字符串拼接，存在SQL注入风险。 #{} 相当于使用占位符，可以防止SQL注入，动态参数。

一级缓存是什么

- 是SqlSession级别的，操作数据库时需要创建SqlSession对象，对象中有一个HashMap存储缓存数据，不同的SqlSession之间缓存数据区域互不影响。在同一个SqlSession中执行两次相同的SQL语句，第一次执行完毕会将结果保存到缓存中，第二次查询时直接从缓存中获取。
- 如果SqlSession执行了DML操作(IUD)，MyBatis必须将缓存清空保证数据有效性。

二级缓存是什么

二级缓存是Mapper级别的，默认关闭。多个SqlSession可以共用二级缓存，作用域是Mapper的同一个namespace。

SpringMVC

SpringMVC的处理流程

1. Web容器启动时，会通知Spring初始化容器，加载Bean的定义信息并初始化所有单例Bean，然后遍历容器中的Bean，获取每一个Controller中的所有方法访问的URL，将URL和对应的Controller放到一个Map中。
2. 所有请求都发给DispatcherServelet前端处理器处理，DispatcherServelet会请求HandlerMapping寻找被Controller注解修饰的Bean和被RequestMapping修饰的方法和类。生成Handler和HandlerInterceptor，并以HandlerExecutionChain这样一个执行器链的形式返回。
3. DispatcherServlet使用Handler寻找对应的HandlerAdapter，HandlerAdapter执行相应的Handler方法，并把请求参数绑定到方法形参上，执行方法获得ModelAndView。
4. DispatcherServelet将ModelAndView讲给ViewResolver进行解析，得到View的物理视图，然后对视图进行渲染，将数据填充到视图中返回给客户端。

DispatcherServlet的作用

- DispatcherServlet是前端控制器设计模式的实现，提供SpringWebMVC的集中访问点，而且负责职责的分派，而且与SpringIoC容器无缝集成，从而可以获得Spring的所有好处。DispatcherServlet主要用作职责调度工作，本身主要用于流程控制

1. 文件上传解析，如果请求类型是multipart将通过MultipartResolver进行文件上传解析。
2. 通过HandlerMapping，将请求映射到处理器(返回一个HandlerExecutionChain，它包含一个处理器、多个HandlerInterceptor拦截器)。
3. 通过HandlerAdapter支持多种类型的处理器(HandlerExecutionChain处理器)。
4. 通过ViewResolver解析逻辑视图名到具体视图实现。
5. 本地化渲染。
6. 渲染具体的视图等。
7. 如果执行过程中遇到异常交给HandlerExceptionResolver来解析。

DispatcherServlet初始化顺序

- HttpServletBean继承HttpServlet，在Web容器启动时将调用它的init方法，(1)该初始化方法的主要作用是将Servlet初始化参数设置(init-param)到该组件上(如contextAttribute、contextClass、namespace、contextConfigLocation)，将通过BeanWrapper简化设值过程，方便后续使用。(2) 提供子类初始化扩展点，initServletBean()，该方法由FrameServlet覆盖。

- FrameServlet继承HttpServletBean，通过initServletBean()进行Web进行上下文初始化，该方法主要覆盖两件事情，初始化Web上下文，提供给子类初始化扩展点。
- DispatcherServlet继承FrameworkServlet，并实现了onRefresh()方法提供一些前端控制器的相关配置。

综上，DispatcherServlet初始化主要做了两件事

- 初始化SpringWebMVC使用的Web上下文，并且可能指定父容器为ContextLoaderListener。
- 初始化DispatcherServlet使用的策略，如HandlerMapping、HandlerAdapter等

ContextLoaderListener初始化的上下文和DispatcherServlet初始化的上下的关系

- ContextLoaderListener初始化的上下文加载的Bean是对于整个应用程序共享的，一般是DAO层，Service层Bean。
- DispatcherServlet初始化上下文加载的Bean是只对SpringWebMVC有效的Bean，如Controller、HandlerMapping、HandlerAdapter等，该初始化上下文应该只是加载Web相关组件

SpringMVC有哪些组件

SpringMVC有哪些注解

处理器拦截器

- 常见应用场景：日志记录，权限检查，性能监控，通用行

SpringBoot

Springboot的优点

1. 独立运行：SpringBoot内嵌了各种servlet容器，Tomcat等，现在不需要达成war包部署到容器中，SpringBoot只需要达成一个jar包就能独立运行，所有的依赖包都在一个jar包中。
2. 简化配置：spring-boot-starter-web启动器自动依赖其他组件，减少了很多maven的配置。
3. 自动配置：springboot能够根据当前类路径下的类，jar包来自动配置bean，如添加一个spring-boot-starter-web启动器就能拥有web的功能，无需其他配置。
4. 无代码生成和xml配置
5. 避免大量的mave导入和各种版本冲突。
6. 应用监控。

Springboot的自动配置原理

1. SpringBoot启动时会加载大量的自动配置类
2. 从类路径的META-INF/spring.factories中获取EnableAutoConfiguration指定的值。将这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作。
3. 我们看我们需要的功能有没有在SpringBoot默认写好的配置类中。
4. 再来看这个自动配置类配置了哪些组件
5. 给容器中自动配置类添加组件的时候，会从properties类中获取某个属性，只需要在配置文件中指定这些属性的值即可。xxxAutoConfiguration：自动配置类，xxxProperties：封装配置文件中的相关属性。

什么是CSRF攻击

什么是WebSockets

SpringBoot达成的jar和普通的jar有什么区别?

Dubbo

Dubbo的相关介绍

- Dubbo是阿里巴巴开源的一个基于Java的RPC框架。主要分为一下几个角色
- Consumer：需要调用远程服务的服务消费方
- Provider：服务提供方
- Container：服务运行的容器
- Monitor：监控中心
- Registry：注册中心
- 整体流程：服务提供方启动然后向注册中心注册自己能提供的服务。服务消费方启动向消费中心订阅自己所需的服务，然后注册中心提供元信息给服务消费方。因为服务消费方已经从注册中心获取了提供者的地址，因此可以通过负载均衡选择一个服务提供方进行直接调用。当服务消费方的元数据变更的话注册中心会把变更推送给服务消费方。服务提供方和消费者都会在内存中记录调用的次数和时间，然后定时的发送统计数据给监控中心。

服务暴露

服务的暴露起始于Spring IOC容器刷新完毕之后，会根据配置参数组装成URL，然后根据URL的参数进行本地调用或者远程调用。 会通过`proxtFactory.getInvoker`，利用javassist来进行动态代理，封装真正的实现类，然后再通过URL参数选择对应的协议来进行`protocol.export`，默认是Dubbo协议。 在第一次暴露的时候会调用`createServer`来创建Server，默认是NettyServer。 然后将export得到的exporter存入一个map中，供之后的远程调用查找，然后会向注册中心注册提供者的消息。

服务引用

服务的引入时机有两种，一种是饿汉式，一种是懒汉式。 饿汉式就是加载完毕就会引入，懒汉式是只有当这个服务被注入到其他类中时启动引入流程，默认是懒汉式。 会先根据配置参数组装成URL，一般而言我们都会配置注册中心，所以构建RegistryDirectory向注册中心注册消费者的消息，并且订阅提供者、配置、路由等节点。 得知提供者的信息之后就会进入Dubbo协议的引入，会创建Invoker，期间会包含NettyClient，来进行远程通信，最后通过Cluster来包装Invoker，默认是FailoverCluster，最后返回代理类。

服务调用

调用某个接口的方法会调用之前生成的代理类，然后会从cluster中经过路由的过滤、负载均衡选择一个Invoker发起远程调用，此时会记录此请求和请求的ID等待服务端的响应。 服务端接受请求之后会通过参数找到之前暴露存储的map，得到相应的exporter，然后最终调用真正的实现类，再组装好后结果返回，这个响应会带上之前请求的ID。 消费者收到这个响应之后会通过ID去找之前记录的请求，然后找到请求之后将响应塞到对应的future中，唤醒等待的线程，最后消费者得到响应，一个流程完毕。

SPI

SPI是Service Provider Interface，主要用于框架中，框架定义好接口，不同的使用者有不同的需求，因此需要有不同的实现，而SPI就通过定义一个特定的位置，约定在ClassPath的/META-INF/services/目录下创建一个以服务接口命名的文件，然后文件里面记录此jar包提供的具体实现类的全限定类名。所以就可以通过接口找到对应的文件，获取具体的实现类然后加载即可。

为什么Dubbo不用java的SPI，而要自己实现?

因为 Java SPI 在查找扩展实现类的时候遍历 SPI 的配置文件并且将实现类全部实例化，假设一个实现类初始化过程比较消耗资源且耗时，但是你的代码里面又用不上它，这就产生了资源的浪费。因此 Dubbo 就自己实现了一个 SPI，给每个实现类配了个名字，通过名字去文件里面找到对应的实现类全限定名然后加载实例化，按需加载。

Adaptive注解：自适应扩展

- 一个场景：首先我们根据配置来进行SPI扩展的加载，但是我不想在启动的时候让扩展被加载，我想根据请求时候的参数来动态选择对应的扩展。
- Dubbo通过一个代理机制实现了自适应扩展，简单地说就是为你想扩展的接口生成一个代理类，可以通过JDK或javassit编译你生成的代理类代码，然后通过反射创建实例。
- 这个实例里面的实现会根据本来方法的请求参数得知需要的扩展类，然后通过 `ExtensionLoader.getExtensionLoader(type.class).getExtension(从参数得到的name)`，来获取真正的实例调用。

如何设计一个RPC

- 首先需要实现高性能的网络传输，可以采用 Netty 来实现，不用自己重复造轮子，然后需要自定义协议，毕竟远程交互都需要遵循一定的协议，然后还需要定义好序列化协议，网络的传输毕竟都是二进制流传输的。
- 然后可以搞一套描述服务的语言，即 IDL（Interface description language），让所有的服务都用 IDL 定义，再由框架转换为特定编程语言的接口，这样就能跨语言了。
- 此时最近基本的功能已经有了，但是只是最基础的，工业级的话首先得易用，所以框架需要把上述的细节对使用者进行屏蔽，让他们感觉不到本地调用和远程调用的区别，所以需要代理实现。
- 然后还需要实现集群功能，因此的要服务发现、注册等功能，所以需要注册中心，当然细节还是需要屏蔽的。
- 最后还需要一个完善的监控机制，埋点上报调用情况等等，便于运维。

分布式

理论

拜占庭将军问题

CAP

- 一致性、可用性、分区容错性
- 在进行分布式系统设计和开发时，我们不应该仅仅局限在 CAP 问题上，还要关注系统的扩展性、可用性等等。在系统发生“分区”的情况下，CAP 理论只能满足 CP 或者 AP。要注意的是，这里的前提是系统发生了“分区”。如果系统没有发生“分区”的话，节点间的网络连接通信正常的话，也就不存在 P

了。这个时候，我们就可以同时保证 C 和 A 了。总结：如果系统发生“分区”，我们要考虑选择 CP 还是 AP。如果系统没有发生“分区”的话，我们要思考如何保证 CA 。

BASE

- 基本可用性(Basic Available)、软状态(Soft-state)、最终一致性(Eventually Consistent)
- 核心思想：即时无法做到强一致性，但每个应用都可以根据自身特点，采用适当的方式来使系统达到最终一致性。

1. 基本可用性 分布式系统在出现不可预测的故障的时候，允许损失部分可用性。但是，这绝不等价于系统不可用。 **允许损失部分可用性**
    - 响应时间上的损失：正常情况下，处理用户请求需要0.5s返回结果，但是由于系统故障，处理用户请求的时间变为3s。
    - 系统功能上的损失：正常情况下，用户可以使用系统的全部功能，但是由于系统访问量突然剧增，系统的部分非核心功能无法使用。
2. 软状态 系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不用节点的数据副本之间进行数据同步的过程存在延时。
3. 最终一致性 系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

分布式事务

概念

事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于不同的分布式系统的不同的节点上。

一次大的操作由多个小操作完成。这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小的事务要么全部成功，要么全部失败。本质上说分布式事务就是为了保证不同数据库的数据一致性。

2PC(二阶段提交)(同步阻塞协议)

- 是一种强一致性设计，引入了一个事务协调者的角色来协调管理各参与者的提交和回滚。二阶段是指准备和提交两个阶段。

1. 准备阶段：协调者会给参与者发送准备命令，准备命令理解成除了提交事务其他事情都做完了。同步等待全部资源响应之后就进入第二阶段，提交阶段。
2. 假如第二阶段的全部参与者都返回准备成功，那么协调者将向所有参与者发送提交命令，然后等待所有事务都提交成功之后，返回事务提交成功。
3. 假如第一阶段有参与者返回失败，那么协调者向所有参与者发送回滚事务的请求，即分布式事务执行失败。
4. 假如第二阶段有参与者返回失败，分两种情况，①回滚事务，不断重试，直到所有参与者回滚事务成功。②继续尝试提交事务，直到提交成功，最后实在不行人工参与处理。

- 适用于数据库层面的分布式事务。二阶段提交是阻塞同步的，阻塞同步就会导致长久资源的锁定，总体而言效率低，并且存在单点故障，极端情况下存在数据不一致的风险。

3PC(三阶段提交)

- 3PC的出现是为了解决2PC的一些问题，相比于2PC它在参与者中也引入了超时机制，并且新增了一个阶段使得参与者可以利用这个阶段统一各自的状态。
- 三个阶段：准备阶段，预提交阶段，提交阶段。准备阶段也只是询问参与者自身情况。
- 三阶段提交的阶段变更有什么影响？：①首先准备阶段变更成不会直接执行事务，而是先去询问参与者有没有条件去接这个事务。因此不会一来就干活直接锁资源，使得在某个资源不可用的情况下所有参与者都阻塞着。②预提交阶段的引入起到了一个统一状态的作用。它像一个栅栏，在预提交阶段之前所有参与者都未回应，在预提交阶段表名所有参与者都已经回应过了。但是多一个阶段的引入多一个交互，会造成性能上的损失，而且资源在绝大多数情况下都是可用的。
- 参与者超时会带来什么影响？：引入了超时机制，参与者就不会傻等了。如果等待提交命令超时了，那么参与者就会提交事务，绝大多数情况下是提交事务。如果等待预提交命令超时了，该干啥干啥，反正啥也没做。也可能带着数据不一致的问题。当等待提交命令超时，应该提交事务，有的可能执行回滚，导致数据不一致问题。
- 3PC解决的是提交阶段2PC协调者和某些参与者都挂了，选举之后的新的协调者不知道当前应该是提交还是回滚的问题。
- 通过预提交阶段可以减少故障恢复时的复杂性，但不能保证数据一致，除非挂了的那个参与者恢复。

TCC

- 2PC、3PC都是基于数据库层面的，TCC是基于业务层面的分布式事务。
- TCC包括Try、Confirm、Cancel三个步骤。Try指的是预留，即资源的预留和锁定。Confirm指的是确认操作，这一步其实就是真正的执行了。Cancel指的是撤销的操作，可以理解为把预留阶段的动作撤销了。存在一个事务管理者的身份，用来执行Confirm或Cancel操作。比如一个事务要执行A、B、C三个操作，那么先对三个操作执行预留动作。如果都预留成功了就执行确认操作，否则就全都执行撤销操作。

本地消息表(实现的最终一致性，容忍了数据暂时不一致性的情况)

- 本地消息表其实就是利用了各系统本地事务来实现分布式事务。
- 本地消息表就是一张存放本地消息的表，一般都是放在数据库中，然后在执行业务的时候**将业务的执行和将消息放入消息表中的操作放在同一个事务中**，这样就能保证消息放入本地表中的业务肯定是执行成功的。
- 然后再去调用下一个操作，如果下一个操作调用成功了，消息表的消息状态可以直接改成已成功。如果调用失败了，**后台任务定时去读取本地消息表**，筛选出还未成功的消息再调用对应的服务，服务更新成功了再变更消息的状态。

消息事务

一致性Hash算法

一致性Hash算法引入

在分布式集群中，对机器的添加删除，或者机器故障后自动脱离集群这些操作是分布式集群管理最基本的功能。如果采用常用的hash(object)%N算法，那么在有机器添加或者删除后，很多原有的数据就无法找到了，这样严重违反了单调性原则。

一致性Hash算法简介

一致性Hash算法提出了在动态变化的Cache环境中，判定哈希算法好坏的四个定义：

- 平衡性：哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间得到利用。
- 单调性：如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入系统，哈希的结果应能够保证原有已分配的内容能够被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。
- 分散性：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况应当避免，降低了系统存储的效率。分散性就是上述情况发生的严重程度。
- 负载：从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

一致性Hash算法

Paxos算法

- 一种基于消息传递且具有高度容错性的一致性算法
- 解决的问题：如何快速正确的在某个系统中对某个数值达成一致，并且保证不论发生任何异常，都不会破坏整个系统的一性。

Raft算法

ZAB算法

SnowFlake算法

SnowFlake，雪花算法是由Twitter开源的分布式ID生成算法，以划分命名空间的方式将64-bit位分割成多个部分，每个部分代表不同的含义。而Java中64bit的整数是long类型，所有在Java中SnowFlake算法开源的ID就是long来存储的。

- **第1位**占用1bit，其值始终是0，可看做是符号位不使用。
- **第2位开始的41位**是时间戳，41-bit位可表示2^41个数，每个数代表毫秒。
- **中间的10-bit位**可表示机器数，即2^10=1024台机器，但是一般情况下我们不会部署这么多台机器。如果我们对IDC(互联网数据中心)有需求，还可以将10-bit分5-bit给IDC，分5-bit给工作机器。这样就可以表示32个IDC，每个IDC下可以有32台机器。
- **最后12-bit**是自增序列，可表示2^12=4096个数。

这样的划分之后相当于**在一毫秒一个数据中心的一台机器上可产生4096个有序的不重复的ID**。但是我们 IDC 和机器数肯定不止一个，所以毫秒内能生成的有序ID数是翻倍的。

高并发

消息队列

削峰和解耦

读写分离 & 分库分表

读写分离主要是为了将数据库的读和写操作分布到不同的数据库节点上。主服务器负责写，从服务器负责读。另外，一主一从或者一主多从都可以。

读写分离可以大幅提高读性能，小幅提高写性能。因此读写分离更适合单机并发读请求比较多的情况。

分库分表是为了解决由于库、表数据量过大，而导致数据库性能持续下降的问题。

负载均衡算法

负载均衡系统通常用于将任务比如用户请求处理分配到多个服务器处理以提高网站、应用或者数据库的性能和可靠性。 常见的负载均衡系统包括3种：

1. DNS负载均衡：一般通过地理级别的均衡。
2. 硬件负载均衡：通过单独的硬件设备比如F5来实现负载均衡功能。
3. 软件负载均衡：通过负载均衡软件比如Nginx来实现负载均衡功能。

常见的负载均衡算法

常见的负载均衡算法包括：

- 轮询法(Round Robin)
- 加权轮询法(Weight Round Robin)
- 平滑加权轮询法(Smooth Weight Round Robin)
- 随机法(Random)
- 加权随机法(Weight Random)
- 源地址哈希法(Hash)
- 最小连接数法(Least Connections)

轮询法(Round Robin)

将请求按顺序轮流地分配给后端服务器上，它均衡地对待每一台服务器，而不关心服务器实际的连接数和当前的系统负载。

加权轮询法(Weight Round Robin)

不同的后端服务器可能机器的配置和当前系统的负载并不相同，因此他们的抗压能力也不相同。给配置高、负载低的机器配置更高的权重，让其处理更多的请求。而配置低、负载高的机器，给其分配较低的权重，降低其系统负载，加权轮询能很多的处理这一问题，并将请求顺序且按照权重分配到后端。

随机法(Random)

通过系统的随机算法，根据后端服务器的列表大小来随机选取其中的一台服务器进行访问。由概率统计理论可以得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配调用量到后端的每一台服务器，也就是轮询的结果。

加权随机法(Weight Random)

与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

源地址哈希法(Hash)

源地址哈希的思想是根据获取客户端的IP地址，通过哈希函数计算得到的一个数值，用该数值对服务器列表的大小进行取模运算，得到的结果便是客服端要访问服务器的序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会映射到同一台后端服务器进行访问。

最小连接数法(Least Connections)

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前的请求，尽可能地提高后端服务的利用效率，将请求合理地分流到每一台服务器。

Nginx的5种负载均衡算法

1. 轮询法
2. 加权轮询法
3. 源地址哈希法
4. fair(第三方)：按后端服务器的响应时间来分配请求，响应时间短的优先分配
5. url_hash(第三方)：按访问url的哈希结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。

高可用性

熔断

降级是应对自身系统的故障，熔断是用来应对当前依赖的外部系统或者第三方系统故障。

降级

从系统功能优先级的角度来考虑如何应对系统故障。 服务降级是指当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源来保证核心业务的正常运行。

限流

从用户访问压力的角度来考虑如何应对系统故障。 限流为了对服务端的接口接受请求的频率进行限制，防止服务挂掉。比如某一接口的请求限制为 100 个每秒, 对超过限制的请求放弃处理或者放到队列中等待处理。限流可以有效应对突发请求过多。

固定窗口计数器

- 固定窗口计时器的算法概念

1. 将时间划分为多个窗口
2. 在每个窗口内每有一个请求就将计数器加一
3. 如果计数器超过了限定数量，就将后来的请求丢弃。当时间到达下一个时间窗口时就将计时器重置。

- 这个算法有时候会让通过请求量允许为限制的两倍。例如限制一秒内允许通过5个请求，在最后半秒内到达了5个请求，下一个时间窗口前半秒又到达了5个请求，这样一秒内到达了10个请求。

滑动窗口计数器

- 滑动窗口计数器算法

1. 将时间划分为多个区间。
2. 在每个区间内每有一次请求就将计数器加一。维持一个时间窗口，占据多个区间。
3. 每经过一个区间的时间，就抛弃老的区间，加入最新的区间。
4. 如果当前窗口内区间的请求计数总和超出了限制数量，则本窗口内的所有请求都将被丢弃。

- 滑动窗口计数器通过将窗口再细分，并且按照时间滑动。这种算法避免了固定窗口计数器带来的双倍突发请求，但是滑动窗口的精度越高，算法所需的空间容量就越大。

漏桶

- 漏桶的算法概念

1. 每个请求都当做"水滴"放入"漏桶"中。
2. "漏桶"以一定的速率向外滴请求，如果"漏桶"空了则停止"漏水"。
3. 如果"漏桶"满了多余的"水滴"会被直接丢弃。

- 漏桶算法多用队列实现，服务的请求会存到队列中，服务的提供方则按照固定的速率从队列中取出请求并执行，过多的请求则放到队列中或直接拒绝。
- 漏桶算法的缺陷也很明显，当短时间内有大量突发请求时，即时此时服务器没有任何负载，每个请求也都得在队列中请求一段时间才能得到执行。

令牌桶

- 令牌桶的算法概念

1. 令牌以固定速率生成
2. 生产的令牌放入令牌桶中存放，如果令牌桶满了则多余的令牌会直接丢弃。当请求到达时，会尝试从令牌桶中去令牌，取到令牌的请求可以执行。
3. 如果令牌桶空了，那么尝试去令牌的请求会被直接丢弃。

- 令牌桶算法既能将所有请求平均分布到所有时间内，又能接受服务器能够承受范围内的突发请求。

排队

另类的一种限流，类比于现实世界的排队。玩过英雄联盟的小伙伴应该有体会，每次一有活动，就要经历一波排队才能进入游戏。

集群

相同的服务备份多份，避免单点故障。

超时和重试机制

一旦服务请求超过一段时间没有响应就结束此次请求并抛出异常。如果不进行超时设置可能会导致请求响应速度慢，甚至导致请求积压进而让系统无法再处理请求。

另外重试次数一般设为3次，再多次重试没有好处，反而会加重服务器压力。

分布式锁

分布式锁需要哪些特性

1. 互斥性
2. 可重入性
3. 锁超时：一旦锁超时即释放拥有的锁资源
4. 非阻塞：支持获取锁的时候直接返回结果值，而不是在没有获取到锁的时候阻塞线程的执行。
5. 公平锁和非公平锁

常见的分布式锁实现

基于MySQL数据库实现分布式锁

1. 悲观锁：利用select ... where ... for update排它锁。where后面的索引会锁行或者锁表，这样其他操作就会被阻塞。有些情况下，表不大的情况下，不会走索引。
2. 乐观锁：基于CAS的思想，是不具有互斥性的，不会产生锁等待而消耗资源，操作过程认为不存在并发冲突，只有update version失败后才会察觉到。抢购、秒杀就是使用了这种方式来防止超卖。 通过增加递增的版本号来实现乐观锁。 进程A

```
select version,account from personal_blank where id = "xxx";

# newAccount = 100 + 100;

update personal_blank set account=200,version = oldVersion + 1 where id =
"xxx" and version = oldVersion;

# 更新失败了, version != oldVersion
```

进程B

```
select version,account from personal_blank where id = "xxx";

# newAccount = 100 - 100;

update personal_blank set account=200,version = oldVersion + 1 where id =
"xxx" and version = oldVersion;

# 更新成功了
```

基于Redis实现的分布式锁

1. 使用命令介绍 (1) SETNX key value：当前仅当key不存在的时候，set一个key为value的字符串，如果当前key存在，什么都不做，返回0。

(2) expire key timeout：为key设置一个过期时间，单位为second，超过这个时间自动释放key，避免死锁。

(3) delete：删除key。 2. 实现思想 (1) 获取锁的时间通过SETNX进行加锁，并expire设置一个过期时间，超过这个时间自动释放key。key的value是一个随机生成的UUID，在释放锁的时候进行判断。

(2) 获取锁的时候还设置一个超时时间，超过这个时间自动释放这个key。

(3) 释放锁的时候通过UUID进行判断是不是持有该锁，如果持有该锁，使用delete释放该锁。

基于Zookeeper实现的分布式锁

1. 实现步骤 (1) 创建一个目录mylock

(2) 线程A想要获取锁，就在mylock目录下创建一个临时顺序节点。

(3) 获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，说明当前线程顺序号最小，获取锁。

(4) 线程B想要获取锁，在mylock目录下创建一个临时顺序节点，并获取所有比自己的小的兄弟节点，如果存在，监听比自己次小的节点。

(5) 线程A处理完成删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获取锁。

分布式锁的对比

基于数据库的分布式锁 缺点：

1. db操作性较差，并且有锁表的风险。
2. 非阻塞失败后，需要轮询，占用cpu资源。
3. 长时间不commit或者长时间轮询，可能会占用较多的资源。

基于Redis的分布式锁 缺点：

1. 锁删除失败，过期时间不好控制
2. 非阻塞，操作失败后，需要轮询，占用CPU资源。

基于Zookeeper的分布式锁 缺点：

1. 性能不如Redis，主要写操作要在leader上执行，然后同步到其他的follower上。

大数据处理

分治/Hash/排序

思路简介

> 先映射，后统计，最后排序

- 分而治之/hash映射：针对数据太大，内存受限，只能是把大文件划分成(映射取模)多个小文件。
- HashMap统计：当大文件划分成小文件后，可以使用hashmap进行频率统计。
- 堆/快速排序：统计完了之后就可以进行排序，得到次数最多的IP。

案例分析

海量日志数据，提取出某日访问百度次数最多的那个IP

寻找热门查询，300万个查询字符串中统计最热门的10个查询

有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。

海量数据分布在100台电脑中，想个办法高效统计出这批数据的TOP10。

有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序。

给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url?

怎么在海量数据中找出重复次数最多的一个?

上千万或上亿数据(有重复)，统计其中出现次数最多的前N个数据。

一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。

一个文本文件，找出前10个经常出现的词，但这次文件比较长，说是上亿行或十亿行，总之无法一次读入内存，问最优解

100w个数中找出最大的100个数。

Bitmap & Bloom Filter

双层桶划分

Trie树/数据库/倒排索引

外排序