# 第一部分：Java基础

## 基础语法

### ==和equals的区别？

==比较的是对象的内存地址，equals比较的是对象的值。

==可以比较基本数据类型，equals不能比较。

在Object类中==和equals一样，所以我们通常需要重写equals方法。

### 截取字符串的几个方法？

Str.Split() 按某个正则表达式切割字符串，返回一个字符串数组。

Str.subString() 不同的参数来截取字符串。

String sb = "bbbdsajjds";  
sb.substring(2);

将字符串从索引号为2开始截取，一直到字符串末尾。（索引值从0开始,含头不含尾）；

**2.2传入2个索引值**

String sb = "bbbdsajjds";  
sb.substring(2, 4);

从索引号2开始到索引好4结束（并且不包含索引4截取在内，也就是说实际截取的是2和3号字符）；

StringUtils

### String、StringBuffer、StringBuilder的区别？

String不可变得，对String类型进行改变的时候其实都是生成了一个新的String对象，然后将引用指向新的string对象。

StringBuffer对象是线程安全的可变字符序列。常用的方法是append和insert，append是添加到末端，insert是在指定的点添加字符。

StringBuilder是5.0新增的，和StringBufer差不错，但是是线程不安全的，大多数实现中，比StringBuffer要快。

### Java中的object常用方法

1、clone()  
2、equals()  
3、finalize()  
4、getclass()

5、hashcode()

6、notify()

7、notifyAll()

8、toString()

9、wait()

### 值类型和引用类型的区别？

直白点儿说：值类型就是现金，要用直接用；引用类型是存折，要用还得先去银行取现。

声明一个值类型变量，编译器会在栈上分配一个空间，这个空间对应着该值类型变量，空间里存储的就是该变量的值。引用类型的实例分配在堆上，新建一个引用类型实例，得到的变量值对应的是该实例的内存分配地址，这就像您的银行账号一样。

### 什么是值传递和引用传递？

对象被值传递，意味着传递了对象的一个副本。因此，就算是改变了对象副本，也不会影响源对象的值。

对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象所做的改变会反映到所有的对象上

### 构造函数详解？

作用：给类进行初始化，创建对象的时候会被调用。

### Static关键字的作用？

修饰变量：静态变量，可以通过类型直接访问，JVM只静态分配一次内存

修饰方法：该方法不能被覆盖，可以使用类名.方法名访问。

静态内部类：

静态导入：

静态代码块：

### Final关键字的作用？

答：(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

Static+final表示全局常量。

### Transient关键字的作用？

作用：在对象序列化的时候，有些变量不需要序列化，比如密码等，可以使用transient关键字来解决这个问题，transient修饰的变量不会被序列化。

### Foreach循环的底层原理是什么？

在平时Java程序中，应用比较多的就是对Collection集合类的foreach遍历，foreach之所以能工作，是因为这些集合类都实现了Iterable接口，该接口中定义了Iterator迭代器的产生方法，并且foreach就是通过Iterable接口在序列中进行移动。

### Volatile关键字的作用？

1. 保证程序的有序性。比如说new User对象，实际上在内存中有三个步骤，

1.为user指向的对象分配内存空间，并设置初始值为null。

2.执行构造函数初始化对象。

3.将对象指向分配的内存空间(分配内存空间之后user就是非null)。

在多线程的环境下，为了保证效率，JVM有时会对指令进行重排序，也就是说这三个步骤不是有序的，在多线程的情况下，如果发生指令重排，比如步骤2.3的顺序颠倒，很可能产生并发问题。比如在单例模式中，在创建单例对象User user = new User() 这一步时，如果线程一先执行了步骤3，在执行步骤2的时候，CPU资源被线程二抢走，这时线程二是判断user为非null的，但是user其实还没有被初始化。

|  |
| --- |
| class Singleton { private volatile static Singleton instance; public static Singleton getInstance() { if (instance == null) {  synchronized (Singleton.class) {  if (instance == null)  instance = new Singleton();  } }  return instance; }  public static void main(String[] args) { Singleton.getInstance(); }  } |

2.保证可见性。保证操作的数据是最新的。比如要修改a的值，首先需要将a的值从主存赋值过来+1，然后复制回主存，在多线程环境下可能在执行这些操作的时候会有其他线程来修改我正在操作的a的值，volatile可以保证在线程1操作a的值的时候，线程2拿到的值是线程1操作之后的值，也就是最新的数据，保证了可见性。

### Equals 和 HashCode的区别和联系。

两者都是比较对象是否相等的。一般equals的实现比较复杂，HashCode效率比较高。HashCode不相等的对象，equals一定不相等。

HashCode相等的，equals不一定相等。

所以在实际比较中，可以先用hashCode比较，如果hashCode相同，则在比较equals。

## 面相对象

### 谈谈你对面相对象的理解？

### 对于java多态的理解？

多态简单点说就是多种形态。Java实现多态有三个必要的条件：继承、重写、父类引用指向子类对象。比如汽车和飞机都是交通工具，他们有一个共同的方法run()，使用接口类型指向不用的实现类，就会有不同的结果。

### 方法的重写和重载的区别？

重载：方法名相同，参数列表不同，与返回值无关。重载是一个类中多态性的表现。

重写：发生在父类和子类之间，方法名相同，参数列表相同，返回值相同。

## 集合

### Set接口、list接口、map接口的区别？

set接口规定其实现无序，不可重复。

     HashSet（底层是HashMap） LinkedHashSet

list接口规定其实现有序，可重复。

     实现类有：LinkedList，ArrayList，Vector

map接口是双列集合的顶层接口，存储的key-value键值对

     HashMap HashTable StoredMap

### ArrayList的底层实现原理？

ArrayList实现了List<E>接口，list接口继承collection,collection接口继承iteable接口，表明arraylist可以使用for each循环

ArrayList实现了RandomAccess接口，这是一个标记接口，实现这个接口的在随机访问元素上有良好的性能，可以使用下标遍历。Linkedlist就没有实现这个接口，要使用迭代器遍历

ArrayList实现了Cloneable和Serializable两个标记接口，表明是可以克隆，可以序列化的。

然后ArrayList继承了AbstractList，在AbstractList里面有一个modCount属性，这个属性主要是由集合的迭代器使用，当使用iterator()或者listIterator生成一个迭代器时，这个迭代器会将list的modCount保存起来，在迭代过程中会去检查当前list的modCount是否发生变化，如果发生变化，马上抛出ConcurrentModificationException异常，这是fail-fast行为。

    ArrayList底层是动态数组，默认的长度是10，如果元素个数超过了数组的容量，那么就会扩容到原来的1.5倍+1，并将当前数组中的所有元素都复制到新数组中。

并且内部还维护了一个变量来记录实际的size大小，我们使用size获取长度的时候就是获取的这个值，并且也提供了释放多余空间的方法：trimToSize()

     LinkedList：

          底层是链表。

     ArrayList和LinkedList的区别：

          ArrayList查询快，增删慢。

          LinkedList查询慢，增删快。

### ArrayList和Vector的区别

        这两个类都实现了List接口（List接口继承了Collection接口），他们都是有序集合，即存储在这两个集合中的元素的位置都是有顺序的，相当于一种动态的数组，我们以后可以按位置索引号取出某个元素，并且其中的数据是允许重复的，这是与HashSet之类的集合的最大不同处，HashSet之类的集合不可以按索引号去检索其中的元素，也不允许有重复的元素。

         ArrayList与Vector的区别主要包括两个方面：.  
（1）同步性：

       Vector是线程安全的，也就是说是它的方法之间是线程同步的，而ArrayList是线程序不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用ArrayList，因为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用Vector，因为不需要我们自己再去考虑和编写线程安全的代码。

（2）数据增长：

       ArrayList与Vector都有一个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加ArrayList与Vector的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要取得一定的平衡。Vector默认增长为原来两倍，而ArrayList的增长策略在文档中没有明确规定（从源代码看到的是增长为原来的1.5倍+1）。ArrayList与Vector都可以设置初始的空间大小，Vector还可以设置增长的空间大小，而ArrayList没有提供设置增长空间的方法。

    总结：即Vector增长原来的二倍，ArrayList增加原来的1.5倍+1。

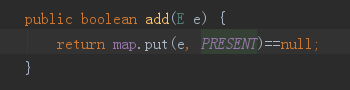
### ArrayList和Array和有什么区别？什么时候应该使用Array而不是ArrayList？

* Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。
* Array大小是固定的，ArrayList的大小是动态变化的。
* ArrayList提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。
* 对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

### HashSet底层原理？

HashSet实现set接口，内部维护一个HashMap实例变量，不保证顺序，允许null元素。线程不安全

public HashSet() {  
 map = new HashMap<>();  
}



HashSet add的时候实际上是把对象当做key存进了HashMap中，value是：

private static final Object *PRESENT* = new Object();

### HashMap和HashTable的底层实现原理，和区别

HashMap的底层是基于数组和链表以及红黑树(1.8+)实现的，存元素的时候，会根据key来计算hash值，然后放到一个Entry[]数组中，Entry[]对象包含了k,v，如果hash值相同形成了Hash冲突，就会链表的形式进行存储。如果链表的长度超过8，则会转为红黑树

HashMap有一个比较关键的属性就是loadFactor加载因子，它表示的是Hash表中的填满程度。

如果加载因子越大，填满的元素越多，空间利用率越高，但是冲突的机会加大了，链表的长度越来越长，查找效率也会降低。

加载因子越小，填满的元素越少，冲突的机会越小，查找成本低，但是空间浪费多。

所以如果想提高查询速度，可以将加载因子调低点，反之可以调高点，一般取默认值0.75就可以了。

HashTable底层也是数组的链表，解决冲突的问题和HashMap一样。

区别：

1. 继承不同，HashMap继承Abstractmap、HashTable继承Dictionary。
2. HashMap默认容量是16，HashTable默认容量是11.
3. Hashmap key和value都允许是null值,不过key只能有一个null值，HashTable都不允许为null
4. 对hash值的计算有些不一样。
5. HashTable线程安全，Hashmap线程不安全。

### HashMap的get()方法的原理。时间复杂度是多少？

1. 先调用Key的hashcode方法拿到对象的hash值。
2. 然后用hash值对第一维数组的长度进行取模，得到数组的下标。这个数组下标所在的元素就是第二维链表的表头。
3. 然后遍历这个链表，使用Key的equals同链表元素进行比较，匹配成功即返回链表元素里存放的值。

最好的情况是O(1)，如果发生hash冲突，最坏的情况是O(n)

### ConcurrentHashMap是什么？原理是什么？

ConcurrentHashMap是JDK1.5从concurrent包中引入的。

它引入了“分段锁”的概念。就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中或从哪个Segment中取。每个segment都是线程安全的，默认segment的容量是16，效率提升16倍。

并发情况下，在往ConcurrentHashMap中存元素时，如果key1和key2 计算的hash值相同，key3和前面的不同。在存的时候，key1存的时候，key2会阻塞，因为他们会放到一个segment去，而key3就不会阻塞。

### Arrays和Collections对于sort的不同实现原理

1、Arrays.sort()  
该算法是一个经过调优的快速排序，此算法在很多数据集上提供N\*log(N)的性能，这导致其他快速排序会降低  
二次型性能。  
2、Collections.sort()  
该算法是一个经过修改的合并排序算法（其中，如果低子列表中的最高元素效益高子列表中的最低元素，则忽略  
合并）。此算法可提供保证的N\*log(N)的性能，此实现将指定列表转储到一个数组中，然后再对数组进行排

序，在重置数组中相应位置处每个元素的列表上进行迭代。这避免了由于试图原地对链接列表进行排序而产生的 n2log(n)性能。

### Iterator和ListIterator的区别是什么？

* Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。
* Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。
* ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

### 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

Iterator的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。java.util包下面的所有的集合类都是快速失败的，**而java.util.concurrent包下面的所有的类都是安全失败的**。快速失败的迭代器会抛出ConcurrentModificationException异常，而安全失败的迭代器永远不会抛出这样的异常。

比如：

ArrayList继承了AbstractList，在AbstractList里面有一个modCount属性，这个属性主要是由集合的迭代器使用，当使用iterator()或者listIterator生成一个迭代器时，这个迭代器会将list的modCount保存起来，在迭代过程中会去检查当前list的modCount是否发生变化，如果发生变化，马上抛出ConcurrentModificationException异常，这是fail-fast行为

安全失败例子：

比如在遍历concurentHashMap时修改其结构，便不会抛出异常

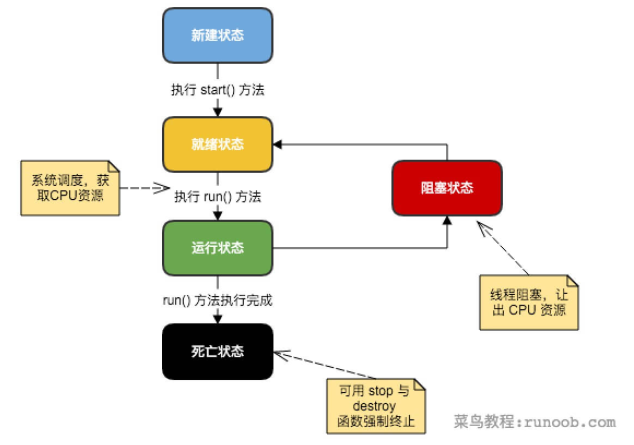
fail-safe机制有两个问题

（1）需要复制集合，产生大量的无效对象，开销大

（2）无法保证读取的数据是目前原始数据结构中的数据。

## 多线程

### **线程的基本状态以及状态之间的转化关系？**



新建态—就绪态—运行态—阻塞态—死亡态

|  |
| --- |
| **新建状态:** 使用 new 关键字和 Thread 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 start() 这个线程。  **就绪状态:** |

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的

线程处于就绪队列中，要等待JVM里线程调度器的调度。

**运行状态:**  
如果就绪状态的线程获取 CPU 资源，就可以执行 run()，此时线程便处于  
运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状

态和死亡状态。

**阻塞状态:**  
如果一个线程执行了sleep（睡眠）、suspend（挂起），失去所占  
用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得  
设备资源后可以重新进入就绪状态。可以分为三种：

**等待阻塞**：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。

**同步阻塞**：线程在获取 synchronized 同步锁失败(因为同步锁被其

他线程占用)。

**其他阻塞**：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。

**死亡状态:**  
一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到

终止状态。

### Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?

 答：sleep()方法（休眠），会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

### 线程的sleep()方法和yield()方法有什么区别？

sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；   
② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；   
③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；   
④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

### 当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（**注意不是等待池哦**）中等待对象的锁。

### 请说出与线程同步以及线程调度相关的方法。

- wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；   
- sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；   
- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；   
- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

### **编写多线程程序有几种实现方式？**

答：Java 5以前实现多线程有两种实现方法：一种是继承Thread类，重写run()方法；另一种是实现Runnable接口，重写run()方法，然后根据Thread(Runnable)这个构造方法来创建线程。两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

补充：Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值，代码如下所示：

|  |
| --- |
| class MyTask implements Callable<Integer> {     private int upperBounds;     public MyTask(int upperBounds) {         this.upperBounds = upperBounds;     }     @Override     public Integer call() throws Exception {         int sum = 0;          for(int i = 1; i <= upperBounds; i++) {             sum += i;         }         return sum;     } } class Test {     public static void main(String[] args) throws Exception {         List<Future<Integer>> list = new ArrayList<>();         ExecutorService service = Executors.newFixedThreadPool(10);         for(int i = 0; i < 10; i++) {             list.add(service.submit(new MyTask((int) (Math.random() \* 100))));         }         int sum = 0;         for(Future<Integer> future : list) {             // while(!future.isDone()) ;             sum += future.get();         }         System.out.println(sum);     } } |

### **Runnable接口和Callable接口的区别**

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为**多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性**，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，真的是非常有用。

### **synchronized关键字的用法？**

答：synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。

### **举例说明同步和异步。**

答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

### **启动一个线程是调用run()还是start()方法？**

答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

### **什么是线程池（thread pool）？**

答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。   
Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：   
- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。   
- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。   
- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。   
- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。   
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

第60题的例子中演示了通过Executors工具类创建线程池并使用线程池执行线程的代码。如果希望在服务器上使用线程池，强烈建议使用newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

注：

阿里规范中，禁止使用Executors创建线程池，而是推荐使用ThreadPoolExecutors的方式.因为Executors可能造成OOM

|  |
| --- |
|  |

### **简述synchronized 和java.util.concurrent.locks.Lock的异同？**

答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；synchronized在并发量高的时候，性能会严重下降，这个时候可以使用Lock作为替代方案。两者主要的主要不同点：

（1）Lock是一个接口，是JDK层面的实现；而synchronized是Java中的关键字，  
是Java的内置特性，是JVM层面的实现；  
(2) synchronized 在发生异常时，会自动释放线程占有的锁，因此不会导致死锁  
现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能  
造成死锁现象，因此使用Lock时需要在finally块中释放锁；  
(3) Lock 可以让等待锁的线程响应中断，而使用synchronized时，等待的线程会  
一直等待下去，不能够响应中断；  
(4) 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到；  
(5) Lock可以提高多个线程进行读操作的效率

### 死锁的产生？怎么防止死锁？

两个线程相互等待就会产生死锁。比如线程a持有锁m，需要锁n。线程B持有锁n，需要锁m，就会产生死锁。

|  |
| --- |
| public class DeadLock  {  private final Object left = new Object();  private final Object right = new Object();    public void leftRight() throws Exception  {  synchronized (left)  {  Thread.sleep(2000);  synchronized (right)  {  System.out.println("leftRight end!");  }  }  }    public void rightLeft() throws Exception  {  synchronized (right)  {  Thread.sleep(2000);  synchronized (left)  {  System.out.println("rightLeft end!");  }  }  }  } |

死锁的解决：

保证只有一个锁，不过在多线程情况下，会严重影响效率。

指定获取锁的顺序，所有的线程有以同样的顺序加锁和释放锁。

可以使用Lock类中的tryLock（）方法去尝试获取锁，这个方法可以执行一个超时时限，在等待超过该时限之后便会返回一个失败信息。

### 多线程有什么用？

发挥多核CPU的优势、防止阻塞。

### **现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？**

Join()方法。

Example:

|  |
| --- |
| **public class** TestJoinMethod {  **public static void** main(String[] args) **throws** InterruptedException {  Thread thread = **new** Thread(){  **public void** run() {  System.***out***.println(**"Thread1 is running"**);  **try** {  *sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }**finally** {  System.***out***.println(**"Thread1 is stoped"**);  }  **super**.run();  }  };  Thread thread2 = **new** Thread(){  **public void** run() {  System.***out***.println(**"Thread2 is running"**);  **try** {  *sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }**finally** {  System.***out***.println(**"Thread2 is stoped"**);  }  **super**.run();  }  };  Thread thread3 = **new** Thread(){  **public void** run() {  System.***out***.println(**"Thread3 is running"**);  **try** {  *sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }**finally** {  System.***out***.println(**"Thread3 is stoped"**);  }  **super**.run();  }  };   thread.start();  thread.join();  thread2.start();  thread2.join();  thread3.start();  thread3.join();  } } |

### 使用Java实现阻塞队列

可以使用wait()和notify()、await()和signal()以及BlockingQueue实现。

阻塞队列是Java5线程新特征中的内容，Java定义了阻塞队列的接口java.util.concurrent.BlockingQueue，阻塞队列的概念是，一个指定长度的队列，如果队列满了，添加新元素的操作会被阻塞等待，直到有空位为止。同样，当队列为空时候，请求队列元素的操作同样会阻塞等待，直到有可用元素为止。

简单的APi：



插入操作是指向队列中添加一个元素，至于元素存放的位置与具体队列的实现有关。移除操作将会移除队列的头部元素，并将这个移除的元素作为返回值反馈给调用者。检查操作是指返回队列的头元素给调用者，队列不对这个头元素进行删除处理。

       抛出异常形式的操作，在队列已满的情况下，调用add方法将会抛出IllegalStateException异常。如果调用remove方法时，队列已经为空，则抛出一个NoSuchElementException异常。（实际上，remove方法还可以附带一个参数，用来删除队列中的指定元素，如果这个元素不存在，也会抛出NoSuchElementException异常）。如果调用element检查头元素，队列为空时，将会抛出NoSuchElementException异常。

       特殊值操作与抛出异常不同，在出错的时候，返回一个空指针，而不会抛出异常。

       阻塞形式的操作，调用put方法时，如果队列已满，则调用线程阻塞等待其它线程从队列中取出元素。调用take方法时，如果阻塞队列已经为空，则调用线程阻塞等待其它线程向队列添加新元素。

       超时形式操作，在阻塞的基础上添加一个超时限制，如果等待时间超过指定值，抛出InterruptedException。

       阻塞队列实现了Queue接口，而Queue接口实现了Collection接口，因此BlockingQueue也提供了remove(e)操作，即从队列中移除任意指定元素，但是这个操作往往不会按预期那样高效的执行，所以应当尽量少的使用这种操作。

       阻塞队列与并发队列（例如ConcurrentLinkQueue）都是线程安全的，但使用的场合不同。

|  |
| --- |
| import java.util.concurrent.BlockingQueue;  import java.util.concurrent.ArrayBlockingQueue;   /\*\*  \* Java线程：新特征-阻塞队列  \*  \* @author leizhimin 2009-11-5 14:59:15  \*/  public class Test {          public static void main(String[] args) throws InterruptedException {                  BlockingQueue bqueue = new ArrayBlockingQueue(20);                  for (int i = 0; i < 30; i++) {                          //将指定元素添加到此队列中，如果没有可用空间，将一直等待（如果有必要）。                          bqueue.put(i);                          System.out.println("向阻塞队列中添加了元素:" + i);                  }                  System.out.println("程序到此运行结束，即将退出----");          }  } |

### **用Java写代码来解决生产者——消费者问题**

什么是生产者、消费者问题？

比如说我现在有一个仓库，仓库的容量是30，对于生产者而言，现在要生产20件商品，但是仓库已经有20件商品，那么生产的这20件商品会放不下，这个时候就需要让这个生产线程阻塞。对于消费者而言，现在要消费10件商品，但是仓库只有5件商品，那么这个线程也会被阻塞。

在代码中，检测仓库的状态，满足条件则唤醒线程。

wait()和notify()实现

Await()和signal()实现

BlockingQuene实现

### Wait和await()的区别？

Wait以及nofity是Object类中的方法，

Await()和signal()是1.5以后Lock接口中的方法

Wait需要在syn()中使用，而await()是在lock.lock()…lock.unLock()中间使用。

### CountDownLatch用法

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

CountDownLatch类只提供了一个构造器：

|  |  |
| --- | --- |
|  | public CountDownLatch(int count) {  };  //参数count为计数值 |

然后下面这3个方法是CountDownLatch类中最重要的方法：

|  |  |
| --- | --- |
|  | public void await() throws InterruptedException { };   //调用await()方法的线程会被挂起，它会等待直到count值为0才继续执行  public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };  //和await()类似，只不过等待一定的时间后count值还没变为0的话就会继续执行  public void countDown() { };  //将count值减1 |

### 什么是竞态条件？ 举个例子说明

当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在竞态条件。

导致竞态条件发生的代码区称作临界区。

在临界区中使用适当的同步就可以避免竞态条件。

临界区实现方法有两种，一种是用synchronized，一种是用Lock显式锁实现。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Counter {      protected long count = 0;      public void add(long value) {          this.count = this.count + value;      }  } |

 观察线程A和B交错执行会发生什么，两个线程分别加了2和3到count变量上，两个线程执行结束后count变量的值应该等于5。然而由于两个线程是交叉执行的，两个线程从内存中读出的初始值都是0。然后各自加了2和3，并分别写回内存。最终的值并不是期望的5，而是最后写回内存的那个线程的值，上面例子中最后写回内存的是线程A，但实际中也可能是线程B。如果没有采用合适的同步机制，线程间的交叉执行情况就无法预料。  
add()方法就是一个临界区,它会产生竞态条件。

## IO

### 阻塞IO、非阻塞IO、异步IO、非异步IO的区别？

举个简单的例子，和女朋友出去吃饭，饭店人很多，需要排队，这个时候就有几种处理方式：

第一种，点完餐之后等待饭做好。这就是典型的阻塞。

第二种，点完餐之后去逛街了，并且不断的问服务员饭做好了没有，这就是非阻塞。

第三种，不在饭店吃了，点外卖，外卖点了之后做自己的事情，等别人送货上门，这就是异步。

程序和内核交互的时候，以read为例，内核有一个等待数据可读的过程，阻塞IO会在这一过程等待，非阻塞不会等待。

同步过程中进程触发IO操作并等待或者轮询的去查看IO操作是否完成。异步过程中进程触发IO操作以后，直接返回，做自己的事情，IO交给内核来处理，完成后内核通知进程IO完成。

### IO和NIO的区别

|  |
| --- |
| **IO**                **NIO**  面向流            面向缓冲  阻塞IO           非阻塞IO  无 选择器 |

**1.面向流与面向缓冲**

Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

**2.阻塞与非阻塞IO**

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。 非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。 线程通常将非阻塞IO的空闲时间用于在其它通道上执行IO操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

**3.选择器（**Selectors**）**

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### NIO的三大核心是什么？

* Channels

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中

* Buffers

Buffer的三个属性

capacity：含义与模式无关；Buffer的一个固定的大小值；Buffer满了需要将其清空才能再写；

* + ByteBuffer.allocate(48)；该buffer的capacity为48byte
  + CharBuffer.allocate(1024);该buffer的capacity为1024个char

position：含义取决于Buffer处在读模式还是写模式（初始值为0，写或者读操作的当前位置）

* + 写数据时，初始的position值为0；其值最大可为capacity-1
  + 将Buffer从写模式切换到读模式，position会被重置为0

limit：含义取决于Buffer处在读模式还是写模式（写limit=capacity；读limit等于最多可以读取到的数据）

* + 写模式下，limit等于Buffer的capacity
  + 切换Buffer到读模式时， limit表示你最多能读到多少数据；
* Selectors

 Selector允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。例如，在一个聊天服务器中。

 要使用Selector，得向Selector注册Channel，然后调用它的select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

### Java的序列化和反序列化？

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

为什么要序列化？

主要是为了网络之间数据的传输，将Java对象序列化为持久化文件，避免了数据的丢失。

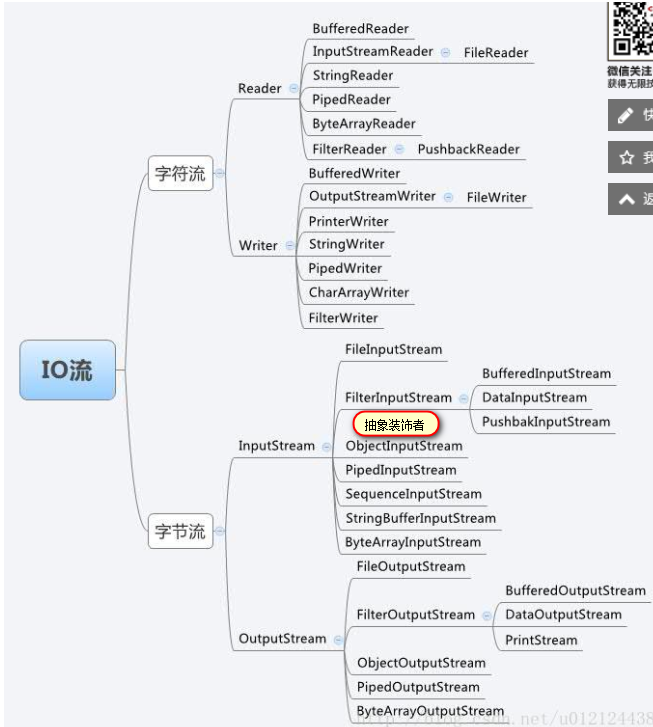
### **Java中如何实现序列化，有什么意义？**

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。   
要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第29题）

### **Java中有几种类型的流？**

答：字节流和字符流。字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是两种对称性（输入和输出的对称性，字节和字符的对称性）；二是两种设计模式（适配器模式和装饰模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

### 说说IO框架？



### **面试题** - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

import java.io.FileInputStream;  
import java.io.FileOutputStream;import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
import java.nio.ByteBuffer;  
import java.nio.channels.FileChannel;  
public final class MyUtil {  
    private MyUtil() {  
        throw new AssertionError();  
    }  
    public static void fileCopy(String source, String target) throws IOException {  
        try (InputStream in = new FileInputStream(source)) {  
            try (OutputStream out = new FileOutputStream(target)) {  
                byte[] buffer = new byte[4096];  
                int bytesToRead;  
                while((bytesToRead = in.read(buffer)) != -1) {  
                    out.write(buffer, 0, bytesToRead);  
                }  
            }  
        }  
    }  
    public static void fileCopyNIO(String source, String target) throwsIOException {  
        try (FileInputStream in = new FileInputStream(source)) {  
            try (FileOutputStream out = new FileOutputStream(target)) {  
                FileChannel inChannel = in.getChannel();  
                FileChannel outChannel = out.getChannel();  
                ByteBuffer buffer = ByteBuffer.allocate(4096);  
                while(inChannel.read(buffer) != -1) {  
                    buffer.flip();  
                    outChannel.write(buffer);  
                    buffer.clear();  
                }  
            }  
        }  
    }  
}

### **写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。**

|  |
| --- |
| import java.io.BufferedReader; import java.io.FileReader; public final class MyUtil {     // 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)     private MyUtil() {         throw new AssertionError();     }      /\*\*     \* 统计给定文件中给定字符串的出现次数     \*     \* @param filename  文件名     \* @param word 字符串     \* @return 字符串在文件中出现的次数     \*/     public static int countWordInFile(String filename, String word) {         int counter = 0;         try (FileReader fr = new FileReader(filename)) {             try (BufferedReader br = new BufferedReader(fr)) {                 String line = null;                 while ((line = br.readLine()) != null) {                     int index = -1;                     while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {                         counter++;                         line = line.substring(index + word.length());                     }                 }             }         } catch (Exception ex) {             ex.printStackTrace();         }         return counter;     } } |

### 如何用Java代码列出一个目录下所有的文件？

如果只要求列出当前文件夹下的文件，代码如下所示：

import java.io.File;  
class Test12 {  
    public static void main(String[] args) {  
        File f = new File("/Users/Hao/Downloads");  
        for(File temp : f.listFiles()) {  
            if(temp.isFile()) {  
                System.out.println(temp.getName());  
            }  
        }  
    }  
}

如果需要对文件夹继续展开，代码如下所示：

import java.io.File;  
class Test12 {  
    public static void main(String[] args) {  
        showDirectory(new File("/Users/Hao/Downloads"));  
    }  
    public static void showDirectory(File f) {  
        \_walkDirectory(f, 0);  
    }  
    private static void \_walkDirectory(File f, int level) {  
        if(f.isDirectory()) {  
            for(File temp : f.listFiles()) {  
                \_walkDirectory(temp, level + 1);  
            }  
        }  
        else {  
            for(int i = 0; i < level - 1; i++) {  
                System.out.print("\t");  
            }  
            System.out.println(f.getName());  
        }  
    }  
}

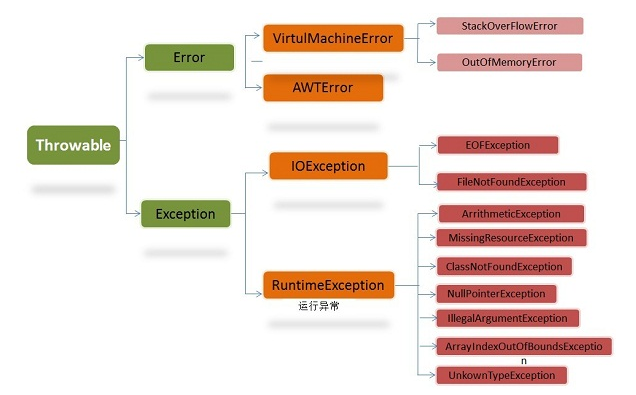
在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

class ShowFileTest {  
    public static void main(String[] args) throws IOException {  
        Path initPath = Paths.get("/Users/Hao/Downloads");  
        Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {  
            @Override  
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)   
                    throws IOException {  
                System.out.println(file.getFileName().toString());  
                return FileVisitResult.CONTINUE;  
            }  
        });  
    }  
}

## JDK

## 异常

### Java异常的体系结构：



首先说明一点，java中的Exception类的子类不仅仅只是像上图所示只包含IOException和RuntimeException这两大类，事实上Exception的子类很多很多，主要可概括为：运行时异常与非运行时异常。

Java异常的超类是Thorwable类，两个直接子类为Error和Exception。异常又分为运行时异常和非运行时异常。

Error与Exception  
    Error是程序无法处理的错误，它是由JVM产生和抛出的，比如OutOfMemoryError、ThreadDeath等。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。  
    Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。程序中应当尽可能去处理这些异常。

运行时异常和非运行时异常  
    运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。  
    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。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

## 反射

### 内省和反射的区别？

内省是基于反射的，它只能获得类里面的属性以及属性对应的get/set方法。

反射可以获得所有的属性，所有方法

### Java的反射机制

#### 什么是反射？

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

#### 反射入口

反射的入口是Class类，获取Class对象的方式有三种：

1. obj.getClass()
2. Object.class
3. Class.forName(“全类名”)

#### 反射API

1.获取构造方法

* 返回一个构造方法
  + public Constructor<T> getConstructor(Class<?>... parameterTypes) 获取public修饰, 指定参数类型所对应的构造方法
  + public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes) 获取指定参数类型所对应的构造方法(包含私有的)
* 返回多个构造方法
  + public Constructor<?>[] getConstructors() 获取所有的public 修饰的构造方法
  + public Constructor<?>[] getDeclaredConstructors() 获取所有的构造方法(包含私有的)

1. 反射获取成员变量

返回一个成员变量

* 1. public Field getField(String name) 获取指定的 public修饰的变量
  2. public Field getDeclaredField(String name) 获取指定的任意变量

返回多个成员变量

* 1. public Field[] getFields() 获取所有public 修饰的变量
  2. public Field[] getDeclaredFields() 获取所有的 变量 (包含私有)

1. 反射获取成员方法

返回获取一个方法：

* 1. public Method getMethod(String name, Class<?>... parameterTypes)

获取public 修饰的方法

* 1. public Method getDeclaredMethod(String name, Class<?>... parameterTypes)

获取任意的方法，包含私有的

参数1: name 要查找的方法名称； 参数2： parameterTypes 该方法的参数类型

返回获取多个方法：

* 1. public Method[] getMethods() 获取本类与父类中所有public 修饰的方法

public Method[] getDeclaredMethods() 获取本类中所有的方法(包含私有的)

#### 泛型擦除

思考，将已存在的ArrayList<Integer>集合中添加一个字符串数据，如何实现呢？

其实程序编译后产生的.class文件中是没有泛型约束的，这种现象我们称为泛型的擦除。那么，我们可以通过反射技术，来完成向有泛型约束的集合中，添加任意类型的元素

**public** **class** ReflectTest {

**public** **static** **void** main(String[] args) **throws** ClassNotFoundException, NoSuchMethodException, SecurityException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {

ArrayList<Integer> list = **new** ArrayList<Integer>();

//添加元素到集合

list.add(**new** Integer(30));

list.add(**new** Integer("12345"));

list.add(123);

//list.add("哈哈");//因为有泛型类型的约束

System.***out***.println(list);

//通过反射技术，实现添加任意类型的元素

//1, 获取字节码文件对象

//Class c = list.getClass();

//Class c = ArrayList.class;

Class c = Class.*forName*("java.util.ArrayList");

//2, 找到add()方法

// public boolean add(E e)

Method addMethod = c.getMethod("add", Object.**class**);

//3， 执行add()方法

addMethod.invoke(list, "哈哈");// list.add("哈哈");

System.***out***.println(list);

}

}

#### 反射配置文件

* 通过反射配置文件，运行配置文件中指定类的对应方法

读取Peoperties.txt文件中的数据，通过反射技术，来完成Person对象的创建

Peoperties.txt文件内容如下：

className=cn.itcast\_01\_Reflect.Person

methodName=method5

* 读取配置文件，调用指定类中的对应方法

**public** **class** ReflectTest2 {

**public** **static** **void** main(String[] args)

**throws** FileNotFoundException, IOException, ClassNotFoundException, NoSuchMethodException, SecurityException,

InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {

// 通过Properties集合从文件中读取数据

Properties prop = **new** Properties();

// 读取文件中的数据到集合中

prop.load(**new** FileInputStream("properties.txt"));

// 获取键所对应的值

String className = prop.getProperty("className");

System.***out***.println(className);

// 1，获取Person.class 字节码文件对象

Class c = Class.*forName*(className);

// 2，获取构造方法

// public Person(String name, int age, String address)

Constructor con = c.getConstructor(String.**class**, **int**.**class**, String.**class**);

// 3,创建对象

Object obj = con.newInstance("小明", 20, "中国");

System.***out***.println(obj);

// 4，获取指定的方法

// private void method5(){}

String methodName = prop.getProperty("methodName");

Method m5 = c.getDeclaredMethod(methodName, **null**);

// 5,开启暴力访问

m5.setAccessible(**true**);

// 6，执行找到的方法

m5.invoke(obj, **null**);

}

}

## 注解

## 网络编程

### UDP协议和TCP协议的区别？

UDP是无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用UDP协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输例如视频会议都使用UDP协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

TCP协议是面向连接的通信协议，即在传输数据前先在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。在TCP连接中必须要明确客户端与服务器端，由客户端向服务端发出连接请求，每次连接的创建都需要经过“三次握手”。第一次握手，客户端向服务器端发出连接请求，等待服务器确认，第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求，第三次握手，客户端再次向服务器端发送确认信息，确认连接。

由于TCP协议的面向连接特性，它可以保证传输数据的安全性，所以是一个被广泛采用的协议，例如在下载文件时，如果数据接收不完整，将会导致文件数据丢失而不能被打开，因此，下载文件时必须采用TCP协议。

## 泛型

# 第二部分：设计模式

## 手写单例模式？

|  |
| --- |
| **public** **class** Singleton {  /\*\*  \* 使用静态内部类实现单例模式  \*/    //私有化构造器  **private** Singleton(){    }    //静态内部类*在使用的时候才会加载，并且只会加载一次*    **private** **static** **class** singletonHolder{  **private** **static** **final** Singleton ***instance***;  **static**{  ***instance*** = **new** Singleton();  }  }    **public** **static** Singleton getInstance(){  **return** singletonHolder.***instance***;  }  }  **public** **class** Singleton2 {  /\*\*  \* 双重锁机制+volatile关键字实现单例  \*/  //声明单例对象  **private** **static** **volatile** Singleton2 *instance*;    //私有化构造器  **private** Singleton2(){    }  //双重检查加锁  **public** **static** Singleton2 getInstance(){  **if** (*instance* == **null**) {  **synchronized** (Singleton2.**class**) {  **if** (*instance* == **null**) {  *instance*=**new** Singleton2();  }  }  }  **return** *instance*;  }  } |

Volatile关键字的作用：

New Singleton2()这个步骤并不是原子操作，底层实际上有三个无序的操作：

*// 1. 为 Singleton 对象分配内存* memory = allocate();

*// 2. 注意现在 instance 是非空的，但还没初始化* instance = memory; *// 3. 调用 Singleton 的构造函数，传递 instance* ctorSingleton(instance);

而volatile关键字保证了程序的有序性以及可见性。

## 工厂模式

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

面试题，我们为什么用IOC而不用工厂模式创建对象？

本质上还是因为IOC是通过反射机制来实现的。当我们的需求出现变动时，工厂模式会需要进行相应的变化。但是IOC的反射机制允许我们不重新编译代码，因为它的对象都是动态生成的。

## 抽象工厂模式

抽象工厂模式（Abstract Factory Pattern）是围绕一个超级工厂创建其他工厂。该超级工厂又称为其他工厂的工厂。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

在抽象工厂模式中，接口是负责创建一个相关对象的工厂，不需要显式指定它们的类。每个生成的工厂都能按照工厂模式提供对象

## 建造者模式

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

例如：StringBuilder

## 原型模式

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

## 适配器模式

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。举个真实的例子，读卡器是作为内存卡和笔记本之间的适配器。您将内存卡插入读卡器，再将读卡器插入笔记本，这样就可以通过笔记本来读取内存卡。

我们通过下面的实例来演示适配器模式的使用。其中，音频播放器设备只能播放 mp3 文件，通过使用一个更高级的音频播放器来播放 vlc 和 mp4 文件

## 桥接模式

桥接（Bridge）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

我们通过下面的实例来演示桥接模式（Bridge Pattern）的用法。其中，可以使用相同的抽象类方法但是不同的桥接实现类，来画出不同颜色的圆

## 装饰者模式

装饰器模式（Decorator Pattern）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。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

我们通过下面的实例来演示装饰器模式的用法。其中，我们将把一个形状装饰上不同的颜色，同时又不改变形状类。

例子：  
 Java中的I/O  
 IO中，FileInputStream是一个被装饰的具体组件。BufferedInputStream是一个具体的装饰者，它加入两种行为：利用缓冲输入  
 来改进功能；用一个readLine()方法来增强接口。而LineNubmerInputStream又对BufferedInputStream进行了封装。  
 BufferedInputStream和LineNubmerInputStream都扩展自FilterInputStream，而FilterInputStream是一个抽象的装饰类。

## State设计模式理解

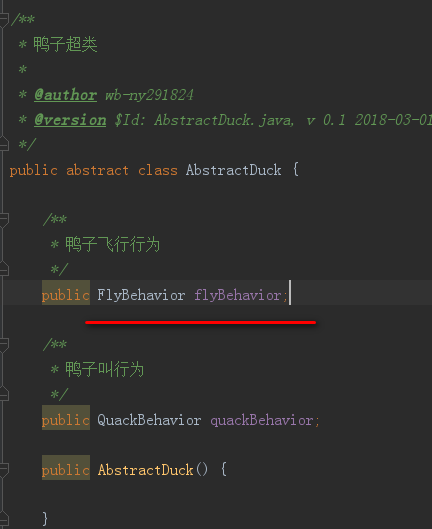
就像activiti工作流里面一个个球进行判断，每一个球就是一个状态，可以进行条件判断，审计等等操作。

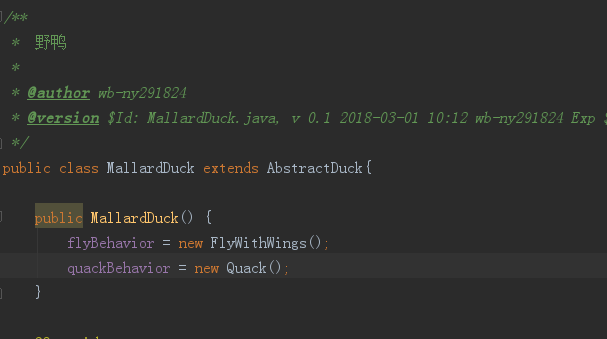
## 策略模式

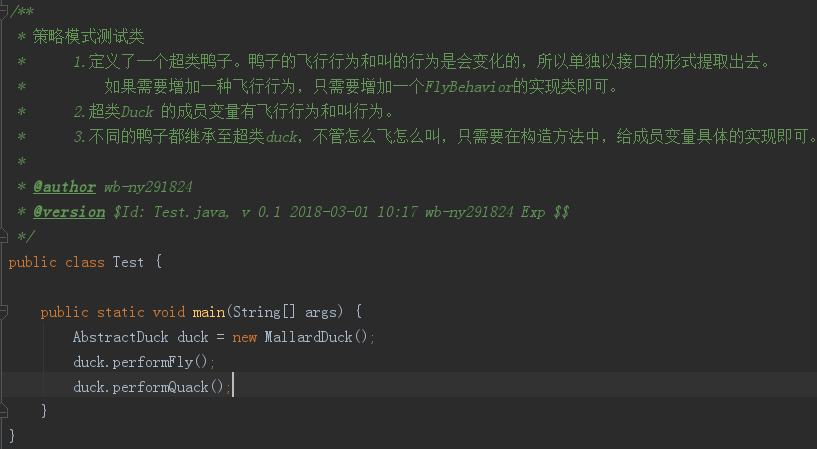
*策略模式定义了算法族，分别封装起来，让他们之间可以互相替换。此模式让算法的变化独立于使用算法的客户。*

比如：

鸭子，有一种飞行行为。有的鸭子会飞，有的不会。这个时候就可以使用策略设计模式，将飞行这个变化的行为抽取出来。让不同的鸭子可以有不同的行为。

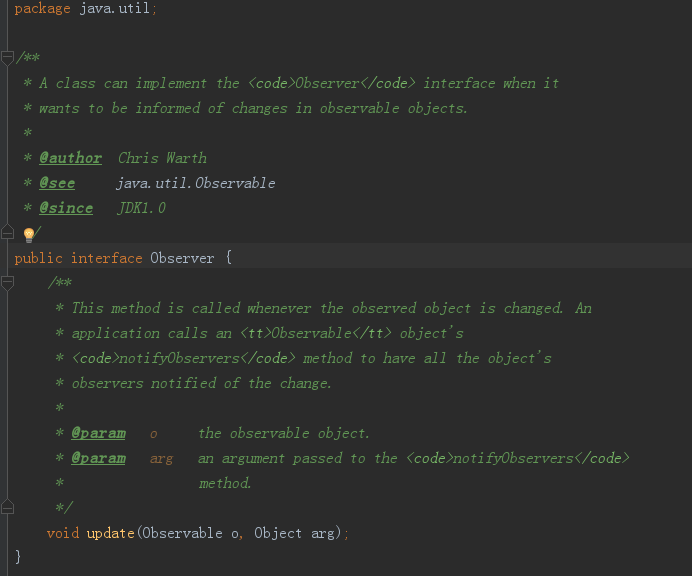






## 观察者模式

定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，他的所有依赖者都会受到通知并自动更新。  
主题(subject)+观察者(observer)=观察者模式  
  
观察者模式的两种形态：  
 a.主题主动将信息推送给所有观察者。  
 b.观察者主动索取数据  
Java内置了观察者模式，Observer接口支持这两种模式。  
JavaBeans和Swing使用的观察者模式



## 构建模式

在mybatis中，构建SqlSessionFactory的时候：

1. 通过XMlConfigBuilder解析配置的XML文件。读出配置参数，并保存在Configuration类
2. 使用Configuration对象去创建SqlSessionFactory。

这里就使用了构建(Builder)模式，对于复杂对象而言，直接使用构造方法构建是有困难的，会导致大量的逻辑在构造方法中，由于对象的复杂性，在构建的时候，我们更希望一步步有秩序的来构建它，从而降低复杂性。这个时候用一个参数类总领全局，然后分布构建，就可以创建一个复杂对象。

## 设计模式的六大原则

**1、开闭原则（Open Close Principle）**

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。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

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。

**3、依赖倒转原则（Dependence Inversion Principle）**

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。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：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。

**5、迪米特法则，又称最少知道原则（Demeter Principle）**

最少知道原则是指：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。

**6、合成复用原则（Composite Reuse Principle）**

合成复用原则是指：尽量使用合成/聚合的方式，而不是使用继承

# 第三部分：数据库

## 数据库事务的4个特性以及含义。

数据库事务transanction正确执行的四个基本要素。ACID,

原子性(Atomicity)、

一致性(Correspondence)、

隔离性(Isolation)、

持久性(Durability)。  
**原子性**:强调事务的无可分割，一次事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。  
**一致性**:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

|  |
| --- |
| 保证数据库的一致性是数据库管理系统的一项功能.比如有两个表(员工\职位),员工表中有员工代码、姓名、职位代码等属性，职位表中有职位代码、职位名称、职位等级等属性。你在其中员工表中进行了插入操作,你插入了一个新员工的信息，而这个新员工的职位是公司新创建的一个职位。如果没有一致性的保证，就会出现有这么一个员工，但是不知道他到底担当什么职责！这个只是它的一个小小方面。 |

**隔离性**:隔离状态执行事务，使它们好像是[系统](http://www.2cto.com/os/)在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，为了防止事务操作间的混淆，必须串行化或序列化请 求，使得在同一时间仅有一个请求用于同一数据。  
**持久性**:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

## Mysql数据库事务的几种隔离级别？

#### 如果不考虑事务的隔离性,引发一些安全性问题:

* 一类读问题:

**\* 脏读 :一个事务读到另一个事务还没有提交的数据.**

**\* 不可重复读 :一个事务读到了另一个事务已经提交的update的数据,导致在当前的事务中多次查询结果不一致.**

**\* 虚读/幻读 :一个事务读到另一个事务已经提交的insert的数据,导致在当前的事务中多次的查询结果不一致.**

* 一类写问题:

\* 引发两类丢失更新:

#### 解决引发的读问题:

设置事务的隔离级别:

**\* read uncommitted :未提交读，在这个隔离级别，所有事务都可以看到其他未提交的事务的执行结果.脏读，不可重复读，虚读都可能发生.**

**\* read committed :已提交读，一个事务只能看见已经提交事务所做的改变。.避免脏读.但是不可重复读和虚读有可能发生.**

**\* repeatable read :可重复读，它确保同一事务在读取同一个数据时，在并发的情况下会看到同样的数据。避免脏读,不可重复读.但是虚读有可能发生.**

**\* serializable :串行化的.避免脏读，不可重复读，虚读的发生.**

\*\*\*\*\* MYSQL隔离级别：repeatable read Oracle隔离级别:read committed

## 索引的原理？索引是不是越多越好？

索引，实际上也是一张表，该表保存了主键与索引字段(索引字段一般是where子句后面的查询条件),并指向实体表的记录。

索引提高了查询表的速度，但是索引过多会降低更新表的速度。对表进行增删改的时候，不仅要保存数据，还要保存 索引文件。

## 什么时候需要建索引，什么时候不建索引？

适合建索引

①主键自动建立唯一索引

　　　　②频繁作为查询条件的字段应该创建索引

　　　　③查询中与其他表关联的字段，外键关系建立索引

　　　　④频繁更新的字段不适合建立索引，因为每次更新不单单是更新了记录还会更新索引

　　　　⑤WHERE条件里用不到的字段不创建索引

　　　　⑥单键/组合索引的选择问题，who?(在高并发下倾向创建组合索引)

　　　　⑦查询中排序的字段，排序的字段若通过索引去访问将大大提高排序速度

　　　　⑧查询中统计或者分组字段

不建索引

①表记录太少

　　②经常增删改的表

　　　　提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE、和DELETE。

　　　　因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

　　　　数据重复且分布平均的表字段，因此应该只为最经常查询和最经常排序的数据建立索引。

　　③注意，如果某个数据列包含许多重复的内容，为它建立索引就没有太大的实际效果。

## 数据库的分库分表？

分库分表，又可以说为纵向拆分和横向拆分。

纵向拆分，就是将业务相似的表放在一个库里面，比如用户信息关联的表放在一起。

横向拆分,如果表的数据非常多，那就适合水平拆分，适合分表进行分布式存储，在餐饮加工系统中，是根据warehouseCode进行分表，一个门店一张表

## 视图是什么？视图的作用？视图可以更改吗？

简单的讲，视图就是一条SELECT语句执行后返回的结果集。

视图最大的作用就是方便查询操作，减少复杂的sql语句，增强可读性。

也可以增加安全性，数据库授权命令不能限定到特性行和特定列，但是通过合理创建视图，可以把权限限定到行级别。

视图不能被索引，也不能有关联的触发器或默认值，如果视图本身内有order by 则对视图再次order by将被覆盖。  
创建视图：create view XXX as XXXXXXXXXXXXXX;  
对于某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的，对视图的更新将对基表进行更新；但是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。

## Drop，delete和truncate的区别？

1. Drop是直接删掉表的数据以及表的结构，属于DLL语句，操作立即生效，并且不能回滚
2. Truncate table是一次性删除表中的所有数据，并且不能回滚，删除过程不会激活与表有关的删除触发器。执行速度快。并且删除之后，表和索引所占空间会恢复。只能删除TABLE，不能删除VIEW。
3. Delete属于DMl语句，删除操作可以回滚，删除之后，索引不会恢复，可以删除table和view，删除数据是一条一条的删。

## Mysql数据库连接的种类

1. 外链接

左外连接：left join或left outer join。

结果集为left语句中指定的左表的所有行，以及右表匹配到的行，没有匹配到的都是null值

右外链接：right oin 或right outer join

结果集为右表的所有行，和左表匹配到的元素，没有匹配到的为null。

完全外链接：full join或full outer join

返回左表和右表的所有行。

1. 内连接

Join或inner join

select \* from table1 join table2 on table1.id=table2.id

返回符合条件的table1和table2的列。

等价于：

A:select a.\*,b.\* from table1 a,table2 b where a.id=b.id  
B:select \* from table1 cross join table2 where table1.id=table2.id (**注：cross join后加条件只能用where,不能用on**)

1. 交叉连接

Cross join

没有where子句的交叉连接将产生连接所涉及的笛卡尔积。

select \* from table1 cross join table2

等价于：

select \* from table1,table2

## 数据库的范式

设计关系数据库时，遵从不同的规范要求，设计出合理的关系型数据库，这些不同的规范要求被称为不同的范式，各种范式呈递次规范，越高的范式数据库冗余越小。一般说来，数据库只需满足第三范式(3NF）就行了。

**1 第一范式（1NF）**  
  
在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。  
所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，**第一范式就是无重复的列。**

**2 第二范式（2NF）**  
  
第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。  
第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，**第二范式就是非主属性非部分依赖于主关键字。**

**3 第三范式（3NF）**  
  
满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，**第三范式就是属性不依赖于其它非主属性。（我的理解是消除冗余）**

## 数据库优化的思路

#### 1.SQL语句优化

1）应尽量避免在 where 子句中使用!=或<>操作符，否则引擎将放弃使用索引而进行全表扫描。  
2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
select id from t where num is null  
**可以在num上设置默认值0，确保表中num列没有null值**，然后这样查询：  
select id from t where num=0  
3）很多时候用 exists 代替 in 是一个好的选择  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤

#### 2.索引优化

看上文索引

#### 3.数据库结构优化

1）范式优化： 比如消除冗余（节省空间。。） 2）反范式优化：比如适当加冗余等（减少join） 3）拆分表： 分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。  
4）拆分其实又分垂直拆分和水平拆分： 案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量10w，稳定） 2.订单表（数据量200w，且有增长趋势） 3.用户表 （数据量100w，且有增长趋势） 以[mysql](http://www.2cto.com/database/MySQL/)为例讲述下水平拆分和垂直拆分，mysql能容忍的数量级在百万静态数据可以到千万 **垂直拆分：**解决问题：表与表之间的io竞争 不解决问题：单表中数据量增长出现的压力 方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上 **水平拆分：** 解决问题：单表中数据量增长出现的压力 不解决问题：表与表之间的io争夺  
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单 产品表 未完成订单放一个server上 已完成订单表盒男用户表放一个server上 女用户表放一个server上(女的爱购物 哈哈)

#### 4.服务器硬件优化

这个么多花钱咯！

## 常用SQL语句优化

1. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。   
     
   2.应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。   
     
   3.应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：   
   select id from t where num is null   
   可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：   
   select id from t where num=0   
     
   4.应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：   
   select id from t where num=10 or num=20   
   可以这样查询：   
   select id from t where num=10   
   union all   
   select id from t where num=20   
     
   5.模糊查询也将导致全表扫描：   
   select id from t where name like '%abc%'   
   若要提高效率，可以考虑全文检索。   
     
   6.in 和 not in 也要慎用，否则会导致全表扫描，如：   
   select id from t where num in(1,2,3)   
   对于连续的数值，能用 between 就不要用 in 了：   
   select id from t where num between 1 and 3   
     
   7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：   
   select id from t where num=@num   
   可以改为强制查询使用索引：   
   select id from t with(index(索引名)) where num=@num   
     
   8.应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
   select id from t where num/2=100   
   应改为:   
   select id from t where num=100\*2   
     
   9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：   
   select id from t where substring(name,1,3)='abc'--name以abc开头的id   
   select id from t where datediff(day,createdate,'2005-11-30')=0--'2005-11-30'生成的id   
   应改为:   
   select id from t where name like 'abc%'   
   select id from t where createdate>='2005-11-30' and createdate<'2005-12-1'   
     
   10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。   
     
   13.很多时候用 exists 代替 in 是一个好的选择：   
   select num from a where num in(select num from b)   
   用下面的语句替换：   
   select num from a where exists(select 1 from b where num=a.num)   
     
   14.并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。   
     
   15.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。   
     
     
   17.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。   
     
   18.尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。   
     
   19.任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。   
     
     
   21.避免频繁创建和删除临时表，以减少系统表资源的消耗。

## 事务处理

### MySql事务处理

一个事务是一个连续的一组[数据库](http://www.111cn.net/database/database.html)操作，就好像它是一个单一的工作单元进行。换言之，永远不会是完整的事务，除非该组内的每个单独的操作是成功的。如果在事务的任何操作失败，则整个事务将失败。

在MySQL中，事务开始使用COMMIT或ROLLBACK语句开始工作和结束。开始和结束语句的SQL命令之间形成了大量的事务。

如果打算使用MySQL事务编程，那么就需要一种特殊的方式创建表。有很多支持事务但最流行的是InnoDB表类型。

如果MySQL安装支持InnoDB表，只需添加一个的TYPE=InnoDB 定义表创建语句。

root@host# mysql -u root -p pass[word](http://www.111cn.net/list-189/);  
Enter password:\*\*\*\*\*\*\*  
mysql> use TUTORIALS;  
Database changed  
mysql> create table tcount\_tbl  
    -> (  
    -> tutorial\_author varchar(40) NOT NULL,  
    -> tutorial\_count  INT  
    -> ) TYPE=InnoDB;  
Query OK, 0 rows affected (0.05 sec)

设置“自动提交”模式  
SET AUTOCOMMIT = 0

需要用以下模式来设定锁定模式

1、SELECT …… LOCK IN SHARE MODE（共享锁）  
查询到的数据，就是数据库在这一时刻的数据（其他已commit事务的结果，已经反应到这里了）  
SELECT 必须等待，某个事务结束后才能执行

2、SELECT …… FOR UPDATE（排它锁）  
例如 SELECT \* FROM tablename WHERE id<200  
那么id<200的数据，被查询到的数据，都将不能再进行修改、删除、SELECT …… LOCK IN SHARE MODE操作  
一直到此事务结束

共享锁 和 排它锁 的区别：在于是否阻断其他客户发出的 SELECT …… LOCK IN SHARE MODE命令

3、INSERT / UPDATE / DELETE  
所有关联数据都会被锁定，加上排它锁

4、防插入锁  
例如 SELECT \* FROM tablename WHERE id>200  
那么id>200的记录无法被插入

5、死锁  
自动识别死锁  
先进来的进程被执行，后来的进程收到出错消息，并按ROLLBACK方式回滚  
innodb\_lock\_wait\_timeout = n 来设置最长等待时间，默认是50秒

隔离模式

   READ UNCOMMITTED  
不隔离SELECT  
其他事务未完成的修改（未COMMIT），其结果也考虑在内

   READ COMMITTED  
把其他事务的 COMMIT 修改考虑在内  
同一个事务中，同一 SELECT 可能返回不同结果

   REPEATABLE READ（mysql默认）  
不把其他事务的修改考虑在内，无论其他事务是否用COMMIT命令提交过  
同一个事务中，同一 SELECT 返回同一结果（前提是本事务，不修改）

   SERIALIZABLE  
和REPEATABLE READ类似，给所有的SELECT都加上了 共享锁

## 如果数据库在执行事务的时候突然断电，这时怎么保证数据的一致性?

我们使用SQL Server来举例，我们知道我们在使用 SQL Server 数据库是由两个文件组成的，一个数据库文件和一个日志文件，通常情况下，日志文件都要比数据库文件大很多。数据库进行任何写入操作的时候都是要先写日志的，同样的道理，我们在执行事务的时候数据库首先会记录下这个事务的redo操作日志，然后才开始真正操作数据库，在操作之前首先会把日志文件写入磁盘，那么当突然断电的时候，即使操作没有完成，在重新启动数据库时候，数据库会根据当前数据的情况进行undo回滚或者是redo前滚，这样就保证了数据的强一致性。

### Spring对事务的处理

#### 编程式的事务处理

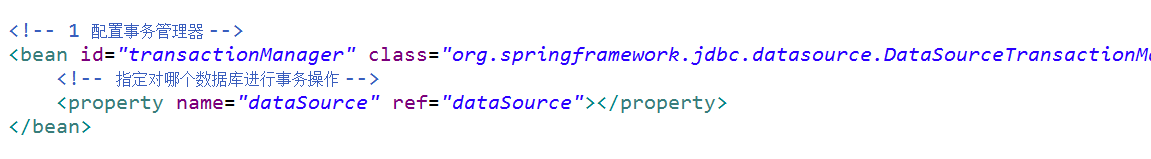
不推荐使用

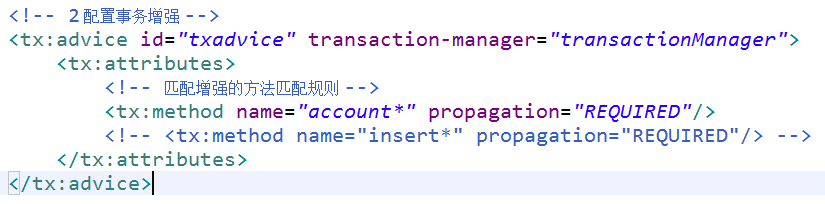
#### 基于xml方式

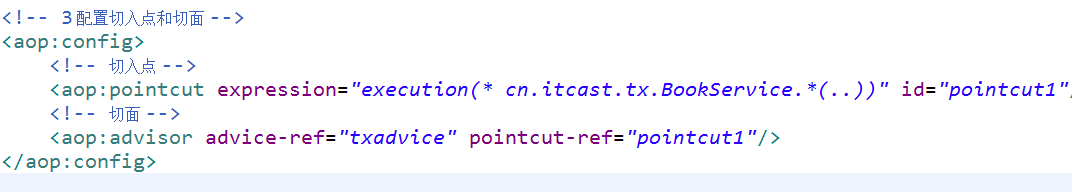
Spring为不同的框架提供了不同的事务管理器。

JDBC或MyBatis使用的是DataSourceTransactionManager

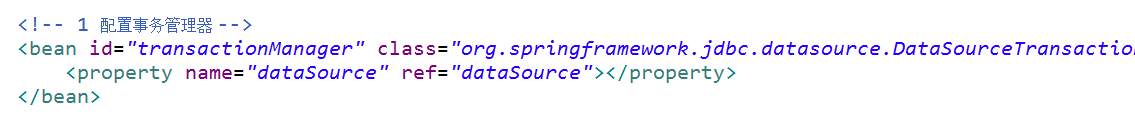
Hibernate使用的是HibernateTransactionManager

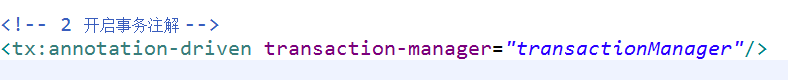


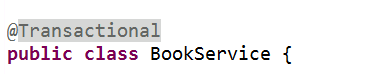
****

****

#### 基于注解范式

****

****

****

### 分布式事务处理

1.什么是分布式事务？

简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

2.常见的解决方案：

基于XA协议的两阶段提交。XA协议大致为分两部分：事务管理器和本地资源管理器（由数据库实现，比如Oracle、DB2都实现了XA接口）。基于XA协议的分布式事务，成本比较低，但是不能满足高并发的场景。Mysql5.5之后开始支持

补偿事务(TCC)

TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

* Try 阶段主要是对业务系统做检测及资源预留
* Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。
* Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

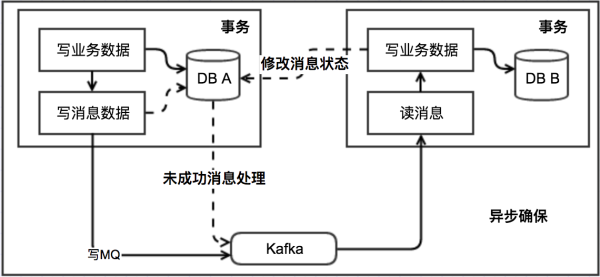
举个例子，假入 Bob 要向 Smith 转账，思路大概是：  
我们有一个本地方法，里面依次调用  
1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。  
2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。  
3、如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

**优点：** 跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

**缺点：** 缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

消息事务+最终一致性

本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理，这种思路是来源于ebay。我们可以从下面的流程图中看出其中的一些细节：



基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

这种方案遵循BASE理论，采用的是最终一致性，笔者认为是这几种方案里面比较适合实际业务场景的，即不会出现像2PC那样复杂的实现(当调用链很长的时候，2PC的可用性是非常低的)，也不会像TCC那样可能出现确认或者回滚不了的情况。

**优点：** 一种非常经典的实现，避免了分布式事务，实现了最终一致性。在 .NET中 有现成的解决方案。

**缺点：** 消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

### Hibernate的事务管理？

Hibernate可以使用JDBC的事务，也可以使用JTA的事务。具体使用哪一种，可以在Hibernate.properties中配置具体的事务工厂类，JDBCTransactionFactory和JTATransactionFactory，如果不配置默认使用JDBC事务。

JDBC模式下，使用

Transaction tx = session.beginTransaction();//方法开启事务。

Tx.commit()//提交事务

Tx.rollback()//回滚事务。

JTA事务：

|  |  |
| --- | --- |
| |  | | --- | | javax.transaction.UserTransaction tx = null;  tx = new initialContext().lookup(” javax.transaction.UserTransaction ”) ;  tx.begin();  Session s1 = sf.openSession();  ……  s1.flush();  s1.close();  Session s2 = sf.openSession();  ……  s2.flush();  s2.close();  tx.commit(); | |

* + - 1. 获得UserTransaction对象
      2. Tx.begin()开启事务
      3. Tx.commit()提交事务
      4. Tx.rollback()回滚事务。

### JDBC事务和JTA事务的区别？

简单点说：JDBC事务是单库事务、JTA是多库事务。

Jdbc事务由Connection对象管理，默认是自动提交，主要用于一个数据库连接，一个JDBC事务不能跨越多个数据库。

JTA提供了跨数据库连接的事务管理能力。UserTransaction接口、TransactionManager接口、Transaction接口、XAResource。

a、首先配置JTA ，建立相应的数据源  
b、建立事务：通过创建UserTransaction类的实例来开始一个事务。代码如下：  
　　Context ctx = new InitialContext(p) ;  
　  UserTransaction trans = (UserTransaction) ctx.lookup("javax. Transaction.UserTransaction")  
c、开始事务：代码为 trans.begin() ;  
d、找出数据源：从Weblogic Server上找到数据源，代码如下：  
　DataSource ds = (DataSource) ctx.lookup(“mysqldb") ;  
e、建立数据库连接：Connection mycon = ds.getConnection() ;  
f、执行SQL操作：stmt.executeUpdate(sqlS);  
g、完成事务：trans.commit(); / trans.rollback();  
h、关闭连接：mycon.close() ;

# 第四部分：常用框架

## Struts2

### Struts2的执行流程？

发送一个请求，会被struts2的核心过滤器拦截，之后根据struts.xml配置文件的配置，找到相应的action类，执行指定的方法，方法返回一个字符串，有struts.xml中的<result../>元素接受，然后跳转到要响应的结果视图。

### 创建Action类的几种方式？

1. 创建一个普通的类，不继承任何类，不实现任何接口
2. 创建一个类，实现Action接口。
3. 创建一个类，继承ActionSupport类。

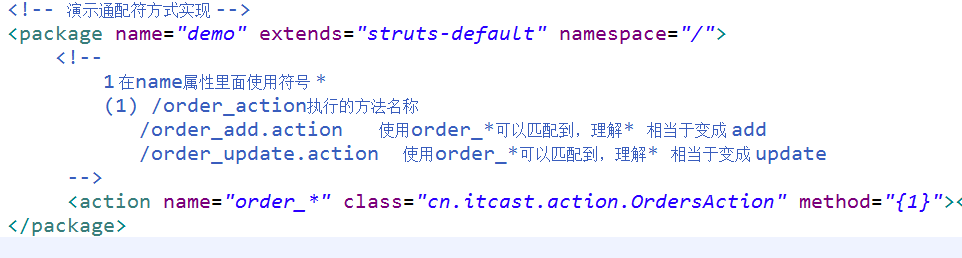
### 创建Action类为什么常用继承ActionSupport类的方式？

Actionsupport这个工具类在实现了Action接口的基础上还定义了一个validate()方法,重写该方法,它会在execute()方法之前执行,如校验失败,会转入input处，必须在配置该Action时配置input属性。

另外,Actionsupport还提供了一个getText(String key)方法还实现国际化,该方法从资源文件上获取国际化信息. 这样在自定义标签时可以定义一个变量为new actionsupport对象实现国际化。

### Struts2访问Action中方法的三种方式？

1. 通过action标签里面的method属性可以定位到要执行的方法名称，这种方式的缺点在于如果方法过多，配置会非常的臃肿。
2. 使用通配符的形式，比如name属性可以是user\_\*,method={1}。

****

1. 使用动态访问的方式。

使用这种方式，首先需要开启一个常量，然后在访问的路径里面，使用!方法名称。

### Struts2封装数据的三种方式

1. 传统方式封装数据:

所谓传统方式就是先从域对象中获得数据,然后new对象,调用setter方法,封装数据.

1. 属性封装的方式:

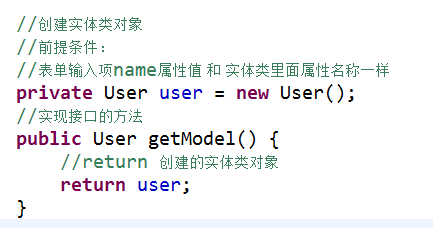
2.1 在action中定义成员变量,变量的名称和表单中的name属性相同,然后生成其getter和setter方法.之后new对象,调用setter方法,封装数据

2.2 在action中定义对象,生成对象的getter和setter方法.在jsp页面中数据的name属性为对象.属性的方式如:user.username.

1. 模型封装(重点):

直接把数据封装到实体对象里面

1. 让action类实现ModelDriven<T>接口 T是泛型
2. 在成员变量手动创建实体对象
3. 重写getModel()方法,return user;

****

**注意:属性封装和模型封装如果同时使用,只会使用模型封装.**

### Struts2如何访问Servlet API

1. 使用ActionContext类获取

1.1使用ActionContext类的静态方法getContext()获得Context

ActionContext context = ActionContxt.getContext();

1.2调用方法

context.get(String key);

context.getParameters();

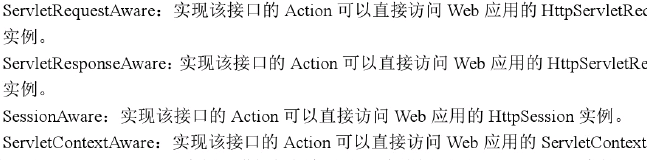
1. 使用ServletActionContext类获取

直接调用ServletActionContext类的静态方法可以获得各个域对象,然后从域对象中获取数据即可

ServletActionContext.getRequest();

1. action类实现接口的方式获得数据.

实现响应的接口,然后提供域对象的成员变量和setter方法,就可以获得域对象.

****

### 什么是拦截器？怎么自定义拦截器？

概述:

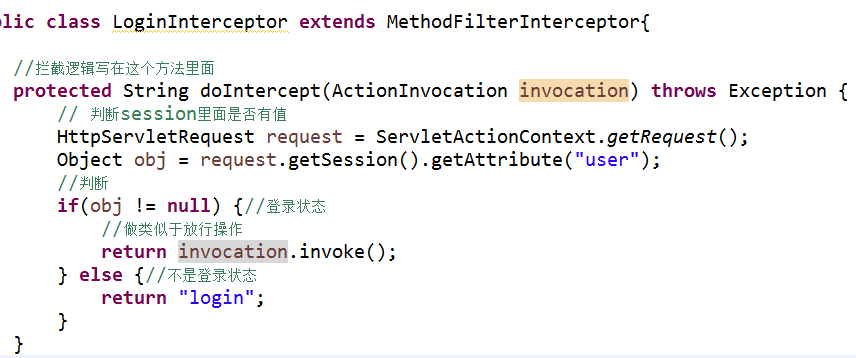
Struts2的拦截器只对action进行拦截.在action的方法执行之前进行拦截

自定义拦截器的两种方式:

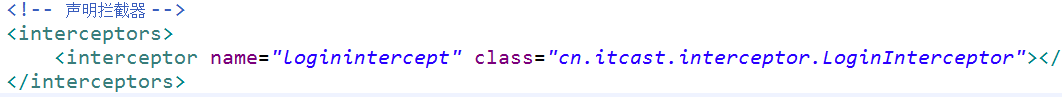
1. 继承AbstractInterceptor
2. 继承MethodFilterInterceptor(常用)

自定义拦截器的步骤:

1. 定义拦截器类继承MethodFilterInterceptor

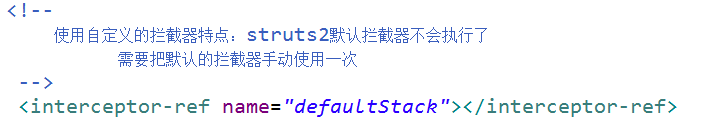
****

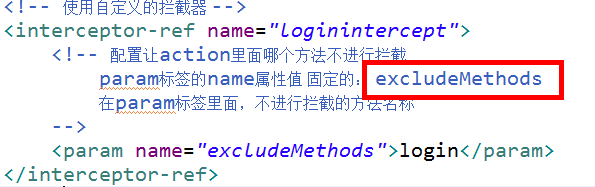
1. 配置文件,使自定义拦截器关联action 类

****

**在action中使用拦截器**

****

****

****

### 项目中遇到什么问题

答：  
1.表单重复提交。  
在页面使用标签<s:token/>(令牌机制)，使用检验表单从复提交的拦截器tokenSession。。  
2.国际化必须经过Action  
原因：因为国际化是通过i18拦截器来实现的，而拦截器是在访问Action的时候才执行。  
解决方式：在对jsp的访问之前进行对action的访问。  
3.使用模型驱动拿返回值的时候地址内存不一样，会拿不到值。  
原因：因为struts2最开始是把创建action的实例放到ActionContext的栈顶，后来把创建模型驱动的实例放到栈顶，  
而通过struts2的标签去取值会取栈顶的实例，而此时拿到的还是最开始的创建的那个实例，所以为空。  
解决方式：利用原型进行对象拷贝，使用拷贝工具，BeanUtils或者是propertyUtils。  
4.在页面使用转发会报404错。  
原因：struts2使用的是filter机制,<jsp:forward/>的机制是servlet。  
解决方式：可以通过过滤器改变请求地址。

5.在通过shiro框架进行权限控制的时候,在UserAction中使用了注解进行权限限制,但是在运行的时候却报出 一个类型转换的异常,但是在这之前系统运行都是正常的.最后发现,原来UserAction是继承自一个BaseAction类,而BaseAction类中使用了泛型,构造方法中有一个类型化参数的强转的代码,之前都是直接得到当前实例对象即UserAction的父类即BaseAction之后强转成参数化类型对象.但是使用注解进行权限控制后,由于注解底层是使用的反射,这时候当前实例对象就变成了UserAction的子类,UserAction也就成了当前实例化对象的父类,那原来的代码就把UserAction强转成参数化类型对象,最后也就报错了.

解决的时候,也就是在baseAction中增加判断条件,当是参数化类型的实例后再进行强转.

### ActionContext是做什么的？

答：ActionContext是Action的上下文。  
通过ActionContext这个类来可以获得:request，application，session，Locale、ValueStack,ConversionErrors  
1 可以获得servlet的api：request，application，session。  
2 进行国际化：获得语言环境 Locale。  
3 收集错误信息：ConversionErrors。  
4 有个值栈来装action实例与创建的模型驱动的实例:ValueStack。  
5 ActionContext是线程安全的-用本地线程实现的。

### Struts2是如何实现mvc的？

### Struts2怎么进行文件的上传和下载？

Struts2的文件上传功能对FileUpload进行了封装。Struts使用拦截器实现文件上传。

1.页面的要求：

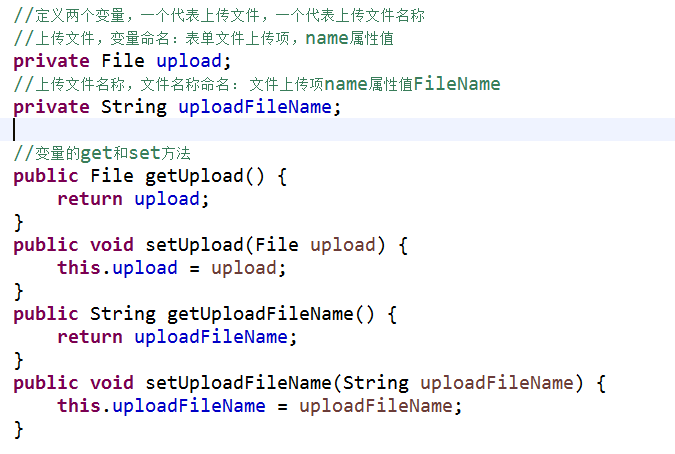
表单提交方式post

Form标签的enctype修改为multipart/form-data

表单里面有文件上传项：

****

2.在Action中直接得到上传文件名和上传文件

****

（3）在具体的action的方法里面写上传逻辑

- 在服务器里面创建文件

- 把本地文件复制到服务器文件里面

****

****

### 说出至少5个Struts2的标签？

s:iterater ?s:property s:form s:url s:if

### OGNL表达式

1. OGNL是什么?

OGNL不是struts2的一部分,类似于EL表达式,它可以从Struts2的ValueStack中获得数据.

1. OGNL怎么用?

经常和struts的标签一起使用,在value后面也OGNl表达式

1. 什么是ValueStack?

每次访问action的时候,都会创建action对象,在每个对象中,都维护了一个ValueStack,类似于域对象,我们可以在其中存值和取值.

1. 怎么获得值栈?

使用actionContext对象的方法getValueStack()获得值栈对象.

1. 值栈的内部结构.
2. root list集合
3. context map集合
4. 向值栈中放数据的三种方式:
5. 获得值栈对象,调用set方法

stack.set(“username”,”abc”)

1. 获得值栈对象,调用push方法

stack.puch(“abcd”)

1. 在action类定义成员变量,生成其get方法.

放对象,list集合

1. 从值栈中取数据:

放:stack.set(“username”,”abc”)

取:<s:property value=”username”>

放:对象

取:<s:property value=”user.username”>

放:list集合

取:<s:property value=”list[0].username”>

或:<iterator value=”list”>

<s:property value=”username”>

</iterator>

****

**总结:从root中获取数据,直接写名称即可,如果从context中获取数据,需要加#,如果从context里面的Request,ServletContext等域对象中获得数据,还需要加域名#request.username**

### Struts2的组件？

组件分为：

控制器组件：ActionServlet和自定义的ActionBean

视图组件 ：Jsp页面、Formbean等

模型组件 ：本身不提供模型组件，由普通的javabean组成。

## 2.SpringMvc

### SpringMvc的执行流程？

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器
5. 执行处理器(Controller，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter处理器适配器将ModelAndView返回给DispatcherServlet前端控制器
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户

### SpringMVC和Struts2的不同？

1. springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过虑器。
2. **springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。**
3. Struts采用值栈存储请求和响应的数据，通过OGNL存取数据， springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

**64. 什么是Spring的MVC框架？**

Spring 配备构建Web 应用的全功能MVC框架。Spring可以很便捷地和其他MVC框架集成，如Struts，Spring 的MVC框架用控制反转把业务对象和控制逻辑清晰地隔离。它也允许以声明的方式把请求参数和业务对象绑定。

**65. DispatcherServlet**

Spring的MVC框架是围绕DispatcherServlet来设计的，它用来处理所有的HTTP请求和响应。

**66. WebApplicationContext**

WebApplicationContext 继承了ApplicationContext 并增加了一些WEB应用必备的特有功能，它不同于一般的ApplicationContext ，因为它能处理主题，并找到被关联的servlet。

**67. 什么是Spring MVC框架的控制器？**

控制器提供一个访问应用程序的行为，此行为通常通过服务接口实现。控制器解析用户输入并将其转换为一个由视图呈现给用户的模型。Spring用一个非常抽象的方式实现了一个控制层，允许用户创建多种用途的控制器。

**68. @Controller 注解**

该注解表明该类扮演控制器的角色，Spring不需要你继承任何其他控制器基类或引用Servlet API。

**69. @RequestMapping 注解**

该注解是用来映射一个URL到一个类或一个特定的方处理法上。

## Spring

**1. 什么是spring?**

[**spring**](http://lib.csdn.net/base/javaee) 是个[**Java**](http://lib.csdn.net/base/javase)企业级应用的开源开发框架。Spring主要用来开发Java应用，但是有些扩展是针对构建J2EE平台的web应用。Spring 框架目标是简化Java企业级应用开发，并通过POJO为基础的编程模型促进良好的编程习惯。

Spring的IoC容器，简化了开发人员对类与类之间依赖关系的维护，在利用Spring进行开发的过程中，只需定义需要的接口，而对应的实例化过程则交给IoC容器来进行注入。这实际上也是一种解耦的操作，通过结合Spring和面向接口的编程思想，实际开发出来的代码十分的简洁，而且逻辑清晰明确。同时也提高了功能类的代码复用率。

Spring的AOP，带来最大的好处就是让开发人员从繁琐的事务操作代码中解放出来，让开发人员更加专注于业务逻辑代码的实现。



**2. 使用Spring框架的好处是什么？**

* **轻量：**Spring 是轻量的，基本的版本大约2MB。
* **控制反转：**Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。
* **面向切面的编程(AOP)：**Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。
* **容器：**Spring 包含并管理应用中对象的生命周期和配置。
* **MVC框架**：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。
* **事务管理：**Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。
* **异常处理：**Spring 提供方便的API把具体技术相关的异常（比如由JDBC，[**hibernate**](http://lib.csdn.net/base/javaee) or JDO抛出的）转化为一致的unchecked 异常。

**3. Spring由哪些模块组成?**

以下是Spring 框架的基本模块：

* Core module
* Bean module
* Context module
* Expression Language module
* JDBC module
* ORM module
* OXM module
* [**Java**](http://lib.csdn.net/base/java)Messaging Service(JMS) module
* Transaction module
* Web module
* Web-Servlet module
* Web-Struts module
* Web-Portlet module

**4. 核心容器（应用上下文) 模块。**

这是基本的Spring模块，提供spring 框架的基础功能，BeanFactory 是 任何以spring为基础的应用的核心。Spring 框架建立在此模块之上，它使Spring成为一个容器。

**5. BeanFactory – BeanFactory 实现举例。**

Bean 工厂是工厂模式的一个实现，提供了控制反转功能，用来把应用的配置和依赖从正真的应用代码中分离。

最常用的BeanFactory 实现是XmlBeanFactory 类。

**6. XMLBeanFactory**

最常用的就是org.springframework.beans.factory.xml.XmlBeanFactory ，它根据XML文件中的定义加载beans。该容器从XML 文件读取配置元数据并用它去创建一个完全配置的系统或应用。

**7. 解释AOP模块**

AOP模块用于发给我们的Spring应用做面向切面的开发， 很多支持由AOP联盟提供，这样就确保了Spring和其他AOP框架的共通性。这个模块将元数据编程引入Spring。

**8. 解释JDBC抽象和DAO模块。**

通过使用JDBC抽象和DAO模块，保证[**数据库**](http://lib.csdn.net/base/mysql)代码的简洁，并能避免数据库资源错误关闭导致的问题，它在各种不同的数据库的错误信息之上，提供了一个统一的异常访问层。它还利用Spring的AOP 模块给Spring应用中的对象提供事务管理服务。

**9. 解释对象/关系映射集成模块。**

Spring 通过提供ORM模块，支持我们在直接JDBC之上使用一个对象/关系映射映射(ORM)工具，Spring 支持集成主流的ORM框架，如Hiberate,JDO和 iBATIS SQL Maps。Spring的事务管理同样支持以上所有ORM框架及JDBC。

**10. 解释WEB 模块。**

Spring的WEB模块是构建在application context 模块基础之上，提供一个适合web应用的上下文。这个模块也包括支持多种面向web的任务，如透明地处理多个文件上传请求和程序级请求参数的绑定到你的业务对象。它也有对Jakarta Struts的支持。

**12. Spring配置文件**

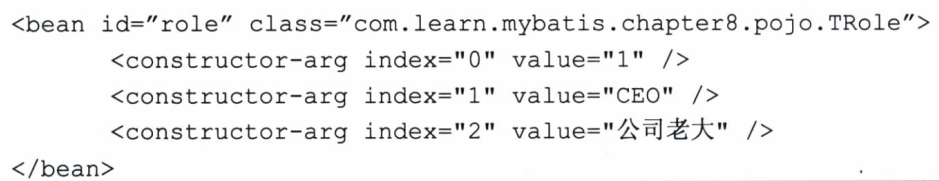
Spring配置文件是个XML 文件，这个文件包含了类信息，描述了如何配置它们，以及如何相互调用。

**13. 什么是Spring IOC 容器？**

Spring IOC 负责创建对象，管理对象（通过依赖注入（DI），装配对象，配置对象，并且管理这些对象的整个生命周期。

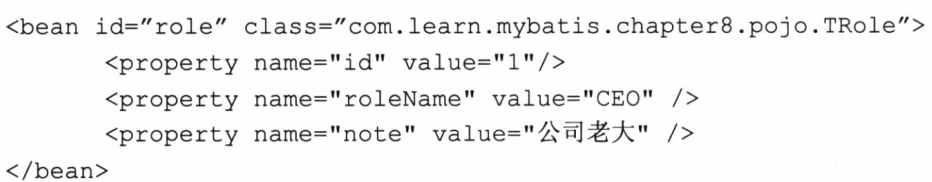
#### IOC注入的三种方式

1.构造方法注入



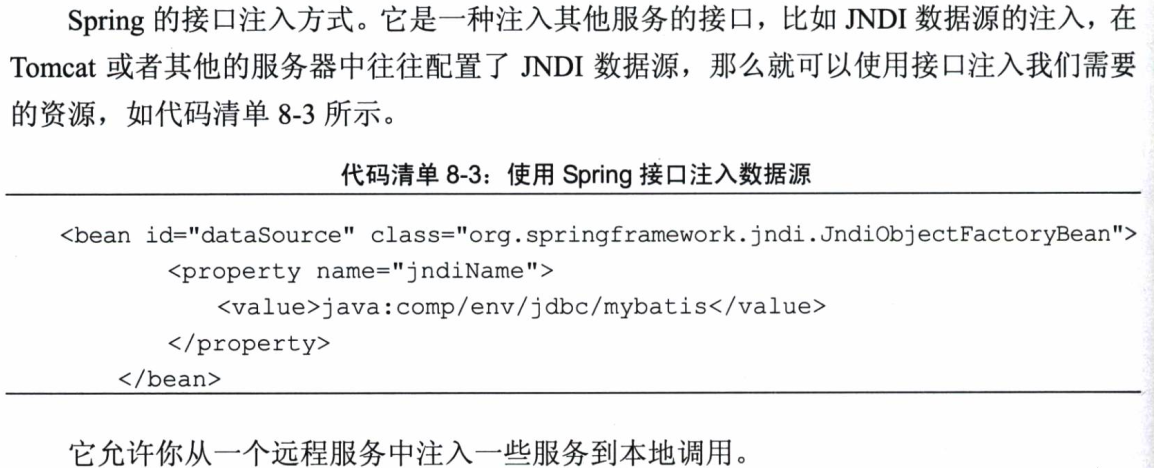
如果构造方法多，构造注入就不是很好了。因为会受到构造方法的参数个数，顺序这些引入干扰。Spring更推荐使用setter注入

2.setter方法注入



首选

3.接口注入



**14. IOC的优点是什么？**

IOC 或 依赖注入把应用的代码量降到最低。它使应用容易[**测试**](http://lib.csdn.net/base/softwaretest)，单元测试不再需要单例和JNDI查找机制。最小的代价和最小的侵入性使松散耦合得以实现。IOC容器支持加载服务时的饿汉式初始化和懒加载。

**15. ApplicationContext通常的实现是什么?**

* **FileSystemXmlApplicationContext ：**此容器从一个XML文件中加载beans的定义，XML Bean 配置文件的全路径名必须提供给它的构造函数。
* **ClassPathXmlApplicationContext：**此容器也从一个XML文件中加载beans的定义，这里，你需要正确设置classpath因为这个容器将在classpath里找bean配置。
* **WebXmlApplicationContext：**此容器加载一个XML文件，此文件定义了一个WEB应用的所有bean。

**16. Bean 工厂和 Application contexts 有什么区别？**

Application contexts提供一种方法处理文本消息，一个通常的做法是加载文件资源（比如镜像），它们可以向注册为监听器的bean发布事件。另外，在容器或容器内的对象上执行的那些不得不由bean工厂以程序化方式处理的操作，可以在Application contexts中以声明的方式处理。Application contexts实现了MessageSource接口，该接口的实现以可插拔的方式提供获取本地化消息的方法。

**17. 一个Spring的应用看起来象什么？**

* 一个定义了一些功能的接口。
* 这实现包括属性，它的Setter ， getter 方法和函数等。
* Spring AOP。
* Spring 的XML 配置文件。
* 使用以上功能的客户端程序。

### 依赖注入

**18. 什么是Spring的依赖注入？**

依赖注入，是IOC的一个方面，是个通常的概念，它有多种解释。这概念是说你不用创建对象，而只需要描述它如何被创建。你不在代码里直接组装你的组件和服务，但是要在配置文件里描述哪些组件需要哪些服务，之后一个容器（IOC容器）负责把他们组装起来。

**19. 有哪些不同类型的IOC（依赖注入）方式？**

* **构造器依赖注入：**构造器依赖注入通过容器触发一个类的构造器来实现的，该类有一系列参数，每个参数代表一个对其他类的依赖。
* **Setter方法注入：**Setter方法注入是容器通过调用无参构造器或无参static工厂 方法实例化bean之后，调用该bean的setter方法，即实现了基于setter的依赖注入。
* **注解方式**

**20. 哪种依赖注入方式你建议使用，构造器注入，还是 Setter方法注入？**

你两种依赖方式都可以使用，构造器注入和Setter方法注入。最好的解决方案是用构造器参数实现强制依赖，setter方法实现可选依赖。

### Spring Beans

**21.什么是Spring beans?**

Spring beans 是那些形成Spring应用的主干的java对象。它们被Spring IOC容器初始化，装配，和管理。这些beans通过容器中配置的元数据创建。比如，以XML文件中<bean/> 的形式定义。

Spring 框架定义的beans都是单件beans。在bean tag中有个属性”singleton”，如果它被赋为TRUE，bean 就是单件，否则就是一个 prototype bean。默认是TRUE，所以所有在Spring框架中的beans 缺省都是单件。

**22. 一个 Spring Bean 定义 包含什么？**

一个Spring Bean 的定义包含容器必知的所有配置元数据，包括如何创建一个bean，它的生命周期详情及它的依赖。

**23. 如何给Spring 容器提供配置元数据?**

这里有三种重要的方法给Spring 容器提供配置元数据。

XML配置文件。

基于注解的配置。

基于java的配置。

**24. 你怎样定义类的作用域?**

当定义一个<bean> 在Spring里，我们还能给这个bean声明一个作用域。它可以通过bean 定义中的scope属性来定义。如，当Spring要在需要的时候每次生产一个新的bean实例，bean的scope属性被指定为prototype。另一方面，一个bean每次使用的时候必须返回同一个实例，这个bean的scope 属性 必须设为 singleton。

**25. 解释Spring支持的几种bean的作用域。**

Spring框架支持以下五种bean的作用域：

* **singleton :**bean在每个Spring ioc 容器中只有一个实例。
* **prototype**：一个bean的定义可以有多个实例。
* **request**：每次http请求都会创建一个bean，该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **session**：在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
* **global-session**：在一个全局的HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。

缺省的Spring bean 的作用域是Singleton.

**26. Spring框架中的单例bean是线程安全的吗?**

不，Spring框架中的单例bean不是线程安全的。

**27. 解释Spring框架中bean的生命周期。**

* Spring容器 从XML 文件中读取bean的定义，并实例化bean。
* Spring根据bean的定义填充所有的属性。
* 如果bean实现了BeanNameAware 接口，Spring 传递bean 的ID 到 setBeanName方法。
* 如果Bean 实现了 BeanFactoryAware 接口， Spring传递beanfactory 给setBeanFactory 方法。
* 如果有任何与bean相关联的BeanPostProcessors，Spring会在postProcesserBeforeInitialization()方法内调用它们。
* 如果bean实现IntializingBean了，调用它的afterPropertySet方法，如果bean声明了初始化方法，调用此初始化方法。
* 如果有BeanPostProcessors 和bean 关联，这些bean的postProcessAfterInitialization() 方法将被调用。
* 如果bean实现了 DisposableBean，它将调用destroy()方法。

**28. 哪些是重要的bean生命周期方法？ 你能重载它们吗？**

有两个重要的bean 生命周期方法，第一个是setup ， 它是在容器加载bean的时候被调用。第二个方法是 teardown 它是在容器卸载类的时候被调用。

The bean 标签有两个重要的属性（init-method和destroy-method）。用它们你可以自己定制初始化和注销方法。它们也有相应的注解（@PostConstruct和@PreDestroy）。

**29. 什么是Spring的内部bean？**

当一个bean仅被用作另一个bean的属性时，它能被声明为一个内部bean，为了定义inner bean，在Spring 的 基于XML的 配置元数据中，可以在 <property/>或 <constructor-arg/> 元素内使用<bean/> 元素，内部bean通常是匿名的，它们的Scope一般是prototype。

**30. 在 Spring中如何注入一个java集合？**

Spring提供以下几种集合的配置元素：

* <list>类型用于注入一列值，允许有相同的值。
* <set> 类型用于注入一组值，不允许有相同的值。
* <map> 类型用于注入一组键值对，键和值都可以为任意类型。
* <props>类型用于注入一组键值对，键和值都只能为String类型。

**31. 什么是bean装配?**

装配，或bean 装配是指在Spring 容器中把bean组装到一起，前提是容器需要知道bean的依赖关系，如何通过依赖注入来把它们装配到一起。

**32. 什么是bean的自动装配？**

Spring 容器能够自动装配相互合作的bean，这意味着容器不需要<constructor-arg>和<property>配置，能通过Bean工厂自动处理bean之间的协作。

**33. 解释不同方式的自动装配 。**

有五种自动装配的方式，可以用来指导Spring容器用自动装配方式来进行依赖注入。

* **no**：默认的方式是不进行自动装配，通过显式设置ref 属性来进行装配。
* **byName：**通过参数名 自动装配，Spring容器在配置文件中发现bean的autowire属性被设置成byname，之后容器试图匹配、装配和该bean的属性具有相同名字的bean。
* **byType:：**通过参数类型自动装配，Spring容器在配置文件中发现bean的autowire属性被设置成byType，之后容器试图匹配、装配和该bean的属性具有相同类型的bean。如果有多个bean符合条件，则抛出错误。
* **constructor：这个方式类似于**byType， 但是要提供给构造器参数，如果没有确定的带参数的构造器参数类型，将会抛出异常。
* **autodetect：**首先尝试使用constructor来自动装配，如果无法工作，则使用byType方式。

**34.自动装配有哪些局限性 ?**

自动装配的局限性是：

* **重写**： 你仍需用 <constructor-arg>和 <property> 配置来定义依赖，意味着总要重写自动装配。
* **基本数据类型**：你不能自动装配简单的属性，如基本数据类型，String字符串，和类。
* **模糊特性：**自动装配不如显式装配精确，如果有可能，建议使用显式装配。

**35. 你可以在Spring中注入一个null 和一个空字符串吗？**

**可以。**

### Spring注解

**36. 什么是基于Java的Spring注解配置? 给一些注解的例子.**

基于Java的配置，允许你在少量的Java注解的帮助下，进行你的大部分Spring配置而非通过XML文件。

以@Configuration 注解为例，它用来标记类可以当做一个bean的定义，被Spring IOC容器使用。另一个例子是@Bean注解，它表示此方法将要返回一个对象，作为一个bean注册进Spring应用上下文。

**37. 什么是基于注解的容器配置?**

相对于XML文件，注解型的配置依赖于通过字节码元数据装配组件，而非尖括号的声明。

开发者通过在相应的类，方法或属性上使用注解的方式，直接组件类中进行配置，而不是使用xml表述bean的装配关系。

**38. 怎样开启注解装配？**

注解装配在默认情况下是不开启的，为了使用注解装配，我们必须在Spring配置文件中配置 <context:annotation-config/>元素。

**39. @Required 注解**

这个注解表明bean的属性必须在配置的时候设置，通过一个bean定义的显式的属性值或通过自动装配，若@Required注解的bean属性未被设置，容器将抛出BeanInitializationException。

**40. @Autowired 注解**

@Autowired 注解提供了更细粒度的控制，包括在何处以及如何完成自动装配。它的用法和@Required一样，修饰setter方法、构造器、属性或者具有任意名称和/或多个参数的PN方法。

**41. @Qualifier 注解**

当有多个相同类型的bean却只有一个需要自动装配时，将@Qualifier 注解和@Autowire 注解结合使用以消除这种混淆，指定需要装配的确切的bean。

### Spring数据访问

**42.在Spring框架中如何更有效地使用JDBC?**

使用SpringJDBC 框架，资源管理和错误处理的代价都会被减轻。所以开发者只需写statements 和 queries从数据存取数据，JDBC也可以在Spring框架提供的模板类的帮助下更有效地被使用，这个模板叫JdbcTemplate （例子见这里here）

**43. JdbcTemplate**

JdbcTemplate 类提供了很多便利的方法解决诸如把数据库数据转变成基本数据类型或对象，执行写好的或可调用的数据库操作语句，提供自定义的数据错误处理。

**44. Spring对DAO的支持**

Spring对数据访问对象（DAO）的支持旨在简化它和数据访问技术如JDBC，Hibernate or JDO 结合使用。这使我们可以方便切换持久层。编码时也不用担心会捕获每种技术特有的异常。

**45. 使用Spring通过什么方式访问Hibernate?**

在Spring中有两种方式访问Hibernate：

* 控制反转 Hibernate Template和 Callback。
* 继承 HibernateDAOSupport提供一个AOP 拦截器。

**46. Spring支持的ORM**

Spring支持以下ORM：

* Hibernate
* iBatis
* JPA (Java Persistence API)
* TopLink
* JDO (Java Data Objects)
* OJB

**47.如何通过HibernateDaoSupport将Spring和Hibernate结合起来？**

用Spring的 SessionFactory 调用 LocalSessionFactory。集成过程分三步：

* 配置the Hibernate SessionFactory。
* 继承HibernateDaoSupport实现一个DAO。
* 在AOP支持的事务中装配。

**48. Spring支持的事务管理类型**

Spring支持两种类型的事务管理：

* **编程式事务管理**：这意味你通过编程的方式管理事务，给你带来极大的灵活性，但是难维护。
* **声明式事务管理：**这意味着你可以将业务代码和事务管理分离，你只需用注解和XML配置来管理事务。

**49. Spring框架的事务管理有哪些优点？**

* 它为不同的事务API 如 JTA，JDBC，Hibernate，JPA 和JDO，提供一个不变的编程模式。
* 它为编程式事务管理提供了一套简单的API而不是一些复杂的事务API如
* 它支持声明式事务管理。
* 它和Spring各种数据访问抽象层很好得集成。

**50. 你更倾向用那种事务管理类型？**

大多数Spring框架的用户选择声明式事务管理，因为它对应用代码的影响最小，因此更符合一个无侵入的轻量级容器的思想。声明式事务管理要优于编程式事务管理，虽然比编程式事务管理（这种方式允许你通过代码控制事务）少了一点灵活性。

### Spring面向切面编程（AOP）

**51. 解释AOP以及Spring aop的原理**

**Aop是面相切面编程，可以说是OOP（面相对象编程）的补充和完善。OOP是定义从上到下的关系集合，但不适合定义从左到右的关系。对于一些和业务无关的操作，比如日志，安全性，异常处理等等这些如果封装在OOP里面会导致代码大量重复。**

**而aop可以将这些行为封装到一个可重用模块（方面），降低代码重复，提高复用性、可维护性、可操作性。Aop的核心思想是将应用程序中的商业逻辑和对其提供支持的通用服务进行分离。**

**实现AOP技术主要分为两大类，一是动态代理，spring默认的策略是目标类是接口，则使用JDK动态代理，否则使用Cglib生成代理。二是静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。**

#### Aspect 切面

AOP核心就是切面，它将多个类的通用行为封装成可重用的模块，该模块含有一组API提供横切功能。比如，一个日志模块可以被称作日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

#### 在Spring AOP 中，关注点和横切关注的区别是什么？

关注点是应用中一个模块的行为，一个关注点可能会被定义成一个我们想实现的一个功能。

横切关注点是一个关注点，此关注点是整个应用都会使用的功能，并影响整个应用，比如日志，安全和数据传输，几乎应用的每个模块都需要的功能。因此这些都属于横切关注点。

#### 连接点

连接点代表一个应用程序的某个位置，在这个位置我们可以插入一个AOP切面，它实际上是个应用程序执行Spring AOP的位置。

#### 通知

通知是个在方法执行前或执行后要做的动作，实际上是程序执行时要通过SpringAOP框架触发的代码段。

Spring切面可以应用五种类型的通知：

* **before**：前置通知，在一个方法执行前被调用。
* **after:**在方法执行之后调用的通知，无论方法执行是否成功。
* **after-returning:**仅当方法成功完成后执行的通知。
* **after-throwing:**在方法抛出异常退出时执行的通知。
* **around:**在方法执行之前和之后调用的通知。

#### 切点

切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

#### 什么是引入?

引入允许我们在已存在的类中增加新的方法和属性。

#### 什么是目标对象?

被一个或者多个切面所通知的对象。它通常是一个代理对象。也指被通知（advised）对象。

#### 什么是代理?

代理是通知目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。

#### 有几种不同类型的自动代理？

BeanNameAutoProxyCreator

DefaultAdvisorAutoProxyCreator

Metadata autoproxying

#### 什么是织入。什么是织入应用的不同点？

织入是将切面和到其他应用类型或对象连接或创建一个被通知对象的过程。

织入可以在编译时，加载时，或运行时完成。

#### 解释基于XML Schema方式的切面实现。

在这种情况下，切面由常规类以及基于XML的配置实现。

#### 解释基于注解的切面实现

在这种情况下(基于@AspectJ的实现)，涉及到的切面声明的风格与带有java5标注的普通java类一致。

### 说说spring框架常用功能？并且谈谈你对spring的理解？

常用功能：

Bean管理

分层通过接口接耦合

AOP实现业务分离（日志记录，异常处理，事务管理，权限管理）

Spring是一个轻量级企业级框架。其核心是IOC和AOP。

IOC即是控制反转，使得我们只需要定义接口，具体的实现类在配置文件中配置，实现了类与类之间的解耦。

AOP就是面向切面编程，可以让我们在不修改源代码的情况下，实现功能的扩展。比如spring的事务，权限管理等。

### Spring底层使用了哪些设计模式？

#### 1.简单工厂模式

spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象

#### 2.工厂方法

通常由应用程序直接使用new创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象。

一般情况下,应用程序有自己的工厂对象来创建bean.如果将应用程序自己的工厂对象交给Spring管理,那么Spring管理的就不是普通的bean,而是工厂Bean。

#### 单例模式

#### 适配器模式

#### 代理模式

#### 模板方法

JDBCTemplate

#### 包装器模式

#### 观察者模式

#### 策略模式

## Hibernate

### Hibernate的事务管理？

Hibernate可以使用JDBC的事务，也可以使用JTA的事务。具体使用哪一种，可以在Hibernate.properties中配置具体的事务工厂类，JDBCTransactionFactory和JTATransactionFactory，如果不配置默认使用JDBC事务。

JDBC模式下，使用

Transaction tx = session.beginTransaction();//方法开启事务。

Tx.commit()//提交事务

Tx.rollback()//回滚事务。

JTA事务：

|  |  |
| --- | --- |
| |  | | --- | | javax.transaction.UserTransaction tx = null;  tx = new initialContext().lookup(” javax.transaction.UserTransaction ”) ;  tx.begin();  Session s1 = sf.openSession();  ……  s1.flush();  s1.close();  Session s2 = sf.openSession();  ……  s2.flush();  s2.close();  tx.commit(); | |

* + - 1. 获得UserTransaction对象
      2. Tx.begin()开启事务
      3. Tx.commit()提交事务
      4. Tx.rollback()回滚事务。

### Hibernate和MyBatis的优缺点？

HibernateDao层开发比MyBatis简单，MyBatis需要维护SQL和结果映射

Hibernate对对象的维护和缓存比MyBatis好，对增删改查的对象的维护要方便。

Hibernate数据库的移植性好。MyBatis不同的数据库要写不同的SQL。

MyBatis门槛低，容易掌握。

MyBatis可以写出更为细致的SQL语句，SQL优化方便 。

### **Hibernate中SessionFactory是线程安全的吗？Session是线程安全的吗（两个线程能够共享同一个Session吗）？**

答：SessionFactory对应Hibernate的一个数据存储的概念，它是线程安全的，可以被多个线程并发访问。SessionFactory一般只会在启动的时候构建。对于应用程序，最好将SessionFactory通过单例模式进行封装以便于访问。Session是一个轻量级非线程安全的对象（线程间不能共享session），它表示与数据库进行交互的一个工作单元。Session是由SessionFactory创建的，在任务完成之后它会被关闭。Session是持久层服务对外提供的主要接口。Session会延迟获取数据库连接（也就是在需要的时候才会获取）。为了避免创建太多的session，可以使用ThreadLocal将session和当前线程绑定在一起，这样可以让同一个线程获得的总是同一个session。Hibernate 3中SessionFactory的getCurrentSession()方法就可以做到。

### **Hibernate中Session的load和get方法的区别是什么？**

答：主要有以下三项区别：   
① 如果没有找到符合条件的记录，get方法返回null，load方法抛出异常。   
② get方法直接返回实体类对象，load方法返回实体类对象的代理。   
③ 在Hibernate 3之前，get方法只在一级缓存中进行数据查找，如果没有找到对应的数据则越过二级缓存，直接发出SQL语句完成数据读取；load方法则可以从二级缓存中获取数据；从Hibernate 3开始，get方法不再是对二级缓存只写不读，它也是可以访问二级缓存的。

**说明：**对于load()方法Hibernate认为该数据在数据库中一定存在可以放心的使用代理来实现延迟加载，如果没有数据就抛出异常，而通过get()方法获取的数据可以不存在。

### **Session的save()、update()、merge()、lock()、saveOrUpdate()和persist()方法分别是做什么的？有什么区别？**

答：Hibernate的对象有三种状态：瞬时态（transient）、持久态（persistent）和游离态（detached），如第135题中的图所示。瞬时态的实例可以通过调用save()、persist()或者saveOrUpdate()方法变成持久态；游离态的实例可以通过调用 update()、saveOrUpdate()、lock()或者replicate()变成持久态。save()和persist()将会引发SQL的INSERT语句，而update()或merge()会引发UPDATE语句。save()和update()的区别在于一个是将瞬时态对象变成持久态，一个是将游离态对象变为持久态。merge()方法可以完成save()和update()方法的功能，它的意图是将新的状态合并到已有的持久化对象上或创建新的持久化对象。对于persist()方法，按照官方文档的说明：① persist()方法把一个瞬时态的实例持久化，但是并不保证标识符被立刻填入到持久化实例中，标识符的填入可能被推迟到flush的时间；② persist()方法保证当它在一个事务外部被调用的时候并不触发一个INSERT语句，当需要封装一个长会话流程的时候，persist()方法是很有必要的；③ save()方法不保证第②条，它要返回标识符，所以它会立即执行INSERT语句，不管是在事务内部还是外部。至于lock()方法和update()方法的区别，update()方法是把一个已经更改过的脱管状态的对象变成持久状态；lock()方法是把一个没有更改过的脱管状态的对象变成持久状态。

### **Hibernate如何实现分页查询？**

答：通过Hibernate实现分页查询，开发人员只需要提供HQL语句（调用Session的createQuery()方法）或查询条件（调用Session的createCriteria()方法）、设置查询起始行数（调用Query或Criteria接口的setFirstResult()方法）和最大查询行数（调用Query或Criteria接口的setMaxResults()方法），并调用Query或Criteria接口的list()方法，Hibernate会自动生成分页查询的SQL语句。

### **阐述实体对象的三种状态以及转换关系。**

答：最新的Hibernate文档中为Hibernate对象定义了四种状态（原来是三种状态，面试的时候基本上问的也是三种状态），分别是：瞬时态（new, or transient）、持久态（managed, or persistent）、游状态（detached）和移除态（removed，以前Hibernate文档中定义的三种状态中没有移除态），如下图所示，就以前的Hibernate文档中移除态被视为是瞬时态。

1. 瞬时态：当new一个实体对象后，这个对象处于瞬时态，即这个对象只是一个保存临时数据的内存区域，如果没有变量引用这个对象，则会被JVM的垃圾回收机制回收。这个对象所保存的数据与数据库没有任何关系，除非通过Session的save()、saveOrUpdate()、persist()、merge()方法把瞬时态对象与数据库关联，并把数据插入或者更新到数据库，这个对象才转换为持久态对象。
2. 持久态：持久态对象的实例在数据库中有对应的记录，并拥有一个持久化标识（ID）。对持久态对象进行delete操作后，数据库中对应的记录将被删除，那么持久态对象与数据库记录不再存在对应关系，持久态对象变成移除态（可以视为瞬时态）。持久态对象被修改变更后，不会马上同步到数据库，直到数据库事务提交。
3. 游离态：当Session进行了close()、clear()、evict()或flush()后，实体对象从持久态变成游离态，对象虽然拥有持久和与数据库对应记录一致的标识值，但是因为对象已经从会话中清除掉，对象不在持久化管理之内，所以处于游离态（也叫脱管态）。游离态的对象与临时状态对象是十分相似的，只是它还含有持久化标识。

### **如何理解Hibernate的延迟加载机制？在实际应用中，延迟加载与Session关闭的矛盾是如何处理的？**

答：延迟加载就是并不是在读取的时候就把数据加载进来，而是等到使用时再加载。Hibernate使用了虚拟代理机制实现延迟加载，我们使用Session的load()方法加载数据或者一对多关联映射在使用延迟加载的情况下从一的一方加载多的一方，得到的都是虚拟代理，简单的说返回给用户的并不是实体本身，而是实体对象的代理。代理对象在用户调用getter方法时才会去数据库加载数据。但加载数据就需要数据库连接。而当我们把会话关闭时，数据库连接就同时关闭了。

延迟加载与session关闭的矛盾一般可以这样处理：   
① 关闭延迟加载特性。这种方式操作起来比较简单，因为Hibernate的延迟加载特性是可以通过映射文件或者注解进行配置的，但这种解决方案存在明显的缺陷。首先，出现"no session or session was closed"通常说明系统中已经存在主外键关联，如果去掉延迟加载的话，每次查询的开销都会变得很大。   
② 在session关闭之前先获取需要查询的数据，可以使用工具方法Hibernate.isInitialized()判断对象是否被加载，如果没有被加载则可以使用Hibernate.initialize()方法加载对象。   
③ 使用拦截器或过滤器延长Session的生命周期直到视图获得数据。Spring整合Hibernate提供的OpenSessionInViewFilter和OpenSessionInViewInterceptor就是这种做法。

### **简述Hibernate常见优化策略。**

答：这个问题应当挑自己使用过的优化策略回答，常用的有：   
① 制定合理的缓存策略（二级缓存、查询缓存）。   
② 采用合理的Session管理机制。   
③ 尽量使用延迟加载特性。   
④ 设定合理的批处理参数。   
⑤ 如果可以，选用UUID作为主键生成器。   
⑥ 如果可以，选用基于版本号的乐观锁替代悲观锁。   
⑦ 在开发过程中, 开启hibernate.show\_sql选项查看生成的SQL，从而了解底层的状况；开发完成后关闭此选项。   
⑧ 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的DBA（数据库管理员）提供支持。

### **锁机制有什么用？简述Hibernate的悲观锁和乐观锁机制。**

答：有些业务逻辑在执行过程中要求对数据进行排他性的访问，于是需要通过一些机制保证在此过程中数据被锁住不会被外界修改，这就是所谓的锁机制。   
Hibernate支持悲观锁和乐观锁两种锁机制。

悲观锁，顾名思义悲观的认为在数据处理过程中极有可能存在修改数据的并发事务（包括本系统的其他事务或来自外部系统的事务）。

处理方法是在修改数据的手给数据加锁，如果加锁失败说明有其他事物在占用。Mysql InnoDB需要关闭自动提交属性。

乐观锁，顾名思义，对并发事务持乐观态度（认为对数据的并发操作不会经常性的发生），通过更加宽松的锁机制来解决由于悲观锁排他性的数据访问对系统性能造成的严重影响。

最常见的乐观锁是通过数据版本标识来实现的，需要给数据所在的表添加一个字段，读取数据时获得数据的版本号，更新数据时将此版本号加1，然后和数据库表对应记录的当前版本号进行比较，如果提交的数据版本号大于数据库中此记录的当前版本号则更新数据，否则认为是过期数据无法更新。

另外一种是使用时间戳，写需要给数据所在表添加一个字段记录时间，提交时检测时间与取出时间戳是否一致，不一致说明数据被修改。

Hibernate中通过Session的get()和load()方法从数据库中加载对象时可以通过参数指定使用悲观锁；而乐观锁可以通过给实体类加整型的版本字段再通过XML或@Version注解进行配置。

**提示：**使用乐观锁会增加了一个版本字段，很明显这需要额外的空间来存储这个版本字段，浪费了空间，但是乐观锁会让系统具有更好的并发性，这是对时间的节省。因此乐观锁也是典型的空间换时间的策略。

### Hibernate的几种查询方式？

sessionFactory.getcurrentSession.save()、get()|load()、delete()、update()

HQL语句：

编写HQL语句

创键Query对象

执行查询，得到结果

QBC查询：

### 简述Hibernate的步骤：

* + - 1. 读取并解析配置文件hibernate.cfg.xml，创建Configuration对象。
      2. 读取并解析映射文件，创建SessionFactory对象。
      3. 打开session
      4. 开启事务。
      5. 完成持久化操作
      6. 提交事务
      7. 关闭session

### Hibernate 的缓存机制

|  |
| --- |
| Hibernate中提供了两级缓存，一级缓存是Session级别的缓存，它属于事务范围的缓存， 该级缓存由hibernate管理，应用程序无需干预；二级缓存是SessionFactory级别的缓 存，该级缓存可以进行配置和更改，并且可以动态加载和卸载，hibernate还为查询结果提 供了一个查询缓存，它依赖于二级缓存；  Hibernate的**一级缓存**由Session提供，只存在于Session的生命周期中，当应用程序调用 Session接口的save(),update(),saveOrupDate(),get(),load()或者Query和Criteria实 例的list(),iterate()等方法时，如果Session缓存中没有相应的对象，hibernate就会把对 象加入到一级缓存中，当session关闭时，该Session所管理的一级缓存也会立即被清除；  **二级缓存**是一个可插拔的缓存插件，它是由SessionFactory负责管理的； 由于SessionFactory对象的生命周期与应用程序的整个过程对应，通常一个应用程序对应 一个SessionFactory，因此，二级缓存是进程范围或者集群范围的缓存； 与一级缓存一样，二级缓存也是根据对象的id来加载与缓存，当执行某个查询获得结果集 为实体对象集时，hibernate就会把它们按照对象id加载到二级缓存中，在访问指定的id的 对象时，首先从一级缓存中查找，找到就直接使用，找不到则转到二级缓存中查找(必须配 置且启用二级缓存)，如果二级缓存中找到，则直接使用，否则会查询数据库，并将查询结 果根据对象的id放到缓存中； |

## MyBatis

### **MyBatis中使用**#**和**$**书写占位符有什么区别？**

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换，#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

“%”#{name}”%”

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

### **解释一下MyBatis中命名空间（namespace）的作用。**

答：在大型项目中，可能存在大量的SQL语句，这时候为每个SQL语句起一个唯一的标识（ID）就变得并不容易了。为了解决这个问题，在MyBatis中，可以为每个映射文件起一个唯一的命名空间，这样定义在这个映射文件中的每个SQL语句就成了定义在这个命名空间中的一个ID。只要我们能够保证每个命名空间中这个ID是唯一的，即使在不同映射文件中的语句ID相同，也不会再产生冲突了。

### **MyBatis中的动态SQL是什么意思？**

答：对于一些复杂的查询，我们可能会指定多个查询条件，但是这些条件可能存在也可能不存在，例如在58同城上面找房子，我们可能会指定面积、楼层和所在位置来查找房源，也可能会指定面积、价格、户型和所在位置来查找房源，此时就需要根据用户指定的条件动态生成SQL语句。如果不使用持久层框架我们可能需要自己拼装SQL语句，还好MyBatis提供了动态SQL的功能来解决这个问题。MyBatis中用于实现动态SQL的元素主要有：   
- if   
- choose / when / otherwise   
- trim   
- where   
- set   
- foreach

下面是映射文件的片段。

<select id="foo" parameterType="Blog" resultType="Blog">  
        select \* from t\_blog where 1 = 1  
        <if test="title != null">  
            and title = #{title}  
        </if>  
        <if test="content != null">  
            and content = #{content}  
        </if>  
        <if test="owner != null">  
            and owner = #{owner}  
        </if>  
   </select>

当然也可以像下面这些书写。

 <select id="foo" parameterType="Blog" resultType="Blog">  
        select \* from t\_blog where 1 = 1   
        <choose>  
            <when test="title != null">  
                and title = #{title}  
            </when>  
            <when test="content != null">  
                and content = #{content}  
            </when>  
            <otherwise>  
                and owner = "owner1"  
            </otherwise>  
        </choose>  
    </select>

再看看下面这个例子。

<select id="bar" resultType="Blog">  
        select \* from t\_blog where id in  
        <foreach collection="array" index="index"   
            item="item" open="(" separator="," close=")">  
            #{item}  
        </foreach>  
    </select>

### MyBatis的执行流程？



1. mybatis配置

SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

1. 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂
2. 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。
3. mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。
4. Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。
5. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
6. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

### Mybatis原理解析？

mybatis的底层原理主要是动态代理和反射技术。动态代理默认使用JDK，如果用到了延迟加载，则会使用CGLIBS。

Mybatis的运行过程主要分为两步：

1.读取配置文件缓存到Configuration对象，并根据该对象创建SqlSessionFactory

2.SqlSession的运行过程。

详细步骤：

1.XMLConfigBuilder读取配置文件，并保存在Configuration对象中（单例）。

2.使用Configuration对象创建SqlSessionFactory，这是一个接口，默认实现类是DefaultSqlSessionFactory。

3.使用SqlSessionFactory构建SqlSession。

SqlSession里面给出了增删改查的方法，不过我们一般使用自己定义的mapper接口。

4.MapperProxyFactory会为mapper接口生成动态代理对象，代理的方法会放到MapperProxy类中。

Configuration作用

1.读入配置文件，包括基础配置的XML文件和映射器文件。

2.初始化基础配置，比如mybatis别名等一些重要的类对象，例如，插件，映射器，ObjectFactory和typeHandler对象

3．提供单例，为后续创建SessionFactory服务并提供配置的参数。

4.执行一些重要的对象方法，初始化配置信息。

映射器的内部组成

MappedStatement，它保存映射器的一个节点（select|insert|delete|update）。包括许多我们配置的sql、sql的Id、缓存信息、resultMap、parameterType、resultType、languageDriver等重要配置内容。

SqlSource 它是提供BoundSql对象的地方，它是MappedStatement的一个属性。

BoundSql 它是建立Sql和参数的地方。它有3个常用的属性：Sql、parameterObject、parameterMappings

### #{}和${}的区别是什么？

注：这道题是面试官面试我同事的。

答：${}是Properties文件中的变量占位符，它可以用于标签属性值和sql内部，属于静态文本替换，比如${driver}会被静态替换为com.mysql.jdbc.Driver。#{}是sql的参数占位符，Mybatis会将sql中的#{}替换为?号，在sql执行前会使用PreparedStatement的参数设置方法，按序给sql的?号占位符设置参数值，比如ps.setInt(0, parameterValue)，#{item.name}的取值方式为使用反射从参数对象中获取item对象的name属性值，相当于param.getItem().getName()。

### Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签？

注：这道题是京东面试官面试我时问的。

答：还有很多其他的标签，<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态sql的9个标签，trim|where|set|foreach|if|choose|when|otherwise|bind等，其中<sql>为sql片段标签，通过<include>标签引入sql片段，<selectKey>为不支持自增的主键生成策略标签。

### 最佳实践中，通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

注：这道题也是京东面试官面试我时问的。

答：Dao接口，就是人们常说的Mapper接口，接口的全限名，就是映射文件中的namespace的值，接口的方法名，就是映射文件中MappedStatement的id值，接口方法内的参数，就是传递给sql的参数。Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MappedStatement，举例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到namespace为com.mybatis3.mappers.StudentDao下面id = findStudentById的MappedStatement。在Mybatis中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个MappedStatement对象。

Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

Dao接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。

### Mybatis是如何进行分页的？分页插件的原理是什么？

答：Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页，可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

### 举例：select \* from student，拦截sql后重写为：select t.\* from （select \* from student）t limit 0，10简述Mybatis的插件运行原理，以及如何编写一个插件。

答：Mybatis仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

### 实现Mybatis的Interceptor接口并复写intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。Mybatis执行批量插入，能返回数据库主键列表吗？

答：能，JDBC都能，Mybatis当然也能。

### Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理不？

答：Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态拼接sql的功能，Mybatis提供了9种动态sql标签trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能。

### Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？

答：第一种是使用<resultMap>标签，逐一定义列名和对象属性名之间的映射关系。第二种是使用sql列的别名功能，将列别名书写为对象属性名，比如T\_NAME AS NAME，对象属性名一般是name，小写，但是列名不区分大小写，Mybatis会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成T\_NAME AS NaMe，Mybatis一样可以正常工作。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

### Mybatis能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别。

答：能，Mybatis不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把selectOne()修改为selectList()即可；多对多查询，其实就是一对多查询，只需要把selectOne()修改为selectList()即可。

关联对象查询，有两种实现方式，一种是单独发送一个sql去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用join查询，一部分列是A对象的属性值，另外一部分列是关联对象B的属性值，好处是只发一个sql查询，就可以把主对象和其关联对象查出来。

那么问题来了，join查询出来100条记录，如何确定主对象是5个，而不是100个？其去重复的原理是<resultMap>标签内的<id>子标签，指定了唯一确定一条记录的id列，Mybatis根据<id>列值来完成100条记录的去重复功能，<id>可以有多个，代表了联合主键的语意。

同样主对象的关联对象，也是根据这个原理去重复的，尽管一般情况下，只有主对象会有重复记录，关联对象一般不会重复。

举例：下面join查询出来6条记录，一、二列是Teacher对象列，第三列为Student对象列，Mybatis去重复处理后，结果为1个老师6个学生，而不是6个老师6个学生。

       t\_id    t\_name           s\_id

|          1 | teacher      |      38 |  
|          1 | teacher      |      39 |  
|          1 | teacher      |      40 |  
|          1 | teacher      |      41 |  
|          1 | teacher      |      42 |  
|          1 | teacher      |      43 |

### Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？

答：Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

### Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？

注：我出的。

答：不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；毕竟namespace不是必须的，只是最佳实践而已。

原因就是namespace+id是作为Map<String, MappedStatement>的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

### Mybatis中如何执行批处理？

注：我出的。

答：使用BatchExecutor完成批处理。

### Mybatis都有哪些Executor执行器？它们之间的区别是什么？

注：我出的

答：Mybatis有三种基本的Executor执行器，**SimpleExecutor、ReuseExecutor、BatchExecutor。**

**SimpleExecutor：**每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

**ReuseExecutor：**执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

**BatchExecutor：**执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

### Mybatis中如何指定使用哪一种Executor执行器？

答：在Mybatis配置文件中，可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数。

### Mybatis是否可以映射Enum枚举类？

答：Mybatis可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和getResult()接口方法。TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占位符参数和获取列查询结果。

### Mybatis映射文件中，如果A标签通过include引用了B标签的内容，请问，B标签能否定义在A标签的后面，还是说必须定义在A标签的前面？

答：虽然Mybatis解析Xml映射文件是按照顺序解析的，但是，被引用的B标签依然可以定义在任何地方，Mybatis都可以正确识别。

原理是，Mybatis解析A标签，发现A标签引用了B标签，但是B标签尚未解析到，尚不存在，此时，Mybatis会将A标签标记为未解析状态，然后继续解析余下的标签，包含B标签，待所有标签解析完毕，Mybatis会重新解析那些被标记为未解析的标签，此时再解析A标签时，B标签已经存在，A标签也就可以正常解析完成了。

### 简述Mybatis的Xml映射文件和Mybatis内部数据结构之间的映射关系？

答：Mybatis将所有Xml配置信息都封装到All-In-One重量级对象Configuration内部。在Xml映射文件中，<parameterMap>标签会被解析为ParameterMap对象，其每个子元素会被解析为ParameterMapping对象。<resultMap>标签会被解析为ResultMap对象，其每个子元素会被解析为ResultMapping对象。每一个<select>、<insert>、<update>、<delete>标签均会被解析为MappedStatement对象，标签内的sql会被解析为BoundSql对象。

### 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

答：Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

### sqlSession线程安全吗？

不安全。

## Activiti工作流

### 什么是工作流？

具体的概念不是很好懂，还是直接举一个例子吧。

比如请假流程，我现在要请假，首先需要项目经理批准，然后部门经理批准，人事批准，老板批准。期间任何一个环节出了问题，否决我的请假申请，这个流程都执行不下去。

### Activiti工作流的核心对象是什么？

ProcessEngine对象，调用service，从而操作数据库中的表。

### 工作流中RepositoryService、RuntimeService、TaskService、HistoryService分别表示什么操作

|  |  |
| --- | --- |
| **RepositoryService** | **管理流程定义** |
| **RuntimeService** | **执行管理，包括启动、推进、删除流程实例等操作** |
| **TaskService** | **任务管理** |

|  |  |
| --- | --- |
| HistoryService | 历史管理(执行完的数据的管理) |

### 什么是流程定义？什么是流程实例？什么是执行对象？

流程定义就是我们自己定义的一个工作流程，流程实例是流程定义的一次执行过程，执行对象就是按照流程定义的规则执行一次的操作，一个流程中，执行对象可以有多个

### 流程变量在项目中的作用

 \* 1：用来传递业务参数，目的就是审核人可以通过流程变量查看申请人的一些审核信息  
   2：在连线的condition中设置流程变量，用来指定应该执行的连线${message=='重要'}

   3：使用流程变量指定个人任务和组任务的办理人#{userID}

### 6.activiti工作流中，排他网关和并行网关有什么区别？

排他网关只有一个分支执行。

并行网关，可以同时开启多条运行的流程实例同时进行。而且只有所有并行网关的分支流程全部执行完毕，流程汇聚，当前任务才完成。

### 7.个人任务和组任务的查询一样吗？

\* 不一样  
  \* 都是用TaskService完成（TaskService.createTasQuery）  
  \* 个人任务（taskAssgnee），组任务（taskCandidateUser）  
  \* [**数据库**](http://lib.csdn.net/base/mysql)存放，个人任务（类型：参与），组任务（类型，参与，候选）

## Shiro

### Shiro 的优点

  > 简单的身份认证, 支持多种数据源  
  > 对角色的简单的授权, 支持细粒度的授权(方法级)  
  > 支持一级缓存，以提升应用程序的性能，Ehceche；  
  > 内置的基于 POJO 企业会话管理, 适用于 Web 以及非 Web 的环境  
  > 非常简单的加密 API  
  > 不跟任何的框架或者容器捆绑, 可以独立运行。

### 简述 Shiro 的核心组件

 Shiro 架构 3 个核心组件:  
    > Subject: 正与系统进行交互的人, 或某一个第三方服务.   
    所有 Subject 实例都被绑定到（且这是必须的）一个SecurityManager 上。  
    > SecurityManager: Shiro 架构的心脏, 用来协调内部各安全组件, 管理内部组件实例, 并通过它来提供安全管理的各种服务.   
    当 Shiro 与一个 Subject 进行交互时, 实质上是幕后的 SecurityManager 处理所有繁重的 Subject 安全操作。  
    > Realms: 本质上是一个特定安全的 DAO. 当配置 Shiro 时, 必须指定至少一个 Realm 用来进行身份验证和/或授权.   
    Shiro 提供了多种可用的 Realms 来获取安全相关的数据. 如关系数据库(JDBC), INI 及属性文件等.   
    可以定义自己 Realm 实现来代表自定义的数据源。

### Shiro认证过程

1. . 应用程序代码调用 Subject.login 方法，传递创建好的包含终端用户的 Principals(身份)和 Credentials(凭证)的 AuthenticationToken 实例  
     ②. Subject 实例: 通常为 DelegatingSubject(或子类)委托应用程序的 SecurityManager 通过调用securityManager.login(token) 开始真正的验证。  
     ③. SubjectManager 接收 token，调用内部的 Authenticator 实例调用 authenticator.authenticate(token).Authenticator 通常是一个 ModularRealmAuthenticator 实例, 支持在身份验证中协调一个或多个Realm 实例  
     ④. 如果应用程序中配置了一个以上的 Realm, ModularRealmAuthenticator 实例将利用配置好的AuthenticationStrategy 来启动 Multi-Realm 认证尝试. 在Realms 被身份验证调用之前, 期间和以后,AuthenticationStrategy 被调用使其能够对每个Realm 的结果作出反应.  
     ⑤. 每个配置的 Realm 用来帮助看它是否支持提交的 AuthenticationToken. 如果支持, 那么支持 Realm 的 getAuthenticationInfo 方法将会伴随着提交的 token 被调用. getAuthenticationInfo 方法有效地代表一个特定 Realm 的单一的身份验证尝试。

### Shiro授权过程

1. . 应用程序或框架代码调用任何 Subject 的hasRole\*, checkRole\*, isPermitted\*,或者checkPermission\*方法的变体, 传递任何所需的权限  
     ②. Subject 的实例—通常是 DelegatingSubject(或子类), 调用securityManager 的对应的方法.   
     ③. SecurityManager 调用 org.apache.shiro.authz.Authorizer 接口的对应方法.默认情况下，authorizer 实例是一个 ModularRealmAuthorizer 实例, 它支持协调任何授权操作过程中的一个或多个Realm 实例  
     ④. 每个配置好的 Realm 被检查是否实现了相同的 Authorizer 接口. 如果是, Realm 各自的 hasRole\*, checkRole\*,isPermitted\*，或 checkPermission\* 方法将被调用。

### 如何配置在 Spring 中配置使用 Shiro

1. . 在 web.xml 中配置 Shiro 的 Filter  
   ②. 在 Spring 的配置文件中配置  Shiro:  
       > 配置自定义 Realm：实现自定义认证和授权  
       > 配置 Shiro 实体类使用的缓存策略  
       > 配置 SecurityManager  
       > 配置保证 Shiro 内部 Bean 声明周期都得到执行的 Lifecycle Bean 后置处理器  
       > 配置AOP 式方法级权限检查   
       > 配置 Shiro Filter

### 如何在html静态页面根据权限动态展示按钮，页面？

面试官问我的这个问题，回来想一想，有些问题。

你要获取权限，首先你就需要登录，登录之后，根据你权限的不同，来生成不同的静态页面。

如果你没有登录，由于过滤器的存在，那么我直接不显示。

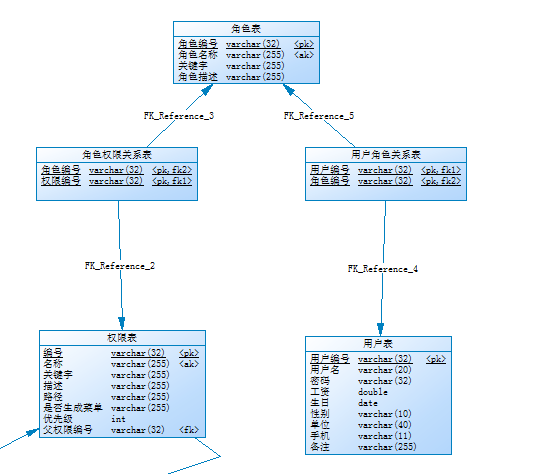
自己使用shiro加freemarker实现了控制到按钮级别的页面

|  |
| --- |
| @Component public class ShiroTag {   */\*\*  \* 是否拥有该权限  \** ***@param*** *permission 权限标识  \** ***@return*** *true：是 false：否  \*/* public boolean hasPermission(String permission) {  Subject subject = SecurityUtils.*getSubject*();  return subject != null && subject.isPermitted(permission);  }  }  */\*\*  \* 自定义Shiro标签  \*  \** ***@param*** *shiroTag  \** ***@return*** *\*/* @Bean public FreeMarkerConfigurer freeMarkerConfigurer(ShiroTag shiroTag) {  FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();  configurer.setTemplateLoaderPath("classpath:/templates/");   Map<String, Object> variables = new HashMap<>(1);  variables.put("shiro", shiroTag);  configurer.setFreemarkerVariables(variables);   Properties settings = new Properties();  settings.setProperty("default\_encoding", "utf-8");  settings.setProperty("number\_format", "0.##");  configurer.setFreemarkerSettings(settings);   return configurer; } |

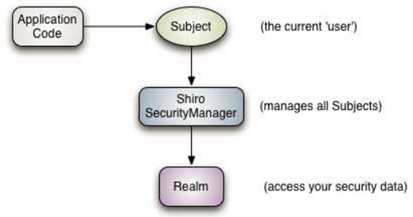
### Shiro权限控制模型有几张表？

5张表：

权限表—权限角色表--角色表—用户角色--用户表



### Shiro进行权限控制的流程



Application Code：应用程序代码，由开发人员负责开发

Subject：当前用户

SecurityManager：安全管理器，管理所有的用户，认证、授权等。

Realm：安全数据桥，类似于Dao，负责访问安全数据

应用程序的代码，需要通过subject（当前用户）来进行认证和授权，而subject又委托SecurityManager（安全管理器），而安全管理器需要从Realm获取相应的用户、角色权限进行比较以确定用户身份是否合法。

### Shiro提供权限控制的方式

* url拦截进行权限控制----框架提供了很多过滤器进行url拦截
* 方法注解进行权限控制----框架提供了在方法上使用的注解，为Action创建代理对象，进行权限控制
* 页面标签权限控制----框架提供一套标签，用于根据权限展示或者隐藏按钮或者菜单
* 代码级别权限控制-----框架提供API，使用编程方式进行权限控制(不建议使用)

## 8．ActiveMQ

### ActiveMQ漏接消息怎么办？

重试。

创建消息表，定期扫描未完成的消息，重新发送。

### 项目中什么地方使用了ActiveMq？

添加商品同步索引库。发送消息保证数据及时同步。

## 9. Redis

### 说说redis集群的搭建？

1.就说一个最简单的集群吧，redis集群由于存在一个投票机制，所以至少需要3个节点，并且为了保证高可用，每个节点需要一个备用的节点，所以至少需要6台服务器。

2.将6个redis实例发布到服务器上面，如果是分布式或者伪分布式，还需要更改一下端口号（7001-7006）。

3.修改redis.conf配置文件，将luster-enabled yes注释打开，使用redis集群管理工具redis-trib将节点连接起来。由于redis-trib是ruby脚本，所以还需要安装ruby的环境。

4.之后使用redis-cli 7002 -c连接集群就可以了。

### Redis里面有几种数据类型？

String：key-value

Hash ：key-fields-values 对key进行分类。

List ：链表，有顺序可重复，做队列，定时任务。取出数据之后，数据会消失

Set ：无顺序，不能重复，可以做共同关注

SortedSet：有顺序，不能重复，做排行榜。

### Redis客户端的使用？

使用jedis对象调用get set方法即可

单击版：Jedis，JedisPool（带连接池）

集群版：JedisCluster

### redis持久化的几种方式

　　1、快照（snapshots）默认

**缺省情况情况下，Redis把数据快照存放在磁盘上的二进制文件中，文件名为dump.rdb。你可以配置Redis的持久化策略，例如数据集中每N秒钟有超过M次更新，就将数据写入磁盘；或者你可以手工调用命令SAVE或BGSAVE。**

**工作原理**

　　． Redis forks.

　　． 子进程开始将数据写到临时RDB文件中。

　　． 当子进程完成写RDB文件，用新文件替换老文件。

　　． 这种方式可以使Redis使用copy-on-write技术。

**2、AOF**

**快照模式并不十分健壮，当系统停止，或者无意中Redis进程被kill掉，最后写入Redis的数据就会丢失。这对某些应用也许不是大问题，但对于要求高可靠性的应用来说，**

**Append-only文件模式是另一种选择。**把所有对redis数据库操作的命令,增删改操作,保存到一个文件中.数据库恢复的时候,只需要将所有的命令执行一遍就可以了.

**你可以在配置文件中打开AOF模式**

**3、虚拟内存方式**

　　4、diskstore方式

### mySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？

1.MySQL可以通过使用时间戳对数据表进行分区，将热点数据聚集到一个分区，然后Redis只从这个表的分区中取数据

2.redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。redis 提供 6种数据淘汰策略：  
volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰  
volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰  
volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰  
allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰  
allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰  
no-enviction（驱逐）：禁止驱逐数据

### 使用[Redis](http://lib.csdn.net/base/redis)有哪些好处？

(1) 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

(2) 支持丰富数据类型，支持string，list，set，sorted set，hash

(3) 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

(4) 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

5.缓解服务器的压力，比如一个商品需要经常被查询，如果每次查询都操作数据库，非常慢，这时可以将这个商品放入缓存中，查询的时候直接从缓存中获取就行了。

### redis常见性能问题和解决方案：

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变

### Redis存了10w条数据，存的时候没有考虑顺序，取出来的时候想要根据顺序取出来怎么办？

List或者SortedSet

### **Memcache与Redis的区别都有哪些？**

1)、存储方式

Memcache把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。

Redis有部份存在硬盘上，这样能保证数据的持久性。

2)、数据支持类型

Memcache对数据类型支持相对简单。

Redis有复杂的数据类型。

3)、使用底层模型不同

它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。

Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

4），value大小

redis最大可以达到1GB，而memcache只有1MB

### **,redis 最适合的场景**

Redis最适合所有数据in-momory的场景，虽然Redis也提供持久化功能，但实际更多的是一个disk-backed的功能，跟传统意义上的持久化有比较大的差别，那么可能大家就会有疑问，似乎Redis更像一个加强版的Memcached，那么何时使用Memcached,何时使用Redis呢?

       如果简单地比较Redis与Memcached的区别，大多数都会得到以下观点：

     1 、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等[**数据结构**](http://lib.csdn.net/base/datastructure)的存储。  
     2 、Redis支持数据的备份，即master-slave模式的数据备份。  
     3 、Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

### （1）、会话缓存（Session Cache）

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？

幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

### （2）、全页缓存（FPC）

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似[**PHP**](http://lib.csdn.net/base/php)本地FPC。

再次以Magento为例，Magento提供一个插件来使用Redis作为[全页缓存后端](https://github.com/colinmollenhour/Cm_Cache_Backend_Redis)。

此外，对WordPress的用户来说，Pantheon有一个非常好的插件  [wp-redis](https://wordpress.org/plugins/wp-redis/)，这个插件能帮助你以最快速度加载你曾浏览过的页面。

### （3）、队列

Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如[**Python**](http://lib.csdn.net/base/python)）对 list 的 push/pop 操作。

如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从[这里](http://celery.readthedocs.org/en/latest/getting-started/brokers/redis.html)去查看。

### （4），排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user\_scores”，我们只需要像下面一样执行即可：

当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：

ZRANGE user\_scores 0 10 WITHSCORES

Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

### （5）、发布/订阅

最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

Redis提供的所有特性中，我感觉这个是喜欢的人最少的一个，虽然它为用户提供如果此多功能。

### **请用Redis和任意语言实现一段恶意登录保护的代码，限制1小时内每用户Id最多只能登录5次。具体登录函数或功能用空函数即可，不用详细写出。**

用户登录之后，在redis里面存用户信息，并且增加一个登录次数的字段，设置有效时间，如果用户在有效时间内重复登录次数达到五次，则不让登录。

### redis的并发竞争问题如何解决?

　　Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念，Redis对于多个客户端连接并不存在竞争，但是在Jedis客户端对Redis进行并发访问时会发生连接超时、数据转换错误、阻塞、客户端关闭连接等问题，这些问题均是由于客户端连接混乱造成。对此有2种解决方法：

　　1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

　　2.服务器角度，利用setnx实现锁。

　　对于第一种，需要应用程序自己处理资源的同步，可以使用的方法比较通俗，可以使用synchronized也可以使用lock；第二种需要用到Redis的setnx命令，但是需要注意一些问题。

### redis事物的了解CAS(check-and-set 操作实现乐观锁 **)?**

　　和众多其它数据库一样，Redis作为NoSQL数据库也同样提供了事务机制。在Redis中，MULTI/EXEC/DISCARD/WATCH这四个命令是我们实现事务的基石。相信对有关系型数据库开发经验的开发者而言这一概念并不陌生，即便如此，我们还是会简要的列出Redis中事务的实现特征：

　　1). 在事务中的所有命令都将会被串行化的顺序执行，事务执行期间，Redis不会再为其它客户端的请求提供任何服务，从而保证了事物中的所有命令被原子的执行。

　　2). 和关系型数据库中的事务相比，在Redis事务中如果有某一条命令执行失败，其后的命令仍然会被继续执行。

　　3). 我们可以通过MULTI命令开启一个事务，有关系型数据库开发经验的人可以将其理解为"BEGIN TRANSACTION"语句。在该语句之后执行的命令都将被视为事务之内的操作，最后我们可以通过执行EXEC/DISCARD命令来提交/回滚该事务内的所有操作。这两个Redis命令可被视为等同于关系型数据库中的COMMIT/ROLLBACK语句。

　　4). 在事务开启之前，如果客户端与服务器之间出现通讯故障并导致网络断开，其后所有待执行的语句都将不会被服务器执行。然而如果网络中断事件是发生在客户端执行EXEC命令之后，那么该事务中的所有命令都会被服务器执行。

　　5). 当使用Append-Only模式时，Redis会通过调用系统函数write将该事务内的所有写操作在本次调用中全部写入磁盘。然而如果在写入的过程中出现系统崩溃，如电源故障导致的宕机，那么此时也许只有部分数据被写入到磁盘，而另外一部分数据却已经丢失。Redis服务器会在重新启动时执行一系列必要的一致性检测，一旦发现类似问题，就会立即退出并给出相应的错误提示。此时，我们就要充分利用Redis工具包中提供的redis-check-aof工具，该工具可以帮助我们定位到数据不一致的错误，并将已经写入的部分数据进行回滚。修复之后我们就可以再次重新启动Redis服务器了。

### WATCH命令和基于CAS的乐观锁：

　　在Redis的事务中，WATCH命令可用于提供CAS(check-and-set)功能。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Null multi-bulk应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

　　val = GET mykey

　　val = val + 1

　　SET mykey $val

　　以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

　　WATCH mykey

　　val = GET mykey

　　val = val + 1

　　MULTI

　　SET mykey $val

　　EXEC

　　和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

### redis的缓存失效策略和主键失效机制

　　作为缓存系统都要定期清理无效数据，就需要一个主键失效和淘汰策略.

　　在Redis当中，有生存期的key被称为volatile。在创建缓存时，要为给定的key设置生存期，当key过期的时候（生存期为0），它可能会被删除。

**1、影响生存时间的一些操作**

　　生存时间可以通过使用 DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据，也就是说，修改key对应的value和使用另外相同的key和value来覆盖以后，当前数据的生存时间不同。

　　比如说，对一个 key 执行INCR命令，对一个列表进行LPUSH命令，或者对一个哈希表执行HSET命令，这类操作都不会修改 key 本身的生存时间。另一方面，如果使用RENAME对一个 key 进行改名，那么改名后的 key的生存时间和改名前一样。

　　RENAME命令的另一种可能是，尝试将一个带生存时间的 key 改名成另一个带生存时间的 another\_key ，这时旧的 another\_key (以及它的生存时间)会被删除，然后旧的 key 会改名为 another\_key ，因此，新的 another\_key 的生存时间也和原本的 key 一样。使用PERSIST命令可以在不删除 key 的情况下，移除 key 的生存时间，让 key 重新成为一个persistent key 。

**2、如何更新生存时间**

　　可以对一个已经带有生存时间的 key 执行EXPIRE命令，新指定的生存时间会取代旧的生存时间。过期时间的精度已经被控制在1ms之内，主键失效的时间复杂度是O（1），

　　EXPIRE和TTL命令搭配使用，TTL可以查看key的当前生存时间。设置成功返回 1；当 key 不存在或者不能为 key 设置生存时间时，返回 0 。

**最大缓存配置**

　　在 redis 中，允许用户设置最大使用内存大小

　　server.maxmemory

　　默认为0，没有指定最大缓存，如果有新的数据添加，超过最大内存，则会使redis崩溃，所以一定要设置。redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

**redis 提供 6种数据淘汰策略：**

　　． volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

　　． volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

　　． volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

　　． allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

　　． allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

　　． no-enviction（驱逐）：禁止驱逐数据

　　注意这里的6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略。

**使用策略规则：**

　　1、如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru

　　2、如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random

**三种数据淘汰策略：**

　　ttl和random比较容易理解，实现也会比较简单。主要是Lru最近最少使用淘汰策略，设计上会对key 按失效时间排序，然后取最先失效的key进行淘汰

### redis主从之间的数据共享问题

主要是通过master server持久化的rdb文件实现的。master server 先dump出内存快照文件，然后将rdb文件传给slave server，slave server 根据rdb文件重建内存表。

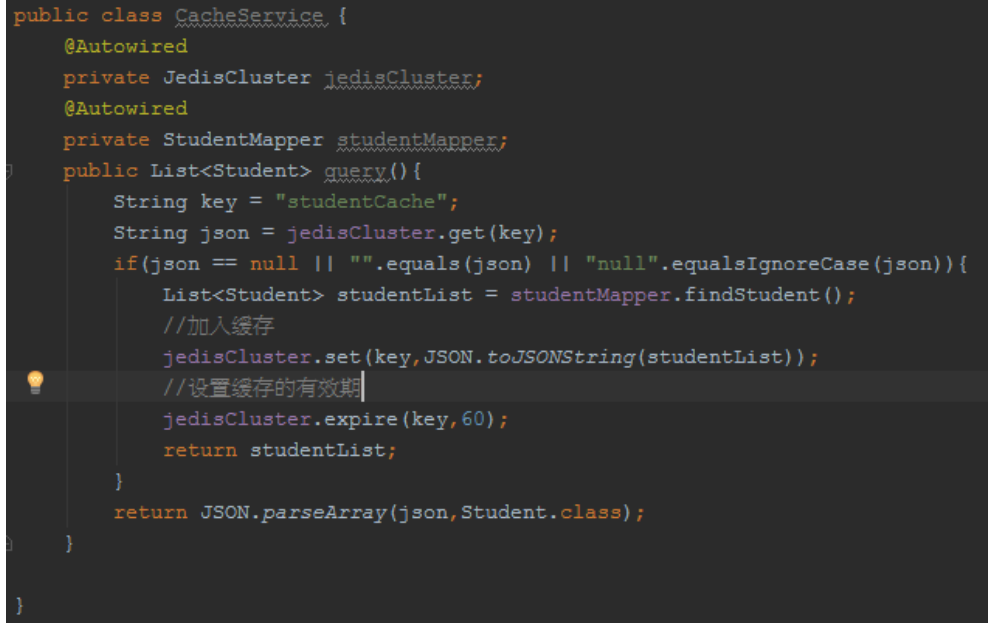
|  |
| --- |
| redis复制过程如下： 1、slave server启动连接到master server之后，salve server主动发送SYNC命令给master server 2、master server接受SYNC命令之后，判断，是否有正在进行内存快照的子进程，如果有，则等待其结束，否则，fork一个子进程，子进程把内存数据保存为文件，并发送给slave server 3、master server子进程进程做数据快照时，父进程可以继续接收client端请求写数据，此时，父进程把新写入的数据放到待发送缓存队列中 4、slave server 接收内存快照文件之后，清空内存数据，根据接收的快照文件，重建内存表数据结构 5、master server把快照文件发送完毕之后，发送缓存队列中保存的子进程快照期间改变的数据给slave server，slave server做相同处理，保存数据一致性 6、master server 后续接收的数据，都会通过步骤1建立的连接，把数据发送到slave server 需要注意：slave server如果因为网络或其他原因断与master server的连接，当slave server重新连接时，需要重新获取master server的内存快照文件，slave server的数据会自动全部清空，然后再重新建立内存表，这样会让slave server 启动恢复服务比较慢，同时也给master server带来较大压力，可以看出redis的复制没有增量复制的概念，这是redis主从复制的一个主要弊端，在实际环境中，尽量规避中途增加从库 redis2.8之前不支持增量，到2.8之后就支持增量了！ |

### 使用Redis解决分布式session共享问题

使用redis模拟session，key是sessionId，value是session的信息。客户每次发送请求都会从cookie中将sessionid带到服务器，根据sessionId从redis中回去session信息

### 关于缓存击穿

#### 1.缓存击穿怎么形成的。



上面的代码，是一个典型的写法：当查询的时候，先从Redis集群中取，如果没有，那么再从DB中查询并设置到Redis集群中。假设从DB中查询需要2S，那么显然这段时间内过来的请求，在上述的代码下，会全部走DB查询，相当于缓存被直接穿透，这样的现象就称之为“缓存击穿”！

#### 2.解决方案



这种思想，类似于单例模式的双检索。

## Solr面试题

### 做搜索模块遇到过哪些问题？

1.时间格式化问题。

Solr使用的是标准的格林威治时间，北京在东8区，默认时间会-8小时。

解决：

为了满足-8，创建索引格式化日期类型的时候，判断下，把时间+8

### solr怎么设置搜索结果排名靠前（得分）？

可以设置文档中域的boost值，boost值越高计算出来的相关度得分就越高，排名也就越靠前。此方法可以把热点商品或者是推广商品的排名提高。

### solr是什么？solr和lucene有什么区别？

Solr 是Apache下的一个顶级开源项目，采用Java开发，它是基于Lucene的全文搜索服务器。Solr提供了比Lucene更为丰富的查询语言，同时实现了可配置、可扩展，并对索引、搜索性能进行了优化。

区别：

Lucene是一个开放源代码的全文检索引擎工具包，它不是一个完整的全文检索引擎，Lucene提供了完整的查询引擎和索引引擎，目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者以Lucene为基础构建全文检索引擎。

Solr的目标是打造一款企业级的搜索引擎系统，它是一个搜索引擎服务，可以独立运行，通过Solr可以非常快速的构建企业的搜索引擎，通过Solr也可以高效的完成站内搜索功能。

### 对于Solr都用过哪些功能，Solr集群主要是怎么搭建的？版本变化情况等等。

单机版：

Solr是一个war包，直接部署在tomcat里面就可以了，部署之后，需要修改solr工程的web.xml文件，指定一下solrhome目录用来存放索引文件。

使用solrJ：

首先需要在schema.xml中定义业务域和数据库中的字段进行对应。

* + - 1. 添加文档、删除文档：

solrServer.add(SolrInputDocument document)

solrServer.deleteById(String id)

solrServer.deleteByQuery(Query query)

* + - 1. 查询索引库

Query.setQuery()

使用solrServer.query(SolrQuery query)

* + - 1. 导入商品到索引库

查询所有的商品，为每个商品创建一个SolrInputDocument对象，并且给文档添加域addField(“id”,”searchItem.getId()”)

使用solrServer.add(document)添加每一个文档到索引库。

集群版：

SolrCloud是基于solr和zookeeper的分布式搜索方案，使用zookeeper来作为集群的配置管理中心，也是SolrCloud的入口，为了实现入口的高可用，需要搭建zookeeper集群，需要三个zookeeper。

SolrCloud至少需要分为两片（shard），每片有一个主节点，一个从节点，需要4台服务器。

要搭建SolrCloud的话，首先需要搭建一个zookeeper集群，修改配置文件中数据保存的目录，然后加上server.1、server.2、server.3+一个zookeeper节点的地址。启动每一个zookeeper实例就可以了。

搭建solr集群，将solr工程复制到4个tomcat，指定每一个工程的solrhome，修改每一个solr工程的web.xml，将4个solrhome关联起来。每一个solrhome里面都有一个solr.xml，配一下ip以及端口号。

将solr目录中的conf目录上传到zookeeper统一管理。

修改tomcet/bin的catalina.sh文件，关联solr和zookeeper。

集群版的使用：

使用CloudSolrServer对象，指定一个默认的collection，其他的和solr的单机版一样

在spring中的配置：

<!-- 单机版solr服务配置 -->

<!-- <bean id="httpSolrServer" class="org.apache.solr.client.solrj.impl.HttpSolrServer">

<constructor-arg name="baseURL" value="http://192.168.25.154:8080/solr"></constructor-arg>

</bean> -->

<!-- 集群版solr服务 -->

<bean id=*"cloudSolrServer"* class=*"org.apache.solr.client.solrj.impl.CloudSolrServer"*>

<constructor-arg name=*"zkHost"* value=*"192.168.25.154:2181,192.168.25.154:2182,192.168.25.154:2183"*></constructor-arg>

<property name=*"defaultCollection"* value=*"collection2"*></property>

</bean>

### Solr集群怎么搭建？

SolrCloud是基于solr和zookeeper的分布式搜索方案，使用zookeeper来作为集群的配置管理中心，也是SolrCloud的入口，为了实现入口的高可用，首先需要搭建zookeeper集群，需要三个zookeeper。Zookeeper集群搭建比较复杂，不说了。

SolrCloud至少需要分为两片（shard），每片都有两个solrCore：一个主节点，一个从节点，所以至少需要4台服务器。

1.关联solrCore：

搭建solr集群，将solr工程复制到4个tomcat，指定每一个工程的solrhome，修改每一个solr工程的web.xml，将4个solrhome关联起来。每一个solrhome里面都有一个solr.xml，配一下ip以及端口号。

2.将配置文件上传到zookeeper，关联solrCloud和zookeeper。

将solr目录中的conf目录上传到zookeeper统一管理。

修改tomcet/bin的catalina.sh文件，关联solr和zookeeper。

SolrCloud的操作，使用CloudSolrServer。

Zookeeper在solrCloud中的作用：

* 1. 容错、负载均衡。
  2. solr配置文件的集中管理
  3. solrCloud的入口

### SolrCloud的Query过程，怎么进行Query？怎么计算Score，汇总返回的流程。

### 添加一个商品后，怎么实时同步到索引库？

在将商品添加到数据库中后，Producer给activiteMQ发送一个消息，将商品ID发送过去。然后Consumer接受消息，取出ID，根据ID查询商品，给商品建立一个文档，添加到索引库。

### 怎么发送消息？怎么接受消息？

发送消息：

JmsTemplate.send()方法，里面需要传一个主题的目的地和message。主题目的地在交给spring管理，消息的话，直接调用new MessageCreate获得messageCreate对象，调用createMessage（）方法获得。

接受消息：

创建一个类继承MessageListener，重写里面的onMeassage方法。

### 搜索系统用什么做的？

Solr

### 简单介绍一下solr？

Solr是一个基于lucence的搜索引擎服务器，可以独立运行在服务器。我们使用solr的话，首先将solr搭建起来，然后在sechma.xml中将业务域和数据库中表的字段进行映射，然后只需要将需要搜索的商品建立文档，将这些文档添加到索引库，然后调用api查询索引就可以了。

### 涛涛项目中，solr的流程：

1. 搭建好solr服务器，后台有一个按钮，一键导入商品到索引库，在业务层查询所有商品，遍历集合，为每一个商品创建一个文档(SolrInputDocument对象)，使用solrServer.add(document)添加文档到索引库。
2. 查询索引库，使用solrServer.query（Query query），需要SolrServer和SolrQuery对象，接受查询条件，调用到查询。

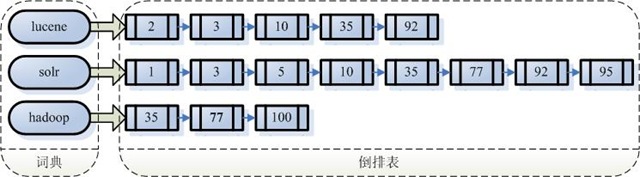
### Solr的索引是一个什么样的索引？

倒排索引。

注意：创建索引是对语汇单元索引，通过词语找文档，这种索引的结构叫**倒排索引结构**。

传统方法是根据文件找到该文件的内容，在文件内容中匹配搜索关键字，这种方法是顺序扫描方法，数据量大、搜索慢。

**倒排索引结构**是根据内容（词语）找文档，如下图：



**倒排索引结构也叫反向索引结构，包括索引和文档两部分，索引即词汇表，它的规模较小，而文档集合较大。**

### 添加商品同步到索引库的时候消息漏接的问题

添加商品的时候，要给ActiveMQ发送消息，传递参数商品id过去。那么这个消息ActiveMQ没有接受到怎么办？这个时候我的数据已经持久化到数据库了？

1.如果没有添加到索引库，那就阻塞好了，不让添加到数据库。

2.创建消息表，定时扫描异常状态的消息，重新发送。

## 10.FreeMarker

### 除了FreeMarker，还有什么框架可以做静态化？

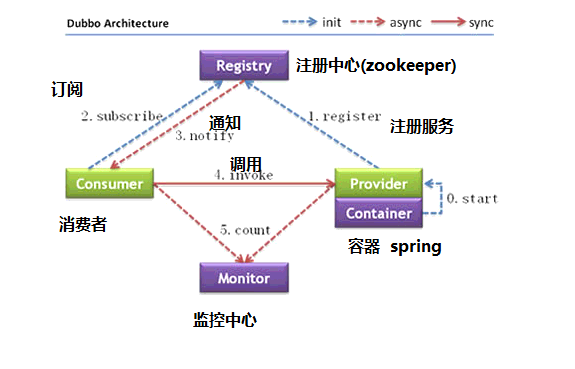
## 11.Dubbo

### WebService和dubbo的区别和联系？

1. Webservice：效率不高基于soap协议。项目中不推荐使用。
2. 使用restful形式的服务：http+json。很多项目中应用。如果服务太多，服务之间调用关系混乱，需要治疗服务。

3、使用dubbo。使用rpc协议进行远程调用，直接使用socket通信。传输效率高，并且可以统计出系统之间的调用关系、调用次数。

### Dubbo主要解决服务器之间的通信问题。



架构：

服务提供者在注册中心注册服务，消费者从注册中心订阅服务。还有一个用于统计服务的调用次数和调用时间的监控中心。

## 12.Zookeeper

## zookeeper的底层？使用场景？如何实现zookeeper的分布式锁？

ZooKeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，它提供了一项基本服务：**分布式锁服务**。由于ZooKeeper的开源特性，后来我们的开发者在分布式锁的基础上，摸索了出了其他的使用方法：**配置维护、组服务、分布式消息队列**、**分布式通知/协调**等。

ZooKeeper在实现这些服务时，首先它设计一种新的**数据结构——Znode**，然后在该数据结构的基础上定义了一些**原语**，也就是一些关于该数据结构的一些操作。有了这些数据结构和原语还不够，因为我们的ZooKeeper是工作在一个分布式的环境下，我们的服务是通过消息以网络的形式发送给我们的分布式应用程序，所以还需要一个**通知机制**——Watcher机制。那么总结一下，ZooKeeper所提供的服务主要是通过：数据结构+原语+watcher机制，三个部分来实现的。

# 第五部分：JavaWeb

## Java有哪些web容器？

Web容器主要是来处理请求的，常用的tomat就是web容器，还有jboss、apache

## Servlet的本质？

Servlet其实就是一个实现了java特殊接口的类。

## Servlet的生命周期

1.init()

init 方法被设计成只调用一次。它在第一次创建 Servlet 时被调用，在后续每次用户请求时不再调用。因此，它是用于一次性初始化，就像 Applet 的 init 方法一样。

Servlet 创建于用户第一次调用对应于该 Servlet 的 URL 时，但是您也可以指定 Servlet 在服务器第一次启动时被加载。

当用户调用一个 Servlet 时，就会创建一个 Servlet 实例，每一个用户请求都会产生一个新的线程，适当的时候移交给 doGet 或 doPost 方法。init() 方法简单地创建或加载一些数据，这些数据将被用于 Servlet 的整个生命周期。

2.service方法

service() 方法是执行实际任务的主要方法。Servlet 容器（即 Web 服务器）调用 service() 方法来处理来自客户端（浏览器）的请求，并把格式化的响应写回给客户端。

每次服务器接收到一个 Servlet 请求时，服务器会产生一个新的线程并调用服务。service() 方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet、doPost、doPut，doDelete 等方法。

3.doGet()

GET 请求来自于一个 URL 的正常请求，或者来自于一个未指定 METHOD 的 HTML 表单，它由 doGet() 方法处理。

4.doPost()

POST 请求来自于一个特别指定了 METHOD 为 POST 的 HTML 表单，它由 doPost() 方法处理。

5.destory()

destroy() 方法只会被调用一次，在 Servlet 生命周期结束时被调用。destroy() 方法可以让您的 Servlet 关闭数据库连接、停止后台线程、把 Cookie 列表或点击计数器写入到磁盘，并执行其他类似的清理活动。在调用 destroy() 方法之后，servlet 对象被标记为垃圾回收

## Servlet是否线程安全？如果不安全，怎么变成线程安全的。

Servlet默认是单例模式的，所以本身是线程不安全的。当我们在类中定义成员变量，在高并发情况下，可能就会产生线程安全问题。

针对这种情况，有以下几种方式将servlet变成线程安全。

1.servlet实现SingleThreadModel接口。

实现该接口后,Servlet引擎会为每一个请求都创建一个servlet实例。这种方式开销太大，以不推荐使用。

2.代码级别的控制。

比如:在代码中加synchronized关键字；使用Lock锁等。

3.避免使用成员变量

Servlet之所以线程不安全，就是因为成员变量造成的。所以最好将成员变量改成局部的，将变量放在service方法中。由于方法中的局部变量是在栈中，彼此各自都拥有独立的运行空间而不会互相干扰，因此才做到线程安全。

## Cookies和Session的区别

Cookie保存在客户端浏览器，session保存在服务端。

Cookie不是很安全，别人可以分析本地cookie，进行cookie欺骗。Session相对安全很多。

重要的信息存sssion，其他的存cookie。

## Ajax的好处？

节约网络带宽，只更新需要更新的内容，不用重新请求整个页面

给客户提供更连贯的客户体验

可以开发类似桌面程序运行效果的B/S程序

## 异步请求的跨域问题

使用JSONP来解决Ajax的跨域问题：Ajax可以跨域请求服务，也可以相应到浏览器，但是json数据被浏览器限制，取不出来。

JS不能跨域请求数据，但是可以跨域加载JS文件。Jquery支持jsonp，已经帮我们封装好了。

原理：

1.首先定义一个Function mycall(data) 。使用Ajax跨域请求，带上参数callback=mycall.

2.在服务端接受请求，判断参数中是否含有callback参数，如果有，说明是跨域请求，就返回一个JS文件（拼接JS语句），mycall({json数据})。

3.页面接受响应之后，执行mycall方法，可以获得响应的JS数据了。

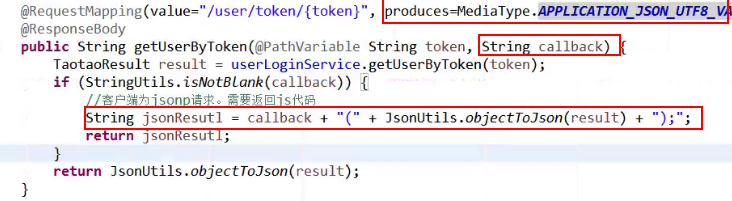
使用：

使用Jquery发送Ajax请求的时候，设置dataType=jsonP。



服务端：

接受参数callback，对callback判断，如果不是空，位jsonp请求，返回js代码。



## Forward和redirect的区别

redirect：请求重定向：客户端行为，本质上为2次请求，地址栏改变，前一次请求对象消失。

forward：请求转发:服务器行为，地址栏不变。

## 说说http、https

**HTTP：**

是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

**HTTPS：**

是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

**区别：**

1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## http状态码

200 请求成功

302 重定向失败

403 服务器拒绝

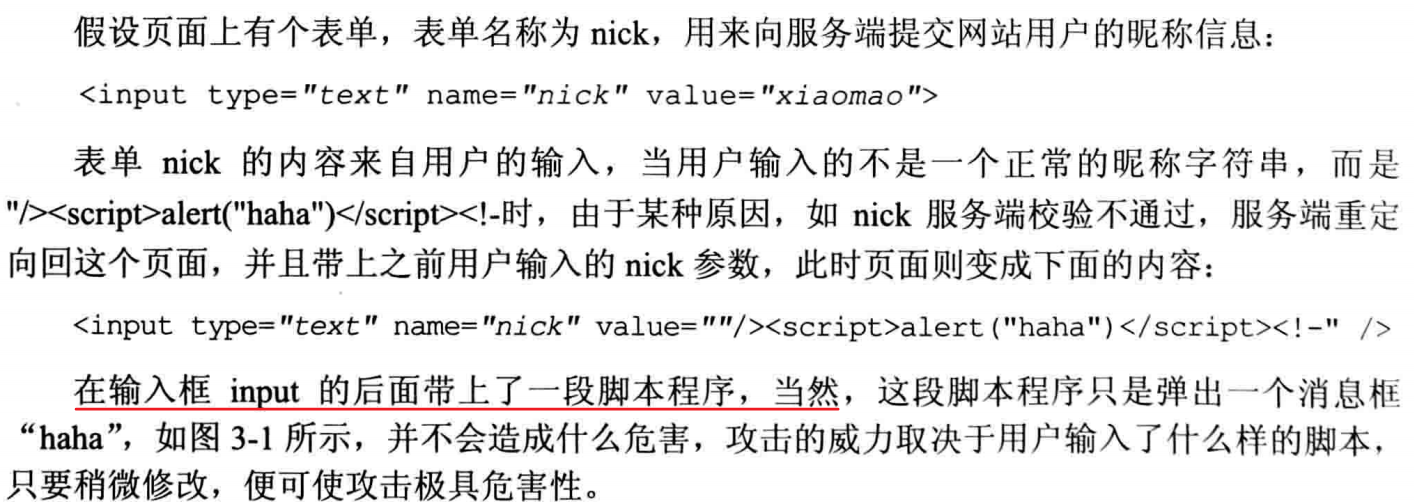
404 找不到

500 服务端错误

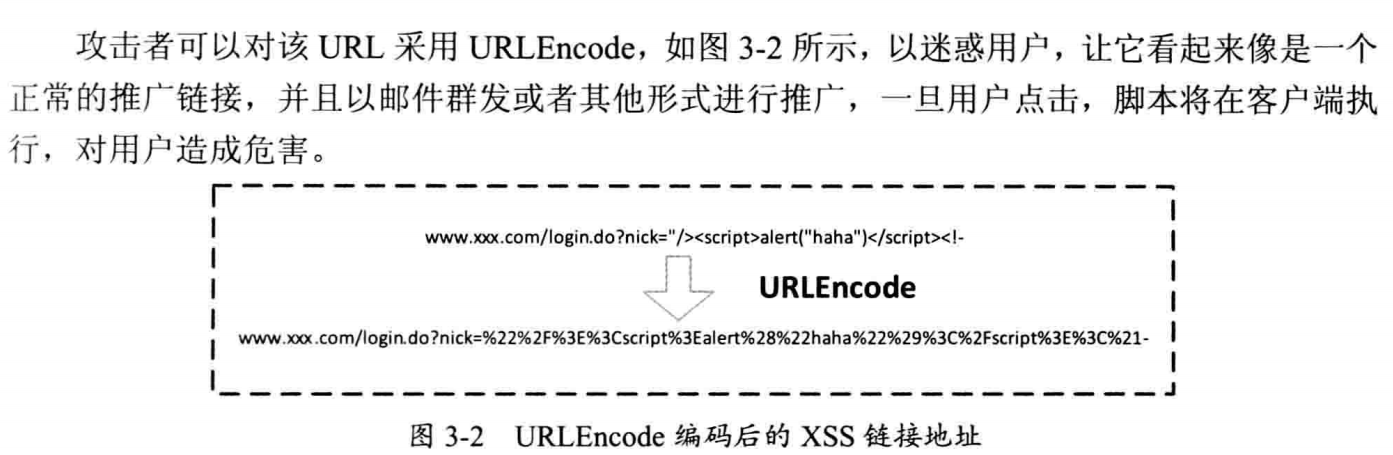
## 常见的Web项目攻击手段

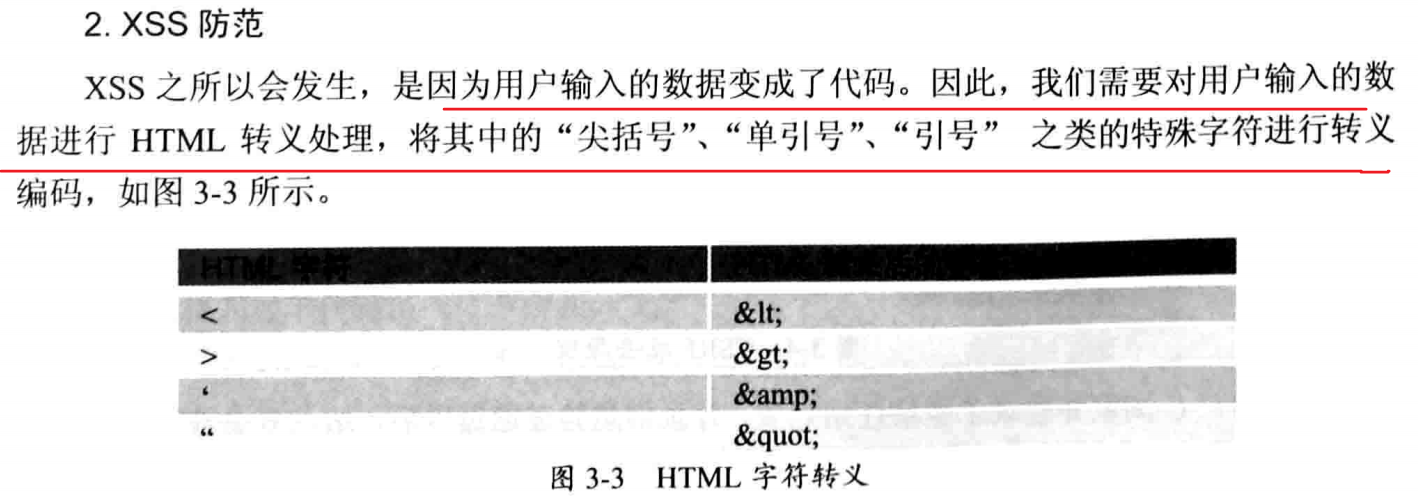
### 1.XSS攻击

XSS全称跨站脚本攻击

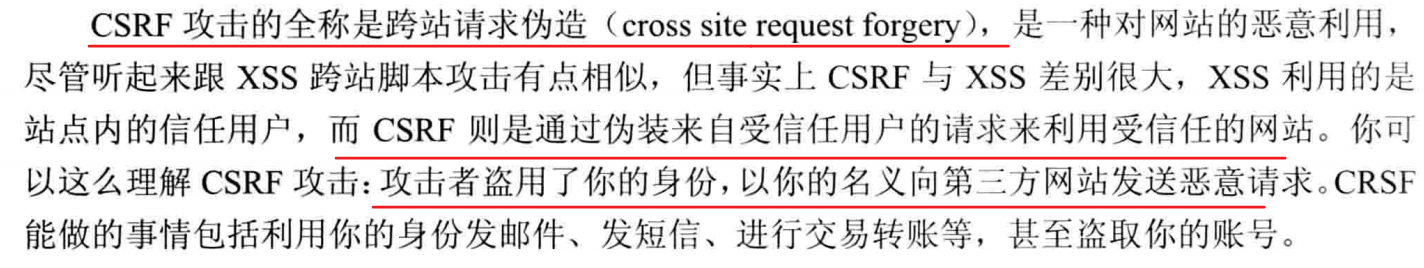


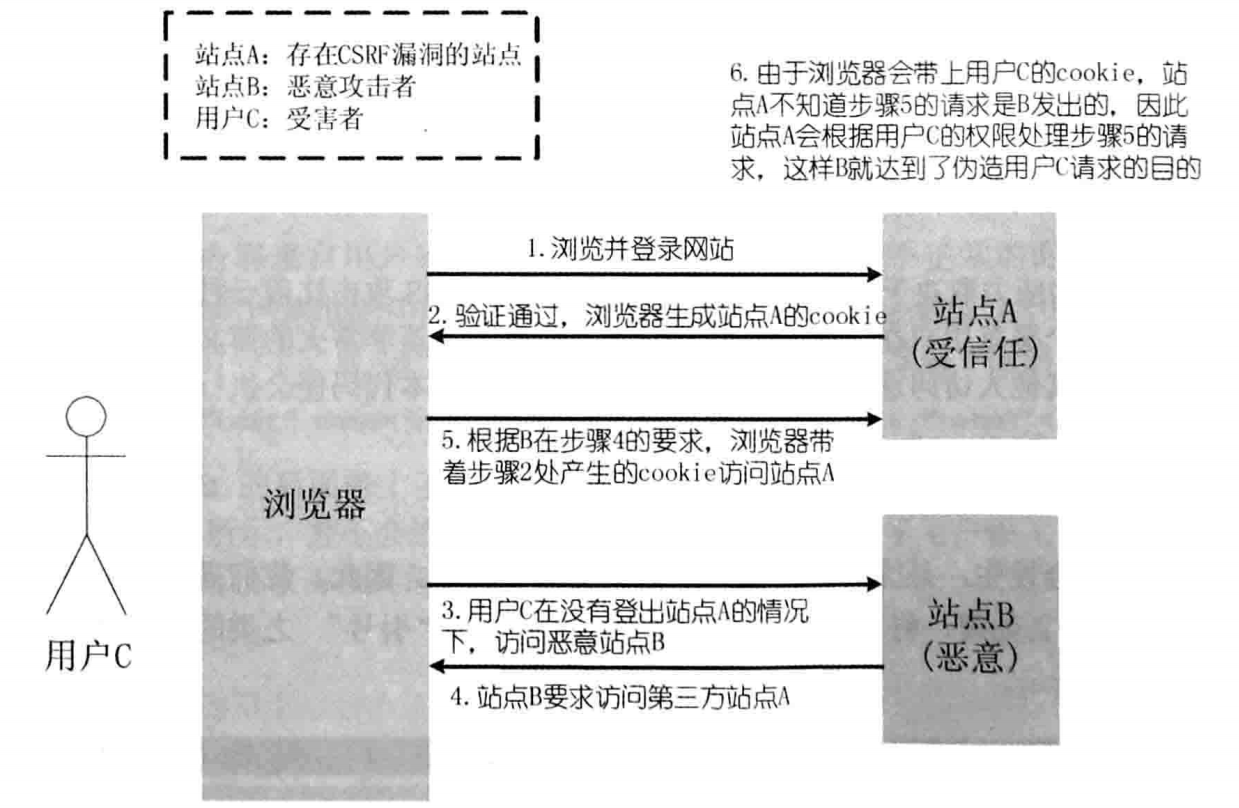


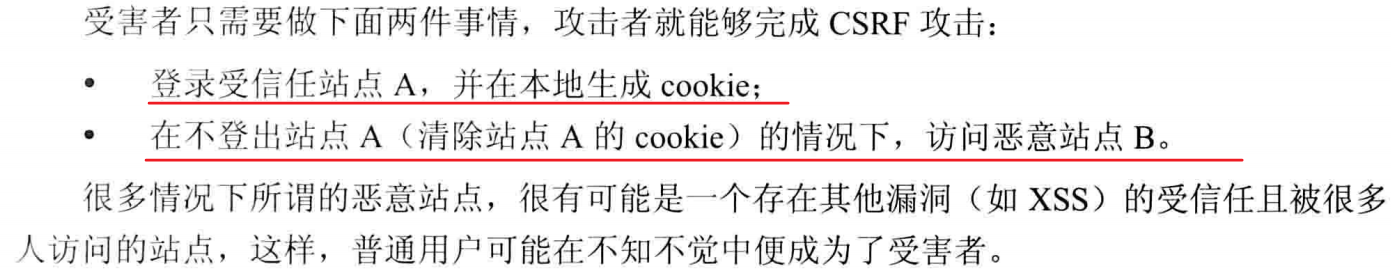


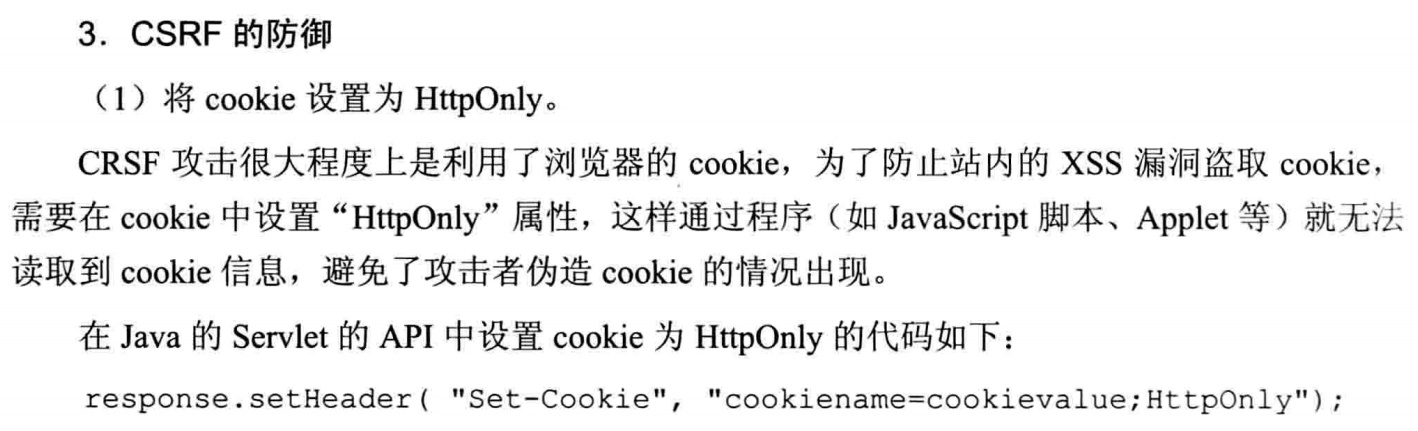


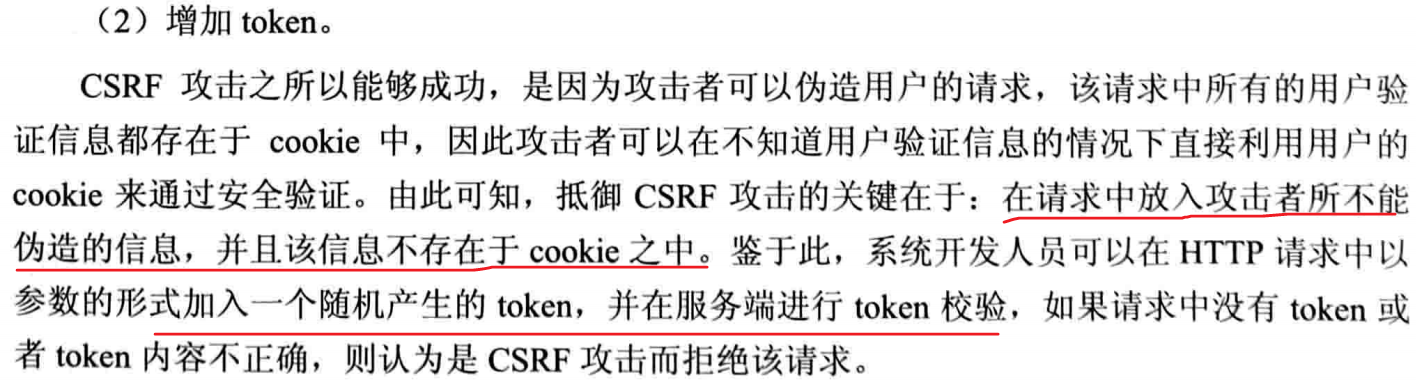
### 2.CSRF攻击





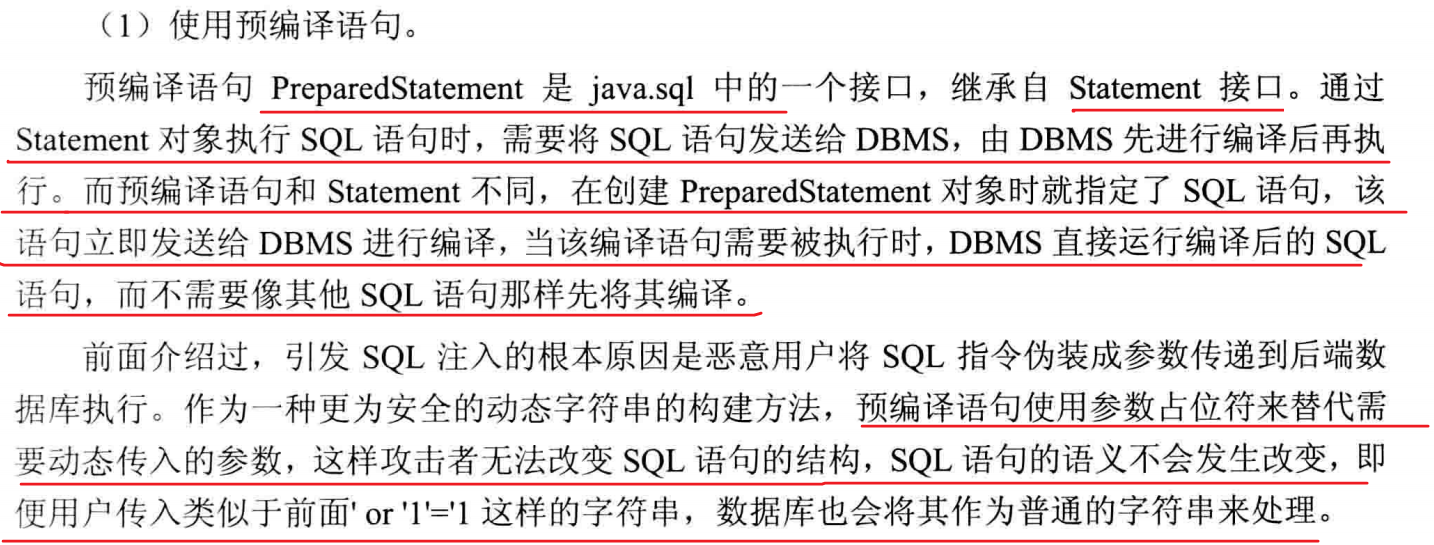


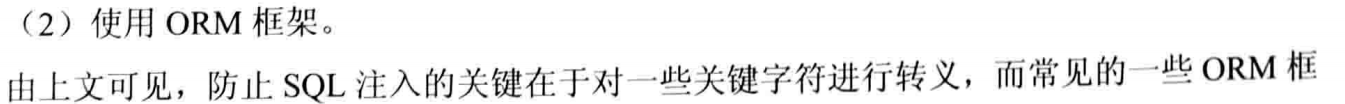


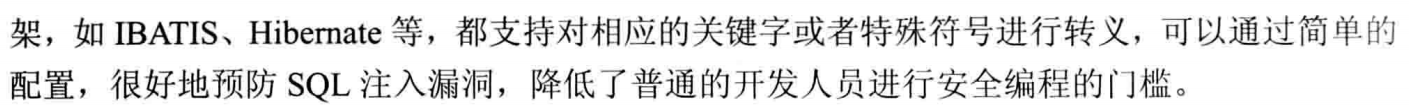


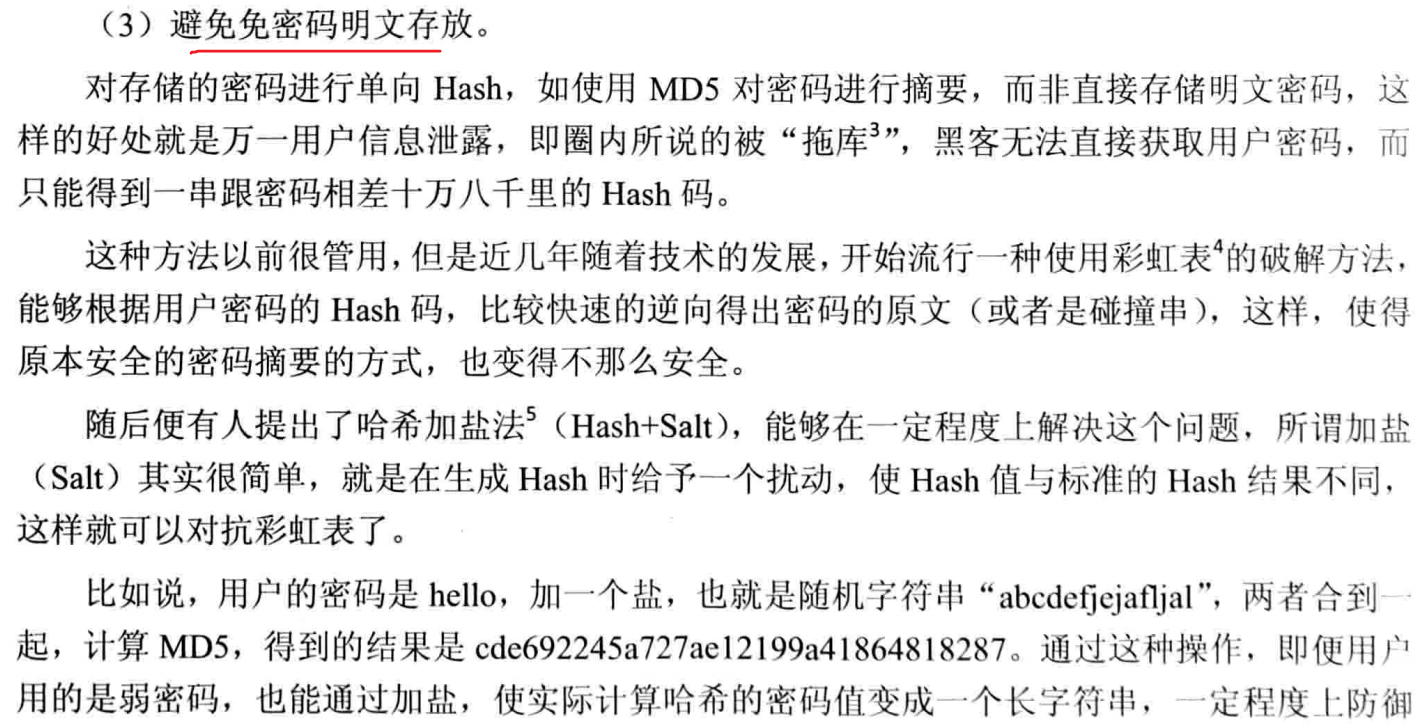


### 3.SQL注入攻击

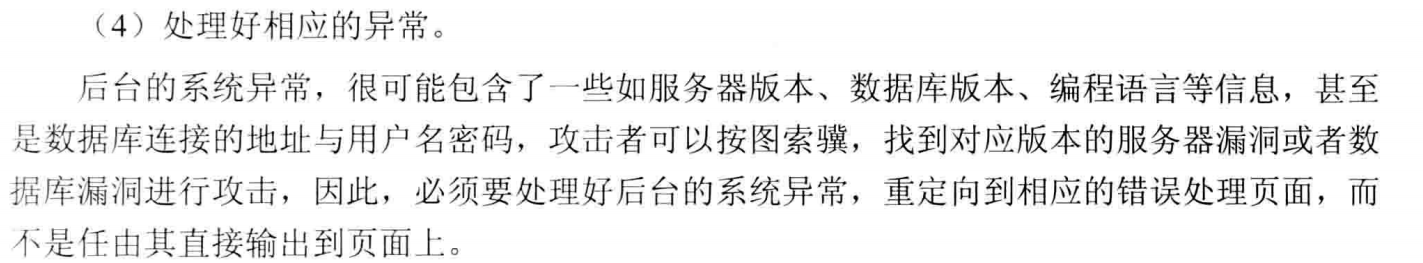




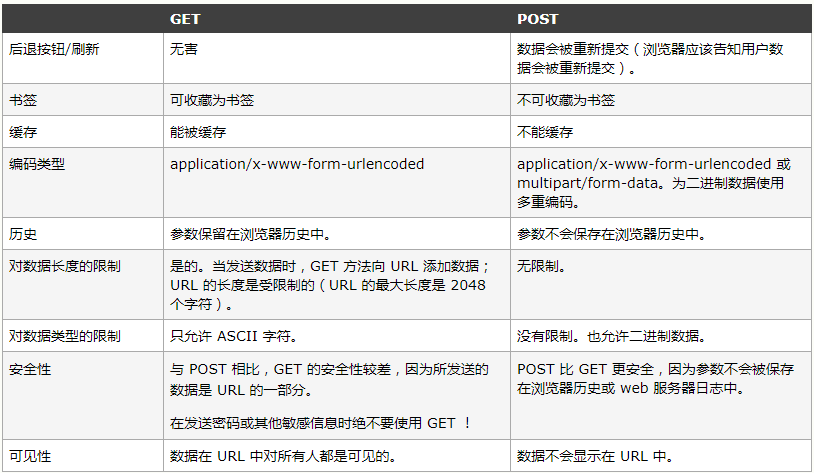








## get 和 post请求的区别



# 第六部分：项目相关

## 电商项目的知识点，以及用途

后台管理系统：管理商品、订单、类目、商品规格属性、用户管理以及内容发布等功能。

前台系统：用户可以在前台系统中进行注册、登录、浏览商品、首页、下单等操作。

会员系统：用户可以在该系统中查询已下的订单、收藏的商品、我的优惠券、团购等信息。

订单系统：提供下单、查询订单、修改订单状态、定时处理订单。

搜索系统：提供商品的搜索功能。

单点登录系统：为多个系统之间提供用户登录凭证以及查询登录用户的信息。

1.dubbo：解决了系统之间的通信问题。

## 介绍一下你最近做过的一个项目？

答：我最近做的一个项目，是受一家生物试剂销售公司委托，帮他们开发一个在线销售生物试剂，生物耗材相关产品的网站，叫做健隆生物网。项目采用的是主流ssm架构。

系统主要包括前台系统、后台系统、单点登录系统、搜索系统、订单系统等。用户可以在商城浏览商品，参加各种活动，下单等。管理员可以在后台上下架商品，管理订单，管理网站的内容，我主要负责的是搜索系统和单点登录系统。搜索系统主要是解决了网站的全文检索问题，单点登录系统让用户只需要登录一个系统就可以访问网站的其他系统。

答: 目前我们的盒马店内餐厅都是线下点餐、屏幕和短信叫号；就餐高峰时段，顾客排队结账和等餐时间超过 2 小时， 并且等餐过程信息不共享导致顾客体验很差；  
为了降低等餐时长，并且给针对等餐时间有提前告知和加工过程即时信息查看， 从而ᨀ升顾客体验，从而诞生了餐饮加工系统。这套系统可以让：

加工人员： 使用系统查看待加工工单、 超时ᨀ醒、 工作量可视化；  
顾客： 点餐期间查看排队预期，等餐过程中查看等餐倒计时；  
餐饮运营：通过系统各工序的数据分析（时长、 工作量），优化产能瓶颈、实现员工计件考核；

## 什么是分布式架构？分布式架构的原理以及好处？

简单点说，分布式系统就是将一个大的系统拆分为一个个的小的子系统（组件），这些子系统之间通过消息进行通信和协作，每个子系统都单独完成某一个功能，对外提供服务。相对于传统的架构来说，分布式系统更加的稳定，更容易维护，系统的耦合度也低一些。

比如传统的系统，如果系统中的某一个功能挂掉了，那么整个系统都不得不停止工作。而分布式系统，即使系统某一个组件挂掉了，其他的组件依然可以正常的工作。而且分布式系统也便于协同开发。

## 怎么让分布式系统相互信任？

根据域名，域名相同说明是可以信任的系统。

## 关于分布式事务的？

### 什么是分布式事务？

分布式事务，简单点说，就是一个业务需要调用不同系统的服务，这些服务要么一起成功，要么一起失败。

### 分布式事务的解决方案？

1. 基于XA协议的两阶段提交。

XA是一个分布式事务协议。

第一阶段：事务协调器要求每个涉及到事务的数据库预提交(precommit)此操作，并反映是否可以提交.  
第二阶段：事务协调器要求每个数据库提交数据。  
其中，如果有任何一个数据库否决此次提交，那么所有数据库都会被要求回滚它们在此事务中的那部分信息。

XA比较简单，成本比较低。但是性能较低，并发量高的话无法满足需求，而且MYSQL对XA的支持也不是很好。

1. 消息事务+最终一致性 （补偿的方式）

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说  
他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。  
消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，  
如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方  
进行回滚等操作。  
生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对  
账补账逻辑，这种方案还是非常实用的。

TCC编程模式

这种模式是两阶段提交的一个变种，通过代码认为实现两阶段提交，不同的业务场景，代码不一样，复杂度也不一样，复用性差，不建议使用。

## 什么是soa架构？soa架构有什么好处？

Soa是面向服务的架构，就是将工程分成服务层、表现层。服务层里面是一些具体的业务逻辑，表现层调用服务层获取数据展示给用户。

Soa架构将业务的调用，和业务具体的实现分开了。降低了系统之间的耦合度，更容易维护。而且由于耦合性降低，系统的扩展性也更加好。

## 项目中遇到的难点，怎么解决的？

在机器人项目，编写餐桌模块的时候，由于每个餐桌需要生成一个二维码，并且在二维码的上下有中文字体。在开发完成之后，本地测试通过，但是到Linux服务器上面后，发现中文字体出现乱码。

Debug排查原因之后，发现是由于Linux系统缺少一个中文字体造成的。由于没有权限上传字体文件到服务器。所以当时在这里卡了大半天的时间，摸索了各种办法都不行。

最后冷静下来，首先心中坚信这个问题肯定有其他方式来解决。然后开始看java的Font类源码。Font类有一个很关键的构造方法，这个构造方法可以传入一个输入流。看到这个我就知道怎么解决了。

最后，我将字体文件放在了工程目录内，以流的形式读取出来，然后生成Font对象。

# 分布式session解决方案

1.使用Nginx的粘性session。

用户第一次请求时，负载均衡器将用户的请求转发到了A服务器上，如果负载均衡器设置了粘性Session的话，那么用户以后的每次请求都会转发到A服务器上，相当于把用户和A服务器粘到了一块，这就是粘性Session机制。

优点：简单，不需要对session做任何处理。

缺点：缺乏容错性，如果当前访问的服务器发生故障，用户被转移到第二个服务器上时，他的session信息都将失效。

适用场景：发生故障对客户产生的影响较小；服务器发生故障是低概率事件。

2.服务器之间的session复制，广播机制。

原理：任何一个服务器上的session发生改变（增删改），该节点会把这个 session的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要session，以此来保证Session同步。

优点：可容错，各个服务器间session能够实时响应。

缺点：会对网络负荷造成一定压力，如果session量大的话可能会造成网络堵塞，拖慢服务器性能。

1. 使用redis集群解决session共享问题。

使用redis模拟session，key是sessionId，value是session的信息。客户每次发送请求都会从cookie中将sessionid带到服务器，根据sessionId从redis中回去session信息

4.session持久化到数据库

原理：就不用多说了吧，拿出一个数据库，专门用来存储session信息。保证session的持久化。

优点：服务器出现问题，session不会丢失

缺点：如果网站的访问量很大，把session存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。

第七部分 项目管理工具

## Svn怎么解决代码冲突？

## Maven热运行时什么？冷运行时什么？

## Maven的声明周期

maven包括三套项目生命周期：

Clean Lifecycle 在进行真正的构建之前进行一些清理工作。

Default Lifecycle 构建的核心部分，编译，测试，打包，部署等等。

Site Lifecycle 生成项目报告，站点，发布站点。

注意：

执行一个生命周期阶段，其前面的所有阶段都会完成。

## Maven的Profile是什么？

Profile可以让我们定义一系列的配置信息，然后设定其激活条件。这样我们可以定义多个profile，然后每个profile对应不同的激活条件和配置信息，从而达到不同环境使用不同配置信息的效果。比如说，我们可以通过profile定义在jdk1.5以上使用一套配置信息，在1.5以下使用另外一套配置信息；

# 第八部分：算法题

## 选择排序

|  |
| --- |
| */\*\*  \* Created by wb-ny291824 on 2017/8/19.  \* 选择排序  \*  这一种简单的排序方法，它的基本思想是：  R[n]  第一次从R[0]~R[n-1]中选取最小值，与R[0]交换;  第二次从R[1]~R[n-1]中选取最小值，与R[1]交换;  ....，  第i次从R[i-1]~R[n-1]中选取最小值，与R[i-1]交换;  .....，  第n-1次从R[n-2]~R[n-1]中选取最小值，与R[n-2]交换;  总共通过n-1次,得到一个按排序码从小到大排列的有序序列.  选择排序的平均时间复杂度是O(n²)的。  \*/*  **public class** ChooseSort {   **public static void** main(String[] args) {  **int**[] arr = {1,5,6,21,24,121,11,14,23};  *chooseSort*(arr);  **for** ( **int** a :arr) {  System.***out***.println(a);  }   }   **public static void** chooseSort(**int**[] data) {  **for** (**int** i = 0; i < data.**length**; i++) {  **int** min = i;  **for** (**int** j = i+1; j <data.**length** ; j++) {  **if** (data[min] > data[j]) {  min = j;  }  }   **if** (min != i) {  **int** temp = data[min];  data[min] = data[i];  data[i] = temp;  }  }  }  } |

## 冒泡排序

|  |
| --- |
| **package** com.keji.blog.controller.airthmetic.sort;  */\*\*  \* Created by wb-ny291824 on 2017/8/19.  \* 冒泡排序  \* 依次比较相邻的两个数，将小数放在前面，大数放在后面。  即在第一趟：  　　首先比较第1个和第2个数，将小数放前，大数放后;  　　然后比较第2个数和第3个数，将小数放前，大数放后，如此继续，直至比较最后两个数，将小数放前，大数放后;  　　至此第一趟结束，将最大的数放到了最后。  在第二趟：  　　仍从第一对数开始比较（因为可能由于第2个数和第3个数的交换，使得第1个数不再小于第2个数），将小数放前，大数放后;  　　一直比较到倒数第二个数（倒数第一的位置上已经是最大的）;  　　第二趟结束，在倒数第二的位置上得到一个新的最大数（其实在整个数列中是第二大的数）。  如此下去，重复以上过程，直至最终完成排序。  \*/* **public class** BubbleSort {   **public static void** main(String[] args) {  **int**[] arr = {1,5,6,21,24,121,11,14,23};  BubbleSort bubbleSort = **new** BubbleSort();  bubbleSort.bubbleSort(arr);  **for** (**int** i = 0; i < arr.**length**; i++) {  System.***out***.println(arr[i]);  }  }   **public void** bubbleSort(**int**[] data) {  **int** temp;  **for** (**int** i = 0; i < data.**length**; i++) {  **for** (**int** j = 0; j < data.**length** - i - 1; j++) {  **if** (data[j] > data[j + 1]) {  temp = data[j];  data[j] = data[j + 1];  data[j + 1] = temp;  }  }  }  } } |

## 按绝对值排序，比如-10 -5 1 2 30 🡪 1,2，-5，-10，30

|  |
| --- |
| **package** com.keji.blog.controller.airthmetic.sort;  */\*\*  \* Created by wb-ny291824 on 2017/8/20.  \* 按绝对值排序。  \* 在冒泡排序的基础上，使用Math.abs()方法来获得绝对值进行比较。  \*/* **public class** ABSSort {  **public static void** main(String[] args) {  **int**[] arr = {-10, -1, -5, 6, 2, 20};   **for** (**int** i = 0; i < arr.**length**; i++) {  **for** (**int** j = 0; j < arr.**length** - i - 1; j++) {  **if** (Math.*abs*(arr[j])>Math.*abs*(arr[j+1])) {  **int** temp = arr[j];  arr[j] = arr[j + 1];  arr[j+1] = temp;  }  }  }   **for** (**int** i = 0; i < arr.**length**; i++) {  System.***out***.println(arr[i]);  }  } } |

# 第九部分：HR面试

为什么从上家公司辞职？

# 第十部分：Linux

## Linux常用命令

cd

ls ll

mkdir rmdir

cat more less tail

rm cp mv tar

grep 查找复合条件的字符串

pwd

touch 创建一个空文件

ps –ef 查看所有进程 ps –ef | grep ssh 查找ssh进程

# 第十一部分：JVM

## 1.JVM的内存是线程共享的吗？

大多数 JVM 将内存区域划分为 **Method Area（Non-Heap）（方法区）** ,**Heap（堆）** , **Program Counter Register（程序计数器）** ,   **VM Stack（虚拟机栈，也有翻译成JAVA 方法栈的）,Native Method Stack**  （ **本地方法栈** ），其中**Method Area** 和 **Heap**是线程共享的 **，VM Stack，Native Method Stack  和Program Counter Register**是非线程共享的。为什么分为 线程共享和非线程共享的呢?请继续往下看。

首先我们熟悉一下一个一般性的 Java 程序的工作过程。一个 Java 源程序文件，会被编译为字节码文件（以 class 为扩展名），每个java程序都需要运行在自己的JVM上，然后告知 JVM 程序的运行入口，再被 JVM 通过字节码解释器加载运行。那么程序开始运行后，都是如何涉及到各内存区域的呢？

概括地说来，JVM初始运行的时候都会分配好 **Method Area（方法区）** 和**Heap（堆）** ，而JVM 每遇到一个线程，就为其分配一个 **Program Counter Register（程序计数器）** ,   **VM Stack（虚拟机栈）和Native Method Stack  （本地方法栈），**当线程终止时，三者（虚拟机栈，本地方法栈和程序计数器）所占用的内存空间也会被释放掉。这也是为什么我把内存区域分为线程共享和非线程共享的原因，非线程共享的那三个区域的生命周期与所属线程相同，而线程共享的区域与JAVA程序运行的生命周期相同，所以这也是系统垃圾回收的场所只发生在线程共享的区域（实际上对大部分虚拟机来说知发生在Heap上）的原因。

## 2.JVM内存参数配置

-Xmx：最大堆大小

-Xms：初始堆大小

-Xmn:年轻代大小

-XXSurvivorRatio：年轻代中Eden区与Survivor区的大小比值

年轻代5120m， Eden：Survivor=3，Survivor区大小=1024m（Survivor区有两个，即将年轻代分为5份，每个Survivor区占一份），总大小为2048m。

-Xms初始堆大小即最小内存值为10240m

## 3.Java文件被虚拟机加载运行的过程

类被加载的方式有两种，一种是显示加载，比如Class.forName()的时候。一种是隐式加载，比如使用new关键字创建对象的时候。类的加载需要用到类加载器，Java有3种类加载器，分别负责加载系统类、扩展类、自定义类。这3种加载器通过委托的方式实现协调工作。具体来说，当类需要被加载时，类加载器会请求父类来完成这个载入工作。父类完成不了才会交给子类完成。

类的加载主要分为三步：

1.装载

根据查找路径找到相对应的class文件，然后导入。

2.链接

a.检查 检查待加载的class文件的正确性

b.准备 给类中的静态变量分配存储空间

c.解析 将符号引用转换成直接引用

3.初始化

对静态变量和静态代码块执行初始化工作。

## 4.JVM内存模型

1.程序计数器。

程序计数器可以看做是当前线程所执行的字节码的行数指示器。在虚拟机概念模型中，字节码解释器就是通过改变这个计数器的值来选取下一条执行的字节码指令。

程序计数器是线程私有的，每一个线程都会有一个独立的程序计数器。

如果线程正执行Java程序，程序计数器存储的是正在执行的字节码的地址。

如果线程正执行Native方法，程序计数器的值为null。

2.Java虚拟机栈。

Java虚拟机栈也是线程私有的，它的生命周期和线程相同。每个方法在执行的同时都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。每一个方法从调用到结束的过程，就对应着帧栈在虚拟机中的入栈和出栈的过程。

局部变量表存放了编译器可知的各种基本数据类型、对象引用类型和returnAddress类型。

3.本地方法栈。

本地方法栈与虚拟机栈所发挥的作用非常相似，他们的区别不过是虚拟机栈为虚拟机所执行Java方法服务，而本地方法栈则为虚拟机使用到的native方法服务。有的虚拟机会将本地方法栈和Java虚拟机栈合并。

4.堆内存

一般堆内存是Java虚拟机所管理内存中的最大一块。堆内存是线程共享的。此区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。

堆内存是java gc的主要区域。

5.方法区

线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量、即使编译器编译后的代码等、数据

6.常量池

常量池是方法区的一部分。Class文件除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，用于存放编译期生成的各种字面量和符号引用。这部分内存内容将在类加载后进入方法区的运行时常量池的存放。

## JDK1.8的新特性？

1. 接口中有默认方法和静态方法，也就是接口中也可以有实现类、
2. Lambda表达式
3. 1.8使用元空间代替了持久代。
4. stream

## Java的GC机制

#### 可达性分析判断垃圾对象

谈到垃圾回收，首先要知道怎么确定对象是可以回收的。最常见的就是引用计数法，就是判断这个对象是否有引用与之关联，没有说明可以回收。但是这种办法不能解决循环引用的问题。Java采用的是**可达性分析法**。这个方法的基本思想就是通过一系列“GC Roots”对象作为起点进行搜索，如果“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，但是不可达对象成为可回收对象必须经历两次标记的过程，如果这两次标记还没有逃脱成为可回收对象的可能，那么基本就可以判断是“垃圾对象”了。

#### 分代收集算法

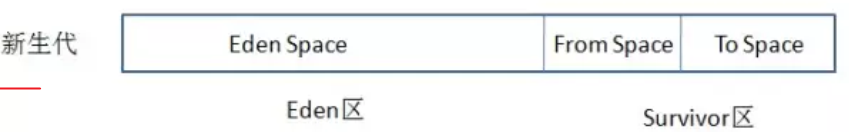
分代收集发的核心思想是根据对象的声明周期将内存划分为若干个不同的区域。一般情况下将堆内存划分为新生代和老年代。

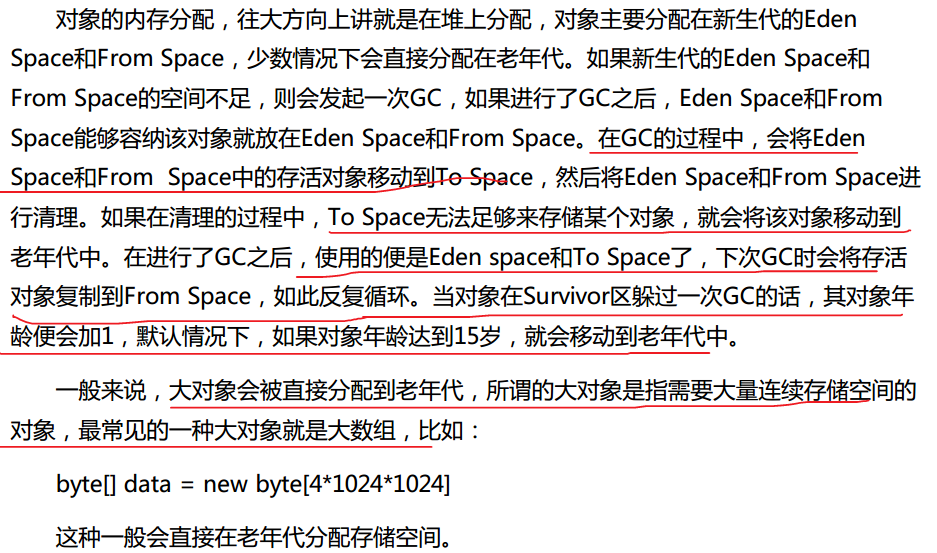
新生代的特点是每次垃圾回收都有大量垃圾被回收，对于新生代，又划分为一个Eden空间和两个Survivor空间，每次使用Eden空间和其中一块Survivor空间。垃圾回收的时候将Eden和Survivor中还存活的对象复制到另一块Survivor空间，然后清理到刚才使用的Eden和Survivor空间。

老年代的特点是每次回收都只回收少量对象，一般使用标记整理法。一般大对象会直接分配到老年代。

在堆区之外还有一个永久代，用来存储class类、常量、方法描述等。对永久代的回收主要回收两部分：废弃常量和无用的类。

#### G1垃圾回收器





## 操作系统中heap和stack的区别

### 一、堆栈空间分配区别：

　　1、栈（操作系统）：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈；   
　　2、堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

### 二、堆栈缓存方式区别：

　　1、栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放；   
　　2、堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

### 三、堆栈数据结构区别：

　　堆（数据结构）：堆可以被看成是一棵树，如：堆排序；   
　　栈（数据结构）：一种先进后出的数据结构

### Java中栈和堆的区别：

　　栈(stack)与堆(heap)都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。   
　　在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。当在一段代码块定义一个变量时，Java就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。   
　　堆内存用来存放由new创建的对象和数组，在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。   
**Java中变量在内存中的分配**：   
　　1、类变量（static修饰的变量）：在程序加载时系统就为它在堆中开辟了内存，堆中的内存地址存放于栈以便于高速访问。静态变量的生命周期–一直持续到整个”系统”关闭。   
　　2、实例变量：当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量（比如说类实例），然后根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的”物理位置”。 实例变量的生命周期–当实例变量的引用丢失后，将被GC（垃圾回收器）列入可回收“名单”中，但并不是马上就释放堆中内存。   
　　3、局部变量：局部变量，由声明在某方法，或某代码段里（比如for循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放。   
这里要涉及到Java内存问题，可以参考：[Java的内存机制](http://blog.csdn.net/u014306011/article/details/51068495)