# Total1



# 操作系统

# 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：用户态的程序就不能 随意操作内核地址空间，具有一定的安全保护作用。
- 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：进程在创建时需要申请一个空白PCB，向其中填写控制和管理进程的信息，完成资源分配。如果创建工作无法完成，比如资源无法满足，就无法被调度运行，把此时进程所处状态称为创建状态
- 就绪状态：进程已经准备好，已分配到所需资源，只要分配到CPU就能够立即运行
- 运行状态：进程处于就绪状态被调度后，进程进入执行状态
- 阻塞状态：正在执行的进程由于某些事件（I/O请求，申请缓存区失败）而暂时无法运行，进程受到阻塞。在满足请求时进入就绪状态等待系统调用
- 结束状态：进程结束，或出现错误，或被系统终止，进入终止状态。无法再执行

## 线程的6种状态

- 创建状态
- 运行状态
- 阻塞状态
- 等待状态
- 超时等待状态
- 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：两个主机之间的进程进行通信。

## 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

## 进程调度算法的优缺点

1. 时间片轮转调度算法（RR）：给每个进程固定的执行时间，根据进程到达的先后顺序让进程在单位时间片内执行，执行完成后便调度下一个进程执行，时间片轮转调度不考虑进程等待时间和执行时间，属于抢占式调度。优点是兼顾长短作业；缺点是平均等待时间较长，上下文切换较费时。适用于分时系统。
2. 先来先服务调度算法（FCFS）：根据进程到达的先后顺序执行进程，不考虑等待时间和执行时间，会产生饥饿现象。属于非抢占式调度，优点是公平，实现简单；缺点是不利于短作业。
3. 优先级调度算法（HPF）：在进程等待队列中选择优先级最高的来执行。
4. 多级反馈队列调度算法：将时间片轮转与优先级调度相结合，把进程按优先级分成不同的队列，先按优先级调度，优先级相同的，按时间片轮转。优点是兼顾长短作业，有较好的响应时间，可行性强，适用于各种作业环境。
5. 高响应比优先调度算法：根据“响应比=（进程执行时间+进程等待时间）/ 进程执行时间”这个公式得到的响应比来进行调度。高响应比优先算法在等待时间相同的情况下，作业执行的时间越短，响应比越高，满足段任务优先，同时响应比会随着等待时间增加而变大，优先级会提高，能够避免饥饿现象。优点是兼顾长短作业，缺点是计算响应比开销大。

## 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
3. 生产者消费者问题

# 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。

## 死锁和CPU利用率之间的关系

死锁和CPU利用率之间的关系与锁的实现有关系

1. 锁的实现形式不是CAS的时候，此时没有获取到锁，不会分配到cpu，此时cpu利用率是0.
2. 当锁的实现形式是CAS的时候，此时一直处于忙等待，这时候CPU利用率会飙升。

## 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

## 操作系统的内存分配算法有哪些?

1. 位图算法：概念： 这种位图即二维数组，通过二维数组来保存内存的使用情况，每个位的值代表这个块的使用情况，0为空闲，1为被占用。优点： 查找快；对于内存的管理比较清晰可见。缺点： 需要通过遍历的方式才能知道哪些内存是可以使用的，并且这种内存分配方式不能避免内存碎片的问题。
2. 链式算法：这种分配算法通过链表来保存和维护块的使用信息，它包括多个单元，每个单元是一个连续的数组，数组的第一位用来表示这个数组是用来表示的是已占用块信息还是空闲块信息，第二位是块的起始点，第三位表示从这个位置出发有多少个块是被占用（空闲）的，数组的最后一位存放的是指向下一个数组的首部地址的指针。当某个块可以直接被使用时，就可以修改它的状态并分配内存；如果只需要使用其中的一部分，那么可以把它拆开，分为两个链表元素，当被分配的块被释放后，再合并这两块内存。
3. 伙伴(buddy)算法：这种方式基于数组结合链表来实现，它解决了链式算法查找慢的问题，伙伴算法对不同大小的内存进行分类管理，可以根据内存分配的需求直接定位到相应位置，提升内存查找速度。伙伴算法的数据结构是通过一个free_area的数组，找到合适的数量级的位置，这个位置维护着一段链表，如果对应链表元素可用，就进行分配，同时还可以对剩下的内存进行拆分，并把它们调整到低数量级，可用于分配小对象，当这块内存对应区域被释放后，就可以和相邻的内存区间合并，再添加到高一数量级的数组位置。

4. slab算法：slab在一定程度上优化了伙伴算法的小对象浪费内存的问题，slab分配器在操作系统初始化的时候就维护了一个缓存区域，维护了一个slabs列表，每个slab中存储着若干对象，用来缓存内核中需要频繁创建和销毁的小对象，对它们完成初始化操作，并放置在缓存中，以实现通用对象的复用。

## 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：**时间换空间的概念**。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：用户的实际应用多种多样，这就要求应用层采取不同的应用协议来解决不同类型的应用要求。
- 传输层：为端到端提供可靠的传输服务（为不同主机的进程间提供通信服务），为端到端连接提供流量控制、差错控制、服务质量、数据传输管理等服务。
- 网络层：把网络层的协议数据单元从源端传到目的端，为分组交换网上的不同主机提供通信服务。对分组进行路由选择，并实现流量控制、拥塞控制、差错控制和网际互联等功能。
- 数据链路层：将网络层传来的IP数据包组装成帧。差错控制、流量控制和传输管理等。
- 物理层：在物理媒体上为数据端设备透明的传输原始比特流。

# 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回答报文放入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。

## 重定向与转发

1. 重定向：客户端行为，两次请求，显示新的地址，请求域中数据会丢失。
2. 转发：服务器行为，只有一次请求，地址栏不会变化，请求域中数据不会丢失。

## 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
   - Token
     - token的组成部分：头部：类型及其签名使用的算法。载荷：该JWT的签发者、该JWT所面向的用户、接收JWT的一方，什么时候过期，在什么时候签发的。签名：将头部和载荷分别进行BASE64编码之后得到两个字符串，然后再将两个字符串用英文句号连接在一起，之后用头部的算法将拼接后的字符串进行加密。
     - 优点：①可扩展性好，应用程序分布式部署的情况下，session需要做多机数据共享，通常可以存放在数据库或者redis里面。而jwt不需要。②无状态，jwt不在服务器端存储任何状态。
     - 缺点：①安全性，因为载荷使用base64编码，并没有加密，因此jwt中不能存储敏感信息。而session存储在服务端相对来说安全一些。②性能：jwt太长，所有数据都放在jwt中，经过编码之后的jwt非常长，cookie的限制一般是4k，cookie很可能放不下，所以jwt一般放在local storage里面。每次请求都会带上jwt，导致请求的头部比body还要大，而sessionId只是很短的一个字符串，因此使用jwt的http请求比使用sessionId的开销大得多。③一次性：jwt是一次性的，想修改里面的内容必须重新签发一个jwt。
   - Session和Token的对比
     - 区别：用户状态保存的位置不同，session保存在服务端，jwt保存在客户端。

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
   HTTP/1.1 相比 HTTP/1.0 性能上的改进：使用 TCP 长连接的方式改善了 HTTP/1.0 短连接造成的性能开销。支持 管道（pipeline）网络传输，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。

但 HTTP/1.1 还是有性能瓶颈：请求 / 响应头部（Header）未经压缩就发送，首部信息越多延迟越大。只能压缩 Body 的部分；发送冗长的首部。每次互相发送相同的首部造成的浪费较多；服务器是按请求的顺序响应的，如果服务器响应慢，会招致客户端一直请求不到数据，也就是队头阻塞；没有请求优先级控制；请求只能从客户端开始，服务器只能被动响应。

10. HTTP和HTTPS的区别
    1. HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。

2. HTTP 连接建立相对简单， TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
3. HTTP 的端口号是 80，HTTPS 的端口号是 443。
4. HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

11. HTTP缓存
12. get post put的概念区别

- get 方法一般用于请求，比如你在浏览器地址栏输入 www.cxuanblog.com 其实就是发送了一个 get 请求，它的主要特征是请求服务器返回资源，而 post 方法一般用于表单的提交，相当于是把信息提交给服务器，等待服务器作出响应，get 相当于一个是 pull/拉的操作，而 post 相当于是一个 push/推的操作。
- put客户端向服务端传输文件
- 安全和幂等的概念：安全是请求方法不会破坏服务器上的资源。幂等是多次执行相同的操作，结果都是相同的。
- get方法是安全幂等的，post方法不是安全和幂等的。

12. HTTP1.0和HTTP1.1的比较

- 使用TCP长连接的方式改善了1.0短连接造成的性能开销。
- 支持管道网络传输，只要第一个请求发送出去了，不必等待其回来，就可以发第二个请求出去，可以减少整体的响应时间。
- 缺点：头部未经压缩就发送出去，首部信息较多延迟越大，只能压缩body部分。发送冗长的头部，每次互相发送相同的头部造成的浪费较多。服务器是按照请求的顺序响应的，如果响应慢，会导致客户端一直请求不到数据，队头阻塞。没有请求优先级控制。请求只能从客户端开始，服务器只能被动的响应。

13. HTTP1.1和HTTP2.0的比较

- 使用HPACK算法，在客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送相同字段了，只发送索引号。头部压缩
- 二进制格式，增加了数据传输的效率。
- 数据流：数据包不是按顺序发送的，同一个连接里面连续的数据包可能属于不同的回应。因此必须要对数据包做标记，指出它属于哪个回应。每个数据流都标记着一个独一无二的编号，其中规定客户端发出的数据流编号为奇数，服务器发出的数据流编号为偶数。
- 客户端还可以指定数据流的优先级。优先级高的请求，服务器就先响应该请求。
- 一个连接中并发多个请求或回应，而不是按照顺序一一对应。不需要排队等待，降低了延迟，避免了队头阻塞的问题，大幅度提高了连接的利用率。
- 服务器推送。
- 缺点：因为多个HTTP请求在复用一个TCP连接，下层的TCP协议是不知道有多少个HTTP请求的，如果发生了丢包，就会触发TCP的重传机制，这样在一个TCP连接中的所有HTTP请求都必须等待这个丢了的包被重传回来。

14. HTTP2.0和HTTP3.0的比较

- 基于HTTP2.0的问题，HTTP3.0把HTTP下层的TCP改成了UDP。基于UDP的QUIC协议可以实现类似TCP的可靠传输协议。
- QUIC有自己的一套机制可以保证传输的可靠性。当某个流发生丢包时，只会阻塞这个流，其他流不会受到影响。
- TLS升级成最新的1.3版本，头部压缩算法也升级成了QPack。
- HTTPS要建立一个连接，要花费6次交互，先是建立三次握手，然后是TSL/1.3的三次握手。QUIC直接把以往的TCP和TLS/1.3的6次交互合并成了3次，减少了交互次数。

15. HTTP常见的请求头

- 通用标头：Date(创建报文的日期时间)，Cache-Control(控制缓存的行为)，Connection(持久性连接、非持久性连接)
- 实体标头：Content-Encoding(编码方式)，Content-Language(客户端或服务端能够接受的语言)，Content-Length(实体主体的大小)，Content-Type(媒体类型),Expires(实体主题过期的日期时间),Last-Modified(资源最后修改日期时间)。
- 请求标头：Accept(用户代理可处理的媒体类型)，Accept-Charset(优先的字符集)，Accept-Encoding(优先的内容编码)，If-Modified-Since(比较资源的更新时间，200，304)，If-None-Match(比较实体标记)。
- 响应标头：Keep-Alive(Connection非持久连接的存活时间，可以进行指定)，ETag(资源的匹配信息)。

## 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，cwnd从1开始。
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。**快重传：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，进入快速恢复算法。快恢复：cwnd = ssthresh + 3，重传丢失的数据包，如果接收到重复的ACK，那么cwnd+1，如果接收到新数据的ack，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

### 组网划分

## 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



## 数据库三范式

1. 第一范式：属性不可分。
2. 第二范式：符合1NF，并且非主属性完全依赖码。
3. 第三范式：符合2NF，并且消除传递依赖。非主属性不传递函数依赖于键码。

## 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。

## 多个数据库表怎么去处理事务

- 我们可以通过SQL的事务来对相关数据库操作进行处理，在开始conn.setAutoCommit(false);（conn是或得的连接）把本次运行的SQL操作改为非自动运行，在配置好各SQL语句之后，调用conn.commit();来运行，其中通过try{……}catch……来捕捉异常，如果遇到错误时，就调用conn.rollback();来对本次操作进行回滚到操作前的状态，防止存在错误数据和脏数据。
- 聊一聊分布式事务

## 数据库死锁如何解决?

死锁是指多个事务在同一资源上相互占用并请求锁定对方占用的资源而导致恶性循环的现象。当多个事务试图以不同顺序锁定资源时就可能会产生死锁，多个事务同时锁定同一个资源时也会产生死锁。

InnoDB 目前处理死锁的方法是将持有最少行级排它锁的事务进行回滚。

死锁发生之后，只有部分或者完全回滚其中一个事务，才能打破死锁。对于事务型系统这是无法避免的，所以应用程序在设计时必须考虑如何处理死锁。大多数情况下只需要重新执行因死锁回滚的事务即可。

# 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。

## 为什么非叶子节点只存储索引值

1. 保持一致性：当数据库表进行DML操作时，同一行记录的页地址会发生改变，因非主键索引保存的是主键的值，无需进行更改。
2. 节省存储空间：Innodb数据本身就已经汇聚到主键索引所在的B+树上了， 如果普通索引还继续再保存一份数据，就会导致有多少索引就要存多少份数据。

## 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，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。
    - 例如新建a,b,c这样的联合索引，查询条件a = ? and b > ? and c = ?这样的情况下当查询出b的范围后，不会再走索引，而是按照这样的范围进行查找。

### 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。

### 主键索引和唯一索引的区别

主键是一种约束，唯一索引是一种索引，两者在本质上是不同的。主键创建后一定包含一个唯一性索引，唯一性索引并不一定就是主键。唯一性索引列允许空值，而主键列不允许为空值。主键列在创建时，已经默认为空值 + 唯一索引了。主键可以被其他表引用为外键，而唯一索引不能。一个表最多只能创建一个主键，但可以创建多个唯一索引。主键更适合那些不容易更改的唯一标识，如自动递增列、身份证号等。

### 性别字段为什么不适合做索引

因为你访问索引需要付出额外的IO开销，你从索引中拿到的只是地址，要想真正访问到数据还是要对表进行一次IO。假如你要从表的100万行数据中取几个数据，那么利用索引迅速定位，访问索引的这IO开销就非常值了。但如果你是从100万行数据中取50万行数据，就比如性别字段，那你相对需要访问50万次索引，再访问50万次表，加起来的开销并不会比直接对表进行一次完整扫描小。
如果把性别字段设为表的聚集索引，那么就肯定能加快大约一半该字段的查询速度了。聚集索引指的是表本身中数据按哪个字段的值来进行排序。因此，聚集索引只能有一个，而且使用聚集索引不会付出额外IO开销。

### delete和alter触发什么锁?

1. 在delete时，如果where字段存在中存在某个索引，则会触发行级锁，如果锁定的行过多时会触发表级锁。
2. alter table时，会停止整个表的读入。触发表级锁。

# 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。

## mysql缓存机制

- 缓存sql文本和缓存结果，用KV形式保存在服务器内存中，如果运行相同的sql，服务器直接从缓存中去获取结果，不需要再去解析、优化、执行sql。如果这个表改变了，那么使用是这个表中的所有缓存将不再有效，查询缓存值的相关条目将被清空。对于频繁更新的表，查询缓存不合适，对于一些不变的数据且有大量相同sql查询的表，查询缓存会节省很大的性能。
- 命中条件：缓存存在一张hash表中，通过查询sql，查询数据库，客户端协议等作为key，在判断命中前，mysql不会解析sql，而是使用sql去查询缓存，sql上的任何字符的不同都会导致缓存不命中。如果查询有不确定的数据，那么查询完后结果不会被缓存。
- InnoDB查询缓存：InnoDB会对每个表设置一个事务计数器，里面存储着当前最大的事务ID。当一个事务提交时，InnoDB会使用MVCC中系统事务ID最大的事务ID更新当前表的计数器。只有比这个最大ID大的事务才能使用查询缓存，其他比这个ID小的事务则不能使用查询缓存。另外，在InnoDB中，所有加锁的操作的事务都不能使用任何查询缓存。查询必须是完全相同的(逐字节相同)才能够被认为是相同的。另外，同样的查询字符串由于其它原因可能认为是不同的。使用不同的数据库、不同的协议版本或者不同默认字符集的查询被认为是不同的查询并且分别进行缓存。

## 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。

## drop delete truncate

1. drop table：1)属于DDL。2)不可回滚。3)不可带where。4)表内容和结构删除。5)删除速度快
2. truncate table：1)属于DDL。2)不可回滚。3)不可带where。4)表内容删除。5)删除速度快
3. delete from：1)属于DML。2)可回滚。3)可带where。4)表结构在，表内容要看where执行的情况。5)删除速度慢,需要逐行删除

- 不再需要一张表的时候，用drop。想删除部分数据行时候，用delete，并且带上where子句。保留表而删除所有数据的时候用truncate。

## DDL，DML，DCL

1. DDL：Create、Drop和Alter
2. DML：insert、update，select、delete
3. DCL：grant授权，revoke：废除权限

## 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。(账单流水业务)
- 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。

# IO



## IO模型简介

## IO分类

- 磁盘操作：file
- 字节操作：InputStream和OutputStream
- 字符操作：Reader和Writer
- 对象操作：Serializable
- 网络操作：Socket

### java中为什么有了字节流还要有字符流?

字节流是java虚拟机将字节转换得到的，这个过程还算是非常耗时，并且如果我们不知道编码类型，就很容易出现乱码问题。所以IO流就直接提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话用字符会比较好。

## 五种IO模型

- 阻塞式IO(BIO)
- 非阻塞式IO(NIO)
- IO复用(select和poll)
- 信号驱动式IO
- 异步IO(AIO)

## 输入操作两个阶段

- 等待数据准备好
- 从内核向进程复制数据

## 套接字输入操作

- 等待数据从网络中达到，当所等待分组到达时，它被复制到内核中的某个缓冲区。
- 把数据从内核缓冲区复制到进程缓冲区。

## BIO

- 同步阻塞IO模式，数据的读取写入必须阻塞在一个线程内等待完成。应用程序被阻塞，直到数据复制到应用程序缓冲区中才返回。
- 这里阻塞过程中，其他程序还可以继续执行。
- 对于十万甚至百万级连接时，传统的BIO模型是无能为力的。
- 一请求一应答通信模型

## NIO

- 同步非阻塞IO模式，在java1.4中引入了NIO框架，对于java.nio包，提供了Channel、Selector、Buffer等抽象。
- 支持面向缓冲区，基于通道的I/O操作方法。
- 提供了 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，也支持阻塞模式，相对于BIO中的 `Socket` 和 `ServerSocket` 。
- 非阻塞模式对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发效率和更好的维护性。对于高负载和高并发的应用，使用NIO的非阻塞模式来开发。

### NIO的特性、NIO与IO的区别

- **IO流是阻塞的，NIO是非阻塞的**

1. 单线程中从通道读取数据到buffer，同时可以继续做别的事情，当在通道中读取到buffer的时候，线程再继续处理数据。
2. 非阻塞写，一个线程写一些数据到通道中，不需要等待它完全写入，就可以去做别的事情。

- **Buffer(缓冲区)**

1. IO是面向流的，NIO是面向缓冲区的
2. Buffer是一个对象，它包含一些要写入或读出的数据。NIO引入了Buffer这个对象，这是和IO一个重要的区别。
3. IO是Stream oriented，虽然Stream也有Buffer开头的扩展类，但是只是流的包装类，最终还是从流到缓冲区。
4. Buffer是直接将数据读到缓冲区中进行处理的。

- **Channel(通道)**

1. 通道是双向的，可读可写，但流的读写是单向的。无论读写，通道都是和Buffer交互。因为Buffer，通道可以异步的读写。

- **Selector(选择器)**

1. NIO有选择器，IO没有选择器
2. 选择器用于单线程处理多个通道。对于操作系统来说，线程之间的切换是比较昂贵的，所以需要使用较少的线程来处理多个通道，所以使用选择器对于提高操作系统的效率是比较有用的。

# AIO

- 异步非阻塞的I/O模型，基于事件和回调机制实现的。

# 阻塞/非阻塞 异步/同步

- 同步和异步是通信机制，阻塞和非阻塞是调用状态。
- 阻塞是在调用revfrom的时候，因为还没有数据准备好，线程阻塞，直到数据准备好再返回。非阻塞是在调用revfrom的时候，不论数据有没有准备好，都返回一个状态码。
- 同步是在数据准备好后，调用revfrom的线程发起系统调用，将数据从内核复制到用户空间。异步是当数据返回好后，主动将数据从内核复制到用户空间。然后返回一个状态码给相应的线程。

# NIO

## 流与块

- IO与NIO最重要的区别就是数据打包方式和传输方式。IO以流的方式处理数据，而NIO以块的方式处理数据。
- 面向流的IO一次处理一个字节数据：一个输入流产生一个字节数据，一个输出流产生一个字节数据。
- 面向块的IO一次处理一个数据块，按块处理数据比流处理数据要快得多。

## 通道与缓冲区

- **通道**
  通道Channel是对原IO包中流的模拟，可以通过它读取和写入数据。通道和流的不同之处在于通道是双向的，可以用于读或写或者同时读写。流是单向的。
- **缓冲区**
  发送给一个通道的所有数据必须先放到缓冲区中，同样的，从通道中读取的任何数据都要先读到缓冲区中，也就是说不会直接对通道进行读写数据，而是要先经过缓冲区。
  缓冲状态变量：capacity 最大容量，position 已经读写的字节数，limit 还可以读写的字节数。
  缓冲区实际上是一个数组，但它不仅仅是一个数组，缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

## 选择器

- **选择器** NIO实现了IO多路复用中的Reactor模型，一个线程Thread使用一个选择器Selector通过轮询的方式监听多个通道Channel上的事件，从而让一个线程就可以处理多个事件。
  通过配置监听的通道Channel为非阻塞，那么当Channel上的IO事件还未到达时，就不会进入阻塞状态一直等待，而是继续轮询其他Channel，直到IO事件已经到达的Channel执行。

# 多路IO复用

## Reactor模型

- 并发读写
  Reactor分为mainReactor和subReactor，mainReactor主要进行客户端的连接的处理，处理完成之后将该连接交由sunReactor以处理客户端的网络读写。这里的subReactor则是使用一个线程池来支撑的，其读写能力将会随着线程数的增多而大大增加。对于业务操作，则是使用一个线程池，每个业务请求都只需要进行编解码和业务计算。

# 零拷贝

1. mmap + write(两次次系统调用+三次拷贝)
   mmap()系统调用函数会直接把内核缓冲区里的数据 映射 到用户空间，这样，操作系统内核与用户空间就不需要再进行任何的数据拷贝操作。
   具体过程如下：

- 应用程序调用了mmap()后，DMA会把磁盘的数据拷贝到内核的缓冲区里。接着，应用程序和操作系统 共享 这个缓冲区。

- 应用程序再调用write()，操作系统直接把内核缓冲区的数据拷贝到socket缓冲区中，这一切都发生在内核态，由CPU来搬运数据。
- 最后把内核的socket缓冲区里的数据，拷贝到网卡的缓冲区里，这个过程是由DMA搬运的。

仍需要通过CPU把内核缓冲区的数据拷贝到socket缓冲区里，仍然需要4次上下文切换，因为系统调用还是2次。

2. sendfile(一次系统调用+三次拷贝)

```
#include <sys/socket.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

前两个参数是目的端和源端的文件描述符，后两个参数是源端的偏移量和复制数据的长度，返回值是实际复制数据的长度。

它可以替代前面的read()和write()这两个系统调用，这样就可以减少一次系统调用，也就减少了两次上下文切换的开销。

该系统调用可以直接把内核缓冲区里的数据拷贝到socket缓冲区里，不再拷贝到用户态，这样就只有2次上下文切换和3次数据拷贝。

(一次系统调用+两次拷贝)
如果网卡支持SG-DMA技术，可以进一步减少通过CPU把内核缓冲区里的数据拷贝到socket缓冲区的过程。

在linux内核2.4版本开始，对于支持网卡支持SG—DMA技术的情况下，sendfile()系统调用的过程具体如下：

- 第一步：通过DMA将磁盘上的数据拷贝到内核缓冲区里。
- 第二步：缓冲区描述符和数据长度传到socket缓冲区，这样网卡的SG-DMA控制器就可以直接将内核缓冲中的数据拷贝到网卡的缓冲区里，此过程不需要将数据从操作系统内核缓冲区拷贝到socket缓冲区中，这样就减少了一次数据拷贝。

3. 大文件使用异步 + 直接IO，不使用零拷贝，因为零拷贝会用到PageCache(磁盘高速缓存)，PageCache主要用来就IO的数据进行合并以及预读。将大文件放到PageCache中会造成大文件很快填满PageCache，造成很多小的热点文件不能被读到。(将数据拷贝到内核缓冲区中，这个内核缓冲区说的就是磁盘高速缓存)。

## select、poll、epoll

### select

```
int select(int maxfdp, fd_set *readset, fd_set *writeset,
           fd_set *exceptset, struct timeval *timeout);
```

- 基本原理：select函数监视的文件描述符分为三类，writefds，readfds和exceptfds。调用select时会被阻塞，直到有fd就绪或者超时，函数返回一个大于0的值，遍历文件描述符fd_set，来找到就绪的描述符。
- maxfdp是一个整数值，是指集合中所有文件描述符的范围，即所有最大文件描述符+1。fd_set是以位图的形式存储这些文件描述符，maxfdp定义了位图中有效的位的个数。
- 时间复杂度O(n)，n为文件描述符fd_set的大小。文件描述符最大限制1024。

### poll

```
int poll ( struct pollfd * fds, unsigned int nfds, int timeout);
```

- poll定义了一个pollfd结构的数组，用于存放需要检测其状态的所有文件描述符，调用poll的时候不会清空这个数组，特别对于文件描述符比较多的情况，在一定程度上提高了查询的效率。这和select有很大的不同，在每次调用select之后，都会清空它检测的文件描述符集合，导致每次调用select都必须把文件描述符重新加入到待检测的文件描述符集合中。因此select适合只检测一个文件描述符的情况，而poll适合检测大量文件描述符的情况。
- 时间复杂度：O(n)，n是文件描述符集合的大小。文件描述符没有最大限制。

### epoll

- 将所有文件描述符存储在共享内存中(用户空间和内核空间可以直接访问)。调用epoll_create函数创建epoll对象，并以红黑树的结构存储在内核空间，epoll_ctl函数用来在红黑树中添加或注销监视文件描述符，最后调用epoll_wait函数直到有就绪的文件描述符，立即返回给用户态进程。
- 时间复杂度O(1)，文件描述符最大限制能打开的fd的上限远大于1024。
- epoll的两种工作方式：LT：水平触发，若就绪事件一次没有处理完所有要做的事件，就会一直处理，即就会把没有处理完的事件继续放回就绪队列中，一直进行处理。ET：边缘触发，就绪处理事件只处理一次，若没有处理完，会在下次事件就绪时继续处理。如果后续没有就绪的事件，那么剩余的那部分数据也会随之而丢失。

## select、poll、epoll的区别

1. select 应用场景 select 的 timeout 参数精度为 1ns，而 poll 和 epoll 为 1ms，因此 select 更加适用于实时要求更高的场景，比如核反应堆的控制。**select 可移植性更好，几乎被所有主流平台所支持**。
2. poll 应用场景 poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。 需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。 需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll_ctl() 进行系统调用，频繁系统调用降低效率。并且epoll 的描述符存储在内核，不容易调试。
3. epoll 应用场景 只需要运行在 Linux 平台上，并且有非常大量的描述符需要同时轮询，而且这些连接最好是长连接。
4. 传递方式：select和poll都是内核到用户，epoll则是共享内存。
5. 在select和poll中，进程只有在调用方法后，内核才对所有监视的文件描述符进行扫描，发现有任何一个文件描述符就绪或者超时，就会立即返回。而epoll则是基于事件的就绪通知方式，事先通过epoll_ctl来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait时就会得到通知。

# Java基础

# 为什么说java是编译和解释并存?

编译是指编译器是针对特定的操作系统将源码一次性翻译成可被平台执行的机器码。解释是指解释器对源码逐行解释成特定平台的机器码并立即执行。
java要先经过编译再经过解释两个步骤，由java编写的程序首先经过编译步骤，生成字节码文件，这种字节码文件必须由java解释器来解释执行。

# == 和 equal的区别?

== 对于普通类型来说是进行值的比较，对于引用类型来说是进行地址的比较。
equals 不能用于普通类型的比较，只能用来判断两个对象是否相等。对于没有重写equal的类，这时候equal和==的作用是相同的，比较的是两个对象的地址。对于覆盖了equal的类，比较的是equal方法中的，覆盖了equal方法，必须覆盖hashcode，hashcode的作用是获取哈希码，这个hash码的作用是确定该对象在哈希表中的位置，在使用hashset时，使用hashcode来确定对象加入的位置，如果发现有hashcode相同的对象，再调用equals来判断两个对象是否相等。如果两个对象相等，那么他们的hashcode一定是相等的。两个对象相等，对两个对象分别调用equals分别返回true。但是两个对象hashcode相等，他们也不一定是相等的。因此equals被覆盖过，hashcode也必须被覆盖。如果没有覆盖hashcode，这两个对象是不可能相等的，即使两个对象指向了相同的数据。

# 为什么java只有值传递没有引用传递?

值传递：方法调用得到的是调用者提供的值，引用传递：方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，但是不能修改传递值调用所对应的变量值。java总是按值调用，方法得到的是所有参数值的一个拷贝。

# 深拷贝和浅拷贝

浅拷贝：对于基本数据类型进行值传递，对于引用数据类型进行引用传递般的拷贝。
深拷贝：对于基本数据类型进行值传递，对于引用数据类型，创建一个新对象，并复制其内容。

## 深拷贝怎么实现?

1. 实现Clonable接口，并且重写Object类中的clone()方法。
2. 实现Serializable接口序列化。

# 重载和重写的区别?

重载就是同一个方法能够根据输入数据的不同，做出不同的处理。
重写是当子类继承父类的的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类的方法。方法的重写要遵循两同两小一大。两同：方法名相同，形参列表相同。两小：子类的返回类型要比父类的返回类型更小或相等。子类方法声明抛出的异常要比父类方法声明抛出的异常更小或相等。一大：子类方法的访问权限要比父类方法的访问权限更大或相等。

# 四个修饰符

1. public
2. private
3. protected：包内调用和子类调用
4. 默认：包内调用

# String、StringBuffer和StringBuilder的区别

## String不可变的原因、好处

原因：Java9之后使用byte数组存储字符串，被声明为final，这意味着 value 数组初始化之后就不能再引用其它数组。并且 String 内部没有改变 value 数组的方法，因此可以保证 String 不可变。

- 好处：

1. 可以缓存hash值：String的hash值被经常使用（例如用作Hashmap的key），保证不变hash值不会改变，只需要计算一次。
2. String Pool：String对象创建过，会到StringPool中引用。（StringPoll：创建字符串时，如果存在池中，则返回现有字符串的引用，不创建新对象）。
3. 安全：String经常作为参数，不可改变保证参数不便。（例如网络连接参数）
4. 线程安全：不可变，天生线程安全，不用考虑同步问题。

## StringBuilder、StringBuffer

StringBuffer是线程安全的，类中的方法都添加了**synchronized关键字**，也就是给这个方法添加了一个锁，用来保证线程安全。，而StringBuilder则没有实现线程安全功能，所以性能略高。

# new String在内部存储结构发生什么?

使用这种方式创建两个对象（StringPool中没有）。

1. “abc”是字符串字面量，编译时会在StringPool中创建一个对象，指向字面量。
2. 使用new的方式会在堆中创建一个字符串对象。

# abstract 和 interface

1. abstract：含有abstract修饰符的类即为抽象类。抽象类不能有实例对象。含有abstract的抽象方法的类必须声明为抽象类。抽象类中的方法不一定都是抽象方法。abstract类中的抽象方法必须在子类中实现，所以不能有抽象静态方法和抽象构造方法。如果子类中没有实现所有的abstract方法，子类也必须被声明为abstract类。接口可以说成是抽象类的一个特例，接口中的所有方法必须为抽象方法。接口中的成员方法定义为public abstract类型，所有的成员变量默认定义为public static final。
2. abstract和interface的区别：①接口中不能有构造方法，抽象类中可以有构造方法。②抽象类中可以有普通成员变量，接口中不可以有普通成员变量。③抽象类中可以包含非抽象的普通方法，接口中只能包含抽象方法，不能包含非抽象的普通方法。④抽象类中的抽象方法可以使public和protected，接口中的抽象方法只能是public的，默认是public abstract。⑤抽象类中可以有静态方法，接口中不能有静态方法。⑥抽象类中可以包含任意类型的静态成员变量，接口中只能包含public static final的静态成员变量，并且默认为public static final。⑦一个类可以实现多个接口，但只能继承一个类。
3. 接口更多的是在系统架构设计方面发挥作用，定义模块之间的通信契约。抽象类更多的是在代码实现方面发挥作用，可以实现代码复用。
4. 模板方法设计模式是抽象类的一个典型应用。

# 你知道的内部类有哪些java当中的?

1. AQS中的node和conditionobject
2. 建造者模式中的内部类
3. ReentranLock的内部类，线程安全集合中的内部类node。
4. HashMap
5. ThreadLocal

# 静态内部类和内部类有什么区别?

## 普通内部类

1. 普通内部类中不能有静态成员变量和静态成员方法
2. 想要引用内部类，必须创建外部类的引用，即内部类不能离开外部类而独立存在
3. 可以内部类理解为外部类的一个成员，一个内部类对象可以访问创建它的外部类对象的内容，需要注意的是内部类里的一个成员变量与外部类的一个成员变量同名，也即外部类的同名成员变量被屏蔽，那么如果需要访问外部类的成员变量可以通过：外部类名.this.变量名
4. 所有的内部类都需要注意其构造方法，所有的内部类都没有无参的构造器，系统都会为其增加一个默认的构造参数。因为所有的内部类都有一个默认的构造参数，其值为外部类对象，这样也就更好的理解了对于外部类。

### 静态内部类

1. 普通静态和非静态类的区别就在于静态内部类没有了指向外部的引用。
2. 普通静态内部类如果想要使用外部类中的成员（包括属性和方法），那么要求对应的成员是static的。
3. 静态内部类是属于相应的外部类的静态成员。

### 初始化的时候静态内部类什么时候初始化?

静态内部类的加载不需要依附外部类，在使用时才加载。不过在加载静态内部类的过程中也会加载外部类。

### 为什么静态方法调用非静态方法是错误的?

1. 静态方法属于类，当调用非静态方法时，不知道调用哪个对象的非静态方法。
2. 在加载过程来看，静态方法在类加载的时候被保存在方法区中，非静态方法随着对象的创建被保存在堆中。非静态方法可以直接通过类名进行访问，此时不需要类的实例化，也就是说非静态方法是不存在的，对于访问一个不存在的方法，就会报错。

## 面向对象和面向过程的区别

面向过程：面向过程性能比较高。因为类调用时需要实例化，开销比较大，比较消耗资源，所有当性能是最重要的考量因素的时候。面向过程没有面向对象易维护、易复用、易扩展等。
面向对象：面向对象易维护、易复用、易扩展。因为面向对象有封装、继承、多态的特性，所有可以设计出低耦合的系统，是系统更加灵活、更加易于维护。但是，面向对象性能比面向过程低。

面向过程让计算机有步骤地顺序做一件事，是过程化思维，使用面向过程语言开发大型项目，软件复用和维护存在很大问题，模块之间耦合严重。面向对象相对面向过程更适合解决规模较大的问题，可以拆解问题复杂度，对现实事物进行抽象并映射为开发对象，更接近人的思维。

例如开门这个动作，面向过程是 open(Door door)，动宾结构，door 作为操作对象的参数传入方法，方法内定义开门的具体步骤。面向对象的方式首先会定义一个类 Door，抽象出门的属性（如尺寸、颜色）和行为（如 open 和 close），主谓结构。

面向过程代码松散，强调流程化解决问题。面向对象代码强调高内聚、低耦合，先抽象模型定义共性行为，再解决实际问题。

## 面向对象的三大特征?

封装
继承
多态：编译时多态，主要指方法的重载。运行时多态：程序中定义的对象引用所指向的具体类型在运行期间才确定。运行时多态的三个条件：继承，覆盖(重写)，向上转型

## 异常

异常分为error和exception。都继承Throwable。exception能被程序本身处理，而error不能被程序本身处理，只能尽量避免。
exception分为checkexception和uncheckexception。IOexception为checkexception，在编写代码的时候必须被捕获。而RuntimeException为uncheckexception，在编写代码的时候可以不被捕获。
不受检查的异常：NullPointException、ClassCastException、NumberFormatException、ArrayOutOfBoundException、ArithmeticException。
受检查的异常：FileNotFoundException、SQLException。

## try catch finally

在以下三种特殊情况下finally不会被执行：System.exit()在finally之前。CPU关闭，线程死亡。

## Object有什么方法，什么时候用到hashcode方法?

1. clone，getClass，toString，finalize，equals，hashcode，wait，notify，notifyAll
2. 在重写equals的时候，必须重写hashcode。首先用hashcode判断两个类是否相等，接着在用equals判断。

# 反射

1. 可以获取一个类的所有属性和方法，还可以调用这些属性和方法。
2. 缺点：让我们在运行时有了分析操作类的能力，同样增加了安全问题比如无视泛型参数的安全检查。

# cglib反射和jdk反射的区别

- JDK是基于反射机制,生成一个实现代理接口的匿名类,然后重写方法,实现方法的增强。它生成类的速度很快,但是运行时因为是基于反射,调用后续的类操作会很慢。而且他是只能针对接口编程的.
- CGLIB是基于继承机制,继承被代理类,所以方法不要声明为final,然后重写父类方法达到增强了类的作用。它底层是基于asm第三方框架,是对代理对象类的class文件加载进来,通过修改其字节码生成子类来处理。生成类的速度慢,但是后续执行类的操作时候很快。可以针对类和接口。
- 因为jdk是基于反射,CGLIB是基于字节码.所以性能上会有差异。

# 泛型

泛型提供了编译时类型检查机制，该机制允许程序员在编译时检测到非法类型。泛型的本质是参数化类型，也就是说操作的数据类型被指定为一个参数。

# SPI机制

1. 实现原理：
    1. 首先，ServiceLoader实现了Iterable接口，实现了迭代器的hasNext和next方法。这里主要都是调用的lookupIterator的相应hasNext和next方法，lookupIterator是懒加载迭代器。 其次，LazyIterator中的hasNext方法，静态变量PREFIX就是”META-INF/services/”目录。 最后，通过反射方法Class.forName()加载类对象，并用newInstance方法将类实例化，并把实例化后的类缓存到providers对象中，(LinkedHashMap<String,S>类型）然后返回实例对象。
    2. 所以我们可以看到ServiceLoader不是实例化以后，就去读取配置文件中的具体实现，并进行实例化。而是等到使用迭代器去遍历的时候，才会加载对应的配置文件去解析，调用hasNext方法的时候会去加载配置文件进行解析，调用next方法的时候进行实例化并缓存。 所有的配置文件只会加载一次，服务提供者也只会被实例化一次，重新加载配置文件可使用reload方法。
2. SPI机制的缺陷：
    1. 不能按需加载，需要遍历所有的实现，并实例化，然后在循环中才能找到我们需要的实现。如果不想用某些实现类，或者某些类实例化很耗时，它也被载入并实例化了，这就造成了浪费。
    2. 获取某个实现类的方式不够灵活，只能通过Iterator形式获取，不能根据某个参数来获取对应的实现类。
    3. 多个并发多线程使用 ServiceLoader 类的实例是不安全的。
3. SPI和API的区别
    1. SPI：接口位于调用方所在的包中。概念上更依赖调用方。组织上位于调用方所在的包中。实现位于独立的包中。常见的例子是：插件模式的插件。
    2. API：接口位于实现方所在的包中。概念上更接近实现方。组织上位于实现方所在的包中。实现和接口在一个包中。

# 什么是序列化，什么是反序列化

如果我们需要持久化java对象必须把java对象保存到文件中或者在网络中传输java对象，都需要用到序列化。

1. 序列化：把数据结构或对象转换成二进制字节流的过程。
2. 反序列化：把二进制字节流转换成数据结构或对象的过程。
   如果不想序列化可以使用transient关键字。transient只能修饰变量，不能修饰类和方法。在反序列化的过程中，被transient修饰的变量值会被置为类型的默认值。static变量属于类，不属于某个对象，所以不论是否被transient修饰，都不会被序列化。

# java中IO流分为几种?

# 既然有了字节流为什么还要字符流?

字符流是java虚拟机将字节转换得到的，这个过程是十分耗时的，如果我们不知道编码类型很容易出现乱码问题。所以IO流就直接提供了一个操作字符的接口，方便我们平时对字符进行流处理。对于音频，图片等媒体文字采用字节流比较合适，如果涉及到字符的话使用字符流比较好。

# socket套接字

套接字是网络编程中的一种通信机制，是支持TCP/IP的网络通信的基本操作单元，可以看做是不同主机的进程进行双向通信的端点。是通信双方的一种约定，用套接字的相关函数来完成通信过程。处于应用层和传输层之间。

# 常用注解说一下

- spring中的各种注解
- Java自带的标准注解，包括@Override、@Deprecated和@SuppressWarnings，分别用于标明重写某个方法、标明某个类或方法过时、标明要忽略的警告，用这些注解标明后编译器就会进行检查。
- 元注解，元注解是用于定义注解的注解，包括@Retention、@Target、@Inherited、@Documented，@Retention用于标明注解被保留的阶段，@Target用于标明注解使用的范围，source，源代码，class，编译期保留，runtime：运行期保留，可利用反射获取注解信息。@Inherited用于标明注解可继承，@Documented用于标明是否生成javadoc文档。
- 注解不支持继承，不能使用关键字extends来继承某个@interface，但注解在编译后，编译器会自动继承java.lang.annotation。Annotation接口. 虽然反编译后发现注解继承了Annotation接口，请记住，即使Java的接口可以实现多继承，但定义注解时依然无法使用extends关键字继承@interface。 区别于注解的继承，被注解的子类继承父类注解可以用@Inherited： 如果某个类使用了被@Inherited修饰的Annotation，则其子类将自动具有该注解。
- 注解是一个接口，是一个继承自Annotation的接口。里面的每一个属性其实就是接口的一个抽象方法。注解@interface 是一个实现了Annotation接口的接口， 然后在调用getDeclaredAnnotations()方法的时候，返回一个代理$Proxy对象，这个是使用jdk动态代理创建，使用Proxy的newProxyInstance方法，传入接口 和InvocationHandler的一个实例(也就是 AnotationInvocationHandler )，最后返回一个实例。期间，在创建代理对象之前，解析注解时候从该注解类的常量池中取出注解的信息，包括之前写到注解中的参数，然后将这些信息在创建 AnnotationInvocationHandler时候 ，传入进去作为构造函数的参数。

# 面向对象编程的五大原则

1. 单一职责原则：一个类只做一件事，一个类应该只有一个引起它修改的原因。
2. 开闭原则：一个软件实体如类、模块和函数应该对修改封闭，对扩展开放。
3. 里氏替换原则：子类应该完全替换父类。应该在使用继承的时候，只扩展新功能，不破坏父类原有的功能。
4. 依赖倒置原则：细节应该依赖于抽象，抽象不应该依赖于细节。把抽象层放到程序设计的高层，并保持稳定，程序的细节变化由低层的实现层来完成。
5. 迪米特法则：最少知道原则，一个类不应该直到自己操作类的细节。
6. 接口隔离原则：客户端不应依赖它不需要的接口。如果一个接口在实现时，部分方法由于冗余被客户端空实现，应该将接口拆分，让实现类只需依赖自己需要的接口方法。

# 设计模式

## 构建型模式

### 工厂方法模式

#### 简单工厂模式

将实例化的工作交给工厂来做，我们只需要告诉工厂我们需要什么东西就好了。
缺点：如果需要生产的产品过多，此模式会导致工厂类过于庞大，承担过多的职责，变成超级类。修改此工厂的原因不止一个，违背了**单一工厂原则**。当要生产新的产品时，必须在工厂类中添加新的分支，违背了**开闭原则**。

#### 工厂方法模式

为了解决简单工厂模式的两个弊端，规定每个产品都有一个专属工厂。

#### 抽象工厂模式

在创建时指定了具体的工厂类后，在使用时就无需关心是哪个工厂类，只需要将此工厂当做抽象的IFactory接口使用即可。
缺点：抽象工厂模式太重了，如果IFactory接口需要新增功能，则会影响到所有的具体工厂类。
抽象工厂模式适合新增同类工厂这样的横向扩展，不适合新增功能这样的纵向扩展。

### 单例模式

某个对象全局只需要一个实例时，就可以使用单例模式，能够避免对象的重复创建，节约空间提升效率。避免由于操作不同实例导致的逻辑错误。分为饿汉式和懒汉式。**构造方法为private**，这样就保证了其他类无法实例化此类，必须通过getInstance方法才能获取到唯一的instance实例。

```
public class Singleton {
    //私有化构造方法
    private Singleton() {
    }

    //在类中创建一个对象
    public volatile static Singleton instance;

    //添加公开的方法，返回这个对象
    public static Singleton getInstance() {
        if (instance == null) {//DCL--Double Check Lock 双重检查锁,提高效率
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                    //1 在堆中开辟空间，属性默认值
                    //2 调用构造方法，给属性赋值
                    //3 把地址放入局部变量
                }
            }
        }
        return instance;
    }
}
```

### 建造型模式

用于创建过程稳定，但配置多变的对象。将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。静态内部类Builder，链式调用生成不同的配置。构造方法也是私有的，只能通过静态内部类来创建实例。

### 原型模式

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

## 结构型模式

### 适配器模式

将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的类能一起工作。

### 桥接模式

面向两个或多个同等级的接口。将抽象部分与实现部分分离，二者相互独立，实现抽象化与实现化解耦。 比如对形状和颜色进行分离，然后根据需要对颜色和形状进行组合。

### 组合模式

组合模式用于整体和部分的结构。当整体和部分有相似的结构，在操作时可以被一直对待时，就可以使用组合模式。

### 装饰器模式

用于增强一个类原有的功能，为添加一个类新的功能。

### 代理模式

给某一个对象提供一个代理，并由代理对象控制原对象的引用。

## 行为型模式

### 责任链模式

责任链主要用于助力职责相同，程度不同的类。 比如在对不同难度的任务进行处理的时候，可以将初级工程师、中级工程师、高级工程师这三种角色用链串联起来，然后将任务在初级工程师开始，谁有能力处理就在谁那处理。

### 观察者模式

一个对象发生一个事件后，逐一通知监听这个对象的监听者。监听者可以马上对这个事件做出反应。

### 模板方法模式

定义了一个操作中的算法的骨架，而将一些操作延迟到子类，模板方法使得子类可以不改变一个算法的结构即可重新定义该算法的某些步骤。
如果不希望子类覆写模板中的某些方法，使用final修饰该方法，如果希望子类中必须覆写模板中的某些方法，使用abstract修饰该方法。如果没有特殊要求，可使用protected或public修饰该方法。子类可以根据实际情况考虑是否覆写。

# bitset、bitmap

## bitmap算法

- bitmap的基本思想就是用一个bit位来标记某个元素对应的Value，而Key即是该元素。由于采用了Bit为单位来存储数据，可以很大力度的节省空间，常用于对大量整数做去重和查询操作。
- 32位机器上，对于一个整型数，比如int a=1 在内存中占32bit位，这是为了方便计算机的运算。但是对于某些应用场景而言，这属于一种巨大的浪费，因为我们可以用对应的32bit位对应存储十进制的0-31个数，而这就是Bit-map的基本思想。Bit-map算法利用这种思想处理大量数据的排序、查询以及去重。
- 在20亿个随机整数中找出某个数m是否存在其中，并假设32位操作系统，4G内存。如果将20亿个整数放入内存中，需要占用多少内存？如果每个数字都以int类型存储，20亿整数占用内存=20亿*4byte/1024/1024/1024 约等于 7.45G。如果使用bit-map的思想，按位存储，即一位就可以代表一个数字。20亿整数占用内存=20亿*1bit/8/1024/1024/1024=约等于0.233G。从上述结果来看，按位存储比按字节存储数字节约了（7.45/0.233-1约等于31倍空间），而且按字节存储根据内存4G的要求无法一次性在内存中进行处理。
- 快速去重：20亿个整数中找出不重复的整数的个数，内存不足以容纳这20亿个整数。 首先，根据“内存空间不足以容纳这20亿个整数”我们可以快速的联想到Bit-map。下边关键的问题就是怎么设计我们的Bit-map来表示这20亿个数字的状态了。其实这个问题很简单，一个数字的状态只有三种，分别为不存在，只有一个，有重复。因此，我们只需要2bits就可以对一个数字的状态进行存储了，假设我们设定一个数字不存在为00，存在一次01，存在两次及其以上为11。那我们大概需要存储空间2G左右。接下来的任务就是把这20亿个数字放进去（存储），如果对应的状态位为00，则将其变为01，表示存在一次；如果对应的状态位为01，则将其变为11，表示已经有一个了，即出现多次；如果为11，则对应的状态位保持不变，仍表示出现多次。最后，统计状态位为01的个数，就得到了不重复的数字个数，时间复杂度为O(n)。

## bitset

BitSet实现了一个位向量，它可以根据需要增长。每一位都有一个布尔值。一个BitSet的位可以被非负整数索引。可以查找、设置、清除某一位。通过逻辑运算符可以修改另一个BitSet的内容。默认情况下，所有的位都有一个默认值false。

## 布隆过滤器

1. 首先需要 k 个 hash 函数，每个函数可以把 key 散列成为 1 个整数；
2. 初始化时，需要一个长度为 n 比特的数组，每个比特位初始化为 0；
3. 某个 key 加入集合时，用 k 个 hash 函数计算出 k 个散列值，并把数组中对应的比特位置为 1；
4. 判断某个 key 是否在集合时，用 k 个 hash 函数计算出 k 个散列值，并查询数组中对应的比特位，如果所有的比特位都是1，认为在

## 怎么解决hash冲突

开放地址法，拉链法。hashmap中使用拉链法，threadlocal中使用开放地址法。

# tomcat出问题了，怎么通过日志排查

# tomcat线程安全问题

# 红黑树

1. 每个节点要么是红色，要么是黑色。
2. 根节点必须是黑色
3. 红色节点不能连续(也即是，红色节点的孩子和父亲都不能是红色)。
4. 对于每个节点，从该点至null(树尾端)的任何路径，都含有相同个数的黑色节点.

# java集合

# ArrayList、Vector、LinkedList

## ArrayList的扩容机制

1. 有参构造方法，创建指定容量大小的数组。无参构造方法则是创建默认大小10的数组。
2. `ensureCapacity`，供外部使用，在进行插入的时候，如果每次只插入一个，那么频繁的插入就导致频繁的拷贝，降低性能，为了避免这种情况，在插入之前先调用这个方法，以减少增量重新分配的次数。
3. `ensureCapacityInternal`，得到最小扩容量，调用 `ensureExplicitCapacity` 来判断是否需要扩容。
4. `ensureExplicitCapacity`，如果最小扩容量比当前数组中的容量大小大，则调用 `grow` 进行扩容。
5. `grow`，每次扩容进行1.5倍的扩容操作。
6. 在进行第一次插入的时候会进行扩容，因为之前初始化的是一个空的数组，长度为0。

## ArrayList和LinkedList的区别

1. 是否保证线程安全： ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；
2. 底层数据结构： Arraylist 底层使用的是 Object 数组；LinkedList 底层使用的是 双向链表 数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别.）
3. 插入和删除是否受元素位置的影响： ① ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。 比如：执行add(E e)方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（add(int index, E element)）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。② LinkedList 采用链表存储，所以对于add(E e)方法的插入，删除元素时间复杂度不受元素位置的影响，近似 O（1），如果是要在指定位置i插入和删除元素的话（(add(int index, E element)） 时间复杂度近似为o(n))因为需要先移动到指定位置再插入。
4. 是否支持快速随机访问： LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于get(int index)方法)。
5. 内存空间占用： ArrayList的空间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

## ArrayList和Vector的区别，为什么要用ArrayList取代Vector?

1. Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。
2. Arraylist不是同步的，所以在不需要保证线程安全时建议使用Arraylist。

# HashMap、TreeMap、HashTable、LinkHashMap

## HashMap底层实现

JDK1.8 之前HashMap底层是数组和链表结合在一起使用也就是链表散列。1.8之后通过数组链表和红黑树的结合。HashMap通过key的hashCode 经过扰动函数处理过后得到hash值，然后通过 (n - 1) & hash 判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。
所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法扰动函数之后可以减少碰撞。

## HashMap的长度为什么是2的幂次方

取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说hash%length==hash&(length-1)的前提是length是2的n次方；）。并且 采用二进制位操作&，相对于%能够提高运算效率，这就解释了HashMap的长度为什么是2的幂次方。

## 负载因子为什么是0.75?

负载因子是0.75的时候，空间利用率比较高，而且避免了相当多的Hash冲突，使得底层的链表或者是红黑树的高度比较低，提升了空间效率。

## HashMap和HashTable的区别

1. 线程是否安全： HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过synchronized 修饰。
2. 效率： 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。
3. 对Null key 和Null value的支持： HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键所对应的值为 null。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。hashmap在put空值时做了特殊处理。这是因为Hashtable使用的是安全失败机制（fail-safe），这种机制会使你此次读到的数据不一定是最新的数据。如果你使用null值，就会使得其无法判断对应的key是不存在还是为空，因为你无法再调用一次contain(key）来对key是否存在进行判断，ConcurrentHashMap同理。
4. 初始容量大小和每次扩充容量大小的不同： ①创建时如果不指定容量初始值，**Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的tableSizeFor()方法保证）**。也就是说 HashMap 总是使用2的幂作为哈希表的大小。
5. 底层数据结构： JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制

## LinkHashMap

LinkedHashMap拥有HashMap的所有特性，它比HashMap多维护了一个双向链表，因此可以按照插入的顺序从头部或者从尾部迭代，是有序的，不过因为比HashMap多维护了一个双向链表，它的内存相比而言要比 HashMap 大，并且性能会差一些。

## TreeMap

底层实现红黑树，继承Map接口，不允许出现重复的key，可以插入null键，null值，可以对元素进行排序。

## ConcurrentHashMap 和 Hashtable 的区别

1. 底层数据结构： JDK1.7的 ConcurrentHashMap 底层采用分段的数组+链表实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
2. 实现线程安全的方式（重要）： ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。② Hashtable(同一把锁) :使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

## comparable 和 Comparator的区别

- comparable接口实际上是出自java.lang包 它有一个 compareTo(Object obj)方法用来排序。
- comparator接口实际上是出自 java.util 包它有一个compare(Object obj1, Object obj2)方法用来排序。
  一般我们需要对一个集合使用自定义排序时，我们就要重写compareTo()方法或compare()方法，当我们需要对某一个集合实现两种排序方式，比如一个song对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写compareTo()方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 Collections.sort()。

## TreeSet

TreeSet对TreeMap做了一层包装，也就是说TreeSet里面有一个TreeMap(适配器模式)。

# java多线程

# 并发出现问题的根源：并发三要素

1. 原子性：分时复用引起的，操作系统增加了进程、线程以分时复用CPU，进而均衡CPU与IO设备之间的速度差异。引出了原子性问题
2. 可见性：CPU缓存引起的，CPU增加了缓存，以均衡和内存之间的速度差异，引出了可见性的问题。
3. 有序性：执行重排序引起的，编译程序优化指令执行执行次序，使得缓存能够得到更合理的利用。引出了有序性的问题。编译器优化重排序，指令级并行重排序，内存系统重排序。

# JAVA是怎么解决并发问题的

1. volatile，synchronized，final三个关键字。
2. happen-before规则
   ①单一线程原则，在一个程序内，程序前面的操作先于发生程序后面的操作。
   ②管程锁定规则，一个unlock操作先行发生于后面对同一个锁的lock操作。
   ③volatile变量规则，对于一个volatile变量的写操作先行发生于后面对这个变量的读操作。
   ④线程启动规则，Thread对象的start发生先行发生于此线程的每一个动作。
   ⑤线程加入规则，Thread对象结束先行发生于join方法返回。
   ⑥线程中断规则，对线程interrupt方法的调用先行发生于被中断线程的代码检测到中断事件的发生。可以通过 interrupted() 方法检测到是否有中断发生。

⑦对象终结规则，一个对象的初始化先行发生于它的finalize方法的开始。
⑧传递性，如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A先行发生于操作C。

# 线程安全

不可变，绝对线程安全，相对线程安全，线程兼容，线程对立

## 线程安全的实现方法

1. 互斥同步：Synchronized，Reentrantlock
2. 非阻塞同步：CAS
3. 无同步方案：线程隔离ThreadLocal

# 线程状态转换

新建状态，就绪状态，运行状态，阻塞状态，结束状态

# 线程使用方式

1. 实现 Runnable 接口
2. 实现 Callable 接口
3. 继承 Thread 类

# 线程互斥同步

## Synchronized和Reentranlock的对比

1. 锁的实现 synchronized 是 JVM 实现的，而 ReentrantLock 是 JDK 实现的。
2. 性能 新版本 Java 对 synchronized 进行了很多优化，例如自旋锁等，synchronized 与 ReentrantLock 大致相同。
3. 等待可中断 当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。 ReentrantLock 可中断，而 synchronized 不行。
4. 公平锁 公平锁是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁。 synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但是也可以是公平的。
5. 锁绑定多个条件 一个 ReentrantLock 可以同时绑定多个 Condition 对象。

# synchronized

1. 使用方法，对象锁，this，普通成员方法上。类锁，静态方法，当前类

## 加锁和释放锁的原理：

1. 使用synchronized之后，会在编译之后在同步的代码块前后加上monitorenter和monitorexit字节码指令，他依赖操作系统底层互斥锁实现。他的作用主要就是实现原子性操作和解决共享变量的内存可见性问题。执行monitorenter指令时会尝试获取对象锁，如果对象没有被锁定或者已经获得了锁，锁的计数器+1。此时其他竞争锁的线程则会进入等待队列中。执行monitorexit指令时则会把计数器-1，当计数器值为0时，则锁释放，处于等待队列中的线程再继续竞争锁。**从内存语义来说，加锁的过程会清除工作内存中的共享变量，再从主内存读取，而释放锁的过程则是将工作内存中的共享变量写回主内存**。
2. 深入底层源码：synchronized实际上有两个队列waitSet和entryList。当多个线程进入同步代码块时，首先进入entryList。有一个线程获取到monitor锁后，就赋值给当前线程，并且计数器+1。如果线程调用wait方法，将释放锁，当前线程置为null，计数器-1，同时进入waitSet等待被唤醒，调用notify或者notifyAll之后又会进入entryList竞争锁。如果线程执行完毕，同样释放锁，计数器-1，当前线程置为null。

## 锁升级过程

无锁，偏向锁，轻量级锁，重量级锁

## 锁优化

1. 自旋锁，自适应自旋锁，锁消除，锁粗化，偏向锁，轻量级锁

2. 自旋锁：由于大部分时间锁被占用的时间很短，共享变量的锁定时间也很短，没必要挂起线程，用户态和内核态的来回上下文切换严重影响性能。让线程执行一个忙循环，防止用户态进入内核态。默认10次。
3. 自适应自旋锁：自选的时间不是固定的，而是由前一次在同一个锁上的自选时间和锁持有者的状态决定的。
4. 锁消除：检测到一些同步代码块完全不存在数据竞争的场景，也就不需要加锁，就会进行锁消除。
5. 锁粗化：有很多操作都是对同一个对象加锁，就会把锁的同步范围扩展到整个操作序列之外。
6. 偏向锁：当线程访问同步块获取锁时，会在对象头和栈帧中的锁记录里面存储偏向锁的线程ID，之后这个线程再次进入代码块时，都不需要CAS来加锁和解锁了，偏向锁会永远偏向第一个获得锁的线程，如果后续没有线程去竞争这个锁，持有锁的线程永远不需要进行同步。反之当有其他线程竞争当前偏向锁时，持有偏向锁的线程就会释放当前偏向锁。
7. 轻量级锁：在线程栈帧中创建锁记录的空间用于存储当前锁对象markword的拷贝，JVM会使用CAS操作将markword拷贝到锁记录中，并将markword指向当前线程栈帧中的锁记录的指针。这个对象就拥有了当前对象的锁，并将markword中的锁标志位更新为00，表示此对象处于轻量级锁定状态。如果操作失败，jvm会检查当前markword是否已经指向了当前线程栈帧中的锁记录空间，如果已指向，可以直接调用，如果没有指向，那么就说明该锁被其他线程抢占了，如果有两条以上的线程竞争该锁，轻量级锁失效，直接膨胀为重量级锁。轻量级锁解锁时，会原子的使用CAS将线程栈帧中的markword拷贝替换回对象头中，如果失败，说明当前锁存在竞争关系，直接膨胀为重量级锁。
8. 偏向锁就是通过对象头的偏向线程ID来对比，甚至都不需要CAS了，而轻量级锁主要就是通过CAS修改对象头锁记录和自旋来实现，重量级锁则是除了拥有锁的线程其他全部阻塞。

# volatile

## 可见性

总线嗅探机制，CPU需要每时每刻监听总线上的一切活动，总线嗅探只是保证了某个CPU核心的Cache更新数据这个事件能被其他的CPU核心知道，但不能保证事务串行化。volatile修饰的变量会直接强制刷回主存，此时缓存中的该变量失效，读取时重新在主存中读取该变量。

## MESI机制

- 四个状态：已失效、独占、共享、已修改

### 有了MESI为什么还需要Volatile

1. 不止操作系统做了指令的重排序，编译器和虚拟机都做了指令的重排序，所以需要volatile。
2. mesi只是保证多核cpu的独占cache之间的一致性，但是cpu的并不是直接把数据写入L1 cache的，中间可能还有store buffer，因此有mesi机制是远远不够的。
3. mesi协议最多只是保证了对于一个变量，在多个核上的读写顺序，对于多个变量而言是没有任何保证的。
4. mesi对于这种弱一致性的cpu来说，只会保证指令之间的有比如控制依赖，数据依赖，地址依赖等等依赖关系的指令间的提交的先后顺序，而对于完全没有依赖关系的指令，它们是不会保证执行提交的顺序的，除非你使用了volatile，java把volatile编译成arm和power能够识别的barrier指令，这时候才是按照顺序的。

## 有序性

内存屏障，storestore 禁止上面的普通写和下面的volatile写重排序。，storeload防止上面的 volatile 写与下面可能有的 volatile 读/写重排序。，loadload，loadstore禁止下面所有的普通读写操作和上面的 volatile 读重排序。

# final

## 知识点

1. final修饰的方法是可以被重载的
2. final不都是编译器常量，当修饰的变量指向一个随机数时，只有当前变量被初始化后无法被更改。

## 有序性

1. JMM禁止编译器把final域的写重排序到构造函数之外。
2. JMM禁止编译器把final域的读重排序到构造函数之外。
3. 在构造函数内对一个final修饰的对象的成员域的写入，与随后在构造函数之外把这个被构造的对象的引用赋给一个引用变量，这两个操作是不能被重排序的。

# CAS，Unsafe和原子类

1. CAS：CAS叫做CompareAndSwap，比较并交换，主要是通过处理器的指令来保证操作的原子性，它包含三个操作数：变量内存地址，V表示。旧的预期值，A表示。准备设置的新值，B表示。当执行CAS指令时，只有当V等于A时，才会用B去更新V的值，否则就不会执行更新操作。
2. 缺点：ABA问题。循环时间长。只能保证一个共享变量的原子操作：只对一个共享变量操作可以保证原子性，但是多个则不行，多个可以通过AtomicReference来处理或者使用锁synchronized实现。
3. Unsafe：compareAndSwapInt,compareAndSwapObject,compareAndSwapLong。
4. 原子类：AtomicInteger,AtomicBoolean,AtomicLong,AtomicIntegerArray,AtomicLongArray,AtomicReference,AtomicReference,AtomicStampedReference,AtomicMarkableReference,AtomicIntegerFieldUpdater,AtomicLongFieldUpdater,AtomicStampedFieldUpdater,AtomicReferenceFieldUpdater。

# LockSupport

1. LockSupport用来创建锁和其他同步类的基本线程阻塞原语。简而言之，当调用LockSupport.park时，表示当前线程将会等待，直至获得许可，当调用LockSupport.unpark时，必须把等待获得许可的线程作为参数进行传递，好让此线程继续运行。

# 几种对比

2. Thread.sleep()和Object.wait()的区别：①释不释放锁资源。②必须传入时间，另一个可不传。③sleep唤醒后继续执行，对于wait不传时间的必须通过notify或notifyAll来唤醒，带时间的立刻获取锁或没有立刻获取锁，进入同步队列。
3. Thread.sleep()和Condition.await()的区别：Object.wait()和Condition.await()的原理是基本一致的，不同的是Condition.await()底层是调用LockSupport.park()来实现阻塞当前线程的。 实际上，它在阻塞当前线程之前还干了两件事，一是把当前线程添加到条件队列中，二是“完全”释放锁，也就是让state状态变量变为0，然后才是调用LockSupport.park()阻塞当前线程。
4. Thread.sleep()和LockSupport.park()的区别：①都是阻塞当前线程的执行，且不会释放当前线程所占用的锁资源。②Thread.sleep()无法从外部唤醒，只能自己醒过来，park可被unpark唤醒。③Thread.sleep()需要捕获异常，park不需要捕获异常。④Thread.sleep()是一个native方法，park调用底层的unsafe的native方法。
5. Object.wait和LockSupport.park的区别：①wait需要在同步代码块中执行，park可以在任意地方执行。②wait需要捕获异常，park不需要捕获异常。③wait不带超时的，必须由notify唤醒，不一定执行后面的操作，park不带超时的必须由unpark唤醒，一定执行后面的操作。④park和unpark可以换序执行。
6. LockSupprt.park不会释放当前锁资源，锁资源的释放是由Condition.await()中实现的。

# AQS

1. AQS的思想：如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设定为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时所分配的机制。这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。
2. AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。
3. AQS的模板方法 `isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回`
4. Node的waitStatus：表示节点等待在队列中的状态。 `CANCELLED` ： 表示线程取消了等待。如果取得锁的过程中发生了一些异常，则可能出现取消的情况，比如等待过程中出现了中断异常或者出现了timeout, `SIGNAL` ： 表示后继节点需要被唤醒, `CONDITION` ： 线程等待在条件变量队列中, `PROPAGATE` ： 在共享模式下，无条件传播releaseShared状态, `0` ： 默认状态。
5. 加锁释放锁过程：
    1. **加锁** ①首先尝试获取锁， `tryAcquire()` ，如果抢到锁，将state设置1，独占线程设置为自己。如果没有抢到锁，则将当前线程包装成Node类型，放入同步队列中。②首先将节点快速加入队尾，如果加入不成功，通过调用 `enq()` 方法进行入队。入队过程中，如果同步队列中是空的，则首先创建一个空节点，接着将当前节点的前驱指向空节点，并让 `tail` 指向新创建的节点。③接着调用 `acquireQueued()` 方法，如果当前节点的前驱节点是 `head` 节点，并且尝试获取锁成功，则设置当前节点为头结点，gc前驱节点。如果当前节点的前驱节点不是 `head` 节点，则判断当前节点是否需要挂起，如果需要则将当前线程挂起。直到有其他线程唤醒当前线程，for循环进行再次进行判断。
    2. **解锁**当线程释放资源时，调用 `tryRelease()` 方法，将CAS将state置为初始值，并将独占线程设置为null。然后调用 `unparkSuccessor()` 方法，唤醒后继线程，从后向前找，找到位于最前的节点的 `waitStatus` 值小于等于0的，然后将该节点的线程唤醒。被唤醒的节点继续执行 `acquireQueued()` 方法中的逻辑。
6. 非公平锁：当某一个节点被唤醒时，恰好有一个节点还没入队，抢到了锁，这时候被唤醒的锁再次被挂起。
7. 公平锁：在 `tryAcquire()` 的时候，会调用 `hasQueuedPredecessors()` 来判断队列中是否有等待时间更长的节点，如果没有的话才将独占线程设置为自己，如果有的话将自己加入等待队列尾部。

# ReentrantLock

1. ReentrantLock类内部总共存在Sync、NonfairSync、FairSync三个类，NonfairSync与FairSync类继承自Sync类，Sync类继承自AbstractQueuedSynchronizer抽象类。

2. AQS怎么实现的非公平锁，怎么实现的公平锁

# ReentrantReadWriteLock

1. Sync继承自AQS、NonfairSync继承自Sync类、FairSync继承自Sync类；ReadLock实现了Lock接口、WriteLock也实现了Lock接口。
2. 高16位为读锁，低16位为写锁
3. 锁升降级：锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住(当前拥有的)写锁，再获取到读锁，随后释放(先前拥有的)写锁的过程。

# CopyOnWriteArrayList

# ConcurrentHashMap

1.7使用Segment+HashEntry分段锁的方式实现，1.8则抛弃了Segment，改为使用CAS+synchronized+Node实现，同样也加入了红黑树，避免链表过长导致性能的问题。

## ConcurrentHashMap怎么实现线程安全

## 1.7

1. 1.7版本的ConcurrentHashMap采用分段锁机制，里面包含一个Segment数组，Segment继承与ReentrantLock，Segment则包含HashEntry的数组，HashEntry本身就是一个链表的结构，具有保存key、value的能力能指向下一个节点的指针。实际上就是相当于每个Segment都是一个HashMap，默认的Segment长度是16，也就是支持16个线程的并发写，Segment之间相互不会受到影响。
2. 初始的默认大小是2，当put第二个的时候，进行扩容。先扩容再添加元素。
3. put流程：①计算hash，定位到segment，segment如果是空就先初始化。②使用ReentrantLock加锁，如果获取锁失败则尝试自旋，自旋超过次数就阻塞获取，保证一定获取锁成功。③遍历HashEntry，就是和HashMap一样，数组中key和hash一样就直接替换，不存在就再插入链表，链表同样。
4. get流程：get也很简单，key通过hash定位到segment，再遍历链表定位到具体的元素上，需要注意的是value是volatile的，所以get是不需要加锁的。
5. get方法的安全性分析：（1）put 操作的线程安全性。初始化槽，这个我们之前就说过了，使用了 CAS 来初始化 Segment 中的数组。 添加节点到链表的操作是插入到表头的，所以，如果这个时候 get 操作在链表遍历的过程已经到了中间，是不会影响的。当然，另一个并发问题就是 get 操作在 put 之后，需要保证刚刚插入表头的节点被读取，这个依赖于 setEntryAt 方法中使用的 UNSAFE.putOrderedObject。 扩容。扩容是新创建了数组，然后进行迁移数据，最后面将 newTable 设置给属性 table。所以，如果 get 操作此时也在进行，那么也没关系，如果 get 先行，那么就是在旧的 table 上做查询操作；而 put 先行，那么 put 操作的可见性保证就是 table 使用了 volatile 关键字。 remove 操作的线程安全性。（2）remove 操作。get 操作需要遍历链表，但是 remove 操作会"破坏"链表。 如果 remove 破坏的节点 get 操作已经过去了，那么这里不存在任何问题。 如果 remove 先破坏了一个节点，分两种情况考虑。 1、如果此节点是头结点，那么需要将头结点的 next 设置为数组该位置的元素，table 虽然使用了 volatile 修饰，但是 volatile 并不能提供数组内部操作的可见性保证，所以源码中使用了 UNSAFE 来操作数组，请看方法 setEntryAt。2、如果要删除的节点不是头结点，它会将要删除节点的后继节点接到前驱节点中，这里的并发保证就是 next 属性是 volatile 的。
6. 扩容机制：从头结点开始向后遍历，找到当前链表的最后几个下标相同的连续的节点。从lastRun节点到尾结点的这部分就可以整体迁移到新数组的对应下标位置了，因为它们的下标都是相同的，可以这样统一处理。从头结点到 lastRun 之前的节点，无法统一处理，只能一个一个去复制了。且注意，这里不是直接迁移，而是复制节点到新的数组，旧的节点会在不久的将来，因为没有引用指向，被 JVM 垃圾回收处理掉。
7. size()：采用乐观的方式，认为在统计size的过程中没有发生put，remove等改变segment结构的操作。但是如果发生了就需要重试。如果重试两次都不成功，就只能强制把所有的Segment加锁后，再统计。

## 1.8

1. put流程：①首先计算hash，遍历node数组，如果node是空的话，就通过CAS+自旋的方式初始化。②如果当前数组位置是空则直接通过CAS自旋写入数据。③如果hash==MOVED，说明需要扩容，执行扩容。④如果都不满足，就使用synchronized写入数据，写入数据同样判断链表、红黑树，链表写入和HashMap的方式一样，key hash一样就覆盖，反之就尾插法，链表长度超过8就转换成红黑树。
2. get流程：通过key计算hash，如果key hash相同就返回，如果是红黑树按照红黑树获取，都不是就遍历链表获取。
3. 扩容机制：在元素迁移的时候，所有线程遵循从后向前推进的规则，在线程迁移过程中会确定一个范围，限定它此次迁移的数据范围。比如A线程第一个进来，只能迁移index=7 和 6 的数据，此时其他线程不能迁移这部分数据了，只能继续向前推进，寻找其他可以迁移的数据范围。且每次推进的步长都是固定值。线程B发现线程A正在迁移6 7的数据，只能向前寻找，迁移bound4和5的数据。维护了一个全局的transferIndex,来表示所有线程总共推进到元素下标的位置。比如当线程A第一次迁移成功后又向前推进，迁移2，3的数据，此时托没有其他线程在帮助迁移，则transferIndex为2。
4. 为什么到8才转成红黑树：因为通常情况下，链表长度很难达到8，但是特殊情况下链表长度为8，哈希表容量又很大，造成链表性能很差的时候，只能采用红黑树提高性能，这是一种应对策略。

# BlockingQueue

- ArrayBlockingQueue(数组阻塞队列，先进先出)，DelayQueue(延迟阻塞队列)，LinkedBlockingQueue(链阻塞队列，先进先出)，PriorityBlockingQueue(具有优先级的阻塞队列)，SynchronousQueue(同步队列)。
- 四组不同的行为方式：(add,remove,element) 抛出异常。(offer,poll,peek) 返回特定值。(put,take) 阻塞。(offer,poll) 超时。
- SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

# JUC线程池ThreadPoolExecutor

1. 核心概念：核心线程池大小，最大线程池大小，存活时间单位，存活时间大小，阻塞队列，拒绝策略。
2. 当提交一个新任务到线程池时，具体的执行流程如下：当我们提交任务，线程池会根据corePoolSize大小创建若干任务数量线程执行任务。当任务的数量超过corePoolSize数量，后续的任务将会进入阻塞队列阻塞排队。当阻塞队列也满了之后，那么将会继续创建(maximumPoolSize-corePoolSize)个数量的线程来执行任务，如果任务处理完成，maximumPoolSize-corePoolSize额外创建的线程等待keepAliveTime之后被自动销毁。如果达到maximumPoolSize，阻塞队列还是满的状态，那么将根据不同的拒绝策略对应处理。
3. 拒绝执行策略：AbortPolicy：直接丢弃任务，抛出异常，这是默认策略。CallerRunsPolicy：只用调用者所在的线程来处理任务。DiscardOldestPolicy：丢弃等待队列中最旧的任务，并执行当前任务。DiscardPolicy：直接丢弃任务，也不抛出异常。

## 核心方法

线程池的工作线程通过Woker类实现，在ReentrantLock锁的保证下，把Woker实例插入到HashSet后，并启动Woker中的线程。
从Woker类的构造方法实现可以发现: 线程工厂在创建线程thread时，将Woker实例本身this作为参数传入，当执行start方法启动线程thread时，本质是执行了Worker的runWorker方法。
firstTask执行完成之后，通过getTask方法从阻塞队列中获取等待的任务，如果队列中没有任务，getTask方法会被阻塞并挂起，不会占用cpu资源

1. execute：1）如果当前线程池中的线程数小于核心线程数，执行addWorker创建新线程执行command任务。2）否则，当线程池处于Running状态，把提交的任务成功放入阻塞队列中时，double check线程池的状态，如果线程池没有running，成功从阻塞队列中删除任务，执行reject方法处理任务。如果当前线程池处于running状态但是没有线程，创建一个空的线程。3）往线程池中创建新的线程失败，执行reject策略。
2. addWorker：addWorker主要负责创建新的线程并执行任务，线程池创建新线程执行任务时，需要获取全局锁。
3. runworker：线程启动之后，通过unlock方法释放锁，设置AQS的state为0，表示运行可中断； Worker执行firstTask或从workQueue中获取任务：(1)进行加锁操作，保证thread不被其他线程中断(除非线程池被中断) (2)检查线程池状态，倘若线程池处于中断状态，当前线程将中断。 (3)执行beforeExecute (4)执行任务的run方法 (5)执行afterExecute方法 (6)解锁操作。

## exectue方法中为什么double check线程池的状态

在多线程环境下，线程池的状态时刻在变化，而ctl.get()是非原子操作，很有可能刚获取了线程池状态后线程池状态就改变了。判断是否将command加入workque是线程池之前的状态。倘若没有double check，万一线程池处于非running状态(在多线程环境下很有可能发生)，那么command永远不会执行。

## 几种常见的线程池

- newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

线程池里的线程数量达到核心线程数后，即时线程池没有可执行任务，也不会释放线程。FixedThreadPool的工作队列为无界队列，线程池里的线程数量不会超过核心线程数，这导致最大线程数和存活时间是一个无用参数。饱和策略也失效。

- newSingleThreadExecutor:只会用一个线程来执行任务，保证任务的先进先出

```
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

初始化的线程池只有一个线程，该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行。由于使用无界队列，饱和策略失效。

- newCachedThreadPool：可灵活的回收线程，超过60秒就会自动回收。

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列。

### 关闭线程池

- shutdown()：中断所有没有正在执行任务的线程，线程池中的线程状态设置为SHUTDOWN状态。
- shutdownnow()：线程池里的线程状态设置为STOP状态，然后停止所有正在执行或暂停任务的线程。

## 线程工具类，CountDownLatch，CyclicBarrier，Semaphore

1. CountDownLatch 是一个线程等待其他线程， CyclicBarrier 是多个线程互相等待。
2. CountDownLatch 的计数是减 1 直到 0，CyclicBarrier 是加 1，直到指定值。
3. CountDownLatch 是一次性的， CyclicBarrier 可以循环利用。
4. CyclicBarrier 可以在最后一个线程达到屏障之前，选择先执行一个操作。
5. Semaphore ，需要拿到许可才能执行，并可以选择公平和非公平模式

## ThreadLocal

1. ThreadLocal可以理解为线程本地变量，他会在每个线程都创建一个副本，那么在线程之间访问内部副本变量就行了，做到了线程之间互相隔离，相比于synchronized的做法是用空间来换时间。**ThreadLocal有一个静态内部类ThreadLocalMap**，ThreadLocalMap又包含了一个Entry数组，Entry本身是一个弱引用，他的key是指向ThreadLocal的弱引用，Entry具备了保存key value键值对的能力。
2. 弱引用的目的是为了防止内存泄露，如果是强引用那么ThreadLocal对象除非线程结束否则始终无法被回收，弱引用则会在下一次GC的时候被回收。但是这样还是会存在内存泄露的问题，假如key和ThreadLocal对象被回收之后，entry中就存在key为null，但是value有值的entry对象，但是永远没办法被访问到，同样除非线程结束运行。但是只要ThreadLocal使用恰当，在使用完之后调用remove方法删除Entry对象，实际上是不会出现这个问题的。
3. 清理过期entry的方法，探测式清理和启发式清理。

- 应用场景：数据库管理，SimpleDateFormat，每个线程内保存类似于全局变量的信息，可以让不同方法直接使用，避免参数传递的麻烦，却不想被多线程共享。全局存储用户信息。

# JVM

# 类加载过程

加载-验证-准备-解析-初始化

## 类的加载

1. 通过一个类的全限定名来获取其定义的二进制流
2. **将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构**
3. 在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口

- 该阶段完成后，虚拟机外部的二进制字节流按照虚拟机所需的格式存储在存储在方法区之中，而且在java堆中也创建了一个 `java.lang.Class` 类的对象，这样就可以通过该对象访问方法区中的这些数据。
- JVM允许类加载器预先加载类，但是碰到.class文件缺失或存在错误，要等到程序首次主动使用此类才报错，一直不被使用一直不报错。
- 加载类的方式：本地系统加载，网络下载.class文件，zip、jar中加载，从专有数据库中提取.class文件，将java源文件编译成.class文件。

## 验证

- 确保class文件的字节流中包含的信息符号符合当前虚拟机的要求，并不会危害虚拟机自身的安全。

1. 文件格式验证：验证字节流是否符合Class文件格式的规范，开头是否是0xCAFEBABE，主次版本号是否在虚拟机的处理范围之内，常量池中的数据类型是否有不被支持的。
2. 元数据验证：对字节码描述的信息进行语义分析，以保证描述的信息符合java语言描述的要求。
3. 字节码验证：通过数据流和控制流分析，确定程序语义是合法的，符合逻辑的。
4. 符号引用验证：确保解析动作能正确执行。

## 准备

- 为类的静态变量分配内存和初始化为默认值，**这些内存都将在方法区中分配**

1. 这时候进行内存分配的仅为类变量，不包括实例变量，**实例变量会在对象实例化时随着对象一起分配在Java堆中**
2. 初始值通常情况下是数据类型默认的零值，不是被在Java代码中被显式地赋的值。显式赋的值在初始化阶段才会执行。

- 对于局部变量，必须显示的赋值，否则编译时不通过。
- 对于同时被static和final定义的变量，声明的时候必须显式的赋值。只被final修饰的变量可以在声明时显式的赋值，也可以在类初始化的时候显式的赋值。
- 对于引用数据类型reference，如数组引用、对象引用等，如果没有显式的赋值而直接使用，系统默认为零值
- 数组初始化时没有对数组中的各元素赋值，默认零值

## 解析

- 解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行。
- 符号引用就是一组符号来描述目标，可以是任何字面量
- 直接引用直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄

## 初始化

- 真正执行类中定义的Java程序代码(字节码)，初始化阶段是在执行初始化方法<clinit>()方法的过程。
- 对于<clinit>()方法的调用，虚拟机会确保自己在多线程环境中的安全性，是带锁线程安全，所以在多线程环境下进行类初始化可能会引起死锁。
- 主要对类变量进行初始化
- 初始化步骤:

1. 假设这个类还没有被加载和连接，则程序先加载并连接该类
2. 假设该类的父类还没有被初始化，先初始化其直接父类
3. 假设类中有初始化语句，系统依次执行这个初始化语句

- 类初始化时机：只有对类的主动使用的时候才会导致类的初始化

1. 当遇到new、getstatic、putstatic、invokestatic时。
2. 初始化某个类的子类，该类父类会被初始化。
3. 反射机制
4. 虚拟机启动时被表明为启动类的类

## 卸载

- 系统自带的类加载器(BootstrapClassLoader、ExtClassLoader、AppClassLoader)的实例不会被回收的，程序员自定义的类加载器的实例是可以被回收的，所以使用我们自定义加载器加载的类是可以被卸载掉的。
- 卸载需要满足的要求：

1. 该类中所有实例对象都被GC，堆中不存在该类的实例对象
2. 该类没有在任何地方被引用
3. 该类的类加载器的实例已被GC

## JVM类加载机制

### 类加载机制

- 全盘负责：当一个类加载器负责加载某个类时，该类所依赖和引用的其他的class也要由该类加载器进行加载，除非显示使用另一个类加载器来载入。
- 父类委托：先让父类加载器尝试加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类。
- 缓存机制：保证所有被加载过的类都会被缓存，当程序需要使用某个类时，首先去缓存区中寻找该类，如果没有找到才去加载二进制数据，转换成class对象，存入缓存区。
- 双亲委派机制：如果一个类加载器收到类加载的请求，首先不会自己去尝试加载这个类，而是把请求委托加载给父加载器去完成。所有类加载请求都会到顶层的启动类加载器中，只有当父加载器无法加载该类时，子加载器才会去自己加载该类。

### 双亲委派机制过程

1. 当AppClassLoader加载一个class时，它不会自己去尝试加载某个类，而是将类加载请求委派给父类加载器ExtClassLoader去完成。
2. 当ExtClassLoader加载一个class时，它不会自己去尝试加载某个类，而是将类加载请求委派给父类加载器BootStrapClassLoader去完成。
3. 如果BootStrapClassLoader加载失败，会使用ExtClassLoader去尝试加载。
4. 如果ExtClassLoader也加载失败，会使用AppClassLoader去尝试加载。如果加载失败，会报出ClassNotFoundException异常。

### 双亲委派机制的好处

- 保证了app的稳定运行，避免了类被重复加载，也保证了java的核心api不被篡改。

### 什么时候会打破双亲委派模型?

1. jdk中的基础类作为用户典型的api被调用，但是也存在被api调用用户的代码的情况。
2. SPI机制简介 SPI的全名为Service Provider Interface，SPI机制的思想：我们系统里抽象的各个模块，往往有很多不同的实现方案，比如日志模块、xml解析模块、jdbc模块等方案。面向的对象的设计里，我们一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。Java SPI就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。

3. 使用线程上下文类加载器，在jdbc的使用中，获取jdbc.drivers的属性，得到类的路径，然后通过系统类加载器加载，这时候系统类加载器加载了应用类加载器应该加载的类。

# JVM内存结构

## 程序计数器

- 较小的一块内存空间，**可以看作是当前线程所执行的字节码的行号指示器**。用来存储指向下一条指令的地址，即将执行的指令代码，由执行引擎读取下一条指令。
- 程序计数器的作用

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制：顺序执行、选择、循环、异常处理。
2. 多线程环境中，程序计数器记录当前线程执行的位置，下次执行的时候能够继续执行。

- PC寄存器为什么线程私有？ 多线程在一个特定时间内只会执行其中某一个线程方法，CPU会不停的做任务切换，这样必然会导致经常中断和恢复。为了能够准确记录当前线程的执行位置，为每个线程分配一个PC寄存器，每个线程独立计算，不会相互影响。(上下文切换)

## 虚拟机栈

### 栈帧

- 方法的调用都对应栈帧的出栈和入栈
- 内部结构：局部变量表，操作数栈，动态链接（指向运行时常量池的方法引用），方法返回地址（方法正常退出或异常退出的地址）

#### 局部变量表

- 主要用于存放方法参数和定义在方法体内部的局部变量，包括编译期可知的各种java虚拟机的基本数据类型，对象引用可能是指向对象起始地址的引用指针，也可能是指向一个代表一个对象的句柄。
- 如果该帧是由构造方法或者实例方法创建的，那么该对象引用this就会存在index为0的slot处。静态方法为什么不可以引用this，就是因为this不存在当前方法的局部变量表里。

#### 操作数栈

- 主要用于保存计算过程中的中间结果，同时作为计算过程中变量的临时存储空间。
- 栈顶缓存：基于栈式架构的虚拟机所使用的零地址指令更加紧凑，但完成一项操作的时候必然需要使用更多的入栈和出栈指令，这同时也就意味着将需要更多的指令分派（instruction dispatch）次数和内存读/写次数。由于操作数是存储在内存中的，因此频繁的执行内存读/写操作必然会影响执行速度。为了解决这个问题，HotSpot JVM 设计者们提出了栈顶缓存技术，将栈顶元素全部缓存在物理 CPU 的寄存器中，以此降低对内存的读/写次数，提升执行引擎的执行效率。

#### 动态链接

- 动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用
- 静态链接：当一个字节码文件被装载进JVM内部时，如果被调用的目标方法在编译期可知，且运行期保持不变，这种情况下将调用方法的符号引用转换为直接引用的过程称为静态链接。
- 动态链接：如果被调用的方法在编译期无法被确定下来，也就是说，只有在程序运行期将调用方法的符号引用转换为直接引用，由于这种引用转换过程具有动态性，所以称为动态链接。

## 本地方法栈

## 堆

### 内存划分

1. 分代的唯一理由就是优化GC性能。
2. 新生代，老年代，元空间。
3. TLAB：处于伊甸区，多线程同时分配内存时，使用TLAB可以避免一系列的非线程安全问题，同时还能提升内存分配的吞吐量。一旦对象在TLAB空间分配内存失败时，JVM就会尝试着通过使用加锁机制确保数据操作的原子性，从而直接在Eden空间中分配内存。在内存读取方面堆是线程共享的，在内存分配方面，存在不共享的情况。

## 方法区

1. 用于存储类信息，常量池，静态变量，即时编译器编译后的代码缓存等。

2. 运行时常量池：一个有效的字节码文件中除了包含类的版本信息，字段，方法以及接口等描述信息外，还包含一项信息就是常量池表，包含这种字面量和对类型、域和方法的符号引用。
3. 常量池表是Class文件的一部分，用于存储编译器生成的各种字面量和符号引用，这部分内容在类加载后存放到方法区的运行时常量池中。具有动态性，不要求常量一定在编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。
4. Java1.8之后的变化：(1)移除了永久代(PermGen)，替换为元空间(Metaspace)；(2)永久代中的class metadata转移到了native memory(本地内存，而不是虚拟机)；(3)永久代中的字符串常量池和静态变量转移到了Java堆中； (4)永久代参数(PermSize MaxPermSize) -> 元空间参数(MetaspaceSize MaxMetaspaceSize)

## 对象的内存布局

对象在堆内存的存储布局可分为对象头、实例数据和对齐填充。

对象头占 12B，包括对象标记和类型指针。对象标记存储对象自身的运行时数据，如哈希码、GC 分代年龄、锁标志、偏向线程 ID 等，这部分占 8B，称为 Mark Word。Mark Word 被设计为动态数据结构，以便在极小的空间存储更多数据，根据对象状态复用存储空间。

类型指针是对象指向它的类型元数据的指针，占 4B。JVM 通过该指针来确定对象是哪个类的实例。

实例数据是对象真正存储的有效信息，即本类对象的实例成员变量和所有可见的父类成员变量。存储顺序会受到虚拟机分配策略参数和字段在源码中定义顺序的影响。相同宽度的字段总是被分配到一起存放，在满足该前提条件的情况下父类中定义的变量会出现在子类之前。

对齐填充不是必然存在的，仅起占位符作用。虚拟机的自动内存管理系统要求任何对象的大小必须是 8B 的倍数，对象头已被设为 8B 的 1 或 2 倍，如果对象实例数据部分没有对齐，需要对齐填充补全。

### 对象的访问定位有哪两种方式?

- 使用句柄:Java 堆中将会划分出一块内存来作为句柄池，栈中的reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息
- 直接指针
- 使用句柄来访问的最大好处是 栈中的reference 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而reference 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

### Java对象的创建过程

- 类加载检查：能够在常量池中定位到该类的符号引用，并检查这个类是否已经被加载过，解析和初始化过，如果没有必须执行相应的加载过程。
- 分配内存
  - 内存分配的两种方式：指针碰撞和空闲列表
  - 内存分配并发问题：CAS+TLAB
- 初始化零值
- 设置对象头
- 执行init方法

## JMM模型

1. as-if-serial:不管怎么重排序，单线程下的执行结果不能被改变
2. happen-before规则
   ①单一线程原则，在一个程序内，程序前面的操作先于发生程序后面的操作。
   ②管程锁定规则，一个unlock操作先行发生于后面对同一个锁的lock操作。
   ③volatile变量规则，对于一个volatile变量的写操作先行发生于后面对这个变量的读操作。
   ④线程启动规则，Thread对象的start发生先行发生于此线程的每一个动作。
   ⑤线程加入规则，Thread对象结束先行发生于join方法返回。
   ⑥线程中断规则，对线程interrupt方法的调用先行发生于被中断线程的代码检测到中断事件的发生。可以通过 interrupted() 方法检测到是否有中断发生。
   ⑦对象终结规则，一个对象的初始化先行发生于它的finalize方法的开始。
   ⑧传递性，如果操作A先行发生于操作B，操作B先行发生于操作C，那么操作A先行发生于操作C。

- 所有的共享变量都存储于主内存，这里所说的变量指的是实例变量和类变量，不包含局部变量，因为局部变量是线程私有的，因此不存在竞争问题。
- 每一个线程还存在自己的工作内存，线程的工作内存，保留了被线程使用的变量的工作副本。
- 线程对变量的所有的操作(读，取)都必须在工作内存中完成，而不能直接读写主内存中的变量。
- 不同线程之间也不能直接访问对方工作内存中的变量，线程间变量的值的传递需要通过主内存中转来完成。

# Java垃圾回收

## 判断一个对象是否可被回收

1. 引用计数器法
2. 可达性分析算法：GC Roots一般包含：虚拟机栈中引用的对象，本地方法栈中引用的对象，方法区类静态属性引用的对象，方法区中常量引用的对象。
3. 方法区的回收：该类的所有实例已经被回收，也就是堆中不存在该类的任何实例。该类的类加载器已经被回收。该类的类对象没有在任何地方被引用，也就是说无法通过反射访问该类方法。

## 引用类型

- 强引用，软引用，弱引用，虚引用。
- 强引用，任何时候都不会被jvm回收，即使oom。软引用：当内存不足时，会被回收。弱引用：每次gc的时候都会被回收，不管内存是否充足。虚引用：需要和referencequeue联合使用，当jvm扫描到虚引用的对象时，会先将此对象放到关联的队列中，因此我们可以通过判断队列中是否存在该对象，来了解被引用的对象是否将要被回收，进行一些回收前的处理。
- 使用场景：软引用：创建缓存，弱引用：WeakHashMap类中的key，可以使用WeakHashMap<String,Map<k,v>>保存事务的信息，在事务周期中，String对象的强引用一直存在，我们就一直可以获取信息，当事务结束时，弱引用可以自动帮我们清除map信息。虚引用：用于对象销毁前的一些操作，比如资源释放等。

## 垃圾回收算法

1. 标记-清除
2. 标记-整理
3. 复制
4. 分代收集

## 垃圾回收器

1. Serial垃圾回收器
   Serial垃圾回收器，只会使用一个线程进行垃圾回收，新生代：复制算法，暂停所有用户线程，老年代：标记整理，暂停所有用户线程。
2. Serial Old垃圾回收器
3. CMS垃圾回收器
4. G1垃圾回收器

## CMS详解

- 初始标记：只是简单地标记一下gc root能关联的对象，时间很短，需要停顿，并发标记：进行GC Root Tracing的过程，耗时最长，不需要停顿，重新标记：修正并发标记期间因用户程序继续运行而产生变动的那一部分对象的标记记录，需要停顿。并发清除：不需要停顿。
- 缺点：吞吐量低，低停顿时间是以牺牲吞吐量为代价的，导致CPU利用率不高。无法处理浮动垃圾，可能会导致Concurrent Mode Failure。浮动垃圾是指并发清除阶段用户程序继续运行而产生的，只能等到下次GC的时候才能会被回收。由于浮动垃圾的存在，老年代需要预存一段内存，CMS不能像其他垃圾回收器一样等到老年代快满时，再进行垃圾回收。如果预留的空间不足时，可能会导致Concurrent Mode Failure，将临时启用Serial Old来代替CMS。标记-清除算法导致碎片，往往出现老年代剩余，导致无法找到一块连续的足够大的内存空间存放对象，不得不提前触发一次full gc。

## G1详解

- G1垃圾回收器：其他垃圾回收器都是面向新生代或者老年代，G1可以直接对新生代和老年代一起回收。G1把堆分成多个大小相等的独立区域，新生代和老年代不再物理隔离。对每一个小块进行单独的垃圾回收，这种划分方法带来很大的灵活性，使得可预测的停顿时间模型成为可能。通过记录每个Region垃圾回收时间以及回收可获得的空间，并维护一个优先列表，每次根据允许的收集时间，回收价值最大的Region。每个Region都有一个Remembered set，用来记录Region对象的引用所在的Region。通过使用Remembered set，避免在做可达性分析的时候进行全堆扫描。
- 初始标记，并发标记，最终标记，为了修正并发标记期间用户程序继续运行而产生变动的那部分记录，虚拟机将这段时间对象变化记录在线程的Remembered set logs中，最终标记阶段需要把Remebered set logs的数据合并到remembered set中，需要停顿用户线程，但是可以并行执行。筛选回收：首先对region中回收价值和成本进行排序，根据用户期望的停顿时间来制定回收计划，此阶段也可以做到并发执行，但是因为只回收一部分region，时间是用户可控制的，而且停顿用户线程可以大幅度提高收集效率。
- 特点：空间整合，整体来看是基于标记-整理，局部来看(两个Region)，是基于复制算法，这意味着运行期间不会产生空间碎片。可停顿的预测：能让使用者指定在一个M毫秒的时间内，消耗在GC上的时间不超过N毫秒。

## 三色标记法

白色标记为未被标记的对象，灰色为自身被标记，成员变量未被标记，黑色为自身被标记，成员变量也被标记。

## 内存分配与回收策略

- 对象优先在伊甸区，大对象直接在老年代，空间分配担保，长期存活的对象进入老年代，动态对象年龄判定(当幸存区中的同年龄对象总和超过所有对象的一半时，大于等于年龄的对象直接进入老年代)。
- 空间分配担保：在发生 Minor GC 之前，虚拟机先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果条件成立的话，那么 Minor GC 可以确认是安全的。 如果不成立的话虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败，如果允许那么就会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次 Minor GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那么就要进行一次 Full GC。

### 什么时候触发full gc

System.gc，空间担保失败，Concurrent Mode Failure，老年代空间不足，1.7及之前的永久代空间不足。

## 调优参数

jps,jstack,jmap

## 内存泄漏和内存溢出

内存泄漏：在申请内存后无法释放已申请的内存空间就造成了内存泄漏。
内存溢出：程序在申请内存时，没有足够的内存供申请者使用。

# 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

@Autowired 可以对成员变量、方法以及构造函数进行注释

## 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：把当前类对象存入 Spring 容器中，相当于在 xml 中配置一个 bean 标签。value 属性指定 bean 的 id，默认使用当前类的首字母小写的类名。
2. @Controller
3. @Service
4. @Repository
5. @Bean 如果想将第三方的类变成组件又没有源代码，也就没办法使用 @Component 进行自动配置，这种时候就要使用 @Bean 注解。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

1. @Configuration 用于指定当前类是一个 spring 配置类，当创建容器时会从该类上加载注解，value 属性用于指定配置类的字节码。
2. @ComponentScan 用于指定 Spring 在初始化容器时要扫描的包。basePackages 属性用于指定要扫描的包。
3. @PropertySource 用于加载 .properties 文件中的配置。value 属性用于指定文件位置，如果是在类路径下需要加上 classpath。
4. @Import 用于导入其他配置类，在引入其他配置类时可以不用再写 @Configuration 注解。有 @Import 的是父配置类，引入的是子配置类。value 属性用于指定其他配置类的字节码。

# 自动注入是怎么注入的

1. 先讲到populateBean方法进行属性注入。
2. 属性注入方法中会从IOC容器中得到所有的后置处理器，其中就包括 AutowiredAnnotationBeanPostProcessor，然后调用 postProcessPropertyValues 进行 autowire 自动注入。
3. AutowiredAnnotationBeanPostProcessor利用反射得到autowire元注解信息，得到需要注入的bean,封装成InjectionMetadata
   调用 InjectionMetadata.inject 进行自动注入，之后解析注入的Bean，得到Bean的实例 ，然后通过反射机制给autowire的字段注入Bean的实例。

# 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?

在完成Bean的实例化后，属性注入之前Spring将Bean包装成一个工厂添加进了三级缓存中

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。然后将A放入二级缓存中，删除三级缓存中的A。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，放入一级缓存，删除二级缓存中的B。接着会将B再注入到A中，此时A再完成它的整个生命周期。将A放入一级缓存中，删除二级缓存中的A。至此，循环依赖结束。

### 能不能解决传参的循环依赖？你自己实现一个解决传参的，怎么实现?

- 依赖注入的方式不能全是构造器注入的方式

### 为什么要使用三级缓存呢？二级缓存能解决循环依赖吗?

这个工厂的目的在于延迟对实例化阶段生成的对象的代理，只有真正发生循环依赖的时候，才去提前生成代理对象，否则只会创建一个工厂并将其放入到三级缓存中，但是不会去通过这个工厂去真正创建对象中

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。

## 三级缓存大小

# 什么是AOP

AOP（Aspect Oriented Programming）是基于切面编程的，可无侵入的在原本功能的切面层添加自定义代码，一般用于日志收集、权限认证等场景。

通过动态代理实现invocationHandler接口，并且把目标对象注入到代理对象中，获取应用到此方法的执行器链，如果有执行器链则创建MethodInvocation，并调用proceed方法，否则直接反射调用目标方法。

# 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生命周期事件，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

## 只有public能让事务有效

只要是以代理方式实现的声明式事务，无论是JDK动态代理，还是CGLIB直接写字节码生成代理，都只有public方法上的事务注解才起作用。而且必须在代理类外部调用才行，如果直接在目标类里面调用，事务照样不起作用。

# Spring事务基于注解能用于分布式么?

# spring @Transactional使用过程中踩过什么坑

- 在 @Transactional 注解中如果不配置rollbackFor属性,那么事物只会在遇到RuntimeException的时候才会回滚，加上 rollbackFor=Exception.class,可以让事物在遇到非运行时异常时也回滚。
- 非public方法。
- 事务方法内部捕捉了异常，没有抛出新的异常，导致事务操作不会进行回滚

# 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)

```java
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
    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TransactionDefinitic
        // 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，
        // 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

        // -------------  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

## MyBatis

## #{} 和 ${} 的区别

${} 相当于使用字符串拼接，存在SQL注入风险。${}的变量替换在DBMS外，变量替换后，{}对应的变量不会加上单引号。
#{} 相当于使用占位符，可以防止SQL注入，动态参数。#{}的变量替换在DBMS中，变量替换后，#{}对应的变量自动加上单引号。

## 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。一个会话，查询一条数据，这个数据会被放在当前会话的一级缓存中。如果会话被关闭了，一级缓存中的数据会被保存到二级缓存。新的会话查询信息就会参照二级缓存。
缓存首先一进来去查二级缓存，二级缓存没有去找一级缓存，一级缓存没有去找数据库。

# 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

跨站点请求伪造

1. 用户C打开浏览器，访问受信任网站A，输入用户名和密码请求登录网站A；
2. 在用户信息通过验证后，网站A产生Cookie信息并返回给浏览器，此时用户登录网站A成功，可以正常发送请求到网站A；
3. 用户未退出网站A之前，在同一浏览器中，打开一个TAB页访问网站B；
4. 网站B接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点A；
5. 浏览器在接收到这些攻击性代码后，根据网站B的请求，在用户不知情的情况下携带Cookie信息，向网站A发出请求。网站A并不知道该请求其实是由B发起的，所以会根据用户C的Cookie信息以C的权限处理该请求，导致来自网站B的恶意代码被执行。

CSRF防护的一个重点是要对“用户凭证”进行校验处理，通过这种机制可以对用户的请求是合法进行判断，判断是不是跨站攻击的行为。因为“用户凭证”是Cookie中存储的，所以防护机制的处理对象也是Cookie的数据，我们要在防护的数据中加入签名校验，并对数据进行生命周期时间管理，就是数据过期管理。

# XSS攻击

XSS攻击是Web攻击中最常见的攻击方法之一，它是通过对网页注入可执行代码且成功地被浏览器执行，达到攻击的目的，形成了一次有效XSS攻击，一旦攻击成功，它可以获取用户的联系人列表，然后向联系人发送虚假诈骗信息，可以删除用户的日志等等，有时候还和其他攻击方式同时实施比如SQL注入攻击服务器和数据库、Click劫持、相对链接劫持等实施钓鱼，它带来的危害是巨大的，是web安全的头号大敌。

# 什么是WebSockets

- 服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，是真正的双向平等对话。
  优势：

1. 建立在 TCP 协议之上，服务器端的实现比较容易。
2. 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。
3. 数据格式比较轻量，性能开销小，通信高效。
4. 可以发送文本，也可以发送二进制数据。
5. 没有同源限制，客户端可以与任意服务器通信。
6. 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

## SpringBoot达成的jar和普通的jar有什么区别?

springboot的jar包不能作为普通的jar包被其他项目依赖，主要是普通jar的结构不同，普通的jar包解压后直接就是包名，包里是我们的代码，springboot的jar包解压后在\BOOT-INF\classes目录下才是我们的代码。

# Dubbo



## 一个网站的架构模型是怎样的?

1. 单一应用架构：当网站流量很小的时候，将所有功能都部署在一起，以减少部署节点和成本，
2. 垂直应用架构：当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，提升效率的方法之一是将应用拆成互不相干的几个应用，以提升效率。
3. 分布式服务架构：当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用以及整合的分布式服务器框架是关键。
4. 流动计算架构：当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心是关键。

## RPC

RPC(Remote Procedure Call)：远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的思想。计算机 A 上的进程，调用另外一台计算机 B 上的进程，其中 A 上的调用进程被挂起，而 B 上的被调用进程开始执行，当值返回给 A 时，A 进程继续执行。调用方可以通过使用参数将信息传送给被调用方，而后可以通过传回的结果得到信息。而这一过程，对于开发人员来说是透明的。

## 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 `proxyFactory.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

服务的引入又分为了三种，第一种是本地引入、第二种是直接连接引入远程服务、第三种是通过注册中心引入远程服务。
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(可能一个服务有多个提供者)，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。

# 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。

## 关系型数据库和非关系型数据库的区别?

1. 关系型数据库一般有固定的表结构，不是很容易进行扩展，非关系型数据库存储比较灵活，比如基于图的，键值对的，文档的，方便扩展。
2. 传统的关系型数据库横向扩展难，不好对数据进行分片等。非关系型数据库原生的支持水平扩展。
3. 关系型数据库强调的是强一致性，非关系型数据库强调的是最终的一致性。

## 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

## redis哪些数据结构可以用作队列?

- 一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

- 如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。
- 如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。
- 如果对方追问pub/sub有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。
- 如果对方追问redis如何实现延时队列？使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

## 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Leader的选举机制

1. 拿到半数以上的赞成票
2. 拿到的票数同时大于等于哨兵配置文件中的 quorum 值。

### 主库下线的判定：

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。

### 主库的选出

1. 过滤掉不健康的(下线或掉线)，没有回复过哨兵ping的从节点。
2. 选出slave-priority从节点优先级最高的(redis.conf)的。
3. 选择复制偏移量最大，指复制最完整的从节点

### 故障的转移

1. 将slave1脱离原从节点，升级为主节点。
2. 将从节点指向新的主节点
3. 通知客户端主节点已更换
4. 将原主节点变为从节点，指向新的主节点。

# Redis集群

## 三种集群模式

主从复制、哨兵机制、Redis Cluster

## Redis Cluster

### 概念

redis哨兵模式已经可以实现高可用、读写分离，但是在这种模式下每台服务器都存储相同的数据，很浪费内存，所以在3.0版本中加入了cluster模式，实现redis的分布式存储，也就是每台redis服务器上存储不同的数据。
Redis-Cluster采用无中心结构，它的特点如下：

- 所有redis节点相连(PING-PONG机制)，内部使用二进制协议优化传输速度和带宽。
- 节点的fail是通过集群中超过半数的节点检测失效时才生效。
- 客户端与redis节点相连不需要中间代理层，只需要与集群中任一可用的redis节点相连即可。

**实现方式**

在redis节点上都有这么两个东西，插槽(slot)，它的取值范围是0-16383，还有一个是集群，可以理解为是集群管理的插件。当我们存取的key到达的时候，redis会根据crc16算法得出一个结果，然后把结果对16384取余数，这样每个key都会对应一个0-16383之间的哈希槽，通过这个值去找到对应这个插槽所对应的节点，然后直接跳转到这个对应节点上进行存取操作。

为了保证高可用性，redis集群引入了主从模式，一个主节点对应一个或多个从节点，当主节点宕机的时候，就会启用从节点。当其他主节点ping一个主节点A的时候，如果半数以上的主节点与A通信超时，那么认为节点A宕机了。如果主节点A和从节点A1都宕机了，那么该集群就无法再提供服务了。

### Hash槽

**为什么redis中的hash slot是16384**

在redis节点发送心跳包时需要把所有的槽放到这个心跳包里，以便让节点知道当前集群信息，16384=16k，在发送心跳包时使用char进行bitmap压缩后是2k（2 * 8 (8 bit) * 1024(1k) = 16K），也就是说使用2k的空间创建了16k的槽数。

虽然使用CRC16算法最多可以分配65535（2^16-1）个槽位，65535=65k，压缩后就是8k（8 * 8 (8 bit) * 1024(1k) =65K），也就是说需要需要8k的心跳包，作者认为这样做不太值得；并且一般情况下一个redis集群不会有超过1000个master节点，所以16k的槽位是个比较合适的选择。

### 请求重定向

在集群模式下，节点对请求的处理过程：检查当前key是否存在当前node：通过crc16(key) % 16384计算出槽，查询出负责该槽的节点，得到节点指针，该指针与自身节点比较。如果该槽不是当前节点负责，返回MOVED重定向。如果该槽是当前节点负责，且key在该槽中，返回该key对应的结果。如果该key不在slot中，检查该slot是否正在迁出，如果正在迁出返回ASK错误重定向客户端到目的服务器。若该slot未迁出，检查slot是否导入中，如果slot正在导入中且有ASKING标记，则直接操作，否则返回MOVED重定向。

### 故障恢复

当slave发现master变为fail时，便尝试进行failover，以期望称为新的master。可能挂掉的master有多个slave，需要经过类raft协议的过程在集群中达成一致。具体过程如下：slave发现自己的master变为fail，将自己记录的集群currentEpoch加1，并广播failover request信息。其他节点收到该信息，只有master响应，判断请求者的合法性，并发送FAILOVER_AUTH_ACK，对每个epoch只发送一次ack。尝试failover的slave收集FAILOVER_AUTH_ACK。超过半数称为新的master，并广播PONG通知其他集群节点。

### 扩容和缩容

**扩容**

首先将新节点加入集群，默认是主节点，首先需要确认哪些槽需要被迁移到目标节点，然后获取槽中的key，将槽中的key全部迁移到目标节点，然后向集群所有节点广播槽全部迁移到了目标节点，

**缩容**

首先判断下线的节点是否是主节点，以及主节点上是否有槽，若主节点上有槽，需要将槽迁移到集群中的其他主节点，槽迁移完成之后需要向所有节点广播该节点准备下线。最后需要将该下线主节点的从节点指向其他主节点。

### 为什么redis集群不建议使用发布订阅模式

在集群模式下，所有的publish命令都会向所有节点（包括从节点）进行广播，造成每条publish数据都会在集群内所有节点传播一次，加重了带宽负担，对于在有大量节点的集群中频繁使用pub，会严重消耗带宽，不建议使用

# 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

# Netty

# Netty是什么?

- Netty成功找到了一种在不妥协可维护性和性能的情况下实现易于开发，性能，稳定性和灵活性的方法。

1. Netty是一个基于NIO的CS(客户端-服务端)框架，使用它可以快速简单的开发网络应用框架。
2. 它极大地简化并优化了TCP和UDP套接字服务器等网络编程，并且性能以及安全性等很多方面甚至都要更好。
3. 支持多种协议，如FTP、SMTP、HTTP以及各种二进制和基于文本的传统协议。

# 为什么要用Netty?

- 相比较于直接使用JDK自带的NIO相关的API来说，更加易用，具有下面这些优点：

1. 统一的API，支持多种传输协议，阻塞和非阻塞的
2. 简单且强大的线程模式
3. 自带编码器解决TCP粘包/拆包问题
4. 自带各种协议栈
5. 真正的无连接数据包套接字支持
6. 比直接使用Java核心API有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
7. 安全性不错，有完整的SSL/TLS和StartTLS支持。
8. 社区活跃
9. 成熟稳定，经历了大项目的使用和考验，而且很多开源项目都使用到了Netty。

# Netty应用场景

- 主要用作网络通信
- 作为RPC框架的网络通信工具
- 实现一个自己的HTTP服务器
- 实现一个即时通讯系统
- 实现一个消息推送系统

# Netty核心组件有哪些？分别有什么作用?

## Channel

- Channel接口是Netty对网络操作抽象类。
- 比较常用的Channel接口实现类是NioServerSocketChannel(服务端)和NioSocketChannel(客户端)

## EventLoop

- 负责监听网络事件中并调用事件处理器进行相关的I/O操作。
- Channel为Netty网络操作抽象类，EventLoop负责处理注册到上面的Channel处理I/O操作，两者配合参与I/O操作。

## ChannelFuture

- Netty是异步非阻塞的，所有操作都是异步的。因此我们不能立刻知道操作是否执行成功，我们可以通过ChannelFuture接口的addListener方法注册一个ChannelFutureListener，当操作成功或失败时，监听会自动触发并返回结果。

## ChannelHandler、ChannelPipeline

- ChannelHandler是消息的具体处理器，负责读写操作、客户端连接等事情。
- ChannelPipline是ChannelHandler的链，提供一个容器并定义了用于沿着链传播入站和出站事件流的API。
- 一个数据或事件可能会被多个Handler处理，当一个ChannelHander处理完后就将数据交给写一个Handler。

# EventLoopGroup了解么？和EventLoop啥关系?

- EventLoopGroup包含多个EventLoop，BossEventGroup用于接收连接，WorkerEventGroup用于具体的处理(消息的读写以及其他逻辑处理)。
- bossGroup处理客户端连接，当客户端处理完成后，会将这个连接提交给workGroup来处理，然后workGroup负责处理其IO相关操作。

# Bootstrap和ServerBootstrap了解么?

- Bootstrap是客户端的启动引导类/辅助类
- ServerBootstrap是服务端的启动引导类/辅助类
- Bootstrap通常使用connect()方法连接到远程的主机和端口，作为一个Netty TCP协议中通信的客户端。另外Bootstrap也可以绑定一个本地端口作为UDP协议通信中的一端。
- ServerBootstrap通常使用绑定本地端口，等待客户端的连接。
- Bootstrap只需要配置一个EventGroup，而ServerBootstrap需要配置两个EventstrapGroup，一个用于接收连接，一个用于具体的处理。

# NioEventLoopGroup默认的构造函数会起多少线程?

- CPU核心数 * 2

# Netty线程模型了解么?

- Reactor模式：基于事件驱动，采用多路复用将事件分发给相应的Handler处理，非常适合处理海量IO的场景。
- 大部分网络框架基于Reactor模式设计开发。
- Netty主要利用EventLoopGroup线程池来实现具体的线程模型的，实现服务端的时候一般会初始化两个线程池，bossGroup和workerGroup，bossGroup接收连接，workerGroup负责具体的处理，交由相应的Handler处理。

1. 单线程模型
2. 多线程模型：一个Acceptor线程负责客户端的连接，一个NIO线程组负责具体的处理。
3. 主从多线程模型：从一个主线程组中选一个线程负责客户端的连接，其他的线程负责后续的接入认证工作。连接建立完成后，从NIO线程组负责具体的IO处理。

# Netty服务端和客户端的启动过程了解么?

## 服务端：

1. 创建两个NioEventLoopGroup实例，一个bossGroup、一个workerGroup
2. 创建一个ServerBootstrap服务端启动引导类，这个类引导我们完成服务端的启动工作。
3. .group()方法为这个引导类配置两个线程组，确定了线程模型。
4. 通过.channel()方法为这个引导类指定了IO模型为NIO
5. 通过.childHandler()给引导类创建一个ChannelInitializer，然后指定了服务端消息的业务处理逻辑HelloServerHandler对象
6. 调用ServerBootstrap类的bind()方法绑定端口

## 客户端：

1. 创建一个NioEventLoopGroup实例
2. 创建一个Bootstrap启动引导类，这个类引导我们完成客户端的启动工作。
3. .group()方法为这个引导类配置一个线程组
4. 通过.channel()方法为这个引导类指定了IO模型为NIO
5. 通过.childHandler()给引导类创建一个ChannelInitializer，然后指定了客户端消息的业务处理逻辑HelloClientHander对象
6. 调用Bootstrap类的connect()方法进行连接，这个方法指定两个参数：inetHost：ip地址，inetPort：端口号

# 什么是TCP粘包/拆包？有什么解决方法呢?

- 基于TCP发送数据的时候，多个字符串“粘”在了一起或者一个字符串被拆开的问题。
- 使用Netty自带的解码器：LineBasedFrameDecoder：发送端发送数据包的时候，每个数据包之间以换行符作为分割。DelimiterFrameDecoder：可以自定义分割符解码器。FixedLengthFrameDecoder：固定长度解码器。
- 自定义序列化编解码器

# Netty长链接、心跳机制了解么?

- TCP在进行读写的时候，建立连接时会进行三次握手，断开时会进行四次挥手的操作，这个过程是比较消耗网络资源并且有时间延迟的。
- 短连接：在进行完读写之后就会断开连接，如果下次再要相互发送消息，就重新建立连接。每一次的读写都要建立连接必然会带来大量网络资源的消耗，并且连接的建立也需要消耗时间。
- 长连接：建立连接之后，完成一次读写不会主动关闭它们之间的连接，省去了较多的TCP建立和关闭的操作，降低对网络资源的依赖，节约时间。
- 在保持长连接期间，网络异常出现的时候，client和server之间如果没有交互的话是不会知道对方已经掉线的，这时候需要引入心跳机制。
- 在client与server之间在一段时间内没有数据交互的时候，会发送特殊的数据包给对方，当接收方收到这个数据报文后，也会回一个特殊的数据包给对方，这样就是一个PING-PONG交互。
- TCP实际上有长连接选项，本身也有心跳包机制，但是TCP协议层面的长连接灵活性不够，一般都是需要在应用层协议上自定义心跳机制，也就是在Netty层面通过编码实现。

## Netty的零拷贝了解么?

- 零拷贝(零复制)：计算机执行操作时，CPU不需要先将数据从某处内存复制到另一个特定的区域。这种技术通常用于通过网络传输文件时节省CPU周期和内存带宽。
- 在OS层面的零拷贝一般指避免在用户态和内核态之间来回拷贝数据。而在Netty层面，零拷贝体现在对于数据操作的优化。
- Netty的零拷贝体现在一下几个方面：

1. 使用Netty提供的CompositeByteBuf类，可以将多个ByteBuf合并为一个逻辑上的ByteBuf，避免了多个ByteBuf之间的拷贝。
2. ByteBuf支持slice操作，可以将一个ByteBuf分割成多个共享同一个存储区域的ByteBuf，避免了内存的拷贝。
3. 通过FileRegion包装的FileChannel.tranferTo实现文件传输，可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致内存拷贝的问题。

# 分布式

# 理论

## 拜占庭将军问题

## CAP

- 一致性、可用性、分区容错性
- 一致性：一个写操作返回成功，那么之后的读请求必须读到这个数据。如果返回失败，那么所有的读操作都不能读到这个数据，所有节点访问同一份最新的数据。
- 可用性：对数据更新具备高可用性，请求能够及时处理，不会一直等待，除非出现节点失效。
- 分区容错性：能容忍网络分区，在网络断开的情况下，被分割的节点仍能正常对外提供服务。
- 两个副本之间网络断开，不能通信，这时候如果一个副本更新，导致两个副本之间数据不一致，丧失了A的性质。为了保证不一致，将另一个副本置为不可用，丧失了A的性质。两个副本之间通信恢复，既保证了C和P，但是丧失了P的性质。
- 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，在预提交阶段表明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

- 需要消息队列提供相应的功能才能实现。
- 在订单系统中，存在创建订单和删除购物车两个功能，即存在订单系统和购物车系统。当用户下单时，删除购物车中的某一项是可以异步来完成的，允许非实时，在这样的情况下，可以通过消息队列，购物车系统订阅响应的消息来进行消费。但是也会存在着数据不一致等问题。必须订单创建失败但是购物车却被删除了。在这种情况下需要使用到分布式事务来解决。
- 当用户提交订单时，订单系统开启事务，向消息系统(消息队列)发送一个半消息，即事务提交之前该消息对消费者是透明的。之后订单系统再进行订单的增删改查，如果操作成功，向消息系统进行消息提交操作，否则进行回滚操作。如果进行了消息提交的操作，购物车系统就能够消费到这条消息的后续流程。

## 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。

**性能**
缓存 > Zookeeper >= 数据库
**可靠性**
Zookeeper > 缓存 > 数据库

# 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：一个partition只能被消费者组里的一个消费者消费。
- 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的特点

1. 高吞吐量、低延迟：kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个主题可以 分多个分区, 消费组对分区进行消费操作；
2. 可扩展性：kafka集群支持热扩展；
3. 持久性、可靠性：消息被持久化到本地磁盘，并且支持数据备份防止数据丢失；
4. 容错性：允许集群中节点失败（若副本数量为n,则允许n-1个节点失败）；
5. 高并发：支持数千个客户端同时读写；

# 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。

## 为什么Kafka的offset放到了Kafka的名为__consumer_offsets 的Topic中?

Kafka其实存在一个比较大的隐患，就是利用Zookeeper来存储记录每个消费者/消费者组的消费进度，虽然在使用过程中JVM帮助我们完成了一些优化，但是消费者需要频繁的去与Zookeeper进行交互，而ZKClient的API操作Zookeeper频繁的Write其本身是一个比较低效的action，对于后期水平扩展也是一个比较头疼的问题。如果期间Zookeeper集群发生了变化，那Kafka集群的吞吐量也跟着受影响。

# 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分区的选举机制

- kafka在所有broker中选出一个controller，所有partition的leader的选举都由controller决定。controller会将leader的改变直接通过rpc的方式通知需要为此做出反应的broker。同时controller也负责增删topic以及replica的重新分配。

1. controller在zookeeper注册watch，一旦有broker宕机，在zk中的节点会被删除。controller读取最新的幸存的broker。
2. controller决定set_p，该集合包含了宕机的broker上的所有partition。
3. 对set_p中的每一个partition，读取该partition当前的ISR。决定该partition的新leader。如果当前ISR中有至少一个Replica还幸存，则选择其中一个作为新的leader，新的ISR包含当前ISR中所有幸存的Replica。否则选择该partition中任意一个幸存的replica作为新的leader以及ISR。如果该partition的所有replica都宕机了，则将新的leader设置为-1。
4. 将新的leader，ISR和新的leader_epoch以及controller_epoch写入目录下。
5. 直接通过RPC向set_p相关的broker发送leaderandisrrequest命令。controller可以在一个rpc操作中发送多个命令从而提升效率。

- leaderandisrrequest响应过程：①若请求中的controller epoch小于最新的controller epoch，则直接返回stalecontrollerepochcode。②对于请求中partitionStateInfos中的每一个元素，若partitionStateInfo的leader epoch小于当前replicmanager中存储的leader epoch，返回staleleaderepochcode。否则如果当前brokerid在partitionstateinfo中，则将该partition及partition存入一个名为partitionState的Map中。③筛选出partitionState中leader与当前broker id相等的所有记录存入partitionsTobeleader中，其他记录存入partitionstobefollower中。④如果partitiontobeleader不为空，对其执行makeleaders方法。⑤如果partitiontobefollower不为空，对其执行makefollowers方法。⑥若高水位线程还未启动，将其启动，并将hwThreadInitialized设为true。⑦关闭所有idle状态的fetcher。

# OffSet的作用

kafka是顺序读写，具备很好的吞吐量。实现原理是

- 每次生产消息时，都是往对应partition的文件中追加写入，而消息的被读取状态是由consumer来维护的。所以每个partition中offset一般都是连续递增的（如果开启了压缩，因为对旧数据的merge会导致不连续）
- 被读取的消息并不会删除，所以每次都是追加写入顺序读写，具备很好的吞吐量。
- 实现过程是 consumer在消费消息后，向broker中有个专门维护每个consumer的offset的topic生产一条消息，记录自己当前已读的消息的offset+1的值作为新的offset的消息。当然在旧版本的实现是在zookeeper上有个节点存放这个offset，当时后面考虑性能问题，kafka改到了topic里，同时可以自由配置使用zookeeper还是使用topic。

# 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。

- **事务的详细解释**

1. 至少一次：(重试，消息可能会被重复被消费)分两种情况：①如果生产者的acks设置为-1或all，并且生产者在发送消息最后也收到了ack，这就意味着消息已经被精确一次写入了Kafka的topic。②如果生产者接收ack超时或者收到了错误，它就会认为消息没有写入Kafka的topic，然后会尝试重新发送消息。③如果broker恰好在消息已经成功写入Kafka的topic，但是在发送ack前发送了故障，那么生产者的重试机制就会导致这条消息被写入Kafka两次。
2. 至多一次：如果生产者在ack超时或返回错误的时候不重试发送消息。那么消息有可能最终并没有写入Kafka的topic中。这样就可能出现消息并没有被消费者消费的消息。
3. 精确一次：生产者有重试机制，且发送的消息只会被消费一次。即时生产者重试发送消息，也只会让消息被发送给消费者一次。

## 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。

# 软件开发过程



## 软件开发

一般软件设计或者说软件开发分两种：瀑布式，敏捷式。

前者一般是项目经理经过大量的业务分析后，会基于现有需求整理出一个基本模型，再将结果传递给开发人员，这就是开发人员的需求文档，他们只需要照此开发便是。这种模式下，是很难频繁的从用户那里得到反馈，因此在前期分析时就已经默认了这个业务模型是正确的，那么结果可想而之，数月甚至数年后交付的时候，必然和客户的预期差距较大。

后者在此基础上进行了改进，它也需要大量的分析，范围会设计到更精细的业务模块，它是小步迭代，周期性交付，那么获取客户的反馈也就比较频繁和及时。可敏捷也不能够将业务中的方方面面都考虑到，并且敏捷是拥抱变化的，大量的需求或者业务模型变更必将带来不小的维护成本，同时，对人（Developer）的要求也必然会更高。

DDD则不同：它像是更小粒度的迭代设计，它的最小单元是领域模型(Domain Model)，所谓领域模型就是能够精确反映领域中某一知识元素的载体，这种知识的获取需要通过与领域专家(Domain Expert)进行频繁的沟通才能将专业知识转化为领域模型。领域模型无关技术，具有高度的业务抽象性，它能够精确的描述领域中的知识体系；同时它也是独立的，我们还需要学会如何让它具有表达性，让模型彼此之间建立关系，形成完整的领域架构。通常我们可以用象形图或一种通用的语言(Ubiquitous Language)去描述它们之间的关系。在此之上，我们就可以进行领域中的代码设计(Domain Code Design)。如果将软件设计比做是造一座房子，那么领域代码设计就好比是贴壁纸。前者已经将房子的蓝图框架规划好，而后者只是一个小部分的设计：如果墙纸贴错了，我们可以重来，可如果房子结构设计错了，那可就悲剧了。

# 领域驱动设计分为两个阶段

1. 以一种领域专家、设计人员、开发人员都能理解的通用语言作为相互交流的工具，在交流的过程中发现领域概念，然后将这些概念设计成一个领域模型；
2. 由领域模型驱动软件设计，用代码来实现该领域模型；

# 领域驱动模式是很重要的，有一下几个特点

1. 领域模型是对具有某个边界的领域的一个抽象，反映了领域内用户业务需求的本质；领域模型是有边界的，只反应了我们在领域内所关注的部分；
2. 领域模型只反映业务，和任何技术实现无关；领域模型不仅能反映领域中的一些实体概念，如货物，书本，应聘记录，地址，等；还能反映领域中的一些过程概念，如资金转账，等；
3. 领域模型确保了我们的软件的业务逻辑都在一个模型中，都在一个地方；这样对提高软件的可维护性，业务可理解性以及可重用性方面都有很好的帮助；
4. 领域模型能够帮助开发人员相对平滑地将领域知识转化为软件构造；
5. 领域模型贯穿软件分析、设计，以及开发的整个过程；领域专家、设计人员、开发人员通过领域模型进行交流，彼此共享知识与信息；因为大家面向的都是同一个模型，所以可以防止需求走样，可以让软件设计开发人员做出来的软件真正满足需求；
6. 要建立正确的领域模型并不简单，需要领域专家、设计、开发人员积极沟通共同努力，然后才能使大家对领域的认识不断深入，从而不断细化和完善领域模型；
7. 为了让领域模型看的见，我们需要用一些方法来表示它；图是表达领域模型最常用的方式，但不是唯一的表达方式，代码或文字描述也能表达领域模型；
8. 领域模型是整个软件的核心，是软件中最有价值和最具竞争力的部分；设计足够精良且符合业务需求的领域模型能够更快速的响应需求变化；

# 领域通用语言

UML、伪代码

# 将领域模型转换为代码的最佳实践

开发人员会被加入到建模的过程中来。主要的想法是选择一个能够恰当在软件中表现的模型，这样设计过程会很顺畅并且基于模型。代码和其下的模型紧密关联会让代码更有意义并与模型更相关。有了开发人员的参与就会有反馈。

# 领域建模时思考问题的角度

1. 设计领域模型时不要以用户为中心作为出发点去思考问题，不能老想着用户会对系统做什么，而是应该从一个客观的角度，根据用户的需求挖掘出领域内的相关事务，思考这些事务的本质关联及其变化规律作为出发点去思考问题。
2. 领域建模是建立虚拟模型让我们现实的人使用，而不是建立虚拟空间，去模仿现实。

# 领域驱动设计的经典分层架构

1. 展现层：请求应用层以获取用户所需要展现的数据，发送命令给应用层要求执行某个用户的命令。
2. 应用层：定义软件要完成的所有任务，对外为展现层提供各种应用功能，对内调用领域层完成各种业务逻辑，应用层不应包含业务逻辑。
3. 领域层：负责表达业务概念，业务状态信息以及业务规则，领域模型处于这一层，是业务软件的核心。

4. 基础设施层：为其它层提供通用的技术能力，提供层与层之间的通信，为领域层实现持久化机制，总之，基础设施层可以通过架构和框架来支持其它层的技术需求。

## 领域驱动设计过程中使用的模式

1. 关联：关联本身不是一种模式，但是在设计过程中应该让关联尽量的少，复杂的关联容易形成对象的关系网，理解和维护单个对象不利。关联尽量保持单向的关联，找到关联的限制条件，往往这样的限制条件能够将关联化繁为简。
2. 实体：不应该为实体定义太多的属性和行为，应该寻找关联，发现其他一些实体或值对象，将属性或行为转移到其他关联的实体或值对象上。
3. 值对象
4. 领域服务：跨多个对象的操作。只有行为没有状态，强调是无状态的。它存在的意义是协调领域对象共同完成某个操作，所有的状态还都保存在对应的领域对象中。
5. 聚合及聚合根：聚合通过定义对象之间清晰的所属关系和边界来实现领域模型的内聚，并避免了错综复杂的难以维护的对象关系网的形成。聚合定义了一组具有内聚关系的相关对象的集合，我们把聚合看做是一个修改数据的单元。
6. 工厂：根据参数创建出领域对象。
7. 仓储：领域模型中的对象自从被创建出来就不会一直留在内存中，当它不活动的时候会被持久化到数据库中，然后当我们需要的时候会重建这个对象。

## 设计领域模型的一般步骤

1. 根据需求建立一个初步的领域模型，识别出一些明显的领域概念以及它们的关联，关联可以暂时没有方向但需要有（1：1，1：N，M：N）这些关系；可以用文字精确的没有歧义的描述出每个领域概念的涵义以及包含的主要信息；
2. 分析主要的软件应用程序功能，识别出主要的应用层的类；这样有助于及早发现哪些是应用层的职责，哪些是领域层的职责；
3. 进一步分析领域模型，识别出哪些是实体，哪些是值对象，哪些是领域服务；
4. 分析关联，通过对业务的更深入分析以及各种软件设计原则及性能方面的权衡，明确关联的方向或者去掉一些不需要的关联；
5. 找出聚合边界及聚合根，这是一件很有难度的事情；因为你在分析的过程中往往会碰到很多模棱两可的难以清晰判断的选择问题，所以，需要我们平时一些分析经验的积累才能找出正确的聚合根；
6. 为聚合根配备仓储，一般情况下是为一个聚合分配一个仓储，此时只要设计好仓储的接口即可；
7. 走查场景，确定我们设计的领域模型能够有效地解决业务需求；
8. 考虑如何创建领域实体或值对象，是通过工厂还是直接通过构造函数；
9. 停下来重构模型。寻找模型中觉得有些疑问或者是蹩脚的地方，比如思考一些对象应该通过关联导航得到还是应该从仓储获取？聚合设计的是否正确？考虑模型的性能怎样，等等；

# 算法

- KMP

# 几种常见的排序算法

## 时间空间复杂度

| 插入方法 | 平均时间复杂度 | 最差时间复杂度 | 最好时间复杂度 | 空间复杂度 |
| --- | --- | --- | --- | --- |
| 插入排序 | O(n2) | O(n2) | O(n) | O(1) |
| 希尔排序 | O(n1.3) | O(n2) | O(n) | O(1) |
| 选择排序 | O(n2) | O(n2) | O(n2) | O(1) |
| 堆排序 | O(nlogn) | O(nlogn) | O(nlogn) | O(1) |
| 冒泡排序 | O(n2) | O(n2) | O(n) | O(1) |
| 快速排序 | O(nlogn) | O(n2) | O(nlogn) | O(nlogn) |
| 归并排序 | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |

## 简单插入排序

```
public static void insertionSort(int[] a) {
        int size = a.length;
        for (int i = 1; i < size; i++) {
                int j = i - 1;
                for (; j >= 0; j--) {
                        if (a[i] > a[j]) {
                                break;
                        }
                }
                if (j != i - 1) {
                        int k = i-1;
                        int temp = a[i];
                        for (; k > j; k--) {
                                a[k + 1] = a[k];
                        }
                        a[j + 1] = temp;
                }
        }
}
```

## 希尔排序

```
public static void shellSort(int[] a) {
        int size = a.length;
        int gap, i, j;
        for (gap = size / 2; gap > 0; gap /= 2) {
                for (i = 0; i < gap; i++) {
                        for (j = i + gap; j < size; j += gap) {
                                if (a[j - gap] > a[j]) {
                                        int temp = a[j];
                                        int k = j - gap;
                                        while (k >= i && a[k] > temp) {
                                                a[k + gap] = a[k];
                                                k -= gap;
                                        }
                                        a[k + gap] = temp;
                                }
                        }
                }
        }
}
```

## 选择排序

```
public static void selectionSort(int[] a){
        int size = a.length;
        int min;
        for(int i =0;i<size;i++){
                min = i;
                for(int j = i;j<size;j++){
                        if(a[j] < a[min]){
                                min = j;
                        }
                }
                if(min != i){
                        int temp = a[i];
                        a[i] = a[min];
                        a[min] = temp;
                }
        }
}
```

## 快速排序

```
public static void quickSort(int[] a) {
        int size = a.length;
        helper(a, 0, size-1);
}
public static void helper(int[] a, int left, int right) {
        int l = left, r = right;
        if (l < r) {
                int x = a[l];
                while (l < r) {
                        while (l < r && a[r] > x) {
                                r--;
                        }
                        if (l < r) {
                                a[l++] = a[r];
                        }
                        while (l < r && a[l] < x) {
                                l++;
                        }
                        if (l < r) {
                                a[r--] = a[l];
                        }
                }
                a[l] = x;
                helper(a, left, l - 1);
                helper(a, l + 1, right);
        }
}
```

## 堆排序

```
public static void maxHeapDown(int[] a, int start, int end) {
        int p = start, c = 2 * p + 1;
        int temp = a[p];
        for (; c <= end; p = c, c = 2 * p + 1) {
                if (c < end && a[c] < a[c + 1]) {
                        c++;
                }
                if (temp < a[c]) {
                        a[p] = a[c];
                        a[c] = temp;
                } else {
                        break;
                }
        }
}
public static void heapSort(int[] a) {
        int size = a.length;
        for (int i = size / 2 - 1; i >= 0; i--) {
                maxHeapDown(a, i, size - 1);
        }
        for (int i = size - 1; i > 0; i--) {
                int temp = a[0];
                a[0] = a[i];
                a[i] = temp;
                maxHeapDown(a, 0, i - 1);
        }
}
```

## 归并排序

```
public static void merge(int[] a, int start, int mid, int end) {
        int[] res = new int[end - start + 1];   // 定义结果数组存放排好序的数组
        int s1 = start;                         // 遍历第一个有序数组的索引
        int s2 = mid + 1;                       // 遍历第二个有序数组的索引
        int k = 0;                              // 新建结果数组的索引
        while (s1 <= mid && s2 <= end) {
                if (a[s1] < a[s2]) {
                        res[k++] = a[s1++];
                } else {
                        res[k++] = a[s2++];
                }
        }
        while (s1 <= mid) {
                res[k++] = a[s1++];
        }
        while (s2 <= end) {
                res[k++] = a[s2++];
        }
        for (int i = 0; i < end - start + 1; i++) {
                a[start + i] = res[i];
        }
        res = null;
}
public static void mergeSortUpToDown(int[] a, int start, int end) {
        if (a == null || start >= end) {
                return;
        }
        int mid = start + (end - start) / 2;        // 取中间值
        mergeSortUpToDown(a, start, mid);           // 归并排序a[start...mid]
        mergeSortUpToDown(a, mid + 1, end);   // 归并排序a[mid+1...end]

        merge(a, start, mid, end);     // 将两个有序数组进行合并

}
public static void mergeGroups(int[] a, int length, int gap) {
        int i;
        int twoLen = 2 * gap;       //两个子数组的长度
        for (i = 0; i + twoLen - 1 < length; i += (twoLen)) {
                merge(a, i, i + gap - 1, i + 2 * gap - 1);
        }
        if (i + gap < length) {
                merge(a, i, i + gap, length - 1);
        }
}
public static void mergeSortDownToUp(int[] a) {
        int size = a.length;
        for (int i = 1; i < a.length; i *= 2) {
                mergeGroups(a, size, i);
        }
}
```

## 冒泡排序

```
public static void bubbleSort(int[] a) {
        boolean flag = false;
        int size = a.length;
        for (int i = size - 1; i > 0; i--) {
                flag = false;
                for (int j = 0; j < i; j++) {
                        if (a[j] > a[j + 1]) {
                                int temp = a[j];
                                a[j] = a[j+1];
                                a[j+1] = temp;
                                flag = true;
                        }
                }
                if(!flag){
                        break;
                }
        }
}
```

# LRU LFU

# LRU 哈希+双向链表版

```
class LRUCache {
        DNode head = new DNode(-1,-1);
        DNode tail = new DNode(-1,-1);
        int cap = 0;
        int maxCap = 0;
        Map<Integer, DNode> map = new HashMap<>();
        public LRUCache(int capacity) {
                this.maxCap = capacity;
                head.next = tail;
                tail.pre = head;
        }
        public int get(int key) {
                if (map.containsKey(key)) {
                        DNode node = map.get(key);
                        node.pre.next = node.next;
                        node.next.pre = node.pre;
                        node.pre = head;
                        node.next = head.next;
                        head.next = node;
                        node.next.pre = node;
                        return map.get(key).val;
                }
                return -1;
        }
        public void put(int key, int value) {
                if (map.containsKey(key)) {
                        // 如果存在该节点，更新该节点，并把该节点放到头部
                        DNode node = map.get(key);
                        node.val = value;
                        node.pre.next = node.next;
                        node.next.pre = node.pre;
                        node.pre = head;
                        node.next = head.next;
                        head.next = node;
                        node.next.pre = node;
                } else {
                        // 如果超过容量，找到最近未访问的节点，然后更新过期节点的前后节点关系。
                        // 之后删除哈希对应的key，然后将最新的值放到过期节点中，使其复活，并将当前节点放到头部
                        if (cap + 1 > maxCap) {
                                DNode staleNode = tail.pre;
                                staleNode.pre.next = staleNode.next;
                                staleNode.next.pre = staleNode.pre;
                                map.remove(staleNode.key);
                                staleNode.val = value;
                                staleNode.key = key;
                                map.put(key, staleNode);
                                staleNode.pre = head;
                                staleNode.next = head.next;
                                head.next = staleNode;
                                staleNode.next.pre = staleNode;
                        } else {
                                // 未超过容量，直接新建并放到头部
                                cap++;
                                DNode node = new DNode(key,value);
                                node.pre = head;
                                node.next = head.next;
                                head.next = node;
                                node.next.pre = node;
                                map.put(key, node);
                        }
                }
        }
}
class DNode {
        DNode pre;
        DNode next;
        int val;
        int key;
        public DNode(int key,int val) {
                this.val = val;
                this.key = key;
        }
}
```

## LRU LinkHashMap版

```
class LruCache extends LinkedHashMap<String,Object>{
        private int maxCapacity;
        public LruCache(int maxCapacity) {
                super(maxCapacity,0.75f, true);
                this.maxCapacity=maxCapacity;
        }
        @Override
        protected boolean removeEldestEntry(java.util.Map.Entry<String, Object> eldest) {
                return size()>maxCapacity;
        }
}
```

## LFU

```
class LFUCache {
        DFNode head = new DFNode(-1, -1, Integer.MAX_VALUE);
        DFNode tail = new DFNode(-1, -1, Integer.MIN_VALUE);
        int cap = 0;
        int maxCap = 0;
        Map<Integer, DFNode> map = new HashMap<>();
        public LFUCache(int capacity) {
                this.maxCap = capacity;
                head.next = tail;
                tail.pre = head;
        }
        public int get(int key) {
                if (map.containsKey(key)) {
                        DFNode node = map.get(key);
                        node.freq++;
                        updateNodePos(node);
                        return map.get(key).val;
                }
                return -1;
        }
        public void put(int key, int value) {
                if(maxCap == 0){
                        return;
                }
                if (map.containsKey(key)) {
                        DFNode node = map.get(key);
                        node.val = value;
                        node.freq++;
                        updateNodePos(node);
                } else {
                        if (cap + 1 > maxCap) {
                                DFNode staleNode = tail.pre;
                                staleNode.pre.next = staleNode.next;
                                staleNode.next.pre = staleNode.pre;
                                map.remove(staleNode.key);
                                staleNode.val = value;
                                staleNode.key = key;
                                staleNode.freq = 1;
                                map.put(key, staleNode);
                                staleNode.next = tail;
                                staleNode.pre = tail.pre;
                                tail.pre = staleNode;
                                staleNode.pre.next = staleNode;
                                updateNodePos(staleNode);
                        } else {
                                cap++;
                                DFNode node = new DFNode(key, value, 1);
                                node.next = tail;
                                node.pre = tail.pre;
                                tail.pre = node;
                                node.pre.next = node;
                                map.put(key, node);
                                updateNodePos(node);
                        }
                }
        }
        private void updateNodePos(DFNode node) {
                DFNode preNode = node.pre;
                node.next.pre = node.pre;
                node.pre.next = node.next;
                while (preNode.freq <= node.freq) {
                        preNode = preNode.pre;
                }
                node.pre = preNode;
                node.next = preNode.next;
                preNode.next = node;
                node.next.pre = node;
        }
}
class DFNode {
        DFNode pre;
        DFNode next;
        int val;
        int key;
        int freq = 1;
        public DFNode(int key, int val, int freq) {
```

```
                this.key = key;
                this.val = val;
                this.freq = freq;
        }
}
```

## LFU hash版本

```
class LFUCache {
    Map<Integer, Node> cache;  // 存储缓存的内容
    Map<Integer, LinkedHashSet<Node>> freqMap; // 存储每个频次对应的双向链表
    int size;
    int capacity;
    int min; // 存储当前最小频次
    public LFUCache(int capacity) {
        cache = new HashMap<> (capacity);
        freqMap = new HashMap<>();
        this.capacity = capacity;
    }
    public int get(int key) {
        Node node = cache.get(key);
        if (node == null) {
            return -1;
        }
        freqInc(node);
        return node.value;
    }
    public void put(int key, int value) {
        if (capacity == 0) {
            return;
        }
        Node node = cache.get(key);
        if (node != null) {
            node.value = value;
            freqInc(node);
        } else {
            if (size == capacity) {
                Node deadNode = removeNode();
                cache.remove(deadNode.key);
                size--;
            }
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            addNode(newNode);
            size++;
        }
    }
    void freqInc(Node node) {
        // 从原freq对应的链表里移除, 并更新min
        int freq = node.freq;
        LinkedHashSet<Node> set = freqMap.get(freq);
        set.remove(node);
        if (freq == min && set.size() == 0) {
            min = freq + 1;
        }
        // 加入新freq对应的链表
        node.freq++;
        LinkedHashSet<Node> newSet = freqMap.get(freq + 1);
        if (newSet == null) {
            newSet = new LinkedHashSet<>();
            freqMap.put(freq + 1, newSet);
        }
        newSet.add(node);
    }
    void addNode(Node node) {
        LinkedHashSet<Node> set = freqMap.get(1);
        if (set == null) {
            set = new LinkedHashSet<>();
            freqMap.put(1, set);
        }
        set.add(node);
        min = 1;
    }
    Node removeNode() {
        LinkedHashSet<Node> set = freqMap.get(min);
        Node deadNode = set.iterator().next();
        set.remove(deadNode);
        return deadNode;
    }
}
class Node {
    int key;
    int value;
    int freq = 1;
```

```
    public Node() {}
    public Node(int key, int value) {
        this.key = key;
        this.value = value;
    }
}
```

## 前缀树

```
class Trie {
        Trie[] children;
        boolean isEnd;
        public Trie() {
                children = new Trie[26];
        }
        public void insert(String word) {
                int len = word.length();
                Trie node = this;
                for (int i = 0; i < len; i++) {
                        int c = word.charAt(i) - 'a';
                        if (node.children[c] == null) {
                                node.children[c] = new Trie();
                        }
                        node = node.children[c];
                }
                node.isEnd = true;
        }
        public boolean search(String word) {
                Trie node = searchPrefix(word);
                return node != null && node.isEnd;
        }
        public boolean startsWith(String prefix) {
                return searchPrefix(prefix) != null;
        }
        public Trie searchPrefix(String word) {
                Trie node = this;
                int len = word.length();
                for (int i = 0; i < len; i++) {
                        int c = word.charAt(i) - 'a';
                        if (node.children[c] == null) {
                                return null;
                        }
                        node = node.children[c];
                }
                return node;
        }
}
```

## 并查集

```
public class QuickUnion {
        int[] root;
        int[] rank;
        public QuickUnion(int size) {
                root = new int[size];
                rank = new int[size];
                for (int i = 0; i < size; i++) {
                        root[i] = -1;
                        rank[i] = 1;
                }
        }
        public int find(int x) {
                if (root[x] == x) {
                        return x;
                }
                return root[x] = find(root[x]);
        }
        public void union(int x, int y) {
                int xRoot = find(x);
                int yRoot = find(y);
                if (xRoot != yRoot) {
                        if (rank[xRoot] > rank[yRoot]) {
                                root[yRoot] = xRoot;
                        } else if (rank[xRoot] < rank[yRoot]) {
                                root[xRoot] = yRoot;
                        } else {
                                root[xRoot] = yRoot;
                                rank[yRoot]++;
                        }
                }
        }
        public boolean connected(int x, int y) {
                return find(x) == find(y);
        }
}
```

# 设计模式

## 单例模式

```
public class Singleton {
    private Singleton(){}
    private static volatile Singleton singleton = null;
    private static Singleton getInstance(){
        if(singleton == null){
            synchronized (Singleton.class){
                if(singleton == null){
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

# 多线程相关算法

## 生产者消费者问题

```
class Depot {
    private int size;
    private int capacity;
    private Lock lock;
    private Condition fullCondition;
    private Condition emptyCondition;
    public Depot(int capacity) {
        this.capacity = capacity;
        lock = new ReentrantLock();
        fullCondition = lock.newCondition();
        emptyCondition = lock.newCondition();
    }
    public void producer(int no) {
        lock.lock();
        int left = no;
        try {
            while (left > 0) {
//              如果当前产品数量达到最大容量，将当前线程加入等待队列
                while (size >= capacity) {
                    System.out.println(Thread.currentThread() + " : before await");
//                  挂起线程，释放锁权限
                    fullCondition.await();
                    System.out.println(Thread.currentThread() + " : after await");
                }
//              增量等于正常加入的大小或者剩余容量的大小
                int inc = (capacity - size) > left ? left : (capacity - size);
//              用来判断当前是否已经完成生产，若left大于inc，则说明还有部分产品没有生产，等待当前队列唤醒后，继续生产。
//              否则不再生产，退出while循环。
                left -= inc;
                size += inc;
                System.out.println("producer : " + inc + ", size= " + size);
//              唤醒消耗产品队列
                emptyCondition.signal();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
//      生产完成，解锁
        lock.unlock();
    }

    //  消费者
    public void consumer(int no) {
//      上锁，该线程不能被其他线程打断
        lock.lock();
        int left = no;
        try {
//          要消耗的数量大于0时，继续消耗产品
            while (left > 0) {
//              当现有产品清零时，将消费线程阻塞，emptyCondition进入等待队列
                while (size <= 0) {
                    System.out.println(Thread.currentThread() + " : before await");
//                  挂起线程，释放锁权限
                    emptyCondition.await();
                    System.out.println(Thread.currentThread() + " : after await");
                }
//              消耗产品数量
                int dec = (size - left) > 0 ? left : size;
                left -= dec;
                size -= dec;
                System.out.println("consumer : " + dec + ", size= " + size);
                fullCondition.signal();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
//      消费完成，解锁
        lock.unlock();
    }
}
class ProducerCondition {
    private Depot depot;

    public ProducerCondition(Depot depot) {
        this.depot = depot;
    }
```

```
    public void produce(int no) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                depot.producer(no);
            }
        }, no + " producer thread").start();
    }
}
class ConsumerCondition {
    private Depot depot;

    public ConsumerCondition(Depot depot) {
        this.depot = depot;
    }
    public void consume(int no) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                depot.consumer(no);
            }
        }, no + " consumer thread").start();
    }
}
public class ProducerAndConsumerReentrantLockExample {
    public static void main(String[] args) {
//      初始化为500的容量
        Depot depot = new Depot(500);
        new ProducerCondition(depot).produce(500);
        new ProducerCondition(depot).produce(200);
        new ConsumerCondition(depot).consume(200);
        new ConsumerCondition(depot).consume(500);
    }
}
```

## 读者写者问题

```
public class Main {
    private final ReentrantLock lock ;   //定义锁
    private static int readCount = 0;    //读者的数量
    private Semaphore writeSemaphore ;   //写信号量
    public Main() {
        lock = new ReentrantLock();
        writeSemaphore = new Semaphore(1);
    }
    public static void main(String[] args) {
        Main main = new Main();
        Executor executors = Executors.newFixedThreadPool(4);
        executors.execute(main.new Reader());
        executors.execute(main.new Reader());
        executors.execute(main.new Writer());
        executors.execute(main.new Reader());

    }
    class Reader implements Runnable {

        @Override
        public void run() {
            before();             //读操作之前的操作
            read();               //读操作
            after();             //读操作之后的操作
        }
        public void before() {    //读操作之前的操作
            final ReentrantLock l = lock;
            l.lock();
            try {
                if(readCount == 0) {   //当有读者时，写者不能进入
                    writeSemaphore.acquire(1);
                }
                readCount += 1;
                System.out.println("有1位读者进入");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                l.unlock();
            }
        }

        public void read() {         //读操作
            System.out.println("当前有 " + readCount + " 位读者");
        }

        public void after() {        //读操作之后的操作
            final ReentrantLock l = lock;
            l.lock();
            try {
                readCount -= 1;
                System.out.println("有1位读者离开" );
                if(readCount == 0)     //当读者为0时，写者才可以进入
                    writeSemaphore.release(1);

            } finally {
                l.unlock();
            }
        }

    }
    class Writer implements Runnable {

        @Override
        public void run() {
            final ReentrantLock l = lock;
            l.lock();
            try {
                try {
                    writeSemaphore.acquire(1);     //同时只有一个写者可以进入
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("写者正在写");
                writeSemaphore.release(1);
            }  finally {
                l.unlock();
```

```
            }
        }
    }
}
```

## 哲学家进餐问题

```
class DiningPhilosophers {
//  叉子锁
    ReentrantLock[] reentrantLock = new ReentrantLock[5];
//  一个一个串行吃
    ReentrantLock reentrantLock4 = new ReentrantLock();
//  限制最多4个人拿起叉子
    Semaphore semaphore = new Semaphore(4);
    public DiningPhilosophers() {
        for(int i = 0; i<reentrantLock.length;i++){
            reentrantLock[i] = new ReentrantLock();
        }
    }
    /**
     * 思路1，打破循环等待条件，哲学家先拿编号小的叉子,这种思路可能产生死锁
     * @throws InterruptedException
     */
    public void wantsToEat(int philosopher,
                           Runnable pickLeftFork,
                           Runnable pickRightFork,
                           Runnable eat,
                           Runnable putLeftFork,
                           Runnable putRightFork) throws InterruptedException {

        int forks1 = philosopher;
        int forks2 = (philosopher+1)%5;
        reentrantLock[Math.min(forks1,forks2)].lock();
        reentrantLock[Math.max(forks1,forks2)].lock();
        pickLeftFork.run();
        pickRightFork.run();
        eat.run();
        putLeftFork.run();
        putRightFork.run();
        reentrantLock[Math.min(forks1,forks2)].unlock();
        reentrantLock[Math.max(forks1,forks2)].unlock();
    }
    /**
     * 思路2，奇数哲学家先拿起左边的叉子，再拿起右边的叉子，偶数哲学家先拿起右边的叉子，再拿起左边的叉子
     */
    public void wantsToEat1(int philosopher,
                           Runnable pickLeftFork,
                           Runnable pickRightFork,
                           Runnable eat,
                           Runnable putLeftFork,
                           Runnable putRightFork) throws InterruptedException {
        int left = philosopher;
        int right = (philosopher+1)%5;
        if(philosopher % 2 == 0){
            reentrantLock[right].lock();
            reentrantLock[left].lock();
        }else{
            reentrantLock[left].lock();
            reentrantLock[right].lock();
        }
        pickLeftFork.run();
        pickRightFork.run();
        eat.run();
        putLeftFork.run();
        putRightFork.run();
        if(philosopher % 2 == 0){
            reentrantLock[right].unlock();
            reentrantLock[left].unlock();
        }else{
            reentrantLock[left].unlock();
            reentrantLock[right].unlock();
        }
    }
    /**
     * 思路2，保证最多只有4个哲学家同时持有叉子，这样保证最少1个哲学家能够吃到面条
     */
    public void wantsToEat2(int philosopher,
                            Runnable pickLeftFork,
                            Runnable pickRightFork,
                            Runnable eat,
                            Runnable putLeftFork,
                            Runnable putRightFork) throws InterruptedException {
```

```
        int left = philosopher;
        int right = (philosopher+1)%5;
//      就餐人数加一
        semaphore.acquire();
//      拿起叉子
        reentrantLock[left].lock();
        reentrantLock[right].lock();
        pickLeftFork.run();
        pickRightFork.run();
        eat.run();
        putLeftFork.run();
        putRightFork.run();
//      放下叉子
        reentrantLock[left].unlock();
        reentrantLock[right].unlock();
//      就餐人数减一
        semaphore.release();
    }
    /**
     * 思路4，一个一个串行吃
     */
    public void wantsToEat3(int philosopher,
                            Runnable pickLeftFork,
                            Runnable pickRightFork,
                            Runnable eat,
                            Runnable putLeftFork,
                            Runnable putRightFork) throws InterruptedException {
        int left = philosopher;
        int right = (philosopher+1)%5;
//      拿起叉子
        reentrantLock4.lock();
        pickLeftFork.run();
        pickRightFork.run();
        eat.run();
        putLeftFork.run();
        putRightFork.run();
//      放下叉子
        reentrantLock4.unlock();

    }
}
```

# KMP

```
public class KMP {
        private int[][] dp;
        private String pat;
        public KMP(String pat) {
                int M = pat.length();
                dp = new int[M][256];
                this.pat = pat;
                dp[0][pat.charAt(0)] = 1;
                int X = 0;
                for (int j = 1; j < M; j++) {
                        for (int c = 0; c < 256; c++) {
                                dp[j][c] = dp[X][c];
                        }
                        dp[j][pat.charAt(j)] = j + 1;
                        X = dp[X][pat.charAt(j)];
                }
        }
        public int search(String txt) {
                int M = pat.length();
                int N = txt.length();
                int j = 0;
                for (int i = 0; i < N; i++) {
                        j = dp[j][txt.charAt(i)];
                        if (j == M) {
                                return i - M + 1;
                        }
                }
                return -1;
        }
}
```