# 你离BAT之间，只差这一套Java面试题

必看！ 进阶% 了解?

# 计算机基础知识

## 数据结构

### ！ 1. 什么是队列、栈、链表

- 队列：是只允许在一端进行插入操作，而在另一端进行删除操作的线性表。允许插入的一端称为队尾，允许删除的端称为队尾，先进先出的线性表。
- 栈：是限定尽在表尾进行插入和删除操作的线性表。允许插入的一段称为栈顶，另一端称为栈底，不含任何数据元素的栈称为空栈，后进先出的线性表
- 链表：是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针(Pointer)

### ！ 2. 什么是树（平衡树，排序树，B树，B+树，R树，红黑树）、堆（大根堆，小根堆）、图（有向图，无向图，拓扑）

- 树：是n(n>=0)个节点的有限集。n=0是称为空树。

  在任意一个非空树中：

  1. 有且仅有一个特定的称为根（root）的结点；
  2. 当n>1时，其余结点看分为m(m>0)个互不相交的有限集T1、T2、……、Tm，其中每一个集合本身又是一棵树，并且称为根的子树
  - 平衡树：
  - 排序树：
  - B树
  - B+树
  - R树
  - 红黑树
- 堆：
  - 大根堆
  - 小根堆
- 图：
  - 有向图
  - 无向图
  - 拓扑

### ！ 3. 栈和队列的相同和不同之处

- 栈与队列的相同点：
  1. 都是线性结构。
  2. 插入操作都是限定在表尾进行。

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

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

### ？ 4. 栈通常采用的两种存储结构

### %5. 两个栈实现队列，两个队列实现栈

## 算法

### ！ 1. 排序都有哪几种方法

- 冒泡排序
- 简单选择排序
- 直接插入排序
- 希尔排序
- 堆排序
- 归并排序
- 快速排序

### ！ 2. 会写常用的排序算法，如快排，归并等

### %3. 各种排序算法的时间复杂度和稳定性，重点快排

### ！ 4. 单链表的遍历和逆序

### ！ 5. 深度优先搜索和官渡优先搜索

### ？ 6. 最小生成树

### ！ 7. 常见的Hash算法，哈希的原理和代价

### %8. 全排列，贪心算法，kmp算法，hash算法

### ？ 9. 一致性Hash算法

## 操作系统

### ？ 1. 虚拟内存管理

### ？ 2. 换页算法

### ！ 3. 进程间通信

进程间通信的方式——信号、管道、消息队列、共享内存

### ？4. 进程同步：生产者消费者问题，哲学家就餐问题，读者写者问题

### ！5. 死锁的四个必要条件，避免方法

- 四个必要条件：
  - 互斥条件：资源是独占的且排他使用，进程互斥使用资源，即任意时刻一个资源只能给一个进程使用，其他进程若申请一个资源，而该资源被另一进程占有时，则申请者等待直到资源被占有者释放。
  - 不可剥夺条件：进程所获得的资源在未使用完毕之前，不被其他进程强行剥夺，而只能由获得该资源的进程资源释放。
  - 请求和保持条件：进程每次申请它所需要的一部分资源，在申请新的资源的同时，继续占用已分配到的资源。
  - 循环等待条件：在发生死锁时必然存在一个进程等待队列{P1,P2,...,Pn},其中P1等待P2占有的资源，P2等待P3占有的资源，...，Pn等待P1占有的资源，形成一个进程等待环路，环路中每一个进程所占有的资源同时被另一个申请，也就是前一个进程占有后一个进程所深情地资源。

  以上给出了导致死锁的四个必要条件，只要系统发生死锁则以上四个条件至少有一个成立。事实上循环等待的成立蕴含了前三个条件的成立，似乎没有必要列出然而考虑这些条件对死锁的预防是有利的，因为可以通过破坏四个条件中的任何一个来预防死锁的发生。

- 避免方法

  死锁避免的基本思想：系统对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，如果分配后系统可能发生死锁，则不予分配，否则予以分配，这是一种保证系统不进入死锁状态的动态策略。

### ！6. Linux 的一些基本命令，如 `ls` 、 `tail` 、 `chmod` 等

- 档案与目录的显示：ls
- 只看尾几行：tail
- 改变文件的权限：chmod

## 计算机网络

### ！1、tcp,udp区别

1. TCP面向连接（如打电话要先拨号建立连接）;UDP是无连接的，即发送数据之前不需要建立连接
2. TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保 证可靠交付
3. TCP面向字节流，实际上是TCP把数据看成一连串无结构的字节流;UDP是面向报文的

   UDP没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如IP电话，实时视频会议等）
4. 每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信
5. TCP首部开销20字节;UDP的首部开销小，只有8个字节
6. TCP的逻辑通信信道是全双工的可靠信道，UDP则是不可靠信道

### ！2、HTTP请求和响应的全过程

http请求与响应全过程

### ！3、HTTP常见响应码：200、301、302、404、500

- 200：正确的请求返回正确的结果，如果不想细分正确的请求结果都可以直接返回200
- 301：请求成功，但是资源被永久转移。比如说，我们下载的东西不在这个地址需要去到新的地址
- 302：请求的资源临时从不同的URI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
- 404：请求的内容不存在
- 500：服务器错误

## ！4、get和post的区别

- 区别

  1. GET在浏览器回退时是无害的，而POST会再次提交请求
  2. GET产生的URL地址可以被Bookmark，而POST不可以。
  3. GET请求会被浏览器主动cache，而POST不会，除非手动设置。
  4. GET请求只能进行url编码，而POST支持多种编码方式
  5. GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
  6. GET请求在URL中传送的参数是有长度限制的，而POST么有。
  7. 对参数的数据类型，GET只接受ASCII字符，而POST没有限制。
  8. GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
  9. GET参数通过URL传递，POST放在Request body中。

- 事实上GET和POST是HTTP协议中的两种发送请求的方法，HTTP是基于TCP/IP的关于数据如何在万维网中如何通信的协议，HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET和POST能做的事情是一样一样的。你要给GET加上request body，给POST带上url参数，技术上是完全行的通的。 GET和POST只是把tcp标识成了两种不同的方法，但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。

- GET和POST还有一个重大区别，简单的说：GET产生一个TCP数据包；POST产生两个TCP数据包。

  对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

  而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

  因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑！跳入需谨慎。为什么？

  1. GET与POST都有自己的语义，不能随便混用。
  2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。
  3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

## ！5、forward和redirect的区别

- 直接请求转发（forward）：“A找B借钱，B说没有，B去找C借，借到借不到都会把消息传递给A”

  RequestDispatcher类的forward()方法

  客户端浏览器只发出一次请求，Servlet把请求转发给Servlet、HTML、JSP或其它信息资源，由第2个信息资源响应该请求，两个信息资源共享同一个request对象。

- 间接请求转发（redirect）："A找B借钱，B说没有，让A去找C借"。

  HttpServletRequest类的sendRedirect()方法

  服务器端在响应第一次请求的时候，让浏览器再向另外一个URL发出请求，从而达到转发的目的。它本质上是两次HTTP请求，对应两个request对象。

## ！6、`osi`七层模型

- 第七层：应用层

  网络服务与最终用户的一个接口。

  协议有：**HTTP FTP TFTP SMTP SNMP DNS TELNET HTTPS POP3 DHCP**
- 第六层：表示层

  数据的表示、安全、压缩。（在五层模型里面已经合并到了应用层）

  格式有，**JPEG、ASCll、DECOIC、加密格式等**
- 第五层：会话层

  建立、管理、终止会话。（在五层模型里面已经合并到了应用层）

  **对应主机进程，指本地主机与远程主机正在进行的会话**
- 第四层：传输层

  定义传输数据的协议端口号，以及流控和差错校验。

  协议有：**TCP UDP，数据包一旦离开网卡即进入网络传输层**
- 第三层： 网络层

  进行逻辑地址寻址，实现不同网络之间的路径选择。

  协议有：**ICMP IGMP IP（IPV4 IPV6） ARP RARP**
- 第二层：数据链路层

  建立逻辑连接、进行硬件地址寻址、差错校验 [2] 等功能。（由底层网络定义协议）

  将比特组合成字节进而组合成帧，用MAC地址访问介质，错误发现但不能纠正。
- 第一层：物理层

  建立、维护、断开物理连接。（由底层网络定义协议）

TCP/IP 层级模型结构，应用层之间的协议通过逐级调用传输层（Transport layer）、网络层（Network Layer）和物理数据链路层（Physical Data Link）而可以实现应用层的应用程序通信互联。

应用层需要关心应用程序的逻辑细节，而不是数据在网络中的传输活动。应用层其下三层则处理真正的通信细节。在 Internet 整个发展过程中的所有思想和着重点都以一种称为 RFC（Request For Comments）的文档格式存在。针对每一种特定的 TCP/IP 应用，有相应的 RFC [3] 文档。

一些典型的 TCP/IP 应用有 FTP、Telnet、SMTP、SNTP、REXEC、TFTP、LPD、SNMP、NFS、INETD 等。RFC 使一些基本相同的 TCP/IP 应用程序实现了标准化，从而使得不同厂家开发的应用程序可以互相通信

## ！7、`tcp/ip`四层模型及原理

- tcp/ip 四层模型

| OSI 体系结构 | TCP/IP 协议集 | |
|---|---|---|
| 应用层 | 应用层 | TELNET、FTP、HTTP、SMTP、DNS 等 |
| 表示层 | | |
| 会话层 | | |
| 传输层 | 传输层 | TCP、UDP |
| 网络层 | 网络层 | IP、ICMP、ARP、RARP |
| 数据链路层 | 网络接口层 | 各种物理通信网络接口 |
| 物理层 | | |

1) 网络接口层: 主要是指物理层次的一些接口,比如电缆等.

2) 网络层: 提供独立于硬件的逻辑寻址,实现物理地址与逻辑地址的转换.

在 TCP / IP 协议族中，网络层协议包括 IP 协议（网际协议），ICMP 协议（Internet 互联网控制报文协议），以及 IGMP 协议（Internet 组管理协议）.

3) 传输层: 为网络提供了流量控制,错误控制和确认服务.

在 TCP / IP 协议族中有两个互不相同的传输协议： TCP（传输控制协议）和 UDP（用户数据报协议）.

4) 应用层: 为网络排错,文件传输,远程控制和 Internet 操作提供具体的应用程序

- 原理

### ！ 8、TCP和UDP区别

### ！ 9、TCP的三次握手，四次关闭

TCP建立连接三次握手和释放连接四次握手

### %10、丢包,粘包，

### ？ 11、容量控制，拥塞控制

### ？ 12、子网划分

### %13、IPV4和IPV6

### ？ 14、HTTPS和HTTP/2

参考资料

> TCP和UDP的最完整的区别
>
> GET和POST两种基本请求方法的区别
>
> 面试时，你被问到过 TCP/IP 协议吗?

## 数据库

### ！ 1、范式

- 第一范式：无重复的列
- 第二范式：非主属性非部分依赖于主关键字
- 第三范式：属性不依赖于其它非主属性

## ！2、数据库事务和隔离级别

- 数据库事物：
  - **原子性：**事务的原子性指的是，事务中包含的程序作为数据库的逻辑工作单位，它所做的对数据修改操作要么全部执行，要么完全不执行，这种特性称为原子性。（简单地说就是，几个对于数据库的操作要么全执行，要么全不执行，即同时成功起作用或同时失败没影响）
  - **一致性：**事务一致性值得是在一个事务执行之前和执行之后数据库都必须处于一致性状态（中途是否一致不用管），这种特性称为一致性。（如果数据库的状态满足所有的完整性约束，就说该数据库是一致的。一致性处理数据库中对所有语义的保护。如：客户K1要向客户K2转账，K1账户减少的金额就是K2账户增加的金额，在转账之前K1和K2账户的金额之和与转账之后K1和K2账户的金额之和是一样的，在转账期间可能不满足这种一致性，但事务前后是数据库数据是一致的）
  - **隔离性：**隔离性指的是并发的事务是相互隔离的。（一个事务内部的操作及正在操作的数据必须封装起来，不被其它企图进行修改的事务看到）
  - **持久性：**持久性指当系统或介质发生故障时，确保已提交的更新不能丢失。（一个事务提交，DBMS保证它对数据库中数据的改变应该是永久性的，可以经受任何系统故障，持久性通过数据库备份和恢复来保证）
- 隔离级别：
  - **READ UNCOMMITTED**（读未提交数据）：允许事务读取未被其他事务提交的变更数据，会出现脏读、不可重复读和幻读问题。
  - **READ COMMITTED**（读已提交数据）：只允许事务读取已经被其他事务提交的变更数据，可避免脏读，仍会出现不可重复读和幻读问题。
  - **REPEATABLE READ**（可重复读）：确保事务可以多次从一个字段中读取相同的值，在此事务持续期间，禁止其他事务对此字段的更新，可以避免脏读和不可重复读，仍会出现幻读问题。
  - **SERIALIZABLE**（序列化）：确保事务可以从一个表中读取相同的行，在这个事务持续期间，禁止其他事务对该表执行插入、更新和删除操作，可避免所有并发问题，但性能非常低。

## ！3、为什么需要锁，锁分类，锁粒度

- 为什么需要锁：在数据库中就会产生多个事务同时存取同一数据的情况。若对并发操作不加控制就可能会读取和存储不正确的数据，破坏数据库的一致性(脏读，不可重复读，幻读等)，可能产生死锁。
- 锁分类：行级锁，表级锁，乐观锁，悲观锁

## %4、`乐观锁`，`悲观锁`的概念及实现方式

## ！5、`分页`如何实现（`Oracle`，`MySql`）

- MySQL实现分页

  MySQL实现分页效果比较简单，只有一个limit关键字就可以解决。

  示例：SELECT username,password FROM tb_user WHERE id = 1 LIMIT 100,10;

  具体：select * from tableName where 条件 limit 当前页码*页面容量-1,页面容量

- Oralce实现分页

  SELECT *  FROM (SELECT A.*, ROWNUM RN FROM (SELECT * FROM tableName order by id) A WHERE ROWNUM <=20)  WHERE RN >= 11;

select count(*) from tableName where 条件

## ！6、Mysql引擎

| | Innodb | Myisam | Memory |
| --- | --- | --- | --- |
| 事务 | 支持 | 不支持 | |
| 执行速度 | 比较快 | 快 | |
| 大容量数据 | 是 | | |
| 创建表->存储位置 | 数据库系统（缓存池） ->表空间 | 单独的文件 | 内存中->磁盘文件 |

### ? 7、MYSQL语句优化

### %8、从一张大表读取数据，如何解决性能问题

### ! 9、`内连接`，`左连接`，`右连接`作用及区别

左连接：左边有的，右边没有的为null

右连接：左边没有的，右边有的为null

内连接：显示左边右边共有的

### ! 10、`Statement`和`PreparedStatement`之间的区别

1. PreparedStatement是预编译的,对于批量处理可以大大提高效率. 也叫JDBC存储过程
2. 使用 Statement 对象。在对数据库只执行一次性存取的时侯,用 Statement 对象进行处理。PreparedStatement 对象的开销比Statement大,对于一次性操作并不会带来额外的好处。
3. statement每次执行sql语句,相关数据库都要执行sql语句的编译,preparedstatement是预编译得,preparedstatement支持批处理

### %11、`索引`以及`索引的实现`(B+树介绍、和B树、R树区别

### ? 12、什么是数据库连接池

MySql数据库连接池专题

参考资料

> Statement和PreparedStatement之间的区别

## 海量数据处理

### %1、海量日志数据，如何提取出某日访问淘宝次数最多的IP

### %2、上亿数据，统计其中出现次数最多的前N个数据

### %3、5亿个int，找出他们的中位数

### %4、两个文件，各存放50亿条URL，每个URL占64字节。内存限制是4G，找出两个文件中相同的URL

### %5、有40亿个不重复的unsigned int的整数，没排过序，现在给一个数，如何快速判断这个数是否在这40亿个数当中。

### ？6、提示：分治、Hash映射、堆排序、双层桶划分、Bloom filter、bitmap、数据库索引、mapreduce

# C语言基础

## 构造函数，析构函数

## C++相关

## 其他

# java基础

## 封装，继承，多态

### ！1、Java中实现多态的机制是什么，动态多态和静态多态的区别

- 多态：就是同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。
- Java中实现多态的机制：动态多态，表现形式为重写。
- 动态多态实现的条件：有类继承或者接口实现、子类要重写父类的方法、父类的引用指向子类的对象。
- 动态多态：在运行期的时候，动态调用类方法。
- 静态多态：在编译期，手动选择调用哪个方法。

### ！2、接口和抽象类的区别，如何选择

- 区别：

  > - 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
  > - 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。
  > - 接口中不能含有静态代码块以及静态方法(用 static 修饰的方法)，而抽象类是可以有静态代码块和静态方法。
  > - 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

- 选择：通常是实现多个接口，在实现过程中把可复用的公用代码抽取成抽象类

### ！3、Java能不能多继承，可不可以多实现

```
Java中不能多继承，可以多实现。
即定义一个类的时候，可以同时实现多个接口，但是不能同时继承多个类。
```

### %4、Static Nested Class 和 Inner Class的不同

- Inner Class 指内部类，Static Nested Class指静态内部类，两者最大的区别就在于是否有指向外部的引用上，前者的创建需要依赖于外部类的实例，后者的创建不依赖于外部类的实例；
- 由于静态对象是默认加载，那么静态内部类会先于外部类被加载到内存中，因此非静态内部类中不能定义静态成员；而静态内部类中既可以有静态成员，也可以有非静态成员，但非静态成员需要静态内部类实例化后才能调用；

### ！5、重载和重写的区别。

- 重载(Overload) 定义：方法重载是指同一个类中的多个方法具有相同的名字，但这些方法具有不同的参数列表，即参数的数量或参数类型不能完全相同。返回值类型可以相同也可以不相同。无法以返回类型作为重载函数的区分标准 规则：
    1. 参数具有不同的参数列表；
    2. 可以有不同的返回类型，只要参数列表不同就可以
    3. 可以有不同的访问修饰符；
    4. 可以抛出不同的异常；
- 重写(Override)： 定义：方法重写是存在子父类之间的,又称为方法覆盖，子类定义的方法与父类中的方法具有相同的方法名字,相同的参数表和相同的返回类型 规则：
    1. 参数列表必须完全与被重写的方法相同；
    2. 返回类型必须与被重写的方法的返回类型相同；
    3. 访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）
    4. 重写方法一定不能抛出新的检查异常或者比被重写方法声明更宽泛的检查性异常；

## ！6、是否可以继承String类

- 回答：不可以继承String类，因为String类是final类型，final类是不能被继承的。
- final类：类不可以继承扩展，变量不可以被修改，方法不可被重写。

  作用：保证平台安全的必要手段。
- String类被设计成final类型的原因：

```
1. 一个String类型的引用一定是String类型的，不会是它的子类。
2. String一旦被创建就不可以被修改，是线程安全的，可以在多个线程之间共享。
3. 高效，字符串的不可改变性会使字符串的hashcode保持不变，所以每次使用hashcode时不用重新计算一次。
4. 设计成final，JVM才不用对相关方法在虚函数表中查询，而直接定位到String类的相关方法上，提高了执行效率。
```

## ！7、构造器是否可被override?

- 回答：构造器不能被override。
- 构造器或构造方法：

> 构造方法是一种特殊的方法，它是一个**与类同名且返回值类型为同名类类型**的方法。对象的创建就是通过构造方法来完成，其功能主要是**完成对象的初始化**。当类实例化一个对象时会自动调用构造方法。构造方法和其他方法一样也可以重载。

如果子类能够继承父类的构造方法，那么在子类的构造方法中就有不同于子类类名的构造方法，所以构造方法不能被继承，所以构造方法不能被重写、覆盖父类的构造方法，并且必须优先调用父类的构造方法。

## ！8、public,protected,private的区别?

1. public ： 公有，该成员变量或者方法对任何地方的所有类或者对象都是可见的。
2. protected：修饰符指定该成员或者方法只能在其自己的包中访问，此外还可以由另一个包中的该类的子类访问。
3. no modifier：默认，没有任何修饰符，该成员变量或者方法只能在其自己的包中访问，其他包不能访问，即便是它的子类
4. private：私有，该成员变量或者方法只能在自己的类中访问。

- 下表显示了对每个修饰符允许的成员的访问权限。

| | 类 | 包 | 子类 | 全部 |
|---|---|---|---|---|
| public | Y | Y | Y | Y |
| protected | Y | Y | Y | N |
| no modifier | Y | Y | N | N |
| private | Y | N | N | N |

### 参考资料

- 构造方法--百度百科
- java中private，public，protected详解
- String类为什么是final的
- Java接口-菜鸟教程
- 重载与重写的区别

> Contributes: zhangyue, Leo
>
> Reviewers : Hollis, Kevin Lee

## 集合相关

### ！1、列举几个Java中Collection类库中的常用类

- Set接口：特点：没有重复元素，可以有最多一个null，无序
  - HashSet

    ```
    - 实现：基于HashMap
    - 特点：没有顺序，用equals()和hashCode()协同判断元素是否存在
    ```

  - LinkedHashSet

    ```
    - 实现：基于LinkedHashMap
    - 特点：有顺序(按照插入顺序
    ```

  - SortedSet接口

    ```
    特点：有顺序(按照元素的大小排序)
    ```

  - TreeSet

    ```
    - 实现：基于TreeMap
    - 特点：排过序的，如果里面的元素是自定义类型，需要指定排序方式
    ```

  - ConcurrentSkipListSet

```
- 实现：基于ConcurrentSkipListMap
- 特点：线程安全，支持并发，有排序(可以手动设置)
```

- List接口
  - LinkedList

```
- 实现：基于双链表，元素由Entry对象维护
- 特点：随机访问性能差，插入删除性能好，非线程安全
```

  - ArrayList

```
- 实现：基于数组，初始长度10，扩展长度1.5倍+1
- 特点：随机访问性能好，插入删除性能差，非线程安全
```

  - Vector

```
- 实现：基于数组，初始长度10(可以手动设置)，扩展长度2倍(可以手动设置)
- 特点：线程安全
```

## ！2、List、Set、Map是否都继承自Collection接口？存储特点分别是什么?

List、Set是，Map不是

- List 保证以某种特定插入顺序来维护元素顺序，即保持插入的顺序，另外元素可以重复
- Set 维持它自己的内部排序，随机访问不具有意义。另外元素不可重复。
- Map 保存key-value值，value可多值,key可为null。

## ！3、ArrayList、LinkedList和Vector之间的区别与联系

- 联系：

```
三者都来自于List，其中vector，ArrayList底层实现是数组，而LinkedList的底层实现是双向链表。
```

- 三者各自特点：

```
1.Vector，ArrayList查找快，增删慢。LinkedList相反，查找慢，增删快；
2.vector是线程安全的。ArrayList，LinkedList是线程不安全的；
```

## ！4、HashMap和Hashtable、TreeMap以及ConcurrentHashMap的区别

- HashTable和HashMap及ConcurrentHashMap是基于散列表实现的，但HashTable（线程安全）已被废弃，大多数场合不被建议使用，建议使用ConcurrentHashMap替代HashTable;TreeMap基于红黑树;
- HashMap适用于增删改查，TreeMap适用于遍历排序；HashMap是无序的，TreeMap实现了NavigableMap接口，支持一系列导航方法，是有序的；
- ConcurrentHashMap被用于代替HashTable,在JDK1.7版本中，ConcurrentHashMap采用了分段锁的设计，相比较于HashTable，拥有更高的性能，但相对应的,ConcurrentHashMap放弃了全局锁，使用了局部锁，使

其无法提供强一致性的保障，如果有强一致性的需求，那么仍然需要是否使用HashTable；

### ！5、Collection 和 Collections的区别

- Collection是集合类的上级接口，继承与他有关的接口主要有List和Set;
- Collections是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作;常用的方法有sort，SynchronizedMap；

### %6、其他的集合类：treeset,linkedhashmap等。

- treeset

```
- TreeSet主要可用于排序，TreeSet是SortedSet的实现类；
- 如果试图把一个对象添加到TreeSet时，则该对象的类必须实现Comparable接　　口，否则程序会抛出异常java.lang.ClassCastException
- 原因：TreeSet集合中添加两个Err对象，添加第一个对象时，TreeSet里没有　　任何元素，所以不会出现任何问题；当添加第二个Err对象时，TreeSet就会　　调用该对象的compareTo(Object obj)方法与集合中的其他元素进行比较–　　如果其对应的类没有实现Comparable 接口，则会引发ClassCastException　　异常。
- TreeSet判断两个对象是否相等的唯一标准是：
    两个对象通过compareTo(Object o)方法比较是否返回0 –如果返回0，　TreeSet则会认为它们相等：否则就认为它们不相等。
```

- linkedhashmap

```
    LinkedHashMap是HashMap的子类，拥有HashMap所有的特性。HashMap和双向链表即是LinkedHashMap，由于LinkedHashMap额外维护了一个双向列表，所以可以保证迭代的顺序（HashMap是无序的），根据链表中元素的顺序可以将LinkedHashMap分为：保持插入顺序的LinkedHashMap 和 保持访问顺序的LinkedHashMap，其中LinkedHashMap的默认实现是按插入顺序排序的。
```

### 参考资料

> Contributes: 木
>
> Reviewers : Hollis, Kevin Lee

## 异常相关

### ！1、Error和Exception的区别

Error类和Exception类的父类都是throwable类，他们的区别是：

- Error类一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。
- Exception类表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

### ！2、异常的类型，什么是运行时异常

Exception类又分为运行时异常（Runtime Exception）和受检查的异常(Checked Exception )：

- 运行时异常;ArithmaticException,IllegalArgumentException，编译能通过，但是一运行就终止了，程序不会处理运行时异常，出现这类异常，程序会终止。

- 受检查的异常，要么用try-catch捕获，要么用throws字句声明抛出，交给它的父类处理，否则编译不会通过。

## ！3、final、finally和finalize的区别

- final：用来修饰类，方法，变量等，被final修饰的类不可以继承扩展，变量不可以修改，方法不可以重写。是保证平台安全的必要手段。
- finally：表示代码块一定要被执行，一般用来关闭JDBC，保证Unlock锁等动作。但是也有例外，在finally执行之前强制退出进程，finally代码块不执行。
- finalize：在垃圾回收器回收对象之前调用的方法以回收特定的垃圾，现在已经不推荐使用，因为finalize执行具有不确定性，有可能造成严重后果。

## %4、try-catch-finally中，如果在catch中return了，finally中的代码还会执行么，原理是什么?

- 回答：还会执行
- 原理：finally 块无论是否捕获或处理异常，finally块里的语句都会被执行。当在try块或catch块中遇到return语句时，finally语句块将在方法返回之前被执行。在以下4种特殊情况下，finally块不会被执行：

  1）在finally语句块中发生了异常。

  2）在前面的代码中用了System.exit()退出程序。

  3）程序所在的线程死亡。

  4）关闭CPU。

## ！5、列举3个以上的RuntimeException

```
NullPointerException - 空指针引用异常
ClassCastException - 类型强制转换异常。
IllegalArgumentException - 传递非法参数异常。
ArithmeticException - 算术运算异常
ArrayStoreException - 向数组中存放与声明类型不兼容对象异常
IndexOutOfBoundsException - 下标越界异常
NegativeArraySizeException - 创建一个大小为负数的数组错误异常
NumberFormatException - 数字格式异常
SecurityException - 安全异常
UnsupportedOperationException - 不支持的操作异常
```

## ！6、Java中的异常处理机制的简单原理和应用

- 在 Java 应用程序中，异常处理机制为：抛出异常，捕捉异常。
- 应用：

```
Try{
    //可能发现异常的语句块
}catch(异常类型,e){
   //发生异常时候的执行语句块
} finnally{
  //不管是否发生异常都执行的语句块

}
```

### 参考资料

- 常见的几种RuntimeException
- Java中处理异常中return关键字
- 深入理解java异常处理机制

> Contributes: zhangyue
>
> Reviewers : Hollis, Kevin Lee

## 其他

**！1、String和StringBuffer、StringBuilder的区别**

- 属性

  String 为字符串常量，是不可变类；

  StringBuffer 和 StringBuilder 为字符串变量，是可变类；

- 线程安全

  String 线程安全

  StringBuffer 线程安全

  StringBuilder 线程不安全

- 速度

  String < StringBuffer < StringBuilder

  String：对 String 对象的操作实际上在不断创建新的对象并且回收旧对象，执行速度很慢；并且通过查看反编译后的字节码，对 String 对象进行加操作，实际上会自动被 JVM 优化成 StringBuilder 对象的 append 操作，并通过 toString 方法返回 String 对象；

  StringBuffer 和 StringBuilder：因为 StringBuffer 是线程安全的，所以在速度上 StringBuffer < StringBuilder；

总结：

String：适用于少量的字符串操作的情况

StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

**！2、==和equals的区别**

在栈中，基本类型存储的是值，引用类型存储的是地址，指向堆中的实例；

无论是基本类型还是引用类型，“== ” 比较的都是值，引用类型的值实际上是地址，所以 “== ” 比较的是值实际上是地址

- 未重写 equals

  不重写 equals ，那么实际上 equals 是继承自 Object 类中的 equals 方法，Object 中的equals 方法也只是返回 “== ” 的结果， 此时 == 和 equals 是没有区别的。

- 重写 equals

重写 equals 后，可以根据自己的需要确定 equals 比较的内容，一般都是比较两个对象的值。此时 “== ” 比较的是地址，equals 比较的是自定义的比较内容；

例子：在 String 中，equals 已经被重写了，并且比较的是 String 对象所表示的字符串常量，所以 equals 比较的是 String 所指的常量值，“==” 比较的仍然是地址；

**%3、hashCode的作用，和equals方法的关系**

- hashCode的作用
  1. hashCode 方法返回对象的哈希码值，在 Object 中，返回对象地址 hash 之后的结果，自定义对象如果不重写 hashCode 方法，默认使用 Object 对象的 hashCode 方法，返回对象地址经过哈希函数计算之后的结果；
  2. .hashCode是为了提高在散列结构存储中查找的效率，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；
- hashCode() 与 equals() 的关系
  1. 若重写了 equals() 方法，则有必要重写 hashCode() 方法；
  2. 若两个对象 equals() 返回 true，则 hashCode() 有必要也返回相同的 int 数；
  3. 若两个对象 equals() 返回 false，则 hashCode() 不一定返回不同的 int 数；
  4. 若两个对象 hashCode() 返回相同的 int 数，则 equals() 不一定返回 true；
  5. 若两个对象 hashCode() 返回不同的 int 数，则 equals() 一定返回 false；
  6. 同一对象在执行期间若已经存储在集合中，则不能修改影响 hashCode 值的相关信息，否则会导致内存泄露问题

**！4、Input/OutputStream和Reader/Writer有什么区别**

```
（1）Reader和Writer类（文本字符流读写类）：提供的对字符流处理的类，它们为抽象类。一般通过其子类来实现。
（2）InputStreamReader(InputStream in) 和OutputStreamWriter(OutputStream out)：它们可以使用指定的编码规范并基于字节流生成对应的字符流。
（3）BufferedReader(InputStreamReader isr, int size) 和 BufferedWriter(OutputStreamWriter osr, int size)：为提高字符流的处理效率，可以采用缓冲机制的流实现对字符流作成批的处理，避免了频繁的从物理设备中读取信息 。
```

**！5、如何在字符流和字节流之间转换?**

```
- InputStreamReader 是字符流Reader的子类，是字节流通向字符流的桥梁。 你可以在构造器重指定编码的方式，如果不指定的话将采用底层操作系统的默认编码方式，例如 GBK 等
- OutputStreamWriter 是字符流Writer的子类，是字符流通向字节流的桥梁。 每次调用 write() 方法都会导致在给定字符（或字符集）上调用编码转换器。在写入底层输出流之前，得到的这些字节将在缓冲区中累积
```

**！6、switch可以使用那些数据类型**

- JDK1.6以前5种， byte、char、short、int、枚举 ，JDK1.7 增加String 共六种
- 在JDK1.5之前,switch循环只支持byte short char int四种数据类型.

  JDK1.5 在switch循环中增加了枚举类与byte short char int的包装类,对四个包装类的支持是因为java编译器在底层手动进行拆箱,而对枚举类的支持是因为枚举类有一个ordinal方法,该方法实际上是一个int类型的数值.

  jdk1.7开始支持String类型,但实际上String类型有一个hashCode算法,结果也是int类型.而byte short char类型可以在不损失精度的情况下向上转型成int类型.所以总的来说,可以认为switch中只支持int.

**%7、Java的四种引用**

- 强引用（Strong Reference）

  我们平时所使用的引用就是强引用。 A a = new A(); 也就是通过关键字new创建的对象所关联的引用就是强引用。 只要强引用存在，该对象永远也不会被回收。

- 软引用（Soft Reference）

  只有当堆即将发生OOM异常时，JVM才会回收软引用所指向的对象。 软引用通过SoftReference类实现。 软引用的生命周期比强引用短一些。

- 弱引用（Weak Reference）

  只要垃圾收集器运行，软引用所指向的对象就会被回收。 弱引用通过WeakReference类实现。 弱引用的生命周期比软引用短。

- 虚引用（Phantom Reference）

  虚引用也叫幽灵引用，它和没有引用没有区别，无法通过虚引用取得一个对象实例。 一个对象关联虚引用唯一的作用就是在该对象被垃圾收集器回收之前会受到一条系统通知。 虚引用通过PhantomReference类来实现。

**！8、序列化与反序列化**

- Java序列化与反序列化

  ```
  Java序列化是指把Java对象转换为字节序列的过程；而Java反序列化是指把字节序列恢复为Java对象的过程。
  ```

- 为什么需要序列化与反序列化

  ```
  在两个进程进行通信时，实现进程之间的对象传递；实现数据持久化，把数据永久地保存到硬盘上（通常存放在文件里）。
  ```

- 如何实现Java序列化与反序列化

  ```
  - 只要一个类实现了java.io.Serializable接口，那么它就可以被序列化。
  - 通过ObjectOutputStream和ObjectInputStream对对象进行序列化及反序列化
  ```

- 相关类与接口

  ```
  - java.io.Serializable接口：标识可序列化的语义
  - java.io.Externalizable接口：继承了Serializable接口 需要重写 writeExternal()与readExternal()方法
  - ObjectOutput接口与ObjectInput接口
  - ObjectOutputStream类与ObjectInputStream类
  ```

**！9、正则表达式**

调研了一下，正则表达式的面试题绝大多数都是以实际的题目,网上的相关学习教程也有很多，这里就不再赘述，仅仅以一片相关面试题为引，作为参考，地址：正则表达式学习笔记

**！10、int和Integer的区别，什么是自动装箱和自动拆箱**

- 区别：

```
1. int是基本数据类型，Integer是int的包装类。
2. int的默认值是0，Integerde默认值是null。
3. int可以直接使用，Integer必须实例化才能使用。
4. int指向实际值，Integer指向的是Integer对象。
```

- 自动装箱和自动拆箱：装箱就是Java把基本数据类型转变为其包装类，编译器通过调用包装类型的valueOf()方法实现自动装箱，调用xxxValue()方法自动拆箱。

**参考资料**

- Java基础之int和Integer有什么区别
- InputStreamReader和OutputStreamWriter的用法
- Java中switch都可以支持哪些数据类型
- Java序列化与反序列化
- 从一道面试题彻底搞懂hashCode与equals的作用与区别及应当注意的细节
- Java中hashCode的作用

> Contributes: Leo, zhangyue, 木
>
> Reviewers : Hollis, Kevin Lee

# java高级

## 多线程

### ！1、进程和线程的区别

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位. 线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位. 通俗一点说，进程就是程序的一次执行，而线程可以理解为进程中的执行的一段程序片段。

### ！2、并行和并发的区别和联系

并发（concurrency）和并行（parallellism）区别：

- 解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。
- 解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。
- 解释三：在一台处理器上“同时”处理多个任务，在多台处理器上同时处理多个任务。如hadoop分布式集群

> 所以并发编程的目标是充分的利用处理器的每一个核，以达到最高的处理性能。

并发（concurrency）和并行（parallellism）联系： 都是同时处理多路请求的意思.

### ！3、同步与异步

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，就得到返回值了。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。

## ！4、多线程的实现方式，有什么区别

继承Thread和实现Runable接口，最重要的一个区别是实现Runable接口避免Java的单继承特性带来的局限；

## ！5、什么叫守护线程

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

> 用户线程即运行在前台的线程，而守护线程是运行在后台的线程。 守护线程作用是为其他前台线程的运行提供便利服务，而且仅在普通、非守护线程仍然运行时才需要， 比如垃圾回收线程就是一个守护线程。当VM检测仅剩一个守护线程，而用户线程都已经退出运行时，VM就会退出，因为没有如果没有了被守护这，也就没有继续运行程序的必要了。 如果有非守护线程仍然存活，VM就不会退出。

> 守护线程并非只有虚拟机内部提供，用户在编写程序时也可以自己设置守护线程。用户可以用Thread的setDaemon（true）方法设置当前线程为守护线程。

> 虽然守护线程可能非常有用，但必须小心确保其他所有非守护线程消亡时，不会由于它的终止而产生任何危害。因为你不可能知道在所有的用户线程退出运行前，守护线程是否已经完成了预期的服务任务。 一旦所有的用户线程退出了，虚拟机也就退出运行了。 因此，不要在守护线程中执行业务逻辑操作（比如对数据的读写等）。

> 另外有几点需要注意：

- setDaemon(true)必须在调用线程的start（）方法之前设置，否则会跑出IllegalThreadStateException异常。
- 在守护线程中产生的新线程也是守护线程。
- 不要认为所有的应用都可以分配给守护线程来进行服务，比如读写操作或者计算逻辑。

## %6、如何停止一个线程?

调用stop,interrupt，或者设置标志位，代码根据标志位退出线程。

## ！7、什么是线程安全?

保证在多线程的环境下，数据不会被污染。

## ！8、synchronized 和 lock的区别

synchronized不需要显示的释放锁，程序结束时会自动释放，lock需要手动调用unlock释放. lock的在读写多的情况下性能更好，因为提供了读写锁，读读不互斥.

## ！9、当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?

在锁没有释放之前，其他线程不可以进入此方法

## ！10、启动一个线程是用run()还是start()?

start(),调用run方法相当于一个普通方法的调用，不会开启一个子线程。

## ！12、wait和sleep的区别

- sleep是Thread类的方法, wait是Object类中定义的方法
- sleep 方法可能抛出一个InterruptedException,这是一个必检异常。当一个休眠线程的interrupt()方法被调用时，就会发生这个异常。
  因为java强制捕获必检的异常，所以，必须将他放到try-catch块中。如果在一个循环中调用了sleep方法，那

就应该将这个循环放在
try-catch 块中，如果循环在try-catch块外，即使线程被中断，它也可能会继续执行。而wait，notify和notifyAll不需要捕获异常。

- 监视器（monitor)是一个相互排斥且具备同步能力的对象。监视器中的一个时间点上，只能有一个线程执行一个方法。 线程通过获取监视器上的锁进入监视器，
  并且通过释放锁退出监视器。任意对象都可能是一个监视器。一旦一个线程锁住对象，该对象就成为监视器。 在执行同步方法或块之前，线程必须获取锁。
  Thread.sleep()和Object.wait()都会暂停当前的线程, sleep() 不会释放monitor ,调用wait()方法可以释放monitor, 调用wait后，需要别的线程执行notify/notifyAll
  来通知一个或所有的等待线程重新获取锁并且恢复执行。
- waite()和notify()因为会对对象的"锁标志"进行操作，所以它们必须在synchronized函数或synchronized block中进行调用。如果在non-synchronized函数或
  non-synchronizedblock中进行调用，虽然能编译通过，但在运行时会发生illegalMonitorStateException的异常

## %13、notify和notifyAll的区别

> 锁池和等待池

- 锁池: 假设线程A已经拥有了某个对象(注意:不是类)的锁，而其它的线程想要调用这个对象的某个synchronized方法(或者synchronized块)，
  由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权， 但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中。
- 等待池:假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁后，进入到了该对象的等待池中

> 然后再来说notify和notifyAll的区别

1. 如果线程调用了对象的 wait()方法， 那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。
2. 当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，
   锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，
3. 等待锁竞争优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法， 它才会重新回到等待池中。
   而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

> 综上，所谓唤醒线程，另一种解释可以说是将线程由等待池移动到锁池 notifyAll调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行， 如果不成功则留在锁池等待锁被释放后再次参与竞争。而notify只会唤醒一个线程。有了这些理论基础，后面的notify可能会导致死锁，而notifyAll则不会的例子也就好解释了

notifyAll使所有原来在该对象上等待被notify的线程统统退出wait的状态，变成等待该对象上的锁，一旦该对象被解锁，他们就会去竞争。 notify则文明得多他只是选择一个wait状态线程进行通知，并使它获得该对象上的锁，但不惊动其他同样在等待被该对象notify的线程们， 当第一个线程运行完毕以后释放对象上的锁此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程由 于没有得到该对象的通知，继续处在wait状态，直到这个对象发出一个notify或notifyAll

## %14、线程池的作用

减少频繁创建线程和切换，销毁带来的性能开销；

### %15、Java中线程池相关的类

列举几个常用的线程池： 1.newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程 2.newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待 3.newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行 4.newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行

### 参考资料

1. java中的锁

> Contributes: hueizhe
>
> Reviewers : Hollis, Kevin Lee

## jvm底层技术

### ！ 1、gc的概念，如果A和B对象循环引用，是否可以被GC?

在正式回答这个问题之前先简单说说 Java运行时内存区，划分为线程私有区和线程共享区：

1. 线程私有区：
    - 程序计数器，记录正在执行的虚拟机字节码的地址；
    - 虚拟机栈：方法执行的内存区，每个方法执行时会在虚拟机栈中创建栈帧；
    - 本地方法栈：虚拟机的Native方法执行的内存区；
2. 线程共享区：
    - Java堆：对象分配内存的区域，这是垃圾回收的主战场；
    - 方法区：存放类信息、常量、静态变量、编译器编译后的代码等数据，另外还有一个常量池。当然垃圾回收也会在这个区域工作。

目前虚拟机基本都是采用可达性算法，

为什么不采用引用计数算法呢？下面就说说引用计数法是如何统计所有对象的引用计数的，再对比分析可达性算法是如何解决引用技术算法的不足。

先简单说说这两个算法：

- 引用计数算法：
  每个对象有一个引用计数器，当对象被引用一次则计数器加1，当对象引用失效一次则计数器减1，对于计数器为0的对象意味着是垃圾对象，可以被GC回收。
- 可达性算法 (GC Roots Tracing)：从GC Roots作为起点开始搜索，那么整个连通图中的对象便都是活对象，对于GC Roots无法到达的对象便成了垃圾回收的对象，随时可被GC回收。

采用引用计数算法的系统只需在每个实例对象创建之初，通过计数器来记录所有的引用次数即可。 而可达性算法，则需要再次GC时，遍历整个GC根节点来判断是否回收。下面通过一段代码来对比说明：

```
public class GcDemo {

    public static void main(String[] args) {
        //分为6个步骤
```

```
        GcObject obj1 = new GcObject(); //Step 1
        GcObject obj2 = new GcObject(); //Step 2

        obj1.instance = obj2; //Step 3
        obj2.instance = obj1; //Step 4

        obj1 = null; //Step 5
        obj2 = null; //Step 6
    }
}

class GcObject{
    public Object instance = null;
}
```

很多文章以及Java虚拟机相关的书籍，都会告诉你如果采用引用计数算法，上述代码中obj1和obj2指向的对象已经不可能再被访问，彼此互相引用对方导致引用计数都不为0，最终无法被GC回收，而可达性算法能解决这个问题。

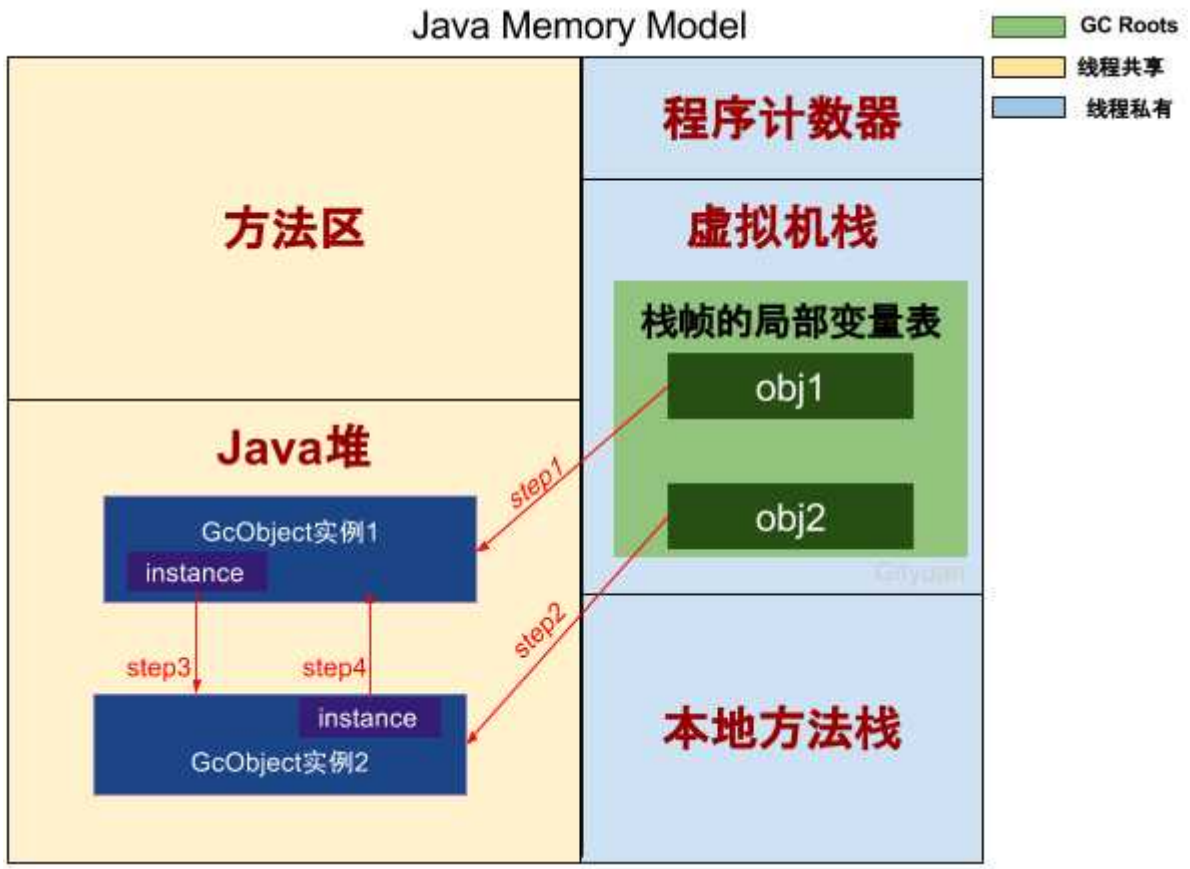
但这些文章和书籍并没有真正从内存角度来阐述这个过程是如何统计的，很多时候大家都在相互借鉴、翻译，却也都没有明白。或者干脆装作讲明白，或者假定读者依然明白。

其实很多人并不明白为什么引用计数法不为0，引用计数到底是如何维护所有对象引用的，可达性是如何可达的?

接下来结合实例，从Java内存模型以及数学的图论知识角度来说明，希望能让大家彻底明白该过程。

> 引用计数算法

如果采用的是引用计数算法：




再回到前面代码GcDemo的main方法共分为6个步骤：

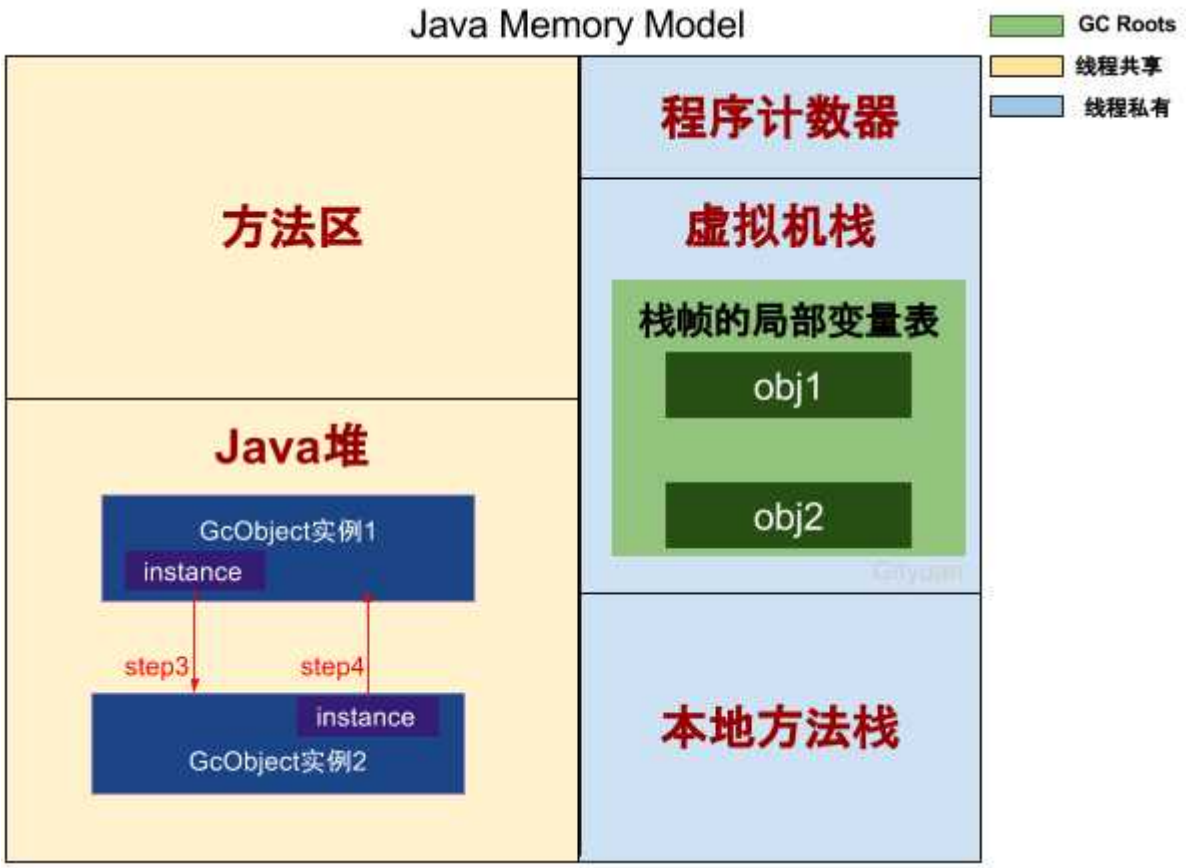
Step1：GcObject实例1的引用计数加1，实例1的引用计数=1；

Step2：GcObject实例2的引用计数加1，实例2的引用计数=1；

Step3：GcObject实例2的引用计数再加1，实例2的引用计数=2；

Step4：GcObject实例1的引用计数再加1，实例1的引用计数=2；

执行到Step 4，则GcObject实例1和实例2的引用计数都等于2。 接下来继续结果图：




Step5：栈帧中obj1不再指向Java堆，GcObject实例1的引用计数减1，结果为1； Step6：栈帧中obj2不再指向Java堆，GcObject实例2的引用计数减1，结果为1。

到此，发现GcObject实例1和实例2的计数引用都不为0，那么如果采用的引用计数算法的话，那么这两个实例所占的内存将得不到释放，这便产生了内存泄露。

> 可达性算法

这是目前主流的虚拟机都是采用GC Roots Tracing算法，比如Sun的Hotspot虚拟机便是采用该算法。

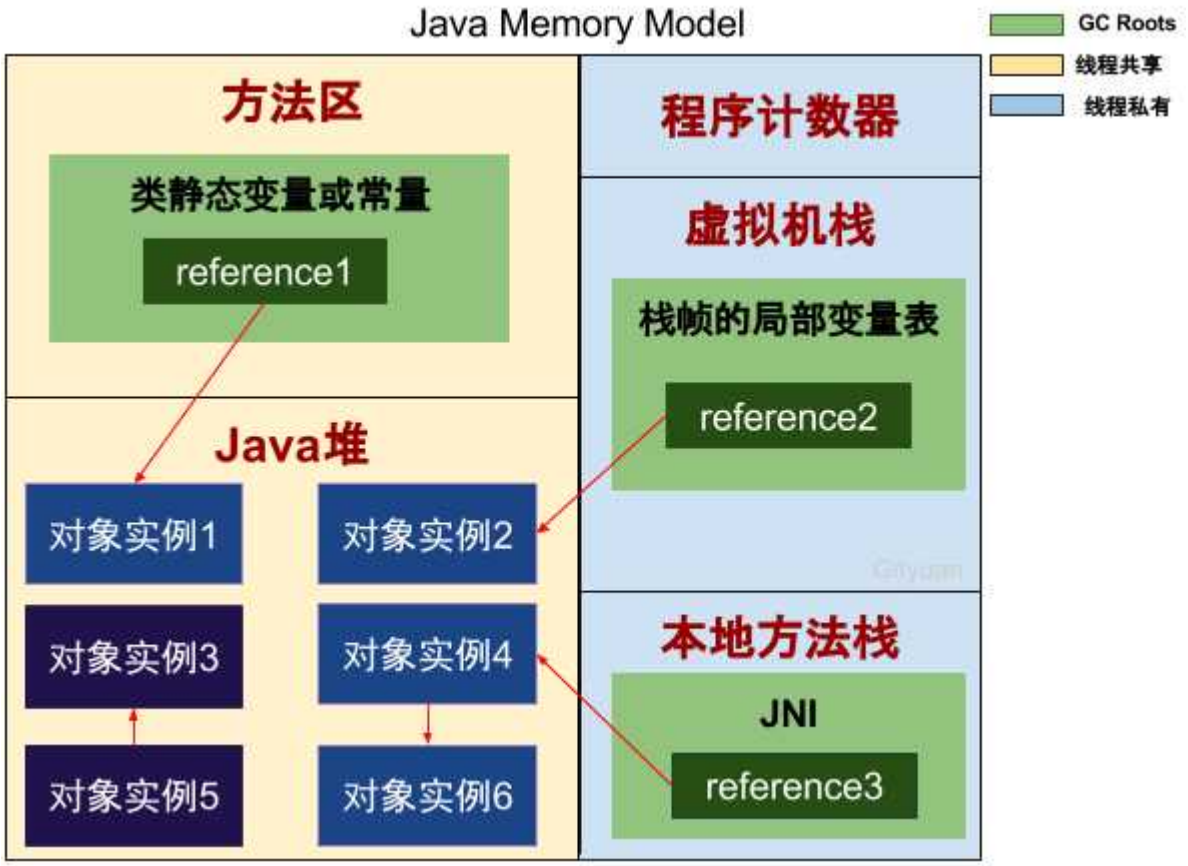
该算法的核心算法是从GC Roots对象作为起始点，利用数学中图论知识，图中可达对象便是存活对象，而不可达对象则是需要回收的垃圾内存。

这里涉及两个概念，一是GC Roots，一是可达性。

那么可以作为GC Roots的对象（见下图）：

- 虚拟机栈的栈帧的局部变量表所引用的对象；
- 本地方法栈的JNI所引用的对象；
- 方法区的静态变量和常量所引用的对象；

关于可达性的对象，便是能与GC Roots构成连通图的对象，如下图：




从上图，reference1、reference2、reference3都是GC Roots，

可以看出：

- reference1-> 对象实例1；
- reference2-> 对象实例2；
- reference3-> 对象实例4；
- reference3-> 对象实例4 -> 对象实例6；

可以得出对象实例1、2、4、6都具有GC Roots可达性，也就是存活对象，不能被GC回收的对象。

而对于对象实例3、5直接虽然连通，但并没有任何一个GC Roots与之相连，这便是GC Roots不可达的对象， 这就是GC需要回收的垃圾对象。

到这里，相信大家应该能彻底明白引用计数算法和可达性算法的区别吧。

再回过头来看看最前面的实例，GcObject实例1和实例2虽然从引用计数虽然都不为0，但从可达性算法来看，都是GC Roots不可达的对象。

> 总之，对于对象之间循环引用的情况，引用计数算法，则GC无法回收这两个对象，而可达性算法则可以正确回收。

### %2、jvm gc如何判断对象是否需要回收，有哪几种方式?

见问题1

### ！3、Java中能不能主动触发GC

答案是可以。

Java的公有API可以主动调用GC的有两种办法，

一个是

```
System.gc();
// 或者下面，两者等价
Runtime.getRuntime().gc()
```

还有一个是JMX：

```
java.lang.management.MemoryMXBean.gc()
```

规范是通知虚拟机尽快执行，没有强制规定执行时间

但是一般不要主动去调用GC，System.gc主动进行垃圾回收时一个非常危险的动作。因为它要停止所有的响应，才能检查内存中是否有可回收的对象，这对一个应用系统风险极大。
如果一个Web应用，所有的请求都会暂停，等待垃圾回收器执行完毕，若此时堆内存（Heap）中的对象少的话则可以接受，一旦对象较多（现在的Web项目越做越大，框架工具越来越多，加载到内存中的对象就更多了），这个过程非常耗时，可能是0.01秒，也可能是1秒，甚至可能是20秒，这就会严重影响到业务的正常运行。

## ！4、JVM的内存结构，堆和栈的区别

jvm 内存结构

- 栈：存放局部变量
- 堆：存放对象以及数组的数据，
- 方法区：被虚拟机加载的类信息、常量、静态常量等。
- 程序计数器(和系统相关)
- 本地方法栈

堆和栈的区别

Java把内存划分成两种：一种是堆内存，一种是栈内存。

- 堆：
    - 主要用于存储实例化的对象，数组。由JVM动态分配内存空间。一个JVM只有一个堆内存，线程是可以共享数据的。
    - 堆内存用来存放由new创建的对象和数组。
    - 在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理
    - 如果堆内存没有可用的空间存储生成的对象，JVM会抛出java.lang.OutOfMemoryError
    - -Xms选项可以设置堆的开始时的大小，-Xmx选项可以设置堆的最大值
- 栈：
    - 主要用于存储局部变量和对象的引用变量，每个线程都会有一个独立的栈空间，所以线程之间是不共享数据的。
    - 基本类型的变量和对象的引用变量都在函数的栈内存中分配
    - 当在一段代码块定义一个变量时，Java就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用
    - 如果栈内存没有可用的空间存储方法调用和局部变量，JVM会抛出java.lang.StackOverFlowError
    - 栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。如果递归没有及时跳出，很可能发生StackOverFlowError问题
    - -Xss选项设置栈内存的大小

## ！5、JVM堆的分代

堆内存同样被划分成了多个区域：

- JVM堆（Heap）= 新生代（Young） + 旧生代（Tenured）
- 新生代（Young）= Eden区 + Survivor区

不同区域的存放的对象拥有不同的生命周期：

- 新建（New）或者短期的对象存放在Eden区域；
- 幸存的或者中期的对象将会从Eden区域拷贝到Survivor区域；
- 始终存在或者长期的对象将会从Survivor拷贝到Old Generation；

生命周期来划分对象，可以消耗很短的时间和CPU做一次小的垃圾回收（GC）。原因是跟C一样，内存的释放（通过销毁对象）通过2种不同的GC实现：Young GC、Full GC。

为了检查所有的对象是否能够被销毁，Young GC会标记不能销毁的对象，经过多次标记后，对象将会被移动到老年代中

## %6、Java中的内存溢出是什么，和内存泄露有什么关系

Java内存泄漏就是程序中动态分配内存给一些临时对象，但是对象不会被GC所回收，它始终占用内存。即被分配的对象可达但已无用, Java内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。 从定义上可以看出内存泄露是内存溢出的一种诱因，不是唯一因素。

一、内存泄漏的几种场景 1、长生命周期的对象持有短生命周期对象的引用

> 这是内存泄露最常见的场景，也是代码设计中经常出现的问题。
> 例如：在全局静态map中缓存局部变量，且没有清空操作，随着时间的推移，这个map会越来越大，造成内存泄露。

2、修改hashset中对象的参数值，且参数是计算哈希值的字段

> 当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段，否则对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中删除当前对象，造成内存泄露。

3、机器的连接数和关闭时间设置

> 长时间开启非常耗费资源的连接，也会造成内存泄露。

二、内存溢出的几种场景

1、堆内存溢出（outOfMemoryError：java heap space） 在jvm规范中，堆中的内存是用来生成对象实例和数组的。 如果细分，堆内存还可以分为年轻代和年老代，年轻代包括一个eden区和两个survivor区。 当生成新对象时，内存的申请过程如下： a、jvm先尝试在eden区分配新建对象所需的内存； b、如果内存大小足够，申请结束，否则下一步； c、jvm启动youngGC，试图将eden区中不活跃的对象释放掉，释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区； d、Survivor区被用来作为Eden及old的中间交换区域，当OLD区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区； e、 当OLD区空间不够时，JVM会在OLD区进行full GC； f、full GC后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对

象，导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory错误”： outOfMemoryError：java heap space

代码举例：

```
/**
 * 堆内存溢出
 *
 * jvm参数: -Xms5m -Xmx5m -Xmn2m -XX:NewSize=1m
 *
 */
public class MemoryLeak {

    private String[] s = new String[1000];

    public static void main(String[] args) throws InterruptedException {
        Map<String,Object> m =new HashMap<String,Object>();
        int i =0;
        int j=10000;
        while(true){
            for(;i<j;i++){
                MemoryLeak memoryLeak = new MemoryLeak();
                m.put(String.valueOf(i), memoryLeak);
            }
        }
    }
}
```

2、方法区内存溢出（outOfMemoryError：permgem space） 在jvm规范中，方法区主要存放的是类信息、常量、静态变量等。 所以如果程序加载的类过多，或者使用反射、gclib等这种动态代理生成类的技术，就可能导致该区发生内存溢出，一般该区发生内存溢出时的错误信息为： outOfMemoryError：permgem space

举例：

```
jvm参数: -XX:PermSize=2m -XX:MaxPermSize=2m
将方法区的大小设置很低即可，在启动加载类库时就会出现内存不足的情况
```

3、线程栈溢出（java.lang.StackOverflowError） 线程栈时线程独有的一块内存结构，所以线程栈发生问题必定是某个线程运行时产生的错误。 一般线程栈溢出是由于递归太深或方法调用层级过多导致的。 发生栈溢出的错误信息为： java.lang.StackOverflowError

代码举例：

```
/**
 * 线程操作栈溢出
 *
 * 参数: -Xms5m -Xmx5m -Xmn2m -XX:NewSize=1m -Xss64k
 *
 */
public class StackOverflowTest {
```

```
    public static void main(String[] args) {
        int i =0;
        digui(i);
    }

    private static void digui(int i){
        System.out.println(i++);
        String[] s = new String[50];
        digui(i);
    }

}
```

最后说一些建议：

- 使用字符串处理，避免使用String，应大量使用StringBuffer，每一个String对象都得独立占用内存一块区域
- 尽量少用静态变量，因为静态变量存放在永久代（方法区），永久代基本不参与垃圾回收
- 避免在循环中创建对象
- 开启大型文件或从数据库一次拿了太多的数据很容易造成内存溢出，所以在这些地方要大概计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。

## ！7、Java的类加载机制，什么是双亲委派

- 什么是类的加载

  类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。

  类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

  类加载器并不需要等到某个类被“首次主动使用”时再加载它，JVM规范允许类加载器在预料某个类将要被使用时就预先加载它，

  如果在预先加载的过程中遇到了.class文件缺失或存在错误，类加载器必须在程序首次主动使用该类时才报告错误（LinkageError错误）如果这个类一直没有被程序主动使用，那么类加载器就不会报告错误

- 加载.class文件的方式
  - 从本地系统中直接加载
  - 通过网络下载.class文件
  - 从zip，jar等归档文件中加载.class文件
  - 从专有数据库中提取.class文件
  - 将Java源文件动态编译为.class文件
- JVM类加载机制
  - 全盘负责，当一个类加载器负责加载某个Class时，该Class所依赖的和引用的其他Class也将由该类加载器负责载入，除非显示使用另外一个类加载器来载入
  - 父类委托，先让父类加载器试图加载该类，只有在父类加载器无法加载该类时才尝试从自己的类路径中加载该类
  - 缓存机制，缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效
- 双亲委派

- 双亲委派模型的工作流程是：

  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，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

- 双亲委派机制:

  1. 当AppClassLoader加载一个class时，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器ExtClassLoader去完成。
  2. 当ExtClassLoader加载一个class时，它首先也不会自己去尝试加载这个类，而是把类加载请求委派给BootStrapClassLoader去完成。
  3. 如果BootStrapClassLoader加载失败（例如在$JAVA_HOME/jre/lib里未查找到该class），会使用ExtClassLoader来尝试加载；
  4. 若ExtClassLoader也加载失败，则会使用AppClassLoader来加载，如果AppClassLoader也加载失败，则会报出异常ClassNotFoundException。

- ClassLoader源码分析：

```
public Class<?> loadClass(String name)throws ClassNotFoundException {
            return loadClass(name, false);
    }

    protected synchronized Class<?> loadClass(String name, boolean resolve)throws
ClassNotFoundException {
            // 首先判断该类型是否已经被加载
            Class c = findLoadedClass(name);
            if (c == null) {
                //如果没有被加载，就委托给父类加载或者委派给启动类加载器加载
                try {
                    if (parent != null) {
                         //如果存在父类加载器，就委派给父类加载器加载
                        c = parent.loadClass(name, false);
                    } else {
                    //如果不存在父类加载器，就检查是否是由启动类加载器加载的类，通过调用本地
方
                    //法native Class findBootstrapClass(String name)
                        c = findBootstrapClass0(name);
                    }
                } catch (ClassNotFoundException e) {
                 // 如果父类加载器和启动类加载器都不能完成加载任务，才调用自身的加载功能
                    c = findClass(name);
                }
            }
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
```

- 双亲委派模型意义：
    - 系统类防止内存中出现多份同样的字节码
    - 保证Java程序安全稳定运行

### ！ 8、ClassLoader的类加载方式

- 隐式加载：不通过在代码里调用ClassLoader来加载需要的类，而是通过JVM来自动加载需要的类到内存，例如：当类中继承或者引用某个类时，JVM在解析当前这个类不在内存中时，就会自动将这些类加载到内存中。
- 显示加载：在代码中通过ClassLoader类来加载一个类，例如调用this.getClass.getClassLoader().loadClass()或者Class.forName()。

### 参考资料


1. gc的概念，如果A和B对象循环引用，是否可以被GC
2. JAVA的内存模型及结构
3. java oracle jvm docs
4. Java堆和栈的区别和介绍
5. 在Java中如何主动调用GC
6. java面试题-基础知识
7. JVM（1）：Java 类的加载机制


> Contributes: hueizhe
>
> Reviewers : Hollis, Kevin Lee

## IO

### ！ 1、NIO、AIO和BIO 之间的区别

- BIO：同步阻塞IO模式，必须等待这件事情做完了才去做下一件事情
- AIO：异步非阻塞IO模式，不用等待这件事情做完了才去做下一件事，这件事情做完了就会自动告诉我他做完了
- NIO：同时支持阻塞与非阻塞模式

### ？ 2、`IO`和`NIO`常用用法

## 其他

### ？ 1、hashcode 有哪些算法

- 加法Hash
- 位运算Hash
- 乘法Hash
- 除法Hash
- 查表Hash
- 混合Hash

详见 hash.md

## ！2、反射的基本概念，反射是否可以调用私有方法

- 定义： 主要是指程序可以访问，检测和修改它本身状态或行为的一种能力，并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关的语义
- 可以调用私有方法

```
Class c = Class.forName("User");
//获取id属性
Field idF = c.getDeclaredField("id");
//实例化这个类赋给o
Object o = c.newInstance();
//打破封装
idF.setAccessible(true); //使用反射机制可以打破封装性，导致了java对象的属性不安全。
//给o对象的id属性赋值"110"
idF.set(o, "110"); //set
```

## ！3、Java中范型的概念

- 泛型程序设计（generic programming）是程序设计语言的一种风格或范式。
- 泛型允许程序员在强类型程序设计语言中编写代码时使用一些以后才指定的类型，在实例化时作为参数指明这些类型。
- 泛型的定义主要有以下两种：
    - 在程序编码中一些包含类型参数的类型，也就是说泛型的参数只可以代表类，不能代表个别对象。（这是当今较常见的定义）
    - 在程序编码中一些包含参数的类。其参数可以代表类或对象等等。（现在人们大多把这称作模板）
      不论使用那个定义，泛型的参数在真正使用泛型时都必须作出指明。
- Java 泛型的参数只可以代表类，不能代表个别对象。由于Java泛型的类型参数之实际类型在编译时会被消除，所以无法在运行时得知其类型参数的类型，而且无法直接使用基本值类型作为泛型类型参数。Java编译程序在编译泛型时会自动加入类型转换的编码，故运行速度不会因为使用泛型而加快。
- 由于运行时会消除泛型的对象实例类型信息等缺陷经常被人诟病，Java及JVM的开发方面也尝试解决这个问题，例如Java通过在生成字节码时添加类型推导辅助信息，从而可以通过反射接口获得部分泛型信息。通过改进泛型在JVM的实现，使其支持基本值类型泛型和直接获得泛型信息等。
- Java允许对个别泛型的类型参数进行约束，包括以下两种形式（假设T是泛型的类型参数，C是一般类、泛类，或是泛型的类型参数）：
    - T实现接口I。
    - T是C，或继承自C

## ？4、JVM启动参数，-Xms和 -Xmx

| 参数名称 | 含义 | 默认值 | 说明 |
| --- | --- | --- | --- |
| -Xms | 初始堆大小 | 物理内存的1/64(<1GB) | 默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制. |
| -Xmx | 最大堆大小 | 物理内存的1/4(<1GB) | 默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制 |

## %5、代理机制的实现

JAVA 代理实现 代理的实现分动态代理和静态代理，

- 静态代理的实现是对已经生成了的JAVA类进行封装。
- 动态代理则是在运行时生成了相关代理类，在JAVA中生成动态代理一般有两种方式:

### 1. JDK自带实现方法

JDK实现代理生成，是用类 java.lang.reflect.Proxy, 实现方式如下:

```
public class JDKProxy {
      public static Object getPoxyObject(final Object c) {
            // JDK实现动态代理，但JDK实现必须需要接口
            return Proxy.newProxyInstance(c.getClass().getClassLoader(),
c.getClass().getInterfaces(),
                        new InvocationHandler() {
                              public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                                    Object reObj = null;
                                    System.out.print("you say: ");
                                    reObj = method.invoke(c, args);
                                    System.out.println(" [" +
Calendar.getInstance().get(Calendar.HOUR) + ":"
                                                + Calendar.getInstance().get(Calendar.MINUTE) +
" "
                                                + Calendar.getInstance().get(Calendar.SECOND) +
"]");
                                    return reObj;
                              }
                        });
      }
}
```

测试代理类方法

```
public class TestForPoxy {
      public static void main(String[] args) {
            ServiceTest service = new ServiceTestImpl();
            System.out.println(service.getClass().getSimpleName());
            ServiceTest poxyService = (ServiceTest) JDKProxy.getPoxyObject(service);
            System.out.println(poxyService.getClass().getSuperclass());
            poxyService.saySomething("hello,My QQ code is 107966750.");
            poxyService.saySomething("what 's your name?");
            poxyService.saySomething("only for test,hehe.");
      }
}
```

- Proxy实现代理的目标类必须有实现接口
- 生成出来的代理类为接口实现类，和目标类不能进行转换，只能转为接口实现类进行调用
- 明显特点：通过此方法生成出来的类名叫做 $Proxy0

**2. 用CGLIB包实现**

- CGLIB实现方式是对代理的目标类进行继承
- 生成出了的代理类可以没方法，生成出来的类可以直接转换成目标类或目标类实现接口的实现类

**3. Spring AOP 的代理类机制**

Spring AOP中，当拦截对象实现了接口时，生成方式是用JDK的Proxy类。当没有实现任何接口时用的是GCLIB开源项目生成的拦截类的子类.

### ！ 6、String s = new String("s")，创建了几个对象。

这个产生了2个对象，

一个是new关键字创建的new Sring（）；

另一个是"s"对象，abc在一个字符串池中s这个对象指向这个串池

### 参考资料

> Contributes: hueizhe
>
> Reviewers : Hollis, Kevin Lee

# java Web

## servlet

### ！ 1、JSP和Servlet的区别，Servelt的概念。

**1.1) JSP和Servlet的区别**

- 二者联系：

```
- JSP在本质上就是SERVLET。


- JSP会被Web容器转译为Servlet的“.java”源文件，编译为“.class”文件，然后加载容器之中，所以最后提供服务的
```

```
还是Servlet实例（Instance）。
```

- 主要区别：

```
- Servlet在Java代码中通过HttpServletResponse对象动态输出HTML内容
- JSP在静态HTML内容中嵌入Java代码，Java代码被动态执行后生成HTML内容
```

- 各自的优缺点：

```
- Servlet能够很好地组织业务逻辑代码，但是在Java源文件中通过字符串拼接的方式生成动态HTML内容会导致代码维护困难、可读性差
- JSP虽然规避了Servlet在生成HTML内容方面的劣势，但是在HTML中混入大量、复杂的业务逻辑同样也是不可取的
```

- 解决方案：

```
- 使用MVC模式规避JSP与Servlet各自的短板，提高代码的可读性和可维护性；
- Servlet只负责业务逻辑而不会通过 out.append()动态生成HTML代码；
- JSP中也不会充斥着大量的业务代码。
```

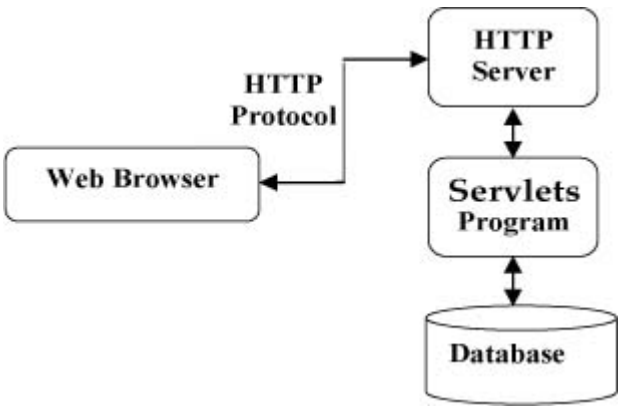
**1.2) Servlet的概念**

- Servlet是什么？

```
Java Servlet 是运行在 Web 服务器或应用服务器上的程序。
```

- Servlet在架构中的位置：

```
Web 浏览器或其他 HTTP 客户端的请求
中间层（servlet）
HTTP 服务器上的数据库或应用程序
```




- Servlet作用：

```
- 读取客户端请求：
-- 显式数据：包括网页上的HTML 表单，applet 或自定义的HTTP 客户端程序的表单。
-- 隐式数据：包括 cookies、媒体类型和浏览器能理解的压缩格式等等。

- 处理数据并生成结果，这个过程可能需要：
-- 访问数据库
-- 执行 RMI 或 CORBA 调用
-- 调用 Web 服务
-- 直接计算

- 发送数据到客户端（浏览器）：
-- 显式文档：格式包括文本文件（HTML 或 XML）、二进制文件（GIF 图像）、Excel 等。
-- 隐式响应：包括告诉浏览器或其他客户端被返回的文档类型（例如 HTML），设置 cookies 和缓存参数，以及其他类似的任务。
```

**参考来源：**

a) Jsp和Servlet有什么区别?
b) jsp与servlet有什么区别?
c) Servlet、Filter、Listener深入理解
d) Servlet 简介(w3cschool)
e) 《JSP&Servlet学习笔记(第2版)》作者林信良

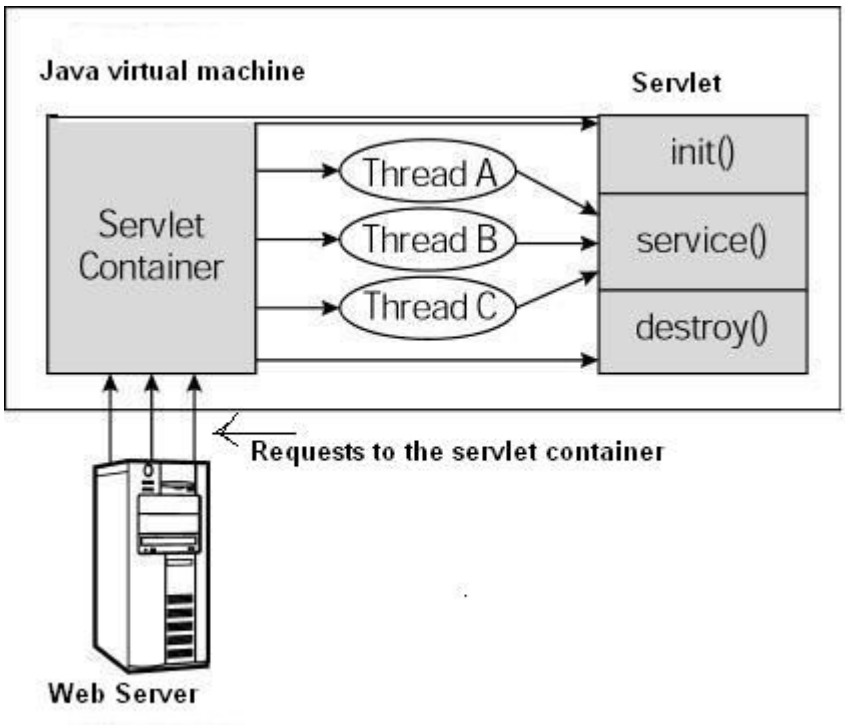
## ！2、Servlet的生命周期

### 2.1) Servlet 生命周期描述

```
Servlet 生命周期可被定义为从创建直到毁灭的整个过程
- Servlet 通过调用 init() 方法进行初始化
- Servlet 调用 service() 方法来处理客户端的请求
- Servlet 通过调用 destroy() 方法终止（结束）
- Servlet 最终由 JVM 的垃圾回收器进行垃圾回收
```

### 2.2) 一个典型的 Servlet 生命周期方案

```
- 第一个到达服务器的 HTTP 请求被委派到 Servlet 容器
- Servlet 容器在调用 service() 方法之前加载 Servlet
- 然后 Servlet 容器处理由多个线程产生的多个请求，每个线程执行一个单一的Servlet实例的 service() 方法
```




**参考来源：**

a) Servlet 生命周期(w3cSchool)

## ！3、Servlet中的session工作原理，以及设置过期时间的方式

### 3.1) Session工作原理

- 一句话概述

```
每个HttpSession都有一个唯一的Session ID，
当浏览器访问应用程序时，会将Cookie中存放的Session ID一并发送给应用程序，
Web容器会根据Session ID找出对应的HttpSession对象，这样就能在不同请求中获取相同的数据。
```

- 为什么使用Session

```
Web应用程序的请求与响应基于Http，为无状态的通信协议，每次请求对服务器来说都是新请求。
```

- HttpSession对象：

```
当用户使用浏览器访问服务器的资源时，通过运行HttpServletRequest对象的 getSession()方法，Web容器就会获取
已经存在的HttpSession实例或创建一个新HttpSession实例。
由于Web容器本身是执行于JVM中的一个Java程序，HttpSession是Web容器中的一个Java对象。
```

- Session ID：

每个HttpSession对象都有一个特殊的Session ID作为标识，可以通过执行HttpSession的 getId()方法取得这个Session ID。Session ID 默认使用Cookie存放在浏览器中。在Tomcat中，Cookie的名称是JSESSIONID（在PHP中为PHPSESSID）

#### 3.2) Session设置超时的方式

- 方式一：

在Servlet中执行HttpSession的 setMaxInactiveInterval()方法，参数单位是“秒”；

- 方式二：

```
<!-- 在程序中的web.xml里设置HttpSession默认失效时间，单位是“分钟” -->
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

- 方式三：

```
<!-- 在Tomcat的/conf/web.xml中session-config，单位也是分钟 -->
<session-config>
     <session-timeout>30</session-timeout>
</session-config>
```

- 注意：

默认关闭浏览器马上失效的是浏览器上的Cookie，不是HttpSession。
存在Cookie中的Session ID随着Cookie失效而丢失，
所以再次打开浏览器向服务器发送请求时，HttpSession尝试 getSession()，Web容器会产生新的HttpSession实例。
要让HttpSession立即失效必须调用 invalidate()方法，
否则HttpSession实例要等到失效时间过后才会被容器销毁回收

**参考来源：**

a) Servlet中不可不知的Session技术
b) Java Web开发Session超时设置
c) 《JSP&Servlet学习笔记(第2版)》作者林信良

## ！4、Servlet中，filter的应用场景有哪些?

#### 4.1) Filter应用场景

- 统一请求或响应的字符集

如将编码统一为UTF-8

- 替换特殊字符

如将请求参数中的一些HTML标签去掉

- 控制浏览器缓存页面中的静态资源

有些动态页面中引用了一些图片或css文件以修饰页面效果，
这些图片和css文件经常是不变化的，
所以为减轻服务器的压力，
可以使用filter控制浏览器缓存这些文件，
以提升服务器的性能。

- 使用Filter实现URL级别的权限认证

在实际开发中我们经常把一些执行敏感操作的servlet映射到一些特殊目录中，
并用filter把这些特殊目录保护起来，
限制只能拥有相应访问权限的用户才能访问这些目录下的资源。
从而在我们系统中实现一种URL级别的权限功能。

- 实现用户自动登陆

首先，在用户登陆成功后，
发送一个名称为user的cookie给客户端，
cookie的值为用户名和md5加密后的密码。
编写一个AutoLoginFilter，
这个filter检查用户是否带有名称为user的cookie，
如果有，则调用dao查询cookie的用户名和密码是否和数据库匹配，
匹配则向session中存入user对象（即用户登陆标记），
以实现程序完成自动登陆。

### 4.2) 更多细分过滤器

- 身份验证过滤器（Authentication Filters）。
- 数据压缩过滤器（Data compression Filters）。
- 加密过滤器（Encryption Filters）。
- 触发资源访问事件过滤器。
- 图像转换过滤器（Image Conversion Filters）。
- 日志记录和审核过滤器（Logging and Auditing Filters）。
- MIME-TYPE 链过滤器（MIME-TYPE Chain Filters）。
- 标记化过滤器（Tokenizing Filters）。
- XSL/T 过滤器（XSL/T Filters），转换 XML 内容。

### 4.3) 过滤器的使用

- 为什么使用过滤器：

类似性能评测、用户验证、字符替换、编码设置这类需求，
与应用程序的业务需求没有直接关系，
应该设计为独立的元件，可以随意插拔，
或者随意修改设置而无需修改业务代码。

- 过滤器概念：

过滤器是一个实现了 javax.servlet.Filter接口的Java类，
作用于浏览器和Servlet中间，过滤请求与响应。

- 过滤器的定义：

- Servlet/JSP要实现过滤器，必须实现Filter接口，并在web.xml中定义过滤器，让过滤器知道加载哪个过滤器类。
- Filter接口有三个要实现的方法， init()、 doFilter() 与 destroy()。

- 过滤器的执行过程：

- 当过滤器类被载入容器并实例化后，容器会运行 init()方法并传入FilterConfig对象作为参数。
- 当请求来到过滤器时，会调用 doFilter()方法， doFilter()上除了ServletRequest和ServletResponse之外，还有一个FilterChain参数；
- 如果调用了FilterChain的 doFilter()方法，就会运行下一个过滤器，如果没有下一个过滤器，就调用请求目标Servlet的 service()方法；
- 在到达 service()方法之后，流程会以堆栈顺序返回，所以在FilterChain的 doFilter()运行完毕后，就可以针对service()方法做后续处理；
- 如果因为某个条件（如用户没有通过验证）而不调用FilterChain的 doFilter()，则请求就不会继续至目标Servlet，此时过滤器就起到了拦截请求的作用；

- Filter接口：

```
public interface Filter {
    //用于完成Filter的初始化
    //该方法由 Web 容器调用，指示一个过滤器被放入服务
    public void init(FilterConfig filterConfig) throws ServletException;

    //实现过滤功能
    //该方法在每次一个请求/响应对因客户端在链的末端请求资源而通过链传递时由容器调用
    public void doFilter ( ServletRequest request, ServletResponse response,
            FilterChain chain ) throws IOException, ServletException;

    //用于销毁Filter前，完成某些资源的回收
    //该方法由 Web 容器调用，指示一个过滤器被取出服务。
    public void destroy();
}
```

### 4.4) 过滤器的生命周期

web.xml 中声明的每个 filter 在每个虚拟机中仅有一个实例。

- (1) 加载和实例化

```
Web容器启动时，即会根据 web.xml中声明的 filter顺序依次实例化这些 filter。
```

- (2) 初始化

```
Web容器调用init(FilterConfig)来初始化过滤器。
容器在调用该方法时，向过滤器传递 FilterConfig对象，FilterConfig的用法和 ServletConfig类似。
利用 FilterConfig对象可以得到 ServletContext对象，以及在 web.xml 中配置的过滤器的初始化参数。
在这个方法中，可以抛出 ServletException 异常，通知容器该过滤器不能正常工作。
此时的 Web 容器启动失败，整个应用程序不能够被访问。
实例化和初始化的操作只会在容器启动时执行，而且只会执行一次。
```

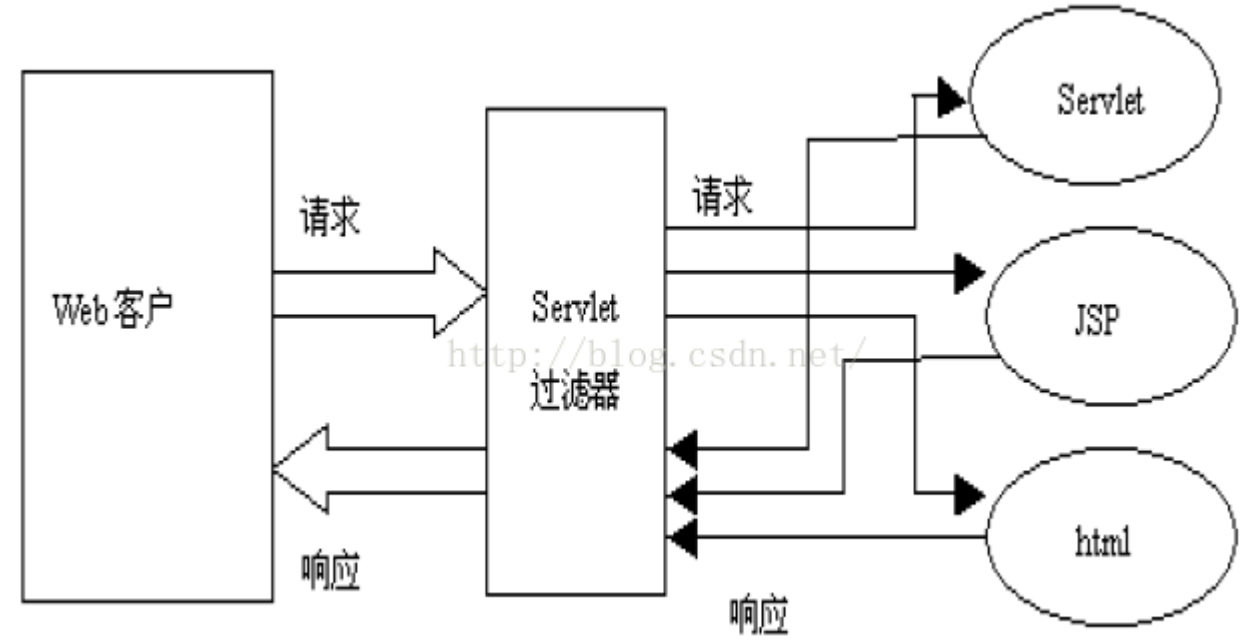
- (3) doFilter

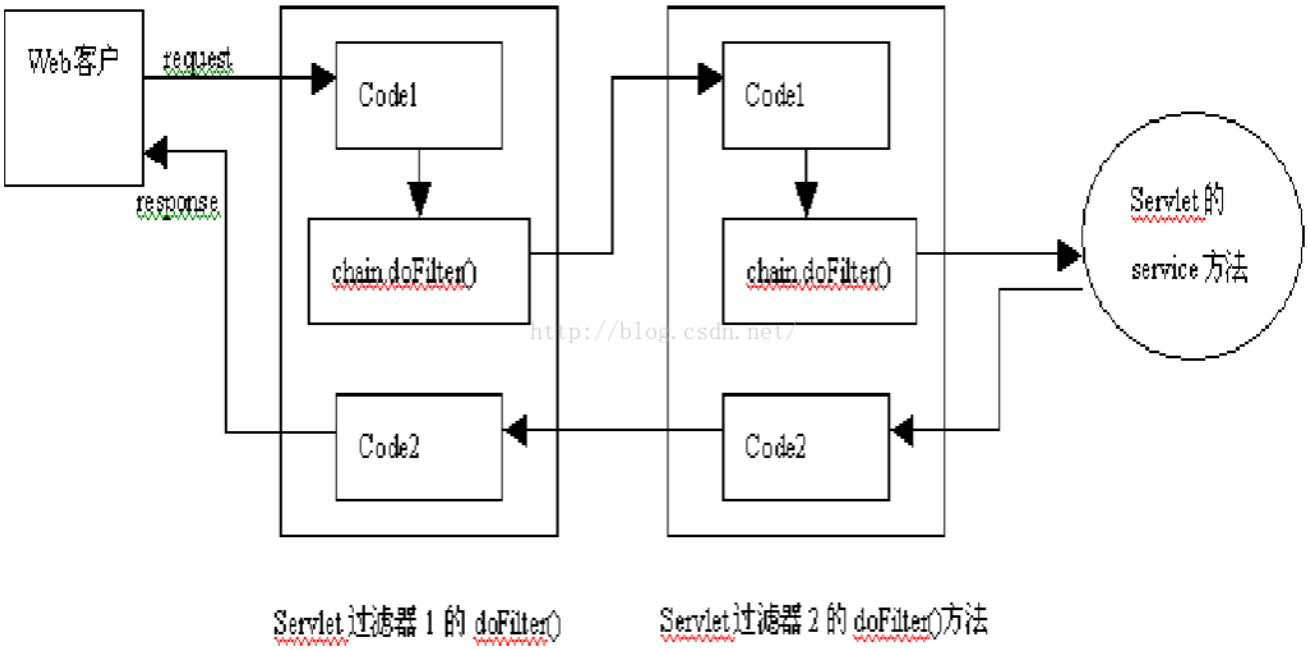
```
doFilter方法类似于Servlet接口的 service()方法。
当客户端请求目标资源的时候，容器会筛选出符合 filter-mapping中url-pattern的filter，
并按照声明filter-mapping的顺序依次调用这些 filter的doFilter方法。
在这个链式调用过程中，
可以调用 chain.doFilter(ServletRequest, ServletResponse)将请求传给下一个过滤器(或目标资源)，
也可以直接向客户端返回响应信息，
或者利用 RequestDispatcher的forward和include方法，
以及 HttpServletResponse的sendRedirect方法将请求转向到其它资源。
需要注意的是，这个方法的请求和响应参数的类型是 ServletRequest和ServletResponse，
也就是说，过滤器的使用并不依赖于具体的协议。
```

- (4)销毁

```
Web容器调用 destroy()方法指示过滤器的生命周期结束。
在这个方法中，可以释放过滤器使用的资源。
```

#### 4.5) 过滤器的运行原理




**参考来源：**

a) Servlet、Filter、Listener深入理解
b) Servlet 编写过滤器
c) 《JSP&Servlet学习笔记(第2版)》作者林信良

## ？ 5、JSP的动态include和静态include

**5.1) 概念：**

- 5.1.1) 静态include：

- <%@include file=”xxx.jsp”%> 是JSP指示（或者指令）元素的一种，用于告知容器将其他JSP页面包括进来进行转译。
- 使用include来包括其他页面内容时，会在转译时就决定转译后的Servlet内容，所以说是一种静态的包括方式。

- 5.1.2) 动态include：

- <jsp:include page="xxx.jsp" /> 是JSP行为（或者动作）元素的一种，使用动态include将页面包含进来，当前页面会生成一个Servlet类，被include的页面也会独立生成一个Servlet类。
- 当前页面转译而成的Servlet中，会取得RequestDispatcher对象，并执行 include()方法，也就是请求时将动态include的页面转交给另一个Servlet，而后再回到自己的Servlet

**5.2) 区别**

- 5.2.1) 执行时间不同:

- 静态include是在转译阶段执行
- 动态include在请求处理阶段执行.

- 5.2.2) 引入内容不同:

- 静态include引入静态文本(html,jsp)，在JSP页面被转化成servlet之前和它融和到一起
- 动态include引入执行页面或servlet所生成的应答文本.

- 5.2.3) 属性区别：

- 静态include通过file属性指定被包含的文件，并且file属性不支持任何表达式；
- 动态include通过page属性指定被包含的文件，且page属性支持JSP表达式；

- 5.2.4) 编译区别：

- 静态include时，被包含的文件内容会原封不动的插入到包含页中，然后JSP编译器再将合成后的文件最终编译成一个Java文件，因此被导入页面甚至不需要是一个完整的页面；
- 动态inlcude时，当该标识被执行时，程序会将请求转发（不是请求重定向）到被包含的页面，并将执行结果输出到浏览器中，然后返回包含页继续执行后面的代码。因为服务器执行的是多个文件，所以JSP编译器会分别对这些文件进行编译；

- 5.2.5) 编译指令区别：

- 静态inlcude时被导入页面的编译指令会起作用；
- 而动态include时被导入页面的编译指令则失去作用，只是插入被导入页面的body内容。

- 5.2.6) 变量作用域区别：

- 静态include时，由于被包含的文件最终会生成一个文件，所以在被包含、包含文件中不能有重名的变量或方法；
- 动态include时，由于每个文件是单独编译的，所以在被包含文件和包含文件中重名的变量和方法是不相冲突的。

- 5.2.7) 传参区别：

```
- 静态include会导致两个jsp合并成为同一个java文件，所以不存在传参问题
- 动态include对被包含的jsp文件进行了一次独立的访问，通过在<jsp:include>标签中嵌入子标签<jsp:param>的形式传递参数
<jsp:include page="xxx.jsp">
    <jsp:param name="参数名" value="参数值" />
</jsp:include>
在被包含的JSP页面中通request.getParameter("参数名")获取参数值
<p>获取的参数值在此显示：<%=request.getParameter("参数名")%></p>
```

**参考来源：**

a) JSP include
b) JSP之静态include指令、动态Include指令
c) JSP中动态INCLUDE与静态INCLUDE的区别
d) 《JSP&Servlet学习笔记(第2版)》作者林信良

## %6、web.xml中常用配置及作用

### 6.1) 常用web.xml配置场景

- 6.1.1) 指定欢迎页面：

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index1.jsp</welcome-file>
</welcome-file-list>

<!-- 上面的例子指定了2个欢迎页面，显示时按顺序从第一个找起 -->
<!-- 如果第一个存在，就显示第一个，后面的不起作用 -->
<!-- 如果第一个不存在，就找第二个，以此类推 -->
```

- 6.1.2) 命名与定制URL

```
<!-- 我们可以为Servlet和JSP文件命名并定制URL-->
<!-- 其中定制URL是依赖一命名的，命名必须在定制URL前-->

<!-- (1) 为Servlet命名 -->
<servlet>
  <servlet-name>servlet1</servlet-name>
  <servlet-class>net.test.TestServlet</servlet-class>
</servlet>

<!-- (2) 为Servlet定制URL -->
<servlet-mapping>
  <servlet-name>servlet1</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

- 6.1.3) 定制初始化参数：

```
<!-- 可以定制servlet、JSP、Context的初始化参数 -->
<!-- 然后可以再servlet、JSP、Context中获取这些参数值 -->
<servlet>
  <servlet-name>servlet1</servlet-name>
  <servlet-class>net.test.TestServlet</servlet-class>
  <init-param>
    <param-name>userName</param-name>
    <param-value>Tommy</param-value>
  </init-param>
  <init-param>
    <param-name>E-mail</param-name>
    <param-value>Tommy@163.com</param-value>
  </init-param>
</servlet>
<!-- 经过上面的配置，在servlet中能够调用getServletConfig().getInitParameter("param1")获得参数名对应的值-->
```

- 6.1.4) 指定错误处理页面

```
<!-- 可以通过“异常类型”或“错误码”来指定错误处理页面 -->
<error-page>
  <error-code>404</error-code>
  <location>/error404.jsp</location>
</error-page>

<error-page>
  <exception-type>java.lang.Exception<exception-type>
  <location>/exception.jsp<location>
</error-page>
```

- 6.1.5) 设置过滤器：

```
<!--  比如设置一个编码过滤器，过滤所有资源 -->
<filter>
  <filter-name>XXXCharaSetFilter</filter-name>
  <filter-class>net.test.CharSetFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>XXXCharaSetFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 6.1.6) 设置监听器：

```
<listener>
  <listener-class>net.test.XXXLisenet</listener-class>
</listener>
```

- 6.1.7) 设置会话(Session)过期时间

```
<!-- 时间以分钟为单位，假如设置60分钟超时 -->
<session-config>
  <session-timeout>60</session-timeout>
</session-config>
```

### 6.2) web.xml标签及说明

```
<web-app>

    <!--定义了WEB应用的名字 -->
    <display-name></display-name>

    <!--声明WEB应用的描述信息 -->
    <description></description>

    <!--context-param元素声明应用范围内的初始化参数 -->
    <context-param></context-param>

    <!--过滤器元素将一个名字与一个实现javax.servlet.Filter接口的类相关联 -->
    <filter></filter>

    <!--一旦命名了一个过滤器，就要利用filter-mapping元素把它与一个或多个servlet或JSP页面相关联 -->
    <filter-mapping></filter-mapping>

    <!--servlet API的版本2.3增加了对事件监听程序的支持，事件监听程序在建立、修改和删除会话或servlet环境时得到通知。 Listener元素指出事件监听程序类 -->
    <listener></listener>

    <!--在向servlet或JSP页面制定初始化参数或定制URL时，必须首先命名servlet或JSP页面。 Servlet元素就是用来完成此项任务的 -->
    <servlet></servlet>

    <!--服务器一般为servlet提供一个缺省的URL：http://host/webAppPrefix/servlet/ServletName。
        但是，常常会更改这个URL，以便servlet可以访问初始化参数或更容易地处理相对URL。 在更改缺省URL时，使用servlet-mapping元素 -->
    <servlet-mapping></servlet-mapping>

    <!--如果某个会话在一定时间内未被访问，服务器可以抛弃它以节省内存。可通过使用HttpSession的setMaxInactiveInterval方法明确设置单个会话对象的超时值，或者可利用session-config元素制定缺省超时值 -->
    <session-config></session-config>

    <!--如果Web应用具有想到特殊的文件，希望能保证给他们分配特定的MIME类型，则mime-mapping元素提供这种保证 -->
    <mime-mapping></mime-mapping>

    <!--指示服务器在收到引用一个目录名而不是文件名的URL时，使用哪个文件 -->
    <welcome-file-list></welcome-file-list>

    <!--在返回特定HTTP状态代码时，或者特定类型的异常被抛出时，能够制定将要显示的页面 -->
```

```
    <error-page></error-page>

    <!--对标记库描述符文件（Tag Libraryu Descriptor file）指定别名。此功能使你能够更改TLD文件的位置，而不用编辑使用这些文件的JSP页面 -->
    <taglib></taglib>

    <!--声明与资源相关的一个管理对象 -->
    <resource-env-ref></resource-env-ref>

    <!--声明一个资源工厂使用的外部资源 -->
    <resource-ref></resource-ref>

    <!--制定应该保护的URL。它与login-config元素联合使用 -->
    <security-constraint></security-constraint>

    <!--指定服务器应该怎样给试图访问受保护页面的用户授权。它与sercurity-constraint元素联合使用 -->
    <login-config></login-config>

    <!--给出安全角色的一个列表，这些角色将出现在servlet元素内的security-role-ref元素的role-name子元素中。 分别地声明角色可使高级IDE处理安全信息更为容易 -->
    <security-role></security-role>

    <!--声明Web应用的环境项 -->
    <env-entry></env-entry>

    <!--声明一个EJB的主目录的引用 -->
    <ejb-ref></ejb-ref>

    <!--声明一个EJB的本地主目录的应用 -->
    <ejb-local-ref></ejb-local-ref>

</web-app>
```

### 6.3) 更多web.xml配置

- 6.3.1) Web应用图标：

```
<!-- 指出IDE和GUI工具用来表示Web应用的大图标和小图标 -->
<icon>
  <small-icon>/images/app_small.gif</small-icon>
  <large-icon>/images/app_large.gif</large-icon>
</icon>
```

- 6.3.2) Web 应用名称：

```
<!-- 提供GUI工具可能会用来标记这个特定Web应用的一个名称    -->
<display-name>Tomcat Example</display-name>
```

- 6.3.3) Web 应用描述：

```
<!--给出与此相关的说明性文本 -->
<disciption>Tomcat Example servlets and JSP pages.</disciption>
```

- 6.3.4) 上下文参数：

```
<!--声明应用范围内的初始化参数 -->
<context-param>
  <param-name>ContextParameter</para-name>
  <param-value>test</param-value>
  <description>It is a test parameter.</description>
</context-param>

<!-- 在servlet里面可以通过getServletContext().getInitParameter("context/param")得到 -->
```

- 6.3.5) 过滤器配置：

```
<!--将一个名字与一个实现javaxs.servlet.Filter接口的类相关联 -->
<filter>
  <filter-name>setCharacterEncoding</filter-name>
  <filter-class>com.myTest.setCharacterEncodingFilter</filter-class>
  <init-param>
  <param-name>encoding</param-name>
  <param-value>GB2312</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>setCharacterEncoding</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- 6.3.6) 监听器配置

```
<listener>
  <listerner-class>listener.SessionListener</listener-class>
</listener>
```

- 6.3.7) Servlet配置

```
<!--基本配置 -->
<servlet>
  <servlet-name>snoop</servlet-name>
  <servlet-class>SnoopServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>snoop</servlet-name>

  <url-pattern>/snoop</url-pattern>
```

```
</servlet-mapping>

<!-- 高级配置-->
<servlet>
  <servlet-name>snoop</servlet-name>
  <servlet-class>SnoopServlet</servlet-class>
  <init-param>
  <param-name>foo</param-name>
  <param-value>bar</param-value>
  </init-param>
  <run-as>
  <description>Security role for anonymous access</description>
  <role-name>tomcat</role-name>
  </run-as>
</servlet>

<servlet-mapping>
  <servlet-name>snoop</servlet-name>
  <url-pattern>/snoop</url-pattern>
</servlet-mapping>

<!--元素说明 -->
<servlet></servlet> <!--用来声明一个servlet的数据，主要有以下子元素-->
<servlet-name></servlet-name> <!--指定servlet的名称 -->
<servlet-class></servlet-class> <!--指定servlet的类名称 -->
<jsp-file></jsp-file> <!--指定web站台中的某个JSP网页的完整路径-->
<init-param></init-param> <!--用来定义参数，可有多个init-param。在servlet类中通过
getInitParamenter(String name)方法访问初始化参数 -->
<load-on-startup></load-on-startup> <!--指定当Web应用启动时，装载Servlet的次序。 当值为正数或零时：
Servlet容器先加载数值小的servlet，再依次加载其他数值大的servlet. 当值为负或未定义：Servlet容器将在Web
客户首次访问这个servlet时加载它 -->
<servlet-mapping></servlet-mapping> <!--用来定义servlet所对应的URL，包含两个子元素 -->
<servlet-name></servlet-name> <!--指定servlet的名称 -->
<url-pattern></url-pattern> <!--指定servlet所对应的URL -->
```

- 6.3.8) 会话超时配置（单位为分钟）

```
<session-config>
  <session-timeout>120</session-timeout>
</session-config>
```

- 6.3.9) MIME类型配置

```
<mime-mapping>
  <extension>htm</extension>
  <mime-type>text/html</mime-type>
</mime-mapping>
```

- 6.3.10) 指定欢迎文件页配置

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
  <welcome-file>index.htm</welcome-file>
</welcome-file-list>
```

- 6.3.11) 配置错误页面

```
<!-- 1. 通过错误码来配置error-page -->
<error-page>
  <error-code>404</error-code>
  <location>/NotFound.jsp</location>
</error-page>

<!-- 上面配置了当系统发生404错误时，跳转到错误处理页面NotFound.jsp -->

<!-- 2. 通过异常的类型配置error-page -->
<error-page>
  <exception-type>java.lang.NullException</exception-type>
  <location>/error.jsp</location>
</error-page>

<!--上面配置了当系统发生java.lang.NullException（即空指针异常）时，跳转到错误处理页面error.jsp -->
```

- 6.3.12) TLD配置

```
<taglib>
  <taglib-uri>http://jakarta.apache.org/tomcat/debug-taglib</taglib-uri>
  <taglib-location>/WEB-INF/jsp/debug-taglib.tld</taglib-location>
</taglib>

<!-- 如果IDE一直在报错,应该把<taglib> 放到 <jsp-config>中-->
<jsp-config>
  <taglib>
    <taglib-uri>http://jakarta.apache.org/tomcat/debug-taglib</taglib-uri>
    <taglib-location>/WEB-INF/pager-taglib.tld</taglib-location>
  </taglib>
</jsp-config>
```

- 6.3.13) 资源管理对象配置

```
<resource-env-ref>
  <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
</resource-env-ref>
```

- 6.3.14) 资源工厂配置

```
<resource-ref>
  <res-ref-name>mail/Session</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

<!--配置数据库连接池就可在此配置 -->
<resource-ref>
  <description>JNDI JDBC DataSource of shop</description>
  <res-ref-name>jdbc/sample_db</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

- 6.3.15) 安全限制配置

```
<security-constraint>
  <display-name>Example Security Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/jsp/security/protected/*</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>tomcat</role-name>
    <role-name>role1</role-name>
  </auth-constraint>
</security-constraint>
```

- 6.3.16) 登陆验证配置

```
<login-config>
  <auth-method>FORM</auth-method>
  <realm-name>Example-Based Authentiation Area</realm-name>
  <form-login-config>
    <form-login-page>/jsp/security/protected/login.jsp</form-login-page>
    <form-error-page>/jsp/security/protected/error.jsp</form-error-page>
  </form-login-config>
</login-config>
```

- 6.3.17) 安全角色

```
<!-- security-role元素给出安全角色的一个列表 -->
<!-- 这些角色将出现在servlet元素内security-role-ref元素的role-name子元素中-->
<!-- 分别地声明角色可使高级IDE处理安全信息更为容易 -->
<security-role>
  <role-name>tomcat</role-name>
</security-role>
```

- 6.3.18) Web环境参数

```
<!-- env-entry元素声明Web应用的环境项 -->
<env-entry>
  <env-entry-name>minExemptions</env-entry-name>
  <env-entry-value>1</env-entry-value>
  <env-entry-type>java.lang.Integer</env-entry-type>
</env-entry>
```

- 6.3.19) EJB 声明

```
<ejb-ref>
  <description>Example EJB reference</decription>
  <ejb-ref-name>ejb/Account</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <home>com.mycompany.mypackage.AccountHome</home>
  <remote>com.mycompany.mypackage.Account</remote>
</ejb-ref>
```

- 6.3.20) 本地EJB声明

```
<ejb-local-ref>
  <description>Example Loacal EJB reference</decription>
  <ejb-ref-name>ejb/ProcessOrder</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.mycompany.mypackage.ProcessOrderHome</local-home>
  <local>com.mycompany.mypackage.ProcessOrder</local>
</ejb-local-ref>
```

- 6.3.21) 配置DWR

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

- 6.3.22) 配置Struts

```
<display-name>Struts Blank Application</display-name>
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <init-param>
    <param-name>detail</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>debug</param-name>
    <param-value>2</param-value>
  </init-param>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>application</param-name>
    <param-value>ApplicationResources</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

<!-- Struts Tag Library Descriptors -->
<taglib>
  <taglib-uri>struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-bean.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts-html</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-html.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts-nested</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-nested.tld</taglib-location>
</taglib>

<taglib>
```

```
  <taglib-uri>struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-logic.tld</taglib-location>
</taglib>

<taglib>
  <taglib-uri>struts-tiles</taglib-uri>
  <taglib-location>/WEB-INF/tld/struts-tiles.tld</taglib-location>
</taglib>
```

- 6.3.23) 配置spring

```
<!-- 指定spring配置文件位置 -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <!--加载多个spring配置文件 -->
  <param-value>
    /WEB-INF/applicationContext.xml, /WEB-INF/action-servlet.xml
  </param-value>
</context-param>

<!-- 定义SPRING监听器，加载spring -->
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<listener>
  <listener-class>
    org.springframework.web.context.request.RequestContextListener
  </listener-class>
</listener>
```

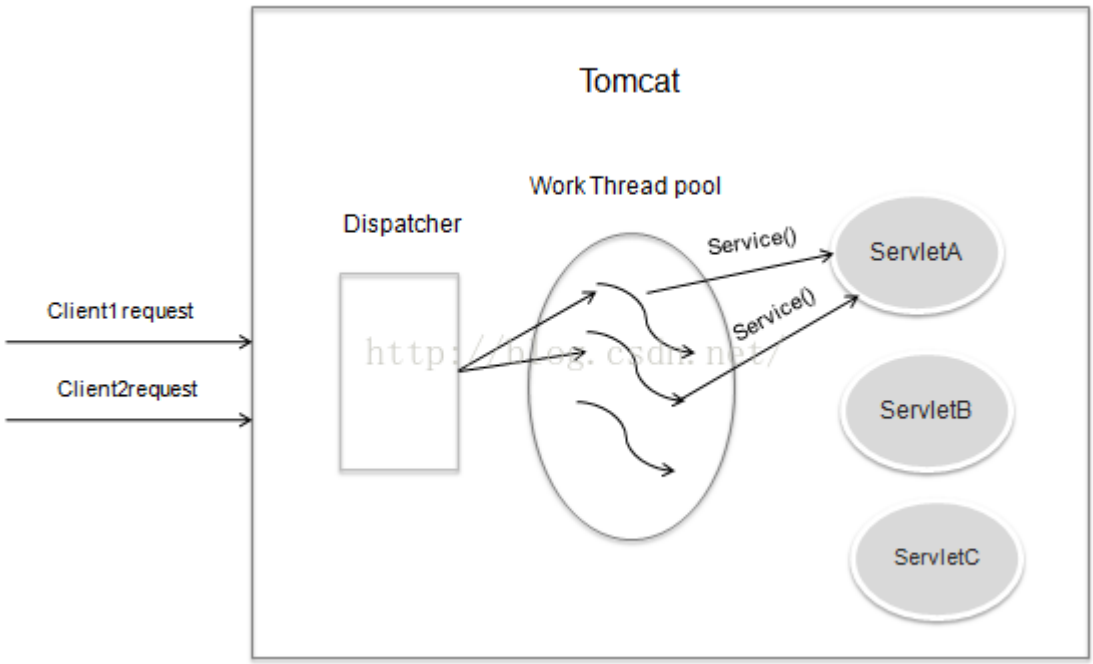
**参考来源:**

a) Java Web的web.xml文件作用及基本配置
b) 史上最全web.xml配置文件元素详解

---

## %7、Servlet的线程安全问题

### 7.1) 单实例多线程的Servlet模型




```
Servlet规范中定义，默认情况下（Servlet不是在分布式的环境中部署），
Servlet容器对声明的每一个Servlet，只创建一个实例。
如果有多个客户端请求同时访问这个Servlet，Servlet容器如何处理多个请求呢?
答案是采用多线程，Servlet容器维护一个线程池来服务请求。
当容器接收到一个访问Servlet的请求，调度者线程从线程池中选取一个工作线程，
将请求传递给该线程，然后由这个线程执行Servlet的 service()方法
```

### 7.2) 线程安全的Servlet

- 7.2.1) 变量的线程安全

```
Servlet是单实例多线程模型，多个线程共享一个Servlet实例，因此对于实例变量的访问是非线程安全的。
建议：在Servlet中尽可能的使用局部变量，应该只使用只读的实例变量和静态变量。
如果非得使用共享的实例变量或静态变量，在修改共享变量时应该注意线程同步。
```

- 7.2.2) 属性的线程安全

```
在Servlet中，可以访问保存在ServletContext、HttpSession和ServletRequest对象中的属性。
那么这三种不同范围的对象，属性访问是否是线程安全的呢?
- ServletContext：
  该对象被Web应用程序的所有Servlet共享，多线程环境下肯定是非线程安全的。

- HttpSession：
  HttpSession对象只能在同属于一个Session的请求线程中共享。
  对于同一个Session，我们可能会认为在同一时刻只有一个用户请求。
  因此，Session对象的属性访问是线程安全的。
  但是，如果用户打开多个同属于一个进程的浏览器窗口，
  在这些窗口中的访问请求同属于一个Session，对于多个线程的并发修改显然不是线程安全的。
```

```
- ServletRequest：
  因为Servlet容器对它所接收到的每一个请求，
  都创建一个新的ServletRequest对象，
  所以ServletRequest对象只在一个线程中被访问，
  因此对ServletRequest的属性访问是线程安全的。
  但是，如果在Servlet中创建了自己的线程，
  那么对ServletRequest的属性访问的线程安全性就得自己去保证。
  此外，如果将当前请求的Servlet通过HttpSession或者ServletContext共享，
  当然也是非线程安全的。
```

**参考来源:**

a) Servlet、Filter、Listener深入理解

> Contributes: bigablecat
>
> Reviewers : Hollis, Kevin Lee

# MVC框架

## ！1、介绍几个常用的MVC框架

### 1.1) Struts MVC

- Controller：FilterDispatcher
- View：JSP，volecity等
- Model：Action

```
系统核心控制器FilterDispatcher根据请求自动调用Action，
在Action中调用相应的业务逻辑组件完成处理后通过FilterDispatcher返回视图
```

### 1.2) Spring MVC

- Controller：DispatcherServlet
- View：JSP，volecity等
- Model：Controller

```
Spring MVC中，controller接收参数request和response，返回ModelAndView
```

**<参考来源>:**

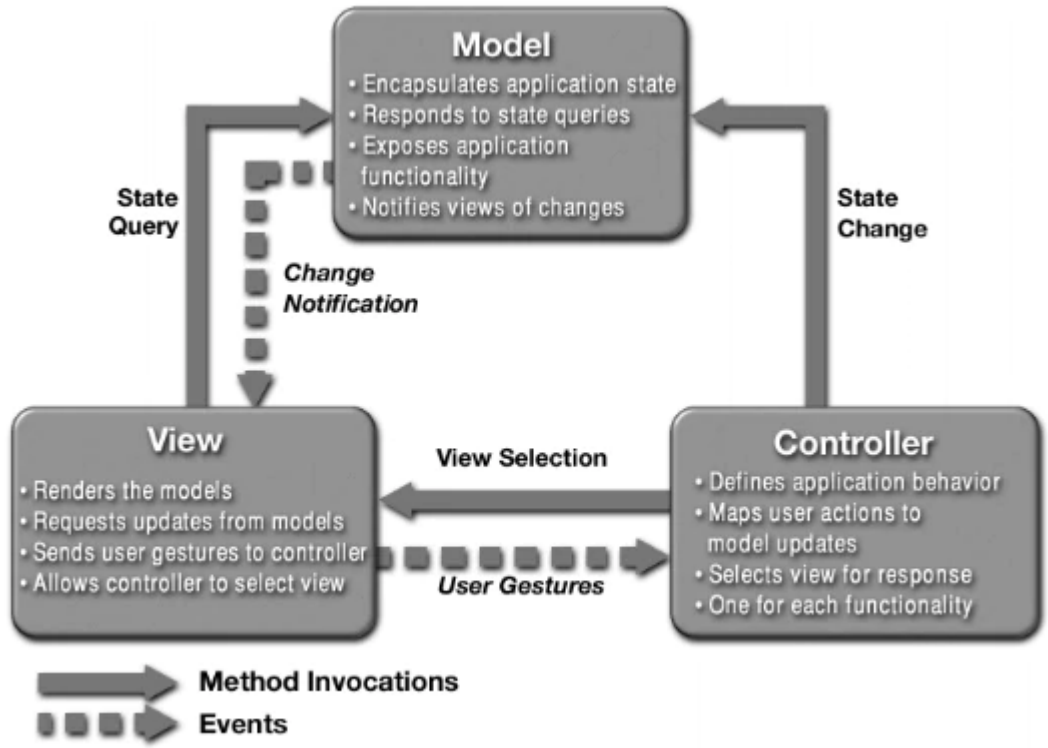
a) 《Java Web整合开发实战——基于Struts 2+Hibernate+Spring》（作者：贾蓓 镇明敏 杜磊 等）

## ！2、什么是MVC

### 2.1) MVC定义

```
MVC模式（Model-view-controller）是软件工程中的一种软件架构模式，
把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller），
MVC遵循职责分离原则，通常由Controller处理消息，Model掌管数据源，View负责数据显示
```

### 2.2) MVC一般模型




- Model

> 封装应用状态，对状态查询作出响应，暴露应用功能，通知视图改变

- View

> 渲染模型，向模型请求更新，发送用户动作至Controller，允许Controller选择视图

- Controller

> 定义应用行为，将用户动作映射到模型，选择视图用于响应，与模型中的功能一一对应

### 2.3) MVC具体实践

- Model

> Model负责数据访问，
> 较现代的 Framework 都会建议使用独立的数据对象 (DTO, POCO, POJO 等)来替代弱类型的集合对象。
> 数据访问的代码会使用 Data Access的代码或是 ORM-based Framework，
> 也可以进一步使用 Repository Pattern与 Unit of Works Pattern来切割数据源的相依性。

- View

```
View负责显示数据，这个部分多为前端应用，
而Controller会有一个机制将处理的结果（可能是 Model，集合或是状态等)交给 View，
然后由 View 来决定怎么显示。
例如 Spring Framework使用JSP或相应技术，
ASP.NET MVC则使用Razor处理数据的显示。
```

- Controller

```
Controller负责处理消息，较高阶的 Framework会有一个默认的实现来作为 Controller的基础，
例如Spring的 DispatcherServlet或是 ASP.NET MVC的Controller等，
在职责分离原则的基础上，每个 Controller负责的部分不同，
因此会将各个 Controller切割成不同的文件以利维护。
```

**<参考来源>：**

a) Java BluePrints Model-View-Controller
b) 维基百科:MVC

## ！3、Struts中请求的实现过程

### 3.1) 从Struts2框架的角度描述工作流程

- 1. 发出请求：

```
用户发出一个HttpServletRequest请求
```

- 1. 过滤请求：
  经过一系列过滤器过滤

1. 调用FilterDispatcher：
   FilterDispatcher是控制器的核心，
   通过询问ActionMapper来确定请求是否需要调用某个Action，
   如果需要就将请求转交给ActionProxy
2. ActionProxy找到Action类：
   通过配置管理器Configuration Manager询问框架配置文件struts.xml，
   找到要调用的Action类
3. ActionInvocation加载拦截器：
   ActionProxy创建一个ActionInvocation实例，
   该实例使用命名模式来调用，
   在Action执行的前后，
   ActionInvocation实例根据配置文件加载与Action相关的所有拦截器Interceptor
4. 返回结果：
   一旦Action执行完毕，
   ActionInvocation实例根据strtus.xml文件中的配置找到相对应的返回结果，
   返回结果通常是一个JSP或FreeMarker模板
5. 返回响应：
   最后，HttpServletResponse通过web.xml中配置的过滤器返回

```
* 3.2) MVC角度描述Struts2工作流程:
```bash
1. 浏览器发出请求
2. 控制层中的核心控制器FilterDispatcher根据请求调用相应的Action
3. Struts2的拦截器链自动对请求调用一些通用的控制逻辑，如数据校验，数据封装和文件上传等
4. 回调Action中的excute()方法并在方法体内调用业务逻辑组件，即自定义的Javabean来处理请求，如数据查询等
5. execute()方法返回后会产生一个输出
6. 该输出经过浏览器拦截链自动处理，和开始的拦截器链处理顺序相反
7. 控制层最后将数据返还并更新视图层
```

**<参考来源>:**

a) 《Java Web整合开发实战——基于Struts 2+Hibernate+Spring》（作者：贾蓓 镇明敏 杜磊 等）

### %4、Spring mvc与Struts mvc的区别

| Name | Academy | score |
| --- | --- | --- |
| Harry Potter | Gryffindor | 90 |
| Hermione Granger | Gryffindor | 100 |
| Draco Malfoy | Slytherin | 90 |

### ? 5、Service嵌套事务处理，如何回滚

### ! 6、struts2 中拦截器与过滤器的区别及执行顺序

### %7、struts2拦截器的实现原理

### 参考资料

> Contributes: xxx
>
> Reviewers : Hollis, Kevin Lee

## http相关

### ! 1、`session`和`cookie`的区别

### ! 2、HTTP请求中`session`实现原理?

### %3、如果客户端禁止Cookie能实现Session吗?

### ! 4、http中`get`和`post`区别

### ! 5、`redirect`与`forward`的区别

### ! 6、常见的web请求返回的状态码。404、302、301、500分别代表什么

## SSH相关
```

### ？1、`Hibernate`/`Ibatis`/`MyBatis`之间的区别

### ？2、什么是`OR Mapping`

### %3、hibernate的缓存机制、一级和二级缓存

### ！4、使用Spring的好处是什么，Spring的核心理念

### ！5、什么是`AOP`和`IOC`，实现原理是什么

### ！6、spring bean的初始化过程

### ！7、Spring的`事务管理`，`Spring bean注入`的几种方式

### %8、spring四种依赖注入方式

## 容器相关

### ！1、什么是web服务器、什么是应用服务器

### ！2、常用的web服务器有哪些?

### ？3、`Tomcat`和`weblogic`的区别

## web安全

### ！1、什么是SQL注入，如何避免。

> 定义

- SQL注入 就是通过把SQL命令插入到Web表单递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

> 解决方法

- 避免数据变成代码被执行，时刻分清代码和数据的界限。具体到SQL注入来说，被执行的恶意代码是通过数据库的SQL解释引擎编译得到的，所以只要避免用户输入的数据被数据库系统编译就可以了
- 用户输入的数据进行转义从而来防止SQL注入

### %2、什么是XSS攻击，如何避免

> 定义

XSS又称CSS，全称Cross SiteScript，跨站脚本攻击，是Web程序中常见的漏洞，XSS属于被动式且用于客户端的攻击方式，所以容易被忽略其危害性。

其原理是攻击者向有XSS漏洞的网站中输入(传入)恶意的HTML代码，当其它用户浏览该网站时，这段HTML代码会自动执行，从而达到攻击的目的。

如，盗取用户Cookie、破坏页面结构、重定向到其它网站等

xss攻击可以分成两种类型：

- 非持久型攻击
- 持久型攻击

**1. 非持久型xss攻击**

顾名思义，非持久型xss攻击是一次性的，仅对当次的页面访问产生影响。非持久型xss攻击要求用户访问一个被攻击者篡改后的链接，用户访问该链接时，被植入的攻击脚本被用户游览器执行，从而达到攻击目的。

**2.持久型xss攻击**

持久型xss攻击会把攻击者的数据存储在服务器端，攻击行为将伴随着攻击数据一直存在。

> 解决方法

- 完善的过滤体系

  永远不相信用户的输入。需要对用户的输入进行处理，只允许输入合法的值，其它值一概过滤掉。

- Html encode

  假如某些情况下，我们不能对用户数据进行严格的过滤，那我们也需要对标签进行转换。

## %3、什么是CSRF攻击，如何避免

> 定义

CSRF（Cross-site request forgery），中文名称：跨站请求伪造，也被称为：one click attack/session riding，缩写为：CSRF/XSRF。

CSRF是一种诱骗受害者提交恶意请求的攻击。它继承了受害者的身份和特权，代表受害者执行不需要的功能。对于大多数网站，浏览器请求会自动包含与该网站相关的任何凭据，例如用户的会话Cookie，IP地址，Windows域凭据等。因此，如果用户当前对站点进行身份验证，站点将无法区分受害者发送的伪造请求和受害者发送的合法请求

CSRF攻击的目标是导致服务器状态发生变化的功能，例如更改受害者的电子邮件地址或密码或购买某些内容。强制受害者检索数据不会使攻击者受益，因为攻击者没有收到响应，受害者会这样做。因此，CSRF攻击针对状态改变请求。

> 解决方法

服务端的CSRF方式方法很多样，但总的思想都是一致的，就是在客户端页面**增加伪随机数。**

验证 HTTP Referer 字段；

在请求地址中添加 token 并验证；

在 HTTP 头中自定义属性并验证

## 参考资料


1. xss攻击
2. CSRF 攻击的应对之道
3. CSRF
4. 浅谈CSRF攻击方式


> Contributes: hueizhe
>
> Reviewers : Hollis, Kevin Lee

# 动态代理

## ！ 1、Java的动态代理的概念

%2、Java的动态代理的实现

### 编码问题

！1、常用的字符编码

！2、如何解决中文乱码问题

### 其他

%1、XML的解析方式，以及优缺点。

%2、什么是ajax，`Ajax`如何解决跨域问题

## 设计模式

%1、谈一下自己了解或者熟悉的`设计模式`

！2、`Singleton`的几种实现方式，实现一个线程安全的单例。

？3、`工厂模式`和`抽象工厂模式`之间的区别

## 知识综合能力

！1、请介绍一下一个http请求的全过程，描述的越全面越好

！2、当你在浏览器地址栏输入www.taobao.com，敲下回车之后都发生了什么

## 工具使用

！1、知道git/svn是干什么的吗？用过吗

！2、知道maven/gradle是干什么的吗？用过吗

！3、平常使用什么IDE，为什么

！4、平常使用什么浏览器，为什么

！5、平常开发机器是什么操作系统的

！6、会在Linux上开发吗。Linux常用命令会吗

## 项目相关

！1、请简单介绍一下你的这个项目

！2、你在这个项目中充当什么角色

！3、这个项目的技术选型有做过么。

！4、选择某项技术做过哪些调研和对比

**！5、这个项目中遇到的最大的问题是什么？你是如何解决的。**

**！6、项目中是否考虑过性能、安全性等问题**

## 技术热情

**！1、当前Java的最新版本**

**！2、Java8的lambda表达式**

**%3、Java8的stream API**

**%4、Java9的模块化**

**%5、Java10的局部变量类型推断**

**%6、Spring Boot2.0**

**%7、HTTP/2**

**%8、会翻墙么，知道翻墙的原理吗**

**！9、你最近在读什么书**

## 表达能力

**！1、能不能简单做一个自我介绍。**

**！2、能不能描述一下杭州给你的印象。用三句话概括一下。**

## 思考方式

**！1、如何估算杭州有多少软件工程师**

**！2、你最近读过的印象最深的文章是什么**

**！3、这篇文章中有几个观点，你最赞成哪一个，最不赞成哪一个**

## 其他

**！1、你对加班怎么看**

**！2、你还有什么问题要问我（面试官）的么**