## Core Java

1. hashcode相等两个类一定相等吗? equals呢? ?相反呢

|  |
| --- |
| <https://www.cnblogs.com/ChrisMurphy/p/5005906.html> （equals和==的区别，注意对string类的研究，equals(), ==, intern()方法的特点）  java.lang.String的intern()方法"abc".intern()方法的返回值还是字符串"abc"，表面上看起来好像这个方 法没什么用处。但实际上，它做了个小动作：检查字符串池里是否存在"abc"这么一个字符串，如果存在，就返回池里的字符串；如果不存在，该方法会 把"abc"添加到字符串池中，然后再返回它的引用。  <https://www.cnblogs.com/keyi/p/7119825.html> （hashcode（）和equals（）的作用、区别、联系）  所有对于需要大量并且快速的对比的话如果都用equal()去做显然效率太低，所以解决方式是（类似的hashMap, hashSet, hashTable也是这种思想）：  每当需要对比的时候，首先用hashCode()去对比，如果hashCode()不一样，则表示这两个对象肯定不相等（也就是不必再用equal()去再对比了）, 如果hashCode()相同，此时再对比他们的equal()，如果equal()也相同，则表示这两个对象是真的相同了，这样既能大大提高了效率也保证了对比的绝对正确性！  在Java中任何一个对象都具备equals(Object obj)和hashCode()这两个方法，因为他们是在Object类中定义的。 equals(Object obj)方法用来判断两个对象是否“相同”，如果“相同”则返回true，否则返回false。 hashCode()方法返回一个int数，在Object类中的默认实现是“将该对象的内部地址转换成一个整数返回”。  常用的hash算法，object, string，Integer各自实现介绍？ |

1. 请用至少四种写法写一个单例模式呗????

<https://blog.csdn.net/jason0539/article/details/23297037/> （普通玩法）

第一次实例化（不考虑线程安全），考虑线程安全，考虑读不加锁，静态内部类实现（final的作用，真正延迟加载）

对于单例模式的几种实现方式，知道饿汉式和懒汉式的区别，线程安全，资源加载的时机，还有懒汉式为了实现线程安全的3种方式的细微差别。

加了volatile关键字来声明单例对象的原因？

<http://www.iteye.com/topic/652440> （单例的高级玩法， 延迟加载，sychronized，volatile, threadLocal）

1. 接口的幂等性概念，接口如何处理重复请求？

|  |
| --- |
| <http://blog.csdn.net/jks456/article/details/71453053>  如何保证幂等性？如果使用全局唯一ID，就是根据业务的操作和内容生成一个全局ID，在执行操作前先根据这个全局唯一ID是否存在，来判断这个操作是否已经执行。如果不存在则把全局ID，存储到存储系统中，比如数据库、redis等。如果存在则表示该方法已经执行。  使用全局唯一ID是一个通用方案，可以支持插入、更新、删除业务操作。但是这个方案看起来很美但是实现起来比较麻烦，下面的方案适用于特定的场景，但是实现起来比较简单：   1. 去重表 2. Insert or update 3. 多版本控制（使用版本号） 4. 通过状态字段控制   <https://www.cnblogs.com/dingcee/p/5047752.html> |

1. Maven出现版本冲突如何解决？

<https://www.cnblogs.com/godtrue/p/6220512.html>

1. 在工作中遇到哪些设计模式？如何应用的
2. **深入分析几个设计模式**
3. 增强for循环的底层原理？

<https://blog.csdn.net/H_Gao/article/details/52901297>

## IO问题

1. 熟悉io吗，与nio的区别，阻塞和非阻塞的区别
2. 使用无界阻塞队列会出现什么问题？
3. Java NIO的使用

|  |
| --- |
| <http://www.importnew.com/22623.html> (基本概念解释详细)  现在的java应用程序更多的是受IO的束缚，也就是将时间花在等待数据传输上。现在有了NIO，就可以减少IO的等待时间，从而提升IO的效率。  传统的IO和NIO的区别，传统的IO又称BIO，即阻塞式IO，NIO就是非阻塞IO了。还有一种AIO就是异步IO，这里不加阐述了。  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）。  Buffer的属性和方法？Channel？选择器？  通道是一种途径,借助该途径,可以用最小的总开销来访问操作系统本身的 I/O 服务。缓冲区则是通道内部用来发送和接收数据的端点。通道channel充当连接I/O服务的导管，  selector充当一个监视者，您需要将之前创建的一个或多个可选择的通道注册到选择器对象中。  <http://blog.csdn.net/a953713428/article/details/64907250> （总结的更好理解）  总的来说java 中的IO 和NIO的区别主要有3点：  IO是面向流的，NIO是面向缓冲的；  IO是阻塞的，NIO是非阻塞的；  IO是单线程的，NIO 是通过选择器来模拟多线程的；  NIO在基础的IO流上发展处新的特点，分别是：内存映射技术，字符及编码，非阻塞I/O和文件锁定。下面我们分别就这些技术做一些说明。  内存映射文件它虽然最终也是要从磁盘读取数据，但是它并不需要将数据读取到OS内核缓冲区，而是直接将进程的用户私有地址空间中的一部分区域与文件对象建立起映射关系，就好像直接从内存中读、写文件一样，速度当然快了。  MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ\_ONLY,0,channel.size());  内存映射文件的优点? 编码？  一般来说 I/O 模型可以分为：同步阻塞，同步非阻塞，异步阻塞，异步非阻塞 四种IO模型。模型解释？  同步非阻塞 IO:  在此种方式下，用户进程发起一个 IO 操作以后可以返回做其它事情，但是用户进程需要时不时的询问 IO 操作是否就绪，这就要求用户进程不停的去询问，从而引入不必要的 CPU 资源浪费。其中目前 JAVA 的 NIO 就属于同步非阻塞 IO 。  对于非阻塞IO就有一个非常严重的问题，在while循环中需要不断地去询问内核数据是否就绪，这样会导致CPU占用率非常高，因此一般情况下很少使用while循环这种方式来读取数据。所以这就不得不说到下面这个概念–多路复用IO模型。  在多路复用IO模型中，会有一个线程不断去轮询多个socket的状态，只有当socket真正有读写事件时，才真正调用实际的IO读写操作。因为在多路复用IO模型中，只需要使用一个线程就可以管理多个socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有socket读写事件进行时，才会使用IO资源，所以它大大减少了资源占用。  NIO 的非阻塞 I/O 机制是围绕 **选择器和通道**构建的。 Channel 类表示服务器和客户机之间的一种通信机制。Selector 类是 Channel 的多路复用器。 Selector 类将传入客户机请求多路分用并将它们分派到各自的请求处理程序。 NIO设计背后的基石是反应器(Reactor)设计模式。NIO的设计则是完全按照Reactor模式来设计的。Selector发现某个channel有数据时，会通过SelectorKey来告知，然后实现事件和handler的绑定。  利用多路复用机制避免了线程的阻塞，提高了连接的数量。一个线程就可以管理多个socket，只有当socket真正有读写事件发生才会占用资源来进行实际的读写操作。虽然多线程+阻塞IO 达到类似的效果，但是由于在多线程 + 阻塞IO 中，每个socket对应一个线程，这样会造成很大的资源占用，并且尤其是对于长连接来说，线程的资源一直不会释放，如果后面陆续有很多连接的话，就会造成性能上的瓶颈。  另外多路复用IO为何比非阻塞IO模型的效率高是因为在非阻塞IO中，不断地询问socket状态时通过用户线程去进行的（需要用户自己去实现询问），而在多路复用IO中，轮询每个socket状态是内核在进行的，这个效率要比用户线程要高的多。  文件锁定：NIO中的文件通道（FileChannel）在读写数据的时候主 要使用了阻塞模式，它不能支持非阻塞模式的读写，而且FileChannel的对象是不能够直接实例化的， 他的实例只能通过getChannel()从一个打开的文件对象上边读取（RandomAccessFile、 FileInputStream、FileOutputStream），并且通过调用getChannel()方法返回一个Channel对象去连接同一个文件，也就是针对同一个文件进行读写操作。FileChannel的锁模型主要针对的是每一个文件，并不是每一个线程和每一个读写通道，也就是以文件为中心进行共享以及独占，也就是文件锁本身并不适合于同一个JVM的不同线程之间。 |

1. 最近做的比较熟悉的项目是哪个，画一下项目的技术架构
2. 具体的处理方案是什么
3. mapReduce过程
4. 多路归并的时间复杂度
5. 海量URL去重类问题
6. 倒排索引的原理(hadoop里面的内容)
7. 对分词技术的了解
8. 场景问题，大量用户数据如何在内存中排序和去重？缓存机器增删如何对系统影响最小？一致性哈希实现？

<https://blog.csdn.net/hustwht/article/details/52181632>

<http://itindex.net/detail/37638-%E5%A4%A7%E6%95%B0%E6%8D%AE-%E6%8E%92%E5%BA%8F-%E7%9B%B8%E5%85%B3>

1. 实际场景，典型的TOP K问题
2. 海量登录日志如何排序和处理？
3. 设计一个对外服务的接口实现类，在1，2，3这三个主机（对应不同ip）上实现负载均衡和顺序轮询机制（考虑并发）

是不是可以考虑使用一致性hash，来实现负载均衡

**上述这些典型的问题, 都可以在这篇博客中找到答案或思路.**

## 架构设计

**高并发高可用**

1. 如何设计一个高并发高可用系统？
2. 如何限流？工程怎么做？说一下具体实现
3. 缓存如何使用的？缓存使用不当会造成什么后果？
4. 如何熔断？熔断框架有哪些？具体实现原理知道吗？
5. 如何降级？如何进行系统拆分？如何数据库拆分？

## JVM

1. JVM垃圾回收机制，何时触发MinorGC和FGC等操作？

|  |
| --- |
| 博客: <https://www.cnblogs.com/wabi87547568/p/5282892.html> （JVM的组成、垃圾回收机制，重点看第2节）  虚拟机的构成? 各种垃圾收集算法？system.gc()和finalize()方法？  JVM定义了控制Java代码解释执行和具体实现的五种规格，因此把JVM分成了6个部分：JVM解释器、指令系统、寄存器、栈、存储区和碎片回收区。  分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。  目前大部分垃圾收集器对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。而由于老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。  新年代：新创建的对象都存放在这里。因为大多数对象很快变得不可达，所以大多数对象在年轻代中创建，然后消失。当对象从这块内存区域消失时，我们说发生了一次“minor GC”。  老年代：没有变得不可达，存活下来的年轻代对象被复制到这里。这块内存区域一般大于年轻代。因为它更大的规模，GC发生的次数比在年轻代的少。对象从老年代消失时，我们说“major GC”（或“full GC”）发生了。  上图中的永久代(permanent generation)也称为“方法区(method area)”，他存储class对象和字符串常量。所以这块内存区域不是永久存放从老年代存活下来的对象的。在这块内存中有可能发生垃圾回收。发生在这里垃圾回收也被称为major GC。  博客: <http://blog.csdn.net/yhyr_ycy/article/details/52566105>（GC详解及Minor GC和Full GC触发条件总结）  Java采用的是类似于树形结构的可达性分析法来判断对象是否还存在引用。  具体过程：当GC线程启动时，会通过可达性分析法把Eden区和From Space区的存活对象复制到To Space区，然后把Eden Space和From Space区的对象释放掉。当GC轮训扫描To Space区一定次数后，把依然存活的对象复制到老年代，然后释放To Space区的对象。对于用可达性分析法搜索不到的对象，GC并不一定会回收该对象。要完全回收一个对象，至少需要经过两次标记的过程。  这篇文章对JVM内存的解释比较好！！  程序计数器：线程私有。是一块较小的内存，是当前线程所执行的字节码的行号指示器。是Java虚拟机规范中唯一没有规定OOM（OutOfMemoryError）的区域。  方法区：所有线程共享。用于存储已被虚拟机加载的类信息、常量、静态变量等数据。又称为非堆（Non – Heap）。**方法区又称“永久代”。**GC很少在这个区域进行，但不代表不会回收。这个区域回收目标主要是针对常量池的回收和对类型的卸载。当内存申请大于实际可用内存，抛OOM。  本地方法栈：线程私有。与Java栈类似，但是不是为Java方法（字节码）服务，而是为本地非Java方法服务。也会抛StackOverflowError和OOM。  Minor GC触发条件：当Eden区满时，触发Minor GC。  Full GC触发条件：  （1）调用System.gc时，系统建议执行Full GC，但是不必然执行  （2）老年代空间不足  （3）方法区空间不足  **（4）**通过Minor GC后进入老年代的平均大小大于老年代的可用内存  **（5）**由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小 |

1. 新生代和老年代的内存回收机制（见上题解答）

标记-清除, 复制, 标记-整理

1. Eden和Survivor的比例分配

<https://blog.csdn.net/lojze_ly/article/details/49456255> 这里对新生代和老年代的解释比较到位！！

1. JVM内存分代

Java堆分为新生代和老年代, 新生代怎么样, 老年代怎么样？还有永久代？

见上题分析。

1. Jvm是否熟悉，说说类加载过程，有哪些操作？GC和内存管理，平时在tomcat里有没有相关的配置？

|  |
| --- |
| 类加载: [*http://blog.csdn.net/fgets/article/details/52934178*](http://blog.csdn.net/fgets/article/details/52934178)  [*http://hammer.coding.me/2016/10/26/jvm-1/*](http://hammer.coding.me/2016/10/26/jvm-1/)  这篇文章解释的非常详细，类加载过程做了什么？验证阶段包含哪些？整个类的生命周期？  相对于类加载过程的其他阶段，加载阶段是开发期相对来说可控性比较强，该阶段既可以使用系统提供的类加载器完成，也可以由用户自定义的类加载器来完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。  类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括了：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（using）、和卸载（Unloading）七个阶段。  Tomcat方面的问题：<http://michaeltangbin.iteye.com/blog/1133386>  这篇文章对tomcat平时出现较多的虚拟机内存方面问题，做了非常专业的讲解，包含解决方案和经验公式！！  <https://blog.csdn.net/axbyc1234/article/details/52806845>  这篇文章对tomcat优化配置，介绍比较详细，平时可以试一下！除了有关jvm的优化，还有日志，压缩，线程池，连接等的优化配合！  jmap –heap pid检查内存分配情况：  jinfo –flags pid检查参数配置情况： |

1. Java8的内存分代改进
2. 深入分析classLoader，双亲委派机制✔

|  |
| --- |
| 关于jvm类加载过程，以及类加载器的委派机制，具体的分析及论述，参考《深入理解JVM》书中第七章！！***这里介绍的比博客更加清楚易懂。***  什么时候加载没有强制约束，但是严格规定了有且只有5种情况必须立即对类进行初始化？（new, getstatic, putstatic, invokestatic），P211有个被动引用的案例很好。   1. 子类调用父类静态变量不会使子类初始化， 2. 通过数组定义来引用类，不会触发此类的初始化。 3. 被动使用类字段演示三：常量在编译阶段会存入调用类的常量池中，本质上没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。P213   接口初始化过程与类初始化过程的区别，见P214，接口只有在真正使用到父接口的时候，才会对父接口初始化。  加载阶段，加载二进制字节流很灵活，在实际中产生意义，见P214。相对于类加载过程其他阶段，加载阶段中获取类二进制字节流动作是开发人员可控性最强的。  验证阶段，主要完成4个检验：文件格式验证，元数据验证、字节码验证、符号引用验证。注意，只有先通过文件格式验证，字节流才会进入内存的方法区中进行存储，而后面的3个验证阶段，全是基于方法区的存储结构进行，不会直接操作字节流！！  符号验证，发生在虚拟机将符号引用转化为直接引用的时候，可以看做是对类自身之外（常量池中的各种符号引用）的信息进行匹配性校验。需要校验的内容？P219, 符号验证目的是确保解析动作正常执行。  准备阶段的核心，伪类变量分配内存，并设置类变量初始值，注意这个初始值是什么？P219。但是如果类字段有常量属性，那么准备阶段就会初始化为指定的值。  解析阶段，是将常量池内的符号引用，替换为直接引用的过程。符号引用和直接引用到底是什么？P220。主要讲解，1.类或接口解析，2.字段解析，3.类方法解析，4.接口方法解析。  到了初始化阶段，才真正开始执行类中定义的java代码。<clinit>方法的有关说明，见P225. 代码清单7-5有意思？<clinit>方法对于类和接口来说不是必须的，如果一个类中没有静态语句块也没有对变量的赋值操作，那么编译器可以不为这个类生成<clinit>方法。接口与类在初始化的时候，区别？见P226。虚拟机会保证一个类的<clinit>方法在多线程环境中正确地加锁、同步。  比较两个类是否相等，只有这两个类是由同一个类加载器加载的前提下才有意义。否则，即使这两个类来源于同一个class文件，被同一个虚拟机加载，只要类加载器不同，那必定不相等。  双亲委派模型：7.4.2章节。双亲委派模型的工作过程和好处？P231。 |

1. 对java内存模型的理解，以及在并发中的应用✔

|  |
| --- |
| 内存模型在并发中的应用，也应该参考书中所讲，见第12、13章节。  12.3.2定义了内存间的交互操作。  12.3.3 对于volatile变量有特殊的规则。  12.3.4 对于long和double变量有特殊的规则，非原子性协定，  12.3.5 原子性、可见性、有序性  P376 有个java内存模型的天然先行发生原则，比较详细！！  12.4.1java与线程  程序一般不会直接使用内核线程，而是使用内核线程的一种高级接口，轻量级进程，就是我们通常意义上说的线程。  内核线程实现，用户线程实现，使用用户线程加轻量级进程混合实现、java线程实现。  12.4.2 java线程调度：协同式线程调度和抢占式线程调度，概念？各自的优缺点？线程优先级？P382。5种线程状态？  比较严谨地定义“线程安全”？P386.  13.2.1 java语言中的线程安全。各种操作共享数据分为5类：不可变、绝对线程安全、相对线程安全、线程兼容、线程对立。  13.2.2 线程安全的实现方法：1. 互斥同步；2. 非阻塞同步；3.无同步方案，包括：可重入代码，线程本地存储（threadLocal storage，这个很重要！！ P397）。  P392有个对synchronized和reentrantLock性能的比较，提倡在synchronized能满足需要的情况下，优先使用。  P394对cas的讲解！P396有对CAS逻辑漏洞的分析？  13.3章节：锁优化！！包含各种锁优化技术：自旋锁和自适应自旋；锁消除；锁粗化；轻量级锁；偏向锁。  <https://blog.csdn.net/qq_33938256/article/details/52584863#%E7%89%A9%E7%90%86%E6%9C%BA%E4%B8%AD%E7%9A%84%E5%B9%B6%E5%8F%91%E7%A1%AC%E4%BB%B6%E6%95%88%E7%8E%87%E4%B8%8E%E4%B8%80%E8%87%B4%E6%80%A7>  这篇文章相当于对书中重点进行了概述，刚好对比来看！！ |

1. Jvm内存模型

博客：<http://blog.csdn.net/zjf280441589/article/details/53437703>

1. Jstack，jmap，jutil分别的意义？如何线上排查jvm的相关问题

|  |
| --- |
| 参考书中第四章，书中介绍的比较系统权威！  常用的虚拟机命令，平时也是非常实用的！  Jps: 查询虚拟机进程  Jstat：查询指定的虚拟机进程状况信息，可以指定查询哪方面的信息，如gc, compile  Jinfo: java配置信息工具。可以实时地查看和调整虚拟机的各项参数。  JMap：用于生成堆转储快照。还有一些暴力的方式用来获取堆转储文件，见P107。  Jhat：虚拟机堆转储快照分析工具。与jmap搭配使用。Jhat内置了一个http/html服务器，生成dump文件后，可以在浏览器中查看。<http://localhost:7000/>.不是很实用  Jstack: java堆栈跟踪工具。用于生成虚拟机当前时刻的线程快照。一般称为threadDump或javacore文件。这个还比较实用。在分析多线程或线程问题很有用。代码清单4-5，可以利用Thread.getAllstackTraces()方法，设计一个网页，可以随时查看线程情况。  JDK可视化工具：  Jconsole, 和 visualVM |

1. 请介绍一下JVM内存模型? ? 用过什么垃圾回收器？都说说呗?
2. 线上发送频繁Full gc如何处理?  CPU 使用率过高怎么办? 如何定位问题如何解决说一下解决思路和处理方法。
3. 知道字节码吗？字节码都有哪些? Integer x =5, int y =5, 比较x ==y 都经过哪些步骤?

<https://blog.csdn.net/kwame211/article/details/77677662> 查看字节码的各种方式！！很实用

1. 讲讲类加载机制呗？都有哪些类加载器，这些类加载器都加载哪些文件? 手写一下类加载Demo
2. 知道osgi吗?  他是如何实现的???

这个书中有过介绍！！在类加载器章节的，7.4.3破坏双亲委派模型，和9.2.2 OSGI专门介绍。

1. 请问你做过哪些JVM优化? 使用什么方法？到达什么效果???
2. class.forName("java.lang.String")和String. class.getClassLoader(). loadClass("java.lang.String") 什么区别啊??

|  |
| --- |
| <https://blog.csdn.net/lang_man_xing/article/details/7252864>  Class.forName()是调用Class.forName(name, initialize, loader);也就是Class.forName("XXX.XXX"); 等同与Class.forName("XXX.XXX", true, CALLCLASS.class.getClassLoader());  第二次参数表示装载类的时候是否初始化该类（这个方法默认是直接初始化类的）， 即调用类的静态块的语句及初始化静态成员变量。  Class clazz = cl.loadClass("XXX.XXX"); 没有指定是否初始化的选项。只有执行clazz.newInstance()；时才能够初始化类。可以说Class.forName("XXX.XXX", false, cl)执行过程是一致的。只是ClassLoader.loadClass()是更底层的操作。  看一下JDBC驱动的装载：Class.forName("com.mysql.jdbc.Driver");  Connection conn = DriverManager.getConnection("jdbcurl");  当调用Class.forName("com.mysql.jdbc.Driver");是Driver已经被初始化并注册到DriverManager中。可以直接看源码，源码中DriverManager.registerDriver(new Driver());是直接在静态块中。 |

1. 堆和栈在内存中的区别是什么(数据结构方面以及实际实现方面)；

<https://www.cnblogs.com/hemoely/p/5708218.html> 这个是C++进行的分析。

<http://gityuan.com/2016/01/09/java-memory/> 这个对于jvm内存各个区域划分和解释比较直观。

<https://blog.csdn.net/weiyastory/article/details/51513602>

大神1和java中内存分配策略及堆和栈的比较值得一看。按照编译原理的观点,程序运行时的内存分配有三种策略,分别是静态的,栈式的,和堆式的.

JVM中的堆和栈的分析！（值得一看）

从Java的这种分配机制来看,堆栈又可以这样理解:堆栈(Stack)是操作系统在建立某个进程时或者线程(在支持多线程的操作系统中是线程)为这个线 程建立的存储区域，该区域具有先进后出的特性。Java中所有对象的存储空间都是在堆中分配的，但是这个对象的引用却是在堆栈中分配,也 就是说在建立一个对象时从两个地方都分配内存，在堆中分配的内存实际建立这个对象，而在堆栈中分配的内存只是一个指向这个堆对象的指针(引用)而已。

1. 阐述jvm垃圾回收优点和原理，以及劣势，如果直接调用system.gc，jvm会立即回收吗？

## 多线程

1. Synchronized和lock(显式锁)的区别(再加一个volatile)

|  |
| --- |
| 锁不仅能保证临界区中代码能读取到共享变量的最新值，对引用型共享变量，锁还能保证临界区中的代码能读取到该变量所引用对象的字段（实例变量和静态变量）的最新值。  锁能保护共享数据安全，包括保障原子性、可见性、有序性。  锁的几个概念：   1. 可重入性？ 一个线程在持有一个锁的时候，能否再次或多次申请该锁。可重入锁是如何实现的？P85。 2. 锁的争用和调度。公平锁和非公平锁 3. 锁的粒度   锁的开销？申请和释放，锁可能导致上下文切换，锁不正确使用也会导致一些活性故障：锁泄露？死锁、锁死？P86。  内部锁：任何一个对象都有唯一一个与之关联的锁，这种锁被称为监视器！或内部锁。锁句柄就是一个对象的引用，或者是能返回对象的表达式，可以填写为this。作为锁句柄通常用final修饰。通常还会使用private修饰。  同步静态方法相当于以当前类对象为引导锁的同步快。线程对内部锁的申请和释放动作是由jvm负责实施。内部锁使用不会导致锁泄露。原因？P89。  内部锁的调度？入口集，等待线程，jvm对内部锁的调度仅支持非公平调度，所以被唤醒的不一定就能抢到锁，称为持有线程！！  显示锁：Lock接口示例。reentrantLock是默认实现类。显示锁的使用模版伪代码？P90。显示锁的释放必须放到finally中，避免锁泄露，因为显示锁不会像内部锁那样可以由编译器代为规避锁泄露问题。  显示锁调度：支持公平和非公平，公平性往往是以增加线程暂停和唤醒可能性，即增加了上下文切换为代价。使用场景：P91。  显示锁和内部锁的比较：3.4.2章节。  灵活性、内部锁无法跨方法，内部锁优势简单易用，不会导致锁泄露  显示锁容易锁泄露，但是显示锁使用tryLock()可以避免同步该锁的线程直接阻塞而等待。tryLock使用方法模版代码，见P92。  调度方面的比较，公平非公平。问题定位方面，线程转储是否能包含显示锁信息？显示锁的isLocked()，getQueueLength()可用来监控锁信息。  锁选用方面：新项目可考虑显示锁，默认优先选用内部锁，防止锁泄露，在需要显示锁一些特性的时候选之。  改进型锁：读写锁。ReadWriteLock是读写锁的抽象，默认实现ReentrantReadWriteLock。readLock(),writeLock()，读写锁的使用模版代码，见P97。  读写锁使用场景：读操作比写操作频繁多，读操作持有锁时间长。ReentrantReadWriteLock是个可重入锁，支持锁降级。  总结，锁的适用场景！！P99。   1. Check-then-act操作，read-modify-write操作、 2. 多个线程对多个共享数据更新，如果共享数据之间还存在关联关系，为保障操作原子性需要锁。 |

1. Volatile关键字（见3.8章节）volitile的工作原理??

|  |
| --- |
| 一个是保证了可见性（可见性可以防止不可见的问题），另一个是可见性和原子性都能保证    Volatile不能和final共用，不稳定性意味着这种变量的读写都必须从高速缓存或主内存中读取，以读取变量的相对新值。因此，volatile变量不会被编译器分配到寄存器中存储，其读写操作都是内存访问。  被称为轻量级锁，作用：保证可见性和有序性，保证long/double的读写原子性。不同的是，原子性上仅能保证写volatile变量操作的原子性，但没有锁排他性；其次volatile不会引起上下文切换（），因此更像是轻量级锁。  对Long/double原子性的论述？P105。  Count1=count2+1; 不能保证原子操作，volatile Map map=new HashMap();可保证原子操作？P106。  写线程对volatile变量写操作会产生类似于释放锁效果，度线程对volatile变量的读操作会产生类似于获得锁效果。因此volatile保障有序性和可见性作用。  原理P107-109。：对volatile写操作，jvm在操作之前插入一个释放屏障，在操作之后插入一个存储屏障；很好的demo案例：3-6。对volatile读操作，jvm在操作之前插入一个加载屏障，在操作之后插入一个获取屏障。  保证了写线程对volatile变量的写操作和这之前所执行的其他内存操作结果对读线程可见，即保障可见性。这种类似于锁的可见性，不同的是这种可见没有排他性！！  Volatile在保障有序性方面从禁止重排序的角度理解？？P109  Volatile在可见性方面：仅仅是保证读线程能读取到共享变量的而相对新值，对于引用型变量和数组变量，volatile不能保证读线程能读取到相应对象的字段、元素相对新值！！  开销：不会导致上下文切换、因每次都需从高速缓存或主内存读取，无法暂存在寄存器中，从而比普通变量开销大。  重要知识点：volatile的典型应用场景3.8.3章节！！   1. Volatile变量作为状态标志，其实就是利用了其可见性，其他线程在无须加锁的情况下能即使看到更新。 2. 对于多个线程共享的一组可变变量，我们可以将这组变量封装成一个对象，然后在使用volatile修饰这个对象，从而保障了原子性和可见性。 3. 还可以使用volatile和锁联合使用实现简易读写锁。比如计数器。见P112。   典型案例：系统调用下游部件服务，实现对下游服务调用的负载均衡，在不重启应用程序、服务器情况下需要满足要求？？？P112. 尤其是里面EndPoint类里的volatile修饰的online变量，保证更新之后能及时通知。ServiceInvoke里的volatile修饰的LoadBalancer，AbstractLoadBalancer里也有一个volatile修饰的Candidate。同时注意：candidate类使用final进行特殊处理。 |

1. 什么是上下文切换问题？

|  |
| --- |
| 2.7章节  某种程度上，可以看做多个线程共享一个处理器的产物。时间片分配  一个线程被迫或主动暂停其运行，另一个线程可以被操作系统（线程调度器）选中占用处理器开始或继续运行，这样一个被剥夺处理器使用权，另一个被选中开始或继续运行的过程就叫做线程上下文切换。  切出和切入，需要保存和恢复相应线程的进度信息，这个进度信息就是上下文。一般包括通用寄存器和程序计数器内容。是要保存到内存中，切入时再从内存中加载。  一个线程在runnable和其他非runnable状态之间的切换，就是一个上线文切换过程。  2.7.2 上下文切换的分类和具体原因  自发性和非自发性，自发性：执行这些方法，wait, sleep(), join(), yield(), LockSupport.park()，还有线程发起了I/O，或等待其他线程持有的锁，也会导致切换。  非自发性，由于线程调度器原因被迫切出。常见因素：时间片用完或有一个优先级更高的线程要运行。比如Jvm的垃圾回收线程。  上下文切换开销：  直接开销：保存和恢复上线文开销，线程调度器进行线程调度开销  间接开销：一个被切出线程，可能稍后在另一个处理器切入运行，造成处理器高速缓存重新加载的开销；可能导致整个一级高速缓存中的内容被冲刷。 |

1. ThreadLocal原理

|  |
| --- |
| 6.4章节！那一章节讲的很好，尤其是demo非常经典！！  对一个非线程安全对象，每个线程都创建一个该对象实例，各个线程仅访问各自创建的实例。实例被称为线程持有对象，相应线程称为持有线程。既保证了访问线程安全，又避免了锁开销。由于不会被多个线程共享，因此线程特有对象具有固有的线程安全性。  ThreadLocal<T>相当于线程访问其线程特有对象的代理。  线程局部变量通常会声明为某个类的静态变量。  线程特有对象可能导致的问题：  一个线程有很多执行任务，线程特有对象相当于一个线程的多个任务之间的共享对象，有些情况下我们需要确保在执行每个任务前，线程特有对象不能受上一个任务的影响，那就需要将其重置。案例伪代码见P258。  可能导致内存泄漏、伪内存泄漏！！概念？ThreadLocal内部实现原理？？P259。  Java平台中，每个线程内部都维护一个类似HashMap对象，即ThreadLocalMap，每个threadLocalMap内部包含若干个Entry条目，Entry(ThreadLocal k, Object v)，key是一个ThreadLocal实例，value是一个线程特有对象。Entry作用相当于为其所属主线程建立起一个ThreadLocal实例与一个线程特有对象之间的对应关系。 |

1. **指令重排序，内存栅栏等, 内存屏障？？**

|  |
| --- |
| 《java多线程编程实战指南》里描述的比较清晰  **重排序：**首先，从编译上的重排序，其次，从处理器执行指令的时候乱序执行来说；另外一个处理器上执行的多个操作，从其他处理器角度来看其顺序可能与目标代码不一致。重排序的来源：JIT编译器，处理器，存储子系统（写缓冲器store buffer、高速缓存cache）。P57，表2-3重排序类型  编译器出于性能考虑，在其认为不影响程序正确性（单线程）情况下，可能会对源码顺序进行调整，从而造成程序顺序与源码不一致。Java上一般都是JIT编译器可能执行重排序。  处理器执行指令重排序的机制？？P62。顺序读取，乱序执行，顺序提交。还采用了一种猜测执行技术。  存储子系统重排序？2.6.3章节。高速缓存、写缓冲器（store buffer）。即使在处理器严格按照程序顺序执行了两个内存访问操作情况下，在存储子系统作用下，其他处理器对这两个操作的感知顺序可能与程序顺序不一致。这就是存储子系统的重排序，也叫内存重排序。只有四种可能：loadload, storestore, loadstore, storeload。  内存重排序可能导致线程安全问题。具体案例分析，见P65。  **内存栅栏：**这个主要是多线程与计算机内存硬件之间的关联  **内存屏障：**3.6章节。刷新处理器缓存和冲刷处理器缓存。内存屏障是插入到两个指令之间使用的，作用是禁止编译器、处理器重排序从而保障有序性。内部锁的申请和释放对应的字节码指令分别是，monitorenter和monitorexit.  分类：1. 按可见性保障！分为加载屏障和存储屏障  2.按有序性保障划分：获取屏障和释放屏障。锁对有序性的保障，是通过写线程和读线程配对使用释放屏障和加载屏障实现的。 |

1. 线程池构造类的方法的5个参数的具体意义？线程池用过吗？都有什么参数？底层如何实现的??

|  |
| --- |
| 8.5章节，线程池的工作模式P321  几个概念：线程池的大小，线程池大小的3种形态？ThreadPoolExecutor构造器的几个重要参数？  线程池将任务存入工作队列调用的方法是非阻塞方法BlockingQueue.offer(E e), 当工作队列满的时候，线程池会继续创建新的工作者线程，直到达到最大线程池大小。当线程池饱和时，即工作者队列满并且当前线程池大小达到最大线程池大小，再提交任务会被拒绝。拒绝策略？？P323.调用void rejectedExecution(Runnable r, ThreadPoolExecutor executor);  线程池关闭方法: shutdown()/shutdownNow()  有关线程池的类继承关系：  ThreadPoolExecutur-> AbstractExecutorService-> ExecutorService-> Executor  Executor接口使任务的提交与任务执行的具体细节解耦，一定程度上缩小了同步编程与异步编程的代码编写方式差异。  9.2.1实用工具类Executors。1. 方便创建线程工厂，2. 能将runnable实例转换为callable实例。使用Executors能返回几种线程池的方法，及使用场景？见P337。  NewCachedThreadPool()适用于大量耗时较短且提交频率较高的任务，内部使用SynchronousQueue<Runnable>；newFixedThreadPool()线程池空闲线程不会清理，一旦达到核心线程大小，既不增加也不减少，内部使用LinkedBlockingQueue<Runnable>，可以一直往队列添加；newSingleThreadExecutor()返回的是封装了ThreadPoolExecutor的FinalizableDelegatedExecutorService，对外只暴露ExecutorService的方法，内部维护了LinkedBlockingQueue<Runnable> |

1. 单机上一个线程池正在处理服务，如果忽然断电怎么办（正在处理和阻塞队列里的请求怎么处理？）
2. 如何保证共享变量修改时的原子性？

<https://blog.csdn.net/u012454084/article/details/79951961>

volatile保证可见性，但不保证操作原子性（volatile仅能保证变量写操作的原子性）。

加锁、cas

1. sychronized和Lock什么区别? sychronized 什么情况情况是对象锁? 什么时候是全局锁为什么??

已解。

1. ThreadLocal 是什么? 底层如何实现？写一个例子呗?

见上面的分析，包括例子。

1. 自己实现线程安全类
2. CAS, 原子变量类，CAS知道吗？如何实现的?

|  |
| --- |
| 3.10章节：CAS与原子变量  CAS是对一种处理器指令的称呼，也是一种Java线程同步机制。不少多线程相关的类库实现都借助CAS。锁的开销是最大的,volatile无法保障count++操作的原子性。CAS能将read-modify-write和check-and-act之类的操作转换为原子操作。P127对CAS的论述比较精彩。CAS是一个原子的if-then-act的操作！！其更新机制的原子性是由处理器来保障。经典案例，P128。  注意：CAS仅保障共享变量更新操作的原子性，但是不保障可见性  与CAS有关的类，都是在java.util.concurrent.atomic包下  原子变量类（Atomics）是基于CAS实现的，能保障对共享变量进行read-modify-write更新操作的原子性和可见性的工具类。原子变量类内部实现通常借助一个volatile变量并保障对该变量的更新操作原子性，因此被看做是增强型的volatile变量。  原子类一些常用方法？  Volatile修饰的数组，无法保障其元素的读写操作可见性和原子性，因此专门引入了AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray。  注意：CAS原子操作不能规避ABA问题，如果想要规避，需要为共享变量的更新引入一个修订号（比如时间戳），AtomicStampedReference类就是基于这种思想!!见P135。 |

1. Object.wait()/notify()的**过早唤醒**问题？

|  |
| --- |
| 5.1章节，应用场景：多线程编程中，保护条件未满足的时候暂停当前线程，稍后其他线程可能更新了保护条件涉及的共享变量而使其成立，这时候之前的线程被唤醒然后继续执行。  有关等待和唤醒的概念：P195。  等待线程和通知线程，以及Object.wait()的模版伪代码，执行的条件，受保护方法和其要素，wait()执行过程原理分析，何时释放内部锁？wait()何时有返回？：P196  等待线程被唤醒，继续运行到再次持有相应对象内部锁的这段时间，可能会遇到保护条件再次不成立的情况：见P197，这也是为什么wait()要放在循环里的原因！！  目标线程对保护条件的判断和对目标动作的执行必须是个原子操作的原因，几大注意事项（while循环、内部锁临界区内、其他锁）：P197  Object.notify()的模版伪代码，通知方法的两个要素，执行通知方法的条件，何时释放内部锁？P197    Object.wait()/notify()的内部实现原理（讲的比较清楚！）：P198（纸质P182）  **Wait/notify的各种问题（5.1.2章节）：**  1. 过早唤醒问题：P204，大白话,就是线程2的保护条件还没满足,就被唤醒了,但是没用,还是无法跳出循环；  2.信号丢失，简单说就是,通知线程过早执行了,导致后面等待线程等待之后,没人通知了;或者通知线程使用了notify()只通知了一个等待,其他等待没人通知!；  3. 导致过多的上下文切换问题！造成的原因比较多？以及减少或避免的方法？具体见P206。  Notify()/notifyAll()的选用策略和考虑因素：P207。（纸质P191）  Wait()实现的是无限等待，wait(long time)是等待指定时间，但如果超时了，虚拟机会自动唤醒，既无返回值也不抛出异常！解决区分是否是等待超时的经典案例：P203。  通过报警器案例个人理解: wait()的意义在于, 先判断是否符合条件(保护条件), 不符合条件在循环里执行等待, 符合条件wait()执行结束并跳出循环(或者直接满足条件, 不进入循环)  相应的, notify()意义在于, 更改共享变量至满足条件, 然后通知等待线程, 继续执行!! |

1. Thread.join()的含义和源码解析：（纸质P191）

P208。底层就是使用wait()和isAlive()联合实现的。内部源码就是，检测目标线程是否isAlive()（这个其实就是保护条件），如果目标线程没结束，就一直等待wait()。Java虚拟机会在目标线程run方法执行结束后，执行该线程对象的notifyAll()来通知所有等待线程。

1. Condition如何防止过早唤醒的问题？

|  |
| --- |
| 有关java条件变量Condition的讲解：P208，《java多线程实战》5.2章节，示例代码P209  Condition如何解决过早唤醒问题？ 见p210  Condition还解决了Object.wait(long time)存在的问题，即无法区分是由于通知还是等待超时了？P211（书P195），即使用了Condition.awaitUtil(Date deadline)，通过返回值来判断！  但要注意一个问题：执行了condition.await()/awaitUtil()方法的等待线程，在被唤醒之后，还是需要再次申请到显示锁才有返回值！见P212的描述！！还有代码示例5-3  每个Condition实例内部都维护了一个用于存储等待线程的队列。使用代码模版，见P193（书）。 |

1. Static和final在多线程中的特殊作用？(java多线程实战指南中的3.11.1介绍)

|  |
| --- |
| 对于引用型静态变量，static还能保障一个线程读取到该变量初始值时，这个值所指向的对象已经初始化完毕！！  Static只能保障线程初次读取到相应字段的初始值，而不是相对新值！！ |
| Final修饰的字段可以防止其被重排序，因此可以达到上述的效果。这里注意，final只能保障有序性，即保障一个对象对外可见的时候，该对象final字段必然是初始化完毕的，但并不保障对象引用本身对外的可见性。  本书第三章的总结概括的很好 |

1. 线程同步有几种实现方式？优缺点？

也就是第三章讲的java线程同步机制。包括锁，volatile，CAS，final，static

## Java中数据结构

1. 开发中JAVA用了比较多的数据结构有哪些？（其实就是Map, set, List）

|  |
| --- |
| <http://blog.csdn.net/wufaliang003/article/details/55261097> 这个里面的类图丰富些  Map：sortedMap, abstractMap, TreeMap, HashMap, LinkedHashMap, HashTable  LinkedHashMap，里面维护的是双向链表，在遍历的时候遍历的顺序就是插入时的顺序！  注意：TreeMap的排序是按照红黑二叉树的排序规则排列的  <http://blog.csdn.net/kiss_the_sun/article/details/7848920> (有关linkedHashMap使用)  <https://www.cnblogs.com/web424/p/6824298.html> 这个来看collection  sortedMap，sortedSet接口里面都维护了comparator，可以帮助子类实现排序  Collection:  分Set和List两大子接口：  Set: SortedSet, AbstractSet, HashSet, TreeSet, LinkedHashSet  <http://blog.csdn.net/micheal100/article/details/39287919> （hashSet和TreeSet的区别）  HashSet是通过HashMap实现的, TreeSet（里面多了一个排序）是通过TreeMap实现的,只不过Set用的只是Map的key，value是一个静态的new object()，也为了节省空间！  List: AbstractList, ArrayList, LinkedList, Vector  ArrayList内部维护的是数组，自动扩容一半，增删需要copy数组其余元素，查找快，可能会有剩余空间；  LinkedList内部维护链表，并且是双向链表，有效元素空间复杂度要高点，查找会比较中间位置，采取从前到后还是从后到前；  Vector内部维护数组，自动扩容是一倍，也可指定扩容量，同步方法，线程安全！  <http://blog.csdn.net/zhangyunpengchang/article/details/51144656> （三种list的区别） |

1. HashMap的并发问题
2. 谈谈对HashMap的理解，底层基本实现，HashMap怎么解决碰撞问题，是否线程安全，hashTable是线程安全的，有没有线程安全的Map，接下类问concurrent包

|  |
| --- |
| 答：<http://blog.csdn.net/skiof007/article/details/53102221>  (碰撞处理, 底层实现)  concurrentHashMap由多个segment组成，每个segment中维护了一个HashEntry[], 只是在操作的时候都会先加上显式锁, 操作完成之后释放。 |

1. treeMap
2. 二叉树和红黑树等
3. hashmap 和hastable 底层实现什么区别? hashtable和ConcurrentHashMap呢

|  |
| --- |
| Hashmap底层Entry<>数组+Entry<>链表，其实不同种类的map，最根本的区别就是Entry<>内部类的定义上。HashTable就是HashMap的暴力加锁，实现线程安全！concurrentMap底层维护的segments数组，每个segment里面才是真正的Entry<>，它里面使用显式锁来实现线程安全，多个segments可以实现并发操作，提高效率 |

1. hashmap和treemap什么区别？底层数据结构是什么？

|  |
| --- |
| 红黑树的定义和插入删除规则：  <http://blog.csdn.net/sun_tttt/article/details/65445754>  treemap的底层数据结构是红黑二叉树（本身又是二叉查找树），treemap插入时刚开始是按照查找树来寻找位置，之后调整红黑树，删除也类似。  TreeMap实现SortMap接口，能够把它保存的记录根据键排序, 默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。  注意与LinkedHashMap的区别，它里面的排序是按照插入顺序来排的，同时还可以设置是否按访问顺序来排序（将访问的元素排到末尾） |

1. arrayList和linkedList的区别
2. 了解LinkedHashMap的应用吗？
3. ConCurrentMap的机制，附带也说说HashTable原理

<https://www.cnblogs.com/chengxiao/p/6842045.html>

1. 序列化有哪两种方式？transient关键字的含义？

|  |
| --- |
| <https://www.cnblogs.com/lanxuezaipiao/p/3369962.html>  实现Externalizable接口（这个接口内部其实继承了Serializable，内部可以指定需要序列化的变量），或者Serializable接口。 |

1. 介绍一下集合框架?

## 通信协议

1. 说一下tcp/ip四层？
2. HTTP请求报文和响应报文格式

<http://www.cnblogs.com/biyeymyhjob/archive/2012/07/28/2612910.html>

<http://blog.csdn.net/a19881029/article/details/14002273>

1. http的工作流程？http1.0, 1.1, 2.0有哪些区别？

|  |
| --- |
| <http://blog.csdn.net/hguisu/article/details/8680808>（包含http工作过程，https工作原理，讲的挺好）  <http://blog.csdn.net/elifefly/article/details/3964766/>（1.0与1.1区别）  HTTP 1.1支持持久连接，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟。HTTP 1.1还允许客户端不用等待上一次请求结果返回，就可以发出下一次请求，但服务器端必须按照接收到客户端请求的先后顺序依次回送响应结果，在HTTP 1.1中增加Host请求头字段后，WEB浏览器可以使用主机头名来明确表示要访问服务器上的哪个WEB站点，这才实现了在一台WEB服务器上可以在同一个IP地址和端口号上使用不同的主机名来创建多个虚拟WEB站点。  <http://blog.csdn.net/sjhuangx/article/details/52299726> |

1. Tcp的三次握手，四层分手的工作流程？画一下流程图？为什么不是四次五次或两次？

|  |
| --- |
| <https://www.cnblogs.com/shihaochangeworld/p/5770294.html>  （牛逼，尤其是为什么不是的解释） |

1. 画一下https的工作流程？具体如何实现？如何防止被抓包？

|  |
| --- |
| http://img.my.csdn.net/uploads/201303/19/1363688966_6720.jpg  <http://blog.csdn.net/buster2014/article/details/47185459>  tomcat修改配置，web.xml里添加一个配置  <https://www.cnblogs.com/jay54520/p/6181604.html?utm_source=itdadao&utm_medium=referral>  抓包问题 |

1. http协议，get，post基本区别，tcp/ip协议，三次握手，窗口滑动机制
2. Tcp/IP协议， Http、TCP/IP协议与Socket之间的区别？

|  |
| --- |
| <http://blog.csdn.net/done58/article/details/50996680> （里面还包含三次握手）这篇文章内容比较全面!!! 还包括长连接、短链接，  Socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口（API），通过Socket，我们才能使用TCP/IP协议。  ISO制定的OSI参考模型过于庞大、复杂招致了许多批评。与此对照，由技术人员自己开发的TCP/IP协议栈获得了更为广泛的应用。如图所示，是TCP/IP参考模型和OSI参考模型的对比示意图。  iso 7 4  建立连接是三次握手， 断开连接是四次握手  TPC/IP协议是传输层协议，主要解决数据 如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。  三次握手建立连接：  第一次握手：建立连接时，客户端发送syn包（syn=j）到服务器，并进入SYN\_SENT状态，等待服务器确认；SYN：同步序列编号（Synchronize Sequence Numbers）。  第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入SYN\_RECV状态；  第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。  四次握手断开连接：  对于一个已经建立的连接，TCP使用改进的三次握手来释放连接（使用一个带有FIN附加标记的报文段）。TCP关闭连接的步骤如下：  第一步，当主机A的应用程序通知TCP数据已经发送完毕时，TCP向主机B发送一个带有FIN附加标记的报文段（FIN表示英文finish）。  第二步，主机B收到这个FIN报文段之后，并不立即用FIN报文段回复主机A，而是先向主机A发送一个确认序号ACK，同时通知自己相应的应用程序：对方要求关闭连接（先发送ACK的目的是为了防止在这段时间内，对方重传FIN报文段）。  第三步，主机B的应用程序告诉TCP：我要彻底的关闭连接，TCP向主机A送一个FIN报文段。  第四步，主机A收到这个FIN报文段后，向主机B发送一个ACK表示连接彻底释放。 |

1. 长连接和短链接的区别？

<http://blog.csdn.net/jasonjwl/article/details/52085264>

## 算法

1. 有一个文件，有45亿个阿拉伯数字，如何进行去重？如何找出最大的那个数？
2. 快速排序算法
3. 广度优先搜索（队列实现）

## 数据结构

1. Json的数据格式
2. 继承的好处与坏处

## 数据库

1. 使用mysql索引有哪些原则？索引是什么数据结构？B+tree和B tree什么区别？

|  |
| --- |
| <http://blog.csdn.net/u013412790/article/details/51612304> 索引的原则  **mysql索引最左匹配原则理解**：mysql创建复合索引的规则是首先会对复合索引的最左边的，也就是第一个name字段的数据进行排序，在第一个字段的排序基础上，然后再对后面第二个的cid字段进行排序。其实就相当于实现了类似 order by name cid这样一种排序规则。所以：第一个name字段是绝对有序的，而第二字段就是无序的了。所以通常情况下，直接使用第二个cid字段进行条件判断是用不到索引的，当然，可能会出现上面的使用index类型的索引。这就是所谓的mysql为什么要强调最左前缀原则的原因（要求复合索引要想使用第二个索引，必须先使用第一个索引的原因）。链接：<https://www.zhihu.com/question/36996520/answer/93256153>  **Mysql索引背后的数据结构及算法原理**（经典）：  <http://blog.codinglabs.org/articles/theory-of-mysql-index.html> 索引（Index）是帮助MySQL高效获取数据的数据结构，索引的本质：索引是数据结构。 |

1. 对比一下使用B+tree和Hash作为索引算法的优劣？
2. 数据库索引，组合索引，索引失效问题。（58**实战**）
3. Mysql有哪些存储引擎？有啥区别？要详细！！
4. 设计高并发系统数据库层面怎么设计？数据库锁有哪些类型？如何实现？

|  |
| --- |
| 共享锁（s）、排它锁（x）、更新锁（u）、  相对其他数据库而言，MySQL的锁机制比较简单，其最 显著的特点是不同的存储引擎支持不同的锁机制。  表级锁、行级锁、页面锁（mysql这三种级别锁的特性）  <http://blog.csdn.net/itleochen/article/details/13169665> （mysql数据库锁的介绍）  <http://www.jb51.net/article/125918.htm> （分布式锁实现三种方式）  <http://blog.csdn.net/tianjiabin123/article/details/72625156> （基于数据库的分布式锁实现， 隔离级别介绍） |

1. 数据库事务有哪些？数据库隔离级别？

<https://www.cnblogs.com/fjdingsd/p/5273008.html> （数据库事务的四大特性及事务隔离级别）。结合数据库锁的级别，就比较容易理解数据库隔离级别的概念，以及几种比较典型的事务问题！例如：可重复读应该是通过记录锁实现，而串行化是通过锁表实现。

1. 开发中用了哪些数据库？mysql存储引擎有哪些，悲观锁和乐观锁的问题使用场景，分布式集群实现的原理？

|  |
| --- |
| <http://blog.csdn.net/qq_27028821/article/details/52267991>  （MySQL中四种常用存储引擎的介绍）  <https://www.cnblogs.com/dangjunhui/p/5477622.html>  MyISAM，InnoDB，MEMORY，MERGE  悲观锁和乐观锁：<http://blog.csdn.net/claram/article/details/53959367>  <http://www.hollischuang.com/archives/934>  <https://www.cnblogs.com/qjjazry/p/6581568.html>  悲观锁和乐观锁,以及CAS实现, 讲的比较好 |

1. 数据库垂直和水平拆分
2. Mysql中锁详解：<https://blog.csdn.net/tanga842428/article/details/52748531#t0> （比较清晰，并且易懂！！）

|  |
| --- |
| MySQL的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write Lock）。（MyISAM存储引擎的写阻塞读例子）  MyISAM（引擎）在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT等）前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用LOCK TABLE命令给MyISAM表显式加锁。(如何加表锁)  （MyISAM存储引擎的读阻塞写例子）  注意，当使用LOCK TABLES时，不仅需要一次锁定用到的所有表，而且，同一个表在SQL语句中出现多少次，就要通过与SQL语句中相同的别名锁定多少次，否则也会出错！举例说明如下。  MyISAM存储引擎有一个系统变量concurrent\_insert，专门用以控制其并发插入的行为，其值分别可以为0、1或2。  当concurrent\_insert设置为0时，不允许并发插入。  当concurrent\_insert设置为1时，如果MyISAM表中没有空洞（即表的中间没有被删除的行），MyISAM允许在一个进程读表的同时，另一个进程从表尾插入记录。这也是MySQL的默认设置。  当concurrent\_insert设置为2时，无论MyISAM表中有没有空洞，都允许在表尾并发插入记录。  可以利用MyISAM存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。例如，将concurrent\_insert系统变量设为2，总是允许并发插入；同时，通过定期在系统空闲时段执行 OPTIMIZE TABLE语句来整理空间碎片，收回因删除记录而产生的中间空洞。  前面讲过，MyISAM存储引擎的读锁和写锁是互斥的，读写操作是串行的。那么，一个进程请求某个 MyISAM表的读锁，同时另一个进程也请求同一表的写锁，MySQL如何处理呢？答案是写进程先获得锁。但是可以通过一些设置来调节，如low-priority-updates。  InnoDB锁问题  InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁。行级锁与表级锁本来就有许多不同之处，另外，事务的引入也带来了一些新问题。  并发事务处理带来的问题？更新丢失、脏读、不可重复读、幻读，这些问题的发生都是在同一个事务中。  数据库实现事务隔离的方式，基本上可分为以下两种：  一种是在读取数据前，对其加锁，阻止其他事务对数据进行修改。  另一种是不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot)，并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好像是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据多版本并发控制（MultiVersion Concurrency Control，简称MVCC或MCC），也经常称为多版本数据库。  对经典的4种隔离级别的说明。（4种隔离级别比较）  （InnoDB的行锁模式及加锁方法），InnoDB实现了以下两种类型的行锁。意向锁是InnoDB自动加的，不需用户干预。对于UPDATE、DELETE和INSERT语句，InnoDB会自动给涉及数据集加排他锁（X)；对于普通SELECT语句，InnoDB不会加任何锁；事务可以通过以下语句显示给记录集加共享锁或排他锁。  用SELECT ... IN SHARE MODE获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行UPDATE或者DELETE操作。对于锁定行记录后需要进行更新操作的应用，应该使用SELECT... FOR UPDATE方式获得排他锁。（InnoDB存储引擎的共享锁例子）（InnoDB存储引擎的排他锁例子）这些例子很好。  InnoDB行锁是通过给索引上的索引项加锁来实现的，这一点MySQL与Oracle不同，后者是通过在数据块中对相应数据行加锁来实现的。InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！（InnoDB存储引擎的表在不使用索引时使用表锁例子）  由于MySQL的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。应用设计的时候要注意这一点。（InnoDB存储引擎使用相同索引键的阻塞例子）  当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，另外，不论是使用主键索引、唯一索引或普通索引，InnoDB都会使用行锁来对数据加锁。（InnoDB存储引擎的表使用不同索引的阻塞例子）  即便在条件中使用了索引字段，但是否使用索引来检索数据是由MySQL通过判断不同执行计划的代价来决定的，如果MySQL认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引，这种情况下InnoDB将使用表锁，而不是行锁。因此，在分析锁冲突时，别忘了检查SQL的执行计划，以确认是否真正使用了索引。（explain select \* from tab\_with\_index where name = 1）  （间隙锁）当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）。（InnoDB使用间隙锁的目的）（ InnoDB存储引擎的间隙锁阻塞例子）（CTAS操作给原表加锁例子）（CTAS操作不给原表加锁带来的安全问题例子）  （InnoDB存储引擎中不同SQL在不同隔离级别下锁比较）详细  实际上，通过优化事务逻辑，大部分应用使用Read Commited隔离级别就足够了。对于一些确实需要更高隔离级别的事务，可以通过在程序中执行SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ或SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE动态改变隔离级别的方式满足需求。  （什么时候使用表锁）  在用 LOCK TABLES对InnoDB表加锁时要注意，要将AUTOCOMMIT设为0，否则MySQL不会给表加锁；事务结束前，不要用UNLOCK TABLES释放表锁，因为UNLOCK TABLES会隐含地提交事务；COMMIT或ROLLBACK并不能释放用LOCK TABLES加的表级锁，必须用UNLOCK TABLES释放表锁。正确的方式见如下语句：  （InnoDB存储引擎中的死锁例子）  （下面就通过实例来介绍几种避免死锁的常用方法。）这个比较实用！！ |

**注：通过（）里面的内容可以直接在文章中搜索定位！**

### 分库分表

1. 如何设计可以动态扩容缩容的分库分表方案
2. 用过哪些分库分表的中间件？有啥优点和缺点？讲一下你了解的分库分表中间件的底层实现原理？
3. 现在有一个未分库分表的系统，以后系统需要分库分表，如何设计？让未分库分表系统动态切换到分库分表的系统？
4. 为什么要分库分表

|  |
| --- |
| 数据库分表可以解决单表海量数据的查询性能问题，分库可以解决单台数据库的并发访问压力问题。有时候，我们需要同时考虑这两个问题，因此，我们既需要对单表进行分表操作，还需要进行分库操作，以便同时扩展系统的并发处理能力和提升单表的查询性能，就是我们使用到的分库分表。  分库分表的策略相对于前边两种复杂一些，一种常见的路由策略如下：  １、中间变量　＝ user\_id%（库数量\*每个库的表数量）;  ２、库序号　＝　取整（中间变量／每个库的表数量）;  ３、表序号　＝　中间变量％每个库的表数量;  <http://blog.csdn.net/xlgen157387/article/details/53976153> （数据库分库分表具体实现方案）  mysql只要是通过二进制日志来复制数据。（主从复制、读写分离）  <https://www.cnblogs.com/jshen/p/7682502.html> （分库分表的基本思想、主键问题、分页查询问题、分布式事务问题）  Sharding的基本思想就要把一个[数据库](http://lib.csdn.net/base/mysql)切分成多个部分放到不同的数据库(server)上，从而缓解单一数据库的性能问题。不太严格的讲，对于海量数据的数据库，如果是因为表多而数据多，这时候适合使用垂直切分，即把关系紧密（比如同一模块）的表切分出来放在一个server上。如果表并不多，但每张表的数据非常多，这时候适合水平切分，即把表的数据按某种规则（比如按ID散列）切分到多个数据库(server)上。 |

1. 如何解决分库分表主键问题？有什么方案？

|  |
| --- |
| <http://www.infoq.com/cn/articles/key-steps-and-likely-problems-of-split-table>  （分库分表的几种常见形式以及可能遇到的难InfoQ的文章）  [http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency#](http://www.infoq.com/cn/articles/solution-of-distributed-system-transaction-consistency)  （分布式事务方面的问题InfoQ文章，回滚接口、事务消息、非事务消息介绍） |

## Mybatis 研究

|  |
| --- |
| 按照《深入分析javaweb技术内幕》查看源码过程。还包括spring，servlet, tomcat  Mybatis的工作原理（初始化过程，源码比较简单）：  <http://blog.csdn.net/pfnie/article/details/51491934> （时序图画的不错）  <https://www.cnblogs.com/luoxn28/p/6417892.html> （貌似更叼、更全， 初始化、参数封装、结果集封装的源码过程）  Mybatis配置方式:1. 最基础的xml配置<configuration>形式; 2. 与spring集成bean的配置方式; 3. Java编码配置方式  Mybatis动态代理原理分析: <https://www.cnblogs.com/hayasi/p/6361067.html> 从getMapper()开始，获取要执行接口方法的代理类，然后使用代理类的invoke()方法，最后回到了sqlSession.select()等方法上去，然后就跟上面的源码分析连接起来！  过程：初始化configuration——》getMapper()得到动态代理类——》执行select()方法，调用代理类的invoke()——》回到sqlSession的执行方法  Mybatis插件的拦截功能实现，也是通过jdk的动态代理实现，Mybatis源码分析项目里的ExamplePlugin是demo案例，主要用到Plugin类（参考8.1拦截器接口介绍）  Mybatis与Spring整合原理：<http://blog.csdn.net/fighterandknight/article/details/51448161> （从sqlSessionFactoryBean的初始化，MapperFactoryBean的创建，如何通过包扫描获取到所有的MapperFactoryBean）  Mybatis在与spring整合过程中，有关使用包扫描来加载所有的mapper源码分析：  <http://blog.csdn.net/u012291108/article/details/51943688>  <http://blog.csdn.net/u012291108/article/details/51943688> 这个对于包扫描加载的解释更详细 |

1. 一级缓存和二级缓存的区别？配置？
2. SpringMVC和Mybatis的工作原理，有没有看过底层源码？
3. Mybatis如何分页，如何设置缓存，mysql分页

## Tomcat 研究

1. tomcat如何分发请求，如何处理多用户的同时请求？多级容器如何协调工作？
2. standardService中修改container如何通知到connector的？
3. 描述一下Servlet的生命周期，重点说清楚什么阶段调用什么方法？

|  |
| --- |
| Init(), service(), destroy()  <http://blog.csdn.net/danielzhou888/article/details/70835418>  <http://blog.csdn.net/Jassy1994/article/details/54025471>  <http://blog.csdn.net/dhfzhishi/article/details/77781336>  servlet实质就是按照servlet规范编写的java类, 可以处理web应用的相关请求, 是一个标准.servlet对象在容器启动时初始化，第一次请求的时候容器将其实例化。如果存在多个请求，servlet不会再实例化，每一个请求分配一个线程来处理。 |

1. Servlet与jsp的区别

|  |
| --- |
| Servlet是使用java servlet接口运行在web应用服务器上的java程序，  <http://blog.csdn.net/forward__/article/details/57079115> |

1. Session与cookie的区别与联系

|  |
| --- |
| <http://blog.csdn.net/duan1078774504/article/details/51912868>  不要把敏感的信息放到cookie里，session不能共享的问题需要解决，cookie还受到存储大小的限制。Cookie的典型应用：1.判断用户是否登录过网站，2. 购物车的设计(一般是未登录状态下)，用户可能在一段时间内在同一家网站的不同页面选择不同的商品，可以将这些信息都写入cookie，在最后付款时从cookie中提取这些信息。  Cookie的存活时间如果设置很长，会以文件方式在硬盘上存储。  Cookie在分布式项目中，经常会在sso中使用，存入token信息，案例见taotao的doSetCookie()  分布式session框架，cookie解决跨域共享，引出ajax跨域请求，多终端共享cookie问题（手机扫码pc端） |

|  |
| --- |
| 1. Tomcat的类加载过程：<http://blog.csdn.net/beliefer/article/details/50995516>  2. server.xml的加载和解析过程：<http://blog.csdn.net/beliefer/article/details/51035923> （可以对照着server.xml配置文件的标签进行理解）  3. Tomcat生命周期管理，重点容器的初始化、启动，由父容器调用子容器，通过LifecycleBase来统一管理生命周期：<http://blog.csdn.net/beliefer/article/details/51473807>  4. Tomcat启动与停止服务原理: <http://blog.csdn.net/beliefer/article/details/51585006>  启动就是通过脚本调用Bootstrap.java的main方法，main方法里面通过判断脚本命令，来执行响应的操作。注意：使用脚本启动使用的是bootstrap这种引导程序启动的；但是如果要通过代码实现，一般是从Tomcat类里的start()方法可以直接启动。（这种时序图可以参考《深入分析javaWeb》）  5.对session的管理分析：<http://blog.csdn.net/beliefer/article/details/52450268> （上）  主要研究standardManager初始化过程是怎么回事，如何加载持久化的session信息（大概的流程描述一下）  <https://blog.csdn.net/beliefer/article/details/52451061> （下）介绍了Filter的职责链，Tomcat接收到的请求会经过Filter职责链，最后交给具体的Servlet处理   1. Tomcat请求处理的原理：   <https://blog.csdn.net/beliefer/article/details/51645799> （上），首先是如何初始化构造connector，设置请求协议解析，ProtocolHandler是如何创建对象的？（博客里不是很清晰，应该还是通过Digester在解析到标签的时候，会执行startElement方法，startElement中会调用Rule的begin(String namespace, String name, Attributes attributes)方法，Connector对应的Rule就包括了ConnectorCreateRule。）还有将ProtocolHandler、MapperListener注册到JMX？  由connector的初始化（入口），调用子容器ProtocolHandler（以Http11Protocol为代表）的init方法，这里头完成了一些JIoEndpoint、配置ServerSocketFactory操作；  然后是启动connector, ProtocolHandler（创建线程池和任务队列，创建接收请求线程）, 对请求的处理，初始化mapperListener  <https://blog.csdn.net/beliefer/article/details/51828607> （中）  http://img.blog.csdn.net/20160706222248126?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  Acceptor和SocketProcessor线程数设置的讨论，有关性能调优。  本篇文章的核心就是讲大致的请求处理调用过程（注意：从后往前看）：  postParseSuccess = postParseRequest(req, request, res, response); （处理前解析）  connector.getService().getContainer().getPipeline().getFirst().invoke(request, response);（真正处理）  《—— CoyoteAdapter.service(Request, Response) (org.apache.catalina.connector)  《—— AbstractHttp11Processor.process(SocketWrapper<S>) (org.apache.coyote.http11)  《—— AbstractConnectionHandler in AbstractProtocol.process(SocketWrapper<S>, SocketStatus) 《—— SocketProcessor的run()《—— Acceptor线程的run()里面的processSocket(socket)  <https://blog.csdn.net/beliefer/article/details/51894747> （下）  主要将connector.getService().getContainer().getPipeline().getFirst().invoke(request, response); 如何获取每个容器的vavle，怎么向下传递，这里的getContainer()获取的是StandardService中的StandardEngine容器！StandardEngine继承自ContainerBase，所以这里的getPipeline方法实际是ContainerBase实现的  主要讲了StandardWrapperValve的invoke()方法实现，过滤链执行原理。整个调用过程  HttpServlet.doGet(HttpServletRequest req, HttpServletResponse resp) (javax.servlet.http;)  HttpServlet.service(ServletRequest, ServletResponse) (javax.servlet.http)  HttpServlet.service(ServletRequest, ServletResponse) (javax.servlet.http)  ApplicationFilterChain.internalDoFilter(ServletRequest, ServletResponse) (2 usages) (org.apache.catalina.core)  ApplicationFilterChain.doFilter(ServletRequest, ServletResponse) (org.apache.catalina.core)  🡨StandardWrapperValve.invoke(Request, Response)(2 usages) (org.apache.catalina.core)  🡨StandardEngineValve.invoke(Request, Response)(2 usages) (org.apache.catalina.core)  ContainerBase. getPipeline()创建了new StandardPipeline(this)，随后调用了getFirst()用来获取管道中的第一个Valve，其实就是使用里面basic，basic的类型是StandardEngineValve，再结合下面的valve流程图，最后调用的valve出口就是StandardWrapperValve。然后方法的调用就是上面的调用流程。这个调用过程，可以参照下面的流程  http://img.blog.csdn.net/20160727093122710?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center  Tomcat主要流程概括总结：<https://www.2cto.com/kf/201606/518709.html>  <http://donald-draper.iteye.com/blog/2327119> 重点如下：  Engine的初始化与启动实际上是委托给ContainerBase，而ContainerBase初始化与启动，主要做的工作是，创建后台子容器线程执行器，执行子容器的启动任务，启动context更新守候线程。ContainerBase拥有一个StandardPipeline，StandardPipeline是处理HTTP请求的管道，在管道内部有很多Valve（ Valve, Filter, or Servlet），这些Value是一个只有后继的链，当请求进来的时候，调用invoke(Request request, Response response)处理请求。  Tomcat Server处理一个http请求的过程  假设来自客户的请求为：  http://localhost:8080/wsota/wsota\_index.jsp  1) 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得  2) Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应  3) Engine获得请求localhost/wsota/wsota\_index.jsp，匹配它所拥有的所有虚拟主机Host  4) Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）  5) localhost Host获得请求/wsota/wsota\_index.jsp，匹配它所拥有的所有Context  6) Host匹配到路径为/wsota的Context（如果匹配不到就把该请求交给路径名为”"的Context去处理）  7) path=”/wsota”的Context获得请求/wsota\_index.jsp，在它的mapping table中寻找对应的servlet  8) Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类  9) 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法  10)Context把执行完了之后的HttpServletResponse对象返回给Host  11)Host把HttpServletResponse对象返回给Engine  12)Engine把HttpServletResponse对象返回给Connector  13)Connector把HttpServletResponse对象返回给客户browser  Tomcat解析web.xml的从调用开始到解析配置到context容器完成的过程（手动过程）：  ContextConfig.webConfig()(2 usages) (org.apache.catalina.startup)  ContextConfig.configureStart() (org.apache.catalina.startup)  ContextConfig.lifecycleEvent(LifecycleEvent) (org.apache.catalina.startup)  这里主要也是采用了监听器模式触发的，contextConfig作为standardContext的监听者，是在tomcat.addWebapp()的时候，绑定到新建的standardContext容器中的！  首先是standardContext里initInternal()时，发送通知，随后contextConfig调用init()完成初始化；然后standardContext调用startInternal()时，发送启动通知，contextConfig调用configureStart(); 注意：contextConfig作为一个监听者，所有对通知事件做出的响应入口都是lifecycleEvent(LifecycleEvent event)方法！  而使用脚本启动，主要还是依靠从父容器启动到子容器启动过程的传递，一直传递到standardService，然后是standardEngine，standardHost，standardHost的startInternal()里面会调用containerBase里的相应方法启动子容器，从而造成了子容器standardContext的init()。  Tomcat中很多容器都继承了containerBase抽象类，这个是经典的模版设计模式，在这个模版中实现了很多通用的方法比如initInternal(),startInternal(),而这些方法中，又关联到各自的子容器和监听者，真是牛逼！  Tomcat启动过程中，解析servlet的过程是在configureStart()方法，初始化servlet过程：  servlet.init(facade);  StandardWrapper.initServlet(Servlet) (org.apache.catalina.core)  StandardWrapper.load() (org.apache.catalina.core)  StandardContext.loadOnStartup(Container[]) (org.apache.catalina.core)  StandardContext.startInternal() (org.apache.catalina.core)  Tomcat中servletConfig其实是standardWrapperFacade, servletContext实际对象也是ApplicationContextFacade，这些都是使用了门面设计模式！    servlet中提供了很多listener，用于实现观察者设计模式！比如spring的contextLoaderListener就实现了一个servletContextListener，当容器加载时就启动spring容器！  Tomcat的FilterConfig和FilterChain说明： |

## Spring 研究

### Spring

1. spring都有哪些机制啊？AOP底层如何实现的啊？IOC呢??
2. Spring事务实现原理
3. cglib知道吗?他和jdk动态代理什么区别？手写一个jdk动态代理呗?

|  |
| --- |
| <https://www.cnblogs.com/ygj0930/p/6542259.html> （java动态代理jdk实现和cglib实现，以及静态代理）   静态代理的缺点很明显：一个代理类只能对一个业务接口的实现类进行包装，如果有多个业务接口的话就要定义很多实现类和代理类才行。而且，如果代理类对业务方法的预处理、调用后操作都是一样的（比如：调用前输出提示、调用后自动关闭连接），则多个代理类就会有很多重复代码。这时我们可以定义这样一个代理类，它能代理所有实现类的方法调用：根据传进来的业务实现类和方法名进行具体调用。——那就是动态代理。  **JDK动态代理所用到的代理类在程序调用到代理类对象时才由JVM真正创建，JVM根据传进来的 业务实现类对象 以及 方法名 ，动态地创建了一个代理类的class文件并被字节码引擎执行，然后通过该代理类对象进行方法调用。**  **里面的案例比较经典，**  **cglib是针对类来实现代理的，原理是对指定的业务类生成一个子类，并覆盖其中业务方法实现代理。因为采用的是继承，所以不能对final修饰的类进行代理。** |

1. 执行某操作，前50次成功，第51次失败，a全部回滚，b前50次提交，第51次抛出异常，ab场景如何设置spring（传播特性）

|  |
| --- |
| 1. Spring 容器的加载过程？   这个就是从org.springframework.context.support.AbstractApplicationContext.refresh()方法开始分析，总共有几大加载动作！   1. Bean的解析和登记时序问题（对照 图13-11）？   XmlBeanDefinitionReader.doLoadDocument(InputSource, Resource) (org.springframework.beans.factory.xml)  XmlBeanDefinitionReader.doLoadBeanDefinitions(InputSource, Resource) (org.springframework.beans.factory.xml)  XmlBeanDefinitionReader.loadBeanDefinitions(InputSource, String) (org.springframework.beans.factory.xml)  XmlBeanDefinitionReader.loadBeanDefinitions(EncodedResource) (org.springframework.beans.factory.xml)  GroovyBeanDefinitionReader.loadBeanDefinitions(EncodedResource) (org.springframework.beans.factory.groovy)  XmlBeanDefinitionReader.loadBeanDefinitions(Resource) (org.springframework.beans.factory.xml)  AbstractBeanDefinitionReader.loadBeanDefinitions(Resource...) (org.springframework.beans.factory.support)  AbstractXmlApplicationContext.loadBeanDefinitions(XmlBeanDefinitionReader) (org.springframework.context.support)  AbstractXmlApplicationContext.loadBeanDefinitions(DefaultListableBeanFactory) (org.springframework.context.support)  AbstractRefreshableApplicationContext.refreshBeanFactory() (org.springframework.context.support)  解析和注册核心方法：<https://blog.csdn.net/lisongjia123/article/details/52129340>  其实最后解析的重任交给了processBeanDefinition这个方法，而这个方法里面的实现过程在BeanDefinitionParserDelegate这个类当中!! 方法如下：  org.springframework.beans.factory.xml.DefaultBeanDefinitionDocumentReader.processBeanDefinition()  注意：这个方法里面包含了两大子过程， 1. 使用BeanDefinitionParserDelegate委托类来完成bean的element元素解析成BeanDefinition对象； 2. 然后再将统一数据结构的beanDefinition对象，注册到IOC容器中，才能使用！这个注册方法在DefaultListableBeanFactory.registerBeanDefinition()完成，注册之后的beanDefinition放在工厂中的beanDefinitionMap里面维护。   1. Bean配置文件解析完成，并登记之后，才开始进行Bean的实例化流程，实例化的入口流程（对照图13-12，13-13）:   DefaultListableBeanFactory.preInstantiateSingletons() (org.springframework.beans.factory.support.)  AbstractApplicationContext.finishBeanFactoryInitialization(ConfigurableListableBeanFactory) (org.springframework.context.support)  AbstractApplicationContext.refresh() (org.springframework.context.support)  上面这个是大的入口方法，从preInstantiateSingletons()开始，bean实例的创建过程入口!!   1. 建立bean对象实例关系（bean实例化中间的一子流程）的时序分析？参照图 13-14   这个时序分析的入口是紧接着，上面bean实例化的过程，也就是从实例化过程中判断单例和多例之后，AbstractBeanFactory.doGetBean()方法中，调用AbstractBeanFactory.createBean()方法，由子类实现AbstractAutowireCapableBeanFactory.createBean()，从这个实现方法中进行分析！   1. Spring中核心IOC容器实现原理?   <https://blog.csdn.net/lisongjia123/article/details/52129340> （清晰）  这里主要从一个最原始的IoC容器XmlBeanFactory的使用，介绍了在使用中的调用过程。简单的使用过程：  1.创建Ioc配置文件的抽象资源，这个抽象资源包含了BeanDefinition的定义信息  2.创建一个BeanFactory，这里使用了DefaultListableBeanFactory  3.创建一个载入BeanDefinition的读取器，这里使用XmlBeanDefinitionReader来载入XML文件形式的BeanDefinition  4.然后将上面定位好的Resource，通过一个回调配置给BeanFactory  5.从定位好的资源位置读入配置信息，具体的解析过程由XmlBeanDefinitionReader完成  6.完成整个载入和注册Bean定义之后，需要的Ioc容器就初步建立起来了  然后，可以对最原始的Ioc容器做一定改写实现功能！然后，分析了资源加载类ClassPathResource，  BeanDefinition类似于Resource接口的功能，起到的作用就是对所有的Bean进行一层抽象的统一，把形式各样的对象统一封装成一个便于Spring内部进行协调管理和调度的数据结构，BeanDefinition屏蔽了不同对象对于Spring框架的差异。之后，这里介绍了bean的解析和注册的核心过程！   1. Spring依赖注入的原理实现？@autowired和@Resource的区别，包括底层实现区别？   <https://blog.csdn.net/lisongjia123/article/details/52134396> （源码剖析spring依赖注入）这篇文章的讲解过程，基本上和书中的过程一样！不过更加详细一些！其中的核心在于： AbstractAutowireCapableBeanFactory.doCreateBean()方法中createBeanInstance(beanName, mbd, args)方法和populateBean(beanName, mbd, instanceWrapper)，分别完成bean实例的创建和依赖关系的注入。最后，还有bw.setPropertyValues(new MutablePropertyValues(deepCopy))，实现对属性值的注入。  上篇文章介绍Bean经过 **载入** 、**解析** 和 **注册** 这三个过程后，在框架内部被抽象封装成BeanDefinition这种类型，然后交由BeanFactory当中的beanDefinitionMap统一管理起来，到这一步，Spring还不能提供给我们一个完整可用的Bean，虽然所有的Bean已经在Spring经过解析和转化过程被集中组织起来，但此时这些BeanDefinition对象都是彼此分离的，并没有建立起依赖关系！  依赖注入起到的作用就是讲对象之间的依赖关系从原先的代码中解耦出来，通过配置文件或注解等方式加上Spring框架的处理让我们对依赖关系灵活集中的进行管理。   1. SpringBean生命周期详解 <https://blog.csdn.net/lisongjia123/article/details/52091013>   里面有个bean生命周期示意图比较好！有关bean生命周期的方法分类如下表！最后，关于bean周期的案例特别经典！   1. Spring在使用ByName方式，注入同名bean的问题，如何设置不能同名注入？   <https://blog.csdn.net/lisongjia123/article/details/52041945>  默认如果同名, 后面的会覆盖前面的，不会报错。  怎么样在项目中避免这种同名造成的问题？解决方案：不允许重名，如果重名就报错！（源码里面如果设置了allowBeanDefinitionOverriding为false即可。）可以使用定义监听器，来设置context.setAllowBeanDefinitionOverriding(false); 也可以自定义ApplicationContextInitializer类，启动的时候在初始化的时候就设置applicationContext.setAllowBeanDefinitionOverriding(false);   1. 刚好上面这个问题和“@Resource和@Autowired的区别”问题，比较来学习？   <https://blog.csdn.net/wangzuojia001/article/details/54312074/>  <https://www.cnblogs.com/think-in-java/p/5474740.html>  其实就是@resource默认按名称自动注入，注入的时候，可以带上name属性（resource也有type属性，也可以按类型来注入），而@autowired默认按类型注入，但如果有同一个接口多个实现类，就可以结合@qualifier注解来确定哪一个实现类。所以二者都可以按类型或按名称注入，主要是默认的不一样。   1. beanFactory和FactoryBean和ProxyFactoryBean   BeanFactory它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。  FactoryBean(通常情况下，bean无须自己实现工厂模式，Spring容器担任工厂角色；但少数情况下，容器中的bean本身就是工厂，其作用是产生其它bean实例),作用是产生其他bean实例。通常情况下，这种bean没有什么特别的要求，仅需要提供一个工厂方法，该方法用来返回其他bean实例。由工厂bean产生的其他bean实例，不再由Spring容器产生，因此与普通bean的配置不同，不再需要提供class元素。  ProxyFactoryBean用于创建代理(根据Advisor生成的Bean，也就是TargetBean的代理)，我们的Advisor，PointCut等等，其最终目的都是为了创建这个代理。   1. AOP在源码中的原理？ 2. Spring动态代理实现的方式？特点？ |

|  |  |
| --- | --- |
| **分类类型** | **所包含方法** |
| Bean自身的方法 | 配置文件中的init-method和destroy-method配置的方法、Bean对象自己调用的方法 |
| Bean级生命周期接口方法 | BeanNameAware、BeanFactoryAware、InitializingBean、DiposableBean等接口中的方法 |
| 容器级生命周期接口方法 | InstantiationAwareBeanPostProcessor、BeanPostProcessor等后置处理器实现类中重写的方法 |

### Spring研究(基础)

说明：这次研究是基于《Spring in action 4th Edition》中文版，本书言简意赅！

#### 装配技术

1. Spring 自动转配，可以使用的注解除了自己的@AutoWired还可以使用@Resource，以及javax.inject包中提供的@Inject、@Named、@Qualifier和@Provider用法。

<https://blog.csdn.net/binxigogo/article/details/7851924>

1. 使用java代码装配bean的应用场景

你想要将第三方库中的组件装配到你的应用中， 在这种情况下， 是没有办法在它的类上添加@Component和@Autowired注解的，因此就不能使用自动化装配的方案了。

在这种情况下，你必须要采用显式装配的方式。在进行显式配置的时候，有两种可选方案：Java和XML。

1. @Bean注解会告诉Spring这个方法将会返回一个对象，该对象要注册为Spring应用上下文中的bean。方法体中包含了最终产生bean实例的逻辑。默认情况下， bean的ID与带有@Bean注解的方法名是一样的。也可以通过name属性指定一个不同的名字。
2. 有关spring使用依赖注入时，对引用的Bean是使用同一个bean还是每次都创建的论述

《spring实战》75页。同时，这里还有通过方法和@Bean结合使用，注入bean的两种方法说明：1. 调用sgtPeppers()方法，2. 直接在注入方法中传参方式注入（这种方式通常是最佳选择）

1. Spring使用配置方式实例化bean的时候，调用的是默认的无参数构造器！所以xml配置方式没有使用@bean方式灵活，如果想设置参数，需要<constructor-arg>来完成注入；还可以使用spring的c-命名空间（使用方法见P81）。如何解决构造器参数中的list类型参数注入？（使用<list>标签或<set>）
2. 当注入一个属性的时候，是使用属性注入还是构造器注入？原则？

该选择构造器注入还是属性注入呢？ 作为一个通用的规则， 我倾向于对强依赖使用构造器注入， 而对可选性的依赖使用属性注入。<property>元素为属性的Setter方法所提供的功能，类似地还有p-命名空间。同样，<property>标签中也可以使用<list>解决list，set或数组属性的注入。

1. 可以使用util-命名空间来简化bean中一些复杂属性的注入，比方说，list,map,set, property
2. 在典型的Spring应用中， 我们可能会同时使用自动化和显式配置。这些配置方案之间并不互相排斥
3. 如何解决如果有多个和多种配置，这些配置中都有自己定义的bean，可能采用显式的编程式，也可能是xml自动配置式，怎么将这么多配置文件和定义的bean统一到一起，这个问题其实非常具有使用价值。解决方式: 2.5章 导入和混合配置（一个是javaConfig引入xml，还有一个是xml中引入javaConfig），总之最后都合并到了一个根配置中！

#### 高级装配

1. 解决不同运行环境中的问题的概述

Spring为环境相关的bean所提供的解决方案其实与构建时的方案没有太大的差别。当然，在这个过程中需要根据环境决定该创建哪个bean和不创建哪个bean。 不过Spring并不是在构建的时候做出这样的决策， 而是等到运行时再来确定。这样的结果就是同一个部署单元（可能会是WAR文件）能够适用于所有的环境，没有必要进行重新构建。在3.1版本中， Spring引入了bean profile的功能。 要使用profile， 你首先要将所有不同的bean定义整理到一个或多个profile之中， 在将应用部署到每个环境时，要确保对应的profile处于激活（active）的状态。

1. 使用@Profile配置profile bean，在Spring 3.1中， 只能在类级别上使用@Profile注解。 不过， 从Spring 3.2开始， 你也可以在方法级别上使用@Profile注解，与@Bean注解一同使用。这样的话， 就能将这两个bean的声明放到同一个配置类之中。
2. 还可以使用xml配置文件中使用Profile，在<beans>元素中设置profile属性！你还可以在根<beans>元素中嵌套定义<beans>元素， 而不是为每个环境都创建一个profile XML文件。见P104
3. 怎样激活profile？

Spring依赖两个属性：spring.profiles.active和spring.profiles.default。

有多种方式来设置这两个属性：

作为DispatcherServlet的初始化参数；

作为Web应用的上下文参数；

作为JNDI条目；

作为环境变量；

作为JVM的系统属性；

在集成测试类上， 使用@ActiveProfiles注解设置

例如，在web.xml中设置，见P106。你可能已经注意到了， 在spring.profiles.active和

spring.profiles.default中， profile使用的都是复数形式。 这意味着你可以同时激活多个profile， 这可以通过列出多个profile名称， 并以逗号分隔来实现。当然， 同时启用dev和prod profile可能也没有太大的意义， 不过你可以同时设置多个彼此不相关的profile。

1. 测试代码中，可以使用@ActiveProfiles激活测试的profile！
2. Spring 4.0中提供了一种更为通用的机制来实现条件化的bean定义！

在这种机制之中， 条件完全由你来确定。条件化bean的应用场景：

假设你希望一个或多个bean只有在应用的类路径下包含特定的库时才创建。 或者我们希望某个bean只有当另外某个特定的bean也声明了之后才会创建。 我们还可能要求只有某个特定的环境变量设置之后， 才会创建某个bean！

Spring 4引入了一个新的**@Conditional**注解， 它可以用到带有@Bean注解的方法上。 如果给定的条件计算结果为true， 就会创建这个bean，否则的话，这个bean会被忽略。设置给@Conditional的类可以是任意实现了Condition接口的类型。 可以看出来， 这个接口实现起来很简单直接（可以自定义化具体的条件，只需要实现Condition接口即可，而这个实现类会自动地被创建为bean，见P108）， 只需提供matches()方法的实现即可！

1. 注意：Condition接口中的match()方法中的两个参数！

boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata); 这个两个参数可以获得判断条件的非常多的属性，见P109。非常有意思的是，从Spring 4开始， @Profile注解进行了重构，使其基于@Conditional和Condition实现（具体实现P110）。

1. Spring如何解决自动装配中，发生歧义，不唯一的问题？（见3.3章节）

经典场景：自动装配的bean，是同一个接口实现的多个不同的实现类，并且都是bean，没有特殊指定！

解决方案：当确实发生歧义性的时候， Spring提供了多种可选方案来解决这样的问题。 你可以将可选bean中的某一个设为首选（primary） 的bean， 或者使用限定符（qualifier） 来帮助Spring将可选的bean的范围缩小到只有一个bean。

1. 使用primary，可以使用注解应用，也可以在<bean>标签中作为属性设置，注意只有一个可用
2. @Qualifier注解是使用限定符的主要方式。 它可以与@Autowired和@Inject协同使用， 在注入的时候指定想要注入进去的是哪个bean。当使用java配置显式定义bean的时候，@Qualifier可以和@Bean结合使用！
3. 使用自定义的限定符注解！（高级）

场景：面向特性的限定符要比基于bean ID的限定符更好一些。 但是， 如果多

个bean都具备相同特性的话， 这种做法也会出现问题。由于一个bean上面不能同时使用多个@Qualifier限定符，所以需要自定义限定符注解！P115，

1. Spring定义了多种作用域，可以基于这些作用域（还包括session和request）创建bean！见P118。如何配置多例的Bean？（组建扫描的自动注入，显式配置，xml配置）

在默认情况下， Spring应用上下文中所有bean都是作为以单例（singleton） 的形式创建的。 也就是说， 不管给定的一个bean被注入到其他bean多少次， 每次所注入的都是同一个实例。

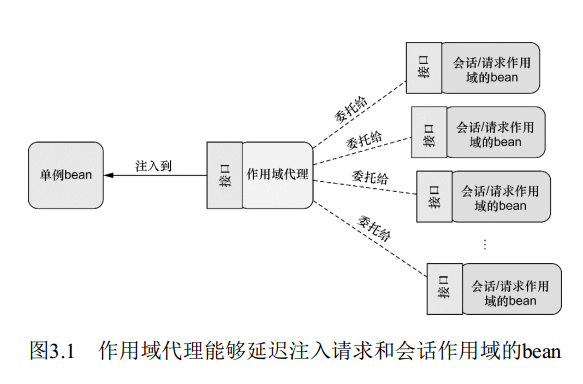
1. 会话和请求作用域，典型的购物车应用场景，以及ProxyMode解决的问题（P119-P120）！（比较有实用意义，作用域代理！）

首先是，会话作用域范围的购物车bean应该如何定义？其次是将这种会话作用域的bean注入到普通的StoreService这种bean中，会涉及到初始化、和多个会话的问题！

Spring并不会将实际的ShoppingCart bean注入到StoreService中，Spring会注入一个到ShoppingCart bean的代理， 如图3.1所示。 这个代理会暴露与ShoppingCart相同的方法， 所以StoreService会认为它就是一个购物车。 但是， 当StoreService调用ShoppingCart的方法时， 代理会对其进行懒解析并将调用委托给会话作用域内真正的ShoppingCart bean。

上面的对ShoppingCart这个session域的bean是使用注解进行的配置，也可以使用xml配置，来设置session域，同时使用<aop:scoped-proxy>属性来设置委托创建代理的方式！具体见3.4.2章节

1. Spring在运行时注入，防止硬编码的解决方案？
2. 使用@PropertySource注解和Environment，案例见P123， Environment还提供了几个与属性相关的方法。Environment还提供了一些方法来检查哪些profile处于激活状态！
3. 使用属性占位符。如果我们依赖于组件扫描和自动装配来创建和初始化应用组件的话，那么就没有指定占位符的配置文件或类了。 在这种情况下， 我们可以使用@Value注解， 它的使用方式与@Autowired注解非常相似。为了使用占位符， 我们必须要配置一个PropertyPlaceholderConfigurer bean或PropertySourcesPlaceholderConfigurer bean。还有就是xml配置中，可以配置出这个bean。



#### 面向切面编程

1. 应用场景问题

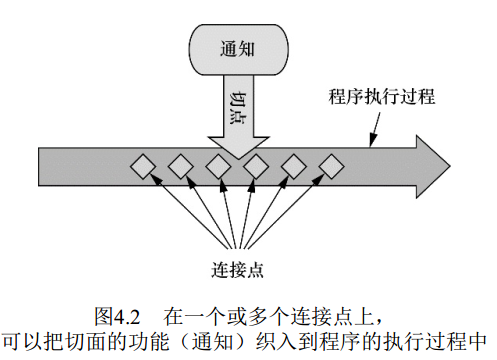
在软件开发中， 散布于应用中多处的功能被称为横切关注点（crosscutting concern） 。 通常来讲， 这些横切关注点从概念上是与应用的业逻辑相分离的（但是往往会直接嵌入到应用的业务逻辑之中）。把这些横切关注点与业务逻辑相分离正是面向切面编程（AOP） 所要解决的问题。

比如：日志、 安全和事务管理、缓存！

DI有助于应用对象之间的解耦，而AOP可以实现横切关注点与它们所影响的对象之间的解耦。每个模块的核心功能都是为特定业务领域提供服务， 但是这些模块都需要类似的辅助功能，例如安全和事务管理。

仍然在一个地方定义通用功能， 但是可以通过声明的方式定义这个功能要以何种方式在何处应用， 而无需修改受影响的类。

1. 有关AOP的术语，示意图非常好，具体的解释见4.1.1章节



切面的工作被称为通知，Spring可以定义五种类型的通知（P140，重要，核心方法）！切点的匹配方法，可以是类，方法或是正则匹配。切面就是切点和通知的结合。通知和切点共同定义了切面的全部内容——它是什么，在何时和何处完成其功能！织入是把切面应用到目标对象并创建新的代理对象的过程。目标对象在生命周期中，有多少个点可以被织入？P141

1. Spring提供了四种AOP的支持：P142，spring经典的aop太笨重，

借助Spring的aop命名空间， 我们可以将纯POJO转换为切面。 实际上， 这些POJO只是提供了满足切点条件时所要调用的方法，但是需要xml配置一起用才行。

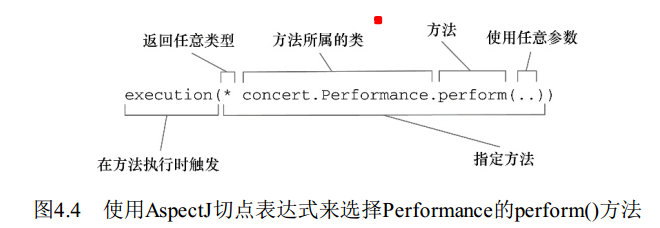
Spring AOP构建在动态代理基础之上， 因此， Spring对AOP的支持局限于方法拦截。

Spring借鉴了AspectJ的切面，以提供注解驱动的AOP。本质上，它依然是Spring基于代理的AOP，**但是编程模型几乎与编写成熟的AspectJ注解切面完全一致。**这种AOP风格的好处在于能够不使用XML来完成功能！

1. Spring在运行时通知对象，只支持方法级别的连接点！

因为Spring基于动态代理，所以Spring只支持方法连接点。这与一些其他的AOP框架是不同的， 例如AspectJ和JBoss，除了方法切点，它们还提供了字段和构造器接入点。 Spring缺少对字段连接点的支持，无法让我们创建细粒度的通知，例如拦截对象字段的修改。而且它不支持构造器连接点，我们就无法在bean创建时应用通知！但是，方法级别的连接点，已经能满足大部分需求。

1. Spring使用切点来选择连接点，Spring借助AspectJ的切点表达式语言来定义Spring切面，具体见P145的表，之后的示例，还对一些额外的配置结合使用。



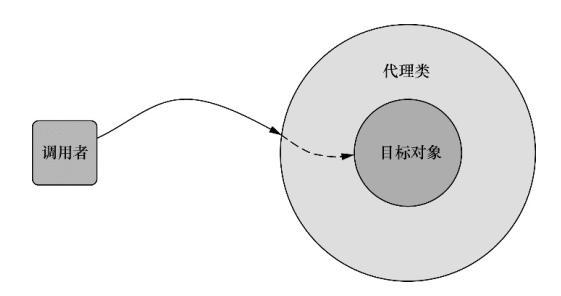
使用注解来创建切面，这种是比较常用和经典的做法。使用@AspectJ注解，该注解表

明Audience不仅仅是一个POJO， 还是一个切面。Spring使用AspectJ注解来声明通知方法，具体解释见P149的表。Aspect里头具体的通知，在目标方法不同阶段的定义切点表达式，可以不一样，但如果一样的话，可以有个统一的设置。@Pointcut注解能够在一个@AspectJ切面内定义可重用的切点。

最后，要真正将@AspectJ的注解类转换为真正的切面代理，注意！还得将这个注解类定义为bean！

1. **切面的原理**

通过在代理类中包裹切面，Spring在运行期把切面织入到Spring管理的bean中。如图4.3所示，**代理类封装了目标类，**并拦截被通知方法的调用，再把调用转发给真正的目标bean。 **当代理拦截到方法调用时，在调用目标bean方法之前，会执行切面逻辑。**直到应用需要被代理的bean时，Spring才创建代理对象。如果使用的是ApplicationContext的话，在ApplicationContext从BeanFactory中加载所有bean的时候， Spring才会创建被代理的对象。Spring运行时才创建代理对象，所以我们不需要特殊的编译器来织入Spring AOP的切面。



**图4.3 Spring的切面由包裹了目标对象的代理类实现。代理类处理方法的调用，执行额外的切面逻辑，并调用目标方法**

不管你是使用JavaConfig还是XML， AspectJ自动代理都会为使用@Aspect注解的bean创建一个代理，这个代理会围绕着所有该切面的切点所匹配的bean。在这种情况下，将会为Concretebean创建一个代理，Audience类中的通知方法将会在perform()调用前后执行。我们需要记住的是， Spring的AspectJ自动代理仅仅使用@AspectJ作为创建切面的指导， 切面依然是基于代理的。在本质上，它依然是Spring基于代理的切面。这一点非常重要，因为这意味着尽管使用的是@AspectJ注解， 但我们仍然限于代理方法的调用。

1. 环绕通知的使用

见P152清单4.5，注意方法的参数非常重要！ 环绕通知是最为强大的通知类型。 它能够让你所编写的逻辑将被通知的目标方法完全包装起来。 实际上就像在一个通知方法中同时编写前置通知和后置通知。

ProceedingJoinPoint的proceed()方法的高级应用场景！你可以不调用proceed()方法， 从而阻塞对被通知方法的访问， 与之类似， 你也可以在通知中对它进行多次调用。 要这样做的一个场景就是实现重试逻辑！

1. 如果切面所通知的方法确实有参数该怎么办呢？ 切面能访问和使用传递给被通知方法的参数吗？

具体的讲解和实现案例，见P154。

1. 通过注解引入新功能，也就是为类增加新方法，需要使用@DeclareParents注解，具体示例见P158。以及使用Aspect注解的劣势。4.3.4章节！

回顾一下， 在Spring中，切面只是实现了它们所包装bean相同接口的代理。如果除了实现这些接口，代理也能暴露新接口的话，会怎么样呢？那样的话，切面所通知的bean看起来像是实现了新的接口，即便底层实现类并没有实现这些接口也无所谓。

我们需要注意的是， 当引入接口的方法被调用时， 代理会把此调用委托给实现了新接口的某个其他对象。 实际上， 一个bean的实现被拆分到了多个类中。

Spring的自动代理机制将会获取到它的声明， 当Spring发现一个bean使用了@Aspect注解时， Spring就会创建一个代理， 然后将调用委托给被代理的bean或被引入的实现， 这取决于调用的方法属于被代理的bean还是属于被引入的接口。

1. 使用xml来声明切面，aop命名空间的其他元素能够让我们直接在Spring配置中声明切面， 而不需要使用注解。有关aop命名空间的详解：P160的表，案例：P162的清单。对应关系的详解：P163清单，大致的配置和注解式相似，包括环绕通知。
2. Xml方式，处理通知中的参数示例：P167的清单。有关AspectJ与spring注入的协同使用，将Aspect也让spring容器管理的特殊设置：P171，要在这个bean设置factory-method=“aspectOf”。

因为Spring不能负责创建CriticAspect， 那就不能在 Spring中简单地把CriticAspect声明为一个bean。 相反， 我们需要一种方式为Spring获得已经由AspectJ创建的CriticAspect实例的句柄， 从而可以注入CriticismEngine。 幸好， 所有的AspectJ切面都提供了一个静态的aspectOf()方法， 该方法返回切面的一个单例。 所以为了获得切面的实例， 我们必须使用factory-method来调用asepctOf()方法而不是调用CriticAspect的构造器方法。

#### Spring 数据库访问

1. Spring将数据访问过程中固定的和可变的部分明确划分为两个不同的类： 模板（template） 和回调（callback）。模板管理过程中固定的部分，而回调处理自定义的数据访问代码。示意图和具体的解析，还有典型的模版见表：P350
2. Spring提供了在Spring上下文中配置数据源bean的多种方式， 包括：
3. 通过JDBC驱动程序定义的数据源；
4. 通过JNDI查找的数据源；
5. 连接池的数据源。

对于即将发布到生产环境中的应用程序， 我建议使用从连接池获取连接的数据源。 如果可能的话， 我倾向于通过应用服务器的JNDI来获取数据源。

1. 使用数据源连接池，典型的xml配置和java配置方式见P353，连接池的方式比较适合生产环境
2. 基于JDBC驱动的数据源，spring提供了三种， 见10.2.3章节

尽管这些数据源对于小应用或开发环境来说是不错的， 但是要将其用于生产环境， 你还是需要慎重考虑。 因为SingleConnectionDataSource有且只有一个数据库连接， 所以不适合用于多线程的应用程序， 最好只在测试的时候使用。而DriverManagerDataSource和SimpleDriverDataSource尽管支持多线程， 但是在每次请求连接的时候都会创建新连接， 这是以性能为代价的。 鉴于以上的这些限制， 我强烈建议应该使用数据源连接池。

1. 使用profile选择数据源

借助Spring的profile特性能够在运行时选择数据源，java配置具体实例见P358清单，xml配置清单见P359，注意<beans>根标签和子标签

1. 古老的jdbc相比目前很多的持久层框架有什么优点？见P359，10.3章节讨论。传统jdbc典型的操作数据库程序代码（包括插入，更新，以及查询！），见P360，以及缺点的论述！P362的传统jdbc查询操作代码，需要重点关注！（包括处理结果！）
2. Spring的JDBC框架承担了资源管理和异常处理的工作， 从而简化了JDBC代码， 让我们只需编写从数据库读写数据的必需代码。

**Spring为JDBC提供了三个模板类供选择：**

JdbcTemplate： 最基本的Spring JDBC模板，这个模板支持简单的JDBC数据库访问功能以及基于索引参数的查询；（大多数就选这个最好）

NamedParameterJdbcTemplate：使用该模板类执行查询时可以将值以命名参数的形式绑定到SQL中，而不是使用简单的索引参数；

SimpleJdbcTemplate：该模板类利用Java 5的一些特性如自动装箱、泛型以及可变参数列表来简化JDBC模板的使用。（从3.1版本，基本废弃）

1. 使用JdbcTemplate操作数据库！Spring中配置非常简单！见P363。操作查询的示例（比较经典）见P365
2. 使用可以对参数进行命名的NamedParameterJdbcTemplate，这样就不用太注意参数的顺序了。
3. 数据操作的一些复杂的特性：P369。注意这些复杂的特性在不同框架中的处理方式，主要是配置！（hibernate, mybatis）

一些可用的框架提供了这样的服务， 这些服务的通用名称是对象/关系映射（object-relational mapping， ORM） 。 在持久层使用ORM工具， 可以节省数千行的代码和大量的开发时间。 ORM工具能够把你的注意力从容易出错的SQL代码转向如何实现应用程序的真正需求。Spring对ORM框架的支持提供了与这些框架的集成点以及一些附加的服务：

支持集成Spring声明式事务；

透明的异常处理；

线程安全的、 轻量级的模板类；

DAO支持类；

资源管理

1. **Hibernate**

使用Hibernate所需的主要接口是org.hibernate.Session。 Session接口提供了基本的数据访问功能， 如保存、 更新、 删除以及从数据库加载对象的功能。获取Hibernate Session对象的标准方式是借助于HibernateSessionFactory接口的实现类。

如何获取需要映射的实体类？xml配置或包扫描注解类

1. 使用xml定义映射，配置LocalSessionFactoryBean方式（java）P372。主要配置dataSource，mapping文件，以及properties设置细节。
2. 如果使用注解来定义持久化。那么需要使用AnnotationSessionFactoryBean来代替LocalSessionFactoryBean：见P372java配置
3. 你还可以使用annotatedClasses属性来将应用程序中所有的持久化类以全限定名的方式明确列出。

这里不再列出Hibernate配置文件，而是使用packagesToScan属性告诉Spring扫描一个或多个包以查找域类，这些类通过注解的方式表明要使用Hibernate进行持久化，这些类可以使用的注解包括JPA的@Entity或@MappedSuperclass以及Hibernate的@Entity。

1. 构建不依赖于Spring的Hibernate代码，实现repository？不需要spring耦合的HibernateTemplate，而是使用而是使用上下文Session（Contextual session）：见P374
2. **Spring与Java持久化API（JPA）**

JPA是基于POJO的持久化机制， 它从Hibernate和Java数据对象（Java Data Object， JDO） 上借鉴了很多理念并加入了Java 5注解的特性。在Spring中使用JPA的第一步是要在Spring应用上下文中将实体管理器工厂（entity manager factory）按照bean的形式来进行配置。

1. JPA实体管理器类型，以及JavaEE适用于哪种？这两种实体管理器在spring中是如何创建的？见P376

不管你希望使用哪种EntityManagerFactory，Spring都会负责管理EntityManager。 如果你使用的是应用程序管理类型的实体管理器，Spring承担了应用程序的角色并以透明的方式处理EntityManager。在容器管理的场景下，Spring会担当容器的角色。

1. 应用程序管理类型和容器管理类型的实体管理器工厂之间唯一值得关注的区别是在Spring应用上下文中如何进行配置。
2. 配置应用程序管理类型的JPA，persistence.xml的配置文件位置、内容，以及作用，注入spring？缺点？P377
3. 使用容器管理类型的JPA（更值得关注），将数据源信息配置在Spring应用上下文中， 而不是在persistence.xml，在Spring中如何使用LocalContainerEntity-ManagerFactoryBean来配置容器管理类型的JPA？配置的重要属性（注意后面还设置了扫描包）？Hibernate的JPA适配器？见P379
4. 编写基于JPA的Repository。不使用spring的JpaTemplate，使用纯粹的JPa开发repository示例，见P382。EntityManager并不是线程安全的，如何解决每次操作数据库就创建EntityManager的问题？解决方案以及解析，由JPA规范提供的@PersistenceUnit和@PersistenceContext，见P383。

正如Spring对其他持久化方案的集成一样， Spring对JPA集成也提供了JpaTemplate模板以及对应的支持类JpaDaoSupport。但是鉴于纯粹的JPA方式远胜于基于模板的JPA！

1. 借助Spring Data实现自动化的JPARepository，注意spring-data是spring一个独立的工程，需要添加依赖才能使用！如何实现自动化？如何配置spring data jpa？最重要的是提供18个便利的方法来进行通用的JPA操作（无需编写），如果要自己扩展其他方法呢？见11.3开篇！

JpaRepository又扩展自Repository标记接口（虽然是间接的），也就是Repository扫描时所要查找的接口。 当Spring Data找到它后， 就会创建SpitterRepository的实现类， 其中包含了继承自JpaRepository、 PagingAndSortingRepository和CrudRepository的18个方法。很重要的一点在于Repository的实现类是在应用启动的时候生成的，也就是Spring的应用上下文创建的时候。 它并不是在构建时通过代码生成技术产生的， 也不是接口方法调用时才创建的。

1. 如何使用spring data jpa提供的语法结构，扩展一些方法 见13.3.1
2. 如何使用纯自定义的扩展，见13.3.2，13.3.3

当Spring Data JPA为Repository接口生成实现的时候， 它还会查找名字与接口相同， 并且添加了Impl后缀的一个类。如果这个类存在的话， Spring Data JPA将会把它的方法与Spring Data JPA所生成的方法合并在一起。 对于SpitterRepository接口而言， 要查找的类名为SpitterRepositoryImpl。案例见P393代码清单。

注意：自己的实现类SpitterRepositoryImpl不需要实现SpitterRepository，只需要实现自己的接口，但是新增的方法必须首先在自己的接口中声明，然后让SpitterRepository继承这个接口，具体看上面的案例！还可以自己定义这个扩展名不是Impl，而是使用自己定义的：见P394。

## SpringMVC 研究

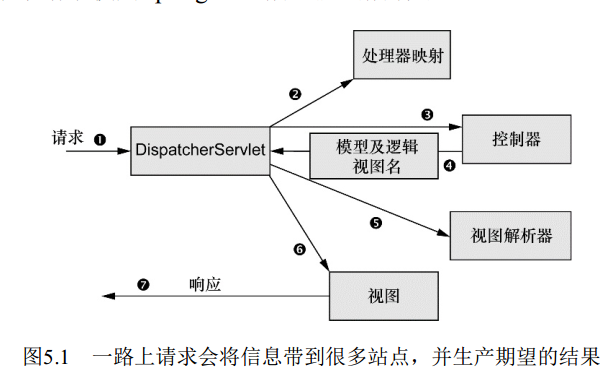
说明：这次研究是基于《Spring in action 4th Edition》中文版，本书言简意赅！

#### 基础

1. SpringMVC的请求流程

Spring MVC所有的请求都会通过一个前端控制器（front controller） Servlet。 前端控制器是常用的Web应用程序模式， 在这里一个单实例的Servlet将请求委托给应用程序的其他组件来执行实际的处理。

详细的解析工艺流程见5.1.1章节，这比其他地方看到的清晰多了！



1. 配置DispatchServlet，如何使用java直接将DispatcherServlet配置在servlet容器中？以及原理的解释（P177）

按照传统的方式， 像DispatcherServlet这样的Servlet会配置在web.xml文件中， 这个文件会放到应用的WAR包里面。 当然， 这是配置DispatcherServlet的方法之一。 但是， 借助于Servlet 3规范和Spring 3.1的功能增强， 这种方式已经不是唯一的方案了。

1. 理解DispatcherServlet和servlet监听器，也就是ContextLoaderListener的关系？AbstractAnnotationConfigDispatcherServletInitializer这个类都做了什么工作？（p178剖析）
2. 启动springMVC的方式？
3. 对springMVC使用java注解配置方式的解析？主要分为rootConfig和webConfig两方面的配置

示例如P181，代码清单，

1. Spring 中提供了多种视图解析器？具体见P211，这个介绍比较清晰！！

ResourceBundleViewResolver会直接将逻辑视图名映射为特定的View接口实现，而InternalResourceViewResolver所采取的方式并不那么直接。 它遵循一种约定， 会在视图名上添加前缀和后缀， 进而确定一个Web应用中视图资源的物理路径。

InternalResourceViewResolver会将视图名解析为JSP文件。另外， 如果在你的JSP页面中使用了JSP标准标签库（JavaServer Pages Standard Tag Library， JSTL）的话，InternalResourceViewResolver能将视图名解析成JstlView形式的jsp文件， 从而将JSTL本地化和资源bundle变量暴露给JSTL的格式化（formatting） 和信息（message） 标签。

1. 如果InternalResourceViewResolver将视图解析成JstlView的用途？JSTL标签来处理格式化和信息？Locale对象？以及使用java配置或xml如何配置？见P215。
2. Jsp作为视图展示的缺点？ 见P237，Thymeleaf的优势？
3. 配置Thymeleaf视图解析器（需要配置三个启用Thymeleaf与Spring集成的bean）？（包括java配置和xml配置两种方式）不管使用哪种配置方式， 它可以将响应中的模板渲染到Spring MVC控制器所处理的请求中！实现原理？见6.4.1章节
4. Thymeleaf模版的定义？见6.4.2章节

这意味着Thymeleaf模板与JSP不同， 它能够按照原始的方式进行编辑甚至渲染， 而不必经过任何类型的处理器。

完整的使用Thymeleaf模板的表单实现，见P243。

“${}”表达式（如${spitter}） 是变量表达式，对于“\*{}”表达式， 它们是选择表达式（selection expression） 。 变量表达式是基于整个SpEL上下文计算的， 而选择表达式是基于某一个选中对象计算的。

#### 高级特性

1. 如何用其他的方式来搭建DispatcherServlet和ContextLoaderListener？

应用场景：P247，

自定义DispatcherServlet配置（7.1.1章节），在SpittrWebAppInitializer中我们所编写的三个方法仅仅是必须要重载的abstract方法。 但实际上还有更多的方法可以进行重载， 从而实现额外的配置。其中之一，就是customizeRegistration()。借助customizeRegistration()方法中的ServletRegistration.Dynamic， 我们能够完成多项任务， 包括通过调用tLoadOnStartup()设置load-on-startup优先级， 通过setInitParameter()设置初始化参数， 通过调用setMultipartConfig()配置Servlet 3.0对multipart的支持。

1. 按照AbstractAnnotationConfigDispatcherServletInitializer的定义，它会创建DispatcherServlet和ContextLoaderListener。但是，如果你想注册其他的Servlet、Filter或Listener的话，那该怎么办呢？

见7.1.3

1. 使用web.xml声明DispatcherServlet

见7.1.3，其实配置中主要也是让DispatcherServlet和ContextLoaderListener从XML中加载各自的应用上下文。

传统的都是使用xml配置上下文，还有可以使用web.xml与java配置类的结合，见P253

1. SpringMVC中的文件上传，配置multipart解析器？限制用户上传文件的大小，该怎么实现？ 如果我们想要指定文件在上传时， 临时写入目录在什么位置的话， 该如何实现？注意两种解析器的实现！

一般来讲， 在这两者之间， StandardServletMultipartResolver可能会是优选的方案。 它使用Servlet所提供的功能支持，并不需要依赖任何其他的项目。如果我们需要将应用部署到Servlet 3.0之前的容器中，或者还没有使用Spring 3.1或更高版本，那么可能就需要CommonsMultipartResolver了。

1. 如何对StandardServletMultipartResolver配置限制条件？包括java配置和xml配置？P256-257
2. 采用Servlet初始化类的方式 ，（这里用到了servletContext中的public ServletRegistration.Dynamic addServlet( String servletName, Servlet servlet)方法返回一个用于该servlet进一步配置的额Dynamic注册类）
3. 如果我们配置DispatcherServlet的Servlet初始化类继承了Abstract AnnotationConfigDispatcherServletInitializer或AbstractDispatcher-ServletInitializer （这里就是重写customizeRegistration()方法）
4. 如何配置CommonsMultipartResolver解析器？P258，如何设置目录，大小限制等？

与StandardServletMultipartResolver有所不同， CommonsMultipart-Resolver不会强制要求设置临时文件路径。 默认情况下， 这个路径就是Servlet容器的临时目录。 不过， 通过设置uploadTempDir属性， 我们可以将其指定为一个不同的位置。

1. 处理MultipartFile请求，接收上传文件的参数？见7.2.2章节

可以使用byte[]类型直接在@requestParam中接收，也可以使用MultipartFile类型接收（信息更加丰富！），还可以使用Spring MVC也能接受javax.servlet.http.Part作为控制器方法的参数。

1. 处理异常? Spring提供了多种方式将异常转换为响应：见P263。

不管发生什么事情， 不管是好的还是坏的， Servlet请求的输出都是一个Servlet响应。 如果在请求处理的时候， 出现了异常， 那它的输出依然会是Servlet响应。 异常必须要以某种方式转换为响应。

将异常映射为HTTP状态码，Spring的一些异常会默认映射为HTTP状态码（见P263表），可以使用@ResponseStatus注解将异常映射为HTTP状态码。（P265清单）

1. 在spring中的异常处理方法，如何实现？主要实现功能？见7.3.2章节，

对于@ExceptionHandler注解标注的方法来说，比较有意思的一点在于它能处理同一个控制器中所有处理器方法所抛出的异常。所以，尽管我们从saveSpittle()中抽取代码创建了handleDuplicateSpittle()方法，但是它能够处理SpittleController中所有方法所出的DuplicateSpittleException异常。我们不用在每一个可能抛出DuplicateSpittleException的方法中添加异常处理代码， 这一个方法就涵盖了所有的功能。但是如何处理所有控制器的异常呢？

1. 控制器通知的应用场景？@ControllerAdvice ？P268

控制器通知。 控制器通知（controller advice） 是任意带有@ControllerAdvice注解的类。这个类会包含一个或多个如下类型的方法：

@ExceptionHandler注解标注的方法；

@InitBinder注解标注的方法；

@ModelAttribute注解标注的方法

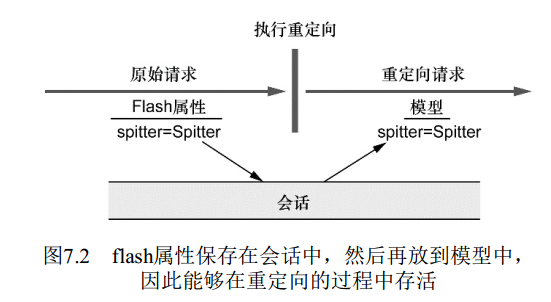
这些方法会运用到整个应用程序所有控制器中带有@RequestMapping注解的方法上。@ControllerAdvice最为实用的一个场景就是将所有的@ExceptionHandler方法收集到一个类中，这样所有控制器的异常就能在一个地方进行一致的处理。示例清单：P269

1. 跨重定向请求传递数据，应用场景？解决方案？见7.5

当控制器方法返回的String值以“redirect:”开头的话， 那么这个String不是用来查找视图的，而是用来指导浏览器进行重定向的路径。

Spring为重定向功能还提供了一些其他的辅助功能：正在发起重定向功能的方法该如何发送数据给重定向的目标方法呢？（很实用的问题！具体分析和解决方案见Ｐ269）！

1. 使用URL模板以路径变量和/或查询参数的形式传递数据；（存在的问题！解决方式，使用占位符）。通过路径变量和查询参数的形式跨重定向传递数据是很简单直接的方式， 但它也有一定的限制。 它只能用来发送简单的值， 如String和数字的值。 在URL中， 并没有办法发送更为复杂的值。（注意model里的值失效的问题！）
2. 通过flash属性发送数据。解决的问题：直接可以在重定向中传递对象，而不仅仅是简单的属性。
3. 重定向过程过程中传递复杂对象的解决方案？见P272
4. 可以考虑放在会话中，
5. 而spring是考虑放在flash属性中，原理：



2、最快的排序算法是哪个？给阿里2万多名员工按年龄排序应该选择哪个算法？堆和树的区别；写出快排代码；链表逆序代码；

3、求1000以内的水仙花数以及40亿以内的水仙花数；

4、子串包含问题(KMP 算法)写代码实现；

5、万亿级别的两个URL文件A和B，如何求出A和B的差集C,(Bit映射->hash分组->多文件读写效率->磁盘寻址以及应用层面对寻址的优化)

6、蚁群算法与蒙特卡洛算法；

7、写出你所知道的排序算法及时空复杂度，稳定性；

8、百度POI中如何试下查找最近的商家功能(坐标镜像+R树)。

9、遍历二叉树

10、自己集合实现一个队列

12、快速排序和冒泡的排序，怎么转换一下。

## 分布式问题（短板）

1. 分布式系统怎么做服务治理？✔
2. 如何保障请求顺序
3. 分布式事务和分布式锁
4. 分布式session设置
5. 分布式seesion的一致性
6. 分布式接口的幂等性设计（不能重复扣款）
7. 分布式事务的理解
8. 分布式事务知道吗？怎么解决的？TCC？若出现网络原因，网络连不通怎么办？
9. 分布式寻址方式都有哪些算法？知道一致性hash吗？手写一下java实现代码？若userId取摸分片，那我要查一段连续时间里的数据怎么办？

### 分布式搜索引擎

1. Es工作过程如何实现？如何实现分布式的？
2. Es在数据量很大的情况（数十亿级别），如何提高查询效率？
3. Es的查询是一个怎么的工作过程？底层的lucence介绍一下？倒排索引知道吗？
4. Es和mongodb什么区别？都在什么场景使用？

### 分布式缓存Redis

说明：书籍参考《Redis入门指南 2nd Edition》，其余参考网上博客！Redis是一个开源的，高性能的，基于键值对的缓存与存储系统，通过提供多种键值数据类型来适应不同场景下的缓存与存储需求，同时redis的诸多高层级功能使其可以胜任消息队列、任务队列等不同的角色！

主要涉及的问题：1. 持久化的问题；2.与关系型数据库的同步问题；3. 数据类型以及应用场景；4. 过期的问题； 5. Master-slave模式的数据备份（集群）；

1. redis和memcache什么区别？为什么单线程的redis比多线程的memcache的效率高？

|  |
| --- |
| 书中对二者比较的论述：P3。  有关redis存储结构的论述：1.2.1章节，有关丰富功能的介绍：1.2.3章节  《Redis实战》中的1.1.1章节有关对redis和memcache的比较！  二者的区别：<http://www.jb51.net/article/50403.htm> （比较概括性）  详细比较：<https://blog.csdn.net/session_time/article/details/52618215> （较详细）  Memcache是一个高性能，分布式内存对象缓存系统，通过在内存里维护一个统一的巨大的hash表。每个进程最大2G，当内存满后，通过LRU（最近最少使用）算法自动删除不使用的缓存，但没有考虑数据的容灾问题，重启服务，所有数据会丢失。  Redis 是一个开源（BSD许可）的，基于内存的，多数据结构存储系统。可以用作数据库、缓存和消息中间件。内置了 复制（replication），LUA脚本（Lua scripting），LRU驱动事件（LRU eviction），事务（transactions） 和不同级别的 磁盘持久化（persistence）， 并通过Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）。  有关各个数据类型的实现方式、数据存储优化，写的比较到位  额外问题：   1. Redis为什么是单线程？   因为cpu不是redis的瓶颈，最有可能是机器的内存或网络带宽！详细的解释见官方文档的FAQ：<https://redis.io/topics/faq> （Redis is single threaded. How can I exploit multiple CPU / cores?）  官方文档，有关redis压力测试性能文章：<https://redis.io/topics/benchmarks>  Redis内存优化方面的官方文档：<https://redis.io/topics/memory-optimization>   1. 一个redis实例中可以支持最多多少个key, 还有hash,list,set,sorted set最多多少元素？也是官方的FAQ（What is the maximum number of keys a single Redis instance can hold?） 2. Redis常用的操作命令？ |

1. Redis持久化的几种方式？
2. 知道redis持久化吗？有什么优缺点？具体底层如何实现？

|  |
| --- |
| Redis需要持久化的场景：P157，持久化的两种方式？以及如何使用？  7.1章节：RDB方式的原理？什么情况下会执行快照？  7.1.1使用配置规则自动执行快照的原理？7.1.2手动执行快照操作的两个命令？  7.1.3 使用flushAll命令，触发自动快照的前提？  7.1.5 快照的原理？执行fork命令时的写时复制策略？注意执行快照是新起一个进程，而不线程！！任何时候RDB文件都是完整的，RDB持久化方式的缺点?  7.2 AOF持久化的方式！这种方式的持久化原理？默认这种redis不开启这种方式。  AOF如何通过重写来优化保存文件？  AOF重写机制：AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件最后再rename)，遍历新进程的内存中数据，每条记录有一条的Set语句。重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似。  重写触发机制：Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发。  且使用BGREWRITEAOF命来来手动执行重写，见P172，这种方式启动重载的速度要比AOF慢。AOF是以纯文本的形式记录了redis执行的写命令！  7.2.3AOF同步数据到硬盘存在的问题？如何解决？使用appendfsync设置。 |

1. 为什么Redis进行RDB持久化数据时，新起一个进程而不是在原进程中起一个线程来持久化数据

|  |
| --- |
| <https://blog.csdn.net/u011784767/article/details/76824822> 对这个问题有论述。  这篇文章主要论述了两种持久化的方式，各自的优缺点？比较到位！  AOF设置不同的appendfsync代表的意义？  AOF重写机制，bgrewriteaof命令具体执行过程？  1、 同时开启两种方式优先使用AOF方式。  2、 一般来说，如果想达到足以媲美 PostgreSQL 的数据安全性，你应该同时使用两种持久化功能。  3、 如果你非常关心你的数据，但仍然可以承受数分钟以内的数据丢失，那么你可以只使用 RDB 持久化。  4、 有很多用户都只使用 AOF 持久化，但我们并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快， 除此之外，使用 RDB 还可以避免之前提到的 AOF 程序的 bug。 |

1. Redis一般就问和mysql数据库同步的问题，怎么保证一致性？事务，（实际）

|  |
| --- |
| 关于这个同步，应该是有些数据本身只存在mysql中的，但是为了提高查询速度，需要在redis中也同步一份。这是就会涉及到两个数据源的同步问题，尤其是在任何一个数据源中发生了写操作！有关redis的事务是不是也是这个问题？  Redis和MySQL数据同步及Redis使用场景：  <https://blog.csdn.net/Luomingkui1109/article/details/78403710> 场景说明比较概括性！  通过MySql自动同步刷新redis：  <https://www.cnblogs.com/zangrunqiang/p/5927154.html>  <https://www.cnblogs.com/lanbo203/p/7494587.html> 这篇应该是写的最好的 |

1. Redis有什么数据类型？都在哪些场景使用，Redis基本存储类型，使用场景

|  |
| --- |
| <http://www.cnblogs.com/mrhgw/p/6278619.html> 有关应用场景的讨论  这篇文章通俗易懂，NoSQL数据库主要用于解决哪些问题？  Redis最为常用的数据类型？Redis内部内存管理中是如何描述这些不同数据类型的？这个对每个数据类型的典型使用场景讲的很不错！！  注意：书中对每个数据类型也有详细的应用场景说明！！并且在第4章节中，对典型的高级应用做了讲解！！  一个字符串的键,允许存储的数据最大容量是512M。注意：包括INCR在内的所有redis命令都是原子操作。Redis对于键的命名最好具有可读性，便于以后的维护。  书中所讲：   1. 字符串键，使用INCR方法，可以实现文章访问量的应用，如何为每篇文章生成一个唯一的ID？见P26。 2. 散列类型，书中的例子也是为了防止序列化和反序列化，去修改某个属性。一个散列类型键可以包含至多232-1个字段！注意：除了散列类型，redis其他数据类型同样不支持数据类型的嵌套，比如集合类型的每个元素只能是字符串，不能是另一个集合或散列表！散列类型适合存储对象。并且，redis的散列类型中并不要求每个键值都拥有同样多的字段，可以自由增减！！HSET命令方便之处在于不区分插入和更新操作，注意区分set和hset命令差别，hset和hsetnx区别（说明见P35）。   如果使用散列类型存储文章，如何为文章生成唯一id？见P38。   1. 列表类型，应用场景的介绍和传统分页遇到的问题？见P40，列表类型可以存储一个有序的字符串列表，内部使用双向链表来实现。但是代价是通过索引访问元素比比较慢。应用场景：P41，如查找最新的内容，还可以作为队列（P42）使用，一个列表类型键最多容纳232-1个元素。LRange命令是最常用的之一，注意删除命令LREM的使用（P44）。应用：如何使用列表类型来实现分页展示，以及这种方法存在的问题？P45。获取列表指定索引元素，LIndex，LSet，只保留列表指定片段，LTrim，还有向列表中间插入元素LInsert。RPopLPush命令的使用场景：见P48。 2. 集合类型，集合间的运算，SDIFF,SINTER,SUNION，集合的应用场景P52，存储文章标签，还可以通过标签搜索文章。额外命令：进行集合运算并将结果存储、随机获取集合中元素、从集合中弹出一个元素，见3.6.4章节。 3. 有序集合，和列表的相同点和区别？见P57。往有序集合中增加元素，zadd key score member, 就是每个元素都是带一个分数的。ZRange获得排名在某个范围的元素列表（从小到大），而ZRevRange是从大到小；还可以获取指定分数范围的元素，ZRangeByScore，还可以跟上limit（见P60）；可以增加某个元素的分数，ZIncrby key increment member。   应用场景，实现按点击量的排序，实现P62。还可以实现按照文章发布时间来排名。  命令补充：zcard, zcount, zrem, 注意：ZRemRangeByRank和ZRemRangeByScore的删除区别，见P3.6.4章节！获取排名ZRank，计算有序集合的交集等 |

1. Redis主从复制是怎么实现的？redis的集群模式是如何实现的？redis的key是如何寻址的？

|  |
| --- |
| <https://www.cnblogs.com/hepingqingfeng/p/7263782.html>  <https://blog.csdn.net/zhangguanghui002/article/details/78524533>  上面这两篇文章关于**主从复制**介绍的都比较简略，但是表明了哪些是重点问题？然后对照书中的讲解比较到位！问题汇总：   1. 如何实现复制（从库），2. 复制的原理是啥？3. 全量复制和增量复制？4. 主从复制的策略？5. 如何建立哨兵并启动？6. 哨兵工作过程？7.哨兵的工作原理？8.在故障恢复中，怎么挑选从库作为新的主库？   **Redis主从复制：**  8.1.1如何配置主从数据库，如果一个数据库已经是别的数据库的从库，此时再执行SLAVEOF配置另一个新的库，就会变成新库的从库。还可以SLAVE NO ONE使当前库重新变为主库。  Redis复制的原理：8.1.2，比较清楚。复制初始化？从库与主库断开重连后如何同步？只要进行复制，就会进行快照。Redis的乐观复制策略？P169。就是将执行结果的命令发送给从库是异步的，主库会优先返回结果给客户端，所以就有个时间窗口造成主从数据不一致！可以设置主库至少同步了几个从库，主库才可写的：min-slaves-to-write  8.1.3图结构比较特别！8.1.5从数据库持久化的问题，如果主库崩了，如何从从库恢复到主库？注意：如果主库关闭了持久化，千万不能让主库设置自动重启的功能，否则重启后会造成从库也同步为空的问题，从库的持久化就没意义了。  8.1.6无硬盘复制：传统的主从复制中，主库在后台保存RDB快照，从库接收并载入快照文件，但是有很多缺点：P172。从2.8.18版本，redis引入了无硬盘复制，开启后，redis在与从库进行复制初始化的时候，就不会将快照内容存储到硬盘，而是直接通过网络发送给从库，避免硬盘的性能瓶颈。  8.1.7增量复制，是针对主从断开重连的情况，传统的做法是从库重连之后，从库会发送给主库一个SYNC命令，重新对主库进行一个完整的复制操作。这样比较耗时。  增量复制的基础？主库受到PSYNC命令后，会如何判断决定是否本次重连可以执行增量复制？P172，积压队列的概念？  **8.2哨兵：**  redis2.8提供了哨兵工具，来实现自动化的系统监控和故障恢复功能。主要功能：1.监控主从库是否正常运行，2. 主库故障时，自动将从库转换为主库。  建立哨兵配置文件：sentinel monitor *master-name ip redis-port quorum*  启动sentinel进程：redis-sentinel /path/to/sentinel.conf  配置哨兵监控一个系统时，只需要配置监控主库即可，会自动发现所有的从库。  工作过程，当主库挂掉，哨兵会检测到，然后会从从库中自动选一个作为新的主库，并设置其他从库的关系，同时也会保留挂掉的库，如果旧库重新启动，会自动将其作为新主库的从库。  哨兵的实现原理：8.2.3章节。一个哨兵节点可以同时监控多个redis主从系统，只需要提供多个sentinel monitor配置即可。同时多个哨兵节点也可以同时监控同一个redis主从系统。工作原理的详细解说：P178。  如果ping之后，超过指定时间没回复，认为**主观下线**，如果该库是主库，哨兵会进一步判断是否需要对其故障恢复；之后哨兵发送SENTINEL is-master-down-by-addr命令询问其他哨兵是否也认为该主库主观下线，如果达到指定数量，哨兵会认为**客观下线**，并选举领头哨兵节点对主从系统发起故障恢复。  故障恢复过程？见180。首先是，选哪个作为新的主库？然后领头哨兵发送向新主库发送SLAVEOF NO ONE，然后向其他从库发送SLAVEOF命令，最后一步将已停的旧主库更新为新主库的从库。  哨兵的部署问题：8.2.4章节。哨兵是以独立的进程方式对一个主从系统进行监控，监控好坏取决于哨兵视角是否有代表性。  **8.3集群**  有关旧版的客户端分片，预分片、哨兵和集群的讨论（8.3章节总介绍）。  集群创建的具体过程：P183。分配节点的原则：尽量保证每个主库运行在不同的ip，每个从库和主库不运行在同一个ip，保证系统的容灾能力。  节点分配好之后，会为每个主库分配插槽，分配插槽其实就是分配哪些键归哪个节点负责。  Redis-trib.rb是很好的辅助工具，本质通过执行redis命令来实现集群管理任务。  增加新节点，只需要向新节点发送命令：CLUSTER MEET ip port即可。  插槽的分配：8.3.3章节。新节点加入集群，要么使用CLUSTER REPLICATE命令复制每个主库来以从库形式运行，要么向集群申请分配插槽，来以主库形式运行。  一个集群中，所有的键被分配16384个插槽，而每个主库会负责处理其中一部分插槽。Redis中键与插槽的对应关系，见P185，redis将每个键的键名有效部分使用CRC16算法，算出散列值，然后对16384取余。有效部分的说明？  然后是将插槽与节点的对应关系，见P186，如果像某个节点分配插槽，直接在该节点上执行：CLUSTER ADDSLOTS slot1 …[slotN]；如果某个插槽之前分配过，现在迁移到新节点，首先之前那个所在的节点上执行 redis-trib.rb reshard ip:port，进行重新分片。  8.3.4操作一个键时，插槽如何找到对应的节点，如果当前节点不是目的节点，会重定向到目标节点，从而造成性能影响。Redis命令行客户端，提供了集群模式，自动支持重定向的参数-c：redis-cli –c –p port。  8.3.5集群的故障恢复。如何才能确定一个节点下线？如果一个节点下线，而这个主节点有从节点，则选一个从节点作为新的主节点，整个集群没问题；但是如果一个节点下线，并且没有一个从节点能正常负责插槽分配，则整个集群进入下线状态，无法工作。 |

1. 使用redis如何设计分布式锁？使用zookeeper可以吗？如何实现？这两种哪个效率更高？

|  |
| --- |
| Redis实现分布式锁，在官方文档里面有文章描述：<https://redis.io/topics/distlock>  <http://ifeve.com/redis-lock/> 官方文档翻译版本。  **采用单实例的正确实现：**   1. 要获得锁，要用下面这个命令： SET resource\_name my\_random\_value NX PX 30000 这个命令的作用是在只有这个key不存在的时候才会设置这个key的值（NX选项的作用），超时时间设为30000毫秒（PX选项的作用） 这个key的值设为“my\_random\_value”。这个值必须在所有获取锁请求的客户端里保持唯一。 2. 删除这个key当且仅当这个key存在而且值是我期望的那个值。也就是一个简单的判断之后再删除，防止删除已经被其他客户端拿到的锁。 3. key值的超时时间，也叫做”锁有效时间”。这个是锁的自动释放时间，也是一个客户端在其他客户端能抢占锁之前可以执行任务的时间，这个时间从获取锁的时间点开始计算。   Redlock算法：  如果一个客户端获取大多数节点锁的耗时接近甚至超过锁的最大有效时间时（就是我们为SET操作设置的TTL值），那么系统会认为这个锁是无效的同时会释放这些节点上的锁，所以我们仅仅需要考虑获取大多数节点锁的耗时小于有效时间的情况。  Java中使用jedis实现分布式锁的案例：<https://www.jianshu.com/p/c1f5d26cb1c9> （见code，很实用简单！）  使用zk实现分布式锁：<https://blog.csdn.net/peace1213/article/details/52571445> （这篇文章写得好，介绍了为什会要用分布式锁？）  Zookeeper书中还介绍了如果使用数据库自己的锁会存在的问题？见P188 |

1. Redis过期策略有哪些？LRU算法？写一下java版本的代码？

|  |
| --- |
| <https://blog.csdn.net/wangxilong1991/article/details/70172302> LRU算法的java实现（源码中有，最简单的是使用LinkedHashMap）  <https://www.cnblogs.com/WJ5888/p/4371647.html> 这里具体阐述了redis的六种过期策略，以及在配置文件中的体现！ |

1. 书中Redis的高级用法：Redis的事务，过期时间问题，排序，

|  |
| --- |
| **事务：**  Redis的事务是一组命令的集合，就是先将属于一个事务的命令发送给redis，然后再让redis依次执行这些命令。有关redis事务的介绍：4.1.1。注意：redis的错误，如果是语法错误，会在输入的时候就会报错，所有命令不会执行；但是如果是运行时的错误，redis仍然会执行正确的，不支持回滚。注意watch命令的使用：4.1.3，watch命令可以监控一个或多个键，一旦其中一个键被修改（或删除），之后的事务就不执行了。监控一直持续到EXEC命令。如果取消watch，使用unwatch命令。  **过期时间**：  使用set或getset命令为键赋值同时也会清除键的过期时间，expire会重置过期时间，其他命令不会影响键的过期时间。  过期时间应用：实现对访问频率的控制（同时注意和事务一起使用，保证不会出错）！！4.2.2章节（实现一），4.2.3（实现二，更加精细，与列表类型的键结合使用！！）  实现缓存（很实用）：为了提高网站的负载能力，常常需要将一些访问频率较高，但是对CPU或IO资源消耗较大的操作的结果缓存起来，并希望让这些缓存过段时间自动过期！注意：redis中类似既有写操作，又有别的操作，最好使用事务。  为redis设置使用的最大内存：maxmemory，maxmemory-policy  **排序（4.3.2）：**  应用场景P79（SORT tag:ruby:posts），对同一标签的文章按发布时间（这里默认按id）排序。sort命令可以对列表类型，**集合**类型和有序集合类型键进行排序（默认从小到大），并且可以完成与关系数据库中的连接查询相似的任务！对有序集合类型排序的时候，会忽略元素的分数，只针对元素自身的值进行排序！如果添加Alpha参数，还可以按字母排序。Sort还支持limit offset count。  4.3.3使用By参数，根据id对应的对象的某个属性进行排序（主要应用散列的排序），注意By的参考键可以是字符串类型或散列类型键的某个字段（格式：键名->字段名）。**案例：**P81（SORT tag:ruby:posts BY post:\*->time DESC），注意这里相当于是个关联排序了,排序的对象是tag:ruby:posts这个集合键里的元素,但是排序的条件是post:\*散列类型里的某个属性!!  Sort命令的get参数的应用场景：get不影响排序，作用是使sort命令的返回结果不再是元素自身，而是get参数中指定的键值。示例：P83。Get也是支持字符串类型和散列类型。  Store参数，保存排序后的结果，保存后键的类型是列表类型。这句命令的理解：SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title GET post:\*->time GET # STORE sort.result  注意：sort命令是redis最强大最复杂的命令之一，sort命令的时间复杂度是O(n+mlogm)，注意性能的影响，见4.3.6章节。  **消息通知：**  使用场景和传统方案的问题？别人可以订阅你的博客，如果发布了新的文章就邮件通知别人。见4.4描述  任务队列：使用redis实现任务队列，LPUSH和RPOP组合使用就可以实现队列，可以是使用BRPOP命令来优化取操作，这个命令如果没取到值会一直阻塞。  优先队列的使用场景：4.4.3描述，当发送确认邮件和发送通知邮件两种任务同时存在的时候，优先执行前者。如何实现优先的，见P89，也就是如果两个集合类型键，使用BRPOP或BLPOP，这两个集合谁在前面，就先执行谁。  发布/订阅模式（4.4.4）：模式的原理？以及PUBLISH, SUBSCRIBE命令，订阅之后的客户端就不能使用其他的命令了。注意订阅不同的返回值类型，还可以使用PSUBSCRIBE命令按照指定的规则进行订阅频道（见4.4.5），这个命令还允许重复订阅一个频道。对应按规则退订，PUNSUBSCRIBE。  **管道：**  因为客户端和redis使用TCP协议连接，二者之间的通信比较耗时，往返会存在延时，在执行多个命令，每个命令都要等待上一条执行完，而redis底层通信协议对管道提供了支持，当一组命令中的每条命令都不互相依赖，可以一次性全部发送出去。  **优化占用空间：4.6章节**  redis使用内存空间，所以要尽量优化，优化的点都有哪些？  键名和键值的优化、redis内部编码优化，redisObject结构体的定义见P94，注意redis中每种数据类型都有两种编码方式，redis的一个键值可以被多个键引用，也就是键值共享，但如果设置了maxmemory就不会使用共享对象了，每个键都要新建一个redisObject来记录。 |

1. 什么叫做分片？

分片是一种将数据划分为多个部分的方法，对数据的划分可以基于键包含的id，基于键的散列值，或者基于以上两者的某种组合。通过对数据进行分片，用户可以将数据存储到多台机器里，也可以从多台机器里获取数据，这种方法在解决某些问题时候可以获得线性级别的性能提升！

### Dubbo分布式服务框架

1. 说一下dubbo的实现过程？注册中心挂了可以继续通信吗？（可以，进行了缓存）

|  |
| --- |
| <https://blog.csdn.net/u013322876/article/details/72846054> dubbo的底层架构和实现原理  dubo支持多种协议？Dubbo协议，Hessian协议，HTTP协议，RMI协议，WebService协议，Thrift协议，Memcached协议，Redis协议。  在通信过程中，不同的服务等级一般对应着不同的服务质量，那么选择合适的协议便是一件非常重要的事情。你可以根据你应用的创建来选择。例如，使用RMI协议，一般会受到防火墙的限制，所以对于外部与内部进行通信的场景，就不要使用RMI协议，而是基于HTTP协议或者Hessian协议。  文章中对默认使用的dubbo协议介绍比较详细。包括，连接个数，方式，传输协议，传输方式，序列化，使用范围，使用场景。  第四节，服务暴露和消费的详细过程讲的比较详细！服务暴露和服务消费的两个图画反了。  <https://blog.csdn.net/chao_19/article/details/51764150> 原理浅析作为补充  rmi协议：<https://blog.csdn.net/a19881029/article/details/9465663>  关于注册中心挂了的问题：  Userbook的1.3架构章节有详细的说明。 |

1. dubbo支持哪些序列化协议，hessian 说一下？hessian的数据结构PB知道吗？为啥PB效率是最高的啊??

|  |
| --- |
| <https://blog.csdn.net/xiaojin21cen/article/details/79834222> （比较经典）  这篇文章比较详细的介绍了dubbo支持的7种协议，不是专指序列化协议，包括：  1. dubbo协议的配置，为什么消费者比提供者个数多，单一长连接？不能传大包？为什么采用异步单一长连接？（重点）  2. Rmi协议的配置方式和讲解。  3. Hession协议（重点）可以和原生Hessian服务互操作，即： 提供者用Dubbo的Hessian协议暴露服务，消费者直接用标准Hessian接口调用，或者提供方用标准Hessian暴露服务，消费方用Dubbo的Hessian协议调用。  Hession协议的约束  参数及返回值需实现Serializable接口; 参数及返回值不能自定义实现List, Map, Number, Date, Calendar等接口，只能用JDK自带的实现，因为hessian会做特殊处理，自定义实现类中的属性值都会丢失。  4. Http协议； 5. Webservice协议，6.thrift协议， 7. Memcached协议 8.redis协议  <http://www.iteye.com/topic/245238> hession序列化和反序列化的实现机制  <https://blog.csdn.net/u014401141/article/details/72475914> hession原理分析  <https://blog.csdn.net/moonpure/article/details/53175519> 这篇文章，有对**dubbo各种序列化方式**性能的比较（重点在Kryo，FST），在dubbo RPC中，同时支持多种序列化方式，例如：  dubbo序列化：阿里尚未开发成熟的高效java序列化实现，阿里不建议在生产环境使用它；  hessian2序列化：hessian是一种跨语言的高效二进制序列化方式。但这里实际不是原生的hessian2序列化，而是阿里修改过的hessian lite，它是dubbo RPC默认启用的序列化方式；  json序列化：目前有两种实现，一种是采用的阿里的fastjson库，另一种是采用dubbo中自己实现的简单json库，但其实现都不是特别成熟，而且json这种文本序列化性能一般不如上面两种二进制序列化；  java序列化：主要是采用JDK自带的Java序列化实现，性能很不理想。  最近几年，各种新的高效序列化方式层出不穷，不断刷新序列化性能的上限，最典型的包括：  专门针对Java语言的：Kryo，FST等等  跨语言的：Protostuff，ProtoBuf，Thrift，Avro，MsgPack等等 |

1. Dubbo负载均衡策略和高可用策略都有哪些啊？动态代理策略呢?

|  |
| --- |
| dubbo负载均衡策略及源码分析：  <https://www.cnblogs.com/leeSmall/p/7620467.html>  Dubbo源码分析-服务调用的动态代理和负载均衡（经典）：  <https://blog.csdn.net/manzhizhen/article/details/52606748> 主要涉及问题：   1. 为什么在Spring中我们能像注入普通本地服务JavaBean一样注入远程的Dubbo服务Bean？   AbstractProxyFactory有两个实现类：JdkProxyFactory和JavassistProxyFactory，顾名思义，一个是使用JDK的动态代理，一个是使用Javaassist来实现动态代理。如果不配置，由于ProxyFactory接口上有@SPI("javassist")注解，所以默认是使用Javassist来实现动态代理！   1. 当有多个服务提供者存在时，Dubbo是怎么做负载均衡的？   但如果在随机过程中加入权重这一属性的话，随机的优势不言而喻了，比如可以做到“预热”功能，给刚上的服务器在某段时间内分配比其他服务器更少的请求，让刚上的服务器能先“热热身”，这一点，普通轮询方式是很难做到的。没错，Dubbo默认的随机负载方式就加入了权重这一因数，权重精确到某个接口的某个方法，这也是我们所期望的，其默认“预热”时间是10分钟。  这里对每种负载均衡策略的简述比较言简意赅！！应该先看这里的表述，在分析源码。  另外，这哥们对dubbo源码进行了一系列的解析，具体可见其系列文章：  <https://blog.csdn.net/manzhizhen/article/details/52606730> |

1. 为什么要进行系统拆分啊？拆分不用dubbo可以吗？dubbo和thrift什么区别啊?
2. Dubbo服务请求失败如何处理（有容错处理）

|  |
| --- |
|  |

1. Dubbo底层原理，
2. Dubbo超时重试，超时时间设置（分服务方和调用方）

|  |
| --- |
| <https://www.cnblogs.com/binyue/p/5380322.html> Dubbo超时和重连机制 |

<https://blog.csdn.net/mingdry0304/article/details/72356497> dubbo各种标签解释

**用户手册学习：**<http://dubbo.apache.org/books/dubbo-user-book/>

|  |
| --- |
| 1.3 架构  讲解了调用关系，连通性问题，  注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表。注册中心和监控中心都是可选的，服务消费者可以直连服务提供者  服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复。  注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心  服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者  <dubbo:service/>用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心  <dubbo:application/> 应用配置 用于配置当前应用信息，不管该应用是提供者还是消费者. 建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一个消费方同时引用多个服务，就不需要关心每个服务的超时设置。如果公共配置很简单，没有多注册中心，多协议等情况，或者想多个 Spring 容器想共享配置，可以使用 dubbo.properties 作为缺省配置。Dubbo 将自动加载 classpath 根目录下的 dubbo.properties，可以通过JVM启动参数 -Ddubbo.properties.file=xxx.properties 改变缺省配置位置。  5.3 API配置， 5.4注解配置  **第六章重点！！**  6.1章节启动时检查：Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 check="true"。  可以通过 check="false" 关闭检查，比如，测试时，有些服务不关心，或者出现了循环依赖，必须有一方先启动。  另外，如果你的 Spring 容器是懒加载的，或者通过 API 编程延迟引用服务，请关闭 check，否则服务临时不可用时，会抛出异常，拿到 null 引用，如果 check="false"，总是会返回引用，当服务恢复时，能自动连上。  可以设置，关闭消费者（可以关闭某个，或者关闭全部）和注册中心的启动检查。  6.2章节集群容错  在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。具体方案可见图示。以及图中具体逻辑解释。Failover Cluster，失败自动切换，当出现失败，重试其它服务器 。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)  6.3 负载均衡。在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 random 随机调用。Dubbo的负载均衡策略有：  **随机**，按权重设置随机概率。在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。  **轮循**，按公约后的权重设置轮循比率。存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。  **最少活跃调用数，**相同活跃数的随机，活跃数指调用前后计数差。使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。  **一致性 Hash，**相同参数的请求总是发到同一提供者。当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。  算法参见：http://en.wikipedia.org/wiki/Consistent\_hashing  缺省只对第一个参数 Hash，如果要修改，请配置 <dubbo:parameter key="hash.arguments" value="0,1" />，缺省用 160 份虚拟节点，如果要修改，请配置 <dubbo:parameter key="hash.nodes" value="320" />  服务端配置：<dubbo:service interface="..." loadbalance="roundrobin" />  客户端配置：<dubbo:reference interface="..." loadbalance="roundrobin" />  还分别可以对方法级别进行配置！！  6.4 dubbo的线程模型。几种场景的分析，需要通过不同的派发策略和不同的线程池配置的组合来应对不同的场景: <dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed" threads="100" />，对具体参数的分析，不同dispatcher代表的含义。不同种类的threadpool，使用时的不同。  6.5直连提供者。在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。  配置方式，可以通过xml配置，通过启动指定jvm参数，还可以通过文件映射。为了避免复杂化线上环境，不要在线上使用这个功能，只应在测试阶段使用。  6.6只订阅。为方便开发测试，经常会在线下共用一个所有服务可用的注册中心，这时，如果一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行。  可以让服务提供者开发方，只订阅服务(开发的服务可能依赖其它服务)，而不注册正在开发的服务，**通过直连**测试正在开发的服务。  6.7 只注册。如果有两个镜像环境，两个注册中心，有一个服务只在其中一个注册中心有部署，另一个注册中心还没来得及部署，而两个注册中心的其它应用都需要依赖此服务。这个时候，可以让服务提供者方只注册服务到另一注册中心，而不从另一注册中心订阅服务。  6.8静态服务。有时候希望人工管理服务提供者的上线和下线，此时需将注册中心标识为非动态管理模式。<dubbo:registry address="10.20.141.150:9090" dynamic="false" />，服务提供者初次注册时为禁用状态，需人工启用。断线时，将不会被自动删除，需人工禁用。  6.9多协议。Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议。具体配置见网页。  6.10多注册中心。Dubbo 支持同一服务向多注册中心同时注册，或者不同服务分别注册到不同的注册中心上去，甚至可以同时引用注册在不同注册中心上的同名服务。另外，注册中心是支持自定义扩展的 。1. 如何将服务注册到不同的注册中心？2. 如何调用不同注册中心的服务，见配置？  6.11 服务分组。当一个接口有多种实现时，可以用 group 区分。服务方和消费方都需要配置一个group属性。  6.12多版本。当一个接口实现，出现不兼容升级时，**可以用版本号过渡（好处就是能平滑过渡升级）**，版本号不同的服务相互间不引用。可以按照以下的步骤进行版本迁移：  1. 在低压力时间段，先升级一半提供者为新版本  2. 再将所有消费者升级为新版本  3. 然后将剩下的一半提供者升级为新版本  这个非常具有实用价值，具体的配置见网页。  6.13分组聚合：按组合并返回结果 1，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现聚合菜单项。这个聚合配置场景也很丰富？指定合并策略，缺省根据返回值类型自动匹配，如果同一类型有两个合并器时，需指定合并器的名称。注意：已知的merger类型有很多，具体可以看源码。com.alibaba.dubbo.rpc.cluster.Merger包下。  6.14参数验证：参数标注示例，分组验证示例，关联验证示例，参数验证示例。这个有具体的demo，在源码中  6.15 结果缓存。结果缓存 用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量 。缓存类型：  lru 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。  threadlocal 当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。  jcache 与 JSR107 集成，可以桥接各种缓存实现。  6.16 使用泛化调用（使用的场景和意义？）。泛化接口调用方式主要用于客户端没有 API 接口及模型类元的情况，参数及返回值中的所有 POJO 均用 Map 表示，通常用于框架集成，比如：实现一个通用的服务测试框架，可通过 GenericService 调用所有服务实现。  1. 使用spring使用泛化调用，主要是设置generic属性为true；2. 通过API方式使用泛化调用。// 用com.alibaba.dubbo.rpc.service.GenericService可以替代所有接口引用。使用demo参考源码：com.alibaba.dubbo.config.validation.ValidationTest#testGenericValidation  6.17 泛化实现。泛接口实现方式主要用于服务器端没有API接口及模型类元的情况，参数及返回值中的所有POJO均用Map表示，通常用于框架集成，比如：实现一个通用的远程服务Mock框架，可通过实现GenericService接口处理所有服务请求。  6.18 回声测试。回声测试用于检测服务是否可用，回声测试按照正常请求流程执行，能够测试整个调用是否通畅，可用于监控。所有服务自动实现 EchoService 接口，只需将任意服务引用强制转型为 EchoService，即可使用。  6.19 上下文信息。上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册 中的对应URL参数一列。  RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。  这个比较实用，可以获取一些服务端和消费端的一些基本信息，比方是否为服务方？调用方地址？当前url配置信息？  6.20 隐式参数。可以通过 RpcContext 上的 setAttachment 和 getAttachment 在服务消费方和提供方之间进行参数的隐式传递。在服务消费方端设置隐式参数，在服务提供方端获取隐式参数。  6.21异步调用。基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。这个比较重要，类似于java的多线程。实用asyn，sent,return属性设置，代码中也有个future用来接收执行结果。  6.22 本地调用。本地调用使用了 injvm 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。从 2.2.0 开始，每个服务默认都会在本地暴露。在引用服务的时候，默认优先引用本地服务。如果希望引用远程服务可以使用一下配置强制引用远程服务。<dubbo:reference ... scope="remote" />  6.23 参数回调。参数回调方式与调用本地 callback 或 listener 相同，只需要在 Spring 的配置文件中声明哪个参数是 callback 类型即可。Dubbo 将基于长连接生成反向代理，这样就可以从服务器端调用客户端逻辑 。可以参考 dubbo 项目中的示例代码。这个示例，如果修改为直连方式，好像不行。  6.24事件通知。在调用之前、调用之后、出现异常时，会触发 oninvoke、onreturn、onthrow 三个事件，可以配置当事件发生时，通知哪个类的哪个方法。首先进行实现，具体方法，然后在配置中指定，只需要在服务消费者中指定即可。见网页代码。  6.25本地存根。远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub，然后把Stub暴露给用户，Stub 可以决定要不要去调 Proxy。代码中，其实就是使用装饰者模式对原服务进行代理，在调用原服务之前，增加了参数校验等逻辑，然后就是在配置中设置stub属性  6.26 本地伪装。本地伪装通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。在消费方配置mock属性。可以自定义mock实现，// 你可以伪造容错数据，此方法只在出现RpcException时被执行。  Mock 是 Stub 的一个子集，便于服务提供方在客户端执行容错逻辑，因经常需要在出现 RpcException (比如网络失败，超时等)时进行容错，而在出现业务异常(比如登录用户名密码错误)时不需要容错，如果用 Stub，可能就需要捕获并依赖 RpcException类，而用 Mock 就可以不依赖 RpcException，因为它的约定就是只有出现 RpcException 时才执行。 ↩  6.27 延迟暴露，如果你的服务需要预热时间，比如初始化缓存，等待相关资源就位等，可以使用 delay 进行延迟暴露。可以指定延迟的时间，指定延迟到spring初始化完成之后，再暴露服务。  文中对spring2.x初始化死锁问题的分析，这里会涉及到bean初始化同步问题（在看spring源码中，已有了解，singletonObjects和beanDefinitionMap）！一个是请求线程的applicationContext.getBean()调用，另一个是spring初始化线程，不需判断，直接同步初始化。  规避方法！服务实现类中不要进行getBean的调用，全部使用IOC；如果实在要调 getBean()，可以将 Dubbo 的配置放在 Spring 的最后加载；如果不想依赖配置顺序，可以使用 <dubbo:provider delay=”-1”/>，使Dubbo在Spring 容器初始化完后，再暴露服务。  6.28并发控制！executes属性可以控制服务端并发执行的线程数量，也可以精确到方法级别；actives属性可以控制客户端并发执行（或占用连接请求数）数量，可以在服务端或客户端设置，客户端优先级高。配置服务的客户端的loadbalance属性为leastactive，此 Loadbalance 会调用并发数最小的 Provider（Consumer端并发数），同样都可以配置。  6.29连接控制。限制服务器端接收的连接数，使用accepts属性，可以在<dubbo:provider>或<dubbo:protocol>设置。限制客户端连接数，使用connections属性设置，都可以设置。  6.30 延迟连接。延迟连接用于减少长连接数。当有调用发起时，再创建长连接，<dubbo:protocol name="dubbo" lazy="true" />，该配置只对使用长连接的 dubbo 协议生效。  6.31粘滞连接。 粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者挂了，再连另一台。粘滞连接将自动开启延迟连接，以减少长连接数。<dubbo:protocol name="dubbo" sticky="true" />  6.32令牌验证。通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者。整个过程图示比较清楚。  可以全局设置开启令牌验证，使用token属性，可在<dubbo:provider，<dubbo:service上进行设置，也可以在协议级别设置，<dubbo:protocol，如果是true，随机生成令牌，如果是固定值，就相当于密码。  6.33 路由规则。路由规则决定一次 dubbo 服务调用的目标服务器，分为条件路由规则和脚本路由规则，并且支持可扩展。向注册中心写入路由规则的操作通常由监控中心或治理中心的页面完成。例如：  registry.register(URL.valueOf("condition://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" + URL.encode("host = 10.20.153.10 **=>** host = 10.20.153.11") + "));黄色的才是路由规则内容。  对路由规则参数的详细解析？基于条件表达式的路由规则，支持表达式参数适配，这个介绍的比较详细？可以实现各种特殊场景！！  示例比较有代表性：1. 排除预发布机，2. 白名单， 3. 黑名单，4. 服务寄宿在应用上，只暴露一部分的机器，防止整个集群挂掉；5. 为重要应用提供额外的机器；6.读写分离；7.前后台分离；8.隔离不同机房网段；9. 提供者与消费者部署在同集群内，本机只访问本机的服务。  脚本路由规则 4 支持 JDK 脚本引擎的所有脚本，比如：javascript, jruby, groovy 等，通过 type=javascript 参数设置脚本类型，缺省为 javascript。比方说可以使用js方法来定义具体的基于脚本引擎的路由规则。  6.34配置规则。  向注册中心写入动态配置覆盖规则。该功能通常由监控中心或治理中心的页面完成。  例如：  "override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&timeout=1000"  代表性示例：1. 禁用提供者：(通常用于临时踢除某台提供者机器，相似的，禁止消费者访问请使用路由规则)（disbaled=true）；2.调整权重，&weight=200；3. 调整负载均衡策略：(缺省负载均衡策略为 random)，如&loadbalance=leastactive；4. 服务降级：(通常用于临时屏蔽某个出错的非关键服务) &mock=force:return+null。  6.35 服务降级。  可以通过服务降级功能，临时屏蔽某个出错的非关键服务，并定义降级后的返回策略。向注册中心写入动态配置覆盖规则，就是上面列出的第四个应用。  mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。  还可以改为 mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回 null值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。  6.36优雅停机。  Dubbo 是通过JDK 的ShutdownHook来完成优雅停机的，所以如果用户使用 kill -9 PID等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID时，才会执行。优雅停机的原理？分为服务方和消费方：  服务方：停止时，先标记为不接收新请求，新请求过来时直接报错，让客户端重试其它机器，然后，检测线程池中的线程是否正在运行，如果有，等待所有线程执行完成，除非超时，则强制关闭。  消费方：停止时，不再发起新的调用请求，所有新的调用在客户端即报错。然后，检测有没有请求的响应还没有返回，等待响应返回，除非超时，则强制关闭。  使用dubbo.service.shutdown.wait=15000来设置，默认是10s。使用tomcat等容器部署的場景，建议通过扩展ContextListener等自行调用以下代码实现优雅停机：  ProtocolConfig.destroyAll();  6.37 主机绑定 。  缺省主机 IP 查找顺序：通过LocalHost.getLocalHost() 获取本机地址；如果是 127.\* 等 loopback 地址，则扫描各网卡，获取网卡 IP。注册的地址如果获取不正确，比如需要注册公网地址，具体操作？缺省主机端口与协议相关，各个协议对应的默认端口？这里最后介绍了配置dubbo端口的方法，这也正是很多demo中使用的指定dubbo端口号？  6.38 日志适配。  自2.2.1开始，dubbo 开始内置 log4j、slf4j、jcl、jdk 这些日志框架的适配，也可以通过以下方式显示配置日志输出策略：  1.启动的时候使用java命令指定；2. 在dubbo.properties 中指定；3. 在 dubbo.xml 中配置， 这些都是通过dubbo.application.logger全局属性来设置。  6.39 访问日志  如果你想记录每一次请求信息，可开启访问日志，类似于apache的访问日志。注意：此日志量比较大，请注意磁盘容量。将访问日志输出到当前应用的log4j日志：  <dubbo:protocol accesslog="true" />，将访问日志输出到指定文件：  <dubbo:protocol accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />  6.40 服务容器  服务容器是一个 standalone 的启动程序，因为后台服务不需要Tomcat 或 JBoss 等 Web 容器的功能，如果硬要用 Web 容器去加载服务提供方，增加复杂性，也浪费资源。  服务容器只是一个简单的 Main 方法，并加载一个简单的 Spring 容器，用于暴露服务。  服务容器的加载内容可以扩展，内置了 spring, jetty, log4j 等加载，可通过容器扩展点进行扩展。配置配在 java 命令的 -D 参数或者 dubbo.properties 中。  比如：可以指定spring配置加载路径，启动jetty容器，配置log4j。这些应用讲的很详细？容器启动默认只加载spring容器，可以通过main参数（或jvm参数，或通过 classpath 下的 dubbo.properties 配置传入要加载的容器）传入要加载的容器？使用的是dubbo.container参数。  6.41 ReferenceConfig 缓存  ReferenceConfig 实例很重，封装了与注册中心的连接以及与提供者的连接，需要缓存。否则重复生成 ReferenceConfig 可能造成性能问题并且会有内存和连接泄漏。在 API 方式编程时，容易忽略此问题。因此，自2.4.0 版本开始，dubbo 提供了简单的工具类 ReferenceConfigCache用于缓存 ReferenceConfig 实例。使用方式？  缺省ReferenceConfigCache 把相同服务 Group、接口、版本的 ReferenceConfig 认为是相同，缓存一份。即以服务 Group、接口、版本为缓存的 Key。可以修改这个策略？  6.42分布式事务  分布式事务基于 JTA/XA 规范实现，但是这个功能在dubbo里还暂未实现！！两阶段提交示意图？  6.43 线程栈自动dump。  当业务线程池满时，我们需要知道线程都在等待哪些资源、条件，以找到系统的瓶颈点或异常点。dubbo通过Jstack自动导出线程堆栈来保留现场，方便排查问题（保持了jvm的对线程快照的处理方法）。  默认策略: 导出路径，user.home标识的用户主目录；导出间隔，最短间隔允许每隔10分钟导出一次。可以使用dubbo.application.dump.directory属性来指定导出路径。  6.44 通信层框架Netty  dubbo 2.5.6版本新增了对netty4通信模块的支持，启用方式如下：  provider端：<dubbo:protocol server="netty4" />，或<dubbo:provider server="netty4" />  consumer端：<dubbo:consumer client="netty4" />  如果provider端不同协议使用了不同的通信层框架，则分别设置；而对应如果消费端同时调用多种协议的服务，需要分别设置，如下：  <dubbo:consumer client="netty"><dubbo:reference /></dubbo:consumer>  <dubbo:consumer client="netty4"><dubbo:reference /></dubbo:consumer>  补充：   1. 所有配置最终都将转换为URL表示，并由服务提供方生成，经注册中心传递给消费方，各属性对应 URL 的参数，参见配置项一览表中的 "对应URL参数" 列。 2. 注意：只有 group，interface，version 是服务的匹配条件，三者决定是不是同一个服务，其它配置项均为调优和治理参数。 |

### Zookeeper研究

《从paxos到zookeeper》

1. 重连机制会不会造成错误
2. zookeeper是什么

|  |
| --- |
| Apache的顶级项目，为分布式应用提供了高效且可靠的分布式协调服务，提供了如：统一命名服务、配置管理、分布式锁等分布式基础服务。解决一致性方面，没有直接采用paxos算法，而是采用一种成为ZAB的一致性协议。  Zookeeper需要掌握的重点，其实已经在书中做了说明？Zookeeper可以保证的分布式一致性的特性有？zk的设计目标（附带重点的概括）？见P60。  Zookeeper的数据模型见7.1.1章节。Zk工作原理见7.3章节。**7.1章节非常重要！**  Zookeeper的基本概念：4.1.3。包括集群角色、会话（session）、数据节点（Znode）、版本、Watcher（事件监听器）、ACL（access control lists）策略、  节点分类：1 . 机器节点、数据节点；2. 数据节点又分为持久节点、临时节点！见P63。  在解决分布式数据一致性上，除了zk之外，还没有一个成熟稳定且被大规模应用的解决方案。  ZAB协议核心是定义了对于那些会改变zk服务器数据状态的事务请求的处理方式，具体处理方式见P66。  ZAB协议的两种基本模式**（重点）**：崩溃模式和消息广播，详解见P66，以及后面的具体讲解。  Leader选举算法要求：  ZAB协议需要确保那些已经在leader服务器上提交的事务最终被所有服务器都提交，但是需要丢弃那些只在leader服务器上被提出的事务。具体的案例见P68.  针对leader选举算法要求，如果能保证新选举出来的leader服务器拥有集群中所有机器最高编号（即ZXID最大）的事务proposal即可。  新选举出leader后，如何进行数据同步（包含两方面：1. 对正常情况数据同步2. 对需要丢弃的事务处理，还包括ZXID的实现方案）？见P70。  JavaAPI，zookeeper构造方法参数详解，见P92。使用API创建数据节点的方法详解，见P96；同步接口创建和异步接口创建的区别，以及异步创建的回调方法解析，见P99；读取节点，getChildren()参数的具体说明，见P100。更新数据接口，setData()参数说明，见109。Exists()接口详解见P115。  注意，更新数据接口中，有个version版本的参数，这个参数的意义？见P109。保证更新操作的原子性，刚好也印证了之前看到的关于如何通过版本信息来保证原子操作的解决方案。注意版本号，如果是-1，代表的含义？见P111。  注意：watcher通知是一次性的，即一旦触发一次通知后，该watcher就失效了，因此客户端需要反复注册watcher。  关于zk的会话和心跳检测，在7.4章节讲解。Zk客户端和服务端会话的建立是一个异步过程，当会话真正创建完毕后，zk服务端会向客户端发送一个事件通知。  权限控制：1. 获取数据节点内容的权限控制， 2. 删除节点权限控制，这里比较特殊，删除操作的权限控制范围只在子节点。  开源zk客户端工具zkClient的使用介绍，见5.4.1，IZkChildListener监听的使用说明，见P127。 |

1. Zookeeper有哪些作用? zk都可以干什么？

|  |
| --- |
| 6.1 章节，zk的典型应用场景。  Zk是一个典型的发布/订阅模式的分布式数据管理和协调框架，可以使用它来进行分布式数据的发布与订阅，另一方面，通过对zk中丰富的数据节点类型交叉使用，配合watcher事件通知机制，可以很方便构建一系列分布式应用中都会涉及的核心功能，如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、master选举、分布式锁和分布式队列等。  6.1.1数据发布与订阅  数据发布订阅系统的定义？两种设计模式（push和pull）？zk是两种模式的结合？具体的实现逻辑？一般全局配置信息的特性？对配置信息普通处理方式？机器规模变大，配置信息变更频繁时，需要采用更为分布式化解决方案？使用zk进行配置信息集中管理的方案？  6.1.2 负载均衡  分布式系统具有对等性，为了保证高可用，通常采用副本的方式来对数据和服务进行部署。而对消费者而言，需要在这些对等的服务提供方中选一个来执行相关业务逻辑。  基于zk实现动态的DNS方案，P167。主要包括：域名配置、域名解析、域名变更， DDNS中，域名解析过程都由每一个应用自己负责，通常应用首先从域名节点上获取一份ip地址和端口的配置，自行解析，同时还会在域名节点上注册一个数据变更watcher监听，以便及时受到域名变更通知。  进一步自动化的DNS服务（P169）包括：域名注册、域名解析、域名探测。  P169这个架构图中，将那些目标ip地址和端口抽象为服务提供者，而那些需要使用域名解析的客户端被抽象为服务的消费者。  6.1.3 章节命名服务的定义？比如分布式服务框架的服务地址列表。  如何使用zk实现一套分布式全局唯一ID的分配机制（重点）？使用UUID来生成全局唯一ID的缺陷？见P171。详细步骤见P172，主要是利用一个特性：通过调用zk节点创建的API接口可以创建一个顺序节点，并且在API返回值中会返回这个节点的完整名字。  6.1.4分布式协调与通知（开篇的概念很好！）。引入一个协调者，将分布式协调的职责从应用中分离出来，大大减少系统之间的耦合性，显著提高系统的可扩展性！  应用场景：Mysql数据复制总线（mysql\_replicator），概念？整体结构图？见P173。Zk负责分布式协调工作，根据功能将数据复制组件划分为三个子模块：core, server, monitor。主要过程包括：任务注册、任务热备份、记录执行状态、控制台协调、冷备切换。  热备份和冷备份的概念？见P176。冷热备份对比，P177。  一种通用的分布式系统机器间通信方式。绝大多数分布式系统，机器间通信无外乎心跳检测、工作进度汇报、系统调度这三种类型。如何使用zk来实现分布式机器间心跳检测？工作进度汇报？系统调度应用，见P179。使用zk实现分布式通信，不仅省去大量底层网络通信和协议上重复设计，还大大降低了系统之间的耦合。其实主要就是基于对zk节点各种事件的监听。  6.1.5 集群管理，包括集群监控和控制两块。传统基于agent的分布式集群管理的缺点？zk实现集群管理，主要基于两大特性（watcher监听和临时节点在会话失效时自动清除）？P180。  Zk实现分布式集群管理的两大典型例子（比较实用的场景）：分布式日志收集系统和在线云主机管理，两大例子详解？如何快速、合理、动态为每个收集器机器分配对应的日志源机器，这个是最大的挑战。Zk的解决方案？见P181  Zk实现集群机器存活性监控系统，见P184。除了能监控主机的上下线，还能监控主机的运行状态信息。  6.1.6 Master选举的应用场景。典型的场景系统架构？  比较典型的就是，前端需要后台从一系列海量数据处理中计算得到一个数据，这通常是比较好肥I/O和CPU资源的，所以最好是只让其中一台机器进行计算处理，计算出结果之后，在共享给整个集群的其他所有客户端机器，这样可以大大减少重复劳动。使用常见关系型数据库的主键特性实现master选举？见P187。  如何使用zk来实现？基于zk的API创建节点时，无法重复创建一个已经存在的数据节点特性，可以利用这个特性在分布式环境中进行master选举。详解见P187。  综上，其实如果仅仅是想实现master选举，只需要有一个能够保证数据唯一性的组件即可，但是如果希望能快速进行集群master动态选举，基于zk是一个不错的思路。  6.1.7 zk实现分布式锁  应用场景：如果依赖关系型数据库的固有排他性来实现不同进程之间的互斥，那么就会更加加大数据库操作在大型分布式系统的性能瓶颈。  Zk实现，主要分为排它锁和共享锁两种！！  排它锁：通过zk上的数据节点来表示一个锁，获取锁和释放锁的原理？（P189）创建临时子节点（确定的一个临时子节点）成功的客户端获取到了锁，而没有创建成功子节点的客户端在节点上注册一个子节点变更的watcher监听。  共享锁：概念？定义锁？P190。共享锁，是在zk创建临时的顺序子节点，并且还分读请求还是写请求？如何判断读写顺序？实现过程流程图？如果节点变更向比当前大的所有节点都发送，会造成zk性能问题以及很多节点接收无用的通知问题，这个就是羊群效应。因此需要进一步改进分布式锁的实现，实际就是精确化需要监听子节点变更事件的范围，见P193。如果集群规模不大，不用考虑精确化通知机器范围，而如果达到一定规模之后，需要考虑。  6.1.8 分布式队列。分布式队列分两大类：FIFO，还有等到队列元素集聚之后才统一安排执行的barrier模型。  使用zk实现FIFO，具体步骤讲解见P195，设计思路？如何确定执行顺序？  使用zk实现Barrier分布式屏障。特指系统之间一个协调条件，规定了一个队列元素必须都聚集齐后才能统一进行安排，否则一直等待！！这往往应用在大规模分布式并行计算的场景里。具体解析，设计思想？创建完节点之后，如何确定执行顺序？见P196。 |

1. Zk原理知道吗？ paxos算法知道吗？说一下原理和实现？
2. Zk如何实现分布式锁？

|  |
| --- |
| <https://www.cnblogs.com/liuyang0/p/6800538.html> 这篇文章比较好，最后也有与redis实现分布式锁的对比简述。 |

### RabbitMQ分布式消息队列

**主要涉及的问题**有：不重复消费消息问题、消息丢失问题、消息持久化问题、数据一致性问题、基于MQ的多任务处理问题！！

1. 为什么使用消息队列，消息队列有什么优缺点？
2. 如何保证消息队列的高可用？如何保证消息不被重复消费？

|  |
| --- |
| 使用了rabbitmq作为消息队列，利用rabbitmq的ack机制来确认消息的可靠性。 但是rabbitmq本身是没有绝对的消息顺序机制的，单个queue在多消费者下不能保证其先后顺序。 另外ack的机制会触发消息重复消费的，需要我们在设计上避免该问题。   1. 关于消息的重复执行   触发消息重复执行的条件会是很苛刻的！在大多数场景下不会触发该条件！！！一般出在任务超时，或者没有及时返回状态，引起任务重新入队列，重新消费！在rabbtimq里连接的断开也会触发消息重新入队列。  消费任务类型最好要支持幂等性，这样的好处：任务执行多少次都没关系，顶多消耗一些性能！ 如果不支持幂等，比如发送信息？那么需要构建一个map来记录任务的执行情况！ 不仅仅是成功和失败，还要有心跳！！！这个map在消费端实现就可以了！！！ 这里会出现一个问题，有两个消费者 c1, c2，一个任务有可能被c1消费，如果再来一次，被c2执行？那么如何得知任务的情况？任务派发！任务做成hash，固定消费者！坚决不要想方设法在mq扩展这个future。  一句话，要不保证消息幂等性，要不就用map记录任务状态.   1. 关于消息的绝对顺序执行   我们遇到的大多数场景都不需要消息的有序的，如果对于消息顺序敏感，那么我们这里给出的方法是：消息体通过hash分派到队列里，每个队列对应一个消费者，多分拆队列。  为什么要这么设计？同一组的任务会被分配到同一个队列里，每个队列只能有一个worker来消费，这样避免了同一个队列多个消费者消费时，乱序的可能！t1, t2 两个任务，t1虽然被c1先pop了，但是有可能c2先把 t2 任务给完成了。  一句话，主动去分配队列，单个消费者。   1. 基于Rabbitmq的**多任务处理框架 （java中是否可以用线程池实现？）**   这里提一下，我们用rabbitmq实现了一个颇为复杂的架构，节省了太多的mq连接及消耗。通过python pika gevent实现的，因py有gil，所以使用多进程来跑多核，进程之间不使用共享变量，而是用队列来传递ack信号。这里实现的场景是用pika消费rabbitmq，然后把获取到的任务提丢到队列，另一个进程去消费该任务，然后触发ack。 |

1. Kafka, activemq, rabbitmq, rocketmq都有什么优缺点？发送数据如何处理？如果丢失数据，如何排查问题？（实际）

|  |
| --- |
| <https://www.cnblogs.com/tianqing/p/6701534.html> 消息队列的基本概念（主要提出消息的几个重要问题）  <https://www.cnblogs.com/luxiaoxun/p/3918054.html> RabbitMQ的几种典型使用场景（**非常好！**），还主要介绍了rm里面的基本概念（详细），AMQP协议，不同使用场景包括：  1.单发单接收 2. 单发多接收 3. 发布订阅模式 4. Routing (按路线发送接收) 5. Topics (按topic发送接收)  <https://blog.csdn.net/u010310183/article/details/51683141> activeMq的常见问题（也适用于rabbit, <https://yq.aliyun.com/articles/42160> ）  <https://blog.csdn.net/lsblsb/article/details/53286878> activeMq简单学习 |

1. 如果让你写一个消息队列，该如何架构设计，说说思路，
2. 京东电话面试问：mq不同系统调用数据一致性的问题，如何保证数据一致性？（**待解决，没懂！）**

|  |
| --- |
| <https://segmentfault.com/a/1190000011479826>消息最终一致性方案（比较详细）  <https://blog.csdn.net/u011424653/article/details/79749426> 简述rm自带的哪些机制可以解决一致性问题  <http://blog.51cto.com/4925054/2096781> 消息最终一致性解决方案之RabbitMQ实现 |

1. 消息中间件如何解决**消息丢失的问题**

|  |
| --- |
| <http://www.txtlxg.com/228/meifage/article/details/71699151> 这里对常见的消息丢失问题博客做了汇总。  rabbitMq保证消息不丢失的机制：  <https://blog.csdn.net/qbian/article/details/70198066?utm_source=itdadao&utm_medium=referral> 其实就是书上的内容  <https://blog.csdn.net/is_zhoufeng/article/details/10022859> 这篇介绍了，rm中比较典型的几个问题，思路比较简单易懂。包括不被重复消费。  <https://blog.csdn.net/yeweiouyang/article/details/74943278> 这篇文章有个很不错的架构图（结合db的使用），有个很切实的丢失消息的**经典案例**：  本例中消费者主进程将持续监听MQ，一旦MQ有消息将会拉取，随后从进程池中启动子进程来处理消息，但是从进程池启动子进程的过程并不一定成功（若当前进程池没有空闲子进程），而主进程不管任何情况下都给MQ发送ACK状态码，从而MQ将未处理的消息移除掉，导致消息丢失。  并在第三节分析了这种架构的缺点，并给出一个解决方案，但是这个新的解决方案，又会造成消息重复消费的问题：例如，用户查询一半，停止查询，（各种原因导致业务没执行完成，则消息也不会发送ack确认），文中给出的解决方案是使用一个DB来记录消息执行状态，消费者每次执行前查询消息的状态。  <http://www.cnblogs.com/littleatp/p/6087856.html> ：这篇文章讲解了“重复ACK导致消息丢失”的案例！并分析了内部的工作机制：  rabbitmq 为每一个channel维护了一个delivery tag的计数器，这里采用正向自增，新消息投递时自增，当消息响应时自减；在连续收发的场景中，由于消息发送的间隔较短，部分消息因 consumer的重复确认被rabbitmq 当做已处理而丢弃。 |

1. **rabbitMq的系列学习博客**

|  |
| --- |
| <http://www.cnblogs.com/4----/p/6518801.html> （一）单个生产者，和单个或多个消费者  如果发送消息的速率非常快，多个消息消费者就不按照顺序进行消息的消费了。  <http://www.cnblogs.com/4----/p/6526033.html> （二）防止消息丢失   1. 首先文章中模拟了消息的丢失，通过线程等待中停止进程，可以实现。刚开始consumer使用自动确认，然后在consumer接收到消息之后，但是还没执行完成时候，停止进程，这样就会造成消息丢失。 2. 然后，又模拟了通过消息确认机制来防止丢失消息，关闭consumer的自动确认，设置在消息确认之前，不能接收其他消息。其实可以通过两个App，直接模拟两个独立的consumer，这样在一个consumer接收一个消息之后，但是还没确认之前，关闭此consumer进程，会发现这个消息自动又被另一个consumer接收并处理。这样就防止了丢失消息。 3. RabbitMQ只有在收到消费者确认后，才会从内存中删除消息，如果消费者忘了确认（更多情况是因为代码问题没有执行到确认的代码），将会导致内存泄漏 4. 消息生产者和消费者，对于同一个消息持久化设置，必须要相同，否则会报错   <http://www.cnblogs.com/4----/p/6549865.html> 消息的交换（三）   1. 正确的概念是，生产者会把消息发送给RabbitMQ的交换中心（Exchange），Exchange的一侧是生产者，另一侧则是一个或多个队列，由Exchange决定一条消息的生命周期--发送给某些队列，或者直接丢弃掉。这个概念在官方文档中被称作RabbitMQ消息模型的核心思想（core idea） 2. RabbitMq中有四种exchange。文章中模拟了一个广播给多个消费者的案例，每个消费者都能接收到消息，进行自己的处理。   <http://www.cnblogs.com/4----/p/6590459.html> 消息交换（四）  这里主要是在上篇文章的基础上，做了改进，sender修改exchange为direct，并指定不同的routingKey；对应的consumer也修改exchange为direct类型，并通过设置channel.bind，来指定绑定接收具体的routingKey的消息，这样就可以在不同的场景中接收不同的消息。  <http://www.cnblogs.com/4----/p/6593486.html> 消息交换（五）  进一步改进需求：考虑另一个问题，我们的日志系统不仅要分级别级别（error，info）记录日志，还需要通过发送日志的系统来匹配，比如说有一个“核心”系统，它发出的所有级别日志，都需要记录到硬盘，其他系统只需要把error级别的日志记录到硬盘。  本文重点学习topic exchange。  我们测试发现，topic中的通配符，只有在Queue绑定的时候才能起到通配符的作用，如果在发布消息的时候使用通配符，将作为普通的字符处理，发送的routingKey=a.\* 并不能把消息发送到routingKey=a.b的Queue上，a.#同理，也不能把消息发送到routingKey=a.b.c的Queue上。  <http://www.cnblogs.com/4----/p/6668467.html> 与spring的集成   1. 使用java编码配置 2. 使用xml配置文件集成 |

**《RabbitMq实战-高效部署分布式消息队列》一书学习**

|  |
| --- |
| 2.1 消费者和生产者  Rm是消息投递服务，在应用程序和服务器之间扮演着路由器角色，当应用程序连接到rm时，他必须做个决定：我是在发送还是在接收？  消息包含两部分：有效载荷和标签。Rm会根据标签把消息发送给感兴趣的接收方，这种通信方式是一种“发后即忘”。从生产者到消费者的消息流，见P37。  消费者连接到代理服务器上，并订阅到队列上。具体的过程描述？什么是信道？为什么需要信道，而不是直接通过TCP连接发送AMQP命令？使用消息进行多线程任务调度，使用信道可以各自独立，互不干扰。分析见P38，还有很形象的图示。信道是建立在真实的TCP连接内的虚拟连接。  Rm实现任务实例分析，见P39。有了AMQP，可以灵活地使用多个信道来满足应用程序的需求，而不会有众多的TCP连接的开销。总体来说，消息通信特别是AMQP，可以被当作加强版的传输层。将rm看作是软件的路由器。  2.2 队列  队列的概念和工作原理？原理图？消费者通过哪两种方式从特定队列中接收消息？（basic.consume和basic.get命令）见P40。队列中的消息发送给消费者的不同情况分析，见P41. 通过basic.ack命令显式地发哦是那个确认，或者订阅队列的时候，将auto\_ack参数设置为true。  如果接受到一条消息，但是在确认之前从rabbit断开连接，或从队列上取消订阅，rm会认为这条消息没分发，会重新分发给下一个消费者；另一方面，如果应用程序忘记确认消息，rm将不会给该消费者发送更多的消息。也可以利用这点，对于处理消息非常耗时的应用，可以延迟确认，防止过载；如果想要明确拒绝，而不是确认收到消息？两种处理方式，使用basic.reject命令，并设置requeue为false在消息格式错误的应用场景。具体分析见P42。注意：这里的**死信队列（将requeue设置为false则被拒绝的消息存入死信队列）**，可以方便检测未送达的消息。  消费者和生产者都能通过queue.declare来创建队列。如何创建队列？见P42. 指定队列名称，如果不指定会随机分配一个名称。Queue.declare()方法参数具体解释。如果尝试声明一个已经存在的队列会怎样？如何检测队列是否存在？该由生产者还是消费者来创建所需队列的讨论（主要看是否能接受丢失消息）！队列的作用？见P43.  2.3 **交换器和绑定**  消息是如何到达队列的？路由键（routingKey）？P44。服务器会根据路由键将消息从交换器路由到队列，但如何处理投递到多个队列的情况呢？不同类型的交换器发挥了作用，有四种：direct（默认），fanout，topic、headers，四种交换器重点讲解见P45。  注意：服务器必须实现direct类型的交换器，是包含**一个空字符串名称**的默认交换器，当声明一个队列时，它会自动绑定到默认交换器，并以队列名称作为路由键。  Fanout交换器的使用场景？主要就是发一个消息，不同的消费者能做不同的事，也就是简单的广播；topic交换器的使用场景？使来自不同源头的消息能到达同一个队列，如日志处理，可以在队列绑定到交换器的时候，使用通配符（使用规则）见P47。P48总结不错！  2.3 虚拟主机和隔离  每一个rm服务器都能创建虚拟消息服务器，称为虚拟主机（vhost）。Vhost本质上是一个mini版的rm服务器，还有自己的权限控制，vhost之于rm，就像虚拟机之于物理机一样。各个实例间逻辑上分离。Vhost是amqp概念的基础，连接时需要进行指定，默认vhost:”/”。  Vhost不仅消除了为基础架构中每一层运行一个rm服务器的需要，还避免了为每一层创建不同的集群。如何创建vhost？如何删除？见P50。通过vhost保障了队列和交换器的安全。   * 1. 持久化和策略   如果rm崩溃了或重启了，如何确保关键信息不丢失(核心问题，重点看)？  首先是如何持久化？具体讲解见P51。  默认rm无法幸免于服务器重启，原因是每个队列和交换器的durable属性默认是false（仅仅设置为true是远远不够的）。如果要将消息从rabbit崩溃中恢复必须如何设置？rm确保持久化消息能从服务器重启中恢复的方式是，将他们写入磁盘上一个持久化日志文件。具体见P51。  但是不能为所有的消息都启用持久化消息通信，这样会极大地降低性能。并且持久化消息在rm内建集群环境下工作并不好。  什么时候应该使用持久化？具体分析！最终确保消息安全到达都取决于你的策略。另一个和持久化相关的概念是AMQP事务！有关事务发挥作用场景的说明。P52  AMQP事务：信道设置成事务模式后的具体执行过程？代价很大，会使生产者程序产生同步！更好的方案来保证消息投递：发送方确认模式！信道设置成发送方确认模式，具体执行过程？这种最大好处是异步！轻量级，对rm服务器性能影响可以忽略，这两种模式的具体讲解，见P53。  2.6 一条消息的生命周期  2.7 使用发送方确认模式来确认投递  一条信道只能被单个线程使用，可以确保信道上发布的消息时连续的。   * 1. 用户和权限控制   Rm也是先创建用户，然后赋予权限。原理展示图见P71。Rm权限系统有个比较好的地方在于，单个用户可以跨越多个vhost进行授权！用户连接到rm主机的应用程序可以设置成不同级别的权限！用户是访问控制的基本单元。  创建用户命令：./rabbitmqctl add\_user username password  删除用户：./rabbitmqctl delete\_user username，注意删除用户时，任何引用该用户的访问控制条目都会从rm权限数据库中自动删除。  查看当前rm服务器上有哪些用户：./rabbitmqctl list\_users  修改密码：./rabbitmqctl change\_password username newPassword  3.2.2 章节Rabbit的权限系统。读、写、配置权限的说明？每一条访问控制条目的组成部分？P72。注意：访问控制条目是无法跨越vhost的。为用户设置权限的具体操作见P74。  删除权限控制：./rabbitmqctl clear\_permissions –p vhost username  匹配消息通信发后即忘模式的两种一般类型的任务（应用场景）：1. 批处理；2. 通知，见P96。  **具体实例应用的设计：**  1. 发送告警设计，见4.2.1章节；  2. 并行处理设计，见4.2.2章节；  3.使用rabbit来设计rpc解决方案，见4.3.1章节，**P115讨论。**  核心问题：使用消息发回应答？怎样阻止其他客户端读到消息？。 注意使用rm实现rpc和普通的消息机制不同的地方，见P117、P118的讨论部分！  <https://blog.csdn.net/u013256816/article/details/55218595> （java实现rpc）  <https://blog.csdn.net/xiaoxian8023/article/details/48826857>详细的解析rpc实现（案例和上面这篇一样）  <https://blog.csdn.net/nullo_cns/article/details/48395837> spring与rabbit整合实现rpc，包含使用rabbit实现rpc的优点，初级实现，和针对原始实现的优化！！  <https://blog.csdn.net/younger_z/article/details/53243990> RabbitMQ与java、Spring结合实例详细讲解。**重点内容：**  这篇文章提到，ActiveMQ曝光率最高，但是可能会丢消息。临时队列和持久化队列、自动删除的队列、FIFO处理机制、  Exchange类似于数据通信网络中的交换机，提供消息路由策略。rabbitmq中，producer不是通过信道直接将消息发送给queue，而是先发送给Exchange。一个Exchange可以和多个Queue进行绑定，producer在传递消息的时候，会传递一个ROUTING\_KEY，Exchange会根据这个ROUTING\_KEY按照特定的路由算法，将消息路由给指定的queue。和Queue一样，Exchange也可设置为持久化，临时或者自动删除。  Binding，所谓绑定就是将一个特定的 Exchange 和一个特定的 Queue 绑定起来。Exchange 和Queue的绑定可以是多对多的关系。 |

## 补充

1. Spring中进行bean的依赖注入有几种方式？什么是aop，aop的作用？
2. 比较string, stringBuffer, stringBuilder的区别
3. Jsp跳转redirect和forward的用法区别
4. 对象克隆的几种方法？
5. 解释内存中的栈，堆，静态区的用法 ✔
6. Jdbc操作数据库的步骤
7. 静态变量和实例变量的区别
8. Overload和override的区别，重载的方法能否根据返回类型进行区分
9. Java中如何跳出当前的多层嵌套循环
10. 列出至少5个运行时异常
11. Oracle的什么特性能实现主键自增？mysql呢
12. 面向对象设计的3个基本特征？
13. 自己定义一个类加载器的实现?

|  |
| --- |
| <https://www.cnblogs.com/xrq730/p/4847337.html> |

1. Autowired和resource的区别?
2. Log4j在不同框架下的使用？

<https://blog.csdn.net/sinat_30185177/article/details/73550377>

**补充2（来自面试实战）：**

1. 那会问我开启的多个线程怎么保证都执行完毕了？
2. **一致性Hash原理和常见的应用场景？实现方法？**

|  |
| --- |
| 大白话解析一致性Hash，非常好！<http://www.zsythink.net/archives/1182>  包括：常见应用场景，分布式缓存问题？场景描述，普通Hash暴露出来的问题（也就是一致性Hash要解决的问题）？一致性hash的基本概念？缓存服务器在Hash环上映射，之后缓存的对象也要在Hash环上进行映射，映射的算法思想？缓存对象到底存在哪个服务器的算法思想？一致性hash的优点？Hash环的偏斜问题？虚拟节点？ <https://www.cnblogs.com/lpfuture/p/5796398.html>  这篇文章在最后提出如何如何创建虚拟节点的一个思想，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器ip或主机名的后面增加编号来实现。例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算 “Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点！  <https://www.cnblogs.com/xrq730/p/5186728.html> （java对一致性hash的实现）  **这篇文章非常好！**主要介绍了使用不同数据结构实现一致性hash的复杂度的比较！java具体实现不带虚拟节点和带虚拟节点的一致性Hash代码！！！非常实用。还包含对虚拟节点与真实节点配比图。 |

1. javaGC常用算法，如何进行调优，例如一个方法每次调用都产生大量临时对象，怎样优化？衡量GC算法的主要目标是什么。
2. 设计一个无线app的接口交互协议，出入参数请用类表示。接口根据用户id 、性别、历史浏览记录等信息返回个性化推荐的内容，app 上使用瀑布流方式加载展示内容。返回内容可能有多种形式，包括帖子(含有帖子id,帖子简介文本，图片列表、发帖人呢称和头像，跳转链接)，人工活动(含有活动id,活动简介，活动图片列表，跳转链接)，图片导航(含有图片导航id，一个大图片，跳转链接)

## Netty研究

1. 知道netty吗？netty可以干嘛呀？NIO,BIO,AIO 都是什么啊？有什么区别啊?

|  |
| --- |
| <https://blog.csdn.net/haoyuyang/article/details/53243785> 基本介绍  基于Tcp/IP协议传输的数据，在缓存中接收后，不是按照包的队列组织，而是字节的队列组织，这就很容易导致，碎片化的读取，所以需要需要额外的碎片化整理技术来解决这个问题。  Netty是完全基于NIO实现的，所以整个Netty都是异步的。网络应用程序通常需要有较高的可扩展性，无论是Netty还是其他的基于Java Nio的框架，都会提供可扩展性的解决方案。Netty中一个关键组成部分是它的异步特性！   1. NIO的通信步骤？ 2. Netty通信的步骤？ 3. 有关服务端与客户端的三次握手以及TCP缓冲区设置解释：   对于ChannelOption.SO\_BACKLOG的解释：  服务器端TCP内核维护有两个队列，我们称之为A、B队列。客户端向服务器端connect时，会发送带有**SYN**标志的包（第一次握手），服务器端接收到客户端发送的SYN时，向客户端发送SYN ACK确认（第二次握手），此时TCP内核模块把客户端连接加入到A队列中，然后服务器接收到客户端发送的ACK时（第三次握手），TCP内核模块把客户端连接从A队列移动到B队列，连接完成，应用程序的accept会返回。也就是说accept  从B队列中取出完成了三次握手的连接。A队列和B队列的长度之和就是backlog。当A、B队列的长度之和大于ChannelOption.SO\_BACKLOG时，新的连接将会被TCP内核拒绝。所以，如果backlog过小，可能会出现accept速度跟不上，A、B队列满了，导致新的客户端无法连接。要注意的是，backlog对程序支持的连接数并无影响，backlog影响的只是还没有被accept取出的连接   1. Netty中有关tcp粘包、拆包的问题？以及解决方法？都是上篇博客的内容   <http://www.importnew.com/15656.html> Netty实现原理浅析  Netty是典型的Reactor模型结构，Reactor模式的典型实现？见其中的图示，总共有三种，哪种是netty的默认实现？  Netty中的Boss类充当mainReactor，NioWorker类充当subReactor（默认 NioWorker的个数是Runtime.getRuntime().availableProcessors()）。在处理新来的请求 时，NioWorker读完已收到的数据到ChannelBuffer中，之后触发ChannelPipeline中的ChannelHandler流。  Netty是事件驱动的，其通过ChannelEvent来确定事件流的方向。一个ChannelEvent是依附于Channel的 ChannelPipeline来处理，并由ChannelPipeline调用ChannelHandler来做具体的处理。  Netty 在事件处理上，是通过ChannelPipeline来控制事件流，通过调用注册其上的一系列ChannelHandler来处理事件，这也是典型的拦截 器模式。  各种解码器的介绍？  Netty和Mina是Java世界非常知名的通讯框架。它们都出自同一个作者，Mina诞生略早，属于Apache基金会，而Netty开始在Jboss名下  一个系列的文章：<http://www.importnew.com/17647.html>   1. 基于NIO编写网络程序，你需要做什么(以Server端TCP连接为例，这里我们使用Reactor模型)？ 2. 对netty事件驱动机制的解析。 |

## SQL研究

1. 如何实现insert or update的插入或更新操作？

<https://blog.csdn.net/baidu_28514449/article/details/52783075> 有关insert or update比较好的案例，非常实用，在一些应用场景中。注意Mybatis的写法。

注意，这种判断duplicate key是建立在唯一索引的基础上，例如主键id，或者是手动加上的联合唯一索引也行！

1. Mybatis中使用choose…when进行多条件判断的时候，要注意如果是判断字符串类型的某个字段是否==”1”，这种数字字符串的时候，注意在xml的写法，是会区分’’和””的。

|  |
| --- |
| 例如：  <if test="merBusType!=null and merBusType!=''">  <choose>  <when test="merBusType == '\_UNDEFINED'">  and a.mer\_bus\_type is null  </when>  <when test='merBusType == "1"'>  and a.mer\_bus\_type is not null  </when>  <otherwise>  and a.mer\_bus\_type=#{merBusType}  </otherwise>  </choose>  </if>  这里面，如果是==”1”会区分单引号还是双引号，但是如果是==’ \_UNDEFINED’是不区分的。 |

1. Oracle的索引问题

<https://www.cnblogs.com/djcsch2001/articles/1823459.html>

1. Oracle基本建表语句

<https://blog.csdn.net/zx_highgo/article/details/52638266>