

## 目录

一 java 基础.....	9
1.1java 的 8 种基本数据类型 装箱 拆箱.....	9
1.1.1 . 8 种基本数据类型.....	9
1.1.2 . 装箱和拆箱.....	10
1.1.3. String 转出 int 型, 判断能不能转? 如何转? .....	10
1.1.4 short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 +=1;有什么错?.....	11
1.1.5 . Int 与 Integer 区别.....	11
1.1.6 . 字节字符区别.....	12
1.1.7 java 基本类型与引用类型的区别.....	12
1.2 重写重载封装继承多态.....	12
1.3 Stack Queue.....	13
1.3.1 PriorityQueue.....	13
1.7 Concurrent 包.....	18
1.8 面向对象.....	19
1.9 String StringBuffer StringBuilder hashCode equals.....	19
String 中的 hashCode 以及 toString.....	20
1.10 java 文件读取.....	21
1.11 Java 反射.....	22
1.12 JDK NDK JRE JNI.....	23
1.13 static 和 final 的区别.....	23
1.14 map, list, set 区别.....	25
1.16 Session 和 COOKIE.....	25
1.19 IO NIO BIO AIO select epoll.....	26
1.19.1 NIO 的原理.....	28
1.20 ThreadLocal.....	30
1.22 finalize finalization finally.....	30
1.23 public private default protected.....	31
1.25 Object.....	31
1.26 equals 和 == 的区别.....	31
1.27 异常.....	32
1.28 序列化.....	34
1.30 Comparable 接口和 Comparator 接口.....	35
1.33 接口和抽象类.....	36
1.34 Socket.....	38
1.35 Runtime 类.....	38
1.36 值传递与引用传递.....	38
1.37 泛型 ? 与 T 的区别.....	38
1.38 枚举类型字节码层面理解 Enum.....	39
1.39 java 注解类型.....	40
1.40 字节流 字符流.....	41
1.41 静态内部类 匿名类.....	43
1.41.2 匿名类.....	44
二 . 集合类 Set.....	47

2.1 HashMap.....	47
(1) hashMap 的原理.....	47
(3) get.....	48
(4) HashMap 的 put 方法源码.....	48
(5) HashMap 问题 jdk1.8 优化.....	49
<b>7.Hashmap 中的 key 可以为任意对象或数据类型吗 ? .....</b>	<b>50</b>
2.2 CurrentHashMap.....	51
0. JDK1.7 ConcurrentHashMap 原理.....	51
1. JDK1.7 Get.....	52
2. JDK1.7 PUT.....	54
3. JDK1.7 Remove.....	56
4.JDK1.7 & JDK1.8 size().....	57
5.JDK 1.8.....	57
6.JDK1.8 put.....	58
7. JDK1.8 get 方法.....	60
8. rehash 过程.....	60
2.3 . Hashtable.....	60
0.参数.....	60
1.put.....	61
2.get.....	61
3.Remove.....	62
4.扩容.....	62
2.4 hashtable 和 hashmap 的区别.....	63
2.5 HashMap 和 ConcurrentHashMap 区别.....	63
2.6 ConcurrentHashMap 和 HashTable 区别.....	63
2.7 linkedHashMap.....	64
2.8 Linkedhashmap 与 hashmap 的区别.....	64
2.9 HashSet.....	64
2.10 hashmap 与 hashset 区别.....	65
2.11 Collections.sort 内部原理.....	66
2.12 hash 算法.....	67
2.13 迭代器 Iterator Enumeration.....	68
2.14 LIST ArrayList, LinkedList 和 Vector 的区别和实现原理.....	68
2.15 快速失败(fail-fast)和安全失败(fail-safe).....	70
<b>三 锁 volatile synchronized Lock ReentrantLock AQS CAS.....</b>	<b>71</b>
3.1 .volatile 和 synchronized.....	71
3.1.1 Volatile 与 synchronized 区别.....	73
3.1.2 Volatile.....	73
3.1.3 Synchronized 原理.....	73
3.2 CAS.....	76
3.3 可重入锁 ReentrantLock.....	76
1.5.3 乐观锁和悲观锁 阻塞锁, 自旋锁, 偏向锁, 轻量锁, 重量锁。公平锁 非公平锁.....	76

3.3 ReentrantLock 和 synchronized 区别.....	77
1.5.3 重入锁、对象锁、类锁的关系.....	78
四 java 多线程； .....	78
4.1 . 如何创建线程？哪种好？ .....	78
4.2 . 线程状态.....	79
4.3 . 一般线程和守护线程的区别.....	79
4.4 . sleep wait yield notify notifyAll join.....	80
4.5 中断线程.....	81
4.6 多线程如何避免死锁.....	81
4.7 多线程的好处以及问题.....	82
4.8 多线程共用一个数据变量注意什么？ .....	82
4.9 线程通信方式.....	83
4.10 线程池.....	83
4.11.线程中抛出异常怎么办.....	84
五 . Java 进阶 ssh/ssm 框架.....	85
2.1Spring.....	85
2.1.0 什么是 Spring 以及优点.....	85
2.1.1 ApplicationContext 和 beanfactory 的区别.....	85
2.1.2 Spring Bean 生命周期.....	86
2.1.3 spring 中 bean 的作用域.....	87
2.1.4 Spring IOC.....	87
2.1.5 Spring AOP.....	87
2.1.6 事务.....	89
2.1.7 Spring MVC.....	91
2.1.8 Spring 中设计模式.....	92
2.2 Servlet.....	94
2.2.1 Servlet 生命周期.....	94
2.2 Struts.....	94
2.2.1 Struts 工作流程.....	94
2.2.2 Struts 工作原理.....	94
2.2.3 do Fliter.....	95
2.2.4 拦截器与过滤器的区别.....	96
2.2.5 Struts 中为什么不用考虑线程安全.....	97
2.2.6 Struts2 和 Struts1 区别.....	97
2.3 Hibernate.....	98
2.4 Redis.....	98
2.5 Tomcat.....	98
2.6 netty.....	98
2.7 Hadoop.....	98
2.8 Volley.....	98
六 . Java 内存模型 和 垃圾回收.....	99
3.0 什么是 JMM 内存模型？ (JMM 和内存区域划分不是一回事) .....	99
3.0.1 JMM 中的 happens-before 原则.....	100
3.1 内存分区.....	101

3.2 GC 算法 (YGC and FGC) .....	104
3.2.1 YGC.....	105
3.2.2 FGC.....	105
3.3 垃圾收集器 CMS.....	105
3.4 java 类加载机制 双亲委派.....	106
3.4.1 java 类加载的过程.....	106
3.4.2 双亲委派机制.....	108
3.4.3 破坏双亲委派模型.....	109
3.5 内存泄露.....	110
3.6 .内存泄露的案例分析 jvm 调优.....	110
3.6.1 jvm 调优目的.....	110
3.6.2 案例分析.....	110
3.7 jstat jmap jps jinfo jconsole.....	113
3.7.1 jstat.....	113
3.7.2 jmap.....	114
3.7 JVM 参数设置.....	114
3.8 内存分配与回收策略.....	116
3.9 面试问题.....	116
3.9.1 一般 Java 堆是如何实现的 ? .....	116
3.9.2 对象在内存中的初始化过程.....	117
3.9.3 对象的强、软、弱和虚引用.....	117
3.9.4 如何减少 GC 的次数.....	117
3.9.5 新生代 老年代 永久代.....	118
七 . juc 包.....	120
7.0 juc 概况.....	120
7.1 Tools.....	120
7.1.1 CountDownLatch.....	120
7.1.2 CyclicBarrier.....	120
7.1.3 Semaphore.....	121
7.1.4 Exchanger.....	121
7.2 List Set.....	121
7.3 Map.....	121
7.4 Queue.....	121
7.4.1 ArrayBlockingQueue.....	121
7.4.2 LinkedBlockingQueue.....	122
7.4.3 LinkedBlockingQueue 和 ArrayBlockingQueue 差异.....	122
7.5 线程池.....	123
1.线程池工作原理.....	123
2.线程池分类.....	123
3.线程池底层实现类 ThreadPoolExecutor 类.....	124
4.线程池状态.....	124
四.设计模式.....	125
4.0 什么是设计模式.....	125
4.1.常见的设计模式及其 JDK 中案例 : .....	125

4.1.1 适配器模式.....	125
4.1.2 迭代器模式.....	125
<b>4.1.3 代理模式.....</b>	<b>125</b>
4.1.4 观察者模式.....	126
4.1.5 装饰器模式.....	127
4.1.6 工厂模式.....	128
4.1.7 建造者模式.....	128
4.1.8 命令模式.....	129
4.1.9 责任链模式.....	129
4.1.10 享元模式.....	129
4.1.11 中介者模式.....	130
4.1.12 备忘录模式.....	130
4.1.13 组合模式.....	130
4.1.14 模板方法模式.....	130
<b>4.1.15 单例模式.....</b>	<b>131</b>
1. 非线程安全懒汉模式.....	131
2、 线程安全懒汉模式.....	131
3. 饿汉模式.....	131
4. 静态类内部加载.....	132
5. 双重锁校验模式.....	132
7. 懒汉模式与饿汉模式区别.....	133
8. 双重校验锁方法与线程安全的懒汉模式区别.....	133
<b>4.2 设计模式六大原则.....</b>	<b>133</b>
<b>4.3 java 动态代理.....</b>	<b>134</b>
算法.....	138
海量数据.....	142
七 . 数据结构与算法.....	143
<b>7.1 排序.....</b>	<b>143</b>
7.1.1 直接插入排序.....	143
7.1.2 希尔排序.....	145
7.1.3 冒泡排序.....	145
7.1.4 快速排序.....	145
7.1.5 直接选择排序.....	148
7.1.6 堆排序.....	149
7.1.7 归并排序.....	150
7.1.8 基数排序.....	153
<b>7.2 树.....</b>	<b>153</b>
7.2.1 二分查找树.....	153
八.数据库.....	155
<b>8.1 索引 B 树 B+树.....</b>	<b>155</b>
8.1.1 索引特点优缺点适用场合.....	155
8.1.2 Mysql 索引原理 B+树：.....	156
8.1.3 索引分类.....	159
<b>8.2 innodb 与 MyISAM 引擎区别.....</b>	<b>162</b>

8.3 事务隔离级别（恶果：脏读 幻读 不可重复读） .....	164
8.4 数据库特性 ACID.....	165
8.5 sql.....	165
8.5.1 . Sql 优化.....	165
8.6 5 种连接 left join、right join、inner join, full join cross join.....	168
8.7 数据库范式.....	171
8.8 数据库连接池.....	171
8.8.1 数据库连接池原理.....	171
8.8.2 数据库连接池的示例代码.....	172
8.9 DDL DML DCL.....	173
8.10 explain.....	173
8.11 分库分表.....	175
8.12 数据库锁.....	178
8.12.1 封锁.....	178
8.12.2 封锁协议（解决脏读不可重复读） .....	179
8.12.3 死锁活锁.....	180
8.12.3 解决死锁的方法.....	181
8.12.4 两段锁协议.....	183
8.12.5 GAP 锁（解决幻读） .....	183
8.12.6 next-key 锁.....	184
8.13 其它问题.....	184
8.13.1 limit20000 如何优化.....	185
8.13.2 数据库的隔离级别 隔离级别如何实现.....	185
8.13.3 char varchar text 区别.....	185
8.13.4 drop delete truncate 区别.....	186
8.13.5 事务.....	186
8.13.6 超键、候选键、主键、外键 视图.....	186
8.13.7 存储过程与触发器.....	186
九.网络.....	189
9.1.HTTP.....	189
9.1.1 http 请求报文 & http 响应报文.....	189
9.1.2 http 报文头部请求头和响应头.....	189
9.1.3 http 请求方法.....	190
9.1.4 http 请求过程.....	190
9.1.4 Get 和 Post 区别.....	191
9.1.5 http 状态码.....	192
9.1.6 http 长连接 短连接 HTTP 协议是无状态.....	194
9.1.7 http1.1 与 http1.0 的区别.....	195
9.1.8 http2.0 与 http1.0 的区别.....	195
9.1.9 转发与重定向的区别.....	195
9.2.TCP UDP.....	196
9.2.0 TCP 头部.....	196
9.2.1 TCP 与 UDP 区别.....	198

9.2.2 TCP 三次握手.....	200
9.2.3 TCP 四次挥手.....	201
9.2.4 tcp 粘包问题 nagle 算法.....	202
9.2.5 tcp 如何保证可靠性传输.....	203
9.2.6 TCP 流量控制 拥塞控制.....	204
9.2.7 滑动窗口机制.....	206
9.2.8 TCP 状态转移.....	208
9.2.9 TIME_WAIT 和 CLOSE_WAIT.....	210
9.7.计算机网络分层模型.....	212
9.7.1 osi 七层.....	212
9.7.2 APR.....	213
9.7.3 ICMP 协议.....	213
9.7.4 DNCP 协议.....	214
9.7.5 RARP 协议.....	214
9.7.6 路由选择协议 OSPF RIP.....	214
9.7.7 SNMP.....	215
9.7.8 SMTP.....	215
9.9 IP.....	216
1.IP 报文.....	216
2. IP 地址类别.....	216
3. 特殊的地址.....	216
4. 私有地址.....	217
9.10 网络攻击.....	217
1.SYN Flood 攻击.....	217
2. DDOS 攻击.....	217
3.DNS 欺骗.....	217
4. 重放攻击.....	217
5.SQL 注入.....	217
9.11 DNS 浏览器中输入 URL 到页面加载的发生了什么.....	218
CDN.....	219
9.12 https ssl.....	220
9.12.1 什么是 https.....	220
9.12.2 https 与 http 区别.....	220
9.12.3 https 的通信过程.....	221
9.12.4 SSL 工作原理.....	222
十 操作系统.....	224
10.1 进程线程.....	224
10.1.1 . 进程线程区别.....	224
10.1.2 进程通信方式.....	225
10.1.3 僵尸进程.....	225
10.1.4 进程同步 PV 信号量.....	225
10.2 死锁.....	227
10.2.1 死锁避免-银行家算法.....	227
10.2.2 死锁避免-安全序列.....	228

10.3 同步 异步 阻塞 非阻塞.....	228
10.4 操作系统 CPU 调度算法.....	229
10.5 内存管理方式（页存储 段存储 段页存储） .....	230
10.6 页面置换算法.....	232
10.6.1 概念.....	232
10.6.2 OPT 最优页面置换算法.....	232
10.6.3 先进先出置换算法（FIFO） .....	232
10.6.4 最近最久未使用（LRU） 算法.....	232
10.7 IO 种类 IO 的原理.....	233
1. IO 种类.....	234
2. 设备 I/O 输入输出控制方式.....	234
10.8 进程打开同一个文件 那么这两个进程得到的文件描述符（fd）相同.....	235
10.9 select epoll.....	235
10.10 物理地址 虚拟地址 逻辑地址.....	237
十一 Linux 命令.....	239
10.1 Vim.....	239
10.2 linux 如何查看端口被哪个进程占用？ .....	240
10.3 查看进程打开了哪些文件.....	241
10.4 top.....	241
10.5 查看 cpu 核的个数主频.....	241
10.6 Linux 如何创建守护进程.....	241
10.7 Linux 管道机制原理.....	242
10.8 查看进程下的线程.....	242
10.9 linux 锁.....	242
10.10 查看行数指令（比如第 100 行到第 150 行 top IP） .....	243
10.11 linux 进程调度.....	244
10.14 系统调用与库函数的区别.....	245
10.15 free.....	246
10.16 cache 和 buffer 的区别： .....	246
10.13 其它的小小问题.....	247
十一. 安全加密.....	248
11.1 数字签名.....	248
11.2 数字证书.....	249
11.3 公私钥.....	249
11.4 非对称加密 RSA.....	249
11.5 对称密钥 DES.....	250
11.6 DH 加密算法.....	250
11.7 SHA MD5.....	250
十二. 代码.....	251
12.1 读写文件(BufferedReader).....	251
12.2 反射.....	252
12.3 LRU.....	255

十三 . 面经.....	257
十四 . 项目.....	259
14.1. jieba 分词原理.....	259
Python.....	260
Git.....	260
计算机磁盘.....	260
Socket.....	261
其它.....	261

## — java 基础

### 1.1java 的 8 种基本数据类型 装箱 拆箱

<https://blog.csdn.net/daidaineteasy/article/details/51088269>

#### 1.1.1 . 8 种基本数据类型

Byte short int long float double boolean char

数据类型	关键字	在内存中占用字节数	取值范围	默认值
布尔型	boolean	1	true,false	false
字节型	byte	1	-128~127	0
短整型	short	2	-2 <sup>15</sup> ~ 2 <sup>15</sup> -1	0
整型	int	4	-2 <sup>31</sup> ~ 2 <sup>31</sup> -1	0
长整型	long	8	-2 <sup>63</sup> ~ 2 <sup>63</sup> -1	0
字符型	char	2	0 ~ 2 <sup>16</sup> -1	'\u0000'
单精度浮点型	float	4	1.4013E-45 ~ 3.4028E+38	0.0F
双精度浮点型	double	8	4.9E-324 ~ 1.7977E+308	0.0D

### 1.1.2 . 装箱和拆箱

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。

原始类型: boolean, char, byte, short, int, long, float, double

封装类型: Boolean, Character, Byte, Short, Integer, Long, Float, Double

### 1.1.3. String 转出 int 型，判断能不能转？如何转？

答：可以转，得处理异常 Integer.parseInt(s) 主要为 NumberFormatException : 1) 当你输入为字母时，也就是内容不是数字时，如 abcd 2) 当你输入为空时 3) 当你输入超出 int 上限时 Long.parseLong("123")转换为 long

## 1.1.4 short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 +=1;有什么错?

- 1) 对于 short s1=1;s1=s1+1 来说, 在 s1+1 运算时会自动提升表达式的类型为 int, 那么将 int 赋予给 short 类型的变量 s1 会出现类型转换错误。
- 2) 对于 short s1=1;s1+=1 来说 +=是 java 语言规定的运算符, java 编译器会对它进行特殊处理, 因此可以正确编译。

## 1.1.5 . Int 与 Integer 区别

<https://www.cnblogs.com/guodongdidi/p/6953217.html>

### int和Integer的区别

- 1、Integer是int的包装类, int则是java的一种基本数据类型
  - 2、Integer变量必须实例化后才能使用, 而int变量不需要
  - 3、Integer实际是对象的引用, 当new一个Integer时, 实际上是生成一个指针指向此对象; 而int则是直接存储数据值
  - 4、Integer的默认值是null, int的默认值是0
- 延伸 :
- 关于Integer和int的比较
1. 由于Integer变量实际上是对一个Integer对象的引用, 所以两个通过new生成的Integer变量永远是不相等的(因为new生成的是两个对象, 其内存地址不同)。
  2. Integer变量和int变量比较时, 只要两个变量的值是相等的, 则结果为true (因为包装类Integer和基本数据类型int比较时, java会自动拆包装为int, 然后进行比较, 实际上就变为两个int变量的比较)
  3. 非new生成的Integer变量和new Integer()生成的变量比较时, 结果为false。(因为非new生成的Integer变量指向的是java常量池中的对象, 而new Integer()生成的变量指向堆中新建的对象, 两者在内存中的地址不同)
  4. 对于两个非new生成的Integer对象, 进行比较时, 如果两个变量的值在区间-128到127之间, 则比较结果为true, 如果两个变量的值不在此区间, 则比较结果为false
- ```
Integer i = new Integer(100);
Integer j = new Integer(100);
System.out.print(i == j); //false
```
- ```
Integer i = new Integer(100);
int j = 100;
System.out.print(i == j); //true
```
- ```
Integer i = new Integer(100);
Integer j = 100;
System.out.print(i == j); //false
```
- ```
Integer i = 100;
Integer j = 100;
System.out.print(i == j); //true
```
- ```
Integer i = 128;
Integer j = 128;
System.out.print(i == j); //false
```

对于第4条的原因：

java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)；，而java API中对Integer类型的valueOf的定义如下：

```
public static Integer valueOf(int i){  
    assert IntegerCache.high >= 127;  
    if (i >= IntegerCache.low && i <= IntegerCache.high){  
        return IntegerCache.cache[i + (-IntegerCache.low)];  
    }  
    return new Integer(i);  
}
```

java对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，  
new了

## 1.1.6 . 字节字符区别

字节是存储容量的基本单位，字符是数字，字母，汉字以及其他语言的各种符号。  
1字节=8个二进制单位：一个一个字符由一个字节或多个字节的二进制单位组成。

## 1.1.7 java 基本类型与引用类型的区别

基本类型保存原始值，引用类型保存的是引用值（引用值就是指对象在堆中所处的位置/地址）

## 1.2 重写重载封装继承多态

- Java的四个基本特性
  - 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。
  - 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。
  - 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口。
  - 多态性是指允许不同子类型的对象对同一消息作出不同的响应。
- 多态的理解(多态的实现方式)
  - 方法重载 (overload) 实现的是编译时的多态性（也称为前绑定）。
  - 方法重写 (override) 实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西。
  - 要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2). 对象造型（用父类型引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。
- 项目中对多态的应用
  - 举一个简单的例子，在物流信息管理系统中，有两种用户：订购客户和卖房客户，两个客户都可以登录系统，他们有相同的方法Login，但登陆之后他们会进入到不同的页面，也就是在登录的时候会有不同的操作，两种客户都继承父类的Login方法，但对于不同的对象，拥有不同的操作。

## 一. 继承的好处和坏处

好处: 1. 子类能自动继承父类的接口

2. 创建子类的对象时, 无须创建父类的对象

坏处: 1. 破坏封装, 子类与父类之间紧密耦合, 子类依赖于父类的实现, 子类缺乏独立性

2. 支持扩展, 但是往往以增加系统结构的复杂度为代价

3. 不支持动态继承。在运行时, 子类无法选择不同的父类

4. 子类不能改变父类的接口

## 二. 重载、重写

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。

- 重载: 重载发生在同一个类中, 同名的方法如果有不同的参数列表(参数类型不同、参数个数不同或者二者都不同)则视为重载。
- 重写: 重写发生在子类与父类之间, 重写要求子类被重写方法与父类被重写方法有相同的返回类型, 比父类被重写方法更好访问, 不能比父类被重写方法声明更多的异常(里氏代换原则)。根据不同的子类对象确定调用的那个方法。

<https://blog.csdn.net/cey009008/article/details/46331619>

重写 (override) 又名覆盖:

1. 不能存在同一个类中, 在继承或实现关系的类中;
2. 名相同, 参数列表相同, 方法返回值相同,
3. 子类方法的访问修饰符要大于父类的。
4. 子类的检查异常类型要小于父类的检查异常。

重载 (overload)

1. 可以在一个类中也可以在继承关系的类中;
2. 名相同;
3. 参数列表不同(个数, 顺序, 类型) 和方法的返回值类型无关。

## 三. Java 中是否可以覆盖 override一个 private 或者是 static 的方法?

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。

java 中也不可以覆盖 private 的方法, 因为 private 修饰的变量和方法只能在当前类中使用, 如果是其他的类继承当前类是不能访问到 private 变量或方法的, 当然也不能覆盖。

## 1.3 Stack Queue

### 1.3.1 PriorityQueue

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  $O(\log(n))$ 。

堆树的定义如下：

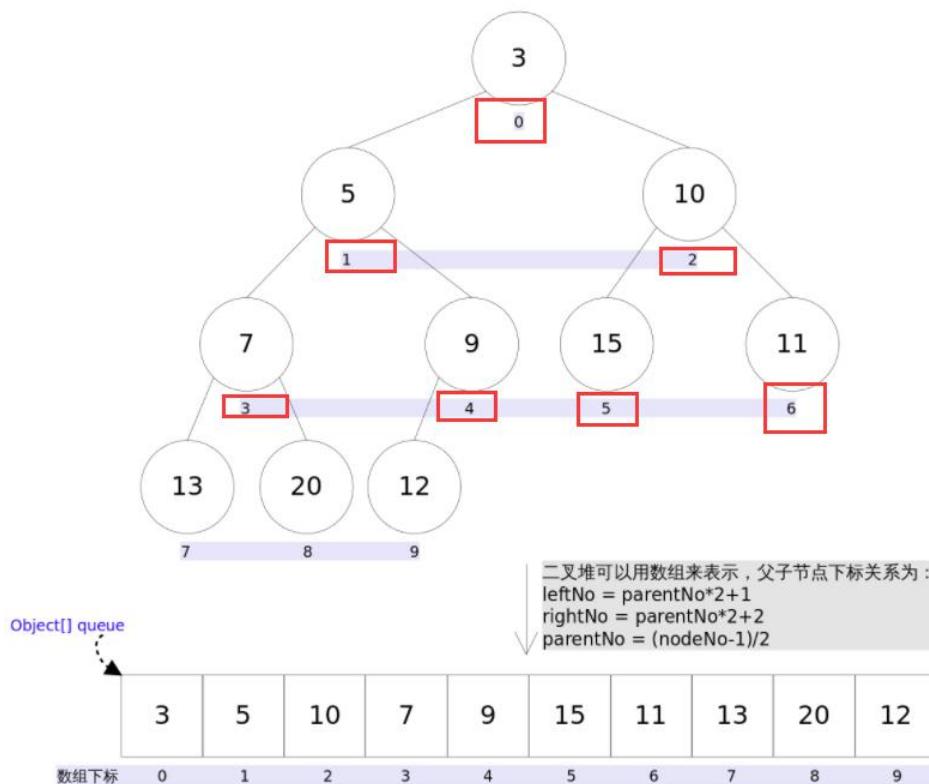
- (1) 堆树是一颗完全二叉树；
- (2) 堆树中某个节点的值总是不大于或不小于其孩子节点的值；
- (3) 堆树中每个节点的子树都是堆树。

## 1. 原理

<https://www.cnblogs.com/CarpenterLee/p/5488070.html>

孩子节点的下标例如下标 2 5 6    (5-1)/2=2 (6-1)/2=2 每个父节点的值都比孩子节点的值要比孩子节点的值要小。

PriorityQueue 通过用数组表示的小顶堆实现

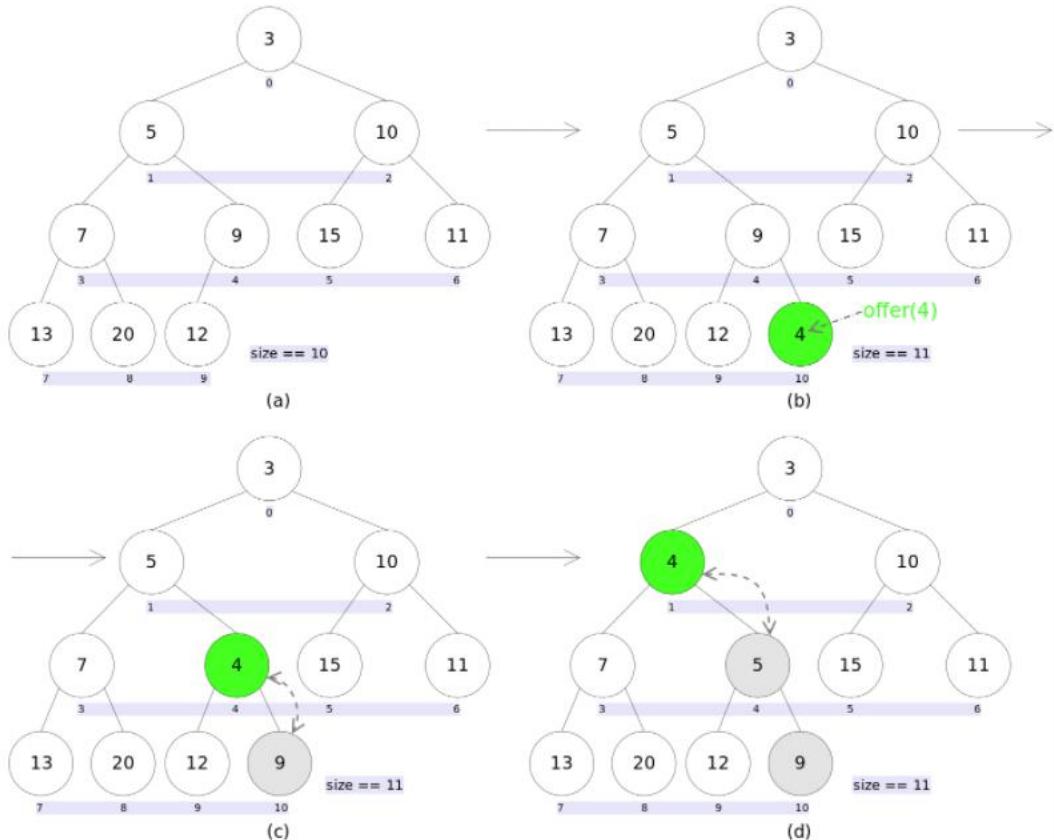


### 1. 添加元素 add() 和 offer()

原理：添加元素位于末尾，同时队列长度加 1，然后这个元素与它的父节点进行比较，如果比父节点小那么就与父节点进行交换，然后再与交换后的位置的父节点进行比较，重复这个过程，直到该元素的值大于父节点结束这个过程。

区别：add(E e) 和 offer(E e) 的语义相同，都是向优先队列中插入元素，只是 Queue 接口规定二者对插入失败时的处理不同，前者在插入失败时抛出异常，后者则会返回 false。对于 PriorityQueue 这两个方法其实没什么差

PriorityQueue.offer(E e) siftUp过程图解



## 2. 寻找队列的头部元素 element() 和 peek() 头部元素 时间复杂度为 1

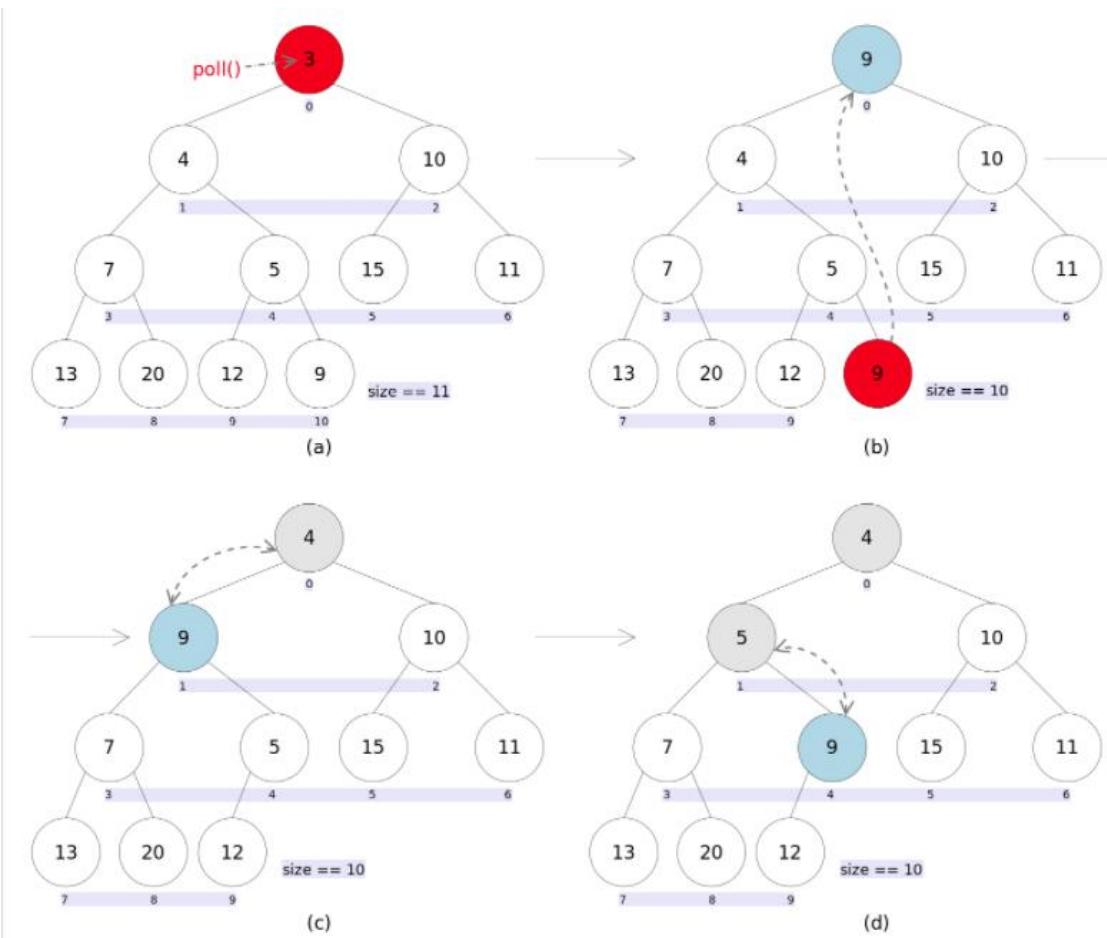
element() 和 peek() 的语义完全相同，都是获取但不删除队首元素，也就是队列中权值最小的那个元素，二者唯一的区别是当方法失败时前者抛出异常，后者返回 null。根据小顶堆的性质，堆顶那个元素就是全局最小的那个；由于堆用数组表示，根据下标关系，0 下标处

的那个元素既是堆顶元素。所以直接返回数组 0 下标处的那个元素即可。

## 4. 删除元素 remove() 和 poll()

区别：remove() 和 poll() 方法的语义也完全相同，都是获取并删除队首元素，区别是当方法失败时前者抛出异常，后者返回 null。由于删除操作会改变队列的结构，为维护小顶堆的性质，需要进行必要的调整。

原理：该方法的作用是从 k 指定的位置开始，将 x 逐层向下与当前点的左右孩子中较小的那个交换，直到 x 小于或等于左右孩子中的任何一个为止



### 3. 最大堆 获取数组中最小的几个数 最小堆 获取数组中最大的几个数

```

public static ArrayList<Integer> GetLeastNumbers_Solution(int[] input, int k) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    int length = input.length;
    if (k > length || k == 0) {
        return result;
    }
    PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(k, new Comparator<Integer>() {

        public int compare(Integer o1, Integer o2) {
            return o2.compareTo(o1);
        }
    });
    for (int i = 0; i < length; i++) {
        if (maxHeap.size() != k) {
            maxHeap.offer(input[i]);
        } else if (maxHeap.peek() > input[i]) {
            Integer temp = maxHeap.poll();
            temp = null;
            maxHeap.offer(input[i]);
        }
    }
    for (Integer integer : maxHeap) {
        System.out.println(integer);
        result.add(integer);
    }
    return result;
}

```

上述代码是构建最大堆，最大堆的栈顶是堆的最大的元素，最大的元素比数组中任意一个元素小，说明了最大堆这些元素是数组中最小的几个元素。

#### 上述代码为何需要重写比较器函数 compare() ???

答：需要查看优先队列的源码，如下源码所示，添加元素需要比较新的元素与父节点的元素，

如果比较器比较结果大于 0，那么结束添加过程添加完成，说明在构建最大堆时候，要想使得元素是对父节点的元素小才结束循环，那么必须重新写比较器函数，调换两者的比较顺序即可。

最小堆，与上述过程相反。

```

//offer(E e)
public boolean offer(E e) {
    if (e == null)//不允许放入null元素
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);//自动扩容
    size = i + 1;
    if (i == 0)//队列原来为空，这是插入的第一个元素
        queue[0] = e;
    else
        siftUp(i, e);//调整
    return true;
}

```

```
//siftUp()
private void siftUp(int k, E x) {
    while (k > 0) {
        int parent = (k - 1) >>> 1;//parentNo = (nodeNo-1)/2
        Object e = queue[parent];
        if (comparator.compare(x, (E) e) >= 0)//调用比较器的比较方法
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = x;
}
```

## 1.7 Concurrent 包

Concurrent 包里的其他东西：ArrayBlockingQueue、CountDownLatch 等等

- `LinkedBlockingQueue`
- 所用过的类
  - `Executor`
- concurrent下面的包
  - `Executor` 用来创建线程池，在实现`Callable`接口时，添加线程。
  - `FutureTask` 此 `FutureTask` 的 `get` 方法所返回的结果类型。
  - `TimeUnit`
  - `Semaphore`

## 1.8 面向对象

面向对象开发的六个基本原则(单一职责、开放封闭、里氏替换、依赖倒置、合成聚合复用、接口隔离)，迪米特法则。在项目中用过哪些原则

- 六个基本原则
  - 单一职责：一个类只做它该做的事情(高内聚)。在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。
  - 开放封闭：软件实体应当对扩展开放，对修改关闭。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而混乱。
  - 里氏替换：任何时候都可以用子类型替换掉父类型。子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。
  - 依赖倒置：面向接口编程。(该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代)
  - 合成聚合复用：优先使用聚合或合成关系复用代码。
  - 接口隔离：接口要小而专，绝不能大而全。臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。
- 迪米特法则
  - 迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。
- 项目中用到的原则
  - 单一职责、开放封闭、合成聚合复用(最简单的例子就是String类)、接口隔离

### Java 创建对象的四种方法：

1. 使用 `new` 关键字
  2. 使用 `Class` 类的 `newInstance` 方法，`newInstance` 方法调用无参的构造器创建对象(**反射**)  
`Class.forName().newInstance()`
  3. 使用 `clone` 方法
  4. 反序列化
- 使用**构造器**的三种 (`new` 和**反射**的两种 `newInstance`)，没用**构造器**的两种 (`clone` 和**反序列化**)

面向对象特点、C++Java 面向对象设计的不同、

面向对象的设计规范

对面向对象的认识

面向对象的特征

## 1.9 String StringBuffer StringBuilder hashCode equals

### 一. String StringBuffer StringBuilder 的区别

- String、StringBuffer、StringBuilder
  - 都是 final 类，都不允许被继承；
  - String 长度是不可变的，StringBuffer、StringBuilder 长度是可变的；
  - StringBuffer 是线程安全的，StringBuilder 不是线程安全的，但它们两个中的所有方法都是相同的，StringBuffer 在 StringBuilder 的方法之上添加了 synchronized 修饰，保证线程安全。
  - StringBuilder 比 StringBuffer 拥有更好的性能。
  - 如果一个 String 类型的字符串，在编译时就可以确定是一个字符串常量，则编译完成之后，字符串会自动拼接成一个常量。

量。此时String的速度比StringBuffer和StringBuilder的性能好的多。

## 二. String 不可变?

### 1.String不可变? final?

- final

- 首先因为String不可变，如果String不是final，那么就可以有子类继承String类，然后子类覆盖其方法，使得这些方法可以修改字符串，这样就违背了String的不可变性

- 不可变原因

- 提高效率：比如一个字符串String s1=“abc”，“abc”被放到常量池里面去了，我再String s2 = "abc"并不会复制字符串“abc”，只会多个引用指向原来那个常量，这样就提高了效率，而这一前提是string不可变，如果可变，那么多个引用指向同一个字符串常量，我就可以通过一个引用改变字符串，然后其他引用就被影响了
- 安全：string常被用来表示url，文件路径，如果string可变，或存在安全隐患
- string不可变，那么他的hashcode就一样，不用每次重新计算了

- String不变性的理解

- String 类是被final进行修饰的，不能被继承。
- 在用+号链接字符串的时候会创建新的字符串。
- String s = new String("Hello world"); 可能创建两个对象也可能创建一个对象。如果静态区中有“Hello world”字符串常量对象的话，则仅仅在堆中创建一个对象。如果静态区中没有“Hello world”对象，则堆上和静态区中都需要创建对象。
- 在 java 中，通过使用 "+" 符号来串联字符串的时候，实际上底层会转成通过 StringBuilder 实例的 append() 方法来实现。

## String 中的 hashCode 以及 toString

### 12. String有重写Object的hashCode和toString吗？如果重写equals不重写hashCode会出现什么问题？

- String有重写Object的hashCode和toString吗？
  - String重写了Object类的hashCode和toString方法。
- 当equals方法被重写时，通常有必要重写hashCode方法，以维护hashCode方法的常规协定，该协定声明相对等的两个对象必须有相同的hashCode
  - object1.equals(object2)时为true，object1.hashCode() == object2.hashCode() 为true
  - object1.hashCode() == object2.hashCode() 为false时，object1.equals(object2)必定为false
  - object1.hashCode() == object2.hashCode() 为true时，但object1.equals(object2)不一定为true
- 重写equals不重写hashCode会出现什么问题
  - 在存储散列集合时(如Set类)，如果原对象.equals(新对象)，但没有对hashCode重写，即两个对象拥有不同的hashCode，则在集合中将会存储两个值相同的对象，从而导致混淆。因此在重写equals方法时，必须重写hashCode方法。

- ⑥ 因为别人知道源码怎么实现的，故意构造相同的hash的字符串进行攻击，怎么处理？那jdk7怎么办？
  - 怎么处理构造相同hash的字符串进行攻击？
    - 当客户端提交一个请求并附带参数的时候，web应用服务器会把我们的参数转化成一个HashMap存储，这个HashMap的逻辑结构如下：key1-->value1；
    - 但是物理存储结构是不同的，key值会被转化成HashCode，这个HashCode有会被转成数组的下标：0-->value1；
    - 不同的String就会产生相同HashCode而导致碰撞，碰撞后的物理存储结构可能如下：0-->value1-->value2；
    - 1、限制post和get的参数个数，越少越好
    - 2、限制post数据包的大小
    - 3、WAF
  - Jdk7 如何处理hashCode字符串攻击
    - HashMap会动态的使用一个专门的treemap实现来替换掉它。

使用场景

#### 四 . String, 是否可以继承, “+”怎样实现, 与 StringBuffer 区别

在 java 中, 通过使用 "+" 符号来串联字符串的时候, 实际上底层会转成通过 StringBuilder 实例的 append() 方法来实现。

## 1.10 java 文件读取

FileReader 类是将文件按字符流的方式读取 char 数组或者 String

, FileInputStream 则按字节流的方式读取文件 byte 数组

1. 首先获得一个文件句柄。File file = new File(); file 即为文件句柄。两人之间连通电话网络了。接下来可以开始打电话了。
2. 通过这条线路读取甲方的信息: new FileInputStream(file) 目前这个信息已经读进来内存当中了。接下来需要解读成乙方可以理解的东西
3. 既然你使用了 FileInputStream()。那么对应的需要使用 InputStreamReader() 这个方法进行解读刚才装进来内存当中的数据
4. 解读完成后要输出呀。那当然要转换成 IO 可以识别的数据呀。那就需要调用字节码读取的方法 BufferedReader()。同时使用 bufferedReader() 的 readLine() 方法读取 txt 文件中的每一行数据哈。

## 1.11 Java 反射

反射机制是在**运行状态中**，对于任意一个类，都能够**知道这个类的所有属性和方法**；对于任意一个对象，都能够**调用它的任意一个方法和属性**；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。

获取 Class 对象方法: class.getclass(), Class.forName, ClassLoader.loadClass

.Class.forName 和 classloader.loadClass 的区别

初始化不同：

1. : Class.forName()会对类初始化，而 loadClass()只会装载或链接。
2. forname 在类加载的时候回执行静态代码块（初始化了），loadclass 只有在调用 newInstance 方法的时候才会执行静态代码块

类加载器不同：

1. Class.forName(String) 方法(只有一个参数)， 使用调用者的类加载器来加 载，也 就 是 用加载了调用 forName 方法的代码的那个类加载器

ClassLoader.loadClass()方法是一个实例方法(非静态方法)，调用时需要自己指定类加载器，

### 1. 反射机制fsh

- 概念：就是运行时动态地获取类的信息和操作类的方法和属性
- 获取Class对象方法: class.getclass(), Class.forName, ClassLoader.loadClass
- **Class类方法：**  
getClasLoader, forname, getdeclareemethod, getdeclaredfield, getconstrutor(declared可以获取所有方法，不加只获取public方法)

### 2. Class.forName 和 classloader.loadClass 的区别

- forname在类加载的时候回执行静态代码块
- loadclass只有在调用newInstance方法的时候才会执行静态代码块
- 初始化不同：Class.forName()会对类初始化，而loadClass()只会装载或链接。可见的效果就是类中静态初始化段及字节码中对所有静态成员的初始工作的执行(这个过程在类的所有父类中递归地调用)，这点就与 ClassLoader.loadClass() 不同。ClassLoader.loadClass() 加载的类对象是在第一次被调用时才进行初始化的。你可以利用上述的差异。比如，要加载一个静态初始化开销很大的类，你就可以选择提前加载该类(以确保它在classpath下)，但不进行初始化，直到第一次使用该类的域或方法时才进行初始化
- 类加载器不用：Class.forName(String) 方法(只有一个参数)，使用调用者的类加载器来加载，也就是用加载了调用forName方法的代码的那个类加载器。当然，它也有个重载的方法，可以指定加载器。相应的，ClassLoader.loadClass()方法是一个实例方法(非静态方法)，调用时需要自己指定类加载器，那么这个类加载器就可能是也可能不是加载调用代码的类加载器（调用代用代码类加载器通过getClassLoader()获得）

Java 反射的概念和应用场景

反射机制中可以获取 private 成员的值吗（没有 set 和 get 函数）可以

反射的所有包，怎实现反射

反射的定义

Java.lang.reflect 里常用方法

## 1.12 JDK NDK JRE JNI

### 1.JDK 与 JRE 的区别

Java 运行时环境(JRE)。它包括 Java 虚拟机、Java 核心类库和支持文件。它不包含开发工具 (JDK) --编译器、调试器和其他工具。

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。

### 2. 版本特性

#### 1.jdk1.8新特性

- 接口可用default关键字添加非抽象方法
- lambda表达式:以前Java集合是不能够进行内部迭代的，只能用循环外部迭代，有了lambda就能内部迭代
- [详细内同](#)

#### 2.jdk1.7新特性

- 可以通过[],{}的形式向集合内添加元素
- switch支持string类型数据
- 数值可加下划线

#### 3.jdk1.5新特性

- for each循环
- 自动装拆箱：基本数据类型和它的包装类自动转换，Integer.valueOf Integer.intValue
- 泛型：其实是类型擦除，编译完类型就没了，执行方法又回到了原来的类型强转
- 可变参数

JDK 中哪些体现了命令模式

JDK 发展

了解 NDK 吗？他和 JDK 有什么区别呢？

使用过 JNI 么？NDK 技术使用过哪些呢？

## 1.13 static 和 final 的区别

- Static
  - 修饰变量：静态变量随着类加载时被完成初始化，内存中只有一个，且JVM也只会为它分配一次内存，所有类共享静态变量。
  - 修饰方法：在类加载的时候就存在，不依赖任何实例；static方法必须实现，不能用abstract修饰。
  - 修饰代码块：在类加载完之后就会执行代码块中的内容。
  - 父类静态代码块->子类静态代码块->父类非静态代码块->父类构造方法->子类非静态代码块->子类构造方法
- Final
  - 修饰变量：
    - 编译期常量：类加载的过程完成初始化，编译后带入到任何计算式中。只能是基本类型。
    - 运行时常量：基本数据类型或引用数据类型。引用不可变，但引用的对象内容可变。
  - 修饰方法：不能被继承，不能被子类修改。
  - 修饰类：不能被继承。
  - 修饰形参：final形参不可变

**final 的好处：**

1. final 关键字提高了性能。JVM 和 Java 应用都会缓存 final 变量。
2. final 变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。
3. 使用 final 关键字，JVM 会对方法、变量及类进行优化。

### 一、 static 方法是否可以覆盖？

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。

### 二. 是否可以在 static 环境中访问非 static 变量？

static 修饰的变量并发下怎么保证变量的安全

static 修饰的变量什么时候赋值

static 什么时候使用

```
class A {
    private static A a=new A();
    static{
        System.out.print("static");
    }
    public A(){
        System.out.print("A");
    }
}
public class B extends A{
    public B(){
        System.out.print("B");
    }
    public static void main(String[] args) {
        B b=new B();
    }
}
```

上述运行结果 AstaticAB

```
class A {
    private static A a=new A();
    static{
        System.out.print("static");
    }
    {
        System.out.print("A");
    }
}
public class B extends A{
    public B(){
        System.out.print("B");
    }
    public static void main(String[] args) {
        B b=new B();
    }
}
```

```
    }  
}
```

上述结果也是 AstaticAB 匿名构造器 构造器先与静态代码块执行，静态代码块只执行一次

```
class A {  
    public A() {  
        System.out.print("A gouzhaos");  
    }  
    private static A a=new A();  
    static {  
        System.out.print("static");  
    }  
    {  
        System.out.print("A1");  
    }  
}  
  
public class B extends A{  
    public B() {  
        System.out.print("B");  
    }  
    public static void main(String[] args) {  
        System.out.println("0000");  
        B b=new B();  
    }  
}
```

运行结果：A1A gouzhaostatic0000

A1A gouzhaosB

**解释：静态变量，静态代码块先于 System.out.println("0000") 执行，静态的东西只执行一次，相当于全局变量。**

## 1.14 map, list, set 区别

有哪些类，这些类有什么区别

## 1.16 Session 和 COOKIE

1.session 和 cookie 区别

- session 在服务器端，cookie 在客户端（浏览器）
- 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，同时 session 也会失效（但是可以通过其它方式实现，比如在 url 中传递 session\_id）
- session 可以放在文件、数据库、或内存中都可以。
- 用户验证这种场合一般会用 session
- cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗 考虑到安全应当使用 session。
- session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE。
- 单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

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。

服务器端 Session 的保存

Cookie 和 session 区别

session 在服务器上以怎样的形式存在 session 持久化

怎么设置 session 和 cookie 的有效时间

Session 的实现原理和应用场景 Session 原理；既然 Session 是存储在服务器内存的，如果服务器的负载量很高内存负荷不住要怎么办？

## 1.19 IO NIO BIO AIO select epoll

- BIO
  - 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
  - BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。
- NIO
  - 同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有 I/O 请求时才启动一个线程进行处理。
  - NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。
- AIO
  - 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理。
  - AIO 方式适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

## os (select/epoll) 区别 —— 知识点 IO 多路复用

IO 多路复用的机制

多路复用是同步非阻塞 I/O，即 Synchronous I/O Multiplexing，它是利用单独的线程（内核级）统一监测所有 Socket，一旦某个 Socket 有了 I/O 数据，则启动相应的 Application 处理，在 select 和 poll 中利用轮询 socket 句柄的方式来实现监测 socket 中是否有 I/O 数据到达，这种方式有开销，epoll 等则改进了这种方式，利用底层 notify 机制，即 Reactor 方式来监测，[Java NIO](#) 也是采用这种机制。这里需要注意，其实多路复用还是有阻塞的（这个阻塞并非以上定义的阻塞，这里指 Socket 无 I/O 数据时还是被 wait，此外当使用 select 函数 copy I/O 数据入 Application Buffer 时，Application 还是被阻塞的）

Select 和 epoll 区别：

1. 每次调用 select，都需要把 fd 集合从用户态拷贝到内核态，这个开销在 fd 很多时会很大
2. 同时每次调用 select 都需要在内核遍历传递进来的所有 fd，这个开销在 fd 很多时也很大
3. select 支持的文件描述符数量太小了，默认是 1024

poll

它没有最大连接数的限制，原因是它是基于链表来存储的，但是同样有一个缺点：大量的 fd 的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。

4. epoll 为每个 fd 指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数
5. epoll 所支持的 FD 上限是最大可以打开文件的数目

int,long 在 32/64 位操作系统大小

int 32/64 4 字节 long 32 位 4 字节，64 位 8 字节

协程：看上去也是子程序，但执行过程中，在子程序内部可中断，然后转而执行别的子程序，在适当的时候再返回来接着执行。yield

什么是阻塞和非阻塞，什么是同步和异步，同步和异步是针对应用程序和内核的交互而言的，

同步指的是用户进程触发 IO 操作并等待或者轮询的去查看 IO 操作是否就绪，而异步是指

用户进程触发 IO 操作以后便开始做自己的事情，而当 IO 操作已经完成的时候会得到 IO 完

成的通知。而阻塞和非阻塞是针对于进程在访问数据的时候，根据 IO 操作的就绪状态来采

取的不同方式，说白了是一种读取或者写入操作函数的实现方式，阻塞方式下读取或者写入

函数将一直等待，而非阻塞方式下，读取或者写入函数会立即返回一个状态值。

一般来说 I/O 模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞 IO

**同步阻塞 IO** : 在此种方式下 , 用户进程在发起一个 IO 操作以后 , 必须等待 IO 操作的完成 , 只有当真正完成了 IO 操作以后 , 用户进程才能运行。 JAVA 传统的 IO 模型属于此种方式 !

**同步非阻塞 IO** : 在此种方式下 , 用户进程发起一个 IO 操作以后便可返回做其它事情 , 但是用户进程需要时不时的询问 IO 操作是否就绪 , 这就要求用户进程不停的去询问 , 从而引入不必要的 CPU 资源浪费。其中目前 JAVA 的 NIO 就属于同步非阻塞 IO。

**异步阻塞 IO** : 此种方式下是指应用发起一个 IO 操作以后 , 不等待内核 IO 操作的完成 , 等内核完成 IO 操作以后会通知应用程序 , 这其实就是同步和异步最关键的区别 , 同步必须等待或者主动的去询问 IO 是否完成 , 那么为什么说是阻塞的呢 ? 因为此时是通过 select 系统调用完成的 , 而 select 函数本身的实现方式是阻塞的 , 而采用 select 函数有个好处就是它可以同时监听多个文件句柄 , 从而提高系统的并发性 !

**异步非阻塞 IO** : 在此种模式下 , 用户进程只需要发起一个 IO 操作然后立即返回 , 等 IO 操作真正的完成以后 , 应用程序会得到 IO 操作完成的通知 , 此时用户进程只需要对数据进行处理就好了 , 不需要进行实际的 IO 读写操作 , 因为真正的 IO 读取或者写入操作已经由内核完成了。

### 1.19.1 NIO 的原理

NIO 中有几个核心对象需要掌握：缓冲区（Buffer）、通道（Channel）、选择器（Selector）。

所以使用 NIO 读取数据可以分为下面三个步骤：

1. 从 FileInputStream 获取 Channel
2. 创建 Buffer
3. 将数据从 Channel 读取到 Buffer 中

flip 方法将 Buffer 从写模式切换到读模式。调用 flip()方法会将 position 设回 0，并将 limit 设置成之前 position 的值。

从 Buffer 中读取数据有两种方式：

从 Buffer 读取数据到 Channel。

使用 get()方法从 Buffer 中读取数据。

写数据到 Buffer 有两种方式：

从 Channel 写到 Buffer。

通过 Buffer 的 put()方法写到 Buffer 里。

为了理解 Buffer 的工作原理，需要熟悉它的三个属性：

capacity

position

limit

为什么使用 Selector? 仅用单个线程来处理多个 Channels 的好处是，只需要更少的线程来处理通道

与 Selector 一起使用时，Channel 必须处于非阻塞模式下。这意味着不能将 FileChannel 与 Selector 一起使用，因为 FileChannel 不能切换到非阻塞模式。而套接字通道都可以。

监听四种不同类型的事件

Connect

Accept

Read

Write

一旦调用了 select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用 selector 的 selectedKeys()方法，访问“已选择键集（selected key set）”中的就绪通道。

Java NIO 和 IO 的主要区别

1. Java NIO 和 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。
2. Java IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程被阻塞，直到有一些数据被读取  
Java NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。
3. Java NIO 的选择器允许一个单独的线程来监视多个输入通道

NIO 的 DirectByteBuffer 和 HeapByteBuffer

.IO 哪个类可以 Byte 转 String。

java NIO 的实现原理，我给他将了阻塞非阻塞，同步异步，Buffer 与 Channel 以及 Selector 的运行机制，然后又问 NIO 主要解决的是什么问题

## 1.20 ThreadLocal

Threadlocal 不是解决对象的共享访问问题，通过 `ThreadLocal.set()` 到线程中的对象是该线程自己使用的对象，其他线程是不需要访问的，也访问不到的。各个线程中访问的是不同的对象。

各个线程独立的对象不是通过 `Threadlocal.set` 创建的，而是在每个线程 `new` 的对象。

`Threadlocal.set` 将新建对象的引用保存到线程独有的 map 中，当有其他线程对这个引用指向的对象做修改时，当前线程 `Thread` 对象中保存的值也会发生变化

Threadlocal 源码：

Thread 类中有 `threadlocalmap` 对象

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        return (T)map.get(this);

    // Maps are constructed lazily.  if the map for this thread
    // doesn't exist, create it, with this ThreadLocal and its
    // initial value as its only entry.
    T value = initialValue();
    createMap(t, value);
    return value;
}
```

当使用 `ThreadLocal` 维护变量时，`ThreadLocal` 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

## 1.22 finalize finalization finally

### 一. finalize 用途

答：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。

### 二. finally

Try catch finally

在 try 中 `return` 之前会执行 `finally` 中的代码，如果 `finally` 中有 `return` 则直接 `return`，值为 `finally` 中修改的；

如果 `finally` 中没有 `return`，则执行 try 中 `return`，数值仍然是 try 中的

`finally` 一定会被执行，如果 `finally` 里有 `return` 语句，则覆盖 `try/catch` 里的 `return`，  
比较爱考的是 `finally` 里没有 `return` 语句，这时虽然 `finally` 里对 `return` 的值进行了  
修改，但 `return` 的值并不改变这种情况

### 三. `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。

## 1.23 public private default protected

访问修饰符作用域

| 访问级别 | 访问控制修饰符                | 同类 | 同包 | 子类 | 不同的包 |
|------|------------------------|----|----|----|------|
| 公开   | <code>public</code>    | ✓  | ✓  | ✓  | ✓    |
| 受保护  | <code>protected</code> | ✓  | ✓  | ✓  | --   |
| 默认   | 没有访问控制修饰符              | ✓  | ✓  | -- | --   |
| 私有   | <code>private</code>   | ✓  | -- | -- | --   |

不写时默认为 `default`。默认对于同一个包中的其他类相当于公开（`public`），  
对于不是同一个包中的其他类相当于私有（`private`）。受保护（`protected`）对子类  
相当于公开，对不是同一包中的没有父子关系的类相当于私有。

不可以覆盖 `private` 的方法，因为 `private` 修饰的变量和方法只能在当前类中使用，  
如果是其他的类继承当前类是不能访问到 `private` 变量或方法的，当然也不能覆

## 1.25 Object

`hashcode()` `equals()` `toString()` `getClass()`  
`wait` `notify()` `notifyAll()`  
`finalize()`

## 1.26 `eqls` 和 `==` 的区别

[https://blog.csdn.net/love\\_xsq/article/details/43760771](https://blog.csdn.net/love_xsq/article/details/43760771)

基本数据类型用（==）进行比较的时候，比较的是实际值

复合数据类型，（==）比较的是他们在内存中的存放地址

复合数据类型中 equal , (String ‘Ingeter’ , date) 等重写了 equal, 比较的是内容

没有重写 equal 的，比较的还是内存地址

StringBuffer 和 StringBuilder 特殊， ==和 equal 都是比较地址

## 1.27 异常

### Java 中的两种异常类型是什么？他们有什么区别？

答：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。

### throw 和 throws 有什么区别？

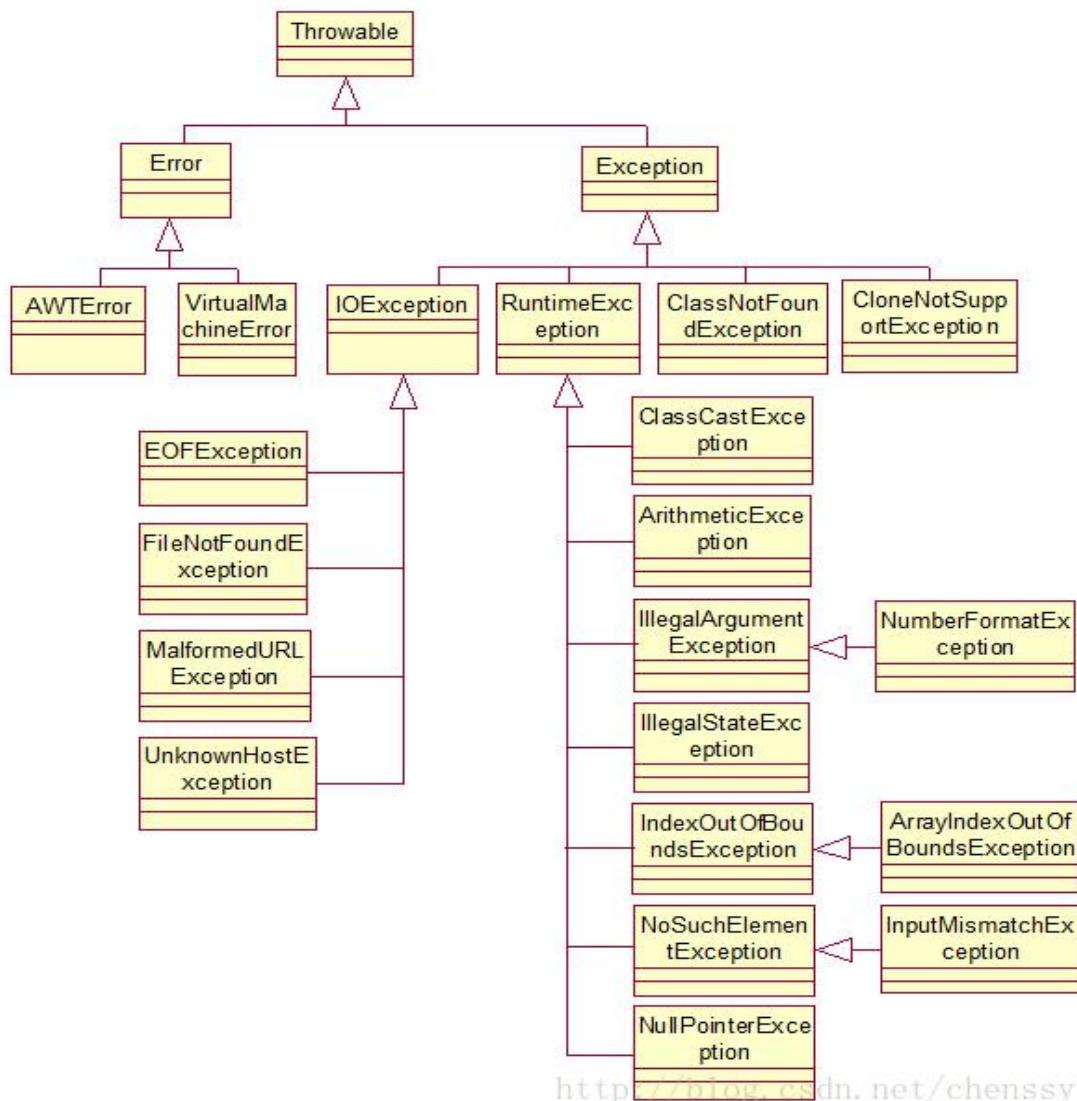
throw 关键字用来在程序中明确的抛出异常，相反， throws 语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理，所以方法的调用者才能够确保处理可能发生的异常，多个异常是用逗号分隔的。

java 提供了两种异常机制。一种是运行时异常(RuntimeException)，一种是检查式异常(checked exception)。

检查式异常：我们经常遇到的 IO 异常及 sql 异常就属于检查式异常。对于这种异常， java 编译器要求我们必须对出现的这些异常进行 catch 所以 面对这种异常不管我们是否愿意，只能自己去写一堆 catch 来捕捉这些异常。

运行时异常(未受检)：我们可以不处理。当出现这样的异常时，总是由虚拟机接管。比如：我们从来没有人去处理过 NullPointerException 异常，它就是运行时异常，并且这种异常还是最常见的异常之一。

- Throwable 是 java 语言中所有错误和异常的超类（万物即可抛）。它有两个子类：Error、Exception。
- 异常种类
  - Error : Error 为错误，是程序无法处理的，如 OutOfMemoryError、ThreadDeath 等，出现这种情况你唯一能做的是听之任之，交由 JVM 来处理，不过 JVM 在大多数情况下会选择终止线程。
  - Exception : Exception 是程序可以处理的异常。它又分为两种 CheckedException ( 受检异常 )，一种是 UncheckedException ( 不受检异常 )。
    - CheckedException 发生在编译阶段，必须要使用 try...catch ( 或者 throws ) 否则编译不通过。
    - UncheckedException 发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的，难以排查，我们一般都需要纵观全局才能够发现这类的异常错误，所以在程序设计中我们需要认真考虑，好好写代码，尽量处理异常，即使产生了异常，也能尽量保证程序朝着有利方向发展。
- 常见异常的基类
  - IOException
  - RuntimeException
- 常见的异常



<http://blog.csdn.net/chenssy>

ClassCastException(类转换异常)

IndexOutOfBoundsException(数组越界)

NullPointerException(空指针)

ArrayStoreException(数据存储异常，操作数组时类型不一致)

未检查异常和已检查异常

Java 的异常处理机制，说说受检异常和运行时异常的区别并举例

**Java 异常体系描述一下**

## 1.28 序列化

- Java序列化定义
  - 将那些实现了Serializable接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象，序列化可以弥补不同操作系统之间的差异。
- Java序列化的作用
  - Java远程方法调用（RMI）
  - 对JavaBeans进行序列化
- 如何实现序列化和反序列化
  - 实现序列化方法
    - 实现Serializable接口
      - 该接口只是一个可序列化的标志，并没有包含实际的属性和方法。
      - 如果不在改方法中添加readObject()和writeObject()方法，则采取默认的序列化机制。如果添加了这两个方法之后还想利用Java默认的序列化机制，则在这两个方法中分别调用defaultReadObject()和defaultWriteObject()两个方法。
      - 为了保证安全性，可以使用transient关键字进行修饰不必序列化的属性。因为在反序列化时，private修饰的属性也能发查看到。
    - 实现ExternalSerializable方法
      - 自己对要序列化的内容进行控制，控制哪些属性能被序列化，那些不能被序列化。
  - 反序列化
    - 实现Serializable接口的对象在反序列化时不需要调用对象所在类的构造方法，完全基于字节。
    - 实现ExternalSerializable接口的方法在反序列化时会调用构造方法。
  - 注意事项
    - 被static修饰的属性不会被序列化
    - 对象的类名、属性都会被序列化，方法不会被序列化
    - 要保证序列化对象所在类的属性也是可以被序列化的
  - 当通过网络、文件进行序列化时，必须按照写入的顺序读取对象。
  - 反序列化时必须有序列化对象时的class文件
  - 最好显式的声明serializableID，因为在不同的JVM之间，默认生成serializableID可能不同，会造成反序列化失败。
- 常见的序列化协议有哪些
  - COM主要用于Windows平台，并没有真正实现跨平台，另外COM的序列化的原理利用了编译器中虚表，使得其学习成本巨大。
  - CORBA是早期比较好的实现了跨平台，跨语言的序列化协议。CORBA的主要问题是参与方过多带来的版本过多，版本之间兼容性较差，以及使用复杂晦涩。
  - XML&SOAP
    - XML是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点。
    - SOAP (Simple Object Access protocol) 是一种被广泛应用的，基于XML为序列化和反序列化协议的结构化消息传递协议。SOAP具有安全、可扩展、跨语言、跨平台并支持多种传输层协议。
  - JSON (Javascript Object Notation)
    - 这种Associative array格式非常符合工程师对对象的理解。
    - 它保持了XML的人眼可读 (Human-readable) 的优点。
    - 相对于XML而言，序列化后的数据更加简洁。
    - 它具备Javascript的先天性支持，所以被广泛应用于Web browser的应用场景中，是Ajax的事实标准协议。
    - 与XML相比，其协议比较简单，解析速度比较快。
    - 松散的Associative array使得其具有良好的可扩展性和兼容性。
  - Thrift是Facebook开源提供的一个高性能，轻量级RPC服务框架，其产生正是为了满足当前大数据量、分布式、跨语言、跨平台数据通讯的需求。Thrift在空间开销和解析性能上有了比较大的提升，对于对性能要求比较高的分布式系统，它是一个优秀的RPC解决方案；但是由于Thrift的序列化被嵌入到Thrift框架里面，Thrift框架本身并没有透出序列化和反序列化接口，这导致其很难和其他传输层协议共同使用

- Protobuf具备了优秀的序列化协议的所需的众多典型特征
  - 标准的IDL和DL编译器，这使得其对工程师非常友好。
  - 序列化数据非常简洁，紧凑，与XML相比，其序列化之后的数据量约为1/3到1/10。
  - 解析速度非常快，比对应的XML快约20-100倍。
  - 提供了非常好的动态库，使用非常简介，反序列化只需要一行代码。由于其解析性能高，序列化后数据量相对少，非常适合应用层对象的持久化场景
- Avro的产生解决了JSON的冗长和没有IDL的问题，Avro属于Apache Hadoop的一个子项目。Avro提供两种序列化格式：JSON格式或者Binary格式。Binary格式在空间开销和解析性能方面可以和Protobuf媲美，JSON格式方便测试阶段的调试。适合于高性能的序列化服务。
- 几种协议的对比
  - XML序列化（Xstream）无论在性能和简洁性上比较差；
  - Thrift与Protobuf相比在时空开销方面都有一定的劣势；
  - Protobuf和Avro在两方面表现都非常优越。

序列化是将对象编码成字节流以及从字节流中重新构建对象的操作

序列化实现原理：

`Serializable` 该接口没有方法，只是一种标记，序列化传输时使用 `writeObject` 和 `readObject` 并通过反射调用 `objectInputStream` 和 `objectOutputStream`，如果没有设置 `Serializable` 标志，则报错

序列化实现 `Serializable` 接口

1. 序列化只保存对象的状态，不保存对象的方法
2. 父类实现序列化，子类自动实现序列化，不需要显式实现 `Serializable` 接口
3. 当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；
4. 并非所有的对象都可以序列化，至于为什么不可以，有很多原因了，比如：
  1. 安全方面的原因，比如一个对象拥有 `private`, `public` 等 `field`，对于一个要传输的对象，比如写到文件，或者进行 `rmi` 传输 等等，在序列化进行传输的过程中，这个对象的 `private` 等域是不受保护的。
  2. 资源分配方面的原因，比如 `socket`, `thread` 类，如果可以序列化，进行传输或者保存，也无法对他们进行重新的资源分配，而且，也是没有必要这样实现。
5. 序列化会忽略静态变量，即序列化不保存静态变量的状态，`transient`，所以不能序列化静态成员属于类级别的，所以不能序列化。即 序列化的是对象的状态不是类的状态。
6. 序列化前和序列化后的对象的关系是 “==” 还是 `equal?` or 是浅复制还是深复制？（深复制、`equal`）

序列化，以及 json 传输

## 1.30 comparable 接口和 comparator 接口

comparable 接口和 comparator 接口实现比较的区别和用法

### 1. 定义

`Comparable` 接口：使用 `Array` 或 `Collection` 的排序方法时，自定义类需要实现 Java 提供 `Comparable` 接口的 `compareTo(T obj)` 方法，它被排序方法所使用，应该重写这个方法，如果“`this`”对象比传递的对象参数更小、相等或更大时，它返回一个负整数、0 或正整数。

使用 Comparator 接口的情景：在大多数实际情况下，我们想根据不同参数进行排序。比如，作为一个 CEO，我想对雇员基于薪资进行排序，一个 HR 想基于年龄对他们进行排序。这就是我们需要使用 Comparator 接口的情景。因为 Comparable.compareTo(Object o)方法实现只能基于一个字段进行排序，不能根据需要选择对象字段来对对象进行排序。

Comparator 接口：可以实现两个对象的特定字段的比较（比如，比较员工这个对象的年龄），该接口的 compare(Object o1, Object o2) 方法的实现需要传递两个对象参数，若第一个参数小于、等于、大于第二个参数，返回负整数、0、正整数。

## 2. comparable 接口和 comparator 接口区别

Comparable 和 Comparator 接口被用来对对象集合或者数组进行排序。

Comparable 接口被用来提供对象的自然排序，可使用它来提供基于单个逻辑的排序。

Comparator 接口被用来提供不同的排序算法，可根据制定字段选择需要使用的 Comparator 来对指定的对象集合进行排序。

## 1.33 接口和抽象类

### 1. 接口和抽象类的区别

- 1, 抽象类里可以有构造方法，而接口内不能有构造方法。
- 2, 抽象类中可以有普通成员变量，而接口中不能有普通成员变量。
- 3, 抽象类中可以包含非抽象的普通方法，而接口中所有的方法必须是抽象的，不能有非抽象的普通方法。
- 4, 抽象类中的抽象方法的访问类型可以是 public , protected 和 private , 但接口中的抽象方法只能是 public 类型的，并且默认即为 public abstract 类型。
- 5, 抽象类中可以包含静态方法，接口内不能包含静态方法。
- 6, 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是 public static 类型，并且默认为 public static final 类型。
- 7, 一个类可以实现多个接口，但只能继承一个抽象类。

### 1. Java 抽象类可以有构造函数吗？

可以有，抽象类可以声明并定义构造函数。因为你不可以创建抽象类的实例，所以构造函数只能通过构造函数链调用（Java 中构造函数链指的是从其他构造函数调用一个构造函数），例如，当你创建具体的实现类。现在一些面试官问，如果你不能对抽象类实例化那么构造函数的作用是什么？好吧，它可以用来初始化抽象类内部声明的通用变量，并被各种实现使用。另外，即使你没有提供任何构造函数，编译器将为抽象类添加默认的无参数

的构造函数，没有的话你的子类将无法编译，因为在任何构造函数中的第一条语句隐式调用 super()，Java 中默认超类的构造函数。

### 2. Java 抽象类可以实现接口吗？它们需要实现所有的方法吗？

可以，抽象类可以通过使用关键字 implements 来实现接口。因为它们是抽象的，所以它们不需要实现所有的方法。好的做法是，提供一个抽象基类以及一个接口来声明类型。这样的例子是，java.util.List 接口和相应的 java.util.AbstractList 抽象类。因为 AbstractList 实现了所有的通用方法，具体的实现像 LinkedList 和 ArrayList 不受实现所有方法的负担，它们可以直接实现 List 接口。这对两方面都很好，你可以利用接口声明类型的优点和抽象类的灵活性在一个地方实现共同的行为。Effective Java 有个很好的章节，介绍如何使用 Java 的抽象类和接口，值得阅读。

### 3. Java 抽象类可以是 final 的吗？

不可以，Java 抽象类不能是 final 的。将它们声明为 final 的将会阻止它们被继承，而这正是使用抽象类唯一的方法。它们也是彼此相反的，关键字 abstract 强制继承类，而关键字 final 阻止类被扩张。在现实世界中，抽象表示不完备性，而 final 是用来证明完整性。底线是，你不能让你的 Java 类既 abstract 又 final，同时使用，是一个编译时错误。

### 4. Java 抽象类可以有 static 方法吗？

可以，抽象类可以声明并定义 static 方法，没什么阻止这样做。但是，你必须遵守 Java 中将方法声明为 static 的准则，

### 5. 可以创建抽象类的实例吗？

不可以，你不能创建 Java 抽象类的实例，它们是不完全的。即使你的抽象类不包含任何抽象方法，你也不能对它实例化。将类声明为 abstract 的，就等同于你告诉编译器，它是不完全的不应该被实例化。当一段代码尝试实例化一个抽象类时 Java 编译器会抛错误。

### 6. 抽象类必须有抽象方法吗？

不需要，抽象类有抽象方法不是强制性的。你只需要使用关键字 abstract 就可以将类声明为抽象类。编译器会强制所有结构的限制来适用于抽象类，例如，现在允许创建一些实例。是否在抽象类中有抽象方法是引起争论的。我的观点是，抽象类应该有抽象方法，因为这是当程序员看到那个类并做假设的第一件事。这也符合最小惊奇原则。

### 8. 何时选用抽象类而不是接口？

这是对之前抽象类和接口对比问题的后续。如果你知道语法差异，你可以很容易回答这个问题，因为它们可以令你做出抉择。当关心升级时，因为不可能在一个发布的接口中添加一个新方法，用抽象类会更好。类似地，如果你的接口中有很多方法，你对它们的实现感到很头疼，考虑提供一个抽象类作为默认实现。这是 Java 集合包中的模式，你可以使用提供默认实现 List 接口的 AbstractList。

### 9. Java 中的抽象方法是什么？

抽象方法是一个没有方法体的方法。你仅需要声明一个方法，不需要定义它并使用关键字 abstract 声明。Java 接口中所有方法的声明默认是 abstract 的。这是抽象方法的例子

```
public void abstract printVersion();
```

现在，为了实现这个方法，你需要继承该抽象类并重载这个方法。

### 10. Java 抽象类中可以包含 main 方法吗？

是的，抽象类可以包含 main 方法，它只是一个静态方法，你可以使用 main 方法执行抽象类，但不可以创建任何实例。

## 1.34 Socket

Socket 通信具体原理

## 1.35 Runtime 类

Runtime:运行时，是一个封装了 JVM 的类。每一个 JAVA 程序实际上都是启动了一个 JVM 进程，每一个 JVM 进程都对应一个 Runtime 实例，此实例是由 JVM 为其实例化的。所以我们不能实例化一个 Runtime 对象，应用程序也不能创建自己的 Runtime 类实例，但可以通过 getRuntime 方法获取当前 Runtime 运行时对象的引用。一旦得到了一个当前的 Runtime 对象的引用，就可以调用 Runtime 对象的方法去控制 Java 虚拟机的状态和行为。

查看官方文档可以看到，Runtime 类中没有构造方法，本类的构造方法被私有化了，所以才会有 getRuntime 方法返回本来的实例化对象，这与单例设计模式不谋而合

```
public static Runtime getRuntime()
```

直接使用此静态方法可以取得 Runtime 类的实例

## 1.36 值传递与引用传递

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。一般认为，java 内的传递都是值传递。java 中实例对象的传递是引用传递

## 1.37 泛型 ? 与 T 的区别

<https://blog.csdn.net/woshizisezise/article/details/79374460>

```
public static <T> void show1(List<T> list) {
    for (Object object : list) {
        System.out.println(object.toString());
    }
}

public static void show2(List<?> list) {
    for (Object object : list) {
        System.out.println(object);
    }
}

public static void test() {
    List<Student> list1 = new ArrayList<>();
    list1.add(new Student("zhangsan", 18, 0));
    list1.add(new Student("lisi", 28, 0));
    list1.add(new Student("wangwu", 24, 1));
    //这里如果 add(new Teacher(...)); 就会报错，因为我们已经给 List 指定了数据类型为 Student
    show1(list1);
}
```

```
System.out.println("*****分割线*****");
```

```
//这里我们并没有给 List 指定具体的数据类型，可以存放多种类型数据
List list2 = new ArrayList<>();
list2.add(new Student("zhaoliu",22,1));
list2.add(new Teacher("sunba",30,0));
show2(list2);
}
```

从 show2 方法可以看出和 show1 的区别了，list2 存放了 Student 和 Teacher 两种类型，同样可以输出数据，所以这就是 T 和?的区别啦

## 1.38 枚举类型字节码层面理解 Enum

<https://blog.csdn.net/javazejian/article/details/71333103>

```
1 /**
2  * Created by zejian on 2017/5/7.
3  * Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]
4 */
5 public class EnumDemo {
6
7     public static void main(String[] args){
8         //直接引用
9         Day day =Day.MONDAY;
10    }
11
12 }
13 //定义枚举类型
14 enum Day {
15     MONDAY, TUESDAY, WEDNESDAY,
16     THURSDAY, FRIDAY, SATURDAY, SUNDAY
17 }
//反编译 Day.class
final class Day extends Enum
{
    //编译器为我们添加的静态的 values()方法
    public static Day[] values()
    {
        return (Day[])$VALUES.clone();
    }
    //编译器为我们添加的静态的 valueOf()方法，注意间接调用了 Enum 也类的 valueOf
    //方法
    public static Day valueOf(String s)
    {
        return (Day)Enum.valueOf(com/zejian/enumdemo/Day, s);
    }
    //私有构造函数
    private Day(String s, int i)
    {
        super(s, i);
    }
}
```

```
}

//前面定义的 7 种枚举实例
public static final Day MONDAY;
public static final Day TUESDAY;
public static final Day WEDNESDAY;
public static final Day THURSDAY;
public static final Day FRIDAY;
public static final Day SATURDAY;
public static final Day SUNDAY;
private static final Day $VALUES[];

static
{
    //实例化枚举实例
    MONDAY = new Day("MONDAY", 0);
    TUESDAY = new Day("TUESDAY", 1);
    WEDNESDAY = new Day("WEDNESDAY", 2);
    THURSDAY = new Day("THURSDAY", 3);
    FRIDAY = new Day("FRIDAY", 4);
    SATURDAY = new Day("SATURDAY", 5);
    SUNDAY = new Day("SUNDAY", 6);
    $VALUES = (new Day[] {
        MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
    });
}
```

### 1.39 java 注解类型

- @Override : 用于标明此方法覆盖了父类的方法，源码如下

```

1 @Target(ElementType.METHOD)
2 @Retention(RetentionPolicy.SOURCE)
3 public @interface Override {
4 }

```

- @Deprecated : 用于标明已经过时的方法或类，源码如下，关于@Documented稍后分析：

```

1 @Documented
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE})
4 public @interface Deprecated {
5 }

```

- @SuppressWarnings: 用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告，其实现源码如下：

```

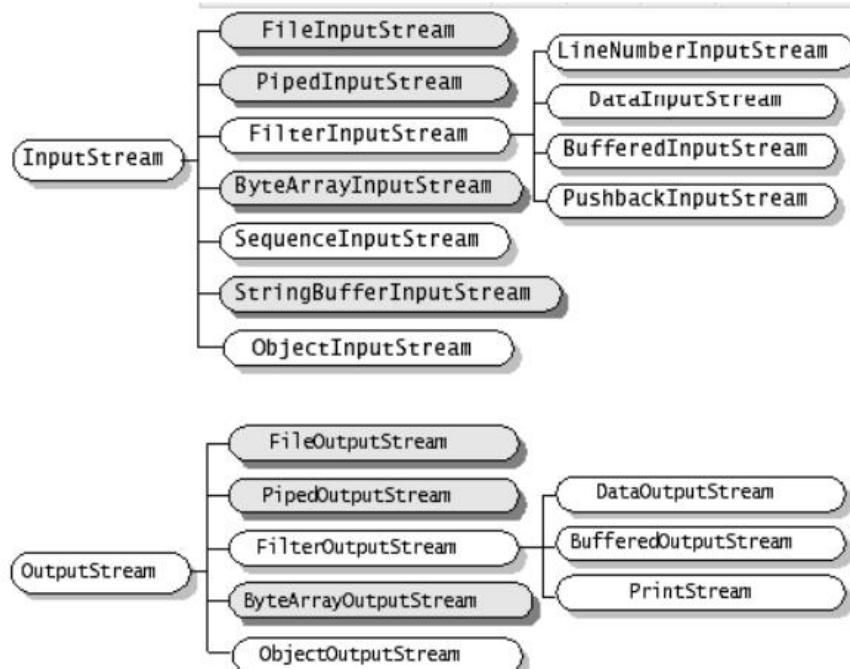
1 @Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
2 @Retention(RetentionPolicy.SOURCE)
3 public @interface SuppressWarnings {
4     String[] value();
5 }

```

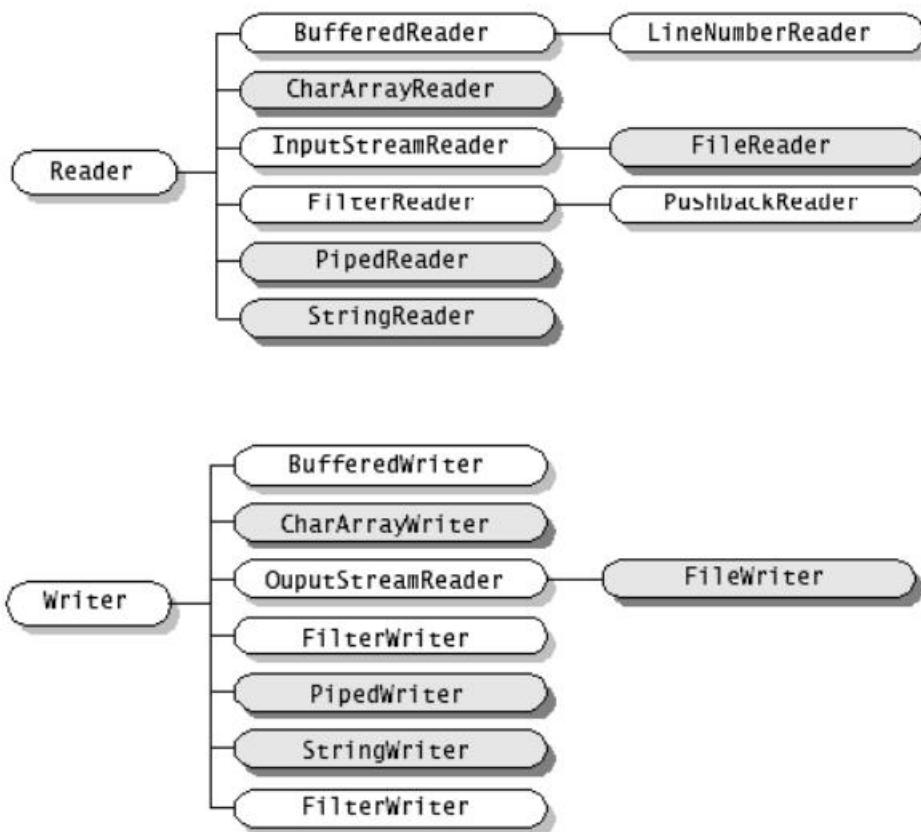
## 1.40 字节流 字符流

<https://www.cnblogs.com/huanglitong/p/5746950.html>

### 字节流:



## 字符流：



InputStreamReader 类是从字节流到字符流的桥梁：它读入字节，并根据指定的编码方式，将之转换为字符流。

### 字节流与字符流的区别

字节流和字符流使用是非常相似的，那么除了操作代码的不同之外，还有哪些不同呢？

#### 区别：

- 1、字节流在操作的时候本身是不会用到缓冲区（内存）的，是与文件本身直接操作的，而字符流在操作的时候是使用到缓冲区的
- 2、字节流在操作文件时，即使不关闭资源（close方法），文件也能输出，但是如果字符流不使用close方法的话，则不会输出任何内容，说明字符流用的是缓冲区，并且可以使用flush方法强制进行刷新缓冲区，这时才能在不close的情况下输出内容
- 3、Reader类的read()方法返回类型为int：作为整数读取的字符（占两个字节共16位），范围在0 到 65535 之间（0x00-0xffff），如果已到达流的末尾，则返回 -1  
InputStream的read()虽然也返回int，但由于此类是面向字节流的，一个字节占8个位，所以返回0 到 255 范围内的 int 字节值。如果因为已经到达流末尾而没有可用的字节，则返回值 -1。因此对于不能用0-255来表示的值就得用字符流来读取！比如说汉字。
- 4、字节流与字符流主要的区别是他们的的处理方式

字节流：处理字节和字节数组或二进制对象；

字符流：处理字符、字符数组或字符串。

## 1.41 静态内部类 匿名类

如果你不需要内部类对象与其外围类对象之间有联系，那你可以将内部类声明为 `static`。这通常称为嵌套类（nested class）。Static Nested Class 是被声明为静态（`static`）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化。想要理解 `static` 应用于内部类时的含义，你就必须记住，普通的内部类对象隐含地保存了一个引用，指向创建它的外围类对象。然而，当内部类是 `static` 的时，就不是这样了。嵌套类意味着：

1. 嵌套类的对象，并不需要其外围类的对象。
2. 不能从嵌套类的对象中访问非静态的外围类对象。

如下所示代码为定义一个静态嵌套类

```
public class StaticTest{  
    private static String name = "woobo";  
    private String num = "X001";  
    static class Person{ // 静态内部类可以用 public,protected,private 修饰  
        // 静态内部类中可以定义静态或者非静态的成员  
        private String address = "China";  
  
        Private Static String x="as";  
        public String mail = "kongbowoo@yahoo.com.cn";//内部类公有成员  
        public void display(){  
            //System.out.println(num);//不能直接访问外部类的非静态成员  
  
        // 静态内部类不能访问外部类的非静态成员(包括非静态变量和非静态方法)  
        System.out.println(name);//只能直接访问外部类的静态成员  
  
        //静态内部类只能访问外部类的静态成员(包括静态变量和静态方法)  
        System.out.println("Inner " + address);//访问本内部类成员。  
    }  
}  
    public void printInfo(){  
        Person person = new Person();  
  
        // 外部类访问内部类的非静态成员:实例化内部类即可  
  
        person.display();  
  
        //System.out.println(mail);//不可访问  
        //System.out.println(address);//不可访问  
        System.out.println(person.address);//可以访问内部类的私有成员  
  
        System.out.println(Person.x);// 外部类访问内部类的静态成员：内部类.静态成员  
        System.out.println(person.mail);//可以访问内部类的公有成员  
    }  
    public static void main(String[] args){
```

```
    StaticTest staticTest = new StaticTest();
    staticTest.printInfo();
}
}
```

在静态嵌套类内部，不能访问外部类的非静态成员，这是由 Java 语法中“静态方法不能直接访问非静态成员”所限定。注意，外部类访问内部类的成员有些特别，不能直接访问，但可以通过内部类实例来访问，这是因为静态嵌套内的所有成员和方法默认为静态的了。同时注意，**内部静态类 Person 只在类 StaticTest 范围内可见**，若在其它类中引用或初始化，均是错误的。

- 一 . 静态内部类可以有静态成员，而非静态内部类则不能有静态成员。
- 二 . 静态内部类的非静态成员可以访问外部类的静态变量，而不可访问外部类的非静态变量；
- 三 . 非静态内部类的非静态成员可以访问外部类的非静态变量。

四. 在非静态内部类中不可以声明静态成员，一般的非静态内部类，可以随意的访问外部类中的成员变量与成员方法。即使这些成员方法被修饰为 **private**(私有的成员变量或者方法)，其非静态内部类都可以随意的访问。则是非静态内部类的特权。不能够从静态内部类的对象中访问外部类的非静态成员(包括成员变量与成员方法)。

生成一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成：`Outer.Inner in = new Outer.Inner();`而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类(正常情况下，你不能在接口内部放置任何代码，但嵌套类可以作为接口的一部分，因为它是 **static** 的。只是将嵌套类置于接口的命名空间内，这并不违反接口的规则)

### 1.41.2 匿名类

匿名内部类就是没有名字的内部类；

匿名内部类不能定义任何静态成员、方法。

匿名内部类中的方法不能是抽象的；

匿名内部类必须实现接口或抽象父类的所有抽象方法。

匿名内部类访问的外部类成员变量或成员方法必须用 **static** 修饰

- 1、匿名内部类因为没有类名，可知**匿名内部类不能定义构造器。**

2、因为在创建匿名内部类的时候，会立即创建它的实例，可知**匿名内部类不能是抽象类，必须实现**

**接口或抽象父类的所有抽象方法。**

3、匿名内部类会继承一个父类（有且只有一个）或实现一个接口（有且只有一个），实现父类或接口中所有抽象方法，可以改写父类中的方法，添加自定义方法。

5、当匿名内部类和外部类有同名变量（方法）时，默认访问的是匿名内部类的变量（方法），要访问外部类的变量（方法）则需要加上外部类的类名。

Java I/O 底层细节，注意是底层细节，而不是怎么用

如何实现分布式缓存

浏览器的缓存机制

JVM tomcat 容器启动，jvm 加载情况描述

当获取第一个获取锁之后，条件不满足需要释放锁应当怎么做？

HashMap 实现原理，扩容因子过大过小的缺点，扩容过程 采用什么方法能保证每个 bucket 中的数据更均匀 解决冲突的方式，还有没有其他方式（全域哈希）

Collection 集合类中只能在 Iterator 中删除元素的原因

java 地址和值传递的例子

java NIO，java 多线程、线程池，java 网络编程解决并发量，

java 的 I/O

手写一个线程安全的生产者与消费者。

ConcurrentHashMap 和 LinkedHashMap 差异和适用情形

ConcurrentHashMap 分段锁是如何实现的

ConcurrentHashmap jdk1.8 访问的时候是怎么加锁的，插入的时候是怎么加锁的 访问不加锁插入的时候对头结点加锁

ArrayDeque 的使用场景

JDBC 连接的过程 手写 jdbc 连接过程

可重入锁，实现原理

Java IO 模型(BIO,NIO 等) Tomcat 用的哪一种模型

ArrayBlockingQueue 源码

多进程和多线程的区别

说出三个遇到过的程序报异常的情况

Java 无锁原理

hashmap 和 treemap 的区别

rehash 过程

网络编程的 accept 和 connect，

HashMap 的负载因子

.try catch finally 可不可以没有 catch (try return,finally return)  
mapreduce 流程 如何保证 reduce 接受的数据没有丢失，数据如何去重 mapreduce 原理，  
partition 发生在什么阶段  
直接写一个 java 程序，统计 IP 地址的次数  
讲讲多线程，多线程的同步方法  
**list, map, set** 之间的区别  
socket 是靠什么协议支持的  
java io 用到什么设计模式  
**serviable** 的序列化，其中 **uuid** 的作用  
什么时候会用到 **HashMap**  
什么情景下会用到反射（注解、Spring 配置文件、动态代理）  
浅克隆与深克隆有什么区别  
如何实现深克隆  
常见的线程安全的集合类  
Java 8 函数式编程  
回调函数，函数式编程，面向对象之间区别  
Java 8 中 **stream** 迭代的优势和区别？  
同步等于可见性吗？保证了可见性不等于正确同步，因为还有原子性没考虑。  
还了解除 util 其他包下的 List 吗？**CopyOnWriteArrayList**  
反射能够使用私有的方法属性吗和底层原理？  
处理器指令优化有些什么考虑？ 禁止重排序  
**object** 对象的常用方法  
**Stack** 和 **ArrayList** 的区别  
**statement** 和 **prestatement** 的区别  
手写模拟实现一个阻塞队列  
怎么使用父类的方法  
util 包下有哪几种接口  
**cookie** 禁用怎么办  
**Netty**  
new 实例化过程  
**socket** 实现过程，具体用的方法；怎么实现异步 **socket**。  
很常见的 **NullPointerException**，你是怎么排查的，怎么解决的；  
**Binder** 的原理  
C++/JAVA/C 的区别  
java 线程安全都体现在哪些方面，如果维护线程安全  
如果想实现一个线程安全的队列，可以怎么实现？  
JUC 包里的 **ArrayBlockingQueue** 还有 **LinkedBlockingQueue** 啥的又结合源码说了一通。  
静态内部类和非静态内部类的区别是什么？怎么创建静态内部类和非静态内部类？  
。  
断点续传的原理  
Xml 解析方式，原理优缺点  
数据缓存处理  
四大组件生命周期

Java 和 JavaScript 的区别

信号量

静态变量和全局变量的区别

## 二 . 集合类 Set

说出所有 java 集合类

### 2.1 HashMap

<https://blog.csdn.net/jiary5201314/article/details/51439982>

#### (1) hashMap 的原理

hashmap 是数组和链表的结合体，数组每个元素存的是链表的头结点。往 hashmap 里面放键值对的时候先得到 key 的 hashCode，然后重新计算 hashCode，（让 1 分布均匀因为如果分布不均匀，低位全是 0，则后来计算数组下标的时候会冲突），然后与 length-1 按位与，计算数组出数组下标。如果该下标对应的链表为空，则直接把键值对作为链表头结点，如果不为空，则遍历链表看是否有 key 值相同的，有就把 value 替换，没有就把该对象最为链表的第一个节点，原有的节点最为他的后续节点。

#### 2. hashCode 的计算

<https://www.zhihu.com/question/20733617/answer/111577937>

<https://blog.csdn.net/justloveyou/article/details/62893086>

Key.hashCode 是 key 的自带的 hashCode 函数是一个 int 值 32 位

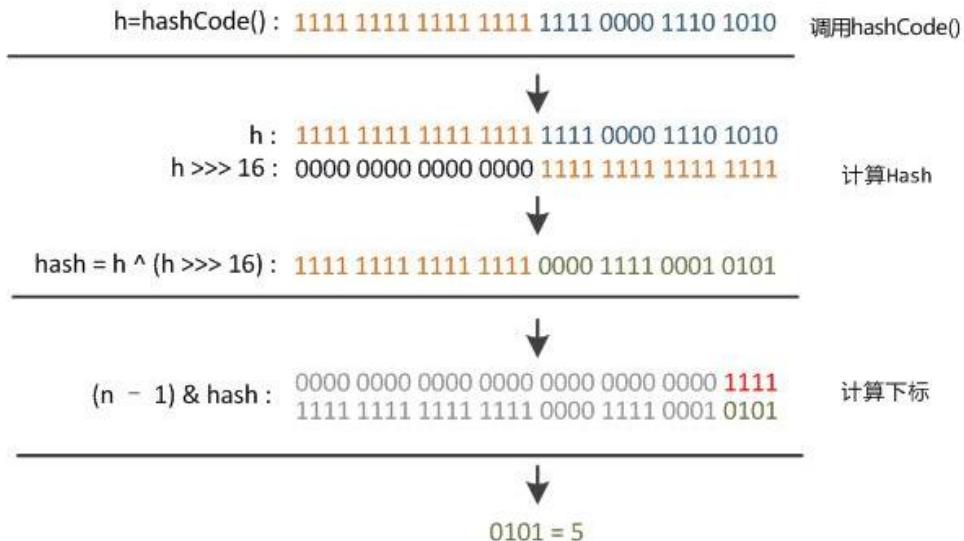
关于 hashCode 计算 >>>

hashmap jdk1.8 中 hashmap 重计算 hashCode 方法改动：高 16 位异或低 16 位

```
return (key == null) ? 0 : h = key.hashCode() ^ (h >>> 16);
```

首先确认：当 length 总是 2 的 n 次方时， $h \& (length - 1)$  等价于 hash 对 length 取模，但是 & 比 % 具有更高的效率；

Jdk1.7 之前： $h \& (length - 1)$ ; // 第三步，取模运算



右位移16位，正好是32bit的一半，自己的高半区和低半区做异或，就是为了混合原始哈希码的高位和低位，以此来加大低位的随机性。而且混合后的低位掺杂了高位的部分特征，这样高位的信息也被变相保留下来。

## (2) hashMap 参数以及扩容机制

初始容量 16，达到阀值扩容，阀值等于最大容量\*负载因子，扩容每次 2 倍，总是 2 的 n 次方

扩容机制：

使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将原有 Entry 数组的元素拷贝到新的 Entry 数组里，Java1.重新计算每个元素在数组中的位置。Java1.8 中不是重新计算，而是用了一种更巧妙的方式。

## (3) get

因为整个过程都不需要加锁

```
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    //先定位到数组元素，再遍历该元素处的链表
    for (Entry<K,V> e = table[indexFor(hash, table.length)];  
         e != null;  
         e = e.next) {  
        Object k;  
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}
```

## (4) HashMap 的 put 方法源码

这里 HashMap 里面用到链式数据结构的一个概念。上面我们提到过 Entry 类里面有一个 next 属性，作用是指向下一个 Entry。打个比方，第一个键值对 A 进来，通过计算其 key 的 hash 得到的 index=0，记做:Entry[0] = A。一会后又进来一个键值对 B，

通过计算其 index 也等于 0, 现在怎么办? HashMap 会这样做:B.next = A,Entry[0] = B, 如果又进来 C,index 也等于 0,那么 C.next = B,Entry[0] = C; 这样我们发现 index=0 的地方其实存取了 A,B,C 三个键值对,他们通过 next 这个属性链接在一起。所以疑问不用担心。也就是说数组中存储的是最后插入的元素。到这里为止, HashMap 的大致实现。

```
public V put(K key, V value) {  
    if (key == null)  
        return putForNullKey(value); //null总是放在数组的第一个链表中  
    int hash = hash(key.hashCode());  
    int i = indexFor(hash, table.length);  
    //遍历链表  
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
        Object k;  
        //如果key在链表中已存在，则替换为newValue  
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
            V oldValue = e.value;  
            e.value = value;  
            e.recordAccess(this);  
            return oldValue;  
        }  
    }  
    modCount++;  
    addEntry(hash, key, value, i);  
    return null;  
}  
  
void addEntry(int hash, K key, V value, int bucketIndex) {  
    Entry<K,V> e = table[bucketIndex];  
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e); //参数e, 是Entry.next  
    //如果size超过threshold，则扩充table大小。再散列  
    if (size++ >= threshold)  
        resize(2 * table.length);  
}
```

## (5) HashMap 问题 jdk1.8 优化

(1) HashMap 如果有很多相同 key, 后面的链很长的话, 你会怎么优化? 或者你会用什么数据结构来存储? 针对 HashMap 中某个 Entry 链太长, 查找的时间复杂度可能达到  $O(n)$ , 怎么优化?

Java8 做的改变:

1.HashMap 是数组+链表+红黑树 (JDK1.8 增加了红黑树部分), 当链表长度 $\geq 8$  时转化为红黑树

在 JDK1.8 版本中, 对数据结构做了进一步的优化, 引入了红黑树。而当链表长度太长(默认超过 8) 时, 链表就转换为红黑树, 利用红黑树快速增删改查的特点提高 HashMap 的性能, 其中会用到红黑树的插入、删除、查找等算法。

1. **java8 中对 hashmap 扩容不是重新计算所有元素在数组的位置**, 而是我们使用的是 2 次幂的扩展(指长度扩为原来 2 倍), 所以, 元素的位置要么是在原位置, 要么是在原位置再移动 2 次幂的位置在扩充 HashMap 的时候, 不需要像 JDK1.7 的实现那样重新计算 hash, 只需要看看原来的 hash 值新增的那个 bit 是 1 还是 0 就好了, 是 0 的话索引没变, 是 1 的话索引变成“原索引+oldCap”。
2. **HashMap 高并发情况下会出现什么问题?** 扩容问题
3. **HashMap 的存放自定义类时, 需要实现自定义类的什么方法?**  
答: **hashCode 和 equals**。通过 `hash(hashCode)` 然后模运算 (其实是与的位操作) 定位在 `Entry` 数组中的下标, 然后遍历这之后的链表, 通过 `equals` 比较有没有相同的 key, 如果有直接覆盖 value, 如果没有就重新创建一个 `Entry`。

**HashMap 为什么可以插入空值?**

**hashmap 为什么可以插入空值?**

HashMap 中添加 `key==null` 的 `Entry` 时会调用 `putForNullKey` 方法直接去遍历

`table[0]Entry` 链表, 寻找 `e.key==null` 的 `Entry` 或者没有找到遍历结束

如果找到了 `e.key==null`, 就保存 `null` 值对应的原值 `oldValue`, 然后覆盖原值, 并返回

`oldValue`

如果在 `table[0]Entry` 链表中没有找到就调用 `addEntry` 方法添加一个 `key` 为 `null` 的 `Entry`

#### 6. **Hashmap 为什么线程不安全 (hash 碰撞和扩容导致)**

**HashMap 扩容的时候可能会形成环形链表, 造成死循环。**

HashMap 底层是一个 `Entry` 数组, 当发生 `hash` 冲突的时候, `hashmap` 是采用链表的方式来解决的, 在对应的数组位置存放链表的头结点。对链表而言, 新加入的节点会从头结点加入。假如 A 线程和 B 线程同时对同一个数组位置调用 `addEntry`, 两个线程会同时得到现在的头结点, 然后 A 写入新的头结点之后, B 也写入新的头结点, 那 B 的写入操作就会覆盖 A 的写入操作造成 A 的写入操作丢失

删除键值对的代码如上: 当多个线程同时操作同一个数组位置的时候, 也都会先取得现在状态下该位置存储的头结点, 然后各自去进行计算操作, 之后再把结果写会到该数组位置去, 其实写回的时候可能其他的线程已经把这个位置给修改过了, 就会**覆盖其他线程的修改**

当多个线程同时检测到总数量超过门限值的时候就会同时调用 `resize` 操作, 各自生成新的数组并 `rehash` 后赋给该 `map` 底层的数组 `table`, 结果最终只有最后一个线程生成的新数组被赋给 `table` 变量, 其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候, 就会用已经被赋值的 `table` 作为原始数组, 这样也会有问题。

**要想实现线程安全, 那么需要调用 collections 类的静态方法**

**synchronizeMap ( ) 实现**

**7. Hashmap 中的 key 可以为任意对象或数据类型吗?**

可以为 `null`, 但不能是可变对象, 如果是可变对象的话, 对象中的属性改变, 则对象 `HashCode` 也进行相应的改变, 导

致下次无法查找到已存在Map中的数据。

- 如果可变对象在HashMap中被用作键，那就要小心在改变对象状态的时候，不要改变它的哈希值了。我们只需要保证成员变量的改变能保证该对象的哈希值不变即可。

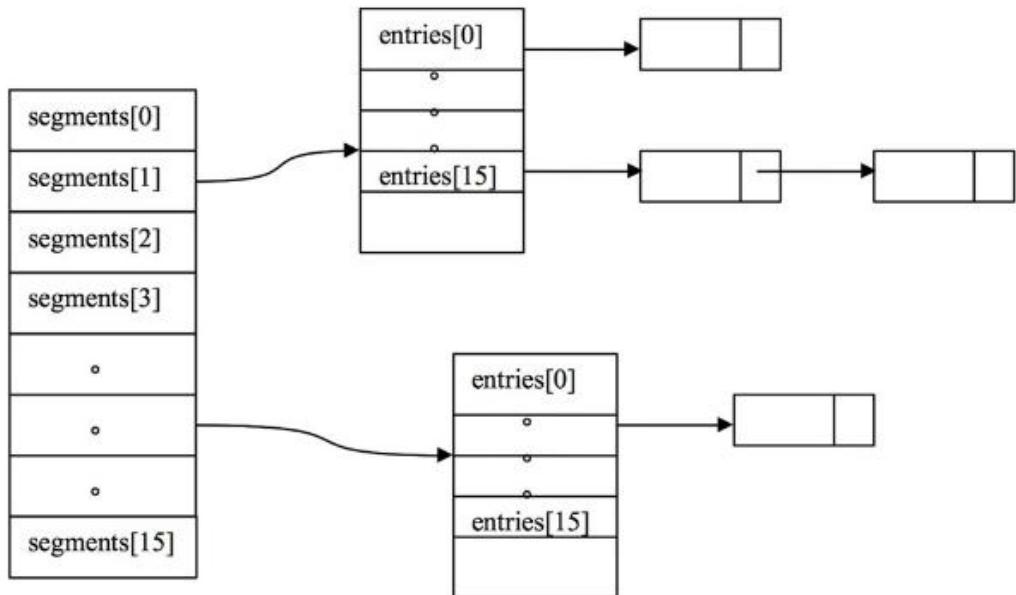
## 2.2 CurrentHashMap

<http://www.importnew.com/21781.html>

[https://blog.csdn.net/dingji\\_ping/article/details/51005799](https://blog.csdn.net/dingji_ping/article/details/51005799)

<https://www.cnblogs.com/chengxiao/p/6842045.html>

<http://ifeve.com/hashmap-concurrenthashmap-%E7%9B%B8%E4%BF%A1%E7%9C%8B%E5%AE%8C%E8%BF%99%E7%AF%87%E6%B2%A1%E4%BA%BA%E8%83%BD%E9%9A%BE%E4%BD%8F%E4%BD%A0%EF%BC%81/>



一个 ConcurrentHashMap 维护一个 Segment 数组，一个 Segment 维护一个 HashEntry 数组。

## 0. JDK1.7 ConCurrentHashMap 原理

其中 Segment 继承于 ReentrantLock

ConcurrentHashMap 使用分段锁技术，将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问，能够实现真正的并发访问。

继续看每个segment是怎么定义的：

```
1 | static final class Segment<K,V> extends ReentrantLock implements Serializable
```

Segment 继承了 ReentrantLock，表明每个 segment 都可以当做一个锁。这样对每个 segment 中的数据需要同步操作的话都是使用每个 segment 容器对象自身的锁来实现。只有对全局需要改变时锁定的是所有的 segment。

- Concurrent HashMap 线程安全吗，ConcurrentHashMap如何保证线程安全？
  - HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。
  - get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空的才会加锁重读。get方法里将要使用的共享变量都定义成volatile，如用于统计当前Segment大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。
  - Put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置然后放在HashEntry数组里。

## 2.ConcurrentHashMap Segment 内部 Get PUT REMOVE

### 1. JDK1.7 Get

#### 先看get方法

```
1 | public V get(Object key) {  
2 |     int hash = hash(key); // throws NullPointerException if key null  
3 |     return segmentFor(hash).get(key, hash);  
4 | }
```

它没有使用同步控制，交给segment去找，再看Segment中的get方法

```
1 | V get(Object key, int hash) {  
2 |     if (count != 0) { // read-volatile // ⚡  
3 |         HashEntry<K,V> e = getFirst(hash);  
4 |         while (e != null) {  
5 |             if (e.hash == hash && key.equals(e.key)) {  
6 |                 V v = e.value;  
7 |                 if (v != null) // ⚡ 注意这里  
8 |                     return v;  
9 |                 return readValueUnderLock(e); // recheck  
10 |             }  
11 |             e = e.next;  
12 |         }  
13 |     }  
14 |     return null;  
15 | }
```

ConcurrentHashMap 是否使用了锁？？？

它也没有使用锁来同步，只是判断获取的 entry 的 value 是否为 null，为 null 时才使用加锁的方式再次去获取。这里可以看出并没有使用锁，但是 value 的值为 null 时候才是使用了加锁！！！

Get 原理：

第一步，先判断一下 count != 0； count 变量表示 segment 中存在 entry 的个数。如果为 0 就不用找了。假设这个时候恰好另一个线程 put 或者 remove 了这个 segment 中的一个 entry，会不会导致两个线程看到的 count 值不一致呢？看一下 count 变量的定义： transient volatile int count；

它使用了 volatile 来修改。我们前文说过，Java5 之后，JMM 实现了对 volatile

的保证：对 volatile 域的写入操作 happens-before 于每一个后续对同一个域的读写操作。所以，每次判断 count 变量的时候，即使恰好其他线程改变了 segment 也会体现出来

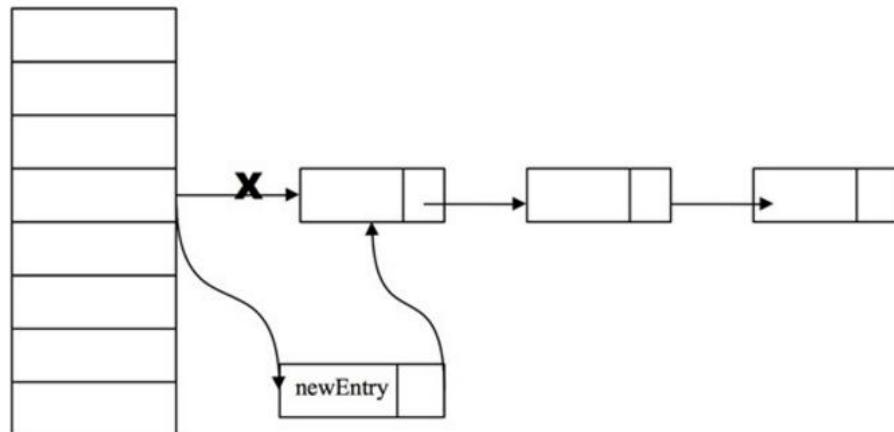
先看一下 HashEntry 类结构。

```
1 | static final class HashEntry<K,V> {
2 |     final K key;
3 |     final int hash;
4 |     volatile V value;
5 |     final HashEntry<K,V> next;
6 |     ...
7 | }
```

除了 value，其它成员都是 final 修饰的，也就是说 value 可以被改变，其它都不可以改变，包括指向下一个 HashEntry 的 next 也不能被改变。（那删除一个 entry 时怎么办？后续会讲到。）

### 1) 在 get 代码的①和②之间，另一个线程新增了一个 entry

如果另一个线程新增的这个 entry 又恰好是我们要 get 的，这事儿就比较微妙了。下图大致描述了 put 一个新的 entry 的过程。



因为每个 HashEntry 中的 next 也是 final 的，没法对链表最后一个元素增加一个后续 entry 所以新增一个 entry 的实现方式只能通过头结点来插入了。 newEntry 对象是通过 new HashEntry(K k, V v, HashEntry next) 来创建的。如果另一个线程刚好 new 这个对象时，当前线程来 get 它。因为没有同步，就可能会出现当前线程得到的 newEntry 对象是一个没有完全构造好的对象引用。如果在这个 new 的对象的后面，则完全不影响，如果刚好是这个 new 的对象，那么当刚好这个对象没有完全构造好，也就是说这个对象的 value 值为 null，就出现了如下所示的代码，需要重新加锁再次读取这个值！

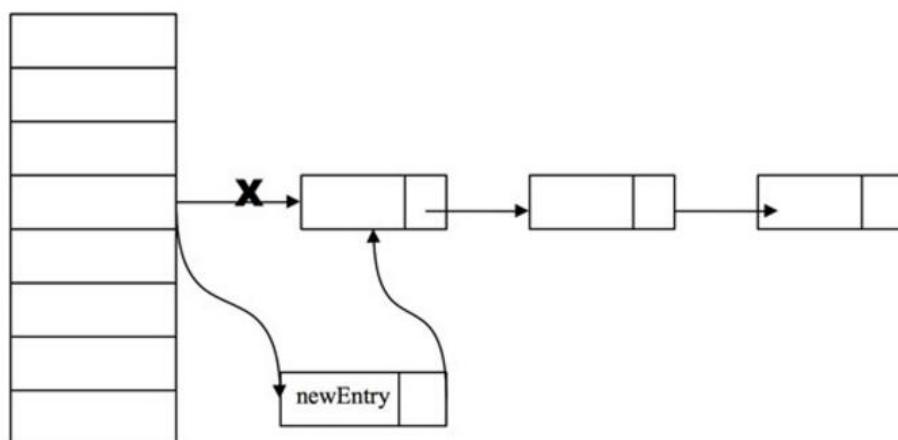
```
7 | if (v != null) // ② 注意这里
8 |     return v;
9 | return readValueUnderLock(e); // recheck
```

## 2) 在 get 代码的①和②之间，另一个线程修改了一个 entry 的 value

value 是用 volatile 修饰的，可以保证读取时获取到的是修改后的值。

## 3) 在 get 代码的①之后，另一个线程删除了一个 entry

假设我们的链表元素是: e1->e2->e3->e4 我们要删除 e3 这个 entry, 因为 HashEntry 中 next 的不可变，所以我们无法直接把 e2 的 next 指向 e4, 而是将要删除的节点之前的节点复制一份，形成新的链表。它的实现大致如下图所示：



如果我们 get 的也恰巧是 e3，可能我们顺着链表刚找到 e1，这时另一个线程就执行了删除 e3 的操作，而我们线程还会继续沿着旧的链表找到 e3 返回。这里没有办法实时保证了，也就是说没办法看到最新的。

我们第①处就判断了 count 变量，它保障了在 ①处能看到其他线程修改后的。①之后到②之间，如果再次发生了其他线程再删除了 entry 节点，就没法保证看到最新的了，这时候的 get 的实际上是未更新过的！！！。

不过这也没什么关系，即使我们返回 e3 的时候，它被其他线程删除了，暴露出去的 e3 也不会对我们新的链表造成影响。

## 2. JDK1.7 PUT

1. 将当前 Segment 中的 table 通过 key 的 hashCode 定位到 HashEntry。
2. 遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value。

3. 不为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容。

4. 最后会解除在 1 中所获取当前 Segment 的锁。

可以说是首先找到 segment，确定是哪一个 segment，然后在这个 segment 中遍历查找 key 值是要查找的 key 值得 entry，如果找到，那么就修改该 key，如果没找到，那么就在头部新加一个 entry.

```
1 public V put(K key, V value) {
2     if (value == null) //ConcurrentHashMap 中不允许用 null 作为映射值
3         throw new NullPointerException();
4     int hash = hash(key.hashCode()); //计算键对应的散列码
5
6     //根据散列码找到对应的 Segment
7     return segmentFor(hash).put(key, hash, value, false);
8 }
9
10 V put(K key, int hash, V value, boolean onlyIfAbsent) {
11     lock(); //当前的segment加锁
12     try {
13         int c = count;
14         if (c++ > threshold) //如果超过再散列的阈值
15             rehash(); //执行再散列，table 数组的长度将扩充一倍
16         HashEntry<K,V>[] tab = table;
```

```
17     //把散列码值与 table 数组的长度减 1 的值相“与”
18     //得到该散列码对应的 table 数组的下标值
19     int index = hash & (tab.length - 1);
20
21     //找到散列码对应的具体的那个桶
22     HashEntry<K,V> first = tab[index];
23     HashEntry<K,V> e = first;
24     while (e != null && (e.hash != hash || !key.equals(e.key)))
25         e = e.next;
26
27     V oldValue;
28     if (e != null) { //如果键/值对已经存在
29         oldValue = e.value;
30     }
```

```

31         if (!onlyIfAbsent)
32             e.value = value; // 设置 value 值
33     }
34     else { //键/值对不存在
35         oldValue = null;
36         ++modCount; //添加新节点到链表中, modCount 要加 1
37
38         // 创建新节点, 并添加到链表的头部
39         tab[index] = new HashEntry<K,V>(key, hash, first, value);
40         count = c; //写 count 变量
41     }
42     return oldValue;
43 } finally {
44     unlock(); //解锁
45 }
46 }
```

### 3. JDK1.7 Remove

```

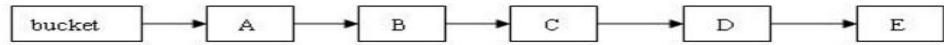
1 V remove(Object key, int hash, Object value) {
2     lock(); //加锁
3     try{
4         int c = count - 1;
5         HashEntry<K,V>[] tab = table;
6         //根据散列码找到 table 的下标值
7         int index = hash & (tab.length - 1);
8         //找到散列码对应的那个桶
9         HashEntry<K,V> first = tab[index];
10        HashEntry<K,V> e = first;
11        while(e != null&& (e.hash != hash || !key.equals(e.key)))
12            e = e.next;
13
14        V oldValue = null;
15        if(e != null) {
16            V v = e.value;
17            if(value == null|| value.equals(v)) { //找到要删除的节点
```

```

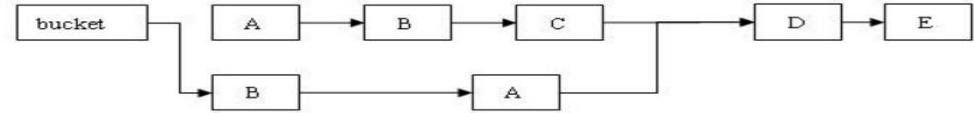
18                     oldValue = v;
19                     ++modCount;
20                     //所有处于待删除节点之后的节点原样保留在链表中
21                     //所有处于待删除节点之前的节点被克隆到新链表中
22                     HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点
23                     for(HashEntry<K,V> p = first; p != e; p = p.next)
24                         newFirst = new HashEntry<K,V>(p.key, p.hash,
25   newFirst, p.value);
26                     //把桶链接到新的头结点
27                     //新的头结点是原链表中, 删除节点之前的那个节点
28                     tab[index] = newFirst;
29                     count = c;           //写 count 变量
30                 }
31             }
32             return oldValue;
33         } finally{

```

执行删除之前的原链表：



执行删除之后的新链表



#### 4. JDK1.7 & JDK1.8 size()

```
private transient volatile CounterCell[] counterCells;
```

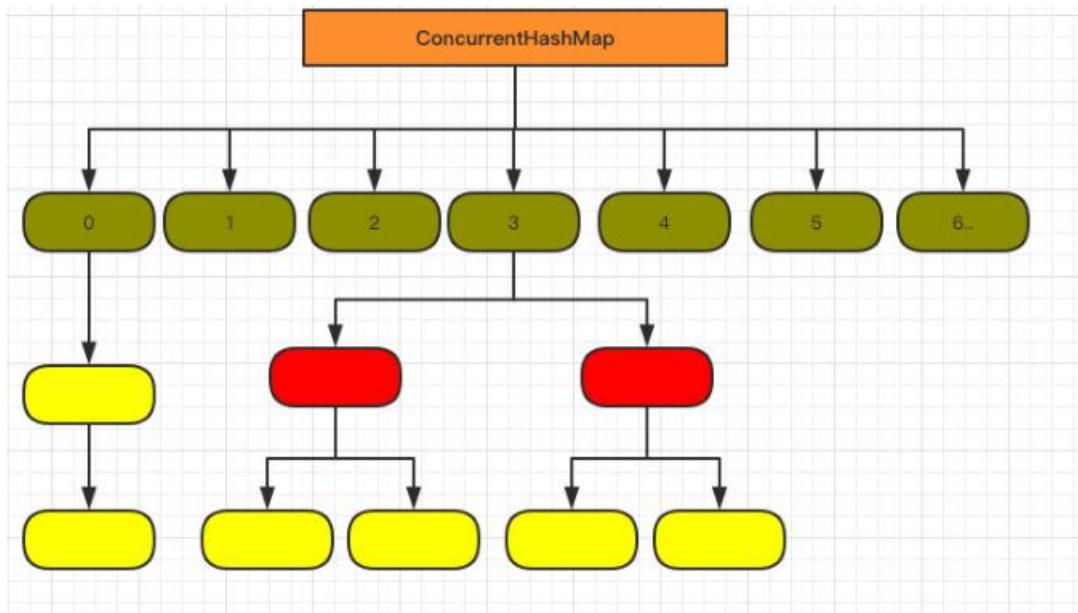
```

public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
        (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
        (int)n);
}

```

volatile 保证内存可见，最大是 65535.

#### 5. JDK 1.8



1. 其中抛弃了原有的 **Segment** 分段锁，而采用了 CAS + synchronized 来保证并发安全性。
2. 大于 8 的时候才去红黑树链表转红黑树的阀值，当 **table[i]**下面的链表长度大于 8 时就转化为红黑树结构。

## 6.JDK1.8 put

- 根据 key 计算出 hashCode 。
- 判断是否需要进行初始化。
- f 即为当前 key 定位出的 Node，如果为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
- 如果当前位置的 hashCode == MOVED == -1，则需要进行扩容。
- 如果都不满足，则利用 synchronized 锁写入数据(分为链表写入和红黑树写入)。
- 如果数量大于 TREEIFY\_THRESHOLD 则要转换为红黑树。

```

final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) { ①
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0) ②
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {③
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break; // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED) ④
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) { ⑤
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        for (Node<K,V> e = f;; ++binCount) {
                            K ek;
                            if (e.hash == hash &&
                                ((ek = e.key) == key ||
                                 (ek != null && key.equals(ek)))) {
                                oldVal = e.val;
                                if (!onlyIfAbsent)
                                    e.val = value;
                                break;
                            }
                            Node<K,V> pred = e;
                            if ((e = e.next) == null) {
                                pred.next = new Node<K,V>(hash, key,
   value, null);
                                break;
                            }
                        }
                    }
                }
            }
            else if (f instanceof TreeBin) {
                Node<K,V> p;
                binCount = 2;
                if ((p = ((TreeBin<K,V>)f).putTreeVal(hash, key,
  value)) != null) {
                    oldVal = p.val;
                    if (!onlyIfAbsent)
                        p.val = value;
                }
            }
        }
        if (binCount != 0) {
            if (binCount >= TREEIFY_THRESHOLD) ⑥
                treeifyBin(tab, i);
            if (oldVal != null)
                return oldVal;
            break;
        }
    }
}

```

## 7. JDK1.8 get 方法

```
934 public V get(Object key) {  
935     Node<K,V>[] tab; Node<K,V> e, p; int n, eh; K ek;  
936     int h = spread(key.hashCode());  
937     if ((tab = table) != null && (n = tab.length) > 0 &&  
938         (e = tabAt(tab, (n - 1) & h)) != null) {  
939         if ((eh = e.hash) == h) {  
940             if ((ek = e.key) == key || (ek != null && key.equals(ek)))  
941                 return e.val;  
942         }  
943         else if (eh < 0)  
944             return (p = e.find(h, key)) != null ? p.val : null;  
945         while ((e = e.next) != null) {  
946             if (e.hash == h &&  
947                 (ek = e.key) == key || (ek != null && key.equals(ek)))  
948                 return e.val;  
949         }  
950     }  
951     return null;  
952 }
```

- 根据计算出来的 hashCode 寻址，如果就在桶上那么直接返回值。
- 如果是红黑树那就按照树的方式获取值。
- 就不满足那就按照链表的方式遍历获取值。

## 8. rehash 过程

Redis rehash : dictRehash 每次增量 rehash n 个元素，由于在自动调整大小时已设置了 ht[1] 的大小，因此 rehash 的主要过程就是遍历 ht[0]，取得 key，然后将该 key 按 ht[1] 的桶的大小重新 rehash，并在 rehash 完后将 ht[0] 指向 ht[1]，然后将 ht[0] 清空。在这个过程中 rehashidx 非常重要，它表示上次 rehash 时在 ht[0] 的下标位置。

可以看到，redis 对 dict 的 rehash 是分批进行的，这样不会阻塞请求，设计的比较优雅。

但是在调用 dictFind 的时候，可能需要对两张 dict 表做查询。唯一的优化判断是，当 key 在 ht[0] 不存在且不在 rehashing 状态时，可以快速返回空。如果在 rehashing 状态，当在 ht[0] 没值的时候，还需要在 ht[1] 里查找。

dictAdd 的时候，如果状态是 rehashing，则把值插入到 ht[1]，否则 ht[0]

## 2.3 . Hashtable

[https://blog.csdn.net/ns\\_code/article/details/36191279](https://blog.csdn.net/ns_code/article/details/36191279)

### 0. 参数

(1) table 是一个 Entry[] 数组类型，而 Entry 实际上就是一个单向链表。哈希表的 "key-value" 键值对都是存储在 Entry 数组中的。

(2) count 是 Hashtable 的大小，它是 Hashtable 保存的键值对的数量。

(3) threshold 是 Hashtable 的阈值，用于判断是否需要调整 Hashtable 的容量。threshold 的值 = "容量 \* 加载因子"。

(4) `loadFactor` 就是加载因子。

(5) `modCount` 是用来实现 fail-fast 机制的

## 1.put

从下面的代码中我们可以看出，`Hashtable` 中的 `key` 和 `value` 是不允许为空的，当我们想要在 `Hashtable` 中添加元素的时候，首先计算 `key` 的 `hash` 值，然

后通过 `hash` 值确定在 `table` 数组中的索引位置，最后将 `value` 值替换或者插入新的元素，如果容器的数量达到阈值，就会进行扩充。

```
1. public synchronized V put(K key, V value) {
2.     // 确保value不为null
3.     if (value == null) {
4.         throw new NullPointerException();
5.     }
6.
7.     /*
8.      * 确保key在table[]是不重复的
9.      * 处理过程：
10.      * 1、计算key的hash值，确认在table[]中的索引位置
11.      * 2、迭代index索引位置，如果该位置处的链表中存在一个一样的key，则替换其value，返回旧值
12.      */
13.     Entry tab[] = table;
14.     int hash = hash(key);      //计算key的hash值
15.     int index = (hash & 0x7FFFFFFF) % tab.length;    //确认该key的索引位置
16.     //迭代，寻找该key，替换
17.     for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
18.         if ((e.hash == hash) && e.key.equals(key)) {
19.             V old = e.value;
20.             e.value = value;
21.             return old;
22.         }
23.     }
24.
25.     modCount++;
26.     if (count >= threshold) { //如果容器中的元素数量已经达到阈值，则进行扩容操作
27.         rehash();
28.         tab = table;
29.         hash = hash(key);
30.         index = (hash & 0x7FFFFFFF) % tab.length;
31.     }
32.
33.     // 在索引位置处插入一个新的节点
34.     Entry<K,V> e = tab[index];
35.     tab[index] = new Entry<>(hash, key, value, e);
36.     //容器中元素+1
37.     count++;
38.     return null;
39. }
```

## 2.get

同样也是先获得索引值，然后进行遍历，最后返回

```
[java] ━ ━
1. public synchronized V get(Object key) {
2.     Entry tab[] = table;
3.     int hash = hash(key);
4.     int index = (hash & 0x7FFFFFFF) % tab.length;
5.     for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
6.         if ((e.hash == hash) && e.key.equals(key)) {
7.             return e.value;
8.         }
9.     }
10.    return null;
11. }
```

### 3.Remove

在下面代码中，如果 prev 为 null 了，那么说明第一个元素就是要删除的元素，那么就直接指向第一个元素的下一个即可。

```
// 删除Hashtable中键为key的元素
public synchronized V remove(Object key) {
    Entry tab[] = table;
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;

    //从table[index]链表中找出要删除的节点，并删除该节点。
    //因为是单链表，因此要保留带删节点的前一个节点，才能有效地删除节点
    for (Entry<K,V> e = tab[index], prev = null ; e != null ; prev = e, e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            modCount++;
            if (prev != null) {
                prev.next = e.next;
            } else {
                tab[index] = e.next;
            }
            count--;
            V oldValue = e.value;
            e.value = null;
            return oldValue;
        }
    }
    return null;
}
```

### 4.扩容

默认初始容量为 11

线程安全，但是速度慢，不允许 key/value 为 null

加载因子为 0.75：即当 元素个数 超过 容量长度的 0.75 倍 时，进行扩容

扩容增量：2\*原数组长度+1

如 **HashTable** 的容量为 11，一次扩容后是容量为 23

## 2.4 hashtable 和 hashmap 的区别

- Hash Map和Hash Table的区别
  - Hashtable的方法是同步的，HashMap未经同步，所以在多线程场合要手动同步HashMap这个区别就像Vector和ArrayList一样。
  - Hashtable不允许 null 值(key 和 value 都不可以)，HashMap允许 null 值(key和value都可以)。
  - 两者的遍历方式大同小异，Hashtable仅仅比HashMap多一个elements方法。  
    Hashtable 和 HashMap 都能通过values()方法返回一个 Collection，然后进行遍历处理。  
    两者也都可以通过entrySet() 方法返回一个 Set，然后进行遍历处理。
  - HashTable使用Enumeration，HashMap使用Iterator。
  - 哈希值的使用不同，Hashtable直接使用对象的hashCode。而HashMap重新计算hash值，而且用于代替求模。
  - Hashtable中hash数组默认大小是11，增加的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。
  - HashTable基于Dictionary类，而HashMap基于AbstractMap类

## 2.5 HashMap 和 ConCurrentHashMap 区别

- HashMap和ConcurrentHashMap区别？
  - HashMap是非线程安全的，ConcurrentHashMap是线程安全的。
  - ConcurrentHashMap将整个Hash桶进行了分段segment，也就是将这个大的数组分成了几个小的片段segment，而且每个小的片段segment上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段segment，然后再在这个片段上面进行插入，而且这里还需要获取segment锁。
  - ConcurrentHashMap让锁的粒度更精细一些，并发性能更好。

## 2.6 ConcurrentHashMap 和 HashTable 区别

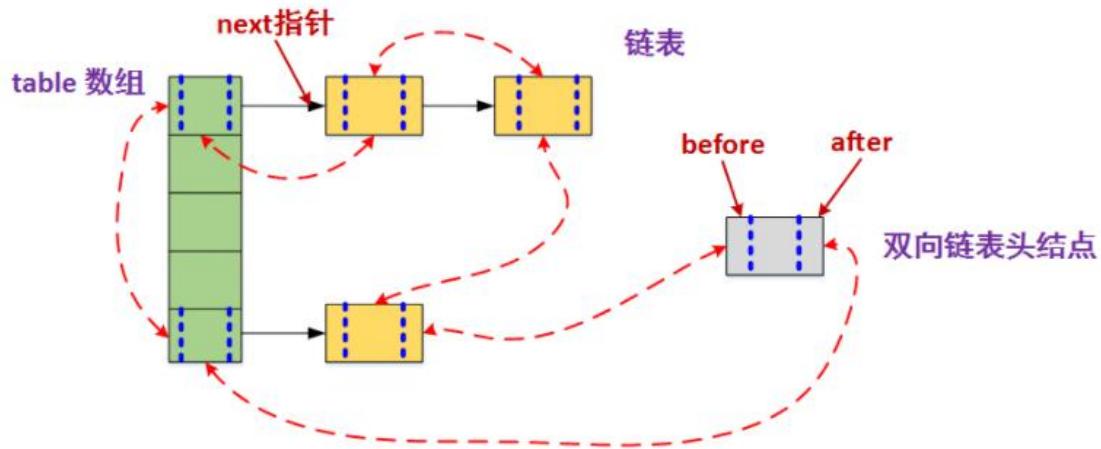
ConcurrentHashMap 仅仅锁定 map 的某个部分，而 Hashtable 则会锁定整个 map。

hashtable(同一把锁):使用 synchronized 来保证线程安全，但效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

concurrenthashmap(分段锁):(锁分段技术)每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。concurrenthashmap 是由 Segment 数组结构和 HashEntry 数组结构组成。Segment 是一种可重入锁 ReentrantLock，扮演锁的角色。HashEntry 用于存储键值对数据。一个 concurrenthashmap 里包含一个 Segment 数组。Segment 的结构和 Hashmap 类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个 HashEntry 数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment。

## 2.7 linkedHashMap

<https://blog.csdn.net/justloveyou/article/details/71713781>



本示意图中，LinkedHashMap一共包含五个节点。除去红色双向虚线来看，其就是一个正宗的HashMap。在这里，用额外的红色虚线串起来的节点就是一个双向链表了。由此可见，LinkedHashMap就是一个标准的HashMap与LinkedList的融合体。

<http://blog.csdn.net/justloveyou>

## 2.8 LinkedHashMap 与 hashmap 的区别

1. LinkedHashMap 是 HashMap 的子类
2. LinkedHashMap 中的 Entry 增加了两个指针 **before** 和 **after**，它们分别用于维护双向链接列表。
3. 在 put 操作上，虽然 LinkedHashMap 完全继承了 HashMap 的 put 操作，但是在细节上还是做了一定的调整，比如，在 LinkedHashMap 中向哈希表中插入新 Entry 的同时，还会通过 Entry 的 addBefore 方法将其链入到双向链表中。
4. 在扩容操作上，虽然 LinkedHashMap 完全继承了 HashMap 的 resize 操作，但是鉴于性能和 LinkedHashMap 自身特点的考量，LinkedHashMap 对其中的重哈希过程(transfer 方法)进行了重写
5. 在读取操作上，LinkedHashMap 中重写了 HashMap 中的 get 方法，通过 HashMap 中的 getEntry 方法获取 Entry 对象。在此基础上，进一步获取指定键对应的值。

## 2.9 HashSet

对于 HashSet 而言，它是基于 HashMap 实现的

Hashset 源码 <http://zhangshixi.iteye.com/blog/673143>

Hashset 如何保证集合的没有重复元素？

可以看出 HashSet 底层是 hashmap 但是存储的是一个对象，HashSet 实际将该元

素 e 作为 key 放入 hashmap, 当 key 值(该元素 e)相同时, 只是进行更新 value, 并不会新增加, 所以 set 中的元素不会进行改变。

```
public HashSet() {  
    map = new HashMap<E, Object>();  
}  
  
/**  
 * 如果此set中尚未包含指定元素，则添加指定元素。  
 * 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))  
 * 的元素e2，则向此set 添加指定的元素e。  
 * 如果此set已包含该元素，则该调用不更改set并返回false。  
 *  
 * 底层实际将将该元素作为key放入HashMap。  
 * 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key  
 * 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），  
 * 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变，  
 * 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，  
 * 原来的元素也不会有任何改变，这也满足了Set中元素不重复的特性。  
 * @param e 将添加到此set中的元素。  
 * @return 如果此set尚未包含指定元素，则返回true。  
 */  
public boolean add(E e) {  
    return map.put(e, PRESENT)==null;  
}
```

## 2.10 hashmap 与 hashset 区别

| HashMap                            | HashSet                                                                                                |
|------------------------------------|--------------------------------------------------------------------------------------------------------|
| 实现了Map接口                           | 实现Set接口                                                                                                |
| 存储键值对                              | 仅存储对象                                                                                                  |
| 调用put( ) 向map中添加元素                 | 调用add( ) 方法向Set中添加元素                                                                                   |
| HashMap使用键 ( Key ) 计算HashCode      | HashSet使用成员对象来计算hashcode值，<br>对于两个对象来说hashcode可能相同，<br>所以equals()方法用来判断对象的相等性，<br>如果两个对象不同的话，那么返回false |
| HashMap相对于HashSet较快，因为它是使用唯一的键获取对象 | HashSet较HashMap来说比较慢                                                                                   |

## 2.11 Collections.sort 内部原理

1.1 Collections.sort 排序通过泛化实现对所有类型的排序，对于基础类型如 int, string, 按照字符表，数字大小排序，对于自定义类型，通过实现 Comparable 接口，重写 compareTo 函数自定义比较大小的方式。接收对象类型 extends Comparable<? super T> 或者 Comparator 外比较器，Comparable 接口的方式比实现 Comparator 接口的耦合性 要强一些

1.2 Collections.sort 内部调用的是 Arrays.sort 方法，对于 Arrays 类，有两个 sort 方法，sort (Object) 和 sort (int)。前者使用的是归并排序，后者是快排。

源码中的优化：

1. 短数组使用插入排序快
2. 混乱度判断（通过找出所有的递增递减子数组，判断）

Collections.sort 函数 jdk7 和 jdk8 分别怎么实现的

重写 Collections.sort()

```
import java.util.*;
class xd{
    int a;
    int b;
    xd(int a, int b) {
        this.a = a;
        this.b = b;
    }
}
public class Main {
    public static void main(String[] arg) {
        xd a = new xd(2, 3);
        xd b = new xd(4, 1);
        xd c = new xd(1, 2);
        ArrayList<xd> array = new ArrayList<>();
        array.add(a);
        array.add(b);
        array.add(c);
        Collections.sort(array, new Comparator<xd>() {
            @Override
            public int compare(xd o1, xd o2) {
                if(o1.a > o2.a)
                    return 1;
                else if(o1.a < o2.a)
                    return -1;
                return 0;
            }
        });
    }
}
```

```
        for(int i=0;i<array.size();i++)
            System.out.println(array.get(i).a);
        for(int i=0;i<array.size();i++)
            System.out.println(array.get(i).b);
    }

}
```

## 2.12 hash 算法

- 加法Hash；把输入元素一个一个的加起来构成最后的结果
- 位运算Hash；这类型Hash函数通过利用各种位运算（常见的是移位和异或）来充分的混合输入元素
- 乘法Hash；这种类型的Hash函数利用了乘法的不相关性（乘法的这种性质，最有名的莫过于平方取头尾的随机数生成算法，虽然这种算法效果并不好）；jdk5.0里面的String类的hashCode()方法也使用乘法Hash；32位FNV算法
- 除法Hash；除法和乘法一样，同样具有表面上看起来的不相关性。不过，因为除法太慢，这种方式几乎找不到真正的应用
- 查表Hash；查表Hash最有名的例子莫过于CRC系列算法。虽然CRC系列算法本身并不是查表，但是，查表是它的一种最快的实现方式。查表Hash中有名的例子有：Universal Hashing和Zobrist Hashing。他们的表格都是随机生成的。
- 混合Hash；混合Hash算法利用了以上各种方式。各种常见的Hash算法，比如MD5、Tiger都属于这个范围。它们一般很少在面向查找的Hash函数里面使用

java map 底层实现，最好看源码，还有各种集合类的区别

4. TreeMap 和 TreeSet 区别和实现原理

5. 集合和有序集合有什么区别

6. Set 是无序的，那怎么保证它有序？有什么办法吗？提到了 TreeSet，那说一下 TreeSet 为什么能够保证有序？

7. java 中 hashMap 结构，处理冲突方法，还有啥方法，各个方法优缺点

.Collections.sort() 的原理

集合框架的理解 对 Java 的集合框架有什么样的了解，用过哪些集合类，各自的效率以及适用场景

cas 的实现原理，以及 aba 问题

List/Set/Queue 接口及其实现类

HashSet/TreeSet/HashMap/TreeMap/hashTable 这些类的底层实现。

常问： HashSet 和 HashMap 有什么区别。各自的底层实现是基于什么的。  
这里紧接着的问题通常是：我们来聊聊多线程（微笑脸），或者我们来聊聊红黑树。

## 2.13 迭代器 Iterator Enumeration

### 1. Iterator 和 ListIterator 的区别是什么？

答：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。

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

答：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。

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

答：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。

## 2.14 LIST ArrayList, LinkedList 和 Vector 的区别和实现原理

Vector : <https://blog.csdn.net/chenssy/article/details/37520981>

### 一. ArrayList 与 LinkedList 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。

### 二. Vetor arraylist Linkedlist 区别

ArrayList 就是动态数组，是 Array 的复杂版本，动态的增加和减少元素。当更多的元素加入到 ArrayList 中时，其大小将会动态地增长。它的元素可以通过 get/set 方法直接访问，

因为 ArrayList 本质上是一个数组。初始容量为 10。1.插入元素的时候可能扩容，删除元素时

不会缩小容量。2.扩容增长为 ArrayList 增长原来的 0.5 倍 3. 而 ArrayList 没有设置增长空间的方法。4.线程不同步

Vector 和 ArrayList 类似，区别在于 Vector 是同步类(synchronized)。因此，开销就比 ArrayList 要大。初始容量为 10。实现了随机访问接口，可以随机访问。Vector 是内部是以

动态数组的形式来存储数据的。1.Vector 还可以设置增长的空间大小，2. 及 Vector 增长原来的 1 倍 3.vector 线程同步

LinkedList 是一个双链表，在添加和删除元素时具有比 ArrayList 更好的性能。但在 get 与 set 方面弱于 ArrayList。当然，这些对比都是指数据量很大或者操作很频繁的情况下的对比。它还实现了 Queue 接口，该接口比 List 提供了更多的方法，包括 offer(), peek(), poll() 等。

ArrayList 和 LinkedList 的使用场景，其中 add 方法的实现 ArrayList, LinkedList 的实现以及插入，查找，删除的过程

ArrayList 如何实现排序

### 三. 使用 ArrayList 的迭代器会出现什么问题？单线程和多线程环境下；

答：常用的迭代器设计模式， iterator 方法返回一个父类实现的迭代器。

1、迭代器的 hasNext 方法的作用是判断当前位置是否是数组最后一个位置，相等为 false，否则为 true。

2、迭代器 next 方法用于返回当前的元素，并把指针指向下一个元素，值得注意的是，每次使用 next 方法的时候，都会判断创建迭代器获取的这个容器的计数器 modCount 是否与此时的不相等，不相等说明集合的大小被修改过，如果是会抛出 ConcurrentModificationException 异常，如果相等调用 get 方法返回元素即可。

### 四. 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？

答：不同点：定义上：Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。容量上：Array 大小固定，ArrayList 的大小是动态变化的。操作上：ArrayList 提供更多的方法和特性，如：addAll(), removeAll(), iterator() 等等。使用基本数据类型或者知道数据元素数量的时候可以考虑 Array; ArrayList 处理固定数量的基本类型数据类型时会自动装箱来减少编码工作量，但是相对较慢。

### 五. ArrayList 和 Vector 有何异同点？

相同点：

- (1) 两者都是基于索引的，都是基于数组的。
- (2) 两者都维护插入顺序，我们可以根据插入顺序来获取元素。
- (3) ArrayList 和 Vector 的迭代器实现都是 fail-fast 的。
- (4) ArrayList 和 Vector 两者允许 null 值，也可以使用索引值对元素进行随机访问。

不同点：

- (1) Vector 是同步，线程安全，而 ArrayList 非同步，线程不安全。对于 ArrayList，如果迭代时改变列表，应该使用 CopyOnWriteArrayList。

(2) 但是, `ArrayList` 比 `Vector` 要快, 它因为有同步, 不会过载。

(3) 在使用上, `ArrayList` 更加通用, 因为 `Collections` 工具类容易获取同步列表和只读列表。

## 2.15 快速失败(fail-fast)和安全失败(fail-safe)

### 一: 快速失败 (fail—fast)

在用迭代器遍历一个集合对象时, 如果遍历过程中对集合对象的内容进行了修改(增加、删除、修改), 则会抛出 `Concurrent Modification Exception`。

原理: 迭代器在遍历时直接访问集合中的内容, 并且在遍历过程中使用一个 `modCount` 变量。集合在被遍历期间如果内容发生变化, 就会改变 `modCount` 的值。每当迭代器使用 `hashNext()/next()` 遍历下一个元素之前, 都会检测 `modCount` 变量是否为 `expectedmodCount` 值, 是的话就返回遍历; 否则抛出异常, 终止遍历。

注意: 这里异常的抛出条件是检测到 `modCount! =expectedmodCount` 这个条件。如果集合发生变化时修改 `modCount` 值刚好又设置为了 `expectedmodCount` 值, 则异常不会抛出。因此, 不能依赖于这个异常是否抛出而进行并发操作的编程, 这个异常只建议用于检测并发修改的 bug。

场景: `java.util` 包下的集合类都是快速失败的, 不能在多线程下发生并发修改(迭代过程中被修改)。

### 二: 安全失败 (fail—safe)

采用安全失败机制的集合容器, 在遍历时不是直接在集合内容上访问的, 而是先复制原有集合内容, 在拷贝的集合上进行遍历。

原理: 由于迭代时是对原集合的拷贝进行遍历, 所以在遍历过程中对原集合所作的修改并不能被迭代器检测到, 所以不会触发 `Concurrent Modification Exception`。

缺点: 基于拷贝内容的优点是避免了 `Concurrent Modification Exception`, 但同样地, 迭代器并不能访问到修改后的内容, 即: 迭代器遍历的是开始遍历那一刻拿到的集合拷贝, 在遍历期间原集合发生的修改迭代器是不知道的。

场景: `java.util.concurrent` 包下的容器都是安全失败, 可以在多线程下并发使用, 并发修改。

快速失败和安全失败是对迭代器而言的。 快速失败: 当在迭代一个集合的时候, 如果有另外一个线程在修改这个集合, 就会抛出 `ConcurrentModification` 异常, `java.util` 下都是快速失败。 安全失败: 在迭代时候会在集合二层做一个拷贝, 所以在修改集合上层元素不会影响下层。在 `java.util.concurrent` 下都是安全失败

# 三 锁 volatile synchronized Lock

## ReentrantLock AQS CAS

### 3.1 volatile 和 synchronized

#### 1.volatile 和 synchronized 实现原理

如果主内存 count 变量发生修改之后，线程工作内存中的值由于已经加载，不会产生对应的变化，所以计算出来的结果会和预期不一样

对于 volatile 修饰的变量，jvm 虚拟机只是保证从主内存加载到线程工作内存的值是最新的  
Volatile 是如何来保证可见性？

lock 前缀的指令在多核处理器下会引发了两件事情：

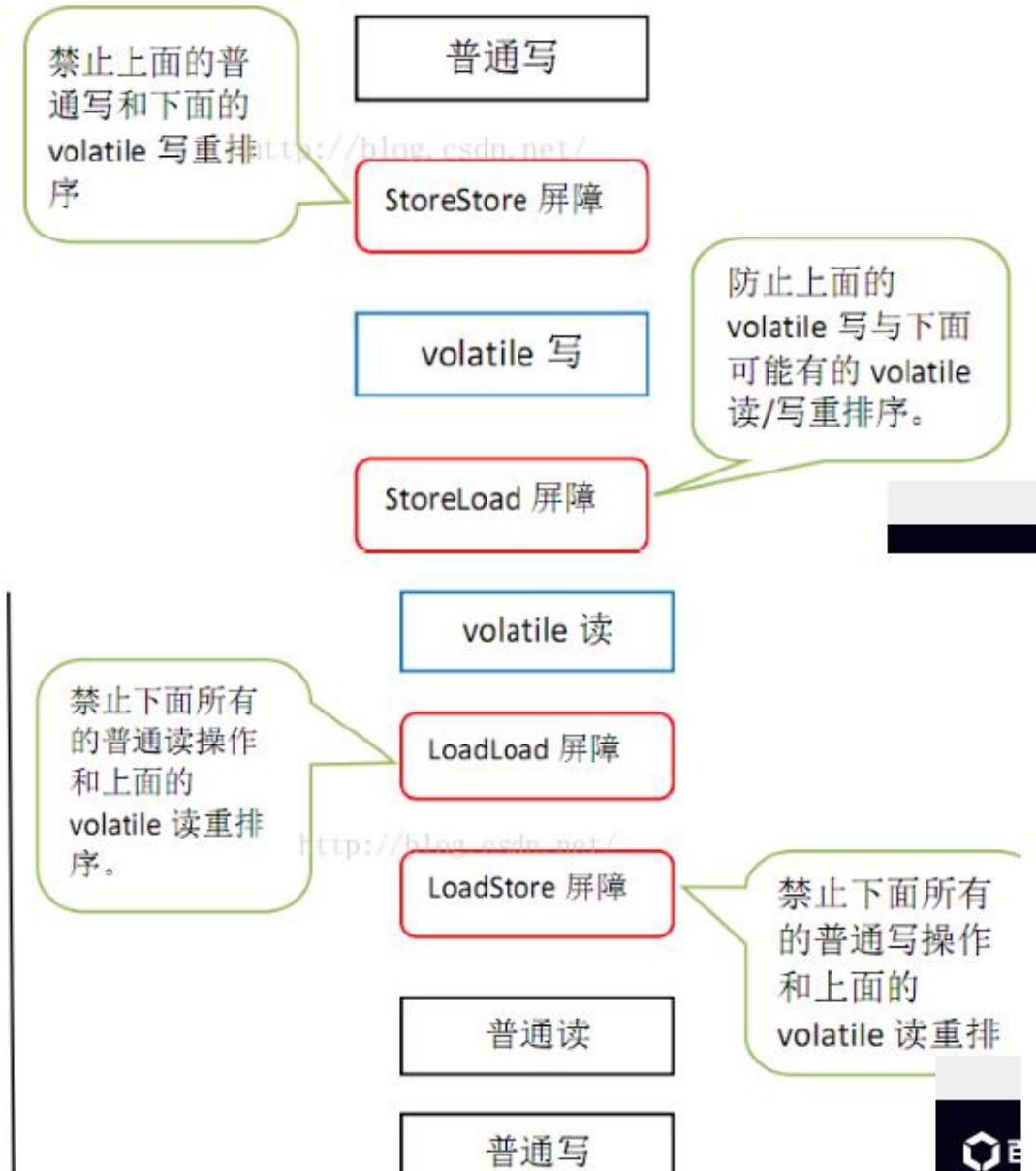
1. 将当前处理器缓存行的数据会写回到系统内存
2. 这个写回内存的操作会引起在其他 CPU 里缓存了该内存地址的数据无效。

如果对声明了 Volatile 变量进行写操作，JVM 就会向处理器发送一条 Lock 前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

Volatile 的内存屏障：——防止指令重排

为了实现 volatile 的语义，JMM 在编译器和处理器层面限制指令重排序。

JMM 的内存屏障插入策略是保守策略：



对代码块同步：

Synchronized 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 monitorenter 指令时尝试获取 monitor 的所有权，过程如下：

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

同步方法：

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。Synchronize 和 lock 都属于同步阻塞。

使用 CAS (乐观锁,) 同步非阻塞。

### 3.1.1 Volatile 与 synchronized 区别

volatile和synchronized区别

- volatile不会进行加锁操作：  
volatile变量是一种稍弱的同步机制在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。
- volatile 变量作用类似于同步变量读写操作：  
从内存可见性的角度看，写入volatile变量相当于退出同步代码块，而读取volatile变量相当于进入同步代码块。
- volatile 不如 synchronized安全：  
在代码中如果过度依赖volatile变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当volatile变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。
- volatile 无法同时保证内存可见性和原则性：  
加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性，原因是声明为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用，也就是说如下的表达式都不是原子操作：“count++”、“count = count+1”。

### 3.1.2 Volatile

volatile 关键字如何保证内存可见性

- volatile 关键字的作用
  - 保证内存的可见性
  - 防止指令重排
  - 注意：volatile 并不保证原子性
- 内存可见性
  - volatile保证可见性的原理是在每次访问变量时都会进行一次刷新，因此每次访问都是主内存中最新的版本。所以volatile关键字的作用之一就是保证变量修改的实时可见性。
- 当且仅当满足以下所有条件时，才应该使用volatile变量
  - 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
  - 该变量没有包含在具有其他变量的不变式中。
- volatile 使用建议
  - 在两个或者更多的线程需要访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，没必要使用volatile。
  - 由于使用volatile屏蔽掉了JVM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

### 3.1.3 Synchronized 原理

<https://blog.csdn.net/javazejian/article/details/72828483>

<https://blog.csdn.net/chen77716/article/details/6618779>

Synchronized 是可重入的

Jvm 对象都有对象头，对象头是实现 synchronized 的基础，对象头都有如下结构

| 虚拟机位数    | 头对象结构                  | 说明                                   |
|----------|------------------------|--------------------------------------|
| 32/64bit | Mark Word              | 存储对象的hashCode、锁信息或分代年龄或GC标志等信息       |
| 32/64bit | Class Metadata Address | 类型指针指向对象的美元数据，JVM通过这个指针确定该对象是哪个类的实例。 |

而 Mark Word 的结构（不固定）如下所示

| 锁状态  | 25bit      | 4bit   | 1bit是否是偏向锁 | 2bit 锁标志位 |
|------|------------|--------|------------|-----------|
| 无锁状态 | 对象HashCode | 对象分代年龄 | 0          | 01        |

| 锁状态  | 25 bit         |       | 4bit   | 1bit | 2bit |  |  |
|------|----------------|-------|--------|------|------|--|--|
|      | 23bit          | 2bit  |        |      |      |  |  |
| 轻量级锁 | 指向栈中锁记录的指针     |       |        |      | 00   |  |  |
| 重量级锁 | 指向互斥量（重量级锁）的指针 |       |        |      | 10   |  |  |
| GC标记 | 空              |       |        |      | 11   |  |  |
| 偏向锁  | 线程ID           | Epoch | 对象分代年龄 | 1    | 01   |  |  |

从上图可以看到 Mark Word 包括重量级锁，重量级锁的指针指向 monitor 对象（监视器锁）所以每一个对象都与一个 monitor 关联，当一个 monitor 被某个线程持有后，它便处于锁定状态。谈及 monitor 来看一下 monitord 的实现，monitor 是由 ObjectMonitor 实现的，其主要数据结构如下

```

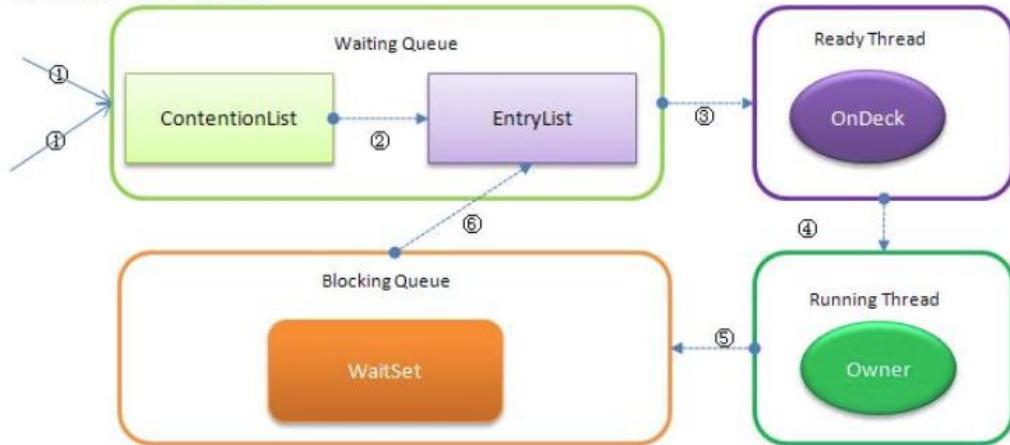
1  ObjectMonitor() {
2      _header      = NULL;
3      _count      = 0; //记录个数
4      _waiters     = 0,
5      _recursions   = 0;
6      _object      = NULL;
7      _owner       = NULL;
8      _WaitSet    = NULL; //处于wait状态的线程，会被加入到_WaitSet
9      _WaitSetLock = 0 ;
10     _Responsible = NULL ;
11     _succ        = NULL ;
12     _cxq         = NULL ;
13     _FreeNext    = NULL ;
14     _EntryList   = NULL ; //处于等待锁block状态的线程，会被加入到该列表
15     _SpinFreq    = 0 ;
16     _SpinClock   = 0 ;
17     OwnerIsThread = 0 ;
18 }
```

当多个线程同时请求某个对象监视器时，对象监视器会设置几种状态用来区分请求的线程：

- **Contention List:** 所有请求锁的线程将被首先放置到该竞争队列
- **Entry List:** Contention List 中那些有资格成为候选人的线程被移到 Entry List
- **Wait Set:** 那些调用 wait 方法被阻塞的线程被放置到 Wait Set

- **OnDeck:** 任何时刻最多只能有一个线程正在竞争锁，该线程称为 OnDeck
- **Owner:** 获得锁的线程称为 Owner
- **!Owner:** 释放锁的线程

下图反映了个状态转换关系：



#### 对代码块同步：

**Synchronized** 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 monitorenter 指令时尝试获取 monitor 的所有权，过程如下：

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

#### 同步方法：

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  
**Synchronize** 和 **lock** 都属于同步阻塞。

使用 **CAS**（乐观锁，）同步非阻塞。

**synchronized** 和 **lock** 差别 **Lock** 多出的三大优势（尝试非阻塞加锁，尝试超时加锁，尝试可相应中断加锁）

**Lock** 内部实现

**synchronized** 可以替代读写锁吗

**ReentrantLock** 源码

**volatile** 理解，这个问题很常见，答出要点： 可见性、防止指令重排即可

**java 锁机制**

**java 中的同步机制**，锁（重入锁）机制，其他解决同步的方 **volatile** 关键字 **ThreadLocal** 类的实现原理要懂。

要减小锁粒度呢（用 Java 提供的原子类，比如 **AtomicInteger**），

**AtomicInteger** 怎么实现原子修改的（核心方法是 **compareAndSwap** 方法，俗称 CAS，源代码没有公开），CAS 方法的主要功能是什么？

## Java 编程写一个会导致死锁的程序

## 3.2 CAS

<https://www.jianshu.com/p/fb6e91b013cc>

CAS 的思想很简单：三个参数，一个当前内存值 V、旧的预期值 A、即将更新的值 B，当且仅当预期值 A 和内存值 V 相同时，将内存值修改为 B 并返回 true，否则什么都不做，并返回 false。

实现原理：CAS 中有 Unsafe 类中的 compareAndSwapInt 方法，Unsafe 类中的 compareAndSwapInt，是一个本地方法，该方法的实现位于 `unsafe.cpp` 中 `Unsafe.cpp`：

先想办法拿到变量 value 在内存中的地址。

通过 `Atomic::cmpxchg` 实现比较替换，其中参数 x 是即将更新的值，参数 e 是原内存的值。其中 `Atomic::cmpxchg` 中对此指令加 lock 前缀，而 lock 前缀有两个特性 1，禁止该指令与前面和后面的读写指令重排序 2，把写缓冲区的所有数据刷新到内存中

这两点保证了内存屏障效果，保证了 CAS 同时具有 volatile 读和 volatile 写的内存语义。

**CAS 解读示例：**

假设线程A和线程B同时执行getAndAdd操作（分别跑在不同CPU上）：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据 Java内存模型，线程A和线程B各自持有一份value的副本，值为3。
2. 线程A通过 `getIntVolatile(var1, var2)` 拿到value值3，这时线程A被挂起。
3. 线程B也通过 `getIntVolatile(var1, var2)` 方法获取到value值3，运气好，线程B没有被挂起，并执行 `compareAndSwapInt` 方法比较内存值也为3，成功修改内存值为2。
4. 这时线程A恢复，执行 `compareAndSwapInt` 方法比较，发现自己手里的值(3)和内存的值(2)不一致，说明该值已经被其它线程提前修改过了，那只能重新来一遍了。
5. 重新获取value值，因为变量value被volatile修饰，所以其它线程对它的修改，线程A总是能够看到，线程A继续执行 `compareAndSwapInt` 进行比较替换，直到成功。

## 3.3 可重入锁 ReentrantLock

<https://www.cnblogs.com/xrq730/p/4979021.html>

### 1.5.3 乐观锁和悲观锁 阻塞锁，自旋锁，偏向锁，轻量锁，重量锁。公平锁 非公平锁

自旋锁引入背景：那些处于 ContetionList、EntryList、WaitSet 中的线程均处于阻塞

状态，阻塞操作由操作系统完成（在 Linux 下通过 `pthread_mutex_lock` 函数）。线程被阻塞后便进入内核（Linux）调度状态，这个会导致系统在用户态与内核态之间来回切换，严重影响锁的性能

#### 偏向锁引入背景：避免重入锁的 CAS 操作

<https://www.tuicool.com/articles/YVrQFj>

##### 1. 公平锁与非公平锁

公平锁和非公平锁，顾名思义，公平锁就是获得锁的顺序按照先到先得的原则，从实现上说，要求当一个线程竞争某个对象锁时，只要这个锁的等待队列非空，就必须把这个线程阻塞并塞入队尾（插入队尾一般通过一个 CAS 保持插入过程中没有锁释放）。相对的，非公平锁场景下，每个线程都先要竞争锁，在竞争失败或当前已被加锁的前提下才会被塞入等待队列，在这种实现下，后到的线程有可能无需进入等待队列直接竞争到锁。

（机制）`synchronized` 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。`Lock` 用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是 CAS 操作（Compare and Swap）。

## 3.3 ReentrantLock 和 synchronized 区别

| 类别   | <code>synchronized</code>                      | <code>Lock</code>                                   |
|------|------------------------------------------------|-----------------------------------------------------|
| 存在层次 | Java 的关键字，在 jvm 层面上                            | 是一个类                                                |
| 锁的释放 | 1、以获取锁的线程执行完同步代码，释放锁<br>2、线程执行发生异常，jvm 会让线程释放锁 | 在 finally 中必须释放锁，不然容易造成线程死锁                         |
| 锁的获取 | 假设 A 线程获得锁，B 线程等待。如果 A 线程阻塞，B 线程会一直等待          | 分情况而定，Lock 有多个锁获取的方式，具体下面会说道，大致就是可以尝试获得锁，线程可以不用一直等待 |
| 锁状态  | 无法判断                                           | 可以判断                                                |
| 锁类型  | 可重入 不可中断 非公平                                   | 可重入 可判断 可公平（两者皆可）                                   |
| 性能   | 少量同步                                           | 大量同步                                                |

`synchronized` 原语和 `ReentrantLock` 在一般情况下没有什么区别，但是在非常复杂的同步应用中，请考虑使用 `ReentrantLock`，特别是遇到下面 2 种需求的时候。

1. 某个线程在等待一个锁的控制权的这段时间需要中断
  2. 需要分开处理一些 `wait-notify`, `ReentrantLock` 里面的 `Condition` 应用，能够控制 `notify` 哪个线程
  3. 具有公平锁功能，每个到来的线程都将排队等候
3. `ReentrantLock` 可中断 可超时尝试非阻塞加锁，尝试超时加锁，尝试可相应中断加锁
  4. `ReentrantLock`

`lock()`: 获取锁，如果锁被暂用则一直等待

`unlock()`: 释放锁

`tryLock()`: 注意返回类型是 `boolean`，如果获取锁的时候锁被占用就返回 `false`，否则返回 `true`

`tryLock(long time, TimeUnit unit)`: 比起 `tryLock()` 就是给了一个时间期限，保证等待参数时间

`lockInterruptibly()`: 用该锁的获得方式，如果线程在获取锁的阶段进入了等待，那么可以中断此线程，先去做别的事

### 1.5.3 重入锁、对象锁、类锁的关系

## 四 java 多线程；

<https://blog.csdn.net/evankaka/article/details/44153709#t3>

<https://www.cnblogs.com/gaopeng527/p/4234211.html>

<https://www.cnblogs.com/lixuan1998/p/6937986.html>

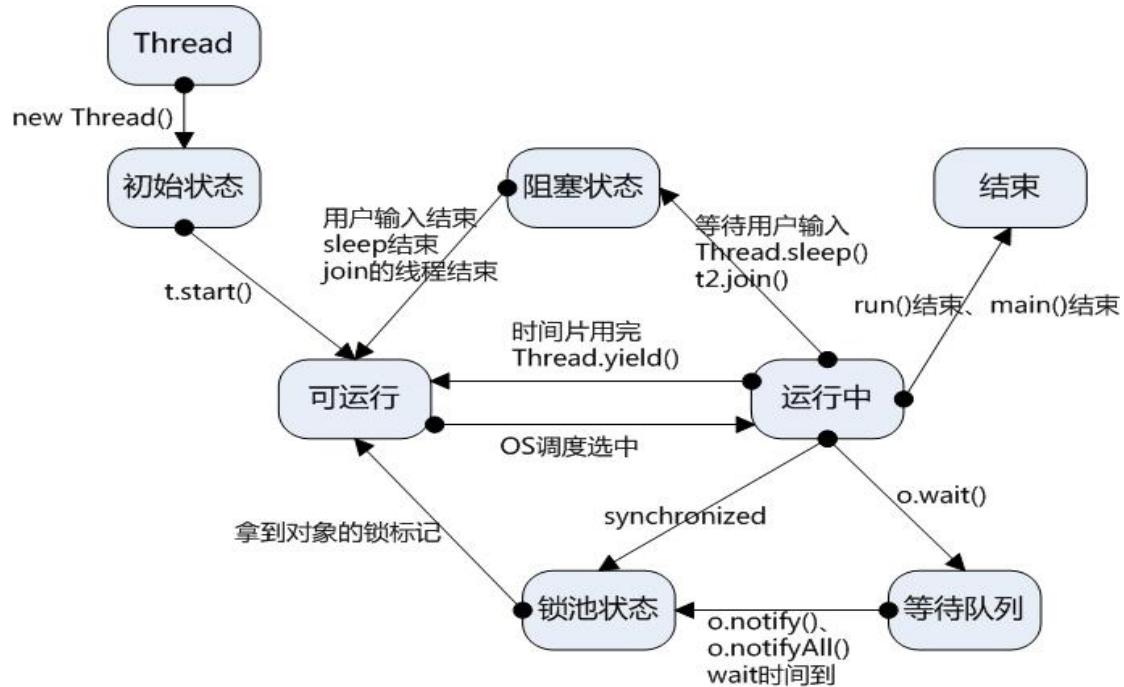
<https://www.cnblogs.com/felixzh/p/6036074.html>

### 4.1 . 如何创建线程？哪种好？

有 4 种方式可以用来创建线程：1. 继承 `Thread` 类 2. 实现 `Runnable` 接口 3. 应用程序可以使用 `Executor` 框架来创建线程池 4. 实现 `Callable` 接口

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

## 4.2 . 线程状态



- 1、新建状态（New）：新创建了一个线程对象。
- 2、就绪状态（Runnable）：线程对象创建后，其他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，变得可运行，等待获取 CPU 的使用权。
- 3、运行状态（Running）：就绪状态的线程获取了 CPU，执行程序代码。
- 4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃 CPU 使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：
  - (一)、等待阻塞：运行的线程执行 `wait()`方法，JVM 会把该线程放入等待池中。（`wait`会释放持有的锁）
  - (二)、同步阻塞：运行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中。
  - (三)、其他阻塞：运行的线程执行 `sleep()`或 `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，线程重新转入就绪状态。（注意，`sleep`是不会释放持有的锁）
- 5、死亡状态（Dead）：线程执行完了或者因异常退出了 `run()`方法，该线程结束生命周期。

## 4.3 . 一般线程和守护线程的区别

所谓守护线程是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

区别：唯一的区别是判断虚拟机(JVM)何时离开，**Daemon** 是为其他线程提供服务，如果全部的 **User Thread** 已经撤离，**Daemon** 没有可服务的线程，JVM 撤离。也可以理解为守护线程是 JVM 自动创建的线程（但不一定），用户线程是程序创建的线程；比如 JVM 的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。

在使用守护线程时需要注意一下几点：

- (1) `thread.setDaemon(true)` 必须在 `thread.start()` 之前设置，否则会跑出一个 `IllegalThreadStateException` 异常。你不能把正在运行的常规线程设置为守护线程。
- (2) 在 **Daemon** 线程中产生的新线程也是 **Daemon** 的。
- (3) 守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至在一个操作的中间发生中断。

## 4.4 . sleep wait yield notify notifyAll join

### 一. Sleep 与 wait 区别

1. **sleep** 是线程类 (**Thread**) 的方法，导致此线程暂停执行指定时间，给执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复。调用 **sleep** 不会释放对象锁。**sleep()** 使当前线程进入阻塞状态，在指定时间内不会执行。

2. **wait** 是 **Object** 类的方法，对此对象调用 **wait** 方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出 **notify** 方法（或 **notifyAll**）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

区别比较：

- 1、这两个方法来自不同的类分别是 **Thread** 和 **Object**
- 2、最主要是 **sleep** 方法没有释放锁，而 **wait** 方法释放了锁，使得其他线程可以使用同步控制块或者方法。
- 3、**wait**, **notify** 和 **notifyAll** 只能在同步控制方法或者同步控制块里面使用，而 **sleep** 可以在任何地方使用(使用范围)
- 4、**sleep** 必须捕获异常，而 **wait**, **notify** 和 **notifyAll** 不需要捕获异常

(1) **sleep** 方法属于 **Thread** 类中方法，表示让一个线程进入睡眠状态，等待一定的时间之后，自动醒来进入到可运行状态，不会马上进入运行状态，因为线程调度机制恢复线程的运行也需要时间，一个线程对象调用了 **sleep** 方法之后，并不会释放他所持有的所有对象锁，所以也就不会影响其他进程对象的运行。但在 **sleep** 的过程中过程中有可能被其他对象调用它的 **interrupt()**，产生 **InterruptedException** 异常，如果你的程序不捕获这个异常，线程就会异常终止，进入 **TERMINATED** 状态，如果你的程序捕获了这个异常，那么程序就会继续执行 **catch** 语句块(可能还有 **finally** 语句块)以及以后的代码。

注意 **sleep()** 方法是一个静态方法，也就是说他只对当前对象有效，通过 **t.sleep()** 让 **t** 对象进入 **sleep**，这样的做法是错误的，它只会是使当前线程被 **sleep** 而不是 **t** 线程

(2) **wait** 属于 **Object** 的成员方法，一旦一个对象调用了 **wait** 方法，必须要采用 **notify()** 和 **notifyAll()** 方法唤醒该进程；如果线程拥有某个或某些对象的同步锁，那么在调用了 **wait()** 后，这个线程就会释放它持有的所有同步资源，而不限于这个被调用了 **wait()** 方法的对象。**wait()** 方法也同样会在 **wait** 的过程中有可能被其他对象调用 **interrupt()** 方法而产生

### 二 yield join notify notifyAll

**yield()** 方法是停止当前线程，让同等优先权的线程或更高优先级的线程有执行的机会。如果没有的话，那么 **yield()** 方法将不会起作用，并且由可执行状态后马上又被执行。

`join` 方法是用于在某一个线程的执行过程中调用另一个线程执行，等到被调用的线程执行结束后，再继续执行当前线程。如：`t.join();`//主要用于等待 t 线程运行结束，若无此句，`main` 则会执行完毕，导致结果不可预测。

`notify` 方法只唤醒一个等待（对象的）线程并使该线程开始执行。所以如果有多个线程等待一个对象，这个方法只会唤醒其中一个线程，选择哪个线程取决于操作系统对多线程管理的实现。

`notifyAll` 会唤醒所有等待(对象的)线程，尽管哪一个线程将会第一个处理取决于操作系统的实现

## 4.5 中断线程

中断线程有很多方法：（1）使用退出标志，使线程正常退出，也就是当 `run` 方法完成后线程终止。（2）通过 `return` 退出 `run` 方法（3）通过对有些状态中断抛异常退出 `thread.interrupt()` 中断。（4）使用 `stop` 方法强行终止线程（过期）

中断线程可能出现的问题：

使用：`Thread.interrupt()`并不能使得线程被中断，线程还是会执行。最靠谱的方法就是设置一个全局的标记位，然后再 `Thread` 中去检查这个标记位，发现标记位改变则中断线程。

```
1 class Example2 extends Thread {  
2     volatile boolean stop = false;  
3     public static void main( String args[] ) throws Exception {  
4         Example2 thread = new Example2();  
5         System.out.println( "Starting thread..." );  
6         thread.start();  
7         Thread.sleep( 3000 );  
8         System.out.println( "Asking thread to stop..." );  
9  
10        thread.stop = true;  
11        Thread.sleep( 3000 );  
12        System.out.println( "Stopping application..." );  
13        //System.exit( 0 );  
14    }  
15  
16    public void run() {  
17        while ( !stop ) {  
18            System.out.println( "Thread is running..." );  
19            long time = System.currentTimeMillis();  
20            while ( (System.currentTimeMillis()-time < 1000) && (!stop) ) {  
21                }  
22            }  
23            System.out.println( "Thread exiting under request..." );  
24        }  
25    }
```

## 4.6 多线程如何避免死锁

什么是死锁？

所谓死锁是指多个进程因竞争资源而造成的一种僵局(互相等待),若无外力作用,这些进程都将无法向前推进。死锁产生的4个必要条件:

- 互斥条件:进程要求对所分配的资源(如打印机)进行排他性控制,即在一段时间内某资源仅为一个进程所占有。此时若有其他进程请求该资源,则请求进程只能等待。
- 不剥夺条件:进程所获得的资源在未使用完毕之前,不能被其他进程强行夺走,即只能由获得该资源的进程自己来释放(只能是主动释放)。
- 请求和保持条件:进程已经保持了至少一个资源,但又提出了新的资源请求,而该资源已被其他进程占有,此时请求进程被阻塞,但对自己已获得的资源保持不放。
- 循环等待条件:存在一种进程资源的循环等待链,链中每一个进程已获得的资源同时被链中下一个进程所请求。

#### 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。

<https://blog.csdn.net/l55718/article/details/51896159>

1. 加锁顺序(线程按照一定的顺序加锁)
2. 加锁时限(线程尝试获取锁的时候加上一定的时限,超过时限则放弃对该锁的请求,并释放自己占有的锁)
3. 死锁检测

## 4.7 多线程的好处以及问题

- (1) 发挥多核CPU的优势
- (2) 防止阻塞
- (3) 便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A,单线程编程,那么就要考虑很多,建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务,任务B、任务C、任务D,分别建立程序模型,并通过多线程分别运行这几个任务,那就简单很多了。

问题:线程安全问题

## 4.8 多线程共用一个数据变量注意什么?

16. 多线程如何进行信息交互
  - Object中的方法,wait(),notify(),notifyAll();
17. 多线程共用一个数据变量需要注意什么?
  - 当我们在线程对象(Runnable)中定义了全局变量,run方法会修改该变量时,如果有多个线程同时使用该线程对象,那么就会造成全局变量的值被同时修改,造成错误。
  - ThreadLocal是JDK引入的一种机制,它用于解决线程间共享变量,使用ThreadLocal声明的变量,即使在线程中属于全局变量,针对每个线程来讲,这个变量也是独立的。
  - volatile变量每次被线程访问时,都强迫线程从主内存中重读该变量的最新值,而当该变量发生修改变化时,也会强迫线程将最新的值刷新回主内存中。这样一来,不同的线程都能及时的看到该变量的最新值。

JVM线程死锁，你该如何判断是因为什么？如果用VisualVM，dump线程信息出来，会有哪些信息

- 常常需要在隔两分钟后再次收集一次thread dump，如果得到的输出相同，仍然是大量thread都在等待给同一个地址上锁，那么肯定是死锁了。

## 4.9 线程通信方式

### 1. 同步

同步是指多个线程通过 `synchronized` 关键字这种方式来实现线程间的通信。

### 2. `wait/notify` 机制

## 4.10 线程池

### • 什么是线程池

- 线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

### • 设计一个动态大小的线程池，如何设计，应该有哪些方法

- 一个线程池包括以下四个基本组成部分：

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

- 所包含的方法

- `private ThreadPool()` 创建线程池
- `public static ThreadPool getThreadPool()` 获得一个默认线程个数的线程池
- `public void execute(Runnable task)` 执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器决定
- `public void execute(Runnable[] task)` 批量执行任务,其实只是把任务加入任务队列，什么时候执行有线程池管理器决定
- `public void destroy()` 销毁线程池,该方法保证在所有任务都完成的情况下才销毁所有线程，否则等待任务完成才销毁
- `public int getWorkThreadNumber()` 返回工作线程的个数
- `public int getFinishedTasknumber()` 返回已完成任务的个数,这里的已完成是只出了任务队列的任务个数，可能该任务并没有实际执行完成
- `public void addThread()` 在保证线程池中所有线程正在执行，并且要执行线程的个数大于某一值时。增加线程

### ◦ 池中线程的个数

- `public void reduceThread()` 在保证线程池中有很大一部分线程处于空闲状态，并且空闲状态的线程在小于某一值时，减少线程池中线程的个数

## Java 线程池实现

说了一般线程池有 corePoolSize, maximumPoolSize, 任务队列, 等待时间这几个比较重要参数。(1) 当线程数小于 corePoolSize 时, 直接创建线程执行 task (2) 当线程数大于等于 corePoolSize 时, 将 task 放入队列 (3) 当队列放不了 task 时, 又创建线程来执行 task (4) 当线程数大于 maximumPoolSize 时, 根据不同策略抛弃 task 之类的。又谈了谈 Java 中 Executor 框架对线程池的实现。比如 FixedThreadPool, SingleThreadExecutor 和 CachedThreadPool 之间区别, 和它们构造时, 上述几个参数的不同。

**CachedThreadPool** 如果需要就创建线程, 使用完的线程会暂时缓存, 不会立刻释放, 只有当空闲时间超出一段时间(默认为 60s)后, 线程池才会销毁该线程适用于耗时较短的任务, 任务处理速度 > 任务提交速度

**FixedThreadPool** 定长大小的线程池

**SingleThreadExecutor** 可以确保任何线程中都只有唯一的任务在运行, 相当于 FixedThreadPool (1)

### 4.11. 线程中抛出异常怎么办

当单线程的程序发生一个未捕获的异常时我们可以采用 try....catch 进行异常的捕获, 但是在多线程环境中, 线程抛出的异常是不能用 try....catch 捕获的, 这样就有可能导致一些问题的出现, 比如异常的时候无法回收一些系统资源, 或者没有关闭当前的连接等等。

```
1 package com.exception;
2
3 import java.lang.Thread.UncaughtExceptionHandler;
4
5 public class WitchCaughtThread
6 {
7     public static void main(String args[])
8     {
9         Thread thread = new Thread(new Task());
10        thread.setUncaughtExceptionHandler(new ExceptionHandler());
11        thread.start();
12    }
13 }
14
15 class ExceptionHandler implements UncaughtExceptionHandler
16 {
17     @Override
18     public void uncaughtException(Thread t, Throwable e)
19     {
20         System.out.println("==Exception: "+e.getMessage());
21     }
22 }
```

简单的说, 如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用 Thread.getUncaughtExceptionHandler() 来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException() 方法进行处理。

java 线程池达到提交上限的具体情况  
线程池用法。  
Java 多线程，线程池有哪几类，每一类的差别 .要你设计的话，如何实现一个线程池 线程池的类型，固定大小的线程池内部是如何实现的，等待队列是用了哪一个队列实现  
线程池种类和工作流程（重点讲 newcached 线程池）  
线程池工作原理  
比如 corePoolSize 和 maxPoolSize 这两个参数该怎么调  
线程池使用了什么设计模式  
线程池使用时一般要考虑哪些问题  
线程池的配置 Executor 以及 Connector 的配置、  
AysncTask 每来一个任务都会创建一个线程来执行吗？(否，线程池的方式实现的)  
介绍下 AsyncTask 的实现原理；

## 五 . Java 进阶 ssh/ssm 框架

### 2.1Spring

#### 2.1.0 什么是 Spring 以及优点

##### 1. 什么是 spring？

Spring 是个 java 企业级应用的开源开发框架。Spring 主要用来开发 Java 应用，但是有些扩展是针对构建 J2EE 平台的 web 应用。Spring 框架目标是简化 Java 企业级应用开发，并通过 POJO 为基础的编程模型促进良好的编程习惯。

##### 2. 使用 Spring 框架的好处是什么？

- **轻量：** Spring 是轻量的，基本的版本大约 2MB。
- **控制反转：** Spring 通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
- **面向切面的编程(AOP)：** Spring 支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
- **容器：** Spring 包含并管理应用中对象的生命周期和配置。
- **MVC 框架：** Spring 的 WEB 框架是个精心设计的框架，是 Web 框架的一个很好的替代品。
- **事务管理：** Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
- **异常处理：** Spring 提供方便的 API 把具体技术相关的异常（比如由 JDBC, Hibernate or JDO 抛出的）转化为一致的 unchecked 异常。

#### 2.1.1 ApplicationContext 和 beanfactory 的区别

##### 1. 利用 MessageSource 进行国际化

BeanFactory 是不支持国际化功能的，因为 BeanFactory 没有扩展 Spring 中

`MessageResource` 接口。相反，由于 `ApplicationContext` 扩展了 `MessageResource` 接口，因而具有消息处理的能力(i18N)，具体 `spring` 如何使用国际化，以后章节会详细描述。

### 2. 强大的事件机制(Event)

基本上牵涉到事件(Event)方面的设计，就离不开观察者模式。不明白观察者模式的朋友，最好上网了解下。因为，这种模式在 java 开发中是比较常用的，又是比较重要的。`ApplicationContext` 的事件机制主要通过 `ApplicationEvent` 和 `ApplicationListener` 这两个接口来提供的，和 `java swing` 中的事件机制一样。即当 `ApplicationContext` 中发布一个事件的时，所有扩展了 `ApplicationListener` 的 Bean 都将会接受到这个事件，并进行相应的处理。

### 3. 底层资源的访问

`ApplicationContext` 扩展了 `ResourceLoader`(资源加载器)接口，从而可以用来加载多个 Resource，而 `BeanFactory` 是没有扩展 `ResourceLoader`

4. `.BeanFactroy` 采用的是延迟加载形式来注入 Bean 的，即只有在使用到某个 Bean 时(调用 `getBean()`)，才对该 Bean 进行加载实例化，这样，我们就不能发现一些存在的 Spring 的配置问题。而 `ApplicationContext` 则相反，它是在容器启动时，一次性创建了所有的 Bean。这样，在容器启动时，我们就可以发现 Spring 中存在的配置错误。

## 2.1.2 Spring Bean 生命周期

Spring 上下文中的 Bean 也类似，如下

- 1、实例化一个 Bean——也就是我们常说的 `new`；
- 2、按照 Spring 上下文对实例化的 Bean 进行配置——也就是 IOC 注入；
- 3、如果这个 Bean 已经实现了 `BeanNameAware` 接口，会调用它实现的 `setBeanName(String)` 方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值
- 4、如果这个 Bean 已经实现了 `BeanFactoryAware` 接口，会调用它实现的 `setBeanFactory(BeanFactory)` 传递的是 Spring 工厂自身（可以用这种方式来获取其它 Bean，只需在 Spring 配置文件中配置一个普通的 Bean 就可以）；
- 5、如果这个 Bean 已经实现了 `ApplicationContextAware` 接口，会调用 `setApplicationContext(ApplicationContext)` 方法，传入 Spring 上下文（同样这个方式也可以实现步骤 4 的内容，但比 4 更好，因为 `ApplicationContext` 是 `BeanFactory` 的子接口，有更多的实现方法）；
- 6、如果这个 Bean 关联了 `BeanPostProcessor` 接口，将会调用 `postProcessBeforeInitialization(Object obj, String s)` 方法，`BeanPostProcessor` 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用那个的方法，也可以被应用于内存或缓存技术；
- 7、如果 Bean 在 Spring 配置文件中配置了 `init-method` 属性会自动调用其配置的初始化方法。

8、如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法、；

注：以上工作完成以后就可以应用这个 Bean 了，那这个 Bean 是一个 Singleton 的，所以一般情况下我们调用同一个 id 的 Bean 会是在内容地址相同的实例，当然在 Spring 配置文件中也可以配置非 Singleton，这里我们不做赘述。

9、当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用那个其实现的 destroy()方法；

10、最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

### 2.1.3 spring 中 bean 的作用域

#### 默认是单例

| 作用域 | 字符        | 描述           |
|-----|-----------|--------------|
| 单例  | singleton | 整个应用中只创建一个实例 |
| 原型  | prototype | 每次注入时都新建一个实例 |
| 会话  | session   | 为每个会话创建一个实例  |
| 请求  | request   | 为每个请求创建一个实例  |

#### 2.1.4 Spring IOC

[https://blog.csdn.net/it\\_man/article/details/4402245](https://blog.csdn.net/it_man/article/details/4402245)

IOC：IOC 利用 java 反射机制，AOP 利用代理模式。所谓控制反转是指，本来被调用者的实例是有调用者来创建的，这样的缺点是耦合性太强，IOC 则是统一交给 spring 来管理创建，将对象交给容器管理，你只需要在 spring 配置文件总配置相应的 bean，以及设置相关的属性，让 spring 容器来生成类的实例对象以及管理对象。在 spring 容器启动的时候，spring 会把你配置在配置文件中的 bean 都初始化好，然后在你需要调用的时候，就把它已经初始化好的那些 bean 分配给你需要调用这些 bean 的类。

IoC 的一个重点是在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过 DI (Dependency Injection，依赖注入) 来实现的。比如对象 A 需要操作数据库，以前我们总是要在 A 中自己编写代码来获得一个 Connection 对象，有了 spring 我们就只需要告诉 spring，A 中需要一个 Connection，至于这个 Connection 怎么构造，何时构造，A 不需要知道。在系统运行时，spring 会在适当的时候制造一个 Connection，然后像打针一样，注射到 A 当中，这样就完成了对各个对象之间关系的控制。A 需要依赖 Connection 才能正常运行，而这个 Connection 是由 spring 注入到 A 中的，依赖注入的名字就这么来的。那么 DI 是如何实现的呢？Java 1.3 之后一个重要特征是反射 (reflection)，它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring 就是通过反射来实现注入的。

#### 2.1.5 Spring AOP

##### 2.5.1.1.什么是 AOP

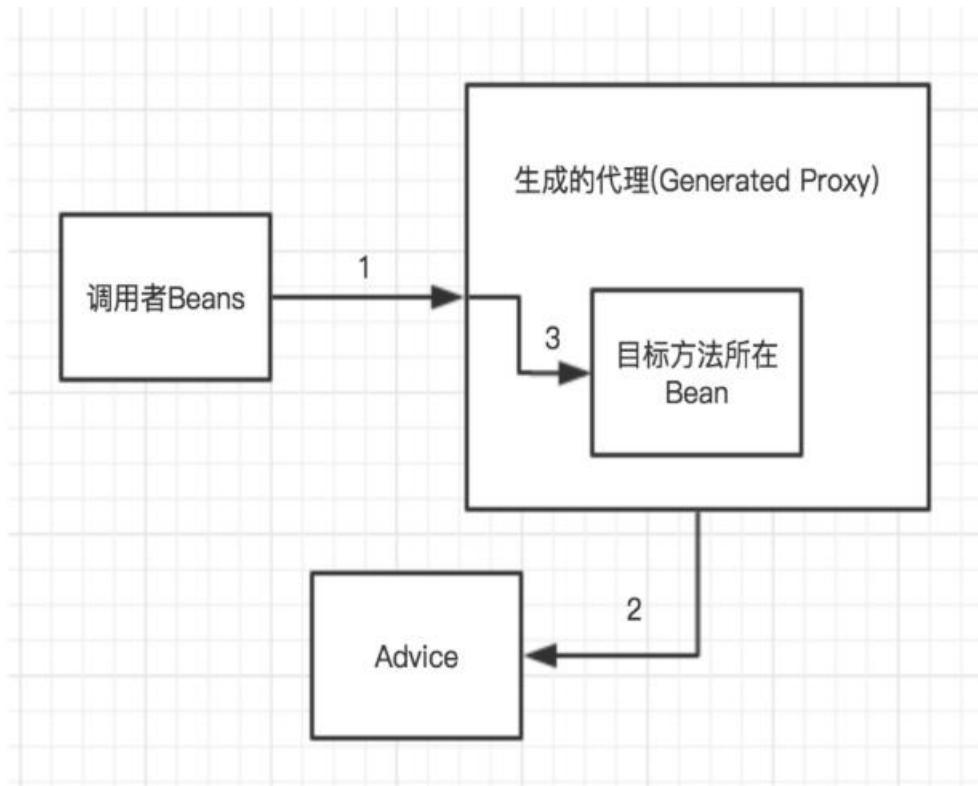
AOP 技术利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

## 2.AOP 原理

基于动态代理

<https://www.cnblogs.com/CHENJIAO120/p/7080790.html>

首先，这是一种基于代理(Proxy)的实现方式。下面这张图很好地表达了这层关系：



它们之间的调用先后次序反映在上图的序号中：

调用者 Bean 尝试调用目标方法，但是被生成的代理截住了  
代理根据 Advice 的种类(本例中是@Before Advice)，对 Advice 首先进行调用  
代理调用目标方法  
返回调用结果给调用者 Bean(由代理返回，没有体现在图中)

### 2.1.5.2 基于接口的动态代理

<http://www.cnblogs.com/xiaolu0501395377/p/3383130.html>

在 java 的动态代理机制中，有两个重要的类或接口，一个是 `InvocationHandler`(`I  
nterface`)、另一个则是 `Proxy`(`Class`)，这一个类和接口是实现我们动态代理所必须用到的。首先我们先来看看 `java` 的 API 帮助文档是怎么样对这两个类进行描述的：

#### 1. `InvocationHandler`:

每一个动态代理类都必须要实现 `InvocationHandler` 这个接口，并且每个代理类的实例都关联到了一个 `handler`，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由 `InvocationHandler` 这个接口的 `invoke` 方法来进行调用。我们来看看 `Inv  
ocationHandler` 这个接口的唯一一个方法 `invoke` 方法

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable  
  
proxy: 指代我们所代理的那个真实对象  
method: 指代的是我们所要调用真实对象的某个方法的 Method 对象  
args: 指代的是调用真实对象某个方法时接受的参数
```

#### 2. `Proxy`

`Proxy` 这个类的作用就是用来动态创建一个代理对象的类，它提供了许多的方法，但是我们用的最多的就是 `newProxyInstance` 这个方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, Inv  
ocationHandler h) throws IllegalArgumentException
```

### 2.5.2.4 实现子类的动态代理

`CGLib` 动态代理是通过继承业务类，生成的动态代理类是业务类的子类，通过重写业务方法进行代理；

## cglib 动态代理

`CGLib` 创建的动态代理对象性能比 `JDK` 创建的动态代理对象的性能高不少，但是 `CGLib` 在创建代理对象时所花费的时间却比 `JDK` 多得多，所以对于单例的对象，因为无需频繁创建对象，用 `CGLib` 合适，反之，使用 `JDK` 方式要更为合适一些。同时，由于 `CGLib` 由于是采用动态创

建子类的方法，对于 `final` 方法，无法进行代理。

运行时动态的生成一个被代理类的子类（通过 `ASM` 字节码处理框架实现），子类重写了被代理类中所有非 `final` 的方法。在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势植入横切逻辑。

## 2.1.6 事务

### 2.1.6.1 事务的实现方式

(1) 编程式事务管理对基于 `POJO` 的应用来说是唯一选择。我们需要在代码中调用 `beginTransaction()`、`comm  
it()`、`rollback()` 等事务管理相关的方法，这就是编程式事务管理。

- (2) 基于 `TransactionProxyFactoryBean` 的声明式事务管理
- (3) 基于 `@Transactional` 的声明式事务管理
- (4) 基于 `AspectJ AOP` 配置事务

根据代理机制的不同，`Spring` 事务的配置又有几种不同的方式：

第一种方式：每个 `Bean` 都有一个代理

第二种方式：所有 Bean 共享一个代理基类

第三种方式：使用拦截器

第四种方式：使用 tx 标签配置的拦截器

第五种方式：全注解

### 2.1.6.2 事务的传播级别

1) PROPAGATION\_REQUIRED，默认的 spring 事务传播级别，使用该级别的特点是，如果上下文中已经存在事务，那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

2) PROPAGATION\_SUPPORTS，从字面意思就知道，supports，支持，该传播级别的特点是，如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行。所以说，并非所有的包在 transactionTemplate.execute 中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3) PROPAGATION\_MANDATORY，该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4) PROPAGATIONQUIRES\_NEW，从字面即可知道，new，每次都要一个新事务，该传播级别的特点是，每次都会新建一个事务，并且同时将上下文中的事务挂起，**执行当前新建事务完成以后，上下文事务恢复再执行。**

这是一个很有用的传播级别，举一个应用场景：现在有一个发送 100 个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送 100 封红包，然后再记录发送日志，发送日志要求 100% 的准确，如果日志不准确，那么整个父事务逻辑需要回滚。

怎么处理整个业务需求呢？就是通过这个 PROPAGATION\_REQUIRES\_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5) PROPAGATION\_NOT\_SUPPORTED，这个也可以从字面得知，not supported，不支持，当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环 1000 次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6) PROPAGATION\_NEVER，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而 PROPAGATION\_NEVER 传播级别要求上下文中不能存在事务，一旦有事务，就抛出 runtime 异常，强制停止执行！这个级别上辈子跟事务有仇。

7) PROPAGATION\_NESTED，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

### 2.1.6.3 事务的嵌套失效

那么什么是嵌套事务呢？很多人都不理解，我看过去一些博客，都是有些理解偏差。

嵌套是子事务套在父事务中执行，子事务是父事务的一部分，在进入子事务之前，父事务建立一个回滚点，叫 **save point**，然后执行子事务，这个子事务的执行也算是父事务的一部分，然后子事务执行结束，父事务继续执行。重点就在于那个 **save point**。看几个问题就明了了：

如果子事务回滚，会发生什么？

父事务会回滚到进入子事务前建立的 **save point**，然后尝试其他的事务或者其他的业务逻辑，父事务之前的操作不会受到影响，更不会自动回滚。

如果父事务回滚，会发生什么？

父事务回滚，子事务也会跟着回滚！为什么呢，因为父事务结束之前，子事务是不会提交的，我们说子事务是父事务的一部分，正是这个道理。那么：

事务的提交，是什么情况？

是父事务先提交，然后子事务提交，还是子事务先提交，父事务再提交？答案是第二种情况，还是那句话，子事务是父事务的一部分，由父事务统一提交。

## 2.1.7 Spring MVC

### 2.1.7.0 什么是 Spring MVC

Spring MVC 是一个基于 MVC 架构的用来简化 web 应用程序开发的应用开发框架，它是 Spring 的一个模块，无需中间整合层来整合，它和 Struts2 一样都属于表现层的框架。在 web 模型中，MVC 是一种很流行的框架，通过把 Model, View, Controller 分离，把较为复杂的 web 应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

### 2.1.7.1 Spring MVC 执行流程（工作原理）

答：1. 用户发送请求至前端控制器 DispatcherServlet

2. DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3. 处理器映射器根据请求 url 找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4. DispatcherServlet 通过 HandlerAdapter 处理器适配器调用处理器

5. 执行处理器(Controller，也叫后端控制器)。

6. Controller 执行完成返回 ModelAndView

7. HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet

8. DispatcherServlet 将 ModelAndView 传给 ViewResolver 视图解析器

9. ViewResolver 解析后返回具体 View

10. DispatcherServlet 对 View 进行渲染视图（即将模型数据填充至视图中）。

## 11.DispatcherServlet 响应用

### 2. 1. 7. 2 SpringMVC 加载流程

springmvc 加载流程

- 1.Servlet 加载（监听器之后即执行）Servlet 的 init()
- 2.加载配置文件
- 3.从 ServletContext 拿到 spring 初始化 springmvc 相关对象
- 4.放入 ServletContext

### 2. 1. 7. 2 springMVC 和 struts2 的区别

(1) springmvc 的入口是一个 servlet 即前端控制器（DispatcherServlet），而 struts2 入口是一个 filter 过滤器（StrutsPrepareAndExecuteFilter）。

(2) springmvc 是基于方法开发（一个 url 对应一个方法），请求参数传递到方法的形参，可以设计为单例或多例（建议单例），struts2 是基于类开发，传递参数是通过类的属性，只能设计为多例。

(3) Struts 采用值栈存储请求和响应的数据，通过 OGNL 存取数据，springmvc 通过参数解析器是将 request 请求内容解析，并给方法形参赋值，将数据和视图封装成 ModelAndView 对象，最后又将 ModelAndView 中的模型数据通过 request 域传输到页面。Jsp 视图解析器默认使用 jstl。

## 2.1.8 Spring 中设计模式

<https://www.cnblogs.com/jifeng/p/7398852.html>

### 第一种：简单工厂

又叫做静态工厂方法（StaticFactory Method）模式，但不属于 23 种 GOF 设计模式之一。

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

spring 中的 BeanFactory 就是简单工厂模式的体现，根据传入一个唯一的标识来获得 bean 对象，但是是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。如下配置，就是在 HelloItxxz 类中创建一个 itxxzBean。

### 第三种：单例模式（Singleton）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

spring 中的单例模式完成了后半句话，即提供了全局的访问点 BeanFactory。但没有从构造器级别去控制单例，这是因为 spring 管理的是任意的 java 对象。

核心提示点：Spring 下默认的 bean 均为 singleton，可以通过 singleton="true|false" 或者 scope="?" 来指定

### 第四种：适配器（Adapter）

在 Spring 的 Aop 中，使用的 Advice（通知）来增强被代理类的功能。Spring 实现这一 AOP 功能的原理就使用代理模式（1、JDK 动态代理。2、CGLib 字节码生成技术代理。）对类进行方法级别的切面增强，即，生成被代理类的代理类，并在代理类的方法前，设置拦截器，通过执行拦截器重的内容增强了代理方法的功能，实现的面向切面编程。

### 第六种：代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。从结构上来看和 Decorator 模式类似，但 Proxy 是控制，更像是一种对功能的限制，而 Decorator 是增加职责。

spring 的 Proxy 模式在 aop 中有体现，比如 `JdkDynamicAopProxy` 和 `Cglib2AopProxy`。

#### 第七种：观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

spring 中 Observer 模式常用的地方是 `listener` 的实现。如 `ApplicationListener`。

#### 第八种：策略（Strategy）

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

spring 中在实例化对象的时候用到 Strategy 模式

在 `SimpleInstantiationStrategy` 中有如下代码说明了策略模式的使用情况：

#### 第九种：模板方法（Template Method）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method 模式一般是需要继承的。这里想要探讨另一种对 Template Method 的理解。spring 中的 `JdbcTemplate`，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到 `JdbcTemplate` 已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入 `JdbcTemplate` 的方法中。但是变化的东西是一段代码，而且这段代码会用到 `JdbcTemplate` 中的变量。怎么办？那我们就用回调对象吧。在这个回调对象中定义一个操纵 `JdbcTemplate` 中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到 `JdbcTemplate`，从而完成了调用。这可能是 Template Method 不需要继承的另一种实现方式吧。

以下是一个具体的例子：

`JdbcTemplate` 中的 `execute` 方法

Spring 的加载流程，Spring 的源码中 Bean 的构造的流程

Spring 事务源码，IOC 源码，AOP 源码

spring 的作用及理解

事务怎么配置

Spring 的 annotation 如何实现

SpringMVC 工作原

了解 SpringMVC 与 Struct2 区别

了解 SpringMVC 请求流程

springMVC 和 spring 是什么关系

项目中 Spring 的 IOC 和 AOP 具体怎么使用的

spring mvc 底层实现原理

动态代理的原理

如果使用 spring mvc，那 post 请求跟 put 请求有什么区别啊；

然后开始问 springmvc 描述从 tomcat 开始到 springmvc 返回到前端显示的整个流程...

接着问 springmvc 中的 handlerMapping 的内部实现；

.然后又问 spring 中从载入 xml 文件到 getbean 整个流程，描述一遍

## 2.2 Servlet

### 2.2.1 Servlet 生命周期

1. 创建 **Servlet** 对象，通过服务器反射机制创建 **Servlet** 对象，第一次请求时才会创建。（默认）
2. 调用 **Servlet** 对象的 **init()**方法，初始化 **Servlet** 的信息，**init()**方法只会在创建后被调用一次；
3. 响应请求，调用 **service()**或者是 **doGet()**, **doPost()**方法来处理请求，这些方法是运行在多线程状态下的。
4. 在长时间没有被调用或者是服务器关闭时，会调用 **destroy()**方法来销毁 **Servlet** 对象。

### 2.2.2 servlet 是什么

**Servlet 定义：**Servlet 是基于 Java 技术的 Web 组件，由容器管理并产生动态的内容。  
**Servlet 引擎作为 WEB 服务器**

的扩展提供支持 **Servlet** 的功能。**Servlet** 与客户端通过 **Servlet 容器**实现的请求/响应模型进行交互

**Servlet** 知道是做什么的吗？和 **JSP** 有什么联系？**JSP** 的运行原理？**JSP** 属于 Java 中的吗？

**Servlet** 是线程安全

**scala** 写的大型框架吗 **spark**

如何确保分布式环境下异步消息处理的顺序性？

**servlet** 是单例

**servlet** 和 **filter** 的区别。

**servlet** 流程

## 2.2 Struts

### 2.2.1 Struts 工作流程

二 工作流程

- 1、客户端浏览器发出 HTTP 请求。
- 2、根据 **web.xml** 配置，该请求被 **FilterDispatcher** 接收
- 3、根据 **struts.xml** 配置，找到需要调用的 **Action** 类和方法，并通过 **IoC** 方式，将值注入给 **Action**
- 4、**Action** 调用业务逻辑组件处理业务逻辑，这一步包含表单验证。
- 5、**Action** 执行完毕，根据 **struts.xml** 中的配置找到对应的返回结果 **result**，并跳转到相应页面
- 6、返回 HTTP 响应到客户端浏览器

### 2.2.2 Struts 工作原理

## 工作原理

在 Struts2 框架中的处理大概分为以下几个步骤

- 1 客户端初始化一个指向 **Servlet** 容器（例如 **Tomcat**）的请求
- 2 这个请求经过一系列的过滤器（**Filter**）（这些过滤器中有一个叫做 **ActionContextCleanUp** 的可选过滤器，这个过滤器对于 **Struts2** 和其他框架的集成很有帮助，例如： **SiteMesh Plugin**）
- 3 接着 **FilterDispatcher** 被调用，**FilterDispatcher** 询问 **ActionMapper** 来决定这个请是否需要调用某个 **Action**
- 4 如果 **ActionMapper** 决定需要调用某个 **Action**，**FilterDispatcher** 把请求的处理交给 **ActionProxy**
- 5 **ActionProxy** 通过 **Configuration Manager** 询问框架的配置文件，找到需要调用的 **Action** 类
- 6 **ActionProxy** 创建一个 **ActionInvocation** 的实例。
- 7 **ActionInvocation** 实例使用**命名(令)模式**来调用，在调用 **Action** 的过程前后，涉及到相关拦截器（**Interceptor**）的调用。
- 8 一旦 **Action** 执行完毕，**ActionInvocation** 负责根据 **struts.xml** 中的配置找到对应的返回结果。返回结果通常是（但不总是，也可能是一个 **Action** 链）一个需要被表示的 **JSP** 或者 **FreeMarker** 的模版。在表示的过程中可以使用 **Struts2** 框架中继承的标签。在这个过程中需要涉及到 **ActionMapper**

### 2.2.3 do Filter

**doFilter** 是过滤器的执行方法，它拦截提交的 **HttpServletRequest** 请求，**HttpServletResponse** 响应，作为 **struts2** 的核心拦截器

```
[java] [运行] [复位]

public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException {
    //父类向子类转：强转为http请求、响应
    HttpServletRequest request = (HttpServletRequest) req;
    HttpServletResponse response = (HttpServletResponse) res;

    try {
        //设置编码和国际化
        prepare.setEncodingAndLocale(request, response);
        //创建Action上下文（重点）
        prepare.createActionContext(request, response);
        //把当前Dispatcher实例放入ThreadLocal里面
        prepare.assignDispatcherToThread();
        if (excludedPatterns != null && prepare.isUrlExcluded(request, excludedPatterns)) {
            //如果当前路径是排斥的路径,那么此过滤器不拦截
            chain.doFilter(request, response);
        } else {
            //包装这个Request(如果是multipart/form-data,包装成MultiPartRequestWrapper, 否则包装成StrutsRequestWrapper)
            request = prepare.wrapRequest(request);
            //寻找Action的映射
            ActionMapping mapping = prepare.findActionMapping(request, response, true);
            if (mapping == null) {
                //如果没有映射,则需要判断是否是静态资源
                boolean handled = execute.executeStaticResourceRequest(request, response);
                if (!handled) {
                    chain.doFilter(request, response);
                }
            } else {
                execute.executeAction(request, response, mapping);
            }
        }
    } finally {
        prepare.cleanupRequest(request);
    }
}
```

Filter 主要有三层过滤器，ActionContextCleanUp、SiteMesh、FilterDispatcher

ActionContextCleanUp 作用：FilterDispatcher 检测到这个属性,就不会清除 ActionContext 中的内容了,而由 ActionContextCleanUp 后续的代码来清除,保证了一系列的 Filter 访问正确的 ActionContext.

SiteMesh 的介绍就不多说了，主要是用来统一页面风格，减少重复编码的。

它定义了一个过滤器，然后把页面都加上统一的头部和底部

FilterDispatcher 拦截请求和回复，读取 `structs.xml`，创建对应的 action 对象

## 2.2.4 拦截器与过滤器的区别

- 1、拦截器是基于 Java 的反射机制的，而过滤器是基于函数回调。
- 2、拦截器不依赖与 servlet 容器，过滤器依赖与 servlet 容器。
- 3、拦截器只能对 action 请求起作用，而过滤器则可以对几乎所有的请求起作用。
- 4、拦截器可以访问 action 上下文、值栈里的对象，而过滤器不能访问。
- 5、在 action 的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次
- 6、执行顺序：过滤前 - 拦截前 - Action 处理 - 拦截后 - 过滤后。  
过滤是一个横向的过程，首先把客户端提交的内容进行过滤(例如未登录用户不能访问内部页面的处理)；过滤通过后，拦截器将检查用户提交数据的验证，做一些前期的数据处理，接着把处理后的数据发给对应的 Action；Action 处理完成返回后，拦截器还可以做其他过程(还没想到要做啥)，再向上返回到过滤器的后续操作。

## 2.2.5 Struts 中为什么不用考虑线程安全

面试题常见问题: 为什么在 Struts2 中我们不用考虑线程安全？为什么交给 Spring 管理的时候我们要把 Bean 的 scope 属性设置成 prototype？

答: 从上述源码中我们看到了，每一次被 Struts2 的这个过滤器拦截的时候，都会调用 `createActionContext` 方法，无论当前是否有 `ActionContext` 实例存在，都会 `new` 一个 `ActionContext`，只不过是如果是 action 之间转发的请求，那么 `new` 的时候会把之前的那个 `ActionContext` 放进去.. 而每个 `ActionContext` 都只与自己的 `ThreadLocal` 挂钩，所以是不用

考虑线程安全的。然而后面读源码也会读到，每次请求都会产生一个 Action 的实例，而 Spring 默认创建实例是单例的。

## 2.2.6 Struts2 和 Struts1 区别

- Action 类: • Struts1 要求 Action 类继承一个抽象基类。• Struts 2 Action 类可以实现一个 Action 接口，也可继承 ActionSupport 基类去实现常用的接口。Action 接口不是必须的，任何有 execute 标识的对象都可以用作 Struts2 的 Action 对象。
- 线程模式: • Struts1 Action 是单例模式并且必须是线程安全的，• Struts2 Action 对象为每一个请求产生一个实例，因此没有线程安全问题。（实际上，servlet 容器给每个请求产生许多可丢弃的对象，并且不会导致性能和垃圾回收问题）
- Action 执行的控制: • Struts1 支持每一个模块有单独的 Request Processors（生命周期），但是模块中的所有 Action 必须共享相同的生命周期。• Struts2 支持通过拦截器堆栈（Interceptor Stacks）为每一个 Action 创建不同的生命周期。堆栈能够根据需要和不同的 Action 一起使用。
- Servlet 依赖: • Struts1 Action 依赖于 Servlet API，因为当一个 Action 被调用时 `HttpServletRequest` 和 `HttpServletResponse` 被传递给 `execute` 方法。• Struts 2 Action 不依赖于容器（，允许 Action 脱离容器单独被测试。如果需要，Struts2 Action 仍然可以访问初始的 `request` 和 `response`。但是，其他的元素减少或者消除了直接访问 `HttpServletRequest` 和 `HttpServletResponse` 的必要性。）
- 可测性: • 测试 Struts1 Action 的一个主要问题是 `execute` 方法暴露了 servlet API（这使得测试要依赖于容器）。一个第三方扩展——Struts TestCase——提供了一套 Struts1 的模拟对象（来进行测试）。• Struts 2 Action 可以通过初始化、设置属性、调用方法来测试，“依赖注入”支持也使测试更容易。

## 2.3 Hibernate

Hibernate 的生成策略，主要说了 native 、uuid  
Hibernate 与 Mybatis 区别

## 2.4 Redis

Redis 数据结构 Redis 持久化机制  
Redis 的一致性哈希算法  
redis 了解多少 redis 五种数据类型, 当散列类型的 value 值非常大的时候怎么进行压缩,  
用 redis 怎么实现摇一摇与附近的人功能, redis 主从复制过程,  
Redis 如何解决 key 冲突  
redis 的五种数据结构  
redis 是怎么存储数据的  
redis 使用场景

## 2.5 Tomcat

Tomcat 的结构  
tomcat 均衡方式 , netty

## 2.6 netty

netty 源码

## 2.7 Hadoop

## 2.8 Volley

Volley 的原理及使用

springMVC 和 spring 是什么关系

一些 java 的常用框架的架构  
.Servlet 的 Filter 用的什么设计模式  
Spring 读过哪些源码吗  
利用 Bean 的初始化可以做什么事情  
排行榜可以使用 redis 哪种数据结构  
RESTful 架构

Hibernate、Mybatis 与 JDBC 区别

springmvc 的流程 一个请求来了之后如何处理 (handler 链)

框架封装 jdbc 受检异常的考虑和原因？

zookeeper 的常用功能，自己用它来做什么

hadoop/spark/impala/lucene/RocksDB/redis 这些框架的技术点

ibatis 跟 hibernate 的区别

ibatis 是怎么实现映射的，它的映射原理是什么

redis 的操作是不是原子操作

秒杀业务场景设计

WebSocket 长连接问题

如何设计淘宝秒杀系统（重点关注架构，比如数据一致性，数据库集群一致性哈希，缓存，分库分表等等）？

List 接口去实例化一个它的实现类 (ArrayList) 以及直接用 ArrayList 去 new 一个该类的对象，这两种方式有什么区别，为什么大多数情况下会用到

Tomcat 关注哪些参数

Mapreduce

Spring 配置过滤器 和 Struts 的拦截器配置与使用。

对后台的优化有了解吗？比如负载均衡。我给面试官说了 Nginx+Tomcat 负载均衡，异步处理(消息缓冲服务器)，缓存(Redis, Memcache)，

NoSQL，数据库优化，存储索引优化

开源项目

Netty 框架源码看过吗

MapReduce

Volley 机制讲完之后问我 Volley 的缺点是什么，怎么改进

对 Restful 了解

Restful 的认识，优点，以及和 soap 的区别

lruCache 的基本原理

service 中启动方式有哪些区别是？

## 六 . Java 内存模型 和 垃圾回收

### 3.0 什么是 JMM 内存模型？(JMM 和内存区域划分不是一回事)

<https://blog.csdn.net/javazejian/article/details/72772461>

大体：JMM 就是一组规则，这组规则意在解决在并发编程可能出现的线程安全问题，JMM (Java Memory Model) 是 Java 内存模型，JMM 定义了程序中各个共享变量的访问规则，即在虚拟机中将变量存储到内存和从内存读取变量这样的底层细节，并提供了内置解决方案（happen-before 原则）及其外部可使用的同步手段(synchronized/volatile 等），确保了程序执行在多线程环境中的应有的原子性，可视性及其有序性。

2.JMM 规定了所有的变量都存储在主内存 (Main Memory) 中。每个线程还有自

己的工作内存（Working Memory），线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（**volatile** 变量仍然有工作内存的拷贝，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。

### 3.0.1 JMM 中的 happens-before 原则

**happens-before** 原则内容如下

程序顺序原则，即在一个线程内必须保证语义串行性，也就是说按照代码顺序执行。

锁规则 解锁(unlock)操作必然发生在后续的同一个锁的加锁(lock)之前，也就是说，如果对于一个锁解锁后，再加锁，那么加锁的动作必须在解锁动作之后(同一个锁)。

**volatile** 规则 **volatile** 变量的写，先发生于读，这保证了 **volatile** 变量的可见性，简单的理解就是，**volatile** 变量在每次被线程访问时，都强迫从主内存中读该变量的值，而当该变量发生变化时，又会强迫将最新的值刷新到主内存，任何时刻，不同的线程总是能够看到该变量的最新值。

线程启动规则 线程的 **start()**方法先于它的每一个动作，即如果线程 A 在执行线程 B 的 **start** 方法之前修改了共享变量的值，那么当线程 B 执行 **start** 方法时，线程 A 对共享变量的修改对线程 B 可见

传递性 A 先于 B ， B 先于 C 那么 A 必然先于 C

线程终止规则 线程的所有操作先于线程的终结，**Thread.join()**方法的作用是等待当前执行的线程终止。假设在线程 B 终止之前，修改了共享变量，线程 A 从线程 B 的 **join** 方法成功返回后，线程 B 对共享变量的修改将对线程 A 可见。

线程中断规则 对线程 **interrupt()**方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过 **Thread.interrupted()**方法检测线程是否中断。

对象终结规则 对象的构造函数执行，结束先于 **finalize()**方法

为什么要实现内存模型？

- 内存模型的就是为了在现代计算机平台中保证程序可以正确性的执行，但是不同的平台实现是不同的。
- 编译器中生成的指令顺序，可以与源代码中的顺序不同；
- 编译器可能把变量保存在寄存器而不是内存中；
- 处理器可以采用乱序或并行等方式来执行指令；
- 缓存可能会改变将写入变量提交到主内存的次序；
- 保存在处理器本地缓存中的值，对其他处理器是不可见的；

`Student s = new Student();`在内存中做了哪些事情？

- 加载Student.class文件进内存
- 在栈内存为s开辟空间
- 在堆内存为学生对象开辟空间
- 对学生对象的成员变量进行默认初始化
- 对学生对象的成员变量进行显示初始化
- 通过构造方法对学生对象的成员变量赋值
- 学生对象初始化完毕，把对象地址赋值给s变量

### 3.1 内存分区

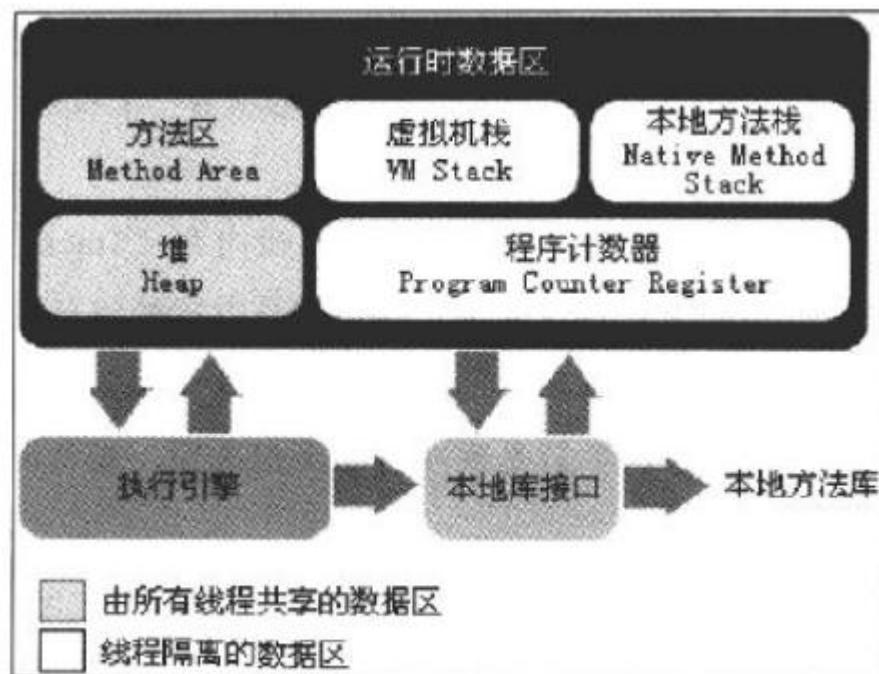


图 2-1 Java 虚拟机运行时数据区

- 堆
  - 存放对象的实例，数组
  - 线程共享，最大，虚拟机启动时创建
  - 堆上无内存可完成实例分配且堆无法扩展时->OutOfMemoryError
  - 逻辑上连续，-Xmx -Xms控制堆大小
- 方法区（永久代）
  - 存放虚拟机加载的类信息，常量，静态变量，（即时编译器编译后的代码）
  - 无法满足内存分配需求时->OutOfMemoryError
- 虚拟机栈
  - 有个局部变量表，存放了基本数据类型，对象的引用，returnAddress类型（指向一条字节码指令的地址）
  - 线程请求栈深度大于虚拟机所允许深度->StackOverflowError
  - 虚拟机栈扩展时无法申请到足够的内存->OutOfMemoryError
- 本地方法栈
  - 与虚拟机栈类似，区别是为本地方法服务
- 程序计数器
  - 记录正在执行的虚拟机字节码指令地址
  - 执行本地方法时计数器为空（undefined）
  - 线程私有，唯一一个无OutOfMemoryError的区域
- 运行时常量池
  - 方法区的一部分，存放编译期生成的字面量，和符号引用
  - 常量池无法申请内存时->OutOfMemoryError

### 什么是堆中的永久代？

答：永久代是用于存放静态文件，如 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类，持久代中一般包含：

- 类的方法(字节码...)
- 类名(String 对象)
- .class 文件读到的常量信息
- class 对象相关的对象列表和类型列表 (e.g., 方法对象的 array).
- JVM 创建的内部对象
- JIT 编译器优化用的信息

- 程序计数器(线程私有)
  - 线程创建时创建，执行本地方法时其值为undefined。
- 虚拟机栈(线程私有)
  - (栈内存) 为虚拟机执行java方法服务：方法被调用时创建栈帧-->局部变量表->局部变量、对象引用
  - 如果线程请求的栈深度超出了虚拟机所允许的深度，就会出现StackOverflowError。-Xss规定了栈的最大空间
  - 虚拟机栈可以动态扩展，如果扩展到无法申请到足够的内存，会出现OOM
- 本地方法栈(线程私有)
  - java虚拟机栈是为虚拟机执行java方法服务的，而本地方法栈则为虚拟机执行到的Native方法服务。
  - Java虚拟机没有对本地方法栈的使用和数据结构做强制规定。Sun HotSpot把Java虚拟机栈和本地方法栈合二为一
  - 会抛出StackOverflowError和OutOfMemoryError
- Java堆
  - 被所有线程共享，在Java虚拟机启动时创建，几乎所有的对象实例都存放到堆中
  - GC的主要区域
  - 物理不连续，逻辑上连续，并可以动态扩展，无法扩展时抛出OutOfMemoryError
- 方法区
  - 用于存储已被虚拟机加载的类信息、常量、静态变量、即使编译器编译后的代码的数据
  - Sun HotSpot虚拟机把方法区叫做永久代(Permanent Generation)
- 运行时常量池
  - 受到方法区的限制，抛出OutOfMemoryError

## 3.2 GC 算法 (YGC and FGC)

### 1.什么样的对象需要回收？

- 对象到GC Roots没有引用链，那么这个对象不可用，需要回收

### 2.可作为GC Roots的对象？

- 虚拟机栈中引用的对象
- 方法区内类静态属性引用的对象
- 方法区内常量引用的对象
- 本地方法栈中JNI (Native方法) 引用的对象

### 3.有哪些GC算法

#### • 标记-清除算法

- 标记所有要回收对象，标记完成后统一回收所有被标记对象
- 缺点是标记和清除两个过程效率低
- 而且清除之后会产生大量的不连续的内存碎片，导致分配较大对象时无法找到足够的连续内存空间，而提前触发另一次垃圾回收

#### • 复制算法

- 将内存分为大小相等的两块，每次只用其中一块，一块内存用完之后，将其中不需要回收的对象复制到另一块内存区域，然后将原来的半块内存区域全部回收
- 实现简单，效率高，但是将内存缩小为原来的一半，代价高。
- 大多数对象（98%）存活时间很短，不需要1：1划分两块区域
- 1块较大的Eden和2块较小的Survivor空间，每次使用eden和一块survivor，之后将存活的对象复制到另一块survivor区域里
- survivor空间不够需要老年代进行分配担保

#### • 标记-整理算法

- 标记需要回收对象，将存活对象移动到一端，然后将端边界以外的内存回收

#### • 分代收集算法(当前商业虚拟机采用的垃圾收集算法)

- 将堆分成新生代和老年代
- 新生代每次只有少量对象存活，用复制算法，只需付出复制少量存活对象的成本
- 老年代对象存活率高，用标记-清除或者标记-整理算法（没有额外空间进行分配担保）

## 4.Minor GC(Young GC),Full GC(Major Gc)

#### • minor gc

- 新生代上的gc，当新生代的eden区满时触发

#### • full gc

- 经过minor gc之后存活的一部分对象会进入老年代，如果老年代剩余空间不足的时候会触发full gc

其中永久代如何内存不足也会触发 fullGC

- MinGC
  - 新生代中的垃圾收集动作，采用的是复制算法
  - 对于较大的对象，在Minor GC的时候可以直接进入老年代
- FullGC
  - Full GC是发生在老年代的垃圾收集动作，采用的是标记-清除/整理算法。
  - 由于老年代的对象几乎都是在Survivor区熬过来的，不会那么容易死掉。因此Full GC发生的次数不会有Minor GC那么频繁，并且Time(Full GC) > Time(Minor GC)

### 3.2.1 YGC

答：说白了就是复制算法，对象只会存在于 Eden 区和名为“From”的

Survivor 区，Survivor 区“To”是空的。紧接着进行 GC，Eden 区中所有存活的对象都会被复制到“To”，而在“To”区中，仍存活的对象会根据他们的年龄值来决定去向。年龄达到一定值(年龄阈值，可以通过 -XX:MaxTenuringThreshold 来设置)的对象会被移动到年老代中，没有达到阈值的对象会被复制到“To”区域。经过这次 GC 后，Eden 区和 From 区已经被清空。这个时候，“From”和“To”会交换他们的角色，也就是新的“To”就是上次 GC 前的“From”，新的“From”就是上次 GC 前的“To”。不管怎样，都会保证名为 To 的 Survivor 区域是空的。Minor GC 会一直重复这样的过程，直到“To”区被填满，“To”区被填满之后，会将所有对象移动到年老代中。

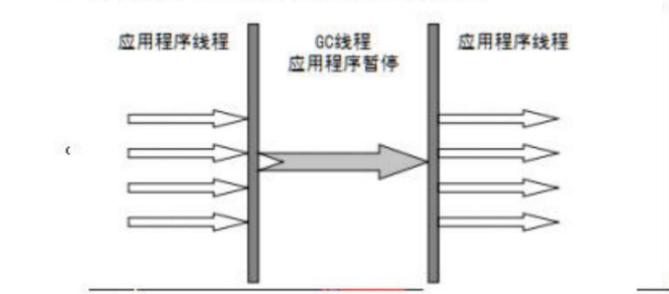
其中如果发生晋升失败的情况，那么说明老年代的内存空间不够用了，需要进行一次 FullGC

### 3.2.2 FGC

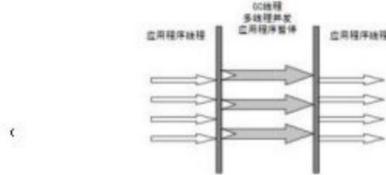
答：FGC 就是标记整理或者是标记清除算法来清除老年代。

## 3.3 垃圾收集器 CMS

- Serial 收集器
  - 是一个单线程的收集器，不是只能使用一个CPU。在进行垃圾收集时，必须暂停其他所有的工作线程，直到收集结束。
  - 新生代采用复制算法，Stop-The-World
  - 老年代采用标记-整理算法，Stop-The-World
  - 简单高效，Client模式下默认的新一代收集器



- ParNew收集器
  - ParNew收集器是Serial收集器的多线程版本
  - 新生代采用复制算法，Stop-The-World
  - 老年代采用标记-整理算法，Stop-The-World
  - 它是运行在Server模式下首选新生代收集器
  - 除了Serial收集器之外，只有它能和CMS收集器配合工作

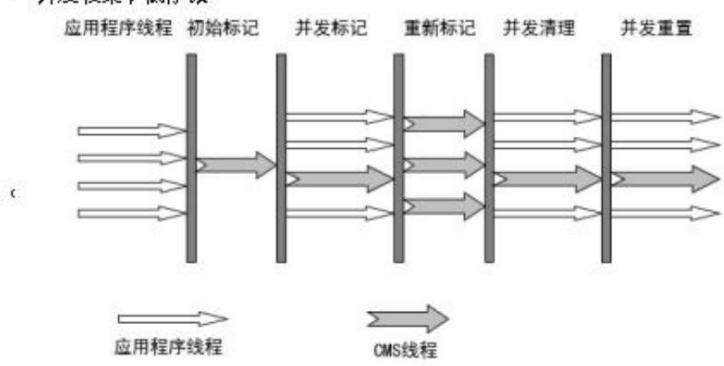


0.834: [GC 0.834: [ParNew: 13184K->1600K(14784K), 0.0092203 secs] 13184K->1921K(63936K), 0.0093401 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

- ParNew Scanvenging收集器
  - 类似ParNew，但更加关注吞吐量。目标是：达到一个可控制吞吐量的收集器。
  - 停顿时间和吞吐量不可能同时调优。我们一方面希望停顿时间少，另外一方面希望吞吐量高，其实这是矛盾的。因为：在GC的时候，垃圾回收的工作总量是不变的，如果将停顿时间减少，那频率就会提高；既然频率提高了，说明就会频繁的进行GC，那吞吐量就会减少，性能就会降低。
- G1收集器
  - 是当今收集器发展的最前言成果之一，对垃圾回收进行了划分优先级的操作，这种有优先级的区域回收方式保证了它的高效率
  - 最大的优点是结合了空间整合，不会产生大量的碎片，也降低了进行gc的频率
  - 让使用者明确指定指定停顿时间

#### • CMS收集器：(Concurrent Mark Sweep：并发标记清除老年代收集器)

- 一种以获得最短回收停顿时间为为目标的收集器，适用于互联网站或者B/S系统的服务器上
- 初始标记(Stop-The-World)：根可以直接关联到的对象
- 并发标记(和用户线程一起)：主要标记过程，标记全部对象
- 重新标记(Stop-The-World)：由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正
- 并发清除(和用户线程一起)：基于标记结果，直接清理对象
- 并发收集，低停顿



## 3.4 java 类加载机制 双亲委派

[https://blog.csdn.net/ns\\_code/article/details/17881581](https://blog.csdn.net/ns_code/article/details/17881581)

### 3.4.1 java 类加载的过程

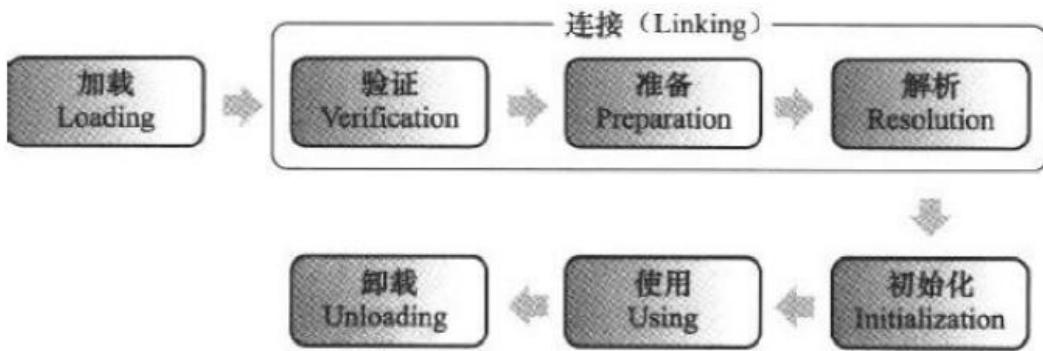


图 7-1 类的生命周期

类加载过程

加载

1. 通过一个类的全限名来获取定义此类的二进制节流。（实现这个代码模块就是类加载器）
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口。

验证

文件格式验证

- [x] 是否以魔数 0xCAFEBAE 开头
- [x] 主次版本号是否在当前虚拟机处理范围之内
- [x] 常量池中的常量是否有不被支持的常量类型
- [x] 指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量
- [x] CONSTANTUtf8info 型的常量中是否有不符合 UTF8 编码的数据
- [x] Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息
- [x] 等等

元数据验证

- [x] 这个类是否有父类
- [x] 这个类的父类是否继承了不准许被继承的类

- [x] 如果这个类不是抽象类,是否实现了其父类或者接口之中要求实现的所有方法

- [x] 类中的字段方法是否与父类产生矛盾

-

字节码验证

- [x] 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作

- [x] 保证跳转指令不会跳转到方法体以外的字节码指令上

- [x] 保证方法体重的类型转换是有效的

符号引用验证

- [x] 符号引用中通过字符串描述的全限定名是否找到相应的类

- [x] 在指定的类中是否存在符合方法的字段描述符以及简单名称说描述的方法和字段

- [x] 符号引用中的类、字段、方法的访问性是否被当前类访问

准备

准备阶段是正式为类变量分配内存并设置类变量初始值(被 static 修饰的变量)的阶段,这些变量所使用的内存都将在方法区中进行分配

解析

解析阶段就是虚拟机将常量池内的符号引用替换为直接引用的过程

初始化

初始化就是执行类构造器方法的过程

### 3.4.2 双亲委派机制

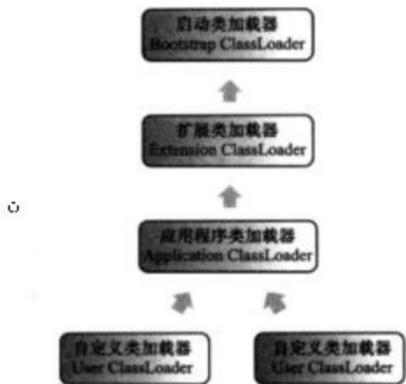


图 7-2 类加载器双亲委派模型

- 双亲委派概念
  - 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的加载器都是如此，因此所有的类加载请求都会传给顶层的启动类加载器，只有当父加载器反馈自己无法完成该加载请求（该加载器的搜索范围内没有找到对应的类）时，子加载器才会尝试自己去加载。
- 加载器
  - 启动 (Bootstrap) 类加载器：是用本地代码实现的类装入器，它负责将 <Java Runtime Home>/lib下面的类库加载到内存中（比如rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
  - 标准扩展 (Extension) 类加载器：是由 Sun 的 ExtClassLoader { sun.misc.Launcher\$ExtClassLoader } 实现的。它负责将 < Java Runtime Home >/lib/ext 或者由系统变量 java.ext.dir 指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。、
  - 系统 (System) 类加载器：由 Sun 的 AppClassLoader { sun.misc.Launcher\$AppClassLoader } 实现的。它负责将系统类路径 (CLASSPATH) 中指定的类库加载到内存中。开发者可以直接使用系统类加载器。除了以上列举的三种类加载器，还有一种比较特殊的类型 — 线程上下文类加载器。
- 如果加载同一个类，该使用哪一个类？
  - 父类的

### 3.4.3 破坏双亲委派模型

1. 在 JDK1.2 之前，用户去继承 java.lang.ClassLoader 的唯一目的就是为了重写 loadClass 方法，由于用户自己重写了 loadClass，那么也就是用户自己去自定义加载类，故事破坏
- 2.JDBC,JDNI 等的 SPI 的加载都是父类的加载器去请求子类的加载器去加载累；
- 3.OSGI 的热部署就是自定义类加载器机制的实现；

## 3.5 内存泄露

### 1. 内存溢出

- 堆上无内存可完成实例分配且堆无法扩展时->OutOfMemoryError
- 方法区（以及内部的常量池）无法满足内存分配需求时->OutOfMemoryError
- 虚拟机栈（本地方法栈）扩展时无法申请到足够的内存->OutOfMemoryError

### 2. 内存泄漏

- 程序动态分配了内存，但在程序结束时没有释放这部分内存，导致那部分内存不可用（代码设计引起的）
- java也会有内存泄漏：当被分配的对象可达但已经没有作用时。比如Student s1 = new Student(); Student s2 = new Student();,然后往ArrayList里放s1,s2,再把s1, s2设成null，但是这两个对象所占内存并没有被释放，因为ArrayList里还存着对象的引用

### 3. 内存溢出，内存泄漏区别？

- 内存泄露是导致内存溢出的原因之一；内存泄露积累起来将导致内存溢出。
- 内存泄露可以通过完善代码来避免；内存溢出可以通过调整配置来减少发生频率，但无法彻底避免。

### 4. 如何检测内存泄露？

- 可以通过一些性能监测分析工具，如 JProfiler、Optimizeit Profiler。

### 5. 如何避免内存泄露、溢出？

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

### 6. 检查内存泄露的工具

用什么工具可以查出内存泄漏

- MemoryAnalyzer：一个功能丰富的JAVA堆转储文件分析工具，可以帮助你发现内存漏洞和减少内存消耗
- EclipseMAT：是一款开源的JAVA内存分析软件，查找内存泄漏，能容易找到大块内存并验证谁在一直占用它，它是基于Eclipse RCP(Rich Client Platform)，可以下载RCP的独立版本或者Eclipse的插件
- JProbe：分析Java的内存泄漏。

## 3.6 .内存泄露的案例分析 jvm 调优

<https://www.cnblogs.com/csniper/p/5592593.html>

[https://mp.weixin.qq.com/s/ydkEkh\\_Uc1paftJLKIsm0w](https://mp.weixin.qq.com/s/ydkEkh_Uc1paftJLKIsm0w)

### 3.6.1 jvm 调优目的

1. 将转移到老年代的对象数量降低到最小；
2. 减少 fullGC 的执行时间

### 3.6.2 案例分析

1. 案例 1（修改 NewRatio 可以说有时候访问访问着就会发生卡顿）

（[http://www.360doc.com/content/13/0305/10/15643\\_269388816.shtml](http://www.360doc.com/content/13/0305/10/15643_269388816.shtml)）

一个服务系统，经常出现卡顿，分析原因，发现 **Full GC** 时间太长：

jstat -gcutil:

| S0    | S1   | E    | O     | P     | YGC | YGCT  | FGC | FGCT  | GCT   |
|-------|------|------|-------|-------|-----|-------|-----|-------|-------|
| 12.16 | 0.00 | 5.18 | 63.78 | 20.32 | 54  | 2.047 | 5   | 6.946 | 8.993 |

分析上面的数据，发现 Young GC 执行了 54 次，耗时 2.047 秒，每次 Young GC 耗时 37ms，在正常范围，而 Full GC 执行了 5 次，耗时 6.946 秒，每次平均 1.389s，数据显示出来的问题是：Full GC 耗时较长，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为 1:9，这就是问题的原因：

1，新生代太小，导致对象提前进入老年代，触发老年代发生 Full GC；

2，老年代较大，进行 Full GC 时耗时较大；

优化的方法是调整 NewRatio 的值，调整到 4，发现 Full GC 没有再发生，只有 Young GC 在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是有用的，但并不是对所有应用都要这么做）

## 2.案例 2（与案例 1 重复）

下面这个例子是针对 **Service S** 的优化，对于最近刚开发出来的 Service S，执行Full GC需要消耗过多的时间。

现在看一下执行 jstat -gcutil 的结果

| S0    | S1   | E    | O     | P     | YGC | YGCT  | FGC | FGCT  | GCT   |
|-------|------|------|-------|-------|-----|-------|-----|-------|-------|
| 12.16 | 0.00 | 5.18 | 63.78 | 20.32 | 54  | 2.047 | 5   | 6.946 | 8.993 |

左边的Perm区的值对于最初的GC优化并不重要，而YGC参数的值更加对于这次优化更为重要。

平均执行一次Minor GC和Full GC消耗的时间如下表所示：

表3：Service S的Minor GC 和Full GC的平均执行时间

| GC类型     | GC执行次数 | GC执行时间 | 平均值    |
|----------|--------|--------|--------|
| Minor GC | 54     | 2.047s | 37ms   |
| Full GC  | 5      | 6.946s | 1.389s |

37ms对于Minor GC来说还不赖，但1.389s对于Full GC来说意味着当GC发生在数据库Timeout设置为1s的系统中时，可能会频繁出现超时现象。

首先，你需要检查开始GC优化前内存的使用情况。使用 jstat -gccapacity 命令可以检查内存用量情况。在笔者的服务器上查看到的结果如下：

```
NGCMN NGCMX NGC SOC SIC EC OGCMN OGCMX OGC OC PGCMN PGCMX PGC PC YGC FGC  
212992.0 212992.0 212992.0 21248.0 21248.0 170496.0 1884160.0 1884160.0 1884160.0 1884160.0 262
```

其中的关键值如下：

- 新生代内存用量：212,992 KB
- 老年代内存用量：1,884,160 KB

因此，除了永久代以外，被分配的内存空间加起来有2GB，并且新生代：老年代=1：9，为了得到比使用 jstat 更细致的结果，还需加上 -verbosegc 参数获取日志，并把三台服务器按照如下方式设置（除此以外没有使用任何其他参数）：

- NewRatio=2
- NewRatio=3
- NewRatio=4

一天后我得到了系统的GC log，幸运的是，在设置完NewRatio后系统没有发生任何Full GC。

**这是为什么呢？**这是因为大部分对象在创建后很快就被回收了，所有这些对象没有被传入老年代，而是在新生代就被销毁回收了。

在这样的情况下，就没有必要去改变其他的参数值了，只要选择一个最合适的 NewRatio 值即可。那么，**如何确定最佳的NewRatio值呢？**为此，我们分析一下每种 NewRatio 值下 Minor GC 的平均响应时间。

在每种参数下 Minor GC 的平均响应时间如下：

- NewRatio=2 : 45ms
- NewRatio=3 : 34ms
- NewRatio=4 : 30ms

我们可以根据GC时间的长短得出NewRatio=4是最佳的参数值（尽管NewRatio=4时新生代空间是最小的）。在设置完GC参数后，服务器没有发生Full GC。

为了说明这个问题，下面是服务执行一段时间后执行 jstat - gcutil 的结果：

```
S0 S1 E O P YGC YGCT FGC FGCT GCT  
8.61 0.00 30.67 24.62 22.38 2424 30.219 0 0.000 30.219
```

你可能会认为是服务器接收的请求少才使得GC发生的频率较低，实际上，虽然Full GC没有执行过，但Minor GC被执行了2424次。

**3.案例 3 (MinorGC 时间过长)** 可以这样说 人物关系提取模块 需要业务上每 30 分钟加载一个 80MB 的数据文件到内存进行数据分析（人名字和政党），这些数据会在内存中形成超过 100w 个 HashMap<String, String> Entry，这段时间会造成 1000ms 左右的 minorGC 的停顿）

观察这个案例，发现平时的 Minor GC 时间很短，原因是新生代的绝大部分对象都是可清除的，在 Minor GC 之后 Eden 和 Survivor 基本上处于完全空闲的状态。而在分析数据文件期间，800MB 的 Eden 空间很快被填满从而引发 GC，但 Minor GC 之后，新生代中绝大部分对象依然是存活的。我们知道 ParNew 收集器使用的是复制算法，这个算法的高效是建立在大部分对象都“朝生夕灭”的特性上的，如果存活对象过多，把这些对象复制到 Survivor 并维持这些对象引用的正确就成为一个沉重的负担，因此导致 GC 暂停时间明显变长。

如果不修改程序，仅从 GC 调优的角度去解决这个问题，可以考虑将 Survivor 空间去掉（加入参数 -XX:SurvivorRatio=65536、-XX:MaxTenuringThreshold=0 或者 -XX:+AlwaysTenure），让新生代中存活的对象在第一次 Minor GC 后立即进入老年带，等到 Major GC 的时候再清理它们。这种措施可以治标，但也有很大副作用，治本的方案需要修改程序，因为这里的问题产生的根本原因是用 HashMap<Long, Long> 结构来存储数据文件空间效率太低。

下面具体分析一下空间效率。在 HashMap<Long, Long> 结构中，只有 Key 和 Value 所存放的两个长整型数据是有效数据，共 16B ( $2 \times 8B$ )。这两个长整型数据包装成 java.lang.Long 对象之后，就分别具有 8B 的 MarkWord、8B 的 Klass 指针，在加 8B 存储数据的 long 值。在这两个 Long 对象组成 Map.Entry 之后，又多了 16B 的对象头，然后一个 8B 的 next 字段和 4B 的 int 型的 hash 字段，为了对齐，还必须添加 4B 的空白填充，最后还有 HashMap 中对这个 Entry 的 8B 的引用，这样增加两个长整型数字，实际耗费的内存为  $(\text{Long}(24B) \times 2) + \text{Entry}(32B) + \text{HashMap Ref } (8B) = 88B$ ，空间效率为  $16B/88B=18\%$ ，实在太低了。

### 3.7 jstat jmap jps jinfo jconsole

#### 3.7.1 jstat

除去日志文件分析外，还可以直接通过 JVM 自带的一些工具直接分析，如 jstat，使用格式为 jstat -gcutil [pid] [interval] [count]，如下面这个日志：

| [junshan@tbskip027085 junshan]\$ sudo /opt/taobao/java/bin/jstat -gcutil 29723 500 100 |      |       |       |       |      |        |     |        |        |  |
|----------------------------------------------------------------------------------------|------|-------|-------|-------|------|--------|-----|--------|--------|--|
| S0                                                                                     | S1   | E     | O     | P     | YGC  | YGCT   | FGC | FGCT   | GCT    |  |
| 0.00                                                                                   | 4.05 | 30.56 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |
| 0.00                                                                                   | 4.05 | 32.79 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |
| 0.00                                                                                   | 4.05 | 35.96 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |
| 0.00                                                                                   | 4.05 | 38.23 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |
| 0.00                                                                                   | 4.05 | 40.45 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |
| 0.00                                                                                   | 4.05 | 42.68 | 39.49 | 45.22 | 5401 | 68.689 | 66  | 22.519 | 91.209 |  |

上面日志中参数含义如下：

- S0 表示 Heap 上的 Survivor space 0 区已使用空间的百分比。
- S1 表示 Heap 上的 Survivor space 1 区已使用空间的百分比。
- E 表示 Heap 上的 Eden space 区已使用空间的百分比。
- O 表示 Heap 上的 Old space 区已使用空间的百分比。
- P 表示 Perm space 区已使用空间的百分比。
- YGC 表示从应用程序启动到采样时发生 Young GC 的次数。
- YGCT 表示从应用程序启动到采样时 Young GC 所用的时间（单位秒）。
- FGC 表示从应用程序启动到采样时发生 Full GC 的次数。
- FGCT 表示从应用程序启动到采样时 Full GC 所用的时间（单位秒）。
- GCT 表示从应用程序启动到采样时用于垃圾回收的总时间（单位秒）。

### 3.7.2 jmap

可通过命令 `jmap -dump:format=b,file=[filename] [pid]` 来记录下堆的内存快照，然后利用第三方工具（如 mat）来分析整个 Heap 的对象关联情况。

## 3.7 JVM 参数设置

### 参数说明

`-Xmx3550m`：设置 JVM 最大堆内存为 3550M。

`-Xms3550m`：设置 JVM 初始堆内存为 3550M。此值可以设置与 `-Xmx` 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

`-Xss128k`：设置每个线程的栈大小。JDK5.0 以后每个线程栈大小为 1M，之前每个线程栈大小为 256K。应当根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。需要注意的是：当这个值被设置的较大（例如 >2MB）时将会在很大程度上降低系统的性能。

`-Xmn2g`：设置年轻代大小为 2G。在整个堆内存大小确定的情况下，增大年轻代将会减小年老代，反之亦然。此值关系到 JVM 垃圾回收，对系统性能影响较大，官方推荐配置为整个堆大小的 3/8。

`-XX:NewSize=1024m`：设置年轻代初始值为 1024M。

`-XX:MaxNewSize=1024m`：设置年轻代最大值为 1024M。

`-XX:PermSize=256m`：设置持久代初始值为 256M。

- XX:MaxPermSize=256m : 设置持久代最大值为 256M。
- XX:NewRatio=4 : 设置年轻代（包括 1 个 Eden 和 2 个 Survivor 区）与年老代的比值。表示年轻代比年老代为 1:4。
- XX:SurvivorRatio=4 : 设置年轻代中 Eden 区与 Survivor 区的比值。表示 2 个 Survivor 区（JVM 堆内存年轻代中默认有 2 个大小相等的 Survivor 区）与 1 个 Eden 区的比值为 2:4，即 1 个 Survivor 区占整个年轻代大小的 1/6。
- XX:MaxTenuringThreshold=7 : 表示一个对象如果在 Survivor 区（救助空间）移动了 7 次还没有被垃圾回收就进入年老代。如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直接进入年老代，对于需要大量常驻内存的应用，这样做可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象在年轻代存活时间，增加对象在年轻代被垃圾回收的概率，减少 Full GC 的频率，这样做可以在某种程度上提高服务稳定性。
- XX:PretenureSizeThreshold 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配。
- XX : MaxTenuringThreshold 每次 minorGC 就增加一次，超过这个值，在 from 中的对象直接进入到老年代

表 3-2 垃圾收集相关的常用参数

| 参 数                    | 描 述                                                                                                              |
|------------------------|------------------------------------------------------------------------------------------------------------------|
| UseSerialGC            | 虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial + Serial Old 的收集器组合进行内存回收                                                 |
| UseParNewGC            | 打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收                                                                       |
| UseConcMarkSweepGC     | 打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用 |
| UseParallelGC          | 虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old (PS MarkSweep) 的收集器组合进行内存回收                       |
| UseParallelOldGC       | 打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收                                                          |
| SurvivorRatio          | 新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden : Survivor=8 : 1                                                    |
| PretenureSizeThreshold | 直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配                                                                         |
| MaxTenuringThreshold   | 晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就增加 1，当超过这个参数值时就进入老年代                                                       |
| UseAdaptiveSizePolicy  | 动态调整 Java 堆中各个区域的大小以及进入老年代的年龄                                                                                    |
| HandlePromotionFailure | 是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况                                                   |
| ParallelGCThreads      | 设置并行 GC 时进行内存回收的线程数<br>激活 Win<br>禁用 Linux                                                                        |

| 参 数                            | 描 述                                                               |
|--------------------------------|-------------------------------------------------------------------|
| GCTimeRatio                    | GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效 |
| MaxGCPauseMillis               | 设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效                       |
| CMSInitiatingOccupancyFraction | 设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集。默认值为 68%，仅在使用 CMS 收集器时生效             |
| UseCMSCompactAtFullCollection  | 设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效                   |
| CMSFullGCsBeforeCompaction     | 设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效                  |

### 3.8 内存分配与回收策略

1. 对象优先在 Eden 分配
2. 大对象直接进入老年代

虚拟机提供了一个 -XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制（复习一下：新生代采用复制算法收集内存）。

3. 长期存活的对象将进入老年代

对象应放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并且对象年龄设为 1。对象在 Survivor 区中每“熬过”一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就将被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数 -XX:MaxTenuringThreshold 设置。

4. 动态对象年龄判定

如果在 Survivor 空间中相同年龄所有对象大小总和大于 Survivor 空间的一半，（比如说 Survivor 空间大小为 1M，而有两个年龄为 1 的对象大小和是大于 512K 的），那么年龄大于等于该年龄的对象都可以直接进入到老年代。

5. 空间分配担保

在进行 MinorGC 前，虚拟机会查看 HandlePromotionFailure 设置值是否为 True，那么说明允许担保失败（会检查虚拟机老年代剩余空间的大小与平均晋升到老年代空间的大小，如果大于说明“可能”是安全的），为 True 那么进行一次 MinorGC，如果此时刻发现进入到老年代的新对象的大小是大于老年代的剩余空间，说明担保失败了，只能进行一次 FullGC 清除老年代的剩余空间。

### 3.9 面试问题

#### 3.9.1 一般 Java 堆是如何实现的？

我：在 HotSpot 虚拟机实现中，Java 堆分成了新生代和老年代，我当时看的是 1.7 的实现，所有还有永久代，新生代中又分为了 eden 区和 survivor 区，survivor 区又分成了 S0 和 S1，或则是 from 和 to，（这个时候，我要求纸和笔，因为我觉得这个话题可以聊蛮长时间，又是我比较熟悉的...一边画图，一边描述），其中 eden，from 和 to 的内存大小默认是 8:1:1（各种细节都要说出来...），此时，我已经在纸上画出了新生代和老年代代表的区域

## 3.9.2 对象在内存中的初始化过程

参考：1.<https://blog.csdn.net/WantFlyDaCheng/article/details/81808064>  
2.《深入理解 java 虚拟机》

原文：<https://blog.csdn.net/WantFlyDaCheng/article/details/81808244>

`Student s = new Student()` 为例

1.首先查看类的符号引用，看是否已经在常量池中，在说明已经加载过了，不在的话需要进行类的加载，验证，准备，解析，初始化的过程。

2.上诉过程执行完毕以后，又将 `Student` 加载进内存，也就是存储 `Student.class` 的字段信息和方法信息，存储到方法区中

字段信息：存放类中声明的每一个字段的信息，包括字段的名、类型、修饰符。

方法信息：类中声明的每一个方法的信息，包括方法名、返回值类型、参数类型、修饰符、异常、方法的字节码。

3。然后在自己的线程私有的虚拟机栈中，存储该引用，然后在每个线程的私有空间里面去分配空间存储 `new Student()`,如果空间不足在 `eden` 区域进行分配空间

4，对类中的成员变量进行默认初始化

5，对类中的成员变量进行显示初始化

6，有构造代码块就先执行构造代码块，如果没有，则省略(此步上文未体现)

7，执行构造方法，通过构造方法对对象数据进行初始化

8，堆内存中的数据初始化完毕，把内存值复制给 `s` 变量

## 3.9.3 对象的强、软、弱和虚引用

### (1) 强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。

### (2) 软引用（SoftReference）

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

### (3) 弱引用（WeakReference）

弱引用与软引用的区别在于：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

### (4) 虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，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。

## 3.9.4 如何减少 GC 的次数

## 1. 对象不用时最好显示置为 **NULL**

一般而言，为 **NULL** 的对象都会被作为垃圾处理，所以将不用的对象置为 **NULL**，有利于 **GC** 收集器判定垃圾，从而提高了 **GC** 的效率。

## 2. 尽量少使用 **System.gc()**

此函数建议 **JVM** 进行主 **GC**，会增加主 **GC** 的频率，增加了间接性停顿的次数。

## 3. 尽量少使用静态变量

静态变量属于全局变量，不会被 **GC** 回收，他们会一直占用内存

## 4. 尽量使用 **StringBuffer**,而不使用 **String** 来累加字符串

## 5. 分散对象创建或删除的时间

集中在短时间内大量创建新对象，特别是大对象，会导致突然需要大量内存，**JVM** 在这种

情况下只能进行主 **GC** 以回收内存，从而增加主 **GC** 的频率。

## 6. 尽量少用 **finaliza** 函数

它会加大 **GC** 的工作量。

## 7. 如果有需要使用经常用到的图片，可以使用软引用类型，将图片保存在内存中，而不引起 **outofmemory**

## 8. 能用基本类型入 **INT** 就不用对象 **Integer**

## 9. 增大 **-Xmx** 的值

### 3.9.5 新生代 老年代 永久代

#### 年轻代：

事实上，在上一节，已经介绍了新生代的主要垃圾回收方法，在新生代中，使用“停止-复制”算法进行清理，将新生代内存分为 2 部分，1 部分 **Eden** 区较大，1 部分 **Survivor** 较小，并被划分为两个等量的部分。每次进行清理时，将 **Eden** 区和一个 **Survivor** 中仍然存活的对象拷贝到 另一个 **Survivor** 中，然后清理掉 **Eden** 和刚才的 **Survivor**。

这里也可以发现，停止复制算法中，用来复制的两部分并不总是相等的（传统的停止复制算法两部分内存相等，但新生代中使用 1 个大的 Eden 区和 2 个小的 Survivor 区来避免这个问题）

由于绝大部分的对象都是短命的，甚至存活不到 Survivor 中，所以，Eden 区与 Survivor 的比例较大，HotSpot 默认是 8:1，即分别占新生代的 80%，10%，10%。如果一次回收中，Survivor+Eden 中存活下来的内存超过了 10%，则需要将一部分对象分配到 老年代。用-XX:SurvivorRatio 参数来配置 Eden 区域 Survivor 区的容量比值，默认是 8，代表 Eden: Survivor1: Survivor2=8:1:1。

#### 老年代：

老年代存储的对象比年轻代多得多，而且不乏大对象，对老年代进行内存清理时，如果使用停止-复制算法，则相当低效。一般，老年代用的算法是标记-整理算法，即：标记出仍然存活的对象（存在引用的），将所有存活的对象向一端移动，以保证内存的连续。

在发生 Minor GC 时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次 Full GC，否则，就查看是否设置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行 MinorGC，此时可以容忍内存分配失败；如果不允许，则仍然进行 Full GC（这代表着如果设置-XX:+Handle PromotionFailure，则触发 MinorGC 就会同时触发 Full GC，哪怕老年代还有很多内存，所以，最好不要这样做）。

#### 方法区（永久代）：

永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证 3 点：

1. 类的所有实例都已经被回收
2. 加载类的 ClassLoader 已经被回收
3. 类对象的 Class 对象没有被引用（即没有通过反射引用该类的地方）

永久代的回收并不是必须的，可以通过参数来设置是否对类进行回收。HotSpot 提供-Xnoclassgc 进行控制

使用-verbose、-XX:+TraceClassLoading、-XX:+TraceClassUnLoading 可以查看类加载和卸载信息  
-verbose、-XX:+TraceClassLoading 可以在 Product 版 HotSpot 中使用；  
-XX:+TraceClassUnLoading 需要 fastdebug 版 HotSpot 支持

## 如何加快 gc 的速度 快速判断对象生死

垃圾回收机制简述，堆，栈，如何判断对象已死，有环 root 链如何找到了？

### 线程安全 java 里面的实现方式

如果我们一个项目，理论上需要 1.5G 的内存就足够，但是项目上线后发现隔了几个星期，占用内存到了 2.5G，这时候你会考虑是什么问题？怎么解决？

可能造成内存泄漏的原因有哪些？检查内存泄漏的工具有哪些？你平时是怎么检查内存泄漏的？

jvm 多态原理。invokestatic invokeinterface 等指令。常量池中的符号引用 找到直接引用。在堆中找到实例对象，获取到偏移量，由偏移量在方法表中指出调用的具体方法。接口是在方法表中进行扫描）等等扯了半天

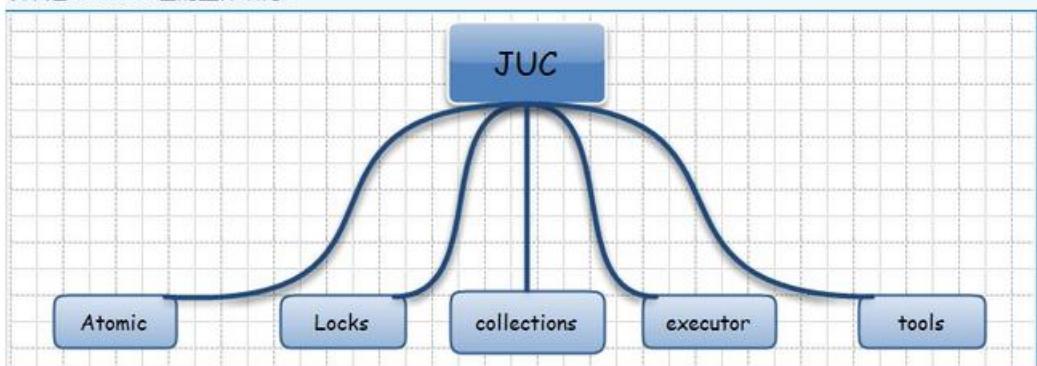
## 七 . juc 包

[https://blog.csdn.net/china\\_wanglong/article/details/38828407](https://blog.csdn.net/china_wanglong/article/details/38828407)

### 7.0 juc 概况

#### 1. JUC概况

以下是Java JUC包的主体结构：



- Atomic : AtomicInteger
- Locks : Lock, Condition, ReadWriteLock
- Collections : Queue, ConcurrentMap
- Executor : Future, Callable, Executor
- Tools : CountDownLatch, CyclicBarrier, Semaphore

### 7.1 Tools

#### 7.1.1 CountDownLatch

这个类是一个同步计数器，主要用于线程间的控制，当 CountDownLatch 的 count 计数>0 时，await()会造成阻塞，直到 count 变为 0，await()结束阻塞，使用 countDown()会让 count 减 1。CountDownLatch 的构造函数可以设置 count 值，当 count=1 时，它的作用类似于 wait()和 notify()的作用。如果我想让其他线程执行完指定程序，其他所有程序都执行结束后我再执行，这时可以用 CountDownLatch，但计数无法被重置，如果需要重置计数，请考虑使用 CyclicBarrier 。

#### 7.1.2 CyclicBarrier

该类从字面理解为循环屏障，它可以协同多个线程，让多个线程在这个屏障前等到，直到所有线程都到达了这个屏障时，再一起执行后面的操作。假如每个线程各有一个 await，任何一个线程运行到 await 方法时就阻塞，直到最后一个线程运行到 await 时才同时返回。和之前的 CountDownLatch 相比，它只有 await 方法，而 CountDownLatch 是使用 countDown()方法将计数器减到 0，它创建的

参数就是 countDown 的数量； CyclicBarrier 创建时的 int 参数是 await 的数量。

### 7.1.3 Semaphore

该类用于控制信号量的个数，构造时传入个数。总数就是控制并发的数量。假如是 5，程序执行前用 acquire()方法获得信号，则可用信号变为 4，程序执行完通过 release()方法归还信号量，可用信号又变为 5.如果可用信号为 0, acquire 就会造成阻塞，等待 release 释放信号。acquire 和 release 方法可以不在同一个线程使用。Semaphore 实现的功能就类似厕所所有 5 个坑，假如有 10 个人要上厕所，那么同时只能有多少个人去上厕所呢？同时只能有 5 个人能够占用，当 5 个人中的任何一个人让开后，其中等待的另外 5 个人中又有一个人可以占用了。另外等待的 5 个人中可以是随机获得优先机会，也可以是按照先来后到的顺序获得机会，这取决于构造 Semaphore 对象时传入的参数选项。单个信号量的 Semaphore 对象可以实现互斥锁的功能，并且可以是由一个线程获得了“锁”，再由另一个线程释放“锁”，这可应用于死锁恢复的一些场合。

### 7.1.4 Exchanger

这个类用于交换数据，只能用于两个线程。当一个线程运行到 exchange()方法时会阻塞，另一个线程运行到 exchange()时，二者交换数据，然后执行后面的程序。

## 7.2 List Set

CopyOnWriteArrayList, CopyOnWriteArraySet 和 ConcurrentSkipListSet

## 7.3 Map

ConcurrentHashMap 和 ConcurrentSkipListMap

## 7.4 Queue

ArrayBlockingQueue, LinkedBlockingQueue, LinkedBlockingDeque, ConcurrentLinkedQueue 和 ConcurrentLinkedDeque

### 7.4.1 ArrayBlockingQueue

1.基于数组实现，保证并发的安全性是基于 ReentrantLock 和 Condition 实现的。其中有两个重要的成员变量 putindex 和 takeindex,这两个需要搞懂, putindex 就是指向数组中上一个添加完元素的位置的下一个地方，比如刚在 index=1 的位置添加完，那么 putindex 就是 2，其中有一点特别注意的就是当 index=数组的长度减一的时候，意味着数组已经到了满了，那么需要将 putindex 置位 0，原因是数组在被消费的也就是取出操作的时候，是从数组的开始位置取得，所以最开始的位置容易是空的，所以要把要添加的位置置位 0; takeindex 也是一样的，当 takeindex 到了数组的长度减一的时候，也需要将 takeindex 置为 0.

#### 2.add offer put

add 调用了 offer 方法 add 方法数组满了则抛出异常

offer 方法：用 ReentrantLock 加锁，首先判断数组是否满了，数组满了则返回 false，数组不满的话直接入队，也就是将 putindex 索引处的值置为新要加入的数，如果加入以后发现 putindex++ = 数组的长度，那么说明后面的全部已经填满了，因此 putindex 置为 0，因为前面的可能出队的过程空出来了，所以变为 0，最后一步就是执行 notEmpty.signal 去唤醒消费的执行了 take 的线程，只可能是执行了 take 的方法的线程，因为执行了其它方法 remove,poll 不会产生线程的挂起操作。

put:首先是 ReentrantLock 加锁，然后判断是否满了，队列满了，则执行 notFull.await()操作挂起，等

待 `notFull.signal()` 唤醒。没满，则直接进行入队，入队和 `offer` 操作一样，也就是将 `putindex` 索引处的值置为新要加入的数，如果加入以后发现 `putindex++ = 数组的长度`，那么说明后面的全部已经填满了，因此 `putindex` 置为 0，因为前面的可能出队的过程空出来了，所以变为 0，最后一步就是执行 `notEmpty()` 去唤醒消费的执行了 `take` 的线程

### 3.remove,poll,take

`poll` 首先加锁 `ReetranLock`，然后判断队列是否为空，不为空，则将 `putindex` 出的值用副本 `copy`，然后置位 `null`，然后去执行唤醒 `notFull()` 操作，也就是唤醒调用了 `put` 操作的线程，唤醒操作并不一定总是发生。

`take` 操作，先加锁，然后如果队列空则 `notEmpty.await()` 方法，不为空，则执行和 `poll` 一样的出队操作：则将 `putindex` 出的值用副本 `copy`，然后置位 `null`，然后去执行唤醒 `notFull()` 操作

## 7.4.2 LinkedBlockingQueue

1. 基于链表实现，有 `takelock` 和 `putlock`，也就是说可以同时在首尾两端进行操作，因此吞吐量比 `ArrayBlockingQueue` 大，同时由于首尾两端都可以进行操作，所以当在进行添加的操作的过程可以一直去添加，直到没有被阻塞的添加线程为止，然后才去执行消费的线程。

### 1.add,offer,put

`add` 调用 `offer`，满了抛出异常

`offer` 方法 `putlock` 锁，然后不满则加入，同时获取一个 `c` 值，`c` 值代表本次队列增加前的队列的数目（一开始长度 2，增加 1，现在长度是 3，那么 `c` 就是 2），然后判断如果不满则继续去唤醒 `notFull.signl`，去唤醒添加线程去添加（添加过程是直接 `last` 节点指向下一个，简单的节点后增加一个节点，然后 `last` 指向最后一个节点），上述过程结束，然后去判断 `c==0`（`c` 代表了之前的队列长度，如果添加之前队列长度是 0 那么说明可能有挂起的消费线程，需要从队列取元素，但队列长度为 0 没有元素；判断 `c>0` 没有意义，因为添加之前队列不为空，说明不存在挂起的消费线程，挂起的原因是因为队列为空，所以不存在因此源码是判断 `c==0`），`c` 如果等于 0 那么去唤醒阻塞的 `notEmpty` 上的条件等待线程。`put` 操作就是满了则挂起，不满则执行，同时添加完一个后，发现没满继续去唤醒挂起的添加线程

### 2.poll take

反之，一样的逻辑

`poll` 则获取 `takelock`，然后不为空则出队一个元素，也就是链表的删除头结点操作，通过是如果队列不为空，那么继续去唤醒被挂起的消费线程（消费线程就是执行了队列的 `take` 操作的线程），直到没有消费线程或者队列为空，结束，然后如果 `c`（也是队列消费一个头节点的元素后，没消费之前的长度，没发生删除的时候队列的长度），如果 `c` 的长度已经是队列的长度，则去唤醒被挂起的执行了 `put` 方法的线程，然后释放 `takelock` 锁

`take` 方法一样的道理，为空则挂起，不为空一直消费，唤起消费线程一直消费，直到条件不满足，那么去尝试判断 `c` 的值，`c` 是队列长度减一，那么去唤醒执行了 `put` 方法的被挂起的线程。

以下内容来自 [深入剖析 java 并发之阻塞队列 LinkedBlockingQueue 与 ArrayBlockingQueue](#)

## 7.4.3 LinkedBlockingQueue 和 ArrayBlockingQueue 迥异

通过上述的分析，对于 `LinkedBlockingQueue` 和 `ArrayBlockingQueue` 的基本使用以及内部实现原理我们已较为熟悉了，这里我们就对它们两间的区别来个小结

1. 队列大小有所不同，`ArrayBlockingQueue` 是有界的初始化必须指定大小，而 `LinkedBlockingQueue` 可以是有界的也可以是无界的(`Integer.MAX_VALUE`)，对于后者而言，当添加速度大于移除速度时，在无界的情况下，可能会造成内存溢出等问题。

2. 数据存储容器不同，`ArrayBlockingQueue` 采用的是数组作为数据存储容器，而 `LinkedBlockingQueue` 采用的则是以 `Node` 节点作为连接对象的链表。

3. 由于 `ArrayBlockingQueue` 采用的是数组的存储容器，因此在插入或删除元素时不会产生或销毁任何额外的对象实例，而 `LinkedBlockingQueue` 则会生成一个额外的 `Node` 对象。这可能在长时间内需要高效并发地处理大批量数据时，对于 GC 可能存在较大影响。

4. 两者的实现队列添加或移除的锁不一样，`ArrayBlockingQueue` 实现的队列中的锁是没有分离的，即添加操作和移除操作采用的同一个 `ReentrantLock` 锁，而 `LinkedBlockingQueue` 实现的队列中的锁是分离的，其添加采用的是 `putLock`，移除采用的是 `takeLock`，这样能大大提高队列的吞吐量，也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。

## 7.5 线程池

# 1. 线程池工作原理

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于 `corePoolSize`；如果当前线程数为 `corePoolSize`，继续提交的任务被保存到阻塞队列中，等待被执行；如果阻塞队列满了，那就创建新的线程执行当前任务；直到线程池中的线程数达到 `maxPoolSize`，这时再有任务来，只能执行 `reject()` 处理该任务；

# 2. 线程池分类

4 种类型的线程池：

## `newFixedThreadPool()`

说明：初始化一个指定线程数的线程池，其中 `corePoolSize == maxPoolSize`，使用 `LinkedBlockingQueue` 作为阻塞队列

特点：即使当线程池没有可执行任务时，也不会释放线程。

## `newCachedThreadPool()`

说明：初始化一个可以缓存线程的线程池，默认缓存 60s，线程池的线程数可达到 `Integer.MAX_VALUE`，即 2147483647，内部使用 `SynchronousQueue` 作为阻塞队列；

特点：在没有任务执行时，当线程的空闲时间超过 `keepAliveTime`，会自动释放线程资源；当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销；

因此，使用时要注意控制并发的任务数，防止因创建大量的线程导致而降低性能。

## `newSingleThreadExecutor()`

说明：初始化只有一个线程的线程池，内部使用 `LinkedBlockingQueue` 作为阻塞队列。

特点：如果该线程异常结束，会重新创建一个新的线程继续执行任务，唯一的线程可以保证所提交任务的顺序执行

## `newScheduledThreadPool()`

特定：初始化的线程池可以在指定的时间内周期性的执行所提交的任务，在实际的业务场景中可以使用该线程池定期的同步数据。

总结：除了 newScheduledThreadPool 的内部实现特殊一点之外，其它线程池内部都是基于 ThreadPoolExecutor 类（Executor 的子类）实现的。

### 3. 线程池底层实现类 ThreadPoolExecutor 类

**ThreadPoolExecutor (corePoolSize,maxPoolSize,keepAliveTime,timeUnit,workQueue,threadFactory,handle);**

#### **corePoolSize**

线程池中的核心线程数，当提交一个任务时，线程池创建一个新线程执行任务，直到当前线程数等于 corePoolSize；如果当前线程数为 corePoolSize，继续提交的任务被保存到阻塞队列中，等待被执行；如果执行了线程池的 prestartAllCoreThreads()方法，线程池会提前创建并启动所有核心线程。

#### **maximumPoolSize**

线程池中允许的最大线程数。如果当前阻塞队列满了，且继续提交任务，则创建新的线程执行任务，前提是当前线程数小于 maximumPoolSize；

#### **keepAliveTime**

线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间；默认情况下，该参数只在线程数大于 corePoolSize 时才有用；

#### **unit**

keepAliveTime 的单位；

#### **workQueue**

用来保存等待被执行的任务的阻塞队列，且任务必须实现 Runnable 接口，在 JDK 中提供了如下阻塞队列：

- 1、ArrayBlockingQueue：基于数组结构的有界阻塞队列，按 FIFO 排序任务；
- 2、LinkedBlockingQuene：基于链表结构的阻塞队列，按 FIFO 排序任务，吞吐量通常要高于 Array BlockingQuene；
- 3、SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQuene；
- 4、priorityBlockingQuene：具有优先级的无界阻塞队列；

#### **threadFactory**

创建线程的工厂，通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名。

#### **handler**

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了 4 种策略：

- 1、AbortPolicy：直接抛出异常，默认策略；
- 2、CallerRunsPolicy：用调用者所在的线程来执行任务；
- 3、DiscardOldestPolicy：丢弃阻塞队列中靠最前的任务，并执行当前任务；
- 4、DiscardPolicy：直接丢弃任务；

当然也可以根据应用场景实现 **RejectedExecutionHandler** 接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。

### 4. 线程池状态

RUNNING 自然是运行状态，指可以接受任务执行队列里的任务

SHUTDOWN 指调用了 `shutdown()` 方法，不再接受新任务了，但是队列里的任务得执行完毕。

STOP 指调用了 `shutdownNow()` 方法，不再接受新任务，同时抛弃阻塞队列里的所有任务并中断所有正在执行任务。

TIDYING 所有任务都执行完毕，在调用 `shutdown()/shutdownNow()` 中都会尝试更新为这个状态。

TERMINATED 终止状态，当执行 `terminated()` 后会更新为这个状态

## 四.设计模式

### 4.0 什么是设计模式

在软件工程中，设计模式（design pattern）是对软件设计中普遍存在（反复出现）的各种问题，所提出的解决方案。

#### 4.1.常见的设计模式及其 JDK 中案例：

##### 4.1.1 适配器模式

**适配器模式（Adapter）**，将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。[DP]

```
java.util.Arrays.asList()  
java.io.InputStreamReader(InputStream)  
java.io.OutputStreamWriter(OutputStream)
```

##### 4.1.2 迭代器模式

**迭代器模式（Iterator）**，提供一种方法顺序访问一个聚合对象中各个元素，而又不暴露该对象的内部表示。[DP]

提供一个一致的方法来顺序访问集合中的对象，这个方法与底层的集合的具体实现无关。

JDK 中：

```
java.util.Iterator  
java.utilEnumeration
```

##### 4.1.3 代理模式

**代理（proxy）模式**：指目标对象给定代理对象，并由代理对象代替真实对象控制客户端对真实对象的访问。

代理模式模式有以下角色：

**抽象主题（subject）角色**：声明真实主题和代理主题的共同接口。

**真实主题（real subject）角色**：定义代理对象需要代理的真实对象。

**代理主题（proxy subject）角色**：代替真实对象来控制对真实对象的访问，代

理对象持有真实对象的应用，从而可以随时控制客户端对真实对象的访问。

实例：大话设计模式：

### 代理类如下

```
class Proxy : GiveGift
{
    Pursuit gg;
    public Proxy(SchoolGirl mm)
    {
        gg = new Pursuit(mm);
    }
    public void GiveDolls()
    {
        gg.GiveDolls();
```

让“代理”也去实现  
“送礼物”接口

```

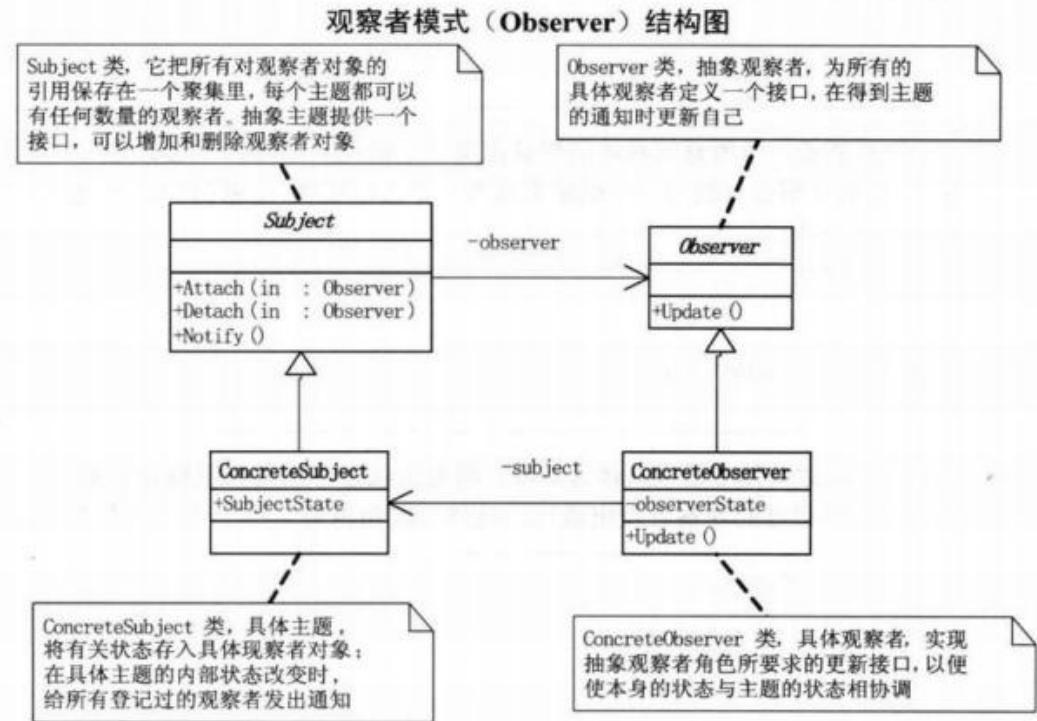
    }
    public void GiveFlowers()
    {
        gg.GiveFlowers();
    }
    public void GiveChocolate()
    {
        gg.GiveChocolate();
    }
}
```

在实现方法中去调用“追求者”类的相关方法

JDK: java.lang.reflect.Proxy      RMI

#### 4.1.4 观察者模式

观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。[DP]



Jdk : `java.util.EventListener`

`javax.servlet.http.HttpSessionBindingListener`

`javax.servlet.http.HttpSessionAttributeListener`

`javax.faces.event.PhaseListener`

#### 4.1.5 装饰器模式

动态的给一个对象附加额外的功能，这也是子类的一种替代方式。可以看到，在创建一个类型的时候，同时也传入同一类型的对象。这在 JDK 里随处可见，你会发现它无处不在，所以下面这个列表只是一小部分。

装饰原有对象、在不改变原有对象的情况下扩展增强新功能/新特征。当不能采用继承的方式对系统进行扩展或者采用继承不利于系统扩展和维护时可以使用装饰模式。

Jdk : 装饰者模式通过包含一个原有的 `InputStream` 对象，并且将 `InputStream` 原有的方法或直接暴露，或进行装饰后暴露，又或者添加了新的特性，如 `DataInputStream` 中的 `readInt()`，`BufferedInputStream` 中的缓存功能。（这些新的功能就是装饰后新添加的）

`java.io.BufferedInputStream(InputStream)`

`java.io.DataInputStream(InputStream)`

`java.io.BufferedOutputStream(OutputStream)`

`java.util.zip.ZipOutputStream(OutputStream)`

```
java.util.Collections#checkedList|Map|Set|SortedSet|SortedMap
```

#### 4.1.6 工厂模式

分为简单工厂模式、工厂方法模式、抽象工厂模式

简单工厂模式：由工厂类决定实例化哪个产品类

工厂方法模式：定义一个用于创建对象的接口，让子类决定创建那个类。**Factory Method** 使一个类的实例化延迟到其子类中

抽象工厂模式：定义一个抽象的工厂，里面定义多个产品的类

简单工厂：把对象的创建放到一个工厂类中，通过参数来创建不同的对象。

工厂方法：每种产品由一种工厂来创建。（不这样会有什么问题？）

抽象工厂：感觉只是工厂方法的复杂化，产品系列复杂化的工厂方法。

**工厂方法模式：**就是一个返回具体对象的方法。

```
java.lang.Proxy#newProxyInstance()  
java.lang.Object#toString()  
java.lang.Class#newInstance()  
java.lang.reflect.Array#newInstance()  
java.lang.reflect.Constructor#newInstance()  
java.lang.Boolean#valueOf(String)  
java.lang.Class#forName()
```

#### 抽象工厂模式

抽象工厂模式提供了一个协议来生成一系列的相关或者独立的对象，而不用指定具体对象的类型。它使得应用程序能够和使用的框架的具体实现进行解耦。这在 JDK 或者许多框架比如 Spring 中都随处可见。它们也很容易识别，一个创建新对象的方法，返回的却是接口或者抽象类的，就是抽象工厂模式了。

```
java.util.Calendar#getInstance()  
java.util.Arrays#asList()  
java.util.ResourceBundle#getBundle()  
java.sql.DriverManager#getConnection()  
java.sql.Connection#createStatement()  
java.sql.Statement#executeQuery()  
java.text.NumberFormat#getInstance()  
javax.xml.transform.TransformerFactory#newInstance()
```

#### 4.1.7 建造者模式

定义了一个新的类来构建另一个类的实例，以简化复杂对象的创建。建造模式通常也使用方法链接来实现。

```
java.lang.StringBuilder#append()  
java.lang.StringBuffer#append()  
java.sql.PreparedStatement  
javax.swing.GroupLayout.Group#addComponent()
```

例如 `StringBuilder` 就是定义了一个新类 `StringBuilder` 来完成 “aa” + “bb”的创建

```
System.Text.StringBuilder sb = new StringBuilder();
```

```
sb.Append("aa");//添加的子对象部分(这就是创建 子对象的部分)
```

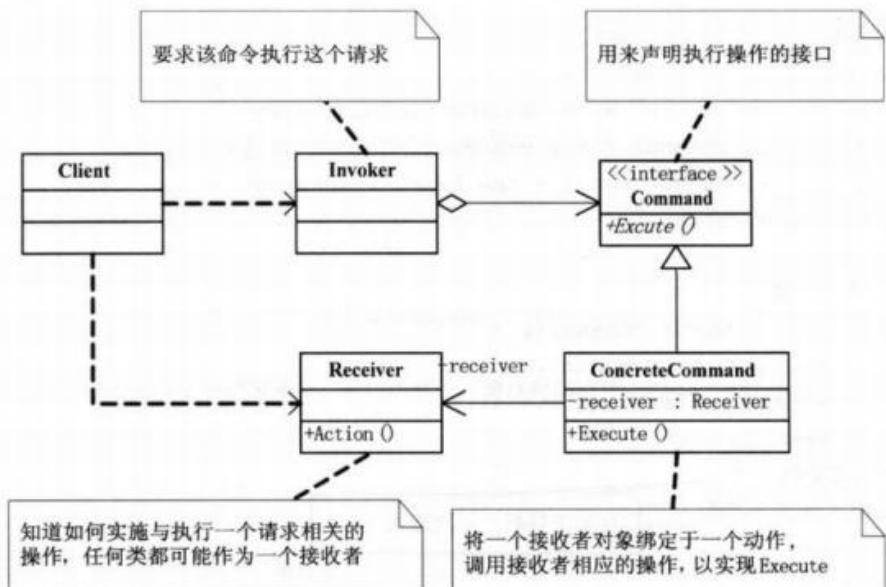
```
sb.Append("bb");(这个就对应 GetResult())
```

```
string str= sb.ToString();//最终 都演变成 最后一种形式
```

#### 4.1.8 命令模式

**命令模式 (Command)**, 将一个请求封装为一个对象, 从而使你可用不同的请求对客户进行参数化; 对请求排队或记录请求日志, 以及支持可撤销的操作。[DP]

命令模式 (Command) 结构图



#### 4.1.9 责任链模式

通过把请求从一个对象传递到链条中下一个对象的方式, 直到请求被处理完毕, 以实现对象间的解耦。

```
java.util.logging.Logger#log()  
javax.servlet.Filter#doFilter()
```

#### 4.1.10 享元模式

**享元模式（Flyweight）**，运用共享技术有效地支持大量细粒度的对象。[DP]

Flyweight 享元模式：

使用缓存来加速大量小对象的访问时间。

```
java.lang.Integer#valueOf(int)  
java.lang.Boolean#valueOf(boolean)  
java.lang.Byte#valueOf(byte)  
java.lang.Character#valueOf(char)
```

#### 4.1.11 中介者模式

中介者模式

通过使用一个中间对象来进行消息分发以及减少类之间的直接依赖。

```
java.util.Timer  
java.util.concurrent.Executor#execute()  
java.util.concurrent.ExecutorService#submit()  
java.lang.reflect.Method#invoke()
```

#### 4.1.12 备忘录模式

生成对象状态的一个快照，以便对象可以恢复原始状态而不用暴露自身的内容。Date 对象通过自身内部的一个 long 值来实现备忘录模式。

```
java.util.Date  
java.io.Serializable
```

#### 4.1.13 组合模式

Composite 组合模式：

又叫做部分-整体模式，使得客户端看来单个对象和对象的组合是同等的。换句话说，某个类型的方法同时也接受自身类型作为参数。

```
avax.swing.JComponent#add(Component)  
java.util.Map#putAll(Map)  
java.util.List#addAll(Collection)  
java.util.Set#addAll(Collection)
```

#### 4.1.14 模板方法模式

模板方法模式

让子类可以重写方法的一部分，而不是整个重写，你可以控制子类需要重写那些操作。

```
java.util.Collections#sort()  
java.io.InputStream#skip()  
java.io.InputStream#read()  
java.util.AbstractList#indexOf()
```

#### 4.1.15 单例模式

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

<https://www.cnblogs.com/cielosun/p/6582333.html>

<https://blog.csdn.net/u014590757/article/details/79818702>

##### 1. 非线程安全懒汉模式

```
public class SingletonDemo {  
    private static SingletonDemo instance;  
    private SingletonDemo(){  
  
    }  
    public static SingletonDemo getInstance(){  
        if(instance==null){  
            instance=new SingletonDemo();  
        }  
        return instance;  
    }  
}
```

##### 2、线程安全懒汉模式

```
public class SingletonDemo {  
    private static SingletonDemo instance;  
    private SingletonDemo(){  
  
    }  
    public static synchronized SingletonDemo getInstance(){  
        if(instance==null){  
            instance=new SingletonDemo();  
        }  
        return instance;  
    }  
}
```

##### 3. 饿汉模式

直接在运行这个类的时候进行一次 **loading**，之后直接访问。显然，这种方法没有起到 **lazy loading** 的效果，考虑到前面提到的和静态类的对比，这种方法只比静态类多了一个内存常驻而已。

```
public class SingletonDemo {  
    private static SingletonDemo instance=new SingletonDemo();  
    private SingletonDemo(){  
}
```

```

    }
    public static SingletonDemo getInstance() {
        return instance;
    }
}

```

#### 4. 静态类内部加载

使用内部类的好处是，静态内部类不会在单例加载时就加载，而是在调用 `getInstance()` 方法时才进行加载，达到了类似懒汉模式的效果，而这种方法又是线程安全的。

```

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

```

#### 5. 双重锁校验模式

```

public class SingletonDemo {
    private volatile static SingletonDemo instance;
    private SingletonDemo(){
        System.out.println("Singleton has loaded");
    }
    public static SingletonDemo getInstance(){
        if(instance==null){
            synchronized (SingletonDemo.class){
                if(instance==null){
                    instance=new SingletonDemo();
                }
            }
        }
        return instance;
    }
}

```

接下来我解释一下在并发时，双重校验锁法会有怎样的情景：

**STEP 1.** 线程 A 访问 `getInstance()` 方法，因为单例还没有实例化，所以进入了锁定块。

**STEP 2.** 线程 B 访问 `getInstance()` 方法，因为单例还没有实例化，得以访问接下来代码块，而接下来代码块已经被线程 1 锁定。

**STEP 3.** 线程 A 进入下一判断，因为单例还没有实例化，所以进行单例实例化，成功实例化后退出代码块，解除锁定。

**STEP 4.** 线程 B 进入接下来代码块，锁定线程，进入下一判断，因为已经实例化，退出代码块，解除锁定。

**STEP 5.** 线程 A 初始化并获取到了单例实例并返回，线程 B 获取了在线程 A 中初始化的单例。

理论上双重校验锁法是线程安全的，并且，这种方法实现了 **lazyloading**。

## 7. 懒汉模式与饿汉模式区别

饿汉模式是最简单的一种实现方式，饿汉模式在类加载的时候就对实例进行创建，实例在整个程序周期都存在。它的好处是只在类加载的时候创建一次实例，不会存在多个线程创建多个实例的情况，避免了多线程同步的问题。它的缺点也很明显，即使这个单例没有用到也会被创建，而且在类加载之后就被创建，内存就被浪费了。

## 8. 双重校验锁方法与线程安全的懒汉模式区别

可以看到上面在同步代码块外多了一层 `instance` 为空的判断。由于单例对象只需要创建一次，如果后面再次调用 `getInstance()` 只需要直接返回单例对象。因此，大部分情况下，调用 `getInstance()` 都不会执行到同步代码块，从而提高了程序性能。不过还需要考虑一种情况，假如两个线程 A、B，A 执行了 `if (instance == null)` 语句，它会认为单例对象没有创建，此时线程切到 B 也执行了同样的语句，B 也认为单例对象没有创建，然后**两个线程依次执行同步代码块，并分别创建了一个单例对象**。为了解决这个问题，还需要在同步代码块中增加 `if(instance == null)` 语句，也就是上面看到的代码 2。

## 9. 单例模式与静态变量区别

首先解决引用的唯一实例可能被重新赋值的问题，单例模式中的 `getInstance()` 静态方法实现时，**1.采用懒汉式创建一个对象（当然这只是创建方式的一种），规避了这一风险，无则创建，有则跳过创建。****2.其次，`getInstance()` 静态方法定义在该类的内部，获取该类对象的引用位置非常明确，无需额外的沟通商定，团队成员拿起即用。**最后一个区别并不是很明显，声明一个静态变量，**4.实际上，我们会直接对其进行初始化赋值，这样，在内存占用上，所占用的内存为该初始化赋值对象实际的内存。而单例模式可以通过懒汉创建法延迟该内存的占用**，要知道，当一个静态变量只进行声明，而不进行初始化时，实际的内存占用只有 4 个字节

## 4.2 设计模式六大原则

<https://www.cnblogs.com/dolphin0520/p/3919839.html>

1. **单一职责原则(Single Responsibility Principle, SRP):** 一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。
2. **开闭原则(Open-Closed Principle, OCP):** 一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。
3. **里氏代换原则(Liskov Substitution Principle, LSP):** 所有引用基类（父类）的地方必须能透明地使用其子类的对象。
4. **依赖倒转原则(Dependency Inversion Principle, DIP):** 抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程
5. **接口隔离原则(Interface Segregation Principle, ISP):** 使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。
6. **迪米特法则(Law of Demeter, LoD):** 一个软件实体应当尽可能少地与其他实体发生相互作用

#### 4.3 java 动态代理

<http://www.cnblogs.com/xiaolu501395377/p/3383130.html>

在 java 的动态代理机制中，有两个重要的类或接口，一个是 `InvocationHandler`(Interface)、另一个则是 `Proxy`(Class)，这一个类和接口是实现我们动态代理所必须用到的。首先我们先来看看 java 的 API 帮助文档是怎么样对这两个类进行描述的：

##### 1. `InvocationHandler`:

每一个动态代理类都必须要实现 `InvocationHandler` 这个接口，并且每个代理类的实例都关联到了一个 `handler`，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由 `InvocationHandler` 这个接口的 `invoke` 方法来进行调用。我们来看看 `InvocationHandler` 这个接口的唯一一个方法 `invoke` 方法

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable  
  
proxy: 指代我们所代理的那个真实对象  
method: 指代的是我们所要调用真实对象的某个方法的 Method 对象  
args: 指代的是调用真实对象某个方法时接受的参数
```

##### 2. `Proxy`

`Proxy` 这个类的作用就是用来动态创建一个代理对象的类，它提供了许多的方法，但是我们用的最多的就是 `newProxyInstance` 这个方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException
```

`loader:` 一个 `ClassLoader` 对象，定义了由哪个 `ClassLoader` 对象来对生成的代理对象进行加载

`interfaces:` 一个 `Interface` 对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了

h: 一个 InvocationHandler 对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个 InvocationHandler 对象上

个人理解：就是说通过 Proxy 的 ProxyInstance 类创建出一个代理类，这个代理类执行的关于它代理的对象（真正的对象）的方法（代理类可以自己定义自己的方法，要区别），通过一个 InvocationHandler，InvocationHandler 是一个接口，接口中有 invoke 方法， invoke 方法关联到一个真正的对象，然后去执行真正对象的方法，来实现代理。

- 所了解的设计模式
  - 工厂模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method 使一个类的实例化延迟到了子类。
  - 单例模式：保证一个类只有一个实例，并提供一个访问它的全局访问点；
  - 适配器模式：将一类的接口转换成客户希望的另外一个接口，Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。
  - 装饰者模式：动态地给一个对象增加一些额外的职责，就增加的功能来说，Decorator 模式相比生成子类更加灵活。
  - 代理：为其他对象提供一种代理以控制对这个对象的访问
  - 迭代器模式：提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。
- 单例模式的注意事项
  - 尽量使用懒加载
  - 双重检测实现线程安全
  - 构造方法为 private
  - 定义静态的 Singleton instance 对象和 getInstance() 方法
- jdk 源码中用到的设计模式
  - 装饰器模式：IO 流中
  - 迭代器模式：Iterator
  - 单例模式：java.lang.Runtime
- 代理模式：RMI

**单例模式、工厂模式、适配器模式、迭代器模式、装饰器模式、代理模式、观察者模式**

**单例模式：**只有一个实例：最优实现，定义一个 static volatile 的实例对象不初始化，

定义一个 getInstance 的方法，其中使用 synchronized 进行实例未 null 时的创建。

注意事项：线程安全、懒加载（getInstance 方法）、构造方法 private

**工厂模式：**为创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的

分为简单工厂模式、工厂方法模式、抽象工厂模式

**简单工厂模式：**由工厂类决定实例化哪个产品类

**工厂方法模式：**定义一个用于创建对象的接口，让子类决定创建那个类。Factory Method 使一个类的实例化延迟到其子类中

**抽象工厂模式：**定义一个抽象的工厂，里面定义多个产品的类

**适配器模式：**将一个类的接口转换成客户希望的另外一个接口。具体实现，适配类继承了被适配类，实现了标准接口方法。在标准接口方法中调用了被适配的方法

**迭代器模式：**Iterator

**装饰器模式：**实现动态的为对象添加功能。对象接口，实现接口的实例，继承接口的装饰器（构造方法中传递被装饰的实例），继承装饰器的实例

**代理模式：**

代理类和被代理类实现同一接口，代理类里面有被代理类的实力，程序直接调用代理类的方法，然后代理类去调用被代理类相同的方法，但是代用方法的前后可以添加其他操作

装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问

**观察者模式：**角色（被观察对象）有一个包含所有观察者对象的引用集合，通过轮询集合保证同步：

**适用场景：**紧耦合部分对象/

```

public class Singleton {

    // 单例模式 饿汉
    private static final Singleton instance = new Singleton();
    // 静态方法返回该类的实例
    public static Singleton getInstance() {
        return instance;
    }

    //单例模式 饱汉
    private static volatile Singleton anotherinstance;
    public static Singleton getInstance2(){
        if(anotherinstance == null){
            synchronized (Singleton.class){
                if (anotherinstance == null) {
                    anotherinstance = new Singleton();
                }
            }
        }
        return anotherinstance;
    }
}

```

单例模式（双检锁模式）、简单工厂、观察者模式、适配器模式、职责链模式等等  
享元模式模式 选两个画下 UML 图

手写单例

写的是静态内部类的单例，然后他问我这个地方为什么用 `private`，这儿为啥用 `static`，  
这就考察你的基本功啦

静态类与单例模式的区别

单例模式 `double check`

单例模式都有什么，都是否线程安全，怎么改进（从 `synchronized` 到 双重检验锁  
到 枚举 `Enum`）

- 1、 基本的设计模式及其核心思想
- 2、 来，我们写一个单例模式的实现。这里有一个深坑，详情请见《JVM》  
第 370 页
- 3、 剩下的什么的就随缘吧。适配器类图什么的我也画过啊。

4、 基本的设计原则。如果有人问你接口里的属性为什么都是 final static 的，记得和他聊一聊设计原则。

### 设计模式了解哪些

#### jdk 中哪些类用了哪些设计模式

# 算法

- 1.最小堆；
- 2.大数据归并排序、遗传算法 sqrt () 是实现，归并排序实现，mapreduce 排序
- 3.快速排序和堆排序的优缺点，为什么？
- 4.一个是链表相加，思路就是反转 然后求和，另一个是多个有序数组 归并，用优先队列就好
- 5.最后一个算法题，是一个装水的问题，问在装多少，我用的双指针
- 6.LintCode -最小子串覆盖
- 7.查找数组中的最小元素 二分
- 8.第二题是算两个没有公共字母的字符串的最大长度积
- 9.LintCode - 反转二叉树
- 10.LintCode - 翻转字符串
- 11.单链表的快速排序
- 12.LintCode - 接雨水 III, 写具体的方法和算法
- 13.整数去重问题
- 14.找出增序排列中一个数字第一次和最后一次出现的数组下标
- 15.海量数据去重
- 16.找出海量数据中前 10 个最大的数（数据有重复）
- 17.数组先升序再降序，找出最大数
- 18.正整数数组，拼出一个最大的正数
- 19.一个正整数数组，给其中一个数字加 1，使得所有数乘积最大，找出加 1 的那个数字
- 20.手写快排、堆排 二分查找
- 21.单词接龙的程序
- 22.括号匹配；
- 23.一个数组存着负数与正数，将正数放在前年，负数放在后面
- 24.母鸡、公鸡和小鸡问题：公鸡五块一只，母鸡三块一只，小鸡一块三只，用 100 元买 100 只鸡的
- 25.各种排序算法的时间复杂度和空间复杂度
- 26.Dijkstra(求最短路径)
- 27.旋转数组找某个数
- 28.哲学家问题
- 29.最大连续子序列和
- 30.最左前缀匹配
- 31.单链表反转并输出

32. 找到非排序数组中未出现的第一个正整数
33. 在 0 到 n 这  $n+1$  个数中取 n 个数，如何找到缺少的那个。
34. 链表中如何判断有环路
35. 一个二维矩阵  $n \times n$  中，n 对应表示各个节点，每个节点之间有连线就在相应位置上标识 1，如何在其中判断出是不是一个图  
(任一节点开始遍历，深度遍历，每遍历一个点进行一个标记，当深度遍历到自己访问过的点时，代表存在环，即是个图)
36. 二叉树中找出从根到叶子节点中和最大的那条路径
37. 实现二叉树的广度优先遍历
38. 手写直接插入排序
39. 在一个字符串中找出第一个字符出现的位置，保证高效
40. N 级楼梯，一次一步或两步
41. 深度优先遍历，广度优先遍历算法 在什么地方可以应用
42. 杨辉三角形的算法，第 N 行的数的计算
43. 给定两个全都是大写的字符串 a, b a 的长度大于 b 的长度，问如何判断 b 中的所有字符都在 a 中（首先 a, b 排序，然后再两列比较）
44. 一致性哈希算法
45. 手写双向链表删除倒数第二个结点并分析
46. 找到数组第三大数，没有则返回最大数
47. 如何找到一条单链表的中间结点
48. 从 10 亿个数中找不重复的数  
将 10 亿个数排序后存在不同子文件中，每个子文件在内存中用 HashMap 来进行判断，比如放入 map 中是 (int 和 boolean 它们封装类的键值对)，第一次放进去时候 boolean 为 false，当 map 中有这个数之后再放进去时，将 false 改为 true。最后遍历 map 找出为 false 的数就是不重复的。
49. 判断二叉树是否为平衡二叉树。
50. 0G 文件的淘宝商品编号，只有 512M 内存，怎么判断究竟是不是合法编号（即编号是否存在）
51. 假如淘宝存着一个包含 10w 个敏感词的词库，紧接着需要从多个商品标题中随机抽查 3 个有没有包含敏感词的商品
52. 查找中间链表元素
53. 图算法
54. 平衡树的旋转。
55. 一道算法题，在一个整形数组中，有正数有负数，找出和最大的子串
56. 动态规划的思想
57. 给出一个字符数组，找出第一个出现次数最多的字符，注意是第一个
58. 一个无序数组找第 K 大的元素
59. 找出数组两个数的和等于给定的数
60. 无序数组找中位数（时间复杂度为  $\log N$ ），
61. 两个有序数组找中位数（时间复杂度为  $\log N$ ）
62. 写大数加法代码
63. 输出二叉树从左边看过去能看到的所有节点
64. 算法题：给定一个翻转过的有序数组，找出翻转点的下标，如：原数组 1, 2, 3, 5, 6, 7, 8，翻转后的数组 5, 6, 7, 8, 1, 2, 3，翻转点下标是 5

- 65.给定一个整数数组，数组中元素无重复。和一个整数 `limit`, 求数组元素全排列，要求相邻两个数字和小于 `limit`
- 66.算法题：行列都有序二维数组，找出指定元素的位置，扩展到三维数组呢
- 67.输入指定个数的字符串,按照字符串长度进行排序,然后重新从短到长输出,排序算法要自己写不能用自带的
- 68.求二叉树深度,比较坑的是,牛客网没有提供二叉树构造的输入样例,所以还要自己写个构造二叉树的算法
- 69.二叉树的几种遍历方式
- 70.对整数分解质因数,  $90=2^2 \cdot 3^2 \cdot 5$
- 71.二叉树非递归后续遍历
- 72.实现三个线程轮流打印 ABC 十次
- 73.列举集合的所有子集
- 74.给单链表排序，时间复杂度  $O(n \log n)$ ,空间复杂度  $O(1)$
- 75.判断一个字符串能否被字典完全分词(dp)
- 76.找出只出现一次的数字…链表的中间节点，链表的第  $n/m$  个节点 找出链表的中间节点，找出链表的三分之一节点，五分之一节点...
- 77.打印杨辉三角
- 78、手写栈实现队列
- 79.给定一个 2 叉树，打印每一层最右边的结点
- 80.给定一个数组，里面只有一个数出现了一次，其他都出现了两次。怎么得到这个出现了一次的数？
- 81.在 6 基础上，如果有两个不同数的出现了一次，其他出现了两次，怎么得到这两个数？
- 82.查找有序数组和为 S 的数
- 83.如有个公司有 10000 名员工，要求按照年龄来排序要求时间复杂度  $O(N)$
- 83.两个 int32 整数 m 和 n 的二进制表达有多少位不同
84. 全排列的算法思路

1、两个 int32 整数 m 和 n 的二进制表达有多少位不同

// 思路：①= 进制的位比较，可以想到异或运算。相同

// ② 根据异或的结果，只需要查看该结果的二进制

```
public int getDifferentCount (int m, int n) {  
    int temp = m ^ n; // 先异或  
    int count = 0;  
    while (temp != 0) { // 统计二进制表示  
        temp = temp & (temp - 1);  
        count++;  
    }  
    return count;  
}
```

84. 字符串反转

85. 拓扑排序

86. .树的中序遍历，除了递归和栈还有什么实现方式 中序遍历的非递归做法？引出 BFS 和 DFS 的区别

87. 拓扑排序思想

88. 给定 n 个数，寻找第 k 小的数

89. 写了一个小程序，给定一段字符串，主要为赋值型的字符串，让把它们对应到 map 里面

90. 1000 以内的素数

91. 手写希尔排序

92. 利用数组，实现一个循环队列类

93. 写一个汉诺塔问题，打印出转移路径，接着写一个二叉树前序遍历的代码，最后让写一个满二叉树实现，并层次遍历的代码，连写四个代码

94.

。第一道题是一个字母组成的字符串，对该字符串字母序进行排序，大写在小写前面，时间复杂度  $O(n)$ ，如 AaB 是有序的，ABa 是无序的。第二道题计算  $f(x,n)=x+x^2+\dots+x^n$ , 要求乘法次数最少。

### 95. 拓扑排序思想

96.一个字符串数组，现给定一个 string 去进行找出对应的数组中字符串的下标（可以有容错，但两字符串长度必须一致，容错为 2）

例如： ["hello","hj","abc"] key="hellg" 返回下角标 0

97. 图的 prime 算法 kruskal 算法 dijkstra 算法 解决什么问题？分别写一下 伪代码

98. 从一堆字符串中，去除重复的字符，并输出

99. 手写 Kmp 算法

100. 对一个基本有序的数组应该采用什么方式进行排序，对一个乱序的数组应该采用什么方式排序能快速找到前 n 个数？为什么？

101. 给定一个数组，里面放置任意数量的随机数，如何快速统计出数组中重复的数字以及出现次数

102. 给定字母集合(a-z)，求出由集合中这些字母组成的所有非空子集

103. 第一道题是用 5 行代码实现字符个数统计；第二题是反转单链表；第三题快速排序

104. 接着推导快速排序的时间复杂度为什么是  $O(n\log n)$ ？

105. 并发场景下的多线程代码水题

106. 算法题 一个数组里的数据只有一个是 3 个相同的，其他都是两个相同 怎么找出这个数 围绕上一题优化

107. 字符转 int 型，考虑负数，异常等问题

108. 跳表

109. 给定 n 个左括号以及 n 个右括号，打印出所有合法的括号组合

110. 给定四个点如何判断是否为矩形

## 海量数据

海量 URL 数据，低内存情况下找重复次数最高的那一个

10 亿个数求 100 个最大的

大文件排序

给定三个大于 10G 的文件（每行一个数字）和 100M 内存的主机，找到在三个文件都出现且次数最多的 10 个字符串

求两个 int 数组的并集、交集

1t query 统计前 k 个热门的

对 10G 个数进行排序，限制内存为 1G 大数问题，但是这 10G 个数可能是整数，字符串以及中文该如何排序，

假如有 100 万个玩家，需要对这 100W 个玩家的积分中前 100 名的积分，按照顺序显示在网站中，要求是实时更新的。积分可能由做的任务和获得的金钱决定。问如何对着 100

万个玩家前 100 名的积分进行实时更新？

我跟他讨论了什么分治啊、Hash 啊，但后来他都说我的方法都是从全局的数据进行考虑的，这样空间和时间要求太多，并且不现实。后来我跟他一聊，最后他给出了解决方法，就是利用缓存机制，缓存---tomcat---DB，层级计算，能不用到 DB 层就别用，因为每进一层，实现起来都会更复杂和更慢。解决的思路就是，考虑出了前 100 名的后 100W-100 名玩家的积分，让变化的积分跟第 100 名比较，如果比第 100 名高，那就替换的原则。

10 亿条短信，找出前一万条重复率高的

对一万条数据排序，你认为最好的方式是什么

一个大文件，里面是很多字符串，用最优的方式计算出一个字符串是否存在

QQ 每天都会产生大量的在线日志记录，假设每天的在线日志记录有十亿条，请设计一个算法快速找出今天的在线人数

有 4 个文件，每个文件大小为 10G，每一行是一个单词，最后统计出 Top10 的单词

## 七 . 数据结构与算法

### 7.1 排序

<https://blog.csdn.net/whuslei/article/details/6442755>

| 排序方法   | 时间复杂度           |                 |                 | 空间复杂度           | 稳定性 | 复杂性 |
|--------|-----------------|-----------------|-----------------|-----------------|-----|-----|
|        | 平均情况            | 最坏情况            | 最好情况            |                 |     |     |
| 直接插入排序 | $O(n^2)$        | $O(n^2)$        | $O(n)$          | $O(1)$          | 稳定  | 简单  |
| 希尔排序   | $O(n \log_2 n)$ | $O(n \log_2 n)$ |                 | $O(1)$          | 不稳定 | 较复杂 |
| 冒泡排序   | $O(n^2)$        | $O(n^2)$        | $O(n)$          | $O(1)$          | 稳定  | 简单  |
| 快速排序   | $O(n \log_2 n)$ | $O(n^2)$        | $O(n \log_2 n)$ | $O(n \log_2 n)$ | 不稳定 | 较复杂 |
| 直接选择排序 | $O(n^2)$        | $O(n^2)$        | $O(n^2)$        | $O(1)$          | 不稳定 | 简单  |
| 堆排序    | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(1)$          | 不稳定 | 较复杂 |
| 归并排序   | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n \log_2 n)$ | $O(n)$          | 稳定  | 较复杂 |
| 基数排序   | $O(d(n+r))$     | $O(d(n+r))$     | $O(d(n+r))$     | $O(n+r)$        | 稳定  | 较复杂 |

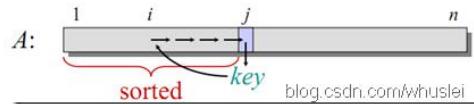
#### 7.1.1 直接插入排序

## 一、直接插入排序(插入排序)。

### 1、算法的伪代码(这样便于理解)：

```
INSERTION-SORT (A, n)
for j ← 2 to n
    do key ← A[j]
    i ← j - 1
    while i > 0 and A[i] > key
        do A[i+1] ← A[i]
        i ← i - 1
    A[i+1] = key
```

2、思想：如下图所示，每次选择一个元素K插入到之前已排好序的部分A[1…i]中，插入过程中K依次由后向前与A[1…i]中的元素进行比较。若发现A[x]>=K，则将K插入到A[x]的后面，插入前需要移动元素。



### 3、算法时间复杂度。

最好的情况下：正序有序(从小到大)，这样只需要比较n次，不需要移动。因此时间复杂度为O(n)

最坏的情况下：逆序有序，这样每一个元素就需要比较n次，共有n个元素，因此实际复杂度为O(n<sup>2</sup>)

平均情况下：O(n<sup>2</sup>)

### 4、稳定性。

理解性记忆比死记硬背要好。因此，我们来分析下。稳定性，就是有两个相同的元素，排序先后的相对位置是否变化，主要用在排序时有多个排序规则的情况下。在插入排序中，K1是已排序部分中的元素，当K2和K1比较时，直接插到K1的后面(没有必要插到K1的前面，这样做还需要移动！！)，因此，插入排序是稳定的。

```
public class Solution {
    public void sortIntegers2(int[] A) {
        InsertSort(A);
    }
    public void InsertSort(int[] A)
    {
        int i, j, k;
        for(i=1; i<A.length; i++)
        {
            for(j=i-1; j>=0; j--)
            {
                if(A[j] <= A[i])//寻找第一个小于 A[i]的位置, 也就是[i]
                该插入的地方
                    break;
            }
            if(j != i-1) //这个判断的意思是如果是刚好 A[i-1]这个地方小于 A[i], 那么不需要操作
            {
                int temp = A[i];
                for(k=i-1; k>j; k--)
                    A[k+1] = A[k];
                A[k+1] = temp;//循环结束 k=j, 故需要 k+1 放在该放的位置
            }
        }
    }
}
```

```

        }
    }
}
}

```

### 7.1.2 希尔排序

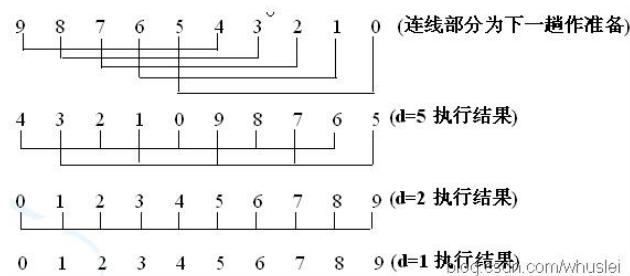
1、思想：希尔排序也是一种插入排序方法，实际上是一种分组插入方法。先取定一个小于n的整数d1作为第一个增量，把表的全部记录分成d1个组，所有距离为d1的倍数的记录放在同一个组中，在各组内进行直接插入排序；然后，取第二个增量d2( $d_2 < d_1$ )，重复上述的分组和排序，直至所取的增量 $dt=1$ ( $dt < dt-1 < \dots < d_2 < d_1$ )，即所有记录放在同一组中进行直接插入排序为止。

例如：将 n 个记录分成 d 个子序列：

```

{ R[0],      R[d],      R[2d], ..., R[kd] }
{ R[1],      R[1+d],   R[1+2d], ..., R[1+kd] }
...
{ R[d-1],   R[2d-1], R[3d-1], ..., R[(k+1)d-1] }

```



说明： $d=5$  时，先从 $A[d]$ 开始向前插入，判断 $A[d-d]$ ，然后 $A[d+1]$ 与 $A[(d+1)-d]$ 比较，如此类推，这一回合后将原序列为d个组。由后向前

#### 2、时间复杂度。

**最好情况：**由于希尔排序的好坏和步长d的选择有很多关系，因此，目前还没有得出最好的步长如何选择(现在有些比较好的选择了，但不确定是否是最好的)。所以，不知道最好的情况下的算法时间复杂度。

**最坏情况下：**  $O(N \log N)$ ，最坏的情况下和平均情况下差不多。

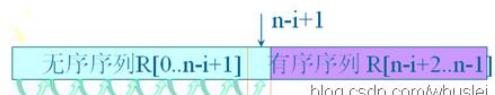
**平均情况下：**  $O(N \log N)$

#### 3、稳定性。

由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是**不稳定的**。(有个猜测，方便记忆：一般来说，若存在不相邻元素间交换，则很可能是不稳定的排序。)

### 7.1.3 冒泡排序

1、基本思想：通过无序区中相邻记录关键字间的比较和位置的交换，使关键字最小的记录如气泡一般逐渐往上“漂浮”直至“水面”。



#### 2、时间复杂度

**最好情况下：** 正序有序，则只需要比较n次。故，为 $O(n)$

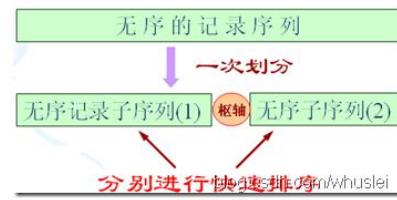
**最坏情况下：** 逆序有序，则需要比较 $(n-1)+(n-2)+\dots+1$ ，故，为 $O(N^2)$

#### 3、稳定性

排序过程中只交换相邻两个元素的位置。因此，当两个数相等时，是没必要交换两个数的位置的。所以，它们的相对位置并没有改变，冒泡排序算法是稳定的！

### 7.1.4 快速排序

1、思想：它是由冒泡排序改进而来的。在待排序的n个记录中任取一个记录(通常取第一个记录)，把该记录放入适当位置后，数据序列被此记录划分成两部分。所有关键字比该记录关键字小的记录放置在前一部分，所有比它大的记录放置在后一部分，并把该记录排在这两部分的中间(称为该记录归位)，这个过程称作一趟快速排序。



说明：最核心的思想是将小的部分放在左边，大的部分放到右边，实现分割。

## 2、算法复杂度

最好的情况下：因为每次都将序列分为两个部分(一般二分都复杂度都和 $\log N$ 相关)，故为  $O(N \log N)$

最坏的情况下：基本有序时，退化为冒泡排序，几乎要比较 $N^2$ 次，故为 $O(N^2)$

## 3、稳定性

由于每次都需要和中轴元素交换，因此原来的顺序就可能被打乱。如序列为 5 3 3 4 3 8 9 10 11会将3的顺序打乱。所以说，快速排序是不稳定的！

```
public class Solution {

    public void sortIntegers2(int[] A) {
        // write your code here
        quicksort(A, 0, A.length-1);
    }

    public void quicksort(int[] A, int begin, int end)
    {
        int i = begin;
        int j = end;
        if(i >= j)
        {
            return;
        }
        int keng = A[i];
        while(i < j)
        {
            while(i < j && A[j] > keng)
            {
                j--;
            }
            if(i < j && A[j] <= keng)
            {
                A[i] = A[j];
                i++;
            }
        }
        while(i < j && A[i] < keng)
        {
            i++;
        }
        if(i < j && A[i] >= keng)
    }
}
```

```

    {
        A[j] = A[i];
        j--;
    }
}
A[i] = keng;
quicksort(A, begin, i-1);
quicksort(A, i+1, end);
}
}

```

```

import java.util.*;
public class Solution {
    /**
     * @param A: an integer array
     * @return: nothing
     */
    public void sortIntegers2(int[] A) {
        quicksort(A, 0, A.length-1);
    }
    public int partition(int[] A, int low, int high)
    {
        int hole = A[low];
        int i = low;
        int j = high;
        if(low < high)
        {
            while(i < j)
            {
                while(i < j && A[j] >= hole)
                    j--;
                if(i < j && A[j] < hole)
                {
                    A[i] = A[j];
                    i++;
                }
                while(i < j && A[i] <= hole)
                    i++;
                if(i < j && A[i] > hole)
                {
                    A[j] = A[i];
                    j--;
                }
            }
        }
    }
}

```

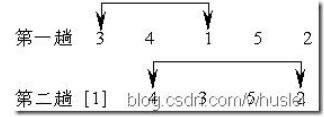
```

        }
        A[i] = hole;
        return i;
    }
    return -1;
}
public void quicksort(int[] A, int low, int high)
{
    LinkedList<Integer> stack = new LinkedList<>();
    int k = -1;
    if(low < high)
    {
        stack.offerFirst(low);
        stack.offerFirst(high);
        while(stack.size() != 0)
        {
            int right = (int)stack.poll();
            int left = (int)stack.poll();
            k = partition(A, left, right);
            if(k-1 > left)
            {
                stack.offerFirst(left);
                stack.offerFirst(k-1);
            }
            if(k+1 < right)
            {
                stack.offerFirst(k+1);
                stack.offerFirst(right);
            }
        }
    }
}

```

### 7.1.5 直接选择排序

1、思想：首先在未排序序列中找到最小元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小元素，然后放到排序序列末尾。以此类推，直到所有元素均排序完毕。具体做法是：选择最小的元素与未排序部分的首部交换，使得序列的前面为有序。



## 2、时间复杂度。

最好情况下：交换0次，但是每次都要找到最小的元素，因此大约必须遍历 $N \times N$ 次，因此为 $O(N \times N)$ 。减少了交换次数！

最坏情况下，平均情况下： $O(N \times N)$

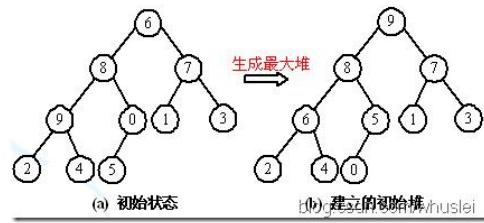
## 3、稳定性

由于每次都是选取未排序序列A中的最小元素x与A中的第一个元素交换，因此跨距离了，很可能破坏了元素间的相对位置，因此选择排序是不稳定的！

```
public class Solution {  
    public void sortIntegers2(int[] A) {  
        SelectSort(A);  
    }  
    public void SelectSort(int[] A)  
    {  
        for(int i=0;i<A.length;i++)  
        {  
            int minIndex = i;  
            for(int j=i;j<A.length;j++)  
            {  
                if(A[j] < A[minIndex])  
                    minIndex = j;  
            }  
            int temp = A[i];  
            A[i] = A[minIndex];  
            A[minIndex] = temp;  
        }  
    }  
}
```

## 7.1.6 堆排序

1、思想：利用完全二叉树中双亲节点和孩子节点之间的内在关系，在当前无序区中选择关键字最大(或者最小)的记录。也就是说，以最小堆为例，根节点为最小元素，较大的节点偏向于分布在堆底附近。



## 2、算法复杂度

最坏情况下，接近于最差情况下： $O(N \times \log N)$ ，因此它是一种效果不错的排序算法。

## 3、稳定性

堆排序需要不断地调整堆，因此它是一种不稳定的排序！

## 4、代码(c版，看代码后更容易理解！)

## 4、代码(c版，看代码后更容易理解！)

```
void heapSort(MyStore *s){//此处为最大堆
    //初始化堆
    int n=s->length,i,tmp;
    for(i=n/2;i>1;i--){
        adjustHeap(s->r,i,n);
    }

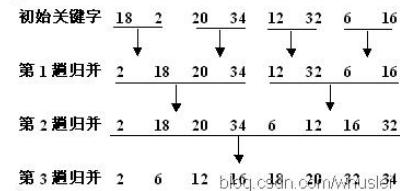
    //每次输出一个堆顶元素，就需要重新调整一次堆，这里的输出，其实是依次将最大数放到数组的尾部
    for(i=n;i>2;i--){//i=1时，这个只有一个元素，不用调整了
        //将堆顶的元素放到数组的尾部
        tmp=s->r[1];//先保存堆顶
        s->r[1]=s->r[i];//将最后一个值放到堆顶
        s->r[i]=tmp;//最后，将目前的最大值保存到数组尾部
        adjustHeap(s->r,1,i-1);//此时堆的大小是i-1，不是i
    }

    void adjustHeap(int a[],int s,int e){//s代表要调整的那个节点，e代表已经是堆的节点的最后一个节点
        int i=s,j=2*i,tmp;//j是[i]的左子树节点
        tmp=a[i];//先保存这个要调整的节点。下降法。
        while(j<=e){//左子树
            //如果j==e，那将不会有右子树。就不比比较和右子树的大小了。最后j指向的是左右子树中较小的那个
            if(j<e && a[j]<a[j+1]) { j++; }
            if(a[j]>tmp){//要调整了
                a[i]=a[j];
                i=j;
                j=2*i;//进行下一轮调整
            }else{
                break;//调整结束
            }
        }
        a[i]=tmp;
    }
}
```

[blog.csdn.net/whuslei](http://blog.csdn.net/whuslei)

## 7.1.7 归并排序

1、思想：多次将两个或两个以上的有序表合并成一个新的有序表。



## 2、算法时间复杂度

最好的情况下：一趟归并需要 $n$ 次，总共需要 $\log N$ 次，因此为 $O(N \times \log N)$

最坏的情况下，接近于平均情况下，为 $O(N \times \log N)$

说明：对长度为 $n$ 的文件，需进行 $\log N$ 趟二路归并，每趟归并的时间为 $O(n)$ ，故其时间复杂度无论是在最好情况下还是在最坏情况下均是 $O(n \log n)$ 。

## 3、稳定性

归并排序最大的特色就是它是一种稳定的排序算法。归并过程中是不会改变元素的相对位置的。

4、缺点是，它需要 $O(n)$ 的额外空间。但是很适合于多链表排序。

5、代码(略)[blog.csdn.net/whuslei](http://blog.csdn.net/whuslei)

```

public class Solution {
    public void sortIntegers2(int[] A) {
        int [] temp = new int[A.length];
        mergeSort(A, 0, A.length-1, temp);
    }

    public void mergeArray(int[] A, int leftBegin, int leftEnd, int rightBegin, int rightEnd, int[] temp)
    {
        int i = leftBegin, j = rightBegin;
        int k = 0;
        while (i <= leftEnd && j <= rightEnd) //谁小就把谁弄到 temp 数组里, 直到有一方先全部弄完
        {
            if (A[i] < A[j])
                temp[k++] = A[i++];
            else
                temp[k++] = A[j++];
        }
        while(i <= leftEnd) //无法确定哪一方先全部遍历结束, 所以写了两个循环, 只要没结束就会继续复制到 temp 中
            temp[k++] = A[i++];
        while(j <= rightEnd)
            temp[k++] = A[j++];
        for(i=0;i<k;i++)
            A[leftBegin+i] = temp[i]; //将排序好的 temp 复制到 A[leftBegin] 到 A[rightEnd] 中, 完成排序
    }

    public void mergeSort(int[] A, int left, int right, int[] temp)
    {
        if(left < right)
        {
            int mid = (left + right) / 2; //递归的过程中一层层往下, 直到 left=right, 也就是当前要归并的两数组都是只有一个元素
            mergeSort(A, left, mid, temp);
            mergeSort(A, mid+1, right, temp);
            mergeArray(A, left, mid, mid+1, right, temp);
        }
    }
}

```

```

import java.util.*;
public class Solution {
    public void sortIntegers2(int[] A) {
        if(A.length == 0)

```

```

        return;
    System.out.println(A.length+"");
    int[] temp = new int[A.length];
    mergeSort(A, 0, A.length-1, temp);
}

public void mergeArray(int[] A, int leftBegin, int leftEnd, int
rightBegin, int rightEnd, int[] temp)
{
    int i = leftBegin, j = rightBegin;
    int k = 0;
    while (i <= leftEnd && j <= rightEnd) //谁小就把谁弄到 temp 数
组里, 直到有一方先全部弄完
    {
        if (A[i] < A[j])
            temp[k++] = A[i++];
        else
            temp[k++] = A[j++];
    }
    while(i <= leftEnd) //无法确定哪一方先全部遍历结束, 所以写了两
个循环, 只要没结束就会继续复制到 temp 中
        temp[k++] = A[i++];
    while(j <= rightEnd)
        temp[k++] = A[j++];
    for(i=0;i<k;i++)
        A[leftBegin+i] = temp[i]; //将排序好的 temp 复制到
A[leftBegin]到 A[rightEnd]中, 完成排序
}
public void mergeSort(int[] A, int left, int right, int[] temp)
{
    int s=2, i=0;
    while(s <= A.length-1)
    {
        i = 0;
        while(i+s <= A.length) //按照跨度 s 进行两两合并, s=2 两两
合并 s=4 四四合并
        {
            mergeArray(A, i, (i+i+s-1)/2, (i+i+s-1)/2+1, i+s-1, temp);
            i += s;
        }
        mergeArray(A, i-s, i-1, i, A.length-1, temp); //合并尾部剩下的
数据, 具体见我举的例子
        s *= 2;
    }
    mergeArray(A, 0, s/2-1, s/2, A.length-1, temp); //合并尾部与前面一
}

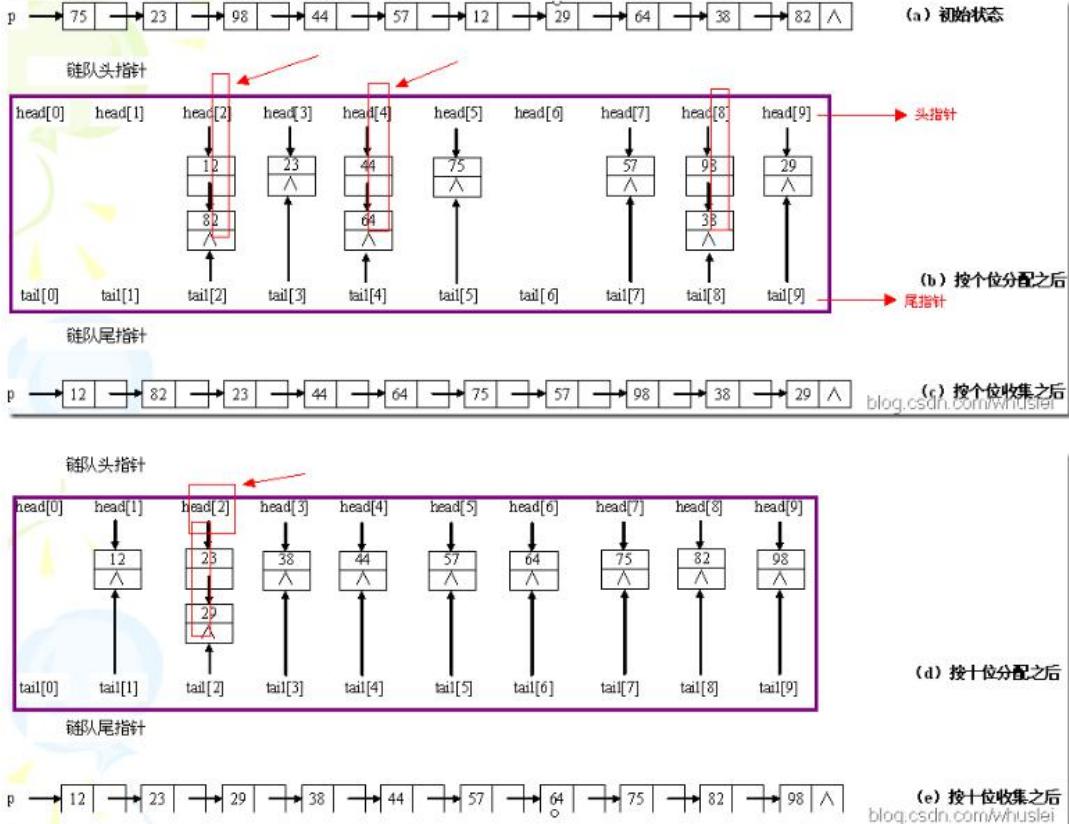
```

大片已经排序序的部分。

```
}
```

### 7.1.8 基数排序

1、思想：它是一种非比较排序。它是根据位的高低进行排序的，也就是先按个位排序，然后依据十位排序……以此类推。示例如下：



#### 2、算法的时间复杂度

分配需要 $O(n)$ , 收集为 $O(r)$ , 其中 $r$ 为分配后链表的个数, 以 $r=10$ 为例, 则有 $0 \sim 9$ 这样10个链表来将原来的序列分类。而 $d$ , 也就是位数(如最大的数是1234, 位数是4, 则 $d=4$ ), 即“分配-收集”的趟数。因此时间复杂度为 $O(d*(n+r))$ 。

#### 3、稳定性

基数排序过程中不改变元素的相对位置, 因此是稳定的!

4、适用情况: 如果有一个序列, 知道数的范围(比如 $1 \sim 1000$ ), 用快速排序或者堆排序, 需要 $O(N*\log N)$ , 但是如果采用基数排序, 则可以达到 $O(4*(n+10))=O(n)$ 的时间复杂度。算是这种情况下排序最快的!

#### 5、代码(略)

总结: 每种算法都要它适用的条件, 本文也仅仅是回顾了下基础。如有不懂的地方请参考课本。

如有转载, 请注明: blog.csdn.com/wluslei

## 7.2 树

### 7.2.1 二分查找树

- 定义：二叉搜索树或者是一棵空树，或者是这样的一棵二叉树：（1）若左子树不空，则左子树上所有结点的值均小于它的根结点的值；（2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值；（3）同时左、右子树都是二叉搜索树
- 插入：1.若当前的二叉查找树为空，则插入的元素为根节点，2.若插入的元素值小于根节点值，则将元素插入到左子树中，3.若插入的元素值不小于根节点值，则将元素插入到右子树中。
- 删除：1.p为叶子节点，直接删除该节点，再修改其父节点的指针（注意分是根节点和不是根节点），如图a。  
2.p为单支节点（即只有左子树或右子树）。让p的子树与p的父亲节点相连，删除p即可；（注意分是根节点和不是根节点）；如图b。  
3.p的左子树和右子树均不空。找到p的后继y，因为y一定没有左子树，所以可以删除y，并让y的父亲节点成为y的右子树的父亲节点，并用y的值代替p的值；或者方法二是找到p的前驱x，x一定没有右子树，所以可以删除x，并让x的父亲节点成为y的左子树的父亲节点
- 查找：如果根结点的关键字值等于查找的关键字，成功。否则，若小于根结点的关键字值，递归查左子树。若大于根结点的关键字值，递归查右子树。若子树为空，查找不成功。复杂度 $O(\log n)$

### 7.3 LRU 实现

<https://blog.csdn.net/hxqneuq2012/article/details/52709652>

根据操作系统所学：cache（缓存）可以帮助快速存取数据，但是容量小。

本题要求实现的是LRU cache，LRU的思想来自“最近用到的数据被重用的概率比最早用到的数据大的多”，是一种十分高效的cache。

解决本题的方法是：**双向链表+HashMap**。

注：HashMap O(1) 大家都懂。

对于双向链表的使用，基于两个考虑。

首先，Cache中块的命中是随机的，和Load进来的顺序无关。

其次，双向链表插入、删除很快，可以灵活的调整相互间的次序，时间复杂度为O(1)。

新建数据类型Node节点，Key-Value值，并有指向前驱节点后继节点的指针，构成双向链表的节点。

.红黑树与二叉树有什么区别、红黑树用途

直接上手红黑树和平衡二叉树区别

AVL 树的概念，四种旋转方式，AVL 树左右旋转的例子

红黑树的旋转 2node 节点插入和 3node 节点插入时候旋转的情况 简述伪代码

字典树

链表使用的循环链表还是双向链表

树和图的区别

了解哈夫曼树、b+/b-树、红黑树

栈和队列的区别，如何实现栈

有序集合底层数据结构是

找到二叉树第 m 层的第 n 个节点

.链表的结构，操作的时间复杂度分析

前缀树

堆与普通二叉树有什么区别

什么是递归，递归的几个条件？写递归要注意些什么？

## 八.数据库

### 8.1 索引 B 树 B+树

#### 8.1.1 索引特点优缺点适用场合

<http://blog.jobbole.com/24006/>

<http://www.yuanrengu.com/index.php/2017-01-13.html>

##### 8.1.1.1 索引的优缺点特点

- 索引的特点
  - 可以加快数据库的检索速度
  - 降低数据库插入、修改、删除等维护的速度
  - 只能创建在表上，不能创建到视图上
  - 既可以创建又可以间接创建
  - 可以在优化隐藏中使用索引
  - 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
  - 创建唯一性索引，保证数据库表中每一行数据的唯一性
  - 大大加快数据的检索速度，这是创建索引的最主要的原因
  - 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
  - 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间
  - 通过使用索引，可以在查询中使用优化隐藏器，提高系统的性能

##### • 索引的缺点

- 创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加
- 索引需要占用物理空间，除了数据表占用数据空间之外，每一个索引还要占一定的物理空间，如果建立聚簇索引，那么需要的空间就会更大
- 当对表中的数据进行增加、删除和修改的时候，索引也需要维护，降低数据维护的速度

##### 8.1.1.2 索引使用的注意事项

```

SELECT `sname` FROM `stu` WHERE `age`+10=30;-- 不会使用索引,因为所有索引列参与了计算

SELECT `sname` FROM `stu` WHERE LEFT(`date`,4) <1990; -- 不会使用索引,因为使用了函数运算,原理与上面相同

SELECT * FROM `houdunwang` WHERE `uname` LIKE '后盾%' -- 走索引

SELECT * FROM `houdunwang` WHERE `uname` LIKE "%后盾%" -- 不走索引

-- 正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因

-- 字符串与数字比较不使用索引;
CREATE TABLE `a` (`a` char(10));
EXPLAIN SELECT * FROM `a` WHERE `a`="1" -- 走索引
EXPLAIN SELECT * FROM `a` WHERE `a`=1 -- 不走索引

select * from dept where dname='xxx' or loc='xx' or deptno=45 --如果条件中有or,即使其中有条件带索引也不会使用。换言之,就是要求使用的所有字段,都必须建立索引,我们建议大家尽量避免使用or关键字

-- 如果mysql估计使用全表扫描要比使用索引快,则不使用索引

```

- 索引失效
  - 如果条件中有or,即使其中有条件带索引也不会使用(这就是问什么尽量少使用or的原因)
  - 对于多列索引,不是使用的第一部分,则不会使用索引
  - like查询是以%开头
  - 如果列类型是字符串,那一定要在条件中使用引号起来,否则不会使用索引
  - 如果mysql估计使用全表扫描比使用索引快,则不适用索引。

### 8.1.1.3 索引的适用场景

- 在什么情况下适合建立索引
  - 为经常出现在关键字order by、group by、distinct后面的字段,建立索引。
  - 在union等集合操作的结果集字段上,建立索引。其建立索引的目的同上。
  - 为经常用作查询选择的字段,建立索引。
  - 在经常用作表连接的属性上,建立索引。
  - 考虑使用索引覆盖。对数据很少被更新的表,如果用户经常只查询其中的几个字段,可以考虑在这几个字段上建立索引,从而将表的扫描改变为索引的扫描。

性别不适用建立索引?为什么?

因为你访问索引需要付出额外的IO开销,你从索引中拿到的只是地址,要想真正访问到数据还是要对表进行一次IO。假如你要从表的100万行数据中取几个数据,那么利用索引迅速定位,访问索引的这IO开销就非常值了。但如果你是从100万行数据中取50万行数据,就比如性别字段,那你相对需要访问50万次索引,再访问50万次表,加起来的开销并不会比直接对表进行一次完整扫描小。

## 8.1.2 Mysql 索引原理 B+树:

[https://blog.csdn.net/v\\_july\\_v/article/details/6530142](https://blog.csdn.net/v_july_v/article/details/6530142)

<https://blog.csdn.net/guoqiqing506/article/details/64122287>

索引是怎么优化查询效率的?这中间的过程能描述下吗?

可以从 B+树原理回答

### 1. 索引为什么采用 B+树

B+树更有利于对数据库的扫描

B 树在提高了磁盘 IO 性能的同时并没有解决元素遍历的效率低下的问题，而 B+树只需要遍历叶子节点就可以解决对全部关键字信息的扫描，所以对于数据库中频繁使用的 range query，B+树有着更高的性能。

B+树的磁盘读写代价更低

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 I/O 读写次数也就降低了。

B+树的查询效率更加稳定

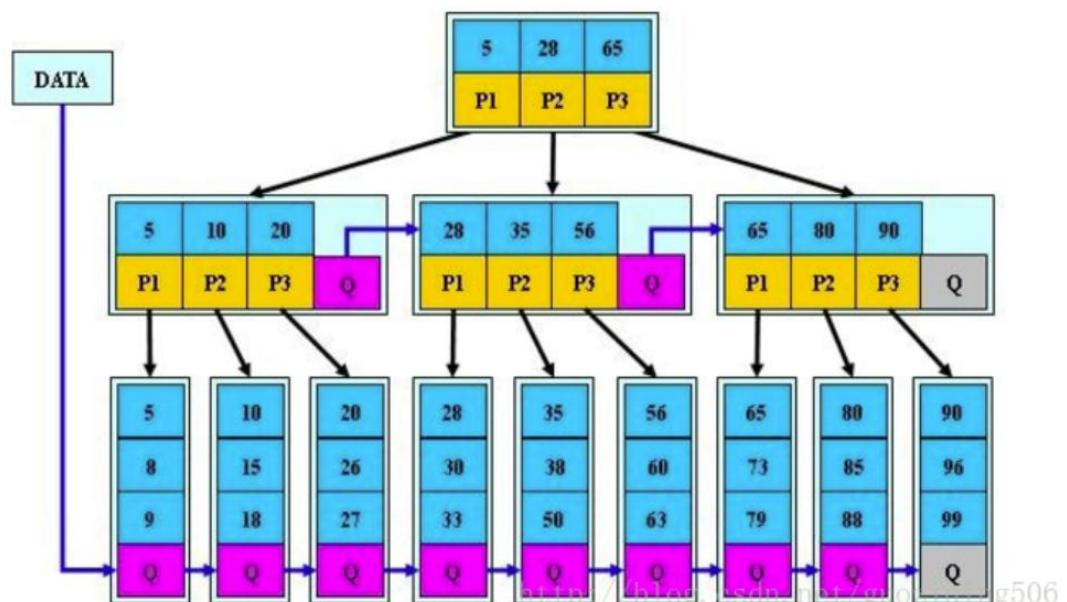
由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

### 2. B+树

B+树是 B 树的一种变形，它更适合实际应用中操作系统的文件索引和数据库索引。

定义：

1. 除根节点外的内部节点，每个节点最多有  $m$  个关键字，最少有  $\lceil \frac{m}{2} \rceil$  个关键字。其中每个关键字对应一个子树（也就是最多有  $m$  棵子树，最少有  $\lceil \frac{m}{2} \rceil$  棵子树）；
2. 根节点要么没有子树，要么至少有 2 棵子树；
3. 所有的叶子节点包含了全部的关键字以及这些关键字指向文件的指针，并且：
  1. 所有叶子节点中的关键字按大小顺序排列
  2. 相邻的叶子节点顺序链接（相当于是构成了一个顺序链表）
  3. 所有叶子节点在同一层

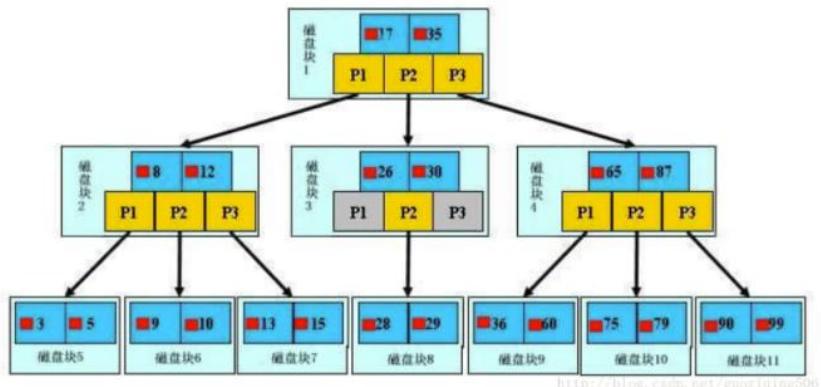


### 3. B-树（也就是 B 树）

## B-树

性质：是一种多路搜索树（并不是二叉的）：

1. 定义任意非叶子结点最多只有  $M$  个儿子；且  $M > 2$ ；
2. 根结点的儿子数为  $[2, M]$ ；
3. 除根结点以外的非叶子结点的儿子数为  $[M/2, M]$ ；
4. 每个结点存放至少  $M/2-1$  (取上整) 和至多  $M-1$  个关键字；(至少 2 个关键字)
5. 非叶子结点的关键字个数=指向儿子的指针个数-1；
6. 非叶子结点的关键字： $K[1], K[2], \dots, K[M-1]$ ；且  $K[i] < K[i+1]$ ；
7. 非叶子结点的指针： $P[1], P[2], \dots, P[M]$ ；其中  $P[1]$  指向关键字小于  $K[1]$  的子树， $P[M]$  指向关键字大于  $K[M-1]$  的子树，其它  $P[i]$  指向关键字属于  $(K[i-1], K[i])$  的子树；
8. 所有叶子结点位于同一层；



先简单对上图说明一下：

- 图中的小红方块表示对应关键字所代表的文件的存储位置，实际上可以看做是一个地址，比如根节点中17旁边的小红块表示的就是关键字17所对应的文件在硬盘中的存储地址。
- $P$  是指针，不用多说了，需要注意的是：指针，关键字，以及关键字所代表的文件地址这三样东西合起来构成了B树的一个节点，这个节点存储在一个磁盘块上

B-树搜索：

搜索：从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

### 4. B+树与 B-树的区别

答：1. 内部节点中，关键字的个数与其子树的个数相同，不像 B 树种，子树的个数

总比关键字个数多 1 个

2.所有指向文件的关键字及其指针都在叶子节点中，不像 B 树，有的指向文件的关键字是在内部节点中。换句话说，B+树中，内部节点仅仅起到索引的作用，

3.B+在搜索过程中，如果查询和内部节点的关键字一致，那么搜索过程不停止，而是继续向下搜索这个分支，B+为了找到这个关键字的指针。

### 8.1.3 索引分类

<https://www.cnblogs.com/heyonggang/p/6610526.html>

#### 8.1.3.0 索引分类

- 组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一
- 聚集索引：定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。
- 非聚集索引：唯一索引 普通索引 主键索引 全文索引
- UNIQUE(唯一索引)：不可以出现相同的值，可以有 NULL 值
- INDEX(普通索引)：允许出现相同的索引内容
- PRIMARY KEY(主键索引)：不允许出现相同的值
- fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维
- 

#### 8.1.3.1 创建索引语句

创建唯一，主键，普通，全文索引

```
ALTER TABLE 表名 ADD 索引类型 (unique,primary key,fulltext,index) [索引名] (字段名)

//普通索引
alter table table_name add index index_name (column_list) ;
//唯一索引
alter table table_name add unique (column_list) ;
//主键索引
alter table table_name add primary key (column_list) ;
```

创建组合索引

```
ALTER TABLE USER_DEMO ADD INDEX name_city_age (LOGIN_NAME(16),CITY,AGE);
```

建表时，LOGIN\_NAME长度为100，这里用16，**是因为一般情况下名字的长度不会超过16，这样会加快索引查询速度**，还会减少索引文件的大小，提高INSERT，UPDATE的更新速度。

如果分别给LOGIN\_NAME,CITY,AGE建立单列索引，让该表有3个单列索引，查询时和组合索引的效率是大不一样的，甚至远远低于我们的组合索引。虽然此时有三个索引，但mysql只能用到其中的那个它认为似乎是最有效的单列索引，另外两个是用不到的，也就是说还是一个全表扫描的过程。

建立这样的组合索引，就相当于分别建立如下三种组合索引：

```
LOGIN_NAME,CITY,AGE  
LOGIN_NAME,CITY  
LOGIN_NAME
```

为什么没有CITY,AGE等这样的组合索引呢？**这是因为mysql组合索引“最左前缀”的结果。**简单的理解就是只从最左边的开始组合，并不是只要包含这三列的查询都会用到该组合索引。也就是说**name\_city\_age(LOGIN\_NAME(16),CITY,AGE)从左到右进行索引，如果没有左前索引，mysql不会执行索引查询。**

### 8.1.3.2 索引类型

#### 1.直接创建索引和间接创建索引

直接创建索引：CREATE INDEX mycolumn\_index ON mytable (mycolumn)

间接创建索引：定义主键约束或者唯一性键约束，可以间接创建索引

#### 2.普通索引和唯一性索引

普通索引：CREATE INDEX mycolumn\_index ON mytable (mycolumn)

唯一性索引：保证在索引列中的全部数据是唯一的，对聚簇索引和非聚簇索引都可以使用

```
CREATE UNIQUE COUSTERED INDEX mycolumn_cindex ON mytable(mycolumn)
```

普通索引允许被索引的数据列包含重复的值。比如说，因为人有可能同名，所以同一个姓名在同一个“员工个人资料”数据表里可能出现两次或更多次。

如果能确定某个数据列将只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字 UNIQUE 把它定义为一个唯一索引。这么做的好处：一是简化了 MySQL 对这个索引的管理工作，这个索引也因此而变得更有效率；二是 MySQL 会在有新记录插入数据表时，自动检查新记录的这个字段的值是否已经在某个记录的这个字段里出现过了；如果是，MySQL 将拒绝插入那条新记录。也就是说，唯一索引可以保证数据记录的唯一性。事实上，在许多场合，人们创建唯一索引的目的往往不是为了提高访问速度，而只是为了避免数据出现重复。

#### 3.单个索引和复合索引

单个索引：即非复合索引

复合索引：又叫组合索引，在索引建立语句中同时包含多个字段名，最多 16 个字段

```
CREATE INDEX name_index ON username(firstname,lastname)
```

#### 4.聚簇索引和非聚簇索引(聚集索引，群集索引)

<https://www.cnblogs.com/s-b-b/p/8334593.html>

#### MySQL 里主键就是聚集索引

定义：数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引。

| 地址   | id  | username | score |
|------|-----|----------|-------|
| 0x01 | 1   | 小明       | 90    |
| 0x02 | 2   | 小红       | 80    |
| 0x03 | 3   | 小华       | 92    |
| ..   | ..  | ..       | ..    |
| 0xff | 256 | 小英       | 70    |

注：第一列的地址表示该行数据在磁盘中的物理地址，后面三列才是我们SQL里面用的表里的列，其中id是主键，建立了聚集索引。

结合上面的表格就可以理解这句话了吧：数据行的物理顺序与列值的顺序相同，如果我们查询id比较靠后的数据，那么这行数据的地址在磁盘中的物理地址也会比较靠后。而且由于物理排列方式与聚集索引的顺序相同，所以也就只能建立一个聚集索引了。

## 5. 主索引 和外键索引

在前面已经反复多次强调过：必须为主键字段创建一个索引，这个索引就是所谓的“主索引”。

主索引与唯一索引的区别是：前者在定义时使用的关键字是 PRIMARY 而不是 UNIQUE。

主索引：CREATE PRIMARY COUSTERED INDEX mycolumn\_cindex ON mytable(mycolumn) 或者 mysql>ALTER TABLE `table\_name` ADD PRIMARY KEY ( `column` )

外键索引：外键索引 mysql>ALTER TABLE `table\_name` ADD FULLTEXT ( `column` )

## 6. 全文索引

[http://www.360doc.com/content/17/1211/13/33260087\\_712076317.shtml](http://www.360doc.com/content/17/1211/13/33260087_712076317.shtml)

| 文章id | 文章标题  | 文章内容                                         |
|------|-------|----------------------------------------------|
| 1    | 超级塞亚人 | 我是超级塞亚人我喜欢吃苹果，我不是天朝的人，也不是地球人                 |
| 2    | 天朝大国  | 我大天朝威武，我大天朝13亿人，我大天朝                         |
| 3    | 我喜欢游泳 | 游泳有很多好方法                                     |
| 4    | 动画片   | 我儿子喜欢看动画片，尤其是七龙珠，因为里面有塞亚人，而且塞亚人喜欢吃苹果，他们不是地球人 |
| 5    | 运动    | 我喜欢运动，喜欢跑步，喜欢游泳，喜欢健身，喜欢xxoo                  |
| 6    | 打炮    | 我是一个二战的老兵，这是我的回忆录，我最幸福的时光就是在天朝吃着苹果打炮         |
| 7    | 。。。   |                                              |
| 8    | 。。。   |                                              |
| 9    | 。。。   |                                              |

然后，根据以上的文章内容，如果建立了一个索引文件（这里忽略索引文件的数据结构，仅仅以一种易于理解的方式呈现）：

| 关键词 | 文章id    |
|-----|---------|
| 塞亚人 | 1,4     |
| 苹果  | 1,4,6   |
| 天朝  | 1,2,6   |
| 地球  | 1,4     |
| 游泳  | 3,5     |
| 七龙珠 | 4       |
| 喜欢  | 1,4,5,6 |

那么当我想搜索“塞亚人”的时候，这个索引文件直接告诉我在文章id为1和4的文章里有这个词。

这个索引文件就是“全文索引”。

## 7.空间索引

空间索引是指依据空间对象的位置和形状或空间对象之间的某种空间关系按一定的顺序排列的一种数据结构，其中包含空间对象的概要信息，如对象的标识、外接矩形及指向空间对象实体的指针。

区别：它将空间对象按范围划分，每个结点都对应一个区域和一个磁盘页，非叶结点的磁盘页中

数据库从左到右原则

## 8.2 innodb 与 MyISAM 引擎区别

- 主要区别：

- MyISAM是非事务安全型的，而InnoDB是事务安全型的。
- MyISAM锁的粒度是表级，而InnoDB支持行级锁定。
- MyISAM支持全文类型索引，而InnoDB不支持全文索引。
- MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。
- MyISAM表是保存成文件的形式，在跨平台的数据转移中使用MyISAM存储会省去不少的麻烦。
- InnoDB表比MyISAM表更安全，可以在保证数据不会丢失的情况下，切换非事务表到事务表（`alter table tablename type=innodb`）。

- 应用场景：

- MyISAM管理非事务表。它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的SELECT查询，那么MyISAM是更好的选择。
- InnoDB用于事务处理应用程序，具有众多特性，包括ACID事务支持。如果应用中需要执行大量的INSERT或UPDATE操作，则应该使用InnoDB，这样可以提高多用户并发操作的性能。

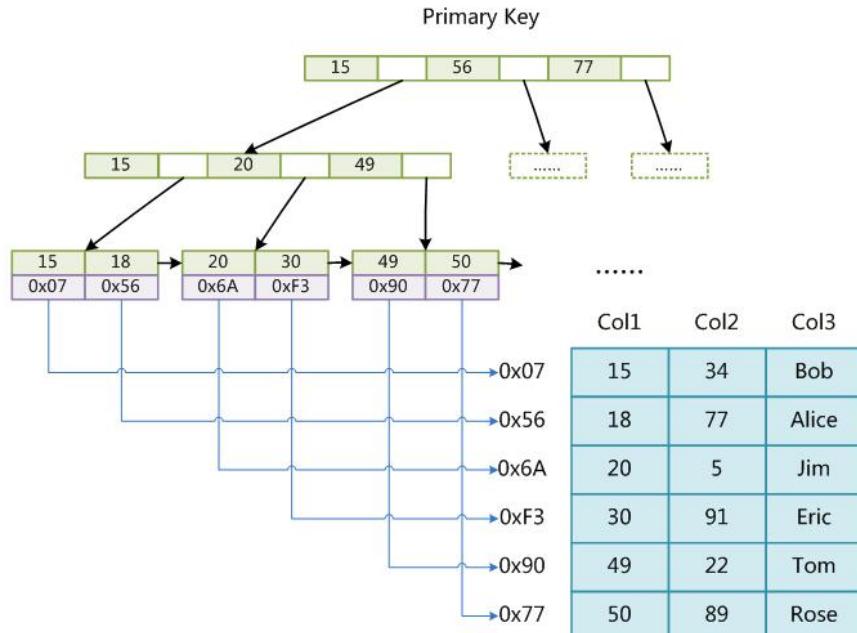
两种存储引擎在索引上的区别：

<https://blog.csdn.net/zsq520520/article/details/68954646>

## 1. MyISAM索引实现：

### 1) 主键索引：

MyISAM引擎使用B+Tree作为索引结构。叶节点的data域存放的是数据记录的地址。下图是MyISAM主键索引的原理图：

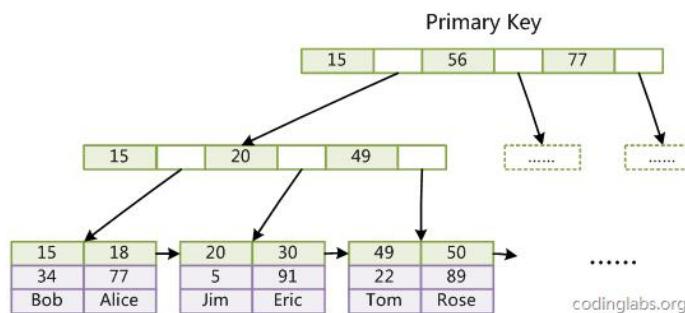


## 2. InnoDB索引实现

然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

### 1) 主键索引：

MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构。这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



(图innodb主键索引 )

(图innodb主键索引)是InnoDB主索引(同时也是数据文件)的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做**聚集索引**。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键( MyISAM可以没有)，如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

## 8.3 事务隔离级别 (恶果：脏读 幻读 不可重复读)

### Mysql 默认级别可重复读

#### READ UNCOMMITTED (未提交读)

在 READ UNCOMMITTED 级别，事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读 (Dirty Read)。这个级别会导致很多问题，从性能上来说，READ UNCOMMITTED 不会比其他的级别好太多，但却缺乏其他级别的很多好处，除非真的有非常必要的理由，在实际应用中一般很少使用。

#### READ COMMITTED (提交读)

大多数数据库系统的默认隔离级别都是 READ COMMITTED (但 MySQL 不是)。READ COMMITTED 满足前面提到的隔离性的简单定义：一个事务开始时，只能“看见”已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。这个级别有时候也叫做不可重复读 (nonrepeatable read)，因为两次执行同样的查询，可能会得到不一样的结果。

#### REPEATABLE READ (可重复读)

REPEATABLE READ 解决了脏读的问题。该级别保证了在同一个事务中多次读取同样记录的结果是一致的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读 (Phantom Read) 的问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另外一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行 (Phantom Row)。InnoDB 和 XtraDB 存储引擎通过多版本并发控制 (MVCC, Multiversion Concurrency Control) 解决了幻读的问题。本章稍后会做进一步的讨论。

可重复读是 MySQL 的默认事务隔离级别。

#### SERIALIZABLE (可串行化)

SERIALIZABLE 是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。简单来说，SERIALIZABLE 会在读取的每一行数据上都加锁，所以可能导致大量的超时和锁争用的问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

| 隔离级别             | 脏读可能性 | 不可重复读可能性 | 幻读可能性 | 加锁读 |
|------------------|-------|----------|-------|-----|
| READ UNCOMMITTED | Yes   | Yes      | Yes   | No  |
| READ COMMITTED   | No    | Yes      | Yes   | No  |
| REPEATABLE READ  | No    | No       | Yes   | No  |
| SERIALIZABLE     | No    | No       | No    | Yes |

## 8.4 数据库特性 ACID

1. 原子性 (Atomicity) : 原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。

2. 一致性 (Consistency) : 从一个一致性状态到另一个一致性状态。

例如现有完整性约束  $a+b=10$ ，如果一个事务改变了  $a$ ，那么必须得改变  $b$ ，使得事务结束后依然满足  $a+b=10$ ，否则事务失败。

3. 隔离性 (Isolation) : 一个事务所做的修改在最终提交以前，对其它事务不可见。

比如现有有个交易是从 A 账户转 100 元至 B 账户，在这个交易还未完成的情况下，如果此时 B 查询自己的账户，是看不到新增加的 100 元的。

4. (Durability) 持久性

持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

## 8.5 sql

### 8.5.1 . Sql 优化

## Sql 优化方法:

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。  
索引并不是越多越好，会降低维护的效率
2. in 和 not in 也要慎用，
3. 避免在字段上进行**计算操作**
4. 临时表也可以用，要记得显示的删除
5. 避免频繁的创建删除数据表
6. **拆分大的 insert update 操作**，提高并发性能

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 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

3.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

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.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。

6.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

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

## 实践中如何优化 MySQL

我当时是按以下四条依次回答的，他们四条从效果上第一条影响最大，后面越来越小。

② SQL 语句及索引的优化

② 数据库表结构的优化

数字类型：非万不得已不要使用 DOUBLE

字符类型：非万不得已不要使用 TEXT 数据类型

时间类型：尽量使用 TIMESTAMP 类型，因为其存储空间只需要 DATETIME 类型的一半。

对于只需要精确到某一天的数据类型，建议使用 DATE 类型，因为他的存储空间只需要 3 个字节，比 TIMESTAMP 还少

VARCHAR2 虽然比 CHAR 节省空间，但是如果一个 VARCHAR2 列经常被修改，而且每次被修改的数据的长度不同，这会引起‘行迁移’(Row Migration)现象，而这造成多余的 I/O，是数据库设计和调整中要尽力避免的，在这种情况下用 CHAR 代替 VARCHAR2 会更好一些。

适度冗余

数据库引擎选择，对于 select 执行比较多的操作，很少插入删除更新的表使用 myisam 引擎

许多 SQL 命令都有一个 **DELAY\_KEY\_WRITE** 项。这个选项的作用是暂时制止 MySQL 在该命令每插入一条新记录和每修改一条现有之后立刻对索引进行刷新，对索引的刷新将等到全部记录插入/修改完毕之后再进行。在需要把许多新记录插入某个数据表的场合，**DELAY\_KEY\_WRITE** 选项的作用将非常明显。[

mysql 表级锁，mysql 什么情况下会触发表锁用索引字段做为条件进行修改时，是否表锁的取决于这个索引字段能否确定记录唯一，当索引值对应记录不唯一，会进行锁表，相反则行锁。

## 8.6 5 种连接 left join、right join、inner join, full join cross join

<https://www.cnblogs.com/pcjim/articles/799302.html>

sql 之 left join、right join、inner join 的区别

**left join**(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录

**right join**(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录

**inner join**(等值连接) 只返回两个表中联结字段相等的行

举例如下：

表 A 记录如下：

| aID | aNum      |
|-----|-----------|
| 1   | a20050111 |
| 2   | a20050112 |
| 3   | a20050113 |
| 4   | a20050114 |
| 5   | a20050115 |

表 B 记录如下：

| bID | bName     |
|-----|-----------|
| 1   | b20050111 |

```
1      2006032401  
2      2006032402  
3      2006032403  
4      2006032404  
8      2006032408
```

---

### 1.left join

sql 语句如下：

```
select * from A  
left join B  
on A.aID = B.bID
```

结果如下：

| aID | aNum      | bID  | bName      |
|-----|-----------|------|------------|
| 1   | a20050111 | 1    | 2006032401 |
| 2   | a20050112 | 2    | 2006032402 |
| 3   | a20050113 | 3    | 2006032403 |
| 4   | a20050114 | 4    | 2006032404 |
| 5   | a20050115 | NULL | NULL       |

(所影响的行数为 5 行)

结果说明：

**left join** 是以 A 表的记录为基础的,A 可以看成左表,B 可以看成右表,**left join** 是以左表为准的.

换句话说,左表(A)的记录将会全部表示出来,而右表(B)只会显示符合搜索条件的记录(例子中为: A.aID = B.bID).

B 表记录不足的地方均为 NULL.

---

### 2.right join

sql 语句如下：

```
select * from A  
right join B  
on A.aID = B.bID
```

结果如下：

| aID  | aNum      | bID | bName      |
|------|-----------|-----|------------|
| 1    | a20050111 | 1   | 2006032401 |
| 2    | a20050112 | 2   | 2006032402 |
| 3    | a20050113 | 3   | 2006032403 |
| 4    | a20050114 | 4   | 2006032404 |
| NULL | NULL      | 8   | 2006032408 |

(所影响的行数为 5 行)

结果说明：

仔细观察一下,就会发现,和 **left join** 的结果刚好相反,这次是以右表(B)为基础的,A 表不足的地方用 NULL 填充.

---

### 3.inner join

sql 语句如下：

```
select * from A  
innerjoin B  
on A.aID = B.bID
```

结果如下：

| aID | aNum      | bID | bName      |
|-----|-----------|-----|------------|
| 1   | a20050111 | 1   | 2006032401 |
| 2   | a20050112 | 2   | 2006032402 |
| 3   | a20050113 | 3   | 2006032403 |
| 4   | a20050114 | 4   | 2006032404 |

结果说明：

很明显,这里只显示出了 A.aID = B.bID 的记录,这说明 inner join 并不以谁为基础,它只显示符合条件的记录.

---

注:

LEFT JOIN 操作用于在任何的 FROM 子句中, 组合来源表的记录。使用 LEFT JOIN 运算来创建一个左边外部联接。左边外部联接将包含了从第一个（左边）开始的两个表中的全部记录，即使在第二个（右边）表中并没有相符值的记录。

语法: FROM table1 LEFT JOIN table2 ON table1.field1 compopr table2.field2

说明: table1, table2 参数用于指定要将记录组合的表的名称。

field1, field2 参数指定被联接的字段的名称。且这些字段必须有相同的数据类型及包含相同类型的数据，但它们不需要有相同的名称。

compopr 参数指定关系比较运算符: "=", "<", ">", "<=", ">=" 或 "<>"。

如果在 INNER JOIN 操作中要联接包含 Memo 数据类型或 OLE Object 数据类型数据的字段，将会发生错误.

### 4.全外连接 ( full join ...on... )

select \* from table1 a full join table2 b on a.id=b.id 全外连接其实是左连接和右

连接的一个合集，也就是说他会查询出左表和右表的全部数据，匹配不上的会显示为 null；

如下图：

### 5.交叉连接 ( cross join... )

```
select * from table1 a crossjoin table2 b ;
```

也可以写为 select \* from table1,table2;

交叉连接，也称为笛卡尔积，查询返回结果的行数等于两个表行数的乘积

## 8.7 数据库范式

- 1NF：符合1NF的关系中的每个属性都不可再分
- 2NF：属性完全依赖于主键 [消除部分子函数依赖]
- 3NF：属性不依赖于其它非主属性 [消除传递依赖]
- BCNF：在1NF基础上，任何非主属性不能对主键子集依赖 [在3NF基础上消除对主码子集的依赖]
- 4NF：要求把同一表内的多对多关系删除。
- 5NF：从最终结构重新建立原始结构。

## 8.8 数据库连接池

### 8.8.1 数据库连接池原理

<https://blog.csdn.net/xiebaochun/article/details/28901363>

<https://blog.csdn.net/shuaihj/article/details/14223015>

连接池的工作原理主要由三部分组成，分别为连接池的建立、连接池中连接的使用管理、连接池的关闭。

第一、连接池的建立。一般在系统初始化时，连接池会根据系统配置建立，并在池中创建了几个连接对象，以便使用时能从连接池中获取。连接池中的连接不能随意创建和关闭，这样避免了连接随意建立和关闭造成的系统开销。Java 中提供了很多容器类可以方便的构建连接池，例如 Vector、Stack 等。

第二、连接池的管理。连接池管理策略是连接池机制的核心，连接池内连接的分配和释放对系统的性能有很大的影响。其管理策略是：

当客户请求数据库连接时，首先查看连接池中是否有空闲连接，如果存在空闲连接，则将连接分配给客户使用；如果没有空闲连接，则查看当前所开的连接数是否已经达到最大连接数，如果没达到就重新创建一个连接给请求的客户；如果达到就按设定的最大等待时间进行等待，如果超出最大等待时间，则抛出异常给客户。当客户释放数据库连接时，先判断该连接的引用次数是否超过了规定值，如果超过就从连接池中删除该连接，否则保留为其他客户服务。

该策略保证了数据库连接的有效复用，避免频繁的建立、释放连接所带来的系统资源开销。

第三、连接池的关闭。当应用程序退出时，关闭连接池中所有的连接，释放连接池相关的资源，该过程正好与创建相反

### 8.8.2 数据库连接池的示例代码

```
public class MyDataSource implements DataSource {  
    //链表 --- 实现栈结构  
    private LinkedList<Connection> dataSources = new LinkedList<Connection>();  
  
    //初始化连接数量  
    public MyDataSource() {  
        //一次性创建10个连接  
        for(int i = 0; i < 10; i++) {  
            try {  
                //1、加载sqlServer 驱动对象  
                DriverManager.registerDriver(new SQLServerDriver());  
                //2、通过JDBC建立数据库连接  
                Connection con = DriverManager.getConnection(  
                    "jdbc:sqlserver://192.168.2.6:1433;DatabaseName=customer", "sa", "123");  
                //3、将连接加入连接池中  
                dataSources.add(con);  
            } catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
  
    @Override  
    public Connection getConnection() throws SQLException {  
        //取出连接池中一个连接  
        final Connection conn = dataSources.removeFirst(); // 删除第一个连接返回  
        return conn;  
    }  
  
    //将连接放回连接池  
    public void releaseConnection(Connection conn) {  
        dataSources.add(conn);  
    }  
}
```

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。

数据库并发和连接数：

Mysql 最大连接数只有 100，要增大，同时客户端 datasource 连接池

**rmoveAbandoned=true** 那么在 **getNumActive()** 快要到 **getMaxActive()** 的时候，系统会进行无效的 Connection 的回收，回收的 Connection 为 **removeAbandonedTimeout**(默认 300 秒) 中设置的秒数后没有使用的 Connection **dataSource.maxWait** 设置多久没使用就超时释放连接

数据库分页查询

数据库中的分页查询语句怎么写 ? <http://qimo601.iteye.com/blog/1634748>

- Mysql的limit用法
  - SELECT \* FROM table LIMIT [offset,] rows | rows OFFSET offset
  - LIMIT 接受一个或两个数字参数。参数必须是一个整数常量。如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1)
- 最基本的分页方式 : SELECT ... FROM ... WHERE ... ORDER BY ... LIMIT ...
- 子查询的分页方式 :

## 8.9 DDL DML DCL

**DML (data manipulation language) :**

它们是 SELECT、UPDATE、INSERT、DELETE，就象它的名字一样，这 4 条命令是用来对数据库里的数据进行操作的语言

**DDL is Data Definition Language statements. Some examples:** 数据定义语言，用于定义和管理 SQL 数据库中的所有对象的语言

- 1.CREATE - to create objects in the database 创建
- 2.ALTER - alters the structure of the database 修改
- 3.DROP - delete objects from the database 删除
- 4.TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

**DCL is Data Control Language statements. Some examples:** 数据控制语言，用来授予或收回访问数据库的某种特权，并控制数据库操纵事务发生的时间及效果，对数据库实行监视等

- 1.COMMIT - save work done 提交
- 2.SAVEPOINT - identify a point in a transaction to which you can later roll back 保存点
- 3.ROLLBACK - restore database to original since the last COMMIT 回滚
- 4.SET TRANSACTION - Change transaction options like what rollback segment to use 设置当前事务的特性，它对后面的事务没有影响。

## 8.10 explain

<https://www.cnblogs.com/gomysql/p/3720123.html>

ml

### 8.10.1.什么是 explain?

需要知道该 SQL 的执行计划，比如是全表扫描，还是索引扫描，这些都需要通过 EXPLAIN 命令去完成。EXPLAIN 命令是查看优化器如何决定执行查询的主要方法。

### 8.10.2 explain 命令详解

执行计划包含的信息

| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
|----|-------------|-------|------|---------------|-----|---------|-----|------|-------|
|    |             |       |      |               |     |         |     |      |       |

#### 1.id

包含一组数字，表示查询中执行 **select** 子句或操作表的顺序（第一个 **select** 是 1，子查询是 2.子子查询是 3.依次类推）

## 2.select\_type

示查询中每个 **select** 子句的类型（简单 OR 复杂）

a. SIMPLE: 查询中不包含子查询或者 UNION

b. 查询中若包含任何复杂的子部分，最外层查询则被标记为： PRIMARY

c. 在 **SELECT** 或 **WHERE** 列表中包含了子查询，该子查询被标记为： SUBQUERY

d. 在 **FROM** 列表中包含的子查询被标记为： DERIVED（衍生）用来表示包含在 **from** 子句中的子查询的 **select**，mysql 会递归执行并将结果放到一个临时表中。服务器内部称为“派生表”，因为该临时表是从子查询中派生出来的

e. 若第二个 **SELECT** 出现在 UNION 之后，则被标记为 UNION；若 UNION 包含在 **FROM** 子句的子查询中，外层 **SELECT** 将被标记为： DERIVED

f. 从 UNION 表获取结果的 **SELECT** 被标记为： UNION RESULT

SUBQUERY 和 UNION 还可以被标记为 DEPENDENT 和 UNCACHEABLE。

DEPENDENT 意味着 **select** 依赖于外层查询中发现的数据。

UNCACHEABLE 意味着 **select** 中的某些 特性阻止结果被缓存于一个 item\_cache 中。

## 3.type

表示 MySQL 在表中找到所需行的方式，又称“访问类型”，常见类型如下：

ALL, index, range, ref, eq\_ref, const, system, NULL（全局遍历啦，还是索引查询等）

## 4.possible\_keys

指出 MySQL 能使用哪个索引在表中找到记录，查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用（表中的某一列上有多个索引都和这个有关系，索引建立在这个字段上了，那么都列出来）

## 5.key

显示 MySQL 在查询中实际使用的索引，若没有使用索引，显示为 NULL

## 6.key\_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度

### 7. ref

表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

### 8. rows

表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

### 9. Extra

包含不适合在其他列中显示但十分重要的额外信息

### 10.table

Select 查询的表的名字

## 8.11 分库分表

[https://blog.csdn.net/winy\\_lm/article/details/50708493](https://blog.csdn.net/winy_lm/article/details/50708493)

### 8.11.1 概念

#### 1. 分表

用户 **id** 直接 **mod** 分成表的数目大小，将大表拆成小表

#### 2. 分库

#### 同分表

#### 3. 分库分表

1. 中间变量 = `user_id % (分库数量 * 每个库的表数量)`
2. 库 = 取整数 (中间变量 / 每个库的表数量)
3. 表 = 中间变量 % 每个库的表数量

还有一种就是分库分表的垂直切分和水平划分 个人感觉这个垂直划分就是上文的分库

##### 1. 垂直切分

将关联的表切分到数据库中（不涉及到切分数据表的操作）。

##### 2. 水平切分

将一个表中的数据切分到不同的数据库中。

## 8.11.2 垂直切分水平切分的坏处

- 垂直拆分会带来如下影响：

1. 单机的 ACID 被打破了。数据到了多机之后，原来在单机通过事务来进行的处理逻辑会受到很大的影响。我们面临的选择是：要么放弃原来的单机事务，修改实现，要么引入分布式事务；

2. 一些 Join 操作会变得比较困难，因为数据可能已经在两个数据库中了，所以不能很方便地利用数据库自身的 Join 了，需要应用或者其他方式解决；

3. 靠外键去进行约束的场景会受到影响。

- 水平拆分会带来如下影响：

1. 同样可能会有 ACID 被打破的情况;
2. 同样有可能有 Join 操作被影响的情况;
3. 靠外键去进行约束的场景会有影响;
4. 依赖单库的自增序列生成唯一 ID 会受影响;
5. 针对单个逻辑意义上的表的查询要跨库了。

### 8.11.3 如何解决分库分表带来的坏处

#### 8.11.3.1 ACID 解决方法

6. **分布式事务问题:** 使用两阶段提交协议。我们在单库上完成相关的数据操作之后，就会直接提交或者回滚，而在分布式系统中，在提交之前增加了准备的阶段，所以称为两阶段提交。事务在第一阶段对资源进行准备，如果在准备阶段有一个资源失败，那么在第二阶段的处理就是回滚所有资源，否则进行 Commit 操作。但是在实际应用当中，由于事务管理器自身的稳定性、可用性的影响，以及网络通信中可能产生的问题，出现的情况会复杂很多。此外，事务管理器在多个资源之间进行协调，它自身要进行很多日志记录的工作。网络上的交互次数的增多以及引入事务管理器的开销，是使用两阶段提交协议使得分布式事务的开销增大的两个方面。因此，在进行垂直拆分和水平拆分后，需要想清楚是否一定要引入两阶段的分布式事务，在必要的情况下才建议使用。

#### 8.11.3.2 水平切分 ID 被破坏（递增 ID）

7. **多机 Sequence 问题:** 当转变为水平分库时，原来单库的 Sequence 以及 ID 的做法需要改变。我们需要从连续

**性和唯一性**两方面来考虑问题。如果只从**唯一性**考虑，我们可以参考 UUID 的生成方式，或者根据自己的业务情况使用各个种子(不同维度的标识，例如 IP、MAC、机器名、时间、本机计数器等因素)来生成唯一的 ID。这样生成的 ID 虽然保证了唯一性，但在整个分布式系统中的连续性不好。接下来看看**连续性**。这里的连续性指的是在整个分布式环境中生成的 ID 的连续性。在单机环境中，其实就是一个单点来完成这个任务，在分布式系统中，我们可以用一个独立的系统来完成这个工作。

这里提供一个解决方案：

我们把所有 ID 集中放在一个地方进行管理，对每个 ID 序列独立管理，每台机器使用 ID 时都从这个 ID 生成器上进行获取。

#### 8.11.3.3 跨库 join

①在应用层把原来数据库中的 Join 操作分成多次数据库操作。举个例子，我们有用户基本信息表，也有用户出售的商品的信息表，需求是查出来登记手机号为 152XXXXXXXX 的用户在售的商品总数。这在单库时用一个 SQL 的 Join 就解决了，而如果商品信息与用户信息分离了，我们就需要现在应用层根据手机号找到用户 id，然后再根据用户 id 找到相关的商品总数。

数据冗余就是分开表以后，在每个表中多添加一些被分走的数据的信息，可能会重复，但是可以在一个库中进行 join(就是把需要的数据就算其它数据库有了，我还是要添加到我的数据库中)

②数据冗余。也就是对一些常用的信息进行冗余，这样就可以把原来需要 Join 的操作变为单表查询。这需要结合具体业务场景。

③借助外部系统(例如搜索引擎)解决一些跨库问题。

#### 8.11.3.4 外键约束解决

2. **外键约束问题：**外键约束的问题比较难解决，不能完全依赖数据库本身来完成之前的功能。如果需要对分库后的单库做外键约束，就需要分库后每个单库的数据是内聚的，否则就只能靠应用层的判断了、容错方式了。

## 8.12 数据库锁

### 8.12.1 封锁

封锁是实现并发控制的一个非常重要的技术。所谓封锁就是事务 T 在对某个数据对象例如表、记录等操作之前，先向系统发出请求，对其加锁。加锁后事务 T 就对该数据对象有了一定的控制，在事务 T 释放它的锁之前，其他事务不能更新此数据对象。例如，事务 T1 要修改 A，若在读出 A 之前先锁住 A，其他事务就不能再读取和修改 A 了，直到 T1 修改并写回 A 后解除了对 A 的封锁为止。这样，就不会丢失 T1 的修改。

确切的控制由封锁的类型决定。基本的封锁类型有两种：排他锁(exclusive locks, 简称 X 锁)和共享锁(share locks, 简称 S 锁)

- **X 锁(排他写锁):** 若事务 T1 对数据对象 A 加上 X 锁，则只允许 T 读取和修改 A，其他任何事物都不能再对 A 加任何类型的锁，直到 T 释放 A 上的锁为止。这就保证了其他事务在 T 释放 A 上的锁之前不能再读取和修改 A；
- **S 锁(共享读锁): 阻塞写锁** 若事务 T 对数据 A 加上 S 锁，则事务 T 可以读 A 但是不能修改 A，其他事务只能对 A 加 S 锁而不能加 X 锁，直到 T 释放 A 上的 S 锁为止。这就保证了其他食物可以读 A，但在 T 释放 A 上的 S 锁之前不能对 A 进行任何修改。

### 8.12.2 封锁协议（解决脏读不可重复读）

- 一级封锁协议：事务 T 在对数据对象 A 进行修改之前，必须对其加 X 锁，直至事务结束才释放。事务结束包括正常结束(COMMIT)和非正常结束(ROLLBACK)；

在一级加锁协议中，如果仅仅是对数据进行读操作而不进行修改，是不需要进行加锁的。所以只能避免修改丢失而不能避免不可重复读和脏读。
- 二级封锁协议：在一级加锁协议的基础上增加事务 T 在读取数据 R 之前必须先对其加 S 锁，读完后即可释放 S 锁；

二级加锁协议除防止了丢失修改，还可进一步**防止读脏数据**。例如：事务 T1 正在对数据对象 R 进行修改，此前已经对 R 加上了 X 锁，此时事务 T2 想读取 R，就必须对 R 加上 S 锁，但是 T2 发现 R 已经被 T1 加上了 X 锁，于是 T2 只能等待 T1 释放了在 R 上加的锁之后才能对 R 加 S 锁并读取。这能防止 T2 读取到 T1 未提交的数据，从而避免了脏读。

但是在二级封锁协议中，由于读完数据后即可释放 S 锁，所以它不能保证可重复读。

- **三级封锁协议：**三级封锁协议是指，在一级封锁协议的基础上增加  
    **事务 T 在读取数据 R 之前对其加 S 锁直至事务结束才释放。**

三级封锁协议除了防止丢失修改和读“脏”数据之外，还进一步防止了不可重复读

### 8.12.3 死锁活锁

- **活锁：**如果事务 T1 封锁了数据 R，事务 T2 又请求封锁 R，于是 T2 等待。T3 也请求封锁 R，当 T1 释放了 R 上的锁之后系统首先批准了 T3 的请求，T2 继续等待；然后 T4 又请求封锁 R，T3 在释放 R 上的锁之后系统又批准了 T4 的请求，T2 有可能永远等待，这就是活锁的情形。

避免活锁的简单方法就是采用先来先服务的策略。当多个事务请求封锁同一数据对象时，封锁子系统按请求锁的先后次序对事务进行排队，数据对象上的锁一旦释放就批准批准申请队列中第一个事务获得锁。

- 死锁：事务 T1 封锁了数据 R1，事务 T2 封锁了数据 R2；同时，事务 T1 请求封锁 R2，因为 T2 已经封锁了 R2，所以 T1 只能等待。T2 也请求封锁 R1，由于 R1 被 T1 封锁了，R2 也只能等待。由于它们互相等待，T1 和 T2 两个事务永远也不能结束，于是就形成了死锁。

### 8.12.3 解决死锁的方法

#### 8.12.3.1 死锁的预防

在数据库中，产生死锁的原因是两个或多个事务都已经封锁了一些数据对象，然后又都请求对已被事务封锁的对象加锁，从而出现死锁。防止死锁的发生其实就是要破坏产生死锁的条件。预防死锁发生通常有以下两种方法。

- 一次封锁法：一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行下去。一次封锁法虽然可以有效防止死锁的发生，但是增加了锁的粒度，从而降低了系统的并发性。并且数据库是不断变化的，所以事先很难精确地确定每个事务所需

进行加锁的对象，为此只能扩大封锁范围，将事务在执行过程中可能需要封锁的数据对象全部加锁，这就进一步降低了并发度；

- 顺序封锁法：顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实施封锁。例如在 B 树结构的索引中，可规定封锁的顺序必须是从根节点开始，然后是下一级的子节点，逐级封锁。顺序封锁法可以有效地避免死锁，但是要实现顺序封锁法十分的困难，因为很难事先确定每一个事务要封锁哪些对象，因此也就很难按规定的顺序去实施加锁。

由此可见数据库中不适合预防死锁，只适合进行死锁的诊断与解除

### 8.12.3.2 死锁的诊断与解除

数据库系统中诊断死锁的方法与操作系统类似，一般使用超时法或事务等待图法

- 超时法：如果一个事务的等待时间超过了规定的时限，那么就认为其发生了死锁。超时法实现简单，但其不足也十分明显，一是有可能误判了死锁，如事务因为其他原因而使等待时间超过时限，系统就会误认为发生了死锁；二是若时限设置得太长，则不能及时发现死锁。
- 事务等待图法：事务等待图是一个有向图  $G=(T,U)$ ,  $T$  为结点的集合，每个结点表示正在运行的事务； $U$  为边的集合，每条边表示事务等待的情况。若  $T_1$  等待  $T_2$ ，则在  $T_1, T_2$  之间画一条有向边，从  $T_1$  指向  $T_2$ 。事务等待图动态地反应了所有事务的等待情况。并发控制

子系统周期性(比如每隔数秒)生成事务等待图，并进行检测。如果发现图中存在回路，则表示系统中出现了死锁。

数据库管理系统的并发控制系统一旦检测到系统中存在死锁，就要设法解除。通常采用的方法是选择一个处理死锁代价最小的事务，将其撤销，释放此事务持有的所有的锁，使其他事务得以继续运行下去。当然，对撤销的事务所进行的数据修改必须加以恢复。

#### 8.12.4 两段锁协议

##### 两段锁协议 申请不释放 释放不申请

- 在对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁；
- 在释放一个锁之后，事务不再申请和获得任何其他封锁。

所谓两段锁的含义是，事务分为两个阶段：第一个阶段是获得封锁，也称为拓展阶段，在这个阶段，事务可以申请获得任何数据项上的任何类型的锁，但是不能释放锁；第二个阶段是释放封锁，也称为收缩阶段，在这个阶段，事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁。

#### 8.12.5 GAP 锁（解决幻读）

在索引记录的间隙上加锁，禁止插入，这样就避免了幻读  
具体例子：

[https://blog.csdn.net/cug\\_jiang126com/article/details/50596729](https://blog.csdn.net/cug_jiang126com/article/details/50596729)

`locking reads, UPDATE` 和 `DELETE` 时，除了对唯一索引的唯一搜索外都会获取 `gap` 锁或 `next-key` 锁。即锁住其扫描的范围。

简单讲就是防止幻读。通过锁阻止特定条件的新记录的插入，因为插入时也要获取 `gap` 锁 (`Insert Intention Locks`)。

更新一条记录不提交如：

```
update tb2 set c1=2 where id=20;
```

执行插入操作，发现[10,30)范围不能插入数据

对于更新操作，仅 20 这条记录不能更新，因为更新操作不会去获取 `gap` 锁。

如果 `SESSION 1` 的表扫描没有用到索引，那么 `gap` 或 `next-key` 锁住的范围是整个表，即任何值都不能插入。

在“`consistent-read`”时即普通的查询，`REPEATABLE READ` 下看到是事务开始时的快照，即使其它事务插入了新行通常也是看不到的，所以在常见的场合可以避免幻读。但是，“`locking read`”或更新、删除时是会看到已提交的修改的，包括新插入的行。

- **快照读：**简单的 `select` 操作，属于快照读，不加锁。(当然，也有例外，下面会分析)
  - `select * from table where ?;`
  - **当前读：**特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。
    - `select * from table where ? lock in share mode;`
    - `select * from table where ? for update;`
    - `insert into table values (...);`
    - `update table set ? where ?;`
    - `delete from table where ?;`
- 所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。

## 8.12.6 next-key 锁

<https://blog.csdn.net/xifeijian/article/details/20313977>

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP）”，InnoDB 也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key 锁）。

举例来说，假如 `emp` 表中只有 101 条记录，其 `empid` 的值分别是 1,2,...,100,101，下面的 SQL：

```
Select * from emp where empid > 100 for update;
```

是一个范围条件的检索，InnoDB 不仅会对符合条件的 `empid` 值为 101 的记录加锁，也会对 `empid` 大于 101（这些记录并不存在）的“间隙”加锁。

InnoDB 使用间隙锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了 `empid` 大于 100 的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另外一方面，是为了满足其恢复和复制的需要。有关其恢复和复制对锁机制的影响，以及不同隔离级别下 InnoDB 使用间隙锁的情况，在后续的章节中会做进一步介绍。

## 8.13 其它问题

## 8.13.1 limit20000 如何优化

<http://uule.iteye.com/blog/2422189>

### 1.子查询优化法

原始 sql 语句: select \* from Member limit 10000,100

先找出第一条数据, 然后大于等于这条数据的 id 就是要获取的数据

缺点: 数据必须是连续的, 可以说不能有 where 条件, where 条件会筛选数据, 导致数据失去连续性

这种:

Sql 代码

```
1. select * from Member where MemberID >= (select MemberID from Member limit 100000,1) limit 100
```

### 2、使用 id 限定优化

原始 sql 语句: select\* from orders\_history where type=2 limit 100000,100.

这种方式假设数据表的 id 是连续递增的, 则我们根据查询的页数和查询的记录数可以算出查询的 id 的范围, 可以使用 id between and 来查询:

Java 代码

```
1. select * from orders_history where type=2 and id between 100000 and 1000100 limit 100;
```

### 3.反向查找优化法

当偏移超过一半记录数的时候, 先用排序, 这样偏移就反转了

缺点: order by 优化比较麻烦, 要增加索引, 索引影响数据的修改效率, 并且要知道总记录数

, 偏移大于数据的一半

正向查找SQL:

Sql代码

```
1.SELECT * FROM `abc` WHERE `BatchID` = 123 LIMIT 1199960, 40  
SELECT * FROM `abc` WHERE `BatchID` = 123 LIMIT 1199960, 40
```

时间 : 2.6493 秒

反向查找sql:

Sql代码

```
1.SELECT * FROM `abc` WHERE `BatchID` = 123 ORDER BY InputDate DESC LIMIT 428775, 40  
SELECT * FROM `abc` WHERE `BatchID` = 123 ORDER BY InputDate DESC LIMIT 428775, 40  
时间 : 1.0035 秒
```

注意, 反向查找的结果是降序desc的, 并且InputDate是记录的插入时间, 也可以用主键联合索引, 但是不方

## 8.13.2 数据库的隔离级别 隔离级别如何实现

(封锁协议 X 锁 S 锁 GAP 锁)

## 8.13.3 char varchar text 区别

<https://blog.csdn.net/wxq1987525/article/details/6564380>

具体对这三种类型的说明不做阐述可以查看 mysql 帮助文档。

### char 的总结:

char 最大长度是 255 字符, 注意是字符数和字符集没关系。可以有默认值, 尾部有空格会被截断。

### varchar 的总结:

varchar 的最大长度 65535 是指能存储的字节数, 其实最多只能存储 65532 个字节, 还

有 3 个字节用于存储长度。注意是**字节数**这个和**字符集**有关系。一个汉字字符用 utf8 占用 3 字节，用 gbk 占用 2 字节。可以有默认值，尾部有空格不会截断。

#### text 的总结：

text 和 varchar 基本相同。text 会忽略指定的大小这和 varchar 有所不同，text 不能有默认值。尾部有空格不会被截断。text 使用额外的 2 个字节来存储数据的大小，varchar 根据存储数据的大小选择用几个字节来存储。text 的 65535 字节全部用来存储数据，varchar 则会占用 1—3 个字节去存储数据大小。

上面所说的一切只针对 mysql，其他数据库可能不同。有不妥的地方请指出

#### 8.13.4 drop delete truncate 区别

- **delete 和 truncate** 只删除表的数据不删除表的结构
- 速度,一般来说: **drop > truncate > delete**
- **delete** 语句是 dml,这个操作会放到 **rollback segment** 中,事务提交之后才生效;  
如果有相应的 trigger,执行的时候将被触发. **truncate,drop** 是 ddl, 操作立即生效,原数据不放到 **rollback segment** 中,不能回滚. 操作不触发 trigger.

**drop** 直接删掉表

**truncate** 删除表中数据，再插入时自增长 id 又从 1 开始

**delete** 删除表中数据，可以加 **where** 字句。

#### 8.13.5 事务

事务（Transaction）是并发控制的基本单位。所谓的事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。

#### 8.13.6 超键、候选键、主键、外键 视图

**超键：**在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。

**候选键：**是最小超键，即没有冗余元素的超键。

**主键：**数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。

**外键：**在一个表中存在的另一个表的主键称此表的外键。

视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，试图通常是有 一个表或者多个表的行或列的子集。对视图的修改不影响基本表。它使得我们获取数据更容易，相比 多表查询。

#### 8.13.7 存储过程与触发器

**存储过程**（Stored Procedure）是一组为了完成特定功能的 SQL 语句集，经编译后存储在数据库中，用户通过指定 存储过程的名字并给定参数（如果该存储过程带有参数）来调用执行它。

什么是触发器？

某个条件成立的时候，你触发器里面所定义的语句就会被自动的执行。因此触发器不需要人为的去调用，也不能调用。

多用于数据同步

触发器有 `after(for)` 和 `instead of` 两类。

`AFTER` 触发器（也叫“FOR”触发器）会在触发 `insert`、`update` 或是 `delete` 动作之后执行  
“Instead of”触发器在执行真正“插入”之前被执行，“Instead of”触发器会替代所要执行的  
SQL 语句

一个表上最多可以有 12 个触发器

好处：相对于外部程序、存储过程，触发器可以更快更高效的维护数据

坏处：（我自己的经验）触发器要用的恰到好处，一个大型应用里，触发器越少越好，触发器会使编程时源码的结构被迫打乱，为将来的程序修改、源码阅读带来很大不便

`Create trigger trigger_name`

```
On {table_name|view_name}  
{After|Instead of} {insert|update|delete}  
as 相应 T-SQL 语句
```

B+树和 B 树的区别 插入节点怎么分裂

有人建议给每张表都建一个自增主键，这样做有什么优点跟缺点

MyISAM、InnoDB 的区别(一个 B+树叶子节点存的地址，一个是直接存的数据)

sql, group by 的理解

对 MySQL 的了解，和 oracle 的区别

mybatis 中 %与\$的区别

一个成绩表求出有不及格科目的同学

500 万数字排序，内存只能容纳 5 万个，如何排序，如何优化？

平时怎么写数据库的模糊查询（由字典树扯到模糊查询，前缀查询，例如“abc%”，还是索引策略的问题）

数据库里有 10000000 条用户信息，需要给每位用户发送信息（必须发送成功），要求节省内存（主键索引、分区技术、异步处理）

数据库连接池（druid）、线程池作用等等

数据库设计与优化能力：

数据库基本知识（存取控制、触发器、存储过程（了解作用）、游标（了解作用）

基本数据库安全

数据库设计，不多讲看看 i 西科和圈子表结构设计（满足三范式等等），多思考。

并发控制（并发数据不一致性、事务隔离级别、乐观锁与悲观锁等）

mysql 锁机制

项目中如何实现事务

数据库设计一般设计成第几范式

mysql 用的什么版本 5.7 跟 5.6 有啥区别 提升 MySQL 安全性

问了一个这样的表(三个字段:姓名, id, 分数)要求查出平均分大于 80 的 id 然后分数降序排序。然后经过提示用聚合函数 avg。

`select id from table group by id having avg(score) > 80 order by avg(score) desc。`

, 为什么 mysql 事务能保证失败回滚 ACID

学生表, 成绩表, 说出查找学生总成绩大于 500 的学生的姓名跟总分

一道算法题, 在一个整形数组中, 找出第三大的数, 注意时间效率

主键索引底层的实现原理? B+树

经典的 01 索引问题?

如何在长文本中快捷的筛选出你的名字? 全文索引

多列索引及最左前缀原则和其他使用场景

数据库的完整性约束, 事务隔离级别, 写一个 SQL 语句, 索引的最左前缀原则

数据库悲观锁怎么实现的

建表的原则

索引的内涵和用法

写怎么创建表

给了两条 SQL 语句, 让根据这两条语句建索引 (个人想法: 主要考虑复合索引只能匹配前缀列的特点)

`select` 语句实现顺序

那么我们来聊一下数据库。A 和 B 两个表做等值连接(`Inner join`) 怎么优化 哈希

数据库连接池的理解和优化

.数据库事物, 什么是事物, 什么情况下会用到事物, 举例说明

Sql 语句。增删改查基本语句, 建表建索引。需要注意细节, 比如 `count` 是否计算 `null` 值的行等。

Sql 语句 分组排序

SQL 语句的 5 个连接概念

- 数据库优化和架构 (主要是主从分离和分库分表相关) 分库分表 跨库 `join` 实现 探讨 主从分离和分库分表相关

数据库中间件

- 跨库 `join`
- 读写分离在中间件的实现
- 限流 `and` 熔断
- 行锁适用场景

# 九.网络

## 9.1.HTTP

### 9.1.1 http 请求报文 & http 响应报文

#### 1.http 请求报文

HTTP 请求报文由请求行 (request line)、请求头部 (header)、空行和请求数据 4 个部分组成。

请求行：(get/post 方法, url 中的 path 路径, http 版本)

请求头部 (header) 关键字/值对组成, 每行一对, 关键字和值用英文冒号

请求数据 body

http 响应报文由状态行, 响应头部, 空行, 响应数据组成

#### 2.http 响应报文

HTTP 响应由四个部分组成：

1.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`。

2.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。

3.空行

4.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。

### 9.1.2 http 报文头部请求头和响应头

键值对 , 例 : [常见的请求头](#)

```
1 Accept: text/html,image/*          -- 浏览器接受的数据类型
2 Accept-Charset: ISO-8859-1        -- 浏览器接受的编码格式
3 Accept-Encoding: gzip,compress     -- 浏览器接受的数据压缩格式
4 Accept-Language: en-us,zh-         -- 浏览器接受的语言
5 Host: www.it315.org:80             -- (必须的) 当前请求访问的目标地址 (主机:端口)
6 If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT      -- 浏览器最后的缓存时间
7 Referer: http://www.it315.org/index.jsp      -- 当前请求来自于哪里 (可用来判断非法链接)
8 User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)      -- 浏览器类型
9 Cookie:name=123                   -- 浏览器保存的cookie信息
10 Connection: close/Keep-Alive       -- 浏览器跟服务器连接状态。close: 连接关闭 keep-alive: 保持连接
11 Date: Tue, 11 Jul 2000 18:23:51 GMT      -- 请求发出的时间
```

键值对，例：常见的响应头

```
1 Location: 重定向地址      #-#表示重定向的地址，该头和302的状态码一起使用。-----  
2 Server:apache tomcat       --表示服务器的类型  
3 Content-Encoding: gzip        --表示服务器发送给浏览器的数据压缩类型  
4 Content-Length: 80          --表示服务器发送给浏览器的实体正文长度  
5 Content-Language: zh-cn       --表示服务器支持的语言  
6 Content-Type: text/html; charset=GB2312  #-#表示服务器发送给浏览器的数据类型 及 内容编码-----  
7 Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT      --表示服务器资源的最后修改时间  
8 Refresh: 秒数;url=地址     #-#表示定时刷新到指定页面-----  
9 Content-Disposition: attachment; filename=aaa.zip    --表示告诉浏览器以下载方式打开资源（下载文件时用到）  
10 Transfer-Encoding: chunked  
11 Set-Cookie:SS=Q0=5Lb_nQ; path=/search           --表示服务器发送给浏览器的cookie信息（会话管理用到）  
12 Expires: -1            --表示通知浏览器不进行缓存  
13 Cache-Control: no-cache      --同上  
14 Pragma: no-cache        --同上  
15 Connection: close/Keep-Alive    --表示服务器和浏览器的连接状态。close: 关闭连接 keep-alive:保存连接
```

### 9.1.3 http 请求方法

#### 1、OPTIONS

返回服务器针对特定资源所支持的 HTTP 请求方法，也可以利用向 web 服务器发送 ‘\*’ 的请求来测试服务器的功能性

#### 2、HEAD

向服务器索与 GET 请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应小消息头中的元信息。

#### 3、GET

向特定的资源发出请求。注意：GET 方法不应当被用于产生“副作用”的操作中，例如在 Web Application 中，其中一个原因是 GET 可能会被网络蜘蛛等随意访问。Loadrunner 中对应 get 请求函数：web\_link 和 web\_url

#### 4、POST

向指定资源提交数据进行处理请求(例如提交表单或者上传文件)。数据被包含在请求体中。POST 请求可能会导致新的资源的建立和/或已有资源的修改。Loadrunner 中对应 POST 请求函数：web\_submit\_data,web\_submit\_form

#### 5、PUT

向指定资源位置上传其最新内容

#### 6、DELETE

请求服务器删除 Request-URL 所标识的资源

#### 7、TRACE

回显服务器收到的请求，主要用于测试或诊断

#### 8、CONNECT

HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。

### 9.1.4 http 请求过程

以下是 HTTP 请求/响应的步骤：

## 1、客户端连接到 Web 服务器

一个 HTTP 客户端，通常是浏览器，与 Web 服务器的 HTTP 端口（默认为 80）建立一个 TCP 套接字连接。例如，<http://www.oakcms.cn>。

## 2、发送 HTTP 请求

通过 TCP 套接字，客户端向 Web 服务器发送一个文本的请求报文，一个请求报文由请求行、请求头部、空行和请求数据 4 部分组成。

## 3、服务器接受请求并返回 HTTP 响应

Web 服务器解析请求，定位请求资源。服务器将资源复本写到 TCP 套接字，由客户端读取。一个响应由状态行、响应头部、空行和响应数据 4 部分组成。

## 4、释放连接 TCP 连接

若 connection 模式为 close，则服务器主动关闭 TCP 连接，客户端被动关闭连接，释放 TCP 连接；若 connection 模式为 keepalive，则该连接会保持一段时间，在该时间内可以继续接收请求；

## 5、客户端浏览器解析 HTML 内容

客户端浏览器首先解析状态行，查看表明请求是否成功的状态代码。然后解析每一个响应头，响应头告知以下为若干字节的 HTML 文档和文档的字符集。客户端浏览器读取响应数据 HTML，根据 HTML 的语法对其进行格式化，并在浏览器窗口中显示。

1.http 协议， ajax ； 协议的划分

6. http 请求流程

**http 长连接设置 短链接**

**http 怎么记住状态**

http 是无状态的怎么来

### 9.1.4 Get 和 Post 区别

- GET 被强制服务器支持
- 浏览器对URL的长度有限制，所以GET请求不能代替POST请求发送大量数据
- 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
- 在以下情况中，请使用 POST 请求：
  1. 无法使用缓存文件（更新服务器上的文件或数据库）
  2. 向服务器发送大量数据（POST 没有数据量限制）
  3. 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠
  4. post比Get安全性更高
- GET 使用 URL 或 Cookie 传参。而 POST 将数据放在 BODY 中。GET 的 URL 会有长度上的限制，则 POST 的数据则可以非常大。POST 比 GET 安全，因为数据在地址栏上不可见

幂等

从定义上看，HTTP 方法的幂等性是指一次和多次请求某一个资源应该具有同样的副作用。

*post* 并不是幂等的。

### 9.1.5 http 状态码

代码 说明

100 (继续) 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分。

101 (切换协议) 请求者已要求服务器切换协议，服务器已确认并准备切换。

2xx (成功)

表示成功处理了请求的状态代码。

代码 说明

200 (成功) 服务器已成功处理了请求。通常，这表示服务器提供了请求的网页。

201 (已创建) 请求成功并且服务器创建了新的资源。

202 (已接受) 服务器已接受请求，但尚未处理。

203 (非授权信息) 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 (无内容) 服务器成功处理了请求，但没有返回任何内容。

205 (重置内容) 服务器成功处理了请求，但没有返回任何内容。

206 (部分内容) 服务器成功处理了部分 GET 请求。

3xx (重定向)

表示要完成请求，需要进一步操作。通常，这些状态代码用来重定向。

#### 代码 说明

300 (多种选择) 针对请求, 服务器可执行多种操作。服务器可根据请求者 (*user agent*) 选择一项操作, 或提供操作列表供请求者选择。

301 (永久移动) 请求的网页已永久移动到新位置。服务器返回此响应 (对 GET 或 HEAD 请求的响应) 时, 会自动将请求者转到新位置。

302 (临时移动) 服务器目前从不同位置的网页响应请求, 但请求者应继续使用原有位置来进行以后的请求。

303 (查看其他位置) 请求者应当对不同的位置使用单独的 GET 请求来检索响应时, 服务器返回此代码。

304 (未修改) 自从上次请求后, 请求的网页未修改过。服务器返回此响应时, 不会返回网页内容。

305 (使用代理) 请求者只能使用代理访问请求的网页。如果服务器返回此响应, 还表示请求者应使用代理。

307 (临时重定向) 服务器目前从不同位置的网页响应请求, 但请求者应继续使用原有位置来进行以后的请求。

#### 4xx (请求错误)

这些状态代码表示请求可能出错, 妨碍了服务器的处理。

#### 代码 说明

400 (错误请求) 服务器不理解请求的语法。

401 (未授权) 请求要求身份验证。对于需要登录的网页, 服务器可能返回此响应。

403 (禁止) 服务器拒绝请求。

404 (未找到) 服务器找不到请求的网页。

405 (方法禁用) 禁用请求中指定的方法。

406 (不接受) 无法使用请求的内容特性响应请求的网页。

407 (需要代理授权) 此状态代码与 401 (未授权) 类似, 但指定请求者应当授权使用代理。

408 (请求超时) 服务器等候请求时发生超时。

409 (冲突) 服务器在完成请求时发生冲突。服务器必须在响应中包含有关冲突的信息。

410 (已删除) 如果请求的资源已永久删除, 服务器就会返回此响应。

411 (需要有效长度) 服务器不接受不含有效内容长度标头字段的请求。

412 (未满足前提条件) 服务器未满足请求者在请求中设置的其中一个前提条件。

413 (请求实体过大) 服务器无法处理请求, 因为请求实体过大, 超出服务器的处理能力。

414 (请求的 URI 过长) 请求的 URI (通常为网址) 过长, 服务器无法处理。

415 (不支持的媒体类型) 请求的格式不受请求页面的支持。

416 (请求范围不符合要求) 如果页面无法提供请求的范围, 则服务器会返回此状态代码。

417 (未满足期望值) 服务器未满足“期望”请求标头字段的要求。

## 5xx (服务器错误)

这些状态代码表示服务器在尝试处理请求时发生内部错误。这些错误可能是服务器本身的问题，而不是请求出错。

### 代码 说明

500 (服务器内部错误) 服务器遇到错误，无法完成请求。

501 (尚未实施) 服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。

502 (错误网关) 服务器作为网关或代理，从上游服务器收到无效响应。

503 (服务不可用) 服务器目前无法使用（由于超载或停机维护）。通常，这只是暂时状态。

504 (网关超时) 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505 (HTTP 版本不受支持) 服务器不支持请求中所用的 HTTP 协议版本。

600 源站没有返回响应头部,只返回实体内容

## 9.1.6 http 长连接 短连接 HTTP 协议是无状态

无状态：HTTP 协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。也就是说，打开一个服务器上的网页和上一次打开这个服务器上的网页之间没有任何联系。HTTP 是一个无状态的面向连接的协议，无状态不代表 HTTP 不能保持 TCP 连接，更不能代表 HTTP 使用的是 UDP 协议（无连接）。

<https://www.cnblogs.com/gotodsp/p/6366163.html>

短连接：在 HTTP/1.0 中默认使用短连接。也就是说，客户端和服务器每进行一次 HTTP 操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个 HTML 或其他类型的 Web 页中含有其他的 Web 资源（如 JavaScript 文件、图像文件、CSS 文件等），每遇到这样一个 Web 资源，浏览器就会重新建立一个 HTTP 会话。

长连接：而从 HTTP/1.1 起，默认使用长连接，用以保持连接特性。使用长连接的 HTTP 协议，会在响应头加入这行代码：

`Connection:keep-alive`

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输 HTTP 数据的 TCP 连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive 不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如 Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP 协议的长连接和短连接，实质上是 TCP 协议的长连接和短连接。

## 什么时候用长连接，短连接？

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个 TCP 连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就 OK 了，不用建立 TCP 连接。例如：数据库的连接用长连接，如果用短连接频繁的通信会造成 socket 错误，而且频繁的 socket

创建也是对资源的浪费。

而像 WEB 网站的 http 服务一般都用**短链接**，因为长连接对于服务端来说会耗费一定的资源，而像 WEB 网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好

## 9.1.7 http1.1 与 http1.0 的区别

- 1.http1.0 需要 `keep-alive` 参数来告知服务器要建立一个长连接，而 http1.1 默认支持长连接
- 2.HTTP 1.1 支持只发送 `header` 信息(不带任何 `body` 信息)，如果服务器认为客户端有权限请求服务器，则返回 100，否则返回 401。客户端如果接受到 100，才开始把请求 `body` 发送到服务器。这样当服务器返回 401 的时候，客户端就可以不用发送请求 `body` 了，节约了带宽。
- 3.host 域 http1.0 没有 host 域，http1.1 才支持这个参数。
- 4.**带宽优化及网络连接的使用**，HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1 则在请求头引入了 `range` 头域，它允许只请求资源的某个部分，即返回码是 206 (Partial Content)，这样就方便了开发者自由的选择以便于充分利用带宽和连接。

## 9.1.8 http2.0 与 http1.0 的区别

<https://www.cnblogs.com/heluan/p/8620312.html>

**新的二进制格式 (Binary Format)**，HTTP1.x 的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。

**多路复用 (MultiPlexing)**，即连接共享，建立起一个连接请求后，可以在一个链接上一直发送，不要等待上一次发送完并且受到回复后才能发送下一个 (http1.0 是这样)，是可以同时发送多个请求，互相并不干扰。

**header 压缩**，如上文中所言，对前面提到过 HTTP1.x 的 `header` 带有大量信息，而且每次都要重发，HTTP2.0 利用 HPACK 对消息头进行压缩传输，客服端和服务器维护一个动态链表（当一个头部没有出现的时候，就插入，已经出现了就用表中的索引值进行替代），将既避免了重复 `header` 的传输，又减小了需要传输的大小。（Hpack <https://www.jianshu.com/p/f44b930cfcac>）

**服务端推送 (server push)**，就是客户端请求 html 的时候，服务器顺带把此 html 需要的 css,js 也一起发送给客服端，而不像 http1.0 中需要请求一次 html，然后再请求一次 css，然后再请求一次 js。

## 9.1.9 转发与重定向的区别

一句话，**转发是服务器行为，重定向是客户端行为**。为什么这样说呢，这就要看两个动作的工作流程：

**转发过程：**客户浏览器发送 http 请求----> web 服务器接受此请求--> 调用内部的一个方法在容器内部完成请求处理和转发动作----> 将目标资源发送给客户；在这里，转发的路径必须是同一个 web 容器下的 url，其不能转向到其他的 web 路径上去，中间传递的是自己的容器内的 request。在客户浏览器路径栏显示的仍然是其第一次访问的路径，也就是说客户是感觉不到服务器做了转发的。转发行为是浏览器只做了一次访问请求。

**重定向过程：**客户浏览器发送 http 请求----> web 服务器接受后发送 302 状态码响应及对应新的 location 给客户浏览器--> 客户浏览器发现是 302 响应，则自动再发送一个新的 http 请求，请求 url 是新的 location 地址----> 服务器根据此请求寻找资源并发送给客户。在这里 location 可以重定向到任意 URL，既然是浏览器重新发出了请求，则就没有什么 request 传递的概念了。在客户浏览器路径栏显示的是其重定向的路径，客户可以观察到地址的变化。重定向行为是浏览器做了至少两次的访问请求。

## 9.2.TCP UDP

<https://blog.csdn.net/oney139/article/details/8103223>

### 9.2.0 TCP 头部

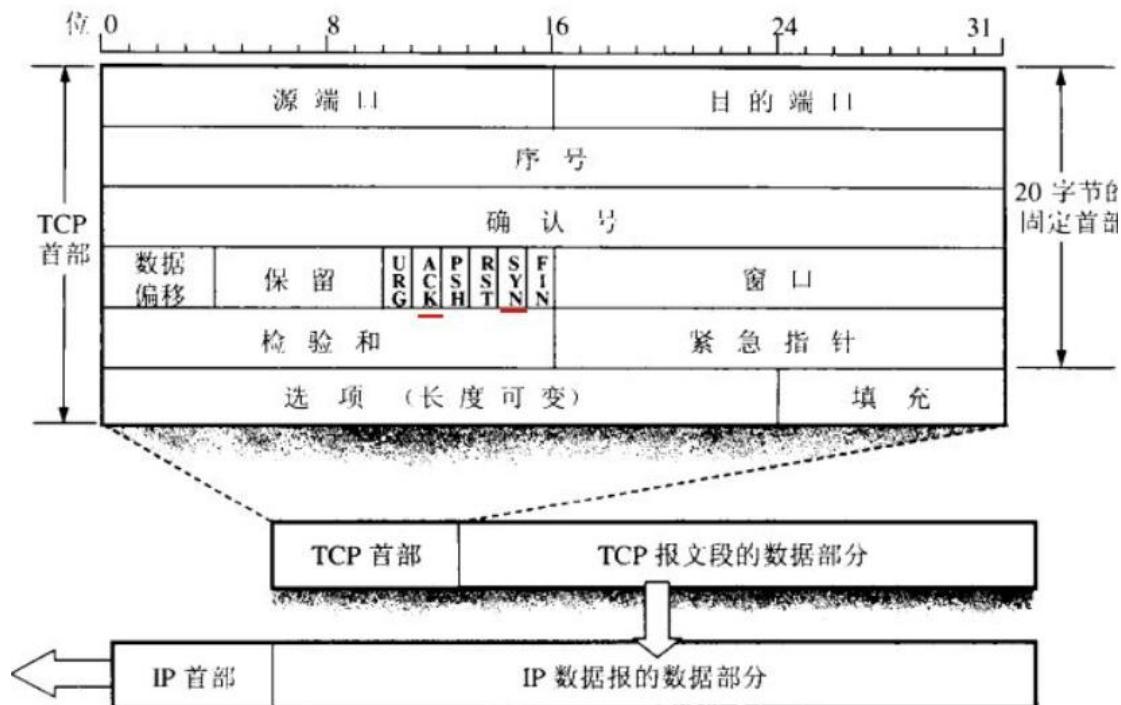


图 5-14 TCP 报文段的头部格式



### Flag : 从左到右 , [URG|ACK|PSH|RST|SYN|FIN]

ACK 设置为 1 表示前面的确认 (ack) 是有效的。

PSH 表示要求对方在接到数据后立即请求递交给应用程序，而不是缓冲起来直到缓冲区收满为止。

RST 用于重置一个已经混乱的连接。

SYN 用于建立连接的过程。

FIN 用来释放一个连接。

**窗口大小**：指定了从被确认的字节算起可以发送多少个字节。要深入理解这个域的含义，可以参看 TCP 拥塞控制和慢启动算法

校验和：校验范围包括 TCP 头、数据报内容和概念性伪头部。概念性伪头部又包括源 IP，目的 IP，TCP 协议号。

URG=1

当 URG 字段被置 1，表示本数据报的数据部分包含紧急信息，此时紧急指针有效。紧急数据一定位于当前数据包数据部分的最前面，紧急指针标明了紧急数据的尾部。如 **control+c**：这个命令要求操作系统立即停止当前进程。此时，这条命令就会存放在数据包数据部分的开头，并由紧急指针标识命令的位置，并 URG 字段被置 1。

PSH=1

当接收方收到 PSH=1 的报文后，会立即将数据交付给应用程序，而不会等到缓冲区满后再提交。一些交互式应用需要这样的功能，降低命令的响应时间。

RST=1

当该值为 1 时，表示当前 TCP 连接出现严重问题，必须要释放重连。

三次握手需要的信息：暂时需要的信息有：

**ACK** : TCP 协议规定, 只有 ACK=1 时有效, 也规定连接建立后所有发送的报文的 ACK 必须为 1。

**SYN(SYNchronization)** : 在连接建立时用来同步序号。当 SYN=1 而 ACK=0 时, 表明这是一个连接请求报文。对方若同意建立连接, 则应在响应报文中使 SYN=1 和 ACK=1. 因此, SYN 置 1 就表示这是一个连接请求或连接接受报文。

**FIN (finis)** 即完, 终结的意思, 用来释放一个连接。当 FIN = 1 时, 表明此报文段的发送方的数据已经发送完毕, 并要求释放连接

### 9.2.1 TCP 与 UDP 区别

|      | TCP                                                                                                                        | UDP                                                                                                                                          |
|------|----------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| 可靠性  | 可靠                                                                                                                         | 不可靠                                                                                                                                          |
| 连接性  | 面向连接                                                                                                                       | 无连接                                                                                                                                          |
| 报文   | 面向字节流                                                                                                                      | 面向报文(保留报文的边界)                                                                                                                                |
| 效率   | 传输效率低                                                                                                                      | 传输效率高                                                                                                                                        |
| 双工性  | 全双工                                                                                                                        | 一对一、一对多、多对一、多对多                                                                                                                              |
| 流量控制 | 有(滑动窗口)                                                                                                                    | 无                                                                                                                                            |
| 拥塞控制 | 有(慢开始、拥塞避免、快重传、快恢复)                                                                                                        | 无                                                                                                                                            |
| 传输速度 | 慢                                                                                                                          | 快                                                                                                                                            |
| 应用场合 | 对效率要求相对低, 但对准确性要求相对高; 或者是要求有连接的场景                                                                                          | 对效率要求相对高, 对准确性要求相对低的场景                                                                                                                       |
| 应用示例 | TCP一般用于文件传输(FTP http 对数据准确性要求高, 速度可以相对慢), 发送或接收邮件(pop imap SMTP 对数据准确性要求高, 非紧急应用), 远程登录(telnet SSH 对数据准确性有一定要求, 有连接的概念)等等; | UDP一般用于即时通信(QQ聊天 对数据准确性和丢包要求比较低, 但速度必须快), 在线视频(rtsp 速度一定要快, 保证视频连续, 但是偶尔花了一个图像帧, 人们还是能接受的), 网络语音电话(VoIP 语音数据包一般比较小, 需要高速发送, 偶尔断音或串音也没有问题)等等。 |

UDP 首部 8 个字节, TCP 首部最低 20 个字节。

**对应的协议不同**

**TCP 对应的协议:**

(1) **FTP**: 定义了文件传输协议, 使用 21 端口。常说某某计算机开了 FTP 服务便是启动了文件传输服务。下载文件, 上传主页, 都要用到 FTP 服务。

(2) **Telnet**: 它是一种用于远程登陆的端口，用户可以以自己的身份远程连接到计算机上，通过这种端口可以提供一种基于 DOS 模式下的通信服务。如以前的 BBS 是纯字符界面的，支持 BBS 的服务器将 23 端口打开，对外提供服务。

(3) **SMTP**: 定义了简单邮件传送协议，现在很多邮件服务器都用的是这个协议，用于发送邮件。如常见的免费邮件服务中用的就是这个邮件服务端口，所以在电子邮件设置中常看到有这么 SMTP 端口设置这个栏，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。通常情况下，POP3 协议所用的是 110 端口。也是说，只要你有相应的使用 POP3 协议的程序（例如 Foxmail 或 Outlook），就可以不以 Web 方式登陆进邮箱界面，直接用邮件程序就可以收到邮件（如是 163 邮箱就没有必要先进入网易网站，再进入自己的邮-箱来收信）。

(5) **HTTP** 协议：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(Trival 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。

## 9.2.2 TCP 三次握手

三次握手的过程：

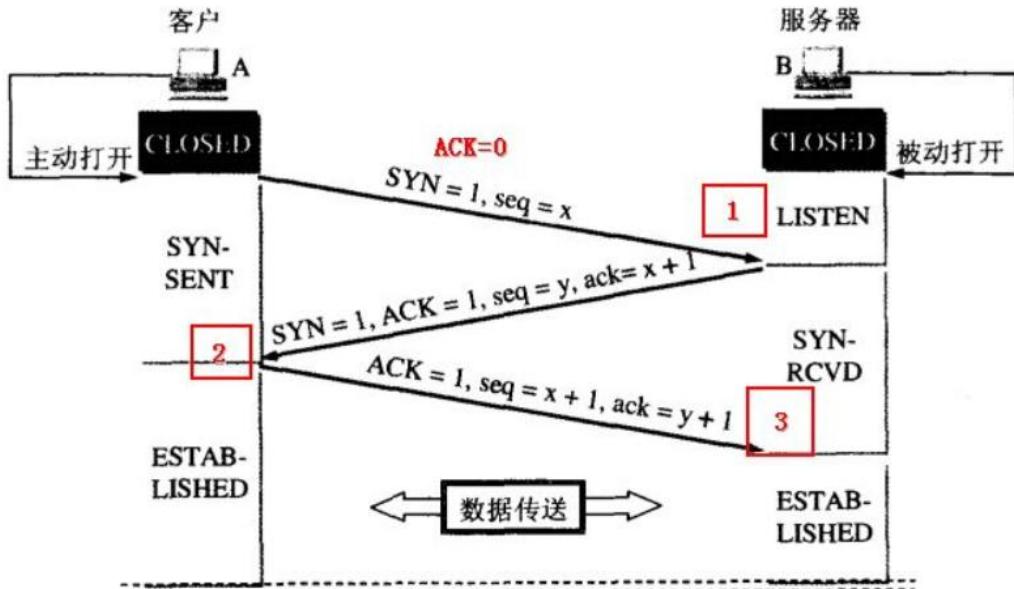


图 5-31 用三次握手建立 TCP 连接

- 首先由 Client 发出请求连接即  $SYN=1\ ACK=0$  (请看头字段的介绍), TCP 规定  $SYN=1$  时不能携带数据, 但要消耗一个序号, 因此声明自己的序号是  $seq=x$
- 然后 Server 进行回复确认, 即  $SYN=1\ ACK=1\ seq=y, ack=x+1$ ,
- 再然后 Client 再进行一次确认, 但不用 SYN 了, 这时即为  $ACK=1, seq=x+1, ack=y+1$ . 然后连接建立, 为什么要进行三次握手呢 (两次确认)。

### 为什么采用三次握手而不是采用两次握手？

为什么 A 还要发送一次确认呢? 这主要是为了防止已失效的连接请求报文段突然又传到了 B, 因而产生错误。

所谓“已失效的连接请求报文段”是这样产生的。考虑一种正常情况。A 发出连接请求, 但因连接请求报文丢失而未收到确认。于是 A 再重传一次连接请求。后来收到了确认, 建立了连接。数据传输完毕后, 就释放了连接。A 共发送了两个连接请求报文段, 其中第一个丢失, 第二个到达了 B。没有“已失效的连接请求报文段”。

现假定出现一种异常情况, 即 A 发出的第一个连接请求报文段并没有丢失, 而是在某些网络结点长时间滞留了, 以致延误到连接释放以后的某个时间才到达 B。本来这是一个早已失效的报文段。但 B 收到此失效的连接请求报文段后, 就误认为是 A 又发出一次新的连接请求。于是就向 A 发出确认报文段, 同意建立连接。假定不采用三次握手, 那么只要 B 发出确认, 新的连接就建立了。

由于现在 A 并没有发出建立连接的请求, 因此不会理睬 B 的确认, 也不会向 B 发送数据。但 B 却以为新的运输连接已经建立了, 并一直等待 A 发来数据。B 的许多资源就这样白白浪费了。

采用三次握手的办法可以防止上述现象的发生。例如在刚才的情况下, A 不会向 B 的确认发出确认。B 由于收不到确认, 就知道 A 并没有要求建立连接。

### 9.2.3 TCP 四次挥手

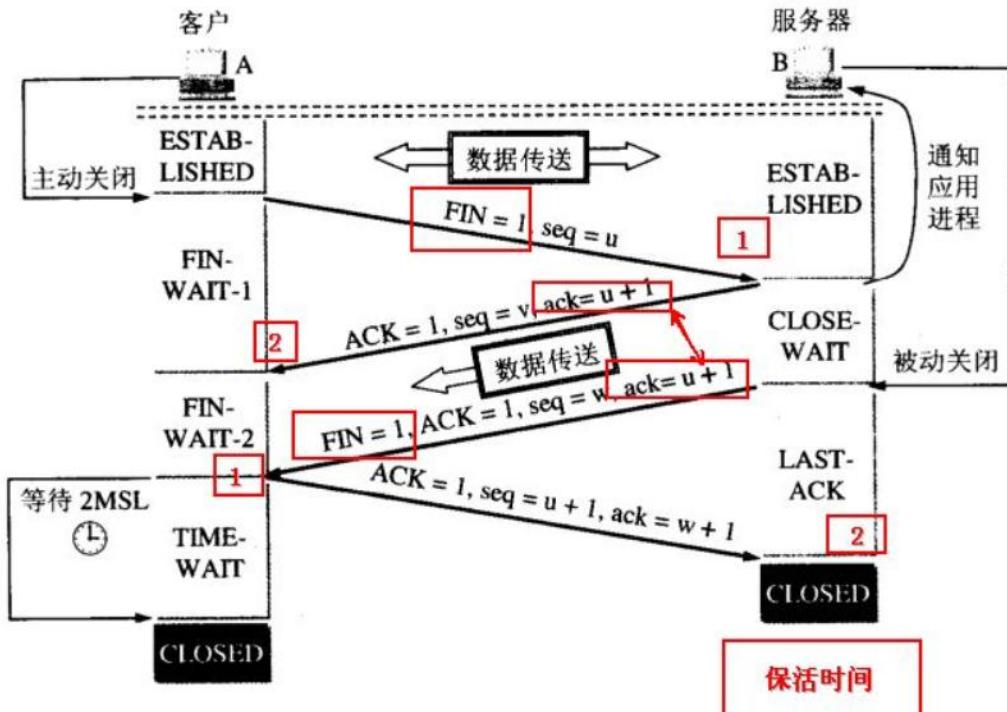


图 5-32 TCP 连接释放的过程

假设 Client 端发起中断连接请求，也就是发送 FIN 报文。Server 端接到 FIN 报文后，意思是说“我 Client 端没有数据要发给你了”，但是如果你还有数据没有发送完成，则不必急着关闭 Socket，可以继续发送数据。所以你先发送 ACK，“告诉 Client 端，你的请求我收到了，但是我还没准备好，请继续你等我的消息”。这个时候 Client 端就进入 FIN\_WAIT 状态，继续等待 Server 端的 FIN 报文。当 Server 端确定数据已发送完成，则向 Client 端发送 FIN 报文，“告诉 Client 端，好了，我这边数据发完了，准备好关闭连接了”。Client 端收到 FIN 报文后，“就知道可以关闭连接了，但是他还是不相信网络，怕 Server 端不知道要关闭，所以发送 ACK 后进入 TIME\_WAIT 状态，如果 Server 端没有收到 ACK 则可以重传。”，Server 端收到 ACK 后，“就知道可以断开连接了”。Client 端等待了 2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，我 Client 端也可以关闭连接了”。

- 为什么是4次挥手：因为TCP有个半关闭状态，假设A.B要释放连接，那么A发送一个释放连接报文给B，B收到后发送确认，这个时候A不发数据，但是B如果发数据A还是要接受，然后B还要发给A连接释放报文，然后A发确认，所以是4次。

TIME\_WAIT 阶段要等待 2 个 MSL 时间才关闭，因为网络原因可能要重发。  
CLOSE\_WAIT 是要等待自己（通常是服务器）把自己传输东西发送完了。

## 9.2.4 tcp 粘包问题 nagle 算法

### tcp 粘包问题

发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（**Nagle 算法**），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。即面向流的通信是无消息保护边界的。

- 对于发送方造成的粘包现象，我们可以通过关闭 Nagle 算法来解决，使用 `TCP_NODELAY` 选项来关闭 Nagle 算法。
- 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
- 包尾加上\r\n 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有\r\n，则会误判为消息的边界。
- 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包体长度，依据包体长度来接收包体。
- 使用更加复杂的应用层协议。

### nagle 算法(tcp 合并小包的算法)

说 tcp 粘包问题要介绍 nagle 算法。**(tcp 中默认开启)**

nagle 算法 如果发送端欲多次发送包含少量字符的数据包，则发送端会先将第一个小包发送出去，而将后面到达的少量字符数据都缓存起来而不立即发送，直到**收到接收端对前一个数据包报文段的 ACK 确认、或当前字符属于紧急数据，或者积攒到了一定数量的数据**（比如缓存的字符数据已经达到数据包报文段的最大长度）等多种情况才将其组成一个较大的数据包发送出去

Nagle 算法在一些场景下的确能提高网络利用率、降低包处理（客户端或服务器）主机资源消耗并且工作得很好，但是在某些场景下却又弊大于利，**Nagle 算法与 ACK 延迟确认的相互作用**

Minshall 对 Nagle 算法所做的改进简而言之就是一句话：在判断当前包是否可发送时，**只需检查最近的一个小包是否已经确认**（其它需要判断的条件，比如包长度是否大于 MSS 等这

些没变，这里假定判断到最后，由此处决定是否发送），如果是，即前面提到的 `tcp_minshall_check(tp)` 函数返回值为假，从而函数 `tcp_nagle_check()` 返回 0，那么表示可以发送（前面图示里的上图），否则延迟等待（前面图示里的下图）。基于的原理很简单，既然发送的小包都已经确认了，也就是说网络上没有当前连接的小包了，所以发送一个即便是比较小的数据包也无关大碍，同时更重要的是，这样做的话，缩短了延迟，提高了带宽利用率。那么对于前面那个例子，由于第一个数据包是大包，所以不管它所对应的 ACK 是否已经收到都不影响对是否发送第二个数据包所做的检查与判断，此时因为所有的小包都已经确认（其实是因为本身就没有发送过小包），所以第二个包可以直接发送而无需等待。

传统 Nagle 算法可以看出是一种包-停-等协议，它在未收到前一个包的确认前不会发送第二个包，除非是“逼不得已”，而改进的 Nagle 算法是一种折中处理，如果未确认的不是小包，那么第二个包可以发送出去，但是它能保证在同一个 RTT 内，网络上只有一个当前连接的小包（因为如果前一个小包未被确认，不会发出第二个小包）；但是，改进的 Nagle 算法在某些特殊情况下反而会出现不利，比如下面这种情况（3 个数据块相继到达，后面暂时也没有其他数据到达），传统 Nagle 算法只有一个小包，而改进的 Nagle 算法会产生 2 个小包（第二个小包是延迟等待超时产生），但这并没有特别大的影响（所以说它是它一种折中处理）：

### 9.2.5 tcp 如何保证可靠性传输

1. 数据包校验：目的是检测数据在传输过程中的任何变化，若校验出包有错，则丢弃报文段并且不给出响应，这时 TCP 发送数据端超时后会重发数据；
2. 对失序数据包重排序：既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。TCP 将对失序数据进行重新排序，然后才交给应用层；
3. 丢弃重复数据：对于重复数据，能够丢弃重复数据；
4. 应答机制：当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒；
5. 超时重发：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段；
6. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间。TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据，这可以防止较快主机致使较慢主机的缓冲区溢出，这就是流量控制。TCP 使用的流量控制协议是可变大小的滑动窗口协议。

**确认和重传：**接收方收到报文就会确认，发送方发送一段时间后没有收到确认就重传。

## 2、数据校验

### 3、数据合理分片和排序：

**UDP：**IP 数据报大于 1500 字节,大于 MTU.这个时候发送方 IP 层就需要分片(fragmentation).把数据报分成若干片,使每一片都小于 MTU.而接收方 IP 层则需要进行数据报的重组.这样就会多做许多事情,而更严重的是,由于 UDP 的特性,当某一片数据传送中丢失时,接收方无法重组数据报.将导致丢弃整个 UDP 数据报.

**tcp** 会按 MTU 合理分片, 接收方会缓存未按序到达的数据, 重新排序后再交给应用层。

**4、流量控制：**当接收方来不及处理发送方的数据, 能提示发送方降低发送的速率, 防止包丢失。

**5、拥塞控制：**当网络拥塞时, 减少数据的发送。发送方让自己的发送窗口等于拥塞窗口  
拥塞控制有哪几种方法?

慢启动

拥塞避免

快重传和快恢复

滑动窗口：安全性考量

<https://coolshell.cn/articles/11564.html>

<https://coolshell.cn/articles/11609.html>

<http://blog.chinaunix.net/uid-26275986-id-4109679.html>

[https://blog.csdn.net/jhh\\_move\\_on/article/details/45770087](https://blog.csdn.net/jhh_move_on/article/details/45770087)

## 9.2.6 TCP 流量控制 拥塞控制

### TCP 流量控制

- 原因：如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失
- 原理：是利用滑动窗口实现的，接收方告诉发送方自己的接收窗口大小，然后发送方发送窗口不能超过接收方给出的接收窗口值

<https://blog.csdn.net/sicofield/article/details/9708383>

拥塞控制

### 9.2.6.1 慢开始与拥塞避免

#### 1. 慢开始

发送方维持一个叫做**拥塞窗口 cwnd (congestion window)** 的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。

慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

这里用报文段的个数的拥塞窗口大小举例说明慢开始算法，实时拥塞窗口大小是以字节为单位的。如下图：

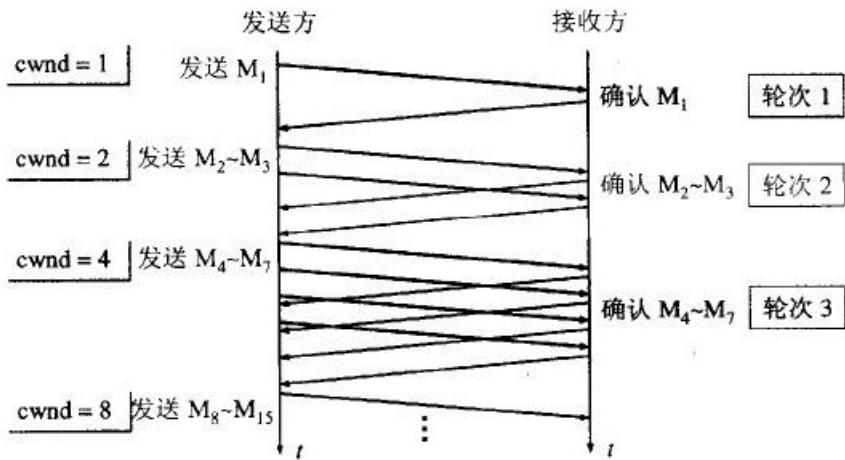


图 5-24 发送方每收到一个确认就把窗口 cwnd 加 1/sicofield

解释一下这张图：就是发送方每收到一个确认就  $cwnd+1$ ，也就是说发送方发送 2 就收到 2 个，所以就是  $cwnd$  就是 4，也就是翻倍成长的道理，每次都是翻倍，也就是指数增长。

为了防止  $cwnd$  增长过大引起网络拥塞，还需设置一个慢开始门限  $ssthresh$  状态变量。 $ssthresh$  的用法如下：

当  $cwnd < ssthresh$  时，使用慢开始算法。

当  $cwnd > ssthresh$  时，改用拥塞避免算法。

当  $cwnd = ssthresh$  时，慢开始与拥塞避免算法任意。

## 2. 拥塞避免

拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口  $cwnd$  加 1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。

无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为 1，执行慢开始算法。如下图：

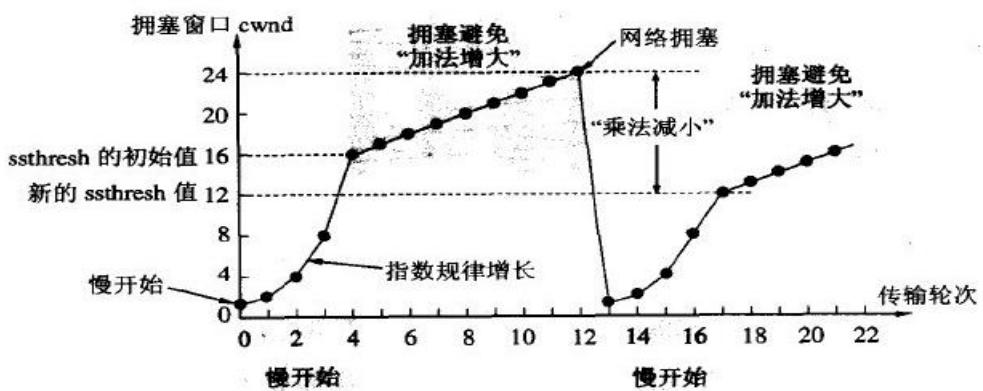


图 5-25 慢开始和拥塞避免算法的实现举例 net/sicofield

### 9.2.6.2 快速重传快速恢复

快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。如下图：

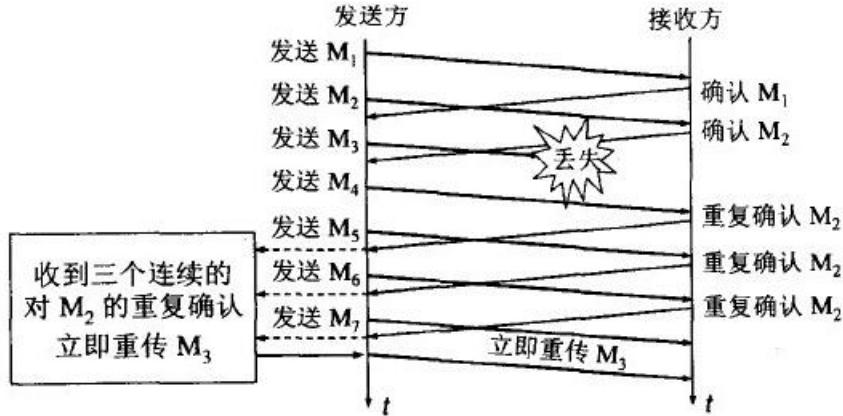


图 5-26 快重传的示意图

快重传配合使用的还有快恢复算法，有以下两个要点：

- ①当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把 ssthresh 门限减半。但是接下去并不执行慢开始算法。
- ②考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将 cwnd 设置为 ssthresh 的大小，然后执行拥塞避免算法。如下图：

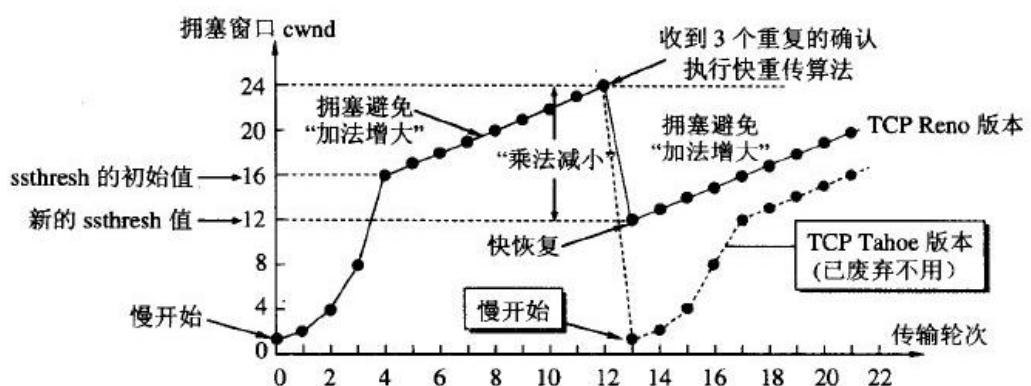
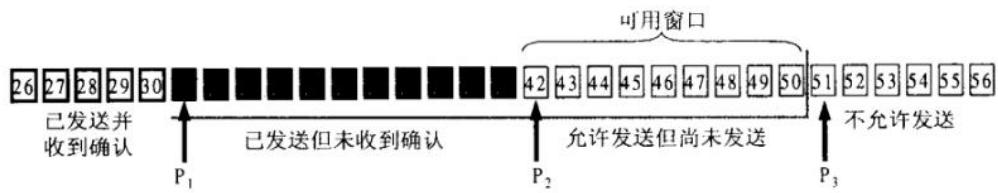


图 5-27 从连续收到三个重复的确认转入拥塞避免

## 9.2.7 滑动窗口机制

<https://blog.csdn.net/GitChat/article/details/78546898>

#### • 发送窗口



发送窗口的大小由接收窗口的剩余大小决定。接收者会把当前接收窗口的剩余大小写入应答TCP报文段的头部，发送者收到应答后根据该值和当前网络拥塞情况设置发送窗口的大小。发送窗口的大小是不断变化的。发送窗口由三个指针构成：

发送者每收到一个应答，后沿就可以向前移动指定的字节。此时若窗口大小仍然没变，前沿也可以向前移动指定字节。当 $p_2$ 和前沿重合时，发送者必须等待确认应答。

- $p_1$

$p_1$ 指向发送窗口的后沿，它后面的字节表示已经发送且已收到应答。

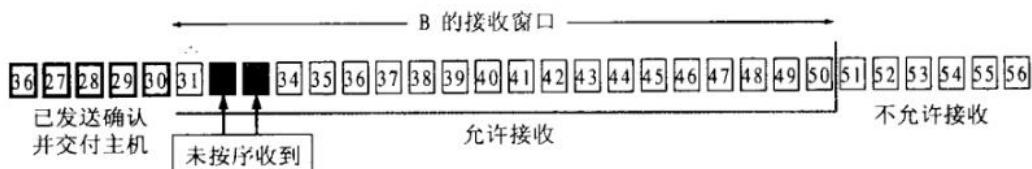
- $p_2$

$p_2$ 指向尚未发送的第一个字节。 $p_1-p_2$ 间的字节表示已经发送，但还没收到确认应答。这部分的字节仍需保留，因为可能还要超时重发。 $p_2-p_3$ 间的字节表示可以发送，但还没有发送的字节。

- $p_3$

$p_3$ 指向发送窗口的前沿，它前面的字节尚未发送，且不允许发送。

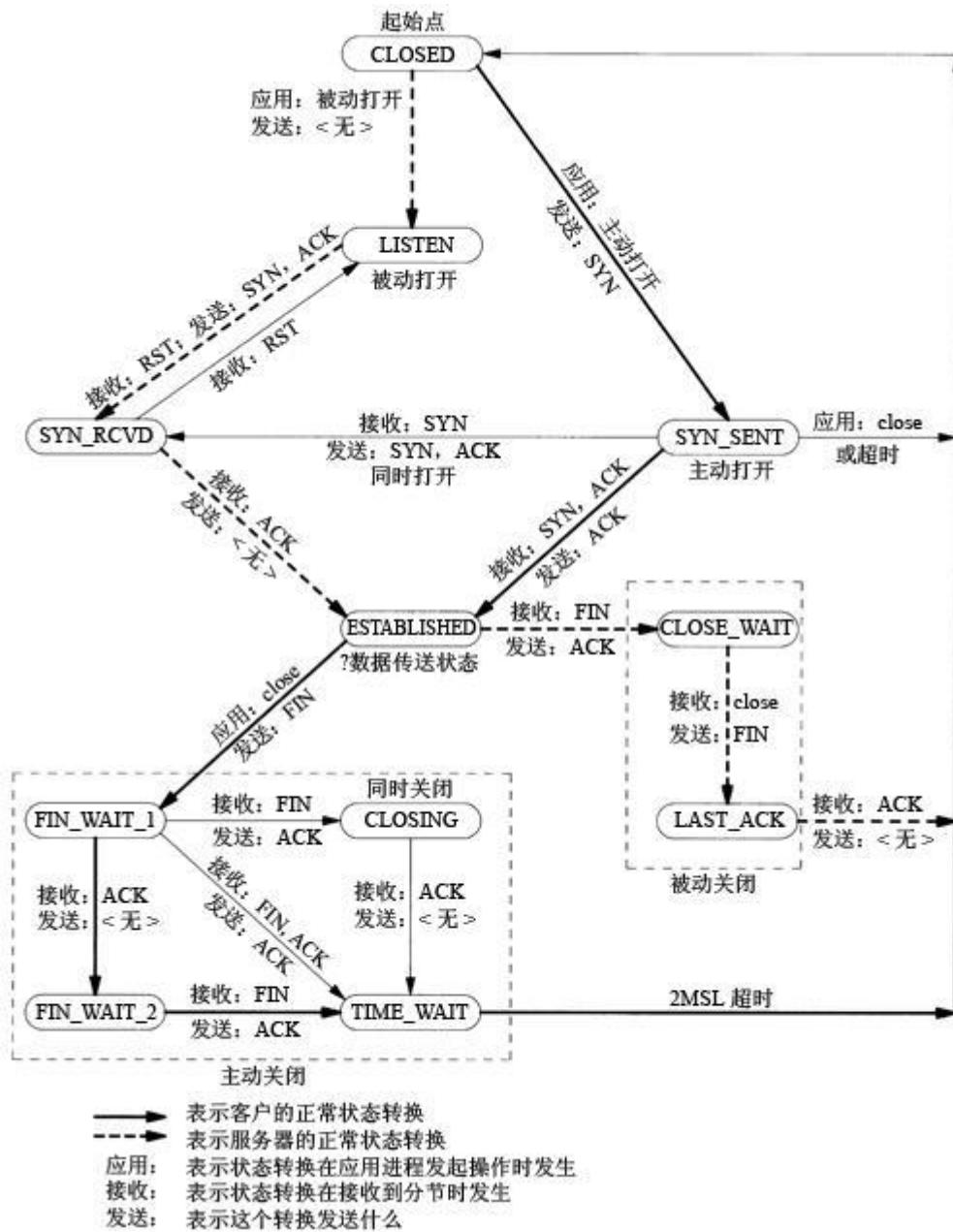
#### • 接收窗口



- 接收者收到的字节会存入接收窗口，接收者会对已经正确接收的有序字节进行累计确认，发送完确认应答后，接收窗口就可以向前移动指定字节。

如果某些字节并未按序收到，接收者只会确认最后一个有序的字节，从而乱序的字节就会被重新发送。

## 9.2.8 TCP 状态转移



1. **CLOSED**: 起始

点，在超时或者连接关闭时候进入此状态。

2. **LISTEN**: svr 端在等待连接过来时候的状态，svr 端为此要调用 socket, bind, listen 函数，就能

进入此状态。此称为应用程序被动打开（等待客户端来连接）。

3. **SYN\_SENT**: 客户端发起连接，发送 SYN 给服务器端。如果服务器端不能连接，则直接进入

CLOSED 状态。

4. **SYN\_RECV**: 跟 3 对应，服务器端接受客户端的 SYN 请求，服务器端由 LISTEN 状态进入

SYN\_RECV 状态。同时服务器端要回应一个 ACK，同时发送一个 SYN 给客户端；另外一种情

况，客户端在发起 SYN 的同时接收到服务器端得 SYN 请求，客户端就会由 SYN\_SENT 到

SYN\_RECV 状态。

5. ESTABLISHED：服务器端和客户端在完成 3 次握手进入状态，说明已经可以开始传输数据了

以上是建立连接时服务器端和客户端产生的状态转移说明。相对来说比较简单明了，如果你

对三次握手比较熟悉，建立连接时的状态转移还是很容易理解。

接下来服务器端和客户端就进行数据传输。。。，当然，里面也大有学问，就此打住，稍

后再表。

下面，我们来看看连接关闭时候的状态转移说明，关闭需要进行 4 次双方的交互，还包括要处

理一些善后工作 (TIME\_WAIT 状态)，注意，这里主动关闭的一方或被动关闭的一方不是指

特指服务器端或者客户端，是相对于谁先发起关闭请求来说的：

6. FIN\_WAIT\_1：主动关闭的一方，由状态 5 进入此状态。具体的动作时发送 FIN 给对方。

7. FIN\_WAIT\_2：主动关闭的一方，接收到对方的 FIN ACK，进入此状态。由此不能再接收对方

的数据。但是能够向对方发送数据。

8. CLOSE\_WAIT：接收到 FIN 以后，被动关闭的一方进入此状态。具体动作时接收到 FIN，同

时发送 ACK。

9. LAST\_ACK：被动关闭的一方，发起关闭请求，由状态 8 进入此状态。具体动作时发送 FIN

给对方，同时在接收到 ACK 时进入 CLOSED 状态。

10. CLOSING：两边同时发起关闭请求时，会由 FIN\_WAIT\_1 进入此状态。具体动作是，接收

到 FIN 请求，同时响应一个 ACK。

11. TIME\_WAIT：最纠结的状态来了。从状态图上可以看出，有 3 个状态可以转化成它，我们一一来分析：

a. 由 FIN\_WAIT\_2 进入此状态：在双方不同时发起 FIN 的情况下，主动关闭的一方在完成自身发

起的关闭请求后，接收到被动关闭一方的 FIN 后进入的状态。

b. 由 CLOSING 状态进入：双方同时发起关闭，都做了发起 FIN 的请求，同时接收到了 FIN 并做了

ACK 的情况下，由 CLOSING 状态进入。

c. 由 FIN\_WAIT\_1 状态进入：同时接受到 FIN (对方发起)，ACK (本身发起的 FIN 回应)，与

b 的区别在于本身发起的 FIN 回应的 ACK 先于对方的 FIN 请求到达，而 b 是 FIN 先

到达。这种情况概率最小。

## 9.2.9 TIME\_WAIT 和 CLOSE\_WAIT

关闭的 4 次连接最难理解的状态是 TIME\_WAIT，存在 TIME\_WAIT 的 2 个理由：

1. 可靠地实现 TCP 全双工连接的终止。
2. 允许老的重复分节在网络中消逝。

<https://blog.csdn.net/wu936754331/article/details/49104497>

<https://blog.csdn.net/u013616945/article/details/77510925>

1. 为什么 time\_wait 需要 2\*MSL 等待时间？

MSL 就是 maximum segment lifetime(最大分节生命期)，这是一个 IP 数据包能在互联网上生存的最长时间，超过这个时间将在网络中消失。

现在我们考虑终止连接时的被动方发送了一个 FIN，然后主动方回复了一个 ACK，然而这个 ACK 可能会丢失，这会造成被动方重发 FIN，这个 FIN 可能会在互联网上存活 MSL。

如果没有 TIME\_WAIT 的话，假设连接 1 已经断开，然而其被动方最后重发的那个 FIN(或者 FIN 之前发送的任何 TCP 分段)还在网络上，然而连接 2 重用了连接 1 的所有的 5 元素(源 IP, 目的 IP, TCP, 源端口, 目的端口)，刚刚将建立好连接，连接 1 迟到的 FIN 到达了，这个 FIN 将以比较低但是确实可能的概率终止掉连接

2. 大量的 time\_wait 如何解决

下面来看一下我们网管对 /etc/sysctl.conf 文件的修改：

```
[plain] view plain copy print ?
1. #对于一个新建连接，内核要发送多少个 SYN 连接请求才决定放弃，不应该大于255，默认值是5，对应于180秒左右时间
2. net.ipv4.tcp_syn_retries=2
3. #net.ipv4.tcp_synack_retries=2
4. #表示当keepalive起用的时候，TCP发送keepalive消息的频度。缺省是2小时，改为300秒
5. net.ipv4.tcp_keepalive_time=1200
6. net.ipv4.tcp_orphan_retries=3
7. #表示如果套接字由本端要求关闭，这个参数决定了它保持在FIN-WAIT-2状态的时间
8. net.ipv4.tcp_fin_timeout=30
9. #表示SYN队列的长度，默认为1024，加大队列长度为8192，可以容纳更多等待连接的网络连接数。
10. net.ipv4.tcp_max_syn_backlog = 4096
11. #表示开启SYN Cookies。当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认为0，表示关闭
12. net.ipv4.tcp_syncookies = 1
13.
14. #表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭
15. net.ipv4.tcp_tw_reuse = 1
16. #表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭
17. net.ipv4.tcp_tw_recycle = 1
18.
19. ##减少超时前的探测次数
20. net.ipv4.tcp_keepalive_probes=5
21. ##优化网络设备接收队列
22. net.core.netdev_max_backlog=3000
```

修改完之后执行`/sbin/sysctl -p`让参数生效。

这里头主要注意到的是`net.ipv4.tcp_tw_reuse`  
`net.ipv4.tcp_tw_recycle`  
`net.ipv4.tcp_fin_timeout`  
`net.ipv4.tcp_keepalive_*`  
这几个参数。

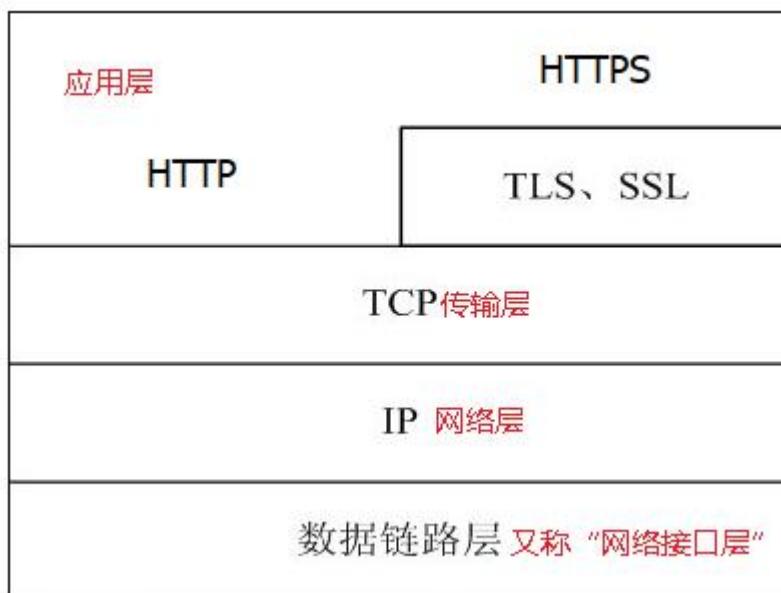
`net.ipv4.tcp_tw_reuse`和`net.ipv4.tcp_tw_recycle`的开启都是为了回收处于`TIME_WAIT`状态的资源。  
`net.ipv4.tcp_fin_timeout`这个时间可以减少在异常情况下服务器从`FIN-WAIT-2`转到`TIME_WAIT`的时间。  
`net.ipv4.tcp_keepalive_*`一系列参数，是用来设置服务器检测连接存活的相关配置。  
关于`keepalive`的用途可以参考：<http://hi.baidu.com/tantea/blog/item/580b9d0218f981793812bb7b.html>

## 2. 当一个 tcp 监听了 80 端口后，Udp 还能否监听 80 端口

答：由于 TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的两个软件模块，因此各自的端口号也相互独立，如 TCP 有一个 255 号端口，UDP 也可以有一个 255 号端口，二者并不冲突。

- 
- 9.tcp 的滑动窗口机制 如何保证不重复发送的
- 10.描述 TCP 滑动窗口机制，如何实现流控
- 11.防止 xxs 攻击和 sql 攻击等等
- 12.TCP/IP 有几层，每层有何含义

<https://blog.csdn.net/xieyutian1990/article/details/23789871>



## 3. 为什么 `TIME_WAIT` 状态还需要等 $2^*MSL$ (Max SegmentLifetime, 最大分段生存期) 秒之后才能返回到 `CLOSED` 状态呢？

因为虽然双方都同意关闭连接了，而且握手的 4 个报文也都发送完毕，按理可以直接回到 `CLOSED` 状态（就好比从 `SYN_SENT` 状态到 `ESTABLISH` 状态那样），但是我们必须假想网络是不可靠的，你无法保证你最后发送的 `ACK` 报文一定会被对方收到，就是说对方处于 `LAST_ACK` 状态下的 `SOCKET` 可能会因为超时未收到 `ACK` 报文，而重发 `FIN` 报文，所以这个 `TIME_WAIT` 状态的作用就是用来重发可能丢失的 `ACK` 报文。

13. tcp 包可以被篡改吗？

## 9.7.计算机网络分层模型

### 9.7.1 osi 七层

**osi**七层协议，每层功能，有哪些协议

#### 7.应用层

- 对应用程序的通信服务
- telnet，HTTP,FTP,NFS,SMTP等。

#### 6.表示层

- 定义数据格式及加密。
- 加密，ASCII等。

#### 5.会话层

- 定义如何开始、控制和结束一个会话（包括对多个双向消息的控制和管理）
- RPC，SQL

#### 4.传输层

- 决定选择差错恢复协议还是无差错恢复协议；在同一主机上对不同应用的数据流的输入进行复用；对收到的顺序不对的数据包的重新排序。
- TCP，UDP，SPX。

#### 3.网络层

- 定义了端到端的包传输，还定义了逻辑地址，恩。。路由实现的方式和学习的方式。
- IP,ICMP，ARP,RARP,IPX等。

#### 2.数据链路层

研究单个链路上如何传输数据。与各种介质有关 ATM(异步传输模式), FDDI(光纤分布式数据接口)。

## 1. 物理层

- 定义了有关传输介质的特性标准
- RJ45, 802.3等。

| OSI七层网络模型            | TCP/IP四层概念模型 | 对应网络协议                                  |
|----------------------|--------------|-----------------------------------------|
| 应用层 ( Application )  | 应用层          | HTTP, TFTP, FTP, NFS, WAIS, SMTP        |
| 表示层 ( Presentation ) |              | Telnet, Rlogin, SNMP, Gopher            |
| 会话层 ( Session )      |              | SMTP, DNS                               |
| 传输层 ( Transport )    | 传输层          | TCP, UDP                                |
| 网络层 ( Network )      | 网络层          | IP, ICMP, ARP, RARP, AKP, UUCP          |
| 数据链路层 ( Data Link )  | 数据链路层        | FDDI, Ethernet, Arpanet, PDN, SLIP, PPP |
| 物理层 ( Physical )     |              | IEEE 802.1A, IEEE 802.2到IEEE 802.11     |

### 9.7.2 ARP

(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。

### 9.7.3 ICMP 协议

ICMP 是 Internet Control Message Protocol，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。控制消息是指网络通不通、主机是否可达、路由器是否可用等网络本身的消息。这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用。ICMP 报文有两种：差错报告报文和询问报文。

#### 9.7.4 DNCP 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。通常被应用在大型的局域网络环境中，主要作用是集中的管理、分配 IP 地址，使网络环境中的主机动态的获得 IP 地址、Gateway 地址、DNS 服务器地址等信息，并能够提升地址的使用率。

#### 9.7.5 RARP 协议

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。

#### 9.7.6 路由选择协议 OSPF RIP

##### RIP：(距离向量路由)

#### 距离向量路由：节点行为

- 节点  $x$  维护一个距离向量  $D_x = \{d(x, y) \mid y \in N\}$ 
  - $x$  知道与每个邻居  $n$  间距离  $d(x, n)$ ，估计与其他  $y$  间距离  $d(x, y)$
- 与所有直接邻居  $n$  间交换距离向量，更新距离向量  $D_x$ ：
$$d(x, y) = \min[d(x, n) + d(n, y), d(x, y)]$$
- 若距离减小，则更新路由表中  $y$  的下一跳为  $n$

<https://blog.csdn.net/xuzhiwangray/article/details/50502233>

**RIP: 1.A** 收到附近节点 **C** 的路由表信息，然后将附近节点 **C** 作为下一跳，那么 **A** 的距离就得在 **C** 上加 1

## 2. 然后合并新的路由节点信息，合并过程：

- (1) 无新信息，不改变
- (2) 新的项目，直接添加
- (3) 相同的下一跳，更新
- (4) 不同的下一跳，距离更短则更新距离和下一跳地址，否则不变

## OSPF：(链路状态路由)

### 链路状态路由：Dijkstra算法

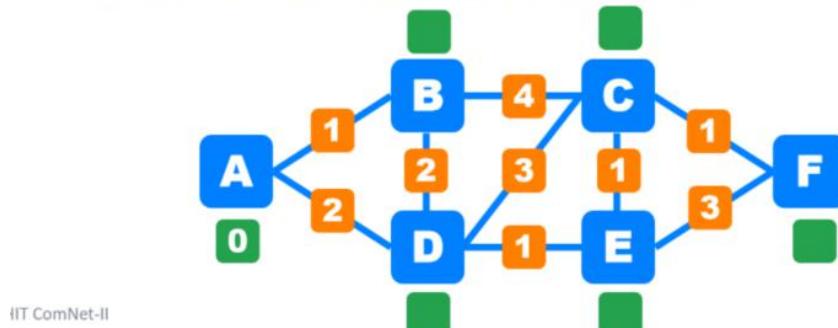
一句话：贪心访问距离源节点S最近的尚未被访问节点

1. 所有节点标记为“未访问”，(除S外)与S距离为无穷大
2. 访问未访问与S最近节点V：遍历V的邻居U，更新U与S间距离

$$d(S, U) = \min[ d(S, V) + w(V, U), d(S, U) ]$$

若距离更新（减少），记录V是U在树上的父节点

3. 重复步骤2，直到所有节点都被访问过
4. 得到一棵以S为根的最短路径树（有向无环图）



17

### 9.7.7 SNMP

简单网络管理协议（SNMP），由一组网络管理的标准组成，包含一个[应用层](#)协议（application layer protocol）、[数据库模型](#)（database schema）和一组资源对象。该协议能够支持[网络管理系统](#)，用以监测连接到网络上的设备是否有任何引起管理上关注的情况。

### 9.7.8 SMTP

SMTP（Simple Mail Transfer Protocol）即[简单邮件传输协议](#)，它是一组用于由源地址到目的地址传送[邮件](#)的规则，由它来控制信件的中转方式。

## 9.9 IP

### 1. IP 报文



分组 id 同一个数据包分片后相同

标记 3 位 第一位没有使用，第二位 DF(Don't Fragment) 设置成 1 表示这个数据包不能被分割，第三个 MF(MoreFragment)，如果一个数据包被分割了，那么除了最后一个分段以外的所有分段都必须设置为 1，用来表示后面还有更多的分段没有到达，最后一个设置为 0，用来表示分割的段全部到达。

段偏移量，这个域有 13bit，也就是每一个数据报最多有 8192 个分段。

### 2. IP 地址类别

| 地址类型 | 特征              | 介绍                                                                                                                                  |
|------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------|
| A类地址 | 第一位为 0，注意是位     | 1、第1字节为网络地址，其它3个字节为主机地址<br>2、地址范围：1.0.0.1—126.255.255.254<br>3、10.X.X.X是私有地址，范围从10.0.0.0—10.255.255.255<br>4、127.X.X.X是保留地址，用做环回测试。 |
| B类地址 | 前两位为 10，注意是位    | 1、第1字节和第2字节为网络地址，后2个字节为主机地址<br>2、地址范围：128.0.0.1—191.255.255.254<br>3、私有地址范围：172.16.0.0—172.31.255.255<br>4、保留地址：169.254.X.X         |
| C类地址 | 前三位为 110，注意是位   | 1、前三个字节为网络地址，最后字节为主机地址<br>2、地址范围：192.0.0.1—223.255.255.254<br>3、私有地址：192.168.X.X，范围从192.168.0.0—192.168.255.255                     |
| D类地址 | 前四位为 1110，注意是位  | 1、不分网络地址和主机地址。<br>2、地址范围：224.0.0.1—239.255.255.254                                                                                  |
| E类地址 | 前五位为 11110，注意是位 | 1、不分网络地址和主机地址<br>2、地址范围：240.0.0.1—255.255.255.254                                                                                   |

### 3. 特殊的地址

ip 地址 = 网络号 + 主机号

| 网络号    | 主机号     | 用途                              |
|--------|---------|---------------------------------|
| 全为0    | 全为0     | 表示本主机，用作源地址，启动时用，获取了 IP 地址后不再使用 |
| 全为0    | host-id | 本地网络上主机号为 host-id 的主机，只作为源地址    |
| 全为1    | 全为1     | 有限广播（本地网络），只作目的地址，各路由器都不转发      |
| net-id | 全为1     | 走向广播（net-id标识的网络），只作目的地址        |
| net-id | 全为0     | 标识一个网络                          |
| 127    | 任意      | 本地软件回送地址                        |

B类中169.254.0.0到169.254.255.255是保留地址。如果你的IP地址是自动获取IP地址，而你在网络上又没有找到可用的DHCP服务器，这时你将会从169.254.0.0到169.254.255.255中获得一个IP地址。

#### 4. 私有地址

私有地址包括3组

A类 : 10.0.0.0~10.255.255.255

B类 : 172.16.0.0~172.31.255.255

C类 : 192.168.0.0~192.168.255.255

学校里的局域网，因为大概有几十万人，所以是10开头的局域网。

而我们买的无线路由器，也要设置局域网，一般为192开头的，比如192.168.0.1或者192.168.199.1（比如极路由就是这个地址）

我们建企业网（单位网络）时，一般是使用私有地址来分配内部主机，小企业使用C类的192.168.0.0网络，中型企业使用172.16.0.0网络，如果还不够用，还有10.0.0.0网络。

## 9.10 网络攻击

### 1.SYN Flood 攻击

关于 SYN Flood 攻击。一些恶意的人就为此制造了 SYN Flood 攻击——给服务器发了一个 SYN 后，就下线了，于是服务器需要默认等 63s 才会断开连接，这样，攻击者就可以把服务器的 syn 连接的队列耗尽，让正常的连接请求不能处理。于是，Linux 下给了一个叫 `tcp_syncookies` 的参数来应对这个事——当 SYN 队列满了后，TCP 会通过源地址端口、目标地址端口和时间戳打造出一个特别的 Sequence Number 发回去（又叫 cookie），如果是攻击者则不会有响应，如果是正常连接，则会把这个 SYN Cookie 发回来，然后服务端可以通过 cookie 建连接（即使你不在 SYN 队列中）。请注意，请先千万别用 `tcp_syncookies` 来处理正常的大负载的连接的情况。因为，`syncookies` 是妥协版的 TCP 协议，并不严谨。对于正常的请求，你应该调整三个 TCP 参数可供你选择，第一个是：`tcp_synack_retries` 可以用他来减少重试次数；第二个是：`tcp_max_syn_backlog`，可以增大 SYN 连接数；第三个是：`tcp_abort_on_overflow` 处理不过来干脆就直接拒绝连接了。

### 2. DDOS 攻击

DDoS 攻击是 Distributed Denial of Service 的缩写，即不法黑客组织通过控制服务器等资源，发动对包括国家骨干网络、重要网络设施、政企或个人网站在内的互联网上任一目标的攻击，致使目标服务器断网，最终停止提供服务。

预防:1.高防服务器 主要是指能独立硬防御 50Gbps 以上的服务器，能够帮助网站拒绝服务攻击，定期扫描网络主节点等 2.DDoS 清洗会对用户请求数据进行实时监控，及时发现 DOS 攻击等异常流量，在不影响正常业务开展的情况下清洗掉这些异常流量。3.CDN 加速 在现实中，CDN 服务将网站访问流量分配到了各个节点中，这样一方面隐藏网站的真实 IP，另一方面即使遭遇 DDoS 攻击，也可以将流量分散到各个节点中，防止源站崩溃。

### 3.DNS 欺骗

DNS 欺骗就是攻击者冒充 域名服务器 的一种欺骗行为

预防：1.使用入侵检测系统 2.使用 DNSSEC

### 4. 重放攻击

重放攻击又称重播攻击、回放攻击，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。

预防：1.加随机数 2.加时间戳

### 5.SQL 注入

所谓 SQL 注入，就是通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。

**预防：**1. **加密处理** 将用户登录名称、密码等数据加密保存。加密用户输入的数据，然后将它与数据库中保存的数据比较，这相当于对用户输入的数据进行了“消毒”处理，用户输入的数据不再对数据库有任何特殊的意义，从而也就防止了攻击者注入 SQL 命令。

2. **确保数据库安全** 只给访问数据库的 web 应用功能所需的最低的权限，撤销不必要的公共许可  
3. **输入验证** 检查用户输入的合法性，确信输入的内容只包含合法的数据。数据检查应当在客户端和服务器端都执行之所以要执行服务器端验证，是为了弥补客户端验证机制脆弱的安全性。

如果客户端不断的发送请求连接会怎样？

- 服务器端回为每个请求创建一个链接，然后向client端发送创建链接时的回复，然后进行等待客户端发送第三次握手数据包，这样会白白浪费资源
- DDos攻击
  - 简单的说就是想服务器发送链接请求，首先进行
  - 第一步：客户端向服务器端发送连接请求数据包（1）
  - 第二步：服务器向客户端回复连接请求数据包（2），然后服务器等待客户端发送tcp/ip链接的第三步数据包（3）
  - 第三步：如果客户端不向服务器端发送最后一个数据包（3），则服务器须等待30s到2min中才能将此链接进行关闭。当大量的请求只进行到第二步，而不进行第三步，服务器又大量的资源等待第三个数据包。则造成DDos攻击。
- DDos预防(没有根治的办法，除非不用TCP/IP链接)。
  - 确保服务器的系统文件是最新版本，并及时更新系统补丁
  - 关闭不必要的服务
  - 限制同时打开SYN的半连接数目
  - 缩短SYN半连接的time out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
  - 启用防火墙的防DDos的属性
  - 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。只要日志出现漏洞或是时间变更，那这台机器就可能遭到了攻击。
  - 限制在防火墙外与网络文件共享。这样会给黑客截取系统文件的机会，主机的信息暴露给黑客，无疑是给了对方入侵的机会。

## 9.11 DNS 浏览器中输入 URL 到页面加载的发生了什么

<https://blog.csdn.net/dojiangv/article/details/51794535>

### 1. DNS 的解析流程

- 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
  - 读取本地的HOST文件
  - 发起一个DNS的系统调用
    - 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
- TCP/IP连接建立起来后，浏览器就可以向服务器发送HTTP请求了
- 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，如果遇到引用的外部JS，CSS,图片等静态资源，它们同样也是一个一个的HTTP请求，都需要经过上面的步骤

### CDN

其中在 DNS 通过域名解析成 IP 地址的过程中，会涉及到 DNS 重定向的问题，

**定义：** CDN，英文 Content Delivery Network，中文翻译是内容分发网络，  
目的就是通过现有的 Internet 中增加一个新的网络架构，将网站内容发布到离用户最近的网络“边缘”，提高用户访问网站的速度，所以更像是增加了一层 CACHE（缓存）层

**功能：** 当用户访问加入 CDN 服务的网站时，域名解析请求将最终交给全局负载均衡 DNS 进行处理。全局负载均衡 DNS 通过一组预先定义好的策略，将当时最接近用户的节点地址提供给用户，使用户能够得到快速的服务。

**组成：** 每个 CDN 节点由两部分组成：负载均衡设备和高速缓存服务器

负载均衡设备负责每个节点中各个 Cache 的负载均衡，保证节点的工作效率；同时，负载均衡设备还负责收集节点与周围环境的信息，保持与全局负载 DNS 的通信，实现整个系统的负载均衡。

高速缓存服务器（Cache）负责存储客户网站的大量信息，就像一个靠近用户的网站服务器一样响应本地用户的访问请求。

## 2. 在浏览器中输入 **www.baidu.com** 后执行的全部过程

1. 客户端浏览器通过 DNS 解析到 www.baidu.com 的 IP 地址为 220.181.0.1，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 220.181.0.1，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，如服务器端用 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，与客户端进行交换，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和传输层的东西，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。

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的 ip 地址和 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。

DNS 均衡

## 9.12 https ssl

### 9.12.1 什么是 https

HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

### 9.12.2 https 与 http 区别

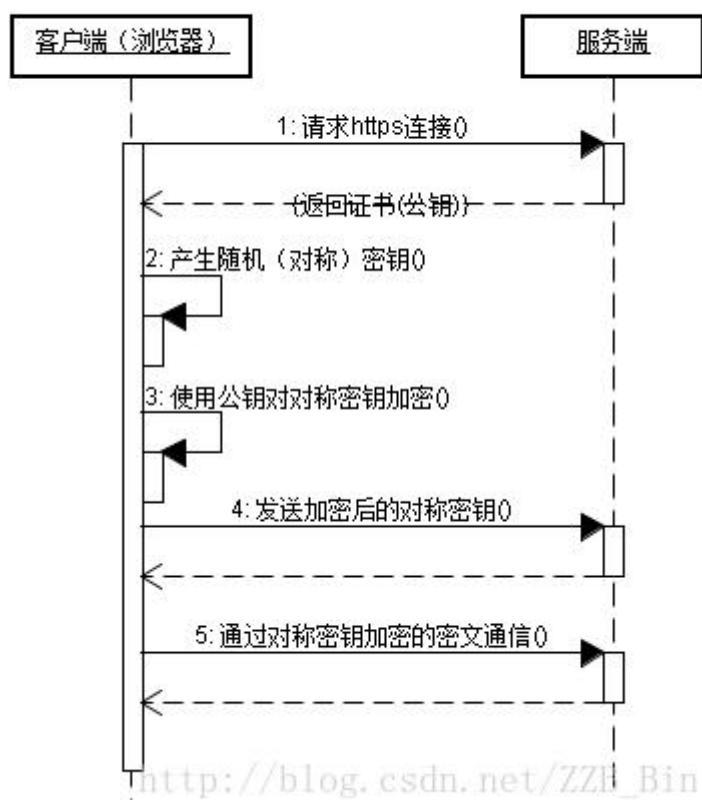
HTTPS 和 HTTP 的区别主要如下：

- 1) https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。
- 2) http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。
- 3) http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。
- 4) http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。
  1. HTTP 的 URL 以 http:// 开头，而 HTTPS 的 URL 以 https:// 开头
  2. HTTP 是不安全的，而 HTTPS 是安全的
  4. 在 OSI 网络模型中，HTTP 工作于应用层，而 HTTPS 工作在传输层
  5. HTTP 无需加密，而 HTTPS 对传输的数据进行加密
  6. HTTP 无需证书，而 HTTPS 需要认证证书

- http是HTTP协议运行在TCP之上。所有传输的内容都是明文，客户端和服务器端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。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方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的连接很简单，是无状态的

### 9.12.3 https 的通信过程

HTTPS通信过程的时序图如下：



文字简述：客户端 A 和服务器 B 之间的交互

1. A 与 B 通过 TCP 建立链接，初始化 SSL 层。
2. 进行 SSL 握手，A 发送 https 请求，传送客户端 SSL 协议版本号、支持的加密算法、随机数等。
3. 服务器 B 把 CA 证书（包含 B 的公钥），把自己支持的加密算法、随机数等回传给 A。
4. A 接收到 CA 证书，验证证书有效性。

5. 校验通过，客户端随机产生一个字符串作为与 **B** 通信的对称密钥，通过 **CA** 证书解出服务器 **B** 的公钥，对其加密，发送给服务器。
6. **B** 用私钥解开信息，得到随机的字符串（对称密钥），利用这个密钥作为之后的通信密钥。
7. 客户端向服务器发出信息，指明后面的数据使用该对称密钥进行加密，同时通知服务器 **SSL** 握手结束。
8. 服务器接收到信息，使用对称密钥通信，通知握手接收。
9. **SSL** 握手结束，使用对称密钥加密数据。

## 9.12.4 SSL 工作原理

<https://blog.csdn.net/ENERGIE1314/article/details/54581411/>

三种协议：1.握手协议 2.记录协议 3. 警报协议

### 1.RSA 握手协议

第一步，**Client** 给出协议版本号、一个客户端生成的随机数（**Client random**），以及客户端支持的加密方法。

第二步，**Server** 确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（**Server random**）。

第三步，**Client** 确认数字证书有效，然后生成一个新的随机数（**Premaster secret**），并使用数字证书中的公钥，加密这个随机数，发给 **Server**。

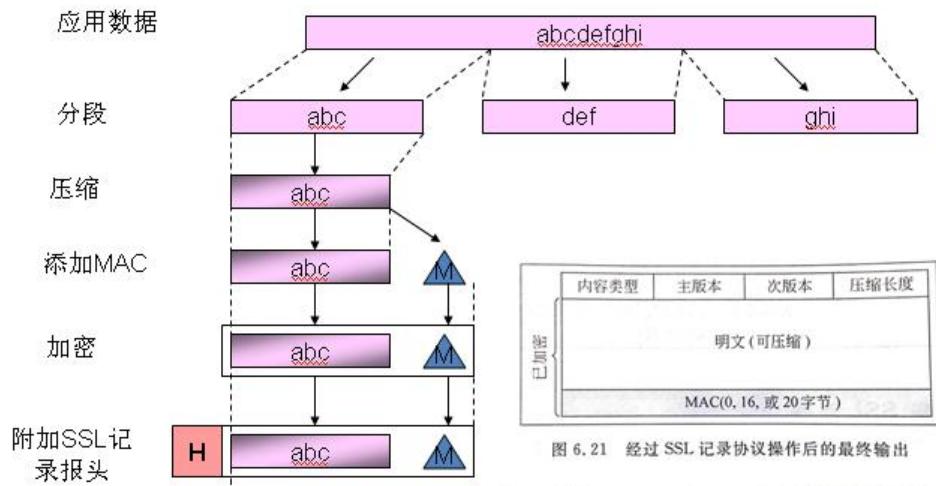
第四步，**Server** 使用自己的私钥，获取 **Client** 发来的随机数（即 **Premaster secret**）。

第五步，**Client** 和 **Server** 根据约定的加密方法，使用前面的三个随机数，生成“对话密钥”（**session key**），用来加密接下来的整个对话过程。

### 2.记录协议

记录协议 对数据传输提供保密性和完整性

记录协议的过程：



### 3. 警报协议

**警报协议** 如果是警报，则值为 1，如果是致命错误，则值为 2

建立连接的过程客户端跟服务端会交换什么信息(参考 TCP 报文结构)

丢包如何解决重传的消耗

traceroute 实现原理

select 和 poll 区别？

在不使用 WebSocket 情况下怎么实现服务器推送的一种方法

可以使用客户端定时刷新请求或者和 TCP 保持心跳连接实现。

查看磁盘读写吞吐量？

PING 位于哪一层

网络重定向，说下流程

controller 怎么处理的请求：路由

.IP 地址分为几类，每类都代表什么，私网是哪些

# 十 操作系统

<http://c.biancheng.net/cpp/html/2611.html>

操作系统概述：

操作系统用来协调软件与底层硬件，相当于彼此的接口

操作系统有进程管理，内存管理，文件管理，输入输出管理。

内存管理的功能有：

- 内存空间的分配与回收：由操作系统完成主存储器空间的分配和管理，使程序员摆脱存储分配的麻烦，提高编程效率。
- 地址转换：在多道程序环境下，程序中的逻辑地址与内存中的物理地址不可能一致，因此存储管理必须提供地址变换功能，把逻辑地址转换成相应的物理地址。
- 内存空间的扩充：利用虚拟存储技术或自动覆盖技术，从逻辑上扩充内存。
- 存储保护：保证各道作业在各自的存储空间内运行，互不干扰。

## 10.1 进程线程

### 10.1.1 . 进程线程区别

进程是系统进行资源分配和调度的一个独立单位，最小的资源管理单位。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，最小的 CPU 执行单元。

线程拥有的资源：`程序计数器 寄存器 栈 状态字`

## 10.1.2 进程通信方式

# 管道( **pipe** )：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

# 有名管道 ( **named pipe** )：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

# 信号量( **semaphore** )：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

# 消息队列( **message queue** )：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

# 信号 ( **signal** )：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

# 共享内存( **shared memory** )：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号量，配合使用，来实现进程间的同步和通信。

# 套接字( **socket** )：套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

**共享内存**：共享内存可以说是最有用的进程间通信方式，也是最快的 IPC 形式。两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

## 10.1.3 僵尸进程

### 1 什么是僵尸进程：

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

### 2 怎样来清除僵尸进程：

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

## 10.1.4 进程同步 PV 信号量

<https://blog.csdn.net/leves1989/article/details/3305609>

首先应弄清 **PV** 操作的含义：**PV** 操作由 **P** 操作原语和 **V** 操作原语组成（原语是不可中断的过程），对信号量进行操作，具体

定义如下：

**P (S)** : ①将信号量  $S$  的值减 1, 即  $S=S-1$ ;  
②如果  $S \geq 0$ , 则该进程继续执行; 否则该进程置为等待状态, 排入等待队列。

**V (S)** : ①将信号量  $S$  的值加 1, 即  $S=S+1$ ;  
②如果  $S > 0$ , 则该进程继续执行; 否则释放队列中第一个等待信号量的进程。

**PV操作的意义**: 我们用信号量及 PV 操作来实现进程的同步和互斥。PV 操作属于进程的低级通信。

什么是信号量? 信号量 (**semaphore**) 的数据结构为一个值和一个指针, 指针指向等待该信号量的下一个进程。信号量的值与相应资源的使用情况有关。当它的值大于 0 时, 表示当前可用资源的数量; 当它的值小于 0 时, 其绝对值表示等待使用该资源的进程个数。注意, 信号量的值仅能由 PV 操作来改变。

一般来说, 信号量  $S \geq 0$  时,  $S$  表示可用资源的数量。执行一次 P 操作意味着请求分配一个单位资源, 因此  $S$  的值减 1; 当  $S < 0$  时, 表示已经没有可用资源, 请求者必须等待别的进程释放该类资源, 它才能运行下去。而执行一个 V 操作意味着释放一个单位资源, 因此  $S$  的值加 1; 若  $S \leq 0$ , 表示有某些进程正在等待该资源, 因此要唤醒一个等待状态的进程, 使之运行下去。

## 10.2 死锁

- 产生死锁的必要条件

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

- 死锁预防

- 打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。
- 打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。
- 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。但是，这种策略也有如下缺点：
  - 在许多情况下，一个进程在执行之前不可能知道它所需要的全部资源。这是由于进程在执行时是动态的，不可预测的；
  - 资源利用率低。无论所分资源何时用到，一个进程只有在占有所需的全部资源后才能执行。即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况。这显然是一种极大的资源浪费；
  - 降低了进程的并发性。因为资源有限，又加上存在浪费，能分配到所需全部资源的进程个数就必然少了。
- 打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁。这种策略与前面的策略相比，资源的利用率和系统吞吐量都有很大提高，但是也存在以下缺点：
  - 限制了进程对资源的请求，同时给系统中所有资源合理编号也是件困难事，并增加了系统开销；
  - 为了遵循按编号申请的次序，暂不使用的资源也需要提前申请，从而增加了进程对资源的占用时间。

- 死锁避免

- 安全序列
- 银行家算法

如何理解分布式锁？

- 分布式锁，是控制分布式系统之间同步访问共享资源的一种方式。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。

### 10.2.1 死锁避免-银行家算法

我们可以把[操作系统](#)看作是银行家，操作系统管理的资源相当于银行家管理的资金，进程向操作系统请求分配资源相当于用户向银行家贷款。

为保证资金的安全，银行家规定：

(1) 当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客；

(2) 顾客可以分期贷款，但贷款的总数不能超过最大需求量；

(3) 当银行家现有的资金不能满足顾客尚需的贷款数额时，对顾客的贷款可推迟支付，但总能使顾客在有限的时间里得到贷款；

(4) 当顾客得到所需的全部资金后，一定能在有限的时间里归还所有的资金。

**操作系统**按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程本次申请的资源数是否超过了该资源所剩余的总量。若超过则拒绝分配资源，若能满足则按当前的申请量分配资源，否则也要推迟分配。

## 10.2.2 死锁避免-安全序列

### 安全序列

安全序列是指对当前申请资源的进程排出一个序列，保证按照这个序列分配资源完成进程，不会发生“酱油和醋”的尴尬问题。

我们假设有进程 P1,P2,...,Pn

则安全序列要求满足：  $P_i(1 \leq i \leq n)$  需要资源  $\leq$  剩余资源 + 分配给  $P_j(1 \leq j < i)$  资源

为什么等号右边还有已经被分配出去的资源？想想银行家那个问题，分配出去的资源就好比第二个开发商，人家能还回来钱，咱得把这个考虑在内。

## 10.3 同步 异步 阻塞 非阻塞

- 线程同步与阻塞的关系
  - 线程同步与阻塞没有一点关系
  - 同步和异步关注的是消息通信机制（synchronous communication/ asynchronous communication）。所谓同步，就是在发出一个“调用”时，在没有得到结果之前，该“调用”就不返回。但是一旦调用返回，就得返回值了。换句话说，就是由“调用者”主动等待这个“调用”的结果。而异步则是相反，“调用”在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，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（消息，返回值）时的状态。阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有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

<https://www.cnblogs.com/George1994/p/6702084.html>

## 1 例子

故事：老王烧开水。

出场人物：老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。

老王想了想，有好几种等待方式

1.老王用水壶煮水，并且站在那里，不管水开没开，每隔一定时间看看水开了没。**一同步阻塞**

老王想了想，这种方法不够聪明。

2.老王还是用水壶煮水，不再傻傻的站在那里看水开，跑去寝室上网，但是还是会每隔一段时间过来看看水开了没有，水没有开就走人。**一同步非阻塞**

老王想了想，现在的办法聪明了些，但是还是不够好。

3.老王这次使用高大上的响水壶来煮水，站在那里，但是不会再每隔一段时间去看水开，而是等水开了，水壶会自动的通知他。**一异步阻塞**

老王想了想，不会呀，既然水壶可以通知我，那我为什么还要傻傻的站在那里等呢，嗯，得换个方法。

4.老王还是使用响水壶煮水，跑到客厅上网去，等着响水壶自己把水煮熟了以后通知他。**一异步非阻塞**

老王豁然，这下感觉轻松了很多。

### 同步和异步

同步就是烧开水，需要自己去轮询（每隔一段时间去看看水开了没），异步就是水开了，然后水壶会通知你水已经开了，你可以回来处理这些开水了。

同步和异步是相对于操作结果来说，会不会等待结果返回。

### 阻塞和非阻塞

阻塞就是说在煮水的过程中，你不可以去干其他的事情，非阻塞就是在同样的情况下，可以同时去干其他的事情。阻塞和非阻塞是相对于线程是否被阻塞。

其实，这两者存在本质的区别，它们的修饰对象是不同的。阻塞和非阻塞是指**进程访问的数据如果尚未就绪，进程是否需要等待**，简单说这相当于函数内部的实现区别，也就是未就绪时是直接返回还是等待就绪。

而同步和异步是指**消息通信机制**，同步一般指主动请求并等待 I/O 操作完毕的方式，当数据就绪后在读写的时候必须阻塞，异步则指主动请求数据后便可以继续处理其它任务，随后等待 I/O 操作完毕的通知，这可以使进程在数据读写时也不阻塞。

## 10.4 操作系统 CPU 调度算法

由于要执行的进程的数目是多于处理器的数目，所以需要处理器去决定下一次运行哪个进程

进程就是作业

1. **先来先服务调度算法 (FCFS)**: 就是按照各个作业进入系统的自然次序来调度作业。这种调度算法的优点是实现简单，公平。其缺点是没有考虑到系统中各种资源的综合使用情况，往往使短作业的用户不满意，因为短作业等待处理的时间可能比实际运行时间长得多。

2. **短作业优先调度算法 (SPF)**: 就是优先调度并处理短作业，所谓短是指作业的运行时间短。而在作业未投入运行时，并不能知道它实际的运行时间的长短，因此需要用户在提交作业时同时提交作业运行时间的估计值。

3. **最高响应比优先算法(HRN)** : FCFS 可能造成短作业用户不满，SPF 可能使得长作业用户不满，于是提出 HRN，选择响应比最高的作业运行。响应比=1+作业等待时间/作业处理时间。

4. **基于优先数调度算法(HPF)** : 每一个作业规定一个表示该作业优先级别的整数，当需要将新的作业由输入并调入内存处理时，优先选择优先数最高的作业。

#### 5. 时间片轮转调度算法

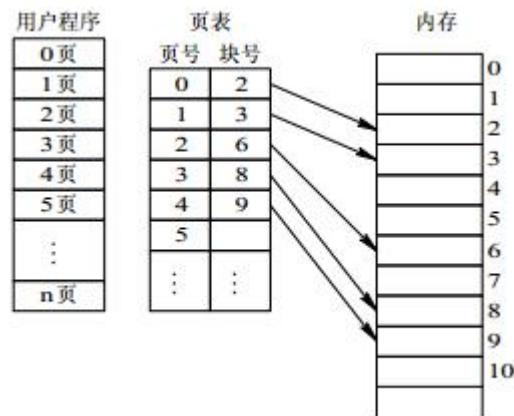
时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片，如 100ms。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

## 10.5 内存管理方式（页存储 段存储 段页存储）

### 页存储

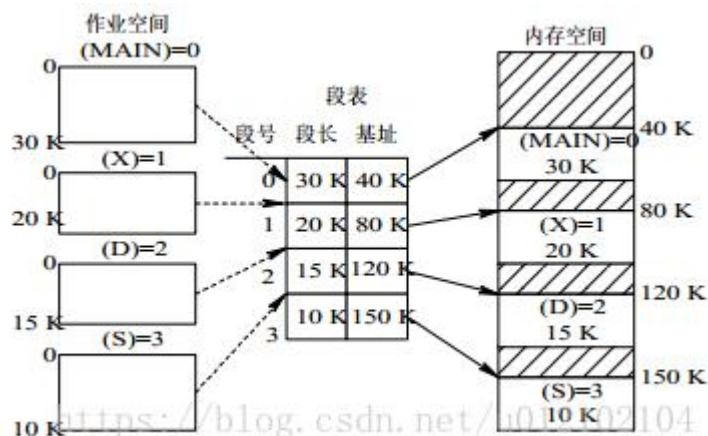
为了便于在内存中找到进程的每个页面所对应的物理块，系统为每个进程建立一张页表，记录页面在内存中对应的物理块号，页表一般存放在内存中。在配置了页表后，进程执行时，通过查找该表，即可找到每页在内存中的物理块号。可见页表作用是实现从页号到物理块号的地址映射，这种是页存

储管理方式。如下图所示：



## 段存储

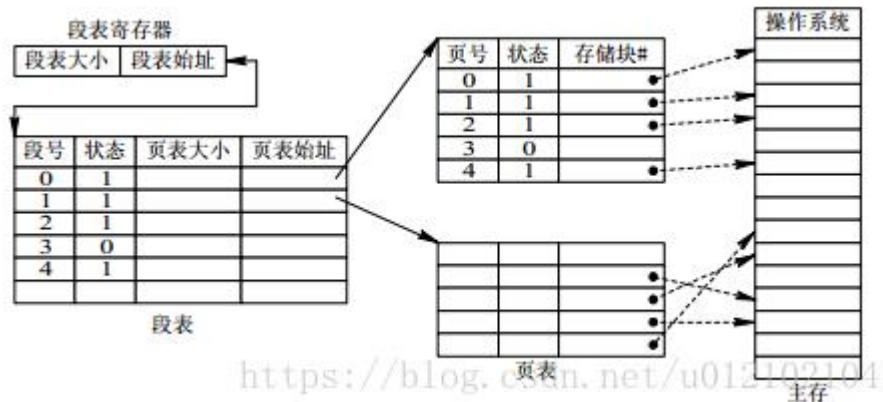
将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配，这种是段存储管理方式。如下图所示：



## 段页存储

作业的地址空间首先被分成若干个逻辑分段，每段都有自己的段号，然后再将每段分成若干个大小相等的页。对于主存空间也分成大小相等的页，主存的分配以页为单位，这种是段页存储管理方式。

如下图所示：



## 10.6 页面置换算法

### 10.6.1 概念

**缺页中断**：缺页中断就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。

**页面置换算法**：在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生**缺页中断**。当发生缺页中断时，如果操作系统内存中没有空闲页面，则**操作系统**必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做**页面置换算法**。

### 10.6.2 OPT 最优页面置换算法

寻找在将来的时间段内，最晚被访问到的页面，然后将其置换到。（无法预知未来，不现实）

### 10.6.3 先进先出置换算法 (FIFO)

**最简单的**页面置换算法是先入先出 (FIFO) 法。这种算法的**实质**是，总是选择在主存中停留时间最长（即最老）的一页置换，即**先进入内存的页，先退出内存**。

### 10.6.4 最近最久未使用 (LRU) 算法

它的**实质**是，当需要置换一页时，选择在最近一段时间里最久没有使用过的页面予以置换。这种算法就称为**最久未使用算法 (Least Recently Used, LRU)**

**LRU** 算法是经常采用的页面置换算法，并被认为是相当好的，但是存在如何实现它的问题。**LRU** 算法需要实际硬件的支持。其问题是**怎么确定最后使用时间的顺序**，对此有两种可行的办法：

1. **计数器**。最简单的情况是使每个页表项对应一个使用时间字段，并给 CPU 增加一个逻辑时钟或计数器。每次存储访问，该时钟都加 1。每当访问一个页面时，时钟寄存器的内容就被复制到相应页表项的使用时间字段中。这样我们就可以始终保留着每个页面最后访问的“时间”。在置换页面时，选择该时间值最小的页面。这样做，不仅要查页表，而且当页表改变时（因 CPU 调度）要维护这个

页表中的时间，还要考虑到时钟值溢出的问题。

2. 栈。用一个栈保留页号。每当访问一个页面时，就把它从栈中取出放在栈顶上。这样一来，栈顶总是放有目前使用的页，而栈底放着目前最少使用的页。由于要从栈的中间移走一项，所以要用具有头尾指针的双向链连起来。在最坏的情况下，移走一页并把它放在栈顶上需要改动 6 个指针。每次修改都要有开销，但需要置换哪个页面却可直接得到，用不着查找，因为尾指针指向栈底，其中有被置换页。

### 10.6.5 时钟(**CLOCK**)置换算法

简单的 **CLOCK** 算法是给每一帧关联一个附加位，称为使用位。当某一页首次装入主存时，该帧的使用位设置为 1；当该页随后再被访问到时，它的使用位也被置为 1。对于页替换算法，用于替换的候选帧集合看做一个循环缓冲区，并且有一个指针与之相关联。当某一页被替换时，该指针被设置成指向缓冲区中的下一帧。当需要替换一页时，操作系统扫描缓冲区，以查找使用位被置为 0 的一帧。每当遇到一个使用位为 1 的帧时，操作系统就将该位重新置为 0；如果在这个过程开始时，缓冲区中所有帧的使用位均为 0，则选择遇到的第一个帧替换；如果所有帧的使用位均为 1，则指针在缓冲区中完整地循环一周，把所有使用位都置为 0，并且停留在最初的位置上，替换该帧中的页。由于该算法循环地检查各页面的情况，故称为 **CLOCK** 算法，又称为最近未用(Not Recently Used, NRU)算法。

**CLOCK** 算法的性能比较接近 **LRU**，而通过增加使用的位数目，可以使得 **CLOCK** 算法更加高效。在使用位的基础上再增加一个修改位，则得到改进型的 **CLOCK** 置换算法。这样，每一帧都处于以下四种情况之一：

1. 最近未被访问，也未被修改( $u=0, m=0$ )。
2. 最近被访问，但未被修改( $u=1, m=0$ )。
3. 最近未被访问，但被修改( $u=0, m=1$ )。
4. 最近被访问，被修改( $u=1, m=1$ )。

算法执行如下操作步骤：

1. 从指针的当前位置开始，扫描帧缓冲区。在这次扫描过程中，对使用位不做任何修改。选择遇到的第一个帧( $u=0, m=0$ )用于替换。
2. 如果第 1)步失败，则重新扫描，查找( $u=0, m=1$ )的帧。选择遇到的第一个这样的帧用于替换。在这个扫描过程中，对每个跳过的帧，把它的使用位设置成 0。
3. 如果第 2)步失败，指针将回到它的最初位置，并且集合中所有帧的使用位均为 0。重复第 1 步，并且如果有必要，重复第 2 步。这样将可以找到供替换的帧。

### 10.7 IO 种类 IO 的原理

## 1. IO 种类

计算机系统中的 I/O 设备按使用特性可分为以下类型：

1) 人机交互类外部设备：用于同计算机用户之间交互的设备，如打印机、显示器、鼠标、键盘等。这类设备数据交换速度相对较慢，通常是以字节为单位进行数据交换。

2) 存储设备：用于存储程序和数据的设备，如磁盘、磁带、光盘等。这类设备用于数据交换，速度较快，通常以多字节组成的块为单位进行数据交换。

3) 网络通信设备：用于与远程设备通信的设备，如各种网络接口、调制解调器等。其速度介于前两类设备之间。网络通信设备在使用和管理上与前两类设备也有很大不同。

除了上面最常见的分类方法，I/O 设备还可以按以下方法分类：

1) 按传输速率分类：

- 低速设备：传输速率仅为每秒几个到数百个字节的一类设备，如键盘、鼠标等。
- 中速设备：传输速率在每秒数千个字节至数万个字节的一类设备，如行式打印机、激光打印机等。
- 高速设备：传输速率在数百个千字节至千兆字节的一类设备，如磁带机、磁盘机、光盘机等。

2) 按信息交换的单位分类：

- 块设备：由于信息的存取总是以数据块为单位，所以存储信息的设备称为块设备。它属于有结构设备，如磁盘等。磁盘设备的基本特征是传输速率较高，以及可寻址，即对它可随机地读/写任一块。
- 字符设备：用于数据输入/输出的设备为字符设备，因为其传输的基本单位是字符。它属于无结构类型，如交互式终端机、打印机等。它们的基本特征是传输速率低、不可寻址，并且在输入/输出时常采用中断驱动方式。

## 2. 设备 I/O 输入输出控制方式

### 程序直接控制方式

计算机从外部设备读取数据到存储器，每次读一个字的数据。对读入的每个字，CPU 需要对外设状态进行循环检查，直到确定该字已经在 I/O 控制器的数据寄存器中。

### 中断驱动方式、

中断驱动方式的思想是，允许 I/O 设备主动打断 CPU 的运行并请求服务，从而“解放”CPU，使得其向 I/O 控制器发送读命令后可以继续做其他有用的工作。

### DMA 方式

在中断驱动方式中，I/O 设备与内存之间的数据交换必须要经过 CPU 中的寄存器，所以速度还是受限，而 DMA（直接存储器存取）方式的基本思想是在 I/O 设备和内存之间开辟直接的数据交换通路，彻底“解放”CPU。

## 通道控制方式

I/O 通道是指专门负责输入/输出的处理器。I/O 通道方式是 DMA 方式的发展，它可以进一步减少 CPU 的干预，即把对一个数据块的读（或写）为单位的干预，减少为对一组数据块的读（或写）及有关的控制和管理为单位的干预。

I/O 通道与 DMA 方式的区别是：DMA 方式需要 CPU 来控制传输的数据块大小、传输的内存位置，而通道方式中这些信息是由通道控制的。另外，每个 DMA 控制器对应一台设备与内存传递数据，而一个通道可以控制多台设备与内存的数据交换。

### 10.8 进程打开同一个文件 那么这两个进程得到的文件描述符（fd）相同

整个系统表包含进程相关信息，如文件在磁盘的位置、访问日期和大小。一个进程打开一个文件，系统打开文件表就会为打开的文件增加相应的条目。当另一个进程执行 open 时，只不过是在其进程打开表中增加一个条目，并指向整个系统表的相应条目。通常，系统打开文件表的每个文件时，还用一个文件打开计数器(Open Count)，以记录多少进程打开了该文件。每个关闭操作 close 则使 count 递减，当打开计数器为 0 时，表示该文件不再被使用。系统将回收分配给该文件的内存空间等资源，若文件被修改过，则将文件写回外存，并将系统打开文件表中相应条目删除，最后释放文件的文件控制块(File Control Block, FCB)。

所以文件描述符不同，count++，不同的 count 不同

## 10.9 select epoll

### select 原理概述

调用 `select` 时，会发生以下事情：

1. 从用户空间拷贝 `fd_set` 到内核空间；
2. 注册回调函数 `_pollwait`；
3. 遍历所有 `fd`，对全部指定设备做一次 `poll`（这里的 `poll` 是一个文件操作，它有两个参数，一个是文件 `fd` 本身，一个是当设备尚未就绪时调用的回调函数 `_pollwait`，这个函数把设备自己特有的等待队列传给内核，让内核把当前的进程挂载到其中）；
4. 当设备就绪时，设备就会唤醒在自己特有等待队列中的【所有】节点，于是当前进程就获取到了完成的信号。`poll` 文件操作返回的是一组标准的掩码，其中的各个位指示当前的不同的就绪状态（全 0 为没有任何事件触发），根据 `mask` 可对 `fd_set` 赋值；
5. 如果所有设备返回的掩码都没有显示任何的事件触发，就去掉回调函数的函数指针，进入有限时的睡眠状态，再恢复和不断做 `poll`，再作有限时的睡眠，直到其中一个设备有事件触发为止。
6. 只要有事件触发，系统调用返回，将 `fd_set` 从内核空间拷贝到用户空间，回到用户态，用户就可以对相关的 `fd` 作进一步的读或者写操作了。

### epoll 原理概述

调用 `epoll_create` 时，做了以下事情：

1. 内核帮我们在 `epoll` 文件系统里建了个 `file` 结点;
2. 在内核 `cache` 里建了个红黑树用于存储以后 `epoll_ctl` 传来的 `socket`;
3. 建立一个 `list` 链表，用于存储准备就绪的事件。

调用 `epoll_ctl` 时，做了以下事情：

1. 把 `socket` 放到 `epoll` 文件系统里 `file` 对象对应的红黑树上；
2. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪 `list` 链表里。

调用 `epoll_wait` 时，做了以下事情：

观察 `list` 链表里有没有数据。有数据就返回，没有数据就 `sleep`，等到 `timeout` 时间到后即使链表没数据也返回。而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，`epoll_wait` 仅需要从内核态 `copy` 少量的句柄到用户态而已。

## 对比

`select` 缺点：

1. 最大并发数限制：使用 32 个整数的 32 位，即  $32 \times 32 = 1024$  来标识 `fd`，虽然可修改，但是有以下第二点的瓶颈；
2. 效率低：每次都会线性扫描整个 `fd_set`，集合越大速度越慢；
3. 内核/用户空间内存拷贝问题。

`epoll` 的提升：

1. 本身没有最大并发连接的限制，仅受系统中进程能打开的最大文件数目限制；
2. 效率提升：只有活跃的 `socket` 才会主动的去调用 `callback` 函数；
3. 省去不必要的内存拷贝：`epoll` 通过内核与用户空间 `mmap` 同一块内存实现。

①了解内存管理页面置换算法（LRU，Java 中如何实现（`LinkedHashMap`））

④了解死锁与饥饿区别

⑥了解如何预防死锁（银行家算法、破坏条件等等）

⑦实现阻塞队列

⑧生产者消费者模型实现

32 位系统的最大寻址空间？ 2 的 32 次方 4GB

不同进程打开了同一个文件，那么这两个进程得到的文件描述符（`fd`）相同吗？

操作系统如何实现输出

三级缓存原理

内存管理：固定分区 动态分区 段 页 都讲讲

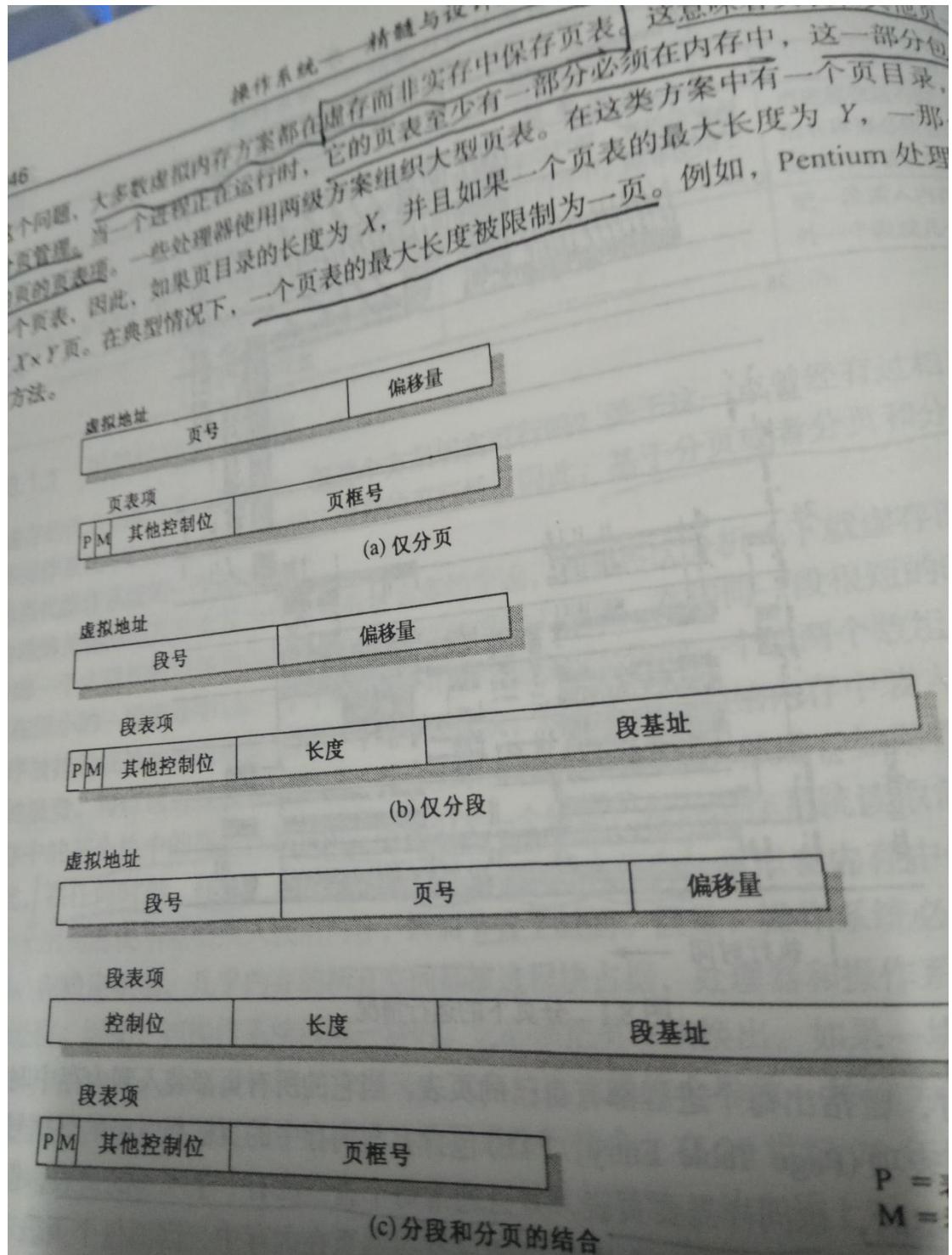
32 位系统的内存寻址空间多大，具体分为哪几种形态？库函数和系统调用有什么区别？  
操作系统内一个进程的内存分段以及对应的作用

## 10.10 物理地址 虚拟地址 逻辑地址

### 物理地址(空间)

用于内存芯片级内存单元寻址。它们与从微处理器的地址引脚按发送到内存总线上的电信号相对应。物理地址由 32 位或 64 位无符号整数表示

虚拟地址：就是在分段 分页的 基础上，比如说将地址分为 32 位，



如图 虚拟地址就是页号加上偏移量这种的表示方法，页号与偏移量结合去找到页表中在物理内存中对应的页。

**逻辑地址 (Logical Address) :**

包含在机器语言指令中用来指定一个操作数或一条指令的地址，每个逻辑地址都由一个段和偏移量组成，偏移量指明了从段开始的地方到实际地址之间的距离。

# 十一 Linux 命令

Cpu 使用率 top 命令 或者 /proc/stat cpu

内存使用率 free 命令 或者 /proc/meminfo

磁盘使用情况 du == disk usage (磁盘使用量, 占用的磁盘空间) (通过搜索文件来计算每个文件的大小然后累加)

磁盘空间及使用情况 df -h (使用文件系统支持)

Netstat 命令用于显示各种网络相关信息, 如网络连接, 路由表, 接口状态

查看进程的端口号: netstat -apn | grep 8080

grep -v 逐行过滤不符合条件的 -e 正则表达式

查看带宽 nload

每个进程的带宽使用 -nethogs

linux 怎么查看网络状态 (vmstat)

linux 怎么查看 IO 状态 (IOTOP)

linux last 命令访问最后登录信息, 就是访问 /var/log/wtmp

logrotate 程序是一个日志文件管理工具。用来把旧的日志文件删除, 并创建新的日志文件

/var/log/messages : centos 记录 Linux 操作系统常见的系统和服务错误信息

/var/log/syslog : ubuntu 记录 Linux 操作系统常见的系统和服务错误信息

Sed 命令编辑文件, 流式

Shell 脚本, 查找历史命令里使用频率最高的 10 条命令

```
history | awk '{CMD[$2]++;count++;} END { for (a in CMD)print CMD[ a ]}'  
CMD[ a ]/count*100 "% " a }' | grep -v "./" | column -c3 -s " " -t |sort -nr | nl | head -n10
```

awk 是一个强大的文本分析工具, 相对于 grep 的查找, sed 的编辑, awk 在其对数据分析并生成报告时, 显得尤为强大。简单来说 awk 就是把文件逐行的读入, 以空格为默认分隔符将每行切片, 切开的部分再进行各种分析处理。

## 统计文件中出现次数最多的前 10 个单词

```
cat words.txt | sort | uniq -c | sort -k1,1nr | head -10
```

sort 命令对一行一个单词的列表进行排序

uniq -c 命令对排序好的单词列表统计每个单词出现的次数。

再用 sort 按照单词出现的次数从大到小排序, 如果出现频率相同, 则再按字母顺序排序。

Uniq -c 只对相邻的行可以进行合并所有一般结合 sort 使用

Sort

-n : 使用『纯数字』进行排序(默认是以文字型态来排序的);

-r 倒序

-k 指定那个字段

-t 指定分隔符

## 10.1 Vim

### (1) 打开与退出

**vi file** : 打开文件 file  
**:q** : 退出 vi 编辑器  
**:wq** : 保存缓冲区的修改并退出编辑器  
**:q!** : 不保存直接退出  
**:w** 保存缓冲区内容至默认的文件  
**:w file** 保存缓冲区内容至 file 文件

(2) 插入文本

**a** : 在当前光标的右边插入文本  
**A** : 在当前光标行的末尾插入文本  
**i** : 在当前光标的左边插入文本  
**I** : 在当前光标所在行的开始处插入文本  
**o** : 在当前行在下面新建一行  
**O** : 在当前行的上面新建一行  
**R** : 替换当前光标位置以及以后的若干文本  
**J** : 连接光标所在行和下一行

(3) 删除文本

**x** : 删除一个字符  
**dd** : 删除一行  
**ndd** : 删除 n 行  
**u** : 撤销上一次操作  
**U** : 撤销对当前行的所有操作

(4) 搜索

**/word** 从前向后搜索第一个出现的 word  
**? word** 从后向前搜索第一个出现的 word

(5) 设置行号

**:set nu** 在屏幕上显示行号  
**:set nonu** 取消行号

表5 vi编辑器的剪切和粘贴键

| 键 | 功能                                           |
|---|----------------------------------------------|
| d | 删除指定位置的文本，并存到临时的缓冲区中。可以使用put操作符（p或P键）访问这个缓冲区 |
| y | 将指定位置的文本复制到临时缓冲区。可以使用put操作符访问这个缓冲区           |
| P | 将指定缓冲区的内容放到当前光标的位置之上                         |
| p | 将指定缓冲区的内容放到当前光标的位置之下                         |

## 10.2 linux 如何查看端口被哪个进程占用？

**lsof -i:端口号**

```
[root@iz0xicth6427347s9n06nvz ~]# lsof -i:48018
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
ssserver 454 root 28u IPv4 67997950 0t0 TCP iz0xicth6427347s9n06nvz:48018->123.125.115.95:http (ESTABLISHED)
[root@iz0xicth6427347s9n06nvz ~]#
```

<https://www.cnblogs.com/bonelee/p/7735479.html>

## 10.3 查看进程打开了哪些文件

lsof -p pid

## 10.4 top

```
[root@iz0xicth6427347s9n06nvz ~]# top -p 16073
top - 08:52:03 up 208 days, 21:22, 1 user,  load average: 0.01, 0.04, 0.05
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.3 us, 0.0 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 1016168 total, 243112 free, 117636 used, 655420 buff/cache
KiB Swap: 0 total, 0 free, 0 used. 724772 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
16073 root      20   0 128128 10784  8576 S  0.3  1.1  55:18.24 AliYunDun
```

PID : 进程的 ID

USER : 进程所有者

PR : 进程的优先级别 , 越小越优先被执行

NInice : 值

VIRT : 进程占用的虚拟内存

RES : 进程占用的物理内存

SHR : 进程使用的共享内存

S : 进程的状态。S 表示休眠 , R 表示正在运行 , Z 表示僵死状态 , N 表示该进程优先值为负数

%CPU : 进程占用 CPU 的使用率

%MEM : 进程使用的物理内存和总内存的百分比

TIME+ : 该进程启动后占用的总的 CPU 时间 , 即占用 CPU 使用时间的累加值。

COMMAND : 进程启动命令名称

top -p 进程 id 查看具体的进程的内存占用

Top -u 用户名 查看用户的进程内存

输入 top 后出来数据了 然后输入 M 按照内存排序 , 输入 P 按照 CPU 排序 输入 T 按照占用 CPU 的时间排序

## 10.5 查看 cpu 核的个数主频

cat /proc/cpuinfo

## 10.6 Linux 如何创建守护进程

(1) 创建子进程 , 父进程退出。

经过这步以后 , 子进程就会成为孤儿进程 (父进程先于子进程退出 , 此时的子进程 , 成为孤儿进程 , 会被 init 进程收养)。

使用 fork () 函数 , 如果返回值大于 0, 表示为父进程 , exit (0) , 父进程退出 , 子进程继续。

(2) 在子进程中创建新会话 , 使当前进程成为新会话组的组长。

使用 `setsid()` 函数，如果当前进程不是进程组的组长，则为当前进程创建一个新的会话期，使当前进程成为这个会话组的首进程，成为这个进程组的组长。

(3) 改变当前目录为根目录。

由于守护进程在后台运行，开始于系统开启，终止于系统关闭，所以要将其目录改为系统的根目录下。进程在执行时，其文件系统不能被卸下。

(4) 重新设置文件权限掩码。

进程从父进程那里继承了文件创建掩码，所以可能会修改守护进程存取权限位，所以要将文件创建掩码清除，`umask(0)`；

(5) 关闭文件描述符。

子进程从父进程那里继承了打开文件描述符。所以使用 `close` 即可关闭。

## 10.7 Linux 管道机制原理

实际上，管道是一个固定大小的缓冲区。在 Linux 中，该缓冲区的大小为 1 页，即 4K 字节，使得它的大小不象文件那样不加检验地增长。使用单个固定缓冲区也会带来问题，比如在写管道时可能变满，当这种情况发生时，随后对管道的 `write()` 调用将默认地被阻塞，等待某些数据被读取，以便腾出足够的空间供 `write()` 调用写。

- 读取进程也可能工作得比写进程快。当所有当前进程数据已被读取时，管道变空。当这种情况发生时，一个随后的 `read()` 调用将默认地被阻塞，等待某些数据被写入，这解决了 `read()` 调用返回文件结束的问题。

## 10.8 查看进程下的线程

<https://www.cnblogs.com/EasonJim/p/8098217.html>

`top -H -p 进程 ID`

```
[root@iz0x1ch6427347s9n06nvz ~]# top -H -p 1438
top - 11:16:29 up 208 days, 23:46,  1 user,  load average: 0.08, 0.06, 0.06
Threads:  1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%cpu(s): 0.3 us, 0.0 sy, 0.0 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 1016168 total, 250944 free, 117564 used, 647660 buff/cache
KiB Swap:      0 total,      0 free,      0 used. 724156 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1438 root      20    0 125888  5896  1700 S  0.0  0.6   1:34.40 python3
```

## 10.9 linux 锁

1.互斥锁

互斥锁只能有对一个线程使用，就是用来互斥的。

以下是互斥锁的基本操作

| 功能       | 函数                                 |
|----------|------------------------------------|
| 初始化互斥锁   | <code>pthread_mutex_init</code>    |
| 阻塞申请互斥锁  | <code>pthread_mutex_lock</code>    |
| 释放互斥锁    | <code>pthread_mutex_unlock</code>  |
| 非阻塞申请互斥锁 | <code>pthread_mutex_trylock</code> |
| 销毁互斥锁    | <code>pthread_mutex_destroy</code> |

### 3. 自旋锁

自旋锁上锁后让等待线程进行忙等待而不是睡眠阻塞，而信号量是让等待线程睡眠阻塞。自旋锁的忙等待浪费了处理器的时间，但时间通常很短，在 1 毫秒以下。

## 10.10 查看行数指令（比如第 100 行到第 150 行 top IP）

1. 要显示一百行到一百五十行的内容怎么输入命令

`sed -n "100,150p" +文件名` sed 命令是一个选取命令 , 它后面单引号括起来的 100,150

就表示 100 行到 150 行的意思 , 100,150 后面的英文字母 p 是 print 显示的意思。

3. 有一个文件 ip.txt, 每行一条 ip 记录, 共若干行, 下面哪个命令可以实现“统计出现次数最多的前 3 个 ip 及其次数”?

`sort ip.txt | uniq -c | sort -rn | head -n 3`

首先 sort 进行排序, 将重复的行都排在了一起, 然后使用 uniq -c 将重复的行的次数放在了行首, 在用 sort -rn 进行反向和纯文本排序, 这样就按照重复次数从高到低进行了排列, 最后利用 head -n 3 输出行首的三行。

4. Linux 下 给定一个文件, 里面存放的是 IP 地址, 统计各个 IP 地址出现的次数

`sort ip.txt | uniq -c | sort -rn`

5. 统计文件中出现最多的前 10 个单词

```
cat words.txt | sort | uniq -c | sort -k1,lnr | head -10
```

sort 命令对一行一个单词的列表进行排序

uniq -c 命令对排序好的单词列表统计每个单词出现的次数。

再用 sort 按照单词出现的次数从大到小排序，如果出现频率相同，则再按字母顺序排序。

Uniq -c 只对相邻的行可以进行合并所有一般结合 sort 使用

Sort

-n : 使用『纯数字』进行排序(默认是以文字型态来排序的)；

-r 倒序

-k 指定那个字段

-t 指定分隔符

## 10.11 linux 进程调度

### 1、进程调度的作用

,进程调度就是对进程进行调度,即负责选择下一个要运行的进程.通过合理的调度,系统资源才能最大限度地发挥作用,多进程才会有并发执行的效果.最终要完成的目标就是为了最大限度的利用处理器时间.即,只要有可以执行的进程,那么就总会有进程正在执行.当进程数大于处理器个数时,某一时刻总会有一些进程不能执行.这些进程等待运行.在这些等待运行的进程中选择一个合适的来执行,是调度程序所需完成的基本工作.

### 2、调度策略

先给一张直观图

**【1】**考虑到进程类型时:I/O 消耗型进程 pk 处理器消耗型进程.

I/O 消耗型进程:指进程大部分时间用来提交 I/O 请求或者是等待 I/O 请求.处理器消耗型进程:与 I/O 消耗型相反,此类进程把时间大多用在执行代码上.此时调度策略通常要在两个矛盾的目标中寻找平衡:进程响应时间短(优先 I/O 消耗型进程)和最大系统利用率(优先处理器消耗型进程).linux 为了保证交互式应用,所以对进程的响应做了优化,即更倾向于优先调度 I/O 消耗型进程.

**【2】**考虑到进程优先级

调度算法中最基本的一类就是基于优先级的调度.调度程序总是选择时间片未用尽而且优先级最高的进程运行.

linux 实现了一种基于动态优先级的调度方法.即:一开始,先设置基本的优先级,然后它允许调度程序根据需要加,减优先级.

eg:如果一个进程在 I/O 等待上消耗的时间多于运行时间,则明显属于 I/O 消耗型进程,那么根据 1 中的考虑,应该动态提高其优先级.

linux 提供了两组独立的优先级范围:

1)nice 值:范围从-20 到+19.默认值是 0,值越小,优先级越高.nice 值也用来决定分配给进程的时间片的长短.

2)实时优先级:范围为 0 到 99.注意,任何实时进程的优先级都高于普通的进程.

### 【3】考虑到进程时间片时

时间片是一个数值,它表明进程在被抢占前所能持续运行的时间.调度策略必须规定一个默认的时间片.时间片过长,则会影响系统的交互性.时间片过短,则会明显增大因进程频繁切换所耗费的时间.调度程度提供较长的默认时间片给交互式程序.此外,linux 调度程序还能根据进程的优先级动态调整分配给它的时间片,从而保证了优先级高的进程,执行的频率高,执行时间长.当一个进程的时间片耗尽时,则认为进程到期了,此时不能在运行.除非所有进程都耗尽了他们的时间片,此时系统会给所有进程重新

## 10.12 零拷贝技术

### 零拷贝技术

#### 直接 I/O

应用程序直接访问磁盘数据,而不经过内核缓冲区,这样做的目的是减少一次从内核缓冲区到用户程序缓存的数据复制

使用 `mmap` 替代 `read`,可以减少 CPU 拷贝次数。当应用程序调用 `mmap()` 之后,数据通过 DMA 拷贝拷贝到内核缓冲区,应用程序和操作系统共享这个缓冲区。这样,操作系统内核和应用程序存储空间不再需要进行任何的数据拷贝操作。

#### 对应用程序地址空间和内核空间的数据传输进行优化的零拷贝技术

对数据在 linux 页缓存和用户进程缓冲区之间的传输进行优化。该零拷贝技术侧重于灵活的处理数据在用户进程中的缓冲区和操作系统的页缓冲区之间的拷贝操作。这种方式延续了传统的通信方式,但是更加灵活。linux 中该方法主要利用写时复制技术。

**写时复制**是计算机编程中常见的一种优化策略,基本思想是这样的:如果多个应用程序需要同时访问一块数据,那么可以为这些应用程序分配指向这块数据的指针,在每个应用程序看来,他们都拥有这块数据的一份拷贝,当其中一个应用程序需要对自己的这份数据进行修改时,就需要将数据真正的拷贝到应用程序的地址空间去。如果应用程序永远不会对这块数据进行修改,那么就永远不需要将数据拷贝到应用程序的地址空间去。在 stl 中 `string` 的实现类似这种策略。

## 10.14 系统调用与库函数的区别

| 函数库调用                           | 系统调用                          |
|---------------------------------|-------------------------------|
| 在所有的ANSI C编译器版本中，C库函数是相同的       | 各个操作系统的系统调用是不同的               |
| 它调用函数库中的一段程序（或函数）               | 它调用系统内核的服务                    |
| 与用户程序相联系                        | 是操作系统的一个入口点                   |
| 在用户地址空间执行                       | 在内核地址空间执行                     |
| 它的运行时间属于“用户时间”                  | 它的运行时间属于“系统”时间                |
| 属于过程调用，调用开销较小                   | 需要在用户空间和内核上下文环境间切换，开销较大       |
| 在C函数库libc中有大约300个函数             | 在UNIX中大约有90个系统调用              |
| 典型的C函数库调用：system fprintf malloc | 典型的系统调用：chdir fork write brk; |

## 10.15 free

我们按照图中来一细细研读(数字编号和图对应)

- 1, total: 物理内存实际总量
- 2, used: 这块千万注意，这里可不是实际已经使用了的内存哦，这里是总计分配给缓存（包含 buffers 与 cache ）使用的数量，但其中可能部分缓存并未实际使用。
- 3, free: 未被分配的内存
- 4, shared: 共享内存
- 5, buffers: 系统分配的，但未被使用的 buffer 剩余量。注意这不是总量，而是未分配的量
- 6, cached: 系统分配的，但未被使用的 cache 剩余量。buffer 与 cache 的区别见后面。
  
- 7, buffers/cache used: 这个是 buffers 和 cache 的使用量，也就是实际内存的使用量，这个非常重  
要了，这里才是内存的实际使用量哦

8, buffers/cache free: 未被使用的 buffers 与 cache 和未被分配的内存之和，这就是系统当前实  
际可用内存。千万注意，这里是 三者之和，也就是第一排的 free+buffers+cached，可不仅仅是未被  
使用的 buffers 与 cache 的和哦，还要加上 free(未分配的和)

9, swap, 这个我想大家都理解，交换分区总量，使用量，剩余量

我想我说得很清晰了

## 10.16 cache 和 buffer 的区别：

**cache 在 cpu 和内存之间，它的速度比内存快，但是造价高**

缓冲区 **buffer** 主要存在于 RAM 中，作为 CPU 暂时存储数据的区域，例如，**当计算机和其他设备具有不同的速度时**，**buffer** 存储着缓冲的数据，这样计算机就可以完成其他任务了

**Cache**：高速缓存，是位于 CPU 与主内存间的一种容量较小但速度很高的存储器。由于 CPU 的速度远高于主内存，CPU 直接从内存中存取数据要等待一定时间周期，Cache 中保存着 CPU 刚用过或循环使用的一部分数据，当 CPU 再次使用该部分数据时可从 Cache 中直接调用，这样就减少了 CPU 的等待时间，提高了系统的效率。Cache 又分为一级 Cache (L1 Cache) 和二级 Cache (L2 Cache)，L1 Cache 集成在 CPU 内部，L2 Cache 早期一般是焊在主板上，现在也都集成在 CPU 内部，常见的容量有 256KB 或 512KB L2 Cache。

**Buffer**：缓冲区，一个用于存储速度不同步的设备或优先级不同的设备之间传输数据的区域。通过缓冲区，可以使进程之间的相互等待变少，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。

Free 中的 **buffer** 和 **cache**：（它们都是占用内存）：

**buffer**：作为 **buffer cache** 的内存，是块设备的读写缓冲区

**cache**：作为 **page cache** 的内存，文件系统的 **cache**

如果 **cache** 的值很大，说明 **cache** 住的文件数很多。如果频繁访问到的文件都能被 **cache** 住，那么磁盘的读 IO 必会非常小。

## 10.13 其它的小问题

1. 查看所有端口的占用情况 **netstat**,

2. 怎么查看一个服务器是否正常运作 **ps aux**

**3. 创建用户命令 adduser 用户名**

**4. Linux : fork 和 wait, 有什么作用** **fork** 用来创建子进程 **wait** 如果子进程状态已经改变，那么 **wait** 调用会立即返回。否则调用 **wait** 的进程将会阻塞直到有子进程改变状态或者有信号来打断（这里所指的状态的改变包括：子进程终止；子进程被一个信号终止来；子进程被一个信号恢复。）这个调用。**wait** 是父进程用来等待来获取子进程的状态信息，获取到以后清除掉子进程。

**5. Linux 线程** 其实在 Linux 中，新建的线程并不是在原先的进程中，而是系统通过一个系统调用 **clone()**。该系统 **copy** 了一个和原先进程完全一样的进程，并在这个进程中执行线程函数。不过这个 **copy** 过程和 **fork** 不一样。**copy** 后的进程和原先的进程共享了所有的变量，运行环境。这样，原先进程中的变量变动在 **copy** 后的进程中便能体现出来。

**6. 内存中 buffer 和 swap, cache 的区别**

**Swap**: 读取数据到内存，内存不够用，这时候把部分内存中的数据写入到磁盘上，这部分磁盘空间就是 **swap**

**Buffer** : **buffer** (缓冲) 是为了提高内存和硬盘 (或其他 I/O 设备) 之间的数据交换的速度而设计的。当程序要写入磁盘时候，不必等写入磁盘这个操作结束再去执行其他的，可以直接写入到 **buffer**

**Cache**: **cache** (缓存) 是为了提高 cpu 和内存之间的数据交换速度而设计的。从磁盘读取的先存入到 **cacache**，以便下次再次访问时候使用。

**7. linux 查看进程的运行堆栈信息命令-gstack gstack 进程 id**

## 8.linux 查看进程消耗的资源

<https://www.cnblogs.com/sparkbj/p/6148817.html>

可以使用一下命令查使用内存最多的 10 个进程

**查看占用 cpu 最高的进程**

`ps aux|head -1;ps aux|grep -v PID|sort -rn -k +3|head`

或者 `top` (然后按下 M, 注意这里是大写)

**查看占用内存最高的进程**

`ps aux|head -1;ps aux|grep -v PID|sort -rn -k +4|head`

或者 `top` (然后按下 P, 注意这里是大写)

9.smp SMP 的全称是"对称多处理" (Symmetrical Multi-Processing) 技术, 是指在一个计算机上汇集了一组处理器(多 CPU),各 CPU 之间共享内存子系统以及总线结构。

进程文件里有哪些信息, ,

`sed` 和 `awk` 的区别

Linux 命令 (有一个文件被锁住, 如何查看锁住它的线程, ,)

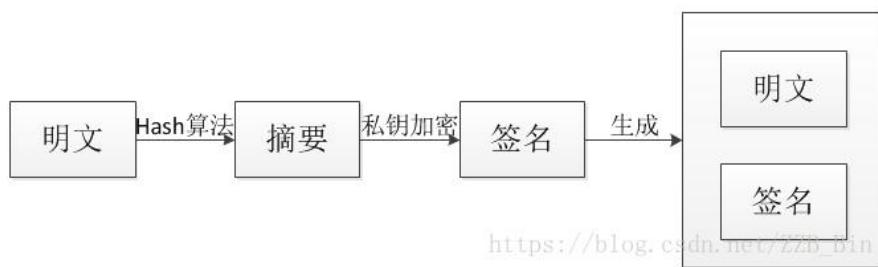
linux 如何查找文件

# 十一 . 安全加密

[http://www.ruanyifeng.com/blog/2011/08/what\\_is\\_a\\_digit\\_of\\_signature.html](http://www.ruanyifeng.com/blog/2011/08/what_is_a_digit_of_signature.html)

## 11.1 数字签名

### 1. 原理



### 2. 验证过程

发送者：将报文通过 hash 算法生成摘要，用私钥加密生成签名。  
接收者：使用公钥解密数字签名，得到摘要 A，再对报文进行 Hash 算法得到摘要 B，比较 A 和 B，一致则表示没有被修改。

## 11.2 数字证书

数字证书则是由证书认证机构（CA, Certificate Authority）对证书申请者真实身份验证之后，用 CA 的根证书对申请人的一些基本信息以及申请人的公钥进行签名（相当于加盖发证书机构的公章）后形成的一个数字文件。



数字证书验证过程：**CA 机构的公钥已经在浏览器发布前就嵌入到浏览器内部了**，所以 CA 的公钥是真实可靠的（如果 CA 机构被黑客攻陷，那么也可能是不可靠的），然后服务器发送自己的公钥给 CA（用 CA 的公钥进行加密），CA 对服务器发来的内容解密得到服务器的公钥，然后 CA 对服务器的公钥进行颁发数字证书（就是数字签名），发给服务器，服务器收到以后，将数字证书，公开密钥发送给客户端，客户端用 CA 的公开密钥验证得到服务器的公开密钥，然后这样客户端就得到了真正可靠的服务器的公开密钥。

## 11.3 公私钥

公钥（Public Key）与私钥（Private Key）是通过一种算法得到的一个密钥对（即一个公钥和一个私钥），公钥是密钥对中公开的部分，私钥则是非公开的部分。

使用这个密钥对的时候，如果用其中一个密钥加密一段数据，必须用另一个密钥解密。比如用公钥加密数据就必须用私钥解密，如果用私钥加密也必须用公钥解密，否则解密将不会成功。

## 11.4 非对称加密 RSA

非对称加密算法需要两个密钥：**公开密钥**（publickey）和**私有密钥**（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解

密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

RSA:

**RSA 算法的步骤主要有以下几个步骤：**

- 1、选择  $p$ 、 $q$  两个超级大的质数，都是 1024 位，
- 2、令  $n = p * q$ 。取  $\phi(n) = (p-1) * (q-1)$ 。计算与  $n$  互质的整数的个数。
- 3、取  $e \in 1 < e < \phi(n)$ ， $(n, e)$  作为公钥对，正式环境中取 65537。  
可以打开任意一个被认证过的 https 证书，都可以看到。
- 4、令  $ed \bmod \phi(n) = 1$ ，计算  $d$ ， $(n, d)$  作为私钥对。计算  $d$  可以利用扩展欧几里的算法进行计算
- 5、销毁  $p$ 、 $q$ 。密文 = 明文  $\wedge e \bmod n$ ，明文 = 密文  $\wedge d \bmod n$ 。  
利用蒙哥马利方法进行计算，也叫反复平方法，非常简单、  
其中  $(n, e)$  是公钥  $(n, d)$  是私钥

## 11.5 对称密钥 DES

**DES:** **DES** 算法是一种分组加密机制，将明文分成  $N$  个组，然后对各个组进行加密，形成各自的密文，最后把所有的分组密文进行合并，形成最终的密文。

对称密钥就是加密使用的密钥是一致的。

## 11.6 DH 加密算法

**Diffie-Hellman 算法概述：**

- (1) Alice 与 Bob 确定两个大素数  $n$  和  $g$ ，这两个数不用保密
- (2) Alice 选择另一个大随机数  $x$ ，并计算  $A$  如下： $A=g^x \bmod n$
- (3) Alice 将  $A$  发给 Bob
- (4) Bob 选择另一个大随机数  $y$ ，并计算  $B$  如下： $B=g^y \bmod n$
- (5) Bob 将  $B$  发给 Alice
- (6) 计算 Alice 的秘密密钥  $K1$  如下： $K1=B^x \bmod n$
- (7) 计算 Bob 的秘密密钥  $K2$  如下： $K2=A^y \bmod n$   $K1=K2$ ，因此 Alice 和 Bob 可以用其进行加解密

## 11.7 SHA MD5

**MD5:** **MD5 消息摘要算法**（英语：MD5 Message-Digest Algorithm），一种被广泛使用的密码散列函数，可以产生出一个 128 位（16 字节）的散列值（hash value），用于确保信息传输完整一致。

**SHA:** **安全散列算法**（英语：Secure Hash Algorithm，缩写为 SHA）是一个密码散列函数家族，是 FIPS 所认证的安全散列算法。能计算出一个数字消息所对应到的，长度固定的字符串（又称消息摘要）的算法。且若输入的消息不同，它们对应到不同字符串的机率很高。

## 十二. 代码

### 12.1 读写文件(BufferedReader)

```
import java.io.*;
public class ReadWriteTxt {
    public static void main(String args[]) {
        try { // 防止文件建立或读取失败，用 catch 捕捉错误并打印，也可以 throw
            /* 读入 TXT 文件 */
            String pathname = "input.txt"; // 绝对路径或相对路径都可以，写入文件时演示相对路径
            File filename = new File(pathname); // 要读取以上路径的 input.txt 文件
            InputStreamReader reader = new InputStreamReader(
                new FileInputStream(filename)); // 建立一个输入流对象 reader
            BufferedReader br = new BufferedReader(reader); // 建立一个对象，它把文件内容转成计算机能读懂的语言
            String line;
            // 网友推荐更加简洁的写法
            while ((line = br.readLine()) != null) {
                // 一次读入一行数据
                System.out.println(line);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            /* 写入 Txt 文件 */
            File writename = new File("output.txt"); // 相对路径，如果没有则要建立一个新的 output.txt 文件
            writename.createNewFile(); // 创建新文件
            BufferedWriter out = new BufferedWriter(new
```

```

        Filewriter(writename));
        out.write("我会写入文件啦 1\r\n"); // \r\n 即为换行
        out.write("我会写入文件啦 2\r\n"); // \r\n 即为换行
        out.flush(); // 把缓存区内容压入文件
        out.close(); // 最后记得关闭文件
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

## 12.2 反射

```

//Test01.java
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
//Exam.java
class Exam{
    private String field1="私有属性";
    public String field2="公有属性";
    public void fun1(){
        System.out.println("fun1:这是一个 public 访问权限方法");
    }

    private void fun2(){
        System.out.println("fun2:这是一个 private 访问权限方法");
    }

    private void fun3(String arg){
        System.out.println("fun3:这是一个 private 访问权限且带参数的方法, 参数为: "+arg);
    }
}

public class ReflectTest {
    public static void main(String args[]){
        Exam e=new Exam();
        try {
            Field field1 = e.getClass().getDeclaredField("field1");
            Field field2 = e.getClass().getDeclaredField("field2");
            field1.setAccessible(true);
            System.out.println("field1: "+field1.get(e));
            field1.set(e,"重新设置一个 field1 值");
        }
    }
}

```

```
        System.out.println("field1: "+field1.get(e));
        System.out.println("field2: "+field2.get(e));
        field2.set(e, "重新设置一个 field2 值");
        System.out.println("field2: "+field2.get(e));
    } catch (NoSuchFieldException e1) {
        e1.printStackTrace();
    }catch (IllegalArgumentException e1) {
        e1.printStackTrace();
    } catch (IllegalAccessException e1) {
        e1.printStackTrace();
    }

try {

    Method method1 = e.getClass().getDeclaredMethod("fun1");
    method1.invoke(e);

    Method method2 = e.getClass().getDeclaredMethod("fun2");
    method2.setAccessible(true);
    method2.invoke(e);

    Method method3 =
e.getClass().getDeclaredMethod("fun3", String.class);
    method3.setAccessible(true);
    method3.invoke(e, "fun3 的参数");
} catch (NoSuchMethodException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (SecurityException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (IllegalAccessException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (IllegalArgumentException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
} catch (InvocationTargetException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
}
```

### 13.3 快排

```
public class Solution {  
    /**  
     * @param A: an integer array  
     * @return: nothing  
     */  
    public void sortIntegers2(int[] A) {  
        // write your code here  
        quicksort(A, 0, A.length-1);  
    }  
    public void quicksort(int[] A, int begin, int end)  
    {  
        int i = begin;  
        int j = end;  
        if(i >= j)  
        {  
            return;  
        }  
        int keng = A[i];  
        while(i < j)  
        {  
            while(i < j && A[j] > keng)  
            {  
                j--;  
            }  
            if(i < j && A[j] <= keng)  
            {  
                A[i] = A[j];  
                i++;  
            }  
            while(i < j && A[i] < keng)  
            {  
                i++;  
            }  
            if(i < j && A[i] >= keng)  
            {  
                A[j] = A[i];  
                j--;  
            }  
        }  
        A[i] = keng;  
        quicksort(A, begin, i-1);  
        quicksort(A, i+1, end);  
    }  
}
```

```
    }  
}
```

## 12.3 LRU

<https://blog.csdn.net/hxqneuq2012/article/details/52709652>

```
import java.util.HashMap;  
public class Main {  
  
    int capacity;  
    HashMap<Integer, Node> map = new HashMap<Integer, Node>();  
    Node head = null;  
    Node end = null;  
  
    public Main(int capacity) {  
        this.capacity = capacity;  
    }  
  
    public int get(int key) {  
        if (map.containsKey(key)) {  
            Node n = map.get(key);  
            remove(n);  
            setHead(n);  
            printNodes("get");  
            return n.value;  
        }  
        printNodes("get");  
        return -1;  
    }  
  
    public void remove(Node n) {  
        if (n.pre != null) {  
            n.pre.next = n.next;  
        } else {  
            head = n.next;  
        }  
  
        if (n.next != null) {  
            n.next.pre = n.pre;  
        } else {  
            end = n.pre;  
        }  
    }  
}
```

```
    }

    public void setHead(Node n) {
        n.next = head;
        n.pre = null;

        if (head != null)
            head.pre = n;

        head = n;

        if (end == null)
            end = head;
    }

    public void set(int key, int value) {
        if (map.containsKey(key)) {
            Node old = map.get(key);
            old.value = value;
            remove(old);
            setHead(old);
        } else {
            Node created = new Node(key, value);
            if (map.size() >= capacity) {
                map.remove(end.key);
                remove(end);
                setHead(created);
            } else {
                setHead(created);
            }
        }

        map.put(key, created);
    }
    printNodes("set");
}

public void printNodes(String explain) {
    System.out.print(explain + ":" + head.toString());
    Node node = head.next;
    while (node != null) {
        System.out.print(node.toString());
    }
}
```

```

        node = node.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    Main lruCacheTest = new Main(5);
    lruCacheTest.set(1, 1);
    lruCacheTest.set(2, 2);
    lruCacheTest.set(3, 3);
    lruCacheTest.set(4, 4);
    lruCacheTest.set(5, 5);
    System.out.println("lruCacheTest.get(1): " +
lruCacheTest.get(1));
    lruCacheTest.set(6, 6);
    System.out.println("lruCacheTest.get(2): " +
lruCacheTest.get(2));
}

class Node {
    int key;
    int value;
    Node pre;
    Node next;

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

    @Override
    public String toString() {
        return this.key + "-" + this.value + " ";
    }
}

```

## 十三 . 面经

13.1 作业帮面经  
一面挂

## 1. 循环有序数组找一个数

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        int [] temp = {3, 2};
        System.out.println(find(temp, 2));
    }
    public static int find(int [] array, int n)
    {
        if(array.length == 1)
        {
            if(array[0] == n)
                return 0;
            return -1;
        }
        int start = 0;
        int end = array.length-1;
        if(start >= end)
            return -1;
        int mid = (start + end) / 2;
        while(start <= end)
        {
            mid = (start + end) / 2;
            int flag = 0;
            if(array[mid] == n)
                return mid;
            if(array[start] == n)
                return start;
            if(array[end] == n)
                return end;
            if(array[mid] > array[start])
            {
                if(array[mid] > n && n > array[start])
                {
                    end = mid - 1;
                }
                else
                {
                    start = mid + 1;
                }
            } else {
                if(array[mid] < n && n < array[end])
                {
                    start = mid + 1;
                }
            }
        }
    }
}
```

```
        }
    else
    {
        end = mid - 1;
    }
}
return -1;
}
}
```

## 十四 . 项目

### 14.1. jieba 分词原理

jieba 分词的原理

**jieba 介绍:**

一、支持三种分词模式：

精确模式，试图将句子最精确地切开，适合文本分析；

全模式，把句子中所有的可以成词的词语都扫描出来，速度非常快，但是不能解决歧义；

搜索引擎模式，在精确模式的基础上，对长词再次切分，提高召回率，适合用于搜索引擎分词。

二、jieba 自带了一个叫做 `dict.txt` 的词典，里面有 2 万多条词，包含了词条出现的次数（这个次数是于作者自己基于人民日报语料等资源训练得出来的）和词性。这个第一条的 `trie` 树结构的词图扫描，说的就是把这 2 万多条词语，放到一个 `trie` 树中，而 `trie` 树是有名的前缀树，也就是说一个词语的前面几个字一样，就表示他们具有相同的前缀，就可以使用 `trie` 树来存储，具有查找速度快的优势。

三、jieba 分词应该属于概率语言模型分词

概率语言模型分词的任务是：在全切分所得的所有结果中求某个切分方案  $S$ ，使得  $P(S)$  最大。

**jieba 用到的算法：**

一、基于 `Trie` 树结构实现高效的词图扫描，生成句子中汉字所有可能成词情况所构成的有向无环图（DAG）

1. 根据 `dict.txt` 生成 `trie` 树。字典在生成 `trie` 树的同时，也把每个词的出现次数转换为了频率；

2. 对待分词句子，根据 `dict.txt` 生成的 `trie` 树，生成 DAG，实际上通俗的说，就是对待分词句子，根据给定的词典进行查词典操作，生成几种可能的句子切分。`jieba` 的作者在 DAG 中记录的是句子中某个词的开始位置，从 0 到  $n-1$  ( $n$  为句子的长度)，每个

开始位置作为字典的键, `value` 是个 `list`, 其中保存了可能的词语的结束位置(通过查字典得到词, 开始位置+词语的长度得到结束位置)注: 所以可以联想到, `jieba` 支持全模式分词, 能把句子中所有的可以成词的词语都扫描出来

例如:`{0:[1,2,3]}` 这样一个简单的 DAG, 就是表示 0 位置开始, 在 1,2,3 位置都是词, 就是说 0~1, 0~2, 0~3 这三个起始位置之间的字符, 在 `dict.txt` 中是词语. 可看示例切分词图。

二、采用了动态规划查找最大概率路径, 找出基于词频的最大切分组合

1. 查找待分词句子中已经切分好的词语 (我觉得这里应该是全模式下的分词 `list`), 对该词语查找该词语出现的频率(次数/总数), 如果没有该词(既然是基于词典查找进行的分词, 应该是有的), 就把词典中出现频率最小的那个词语的频率作为该词的频率, 也就是说  $P(\text{某词语})=\text{FREQ.get('某词语', min\_freq)}$

2. 根据动态规划查找最大概率路径的方法, 对句子从右往左反向计算最大概率 (一些教科书上可能是从左往右, 这里反向是因为汉语句子的重心经常落在后面, 就是落在右边, 因为通常情况下形容词太多, 后面的才是主干, 因此, 从右往左计算, 正确率要高于从左往右计算, 这个类似于逆向最大匹配),  $P(\text{NodeN})=1.0$ ,

$P(\text{NodeN-1})=P(\text{NodeN}) * \text{Max}(P(\text{倒数第一个词}))$ ...依次类推, 最后得到最大概率路径, 得到最大概率的切分组合.

## Python

Python 如何写爬虫

python 全局锁

python 爬虫分为哪几种, 分别是

你爬虫那个项目中是怎么解决反爬虫问题的?

## Git

git 常用命令、有哪些目录或区域 git 是怎么管理代码的, 提交错误的话, 怎么撤销  
介绍数据仓库

## 计算机磁盘

为什么磁盘 I/O 阻塞代价很大

SSD 没有磁头, 为什么 I/O 代价还很高

# Socket

## 其它

项目中权限管理如何实现的

非对称加密

SHA, MD5

如何设计一个秒杀系统(看来你对限流不是很熟悉啊)

程序出现问题，如何定位（哪一行代码）

20亿QQ号的插入与查找最小存储开销实现方案（提示：位图）

读过 JDK 源码吗

提升访问网页效率的方法(缓存:客户端缓存, cdn 缓存, 服务器缓存, 多线程, 负载均衡之类)

分布式服务中,某个服务速度很慢,如何排查(发散性较强,从计算机网络,到多线程到数据库结构都能说说)

**怎么定位查找**

①了解分布式缓存、Zookeeper、阿里 dubbo、Nginx 等

②了解 NoSQL（Redis 等）

③了解 Hadoop 大数据相关知识

简述 Select poll 和 epoll, 还有 direct io 和 buffer io 区别（一脸蒙蔽…）

C 的拷贝构造函数, 深拷贝和浅拷

分布式架构中, 怎么保证数据的一致性

```
1. public A { public void test() {} }  
public B extends A{ protected void test() {} }
```

这样有问题吗? 为什么?

```
2. public A { public long test() {} }  
public B extends A{ public int test() {} }
```

都不行

int i=0; Integer i1=0; Integer i2=new Integer (0); 输出 i==i1; i==i2; i1==i2 分别是  
false 还是 true

分别返回 true, true, false。

jdk 1.5 之后 有了自动装箱和拆箱 所以 前两个是 true 后两个 是不同的对象

maven 冲突如何解决;

大型论坛网站难免会出现敏感评论，如何过滤敏感评论

## 自我介绍：

### 项目：

1. 对你来说影响最大的一个项目（该面试中有关项目问题都针对该项目展开）？
2. 项目哪一部分最难攻克？如何攻克？  
个人建议：大家一定要选自己印象最深的项目回答，首先按模块，然后组成人员，最后你在项目中的角色和发挥的作用。全程组织好语言，最好不要有停顿，面试官可以看出你对项目的熟悉程度
3. 你觉得你在项目运行过程中作为组长是否最大限度发挥了组员的优势？具体事例？
4. 职业规划，今天想发展的工作方向
5. 项目里我遇到过的最大的困难是什么
6. 实验室的新来的研一，你会给他们什么学习上的建议，例如对于内核源码的枯燥如何克服
7. 如何协调团队中多人的工作
8. 当团队中有某人的任务没有完成的很好，如何处理
9. 平时看些什么书，技术 综合
10. 项目解决的什么问题 用到了哪些技术
11. 怎么预防 bug 日志 jvm 异常信息 如何找问题的根源（统计表格）
12. 你是怎么学习的，说完会让举个例子
13. 实习投了哪几个公司？为什么，原因
14. 最得意的项目是什么？为什么？(回答因为项目对实际作用大，并得到认可)
15. 最得意的项目内容，讲了会
16. 你简历上写的是最想去的部门不是我们部门，来我们部门的话对你有影响麽？
17. 你除了在学校还有哪些方式去获取知识和技术？
18. 你了解阿里文化和阿里开源吗？
19. 遇到困难解决问题的思路？
20. 我觉得最成功的一件事了，我说能说几件吗，说了我大学明白明白了自己想干什么，选择了自己喜欢的事，大学里学会了和自己相处，自己一个人的时候也不会感觉无聊，精神世界比较丰富，坚持锻炼，健身，有个很不错的身体，然后顿了顿笑着说，说，有一个对我很好的女朋友算吗？
21. 压力大的时候怎么调整？多个任务冲突了你怎么协调的？
22. 家里有几个孩子，父母对你来北京有什么看法？
23. 职业生涯规划
24. 你在什么情况下可能会离职

25. 对你影响最大的人
26. 1. 优点 3 个, 以及缺点 2. 说说你应聘这个岗位的优势 3. 说说家庭 4. 为什么想来网易, 用过网易的哪些产品, 对比下有什么好的地方 5. 投递了哪些公司, 对第一份工作怎么看待
27. 为什么要选择互联网 (楼主偏底层的)
28. 为什么来网易 (看你如何夸)
29. 在校期间怎样学习
30. 经常逛的技术性网站有哪些?
31. 举出你在开发过程中遇到的原先不知道的 bug, 通过各种方式定位 bug 并最终成功解决的例子
32. 举出一个例子说明你的自学能力

7 次面试记录, 除了京东基本上也都走到了很后面的阶段。硬要说经验可能有三点:

- **不会就不会。**我比较爽快, 如果遇到的不会的甚至是不确定的, 都直接说 :“对不起, 我答不上来”之类的。
- **一技之长。**中间件和架构相关的实习经历, 让我基本上和面试官都可以聊的很多, 也可以看到, 我整个过程没有多少算法题。是因为面试官和你聊完项目就知道你能做事了。其实, 面试官很不愿意出算法题的 (BAT 那个档次除外), 你能和他扯技术他当然高兴了。关键很多人只会算法 (逃) 。
- **基础非常重要。**面试官只要问 Java 相关的基础, 我都有自信让一般的面试官感觉惊讶, 甚至学到新知识 (之前遇到的阿里的面试官, 我并没有做到, 还被按在地上摩擦) 。
- **说话时面带微笑**