# 1 计算机网络

## TCP/IP协议栈



* **应用层**：为特定应用程序提供数据传输服务，数据单位为报文。
* **表示层**：表示层提供各种用于应用层数据的编码和转换功能,确保一个系统的应用层发送的数据能被另一个系统的应用层识别。如果必要，该层可提供一种标准表示形式，用于将计算机内部的多种数据格式转换成通信中采用的标准表示形式。数据压缩和加密也是表示层可提供的转换功能之一。
* **会话层**：会话层就是负责建立、管理和终止表示层实体之间的通信会话。该层的通信由不同设备中的应用程序之间的服务请求和响应组成。
* **传输层**：传输层建立了主机端到端的链接，传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。该层向高层屏蔽了下层数据通信的细节，使高层用户看到的只是在两个传输实体间的一条主机到主机的、可由用户控制和设定的、可靠的数据通路。数据单元也称作数据包(packets)。但是，当你谈论TCP等具体的协议时又有特殊的叫法，TCP的数据单元称为TCP报文段 (segment)而UDP协议的数据单元称为UDP报文或用户数据报(datagram)。
* **网络层**：本层通过IP寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点，正确无误地按照地址传送给目的端的运输层。
* **数据链路层**：将比特组合成字节，再将字节组合成数据帧，使用链路层地址 (以太网使用MAC地址)来访问介质，并进行差错检测。数据链路层又分为2个子层：逻辑链路控制子层（LLC）和媒体访问控制子层（MAC）。MAC子层处理CSMA/CD算法、数据出错校验、成帧等；LLC子层定义了一些字段使上次协议能共享数据链路层。 在实际使用中，LLC子层并非必需的。
* **物理层**：实际最终信号的传输是通过物理层实现的。通过物理介质传输比特流。规定了电平、速度和电缆针脚。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆。

## TCP和UDP区别

* 1、传输控制协议（Transmission Control Protocol）TCP面向连接（如打电话要先拨号建立连接），可靠的数据传输服务，数据单位为报文段; 用户数据报协议UDP（User Datagram Protocol）是无连接的，即发送数据之前不需要建立连接，数据单位为用户数据报。
* 2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付
* 3、UDP具有较好的实时性，工作效率比TCP高，更适用于对高速传输和实时性有较高要求的通信或广播通信。
* 4、每一条TCP连接只能是点到点的，即TCP不支持组播或者广播传输模式。UDP支持一对一，一对多，多对一和多对多的交互通信。
* 5、TCP对系统资源要求较多，UDP对系统资源要求较少。

## TCP与UDP数据格式

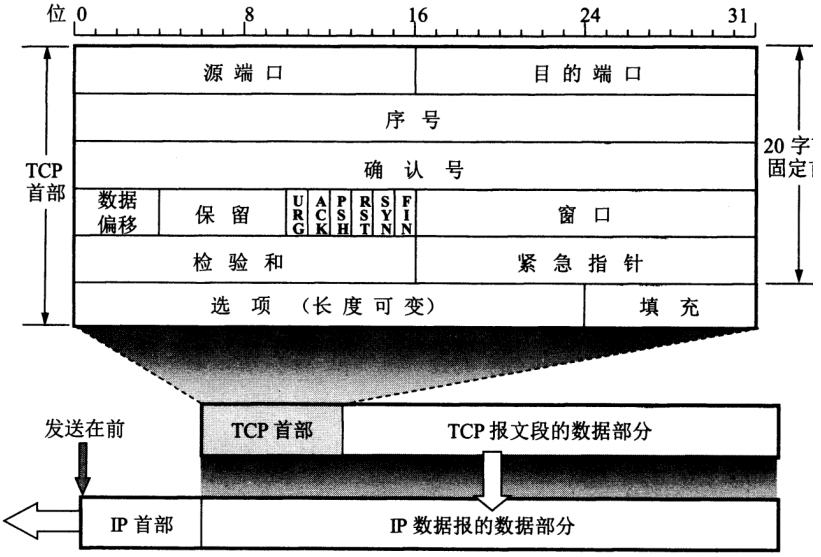


图1 TCP报文段的首部格式

* 序号：用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100字节，那么下一个报文段的序号应为 401。
* 确认号 ：期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
* 数据偏移 ：指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
* 确认ACK ：当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把ACK 置 1。
* 同步SYN ：在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
* 终止FIN ：用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
* 窗口 ：窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。



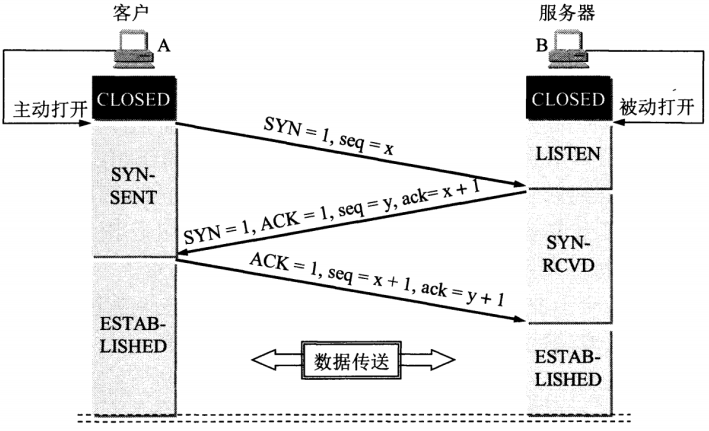
图2 UDP用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

## TCP三次握手与四次挥手

（1）三次握手

三次握手流程



假设 A 为客户端，B 为服务器端。

* 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
* A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
* B 收到连接请求报文，如果同意建立连接，则向 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的序号 y。
* A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
* B 收到 A 的确认后，连接建立。

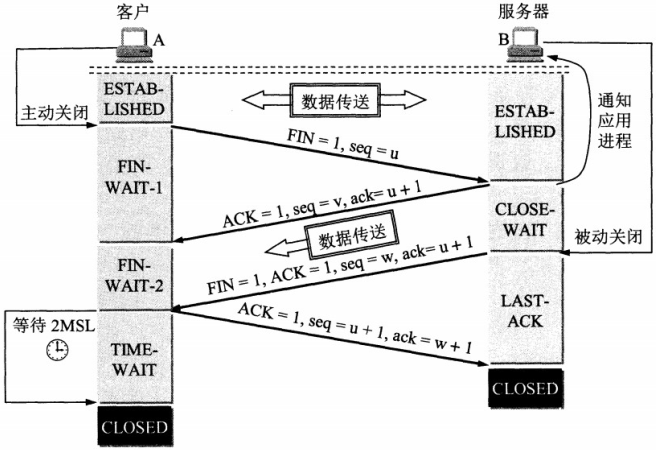
三次握手的原因

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

（2）TCP 的四次挥手

四次挥手过程



以下描述不讨论序号和确认号并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

* A 发送连接释放报文，FIN=1。
* B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
* 当 B 不再需要连接时，发送连接释放报文，FIN=1。
* A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。
* B 收到 A 的确认后释放连接。

四次挥手的原因

客户端发送了FIN连接释放报文之后，服务器收到了这个报文，就进入了CLOSE-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，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME\_WAIT客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

* 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
* 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

## TCP 粘包

在socket网络程序中，TCP和UDP分别是面向连接和非面向连接的。因此TCP的socket编程，收发两端（客户端和服务器端）都要有成对的socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle算法），将多次间隔较小、数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。

对于UDP，不会使用块的合并优化算法，这样，实际上目前认为，是由于UDP支持的是一对多的模式，所以接收端的skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的UDP包，在每个UDP包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。所以UDP不会出现粘包问题。

TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾；

1. 发送方原因

我们知道，TCP默认会使用Nagle算法。而Nagle算法主要做两件事：1）只有上一个分组得到确认，才会发送下一个分组；2）收集多个小分组，在一个确认到来时一起发送。所以，正是Nagle算法造成了发送方有可能造成粘包现象。

1. 接收方原因

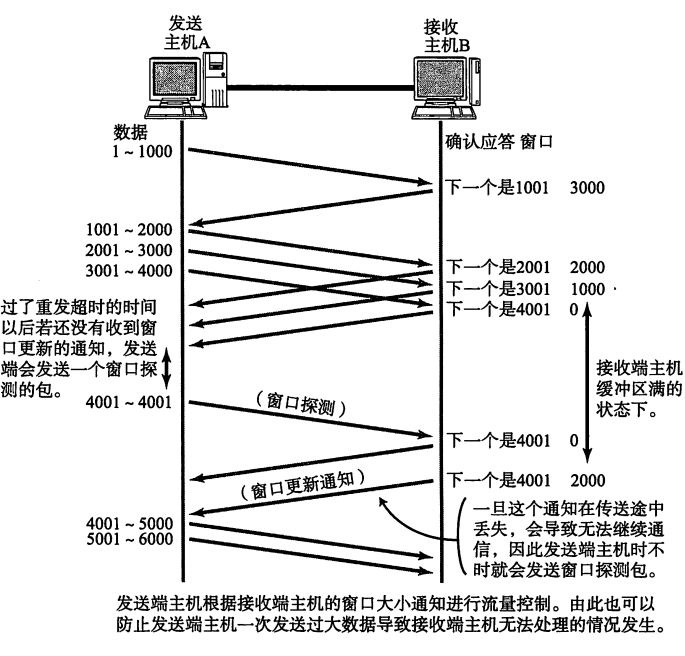
TCP接收到分组时，并不会立刻送至应用层处理，或者说，应用层并不一定会立即处理；实际上，TCP将收到的分组保存至接收缓存里，然后应用程序主动从缓存里读收到的分组。这样一来，如果TCP接收分组的速度大于应用程序读分组的速度，多个包就会被存至缓存，应用程序读时，就会读到多个首尾相接粘到一起的包。

1. 解决方法
2. 发送方：对于发送方造成的粘包现象，我们可以通过关闭Nagle算法来解决，使用TCP\_NODELAY选项来关闭Nagle算法。或者使用特殊字符控制；在包的首部添加数据包的长度。
3. 接收方：遗憾的是TCP并没有处理接收方粘包现象的机制，我们只能在应用层进行处理。
4. 应用层处理：应用层的处理简单易行！可以解决接收方造成的粘包问题，还能解决发送方造成的粘包问题。

## TCP流量控制与拥塞控制

1流量控制

1. 概念：接收端处理数据的速度是有限的，如果发送方的速度太快，就会把缓冲区打满。这时如果继续发送数据，就会导致丢包等一系列连锁反应。所以TCP支持根据接收端能力来决定发送端的发送速度。这个机制叫做流控制。
2. 机理（滑动窗口）
   * + 接收端将自己可以接收的缓冲区大小放入 TCP 首部中16位的“窗口大小”字段，通过ACK端通知发送端；窗口大小字段越大，说明网络的吞吐量越高；（实际窗口大小是窗口字段的值左移M（窗口扩大因子）位）
     + 接收端一旦发现自己的缓冲区快满了，就会将窗口大小设置成一个更小的值通知给发送端；发送端接受到这个窗口之后，就会减慢自己的发送速度；
     + 如果接收端缓冲区满了，就会将窗口置为0，这时发送方不再发送数据，但是需要定期发送一个窗口探测数据段, 使接收端把窗口大小告诉发送端。



* + - 当某个ACK报文丢失了，就会出现A等待B确认，并且B等待A发送数据的死锁状态。为了解决这种问题，TCP引入了持续计时器（Persistence timer），当A收到rwnd=0时，就启用该计时器，时间到了则发送一个1字节的探测报文，询问B是很忙还是上个ACK丢失了，然后B回应自身的接收窗口大小，返回仍为0（A重设持续计时器继续等待）或者会重发rwnd=x。

2拥塞控制

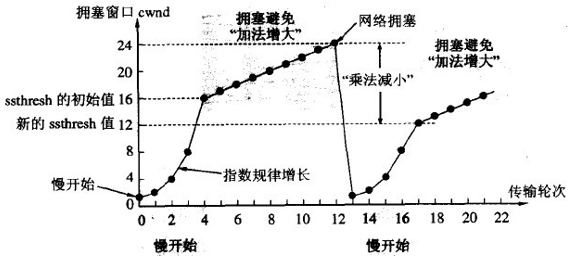
1. 概念

对资源的需求超过了可用的资源。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降。

1. 方法
2. 慢开始
   * + 发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。慢开始算的思路就是：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。初始值为1，每次收到一个ACK应答，拥塞窗口加倍。
     + 为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：

|  |
| --- |
| 当cwnd<ssthresh时，使用慢开始算法。  当cwnd>ssthresh时，改用拥塞避免算法。  当cwnd=ssthresh时，慢开始与拥塞避免算法任意。 |

* + - 拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。
    - 无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。如下图：



A . 超时重传机制

TCP中设计了超时重传机制。该机制规定当发送方A向B发送数据包P1时，开启时长为RTO（Retransmission Timeout）的重传定时器，如果A在RTO内未收到B对P1的确认报文，则认为P1在网络中丢失，此时重新发送P1。由此，引出RTO大小的设定问题。

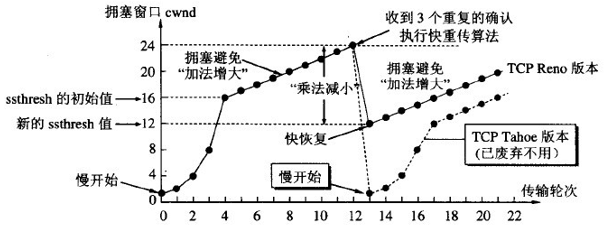
决定报文是否有必要重传的主要机制是重传计时器（retransmission timer），它的主要功能是维护重传超时（RTO）值。当报文使用TCP传输时，重传计时器启动，收到ACK时计时器停止。报文发送至接收到ACK的时间称为往返时间（RTT）。对若干次时间取平均值，该值用于确定最终RTO值。在最终RTO值确定之前，确定每一次报文传输是否有丢包发生使用重传计时器，下图说明了TCP重传过程。

　当报文发送之后，但接收方尚未发送TCP ACK报文，发送方假设源报文丢失并将其重传。重传之后，RTO值加倍；如果在2倍RTO值到达之前还是没有收到ACK报文，就再次重传。如果仍然没有收到ACK，那么RTO值再次加倍。如此持续下去，每次重传RTO都翻倍，直到收到ACK报文或发送方达到配置的最大重传次数。

　　最大重传次数取决于发送操作系统的配置值。默认情况下，Windows主机默认重传5次。大多数Linux系统默认最大15次。两种操作系统都可配置。

1. 快重传和快恢复
   * + 快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认（为什么是3个ACK？）就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。
     + 快重传配合使用的还有快恢复算法，有以下两个要点：

|  |
| --- |
| * 当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半。但是接下去并不执行慢开始算法。 * 考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。 |



3流量控制与拥塞控制的区别

1. 拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制。
2. 所谓流量控制就是让发送发送速率不要过快，让接收方来得及接收。利用滑动窗口机制就可以实施流量控制。原理这就是运用TCP报文段中的窗口大小字段来控制，发送方的发送窗口不可以大于接收方发回的窗口大小。

## 为何快速重传是选择3次ACK？

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。两次ACK时很可能是乱序造成的。三次ACK时很可能是丢包造成的。四次ACK更更更可能是丢包造成的。但是这样的响应策略太慢。丢包肯定会造成三次ACK。综上是选择收到三个重复确认时窗口减半效果最好，这是实践经验。

在没有fast retransmit / recovery 算法之前，重传依靠发送方的retransmit timeout，就是在timeout内如果没有接收到对方的ACK，默认包丢了，发送方就重传；

包的丢失原因

* + - 1）包checksum 出错 ；
    - 2）网络拥塞；
    - 3）网络断，包括路由重收敛；

但是发送方无法判断是哪一种情况，于是采用最笨的办法，就是将自己的发送速率减半，即CWND 减为1/2，这样的方法对2是有效的，可以缓解网络拥塞，3则无所谓，反正网络断了，无论发快发慢都会被丢；但对于1来说，丢包是因为偶尔的出错引起，一丢包就对半减速不合理。于是有了fast retransmit 算法，基于在反向还可以接收到ACK，可以认为网络并没有断，否则也接收不到ACK，如果在timeout 时间内没有接收到> 2 的duplicated ACK，则概率大事件为乱序，乱序无需重传，接收方会进行排序工作；而如果接收到三个或三个以上的duplicated ACK，则大概率是丢包，可以逻辑推理，发送方可以接收ACK，则网络是通的，可能是1、2造成的，先不降速，重传一次，如果接收到正确的ACK，则一切OK，流速依然（包出错被丢）。而如果依然接收到duplicated ACK，则认为是网络拥塞造成的，此时降速则比较合理。

## TCP可靠传输

1. 序号：TCP首部的序号字段用来保证数据能有序提交给应用层，TCP把数据看成无结构的有序的字节流。数据流中的每一个字节都编上一个序号字段的值是指本报文段所发送的数据的第一个字节序号。
2. 确认：TCP首部的确认号是期望收到对方的下一个报文段的数据的第一个字节的序号；
3. 重传：超时重传和冗余ACK重传
4. 流量控制：TCP采用大小可变的滑动窗口进行流量控制，窗口大小的单位是字节。发送窗口在连接建立时由双方商定。但在通信的过程中，接收端可根据自己的资源情况，随时动态地调整对方的发送窗口上限值(可增大或减小)。
   * 接受窗口rwnd，接收端缓冲区大小。接收端将此窗口值放在 TCP 报文的首部中的窗口字段，传送给发送端。
   * 拥塞窗口cwnd，发送缓冲区大小。
   * 发送窗口swnd，发送窗口的上限值= Min [rwnd, cwnd]。
5. 拥塞控制；

## 什么是DDos攻击？

* DoS（拒绝服务，Denial of Service）就是利用合理的服务请求来占用过多的服务资源，从而使合法用户无法得到服务的响应。这是早期非常基本的网络攻击方式。
* 分布式拒绝服务(DDoS：Distributed Denial of Service)攻击，是指攻击者利用大量“肉鸡”对攻击目标发动大量的正常或非正常请求、耗尽目标主机资源或网络资源，从而使被攻击的主机不能为正常用户提供服务。当服务器被DDos攻击时，一般会出现以下现象：
  + - 被攻击主机上有大量等待的TCP连接；
    - 网络中充斥着大量的无用的数据包；
    - 受害主机无法正常和外界通讯；
    - 受害主机无法处理所有正常请求；严重时会造成系统死机。
    - 对于用户来说，在常见的现象就是网站无法访问。
* 解决方式
  + 如果可以识别出攻击源，如机器IP等，可以在防火墙服务器上放置一份 ACL（访问控制列表) 来阻断这些来自这些 IP 的访问。
  + 对于带宽消耗型攻击，最有效的办法那就是增加带宽。
  + 提高服务器的服务能力，增加负载均衡，多地部署等。
  + 启用路由器或防火墙的反IP欺骗功能。
  + 付费，使用第三方的服务来保护你的网站。

## TCP洪水攻击

SYN Flood是当前最流行的DoS（拒绝服务攻击）与DDoS（分布式拒绝服务攻击）的方式之一，这是一种利用TCP协议缺陷，发送大量伪造的TCP连接请求，常用假冒的IP或IP号段发来海量的请求连接的第一个握手包（SYN包），被攻击服务器回应第二个握手包（SYN+ACK包），因为对方是假冒IP，对方永远收不到包且不会回应第三个握手包。导致被攻击服务器保持大量SYN\_RECV状态的“半连接”，并且会重试默认5次回应第二个握手包，塞满TCP等待连接队列，资源耗尽（CPU满负荷或内存不足），让正常的业务请求连接不进来。

方法：

* 第一个参数tcp\_synack\_retries = 0是关键，表示回应第二个握手包（SYN+ACK包）给客户端IP后，如果收不到第三次握手包（ACK包）后，不进行重试，加快回收“半连接”，不要耗光资源。（最关键参数，默认为5，修改为0 表示不要重发）
* 第二个参数(半连接队列长度)net.ipv4.tcp\_max\_syn\_backlog = 200000也重要，具体多少数值受限于内存。

## 一次完整的HTTP请求

* 1.对www.baidu.com这个网址进行DNS域名解析，得到对应的IP地址
* 2.根据这个IP，找到对应的服务器，发起TCP的三次握手
* 3.建立TCP连接后发起HTTP请求
* 4.服务器响应HTTP请求，浏览器得到html代码
* 5.浏览器解析html代码，并请求（从服务器上请求）html代码中的资源（如js、css图片等）（先得到html代码，才能去找这些资源）
* 6.浏览器对页面进行渲染呈现给用户

## HTTP请求格式与响应格式

（1）请求格式。主要有四部分组成，分别是：请求行、请求头、空行、消息体，每部分内容占一行

GET lindex_html HTTP/I _ 1 
Date: Thu, 20 May 2004 21:12:55 GMT 
connection: close 
Host: wwn.v_myfavoriteamazjngsite.com 
From: joebloe@someweositesomewnere.com 
Accept: text/html, text/plain 
User-Agent: Mozjlla/4.O (compatible; MSIE 6.0; Windows NT 5.1) 
Request Line 
General Headers 
Request Headers 
HTTP 
Request 
Entity Headers 
Message Body 

* 请求行：由三部分组成，GET/POST请求方法、请求资源URL、HTTP版本号
* 请求头：和缓存相关的头（Cache-Control，If-Modified-Since），客户端身份信息（User-Agent）等等
* 消息体：客户端发给服务端的请求数据，这部分数据并不是每个请求必须的。

（2）响应格式。包括：状态行、响应头、空行、消息体。每部分内容占一行。

HTTP/I.1 200 OK 
Date: Tnu, 20 May 2004 21:12:58 GMT 
Connection: close 
server: Apacne,'l .3.27 
Accept-Ranges: bytes 
Content-Type: text/html 
content-Length: 170 
Last-Modified: Tue, 18 May 2004 10:14:49 GMT 
<head> 
to the Amazjng 
<bodp 
site is under construction. Please come 
back later 
ontml> 
Status Line 
General Headers 
Response Headers 
Entity Headers 
HTTP 
Response 
Message Body 

* 状态行：HTTP协议版本号，状态码和状态说明三部分构成
* 响应头：响应头是服务器传递给客户端用于说明服务器的一些信息（Content-Type，charset等），以及将来继续访问该资源时的策略。
* 响应体：返回给客户端的HTML文本内容，或者其他格式的数据，比如：视频流、图片或者音频数据。

## HTTP状态码

服务器返回的响应报文中第一行为状态行，包含了状态码以及原因短语，用来告知客户端请求的结果。



**1XX 信息**

* 100 Continue ：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。

2XX 成功

* **200 OK**
* 204 No Content ：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
* 206 Partial Content：表示客户端进行了范围请求，响应报文包含由 Content-Range 指定范围的实体内容。

**3XX 重定向**

* 301 Moved Permanently ：永久性重定向
* 302 Found ：临时性重定向
* 303 See Other ：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。

注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会在 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。

* 304 Not Modified ：如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-NoneMatch，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。
* 307 Temporary Redirect ：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的POST 方法改成 GET 方法。

**4XX 客户端错误**

* 400 Bad Request ：请求报文中存在语法错误。
* 401 Unauthorized ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
* 403 Forbidden ：请求被拒绝。
* 404 Not Found

**5XX 服务器错误**

* 500 Internal Server Error ：服务器正在执行请求时发生错误。
* 503 Service Unavailable ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。
* 504 - Gateway Timeout:网关超时，由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答

## HTTP1.0与HTTP1.1区别

1 可扩展性

* 在消息中增加版本号，用于兼容性判断。
* HTTP/1.1增加了OPTIONS方法，它允许客户端获取一个服务器支持的方法列表。
* 为了与未来的协议规范兼容，HTTP/1.1在请求消息中包含了Upgrade头域，通过该头域，客户端可以让服务器知道它能够支持的其它备用通信协议，服务器可以据此进行协议切换，使用备用协议与客户端进行通信。

2 缓存

* 在HTTP/1.0中，使用Expire头域来判断资源的fresh或stale，并使用条件请求（conditional request）来判断资源是否仍有效。此外，HTTP/1.1在1.0的基础上加入了一些cache的新特性，当缓存对象的Age超过Expire时变为stale对象，cache不需要直接抛弃stale对象，而是与源服务器进行重新激活（revalidation）。

3 带宽优化

* HTTP/1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了。例如，客户端只需要显示一个文档的部分内容，又比如下载大文件时需要支持断点续传功能，而不是在发生断连后不得不重新下载完整的包。

HTTP/1.1中在请求消息中引入了range头域，它允许只请求资源的某个部分。在响应消息中Content-Range头域声明了返回的这部分对象的偏移值和长度。如果服务器相应地返回了对象所请求范围的内容，则响应码为206（Partial Content），它可以防止Cache将响应误以为是完整的一个对象。

另外一种情况是请求消息中如果包含比较大的实体内容，但不确定服务器是否能够接收该请求（如是否有权限），此时若贸然发出带实体的请求，如果被拒绝也会浪费带宽。

* HTTP/1.1加入了一个新的状态码100（Continue）。客户端事先发送一个只带头域的请求，如果服务器因为权限拒绝了请求，就回送响应码401（Unauthorized）；如果服务器接收此请求就回送响应码100，客户端就可以继续发送带实体的完整请求了。注意，HTTP/1.0的客户端不支持100响应码。但可以让客户端在请求消息中加入Expect头域，并将它的值设置为100-continue。

4 长连接

* HTTP 1.0规定浏览器与服务器只保持短暂的连接，浏览器的每次请求都需要与服务器建立一个TCP连接，服务器完成请求处理后立即断开TCP连接，服务器不跟踪每个客户也不记录过去的请求。此外，由于大多数网页的流量都比较小，一次TCP连接很少能通过slow-start区，不利于提高带宽利用率。
* HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。例如：一个包含有许多图像的网页文件的多个请求和应答可以在一个连接中传输，但每个单独的网页文件的请求和应答仍然需要使用各自的连接。
* HTTP 1.1还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，以保证客户端能够区分出每次请求的响应内容，这样也显著地减少了整个下载过程所需要的时间。（支持同时打开多个 TCP 连接）。

5 消息传递

* HTTP消息中可以包含任意长度的实体，通常它们使用Content-Length来给出消息结束标志。但是，对于很多动态产生的响应，只能通过缓冲完整的消息来判断消息的大小，但这样做会加大延迟。如果不使用长连接，还可以通过连接关闭的信号来判定一个消息的结束。

HTTP1.1支持分块传输编码，在HTTP/1.1中引入了Chunkedtransfer-coding来解决上面这个问题，发送方将消息分割成若干个任意大小的数据块，每个数据块在发送时都会附上块的长度，最后用一个零长度的块作为消息结束的标志。这种方法允许发送方只缓冲消息的一个片段，避免缓冲整个消息带来的过载。

在HTTP/1.0中，有一个Content-MD5的头域，要计算这个头域需要发送方缓冲完整个消息后才能进行。而HTTP/1.1中，采用chunked分块传递的消息在最后一个块（零长度）结束之后会再传递一个拖尾（trailer），它包含一个或多个头域，这些头域是发送方在传递完所有块之后再计算出值的。发送方会在消息中包含一个Trailer头域告诉接收方这个拖尾的存在。

6 Host头域

* 在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。
* HTTP1.1的请求消息和响应消息都支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。此外，服务器应该接受以绝对路径标记的资源请求。HTTP1.1支持虚拟主机。

7 错误提示

* HTTP/1.0中只定义了16个状态响应码，对错误或警告的提示不够具体。HTTP/1.1引入了一个Warning头域，增加对错误或警告信息的描述。
* 在HTTP/1.1中新增了24个状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

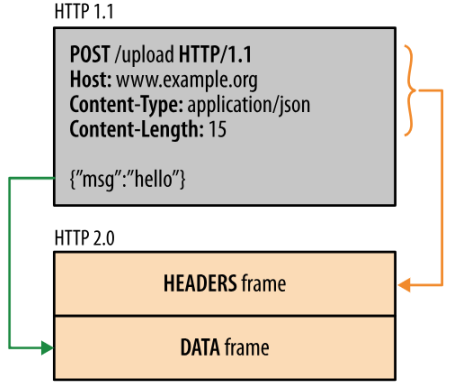
## HTTP2.0

* HTTP1.x缺陷

HTTP/1.x 实现简单是以牺牲性能为代价的：

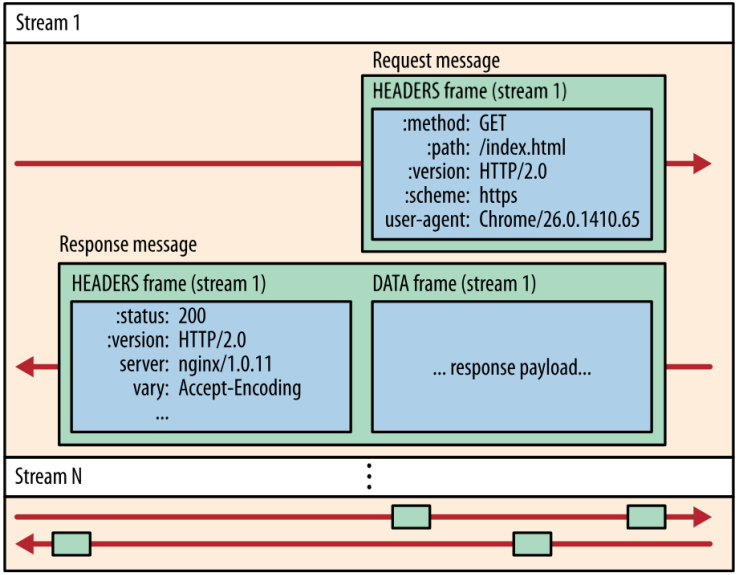
* + - 客户端需要使用多个连接才能实现并发和缩短延迟；
    - 不会压缩请求和响应首部，从而导致不必要的网络流量；
    - 不支持有效的资源优先级，致使底层 TCP 连接的利用率低下。
* 二进制分帧层

HTTP/2.0 将报文分成 HEADERS 帧和 DATA 帧，它们都是二进制格式的。

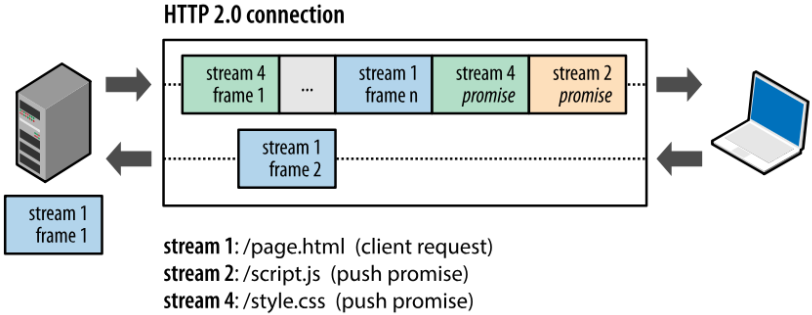


在通信过程中，只会有一个 TCP 连接存在，它承载了任意数量的双向数据流（Stream）。

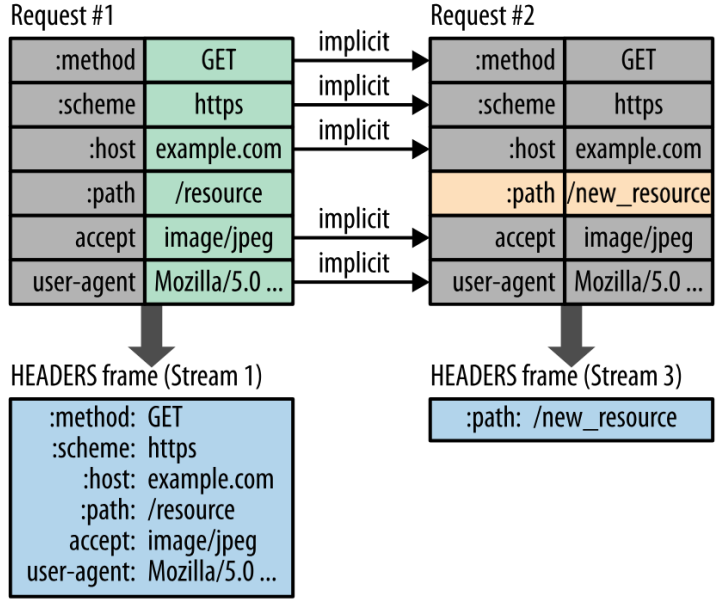
* 一个数据流（Stream）都有一个唯一标识符和可选的优先级信息，用于承载双向信息。
* 消息（Message）是与逻辑请求或响应对应的完整的一系列帧。
* 帧（Frame）是最小的通信单位，来自不同数据流的帧可以交错发送，然后再根据每个帧头的数据流标识符重新组装。



* 服务端推送
* HTTP/2.0 在客户端请求一个资源时，会把相关的资源一起发送给客户端，客户端就不需要再次发起请求了。例如客户端请求page.html 页面，服务端就把script.js和style.css等与之相关的资源一起发给客户端。



* 首部压缩
* HTTP/1.1 的首部带有大量信息，而且每次都要重复发送。
* HTTP/2.0 要求客户端和服务器同时维护和更新一个包含之前见过的首部字段表，从而避免了重复传输。不仅如此，HTTP/2.0 也使用 Huffman 编码对首部字段进行压缩。



## HTTP与HTTPS区别

1 HTTP和HTTPS的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

2 HTTPS和HTTP的区别

* 1、https协议需要到ca申请证书，一般免费证书很少，需要交费。
* 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
* 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443.
* 4、http的连接很简单，是无状态的，https协议是由SSL（安全套接字层）+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。
* 5、HTTPS比HTTP页面加载慢一些，不如HTTP高效，但更安全。
* 在OSI模型中，HTTP工作于应用层，而HTTPS工作于传输层；

3 HTTPS加密原理

* **对称加密**：数据使用 A 密钥加密，变成加密后的数据，加密后的数据使用 A 密钥解密，获取数据。
* **非对称加密**：使用一对非对称的密钥，一把叫私钥，一把叫公钥。**公钥加密的密文只能用私钥解密，私钥加密的密文只能用公钥解密。**公钥我们可以随便公开，因为别人知道了公钥毫无用处，经过公钥加密后的密文只能通过私钥来解密。

但由于公开密钥比共享密钥要慢，所以就需要综合一下两者的优缺点，使他们共同使用，而这也是HTTPS采用的加密方式。**在交换密钥阶段使用公钥加密，之后建立通信交换报文阶段则使用私钥加密。**

**如何确保公钥是真的呢？**为了解决公钥在传输过程中被攻击者替换，可以使用**由数字证书认证机构（CA）颁发的公钥证书。**接收到证书的客户端可以使用数字证书认证机构的公开密钥，对那张证书上的数字签名进行验证，一旦验证通过，客户端便可以明确两件事：

* 1 认证服务器的公开密钥的是真实有效的数字证书认证机构。
* 2 服务器的公开密钥是值得信赖的。
* **HTTPS通信过程**：
* 1. 客户端发起HTTPS请求

这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的443端口。

* 2. 服务端的配置

采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl就是个不错的选择，有1年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

* 3. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

* 4. 客户端解析证书

这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

* 5. 传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

* 6. 服务段解密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

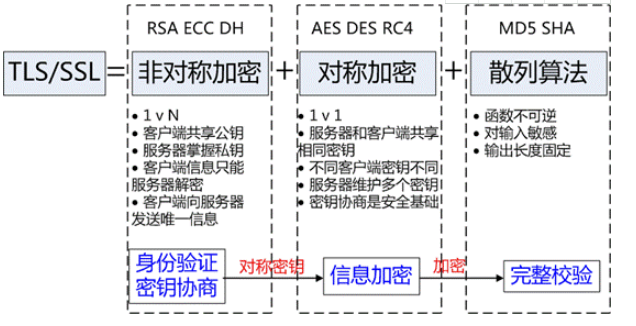
* 7. 传输加密后的信息

这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

* 8. 客户端解密信息

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

TLS/SSL的功能实现主要依赖于三类基本算法：散列函数 Hash、对称加密和非对称加密，其利用非对称加密实现身份认证和密钥协商，对称加密算法采用协商的密钥对数据加密，基于散列函数验证信息的完整性。



TLS的基本工作方式是，客户端使用非对称加密与服务器进行通信，实现身份验证并协商对称加密使用的密钥，然后对称加密算法采用协商密钥对信息以及信息摘要进行加密通信，不同的节点之间采用的对称密钥不同，从而可以保证信息只能通信双方获取。

一般情况下将二者写在一起TLS/SSL，我们可以将二者看做同一类协议，只不过TLS是SSL的升级版。

## GET与POST的区别

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。

1. url可见性：

get，参数url可见

post，url参数不可见

\*\*get把请求的数据放在url上，即HTTP协议头上，其格式为：以?分割URL和传输数据，参数之间以&相连；post把数据放在HTTP的包体内（requrest body）

1. 传输数据的大小：

get一般传输数据大小不超过2k-4k

post请求传输数据的大小根据php.ini 配置文件设定，也可以无限大

\*\*get提交的数据最大是2k（原则上url长度无限制，那么get提交的数据也没有限制咯？限制实际上取决于浏览器，浏览器通常都会限制url长度在2K个字节，即使(大多数)服务器最多处理64K大小的url，也没有卵用）；

post理论上没有限制。实际上IIS4中最大量为80KB，IIS5中为100KB。

1. 数据传输上：

get，通过拼接url进行传递参数

post，通过body体传输参数

\*\*GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据);

POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

1. 后退页面的反应：

get请求页面后退时，不产生影响

post请求页面后退时，会重新提交请求

\*\*GET在浏览器回退时是无影响的，POST会再次提交请求

1. 缓存性：

get请求是可以缓存的

post请求不可以缓存

\*\*GET请求会被浏览器主动cache，而POST不会，除非手动设置。

1. 安全性：

都不安全，原则上post肯定要比get安全，毕竟传输参数时url不可见，但也挡不住部分人闲的没事在那抓包玩，浏览器还会缓存get请求的数据。安全性个人觉得是没多大区别的，防君子不防小人就是这个道理。对传递的参数进行加密，其实都一样。

## Cookie与Session（解决HTTP无状态）？

http无状态协议，是指协议对于交互性场景没有记忆能力。

Cookie的工作原理：（Cookie信息都是放在请求头和响应头里）

（1）浏览器端第一次发送请求到服务器端

（2）服务器端创建Cookie，该Cookie中包含用户的信息，然后将该Cookie发送到浏览器端

（3）浏览器端再次访问服务器端时会携带服务器端创建的Cookie

（4）服务器端通过Cookie中携带的数据区分不同的用户。

Session的工作原理：

（1）浏览器端第一次发送请求到服务器端，服务器端创建一个Session，同时会创建一个特殊的Cookie（name为JSESSIONID的固定值，value为session对象的ID），然后将该Cookie发送至浏览器端

（2）浏览器端发送第N（N>1）次请求到服务器端,浏览器端访问服务器端时就会携带该name为JSESSIONID的Cookie对象

（3）服务器端根据name为JSESSIONID的Cookie的value(sessionId),去查询Session对象，从而区分不同用户。

可是session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

name为JSESSIONID的Cookie不存在（关闭或更换浏览器），返回1中重新去创建Session与特殊的Cookie

name为JSESSIONID的Cookie存在，根据value中的SessionId去寻找session对象

value为SessionId不存在\*\*（Session对象默认存活30分钟）\*\*，返回1中重新去创建Session与特殊的Cookie

value为SessionId存在，返回session对象。

Session和Cookie的区别：

(1)cookie数据存放在客户的浏览器上，session数据放在服务器上

(2)cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗,如果主要考虑到安全应当使用session

(3)session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用COOKIE

(4)单个cookie在客户端的限制是3K，就是说一个站点在客户端存放的COOKIE不能3K。

(5)所以：将登陆信息等重要信息存放为SESSION;其他信息如果需要保留，可以放在COOKIE中。

## 单点登录Cookie被禁用？

单点登录的原理是后端生成一个 session ID，然后设置到 cookie，后面的所有请求浏览器都会带上 cookie，然后服务端从 cookie 里获取 session ID，再查询到用户信息。所以，保持登录的关键不是 cookie，而是通过cookie 保存和传输的 session ID，其本质是能获取用户信息的数据。

系统中的确要考虑cookie被禁用的情况，可以把当前session的ID存入服务器中，请求页面的时候要带上这个ID去本地服务器中去查询当前用户是否已经登录或者存在。

## Token 的介绍和与session的区别？

Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。最简单的token组成:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign(签名，由token的前几位+盐以哈希算法压缩成一定长的十六进制字符串，可以防止恶意第三方拼接token请求服务器)。

使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

传统身份验证

HTTP 是一种没有状态的协议，也就是它并不知道是谁是访问应用。这里我们把用户看成是客户端，客户端使用用户名还有密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证一下。

解决的方法就是，当用户请求登录的时候，如果没有问题，我们在服务端生成一条记录，这个记录里可以说明一下登录的用户是谁，然后把这条记录的 ID 号发送给客户端，客户端收到以后把这个 ID 号存储在 Cookie 里，下次这个用户再向服务端发送请求的时候，可以带着这个 Cookie ，这样服务端会验证一个这个 Cookie 里的信息，看看能不能在服务端这里找到对应的记录，如果可以，说明用户已经通过了身份验证，就把用户请求的数据返回给客户端。

上面说的就是 Session，我们需要在服务端存储为登录的用户生成的 Session ，这些 Session 可能会存储在内存，磁盘，或者数据库里。我们可能需要在服务端定期的去清理过期的 Session 。

基于 Token 的身份验证

使用基于 Token 的身份验证方法，在服务端不需要存储用户的登录记录。大概的流程是这样的：

客户端使用用户名跟密码请求登录

服务端收到请求，去验证用户名与密码

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据。

# 操作系统

## 进程与线程的区别？

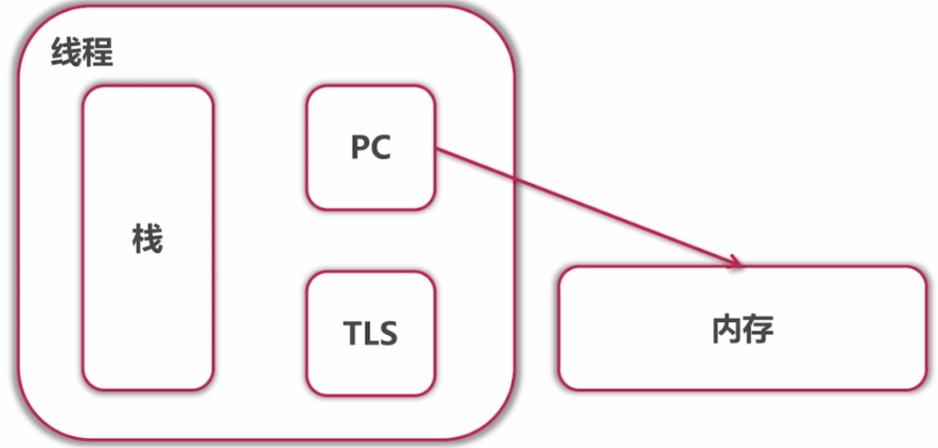
* 什么是进程

1. 进程是程序的一次执行过程，是一个动态概念，是**操作系统资源分配和管理的基本单位**；
2. 每个进程之间的内存是相互独立的，所以一个进程无法直接访问另一个进程的变量和数据结构，如果希望一个进程去访问另一个进程的资源，需要使用进程间的通信，比如：管道、消息队列等。比较常见的就是通过TCP/IP端口来实现。
3. 文件/网络句柄都是进程共享的，即不同的进程都能访问同一个文件或者网络端口。
4. 进程有运行、阻塞、就绪三个基本状态；并且进程调度算法有：先来先服务调度算法、短作业优先调度算法、非抢占式优先级调度算法、抢占式优先级调度算法、高响应比优先调度算法、时间片轮转法调度算法；

进 程 
线 程 
内 存 
文 件 / 网 络 句 柄 

* 什么是线程

1. 线程是进程里面的一部分，一个操作系统可能同时有多个进程在运行，每个进程里可能有多个线程在运行。
2. 线程是**CPU调度和分派的基本单位**，一个进程中可以有多个线程，这么多线程共享该进程的资源。



* 进程与线程的区别

1. 根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位；
2. 开销方面：每个进程有独立的代码和数据空间，进程之间的切换消耗的资源大；线程可以看做是轻量级的进程，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小；
3. 共享资源：同一进程内的所有线程共享本进程的资源，但是进程之间的资源是独立的；
4. 一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程崩溃，所以多进程比多线程健壮；
5. 多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

## 什么情况下会发生死锁？解决策略？

所谓死锁，是指多个进程循环等待它方占有的资源而无限期地僵持下去的局面。很显然，如果没有外力的作用，那麽死锁涉及到的各个进程都将永远处于封锁状态。当两个或两个以上的进程同时对多个互斥资源提出使用要求时，有可能导致死锁。

1. 互斥条件。即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。这种独占资源如CD-ROM驱动器，打印机等等，必须在占有该资源的进程主动释放它之后，其它进程才能占有该资源。这是由资源本身的属性所决定的。如独木桥就是一种独占资源，两方的人不能同时过桥。
2. 不可抢占条件。进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。如过独木桥的人不能强迫对方后退，也不能非法地将对方推下桥，必须是桥上的人自己过桥后空出桥面（即主动释放占有资源），对方的人才能过桥。
3. 占有且申请条件。进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。还以过独木桥为例，甲乙两人在桥上相遇。甲走过一段桥面（即占有了一些资源），还需要走其余的桥面（申请新的资源），但那部分桥面被乙占有（乙走过一段桥面）。甲过不去，前进不能，又不后退；乙也处于同样的状况。
4. 循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，......，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。就像前面的过独木桥问题，甲等待乙占有的桥面，而乙又等待甲占有的桥面，从而彼此循环等待。

死锁的预防是保证系统不进入死锁状态的一种策略。它的基本思想是要求进程申请资源时遵循某种协议，从而打破产生死锁的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

1. 打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。
2. 打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。
3. 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。
4. 打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁

解决策略：

鸵鸟策略：

假设的前提是，这样的问题出现的概率很低。比如，在操作系统中，为应对[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)问题，可以采用这样的一种办法。当系统发生[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)时不会对用户造成多大影响，或系统很少发生[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)的场合采用允许死锁发生的鸵鸟算法，这样一来可能开销比不允许发生死锁及检测和解除死锁的小。如果[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)很长时间才发生一次，而系统每周都会因硬件故障、[编译器](https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)错误或操作系统错误而崩溃一次，那么大多数工程师不会以性能损失或者易用性损失的代价来设计较为复杂的死锁解决策略，来消除死锁。鸵鸟策略的实质：出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价，还不如不做处理，OS中这种置之不理的策略称之为鸵鸟策略（也叫鸵鸟算法）。

银行家算法：

在避免死锁的方法中，所施加的限制条件较弱，有可能获得令人满意的系统性能。在该方法中把系统的状态分为安全状态和不安全状态，只要能使系统始终都处于安全状态，便可以避免发生死锁。

银行家算法的基本思想是：分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免死锁的算法。

## Linux进程的状态

（1）R（task\_running），可执行状态

只有在该状态的进程才可能在CPU上运行。同一时刻有可能有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。

（2）S (task\_interruptible)，可中断的睡眠状态

处于这个状态的进程因为等待某某事件的发生（比如等待socket连接、等待信号量），而被挂起。这些进程的task\_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。

（3）D(task\_uninterruptible)，不可中断的睡眠状态

在这个状态下的进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号（不然的话kill -9就不能杀死进程）。而task\_uninterruptible状态存在的意义就在于：内核的某些处理流程是不能被打断的。

（4）T(task\_stopped or task\_traced)，暂停状态或跟踪状态

表示进程暂停下来了；向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK\_STOPPED状态，向进程发送一个SIGCONT信号，可以让其从TASK\_STOPPED状态恢复到TASK\_RUNNING状态。

（5）Z(TASK\_DEAD - EXIT\_ZOMBIE)，退出状态，进程成为僵尸进程。

进程在退出的过程中，处于task\_read状态，此时进程占有的所有资源将被回收，除了task\_struct结构以及少数资源以外。于是进程就只剩下task\_struct这么个空壳，故称为僵尸。之所以保留task\_struct，是因为task\_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。

（6）X (TASK\_DEAD - EXIT\_DEAD)，退出状态，进程即将被销毁。

## 父进程创建子进程，fork详解

1. 函数原型：pid\_t fork(void); //void代表没有任何形式参数
2. 除了0号进程（系统创建的）之外，linux系统中都是由其他进程创建的。创建新进程的进程，即调用fork函数的进程为父进程，新建的进程为子进程。
3. fork函数不需要任何参数，对于返回值有三种情况：
4. 对于父进程，fork函数返回新建子进程的pid；
5. 对于子进程，fork函数返回 0；
6. 如果出错， fork 函数返回 -1。

|  |
| --- |
| int pid=fork();  if(pid < 0){  //失败，一般是该用户的进程数达到限制或者内存被用光了  ........  } else if(pid == 0){  //子进程执行的代码  ......  } else{  //父进程执行的代码  .........  } |

1. vfork 也可以创建新线程，但是有一点点不一样。fork 后的进程完全复制了父进程的资源，所以它的资源是独立的，因此开销比较大。fork出来的子进程一般会调用 exec ，然后自己就结束了，所以比较浪费。而 vfork 是共享父进程的资源，它改了什么，父进程会跟着受影响，但它减少了开销,因为它不需要将父进程的资源重新拷贝一份。并且，**vfork 后子进程先运行，父进程阻塞，直到子进程结束**.

## 父进程和子进程，死亡？通信？

（1）父进程退出后，子进程会发生什么？

在linux中，所有进程都有一个共同的父进程systemd，如果父进程退出了，子进程还没运行结束，子进程会被systemd（老版本的linux是initd）收养；

也就是说假如父进程创建了一堆子进程，当遇到错误或者操作失误的时候把父进程关闭了，但是子进程还在跑，不得不一个一个地杀死子进程，或者使用ps,grep,awk,kill来配合批量杀死。也可以使用Linux下libc的prctl设置PR\_SET\_PDEATHSIG属性，似乎可以让子进程在父进程自动结束后接收到信号。

（2）父进程与子进程之的通信方式？

1. 在Linux系统中实现父子进程的通信可以采用pipe()和fork()函数进行实现；
2. 对于父子进程，在程序运行时首先进入的是父进程，其次是子进程，在此我个人认为，在创建父子进程的时候程序是先运行创建的程序，其次在复制父进程创建子进程。fork()函数主要是以父进程为蓝本复制一个进程，其ID号和父进程的ID号不同。对于结果fork出来的子进程的父进程ID号是执行fork()函数的进程的ID号。
3. 管道：是指用于连接一个读进程和一个写进程，以实现它们之间通信的共享文件，又称pipe文件。
4. 写进程在管道的尾端写入数据，读进程在管道的首端读出数据。

## 如何让进程(或正在运行的程序)到后台运行?

在Linux中，如果要让进程在后台运行，一般情况下，我们在命令后面加上&即可，实际上，这样是将命令放入到一个作业队列中了；

对于已经在前台执行的命令，也可以重新放到后台执行，首先按ctrl+z暂停已经运行的进程，然后使用bg命令将停止的作业放到后台运行；

但是如上方到后台执行的进程，其父进程还是当前终端shell的进程，而一旦父进程退出，则会发送hangup信号给所有子进程，子进程收到hangup以后也会退出。如果我们要在退出shell的时候继续运行进程，则需要使用nohup忽略hangup信号，或者使用setsid将父进程设为init进程(进程号为1)；

对于已经在后台运行的进程，可以将其放在subshell中执行，从而变成后台进程；方法很简单，将命令用括号() 括起来即可；

## 进程间通信的方式有几种，区别是什么？

1) 管道（管道，通常指无名管道）

1. 半双工的，具有固定的读端和写端；
2. 只能用于具有亲属关系（父子进程、兄弟进程）的进程之间的通信；
3. 可以看成是一种特殊的文件，没有名字，对于它的读写也可以使用普通的read、write函数。但是它不是普通的文件，并不属于其他任何文件系统，只能用于内存中。
4. 无名管道创建的方式：Int pipe(int fd[2]);当一个管道建立时，会创建两个文件文件描述符，fd 数组会被填充两个文件描述符（fd[0]代表读文件描述符，只能用来读，不能用来写；fd[1]代表写文件描述符，只能用来写，不能用来读），要关闭管道只需将这两个文件描述符关闭即可。

2) FiFO（有名管道）

1. FIFO可以在两个独立的进程之间交换数据，与无名管道不同，会在本地文件系统创建一个有名字的文件，用于通信。
2. FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中；
3. Int mkfifo(const char\* pathname,mode\_t mode);

> path 是管道路径名

> mode 是管道的权限

3) 消息队列（消息队列适合短小的数据通信）

1. 消息队列，是消息的连接表，存放在内核中。一个消息队列由一个标识符来标识；
2. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级；
3. 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除；
4. 消息队列可以实现消息的随机查询

4) 信号量

1. 信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据；
2. 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存；
3. 信号量基于操作系统的PV操作，程序对信号量的操作都是原子操作；

> P 的具体涵义是试探的意思，假设信号量取值为S，P的意图就是要减少S的值，直白点就是它想拿到信号量的许可去访问临界资源。比如餐厅还有 3 个位置，某个人进去之后，可用的位置就变成了 2 个。V 的作用相反，它的目的是尝试增加 S 的值。比如餐厅的顾客吃完饭后，可用的座位又多了一个。需要注意的是，一般我们**开发信号量的值取 1、 0、 -1** 这 3 个，代表处理两个进程同步。

5) 共享内存

1. 共享内存，指两个或多个进程共享一个给定的存储区；
2. 共享内存是最快的一种进程通信方式，因为进程是直接对内存进行存取；
3. 因为多个进程可以同时操作，所以需要进行同步；
4. 信号量+共享内存通常结合在一起使用。
5. 共享内存的用处：比较适合在进程间处理容量较大的数据。在 Linux 终端运行 ipcs -m命令，你就可以查看目前系统现存的共享内存。

## 共享内存

共享内存是通过把同一块内存分别映射到不同的进程空间中实现进程间通信。而共享内存本身不带任何互斥与同步机制，但当多个进程同时对同一内存进行读写操作时会破坏该内存的内容，所以，在实际中，同步与互斥机制需要用户来完成。

效率：

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据： 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。

实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

## 同步、异步、阻塞、非阻塞？

1) 同步

就是在发出一个功能调用时，在没有得到结果之前，该调用就不返回。也就是必须一件一件事做,等前一件做完了才能做下一件事。就是我调用一个功能，该功能没有结束前，我死等结果。

2) 异步

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者。就是我调用一个功能，不需要知道该功能结果，该功能有结果后通知我（回调通知）

3) 阻塞

阻塞调用是指调用结果返回之前，当前线程会被挂起（线程进入非可执行状态，在这个状态下，cpu不会给线程分配时间片，即线程暂停运行）。函数只有在得到结果之后才会返回。对于同步调用来说，很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。 就是调用我（函数），我（函数）没有接收完数据或者没有得到结果之前，我不会返回。

4) 非阻塞

指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。就是调用我（函数），我（函数）立即返回，通过select通知调用者。

## 常见的5种IO模型？

1) 阻塞I/O

应用程序调用一个IO函数，导致应用程序阻塞，等待数据准备好。如果数据没有准备好，一直等待….数据准备好了，从内核拷贝到用户空间，IO函数返回成功指示。

2) 非阻塞I/O

我们把一个SOCKET接口设置为非阻塞就是告诉内核，当所请求的I/O操作无法完成时，不要将进程睡眠，而是返回一个错误。这样我们的I/O操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。在这个不断测试的过程中，会大量的占用CPU的时间。

3) I/O复用

I/O复用模型会用到select、poll、epoll函数，这几个函数也会使进程阻塞，但是和阻塞I/O所不同的的，这三个函数可以同时阻塞多个I/O操作。而且可以同时对多个读操作，多个写操作的I/O函数进行检测，直到有数据可读或可写时，才真正调用I/O操作函数。

4) 信号驱动I/O

首先我们允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

5) 异步I/O

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作。

## BIO NIO AIO的区别？

（1）同步阻塞I/O(BIO)：

在JDK1.4出来之前，我们建立网络连接的时候采用BIO模式，需要先在服务端启动一个ServerSocket，然后在客户端启动Socket来对服务端进行通信，默认情况下服务端需要对每个请求建立一堆线程等待请求，而客户端发送请求后，先咨询服务端是否有线程相应，如果没有则会一直等待或者遭到拒绝请求，如果有的话，客户端会线程会等待请求结束后才继续执行。这样的模式很明显的一个缺陷是：由于客户端连接数与服务器线程数成正比关系，可能造成不必要的线程开销，严重的还将导致服务器内存溢出。当然，这种情况可以通过线程池机制改善，但并不能从本质上消除这个弊端。

* 应用场景：BIO方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中。

（2）同步非阻塞I/O(NIO)：

NIO本身是基于事件驱动思想来完成的，其主要想解决的是BIO的大并发问题： 在使用同步I/O的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须使用多线程来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。NIO的服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有IO请求时才启动一个线程进行处理。

* 应用场景：NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。

（3）异步非阻塞I/O(AIO)：

与NIO不同，当进行读写操作时，可以直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。AIO的服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

* 应用场景：AIO方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。更好的建议是采用成熟的网络通信框架Netty。

## select，poll，epoll的区别？

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

**epoll可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）**

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

在Linux下实现I/O复用模型的方式：调用select/poll/epoll其中一个函数，传入多个文件描述符，如果有一个文件描述符就绪，则返回，否则阻塞直到超时。

Select、poll、epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但**select、poll、epoll本质上都是同步I/O**，因为它们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的；

|  |  |  |  |
| --- | --- | --- | --- |
|  | select | poll | epoll |
| 操作方式 | 轮询方式 | | 回调方式 |
| 底层实现 | 数组 | 链表来保存fd | 哈希表 |
| IO效率 | 每次调用都进行线性遍历，时间复杂度为O(n) | | 事件通知方式，每当文件描述符fd就绪，系统注册的回调函数就会被调用，将就绪的fd放入rdlist里面，时间复杂度为O(1) |
| 最大连接数 | 1024(x86) | 无上限 | 无上限 |
| fd拷贝 | 每次调用select（或poll），都需要把fd集合从用户态拷贝到内核态 | | 调用epoll\_ctl时拷贝进内核并保存，之后就不需要再拷贝；epoll通过内核和用户空间共享一块内存来实现的。 |
| 工作方式 | 只能工作在水平触发模式（LT） | | 支持边缘触发模式和水平触发模式（ET、LT） |

* ET模式：如果一个描述符上有数据到达，然后读取这个描述符上的数据，如果没有将数据读取完，当下次epoll\_wait返回的时候这个描述符中的数据就再也读取不到了；
* LT模式：如果对一个描述符的数据没有读取完成，那么下次当epoll\_wait返回的时候会继续触发，也就可以继续获取到这个描述符，从而能够接着读；

总结：

* + select和poll都需要轮询每个文件描述符，epoll基于事件驱动，不用轮询；
  + select和poll每次都需要拷贝文件描述符，epoll不用；
  + select最大连接数受限，epoll和poll最大连接数不受限；

Tips：epoll在内核中的实现，用红黑树管理事件块，这样重复添加的事件就可以通过红黑树而高效的识别出来；

## Linux是如何避免内存碎片的

1. 在固定式分区分配中，为将一个用户作业装入内存，内存分配程序从系统分区表中找出一个能满足作业要求的空闲分区分配给作业，由于一个作业的大小并不一定与分区大小相等，因此，分区中有一部分存储空间浪费掉了，由此可知，固定式分区分配中存在内碎片。
2. 在可变式分区分配中，为把一个作业装入内存，应按照一定的分配算法从系统中找出一个能满足作业需求的空闲分区分配给作业，如果这个空闲分区的容量比作业申请的空间容量要大，则将该分区一分为二，一部分分配给作业，剩下的部分仍然留作系统的空闲分区。由此可知，可变式分区分配中存在外碎片。
3. 伙伴系统：伙伴系统是一种经典的内存管理方法。Linux伙伴系统的引入为内核提供了一种用于分配一组连续的页（Linux中的内存管理的“页”大小为4KB）而建立的一种高效的分配策略，并有效的解决了外碎片问题。
4. 据可移动性组织页避免内存碎片

## 僵尸进程

1 什么是僵尸进程

* 当子进程比父进程先结束， 而父进程又没有回收子进程， 释放子进程占用的资源， 此时子进程将成为一个僵尸进程。

2 怎样来清除僵尸进程

* + 改写父进程，在子进程死后要为它收尸。具体做法是接管SIGCHLD信号。子进程死后，会发送SIGCHLD信号给父进程，父进程收到此信号后，执行waitpid()函数为子进程收尸。这是基于这样的原理：就算父进程没有调用wait，内核也会向它发送 SIGCHLD 消息，尽管对的默认处理是忽略， 如果想响应这个消息， 可以设置一个处理函数。
  + 2.把父进程杀掉。父进程死后，僵尸进程成为”孤儿进程”， 过继给1号进程 init，init始终会负责清理僵尸进程． 它产生的所有僵尸进程也跟着消失。

## 线程同步和互斥

（1）线程同步

* 同步就是协同步调，按预定的先后次序进行运行。线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步），也可以说是在线程之间通过同步建立起执行顺序的关系；

（2）线程互斥

* 指对于共享的进程系统资源，在每一个线程访问时的排它性。当有多个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步；

（3）线程同步的方式

* 1、临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。它并不是核心对象，不是属于操作系统维护的，而是属于进程维护的。
* 2、互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。
* 3、信号量：信号量也是内核对象。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目；一般是将当前可用资源计数设置为最 大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减1 ，只要当前可用资源计数是大于0 的，就可以发出信号量信号。但是当前可用计数减小 到0 时则说明当前占用资源的线程数已经达到了所允许的最大数目，不能在允许其他线程的进入，此时的信号量信号将无法发出。
* 4、事件对象： 通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。事件是内核对象，事件分为手动置位事件和自动置位事件。事件Event内部它包含一个使用计数，一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。事件可以解决线程间同步问题，因此也能解决互斥问题。

## 使用内存映射处理大文件

内存文件映射提供了一个统一的内存管理特征，使应用程序可以通过内存指针对磁盘上的文件进行访问，其过程就如同对加载了文件的内存的访问。通过文件映射这种使磁盘文件的全部或部分内容与进程虚拟地址空间的某个区域建立映射关联的能力，可以直接对被映射的文件进行访问，而不必执行文件I/O操作也无需对文件内容进行缓冲处理。

内存映射文件的一般编程流程框图：



在某些特殊行业，经常要面对十几GB乃至几十GB容量的巨型文件，而一个32位进程所拥有的虚拟地址空间只有232 = 4GB，显然不能一次将文件映像全部映射进来。对于这种情况只能依次将大文件的各个部分映射到进程中的一个较小的地址空间。这需要对上面的一般流程进行适当的更改：

1）映射文件开头的映像。

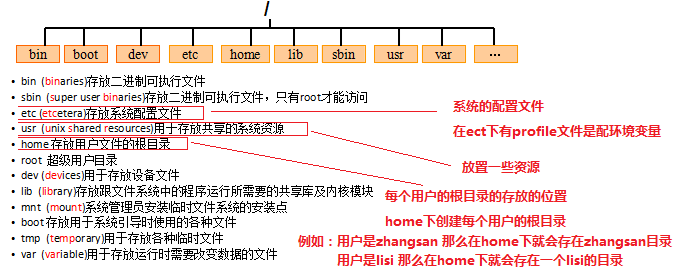
2）对该映像进行访问。

3）取消此映像

4）映射一个从文件中的一个更深的位移开始的新映像。

5）重复步骤2，直到访问完全部的文件数据。

## Linux目录结构



## linux 内核链表和普通的链表有什么区别？

一般的链表结构体中都会包含两部分：一个是指针（前向指针和后向指针）部分，还有就是有效数据部分。将我们链表需要存放的数据放在结构体中的有效数据；其中，通过两个节点分别指向上一个节点和下一个节点。这是一般的链表的使用方法。

* 但是内核链表的结构体中只有两个指针，并没有有效数据区，所以内核链表的用法和我们的普通链表的用法是不一样的；
* 我们使用内核链表的方法就是：自己建立一个结构体去包含这个内核链表，将整个结构体作为一个链表的节点，然后使用内置的链表的前向指针指向下一个结构体中内置的链表，使用内置的链表的后向指针指向上一个结构体中内置的链表。

## Linux常用命令

|  |  |
| --- | --- |
| ls -al | 显示当前目录下所有文件详细信息，包括隐藏文件 |
| cd [~] [-] | 切换目录 [~目前用户身份的主目录] [-前一个工作目录] |
| chmod 770 文件名 | 修改文件的权限 |
| cp A B | 复制文件A为B |
| rm A  rm -r A | 删除A  删除非空目录或文件A |
| mkdir/rmdir | 新建目录、删除一个空目录 |
| man 命令 | 查询命令详细解释 |
| pwd | 显示当前目录 |
| mv 文件名 目标位置  mv A B | 移动文件  重命名A为B（目录或文件都可以） |
| vim 文件名 | 进入后按i可以输入；退出按ESC然后:wq；不保存退出：按ESC 然后:q! |
| reboot | 重启服务器 |
| shift+PgUp\PgDn | 命令行界面上下滚动 |
| tcpdump [host 192.168.1.120] | 截获所有192.168.1.120 的主机收到的和发出的所有的数据包 |
| kill -9 PID | 关闭服务，比如某程序PID=6817 kill -9 6817 就关闭了这个服务 |
| top | 查看当前系统负载情况 |
| netstat -ntlp  netstat -ntulp | 查看当前所有TCP端口  查看当前所有TCP和UDP端口 |
| netstat -nap| grep 5672 | 查看端口号5672是否被监听 |
| lsof –i:端口号 | 查看端口被哪个进程占用 |
| free -m | 查看内存使用情况 |
| ps -A  ps –ef | grep java | 显示所有运行中的进程  搜索java进程 |
| God status  god stop 服务名 | 查看当前部署的服务  停止服务 |

* Tab：命令与文件名补全。
* Ctrl+C：中断正在运行的程序；
* Ctrl+D：结束键盘输入（End Of File，EOF）
* Linux如何对文件内容中的关键字进行查找?
  + 如果是用vi打开文件后，在命令行下输入“/关键字”
  + 如果是在没有打开文件的前提就用"cat 文件名 | grep "关键字""
* nohup命令：如果你正在运行一个进程，而且你觉得在退出帐户时该进程还不会结束，那么可以使用nohup命令。该命令可以在你退出帐户/关闭终端之后继续运行相应的进程。

# java基础

## 基础知识

### 面向对象的三大特性？

1 封装

* 定义：把数据和操作数据的方法绑定起来，对数据的访问只通过已定义的接口，面向对象的本质就是把现实世界的事物描绘成一系列自治、封闭的对象；我们在类中编写的方法就是对实现细节的一种封装，编写的类就是对**属性和行为的封装**。在java中能使用private、protected、public三种修饰符或不用（即默认defalut）对外部对象访问该对象的属性和行为进行限制。

2 继承

* 定义：继承是**从已有的类得到继承信息创建新类的过程**；提供继承信息的类称为父类，得到继承信息的类称为子类；
* 优缺点：提高代码复用性；提高代码的维护性；但是类之间的耦合性增强了，我们开发的时候的原则是：高内聚、低耦合（内聚：是指自己完成事情的能力，自己能完成就不要麻烦别人；耦合：类与类的关系）；

3 多态

* 定义：**指父对象中的同一个行为能在其多个子对象中有不同的表现**。简单的说，就是一句话：**允许父类引用指向子类对象**。多态则是体现在overriding（重写）上，而overload（重载）则不属于面向对象中多态的范畴，因为重载在非面向对象中也存在。重写是面向对象中的多态，因为重写是与继承紧密联系，是面向对象所特有的。实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。
  + 重写：子类重新定义父类中的方法；
  + 重载：一个类中允许有多个同名的函数，而这些函数的参数列表不同（参数的个数不同、参数的类型不同、参数的顺序不同）；
* 优点：消除类型之间的耦合关系；可以实现接口的重用；

### C语言实现面向对象

面向对象编程（OOP）并不是一种特定的语言或者工具，它只是一种设计方法、设计思想。它表现出来的三个最基本的特性就是封装、继承与多态。C语言也同样可以实现OOP，里面具体用到的有C语言中的宏，结构体，函数指针, 聚合组合等。

C语言中的定义：

* 结构体：在C语言中，常把一个对象用结构体进行封装，这样便于对对象进行操作；结构体可以嵌套。因而可以把一个结构体当成另一个结构体的成员；
* 函数指针：函数指针是指针的一种，它指向函数的首地址（函数的函数名即为函数的首地址），可以通过函数指针来调用函数。
* extern与static：extern和static是C语言中的两个修饰符，extern可用于修饰函数或者变量，表示该变量或者函数在其他文件中进行了定义；static也可用于修饰函数或者变量，表示该函数或者变量只能在该文件中使用。可利用它们对数据或者函数进行隐藏或者限制访问权限。
* 封装：在C语言中，可以用结构+函数指针来模拟类的实现，而用这种结构定义的变量就是对象。
* 继承：在C语言中，可以利用“结构在内存中的布局与结构的声明具有一致的顺序”这一事实实现继承。比如我们要设计一个作图工具，其中可能涉及到的对象有Point(点)，Circle(圆)，由于圆是由点组成的，所有可以看成Circle继承自Point。另外，Point和Circle都需要空间申请，空间释放等操作，所有他们有共同的基类Base。
* 多态：可以是用C语言中的万能指针void\* 实现多态。

### Java中泛型的优点？

1.安全：在编译的时候检查类安全  
2.省心：所有强制转换都是自动和隐式的，提高代码的重用率

* 泛型：是对 Java 语言的类型系统的一种扩展，以支持创建可以按类型进行参数化的类。可以把类型参数看作是使用参数化类型时指定的类型的一个占位符，就像方法的形式参数是运行时传递的值的占位符一样。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。
* 使用泛型的优点：
  + 类型安全。可以在编译的时候检查类型安全。
  + 消除强制类型转换。消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。
  + 潜在的性能收益。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，程序员会指定这些强制类型转换）插入生成的字节码中。由于泛型的实现方式，支持泛型（几乎）不需要 JVM 或类文件更改。所有工作都在编译器中完成，编译器生成类似于没有泛型（和强制类型转换）时所写的代码，只是更能确保类型安全而已。
* 泛型的使用规则：
  + 泛型的类型参数只能是类类型，不能使简单的类型；
  + 泛型的类型参数可以有多个；
  + 泛型的参数类型可以使用extends语句，比如<T extends superclass>；
  + 泛型的参数类型还可以是通配符型，比如Class<?> classType = Class.forName(java.lang.String)；

### 抽象类和接口的区别？



### Object类中的方法有哪些？

|  |  |
| --- | --- |
| clone() | 创建并返回此对象的一个副本 |
| hashCode() | 返回该对象的hash值 |
| equals() | 判断一个对象是否与此对象相等 |
| toString() | 返回该对象的字符串表示 |
| finalize() | 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法 |
| getClass() | 是一个native方法，返回的是此Object对象的类对象/运行时类对象Class<?> |
| notify() | 唤醒在此对象监视器上等待的单个线程/所有线程。 |
| notifyAll() | 唤醒在此对象监视器上等待的所有线程。 |
| wait() | 暂停当前线程，直到其他线程调用这个对象的notify()方法 |

### Collection与Collections的区别？

（1）java.util.Collection 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Colleciton是集合接口，继承它的接口有List，Set，注意map集合虽然也属于集合体系，但是map并不继承collection，map和collection是平级关系。

（2）java.util.Collections 是一个包装类。它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，服务于Java的Collection框架 。Collections是针对集合类的一个帮助类，提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作 。比如 排序Collections.sort()，混排Collections.shuffling()，反转Collections.reverse()，替换所有的元素Collections.fill()，拷贝Collections.copy()。

### Collections.sort()或Arrays.sort()底层实现？

（1）不论是Collections.sort方法（是对list集合进行排序）或者是Arrays.sort方法（是对数组进行排序），底层实现都是TimSort实现的，这是jdk1.7新增的，以前是归并排序。

* Colletions.sort()实际会将list转为数组，然后调用Arrays.sort()，排完了再转回List。
* Arrays.sort()，对原始类型(int[],double[],char[],byte[])，JDK6里用的是快速排序，对于对象类型(Object[])，JDK6则使用归并排序。到了JDK7，快速排序升级为双基准快排(双基准快排vs三路快排)；归并排序升级为归并排序的改进版TimSort；

（2）Timsort是一种结合了归并排序和插入排序的混合算法，由Tim Peters在2002年提出，并且已经成为Python 2.3版本以后内置排序算法。针对现实中需要排序的数据分析看，大多数据通常是有部分已经排好序的数据块，在Timsort中称这些已经排好序的数据块为“run”（一个一个的“分区”）。在排序时，Timsort迭代数据元素，将其放到不同的 run 里，同时针对这些 run ，按规则进行合并至只剩一个，则这个仅剩的 run 即为排好序的结果。简单来说，这个算法可以概括为两步：

* 1）判断数组的大小，小于32使用二分插入排序：使用二分查找位置，进行插入排序。
* 2）数组大于32时，先算出一个合适的大小，在将输入按其升序和降序特点进行了分区，排序的输入的单位不是一个个单独的数字，而是一个个的run。将run入栈，当栈顶的run的长度满足：runLen[n-2] <= runLen[n-1] + runLen[n]或者 runLen[n-1] <= runLen[n]，则对两个短run归并为一个新run，则到只剩栈顶元素时排序也完成了。

（3）本质上 Timsort 是一个经过大量优化的归并排序，在最坏的情况下，Timsort 时间复杂度为 O(nlogn)。在最佳情况下，即输入已经排好序，它则以线性时间运行O(n)。

时间复杂度什么时候出现Logn————————当数据量减半时，它的复杂度就是就是logn了

### ArrayList的源码

* ArrayList是可以动态增长和缩减的索引序列，它是基于数组实现的List类。
* 该类封装了一个动态再分配的Object[]数组，每一个类对象都有一个capacity属性，表示它们所封装的Object[]数组的长度，当向ArrayList中添加元素时，该属性值会自动增加。如果想ArrayList中添加大量元素，可使用ensureCapacity方法一次性增加capacity，可以减少增加重分配的次数提高性能。
* ArrayList的用法和Vector向类似，但是Vector是一个较老的集合，具有很多缺点，不建议使用。另外，ArrayList和Vector的区别是：ArrayList是线程不安全的，当多条线程访问同一个ArrayList集合时，程序需要手动保证该集合的同步性，而Vector则是线程安全的。

ArrayList实现了哪些接口？

* List<E>接口：在查看了ArrayList的父类AbstractList也实现了List<E>接口，那为什么子类ArrayList还是去实现一遍呢？这其实是一个mistake，作者说他写这代码的时候觉得这个会有用处，但是其实并没什么用，但因为没什么影响，就一直留到了现在。
* RandomAccess接口：这个是一个标记性接口，通过查看api文档，它的作用就是用来快速随机存取，有关效率的问题，在实现了该接口的话，那么使用普通的for循环来遍历，性能更高，例如arrayList。而没有实现该接口的话，使用Iterator来迭代，这样性能更高，例如linkedList。所以这个标记性只是为了让我们知道我们用什么样的方式去获取数据性能更好。
* Cloneable接口：实现了该接口，就可以使用Object.Clone()方法了。
* Serializable接口：实现该序列化接口，表明该类可以被序列化，什么是序列化？简单的说，就是能够从类变成字节流传输，然后还能从字节流变成原来的类。

ArrayList的扩容：

* 默认初始化arrayList的容量是10。在扩容的时候，比如使用add()方法的时候，里面有一个size属性，表示当前集合拥有的元素的个数，方法中调用了ensureCapacityInternal来保证容量问题，传进去的参数是size+1，来保证新增元素后容量满足要求。判断新增元素后的大小minCapacity是否超过当前集合的容量elementData.length，如果超过，则调用grow方法进行扩容。
* 当然在扩容的时候，里面有一个modCount++的属性，这个参数主要是用在集合的Fail-Fast（快速失败机制）的判断中使用的。
* 扩容容量的计算:int newCapacity = oldCapacity + (oldCapacity >> 1),其中oldCapacity是原来的容量大小，即容量扩大为原来的1.5倍。在grow方法中，调用Arrays.copyof方法，即复制原数组内容到一个新容量的大数组里。这里Arrays.copyof方法实际是调用System.arraycopy方法。

### Vector的源码

* Vector是动态数组实现的List，跟ArrayList一样，Object[]类型的数组，它保存了添加到Vector中的元素。elementData是个动态数组，如果初始化Vector时，没指定动态数组的>大小，则使用默认大小10。随着Vector中元素的增加，Vector的容量也会动态增长,扩容的大小是原来的2倍；
* Vector是JDK1.0引入了，它的很多实现方法都加入了同步语句（synchronized），因此是线程安全的；
* Vector适用于快速访问和修改，不适用随机插入和删除；
* Vector元素允许为null；Vector现在已经基本不再使用，如果不需要线程安全的实现，推荐使用ArrayList代替Vector；

### ArrayList和LinkedList区别

1、数据结构不同

* ArrayList是Array(动态数组)的数据结构，LinkedList是Link(双向链表)的数据结构。

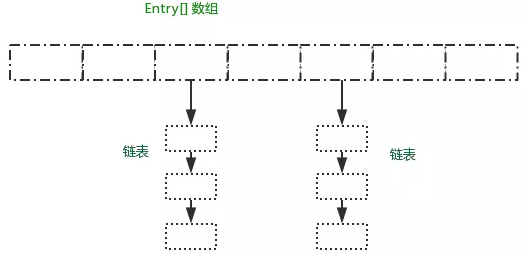
2、效率不同

* 当随机访问List（get和set操作）时，ArrayList比LinkedList的效率更高，因为LinkedList是线性的数据存储方式，所以需要移动指针从前往后依次查找。
* 当对数据进行增加和删除的操作(add和remove操作)时，LinkedList比ArrayList的效率更高，因为ArrayList是数组，所以在其中进行增删操作时，会对操作点之后所有数据的下标索引造成影响，需要进行数据移动。

### HashMap源码问题

#### （1）底层实现原理

HashMap采用Entry数组来存储key-value对，每一个键值对组成了一个Entry实体，Entry类实际上是一个单向的链表结构，它具有Next指针，可以连接下一个Entry实体。在JDK1.8中，链表长度大于8的时候，链表会转成红黑树，（当红黑树的节点数量等于6的时候，再次退化成链表的结构）。



* 数组是用来确定桶的位置，利用元素的key的hash值对数组长度取模得到。
* 链表是用来解决hash冲突问题，当出现hash值一样的情形，就在数组上的对应位置形成一条链表。（这里的hash值并不是指hashcode，而是将hashcode高低十六位异或过的）。
* 在进行增删查等操作时，首先要定位到元素的所在桶的位置，之后再从链表中定位该元素。

#### （2）解决hash冲突的方法有几种？

比较出名的有四种：

* 开放定址法：所谓的开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。<https://blog.csdn.net/mr_doublerun/article/details/49633167>。
* 链地址法：HashMap，HashSet其实都是采用的拉链法来解决哈希冲突的，就是在每个位桶实现的时候，采用链表的数据结构来去存取发生哈希冲突的输入域的关键字。拉链法处理冲突简单，且无堆积现象，平均查找长度较短；拉链法中各链表上的结点空间是动态申请的，它更适合于造表前无法确定表长的情况
* 再哈希法：哈希法又叫双哈希法，有多个不同的Hash函数，当发生冲突时，使用第二个，第三个，….，等哈希函数计算地址，直到无冲突。虽然不易发生聚集，但是增加了计算时间。

#### （3）用LinkedList代替数组结构可以么?

源码中是这样的：Entry[] table = new Entry[capacity];（Entry就是一个链表节点）。那用下面这样表示List<Entry> table = new LinkedList<Entry>();，是否可行?

答案是可以的，那为什么HashMap不采用不用LinkedList,而选用数组？因为用数组效率最高，在HashMap中，定位桶的位置是利用元素的key的哈希值对数组长度取模得到。显然数组的查找效率比LinkedList高。那ArrayList，底层也是数组，查找也快，为啥不用ArrayList？因为采用基本数组结构，扩容机制可以自己定义，HashMap中数组扩容刚好是2的次幂，在做取模运算的效率高。而ArrayList的扩容机制是1.5倍扩容。

#### （4）HashMap的扩容机制？

默认情况下，HashMap 初始容量是16，负载因子为0.75。

如果bucket满了(超过loadFactor\*current capacity)，就要resize。loadFactor为0.75，为了最大程度避免哈希冲突，current capacity为当前数组大小。

HashMap为了存取高效，要尽量较少碰撞，就是要尽量把数据分配均匀，每个链表长度大致相同，这个实现就在把数据存到哪个链表中的算法；这个算法实际就是取模，hash%length。但是，这种运算不如位移运算快。因此，源码中做了优化hash&(length-1)。也就是说hash%length==hash&(length-1)。

那为什么是2的n次方呢？因为2的n次方实际就是1后面n个0，2的n次方-1，实际就是n个1。例如长度为8时候，3&(8-1)=3 2&(8-1)=2 ，不同位置上，不碰撞。而长度为5的时候，3&(5-1)=0 2&(5-1)=0，都在0上，出现碰撞了。所以，保证容积是2的n次方，是为了保证在做(length-1)的时候，每一位都能&1，也就是和1111……1111111进行与运算。

#### （5）负载因子的作用？

对于 HashMap 来说，负载因子是一个很重要的参数，该参数反应了 HashMap 桶数组的使用情况。（假设键值对节点均匀分布在桶数组中）。通过调节负载因子，可使 HashMap 时间和空间复杂度上有不同的表现：

* 当调低负载因子时，HashMap 所能容纳的键值对数量变少。扩容时，重新将键值对存储新的桶数组里，键的键之间产生的碰撞会下降，链表长度变短。此时，HashMap 的增删改查等操作的效率将会变高，这里是典型的拿空间换时间。
* 如果增大负载因子（负载因子可以大于1），HashMap 所能容纳的键值对数量变多，空间利用率高，但碰撞率也高。这意味着链表长度变长，效率也随之降低，这种情况是拿时间换空间。

#### （6）HashMap的get/put的过程?

put

* 对key的hashCode()做hash运算，计算index；
* 如果没碰撞直接放到bucket里；
* 如果碰撞了，以链表的形式存在buckets后；
* 如果碰撞导致链表过长(大于等于TREEIFY\_THRESHOLD)，就把链表转换成红黑树(JDK1.8中的改动)；
* 如果节点已经存在就替换old value(保证key的唯一性)。
* 如果bucket满了(超过load factor\*current capacity)，就要resize。

get

* 对key的hashCode()做hash运算，计算index；
* 如果在bucket里的第一个节点里直接命中，则直接返回；
* 如果有冲突，则通过key.equals(k)去查找对应的Entry;
  + 若为树，则在树中通过key.equals(k)查找，O(logn)；
  + 若为链表，则在链表中通过key.equals(k)查找，O(n)。

#### （7）为什么hashmap的在链表元素数量超过8时改为红黑树?

JDK1.8中对HashMap做了下面三点改进：

* 由数组+链表的结构改为数组+链表+红黑树。
* 优化了高位运算的hash算法：h^(h>>>16)
* 扩容后，元素要么是在原位置，要么是在原位置再移动2次幂的位置，且链表顺序不变。（不会出现死循环问题）

为什么在解决hash冲突的时候，不直接用红黑树?而选择先用链表，再转红黑树？

因为红黑树需要进行左旋，右旋，变色这些操作来保持平衡，而单链表不需要。当元素小于8个当时候，此时做查询操作，链表结构已经能保证查询性能。当元素大于8个的时候，此时需要红黑树来加快查询速度，但是新增节点的效率变慢了。因此，如果一开始就用红黑树结构，元素太少，新增效率又比较慢，无疑这是浪费性能的。

当链表转为红黑树后，什么时候退化为链表？

为6的时候退转为链表。中间有个差值7可以防止链表和树之间频繁的转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

若桶中链表元素个数大于等于8时，链表转换成树结构；若桶中链表元素个数小于等于6时，树结构还原成链表。因为红黑树的平均查找长度是log(n)，长度为8的时候，平均查找长度为3，如果继续使用链表，平均查找长度为8/2=4，这才有转换为树的必要。链表长度如果是小于等于6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

#### （8）HashMap的并发问题?

* + 多线程扩容，引起的死循环问题
  + 多线程put的时候可能导致元素丢失
  + put非null元素后get出来的却是null

在jdk1.8中，死循环问题已经解决。另外两个问题可以采用，比如ConcurrentHashmap，Hashtable等线程安全等集合类。

#### （9）一般用什么作为HashMap的key？

HashMap的key可以为null，key为null的时候，hash算法最后的值以0来计算，放在数组的第一个位置。

|  |
| --- |
| Static final int hash(Object key){  Int h;  Return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  } |

一般用Integer、String这种不可变类当HashMap当key，而且String最为常用。

* + - 因为字符串是不可变的，所以在它创建的时候hashcode就被缓存了，不需要重新计算。这就使得字符串很适合作为Map中的键，字符串的处理速度要快过其它的键对象。这就是HashMap中的键往往都使用字符串。
    - 因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的,这些类已经很规范的覆写了hashCode()以及equals()方法。

如果使用可变类当HashMap的key，hashcode可能发生改变，导致put进去的值，无法get出来

### HashMap和HashTab区别

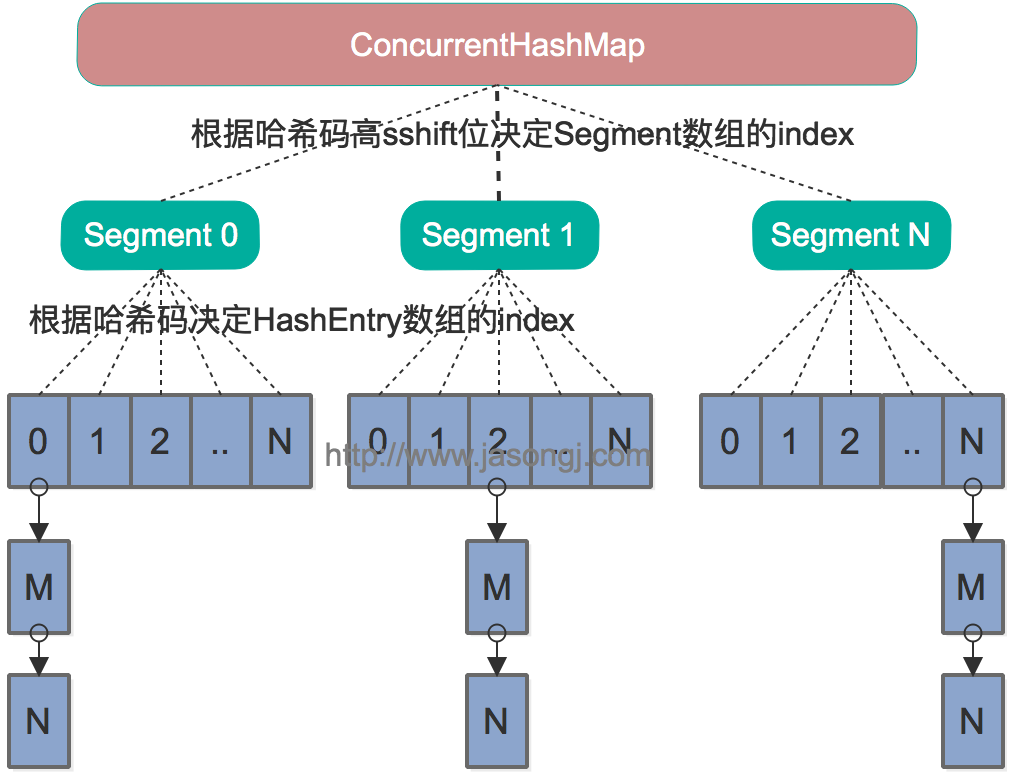
* HashMap 是线程不安全的，因为多线程环境下，使用Hashmap进行put操作会引起死循环，HashMap 是一个接口，是 Map 的一个子接口，不允许键值重复，允许key或value为null。
* HashTable 是线程安全的一个集合，不允许key或value为null；
* HashMap 的效率要较 HashTable 的效率高一些
* HashTable使用synchronized来保证线程安全，但在多线程下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如果要在多线程中使用线程安全的集合，可以使用ConcurrentHashMap这个类。

### ConcurrentHashMap

【JDK1.7】HashTable容器在并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁都可以将容器中的一部分数据锁住，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的**锁分段技术**：

* 首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
* 有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“**按顺序**”是很重要的，否则极有可能出现死锁。
* 在ConcurrentHashMap内部，**段数组是final的**，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

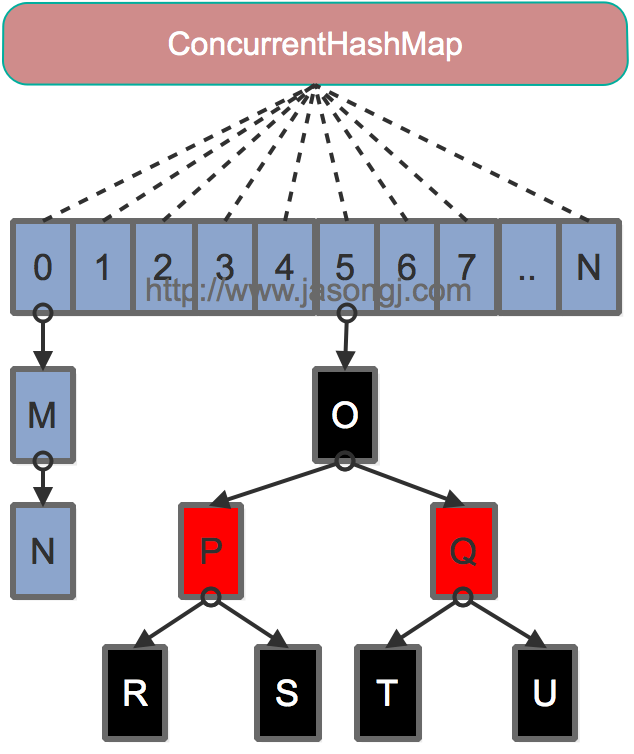
Java1.7中的ConcurrentHashMap的底层数据结构仍然是数组和链表。与HashMap不同的是，ConcurrentHashMap最外层不是一个大的数组，而是一个Segment[]的数组，每个Segment包含一个HashEntry[]数组（这个HashEntry数组与HashMap中的链表结构相似）。



* 在HashEntry类中，其中的变量value、以及链表都是有volatile修饰的，保证了获取值的可见性；
* 虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。Put操作的时候，首先会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁。
* get操作的时候不需要加锁，非常的高效；由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。

【JDK1.8】jdk1.7中已经解决了并发问题，并且能支持N个Segment次数的并发数量。但是依然存在查询遍历链表效率太低的问题。Java 8摒弃了Segment分段锁的方案，而采用了 CAS + synchronized 来保证并发安全性。而是直接使用一个大的数组。同时为了提高哈希碰撞下的寻址性能，Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(long(N))）。

Java 8的ConcurrentHashMap同样是通过Key的哈希值与数组长度取模确定该Key在数组中的索引。Concurrenthashmap扩容的时候首先会创建容量是原容量两倍的数组，然后将原数组里的元素进行再hash后插入到新的数组里。为了高效ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。



* + 对于put操作，如果Key对应的数组元素为null，则通过CAS操作将其设置为当前值。
  + 如果Key对应的数组元素（也即链表表头或者树的根元素）不为null，则对该元素使用synchronized关键字申请锁，然后进行操作。
  + 如果该put操作使得当前链表长度超过一定阈值，则将该链表转换为树，从而提高寻址效率。
  + 对于读操作，由于数组被volatile关键字修饰，因此不用担心数组的可见性问题。同时每个元素是一个Node实例，它的Key值和hash值都由final修饰，不可变更，无须关心它们被修改后的可见性问题。

|  |
| --- |
| static class Node<K,V> implements Map.Entry<K,V> {  final int hash;  final K key;  volatile V val;  volatile Node<K,V> next;  } |

### HashSet

* HashSet实现Set接口，底层是采用HashMap实现的。
* HashSet 的实现比较简单，HashSet 的绝大部分方法都是通过调用 HashMap 的方法来实现的，因此 HashSet 和 HashMap 两个集合在实现本质上是相同的。
* hashset不保证set是顺序存储的，因此只能使用迭代器（Iterator<Integer> iterator = set.iterator()）来遍历，特别是它不保证该顺序恒久不变。HashSet允许使用null元素。
* HashMap的key就是放进HashSet中对象，value是Object类型的（private static final Object PRESENT = new Object();）。
* 当调用HashSet的add方法时，实际上是向HashMap中增加了一行(key-value对)，该行的key就是向HashSet增加的那个对象，该行的value就是一个Object类型的常量。

### TreeMap

* TreeMap实现了SotredMap接口，它是有序的集合。而且是一个红黑树结构，每个key-value都作为一个红黑树的节点。
* 如果在调用TreeMap的构造函数时没有指定比较器，则根据key执行自然排序，如果指定了比较器Comparator则按照比较器来进行排序,具体取决于使用的构造方法。
* 自然排序：TreeMap的所有key必须实现Comparable接口，所有的key都是同一个类的对象；
* 定制排序：创建TreeMap对象传入了一个Comparator对象，该对象负责对TreeMap中所有的key进行排序，采用定制排序不要求Map的key实现Comparable接口。

|  |
| --- |
| public class Main {  public static void main(String[] args) {  TreeMap<Student,String> map = new TreeMap(new Comparator<Student>() {  @Override  public int compare(Student o1, Student o2) {  return o2.getAge() - o1.getAge();  }  });  map.put(new Student("zhangsan", 23), "a");  map.put(new Student("lisi", 24), "b");  map.put(new Student("wangwu", 25), "c");  Set<Student> strings = map.keySet();  for (Student s : strings) {  System.out.println(s.getName()+ " : " + s.getAge());  }  }  } |

* TreeMap中的方法：
* getFirstEntry()：获取第一个元素也就是最小的元素，一直遍历左子树；
* getLastEntry()：获取最后个元素也就是最大的元素，一直遍历右子树；
* keySet()和entrySet()方法，Map没有迭代器，要将Map转化为Set，用Set的迭代器才能进行元素迭代。

### Set集合保证数据的唯一性

set保证里面元素的唯一性其实是靠两个方法，一是equals()和hashCode()方法；

往set里面添加数据的时候一般会有隐式的操作，先是判断set集合中是否有与新添加数据的hashcode值一致的数据，如果有，那么将再进行第二步调用equals方法再进行一次判断，假如集合中没有与新添加数据hashcode值一致的数据，那么将不调用eqauls方法。

### Java集合中的快速失败机制

**fail-fast机制**是java集合（Collection）中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。

例如：当某一个线程A通过iterator去遍历某集合的过程中，若该集合的内容被其他线程所改变了，那么线程A访问集合时，就会抛ConcurrentModificationException异常，产生fail-fast事件；

当然单个线程也会出现fail-fast机制，包括ArrayList、HashMap无论在单线程和多线程状态下，都会出现fail-fast机制，即上面提到的异常

原理：

对于list，map集合类，可以通过迭代器来遍历，而Iterator其实只是一个接口，具体的实现还是要看具体的集合类中的内部类去实现Iterator并实现相关方法。

在ArrayList类中有一个内部类Itr类，类中有一个关键的属性值expectedModCount，它的初始值为ArrayList中的modCount（modCount是抽象类AbstractList中的变量，默认为0，而ArrayList 继承了AbstractList，所以也有这个变量，modCount用于记录集合操作过程中作的修改次数）；

在ArrayList进行add，remove，clear等涉及到修改集合中的元素个数的操作时，modCount就会发生改变(modCount ++)，所以当另一个线程(并发修改)或者同一个线程遍历过程中，调用相关方法使集合的个数发生改变，就会使modCount发生变化，这样在checkForComodification方法中就会抛出ConcurrentModificationException异常。

解决方式：

* 在单线程的遍历过程中，如果要进行remove操作，可以调用迭代器的remove方法而不是集合类的remove方法。迭代器中的remove方法，不会修改modCount的值，并且不会对后面的遍历造成影响，因为该方法remove不能指定元素，只能remove当前遍历过的那个元素，所以调用该方法不会发生fail-fast现象。
* 使用java并发包(java.util.concurrent)中的类来代替ArrayList 和hashMap。
  + 比如使用CopyOnWriterArrayList代替ArrayList，CopyOnWriterArrayList在是使用上跟ArrayList几乎一样，CopyOnWriter是写时复制的容器(COW)，在读写时是线程安全的。该容器在对add和remove等操作时，并不是在原数组上进行修改，而是将原数组拷贝一份，在新数组上进行修改，待完成后，才将指向旧数组的引用指向新数组，所以对于CopyOnWriterArrayList在迭代过程并不会发生fail-fast现象。但CopyOnWrite容器只能保证数据的最终一致性，不能保证数据的实时一致性。
  + 对于HashMap，可以使用ConcurrentHashMap，ConcurrentHashMap采用了锁机制，是线程安全的。在迭代方面，ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据，iterator完成后再将头指针替换为新的数据 ，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。即迭代不会发生fail-fast，但不保证获取的是最新的数据。

### final、finally、finalize的区别

1. final：用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，被其修饰的类不可继承。
2. finally：异常处理语句结构的一部分，表示总是执行。
3. finalize： Object 类的一个方法，在垃圾回收器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。该方法更像是一个对象生命周期的临终方法，当该方法被系统调用则代表该对象即将“死亡”，但是需要注意的是，我们主动行为上去调用该方法并不会导致该对象“死亡”，这是一个被动的方法（其实就是回调方法），不需要我们调用。

### String

String 字符串的一些比较

<https://blog.csdn.net/zhuzbYR/article/details/89736919>

* String 被声明为 final，因此它不可被继承。
* 在 Java 8 中，String 内部使用 char 数组存储数据。

|  |
| --- |
| public final class String  implements java.io.Serializable, Comparable<String>, CharSequence {  /\*\* The value is used for character storage. \*/  private final char value[];  } |

在Java 9 之后，String 类的实现改用 byte 数组存储字符串，同时使用 coder 来标识使用了哪种编码。

|  |
| --- |
| public final class String  implements java.io.Serializable, Comparable<String>, CharSequence {  /\*\* The value is used for character storage. \*/  private final byte[] value;  /\*\* The identifier of the encoding used to encode the bytes in {@code value}. \*/  private final byte coder;  } |

value 数组被声明为 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。

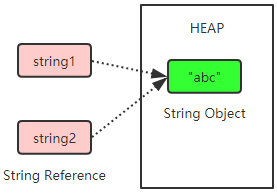
* 不可变的好处：

（1）可以缓存 hash 值

因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

（2）String Pool 的需要

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用String Pool。



（3）安全性

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 对象的那一方以为现在连接的是其它主机，而实际情况却不一定是。

（4）线程安全

String 不可变性天生具备线程安全，可以在多个线程中安全地使用

### String中hashcode的实现？

|  |
| --- |
| public int hashCode() {  int h = hash;  if (h == 0 && value.length > 0) {  char val[] = value;  for (int i = 0; i < value.length; i++) {  h = 31 \* h + val[i];  }  hash = h;  }  return h;  } |

String类中的hashCode计算方法还是比较简单的，就是以31为权，每一位为字符的ASCII值进行运算，用自然溢出来等效取模。哈希计算公式可以计为s[0]31^(n-1) + s[1]31^(n-2) + … + s[n-1]。

那为什么以31为质数呢？主要是因为31是一个奇质数，所以31\*i=32\*i-i=(i<<5)-i，这种位移与减法结合的计算相比一般的运算快很多。如果你对超过 50,000 个英文单词（由两个不同版本的 Unix 字典合并而成）进行 hash code 运算，并使用常数 31, 33, 37, 39 和 41 作为乘子，每个常数算出的哈希值冲突数都小于7个，所以在上面几个常数中，常数 31 被 Java 实现所选用也就不足为奇了。

### String、StringBuilder、StringBuffer的区别

1. 可变性

+ String 不可变

+ StringBuffer 和 StringBuilder 可变

2. 线程安全

+ String 不可变，因此是线程安全的

+ StringBuilder 不是线程安全的

+ StringBuffer 是线程安全的，内部使用 synchronized 进行同步

### 栈和队列的区别

* 相同点：

1. 都是线性结构。
2. 插入操作都是限定在表尾进行。
3. 都可以通过顺序结构和链式结构实现。
4. 插入与删除的时间复杂度都是O(1)，在空间复杂度上两者也一样。
5. 多链栈和多链队列的管理模式可以相同。

* 不同点：

1. 队列先进先出，栈先进后出
2. 删除数据元素的位置不同，栈的删除操作在表尾进行，队列的删除操作在表头进行。
3. 应用场景不同；常见栈的应用场景包括括号问题的求解，表达式的转换和求值，函数调用和递归实现，深度优先搜索遍历等；常见的队列的应用场景包括计算机系统中各种资源的管理，消息缓冲器的管理和广度优先搜索遍历等。
4. 顺序栈能够实现多栈空间共享，而顺序队列不能。

### java的反射机制

（1）反射的定义

反射就是动态加载对象，并对对象进行剖析。在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法，这种动态获取信息以及动态调用对象方法的功能成为Java反射机制。

要想解剖一个类，必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法。所以先要获取到每一个字节码文件对应的Class类型的对象。

（2）如何使用反射？

A获取Class对象的三种方式

* + Class.forName("全限类名"); 例如：com.mysql.jdbc.Driver Driver类已经被加载到 jvm中，并且完成了类的初始化工作就行了.
  + 类名.class; 获取Class<？> clz 对象
  + 对象.getClass();

|  |
| --- |
| public class Fanshe {  public static void main(String[] args) {  //第一种方式获取Class对象  Student stu1 = new Student();//这一new 产生一个Student对象，一个Class对象。  Class stuClass = stu1.getClass();//获取Class对象  System.out.println(stuClass.getName());  //第二种方式获取Class对象  Class stuClass2 = Student.class;  //第三种方式获取Class对象  try {  Class stuClass3 = Class.forName("fanshe.Student");//注意此字符串必须带包名的类路径，包名.类名  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  } |

（3）反射的作用？

* 可以通过反射获取类的构造方法、成员变量、成员方法等并使用，比如通过Class.forName()加载数据库的驱动程序；
* 通过反射运行xml配置文件内容，比如spring通过 XML 配置模式装载 Bean；
* 通过反射越过泛型检查；

（4）反射源码

MethodAccessor 对象其实就是具体去生成反射类的入口；实际的 MethodAccessor 实现有两个版本，一个是 Native 版本，一个是 Java 版本。Native 版本一开始启动快，但是随着运行时间边长，速度变慢。Java 版本一开始加载慢，但是随着运行时间边长，速度变快。正是因为两种存在这些问题，所以第一次加载的时候我们会发现使用的是 NativeMethodAccessorImpl 的实现，而当反射调用次数超过 15 次之后，则使用 MethodAccessorGenerator 生成的 MethodAccessorImpl 对象去实现反射。

### Class.forName()和ClassLoader的区别？

（1）相同点：在java中Class.forName()和ClassLoader都可以对类进行加载。

* ClassLoader就是遵循双亲委派模型最终调用启动类加载器的类加载器，实现的功能是“通过一个类的全限定名来获取描述此类的二进制字节流”，获取到二进制流后放到JVM中。
* Class.forName()方法实际上也是调用的CLassLoader来实现的。

（2）区别：

* Class.forName()除了将类.class文件加载到jvm中，还会对类进行解释，执行类中的static块，也就是对类进行初始化；
* ClassLoader.loadclass()不会对类进行初始化，只有执行newInstance时才会初始化类。

（3）应用场景：

* Spring 框架中的 IOC 的实现就是使用的ClassLoader；
* JDBC通常是使用 Class.forName() 方法来加载数据库连接驱动，因为在 JDBC 规范中明确要求 Driver(数据库驱动)类必须向 DriverManager 注册自己。

### 数组复制的几种方式？高效率？

（1）System.arraycopy（浅拷贝）【推荐使用】

|  |
| --- |
| /\*\*参数：src - 源数组。srcPos - 源数组中的起始位置。dest - 目标数组。destPos - 目标数据中的起始位置。length - 要复制的数组元素的数量\*/  public static native void arraycopy(Object src,int srcPos,Object dest,int destPos,int length) |

（2）使用clone方法（对数组是浅拷贝，对于对象而言是深拷贝）

|  |
| --- |
| /\*\*java.lang.Object类的clone()方法为protected类型，不可直接调用，需要先对要克隆的类进行下列操作：  首先被克隆的类实现Cloneable接口；然后在该类中覆盖clone()方法，并且在该clone()方法中调用super.clone()；这样，super.clone()便可以调用java.lang.Object类的clone()方法。\*/ |

（3）Arrays.copyOf（浅拷贝）

|  |
| --- |
| //实际上它调用的就是System.arraycopy，所以肯定也是浅拷贝  public static byte[] copyOfRange(byte[] original, int from, int to) {  int newLength = to - from;  if (newLength < 0)  throw new IllegalArgumentException(from + " > " + to);  byte[] copy = new byte[newLength];  System.arraycopy(original, from, copy, 0,  Math.min(original.length - from, newLength));  return copy;  } |

（4）for循环遍历赋值

### Java中NIO？

JDK 1.4中的java.nio.\*包中引入新的Java I/O库，其目的是提高速度。实际上，“旧”的I/O包已经使用NIO重新实现过，即使我们不显式的使用NIO编程，也能从中受益。



* + IO是面向流的处理，NIO是面向块(缓冲区)的处理
  + 面向流的I/O：系统一次一个字节地处理数据。
  + 面向块(缓冲区)的I/O：系统以块的形式处理数据。

NIO主要有三个核心部分组成：buffer缓冲区；Channel管道；Selector选择器；

### java的反射机制

Java编写的程序，一次编译，到处运行。这也是Java程序为什么是无关平台的所在，原因在于，java的源代码会被编译成.class文件字节码，只要装有Java虚拟机JVM的地方（Java提供了各种不同平台上的虚拟机制，第一步由Java IDE进行源代码编译，得到相应类的字节码.class文件，第二步，Java字节码由JVM执行解释给目标计算机，第三步，目标计算机将结果呈现给我们计算机用户；因此，Java并不是编译机制，而是解释机制），.class文件畅通无阻。

Java的反射机制，操作的就是这个.class文件，首先加载相应类的字节码（运行idea的时候，.class文件的字节码会加载到内存中），随后取出（反射 reflect）字节码中的构造函数、方法以及变量（字段）。

### == 和 equals( ) 方法的区别

首先，==号在比较基本数据类型时比较的是值，而用==号比较两个对象时比较的是两个对象的地址值，那equals()方法呢？我们可以通过查看源码知道，equals()方法存在于Object类中，因为Object类是所有类的直接或间接父类，也就是说所有的类中的equals()方法都继承自Object类，而通过源码我们发现，Object类中equals()方法底层依赖的是==号，那么，在所有没有重写equals()方法的类中，调用equals()方法其实和使用==号的效果一样，也是比较的地址值，然而，Java提供的所有类中，绝大多数类都重写了equals()方法，重写后的equals()方法一般都是比较两个对象的值。  
HashMap中，如果要比较key是否相等，要同时使用这两个函数！因为自定义的类的hashcode()方法继承于Object类，其hashcode码为默认的内存地址，这样即便有相同含义的两个对象，比较也是不相等的。HashMap中的比较key是这样的，先求出key的hashcode(),比较其值是否相等，若相等再比较equals(),若相等则认为他们是相等的。若equals()不相等则认为他们不相等。如果只重写hashcode()不重写equals()方法，当比较equals()时只是看他们是否为同一对象（即进行内存地址的比较）,所以必定要两个方法一起重写。HashMap用来判断key是否相等的方法，其实是调用了HashSet判断加入元素 是否相等。重载hashCode()是为了对同一个key，能得到相同的Hash Code，这样HashMap就可以定位到我们指定的key上。重载equals()是为了向HashMap表明当前对象和key上所保存的对象是相等的，这样我们才真正地获得了这个key所对应的这个键值对。

<https://www.nowcoder.com/questionTerminal/bf37322cb9554e3887f70c71b558ad98?orderByHotValue=1&page=1&onlyReference=false>

### 3.1.29 java多态 i ++ 和 ++ i 的区别

java多态的必要条件 1 重载 2 继承 3 父类引用指向子类对象

一.i++

i++的操作分三步：

（1）栈中取出i

（2）i自增1

（3）将i存到栈

所以i++不是原子操作，上面的三个步骤中任何一个步骤同时操作，都可能导致i的值不正确自增

二.++i

在多核的机器上，cpu在读取内存i时也会可能发生同时读取到同一值，这就导致两次自增，实际只增加了一次。

综上，我认为i++和++i都不是原子操作

### *ThreadLocal 原理分析*

一句话说就是 ThreadLocal 适用于每个线程需要自己独立的实例且该实例需要在多个方法中被使用（相同线程数据共享），也就是变量在线程间隔离（不同的线程数据隔离）而在方法或类间共享的场景。

从上面的结构图，我们已经窥见ThreadLocal的核心机制：

每个Thread线程内部都有一个Map。  
Map里面存储线程本地对象（key）和线程的变量副本（value）  
Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。  
对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，彼此之间互不干扰。

### java的异常处理机制

Error 类和 Exception 类的父类都是 Throwable 类，他们的区别如下。

Error 类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢出等。

对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。

Exception 类表示程序可以处理的异常，需要捕捉、需要处理的异常，是由与程序设计的不完善而出现的问题，程序必须处理的问题。

Exception 类又分为运行时异常（RuntimeException）和受检查的异常(CheckedException )

一般异常（checkedException）主要是指IO异常、SQL异常等。对于这种异常，JVM要求我们必须对其进行cathc处理，所以，面对这种异常，不管我们是否愿

意，都是要写一大堆的catch块去处理可能出现的异常。如果程序没有处理 CheckedException异常，该程序在编译时就会发生错误无法编译

运行时异常（runtimeException）我们一般不处理，当出现这类异常的时候程序会由虚拟机接管。

例如，数组脚本越界（ArrayIndexOutOfBoundsException），空指针异常（NullPointerException）、类转换异常（ClassCastException）

出现运行时异常的时候，程序会将异常一直向上抛，一直抛到遇到处理代码，如果没有catch块进行处理，到了最上层，如果是多线程就有Thread.run()抛出，如果不是多线程那么就由main.run()抛出。抛出之后，如果是线程，那么该线程也就终止了，如果是主程序，那么该程序也就终止了。

Java的异常处理机制原理

Java通过面向对象的方式对异常进行处理，Java把异常按照不同的类型进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它都是Throwable

或其子类的实例。当一个方法出现异常后就会抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并对异常进行处理。Java的

异常处理是通过5个关键词来实现的：try catch throw throws finally。

一般情况下是用try来执行一段程序，如果出现异常，系统会抛出（throws），我们可以通过它的类型来捕捉它，或最后由缺省处理器来处理它（finally）。

try：用来指定一块预防所有异常的程

catch：紧跟在try后面，用来捕获异常

throw：用来明确的抛出一个异常

throws：用来标明一个成员函数可能抛出的各种异常

finally：确保一段代码无论发生什么异常都会被执行的一段代码

## 多线程编程

### 创建线程的3种方式

（1）继承Thread类创建线程

1. 自定义线程类继承Thread类；
2. 重写run()方法，编写线程执行体；
3. 创建线程对象，调用start()方法启动线程；

|  |
| --- |
| public class ThreadCurrent extends Thread {  @Override  public void run() {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + "..." + i);  }  }  public static void main(String[] args) {  new ThreadCurrent().start();  new ThreadCurrent().start();  }  } |

（2）实现Runnable接口创建线程

1. 定义Runnable接口实现类
2. 重写run()方法，编写线程执行
3. 创建线程对象，调用start()方法启动线程

|  |
| --- |
| public class ThreadCurrent implements Runnable {  @Override  public void run() {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + "..." + i);  }  }  public static void main(String[] args) {  ThreadCurrent current = new ThreadCurrent();  new Thread(current).start();  new Thread(current).start();  }  } |

（3）实现Callable接口创建线程

1. 实现Callable接口，需要返回值类型
2. 重写call方法，需要抛出异常
3. 创建目标对象
4. 创建执行服务：ExecutorService ser = Executors.newFixedThreadPool(1);
5. 提交执行：Future result1 = ser.submit(t1);
6. 获取结果：boolean r1 = result1.get()
7. 关闭服务：ser.shutdownNow()

|  |
| --- |
| public class ThreadCurrent implements Callable<Boolean> {  @Override  public Boolean call() throws Exception {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + "..." + i);  }  return true;  }  public static void main(String[] args) throws Exception {  ThreadCurrent callable = new ThreadCurrent();  //创建执行任务  ExecutorService service = Executors.newFixedThreadPool(2);  //提交执行  Future<Boolean> submit1 = service.submit(callable);  Future<Boolean> submit2 = service.submit(callable);  //获取结果  Boolean aBoolean1 = submit1.get();  Boolean aBoolean2 = submit1.get();  //判断线程是否顺利结束或有异常  System.out.println(aBoolean1);  System.out.println(aBoolean2);  //关闭服务  service.shutdown();  }  } |

（4）比较哪一种好？

实现 Runnable 接口比继承Thread 类所具有的优势：

1. 适合多个相同的程序代码的线程去处理同一个资源；
2. 可以避免 java 中的单继承的限制；
3. 增加程序的健壮性， 代码可以被多个线程共享，代码和数据独立；
4. 线程池只能放入实现 Runable 或 callable 类线程， 不能直接放入继承 Thread 的类；
5. runnable 实现线程可以对线程进行复用， 因为runnable是轻量级的对象，重复new不会耗费太大资源，而 Thread 则不然，它是重量级对象，而且线程执行完就完了，无法再次利用。

### 守护线程和非守护线程

在Java中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)

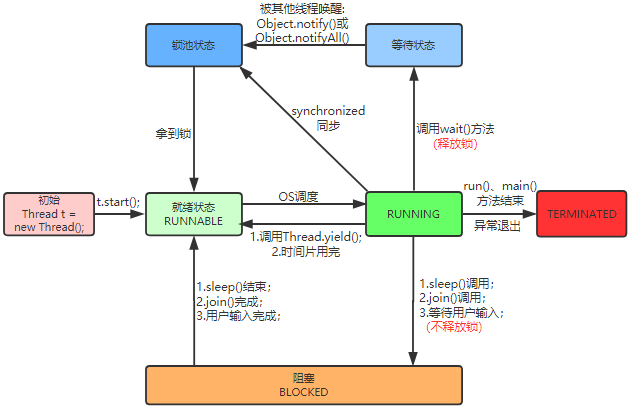
只要当前JVM实例中存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。守护线程的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

(1) thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。

(2) 在Daemon线程中产生的新线程也是Daemon的。

### 线程的生命周期

每一个线程都有自己的局部变量表、程序计数器,以及生命周期等。



线程的生命周期可分为5个主要阶段：

1.线程的NEW状态

当我们用关键字new创建一个Thread对象时，此时它并不处于执行状态，因为没有调用start方法启动该线程,那么线程的状态为NEW状态，准确地说，它只是Thread对象的状态，因为在没有start之前，该线程根本不存在，与你用关键字new创建一个普通的Java对象没什么区别。

2.线程的RUNNABLE状态

线程对象进入RUNNABLE状态必须调用start方法,那么此时才是真正地在JVM进程中创建了一个线程,线程一经启动就可以立即得到执行吗?答案是否定的,线程的运行与否和进程一样都要听令于CPU的调度,那么我们把这个中间状态称为可执行状态 (RUNNABLE),也就是说它具备执行的资格,但是并没有真正地执行起来而是在等待CPU的调度。

由于存在Running状态,所以不会直接进入BLOCKED状态和TERMINATED状态,即使是在线程的执行逻辑中调用wait,sleep或者其他block的IO操作等,也必须先获得CPU的调度执行权才可以,严格来讲, RUNNABLE的线程只能意外终止或者进入 RUNNING状态。

3.线程的RUNNING状态

一旦CPU通过轮询或者其他方式从任务可执行队列中选中了线程,那么此时它才能真正地执行自己的逻辑代码,需要说明的一点是一个正在RUNNING状态的线程事实上也是 RUNNABLE的,但是反过来则不成立。

在该状态中,线程的状态可以发生如下的状态转换。

（1）直接进入TERMINATED状态,比如调用JDK已经不推荐使用的stop方法或者判断某个逻辑标识。

（2）进入BLOCKED状态,比如调用了sleep,或者wait方法而加入了waitSet中。

（3）进行某个阻塞的IO操作,比如因网络数据的读写而进入了BLOCKED状态。

（4）获取某个锁资源,从而加入到该锁的阻塞队列中而进入了BLOCKED状态。

（5）由于CPU的调度器轮询使该线程放弃执行,进入RUNNABLE状态。

（6）线程主动调用yield方法,放弃CPU执行权,进入RUNNABLE状态。

4.线程的BLOCKED状态

上面已经列举了线程进入BLCOKED状态的原因,此处就不再赘述了,线程在 BLOCKED状态中可以切换至如下几个状态。

（1）直接进入TERMINATED状态,比如调用JDK已经不推荐使用的stop方法或者意外死亡(JVM Crash)。

（2）线程阻塞的操作结束,比如读取了想要的数据字节进入到RUNNABLE状态。

（3）线程完成了指定时间的休眠,进入到了RUNNABLE状态。

（4）Wait中的线程被其他线程notify/notifyall唤醒,进入RUNNABLE状态。

（5）线程获取到了某个锁资源,进入RUNNABLE状态。

（6）线程在阻塞过程中被打断,比如其他线程调用了interrupt方法,进入RUNNABLE状态。

5.线程的TERMINATED状态

TERMINATED是一个线程的最终状态,在该状态中线程将不会切换到其他任何状态,线程进入TERMINATED状态,意味着该线程的整个生命周期都结束了,下列这些情况将会使线程进入TERMINATED状态。

（1）程运行正常结束,结束生命周期。

（2）线程运行出错意外结束。

（3）JVM Crash,导致所有的线程都结束

### sleep()、wait()、yield()、join()

（1）sleep()方法：不会释放对象锁

1. sleep()是Thread类中的方法；
2. sleep()方法导致了程序暂停，但是他的监控状态依然保持着，当指定的时间到了又会自动恢复运行状态。在调用sleep()方法的过程中，线程不会释放对象锁。

（2）wait()方法：释放对象锁

1. wait()是Object类中的方法；
2. wait()方法会使当前线程阻塞，直到调用notify()，则被唤醒。
3. wait()和notify()会对对象的“锁标志”进行操作，所以它们必须在synchronized函数或synchronized代码块中进行调用，否则在运行时会发生IllegalMonitorStateException的异常。

（3）yield，()方法:不会释放对象锁

1. yield()是Thread类中的方法，作用：**暂停当前正在执行的线程对象，并执行其他线程**。其实yield()方法是一种启发式的方法，会提醒调度器我愿意放弃当前的CPU资源，如果CPU资源不紧张，则会忽略这种提醒。
2. yield()不会导致线程的等待/睡眠/阻塞状态。

（4）join()方法：释放对象锁

1. join()方法是指等待调用join()方法的线程执行结束，程序才会继续执行下去，在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。
2. join方法中如果传入参数，则表示：如果A线程中掉用B线程的join(10)，则表示A线程会等待B线程执行10毫秒，10毫秒过后，A、B线程并行执行。需要注意的是，jdk规定，join(0)的意思不是A线程等待B线程0秒，而是A线程等待B线程无限时间，直到B线程执行完毕，即join(0)等价于join()。(其实join()中调用的是join(0))
3. 方法join(long)的功能在内部是使用wait(long)来实现的，所以join(long)方法具有释放锁的特点。

（5）sleep()方法和yield()方法区别：

在JDK1.5以前的版本中yield的方法事实上是调用了sleep(0)，但是它们之间存在着本质的区别：

* sleep会导致当前线程暂停指定的时间，没有CPU时间片的消耗。
* yield只是对CPU调度器的一个提示，如果CPU调度器没有忽略这个提示，它会导致线程上下文的切换；
* sleep会使线程短暂block，会在给定的时间内释放CPU资源。
* 调用yield方法会使当前线程从RUNNING状态切换到RUNNABLE状态。
* sleep几乎百分之百地完成了给定时间的休眠，而yield的提示并不能一定担保。
* 一个线程sleep，另一个线程调用interrupt可以捕获到中断信号，而yield则不会。

### JAVA中的锁有哪些？

|  |  |
| --- | --- |
| 锁 | 解释 |
| 公平锁/非公平锁 | 公平锁是指多个线程按照申请锁的顺序来获取锁 |
| 可重入锁 | 同一线程在外层方法已获取锁时，进入内层方法会自动获取锁 |
| 独享锁/共享锁 | 一次可被单个/多个线程所持有，例如ReadWriteLock的写锁/读锁 |
| 互斥锁/读写锁 | 一种互斥锁：ReentrantLock；一种读写锁：ReadWriteLock |
| 乐观锁/悲观锁 | 悲观锁认为对于同一个数据的并发操作一定会发生修改 CAS是乐观锁 |
| 分段锁 | 一种锁的设计，具体应用有ConcurrentHashMap |
| 自旋锁 | 尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取 |
| 偏向锁/轻量级锁/重量级锁 | 指锁的状态，并且是针对synchronized   * 偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。 * 轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。 |

其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，”自旋”一词就是因此而得名。

ABA问题的根本在于cas在修改变量的时候，无法记录变量的状态，比如修改的次数，否修改过这个变量。这样就很容易在一个线程将A修改成B时，另一个线程又会把B修改成A,造成casd多次执行的问题

### 乐观锁和悲观锁

（1）乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

* version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加1。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。update table set x=x+1, version=version+1 where id=#{id} and version=#{version};
* CAS操作方式：即compare and swap 或者 compare and set，是一种**无锁算法**，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

（2）悲观锁

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

（3）适用场景

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

总结：两种所各有优缺点，读取频繁使用乐观锁，写入频繁使用悲观锁。

### CAS底层实现原理？

（1）CAS原理

Conmpare And Swap是用于实现多线程同步的原子指令。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。整个比较并替换的操作是一个原子操作。

JAVA中的CAS操作都是通过sun包下Unsafe类实现，而Unsafe类中的方法都是native方法，由JVM本地实现，也就是说Unsafe类中的方法都直接调用操作系统底层资源执行相应任务。

（2）CAS算法的缺点

* **ABA问题**：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。
* ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A－B－A 就会变成1A-2B－3A。
* 从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference，类中的**compareAndSet方法**就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
* **循环时间长开销大**：自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。
* **只能保证一个共享变量的原子操作**：CAS只对单个共享变量有效，当操作涉及跨多个共享变量时CAS无效。但是从JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行 CAS 操作。

### synchronized底层实现原理？

（1）对象（或代码块）的Synchronized同步

Synchronized进过编译，会在同步块的前后分别形成Monitorenter和Monitorexit这个两个字节码指令。。

* Monitorenter：

每个对象都是一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

* Monitorexit：

释放锁则是通过monitorexit指令，执行monitorexit的线程必须是monitor的所有者，指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

（2）方法的synchronized同步

加了synchronized关键字的方法，相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是据该标示符来实现方法的同步的：

1. 当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。
2. 在方法执行期间，其他任何线程都无法再获得同一个monitor对象。

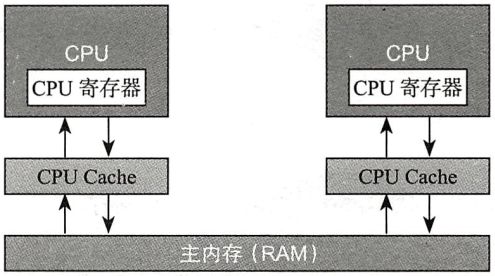
（3）synchronized的缺点：

* 当一个代码块被synchronized修饰的时候，一个线程获取到了锁，并且执行代码块，那么其他的线程需要等待正在使用的线程释放掉这个锁，那么释放锁的方法只有两种，一种是代码执行完毕自动释放，一种是发生异常以后jvm会让线程去释放锁。那么如果这个正在执行的线程遇到什么问题，比如等待IO或者调用sleep方法等等被阻塞了，无法释放锁，而这时候其他线程只能一直等待，将会特别影响效率。
* 当一个文件，同时被多个线程操作时，读操作和写操作会发生冲突，写操作和写操作会发生冲突，而读操作和读操作并不会冲突，但是如果我们用synchronized的话，会导致一个线程在读的时候，其他线程想要读的话只能等待。

### 线程安全的本质？

主内存的读写速度远远落后于CPU的速度。Cache的出现是为了解决CPU直接访问内存效率低下问题的：

* + 程序在运行的过程中，会将运算所需要的数据从主存复制一份到CPU Cache中，这样CPU进行计算时就可以直接对 CPU Cache中的数据进行读取和写人；
  + 当运算结束之后，再将CPU Cache中的最新数据刷新到主内存当中；

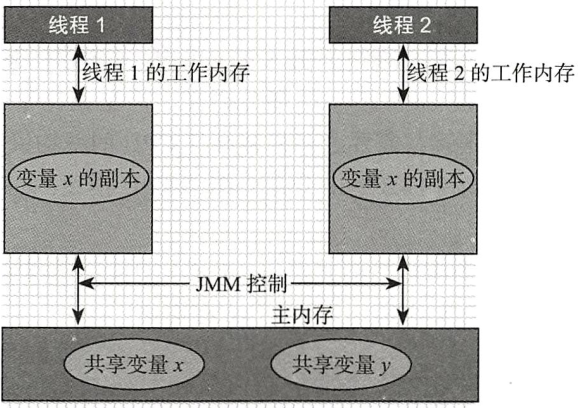


CPU通过直接访问Cache的方式替代直接访问主存的方式极大地提高了CPU的吞吐能力。但是会导致缓存不一致的问题。比如i++;这个操作，多个线程都有自己的工作内存（对应于CPU的Cache），变量i会在对个线程的本地内存中都存在一个副本，如果有两个线程同时执行i++;操作，可能最后的结果任然是1。

解决方式有两种：

* + （1）通过总线加锁的方式：在早期的CPU中使用这种方式，但是会造成阻塞，效率低下；
  + （2）通过缓存一致性协议：当CPU发现变量是共享变量的时候，在读的时候，不做任何操作；在写得时候发出信号通知其他CPU将该变量的Cache line设置成无效状态，这样就只能从主内存中获取。

Java内存模型指定了Java虚拟机如何与计算机的主存进行工作，决定了一个线程对共享变量的写入何时对其他线程可见。



（1）原子性

简单的读取和赋值操作（x = 10）是原子性的，将一个变量赋给另外一个变量的操作（y=x或y++或z=z+1）不是原子性的；Java内存模型(JMM)只保证了基本读取和赋值的原子性操作，其他的均不保证，如果想要使得某些代码片段具备原子性，需要使用关键字synchronized，或者JUC中的lock。如果想要使得int等类型自增操作具备原子性，可以使用JUC包下的原子封装类型java.uil.concurrent.atomic.

（2）可见性

在多线程的环境下，如果某个线程首次读取共享变量,则首先到主内存中获取该变量，然后存入工作内存中，以后只需要在工作内存中读取该变量即可。同样如果对该变量执行了修改的操作，则先将新值写入工作内存中，然后再刷新至主内存中。但是什么时候最新的值会被刷新至主内存中是不太确定的；java提供三种方式保证可见性：

* + 使用关键字volatile：当一个变量被volatile关键字修饰时，对于共享资源的读操作会直接在主内存中进行(当然也会缓存到工作内存中，当其他线程对该共享资源进行了修改，则会导致当前线程在工作内存中的共享资源失效，所以必须从主内存中再次获取)，对于共享资源的写操作当然是先要修改工作内存，但是修改结束后会立刻将其刷新到主内存中。
  + synchronized关键字：synchronized关键字能够保证同一时刻只有一个线程获得锁，然后执行同步方法，并且还会确保在锁释放之前，会将对变量的修改刷新到主内存当中。
  + 通过JUC提供的显式锁Lock也能够保证可见性：Lock的lock方法能够保证在同一时刻只有一个线程获得锁然后执行同步方法，并且会确保在锁释放(unlock方法)之前会将对变量的修改刷新到主内存当中。

（3）顺序性

在Java的内存模型中，允许编译器和处理器对指令进行重排序，在单线程的情况下，重排序并不会引起什么问题，但是在多线程的情况下，重排序会影响到程序的正确运行，Java提供了三种保证有序性的方式：

* + 使用volatile关键字来保证有序性。
  + 使用synchronized关键字来保证有序性。
  + 使用显式锁Lock来保证有序性。

### volaile底层实现原理？

* + volatile关键字不能保证原子性；
  + volatile关键字可以保证变量的可见性；volatile修饰数组时，也可以保证数组中的元素的可见性；
  + volatile修饰的变量不会被编译器优化，即不会造成指令重排；

底层原理：

通过对OpenJDK下的unsafe.cpp的源码查看，发现被volatile修饰的变量有一个“lock”的前缀，这个前缀实际上是一个内存屏障，可以提供下面几个保障：

* + 可以确保指令重排序的时候，不会将其后面的代码排到内存屏障之前；不会将其前面的代码排到内存屏障之后。
  + 确保在执行到内存屏障修饰的指令时，前面的代码全部执行完成。
  + 如果线程工作内存中值的修改了，就强制将其刷新到主内存中。
  + 如果是写操作，则会导致其他线程工作内存(CPU Cache)中的缓存数据失效。

### ReentrantLock底层实现原理？

ReentrantLock是JDK1.5引入的，它拥有与synchronized相同的并发性，并提供了高级功能(例如，中断锁等候、条件变量等)，使用ReentrantLock比synchronized能获得更好的可伸缩性。

ReentrantLock的实现基于AQS(AbstractQueuedSynchronizer)和LockSupport。

* AQS主要利用硬件原语指令(CAS compare-and-swap)，来实现轻量级多线程同步机制，并且不会引起CPU上文切换和调度，并且可以保证内存可见性和原子性（线程安全的三要素：原子性、可见性、顺序性）。
* AQS的本质上是一个同步器/阻塞锁的基础框架，其作用主要是提供加锁、释放锁，并在内部维护一个FIFO等待队列，用于存储由于锁竞争而阻塞的线程。

原理：

ReentrantLock 构造函数中提供了两种锁：公平锁（FairSync）和非公平锁（NonfairSync、默认）。

|  |
| --- |
| Lock unfairLock = new ReentrantLock();  Lock fairLock = new ReentrantLock(true);//调用构造函数时，传入参数true得到的就是公平锁 |

* 在公平的锁上，线程按照他们发出请求的顺序获取锁，FIFO先进先出顺序，每个线程获取锁的过程是公平的，等待时间最长的会最先被唤醒获取锁。但在非公平锁上，则允许插队。
* 当一个线程请求非公平锁时，如果在发出请求的同时该锁变成可用状态，那么这个线程会跳过队列中所有的等待线程而获得锁。
* 非公平的 ReentrantLock 并不提倡插队行为，但是无法防止某个线程在合适的时候进行插队。
* 在公平的锁上，如果有另一个线程持有锁或者有其他线程在等待队列中等待这个锁，那么新发出的请求的线程将被放入到队列中。
* 而非公平锁上，只有当锁被某个线程持有时，新发出请求的线程才会被放入队列中。

ReentrantLock在采用非公平锁构造时，首先检查锁状态，如果锁可用，直接通过CAS设置成持有状态，且把当前线程设置为锁的拥有者。如果当前锁已经被持有，那么接下来进行可重入检查，如果可重入，需要为锁状态加上请求数。如果不属于上面两种情况，那么说明锁是被其他线程持有，当前线程应该放入等待队列。在放入等待队列的过程中，首先要检查队列是否为空队列，如果为空队列，需要创建虚拟的头节点，然后把对当前线程封装的节点加入到队列尾部。由于设置尾部节点采用了CAS，为了保证尾节点能够设置成功，这里采用了无限循环的方式，直到设置成功为止。

在完成放入等待队列任务后，则需要维护节点的状态，以及及时清除处于Cancel状态的节点，以帮助垃圾收集器及时回收。如果当前节点之前的节点的等待状态小于1，说明当前节点之前的线程处于等待状态(挂起)，那么当前节点的线程也应处于等待状态(挂起)。挂起的工作是由LockSupport类支持的，LockSupport通过JNI调用本地操作系统来完成挂起的任务(java中除了废弃的suspend等方法，没有其他的挂起操作)。

在当前等待的线程，被唤起后，检查中断状态，如果处于中断状态，那么需要中断当前线程。

### synchronized和volatile的区别？

volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

* + 使用上：volatile仅能修饰实例变量或者类变量；synchronized可以修饰方法、类、代码块，不能修饰变量。
  + 原子性可见性：volatile可以保证变量的修改可见性，不能保证原子性；而synchronized可以保证变量的修改可见性和原子性。
  + 指令重排：volatile禁止编译器对其进行指令重排，可以保证有序性；synchronized修饰的同步方法也可以保证顺序性，但是代码块中的指令可以发生指令重排。
  + volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

### sychronized和ReentrantLock区别：

（1）相似点

* 两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待。

（2）功能的区别

* Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，就类似于操作系统来控制实现和用户自己敲代码实现的区别。
* Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以要在finally中声明释放锁（unlock()）。

（3）其他

* ReenTrantLock的字面意思就是再进入的锁，其实synchronized关键字所使用的锁也是可重入的。两者都是同一个线程每进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。
* ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。
* ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。
* synchronized多个线程竞争一个锁，会一直尝试获取该锁，直到拿到为止，而ReentrantLock可以使用tryLock()， tryLock(long time,TimeUnit unit)方法尝试去获取锁，如果获取不到，可以执行其他逻辑，更加灵活。

（4）使用场景

* 在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票等，使用 ReentrantLock。
* ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它。
* 一般还是要使用 synchronized 开发。

### Synchronized加静态和非静态方法上？

（1）Synchronzied 修饰非静态方法实际上是对调用该方法的对象加锁，俗称“对象锁”。

Java中每个对象都有一个锁，并且是唯一的。假设分配的一个对象空间，里面有多个方法，相当于空间里面有多个小房间，如果我们把所有的小房间都加锁，因为这个对象只有一把钥匙，因此同一时间只能有一个人打开一个小房间，然后用完了还回去，再由JVM 去分配下一个获得钥匙的人。

|  |
| --- |
| * 情况1：同一个对象在两个线程中分别访问该对象的两个同步方法   结果：会产生互斥。  解释：因为锁针对的是对象，当对象调用一个synchronized方法时，其他同步方法需要等待其执行结束并释放锁后才能执行。   * 情况2：不同对象在两个线程中调用同一个同步方法   结果：不会产生互斥。  解释：因为是两个对象，锁针对的是对象，并不是方法，所以可以并发执行，不会互斥。形象的来说就是因为我们每个线程在调用方法的时候都是new 一个对象，那么就会出现两个空间，两把钥匙， |

（2）Synchronzied 修饰静态方法，实际上是对该类对象加锁，称为类锁；

|  |
| --- |
| * 情况1：用类直接在两个线程中调用两个不同的同步方法   结果：会产生互斥。  解释：因为对静态对象加锁实际上对类（.class）加锁，类对象只有一个，可以理解为任何时候都只有一个空间，里面有N个房间，一把锁，因此房间（同步方法）之间一定是互斥的。  注：上述情况和用单例模式声明一个对象来调用非静态方法的情况是一样的，因为永远就只有这一个对象。所以访问同步方法之间一定是互斥的。   * 情况2：用一个类的静态对象在两个线程中调用静态方法或非静态方法   结果：会产生互斥。  解释：因为是一个对象调用，同上。   * 情况3：一个对象在两个线程中分别调用一个静态同步方法和一个非静态同步方法   结果：不会产生互斥。  解释：因为虽然是一个对象调用，但是两个方法的锁类型不同，调用的静态方法实际上是类对象在调用，即这两个方法产生的并不是同一个对象锁，因此不会互斥，会并发执行。 |

### 3.2.14.2 可重入锁

如果一个线程已经拥有了某一个对象的锁，当执行到需要获取锁的代码段时可以直接进行，无需再次获取锁。不然他会把自己等死。

### 为什么使用线程池？线程池的好处？

（1）什么是线程池：

java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池；多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。一个线程池包括以下四个基本组成部分：

* + 线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；
  + 工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；
  + 任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；
  + 任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

（2）线程池作用：就是限制系统中执行线程的数量。

根据系统的环境情况，可以自动或手动设置线程数量，达到运行的最佳效果；少了浪费了系统资源，多了造成系统拥挤效率不高。用线程池控制线程数量，其他线程排队等候。一个任务执行完毕，再从队列的中取最前面的任务开始执行。若队列中没有等待进程，线程池的这一资源处于等待。当一个新任务需要运行时，如果线程池中有等待的工作线程，就可以开始运行了；否则进入等待队列。

（3）为什么要用线程池：

* + 降低资源消耗：每个工作线程都可以被重复利用，可执行多个任务，减少了创建和销毁线程的次数；
  + 可以根据系统的承受能力，调整线程池中工作线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

Java里面线程池的顶级接口是Executor，但是严格意义上讲Executor并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是ExecutorService。

### 线程池的拒绝策略

    RejectedExecutionHandler rejected = null;

    rejected = new ThreadPoolExecutor.AbortPolicy();//默认，队列满了丢任务抛出异常

    rejected = new ThreadPoolExecutor.DiscardPolicy();//队列满了丢任务不异常

    rejected = new ThreadPoolExecutor.DiscardOldestPolicy();//将最早进入队列的任务删，之后再尝试加入队列

    rejected = new ThreadPoolExecutor.CallerRunsPolicy();//如果添加到线程池失败，那么主线程会自己去执行该任务

原文链接：https://blog.csdn.net/xxj\_jing/article/details/84835476

### 创建线程池的四种方式

Java通过Executors提供四种线程池，分别为：

1. newCachedThreadPool：创建一个可缓存线程池，当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程，当有任务来时，又智能的添加新线程来执行。
2. newFixedThreadPool：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
3. newScheduledThreadPool：创建一个定长线程池，支持定时及周期性任务执行。
4. newSingleThreadPool：创建一个单线程化的线程池，这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池，保证所有任务按照指定顺序(FIFO，LIFO，优先级)执行。

（1）newCachedThreadPool

|  |
| --- |
| public static void main(String[] args) {  // 线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。  // 这里如果去除sleep，你会发现在这个循环中创建了新的线程，因为前一个任务没有执行完，所以创建新线程执行下一个任务。  ExecutorService cachedThreadPool = Executors.newCachedThreadPool();  for (int i = 0; i < 10; i++) {  final int index = i;  try {  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  cachedThreadPool.execute(new Runnable() {  public void run() {  System.out.println(index);  }  });  }  } |

（2）newFixedThreadPool：池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()

|  |
| --- |
| public static void main(String[] args) {  // 创建一个定长的线程池，也是根据需要去调用线程，比如线程定为100个，而循环只有10个，那么也只会用到前10个进程。  ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);  for (int i = 0; i < 10; i++) {  final int index = i;  fixedThreadPool.execute(new Runnable() {  public void run() {  try {  System.out.println(index);  Thread.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  });  }  } |

（3）newScheduledThreadPool

|  |
| --- |
| public static void main(String[] args) {  // 创建一个定长线程池，支持定时及周期性任务执行。  ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);  for (int i = 0; i < 10; i++) {  scheduledThreadPool.schedule(new Runnable() {  public void run() {  System.out.println("delay 3 seconds");  }  }, 3, TimeUnit.SECONDS);  }  } |

（4）newSingleThreadExecutor

|  |
| --- |
| public static void main(String[] args) {  // 按顺序来执行线程任务 但是不同于单线程，这个线程池只是只能存在一个线程，这个线程死后另外一个线程会补上。  ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();  for (int i = 0; i < 10; i++) {  final int index = i;  singleThreadExecutor.execute(new Runnable() {  public void run() {  System.out.println(Thread.currentThread().getName() + "----" + index);  }  });  }  } |

### 线程池的核心参数

通过线程池创建的线程执行完毕之后并不会销毁，而是会回到线程池继续重复利用，执行其他任务。

1核心参数

corePoolSize（核心线程数）

1）核心线程会一直存在，即使没有任务执行；

2）当线程数小于核心线程数的时候，即使有空闲线程，也会一直创建线程直到达到核心线程数；

3）设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭。

queueCapacity（任务队列容量）

也叫阻塞队列，当核心线程都在运行，此时再有任务进来，会进入任务队列，排队等待线程执行。

maxPoolSize（最大线程数）

1）线程池里允许存在的最大线程数量；

2）当任务队列已满，且线程数量大于等于核心线程数时，会创建新的线程执行任务；

3）线程池里允许存在的最大线程数量。当任务队列已满，且线程数量大于等于核心线程数时，会创建新的线程执行任务。

keepAliveTime（线程空闲时间）

1）当线程空闲时间达到keepAliveTime时，线程会退出（关闭），直到线程数等于核心线程数；

2）如果设置了allowCoreThreadTimeout=true，则线程会退出直到线程数等于零。

allowCoreThreadTimeout（允许核心线程超时）

rejectedExecutionHandler（任务拒绝处理器）

1）当线程数量达到最大线程数，且任务队列已满时，会拒绝任务；

2）调用线程池shutdown()方法后，会等待执行完线程池的任务之后，再shutdown()。如果在调用了shutdown()方法和线程池真正shutdown()之间提交任务，会拒绝新任务。

2线程池参数默认值

corePoolSize = 1

queueCapacity = Integer.MAX\_VALUE

maxPoolSize = Integer.MAX\_VALUE

keepAliveTime = 60秒

allowCoreThreadTimeout = false

rejectedExecutionHandler = AbortPolicy()

### 线程池大小应该怎么确定？

对于线程池的大小不能过大，也不能过小。过大会有大量的线程在相对较少的CPU和内存上竞争，过小又会导致空闲的处理器无法工作，浪费资源，降低吞吐率。

对于线程池大小的设定，我们需要考虑的问题有：

* CPU个数；
* 内存大小；
* 任务类型，是计算密集型（CPU密集型）还是I/O密集型；
* 是否需要一些稀缺资源，像数据库连接这种等等。

有种简单的估算方式，设N为CPU个数：

* 对于CPU密集型的应用，线程池的大小设置为N+1。
* 对于I/O密集型的应用，线程池的大小设置为2N+1 。

这种设置方式适合于一台机器上的应用的类型是单一的，并且只有一个线程池，实际情况还需要根据实际的应用进行验证。（Web 服务容器中，可以通过 Filter 来拦截获取该请求前后消耗的时间、CPU 计算时间）；

### 线程池（ThreadPoolExecutor）执行任务的过程

作者：西瓜171  
链接：<https://www.nowcoder.com/discuss/231011?type=post&order=time&pos=&page=1>  
来源：牛客网  
第一个怎么答，1000个线程  
我答的是在线程池中的线程数少于corePoolSize之前，直接创建线程来执行任务，如果是用我们项目中的ArrayBlockingQueue就是将剩余的没有超过队列长度的任务放入队列中，如果队列慢了就创建maxPoolSize-corePoolSize数量的线程来执行任务，如果还是不够就执行拒绝策略。

* 当线程数小于核心线程数时，会一直创建线程，直到线程数等于核心线程的数量；
* 当线程数等于核心线程数时，新加入的任务会被放到任务队列中等待执行；
* 当任务队列满了，又有新的任务时，此时会继续创建线程，直到线程数量等于线程池允许的最大的线程数量；
* 当线程数等于最大线程数，而且任务队列也满的时候，线程池只能将新提交的任务交给饱和策略来处理了。在线程池的创建的时候，有几个参数需要传进去，new ThreadPoolExecutor(

corePoolSize,maximumPoolSize,keepAliveTime,milliseconds,runnableTaskQueue,handler);其中最后一个参数handler就是饱和策略，java线程池框架提供的4种可用的策略：

* + - AbortPolicy：直接抛出异常；
    - CallerRunsPolicy：只用调用者所在线程来运行任务；
    - DiscardOldestPolicy：丢弃队列中最近的一个任务，并执行当前任务；
    - DiscardPolicy：不处理，直接丢弃掉；

### 线程池都有哪些状态?

1. RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
2. SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
3. STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
4. TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
5. TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

### 确保N个线程可以访问N个资源同时又不导致死锁？

使用多线程的时候， 一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。

* 1.加锁顺序（线程按照一定的顺序加锁）
* 2.加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）
* 3.死锁检测（主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景）

参考网址<https://blog.csdn.net/ls5718/article/details/51896159>

### ThreadLocal

ThreadLocal提供了线程的局部变量，每个线程都可以通过set()和get()来对这个局部变量进行操作，但不会和其他线程的局部变量进行冲突，实现了线程的数据隔离。简要言之：往ThreadLocal中填充的变量属于当前线程，该变量对其他线程而言是隔离的。

（1）作用

* 管理Connection：最典型的是管理数据库的Connection，在学习JDBC时，为了使用方便需要创建一个简单的数据库连接池，将数据库连接池交给ThreadLocal管理，能够实现当前线程的操作都是由同一个Connection，保证了事务。
* 避免一些参数传递：浏览器就相当于我们的ThreadLocal，它仅仅会发送我们当前浏览器存在的Cookie(ThreadLocal的局部变量)，不同的浏览器对Cookie是隔离的(Chrome,Opera,IE的Cookie是隔离的【在Chrome登陆了，在IE你也得重新登陆】)，同样地，线程之间ThreadLocal变量也是隔离的…。

（2）实现

* 每个Thread维护着一个ThreadLocalMap的引用；
* ThreadLocalMap是ThreadLocal的内部类，用Entry来进行存储；
* 调用ThreadLocal的set()方法时，实际上就是往ThreadLocalMap设置值，key是ThreadLocal对象，值是传递进来的对象；
* 调用ThreadLocal的get()方法时，实际上就是往ThreadLocalMap获取值，key是ThreadLocal对象；
* ThreadLocal本身并不存储值，它只是作为一个key来让线程从ThreadLocalMap获取value。

正因为这个原理，所以ThreadLocal能够实现“数据隔离”，获取当前线程的局部变量值，不受其他线程影响。

### 公平锁和非公平锁

**公平锁**：表示线程获取锁的顺序是按照加锁的顺序来分配的，及先来先得，先进先出的顺序。  
**非公平锁**：表示获取锁的抢占机制，是随机获取锁的，和公平锁不一样的就是先来的不一定能拿到锁，  
有可能一直拿不到锁，所以结果不公平

### 3.3.24 CountDownLatch的使用

CountDownLatch 的作用是：当一个线程需要另外一个或多个线程完成后，再开始执行。比如主线程要等待一个子线程完成环境相关配置的加载工作，主线程才继续执行，就可以利用 CountDownLatch 来实现。

，首先实例化一个 CountDownLatch ，参数可以理解为一个计数器，这里为 1，然后主线程执行，调用 worker 子线程，接着调用 CountDownLatch 的 await() 方法，表示阻塞主线程。当子线程执行完成后，在 finnaly 块调用 countDown() 方法，表示一个等待已经完成，把计数器减一，直到减为 0，主线程又开始执行。

### 3.3.25 AQS

QS定义了一套多线程访问共享资源的同步器框架. 许多同步类实现都依赖于它，如常用的ReentrantLock/ReentrantReadWriterLock/CountDownLatch  
这些类里面都维护了一套aqs的子类，利用子类实现的功能。该队列里面维护的是一堆线程节点。  
核心思想是当前线程获取锁的时候如果失败了，就被加入到阻塞队列中（fifo双向队列）配合锁一起使用的。  
主要关注getstate().setstate(),compareandsetstate(int expect,int update)  
当状态值为0的时候说明没有线程获取锁，否则有线程获取锁，在此当中在判断是不是当前线程，（可重入锁的特性）  
可重入锁可以设置公平锁和非公平锁，  
非公平锁需要进行2次的cas然后在进入队列，公平锁是看一下对列里面是否有排队的线程，有的话直接添加到队列的尾部排队。  
CountDownLatch 调用 await() 方法时，先去获取 state 的值，当计数器不为0的时候，说明还有需要等待的线程在运行，则调用 doAcquireSharedInterruptibly 方法，进来执行的第一个动作就是尝试加入等待队列 ，即调用 addWaiter（）方法，await() 方法是线程阻塞，直到计数器为0，才会启动；

### 3.3.26 线程的队列有几种

1、ArrayBlockingQueue

是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2、LinkedBlockingQueue

一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。

静态工厂方法Executors.newFixedThreadPool()使用了这个队列

3、SynchronousQueue

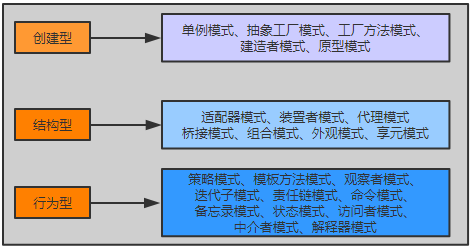
一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool（5）使用了这个队列。

4、PriorityBlockingQueue

一个具有优先级的无限阻塞队列

## 设计模式

### 三种设计模式



### 设计模式遵循6大原则

* 开闭原则：对扩展开放，对修改关闭。
* 里氏代换原则：只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。
* 依赖倒转原则：这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。
* 接口隔离原则：使用多个隔离的借口来降低耦合度。
* 迪米特法则（最少知道原则）：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。
* 合成复用原则：原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

### 单例模式

属于创建性模式，这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建，这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。主要解决“一个全局使用的类频繁的创建和销毁”的问题。

注意：单例类只能有一个实例；单例类必须自己创建自己的实例；单例类必须给其他对象提供这一实例；

* 优点：
  + 在单例模式中，活动的单例只有一个实例，对单例类的所有实例化得到的都是相同的一个实例。这样就防止其它对象对自己的实例化，确保所有的对象都访问一个实例。
  + 在一个内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。
  + 避免对共享资源的多重占用（比如写文件操作）。
* 缺点：
  + 没有接口，不能继承，与单一职责原则（一个类应该只关心内部逻辑，而不关心外面怎么样来实例化）冲突。
  + 不适用于变化的对象，如果同一类型的对象总是要在不同的用例场景发生变化，单例就会引起数据的错误，不能保存彼此的状态。
* 使用场景：
  + 网站的计数器，一般也是采用单例模式实现，否则难以同步。
  + 应用程序的日志应用，一般都何用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
  + Web应用的配置文件的读取，一般也应用单例模式，这个是由于配置文件是共享的资源。
  + 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。数据库软件系统中使用数据库连接池，主要是节省打开或者关闭数据库连接所引起的效率损耗，这种效率上的损耗还是非常昂贵的，因为何用单例模式来维护，就可以大大降低这种损耗。
  + 多线程的线程池的设计一般也是采用单例模式，这是由于线程池要方便对池中的线程进行控制。
  + 操作系统的文件系统，也是大的单例模式实现的具体例子，一个操作系统只能有一个文件系统。
* 单例模式实现：

懒汉模式：（在需要的时候才去创建单例，lazyLoading）

|  |
| --- |
| /\*\*线程不安全的懒汉模式\*/  public class SingletonDemo {  private static SingletonDemo instance;  private SingletonDemo(){ }  public static SingletonDemo getInstance(){  if(instance==null){  instance=new SingletonDemo();  }  return instance;  }  }  /\*\*线程安全的懒汉模式\*/  public class SingletonDemo {  private static SingletonDemo instance;  private SingletonDemo(){ }  public static synchronized SingletonDemo getInstance(){  if(instance==null){  instance=new SingletonDemo();  }  return instance;  }  }  优缺点：资源利用率高，不执行getInstance()就不会被实例化，可以执行该类的其他静态方法； |

饿汉模式：（单例实例在类装载时就构建）

|  |
| --- |
| public class SingletonDemo {  private static SingletonDemo instance=new SingletonDemo();  private SingletonDemo(){ }  public static SingletonDemo getInstance(){  return instance;  }  }  优缺点：  1.线程安全；  2.在类加载的同时已经创建好一个静态对象，调用时反应速度快；  3资源效率不高，可能getInstance()永远不会执行到，但执行该类的其他静态方法或者加载了该类（class.forName)，那么这个实例仍然初始化； |

双重校验锁法：（线程安全的，lazyLoading）

|  |
| --- |
| public class SingletonDemo {  private static SingletonDemo instance;  private SingletonDemo(){  System.out.println("Singleton has loaded");  }  public static SingletonDemo getInstance(){  if(instance==null){  synchronized (SingletonDemo.class){  if(instance==null){  instance=new SingletonDemo();  }  }  }  return instance;  }  }  STEP 1. 线程A访问getInstance()方法，因为单例还没有实例化，所以进入了锁定块。  STEP 2. 线程B访问getInstance()方法，因为单例还没有实例化，得以访问接下来代码块，而接下来代码块已经被线程1锁定。  STEP 3. 线程A进入下一判断，因为单例还没有实例化，所以进行单例实例化，成功实例化后退出代码块，解除锁定。  STEP 4. 线程B进入接下来代码块，锁定线程，进入下一判断，因为已经实例化，退出代码块，解除锁定。  STEP 5. 线程A初始化并获取到了单例实例并返回，线程B获取了在线程A中初始化的单例。 |

静态类内部加载：（线程安全 lazyLoading）

|  |
| --- |
| public class SingletonDemo {  private static class SingletonHolder{  private static SingletonDemo instance=new SingletonDemo();  }  private SingletonDemo(){  System.out.println("Singleton has loaded");  }  public static SingletonDemo getInstance(){  return SingletonHolder.instance;  }  } |

### 工厂模式

常用的工厂模式是静态工厂，利用static方法，作为一种类似于常见的工具类Utils等辅助效果，一般情况下工厂类不需要实例化。

|  |
| --- |
| interface food{}  class A implements food{ }  class B implements food{ }  class C implements food{ }  public class StaticFactory {  private StaticFactory(){ }  public static food getA(){ return new A(); }  public static food getB(){ return new B(); }  public static food getC(){ return new C(); }  }  class Client{  //客户端代码只需要将相应的参数传入即可得到对象  //用户不需要了解工厂类内部的逻辑。  public void get(String name){  food x = null ;  if ( name.equals("A")) {  x = StaticFactory.getA();  }else if ( name.equals("B")){  x = StaticFactory.getB();  }else {  x = StaticFactory.getC();  }  }  } |

### 抽象工厂模式

一个基础接口定义了功能，每个实现接口的子类就是产品，然后定义一个工厂接口，实现了工厂接口的就是工厂，这时候，接口编程的优点就出现了，我们可以新增产品类（只需要实现产品接口），只需要同时新增一个工厂类，客户端就可以轻松调用新产品的代码。

抽象工厂的灵活性就体现在这里，无需改动原有的代码，毕竟对于客户端来说，静态工厂模式在不改动StaticFactory类的代码时无法新增产品，如果采用了抽象工厂模式，就可以轻松的新增拓展类。

|  |
| --- |
| interface food{ }  class A implements food{ }  class B implements food{ }  interface produce{ food get();}  class FactoryForA implements produce{  @Override  public food get() {  return new A();  }  }  class FactoryForB implements produce{  @Override  public food get() {  return new B();  }  }  public class AbstractFactory {  public void ClientCode(String name){  food x= new FactoryForA().get();  x = new FactoryForB().get();  }  } |

### 原型模式

将一个对象作为原型，使用clone()方法来创建新的实例

### 适配器模式

适配器模式的作用就是在原来的类上提供新功能。适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要可分为3种：

|  |
| --- |
| 1类适配：创建新类，继承源类，并实现新接口，例如  class adapter extends oldClass implements newFunc{ }  2对象适配：创建新类持源类的实例，并实现新接口，例如  class adapter implements newFunc { private oldClass oldInstance ;}  3接口适配：创建新的抽象类实现旧接口方法。例如  abstract class adapter implements oldClassFunc { void newFunc();} |

### 装饰者模式

给一类对象增加新的功能，装饰方法与具体的内部逻辑无关。虽然适配器模式和装饰器模式的结构类似，但是每种模式的出现意图不同。适配器模式被用于桥接两个接口，而装饰模式的目的是在不修改类的情况下给类增加新的功能。例如：

|  |
| --- |
| interface Source{ void method();}  public class Decorator implements Source{  private Source source ;  public void decotate1(){  System.out.println("decorate");  }  @Override  public void method() {  decotate1();  source.method();  }  } |

### 代理模式

客户端通过代理类访问，代理类实现具体的实现细节，客户只需要使用代理类即可实现操作。这种模式可以对旧功能进行代理，用一个代理类调用原有的方法，且对产生的结果进行控制。

|  |
| --- |
| interface Source{ void method();}  class OldClass implements Source{  @Override  public void method() {  }  }  class Proxy implements Source{  private Source source = new OldClass();  void doSomething(){}  @Override  public void method() {  new Class1().Func1();  source.method();  new Class2().Func2();  doSomething();  }  } |

### 享元模式

享元模式通过共享对象来避免创建太多的对象。为了使用享元模式，你需要确保你的对象是不可变的，这样你才能安全的共享。JDK 中 String 池、Integer 池以及 Long 池都是很好的使用了享元模式的例子。

### 装饰者模式和代理模式的区别？

两个设计模式看起来很像。对装饰者模式来说，装饰者（decorator）和被装饰者（decoratee）都实现同一个接口。对代理模式来说，代理类（proxy class）和真实处理的类（real class）都实现同一个接口。此外，不论我们使用哪一个模式，都可以很容易地在真实对象的方法前面或者后面加上自定义的方法。

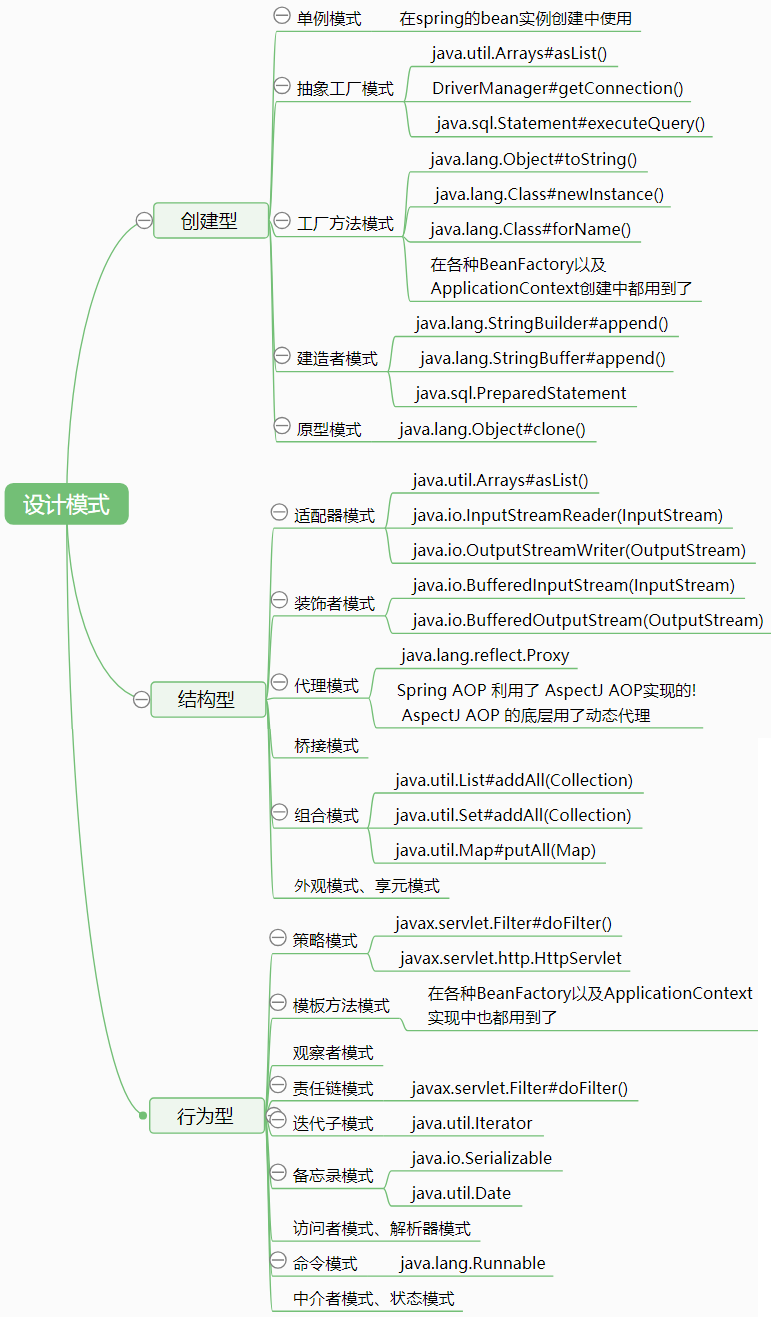
实际上，在装饰者模式和代理模式之间还是有很多差别的：

* 装饰者模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。
* 当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰者模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

总结：

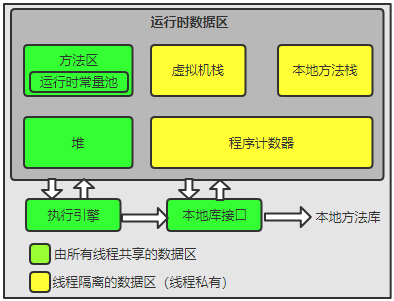
使用代理模式，代理和真实对象之间的的关系通常在编译时就已经确定了，而装饰者能够在运行时递归地被构造。

### 设计模式的使用



# Java虚拟机

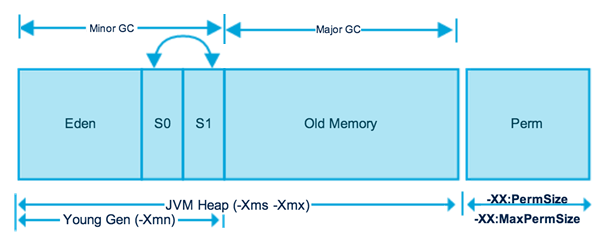
## JVM内存模型（区域）



* 程序计数器：占用内存空间小，通过改变这个计数器的值来选取下一条需要执行的字节码指令，唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域；
* 虚拟机栈：每个方法在执行的同时都会创建一个栈桢(Stack Frame)用于存储局部变量，操作数栈，动态链接，方法出口等信息；局部变量表存放了编译期间可知的各种基本数据类，对象引用类型，它不等同于对象本身和returnAddress类型(一条指向字节码指令的地址)。
* 本地方法栈：为虚拟机使用到的Native方法服务。
* 堆：在虚拟机启动时创建；所有的对象实例和数组都要在堆上分配。（大小由-Xmx和-Xms控制）
* 方法区：存储已被虚拟机加载的类信息，常量，静态变量，即时编译器便后的代码等数据。
* 运行时常量池：Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池(Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。这部分内容将在类加载后进入方法区的运行时常量池中存放。具备动态性，运行期间也可能将新常量放入池中比如String类的intern()方法。
* 直接内存：不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域；NIO(New Input/Output)类，引入了一种基于通道(Channel)与缓冲区(Buffer)de I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作，来避免在Java堆和Native堆中来回复制数据。直接内存的分配不会受到Java堆大小的限制，但是会受到本机总内存大小以及处理器寻址空间的限制，因此也可能导致OutOfMemoryError。

## JAVA堆内存参数

* Java堆（Java Heap）是java虚拟机所管理的内存中最大的一块；
* java堆被所有线程共享的一块内存区域；
* 虚拟机启动时创建java堆；
* java堆的唯一目的就是存放对象实例；
* java堆是垃圾收集器管理的主要区域；
* 从内存回收的角度来看，由于现在收集器基本都采用分代收集算法，所以Java堆可以细分为：新生代（Young）和老年代（Old）。新生代又被划分为三个区域Eden、From Survivor，To Survivor等。无论怎么划分，最终存储的都是实例对象，进一步划分的目的是为了更好的回收内存，或者更快的分配内存；
* java堆的大小是可扩展的，通过-Xmx和-Xms控制；
* 如果堆内存不够分配实例对象， 并且对也无法在扩展时，将会抛出outOfMemoryError异常。



对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。特别要关注Full GC，因为它会对整个堆进行整理，导致Full GC一般由于以下几种情况：

Young Gen：

* 1、绝大多数新建的Object 被放在Eden Memory
* 2、如果 Eden Memory 内存满了，则进行GC操作。同时把未被GC的Object移动到S0或S1中。此时 Minor GC 也会检查和移动 S0 和 S1 中的对象。最后使 S0，S1 其中一个置为空。
* 3、多次 GC 后仍然未被 GC 的 Object 将被移动到 Old Gen 内存区中。通常 Object 会被 GC 设定一个轮询的阀值。

Old Gen：

* Old Gen 内存区存放了经过多次 Minor GC 后仍然不能被 GC 的 Object。与 Young Gen 相同，当 Old Gen 区满了之后将执行 GC 操作，该操作称为：Full GC(Major GC)。耗用的时间也相对较长。
* Young Gen 和 Old Gen都可以主动触发stop-the-world事件，挂起所有任务，执行GC操作。被挂起的任务只有在GC执行完毕后，才会恢复执行。多数情况下，GC性能调优（GC tuning）就是指降低 stop-the-world 时 GC 执行的时间。

Perm Gen：

* 存放应用程序的元数据，用来描述类及其方法的原始信息。注意：Perm Gen 不是 Heap 的一部分。
* Perm Gen 被 JVM 使用于应用程序运行期间（runtime），基于应用所使用到的类。Perm Gen 中同时包括 Java SE 包中的类。
* Perm Gen 只有在执行 Full GC 时才会被 GC。

### 4.2.2 jVm 内存调优

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

1). 新生代设置过小

  一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2). 新生代设置过大

  一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加

  一般说来新生代占整个堆1/3比较合适

3). Survivor设置过小

  导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4). Survivor设置过大

  导致eden过小，增加了GC频率

  另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收

1. 由内存管理和垃圾回收可知新生代和旧生代都有多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，JVM提供两种较为简单的GC策略的设置方式

## 怎样判断对象是否会被回收？

1. 引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。JVM里面并没有选用引用计数算法来管理内存，主要原因是它很难解决对象之间相互循环引用的问题。

1. 可达性分析算法（JVM使用的）

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（也就是从GC Roots到这个对象不可达时），则证明此对象是不可用的。

在java语言中，可以作为GC Roots的对象包括以下几种：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象；
2. 方法区中类静态属性引用的对象；
3. 方法区中常量引用的对象；
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象；

## 垃圾回收算法

（1）标记-清除算法

最基础的垃圾收集器算法，分为“标记”和“清除”两个阶段，先标记处所需要回收的对象，标记完成后，统一回收掉所有被标记的对象。

缺点：

* 效率问题，标记和清除的效率不高。
* 清除后会产生大量的不连续的内存碎片，可能会导致在程序需要为较大对象分配内存时无法找到足够连续的内存，不得不提前触发垃圾收集动作。

（2）复制算法（Serial收集器）

将内存容量分成大小相等的两块，每次只使用其中一块，当一块用完时，将还存活的对象复制到另一块去，然后把之前使用满的那块空间一次性清理掉，如此反复。

* 优点：内存分配的时候不用考虑内存碎片问题，只移动堆顶指针，按顺序分配即可，简单高效。
* 缺点：内存空间浪费大，每次只能使用当前内存空间的一半；当对象存活率较高时，需要有大量的复制操作，效率低。

（3）标记-整理（Serial Old收集器）

标记-整理 是在 标记-清除 基础上改进得来。有标记阶段，标记出所有需要回收的对象，但是不会直接清理，而是将存活的对象向一端移动，在移动过程中清理掉可回收对象。这样就解决了内存碎片的问题。

* 优点：解决了之前内存碎片的问题，特别是在存活率高的时候，效率远高于复制算法。

（4）分代收集算法

根据内存对象的存活周期不同，将内存划分成几块，java虚拟机中一般将内存划分成新生代和老年代，当新建对象时一般在新生代中分配内存，在新生代垃圾收集器回收几次后仍然存活的对象，将被移动到老年代，或者当大的对象在新生代中无法分配到足够连续的内存空间时也会直接分配到老年代。

## 新生代对象转移到老年代的触发条件？

(1)、Eden区满时，进行Minor GC，当Eden和一个Survivor区中依然存活的对象无法放入到Survivor中，则通过分配担保机制提前转移到老年代中。

(2)、若对象体积太大，新生代无法容纳这个对象，就会绕过新生代，直接在老年代分配。

(3)、长期存活的对象将进入老年代。

* + 虚拟机对每个对象定义了一个对象年龄（Age）计数器。当年龄增加到一定的临界值时，就会晋升到老年代中。如果对象在Eden出生并在第一次发生MinorGC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；以后每经历一次Minor GC，该对象还存活的话Age=Age+1。

(4)、动态对象年龄判定。

虚拟机并不总是要求对象的年龄必须达到设定的年龄（MaxTenuringThreshold）才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

## Minor GC和Full GC

1、Minor GC触发机制：

* + 当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。

2、Full GC触发机制：

* + 调用System.gc时，系统建议执行Full GC，但是不必然执行；
  + 年轻带晋升到老年带没有足够的连续空间（很有可能是内存碎片导致的）；
  + 方法区空间不足；
  + 通过Minor GC后进入老年代的平均大小 大于老年代的可用内存；
  + 由Eden区、From survivor区向To survivor区复制时，对象大小大于To survivor区的可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小；

## 垃圾回收器

1. CMS收集器

CMS (Concurrent Mark Swep)收集器是一个比较重要的回收器，现在应用非常广泛，CMS是一种获取最短回收停顿时间为目标的收集器，这使得它很适合用于和用户交互的业务。从名字(Mark Swep)就可以看出，CMS收集器是基于标记清除算法实现的（用于老年代）。它的收集过程分为四个步骤：

* 初始标记(initial mark)：独占PUC，仅标记GCroots能直接关联的对象；
* 并发标记(concurrent mark)：可以和用户线程并行执行，标记所有可达对象；
* 重新标记(remark)：独占CPU(STW)，对并发标记阶段用户线程运行产生的垃圾对象进行标记修正；
* 并发清除(concurrent sweep)：可以和用户线程并行执行，清理垃圾；

注意初始标记和重新标记还是会stop the world，但是在耗费时间更长的并发标记和并发清除两个阶段都可以和用户进程同时工作。

优点：并发，低停顿

缺点：

* + 对CPU非常敏感：在并发阶段虽然不会导致用户线程停顿，但是会因为占用了一部分线程使应用程序变慢；
  + 无法处理浮动垃圾：在最后一步并发清理过程中，用户线程执行也会产生垃圾，但是这部分垃圾是在标记之后，所以只有等到下一次gc的时候清理掉，这部分垃圾叫浮动垃圾；
  + CMS使用“标记-清除”法会产生大量的空间碎片，当碎片过多，将会给大对象空间的分配带来很大的麻烦，往往会出现老年代还有很大的空间但无法找到足够大的连续空间来分配当前对象，不得不提前触发一次FullGC，为了解决这个问题CMS提供了一个开关参数，用于在CMS顶不住，要进行FullGC时开启内存碎片的合并整理过程，但是内存整理的过程是无法并发的，空间碎片没有了但是停顿时间变长了；

1. Parallel Scavenge + Parallel Old（JDK1.8使用的垃圾收集器）

* Parallel Scavenge：是新生代收集器，采用复制算法的收集器，又是并行的多线程收集器。它的目标是达到一个可控制的吞吐量。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，Parallel Scavenge就可以实现这种自适应调整策略。
* Parallel Old：是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

1. G1收集器（JDK9）

G1收集器是一款面向服务端应用的垃圾收集器。HotSpot团队赋予它的使命是在未来替换掉 JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点：

* 并行与并发：G1能更充分的利用CPU、多核环境下的硬件优势来缩短stop the world的停顿时间。
* 分代收集：和其他收集器一样，分代的概念在G1中依然存在，不过G1不需要其他的垃圾回收器的配合就可以独自管理整个GC堆，可以自己独自管理新生代和老年代。
* 空间整合：由于G1使用了独立区域（Region）概念，G1从整体来看是基于“标记-整理”算法实现收集，从局部（两个Region）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片。
* 可预测的非停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N毫秒。

与其它收集器相比，G1变化较大的是它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留了新生代和来年代的概念，但新生代和老年代不再是物理隔离的了它们都是一部分Region（不需要连续）的集合。同时，为了避免全堆扫描，G1使用了Remembered Set来管理相关的对象引用信息。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏了。

## java四种引用类型？使用场景？

Java将引用分为了：强引用、软引用、弱引用、虚引用4 种，这4种引用的强度依次减弱。

（1）强引用

Java中默认声明的就是强引用，比如：

|  |
| --- |
| Object obj = new Object(); //只要obj还指向Object对象，Object对象就不会被回收  obj = null; //手动置null |

只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

（2）软引用

软引用是用来描述一些非必需但仍有用的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。

使用场景：

|  |
| --- |
| 比如在一个博客管理系统里，为了提升访问性能，在用户在点击博文时，如果这篇博文没有缓存到内存中，则需要做缓存动作，这样其它用户在点击同样这篇文章时，就能直接从内存里装载，而不用走数据库，这样能降低响应时间。我们可以通过数据库级别的缓存在做到这点，这里也可以通过软引用来实现，具体的实现步骤如下：   * 第一，可以通过定义Content类来封装博文的内容，其中可以包括ID、内容、引用图片等相关信息。 * 第二，可以定义一个类型为HashMap<String, SoftReference<Content>>的对象类保存缓存内容，其中键是String类型，表示文章ID，值是指向Content的软引用。 * 第三，当用户点击某个ID的文章时，根据ID到第二步定义的HashMap里去找，如果找到，而且所对应的SoftReference<Content>值内容不是null，则直接从这里拿数据并做展示动作，这样不用走数据库，可以提升性能。 * 第四，如果用户点击的某个文章的ID在HashMap里找不到，或者找到了，但对应的值内容是空，那么就从数据库去找，找到后显示这个文章，同时再把它插入到HashMap里，这里请注意，显示后需要撤销掉这个Content类型对象上的强引用，保证它上面只有一个软引用。   来分析下用软引用有什么好处？假设我们用1个G的空间缓存了10000篇文章，这10000篇文章所占的内存空间上只有软引用。如果内存空间足够，那么我们可以通过缓存来提升性能，但万一内存空间不够，我们可以依次释放这10000篇文章所占的1G内存，释放后不会影响业务流程，最多就是降低些性能。  对比一下，如果不用软应用，而是用强引用来缓存，由于不知道文章何时将被点击，所以不知道什么时候可以撤销这些文章对象上的强引用，或者即使我们引入了一套缓存淘汰流程，但这就是额外的工作了，这就没刚才使用“软引用“那样方便了。 |

（3）弱引用

弱引用的引用强度比软引用要更弱一些，无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收。用 java.lang.ref.WeakReference 来表示弱引用。WeakHashMap类同样也利用了弱引用。

|  |  |
| --- | --- |
| private static void testWeakReference() {  for (int i = 0; i < 10; i++) {  byte[] buff = new byte[1024 \* 1024];//buff是强引用  WeakReference<byte[]> sr = new WeakReference<>(buff);  list.add(sr);  }    System.gc(); //主动通知垃圾回收  //后面的程序没有再次直接使用buff对象，使用的是sr对象  for(int i=0; i < list.size(); i++){  Object obj = ((WeakReference) list.get(i)).get();  System.out.println(obj);  }  } | 打印结果：    可以发现所有被弱引用关联的对象都被垃圾回收了。 |

使用场景：

|  |
| --- |
| 比如在某个电商网站项目里，我们会用Coupan这个类来保存优惠券信息，在其中我们可以定义优惠券的打折程度，有效日期和所作用的商品范围等信息。当我们从数据库里得到所有的优惠券信息后，会用一个List<Coupan>类型的coupanList对象来存储所有优惠券。  可以用WeakHashMap<Coupan, <List<WeakReference <User>>>类型的weakCoupanHM对象保存一个优惠券及它所关联的所有用户。其中它的键是Coupan类型，值是指向List<User>用户列表的弱引用。  如果有100个优惠券，那么它们会存储于List<Coupan>类型的coupanList，同时，WeakHashMap<Coupan, <List<WeakReference <User>>>类型的weakCoupanHM对象会以键的形式存储这100个优惠券。而且，如果有1万个用户，那么我们可以用List<User>类型的userList对象来保存它们，假设coupan1这张优惠券对应着100个用户，那么我们一定会通过如下的代码存入这种键值对关系，weakCoupanHM.put(coupan1,weakUserList);，其中weakUserList里以弱引用的方式保存coupan1所对应的100个用户。  这样的话，一旦当优惠券或用户发生变更，它们的对应关系就能自动地更新，具体表现如下。  1 当某个优惠券（假设对应于coupan2对象）失效时，我们可以从coupanList里去除该对象，coupan2上就没有强引用了，只有weakCoupanHM对该对象还有个弱引用，这样coupan2对象能在下次垃圾回收时被回收，从而weakCoupanHM里就看不到了。  2 假设某个优惠券coupan3用弱引用的方式指向于100个用户，当某个用户（假设user1）注销账号时，它会被从List<User>类型的userList对象中被移除。这时该对象上只有weakCoupanHM里的值（也就是<List<WeakReference <User>>）这个弱引用，该对象同样能在下次垃圾回收时被回收，这样coupan3的关联用户就会自动地更新为99个。  如果不用弱引用，而是用常规的HashMap<Coupan,List<User>>来保存对应关系的话，那么一旦出现优惠券或用户的变更的话，那么我们就不得不手动地更新这个表示对应关系的HashMap对象了，这样，代码就会变得复杂，而且很有可能因疏忽而忘记在某个位置添加更新代码。相比之下，弱引用给我们带来的“自动更新“就能给我们带来很大的便利。 |

弱引用非常适合存储元数据，例如：存储ClassLoader引用。如果ClassLoader还可达，那么我们可以获取到该ClassLoader的数据以及相关数据，但如果ClassLoader不可达了，那么只有弱引用的ClassLoader就会被自动回收。具体比如JDK的Proxy类中有一个WeakCache，WeakCache类似一个map的结构，它的key是ClassLoader，value是所有跟该ClassLoader相关的动态类，当WeakCache中的ClassLoader对象不可达了，那么所有相关数据就会从WeakCache中淘汰掉。

（4）虚引用

PhantomReference类中只有一个方法 get()，只是返回null。而且这个类只有一个构造器(软引用和弱引用均有两个构造器)：也就是说，虚引用只能与 ReferenceQueue一起使用。如果一个对象仅有虚引用，那么它就像没有任何引用一样，在任何时候都可能被 gc 回收。幽灵引用主要用来跟踪对象被垃圾回收的活动。

（5）总结

* 软引用、弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果所引用的对象被垃圾回收器回收，Java虚拟机就会把软引用或弱引用加入到与之关联的引用队列中。
* 虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

## 类加载器双亲委派机制

* 什么是双亲委派模型(Parent-Delegation Model)？

JVM中加载类机制采用的是双亲委派模型：当一个类加载器收到一个类加载的请求，它首先会将该请求委派给父类加载器去加载，每一个层次的类加载器都是如此，因此所有的类加载请求最终都应该被传入到顶层的启动类加载器(Bootstrap ClassLoader)中，只有当父类加载器反馈无法完成这个类的加载请求时（它的搜索范围内不存在这个类），子类加载器才尝试加载。



* 启动类加载器：负责加载虚拟机的核心类库。
* 扩展类加载器：负责加载 JAVA\_HOME\lib\ext 目录中的类库。
* 应用程序类加载器：负责加载用户路径（classpath）上的类库。
* 用户自定义的类加载器：1. java.lang.ClassLoader的子类。2. 用户可以定制类的加载方式。
* 为什么使用双亲委派模型？
* 1可以避免重复加载，父类已经加载了，子类就不需要再次加载；
* 2更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心api，会带来相关隐患。

例如类java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都会委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果用户自己写了一个名为java.lang.Object的类，并放在程序的Classpath中，那系统中将会出现多个不同的Object类，java类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。

* 怎样打破双亲委派机制？

在实际的应用中双亲委派解决了java 基础类统一加载的问题，但是却着实存在着一定的缺陷，比如SPI（Service Provider Interface）代码：

|  |
| --- |
| 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。 |

1. 在JDK1.2之前，用户去继承 java.lang.ClassLoader 的唯一目的就是为了重写loadClass 方法，由于用户自己重写了loadClass，那么也就是用户自己去自定义加载类，这样就破坏了双亲委派机制；
2. JDBC的SPI加载都是父类的加载器去请求子类的加载器去加载类；

* 缓存机制

缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效。

## JVM类的加载过程

类的加载机制分为三个部分：加载、连接（验证、准备、解析）、初始化；

（1）加载：查找并加载类的二进制数据

* 通过类的全限定名来获取定义此类的二进制字节流；
* 将这个类字节流代表的静态存储结构转为方法区的运行时数据结构，在堆中生成一个代表此类的java.lang.Class对象，作为访问方法区这些数据结构的入口。
* 这个过程主要就是类加载器完成。

（2）连接

* + 验证：确保被加载的类的正确性；
  + 准备：为类的静态变量分配内存，并将其初始化为默认值；
  + 解析：把常量池内的符号引用替换为直接引用

（3）初始化：为类的静态变量赋予正确的初始值

## 内存屏障

为什么会有内存屏障？

* 每个CPU都会有自己的缓存（有的甚至L1,L2,L3），缓存的目的就是为了提高性能，避免每次都要向内存取。但是这样的弊端也很明显：不能实时的和内存发生信息交换，分在不同CPU执行的不同线程对同一个变量的缓存值不同。
* 用volatile关键字修饰变量可以解决上述问题，那么volatile是如何做到这一点的呢？那就是内存屏障，内存屏障是硬件层的概念，不同的硬件平台实现内存屏障的手段并不是一样，java通过屏蔽这些差异，统一由jvm来生成内存屏障的指令。

内存屏障是什么？

* 硬件层的内存屏障分为两种：Load Barrier 和 Store Barrier即读屏障和写屏障。
* 内存屏障有两个作用：
* 阻止屏障两侧的指令重排序；
* 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。
* 对于Load Barrier来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
* 对于Store Barrier来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

java内存屏障

* java的内存屏障通常所谓的四种即LoadLoad,StoreStore,LoadStore,StoreLoad实际上也是上述两种的组合，完成一系列的屏障和数据同步功能。
* LoadLoad屏障：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
* StoreStore屏障：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
* LoadStore屏障：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
* StoreLoad屏障：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。它的开销是四种屏障中最大的。在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能；

volatile语义中的内存屏障

* + volatile的内存屏障策略非常严格保守，非常悲观且毫无安全感的心态：
* 在每个volatile写操作前插入StoreStore屏障，在写操作后插入StoreLoad屏障；
* 在每个volatile读操作前插入LoadLoad屏障，在读操作后插入LoadStore屏障；
  + 由于内存屏障的作用，避免了volatile变量和其它指令重排序、线程之间实现了通信，使得volatile表现出了锁的特性。

final语义中的内存屏障

* + 对于final域，编译器和CPU会遵循两个排序规则：
* 新建对象过程中，构造体中对final域的初始化写入和这个对象赋值给其他引用变量，这两个操作不能重排序；
* 初次读包含final域的对象引用和读取这个final域，这两个操作不能重排序；（意思就是先赋值引用，再调用final值）
  + 总之上面规则的意思可以这样理解，必需保证一个对象的所有final域被写入完毕后才能引用和读取。这也是内存屏障的起的作用：
* 写final域：在编译器写final域完毕，构造体结束之前，会插入一个StoreStore屏障，保证前面的对final写入对其他线程/CPU可见，并阻止重排序。
* 读final域：在上述规则2中，两步操作不能重排序的机理就是在读final域前插入了LoadLoad屏障。

Java 内存模型底层实现可以简单的认为：通过内存屏障 (memory barrier)禁止重排序，即时编译器根据具体的底层 体系架构，将这些内存屏障替换成具体的 CPU 指令。对 于编译器而言，内存屏障将限制它所能做的重排序优化。 而对于处理器而言，内存屏障将会导致缓存的刷新操作。

## 内存溢出、内存泄漏

（1）内存溢出和内存泄漏的区别？

* 内存泄漏是导致内存溢出的原因之一；内存泄漏积累起来将导致内存溢出；
* 内存泄漏可以通过完善代码来避免；内存溢出可以通过调整配置来减少发生的频率，但是无法彻底避免；

（2）可造成内存溢出情况：

* 堆上没有足够的内存可以完成对象的分配，并且堆没有办法扩展的时候-->OutOfMemoryError
* 方法区无法满足内存分配需求时-->OutOfMemoryError
* 虚拟机栈（本地方法栈）扩展时无法申请足够的内存-->OutOfMemoryError

（3）可造成内存泄漏情况：

* + 程序动态的分配了内存，但在程序结束时没有释放这部分内存，导致那一部分内存不可用。
  + 当被分配的对象可达，但是已经没有作用时，比如Student s1 = new Student1();Student s2 = new Student2();然后将s1,s2放入ArrayList中后，将s1,s2设成null，但是这两个对象所占的内存并没有释放，因为ArrayList里还存放着对象的引用；

（4）如何避免内存泄漏、溢出？

* 尽早释放无用对象的引用。
* 使用临时变量的时候，让引用变量在退出活动域后自动设置为null，暗示垃圾收集器来收集该对象，防止发生内存泄露。
* 程序进行字符串处理时,尽量避免使用String，而应使用StringBuffer，因为每一个String对象都会独立占用内存一块区域。

## JAVA内存泄漏分析

在对Java内存泄漏进行分析的时候，需要对jvm运行期间的内存占用、线程执行等情况进行记录的dump文件，常用的主要有thread dump和heap dump。

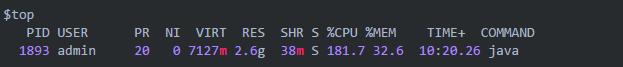
* + thread dump 主要记录JVM在某一时刻各个线程执行的情况，以栈的形式显示，是一个文本文件。通过对thread dump文件可以分析出程序的问题出现在什么地方，从而定位具体的代码然后进行修正。thread dump需要结合占用系统资源的线程id进行分析才有意义。
  + heap dump 主要记录了在某一时刻JVM堆中对象使用的情况，即某个时刻JVM堆的快照，是一个二进制文件，主要用于分析哪些对象占用了太对的堆空间，从而发现导致内存泄漏的对象。

上面两种dump文件都具有实时性，因此需要在服务器出现问题的时候生成，并且多生成几个文件，方便进行对比分析。可以看这个参考文章：<http://cmsblogs.com/?p=5155>

### CPU使用过高的线程？

（1）定位进程

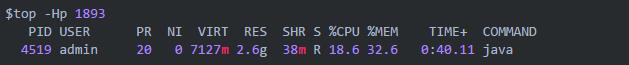
首先登陆服务器，执行top命令，查看CPU的使用情况：



通过以上命令，我们可以看到，进程ID为1893的Java进程的CPU占用率达到了181%，基本可以定位到是我们的Java应用导致整个服务器的CPU占用率飙升。

（2）定位线程

Java是单进程多线程的，那么，我们接下来看看PID=1893的这个Java进程中的各个线程的CPU使用情况，同样是用top命令：



通过top -Hp 1893命令，我们可以发现，当前1893这个进程中，ID为4519的线程占用CPU最高。

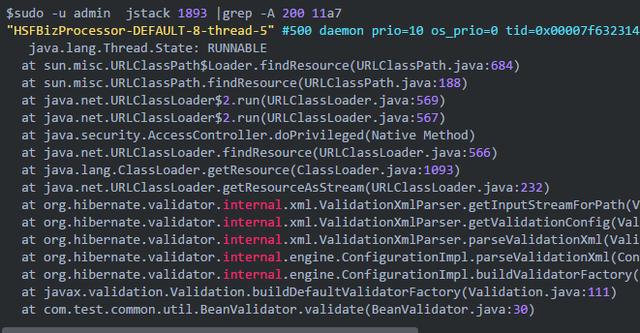
（3）定位代码

通过top命令，目前已经定位到导致CPU使用率较高的具体线程， 那么接下来就定位下到底是哪一行代码存在问题。

首先，我们需要把4519这个线程转成16进制：

|  |
| --- |
| [wsy@wwssyy ~]$ printf %x 4519  11a7 |

接下来，通过jstack（jstack <进程号> | grep –A 200 <线程ID的十六进制>）命令，查看栈信息：



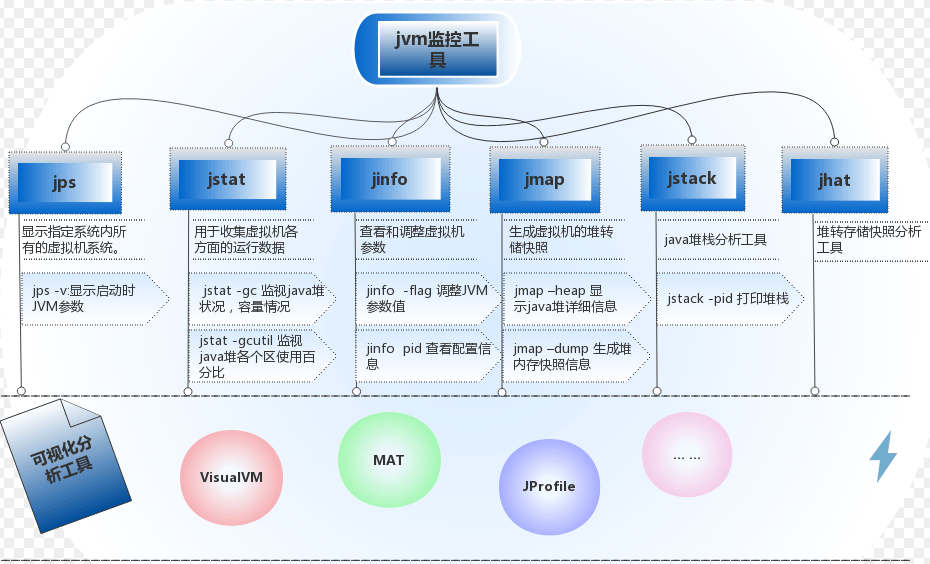
通过以上代码，我们可以清楚的看到，BeanValidator.java的第30行是有可能存在问题的。

（4）问题解决

或者dump下来线程的信息Thread dump日志文件进行分析。

### 频繁Full GC问题排查过程？

（1）JVM监控工具



新功能上线时，可能会出现Full GC次数过多的情况。对于Full GC较多的情况，其主要有如下两个特征：

* + 线上多个线程的CPU都超过了100%，通过jstack命令可以看到这些线程主要是垃圾回收线程；
  + 通过jstat命令监控GC情况，可以看到Full GC次数非常多，并且次数在不断增加。

具体的方式与CPU使用过高查看线程方式相同：

* + 1首先我们通过top命令查看当前CPU消耗过高的进程是哪个，从而得到进程id；
  + 2然后通过top -Hp <pid>来查看该进程中有哪些线程CPU过高，一般超过80%就是比较高的，80%左右是合理情况。这样我们就能得到CPU消耗比较高的线程id。
  + 3接着通过该线程id的十六进制表示，在jstack日志中查看当前线程具体的堆栈信息。
  + 4通过jstat -gcutil <进程id> <间隔时间> <显示个数> 命令查看GC情况；



* + 5dump日志文件；
    - 使用jmap生成堆内存信息，可以执行下面的命令生成Heap Dump：

|  |
| --- |
| jmap -dump:live,format=b,file=heap-dump.bin <pid> |

其中pid是JVM进程的id，heap-dump.bin是生成的文件名称，在执行命令的目录下面。

* + - 使用JConsole 生成：图像化的界面，直接点击dumpHeap 按钮生成 Heap Dump文件。
  + 6对Heap Dump文件进行分析

jhat 是JDK自带的用于分析JVM Heap Dump文件的工具，使用下面的命令可以将堆文件的分析结果以HTML网页的形式进行展示：jhat <heap-dump-file>；执行成功之后：访问 http://localhost:7000/ 就可以看到结果了。

* + 7 最后定位代码，进行分析改进。

如果通过top命令看到CPU并不高，并且系统内存占用率也比较低。此时就可以考虑是否是由于另外三种情况导致的问题。具体的可以根据具体情况分析：

* + 如果是接口调用比较耗时，并且是不定时出现，则可以通过压测的方式加大阻塞点出现的频率，从而通过jstack查看堆栈信息，找到阻塞点；
  + 如果是某个功能突然出现停滞的状况，这种情况也无法复现，此时可以通过多次导出jstack日志的方式对比哪些用户线程是一直都处于等待状态，这些线程就是可能存在问题的线程；
  + 如果通过jstack可以查看到死锁状态，则可以检查产生死锁的两个线程的具体阻塞点，从而处理相应的问题。

Tips：使用jstack还可以帮助检查死锁，并且在日志中打印具体死锁的线程的信息。

# 数据库与缓存

## MySQL

### 关系型数据库与非关系型数据库

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库类型 | 特性 | 优点 | 缺点 |
| 关系型：  SQLite、  Oracle、  mysql | 1、关系型数据库，是指采用了关系模型来组织数据的数据库；简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。  2、最大特点就是事务的一致性； | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型比网状、层次等模型来说更容易理解；  2、使用方便：通用的SQL语言使得操作关系型数据库非常方便；  3、易于维护：实体完整性、用户定义的完整性减低了数据冗余和数据不一致的概率；  4、支持SQL，可用于复杂的查询。 | 1、读写性能比较差，尤其是海量数据的高效率读写；  2、固定的表结构，灵活性欠缺；  3、高并发读写需求，磁盘IO是一个瓶颈； |
| 非关系型：  MongoDB、  redis、  HBase | 1、使用键值对存储数据；  2、分布式；  3、一般不支持ACID特性；  4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层解析，读写性能很高；  2、基于键值对，数据没有耦合性，容易扩展；  3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高；  2、无事务处理；  3、数据结构相对复杂，复杂查询方面稍欠； |

### Mysql数据库常见储存引擎？

|  |  |
| --- | --- |
| MyISAM | 默认的MySQL插件式存储引擎，它是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。 |
| InnoDB | 用于事务处理应用程序，具有众多特性，包括ACID事务支持。(提供行级锁) |
| BDB | 可替代InnoDB的事务引擎，支持COMMIT、ROLLBACK和其他事务特性。 |
| Memory | 将所有数据保存在RAM中，在需要快速查找引用和其他类似数据的环境下，可提供极快的访问。 |
| Federated | 能够将多个分离的MySQL服务器链接起来，从多个物理服务器创建一个逻辑数据库。十分适合于分布式环境或数据集市环境。 |

### Innodb引擎和MyISAM引擎

|  |  |  |
| --- | --- | --- |
| 特性 | InnoDB（5.5版本之后默认的引擎） | MyISAM |
| 事务 | 支持事务； | 不支持事务； |
| 外键 | 支持外键，外键所在的表称为子表而所依赖的表称为父表。支持奔溃后的数据的安全恢复； | 不支持外键；不支持奔溃后的安全恢复； |
| 锁 | 支持多版本并发控制的行锁；InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。 | 用户在操作myisam表时，select，update，delete，insert语句都会给表自动加锁（只支持表级锁，不支持行锁）； |
| 索引 | 5.6版本之后支持全文索引；5.7版本之后通过使用ngram插件开始支持中文；MySQL允许在char、varchar、text类型上建立全文索引 | 支持全文索引；支持延迟更新索引，极大的提升了写入性能； |
| 数据 |  | 对于不会进行修改的表，支持压缩表，减少了磁盘空间的占用； |
| 应用场景 | * 1 InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB； * 2 MyIASM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyIASM； * 3 对于一般的Web应用来说，应该选择MyIASM，效率更高，特定场景再用InnoDB； | |

### 事务的特性（ACID）

数据库事务transanction正确执行的四个基本要素。ACID：原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。

1. 原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
2. 一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。
3. 隔离性：隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆， 必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。
4. 持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

### 数据库事务的隔离级别？

* 丢失更新：一个事务的更新操作会被另一个事务的更新操作覆盖，从而导致数据的不一致。
* 脏读：一个事务可以读到另一个事务未提交的数据；
  + - 脏读与脏页不一样，脏页指的是在缓冲池中已经被修改的页，但是还没有刷新到磁盘中，即数据库实例内存中的页和磁盘中的页的数据是不一致的，当然在刷新到磁盘之前，日志都已经被写人到了重做日志文件中。对于脏页的读取，是非常正常的。脏页是因为数据库实例内存和磁盘的异步造成的，这并不影响数据的一致性（或者说两者最终会达到一致性，即当脏页都刷回到磁盘。并且因为脏页的刷新是异步的，不影响数据库的可用性，因此可以带来性能的提高。
    - 脏读也可以有用处，比如在Mysql的replication环境中的slave节点，并且在slave上的查询不需要特别精确的返回值。（replication功能将一个MySQL实例中的数据复制到另一个MySQL实例中。整个过程是异步进行的，但由于其高效的性能设计，复制的延时非常小。MySQL复制功能在实际的应用场景中被广泛的应用于保证数据系统数据的安全性和可扩展设计中。）
* 不可重复读：（**读取数据本身的对比**）一个事务对同一行数据重复读取两次，但是却得到了不同的结果。
* 幻读：（**读取结果集条数的对比**）一个事务按相同的查询条件查询之前检索过的数据，确发现检索出来的结果集条数变多或者减少，类似产生幻觉。这是因为在两次查询过程中有另外一个事务插入数据造成的。

|  |  |
| --- | --- |
| Read Uncommitted（读未提交） | （1）解决-丢失更新-问题。  （2）如果一个事务已经开始写数据，则不允许其他事务同时进行写操作，但允许其他事务读此行数据。  （3）可通过“排他锁”实现。 |
| Read Committed（读已提交） | （1）解决-丢失更新、脏读-问题。  （2）读取数据的事务允许其他事务继续访问改行数据，但是未提交的写事务将会禁止其他事务访问改行。  （3）可通过“瞬间共享读锁”和“排他锁”实现。 |
| Repeatable Read（可重复读） | （1）解决-丢失更新、脏读和不可重复读-问题。  （2）读取数据的事务将会禁止写事务，但允许读事务，写事务则禁止任何其他事务。  （3）可通过“共享读锁”和“排他锁”实现。 |
| Serializable（可串行化） | （1）提供严格的事务隔离。  （2）要求失去序列化执行，事务只能一个接一个地执行，不能并发执行。  （3）仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。 |

### Mysql解决丢失更新、脏读、不可重复读（幻读）？

（1）解决丢失更新

* 丢失更新：一个事务的更新操作会被另一个事务的更新操作覆盖，从而导致数据的不一致。

|  |
| --- |
| 1）事务1查询一行数据，放入本地内存，并显示给一个终端用户User1；  2）事务2也查询该数据，放入本地内存，并显示给一个终端用户User2；  3）User1修改这行记录，更新数据库并提交；  4）User2修改这行记录，更新数据库并提交； |

在这个过程中，User1用户的更新操作丢失了。

* 解决：要解决这个问题，需要让事务在这种情况下的操作变成串行化，而不是并行操作，在上面的4个步骤1）中，对用户读取的记录加上一个排他X锁，同样在步骤2）中，也加一个排他X锁，通过这种方式，步骤2）就必须等待步骤1）和步骤3）完成，最后完成步骤4）。SELECT ...FOR UPDATE;

（2）解决不可重复读（幻读）

* Mysql实现了多版本并发控制，并且都是靠保存数据快照来实现的。
* 以 InnoDB 为例，每一行中都冗余了两个字断。一个是行的创建版本，一个是行的删除（过期）版本。版本号（trx\_id）随着每次事务的开启自增。事务每次取数据的时候都会取创建版本小于当前事务版本的数据，以及过期版本大于当前版本的数据。普通的select就是快照读，
* 在InnoDB存储引擎中，通过使用Next-Key Lock算法来避免不可重复读的问题，在这个算法下，对于索引的扫描，不仅仅是锁住扫描到的索引，而且还锁住这些索引覆盖的范围，在这个范围内的插入都是不允许的，这样就避免了另一个事务在这个范围内插入数据导致的不可重复读的问题。也就是说next-key锁包含两部分：记录锁（行锁）；间隙锁；记录锁是加在索引上的锁，间隙锁是加在索引之间的。将当前数据行与上一条数据和下一条数据之间的间隙锁定，保证此范围内读取的数据是一致的。

### MySQL为什么要建立索引？

（A）建立索引的优点：创建索引可以大大提高系统的性能。

1. 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。
2. 可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。
3. 可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。
4. 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。
5. 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

（B）增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？

1. 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。
2. 索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间。如果要建立聚簇索引，那么需要的空间就会更大。
3. 当对表中的数据进行增加、删除和修改时，索引也要动态的维护，这样就降低了数据的维护速度。

### 5.1.7.2 数据库有几种索引

**1.主键索引**

**2.唯一索引**

**3.普通索引**

**4.全文索引**

**5.联合索引**

### MySQL建立索引的原则？

1. 选择唯一性索引：唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。例如，学生表中学号是具有唯一性的字段。为该字段建立唯一性索引可以很快的确定某个学生的信息。如果使用姓名的话，可能存在同名现象，从而降低查询速度。
2. 为经常需要排序、分组和联合操作的字段建立索引：经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。
3. 为常作为查询条件的字段建立索引：如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。
4. 限制索引的数目：索引的数目不是越多越好。每个索引都需要占用磁盘空间，索引越多，需要的磁盘空间就越大。修改表时，对索引的重构和更新很麻烦。越多的索引，会使更新表变得很浪费时间。
5. 尽量使用数据量少的索引：如果索引的值很长，那么查询的速度会受到影响。例如，对一个CHAR(100)类型的字段进行全文检索需要的时间肯定要比对CHAR(10)类型的字段需要的时间要多。
6. 尽量使用前缀来索引：如果索引字段的值很长，最好使用值的前缀来索引。例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间。如果只检索字段的前面的若干个字符，这样可以提高检索速度。
7. 删除不再使用或者很少使用的索引：表中的数据被大量更新，或者数据的使用方式被改变后，原有的一些索引可能不再需要。数据库管理员应当定期找出这些索引，将它们删除，从而减少索引对更新操作的影响。
8. 最左前缀匹配原则，非常重要的原则：mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b= 2 c> 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。
9. 尽量选择区分度高的列作为索引：区分度的公式是count(distinct col)/count(\*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些状态、性别字段可能在大数据面前区分度就是0，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录。

### 5.1.8.2 索引的缺点

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

### MySQL中如何强制走指定的索引？

1、mysql强制使用索引：force index

例如:

select \* from table force index(PRI) limit 2;(强制使用主键)

select \* from table force index(name) limit 2;(强制使用索引"name")

select \* from table force index(PRI,age) limit 2;(强制使用索引"PRI和age")

2、mysql禁止某个索引：ignore index

例如:

select \* from table ignore index(PRI) limit 2;(禁止使用主键)

select \* from table ignore index(name) limit 2;(禁止使用索引"name")

select \* from table ignore index(PRI,age) limit 2;(禁止使用索引"PRI,age")

### SQL语句的执行顺序？

|  |
| --- |
| (8)SELECT (9)DISTINCT<select\_list>  (1)FROM<left\_table>  (3)<join\_type>JOIN<right\_table>  (2) ON<join\_condition>  (4)WHERE<where\_condition>  (5)GROUP BY<group\_by\_list>  (6)WITH{CUBE|ROLLUP}  (7)HAVING<having\_condition>  (10)ORDER BY<order\_by\_list>  (11)LIMIT<limit\_number> |

以上的sql语句前的数字代表该sql语句的执行顺序，每一个操作都会产生一个虚拟表，这个虚拟表作为下个操作的输入，依次类推，最后生成的虚拟表作为结果输出。如果没有在查询中指定某一字句，则将跳过相应的步骤:

1. FROM:对FROM子句中的左表<left\_table>和右表<right\_table>执行笛卡尔积，产生虚拟表VT1.
2. ON：对虚拟表VT1应用ON筛选，只有那些符合条件<join\_condition>的行才会被插入虚拟表VT2中。
3. JOIN:如果是外连接，还要根据外连接的类型执行join操作，例如左外连接将左表中的行并且不在VT2中的行加入到VT2中，产生VT3.
4. WHERE：对虚拟表VT3应用where过滤条件，只有符合<where\_condition>的记录才被插入虚拟表VT4中。
5. GROUP BY:根据GROUP BY字句中的列，对VT4中的记录进行分组操作，产生VT5.
6. CUBE|ROLLUP:对表VT5执行cube或者rollup操作，产生虚拟表VT6.
7. HAVING:对虚拟表VT6应用HAVING过滤器，只有符合<having\_condition>的记录才会被插入虚拟表VT7中
8. SELECT:对VT7执行select操作，将指定的数据插入到虚拟表VT8中.
9. DISTINCT:去除重复数据，产生虚拟表VT9.
10. ORDER BY:按照<order\_by\_list>对VT9进行排序操作，产生虚拟表VT10
11. LIMIT:选出指定的行，产生虚拟表VT11,返回给用户。

### DML、DDL、DCL

* DML（Data Manipulation Language）：对数据库中的数据进行操作
* SELECT、UPDATE、INSERT、DELETE。
* DDL（Data Definition Language）：数据库定义语言，用于定义和管理SQL数据库中所有对象的语言
* CREATE、ALTER、DROP、TRUNCATE。
* DML（Data Control Language）：数据控制语言，用来授予或回收访问数据库的某种特权，并控制数据库事务
* COMMIT（提交）、ROLLBACK（回滚）、SAVEPOINT（保存点）、SET TRANSACTION（设置当前事务的属性）。

### MySQL如何实现分页？如何优化limit？

（1）分页查询的语句：\*\*\*\* limit [offset] rows;

一般是用于select语句中用以从结果集中拿出特定的一部分数据。offset是偏移量，表示我们现在需要的数据是跳过多少行数据之后的，可以忽略；rows表示我们现在要拿多少行数据。例子：

|  |
| --- |
| 1. select \* from mytbl limit 10000,100 表示从表mytbl中拿数据，跳过10000行之后，拿100行 2. select \* from mytbl limit 0,100 表示从表mytbl拿数据，跳过0行之后，拿取100行 3. select \* from mytbl limit 100 这条SQL跟②的效果是完全一样的，表示拿前100条数据 |

（2）用处：

目前用到的地方是数据库查询分页，比如前台要展示数据库中数据，需要后台实现分页，传入数据要有“页码page”跟“每页数据条数nums”。对应SQL大概是：select \* from mytbl order by id limit (page-1)\*nums,nums

（3）问题

在数据量不大或者是大数据量的前几页的时候，性能还算不坏，但是大数据量页码稍微大一点性能便下降比较严重。原因出在Limit的偏移量offset上，比如limit 100000,10虽然最后只返回10条数据，但是偏移量却高达100000，数据库的操作其实是拿到100010数据，然后返回最后10条。

（4）解决方式

* 类似于分段。我们给每次只能翻100页、超过100页的需要重新加载后面的100页。这样就解决了每次加载数量数据大、速度慢的问题了；
* 记录每次取出后的最大id，然后 select id from test where id = 最大id limit 10；这样就可以解决了；
* 如果对于有where 条件，又想走索引用limit的，必须设计一个索引，将where用到的字段放第一位，limit用到的字段放第2位，而且只能select 主键；

### 索引为什么采用B+树？

从外存储器中读取信息的步骤，简单来分，大致有两步：

* 找到存储这个数据所对应的磁盘页面，这个过程是机械化的过程，需要依靠磁臂的转动，找到对应磁道，所以耗时长。
* 读取数据进内存，并实施运算，这是电子化的过程，相当快。

综上，对于外存储器的信息读取最大的时间消耗在于寻找磁盘页面。那么一个基本的想法就是能不能减少这种读取的次数，在一个磁盘页面上，多存储一些索引信息。B树的基本逻辑就是这个思路，它要改二叉为多叉，每个节点存储更多的指针信息，树的高度降低了，所以对I/O的操作也降低了。

1. B+树的磁盘读写代价更低：B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说I/O读写次数也就降低了。
2. B+树的查询效率更加稳定：由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
3. B+树更有利于对数据库的扫描：B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题，而B+树只需要遍历叶子节点就可以解决对全部关键字信息的扫描，所以对于数据库中频繁使用的range query，B+树有着更高的性能。

### 数据库B+树索引面试相关？

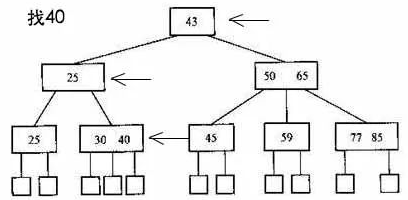
建立索引，MySQL中采用B+树索引。在我们学的数据结构中，二叉排序树可以很快的找到需要的元素，但是在一些极端的情况下，比如插入序列是有序的，二叉树就会出现退化的情况，退化成链表结构，这时候查找性能就会下降，所以就引出平衡二叉树，再插入的时候同时调整树，让其节点尽可能均匀的分布，红黑树就是其中的一种；

B树是一种多路搜索树，每个节点都可以拥有多于两个孩子节点，M路的B树最多可以拥有M个孩子节点。为什么要设计成M路呢？因为要进一步的降低树的高度，路数越多，树的高度越低，但是如果不限制路数，B树就退化成有序数组了。

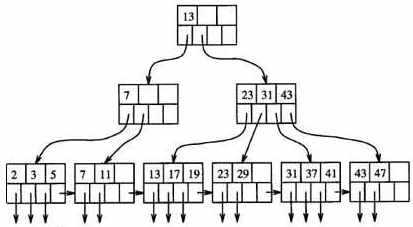
为什么文件系统的索引要使用B树而不是红黑树或数组呢？因为文件系统和数据库的索引都是存在硬盘上的，并且如果数据量大的话，不一定能一次性加载到内存中。如果一棵树都无法一次性加载到内存中那该怎么查找呢？

这时候B树的多路存储优势就出来了，可以每次加载B树的一个节点，然后一步一步的往下找，假设内存一次性只能加载2个数，有序数组是无法一次性的加载进内存的，如果设计成M路的B树，就可以一次只加载进两个节点。





如果在内存中操作，红黑树效率会比B树效率更高，但是涉及到磁盘操作，B树就更优。B+树是在B树的基础上改造的，它的数据都在叶子节点，同时叶子节点之间还加了指针形成链表。



为什么要设计成这样？数据库索引采用的是B+树，比如我们在数据库中select数据，不一定只选一条，很多时候会选多条，比如按照id排序后选10条。如果是范围查找，B树需要做局部的中序遍历，可能要跨层访问。而B+树由于所有数据都在叶子结点，不用跨层，同时由于有链表结构，只需要找到首尾，通过链表就能把所有数据取出来了。比如选出7-19，是需要在叶子节点中就可以找到了。



Hash结构存储索引查询时间是O(1)，B+树的查找时间约为log(n)，为什么不用hash结构呢？

如果只选一个数据，那确实是hash更快。但是数据库中经常会选择多条，这时候由于B+树索引有序，并且又有链表相连，它的查询效率比hash就快很多了。而且数据库中的索引一般是在磁盘上，数据量大的情况可能无法一次装入内存，B+树的设计可以允许数据分批加载，同时树的高度较低，查找效率增加了。

### 5.1.14.2 B+树索引和哈希索引的区别

**B+树的特点**

B+树的非叶子节点只是存储key，占用空间非常小，因此每一层的节点能索引到的数据范围更加的广。换句话说，每次IO操作可以搜索更多的数据。

- 叶子节点两两相连，符合磁盘的预读特性。比如叶子节点存储50和55，它有个指针指向了60和62这个叶子节点，那么当我们从磁盘读取50和55对应的数据的时候，由于磁盘的预读特性，会顺便把60和62对应的数据读取出来。这个时候属于顺序读取，而不是磁盘寻道了，加快了速度。

- 支持范围查询，而且部分范围查询非常高效，每个节点能索引的范围更大更精确,也意味着 B+树单次磁盘IO的信息量大于B-树,I/O效率更高。

原因是数据都是存储在叶子节点这一层，并且有指针指向其他叶子节点，这样范围查询只需要遍历叶子节点这一层，无需整棵树遍历。

**区别**

如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据。

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索。

同理，哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）。

哈希索引也不支持多列联合索引的最左匹配规则。

B+树索引的关键字检索效率比较平均，不像B树那样波动幅度大，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

解决Hash碰撞冲突方法总结 https://blog.csdn.net/zeb\_perfect/article/details/52574915

### 如何优化数据库？

1. 根据服务层面：配置mysql性能优化参数；
2. 从系统层面增强mysql的性能：优化数据表结构

* 将字段较多的表分解成多个表：对于字段较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。因为当一个表的数据量很大时，会由于存在使用频率低的字段而使查询速度变慢。
* 增加中间表：对于经常需要联合查询的表，可以建立中间表以提高查询效率。通过建立中间表，把需要经常联合查询的数据插入中间表，然后将原来的联合查询改为对中间表的查询，以此来提高查询效率。

1. 从数据库层面增强性能：

* 优化SQL语句，合理使用字段索引。①并不是索引越多越好，建少了，用where子句找数据效率比较低，建多了，执行增、删、改操作时，会额外的更新索引，导致性能较慢，同时，索引占用了磁盘空间，过多的索引也会导致空间浪费。②并不是有索引一定比没有索引要好。对于数据量较少的表，不需要创建索引，因为有索引时，系统首先会去寻找索引页，然后再去找实际的数据项。
* 使用缓存和NoSQL数据库方式存储，如MongoDB/ Redis来缓解高并发下数据库查询的压力。
* 减少数据库操作次数，尽量使用数据库访问驱动的批处理方法。
* 不常使用的数据迁移备份，避免每次都在海量数据中去检索。
* 提升数据库服务器硬件配置，或者搭建数据库集群。
* 编程手段防止SQL注入：使用JDBC PreparedStatement按位插入或查询；正则表达式过滤（非法字符串过滤）；

### sql语句调优过程？

（1）第一步：开启慢查询日志，捕获慢SQL

* 查看慢查询日志是否开启，如果没有开启，就开启慢查询日志；
* 查看慢查询日志的阈值（这个值一般表示超过多少时间的SQL语句会被记录到慢查询日志中），当然也可以根据业务需求自己设置慢查询日志的阈值；
* 查看有多少SQL语句的查询时间超过了阈值；
* 进入MQSQL安装目录，使用mysqldumpslow命令查看使用次数最多的SQL语句、查询时间排序的sql；

（2）第二步：explain+慢SQL分析

* 使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理SQL语句的。可以据此分析查询语句或是表结构的性能瓶颈。
* 使用方式：Explain+SQL语句，执行这个命令后，显示如下：

|  |
| --- |
| +---+----+-----+------+------+------+-----+------+-----+-----+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +---+----+-----+------+------+------+-----+------+-----+-----+ |

* type：显示访问类型，结果值从好到坏依次是：

system>const>eq\_ref>ref……… >unique\_subquery>index\_subquery>range>index>ALL

一般来说，得保证查询至少达到range级别，最好能达到ref。

* ref：显示索引的哪一列被使用了，哪些列或常量被用于查找索引列上的值。

（3）第三步：show profile查询SQL语句在服务器中的执行细节和生命周期

* Show Profile是MySQL提供可以用来分析当前会话中语句执行的资源消耗情况，可以用于SQL的调优测量；默认关闭，并保存最近15次的运行结果；比如遇到下面的情况要进行优化：创建临时表；把内存中的临时表复制到磁盘中；

（4）第四步：SQL数据库服务器参数调优

* 当order by 和 group by无法使用索引时，增大max\_length\_for\_sort\_data参数设置和增大sort\_buffer\_size参数的设置；

### 防止SQL注入的方式？

所谓SQL注入，就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。它不是利用操作系统的BUG来实现攻击，而是针对程序员编程时的疏忽，通过SQL语句，实现无帐号登录，甚至篡改数据库。

（1）PreparedStatement

* 采用预编译语句集，它内置了处理SQL注入的能力，只要使用它的setXXX方法传值即可。sql注入只对sql语句的编译过程有破坏作用，而PreparedStatement已经准备好了，执行阶段只是把输入串作为数据处理，而不再对sql语句进行解析,准备,因此也就避免了sql注入问题.

（2）使用正则表达式过滤传入的参数

* 要引入的包：import Java.util.regex.\*；然后判断输入的sql语句是不是与正则表达式匹配；

（3）字符串过滤

* 这是一个比较通用的方法，将输入的sql字符串使用 | 隔开，最后进行分割；

（4）JSP页面判断代码

* 使用JavaScript在客户端进行不安全字符屏蔽；

（5）使用一些持久层的框架，比如mybatis等；

### 聚集索引与非聚集索引？

索引分为两大类：聚集索引和非聚集索引，其中，非聚集索引又可细分为普通索引、唯一索引、组合索引。

（1）聚集索引

* 定义：聚集索引规定了表中数据的物理存放顺序，数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引，可以类比于字典中的按拼音目录。它的优势在于：查询小数据量的表；查询指定范围内的数据。

在MySQL数据库的InnoDB存储引擎中，主键索引就是聚集索引，所有数据都会按照主键索引进行组织；而在MyISAM存储引擎中，就没有聚集索引了，因为MyISAM存储引擎中的数据不是按索引顺序进行存储的。

* 聚集索引特点：
  + 聚集索引的叶节点就是实际数据页；
  + 在数据页中数据按照索引顺序存放；
  + 行的物理位置和其在索引中的位置是相同的；
  + 每个表只能有一个聚集索引。
  + 如果表中数据列频繁更改，就会导致整行数据的移动，这时不适合使用聚集索引。

（2）非聚集索引

* 定义：非聚集索引是独立于数据表之外的结构，即数据存放在一个地方，索引存放在另一个地方，索引叶子节点通过其自带的指针指向数据的存储位置（聚集索引叶子节点）。表中的非聚集索引可以创建多个，可以为表常用的每个列都创建一个非聚集索引（索引个数适当就好，不要贪多）。

（3）两者比较

* 速度：聚集索引，在插入数据时，速度较慢（时间主要花费在“物理存储的排序”上，也就是首先要找到位置然后再插入数据）。但查询数据的速度比非聚集索引要快。

（4）哪些情况下索引会失效？

* + 以“%”开头的LIKE语句，模糊匹配
  + OR语句前后没有同时使用索引
  + 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）
  + 带非操作符，如< > != not in not exist
  + 列上作运算

### 数据库设计三范式？

第一范式：属性不可分割；

所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

第二范式：要有主键，要求其他字段都依赖于主键；

第二范式是在第一范式的基础上建立起来的，即满足第二范式必须先满足第一范式。第二范式要求数据库表中的每个实例或行必须可以被惟一的区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。要求实体的属性完全依赖于主关键字。

第三范式：消除传递依赖；

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。

不能存在传递依赖。即：除主键外，其他字段必须依赖主键。

### MySQL读写分离（主从复制）？主从同步延时？

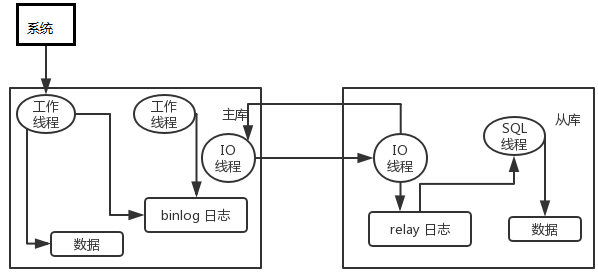
高并发这个阶段，肯定是需要做读写分离的，其实对数据库的访问大多都是读多写少。所以针对这个情况，就是写一个主库，但是主库挂多个从库，然后从多个从库来读，那不就可以支撑更高的读并发压力了吗？

基于主从复制架构可以实现读写分离：一个主库，挂多个从库，然后就单单只是写主库，然后主库会自动把数据给同步到从库上去。

MySQL 主从复制原理的是啥？

MySQL的复制（replication）是一个异步的复制，从一个Master复制到Slave。整个复制操作主要由三个进程完成的，其中两个进程在Slave（Sql进程和IO进程），另外一个进程在Master（IO进程）上。要实施复制，首先必须打开Master端的binarylog（bin-log）功能，否则无法实现。

过程：主库将变更写入binlog日志，然后从库连接到主库之后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个relay中继日志中。接着从库中有一个SQL线程（SHOW PROCESSLIST显示哪些线程正在运行。）会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。



从库同步主库数据的过程是串行化的，也就是说主库上并行的操作，在从库上会串行执行。

* 问题1：由于从库从主库拷贝日志以及串行执行SQL的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。
* 问题2：如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

MySQL有两个机制解决上面的问题：一个是半同步复制，用来解决主库数据丢失问题；一个是并行复制，用来解决主从同步延时问题。

* + 半同步复制，也叫semi-sync复制，指的就是主库写入binlog日志之后，就会强制立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。
  + 并行复制，指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

（2）MySQL主从同步延时问题（精华）

以前线上确实处理过因为主从同步延时问题而导致的线上的 bug，属于小型的生产事故。场景复现：先插入一条数据，再把它查出来，然后更新这条数据。在生产环境高峰期，写并发达到了2000/s，这个时候，主从复制延时大概是在小几十毫秒。线上会发现，每天总有那么一些数据，我们期望更新一些重要的数据状态，但在高峰期时候却没更新。

通过MySQL 命令：show status，查看Seconds\_Behind\_Master，可以看到从库复制主库的数据落后了几ms。一般来说，如果主从延迟较为严重，有以下解决方案：

* + 分库，将一个主库拆分为多个主库，每个主库的写并发就减少了几倍，此时主从延迟可以忽略不计。
  + 打开MySQL支持的并行复制，多个库并行复制。如果说某个库的写入并发就是特别高，单库写并发达到了2000/s，并行复制还是没意义。
  + 重写代码，写代码的同学，要慎重，插入数据时立马查询可能查不到。
  + 如果确实是存在必须先插入，立马要求就查询到，然后立马就要反过来执行一些操作，对这个查询设置直连主库。不推荐这种方法，你要是这么搞，读写分离的意义就丧失了。

### MySQL的悲观锁、乐观锁？

* 乐观锁：
  + 定义：指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。乐观锁不是数据库自带的，需要我们自己去实现。
  + 实现：对表中的数据进行操作时(更新)，先在数据表加一个版本(version)字段，每一次操作，先查询出要更新的那条记录，获取出version字段：如果要对那条记录进行操作(更新),则先判断此刻version的值是否与刚刚查询出来时的version的值相等，如果相等，则说明这段期间，没有其他程序对其进行操作，则可以执行更新，将version字段的值加1；如果更新时发现此刻的version值与刚刚获取出来的version的值不相等，则说明这段期间已经有其他程序对其进行操作了，则不进行更新操作。
  + 案例：

|  |
| --- |
| 下单操作包括3步骤：  1.查询出商品信息：SELECT (status,status,version) FROM t\_goods WHERE id=#{id}  2.根据商品信息生成订单  3.修改商品status为2  UPDATE t\_goods SET status=2,version=version+1 WHERE id=#{id} and version=#{version}; |

* 悲观锁：（包括共享锁和排它锁）
  + 定义：悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。
  + 实现：对表中的数据进行操作时(更新)，先在数据表加一个版本(version)字段，每一次操作，先查询出要更新的那条记录，获取出version字段

### Mysql批量插入数据

在MySQL数据库中插入大量的数据，怎么才能快速插入到MySQL里呢？

数据量非常大的时候，不能使用INSERT INTO一条一条地插入，MySQL提高数据插入效率的基本原则如下：

* 批量插入数据的效率比单数据行插入的效率高
* 插入无索引的数据表比插入有索引的数据表快一些
* 较短的SQL语句的数据插入比较长的语句快

|  |
| --- |
| > use ecommerce;  > DROP PROCEDURE BatchInser IF EXISTS;  > delimiter // -- 把界定符改成双斜杠  > CREATE PROCEDURE BatchInsert(IN init INT, IN loop\_time INT) -- 第一个参数为初始ID号（可自定义），第二个位生成MySQL记录个数  BEGIN  DECLARE Var INT;  DECLARE ID INT;  SET Var = 0;  SET ID = init;  WHILE Var < loop\_time DO  insert into employees(id, fname, lname, birth, hired, separated, job\_code, store\_id) values (ID, CONCAT('chen', ID), CONCAT('haixiang', ID), Now(), Now(), Now(), 1, ID);  SET ID = ID + 1;  SET Var = Var + 1;  END WHILE;  END;  //  > delimiter ; -- 界定符改回分号  > CALL BatchInsert(30036, 200000); -- 调用存储过程插入函数 |

几个结论：

* 使用LOAD DATA语句要比INSERT语句效率高，因为它批量插入数据行。服务器只需要对一个语句（而不是多个语句）进行语法分析和解释。索引只有在所有数据行处理完之后才需要刷新，而不是每处理一行都刷新。
* 如果你只能使用INSERT语句，那就要使用将多个数据行在一个语句中给出的格式：INSERT INTO table\_name VALUES(...),(...),...，这将会减少你需要的语句总数，最大程度地减少了索引刷新的次数。
* 在默认情况下，LOAD DATA语句将假设各数据列的值以制表符（t）分隔，各数据行以换行符（n）分隔，数据值的排列顺序与各数据列在数据表里的先后顺序一致。

### 平衡二叉树和红黑树的区别

红黑树的性质：

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

平衡二叉树的性质：

它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

区别：

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

2、平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

5.1.24 最左匹配原则

<https://www.cnblogs.com/lanqi/p/10282279.html>

顾名思义：最左优先，以最左边的为起点任何连续的索引都能匹配上。同时遇到范围查询(>、<、between、like)就会停止匹配。  
例如：b = 2 如果建立(a,b)顺序的索引，是匹配不到(a,b)索引的；但是如果查询条件是a = 1 and b = 2或者a=1(又或者是b = 2 and b = 1)就可以，因为优化器会自动调整a,b的顺序。再比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，因为c字段是一个范围查询，它之后的字段会停止匹配。

## Redis

### Redis五种数据类型

（1）String

* 格式：set key value
* 说明：二进制安全的，String类型可包含任何数据；String在redis内部存储默认是一个动态字符串，被redisObject所引用，当遇到incr，decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

（2）Hash

* 格式：hmset name key1 value1 key2 value2
* hash 是一个键值对集合，string类型的field和value的映射表，特别适合用于存储对象。
* Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，当我们需要更改数据时，只需要通过key获取value，通过value可以得到具体的字段和属性值。

（3）List：3.2版本后底层采用quicklist双向链表实现的；在3.2版本之前，列表是使用ziplist和linkedlist实现的；

|  |  |
| --- | --- |
| lpush name value | 在 key 对应 list 的头部添加字符串元素 |
| rpush name value | 在 key 对应 list 的尾部添加字符串元素 |
| lrem name index | key 对应 list 中删除 count 个和 value 相同的元素 |
| llen name | 返回 key 对应 list 的长度 |

* 说明：List是简单的字符串列表，按照插入顺序排序。可以添加一个元素到列表的头部或者尾部；
* 使用场景：list的应用场景非常多，也是Redis最重要的数据结构之一，
* twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现。
* Lists的另一个应用就是消息队列，可以利用Lists的PUSH操作，将任务存在Lists中，然后工作线程再用POP操作将任务取出进行执行。
* Redis提供了操作Lists中某一段的api，可以直接查询、删除Lists中某一段的元素，可用于分页查询。

（4）Set：内部的实现是一个value永远为null的HashMap，实际上是通过计算hash的方式快速去重；

* 格式：sadd name value
* 说明：Redis的Set是string类型的无序集合。集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。利用Redis提供的Sets数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好等功能。

（5）Sorted Set有序集合：sorted set的内部使用HashMap和跳跃表（SkipList）来保证数据的存储和有序；

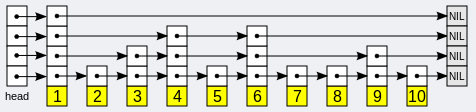
* 格式：zadd name score value
* zset 和 set 一样也是string类型元素的集合，且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的，但分数(score)却可以重复。

### Redis中的跳跃表

跳跃表是一种随机化数据结构，基于并联的链表，其效率可以比拟平衡二叉树，查找、删除、插入等操作都可以在对数期望时间内完成。跳跃表主要有以下几个部分构成：

* + 1、表头head：负责维护跳跃表的节点指针；
  + 2、节点node：实际保存元素值，每个节点有一层或多层；
  + 3、层level：保存着指向该层下一个节点的指针；
  + 4、表尾tail：全部由null组成；

跳跃表的遍历总是从高层开始，然后随着元素值范围的缩小，慢慢降低到低层。



Redis中跳跃表的数据结构对原来的跳跃表进行了一下修改：

* + 1、允许重复的score值：多个不同的元素(member)的score值可以相同；
  + 2、进行元素对比的时候，不仅要检查score值，还需要检查member：当score值相等时，需要比较member域进行比较；
  + 3、结构保存一个tail指针：跳跃表的表尾指针；
  + 4、每个节点都有一个高度为1层的前驱指针，用于从底层表尾向表头方向遍历；

### Redis为什么这么快？

* 1：Redis是纯内存数据库，一般都是简单的存取操作，线程占用的时间很多，时间的花费主要集中在IO上，所以读取速度快。
* 2：Redis使用的是非阻塞IO，IO多路复用，使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争。
* 3：Redis采用了单线程的模型，保证了每个操作的原子性，也减少了线程的上下文切换和竞争。
* 4：数据结构也帮了不少忙，Redis全程使用hash结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度。
* 5：Redis采用自己实现的事件分离器，效率比较高，内部采用非阻塞的执行方式，吞吐能力比较大。

### Redis缓存穿透？缓存雪崩？如何避免？

（1）缓存穿透

* 一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该访问数据库查询。一些恶意的请求会故意查询不存在的key，请求量很大，就会对数据库造成很大的压力。这就叫做缓存穿透。

（2）如何解决缓存穿透？

* 1：对查询结果为空的情况也进行缓存，将缓存的过期时间设置短一点（一般用这个），或者该key对应的数据insert了之后清理缓存。
* 2：利用布隆过滤器对一定不存在的key进行过滤。它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。这样在查询缓存之前先去过滤器中查询缓存是否有存在该key。不过这个适合于数据量固定且较少，实时性低的应用中，因为要维护这一个过滤器，数据大的时候布隆过滤器误算率高。

（3）缓存雪崩

* 当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在缓存层失效的时候，所有的请求都直接穿透到DB层，会给数据库带来很大压力。导致系统崩溃。

（4）如何解决缓存雪崩？

* 1保证缓存层的高可用性，比如Redis的Sentinel哨兵模式（主从模式下，额外有一个Redis当哨兵，如果master挂了，哨兵从Slaver中重新选一个master出来）和Redis的Cluster集群模式都实现了高可用性。
* 2在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
* 3做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期
* 4不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

### Redis持久化方式？

1. RDB：快照形式，定期把内存中当前时刻的数据保存到磁盘。Redis默认支持的持久化方案。速度快但是服务器断电的时候会丢失部分数据。
2. AOF形式：append only file。把所有对redis数据库操作的命令，增删改操作的命令，保存到文件中。数据库恢复时把所有的命令执行一遍即可。两种持久化方案同时开启使用AOF文件来恢复数据库.能保证数据的完整性,但是速度慢。
3. 两者如何选择?

* 不要仅仅使用 RDB，因为那样会导致你丢失很多数据；
* 也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug；
* redis 支持同时开启开启两种持久化方式，我们可以综合使用 AOF 和 RDB 两种持久化机制，用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。

### Redis过期策略与内存淘汰机制

（1）Redis的过期策略：定期删除+惰性删除

定期删除指的是redis默认每隔100ms就**随机抽取**一些设置了过期时间的key，检查是否过期，如果过期就删除。假设 redis 里放了 10w 个 key，都设置了过期时间，你每隔几百毫秒，就检查 10w 个 key，那 redis 基本上就死了，cpu 负载会很高的，消耗在你的检查过期 key 上了。注意，这里可不是每隔 100ms 就遍历所有的设置过期时间的 key，那样就是一场性能上的灾难。实际上 redis 是每隔 100ms随机抽取一些 key 来检查和删除的。

定期删除可能会导致很多过期 key 到了时间并没有被删除掉，可以使用惰性删除：在你获取某个 key 的时候，redis 会检查这个key，如果key过期了就会删除，不会返回任何东西。

但是实际上还是有问题的，如果定期删除漏掉了很多过期的key，没有及时的去查，也没有惰性删除，redis内存依然会耗尽。解决方式：内存淘汰机制。

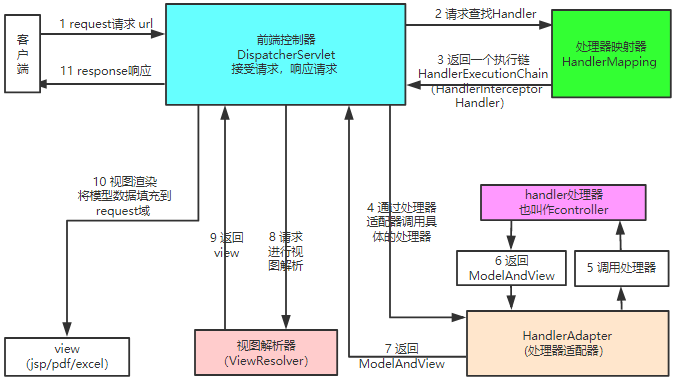
（2）内存淘汰机制

redis内存淘汰机制有以下几个：

|  |  |
| --- | --- |
| noeviction | 当内存不足以容纳新写入数据时，新写入操作会报错（一般不会用）。 |
| allkeys-lru | 当内存不足以容纳新写入数据时，在**键空间**中，移除最近最少使用的 key（最常用）。 |
| allkeys-random | 当内存不足以容纳新写入数据时，在**键空间**中，随机移除某个 key（一般很少用）。 |
| volatile-lru | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，移除最近最少使用的key（这个一般不太合适）。 |
| volatile-random | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，随机移除某个 key。 |
| volatile-ttl | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，有更早过期时间的 key 优先移除。 |

# 框架

## SpringMVC框架流程



第一步：发起请求到前端控制器(DispatcherServlet)；

第二步：前端控制器根据请求信息调用HandlerMapping查找 Handler（可以根据xml配置、注解进行查找）；

第三步：处理器映射器HandlerMapping向前端控制器返回Handler；

第四步：前端控制器调用处理器适配器去执行Handler；

第五步：处理器适配器去执行Handler；

第六步：Handler执行完成给处理器适配器返回ModelAndView；

第七步：处理器适配器向前端控制器返回ModelAndView；

ModelAndView是springmvc框架的一个底层对象，包括 Model和view

第八步：前端控制器请求视图解析器去进行视图解析；

根据逻辑视图名解析成真正的视图(jsp)

第九步：视图解析器向前端控制器返回View；

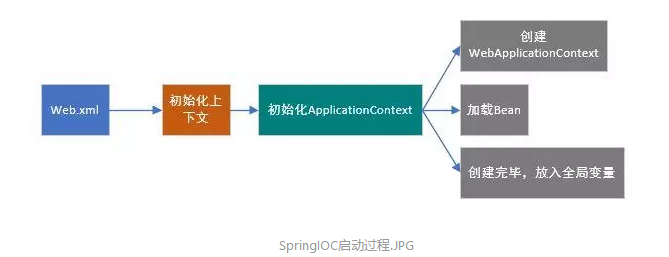
第十步：前端控制器进行视图渲染；

视图渲染将模型数据(在ModelAndView对象中)填充到request域

第十一步：前端控制器向用户响应结果；

## 6.1.2 Spring的启动过程

spring的启动是建筑在servlet容器之上的，所有web工程的初始位置就是web.xml,它配置了servlet的上下文（context）和监听器（Listener）



### 6.1.3 Spring SpringMVC和SpringBoot的区别

spring boot就是一个大框架里面包含了许许多多的东西，其中spring就是最核心的内容之一，当然也就包含spring mvc。

spring mvc 只是spring 处理web层请求的一个模块。因此他们的关系大概就是这样：spring mvc  < spring < springboot。

说得更简便一些：Spring 最初利用“工厂模式”（DI）和“代理模式”（AOP）解耦应用组件。

大家觉得挺好用，于是按照这种模式搞了一个 MVC框架（一些用Spring 解耦的组件），用于开发 web 应用（ SpringMVC ）。

然后又发现每次开发都写很多样板代码，为了简化工作流程，于是开发出了一些“懒人整合包”（starter），这套就是 Spring Boot。

## SpringMVC的过滤器和拦截器的区别？

Spring的拦截器与Servlet的Filter有相似之处,比如二者都是AOP编程思想的体现,都能实现权限检查、日志记录等。不同的是:

* 使用范围不同：Filter是Servlet规范规定的,只能用于Web程序中。而拦截器既可以用于Web程序，也可以用于Application、Swing程序中。
* 规范不同：Fitter是在Servlet规范中定义的，是Servlet容器支持的。而拦截器是在Spring容器内的，是Spring框架支持的。
* 使用的资源不同：同其他的代码块一样，拦截器也是一个Spring的组件，归Spring管理，配置在Spring文件中，因此能使用Spring里的任何资源、对象，例如Service对象、数据源、事务管理等，通过loC注入到拦截器即可；而Flter则不能。
* 深度不同：Filter在只在Serviet前后起作用。而拦截器能够深入到方法前后、异常抛出前后等,因此拦截器的使用具有更大的弹性。所以在Spring构架的程序中，要优先使用拦截器。

## Spring IOC

IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。

DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部资源。

最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

比如对象 A 需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了spring 我们就只需要告诉spring，A中需要一个 Connection，至于这个 Connection 怎么构造，何时构造，A不需要知道。在系统运行时，spring 会在适当的时候制造一个 Connection， 然后注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个 Connection 是由 spring 注入到A中的，依赖注入的名字就这么来的。那么 DI 是如何实现的呢？Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring 就是通过反射来实现注入的。

## Spring AOP

面向切面编程（AOP），作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

1. AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。
2. Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

* JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例，生成目标类的代理对象。
* 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

1. 静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

* InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例；method 是被代理目标实例的某个具体方法；args 是被代理目标实例某个方法的具体入参，在方法反射调用时使用。

## Spring创建bean的三种方式(IOC依赖注入)

1调用构造器（常用）

1. 利用无参构造函数 + setter方法注入值构造Bean对象实例

其本质是 SpringContext 利用无参的构造函数创建一个对象，然后利用setter方法赋值，所以需要在pojo中提供无参数构造函数，并定义对应字段的setter方法。如果无参构造函数不存在，Spring上下文创建对象的时候便会报错。

|  |
| --- |
| <!-- setter方法注入值时bean的配置方式 -->  <bean id="user" class="com.beans.user">  <property name="id" value="666"></property>  <property name="name" value="Leorizon"></property>  </bean> |

setter方法注入值时，xml配置bean时使用的<property name="属性名" value="值"></property>标签对属性进行赋值。Spring容器通过该方法为name属性注入参数。

1. 利用有参构造函数直接注入值构造Bean对象实例

当利用有参构造注入值构造时，pojo中不需要setter方法；

|  |
| --- |
| <!-- 构造注入值时bean的配置 -->  <bean id="user" class="com.beans.user" >  <constructor-arg name="id" value="666"></constructor-arg>  <constructor-arg name="name" value="Leorizon"></constructor-arg>  </bean> |

在构造注入值时，xml配置bean时通过使用<constructor-arg name="属性名" value="值"></constructor-arg>标签的name属性和value属性指定了：构造方法赋值

2调用静态工厂方法创建Bean对象

**静态工厂方法创建bean对象的核心是：class + factory-method**

首先是封装类对象，如果需要对属性注入值，则需要定义有参构造方法或者生产setter方法；

其次是定义静态工厂方法

|  |
| --- |
| public class UserFactory {  // 在这里我们定义创建方法是调用构造方法来创建对象  public static User createPerson(){  return new User();  }  public static User createPerson(Integer id,String name){  return new User(id,name);  }  } |

然后在XML文件定义bean对象

|  |
| --- |
| <!-- 调用静态工厂方法创建bean对象 -->  <bean id="user" class="com.beans.factory.UserFactory" factory-method="createPerson">  <constructor-arg name="id" value="666"></constructor-arg>  <constructor-arg name="name" value="Leorizon"></constructor-arg>  </bean> |

使用静态工厂方法创建Bean实例需要为<bean />元素指定出id如下属性：

* class：指定静态工厂的实现类(即包名.类名)【相当于指定工厂的地址】
* factory-method：指定由静态工厂的哪个方法创建该Bean实例（方法名）【指定由工厂的哪个车间创建Bean】
* 如果静态工厂方法需要参数，则使用<constructor-arg />元素传入

3调用实例工厂方法创建Bean对象

调用实例工厂方法创建bean对象相比较静态工厂方法创建bean对象，就是将静态方法中的factory实现类先实例化一个factorBean对象，然后调用factorBean对象来创建bean对象；

|  |
| --- |
| <!-- 先配置工厂Bean，class指定该工厂的实现类，该Bean负责产生其他Bean实例 -->  <bean id="userFactory" class="com.beans.factory.UserFactory"/>  <!-- 再引用工厂Bean来配置 其他Bean -->  <bean id="user" factory-bean="userFactory" factory-method="createPerson">  <constructor-arg name="id" value="666"></constructor-arg>  <constructor-arg name="name" value="Leorizon"></constructor-arg>  </bean> |

调用实例化工厂需要为<bean />指以下两个属性

* factory-bean ：该属性指定工厂Bean的id
* factory-method：该属性指定实例工厂的工厂方法。
* 如果方法需要参数，则使用<constructor-arg />元素传入

## Spring中bean的作用域？

|  |  |
| --- | --- |
| singleton | 唯一 bean 实例，Spring 中的 bean 默认都是单例的 |
| prototype | 每次请求都会创建一个新的 bean 实例 |
| request | 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。 |
| session | 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。 |
| global-session | 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。 |

## 将一个类声明为Spring的 bean 的注解有哪些?

我们一般使用 @Autowired 注解自动装配 bean，要想把类标识成可用于 @Autowired注解自动装配的 bean 的类,采用以下注解可实现：

* @Component：通用的注解，可标注任意类为 Spring 组件。如果一个Bean不知道属于哪个层，可以使用@Component 注解标注。
* @Repository：对应持久层即 Dao 层，主要用于数据库相关操作。
* @Service：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。
* @Controller：对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

## Spring中bean的生命周期

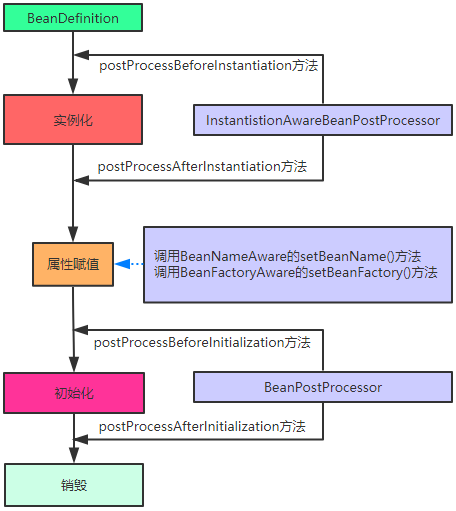
Spring Bean的生命周期只有这四个阶段

* 实例化 Instantiation
* 属性赋值 Populate
* 初始化 Initialization
* 销毁 Destruction

主要逻辑都在AbstractAutowireCapableBeanFactory 的doCreate()方法中，逻辑很清晰，就是顺序调用以下三个方法，这三个方法与三个生命周期阶段一一对应。

|  |
| --- |
| // 忽略了无关代码  protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullable Object[] args) throws BeanCreationException {  // Instantiate the bean.  BeanWrapper instanceWrapper = null;  if (instanceWrapper == null) {  // 实例化阶段！  instanceWrapper = createBeanInstance(beanName, mbd, args);  }  // Initialize the bean instance.  Object exposedObject = bean;  try {  // 属性赋值阶段！  populateBean(beanName, mbd, instanceWrapper);  // 初始化阶段！  exposedObject = initializeBean(beanName, exposedObject, mbd);  }  } |

至于销毁，是在容器关闭时调用的，详见ConfigurableApplicationContext#close()。



实现了下面两个接口的Bean会切入到多个Bean的生命周期中。

* BeanPostProcessor
* InstantiationAwareBeanPostProcessor

InstantiationAwareBeanPostProcessor作用于实例化阶段的前后，BeanPostProcessor作用于初始化阶段的前后。正好和第一、第三个生命周期阶段对应。

具体的生命周期：

* 实例化Bean：对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。
* 设置对象属性（依赖注入）：实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息以及通过BeanWrapper提供的设置属性的接口完成依赖注入。
* 处理Aware接口：Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：
  + - 如果Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法；
    - 如果Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法。
    - 如果Bean实现了ApplicationContextAware接口，会调用setApplicationContext()方法；
* BeanPostProcessor：如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization()方法。由于这个方法是在Bean初始化之前调用的，所以可以被应用于内存或缓存技术；
* InitializingBean与init-method：如果Bean在Spring配置文件中配置了init-method 属性，则会自动调用其配置的初始化方法。
* 如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization()方法；
* 以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。
* DisposableBean：当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；
* destroy-method：最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

## Spring BeanFactory与FactoryBean的区别

1.BeanFactory

BeanFactory，以Factory结尾，表示它是一个工厂类(接口)，用于管理Bean的一个工厂。在Spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

Spring为我们提供了许多易用的BeanFactory实现，XmlBeanFactory就是常用的一个，该实现将以XML方式描述组成应用的对象及对象间的依赖关系。XmlBeanFactory类将持有此XML配置元数据，并用它来构建一个完全可配置的系统或应用。

实例化容器

|  |
| --- |
| Resource resource = new FileSystemResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource);  ------------------------------------------------------------------------------------------------  ClassPathResource resource = new ClassPathResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource);  ------------------------------------------------------------------------------------------------  ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"applicationContext.xml", "applicationContext-part2.xml"});  BeanFactory factory = (BeanFactory) context; |

基本就是这些了，接着使用getBean(String beanName)方法就可以取得bean的实例；BeanFactory提供的方法及其简单，仅提供了六种方法供客户调用：

* boolean containsBean(String beanName)：判断工厂中是否包含给定名称的bean定义，若有则返回true
* Object getBean(String)：返回给定名称注册的bean实例。根据bean的配置情况，如果是singleton模式将返回一个共享实例，否则将返回一个新建的实例，如果没有找到指定bean,该方法可能会抛出异常
* Object getBean(String, Class) ：返回以给定名称注册的bean实例，并转换为给定class类型
* Class getType(String name) ：返回给定名称的bean的Class,如果没有找到指定的bean实例，则排除NoSuchBeanDefinitionException异常
* boolean isSingleton(String) ：判断给定名称的bean定义是否为单例模式
* String[] getAliases(String name) ：返回给定bean名称的所有别名

2.FactoryBean

一般情况下，Spring通过反射机制利用<bean>的class属性指定实现类实例化Bean，在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。

Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。FactoryBean接口对于Spring框架来说占用重要的地位，Spring自身就提供了70多个FactoryBean的实现。它们隐藏了实例化一些复杂Bean的细节，给上层应用带来了便利。从Spring3.0开始，FactoryBean开始支持泛型，即接口声明改为FactoryBean<T>的形式

以Bean结尾，表示它是一个Bean，不同于普通Bean的是：它是实现了FactoryBean<T>接口的Bean，根据该Bean的ID从BeanFactory中获取的实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身，如果要获取FactoryBean对象，请在id前面加一个&符号来获取。

3.BeanFactory和FactoryBean的区别

* BeanFactory是接口，提供了OC容器最基本的形式，给具体的IOC容器的实现提供了规范
* FactoryBean也是接口，为IOC容器中Bean的实现提供了更加灵活的方式，FactoryBean在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，我们可以在getObject()方法中灵活配置。其实在Spring源码中有很多FactoryBean的实现类.
* BeanFactory是个Factory，也就是IOC容器或对象工厂，FactoryBean是个Bean。在Spring中，所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的。但对FactoryBean而言，这个Bean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean，它的实现与设计模式中的工厂模式和修饰器模式类似。

## BeanFactory和ApplicationContext区别

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

> 继承MessageSource，因此支持国际化。

> 统一的资源文件访问方式。

> 提供在监听器中注册bean的事件。

> 同时加载多个配置文件。

> 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（2）BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean，确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

## Spring事务的控制

Spring并不直接管理事务，而是提供了多种事务管理器 ，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。

Spring事务管理器的接口是： org.springframework.transaction.PlatformTransactionManager ，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台自己的事情了。

|  |
| --- |
| Public interface PlatformTransactionManager()...{  // 根据指定的传播行为，返回当前活动的事务或创建一个新事务  TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException;  //提交事务  Void commit(TransactionStatus status) throws TransactionException;  //对执行的事务进行回滚  Void rollback(TransactionStatus status) throws TransactionException;  } |

事务管理器接口 PlatformTransactionManager 通过 getTransaction(TransactionDefinition definition) 方法来得到一个事务，这个方法里面的参数是 TransactionDefinition接口 ，这个接口定义了一些基本的事务属性（隔离级别、传播行为、回滚规则、是否只读、事务超时）。

TransactionDefinition接口中定义了五个表示**隔离级别**的常量：

|  |  |
| --- | --- |
| ISOLATION\_DEFAULT | 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE\_READ隔离级别 |
| ISOLATION\_READ\_UNCOMMITTED | 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读 |
| ISOLATION\_READ\_COMMITTED | 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生 |
| ISOLATION\_REPEATABLE\_READ | 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。 |
| ISOLATION\_SERIALIZABLE | 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，该级别可以防止脏读、不可重复读以及幻读。 |

注释：

* **脏读（Dirty read）**: 当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
* **丢失修改（Lost to modify）**: 指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
* **不可重复读（Unrepeatableread）**: 指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
* **幻读（Phantom read）**: 幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

TransactionDefinition定义中包括了如下几个表示**传播行为**的常量：

|  |  |  |
| --- | --- | --- |
| 支持当前事务的情况 | PROPAGATION\_REQUIRED | 若当前存在事务，则加入该事务；若当前没有事务，则创建一个新的事务 |
| PROPAGATION\_SUPPORTS | 若当前存在事务，则加入该事务；若当前没有事务，则以非事务的方式继续运行 |
| PROPAGATION\_MANDATORY | 若当前存在事务，则加入该事务；若当前没有事务，则抛出异常。 |
| 不支持当前事务的情况 | PROPAGATION\_REQUIRES\_NEW | 创建一个新的事务，若当前存在事务，则把当前事务挂起 |
| PROPAGATION\_NOT\_SUPPORTED | 以非事务方式运行，若当前存在事务，则把当前事务挂起 |
| PROPAGATION\_NEVER | 以非事务方式运行，若当前存在事务，则抛出异常 |
| Spring特有 | PROPAGATION\_NESTED | 若当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；  若当前没有事务，则该取值等价于PROPAGATION\_REQUIRED。 |

## SpringBoot

### Spring和SpringBoot的区别？

（1）Spring

* Spring框架为开发Java应用程序提供了全面的基础架构支持。它包含一些很好的功能，如依赖注入和开箱即用的模块，如：Spring JDBC 、Spring MVC 、Spring Security、 Spring AOP 、Spring ORM 、Spring Test；这些模块缩短应用程序的开发时间，提高了应用开发的效率。

（2）SpringBoot

* Spring Boot基本上是Spring框架的扩展，它消除了设置Spring应用程序所需的XML配置；还有一些独特的优点：
  + - 创建独立的spring应用；
    - 嵌入Tomcat，Jetty，而且不需要部署他们；
    - 提供了starter的poms配置，简化了Maven配置；
    - 自动配置spring容器中的bean，当不满足实际开发场景，可自定义bean的自动化配置；
    - 提供生产指标，健壮检查和外部化配置；

### SpringBoot的核心配置文件？

（1）application.properties

基本可以将所有支持spring boot相关技术的配置文件都统一配置到application.properties里面。即如果你想添加或修改一些配置的话，直接操作application.properites即可。（修改端口号、项目的url、开发环境的配置）

（2）bootstrap 配置文件

bootstrap 配置文件有以下几个应用场景。

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景；

### SpringBoot的启动过程？

SpringBoot启动的时候，不论调用什么方法，都会构造一个SpringApplication的实例，然后调用这个实例的run方法，这样就表示启动SpringBoot。

（1）在run方法调用之前，也就是构造SpringApplication的时候会进行初始化的工作，初始化的时候会做以下几件事：

* + 1.把参数sources设置到SpringApplication属性中，这个sources可以是任何类型的参数。
  + 2.判断是否是web程序，并设置到webEnvironment这个boolean属性中。
  + 3.找出所有的初始化器，默认有5个，设置到initializers属性中。
  + 4.找出所有的应用程序监听器，默认有9个，设置到listeners属性中。
  + 5.找出运行的主类(main class)。

（2）SpringApplication构造完成之后调用run方法，启动SpringApplication，run方法执行的时候会做以下几件事：

* + 1.构造一个StopWatch，观察SpringApplication的执行。
  + 2.找出所有的SpringApplicationRunListener并封装到SpringApplicationRunListeners中，用于监听run方法的执行。监听的过程中会封装成事件并广播出去让初始化过程中产生的应用程序监听器进行监听。
  + 3.构造Spring容器(ApplicationContext)，并返回。
* 3.1判断是否是web环境，是的话构造AnnotationConfigEmbeddedWebApplicationContext，否则构造AnnotationConfigApplicationContext。
* 3.2 初始化过程中产生的初始化器在这个时候开始工作。
* 3.3 Spring容器的刷新(完成bean的解析、各种processor接口的执行、条件注解的解析等等)。
* 4.从Spring容器中找出ApplicationRunner和CommandLineRunner接口的实现类并排序后依次执行。

## 什么是Mybatis

1. Mybatis是一个半ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql，可以严格控制sql执行性能，灵活度高。
2. MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
3. 通过xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过java对象和 statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。（从执行sql到返回result的过程）。

## Mybatis中#{}和${}的区别

1. #{}是预编译处理，${}是字符串替换。
2. Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；使用#{}可以有效的防止SQL注入，提高系统安全性。
3. Mybatis在处理${}时，就是把${}替换成变量的值。

## Mybatis中实体类中的属性名和表中字段名不一样？

（1） 通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

|  |
| --- |
| <select id=”selectorder” parametertype=”int” resultetype=”me.gacl.domain.order”>  select order\_id id, order\_no orderno ,order\_price price form orders where order\_id=#{id};  </select> |

（2） 通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

|  |
| --- |
| <select id="getOrder" parameterType="int" resultMap="orderresultmap">  select \* from orders where order\_id=#{id}  </select>  <resultMap type=”me.gacl.domain.order” id=”orderresultmap”>  <!–用id属性来映射主键字段–>  <id property=”id” column=”order\_id”>  <!–用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性–>  <result property = “orderno” column =”order\_no”/>  <result property=”price” column=”order\_price” />  </reslutMap> |

## Mybatis的一级、二级缓存

1. 一级缓存：基于 PerpetualCache 的HashMap本地缓存，其存储作用域为Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。
2. 二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态)，可在它的映射文件中配置<cache/> ；
3. 对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

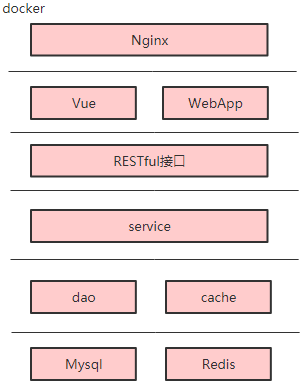
## mybatis返回结果转换为JSON格式？

直接用JSONArray无法实现转换，最后使用google的Gson配合JSONArray成功将结果集转换为json：

|  |
| --- |
| Gson gson = new Gson();  String jsonResult = gson.toJson(dataList, new TypeToken<List>(){}.getType());  JSONArray object = JSONArray.parseArray(jsonResult); |

# 项目总结

## 项目架构



底层mysql是存储数据，redis是缓存，dao层采用mybatis操作数据库，cache层操作redis，service层处理业务逻辑，rest api层为前端提供rest接口。前端这边用Vue+WebApp进行模块化和打包部署。网关nginx进行负载均衡。mysql、redis、nginx和springboot应用都放在docker里部署。

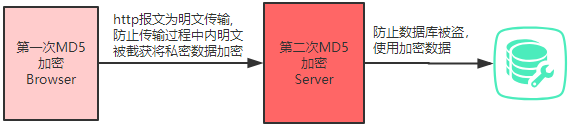
## Shiro密码加密（权限管理）

对于同一密码，同一加密算法会产生相同的hash值。这样，当用户进行身份验证时，对用户输入的明文密码应用相同的hash加密算法，得出一个hash值，然后使用该hash值和之前存储好的密文值进行对照，如果两个值相同，则密码认证成功，否则密码认证失败。

在shiro中也提供了编码，解码，加密，加密算法实现等等一系列的内容。加salt加密，salt是唯一值，比如身份证等，主要是为了解决多用户密码相同时加密后密码相同的问题，加salt后的密码是唯一值；

在Spring配置文件中可以配置自定义realm，指定加密的算法（MD5）以及加密的次数（两次）；

## 为什么使用两次MD5加密？



第一次MD5+固定Salt加密:

第一次MD5是在浏览器端，通过js文件中进行MD5加密，由于http报文的明文传输，如果传输的数据被截获(例如使用fiddler等抓包工具)，那用户密码将暴露无疑，所以我们使用MD5和固定Salt将表单的关键数据(例如密码)进行加密。

第二次MD5+随机Salt加密:

第二次MD5是在服务器端完成的，它将用户的关键信息进行加密后存入到数据库中，这样如果数据库的被盗后也得不到用户原始的密码信息。注意：

* MD5加密是不可逆的，但是你可能会发现有一些md5在线解密的网站能够通过密文获得你原始的数据，但它并不是反推出来的，而是通过反向查询，穷举字符组合的方式，创建了明文密文对应查询数据库，这样获得的你的原始数据。

## MD5加密原理？

MD5算法的过程分为四步：处理原文，设置初始值，循环加工，拼接结果。

* 第一步:处理原文：
  + 首先，我们计算出原文长度(bit)对512求余的结果，如果不等于448，就需要填充原文使得原文对512求余的结果等于448。填充的方法是第一位填充1，其余位填充0。填充完后，信息的长度就是512\*N+448。
  + 之后，用剩余的位置（512-448=64位）记录原文的真正长度，把长度的二进制值补在最后。这样处理后的信息长度就是512\*(N+1)。
* 第二步:设置初始值：
  + MD5的哈希结果长度为128位，按每32位分成一组共4组。这4组结果是由4个初始值A、B、C、D经过不断演变得到。MD5的官方实现中，A、B、C、D的初始值由官方给定。
* 第三步:循环加工：
  + 每一次循环都会让旧的ABCD产生新的ABCD。循环的次数由处理后的原文长度决定。
* 第四步:拼接结果：
  + 把循环加工最终产生的A，B，C，D四个值拼接在一起，转换成（32位的16进制）字符串即可。

## 如何防止通过URL地址栏直接访问页面？

（1）将所有页面放在WEB-INF目录下

* WEB-INF是Java的web应用安全目录，只对服务端开放，对客户端是不可见的。所以可以把除首页（index.jsp）以外的页面都放在WEB-INF目录下，这样就无法通过URL直接访问页面了。

（2）http请求头字段中的 referer值

* 根据HTTP协议，在HTTP头中有一个字段叫Referer，它记录了该HTTP请求的来源地址。在通常情况下，访问一个安全受限页面的请求必须来自于同一个网站。比如，要防御CSRF（跨站点请求伪造）攻击，网站只需要对于每一个请求验证其Referer值，如果是以bank.test开头的域名，则说明该请求是来自自己网站的请求，是合法的。如果Referer是其他网站的话，就有可能是CSRF攻击，则拒绝该请求。在Java 中，拦截器是由Filter来实现的。

（3）在请求地址中添加token并验证

* CSRF攻击之所以能够成功，是因为攻击者可以伪造用户的请求，该请求中所有的用户验证信息都存在于Cookie中，因此攻击者可以在不知道这些验证信息的情况下直接利用用户自己的Cookie来通过安全验证。由此可知，抵御CSRF攻击的关键在于：在请求中放入攻击者所不能伪造的信息，并且该信息不存在于Cookie之中。因此可以在HTTP请求中以参数的形式加入一个随机产生的token，并在服务器端建立一个拦截器来验证这个token，如果请求中没有token或者token内容不正确，则认为可能是CSRF攻击而拒绝该请求。

（4）在HTTP头中自定义属性并验证

* 自定义属性的方法也是使用token并进行验证，和前一种方法不同的是，这里并不是把token以参数的形式置于HTTP请求之中，而是把它放到HTTP头中自定义的属性里。通过XMLHttpRequest这个类，可以一次性给所有该类请求加上csrftoken这个HTTP头属性，并把token值放入其中。这样解决了前一种方法在请求中加入token的不便，同时，通过这个类请求的地址不会被记录到浏览器的地址栏，也不用担心token会通过Referer泄露到其他网站。

（5）其他防御方法

* CSRF攻击是有条件的，当用户访问恶意链接时，认证的cookie仍然有效，所以当用户关闭页面时要及时清除认证cookie，对支持TAB模式（新标签打开网页）的浏览器尤为重要。
* 尽量少用或不要用request()类变量，获取参数时指定是request.form()还是request.querystring ()，这样有利于阻止CSRF漏洞攻击，此方法只不能完全防御CSRF攻击，只是一定程度上增加了攻击的难度。

## 如何防止中间人攻击？

* 对Https数据先自己进行加密（Ex：AES、RSA），再Https传输，这样即使截取到Https数据，也看不到明文。
* Https双向验证 ：服务器端对请求它的客户端要进行身份验证，客户端对自己所请求的服务器也会做身份验证。服务端一旦验证到请求自己的客户端为不可信任的，服务端就拒绝继续通信。客户端如果发现服务端为不可信任的，那么也中止通信。
* 强制使用SSL pinning：这是对抗MiTM（中间人）攻击的另一种方式。使用有效证书颁发机构提供的证书是第一步，它是通过返回的受信任的根证书以及是否与主机名匹配来验证该服务器提供的证书的有效性。通过SSL pinning可以验证客户端检查服务器证书的有效性。
* SSL/TLS 系列中有五种协议：SSL v2，SSL v3，TLS v1.0，TLS v1.1和TLS v1.2：
* SSL v2 是不安全的，不能使用。当与 HTTP一起使用时，SSL v3 是不安全的，也是过时的，不应该被使用。
* TLS v1.0 也是不应该使用的传统协议，但在实践中通常仍然是必需的。其主要弱点（BEAST）在现代浏览器中得到缓解，但其他问题仍然存在。
* TLS v1.1 和 v1.2 都没有已知的安全问题，只有 TLS v1.2 提供了现代的加密算法，因为它是唯一提供现代认证加密（也称为 AEAD）的版本。

## 怎样实现敏感词过滤？

在文字过滤算法中，DFA（有穷自动机）算法是一个比较高效的算法，它是通过event和当前的state得到下一个state，不断地进行状态的转换。

在实现的时候，可以通过查询实现状态的转换，通过S query U、V，通过U query V、P，通过V query U P。通过这样的转变我们可以将状态的转换转变为使用Java集合的查找。这样就可以将敏感词库构建成一颗一颗的树（类似于字典树），再判断一个词是否是敏感词就大大的减少了匹配的范围，比如我们要判断日本人，根据第一个字我们就可以确认需要检索的是哪棵树，然后再在这棵树中进行检索。

但是如何来判断一个敏感词已经结束了呢？利用标识位来判断。

可以使用hashmap来实现DFA算法，具体过程如下：加入查找的敏感词是“日本人”：

* + 1、在hashMap中查询“日”看其是否在hashMap中存在，如果不存在，则证明已“日”开头的敏感词还不存在，则我们直接构建这样的一棵树。判断该字是否为该词中的最后一个字。若是表示敏感词结束，设置标志位isEnd = 1，否则设置标志位isEnd = 0。
  + 2、如果在hashMap中查找到了，表明存在以“日”开头的敏感词，设置hashMap = hashMap.get(“日”)，跳至1，依次匹配“本”、“人”。

## 秒杀系统如何优化

（1）秒杀系统特点：

* 秒杀业务简单，卖家查询，买家下订单减库存。
* 秒杀时网站访问流量激增，出现峰值；
* 访问请求数量远大于实际需求量。

（2）整体优化思路：

* 限流：屏蔽掉无用的流量，允许少部分流量流向后端。
  + 令牌桶算法
    - 令牌桶算法(Token Bucket)和 Leaky Bucket 效果一样但方向相反的算法,更加容易理解.随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),如果桶已经满了就不再加了.新请求来临时,会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.
  + 漏桶算法
  + 漏桶(Leaky Bucket)算法思路很简单,水(请求)先进入到漏桶里,漏桶以一定的速度出水(接口有响应速率),当水流入速度过大会直接溢出(访问频率超过接口响应速率),然后就拒绝请求,可以看出漏桶算法能强行限制数据的传输速率
* 削峰：瞬时大流量峰值容易压垮系统，解决这个问题是重中之重。常用的消峰方法有异步处理、缓存和消息中间件等技术。
* 异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。
* 内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。
* 可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。
* 消息队列：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

（3）具体优化

* 中间代理层：可利用负载均衡（例如反响代理Nginx等）使用多个服务器并发处理请求，减小服务器压力。
* 控制层(网关层)：限制同一UserID访问频率：尽量拦截浏览器请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。
* 服务层：当用户量非常大的时候，拦截流量后的请求访问量还是非常大，此时仍需进一步优化。
  + 业务分离：将秒杀业务系统和其他业务分离，单独放在高配服务器上，可以集中资源对访问请求抗压。
  + 采用消息队列缓存请求：将大流量请求写到消息队列缓存，利用服务器根据自己的处理能力主动到消息缓存队列中抓取任务处理请求，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。
  + 利用缓存应对读请求：对于读多写少业务，大部分请求是查询请求，所以可以读写分离，利用缓存分担数据库压力。
  + 利用缓存应对写请求：缓存也是可以应对写请求的，可把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。
* 数据库层：数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。如果不使用缓存来作为中间缓冲而是直接访问数据库的话，可以对数据库进行优化，减少数据库压力。

（4）案例：利用消息中间件和Redis缓存实现

Redis是一个分布式缓存系统，支持多种数据结构，可利用Redis轻松实现一个强大的秒杀系统。

我们可以采用Redis 最简单的key-value数据结构，用一个原子类型的变量值(AtomicInteger)作为key，把用户id作为value，库存数量便是原子变量的最大值。对于每个用户的秒杀，我们使用RPUSH key value插入秒杀请求， 当插入的秒杀请求数达到上限时，停止所有后续插入。

然后我们可以再启动多个工作线程，使用LPOP key读取秒杀成功者的用户id，然后再操作数据库做最终的下订单减库存操作。

当然，上面Redis也可以替换成消息中间件RocketMQ，也可以将缓存和消息中间件组合起来，缓存系统负责接收记录用户请求，消息中间件负责将缓存中的请求同步到数据库。

## 如何解决高并发问题？（分布式锁）

（1）业务场景

所谓秒杀，从业务角度看，是短时间内多个用户“争抢”资源，这里的资源在大部分秒杀场景里是商品；将业务抽象，技术角度看，秒杀就是多个线程对资源进行操作，所以实现秒杀，就必须控制线程对资源的争抢，既要保证高效并发，也要保证操作的正确。

（2）可能的解决方式

* 秒杀在技术层面的抽象应该就是一个方法，在这个方法里可能的操作是将商品库存-1，将商品加入用户的购物车等等，在不考虑缓存的情况下应该是要操作数据库的。那么最简单直接的实现就是在这个方法上加上synchronized关键字，通俗的讲就是锁住整个方法；
* 锁住整个方法这个策略简单方便，但是似乎有点粗暴。可以稍微优化一下，只锁住秒杀的代码块，比如写数据库的部分；
* 既然有并发问题，那我就让他“不并发”，将所有的线程用一个队列管理起来，使之变成串行操作，自然不会有并发问题。

上面解决方式带来的问题：**锁的粒度比较大**，如果两个线程同时执行秒杀方法，这两个线程操作的是不同的商品,从业务上讲应该是可以同时进行的，但是如果采用第一二种方法，这两个线程也会去争抢同一个锁，这其实是不必要的。那么如何将锁控制在更细的粒度上呢？可以考虑为每个商品设置一个互斥锁，以和商品ID相关的字符串为唯一标识，这样就可以做到只有争抢同一件商品的线程互斥，不会导致所有的线程互斥。分布式锁恰好可以帮助我们解决这个问题。

（3）redis 分布式锁

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。分布式锁有 3 个重要的考量点：互斥（只能有一个客户端获取锁）；不能死锁；容错（只要大部分 redis 节点创建了这把锁就可以）；

第一个最普通的实现方式，就是在redis里使用 **setnx** 命令创建一个 key，这样就算加锁。执行这个命令就可以实现：使用SETNX key value，（通常value使用的是系统的时间戳）如果成功返回1，失败返回0；

当然上面可能出现的问题就是：如果一个持有锁的客户端失败或崩溃了不能释放锁，就会发生阻塞；可以通过锁的键对应的时间戳来判断这种情况是否发生了，如果当前的时间已经大于key对应的value的值，说明该锁已失效，可以被重新使用。不能通过简单的DEL删除锁，然后重新SETNX；因为会出现下面的问题：

|  |
| --- |
| C0操作超时了，但它还持有着锁，C1和C2读取key检查时间戳，先后发现超时了。  C1 发送DEL lock.foo  C1 发送SETNX key 并且成功了。  C2 发送DEL key  C2 发送SETNX key 并且成功了。  这样一来，C1，C2都拿到了锁！就出现问题了。 |

这种问题是可以避免的：

|  |
| --- |
| * C3发送SETNX key 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0 * C3发送GET key 以检查锁是否超时了，如果没超时，则等待或重试。反之，如果已超时，C3通过下面的操作来尝试获得锁：GETSET key <current Unix time + lock timeout + 1> * 通过GETSET（将给定的key的值设为value，并返回key的旧值），C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。 * 如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有获得锁，需要再次等待或重试。 |

## 订单处理流程

* 1首先，扣减服务作为下单流程的入口，会先对商品的库存做扣减。同样它会检查商品是否还有库存，如果库存足够，创建订单，否则返回失败；
* 2由于订单对应的操作步骤比较多，为了让流量变得平滑，这里使用消息队列MQ（为什么使用MQ？主要原因是由于在高并发的环境下，由于服务器来不及同步处理，请求往往会发生堵塞的现象，比如说大量的增加、修改（insert、updata）之类的请求同时到达mysql，直接导致无数的行锁，表锁、甚至最后请求堆积过多，从而触发too many connections错误。通过使用消息队列MQ，我们就可以异步处理请求，从而缓解系统的压力。）存放每个订单请求，等待订单处理服务完成具体业务。订单处理服务实现多线程，它们不断监听队列中的消息。一旦发现有新订单请求，就取出订单进行后续处理。
* 3订单处理服务处理完订单以后会把结果写到数据库。写数据库是 IO 操作，耗时长。所以，在写数据库的同时，会把结果先写入缓存中，这样用户是可以第一时间查询自己是否下单成功了。结果写入数据库，这个操作有可能成功也有可能失败，所以为了保证数据的最终一致性，我们用订单结果同步服务不断的对比缓存和数据库中的订单结果信息，一旦发现不一致，会去做重试操作。如果重试依旧不成功，会重写信息到缓存，让用户知道失败原因。

## 防止超卖

为了防止超卖现象，我会将这个库存信息放到缓存中。比如使用Redis存放库存信息，高并发的时候会出现资源争夺的情况，为了解决这个问题我们需要实现分布式锁：每一次用户去访问订单请求接口的时候，同时会去访问 Redis 中存放的库存信息，每接受用户一次请求，都会从 Redis 的库存中减去 1 个商品库存量。当任何一个进程访问 Redis 中的库存资源时，其他进程是不能访问的。

还需要考虑锁的释放问题：如果锁长期没有释放，需要考虑锁的过期时间，需要设置两个超时时间：

* 1资源本身的超时时间，一旦资源被使用一段时间还没有被释放，Redis 会自动释放掉该资源给其他服务使用。
* 2服务获取资源的超时时间，一旦一个服务获取资源一段时间后，不管该服务是否处理完这个资源，都需要释放该资源给其他服务使用。

## 项目中缓存是如何使用的？

（1）为什么要用缓存？

* 第一：Redis缓存实现session共享

首先我的项目中需要用户进行登录，如果只有一台服务器的话，直接使用session来存储用户的登录信息完全没问题，但是假如我的项目部署到两台或多台服务器上（serverA和serverB，两台都是主服务器，或者一台是主服务器，一台是备用服务器），在A服务器上登录后，并且采用的session存储是进程内存储的方式，在访问其他页面的情况下，有可能将我的请求转发到服务器B，但服务器B进程内并没有保存你的用户信息，在校验权限的时候，会将你重定向到登录页面。

解决方式：利用Redis统一管理session，服务器每次读写session的时候，都去访问Redis服务器。

* 第二：使用Redis做缓存主要有两个用途：**高性能、高并发**。
* **高性能**：假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作 mysql，半天查出来一个结果，耗时 600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？缓存啊，折腾 600ms 查出来的结果，扔缓存里，一个 key 对应一个 value，下次再有人查，别走 mysql了，直接从缓存里，通过一个 key 查出来一个 value，2ms 搞定。性能提升 300 倍。就是说对于一些需要复杂操作耗时查出来的结果，且确定后面不怎么变化，但是有很多读请求，那么直接将查询出来的结果放在缓存中，后面直接读缓存就好。
* **高并发**：mysql 这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql 单机支撑到 `2000QPS` 也开始容易报警了。所以要是你有个系统，高峰期一秒钟过来的请求有 1万，那一个 mysql 单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放 mysql。缓存功能简单，说白了就是 `key-value` 式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发 so easy。单机承载并发量是 mysql 单机的几十倍。（缓存是走内存的，内存天然就支撑高并发）。

（2）用了缓存之后会有什么不良后果？

常见的缓存问题有以下几个：

1. 缓存与数据库双写不一致；
2. 缓存雪崩、缓存穿透；
3. 缓存并发竞争；

* **缓存和数据库双写不一致**：
  + 最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。
    - **读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。**
    - **更新的时候，先更新数据库，然后再删除缓存**。

为什么是删除缓存，而不是更新缓存？

|  |
| --- |
| 原因很简单，很多时候，**在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值**。比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。  另外**更新缓存的代价有时候是很高的**。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于比较复杂的缓存数据计算的场景，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，这个缓存到底会不会被频繁访问到？  例如，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有**大量的冷数据**。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。用到缓存才去算缓存。  其实删除缓存，而不是更新缓存，就是一个 **lazy 计算**的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像 mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的 list，没有必要每次查询部门，把里面的 1000 个员工的数据也同时查出来。80% 的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询 1000 个员工。 |

* + 最初级的缓存不一致问题及解决方案

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。



解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

* + 比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据**，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样…

上面的情况只有在上亿流量高并发场景下才会发生，解决方式：

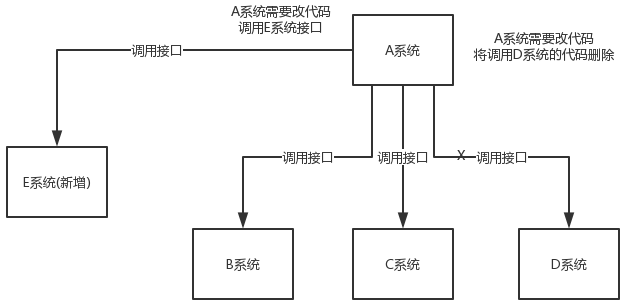
|  |
| --- |
| 更新数据的时候，根据**数据的唯一标识**，将操作路由之后，发送到一个jvm 内部队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个 jvm 内部队列中。  一个队列对应一个工作线程，每个工作线程**串行**拿到对应的操作，然后一条一条的执行。这样的话，一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，没有读到缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。  这里有一个**优化点**，一个队列中，其实**多个更新缓存请求串在一起是没意义的**，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。  待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中。  如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。 |

## 为什么使用消息队列？消息队列优点和缺点？

消息队列常见的使用场景有很多，但是比较核心的有 3 个：**解耦、异步、削峰**。

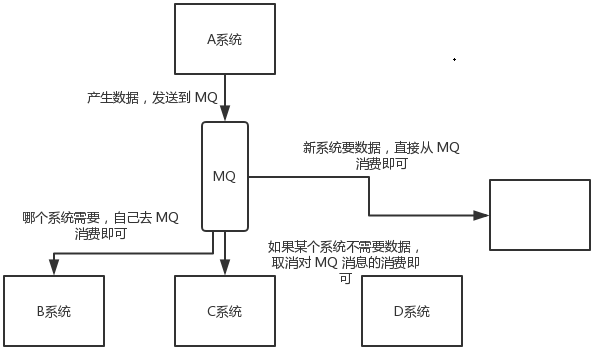
* 解耦

A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃......



在这个场景中，A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？

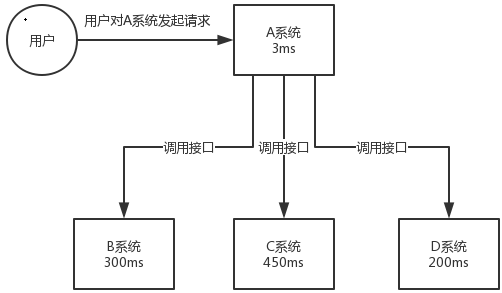
如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。



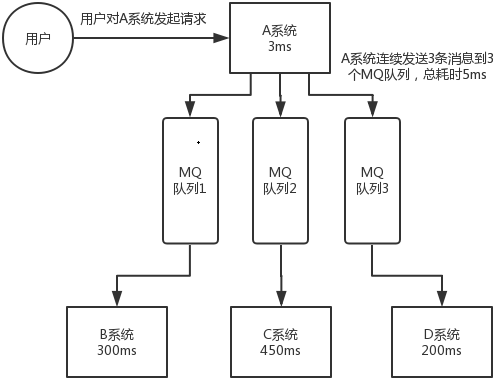
总结：通过一个 MQ，Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。

* 异步

A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求，等待个 1s，这几乎是不可接受的。



对于用户直接的操作，一般要求是每个请求都必须在 200 ms 以内完成，对用户几乎是无感知的。如果**使用 MQ**，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了。

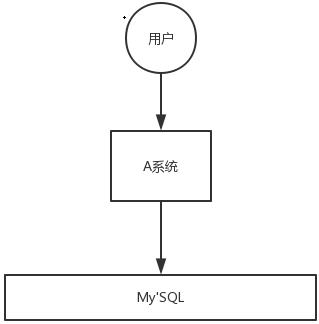


* 削峰

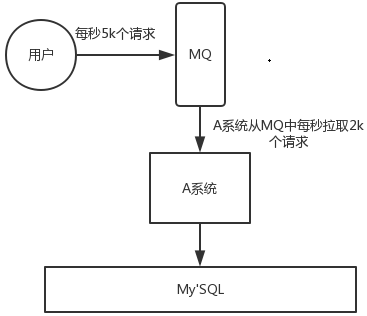
每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00 ~ 13:00 ，每秒并发请求数量突然会暴增到 5k+ 条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。

一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。



如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。



这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。

消息队列有什么缺点？

* 系统可用性降低：系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃，如何保证消息队列的高可用？
* 系统复杂度提高：加个MQ进来，怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？
* 一致性问题：A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，数据就不一致了。

## RocketMQ消息的可靠性？

所有消息队列都可能发生的问题：

（1）生产者发送消息后，队列未成功接收（网络原因或其他）而生产者不知情，消息丢失；

（2）生产者发送消息后，队列接收成功->生产者确认，但消息并未持久化，队列崩溃，消息丢失；

1.生产者可靠性保证

* rocketMQ提供了生产者消息发送确认的模式，比如在生产者发送消息后返回SendResult，如果isSuccess返回为true,则表示消息已经确认发送到服务器，并且被服务器接收存储。整个发送过程是一个同步的过程。保证消息送达服务器并返回结果。

2.broker可靠性保证

* meta服务器收到生产者发送的消息后，做必要的校验和检查之后，然后就将其写入磁盘，写入成功之后返回应答给生产者。因此，可以确认每条发送结果为成功的消息服务器都是写入磁盘的。 写入磁盘，不意味着数据落到磁盘设备上，毕竟我们还隔着一层os，os对写有缓冲。
* Meta有以下刷盘策略：异步刷盘（每1000条（可配置）或者每隔10秒、同步刷盘（需要设置存储配置上的groupCommitEnable选项为true，会在写入消息后，立即强制刷盘）。

2.消费者可靠性保证

* 消费者顺序消费，只有在成功消费一条消息后才会接着消费下一条。如果在消费某条消息失败（如异常），则会尝试重试消费这条消息（默认最大5次），超过最大次数后仍然无法消费，则将消息存储在消费者的本地磁盘，由后台线程继续做重试。而主线程继续往后走，消费后续的消息。因此， 只有在MessageListener确认成功消费一条消息后，meta的消费者才会继续消费另一条消息。由此来保证消息的可靠消费。
* 消费者的另一个可靠性的关键点是offset的存储，也就是拉取数据的偏移量。默认存储在zoopkeeper上，zookeeper通过集群来保证数据的安全性。Offset会定期保存，并且在每次重新负载均衡前都会强制保存一次，因此可能会存在极端情况下的消息的重复消费。

## JWT身份验证

+传统的身份验证：

（1）HTTP Basic Auth：就是每次请求API时都提供用户的username和password，简言之，Basic Auth是配合RESTful API 使用的最简单的认证方式，只需提供用户名密码即可，但是可能把用户名密码暴露给第三方客户端，使用的越来越少。因此，在开发对外开放的RESTful API时，尽量避免采用HTTP Basic Auth。

（2）OAuth（开放授权）：是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。

（3）Cookie Auth：为一次请求认证在服务端创建一个Session对象，同时在客户端的浏览器端创建了一个Cookie对象；通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie会被删除。但可以通过修改cookie 的expire time使cookie在一定时间内有效。

+基于 Token 的身份验证方法

使用基于 Token 的身份验证方法，大概的流程是这样的：

1）客户端使用用户名跟密码请求登录；

2）服务端收到请求，去验证用户名与密码；

3）验证成功后，服务端会签发一个 Token然后保存（缓存或者数据库），再把这个 Token 发送给客户端；

4）客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里；

5）客户端每次向服务端请求资源的时候需要带着服务端签发的 Token；

6）服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据；

Token机制相对于Cookie机制又有什么好处呢？

1）支持跨域访问：Cookie是不允许垮域访问的，这一点对Token机制是不存在的，前提是传输的用户认证信息通过HTTP头传输.

2）无状态：Token机制在服务端不需要存储session信息，因为Token 自身包含了所有登录用户的信息，只需要在客户端的cookie存储状态信息。

3）更适用CDN： 可以通过内容分发网络请求你服务端的所有资料（如：javascript，HTML,图片等），而你的服务端只要提供API即可.

4）去耦：不需要绑定到一个特定的身份验证方案。Token可以在任何地方生成，只要在你的API被调用的时候，你可以进行Token生成调用即可.

5）基于标准化：API可以采用标准化的 JSON Web Token (JWT)，这个标准已经存在多个后端库（.NET, Ruby, Java,Python, PHP）和多家公司的支持（如：Firebase,Google, Microsoft）.

+基于JWT的Token认证机制实现

JWT 标准的 Token 有三个部分，中间用点分隔开，并且都会使用 Base64 编码：

header（头部）：包含了使用的算法，这个 JWT 是不是带签名的或者加密的。主要就是说明一下怎么处理这个 JWT token 。比如一个加密的 JWT 里面要包含使用的加密的算法。唯一在头部里面要包含的是 alg 这个属性，如果是加密的 JWT，这个属性的值就是使用的签名或者解密用的算法。

payload（数据）：Payload 里面是 Token 的具体内容，这些内容里面有一些是标准字段，你也可以添加其它需要的内容。下面是标准字段：[iss：Issuer，发行者]，[sub：Subject，主题]，[aud：Audience，观众]，[exp：Expiration time，过期时间]，[signature（签名）]，

Signature：包含三个部分，先是用 Base64 编码的 header.payload ，再用加密算法加密一下，加密的时候要放进去一个 Secret ，这个相当于是一个密码，这个密码秘密地存储在服务端。

## 分布式系统

一些常问的点：

* + Dubbo 工作原理：服务注册、注册中心、消费者、代理通信、负载均衡；
  + 网络通信、序列化：dubbo 协议、长连接、NIO、hessian 序列化协议；
  + 负载均衡策略、集群容错策略、动态代理策略：dubbo 跑起来的时候一些功能是如何运转的？怎么做负载均衡？怎么做集群容错？怎么生成动态代理？
  + Dubbo SPI 机制：你了解不了解 dubbo 的 SPI 机制？如何基于 SPI 机制对 dubbo 进行扩展？

### 为什么将系统拆分，如何拆分？

（1）系统不拆分的缺点

一个大系统几十万行代码，维护起来非常困难，会造成代码冲突、系统异常、技术不敢随意升级、修改代码后全部重新测试；

（2）系统拆分的优点

将整个系统拆分成不同的服务，每个服务部署到单独的机器上，每个人都维护自己独立的代码，每次就发布自己负责的一个小服务就可以了；不同的服务可以进行技术升级，只要保证接口不变就行；没有代码冲突，开发效率提升。

（3）拆分后不使用Dubbo可以吗？

当然可以了，拆分后的各个系统之间直接基于SpringMVC，利用HTTP接口互相通信。但是这个肯定是有问题的，因为http接口通信维护起来成本很高，要考虑**超时重试**、**负载均衡**等各种问题，比如说订单系统调用商品系统，商品系统部署了5台机器，怎么把请求均匀地分给那5台机器？这不就是负载均衡？

Dubbo是一种RPC框架，就是说本地就是进行接口调用，但是Dubbo会代理这个调用请求，跟远程机器网络通信，处理掉负载均衡、服务实例上下线自动感知、超时重试等等问题。

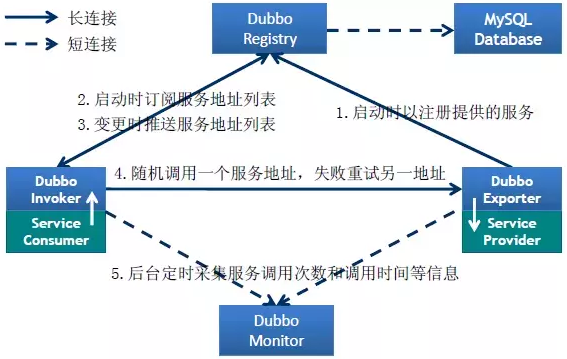
### 分布式CAP理论

一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。CAP理论中的CA和数据库事务中ACID的CA并完全是同一回事儿。两者之中的A都是C都是一致性(Consistency)。CAP中的A指的是可用性（Availability），而ACID中的A指的是原子性（Atomicity)，切勿混为一谈。

* 一致性指更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。
* 可用性指服务一直可用，而且是正常响应时间。对于一个可用性的分布式系统，每一个非故障的节点必须对每一个请求作出响应。所以，一般我们在衡量一个系统的可用性的时候，都是通过停机时间来计算的。
* 分区容错性指分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。分区容错性和扩展性紧密相关。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。好的分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。

### Dubbo的工作原理？

（1）Dubbo角色和设计是怎么样的，原理是怎么样的?

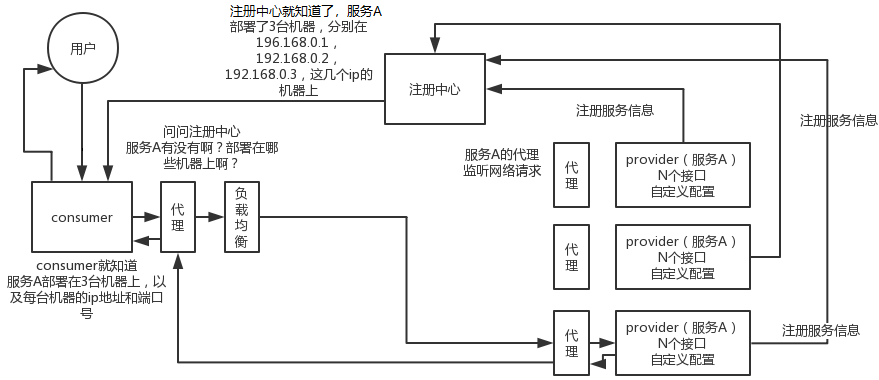


（2）dubbo 工作原理

* 第一层：service 层，接口层，给服务提供者和消费者来实现的
* 第二层：config 层，配置层，主要是对 dubbo 进行各种配置的
* 第三层：proxy 层，服务代理层，无论是 consumer 还是 provider，dubbo 都会给你生成代理，代理之间进行网络通信
* 第四层：registry 层，服务注册层，负责服务的注册与发现
* 第五层：cluster 层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务
* 第六层：monitor 层，监控层，对 rpc 接口的调用次数和调用时间进行监控
* 第七层：protocal 层，远程调用层，封装 rpc 调用
* 第八层：exchange 层，信息交换层，封装请求响应模式，同步转异步
* 第九层：transport 层，网络传输层，抽象 mina 和 netty 为统一接口
* 第十层：serialize 层，数据序列化层

（3）工作流程

* 第一步：provider 向注册中心去注册；
* 第二步：consumer 从注册中心订阅服务，注册中心会通知 consumer 已经注册的服务；
* 第三步：consumer 调用 provider；
* 第四步：consumer 和 provider 都异步通知监控中心；



（3）注册中心挂了可以继续通信吗？

答案是可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信。注册中心对等集群，任意一台宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复；

挂掉是不要紧的，但前提是你没有增加新的服务，如果你要调用新的服务，则是不能办到的。

### Dubbo支持哪些协议？PB？

在一个典型的RPC的使用场景中，包含了服务发现、负载、容错、序列化和网络传输等组件，其中RPC协议指明了程序如何进行序列化和网络传输，也就是说一个RPC协议的实现等于一个非透明的RPC调用。



简单来说，**分布式框架的核心是RPC框架，RPC框架的核心是RPC协议**。

序列化就是把数据结构或者是一些对象，转换为二进制串的过程；

反序列化是将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。

（1）dubbo协议

默认是dubbo协议，单个长连接，进行的是NIO异步通信，基于hessian作为序列化协议。底层是TCP协议。

使用的场景是：消费者个数多；传输数据量小（每次请求在 100kb 以内），但是并发量很高。

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就 100 个连接。然后后面直接基于长连接 NIO 异步通信，可以支撑高并发请求。

dubbo 支持的RPC协议列表



为什么 PB 的效率是最高的？

可能有一些同学比较习惯于JSON or XML数据存储格式，对于Protocol Buffer还比较陌生。Protocol Buffer其实是Google 出品的一种轻量并且高效的结构化数据存储格式，性能比JSON、XML要高很多。

其实PB之所以性能如此好，主要得益于两个：

1. 它使用proto编译器，自动进行序列化和反序列化，速度非常快，比XML和JSON快上了 20~100倍；
2. 它的数据压缩效果好，序列化后的数据量体积小。因为体积小，传输起来带宽和速度上会有优化。

### Dubbo在安全机制方面是如何解决的？

Dubbo通过Token令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo还提供服务黑白名单，来控制服务所允许的调用方。

### dubbo协议为什么不能传大包？

因dubbo协议采用单一长连接，如果每次请求的数据包大小为500KByte，假设网络为千兆网卡(1024Mbit=128MByte)，每条连接最大7MByte(不同的环境可能不一样，供参考)，单个服务提供者的TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

### dubbo协议为什么采用异步单一长连接？

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如Morgan的提供者只有6台提供者，却有上百台消费者，每天有1.5亿次调用，如果采用常规的hessian服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步IO，复用线程池，防止C10K问题。

【C10K问题：

C10K问题的本质上是操作系统的问题。对于Web 1.0/2.0时代的操作系统，传统的同步阻塞I/O模型处理方式都是requests per second。当创建的进程或线程多了，数据拷贝频繁（缓存I/O、内核将数据拷贝到用户进程空间、阻塞），进程/线程上下文切换消耗大， 导致操作系统崩溃，这就是C10K问题的本质。可见，解决C10K问题的关键就是尽可能减少这些CPU资源消耗。

从网络编程技术的角度来说：

* 每个连接分配一个独立的线程/进程；
* 同一个线程/进程同时处理多个连接（Select poll epoll）；

】

### Dubbo的服务发布作了几件事？

* 暴露本地服务
* 暴露远程服务
* 启动netty
* 连接zookeeper
* 到zookeeper注册
* 监听zookeeper

### 服务提供者能实现失效踢出是什么原理(高频题)

在分布式系统中，我们常常需要知道某个机器是否可用，传统的开发中，可以通过Ping某个主机来实现，Ping得通说明对方是可用的，相反是不可用的；ZK中我们让所有的机器都注册一个临时节点，我们判断一个机器是否可用，只需要判断这个节点在ZK中是否存在就可以了，不需要直接去连接需要检查的机器，降低系统的复杂度。

### Dubbo负载均衡策略和集群容错策略都有哪些？动态代理策略呢？

（1）dubbo 负载均衡策略

* random loadbalance：默认情况下，dubbo是 random loadbalance ，即随机调用实现负载均衡，可以对 provider 不同实例设置不同的权重，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。
* roundrobin loadbalance：就是均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

举个栗子：跟运维同学申请机器，有的时候，我们运气好，正好公司资源比较充足，刚刚有一批热气腾腾、刚刚做好的虚拟机新鲜出炉，配置都比较高：8 核 + 16G 机器，申请到 2 台。过了一段时间，我们感觉 2 台机器有点不太够，我就去找运维同学说，“哥儿们，你能不能再给我一台机器”，但是这时只剩下一台 4 核 + 8G 的机器。我要还是得要。这个时候，可以给两台 8 核 16G 的机器设置权重 4，给剩余 1 台 4 核 8G 的机器设置权重 2。<dubbo:service interface= “…” loadbalance=”roundrobin” />

* leastactive loadbalance：自动感知，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。
* consistanthash loadbalance：一致性 Hash 算法，相同参数的请求一定分发到一个 provider 上去，provider 挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性 Hash 策略。

（2）dubbo 集群容错策略

* failover cluster 模式：失败自动切换，自动重试其他机器，默认就是这个，常见于读操作。（失败重试其它机器），可以通过以下几种方式配置重试次数：<dubbo:service retries="2" />
* failfast cluster 模式：一次调用失败就立即失败，常见于非幂等性的写操作，比如新增一条记录（调用失败就立即失败）。
* failsafe cluster 模式：出现异常时忽略掉，常用于不重要的接口调用，比如记录日志。配置示例如下：<dubbo:service cluster="failsafe" />
* failback cluster 模式：失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种。
* forking cluster 模式：并行调用多个 provider，只要一个成功就立即返回。常用于实时性要求比较高的读操作，但是会浪费更多的服务资源，可通过 forks="2" 来设置最大并行数。
* broadcacst cluster：逐个调用所有的 provider。任何一个 provider 出错则报错（从2.1.0版本开始支持）。通常用于通知所有提供者更新缓存或日志等本地资源信息。

（3）dubbo动态代理策略

默认使用 javassist 动态字节码生成，创建代理类。但是可以通过 spi 扩展机制配置自己的动态代理策略。

### 怎么通过dubbo实现服务降级的，降级的方式有哪些，又有什么区别？

当网站处于高峰期时，并发量大，服务能力有限，那么我们只能暂时屏蔽边缘业务，这里面就要采用服务降级策略了。首先dubbo中的服务降级分成两个：屏蔽(mock=force)、容错(mock=fail)。

* mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
* mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

要生效需要在dubbo后台进行配置的修改。

## Zookeeper

### zookeeper是什么？为什么使用？优点？

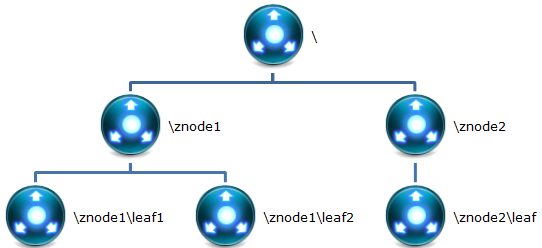
* 定义：Zookeeper 是 Google 的 Chubby一个开源的实现，是 Hadoop 的分布式协调服务；它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等；
* 为什么使用？
  + 大部分分布式应用需要一个主控、协调器或控制器来管理物理分布的子进程（如资源、任务分配等）；
  + 目前，大部分应用需要开发私有的协调程序，缺乏一个通用的机制；
  + 协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器；
  + Keepalived可以提供通用的分布式锁服务，用以协调分布式应用，但是：Keepalived监控节点不好管理；Keepalive 采用优先级监控；没有协同工作；功能单一；Keepalive可扩展性差。
* 优点？

|  |  |
| --- | --- |
| 最终一致性 | 为客户端展示同一个视图，这是非常重要的功能 |
| 可靠性 | 1如果消息被一台服务器接受，那么它将被所有的服务器接受；  2集群中只要有半数以上节点存活，Zookeeper集群就能正常服务 |
| 实时性 | Zookeeper不能保证两个客户端能同时得到刚刚更新的数据，如果需要最新的数据，应该在读数据之前调用sync()接口。 |
| 独立性 | 各个client之间互不干预 |
| 原子性 | 更新只能成功或失败，没有中间状态 |
| 顺序性 | 所有server,，同一消息发布顺序一致 |

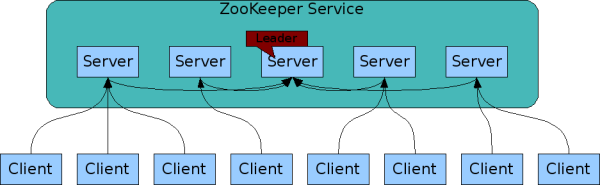
* 数据结构？

ZooKeeper数据模型的结构与Unix文件系统很类似，整体上可看作是**一棵树**，每个节点称作一个**ZNode**。

* 每个 ZNode默认能存储**1MB**数据，每个ZNode都可以**通过其路径唯一标识**。
* 可以使用zkCli命令，登录到zookeeper上，并通过ls、create、delete、sync等命令操作这些znode节点；
* znode除了名称、数据以外，还有一套属性：zxid。这套zid与时间戳对应，记录zid不同的状态；



* 工作原理？
  + 1.每个Server在内存中存储了一份数据；
  + 2.Zookeeper启动时，将从实例中选举一个leader（Paxos协议）；
  + 3.Leader负责处理数据更新等操作；
  + 4.一个更新操作成功，当且仅当大多数Server在内存中成功修改数据。



### 判断Zookeeper节点是持久节点还是临时节点？

* zookeeper持久节点：该数据节点被创建后，就会一直存在于zookeeper服务器上，直到有删除操作来主动删除这个节点。
* zookeeper临时节点：临时节点的生命周期和客户端会话绑定在一起，客户端会话失效，则这个节点就会被自动清除。

创建节点的完成之后，使用get命令查看节点的State数据结构信息，其中有一个字段是ephemeralOwner意思是这个节点的临时拥有者：

* 当ephemeralOwner 值不为0时，表明这个节点是临时节点，值为会话id。
* 当ephemeralOwner 值为0时，表明这个节点是持久节点。

可以给节点加一个watcher（监控）监控节点，如果挂掉了可以重新建立会话；也可以使用高级开发包curator，可以创建一个client不主动close，session就能一直保持。而且内部还有重连机制等等。

### zookeeper的选举机制？

所谓的zookeeper容错是指，当宕掉几个zookeeper服务器之后，剩下的个数必须大于宕掉的个数，也就是剩下的服务数必须大于n/2，zookeeper才可以继续使用，无论奇偶数都可以选举leader。5台机器最多宕掉2台，还可以继续使用，因为剩下3台大于5/2。

说为什么最好为奇数个，是在以最大容错服务器个数的条件下，会节省资源，比如，最大容错为2的情况下，对应的zookeeper服务数，奇数为5，而偶数为6，也就是6个zookeeper服务的情况下最多能宕掉2个服务，所以从节约资源的角度看，没必要部署6（偶数）个zookeeper服务。

A 选举条件

Leader选举是保证分布式数据一致性的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

1、服务器初始化启动。

若进行Leader选举，则至少需要两台机器，这里选取3台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器Server1启动时，其单独无法进行和完成Leader选举，当第二台服务器Server2启动时，此时两台机器可以相互通信，每台机器都试图找到Leader，于是进入Leader选举过程。选举过程如下：

* **1）每个Server发出一个投票**。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。
* **2）接受来自各个服务器的投票。**集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器。
* **3）处理投票。**针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下：
* 优先检查ZXID。ZXID比较大的服务器优先作为Leader。
* 如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器。

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0，再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

* **4）统计投票。**每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了Leader。
* **5）改变服务器状态。**一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING。

2、服务器运行期间无法和Leader保持连接。

在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举，其过程和启动时期的Leader选举过程基本一致。假设正在运行的有Server1、Server2、Server3三台服务器，当前Leader是Server2，若某一时刻Leader挂了，此时便开始Leader选举。选举过程如下：

* 1）变更状态。Leader挂后，余下的非Observer服务器都会讲自己的服务器状态变更为LOOKING，然后开始进入Leader选举过程。
* 2）每个Server会发出一个投票。在运行期间，每个服务器上的ZXID可能不同，此时假定Server1的ZXID为123，Server3的ZXID为122；在第一轮投票中，Server1和Server3都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。
* 3）接收来自各个服务器的投票。与启动时过程相同。
* 4）处理投票。与启动时过程相同，此时，Server1将会成为Leader。
* 5）统计投票。与启动时过程相同。
* 6）改变服务器的状态。与启动时过程相同。

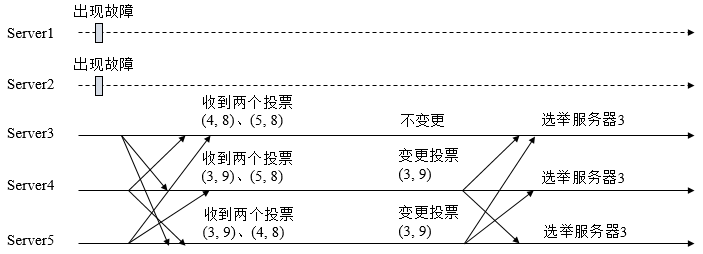
B Leader选举算法分析（FastLeaderElection选举算法）

当一台机器进入Leader选举时，当前集群可能会处于以下两种状态：

* 集群中已经存在Leader。
* 此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器而言，仅仅需要和Leader机器建立起连接，并进行状态同步即可。
* 集群中不存在Leader。
* **（1）第一次投票**。无论哪种导致进行Leader选举，集群的所有机器都处于试图选举出一个Leader的状态，即LOOKING状态，LOOKING机器会向所有其他机器发送消息，该消息称为投票。投票中包含了SID（服务器的唯一标识）和ZXID（事务ID），(SID, ZXID)形式来标识一次投票信息。假定Zookeeper由5台机器组成，SID分别为1、2、3、4、5，ZXID分别为9、9、9、8、8，并且此时SID为2的机器是Leader机器，某一时刻，1、2所在机器出现故障，因此集群开始进行Leader选举。在第一次投票时，每台机器都会将自己作为投票对象，于是SID为3、4、5的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。
* **（2）变更投票**。每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个Leader选举算法的核心所在，其中术语描述如下：
* vote\_sid：接收到的投票中所推举Leader服务器的SID。
* vote\_zxid：接收到的投票中所推举Leader服务器的ZXID。
* self\_sid：当前服务器自己的SID。
* self\_zxid：当前服务器自己的ZXID。

每次对收到的投票的处理，都是对(vote\_sid, vote\_zxid)和(self\_sid, self\_zxid)对比的过程：

* 规则一：如果vote\_zxid大于self\_zxid，就认可当前收到的投票，并再次将该投票发送出去。
* 规则二：如果vote\_zxid小于self\_zxid，那么坚持自己的投票，不做任何变更。
* 规则三：如果vote\_zxid等于self\_zxid，那么就对比两者的SID，如果vote\_sid大于self\_sid，那么就认可当前收到的投票，并再次将该投票发送出去。
* 规则四：如果vote\_zxid等于self\_zxid，并且vote\_sid小于self\_sid，那么坚持自己的投票，不做任何变更。



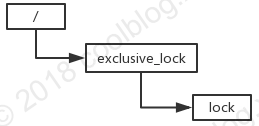
* **（3）确定Leader**。经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的SID机器即为Leader。此时Server3将成为Leader。

由上面规则可知，通常那台服务器上的数据越新（ZXID会越大），其成为Leader的可能性越大，也就越能够保证数据的恢复。如果ZXID相同，则SID越大机会越大。

### Zookeeper实现分布式锁？

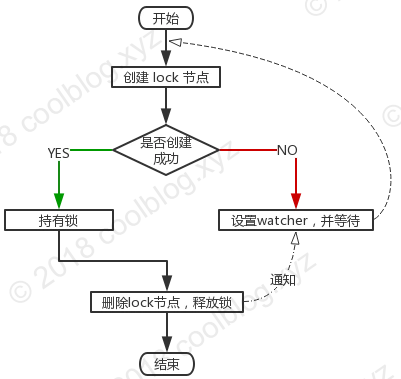
（1）独占锁

独占锁又称排它锁，从字面意思上很容易理解他们的用途。即如果某个操作 O1 对访问资源 R1 的过程加锁，在操作 O1 结束对资源 R1 访问前，其他操作不允许访问资源 R1。



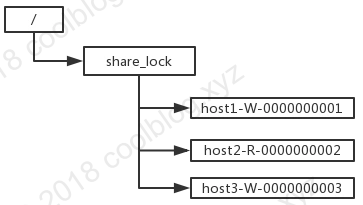
如上图，对于独占锁，可以将资源 R1 看做是 lock 节点，操作 O1 访问资源 R1 看做创建 lock 节点，释放资源 R1 看做删除 lock 节点。这样我们就将独占锁的定义对应于具体的 Zookeeper 节点结构，通过创建 lock 节点获取锁，删除节点释放锁。详细的过程如下：

* 多个客户端竞争创建 lock 临时节点
* 其中某个客户端成功创建 lock 节点，其他客户端对 lock 节点设置 watcher
* 持有锁的客户端删除 lock 节点或该客户端崩溃，由 Zookeeper 删除 lock 节点
* 其他客户端获得 lock 节点被删除的通知
* 重复上述4个步骤，直至无客户端在等待获取锁了



（2）读写锁

读写锁包含一个读锁和写锁，操作 O1 对资源 R1 加读锁，且获得了锁，其他操作可同时对资源 R1 设置读锁，进行共享读操作。如果操作 O1 对资源 R1 加写锁，且获得了锁，其他操作再对资源 R1 设置不同类型的锁都会被阻塞。总结来说，读锁具有共享性，而写锁具有排他性。

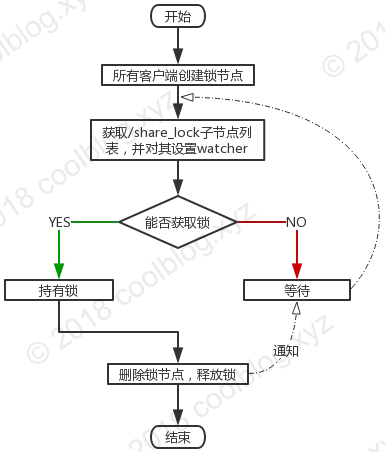


在 Zookeeper 中，由于读写锁和独占锁的节点结构不同，读写锁的客户端不用再去竞争创建 lock 节点。所以在一开始，所有的客户端都会创建自己的锁节点。如果不出意外，所有的锁节点都能被创建成功，此时锁节点结构如上图所示。之后，客户端从 Zookeeper 端获取 /share\_lock 下所有的子节点，并判断自己能否获取锁。

* 如果客户端创建的是读锁节点，获取锁的条件（满足其中一个即可）如下：
  + 自己创建的节点序号排在所有其他子节点前面
  + 自己创建的节点前面无写锁节点
* 如果客户端创建的是写锁节点，由于写锁具有排他性。所以获取锁的条件要简单一些，只需确定自己创建的锁节点是否排在其他子节点前面即可。
* 读写锁第一种实现

第一种实现是对 /share\_lock 节点设置 watcher，当 /share\_lock 下的子节点被删除时，未获取锁的客户端收到 /share\_lock 子节点变动的通知。在收到通知后，客户端重新判断自己创建的子节点是否可以获取锁，如果失败，再次等待通知。详细流程如下：

* 所有客户端创建自己的锁节点
* 从 Zookeeper 端获取 /share\_lock 下所有的子节点，并对 /share\_lock 节点设置 watcher
* 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。否则继续等待
* 持有锁的客户端删除自己的锁节点，其他客户端收到 /share\_lock 子节点变动的通知
* 重复步骤2、3、4，直至无客户端在等待获取锁了

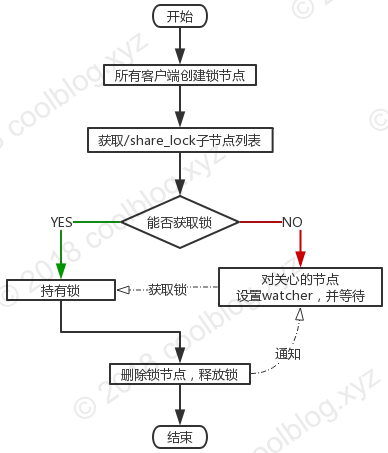


上面获取读写锁流程并不复杂，但却存在性能问题。以图3所示锁节点结构为例，第一个锁节点 host1-W-0000000001 被移除后，Zookeeper 会将 /share\_lock 子节点变动的通知分发给所有的客户端。但实际上，该子节点变动通知除了能影响 host2-R-0000000002 节点对应的客户端外，分发给其他客户端则是在做无用功，因为其他客户端即使获取了通知也无法获取锁。所以这里需要做一些优化，优化措施是让客户端只在自己关心的节点被删除时，再去获取锁。

* 读写锁的第二种实现

在了解读写锁第一种实现的弊端后，我们针对这一实现进行优化。这里客户端不再对 /share\_lock 节点进行监视，而只对自己关心的节点进行监视。还是以图3的锁节点结构进行举例说明，host2-R-0000000002 对应的客户端 C2 只需监视 host1-W-0000000001 节点是否被删除即可。而 host3-W-0000000003 对应的客户端 C3 只需监视 host2-R-0000000002 节点是否被删除即可，只有 host2-R-0000000002 节点被删除，客户端 C3 才能获取锁。而 host1-W-0000000001 节点被删除时，产生的通知对于客户端 C3 来说是无用的，即使客户端 C3 响应了通知也没法获取锁。这里总结一下，不同客户端关心的锁节点是不同的。如果客户端创建的是读锁节点，那么客户端只需找出比读锁节点序号小的最后一个的写锁节点，并设置 watcher 即可。而如果是写锁节点，则更简单，客户端仅需对该节点的上一个节点设置 watcher 即可。详细的流程如下：

* 所有客户端创建自己的锁节点
* 从 Zookeeper 端获取 /share\_lock 下所有的子节点
* 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。否则对自己关心的锁节点设置 watcher
* 持有锁的客户端删除自己的锁节点，某个客户端收到该节点被删除的通知，并获取锁
* 重复步骤4，直至无客户端在等待获取锁了



（4）zookeeper分布式锁

第一种方式

zk 分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时 znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能**注册个监听器**监听这个锁。释放锁就是删除这个 znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新加锁。

第二种方式：创建临时顺序节点：

如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁；后面的每个人都会去监听排在自己前面的那个人创建的 node 上，一旦某个人释放了锁，排在自己后面的人就会被 zookeeper 给通知，一旦被通知了之后，就 ok 了，自己就获取到了锁，就可以执行代码了。

（5）redis 分布式锁和zk分布式锁的对比

* redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。
* zk 分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。
* 如果是 redis 获取锁的那个客户端 出现 bug 挂了，那么只能等待超时时间之后才能释放锁；而 zk 的话，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动释放锁。

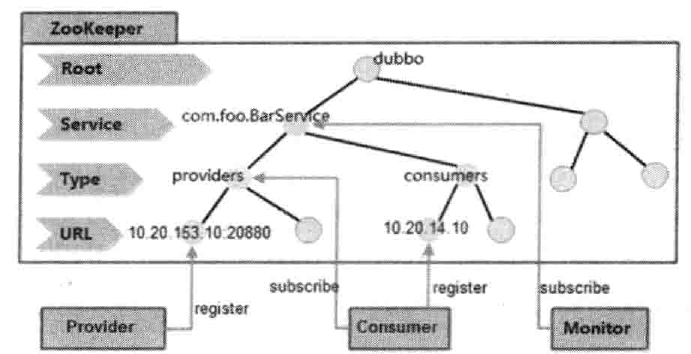
### Zookeeper实现服务注册中心？

1 ZooKeeper中的节点

ZooKeeper是一个树形结构的目录服务，支持变更推送，因此非常适合作为Dubbo服务的注册中心。

|  |
| --- |
| * 在ZooKeeper中，节点分为两类，第一类是指构成集群的机器，称为机器节点；第二类是指数据模型中的数据单元，称为数据节点ZNode。ZooKeeper将所有数据存储在内存中，数据模型是一棵树(ZNode Tree)，由斜杠（/）进行分割的路径，就是一个ZNode。每个ZNode上都会保存自己的数据内容，同时还会保存一系列属性信息。 * 在ZooKeeper中，Znode可分为持久节点和临时节点两类，所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在ZooKeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。 |

2基于ZooKeeper实现的注册中心节点结构示意图：



* /dubbo：这是dubbo在ZooKeeper上创建的根节点；
* /dubbo/com.foo.BarService：这是服务节点，代表了Dubbo的一个服务；
* /dubbo/com.foo.BarService/providers：服务提供者的根节点，其子节点代表了每一个服务真正的提供者；
* /dubbo/com.foo.BarService/consumers：服务消费者的根节点，其子节点代表每一个服务真正的消费者；

3 注册中心的工作流程

* 服务提供方启动
* 服务提供者在启动的时候，会在ZooKeeper上注册服务。所谓注册服务，其实就是在ZooKeeper的/dubbo/com.foo.BarService/providers节点下创建一个子节点，并写入自己的URL地址，这就代表了com.foo.BarService这个服务的一个提供者。
* 服务消费者启动
* 服务消费者在启动的时候，会向ZooKeeper注册中心订阅自己的服务。其实，就是读取并订阅ZooKeeper上/dubbo/com.foo.BarService/providers节点下的所有子节点，并解析出所有提供者的URL地址来作为该服务地址列表。
* 同时，服务消费者还会在ZooKeeper的/dubbo/com.foo.BarService/consumers节点下创建一个临时节点，并写入自己的URL地址，这就代表了com.foo.BarService这个服务的一个消费者。
* 消费者远程调用提供者
* 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一个提供者进行调用，如果调用失败，再选另一个提供者调用。
* 增加服务提供者
* 增加提供者，也就是在providers下面新建子节点。一旦服务提供方有变动，zookeeper就会把最新的服务列表推送给消费者。
* 减少服务提供者
* **所有提供者在ZooKeeper上创建的节点都是临时节点**，利用的是临时节点的生命周期和客户端会话相关的特性，因此一旦提供者所在的机器出现故障导致该提供者无法对外提供服务时，该临时节点就会自动从ZooKeeper上删除，同样，zookeeper会把最新的服务列表推送给消费者。
* ZooKeeper宕机之后
* 消费者每次调用服务提供方是不经过ZooKeeper的，消费者只是从zookeeper那里获取服务提供方地址列表。所以当zookeeper宕机之后，不会影响消费者调用服务提供者，影响的是zookeeper宕机之后如果提供者有变动，增加或者减少，无法把最新的服务提供者地址列表推送给消费者，所以消费者感知不到。

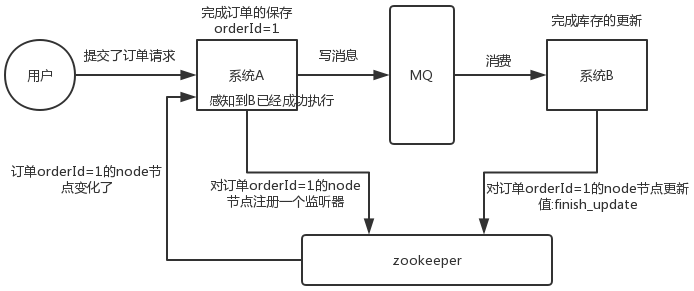
### zookeeper都有哪些使用场景？

zookeeper 是分布式系统中很常见的一个基础系统，zookeeper 的使用场景如下：

* 分布式协调；
* 分布式锁；
* 元数据/配置信息管理；
* HA高可用性；

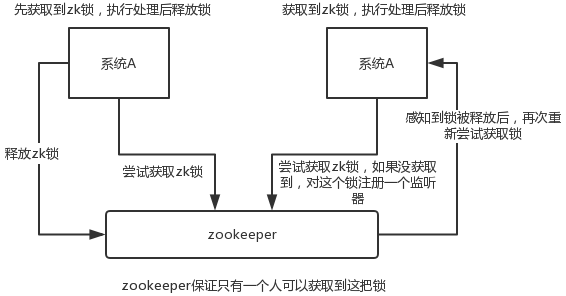
（1）分布式协调

这个其实是 zookeeper很经典的一个用法，简单来说，就好比，你 A 系统发送个请求到 mq，然后 B 系统消息消费之后处理了。那A系统如何知道 B 系统的处理结果？用 zookeeper 就可以实现分布式系统之间的协调工作。A系统发送请求之后可以在zookeeper上**对某个节点的值注册个监听器**，一旦B系统处理完了就修改zookeeper那个节点的值，A 系统立马就可以收到通知，完美解决。



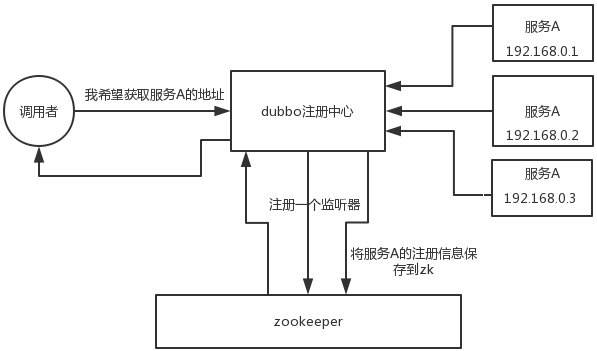
（2）分布式锁（见7.9 的分布式锁实现方式）【项目中使用的】

对某一个数据连续发出两个修改操作，两台机器同时收到了请求，但是只能一台机器先执行完另外一个机器再执行。那么此时就可以使用 zookeeper 分布式锁，一个机器接收到了请求之后先获取 zookeeper 上的一把分布式锁，就是可以去创建一个 znode，接着执行操作；然后另外一个机器也**尝试去创建**那个 znode，结果发现自己创建不了，因为被别人创建了，那只能等着，等第一个机器执行完了自己再执行。



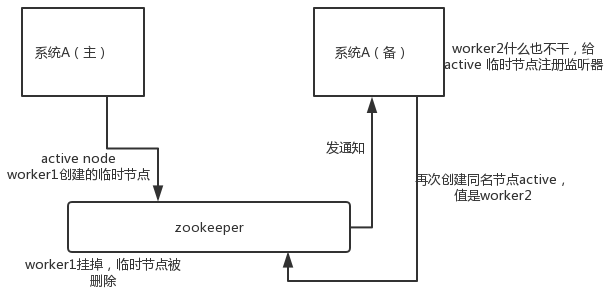
（3）元数据/配置信息管理

zookeeper 可以用作很多系统的配置信息的管理，比如 kafka、storm 等等很多分布式系统都会选用 zookeeper 来做一些元数据、配置信息的管理，包括 **dubbo注册中心不也支持zookeeper** 么？



（4）HA高可用性

这个应该是很常见的，比如 hadoop、hdfs、yarn 等很多大数据系统，都选择基于 zookeeper 来开发 HA 高可用机制，就是一个重要进程一般会做主备两个，主进程挂了立马通过 zookeeper 感知到切换到备用进程。



# 编程及算法

## HashMap排序题

已知一个 HashMap<Integer，User>集合，User有name（String）和age（int）属性。请写一个方法实现对HashMap 的排序功能，该方法接收HashMap<Integer，User>为形参，返回类型为HashMap<Integer，User>，要求对 HashMap 中的User的age倒序进行排序。排序时key=value键值对不得拆散。

|  |
| --- |
| **public class** FunClass {  **public static void** main(String[] args) {  HashMap<Integer, User> users = **new** HashMap<>();  users.put(1, **new** User(**"张三"**, 25));  users.put(3, **new** User(**"李四"**, 22));  users.put(2, **new** User(**"王五"**, 28));  System.***out***.println(users);  HashMap<Integer,User> sortHashMap = *sortHashMap*(users);  System.***out***.println(sortHashMap);  **/\*\*  控制台输出内容  {1=User [name=张三, age=25], 2=User [name=王五, age=28], 3=User [name=李四, age=22]}  {2=User [name=王五, age=28], 1=User [name=张三, age=25], 3=User [name=李四, age=22]}  \*/** }  **public static** HashMap<Integer, User> sortHashMap(HashMap<Integer, User> map) {  **// 首先拿到 map 的键值对集合** Set<Map.Entry<Integer, User>> entrySet = map.entrySet();  **// 将 set 集合转为 List 集合，为什么，为了使用工具类的排序方法** List<Map.Entry<Integer, User>> list = **new** ArrayList<>(entrySet);  **// 使用 Collections 集合工具类对 list 进行排序，排序规则使用匿名内部类来实现** Collections.*sort*(list, **new** Comparator<Map.Entry<Integer, User>>(){  @Override  **public int** compare(Map.Entry<Integer, User> o1, Map.Entry<Integer, User> o2) {  **return** o2.getValue().getAge() - o1.getValue().getAge();  }  });  **//创建一个新的有序的 HashMap 子类的集合** LinkedHashMap <Integer, User> linkedHashMap = **new** LinkedHashMap<>();  **//将 List 中的数据存储在 LinkedHashMap 中  for** (Map.Entry<Integer, User> entry : list) {  linkedHashMap.put(entry.getKey(), entry.getValue());  }  **return** linkedHashMap;  } } |

## 递归输出文件目录中以.txt结尾的文件

|  |
| --- |
| **public class** FindFiles {  **public static void** main(String[] args) {  List<File> list = **new** ArrayList<>();  String strPath = **"E:/aaa"**;  List<File> fileList = *getFileList*(strPath, list);  System.***out***.println(fileList.toString());  **//结果就是：[E:\aaa\1\2\a.txt, E:\aaa\1\b.txt, E:\aaa\1\c.txt, E:\aaa\d.txt]** }  **/\*\*  \* getFileList  \* @param *strPath* :文件夹路径  \* @param *list* : 结果集合  \* @return java.util.List**<**java.io.File**>  **\*/  public static** List<File> getFileList(String strPath,List<File> list) {  File dir = **new** File(strPath);**//由strPath创建File对象** File[] files = dir.listFiles();**//获取所有的目录数组files  if** (files != **null**) {  **for** (**int** i = 0; i < files.**length**; i++) {  **if** (files[i].isDirectory()) {**//判断file是不是文件夹** *getFileList*(files[i].getAbsolutePath(),list);  }**else**{  **if** (files[i].getName().endsWith(**".txt"**)) {**//判断文件是不是以.txt结尾** list.add(files[i]);  }  }  }  }  **return** list;  } } |

## 缓存算法实现（手写LRU）？

LRU算法的设计原则是：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

它支持两个操作：

* get(key)：如果key在cache中，则返回对应的value值，否则返回-1；
* set(key,value)：如果key不在cache中，则将该(key,value)插入cache中（注意，如果cache已满，则必须把最近最久未使用的元素从cache中删除）；如果key在cache中，则重置value的值。

|  |
| --- |
| 实例LeetCode上的146题（<https://leetcode.com/problems/lru-cache/>）：  LRUCache cache = new LRUCache( 2);//缓存容量为2  cache.put(1, 1);  cache.put(2, 2);  cache.get(1); // 返回 1  cache.put(3, 3); // 该操作会使得密钥 2 作废  cache.get(2); // 返回 -1 (未找到)  cache.put(4, 4); // 该操作会使得密钥 1 作废  cache.get(1); // 返回 -1 (未找到)  cache.get(3); // 返回 3  cache.get(4); // 返回 4 |

实现方式：利用链表和hashmap。

当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。



|  |
| --- |
| /\*\*首先定义一个Node节点\*/  public class Node {  int key;  int val;  Node next;  Node prev;  public Node(int key, int value) {  this.key = key;  this.val = value;  next = null;  prev = null;  }  }  /\*\*具体的实现\*/  public class LRUCache {  final Node head = new Node(0, 0);  final Node tail = new Node(0, 0);  final Map<Integer, Node> map;  final int capacity;  public LRUCache(int capacity) {//初始化cache  this.capacity = capacity;  map = new HashMap<>(capacity);  head.next = tail;  tail.prev = head;  }  public int get(int key) {  int res = -1;  if(map.containsKey(key)){//map中包含查找的key  Node n = map.get(key);//取出key对应的Node  remove(n);//移除Node  insertToHead(n);//将查到的Node加入到链表的首部  res = n.val;//返回查到的key对应的Node中的值  }  return res;  }  public void put(int key, int value) {//插入元素  if(map.containsKey(key)){  Node n = map.get(key);  remove(n);  n.val = value;  insertToHead(n);  } else {  if(map.size() == capacity){//当前cache容量达到定义的值  map.remove(tail.prev.key);//移除链表尾部的map值  remove(tail.prev);//移除移除链表尾部的Node  }  Node n = new Node(key, value);  insertToHead(n);  map.put(key, n);  }  }  private void remove(Node n){//移除 Node n 的方法  n.prev.next = n.next;  n.next.prev = n.prev;  }  private void insertToHead(Node n){//将节点插入链表的首部  Node headNext = head.next;  head.next = n;  headNext.prev = n;  n.prev = head;  n.next = headNext;  }  } |

也可以使用JDK自带的数据结构实现：

|  |
| --- |
| public class LRUCache\_JDK<K,V> extends LinkedHashMap<K,V> {  private int CACHE\_SIZE;  public LRUCache\_JDK(int cacheSize){  super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);  CACHE\_SIZE = cacheSize;  }  @Override  protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {  return size() > CACHE\_SIZE;  }  } |

## 排序算法

### 排序算法术语

* 稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
* 不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；
* 内排序：所有排序操作都在内存中完成；
* 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
* 时间复杂度： 一个算法执行所耗费的时间。
* 空间复杂度：运行完一个程序所需内存的大小。

### 算法总结



n：数据规模

k：桶的个数

In-place：占用常数内存，不占用额外内存

Out-place：占用额外内存

### 比较和非比较的区别

* 比较排序：在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。
* 比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。
* 非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。计数排序、基数排序、桶排序则属于非比较排序。
* 非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度O(n)。
* 非比较排序时间复杂度低，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

## 大文件的排序问题

题目：有一个大文件，里面记录了若干数字，把这些数字进行排序。文件大小远大于内存大小。

思路：内存极少的情况下，利用分治策略，利用外存保存中间结果，再用多路归并来排序。

（1）按可用内存的大小，把外存上含有n个记录的文件分成若干个长度为L的子文件，把这些子文件依次读入内存，并利用有效的内部排序方法对它们进行排序，再将排序后得到的有序子文件重新写入外存；

（2）对这些有序子文件逐趟归并，使其逐渐由小到大，直至得到整个有序文件为止

因此，上述问题的解决思路如下：   
1. 对每个小文件排序。   
2. 读取每个文件的首元素，用胜者树找到最小值，记录到大文件上。   
3. 继续采用多路归并排序的方法，继续找最值，直到排完所有的文件

原文链接：<https://blog.csdn.net/okiwilldoit/article/details/80626508>

## 8.5.2 LRU和页面置换算法

地址映射过程中，若在页面中发现所要访问的页面不在内存中，

则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，

则操作系统必须在内存选择一个页面将其移出内存，

以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

https://blog.csdn.net/huyang0304/article/details/82694526

## 八大排序算法

先知道算法的思想，然后再写代码，比较容易理解基记忆。

### 冒泡算法

冒泡排序的思想 稳定 时间复杂度为 o(n2)

依次比较相邻两元素，若前一元素大于后一元素则交换之，直至最后一个元素即为最大；然后重新从首元素开始重复同样的操作，

直至倒数第二个元素即为次大元素；依次类推。如同水中的气泡，依次将最大或最小元素气泡浮出水面

private void sort (int [] a) {

for (int i = 0; i < a.length; i++) {

for (int j = 0 ; j < a.length - i ; j++) {

if (a[j] > a[j + 1 ]) {

// 交换元素

int temp = a[j + 1];

a[j + 1] = a[j];

a[j] = temp;

}

}

}

}

### 选择排序

选择排序的思想 不稳定 时间复杂度为 o(n2)

首先选择第一个元素跟之后的多个元素进行比较得到最小或者最大的元素，然后依次类推，将数组中二，到第N - 1个元素与后面的元素进行比较，然后得到了相应的结果

public void select (int [] a) {

for (int i = 0; i < a.length; i++) {

for (int j = i; j < a.length; j++) {

if (a[j] > a[j + 1]) {

// 交换，把最小的放到前面

}

}

}

}

不稳定性的体现 ：举个例子，序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

### 插入排序

// 插入排序的思想 稳定 时间复杂度为 o(n2)

// 首先从第二个元素开始，第一个元素假定是有序的，然后将得到的元素与第一个元素进行比较，若大于前面的值就将

// 前面的值后移到一位，接着循环，来比较已排序好前面的值和要插入的值的大小，一直找到不符合 a[j] > temp跳出了循环

// 然后，就找到了temp的正确插入位置。

### 希尔排序

// 希尔排序的思想

// 希尔排序是对插入排序的改进，插入排序的时候首先是将要插入的数据一个一个进行比较，然后进行交换，得到要插入的位置

// 而希尔排序首先设置一个gap,通常是长度的一半，然后按照长度除以2的情况进行递减，一直到gap = 1 ,就是普通的插入排序方法

// 这种能够减少插入排序过程中，数据的比较次数。

### 快速排序

// 快速排序的思想

// 选择数组的第一个元素作为基准元素，然后分别从前后进行遍历，

// 将元素进行比较，找到大于基准的元素和小于基准的元素，将其进行交换，到达最后的时候，将

//然后将第一个基准的元素，与a[i] 进行交换，此时，将数据以基准元素分成了两部分，然后再对

// 这两部分，进行了进行排序。进行递归，将所有排好序即可

### 堆排序

// 堆排序的思想

// 首先给你一个初始的数组，对这个数组构造大根堆

，然后得到的大根堆，则数组的第一个元素跟最后

// 一个元素进行比较，然后对n - 1 个元素构造大根堆，依次类推，直到剩余一个元素，则得到的就是

// 升序排列的数组。

### 归并排序

// 归并排序思想

// 首先归并的前提是要先把数组进行二分，一直分到每个单元只有一个元素，但是写程序的时候，不需要

// 一直分，只需要求出middle,就相当于得到了二分的数组了，这种利用二分查找的思想，然后将两个子数组，

// 进行排序，则以此逆着向上就得到了进行排序就得到了要排序的数组了

### 基数排序

// 基数排序的思想

// 首先找到最大的数，然后得到这个最大的数有多少位，再创建十个桶，用来存储从0 - 9 的数字

// 其是用ArrayList实现的，然后，根据遍历最大数位数的次数，计算每个数相应为的数，找到第几个桶，然后将该位的数放入到桶中

// 下面将重新排列得到的数组，注意写新的数组的时候，要将老的ArrayList中的数据删除掉。