

面试总结

2016年3月28日

12:54

思维题总结

2016年9月1日

23:18

1. 16枚硬币，每次只能取1枚，3枚和6枚，两个人（小明和小红），小明先取，最后一个取得人失败，请问是否可以必胜？

硬币数	拿一次后剩余个数	输赢情况
1	0	输
2	1	赢
3	2, 0	输
4	3, 1	赢
5	4, 2	输
6	5, 3, 0	赢
7	6, 4, 1	赢
8	7, 5, 2	赢
9	8, 6, 3	赢
10	9, 7, 4	输
11	10, 8, 5	赢
12	11, 9, 6	输
13	12, 10, 7	赢
14	13, 11, 8	输
15	14, 12, 9	赢
16	15, 13, 10	赢

2. 25 匹马，5 条赛道，一匹马一个赛道，比赛只能得到 5 匹马之间的快慢程度，而不是速度，求决胜 1, 2, 3 名至少多少场。

毫无悬念，一匹马只有跑了才能看出其速度，25 匹马至少都跑了一次，最少五轮，且每轮能排出名次；由于最终只要最快的三名，顾每组只有 1、2、3 有意义继续比下去，4、5 名直接淘汰。每组的 3 有意义的前提是该组的 2 就是总排名的 2、1 就是总排名的 1，每组的 2 有意义的前提是该组的 1 至少第二；归根到底还是看每组第一的情况，故 5 个第一比一次，第一就是总的第一；第四、第五及其所在的组全部被淘汰；故第一的组的二、三名，第二的组第一、二名；第三的组的第一名比最后一次，前两名就是总的二、三名；共七轮。

```
a1,a2,a3,a4,a5;----->a1,a2,a3;  
b1,b2,b3,b4,b5;----->b1,b2,b3;  
c1,c2,c3,c4,c5;----->c1,c2,c3;  
d1,d2,d3,d4,d5;----->d1,d2,d3;  
e1,e2,e3,e4,e5;----->e1,e2,e3;  
a1,b1,c1,d1,e1;-----> a1 ,b1,c1  
a2,a3,b1,b3,c1;-----> a2,a3 ;
```

3. 给一个可以随机生成 n 以内数的随机数生成函数，如何依靠该函数生成一个可以随机生成 M 以内数的随机数生成函数

`(random(n)-1) * n + random(n)`

例：

给定随机数生成函数 `rand1_5`，它可以等概率地生成 1~5 中的每一个整数，如何利用这个函数等概率地生成 1 到 7 呢？

答案：注意到我们平时生成 1~N 的随机数采用的一般是 `rand() % N + 1` 这种方法，如果我们可以随机地等概率生成一组序列始得 $N > 7$ ，那么也可以采用类似的方法。可以考虑利用 `rand1_5` 生成两个数，将这两个数作变换使结果分别取到 1~25，这时就可以选择前面 1 到 21，然后对 7 取模了。

1	<code>func randGen1To5() int {</code>
2	<code> return (rand.Int() % 5) + 1</code>
3	<code>}</code>
4	
5	<code>func randGen() int {</code>

```
6     res := 25
7     f := func(n int) int {
8         return rand.Int() % n + 1
9     }
10    for {
11        if res <= 21 { break }
12        res = 5*(f(5) - 1) + f(5)
13    }
14    return (res-1) % 7 + 1
15 }
```

随机数概率题目

星期日, 九月 4, 2016

3:44 下午

1、给你一个数组，设计一个既高效又公平的方法随机打乱这个数组（此题和洗牌算法的思想一致）

方法比较简单，基本思想是每次随机取一个数，然后把它交换到最后的位置。然后对前 (n-1) 个数使用递归的算法。

2、有一苹果，两个人抛硬币来决定谁吃这个苹果，先抛到正面者吃。问先抛这吃到苹果的概率是多少？

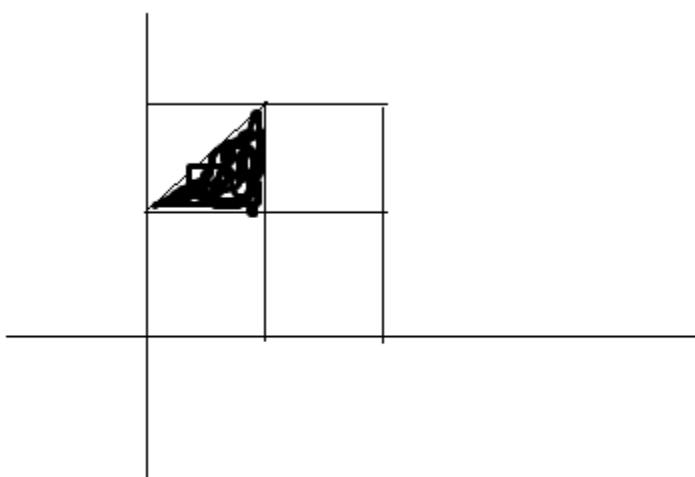
这种题目一看似乎答案就是 $1/2$ ，但其实认真细想并没有那么简单。给所有的抛硬币操作从 1 开始编号，显然先手者只可能在奇数 (1,3,5,7...) 次抛硬币得到苹果，而后手只可能在偶数次 (2,4,6,...) 抛硬币得到苹果。设先手者得到苹果的概率为 p ，第 1 次抛硬币得到苹果的概率为 $1/2$ ，在第 3 次 (3,5,7...) 以后得到苹果的概率为 $p/4$ （这是因为这种只有在第 1 次和第 2 次抛硬币都没有抛到正面（概率为 $1/4=1/2*1/2$ ）的时候才有可能发生，而且此时先手者在此面临和开始相同的局面）。所以可以列出等式 $p=1/2+p/4$ ， $p=2/3$ 。如题，大家懂得……

还可以这样算， $1/2 + 1/8 + 1/16 + \dots + 1/2^{奇数} = 2/3$.

3、一条长度为 1 的线段，随机在其上选 2 个点，将线段分为 3 段，问这 3 个子段能组成一个三角形的概率是多少？

设随机选取的两个数为 x, y ，并令 $y > x$ ，则把长度为 1 的线段截得的三段长度为 $x, y-x, 1-y$ ，根据三角形两边和大于第三边以及两边之差小于第三边的定理，可以列出方程组 $y > 1-y; x < 1-x; x+(1-y) > y-x$ ；即 $x < 1/2; y > 1/2; y > x+1/2$ ；

画图可以算得概率为 $1/8$.



4、一个面试题：快速生成 10 亿个不重复的 18 位随机数的算法(从 n 个数中生成 m 个不重复的随机数)

答案尚不明确。

5、你有两个罐子以及 50 个红色弹球和 50 个蓝色弹球，随机选出一个罐子然后从里面随机选出一个弹球，怎么给出红色弹球最大的选中机会？在你的计划里，得到红球的几率是多少？

题目意思是两个罐子里面放了 50 红色和 50 蓝色弹球，然后我任选一个罐子，从中选中一个红球的最大概率，是设计一个两个罐子里怎么放这 100 球的计划。一个罐子：1 个红球另一个罐子：49 个红球，50 个篮球几率
 $=1/2 + (49/99)*(1/2)=74.7\%$ 。

6、一副扑克牌 54 张，现分成 3 等份每份 18 张，问大小王出现在同一份中的概率是多少？（大意如此）

解答 1：

54 张牌分成 3 等份，共有 $M = (C_{54} \text{ 取 } 18) * (C_{36} \text{ 取 } 18) * (C_{18} \text{ 取 } 18)$ 种分法。其中大小王在同一份的分法有 $N = (C_3 \text{ 取 } 1) * (C_{52} \text{ 取 } 16) * (C_{36} \text{ 取 } 18) * (C_{18} \text{ 取 } 18)$ 种。因此所求概率为 $P = N / M = 17/53$ 。

解答 2：

不妨记三份为 A、B、C 份。大小王之一肯定在某一份中，不妨假定在 A 份中，概率为 $1/3$ 。然后 A 份只有 17 张牌中可能含有另一张王，而 B 份、C 份则各有 18 张牌可能含有另一张王，因此 A 份中含有另一张王的概率是 $17/(17+18+18)=17/53$ 。也因此可知，A 份中同时含有大小王的概率为 $1/3 * 17/53$ 。题目问的是出现在同一份中的概率，因此所求概率为 $3 * (1/3 * 17/53) = 17/53$ 。

7、A 和 B 2 人投硬币，正面 A 得 1 元，反面 B 得一元。起始时 A 有 1 元，B 有 100 元。游戏持续进行，直到其中 1 人破产才终止。

问：(出自投行面试题)

1. 如果硬币正反概率相同，游戏的期待长度(expected duration) 是几次投掷？

2. 如果硬币是不公正的，正面概率为 P，反面概率为 Q. ($P+Q=1$)，那么游戏的期待长度(expected duration) 是几次投掷？

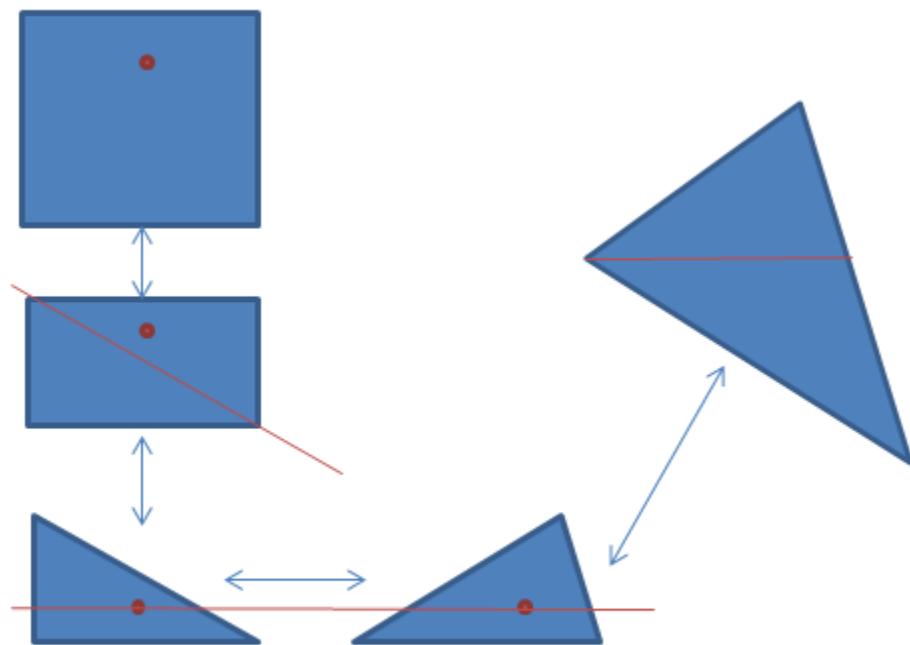
目前认为只有奇数次才可能破产。

第一问： $1 * 1/0.5 + 3 * 1/0.5^3 + 5 * 1/0.5^5$

8、2D 平面上有一个三角形 ABC，如何从这个三角形内部随机取一个点，且使得在三角形内部任何点被选取的概率相同。

在二维坐标系中可以用坐标 (x, y) 来表示图形中的一个点。如下图只要能够在各个带双向箭头的图之间的点能够建立一一映射即可。如把一个长方形

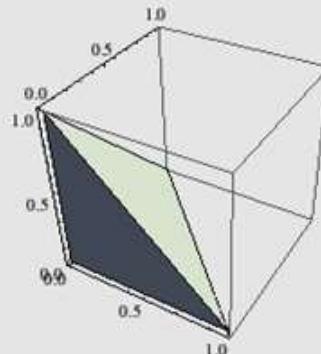
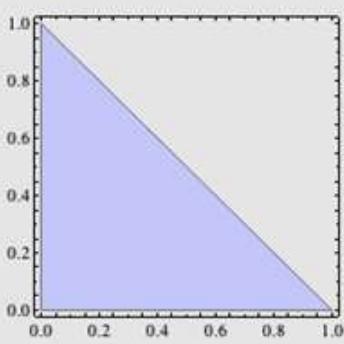
(如正方形)的点映射到另一个长方形的点只要把坐标做相应的放大缩小即可。如把长方形的点映射到一个直角三角形，只要将长方形右上部份的三角形的点映射到对称的左下角的三角形的点即可。而直角三角形映射到一边平行于x轴的三角形的映射只要做x轴相应的偏移即可。而任意三角形可以分割成两个其中有一边平行于x轴的三角形。说的不是很清楚，具体的映射方法可以认真思考并写出公式。



9、平均要取多少个(0,1)中的随机数才能让和超过1。答案：e次，其中e是自然对数的底

为了证明这一点，让我们先来看一个更简单的问题：任取两个 0 到 1 之间的实数，它们的和小于 1 的概率有多大？容易想到，满足 $x+y<1$ 的点 (x, y) 占据了正方形 $(0, 1) \times (0, 1)$ 的一半面积，因此这两个实数之和小于 1 的概率就是 $1/2$ 。类似地，三个数之和小于 1 的概率则是 $1/6$ ，它是平面 $x+y+z=1$ 在单位立方体中截得的一个三棱锥。这个 $1/6$ 可以利用截面与底面的相似比关系，通过简单的积分求得：

$$\int(0..1) (x^2)^{1/2} dx = 1/6$$



可以想到，四个 0 到 1 之间的随机数之和小于 1 的概率就等于四维立方体一角的“体积”，它的“底面”是一个体积为 $1/6$ 的三维体，在第四维上对其进行积分便可得到其“体积”

$$\int(0..1) (x^3)^{1/6} dx = 1/24$$

依此类推， n 个随机数之和不超过 1 的概率就是 $1/n!$ ，反过来 n 个数之和大于 1 的概率就是 $1 - 1/n!$ ，因此加到第 n 个数才刚好超过 1 的概率就是

$$(1 - 1/n!) - (1 - 1/(n-1)!) = (n-1)/n!$$

因此，要想让和超过 1，需要累加的期望次数为

$$\sum_{n=2..\infty} n * (n-1)/n! = \sum_{n=1..\infty} n/n! = e$$

研究方向相关

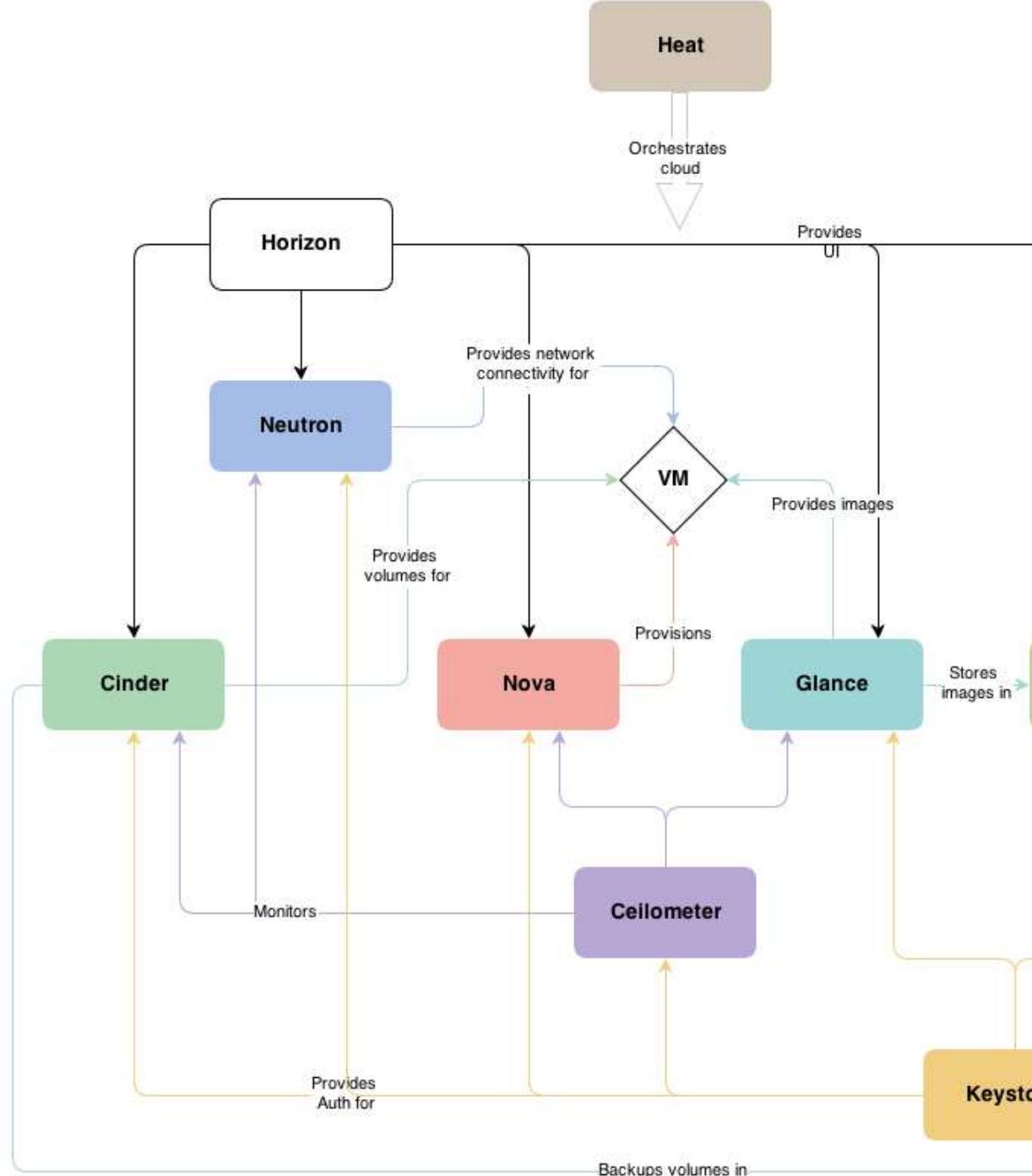
2016 年 8 月 25 日

20:31

- 容器 (docker)
 - windows 下的 docker 和 Linux 下的 docker 的区别
- Openstack
 - 组件介绍：
 - Nova - 计算服务
 - Swift - 对象存储服务
 - Glance - 镜像服务
 - Keystone - 认证服务
 - Horizon - UI 服务
 - Neutron-网络服务

Cinder-块存储服务
Ceilometer-监控服务
Heat-集群服务
Trove-数据库服务

架构图：



- Keepalived+lvs
- memcache 和一致性 Hash

- Memcache
- 一致性 Hash
 - (<http://blog.csdn.net/cywosp/article/details/23397179>)
 - 动态变化的 cache 环境中，判断 Hash 算法好坏的四个标准：
 - **平衡性**：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用；
 - **单调性**：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。
 - **分散性**：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。
 - **负载**：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。
 - 常用的一致性 Hash 算法
 - 环形 Hash 空间：

按照常用的 hash 算法来将对应的 key 哈希到一个具有 2^{32} 次方个桶的空间中，即 0- ($2^{32}-1$) 的数字空间中，将这些数字头尾相连，想象成一个闭合的环。

分布式系统

2016 年 9 月 23 日

18:43

- 分布式系统

分布式系统的理念是让多台服务器协同工作，完成单台服务器无法处理的任务，尤其是高并发或者大数据量的任务。**分布式系统由独立的服务器通过网络松散耦合组成的。**每个服务器都是一台独立的 PC 机，服务器之间通过内部网络连接，内部网络速度一般比较快。因为分布式集群里的服务器是通过内部网络松散耦合，各节点之间的通讯有一定的网络开销，因此分布式系统在设计上尽可能减少节点间通讯。此外，因为网络传输瓶颈，单个节点的性能高低对分布式系统整体性能影响不大。比如，对分布式应用来说，采用不同编程语言开发带来的单个应用服务的性能差异，跟网络开销比起来都可以忽略不计。因此，分布式系统每个节点一般不采用高性能的服务器，而是性能相对一般的普通 PC 服务器。提升分布式系统的整体性能是要通过**横向扩展**（增加更多的服务器），而不是纵向扩展（提升每个节点的服务器性能）。

设计理念：

- 分布式系统对硬件要求很低
- 分布式系统强调横向可扩展性
- 分布式不允许单点失效
- 分布式系统尽可能减少节点之间的通信开销
- 分布式系统应用最好是无状态的
- 分布式事务
 - 分布式事务是指会涉及到操作多个数据库的事务。其实就是将对同一库事务的概念扩大到了对多个库的事务。
 - 目的是为了保证分布式系统中的数据一致性。

- 分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）

- 分布式协议——主要是一致性协议

分布式系统为了保证数据的高可用性，就把数据保留了多个副本，这些副本是放置在不同的物理主机上的，一致性协议主要的就是解决这些副本之间的一致性问题。

典型分布式协议：

- **二阶段提交协议**，主要是用来保证分布式事务的原子性（所有节点要么全做要么全不做）
- **三阶段提交协议**，二阶段提交协议的改进方案
- **Paxos 协议**

- 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。在分布式系统中，常常需要协调他们的动作。如果不同的系统或是同一个系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此干扰来保证一致性，在这种情况下，便需要使用到分布式锁。

可以使用 ZooKeeper 实现分布式锁：

- lock 操作过程：

- 首先为一个 lock 场景，在 zookeeper 中指定对应的一个根节点，用于记录资源竞争的内容
- 每个 lock 创建后，会 lazy 在 zookeeper 中创建一个 node 节点，表明对应的资源竞争标识。（小技巧：node 节点为 EPHEMERAL_SEQUENTIAL，自增长的临时节点）
- 进行 lock 操作时，获取对应 lock 根节点下的所有字节点，也即处于竞争中的资源标识
- 按照 Fair 竞争的原则，按照对应的自增内容做排序，取出编号最小的一个节点做为 lock 的 owner，判断自己的节点 id 是否就为 owner id，如果是则返回，lock 成功。
- 如果自己非 owner id，按照排序的结果找到序号比自己前一位的 id，关注它锁释放的操作（也就是 exist watcher），形成一个链式的触发过程。

- unlock 操作过程：

将自己 id 对应的节点删除即可，对应的下一个排队的节点就可以收到 Watcher 事件，从而被唤醒得到锁后退出

- 其中的几个关键点：

- node 节点选择为 EPHEMERAL_SEQUENTIAL 很重要。
- * 自增长的特性，可以方便构建一个基于 Fair 特性的锁，前一个节点唤醒后一个节点，形成一个链式的触发过程。可以有效的避免 "惊群效应" (一个锁释放，所有等待的线程都被唤醒)，有针对性的唤醒，提升性能。
- * 选择一个 EPHEMERAL 临时节点的特性。因为和 zookeeper 交互是一个网络操作，不可控因素过多，比如网络断了，上一个节点释放锁的操作会失败。临时节点是和对应的 session 挂接的，session 一旦超时或者异常退出其节点就会消失，类似于 ReentrantLock 中等待队列 Thread 的被中断处理。
- 获取 lock 操作是一个阻塞的操作，而对应的 Watcher 是一个异步事件，所以需要使用信号进行通知，正好使用上一篇文章中提到的 BooleanMutex，可以比较方便的解决锁重入的问题。(锁重入可以理解为多次读操作，锁释放为写抢占操作)

- 分布式文件系统

文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

- HDFS

HDFS 中的基本概念

数据块

1. 磁盘是存储文件的地方，文件系统是基于磁盘进行构建的。磁盘有数据块的概念，数据块是磁盘进行数据读/写的基本单位。

磁盘数据块的大小一般为 512 字节，文件系统中也有数据块的概念，文件系统的数据块大小一般为磁盘数据块大小的整数倍，一般为几千字节。

2. hdfs 中也有块的概念，与普通的文件系统一样，hdfs 也会将一个文件分成块大小的多个块 (chunk)，作为独立的存储单

元。 hdfs 中的块很大，默认为 64MB。另外，hdfs 中小于块大小的文件不会占据整个块的空间。将块设置的如此大的原因是為了最小化寻址时间，使数据请求中传输磁盘数据的时间远大于寻址时间（找到文件块的开始位置）。

3. 使用数据块概念的好处：一个文件大小可以超过网络中任意一个磁盘的大小，可以简化子系统的设计（将数据块与元数据分开），分块可以便于数据备份提高容错能力。

namenode 与 datanode

1. hdfs 以 master-slaver 模式工作，其中有一个 namenode 作为主节点，slaver 作为从节点。**namenode 负责管理整个文件系统的命名空间**，它是通过在本地磁盘保存两个文件来实现的，分别是命名空间镜像文件和编辑日志文件。namenode 保存文件块的相关信息，但不永久保存。因为这些信息会在 datanode 启动时重建。

2. datanode 节点是文件系统的工作节点，存储并且检索数据块，并且定期向 namenode 发送所存储的块的列表。

3. namenode 容错机制，一种是备份组成文件系统元数据持久状态的文件，另一种是运行一个辅助 namenode。

4. 联邦 hdfs，因为当文件系统中的文件数目巨大时，namenode 需要将元数据信息存放到内存中，此时内存成为文件系统的瓶颈，可以利用联邦 hdfs 机制，使用多个 namenode 来管理文件系统的元数据，每个 namenode 管理文件系统中的一部分元数据。

- Zookeeper

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，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。Zookeeper 是 hadoop 的一个子项目，其发展历程无需赘述。在分布式应用中，由于工程师不能很好地使用锁机制，以及基于消息的协调机制不适合在某些应用中使用，因此需要有一种可靠的、可扩展的、分布式的、可配置的协调机制来统一系统的状态。Zookeeper 的目的就在于此。

分布式处理之 MapReduce

星期二, 九月 27, 2016

1:23 下午

方法介绍

MapReduce 是一种计算模型，简单的说就是将大批量的工作（数据）分解（MAP）执行，然后再将结果合并成最终结果（REDUCE）。这样做的好处是可以在任务被分解后，可以通过大量机器进行并行计算，减少整个操作的时间。但如果你要我再通俗点介绍，那么，说白了，Mapreduce 的原理就是一个归并排序。

适用范围：数据量大，但是数据种类小可以放入内存

基本原理及要点：将数据交给不同的机器去处理，数据划分，结果归约。

基础架构

想读懂此文，读者必须先要明确以下几点，以作为阅读后续内容的基础知识储备：

1. MapReduce 是一种模式。
2. Hadoop 是一种框架。
3. Hadoop 是一个实现了 MapReduce 模式的开源的分布式并行编程框架。

所以，你现在，知道了什么是 MapReduce，什么是 hadoop，以及这两者之间最简单的联系，而本文的主旨即是，一句话概括：**在 hadoop 的框架上采取 MapReduce 的模式处理海量数据**。下面，咱们可以依次深入学习和了解 MapReduce 和 hadoop 这两个东西了。

MapReduce 模式

前面说了，MapReduce 是一种模式，一种什么模式呢？一种云计算的核心计算模式，一种分布式运算技术，也是简化的分布式编程模式，它主要用于解决问题的程序开发模型，也是开发人员拆解问题的方法。

Ok，光说不上图，没用。如下图所示，MapReduce 模式的主要思想是将自动分割要执行的问题（例如程序）拆解成 Map（映射）和 Reduce（化简）的方式，流程图如下图 1 所示：

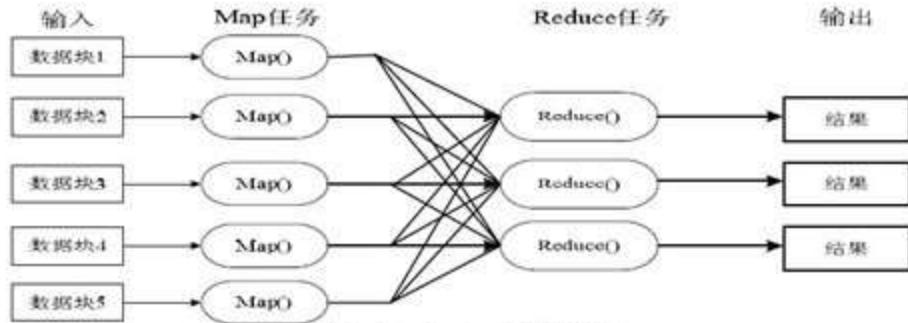


图 1 MapReduce 的处理流程

在数据被分割后通过 Map 函数的程序将数据映射成不同的区块，分配给计算机机群处理达到分布式运算的效果，在通过 Reduce 函数的程序将结果汇整，从而输出开发者需要的结果。

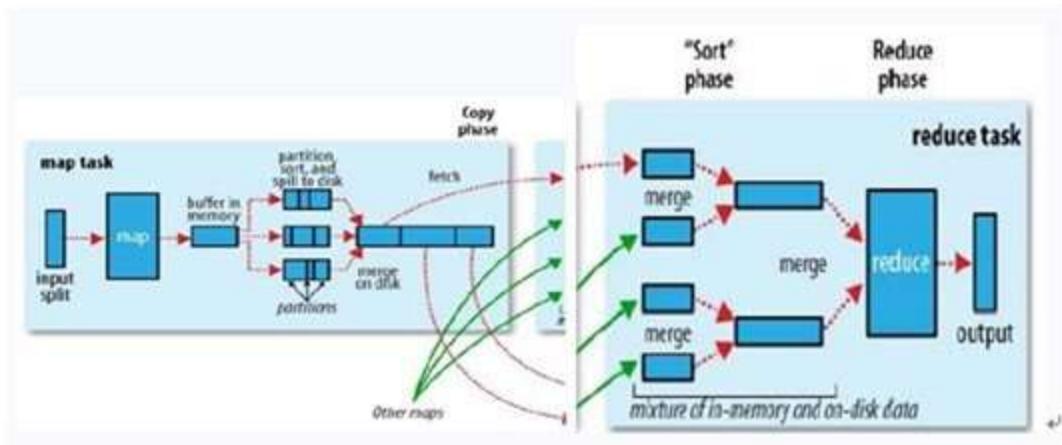
MapReduce 借鉴了函数式程序设计语言的设计思想，其软件实现是指定一个 Map 函数，把键值对(key/value)映射成新的键值对(key/value)，形成一系列中间结果形式的 key/value 对，然后把它们传给 Reduce(规约)函数，把具有相同中间形式 key 的 value 合并在一起。Map 和 Reduce 函数具有一定的关联性。函数描述如表 1 所示：

表 1 Map 函数和 Reduce 函数的描述

函数	输入	输出	说明
Map	<k1,v1>	List(<k2,v2>)	1. 将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理。 2. 每一个输入的<k1,v1>会输出一批<k2,v2>。<k2,v2>是计算的中间结果。
Reduce	<k2,List(v2)>	<k3,v3>	输入的中间结果<k2,List(v2)>中的List(v2)表示是一批属于同一个k2的value

MapReduce 致力于解决大规模数据处理的问题，因此在设计之初就考虑了数据的局部性原理，利用局部性原理将整个问题分而治之。MapReduce 集群由普通 PC 机构成，为无共享式架构。在处理之前，将数据集分布至各个节点。处理时，每个节点就近读取本地存储的数据处理（map），将处理后的数据进行合并（combine）、排序（shuffle and sort）后再分发（至 reduce 节点），避免了大量数据的传输，提高了处理效率。无共享式架构的另一个好处是配合复制（replication）策略，集群可以具有良好的容错性，一部分节点的 down 机对集群的正常工作不会造成影响。

ok，你可以再简单看看下副图，整幅图是有关 hadoop 的作业调优参数及原理，图的左边是 MapTask 运行示意图，右边是 ReduceTask 运行示意图：



如上图所示，其中 map 阶段，当 map task 开始运算，并产生中间数据后并非直接而简单的写入磁盘，它首先利用内存 buffer 来对已经产生的 buffer 进行缓存，并在内存 buffer 中进行一些预排序来优化整个 map 的性能。而上图右边的 reduce 阶段则经历了三个阶段，分别 Copy->Sort->reduce。我们能明显的看出，其中的 Sort 是采用的归并排序，即 merge sort。

问题实例

1. The canonical example application of MapReduce is a process to count the appearances of each different word in a set of documents:
2. 海量数据分布在 100 台电脑中，想个办法高效统计出这批数据的 TOP10。
3. 一共有 N 个机器，每个机器上有 N 个数。每个机器最多存 O(N)个数并对它们操作。如何找到 N^2 个数的中数(median)？

分布式事务、两阶段提交协议、三阶段提交协议

星期五, 九月 23, 2016

10:06 下午

随着大型网站的各种高并发访问、海量数据处理等场景越来越多，如何实现网站的高可用、易伸缩、可扩展、安全等目标就显得越来越重要。为了解决这样一系列问题，大型网站的架构也在不断发展。提高大型网站的高可用架构，不得不提的就是分布式。在[分布式一致性](#)一文中主要介绍了分布式系统中存在的
一致性问题。本文将简单介绍如何有效的解决分布式的一致性问题,其中包括什么
是**分布式事务**，**二阶段提交**和**三阶段提交**。

分布式一致性回顾

在分布式系统中，为了保证数据的高可用，通常，我们会将数据保留多个副本（replica），这些副本会放置在不同的物理的机器上。为了对用户提供正确的增\删\改\差等语义，我们需要保证这些放置在不同物理机器上的副本是一致的。

为了解决这种分布式一致性问题，前人在性能和数据一致性的反反复复权衡过程中总结了许多典型的协议和算法。其中比较著名的有**二阶提交协议**（Two Phase Commitment Protocol）、**三阶提交协议**（Three Phase Commitment Protocol）和**Paxos 算法**。

分布式事务

分布式事务是指会涉及到操作多个数据库的事务。其实就是将对同一库事务的概念扩大到了对多个库的事务。目的是为了保证分布式系统中的数据一致性。分布式事务处理的关键是必须有一种方法可以知道事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）

在分布式系统中，各个节点之间在物理上相互独立，通过网络进行沟通和协调。由于存在事务机制，可以保证每个独立节点上的数据操作可以满足 ACID。但是，相互独立的节点之间无法准确的知道其他节点中的事务执行情况。所以从理论上讲，两台机器理论上无法达到一致的状态。如果想让分布式部署的多台机器中的数据保持一致性，那么就要保证在所有节点的数据写操作，要不全部都执行，要么全部的都不执行。但是，一台机器在执行本地事务的时候无法知道其他机器中的本地事务的执行结果。所以他也就无法知道本次事务到底应该 commit 还是 rollback。所以，常规的解决办法就是引入一个“协调者”的组件来统一调度所有分布式节点的执行。

XA 规范

X/Open 组织（即现在的 Open Group）定义了分布式事务处理模型。X/Open DTP 模型（1994）包括应用程序（AP）、事务管理器（TM）、资源管理器（RM）、通信资源管理器（CRM）四部分。一般，常见的事务管理器（TM）是交易中间件，常见的资源管理器（RM）是数据库，常见的通信资源管理器（CRM）是消息中间件。通常把一个数据库内部的事务处理，如对多个表的操作，作为本地事务看待。数据库的事务处理对象是本地事务，而分布式事务处理的对象是全局事务。所谓全局事务，是指分布式事务处理环境中，多个数据库可能需要共同完成一个工作，这个工作即

是一个全局事务，例如，一个事务中可能更新几个不同的数据库。对数据库的操作发生在系统的各处但必须全部被提交或回滚。此时一个数据库对自己内部所做操作的提交不仅依赖本身操作是否成功，还要依赖与全局事务相关的其它数据库的操作是否成功，如果任一数据库的任一操作失败，则参与此事务的所有数据库所做的所有操作都必须回滚。一般情况下，某一数据库无法知道其它数据库在做什么，因此，在一个 DTP 环境中，交易中间件是必需的，由它通知和协调相关数据库的提交或回滚。而一个数据库只将其自己所做的操作（可恢复）影射到全局事务中。

XA 就是 X/Open DTP 定义的交易中间件与数据库之间的接口规范（即接口函数），交易中间件用它来通知数据库事务的开始、结束以及提交、回滚等。XA 接口函数由数据库厂商提供。

二阶提交协议和**三阶提交协议**就是根据这一思想衍生出来的。可以说二阶段提交其实就是实现 **XA 分布式事务**的关键(确切地说：两阶段提交主要保证了分布式事务的原子性：即所有结点要么全做要么全不做)

2PC

二阶段提交(Two-phase Commit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。通常，二阶段提交也被称为是一种协议(Protocol)。在分布式系统中，每个节点虽然可以知晓自己的操作时成功或者失败，却无法知道其他节点的操作的成功或失败。当一个事务跨越多个节点时，为了保持事务的 ACID 特性，需要引入一个作为协调者的组件来统一掌控所有节点(称作参与者)的操作结果并最终指示这些节点是否要把操作结果进行真正的提交(比如将更新后的数据写入磁盘等等)。因此，**二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。**

所谓的两个阶段是指：第一阶段：**准备阶段(投票阶段)**和第二阶段：**提交阶段(执行阶段)**。

准备阶段

事务协调者(事务管理器)给每个参与者(资源管理器)发送 Prepare 消息，每个参与者要么直接返回失败(如权限验证失败)，要在本地执行事务，写本地的 redo 和 undo 日志，但不提交，到达一种“万事俱备，只欠东风”的状态。

可以进一步将准备阶段分为以下三个步骤：

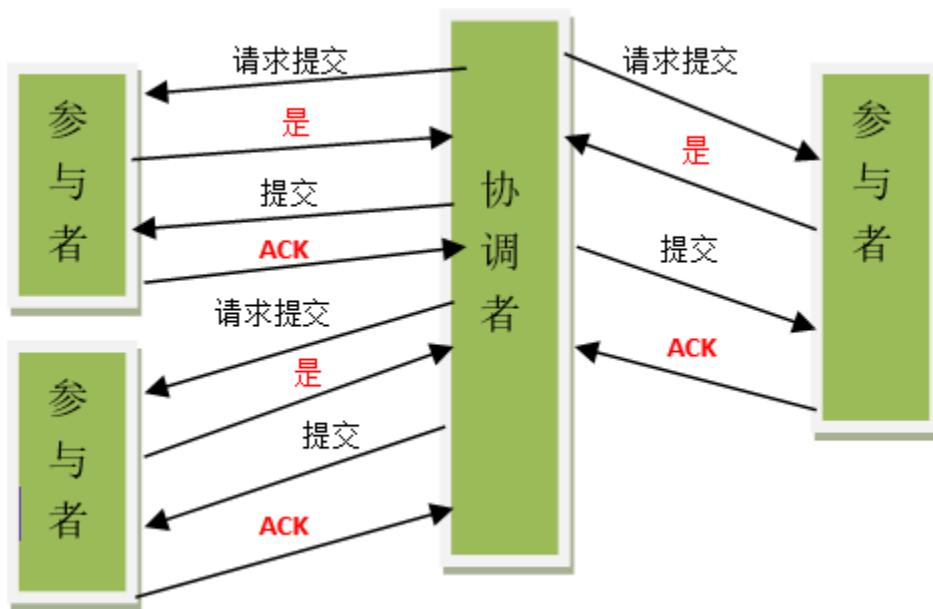
- 1) 协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。
- 2) 参与者节点执行询问发起为止的所有事务操作，并将 Undo 信息和 Redo 信息写入日志。（注意：若成功这里其实每个参与者已经执行了事务操作）
- 3) 各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“同意”消息；如果参与者节点的事务操作实际执行失败，则它返回一个“中止”消息。

提交阶段

如果协调者收到了参与者的失败消息或者超时，直接给每个参与者发送回滚(Rollback)消息；否则，发送提交(Commit)消息；参与者根据协调者的指令执行提交或者回滚操作，释放所有事务处理过程中使用的锁资源。(注意：必须在最后阶段释放锁资源)

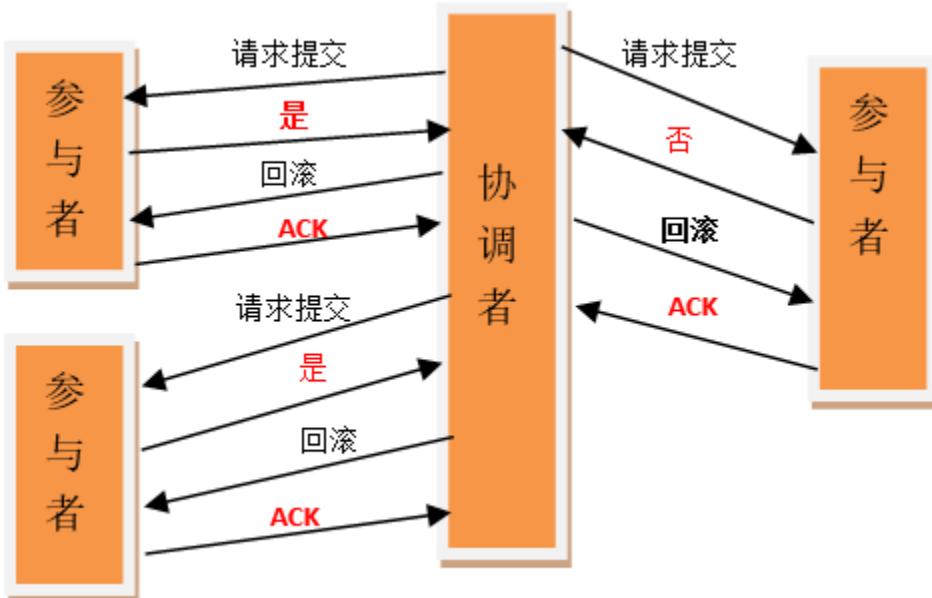
接下来分两种情况分别讨论提交阶段的过程。

当协调者节点从所有参与者节点获得的相应消息都为“同意”时：



- 1) 协调者节点向所有参与者节点发出“正式提交(commit)”的请求。
- 2) 参与者节点正式完成操作，并释放在整个事务期间内占用的资源。
- 3) 参与者节点向协调者节点发送“完成”消息。
- 4) 协调者节点受到所有参与者反馈的“完成”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为“中止”，或者协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时：



- 1) 协调者节点向所有参与者节点发出“回滚操作(rollback)”的请求。
- 2) 参与者节点利用之前写入的 Undo 信息执行回滚，并释放放在整个事务期间内占用的资源。
- 3) 参与者节点向协调者节点发送“回滚完成”消息。
- 4) 协调者节点受到所有参与者反馈的“回滚完成”消息后，取消事务。

不管最后结果如何，第二阶段都会结束当前事务。

二阶段提交看起来确实能够提供原子性的操作，但是不幸的事，二阶段提交还是有几个缺点的：

- 1、**同步阻塞问题**。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。
- 2、**单点故障**。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）
- 3、**数据不一致**。在二阶段提交的阶段二中，当协调者向参与者发送 **commit** 请求之后，发生了局部网络异常或者在发送 **commit** 请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了 **commit** 请求。而在这些部分参与者接到 **commit** 请求之后就会执行 **commit** 操作。但是其他部分未接到 **commit**

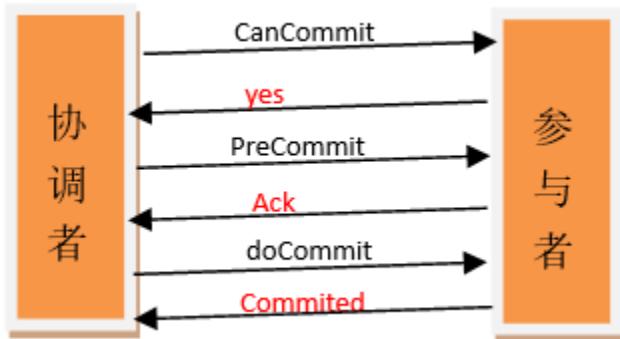
请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

4、二阶段无法解决的问题：协调者再发出 **commit** 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

由于二阶段提交存在着诸如同步阻塞、单点问题、脑裂等缺陷，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。

3PC

三阶段提交（Three-phase commit），也叫三阶段提交协议（Three-phase commit protocol），是二阶段提交（2PC）的改进版本。



与两阶段提交不同的是，三阶段提交有两个改动点。

- 1、引入超时机制。同时在协调者和参与者中都引入超时机制。
- 2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 **CanCommit**、**PreCommit**、**DoCommit** 三个阶段。

CanCommit 阶段

3PC 的 **CanCommit** 阶段其实和 2PC 的准备阶段很像。协调者向参与者发送 **commit** 请求，参与者如果可以提交就返回 **Yes** 响应，否则返回 **No** 响应。

1. 事务询问 协调者向参与者发送 **CanCommit** 请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

2. 响应反馈 参与者接到 **CanCommit** 请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回 **Yes** 响应，并进入预备状态。否则反馈 **No**

PreCommit 阶段

协调者根据参与者的反应情况来决定是否可以记性事务的 PreCommit 操作。
根据响应情况，有以下两种可能。

假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行。

- 1.发送预提交请求** 协调者向参与者发送 PreCommit 请求，并进入 Prepared 阶段。
- 2.事务预提交** 参与者接收到 PreCommit 请求后，会执行事务操作，并将 undo 和 redo 信息记录到事务日志中。
- 3.响应反馈** 如果参与者成功的执行了事务操作，则返回 ACK 响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

- 1.发送中断请求** 协调者向所有参与者发送 abort 请求。
- 2.中断事务** 参与者收到来自协调者的 abort 请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

doCommit 阶段

该阶段进行真正的事务提交，也可以分为以下两种情况。

执行提交

- 1.发送提交请求** 协调接收到参与者发送的 ACK 响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送 doCommit 请求。
- 2.事务提交** 参与者接收到 doCommit 请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。
- 3.响应反馈** 事务提交完之后，向协调者发送 Ack 响应。
- 4.完成事务** 协调者接收到所有参与者的 ack 响应之后，完成事务。

中断事务 协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

- 1.发送中断请求** 协调者向所有参与者发送 abort 请求

2.事务回滚 参与者接收到 `abort` 请求之后，利用其在阶段二记录的 `undo` 信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

3.反馈结果 参与者完成事务回滚之后，向协调者发送 `ACK` 消息

4.中断事务 协调者接收到参与者反馈的 `ACK` 消息之后，执行事务的中断。

在 `doCommit` 阶段，如果参与者无法及时接收到来自协调者的 `doCommit` 或者 `reboot` 请求时，会在等待超时之后，会继续进行事务的提交。（其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了 `PreCommit` 请求，那么协调者产生 `PreCommit` 请求的前提条件是他在第二阶段开始之前，收到所有参与者的 `CanCommit` 响应都是 Yes。（一旦参与者收到了 `PreCommit`，意味他知道大家其实都同意修改了）所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到 `commit` 或者 `abort` 响应，但是他有理由相信：成功提交的几率很大。）

2PC 与 3PC 的区别

相对于 2PC，3PC 主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行 `commit`。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的 `abort` 响应没有及时被参与者接收到，那么参与者在等待超时之后执行了 `commit` 操作。这样就和其他接到 `abort` 命令并执行回滚的参与者之间存在数据不一致的情况。

了解了 2PC 和 3PC 之后，我们可以发现，无论是二阶段提交还是三阶段提交都无法彻底解决分布式的一致性问题。Google Chubby 的作者 Mike Burrows 说过，“there is only one consensus protocol, and that's Paxos” – all other approaches are just broken versions of Paxos. 意即世上只有一种一致性算法，那就是 Paxos，所有其他一致性算法都是 Paxos 算法的不完整版。后面的文章会介绍这个公认为难于理解但是行之有效的 Paxos 算法。

分布式一致性算法——paxos

星期五, 九月 23, 2016

10:12 下午

随着大型网站的各种高并发访问、海量数据处理等场景越来越多，如何实现网站的高可用、易伸缩、可扩展、安全等目标就显得越来越重要。为了解决这样一系列问题，大型网站的架构也在不断发展。提高大型网站的高可用架构，不得不提的就是分布式。在关于分布式事务、两阶段提交协议、三阶段提交协议一文中主要用于解决分布式一致性问题的集中协议，那么这篇文章主要讲解业内公认的比较难的也是最行之有效的 paxos 算法。

我认为对 paxos 算法讲解的最清楚的就是[维基百科](#)了。但是要看懂维基百科中的介绍需要很强的数学思维（paxos 毕竟是一个算法），而且有很多关于定理的推论、证明等过程。那么本篇文章主要站在程序的角度，通俗的，循序渐进的讲解到底什么是 paxos 算法。

背景

Google Chubby 的作者 Mike Burrows 说过，“there is only one consensus protocol, and that's Paxos” – all other approaches are just broken versions of Paxos. 意即世上只有一种一致性算法，那就是 Paxos，所有其他一致性算法都是 Paxos 算法的不完整版。

Paxos 算法是莱斯利·兰伯特（Leslie Lamport，就是 LaTeX 中的“La”，此人现在在微软研究院）于 1990 年提出的一种基于消息传递的一致性算法。为描述 Paxos 算法，Lamport 讲述了这样一个故事：

在古希腊有一个岛屿叫做 Paxos，这个岛屿通过议会的形式修订法律。执法者（legislators，后面称为牧师 priest）在议会大厅（chamber）中表决通过法律，并通过服务员传递纸条的方式交流信息，每个执法者会将通过的法律记录在自己的账目（ledger）上。问题在于执法者和服务员都不可靠，他们随时会因为各种事情离开议会大厅、服务员也有可能重复传递消息（或者直接彻底离开），并随时可能有新的执法者（或者是刚暂时离开的）回到议会大厅进行法律表决，因此，议会协议要求保证上述情况下能够正确的修订法律并且不会产生冲突。

什么是 paxos 算法

Paxos 算法是分布式一致性算法用来解决一个分布式系统如何就某个值（决议）达成一致的问题。

人们在理解 paxos 算法时会遇到一些困境，那么接下来，我们带着以下几个问题来学习 paxos 算法：

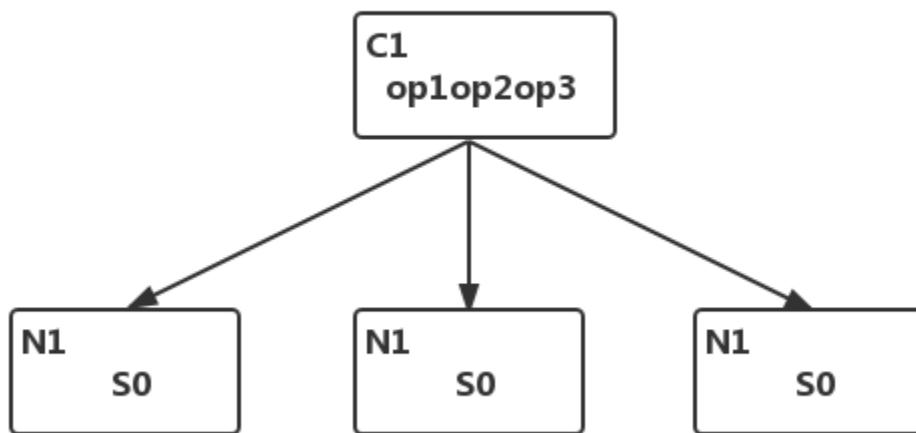
- 1、paxos 到底在解决什么问题？
- 2、paxos 到底如何在分布式存储系统中应用？
- 3、paxos 的核心思想是什么？

paxos 解决了什么问题

在[关于分布式一致性的探究](#)中我们提到过，分布式的一致性问题其实主要是指分布式系统中的数据一致性问题。所以，为了保证分布式系统的一致性，就要保证分布式系统中的数据是一致的。

在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。

所以，paxos 算法主要解决的问题就是如何保证分布式系统中各个节点都能执行一个相同的操作序列。



上图中，C1 是一个客户端，N1、N2、N3 是分布式部署的三个服务器，初始状态下 N1、N2、N3 三个服务器中某个数据的状态都是 S0。当客户端要向服务器请求处理操作序列：op1op2op3 时（op 表示 operation）（这里把客户端的写操作简化成向所有服务器发送相同的请操作序列，实际上可能通过

Master/Slave 模式处理）。如果想保证在处理完客户端的请求之后，N1、N2、N3 三个服务器中的数据状态都能从 S0 变成 S1 并且一致的话（或者没有执行成功，还是 S0 状态），就要保证 N1、N2、N3 在接收并处理操作序列 op1op2op3 时，严格按照规定的顺序正确执行 opi，要么全部执行成功，要不然全部都不执行。

所以，针对上面的场景，paxos 解决的问题就是如何依次确定不可变操作 opi 的取值，也就是确定第 i 个操作什么，在确定了 opi 的内容之后，就可以让各个副本执行 opi 操作。

Paxos 算法详解

Paxos 是一个十分巧妙的一致性算法，但是他十分难以理解，就连他的作者 Lamport 都被迫对他做过多种讲解。我认为对 paxos 算法讲解的最清楚的就是[维基百科](#)了。但是要看懂维基百科中的介绍需要很强的数学思维（paxos 毕竟是一个算法），而且有很多关于定理的推论、证明等过程。那么本篇文章主要站在程序的角度，通俗的，循序渐进的讲解到底什么是 paxos 算法。

我们先把前面的场景简化，把我们现在要解决的问题简化为**如何确定一个不可变变量的取值**（每一个不可变变量可以标识一个操作序列中的某个操作，当确保每个操作都正确之后，就可以按照顺序执行这些操作来保证数据能够准确无误的从一个状态转变成另外一个状态了）。

接下来，请跟我一步一步的学习 paxos 算法。

要学习 paxos 算法，我们就要从他要解决的问题出发，假如没有 paxos 算法，当我们面对**如何确定一个不可变变量的取值**这样一个问题的时候，我们应该如何解决呢？

这里暂不介绍 paxos 中的角色的概念，读者可以自行从[维基百科](#)中了解。不了解的话也可以直接往下看，看着看着就了解了。

问题抽象

我们把确定一个不可变变量的取值问题定义成：

设计一个系统，来存储名称为 var 的变量。

`var` 的取值可以是任意二进制数

系统内部由多个 `Acceptor` 组成，负责管理和存储 `var` 变量。

系统对外提供 API, 用来设置 `var` 变量的值

`propose(var, V) => <ok, f> or <error>`

将 `var` 的值设置为 `V`，系统会返回 `ok` 和系统中已经确定的取值 `f`，或者返回 `error`。

外部有多个 `Proposer` 机器任意请求系统，调用系统 API(`propose(var, V) => <ok, f> or <error>`) 来设置 `var` 变量的值。

如果系统成功的将 `var` 设置成了 `V`，那么返回的 `f` 应该就是 `V` 的值。否则，系统返回的 `f` 就是其他的 `Proposer` 设置的值。

>

系统需要保证 `var` 的取值满足一致性

如果 `var` 没有被设置过，那么他的初始值为 `null`

一旦 `var` 的值被设置成功，则不可被更改，并且可以一直都能获取到这个值

系统需要满足容错特性

可以容忍任意 `proposer` 出现故障

可以容忍少数 `acceptor` 故障（半数以下）

暂时忽略网络分化问题和 `acceptor` 故障导致 `var` 丢失的问题。

到这里，问题已经抽象完成了，读者可以再仔细看看上面的系统描述。如果这样设置一个系统，是不是就可以保证变量 `var` 的不可变性了呢？

这里还是再简单讲解一下，上面的系统确实可以保证变量 `var` 的不可变性。

因为 `var` 的初始值为 `null`，当有 `proposer` 请求接口 `propose (var , v)` 设置 `var` 的值的时候，系统会将 `var` 设置为 `v`，并返回 `f (f==v)`。

`var` 变量被初始化以后，再有 `proposer` 请求 `propose(var,v)` 设置 `var` 的值的时候，系统会直接返回系统中已有的 `var` 的值 `f`，而放弃 `proposer` 提供的 `v`。

系统难点

要设计以上系统存在以下难点：

- 1、管理多个 proposer 并发执行
- 2、容忍 var 变量的不可变性
- 3、容忍任意 Proposer 的故障
- 4、容忍半数以下的 acceptor 的故障

解决方案一

先考虑整个系统由单个 acceptor 组成。通过类似[互斥锁](#)的方式来管理并发的 proposer 的请求。

proposer 向 acceptor 申请 acceptor 的互斥访问权，当取得互斥访问权之后才能调用 api 给 var 变量赋值。

accepter 向 proposer 发放互斥访问权，谁取得了互斥访问权，acceptor 就接收谁的请求。

这样通过互斥访问的机制，proposer 就要按照获取互斥访问权的顺序来请求系统。

一旦 acceptor 接收到一个 proposer 请求，并成功给 var 变量赋值之后，就不再允许其他的 proposer 设置 var 变量的值。每当再有 proposer 来请求设置 var 变量的值的时候，acceptor 就会将 var 里面现有的值返回给他。

基于互斥访问权的 acceptor 的实现

acceptor 会保存变量 var 的值和一个互斥锁 Lock。

提供接口 prepare()

加互斥锁，给予 var 的互斥访问权，并返回当前 var 的取值

提供接口 release()

用于释放互斥访问权

提供接口 accept(var, v)

如果已经加锁，并且当前 var 没有值，则将 var 的值设置成 v，并释放锁。

proposer 采用两阶段来实现

Step1、通过调用 `prepare` 接口来获取互斥性访问权和当前 var 的取值

如果无法获取到互斥性访问权，则返回，并不能进入到下一个阶段，因为其他 proposer 获取到了互斥性访问权。

Step2、根据当前 var 的取值 f 选择执行

1、如果 f 的取值为 null，说明没有被设置过值，则调用接口 `accept(var, v)` 来将 var 的取值设置成 v，并释放掉互斥性访问权。

2、如果 f 的取值不为 null，说明 var 已经被其他 proposer 设置过值，则调用 `release` 接口释放掉互斥性访问权。

总结：方案一通过互斥访问的方式来保证所有的 proposer 能够串行的访问 acceptor，这样其实并没有解决多个 proposer 并发执行的问题。只是想办法绕开了并发执行。虽然可以在一定程度上保证 var 变量的取值是确定的。但是 一旦获取到互斥访问权的 proposer 在执行过程中出现故障，那么就会导致所有其他 proposer 无法再获取到互斥访问权，就会发生死锁。。所以，方案一不仅效率低、而且还会产生死锁问题，不能容忍任意 Proposer 出现故障。

在之前提到的四个系统难点中，方案一可以解决难点 1 和难点 2，但是无法解决难点 3 和难点 4。

解决方案二

通过引入[抢占式访问权](#)来取代互斥访问权。acceptor 有权让任意 proposer 的访问权失效，然后将访问权发放给其他的 proposer。

在方案二中，proposer 向 acceptor 发出的每次请求都要带一个编号（epoch），且编号间要存在全序关系。一旦 acceptor 接收到 proposer 的请求中包含一个更大的 epoch 的时候，马上让旧的 epoch 失效，不再接受他们提交的取值。然后给新的 epoch 发放访问权，让他可以设置 var 变量的值。

为了保证 var 变量取值的不变性，不同 epoch 的 proposer 之前遵守后者认同前者的原则：

在确保旧的 epoch 已经失效后，并且旧的 epoch 没有设置 var 变量的值，新的 epoch 会提交自己的值。

当旧的 epoch 已经设置过 var 变量的取值，那么新的 epoch 应该认同旧的 epoch 设置过的值，并不在提交新的值。

基于抢占式访问权的 acceptor 的实现

体系结构操作系统

2016 年 3 月 28 日

12:58

1. 进程与线程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程是进程的一个实体,是 CPU 调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源

区别：

- 调度：线程是独立调度的基本单位，进程是拥有资源的单位，线程切换不一定引起进程切换，进程切换肯定引起线程切换；
- 资源分配：进程是拥有资源的基本单位（不包括 CPU）；
- 并发性：进程线程都可以并发执行
- 系统开销：系统操作进程会有更多的开销，包括资源分配和释放、进度保存等；操作线程开销则很小，并且线程同步和通信都不需要系统处理；
- 地址空间：进程间地址空间独立，同一进程内的线程共享进程资源；
- 通信：进程通信需要操作系统辅助，线程则可以直接读写数据段。

2. 进程间通信方式

- 管道 (PIPE)

可以用于具有亲缘关系的进程之间的通信，允许一个进程和另一个有共同祖先的进程之间进行通信；

- FIFO (命名管道)

- 克服了管道没有名字的限制，因此，除了具有管道所有的功能外，还允许无亲缘关系的进程间的通信；
- 命名管道在文件系统中有对应的文件名；
- 命名管道通过命令 mkfifo 或者系统调用 mkfifo 来创建。
- 消息队列
 - 消息队列是消息的链接表，包括 Posix 消息队列，System V 消息队列；
 - 有足够权限的进程可以向队列中添加消息；
 - 被赋予读权限的进程可以独奏队列中的消息；
 - 消息队列克服了信号承载信息量。
- 信号量
 - 主要作为进程间以及同一进程不同线程之间的同步手段。
- 共享内存
 - 使得多个进程可以访问同一块内存空间，是最快的 IPC 形式；
 - 是针对其他通信机制运行效率较低而设计的；
 - 往往和其他通信机制配合使用。
- Socket
 - 更为一般的进程间通信机制，用于不同机器之间的进程间的通信。
- 内存映射
 - 内存映射允许多个进程间通信；
 - 每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现。
- 线程通信、同步方式
 - 线程通信方式
 - i. 全局变量
 - ii. 使用消息实现通信
 - iii. 使用事件实现线程通信
 - 线程同步方式
 - i. 临界区，只能是同一个进程内的线程使用
 - ii. 互斥量，可以跨进程使用
 - iii. 信号量，可以跨进程使用
 - iv. 事件，可以跨进程使用

- **进程调度**

CPU 以线程为单位调度，只有在用户级线程系统中采用进程为单位调度

进程状态：就绪态、运行态、阻塞态

进程调度方式：抢占式和非抢占式

调度算法：先进先出、短进程优先、轮转法等

- 5. 虚拟内存和物理内存映射方式

虚拟内存实现：

1. 请求分页式：通过页表映射
2. 请求分段式：通过段表映射
3. 请求段页式：通过段表+页表映射

- 6. 线程安全，怎么保证线程安全？

指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

可以用锁机制来保证线程安全。

- 7. 死锁，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

1. 死锁产生的四个必要条件

互斥，不可抢占，请求和保持，循环等待

2. 死锁预防

- 打破互斥条件，允许进程同时访问某些资源；
- 打破不可抢占条件，允许进程强行从占有者那里夺取某些资源；
- 打破保持和请求条件，实行资源预先分配策略；
- 打破循环等待条件，实行资源有序分配策略。

死锁避免

1. 安全序列

系统中所有进程按照某一种次序分配资源，并依次地运行完毕，这个进程序列就是安全序列。

2. 银行家算法

Dijkstra 提出的算法，用来寻找安全序列

4. 安全状态

安全状态不是死锁状态，死锁状态是不安全状态。

不安全状态可能导致死锁，但是只要状态是安全的，系统就能避免不安全（或者死锁）状态。

8. 原子操作 (atomic operation) 意为"不可被中断的一个或一系列操作"。
加锁，关中断（有个中断寄存器）

9. 硬中断和软中断

硬中断：

1. 硬中断是由硬件产生的，每个硬件设备都有自己的 IRQ (中断请求)。基于 IRQ，CPU 可以把相应的请求发到对应的硬件驱动上；
2. 处理中断的驱动需要运行在 CPU 上，因此，当中断产生的时候，CPU 会中断当前正在运行的任务，来处理中断。在多核心的系统中，一个中断只能中断一个 CPU；
3. 硬件中断可以直接中断 CPU，它会引起内核相关代码被触发，对于那些需要花一段时间去处理的进程，中断代码本身也可以被其他硬中断中断；
4. 对于时钟中断，内核调度代码会将当前正在运行的进程挂起，从而让其他的进程运行。它的存在是为了调度器可以调度多任务。

软中断：

1. 软中断的处理非常像硬中断，不过它们仅仅由当前运行的进程产生；
2. 软中断通常是一些对于 IO 的请求；
3. 软中断仅与内核相联系。而内核主要负责对需要运行的其他任何的进程进行调度。一些内核允许设备驱动的一些部分存在于用户控件，并且当需要的时候内核也会调度这个进程去运行；
4. 软中断不会直接中断 CPU。只有当前正在运行的代码才会产生软中断。这种中断是一种需要内核为正在运行的进程去做一些事情的请求。

10. 硬件如何读取文件

是不是如何读盘。。。。

11. 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个方式。

在分布式系统中，常常需要协调他们的动作。如果不同的系统不着是同一系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此的干扰来保护系统不着是同一系统的不同主机之间共享了一个或一组资源，那么访问这些资源的时候，往往需要互斥来防止彼此的干扰来保证一致性，在这种情况下需要使用分布式锁。

12. 线程同步与阻塞的关系

- 线程同步与阻塞没有关系
- 同步和异步关注的是消息通信机制。
 - 同步，在发出一个“调用”的时候，在没有得到结果之前，该“调用”就不返回。一旦调用返回，就得到了返回值。
 - 异步，“调用”发出之后，这个调用就直接返回了，没有返回结果，一个异步过程，调用者不会立刻得到结果，而是在“调用”发出后，“被调用者”通过状态、通知来通知调用者，或者通过回调函数处理这个调用。
- 阻塞和非阻塞关注的是程序在等待调用结果时的状态。
 - 阻塞调用试着调用结果返回之前，当前线程会被挂起，调用线程在得到结果之后才会返回；
 - 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程。

Linux 如何进行进程调度。

- SCHED_OTHER：普通进程，基于优先级方式调度；
- SCHED_FIFO：实时进程，基于简单先进先出策略调度；
- SCHED_RR：实时进程，基于时间片的 SCHED_FIFO，实时轮流调度算法。

上下文切换

目前流行的 CPU 在同一时间内只能运行一个线程，超线程的处理器（包括多核处理器）可以同一时间运行多个线程，linux 将多核处理器当作多个单独 CPU 来识别的。每个进程都会分到 CPU 的时间片来运行，当某个进程（**线程是轻量级进程，他们是可以并行运行的，并且共享地使用他们所属进程的地址空间资源，比如：内存空间或其他资源**）当进程用完时间片或者被另一个优先级更高的进程抢占的时候，CPU 会将该进程备份到 CPU 的**运行队列**中，其他进程被调度在 CPU 上运行，这个进程切换的过程被称作“上下文切换”，过多的上下文切换会造成系统很大的开销。

进程上下文用进程的 PCB（**进程控制块**，也称为 PCB，即任务控制块）表示，它包括进程状态，CPU 寄存器的值等。

通常通过执行一个状态保存来保存 CPU 当前状态，然后执行一个状态恢复重新开始运行。

在上下文切换过程中，CPU 会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中，上下文切换过程中的“页码”信息是保存在进程控制块（PCB）中的。PCB 还经常被称作“切换帧”

（switchframe）。 “页码”信息会一直保存到 CPU 的内存中，直到它们被再次使用。

z

Linux 系统

2016 年 9 月 6 日

14:07

- MMU 是 Memory Manage Unit 的缩写，即是存储管理单元，其功能是和物理内存之间进行地址转换 在 CPU 和物理内存之间进行地址转换，将地址从**逻辑空间映射到物理地址空间**。
- linux 中调用 write 发送网络数据返回 n(n>0)表示本地已发送 n 个字节。
- Fork

- 使用 fork 函数得到的子进程从父进程的继承了整个进程的地址空间，包括：**进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等**

- **子进程和父进程的区别：**

- 父进程设置的锁，子进程不继承（因为如果是排它锁，被继承的话，矛盾了）
 - 各自的进程 ID 和父进程 ID 不同
 - 子进程的未决告警被清除
 - 子进程的未决信号集设置为空集
- 以下程序打印"-"的个数

```
int main(void){  
    int i;  
    for(i=0;i<4;i++){  
        fork();  
        printf("-\n");  
    }  
    return 0;  
}
```

i=0 时，主进程和其创建的子进程分别打印'-'，打印 2 个

i=1 时，之前两个进程打印'-'，每个进程又创建新的子进程，共打印 4 个'-'

i=2 时，之前的四个进程分别打印'-'，并创建新的子进程，故共打印 8 个'-'

i=3 时，之前的 8 个进程分别打印'-'，并创建新的子进程，故共打印 16 个'-'

综上所述，共打印 $2+4+8+16=30$ 个

- Linux 系统安装方式：

- 光盘安装 (常规情况) 硬盘安装 (无光驱情况)
- 网络安装-NFS 方式 (适合于批量安装大量服务器，和 kickstart 自动安装一起使用)
- 网络安装-FTP 方式 (适合于批量安装大量服务器，和 kickstart 自动安装一起使用)

- 网络安装-HTTP 方式 (适合于批量安装大量服务器，和 kickstart 自动安装一起使用)
- Linux 中，一个端口能够接受 tcp 链接数量的理论上限是**无限个**
- Linux 查看目录大小
 - du -sh 可以显示当前目录及子目录的磁盘占用情况
 - df -h 显示整个文件系统的使用情况，不能得到当前文件夹的情况
- 一些配置文件
 - WEB 服务器配置文件 http.conf
 - 启动脚本配置文件 initd.conf
 - samba 脚本 rc.samba
 - samba 服务配置文件 smb.conf
- Linux 网络编程
 - Socket
 - 两种模式：阻塞和非阻塞
- 网络配置
 - /etc/hosts 主机名到 IP 地址的映射关系的文件
 - /etc/resolv.conf DNS 服务的配置文件
 - /etc/gateways 建立动态路由需要用到的文件
 - /etc/services 文件定义了网络服务和对应的端口号、协议等信息
- Linux 同步相关
 - spinLock，在多处理器多线程环境的场景中有很广泛的使用，一般要求使用 spinlock 的临界区尽量简短，这样获取的锁可以尽快释放，以满足其他忙等的线程。Spinlock 和 mutex 不同，spinlock 不会导致线程的状态切换(用户态->内核态)，但是 spinlock 使用不当(如临界区执行时间过长)会导致 cpu busy 飙高。
- 文件系统
 - ext2：使用索引节点来记录文件信息
 - ext3：使用索引节点来记录文件信息，Ext3 目前所支持的最大 16TB 文件系统和最大 2TB 文件

- ext4 : Ext4 分别支持 1EB (1,048,576TB , 1EB=1024PB , 1PB=1024TB) 的文件系统，以及 16TB 的文件。
- xfs
- 查看文件系统信息：
 - **/etc/mtab** 文件的作用：记载的是现在系统已经装载的文件系统，包括操作系统建立的虚拟文件等；而/etc/fstab 是系统准备装载的
 - **etc/fstab** 文件的作用：记录了计算机上硬盘分区的相关信息，启动 Linux 的时候，检查分区的 fsck 命令，和挂载分区的 mount 命令，都需要 fstab 中的信息，来正确的检查和挂载硬盘。
- Linux 进程相关
 - 组成：进程控制块 PCB , 数据 (段) , 程序 (共享正文段)
 - 进程为进程控制块、正文段、数据段可以实现共享正文，共享数据和可重入
 - 进程通信
 - Linux 进程间通信：管道、 FIFO 、消息队列、 共享内存、 信号量、 套接字(socket) , 文件和记录锁定
 - Linux 线程间通信：互斥量 (mutex) , 信号量 , 条件变量
 - Windows 进程间通信：管道、消息队列、共享内存、信号量 (semaphore) 、套接字(socket)
 - Windows 线程间通信：互斥量 (mutex) , 信号量 (semaphore) 、临界区 (critical section) 、事件 (event)
 - 僵尸进程：一个子进程在其父进程还没有调用 wait() 或 waitpid() 的情况下退出。这个子进程就是僵尸进程。
孤儿进程：一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被 init 进程 (进程号为 1) 所收养，并由 init 进程对它们完成状态收集工作。
僵尸进程将会导致资源浪费，而孤儿则不会。
 - 进程分类：

- 系统进程：可以执行内存资源分配和进程切换等管理工作，该进程运行不受用户影响
- 用户进程：通过执行用户程序，应用程序或内核外的系统程序而产生的进程
 - 交互进程
 - 批处理进程
 - 守护进程
- 交换分区
 - 一般为内存的 1.5 到 2 倍；
- 用户管理相关配置文件
 - /etc/group 设定用户的组名与相关信息
 - /etc/passwd 帐号信息
 - /etc/shadow 密码信息
- 定义 bash 环境的用户文件：
 - bash_profile，是在你每次登录的时候执行。
 - bashrc，是在你新开了一个命令窗口时执行。
- Unix 系统组成：内核、 shell 、文件系统和应用程序。
- 内存碎片：
 - 页式，请求页式：产生内部碎片（内零头），是指进程在向操作系统请求内存分配时，系统满足了进程所需要的内存需求后，还额外还多分了一些内存给该进程，也就是说额外多出来的这部分内存归该进程所有，其他进程是无法访问的。
 - 段式，请求段式：产生外部碎片（外零头），是指内存中存在着一些空闲的内存区域，这些内存区域虽然不归任何进程所有，但是因为内存区域太小，无法满足其他进程所申请的内存大小而形成的内存零头。
- 系统配置文件
 - /sbin/init 在核心完整的加载后，开始运行系统的第一个程序，主要的功能就是准备软件运行的环境，包括系统的主机名称、网络配置、语系处理、文件系统格式及其他服务的启动等。
 - /bin/sh 解释脚本的 shell 命令，开机后运行

- /etc/sysvinit 就是 system V 风格的 init 系统，顾名思义，它源于 System V 系列 UNIX。sysvinit 中运行模式描述了系统各种预订的运行模式。
- /etc/inittab 定义了系统引导时的运行级别，进入或者切换到一个运行级别时做什么。
- Linux 文件权限
 - 10 位分成 4 段，第 1 段 1 位，后面每 3 位一段，分别表示**文件类型/文件所有者权限/文件所有者所在组的权限/其他用户的权限**
- **符号链接**可以建立对于文件和目录的链接。符号链接可以跨文件系统，即可以跨磁盘分区。符号链接的文件类型位是 l，链接文件具有新的 i 节点。
硬链接不可以跨文件系统。它只能建立对文件的链接，硬链接的文件类型位是 -，且硬链接文件的 i 节点同被链接文件的 i 节点相同。
- 管道
 - 一个固定大小的缓冲区，对于管道两端的进程而言，管道就是一个文件系统
 - 半双工的通信机制，可以实现双向数据传输，但是同一时刻最多只能有一个方向的传输
 - 容量大小通常是内存上的一页
 - 管道满的时候，写进程会被阻塞，管道空的时候，读进程会被阻塞
- 内核子系统：
 - 进程管理子系统
 - 内存管理系统
 - I/O 管理系统 Gauhar
 - 文集那管理系统

系统和 Shell 命令

2016 年 9 月 8 日

21:37

- find 命令
 - -mmin n 文件最后一次修改是在 n 分钟之内 : # find . -mmin -60
 - -mtime n 文件最后一次修改是在 n*24 小时之内 : #find / -mtime -1
 - -amin n 文件最后一次访问是在 n 分钟之内
 - -atime n 文件最后一次访问是在 n*24 小时之内
 - -cmin n 文件的状态在 n 分钟内被改变
 - -ctime n 文件状态在 n*24 小时内 (也就是 n 天内) 被改变
 - 仅显示文件 , 不显示文件夹 , 加上 : -type f
 - 当前目录下找.log 结尾的文件 : # find / -name "*.log"
- Linux 查看文件命令
 - cat : 直接跳转到最后一页
 - \$cat file1 file2 , 将两个文件合并到一起输出 ;
 - less : 可以光标上下查看
 - more : 只能 enter 键向下翻页
- tar 命令
 - -x : extract files from an archive 即从归档文件中释放文件 ;
 - -v : verbosely list files processed 即详细列出要处理的文件 ;
 - -z : filter the archive through gzip 即通过 gzip 解压文件 ;
 - -f : use archive file or device ARCHIVE 即使用档案文件或设备 ;
 - 解压到指定目录下 , 在待解压文件名后加上-C (change to directory) 参数。
 - 创建 tar 文件
 - tar -cvf filename.tar directory/file
- Linux 关机命令
 - halt

- init 0
 - poweroff
 - shutdown -h now
- Linux 重启命令
 - reboot , 快速关闭系统 , 可能引起数据丢失
 - shutdown -r now , 可以安全的重启系统
 - init 6
- ifconfig 命令
 - 作用 : ifconfig 用于查看和更改网络接口的地址和参数 , 包括 IP 地址、网络掩码、广播地址 , 使用权限是超级用户。
 - 格式 ifconfig -interface [options] address
 - 主要参数如下 :
 - interface : 指定的网络接口名 , 如 eth0 和 eth1。
 - up : 激活指定的网络接口卡。
 - down : 关闭指定的网络接口。
 - broadcast address : 设置接口的广播地址。
 - pointopoint : 启用点对点方式。
 - address : 设置指定接口设备的 IP 地址。
 - netmask address : 设置接口的子网掩码。
- grep 命令
 - \$ grep -i pattern files : 不区分大小写地搜索。默认情况区分大小写
 - \$ grep -r : 明确要求搜索子目录
 - \$ grep -n : 显示行号
- 用户管理命令
 - 增加用户 : useradd
- history 命令
 - 查看最近的 n 条命令 : \$ history #n
- mysql 相关命令
 - 备份 mysqldump 命令 : mysqldump -h IP -P PORTID -u USERNAME -p DBNAME >bck.sql
 - h 表示主机名或 IP 地址 , 所以 BC 错
 - P 表示数据库连接的 TCP/IP 端口号
 - u 表示用户名而不是数据库名

-p 表示数据库访问密码

- \$相关命令
 - \$# 是传给脚本的参数个数
 - \$0 是脚本本身的名字
 - \$1 是传递给该 shell 脚本的第一个参数
 - \$2 是传递给该 shell 脚本的第二个参数
 - \$@ 是传给脚本的所有参数的列表
 - \$* 是以一个单字符串显示所有向脚本传递的参数，与位置变量不同，参数可超过 9 个
 - \$\$ 是脚本运行的当前进程 ID 号
 - \$? 是显示最后命令的退出状态，0 表示没有错误，其他表示有错误
- traceroute : linux 下侦测主机到目的主机之前所经过的路由的命令
- # 改变权限 chmod 777 filepath
改变所有者 chown test filepath
改变所属组 chgrp user filepath
- netstat : 用于显示各种网络相关信息，如网络连接，路由表，接口状态，连接等信息。
 - -a , 显示所有选项，默认不显示 LISTEN 相关
 - -p , 显示建立相关链接的程序名
 - -n , 拒绝显示别名，能显示数字的全部转化成数字
- 前后台切换命令：
 - fg : 将后台中的命令调至前台继续运行
 - bg : 经一个在后台暂停的命令，变成继续执行
 - Ctrl+z : 可以讲一个正在前台执行的命令放到后台，并暂停
- 统计一个文件 ip.txt 中出现次数最多的前 3 个 ip 和出现次数
 - sort ip.txt | uniq -c | sort -rn | head -n
- 查看网络信息命令
 - tcpdump 是简单可靠网络监控的实用工具
 - netstat 显示网络有关的信息，比如套接口使用情况、路由、接口、协议等
 - ifconfig 是查看活动的网卡信息
- 网关协议
 - 内部网关协议，指在一个自治系统（AS）内部所使用的一种路由协议

- 距离-矢量路由协议，包括路由信息协议（RIP），内部网关路由协议（IGRP）
- 连接状态路由协议，包括开放式最短路径优先协议（OSPF），中间系统到中间系统路由交换协议（IS-IS）
- 高级距离矢量路由协议，包括增强型内部网关路由协议（EIGRP）
 - 外部网关协议

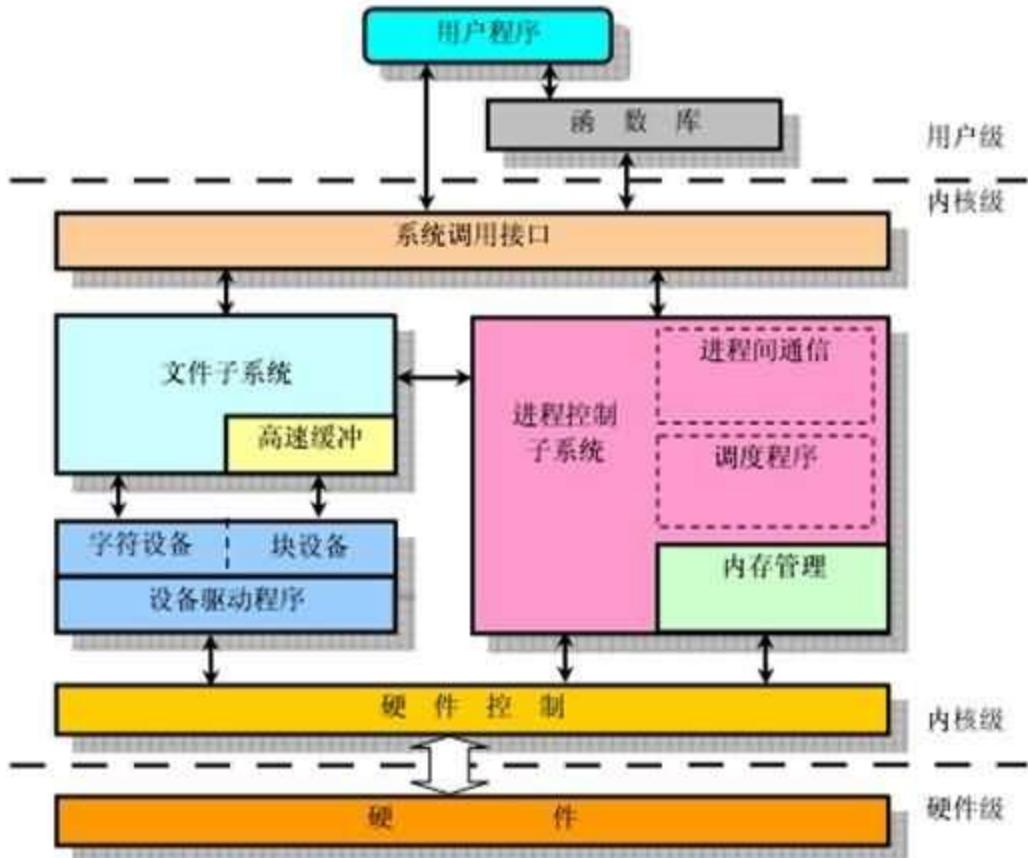
LINUX 内核经典面试题 30 道（附解答）

星期一, 八月 22, 2016

10:08 下午

- 1) Linux 中主要有哪几种内核锁？
- 2) Linux 中的用户模式和内核模式是什么含意？
- 3) 怎样申请大块内核内存？
- 4) 用户进程间通信主要哪几种方式？
- 5) 通过伙伴系统申请内核内存的函数有哪些？
- 6) 通过 slab 分配器申请内核内存的函数有？
- 7) Linux 的内核空间和用户空间是如何划分的（以 32 位系统为例）？
- 8) vmalloc()申请的内存有什么特点？
- 9) 用户程序使用 malloc()申请到的内存空间在什么范围？
- 10) 在支持并使能 MMU 的系统中，Linux 内核和用户程序分别运行在物理地址模式还是虚拟地址模式？
- 11) ARM 处理器是通过几级页表进行存储空间映射的？
- 12) Linux 是通过什么组件来实现支持多种文件系通的？
- 13) Linux 虚拟文件系统的关键数据结构有哪些？（至少写出四个）
- 14) 对文件或设备的操作函数保存在那个数据结构中？

- 15) Linux 中的文件包括哪些?
- 16) 创建进程的系统调用有那些?
- 17) 调用 `schedule()` 进行进程切换的方式有几种?
- 18) Linux 调度程序是根据进程的动态优先级还是静态优先级来调度进程的?
- 19) 进程调度的核心数据结构是哪个?
- 20) 如何加载、卸载一个模块?
- 21) 模块和应用程序分别运行在什么空间?
- 22) Linux 中的浮点运算由应用程序实现还是内核实现?
- 23) 模块程序能否使用可链接的库函数?
- 24) TLB 中缓存的是什么内容?
- 25) Linux 中有哪几种设备?
- 26) 字符设备驱动程序的关键数据结构是哪个?
- 27) 设备驱动程序包括哪些功能函数?
- 28) 如何唯一标识一个设备?
- 29) Linux 通过什么方式实现系统调用?
- 30) Linux 软中断和工作队列的作用是什么?



参考解答如下：

1. 主要有哪几种内核锁？Linux 内核的同步机制是什么？

Linux 的内核锁主要是自旋锁和信号量。

自旋锁：如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋（忙循环，不会引起调用者睡眠）在那里，直到该自旋锁的保持者释放了锁。

自旋锁是专为防止多处理器并发而引入的一种锁，它在内核中大量应用于中断处理等部分（对于单处理器来说，防止中断处理中的并发可简单采用关闭中断的方式，即在标志寄存器中关闭/打开中断标志位，不需要自旋锁）。

`spin_lock_init(x)`

该宏用于初始化自旋锁 `x`。自旋锁在真正使用前必须先初始化。

`spin_lock();`

该宏用于获得自旋锁 `lock`，只有它获得锁才返回。

`spin_trylock(lock)`

该宏尽力获得自旋锁 `lock`，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待 `lock` 被释放。

`spin_unlock();`

该宏释放自旋锁 `lock`

Linux 提供两种信号量：

内核信号量，由内核使用。

IPC 信号量，由用户进程使用。

Linux 内核的信号量在概念和原理上与用户态的 System V 的 IPC 机制信号量是一样的，但是它绝不可能在内核之外使用，因此它与 System V 的 IPC 机制信号量毫不相干。

信号量在创建时需要设置一个初始值，表示同时可以有几个任务可以访问该信号量保护的共享资源，初始值为 1 就变成互斥锁（Mutex），即同时只能有一个任务可以访问信号量保护的共享资源。一个任务要想访问共享资源，首先必须得到信号量，获取信号量的操作将把信号量的值减 1，若当前信号量的值为负数，表明无法获得信号量，该任务必须挂起在该信号量的等待队列等待该信号量可用；若当前信号量的值为非负数，表示可以获得信号量，因而可以立刻访问被该信号量保护的共享资源。当任务访问完被信号量保护的共享资源后，必须释放信号量。释放信号量通过把信号量的值加 1 实现，如果信号量的值为非正数，表明有任务等待当前信号量，因此它也唤醒所有等待该信号量的任务。

`void sema_init (struct semaphore *sem, int val);`

该函数用于数初始化设置信号量的初值，它设置信号量 `sem` 的值为 `val`。

`void down(struct semaphore * sem);`

该函数用于获得信号量 `sem`，它会导致睡眠，

```
int down_interruptible(struct semaphore * sem);
```

该函数功能与 down 类似，不同之处为，down 不会被信号（signal）打断，但 down_interruptible 能被信号打断，因此该函数有返回值来区分是正常返回还是被信号中断，如果返回 0，表示获得信号量正常返回，如果被信号打断，返回-EINTR。

```
int down_trylock(struct semaphore * sem);
```

该函数试着获得信号量 sem，如果能够立刻获得，它就获得该信号量并返回 0，否则，表示不能获得信号量 sem，返回值为非 0 值。因此，它不会导致调用者睡眠，可以在中断上下文使用。

```
void up(struct semaphore * sem);
```

该函数释放信号量 sem，即把 sem 的值加 1.

1. 自旋锁在申请锁失败后，不断忙循环等待锁可用。

2. 信号量在申请失败后，把自己放到等待队列，然后睡眠。

自旋锁可以用在 ISR 中，而 ISR 中不可以使用信号量。但是在中断程序中，就要小心使用，因为当进程拥有一个自旋锁的时候，被中断程序打断，而在中断处理程序中又同样申请该锁，这样就造成了死锁。所以在进程和 中断服务程序都要访问一内核数据的时候，一般只要在进程中或者内核代码中申请自选锁时要禁止中断。就是调用

```
spin_lock_irqsave, spin_unlock_irqrestore 函数。
```

Linux 内核中的同步机制：原子操作、信号量、读写信号量和自旋锁的 API，另外一些同步机制，包括大内核锁、读写锁、大读者锁、RCU (Read-Copy Update，顾名思义就是读-拷贝修改)，和顺序锁。

2. Linux 中的用户模式和内核模式是什么含义？

用户模式是一种受限模式，它对内存和硬件的访问都必须通过系统调用实现，用户程序运行在用户模式。它用于用户进程。内核模式是一种高特权模式，其中的程序代码能直接访问内存和硬件。内核程序运行在内核模式。

3. 怎样申请大块内核内存？

`vmalloc()` 内核用于申请大块内存，特点是线性地址连续，物理地址不一定连续，不能直接用于 DMA。对应的释放函数为 `vfree()`：

`kmalloc()` 内核用于申请小内存，它基于 slab 实现的，slab 是为分配小内存提供的一种高效机制。最多只能开辟大小为 `32XPAGE_SIZE-16` 字节的内存，一般的 `PAGE_SIZE=4kB`，也就是 `128kB-16` 字节的大小的内存，16 个字节是被页描述符结构占用了。`kmalloc` 最大只能开辟 `128k-16`。特点是线性地址连续，物理地址连续。对于要进行 DMA 的设备十分重要。对应的释放函数为 `kfree()`；

4. 用户进程间通信主要哪几种方式？

(1) 管道 (Pipe)：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。

(2) 命名管道 (named pipe)：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令 `mkfifo` 或系统调用 `mkfifo` 来创建。

(3) 信号 (Signal)：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux 除了支持 Unix 早期信号语义函数 `signal` 外，还支持语义符合 Posix.1 标准的信号函数 `sigaction`（实际上，该函数是基于 BSD 的，BSD 为了实现可靠信号机制，又能够统一对外接口，用 `sigaction` 函数重新实现了 `signal` 函数）。

(4) 消息 (Message) 队列：消息队列是消息的链接表，包括 Posix 消息队列 system V 消息队列。有足够的权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

(5) 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用 IPC 形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。

(6) 信号量 (semaphore)：主要作为进程间以及同一进程不同线程之间的同步手段。

(7) 套接字 (Socket)：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由 Unix 系统的 BSD 分支开发出来的，但现在一般可以移植到其它类 Unix 系统上；Linux 和 System V 的变种都支持套接字。

5. 通过伙伴系统申请内核内存的函数有哪些？

在物理页面管理上实现了基于区的伙伴系统（zone based buddy system）。对不同区的内存使用单独的伙伴系统（buddy system）管理，而且独立地监控空闲页。相应接口 alloc_pages(gfp_mask, order), __get_free_pages(gfp_mask, order) 等。

伙伴系统算法

在实际应用中，经常需要分配一组连续的页框，而频繁地申请和释放不同大小的连续页框，必然导致在已分配页框的内存块中分散了许多小块的空闲页框。这样，即使这些页框是空闲的，其他需要分配连续页框的应用也很难得到满足。

为了避免出现这种情况，Linux 内核中引入了伙伴系统算法（buddy system）。把所有的空闲页框分组为 11 个块链表，每个块链表分别包含大小为 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 和 1024 个连续页框的页框块。最大可以申请 1024 个连续页框，对应 4MB 大小的连续内存。每个页框块的第一个页框的物理地址是该块大小的整数倍。

假设要申请一个 256 个页框的块，先从 256 个页框的链表中查找空闲块，如果没有，就去 512 个页框的链表中找，找到了则将页框块分为 2 个 256 个页框的块，一个分配给应用，另外一个移到 256 个页框的链表中。如果 512 个页框的链表中仍没有空闲块，继续向 1024 个页框的链表查找，如果仍然没有，则返回错误。

页框块在释放时，会主动将两个连续的页框块合并为一个较大的页框块。

6) 通过 slab 分配器申请内核内存的函数有？

`kmem_cache_create`/`kmem_cache_alloc` 是基于 slab 分配器的一种内存分配方式，适用于反复分配释放同一大小内存块的场合。首先用 `kmem_cache_create` 创建一个高速缓存区域，然后用 `kmem_cache_alloc` 从该高速缓存区域中获取新的内存块。`kmem_cache_alloc` 一次能分配的最大内存由 `mm/slab.c` 文件中的 `MAX_OBJ_ORDER` 宏定义，在默认的 2.6.18 内核版本中，该宏定义为 5，于是一次最多能申请 $1 \ll 5 * 4KB$ 也就是 128KB 的连续物理内存。分析内核源码发现，`kmem_cache_create` 函数的 `size` 参数大于 128KB 时会调用 `BUG()`。测试结果验证了分析结果，用 `kmem_cache_create` 分配超过 128KB 的内存时使内核崩溃。`kmalloc` 是内核中最常用的一种内存分配方式，它通过调用 `kmem_cache_alloc` 函数来实现。

slab 分配器

slab 分配器源于 Solaris 2.4 的分配算法，工作于物理内存页框分配器之上，管理特定大小对象的缓存，进行快速而高效的内存分配。

slab 分配器为每种使用的内核对象建立单独的缓冲区。Linux 内核已经采用了伙伴系统管理物理内存页框，因此 slab 分配器直接工作于伙伴系统之上。每种缓冲区由多个 slab 组成，每个 slab 就是一组连续的物理内存页框，被划分成了固定数目的对象。根据对象大小的不同，缺省情况下一个 slab 最多可以由 1024 个页框构成。出于对齐等其它方面的要求，slab 中分配给对象的内存可能大于用户要求的对象实际大小，这会造成一定的内存浪费。

7) Linux 的内核空间和用户空间是如何划分的（以 32 位系统为例）？

Linux 内核将这 4G 字节的空间分为两部分。将最高的 1G 字节（从虚拟地址 0xC0000000 到 0xFFFFFFFF），供内核使用，称为“内核空间”。而将较低的 3G 字节（从虚拟地址 0x00000000 到 0xBFFFFFFF），供各个进程使用，称为“用户空间”。因为每个进程可以通过系统调用进入内核，因此，Linux 内核由系统内的所有进程共享。于是，从具体进程的角度来看，每个进程可以拥有 4G 字节的虚拟空间。虽然内核空间占据了每个虚拟空间中的最高 1GB 字节，但映射到物理内存却总是从最低地址（0x00000000）开始。对内核空间来说，其地址映射是很简单的线性映射，0xC0000000 就是物理地址与线性地址之间的位移量。

8) `vmalloc()` 申请的内存有什么特点？

`vmalloc()` 内核用于申请大块内存，特点是线性地址连续，物理地址不一定连续，不能直接用于 DMA。对应的释放函数为 `vfree()`。

9) 用户程序使用 `malloc()` 申请到的内存空间在什么范围？

从虚拟地址 0x00000000 到 0xBFFFFFFF 的用户空间

10) 在支持并使能 MMU 的系统中，Linux 内核和用户程序分别运行在物理地址模式还是虚拟地址模式？

Linux 内核和用户程序都运行在虚拟地址模式。

对于没有 MMU 的系统实际上用户空间和内核空间是不做区分的，如果一定要分用户空间和内核空间也只是形式上的。

uClinux 同标准 Linux 的最大区别就在于内存管理。标准 Linux 是针对有 MMU 的处理器设计的。在这种处理器上，虚拟地址被送到 MMU，MMU 把虚拟地址映射为物理地址。通过赋予每个任务不同的虚拟—物理地址转换映射，支持不同任务之间的保护。对于 uCLinux 来说，其设计针对没有 MMU 的处理器，不能使用处理器的虚拟内存管理技术。

11) ARM 处理器是通过几级页表进行存储空间映射的？

ARM 处理器采用两级页表实现地址映射：

1) 一级页表中包含以段为单位的地址变换条目或者指向二级页表的指针，一级页表实现的地址映射粒度较大。

2) 二级页表中分类有大页、小页和极小页为单位的地址变换条目。

12) Linux 是通过什么组件来实现支持多种文件系统的？

虚拟文件系统（Virtual File System，简称 VFS），是 Linux 内核中的一个软件层，用于给用户空间的程序提供文件系统接口；同时，它也提供了内核中的一个抽象功能，允许不同的文件系统共存。系统中所有的文件系统不但依赖 VFS 共存，而且也依靠 VFS 协同工作。

13) Linux 虚拟文件系统的关键数据结构有哪些？（至少写出四个）Linux 虚拟文件系统的关键数据结构有哪些？（至少写出四个）

超级块对象存储一个已安装的文件系统的控制信息，代表一个已安装的文件系统；

```
struct super_block;
```

索引节点对象存储了文件的相关信息，代表了存储设备上的一个实际的物理文件。。当一个文件首次被访问时，内核会在内存中组装相应的索引节点对象，以便向内核提供对一个文件进行

操作时所必需的全部信息；

```
struct inode;
```

目录项的概念主要是出于方便查找文件的目的。一个路径的各个组成部分，不管是目录还是普通文件，都是一个目录项对象。如，在路径/home/source/test.c 中，目录 /, home, source 和文件 test.c 都对应一个目录项对象。不同于前面的两个对象，目录项对象没有对应的磁盘数据结构，VFS 在遍历路径名的过程中现场将它们逐个地解析成目录项对象。目录项的概念主要是出于方便查找文件的目的。一个路径的各个组成部分，不管是目录还是普通的文件，都是一个目录项对象。

文件对象是已打开的文件在内存中的表示，主要用于建立进程和磁盘上的文件的对应关系。它由 sys_open() 现场创建，由 sys_close() 销毁。

```
struct file
```

14) 对文件或设备的操作函数保存在那个数据结构中？

```
struct file_operations;
```

15) Linux 中的文件包括哪些？

有执行文件，普通文件，目录文件，链接文件和设备文件，管道文件。

16) 创建进程的系统调用有那些?

clone(), fork(), vfork()

fork()创造的子进程复制了父亲进程的资源，使用写时复制技术，子进程与父进程使用相同物理页，只有子进程试图写一个物理页时，才 copy 这个物理页到一个新的物理页，子进程使用新的物理页，子进程与物理地址的内存地址空间分开，开始独立运行。

vfork()创造的子进程完全运行在父进程的地址空间上，子进程对虚拟地址空间任何数据的修改都为父进程所见。这与 fork 是完全不同的，fork 进程是独立的空间。为了防止父进程重写子进程需要的数据，父进程会被阻塞，直到子进程执行 exec() 和 exit()。

clone()可以有选择性的继承父进程的资源，可以选择像 vfork 一样和父进程共享一个虚存空间，从而使创造的是线程，你也可以不和父进程共享，你甚至可以选择创造出来的进程和父进程不再是父子关系，而是兄弟关系

系统调用服务例程 sys_clone, sys_fork, sys_vfork 三者最终都是调用 do_fork 函数完成。

17) 调用 schedule() 进行进程切换的方式有几种?

- 一 系统调用 do_fork()
- 二 定时钟断 do_timer()
- 三 唤醒进程 wake_up_process()
- 四 改变进程的调度策略 setscheduler()
- 五 系统调用礼让 sys_sched_yield()

参考：

一 系统调用 do_fork()

- 1 当前进程调用 fork() 创建子进程，进入 kernel
- 2 当前进程分一半多时间片给子进程，
- 3 如果当前进程时间片剩余为 0，设定当前进程 need_sched=1，
- 4 从系统调用退出
- 5 到达 ret_from_sys_call
- 6 到达 ret_with_reschedule
- 7 发现当前进程要求调度，跳转到 reschedule
- 8 调用 schedule()
- 9 schedule() 处理当前进程的调度要求，
- 10 如果有其他进程可运行，将在 schedule() 内发生切换。

二 定时钟断 do_timer()

- 11 当定时钟断发生时 8235->irq0->do_timer_interrupt() -> do_timer()
- 12 -> update_process_times() 递减当前进程的时间片，
- 13 如果当前进程时间片为 0，设定当前进程 need_sched=1，

14 从中断调用退出，
15 到达 `ret_from_intr`
16 到达 `ret_with_reschedule`，
17 发现当前进程要求调度，跳转到 `reschedule`
18 调用 `schedule()`
19 `schedule()` 处理当前进程的调度要求，
20 如果有其他进程可运行，将在 `schedule()` 内发生切换。

三 唤醒进程 `wake_up_process()`

21 当前进程调用 `fork()` 创建子进程，进入 kernel
22 当前进程调用了 `wake_up_process` 来唤醒进程 x
23 使进程 x 状态为 RUNNING，并加入 runqueue 队列，
24 调用 `reschedule_idle()`
25 发现进程 x 比当前进程更有资格运行，设定当前进程 `need_sched=1`，
26 从系统调用退出
27 到达 `ret_from_sys_call`
28 到达 `ret_with_reschedule`
29 发现当前进程要求调度，跳转到 `reschedule`
30 调用 `schedule()`
31 `schedule()` 处理当前进程的调度要求，
32 如果有其他进程可运行，将在 `schedule()` 内发生切换。这次大多数可能切换到进程 x

四 改变进程的调度策略 `setscheduler()`

- 33 进入系统调用 `setscheduler()`
- 34 改变进程 x 的调度策略
- 35 提前进程 x 在 `runqueue` 队列的位置
- 36 设定当前进程 `need_sched=1`,
- 37 从系统调用退出
- 38 到达 `ret_from_sys_call`
- 39 到达 `ret_with_reschedule`
- 40 发现当前进程要求调度，跳转到 `reschedule`
- 41 调用 `schedule()`
- 42 `schedule()` 处理当前进程的调度要求，
- 43 如果有其他进程可运行，将在 `schedule()` 内发生切换。

五 系统调用礼让 `sys_sched_yield()`

- 44 进入系统调用 `sys_sched_yield()`
- 45 如果有其他的进程，进行礼让，
- 46 设定当前进程 `need_sched=1`,

47 从系统调用退出
48 到达 `ret_from_sys_call`
49 到达 `ret_with_reschedule`
50 发现当前进程要求调度，跳转到 `reschedule`
51 调用 `schedule()`
52 `schedule()` 处理当前进程的调度要求，
53 如果有其他进程可运行，将在 `schedule()` 内发生切换。
`need_sched` 表示 CPU 从系统空间返回到用户空间前夕要进行一次调度。

18) Linux 调度程序是根据进程的动态优先级还是静态优先级来调度进程的？

Linux 调度程序是根据进程的动态优先级来调度进程的，但动态优先级又是依据静态优先级通过算法得到的，两者是两个相关连的值。

因为高优先级的进程总比低优先级的进程先被调度，为防止有多个高优先级且一直占用 CPU 资源，导致其它进程不能占用 CPU，所以引用动态优先级概念。

19) 进程调度的核心数据结构是哪个？

`struct runqueue;`

20) 如何加载、卸载一个模块？

通过命令 `insmod` 加载一个模块，通过命令 `rmmod` 卸载一个模块。

21) 模块和应用程序分别运行在什么空间？

模块运行在内核空间，应用程序运行在用户空间。

22) Linux 中的浮点运算由应用程序实现还是内核实现？

由应用程序实现，Linux 中的浮点运算由数学库函数实现，库函数能被应用程序链接后调用，不能被内核链接调用。

这些运算是在应用程序中进行的，然后再将运算结果反馈给系统。linux 内核如果一定要进行浮点运算，需要在建立内核时选上 `math-emu`，使用软件模拟浮点运算。据说这样作的代价有两个：

1. 用户在安装驱动时需要重建内核；
2. 可能会影响到其他应用程序，使得这些应用程序在进行浮点运算时也使用 `math-emu`，会严重降低效率。

23) 模块程序能否使用可链接的库函数？

模块程序运行在内核空间，不能链接库函数。

24) TLB 中缓存的是什么内容？

TLB(Translation lookaside buffer)即旁路转换缓冲，称为页表缓冲，当线性地址第一次被转换为物理地址时，将线性地址与物理地址存放到 TLB 中，用于下次访问这个线性地址时，加快转换速度。

25) Linux 中有哪几类设备？

可以分为字符设备和块设备。网卡是例外，它不直接与设备文件对应。`mknod()` 系统调用用来创建设备文件。

26) 字符设备驱动程序的关键数据结构是哪个？

字符设备描述符 `struct cdev;` `cdev_alloc()` 用于动态分配 `cdev` 描述符。`cdev_add()` 注册一个 `cdev` 描述符。`Cdev` 包含一个 `struct kobject` 类型的数据结构，它是设备驱动程序的核心数据结构。

27) 设备驱动程序包括哪些功能函数？

`open()`, `read()`, `wirte()`, `ioctl()`, `release()`, `llseek()`

28) 如何唯一标识一个设备？

Linux 使用设备编号来唯一标识一个设备，设备编号分为两部分：主设备号和次设备号。一般主设备号标识设备对应的驱动程序，次设备号用于确定设备文件指向的设备。在内核中使用数据结构 `dev_t` 表示设备编号，一般它是 32 位长度，其中 12 位用于表示主设备号，后 20 位用于表示次设备号。`MKDEV(int major, int minor)` 用于生成一个 `dev_t` 类型的对象。

29) Linux 通过什么方式实现系统调用？

靠软件中断实现的。首先，用户程序为系统调用设置参数。其中一个参数是系统调用编号。参数设置完成后，程序执行“系统调用”指令。`x86` 系统上的软中断由 `int` 产生。这个指令会导致一个异常：产生一个事件，这个事件会致使处理器切换到内核态并跳转到一个新的地址，并开始执行那里的异常处理程序。此时的异常处理程序实际上就是系统调用处理程序。

30) Linux 软中断和工作队列的作用是什么？

Linux 软中断和工作队列的作用是中断处理。

1 软中断一般是“可延迟函数”的总称，它不能睡眠、不能阻塞。它处于中断上下文，不能进程切换，软中断不能被自己打断，只能被硬件中断打断（上半部），可以并发运行在多个 CPU 上（即使同一类型的也可以）。所以软中断必须设计为可重入的函数（允许多个 CPU 同时操作），因此也需要使用自旋锁来保护其数据结构。

2 工作队列中的函数处于进程上下文中，它可以睡眠，能被阻塞。能够在不同的进程间切换，以完成不同的工作。

可延迟函数和工作队列中的函数都不能访问进程的用户空间。可延迟函数执行时不可能有任何正在运行的进程。工作队列的函数由内核进程执行，它不能访问用户空间地址。

数据结构和算法

2016 年 3 月 28 日

12:59

1. 二分查找-Binary Search

二分查找找到目标值返回目标值位置，没找到返回应该插入的位置；

如果升序，则返回位置的值大于目标值，如果降序，则小于目标值。

2. 检查字符串 s2 中出现的字符是不是都在 s1 中出现过，除了用 hash 表，还可以用素数表示法，然后做除法的方式。为每个字母分配一个素数，相乘相除，利用素数除数的性质。

PS：该方法可以通用在很多的字符问题上

3. 先序遍历和中序遍历建树

4. 中序遍历和后序遍历建树

5. 经典动态规划问题

- a. 最长公共子序列

$$dp[i][j] = \begin{cases} dp[i-1][j-1] + 1, & str1[i] == str2[j] \\ \max(dp[i-1][j], dp[i][j-1]), & str1[i] != str2[j] \end{cases}$$

2. 最长公共子串

$$dp[i][j] = \begin{cases} dp[i-1][j-1], & str1[i] == str2[j] \\ 0, & str1[i] != str2[j] \end{cases}$$

6. B 树和 B+ 树的区别

- a. B 树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而 B+ 树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持 B+ 树的平衡。
 - b. 因为 B 树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+ 树相比来说是一种较好的折中。
 - c. B 树的查询效率与键在树中的位置有关，最大时间复杂度与 B+ 树相同（在叶结点的时候），最小时间复杂度为 1（在根结点的时候）。而 B+ 树的时候复杂度对某建成的树是固定的。

7. 求前 K 大个数
8. 红黑树和 AVL 树
9. 桶排序-相邻最大差值
10. 二叉树叶子节点的最近公共父节点
11. 八皇后
12. 最大子数组
13. 把数组中的前 k 个元素循环左移到尾部
14. 两个栈，把其中一个栈顺序存放的元素（栈顶最小，栈底最大）逆序后，存放到原来的栈中

数据结构

2016 年 9 月 6 日

14:09

- 二叉树
 - 二叉树的性质
 - 性质 1: 在二叉树的第 i 层上至多有 $2^{(i-1)}$ 个结点 ($i > 0$)。
比如第 3 层，那么就是 $2^{(3-1)} = 4$ ，第 3 层最多有 4 个节点
 - 性质 2: 深度为 k 的二叉树至多有 $2^k - 1$ 个结点 ($k > 0$)。
比如第 3 层，最多有 $2^3 - 1 = 7$ 个节点
 - 性质 3: 对于任何一棵二叉树，若 2 度的结点数有 n_2 个，则叶子数 (n_0) 必定为 $n_2 + 1$ (即 $n_0 = n_2 + 1$)
度为 2 的节点有 3 个 (A B C)，那么 $n_2 = 3$
叶子数 = $n_2 + 1$ ，那么就是 $3 + 1 = 4$ 个
 - 性质 4: 具有 n 个结点的完全二叉树的深度必为 $\log_2 n$ (向下取整) + 1
 $\log_2 n + 1$ 转换为公式是

$2^x = n$, 此时 n 为 7, 所以 x 约等于 2.7 次方, 向下取整后得

2

深度 = $x + 1$, $2 + 1 = 3$, 所以深度结果就是 3

- 性质 5: 对完全二叉树, 若从上至下、从左至右编号, 则编号为 i 的结点, 其左孩子编号必为 $2i$, 其右孩子编号必为 $2i + 1$; 其双亲的编号必为 $i/2$ ($i = 1$ 时为根, 除外)。
- 平衡二叉树
 - 平衡二叉树是一个每个节点的左右子树的高度差绝对值小于等于 1 的二叉排序树。
- 二叉堆
 - 插入节点时间复杂度为 $O(\log n)$
 - 删除节点时间复杂度为 $O(\log n)$
 - 查询最小元素的复杂度是 $o(1)$
 - 合并两个堆的复杂度是 $o(\lg n)$
- 查找
 - 静态查找
 - 顺序查找
 - 有序表查找 (折半查找)
 - 索引顺序查找
 - 动态查找
 - 动态查找表的特点是表结构本身在查找过程中动态生成, 如二叉查找树
- 连通图和强连通图
 - 连通图: 图中任意两个顶点是联通的, n 个顶点的连通图最少需要 $n*(n-1)/2$ 条边
 - 强连通图: 有向图中任意两个顶点是联通的, n 个定点的强连通图需要最少 n 条边 (形成一个环)

算法

2016 年 9 月 8 日

22:54

- 递归
 - 递归深度计算：阿克曼函数

- 前 K 个数的快排实现

```

class Solution {
    public ArrayList<Integer> GetLeastNumbers_Solution(int [] input, int k) {
        int len = input.length;
        ArrayList<Integer> ret = new ArrayList<>();
        if(len == 0 || k > len || k <= 0)return ret;

        int start = 0, end = len - 1;
        int idx = Partition(input, start, end);
        while(idx != k - 1){
            if(idx > k - 1){
                end = idx - 1;
                idx = Partition(input, start, end);
            }
            else{
                start = idx + 1;
                idx = Partition(input, start, end);
            }
        }

        for(int i = 0; i < k; i++){
            ret.add(input[i]);
        }
        return ret;
    }

    private int Partition(int[] input, int start, int end){
        int key = input[start];
        int lp = start, rp = end;
    }
}

```

```

while(lp < rp){
    while(lp < rp && input[rp] > key)rp--;
    input[lp] = input[rp];
    while(lp < rp && input[lp] < key)lp++;
    input[rp] = input[lp];
}
input[lp] = key;
return lp;
}
}

```

- 素数求法
- 前序中序求后序，后序中序求前序
- LCA，最近公共祖先
<https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.03.md>
- 连续子序列最大积
- 最大公约数
- 最小公倍数
- 阶乘最后一个非零位的数

Java 六种排序算法实现

2016年8月14日

18:49

排序方法	时间复杂度	最好情况	最坏情况	辅助空间	稳定性	总结
------	-------	------	------	------	-----	----

冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定	
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n) - O(n)$	不稳定	快速排序每次优先处理较短的子序列，可以减少递归占用的内存空间
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定	堆排序中，构建堆的时间是 $O(n)$ ，重建堆的时间是 $O(\log n)$

```

class mySort
{
    /*
    冒泡排序
    1. 比较相邻的元素，如果第一个比第二个大，则交换；
    2. 对第 0 个到第 n-1 个元素做相同的工作，这时，最大的元素就浮到了数组最后的位置；
    3. 针对所有的元素重复上述动作，除了最后一个；
    4. 持续每次对越来越少的元素重复上述动作，直到没有任何一个数字需要比较。
    */
    public void bubbleSort(int[] nums)
    {
        int len = nums.length;
        for(int i = 0; i < len; i++)
        {
            for(int j = 1; j < len - i; j++)
            {
                if(nums[j-1] > nums[j])
                {
                    int tmp = nums[j-1];
                    nums[j-1] = nums[j];
                    nums[j] = tmp;
                }
            }
        }
    }
}

```

```
        nums[j] = tmp;
    }
}
return;
}
```

/*

选择排序

1. 在未排序的序列中找到最小的元素，存放到排序序列的起始位置；
2. 再从剩余的未排序元素中寻找最小的元素，放到已排序序列的末尾；
3. 重复步骤 2，直到所有的元素都已经排好序。

*/

```
public void selectionSort(int[] nums)
{
    int len = nums.length;
    for(int i = 0; i < len; i++)
    {
        int min = i;
        for(int j = i+1; j < len; j++)
        {
            if(nums[j] < nums[min])min = j;
        }
        int tmp = nums[min];
        nums[min] = nums[i];
        nums[i] = tmp;
    }
    return;
}
```

/*

插入排序

1. 从第一个元素开始，认为该元素已排好序；
2. 取出下一个元素，在已经排好序的元素序列中从后向前扫描；
3. 如果被扫描的元素大于新元素，该元素后移；
4. 重复步骤 3，直到找到已排序好序列中不大于新元素的位置；
5. 将新元素插入到所找到位置的后面；
6. 重复步骤 2-5.

*/

```
public void insertSort(int[] nums)
{
    int len = nums.length;
    for(int i = 1; i < len; i++)
    {
        int tmp = nums[i];
```

```

        int j = i - 1;
        for(; j >= 0; j--)
        {
            if(nums[j] <= tmp)break;
            else
            {
                nums[j+1] = nums[j];
            }
        }
        nums[j+1] = tmp;
    }
    return;
}

```

/*

归并排序

思想：分治法，先递归分解数组再合并数组。

```

*/
public void mergeSort(int[] nums, int start, int end)
{
    int len = end - start + 1;
    if(len <= 1) return;
    int mid = start + len / 2 - 1;
    mergeSort(nums, start, mid);
    mergeSort(nums, mid+1, end);
    merge(nums, start, mid, end);
    return;
}

public void merge(int[] nums, int start, int mid, int end)
{
    int[] tmpNums = new int[nums.length];
    int left = start, right = mid + 1;
    int tmpIndex = start;
    while(left <= mid && right <= end)
    {
        if(nums[left] < nums[right])
        {
            tmpNums[tmpIndex++] = nums[left++];
        }
        else tmpNums[tmpIndex++] = nums[right++];
    }
    while(left <= mid)tmpNums [tmpIndex++] =
nums [left++];
    while(right <= end)tmpNums [tmpIndex++] =
nums [right++];
    for(int i = start; i <= end; i++)nums[i] =
tmpNums[i];
}

```

```
        return;
    }

/*
快速排序 ( 先处理右指针 , 再处理左指针 )
```

1. 从序列中跳出一个元素作为基准元素
2. 分区过程 , 将比基准数大的放到右边 , 小于或者等于基准数的元素放到左边
3. 再对左右分区递归执行 2 , 直至个区间只有一个元素

```
Public void quickSort(int[] nums, int start, int end)
{
    if (start>=end) return;
    int lp=start, rp=end;
    int key=nums[start];
    while (lp<rp)
    {
        while (lp<rp&&nums[lp]>key) rp--;
        nums[lp]=nums[rp];
        while (lp<rp&&nums[lp]<key) lp++;
        nums[rp]=nums[lp];
    }
    nums[lp]=key;
    quickSort(nums, start, lp-1);
    quickSort(nums, lp+1, end);
    return;
}
```

堆排序

性质 :

1. 父节点的值总是大于或者等于 (小于或者等于) 任何一个子节点的值 ;

2. 每个节点的左右子树都是一个二叉堆 (大堆或者小堆) 。

步骤 :

1. 构造最大堆 : 若数组下标范围是 $0-n$, 考虑到单独一个元素是大根堆 , 则从下标 $n/2$ 开始的元素均为大根堆 (叶子节点) ,

于是只要从 $n/2-1$ 开始 , 依次向前构造大根堆 , 就可以保证 , 构造到某个节点时 , 它的左右子树都已经是大根堆。

2. 堆排序 : 由于堆使用数组模拟的。得到一个大根堆后 , 数组内部并不是有序的。因此需要将堆化数组有序化。

思想是移除根节点 , 并做最大堆调整的递归运算。

第一次将 $\text{heap}[0]$ 和 $\text{heap}[n-1]$ 交换，再对 $\text{heap}[0 \dots n-2]$ 做最大堆调整。

第二次将 $\text{heap}[0]$ 和 $\text{heap}[n-2]$ 交换，再对 $\text{heap}[0 \dots n-3]$ 做最大堆调整。

重复以上操作直至 $\text{heap}[0]$ 和 $\text{heap}[1]$ 交换。

由于每次都是将最大的数并入到后面的有序区间，故操作完后整个数组都是有序的。

3. 最大堆调整：该方法是提供给上述两个过程调用。目的是将堆的末端子节点作调整，使得子节点永远小于父节点。

```
/*
public void heapSort(int[] nums)
{
    int len = nums.length;
    int first = len / 2 - 1;    //最后一个非叶子节点的下标
    for(int j = first; j >= 0; j--)
    {
        maxHeapify(nums, j, len-1);    //从最后一个非叶子
        节点开始，从后向前依次进行最大堆调整
    }
    for(int i = len - 1; i > 0; i--)
    {
        int tmp = nums[0];
        nums[0] = nums[i];
        nums[i] = tmp;
        maxHeapify(nums, 0, i-1);
    }
    return;
}

/*
最大堆调整
*/
public void maxHeapify(int[] nums, int start, int end)
{
    int root = start;
    while(true)
    {
        int child = root * 2 + 1;    //加1是因为数组下标
        从0开始，不是从1开始
        if(child > end) break;
        if(child + 1 <= end && nums[child] <
        nums[child+1]) child = child + 1;
    }
}
```

```
        if (nums[root] < nums[child])
    {
        int tmp = nums[root];
        nums[root] = nums[child];
        nums[child] = tmp;
        root = child;
    }
    else break;
}
return;
}
```

计算机网络

2016 年 3 月 28 日
12:59

- ping 使用什么协议
ICMP 协议
- TCP 和 UDP 的区别
 1. TCP 提供可靠的报文传输 , UDP 传输不可靠的数据报传输
 2. TCP 提供窗口协议和流量控制 , UDP 不提供 , TCP 使用滑动窗口机制来实现流量控制 , 通过动态改变窗口的大小进行拥塞控制
 3. TCP 提供面向连接 (3 次握手 , 4 次挥手) , UDP 无连接
 4. TCP 有序 , UDP 无序
 5. TCP 慢 , UDP 快
 6. TCP 开销比 UDP 大
 7. TCP 不支持广播 , UDP 支持广播

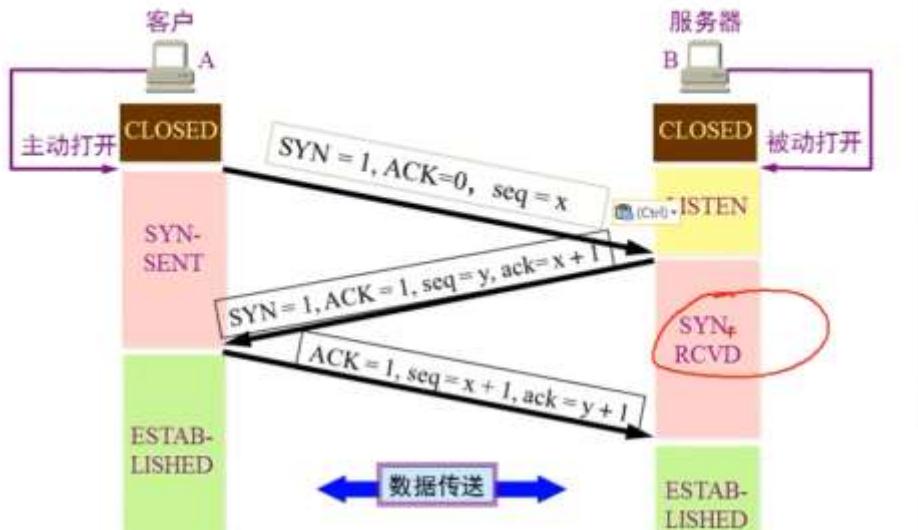
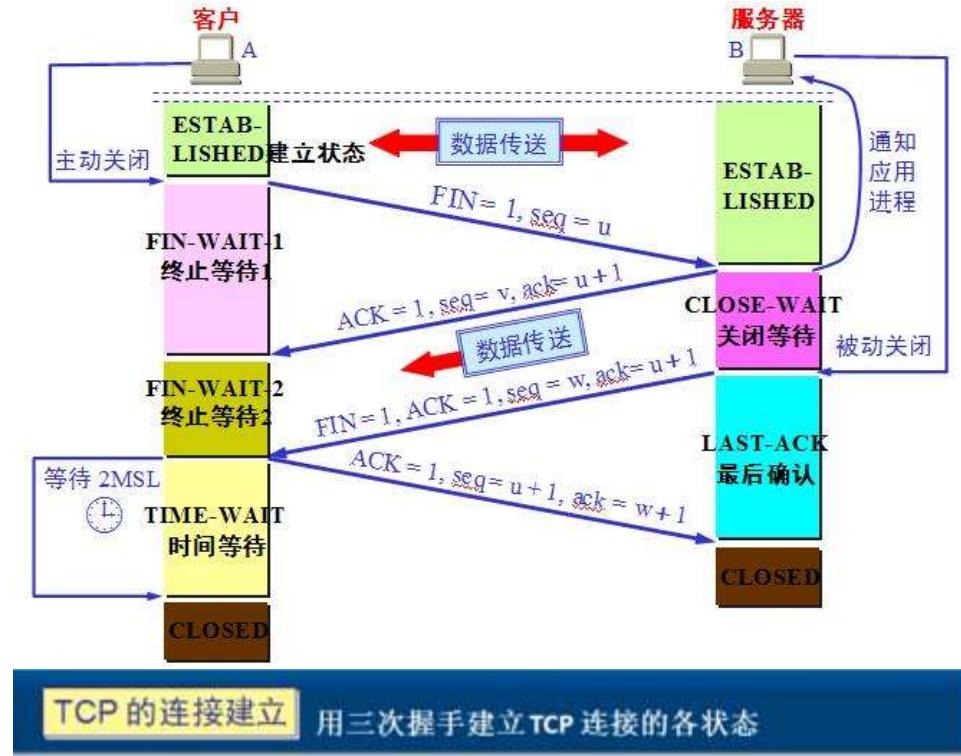
TCP 加快传输效率方法 : 采用一块确认机制。

TCP 拥塞处理 :

- 慢启动
- 避免拥塞
- 拥塞发生
- 快恢复

- TCP 拥塞控制
- 3 次握手，4 次挥手
 - 3 次握手：
 - i. 客户端向服务器发送连接请求 SYN J；
 - ii. 服务器确认客户端请求 ACK J+1，并发送连接请求 SYN K；
 - iii. 客户端确认服务器请求 ACK K+1。
 - 为什么需要三次握手，两次为什么不可以？
 - 为了防止已失效的连接请求报文段突然有传到了服务端而产生错误；
 - “已失效的连接请求报文段”的产生在这样一种情况下：client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 server。本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于是就向 client 发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。”
 - 4 次挥手：
 - i. 客户端向服务器发送关闭请求 FIN M；
 - ii. 服务器确认客户端关闭请求 ACK M+1；
 - iii. 服务器向客户端发送关闭请求 FIN N；
 - iv. 客户端确认服务器关闭请求 ACK N+1。
 - 为什么需要四次挥手？
 - 为了确保服务端在客户端已经单方面断开与服务端的连接后，服务端剩余的数据仍然能够发往客户端；

- 只有在服务端也向客户端申请断开且客户端确认后，服务端才最终断开连接。
- Time_Wait 状态怎么产生，该状态会出现在 TCP 的哪一端（服务器，客户端）？！



- TCP 的 RST 报文（复位报文）：
 - 端口未打开
 - 请求超时

- 提前关闭
 - 在一个已关闭的 socket 上出现数据
-
- 网络编程的步骤
对于 TCP 连接：
1.服务器端 1) 创建套接字 create ; 2) 绑定端口号 bind ; 3) 监听连接 listen ; 4) 接受连接请求 accept , 并返回新的套接字 ; 5) 用新返回的套接字 recv/send ; 6) 关闭套接字。
2.客户端 1) 创建套接字 create; 2) 发起建立连接请求 connect; 3) 发送 /接收数据 send/recv ; 4) 关闭套接字。

TCP 总结：

Server 端 : create -- bind -- listen-- accept-- recv/send-- close
Client 端 : create----- connct-----send/recv-----close.

对于 UDP 连接：

1.服务器端:1) 创建套接字 create ; 2) 绑定端口号 bind ; 3) 接收/发送消息 recvfrom/sendto ; 4) 关闭套接字。
2.客户端:1) 创建套接字 create ; 2) 发送/接收消息 sendto/recvfrom ; 3) 关闭套接字.

UDP 总结:

Server 端 : create----bind ----recvfrom/sendto---close
Client 端 : create---- sendto/recvfrom----close.

- 1XX 消息
2XX 成功
3XX 重定向
400 Bad Request (坏请求)
403 Forbidden(禁止)
405 Method Not Allowed(不允许使用的方法)
411 Length Required (要求长度指示)
413 Request Entity Too Large (请求实体太大)

414 Request URI Too Long(请求 URI 太长)

500 Internal Server Error(内部服务器错误)

501 Not Implemented(未实现)

502 Bad Gateway (网关故障)

505 HTTP Version Not Supported(不支持的 HTTP 版本)

- 如果客户端不断发送请求连接会怎么样？

- 会发生 DDoS 攻击，并且浪费资源；

针对 3 次握手的恶意攻击：DDoS 攻击

假设一个用户向服务器发送了 SYN 报文后突然死机或掉线，那么服务器在发出 SYN+ACK 应答报文后是无法收到客户端的 ACK 报文的（第三次握手无法完成），这种情况下服务器端一般会重试（再次发送 SYN+ACK 给客户端）并等待一段时间后丢弃这个未完成的连接，这段时间的长度我们称为 SYN Timeout，一般来说这个时间是分钟的数量级（大约为 30 秒-2 分钟）；一个用户出现异常导致服务器的一个线程等待 1 分钟并不是什么很大的问题，但如果有一个恶意的攻击者大量模拟这种情况，服务器端将为了维护一个非常大的半连接列表而消耗非常多的资源----数以万计的半连接，即使是简单的保存并遍历也会消耗非常多的 CPU 时间和内存，何况还要不断对这个列表中的 IP 进行 SYN+ACK 的重试。实际上如果服务器的 TCP/IP 栈不够强大，最后的结果往往是堆栈溢出崩溃---即使服务器端的系统足够强大，服务器端也将忙于处理攻击者伪造的 TCP 连接请求而无暇理睬客户的正常请求（毕竟客户端的正常请求比率非常之小），此时从正常客户的角度看来，服务器失去响应，这种情况我们称做：服务器端受到了 SYN Flood 攻击（SYN 洪水攻击）。

- DDoS 预防：

- 确保服务的系统文件是最新版本，并保持更新；
 - 关闭不必要的服务；
 - 限制同时打开 SYN 的半连接数目；
 - 缩短 SYN 半连接 time out 的时间；
 - 正确设置防火墙；
 - 禁止对主机的非开放服务的访问；
 - 限制特定 IP 短地址的访问；

- 启用防火墙防止 DDoS 属性；
 - 严格限制对外开放的服务器的向外访问；
 - 运行端口映射程序或端口扫描程序，要认真检查特权端口和非特权端口；
 - 认真检查网络设备和主机/服务器系统的日志；
 - 限制在防火墙外与网络文件共享。
-
- DNS 怎么寻找域名
 - DNS 解析过程：
 - 客户端查询浏览器的缓存中有没有该域名对应地址的缓存，没有进行下一步；
 - 查询电脑中的 DNS 缓存，找到返回，找不到进行下一步；
 - 向电脑中配置的本地域名服务器申请解析域名，如果本地域名服务器解析失败（没有缓存该域名地址），下一步；
 - 本地域名服务器直接向根域名服务器申请解析；
 - 根域名服务器返回本地域名服务器一个顶级域名服务器地址；
 - 本地域名服务器向顶级域名服务器申请解析域名；
 - 顶级域名服务器查找并返回此域名对应的主域名服务器地址给本地域名服务器；
 - 本地域名服务器向主域名服务器请求解析域名地址；
 - 主域名服务器收到请求，在缓存中查找域名地址，如果没找到，则进入下一级域名服务器查询，重复直至找到地址；
 - 主域名服务器找到域名 IP 地址后，返回给本地域名服务器；
 - 本地域名服务器缓存主域名服务器返回的地址，并将地址反馈给客户端。
 - **DNS 劫持**，指用户访问一个被标记的地址时，DNS 服务器故意将此地址指向一个错误的 IP 地址的行为。范例就是收到各种推送广告等网站。
 - **DNS 污染**，指的是用户访问一个地址，国内的服务器(非 DNS)监控到用户访问的已经被标记地址时，服务器伪装成 DNS 服务器向用户发回错误的地址的行为。比如不能访问 Google、YouTube 等。
 - www.baidu.com 描述一次访问过程，如果在国外访问呢？怎么找到服务器？防火墙会怎么处理？

- 从输入网址到获得页面的过程
 - 查询 DNS，获取域名对应的 IP 地址；
 - 浏览器搜索自身的 DNS 缓存；
 - 搜索操作系统的 DNS 缓存；
 - 读取本地 HOST 文件；
 - 发起一个 DNS 系统调用
 - 宽带运营服务器查看本身缓存
 - 运营商服务器发起一个迭代 DNS 解析请求
 - 浏览器获得域名对应的 IP 地址后，发起 HTTP 三次握手请求；
 - TCP/IP 连接建立起来后，浏览器可以向服务器发送 HTTP 请求；
 - 服务器接收到这个请求，根据路径参数，经过后端的一些处理生成 HTML 页面代码返回给浏览器；
 - 浏览器拿到完整的 HTML 页面代码开始解析和渲染，如果有外部应用的 js/css 等，同样是 HTTP 请求，需要经过以上步骤；
 - 浏览器根据拿到的资源对页面进行渲染，最终把一个完整的页面呈现给用户。
- Http 会话的四个过程
 - 建立 TCP 连接
 - 发出请求信息
 - 回应响应信息
 - 释放 TCP 连接
- Http 和Https 的区别
 - **Http 是 HTTP 协议运行在 TCP 之上，所有的传输内容都是明文，客户端和服务器都无法验证对方的身份；**
 - **Https 是 HTTP 运行在 SSL/TLS 之上，SSL/TLS 运行在 TCP 之上，所有传输的内容都是加密的，加密采用对称加密，但对称加密的密钥用服务器房的证书进行了非对称加密。此外客户端可以验证服务器端的身份，如果配置了客户端验证，服务器方也可以验证客户端的身份；**
 - **Https 协议需要用到 ca 申请证书，一般免费证书很少，需要交费；**

- Http 是超文本传输协议，是明文传输的，https 则是具有安全性的 **ssl 加密传输协议**；
 - Http 和 https 使用的是完全不同的连接方式，用的端口也不一样，http 是 **80** 端口，https 是 **443** 端口；
 - Http 连接是简单的，无状态的；
 - Https 协议是由 SSL+HTTP 协议构建而成的可进行加密传输、身份认证的网络协议，比 http 要更加安全。
-
- Http1.0、Http1.1、Http2.0
 - 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。
 - HTTP1.1：
 - 在同一个 tcp 的连接中可以传送多个 HTTP 请求和响应，HTTP 1.1 的持续连接，也需要增加新的请求头来帮助实现，例如，Connection 请求头的值为 Keep-Alive 时，客户端通知服务器返回本次请求结果后保持连接；Connection 请求头的值为 close 时，客户端通知服务器返回本次请求结果后关闭连接。
 - 多个请求和响应可以重叠，多个请求和响应可以同时进行
 - 更加多的请求头和响应头(比如 HTTP1.0 没有 host 的字段)，HTTP 1.1 还提供了与身份认证、状态管理和 Cache 缓存等机制相关的请求头和响应头。
 - HTTP2.0：
 - HTTP/2 采用二进制格式而非文本格式
 - HTTP/2 是完全多路复用的，而非有序并阻塞的——只需一个连接即可实现并行
 - 使用报头压缩，HTTP/2 降低了开销
 - HTTP/2 让服务器可以将响应主动“推送”到客户端缓存中
-
- TCP 如何保证可靠传输
 - 数据包校验
 - 超时重传机制
 - 应答机制

- 对失序数据包重排序;
 - TCP 使用滑动窗口机制提供流量控制和拥塞控制
- 会话跟踪的方式：
 - Cookie
 - Session
 - 隐藏表单域
 - 重写 URL
 - Cookie 和 session

Cookie : Cookies 是服务器在本地机器上存储的小段文本并随每一个请求发送至同一个服务器。

Session : session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。Session 代表服务器与浏览器的一次会话过程，这个过程是连续的，也可以时断时续的。

区别：

- Session 在服务器端，cookie 在客户端（浏览器里）；
 - Session 默认被存放在一个服务器的文件里（不是内存）；
 - Session 的运行依赖 session id，而 session id 是存在 cookie 中的，也就是说，如果浏览器禁用了 cookie，session 也会失效，不过可以通过其他方式实现（如在 url 中传递 session id）；
 - Session 可以放在文件、数据库或者内存中；
 - 用户验证一般使用 session；
 - 维持一个会话的核心就是客户端的位移标识，即 session id；
 - 单个 cookie 保存的数据不能超过 4K，很多浏览器限制一个站点最多保存 20 个 cookie。
- GET 和 POST 的区别
 - GET 被强制服务器支持；
 - 浏览器对 URL 长度有限制，所以 GET 请求不能替代 POST 请求发送大数据；
 - GET 请求发送数据更小；
 - GET 请求是不安全的；

- GET 请求是幂等的；
- POST 请求不能被缓存；
- POST 请求相对 GET 请求是安全的。

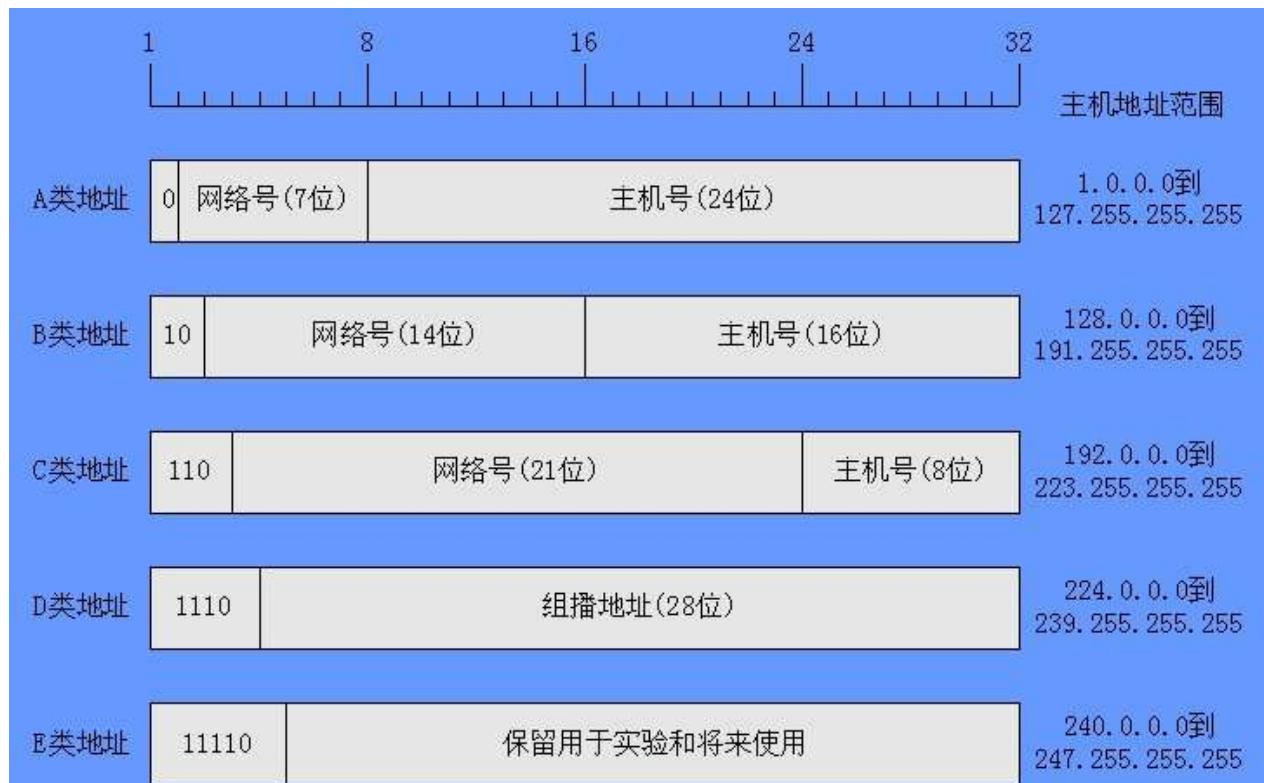
在以下情况，使用 POST 请求：

- 无法使用缓存文件（更新服务器上的文件或数据库）；
- 向服务器发送大量数据（POST 没有数据量限制）；
- 发送包含未知字符的用户输入时，POST 比 GET 更稳定和可靠；
- POST 比 GET 安全性更高。
- TCP/IP 四层模型
 - 应用层
 - 运输层
 - 网络层
 - 网络接口层
- OSI 七层模型
 - **物理层**，通过媒介传输比特，确定机械及电气规范（位 bit）中继器，集线器，双绞线
 - **数据链路层**，将比特组装成帧和点到点的传递（帧 Frame）差错控制 网桥，以太网交换机，网卡（一半物理层，一半数据链路层）
 - **网络层**，负责数据包从源到宿的传递和网际互连（包 PackeT）路由器，三层交换机 网络互连、路由选择、拥塞控制，通过寻址建立节点间连接
 - **传输层**，提供端到端的可靠报文传递和错误恢复（段 Segment）流量控制常规数据传递，为会话层用户提供端到端的可靠、透明和优化的数据传输服务机制。包括全双工或半双工、流量和错误恢复服务；传输层把消息分成若干分组，并在接收端对它们进行充足。
传输层（Transport Layer）是 OSI 中最重要，最关键的一层，是唯一负责总体的数据传输和数据控制的一层。传输层提供端到端的交换数据的机制。传输层对会话层等高三层提供可靠的传输服务，对网络层提供可靠的目的地站点信息。
通常不同应用程序要发送和接收的数据在传输层不同端口发出或者收入，也就是数据分段。

- **会话层**，建立、管理和终止会话（会话协议数据单元 SPDU）在两个结点之间建立端连接。为端系统的应用程序之间提供对话控制机制。此服务包括建立连接是以全双工还是半双工的方式进行设置，尽管可以在第四层中处理双工方式；会话层管理登入和注销过程。
- **表示层**，对数据进行翻译、加密和压缩（表示协议数据单元 PPDU）主要处理两个通信系统中交换信息的表示方式。为上层用户解决用户信息的语法问题。包括数据格式交换、数据加密与解密、数据压缩与终端类型的交换。
- **应用层**，允许访问 OSI 环境的手段（应用协议数据单元 APDU）为特定类型的网络应用提供了访问 OSI 环境的手段。应用层确定进程之间通信的性质，以满足用户的需要。
- 七层模型的协议
 - 物理层： RJ45 、 CLOCK 、 IEEE802.3 （中继器，集线器，网关） -
 - 数据链路： PPP 、 FR 、 HDLC 、 VLAN 、 MAC （网桥，交换机） -
 - 网络层： IP 、 ICMP 、 ARP 、 RARP 、 OSPF 、 IPX 、 RIP （ UDP ）、 IGRP 、（路由器） -
 - 传输层： TCP 、 UDP 、 SPX -
 - 会话层： NFS 、 SQL 、 NETBIOS 、 RPC 、 ASP
 - 表示层： JPEG 、 MPEG 、 ASCII 、 TIFF, GIF, JPEG, PIC
 - 应用层： FTP （ TCP, 21 和 20<被动> ）、 DNS (53) 、 Telnet （ TCP, 23 ）、 SMTP （ TCP, 25 ）、 HTTP （ TCP, 80 , 文本协议 ）、 POP3 (110) 、 SNMP (161) 、 DHCP (67)
- 网络拓扑结构
 - 计算机网络拓扑结构是指网络中各个站点相互连接的形式，各个站点抽象来说都是网络资源。
 - 总线型拓扑、环型拓扑、树型拓扑、星型拓扑、混合型拓扑以及网状拓扑
- 七层模型中的工作设备
 - 物理层：集线器（ hub ）， d 对接收到的信号进行再生整形放大，以扩大网络的传输距离，同时把所有节点集中在以它为中心的节点上，半双工模式

- 数据链路层：交换机，半双工或者全双工模式，交换机隔离冲突域
- 网络层：路由器，路由器隔离广播域
- 网络攻击
 - ARP 欺骗攻击：分为对路由器 ARP 表的欺骗和对内网 PC 的网关欺骗。第一种 ARP 欺骗的原理是——截获网关数据。第二种 ARP 欺骗的原理是——伪造网关。
 - 重放攻击：重放攻击 (Replay Attacks) 又称重播攻击、回放攻击或新鲜性攻击 (Freshness Attacks) 是指攻击者发送一个目的主机已接收过的包，来达到欺骗系统的目的，主要用于身份认证过程，破坏认证的正确性。
 - 暴力攻击：暴力破解攻击是指攻击者通过系统地组合所有可能性（例如登录时用到的账户名、密码），尝试所有的可能性破解用户的账户名、密码等敏感信息。攻击者会经常使用自动化脚本组合出正确的用户名和密码。
 - DNS 欺骗攻击：DNS 欺骗就是攻击者冒充域名服务器的一种欺骗行为。如果可以冒充域名服务器，然后把查询的 IP 地址设为攻击者的 IP 地址。
- 数据链路层
 - 逻辑链路层 LLC
 - 介质访问控制层 MAC
 - 目前广泛只是用的介质访问控制方法
 - 争用型介质访问控制，又称随机型的介质访问控制协议，如 CSMA/CD 方式。CSMA/CD 可以发现冲突，但是没有先知的冲突检测和阻止功能。
 - 确定型介质访问控制，又称有序的访问控制协议，如 Token(令牌) 方式
 - 后退 N 帧协议：收到序号为 N 的帧的确认，则表明 N 和 N 之前的帧都成功发送
 - A 类地址的第一位为 0；
B 类地址的前两位为 10；
C 类地址的前三位为 110；
D 类地址的前四位为 1110；
E 类地址的前五位为 11110。

如下如：



在 A 类地址中，10.0.0.0 到 10.255.255.255 是私有地址

在 B 类地址中，172.16.0.0 到 172.31.255.255 是私有地址。

在 C 类地址中，192.168.0.0 到 192.168.255.255 是私有地址。

- 浏览器和服务器在基于 https 进行请求链接到数据传输过程中
 - 非对称加密技术
 - 对称加密技术
 - 散列（哈希）算法
 - 数字证书
- select 和 epoll 这两个机制都是多路 I/O 机制的解决方案
<https://segmentfault.com/a/1190000003063859>
 - select 为 POSIX 标准的
 - 在 select 中采用轮询处理，其中的数据结构类似一个数组的数据结构
 - select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制
 - 单个进程可监视的 fd 数量受到了限制，在 32 位机器上，他所能管理的 fd 数量最大为 1024;

- 对 socket 进行扫描时是线性扫描，当 socket 文件描述符数量变多时，大量的时间是被白白浪费掉的。
- epoll 为 Linux 所特有的。
 - epoll 的最大好处是不会随着 FD 的数目增长而降低效率，而 epoll 是维护一个队列，直接看队列是不是空就可以了。
 - nginx 就是使用 epoll 来实现 I/O 复用支持高并发，目前在高并发的场景下，nginx 越来越受到欢迎。
 - IO 的效率不会随着监视 fd 的数量的增长而下降。epoll 不同于 select 和 poll 轮询的方式，而是通过每个 fd 定义的回调函数来实现的。只有就绪的 fd 才会执行回调函数；
 - 支持电平触发和边沿触发（只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发）两种方式，理论上边缘触发的性能要更高一些，但是代码实现相当复杂。
 - 有着良好的就绪事件通知机制

网络面试题

2016 年 7 月 25 日

14:35

1. OSI , TCP/IP , 五层协议的体系结构，以及各层协议

答:OSI 分层（7 层）：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

TCP/IP 分层（4 层）：网络接口层、网际层、运输层、应用层。

五层协议（5 层）：物理层、数据链路层、网络层、运输层、应用层。

每一层的协议如下：

物理层：RJ45、CLOCK、IEEE802.3 （中继器，集线器）

数据链路：PPP、FR、HDLC、VLAN、MAC （网桥，交换机）

网络层：IP、ICMP、ARP、RARP、OSPF、IPX、RIP、IGRP、（路由器）

传输层：TCP、UDP、SPX

会话层：NFS、SQL、NETBIOS、RPC

表示层：JPEG、MPEG、ASII

应用层：FTP、DNS、Telnet、SMTP、HTTP、WWW、NFS

每一层的作用如下：

物理层：通过媒介传输比特,确定机械及电气规范（比特 Bit）

数据链路层：将比特组装成帧和点到点的传递（帧 Frame）

网络层：负责数据包从源到宿的传递和网际互连（包 PackeT）

传输层：提供端到端的可靠报文传递和错误恢复（段 Segment）

会话层：建立、管理和终止会话（会话协议数据单元 SPDU）

表示层：对数据进行翻译、加密和压缩（表示协议数据单元 PPDU）

应用层：允许访问 OSI 环境的手段（应用协议数据单元 APDU）

2. IP 地址的分类

答:A 类地址：以 0 开头， 第一个字节范围：0~126 (1.0.0.0 - 126.255.255.255)；

B 类地址：以 10 开头， 第一个字节范围：128~191 (128.0.0.0 - 191.255.255.255)；

C 类地址：以 110 开头， 第一个字节范围：192~223 (192.0.0.0 - 223.255.255.255)；

10.0.0.0—10.255.255.255 , 172.16.0.0—172.31.255.255 , 192.168.0.0—192.168.255.255。 (Internet 上保留地址用于内部)

IP 地址与子网掩码相与得到网络号

3. ARP 是地址解析协议，简单语言解释一下工作原理。

答:1：首先，每个主机都会在自己的 ARP 缓冲区中建立一个 ARP 列表，以表示 IP 地址和 MAC 地址之间的对应关系。

2：当源主机要发送数据时，首先检查 ARP 列表中是否有对应 IP 地址的目的主机的 MAC 地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送 ARP 数据包，该数据包包括的内容有：源主机 IP 地址，源主机 MAC 地址，目的主机的 IP 地址。

3：当本网络的所有主机收到该 ARP 数据包时，首先检查数据包中的 IP 地址是否是自己的 IP 地址，如果不是，则忽略该数据包，如果是，则首先从数据包中取出源主机的 IP 和 MAC 地址写入到 ARP 列表中，如果已经存在，则覆盖，然后将自己的 MAC 地址写入 ARP 响应包中，告诉源主机自己是它想要找的 MAC 地址。

4：源主机收到 ARP 响应包后。将目的主机的 IP 和 MAC 地址写入 ARP 列表，并利用此信息发送数据。如果源主机一直没有收到 ARP 响应数据包，表示 ARP 查询失败。

广播发送 ARP 请求，单播发送 ARP 响应。

4. 各种协议的介绍

答:ICMP 协议：因特网控制报文协议。它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。

TFTP 协议：是 TCP/IP 协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。

HTTP 协议：超文本传输协议，是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。

DHCP 协议：动态主机配置协议，是一种让系统得以连接到网络上，并获取所需要的配置参数手段。

NAT 协议：网络地址转换属接入广域网(WAN)技术，是一种将私有(保留)地址转化为合法 IP 地址的转换技术，

DHCP 协议：一个局域网的网络协议，使用 UDP 协议工作，用途：给内部网络或网络服务供应商自动分配 IP 地址，给用户或者内部网络管理员作为对所有计算机作中央管理的手段。

5. 描述 RARP 协议

答:RARP 是逆地址解析协议，作用是完成硬件地址到 IP 地址的映射，主要用于无盘工作站，因为给无盘工作站配置的 IP 地址不能保存。工作流程：在网络中配置一台 RARP 服务器，里面保存着 IP 地址和 MAC 地址的映射关系，当无盘工作站启动后，就封装一个 RARP 数据包，里面有其 MAC 地址，然后广播到网络上去，当服务器收到请求包后，就查找对应的 MAC 地址的 IP 地址装入响应报文中发回给请求者。因为需要广播请求报文，因此 RARP 只能用于具有广播能力的网络。

6. TCP 三次握手和四次挥手的全过程

答:三次握手 :

第一次握手 : 客户端发送 syn 包 ($syn=x$) 到服务器 , 并进入 SYN_SEND 状态 , 等待服务器确认 ;

第二次握手 : 服务器收到 syn 包 , 必须确认客户的 SYN ($ack=x+1$) , 同时自己也发送一个 SYN 包 ($syn=y$) , 即 SYN+ACK 包 , 此时服务器进入 SYN_RECV 状态 ;

第三次握手 : 客户端收到服务器的 SYN + ACK 包 , 向服务器发送确认包 ACK($ack=y+1$) , 此包发送完毕 , 客户端和服务器进入 ESTABLISHED 状态 , 完成三次握手。

握手过程中传送的包里不包含数据 , 三次握手完毕后 , 客户端与服务器才正式开始传送数据。理想状态下 , TCP 连接一旦建立 , 在通信双方中的任何一方主动关闭连接之前 , TCP 连接都将被一直保持下去。

四次挥手

与建立连接的 “三次握手” 类似 , 断开一个 TCP 连接则需要 “四次挥手” 。

第一次挥手 : 主动关闭方发送一个 FIN , 用来关闭主动方到被动关闭方的数据传送 , 也就是主动关闭方告诉被动关闭方 : 我已经不会再给你发数据了(当然 , 在 fin 包之前发送出去的数据 , 如果没有收到对应的 ack 确认报文 , 主动关闭方依然会重发这些数据) , 但是 , 此时主动关闭方还可以接受数据。

第二次挥手 : 被动关闭方收到 FIN 包后 , 发送一个 ACK 给对方 , 确认序号为收到序号 +1 (与 SYN 相同 , 一个 FIN 占用一个序号) 。

第三次挥手 : 被动关闭方发送一个 FIN , 用来关闭被动关闭方到主动关闭方的数据传送 , 也就是告诉主动关闭方 , 我的数据也发送完了 , 不会再给你发数据了。

第四次挥手 : 主动关闭方收到 FIN 后 , 发送一个 ACK 给被动关闭方 , 确认序号为收到序号 +1 , 至此 , 完成四次挥手。

7. 在浏览器中输入 www.baidu.com 后执行的全部过程

答:1、客户端浏览器通过 DNS 解析到 www.baidu.com 的 IP 地址

220.181.27.48 , 通过这个 IP 地址找到客户端到服务器的路径。客户端浏览器发起一个 HTTP 会话到 220.161.27.48 , 然后通过 TCP 进行封装数据包 , 输入到网络层。

2、在客户端的传输层，把 HTTP 会话请求分成报文段，添加源和目的端口，如服务器使用 80 端口监听客户端的请求，客户端由系统随机选择一个端口如 5000，与服务器进行交换，服务器把相应的请求返回给客户端的 5000 端口。然后使用 IP 层的 IP 地址查找目的端。

3、客户端的网络层不用关心应用层或者传输层的东西，主要做的是通过查找路由表确定如何到达服务器，期间可能经过多个路由器，这些都是由路由器来完成的工作，我不作过多的描述，无非就是通过查找路由表决定通过那个路径到达服务器。

4、客户端的链路层，包通过链路层发送到路由器，通过邻居协议查找给定 IP 地址的 MAC 地址，然后发送 ARP 请求查找目的地址，如果得到回应后就可以使用 ARP 的请求应答交换的 IP 数据包现在就可以传输了，然后发送 IP 数据包到达服务器的地址。

8. TCP 和 UDP 的区别？

答：TCP 提供面向连接的、可靠的数据流传输，而 UDP 提供的是非面向连接的、不可靠的数据流传输。

TCP 传输单位称为 TCP 报文段，UDP 传输单位称为用户数据报。

TCP 注重数据安全性，UDP 数据传输快，因为不需要连接等待，少了许多操作，但是其安全性却一般。

TCP 对应的协议和 UDP 对应的协议

TCP 对应的协议：

- (1) FTP：定义了文件传输协议，使用 21 端口。
- (2) Telnet：一种用于远程登陆的端口，使用 23 端口，用户可以以自己的身份远程连接到计算机上，可提供基于 DOS 模式下的通信服务。
- (3) SMTP：邮件传送协议，用于发送邮件。服务器开放的是 25 号端口。
- (4) POP3：它是和 SMTP 对应，POP3 用于接收邮件。POP3 协议所用的是 110 端口。
- (5) HTTP：是从 Web 服务器传输超文本到本地浏览器的传送协议。

UDP 对应的协议：

- (1) DNS：用于域名解析服务，将域名地址转换为 IP 地址。DNS 用的是 53 号端口。

(2) SNMP：简单网络管理协议，使用 161 号端口，是用来管理网络设备的。由于网络设备很多，无连接的服务就体现出其优势。

(3) TFTP(Trivial File Transfer Protocol)，简单文件传输协议，该协议在熟知端口 69 上使用 UDP 服务。

9. DNS 域名系统，简单描述其工作原理。

答：当 DNS 客户机需要在程序中使用名称时，它会查询 DNS 服务器来解析该名称。客户机发送的每条查询信息包括三条信息：包括：指定的 DNS 域名，指定的查询类型，DNS 域名的指定类别。基于 UDP 服务，端口 53。该应用一般不直接为用户使用，而是为其他应用服务，如 HTTP，SMTP 等在其中需要完成主机名到 IP 地址的转换。

面向连接和非面向连接的服务的特点是什么？

面向连接的服务，通信双方在进行通信之前，要先在双方建立起一个完整的可以彼此沟通的通道，在通信过程中，整个连接的情况一直可以被实时地监控和管理。

非面向连接的服务，不需要预先建立一个联络两个通信节点的连接，需要通信的时候，发送节点就可以往网络上发送信息，让信息自主地在网络上去传，一般在传输的过程中不再加以监控。

10. TCP 的三次握手过程？为什么会采用三次握手，若采用二次握手可以吗？

答：建立连接的过程是利用客户服务器模式，假设主机 A 为客户端，主机 B 为服务器端。

(1) TCP 的三次握手过程：主机 A 向 B 发送连接请求；主机 B 对收到的主机 A 的报文段进行确认；主机 A 再次对主机 B 的确认进行确认。

(2) 采用三次握手是为了防止失效的连接请求报文段突然又传送到主机 B，因而产生错误。失效的连接请求报文段是指：主机 A 发出的连接请求没有收到主机 B 的确认，于是经过一段时间后，主机 A 又重新向主机 B 发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种特殊情况，主机 A 第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机 B，主机 B 以为是主机 A 又发起的新连接，于是主机 B 同意连接，并向主机 A 发回确认，但是此

时主机 A 根本不会理会，主机 B 就一直在等待主机 A 发送数据，导致主机 B 的资源浪费。

(3) 采用两次握手不行，原因就是上面说的实效的连接请求的特殊情况。

11. 了解交换机、路由器、网关的概念，并知道各自的用途

答:1) 交换机

在计算机网络系统中，交换机是针对共享工作模式的弱点而推出的。交换机拥有一条高带宽的背部总线和内部交换矩阵。交换机的所有的端口都挂接在这条背部总线上，当控制电路收到数据包以后，处理端口会查找内存中的地址对照表以确定目的 MAC (网卡的硬件地址) 的 NIC (网卡) 挂接在哪个端口上，通过内部 交换矩阵迅速将数据包传送到目的端口。目的 MAC 若不存在，交换机才广播到所有的端口，接收端口回应后交换机会 “ 学习 ” 新的地址，并把它添加入内部地址表 中。

交换机工作于 OSI 参考模型的第二层，即数据链路层。交换机内部的 CPU 会在每个端口成功连接时，通过 ARP 协议学习它的 MAC 地址，保存成一张 ARP 表。在今后的通讯中，发往该 MAC 地址的数据包将仅送往其对应的端口，而不是所有的端口。因此，交换机可用于划分数据链路层广播，即冲突域；但它不能划分网络层广播，即广播域。

交换机被广泛应用于二层网络交换，俗称 “ 二层交换机 ” 。

交换机的种类有：二层交换机、三层交换机、四层交换机、七层交换机分别工作在 OSI 七层模型中的第二层、第三层、第四层盒第七层，并因此而得名。

2) 路由器

路由器 (Router) 是一种计算机网络设备，提供了路由与转送两种重要机制，可以决定数据包从来源端到目的端所经过 的路由路径 (host 到 host 之间的传输路径)，这个过程称为路由；将路由器输入端的数据包移送至适当的路由器输出端(在路由器内部进行)，这称为转 送。路由工作在 OSI 模型的第三层——即网络层，例如网际协议。

路由器的一个作用是连通不同的网络，另一个作用是选择信息传送的线路。 路由器与交换器的差别，路由器是属于 OSI 第三层的产品，交换器是 OSI 第二层的产品(这里特指二层交换机)。

3) 网关

网关（ Gateway ），网关顾名思义就是连接两个网络的设备，区别于路由器（由于历史的原因，许多有关 TCP/IP 的文献曾经把网络层使用的路由器（ Router ）称为网关，在今天很多局域网采用都是路由来接入网络，因此现在通常指的网关就是路由器的 IP ），经常在家庭中或者小型企业网络中使用，用于连接局域网和 Internet。 网关也经常指把一种协议转成另一种协议的设备，比如语音网关。

在传统 TCP/IP 术语中，网络设备只分成两种，一种为网关（ gateway ），另一种为主机（ host ）。网关能在网络间传递数据包，但主机不能 转送数据包。在主机（又称终端系统， end system ）中，数据包需经过 TCP/IP 四层协议处理，但是在网关（又称中介系统， intermediate system ）只需要到达网际层（ Internet layer ），决定路径之后就可以转送。在当时，网关（ gateway ）与路由器（ router ）还没有区别。

在现代网络术语中，网关（ gateway ）与路由器（ router ）的定义不同。网关（ gateway ）能在不同协议间移动数据，而路由器（ router ）是在不同网络间移动数据，相当于传统所说的 IP 网关（ IP gateway ）。

网关是连接两个网络的设备，对于语音网关来说，他可以连接 PSTN 网络和以太网，这就相当于 VOIP ，把不同电话中的模拟信号通过网关而转换成数字信号，而且加入协议再去传输。在到了接收端的时候再通过网关还原成模拟的电话信号，最后才能在电话机上听到。

对于以太网中的网关只能转发三层以上数据包，这一点和路由是一样的。而不同的是网关中并没有路由表，他只能按照预先设定的不同网段来进行转发。网关最重要的一点就是端口映射，子网内用户在外网看来只是外网的 IP 地址对应着不同的端口，这样看来就会保护子网内的用户。

HTTP 和 HTTPS 的区别

星期五, 九月 16, 2016

12:23 上午

注意⚠️

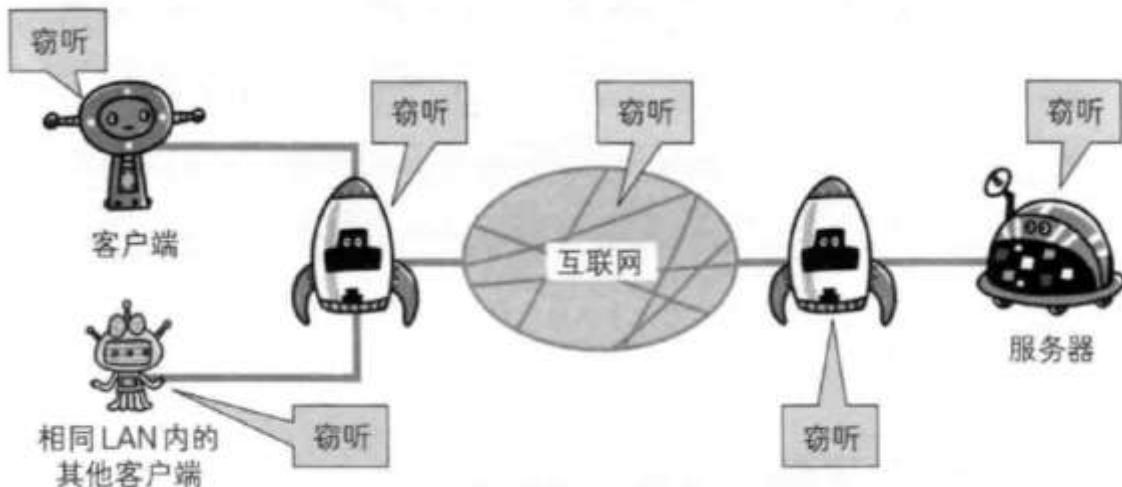
- https 协议需要到 ca 申请证书，一般免费证书很少，需要交费。
- http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议

- 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

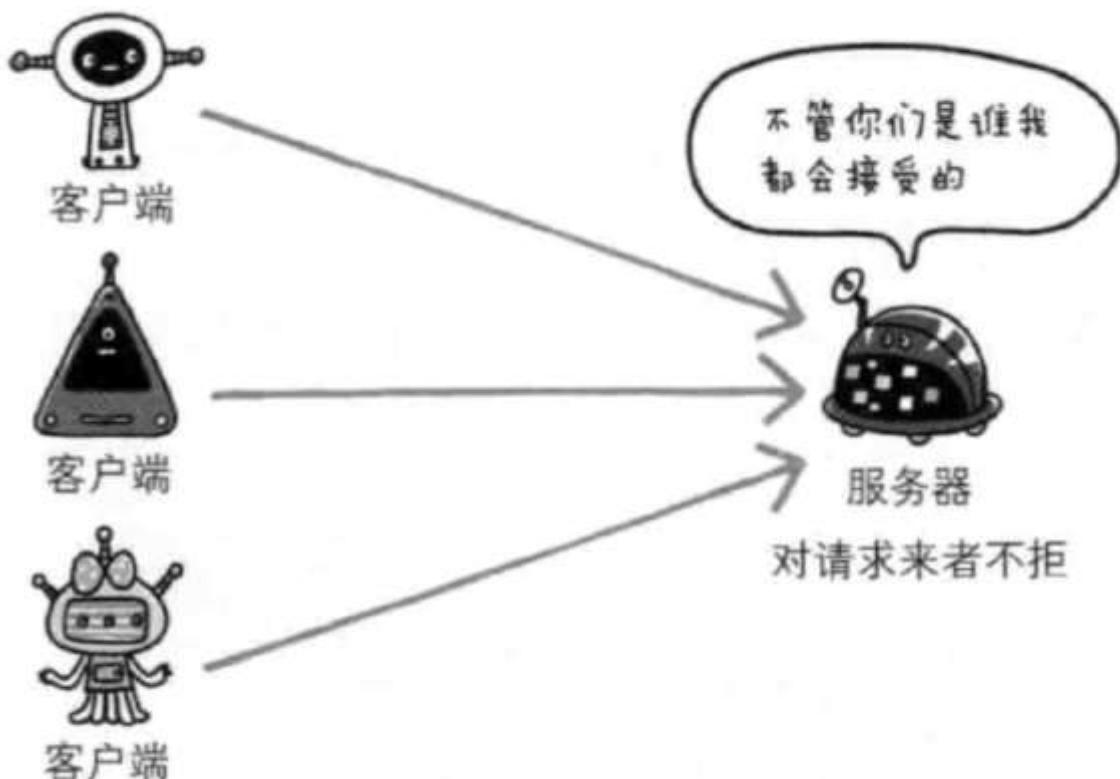
HTTP 的缺点

HTTP 主要有这些不足:

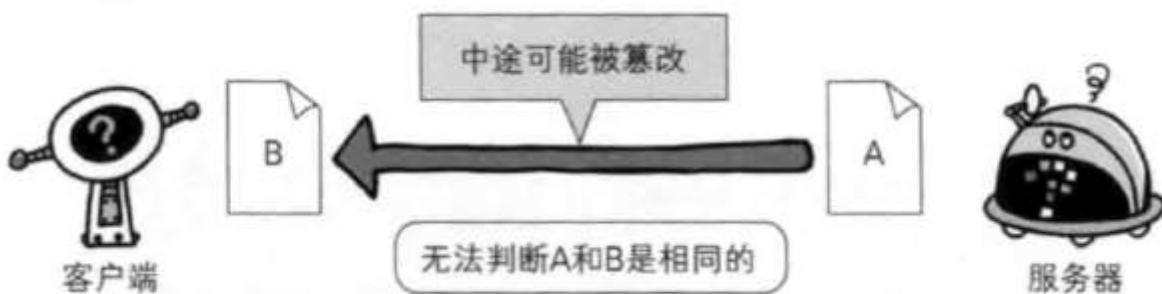
- 通信使用明文,内容可能被窃听



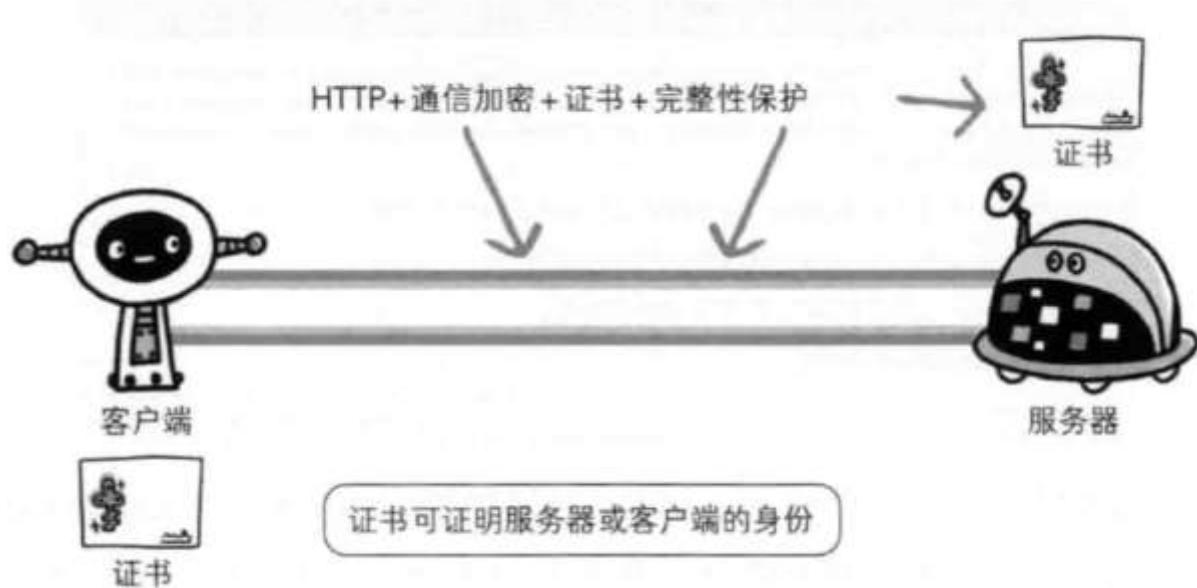
- 不验证通信方身份,因此有可能遭遇伪装



- 无法验证报文的完整性,所有有可能已篡改

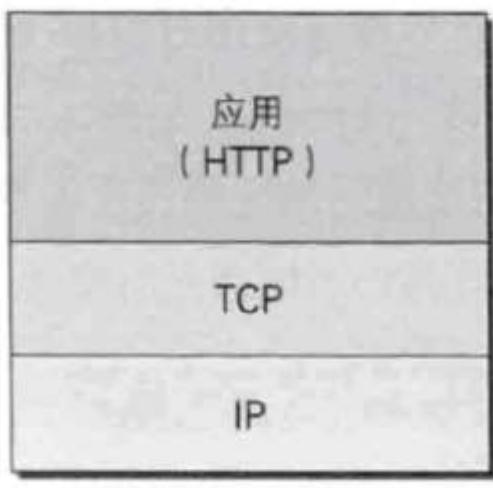


HTTP + 加密 + 认证 + 完整性保护 = HTTPS

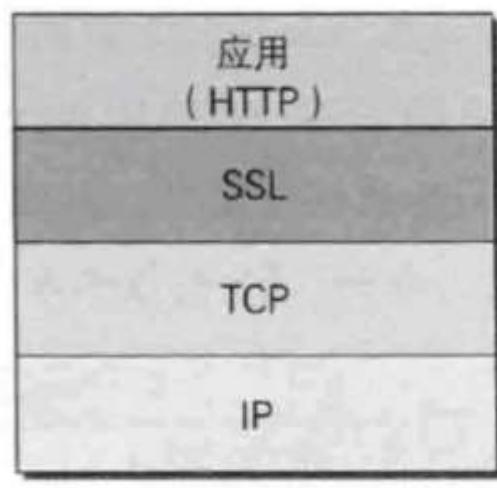


HTTPS 是身披 SSL 外壳的 HTTP

通常情况下 HTTP 是直接和 TCP 层进行通信的。当使用 SSL(安全套字层)时，则演变成 HTTP 先和 SSL 通信，SSL 再和 TCP 通信的了。



HTTP



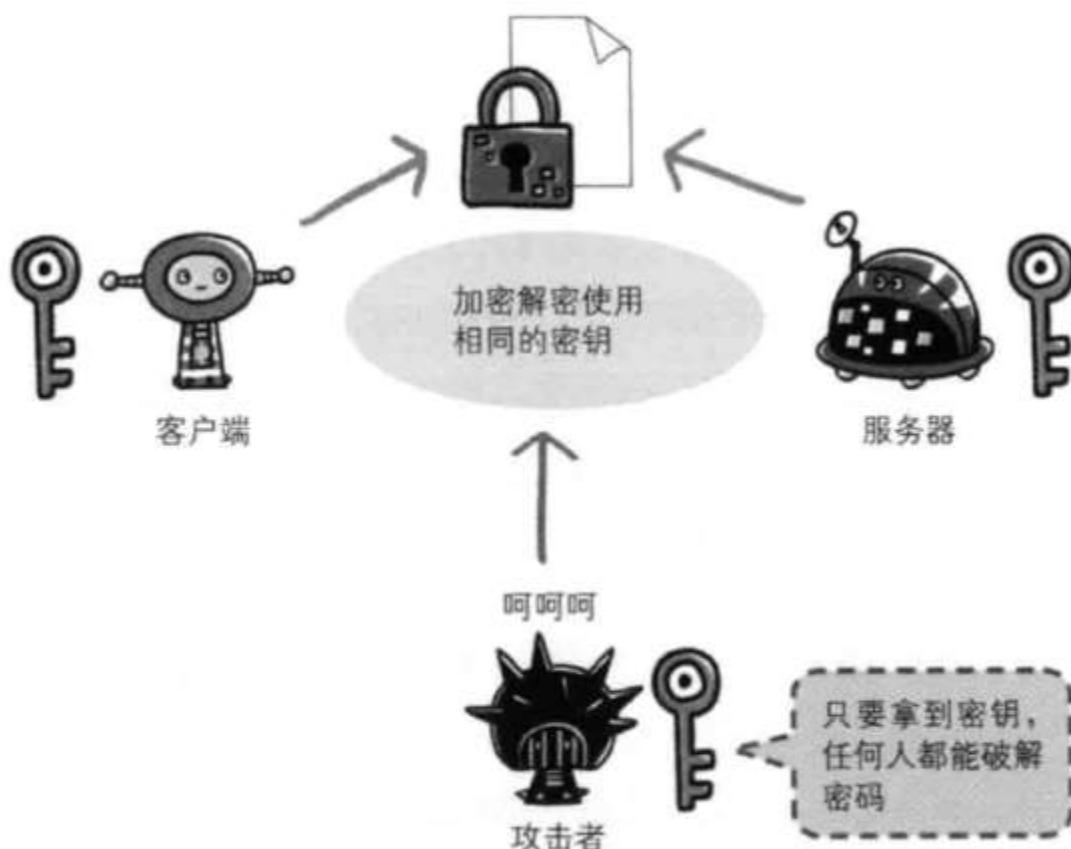
HTTPS

加密技术

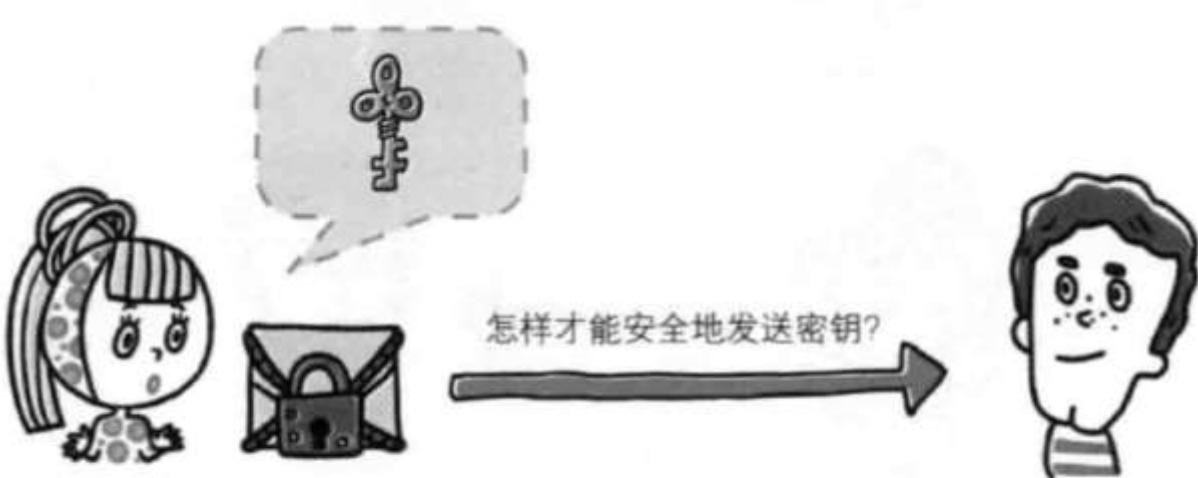
讲解 SSL 前,科普一下加密方法,SSL 采用的是一种叫做公开密钥加密的加密处理方式

对称加密

加密和解密用的一个密钥的方式称为对称加密,也叫做共享密钥加密



对称加密在发送加密信息时也需要将密钥发送给对方,但这样可以被攻击者截取,就不安全啦~



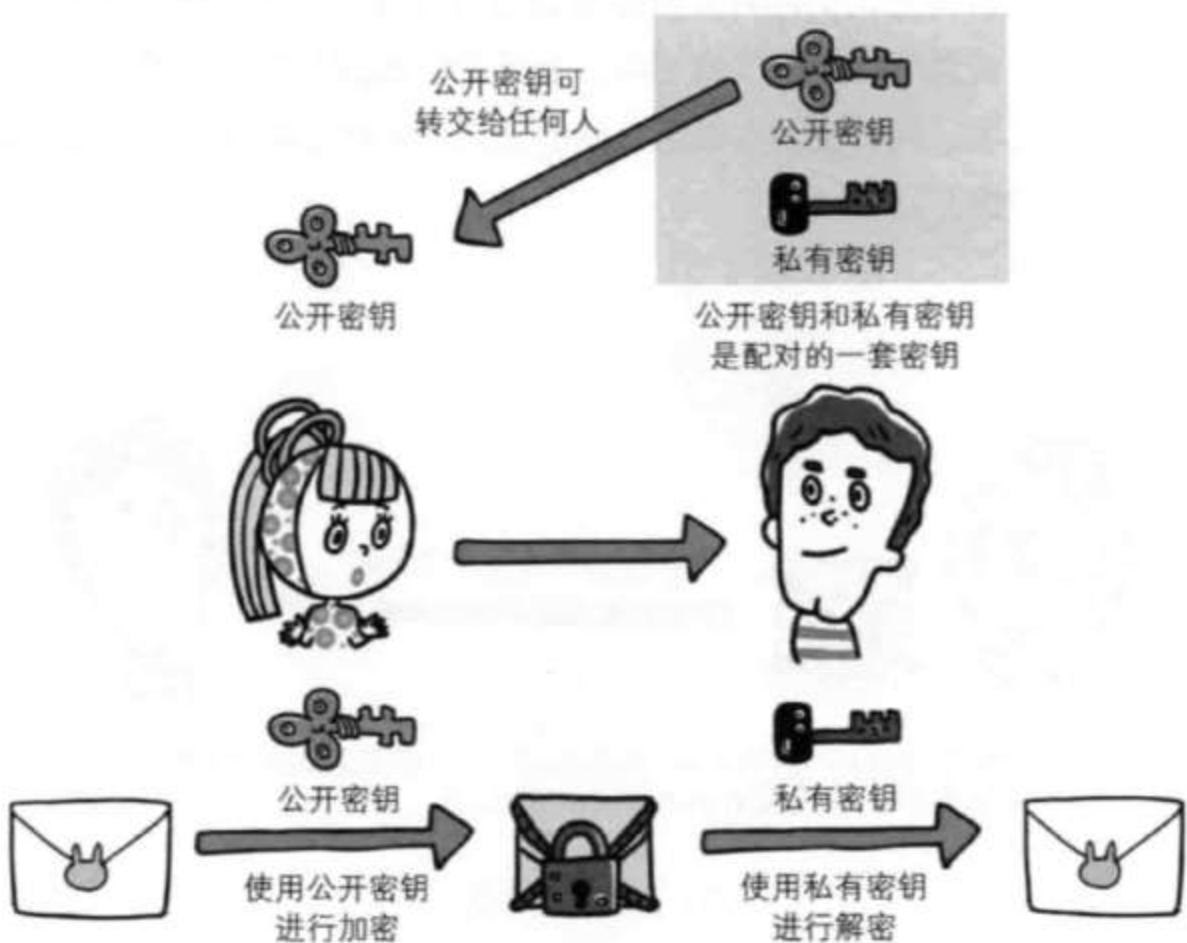
发送密钥就有被窃听的风险，但不发送，对方就不能解密。再说，密钥若能够安全发送，那数据也应该能安全送达。

非对称加密

非对称加密又称作公开密钥加密，它很好的解决了对称加密密钥被截取的问题。

非对称加密采用一对非对称的密钥，一把叫做私有密钥，一把叫做共有密钥。

使用非对称加密，发送密文一方使用对方的共有密钥进行加密处理，对方收到加密信息后，再使用自己的私有密钥进行解密。



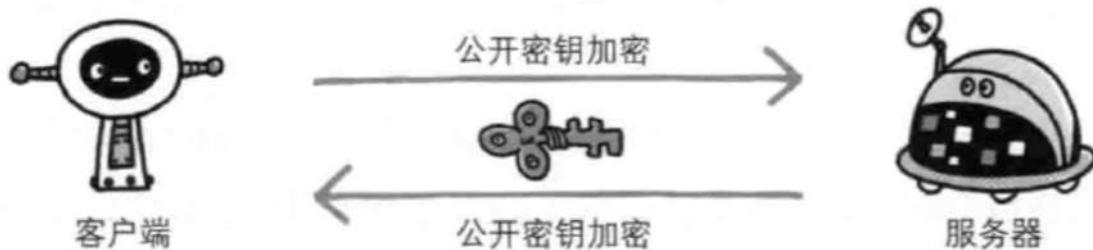
HTTPS 采用混合加密机制

HTTPS 采用对称加密和非对称加密所混合的加密机制。

若密钥能安全交换,那么有可能仅考虑对称加密。

但是非对称加密与对称加密相比,处理速度相对较慢。

① 使用公开密钥加密方式安全地交换在稍后的共享密钥加密中要使用的密钥



② 确保交换的密钥是安全的前提下，使用共享密钥加密方式进行通信



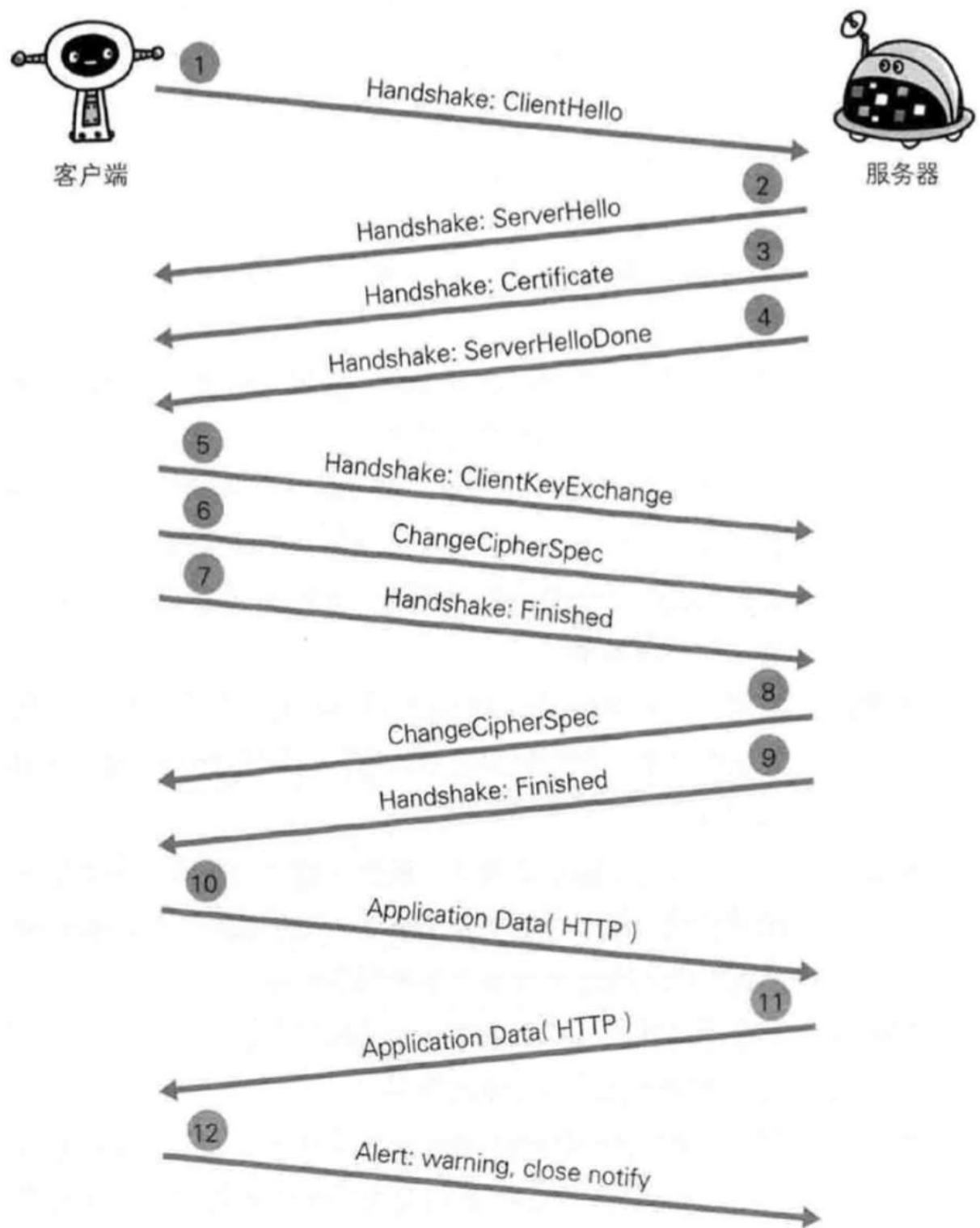
公开密钥的认证

使用数字证书认证机构和其颁布的公开密钥证书进行认证。即让第三方独立机构进行验证。

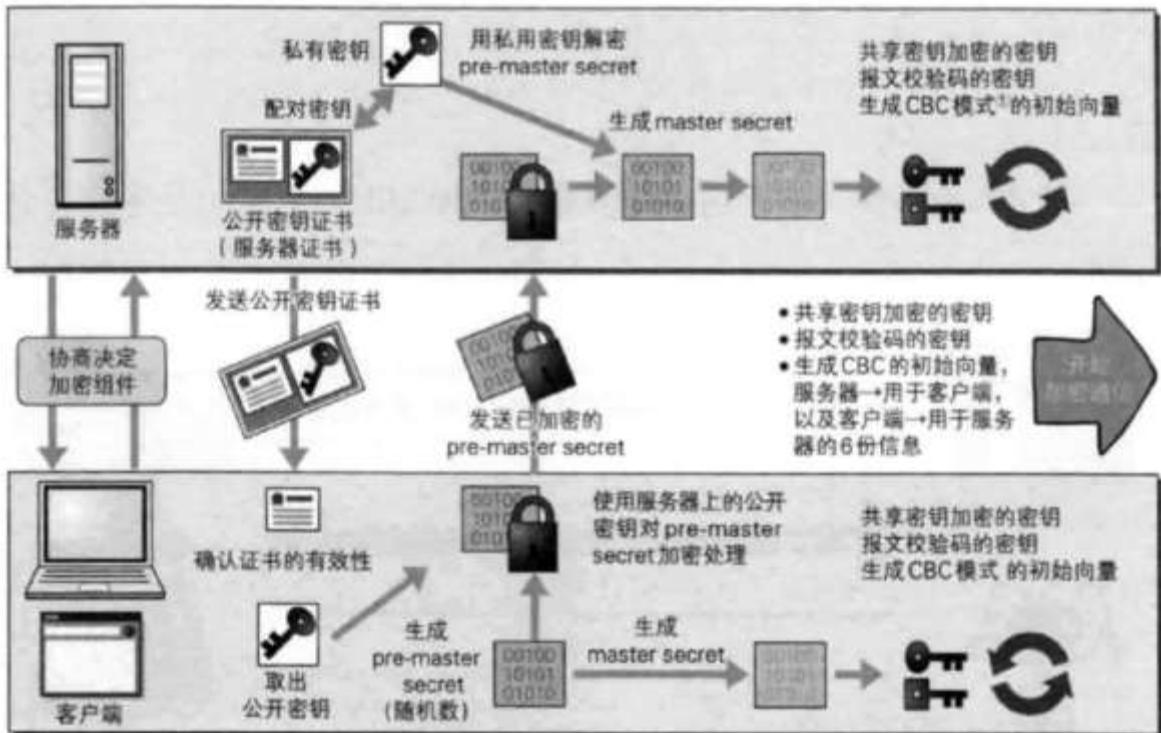
私有密钥是保存在服务器端的 ~

注意⚠️：认证是要钱的！！！

HTTPS 安全通信机制



下图是完整的 HTTPS 的通信过程



为什么 HTTPS 不是那么普及

1. 加密通信与纯文本通信相比,消耗更多的 CPU 和内存资源
2. 购买证书是要钱的!
3. 少许对客户端有要求的情况下,会要求客户端也必须有一个证书.
 - 这里客户端证书,其实就类似表示个人信息的时候,除了用户名/密码,还有一个 CA 认证过的身份. 应为个人证书一般来说上别人无法模拟的,所有这样能够更深的确认自己的身份
 - 目前少数个人银行的专业版是这种做法,具体证书可能是拿 U 盘作为一个备份的载体

HTTPS 一定是繁琐的

1. 本来简单的 http 协议,一个 get 一个 response. 由于 https 要还密钥和确认加密算法的需要. 单握手就需要 6/7 个往返,任何应用中,过多的 round trip 肯定影响性能.
2. 接下来才是具体的 http 协议,每一次响应或者请求,都要求客户端和服务端对会话的内容做加密/解密,尽管对称加密/解密效率比较高,可是仍然要消耗过多的 CPU,为此有专门的 SSL 芯片. 如果 CPU 信能比较低的话,肯定会降低性

能,从而不能 serve 更多的请求,加密后数据量的影响. 所以,才会出现那么多的安全认证提示

参考文档

1. 《图解 HTTP》,【日】上野宣

数据库

2016 年 3 月 28 日

12:59

1. 数据库事务

事务的特性 (ACID) : 原子性 , 一致性 , 隔离性 , 持久性

事务隔离 : 未提交读 , 已提交读 (SqlServer 默认级别) , 可重复读
(MySql 默认级别) , 串行读

事务隔离的实现 : 共享锁 , 更新锁 , 独占锁

隔离级别	脏读(Dirty Read)	不可重复读 (NonRepeatable Read)	幻读(Phantom Read)
未提交读	可能	可能	可能
已提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

2. 脏读、不可重复读和幻读

- 脏读 : 一个事务读取到另一个事务没有提交的数据。
 - 事务 T1 更新了一行记录内容 , 但没有提交修改 ;
 - 事务 T2 读取更新后的行 ;
 - 然后 T1 执行回滚 , 取消了修改 ;
 - T2 读取的就是脏数据。

- 不可重复读：在同一个事务中，两次读取同一数据，得到的内容不同。
 - T1 读取一行记录；
 - T2 修改了 T1 读取的记录；
 - T1 再次读取该行记录，发现读取的结果不同。
- 幻读：同一事务中，用同样的操作读取两次，得到的记录数不同。
 - T1 读取一条指定的 where 子句返回的结果集；
 - T2 新插入一行记录，该记录正好满足 T1 查询要求；
 - T1 再次对表进行检索，看到了 T2 插入的数据。

3. 事务与存储过程区别

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元(unit)。事务通常由高级数据库操纵语言或编程语言(如 SQL, C++ 或 Java)书写的用户程序的执行所引起，并用形如 begin transaction 和 end transaction 语句(或函数调用)来界定。事务由事务开始(begin transaction)和事务结束(end transaction)之间执行的全体操作组成。

存储过程是在大型数据库系统中，一组为了完成特定功能的 SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数(如果该存储过程带有参数)来执行它。存储过程是数据库中的一个重要对象，任何一个设计良好的数据库应用程序都应该用到存储过程。

特点：

1. 存储过程因为 SQL 语句已经编译过了，运行速度很快；
2. 可以保证数据的安全性和完整性：通过存储过程可以使没有权限的用户在控制之下间接的存取数据库，从而保证数据的完全；通过存储过程可以使相关的动作在一起发生，从而可以维护数据库的完整性。
3. 可以减低网络的通信量，存储过程主要是在服务器上运行，减少对客户机的压力；
4. 存储过程可以接受参数，输出参数，返回单个或者多个结果集以及返回值，可以返回错误原因；

5. 存储过程可以包含程序流，逻辑以及对数据库的查询，同时可以实现封装和隐藏数据逻辑。

很明显，他们的区别是：事务是保存在项目里的，存储过程是保存在数据库里的。

4. 数据库中的范式（限制逐渐递增）

第一范式（1NF）：每个属性不可再分，数据库表中的字段都是单一属性的，不可再分。这个单一属性由基本类型构成，包括整型、实数、字符型、逻辑型、日期型等。

第二范式（2NF）：属性完全依赖于主键，消除部分子函数依赖。数据库表中不存在非关键字段对任一候选关键字段的部分函数依赖（部分函数依赖指的是存在组合关键字中的某些字段决定非关键字段的情况），也即所有非关键字段都完全依赖于任意一组候选关键字。

第三范式（3NF）：属性不依赖于其他非主属性，消除传递依赖。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。所谓传递函数依赖，指的是如果存在 " $A \rightarrow B \rightarrow C$ " 的决定关系，则 C 传递函数依赖于 A 。因此，满足第三范式的数据库表应该不存在如下依赖关系：关键字段 \rightarrow 非关键字段 $x \rightarrow$ 非关键字段 y

巴斯-科德范式（BCNF）：在 1NF 的基础上，任何非主属性不能对主键子集依赖，在 3NF 的基础上消除对主键子集的依赖。

第四范式（4NF）：要求把同一表内的多对多关系删除。

第五范式（5NF）：从最终结构重新建立原始结构。

5. 关于 Sql 连接

- 内连接

使用比较运算符（包括=、>、<、<>、>=、<=、!>和!<）进行表间的比较操作，查询与连接条件相匹配的数据。

1. 等值连接

在连接操作中使用等号运算符，查询结果列出被连接表的所有列，包括重复列。

```
select * from T_student s,T_class c where s.classId = c.classId
```

2. 不等值连接

在连接条件中使用除等于号之外运算符（>、<、<>、>=、<=、!>和!<）

```
select * from T_student s inner join T_class c on s.classId <> c.classId
```

2. 外连接

• 左外连接

以左表为基准，返回左表中所有行，如果左表中的行在右表中没有匹配行，则结果中右表中的列返回空值。

```
select * from T_student s left join T_class c on s.classId = c.classId
```

2. 右外连接

与左连接相反，返回右表中的所有行，如果右表在左表没有匹配行，则结果中左表中的列返回空值。

```
select * from T_student s right join T_class c on s.classId = c.classId
```

iii. 全外连接

返回左表右表中的所有行，如果某行在另一表中没有匹配，则另一表的列返回空值。

```
select * from T_student s full join T_class c on s.classId = c.classId
```

6. Redis (NoSql 数据库)

- Redis 是一个速度非常快的非关系型数据库，可以存储键和 5 种不同类型的值之间的映射，可以将存储在内存中的键值对数据持久化到硬盘中。

2. 与 Memcached 相比

- i. 两者都可以用于存储键值映射，彼此性能相差无几；
- ii. Redis 能够自动以两种不同方式将数据写入硬盘；
- iii. Redis 除了能够存储普通的字符串键之外，还可以存储其他 4 种数据结构，memcache 只能存储字符串；
- iv. Redis 能作为主数据库，也能作为其他存储系统的辅助数据库。

7. MySql 有哪些存储引擎

特点	Myisam	BDB	Memory	InnoDB	Archive
存储限制	没有	没有	有	64TB	没有
事务安全		支持		支持	
锁机制	表锁	页锁	表锁	行锁	行锁
B 树索引	支持	支持	支持	支持	
哈希索引			支持	支持	
全文索引	支持				
集群索引				支持	
数据缓存			支持	支持	
索引缓存	支持		支持	支持	
数据可压缩	支持				支持
空间使用	低	低	N/A	高	非常低
内存使用	低	低	中等	高	低
批量插入的速度	高	高	高	低	非常高
支持外键				支持	

8. Myisam 和 InnoDB 的区别：

	MyISAM	InnoDB
构成上的区别：	<p>每个 MyISAM 在磁盘上存储成三个文件。第一个文件的名字以表的名字开始，扩展名指出文件类型。</p> <p>.frm 文件存储表定义。</p> <p>数据文件的扩展名为.MYD (MYData)。</p> <p>索引文件的扩展名是.MYI (MYIndex)。</p>	<p>基于磁盘的资源是 InnoDB 表空间数据文件和它的日志文件，InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB</p>
事务处理上方面：	<p>MyISAM 类型的表强调的是性能，其执行速度比 InnoDB 类型更快，但是不提供事务支持</p>	<p>InnoDB 提供事务支持，外部键 (foreign key) 等高级数据库功能</p>

SELECT UPDATE,INSERT , Delete 操作	如果执行大量的 SELECT , MyISAM 是更好的选择	<p>1.如果你的数据执行大量的 INSERT 或 UPDATE , 出于性能方面的考虑 , 应该使用 InnoDB 表</p> <p>2.DELETE FROM table 时 , InnoDB 不会重新建立表 , 而是一行一行的删除。</p> <p>3.LOAD TABLE FROM MASTER 操作对 InnoDB 是不起作用的 , 解决方法是首先把 InnoDB 表改成 MyISAM 表 , 导入数据后再改成 InnoDB 表 , 但是对于使用的额外的 InnoDB 特性 (例如外键) 的表不适用</p>
对 AUTO_INCREMENT 的操作	每表一个 AUTO_INCREMENT 列的内部处理。	如果你为一个表指定 AUTO_INCREMENT 列 , 在数据词典里的 InnoDB 表句柄

	<p>MyISAM 为 INSERT 和 UPDATE 操作自动更新这一列。这使得</p> <p>AUTO_INCREMENT 列更快（至少 10%）。在序列顶的值被删除之后就不能再利用。（当 AUTO_INCREMENT 列被定义为多列索引的最后一列，可以出现重使用从序列顶部删除的值的情况）。</p> <p>AUTO_INCREMENT 值可用 ALTER TABLE 或 myisamchk 来重置</p> <p>对于 AUTO_INCREMENT 类型的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中，可以和其他字段一起建立联合索引</p>	<p>包含一个名为自动增长计数器的计数器，它被用在为该列赋新值。</p> <p>自动增长计数器仅被存储在主内存中，而不是存在磁盘上</p> <p>关于该计算器的算法实现，请参考 AUTO_INCREMENT 列在 InnoDB 里如何工作</p>
--	---	--

	更好和更快的 auto_increment 处理	
表的具体行数	select count(*) from table,MyISAM 只 要简单的读出保存 好的行数，注意的 是，当 count(*)语 句包含 where 条 件时，两种表的操 作是一样的	InnoDB 中不保存 表的具体行数，也 就是说，执行 select count(*) from table 时， InnoDB 要扫描一遍 整个表来计算有多 少行
锁	表锁	提供行锁(locking on row level)，提 供与 Oracle 类型一 致的不加锁读取 (non-locking read in SELECTs)，另 外，InnoDB 表的行 锁也不是绝对的， 如果在执行一个 SQL 语句时 MySQL 不能确定要扫描的 范围，InnoDB 表同 样会锁全表，例如 update table set num=1 where

		name like "%aaa%"
--	--	----------------------

- InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了。
- MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用
- InnoDB 支持外键，MyISAM 不支持
- MyISAM 是默认引擎，InnoDB 需要指定
- InnoDB 不支持 FULLTEXT 类型的索引
- InnoDB 中不保存表的行数，如 select count(*) from table 时，InnoDB 需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count(*)语句包含 where 条件时 MyISAM 也需要扫描整个表
- 对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引
- 清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表
- InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'
- 数据库索引
 1. 索引的特点
 - i. 可以加快数据库的检索速度
 - ii. 降低数据库插入、修改、删除等维护速度
 - iii. 只能创建在表上，不能创建在视图上
 - iv. 既可以创建，也可以间接创建
 - v. 可以在优化隐藏中使用索引
 - vi. 使用查询处理器执行 SQL 语句，在一个表上，一次只能使用一个索引。
- 索引的优点
 1. 创建唯一性索引，保证数据库表中每一行数据的唯一性；

2. 大大加快数据的检索速度；
3. 加速数据库表之间的连接，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的时候，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，可以在查询中使用优化隐藏器，提高系统性能。

- 索引的缺点

1. 创建和维护索引需要耗费时间，并且耗费的时间和数据量成正比；
2. 创建索引需要占用物理空间，除了数据表占用数据空间外，每一个索引还要占用一定的物理空间，如果建立聚簇索引，那么需要的空间更大；
3. 当对表中的数据进行增删改的时候，索引需要维护，降低数据维护的速度。

- 索引分类

1. 直接创建索引和间接创建索引
 - i. 直接索引。
 - ii. 间接索引，创建主键约束和唯一性约束的时候，数据库会自动为该列添加索引，故该索引成为间接索引。
2. 普通索引和唯一性索引
 - i. 普通索引：列值可以重复。
 - ii. 唯一性索引：列值可以为空，但不可重复。
3. 单个索引和复合索引
 - i. 单个索引：作用的列只有一列。
 - ii. 复合索引：作用于多个列的索引。
4. 聚簇索引和非聚簇索引
 - i. 聚簇索引，不是一个单独的索引，而是存储方式，聚簇索引的叶子节点存的是数据，而普通索引叶子节点存的是指向数据的指针。

- ii. 区别：聚簇索引，当将一列设为主键的时候，Mysql 会自动将该列设置为聚簇索引。当添加新的数据的时候，新数据会在表中的位置按照索引的逻辑来存储，这样查找会比较方便。
- iii. MySQL 聚簇索引并不是一种单独的索引类型，而是一种数据存储方式，是指 innodb 引擎的特性。InnoDB 的聚簇索引实际上在同一个结构中保存了 B-Tree 索引和数据行。

如果需要该索引，只要将索引指定为主键（ primary key ）就可以了。

比如：

```
create table blog_user
(
    user_Name char(15) not null check(user_Name != ''),
    user_Password char(15) not null,
    user_email varchar(20) not null unique,
    primary key(user_Name)

)engine=innodb default charset=utf8 auto_increm
ent=1;
```

其中的 primary key(user_Name) 这个就是聚簇索引索引了；

- 索引失效

1. 如果条件中有 or , 即使其中有条件带索引也不会使用；
2. 对于多列索引，不是使用的第一部分，则不会使用索引；
3. Like 查询是以%开头；
4. 如果列类型是字符串，那一定要在条件中使用引号引起来，否则不会使用索引；
5. 如果 mysql 估计使用全表扫描比使用索引快，则不会使用索引。

- 各种引擎支持的索引

索引	MyISAM 引擎	InnoDB 引擎	Memory 引擎
B-Tree 索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-Tree 索引	支持	不支持	不支持
Full-text 索引	不支持	暂不支持	不支持

7. 创建索引

i. 普通索引：

```
CREATE INDEX indexName
```

```
ON tableName;
```

2. 单列索引

```
CREATE INDEX index_name
```

```
ON table_name (column_name);
```

iii. 唯一索引

```
CREATE UNION INDEX index_name
```

```
ON table_name (column_name);
```

iv. 多列索引

```
CREATE INDEX index_name
```

```
on table_name (column1, column2);
```

10. 数据库视图

1. 视图是什么？

视图是从一个或者多个表中导出的表。视图与表不同，视图是一个虚表，即视图对应的数据不进行实际的存储，数据库只存储视图的定义，在对视图的数据进行操作的时候，系统根据视图的定义去操作与视图相关联的基本表

2. 视图的作用和优点

1. 可以定制用户数据，聚焦特定的数据；

2. 可以简化数据操作；
 3. 基表中的数据就有了一定的安全性；
 4. 可以合并分立的数据，创建分区视图。
3. 视图的类型
1. 普通视图
 2. 索引视图
 3. 分割视图
11. 数据库游标
- 游标是一种从包括多条数据记录的结果集中每次提取一条记录的机制；
游标是系统为用户开设的一个数据缓冲区，存放 SQL 语句的执行结果；
11. 数据库触发器
- 触发器是特定事件出现的时候，自动执行的代码块，类似于存储过程，但是用户不能直接调用。
- 功能：
- 允许/限制对表的修改；
 - 自动生成派生列；
 - 强制数据一致性；
 - 提供审计和日志记录；
 - 防止无效的事务处理；
 - 启用复杂的业务逻辑。
- 数据库主键外键
1. 主键：数据表的唯一索引
 2. 外键：另一表的主键，外键可以有重复的，可以是空值

Sql 语句

2016 年 9 月 16 日

19:36

1. LIMIT，限制结果
2. ORDER BY
 - a. DESC

b. ASC

3. DISTINCT , 去重

a. SELECT DISTINCT * FROM table_name;

4. WHERE , 过滤条件

a. 操作符 : =, <>, !=, <, <=, >, >=, BETWEEN AND, IS NULL

b. 逻辑操作符 : AND, OR, NOT,

c. IN 操作符 , SELECT * FROM table WHERE field IN (lower data, higher data);

d. 通配符 , 配合 LIKE 操作符使用

i. % , 表示任何字符出现任意次数 , 如找出 test 起头的
field : field LIKE "test%";

ii. _ , 匹配单个字符 , 如 SELECT * FROM table WHERE
field LIKE "_test";

e. 正则匹配 , REGEXP 操作符

5. 字段拼接 , Concat

6. 函数

a. 文本处理函数 :

i. Rtrim (field) , 除去列值右边的空格

ii. Upper (field) , 文本转换成大写

iii. 等等

b. 时间和日期处理函数 , 很多

c. 数值处理函数 , 绝对值 , 正余弦等

d. 聚集函数 :

i. AVG(field) , 返回某列平均值

ii. COUNT(field) , 返回某列的函数

iii. MAX(field)

iv. MIN(field)

v. SUM(field)

7. 数据分组 , 使用 GROUP BY 子句 , GROUP BY field

8. 联接

a. table 1 INNER JOIN table2 ON 条件

b. table1 LEFT OUT JOIN table2 ON 条件

c. table1 RIGHT OUT JOIN table2 ON 条件

9. 组合 , UNION 操作符 , 类似求并集操作

a. SQL 语句 1 UNION SQL 语句 2 , 默认去重

b. SQL 语句 1 UNION ALL SQL 语句 2 , 不去重

10. 插入 , INSERT INTO

a. INSERT INTO 表名 VALUES (值 1 , 值 2 , … , 值 N);

b. INSERT INTO 表名 (字段 1 , 字段 2 , …… , 字段 N) VALUES (值 1 , 值 2 , …… , 值 N);

11. 更新数据 , UPDATE , DELETE

a. UPDATE 表名 SET 字段名=字段值 WHERE 条件 ;

b. DELETE FROM 表名 WHERE 条件 ;

12. 表操作

a. 建表 , CREATE TABLE

CREATE TABLE 表名

(

字段名 类型 NULL/NOT NULL(表示是否允许空值)

[DEFAULT 值 (设置默认值)] [AUTO_INCREMENT] ,

……

PRIMARY KEY (字段名) ,

FROEIGN KEY (字段名)

) ;

2. 更新表 , ALTER TABLE

3. 删除表 , DROP TABLE 表名;

MYSQL-索引

星期五, 九月 16, 2016

10:03 下午

概述

用来加快查询的技术很多 , 其中最重要的是索引。通常索引能够快速提高查询速度。如果不适用索引 , MYSQL 必须从第一条记录开始然后读完整个表直到找出相关的行。表越大 , 花费的时间越多。但也不全是这样。本文讨论索引是什么以及如何使用索引来改善性能 , 以及索引可能降低性能的情况。

索引的本质

MySQL 官方对索引的定义为：索引（Index）是帮助 MySQL 高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是数据结构。

数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是顺序查找（linear search），这种复杂度为 $O(n)$ 的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如二分查找（binary search）、二叉树查找（binary tree search）等。如果稍微分析一下会发现，每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，但是数据本身的组织结构不可能完全满足各种数据结构（例如，理论上不可能同时将两列都按顺序进行组织），所以，在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

索引的存储分类

索引是在 MySQL 的存储引擎层中实现的，而不是在服务层实现的。所以每种存储引擎的索引都不一定完全相同，也不是所有的存储引擎都支持所有的索引类型。MySQL 目前提供了一下 4 种索引。

- B-Tree 索引：最常见的索引类型，大部分引擎都支持 B 树索引。
- HASH 索引：只有 Memory 引擎支持，使用场景简单。
- R-Tree 索引(空间索引)：空间索引是 MyISAM 的一种特殊索引类型，主要用于地理空间数据类型。
- Full-text (全文索引)：全文索引也是 MyISAM 的一种特殊索引类型，主要用于全文索引，InnoDB 从 MySQL5.6 版本提供对全文索引的支持。

Mysql 目前不支持函数索引，但是能对列的前面某一部分进行索引，例如标题 title 字段，可以只取 title 的前 10 个字符进行索引，这个特性可以大大缩小索引文件的大小，但前缀索引也有缺点，在排序 Order By 和分组 Group By 操作的时候无法使用。用户在设计表结构的时候也可以对文本列根据此特性进行灵活设计。

语法：create index idx_title on film (title(10))

MyISAM、InnoDB 引擎、Memory 三个常用引擎类型比较

索引	MyISAM 引擎	InnoDB 引擎	Memory 引擎
B-Tree 索引	支持	支持	支持
HASH 索引	不支持	不支持	支持
R-Tree 索引	支持	不支持	不支持
Full-text 索引	不支持	暂不支持	不支持

B-TREE 索引类型

- **普通索引**

这是最基本的索引类型，而且它没有唯一性之类的限制。普通索引可以通过以下几种方式创建：

- (1) 创建索引: **CREATE INDEX 索引名 ON 表名(列名 1 , 列名 2,...);**
- (2) 修改表: **ALTER TABLE 表名 ADD INDEX 索引名 (列名 1 , 列名 2,...);**
- (3) 创建表时指定索引 : **CREATE TABLE 表名 ([...], INDEX 索引名 (列名 1 , 列名 2,...));**

- **UNIQUE 索引**

表示唯一的，不允许重复的索引，如果该字段信息保证不会重复例如身份证号用作索引时，可设置为 unique：

- (1) 创建索引 : **CREATE UNIQUE INDEX 索引名 ON 表名(列的列表);**
- (2) 修改表 : **ALTER TABLE 表名 ADD UNIQUE 索引名 (列的列表);**
- (3) 创建表时指定索引 : **CREATE TABLE 表名([...], UNIQUE 索引名 (列的列表));**

- **主键 : PRIMARY KEY 索引**

主键是一种唯一性索引，但它必须指定为“PRIMARY KEY”。

- (1) 主键一般在创建表的时候指定：“**CREATE TABLE 表名([...], PRIMARY KEY (列的列表));**”。
- (2) 但是，我们也可以通过修改表的方式加入主键：“**ALTER TABLE 表名 ADD PRIMARY KEY (列的列表);**”。

每个表只能有一个主键。 (主键相当于聚合索引，是查找最快的索引)

注：不能用 CREATE INDEX 语句创建 PRIMARY KEY 索引

索引的设置语法

一 设置索引

在执行 CREATE TABLE 语句时可以创建索引，也可以单独用 CREATE INDEX 或 ALTER TABLE 来为表增加索引。

1. ALTER TABLE - ALTER TABLE 用来创建普通索引、**UNIQUE** 索引或 **PRIMARY KEY** 索引。

- ALTER TABLE table_name ADD **INDEX** index_name (column_list)
- ALTER TABLE table_name ADD **UNIQUE** (column_list)
- ALTER TABLE table_name ADD **PRIMARY KEY** (column_list)

2. CREATE INDEX - CREATE INDEX 可对表增加普通索引或 UNIQUE 索引。

- CREATE INDEX index_name ON table_name (column_list)
- CREATE UNIQUE INDEX index_name ON table_name (column_list)

二 删除索引

可利用 ALTER TABLE 或 DROP INDEX 语句来删除索引。类似于 CREATE INDEX 语句，DROP INDEX 可以在 ALTER TABLE 内部作为一条语句处理，语法如下。

- DROP INDEX index_name ON talbe_name
- ALTER TABLE table_name DROP INDEX index_name
- ALTER TABLE table_name DROP PRIMARY KEY

其中，前两条语句是等价的，删除掉 table_name 中的索引 index_name。

第 3 条语句只在删除 PRIMARY KEY 索引时使用，**因为一个表只能有一个 PRIMARY KEY 索引，因此不需要指定索引名**。如果没有创建 PRIMARY KEY 索引，但表具有一个或多个 UNIQUE 索引，则 MySQL 将删除第一个 UNIQUE 索引。

如果从表中删除了某列，则索引会受到影响。对于多列组合的索引，如果删除其中的某列，则该列也会从索引中删除。如果删除组成索引的所有列，则整个索引将被删除。

三 查看索引

```
mysql> show index from tblname;
mysql> show keys from tblname;
```

- **Table** : 表的名称
- **Non_unique** : 如果索引不能包括重复词，则为 0。如果可以，则为 1
- **Key_name** : 索引的名称
- **Seq_in_index** : 索引中的列序列号，从 1 开始
- **Column_name** : 列名称
- **Collation** : 列以什么方式存储在索引中。在 MySQL 中，有值'A' (升序) 或 NULL (无分类)。
- **Cardinality** : 索引中唯一值的数目的估计值。通过运行 ANALYZE TABLE 或 myisamchk -a 可以更新。基数根据被存储为整数的统计数据来计数，所以即使对于小型表，该值也没有必要是精确的。基数越大，当进行联合时，MySQL 使用该索引的机会就越大。
- **Sub_part** : 如果列只是被部分地编入索引，则为被编入索引的字符的数目。如果整列被编入索引，则为 NULL。
- **Packed** : 指示关键字如何被压缩。如果没有被压缩，则为 NULL。
- **Null** : 如果列含有 NULL，则含有 YES。如果没有，则该列含有 NO。
- **Index_type** : 用过的索引方法 (BTREE, FULLTEXT, HASH, RTREE)。
- **Comment** : 更多评注。

索引选择性

一 索引选择原则

1. 较频繁的作为查询条件的字段应该创建索引
2. 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件
3. 更新非常频繁的字段不适合创建索引

当然，并不是存在更新的字段就适合创建索引，从判定策略的用语上也可以看出，是“非常频繁”的字段。到底什么样的更新频率应该算是“非常频繁”呢？每秒？每分钟？还是每小时呢？说实话，还真难定义。很多时候是通过比较同一时间段内被更新的次数和利用该字段作为条件的查询次数来判断的，如果通过该字段的查询并不是很多，可能几个小时或是更长才会执行一次，更新反而比查询更频繁，那这样的字段肯定不适合创建索引。反之，如果我们通过该字段的查询比较频繁，但更新并不是特别多，比如查询几十次或更多才可能会产生一次更新，那我个人觉得更新所带来的附加成本也是可以接受的。

4. 不会出现在 **WHERE** 子句中的字段不该创建索引

二 索引选择原则细述

- 性能优化过程中，选择在哪个列上创建索引是最非常重要的。可以考虑使用索引的主要有 两种类型的列：**在 where 子句中出现的列，在 join 子句中出现的列**，而不是在 SELECT 关键字后选择列表的列；
- 索引列的基数越大，索引的效果越好。例如，存放出生日期的列具有不同的值，很容易区分行，而用来记录性别的列，只有“M”和“F”，则对此进行索引没有多大用处，因此不管搜索哪个值，都会得出大约一半的行，（见索引选择性注意事项对选择性解释；）
- 使用短索引，如果对字符串列进行索引，应该指定一个前缀长度，可节省大量索引空间，提升查询速度；
例如，有一个 CHAR(200)列，如果在前 10 个或 20 个字符内，多数值是唯一的，那么就不要对整个列进行索引。对前 10 个或者 20 个字符进行索引能够节省大量索引空间，也可能会使查询更快。较小的索引涉及的磁盘 IO 较少，较短的值比较起来更快。更为重要的是，对于较短的键值，所以高速缓存中的快能容纳更多的键值，因此，MySQL 也可以在内存中容纳更多的值。这样就增加了找到行而不用读取索引中较多快的可能性。
- 利用最左前缀

三 索引选择注意事项

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，MySQL 在运行时也要消耗资源维护索引，因此索引并不是越多越好。

一般两种情况下不建议建索引：

1. 表记录比较少，例如一两千条甚至只有几百条记录的表，没必要建索引，让查询做全表扫描就好了；
至于多少条记录才算多，这个个人有个人的看法，我个人的经验是以 2000 作为分界线，记录数不超过 2000 可以考虑不建索引，超过 2000 条可以酌情考虑索引。
2. 索引的选择性较低。所谓索引的选择性（Selectivity），是指不重复的索引值（也叫基数，Cardinality）与表记录数（#T）的比值：
$$\text{Index Selectivity} = \text{Cardinality} / \#T$$

显然选择性的取值范围为(0, 1]，选择性越高的索引价值越大，这是由 B+Tree 的性质决定的。例如，上文用到的 employees.titles 表，如果 title 字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

```
SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM employees.titles;
```

Selectivity
0.0000

title 的选择性不足 **0.0001**（精确值为 **0.00001579**），所以实在没有什么必要为其单独建索引。

3. MySQL 只对一下操作符才使用索引：`<`, `<=`, `=`, `>`, `>=`, `between`, `in`, 以及某些时候的 `like`(不以通配符%或_开头的情形)。
4. 不要过度索引，只保持所需的索引。每个额外的索引都要占用额外的磁盘空间，并降低写操作的性能。在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，索引越多，所花的时间越长。

四 索引的弊端

索引的益处已经清楚了，但是我们不能只看到这些益处，并认为索引是解决查询优化的圣经，只要发现 查询运行不够快就将 WHERE 子句中的条件全部放在索引中。

确实，索引能够极大地提高数据检索效率，也能够改善排序分组操作的性能，但有不能忽略的一个问题就是索引是完全独立于基础数据之外的一部分数据。假设在 Table ta 中的 Column ca 创建了索引 idx_ta_ca，那么任何更新 Column ca 的操作，MySQL 在更新表中 Column ca 的同时，都须要更新

Column ca 的索引数据，调整因为更新带来键值变化的索引信息。而如果没有对 Column ca 进行索引，MySQL 要做的仅仅是更新表中 Column ca 的信息。这样，最明显的资源消耗就是增加了更新所带来的 IO 量和调整索引所致的计算量。此外，Column ca 的索引 idx_ta_ca 须要占用存储空间，而且随着 Table ta 数据量的增加，idx_ta_ca 所占用的空间也会不断增加，所以索引还会带来存储空间资源消耗的增加。

引用

- [美团-MySQL 索引原理及慢查询优化](#)
- [MySQL 索引背后的数据结构及算法原理](#)
- [索引的利弊与如何判定，是否需要索引](#)

MySQL

2016 年 10 月 20 日

22:48

- 空值和 NULL
 - NULL，表示数值未知。
 - 空值**不同于空白或零值**。没有两个相等的空值。比较两个空值或将空值与任何其它数值相比均返回未知，这是因为每个空值均为未知。

MySql 优化

2016 年 8 月 4 日

14:02

1. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。
2. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：
`select id from t where num is null`
可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：
`select id from t where num=0`

3. 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

4. 应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
select id from t where num=10 or num=20
```

可以这样查询：

```
select id from t where num=10
```

```
union all
```

```
select id from t where num=20
```

5. in 和 not in 也要慎用，否则会导致全表扫描，如：

```
select id from t where num in(1,2,3)
```

对于连续的数值，能用 between 就不要用 in 了：

```
select id from t where num between 1 and 3
```

6. 下面的查询也将导致全表扫描：

```
select id from t where name like '%abc%'
```

若要提高效率，可以考虑全文检索。

7. 如果在 where 子句中使用参数，也会导致全表扫描。因为 SQL 只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

```
select id from t where num=@num
```

可以改为强制查询使用索引：

```
select id from t with(index(索引名)) where num=@num
```

8. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2=100
```

应改为：

```
select id from t where num=100*2
```

9. 应尽量避免在 where 子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where substring(name,1,3)='abc'--name 以  
abc 开头的 id
```

```
select id from t where datediff(day,createdate,'2005-11-30')=0-- '2005-11-30' 生成的 id
```

应改为：

```
select id from t where name like 'abc%'  
select id from t where createdate>='2005-11-30' and  
createdate<'2005-12-1'
```

10.不要在 where 子句中的 “=” 左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

12.不要写一些没有意义的查询，如需要生成一个空表结构：

```
select col1,col2 into #t from t where 1=0
```

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

```
create table #t(...)
```

13.很多时候用 exists 代替 in 是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where  
num=a.num)
```

14.并不是所有索引对查询都有效，SQL 是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL 查询可能不会去利用索引，如一表中有字段 sex，male、female 几乎各一半，那么即使在 sex 上建了索引也对查询效率起不了作用。

15.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致

整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
18. 尽可能的使用 varchar/nvarchar 代替 char/nchar，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。
19. 任何地方都不要使用 select * from t，用具体的字段列表代替“*”，不要返回用不到的任何字段。
20. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。
21. 避免频繁创建和删除临时表，以减少系统表资源的消耗。
22. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。
23. 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log，以提高速度；如果数据量不大，为了缓和系统表的资源，应先 create table，然后 insert。
24. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table，然后 drop table，这样可以避免系统表的较长时间锁定。
25. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过 1 万行，那么就应该考虑改写。
26. 使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。
27. 与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游

标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28. 在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON，在结束时设置 SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。
29. 尽量避免大事务操作，提高系统并发能力。
30. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

编程基础

2016 年 3 月 28 日

14:39

- 面向对象

面向对象的三个基本特征是：封装、继承、多态

封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

继承

继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。

要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

多态性

多态性（polymorphism）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

实现多态，有两种方式，覆盖和重载。覆盖和重载的区别在于，覆盖在运行时决定，重载是在编译时决定。并且覆盖和重载的机制不同，例如在

Java 中，重载方法的签名必须不同于原先方法的，但对于覆盖签名必须相同。

C++ 通过继承（ inheritance ）和虚函数（ virtual function ）来实现多态性。

- 重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。
重写：是指子类重新定义父类虚函数的方法。

从实现原理上来说：

重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer; 和 function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int_func、str_func。对于这两个函数的调用，在编译器间就已经确定了，是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关！

重写：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

重定义 (redefining): 垂直关系

子类重新定义父类中有相同名称的非虚函数（参数列表可以不同）。

当基类指针操作子类对象的时候，调用子类中独有的方法的时候，要将基类指针强制类型转换成子类指针。

- 内存分区

堆：有程序员手动分配释放

栈：由编译器自动分配和释放

全局（静态）存储区：存放全局变量和静态变量，包括 DATA 段（全局初始化区）和 BSS 段（全局未初始化区）

文字常量区

程序代码区

- 析构函数
 - 析构函数在类里面声明，在类外定义

Java 总结

2016 年 4 月 12 日

11:12

Java 基础：

4. 构造器 Constructor 是否可被 override
6. 是否可以继承 String 类
7. String 和 StringBuffer、StringBuilder 的区别
8. hashCode 和 equals 方法的关系
9. 抽象类和接口的区别
10. 自动装箱与拆箱
11. 什么是泛型、为什么要使用以及泛型擦除
13. HashMap 实现原理(看源代码)
14. HashTable 实现原理(看源代码)
15. HashMap 和 HashTable 区别
16. HashTable 如何实现线程安全(看源代码)

17. ArrayList 和 vector 区别(看源代码)
18. ArrayList 和 LinkedList 区别及使用场景
19. Collection 和 Collections 的区别
20. ConcurrentHashMap 实现原理(看源代码)
21. Error、Exception 区别
22. Unchecked Exception 和 Checked Exception , 各列举几个
23. Java 中如何实现代理机制(JDK、CGLIB)
24. 多线程的实现方式
25. 线程的状态转换
26. 如何停止一个线程
27. 什么是线程安全
28. 如何保证线程安全
29. Synchronized 如何使用
30. synchronized 和 Lock 的区别
31. 多线程如何进行信息交互
32. sleep 和 wait 的区别(考察的方向是是否会释放锁)
33. 多线程与死锁

34. 如何才能产生死锁

35. 什么叫守护线程，用什么方法实现守护线程

36. Java 线程池技术及原理

37. java 并发包 concurrent 及常用的类

38. volatile 关键字

39. Java 中的 NIO , BIO , AIO 分别是什么

40. IO 和 NIO 区别

41. 序列化与反序列化

42. 常见的序列化协议有哪些

43. 内存溢出和内存泄漏的区别

44. Java 内存模型及各个区域的 OOM , 如何重现 OOM

45. 出现 OOM 如何解决

46. 用什么工具可以查出内存泄漏

47. Java 内存管理及回收算法

48. Java 类加载器及如何加载类(双亲委派)

49. xml 解析方式

50. Statement 和 PreparedStatement 之间的区别

JavaEE:

1. servlet 生命周期及各个方法
2. servlet 中如何自定义 filter
3. JSP 原理
4. JSP 和 Servlet 的区别
5. JSP 的动态 include 和静态 include
6. Struts 中请求处理过程
7. MVC 概念
8. Spring mvc 与 Struts 区别
9. Hibernate/Ibatis 两者的区别
10. Hibernate 一级和二级缓存
11. Hibernate 实现集群部署
12. Hibernate 如何实现声明式事务
13. 简述 Hibernate 常见优化策略
14. Spring bean 的加载过程(推荐看 Spring 的源码)

15. Spring 如何实现 AOP 和 IOC

16. Spring bean 注入方式

17. Spring 的事务管理(推荐看 Spring 的源码)

18. Spring 事务的传播特性

19. springmvc 原理

20. springmvc 用过哪些注解

21. Restful 有几种请求

22. Restful 好处

23. Tomcat , Apache , JBoss 的区别

24. memcached 和 redis 的区别

25. 有没有遇到中文乱码问题 , 如何解决的

26. 如何理解分布式锁

27. 你知道的开源协议有哪些

28. json 和 xml 区别

设计模式 :

1. 设计模式的六大原则
2. 常用的设计模式
3. 用一个设计模式写一段代码或画出一个设计模式的 UML
4. 如何理解 MVC
5. 高内聚，低耦合方面的理解

算法：

1. 深度优先、广度优先算法
2. 排序算法及对应的时间复杂度和空间复杂度
3. 写一个排序算法
4. 查找算法
5. B+树和二叉树查找时间复杂度
6. KMP 算法、hash 算法
7. 常用的 hash 算法有哪些
8. 如何判断一个单链表是否有环？
9. 给你一万个数，如何找出里面所有重复的数？用所有你能想到的方法，时间复杂度和空间复杂度分别是多少？

10. 给你一个数组，如何里面找到和为 K 的两个数？

11. 100000 个数找出最小或最大的 10 个？

12. 一堆数字里面继续去重，要怎么处理？

数据结构：

1. 队列、栈、链表、树、堆、图

2. 编码实现队列、栈

Linux:

1. linux 常用命令

2. 如何查看内存使用情况

3. Linux 下如何进行进程调度

操作系统：

1. 操作系统什么情况下会死锁

2. 产生死锁的必要条件

3. 死锁预防

数据库：

1. 范式
2. 数据库事务隔离级别
3. 数据库连接池的原理
4. 乐观锁和悲观锁
5. 如何实现不同数据库的数据查询分页
6. SQL 注入的原理，如何预防
7. 数据库索引的实现(B+树介绍、和 B 树、R 树区别)
8. SQL 性能优化
9. 数据库索引的优缺点以及什么时候数据库索引失效
- 10.Redis 的存储结构

网络：

1. OSI 七层模型以及 TCP/IP 四层模型
2. HTTP 和 HTTPS 区别

3. HTTP 报文内容
4. get 提交和 post 提交的区别
5. get 提交是否有字节限制，如果有是在哪限制的
6. TCP 的三次握手和四次挥手
7. session 和 cookie 的区别
8. HTTP 请求中 Session 实现原理
9. redirect 与 forward 区别
- 10.DNS
11. TCP 和 UDP 区别

安全：

1. 如果客户端不断的发送请求连接会怎样
2. DDos 攻击
3. DDos 预防
4. 那怎么知道连接是恶意的呢？可能是正常连接

智力题：

1. 给你 50 个红球和 50 个黑球，有两个一模一样的桶，往桶里放球，让朋友去随机抽，采用什么策略可以让朋友抽到红球的概率更高？

2. 从 100 个硬币中找出最轻的那个假币？

Java 基础

2016 年 4 月 12 日

14:53

- 和 C++ 的区别
 - a. Java 是解释型语言，C++ 是编译型语言
 - b. Java 是纯面向对象
 - c. Java 没有指针，但是有引用
 - d. Java 不支持多重继承，但是可以实现多个接口
 - e. Java 不需要显示管理内存分配，没有析构函数，但是有 finalize() 方法，gc 回收的时候会调用该方法
 - f. Java 不支持运算符重载，不支持强制类型转换，不支持默认函数参数
 - g. Java 平台无关，每种数据类型分配固定长度
- Java 中 main 方法不能用 abstract 修饰，且必须被 static、public 修饰，并且保证返回值为 void，其他修饰符随意
- Java 程序初始化的原则
 - a. 静态变量优先初始化，且只初始化一次，非静态变量可初始化多次
 - b. 父类优先子类初始化
 - c. 变量成员按照顺序初始化，且需在方法调用前初始化
- Java 程序初始化执行顺序

父类静态变量 -> 父类静态代码块 -> 子类静态变量 -> 子类静态代码块 -> 父类非静态变量 -> 父类非静态代码块 -> 父类构造函数 -> 子类非静态变量 -> 子类非静态代码块 -> 子类构造函数
- 成员变量生命周期和实例化对象生命周期相同
- 成员变量作用域：

- a. public , 所有类和对象都可以访问
 - b. private , 只有当前类具有访问权限
 - c. protected , 自己和子类可以访问 , 同一 package 可以访问
 - d. default , 只能自己和同一 package 可以访问
- 一个 Java 文件可以写多个类 , 但是最多只能有一个被 public 修饰
- Java 没有指针 , 但是 Java 有引用 , 每一个 new 语句都返回的是一个引用 , Java 处理基本数据类型是按值传递 , 处理其他数据类型都是按引用传递 (比如数组) 。
- clone() 复制构造函数 :
 - a. 浅复制 : 浅复制只复制所考虑的对象 , 而不复制它引用的对象
 - b. 深复制 : 不止复制所考虑的对象 , 而且会把复制对象所引用的对象全部复制一遍
- 反射机制
 - a. 可以在运行时动态的创建类的对象
 - b. 可以得到一个对象所属的类
 - c. 可以获取一个类所有的变量和方法
 - d. 可以运行时调用对象的方法
- Java 创建对象的 4 种方法 :
 - a. new 语句
 - b. 反射机制
 - c. clone() 方法
 - d. 反序列化方法
- C# 和 Java 区别
 - 泛型不一样 , Java 泛型只是编译时的 , 但 C# 的泛型在运行时也被维持 , 而且适用于 value types 和 reference types
 - C# 没有 checked exceptions

- Java 不允许建立 user-defined 的 value types
 - Java 不允许运算符重载
 - Java 没有类似 LINQ 的特性
 - Java 不支持委托
 - C# 没有匿名内部类
 - C#没有像 java 那样的内部类，所有的 nested classes 其实都像 Java 的静态 nested classes
 - Java 没有静态类
 - Java 没有 扩展方法 (extension methods)
 - 两者的访问修饰符有一定区别
 - 两者初始化 (initialization) 的顺序有一定不同。C#初始化变量后才调用父类的构造方法
 - Java 没有类似 “properties” 的东西，而是约定俗成为 getter 和 setter
 - Java 没有类似与"unsafe"的特性
 - 两者的枚举 (enums) 有一定的不同，Java 的更加面向对象
 - Java 的参数只能传值，没有类似于 C#的 ref 和 out 传递引用。
(注 : Java 传递对象只是传递对象引用的 copy)
 - Java 没有 partial types
 - C# 的 interface 不能定义字段
 - Java 没有 unsigned 的整形
 - Java 没有类型与 nullable 的 value types
-
- Java1.8 有哪些新特性
 - 接口实现了非抽象的默认方法
 - 加入了 Lambda 表达式 (本质是一种匿名的内部类) 和函数式编程
 - Java 有哪些基本类型
 - int 、 char、 byte、 long、 boolean、 float、 double、 short
 - int 与 Integer 的区别，分别什么场合使用
 - int 是基本数据类型， Integer 是 int 的封装类
 - 都可以用来表示一个整型数

- 因为数据类型不同，两者不能互用
- java 的基础类型和字节大小。
 - byte 1 个字节
 - short 2 个字节
 - int 4 个字节
 - long 8 个字节
 - float 4 个字节
 - double 8 个字节
 - boolean 1 个字节
 - char 2 个字节
 - 默认变量的初始值

数据类型	初始值
byte	0
short	0
int	0
long	0L
char	'\u0000'
float	0.0f
double	0
boolean	false
所有引用类型	null

- 包装类的 “==” 运算在遇到算术运算的时候不会自动拆箱
◦ 包装了的 equals()方法不处理数据类型转换

- 清理函数 finalize()

一旦垃圾回收器准备好释放对象占用的空间，首先调用 finalize()方法，并且在下一次垃圾回收动作发生的时候，才真正回收对象占用的内存空间。

finalize()函数可能引起外泄，是回收过的对象复活
- 静态变量

静态变量在第一个对象创建的时候初始化一次，后面不再进行初始化。

- Java 的 if 语句等进行判断的时候只能判断 boolean 类型，不能判断 int 等类型，和 C++有所区别
- ceil : 大于等于 floor : 小于等于
- Math.toRadians 函数用来把角度转成弧度
- final 修饰的方法可以重载，不能重写
- 数组是对象，对象存储在堆上
- 函数返回可以进行自动类型转换

Java 面向对象技术

2016 年 4 月 12 日

21:48

- 抽象：过程抽象，数据抽象
- 继承
 - a. Java 不支持多重继承，但支持实现多个接口
 - b. 子类只能继承父类的非私有成员变量和方法
 - c. 子类成员变量和父类成员变量重名时会覆盖父类成员变量
 - d. 子类方法与父类方法有相同的函数签名（方法名、参数类型个数），则子类会覆盖父类

组合是在新类里创建原有类的对象，重复利用已有类的功能

- 封装：信息隐藏、数据保护
- 多态：允许不同的类的对象对同一消息做出响应
Java 多态的实现形式：
 - a. 方法的重载：方法多态性，在同一个类中有多个同名的方法，参数列表不同，**编译时多态**
 - b. 方法的覆盖：子类覆盖父类的方法，要求子类重写的方法要与父类中的方法有相同的返回类型，比父类方法更好访问，不能比父类重写方法声明更多的异常，**运行时多态**

重载和覆盖的区别：

- a. 覆盖是子类和父类之间的关系，是垂直关系；重载是同一类中方法之间的关系，是水平关系
- b. 覆盖只能由一个方法或者只能由一对方法产生关系；重载是多个方法产生关系
- c. 覆盖要求参数列表相同；重载要求参数列表不同
- d. 覆盖关系中，方法体是根据对象的类型决定；重载根据调用的实参表和形参表决定方法体

- 抽象类和接口的异同

相同点：

- a. 都不能被实例化
- b. 接口的实现类或者抽象类的子类只有实现了接口和抽象类的方法后才能被实例化

不同点：

- a. 接口只有定义，方法不能在接口中实现；抽象类可以有定义和实现，其方法可以被子类继承
- b. 接口需要实现；抽象类只能被继承
- c. 接口强调特定功能的实现（has-a）；抽象类强调所属关系（is-a）
- d. 接口成员变量默认 public static final，只能拥有静态的不能被修改的数据成员，且必须赋初值，**所有方法只能被 abstract 和 public 修饰**；抽象类可以有自己的变量成员，也可以有非抽象的成员方法
- e. 接口用于实现常用功能，便于日后维护修改；抽象类用于实现公共角色，不适用于日后做修改

- Java 获取类名和父类类名
`this.getClass().getName()`
`this.getClass().getSuperClass().getName()`
- this 和 super 的区别
this 用来指向当前实例的对象，用来区分对象的成员变量和方法的形参
super 可以用来访问父类的方法和成员变量
- 面向对象开发的六个原则：
 - a. 单一职责：一个类只做它该做的事情；
 - b. 开放封闭：软件实体应该对扩展开放，对修改关闭；
 - c. 里氏替换：任何时候都可以用子类型替换掉父类型；
 - d. 依赖倒置：面向接口编程；
 - e. 合成聚合复用：优先使用聚合或者和合成关系复用代码；
 - f. 接口隔离：接口要小而专，决不能大而全。
- 函数重载：
 - a. 如果传入的数据类型（实参类型），小于方法中声明的形式参数类型，实际数据类型就会被提升。
char 型略有不同，如果无法找到可以接受 char 参数的方法，就会把 char 提升至 int。
 - b. 如果传入的实参类型大于方法中声明的形式参数类型，数据类型则会进行强制转换，可能造成数据丢失。
 - c. 区分重载：每个重载的方法都有独一无二的参数类型列表
- 构造函数
如果没有提供构造函数，编译期会给一个无参的默认构造函数；如果提供了构造函数（有参数），则编译期不会提供无参构造函数，也无法调用无参构造函数。
 - a. 注意：
 - i. 构造函数必须与类名相同，且不能有返回值（void 也不行）
 - ii. 可以有多个构造函数，如果没有编译器会提供一个默认的没有参数的构造函数，不执行任何操作
 - iii. 构造函数可以有 0 个及以上的参数

- iv. 构造函数伴随 new 一起调用，且只能由系统调用，对象实例化时自动调用，且只运行一次
 - v. 构造函数主要完成实例初始化工作
 - vi. 构造函数不能被继承但可以被重载
 - vii. 子类可以使用 super 关键字调用父类构造函数，如果父类构造函数有参数，则子类必须显示调用
 - viii. 编译器生成的构造函数默认修饰符只与当前类修饰符相关
 - ix. 普通方法可以和构造函数同名
 - x. 如果没有提供构造函数，编译期会生成一个无参的默认构造函数；如果提供了构造函数（有参数），则编译期不会提供无参构造函数，也无法调用无参构造函数。
-
- 子类构造函数总是优先调用父类构造函数，如果子类构造函数没有显式调用父类构造函数，则默认调用父类的无参构造函数；如果父类没有无参构造函数（系统会自动给一个无参构造函数）。则必须显式调用父类的无参构造函数。

Java 类/接口总结

2016 年 9 月 6 日

22:57

- Java 内部类
 - 内部类是个编译时的概念，一旦编译成功后，它就与外围类属于两个完全不同的类（当然他们之间还是有联系的）。对于一个名为 OuterClass 的外围类和一个名为 InnerClass 的内部类，在编译成功后，会出现这样两个 class 文件：OuterClass.class 和 OuterClass\$InnerClass.class。
 - 使用内部类的好处：
 - 每个内部类能独立继承一个接口，无论外围类是否已经继承这个接口，对内部类没有影响；

- 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围信息独立；
- 在单个外围类中，可以让多个内部类以不同的方式实现统一接口，或者继承同一个类；
- 创建内部类对象的时刻并不依赖于外围对象的创建；
- 内部类没有"is-a"关系，是一个独立的个体；
- 内部类提供了更好的封装，除了该外围类，其他类都不能访问。

- **类型：**

- **成员内部类**：最普通的内部类，它是外围类的一个成员，所以他是可以无限制的访问外围类的所有成员属性和方法，尽管是 private 的，但是外围类要访问内部类的成员属性和方法则需要通过内部类实例来访问。
 1. 成员内部类中不能存在任何的 static 的变量和方法
 2. 成员内部类是依附于外围类，所以只有创建了外围类才能创建内部类
- **局部内部类**：嵌套在方法和作用于内的，对于这个类主要是用来解决比较复杂的问题，局部内部类和成员内部类的区别在于作用域，只能在方法和属性中使用。
- **匿名内部类**：
 1. 匿名内部类没有访问修饰符；
 2. new 匿名内部类，这个类首先是要存在的；
 3. 当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为 final；
 4. 匿名内部类没有构造函数。
- **静态内部类**：static 修饰符修饰的内部类
 1. 与非静态内部类的区别在于：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围类，但是静态内部类却没有。
 2. 匿名内部类的创建不依赖于外围类的创建。
 3. 不能使用任何外围类的非 static 成员变量和方法。

- **内部类访问外部变量，为什么要加 final？**

final 关键字的目的就是为了保证内部类和外部函数对变量“认识”的一致性。

因为生命周期的原因。方法中的局部变量，方法结束后这个变量就要释放掉，final 保证这个变量始终指向一个对象。

首先，内部类和外部类其实是处于同一个级别，内部类不会因为定义在方法中就会随着方法的执行完毕而跟随者被销毁。问题就来了，如果外部类的方法中的变量不定义 final，那么当外部类方法执行完毕的时候，这个局部变量肯定也就被 GC 了，然而内部类的某个方法还没有执行完，这个时候他所引用的外部变量已经找不到了。如果定义为 final，java 会将这个变量复制一份作为成员变量内置于内部类中，这样的话，由于 final 所修饰的值始终无法改变，所以这个变量所指向的内存区域就不会变

- 接口
 - 抽象方法定义的集合，特殊的抽象类，接口中的方法都是抽象的
 - 接口中成员的修饰符都是 public，常量值默认修饰符 public static final
 - 一个类可以实现多个接口，来间接达到多重继承的效果
 - 没有声明任何方法的接口叫标识接口，仅起标识作用，表明实现它的类是一个特定的类型
- Java 注解 (<http://gityuan.com/2016/01/23/java-annotation/>)
 - 注解是插入你代码中的一种注释或者说是一种元数据；
 - 注解信息可以在编译期间使用预编译工具进行处理，也可以在运行期间用反射机制处理；
 - 与 Annotation 相关的 API
 - 注解类型 API

注解类型	含义
@Documented	表示含有该注解类型的元素会通过 javadoc 或类似工具进行文档化
@Inherited	表示注解类型能自动继承
@Retention	表示注解类型的存活时长

<code>@Target</code>	表示注解类型所使用的程序元素的种类
----------------------	-------------------

- 枚举类型

枚举	含义
<code>@ElementType</code>	程序元素类型，用于 Target 注解类型
<code>@RetentionPolicy</code>	注解保留策略，用于 Retention 注解类型

- 异常和错误类型

异常/错误	含义
<code>@AnnotationTypeMismatchException</code>	当注解经过编译(或序列化)后，注解类型改变的情况下，程序视图访问该注解所对应的元素，则抛出此异常
<code>@IncompleteAnnotationException</code>	当注解经过编译(或序列化)后，将其添加到注解类型定义的情况下，程序视图访问该注解所对应的元素，则抛出此异常。
<code>@AnnotationFormatError</code>	当注解解析器试图从类文件中读取注解并确定注解出现异常时，抛出该错误

- Java 用接口实现函数指针的功能
- 枚举
 - 枚举是类吗？和普通类的区别。

- Stream 类 , unsafe 类

Java 泛型

2016 年 9 月 6 日

23:01

- 泛型
 - 泛型最主要的优点就是让编译器追踪参数类型，执行类型检查和类型转换：编译器保证类型转换不会失败
 - 在编译期，是无法知道 K 和 V 具体是什么类型，只有在运行时才会真正根据类型来构造和分配内存。
- Java 泛型的作用和实现原理？List<String> 可不可以给 List<Object> 使用？

Java 关键字总结

2016 年 4 月 13 日

21:48

- Java 标识符只能由字母、数字、下划线和\$组成，且只能以字母、下划线和\$开头
- break、continue 和 return 区别
 - a. break 用于直接强制跳出当前循环
 - b. continue 停止当次循环，进入下一循环
 - c. return 跳转语句，直接从方法放回调用处

跳出多重循环可以在循环外定义一个标识，然后用带有标识的 break 语句，可以跳出多重循环
- final 用来声明属性、方法和类，表明属性不可变（只是引用不可变，只能指向初始化时指向的对象），方法不可覆盖，类不可被继承
finally 是 try/catch 语句的一部分

`finalize` 是 `Object` 类的一个方法，垃圾回收器执行时会调用回收对象的 `finalize` 方法；垃圾回收期准备释放对象占用空间，会先调用 `finalize` 方法，并在下一次垃圾回收动作发生时才真正回收对象占用内存。

- `assert` 用来做代码的正确性检查，一般是针对一个 `boolean` 语句做检查

- `static` 关键字

- 为某特定的数据类型或对象分配单一的存储空间
- 实现某个方法或者属性与类而不是对象关联在一起

四种用法

- `static` 成员变量

与类绑定在一起，在内存中只有一个复制，只要类被加载，静态变量就会分配空间

- `static` 成员方法

`static` 方法与类绑定，不需要创建对象就可以使用，`static` 不能使用 `this` 和 `super` 关键字

不能调用非 `static` 方法，只能访问类的 `static` 成员变量和方法

- `static` 代码块

类中独立于成员变量和成员函数的代码块，JVM 加载类的时候会执行 `static` 代码块，`static` 代码块只会执行一次

- `static` 内部类

不依赖于外部类实例对象而被实例化，只能访问外部类的静态方法和成员

- Java 语言不能在函数内部定义 `static` 变量

- `volatile` 是一个类型修饰符，用来修饰被不同线程访问修改的变量，被 `volatile` 修饰的变量，系统每次用都是从对应的内存读取的，不会利用缓存，保证一致性。

- `instanceof`，判断一个引用类型的变量所指的对象是不是一个类的实例
eg : `s instanceof Object`

- strictfp 用来保证精确浮点运算
- abstract 只能修饰类和方法，不能修饰属性
- static 表明方法是静态方法，静态方法存储在静态存储区，不需要类被实例化就可以调用
static 可以修饰变量，方法，代码块
- Java 中的 foreach 怎么实现的？
Java 中的 foreach 是使用 iterator 实现的，java api 文档中对于 Iterator 的描述中说过：实现这个接口允许对象成为 foreach 语句目标。
- Transient
类实现 Serializable 接口后，将不需要序列化的属性前添加关键字 transient，序列化对象的时候，这个属性就不会序列化到指定的目的地中。

Java JVM 总结

2016 年 8 月 24 日

21:43

- Java 本地方法栈是用来支持 native 方法（指使用 Java 以外的其他语言编写的方法）的执行。
- 介绍 jvm 内存机制（把各个内存区域作用、回收算法、收集器分类统统说了一遍）
思维导图
- Java 垃圾回收
 - 什么时候发生垃圾回收
 - 对象优先在 Eden 中分配，当 Eden 中没有足够空间时，虚拟机将发生一次 Minor GC，因为 Java 大多数对象都是朝生夕灭，所以 Minor GC 非常频繁，而且速度也很快；

每次 Minor GC 的时候都会把 Eden 区和一个 Survivor 区存活的对象拷贝到另一个 Survivor 区，两个 Survivor 区轮换工作，这就是 Minor GC 使用的复制算法的过程。

默认情况下，经历了几次 Minor GC 后，没有回收的对象进入老年空间。

- Full GC，发生在老年代的 GC，当老年代没有足够的空间时即发生 Full GC，发生 Full GC 一般都会有一次 Minor GC。大对象直接进入老年代，如很长的字符串数组，虚拟机提供一个-XX:PretenureSizeThreshold 参数，令大于这个参数值的对象直接在老年代中分配，避免在 Eden 区和两个 Survivor 区发生大量的内存拷贝；
- 发生 Minor GC 时，虚拟机会检测之前每次晋升到老年代的平均大小是否大于老年代的剩余空间大小，如果大于，则进行一次 Full GC，如果小于，则查看 HandlePromotionFailure 设置是否允许担保失败，如果允许，那只会进行一次 Minor GC，如果不允许，则改为进行一次 Full GC。
- 哪些内存需要被回收

jvm 对不可用的对象进行回收。Java 并不是采用引用计数算法来判定对象是否可用，而是采用根搜索算法(GC Root Tracing)，当一个对象到 GC Roots 没有任何引用相连接，用图论的来说就是从 GC Roots 到这个对象不可达，则证明此对象是不可用的，说明此对象可以被 GC。对于这些不可达对象，也不是一下子就被 GC，而是至少要经历两次标记过程：如果对象在进行根搜索算法后发现没有与 GC Roots 相连接的引用链，那它将会第一次标记并且进行一次筛选，筛选条件是此对象有没有必要执行 finalize() 方法，当对象没有覆盖 finalize() 方法或者 finalize() 方法已经被虚拟机调用执行过一次，这两种情况都被视为没有必要执行 finalize() 方法，对于没有必要执行 finalize() 方法的将会被 GC，对于有必要执行的，对象在 finalize() 方法中可能会自救，也就是重新与引用链上的任何一个对象建立关联即可。

- Java 内存模型

不同的平台，内存模型是不一样的，但是 jvm 的内存模型规范是统一的。其实 java 的多线程并发问题最终都会反映在 java 的内存模型上，所谓线程安全无非是要控制多个线程对某个资源的有序访问或修改。总结 java 的内存模型，要解决两个主要的问题：可见性和有序性。我们都知道计算机有高速缓存的存在，处理器并不是每次处理数据都是取内存的。JVM 定义了自己的内存模型，屏蔽了底层平台内存管理细节，对于 java 开发人员，要清楚在 jvm 内存模型的基础上，如果解决多线程的可见性和有序性。

那么，何谓可见性？多个线程之间是不能互相传递数据通信的，它们之间的沟通只能通过共享变量来进行。Java 内存模型（JMM）规定了 jvm 有主内存，主内存是多个线程共享的。当 new 一个对象的时候，也是被分配在主内存中，每个线程都有自己的工作内存，工作内存存储了主存的某些对象的副本，当然线程的工作内存大小是有限制的。当线程操作某个对象时，执行顺序如下：

- (1) 从主存复制变量到当前工作内存 (read and load)
- (2) 执行代码，改变共享变量值 (use and assign)
- (3) 用工作内存数据刷新主存相关内容 (store and write)

JVM 规范定义了线程对主存的操作指令：read，load，use，assign，store，write。当一个共享变量在多个线程的工作内存中都有副本时，如果一个线程修改了这个共享变量，那么其他线程应该能够看到这个被修改后的值，这就是多线程的可见性问题。

那么，什么是有序性呢？线程在引用变量时不能直接从主内存中引用，如果线程工作内存中没有该变量，则会从主内存中拷贝一个副本到工作内存中，这个过程为 read-load，完成后线程会引用该副本。当同一线程再度引用该字段时，有可能重新从主存中获取变量副本(read-load-use)，也有可能直接引用原来的副本 (use)，也就是说 read,load,use 顺序可以由 JVM 实现系统决定。

线程不能直接为主存中中字段赋值，它会将值指定给工作内存中的变量副本(assign)，完成后这个变量副本会同步到主存储区(store-write)，至于何时同步过去，根据 JVM 实现系统决定。有该字段，则会从主内存中将该字段赋值到工作内存中，这个过程为 read-load，完成后线程会引用该变量副本，当同一线程多次重复对字段赋值时，比如：

Java 代码

```
for(int i=0;i<10;i++)  
    a++;  
for(int i=0;i<10;i++)  
    a++;
```

线程有可能只对工作内存中的副本进行赋值，只到最后一次赋值后才同步到主存储区，所以 assign,store,write 顺序可以由 JVM 实现系统决定。假设有一个共享变量 x，线程 a 执行 $x=x+1$ 。从上面的描述中可以知道 $x=x+1$ 并不是一个原子操作，它的执行过程如下：

1 从主存中读取变量 x 副本到工作内存

2 给 x 加 1

3 将 x 加 1 后的值写回主存

如果另外一个线程 b 执行 $x=x-1$ ，执行过程如下：

1 从主存中读取变量 x 副本到工作内存

2 给 x 减 1

3 将 x 减 1 后的值写回主存

那么显然，最终的 x 的值是不可靠的。假设 x 现在为 10，线程 a 加 1，线程 b 减 1，从表面上看，似乎最终 x 还是为 10，但是多线程情况下会有这种情况发生：

1：线程 a 从主存读取 x 副本到工作内存，工作内存中 x 值为 10

2：线程 b 从主存读取 x 副本到工作内存，工作内存中 x 值为 10

3：线程 a 将工作内存中 x 加 1，工作内存中 x 值为 11

4：线程 a 将 x 提交主存中，主存中 x 为 11

5：线程 b 将工作内存中 x 值减 1，工作内存中 x 值为 9

6：线程 b 将 x 提交到主存中，主存中 x 为 9

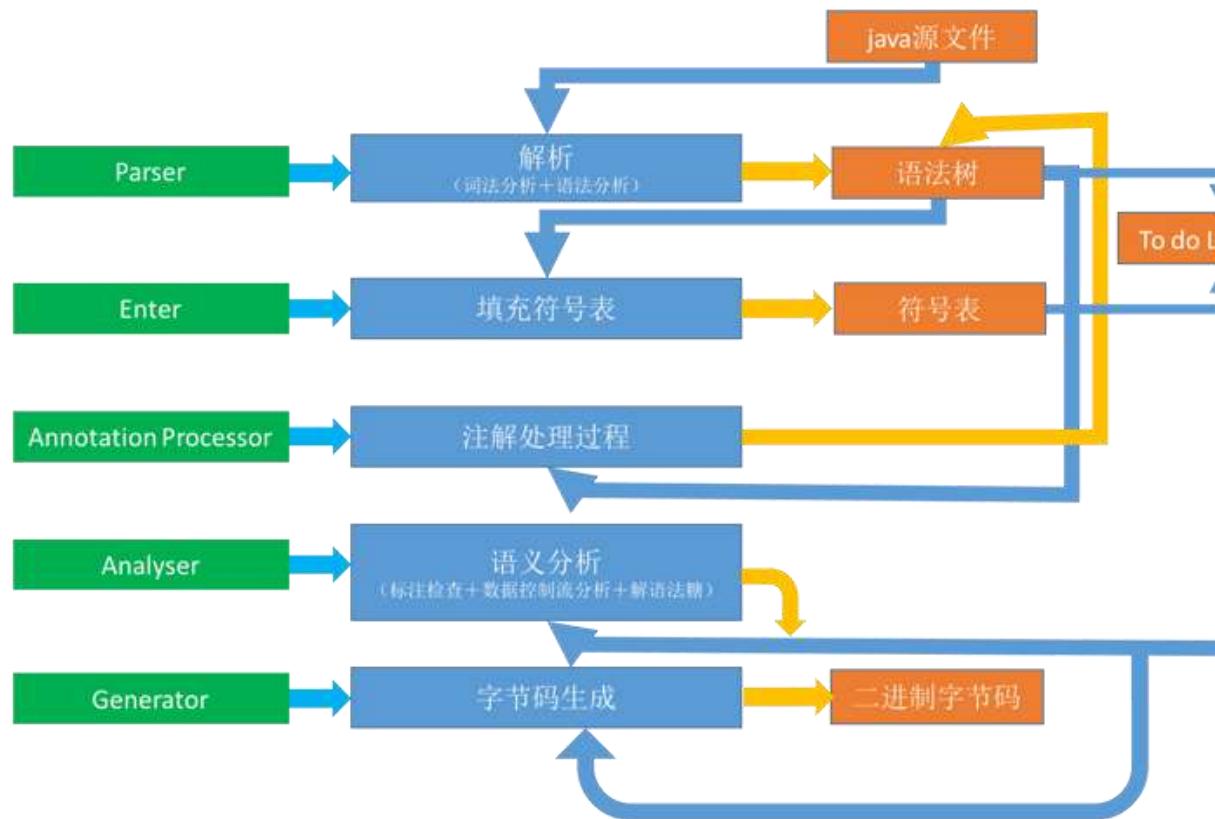
同样，x 有可能为 11，如果 x 是一个银行账户，线程 a 存款，线程 b 扣款，显然这样是有严重问题的，要解决这个问题，必须保证线程 a 和线程 b 是有序执行的，并且每个线程执行的加 1 或减 1 是一个原子操作。

- Minor GC、Major GC 和 Full GC 介绍，什么情况下会触发？
 - Minor GC：从年轻代空间（包括 Eden 和 2 个 Survivor 区域，空间比例为 8 : 1 : 1）回收内存被称为 Minor GC；

- 触发条件：Minor GC 总是在不能为新的对象分配空间的时候触发，例如 Eden 区满了，分配空间的越快，Minor GC 越频繁。
- 所有的 Minor GC 都会触发 stop-the-world 暂停，它意味着会暂停应用的所有线程。
- Minor GC 只会清理年轻代区。
- Major GC 和 Full GC
 - Major GC：清理老年区（Tenured/Old 区）
 - Full GC：清理整个内存堆 – 既包括年轻代也包括年老代
- jvm 参数、调优
- java 有几种引用类型，什么情况下使用软引用，GC 时如何判定哪些软引用需要回收
Java 引用类型：
 - 强引用：绝对不会被回收的引用。
 - 软引用：如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。
可以用来解决 OOM(OutOfMemory)问题，缓存可以使用软引用
- 软引用回收的条件：
 - 当其指示的对象没有任何强引用对象指向它
 - 当虚拟机内存不足时
- 弱引用：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。
- 虚引用：虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
- Java 代码怎么编译成字节码在 JVM 上运行？
编译期静态的 Java 代码编译成二进制字节码的过程：

javac 是编译过程的执行者，流程如下：

- 解析与填充符号表
- 插入式注解处理器的注解处理过程
- 分析与字节码的生成



- Java 虚拟机常用指令：
 - 常量入栈指令
 - const 系列：用于特定常量入栈，入栈常量隐含在指令本身中。
如 aconst_null，将 null 压栈；
 - push 系列：bipush (参数 8 为整数)，sipush (参数 16 位整数)
 - ldc 指令：接受 8 位参数
 - 局部变量入栈指令
 - xload : 指定参数，局部变量压栈
 - xload_n : 对象引用压栈
 - xaload : 数组元素压栈
 - 出栈装入局部变量表指令

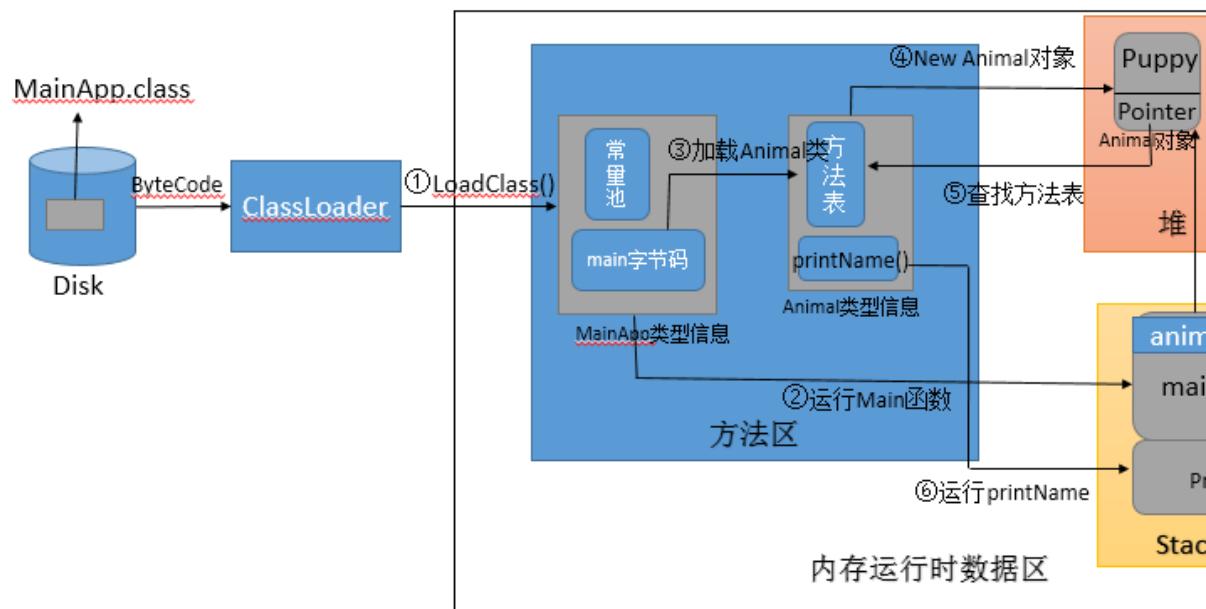
- `xstore`
- `xstore_n`
- `xastore`
- 通用型操作
 - `NOP`, 什么都不做
 - `dup` 指令, 将栈顶元素复制一遍, 并再次压栈
 - `pop` 指令, 栈顶元素弹出
- 类型转换指令
 - `x2y` 形式: `x` 可能是 `i`、`f`、`l`、`d`、`y`, `y` 可能是 `i`、`f`、`l`、`d`、`c`、`s`、`b`
- 运算指令
 - 加法指令: `iadd`、`ladd`、`fadd`、`dadd`
 - 减法指令: `isub`、`lsub`、`fsub`、`dsub`
 - 乘法指令: `imul`、`lmul`、`fmul`、`dmul`
 - 除法指令: `idiv`、`ldiv`、`fdiv`、`ddiv`
 - 取余指令: `irem`、`lrem`、`frem`、`drem`
 - 数值取反: `ineg`、`lneg`、`fneg`、`dneg`
 - 自增指令: `iinc`
 - 位运算指令:
 - 位移指令: `ishl`、`ishr`、`iushr`、`lshl`、`lshr`、`lushr`
 - 按位或指令: `ior`、`lor`
 - 按位与指令: `iand`、`land`
 - 按位异或指令: `ixor`、`lxor`
- 对象/数组操作指令
 - 创建指令:
 - `new`: 创建普通对象
 - `newarray`: 创建基本类型数组
 - `anewarray`: 创建对象数组
 - `multianewarray`: 创建多维数组
 - 字段访问指令:
 - `getfield`、`putfield`: 操作实例对象的字段
 - `getstatic`、`putstatic`: 用于操作类的静态字段
 - 类型检查指令:

- checkcast : 用于检查类型强制转换是否可以进行
 - instanceof : 用于判断给定的对象是否是一个类的实例
- 数组操作指令：
 - xastore
 - xaload
 - arraylength : 获取数组长度
- 比较控制指令
 - 比较指令：比较两个栈顶元素大小，并将结果入栈。dcmpg、dcmpl、fcmpg、fcmpl、lcmp
 - 条件跳转指令：接受两个字节的操作数用于计算跳转位置。同一含义，弹出栈顶元素，测试是否满足某一条件，如果满足，则跳转给定位置。
 - ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull
 - 比较条件跳转指令：类似比较和跳转指令结合体。
 - 接受两个字节的操作数用于计算跳转位置。
 - 执行指令时需要栈顶准备两个元素进行比较。
 - 执行完后，栈顶元素清空，没有任何数据入栈。
 - if_icmpeq、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmpeq、if_acmpne
 - 多条件分支跳转：专为 switch-case 设计，主要有 tableswitch、lookupswitch
 - 无条件跳转：goto
- 函数调用与返回指令
 - 函数调用指令：
 - invokevirtual : 虚函数调用
 - invokeinterface : 接口方法调用
 - invokespecial : 特殊方法调用
 - invokestatic : 静态方法调用
 - invokedynamic : 动态绑定方法调用
 - 返回指令：xreturn，x 根据类型变化
 - 返回 int : ireturn
 - 返回 void : return
 - ...

- 同步控制
 - monitorenter : 监视计数器为 0 , 请求进入 , 不为 0 则判断持有当前监视器的线程是否是自己 , 如果是 , 则进入 , 不是 , 则继续等待。
 - monitorexit : 声明退出。
- class 文件结构
- Java 程序编译和运行的过程
 - java 类运行的过程大概可分为两个过程 : 1、类的加载 2、类的执行。
 - 需要说明的是 : JVM 主要在程序第一次主动使用类的时候 , 才会去加载该类。也就是说 , JVM 并不是在一开始就把一个程序就所有的类都加载到内存中 , 而是到不得不用的时候才把它加载进来 , 而且只加载一次。
 - 下面是程序运行的详细步骤 :
 - 在编译好 java 程序得到 MainApp.class 文件后 , 在命令行上敲 java AppMain 。系统就会启动一个 jvm 进程 , jvm 进程从 classpath 路径中找到一个名为 AppMain.class 的二进制文件 , 将 MainApp 的类信息加载到运行时数据区的方法区内 , 这个过程叫做 MainApp 类的加载。
 - 然后 JVM 找到 AppMain 的主函数入口 , 开始执行 main 函数。
 - main 函数的第一条命令是 Animal animal = new Animal("Puppy"); 就是让 JVM 创建一个 Animal 对象 , 但是这时候方法区中没有 Animal 类的信息 , 所以 JVM 马上加载 Animal 类 , 把 Animal 类的类型信息放到方法区中。
 - 加载完 Animal 类之后 , Java 虚拟机做的第一件事情就是在堆区中为一个新的 Animal 实例分配内存 , 然后调用构造函数初始化 Animal 实例 , 这个 Animal 实例持有着指向方法区的 Animal 类的类型信息 (其中包含有方法表 , java 动态绑定的底层实现) 的引用。
 - 当使用 animal.printName() 的时候 , JVM 根据 animal 引用找到 Animal 对象 , 然后根据 Animal 对象持有的引用定位到方法

区中 Animal 类的类型信息的方法表，获得 printName()函数的字节码的地址。

- 开始运行 printName()函数。



- 如何优化 jvm 参数 (堆大小、xmx 一般和 xms 设成一样大、永久代大小、收集器选择、收集器参数、新生代对象年龄阈值等)
- JVM 判断两个类是否相等，不仅需要判断类名是否相同，还要判断是不是同一个加载器加载的

Java 多线程和并发

2016 年 9 月 6 日

22:25

- java 有哪些线程同步器

- CountDownLatch

- 使用场景：

- 需要等待某个条件达成要求后才能继续后面的事情；

- 同时当线程都完成后也会触发事件，以便进行后面的操作。
 - 【倒计时锁】，线程中调用 countDownLatch.await() 使进程进入阻塞状态，当达成指定次数后（通过 countDownLatch.countDown() ）继续执行每个线程中剩余的内容。
 - AbstractQueuedSynchronizer
 - 提供了一个基于 FIFO 的队列，用于可以构建锁和相关其他同步装置的基础框架。
 - Semaphore
 - 通过计数器控制对共享资源的访问。
 - CyclicBarrier
 - 允许一组线程互相等待，直到达成某个公共屏障点（ common barrier point），然后所有这组线程再同步往后面执行。
 - 【Cyclic 周期，循环的 Barrier 屏障，障碍】循环的等待阻塞的线程个数到达指定数量后使参与计数的线程继续执行并可执行特定线程（使用不同构造函数可以不设定到达后执行），其他线程仍处于阻塞等待再一次达成指定个数。
 - Exchanger
 - 用于线程间进行数据交换
 - Phaser
 - 兼具了 CountDownLatch 与 CyclicBarrier 的功能，并提供了分阶段的能力。
-
- **什么是 CAS？什么状况下不适合用 CAS？什么是 ABA 问题？**
 - CAS : Compare and Swap，**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，否则返回 V**，是实现乐观锁的一种方式。
 - 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。
 - 流程：首先，CPU 会将内存中将要被更改的数据（内存值 V ）与期望的值（预期值 A ）做比较。然后，当这两个值相等时，CPU 才会将内存中的数值（内存值 V ）替换为新的值（新值 B ）。否则便不做操作。最后，CPU 会将旧的数值返回。这一系列的操作是原子的。

CAS 的缺点：

- 会产生 ABA 问题；
- 循环时间长开销大；
- 只能保证一个共享变量的原子操作。

CAS 不适用情况：多线程频繁抢占资源的时候会使自旋时间过长，资源开销大。

ABA 问题：

- CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。通俗点：**当前线程认为共享变量的当前值是期望值，当时可能存在其他线程已经经过几次修改之后仍就保持当前值是期望值，就存在问题。**

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。

- 解决办法

部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少。

- **乐观锁和悲观锁**

- 悲观锁：独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。如 Synchronized。
- 乐观锁：每次不加锁，假设没有冲突去完成某项操作，如果因为冲突失败就重试，直到成功为止。CAS 是一种实现方式。

- **Synchronized 底层实现？**

通过字节码指令 monitorenter 和 monitorexit 来隐式使用 lock 和 unlock 操作，这两个字节码指令反应到 Java 代码就是 Synchronized 关键字。

- **SimpleDateFormat 如何实现线程安全**

```
private static ThreadLocal<SimpleDateFormat> tl = new  
ThreadLocal<SimpleDateFormat>();  
  
public static Date formatDate(String d) {  
    try {  
        SimpleDateFormat sdf = tl.get();  
        if (sdf == null) {  
            sdf = new SimpleDateFormat("yyyy-MM-dd");  
            tl.set(sdf);  
        }  
        return sdf.parse(d);  
    }  
    catch(Exception e)  
    {  
        e.printStackTrace();  
    }  
    return null;  
}
```

- **Java 线程池 (<http://gityuan.com/2016/01/16/thread-pool/>)**

Java 通过 Executors 可以创建四种线程池：

- **newCachedThreadPool** 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。线程池大小上限为 Integer.MAX_VALUE。
- **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（ LinkedBlockingQueue ）中等待。
- **newScheduledThreadPool** 创建一个定长线程池，支持定时及周期性任务执行。
- **newSingleThreadExecutor** 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。所有线程都保存在队列（ LinkedBlockingQueue ）中。

对比：

工厂方法	corePoolSize	maximumPoolSize	keepAliveTime	workQueue
newCachedThreadPool	0	Integer.MAX_VALUE	60s	Synchronous Queue
newFixedThreadPool	nThreads	nThreads	0	LinkedBlockingQueue
newSingleThreadExecutor	1	1	0	LinkedBlockingQueue
newScheduledThreadPool	corePoolSize	Integer.MAX_VALUE	0	DelayedWorkQueue

线程池的好处：

- 重用存在的线程，减少对象创建、消亡的开销，性能更好；
- 可以有效的控制并发线程数，提高系统的资源利用率，同时避免过多的资源竞争，避免堵塞；
- 提供定时执行、定期执行、单线程和并发控制等功能。

创建新线程必须满足的条件：

- corePoolSize (线程池基本大小) 必须大于或等于 0 ；
- maximumPoolSize (线程池最大大小) 必须大于或等于 1 ；
- maximumPoolSize 必须大于或等于 corePoolSize ；
- keepAliveTime (线程存活保持时间) 必须大于或等于 0 ；
- workQueue (任务队列) 不能为空；
- threadFactory (线程工厂) 不能为空，默认为 DefaultThreadFactory 类
- handler (线程饱和策略) 不能为空，默认策略为 ThreadPoolExecutor.AbortPolicy 。

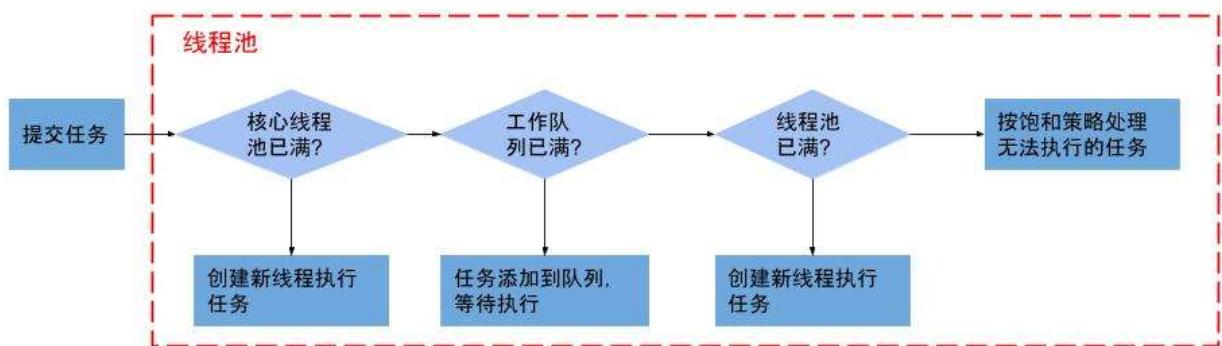
参数详解：

- corePoolSize (线程池基本大小) : 当向线程池提交一个任务时，若线程池已创建的线程数小于 corePoolSize ，即便此时存在空闲线程，

也会通过创建一个新线程来执行该任务，直到已创建的线程数大于或等于 corePoolSize 时，才会根据是否存在空闲线程，来决定是否需要创建新的线程。除了利用提交新任务来创建和启动线程（按需构造），也可以通过 prestartCoreThread() 或 prestartAllCoreThreads() 方法来提前启动线程池中的基本线程。

- maximumPoolSize (线程池最大大小) : 线程池所允许的最大线程个数。当队列满了，且已创建的线程数小于 maximumPoolSize，则线程池会创建新的线程来执行任务。另外，对于无界队列，可忽略该参数。
- keepAliveTime (线程存活保持时间) : 默认情况下，当线程池的线程个数多于 corePoolSize 时，线程的空闲时间超过 keepAliveTime 则会终止。但只要 keepAliveTime 大于 0，allowCoreThreadTimeOut(boolean) 方法也可将此超时策略应用于核心线程。另外，也可以使用 setKeepAliveTime() 动态地更改参数。
- unit (存活时间的单位) : 时间单位，分为 7 类，从细到粗顺序：NANOSECONDS (纳秒) , MICROSECONDS (微妙) , MILLISECONDS (毫秒) , SECONDS (秒) , MINUTES (分) , HOURS (小时) , DAYS (天) ;
- workQueue (任务队列) : 用于传输和保存等待执行任务的阻塞队列。可以使用此队列与线程池进行交互：
 - 如果运行的线程数少于 corePoolSize，则 Executor 始终首选添加新的线程，而不进行排队。
 - 如果运行的线程数等于或多于 corePoolSize，则 Executor 始终首选将请求加入队列，而不添加新的线程。
 - 如果无法将请求加入队列，则创建新的线程，除非创建此线程超出 maximumPoolSize，在这种情况下，任务将被拒绝。
- threadFactory (线程工厂) : 用于创建新线程。由同一个 threadFactory 创建的线程，属于同一个 ThreadGroup，创建的线程优先级都为 Thread.NORM_PRIORITY，以及是非守护进程状态。threadFactory 创建的线程也是采用 new Thread() 方式，threadFactory 创建的线程名都具有统一的风格：pool-m-thread-n (m 为线程池的编号，n 为线程池内的线程编号)；

- handler (线程饱和策略) : 当线程池和队列都满了 , 则表明该线程池已达饱和状态。
 - ThreadPoolExecutor.AbortPolicy : 处理程序遭到拒绝 , 则直接抛出运行时异常 RejectedExecutionException 。(默认策略)
 - ThreadPoolExecutor.CallerRunsPolicy : 调用者所在线程来运行该任务 , 此策略提供简单的反馈控制机制 , 能够减缓新任务的提交速度。
 - ThreadPoolExecutor.DiscardPolicy : 无法执行的任务将被删除。
 - ThreadPoolExecutor.DiscardOldestPolicy : 如果执行程序尚未关闭 , 则位于工作队列头部的任务将被删除 , 然后重新尝试执行任务 (如果再次失败 , 则重复此过程) 。



- Synchronized 实现原理

如果对上面的执行结果还有疑问 , 也先不用急 , 我们先来了解 Synchronized 的原理 , 再回头上面的问题就一目了然了。我们先通过反编译下面的代码来看看 Synchronized 是如何实现对代码块进行同步的 :



```

1 package com.paddx.test.concurrent;
2
3 public class SynchronizedDemo {
4     public void method() {
5         synchronized (this) {
6             System.out.println("Method 1 start");
7     }
  
```

```
8  }
9 }
```



反编译结果：

```
liuxpdeMacBook-Pro:classes liuxp$ javap -c com.paddx.test.concurrent.SynchronizedDemo
Compiled from "SynchronizedDemo.java"
public class com.paddx.test.concurrent.SynchronizedDemo {
    public com.paddx.test.concurrent.SynchronizedDemo();
        Code:
            0: aload_0
            1: invokespecial #1                  // Method java/lang/Object."<init>":()V
            4: return

    public void method();
        Code:
            0: aload_0
            1: dup
            2: astore_1
            3: monitoreenter
            4: getstatic      #2                  // Field java/lang/System.out:Ljava/io/Print
            7: ldc           #3                  // String Method 1 start
            9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava
            12: aload_1
            13: monitorexit
            14: goto          22
            17: astore_2
            18: aload_1
            19: monitorexit
            20: aload_2
            21: athrow
            22: return
```

关于这两条指令的作用，我们直接参考 JVM 规范中描述：

monitoreenter :

每个对象有一个监视器锁（monitor）。当 monitor 被占用时就会处于锁定状态，线程执行 monitoreenter 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 :

执行 monitorexit 的线程必须是 objectref 所对应的 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。

通过这两段描述，我们应该能很清楚的看出 Synchronized 的实现原理，Synchronized 的语义底层是通过一个 monitor 的对象来完成，其实 wait/notify 等方法也依赖于 monitor 对象，这就是为什么只有在同步的块或者方法中才能调用 wait/notify 等方法，否则会抛出 java.lang.IllegalMonitorStateException 的异常的原因。

我们再来看一下同步方法的反编译结果：

源代码：

```
1 package com.paddx.test.concurrent;
2
3 public class SynchronizedMethod {
4     public synchronized void method() {
5         System.out.println("Hello World!");
6     }
7 }
```

反编译结果：

```

public synchronized void method();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
stack=2, locals=1, args_size=1
  0: getstatic    #2                  // Field java/lang/System.out:Ljava/io/PrintWriter;
  3: ldc          #3                  // String Hello World!
  5: invokevirtual #4                // Method java/io/PrintStream.println:(Ljava/lang/String);
  8: return
LineNumberTable:
  line 5: 0
  line 6: 8
LocalVariableTable:
  Start  Length  Slot  Name   Signature
    0       9      0  this   Lcom/paddx/test/concurrent/SynchronizedMethod;

```

从反编译的结果来看，方法的同步并没有通过指令 monitoreenter 和 monitorexit 来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了 ACC_SYNCHRONIZED 标示符。JVM 就是根据该标示符来实现方法的同步的：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。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。其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

- volatile 的作用 (1. 保证内存可见性 , 2. 禁止指令重排序)
 - 保证可见性的原理：让所有的线程读取的变量都是从内存中读取，而不是从缓存（工作内存）中读取，维护变量值的一致性
 - 禁止指令重排的原理
 - 指令重排序是 JVM 为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。
 - volatile 关键字通过提供“内存屏障”的方式来防止指令被重排序，为了实现 volatile 的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。
 - 不能保证内存的原子性，只能保证可见性，因此不能替代 synchronized，并且使用 volatile 会阻止编译器对代码的优化，降低程序执行效率。

- Synchronized 保证了操作的原子性和可见性。

volatile 为什么不能保证线程安全？

顺序性，顾名思义，就是必须保证程序是按照你所预想的逻辑执行的。

可见性，写入缓存与从缓存读出是一个完整的动作，保证变化的值即时可见。这是 volatile 能保证的。

原子性，保证一个处理（或一段代码）不被打断地执行完毕，这是原子性。

原子性和可见性确实容易搞混，可见性比原子性控制范围更小，仅仅控制缓存的一次读写。而原子性控制整段处理。

举个例子：使用 volatile 声明一个变量 a，然后对变量 a 做如下操作。

```
a=a+1;
```

```
a=a+b;
```

volatile 无法保证这两条语句不被打断的执行完毕，只有使用同步化关键字保证它的原子性才可以。

- 都用过哪些 java 容器，LinkedList 都一般都在什么时候用到（经常插入删除时、实现队列和栈时）
- 介绍 ConcurrentHashMap（分段加锁，几乎每个面试都问这个。。。）
- Object 里头都有哪些方法，着重问了 clone(深复制还是浅复制)、finalize（一般在什么时候用，回收时一定能被运行）
- 如何让线程 A 等待线程 B 结束后再执行（join、单线程池），还反问单线程池真的可以吗，所以大致和他介绍了下阻塞队列的机制
- 创建线程方式（实现 runnable 接口、集成 Thread、线程池）
- java 都有哪些加锁方式（synchronized、ReentrantLock、共享锁、读写锁等）
- 想让所有线程都等到一个时刻同时执行有哪些方法（介绍了下 CountDownLatch 和 CyclicBarrier）

- CountDownLatch 和 CyclicBarrier 适用场景都是某一个任务想要往下执行必须依靠其他任务的执行完毕才可以。
- CountDownLatch 是设定一个计数器，当其它任务通过 countDown() 方法将计数器值减为 0 时触发阻塞在 await() 方法的任务。
- CyclicBarrier 和 CountDownLatch 功能一样，不过它能循环使用。

CountDownLatch 和 CycleBarrier 的区别：

- CountDownLatch 是一个或多个线程等待计数达成后继续执行， await()调用并没有参与计数。
- CyclicBarrier 则是 N 个线程等待彼此执行到零界点之后再继续执行， await()调用的同时参与了计数，并且 CyclicBarrier 支持条件达成后执行某个动作，而且这个过程是循环性的。

Java 并发编程的同步器（使线程可以等待另一个线程的对象）

常用的：CountDownLatch、Semaphore

不常用：CyclicBarrier、Exchange

- 线程间通信方法，Java 如何实现
 - 如果使用 Synchronized 同步，则可以使用 wait(), notify(), notifyAll() 三个方法进行通信
 - 如果使用 Lock 进行同步，则可以使用 java.util.concurrent.locks 下的 Condition 类进行通信
 - 可以使用共享变量进行通信
 - 可以使用管道流
 - 线程实现有哪几种方法？有哪种方法是不用 new 也可以创建一个线程的。
 - 实现 Runnable 接口，重写 run 方法，用 start 方法启动线程
 - 继承 Thread 类，重写 run 方法，用 start 方法启动线程
 - 创建 Callable 接口实现类，实现 call 方法
- 利用线程池不用 new 创建线程，线程可复用
- Java 对象锁的概念

- java 中的每个对象都有一个锁，当访问某个对象的 synchronized 方法时，表示将该对象上锁，此时其他任何线程都无法再去访问该 synchronized 方法了，直到之前的那个线程执行方法完毕后，其他线程才有可能去访问该 synchronized 方法。
 - 如果一个对象有多个 synchronized 方法，某一时刻某个线程已经进入到某个 synchronized 方法，那么在该方法没有执行完毕前，其他线程无法访问该对象的任何 synchronized 方法的，但可以访问非 synchronized 方法。
 - 如果 synchronized 方法是 static 的，那么当线程访问该方法时，它锁的并不是 synchronized 方法所在的对象，而是 synchronized 方法所在对象的对应的 Class 对象，因为 java 中无论一个类有多少个对象，这些对象会对应唯一一个 Class 对象，因此当线程分别访问同一个类的两个对象的 static , synchronized 方法时，他们的执行也是按顺序来的，也就是说一个线程先执行，一个线程后执行。
-
- Synchronized 和 lock 有什么缺点？
 - 相同点：Lock 可以完成 Synchronized 所有的功能
 - 不同点：Lock 有比 Synchronized 更精确的线程和更好的性能，Synchronized 自动释放锁，Lock 要求程序员手动释放，并且必须在 finally 中释放。
 - Synchronized 的缺点：当某个线程进入同步方法获得对象锁，那么其他线程访问这里对象的同步方法时，必须等待或者阻塞，这对高并发的系统是致命的，这很容易导致系统的崩溃。如果某个线程在同步方法里面发生了死循环，那么它就永远不会释放这个对象锁，那么其他线程就要永远的等待。这是一个致命的问题。
 - Lock 添加了锁投票、定时锁等候、可中断锁等候等新的特性，在激烈竞争下拥有更好的性能

一、synchronized 与 Lock

当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1) 获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2) 线程执行发生异常 , 此时 JVM 会让线程自动释放锁。

Lock 相对 synchronized 的优点 :

- 1、Lock 不是 Java 语言内置的 , synchronized 是 Java 语言的关键字 , 因此是内置特性。 Lock 是一个类 , 通过这个类可以实现同步访问 ;
- 2、Lock 和 synchronized 有一点非常大的不同 , 采用 synchronized 不需要用户去手动释放锁 , Lock 则必须要用户去手动释放锁 ;
- 3、可以不让等待的线程一直无期限地等待下去 (比如只等待一定的时间或者能够响应中断) ;
- 4、存在读、写锁 , 允许多个线程同时执行读操作 ;
- 5、通过 Lock 可以知道线程有没有成功获取到锁。这是 synchronized 无法办到的 ;

在性能上来说 , 如果竞争资源不激烈 , 两者的性能是差不多的 , 而当竞争资源非常激烈时 (即有大量线程同时竞争) , 此时 Lock 的性能要远远优于 synchronized 。所以说 , 在具体使用时要根据适当情况选择。

二、 java.util.concurrent.locks 包下常用的类

lock() : 用来获取锁。如果锁已被其他线程获取 , 则进行等待 (平常使用得最多的一个方法)

```
Lock lock = ...;  
lock.lock();  
try{  
    //处理任务  
}catch(Exception ex){  
}  
finally{  
    lock.unlock(); //释放锁  
}
```

tryLock() : 有返回值的 , 它表示用来尝试获取锁 , 如果获取成功 , 则返回 true , 如果获取失败 (即锁已被其他线程获取) , 则返回 false ,

也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

tryLock(long time, TimeUnit unit) : 和 tryLock() 是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回 false。如果一开始拿到锁或者在等待期间内拿到了锁，则返回 true。

```
Lock lock = ...;
if(lock.tryLock()) {
    try{
        //处理任务
    }catch(Exception ex){
    }finally{
        lock.unlock(); //释放锁
    }
} else {
    //如果不能获取锁，则直接做其他事情
}
```

lockInterruptibly() : 比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过 lock.lockInterruptibly() 想获取某个锁时，假若此时线程 A 获取到了锁，而线程 B 只有在等待，那么对线程 B 调用 threadB.interrupt() 方法能够中断线程 B 的等待过程。

```
public void method() throws InterruptedException
{ lock.lockInterruptibly(); try {.....} finally { lock.unlock(); }}
```

三. 锁的相关概念介绍

1. 可重入锁

如果锁具备可重入性，则称作为可重入锁。像 synchronized 和 ReentrantLock 都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个 synchronized 方法时，比如说 method1，而在 method1 中会调用另外一个 synchronized 方法

method2，此时线程不必重新去申请锁，而是可以直接执行方法 method2。

```
class MyClass {  
    public synchronized void method1() {  
        method2();  
    }  
    public synchronized void method2() {  
    }  
}
```

2. 可中断锁

在 Java 中，synchronized 就不是可中断锁，而 Lock 是可中断锁。

如果某一线程 A 正在执行锁中的代码，另一线程 B 正在等待获取该锁，可能由于等待时间过长，线程 B 不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

3. 公平锁

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该锁，这种就是公平锁。

在 Java 中，synchronized 就是非公平锁，它无法保证等待的线程获解锁的顺序。

4. 读写锁 读写锁将对一个资源（比如文件）的访问分成了 2 个锁，一个读锁和一个写锁。正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。ReadWriteLock 就是读写锁，它是一个接口，ReentrantReadWriteLock 实现了这个接口。

- 除了 synchronized 和 lock 还有哪些保证线程安全的方法？
 - 同步（Synchronized）
 - 使用原子类（concurrent class）

- 实现并发锁 (Lock)
 - 使用 volatile 关键字
 - 使用不变类和线程安全类
-
- 上下文切换

上下文切换（有时也称做进程切换或任务切换）是指 CPU 从一个进程或线程切换到另一个进程或线程。
上下文是指某一时间点 CPU 寄存器和程序计数器的内容。寄存器是 CPU 内部的数量较少但是速度很快的内存（与之对应的是 CPU 外部相对较慢的 RAM 主内存）。寄存器通过对常用值（通常是运算的中间值）的快速访问来提高计算机程序运行的速度。程序计数器是一个专用的寄存器，用于表明指令序列中 CPU 正在执行的位置，存的值为正在执行的指令的位置或者下一个将要被执行的指令的位置，具体依赖于特定的系统。
稍微详细描述一下，上下文切换可以认为是内核（操作系统的内核）在 CPU 上对于进程（包括线程）进行以下的活动：

 - 挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处，
 - 在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复，
 - 跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程。

上下文切换是计算密集型操作，需要消耗大量的 CPU 时间，Linux 系统优于其他系统的其中一个优点就是上下文切换时间消耗小。

进程线程上下文切换的区别：

进程切换分两步

1. 切换页目录以使用新的地址空间
2. 切换内核栈和硬件上下文。

对于 Linux 来说，线程和进程的最大区别就在于地址空间。

- 线程同步的基本原理

java 会为每个 object 对象分配一个 monitor，当某个对象的同步方法（synchronized methods）或同步块被多个线程调用时，该对象的 monitor 将负责处理这些访问的并发独占要求。

当一个线程调用一个对象的同步方法时，JVM 会检查该对象的 monitor。如果 monitor 没有被占用，那么这个线程就得到了 monitor 的占有权，可以继续执行该对象的同步方法；如果 monitor 被其他线程所占用，那么该线程将被挂起，直到 monitor 被释放。

当线程退出同步方法调用时，该线程会释放 monitor，这将允许其他等待的线程获得 monitor 以使对同步方法的调用执行下去。

注意：java 对象的 monitor 机制和传统的临界检查代码区技术不一样。

java 的一个类一个同步方法并不意味着同时只有一个线程独占执行（不同对象的同步方法可以同时执行），但临界检查代码区技术确会保证同步方法在一个时刻只被一个线程独占执行。

java 的 monitor 机制的准确含义是：任何时刻，对一个指定 object 对象的某同步方法只能由一个线程来调用。

java 对象的 monitor 是跟随 object 实例来使用的，而不是跟随程序代码。

两个线程可以同时执行相同的同步方法，比如：一个类的同步方法是 xMethod()，有 a,b 两个对象实例，一个线程执行 a.xMethod()，另一个线程执行 b.xMethod(). 互不冲突。

- wait , notify 和 notifyAll
 - obj.wait()方法使本线程挂起，并释放 obj 对象的 monitor，只有其他线程调用 obj 对象的 notify()或 notifyAll()时，才可以被唤醒。
 - obj.notifyAll()方法唤醒所有阻塞在 obj 对象上的沉睡线程，然后被唤醒的众多线程竞争 obj 对象的 monitor 占有权，最终得到的那个线程会继续执行下去，但其他线程继续等待。
 - obj.notify()方法是随机唤醒一个沉睡线程，过程更 obj.notifyAll()方法类似。

wait , notify 和 notifyAll 只能在同步控制方法或者同步控制块里面使用。

- Java 的 join 方法和 yield 方法的区别
 - join 方法
线程实例的方法 join()方法可以使一个线程在另一个线程结束后再执行。如果 join()方法在一个线程实例上调用，当前运行着的线程将阻塞直到这个线程实例完成了执行。

在 join()方法内设定超时，使得 join()方法的影响在特定超时后无效。
当超时时，主方法和任务线程申请运行的时候是平等的。然而，当涉及 sleep 时，join()方法依靠操作统计时，所以你不应该假定 join()方法将会等待你指定的时间。

- yield 方法

- Yield 是一个静态的原生(native)方法
- Yield 告诉当前正在执行的线程把运行机会交给线程池中拥有相同优先级的线程。
- Yield 不能保证使得当前正在运行的线程迅速转换到可运行的状态
- 它仅能使一个线程从运行状态转到可运行状态，而不是等待或阻塞状态

Java IO 总结

2016 年 9 月 6 日

22:30

- 讲讲 IO 里面的常见类，字节流字符流。
 - 字符流
 - Reader
 - BufferedReader : 可以用来按行读取文件
 - InputStreamReader
 - FileReader
 - StringReader
 - PipedReader
 - ByteArrayReader
 - FilterReader
 - Writer
 - BufferedWriter
 - OutputStreamWriter
 - FileWriter
 - PrintWriter
 - StringWriter
 - PipedWriter
 - CharArrayWriter
 - FilterWriter

- 字节流
 - inputStream
 - FileInputStream
 - FilterInputStream
 - BufferedInputStream
 - DataInputStream
 - PushbackInputStream
 - ObjectInputStream
 - PipedInputStream
 - SequenceInputStream
 - StringBufferInputStream
 - ByteArrayInputStream
 - OutputStream
 - FileOutputStream
 - FilterOutputStream
 - BufferedOutputStream
 - DataOutputStream
 - PrintStream
 - ObjectOutputStream
 - PipedOutputStream
 - ByteArrayOutputStream
- 讲讲 NIO。
 - NIO 通过 Selector、Channel 和 Buffer 来实现非阻塞性 IO；
 - 使用了反应器模式；
 - 通过轮询方式来查找是否某一个 Channel 有注册的事件发生；
 - 效率比使用 Socket 高效得多。

NIO 和 IO 的区别：

- IO 以流的方式传输数据，面向流的 IO 一次一个字节的处理数据，一个输入流产生一个字节，一个输出流就消费一个字节。对于数据的处理非常的方便，但是处理起来很慢；

- NIO 以块的方式进行数据处理，面向块的 IO 系统以块的形式处理数据。每一个操作都在一步中产生或消费一个数据块。按块要比按流快的多，但面向块的 IO 缺少了面向流 IO 所具有的优雅和简单性。
- IO 中的读和写，对应的是数据和 Stream，NIO 中的读和写，则对应的就是通道和缓冲区。NIO 中从通道中读取：创建一个缓冲区，然后让通道读取数据到缓冲区。NIO 写入数据到通道：创建一个缓冲区，用数据填充它，然后让通道用这些数据来执行写入。

标准 NIO：

- 核心对象：Buffer 和 Channel
- Buffer

Buffer 是一个对象，它包含一些要写入或读出的数据。在 NIO 中，数据是放入 buffer 对象的，而在 IO 中，数据是直接写入或者读到 Stream 对象的。**应用程序不能直接对 Channel 进行读写操作，而必须通过 Buffer 来进行**，即 Channel 是通过 Buffer 来读写数据的。

在 NIO 中，所有的数据都是用 Buffer 处理的，它是 NIO 读写数据的中转池。Buffer 实质上是一个数组，通常是一个字节数据，但也可以是其他类型的数组。但一个缓冲区不仅仅是一个数组，重要的是它提供了对数据的结构化访问，而且还可以跟踪系统的读写进程。

使用 Buffer 读写数据一般遵循以下四个步骤：

- 写入数据到 Buffer；
- 调用 flip() 方法；
- 从 Buffer 中读取数据；
- 调用 clear() 方法或者 compact() 方法。

当向 Buffer 写入数据时，Buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip() 方法将 Buffer **从写模式切换到读模式**。在读模式下，可以读取之前写入到 Buffer 的所有数据。

一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear() 或 compact() 方法。clear() 方法会清空整个缓冲区。compact() 方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

Buffer 主要有如下几种：



控制 Buffer 状态的三个变量：

- position：跟踪已经写了多少数据或读了多少数据，它指向的是下一个字节来自哪个位置
- limit：代表还有多少数据可以取出或还有多少空间可以写入，它的值小于等于 capacity。
- capacity：代表缓冲区的最大容量，一般新建一个缓冲区的时候，limit 的值和 capacity 的值默认是相等的。

Buffer 的 flip 和 clear 方法：

- flip :

```

public final Buffer flip() {
    limit = position;
    position = 0;
    mark = -1;
    return this;
}

```

- clear :

```

public final Buffer clear() {
    position = 0;
    limit = capacity;
    mark = -1;
    return this;
}

```

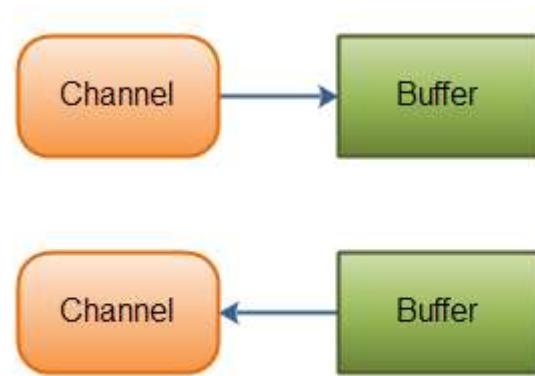
- Channel :

Channel 是一个对象，可以通过它读取和写入数据。可以把它看做 IO 中的流。但是它和流相比还有一些不同：

- Channel 是双向的，既可以读又可以写，而流是单向的
- Channel 可以进行异步的读写
- 对 Channel 的读写必须通过 buffer 对象

正如上面提到的，所有数据都通过 Buffer 对象处理，所以，您永远不会将字节直接写入到 Channel 中，相反，您是将数据写入到 Buffer 中；同样，您也不会从 Channel 中读取字节，而是将数据从 Channel 读入 Buffer，再从 Buffer 获取这个字节。

因为 Channel 是双向的，所以 Channel 可以比流更好地反映出底层操作系统的真实情况。特别是在 Unix 模型中，底层操作系统通常都是双向的。



在 Java NIO 中 Channel 主要有如下几种类型：

- FileChannel：从文件读取数据的
- DatagramChannel：读写 UDP 网络协议数据
- SocketChannel：读写 TCP 网络协议数据
- ServerSocketChannel：可以监听 TCP 连接

网络 NIO (异步 I/O) :

- 异步 I/O :

异步 I/O 是一种**没有阻塞**地读写数据的方法。通常，在代码进行 read() 调用时，代码会阻塞直至有可供读取的数据。同样， write() 调用将会阻塞直至数据能够写入。

另一方面，异步 I/O 调用不但不会阻塞，相反，您可以注册对特定 I/O 事件诸如数据可读、新连接到来等等，而在发生这样感兴趣的事件时，系统将会告诉您。

异步 I/O 的一个优势在于，它允许您同时根据大量的输入和输出执行 I/O。同步程序常常要求助于轮询，或者创建许许多多的线程以处理大量的连接。使用异步 I/O，您可以监听任何数量的通道上的事件，不用轮询，也不用额外的线程。

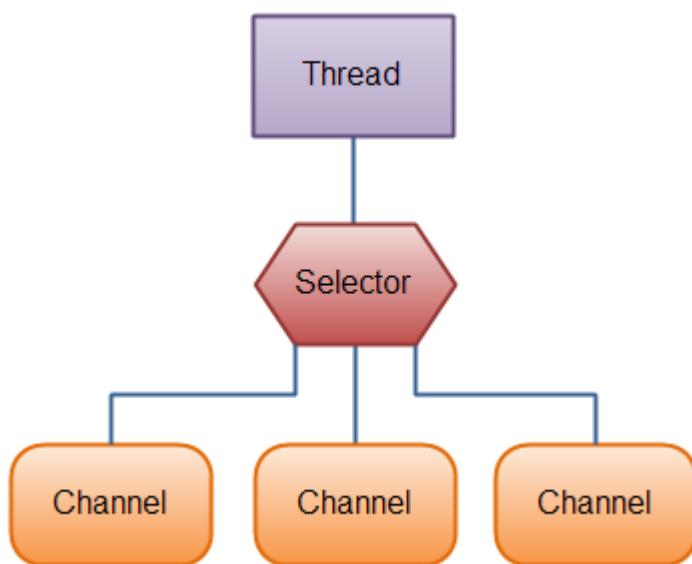
- Selector，**异步 NIO 的核心对象**。

Selector 是一个对象，它可以注册到很多个 Channel 上，监听各个 Channel 上发生的事件，并且能够根据事件情况决定 Channel 读写。这样，通过一个线程管理多个 Channel，就可以处理大量网络连接了。

使用 Selector 的好处：

有了 Selector，我们就可以利用一个线程来处理所有的 channels。线程之间的切换对**操作系统**来说代价是很高的，并且每个线程也会占用一定的系统资源。所以，对系统来说使用的线程越少越好。

Selector 模型：



- java 怎么读入一个文件逐行读出

- 通过 BufferedReader 的 readLine()方法
 - 通过文件 byte 数组转存文件内容，然后把 byte 数组转成 string 数据，然后用回车符进行分割
- Java 同步异步 IO
<http://www.jianshu.com/p/76231686d81f>

Java 集合总结

2016 年 9 月 6 日

22:31

- hashmap 是如何实现的？怎么解决 hash 冲突？
 - 解决 Hash 冲突的方法有 2 种：再哈希法、建立一个公共溢出区、开放地址法和拉链法，HashMap 使用的是拉链法。
 - HashMap 是由数组和链表组合实现的，数组中保存的是元素是链表，每一个链表都保存相同 index 相同的 Entry ($\text{index}=\text{hashCode \% 数组长度}$)，每个链表的头结点都是保存最后一个插入链表的元素（头插法插入）
- Java.util.concurrent 包下面有哪些包？concurrenthashmap 实现原理
Concurrent 下面的包：
 - Executer，用来创建线程池，在实现 Callable 接口时，添加线程；
 - FeatureTask；
 - TimeUnit；
 - Semaphore；
 - LinkedBlockingQueue。
- concurrentHashMap 实现原理：
 - concurrentHashMap 是由 segment 数组和 HashEntry 数组实现的；
 - Segment 是一个可重入锁 reentrantLock，在 concurrentHashMap 中扮演锁的角色；
 - 每个 concurrentHashMap 只有一个 segment 数组，segment 数组结构和 HashMap 类似（数组链表结构）；
 - 每个 segment 元素都有一个 HashEntry 数组，每个 HashEntry 元素都是一个链表；

- 用 segment 数组来实现分段锁；
 - 针对不同段区的操作可以并发进行，如果需要跨段进行操作，可能需要锁定整个表，那样需要顺序获取和释放锁。
- Array 和 arraylist 的区别
 - Array 可以包含基本类型和对象，ArrayList 只能包含对象；
 - Array 大小是固定的，ArrayList 大小可以动态变化；
 - ArrayList 提供了更多的方法和特性，如 addAll(),removeAll(), iterator() 等等；
 - 对于基本类型，集合使用自动装箱的方式来减少编码工作量。但是当处理固定大小的基本数据类型时，使用集合就比较慢了。
- java 有那些集合类，里面分别有那些集合
 - Collection 接口
 - Queue 接口
 - List 接口
 - ArrayList 类
 - LinkedList 类
 - Vector 类
 - Set 接口
 - HashSet 类
 - TreeSet 类
 - LinkedHashSet 类
 - Map 接口
 - HashMap 类
 - HashTable 类
 - TreeMap 类
- HashMap 和 HashTable 的区别
 - HashTable 是线程安全的，效率低，HashMap 不是线程安全的，效率高；

HashMap 的 key-value 值都可以是 null 值 , HashTable 不行 ;
HashTable 使用 Enumeration , HashMap 使用 Iterator ;
Hash 值的使用不同 , HashTable 直接使用对象的 hashCode , 而
HashMap 是重新计算 Hash 值 , 而且用于代替求模 ;
HashTable 基于 Dictionary 类 , 而 HashMap 基于 AbstractMap
类。

- Java 线程安全集合 : Vector、Hashtable、Stack、Enumeration
- 怎么安全的访问线程不安全的集合
SynchronizedList 和 Vector 的区别 , <http://www.hollischuang.com/archives/498>
- TreeMap 总结

Java String 总结

2016 年 9 月 6 日

23:05

- StringBuffer 类没有重写 equals 方法 , 所以不能进行字符串比较
- String 是基本类型吗 , 为什么可以用 + 操作
 - 不是基本类型 , 是不可变的对象类型 ;
 - 可以使用 + 操作是因为字符串拼接操作符被重载了。会调用
StringBuilder 实现。

Java Web/数据库总结

2016 年 9 月 6 日

22:34

- 在 Java 中调用存储过程的方法
 - CallableStatement 对象为所有的 DBMS 提供了一种以标准形式调用
已储存过程的方法
-
- HttpServletRequest 类主要处理 :
 - 读取和写入 HTTP 头标
 - 取得和设置 cookies

- 取得路径信息
 - 标识 HTTP 会话
-
- JSP 内置对象：
 - request
 - response
 - session
 - out
 - page
 - application
 - exception
 - pageContext
 - Config
-
- JSP 分页代码的次序：
 - 取总记录数
 - 得到总页数
 - 再取所有记录
 - 显示本页数据
-
- WEB 开发中实现会话跟踪的技术：session、cookie、地址重写（url 重写）、隐藏表单域

Java 面试题题

2016 年 7 月 25 日

14:33

1. 什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？

Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。

Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

2. JDK 和 JRE 的区别是什么？

Java 运行时环境(JRE)是将要执行 Java 程序的 Java 虚拟机。它同时也包含了执行 applet 需要的浏览器插件。Java 开发工具包(JDK)是完整的 Java 软件开发包，包含了 JRE，编译器和其他的工具(比如：JavaDoc，Java 调试器)，可以让开发者开发、编译、执行 Java 应用程序。

3. "static"关键字是什么意思？Java 中是否可以覆盖 override)一个 private 或者是 static 的方法？

"static" 关键字表明一个成员变量或者是成员方法可以在没有所属的类的实例变量的情况下被访问。

Java 中 static 方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的。static 方法跟类的任何实例都不相关，所以概念上不适用。

4. 是否可以在 static 环境中访问非 static 变量？

static 变量在 Java 中是属于类的，它在所有的实例中的值是一样的。当类被 Java 虚拟机载入的时候，会对 static 变量进行初始化。如果你的代码尝试不用实例来访问非 static 的变量，编译器会报错，因为这些变量还没有被创建出来，还没有跟任何实例关联上。

5. Java 支持的数据类型有哪些？什么是自动拆装箱？

Java 语言支持的 8 种基本数据类型是：

byte
short
int

long
float
double
boolean
char

自动装箱是 Java 编译器在基本数据类型和对应的对象包装类型之间做的一个转化。比如：把 int 转化成 Integer , double 转化成 Double , 等等。反之就是自动拆箱。

6. Java 中的方法覆盖(Overriding)和方法重载(Overloading)是什么意思 ?

Java 中的方法重载发生在同一个类里面两个或者是多个方法的方法名相同但是参数不同的情况。与此相对，方法覆盖是说子类重新定义了父类的方法。方法覆盖必须有相同的方法名，参数列表和返回类型。覆盖者可能不会限制它所覆盖的方法的访问。

7. Java 中，什么是构造函数？什么是构造函数重载？什么是复制构造函数？

当新对象被创建的时候，构造函数会被调用。每一个类都有构造函数。在程序员没有给类提供构造函数的情况下，Java 编译器会为这个类创建一个默认的构造函数。

Java 中构造函数重载和方法重载很相似。可以为一个类创建多个构造函数。每一个构造函数必须有它自己唯一的参数列表。

Java 不支持像 C++ 中那样的复制构造函数，这个不同点是因为如果你不自己写构造函数的情况下，Java 不会创建默认的复制构造函数。

8. Java 支持多继承么？

Java 中类不支持多继承，只支持单继承（即一个类只有一个父类）。但是 java 中的接口支持多继承，即一个子接口可以有多个父接口。（接口的作用是用来扩展对象的功能，一个子接口继承多个父接口，说明子接口扩展了多个功能，当类实现接口时，类就扩展了相应的功能）。

9. 接口和抽象类的区别是什么？

Java 提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

类可以实现很多个接口，但是只能继承一个抽象类

类可以不实现抽象类和接口声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

抽象类可以在不提供接口方法实现的情况下实现接口。

Java 接口中声明的变量默认都是 final 的。抽象类可以包含非 final 的变量。

Java 接口中的成员函数默认是 public 的。抽象类的成员函数可以是 private , protected 或者是 public。

接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，但是，如果它包含 main 方法的话是可以被调用的。

也可以参考 JDK8 中抽象类和接口的区别：[JDK8 中接口可以有 default 的普通方法](#)

10. 什么是值传递和引用传递？

值传递是对基本型变量而言的,传递的是该变量的一个副本,改变副本不影响原变量.

引用传递一般是对于对象型变量而言的,传递的是该对象地址的一个副本,并不是原对象本身，所以对引用对象进行操作会同时改变原对象.

一般认为,java 内的传递都是值传递.

11. 进程和线程的区别是什么？

进程是执行着的应用程序，而线程是进程内部的一个执行序列。一个进程可以有多个线程。线程又叫做轻量级进程。

12. 创建线程有几种不同的方式？你喜欢哪一种？为什么？

有三种方式可以用来创建线程：

继承 Thread 类

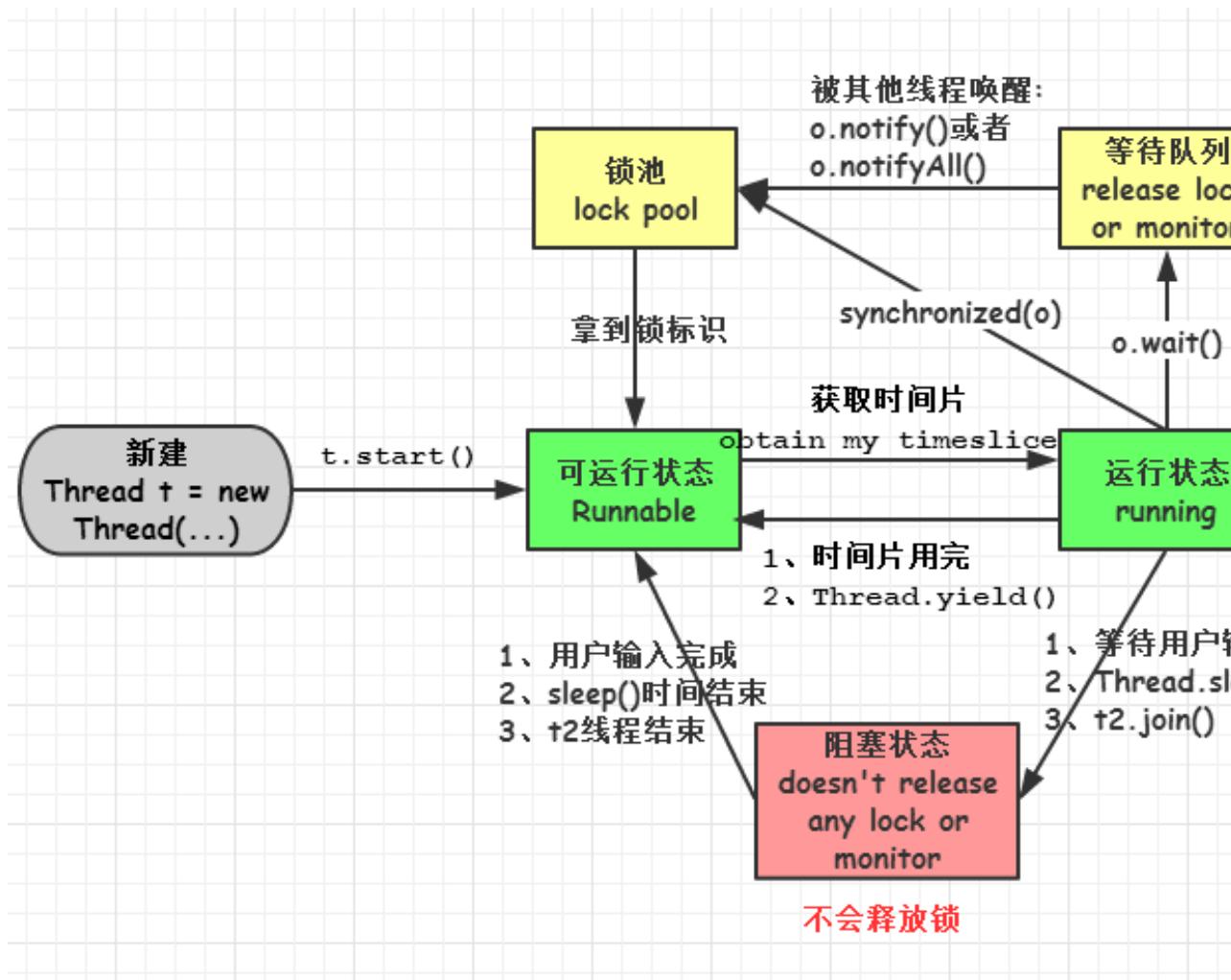
实现 Runnable 接口

应用程序可以使用 Executor 框架来创建线程池

实现 Runnable 接口这种方式更受欢迎，因为这不需要继承 Thread 类。在应用设计中已经继承了别的对象的情况下，这需要多继承（而 Java 不支持多继承），只能实现接口。同时，线程池也是非常高效的，很容易实现和使用。

13. 概括的解释下线程的几种可用状态。

1. 新建(new) : 新创建了一个线程对象。
2. 可运行(runnable) : 线程对象创建后，其他线程(比如 main 线程)调用了该对象的 start ()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取 cpu 的使用权。
3. 运行(running) : 可运行状态(runnable)的线程获得了 cpu 时间片(timeslice)，执行程序代码。
4. 阻塞(block) : 阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
 - (一). 等待阻塞 : 运行(running)的线程执行 o . wait ()方法，JVM 会把该线程放入等待队列(waiting queue)中。
 - (二). 同步阻塞 : 运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池(lock pool)中。
 - (三). 其他阻塞: 运行(running)的线程执行 Thread . sleep (long ms)或 t . join ()方法，或者发出了 I / O 请求时，JVM 会把该线程置为阻塞状态。当 sleep ()状态超时、 join ()等待线程终止或者超时、或者 I / O 处理完毕时，线程重新转入可运行(runnable)状态。
5. 死亡(dead) : 线程 run ()、 main () 方法执行结束，或者因异常退出了 run ()方法，则该线程结束生命周期。死亡的线程不可再次复生。



14. 同步方法和同步代码块的区别是什么？

区别：

同步方法默认用 this 或者当前类 class 对象作为锁；

同步代码块可以选择以什么来加锁，比同步方法要更细颗粒度，我们可以选择只同步会发生同步问题的部分代码而不是整个方法；

15. 在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步？

监视器和锁在 Java 虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码。

16. 什么是死锁(deadlock) ?

两个线程或两个以上线程都在等待对方执行完毕才能继续往下执行的时候就发生了死锁。结果就是这些线程都陷入了无限的等待中。

17. 如何确保 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。

18. Java 集合类框架的基本接口有哪些 ?

集合类接口指定了一组叫做元素的对象。集合类接口的每一种具体的实现类都可以选择以它自己的方式对元素进行保存和排序。有的集合类允许重复的键，有些不允许。

Java 集合类提供了一套设计良好的支持对一组对象进行操作的接口和类。Java 集合类里面最基本的接口有：

Collection : 代表一组对象，每一个对象都是它的子元素。

Set : 不包含重复元素的 Collection。

List : 有顺序的 collection，并且可以包含重复元素。

Map : 可以把键(key)映射到值(value)的对象，键不能重复。

19. 为什么集合类没有实现 Cloneable 和 Serializable 接口 ?

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

20. 什么是迭代器(Iterator) ?

Iterator 接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代器实例的

迭代方法。迭代器可以在迭代的过程中删除底层集合的元素,但是不可以直接调用集合的

`remove(Object Obj)`删除，可以通过迭代器的 `remove()`方法删除。

21. Iterator 和 ListIterator 的区别是什么？

下面列出了他们的区别：

Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。

Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。

ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

22. 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

Iterator 的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。

`java.util` 包下面的所有集合类都是快速失败的，而

`java.util.concurrent` 包下面的所有类都是安全失败的。快速失败的迭代器会抛出 `ConcurrentModificationException` 异常，而安全失败的迭代器永远不会抛出这样的异常。

23. Java 中的 HashMap 的工作原理是什么？

Java 中的 HashMap 是以键值对(key-value)的形式存储元素的。HashMap 需要一个 hash 函数，它使用 `hashCode()` 和 `equals()` 方法来向集合/从集合添加和检索元素。当调用 `put()` 方法的时候，HashMap 会计算 key 的 hash 值，然后把键值对存储在集合中合适的索引上。如果 key 已经存在了，value 会被更新成新值。HashMap 的一些重要的特性是它的容量(capacity)，负载因子(loadfactor)和扩容极限(threshold resizing)。

24. hashCode()和 equals()方法的重要性体现在什么地方？

Java 中的 HashMap 使用 `hashCode()` 和 `equals()` 方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确的实现这两个方法，两个不同的键可能会有相同的 hash 值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现对 HashMap 的精确性和正确性是至关重要的。

25. HashMap 和 Hashtable 有什么区别？

HashMap 和 Hashtable 都实现了 Map 接口，因此很多特性非常相似。但是，他们有以下不同点：

HashMap 允许键和值是 null，而 Hashtable 不允许键或者值是 null。

Hashtable 是同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而 Hashtable 适合于多线程环境。

HashMap 提供了可供应用迭代的键的集合，因此，HashMap 是快速失败的。另一方面，Hashtable 提供了对键的列举(Enumeration)。

一般认为 Hashtable 是一个遗留的类。

26. 数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用 Array 而不是 ArrayList？

下面列出了 Array 和 ArrayList 的不同点：

Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。

Array 大小是固定的，ArrayList 的大小是动态变化的。

ArrayList 提供了更多的方法和特性，比如：addAll()，removeAll()，iterator() 等等。

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

27. ArrayList 和 LinkedList 有什么区别？

ArrayList 和 LinkedList 都实现了 List 接口，他们有以下的不同点：

ArrayList 是基于索引的数据接口，它的底层是数组。它可以以 O(1) 时间复杂度对元素进行随机访问。与此对应，LinkedList 是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是 O(n)。

相对于 ArrayList，LinkedList 的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

LinkedList 比 ArrayList 更占内存，因为 LinkedList 为每一个节点存储了两个引用，一个指向下一个元素，一个指向下一个元素。
也可以参考 ArrayList vs. LinkedList。

28. Comparable 和 Comparator 接口是干什么的？列出它们的区别。

Java 提供了只包含一个 compareTo()方法的 Comparable 接口。这个方法可以给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。

Java 提供了包含 compare()和 equals()两个方法的 Comparator 接口。

compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和 comparator 相等。只有当输入参数也是一个 comparator 并且输入参数和当前 comparator 的排序结果是相同的时候，这个方法才返回 true。

29. 什么是 Java 优先级队列(Priority Queue)？

PriorityQueue 是一个基于优先级堆的无界队列，它的元素是按照自然顺序(natural order)排序的。在创建的时候，我们可以给它提供一个负责给元素排序的比较器。PriorityQueue 不允许 null 值，因为他们没有自然顺序，或者说他们没有任何的相关联的比较器。最后，PriorityQueue 不是线程安全的，入队和出队的时间复杂度是 $O(\log(n))$ 。

30. 你了解大 O 符号(big-O notation)么？你能给出不同数据结构的例子么？

大 O 符号描述了当数据结构里面的元素增加的时候，算法的规模或者是性能在最坏的场景下有多么好。

大 O 符号也可用来描述其他的行为，比如：内存消耗。因为集合类实际上是数据结构，我们一般使用大 O 符号基于时间，内存和性能来选择最好的实现。大 O 符号可以对大量数据的性能给出一个很好的说明。

31. 如何权衡是使用无序的数组还是有序的数组？

有序数组最大的好处在于查找的时间复杂度是 $O(\log n)$ ，而无序数组是 $O(n)$ 。

有序数组的缺点是插入操作的时间复杂度是 $O(n)$ ，因为值大的元素需要往后移动来给新元素腾位置。相反，无序数组的插入时间复杂度是常量 $O(1)$ 。

32. Java 集合类框架的最佳实践有哪些？

根据应用的需要正确选择要使用的集合的类型对性能非常重要，比如：假如元素的大小是固定的，而且能事先知道，我们就应该用 `Array` 而不是 `ArrayList`。有些集合类允许指定初始容量。因此，如果我们能估计出存储的元素的数目，我们可以设置初始容量来避免重新计算 hash 值或者是扩容。

为了类型安全，可读性和健壮性的原因总是要使用泛型。同时，使用泛型还可以避免运行时的 `ClassCastException`。

使用 JDK 提供的不变类(immutable class)作为 Map 的键可以避免为我们自己的类实现 `hashCode()` 和 `equals()` 方法。

编程的时候接口优于实现。

底层的集合实际上是空的情况下，返回长度是 0 的集合或者是数组，不要返回 `null`。

33. Enumeration 接口和 Iterator 接口的区别有哪些？

Enumeration 速度是 Iterator 的 2 倍，同时占用更少的内存。但是，Iterator 远远比 Enumeration 安全，因为其他线程不能够修改正在被 iterator 遍历的集合里面的对象。同时，Iterator 允许调用者删除底层集合里面的元素，这对 Enumeration 来说是不可能的。

34. HashSet 和 TreeSet 有什么区别？

HashSet 是由一个 hash 表来实现的，因此，它的元素是无序的。`add()`，`remove()`，`contains()` 方法的时间复杂度是 $O(1)$ 。

另一方面，TreeSet 是由一个树形的结构来实现的，它里面的元素是有序的。因此，`add()`，`remove()`，`contains()` 方法的时间复杂度是 $O(\log n)$ 。

35. Java 中垃圾回收有什么目的？什么时候进行垃圾回收？

垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

36. System.gc()和 Runtime.gc()会做什么事情？

这两个方法用来提示 JVM 要进行垃圾回收。但是，立即开始还是延迟进行垃圾回收是取决于 JVM 的。

37. finalize()方法什么时候被调用？析构函数(finalization)的目的是什么？

<div> 垃圾回收器(garbage collector)决定回收某对象时，就会运行该对象的 finalize()方法 但是在 Java 中很不幸，如果内存总是充足的，那么垃圾回收可能永远不会进行，也就是说 finalize()可能永远不被执行，显然指望它做收尾工作是靠不住的。 那么 finalize()究竟是做什么的呢？它最主要的用途是回收特殊渠道申请的内存。Java 程序有垃圾回收器，所以一般情况下内存问题不用程序员操心。但有一种 JNI(Java Native Interface)调用 non-Java 程序（C 或 C++），finalize()的工作就是回收这部分的内存。</div>

38. 如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？

不会，在下一个垃圾回收周期中，这个对象将是可被回收的。

39. Java 堆的结构是什么样子的？什么是堆中的永久代(Perm Gen space)?

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。

40. 串行(serial)收集器和吞吐量(throughput)收集器的区别是什么？

吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。而串行收集器对大多数的小应用(在现代处理器上需要大概 100M 左右的内存)就足够了。

41. 在 Java 中，对象什么时候可以被垃圾回收？

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

42. JVM 的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。请参考下 Java8：从永久代到元数据区

(注：Java8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区)

43. Java 中的两种异常类型是什么？他们有什么区别？

Java 中有两种异常：受检查的(checked)异常和不受检查的(unchecked)异常。不受检查的异常不需要在方法或者是构造函数上声明，就算方法或者是构造函数的执行可能会抛出这样的异常，并且不受检查的异常可以传播到方法或者是构造函数的外面。相反，受检查的异常必须要用 throws 语句在方法或者是构造函数上声明。这里有 Java 异常处理的一些小建议。

44. Java 中 Exception 和 Error 有什么区别？

Exception 和 Error 都是 Throwable 的子类。Exception 用于用户程序可以捕获的异常情况。Error 定义了不期望被用户程序捕获的异常。

45. throw 和 throws 有什么区别 ?

throw 关键字用来在程序中明确的抛出异常 , 相反 , throws 语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理 , 所以方法的调用者才能够确保处理可能发生的异常 , 多个异常是用逗号分隔的。

46. 异常处理完成以后 , Exception 对象会发什么变化 ?

Exception 对象会在下一个垃圾回收过程中被回收掉。

47. finally 代码块和 finalize() 方法有什么区别 ?

无论是否抛出异常 , finally 代码块都会执行 , 它主要是用来释放应用占用的资源。 finalize() 方法是 Object 类的一个 protected 方法 , 它是在对象被垃圾回收之前由 Java 虚拟机来调用的。

Java 小应用程序(Applet)

48. 什么是 Applet ?

java applet 是能够被包含在 HTML 页面中并且能被启用了 java 的客户端浏览器执行的程序。 Applet 主要用来创建动态交互的 web 应用程序。

49. 解释一下 Applet 的生命周期

applet 可以经历下面的状态 :

Init : 每次被载入的时候都会被初始化。

Start : 开始执行 applet.

Stop : 结束执行 applet.

Destroy : 卸载 applet 之前 , 做最后的清理工作。

50. 当 applet 被载入的时候会发生什么 ?

首先，创建 applet 控制类的实例，然后初始化 applet，最后开始运行。

51. Applet 和普通的 Java 应用程序有什么区别？

applet 是运行在启用了 java 的浏览器中，Java 应用程序是可以在浏览器之外运行的独立的 Java 程序。但是，它们都需要有 Java 虚拟机。

进一步来说，Java 应用程序需要一个有特定方法签名的 main 函数来开始执行。Java applet 不需要这样的函数来开始执行。

最后，Java applet 一般会使用很严格的安全策略，Java 应用一般使用比较宽松的安全策略。

52. Java applet 有哪些限制条件？

主要是由于安全的原因，给 applet 施加了以下的限制：

applet 不能够载入类库或者定义本地方法。

applet 不能在宿主机上读写文件。

applet 不能读取特定的系统属性。

applet 不能发起网络连接，除非是跟宿主机。

applet 不能够开启宿主机上其他任何的程序。

53. 什么是不受信任的 applet？

不受信任的 applet 是不能访问或是执行本地系统文件的 Java applet，默认情况下，所有下载的 applet 都是不受信任的。

54. 从网络上加载的 applet 和从本地文件系统加载的 applet 有什么区别？

当 applet 是从网络上加载的时候，applet 是由 applet 类加载器载入的，它受 applet 安全管理器的限制。

当 applet 是从客户端的本地磁盘载入的时候，applet 是由文件系统加载器载入的。

从文件系统载入的 applet 允许在客户端读文件，写文件，加载类库，并且也允许执行其他程序，但是，却通不过字节码校验。

55. applet 类加载器是什么？它会做哪些工作？

当 applet 是从网络上加载的时候，它是由 applet 类加载器载入的。类加载器有自己的 java 名称空间等级结构。类加载器会保证来自文件系统的类有唯一的名称空间，来自网络资源的类有唯一的名称空间。

当浏览器通过网络载入 applet 的时候，applet 的类被放置于和 applet 的源相关联的私有的名称空间中。然后，那些被类加载器载入进来的类都是通过了验证器验证的。验证器会检查类文件格式是否遵守 Java 语言规范，确保不会出现堆栈溢出(stack overflow)或者下溢(underflow)，传递给字节码指令的参数是正确的。

56. applet 安全管理器是什么？它会做哪些工作？

applet 安全管理器是给 applet 施加限制条件的一种机制。浏览器可以只有一个安全管理器。安全管理器在启动的时候被创建，之后不能被替换覆盖或者是扩展。

57. 弹出式选择菜单(Choice)和列表(List)有什么区别

Choice 是以一种紧凑的形式展示的，需要下拉才能看到所有的选项。Choice 中一次只能选中一个选项。List 同时可以有多个元素可见，支持选中一个或者多个元素。

58. 什么是布局管理器？

布局管理器用来在容器中组织组件。

59. 滚动条(Scrollbar)和滚动面板(JScrollPane)有什么区别？

Scrollbar 是一个组件，不是容器。而 JScrollPane 是容器。ScrollPane 自己处理滚动事件。

60. 哪些 Swing 的方法是线程安全的 ?

只有 3 个线程安全的方法 : repaint(), revalidate(), and invalidate()。

61. 说出三种支持重绘(painting)的组件。

Canvas, Frame, Panel, 和 Applet 支持重绘。

62. 什么是裁剪(clipping) ?

限制在一个给定的区域或者形状的绘图操作就做裁剪。

63. MenuItem 和 CheckboxMenuItem 的区别是什么 ?

CheckboxMenuItem 类继承自 MenuItem 类 , 支持菜单选项可以选中或者不选中。

64. 边缘布局(BorderLayout)里面的元素是如何布局的 ?

BorderLayout 里面的元素是按照容器的东西南北中进行布局的。

65. 网格包布局(GridBagLayout)里面的元素是如何布局的 ?

GridBagLayout 里面的元素是按照网格进行布局的。不同大小的元素可能会占据网格的多于 1 行或一列。因此 , 行数和列数可以有不同的大小。

66. Window 和 Frame 有什么区别 ?

Frame 类继承了 Window 类 , 它定义了一个可以有菜单栏的主应用窗口。

67. 裁剪(clipping)和重绘(repainting)有什么联系 ?

当窗口被 AWT 重绘线程进行重绘的时候，它会把裁剪区域设置成需要重绘的窗口的区域。

68. 事件监听器接口(event-listener interface)和事件适配器(event-adapter)有什么关系？

事件监听器接口定义了对特定的事件，事件处理器必须要实现的方法。事件适配器给事件监听器接口提供了默认的实现。

69. GUI 组件如何来处理它自己的事件？

GUI 组件可以处理它自己的事件，只要它实现相对应的事件监听器接口，并且把自己作为事件监听器。

70. Java 的布局管理器比传统的窗口系统有哪些优势？

Java 使用布局管理器以一种一致的方式在所有的窗口平台上摆放组件。因为布局管理器不会和组件的绝对大小和位置相绑定，所以他们能够适应跨窗口系统的特定平台的不同。

71. Java 的 Swing 组件使用了哪种设计模式？

Java 中的 Swing 组件使用了 MVC(视图-模型-控制器)设计模式。

72. 什么是 JDBC？

JDBC 是允许用户在不同数据库之间做选择的一个抽象层。JDBC 允许开发者用 JAVA 写数据库应用程序，而不需要关心底层特定数据库的细节。

73. 解释下驱动(Driver)在 JDBC 中的角色。

JDBC 驱动提供了特定厂商对 JDBC API 接口类的实现，驱动必须要提供 java.sql 包下面这些类的实现：Connection, Statement, PreparedStatement, CallableStatement, ResultSet 和 Driver。

74. Class.forName()方法有什么作用？

初始化参数指定的类，并且返回此类对应的 Class 对象

75. PreparedStatement 比 Statement 有什么优势？

PreparedStatements 是预编译的，因此，性能会更好。同时，不同的查询参数值，PreparedStatement 可以重用。

76. 什么时候使用 CallableStatement？用来准备 CallableStatement 的方法是什么？

CallableStatement 用来执行存储过程。存储过程是由数据库存储和提供的。存储过程可以接受输入参数，也可以有返回结果。非常鼓励使用存储过程，因为它提供了安全性和模块化。准备一个 CallableStatement 的方法是：
CallableStatement.prepareCall();

77. 数据库连接池是什么意思？

像打开关闭数据库连接这种和数据库的交互可能是很费时的，尤其是当客户端数量增加的时候，会消耗大量的资源，成本是非常高的。可以在应用服务器启动的时候建立很多个数据库连接并维护在一个池中。连接请求由池中的连接提供。在连接使用完毕以后，把连接归还到池中，以用于满足将来更多的请求。

78. 什么是 RMI？

Java 远程方法调用(Java RMI)是 Java API 对远程过程调用(RPC)提供的面向对象的等价形式，支持直接传输序列化的 Java 对象和分布式垃圾回收。远程方法调用可以看做是激活远程正在运行的对象上的方法的步骤。RMI 对调用者是位

置透明的，因为调用者感觉方法是执行在本地运行的对象上的。看下 RMI 的一些注意事项。

79. RMI 体系结构的基本原则是什么？

RMI 体系结构是基于一个非常重要的行为定义和行为实现相分离的原则。RMI 允许定义行为的代码和实现行为的代码相分离，并且运行在不同的 JVM 上。

80. RMI 体系结构分哪几层？

RMI 体系结构分以下几层：

存根和骨架层(Stub and Skeleton layer)：这一层对程序员是透明的，它主要负责拦截客户端发出的方法调用请求，然后把请求重定向给远程的 RMI 服务。

远程引用层(Remote Reference Layer)：RMI 体系结构的第二层用来解析客户端对服务端远程对象的引用。这一层解析并管理客户端对服务端远程对象的引用。连接是点到点的。

传输层(Transport layer)：这一层负责连接参与服务的两个 JVM。这一层是建立在网络上机器间的 TCP/IP 连接之上的。它提供了基本的连接服务，还有一些防火墙穿透策略。

81. RMI 中的远程接口(Remote Interface)扮演了什么样的角色？

远程接口用来标识哪些方法是可以被非本地虚拟机调用的接口。远程对象必须要直接或者是间接实现远程接口。实现了远程接口的类应该声明被实现的远程接口，给每一个远程对象定义构造函数，给所有远程接口的方法提供实现。

82. java.rmi.Naming 类扮演了什么样的角色？

java.rmi.Naming 类用来存储和获取在远程对象注册表里面的远程对象的引用。Naming 类的每一个方法接收一个 URL 格式的 String 对象作为它的参数。

83. RMI 的绑定(Binding)是什么意思？

绑定是为了查询找远程对象而给远程对象关联或者是注册以后会用到的名称的过程。远程对象可以使用 Naming 类的 bind()或者 rebind()方法跟名称相关联。

84. Naming 类的 bind()和 rebind()方法有什么区别？

bind()方法负责把指定名称绑定给远程对象，rebind()方法负责把指定名称重新绑定到一个新的远程对象。如果那个名称已经绑定过了，先前的绑定会被替换掉。

85. 让 RMI 程序能正确运行有哪些步骤？

为了让 RMI 程序能正确运行必须要包含以下几个步骤：

编译所有的源文件。

使用 rmic 生成 stub。

启动 rmiregistry。

启动 RMI 服务器。

运行客户端程序。

86. RMI 的 stub 扮演了什么样的角色？

远程对象的 stub 扮演了远程对象的代表或者代理的角色。调用者在本地 stub 上调用方法，它负责在远程对象上执行方法。当 stub 的方法被调用的时候，会经历以下几个步骤：

初始化到包含了远程对象的 JVM 的连接。

序列化参数到远程的 JVM。

等待方法调用和执行的结果。

反序列化返回的值或者是方法没有执行成功情况下的异常。

把值返回给调用者。

87. 什么是分布式垃圾回收(DGC)？它是如何工作的？

DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

88. RMI 中使用 RMI 安全管理器(RMISecurityManager)的目的是什么？

RMISecurityManager 使用下载好的代码提供可被 RMI 应用程序使用的安全管理器。如果没有设置安全管理器，RMI 的类加载器就不会从远程下载任何的类。

89. 解释下 Marshalling 和 demarshalling。

当应用程序希望把内存对象跨网络传递到另一台主机或者是持久化到存储的时候，就必须要把对象在内存里面的表示转化成合适的格式。这个过程就叫做 Marshalling，反之就是 demarshalling。

90. 解释下 Serialization 和 Deserialization。

Java 提供了一种叫做对象序列化的机制，它把对象表示成一连串的字节，里面包含了对象的数据，对象的类型信息，对象内部的数据的类型信息等等。因此，序列化可以看成是为了把对象存储在磁盘上或者是从磁盘上读出来并重建对象而把对象扁平化的一种方式。反序列化是把对象从扁平状态转化成活动对象的相反的步骤。

Servlet

91. 什么是 Servlet？

Servlet 是用来处理客户端请求并产生动态网页内容的 Java 类。Servlet 主要是用来处理或者是存储 HTML 表单提交的数据，产生动态内容，在无状态的 HTTP 协议下管理状态信息。

92. 说一下 Servlet 的体系结构。

所有的 Servlet 都必须要实现的核心的接口是 javax.servlet.Servlet。每一个 Servlet 都必须要直接或者是间接实现这个接口，或者是继承 javax.servlet.GenericServlet 或者 javax.servlet.http.HttpServlet。最后，Servlet 使用多线程可以并行的为多个请求服务。

93. Applet 和 Servlet 有什么区别？

Applet 是运行在客户端主机的浏览器上的客户端 Java 程序。而 Servlet 是运行在 web 服务器上的服务端的组件。applet 可以使用用户界面类，而 Servlet 没有用户界面，相反，Servlet 是等待客户端的 HTTP 请求，然后为请求产生响应。

94. GenericServlet 和 HttpServlet 有什么区别？

GenericServlet 是一个通用的协议无关的 Servlet，它实现了 Servlet 和 ServletConfig 接口。继承自 GenericServlet 的 Servlet 应该要覆盖 service() 方法。最后，为了开发一个能用在网页上服务于使用 HTTP 协议请求的 Servlet，你的 Servlet 必须要继承自 HttpServlet。这里有 Servlet 的例子。

95. 解释下 Servlet 的生命周期。

对每一个客户端的请求，Servlet 引擎载入 Servlet，调用它的 init()方法，完成 Servlet 的初始化。然后，Servlet 对象通过为每一个请求单独调用 service() 方法来处理所有随后来自客户端的请求，最后，调用 Servlet(译者注：这里应该是 Servlet 而不是 server)的 destroy()方法把 Servlet 删除掉。

96. doGet()方法和 doPost()方法有什么区别？

doGet：GET 方法会把名值对追加在请求的 URL 后面。因为 URL 对字符数目有限制，进而限制了用在客户端请求的参数值的数目。并且请求中的参数值是可见的，因此，敏感信息不能用这种方式传递。

doPOST : POST 方法通过把请求参数值放在请求体中来克服 GET 方法的限制，因此，可以发送的参数的数目是没有限制的。最后，通过 POST 请求传递的敏感信息对外部客户端是不可见的。

97. 什么是 Web 应用程序？

Web 应用程序是对 Web 或者是应用服务器的动态扩展。有两种类型的 Web 应用：面向表现的和面向服务的。面向表现的 Web 应用程序会产生包含了很多种标记语言和动态内容的交互的 web 页面作为对请求的响应。而面向服务的 Web 应用实现了 Web 服务的端点(endpoint)。一般来说，一个 Web 应用可以看成是一组安装在服务器 URL 名称空间的特定子集下面的 Servlet 的集合。

98. 什么是服务端包含(Server Side Include)？

服务端包含(SSI)是一种简单的解释型服务端脚本语言，大多数时候仅用在 Web 上，用 servlet 标签嵌入进来。SSI 最常用的场景把一个或多个文件包含到 Web 服务器的一个 Web 页面中。当浏览器访问 Web 页面的时候，Web 服务器会用对应的 servlet 产生的文本来替换 Web 页面中的 servlet 标签。

99. 什么是 Servlet 链(Servlet Chaining)？

Servlet 链是把一个 Servlet 的输出发送给另一个 Servlet 的方法。第二个 Servlet 的输出可以发送给第三个 Servlet，依次类推。链条上最后一个 Servlet 负责把响应发送给客户端。

100. 如何知道是哪一个客户端的机器正在请求你的 Servlet？

ServletRequest 类可以找出客户端机器的 IP 地址或者是主机名。
getRemoteAddr()方法获取客户端主机的 IP 地址，getRemoteHost()可以获取主机名。看下这里的例子。

101. HTTP 响应的结构是怎么样的？

HTTP 响应由三个部分组成：

状态码(Status Code)：描述了响应的状态。可以用来检查是否成功的完成了请求。请求失败的情况下，状态码可用来找出失败的原因。如果 Servlet 没有返回状态码，默认会返回成功的状态码 `HttpServletResponse.SC_OK`。

HTTP 头部(HTTP Header)：它们包含了更多关于响应的信息。比如：头部可以指定认为响应过期的过期日期，或者是指定用来给用户安全的传输实体内容的编码格式。如何在 Serlet 中检索 HTTP 的头部看这里。

主体(Body)：它包含了响应的内容。它可以包含 HTML 代码，图片，等等。主体是由传输在 HTTP 消息中紧跟在头部后面的数据字节组成的。

102. 什么是 cookie ? session 和 cookie 有什么区别？

cookie 是 Web 服务器发送给浏览器的一块信息。浏览器会在本地文件中给每一个 Web 服务器存储 cookie。以后浏览器在给特定的 Web 服务器发请求的时候，同时会发送所有为该服务器存储的 cookie。下面列出了 session 和 cookie 的区别：

无论客户端浏览器做怎么样的设置，session 都应该能正常工作。客户端可以选择禁用 cookie，但是，session 仍然是能够工作的，因为客户端无法禁用服务端的 session。

在存储的数据量方面 session 和 cookies 也是不一样的。session 能够存储任意的 Java 对象，cookie 只能存储 String 类型的对象。

103. 浏览器和 Servlet 通信使用的是什么协议？

浏览器和 Servlet 通信使用的是 HTTP 协议。

104. 什么是 HTTP 隧道？

HTTP 隧道是一种利用 HTTP 或者是 HTTPS 把多种网络协议封装起来进行通信的技术。因此，HTTP 协议扮演了一个打通用于通信的网络协议的管道的包装器的角色。把其他协议的请求掩盖成 HTTP 的请求就是 HTTP 隧道。

105. `sendRedirect()`和`forward()`方法有什么区别？

`sendRedirect()`方法会创建一个新的请求，而`forward()`方法只是把请求转发到一个新的目标上。重定向(redirect)以后，之前请求作用域范围以内的对象就失效了，因为会产生一个新的请求，而转发(forwarding)以后，之前请求作用域范围以内的对象还是能访问的。一般认为`sendRedirect()`比`forward()`要慢。

106. 什么是 URL 编码和 URL 解码？

URL 编码是负责把 URL 里面的空格和其他的特殊字符替换成对应的十六进制表示，反之就是解码。

107. 什么是 JSP 页面？

JSP 页面是一种包含了静态数据和 JSP 元素两种类型的文本的文本文档。静态数据可以用任何基于文本的格式来表示，比如：HTML 或者 XML。JSP 是一种混合了静态内容和动态产生的内容的技术。这里看下 JSP 的例子。

108. JSP 请求是如何被处理的？

浏览器首先要请求一个以.jsp 扩展名结尾的页面，发起 JSP 请求，然后，Web 服务器读取这个请求，使用 JSP 编译器把 JSP 页面转化成一个 Servlet 类。需要注意的是，只有当第一次请求页面或者是 JSP 文件发生改变的时候 JSP 文件才会被编译，然后服务器调用 servlet 类，处理浏览器的请求。一旦请求执行结束，servlet 会把响应发送给客户端。这里看下如何在 JSP 中获取请求参数。

109. JSP 有什么优点？

下面列出了使用 JSP 的优点：

JSP 页面是被动态编译成 Servlet 的，因此，开发者可以很容易的更新展现代码。

JSP 页面可以被预编译。

JSP 页面可以很容易的和静态模板结合，包括：HTML 或者 XML，也可以很容易的和产生动态内容的代码结合起来。

开发者可以提供让页面设计者以类 XML 格式来访问的自定义的 JSP 标签库。
开发者可以在组件层做逻辑上的改变，而不需要编辑单独使用了应用层逻辑的
页面。

110. 什么是 JSP 指令(Directive) ? JSP 中有哪些不同类型的指令 ?

Directive 是当 JSP 页面被编译成 Servlet 的时候，JSP 引擎要处理的指令。
Directive 用来设置页面级别的指令，从外部文件插入数据，指定自定义的标签
库。Directive 是定义在 <%@ 和 %>之间的。下面列出了不同类型的
Directive :

包含指令(Include directive) : 用来包含文件和合并文件内容到当前的页面。
页面指令(Page directive) : 用来定义 JSP 页面中特定的属性，比如错误页面和
缓冲区。
Taglib 指令 : 用来声明页面中使用的自定义的标签库。

111. 什么是 JSP 动作(JSP action) ?

JSP 动作以 XML 语法的结构来控制 Servlet 引擎的行为。当 JSP 页面被请求的
时候，JSP 动作会被执行。它们可以被动态的插入到文件中，重用 JavaBean
组件，转发用户到其他的页面，或者是给 Java 插件产生 HTML 代码。下面列
出了可用的动作：

jsp:include-当 JSP 页面被请求的时候包含一个文件。
jsp:useBean-找出或者是初始化 Javabean。
jsp:setProperty-设置 JavaBean 的属性。
jsp:getProperty-获取 JavaBean 的属性。
jsp:forward-把请求转发到新的页面。
jsp:plugin-产生特定浏览器的代码。

112. 什么是 Scriptlets ?

JSP 技术中，scriptlet 是嵌入在 JSP 页面中的一段 Java 代码。scriptlet 是位于
标签内部的所有的東西，在标签与标签之间，用户可以添加任意有效的
scriptlet。

113. 声明(Decalaration)在哪里？

声明跟 Java 中的变量声明很相似，它用来声明随后要被表达式或者 scriptlet 使用的变量。添加的声明必须要用开始和结束标签包起来。

114. 什么是表达式(Expression)？

【列表很长，可以分上、中、下发布】

JSP 表达式是 Web 服务器把脚本语言表达式的值转化成一个 String 对象，插入到返回给客户端的数据流中。表达式是在`<%=`和`%>`这两个标签之间定义的。

115. 隐含对象是什么意思？有哪些隐含对象？

JSP 隐含对象是页面中的一些 Java 对象，JSP 容器让这些 Java 对象可以为开发者所使用。开发者不用明确的声明就可以直接使用他们。JSP 隐含对象也叫做预定义变量。下面列出了 JSP 页面中的隐含对象：

`application`

`page`

`request`

`response`

`session`

`exception`

`out`

`config`

`pageContext`

116. 面向对象软件开发的优点有哪些？

代码开发模块化，更易维护和修改。

代码复用。

增强代码的可靠性和灵活性。

增加代码的可理解性。

面向对象编程有很多重要的特性，比如：封装，继承，多态和抽象。下面的章节我们会逐个分析这些特性。

117. 封装的定义和好处有哪些？

封装给对象提供了隐藏内部特性和行为的能力。对象提供一些能被其他对象访问的方法来改变它内部的数据。在 Java 当中，有 3 种修饰符：public，private 和 protected。每一种修饰符给其他的位于同一个包或者不同包下面对象赋予了不同的访问权限。

下面列出了使用封装的一些好处：

通过隐藏对象的属性来保护对象内部的状态。

提高了代码的可用性和可维护性，因为对象的行为可以被单独的改变或者是扩展。

禁止对象之间的不良交互提高模块化。

参考这个文档获取更多关于封装的细节和示例。

118. 多态的定义？

多态是编程语言给不同的底层数据类型做相同的接口展示的一种能力。一个多态类型上的操作可以应用到其他类型的值上面。

119. 继承的定义？

继承给对象提供了从基类获取字段和方法的能力。继承提供了代码的重用行，也可以在不修改类的情况下给现存的类添加新特性。

120. 抽象的定义？抽象和封装的不同点？

抽象是把想法从具体的实例中分离出来的步骤，因此，要根据他们的功能而不是实现细节来创建类。Java 支持创建只暴漏接口而不包含方法实现的抽象的类。这种抽象技术的主要目的是把类的行为和实现细节分离开。

抽象和封装是互补的概念。一方面，抽象关注对象的行为。另一方面，封装关注对象行为的细节。一般是通过隐藏对象内部状态信息做到封装，因此，封装可以看成是用来提供抽象的一种策略。

Java 设计模式

2016 年 8 月 14 日

15:05

1. 单例模式

```
public class Singleton {  
    private volatile static Singleton instance; //声明成 volatile  
    private Singleton (){}  
  
    public static Singleton getSingleton() {  
        if (instance == null) {  
            synchronized (Singleton.class) {  
                if (instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

双重检验锁模式 (double checked locking pattern) , 是一种使用同步块加锁的方法。程序员称其为双重检查锁，因为会有两次检查 `instance == null`，一次是在同步块外，一次是在同步块内。为什么在同步块内还要再检验一次？因为可能会有多个线程一起进入同步块外的 `if`，如果在同步块内不进行二次检验的话就会生成多个实例了。

`instance = new Singleton()` 这句完成的功能：

1. 给 `instance` 分配内存
 2. 调用 `Singleton` 的构造函数来初始化成员变量
 3. 将 `instance` 对象指向分配的内存空间 (执行完这步 `instance` 就为非 `null` 了)
- 这 3 步过程不能保证原子操作，在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时

instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

因此需要把 instance 声明成 volatile，利用 volatile 禁止指令重排的特点，来保证操作的原子性。

2. 简单工厂模式

3. 观察者模式

Java 文件按行读取和写入

2016 年 8 月 14 日

19:41

```
class JavaFile{
    public void readFileByLine(String fileName)
    {
        try
        {
            FileReader reader = new FileReader(fileName);
            BufferedReader br = new
            BufferedReader(reader);

            String str = null;
            while((str = br.readLine()) != null)
            {
                System.out.println(str);
            }

            br.close();
            reader.close();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    public void writeFile(String fileName)
    {
        try
        {
            FileWriter writer = new FileWriter(fileName);
```

```
        BufferedWriter bw = new  
BufferedWriter(writer);  
        bw.write("Hello World!\n");  
        bw.write("Hello Java\n");  
        bw.write("Hello Netease");  
        bw.close();  
        writer.close();  
    }  
    catch (Exception e)  
{  
        e.printStackTrace();  
    }  
}  
}
```

Java 编程笔记

2016年8月31日

20:47

1. Java 把 String 转成 char[]数组

```
char[] ch = "adfjhj".toCharArray();
```

2. 排序的两种方法

```
Arrays.sort(数组);
```

```
Collections.sort(集合);
```

3. int 数组和 String 数组相互转化

```
String str = "" + num; //产生两个 String 对象
```

```
String str = String.valueOf(num); //静态方法，只产生一个  
String 对象
```

```
int num = Integer.parseInt(str); //静态方法，只产生一个对象
```

```
int num = Integer.valueOf(str).intValue(); //会多产生一个对象
```

Java String 常用方法总结

2016年8月14日

19:43

1. char charAt(int index) 返回字符串中位于第 index 位置的字符

2.int compareTo(String other) 按照字典顺序，如果字符串位于 other 之前，返回一个负数，如果字符串位于 other 之后，返回一个正数，相等返回 0

3.boolean endsWith(String suffix) 如果字符串以 suffix 结尾，返回 true，否则返回 false

4.boolean equals(Object other) 如果字符串和 other 相等则返回 true，否则返回 false

5.boolean equalsIgnoreCase(String other) 如果字符串和 other 相等（忽略大小写）则返回 true，否则返回 false

6.int indexOf(String str) 返回与字符串 str 匹配的第一个字串的开始位置

7.int indexOf (String str,int fromIndex) 从第 fromIndex 个字符开始，返回与字符串 str 匹配的第一个字串的开始位置

8.int lastIndexOf(String str) 返回与字符串 str 匹配的最后一个字串的开始位置

9.int lastIndexOf (String str,int fromIndex) 从第 fromIndex 个字符开始，返回与字符串 str 匹配的最后一个字串的开始位置

10.int length () 返回字符串的长度

11.String replace(CharSequence oldString,CharSequence newString) 用 newString 代替原始字符串中的所有 oldString，并返回替换后的新字符串。CharSequence 可以是 String 或 StringBuilder 对象

12.boolean startsWith(string prefix) 如果字符串以 prefix 开始则返回 true，否则返回 false

13.String subString(int beginIndex) 返回一个从 beginIndex 开始到串尾的子串

14.String subString(int beginIndex , int endIndex)返回一个从 beginIndex 开始到 endIndex 的子串

15.String toLowerCase () 将字符串中的所有大写字母变为小写字母后返回新的字符串

16.String toUpperCase () 将字符串中的所有小写字母变为大写字母后返回新的字符串

17.String trim () 返回一个删除了源字符串的头部和尾部空格的新字符串

18.String concat(String str) 将 str 添加到原字符串串尾构成新的字符串并返回新的字符串

19.boolean contains (CharSequence str) 原字符串如果包含字符串 str 则返回 true , 否则返回 false

20.boolean contentEquals(CharSequence cs)判断原字符串和 cs 的字符串的内容是否相等 , 是就返回 true , 否就返回 false

21.String copyValueOf(char[]str) 复制字符数组 str 的内容到一个字符串中

22.String copyValueOf(char[]str , int offSet,int count) 复制字符数组 str 中从 offSet 开始 , 长度为 count 的内容到一个字符串中

23.String format(String format, Object...args)返回一个经过 format 格式格式化后的字符串

24.void gerChars(int srcBegin,int srcEnd,char[]dst,int dstBegin) 将字符串的第 srcBegin 位到 srcEnd 位的子串复制到字符数组 dst 中，从第 dstBegin 位置开始放置

25.boolean isEmpty() 如果为空串就返回 true , 否则返回 false

26.String join(CharSequence delimiter,CharSequence... elements)将 elements 中的各个字符串用间隔符 delimiter 连接后形成新的字符串并返回

27.boolean matches(String regex) 判断字符串是否匹配正则表达式 regex , 是就返回 true , 否就返回 false

28.String replace(char oldChar,char newChar) 使用 newChar 代替字符串中所有的 oldChar

29.String replace(CharSequence target,CharSequence replacement)使用字符串 replacement 代替原字符串中所有的 target , 而且是从第一个开始匹配的开始 , 比如用"b"代替"aa" , 那么"aaa"就变成了"ba"

30.String replaceAll(String regex,String replacement) 将字符串中所有匹配正则表达式 regex 的字串用 replacement 替换

31.String replaceFirst(String regex,String replacement) 将字符串中第一个匹配正则表达式 regex 的子串用 replacement 替换

32.String[]split(String regex) 将字符串用 regex 为标识 的间隔符分成几个子串并返回一个 String 数组

33.char[]toCharArray() 将字符串变成一个字符数组

34.String toString() 把一个 object 变成一个字符串格式

35.String valueOf(Object obj)参数可以是八种基本数据类型，返回他们的字符串格式

Java Class 的文件结构

星期一, 八月 22, 2016

9:45 下午

学习 Java 的朋友应该都知道 Java 从刚开始的时候就打着平台无关性的旗号，说“一次编写，到处运行”，其实说到无关性，Java 平台还有另外一个无关性那就是语言无关性，要实现语言无关性，那么 Java 体系中的 class 的文件结构或者说是字节码就显得相当重要了，其实 Java 从刚开始的时候就有两套规范，一个是 Java 语言规范，另外一个是 Java 虚拟机规范，Java 语言规范只是规定了 Java 语言相关的约束以及规则，而虚拟机规范则才是真正从跨平台的角度去设计的。今天我们就以一个实际的例子来看看，到底 Java 中一个 Class 文件对应的字节码应该是什么样子。这篇文章将首先总体上阐述一下 Class 到底由哪些内容构成，然后再用一个实际的 Java 类入手去分析 class 的文件结构。

在继续之前，我们首先需要明确如下几点：

1) Class 文件是有 8 个字节为基础的字节流构成的，这些字节流之间都严格按照规定的顺序排列，并且字节之间不存在任何空隙，对于超过 8 个字节的数据，将按照 Big-Endian 的顺序存储的，也就是说高位字节存储在低的地址上面，而低位字节存储到高地址上面，其实这也是 class 文件要跨平台的关键，因为 PowerPC 架构的处理采用 Big-Endian 的存储顺序，而 x86 系列的处理器则采用 Little-Endian 的存储顺序，因此为了 Class 文件在各中处理器架构下保持统一的存储顺序，虚拟机规范必须对起进行统一。

2) Class 文件结构采用类似 C 语言的结构体来存储数据的，主要有两类数据项，无符号数和表，无符号数用来表述数字，索引引用以及字符串等，比如 u1,u2,u4,u8 分别代表 1 个字节，2 个字节，4 个字节，8 个字节的无符号数，而表是多个无符号数以及其它的表组成的复合结构。可能大家看到这里对无符号数和表到底是上面也不是很清楚，不过不要紧，等下面实例的时候，我会再以实例来解释。

明确了上面的两点以后，我们接下来后来看看 Class 文件中按照严格的顺序排列的字节流都具体包含些什么数据：

```

ClassFile {
    u4           magic;
    u2           minor_version;
    u2           major_version;
    u2           constant_pool_count;
    cp_info      constant_pool[constant_pool_count-1];
    u2           access_flags;
    u2           this_class;
    u2           super_class;
    u2           interfaces_count;
    u2           interfaces[interfaces_count];
    u2           fields_count;
    field_info   fields[fields_count];
    u2           methods_count;
    method_info  methods[methods_count];
    u2           attributes_count;
    attribute_info attributes[attributes_count];
}

```

(上图来自 The Java Virtual Machine Specification Java SE 7 Edition)

在看上图的时候，有一点我们需要注意，比如 `cp_info`，`cp_info` 表示常量池，上图中用 `constant_pool[constant_pool_count-1]` 的方式来表示常量池有 `constant_pool_count-1` 个常量，它 这里是采用数组的表现形式，但是大家不要误以为所有的常量池的常量长度都是一样的，其实这个地方只是为了方便描述采用了数组的方式，但是这里并不像编程语言那里，一个 int 型的数组，每个 int 长度都一样。明确了这一点以后，我们在回过头来看看上图中每一项都具体代表了什么含义。

- 1) `u4 magic` 表示魔数，并且魔数占用了 4 个字节，魔数到底是做什么的呢？它其实就是表示一下这个文件的类型是一个 Class 文件，而不是一张 JPG 图片，或者 AVI 的电影。而 Class 文件对应的魔数是 0xCAFEBAE.
- 2) `u2 minor_version` 表示 Class 文件的次版本号，并且此版本号是 `u2` 类型的无符号数表示。
- 3) `u2 major_version` 表示 Class 文件的主版本号，并且主版本号是 `u2` 类型的无符号数表示。`major_version` 和 `minor_version` 主要用来表示当前的虚拟机是否接受当前这种版本的 Class 文件。不同版本的 Java 编译器编译的 Class 文件对应的版本是不一样的。高版本的虚拟机支持低版本的编译器编译的 Class 文件结构。比如 Java SE 6.0 对应的虚拟机支持 Java SE 5.0 的编译器编译的 Class 文件结构，反之则不行。

4) `u2 constant_pool_count` 表示常量池的数量。这里我们需要重点来说一下常量池是什么东西，请大家不要与 **Jvm** 内存模型中的运行时常量池混淆了，**Class** 文件中常量池主要存储了字面量以及符号引用，其中字面量主要包括字符串，`final` 常量的值或者某个属性的初始值等等，而符号引用主要存储类和接口的全限定名称，字段的名称以及描述符，方法的名称以及描述符，这里名称可能大家都容易理解，至于描述符的概念，放到下面说字段表以及方法表的时候再说。另外大家都知道 **Jvm** 的内存模型中有堆，栈，方法区，程序计数器构成，而方法区中又存在一块区域叫运行时常量池，运行时常量池中存放的东西其实也就是编译器生成的各种字面量以及符号引用，只不过运行时常量池具有动态性，它可以在运行的时候向其中增加其它的常量进去，最具代表性的就是 `String` 的 `intern` 方法。

5) `cp_info` 表示常量池，这里面就存在了上面说的各种各样的字面量和符号引用。放到常量池的中数据项在 **The Java Virtual Machine Specification Java SE 7 Edition** 中一共有 14 个常量，每一种常量都是一个表，并且每种常量都用一个公共的部分 `tag` 来表示是哪种类型的常量。

下面分别简单描述一下具体细节等到后面的实例中我们再细化。

- `CONSTANT_Utf8_info` `tag` 标志位为 1，UTF-8 编码的字符串
- `CONSTANT_Integer_info` `tag` 标志位为 3，整形字面量
- `CONSTANT_Float_info` `tag` 标志位为 4，浮点型字面量
- `CONSTANT_Long_info` `tag` 标志位为 5，长整形字面量
- `CONSTANT_Double_info` `tag` 标志位为 6，双精度字面量
- `CONSTANT_Class_info` `tag` 标志位为 7，类或接口的符号引用
- `CONSTANT_String_info` `tag` 标志位为 8，字符串类型的字面量
- `CONSTANT_Fieldref_info` `tag` 标志位为 9，字段的符号引用
- `CONSTANT_Methodref_info` `tag` 标志位为 10，类中方法的符号引用
- `CONSTANT_InterfaceMethodref_info` `tag` 标志位为 11，接口中方法的符号引用
- `CONSTANT_NameAndType_info` `tag` 标志位为 12，字段和方法的名称以及类型的符号引用

6) `u2 access_flags` 表示类或者接口的访问信息，具体如下图所示：

Table 4.1. Class access and property modifiers

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared <code>final</code> ; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the <code>invokespecial</code> instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared <code>abstract</code> ; must not be instantiated.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared as an annotation type.
ACC_ENUM	0x4000	Declared as an <code>enum</code> type.

- 7) u2 this_class 表示类的常量池索引，指向常量池中 `CONSTANT_Class_info` 的常量
- 8) u2 super_class 表示超类的索引，指向常量池中 `CONSTANT_Class_info` 的常量
- 9) u2 interface_counts 表示接口的数量
- 10) u2 interface[interface_counts] 表示接口表，它里面每一项都指向常量池中 `CONSTANT_Class_info` 常量
- 11) u2 fields_count 表示类的实例变量和类变量的数量
- 12) field_info fields[fields_count] 表示字段表的信息，其中字段表的结构如下图所示：

```

field_info {
    u2           access_flags;
    u2           name_index;
    u2           descriptor_index;
    u2           attributes_count;
    attribute_info attributes[attributes_count];
}

```

上图中 access_flags 表示字段的访问表示，比如字段是 public,private , protect 等，name_index 表示字段名 称，指向常量池中类型是 CONSTANT_UFT8_info 的常量，descriptor_index 表示字段的描述符，它也指向常量池中类型为 CONSTANT_UFT8_info 的常量，attributes_count 表示字段表中的属性表的数量，而属性表是则是一种用与描述字段，方法以及 类的属性的可扩展的结构，不同版本的 Java 虚拟机所支持的属性表的数量是不同的。

13) u2 methods_count 表示方法表的数量

14) method_info 表示方法表，方法表的具体结构如下图所示：

```
method_info {
    u2          access_flags;
    u2          name_index;
    u2          descriptor_index;
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

其中 access_flags 表示方法的访问表示，name_index 表示名称的索引，descriptor_index 表示方法的描述 符，attributes_count 以及 attribute_info 类似字段表中的属性表，只不过字段表和方法表中属性表中的属性是不同的，比如方法 表中就 Code 属性，表示方法的代码，而字段表中就没有 Code 属性。其中具体 Class 中到底有多少种属性，等到 Class 文件结构中的属性表的时候再 说说。

15) attribute_count 表示属性表的数量，说到属性表，我们需要明确以下几点：

- 属性表存在于 Class 文件结构的最后，字段表，方法表以及 Code 属性 中，也就是说属性表中也可以存在属性表
- 属性表的长度是不固定的，不同的属性，属性表的长度是不同的

上面说完了 Class 文件结构中每一项的构成以后，我们以一个实际的例子来解释以下上面所说的内容。

```
1      package com.ejushang.TestClass;
2
3      public class TestClass implements Super{
```

```

3
4     private static final int staticVar = 0;
5
6     private int instanceVar=0;
7
8     public int instanceMethod(int param){
9         return param+1;
10    }
11
12 }
13
14 interface Super{ }
15

```

通过 jdk1.6.0_37 的 javac 编译后的 TestClass.java 对应的 TestClass.class 的二进制结构如下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep@%...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	00	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod... (I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;	com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!.
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

下面我们就根据前面所说的 Class 的文件结构来解析以下上图中字节流。

1) 魔数

从 Class 的文件结构我们知道，刚开始的 4 个字节是魔数，上图中从地址 00000000h-00000003h 的内容就是魔数，从上图可知 Class 的文件的魔数是 0xCAFEBABE。

2) 主次版本号

接下来的 4 个字节是主次版本号，有上图可知从 00000004h-00000005h 对应的是 0x0000,因此 Class 的 minor_version 为 0x0000,从 00000006h-00000007h 对应的内容为 0x0032,因此 Class 文件的 major_version 版本为 0x0032,这正好就是 jdk1.6.0 不带 target 参数编译后的 Class 对应的主次版本。

3) 常量池的数量

接下来的 2 个字节从 00000008h-00000009h 表示常量池的数量，由上图可以知道其值为 0x0018，十进制为 24 个,但是对于常量池的数量 需要明确一点，常量池的数量是 constant_pool_count-1，为什么减一，是因为索引 0 表示 class 中的数据项不引用任何常量池中的常量。

4) 常量池

我们上面说了常量池中有不同类型的常量，下面就来看看 TestClass.class 的第一个常量，我们知道每个常量都有一个 u1 类型的 tag 标识来表示 常量的类型，上图中 0000000ah 处的内容为 0x0A，转换成二级制是 10，有上面的关于常量类型的描述可知 tag 为 10 的常量是 Constant_Methodref_info,而 Constant_Methodref_info 的结构如下图所示：

```
CONSTANT_Methodref_info {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

其中 class_index 指向常量池中类型为 CONSTANT_Class_info 的常量，从 TestClass 的二进制文件结构中可以看出 class_index 的值为 0x0004 (地址为 0000000bh-0000000ch) , 也就是说指向第四个常量。

name_and_type_index 指向常量池中类型为 CONSTANT_NameAndType_info 常量。从上图可以看出 name_and_type_index 的值为 0x0013, 表示指向常量池中的第 19 个常量。

接下来又可以通过同样的方法来找到常量池中的所有常量。不过 JDK 提供了一个方便的工具可以让我们查看常量池中所包含的常量。通过 javap -verbose TestClass 即可得到所有常量池中的常量，截图如下：

```
minor version: 0 表示从上图可知Class文件的魔数是0xCAFEBABE。
major version: 50 2 主次版本号
Constant pool:
const #1 = Method minor #4.#19; //000java/lang/Object."<init>":()V
const #2 = Field 0x0032#3.#20; //dk com/ejushang/TestClass/TestClass.instanceVar:I
const #3 = class 3 #21; // com/ejushang/TestClass/TestClass
const #4 = class 3 #22; // java/lang/Object
const #5 = class 3 #23; // com/ejushang/TestClass/Super
const #6 = Asciz staticVar;
const #7 = Asciz I;
const #8 = Asciz ConstantValue;
const #9 = int 0;
const #10 = Asciz instanceVar;
const #11 = Asciz <init>;
const #12 = Asciz ()V;
const #13 = Asciz Code;
const #14 = Asciz LineNumberTable;
const #15 = Asciz instanceMethod;
const #16 = Asciz (I)I;
const #17 = Asciz SourceFile; name_and_type_index;
const #18 = Asciz TestClass.java;
const #19 = NameAndType #11:#12; // "<init>":()V
const #20 = NameAndType #10:#7; // instanceVar:I
const #21 = Asciz com/ejushang/TestClass/TestClass;
const #22 = Asciz 0x0004java/lang/Object;
const #23 = Asciz 其他 CONSTANT_Class_info 常量，从上图中可以看出指向第四个常量。
          为CONSTANT_NameAndType_info常量。
```

从上图我们可以清楚的看到，TestClass 中常量池有 24 个常量，不要忘记了第 0 个常量，因为第 0 个常量被用来表示 Class 中的数据项不引用任何常量池中的常量。从上面的分析中我们得知 TestClass 的第一个常量表示方法，其中 class_index 指向的第四个常量为 java/lang/Object，name_and_type_index 指向的第 19 个常量值为<init>:()V,从这里可以看出第一个表示方法的常量表示的是 java 编译器生成的实例构造器方法。通过同样的方法可以分析常量池的其它常量。OK，分析完常量池，我们接下来再分析下 access_flags。

5) u2 access_flags 表示类或者接口方面的访问信息，比如 Class 表示的是类还是接口，是否为 public,static , final 等。具体访问标示的含义之前已经说过了，下面我们就来看看 TestClass 的访问标示。Class 的访问标示是从 0000010dh-0000010e，期值为 0x0021，根据前面说的 各种访问标示的标志位，我们可以知道：0x0021=0x0001|0x0020 也即 ACC_PUBLIC 和 ACC_SUPER 为真，其中 ACC_PUBLIC 大家好理解，ACC_SUPER 是 jdk1.2 之后编译的类都会带有的标志。

6) u2 this_class 表示类的索引值，用来表示类的全限定名称，类的索引值如下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep@4...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...((I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super.!.
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*.*.*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

从上图可以清楚到看到，类索引值为 0x0003，对应常量池的第三个常量，通过javap 的结果，我们知道第三个常量为 CONSTANT_Class_info 类型的常量，通过它可以知道类的全限定名称为：com/ejushang/TestClass /TestClass

7) u2 super_class 表示当前类的父类的索引值，索引值所指向的常量池中类型为 CONSTANT_Class_info 的常量，父类的索引值如下图所示，其值为 0x0004，查看常量池的第四个常量，可知 TestClass 的父类的全限定名称为：java/lang/Object

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Éþø¾...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

8) **interfaces_count** 和 **interfaces[interfaces_count]** 表示接口数量以及具体的每一个接口，TestClass 的接口数量以及接口如下图所示，其中 0x0001 表示接口数量为 1，而 0x0005 表示接口在常量池的索引值，找到常量池的第五个常量，其类型为 CONSTANT_Class_info，其值为：
com/ejushang/TestClass/Super

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Éþø¾...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

9) **fields_count** 和 **field_info**, **fields_count** 表示类中 **field_info** 表的数量 , 而 **field_info** 表示类的实例变量和类变量 , 这里需要注意的是 **field_info** 不包含从父类继承过来的字段 , **field_info** 的结构如下图所示 :

```
field_info {
    u2           access_flags;
    u2           name_index;
    u2           descriptor_index;
    u2           attributes_count;
    attribute_info attributes[attributes_count];
}
```

其中 **access_flags** 表示字段的访问标示 , 比如 **public,private,protected** , **static,final** 等 , **access_flags** 的取值如下图所示 :

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private ; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared protected ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static .
ACC_FINAL	0x0010	Declared final ; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared volatile ; cannot be cached.
ACC_TRANSIENT	0x0080	Declared transient ; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an enum .

其中 **name_index** 和 **descriptor_index** 都是常量池的索引值 , 分别表示字段的名称和字段的描述符 , 字段的名称容易理解 , 但是字段的描述符如何理解

呢？其实在 JVM 规范中，对于字段的描述符规定如下图所示：

BaseType Character	Type	Interpretation
B	<code>byte</code>	signed byte
C	<code>char</code>	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	<code>double</code>	double-precision floating-point value
F	<code>float</code>	single-precision floating-point value
I	<code>int</code>	integer
J	<code>long</code>	long integer
L <i>Classname</i> ;	<code>reference</code>	an instance of class <i>Classname</i>
S	<code>short</code>	signed short
Z	<code>boolean</code>	<code>true</code> or <code>false</code>
[<code>reference</code>	one array dimension

其中大家需要关注一下上图最后一行，它表示的是对一维数组的描述符，对于 `String[][]` 的描述符将是 `[[Ljava/lang/String;`，而对于 `int[][]` 的描述符为 `[[I`。接下来的 `attributes_count` 以及 `attribute_info` 分别表示属性表的数量以及属性表。下面我们还是以上面的 `TestClass` 为例，来看看 `TestClass` 的字段表吧。

首先我们来看一下字段的数量，`TestClass` 的字段的数量如下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f		
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep0%...2.....	
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;	
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...	
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...	
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar	
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.	
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum	
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta	
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod... (I)I	
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...	
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..	
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;	
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas	
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja	
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..	
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te	
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..	
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;	
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;	
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;	
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.mu	
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	00	0A	00	02	; ..±.....	
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;	
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;	
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;	

从上图中可以看出 TestClass 有两个字段，查看 TestClass 的源代码可知，确实也只有两个字段，接下来我们看看第一个字段，我们知道第一个字段应该为 private int staticVar, 它在 Class 文件中的二进制表示如下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f		
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep0%...2.....	
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;	
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...	
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...	
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar	
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.	
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum	
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta	
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod... (I)I	
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...	
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..	
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;	
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas	
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja	
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..	
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te	
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..	
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;	
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;	
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;	
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.mu	
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	00	0A	00	02	; ..±.....	
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;	
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;	
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;	

其中 0x001A 表示访问标示，通过查看 access_flags 表可知，其为 ACC_PRIVATE,ACC_STATIC,ACC_FINAL,接下 来 0x0006 和 0x0007 分别表示常量池中第 6 和第 7 个常量，通过查看常量池可知，其值分别为： staticVar 和 I，其中 staticVar 为字 段名称，而 I 为字段的描述符，通过上面对描述符的解释，I 所描述的是 int 类型的变量，接下来 0x0001 表示 staticVar 这个字段表中的属性表的 数量，从上图可以 staticVar 字段对应的属性表有 1 个，0x0008 表示常量池中的第 8 个常量，查看常量池可以得知此属性为 ConstantValue 属性，而 ConstantValue 属性的格式如下图所示：

```
ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}
```

其中 attribute_name_index 表述属性名的常量池索引，本例中为 ConstantValue，而 ConstantValue 的 attribute_length 固定长度为 2，而 constantvalue_index 表示常量池中的引用，本例中，其中为 0x0009，查看第 9 个 常量可以知道，它表示一个类型为 CONSTANT_Integer_info 的常量，其值为 0。

上面说完了 private static final int staticVar=0，下面我们接着说一下 TestClass 的 private int instanceVar=0,在本例中对 instanceVar 的二进制表示如下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep23...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super.!.
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*.*.*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

其中 0x0002 表示访问标示为 ACC_PRIVATE, 0x000A 表示字段的名称，它指向常量池中的第 10 个常量，查看常量池可以知道字段名称为 instanceVar，而 0x0007 表示字段的描述符，它指向常量池中的第 7 个常量，查看常量池可以知道第 7 个常量为 I，表示类型为 instanceVar 的类型为 I，最后 0x0000 表示属性表的数量为 0.

10) methods_count 和 method_info， 其中 methods_count 表示方法的数量，而 method_info 表示的方法表，其中方法表的结构如下图所示：

```
method_info {
    u2          access_flags;
    u2          name_index;
    u2          descriptor_index;
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

从上图可以看出 method_info 和 field_info 的结构是很类似的，方法表的 access_flag 的所有标志位以及取值如下图所示：

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared <code>public</code> ; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared <code>private</code> ; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared <code>protected</code> ; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared <code>static</code> .
ACC_FINAL	0x0010	Declared <code>final</code> ; never directly assigned to after object construction (JLS §17.5).
ACC_VOLATILE	0x0040	Declared <code>volatile</code> ; cannot be cached.
ACC_TRANSIENT	0x0080	Declared <code>transient</code> ; not written or read by a persistent object manager.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared as an element of an <code>enum</code> .

其中 `name_index` 和 `descriptor_index` 表示的是方法的名称和描述符，他们分别是指向常量池的索引。这里需要解释一下方法的描述符，方法的描述符的结构为：(参数列表) 返回值，比如 `public int instanceMethod(int param)` 的描述符为：`(I)I`，表示带有一个 `int` 类型参数且返回值也为 `int` 类型的方法，接下来就是属性数量以及属性表了，方法表和字段表虽然都有属性数量和属性表，但是他们里面所包含的属性是不同。接下来我们就以 `TestClass` 来看一下方法表的二进制表示。首先来看一下方法表数量，截图如下：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ép...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;	com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	00	0A	2A	B7	00	01	2A	03	B5
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	00	0A	00	02	00
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	00	04	1B	04	60
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

从上图可以看出方法表的数量为 0x0002 表示有两个方法，接下来我们来分析第一个方法，我们首先来看一下 TestClass 的第一个方法的 access_flag , name_index,descriptor_index , 截图如下：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ép...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;	com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	00	0A	2A	B7	00	01	2A	03	B5
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	00	0A	00	02	00
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	00	04	1B	04	60
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

从上图可以知道 access_flags 为 0x0001，从上面对 access_flags 标志位的描述，可知方法的 access_flags 的取值为 ACC_PUBLIC, name_index 为 0x000B，查看常量池中的第 11 个常量，知道方法的名称为<init>，0x000C 表示 descriptor_index 表示常量池中的第 12 常量，其值为()V, 表示<init>方法没有参数和返回值，其实这是编译器自动生成的实例构造器方法。接下来的 0x0001 表示<init>方法的方法表有 1 个属性，属性截图如下：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...((I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	00	07	00
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	2F	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*.*.*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

从上图可以看出 0x000D 对应的常量池中的常量为 Code, 表示的方法的 Code 属性，所以到这里大家应该明白方法的那些代码是存储在 Class 文件方法表中的属性表中的 Code 属性中。接下来我们在分析一下 Code 属性，Code 属性的结构如下图所示：

```

Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    {
        u2 start_pc;
        u2 end_pc;
        u2 handler_pc;
        u2 catch_type;
    } exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}

```

其中 `attribute_name_index` 指向常量池中值为 `Code` 的常量，`attribute_length` 的长度表示 `Code` 属性表的长度（这里需要注意的时候长度不包括 `attribute_name_index` 和 `attribute_length` 的 6 个字节的长度）。

`max_stack` 表示最大栈深度，虚拟机在运行时根据这个值来分配栈帧中操作数的深度，而 `max_locals` 代表了局部变量表的存储空间。

`max_locals` 的单位为 `slot`, `slot` 是虚拟机为局部变量分配内存的最小单元，在运行时，对于不超过 32 位类型的数据类型，比如 `byte,char,int` 等占用 1 个 `slot`，而 `double` 和 `Long` 这种 64 位的数据类型则需要分配 2 个 `slot`，另外 `max_locals` 的值并不是所有局部变量所需要的内存数量之和，因为 `slot` 是可以重用的，当局部变量超过了它的作用域以后，局部变量所占用的 `slot` 就会被重用。

`code_length` 代表了字节码指令的数量，而 `code` 表示的时候字节码指令，从上图可以知道 `code` 的类型为 `u1`,一个 `u1` 类型的取值为 `0x00-0xFF`,对应的十进制为 0-255，目前虚拟机规范已经定义了 200 多条指令。

`exception_table_length` 以及 `exception_table` 分别代表方法对应的异常信息。

`attributes_count` 和 `attribute_info` 分别表示了 `Code` 属性中的属性数量和属性表，从这里可以看出 `Class` 的文件结构中，属性表是很灵活的，它可以存在于 `Class` 文件，方法表，字段表以及 `Code` 属性中。

接下来我们继续以上面的例子来分析一下，从上面 `init` 方法的 `Code` 属性的截图中可以看出，属性表的长度为 `0x00000026`,`max_stack` 的 值为

0x0002,max_locals 的取值为 0x0001,code_length 的长度为 0x0000000A ,那么 00000149h- 00000152h 为字节码 , 接下来 exception_table_length 的长度为 0x0000 , 而 attribute_count 的值为 0x0001 , 00000157h-00000158h 的值为 0x000E, 它表示常量池中属性的名称 , 查看常量池得知第 14 个常量的值为 LineNumberTable , LineNumberTable 用于描述 java 源代码的行号和字节码行号的对应关系 , 它不是运行时必需的属性 , 如果通过 -g:none 的编译器参数来取消生成这项信息的话 , 最大的影响就是异常发生的时候 , 堆栈中不能显示出出错的行号 , 调试的时候也不能按照源代码来设置断点 , 接下来我们再看一下 LineNumberTable 的结构如下图所示 :

```
LineNumberTable_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    {
        u2 start_pc;
        u2 line_number;
    } line_number_table[line_number_table_length];
}
```

其中 attribute_name_index 上面已经提到过 , 表示常量池的索引 , attribute_length 表示属性长度 , 而 start_pc 和 line_number 分别表示字节码的行号和源代码的行号。本例中 LineNumberTable 属性的字节流如下图所示 :

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Épø¾...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super.!.
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*.*.*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

上面分析完了 TestClass 的第一个方法，通过同样的方式我们可以分析出 TestClass 的第二个方法，截图如下：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Épø¾...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...{I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super.!.
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*.*.*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

其中 access_flags 为 0x0001, name_index 为 0x000F, descriptor_index 为 0x0010，通过查看常量池可以知道此方法为 public int instanceMethod(int param)方法。通过和上面类似的方法我们可以知道 instanceMethod 的 Code 属性为下图所示：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
00000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Éþo¾...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod...((I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	;
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!.!
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

最后我们来分析一下，Class 文件的属性，从 00000191h-00000199h 为 Class 文件中的属性表，其中 0x0011 表示属性的名称，查看常量池可以知道属性名称为 SourceFile，我们再来看看 SourceFile 的结构如下图所示：

```
SourceFile_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

其中 attribute_length 为属性的长度，sourcefile_index 指向常量池中值为源代码文件名称的常量，在本例中 SourceFile 属性截图如下：

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
000000000h:	CA	FE	BA	BE	00	00	00	32	00	18	0A	00	04	00	13	09	; Ep24...2.....
00000010h:	00	03	00	14	07	00	15	07	00	16	07	00	17	01	00	09	;
00000020h:	73	74	61	74	69	63	56	61	72	01	00	01	49	01	00	0D	; staticVar...I...
00000030h:	43	6F	6E	73	74	61	6E	74	56	61	6C	75	65	03	00	00	; ConstantValue...
00000040h:	00	00	01	00	0B	69	6E	73	74	61	6E	63	65	56	61	72	;instanceVar
00000050h:	01	00	06	3C	69	6E	69	74	3E	01	00	03	28	29	56	01	; ...<init>...()V.
00000060h:	00	04	43	6F	64	65	01	00	0F	4C	69	6E	65	4E	75	6D	; ..Code...LineNum
00000070h:	62	65	72	54	61	62	6C	65	01	00	0E	69	6E	73	74	61	; berTable...insta
00000080h:	6E	63	65	4D	65	74	68	6F	64	01	00	04	28	49	29	49	; nceMethod... (I)I
00000090h:	01	00	0A	53	6F	75	72	63	65	46	69	6C	65	01	00	0E	; ...SourceFile...
000000a0h:	54	65	73	74	43	6C	61	73	73	2E	6A	61	76	61	0C	00	; TestClass.java..
000000b0h:	0B	00	0C	0C	00	0A	00	07	01	00	20	63	6F	6D	2F	65	; com/e
000000c0h:	6A	75	73	68	61	6E	67	2F	54	65	73	74	43	6C	61	73	; jushang/TestClas
000000d0h:	73	2F	54	65	73	74	43	6C	61	73	73	01	00	10	6A	61	; s/TestClass...ja
000000e0h:	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63	74	01	00	; va/lang/Object..
000000f0h:	1C	63	6F	6D	2F	65	6A	75	73	68	61	6E	67	2F	54	65	; .com/ejushang/Te
00000100h:	73	74	43	6C	61	73	73	2F	53	75	70	65	72	00	21	00	; stClass/Super!..
00000110h:	03	00	04	00	01	00	05	00	02	00	1A	00	06	00	07	00	;
00000120h:	01	00	08	00	00	00	02	00	09	00	02	00	0A	00	07	00	;
00000130h:	00	00	02	00	01	00	0B	00	0C	00	01	00	0D	00	00	00	;
00000140h:	26	00	02	00	01	00	00	00	0A	2A	B7	00	01	2A	03	B5	; &.....*...*.μ
00000150h:	00	02	B1	00	00	00	01	00	0E	00	00	00	0A	00	02	00	; ..±.....
00000160h:	00	00	03	00	04	00	07	00	01	00	0F	00	10	00	01	00	;
00000170h:	0D	00	00	00	1C	00	02	00	02	00	00	00	04	1B	04	60	;
00000180h:	AC	00	00	00	01	00	0E	00	00	00	06	00	01	00	00	00	;
00000190h:	0A	00	01	00	11	00	00	00	02	00	12						;

其中 attribute_length 为 0x00000002 表示长度为 2 个字节 , 而 sourcefile_index 的值为 0x0012, 查看常量池的第 18 个常量可以知道源代码文件的名称为 TestClass.java

Java 常用集合类

2016 年 7 月 21 日

19:31

```
/***
 * Created by LiuYikang on 2016/7/19.
 */
import java.math.*;
import java.util.*;

class testListNode
{
    int val;
    testListNode next;

    testListNode(int a)
    {
        val = a;
        next = null;
    }
}
```

```
    }

}

public class Test {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        while (in.hasNext()) {
            int a = in.nextInt();
            float b = in.nextFloat();
            byte c = in.nextByte();
            BigInteger d = in.nextBigInteger();
            String sLine = in.nextLine();
            String str = in.next();
        }
    }

    public void testVector() {
        Vector<testListNode> test = new Vector<>(10);
        testListNode tmp = new testListNode(0);

        Iterator it = test.iterator();
        while(it.hasNext())
        {
            it.next();
        }

        //在此向量的指定位置插入指定的元素。
        test.add(0, tmp);
        //将指定元素追加到此向量的末尾。
        test.add(tmp);
        //返回此向量的当前容量。
        test.capacity();
        //返回当前的大小
        test.size();
        //从此向量中移除所有元素。
        test.clear();
        //测试指定的对象是否为此向量中的组件。
        test.contains(tmp);
        //返回向量的一个副本。
        test.clone();
        //返回指定索引处的组件。
        testListNode node1 = test.elementAt(0);
        //返回此向量的组件的枚举。
    }
}
```

```
Enumeration<testListNode> e = test.elements();
//比较指定对象与此向量的相等性。
test.equals(test);
//返回此向量的第一个组件（位于索引 0 处的项）。
testListNode node = test.firstElement();
//返回向量中指定位置的元素。
testListNode node2 = test.get(0);
//搜索给定参数的第一个匹配项，使用 equals 方法测试相等性。
int idx = test.indexOf(tmp);
//判断是否为空
test.isEmpty();
//将指定对象作为此向量中的组件插入到指定的 index 处。
test.insertElementAt(tmp, 0);
//返回此向量的最后一个组件。
test.lastElement();
//返回指定的对象在此向量中最后一个匹配项的索引。
int lastIdx = test.lastIndexOf(tmp);
//移除此向量中指定位置的元素。
test.remove(0);
//移除此向量中指定元素的第一个匹配项，如果向量不包含该元素，则
元素保持不变。
test.remove(tmp);
//从此向量中移除全部组件，并将其大小设置为零。
test.removeAllElements();
//删除指定索引处的组件。
test.removeElementAt(0);
//用指定的元素替换此向量中指定位置处的元素。
test.set(0, tmp);
//设置此向量的大小。
test.setSize(10);
}

public void testStack()
{
    Stack<testListNode> s = new Stack<>();
    testListNode tmp = new testListNode(0);
    s.push(tmp);
    s.peek();
    s.isEmpty();
    s.clear();
    s.pop();
}
```

```
//返回对象在堆栈中的位置，以 1 为基数。  
int idx = s.search(tmp);  
}  
  
public void testQueue()  
{  
    Queue<testListNode> q = new  
    LinkedList<testListNode>();  
    testListNode tmp = new testListNode(0);  
    //添加一个元素并返回 true  
    q.offer(tmp);  
    q.offer(tmp);  
  
    //返回第一个元素，并在队列中删除  
    testListNode a = q.poll();  
    //返回第一个元素  
    a = q.element();  
    //返回第一个元素  
    a = q.peek();  
    //移除并返回队列头部的元素  
    a = q.remove();  
  
    //获取大小  
    q.size();  
    q.clear();  
    q.isEmpty();  
}  
  
public void testDequeue()  
{  
    Deque<testListNode> dq = new  
    LinkedList<testListNode>();  
    testListNode tmp = new testListNode(0);  
  
    //插入到头部  
    dq.addFirst(tmp);  
    dq.addFirst(tmp);  
  
    //插入到末尾  
    dq.offerLast(tmp);  
    dq.addLast(tmp);
```

```
//返回头部元素，不删除该元素
dq.getFirst();
dq.peekFirst();

//返回末尾元素，不删除该元素
dq.getLast();
dq.peekLast();

//返回头部元素并且删除该元素
dq.pollFirst();
dq.removeFirst();

//返回尾部元素并且删除该元素
dq.removeLast();
dq.pollLast();

//获取大小
dq.size();

dq.clear();
dq.isEmpty();
}

public void testPriorityQueue()
{
    //默认是小堆
    Queue<Integer> pq = new PriorityQueue<Integer>();
    pq.add(1);
    pq.add(2);
    pq.add(3);
    pq.size();
    pq.isEmpty();
    pq.peek();
    pq.clear();

    //大堆需要实现 Comparator 接口
    Comparator<Integer> cmp = new Comparator<Integer>()
    {
        @Override
        public int compare(Integer o1, Integer o2) {
            return o2.compareTo(o1);
        }
    };
}
```

```
        }
    };
    Queue<Integer> bigPQ = new PriorityQueue<Integer>(10, cmp);
}

public void testSort()
{
    //测试基本数据的排序
    int a[] = {2,4,5,3,2,5,4,3};
    Arrays.sort(a);
    //逆序
    Integer b[] = new Integer[] {3, 4, 5};
    Arrays.sort(b, Collections.reverseOrder());

    //重载 comparator 方法
    ArrayList<testListNode> c = new ArrayList<>();
    Comparator<testListNode> cmp = new
    Comparator<testListNode>() {
        @Override
        public int compare(testListNode o1, testListNode
o2) {
            if(o1.val == o2.val) return 0;
            return o1.val > o2.val ? 1 : -1;
        }
    };
    Collections.sort(c, cmp);

}

public void testMath()
{
    Math.max(1, 2);
    Math.min(1.0,2.0);
    Math.abs(-1);
    Math.sqrt(2);
    Math.pow(2, 5);
}

public void testHashMap()
{
    HashMap<Integer, Integer> m = new HashMap<>();
    m.clear();
    m.isEmpty();
    m.containsKey(1);
    m.containsValue(2);
```

```
m.get(1);
m.put(3, 4);
m.put(3, 6); //可以直接改变 3 对应的值
m.remove(1);
}

public void testHashSet()
{
    HashSet<Integer> s = new HashSet<>();
    s.clear();
    s.isEmpty();
    s.size();
    s.add(1);
    s.add(2);
    s.contains(2);
    s.remove(1);
    Iterator<Integer> it = s.iterator();
}
}
```

Python 相关

2016 年 3 月 28 日

14:39

1. Python 如何多线程

- a. 函数式：调用 `thread` 模块中的 `start_new_thread()` 函数来产生新线程；
- b. 创建 `threading.Thread` 的子类来包装一个线程对象；

2. 浅拷贝和深拷贝

浅拷贝：创建一个新的组合对象，这个新对象与原对象共享内存中的子对象。

深拷贝：创建一个新的组合对象，同时递归地拷贝所有子对象，新的组合对象与原对象没有任何关联。虽然实际上会共享不可变的子对象，但不影响它们的相互独立性。

3. Python 函数传值是传值还是传引用？

对于不可变的对象，它看起来像 C++ 中的传值方式；对于可变对象，它看起来像 C++ 中的按引用传递。

4. python 写 OJ 的标准化输入

```
import sys
for line in sys.stdin://注意 for 处理多个 case
    a = line.split()
    print int(a[0]) + int(a[1])
```

```
while True:
```

```
    try:
        (x, y) = (int(x) for x in raw_input().split())
        print x + y
    except EOFError:
        break
```

5. Python 字典排序

python dict 按照 key 排序：

1、method 1.

```
items = dict.items()
items.sort()
for key,value in items:
    print key, value # print key,dict[key]
```

2、method 2.

```
print key, dict[key] for key in sorted(dict.keys())
```

python dict 按照 value 排序：

使用 sorted 将字典按照其 value 大小排序

```
>>> record = {'a':89, 'b':86, 'c':99, 'd':100}
>>> sorted(record.items(), key = lambda x : x[1])
[('b', 86), ('a', 89), ('c', 99), ('d', 100)]
>>> sorted(record.items(), key = lambda x : x[1], reverse = True)
```

[('d', 100), ('c', 99), ('a', 89), ('b', 86)]

sorted 第一个参数要可迭代，可以为 tuple, list

OrderedDict 是 collections 中的一个包，能够记录字典元素插入的顺序，常常和排序函数一起使用来生成一个排序的字典。

比如，比如一个无序的字典

```
d = { 'banana' :3, 'apple' :4, 'pear' :1, 'orange' :2}
```

通过排序来生成一个有序的字典，有以下几种方式

```
collections.OrderedDict(sorted(d.items(),key = lambda t:t[0]))
```

或者

```
collections.OrderedDict(sorted(d.items(),key = lambda t:t[1]))
```

或者

```
collections.OrderedDict(sorted(d.items(),key = lambda t:len(t[0])))
```

6. Python sorted 函数

```
sorted(...)
```

```
sorted(iterable, cmp=None, key=None, reverse=False) --> new  
sorted list
```

iterable：待排序的可迭代类型的容器;

cmp：用于比较的函数，比较什么由 key 决定,有默认值，迭代集合中的一项;

key：用列表元素的某个已命名的属性或函数（只有一个参数并且返回一个用于排序的值）作为关键字，有默认值，迭代集合中的一项;

reverse：排序规则. reverse = True 或者 reverse = False，有默认值。

返回值：是一个经过排序的可迭代类型，与 iterable 一样。

11.说说你对 zen of python 的理解，你有什么办法看到它

12.github 上都 fork 过哪些 python 库，列举一下你经常使用的，每个库用一句话描述下其功能

13.你调试 python 代码的方法有哪些

- 14.什么是 GIL
- 15.什么是元类(meta_class)
- 16.对比一下 dict 中 items 与 iteritems
- 17.是否遇到过 python 的模块间循环引用的问题，如何避免它
- 18.有用过 with statement 吗？它的好处是什么？
- 19.说说 decorator 的用法和它的应用场景，如果可以的话，写一个 decorator
- 20.inspect 模块有什么用
- 21.写一个类，并让它尽可能多的支持操作符
- 22.说一说你见过比较 cool 的 python 实现
- 23.python 下多线程的限制以及多进程中传递参数的方式
- 24.Python 是如何进行内存管理的？
- 25.什么是 lambda 函数？它有什么好处？
- 26.如何用 Python 输出一个 Fibonacci 数列？
- 27.介绍一下 Python 中 webbrowser 的用法？
- 28.解释一下 python 的 and-or 语法
- 29.Python 是如何进行类型转换的？
- 30.Python 如何实现单例模式？其他 23 种设计模式 python 如何实现？
- 31.如何用 Python 来进行查询和替换一个文本字符串？
- 32.如何用 Python 来发送邮件？
- 33.有没有一个工具可以帮助查找 python 的 bug 和进行静态的代码分析？
- 34.有两个序列 a,b，大小都为 n,序列元素的值任意整形数，无序；要求：通过交换 a,b 中的元素，使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小。
- 35.如何用 Python 删除一个文件？
- 36.Python 如何 copy 一个文件？
- 37.python 程序中文输出问题怎么解决？
- 38.python 代码得到列表 list 的交集与差集
- 39.写一个简单的 python socket 编程
- 40.python 如何捕获异常
- 41.在 Python 中, list, tuple, dict, set 有什么区别, 主要应用在什么样的场景?
42. 静态函数, 类函数, 成员函数的区别?
43. a=1, b=2, 不用中间变量交换 a 和 b 的值
44. 写一个函数, 输入一个字符串, 返回倒序排列的结果: 如:
string_reverse('abcdef'), 返回: 'fedcba'

45. 请用自己的算法, 按升序合并如下两个 list, 并去除重复的元素:list1 = [2, 3, 8, 4, 9, 5, 6]list2 = [5, 6, 10, 17, 11, 2]

46. 说一下以下代码片段存在的问题 from amodule import * # amodule is an exist module

```
class dummyclass(object):
    def __init__(self):
        self.is_d = True
        pass

class childdummyclass(dummyclass):
    def __init__(self, isman):
        self.isman = isman

    @classmethod
    def can_speak(self): return True

    @property
    def man(self): return self.isman

if __name__ == "__main__":
    object = new childdummyclass(True)
    print object.can_speak()
    print object.man()
    print object.is_d
```

47. 介绍一下 python 的异常处理机制和自己开发过程中的体会
48.解释一下 WSGI 和 FastCGI 的关系 ?
49.解释一下 Django 和 Tornado 的关系、差别
50.解释下 Django 使用 redis 缓存服务器
51.如何进行 Django 单元测试
52.解释下 Http 协议
53.解释下 Http 请求头和常见响应状态码
54.分别简述 OO , OOA

- 55.简述正则表达式中 ? p 的含义
- 56.Python 类中的 self 的具体含义是
- 57.请写出 python 的常用内置函数（至少 3 个），并描述它们具体含义
- 58.可以用 python 进行 POST 数据提交，可以加载什么模块来进行操作？在操作之前需要对数据进行什么操作？
- 59.说出 python 中间件 SQLAlchemy 的具体声明方式？以及模块与 MySQLdb 之间的区别？
- 60.描述出 3 中 python 常用框架，并简要描述这些框架的优缺点
- 61.reactor 是什么？有什么作用？请简要描述。
- 62.请描述 2 种不同语言间数据流转通用格式。
- 63.简述我们使用多线程编程时，锁与信号量之间的关系。
- 64.通常在 python 编写 tcp 服务时，我们使用拆、粘包的模块是什么？如何加载这个模块？
- 65.两个整数数组各有 100 亿条数据，并已经排序，保存在磁盘上，内存 10M。问：（1）如何取得交集？时间和空间效率分别是多少？Python 集合 set() 操作方法（2）如果其中一个数组只有 100 条数据，如何优化算法取得交集？时间和空间效率分别是多少？（3）用自己熟悉的语言实现第 2 个问题，要求可以正确运行；假设已经提供函数 read_elt(arry_name, index) 可以用来读取某个数组的第 index 个元素，元素个数分别用 m=100 和 n=10^10 表示。
- 66.有 100 个磁盘组成的存储系统，当有 3 个磁盘同时损坏时，才会发生数据丢失。如果 1 个磁盘的损坏率是 p，请问整个存储系统丢失数据的概率是多少？
- 67.请描述 B-Tree 插入值的过程
- 68.一个管道可以从 a 端发送字符到 b 端，只能发送 0-9 这 10 个数字，设计消息的协议，让 a 可以通知 b 任意大小的数字，并讨论这种消息协议可能发送的错误的处理能力。
- 69.假设 fd 是一个 socket，read(fd, buf, 1024) 问：可能返回哪些值？其代表什么含义？
- 70.假设网络会丢失消息，进程可能意外终止，磁盘可靠（写入数据后不会丢失）；问：如何构建一个可靠的分布式 key-value 存储系统？答题要求如下：
- 1.客户端向系统发送 1 条写入请求(例如 key=x, value=1)，系统返回'成功'，

客户端一定可以正确读取到 key=y 的值 2.在你设计的系统中，要满足上面第 1 条，并有一定对故障的容错能力。3.如果要尽可能提高写入或读写成功率，如果改进系统设计？分别会有哪些问题？

71.假设你的键盘只有以下键:ACtrl + ACtrl + CCtrl + V 这里 Ctrl+A,Ctrl+C,Ctrl+V 分别代表"全选"，"复制"，"粘贴"。如果你只能按键盘 N 次，请写一个程序可以产生最多数量的 A。也就是说输入是 N(你按键盘的次数)，输出是 M(产生的 A 的个数)。加分项：打印出中间你按下的那些键。

72.假设给你一个月的日志，格式如下：[I 130403 17:26:40] 1 200 GET /question/123 (8.8.9.9) 200.39ms[I 130403 17:26:90] 1 200 GET /topic/456 (8.8.9.9) 300.85ms[I 130403 17:26:90] 1 200 POST /answer/789 (8.8.9.9) 300.85ms...方括号中依次是：级别，日期，时间，后面依次是用户 id，返回码，访问方式，访问路径，用户 ip，响应时间日志文件名格式为：年-月-日-小时.log，如：2013-01-01-18.log，共 30*24 个文件。写个程序，算出一个用户列表和一个路径列表符合以下要求：(1).这些用户每天都会访问 (GET) /topic/**这个路径两次以上 (*代表数字) (2).这些用户每天访问 (GET) 的/topic/**路径中，至少会包含两个不同的路径 (后面的数字不一样) (3).统计出所有以上用户所访问的路径中每天都出现两次以上的路径列表

73.有两个序列 a,b，大小都为 n,序列元素的值任意整形数，无序；要求：通过交换 a,b 中的元素，使[序列 a 元素的和]与[序列 b 元素的和]之间的差最小

74.Python 语言的有哪些缺陷？

75.What are some key differences to bear in mind when coding in Python vs. Java?

76.有哪些 CPython 的替代实现？什么时候，为什么使用他们？

77. Python 是解释型的还是编译型的？

78.为什么要用函数装饰器？请举例 78.现在有一个 dict 对象 adict，里面包含了一百万个元素，查找其中的某个元素的平均需要多少次比较？一千万个元素呢？

79.现在有一个 list 对象 alist，里面的所有元素都是字符串，编写一个函数对它实现一个大小写无关的排序。

80.python 里关于“堆”这种数据结构的模块是哪个？“堆”有什么优点和缺点？举一个游戏开发中可能会用到堆的问题（不限于 python 的堆，可以是其它语言的相关实现）。

81.set 是在哪个版本成为 build-in types 的？举一个你在以往项目中用到这种数据结构的问题（不限是于 python 的 set，可以是其它语言的相关实现），并说明为什么当时选择了 set 这种数据结构。

82.有一个排好序地 list 对象 alist，查找其中是否有某元素 a（尽可能地使用标准库函数）

83.实现一个 stack。

84.编写一个简单的 ini 文件解释器。

85.现有 N 个纯文本格式的英文文件，实现一种检索方案，即做一个小搜索引擎

C++相关

2016 年 3 月 28 日

14:40

1. 虚表

为了支持虚函数机制，编译器为每一个拥有虚函数的类的实例创建了一个虚函数表（virtual table），这个表中有许多的槽（slot），每个槽中存放的是虚函数的地址。虚函数表解决了继承、覆盖、添加虚函数的问题，保证其真实反应实际的函数。

为了能够找到 virtual table，编译器在每个拥有虚函数的类的实例中插入了一个成员指针 vptr，指向虚函数表。

2. const、static、 static const 区别

const 意为只读（常量）const 修饰的变量是只读的，不能被修改。但可通过指针的方式来绕过。

static 修饰的变量存在于内存中的静态区，在 main 函数执行前被加载。

const static 常量存在于内存的常量区，有操作系统加载程序时，加载到内存的常量区。所以可以对其取址，但是不能对该区的内存进行写操作，

因为这个区从操作系统级进行了只读限定，任何对该内存区的写操作会导致程序崩溃。

static const 作为预编译声明使用时，相当于 C 的#define(替换，它们本质不同)，但是需要编译器进行类型检查，保证了程序的健壮性。此时其只能用基本数据类型。static const 作为类的常量属性，因为常量区是在程序真正执行代码前进行初始化，所以其也必须是基本数据类型。

static const 和 static 的区别是，static 存在于静态区，该区可以进行写操作，其初始化的值也会存在常量区，程序启动后由操作系统从常量区取值赋值给 static 变量。

1. static 类型

1. 用 static 可以为类类型的所有对象所共有，像是全局对象，但又被约束在类类型的名字空间中。static 定义的静态常量在函数执行后不会释放其存储空间。
2. 可以实施封装，将其放在 private 或 protected 区域中。
3. static 成员没有 this 指针，它不是任何一个对象的组成部分，推荐用“类名::static 成员名调用”。
4. static 成员函数声明时应写明 static 关键字，在定义时不能加 static 关键字。
5. static 数据成员声明时应写明 static 关键字，在定义时不能加 static 关键字。
6. static 成员函数不能使用 const 以修饰其不改变成员属性。
7. static 成员函数不能使用 virtual 以修饰其虚拟性。
8. static const 数据成员可以在类中声明并且初始化，然后在类定义之外再次进行定义；或者在类中声明，但在类定义外进行定义。非 const 的 static 数据成员仅能在类中声明，并在类定义之外进行定义。
9. static 成员函数主要目的是作为类作用域的全局函数。不能访问类的非静态数据成员。类的静态成员函数没有 this 指针，这导

致：1、不能直接存取类的非静态成员变量，调用非静态成员函数
数 2、不能被声明为 virtual

2. const 类型

1. const 定义的常量在超出其作用域之后其空间会被释放，const 数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其 const 数据成员的值可以不同。所以不能在类的声明中初始化 const 数据成员，因为类的对象没被创建时，编译器不知道 const 数据成员的值是什么。
2. const 数据成员的初始化只能在类的构造函数的初始化列表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现，或者 static const。
3. const 成员函数主要目的是防止成员函数修改对象的内容。即 const 成员函数不能修改成员变量的值，但可以访问成员变量。当方法成员函数时，该函数只能是 const 成员函数。

3. C++关键字 extern

C++名字修饰：C++编译器为了区分重载函数，会把原函数名与参数信息结合，产生一个独特的内部名字，这种技术叫名字修饰。

C 语言中没有名字修饰，也没有函数重载，因此 C++代码中有 C 代码时，需要用 extern "C" 来对 C 代码进行链接指定，告诉编译器不要对这部分代码进行名字修饰，而是生成符合 C 规则的中间符号名。

所有 C 风格的头文件都要在 extern "C" 下声明。

4. C++访问控制

类成员控制

1. public：定义为 public 的成员对普通用户、类的实现者、派生类都是可访问的。public 通常用于定义类的外部接口。

2. `protected` : 定义 `protected` 成员的目的是让派生类可以访问而禁止其他用户访问。所以类的实现者和派生类可以访问，而普通用户不能访问。
3. `private` : 定义为 `private` 的成员只能被类的实现者（成员和友元）访问。`private` 部分通常用于封装（即隐藏）类的实现细节。

继承控制：

1. `public` 继承：如果继承是公有的，则成员将遵循其原有的访问说明符。父类中的 `public`、`protected` 和 `private` 属性在子类中不发生改变。
2. `protected` 继承：比 `protected` 级别高的访问权限会变成 `protected`。即父类中的 `public` 属性在子类中变为 `protected`，父类中的 `protected` 和 `private` 属性在子类中不变。
3. `private` 继承：比 `private` 级别高的访问权限会变成 `private`。即父类中的三种访问属性在子类中都会变成 `private`。

在基类的访问属性	继承方式	在派生类中的访问属性
Private	Public	不可访问，被隐藏
Public	Public	Public
Protected	Public	protected

5. 虚函数和纯虚函数

虚函数的作用是实现多态性（Polymorphism），多态性是将接口与实现进行分离，采用共同的方法，但因个体差异而采用不同的策略。纯虚函数则是一种特殊的虚函数。虚函数联系到多态，多态联系到继承。

虚函数

在 C++ 中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通过在函数之前加上 `virtual` 关键字将其定义为虚函数（`virtual`）。一旦某个函数被声明成虚函数，则所有派生类中它都是虚函数。

纯虚函数

为了让虚函数在基类什么也不做，引进了“纯虚函数”的概念，使函数无须定义。

总结：

1. 虚函数必须实现，不实现编译器会报错。
 2. 父类和子类都有各自的虚函数版本。由多态方式在运行时动态绑定。
 3. 通过作用域运算符可以强行调用指定的虚函数版本。
 4. 纯虚函数声明如下：virtual void function()=0; 纯虚函数无需定义。
包含纯虚函数的类是抽象基类，抽象基类不能创建对象，但可以声明指向抽象基类的指针或引用。
 5. 派生类实现了纯虚函数以后，该纯虚函数在派生类中就变成了虚函数，其子类可以再对该函数进行覆盖。
 6. 析构函数通常应该是虚函数，这样就能确保在析构时调用正确的析构函数版本。
-
6. new, malloc, delete, free
new 和 malloc 的区别：
 1. new 是一个运算符，malloc()是一个库函数。
 2. new 会调用构造函数，而 malloc()不会。
 3. new 返回指定类型的指针，而 malloc()返回 void*。
 4. new 会自动计算需要分配的空间，而 malloc()需要手工计算字节数。
 5. new 可以被重载，而 malloc()不能。

new 和 malloc 的区别：

1. new 是一个运算符，malloc()是一个库函数。
2. new 会调用构造函数，而 malloc()不会。
3. new 返回指定类型的指针，而 malloc()返回 void*。
4. new 会自动计算需要分配的空间，而 malloc()需要手工计算字节数。
5. new 可以被重载，而 malloc()不能。

7. iterator 和引用的实现以及与指针的区别

Iterator 和指针的区别：

指针是一种特殊的变量，它专门用来存放另一变量的地址，而迭代器只是参考了指针的特性进行设计的一种 STL 接口。

相同点：

1. 指针和 iterator 都支持与整数进行 +, - 运算，而且其含义都是从当前位置向前或者向后移动 n 个位置
2. 指针和 iterator 都支持减法运算，指针 - 指针得到的是两个指针之间的距离，迭代器 - 迭代器得到的是两个迭代器之间的距离
3. 通过指针或者 iterator 都能够修改其指向的元素

不同点：

1. cout 操作符可以直接输出指针的值，但是对迭代器进行操作的时候会报错。通过看报错信息和头文件知道，迭代器返回的是对象引用而不是对象的值，所以 cout 只能输出迭代器使用 * 取值后的值而不能直接输出其自身。
2. 指针能指向函数而迭代器不行，迭代器只能指向容器

引用和指针的区别：

相同点：

引用和指针都是地址的概念，引用是一个内存对象的别名，指针指向一个内存对象，保存了这个对象的内存地址。

区别：

1. 引用不能为空，即不存在对空对象的引用，指针可以为空，指向空对象。
2. 引用必须初始化，指定对哪个对象的引用，指针不需要。
3. 引用初始化后不能改变，指针可以改变所指对象的值。
4. 引用访问对象是直接访问，指针访问对象是间接访问。
5. 引用的大小是所引用对象的大小，指针的大小，是指针本身大小，通常是 4 字节。
6. 引用没有 const，指针有 const

7. 引用和指针的++自增运算符意义不同。
 8. 引用不需要分配内存空间，指针需要。
8. static_cast 和 dynamic_cast 区别
 1. dynamic_cast: 运行时检查，用于多态的类型转换（upcast, downcast 和 crosscast），只能转换指针和引用。
 2. static_cast: 编译时检查，用于非多态的转换，可以转换指针及其他。
 3. 使用 dynamic_cast 转换成子类时，基类中必须有虚函数，才不会报错，否则编译失败（因为 dynamic_cast 是运行时检查类型，而这个类型信息存储在虚函数表中），使用 static_cast 转换时，即使基类没有虚函数，编译也不会报错。
 9. strlen 不计算'\0', sizeof 计算
10. debug 有断言保护，release 没有
 - assert 含义是断言，它是标准 C++ 的 cassert 头文件中定义的一个宏，用来判断一个条件表达式的值是否为 true, 如果不为 true, 程序会终止，并且报告出错误，这样就很容易将错误定位
 - 通常我们开发的程序有 2 种模式: Debug 模式和 Release 模式
 1. 在 Debug 模式下, 编译器会记录很多调试信息, 也可以加入很多测试代码, 比如加入断言 assert, 方便我们程序员测试, 以及出现 bug 时的分析解决
 2. Release 模式下, 就没有上述那些调试信息, 而且编译器也会自动优化一些代码, 这样生成的程序性能是最优的, 但是如果出现问题, 就不方便分析测试了
 11. Strcpy()和 memcpy()的区别
 1. 复制的内容不同，strcpy 只能复制字符串
 2. 复制方法不同，strcpy 以'\0'结尾，memcpy 需要指定大小
 3. 用途不同，复制其他类型（除字符串）时才用 memcpy
 12. 文字常量区不允许修改

```
//p 指针指向文字常量区，不允许修改  
//后面是字符常量，存在文字常量区  
char *p = "abcde"  
  
//q 指针指向栈区，可以被修改  
char[] q = "abcde"
```

13. Struct 与 Union 主要有以下区别:

1. struct 和 union 都是由多个不同的数据类型成员组成, 但在任何同一时刻, union 中只存放了一个被选中的成员, 而 struct 的所有成员都存在。在 struct 中, 各成员都占有自己的内存空间, 它们是同时存在的。一个 struct 变量的总长度等于所有成员长度之和。在 Union 中, 所有成员不能同时占用它的内存空间, 它们不能同时存在。Union 变量的长度等于最长的成员的长度。
2. 对于 union 的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于 struct 的不同成员赋值是互不影响的。

14. 大端 : 高字节低地址

小端 : 低字节低地址

15. 赋值操作符的左值必须是非 const 数

16. 柔性数组

```
typedef struct st_type  
{  
    int i;  
    int a[];  
} type_a;  
  
type_a *p = (type_a*)malloc(sizeof(type_a)+100*sizeof(int));
```

1. 不占用 struct 内存空间, 使用 sizeof 计算不会算进来, 所占内存空间不算在结构体变量中
2. 用于动态扩展

17. 静态联编是指在程序编译连接阶段进行联编。这种联编又称为早期联编 , 这是因为这种联编工作是在程序运行之前完成的。 动态联编是指在程序运行时进行的联编 , 这种联编又称为晚期联编。

18. C++ STL 的实现：

- 1.vector 底层数据结构为数组，支持快速随机访问
- 2.list 底层数据结构为双向链表，支持快速增删
- 3.deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问
- 4.stack 底层一般用 23 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
- 5.queue 底层一般用 23 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时
- 6.45 是适配器，而不叫容器，因为是对容器的再封装
- 7.priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现
- 8.set 底层数据结构为红黑树，有序，不重复
- 9.multiset 底层数据结构为红黑树，有序，可重复
- 10.map 底层数据结构为红黑树，有序，不重复
- 11.multimap 底层数据结构为红黑树，有序，可重复
- 12.hash_set 底层数据结构为 hash 表，无序，不重复
- 13.hash_multiset 底层数据结构为 hash 表，无序，可重复
- 14.hash_map 底层数据结构为 hash 表，无序，不重复
- 15.hash_multimap 底层数据结构为 hash 表，无序，可重复

C++常用 STL

2016 年 4 月 3 日

14:56

1. string

2. stack

top()返回栈顶元素，并不移除这个元素

empty()如果栈空返回 true , 否则 false
size () 栈的大小
void push()插入元素到栈顶
void pop()移除栈顶元素

```
#include<iostream>
#include<stack>
using namespace std;
void main()
{
    stack<char> v;
    for(int i=0;i<10;i++)
        v.push(i+97);
    cout<<v.size()<<endl;
    while(!v.empty())
    {
        cout<<v.top()<<" ";
        v.pop();
    }
}
```

3. queue

empty()判空
front()返回队头元素
pop () 删除对头元素
back()返回队尾元素
push () 在队尾加入元素
size () 大小

```
#include<iostream>
#include<queue>
using namespace std;
int main()
{
```

```

queue<int>q;
for(int i=0;i<5;i++)q.push(i);
while(!q.empty()) //or while(q.size())
{
    cout<<q.front();
    q.pop();
}
return 0;
}

```

4. map

1. 定义 map 类型的对象

```

map< int, string > s1;
s1[1] = "hehe";
s1[2] = "haha";
map< int, string > s2(s1); //s2 必须和 s1 有相同的 key value 类型
map< int, string >::iterator start = s1.begin();
map< int, string >::iterator last = s1.end();

map< int, string > s3( start, last ); //start 和 end 指向的类型必须能转换为
pair<const int, string>

```

在使用 map 关联容器的时候，key 不但必须有一个类型，还必须有一个相关的比较函数，默认情况下，标准库使用 key 自带 < 操作符来实现 key 的比较。这是 key 类型的唯一的约束。

2. map 中的类型。

```

map< K, V >::key_type //key 的类型 就是 K 的类型
map< K, V >::mapped_type //value 的类型 就是 V 的类型
map< K, V >::value_type //pair 类型，first 元素是 const 的 key 类型 ( K ) ,
second 元素是 value 类型 ( V )

```

对 map 迭代器进行解引用将会产生 pair 类型的对象，指向容器中的一个 value_type 类型的值。

```

cout << (*start).first << " " << (*start).second << endl; //对上文的代码进行解
引用得到 pair 对象。
(*start).second = "hehexixi"; //OK
(*start).first = 3; //Error: assignment of read-only member 'std::pair<const
int, std::basic_string<char> >::first

```

3.给 map 中添加元素。

1)使用下标访问 map 对象。

```

map< int, string > s1;

s1[1] = "hehe";

```

对于 `s1[1] = "hehe"`,将会发生以下行为 :

1.首先查找 key 为 1 的元素 , 没有找到 2.将一个新的 key-value 插入到 s1 中 , key 为 const int 类型 , value 为 string 类型。 value 值采取初始化 , 为空。 并将这个新的 key-value 对插入到 s1 中。 3.读取 `s1[1]`,并将其 value 赋值为"hehe".

使用 key 访问 map 的时候 , 如果不存在这个元素将会导致在 map 容器中添加一个新的元素。所关联的 value 采用值初始化。内置类型元素初始化为 0 , 类类型则采用默认构造函数初始化。

2) map::insert 的使用

1.m.insert(e) (e 为 pair 类型的值 ,如果 e.first 不再 map 中则插入一个 value 为 e.second 的元素 , 如果存在则保持不变。返回 pair 类型 , 包含指向 e.first 的 map 迭代器 , 和一个 bool 类型的对象 , 表示是否已经插入该元素。 (如果不存在则为 true , 存在则返回 false).

```

map< int, string > s1;
typedef pair< map< int, string >::iterator, bool > returnPair;
returnPair p1 = s1.insert( map < int, string >::value_type ( 1, "haha" ) );
returnPair p2 = s1.insert( map < int, string >::value_type ( 1, "haha" ) );
cout << (*p1.first).second << " " << p1.second << endl;

cout << (*p2.first).second << " " << p2.second << endl;

```

返回类型为 pair 类型，first 元素为插入 map 的迭代器类型，第二个是 bool 对象。

返回值为 haha 1 , haha 0 , 表示第一次成功插入 , 第二次没有成功插入。(已经存在)

2.m.insert(beg,end) beg 和 end 是迭代器。元素必须是 m.value_type 类型的 key- value。该操作将对于 beg 和 end 之间的所有元素插入到 m , 方式如 1 , 如果存在 , 不变 , 如果不存在 , 插入。

```
map< int, string > s1;

s1.insert( map < int, string >::value_type ( 1, "haha" ) );
s1.insert( map < int, string >::value_type ( 2, "hehe" ) );
s1.insert( map < int, string >::value_type ( 3, "xixi" ) );
map< int, string >::iterator beg = s1.begin();
map< int, string >::iterator end = s1.end();

map< int, string > s2;
s2.insert( beg, end );
cout << s2[ 1 ] << " " << s2[ 2 ] << " " << s2[ 3 ] << endl;

map< int, string > s3;
cout << s3[ 1 ] << " " << s3[ 2 ] << " " << s3[ 3 ] << endl;
```

显示值为 haha hehe xixi 和 3 个空 , 表示 s2 插入成功。

4.两种添加元素方式的区别。

我们通过上文可以得到两种插入元素的方法。采用下标和调用 insert 函数。那么他们之间的区别是什么呢 ? 哪个效率更高呢 ? 先看下代码

```
map< int, string > s3;
s3[1] = "haha";
s3[1] = "xixi";
cout << s3[1] << endl;
s3.insert( map< int, string >::value_type( 2,"haha" ) );
s3.insert( map< int, string >::value_type( 2,"xixi" ) );
```

```
cout << s3[2] << endl;
```

输出 : xixi haha 可以看出来区别 , 用下标来插入的话 , 如果已经存在 , 则更新 key 对应的 value 的值 , 如果是 insert 插入 , 则不更新。

另外 : insert 插入的开销比较小。因为下标插入如果不存在的话会首先初始化 value 的值。如果是类对象的话 , 会调用构造函数。然后在根据 value 的值进行更新。造成多余的开销。

5. count 和 find 函数来查找 map 中的元素。

用下标来判断元素是否存在是危险的 , 如果你想判断某个 key 是否存在 , 如果不存在 , 赋给它一个初始值 , 则用下标来操作是非常完美的。如果我们只想判断 map 中是否存在某个 key , 而不想插入它。则不可以使用下标操作来判断元素是否存在 , map 中提供了 2 个函数用来适应后边的这个情况。

```
m.count(key)//返回值为 0 或者 1 , 如果存在返回 1 , 如果不存在返回 0  
m.find(key)//如果存在则返回该 key 对应 pair 对象的迭代器 , 如果不存在返回 map  
的 end 迭代器。
```

```
map< int, string > s3;  
s3[1] = "haha";  
s3.insert( map< int, string >::value_type( 2,"xixi" ) );  
if( s3.count(1) )  
    cout << s3[1] << endl;  
if( s3.find(1) != s3.end() )  
    cout << ( *s3.find(1) ).second << endl;
```

这两个函数都可以用来查找 map 中是否存在某 key 对应的 pair 对象 , 区别是一个返回 1 或者 0 , 另外一个返回迭代器。

6.map 的删除操作

- 1) .m.erase(k) k 必须为 m 中的 key-value 类型中的 key 类型。
删除 m 中 key 为 m 的元素，返回 size_type 类型的值，表示删除的元素个数。0 或者 1
- 2) .m.erase(p) p 是迭代器。删除 p 所指向的元素，且 p 不等与 m.end()，返回为 void
- 3) .m.erase(b,e) b 和 e 都为迭代器，删除一段范围，且 b 和 e 都指向 m 中的元素，或者最后一个元素的下一个位置。且 b 必须在 e 前面。或者相等。

7.map 元素遍历

```
for(auto kv : strHash)
{
    ret.push_back(kv.second);
}
```

7. algorithms

```
#include <algorithm>
using namespace std;
```

1.默认的 sort 函数是按升序排。对应于 1)

```
sort(a,a+n); //两个参数分别为待排序数组的首地址和尾地址
```

2.可以自己写一个 cmp 函数，按特定意图进行排序。对应于 2)

例如：

```
int cmp( const int &a, const int &b ){
    if( a > b )
        return 1;
    else
        return 0;
}
```

```
sort(a,a+n,cmp);
```

是对数组 a 降序排序

又如：

```
int cmp( const POINT &a, const POINT &b ){
    if( a.x < b.x )
        return 1;
    else
        if( a.x == b.x ){
            if( a.y < b.y )
                return 1;
            else
                return 0;
        }
    else
        return 0;
}
```

```
sort(a,a+n,cmp);
```

是先按 x 升序排序，若 x 值相等则按 y 升序排

8. Vector

使用需要的头文件：

```
#include <vector>
```

Vector : Vector 是一个类模板。不是一种数据类型。 Vector<int>是一种数据类型。

- 1.push_back 在数组的最后添加一个数据
- 2.pop_back 去掉数组的最后一个数据
- 3.at 得到编号位置的数据
- 4.begin 得到数组头的指针
- 5.end 得到数组的最后一个单元+1 的指针
- 6. front 得到数组头的引用
- 7.back 得到数组的最后一个单元的引用
- 8.max_size 得到 vector 最大可以是多大
- 9.capacity 当前 vector 分配的大小
- 10.size 当前使用数据的大小
- 11.resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值

12.reserve	改变当前 vecotr 所分配空间的大小
13.erase	删除指针指向的数据项
14.clear	清空当前的 vector
15.rbegin	将 vector 反转后的开始指针返回(其实就是原来的 end-1)
16.rend	将 vector 反转构的结束指针返回(其实就是原来的 begin-1)
17.empty	判断 vector 是否为空
18.swap	与另一个 vector 交换数据

一、 定义和初始化

Vector<T> v1; //默认构造函数 v1 为空

Vector<T> v2(v1); //v2 是 v1 的一个副本

Vector<T> v3(n,i); //v3 包含 n 个值为 i 的元素

Vector<T> v4(n); //v4 含有 n 个值为 0 的元素

二、 值初始化

1> 如果没有指定元素初始化式，标准库自行提供一个初始化值进行值初始化。

2> 如果保存的式含有构造函数的类类型的元素，标准库使用该类型的构造函数初始化。

3> 如果保存的式没有构造函数的类类型的元素，标准库产生一个带初值的对象，使用这个对象进行值初始化。

三、 Vector 对象最重要的几种操作

1. v.push_back(t) 在数组的最后添加一个值为 t 的数据
2. v.size() 当前使用数据的大小
3. v.empty() 判断 vector 是否为空
4. v[n] 返回 v 中位置为 n 的元素
5. v1=v2 把 v1 的元素替换为 v2 元素的副本
6. v1==v2 判断 v1 与 v2 是否相等
7. !=、 <、 <=、 >、 >= 保持这些操作符惯有含义

四、 vector 的插入

std::vector<int> v; //现在容器中有 0 个元素

```
int values[] = {1,3,5,7};  
v.insert(v.end(), values+1, values+3);      //现在容器中有 2 个元素分  
别为:3,5  
  
v.push_back(9);                          //现在容器中有 3 个元素分  
别为:3,5,9  
  
v.erase(&v[1]);                          //现在容器中有 2 个元素分  
别为:3,9  
  
v.insert(v.begin()+1, 4);                //现在容器中有 3 个元素  
分别为:3,4,9  
  
v.insert(v.end()-1, 4, 6);              //现在容器中有 7 个元素  
分别为:3,4,6,6,6,6,9  
  
v.erase(v.begin()+1, v.begin()+3);    //现在容器中有 5 个元素分  
别为:3,6,6,6,9  
  
v.pop_back();                          //现在容器中有 4 个元素分  
别为:3,6,6,6  
  
v.clear();                            //现在容器中有 0 个元素
```

9. Deque

1. Constructors 创建一个新双向队列

语法:

deque(); //创建一个空双向队列

deque(size_type size); // 创建一个大小为 size 的双向
队列

deque(size_type num, const TYPE &val); //放置
num 个 val 的拷贝到队列中

deque(const deque &from);// 从 from 创建一个内容一样的双向队列

deque(input_iterator start, input_iterator end);
// start 和 end - 创建一个队列，保存从 start 到 end

的元素。

2. Operators 比较和赋值双向队列

//可以使用[]操作符访问双向队列中单个的元素

3. assign() 设置双向队列的值

语法:

```
void assign( input_iterator start, input_iterator  
end);  
//start 和 end 指示的范围为双向队列赋值  
void assign( Size num, const TYPE &val );//设置成  
num 个 val。
```

4. at() 返回指定的元素

语法:

reference at(size_type pos); 返回一个引用，指向双向队列中位置 pos 上的元素

5. back() 返回最后一个元素

语法:

reference back();//返回一个引用，指向双向队列中最后一个元素

6. begin() 返回指向第一个元素的迭代器

语法:

iterator begin();//返回一个迭代器，指向双向队列的第一个元素

7. clear() 删除所有元素

8. empty() 返回真如果双向队列为空

9. end() 返回指向尾部的迭代器

10. erase() 删除一个元素

语法:

iterator erase(iterator pos); //删除 pos 位置上的元素

iterator erase(iterator start, iterator end); //删除 start 和 end 之间的所有元素

//返回指向被删除元素的后一个元素

11.front() 返回第一个元素的引用

12.get_allocator() 返回双向队列的配置器

13.insert() 插入一个元素到双向队列中

语法:

```
iterator insert( iterator pos, size_type num, const  
TYPE &val ); //pos 前插入 num 个 val 值
```

```
void insert( iterator pos, input_iterator start,  
input_iterator end );
```

//插入从 start 到 end 范围内的元素到 pos 前面

14.max_size() 返回双向队列能容纳的最大元素个数

15.pop_back() 删除尾部的元素

16.pop_front() 删除头部的元素

17.push_back() 在尾部加入一个元素

18.push_front() 在头部加入一个元素

19.rbegin() 返回指向尾部的逆向迭代器

20.rend() 返回指向头部的逆向迭代器

21.resize() 改变双向队列的大小

22.size() 返回双向队列中元素的个数

23.swap() 和另一个双向队列交换元素

10. priority_queue

empty() 如果队列为空返回真

pop() 删除对顶元素

push() 加入一个元素

size() 返回优先队列中拥有的元素个数

`top()` 返回优先队列对顶元素

头文件：`#include <queue>`

- 11. `unordered_map`
- 12. `unordered_set`
- 13. `set`
- 14. `multiset`
- 15. `priority_queue`
- 16. `to_string()`
- 17. `pair`
- 18. `make_pair, make_heap`

C#相关

2016年3月28日

14:39

1. C#垃圾回收机制

应用程序根：

- 1. 全局对象和静态对象的引用
- 2. 应用程序代码库中局部对象的引用
- 3. 传递进一个方法的对象参数的引用
- 4. 等待被终结（`finalize`，后面介绍）对象的引用
- 5. 任何引用对象的CPU寄存器

垃圾回收的2个步骤：

- 1. 标记对象：检查托管堆中的对象有没有活动根，有活动根的对象进行标记；
- 2. 压缩托管堆：垃圾回收器会对没有标记的对象进行内存的释放

对象的“代”：

- 1. 托管堆上的每一个对象都被指定属于某个“代”；
- 2. 0代：从没有被标记为回收的新分配的对象

- 1 代：在上一次垃圾回收中没有被回收的对象
- 2 代：在一次以上的垃圾回收后仍然没有被回收的对象
- 3. “代”这个概念的基本思想就是，一个对象在托管堆上存在的时间越长，那么它就更可能应该保留。

C#可以使用 System.GC 库通过编程与垃圾回收器交互

强制垃圾回收：

- 1. 应用程序将要进入一段代码，这段代码不希望被可能的垃圾回收中断
- 2. 应用程序刚刚分配非常多的对象，程序想在使用完这些对象后尽快的回收内存空间
- 3. 调用方式：

```
GC.Collect();  
GC.WaitForPendingFinalizers();
```

2. C#如何多线程

C#使用 System.Threading.Thread 类来进行线程的控制；
new 出来新的 Thread 实例就可以生成线程；
用 start 方法执行；
Isbackground()方法可以设置为后台线程；
CLR 线程池并不会在 CLR 初始化的时候立刻建立线程，而是在应用程序要创建线程来执行任务时，线程池才初始化一个线程。线程的初始化与其他的线程一样。在完成任务以后，该线程不会自行销毁，而是以挂起的状态返回到线程池。直到应用程序再次向线程池发出请求时，线程池里挂起的线程就会再度激活执行任务。