Java Interview Question

Java

Java Object 重写 equals() 方法的同时为什么要重写 hashCode()?

因为 equals() 与 hashCode() 必须保持一致；

- 当 obj1.equals(obj2) 为 true，obj1.hashCode() 必须等于 obj2.hashCode()；
- 当obj1.hashCode() == obj2.hashCode()为false时，obj1.equals(obj2)必须为false；

集合

集合的产生：为保存数目不确定的对象，解决数组长度固定的问题；

集合类：可以**存储任意类型的对象**，并且**长度可动态扩展**的类；

集合与数组比较	数组	集合
长度	长度固定	动态扩展
元素类型	一种类型	任意类型

集合 API

Collection：

- List：ArrayList，Vector，LinkedList；
- Set：HashSet，TreeSet；

Map：HashTable，HashMap，TreeMap，LinkedHashMap；

Collection

Collection - 最基本的集合接口，一个 Collection 存储一组 Object。

Collection 子接口：List 接口，Set 接口；

List

List - 列表**【有序可重复】**；

> 有序指插入顺序；

ArrayList

ArrayList - 基于**可变数组**实现的 List；

- 允许多个 null 值（有序可重复）；
- 线程不同步（线程不安全）；
- 查询效率高；

Vector（已过时）

Vector - 基于**可变数组**实现的 List；

- 允许多个 null 值（有序可重复）；
- 线程同步（线程安全）；
- 查询效率高；

LinkedList

LinkedList - 基于**双向链表**实现的 List

- 允许多个 null 值（有序可重复）；
- 线程不同步（线程不安全）；
- 插入删除效率高；

Set

Set - 数学意义上的集合**【无序不重复】**，满足确定性、互异性、无序性；

> 无序指插入顺序；

HashSet

HashSet - 基于 HashMap （哈希表）实现，元素为 HashMap 的 key；

- 允许一个 null 值（无序不重复）；
- 线程不同步（线程不安全）；
- 通过 equals() 和 hashCode() 方法判断重复元素。

TreeSet

TreeSet- 基于 TreeMap （二叉树）实现，元素为 TreeMap 的 key；主要用来**元素排序。**

- 允许一个 null 值（无序不重复）；
- 线程不同步（线程不安全）；
- 通过集合元素类实现 Comparable 接口，重写 compareTo() 方法判断重复元素。

自然排序（Comparable）

自然排序：通过集合元素类实现 Comparable 接口，重写 compareTo() 方法排序。

comparaTo() 返回值：

TreeSet 底层为一个二叉树

- `return 0;` 表示集合中只存一个元素。元素值每次比较，都认为是相同的元素，这时就不再向TreeSet中插入除第一个外的新元素。
- `return 1;` 表示集合正序排列。元素值每次比较，都认为新插入的元素比上一个元素大，于是二叉树存储时，会存在根的右侧，读取时就是正序排列的。

- `return -1;` 表示集合倒序排列。元素值每次比较，都认为新插入的元素比上一个元素小，于是二叉树存储时，会存在根的左侧，读取时就是倒序序排列的。

比较器排序 `Comparator<T>`

创建 TreeSet 类时制定一个 Comparator 接口，重写 compara() 方法制定排序规则。

Iterator 迭代器

hasNext() - 判断是否存在下一个元素；

next() - 获取下一个元素；

ListIterator

Map

Map - 键值对集合**【无序双列集合】**，一个 Map 存储一组键值对，提供键 (key) 与值 (value) 映射。

- 键不允许重复；

HashTable

HashTable - 散列表，

- 线程同步（使用 synchronized 实现同步）

HashMap ★

HashMap - 基于哈希桶数组实现的 Map；

- 生成相同 hashCode 的不同 key 存储在同一个 bucket 下，null key 存储在 0 bucket 下。

HashTable与HashMap的区别

1. 关于null：

- HashTable不支持 null-key 和 null-value。HashTable 遇到 null，抛出 NullPointerException。
- HashMap支持 null-key 和 null-value。HashMap 对 null 做了特殊处理，将 null 的 hashCode 值定为了 0，从而将其存放在哈希表的第0个 bucket 中。

2. 扩容方式：

- HashTable 默认初始化容量大小为11，之后每次扩充为原来的2n+1。
- HashMap默认初始化容量大小为16，之后每次扩充为原来的2倍。

> 在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从hash计算的效率上，又是HashMap更胜一筹。

3. 线程安全：

- HashTable 线程安全（同步）；

- HashMap 线程不安全（不同步）；

> HashTable已经被淘汰了，如果你不需要线程安全，使用HashMap；如果你需要线程安全，使用ConcurrentHashMap；

4. 数据结构：

- HashTable 数组+链表
- HashMap 数组+链表/红黑树（JDK1.8）

ConcurrentHashMap和Hashtable的区别

- HashTable 采用 synchronized 实现同步，单锁锁定整个集合，迭代时其他线程必须等待其迭代完成才能访问map，所以当 Hashtable 的大小增加到一定的时候，性能会急剧下降。
- ConcurrentHashMap 引入了分割（segmentation），仅锁定 map 的某个部分，更适用于高并发。

TreeMap

TreeMap - 基于红黑树实现的 Map；

IO&NIO

NIO流

NIO面向缓冲区的，基于通道的IO操作； (JDK1.4已产生)

缓冲区（Buffer）：存储数据；
通道（Channel）：传输数据；

NIO与IO的区别：

IO：基于流；阻塞式（每次只能操作一种流）；
NIO：面向缓冲区，基于通道，选择器；非阻塞式；

NIO将以更加高效的方式进行文件的读写操作；

缓冲区（Buffer）

缓冲区：一个用于特定基本类型数据的容器；

缓冲区作用：保存数据；进行数据读写；

Buffer常见实现：

ByteBuffer，ShortBuffer，IntBuffer，LongBuffer， FloatBuffer，DoubleBuffer，CharBuffer；无布尔型缓冲区。

①建立缓冲区，分配容量：

ByteBuffer b = ByteBuffer.allocate(1024);

清空缓冲区：Clear()

②缓冲区属性

位置：position();

限制：limit();

容量：capacity();

③读/写：get()；put()；

④ 读写模式切换：filp();

⑤标记：mark()；reset()；

非直接缓冲区：allocate(capacity)；

传输方式：拷贝方式；

内存位置：位于堆区；

特点：占用资源较少，容易被释放；But效率低；

直接缓冲区：allocateDirect(capacity)；

传输方式：内存映射；

内存位置：直接位于内存页；

特点：效率高；But分配资源消耗大，不易被回收；

应用：一般分配给易受基础系统的本机IO操作的大型；

通道（Channel）

通道作用：传输数据；

1.Java 为 Channel 接口提供的最主要实现类如下：

•FileChannel：用于读取、写入、映射和操作文件的通道。

•DatagramChannel：通过 UDP 读写网络中的数据通道。

•SocketChannel：通过 TCP 读写网络中的数据。

•ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

2.获取通道：

本地IO：调用getChannel()；

FileInputStream/FileOutPutStream

RandomAccessFile

网络IO：

Socket

ServerSocket

DatagramSocket

获取通道的其他方式是：

Files类静态方法：newByteChannel() 获取字节通道（JDK1.7）

Channel类静态方法：open(Path path,OpenOpertion ... oo)（JDK1.7）

获取通道：

①本地IO获取通道：本地IO.getChannel()；

②打开通道FileChannle.open(Path path,OpenOpertion ... oo)

使用通道进行数据传输：

①使用通道+非直接缓冲区完成文件复制；

②使用直接缓冲区完成文件复制（内存映射文件）

③直接使用通道完成数据传输；

分散读取和聚集写入

①分散读取：read(ByteBuffer[] bufs)；

②聚集写入：write(Bytebuffer buf1)；

while((inChannel.read(bufs)) != -1) { **for**(ByteBuffer b : bufs) { b.flip(); outChannel.write(b); b.clear(); } }

NIO的非阻塞式网络通信：

空闲通道：多路复用；

网络通信的三要素：

```
            IP地址：可以唯一的定位到一台计算机

            端口号：可以唯一的定位到一个程序

            通信协议：TCP/IP  UDP

    阻塞式：

            客户端：

                    \1. 获取通道

                    \2. 分配指定大小的缓冲区

                    \3. 读取本地文件，并发送到服务端

                    \4. 关闭通道

            服务端：
```

\1．获取通道

\2．绑定连接

\3．获取客户端连接的通道

\4．分配指定大小的缓冲区

\5．接收客户端的数据，并保存到本地

\6．关闭通道

非阻塞式：

客户端

\1．获取通道

\2．切换非阻塞模式

\3．分配指定大小的缓冲区

\4．发送数据给服务端

\5．关闭通道

服务端

\1．获取通道

\2．切换非阻塞模式

\3．绑定连接

\4．获取选择器

\5．将通道注册到选择器上，并且指定“监听接收事件”

\6．轮询式的获取选择器上已经“准备就绪”的事件

\7．获取当前选择器中所有注册的“选择键(已就绪的监听事件)”

\8．获取准备“就绪”的是事件

\9．判断具体是什么事件准备就绪

\10．若“接收就绪”，获取客户端连接

\11．切换非阻塞模式

\12．将该通道注册到选择器上

```
\13．获取当前选择器上“读就绪”状态的通道

\14．读取数据
```

选择器（Selector）

作用：针对非阻塞式IO通信；

打开选择器：Selector.open();

将通道注册到选择器：

Selector.select()：查看选择器注册的通道数；

键：SelectionKey，状态值

值：Channel，通道

多线程

多线程概述

进程：一个正在运行的程序；在计算机中的运行路径；

一个程序只能有一个进程；

程序是静态的，进程是动态的；

线程：进程的组成单元，在一个进程中有一个或多个线程；

并发：多个任务轮询交替执行；

并行：多任务同时执行；

CPU调度方式：

①时间片轮询；

②抢占式调度方式； Java采用抢占式；

进程与线程共享“堆内存和方法区”；一个线程一个“栈内存”；

Java程序的运行原理：

当使用Java命令启动一个程序的时候，JVM启动，就启动了一个进程；main方法会执行，main方法就是该进程中的一个线程（主线程），同时也会启动GC是后台线程（守护线程）；

创建线程

1.创建线程

①继承Thread类；

②实现Runnable接口：本质是Thread(Runnable target);

核心步骤：重写run方法，写入该线程要完成的事情；

2.使用步骤：

```
a）重写run方法；

b）创建线程对象；

c）调用start()；
```

3.两种线程实现方式的区别：

	继承Thread类	**实现Runnable接口**
优点：	可直接使用线程所有方法；	多实现；
缺点：	单继承；	不能直接创建线程；

线程控制

1.线程名称

①设置名称：

Thread(String name)；

setName(String name)；

②String getName()；

③long getId()；//线程终生不变的唯一的标识符；

系统会为创建的线程设置默认Name：Thread-01；

主线程默认Name：main；

2.获取当前执行线程

Thread.currentThread()；返回对当前正在执行的线程对象的引用；

3.线程休眠

Thread.sleep(long millis)；在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）；

4.守护线程

void setDaemon(true)；将线程标记为守护线程；

isAlive()；判断线程是否是活动状态（已启动未终止）；

isDaemon()；判断线程是否是守护线程；

5.加入线程

void join()；当前线程暂停执行，直到加入的线程执行完毕；

应用：可以让另一个线程优先执行；

6.礼让线程

Thread.yield()；暂停当前正在执行的线程对象，其他线程并不一定能抢占到。

7.线程优先级

Java抢占式线程调度算法；优先级1-10；默认优先级5；

getPriority()；返回线程的优先级。

setPriority(int newPriority)；更改线程的优先级。

线程异常只能处理，不能抛出，无接受异常者；

线程的生命周期

线程状态

1.新建（New）：创建线程对象（new）；JVM为其分配内存，并初始化其成员变量的值

2.就绪（Runnable）：线程对象调用start()；JVM为其创建方法调用栈和程序计数器，等待CPU调度运行；

3.运行（Running）：线程获得CPU使用权，开始执行run()；

4.阻塞（Blocked）：当前正在执行的线程由于某种原因（休眠，礼让，时间到达）暂时失去CPU使用权，等待再次获得CPU使用权；

线程进入阻塞状态：

①wait()；

②执行同步锁时，锁对象被其他线程占用；

③IO操作；

④sleep()，join()；

解除阻塞：

\5. 死亡（Dead）：线程执行结束；isAlive()==false；

①run()或call()方法执行完成，线程正常结束。

②线程抛出一个未捕获的Exception或Error。

③直接调用该线程stop()方法来结束该线程——该方法容易导致死锁，通常不推荐使用。

线程同步

线程同步：保护共享数据，防止数据不一致；

①同步代码块：

synchronized(obj){ }

锁对象：可任意对象；但要保证多线程使用同一锁对象，实现多线程同步；

不能在run()中的同步代码块中使用this作为锁对象；

②同步方法：

同步方法：synchronized关键字修饰的方法；

普通方法锁对象：this；

静态方法锁对象：本类Class对象；

多线程环境中安全使用集合API

Collections.synchronizedXxxx()；

Collections中的静态synchronized方法：

①Collection synchronizedCollention(Collection c)；

②List synchronizedList(list l)；

③Set synchronizedSet(Set s)；

④Map synchronizedMap(Map m)；

// Synchronized集合是线程安全的，但Iterator不是线程安全的； List list = Collections.synchronizedList(new ArrayList()); ... //迭代时，阻塞其他线程调用add()或remove()修改元素； synchronized(list) { Iterator i = list.iterator(); // Must be in synchronized block while (i.hasNext()) foo(i.next()); }

定时任务

Timer类

TimerTask：实现 Runnable接口，重写run()方法；由 Timer 安排为一次执行或重复执行的任务。

void schedule(TimerTask task, Date firstTime, long period)；

*task 执行的任务

*firsttime 开始执行的时间

*period 任务的间隔执行时间

Java并发编程Executor框架与线程池

Executor框架：基于并发编程的线程池；

Executor：

ExecutorService：

创建线程池

Executors的静态方法：

①创建一个固定线程数的线程池：

static ExecutorService Executors.newFixedThreadPool(int nThreads)；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则任务进入阻塞状态，等待空闲线程；

线程池执行任务：void execute(Runnable command)；

②创建一个带有缓存的线程池

static ExecutorService newCachedThreadPool()；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则创建一个新线程执行任务；

③创建一个单线程线程池

static ExecutorService newSingleThreadExecutor()；

线程池中，如果有空闲线程，则执行任务；如果没有空闲线程，则创建一个新线程执行任务；

④创建一个定时执行线程池

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

类似FixedThreadPool；

定时执行方法：

ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)；

- command - 要执行的任务
- delay - 从现在开始延迟执行的时间
- unit - 延迟参数的时间单位

一般情况下，生命周期短的CachedThreadPool是首选；但是在特殊情况下（线程数>系统负载），生命周期长的会选择FixedThreadPool；

线程池可执行的任务：

Rannable接口：public void run()；//无返回值；

Callable接口：public V call()； //有返回值；

submit()；

get()；

线程池的生命周期

运行状态：

关闭状态：shutdown()；任务提交，不在执行新任务；

终止状态：任务全部提交完毕

线程协作

多个线程同时完成一个任务时，多个线程共享任务数据；防止数据不一致，使用线程同步（synchronized）机制；

synchronized：

原子性：在每次执行任务时，能保证只有一个线程在执行任务；

可见性：针对每一个线程执行任务时，所见到的数据都真是有效；

1.用5个售票窗口销售100张火车票；

2.银行账户有1000元，一个人去柜台取钱，另一个人去ATM机取钱；

取款方式：柜台，ATM机；

银行类，一个账户，两个线程模拟取钱，

3.火车过山洞：

火车由A到B，中间有一个隧道，每次只允许一趟火车通过，每趟火车经过隧道需要10s，为防止事故发生，每次只允许一趟火车经过，现有5趟火车经过隧道；

线程死锁

多个线程争取锁对象的持有权时的竞争现象；

避免死锁：

①尽量避免锁的嵌套使用；

②让程序执行时，尽可能只获得一个锁；

③超时限制：Lock中的tryLock，在获取锁对象时，可设置时间限制；

线程通信

Object类的监视器方法（monitor）

等待/通知机制

Object监视器方法：wait()、notify()、notifyAll()；

wait()：使线程进入等待状态，如果没有线程来唤醒，则一直处于等待状态；

wait(long timeout)：使线程进入等待状态；

notify()：唤醒当前处于等待状态的随机一个线程；

notifyAll()：唤醒当前处于等待状态的所有线程；

锁对象执行wait()；和notify()；

互斥锁（Lock接口）

互斥锁：在一次执行中只能有一个线程持有锁对象；（JDK5新特性）

Lock接口+Condition接口；

2个以上线程通信，用到互斥锁；

生产者-消费者模型

基于等待/通知机制；

①仓库：共享数据

②生产者：线程同步；不满足条件，wait；满足条件，生产产品+notify；

③消费者：线程同步；不满足条件，wait；满足条件，消费产品+notify；

JVM

https://blog.csdn.net/wdjhzw/article/details/27720445

数据结构

线性表

顺序表

单链表

双向链表

循环链表

栈与队列

栈：先进后出

队列：先进先出

树

二叉树：有且仅有一个根节点，每个节点至多有两个子节点。

B+树：索引

红黑树（自平衡二叉查找树）

哈夫曼树（最优树）

查找

二分查找:

```

```

排序

冒泡排序:

```
// 冒泡排序 : 大的沉底，小的上浮，相邻比较
public void bubbleSort(int[] a) {
    for (int i = a.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

选择排序:

```
// 选择排序：缩小范围，两两比较
public void selectSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] > arr[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

插入排序:

```
// 插入排序：多次比较，一次插入
public void insertSort(int[] arr) {
    int j = 0;
    int key = 0; // 新牌
    for (int i = 1; i < arr.length; i++) {
        key = arr[i];// 抽取新牌
        for (j = i - 1; j >= 0 && key < arr[j]; j--) {
            arr[j + 1] = arr[j];
        }
        arr[j + 1] = key;// 已经j--，必须j+1
    }
}
```

快速排序:

```
/*
 * 快速排序：交换排序
 * pivot：枢轴指针；pivotkey：枢轴值，
 * low：低指针，  high：高指针，
 */
// 分区函数，返回枢轴指针：partition(数组名称,起始位下标,末尾位下标);
```

```java
public int partition(int[] arr, int low, int high) {
    int pivotkey = arr[low];// 设第一个元素为枢轴值
    while (low < high) {
        while (low < high && arr[high] >= pivotkey) {
            high--;
        }
        arr[low] = arr[high];
        while (low < high && arr[low] <= pivotkey) {
            low++;
        }
        arr[high] = arr[low];
    }
    arr[low] = pivotkey;// 将枢轴值 给到 可覆盖区
    return low;// 返回枢轴指针
}
// 分区递归：qSort(数组名称,起始位下标,末尾位下标);
public void qSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);// 枢轴指针
        qSort(arr, low, pivot - 1);// 前半部分
        qSort(arr, pivot + 1, high);// 后半部分
    }
}
// 快速排序
public void quickSort(int[] arr) {
    qSort(arr, 0, arr.length - 1);
}
```

数据库

分页SQL

Oracle分页语句

Oracle使用`ROWNUM`伪列实现分页：

```sql
select *
from (
  select "temp".*, ROWNUM "rn"
  from  <表/查询块> "temp"
  where ROWNUM <= currengPage * pageSize )
where "rn" > (currentPage-1) * pageSize
```

> currentPage：当前页数。 pageSize：每页显示的数据条数。

MySql分页语句

MySql使用`LIMIT`关键字实现分页：

```
select *
from <表/查询块>
limit (currentPage-1)*pageSize, pageSize
```

> currentPage：当前页数。 pageSize：每页显示的数据条数。

事务-TCL

事务：用于**保证数据完整性**。由一组DML语句组成，这组DML语句要么全部成功，要么全部失败。

事务特性：ACID

- **原子性（atomicity）**：一个事务是一个不可分割的工作单位，事务中的一组操作要么全做，要么全不做。
- **一致性（consistency）**：事务必须是使数据库从一个一致性状态变到另一个一致性状态。
- **隔离性（isolation）**：一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- **持久性（durability）**：指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

事务并发不一致：

- **幻读**：事务T1读取一条指定条件的语句，返回结果集。此时事务T2插入一行新记录并commit，恰好满足T1的条件。然后T1使用相同的条件再次查询，结果集中可以看到T2插入的记录，这条新纪录就是幻想。（事务T1查询，并行事务T2插入/删除部分数据并提交，事务T1再次查询，数据发生改变）
- **不可重复读**：事务T1读取一行记录，紧接着事务T2修改了T1刚刚读取的记录并commit，然后T1再次查询，发现与第一次读取的记录不同，这称为不可重复读。（事务T1读取一条记录，并行事务T2修改了该记录，事务T1再次查询，数据与第一次读的不同）
- **脏读**：一个事务读取了另一个未提交的并行事务写的数据。事务T1更新了一行记录，还未提交所做的修改，这个T2读取了更新后的数据，然后T1执行回滚操作，取消刚才的修改，所以T2所读取的行就无效，也就是脏数据。（事务T1更新但未提交，事务T2读取到更新后的数据，事务T1回滚，事务T2读取无效）
- **丢失更新**：当两个或多个事务选择同一数据，并且基于最初选定的值更新该数据时，会发生丢失更新问题。（事务T1读取数据，并行事务T2读取同一数据，事务T1更新并提交，事务T2更新并提交，事务T2覆盖事务T1提交结果）

四种隔离级别：

- 读未提交：一个事务可以读取另一个未提交事务的数据。
- 读已提交：一个事务必须等另一个事务提交后才能读取数据。（Oracle，SQL Server默认）
- 可重复读：在开始读取数据（事务开启）时，不再允许修改操作。（MySQL默认）
- 串行化：事务串行化顺序执行。效率低下，一般不使用。

> 读未提交 < 读已提交 < 可重复读 < 序列化

显式提交&隐式提交

- 显式提交：需要主动提交SQL语句对于数据库的修改，未提交之前可以rollback。如DML操作。
- 隐式提交：SQL语句执行结束自动提交，无法rollback。如DDL，DCL。

SQL优化

JavaWeb

Servlet生命周期

- Servlet 通过调用 **init ()** 方法进行初始化。
- Servlet 调用 **service()** 方法来处理客户端的请求。
- Servlet 通过调用 **destroy()** 方法终止（结束）。
- 最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

请求转发与重定向的区别

- 浏览器显示：重定向会改变URL地址，请求转发不会改变URL地址。
- 资源共享：重定向不可以进行资源共享，请求转发可以资源共享。
- 功能：重定向可以用URL绝对路径访问其他Web服务器的资源，而请求转发只能在一个Web应用程序内进行资源转发，即服务器内部的一种操作。
- 效率：重定向效率低，相当于再一次请求；请求转发效率相对较高，跳转仅发生在服务器端。

静态包含与动态包含的区别

（1）语法不同：

- 静态包含：JSP指令 - `<%@ include file=""%>`
- 动态包含：JSP行为 - `<jsp: include page=""%>`

（2）生成文件数量不同：

- 静态包含：两个文件二合一，整体编译，生成一个servlet和class文件。
- 动态包含：各个jsp分别转换，分别编译，生成多个servlet和class文件。

（3）包含时机不同：

- 静态包含：JSP翻译成Servlet阶段。
- 动态包含：执行class文件阶段，动态加入。

（4）静态包含在两个文件中不能有相同的变量，动态包含允许

（5）静态包含只能包含文件，动态包含还可以包含servlet输出的结果

（6）静态包含不能使用变量作为文件名，动态包含可以使用变量作为文件名

（7）动态包含文件发生变化，包含文件会感知变化

Cookie&Session

Cookie 机制：

- Cookie 机制采用在客户端保持 HTTP 状态信息的方案实现会话跟踪。
- Cookie 是指在浏览器访问 Web 服务器时，Web 服务器在 **HTTP 响应头**中附带的一个小文本文件。Cookie存储在客户端上，保留了各种跟踪信息。其中，会话Cookie保存在内存中，持久Cookie保存在磁盘中。
- Cookie 机制：①服务器脚本向客户端浏览器发送一组 Cookie；②客户端浏览器将这些信息存储在本地计算机上；③当下一次浏览器向 Web 服务器发送请求时，浏览器会将这些 Cookie 信息发送到服务器，服务器通过这些 Cookie 信息识别用户。
- Cookie 底层原理：Web 服务器在 HTTP 响应中增加 `Set-Cookie` 响应头字段将 Cookie 发送给浏览器；浏览器通过在 HTTP 请求中增加 Cookie 请求头字段将 Cookie 回传给服务器。

Session 机制：

- Session 机制采用在服务器端记录客户端会话状态的方案保持会话状态。
- Session 机制：①在客户端浏览器第一次访问服务器时，Web 服务器为客户端浏览器创建一个会话对象（session 对象），并生成一个对应的 SessionID，服务器把客户端会话状态记录在用户独享的 session 对象中。②在客户端再次访问时，服务器根据客户端携带的 SessionID 从 session 域中查找用户的信息。

Ajax

```
$.ajax({
    type: "GET",
    url: "test.json",
    data: {username:"scott", content:"tiger"},
    dataType: "json",
    success: function(data){
        // doSomething...
    }
});
```

跨域：CORS

```
response.setHeader("Access-Control-Allow-Origin","*");
```

> 支持 GET 和 POST 请求

SSM

MyBatis

MyBatis的ORM原理

Spring

Spring IOC

IOC/DI（控制反转/依赖注入），在传统开发中，使用new关键字创建对象，程序主动去创建对象，程序耦合度变高；而在Spring中，由Spring容器管理对象，主动负责控制对象的生命周期和对象间的关系，程序被动接受。即由IoC容器帮对象找相应的依赖对象并注入，而不是由对象主动去找。

Spring AOP

AOP（面向切面编程），将交叉业务逻辑织入到主业务逻辑中。底层是使用动态代理模式实现。

Spring AOP 事务管理

SpringMVC

SpringMVC 的加载流程：

1. 客户端发送请求到 DispatcherServlet（中央调度器）。
2. DispatcherServlet 查询 HandlerMapping（处理器映射器），找到处理请求的 Controller（处理器）。
3. DispatcherServlet 将请求转发给 Controller，Controller 处理请求，返回 ModelAndView（实体与视图）。
4. DispatcherServlet 查询 ViewResolver（视图解析器），找到 ModelAndView 指定的视图，渲染显示到客户端。

设计模式

单例模式（Singleton）

懒汉式：

```
public class Singletonif {
    // 静态实例化
    private static Singletonif s = null;
    // 私有化构造方法
    private Singletonif() {
    }
    // 公开提供静态获取实例的方法
    public static Singletonif getInstance() {
        if (s == null) {
            s = new Singletonif();
        }
        return s;
    }
}
```

饿汉式：

```
public class Singletonfinal {
    // 静态 final 实例化
    private static final Singletonfinal s = new Singletonfinal();
    // 私有化构造方法
    private Singletonfinal() {
    }
    // 公开提供静态获取实例的方法
    public static Singletonfinal getInstance() {
        return s;
    }
}
```

工厂模式

代理模式