**Java常用知识总结**

# 关于前端模板

## jsp

### Jsp内置对象

 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。

## freemaker

## velocity

## thymeleaf

# 数据结构及算法相关（简单描述）

数组与链表的优缺点

数组：连续存储，寻址容易，查询快，增删效率低。

链表：占用不连续空间，寻址困难，查询慢，增删效率高。

## 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。

## 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

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

## 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

## 排序算法

### 冒泡排序

基本原理

以升序为例：

1. 在一个元素个数为n的数组中首先比较相邻的两个元素，如果前面的元素大于后面的元素，则交换顺序，这样经过一轮比较最大的元素会被放在数组的最后，比较一轮比较的次数为n-1次。
2. 第二轮比较可以剔除掉已经确定的最大的数值，只比较前n-1个数，这一轮比较需要n-2次交换
3. 以此类推，经过n-1轮比较能得出最后的排序结果

一个简单的优化:再进行冒泡排序时，可以对每一轮排序是否进行了交换设置一个标志，如果进行了交换，则表示还未排好，需要继续进行下一轮，如果没有进行交换，则表示排序已经完成，可以结束排序。

## 查找算法



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。

# 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

### 2.1 IO的概念

IO框架是java程序与外界通信的工具 ，这里的外界可以指以下几项：

1.本地磁盘、远程磁盘文件.

2.数据库连接.

3.TCP、UDP、HTTP网络通信

IO流指的是输入输出流，用来处理设备上的数据，这里的设备指硬盘、内存、键盘录入、网络传输等。

Java流以程序为中心，程序读取内容使用输入流，程序写内容使用输出流。



1. IO 结构

整个java IO可分为三类：

流式部分：IO主体

非流式部分：主要包含一些辅助流式部分的类，如：File类、RandomAccessFile类和FileDescriptor等类；

其他类：文件读取部分的与安全相关的类，如：SerializablePermission类，以及与本地操作系统相关的文件系统的类，如：FileSystem类和Win32FileSystem类和WinNTFileSystem类。



### 2.2流两种经典分类：

1.字节流与字符流

 因为数据编码的不同，而有了对字符进行高效操作的流对象。

字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节。

字节流一次读入或读出8位二进制 (InputStream、OutputStream)，字节流能读写所有类型文件。

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。

字符流一次读入或读出16位二进制(Reader、Writer)字符流不能用于复制图片和视频，会造成复制后的文件无法打开

字符流与字节流之间的转换：转换流（InputStreamReader/OutputStreamWriter）,转换流有两个作用：

1. 将字节流转换为字符流，只能处理纯文本内容
2. 处理过程中可以指定字符集

2.节点流和处理流

节点流：直接从数据源或目的地读写数据的流称为节点流。

节点流包括：FileInputStream/FileOutputStream（文件字节输入输出流）、ByteArrayInputStream/ByteArrayOutputStream（字节数组输入输出流，不要使用多态操作）、FileReader/FileWriter（文件字符输入输出流）

处理流：不直接连接到数据源或目的地，对其他流进行封装，以简化操作和提高效率的流称为处理流。处理流使用装饰器模式。

处理流包括：

BufferedInputStream/BufferedOutputStream（字节缓冲流）、BufferedReader/BufferedWriter、（字符缓冲流，不要使用多态操作）、**InputStreamReader/OutputStreamWriter(字节字符转换流)、DataInputStream/DataOutputStream（基本数据类型读写流）、ObjectInputStream/OutputStream(对象输入输出流，序列化和反序列化流，被序列化或反序列化的类必须实现了Serializable接口)、**

**PrintStream(打印流，System.out的类型)、PrintWriter(打印输出)、SequenceInputStream（输入合并流，将多个输入流合并程一个流 ）**

处理流释放时只需要释放外层处理流，不需要手动去释放里层的节点流，jdk会自动找到里层的节点流进行释放。

所有的IO流必须使用节点流进行操作，处理流提高了操作效率。

### 2.3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。

File类代表抽象的文件，这个文件可能不存在，可能是目录，可能是文件。

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。

#### 文件乱码问题

文件编码：由字节码到字符码

文件解码：由字符码到字节码

文件乱码的原因：

1.编码解码字符集不一致

2.字节数不够

### 2.4体系结构

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 不要使用FileReader、FileWriter处理图片，会出现文件打不开的情况。
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

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。字节数组流不依赖与系统的io直接操作java内存，不需要直接关闭（不需要直接调用close方法）。

ByteArrayOutputStream不需要指定数据源（不需要直接指定写入的文件），直接使用Java内存

2.4 IO操作的基本步骤

1.确定数据源

2.选择IO流

3.确定读写方式

4.释放系统资源（io流关闭规则：先打开的后关闭，后打开的先关闭）

对于节点流和处理流，处理规则是先关闭外层的流，再关闭里层的流。

关于关闭流可以做如下封装：

//关闭流

public static void close(Closeable... ios){

for(Closeable io:ios){

if(null != io){

try {

io.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

2.5 common io的常用功能

1.文件、目录遍历（过滤等功能）

2.文件、目录复制

3.文件读写

问题1：

阿里面试题：FileInputStream在使用完以后，不关闭流，想二次使用可以怎么操作？

使用public void mark(int readlimit)和public void reset() throws IOException 这两个方法实现，首先在节点流上套一层处理流（这里使用BufferedInputStream，重写了mark/reset方法，支持mark/reset操作，ByteArrayInputStream也重写了mark/reset方法）,首先在文件开始读取时mark一下：

bufferedInputStream.mark((int)file.length());

在读取完成后reset一下即可回到mark的位置重新读取

bufferedInputStream.reset();

说明：mark中的参数readlimit，the maximum limit of bytes that can be read before the mark position becomes invalid.

在标记位置无效之前可以读取的最大字节限制。

mark在JAVA中的实现是和缓冲区相关的。只要缓冲区够大，mark后读取的数据没有超出缓冲区的大小，mark标记就不会失效。如果不够大，mark后又读取了大量的数据，导致缓冲区更新，原来标记的位置自然找不到了。

因此，mark后读取多少字节才失效，并不完全由readlimit参数确定，也和BufferedInputStream类的缓冲区大小有关。 如果BufferedInputStream类的缓冲区大小大于readLimit，在mark以后只有读取超过缓冲区大小的数据，mark标记才会失效。

## Java Exception

### 异常机制：

当程序出现错误时,程序安全退出的机制。Java采用面向对象的方式处理异常。

Java中两种异常的处理方式：

1.抛出异常：在执行一个方法时，如果发生异常，则这个方法生成代理该异常的一个对象（Exception 的子类），停止当前执行路径，并把异常对象提交给jre。

2.捕获异常：jre得到该异常后，寻找相应的代码来处理异常。Jre在方法调用栈中查找，从生成异常的地方开始回溯，直到找到相应的异常处理代码为止。

### 异常体系架构



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；在编程中不需要处理。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).非运行时异常又称CheckedException（需要程序员捕获处理的异常），运行时异常的子类称为UncheckException（不需要捕获处理的异常）。

1.RuntimeException

运行时异常类指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：

ArithmeticException 算数异常

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           强制转型产生的异常

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  数组下标越界异常

NoSuchElementException       表明枚举中没有更多的元素  
NullPointerException      空指针异常

NumberFormatException 数值强转异常，Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

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： 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。

2.UncheckedException

编译期就会发现的异常，程序无法通过编译。

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

UncheckedException处理方式：

1.try-catch-finally捕获出

2.throw抛出异常

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。存在多个catch时，子类异常在前，父类异常在后

• 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。

3.3异常处理原则

1.避免使用异常处理代替错误处理

2.处理异常不可代替简单测试

3.不要进行小粒度的异常处理，影响程序可读性

4.异常往往在高层处理，由调用者处理，层层外抛

## HashMap、HashTable、ConcurrentHashMap

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。Hashtable是个过时的集合类，存在于Java API中很久了。在Java 4中被重写了，实现了Map接口，所以自此以后也成了Java集合框架中的一部分。

1. HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。
2. HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。
3. 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。这条同样也是Enumeration和Iterator的区别。
4. 由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。
5. HashMap不能保证随着时间的推移Map中的元素次序是不变的。

HashMap可以通过下面的语句进行同步：

Map m = Collections.synchronizeMap(hashMap);

HashTable：

1. 底层数组+单向链表实现，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
2. 初始size为11，扩容：newsize = olesize\*2+1
3. 计算index的方法：index = (hash & 0x7FFFFFFF) % tab.length

HashTable：

1. 底层数组+链表实现，可以存储null键和null值，线程不安全
2. 初始size为16，扩容：newsize = oldsize\*2，size一定为2的n次幂
3. 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
4. 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）
5. 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
6. 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

1. 底层采用分段的数组+链表实现，线程安全
2. 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。)
3. 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
4. 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
5. 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。

## 线程、同步与锁、线程池、死锁、任务调度

### 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：

方式一：

步骤一：

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。

步骤二：

启动使用start方法。（start不保证立即运行，由cpu调度）

方式二：

实现Callable接口，可以抛异常，可以返回值,这种方式具体实现步骤：

1.创建目标对象：

2.创建执行服务

3.提交执行

4.获取结果

5.关闭服务

多线程原则：多用实现Runnable接口的方式，少用继承Thread类的方式

线程状态：

新生：new Thread()

就绪：四种方式能处于就绪状态：1.start(),2.阻塞事件解除，3 yield(),4.JVM将本地线程切换到其他线程

运行：（CPU调度到）

阻塞：四种原因： 1.sleep(),2.wait(),3.join(),4.其他的阻塞操作，如IO操作中的read(),write().

死亡：两个原因：1.线程执行完正常推出 2.~~调用stop()~~（不推荐使用）或遇意外终止。

Stop()方法的替换，将stop方法替换为只修改一些变量用于指示目标线程应该停止运行的代码。

守护线程：为用户线程服务，JVM停止不用等待守护线程执行完毕，JVM等待所有用户线程执行完毕后才会停止。将用户线程设置为daemon:

thread.setDaemon(true);

开辟多线程，每个线程都会有自己的工作空间。线程空间与主内存空间的数据存在交换（线程将主存空间的值拷贝到工作空间中，完成run调用后又将工作空间中的值拷贝到主存中，这造成了非同步多线程的不安全问题）。

### 同步与锁

并发的概念：同一个对象被多个线程同时操作，此时存在线程不安全的问题（数据不准确），使用同步解决问题。

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。

线程同步实现的两个条件：

1.形成队列等待

2.加上锁机制（被加锁的是对象）（加锁与释放锁会引起上下文切换，影响性能）

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。

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 。

使用synchronize同步时，要注意锁定范围的问题。范围太大影响效率，范围太小数据又不准确，合理考虑锁定范围的大小，使用双重检查（double check）来避免效率低下的问题。

### 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：避免在同一个代码块中同时持有多个对象锁。使用生产者消费者方式，具体有管程法，信号灯法。

解决线程通信的问题：

管程法（特点是解耦）：

生产者：产生数据的模块（方法、对象、线程、进程）

消费者：处理数据的模块（方法、对象、线程、进程）

缓冲区：生产者将数据放入缓冲区，消费者从缓冲区拿走数据。

信号灯法：使用标志位标识标识当前应该由哪个线程执行。

生产者消费者模式需要使用到Object类中的四个相关方法：

1.wait()：放弃锁,进入阻塞状态

2.wait(long timeout):等待指定的毫秒数

3.notify():唤醒一个处于等待状态的线程

4.notifyAll():唤醒同一个对象上所有调用wait方法的线程，优先级高的线程调用可能性更高。

以上四个方法只能放在同步方法或同步代码块中，否则会抛出异常。

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。

### 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

线程池具体用法

### 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;

1.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2.值对其他线程来说是立即可见的。

3.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。

### 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

Quartz中的DSL 特定领域语言，简单使用：流模式（方法链）

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

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

### happendBefore（指令重排）

指令的步骤

1.从内存中获取指令

2.将指令解码翻译

3.操作

4.将操作结果写回寄存器中

当进行耗时操作时，cpu会判断前后两条指令是否存在关系，不存在关系则会提前执行不相关指令。

### ThreadLocal

ThreadLocal即每个线程的存储区域。

### 可重入锁

当某个线程试图获取一个已经由它自己持有的锁时，这个请求会立刻成功，并且这个锁的计数值会加1.当线程退出同步代码块时，计数器会递减，当基数值为0时，锁释放。

JDK提供了ReetranLock可重入锁类，可以直接使用

另外：悲观锁、乐观锁

悲观锁：Synchronized是独占锁即悲观锁，会导致其他需要锁的线程挂起，等待持有锁的线程释放锁

乐观锁：每次不加锁假设没有冲突去执行，如果因为冲突而失败则重试，直到成功为止。

乐观锁的一种实现：Compare and swap CAS比较并交换。使用原子性的操作：如AtomicInteger

### JUC并发编程

并发高级知识

## 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。

## 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

## 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>

## java常用类、接口

API文档查看方法：

查看某个类的具体使用时，按以下的方式去查阅API：

* 1. 查看类的继承体系
  2. 查看类的构造方法（某些工具类不提供对外的构造方法，只需要调用其静态方法或是要求该类的对象在系统中唯一存在使用了单例模式而提供了单例的方法）
  3. 查看该类的方法（是否时静态方法，参数，返回值）

### 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。

用法：

被拷贝对象必须实现Cloneable接口并重写clone方法

#### 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。

### 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时间进入就绪状态，让其他线程执行。（礼让）

#### 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，实际用法举例：**

**1.模拟网络延时**

**2.模拟倒计时**

#### 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。

public final boolean isAlive(): A thread is alive if it has been started and has not yet died.线程是否存活

### java.lang.String

不可变（finnal修饰）的Unicode字符序列

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

只能赋值一次，不能再改。

关于 == 的比较

//编译器做了优化，这两个值存放在同一个地址中

String str1 = "hello "+"world";

String str2 = "hello world";

System.out.println(str1 == str2);//true

String str3 = "hello ";

String str4 = "world";

String str5 = str3+str4;

System.out.println(str2 == str5);//false,编译器只知道字面量，不知道实际的值

System.out.println(str3+str4 == str5);//false; 编译器只知道字面量，不知道实际的值

#### 3.1Java常量池概念：

Java虚拟机内存分布：



**程序计数器**是jvm执行程序的流水线，存放一些跳转指令。

**本地方法栈**是jvm调用操作系统方法所使用的栈。

**虚拟机栈**是jvm执行java代码所使用的栈。

**方法区**存放了一些常量、静态变量、类信息等，可以理解成class文件在内存中的存放位置。

**虚拟机堆**是jvm执行java代码所使用的堆。

  Java中的常量池，实际上分为两种形态：**静态常量池**和**运行时常量池**。

**静态常量池**，即\*.class文件中的常量池，class文件中的常量池不仅仅包含字符串(数字)字面量，还包含类、方法的信息，占用class文件绝大部分空间。

**运行时常量池**是jvm虚拟机在完成类装载操作后，将class文件中的常量池载入到内存中，并保存在**方法区**中，我们常说的常量池，就是指方法区中的运行时常量池。

运行时常量池相对于CLass文件常量池的另外一个重要特征是**具备动态性**，Java语言并不要求常量一定只有编译期才能产生，也就是并非预置入CLass文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的就是**String类的intern()**方法。

#### 3.2常用方法

1. public native String intern();

关于native关键字：Java平台有个用户和本地C代码进行互操作的API，称为Java Native Interface (Java本地接口)。native是与C++联合开发的时候用的，java自己开发不用的。native关键字的函数都是操作系统实现的，java只能调用

String的intern()方法会查找在常量池中是否存在一份equal相等的字符串,如果有则返回该字符串的引用,如果没有则添加自己的字符串进入常量池。

2. public char charAt(int index)

Returns the char value at the specified index. An index ranges from 0 to length() - 1. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

If the char value specified by the index is a [surrogate](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Character.html#unicode), the surrogate value is returned.

返回下标索引下的char值

3.public boolean equalsIgnoreCase(String anotherString)

忽略大小写比较常量的值

4. public boolean equals(Object anObject)

比较字符串值是否相等

5. public int indexOf(String str, int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

返回第一个出现指定字符串的下标

6. public String substring(int beginIndex)

从指定下标开始截取之后的所有字符串

7. public String substring(int beginIndex, int endIndex)

截取指定区间的字符串

8. public String toLowerCase()/public String toUpperCase()

大小写转换，生成新的字符串，老字符串不变

9. public String trim()

去除前后空格，生成了新的字符串，老字符串不变

注意：String类是不可变字符序列，如果反复给同一个String引用赋值，会产生不同的实际对象，这样会造成程序效率非常低下，千万不要使用该方式操作字符串。这时使用StringBuilder会效率更高。

### java.lang.StringBuilder/ java.lang.StringBuffer

StringBuilder：可变的字符序列，线程不安全，效率高，StringBuilder在实际中用的比较多

StringBuffer：与StringBuilder都继承自AbstractStringBuilder，线程安全，效率低

StringBuilder的常见方法:

public StringBuffer append(char c):追加字符，返回自身

public StringBuffer insert(int offset, char c)：在指定位置插入字符，返回自身,链式调用

public StringBuffer reverse()：倒转字符序列

public int lastIndexOf(String str)：查找字符序列中最后一次出现指定序列的索引

public int indexOf(String str) ：查找字符序列中第一次出现指定序列的索引

public StringBuffer delete(int start, int end)：删除指定区间序列，返回自身

### java.lang.System

The System class contains several useful class fields and methods. It cannot be instantiated.

提供很多有用的系统方法及成员变量，不可被实例化。

5.1.常见方法

1. public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

数组拷贝方法。

利用arraycopy能实现数组的以下操作：

1.数组拷贝，将一个数组指定的某些元素拷贝到另一个数组的指定位置

2.删除数组中的元素并将后面的数组元素向前移（实现方法：利用arraycopy拷贝要删除元素之后的数组到删除元素开始的位置，并把后面多余的数组元素置为null）

3.数组长度扩容（实现方法：首先新建一个长度更大的数组，再将原数组拷贝到新数组中）

### java.util.Arrays

数组工具类

#### 6.1数组定义与初始化

一维数组：

1.声明并开辟数组：

String[] array1 = new String[100];

赋值：

array1[0] = “aa”;

2.静态初始化数组

String[] array2 = {“aa”,”bb”,”cc”};

二维数组：

1.声明并开辟数组：

int[][] array3 = new int[3][10];

int[][] array3 = new int[3][];

后面的长度可指定也可不指定,如果未指定需要再对每个数组的每一维进行具体初始化：

array3[0] = new int[10];

这种方式下，二维数组每一维的长度可以有不同的长度。

2.静态初始化数组

int[][] array5 = {{1,2,3},{2,3,4,5,6},{23,5,6,6567,342,213,1,3,5,6,7,87,8,8,9,9,6,4,3,}};

#### 6.2常用方法

1. public static String toString(Object[] a)

Returns a string representation of the contents of the specified array. If the array contains other arrays as elements, they are converted to strings by the [Object.toString()](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Object.html#toString()) method inherited from Object, which describes their *identities* rather than their contents.

打印一个数组，其中的Object对象需要重写toString()方法

普通的数组toString打印的时数组的哈希值，没有办法打印数组中元素的具体内容，使用该方法能打印数组中的具体元素信息。

2. public static void sort(Object[] a)

排序方法，该方法时在原数组的基础上直接修改的，执行该方法会改变原数组值。对于Object对象如果需要按指定规则排序，需要实现Comparable接口，并实现compareTo方法。

3. public static int binarySearch(Object[] a, Object key)

Searches the specified array for the specified object using the binary search algorithm. The array must be sorted into ascending order according to the [natural ordering](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Comparable.html) of its elements (as by the[sort(Object[])](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/util/Arrays.html#sort(java.lang.Object[])) method) prior to making this call. If it is not sorted, the results are undefined. (If the array contains elements that are not mutually comparable (for example, strings and integers), it *cannot* be sorted according to the natural ordering of its elements, hence results are undefined.) If the array contains multiple elements equal to the specified object, there is no guarantee which one will be found.

二分法查找，查找数组a中值为key的元素返回元素索引，未找到返回-1。注意：在查找前必须对数组进行排序，否则会返回未知结果。

7. java.lang.Comparable

比较类大小并排序接口

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

方法：

int compareTo(T o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

比较当前类与指定类大小，返回负数代表小，返回0表示相等，返回正数表示大。

### jva.util.Scanner

用于从标准输入端（console）读入内容。

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

常用方法：

public String next()：Finds and returns the next complete token from this scanner. 获取输入的字符串，必须时有效字符，通过next方法获取到的字符串不包含空格（除第一个空格外，其他空格后面的字符不会被读取到）

public boolean hasNext(String pattern)：是否包含下一个有效的字符（简单理解为非空格）

public String nextLine()：以Enter为结束符,也就是说 nextLine()方法返回的是输入回车之前的所有字符。

public boolean hasNextLine()：如果此扫描器的输入中有另一行，则返回true。

## cglib与jdk动态代理

动态代理是为了解决静态代理频繁编写代理功能的缺点。

### Cglib

### Jdk动态代理

JDK自带不需要导入额外的jar

缺点：真实对象必须实现接口，利用反射实现，效率不高

使用到的API（反射包下的）

Java.lang.reflect.Proxy

Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.

java.lang.reflect Interface InvocationHandler

InvocationHandler is the interface implemented by the invocation handler of a proxy instance.

## try …catch…finally 语法

语法糖：

语法糖是在语言中增加的某种语法，在不影响功能的情况下为程序员提供更方便的使用方式。

资源：

使用之后需要释放或者回收的都可以称为资源，比如JDBC的connection连接，文件IO的各种类。

在这里我们可以简单理解为实现了  java.lang.AutoCloseable/java.io.Closeable接口的类对象。

**能用try-with-resources语法糖的资源**

实现了 java.lang.AutoCloseable/java.io.Closeable 的类对象

例子：

String bufferSugar = null;

try (BufferedReader readerSugar = new BufferedReader(new FileReader("src/testRead.txt"))) {

bufferSugar = readerSugar.readLine();

System.out.println(bufferSugar);

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

## atomic与 volatile的区别

## 垃圾回收原理

## java容器

概念：能存储其他类型的java类

### 1.总体结构图



### 2.泛型

概念：

数据类型的参数化。泛化的参数实际上也是指的类型，所以调用泛型必须传入实际的参数。

泛型的好处：

简单易用

类型安全 泛型的主要目标是实现java的类型安全。 泛型可以使编译器知道一个对象的限定类型是什么，这样编译器就可以验证这个类型

消除了强制类型转换 使得代码可读性好，减少了很多出错的机会

实现原理：

泛型的实现是靠类型擦除技术，类型擦除是在编译期完成的，也就是在编译期，编译器会将泛型的类型参数都擦除成它的限定类型，如果没有则擦除为object类型之后在获取的时候再强制类型转换为对应的类型。 在运行期间并没有泛型的任何信息，因此也没有优化。

### 3.集合具体接口类分析

#### 1.collection接口

The root interface in the *collection hierarchy*.

容器层级中的顶层接口。

基本方法：

int size()：获取容器大小

boolean isEmpty()：判断容器是否为空

boolean contains(Object o)：判断容器是否包含某个元素

Object[] toArray()：Returns an array containing all of the elements in this collection。将容器中的元素转化为数组。

boolean add(E e)：向容器中添加一个元素

boolean remove(Object o)：移除容器中的一个元素，实际上时移除容器中保存该对象的地址，不是删除实际的对象

boolean containsAll(Collection<?> c)：判断容器是否包含传入容器中的所有内容

boolean addAll(Collection<? extends E> c)：向容器中添加传入容器中的所有内容

boolean removeAll(Collection<?> c)：移除容器中包含传入容器中的所有内容，如果本容器中包含的传入容器的元素存在重复，则重复的元素会都被移除，返回值问题：true， if this list changed as a result of the call，如果被操作的list被改变了，返回true

void clear()：清空容器

boolean retainAll(Collection<?> c)：求两个容器的交集，如果存在相同元素，保留相同的元素，如果不存在相同元素，清空原来容器中的内容

Collection接口下的AbstractCollection抽象类中重写了toString()方法，而AbstractCollection实现了List接口，List的实现类继承了AbstractCollection,所以List的三个子类不需要重写toString方法。

#### 2. List接口

有序容器，允许元素重复。可以通过下标索引元素，允许添加s1,s2元素且s1，s2元素满足s1.equals(s2).

List中的方法：

void add(int index, E element)：在索引位置插入新的元素，其后元素依次后移。

E set(int index, E element)：修改索引位置元素为传入元素，并返回修改之前的元素

E get(int index)：返回指定索引位置的元素

int indexOf(Object o)：返回指定元素在集合中出现的第一个位置，如果找不到该元素，返回-1

int lastIndexOf(Object o)：返回指定元素在集合中出现的最后一个位置，如果找不到该元素，返回-1

List有三个实现类：

ArrayList:底层使用数组实现，查询效率高，增删效率低，线程不安全。

ArrayList的数组扩容机制:定义一个更长的数组（默认长度为10，默认扩容后长度为原来的1.5倍），将原来的数组拷贝过来。

ArrayList提供的两个list转array的方法：

list.toArray();将list直接转为Object[] 数组；注意，如果list中的元素不是Object，可能出现java.lang.ClassCastException，这是因为不能将Object[] 转化为String[]

list.toArray(T[] a); 将list转化为你所需要类型的数组，当然我们用的时候会转化为与list内容相同的类型。

LinkedList：底层使用双向链表，查询效率低，增删效率高，线程不安全。

Vector：底层使用数组，是线程安全的

#### 3.Map接口

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

存储键值对的类，不能包含重复的键，一个键最多只能对应一个值。

Map中的方法：

int size()：有多少个键值对

boolean isEmpty()：是否是空的map

boolean containsKey(Object key)：是否包含指定的键

boolean containsValue(Object value)：是否包含指定的值

V get(Object key)：获取指定的键对应的值

V put(K key, V value)：保存指定键值对

Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for the key, the old value is replaced by the specified value. (A map m is said to contain a mapping for a key k if and only if m.containsKey(k) would return true.)

如果map中已经包含了传入的键(判断是否重复通过equals方法判断)，则该方法会用传入的值覆盖原来的值，返回原来的值。

V remove(Object key)：移除指定键的键值对，返回该键对应的值，找不到该键值对，返回null

void putAll(Map<? extends K,? extends V> m)：将指定map中的值放入原map中

void clear()：清除map中所有的键值对

Set<K> keySet()：返回map中所有键的set集

Collection<V> values()：返回map中所有值的集合

Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa. If the map is modified while an iteration over the collection is in progress (except through the iterator's own remove operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Collection.remove, removeAll, retainAll and clear operations. It does not support the add or addAll operations.

返回此映射中包含的值的集合视图。集合由映射支持，因此对映射的更改将反映在集合中，反之亦然。如果在对集合进行迭代时修改映射(除了通过迭代器自己的删除操作)，迭代的结果是未定义的。集合支持元素删除，元素删除通过迭代器从映射中删除对应的映射。

Set<Map.Entry<K,V>> entrySet()：Returns a Set view of the mappings contained in this map.

Map下的实现类：

**java.util.HashMap**

底层实现采用哈希表（哈希表的基本结构是数组+单项链表），线程不安全

HashMap中的static class Node<K,V> implements Map.Entry<K,V>结构如下(本质是一个单项链表)：



再将Entry对象以数组的形式存放Entry[]



存储过程通过哈希计算。

HashMap源码重点理解：

1.HashMap存储方式：数组加单向链表

2.HahsMap获取键值对方式

3.HashMap遍历（实际上前两个就是HashMap的遍历）

**java.util.TreeMap<K,V>**

底层使用红黑二叉树实现，如果需要做排序，使用TreeMap很方便，TreeMap默认对键值对按键的大小（例如键为Integer类型，按数字升序排序）进行了排序。

TreeMap中的键对象实现了Comparable接口，使用时如果向按照自定义的方式进行排序，让key对应的类型实现Comparable接口中的compare方法。

**Java.util.HashTable<K,V>**

关于hashmap的问题;

HashMap在JDK1.8及以后的版本中引入了红黑树结构，若桶中链表元素个数大于等于8时，链表转换成树结构；若桶中链表元素个数小于等于6时，树结构还原成链表。因为红黑树的平均查找长度是log(n)，长度为8的时候，平均查找长度为3，如果继续使用链表，平均查找长度为8/2=4，这才有转换为树的必要。链表长度如果是小于等于6，6/2=3，虽然速度也很快的，但是转化为树结构和生成树的时间并不会太短。

还有选择6和8，中间有个差值7可以有效防止链表和树频繁转换。假设一下，如果设计成链表个数超过8则链表转换成树结构，链表个数小于8则树结构转换成链表，如果一个HashMap不停的插入、删除元素，链表个数在8左右徘徊，就会频繁的发生树转链表、链表转树，效率会很低。

#### 4. java.util.Set

Set无顺序不可重复。要查找只能遍历，只能放一个null元素。

实现类：

HashSet：采用Hash算法实现，底层使用HashMap实现，本质上就是一个简化的HashMap，通过key来存储set的元素。

TreeSet：底层使用TreeMap实现，内部维护一个简易的TreeMap，使用key来存储元素。TreeSet可自动排序（使用内部的TreeMap的排序功能）

### 4.容器遍历

1.for循环或增强型for循环

2.迭代器Iterator

这两种方式所有的容器都适用（不进行删除元素的情况下）。

Collection接口下的所有的容器都可以使用Iterator来进行迭代遍历，使用iterator()方法获取Iterator对象进行遍历。Map容器需要使用如下两种方式：

//map遍历方法1

Set<Map.Entry<Integer,String>> set = map.entrySet();

for(Iterator<Map.Entry<Integer,String>> it = set.iterator();it.hasNext();){

Map.Entry<Integer,String> temp = it.next();

System.out.println(temp.getKey()+":"+temp.getValue());

}

//map遍历方法2

Set<Integer> set1 = map.keySet();

for(Iterator<Integer> iterator = set1.iterator();iterator.hasNext();){

int key = iterator.next();

System.out.println(key+":"+map.get(key));

}

Iterator的方法：

boolean hasNext():Returns true if the iteration has more elements.判断容器中是否还存在下一个元素

E next()：获取下个元素

void remove()：Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to [next()](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/util/Iterator.html#next()).

从基础集合中移除此迭代器返回的最后一个元素(可选操作)。每次调用next()只能调用此方法一次。

注意：如果需要在遍历集合时执行删除集合元素操作，必须使用iterator进行遍历，使用iterator的remove方法去删除元素，这样才能保证结果的准确性。

看看内部类Itr的remove()方法，在删除元素后，有这么一句expectedModCount = modCount，同步修改expectedModCount 的值。所以，如果需要在使用迭代器迭代时，删除元素，可以使用迭代器提供的remove方法。对于add操作，则在整个迭代器迭代过程中是不允许的。 其他集合(Map/Set)使用迭代器迭代也是一样。

当使用 fail-fast iterator 对 Collection 或 Map 进行迭代操作过程中尝试直接修改 Collection / Map 的内容时，即使是在单线程下运行， java.util.ConcurrentModificationException 异常也将被抛出。

Iterator 被创建之后会建立一个指向原来对象的单链索引表，当原来的对象数量发生变化时，这个索引表的内容不会同步改变，所以当索引指针往后移动的时候就找不到要迭代的对象，所以按照 fail-fast 原则 Iterator 会马上抛出 java.util.ConcurrentModificationException 异常。

所以 Iterator 在工作的时候是不允许被迭代的对象被改变的。

但你可以使用 Iterator 本身的方法 remove() 来删除对象， Iterator.remove() 方法会在删除当前迭代对象的同时维护索引的一致性。

参考：<https://www.cnblogs.com/goody9807/p/6432904.html>

### 5. java.util.Collections工具类

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

常用方法：

public static void shuffle(List<?> list)：随机排序，洗牌

public static void reverse(List<?> list)：逆序操作

public static <T extends Comparable<? super T>> void sort(List<T> list)：默认排序操作

public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)：二分法查找

### java.util.concurrent包下线程安全的容器：

1. CopyOnWriteArrayList:线程安全的list

## 关于多态处理

子类B继承父类A，声明使用父类A而创建使用子类B这种方式成为多态。

关于父子类中的方法存在以下三种情况：

1.父类中有而子类中没有的方法，使用多态时会直接调用父类的方法

2.父类中有而子类中对父类的方法进行了重写，这个时候调用的是子类的方法

3.父类中没有而子类中有的方法，这是调用会报错。

18.boolean类型占几个字节问题

官方文档描述：

boolean: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

布尔类型：布尔数据类型只有两个可能的值：真和假。使用此数据类型为跟踪真/假条件的简单标记。这种数据类型就表示这一点信息，但是它的“大小”并不是精确定义的。

可以看出，boolean类型没有给出精确的定义，《Java虚拟机规范》给出了4个字节，和boolean数组1个字节的定义，具体还要看虚拟机实现是否按照规范来，所以1个字节、4个字节都是有可能的。这其实是运算效率和存储空间之间的博弈，两者都非常的重要。

参考：<https://www.jianshu.com/p/2f663dc820d0>

## Java关键字总结

native：原生函数，不需要java自己实现（由c/c++实现），java直接调用。

transient：如果用transient声明一个[实例变量](https://baike.baidu.com/item/%E5%AE%9E%E4%BE%8B%E5%8F%98%E9%87%8F/3386159)，当对象存储时，它的值不需要维持。transient关键字修饰的变量不参与序列化过程。

Instanceof:二元操作符，判断左边对象是否为右边实例，返回boolean类型

new 所做的操作：1.开辟空间 2.初始化对象信息 3.返回对象的引用

## 运算符优先级问题

总体优先级顺序



网友提供的口诀：

单目乘除为关系，逻辑三目后赋值。

单目：单目运算符+ –(负数) ++ -- 等   
乘除：算数单目运算符\* / % + -   
为：位移单目运算符<< >>   
关系：关系单目运算符> < >= <= == !=   
逻辑：逻辑单目运算符&& || & | ^   
三目：三目单目运算符A > B ? X : Y   
后：无意义，仅仅为了凑字数   
赋值：赋值=

参考：<https://www.cnblogs.com/zjfjava/p/5996666.html>

## 成员变量与局部变量

1、在类中的位置不同

成员变量：在类中方法外面

局部变量：在方法或者代码块中，或者方法的声明上（即在参数列表中）

2、在内存中的位置不同，可以看看Java程序内存的简单分析

成员变量：在堆中（方法区中的静态区）

局部变量：在栈中

3、生命周期不同

成员变量：随着对象的创建而存在，随着对象的消失而消失

局部变量：随着方法的调用或者代码块的执行而存在，随着方法的调用完毕或者代码块的执行完毕而消失

4、初始值

成员变量：有默认初始值

局部变量：没有默认初始值，使用之前需要赋值，否则编译器会报错（The local variable xxx may not have been initialized）

## Lambda表达式（函数式编程）

几种内部类：

1.普通内部类：类的成员变量，类加载时会被加载

2.静态内部类：类的成员变量，静态内部类，使用时才会加载

3.局部（方法）内部类：定义在方法体中的内部类，只在方法体中有效

4.匿名内部内：接口或抽象类作为参数传递给方法调用，必须借用接口或父类

以上几种方式都是为了简化代码而设计的，JDK1.8在匿名内的基础上由简化了一步，通过lambda表达式来实现。

如Runnable的Run方法使用lambda表达式来推导。

new Thread(()->{

for (int i = 0; i < 30; i++) {

System.out.println("in thread4............");

}

});

注意：接口或抽象父类中只能有一个待实现的方法才能使用lambda表达式，且lambda推导必须存在类型。

当待实现方法只有一个参数时，()可以省略，类型也可以省略：

Lambda lambda3 = msg -> {

System.out.println("without () in lambda");

};

## java网络编程

### 几个重要的概念

1.IP：定位电脑 对应类：[java.net.InetAddress](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/net/InetAddress.html)

2.port（端口）:定位某个服务 windows下查看端口占用情况的命令 netstat -ano

Java 中对应端口的类 java.net.InetSocketAddress

3.URL：定位某个资源 包含四个部分： 协议、域名或计算机名，端口号、请求资源，对应java 的类 java.net.URL

4.协议：定义通信的方式（TCP/UDP）通信标准

TCP: （Transmission Control Protocol 传输控制协议）面向连接的（三次握手）、可靠的、基于字节流的点到点的通信，占用资源多、效率低

UDP：user datagram protocol,非面向连接、传输不可靠、简单、开销小

5.socket套接字：在应用层与传输层使用套接字来分离。

### 爬虫原理

操作步骤：

1.获取url

2.下载资源

3.分析资源

4.处理数据

### Socket编程

#### UDP编程

使用到的类：

1.套接字：java.net.DatagramSocket This class represents a socket for sending and receiving datagram packets.

2.数据包：java.net.DatagramPacket This class represents a datagram packet.

操作步骤：

接收端（server）：

1.使用DatagramSocket创建接收端，指定端口号

2.准备封装程DatagramPacket包裹

3.阻塞式接收receive(DatagramPacket p) throws IOException

4.分析数据 getData（） 、getLength（）

5.释放资源

发送端（client）:

1.使用DatagramSocket创建发送端，指定端口号

2.准备数据转成字节数组

3.封装成DatagramPacket包裹，需要指定目的地（ip+port）

4.发送包裹send(DatapramPacket p)

5.释放资源

TCP/UDP实现聊天功能

#### TCP编程

使用到的类：

java.net.ServerSocket 相当于服务端。 This class implements server sockets.

java.net.Socket This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.，相当于客户端，是两个机器进行点到点通信的工具类。

通信步骤：

服务端：

* + 1. 创建ServerSocket对象作为服务器
    2. 执行accept()方法，监听客户端的请求（此处为阻塞式等待连接，返回一个Socket对象）
  1. 接收并处理请求信息，输入流

1. 将请求信息返回给客户端，输出流
2. 关闭流和socket对象

客户端：

* 1. 创建Socket对象作为客户端，指定服务器地址和端口
  2. 向服务端发送连接请求

1. 接收服务端返回结果并处理
2. 释放资源，关闭流和socket对象

## Java解析xml

Xml解析的两种方式：

1. SAX解析（simple API for XML流解析）（JDK中有api用的较多），使用JDK工具包org.xml.sax,解析步骤：
2. 获取解析工厂SAXparserFactory
3. 从解析工厂中获得解析器
4. 加载文档的Document注册处理器
5. 编写处理器
6. DOM解析（将XMl解析为DOM树）

## Java ClassLoader

## Java注解（annotation）

### 使用注解的原因：

使用Annotation之前(甚至在使用之后)，XML被广泛的应用于描述元数据。不知何时开始一些应用开发人员和架构师发现XML的维护越来越糟糕了。他们希望使用一些和代码紧耦合的东西，而不是像XML那样和代码是松耦合的(在某些情况下甚至是完全分离的)代码描述。

### 注解与xml配置的优缺点

假如你想为应用设置很多的常量或参数，这种情况下，XML是一个很好的选择，因为它不会同特定的代码相连。如果你想把某个方法声明为服务，那么使用Annotation会更好一些，因为这种情况下需要注解和方法紧密耦合起来，开发人员也必须认识到这点。

Annotation定义了一种标准的描述元数据的方式。在这之前，开发人员通常使用他们自己的方式定义元数据，例如，使用标记interfaces，注释，transient关键字等等，每个程序员按照自己的方式定义元数据，而不像Annotation这种标准的方式。

从JDK5开始,Java增加对元数据的支持，也就是注解，注解与注释是有一定区别的，可以把注解理解为代码里的特殊标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理。通过注解开发人员可以在不改变原有代码和逻辑的情况下在源代码中嵌入补充信息。

注解就是源代码的元数据。注解可以对程序进行说明，可以被相关程序读取到。

### 注解的作用

1. 生成文档。这是最常见的，也是java 最早提供的注解。常用的有@see @param @return 等
2. 跟踪代码依赖性，实现替代配置文件功能。比较常见的是spring 2.5 开始的基于注解配置。作用就是减少配置。现在的框架基本都使用了这种配置来减少配置文件的数量。也是
3. 在编译时进行格式检查。如@Override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

### 注解使用

 java.lang.annotation提供了四种元注解，专门注解其他的注解：

1. @Documented –注解是否将包含在JavaDoc中，表示是否将注解信息添加在java文档中。
2. @Retention –什么时候使用该注解，定义该注解的生命周期，有以下几个范围;
3. RetentionPolicy.SOURCE – 在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。@Override, @SuppressWarnings都属于这类注解。
4. RetentionPolicy.CLASS – 在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式。
5. RetentionPolicy.RUNTIME– 始终不会丢弃，运行期也保留该注解，因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式
6. @Target –注解用于什么地方表示。如果不明确指出，该注解可以放在任何地方（可以是package、class和interface（TYPE）、field、method、constructor等）。以下是一些可用的参数。需要说明的是：属性的注解是兼容的，如果你想给7个属性都添加注解，仅仅排除一个属性，那么你需要在定义target包含所有的属性，target value有如下几个范围：

ElementType.TYPE:用于描述类、接口或enum声明

ElementType.FIELD:用于描述实例变量

ElementType.METHOD 方法

ElementType.PARAMETER 参数

ElementType.CONSTRUCTOR 构造器

ElementType.LOCAL\_VARIABLE 局部变量

ElementType.ANNOTATION\_TYPE 另一个注释

ElementType.PACKAGE 用于记录java文件的package信息

1. @Inherited – 是否允许子类继承该注解

JDK内置注解：

@Override 重写注解

@Deprecated 不鼓励使用这样的元素

@SuppressWarnings 抑制编译时的警告信息

注解使用：

* + - 创建注解：

@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.SOURCE)  
public @interface Override {  
}

@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL\_VARIABLE, MODULE})

@Retention(RetentionPolicy.SOURCE)

public @interface SuppressWarnings {

String[] value(); //注意这里是参数名和参数类型，并不是方法，一般可以加上default “”表示可以允许不赋值

}

* + - 在类中使用注解
    - 使用反射解析注解

Class clz = Class.forName("com.steven.annotationDemo.Person7");

//获取类的所有注解

Annotation[]annotations = clz.getAnnotations();

AnnotationDemo3 annotationDemo3 = (AnnotationDemo3) clz.getAnnotation(AnnotationDemo3.class);

System.out.println(annotationDemo3);

for (Annotation annotation:annotations

) {

System.out.println(annotation);

}

//获取属性

Field field = clz.getDeclaredField("id");

//获取类的指定注解（属性、方法）

AnnotationFieldDemo4 annotationFieldDemo4 = field.getAnnotation(AnnotationFieldDemo4.class);

System.out.println(annotationFieldDemo4);

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (NoSuchFieldException e) {

e.printStackTrace();

}

## Java反射reflection（java动态性1）

动态：程序运行时可以改变程序变量或结构，如javascript可以以改变变量类型。

Java具有一定的动态性（通过反射、动态编译、动态操作字节码实现）

### 简单理解与实现

通过java.lang.Class类将java类中的各种结构（方法属性构造器、类名）映射程一个个java对象。利用反射可以对类进行解剖，反射式框架设计的灵魂。

要用到的类：

java.lang.Class<T>

Instances of the class Class represent classes and interfaces in a running Java application.

Class类的实例代表当前再运行状态下的java类和接口。类的元信息。

获取Class对象的三种方式：

1. 通过对象的实例获取Class对象（new Object().class）
2. 通过类获取Class对象（Object.class）
3. 通过Class的forName方法获取Class对象（Class.forName(com.steven.Reflection.ReflectionDemo)）

注意：一个类只对应一个Class对象

推荐使用第三种方式。

获取到class对象后可以做一下工作：

1. 创建对象：

有以下两种方式（注意必须存在无参构造器）：

TestModel testModel = (TestModel) clz.newInstance();//JDK9不推荐使用

TestModel testModel1 = clz.getConstructors().newInstance();//JDK9中使用

传参用法：

ReflectionTest02 reflectionTest02 = (ReflectionTest02) clz.getConstructor(int.class,String.class,String.class). newInstance(123,"steven","male");

2．使用反射API获取类的信息（类名、属性、方法、构造器）

Java.lang.Class 方法：

public String getName() 获取类全路径名

public String getSimpleName() 只获取类名

public Field[] getFields() throws SecurityException：获取public类型的属性

public Class<?>[] getDeclaredClasses() throws SecurityException：获得所有属性

public Method[] getMethods() throws SecurityException：获取所有public方法

public Method getDeclaredMethod(String name, Class<?>... parameterTypes) throws NoSuchMethodException, SecurityException ：传递参数类型获取方法（根据参数类型区分重载方法，构造器类似）

public Constructor<?>[] getDeclaredConstructors() throws SecurityException：获取所有构造方法

* 1. 反射调用普通方法：  
     先获取Method类

Method method1 = clz.getDeclaredMethod("setId",int.class);

然后调用invoke方法

method.invoke(reflectionTest02,2354);

* 1. 操作属性：

field.set(reflectionTest02,123123);

System.out.println(method2.invoke(reflectionTest02));

注意：需要时public属性，如果要操作私有属性，需要调用field.setAccessible(true);再进行操作。

### 反射性能问题

使用反射会影响程序新能，可以通过setAccessible提高新能，设为true禁止安全检查，能提高性能。

效率差异，不加setAccessible的反射方法调用与普通类调用方法慢了30倍的效率，与加了setAccessible(true)的反射方法调用差了4倍效率.

### 反射操作泛型

泛型使用类擦除技术，泛型仅仅时给编译器javac使用的，确保数据的安全性，和免去强制类型转换的麻烦，但编译一旦完成，所有和泛型有关的类型全部擦除。

为了通过反射操作这些类型，Java增加了ParameterizedType、GenericArrayType,TypeVariable,WildcardType来代表不能被归一到Class类型但又和原始类型齐名的类型。

java.lang.reflect.Interface GenericArrayType

java.lang.reflect.ParameterizedType 一种参数化的类型

java.lang.reflect. TypeVariable 各种类型变量的公共父接口

java.lang.reflect. WildcardType 通配符类型表达式

## 动态编译（Java动态性2）

JDK1.6引入的动态编译，动态编译的两种实现方式：

1.通过Runtime调用javac启动新的进程去操作

Runime run = Runtime.getRuntime();

Process process = run.extc(“javac -cp d:/java/ HelloWorld.java”);

1. 通过javaCompiler动态编译

int result = compiler.run(null,null,null,path);

## 脚本引擎执行JavaScript代码Rhino引擎（Java动态性3）

JDK1.6新增加功能

ScriptEngineManager sem = new ScriptEngineManager();

ScriptEngine engine = sem.getEngineByName("javascript");

engine.put("msg","steven");

System.out.println(engine.get("msg"));

## Java字节码操作（Java动态性4）

字节码操作的功能：

1.动态生成新的类

2.动态改变某个类的结构（如新增删除修改方法或属性）

字节码操作库：

BCEL

ASM

CGLIB

Javassist

很多开源框架都用到了这几个字节码操作库，例如AOP会使用javassist来实现。

Javassist的使用

API(与反射包中的API类似):

CtClass

CtMethod

CtField

简单使用：创建类、运行类方法、操作类属性（getter、setter方法）、操作注解（都与反射类似）

## 正则表达式

## Hashcode的作用

（1）HashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，HashCode经常用于确定对象的存储地址；

（2）如果两个对象相同， equals方法一定返回true，并且这两个对象的HashCode一定相同；

（3）两个对象的HashCode相同，并不一定表示两个对象就相同，即equals()不一定为true，只能够说明这两个对象在一个散列存储结构中。

（4）如果对象的equals方法被重写，那么对象的HashCode也尽量重写。

# Spring零碎知识

## Spring 的启动过程

1.对于一个web应用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境。

2.在web.xml中会提供有contextLoaderListener。在web容器启动时，会触发容器初始化事件，此时contextLoaderListener会监听到这个事件，其contextInitialized方法会被调用，在这个方法中，spring会初始化一个启动上下文，这个上下文被称为根上下文，即WebApplicationContext，这是一个接口类，确切的说，其实际的实现类是XmlWebApplicationContext。这个就是spring的IoC容器，其对应的Bean定义的配置由web.xml中的context-param标签指定。在这个IoC容器初始化完毕后，spring以WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE为属性Key，将其存储到ServletContext中，便于获取。

1. contextLoaderListener监听器初始化完毕后，开始初始化web.xml中配置的Servlet，这个servlet可以配置多个，以最常见的DispatcherServlet为例，这个servlet实际上是一个标准的前端控制器，用以转发、匹配、处理每个servlet请求。DispatcherServlet上下文在初始化的时候会建立自己的IoC上下文，用以持有spring mvc相关的bean。在建立DispatcherServlet自己的IoC上下文时，会利用WebApplicationContext.ROOT*WEB*APPLICATION*CONTEXT*ATTRIBUTE先从ServletContext中获取之前的根上下文(即WebApplicationContext)作为自己上下文的parent上下文。有了这个parent上下文之后，再初始化自己持有的上下文。这个DispatcherServlet初始化自己上下文的工作在其initStrategies方法中可以看到，大概的工作就是初始化处理器映射、视图解析等。这个servlet自己持有的上下文默认实现类也是mlWebApplicationContext。初始化完毕后，spring以与servlet的名字相关(此处不是简单的以servlet名为Key，而是通过一些转换，具体可自行查看源码)的属性为属性Key，也将其存到ServletContext中，以便后续使用。这样每个servlet就持有自己的上下文，即拥有自己独立的bean空间，同时各个servlet共享相同的bean，即根上下文(第2步中初始化的上下文)定义的那些bean。

全过程：

Web容器上下文ServletContext—>contextLoaderListener的上下文WebApplicationContext—>DispatcherServlet的上下文（拿到WebApplicationContext作为自己的上下文，初始化自己的上下文mlWebApplicationContext）

## springMVC的启动过程

### 两个上下文

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。

## 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。

## 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

## 几种视图模板技术

Jsp

Freemaker

Thymeleaf

Velocity

## 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\*"/>

## web flow

## 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

## token/refresh token的概念

## JSON Web Token (JWT)

## spring mvc接收json参数

### 11.1.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

### 11.2．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>

## Spring mvc接收数组、集合

* 1. 接收checkbox表单元素传过来的参数（基础类型数据）：

<form action="param2" method="post">

<input type="checkbox" value="1" name="param2"/>

<input type="checkbox" value="2" name="param2"/>

<input type="checkbox" value="3" name="param2"/>

<button type="submit">submit</button>

</form>

后端写法

//接收checkbox类型表单

@ResponseBody

@RequestMapping(value = "/param2",method = RequestMethod.POST)

public String getParam2(@RequestParam(value = "param2",required = false) List<Long> param2){

for (Long value : param2) {

System.out.println(value);

}

return "success2";

}

* 1. 接收json数组参数（基础类型数据）

var param = [1,2,3,4];

$.post("http://localhost:8080/demos/testParam/params",{param:param},function(data){

console.log(data)

});

后端写法：

//接收数组

@ResponseBody

@RequestMapping(value = "/params",method = RequestMethod.POST)

public String getParam(@RequestParam(value="param[]", required = false) List<Long> param){

for (int i = 0; i < param.size();i++){

System.out.println(param.get(i));

}

return "success!";

}

3.接收map

var param3 = {'a':123,'b':234};

$.ajax({

url: "params3",

type: "post",

contentType : 'application/json;charset=utf-8',

data: JSON.stringify(param3),

success: function (json) {

console.log(json)

}

});

后端写法：

//接收map

@ResponseBody

@RequestMapping(value = "/params3",method = RequestMethod.POST)

public String getParam3(@RequestBody Map<String,String> map){

for (Map.Entry<String,String> entry:map.entrySet()){

System.out.print("得到键为：==="+entry.getKey());

System.out.println("得到值为：==="+entry.getValue());

}

return "success3!";

}

1. 接收json对象数组（非基础类型数据）

var param4 = [

{"name":"steven","gender":"male"},

{"name":"lu","gender":"female"}

];

$.ajax({

url: "params4",

type: "post",

contentType : 'application/json;charset=utf-8',

data: JSON.stringify(param4),

success: function (json) {

console.log(json)

}

});

后端写法：

//接收数组

@ResponseBody

@RequestMapping(value = "/params4",method = RequestMethod.POST)

public String getParam4(@RequestBody List<TestObj> param4){

for (int i = 0; i < param4.size();i++){

System.out.println(param4.get(i));

}

return "success!";

}

## 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

事务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

### ****Spring 事务****

事务实现的原理：动态代理。（注意：JDK动态代理必须存在接口，CGlib则没有要求）

事务失效原因（只针对声明式事务）：

1.

#### 1.接口

**TransactionDefinition（事务规则：**设置事务的一些属性，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

## DI实现原理

## AOP实现原理

AOP（Aspect Oriented Programming）的含义

软件开发中，分布于应用中的多处功能被称为横切面关注点，这些横切面关注点从概念上是与应用逻辑相分离的，将横切面关注点与业务逻辑相分离正是面向切面编程(AOP编程)所需要解决的。

作用：面向切面编程（AOP）就是对软件系统不同关注点的分离，开发者通过拦截方法调用并在方法调用前后添加辅助代码。

几个概念：

通知（Adivce）

切面的工作目标，定义切面是何时何处使用。有Before、After、After-returning、After-throwing、Around几种

连接点（Joinpoint）

应用在执行过程中能插入切面的一个点，简单来说是程序执行的某个位置。

切点（Pointcut）

匹配通知所织入的一个或多个连接点（spring只支持方法切点）。

切面（Aspect）

切面是通知和切点的结合。通知和切点定义了切面的全部内容。

引入（Introduction）

允许向现有类添加新的方法或属性。

织入（weaving）

将切面应用到目标对象来创建新的代理对象的过程。切面在指定的连接点被织入到目标对象中。

JDK动态代理+cglib动态代理，前者时使用反射，后者直接操作动态操作字节码。

## Spring Annotation大全

### spring @component的作用

1、@controller 控制器（注入服务）

2、@service 服务（注入dao）

3、@repository dao（实现dao访问）

4、@component （把普通pojo实例化到spring容器中，相当于配置文件中的<bean id="" class=""/>）

  @Component,@Service,@Controller,@Repository注解的类，并把这些类纳入进spring容器中管理。   
下面写这个是引入component的扫描组件   
<context:component-scan base-package=”com.mmnc”>      
  
其中base-package为需要扫描的包（含所有子包）   
       1、@Service用于标注业务层组件   
       2、@Controller用于标注控制层组件(如struts中的action)   
       3、@Repository用于标注数据访问组件，即DAO组件.   
       4、@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。      
 @Service

public class UserServiceImpl implements UserService { }   
@Repository

public class UserDaoImpl implements UserDao { }

getBean的默认名称是类名（头字母小写），如果想自定义，可以@Service(“\*\*\*”)               这样来指定，这种bean默认是单例的，如果想改变，可以使用@Service(“beanName”)   
           @Scope(“prototype”)来改变。可以使用以下方式指定初始化方法和销毁方法（方法名任意）： @PostConstruct public void init() { }

### @Autowired与@Resource的区别

1、@Autowired与@Resource都可以用来装配bean. 都可以写在字段上,或写在setter方法上。   
2、@Autowired默认按类型装配（这个注解属于[spring](http://lib.csdn.net/base/javaee)），默认情况下必须要求依赖对象必须存在，如果要允许null 值，可以设置它的required属性为false，如：@Autowired(required=false) ，如果我们想使用名称装配可以结合@Qualifier注解进行使用，如下：

Java代码

1. @Autowired() @Qualifier("baseDao")
2. **private BaseDao baseDao;**

 3、@Resource（这个注解属于J2EE的），默认安照名称进行装配，名称可以通过name属性进行指定，   
如果没有指定name属性，当注解写在字段上时，默认取字段名进行按照名称查找，如果注解写在setter方法上默认取属性名进行装配。 当找不到与名称匹配的bean时才按照类型进行装配。但是需要注意的是，如果name属性一旦指定，就只会按照名称进行装配。

Java代码

1. @Resource(name="baseDao")
2. **private BaseDao baseDao;**

在[Java](http://lib.csdn.net/base/javase)代码中可以使用@Autowire或者@Resource注解方式进行装配，这两个注解的区别是：  
@Autowire 默认按照类型装配，默认情况下它要求依赖对象必须存在如果允许为null，可以设置它required属性为false，如果我们想使用按照名称装配，可 以结合@Qualifier注解一起使用;

@Resource默认按照名称装配，当找不到与名称匹配的bean才会按照类型装配，可以通过name属性指定，如果没有指定name属 性，当注解标注在字段上，即默认取字段的名称作为bean名称寻找依赖对象，当注解标注在属性的setter方法上，即默认取属性名作为bean名称寻找 依赖对象.

**各种注解方式**

1.@Autowired注解(不推荐使用，建议使用@Resource)

@Autowired可以对成员变量、方法和构造函数进行标注，来完成自动装配的工作。@Autowired的标注位置不同，它们都会在Spring在初始化这个bean时，自动装配这个属性。要使@Autowired能够工作，还需要在配置文件中加入以下

Xml代码

1. <bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />

2. @Qualifier注解

@Autowired是根据类型进行自动装配的。例如，如果当Spring上下文中存在不止一个UserDao类型的bean时，就会抛出BeanCreationException异常;如果Spring上下文中不存在UserDao类型的bean，也会抛出BeanCreationException异常。我们可以使用@Qualifier配合@Autowired来解决这些问题。如下：

1). 可能存在多个UserDao实例

Java代码

1. @Autowired
2. @Qualifier("userServiceImpl")
3. public IUserService userService;

或者

Java代码

1. @Autowired
2. public void setUserDao(@Qualifier("userDao") UserDao userDao) {
3. this.userDao = userDao;
4. }

这样，Spring会找到id为userServiceImpl和userDao的bean进行装配。

2). 可能不存在UserDao实例

Java代码

1. @Autowired(required = false)
2. public IUserService userService;

3. @Resource注解

JSR-250标准注解，推荐使用它来代替Spring专有的@Autowired注解。@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按byName自动注入罢了。@Resource有两个属性是比较重要的，分别是name和type，Spring将 @Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。要使@Autowired能够工作，还需要在配置文件中加入以下:

Xml代码

1. <bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />

@Resource装配顺序：

a.如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常

b.如果指定了name，则从上下文中查找名称(id)匹配的bean进行装配，找不到则抛出异常

c.如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，找不到或者找到多个，都会抛出异常

d.如果既没有指定name，又没有指定type，则自动按照byName方式进行装配(见2);如果没有匹配，则回退为一个原始类型(UserDao)进行匹配，如果匹配则自动装配;

4. @PostConstruct(JSR-250)注解

在方法上加上注解@PostConstruct，这个方法就会在Bean初始化之后被Spring容器执行(注：Bean初始化包括，实例化Bean，并装配Bean的属性(依赖注入))。它的一个典型的应用场景是，当你需要往Bean里注入一个其父类中定义的属性，而你又无法复写父类的属性或属性的setter方法时，如：

5. @PreDestroy(JSR-250)注解

在方法上加上注解@PreDestroy，这个方法就会在Bean初始化之后被Spring容器执行。其用法同@PostConstruct。和@PostConstruct 区别在于：@PostConstruct注释的方法将在类实例化后调用，而标注了 @PreDestroy 的方法将在类销毁之前调用。

6. @Component注解 (不推荐使用)

只需要在对应的类上加上一个@Component注解，就将该类定义为一个Bean了。Spring还提供了更加细化的注解形式：@Repository、@Service、@Controller，它们分别对应存储层Bean，业务层Bean，和展示层Bean。目前版本(2.5)中，这些注解与@Component的语义是一样的，完全通用，在Spring以后的版本中可能会给它们追加更多的语义。所以，我们推荐使用@Repository、@Service、@Controller来替代@Component。

7.@Scope注解

在使用XML定义Bean时，我们可能还需要通过bean的scope属性来定义一个Bean的作用范围，我们同样可以通过@Scope注解来完成这项工作：

Java代码

1. @Scope("session")
2. @Component()
3. public class UserSessionBean implements Serializable{
4. ... ...
5. }

二、配置启用注解(注意以下配置需要使用spring2.5的头文件，在spring3.0中不适用)

1.使用简化配置

Spring2.1添加了一个新的context的Schema命名空间，该命名空间对注释驱动、属性文件引入、加载期织入等功能提供了便捷的配置。我们知道注释本身是不会做任何事情的，它仅提供元数据信息。要使元数据信息真正起作用，必须让负责处理这些元数据的处理器工作起来。

AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor就是处理这些注释元数据的处理器。但是直接在Spring配置文件中定义这些Bean显得比较笨拙。Spring为我们提供了一种方便的注册这些BeanPostProcessor的方式，这就是，以下是spring的配置。

Xml代码

1. <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
2. xsi:schemaLocation="http://www.springframework.org/schema/beans
3. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
4. http://www.springframework.org/schema/context
5. http://www.springframework.org/schema/context/spring-context-2.5.xsd">
6. <context:annotation-config />
7. beans>

将隐式地向Spring容器注册了

AutowiredAnnotationBeanPostProcessor 、

CommonAnnotationBeanPostProcessor 、

PersistenceAnnotationBeanPostProcessor

RequiredAnnotationBeanPostProcessor

这4个BeanPostProcessor。

2.使用让Bean定义注解工作起来

Xml代码

1. <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
2. xsi:schemaLocation="http://www.springframework.org/schema/beans
3. http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
4. http://www.springframework.org/schema/context
5. http://www.springframework.org/schema/context/spring-context-2.5.xsd">
6. <context:component-scan base-package="com.kedacom.ksoa" />
7. beans>

这里，所有通过元素定义Bean的配置内容已经被移除，仅需要添加一行配置就解决所有问题了——Spring XML配置文件得到了极致的简化(当然配置元数据还是需要的，只不过以注释形式存在罢了)。的base-package属性指定了需要扫描的类包，类包及其递归子包中所有的类都会被处理。

还允许定义过滤器将基包下的某些类纳入或排除。Spring支持以下4种类型的过滤方式：

过滤器类型 | 表达式范例 | 说明

注解 | org.example.SomeAnnotation | 将所有使用SomeAnnotation注解的类过滤出来

类名指定 | org.example.SomeClass | 过滤指定的类

正则表达式 | com\.kedacom\.spring\.annotation\.web\..\* | 通过正则表达式过滤一些类

AspectJ表达式 | org.example..\*Service+ | 通过AspectJ表达式过滤一些类

以正则表达式为例，我列举一个应用实例：

Xml代码

1. <context:component-scan base-package="com.casheen.spring.annotation">
2. <context:exclude-filter type="regex" expression="com\.casheen\.spring\.annotation\.web\..\*" />
3. context:component-scan>

　值得注意的是配置项不但启用了对类包进行扫描以实施注释驱动Bean定义的功能，同时还启用了注释驱动自动注入的功能（即还隐式地在内部注册了AutowiredAnnotationBeanPostProcessor和CommonAnnotationBeanPostProcessor），因此当使用后，就可以将移除了。

3.   
是不支持spring的@Transcation和EJB的Spring's @Transactional or EJB3's @TransactionAttribute annotation。用此配置可以达到目的。

4. 使用@Scope来定义Bean的作用范围

在使用XML定义Bean时，我们可能还需要通过bean的scope属性来定义一个Bean的作用范围，我们同样可以通过@Scope注解来完成这项工作：

# Web及HTTP理解

## 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

1. 基于请求和响应，客户端请求，服务器端响应
2. 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
3. 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
* DELETE:删除服务器上的资源，这个方法不允许携带请求体（一般通过URL路径上传id来删除指定资源）
* 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。

## 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：

1. HTTP 中包含动词（或方法）： GET、POST、PUT、PATCH 还有 DELETE 是最常用的。
2. REST 是面向资源的，一个资源被一个 URI 所标识，比如 /articles/。
3. 端点（endpoint），一般指动词与 URI 的组合，比如 GET: /articles/。
4. 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”。
5. 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，一般以json格式返回

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

## 过滤器（Filter）、监听器（Listener）、拦截器（Interceptor）、servlet的区别

### Servlet

Servlet（Server let）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。

### Filter（责任链模式）

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.

### Listener（观察者模式）

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。

Listener有如下几个分类：

1.监听域对象的创建和销毁：

|  |  |
| --- | --- |
| ServletContext | ServletContextListener |
| HttpSession | HttpSessionListener |
| HttpServletRequest | ServletRequestListener |

2. 监听三个域对象的属性变更的监听器.(属性添加,属性移除,属性替换)

|  |  |
| --- | --- |
| ServletContext | Servlet ContextAttributeListener |
| HttpServletRequest | ServletRequestAttributeListener |
| HttpSession | HttpSessionAttributeListener |

3. 监听HttpSession对象中的JavaBean的状态的改变.(绑定,解除绑定,钝化和活化)2个

|  |  |
| --- | --- |
| HttpSession | HttpSessionBindingListener(绑定,解除绑定) |
| HttpSession | HttpSessionActivationListener(钝化和活化) |

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
  5. 跟Spring结合

### Interceptor

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。Filter配置必须放在listener前面（具体原因）

## cookies、token与session/reflesh token

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。

## 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。

## 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。

1. 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。
2. Log4j 2 Apache Log4j 2是apache开发的一款Log4j的升级产品。
3. 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。
4. 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）。
5. Logback 一套日志组件的实现(Slf4j阵营)。
6. 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

# DIY webserver

需要用到的知识：

1. OOP思想
2. 容器
3. IO
4. 多线程
5. 网络编程（TCP/UDP）
6. Xml解析（Sax解析）
7. 反射
8. HTML
9. HTTP协议
10. 请求格式例子

包含三个内容：

请求方法URI协议/版本  
请求头(Request Header)  
请求正文

GET /html HTTP/1.1

Host: localhost:8888

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101 Firefox/66.0

Accept: \*/\*

Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2

Accept-Encoding: gzip, deflate

Connection: keep-alive

Pragma: no-cache

Cache-Control: no-cache

POST /html HTTP/1.1

Host: localhost:8888

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101 Firefox/66.0

Accept: \*/\*

Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2

Accept-Encoding: gzip, deflate

Content-Type: text/plain;charset=UTF-8

Origin: moz-extension://bb4aa70b-ec29-4dc6-98d6-70f7456a5570

Content-Length: 15

Connection: keep-alive

Pragma: no-cache

Cache-Control: no-cache

username=steven

1. 响应格式例子

响应包含：

状态行

响应头(Response Header)

响应头与响应正文之间存在一个空行

响应正文

例子：

HTTP/1.1 200 OK Date:Tue May 14 23:36:29 CST 2019

Server:steven server/1.0;charset=GBK

Content-type:text/html

Content-length:90

<html><head><title>response</title></head><body><h1>hello world! aaaaaaa<h1></body></html>

# 电商概念

## 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 boot

* 1. maven setting.xml添加设置

<profiles>

<profile>

<id>jdk-1.8</id>

<activation>

<activeByDefault>true</activeByDefault>

<jdk>1.8</jdk>

</activation>

<properties>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

</properties>

</profile>

</profiles>

* 1. 创建maven工程
  2. 导入spring boot依赖相关包
  3. 编写启动类，启动springboot
  4. 编写相关controller、service、dao等业务逻辑
  5. 部署方式：在pom文件中添加maven插件：

<!--将应用打包成一个可执行jar-->

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

* 1. 使用java -jar package运行

POM文件：

* + 1. 父项目：

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.1.5.RELEASE</version>

<relativePath/> <!-- lookup parent from repository -->

</parent>

上一级的父项目:真正管理spring boot的所有依赖，所以导入的依赖默认不需要写版本号

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-dependencies</artifactId>

<version>2.1.5.RELEASE</version>

<relativePath>../../spring-boot-dependencies</relativePath>

</parent>

* + 1. 导入的依赖：场景启动器（各种starter），导入依赖所需要的组件，将功能场景进行抽取做成启动器（Starter），只需要在项目中引入这些starter，相关场景的依赖都会导入进来。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

* + 1. 主程序类

@SpringBootApplication：springboot配置类，表示这个springboot的配置类。

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration //开启自动配置功能

@ComponentScan(

excludeFilters = {@Filter(

type = FilterType.CUSTOM,

classes = {TypeExcludeFilter.class}

), @Filter(

type = FilterType.CUSTOM,

classes = {AutoConfigurationExcludeFilter.class}

)}

)

public @interface SpringBootApplication {

进入@ SpringBootConfiguration查看

@Target({ElementType.TYPE})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Configuration //spring 定义的注解，也是容器中的一个组件  
public @interface SpringBootConfiguration {  
}

进入@EnableAutoConfiguration

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@AutoConfigurationPackage 自动配置包，将主配置类的所在包及其下的所有子包扫描到容器中（重点）。

@Import({AutoConfigurationImportSelector.class })//@import是spring底层注解，给容器中导入一个组件，导入的组件由AutoConfigurationImportSelector.class指定，AutoConfigurationImportSelector.class将导入的组件以全类名的形式返回，最终会给容器中导入很多自动配置类，给容器中导入场景所需的所有组件，并配置好这些组件。

//自动配置类免去了手动编写注入功能等组件，主要用到了如下的方法

SpringFactoriesLoader.loadFactoryNames (EnableAutoConfiguration.class, classLoader)

springboot启动时会从META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入到容器中，自动配置类就生效，帮助我们自动配置。

public @interface EnableAutoConfiguration {

String ENABLED\_OVERRIDE\_PROPERTY = "spring.boot.enableautoconfiguration";

Class<?>[] exclude() default {};

String[] excludeName() default {};

}

Aaaaaaa

# spring cloud

## 微服务的概念

Spring boot 是 Spring 的一套快速配置脚手架，可以基于spring boot 快速开发单个微服务，Spring Boot，看名字就知道是Spring的引导，就是用于启动Spring的，使得Spring的学习和使用变得快速无痛。不仅适合替换原有的工程结构，更适合微服务开发。

Spring Cloud基于Spring Boot，为微服务体系开发中的架构问题，提供了一整套的解决方案——服务注册与发现，服务消费，服务保护与熔断，网关，分布式调用追踪，分布式配置管理等。

Spring Cloud是一个基于Spring Boot实现的云应用开发工具；Spring boot专注于快速、方便集成的单个个体，Spring Cloud是关注全局的服务治理框架；spring boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring boot来实现。

学过Spring的都知道，Spring开发有非常头疼的三点:

以启动一个带Hibernate的Spring MVC为例。

1. 依赖太多了，而且要注意版本兼容。这个应用，要添加10-20个依赖，Spring相关的包10多个，然后是Hibernate包，Spring与Hibernate整合包，日志包，json包一堆，而且要注意版本兼容性。

2. 配置太多了，要配置注解驱动，要配置数据库连接池，要配置Hibernate，要配置事务管理器，要配置Spring MVC的资源映射，要在web.xml中配置启动Spring和Spring MVC等

3.部署和运行麻烦。要部署到tomcat里面。不能直接用java命令运行。

太多重复和大家都一样的配置了。

Spring Boot的哲学就是约定大于配置。既然很多东西都是一样的，为什么还要去配置。

1. 通过starter和依赖管理解决依赖问题。

2. 通过自动配置，解决配置复杂问题。

3. 通过内嵌web容器，由应用启动tomcat，而不是tomcat启动应用，来解决部署运行问题。

Spring Cloud体系就比较复杂了。基本可以理解为通过Spring Boot的三大魔法，将各种组件整合在一起，非常简单易用。

浅析Spring boot与Spring cloud 之间的关系

你可以把spring boot的官方的包分为两类，一种是为了搭建一个服务用的，比如hibernate jpa，比如 message。另外一种含有cloud关键字的，是为了各个spring boot之前管理和使用的包。

因为当把集群、CI等方法集中进来一起考虑的时候，这件事情就复杂了。

多个小有服务整合成的大服务，要有一个消息总线来用于互相通知和调用，要有一个服务发现程序来管理某个小服务上线可用，同时在服务离线时也要能处理，各个小服务要尽量各自独立，还要考虑服务的依赖性，集群的负载均衡，配置文件的分离。

再把CI和Docker拿进来一起考虑的话，更乱。

但我认为这样完成的一个服务是更具有可插拔性，更容易维护的。而且遵循了上面的cloud方案的话，在服务的健壮性上面也很强。

写到这里对于新接触的我认为可以先从单独的spring boot程序开始入门，当要添加一个新功能时，考虑拆分成另外服务。两个程序间可以通过 jmx或是 其它消息中间件或是rest通讯。最后实现了一个各自独立的功能集群。

总结一句：Spring boot可以离开Spring Cloud独立使用开发项目，但是Spring Cloud离不开Spring boot，属于依赖的关系。

# 微服务

## 概念

「 微服务 」由 Martin Fowler 提出，它是指一种软件架构风格。一个大型的系统可以由多个微服务组成，每个微服务是被独立部署，独立完成自己的任务单元，微服务之间是通过API方式进行通信调用，是松耦合的。

这个模式听着是不是很熟悉的感觉？

因为在提出「 微服务 」概念之前，很多互联网公司的中大型项目早就是按照将业务拆分成独立单元的形式在部署和架构的，这与微服务的思路是一脉相通的，只不过实现方式没有现在这么规范与体系。

那「 微服务 」到底是怎么演变过来的呢？

在做一个新项目的时候，一开始项目大多数都很小，都是「 单体应用 」，这是很常见的做法。在项目规模小的时候，这种方式开发效率和运维效率都最高，符合互联网公司快速响应的要求。

但是随着业务量越来越大，项目也越来越复杂，开发团队人员也越来越多。这个时候还采用单体应用，问题就会很明显了。下面挑选两个最为常见的问题来举例：

协同问题：多个人同时开发一份代码，在工作协同上就会经常遇到代码冲突问题。

可用性问题：因为是单体应用，即使改个最小的功能，也需要整体发布，不仅直接影响了线上可用性，还可能会对正常功能带来风险。

为了解决这些问题，大家就开始考虑将「 单体应用 」进行拆分，进行服务化部署。然后又随着 Martin Fowler对「 微服务 」概念的提出，加上 DevOps 的流行，进一步促进了微服务的火热发展。

「 微服务 」的理念提倡每个服务都是单一职责，且每一个服务都能实现自治，因此可以带来一些明显好处：

部署简单：每个微服务都可以独立去部署，方便快捷。

逻辑清晰：将一个独立功能逻辑封装在单一微服务里面，实现整体项目的逻辑清晰。

可扩展：因为可以随时增加和减少微服务，可以很方便的扩展功能。

可靠性高：某一个功能的异常可以隔离在单一微服务里面，可以提高整体可靠性。

## 基础组件

要保证这么一套微服务架构能成功运行起来，我们起码需要以下这些 微服务的基础组件：

服务注册

部署了一个微服务节点，得让调用者知道啊，当微服务节点有增加或减少的时候，也得让调用者及时知晓啊。这些问题都是通过“服务注册”组件来实现的，服务提供者将自己的服务地址等信息登记到“服务注册”组件中，调用者需要的时候，每次都先去查询“服务注册”即可。免去人工维护微服务节点的信息同步问题。

服务网关

是指提供给外部系统调用的是统一网关。主要做安全和权限控制等。

配置中心

微服务的配置中心是用来统一管理所有微服务节点的配置信息的。因为同一个程序可能要适用于多个环境，所以在微服务实践中要尽量做到程序与配置分离，将配置进行集中管理。包括微服务节点信息、程序运行时配置、变量配置、数据源配置、日志配置、版本配置等。

服务框架

是指用来规范各个微服务节点之间通信标准的。服务间通信采用什么协议、数据是如何传输的、数据格式是什么样的。有了这个统一的“服务框架”就能保证各个微服务节点之间高效率的协同。

服务监控

微服务运行起来之后，为了能够监控节点的健康情况，保障节点的高可行，需要对各个服务节点进行收集数据指标、然后对数据进行实时处理和分析，形成监控报表和预警。

服务追踪

一旦使用了微服务架构，那么当有请求过来时，就会经过多个微服务节点的处理，形成了一个调用链。为了进行问题追踪和故障的定位，需要对请求的完整调用链进行记录。

这里的服务追踪与上面的服务监控是不同维度的，一个是全局的，一个是微观的，发挥的作用也不一样。

服务治理

就是指需要通过准备一些策略和方案，来保障整个微服务架构在生产环境遇到极端情况下也能正常提供服务的措施。比如 熔断、限流、隔离等等。

当然，上述只是一个微服务架构最为核心的基础组件，一旦微服务体系过大，例如有几十上百个微服务节点，那么开发、维护、测试的成本就会非常大。因此一般还会引入 自动化部署 和 自动化测试 来提高协同效率。

## 常见问题

不是所有项目都适用微服务

有些项目规模还比较小，或者项目才刚立项启动，也只有三四个人负责开发维护，这时候是不建议一上来就搞微服务架构的。这种情况下搞微服务，不仅是“杀鸡用牛刀”，而且还无谓的增加了项目的复杂度，本身一个单体结构就可以搞定的事情，非得拆分N多节点，人员又不足以支撑这么多节点的开发维护，这完全是自找苦吃。反而是等项目成熟了、规模大了之后，再开始慢慢将原有结构拆为微服务才是正确的做法。

不要拆分过多过细的服务

即使项目经过评估后适合拆为微服务架构，但也不要过度拆解。有的团队喜欢将项目拆成很细很细的颗粒，最后把项目搞的特别复杂，整个团队都陷进去了。

拆分服务的颗粒度应该根据业务发展和团队现状综合去考虑。这里可以参考一个很火的理论「 康威定律 」。什么样的团队，就产生什么样的架构，微服务拆分的颗粒度是需要和团队结构相匹配的。当你着手拆微服务的时候，得先评估一下团队人员和素质，一般在开发期，2-3个人开发一个服务是合理的，在维护期，1个人维护2-3个服务也是合理的。

如果拆分过细，开发人员跟不上，会严重降低大家的工作效率。并且过细的服务，会导致一个请求的调用链条很长，不仅会影响请求的响应时间，也会对线上问题排查带来增加难度。

没有DevOps就不要急于微服务

一个稳定的微服务架构，是需要 持续集成、自动化部署、自动化测试、健全的监控体系来保障的。如果团队还不具备DevOps，这些基础的建设都没有做好，一上来就搞微服务的话，就会导致实施过程中问题百出，微服务的优势不能发挥。

基本实现

# 设计模式

设计模式的核心思想：

实现系统各个角色的分工，便于系统的维护。（专业的人做专业的事）

分类:

1. 创建型模式：

单例模式、工厂模式、抽象工厂模式、建造者模式、原型模式

1. 结构型模式：

适配器模式、桥接模式、装饰器模式、组合模式、外观模式、享元模式、代理模式

1. 行为型模式：

模板方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式、状态模式、策略模式、责任链模式、访问者模式。

六个原则：

OCP原则（开闭原则）：开闭原则的意思是：**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。

里氏替换原则（LSP）：

里氏代换原则是面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。里氏代换原则是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

依赖倒转原则（DIP）：这个原则是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

接口隔离原则（ISP）：使用多个隔离的接口，比使用单个接口要好。它还有另外一个意思是：降低类之间的耦合度。由此可见，其实设计模式就是从大型软件架构出发、便于升级和维护的软件设计思想，它强调降低依赖，降低耦合。

迪米特法则（最少知道原则，DP）：一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。

合成服用原则（CRP）：尽量使用合成/聚合的方式，而不是使用继承。

## 单例

实现功能：

确保一个类只有一个对象在运行中。

举例：

Windows中的任务管理器、回收站

数据库连接池

Servlet编程中的Application

Spring中每个bean默认是单例模式

Servlet中的每个Servlet也是单例

SpringMVC中的控制器也是单例模式

单例模式的优点：

由于只生成一个实例，内存占用系统开销小，当一个对象的产生需要比较多的资源时，如读取配置文件、产生其他依赖对象时，可以通过在应用中启动时直接产生一个单例对象，然后永久主流内存。

UML：



五种单例模式：

1.饿汉式（线程安全，调用效率高，不能延时加载）

2.懒汉式（线程安全，调用效率不高（最差），但可以延时加载）

3.双重加测锁式（由于JVM底层内部模型原因，偶尔会出问题，不建议使用）

4.静态内部类式（线程安全，调用效率高，可以延时加载）

5.枚举单例（线程安全，调用效率高，不能延时加载，避免序列化与反序列化的漏洞）

### 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，静态属性不进行初始化，懒加载

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(类初始化时，立即加载这个对象（没有延时加载的优势）。加载类时，天然时线程安全的)

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（使用synchronized保证线程安全）

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

在多线程下，以上创建单例的方法可能出现数据不准确的问题（原因是线程的并发以及代码的指令重排），这时可以使用volatile和DoubleChecking 组合（DCL doubleCheckLocking）来保证单例模式的准确性。

### 双重检测机制

由于懒汉式实现synchronized加在了方法上，导致每个线程都需要排队等待进入方法中判断是否已经初始化类，这里可以使用DCL模式缩小同步范围，提高效率：

if(singletonDoubleCheckDemo3 == null){  
 synchronized (SingletonDoubleCheckDemo3.class){  
 if(singletonDoubleCheckDemo3 == null){  
 singletonDoubleCheckDemo3 = new SingletonDoubleCheckDemo3();  
 }  
 }  
 }  
}  
return singletonDoubleCheckDemo3;

注意：这种方式由于JVM内部实现存在指令重排的问题在实际中并不实用，要使用这种方式可以对变量使用volatile禁止指令重排。

### 静态内部类实现

实现过程：

1.定义静态内部类持有外部类的静态final实例

2.提供private构造器

3.提供public的静态方法访问静态类中持有的当前类对象。

### 反序列化及反射漏洞解决方案

通过反射可以破解以上几种单例实现方式（非枚举的实现）

Class<SingletonDemo6> clz = (Class<SingletonDemo6>) Class.forName("com.steven.designModes.singleton.SingletonDemo6");

Constructor<SingletonDemo6> c = clz.getDeclaredConstructor();

//跳过安全类型检查

c.setAccessible(true);

SingletonDemo6 s3 = c.newInstance();

SingletonDemo6 s4 = c.newInstance();

System.out.println(s3.hashCode());

System.out.println(s4.hashCode());

解决方案：

在私有化的构造器中判断是否已经存在了实例，存在则抛出异常：

通过反序列化可以破解以上的单例模式：

fos = new FileOutputStream("D:/a.txt");

oos = new ObjectOutputStream(fos);

oos.writeObject(s2);

ReleaseSourceTools.release(oos,fos);

ois = new ObjectInputStream(new FileInputStream("D:/a.txt"));

SingletonDemo6 s3 = (SingletonDemo6) ois.readObject();

System.out.println(s3.hashCode());

## 建造者模式

使用场景：

需要构建复杂的模型，使用多个简单的对象一步一步构建成一个复杂的对象。将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。

例子：mysql中sessionFactory的构建，StringBuilder、SaxBuilder、DomBuilder都用到了建造者模式。

使用多个简单的对象一步一步构建成一个复杂的对象

主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。



角色：

实体模型：具体需要建造的模型及模型组件

建造者：定义生成实例所需的所有方法

具体建造者：实现生成实例所需的所有方法，并定义获得最终生成实例的方法。

装配者：定义如何装配组建

具体装配者：定义具体装配方式

## 工厂模式

实现创建者和调用者分离

三种分类：

简单工厂

工厂方法

抽象工厂

### 简单工厂

实现方式：

1.创建需要生产的模型接口

2.实现生产模型接口类

2.创建工厂，通过静态方法创建返回具体要生产的实体类。

简单工厂（静态工厂）实现简单，但对生产新的类需要修改工厂代码才能实现，扩展性差。



### 工厂方法

工厂方法与简单工厂的不同在于，简单工厂只有一个工厂类，而工厂方法有一组实现相同接口的工厂类。完全满足（OCP）。

实现方式：

1.创建需要生产的模型接口

2.实现生产模型接口

3.创建工厂接口

4. 实现工厂接口，创建具体生产某种实体的工厂类

5.客户端调用使用具体生产某一实体的工厂类

工厂方法模式结构更复杂，实际使用中简单工厂模式用的更多。



### 抽象工厂

用来生产不同产品族的全部产品，对新增产品无能为力，支持增加产品族。

实现方式：

在一个工厂里聚合多个同类产品。例如，生产汽车需要生产座椅、引擎、轮胎，这三个实体类又对应了高端低端的分类。这时要生产汽车可以对这些不同的分类进行组合，生产不同价位的汽车。如下面图中的高端奢侈汽车工厂及低端汽车工厂，在工厂内聚合了多个对象形成某一特定产品。



## 原型（prototype）模式

是用于创建重复的对象，同时又能保证性能。用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

应用场景：

1、当一个系统应该独立于它的产品创建，构成和表示时。 2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。 3、为了避免创建一个与产品类层次平行的工厂类层次时。 4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

如果需要创建大量相同的对象，而且new方式比较耗时，可以使用原型（clone方法）模式。

实现方式：

 1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。

2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

另外，可以通过IO流的序列化与反序列化实现原型模式



## 装饰器模式

作用:允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。动态地给一个对象添加一些额外的职责。就增加功能来说，装饰器模式相比生成子类更为灵活。

四个组件：

1.抽象组件：需要装饰的抽象对象（接口或抽象父类）

2.具体组件：需要装饰的对象

3.抽象装饰类：包含抽象组件的引用及装饰者共有的方法

4.具体装饰类：被装饰的对象

## 适配器模式

## 代理模式

代理模式核心作用：通过代理控制某个对象的访问。可以详细控制访问某个（某类）对象的方法，在调用这个方法前做前置处理，在调用这个方法后做后置处理。（AOP的核心概念）

动态代理是AOP实现的核心机制

核心角色：

1.抽象角色：定义代理角色和真实角色的公共对外方法

2.真实角色：实现抽象角色，定义真是角色所需实现的业务逻辑

3.代理角色：实现抽象角色，是真实角色的代理，通过真实角色的业务逻辑方法来实现抽象方法，并可以附加自己的操作。

应用场景：

安全代理：屏蔽对真实角色的直接访问。

远程代理：通过代理类处理远程方法调用（RMI）。

延迟加载：先加载轻量级的代理对象，真正需要再加载真实对象。

### 静态代理

静态定义代理类

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

实现：

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。

静态代理UML举例



### 动态代理

动态生成代理类

实现方式：

JDK代理

Javassist

Cglib

Asm

JDK自带的动态代理API（使用反射实现的）

Java.lang.reflect.proxy：生成动态代理类和对象

Java.lang.reflect.Invocationhandler：处理器接口，通过invoke方法实现对真实角色的代理访问，每次通过Proxy生成代理对象时都需要指定对应的处理器对象。

JDK动态代理需要有顶层接口才能使用，但是在只有顶层接口的时候也可以使用，常见是mybatis的mapper文件是代理。

从Proxy.newProxyInstance( ClassLoader paramClassLoader,  Class<?>[] paramArrayOfClass,  InvocationHandler paramInvocationHandler)进行观察，简单来看就是先生成新的class文件，然后加载到jvm中，然后使用反射，先用class取得构造方法，然后使用构造方法反射得到他的一个实例。

Cglib动态代理 可以直接代理类，使用字节码技术，不能对 final类进行继承。使用了动态生成字节码技术。

使用反射完成。使用了动态生成字节码技术。

动态代理UML图举例



Spring中的动态代理：

如果一个类有顶层接口，则默认使用jdk的动态代理来代理，如果直接是一个类，则使用cglib动态代理。 其次，如果没有需要代理的方法，如所有方法都没有@Transactional注解，Aop这种，则不会被代理。

## 迭代器模式

Java容器中的Iterator使用了迭代器模式。

# 数据库

## 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**。

## 数据源、数据库连接池

### 数据源

数据源定义的是连接到实际数据库的一条路径而已，数据源中并无真正的数据，它仅仅记录的是你连接到哪个数据库，以及如何连接的，如odbc数据源。也就是说数据源仅仅是数据库的连接名称，一个数据库可以有多个数据源连接。

### 数据库连接池

数据库连接池是负责分配、管理和释放数据库连接。使用数据库连接池是因为数据库连接是一种关键的有限的昂贵的资源，这一点在多用户的网页应用程序中体现得尤为突出。如weblogic、tomcat、WebSphere容器都实现了数据库连接池，但是数据库连接池是可以独立出来自己编码实现的。

二者关系：数据源是用于访问连接池或多池的JNDI对象

### Jndi

java naming and directory interface。简单点就是你按命名规则给一个东西命名然后你就可以通过该名字在特定环境下直接查找到该东西了。

JNDI是用于向Java程序提供目录和命名功能的API。可以简单地把JNDI理解为一种将对象和名字绑定的技术，对象工厂负责生产出对象，这些对象都和惟一的名字绑定。外部程序可以通过名字来获取对某个对象的引用。在一个文件系统中，文件名被绑定给文件。在DNS中，一个IP地址绑定一个URL。在目录服务中，一个对象名被绑定给一个对象实体。

### Jdbc

java database connectivity standard 是一套规范的面向应用程序的接口，通过它可以访问各类关系数据库。JDBC API 中定义了一些Java类分别用来表示与数据库的连接（connections）, SQL语句（SQL statements）, 结果集（result sets）以及其它的数据库对象, 使得Java程序能方便地与数据库交互并处理所得的结果。

jdbc可以做三件事：与数据库建立连接，发送sql语句，处理结果

# Mybatis

问题1：mapper接口为什么能堆数据库进行访问？

问题2：spring容器中，为什么没有出现sqlSession？

## mybatis原理

mybatis架构图

mybatis核心流程:

1.初始化阶段：读取XML配置文件和注解中的配置信息，创建配置对象（Configuration），并完成各个模块的初始化工作。

2.代理阶段：创建sqlsession响应请求，动态代理封装iBatis编程模型，使用mapper接口访问数据库。

3.数据库读写阶段：遵循JDBC规范通过SqlSession完成SQL的解析、参数的映射，SQL的执行，结果的解析过程。

### 1.1主要成员

* Configuration        MyBatis所有的配置信息都保存在Configuration对象之中，配置文件中的大部分配置都会存储到该类中
* SqlSession            作为MyBatis工作的主要顶层API，表示和数据库交互时的会话，完成必要数据库增删改查功能
* Executor               MyBatis执行器，是MyBatis 调度的核心，负责SQL语句的生成和查询缓存的维护
* StatementHandler 封装了JDBC Statement操作，负责对JDBC statement 的操作，如设置参数等
* ParameterHandler  负责对用户传递的参数转换成JDBC Statement 所对应的数据类型
* ResultSetHandler   负责将JDBC返回的ResultSet结果集对象转换成List类型的集合
* TypeHandler          负责java数据类型和jdbc数据类型(也可以说是数据表列类型)之间的映射和转换
* MappedStatement  MappedStatement维护一条<select|update|delete|insert>节点的封装
* SqlSource              负责根据用户传递的parameterObject，动态地生成SQL语句，将信息封装到BoundSql对象中，并返回
* BoundSql              表示动态生成的SQL语句以及相应的参数信息

### 1.2 简单starter

Java中sql操作全过程：



在mybatis中编写sql过程被写入xml配置文件中，其他步骤都由mybatis自动完成，程序员只需要关注sql的编写与优化

Mybatis简单使用过程：

1.根据xml配置文件（全局配置文件）创建一个SqlSessionFactory对象，SqlSessionFactory中包含了数据源的的配置信息。

2.编写sql映射文件（xml的mapper文件），配置每个sql的封装规则。

3.将映射文件注册在全局配置文件中

4.根据配置文件得到SqlSessionFactory对象

5.使用SqlSessionFactory对象获取sqlSession，使用sqlSession对象执行增删改查

老版本mybatis不适用mapper接口的写法：

6.使用唯一标识告诉mybatis执行哪一个sql。

新版本使用mapper接口的写法：

6.使用sqlSession获取mapper接口实现类，mybatis会为接口自动创建一个代理对象，代理对象去实现增删改查方法。

SqlSession是非线程安全的，每次使用都必须关闭该session。

小知识：

#{}：表示预编译

${}：表示直接注入值

## mybatis与spring整合原理

# 服务器相关

## tomcat

## netty

## redis

## RPC框架

# 协议相关

## 1.TCP

# 前后端分离技术（探讨，没有固定模式）

## 传统开发方式

在以前传统的网站开发中，前端一般扮演的只是切图的工作，只是简单地将UI设计师提供的原型图实现成静态的HTML页面，而具体的页面交互逻辑，比如与后台的数据交互工作等，可能都是由后台的开发人员来实现的，或者是前端是紧紧的耦合后台。比如，以前淘宝的Web基本上都是基于MVC框架webx，架构决定了前端只能依赖后端。所以他们的开发模式依然是，前端写好静态demo，后端翻译成VM模版，这种模式的问题就不说了，被吐槽了很久。

而且更有可能后台人员直接兼顾前端的工作，一边实现API接口，一边开发页面，两者互相切换着做，而且根据不同的url动态拼接页面，这也导致后台的开发压力大大增加。前后端工作分配不均。不仅仅开发效率慢，而且代码难以维护。而前后端分离的话，则可以很好的解决前后端分工不均的问题，将更多的交互逻辑分配给前端来处理，而后端则可以专注于其本职工作，比如提供API接口，进行权限控制以及进行运算工作。而前端开发人员则可以利用nodejs来搭建自己的本地服务器，直接在本地开发，然后通过一些插件来将api请求转发到后台，这样就可以完全模拟线上的场景，并且与后台解耦。前端可以独立完成与用户交互的整一个过程，两者都可以同时开工，不互相依赖，开发效率更快，而且分工比较均衡。

## 前后端分离的简单实现

* 前端的工作：实现整一个前端页面以及交互逻辑，以及利用ajax与nodejs服务器（中间层)交互
* 后端的工作：提供API接口，利用redis来管理session,与数据库交互

一般来说，要实现前后端分离，前端就需要开启一个本地的服务器来运行自己的前端代码，以此来模拟真实的线上环境，并且，也是为了更好的开发。因为你在实际开发中，你不可能要求每一个前端都去搭建一个java(php)环境，并且在java环境下开发，这对于前端来说，学习成本太高了。但如果本地没有开启服务器的话，不仅无法模拟线上的环境，而且还面临到了跨域的问题，因为你如果写静态的html页面，直接在文件目录下打开的话，你是无法发出ajax请求的(浏览器跨域的限制),因此，你需要在本地运行一个服务器，可是又不想搭建陌生而庞大的java环境，怎么办法呢？nodejs正好解决了这个问题。在我们项目中，我们利用nodejs的express框架来开启一个本地的服务器，然后利用nodejs的一个http-proxy-middleware插件将客户端发往nodejs的请求转发给真正的服务器，让nodejs作为一个中间层。这样，前端就可以无忧无虑的开发了。

由于前后端分离后，前端和后台同时开发时，就可能遇到前端已经开发好一个页面了，可是却等待后台API接口的情况。比如说A是负责前端，B是负责后台，A可能用了一周做好了基本的结构，并且需要API接口联调后，才能继续开发，而此时B却还没有实现好所需要的接口，这种情况，怎么办呢？在我们这个项目里，我们是通过了mock来提供一些假数据，我们先规定好了API接口，设计出了一套API文档，然后我们就可以通过API文档，利用mock(http://mockjs.com)来返回一些假数据，这样就可以模拟发送API到接受响应的整一个过程，因此前端也不需要依赖于后端开发了，可以独立开发，等到后台的API全部设计完之后，就可以比较快速的联调。

3.nodeJS的作用

引入nodejs主要是为了分层开发，职责划分，nodejs作为前端服务器，由前端开发人员负责，前端开发人员不需要知道java后台是如何实现的，也不需要知道API接口是如何实现的，我们只需要关心我们前端的开发工作，并且管理好nodejs前端服务器，而后台开发人员也不需要考虑如何前端是如何部署的，他只需要做好自己擅长的部分，提供好API接口就可以；

nodejs本身有着独特的异步、非阻塞I/O的特点，这也就意味着他特别适合I/O密集型操作，在处理并发量比较大的请求上能力比较强，因此，利用它来充当前端服务器，向客户端提供静态文件以及响应客户端的请求。

# Hibernate