10/9/17

# 一.关于前端模板

## 1.jsp的内置对象及其作用

 JSP中一共预先定义了9个这样的对象，分别为：request、response、session、application、out、pagecontext、config、page、exception

**1、request对象**

request 对象是 javax.servlet.httpServletRequest类型的对象。 该对象代表了客户端的请求信息，主要用于接受通过HTTP协议传送到服务器的数据。（包括头信息、系统信息、请求方式以及请求参数等）。request对象的作用域为一次请求。

**2、response对象**

response 代表的是对客户端的响应，主要是将JSP容器处理过的对象传回到客户端。response对象也具有作用域，它只在JSP页面内有效。

**3、session对象**

session 对象是由服务器自动创建的与用户请求相关的对象。服务器为每个用户都生成一个session对象，用于保存该用户的信息，跟踪用户的操作状态。session对象内部使用Map类来保存数据，因此保存数据的格式为 “Key/value”。 session对象的value可以使复杂的对象类型，而不仅仅局限于字符串类型。

**4、application对象**

 application 对象可将信息保存在服务器中，直到服务器关闭，否则application对象中保存的信息会在整个应用中都有效。与session对象相比，application对象生命周期更长，类似于系统的“全局变量”。

**5、out 对象**

out 对象用于在Web浏览器内输出信息，并且管理应用服务器上的输出缓冲区。在使用 out 对象输出数据时，可以对数据缓冲区进行操作，及时清除缓冲区中的残余数据，为其他的输出让出缓冲空间。待数据输出完毕后，要及时关闭输出流。

**6、pageContext 对象**

pageContext 对象的作用是取得任何范围的参数，通过它可以获取 JSP页面的out、request、reponse、session、application 等对象。pageContext对象的创建和初始化都是由容器来完成的，在JSP页面中可以直接使用 pageContext对象。

**7、config 对象**

config 对象的主要作用是取得服务器的配置信息。通过 pageConext对象的 getServletConfig() 方法可以获取一个config对象。当一个Servlet 初始化时，容器把某些信息通过 config对象传递给这个 Servlet。 开发者可以在web.xml 文件中为应用程序环境中的Servlet程序和JSP页面提供初始化参数。

**8、page 对象**

page 对象代表JSP本身，只有在JSP页面内才是合法的。 page隐含对象本质上包含当前 Servlet接口引用的变量，类似于Java编程中的 this 指针。

**9、exception 对象**

exception 对象的作用是显示异常信息，只有在包含 isErrorPage="true" 的页面中才可以被使用，在一般的JSP页面中使用该对象将无法编译JSP文件。excepation对象和Java的所有对象一样，都具有系统提供的继承结构。exception 对象几乎定义了所有异常情况。在Java程序中，可以使用try/catch关键字来处理异常情况； 如果在JSP页面中出现没有捕获到的异常，就会生成 exception 对象，并把 exception 对象传送到在page指令中设定的错误页面中，然后在错误页面中处理相应的 exception 对象。

## 2.freemaker

## 3.velocity

## 4.thymeleaf

# 二.数据结构及算法相关

## 1.线性表

线性表是最常用且最简单的一种数据结构，它是n个数据元素的有限序列。

实现线性表的方式一般有两种，一种是使用数组存储线性表的元素，即用一组连续的存储单元依次存储线性表的数据元素。另一种是使用链表存储线性表的元素，即用一组任意的存储单元存储线性表的数据元素（存储单元可以是连续的，也可以是不连续的）。

### 1.1数组实现

数组是一种大小固定的数据结构，对线性表的所有操作都可以通过数组来实现。虽然数组一旦创建之后，它的大小就无法改变了，但是当数组不能再存储线性表中的新元素时，我们可以创建一个新的大的数组来替换当前数组。这样就可以使用数组实现动态的数据结构。  
上面简单写出了数组实现线性表的两个典型函数，具体我们可以参考Java里面的ArrayList集合类的源码。数组实现的线性表优点在于可以通过下标来访问或者修改元素，比较高效，主要缺点在于插入和删除的花费开销较大，比如当在第一个位置前插入一个元素，那么首先要把所有的元素往后移动一个位置。为了提高在任意位置添加或者删除元素的效率，可以采用链式结构来实现线性表。

### 1.2链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列节点组成，这些节点不必在内存中相连。每个节点由数据部分Data和链部分Next，Next指向下一个节点，这样当添加或者删除时，只需要改变相关节点的Next的指向，效率很高。



单链表的结构

下面主要用代码来展示链表的一些基本操作，需要注意的是，这里主要是以单链表为例，暂时不考虑双链表和循环链表。

上面的几段代码主要展示了链表的几个基本操作，还有很多像获取指定元素，移除元素等操作大家可以自己完成，写这些代码的时候一定要理清节点之间关系，这样才不容易出错。

链表的实现还有其它的方式，常见的有循环单链表，双向链表，循环双向链表。 **循环单链表** 主要是链表的最后一个节点指向第一个节点，整体构成一个链环。 **双向链表** 主要是节点中包含两个指针部分，一个指向前驱元，一个指向后继元，JDK中LinkedList集合类的实现就是双向链表。 **循环双向链表** 是最后一个节点指向第一个节点。

## 2．栈与队列

栈和队列也是比较常见的数据结构，它们是比较特殊的线性表，因为对于栈来说，访问、插入和删除元素只能在栈顶进行，对于队列来说，元素只能从队列尾插入，从队列头访问和删除。

### 2.1栈

栈是限制插入和删除只能在一个位置上进行的表，该位置是表的末端，叫作栈顶，对栈的基本操作有push(进栈)和pop(出栈)，前者相当于插入，后者相当于删除最后一个元素。栈有时又叫作LIFO(Last In First Out)表，即后进先出。



栈的模型

下面我们看一道经典题目，加深对栈的理解。



关于栈的一道经典题目

上图中的答案是C，其中的原理可以好好想一想。

因为栈也是一个表，所以任何实现表的方法都能实现栈。我们打开JDK中的类Stack的源码，可以看到它就是继承类Vector的。当然，Stack是Java2前的容器类，现在我们可以使用LinkedList来进行栈的所有操作。

队列

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。



队列示意图

我们可以使用链表来实现队列，下面代码简单展示了利用LinkedList来实现队列类。

线性结构

- 顺序表

typedef struct {

    int elem[100];

    int length;  // 这里的lenth是指当前分配的长度

} SqList;

由以上结构可以看出， 结点的值存储在 elem 中，而结点之间的关系就是数组隐含, 所以不需要另外在定义关系.

- 单链表

typedef struct LNode{

    int elem;

    struct LNode \*next;

} LNode, \*LinkList;

结点: LNode 是用来保存结点的

关系: LinkList 就是链表头指针, 关系是通过 next 指针联系起来的.

头指针: LinkList 就是头指针, 指向头结点的指针.

头结点: (1)对带头结点的链表, 在表的任何结点之前插入结点或删除表中任何结点, 所要做的都是修改前一个结点的指针域, 而任何元素都有前驱结点, 若链表没有头结点, 则首元素结点没有前驱结点, 在其前插入结点或删除结点时操作会复杂些.(2)对带头结点的链表, 表头指针时指向结点的非空指针, 因此空表与非空表处理是一样的.

- 循环链表

所谓循环链表, 其实

结点: 存储情况, 同上边完全一样.

关系: 头指针的 next 指向自己, 这样的话就是循环链表了, 当插入结点时, 新的结点 p->next = L->next(指向头结点).

- 双向链表

typedef struct DLNode {

    int elem;

    struct DLNode \*prior;

    struct DLNode \*next;

} DLNode, \*DLinkList;

结点: DLNode

结构: 头指针 DLinkList, 通过 next 和 prior 来反映元素之间的线性关系.

- 静态链表

所谓静态链表: 是指用数组模拟操作, 实现的链表, 其中指针域, 使用数组下标表示.

typedef struct {

    int elem;

    int next;

} SLNode, slinklist[MAXSIZE];

结点: SLNode, 其中的 next 就是模拟指针.

关系: slinklist 是一个SLNode的数组, 数组中的 next 隐含关系.

栈和队列: 是限制操作的线性表

- 顺序栈

typedef struct {

    int elem[100];

    int top;

} SqStack;

结点: 数组中的元素;

关系: SqStack.

为什么没有链式栈, 因为栈这种结构限制了, 后进先出, 即只能从栈顶出战, 即 top 会记录栈顶位置, 所以它虽然是顺序结构, 但是插入和删除操作并不需要移动元素, 所以, 当然是顺序栈好一些.

- 顺序队列( 循环队列 )

typedef struct {

    int elem[100];

    int front;

    int rear;

} SqQueue;

结点: elem数组中的元素

关系: 隐含在数组中, 注意 front 和 rear 的位置, 关系还是隐含在数组中, 队列是先进先出, front 记录了队列头, rear 记录了队列尾, 从 front出, rear进, 注意队列判空和判满条件: 如下

因为 出队列时, 头指针 front 会向后移动, 此时, 前一个存储区域虽然出队列了, 但是仍然占据了存储空间没有释放, 这样就势必造成了空间的浪费, 这样最好的办法是使用循环队列, 但是循环队列如何判空和判满呢?

[](http://images.cnitblog.com/blog/370445/201303/19105844-53762154b90148ecaaa5c63ac0f16c8e.png)

如上图: 从结构上看, 队列里只剩下了 3 个存储单元, 前边浪费了大量存储空间, 所以要使用循环队列, 并且不能通过 front == rear 来简单的判断判空或判满, 浪费一个存储空间, 即 (rear + 1) % 存储空间 = front, 则判断为慢, 关键看谁最上谁, 如果 front追上rear 空队列, 如果是 rear追上了 front满队列. 为什么要 (rear + 1)%存储空间呢? 因为当 rear已经在数组最右边时, 如果单纯的 rear+1, 那么已经超过数组最大范围, 但是(rear+1)%存储空间, 如果 rear+1没有超过存储空间, 那么取模与不取模操作都一样, 但是如果 rear+1超过了数组范围,那么取模以后, 又回到了第一个了, 这样就达到了循环的目的, 而 (rear+1)%存储空间 == front 表示 rear 已经循环到了 front的前一个存储空间了.

- 链式队列

typedef struct Qnode {

    int elem;

    struct Qnode \*next;

} Qnode, \*Qlink;

typedef struct SQlink {

    Qlink front;

    Qlink rear;

} \*linkqueue;

结点: Qnode

关系: linkqueue

注意: 链式队列不需要循环队列, 因为不存在空间浪费的情况, 当有出队列的结点时, 直接释放该结点的内存就可以了.

数组相关, 矩阵压缩存储

- 三元组

typedef struct {

    int i, j; // 非零元 的行和列

    int elem;

} Tripe;

typedef struct {

    Tripe Matrix[MAX\_SIZE];

    int mu, nu, tu;  // 矩阵的行, 列数, 及非零元个数

} TMatrix;

结点: Tripe;

关系: TMatrix

特点: 非零元在数组中按行逻辑顺序存储便于进行依次顺序处理矩阵运算, 但是, 如果我想找到一行的非零元, 就比较麻烦, 还是需要从头开始找, 由此引出 行逻辑链接顺序表存储法.

- 行逻辑链接顺序表

typedef struct {

    int i, j; // 非零元 的行和列

    int elem;

} Tripe;

typedef struct {

    Tripe Matrix[MAX\_SIZE];

    int mu, nu, tu; // 矩阵的行, 列数, 及非零元个数

    int rpos[MAXRC+1];  // 各行第一个非零元的位置表

} LMatrix;

结点: Tripe

关系: LMatrix

这个存储结构跟三元组基本上一样, 只是多了一个记录在数组中, 第几个元素还是是第几行的开始非零元. 这里的 rpos[MAXRC+1] 记录的是第几行在数组中的非零元的起始位置, 例如 rpos[2] = 5 表示 第 2 行非零元的起始位置, 在Matrix=[5]

- 十字链表存储发

以上的存储方式, 说白了, 还是顺序存储, 如果矩阵非零元个数和位置变化较大, 就比较适合使用链式存储结构.

typedef struct mxtripe {

    int elem;

    int i, j;

    struct mxtripe \*right;

    struct mxtripe \*end;

}MxTripe, \*OLink;

typedef struct {

    OLink \*rhead;  // rhead 指向的是一个行向量, 该向量指向 元素类型

    OLink \*chead;  // chead 指向的是一个列向量, 该向量指向 元素类型

    int mu, nu, tu;

}CrossList;

rhead, chead 指向的是向量的首地址, 即数组.

[](http://images.cnitblog.com/blog/370445/201303/19105852-16e4616c08ec49249698410f2e1b8099.png)

可见 rhead 指向 行级指针数组, chead 指向 列级指针数组.

十字链表在做矩阵运算时非常方便.

- 树的双亲表示法

typedef struct treenode {

   int elem;

   int parent;

} PT;

typedef struct {

    PT nodes[MAX\_TREE\_SIZE];

    int r, n;  // 根结点和结点总数

} PTree;

[](http://images.cnitblog.com/blog/370445/201303/19135138-8689c60f01fc4b80b723169af07a75f2.png)

结点: PT

关系: PTree, 其中关系也是隐含在结点的 parent中.

这种存储方式, 很显然, 找儿子特别困难. 找parent相对容易.

- 树的孩子链表 表示法

typedef struct CTNode {  // 孩子结点, 此节点如果缺少 child, 保存信息并不完整

    int child;  // 在数组中的下标

    struct CTNode \*next;

} \*ChildPtr;

typedef struct {  // 树中的结点

    int data;

    ChildPtr firstchild;  // 孩子链表头指针

} CTBox;

typedef struct {  // 树结构

    CTBox nodes[MAX\_TREE\_SIZE];

    int n,r;  // 结点数 和 根位置( 在数组中 )

} CTree;

此种结构, 找到孩子很容易, 但是由孩子找 parent 就很麻烦.

[](http://images.cnitblog.com/blog/370445/201303/19135139-4f1c04b0e8f74d4698089a11069480ba.png)

- 树的孩子兄弟 表示法( 也叫二叉树表示法或二叉链表 表示法 ) **推荐**

typedef struct CSNode {

    int elem;

    struct CSNode \*firstchild, \*nextsibling;  // 左孩子, 右兄弟

} CSNode, \*CSTree;

结点: CSNode

关系: 首先定义一个结点为根结点, 然后利用 firstchild 指针指向第一个孩子, 依次继续, 具体结构图, 如下:

[](http://images.cnitblog.com/blog/370445/201303/19163352-b358608bafd04fb0943f3e9d34fad7ba.png)

**二叉树**

- 顺序存储结构

typedef TelemType SqBiTree[MAX\_TREE\_SIZE];

SqBiTree bt;

结点: 存放在数组中.

关系: 通过结点存放在数组中的位置来判断结点之间的关系.

缺点: 浪费很多存储空间, 另外结点之间的关系不明显. 下图中黄颜色的全部是浪费的, 而且还有很多浪费的, 因为是按照完全二叉树的方式存储的.

[](http://images.cnitblog.com/blog/370445/201303/19163353-bdcc9fc19bc8427d919cc25d8b1540a3.png)

- 二叉树, 二叉链表表示法

typedef struct BiNode {

    int elem;

    struct BiNode \*leftChild, \*rightChild;

} BiNode, \*BiTree;

因为 二叉树的特点是最多只有2个儿子, 所以可以分为左右两个儿子, 然后进行存储.

结点: BiNode

关系: leftchild, rigthchild

可以看到, 这种方式的存储方法, 跟实际画图是一样的. 而且这种方式很像 左孩子又兄弟表示法, 这也是树与二叉树互换的依据.

[](http://images.cnitblog.com/blog/370445/201303/19163354-aeac271733cf494eaa76ed6571f299af.png)

- 二叉树, 三叉链表表示法

typedef struct BiNode {

    int elem;

    struct BiNode \*leftChild, \*rightChild, \*parent;

} BiNode, \*BiTree;

从定义上可以看出, 对边二叉链表表示法, 只是多了个指针指向 parent .

结点: BiNode

关系: leftchild, rightchild, parent

[](http://images.cnitblog.com/blog/370445/201303/19163355-ea35ac36ff774d63809b268e4992b3db.png)

**树的遍历**

先序遍历: 根左右

中序遍历: 左根右

后续遍历: 左右根

**森林与二叉树的转换**

由于二叉树和树都可以用 二叉链表作为存储结构, 那么以二叉链表作为媒介可导出树与二叉树之间的一个对应关系, 从物理上, 他们的二叉表是相同的, 只是解释不同. 如下图:

[](http://images.cnitblog.com/blog/370445/201303/19164818-72f682eb63a0411682ee9d8585c42193.png)

[](http://images.cnitblog.com/blog/370445/201303/19164824-cd5425ef9456441988d73325e0d34056.png)

1. 前序遍历、中序遍历、后续遍历

前序遍历：   
    1.访问根节点   
    2.前序遍历左子树   
    3.前序遍历右子树   
中序遍历：   
    1.中序遍历左子树   
    2.访问根节点   
    3.中序遍历右子树   
后序遍历：   
    1.后序遍历左子树   
    2.后序遍历右子树   
    3.访问根节点

层次遍历：只需按层次遍历即可

**一、已知前序、中序遍历，求后序遍历**

例：

前序遍历:         GDAFEMHZ

中序遍历:         ADEFGHMZ

画树求法：第一步，根据前序遍历的特点，我们知道根结点为G

              第二步，观察中序遍历ADEFGHMZ。其中root节点G左侧的ADEF必然是root的左子树，G右侧的HMZ必然是root的右子树。

              第三步，观察左子树ADEF，左子树的中的根节点必然是大树的root的leftchild。在前序遍历中，大树的root的leftchild位于root之后，所以左子树的根节点为D。

              第四步，同样的道理，root的右子树节点HMZ中的根节点也可以通过前序遍历求得。在前序遍历中，一定是先把root和root的所有左子树节点遍历完之后才会遍历右子树，并且遍历的左子树的第一个节点就是左子树的根节点。同理，遍历的右子树的第一个节点就是右子树的根节点。

            第五步，观察发现，上面的过程是递归的。先找到当前树的根节点，然后划分为左子树，右子树，然后进入左子树重复上面的过程，然后进入右子树重复上面的过程。最后就可以还原一棵树了。该步递归的过程可以简洁表达如下：

1 确定根,确定左子树，确定右子树。

2 在左子树中递归。

3 在右子树中递归。

4 打印当前根。

那么，我们可以画出这个二叉树的形状：



那么，根据后序的遍历规则，我们可以知道，后序遍历顺序为：AEFDHZMG

1. 排序算法\*\*\*\*

八种排序算法：



插入排序：

将一个记录插入到已排序好的有序表中，从而得到一个新，记录数增1的有序表。即：先将序列的第1个记录看成是一个有序的子序列，然后从第2个记录逐个进行插入，直至整个序列有序为止。

要点：设立哨兵，作为临时存储和判断数组边界之用。

时间复杂度：**排序是稳定的，**时间复杂度：O（n^2）.

插入排序—希尔排序 缩小增量排序

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

1. 数据库问题需要加强，多表查询
2. 多线程问题

线程的实现方式、死锁、造成死锁的原因、线程的安全性

1. 数据结构问题

2018-9-17

## 3.Java堆栈相关

栈内存：java中栈内存用于存放基本数据类型的值以及引用变量（即对象的引用，存放的是对象在堆内存中的地址），保存局部变量的值

堆内存：java中堆内存用于存放动态产生的数据，即新new出来的对象（区别对象与对象的引用），注意，这里新new出来的对象只有成员变量存放于堆内存中，而同一个类new出来的对象共享该类的同一组成员方法，每创建一个对象时jvm不会在堆内存中复制该类的成员方法。堆内存中包含了常量池。JVM为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量(基本类型，String)和对其他类型、方法、字段的**符号引用**。池中的数据和数组一样通过索引访问。由于常量池包含了一个类型所有的对其他类型、方法、字段的符号引用，所以常量池在Java的动态链接中起了核心作用。**常量池存在于堆中**。

动态链接：java类加载机制，在编译器将java类文件变异成class的二进制文件，在java程序中，当用到某个类时才将该类加载到内存中。

静态链接：C++使用的机制，对于所有的类，C++都将其加载到内存中。

从运行速度上来看，c++优于java，而java的灵活性更好

数据段：用来存放static修饰的静态成员（在java中static的作用就是说明该变量、方法

、代码块是属于类的还是属于实例的）。

代码段：存放从硬盘上读取的源程序代码

## 4.排序算法



String s = new String(“aa”);创建了两个对象：

i> 类加载时，对于一个类，类加载只会进行一次。此类进行加载时，会把字符串abc放进全局的常量池中，进行保存。

ii> 运行时，当你运行程序的时候，常量池中存在字符串abc,于是把字面量abc拿进heap中，使它的引用交给s1。

因此这条语句创建了两个对象。

# 三.Java基本知识零碎

## ==与equeals

==比较的是两个变量的地址。

equals:在object中equals方法直接用的==比较，即比较的也是两个变量的地址。对于基本数据类型int、short、long、boolean、byte、float、double、char，其包装类Integer、Short、Long、Boolean、Byte、Float、Double、Character对equals方法进行了重写，比较的是变量地址所指向的值。String类也重写了equals方法，比较的也是变量的值。理解两个变量==后的结果，必须理解java中常量池的运行原理。

String中重写的equals方法:

public boolean equals(Object anObject) {  
 if (this == anObject) {  
 return true;  
 }  
 if (anObject instanceof String) {  
 String anotherString = (String)anObject;  
 int n = value.length;  
 if (n == anotherString.value.length) {  
 char v1[] = value;  
 char v2[] = anotherString.value;  
 int i = 0;  
 while (n-- != 0) {  
 if (v1[i] != v2[i])  
 return false;  
 i++;  
 }  
 return true;  
 }  
 }  
 return false;  
}

## java IO

IO框架是java程序与外界通信的工具 ，这里的外界可以指代一下几项：1.本地磁盘、远程磁盘文件，2.数据量连接，3.TCP、UDP、HTTP网络通信



1. IO 结构

整个java IO可分为三类：流式部分、非流式部分、其他类

主要的基类：

File：文件特征与管理，用于文件或目录的描述信息，如生成新目录，修改文件名、删除文件、判断文件路径。

InputStream：抽象类，基于字节的输入操作，是所有输入流的父类，定义了所有输入流的共同特征。

OutputStream：抽象类，基于字节的输出操作，是所有输出流的父类，定义了所有输出流都具有的共同特征。

Reader：抽象类，基于字符的输入操作，文件格式操作。

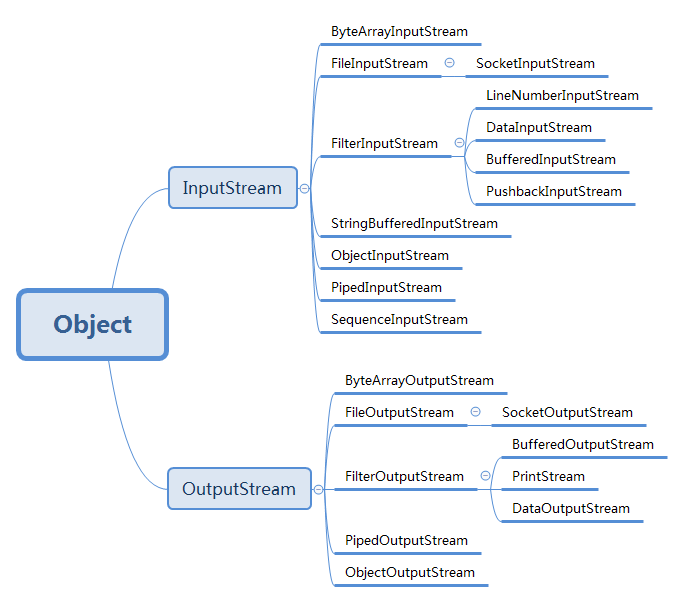
Writer：抽象类，基于字符的输出操作，文件格式操作。

RandomAccessFile：随机文件操作，集成自Object的独立类，功能丰富，可从文件任意位置进行输入输出操作。

流式部分：代表任何有能力产出数据的数据源对象或者是有能力接受数据的接收端对象<Thinking in Java>。流的本质:数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。

java IO的主体部分，使用装饰者模式（Decorator模式）。

流中类的层次结构：





按数据源的不同将io流分为以下几种：

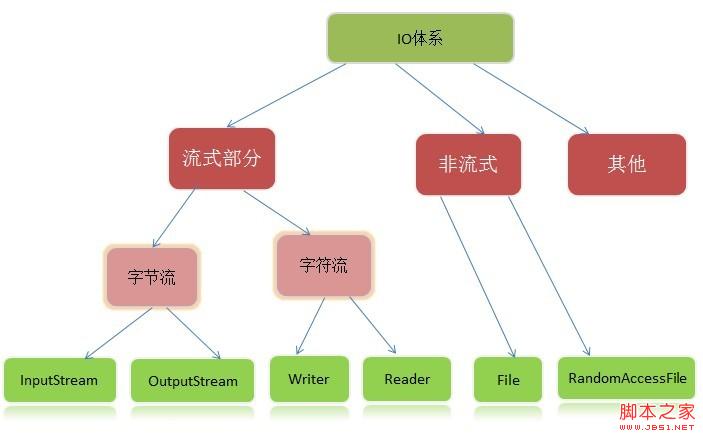
1. 处理文件的流（File）：FileInputStream，FileOutputStream，FileReader、FileWriter
2. 处理字节的流（byte[]）:ByteArrayInputStream、ByteArrayOutputStream
3. 处理字符的流（char）：CharArrayReader、CharArrayWriter
4. 处理字符串的流（String）：StringBufferInputStream、StringReader、StringWriter
5. 网络数据流：InputStream、OutputStream、Reader、Writer

首先看字节流：

字节流以InputStream和OutputStream连个抽象类（abstract class）作为基类，我们平时使用的字节流都是这两个类的子类。

ByteArrayInputStream：字节数组输入流在内存中创建一个字节数组缓冲区，从输入流读取的数据保存在该字节数组缓冲区中。创建字节数组输入流对象有以下几种方式。

接收字节数组作为参数创建：



## java常见异常种类

Java异常都继承了Throwable类，Throwable下又分两个子类：1，Error类代表编译和系统异常，不允许捕获；2.Exception类，代表了java标准库方法所激发的异常，包含运行时异常（RuntimeException）和非运行时异常(Non\_RuntimeException，Exception类下除RuntimeException子类外的其他子类都是非运行时异常类).

运行异常类对应于编译错误，它是指Java程序在运行时产生的由解释器引发的各种异常。运行异常可能出现在任何地方，且出现频率很高，因此为了避免巨大的系统资源开销，编译器不对异常进行检查。所以Java语言中的运行异常不一定被捕获。出现运行错误往往表示代码有错误，如：算数异常（如被0除）、下标异常（如数组越界）等。常见的运行时异常：

ArrayStoreException         试图将错误类型的对象存储到一个对象数组时抛出的异常  
ClassCastException           试图将对象强制转换为不是实例的子类时，抛出该异常  
IllegalArgumentException   抛出的异常表明向方法传递了一个不合法或不正确的参数  
IndexOutOfBoundsException   指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出  
NoSuchElementException       表明枚举中没有更多的元素  
NullPointerException      当应用程序试图在需要对象的地方使用 null 时，抛出该异常

当出现RuntimeException的时候，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。

非运行时异常是RuntimeException以外的异常，类型上都属于Exception类及其子类，又称为检查式异常。如IOException、SQLException等以及用户自定义的Exception异常。对于这种异常，JAVA编译器强制要求我们必需对出现的这些异常进行catch并处理，否则程序就不能编译通过。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆catch块去处理可能的异常。常见的检查式异常：

算术异常类：ArithmeticExecption

空指针异常类：NullPointerException

类型强制转换异常：ClassCastException

数组负下标异常：NegativeArrayException

数组下标越界异常：ArrayIndexOutOfBoundsException

违背安全原则异常：SecturityException

文件已结束异常：EOFException

文件未找到异常：FileNotFoundException

字符串转换为数字异常：NumberFormatException  
操作数据库异常：SQLException  
输入输出异常：IOException  
方法未找到异常：NoSuchMethodException

Java异常机制用到的几个关键字：**try、catch、finally、throw、throws。**  
• **try**        -- 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。  
• **catch**   -- 用于捕获异常。catch用来捕获try语句块中发生的异常。  
• **finally**  -- finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。  
• **throw**  -- 用于抛出异常。  
• **throws**-- 用在方法签名中，用于声明该方法可能抛出的异常。

## HashMap、HashTable、ConcurrentHashMap

HashMap：

* 底层数组+链表实现，无论key还是value都**不能为null**，线程**安全**，实现线程安全的方式是在修改数据时锁住整个HashTable，效率低，ConcurrentHashMap做了相关优化
* 初始size为**11**，扩容：newsize = olesize\*2+1
* 计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

HashTable：

* 底层数组+链表实现，可**以存储null键和null值**，线程**不安全**
* 初始size为**16**，扩容：newsize = oldsize\*2，size一定为2的n次幂
* 扩容针对整个Map，每次扩容时，原来数组中的元素依次重新计算存放位置，并重新插入
* 插入元素后才判断该不该扩容，有可能无效扩容（插入后如果扩容，如果没有再次插入，就会产生无效扩容）
* 当Map中元素总数超过Entry数组的75%，触发扩容操作，为了减少链表长度，元素分配更均匀
* 计算index方法：index = hash & (tab.length – 1)

HashMap的初始值还要考虑加载因子:

* **哈希冲突**：若干Key的哈希值按数组大小取模后，如果落在同一个数组下标上，将组成一条Entry链，对Key的查找需要遍历Entry链上的每个元素执行equals()比较。
* **加载因子**：为了降低哈希冲突的概率，默认当HashMap中的键值对达到数组大小的75%时，即会触发扩容。因此，如果预估容量是100，即需要设定100/0.75＝134的数组大小。
* **空间换时间**：如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率。

HashMap和Hashtable都是用hash算法来决定其元素的存储，因此HashMap和Hashtable的hash表包含如下属性：

* 容量（capacity）：hash表中桶的数量
* 初始化容量（initial capacity）：创建hash表时桶的数量，HashMap允许在构造器中指定初始化容量
* 尺寸（size）：当前hash表中记录的数量
* 负载因子（load factor）：负载因子等于“size/capacity”。负载因子为0，表示空的hash表，0.5表示半满的散列表，依此类推。轻负载的散列表具有冲突少、适宜插入与查询的特点（但是使用Iterator迭代元素时比较慢）

除此之外，hash表里还有一个“负载极限”，“负载极限”是一个0～1的数值，“负载极限”决定了hash表的最大填满程度。当hash表中的负载因子达到指定的“负载极限”时，hash表会自动成倍地增加容量（桶的数量），并将原有的对象重新分配，放入新的桶内，这称为rehashing。

HashMap和Hashtable的构造器允许指定一个负载极限，HashMap和Hashtable默认的“负载极限”为0.75，这表明当该hash表的3/4已经被填满时，hash表会发生rehashing。

“负载极限”的默认值（0.75）是时间和空间成本上的一种折中：

* 较高的“负载极限”可以降低hash表所占用的内存空间，但会增加查询数据的时间开销，而查询是最频繁的操作（HashMap的get()与put()方法都要用到查询）
* 较低的“负载极限”会提高查询数据的性能，但会增加hash表所占用的内存开销

程序猿可以根据实际情况来调整“负载极限”值。

ConcurrentHashMap

* 底层采用分段的数组+链表实现，线程**安全**
* 通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。(读操作不加锁，由于HashEntry的value变量是 volatile的，也能保证读取到最新的值。)
* Hashtable的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术
* 有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁
* 扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

Hashtable和HashMap都实现了Map接口，但是Hashtable的实现是基于Dictionary抽象类的。Java5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

HashMap基于哈希思想，实现对数据的读写。当我们将键值对传递给put()方法时，它调用键对象的hashCode()方法来计算hashcode，然后找到bucket位置来存储值对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当发生碰撞时，对象将会储存在链表的下一个节点中。HashMap在每个链表节点中储存键值对对象。当两个不同的键对象的hashcode相同时，它们会储存在同一个bucket位置的链表中，可通过键对象的equals()方法来找到键值对。如果链表大小超过阈值（TREEIFY\_THRESHOLD,8），链表就会被改造为树形结构。

在HashMap中，null可以作为键，这样的键只有一个，但可以有一个或多个键所对应的值为null。**当get()方法返回null值时，即可以表示HashMap中没有该key，也可以表示该key所对应的value为null**。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个key，应该用**containsKey()**方法来判断。而在Hashtable中，无论是key还是value都不能为null。

Hashtable是线程安全的，它的方法是同步的，可以直接用在多线程环境中。而HashMap则不是线程安全的，在多线程环境中，需要手动实现同步机制。

Hashtable与HashMap另一个区别是HashMap的迭代器（Iterator）是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。

## 同步与锁、线程、线程池、死锁、任务调度

### 1．同步与锁

在并发编程中，经常遇到多个线程访问同一个 共享资源 ，这时候作为开发者必须考虑如何维护数据一致性，在java中synchronized关键字被常用于维护数据一致性。synchronized机制是给共享资源上锁，只有拿到锁的线程才可以访问共享资源，这样就可以强制使得对共享资源的访问都是顺序的，因为对于共享资源属性访问是必要也是必须的。

一般在java中所说的锁就是指的内置锁，每个java对象都可以作为一个实现同步的锁，虽然说在java中一切皆对象， 但是锁必须是引用类型的，基本数据类型则不可以 。每一个引用类型的对象都可以隐式的扮演一个用于同步的锁的角色，执行线程进入synchronized块之前会自动获得锁，无论是通过正常语句退出还是执行过程中抛出了异常，线程都会在放弃对synchronized块的控制时自动释放锁。 获得锁的唯一途径就是进入这个内部锁保护的同步块或方法 。

 对共享资源的访问必须是顺序的，也就是说当多个线程对共享资源访问的时候，只能有一个线程可以获得该共享资源的锁，当线程A尝试获取线程B的锁时，线程A必须等待或者阻塞，直到线程B释放该锁为止，否则线程A将一直等待下去，因此java内置锁也称作互斥锁，也即是说锁实际上是一种互斥机制。

根据使用方式的不同一般我们会将锁分为对象锁和类锁，两个锁是有很大差别的，对象锁是作用在实例方法或者一个对象实例上面的，而类锁是作用在静态方法或者Class对象上面的。一个类可以有多个实例对象，因此一个类的对象锁可能会有多个，但是每个类只有一个Class对象，所以类锁只有一个。 类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定的是实例方法还是静态方法区别的 。

一个synchronized块包含两个部分：锁对象的引用（同步方法），以及这个锁保护的代码块（同步块）。如果作用在实例方法上面，锁就是该方法所在的当前对象，静态synchronized方法会从Class对象上获得锁。

线程同步：多个线程同时执行并访问同一份资源，要确保紫泉正确与否，涉及到线程安全与线程不安全的问题。

并发编程的三个概念：

原子性：一组操作要么全部执行要么全部不执行，不会被任何因素打断。

可见性：多个线程同时访问同一个变量时，一个线程修改了该变量的值，其他线程能立即看到。

有序性：程序的执行按代码的先后顺序执行。

### 2.线程：

#### 程序、进程、线程概念

程序：指令集

进程：程序的一个实例，程序的一次动态执行过程，占用独立的地址空间

线程：在进程内的多条执行路径，进程中的一个单一连续控制过程，进程中的线程共享内存地址空间。

进程线程区别：

1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线；
3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段、数据集、堆等)及一些进程级的资源(如打开文件和信号)，某进程内的线程在其它进程不可见；
4. 调度和切换：线程上下文切换比进程上下文切换要快得多。

线程实现方式：

1.继承Thread并实现run方法

2.实现Runnable并实现run方法，创建Thread类的实例并传入Runnable接口（静态代理模式），避免单继承的局限性，方便共享资源。

启动使用start方法。

3.实现Callable接口，可以抛异常，可以返回值

线程状态：

新生==》就绪==》运行==》阻塞==》死亡

#### 生产者与消费者举例

有两个生产者t1和t2，两个消费者t3和t4，以及一个长度为1的队列。

初始状态，这四个线程全部进入锁池，等待抢占锁。

t3获取到锁，但是队列为空，故t3进入等待池。

t4获取到锁，但是队列为空，故t4进入等待池。

t1获取到锁，生产，队列满，调用notify，唤醒一个线程。由于此时t3和t4都在等待池中，所以会有一个线程从等待池进入锁池，假设此处t3进入锁池。

此时，锁池有t2和t3两个线程，假设t2获取到了锁，但是队列满，故t2进入等待池，放弃锁。

此时，t3获取到锁，消费，notify，由于此时等待池有两个线程t2和t4，假如唤醒的是t2，没问题开始生产，但是若唤醒的是t4，则因队列为空，继续wait。

此时若t1和t3已经执行结束，t1不在生产，t3不再消费，则t2和t4会一直留在锁池，行程死锁。

如果此处使用notifyAll，则会把等待池中所有线程唤醒，不会形成所有线程都位于等待池，无法唤醒的情况，也就不会形成死锁，当然了，使用notifyAll方法会更加低效一些。

如果此处是一个生产者一个消费者的情况，使用notify没有任何问题，且效率更高。

### 3.线程池

为什么使用线程池：如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

线程池的基本要素：

1.任务队列

2.线程池：管理线程 initThread<=core<=maxThread

### 4．死锁

多个线程对同一份资源进行操作会造成线程都处于等待的状态。

死锁是指两个或两个以上的进程（线程）在运行过程中因争夺资源而造成的一种僵局（Deadly-Embrace) ) ，若无外力作用，这些进程（线程）都将无法向前推进。

哲学家吃饭问题。

过多的同步会造成死锁。

解决方案：使用生产者消费者方式，具体由管程法，信号灯法。

### 5.synchronized、voliate

synchronized

概念

synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，也就是平时说的同步方法和同步语句块。如果 再细的分类，synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。

无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁――而且同步方法很可能还会被其他线程的对象访问。每个对象只有一个锁（lock）与之相关联。

实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

用法：

synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线 程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的 synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

某个类的范围，synchronized static aStaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。

几种典型用法：

1.修饰一个代码块，被修饰的代码块称为同步语句块，其作用的范围是大括号{}括起来的代码，作用的对象是调用这个代码块的对象；  
2. 修饰一个方法，被修饰的方法称为同步方法，其作用的范围是整个方法，作用的对象是调用这个方法的对象；  
3. 修改一个静态的方法，其作用的范围是整个静态方法，作用的对象是这个类的所有对象；  
4. 修改一个类，其作用的范围是synchronized后面括号括起来的部分，作用主的对象是这个类的所有对象。

总结：

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。

B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。

C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制

volatile:

一个变量被volatile修饰后具备以下特征;

* 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
* 禁止进行指令重排序。

volatile关键字无法保证操作的原子性

这里需要理解java内存模型等相关概念。

### 6.任务调度

* 使用Timer类及TimerTask类实现：

Timer timer = new Timer();  
timer.schedule(new TimerTask() {  
 @Override  
 public void run() {  
 System.*out*.println("do task....");  
 }  
},new Date(System.*currentTimeMillis*()+2000),300);

* 可以使用Quartz框架

参考文献： <http://www.importnew.com/21897.html>

<http://www.importnew.com/21866.html>

## java反射

## 7.Serializable的作用

工作中我们经常在进行持久化操作和返回数据时都会使用到javabean来统一封装参数，方便操作，一般我们也都会实现Serializable接口，那么问题来了，首先：为什么要进行序列化；其次：每个实体bean都必须实现Serializable接口吗？最后：我做一些项目的时候，没有实现序列化，同样没什么影响，到底什么时候应该进行序列化操作呢？

　　网上找了很多资料，但是感觉大都没有说的很清楚，所以结合自己的理解做一下总结。

　　首先第一个问题，实现序列化的两个原因：1、将对象的状态保存在存储媒体中以便可以在以后重新创建出完全相同的副本；2、按值将对象从一个应用程序域发送至另一个应用程序域。实现Serializable接口的作用是就是可以把对象存到字节流，然后可以恢复，所以你想如果你的对象没实现序列化怎么才能进行持久化和网络传输呢，要持久化和网络传输就得转为字节流，所以在分布式应用中及设计数据持久化的场景中，你就得实现序列化。

　　第二个问题，是不是每个实体bean都要实现序列化，答案其实还要回归到第一个问题，那就是你的bean是否需要持久化存储媒体中以及是否需要传输给另一个应用，没有的话就不需要，例如我们利用fastjson将实体类转化成json字符串时，并不涉及到转化为字节流，所以其实跟序列化没有关系。

　　第三个问题，有的时候并没有实现序列化，依然可以持久化到数据库。这个其实我们可以看看实体类中常用的数据类型，例如Date、String等等，它们已经实现了序列化，而一些基本类型，数据库里面有与之对应的数据结构，从我们的类声明来看，我们没有实现Serializable接口，其实是在声明的各个不同变量的时候，由具体的数据类型帮助我们实现了序列化操作。

另外需要注意的是，在NoSql数据库中，并没有与我们java基本类型对应的数据结构，所以在往nosql数据库中存储时，我们就必须将对象进行序列化，同时在网络传输中我们要注意到两个应用中javabean的serialVersionUID要保持一致，不然就不能正常的进行反序列化。

## 8.transient用法

Java中transient关键字的作用，简单地说，就是让某些被修饰的成员属性变量不被序列化，这一看好像很好理解，就是不被序列化，那么什么情况下，一个对象的某些字段不需要被序列化呢？如果有如下情况，可以考虑使用关键字transient修饰：

1、类中的字段值可以根据其它字段推导出来，如一个长方形类有三个属性：长度、宽度、面积（示例而已，一般不会这样设计），那么在序列化的时候，面积这个属性就没必要被序列化了；

2、其它，看具体业务需求吧，哪些字段不想被序列化；

PS，记得之前看HashMap源码的时候，发现有个字段是用transient修饰的，我觉得还是有道理的，确实没必要对这个modCount字段进行序列化，因为没有意义，modCount主要用于判断HashMap是否被修改（像put、remove操作的时候，modCount都会自增），对于这种变量，一开始可以为任何值，0当然也是可以（new出来、反序列化出来、或者克隆clone出来的时候都是为0的），没必要持久化其值。

Fastjson的@JSONFiled注解的成员变量如果使用了transient标识，该对象json化之后该成员变量不会出现在json字符串中

## 9 fastjson用法

在日志解析,前后端数据传输交互中,经常会遇到字符串(String)与json,XML等格式相互转换与解析，其中json以跨语言，跨前后端的优点在开发中被频繁使用，基本上可以说是标准的数据交换格式。[fastjson](https://github.com/alibaba/fastjson)是一个java语言编写的高性能且功能完善的JSON库，它采用一种“假定有序快速匹配”的算法，把JSON Parse 的性能提升到了极致。它的接口简单易用，已经被广泛使用在缓存序列化，协议交互，Web输出等各种应用场景中。

* JSONArray : 相当于List

 JSONObject: 相当于Map<String,Object>

## 10.java常用类

### 1.java.lang.Object

方法列表：

|  |  |
| --- | --- |
| Method | Description |
| Object clone() | 创建与该对象的类相同的新对象 |
| boolean equals(Object) | 比较两对象是否相等 |
| void finalize() | 当垃圾回收器确定不存在对该对象的更多引用时，对象的圾回收器调用该方法 |
| Class getClass() | 返回一个对象运行时的实例类 |
| int hashCode() | 返回该对象的散列码值 |
| void notify() | 激活等待在该对象的监视器上的一个线程 |
| void notifyAll() | 激活等待在该对象的监视器上的全部线程 |
| String toString() | 返回该对象的字符串表示 |
| void wait() | 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待 |

#### equals方法

public boolean equals(Object obj) {  
 return (this == obj);  
}

比较的是两个对象之间的引用。

#### clone方法

创建一个用户的拷贝，官方描述;

The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:

x.clone() != x

will be true, and that the expression:

x.clone().getClass() == x.getClass()

will be true, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, this is not an absolute requirement.

浅拷贝：

创建一个新对象，然后将当前对象的非静态字段复制到该对象，如果字段类型是值类型（基本类型）的，那么对该字段进行复制；如果字段是引用类型的，则只复制该字段的引用而不复制引用指向的对象。此时新对象里面的引用类型字段相当于是原始对象里面引用类型字段的一个副本，原始对象与新对象里面的引用字段指向的是同一个对象。

Clone方法只能实现浅拷贝

深拷贝：

即将引用类型的属性内容也拷贝一份新的。

如果需要使用clone方法对持有引用对象的对象进行深拷贝，则引用对象也需要实现Cloneable接口并重写clone方法,这样才能实现对象深拷贝。

对于很多引用类型，可以使用序列化对象的方式进行深拷贝。

#### Finalize方法

官方定义

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

The finalize method is never invoked more than once by a Java virtual machine for any given object.

finalize调用时机：

1.所有对象被Garbage Collection时自动调用,比如运行System.gc()的时候.  
2.程序退出时为每个对象调用一次finalize方法。  
3.显式的调用finalize方法

除此以外,正常情况下,当某个对象被系统收集为无用信息的时候,finalize() 将被自动调用,但是jvm不保证finalize()一定被调用,也就是说,finalize()的调用是不确定的,这也就是为什么sun不提倡使用 finalize()的原因. 简单来讲，finalize()是在对象被GC回收前会调用的方法，而System.gc()强制GC开始回收工作纠正，不是强制，是建议，具体执行要看 GC的意思简单地说，调用了 System.gc() 之后，java 在内存回收过程中就会调用那些要被回收的对象的 finalize() 方法。

finalize与system.gc()的区别：

gc 只能清除在堆上分配的内存(纯java语言的所有对象都在堆上使用new分配内存),而不能清除栈上分配的内存（当使用JNI技术时,可能会在栈上分配内 存,例如java调用c程序，而该c程序使用malloc分配内存时）.因此,如果某些对象被分配了栈上的内存区域,那gc就管不着了,对这样的对象进行 内存回收就要靠finalize().  
举个例子来说,当java 调用非java方法时（这种方法可能是c或是c++的）,在非java代码内部也许调用了c的malloc()函数来分配内存，而且除非调用那个了 free() 否则不会释放内存(因为free()是c的函数),这个时候要进行释放内存的工作,gc是不起作用的,因而需要在finalize()内部的一个固有方法 调用它(free()).  
finalize的工作原理应该是这样的：一旦垃圾收集器准备好释放对象占用的存储空间，它首先调用finalize()，而且只有在下一次垃圾收集过程中，才会真正回收对象的内存.所以如果使用finalize()，就可以在垃圾收集期间进行一些重要的清除或清扫工作.

#### wait方法

Causes the current thread to wait until another thread invokes the [notify()](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Object.html#notify()) method or the [notifyAll()](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Object.html#notifyAll()) method for this object. In other words, this method behaves exactly as if it simply performs the call wait(0)

wait方法会释放锁

线程进入等待状态，直到它被其他线程通过notify()或者notifyAll唤醒。该方法只能在**同步方法**中调用。如果当前线程不是锁的持有者，该方法抛出一个IllegalMonitorStateException异常。

Object.wait()和Object.notify()和Object.notifyall()必须写在synchronized方法内部或者synchronized块内部，这是因为：**这几个方法要求当前正在运行object.wait()方法的线程拥有object的对象锁。**即使你确实知道当前上下文线程确实拥有了对象锁，也不能将object.wait()这样的语句写在当前上下文中。

### 2.java.land.Thread

#### 1.join方法

将被join线程合并到当前线程并阻塞当前线程

#### 2.yield方法

A hint to the scheduler that the current thread is willing to yield its current use of a processor.

暂停当前线程让出cpu时间让其他线程执行。

#### 3.sleep方法

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors.

**暂停当前线程，不释放锁**

#### 4. currentThread方法

Returns a reference to the currently executing thread object.

获取当前执行的线程。

#### 5.[setPriority](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Thread.html#setPriority(int))方法

Changes the priority of this thread.

First the checkAccess method of this thread is called with no arguments. This may result in throwing a SecurityException.

设置线程优先级，是线程执行的优先级概率，不是绝对值。

#### getClass方法

Returns the runtime class of this Object. The returned Class object is the object that is locked by static synchronized methods of the represented class.

(实例)object.getClass与（类）Object.class的区别：

object.getClass().getName()是在程序运行时获得运行时实例的类类型(即当前实际实例的类)。

Object.class.getName()是在编译阶段就确定了的，与运行时的状态无关。

#### notify方法

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

只会唤醒等待该锁的其中一个线程。

#### notifyAll方法

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

notifyAll会唤醒所有线程并根据算法选取其中一个线程获取锁

notify与notifyAll方法的区别：

对象内部锁概念：

每个对象都拥有两个池，分别为锁池(EntrySet)和(WaitSet)等待池。

锁池：假如已经有线程A获取到了锁，这时候又有线程B需要获取这把锁(比如需要调用synchronized修饰的方法或者需要执行synchronized修饰的代码块)，由于该锁已经被占用，所以线程B只能等待这把锁，这时候线程B将会进入这把锁的锁池。

等待池：假设线程A获取到锁之后，由于一些条件的不满足(例如生产者消费者模式中生产者获取到锁，然后判断队列为满)，此时需要调用对象锁的wait方法，那么线程A将放弃这把锁，并进入这把锁的等待池。

如果有其他线程调用了锁的notify方法，则会根据一定的算法从等待池中选取一个线程，将此线程放入锁池。  
如果有其他线程调用了锁的notifyAll方法，则会将等待池中所有线程全部放入锁池，并争抢锁。

# 四.Spring零碎知识

## 1.Spring 的启动过程

## 2.springMVC的启动过程

ContextLoaderListener：启动spring容器

DispatcherServlet：启动springmvc

部署web应用时，web容器（比如Tomcat）会读取配置在web.xml中的监听器，从而启动spring容器。有了spring容器之后，我们才能使用spring的IOC AOP等特性。弄清spring容器启动流程，有利于理解spring IOC中的各种特性，比如BeanPostProcessor，MessageSource，ApplicationListener等。我们先来看下容器启动流程中涉及的主要类。

**ContextLoaderListener**：注册在web.xml中，web应用启动时，会创建它，并回调它的initWebApplicationContext()方法，从而创建并启动spring容器。必须继承ServletContextListener。

**WebApplicationContext**：用于web应用的spring容器上下文，它代表了spring容器，继承自ApplicationContext。是一个接口，在ContextLoader.properties配置文件中可以声明它的实现类。默认实现类为XmlWebApplicationContext。ApplicationContext继承自BeanFactory，并扩展了它的很多功能。

**ServletContext**：web容器（如tomcat）的上下文，不要和ApplicationContext搞混了。

Web.xml中的配置：

<context-param>

        <param-name>contextConfigLocation</param-name>

        <param-value>/WEB-INF/applicationContext.xml</param-value>

    </context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

 </listener>

这段是加载spring配置文件，初始化上下文，ContextLoaderListener是一个实现了ServletContextListener接口的监听器，在启动项目时会触发contextInitialized方法（该方法主要完成ApplicationContext对象的创建），在关闭项目时会触发contextDestroyed方法（该方法会执行ApplicationContext清理操作）

①启动项目时触发contextInitialized方法，该方法就做一件事：通过父类contextLoader的initWebApplicationContext方法创建Spring上下文对象。

②initWebApplicationContext方法做了三件事：创建 WebApplicationContext；加载对应的Spring文件创建里面的Bean实例；将WebApplicationContext放入 ServletContext（就是Java Web的全局变量）中。

③createWebApplicationContext创建上下文对象，支持用户自定义的 上下文对象，但必须继承自ConfigurableWebApplicationContext，而Spring MVC默认使用ConfigurableWebApplicationContext作为ApplicationContext（它仅仅是一个接口）的实 现。

④configureAndRefreshWebApplicationContext方法用 于封装ApplicationContext数据并且初始化所有相关Bean对象。它会从web.xml中读取名为 contextConfigLocation的配置，这就是spring xml数据源设置，然后放到ApplicationContext中，最后调用传说中的refresh方法执行所有Java对象的创建。

⑤完成ApplicationContext创建之后就是将其放入ServletContext中，注意它存储的key值常量。

## 3.classpth与classpath\*

classpath：只会到你的class路径中查找找文件。

classpath\*：不仅包含class路径，还包括jar文件中（class路径）进行查找。

注意： 用classpath\*:需要遍历所有的classpath，所以加载速度是很慢的；因此，在规划的时候，应该尽可能规划好资源文件所在的路径，尽量避免使用classpath\*。

当项目中有多个classpath路径，并同时加载多个classpath路径下（此种情况多数不会遇到）的文件，\*就发挥了作用，如果不加\*，则表示仅仅加载第一个classpath路径。

从上面使用的场景看，可以在路径上使用通配符\*进行模糊查找。比如：

<param-value>classpath:applicationContext-\*.xml</param-value>

"\*\*/"表示的是任意目录；"\*\*/applicationContext-\*.xml"表示任意目录下的以"applicationContext-"开头的XML文件。

程序部署到tomcat后，src目录下的配置文件会和class文件一样，自动copy到应用的WEB-INF/classes目录下；classpath:与classpath\*:的区别在于，前者只会从第一个classpath中加载，而 后者会从所有的classpath中加载。

如果要加载的资源，不在当前ClassLoader的路径里，那么用classpath:前缀是找不到的，这种情况下就需要使用classpath\*:前缀。

在多个classpath中存在同名资源，都需要加载时，那么用classpath:只会加载第一个，这种情况下也需要用classpath\*:前缀。

## 4.web.xml中配置DispatcherServlet的url-pattern配置成/\*与/的区别

< url-pattern > / </ url-pattern >   不会匹配到\*.jsp，即：\*.jsp不会进入spring的 DispatcherServlet类 。  
< url-pattern > /\* </ url-pattern > 会匹配\*.jsp，会出现返回jsp视图时再次进入spring的DispatcherServlet 类，导致找不到对应的controller所以报404错。

总之，关于web.xml的url映射的小知识:  
< url-pattern>/</url-pattern>  会匹配到/login这样的路径型url，不会匹配到模式为\*.jsp这样的后缀型url  
< url-pattern>/\*</url-pattern> 会匹配所有url：路径型的和后缀型的url(包括/login,\*.jsp,\*.js和\*.html等)

1. 首先/这个是表示默认的路径，及表示：当没有找到可以匹配的URL就用这个URL去匹配。

2. 在springmvc中可以配置多个DispatcherServlet，比如： 配置多个DispatcherServlet有/和/\*，先匹配的是/\*这个

3. 当配置相同的情况下，DispathcherServlet配置成/和/\*的区别

<一>/：使用/配置路径，直接访问到jsp，不经springDispatcherServlet

<二> /\*：配置/\*路径，不能访问到多视图的jsp

## 5.几种视图模板技术

Jsp

Freemaker

Thymeleaf

Velocity

## 6.多视图解析配置

项目中同时需要使用thymeleaf和jsp时，需要同时使用这两种模板的视图解析器，

首先需要指定解析器的先后顺序，使用order参数，order越小优先级越高,但仅指定order存在一个问题：视图解析器中存在一个buildView方法，该方法一定会创建一个View；所以View永远不为null（即使该视图不存在），所以当视图能够进行解析时而这种解析又没有找到实际的视图，会直接返回404页面，而优先级低的视图不会再进行解析，此时需要使用视图解析器的viewNames属性指定处理视图的规则，具体配置如下：

<!--jsp view resolver-->  
<bean id="viewResolver1" class="org.springframework.web.servlet.view.InternalResourceViewResolver" p:order="1">  
 <!--<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>-->  
 <property name="contentType" value="text/html"/>  
 <property name="prefix" value="/WEB-INF/"/>  
 <property name="suffix" value=".jsp"/>  
 <property name="viewNames" value="jsp\*"/>  
</bean>  
<!--thymeleaf视图解析器-->  
<bean id="templateResolver" class="org.thymeleaf.spring4.templateresolver.SpringResourceTemplateResolver">  
 <property name="prefix" value="/WEB-INF/"/>  
 <property name="suffix" value=".html"/>  
 <property name="templateMode" value="HTML"/>  
 <property name="cacheable" value="true"/>  
 <property name="characterEncoding" value="UTF-8"/>  
</bean>  
<bean id="springTemplateEngine" class="org.thymeleaf.spring4.SpringTemplateEngine"  
p:templateResolver-ref="templateResolver"  
p:enableSpringELCompiler="true"/>  
<bean id="viewResolver2" class="org.thymeleaf.spring4.view.ThymeleafViewResolver"  
p:templateEngine-ref="springTemplateEngine"  
p:characterEncoding="UTF-8"  
p:order="0"  
p:viewNames="thymeleaf\*"/>

## 7.web flow

## 8.security/oAuth

OAuth是一种授权协议，他的作用时提供认证和授权的规范（标准）

Spring Security是OAuth的一种实现(shiro是另一种实现)

OAuth的一些概念：

* 角色
* 第单方应用程序（客户端）
* 资源所有者（用户）
* HTTP服务提供者
  + - 认证服务器
    - 资源服务器
* 客户端授权模式
* 简单模式
* 授权码模式
* 密码模式
* 客户端模式
* 访问令牌
* Access Token
* Refresh Token

**Authentication 鉴权 和 Authorization 授权**

用户名和密码通过验证，登陆成功，这就是**鉴权 Authentication。**

鉴权(authentication): 你是谁

授权(authorization): 你能干什么

## 9.token/refresh token的概念

## 10.JSON Web Token (JWT)

## 11.spring mvc接收json参数

### 1.ajax请求前后端对应格式

在jquery的ajax中，如果没加contentType:"application/json"，那么data就应该对应的是json对象，反之，如果加了contentType:"application/json"，那么ajax发送的就必须是字符串。

不管前台发送的是json字符串还是对象，服务器本质上收到的都是字符流，那么为什么ajax又可以直接传对象呢？因为不加contentType:"application/json"的时候，发送类型变为默认的application/x-www-form-urlencoded，而这种方式会以键值对的形式将对象序列化，所以传进去的对象实际上还是变成了字符流

### 2．put、delete请求

对于rest风格的put、delete请求，需要在web.xml中配置如下的filter，不然接收不到请求参数

<!--1 -->

<filter>

<filter-name>HiddenHttpMethodFilter</filter-name>

<filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>HiddenHttpMethodFilter</filter-name>

<servlet-name>springMvc</servlet-name>

</filter-mapping>

<!--2 -->

<filter>

<filter-name>httpPutFormcontentFilter</filter-name>

<filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>httpPutFormcontentFilter</filter-name>

<servlet-name>springMvc</servlet-name>

</filter-mapping>

## 12. Spring transation

### 1.概念

事务是访问数据库的一个操作序列，数据库应用系统通过事务集来完成对数据库的存取。**事务的正确执行使得数据库从一种状态转换为另一种状态**。Spring 的事务在数据库事务之上。

使用事务的原因：

应用中需要保证用户的操作的可靠性和完整性，有些操作必须作为一组原子操作（如转账、下单减库存等）提交到数据库，如果其中的一个操作失败，其它操作也不应该生效，这就是数据库事务的概念（下单过程中减了库存，随后的下单记录添加失败，那么库存就不应该减，所以应该将这个步骤作为一个事务提交到数据库，以保证数据完整性）。

### 2.事务的ACID原则

* 原子性atomicity：不可分割性，事务要么全部执行，要么全部不执行。
* 一致性consistency：事务的正确执行使数据库从一种状态转移到另一种状态。
* 隔离性isolation：在事务正确提交之前，不允许把事务对该数据的改变提供给任何其他事务，即在事务正确提交之前，它可能的结果不应该显示给其他事务。
* 持久性durability：事务正确提交之后，其结果将永远保存在数据库之中，即使在事务提交之后有了其他故障，事务的处理结果也会得到保存。

### 3.并发事务存在的问题

务A和事务B操纵的是同一个资源，事务A有若干个子事务，事务B也有若干个子事务，事务A和事务B在高并发的情况下，会出现各种各样的问题（读一致性问题）：

* 脏读：指**事务A读到了事务B还没有提交的数据。**比如银行取钱，事务A开启事务，此时切换到事务B，事务B开启事务-->取走100元，此时切换回事务A，事务A读取的肯定是数据库里面的原始数据，因为事务B取走了100块钱，并没有提交，数据库里面的账务余额肯定还是原始余额，这就是脏读。
* 不可重复读：指**在一个事务里面读取了两次某个数据，读出来的数据不一致**。事务A开启事务-->查出银行卡余额为1000元，此时切换到事务B事务B开启事务-->事务B取走100元-->提交，数据库里面余额变为900元，此时切换回事务A，事务A再查一次查出账户余额为900元，这样对事务A而言，在同一个事务内两次读取账户余额数据不一致，这就是不可重复读。
* 幻读：指**在一个事务里面的操作中发现了未被操作的数据**。比如学生信息，事务A开启事务-->修改所有学生当天签到状况为false，此时切换到事务B，事务B开启事务-->事务B插入了一条学生数据，此时切换回事务A，事务A提交的时候发现了一条自己没有修改过的数据，这就是幻读，就好像发生了幻觉一样。幻读出现的前提是并发的事务中有事务发生了插入、删除操作。

### 4.事务隔离级别

为解决以上事务并发产生的问题，规定了事务级别，**事务隔离级别越高，在并发下会产生的问题就越少，但同时付出的性能消耗也将越大，因此很多时候必须在并发性和性能之间做一个权衡。事务隔离级别分为以下几种。**

* **DEFAULT** 默认隔离级别，每种数据库支持的事务隔离级别不一样，如果Spring配置事务时将isolation设置为这个值的话，那么将使用底层数据库的默认事务隔离级别。顺便说一句，如果使用的MySQL，可以使用"**select @@tx\_isolation**"来查看默认的事务隔离级别
* **READ\_UNCOMMITTED** 读未提交，即能够读取到没有被提交的数据，所以很明显这个级别的隔离机制无法解决脏读、不可重复读、幻读中的任何一种，因此很少使用。
* **READ\_COMMITED** 读已提交，即能够读到那些已经提交的数据，自然能够防止脏读，但是无法限制不可重复读和幻读
* **REPEATABLE\_READ**重复读取，即在数据读出来之后加锁，类似"select \* from XXX for update"，明确数据读取出来就是为了更新用的，所以要加一把锁，防止别人修改它。REPEATABLE\_READ的意思也类似，读取了一条数据，这个事务不结束，别的事务就不可以改这条记录，这样就解决了脏读、不可重复读的问题，但是幻读的问题还是无法解决。
* **SERLALIZABLE** 串行化，最高的事务隔离级别，不管多少事务，挨个运行完一个事务的所有子事务之后才可以执行另外一个事务里面的所有子事务，这样就解决了脏读、不可重复读和幻读的问题了。

##### 5.事务传播特性

**当事务开始时，一个事务上下文已经存在，此时可以指定一个事务性方法的执行行为。事务传播特性在spring中的分类如下。**

* propagation\_requierd：如果当前没有事务，就新建一个事务，如果已存在一个事务中，加入到这个事务中，这是最常见的选择。
* propagation\_supports：支持当前事务，如果没有当前事务，就以非事务方法执行。
* propagation\_mandatory：使用当前事务，如果没有当前事务，就抛出异常。
* propagation\_required\_new：新建事务，如果当前存在事务，把当前事务挂起。
* propagation\_not\_supported：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
* propagation\_never：以非事务方式执行操作，如果当前事务存在则抛出异常。
* propagation\_nested：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与propagation\_required类似的操作

### ****5.Spring 事务****

#### 1.接口

**TransactionDefinition（事务规则：**设置事务的一些属性，上文提到的，使用DefaultTransactionDefinition默认实现一般可以满足要求，或者可以扩展接口，实现自己的定义**）**

**PlatformTransactionManager（事务管理：**spring没有直接管理事务，而是将事务管理的责任委托给JTA或持久化机制的某个特定平台的事务实现。spring的事务管理器充当了特定平台事务的代理。）

**TransactionStatus（事务状态：**代表一个新的或已经存在的事务，控制事务执行和查询事务状态**）**

#### 2.实现方式

* 编程式事务：在业务代码中显示编写事务逻辑，编程式事务可基于三个核心接口直接实现，也可以使用TransactionTemplate
* 声明式事务：在配置文件中声明事务

# 五.Web及HTTP理解

## 1.HTTP协议

http hypertext transfer protocol（超文本传输协议），TCP/IP协议的一个应用层协议，用于定义WEB浏览器与WEB服务器之间交换数据的过程。客户端连上web服务器后，若想获得web服务器中的某个web资源，需遵守一定的通讯格式，HTTP协议用于定义客户端与web服务器通迅的格式。

一些特点

* 基于请求和响应，客户端请求，服务器端响应
* 无连接、无状态 请求响应过程完成连接即终止，不会记录连接产生的数据
* 请求报文：请求行、请求报头信息、空行和请求数据4个部分组成
* 请求行：请求方法字段、URL字段和HTTP协议版本字段

post [https://blog.csdn.net/qq\_37964547/article/details/80314412 HTTP/1.1](https://blog.csdn.net/qq_37964547/article/details/80314412%20HTTP/1.1)

请求方法： GET、POST、HEAD、PUT、DELETE、OPTIONS、TRACE、CONNECT

GET：当客户端要从服务器中读取文档时，使用GET方法。GET方法要求服务器将URL定位的资源放在响应报文的数据部分，回送给客户端。使用GET方法时，请求参数和对应的值附加在URL后面，利用一个问号（“?”）代表URL的结尾与请求参数的开始，传递参数长度受限制。

POST：当客户端给服务器提供信息较多时可以使用POST方法。POST方法将 数据发送给目的服务器，并且没有数据长度限制 。

URL(Uniform Resource Locator) 地址用于描述一个网络上的资源

* 请求头部为请求报文添加了一些附加信息，由“名/值”对组成，每行一对，名和值之间使用冒号分隔
* 用于区分请求头部信息和请求正文
* 请求正文（可选）比如GET请求就没有请求正文，若方法字段是POST,则通常来说此处放置的就是要提交的数据；比如要使用POST方法提交一个表单，其中有user字段中数据为“admin”, password字段为123456，那么这里的请求数据就是 user=admin&password=123456，使用&来连接各个字段
* 响应报文：包括响应行、响应头、响应体
* 响应行一般由协议版本、状态码及其描述组成 比如 HTTP/1.1 200 OK；

一般的常见状态码有以下：

100~199：表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程。

200~299：表示成功接收请求并已完成整个处理过程。常用200

300~399：为完成请求，客户需进一步细化请求。例如：请求的资源已经移动一个新地址、常用302(意味着你请求我，我让你去找别人),307和304(我不给你这个资源，自己拿缓存)

400~499：客户端的请求有错误，常用404(意味着你请求的资源在web服务器中没有)403(服务器拒绝访问，权限不够)

500~599：服务器端出现错误，常用500

* 响应头：响应头用于描述服务器的基本信息，以及数据的描述，服务器通过这些数据的描述信息，可以通知客户端如何处理等一会儿它回送的数据。

常见的响应头部信息如下：

Allow：服务器支持哪些请求方法(如GET、POST等)。

Content-Encoding：文档的编码(Encode)方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE4、IE5才支持它。因此，Servlet应该通过查看Accept-Encoding头(即request.getHeader(“Accept- Encoding”))检查浏览器是否支持gzip，为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面。

Content-Length：表示内容长度。只有当浏览器使用持久HTTP连接时才需要这个数据。如果你想要利用持久连接的优势，可以把输出文档写入 ByteArrayOutputStram，完成后查看其大小，然后把该值放入Content-Length头，最后通过byteArrayStream.writeTo(response.getOutputStream()发送内容。

Content- Type：表示后面的文档属于什么MIME类型。Servlet默认为text/plain，但通常需要显式地指定为text/html。由于经常要设置 Content-Type，因此HttpServletResponse提供了一个专用的方法setContentType。

Date：当前的GMT时间，例如，Date:Mon,31Dec200104:25:57GMT。Date描述的时间表示世界标准时，换算成本地时间，需要知道用户所在的时区。你可以用setDateHeader来设置这个头以避免转换时间格式的麻烦。

Expires：告诉浏览器把回送的资源缓存多长时间，-1或0则是不缓存。

Last-Modified：文档的最后改动时间。客户可以通过If-Modified-Since请求头提供一个日期，该请求将被视为一个条件GET，只有改动时间迟于指定时间的文档才会返回，否则返回一个304(Not Modified)状态。Last-Modified也可用setDateHeader方法来设置。

Location：这个头配合302状态码使用，用于重定向接收者到一个新URI地址。表示客户应当到哪里去提取文档。Location通常不是直接设置的，而是通过HttpServletResponse的sendRedirect方法，该方法同时设置状态代码为302。

Refresh：告诉浏览器隔多久刷新一次，以秒计。

Server：服务器通过这个头告诉浏览器服务器的类型。Server响应头包含处理请求的原始服务器的软件信息。此域能包含多个产品标识和注释，产品标识一般按照重要性排序。Servlet一般不设置这个值，而是由Web服务器自己设置。

Set-Cookie：设置和页面关联的Cookie。Servlet不应使用response.setHeader(“Set-Cookie”, …)，而是应使用HttpServletResponse提供的专用方法addCookie。

Transfer-Encoding：告诉浏览器数据的传送格式。

WWW-Authenticate：客户应该在Authorization头中提供什么类型的授权信息?在包含401(Unauthorized)状态行的应答中这个头是必需的。例如，response.setHeader(“WWW-Authenticate”, “BASIC realm=\”executives\”“)。注意Servlet一般不进行这方面的处理，而是让Web服务器的专门机制来控制受密码保护页面的访问

* 响应体 响应体就是响应的消息体，如果是纯数据就是返回纯数据，如果请求的是HTML页面，那么返回的就是HTML代码，如果是JS就是JS代码。

## 2．RESTFUL思想

为什么使用rest：

轻量，直接基于http，不再需要任何别的诸如消息协议。get/post/put/delete为CRUD操作

面向资源，一目了然，具有自解释性。

数据描述简单，一般以xml，json做数据交换。

无状态，在调用一个接口（访问、操作资源）的时候，可以不用考虑上下文，不用考虑当前状态，极大的降低了复杂度。

简单、低耦合

是近年来移动互联网的发展，各种类型的Client层出不穷，RESTful可以通过一套统一的接口为 Web，iOS和Android提供服务。另外对于广大平台来说，比如Facebook platform，微博开放平台，微信公共平台等，它们不需要有显式的前端，只需要一套提供服务的接口，于是RESTful更是它们最好的选择

**Resource Representational State Transfer**,表述性状态转移，其中涉及到的HTTP协议知识点：

* HTTP 中包含**动词**（或方法）： GET、POST、PUT、PATCH 还有 DELETE 是最常用的。
* REST 是**面向资源**的，一个资源被一个 **URI** 所标识，比如 /articles/。
* **端点（endpoint）**，一般指动词与 URI 的组合，比如 GET: /articles/。
* 一个端点可以被解释为对某种资源进行的某个动作。比如， POST: /articles 可能代表“创建一个新的 article”。
* 在业务领域，我们常常可以将**动词**和 **CRUD**（增删查改）关联起来：GET 代表查，POST代表增，PUT 和 PATCH 代表改（注: PUT 通常代表整体更新，而 PATCH 代表局部更新），而 DELETE 代表删。

使用rest需要注意的地方：

1. 不要返回纯文本

2.避免在URI中使用动词

3.使用复数名词来描述资源 GET /article/2/

4.在响应中返回错误详情，当 API 服务器处理错误时，如果能够在返回的 JSON body 中包含错误信息，对于接口调用者来说，会一定程度上帮助他们完成调试。

5.保持HTTP status code 的一致性，应当返回与返回错误类型相一致的具有一定含义的状态码，比如，如果一个 POST 类型的端点返回 201 Created，那么所有的 POST 端点都应返回同样的状态码。这样做的好处在于，调用者无需在意端点返回的状态码取决于某种特殊条件，也就形成了一致性

6.不要嵌套资源

嵌套资源的写法

GET: /authors/12/articles/

扁平化形式优于嵌套形式，应使用如下的形式，同样是articles资源使用与上文一样的路由

GET: /articles/?author\_id=12

7.尾部斜杆问题 大多数的 web 框架都针对 URL 是否包含尾部斜杠，进行了优雅地处理并提供定制选项，如果可以的话，找到它并开启这项功能。

8.使用querystring完成筛选和分页功能：

GET: /articles/?page=1&page\_size=10

9.认证与授权，出现安全错误提示时，容易混淆这两个概念：

* 用户是否未提供身份验证凭据？认证是否还有效？这种类型的错误一般是未认证（401 Unauthorized）。
* 用户经过了正常的身份验证，但没有访问资源所需的权限？这种一般是未授权（403 Forbidden）

10.合理使用restful框架

## 3.过滤器、监听器、拦截器、servlet的区别

servlet是一种运行服务器端的java应用程序，具有独立于平台和协议的特性，可以动态生成web页面它工作在客户端请求与服务器响应的中间。Servlet是配置在web.xml中的。

servlet的生命周期

Servlet生命周期分为三个阶段：

　　1，初始化阶段  调用init()方法

　　2，响应客户请求阶段　　调用service()方法

　　3，终止阶段　　调用destroy()方法

Servlet初始化阶段：

　　在下列时刻Servlet容器装载Servlet：

　　1，Servlet容器启动时自动装载某些Servlet，实现它只需要在web.XML文件中的<Servlet></Servlet>之间添加如下代码：

|  |
| --- |
| <loadon-startup>1</loadon-startup> |

　　2，在Servlet容器启动后，客户首次向Servlet发送请求

　　3，Servlet类文件被更新后，重新装载Servlet

　　Servlet被装载后，Servlet容器创建一个Servlet实例并且调用Servlet的init()方法进行初始化。在Servlet的整个生命周期内，init()方法只被调用一次。

Servlet中的过滤器Filter是实现了javax.servlet.Filter接口的服务器端程序，主要的用途是过滤字符编码、做一些业务逻辑判断等。其工作原理是，只要你在web.xml文件配置好要拦截的客户端请求，它都会帮你拦截到请求，此时你就可以对请求或响应(Request、Response)统一设置编码，简化操作；同时还可进行逻辑判断，如用户是否已经登陆、有没有权限访问该页面等等工作。它是随你的web应用启动而启动的，只初始化一次，以后就可以拦截相关请求，只有当你的web应用停止或重新部署的时候才销毁,过滤器与servlet一样需要在web.xml中配置.

Servlet的监听器Listener，它是实现了javax.servlet.ServletContextListener 接口的服务器端程序，它也是随web应用的启动而启动，只初始化一次，随web应用的停止而销毁。主要作用是： 做一些初始化的内容添加工作、设置一些基本的内容、比如一些参数或者是一些固定的对象等等。监听器也需要在web.xml中进行配置。

用途：

1、统计在线人数和在线用户

2、系统启动时加载初始化信息

3、统计网站访问量

4、记录用户访问路径

4、跟Spring结合

拦截器是在面向切面编程中应用的，就是在你的service或者一个方法前调用一个方法，或者在方法后调用一个方法。是基于JAVA的反射机制。拦截器不是在web.xml，比如struts在struts.xml中配置。

启动先后顺序：

web.xml的加载顺序是：context-param->listener->filter->servlet

加载配置文件后初始化拦截器，当有对action的请求的时候，调用interceptor方法，最后也是根据服务器停止进行销毁。

## 4.cookies、token与session

Session应用

随着交互式Web应用的兴起，像在线购物网站，需要登录的网站等等，马上就面临一个问题，那就是要管理会话，必须记住哪些人登录系统，  哪些人往自己的购物车中放商品，  也就是说我必须把每个人区分开，这就是一个不小的挑战，因为HTTP请求是无状态的，所以想出的办法就是给大家发一个会话标识(session id), 说白了就是一个随机的字串，每个人收到的都不一样，  每次大家向我发起HTTP请求的时候，把这个字符串给一并捎过来， 这样我就能区分开谁是谁了。

每个人只需要保存自己的session id，而服务器要保存所有人的session id ！  如果访问服务器多了， 就得由成千上万，甚至几十万个。

这对服务器说是一个巨大的开销 ， 严重的限制了服务器扩展能力， 比如说我用两个机器组成了一个集群， 小F通过机器A登录了系统，  那session id会保存在机器A上，  假设小F的下一次请求被转发到机器B怎么办？  机器B可没有小F的 session id啊。

有时候会采用一点小伎俩： session sticky ， 就是让小F的请求一直粘连在机器A上， 但是这也不管用， 要是机器A挂掉了， 还得转到机器B去。

那只好做session 的复制了， 把session id  在两个机器之间搬来搬去



后来有个叫Memcached的支了招： 把session id 集中存储到一个地方， 所有的机器都来访问这个地方的数据， 这样一来，就不用复制了， 但是增加了单点失败的可能性， 要是那个负责session 的机器挂了，  所有人都得重新登录一遍， 估计得被人骂死。



也尝试把这个单点的机器也搞出集群，增加可靠性， 但不管如何， 这小小的session 对我来说是一个沉重的负担。

好处：对登陆的用户进行验证，保证系统及用户的安全性，防止非法用户访问资源

缺点：维护session给服务器造成了额外的开销

Token的使用

维护session是为了对用户的合法性进行验证，需要解决问题的本质是验证合法性，于是又有了token。

比如说， 小F已经登录了系统， 我给他发一个令牌(token)， 里边包含了小F的 user id， 下一次小F 再次通过Http 请求访问我的时候， 把这个token 通过Http header 带过来不就可以了。

不过这和session id没有本质区别啊， 任何人都可以可以伪造，  所以我得想点儿办法， 让别人伪造不了。

那就对数据做一个签名吧， 比如说我用HMAC-SHA256 算法，加上一个只有我才知道的密钥，  对数据做一个签名， 把这个签名和数据一起作为token ，   由于密钥别人不知道， 就无法伪造token了。



这个token 我不保存，  当小F把这个token 给我发过来的时候，我再用同样的HMAC-SHA256 算法和同样的密钥，对数据再计算一次签名， 和token 中的签名做个比较， 如果相同， 我就知道小F已经登录过了，并且可以直接取到小F的user id ,  如果不相同， 数据部分肯定被人篡改过， 我就告诉发送者： 对不起，没有认证。



Token 中的数据是明文保存的（虽然我会用Base64做下编码， 但那不是加密）， 还是可以被别人看到的， 所以我不能在其中保存像密码这样的敏感信息。

当然， 如果一个人的token 被别人偷走了， 那我也没办法， 我也会认为小偷就是合法用户， 这其实和一个人的session id 被别人偷走是一样的。

这样一来， 我就不保存session id 了， 我只是生成token , 然后验证token ，  我用我的CPU计算时间获取了我的session 存储空间 ！

解除了session id这个负担，  可以说是无事一身轻， 我的机器集群现在可以轻松地做水平扩展， 用户访问量增大， 直接加机器就行。   这种无状态的感觉实在是太好了！

基于Token的身份验证的过程如下:

1.用户通过用户名和密码发送请求。

2.程序验证。

3.程序返回一个签名的token 给客户端。

4.客户端储存token,并且每次用于每次发送请求。

5.服务端验证token并返回数据。

 每一次请求都需要token。token应该在HTTP的头部发送从而保证了Http请求无状态。我们同样通过设置服务器属性Access-Control-Allow-Origin:\* ，让服务器能接受到来自所有域的请求。需要主要的是，在ACAO头部标明(designating)\*时，不得带有像HTTP认证，客户端SSL证书和cookies的证书。

优点：节省了存储session的空间，无状态、可扩展，有利于服务的横向扩展

缺点：对CPU密集型服务不利，和session存在被盗风险

Cookies的使用

cookie 是一个非常具体的东西，指的就是浏览器里面能永久存储的一种数据，仅仅是浏览器实现的一种数据存储功能。

cookie由服务器生成，发送给浏览器，浏览器把cookie以kv形式保存到某个目录下的文本文件内，下一次请求同一网站时会把该cookie发送给服务器。由于cookie是存在客户端上的，所以浏览器加入了一些限制确保cookie不会被恶意使用，同时不会占据太多磁盘空间，所以每个域的cookie数量是有限的。

## 5.sso

Single Sign On，[单点登录](https://baike.baidu.com/item/%E5%8D%95%E7%82%B9%E7%99%BB%E5%BD%95/4940767)。SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。它是目前比较流行的企业业务整合的解决方案之一。

## 6.https

Hyper Text Transfer Protocol over Secure Socket Layer ，超文本传输安全协议，相比http多了加密操作，一般通过ssl（安全套接字层）或TLS传输层加密协议（补充版的ssl）完成。

## 日志使用

java领域存在多种日志框架，目前常用的日志框架包括Log4j 1，Log4j 2，Commons Logging，Slf4j，Logback，Jul。

* **Log4j** Apache Log4j是一个基于Java的日志记录工具。它是由Ceki Gülcü首创的，现在则是Apache软件基金会的一个项目。 Log4j是几种Java日志框架之一。
* **Log4j 2** Apache Log4j 2是apache开发的一款Log4j的升级产品。
* **Commons Logging** Apache基金会所属的项目，是一套Java日志接口，之前叫Jakarta Commons Logging，后更名为Commons Logging。
* **Slf4j** 类似于Commons Logging，是一套简易Java日志门面，本身并无日志的实现。（Simple Logging Facade for Java，缩写Slf4j）。
* **Logback** 一套日志组件的实现(Slf4j阵营)。
* **Jul** (Java Util Logging),自Java1.4以来的官方日志实现。

Log4j

日志级别

我们现在要调用logger的方法，在这个Logger对象中，有很多方法，所以要先了解log4j的日志级别，log4j规定了默认的几个级别：trace<debug<info<warn<error<fatal。级别之间是包含的关系，意思是如果你设置日志级别是trace，则大于等于这个级别的日志都会输出。

trace： 是追踪，就是程序推进以下，你就可以写个trace输出，所以trace应该会特别多，不过没关系，我们可以设置最低日志级别不让他输出。

debug： 调试么，我一般就只用这个作为最低级别，trace压根不用。是在没办法就用eclipse或者idea的debug功能就好了么。

info： 输出一下你感兴趣的或者重要的信息，这个用的最多了。

warn： 有些信息不是错误信息，但是也要给程序员的一些提示，类似于eclipse中代码的验证不是有error 和warn（不算错误但是也请注意，比如以下depressed的方法）。

error： 错误信息。用的也比较多。

fatal： 级别比较高了。重大错误，这种级别你可以直接停止程序了，是不应该出现的错误么！

日志使用：很简单，1.引入依赖包 2.添加配置文件，如果是log4j则在classpath根路径下添加log4j.properties文件，如果是slf4j+logback则在classpath路径下添加logback.xml或logback-test.xml文件，配置好相关配置项，项目启动后会自动记录日志

# 六．电商概念

## 1.SKU与SPU

SKU=stock keeping unit(库存量单位)。SKU即库存进出计量的单位， 可以是以件、盒、托盘等为单位。在服装、鞋类商品中使用最多最普遍。 例如纺织品中一个SKU通常表示：规格、颜色、款式。也有人解释说SKU就是库存的最小单位，在服装行业，正常情况是“单款单色单码”。

SPU = Standard Product Unit （标准化产品单元）。SPU是商品信息聚合的最小单位，是一组可复用、易检索的标准化信息的集合，该集合描述了一个产品的特性。SPU是商品信息聚合的最小单位，是一组可复用、易检索的标准化信息的集合，该集合描述了一个产品的特性。通俗点讲，属性值、特性相同的商品就可以称为一个SPU。例如，iphone7就是一个SPU，路虎车也是一个SPU，这个与商家无关，与颜色、款式、套餐也无关。在商品信息电子化过程中，商品的特性可以由多个“属性及对应的属性值对”进行描述。“属性及对应的属性值对”完全相同的商品，可以抽象成为一个 SPU。同 时，这些“属性及对应的属性值对”也在SPU中固化下来，逐步标准化。基于SPU的商品信息结构，可以实现丰富的应用，比如商品信息与资讯、评论、以及其它SPU的整合。从这个意义上讲，比较购物的产品库以SPU为标准来建立是最合适的。

# 七、spring boot

# 八、spring cloud

# 九、设计模式

## 1.静态代理

角色：

真是角色：需要执行具体动作的角色

代理角色：持有真是角色的角色

以上两个角色实现相同的接口，由代理角色持有真是角色并执行真实角色的动作。

## 2.单例

确保一个类只有一个对象在运行中。

懒汉式：

1.构造器私有化，避免外部直接创建对象

2.声明一个私有的静态对象

3.创建一个对外的公共的静态方法，访问一个变量，如果没有，创建该对象。

懒汉式是非线程安全的，需要加synchronized进行同步（使用double check双重检车的方式）

饿汉式：

1.构造器私有化

2.声明私有的静态属性，同时创建该对象

3.对外提供访问属性的静态方法

利用类在加载时使用的特性，为了进一步提高效率，可以使用静态内部类

# 十、数据库

## 1.数据库事务配置修改

首先说明一下MySQL查看和修改事务隔离级别的几个命令：

* 查看事务隔离级别使用select @@tx\_isolation
* 修改当前会话事务隔离级别使用**SET session TRANSACTION ISOLATION LEVEL Serializable;**（参数可以为：Read uncommitted|Read committed|Repeatable read|Serializable）
* 修改全局事务隔离级别使用**SET global TRANSACTION ISOLATION LEVEL Serializable;**（参数可以为：Read uncommitted|Read committed|Repeatable read|Serializable）

修改了会话的事务隔离级别，比如MyBatis，getSqlSession()的时候，只针对这一次拿到的Session有效；比如CMD命令行，只对这一次的窗口有效。

修改了全局的事务隔离级别，那么针对此后所有的会话有效，**当前已经存在的会话不受影响**。

# 十一、mybatis

## 1.mybatis原理

## 2.mybatis与spring整合原理