10/9/17

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。

1. 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。

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

### 一、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）。

##### 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。

##### 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。

### 二、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。

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

2018-9-17

## 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



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。

## ==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。

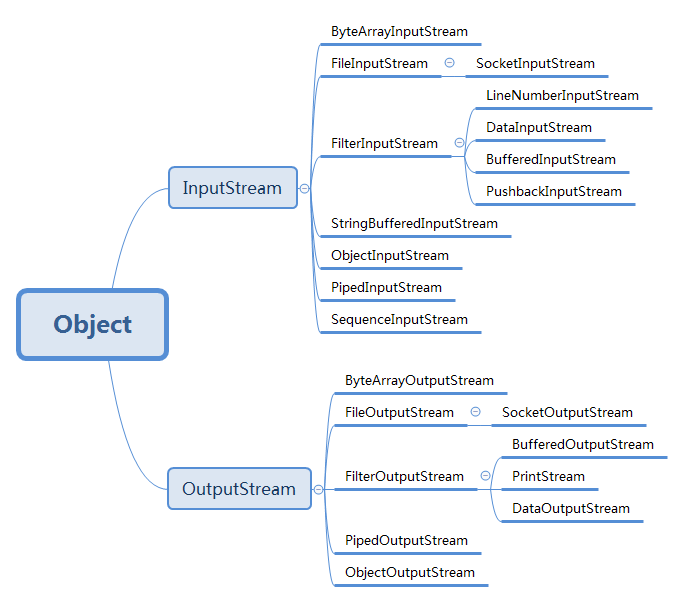
Writer：抽象类，基于字符的输出操作，文件格式操作。

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：

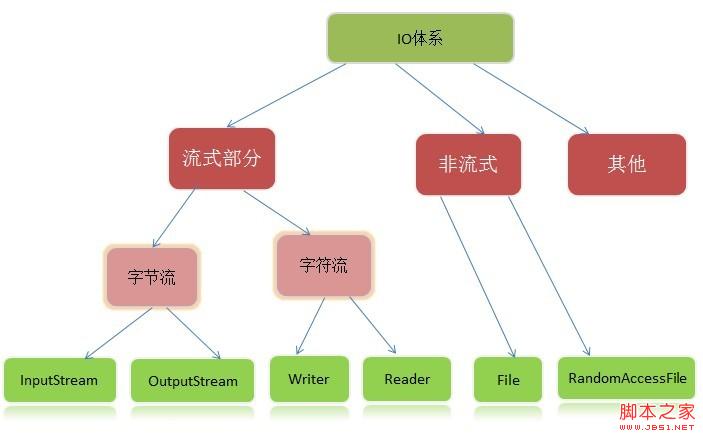
1. 处理文件的流（File）：FileInputStream，FileOutputStream，FileReader、FileWriter
2. 处理字节的流（byte[]）:ByteArrayInputStream、ByteArrayOutputStream
3. 处理字符的流（char）：CharArrayReader、CharArrayWriter
4. 处理字符串的流（String）：StringBufferInputStream、StringReader、StringWriter
5. 网络数据流：InputStream、OutputStream、Reader、Writer

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。

## 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。

## Spring 的启动过程

## springMVC的启动过程

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。

## 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

## 10.java反射