[Java Hashmap工作原理及实现](https://yikun.github.io/2015/04/01/Java-HashMap%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86%E5%8F%8A%E5%AE%9E%E7%8E%B0/)

[ConcurrentHashMap和HashTable的区别](http://www.importnew.com/7166.html)

[HashMap 1.7和1.8的区别](https://juejin.im/post/5aa5d8d26fb9a028d2079264#heading-28)

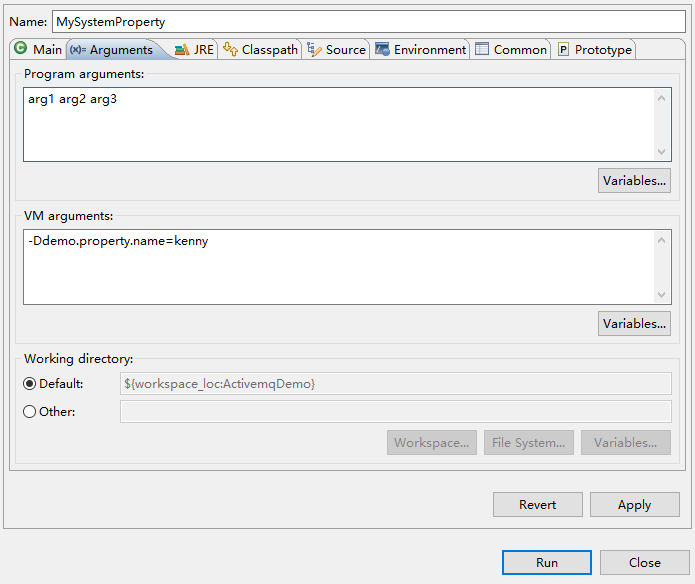
# Java底层原理

## -D<propertyName>=value

在虚拟机的系统属性中设置属性键/值对，运行在此虚拟机之上的应用程序可用System.getProperty(“propertyName”)得到value的值。

如果value中有空格，则需要用双引号将该值括起来，如-Dname=”space string”。

该参数通常用于设置系统级全局变量值，如配置文件路径，因为该属性在程序中任何地方都可访问。



## Java内存分配

|  |  |
| --- | --- |
| 栈 | 存放基本类型的数据和对象的引用。 |
| 堆 | 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。 |
| 静态域 |  |
| 常量池 | 在编译期间被确定，并被保存在已编译的.class文件中的一些数据。 |

## 区分引用变量、数组和对象

引用变量是普通变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后释放。

数组和对象本身在堆中分配，即使程序运行到使用new产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才会变为垃圾，不能再被使用，但仍然占据内存空间不释放，在随后的一个不确定的时间被垃圾回收器收走（释放掉）。这也是Java比较占内存的原因。

栈中的变量指向堆内存中的变量，这就是Java中的指针。

## 成员变量和局部变量

成员变量：方法外部，类的内部定义的变量；

局部变量：方法或语句块内部定义的变量。局部变量必须初始化。形式参数是局部变量。

成员变量：存储在堆中的对象里面，由垃圾回收器负责回收。

局部变量：数据存在于栈内存中，栈内存中的局部变量随着方法的消失而消失。

## 两个包装类的引用是否指向同一个对象

|  |
| --- |
| String s1 = "abc";  String s2 = "abc"; // 对象可能并没有创建!而可能是指向一个先前已经创建的对象  System.***out***.println(s1 == s2); // true  String s3 = **new** String("abc");  String s4 = **new** String("abc");  System.***out***.println(s3 == s4); // false  System.***out***.println(s2 == s3); // false |

用new的方式是生成不同的对象，每一次生成一个。

## String采用连接运算符（+）效率低下的原因？

每做一次+就产生一个StringBuilder对象，然后append后就扔掉。下次循环再重新生成一个StringBuilder对象，然后append字符串，如此循环直至结束。如果我们直接采用StringBuilder对象进行append的话，我们可以节省N-1次创建和销毁对象的时间。所以对于在循环中要进行字符串连接的应用，一般都是用StringBuffer或StringBuilder对象来进行append操作。

## String中的final用法和理解

|  |
| --- |
| **final** StringBuilder a = **new** StringBuilder("aaa");  **final** StringBuilder b = **new** StringBuilder("bbb");  // a = b; // 此句编译不通过  a.append("bbb");  **final** String c = "1234";  // c = "5678"; // 此句编译不通过 |

## Java中的String，StringBuilder，StringBuffer三者的区别

|  |  |  |  |
| --- | --- | --- | --- |
|  | String | StringBuilder | StringBuffer |
| 常量-变量 | 字符串常量 | 字符串变量 | |
| 更改 | 一旦创建后不能更改。 | 对象是变量，是可以更改的。 | |
| 运行速度 执行速度 | 慢 | 快 | 中 |
| 线程安全 |  | 线程不安全 | 线程安全 |
| 线程实现 |  | 在单线程的情况下，建议使用速度比较快的StringBuilder。 | 其中很多方法可以带有synchronized关键字，所以可以保证线程是安全的。 |
| 应用场景 | 适用于少量的字符串操作的情况。 | 适用于单线程下在字符串缓冲区进行大量操作的情况。 | 适用多线程下在字符缓冲区进行大量操作的情况。 |

## TreeMap与TreeSet实现原理是什么？

**相同点**：

1. 都是有序的集合，即他们存储的值都是排好序的。
2. 都是非同步集合，因此它们不能在多线程之间共享，不过可以使用方法Collection.synchronizedMap()来实现同步。
3. 运行速度都要比Hash集合慢，内部对元素的操作时间复杂度为O(logN)，而HashMap/HashSet则为O(1)。

**不同点**：

|  |  |
| --- | --- |
| TreeSet | TreeMap |
| 实现Set接口 | 实现Map接口 |
| 只存储一个对象 | 存储两个对象key和value(仅仅key对象有序) |
| 不能有重复对象 | 可以存在重复对象 |
|  | 底层采用红黑树的实现，完成数据的有序的插入、排序。 |
| 存放的对象所属的类必须实现Comparable接口,该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。 | 存放键值对映射的键必须实现Comparable接口，从而根据键对元素进行排序。 |

## Array和ArrayList的区别

ArrayList想象成一种“会自动扩增容量的Array”。

|  |  |
| --- | --- |
| Array([]) | ArrayList |
| 最高效；但是其容量固定且无法动态改变。 | 容量可动态增长；但牺牲效率； |
|  |  |

## Hashmap实现原理，传参为5的时候，初始容器是多大？

## Java内部数据结构选型指南

|  |  |  |
| --- | --- | --- |
| 数据结构 | 特点 | 复杂度 |
| ArrayList | 列表,顺序存储 |  |
| LinkedList | 双向列表,链式存储 |  |
| HashSet | 元素唯一 |  |
| TreeSet | 元素唯一,带排序 |  |
| Stack | 堆栈 |  |
| LinkedHashSet |  |  |
| Vector | 向量 |  |
| HashMap | Key-Value |  |
| TreeMap | 带排序 |  |
| LinkedHashMap |  |  |
| WeakHashMap |  |  |
| Hashtable |  |  |

# JVM

## JVM的数据区有哪些，作用是什么？

## JVM堆内存结构是怎样的？哪些情况会触发GC？会触发哪些GC？

## JVM调优命令，主要回答jmap、jstack、jps等命令？

## JVM内存模型、JVM堆怎么划分的一些简单的JVM问题？

# Java IO

## Select、Poll与ePoll的区别

|  |  |  |
| --- | --- | --- |
| Select | Poll | ePoll |
| 都是IO多路复用机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。  都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的。 | | |
| 时间复杂度O(n) | O(n) | O(1) |
| 无差别轮询复杂度 | | |
|  | 基于链表存储，没有最大连接数的限制。 | event poll，事件驱动 |

区别异步IO，异步IO无需自己负责进行读写，异步IO的实现会负责把数据从内核拷贝到用户空间。

## BIO和NIO有什么区别

# Java并发编程

## 为什么要使用线程？

|  |
| --- |
| 1 更多的处理器核心。  2 更快的响应时间。  3 更好的编程模型。 |

## 线程的状态

|  |  |
| --- | --- |
| NEW | 初始状态，线程被构建，但是还没有调用start()方法 |
| RUNNABLE | 运行状态 |
| BLOCKED | 阻塞状态，表示线程阻塞于锁 |
| WAITING | 等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断） |
| TIME\_WAITING | 超时等待状态，该状态不会释放锁 |
| TERMINATED | 终止状态 |

## ThreadLocal有什么缺陷？如果是线程池里的线程用ThreadLocal会有什么问题？

ThreadLocal可能会导致内存泄漏的问题。ThreadLocal就是用于线程（Thread）私有（Local）的存储结构。

应用：SpringMVC中有RequestContextHolder，可以使用RequestContextHolder得到各个请求的数据。

## 类的加载机制，为什么要用双亲委托？如何打破双亲委托加载机制？

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过子类来引用父类中的静态字段，只会触发父类的初始化而不会触发子类的初始化。

类实例化过程中，先执行静态代码块，再执行构造函数。

|  |
| --- |
| **public** **class** MyClassLoader {  **public** **static** **void** main(String[] args) {  System.***out***.println(Class3.*value*);  }  }  **class** Class1 {  **static** {  System.***out***.println("Class1 Static");  }  **public** Class1() {  System.***out***.println("Class1 Construct");  }  }  **class** Class2 **extends** Class1 {  **static** {  System.***out***.println("Class2 Static");  }  **public** Class2() {  System.***out***.println("Class2 Construct");  }  **static** **int** *value* = 10;  }  **class** Class3 **extends** Class2 {  **static** {  System.***out***.println("Class3 Static");  }  **public** Class3() {  System.***out***.println("Class3 Construct");  }  } |

程序执行结果：

|  |
| --- |
| Class1 Static  Class2 Static  10 |

“双亲委派”机制知识Java推荐的机制，并不是强制的机制。

我们可以继承java.lang.ClassLoader类，实现自己的类加载器。如果想保持双亲委派模型，就应该重新findClass(name)方法；如果想破坏双亲委派模型，可以重写loadClass(name)方法。

**双亲委派机制**的工作流程：

1 当前ClassLoader首先从自己已经加载的类中查询是否此类已经加载，如果已经加载则直接返回原来已经加载的类。每个类加载器都有自己的加载缓存，当一个类加载了以后就会放入缓存，等下次加载的时候就可以直接返回了。

2 当前ClassLoader的缓存中没有找到被加载的类的时候，委托父类加载器去加载，父类加载器采用同样的策略，首先查看自己的缓存，然后委托父类的父类去加载，一直到bootstrap ClassLoader。

3 当所有的父类加载器都没有加载的时候，再由当前的类加载器加载，并将其放入它自己的缓存中，以便下次有加载请求的时候直接返回。

## Synchronization和Lock有什么区别？

## Synchronized和ReentrantLock区别

Synchronized：

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化synchronized，另外可读性非常好。

ReentrantLock：

ReentrantLock提供了多样化的同步，比如有时间限制的同步，可以被Interrupt的同步（synchronized的同步是不能被Interrupt的）等。在资源竞争不激烈的情况下，性能稍微比synchronized差一点点。但是当同步非常激烈的时候，synchronized的性能一下子能下降好几十倍。而ReentrantLock却还能维持常态。

## 线程池各个参数的作用

|  |  |
| --- | --- |
| corePoolSize | 核心池的大小。 |
| maximumPoolSize | 线程池最大线程数。 |
| keepAliveTime | 线程没有执行任务是最多保持多久时间会终止。 |
| unit | 参数keepAliveTime的时间单位，有7种取值。 |
| workQueue | 一个阻塞队列，用来存储等待执行的任务。 |
| threadFactory | 线程工厂，主要用来创建线程。 |
| handler | 表示当拒绝处理任务时的策略，有四种取值。 |

prestartAllCoreThreads() 初始化所有核心线程

prestartCoreThread() 初始化一个核心线程

allowCoreThreadTimeOut(boolean)

TimeUnit.DAYS 天

TimeUnit.HOURS 小时

TimeUnit.MINUTES 分钟

TimeUnit.SECONDS 秒

TimeUnit.MILLISECONDS 毫秒

TimeUnit.MICROSECONDS 微妙

TimeUnit.NANOSECONDS 纳秒

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和SynchronousQueue。线程池的排队策略与BlockingQueue有关。

shutdown()：不会立即终止线程，而是等待所有缓存队列中的任务都执行完后才终止，但再也不会接受新的任务。

shutdownNow():立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未的任务。

动态调整线程池容量大小的方法：

setCorePoolSize():设置核心池大小。

setMaximumPoolSize():设置线程池最大能创建的线程数目大小。

上面两个函数的参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

在Java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池。

|  |  |
| --- | --- |
| Executors.newCachedThreadPool() | 创建一个缓存，缓存容量大小为Integer.MAX\_VALUE |
| Executors.newSingleThreadExecutor() | 创建容量为1的缓存池 |
| Executors.newFixedThreadPool() | 创建固定容量大小的缓存池 |
| Executors.newWorkStealingPool() |  |
| Executors.newSingleThreadScheduledExecutor() |  |
| Executors.newScheduledThreadPool() |  |

## CAS与synchronized的使用场景

CAS适用于写比较少的情况下（多读场景，冲突一般较少）；

Synchronized适用于写比较多的情况下（多写场景，冲突一般较多）。

## 线程池多线程编程

## 为什么要使用线程池？

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

线程池就是使得线程可以复用，执行完一个任务，并不被销毁，而是可以继续执行其他的任务。

Java中的ThreadPoolExecutor类

深入剖析线程池实现原理

使用示例

如何合理配置线程池的大小

## 如何合理配置线程池的大小

一般需要根据任务的类型来配置线程池大小：如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为Ncpu+1；如果是IO密集型任务，参考值可以设置为2\*Ncpu；这只是一个参考值，具体的设置还需要根据实际情况进行调整，比如可以先将线程池大小设置为参考值，在观察任务运行情况和系统负载、资源利用率来进行调整。

Java多线程间的通信方式有哪些？

|  |
| --- |
| 1 共享内存，通过synchronized关键字实现线程间的通信。while轮询。 |
| 2 wait/notify机制。 |
| 3 管道通信，使用java.io.PipedInputStream和java.io.PipedOutputStream进行通信。消息传递机制。通过管道，将一个线程中的消息发送给另一个。 |

|  |
| --- |
| **public** **class** Piped {  **public** **static** **void** main(String[] args) **throws** Exception {  PipedWriter out = **new** PipedWriter();  PipedReader in = **new** PipedReader();  // 将输出流和输入流进行连接，否则在使用时会抛出IOException异常  out.connect(in);  Thread printThread = **new** Thread(**new** Print(in), "PrintThread");  printThread.start();  **int** receive = 0;  **try** {  **while** ((receive = System.***in***.read()) != -1) {  out.write(receive);  }  } **finally** {  out.close();  }  }  **static** **class** Print **implements** Runnable {  **private** PipedReader in;  **public** Print(PipedReader in) {  **this**.in = in;  }  @Override  **public** **void** run() {  **int** receive = 0;  **try** {  **while** ((receive = in.read()) != -1) {  System.***out***.print((**char**) receive);  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  } |

## sleep()和wait()区别

|  |  |  |
| --- | --- | --- |
|  | sleep() | wait() |
| 方法 | 属于Thread类中的方法。 | 属于Object类中的方法。 |
| 执行流程 | 导致了程序暂停执行指定的时间，让出CPU给其他线程，但是它的监控状态依然保持着，当指定的时间到了又会自动恢复运行状态。 | 当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify方法后本线程才进入对象锁定池准备。 |
| 对象锁 | 调用sleep()方法的过程中，线程不会释放对象锁。 | 调用wait()方法的时候，线程会放弃对象锁 |
| 锁 | 不释放锁 | 释放锁 |

## Java里的阻塞队列

|  |  |
| --- | --- |
| ArrayBlockingQueue | 一个由数组结构组成的有界阻塞队列。 |
| LinkedBlockingQueue | 一个由链表结构组成的有界阻塞队列。 |
| PriorityBlockingQueue | 一个支持优先级排序的无界阻塞队列。 |
| DelayQueue | 一个使用优先级队列实现的无界阻塞队列。 |
| SychronousQueue | 一个不存储元素的阻塞队列。 |
| LinkedTransferQueue | 一个由链表结构组成的无界阻塞队列。 |
| LinkedBlockingQueue | 一个由链表结构组成的双向阻塞队列。 |

## Fork/Join框架

|  |
| --- |
| **public** **class** CountTask **extends** RecursiveTask<Integer> {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  **private** **static** **final** **int** ***THRESHOLD*** = 2;  **private** **int** start;  **private** **int** end;  **public** CountTask(**int** start, **int** end) {  **this**.start = start;  **this**.end = end;  }  @Override  **protected** Integer compute() {  **int** sum = 0;  // 如果任务足够小就计算任务  **boolean** canCompute = (end - start) <= ***THRESHOLD***;  **if** (canCompute) {  **for** (**int** i = start; i <= end; i++) {  sum += i;  }  } **else** {  // 如果任务大于阈值，就分裂成两个子任务计算  **int** middle = (start + end) / 2;  CountTask leftTask = **new** CountTask(start, middle);  CountTask rightTask = **new** CountTask(middle + 1, end);  // 执行子任务  leftTask.fork();  rightTask.fork();  // 等待子任务执行完，并得到其结果  **int** leftResult = leftTask.join();  **int** rightResult = rightTask.join();  // 合并子任务  sum = leftResult + rightResult;  }  **return** sum;  }  **public** **static** **void** main(String[] args) {  ForkJoinPool forkJoinPool = **new** ForkJoinPool();  // 生成一个计算任务，负责计算1+2+3+4  CountTask task = **new** CountTask(1, 4);  // 执行一个任务  Future<Integer> result = forkJoinPool.submit(task);  **try** {  System.***out***.println(result.get());  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (ExecutionException e) {  e.printStackTrace();  }  }  } |

# 互联网架构（分布式、高并发、高可用）

## 如果有一个100万的qps项目，你会从哪些方面考虑系统的设计？

## 你熟悉的分布式技术有哪些？了解他们底层的实现机制吗？

## 用过反向代理服务器吗？用来做什么？nginx负载均衡有哪些参数？

## 你熟悉的消息队列中间件的实现原理是什么？和其他消息中间件比对，有什么优势？

## Redis相关，比如问Redis集群懂多少？

## 分布式事务方案

1. 2PC两阶段提交（事务管理器）

执行多个数据库操作，比较适合两阶段提交。

J2EE提出了JTA，分布式事务管理。（Spring + JTA实现分布式事务）

微服务规范

1. TCC

Try，Confirm，Cancel

和资金相关的场景。

1. 本地消息表方案
2. 可靠消息最终一致性方案
3. 最大努力通知方案

# 设计模式

## 你平时经常用到的设计模式有哪些？

## 熟悉Reactive开发模式吗？

# 数据库

## 数据库事务

ACID：原子、一致、隔离、持久。

|  |  |
| --- | --- |
| 原子性 | Atomicity |
| 一致性 | Consistency |
| 隔离性 | Isolation |
| 持久性 | Durability |

## 数据库是怎么优化的？

## 乐观锁和悲观锁。Java代码中怎么开启悲观锁？面试官问了select for update？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞，直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。传统的关系型数据库里面就用到了很多这种锁机制，比如行锁、表锁等，读锁，写锁等，都是在做操作之前先上锁。Java中的synchronized和ReentrantLock等独占锁就是悲观锁思想的实现。

乐观锁：总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用**版本号机制**和**CAS算法**实现。乐观锁适用于多读的应用场景,这样可以提高吞吐量,像数据库提供的类似于**write\_condition机制**，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

乐观锁常见的两种实现方式：版本号机制或CAS算法。

版本号机制：

在数据库表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。

CAS算法：

即compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，不适用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-Blocking Synchronization）。

CAS算法涉及到的三个操作数：

1. 需要读写的内存值V
2. 进行比较的值A
3. 拟写入的新值B

当且仅当V的值等于A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点：

1. ABA问题。
2. 循环时间长开销大。
3. 只能保证一个共享变量的原子操作。

## MySQL和Postgresql比较优缺点？

## MySQL优化？

# Spring框架

## Spring发展的过程

第一阶段：xml配置，Spring 1.x

第二阶段：注解配置，Spring 2.x

第三阶段：Java配置，Spring 3.x到现在（Spring 4.x和Spring Boot都推荐使用Java配置）

## Spring框架概述

Spring框架是一个轻量级的企业级开发的一站式解决方案，可以基于Spring解决JavaEE开发的所有问题，提供了IoC容器、AOP、数据访问、Web开发、消息、测试等相关技术的支持。

## POJO

Plain Old Java Object，无任何限制的普通Java对象

## Spring模块

1. 核心容器（Core Container）

|  |  |
| --- | --- |
| Spring-Core | 核心工具类，Spring其他模块大量使用Spring-Core |
| Spring-Bean | Spring定义Bean的支持 |
| Spring-Context | 运行时Spring容器 |
| Spring-Context-Support | Spring容器对第三方包的集成支持 |

1. AOP

|  |  |
| --- | --- |
| Spring-AOP | 基于代理的AOP支持 |
| Spring-Aspects | 基于AspectJ的AOP支持 |

1. 消息（Messaging）

|  |  |
| --- | --- |
| Spring-Messaging | 对消息架构和协议的支持 |

1. Web

|  |  |
| --- | --- |
| Spring-Web | 提供基础的Web集成的功能，在Web项目中提供Spring的容器 |
| Spring-Webmvc | 提供基于Servlet的Spring MVC |
| Spring-WebSocket | 提供WebSocket功能 |
| Spring-Webmvc-Portlet | 提供Portlet环境支持 |

1. 数据访问/集成（Data Access/Integration）

|  |  |
| --- | --- |
| Spring-JDBC | 提供以JDBC访问数据库的支持 |
| Spring-TX | 提供编程式和声明式的失误支持 |
| Spring-ORM | 提供对对象/关系映射技术的支持 |
| Spring-OXM | 提供对对象/xml映射技术的支持 |
| Spring-JMS | 提供对JMS的支持 |

## Spring生态

Spring目前提供了大量的基于Spring的项目，可以用来更深入地降低我们的开发难度，提高开发效率。

|  |  |
| --- | --- |
| Spring Boot | 使用默认开发配置来实现快速开发 |
| Spring XD | 用来简化大数据应用开发 |
| Spring Cloud | 为分布式系统开发提供工具集 |
| Spring Data | 对主流的关系型和NoSQL数据库的支持 |
| Spring Integration | 通过消息机制对企业集成模式（EIP）的支持 |
| Spring Batch | 简化及优化大量数据的批处理操作 |
| Spring Security | 通过认证和授权保护应用 |
| Spring HATEOAS | 基于HATEOAS原则简化REST服务开发 |
| Spring Social | 与社交网络API（如Facebook、新浪微博等）的集成 |
| Spring AMQP | 对基于AMQP的消息的支持 |
| Spring Mobile | 提供对手机设备检测的功能，给不同的设备返回不同的页面的支持 |
| Spring for Android | 主要提供在Android上消费RESTful API的功能 |
| Spring Web Flow | 基于Spring MVC提供基于向导流程式的Web应用开发 |
| Spring Web Service | 提供了基于协议有限的SOAP/Web服务 |
| Spring LDAP | 简化使用LDAP开发 |
| Spring Session | 提供一个API及实现来管理用户会话信息 |

## 构建工具

目前主流的项目构建工具有：Ant、Maven、Gradle等。

Apache Maven，软件项目管理工具，基于项目对象模型（Project Object Model，POM）的概念，可用来管理项目的依赖、编译、文档等信息。

## Maven的pom.xml

|  |  |
| --- | --- |
| dependencies | 包含多个项目依赖需要使用的dependency |
| dependency | 通过groupId、artifactId以及version确定唯一的依赖 |
| groupId | 组织的唯一标识 |
| artifactId | 项目的唯一标识 |
| version | 项目的版本 |

变量定义：<properties></properties>可定义变量在dependency中的引用

依赖jar包参考库：http://mvnrepository.com

## 将Maven中心库中没有的jar包，打到本地Maven库后应用：

|  |
| --- |
| mvn install:install-file -DgroupId=com.oracle "-DartifactId=ojdbc14" "-Dversion=10.2.0.2.0" "-Dpackaging=jar" "-Dfile=D:\ojdbc14.jar" |

Spring框架四大原则

|  |
| --- |
| 使用POJO进行轻量级的最小侵入式开发 |
| 通过依赖注入和基于接口编程实现松耦合 |
| 通过AOP和默认习惯进行声明式编程 |
| 使用AOP和模板（template）减少模式化代码 |

## 声明Bean的注解

|  |  |
| --- | --- |
| @Component | 组件，没有明确的角色 |
| @Service | 在业务逻辑层（Service层）使用 |
| @Repository | 在数据访问层（dao层）使用 |
| @Controller | 在表现层（MVC->Spring MVC）使用 |

## 注入Bean的注解

|  |  |
| --- | --- |
| @Autowired | Spring提供的注解 |
| @Inject | JSR-330提供的注解 |
| @Resource | JSR-250提供的注解 |

@Autowired、@Inject、@Resource可注解在set方法上或者属性上，我习惯注解在属性上，优点是代码更少、层次更清晰。

## Autowired和Resource的区别

共同点：两者都可以写在字段和setter方法上。两者如果都写在字段上，那么就不需要写setter方法。

不同点：

@Autowired注解是按照类型（byType）装配依赖对象，默认情况下它要求依赖对象必须存在，如果允许null值，可以设置它的required属性为false。如果想要按照名称（byName）来装配，可以结合@Qualifier注解一起使用。

@Resource默认按照byName自动注入。Spring将@Resource注解的name属性解析为Bean的名字，将type属性解析为bean的类型。

* 如果使用name属性，则使用byName的自动注入策略。
* 使用type属性时，则使用byType自动注入策略。
* 如果不指定name也不指定Type属性时，将通过反射机制使用byName自动注入策略。

@Resource装配顺序

* 1 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常。
* 2 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常。
* 3 如果指定了type，则从上下文中找到匹配的唯一bean进行装配，找不到或是找到多个，都会抛出异常。
* 4 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配
* @Resource的作用相当于@Autowired，只不过@Autowired按照byType自动注入。

## Spring EL和资源调用

Spring EL是Spring表达式语言，支持在xml和注解中使用表达式，类似于JSP的EL表达式语言。

Spring开发中经常涉及调用各种资源的情况，包括普通文件、网址、配置文件、系统环境变量等，我们可以使用Spring的表达式语言实现资源的注入。

Spring主要在注解@Value的参数中使用表达式。

|  |
| --- |
| 注入普通字符 |
| 注入操作系统属性 |
| 注入表达式运算结果 |
| 注入其他Bean的属性 |
| 注入文件内容 |
| 注入网址内容 |
| 注入属性文件 |

## Bean的初始化和销毁

Bean在使用之前或者之后做些必要的操作，Spring对Bean的生命周期的操作提供了支持。

1. Java配置方式：使用@Bean的initMethod和destroyMethod（相当于xml配置的init-method和destroy-method）。initMethod和destroyMethod指定Bean类的init和destroy方法在构造函数之后，Bean销毁之前执行。
2. 注解方式：利用JSR-250的@PostConstruct和@PreDestroy。

@PostConstruct，在构造函数执行完之后执行。

@PreDestroy，在Bean销毁之前执行。

## Profile

Profile为在不同环境下使用不同的配置提供了支持（开发环境下的配置和生产环境下的配置肯定是不同的，例如，数据库配置）。

|  |
| --- |
| 通过设定Environment的ActiveProfiles来设定当前Context需要使用的配置环境。在开发中使用@Profile注解类或者方法，达到在不同情况下选择实例化不同的Bean。 |
| 通过设定jvm的spring.profiles.active参数来设置配置环境。 |
| Web项目设置在Servlet的Context parameter中。 |

## 事件（Application Event）

Spring的事件（Application Event）为Bean与Bean之间的消息通信提供了支持。当一个Bean处理完一个任务之后，希望另一个Bean知道并能做相应的处理，这时我们就需要让另外一个Bean监听当前Bean所发送的事件。

Spring的事件需要遵循如下流程：

|  |
| --- |
| （1）自定义事件，继承ApplicationEvent |
| （2）定义事件监听器，实现ApplicationListener<...> |
| （3）使用容器发布事件，注入ApplicationContext用来发布事件，使用ApplicationContext的publishEvent方法来发布。 |

## Spring Aware

Spring的依赖注入的最大亮点就是所有的Bean对Spring容器的存在是没有意识的。即你可以将你的容器替换成别的容器，如Google Guice，这时Bean之间的耦合度很低。

但是在实际项目中，不可以避免的要用到Spring容器本身的功能资源，这时你的Bean必须要意识到Spring容器的存在，才能调用Spring所提供的资源，这就是所谓的Spring Aware。其实Spring Aware本来就是Spring设计用来框架内部使用的，若使用了Spring Aware，你的Bean将会和Spring框架耦合。

Spring提供的Aware接口

|  |  |
| --- | --- |
| BeanNameAware | 获取到容器中Bean的名称 |
| BeanFactoryAware | 获取当前bean factory，这样可以调用容器的服务 |
| ApplicationContextAware | 当前的Application Context，这样可以调用容器的服务 |
| MessageSourcesAware | 获取Message source，这样可以获得文本信息 |
| ApplicationEventPublisherAware | 应用事件发布器，2.5节的DemoPublisher也可实现这个接口来发布事件 |
| ResourceLoadAware | 获得资源加载器，可以获取外部资源文件 |

Spring Aware的目的是为了让Bean获得Spring容器的服务。因为ApplicationContext接口集成了MessageSource接口、ApplicationEventPublisher接口和ResourceLoader接口，所以Bean集成ApplicationContextAware可以获得Spring容器的所有服务，但原则上我们还是用到什么接口，就实现什么接口。

## 多线程

Spring通过任务执行器（TaskExecutor）来实现多线程和并发编程。使用ThreadPoolTaskExecutor可实现一个基于线程池的TaskExecutor。而实际开发中任务一般是非阻塞的，即异步的，所以我们要在配置类中通过@EnableAsync开启对异步任务的支持，并通过在实际执行的Bean的方法中使用@Async注解来声明其是一个异步任务。

## 计划任务

从Spring 3.1开始，计划任务在Spring中的实现变得异常简单。首先通过在配置类注解@EnableScheduling来开启对计划任务的支持，然后在要执行计划任务的方法上注解@Scheduled，声明这是一个计划任务。

Spring通过@Scheduled支持多种类型的计划任务，包含cron、fixDelay、fixRate等。

|  |
| --- |
| @Scheduled(cron = "0 28 11 ? \* \*") |

cron是UNIX和类UNIX（Linux）系统下的定时任务。

## 条件注解@Conditional

## 组合注解与元注解

## @Enable×注解的工作原理

|  |  |
| --- | --- |
| @EnableAspectJAutoProxy | 开启对AspectJ自动代理的支持。 |
| @EnableAsync | 开启异步方法的支持。 |
| @EnableScheduling | 开启计划任务的支持。 |
| @EnableWebMvc | 开启Web MVC的配置支持。 |
| @EnableConfigurationProperties | 开启对@ConfigurationProperties注解配置Bean的支持。 |
| @EnableJpaRepositories | 开启对Spring Data JPA Repository的支持。 |
| @EnableTransactionManagement | 开启注解式事务的支持。 |
| @EnableCaching | 开启注解式的缓存支持。 |

# Spring MVC 4.x

## 基本概念

MVC：Model + View + Controller（数据模型+视图+控制器）

三层架构：Presentation tier + Application tier + Data tier（展现层+应用层+数据访问层）

实际上MVC只存在三层架构的展现层。

## 使用Servlet 3.0+无web.xml的配置方式

Spring MVC里实现WebApplicationInitializer接口便可实现等同于web.xml的配置。

Spring Boot的页面就放置在src/main/resources下。

1 WebApplicationInitializer是Spring提供用来配置Servlet 3.0+配置的接口，从而实现了替代web.xml的位置。实现此接口将会自动被SpringServletContainerInitializer（用来启动Servlet 3.0容器）获取到。

2 新建WebApplicationContext，注册配置类，并将其和当前ServletContext关联。

3 注册Spring MVC的DispatcherServlet

|  |
| --- |
| mvn package  cd /java/tomcat/bin  ./startup.sh  ./shutdown.sh  ps -ef | grep tomcat  kill -9 [pid]  cd logs  tail -f catalina.out |

Spring MVC的常用注解

|  |  |
| --- | --- |
| @Controller |  |
| @RequestMapping |  |
| @ResponseMapping |  |
| @RequestBody |  |
| @PathVariable |  |
| @RestController |  |

添加Jackson及相关依赖，获得对象和json或xml之间的转换。

# Spring Boot基础

Spring Boot的一项重要工作就是让Spring不再成为你成功路上的绊脚石。

Spring Boot使用“习惯优于配置”的理念让项目快速运行起来。使用Spring Boot很容易创建一个独立运行（运行jar，内嵌Servlet容器）、准生产级别的基于Spring框架的项目，使用Spring Boot你可以不用或者只需要很少的Spring配置。

|  |
| --- |
| mvn spring-boot:run |

## 入口类和@SpringBootApplication

Spring Boot通常有一个名为×Application的入口类，入口类里有一个main方法，这个main方法其实就是一个标准的Java应用的入口方法。

|  |  |
| --- | --- |
| @SpringBootApplication | @Configuration |
| @EnableAutoConfiguration |
| @ComponentScan |

## 关闭特定的自动配置

|  |
| --- |
| @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class}) |

## 定制Banner

修改Banner

在src/main/resources下新建一个banner.txt

关闭Banner

|  |
| --- |
| SpringApplication.*run*(Ch522Application.**class**, args); |

|  |
| --- |
| SpringApplication app = **new** SpringApplication(Ch522Application.**class**);  app.setBannerMode(Banner.Mode.***OFF***);  app.run(args); |

使用fluent API修改：

|  |
| --- |
| **new** SpringApplicationBuilder(Ch522Application.**class**)  .bannerMode(Banner.Mode.***OFF***)  .run(args); |

## SpringBoot的配置文件

全局的配置文件application.properties或application.yml

SpringBoot启动会扫描一下位置的application.properties或者application.yml文件作为SpringBoot的默认配置文件。

|  |
| --- |
| **-file:./config/**  **-file:./**  **-classpath:/config/**  **-classpath:/** |

按照优先级从高到低的顺序，所有位置的文件都会被加载，高优先级配置内容会覆盖低优先级配置内容。

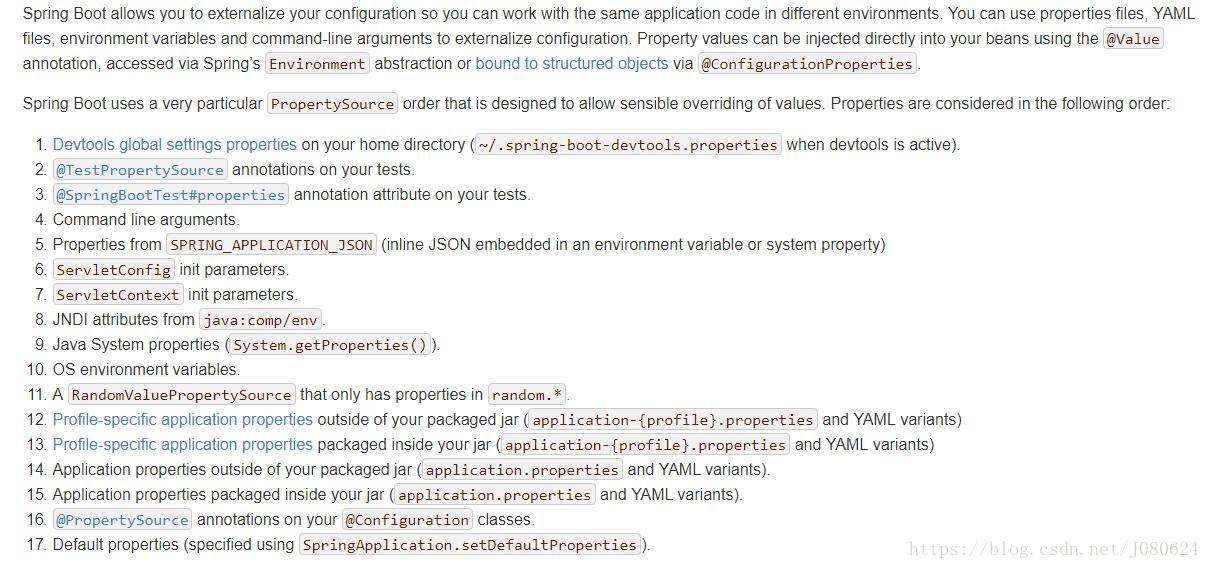
从四个位置全部加载配置文件，如果高优先级中配置文件属性与低优先级配置文件不冲突的的属性，则会共同存在——**互补配置**。

也可以通过*spring.config.location*来改变默认配置：

|  |
| --- |
| java -jar spring-boot-02-config-02-0.0.1-SNAPSHOT.jar **- -spring.config.location=D:/applicaton.properties** |

项目打包好以后，可以使用命令行参数的形式，在启动项目的时候来指定配置文件的新位置。指定配置文件和默认加载的这些配置文件共同起作用形成互补配置。

## 配置文件加载顺序



SpringBoot可以从以下位置加载配置：优先级从高到低；高优先级的配置覆盖低优先级的配置，所有的配置会形成互补配置。

1. 命令行参数

* 所有的配置都可以在命令行上进行指定；
* 多个配置用空格分开；- -配置项=值

|  |
| --- |
| java -jar \*.jar **--server.port=8087** **--server.context-path=/abc** |

1. 来自java:comp/env的JNDI属性
2. Java系统属性（System.getProperties()）
3. 操作系统环境变量
4. RandomValuePropertySource配置的random.\*属性值
5. jar包外部的application-{profile}.properties或application.yml（带spring.profile）配置文件
6. jar包内部的application-{profile}.properties或application.yml（带spring.profile）配置文件
7. jar包外部的application.properties或application.yml（不带spring.profile）配置文件
8. jar包内部的application.properties或application.yml（不带spring.profile）配置文件

|  |
| --- |
| 由jar包外向jar包内进行寻找，优先加载带profile的，再加载不带profile的。 |

1. @Configuration注解类上的@PropertySource
2. 通过SpringApplication.setDefaultProperties指定的默认属性

## SpringBoot的Property

application.yml

|  |
| --- |
| post:  type: T  path: C:\\nckm\\csv\\admission-data.csv  url: http://127.0.0.1:8080/pyxis/medes/adt/admission |

|  |
| --- |
| @Value("${post.type}")  **private** String type;  @Value("${post.path}")  **private** String path;  @Value("${post.url}")  **private** String url; |

# Spring Cloud

## SpringCloud各个组件的运行机制是什么？

## SpringMVC的原理？

## Spring事务相关？

## Spring Security的底层实现原理？

## 动态代理的实现

## SpringJPA、Hibernate和Mybatis的区别、优缺点

## 微服务的优缺点

## 持久层设计有哪些特点：

1. 数据存储逻辑的分离，提供抽象化的数据访问接口。
2. 数据访问底层实现的分离，可以在不修改代码的情况下切换底层实现。
3. 资源管理和调度的分离，在数据访问层实现统一的资源调度。
4. 数据抽象，提供面向对象的数据操作。

# Docker

Docker镜像检索

|  |
| --- |
| docker search 镜像名  docker search redis |

镜像下载

|  |
| --- |
| docker pull 镜像名  docker pull redis |

镜像列表

|  |
| --- |
| docker images |

镜像删除

|  |
| --- |
| docker rmi image-id |

删除所有镜像

|  |
| --- |
| docker rmi ${docker image -q} |

最简单的运行镜像为容器的命令

|  |
| --- |
| docker run - - name container-name -d image-name  docker run - - name test-redis -d redis |

Docker会为我们的容器生成唯一的标识。

容器列表

|  |
| --- |
| docker ps  docker ps -a |

停止容器

|  |
| --- |
| docker stop container-name/container-id  docker stop test-redis |

启动容器

|  |
| --- |
| docker start container-name/container-id  docker start test-redis |

端口映射

|  |
| --- |
| docker run -d -p 6378:6379 - -name port-redis redis |

删除容器

|  |
| --- |
| docker rm container-id |

删除所有容器

|  |
| --- |
| docker rm ${docker ps -a -q} |

容器日志

|  |
| --- |
| docker logs container-name/container-id  docker logs port-redis |

登录容器(登录访问当前容器，登录后我们可以在容器中进行常规的Linux系统操作命令，还可以使用exit命令退出登录)

|  |
| --- |
| docker exec -it container-id/container-name bash |

## Docker Oracle

容器提供的安装信息

|  |  |
| --- | --- |
| hostname | localhost |
| port | 1521 |
| SID | XE |
| username | system/sys |
| password | oracle |

管理界面访问

|  |  |
| --- | --- |
| url | http://localhost:9090/apex |
| workspace | internal |
| username | admin |
| password | oracle sai199311 |

|  |
| --- |
| mvn install:install-file -DgroupId=com.oracle "-DartifactId=ojdbc6" "-Dversion=11.2.0.2.0" "-Dpackaging=jar" "-Dfile=./ojdbc6.jar" |

# Spring Boot

Spring Data REST

声明式事务（Spring的事务机制）

数据缓存Cache（Spring缓存支持）

切换缓存技术

非关系型数据库NoSQL

MongoDB

Redis

## 参考文档

[Spring-boot 2.0.0.RELEASE](https://docs.spring.io/spring-boot/docs/2.0.0.RELEASE/reference/htmlsingle/)

# Spring Web

## Thymeleaf模板引擎

Spring Boot提供了大量模板引擎，包括FreeMarker、Groovy、Thymeleaf、Velocity和Mustache，Spring Boot中推荐使用Thymeleaf作为模板引擎，因为Thymeleaf提供了完美的Spring MVC的支持。

Thymeleaf是一个Java类库，它是一个xml/xhtml/html5的模板引擎，可以作为MVC的Web应用的View层。

## 配置Tomcat

配置Servlet容器

|  |  |
| --- | --- |
| server.port= | #配置程序端口，默认为8080 |
| server.session-timeout= | #用户会话session过期时间，以秒为单位 |
| server.context-path= | #配置访问路径，默认为/ |

配置Tomcat

|  |  |
| --- | --- |
| server.tomcat.uri-encoding= | #配置Tomcat编码，默认为UTF-8 |
| server.tomcat.uri.compression= | #Tomcat是否开启压缩，默认为关闭off |

特定配置

|  |  |
| --- | --- |
| Tomcat | TomcatEmbeddedServletContainerFactory |
| Jetty | JettyEmbeddedServletContainerFactory |
| Undertow | UndertowEmbeddedServletContainerFactory |

替换Tomcat为Jetty

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jetty</artifactId>  </dependency> |

替换Tomcat为Undertow

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-undertow</artifactId>  </dependency> |

## Favicon配置

默认的Favicon：Spring Boot提供了一个默认的Favicon，每次访问应用的时候都能看到。

关闭Favicon

可以在application.properties中设置关闭Favicon，默认为开启

|  |
| --- |
| spring.mvc.facicon.enabled=false |

设置自己的Favicon

需要设置自己的Favicon，则只需要将自己的favicon.ico（文件名不能变动）文件放置在

类路径根目录下

类路径META-INF/resources/下

类路径resources/下

类路径static/下

类路径public/下

这里将favicon.ico放置在src/main/resources/static下

## WebSocket

Spring Boot对内嵌的Tomcat（7或者8）、Jetty9和Undertow使用WebSocket提供支持。配置源码存于org.springframework.boot.autoconfigure.websocket下。

Spring Boot为WebSocket提供的start pom是spring-boot-starter-websocket。

广播式：即服务端有消息时，会将消息发送所有连接了当前endpoint的浏览器。

|  |
| --- |
| @Configuration  @EnableWebSocketMessageBroker  **public** **class** WebSocketConfig **extends** AbstractWebSocketMessageBrokerConfigurer {  @Override  **public** **void** registerStompEndpoints(StompEndpointRegistry registry) {  registry.addEndpoint("/endpointWisely").withSockJS();  }  @Override  **public** **void** configureMessageBroker(MessageBrokerRegistry registry) {  registry.enableSimpleBroker("/topic");  }  } |

1. 通过@EnableWebSocketMessageBroker注解开启使用STOMP协议来传输基于代理（message broker）的消息，这时控制器支持使用@MessageMapping，就像使用@RequestMapping一样。
2. 注册STOMP协议的节点（endpoint），并映射到指定的URL。
3. 注册一个STOMP的endpoint，并指定使用SockJS协议。
4. 配置消息代理（Message Broker）。
5. 广播式应配置一个/topic消息代理。
6. 当浏览器向服务器发送请求时，通过@MessageMapping映射/welcome这个地址，类似于@RequestMapping。
7. 当服务端有消息时，会对订阅了@SendTo中的路径的浏览器发送消息。
8. 添加脚本。将stomp.min.js（STOMP协议的客户端脚本）、sockjs.min.js（SockJS的客户端脚本）以及JQuery放置在src/main/resources/static下。

# J2EE

企业Java组件（Enterprise JavaBean，简称EJB）。JNDI、RMI、JMS。

# Java编程思想

面向对象程序设计（Object-oriented Programming，OOP）。

所有编程语言都提供抽象机制。

OOP允许根据问题来描述问题，而不是根据运行解决方案的计算机来描述问题。

面向对象语言的五个基本特征（这些特征表现了一种纯粹的面向对象程序设计方式）：

1. 万物皆为对象。
2. 程序是对象的集合，它们通过发送消息来告知彼此所要做的。
3. 每个对象都有自己的有其他对象所构成的存储。
4. 每个对象都拥有其类型。（每个对象都是某个类class的一个实例instance）。
5. 某一个特定类型的所有对象都可以接收同样的消息。这种可替代性是OOP中最强有力的概括之一。

# 算法

## 有一个变量list，元素存放集群的实例个数，有一个变量batch批次，根据批次均匀部署应用到集群的所有实例？

## 10亿个数字的文件，只有128M的内存，怎么对这个文件进行排序？

## 倒序数组，查找一个数，倒序数组就是：6,7,8,1,2,3，LeetCode上的

# 服务器

## Tomcat调优？

## Nginx和Tomcat的比较？

## Tomcat的架构实现原理？

# Java新特性

## Jdk8的特性了解多少？

# 分布式缓存

## 缓存与数据库双写不一致

## 缓存雪崩

缓存雪崩，是指在某一个时间段，缓存集中过期失效。

例如在写文本的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么多了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

解决方案：采取不同分类的商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

补充案例：集中过期倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。这对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

## 缓存穿透

缓存穿透，是指查询一个数据库一定不存在的数据。

如果传入的参数为-1，会是怎么样？这个-1，就是一定不存在的对象。就会每次都去查询数据库，而每次查询都是空，每次又都不会进行缓存。假如有恶意攻击，就可以利用这个漏洞，对数据库造成压力，甚至压垮数据库。即便是采用UUID，也很容易找到一个不存在的KEY，进行攻击。

解决方案：采用缓存空值的方式，如果从数据库查询的对象为空，也放入缓存，只是设定的缓存过期时间较短，比如设置为60秒。

## 缓存并发竞争

## 缓存击穿

缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开一个洞。

解决方案：对主打商品都是早早的做好了准备，让缓存永不过期。即便某些商品自己发酵成了爆款，也是直接设为永不过期就好了。

# Zookeeper

## zk都有哪些使用场景？

1. 分布式协调
2. 分布式锁
3. 元数据或配置信息管理
4. HA高可用

## 一般实现分布式锁都有哪些方式？

# Gradle

## Gradle是一个好用的构建工具，优点有哪些？

1. 配置相关依赖代码量少，不会像maven一样xml过多。
2. 打包编译测试发布都有，而且使用起来方便。
3. 利用自定义的任务可以完成自己想要的功能。

Gradle提供了一个域特定语言（DSL），用于描述构建。它使用Groovy语言，使其更容易来形容和构建。Gradle中每一个构建脚本使用UTF-8进行编码保存，并命名为**build.gradle**。

