问题分类整理

Problem分类整理

JVM

1 Java的内存分区

线程私有的内容：程序计数器（指向下一条指令），Java栈_虚拟机栈（局部变量，对象引用）

线程共享：Java堆（对象），方法区-Non_Heap（类信息，常量String,final，静态变量）

本地方法栈

2 Java对象的回收方式，回收算法。

回收方式：
(1).引用计数算法：对象引用计数，为0回收，没有解决对象循环依赖问题（Java没有使用）
(2).根搜索算法（一虚三方）
1. 虚拟机栈(栈桢中的本地变量表)中的引用的对象 ；
2. 方法区中的类静态属性引用的对象 ；
3. 方法区中的常量引用的对象 ；
4. 本地方法栈中JNI的引用的对象；
Java中常用的垃圾收集算法（标记判断对象是否应该回收的方法是G-Root算法）：
(1).标记-清除算法：（内存碎片）年轻代
(2).复制算法：（可用内存减半）年轻代
(3).标记-整理算法：（向一端移动来进行整理，适合于对象存活率高情况） 老年代
(4).分代收集算法：（新生，老年代）

3 CMS和G1了解么，CMS解决什么问题，说一下回收的过程。

CMS（老年代的回收问题）（concurrent mark sweep并发标记清除）
预标记（swt）—并发标记—预清理—再标记（+CMSScavengeBeforeRemark swt）—并发清理(+CMSFullGCsBeforeCompaction)—重置CMS状态
停顿两次的原因可建题5
G1（模糊了分代的概念，而是采用了分区的方式，每个分区当中可以包含年轻代和老年代的混合对象）

4 Java中的垃圾回收器

年轻代：Serial（复制算法）ParNew（并发Serial）Parallel Scavenge
老年代：Serial Old（标记整理算法）Parallel Old （并发Serial）CMS

5 CMS回收停顿了几次，为什么要停顿两次。

初始标记阶段和再次标记阶段会产生Stop the world 停顿。
GC线程标记好了一个对象的时候，此时我们程序的线程又将该对象重新加入了"关系网"中，当执行二次标记的时候，该对象也没有重写finalize()方法，因此回收的时候就会回收这个不该回收的对象。 虚拟机的解决方法就是在一些特定指令位置设置一些"安全点"，当程序运行到这些"安全点"的时候就会暂停所有当前运行的线程（Stop The World 所以叫STW），暂停后再找到"GC Roots"进行关系的组建，进而执行标记和清除。
安全点位置：

- 1、循环的末尾
- 2、方法临返回前 / 调用方法的call指令后
- 3、可能抛异常的位置

6 Java栈什么时候会发生内存溢出，Java堆

所以我们可以理解为栈溢出就是方法执行是创建的栈帧超过了栈的深度。那么最有可能的就是方法递归调用产生这种结果。
heap space表示堆空间，堆中主要存储的是对象。如果不断的new对象则会导致堆中的空间溢出.

7 类加载器结构吧（**）（引导BootStrap rt.jar，扩展Extention ext/*.jar, 系统Systemclasspath， 自定义Custom，符合双亲委派模型）

Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。
引导类加载器（bootstrap class loader）：
它用来加载 Java 的核心库(jre/lib/rt.jar)，是用原生C++代码来实现的，并不继承自java.lang.ClassLoader。
加载扩展类和应用程序类加载器，并指定他们的父类加载器，在java中获取不到。
扩展类加载器（extensions class loader）：
它用来加载 Java 的扩展库(jre/ext/*.jar)。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
系统类加载器（system class loader）：
它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
自定义类加载器（custom class loader）：
除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求。

8 CAS怎么实现原子操作的？（三个操作数，内存地址，预期值，新值）

Java中的CAS操作正是利用了处理器提供的CMPXCHG指令
在CAS中有三个操作数：分别是内存地址（在Java中可以简单理解为变量的内存地址，用V表示）、旧的预期值（用A表示）和新值（用B表示）。CAS指令执行时，当且仅当V符合旧的预期值A时，处理器才会用新值B更新V的值，否则他就不执行更新，但无论是否更新了V的值，都会返回V的旧值。
cas操作存在ABA问题：（取出内存中某时刻的数据（由用户完成），在下一时刻比较并替换（由CPU完成，该操作是原子的）。这个时间差中，会导致数据的变化。例如，A线程从内存地址v取出X,B线程也从内存地址中取出X，B将X改成X1再改成X，在这之后A线程进行比较修改，发现预期值和内存地址内容相同，随机进行修改，在这当中，B线程的修改行为对于A线程而言是完全无感知的）
解决方案：AtomicStampedReference（在修改同时添加类似时间戳的状态戳，做到版本控制的效果，当且仅当状态戳和内存值同时满足期望，才允许其发生更改）| AtomicMarkableReference时间戳的概念换成true/false的状态变化信息

9 说一下GC吧，什么时候进行Full GC呢？（大对象无法分配，系统调用，空间不足）

FULL GC:老年代和年轻代没有足够的连续内存空间分配给大对象；System.gc()调用；永久代空间不足；老年代空间不足

10 虚拟机类加载机制，双亲委派模型，以及为什么要实现双亲委派模型

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。
双亲委派模型的工作过程是：

- 如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成。
- 每一个层次的类加载器都是如此。因此，所有的加载请求最终都应该传送到顶层的启动类加载器中。
- 只有当父加载器反馈自己无法完成这个加载请求时（搜索范围中没有找到所需的类），子加载器才会尝试自己去加载。

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在虚拟机中的唯一性(保证了类加载的唯一性),确保了程序类加载过程的安全性

11 新生代分为几个区？使用什么算法进行垃圾回收？为什么使用这个算法？

eden和Survivor1和Survivor2（默认比例8:1，经验值，存活期端的对象占用比例接近80%）
复制算法，不会产生内存碎片（两个survivor的原因），年轻代存活期短
https://blog.csdn.net/paul_wei2008/article/details/55259579

12 GC可达性分析中哪些算是GC ROOT？（一虚三方）

虚拟机栈中引用的对象、方法区类静态属性引用的对象、方法区常量池引用的对象、本地方法栈JNI引用的对象

13 你在项目中一般怎么调优JVM的呢？

JVM堆栈大小&GC方案选取
jps jstat（分区GC情况） jmap（堆内存）jstack（线程堆栈）jmc（性能分析）

14 JVM如何加载字节码文件

加载：
(1) 通过一个类的全限定名来获取定义此类的二进制字节流
(2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构
(3) 在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口
验证：
二进制字节流解析的数据结构是否符合class对象规范
准备：
准备阶段是正式为类变量分配并设置类变量初始值的阶段，这些内存都将在方法区中进行分配（变量都是类变量，实例变量在堆）
解析：

符号引用替换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用进行初始化：
这个阶段主要是对类变量初始化，是执行类构造器的过程。
换句话说，只对static修饰的变量或语句进行初始化。
如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。
如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。
父类的<clinit>()方法先与子类执行，父类的static语句，父类的非static语句块和构造方法，接下来是子类的非static语句块和构造方法
父类静态（代码块，变量赋值二者按顺序执行）->子类静态->父类构造代码块->父类构造方法->子类构造代码块->子类构造方法

15 JVM GC，GC算法。

minorGC&FullGc. GC算法见第3题

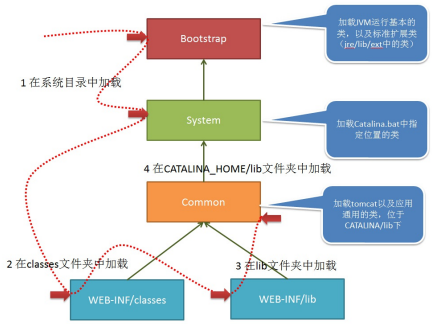
16 Java运行时数据区

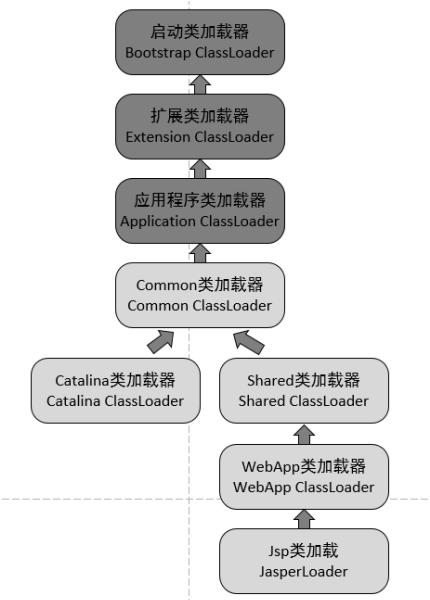
同JVM组成

17 类加载器如何卸载字节码（ClassLoader->Class->Instance都没有被引用）

类加载器没有被引用
类对象没有被引用
没有该类的实例对象存在
满足这三个条件，虚拟机会自动卸载该类。

18 Tomcat的类加载器了解





1 使用bootstrap引导类加载器加载（jre/lib/ext）
2 使用system系统类加载器加载(bootstrap.jar tomcat启动类 catalina.sh中制定)
3 使用应用类加载器在WEB-INF/classes中加载（工程代码编译类）
4 使用应用类加载器在WEB-INF/lib中加载（工程引用类）
5 使用common类加载器在CATALINA_HOME/lib中加载（CATALINA_HOME/lib）
Attention:最后两行违背了双亲加载机制
违反双亲委派机制原因（tomcat服务器可以同时部署多个工程，不同工程依赖不同，需要单独进行类的加载，参考maven依赖的不同版本）

19 Java虚拟机中，数据类型可以分为哪几类

1、整数：int,short,byte,long 2、浮点型：float,double 3、字符：char 4、布尔：boolean
除了以上8种java虚拟机还有一种：索引数据类型

20 为什么不把基本类型放堆中，局部变量的基本类型存储在栈中

基本类型和对象的引用都是存放在栈中，基本类型大小可知，基本类型的处理方式是统一的
单纯的基本类型存储在栈，对象内部的基本类型存储在堆
常量池的操作基本对象减少了重复的开销

Java基础

锁

1 synchronized的Jvm实现

借助与对象关联的monitor对象的monitortry指令和monitorexit指令实现的

2 synchronized锁升级的过程，说了偏向锁到轻量级锁再到重量级锁，然后问我它们分别是怎么实现的，解决的是哪些问题，什么时候会发生锁升级。

偏向锁（线程级别的偏心的锁，单一线程持续持有锁对象的时候使用，对象头，栈帧添加threadId）
线程少，持有竞争资源时间短的情况线程自旋不阻塞，轻量级锁
自旋时间过长，自旋次数超限制，同时存在第三线程竞争的情况的时候，升级为重量级别锁

3 Java中的锁有什么？synchronized与Lock的区别？公平锁和非公平底层怎么实现？AQS原理？

- 公平锁/非公平锁 （ReEntrantLock可以通过AQS实现公平锁）：是否按照申请锁的顺序分配锁资源
- 可重入锁（synchronized&ReEntrantLock）线程可以重复进入已经持有锁资源的方法块 https://www.cnblogs.com/dj3839/p/6580765.html（写得好）
- 独享锁（互斥锁）/共享锁（读写锁）（ReEntrantLock&Synchronied是独享锁，ReadWriteLock是共享锁通过AQS实现）
- 乐观锁/悲观锁（是态度，不是确切的指某一种锁，乐观无锁编程=CAS，悲观全加上锁）
- 分段锁（是设计模式，比如ConcurrentHashMap通过Segments分别加锁的方式来实现分段锁）
- 偏向锁/轻量级锁/重量级锁（针对Synchronized而言）
- 自旋锁（采用循环的方式尝试获得锁，消耗CPU资源，不产生上下文切换，不需要转入内核态，就是死循环的方式来等待）

AQS：抽象队列同步器；
包含一个int类型的State标识和CLH双向队列；
公平锁&非公平锁的实现：通过CLH队列先进先出的特点来实现公平锁，非公平锁通过notify()竞争
独享锁&共享锁 ：独享锁略，共享锁通过CAS的tryAcquireShare(int state)的方式来实现，state标识共享的资源个数，可以参与共享的线程数

4 volatile关键字解决了什么问题，实现原理是什么(缓存一致性协议，刷主存)

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。
2）禁止进行指令重排序。（volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作）
原子性（完成或者失败）/可见性（多线程下资源的共享可见性，包含正确性）/有序性（语句执行顺序），volatile保证了可见性，不能保证原子性，Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。x=x+1并不是原子性质的操作，分为两个原子操作，分别是读取x的值以及对x重新赋值，当x阻塞在读取阶段的时候，即便存在其他的线程修改了x的值，x的值也不会得到更新。

5 synchronized和lock的异同

1.synchronized（Java内置关键字，在jvm层面），Lock是个java类；
2.synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；
3.synchronized会自动释放锁，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；
4.用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；
5.synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平AQS的CLH（两者皆可）
6.Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。
7.sychronied独享锁，lock可以做共享锁（AQS的state）

语法

1 拆箱装箱的原理

装箱就是自动将基本数据类型转换为包装器类型；拆箱就是自动将包装器类型转换为基本数据类型。装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。注意这里的valueOf方法，对于Integer对象而言在-128-127范围内的整形会存储到IntegerCache当中。

2 1.8还采用了红黑树，讲讲红黑树的特性，为什么人家一定要用红黑树而不是AVL、B树之类的？（**）

相对AVL旋转成本低，相对于B树，红黑树操作全在内存，数据库索引使用B+树，B树多路查找树，红黑二叉树

3 深克隆和浅克隆（克隆对象的引用的克隆）

浅克隆是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。
深克隆不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象。

4 用不同的方法实现单例模式

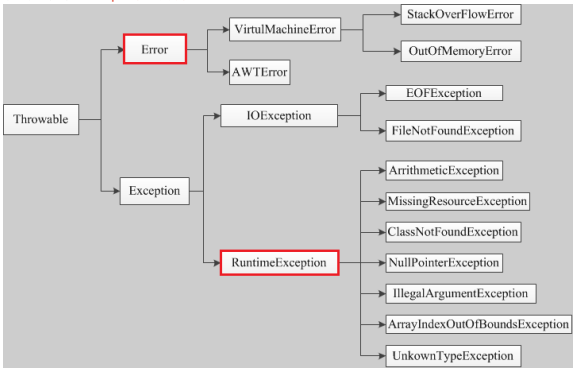
懒汉（线程安全【双重检验/静态内部类】/不安全）/饿汉（静态变量/静态代码块）

5 throwable、Error、Exception、RuntimeException 区别和联系各是什么？

Throwable类是Java语言中所有错误和异常的超类。（Java虚拟机抛出，Java throw语句抛出）
Error是Throwable的子类，合理的应用程序不应该尝试捕获的严重问题
Exception类及其子类是Throwable的一种子类，它表示合理的应用程序可能想要捕获的条件
RuntimeException是在Java虚拟机的正常操作期间可以抛出的那些异常的超类。RuntimeException及其子类是未经检查的异常

6 什么是检查异常，不受检查异常，运行时异常？并分别举例说明。

检查异常：必须处理的异常，ioexception filenotfound throw或者try catch
不受检查异常：空指针异常，数据越界业务处理 ERROR&RuntimeException
运行时异常：RuntimeException类及其子类异常
非运行时异常：Exception类及其子类



7 try、catch、finally语句块的执行顺序

（1）try return表达式 finaly return结果返回值
（2）try catch exception逻辑 exception中return逻辑 finally return结果返回值

8 对Java中Runtime的了解

JVM的运行时环境&饿汉加载的单例模式&多次重载的excute方法（返回process进程，创建子进程执行linux命令）&借助waitfor方法达到子进程同步的效果

9 stringbuffer和stringbuilder的区别

1、StringBuffer 与 StringBuilder 中的方法和功能完全是等价的，
2、只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。
3、在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全
而StringBuffer则每次都需要判断锁，效率相对更低

10 stringbuffer的扩容

扩容算法：使用append()方法在字符串后面追加东西的时候，如果长度超过了该字符串存储空间大小了就需要进行扩容：
默认2倍+2，若不足以实际所需要的容量为主
构建新的存储空间更大的字符串，将旧的复制过去

11 String和stringbuffer的区别

（1）String:是对象不是原始类型.为不可变对象,一旦被创建,就不能修改它的值.对于已经存在的String对象的修改都是重新创建一个新的对象,然后把新的值保存进去.String 是final类,即不能被继承
（2）StringBuffer:是一个可变对象,修改的时不用重新建立对象。它只能通过构造函数来建立对象被建立以后,在内存中就会分配内存空间,并初始保存一个null.向StringBuffer中赋值的时候可以通过它的append方法.

12 Collections.sort底层排序方式？

Collections.sort->Arrays.sort->ComparableTimSort.sort,然后才到sort方法和它的定义,排序主体binarySort的方法，这是排序方法的实现，是通过调用Object的CompareTo进行比较的。
https://blog.csdn.net/m0_37630602/article/details/68945357

13 java事件机制包括哪三个部分（一种设计模式，监听者模式）

事件监听器、事件源、事件。
1、事件。一般继承自java.util.EventObject类，封装了事件源对象及跟事件相关的信息。
2、事件监听器。实现java.util.EventListener接口,注册在事件源上,当事件源的属性或状态改变时,取得相应的监听器调用其内部的回调方法。
3、事件源。事件发生的地方，因为事件监听器要注册在事件源上,所以事件源类中应该要有盛装监听器的容器(List,Set等等)。

反射&注解

1 Java反射原理，注解原理？

反射的原理:JAVA语言编译之后会生成一个.class文件，反射就是通过字节码文件找到某一个类、类中的方法以及属性等。反射的实现主要借助以下四个类：Class：类的对象 Constructor：类的构造方法 Field：类中的属性对象 Method：类中的方法对象
用一个词就可以描述注解，那就是元数据，即一种描述数据的数据。
@Documented –注解是否将包含在JavaDoc中
@Retention –什么时候使用该注解
@Target –注解用于什么地方
@Inherited – 是否允许子类继承该注解
注解本质上是继承了 Annotation 接口的接口，而当你通过反射，也就是我们这里的 getAnnotation 方法去获取一个注解类实例的时候，其实 JDK 是通过动态代理机制生成一个实现我们注解（接口）的代理类。
https://www.cnblogs.com/yangming1996/p/9295168.html

线程 &并发

线程

1 Java线程的状态(join同步化，阻塞调用join方法的主线程)

new,waiting(wait,join,park)
timedwaiting(sleep,join,park)
runnable(start)
blocked(synchronied&wait)
terminated
NEW：新建状态，线程对象已经创建，但尚未启动
RUNNABLE:就绪状态，可运行状态，调用了线程的start方法，已经在java虚拟机中执行，等待获取操作系统资源如CPU，操作系统调度运行。
BLOCKED:堵塞状态。线程等待锁的状态，等待获取锁进入同步块/方法或调用wait后重新进入需要竞争锁
WAITING：等待状态。等待另一个线程以执行特定的操作。调用以下方法进入等待状态。 Object.wait(), Thread.join(),LockSupport.park
TIMED_WAITING: 线程等待一段时间。调用带参数的Thread.sleep, objct.wait,Thread.join，LockSupport.parkNanos,LockSupport.parkUntil
TERMINATED:进程结束状态。

2 进程和线程的区别，进程间如何通讯，线程间如何通讯

计算机分配资源的最小单元（进程）管道PIPE，命名管道，Socket通信，共享内存通信，信号量
任务调度的基本单元（线程）同步Sychronied，wait-notify，join，管道通信，while轮询
https://www.cnblogs.com/xh0102/p/5710074.html

3 什么是threadlocal

ThreadLocal：线程局部的变量；
每个Thread的对象都有一个ThreadLocalMap，当创建一个ThreadLocal的时候，就会将该ThreadLocal对象添加到该Map中，其中键就是ThreadLocal，值可以是任意类型。包含set()和get()方法。

180 同一个线程连续start两次之后的结果是什么样子的

Java的线程是不允许启动两次的，第二次调用必然会抛出IllegalThreadStateException，这是一种运行时异常，多次调用start被认为是编程错误。

线程池

1 线程池有哪些参数？分别有什么用？如果任务数超过的核心线程数，会发生什么？阻塞队列大小是多少？

corePoolSize：核心线程数 一直存活的线程，初始阶段线程数<核心线程，新任务创建新线程，不复用已有空闲线程
当线程数小于核心线程数时，即使有线程空闲，线程池也会优先创建新线程处理
queueCapacity：任务队列容量（阻塞队列）
maxPoolSize：最大线程数，拒绝处理任务的临界点
keepAliveTime：线程空闲时间，除了核心线程外其他线程最大空闲时间
allowCoreThreadTimeout：允许核心线程超时，上限keepAliveTime
rejectedExecutionHandler：任务拒绝处理器
两种情况会拒绝处理任务：

当线程数已经达到maxPoolSize，切队列已满，会拒绝新任务
当线程池被调用shutdown()后，新任务到达
ThreadPoolExecutor类有几个内部实现类来处理这类情况：
AbortPolicy 丢弃任务，抛运行时异常 CallerRunsPolicy 执行任务 DiscardPolicy 忽视，什么都不会发生 DiscardOldestPolicy 从队列中踢出最先进入队列
实现RejectedExecutionHandler接口，可自定义处理器

2 怎样设置线程池的大小（线程等待时间能被其他线程充分利用）

最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）* CPU数目
最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）* CPU数目
要想合理的配置线程池的大小，首先得分析任务的特性，可以从以下几个角度分析：
1.任务的性质：CPU密集型任务、IO密集型任务、混合型任务。
2.任务的优先级：高、中、低。
3.任务的执行时间：长、中、短。
4.任务的依赖性：是否依赖其他系统资源，如数据库连接等。

3 Java线程池的实现原理

1、判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。
2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
3、判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。
实现原理：
内部状态：
借助AtomicInteger变量ctl（标记线程数量和线程池当前状态）
1、RUNNING：-1 << COUNT_BITS，即高3位为111，该状态的线程池会接收新任务，并处理阻塞队列中的任务；
2、SHUTDOWN： 0 << COUNT_BITS，即高3位为000，该状态的线程池不会接收新任务，但会处理阻塞队列中的任务；
3、STOP ： 1 << COUNT_BITS，即高3位为001，该状态的线程不会接收新任务，也不会处理阻塞队列中的任务，而且会中断正在运行的任务；
4、TIDYING ： 2 << COUNT_BITS，即高3位为010；
5、TERMINATED： 3 << COUNT_BITS，即高3位为011；
任务提交：
没有返回值使用Excutor.excute();
需要返回值使用ExcutorService.submit();
线程池的工作线程通过Woker类实现，在ReentrantLock锁的保证下，把Woker实例插入到HashSet后
http://baijiahao.baidu.com/s?id=1580981900587942071&wfr=spider&for=pc(有点意思)

4 数据库连接池实现原理

连接池基本的思想是在系统初始化的时候，将数据库连接作为对象存储在内存中，当用户需要访问数据库时，并非建立一个新的连接，而是从连接池中取出一个已建立的空闲连接对象。使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中，以供下一个请求访问使用。而连接的建立、断开都由连接池自身来管理。同时，还可以通过设置连接池的参数来控制连接池中的初始连接数、连接的上下限数以及每个连接的最大使用次数、最大空闲时间等等。也可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

5 sleep和wait的区别

wait释放锁资源，sleep释放cpu资源

6 如何实现要给简单的线程池

1） 准备一个任务容器
2.）一次性启动10个 消费者线程
3） 刚开始任务容器是空的，所以线程都wait在上面。
4） 直到一个外部线程往这个任务容器中扔了一个"任务"，就会有一个消费者线程被唤醒notify
5） 这个消费者线程取出"任务"，并且执行这个任务，执行完毕后，继续等待下一次任务的到来。
6） 如果短时间内，有较多的任务加入，那么就会有多个线程被唤醒，去执行这些任务。

I 任务容器List<Runnable> tasks;

II 自定义线程类run方法

```
{
 while(true) {
  sychronized(tasks) {
   while(Empty(tasks)) {
   tasks.await();
   Runnable runnable = tasks.removeLast();
   tasks.notifyAll();
   }
  }
 runnable.run();
 }
}
```

tasks的add和初始化方法也需要sychronied方法修饰，先初始化空跑线程再添加任务tasks中

Java同步工具类

1 countdowmlatch，cyclebarrier，semaphore

countdownlatch：线程计数器，提高线程的并发性，并在主线程中确保需要等待的线程任务全部完成
cyclebarrier：类似与countdownlatch，countdown的点设置在线程执行任务的过程中，当所有线程都阻塞在barrier点的时候，才继续并发的往下执行。
semaphore：类似与一种令牌桶的机制，线程在执行任务前需要在令牌桶中获取足量的semaphore才能继续执行，否则阻塞等待

2 AtomicInteger，为什么要用CAS而不是synchronized？

AtomicInteger通过CAS原子方法操作避免了i++非原子操作操作导致的数据一致性问题。
在竞争条件下会阻塞等待资源，如果允许竞争不到资源返回失败，就可以使用cas减少阻塞时间。

```
public class NonBlock {

        private static volatile NonBlock nonBlock;

        private static AtomicBoolean atomicBoolean = new AtomicBoolean(false);

        public static NonBlock getInstance() {
                if (nonBlock == null) {
                        if (atomicBoolean.compareAndSet(false, true)) {
                                nonBlock = new NonBlock();
                        }
                }
                return nonBlock;
        }
}
```

3 并发juc了解么（java.util.concurrent），有哪些线程安全的list

并发容器CountDownLatch,CycleBarrier,Semaphore
CopyOnWriteArrayList&CopyOnWriteArraySet（运用CopyOnWriteArrayList实现，add方法加了equal去重）

同步容器

1 java中的同步容器

https://www.cnblogs.com/dolphin0520/p/3933404.html（基于Synchronized的设计）
1）Vector、Stack(继承了Vector)、HashTable
2）Collections类中提供的静态工厂方法创建的类

同步案例

1 怎么实现一个线程安全的计数器？
AtomicInteger（volatile+CAS）&Sychronized （incrementAndGet）&AtomicInteger

2 高并发时怎么限流
计数器、漏桶和令牌桶算法。

3 阻塞队列不用java提供的自己怎么实现condition和wait不能用
使用ArrayList作为容器，使用Sychronied锁来实现
https://blog.csdn.net/u010348570/article/details/81871252

fork&join算法

1 fork/join（分支递归，类似归并排序算法的思想）
fork分割，join阻塞等待执行结果
1.任务分割 2.多线程执行任务并合并结果（future模式）
实现原理：

- ForkJoinPool：它实现ExecutorService接口和work-stealing算法。它管理工作线程和提供关于任务的状态和它们执行的信息。
- ForkJoinTask： 它是将在ForkJoinPool中执行的任务的基类。它提供在任务中执行fork()和join()操作的机制，并且这两个方法控制任务的状态。通常， 为了实现你的Fork/Join任务，你将实现两个子类：RecursiveAction对于没有返回结果的任务&RecursiveTask 对于返回结果的任务。

工作窃取算法：
工作线程维护子任务双向队列，新任务放到队列头部，工作线程从队列头部抽取任务执行，当队列为空，随机从其他工作线程的队列的尾部获取一个任务(工作窃取算法)。

2 一个任务分成十个任务，最后汇总计算，不能用fork/join
（1）任务分割（2）分派执行 （3）执行结果汇总
并行提交，阻塞等待全任务执行结果，最后进行数据汇总

数据结构

Collection集合类

1 hashset底层实现，hashmap的put操作过程
HashSet的底层实现就是HashMap，使用了虚拟的value Object对象
hashmap的put操作是先Array+List
1 hash到Array的某个节点上； 2. 遍历节点的list； 3.判断是否相等，调用equals方法判断相等，4.判断是否需要进行扩容；5.放入对象

2 说说HaspMap底层原理？再说说它跟HashTable和ConcurrentHashMap他们之间的相同点和不同点？
HashMap:Array+EntryList 线程不安全
HashTable:线程安全
ConcurrentHashMap：分段可重入锁 线程安全

JDK1.8引入了红黑树 CAS+Sychronized换掉了可重入锁 锁细化了 Sychronized不升级成重量级别锁就不会影响效率
JDK1.8将头插法换成尾插法
JDK对hash算法做了升级，亦或操作，找位置使用的&
https://www.jianshu.com/p/939b8a672070

3 TreeMap的底层实现原理
Tree底层采用红黑树来实现
第一：构建排序二叉树，第二：平衡二叉树

4 ArrayList和LinkedList的插入和访问的时间复杂度？
1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。
2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。
3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

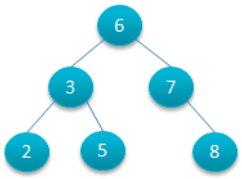
5 ArrayList是如何实现的，ArrayList和LinkedList的区别？ArrayList如何实现扩容。
ArrayList动态数组，初始10，扩容1/2，轮流拷贝，读大于写
LinkedList底层是双向队列，读取效率低，适合写大于读，无初始，无扩容

6 CopyOnWriteArrayList实现原理(双内存)
初始化单个容器，共享读，存在写入操作时Copy容器，写入阶段但读取数据的请求依然读取老的容器，写入完成进行引用覆盖。通过两块内存达到读写分离的效果。

树

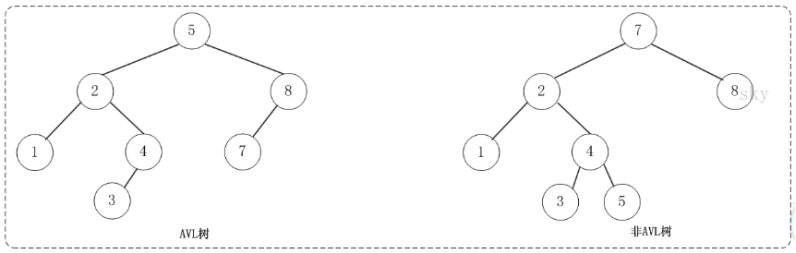
1 说说B+树和B树的区别，优缺点等？
1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；
2.不可能在非叶子结点命中；
3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；
4.更适合文件索引系统；
优点：
B+树的磁盘读写代价更低，节点存储关键字主键更多
B+树的数据信息遍历更加方便，遍历叶子节点就完成全遍历
B+树的查询效率更加稳定，直达叶子节点
缺点：
插入时，主键不是有序递增，插入数据产生大量的数据迁移和空间碎片

2 树结构的分类
二叉查找树：
左子树小于父节点，右字数大于父节点



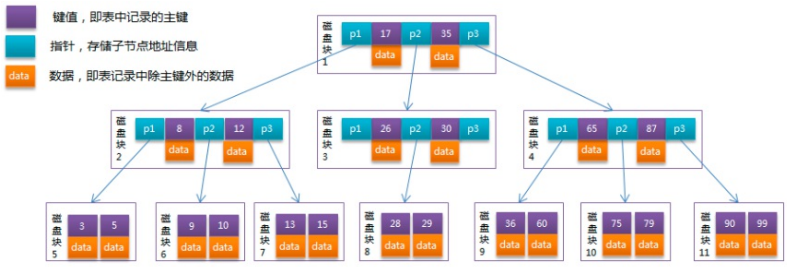
平衡二叉树(AVL)：

左字树和右子树高度之差<=1



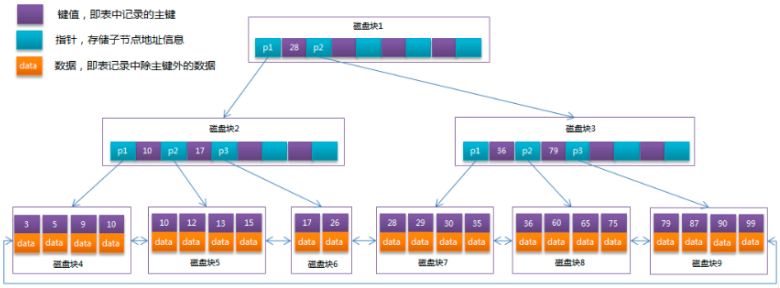
B-树，平衡多路查找树：

不是二叉树，n阶查找树，每一个节点存储升序的真正data信息或data所在节点的地址指针（升序二分查找），键值和data一一对应



B+树，平衡多路查找树：

B+树的优化，叶子节点存储data数据，非叶子节点数据所在地址key



红黑树，相比于平衡二叉树场景中对插入删除不频繁，只是对查找特别有要求，AVL还是优于红黑，红黑树，约束力比较弱的微平衡二叉树。最长路径不会是最短路径的两倍

3 B+树如何实现范围查询

（1）B+树在叶子节点存储数据信息，非叶子节点存放有序的数据索引信息

（2）B+树满足左子树比当前节点小，右子树比节点大的特性

（3）找到范围查询的两个临界点的索引，索引的磁盘地址区间数据即是范围查询的结果

算法

排序

1 了解哪些排序算法，讲讲复杂度

各种常用排序算法

排序方法	时间复杂度			空间复杂度	稳定性	复杂性	特点
	最好	平均	最坏	辅助存储			
直接插入	O(N)	$O(N^2)$	$O(N^2)$	O(1)	稳定	简单	
希尔排序	O(N)	$O(N^{1.3})$	$O(N^2)$	O(1)	不稳定	复杂	
直接选择	O(N)	$O(N^2)$	$O(N^2)$	O(1)	不稳定		
堆排序	$O(N*\log_2N)$	$O(N*\log_2N)$	$O(N*\log_2N)$	O(1)	不稳定	复杂	
冒泡排序	O(N)	$O(N^2)$	$O(N^2)$	O(1)	稳定	简单	1、冒泡排序是一种用时间换空间的排序方法，n小时好 2、最坏情况是把顺序的排列完成逆序，或者把逆序的数列变成顺序，最差时间复杂度O(N^2)只是表示其操作次数的数量级 3、最好的情况是数据本来就有序，复杂度为O(n)
快速排序	$O(N*\log_2N)$	$O(N*\log_2N)$	$O(N^2)$	$O(\log_2n)\sim O(n)$	不稳定	复杂	1、n大时好，快速排序比较占用内存，内存随n的增大而增大，但却是效率高不稳定的排序算法。 2、划分之后一边是一个，一边是n-1个，这种极端情况时间复杂度是O(N^2) 3、最好的情况是每次都能均匀的划分应为，O(N*log2N)
归并排序	$O(N*\log_2N)$	$O(N*\log_2N)$	$O(N*\log_2N)$	O(n)	稳定	复杂	1、n大时好，归并比较占用内存，内存随n的增大而增大，但却是效率高且稳定的排序算法。

2 归并排序(分治递归)

3 对一个链表进行归并排序，链表可能有环

4 堆排序

堆是完全二叉树：每个结点的值都大于等于其左右孩子，称为大顶堆；或者每个结点的值都小于等于其左右孩子，称为小顶堆，左右孩子之间不做约束。

升序大顶堆，降序小顶堆

升序排序的方案：

a.将无需序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆;

b.将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端;

c.重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。O(nlogn)

一个班每个学生都有一个考试分数，找到10-50之间的学生，非典型范围查找，使用的数据结构

跳表

找到无序队列中第二小的数字

使用快速排序找到第K小的数值

无序栈转化为有序栈

汉洛塔

加密&解密

1 md5加密的原理

MD5：安全的散列算法，即其过程不可逆；MD5以512位分组来处理输入的信息，且每一分组又被划分为16个32位子分组，经过了一系列的处理后，算法的输出由四个32位分组组成，将这四个32位分组级联后将生成一个128位散列值。

1.信息填充，使其字节长度对512求余数的结果等于448。信息的字节长度扩展至N*512+448，即N*64+56个字节，N为正整数。填充的方法是在信息的后面填充一个1和无数个0。

2.设置四个链接变量，算法的四轮循环运算，循环的次数是信息中512位信息分组的数目，输出由四个32位分组组成。

数据去重

1 10亿个数去重

空间换时间，使用位图的方式来进行数据的去重。（https://blog.csdn.net/hustwht/article/details/52181643）

2 URL去重（k次hash，n长数组，最多能区别2^n方个数据，每个数据对应一个bit数组）

布隆过滤器：

1. 首先需要k个hash函数，每个函数可以把key散列成为1个整数
2. 初始化时，需要一个长度为n比特的数组，每个比特位初始化为0
3. 某个key加入集合时，用k个hash函数计算出k个散列值，并把数组中对应的比特位置为1
4. 判断某个key是否在集合时，用k个hash函数计算出k个散列值，并查询数组中对应的比特位，如果所有的比特位都是1，认为在集合中。

优点：不需要存储key，节省空间

缺点：1. 算法判断key在集合中时，有一定的概率key其实不在集合中 2. 无法删除

基于MD5压缩映射的存储

一致性Hash算法

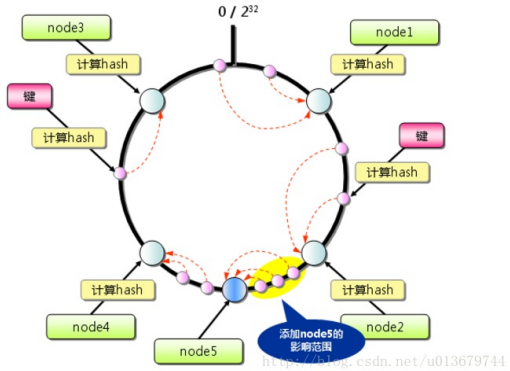
1 一致性Hash算法

https://www.cnblogs.com/lpfuture/p/5796398.html

key个数为2的32次方的圆环

1，求出每个服务器的IP获得hash值，将其配置到一个 0~2^n 的圆环上（n通常取32）称为Node。

2，用同样的方法求出待存储对象的主键 hash值，也将其配置到这个圆环上，然后从数据映射到的位置开始顺时针查找，找到第一个Node进行存储。



节点增删，需要对当前操作节点和逆时针找到的第一个节点之间的数据进行数据迁移

节点雪崩，对热点Node添加虚拟节点，减少节点的压力

算法题

1 LintCode 算法题 - 最小子串覆盖

给定一个字符串 source 和一个目标字符串 target，在字符串 source 中找到包括所有目标字符串字母的子串。如果在 source 中没有这样的子串，返回""，如果有多个这样的子串，返回长度最小的子串。

利用 hash 表来记录字符串中字母出现次数

当 target 中每个字母在 sourcehash 表中出现次数大于在 targethash 表中出现次数则认为满足包含条件

2 回环递增序列找到第k大的值，转化为找最小值的位置的问题

变形二分查找，首尾比较

3 布隆过滤器了解么，讲了ip地址过滤的布隆过滤器实现。

Bit数组，K重Hash

4 如何倒序输出单向链表

采用栈的方式

5 动态规划问题

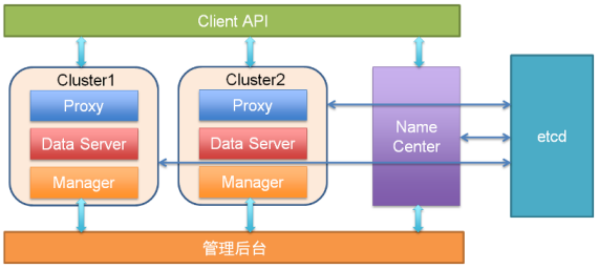
公司框架

Wtable

如果你对WTable内部工作原理感兴趣，可以阅读这部分内容。

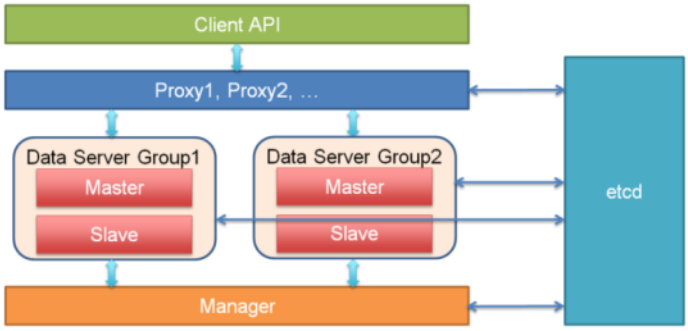
WTable采用了多集群架构，在一个大集群里面可以部署很多小集群，所有小集群都在系统的整体管理之下，请看下面的架构图。每个小集群都可以部署在不同的机器上，物理级别的隔离，相互之间没有影响。采用多集群的架构好处是业务之间可以相互隔离，并且可以控制每个小集群的规模相对较小，整个系统的稳定性有很好的保证。

- Name Center：提供一种“类似域名解析”的服务。Name Center会实时从etcd同步所有小集群Proxy服务器地址信息，剔除有问题的Proxy，并提供接口根据bid拉取Proxy地址列表。
- etcd：系统内部的配置中心，支持实时推送配置变更，高可用。



每个小集群内部的架构如下图，客户端通过Name Center获取了Proxy的地址列表之后，只会和Proxy进行通信，Proxy会将请求路由到合适的Data Server。

- Proxy：代理服务，它知道每个rowKey应该存储在哪个Data Server Group，并进行请求路由。如果是MGET/MSET这类批量请求，它会将请求进行分包，并将结果进行汇总返回给客户端。Proxy本身无状态，每个Proxy提供的功能都是对等的，支持线性无限扩展。
- Data Server：数据存储服务，它是整个系统的核心。一个Data Server Group里面会包含几台Data Server，一般是2至3台，其中一台是Master，其余的是Slave。Master的选举是自动执行的，不用人工干预。所有写操作都先在Master上执行，并实时同步给所有的Slave。读取操作可以在Master或者Slave上执行。Master和Slave数据同步的延迟非常低，因此数据不一致的概率很低，大部分业务从Slave读取数据都是安全的。如果对数据一致性有特殊需要，可以设置CAS请求参数，可以指定只从Master读取数据，或者通过Cas.LOCK参数来加乐观锁，具体请参考CAS相关说明。Data Server数据存储采用了LSM树(Log-Structured Merge Tree)存储引擎，所有数据实时落地到文件，并且避免了随机写，因此有非常好的写性能；数据读取在本机有block cache，采用LRU淘汰策略，因此读取性能非常棒，并且能保证永远不会出现cache数据不一致的问题。
- 数据扩容：如果当前的Data Server容量不够了，可以通过增加新的Data Server Group的方式来扩容。WTable内部有一套自动扩容的机制，只要部署好了新的Data Server，通过管理后台即可方便的扩容，数据会自动平滑的迁移到新的Data Server，这个过程业务无感知，线上服务依然可以正常读写。



SCF

组成 SCF主要由4部分组成
SCF Server：服务容器，用于宿主开发人员所开发的SCF服务，接收和处理来自客户 端的请求。
SCF Client：多种平台客户端，调用者就像在调用本地接口一样方便，同时还提供 了负载均衡，容错等机制。
SCF Serializer：提供了跨平台的二进制序列化解决方案。
SCF Protocol：分为传输协议和数据协议．传输协议用于进行数据传输，数据协议 用于
特性
跨平台： 由于特定系统的需要，会使用特定的语言或平台来开发，便会存在多个平台的情况，比如有 java、.net、c++等，想要实现这些系统的简单高效的跨平台调用是一个非常麻烦的事情， SCF 通过为不同平台提供不同的客户端来实现跨平台，目前已有 java、.net、c++6 等平台 的客户端。服务开发人员在开发服务时无需关心这些细节，所开发出来的服务就能很好的支 持跨平台调用。
高并发： 业务流量大，每天的调用的流量太大。
高性能： 客户端和服务端特定的通讯模型，序列化组件对序列化和反序列化性能以及结果字 节数组大小的严格控制，通讯协议的针对性设计等使得 SCF 比.net 平台的 WCF，Java 平台的 EJB、RMI，跨平台的 WebService 等性能都要好。在追求性能的同时支持客 户端的 HA（High Availability），容错机制为服务提供了良好的可靠性保证。
高可靠性： SCF 可以对调用者进行权限授权，不同的调用者只能调用对他授权的方法，这对有 些对外暴露的服务又想对访问进行授权将会非常有用。
异步多协议： 多台机器之间的会话规则。

WF

Spring&Spring MVC&SpringBoot

Spring

1 Spring是什么？

Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置。
主要由以下几个模块组成：
Spring Core：核心类库，提供IOC服务；
Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；
Spring AOP：AOP服务；
Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；
Spring ORM：对现有的ORM框架的支持；
Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；
Spring MVC：提供面向Web应用的Model-View-Controller实现。

2 Spring 的优点？

（1）spring属于低侵入式设计，代码的污染极低；
（2）spring的DI机制将对象之间的依赖关系交由框架处理，减低组件的耦合性；
（3）Spring提供了AOP技术，支持将一些通用任务，如安全、事务、日志、权限等进行集中式管理，从而提供更好的复用。
（4）spring对于主流的应用框架提供了集成支持。

3 Spring的AOP理解：

OOP面向对象，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。
AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。
AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。
（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。
（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。
Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：
①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例，生成目标类的代理对象。
②如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。
（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。
InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例； method 是被代理目标实例的某个具体方法； args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

4 Spring的IoC理解：

（1）IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部资源。

（2）最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

（3）Spring的IOC有三种注入方式 ：构造器注入、setter方法注入、根据注解注入。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（2）①BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

6 请解释Spring Bean的生命周期？

首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；

Spring上下文中的Bean生命周期也类似，如下：

（1）实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成依赖注入。

（3）处理Aware接口：

接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：

①如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值；

②如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

③如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

（4）BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

（5）InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（7）DisposableBean：

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（8）destroy-method：

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

7 解释Spring支持的几种bean的作用域。

Spring容器中的bean可以分为5个范围：

（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

8 Spring框架中的单例Beans是线程安全的么？

Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

9 Spring如何处理线程并发问题？

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

10 Spring基于xml注入bean的几种方式：

（1）Set方法注入；

（2）构造器注入：①通过index设置参数的位置；②通过type设置参数类型；

（3）静态工厂注入；

（4）实例工厂；

详细内容可以阅读：https://blog.csdn.net/a745233700/article/details/89307518

11 Spring的自动装配：

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

（1）no：默认的方式是不进行自动装配的，通过手工设置ref 属性来进行装配bean。

（2）byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

（3）byType：通过参数的数据类型进行自动装配。

（4）constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。

（5）autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

基于注解的方式：

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会根据名称来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

@Autowired可用于：构造函数、成员变量、Setter方法

注：@Autowired和@Resource之间的区别

(1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

(2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

12 Spring 框架中都用到了哪些设计模式？

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

13 Spring事务的实现方式和实现原理：

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

（1）Spring事务的种类：

spring支持编程式事务管理和声明式事务管理两种方式：

①编程式事务管理使用TransactionTemplate。

②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

（2）spring的事务传播行为：

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

④ PROPAGATION_REQUIRES_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION_NOT_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

（3）Spring中的隔离级别：

① ISOLATION_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION_READ_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION_READ_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION_REPEATABLE_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新

⑤ ISOLATION_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

14 Spring框架中有哪些不同类型的事件？

Spring 提供了以下5种标准的事件：

（1）上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。

（2）上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。

（3）上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。

（4）上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。

（5）请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。

如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

15 解释一下Spring AOP里面的几个名词：

（1）切面（Aspect）：被抽取的公共模块，可能会横切多个对象。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。

（3）通知（Advice）：在切面的某个特定的连接点（Join point）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。

（4）切入点（Pointcut）：切入点是指 我们要对哪些Join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add*、search*。

（5）引入（Introduction）：（也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

16 Spring通知有哪些类型？

https://blog.csdn.net/qq_32331073/article/details/80596084

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。 环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice
before advice
target method 执行
around after advice
after advice
afterReturning

②有异常情况下的执行顺序：

around before advice
before advice
target method 执行
around after advice
after advice
afterThrowing:异常发生
java.lang.RuntimeException: 异常发生

17 说了Spring，问我Spring中如何让A和B两个bean按顺序加载？（**）

https://blog.csdn.net/qq_27529917/article/details/79329809

18 Spring bean的生命周期？（61题）默认创建的模式是什么？不想单例怎么办？

https://www.cnblogs.com/zrtqsk/p/3735273.html

bean创建方式：（1）xml配置 （2）注解@Component@Controller@Service - 功能性质的备案 (3)@Bean注解，纯粹的bean对象

默认采用单例的模式创建

singleton&(单例模式管理bean实例)prototype（原型模式创建bean实例）

19 spring一个bean装配的过程？

第一：如果你使用BeanFactory作为Spring Bean的工厂类，则所有的bean都是在第一次使用该Bean的时候实例化

第二：如果你使用ApplicationContext作为Spring Bean的工厂类，则又分为以下几种情况：

（1）：如果bean的scope是singleton的，并且lazy-init为false（默认是false，所以可以不用设置），则 ApplicationContext启动的时候就实例化该Bean，并且将实例化的Bean放在一个map结构的缓存中，下次再使用该Bean的时候， 直接从这个缓存中取

（2）：如果bean的scope是singleton的，并且lazy-init为true，则该Bean的实例化是在第一次使用该Bean的时候进行实例化

（3）：如果bean的scope是prototype的，则该Bean的实例化是在第一次使用该Bean的时候进行实例化

1、实例化一个Bean
2、按照Spring上下文IOC注入；
3、BeanNameAware&BeanFactoryAware&BeanContextAware 实现则set
4、关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)
5、如果Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法。
6、关联了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法、；
7、当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用那个其实现的destroy()方法；
8、最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

SpringMVC

1 请详细描述springmvc处理请求全流程？（DispatcherServlet->HandlerMapping->HandlerAdapter->ViewResolver->view渲染）

1、DispatcherServlet前端控制器接收发过来的请求，交给HandlerMapping处理器映射器
2、HandlerMapping处理器映射器，根据请求路径找到相应的HandlerAdapter处理器适配器（处理器适配器就是那些拦截器或Controller）
3、HandlerAdapter处理器适配器，处理一些功能请求，返回一个ModelAndView对象（包括模型数据、逻辑视图名）

4、ViewResolver视图解析器，先根据ModelAndView中设置的View解析具体视图
5、然后再将Model模型中的数据渲染到View上

2 什么是Spring MVC？简单介绍下你对springMVC的理解?

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把Model，View，Controller分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

3 Springmvc的优点:

（1）可以支持各种视图技术,而不仅仅局限于JSP;
（2）与Spring框架集成（如IoC容器、AOP等）；
（3）清晰的角色分配：前端控制器(dispatcherServlet)，请求到处理器映射（handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器（ViewResolver）。
（4） 支持各种请求资源的映射策略。

4 Spring MVC的主要组件？

（1）前端控制器 DispatcherServlet（不需要程序员开发）
作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。
（2）处理器映射器HandlerMapping（不需要程序员开发）
作用：根据请求的URL来查找Handler
（3）处理器适配器HandlerAdapter
注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。
（4）处理器Handler（需要程序员开发）
（5）视图解析器 ViewResolver（不需要程序员开发）
作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）
（6）视图View（需要程序员开发jsp）
View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

5 springMVC和struts2的区别有哪些?

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

6 SpringMVC怎么样设定重定向和转发的？

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"
（2）重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

7 SpringMvc怎么和AJAX相互调用的？

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。具体步骤如下 ：
（1）加入Jackson.jar
（2）在配置文件中配置json的映射
（3）在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

8 如何解决POST请求中文乱码问题，GET的又如何处理呢？

（1）解决post请求乱码问题：
在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；

```
<filter>
    <filter-name>CharacterEncodingFilter</filter-name>
    <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>CharacterEncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

（2）get请求中文参数出现乱码解决方法有两个：
①修改tomcat配置文件添加编码与工程编码一致，如下：

```
<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
```

②另外一种方法对参数进行重新编码：

```
String userName = new String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")
```

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码

9 Spring MVC的异常处理 ？

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

10 SpringMvc的控制器是不是单例模式,如果是,有什么问题,怎么解决?

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

11 SpringMVC常用的注解有哪些？

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。
@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。
@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。

12 SpingMvc中的控制器的注解一般用那个,有没有别的注解可以替代？

答：一般用@Conntroller注解,表示是表现层,不能用别的注解代替。

13 如果在拦截请求中，我想拦截get方式提交的方法,怎么配置？

答：可以在@RequestMapping注解里面加上method=RequestMethod.GET。

14 怎样在方法里面得到Request,或者Session？

答：直接在方法的形参中声明request,SpringMvc就自动把request对象传入。

15 如果想在拦截的方法里面得到从前台传入的参数,怎么得到？

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

16 如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象？

答：直接在方法中声明这个对象,SpringMvc就自动会把属性赋值到这个对象里面。

17 SpringMvc中函数的返回值是什么？

答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好。

18 SpringMvc用什么对象从后台向前台传递数据的？

答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

19 怎么样把ModelMap里面的数据放入Session里面？

答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

20 SpringMvc里面拦截器是怎么写的：

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

```
  <!-- 配置SpringMvc的拦截器 -->
<mvc:interceptors>
    <!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->
    <bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></bean>
    <!-- 只针对部分请求拦截 -->
    <mvc:interceptor>
      <mvc:mapping path="/modelMap.do" />
      <bean class="com.zwp.action.MyHandlerInterceptorAdapter" />
    </mvc:interceptor>
</mvc:interceptors>
```

21 注解原理：

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

SpringBoot

1 SpringBoot 单元测试流程

使用SpringRunner（继承自SpringJUnit4ClassRunner）引入Spring对于Junit的支持
使用MockMVC对象模拟调用Controller层发起请求

2 springboot的启动流程（**）

SpringBoot提供了一堆依赖打包，并已经按照使用习惯解决了依赖问题，可以理解为Spring套装
https://blog.csdn.net/u010811939/article/details/80592461

Spring&SpringMvc的关系

1 spring的架构和流程（SpringMVC）

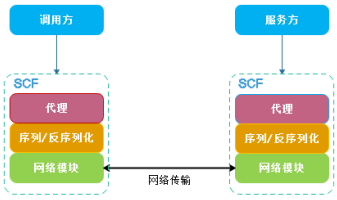
Spring是IOC和AOP的容器框架，SpringMVC是基于Spring功能之上添加的Web框架，想用SpringMVC必须先依赖Spring。

RPC框架

RPC组成

1 自己如何实现RPC?

（1）协议或者说服务方提供的接口（提供客户端使用）
（2）动态代理，客户端能像调用本地方法一样调用远程服务的基础
（3）序列化和反序列化，rpc调用的内容进行解析和传输
（4）RPC框架底层的通信传输模块，一般使用Socket
（5）具体逻辑的实现，使用反射或配置规则映射（服务端使用）



http://dy.163.com/v2/article/detail/E2129K3O0511D3QS.html

Dubbbo

1 dubbo的生产者如何发布服务，注册服务，消费者如何调用服务？

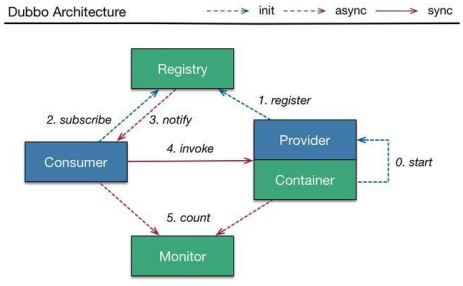
同第三题相同

2 dubbo负载均衡的策略有哪些？一致性哈希详细聊一下？

(权重，轮询，最小调用数，一致性hash)
1.RandomLoadBalance:按权重随机调用，这种方式是dubbo默认的负载均衡策略
2.RoundRobinLoadBalance：轮询，按公约后的权重设置轮询比率
3.LeastActiveLoadBalance：最少活跃次数，dubbo框架自定义了一个Filter，用于计算服务被调用的次数
4.ConsistentHashLoadBalance：一致性hash（见一致性hash算法章节第一题）
增加/删除缓存服务器的情况下，其他缓存服务器的缓存仍然可用，从而不引起雪崩问题。
我们可以通过虚拟节点的方式解决Hash环的偏移。数据不会都存储到同一台服务器上

3 dubbo的基本架构，几个组件说一下

Dubbo的原理



节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

1、容器负责启动，加载，运行服务Provider。
2、Provider向注册中心注册服务
3、消费者在启动时，向注册中心订阅服务。
4、注册中心返回Provider地址列表给消费者，如果有变更，注册中心将基于长连接主动推送变更数据给消费者。
5、consumer基于软负载均衡算法，选一台provider进行调用，如果调用失败，再选另一台调用。
6、服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

4 dubbo的服务容错怎么做，怎么知道服务器宕机了

Failover Cluster 模式（over失败自动切换）
1.失败自动切换，当出现失败，重试其它服务器。(缺省)
2. 通常用于读操作，但重试会带来更长延迟。
3. 可通过retries="2"来设置重试次数(不含第一次)。
Failfast Cluster（fast一次调用，快速失败）
快速失败，只发起一次调用，失败立即报错。
通常用于非幂等性的写操作，比如新增记录。
Failsafe Cluster（safe失败忽略）
失败安全，出现异常时，直接忽略。
通常用于写入审计日志等操作。
Failback Cluster（back失败回复，定时重发）
失败自动恢复，后台记录失败请求，定时重发。
通常用于消息通知操作。
Forking Cluster（fork并行调用，一个成功）
并行调用多个服务器，只要一个成功即返回。
通常用于实时性要求较高的读操作，但需要浪费更多服务资源。
可通过forks="2"来设置最大并行数。
Broadcast Cluster(boadcast广播逐个调用，一个失败)
广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0开始支持)
通常用于通知所有提供者更新缓存或日志等本地资源信息。
怎样知道服务器宕机？zk的心跳机制维持服务器连接

5 dubbo如何一条链接并发多个调用。

多线程并发共用同一个长连接，请求返回都要携带ID属性，返回数据通过ID属性唤起对应的DefaultFuture对象进行进一步处理。

6 Dubbo的序列化：

https://blog.csdn.net/wodediqizhang/article/details/51603512

Spring Cloud

1 Spring Cloud原理以及重要的组件

https://blog.csdn.net/xlgen157387/article/details/77773908

微服务治理中心

1 作为服务注册中心，Eureka比Zookeeper好在哪里

著名的CAP理论指出，一个分布式系统不可能同时满足C(一致性)、A(可用性)和P(分区容错性)。由于分区容错性在是分布式系统中必须要保证的，因此我们只能在A和C之间进行权衡。在此Zookeeper保证的是CP, 而Eureka则是AP。

2 Zookeeper保证CP

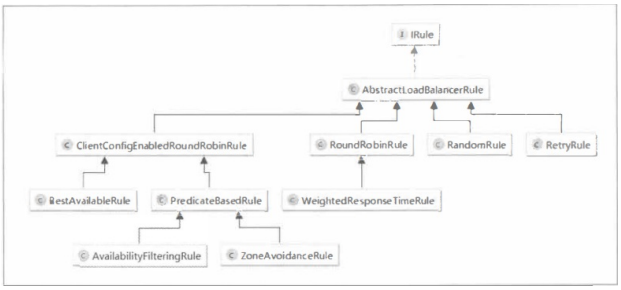
当向注册中心查询服务列表时，我们可以容忍注册中心返回的是几分钟以前的注册信息，但不能接受服务直接down掉不可用。也就是说，服务注册功能对可用性的要求要高于一致性。但是zk会出现这样一种情况，当master节点因为网络故障与其他节点失去联系时，剩余节点会重新进行leader选举。问题在于，选举leader的时间太长，30 ~ 120s, 且选举期间整个zk集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得zk集群失去master节点是较大概率会发生的事，虽然服务能够最终恢复，但是漫长的选举时间导致的注册长期不可用是不能容忍的。

3 Eureka保证AP

Eureka看明白了这一点，因此在设计时就优先保证可用性。Eureka各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而Eureka的客户端在向某个Eureka注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台Eureka还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。除此之外，Eureka还有一种自我保护机制，如果在15分钟内超过85%的节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，此时会出现以下几种情况：
1. Eureka不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
2. Eureka仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
3. 当网络稳定时，当前实例新的注册信息会被同步到其它节点中
因此， Eureka可以很好的应对因网络故障导致部分节点失去联系的情况，而不会像zookeeper那样使整个注册服务瘫痪。
eureka提供了region和zone两个概念来进行分区，这两个概念均来自于亚马逊的AWS:

- region：可以简单理解为地理上的分区，比如亚洲地区，或者华北地区，再或者北京等等，没有具体大小的限制。根据项目具体的情况，可以自行合理划分region。
- zone：可以简单理解为region内的具体机房，比如说region划分为北京，然后北京有两个机房，就可以在此region之下划分出zone1,zone2两个zone。
- 一个region可以包含多个zone，每个eureka服务客户端需要被注册到一个对应的zone当中，所以每个客户端只能对应一个region和一个zonge

eureka负载均衡策略：



内置负载均衡规则类	规则描述
RoundRobinRule	简单轮询服务列表来选择服务器。它是Ribbon默认的负载均衡规则。
AvailabilityFilteringRule	对以下两种服务器进行忽略： （1）在默认情况下，这台服务器如果3次连接失败，这台服务器就会被设置为“短路”状态。短路状态将持续30秒，如果再次连接失败，短路的持续时间就会几何级地增加。 注意：可以通过修改配置loadbalancer.<clientName>.connectionFailureCountThreshold来修改连接失败多少次之后被设置为短路状态。默认是3次。 （2）并发数过高的服务器。如果一个服务器的并发连接数过高，配置了AvailabilityFilteringRule规则的客户端也会将其忽略。并发连接数的上线，可以由客户端的<clientName>.<clientConfigNameSpace>.ActiveConnectionsLimit属性进行配置。
WeightedResponseTimeRule	为每一个服务器赋予一个权重值。服务器响应时间越长，这个服务器的权重就越小。这个规则会随机选择服务器，这个权重值会影响服务器的选择。
ZoneAvoidanceRule	以区域可用的服务器为基础进行服务器的选择。使用Zone对服务器进行分类，这个Zone可以理解为一个机房、一个机架等。
BestAvailableRule	忽略哪些短路的服务器，并选择并发数较低的服务器。
RandomRule	随机选择一个可用的服务器。
Retry	重试机制的选择逻辑

数据库

Redis

1 Redis有哪些数据结构？底层的编码有哪些？有序链表采用了哪些不同的编码？

1.String(Key,Value) function:get,set,del
2.List(Key,List) function:rpush,lrange,lindex,lpop
3.Set(Key,Set) function:sadd,smembers,sismember,srem
4.Hash(Key,Map) function:hset,hget(key,mapkey),hgetall,hdel
5.Zset(Key,Set)function:zadd,zrange,zrangebyscore,zrem -区别在于每个Set每个元素都有一个分值，实现有序set的目的
Redis数据编码：https://blog.csdn.net/snakorse/article/details/78154402

```
#define OBJ_ENCODING_RAW 0     /* Raw representation */
#define OBJ_ENCODING_INT 1     /* Encoded as integer */
#define OBJ_ENCODING_HT 2      /* Encoded as hash table */
define OBJ_ENCODING_ZIPMAP 3  /* Encoded as zipmap */ // 已废弃
define OBJ_ENCODING_LINKEDLIST 4 /* Encoded as regular linked list */
define OBJ_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
define OBJ_ENCODING_INTSET 6  /* Encoded as intset */
define OBJ_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
define OBJ_ENCODING_EMBSTR 8  /* Embedded sds string encoding */
define OBJ_ENCODING_QUICKLIST 9 /* Encoded as linked list of ziplists */
```

有序链表采用的编码：ziplist和skiplist

2 redis的hash数据结构最多能存储多少个元素

Every hash can store up to 2^32 - 1 field-value pairs (more than 4 billion).

3 Redis的高可用方案

（哨兵：监控&提醒&故障迁移&选主 一致性Hash：增删移动少，虚点热点分流 主从：多副本）
集群化通用解决方案。包含上述所有内容
一.哨兵机制（多个，判断节点死亡需要投票超过半数）
监控，检查你的Master和Slave是否运作正常
提醒，Redis出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知
自动故障迁移，重新选主
二.一致性哈希
通过对2^32取模的方式，保证了在增加/删除缓存服务器的情况下，其他缓存服务器的缓存仍然可用，从而不引起雪崩问题。我们可以通过虚拟节点的方式解决Hash环的偏移。数据不会都存储到同一台服务器上
三.Redis官方集群方案 Redis Cluster
http://www.cnblogs.com/huangjuncong/p/8494295.html

4 Redis单线程模型提供高并发

区别并发和并行的关系&单线程减少了线程上下文的切换&采用IO多路复用的策略
https://segmentfault.com/a/1190000013613884

5 Redis数据热点问题，如何发现数据热点

数据热点问题的解决：
（1）请求进入队列，等待热点redis节点重建完成
（2）添加分布式锁，只允许一个线程访问db资源
（3）主从分离，主节点负责读写，从节点负责读操作
热点发现：
（1）数据访问量日志记录进行推算
（2）利用redis4.x自身特性，LFU机制发现热点数据（LFU：最近最少使用算法，每个对象包含24bit空间记录访问信息，高16位访问时间，低8位访问频次）

Mysql

1 mysql默认存储引擎？MyISAM、InnoDB、MEMORY的区别

InnoDB存储引擎：事务，自增列，外键，mvcc，行锁
MyISAM存储引擎：静态型、动态型、压缩型存储结构，并发插入的表锁（针对select量大的情况）
MEMORY存储引擎：数据存储在内存，表结构存储磁盘frm文件，hash索引，数据安全性不高
读锁：共享锁、S锁，S锁存在不可再对同一个对象添加X锁
写锁：排他锁、X锁，重量级锁
表锁：操作对象是数据表。是系统开销最低但并发性最低的一个锁策略。事务对整个表加读锁，其他事务可读不可写，若加写锁，则其他事务增删改都不行。
行级锁：操作对象是数据表中的一行，但行级锁对系统开销较大，处理高并发较好。
MVCC：多版本并发控制。类比CAS，但系统开销比最大(较表锁、行级锁)，这是最求高并发付出的代价。
Autocommit：是mysql一个系统变量，默认情况下autocommit=1表示mysql把没一条sql语句自动的提交，而不用commit语句。所以，当要开启事务操作时，要把autocommit设为0，可以通过“set session autocommit=0; ”来设置

2 什么是脏读，幻读，不可重复读，如何解决？

脏读 ：一个事务对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。（一读一修改）
不可重复读：一个事务多次读同一数据，（前后读取结果不一致），针对单行数据变化产生的错误。（多读一修改）
幻读：例如事务在插入已经检查过不存在的记录时，发生数据插入的冲突。针对的数据行数发生的变化。（多读一增删）
解决方案：
mysql通过事务隔离级别来解决读写不一致的问题
READ UNCOMMITTED（未提交读）：事务修改使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读（Dirty Read）。
READ COMMITTED（提交读）：一个事务开始时，只能“看见”已经提交的事务所做的修改。不可重复读。
REPEATABLE READ（可重复读）：
SERIALIZABLE（可串行化）：通过强制事务串行执行，SERIALIZABLE会在读取的每一行数据上都加锁，所以可能导致大量的超时和锁争用的问题
原理：
1 脏读：修改时加排他锁，读取时加共享锁，读时不可写
2 不可重复读：读取数据时加共享锁，修改排它锁
3 幻读问题：A:串行化事务 B:MVCC 版本并行空值 C:RangeS RangeS_S模式，锁定检索范围为只读

3 事务隔离级别有什么？通过什么来实现的？分别解决了什么问题？

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable）	否	否	否

4 乐观锁与悲观锁的使用场景

读取频繁使用乐观锁，写入频繁使用悲观锁。

5 数据库主从同步数据一致性如何解决？技术方案的优劣势比较？

1.半同步复制办法就是等主从同步完成之后，等主库上的写请求再返回，这就是常说的“半同步复制”。
优点：利用数据库原生功能，比较简单 缺点：主库的写请求时延会增长，吞吐量会降低
2.数据库中间件（读写直接请求中间件，中间件承担请求路由的作用）

1）所有的读写都走数据库中间件，通常情况下，写请求路由到主库，读请求路由到从库
2）记录路由到写库的key，主从同步时间窗口内（假设是500ms），如果有读请求访问就把这个key上的读请求路由到主库。
3）在主从同步时间过完后，对应key的读请求继续路由到从库。

6 数据库如果让你来垂直和水平拆分，谁先拆分，拆分的原则有哪些(单表数据量多大拆)

通俗理解：水平拆分行，行数据拆分到不同表中， 垂直拆分列，表数据拆分到不同表中，单表数据达到千万级

7 数据库索引？B+树？为什么要建索引？什么样字段需要建索引，建索引的时候一般考虑什么？索引会不会使插入、删除作效率变低，怎么解？

定义: 索引是对数据库表中一列或多列的值进行排序的一种结构
原因: 加快对表中记录的**查找**或**排序**
字段选择: 表的主键，外键，非外键的连接字段（用于关联查询的字段），枚举类型的数据(有限的变化区间)，数值类型的数据
考虑：索引的维护成本
索引插入删除慢：是否需要删除不必要的索引字段，优化查询语句，插入删除对于实时性的要求，做临时表延迟处理

8 数据库表怎么设计的？数据库范式？设计的过程中需要注意什么？

9 sql查询中哪些情况不会使用索引？（完全不懂）

查询非索引列的信息（select*），索引列上进行函数计算，隐式转化，表数据小
1、查询谓词没有使用索引的主要边界,换句话说就是select *，可能会导致不走索引。
2、单键值的b树索引列上存在null值，导致COUNT(*)不能走索引。
3、索引列上有函数运算，导致不走索引
4、隐式转换导致不走索引。
5、表的数据库小或者需要选择大部分数据，不走索引
6、cbo优化器下统计信息不准确，导致不走索引
7、！=或者<>(不等于），可能导致不走索引，也可能走 INDEX FAST FULL SCAN
8、表字段的属性导致不走索引，字符型的索引列会导致优化器认为需要扫描索引大部分数据且聚簇因子很大，最终导致弃用索引扫描而改用全表扫描方式，

10 当前读&快照读？

在可重复读级别下，快照读是通过MVVC(多版本控制)和undo log来实现的，当前读是通过加record lock(记录锁)和gap lock(间隙锁)来实现的。
1. 快照读 (snapshot read)
简单的select操作(不包括 select ... lock in share mode, select ... for update)
2.当前读 (current read)
select ... lock in share mode&select ... for update&insert&update&delete

11 mysql同步机制原理，有哪几种同步方法

- 基于 SQL 语句的复制(statement-based replication, SBR);

sql语句写入binllog，binlog小，从服务器版本可能大于主服务器，行锁更多

- 基于行的复制(row-based replication, RBR);

每一条行数据写入binlog，数据安全可靠，可以采用多线程复制，binlog文件大

- 混合模式复制(mixed-based replication, MBR)：混合模式

索引

1 聚集索引和非聚集索引知道吗？什么情况用聚集索引什么情况用非聚集索引

物理内存连续性&表能创建个数
读取聚集快，增删聚集慢，大数据集合查询，范围查询优
1、聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个，这个跟没问题没差别，一般人都知道。
2、聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续，这个大家也都知道。
聚集索引像拼音查字典，拼音的顺序和汉字在字典中存在的位置顺序保持一致，A找到啊，啊在第一页第一个，B找到比，比在第10页，a在b前，a在比前
非聚集索引像是笔画查字典，笔画靠前不一定笔画对应的汉字也靠前
聚集索引对应的叶子节点存放最终数据
非聚集索引的叶子节点存放的对应数据行的内存地址和索引的字段，不包含数据行的全部数据
使用场景：
使用聚集索引的查询效率要比非聚集索引的效率要高，但是如果需要频繁去改变聚集索引的值，写入性能并不高，因为需要移动对应数据的物理位置。
非聚集索引在查询的时候可以的话就避免二次查询，这样性能会大幅提升。
不是所有的表都适合建立索引，只有数据量大表才适合建立索引，且建立在选择性高的列上面性能会更好。
如何解决非聚集索引的二次查询问题？（非聚查主键，聚集主键查内容）
复合索引（覆盖索引）多列索引
PS:在非聚集索引中可以直接查到非聚集索引的索引列内容
非聚集索引分类：
普通索引：最基本的索引，无限制索引
唯一索引：索引列的值唯一允许有空值。
主键索引：不允许有空值的唯一索引。
全文索引：仅可用于 MyISAM 表，针对较大的数据，生成全文索引很耗时好空间。
组合索引：多列索引的组合，遵循"最左前缀"原则。

2 复合索引的最左前缀原则（判断使用index&ref(扫全索引)查询）

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。
1.索引左前缀性的第一层意思：必须用到索引的第一个字段。
2. 索引前缀性的第二层意思：对于索引的第一个字段，用like时左边必须是固定值，通配符只能出现在右边。
3.索引前缀性的第三层意思：如果在字段前加了函数，则索引会被抑制
（a,b,c） 查询（a,c）只有a走索引，c需要走index扫描

name	cid
a	6
c	4
c	5
h	1
z	9

3 数据库索引，底层是怎样实现的，为什么要用B树索引？

数据库索引采用B+树实现，提升查找效率，见149

4 MySQL是怎么用B+树？

https://www.cnblogs.com/tiancai/p/9024351.html

5 索引有什么用？如何建索引？（加快查询效率）

1 选择区分度高的字段作为 索引 字段
2 范围 ，条件不明确的 有索引 速度也会很慢
3 索引字段 不能 通过 *10 avg() 等公式计算
4 最左前缀 --联合索引 (a,b,c,d)

6 存在慢查询日志但是线上查询explain发现走索引的原因？

1）数据后置处理groupby
2）返回的数据集太大
3）使用多列索引的查询语句

7 什么叫做倒排索引

由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引(inverted index)。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件(inverted file)。全文检索就是倒排索引。

8 一条SQL语句执行得很慢的原因

（1）大多数情况是正常的，只是偶尔会出现很慢的情况。
1、数据库在刷新脏页
redo log打满，请求等待redo log同步磁盘完成，任何一条sql都有可能
2、语句涉及到的表加锁了（mysql可以使用show processlist）
（2）在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。
1、扎心了，没用到索引（字段没有索引||字段有索引，但却没有用索引）

缓存

1 缓存失效如何解决？

1 大面积的缓存key失效（缓存的时间分散分布）
2 热点key失效（加锁重构缓存，备份缓存，缓存降级提示等待重试）

2 缓存问题（击穿和雪崩都是缓存失效问题）

什么是缓存穿透
缓存和数据库都查询不到这条数据
就是因为缓存中存储这些空数据的key&布隆过滤器
什么是缓存击穿（实质上是热点数据的缓存失效问题）
大量的请求同时查询一个key时，此时这个key正好失效了，就会导致大量的请求都打到数据库上面
第一个查询数据的请求上使用一个互斥锁来锁住它
什么是缓存雪崩（缓存大规模失效问题）
某一时刻发生大规模的缓存失效的情况
使用集群缓存，保证缓存服务的高可用，缓存失效时间分散
解决热点数据集中失效问题（就是缓存击穿问题）
热点的数据来说，当缓存失效以后会存在大量的请求
备份缓存&缓存降级&重构加锁

3 数据库和redis数据一致性问题（分布式多节点环境 & 单机环境）

单机场景下：写库成功写缓存失败导致数据不一致
解决方案：读缓存再读库，删缓存写库再写缓存

分布式环境：写读并发，1目标更新数据A为B 2目标读取数据A 1删缓缓存A——2读取缓存(null)——2读取数据库(A)——1修改数据库(B)——1更新缓存(B)——2更新缓存(A) 导致缓存数据库不一致最终库中存B,缓存存A
采用队列同步并发操作，根据ID配置队列，保证单个数据的查询修改行为是同步的，优化点，缓存未命中多个查询时，后续的查询不进入队列，自旋等待队列其他查询任务刷缓存

事务

1 事务的实现原理

undo&redo log = transaction log
Atomic:原子性：回滚日志（undo log）
Duration:持久性：一旦事务被提交，那么数据一定会被写入到数据库中并持久存储起来（redo log）
Isolation:隔离性：并发控制机制（锁/时间戳/多版本/快照）
Consistency:一致性：前三个都是为了保证最终的一致性
CAP:

- Consistency
- Availability
- Partition tolerance

它们的第一个字母分别是 C、A、P。

一致性和可用性，为什么不可能同时成立？答案很简单，因为可能通信失败（即出现分区容错）。
如果保证 G2 的一致性，那么 G1 必须在写操作时，锁定 G2 的读操作和写操作。只有数据同步后，才能重新开放读写。锁定期间，G2 不能读写，没有可用性不。如果保证 G2 的可用性，那么势必不能锁定 G2，所以一致性不成立。
综上所述，G2 无法同时做到一致性和可用性。系统设计时只能选择一个目标。如果追求一致性，那么无法保证所有节点的可用性；如果追求所有节点的可用性，那就没法做到一致性。

2 事务的四个关键属性(ACID)

A:原子性，成功或者失败
C:一致性，系统处于一致的状态，最终一致性保证
I:隔离性，事务的隔离级别
D:持久性：更改便持久的保存在数据库之中，并不会被回滚

ORM框架

1 ibatis是怎么实现映射的，它的映射原理是什么

1） SQL Map 将 Java 对象映射成 SQL 语句
2）结果集再转化成 Java 对象

2 hibernate和ibatis的区别

hibernate的o/r mapping实现了pojo（普通的Java Bean对象） 和数据库表之间的映射
ibatis实现了Javabean和sql之间的映射

小结

1 Nosql和关系型数据库的区别

（存储方式表&文档-图-键值对，存储是否结构化，存储扩展方式，事务规范，性能）
目标:应付超大规模，超大流量以及高并发的场景
1.存储方式（关系型-表格型，Nosql-数据集中，就像文档、键值对或者图结构）
2.存储结构(关系型-结构化数据，Nosql-动态结构，非结构化数据)
3.存储规范（关系型-数据分割到最小的关系表，Nosql-平面数据集，存在重复）
4.存储扩展（关系型-提升节点机器性能，Nosql-分布式集群规模扩张）
5.查询方式（关系型-结构化sql查询，Nosql-非结构化查询语言unql）
6.事务（关系型-ACID原子一致隔离持久原则，Nosql-Base原则基本可用柔性一致）
7.性能 （关系型-海量数据读写性能差，Nosql-内存级key-value效率高）
ps:图数据库：Neo4J
·节点与边线可以被赋予属性(键-值对);
·只有边线能够与类别相关联，例如"KNOWS";
·边线可以指定为有指向或无指向。

2 solr和mongodb的区别，存数据为什么不用solr?

都可以做非结构化文档存储。
mongodb 是典型的nosql ,相对于mysql 等传统的关系型数据库出现的，侧重于存储非结构化数据，
基于这些非结构化数据提供一些简单的查询。
solr 是典型的搜索系统，支持结构化数据的搜索，也支持非结构化数据的搜索，侧重点在于搜索。
我的理解：solr是以luence为基础的搜索引擎，solr中存储的数据Field会作为索引项存在，但是作为存储项，很多并不需要作为索引项，仅作存储展示使用，所以存数据不适合使用solr

3 Redis和memcache有什么区别？Redis为什么比memcache有优势？（存储方式，存储格式，集群管理方式）

1、存储方式：（memcache纯内存，redis是内存+硬盘，快照或者AOF文件的方式持久化）
2、数据支持类型：（redis比memcache支持的存储类型更多，memcache仅仅支持String）
3、集群管理方式不同：（redis支持分布式，memcache需要客户端自定义一致性hash算法来实现分布式的memcache集群）
PS：mongodb支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富。mongoDB支持master-slave,replicaset（内部采用paxos选举算法，自动故障恢复）,auto sharding机制，对客户端屏蔽了故障转移和切分机制。
MongoDB从1.8版本开始采用binlog方式支持持久化的可靠性。mongoDB不支持事务。mongoDB内置了数据分析的功能(mapreduce),其他两者不支持。MongoDB:主要解决海量数据的访问效率问题。

分布式

1 分布式全局唯一ID怎样来实现？

数据库自增；UUID（时间，时钟序列，机器识别号）；Snowflake算法（毫秒内序列squence提升毫秒并发量）;通过分布式锁实现全局唯一的ID。
优缺点：
自增：长度短，自增包含时间先后信息&大小等同于业务量，高并发存在锁竞争的问题，表合并复杂
uuid：string类型，应用层生成，不涉及数据库&无序，主键顺序存放uuid会产生随机io
snowflake：应用层，有顺序&机器号不好生成
分布式锁：唯一安全&锁竞争问题

2 分布式session如何实现的

一、Session Replication 方式管理 (即session复制广播)
简介：将一台机器上的Session数据广播复制到集群中其余机器上
使用场景：机器较少，网络流量较小
优点：实现简单、配置较少、当网络中有机器Down掉时不影响用户访问
缺点：广播式复制到其余机器有一定延时，带来一定网络开销
二、Session Sticky 方式管理（粘性session、修改session路由规则，session只会在单机处理）
简介：即粘性Session、当用户访问集群中某台机器后，强制指定后续所有请求均落到此机器上
使用场景：机器数适中、对稳定性要求不是非常苛刻
优点：实现简单、配置方便、没有额外网络开销
缺点：网络中有机器Down掉时、用户Session会丢失、容易造成单点故障
三、缓存集中式管理（缓存管理）
简介：将Session存入分布式缓存集群中的某台机器上，当用户访问不同节点时先从缓存中拿Session信息
使用场景：集群中机器数多、网络环境复杂
优点：可靠性好

缺点：实现复杂、稳定性依赖于缓存的稳定性、Session信息放入缓存时要有合理的策略写入

2 微服务你的理解？以及常用的微服务方案dubbo、spring cloud的比较？

微服务相对于分布式而言，粒度更小，每个服务承担指责更加单一，可以单独部署运行
数据传输不同，dubbo是二进制传输，sc是http协议传输
管理方式不同，dubbo使用zookeeper，sc使用eureka服务治理
是否跨平台，dubbo使用rpc不支持跨平台，sc使用restapi支持跨平台使用

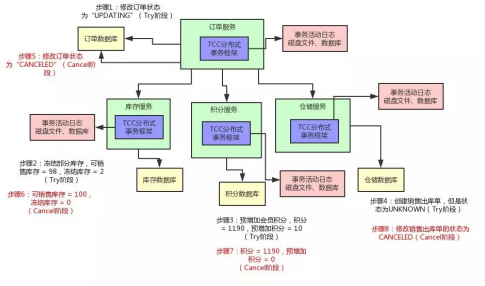
3 分布式锁的实现方式你知道有哪些？主流的解决方案是什么？

- 基于数据库实现（版本号的乐观锁，行级锁的悲观锁，表唯一互斥索引的增删）
- 基于Redis实现（非分布式：基本的key-value增删操作，分布式-Redlock，setnx和watch方法实现）
- 基于ZooKeeper实现（节点目录创建序号临时节点，exist()阻塞等待）

ps(zookeeper的实现):就在zookeeper上的某个指定节点的目录下，去生成一个唯一的临时有序节点， 然后判断自己是否是这些有序节点中序号最小的一个，如果是，则算是获取了锁。如果不是，则说明没有获取到锁，那么就需要在序列中找到比自己小的那个节点，并对其调用exist()方法，对其注册事件监听，当监听到这个节点被删除了，那就再去判断一次自己当初创建的节点点是否变成了序列中最小的。如果是，则获取锁，如果不是，则重复上述步骤。
当释放锁的时候，只需将这个临时节点删除即可。

4 描述分布式事务之TCC服务设计？(Try,comfrim,cancel)

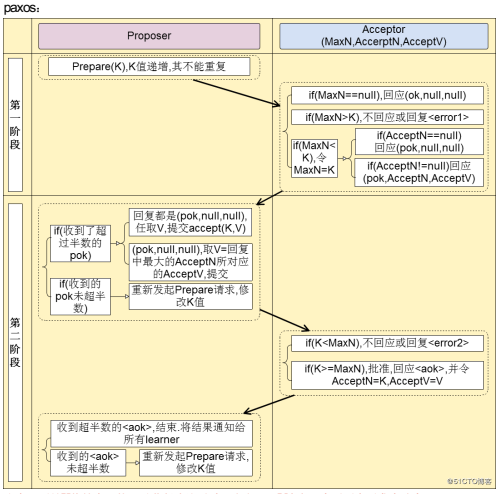
针对业务需要调用多个不同服务需要保证同时成功或者同时失败的情况的解决方案
其中Try操作作为一阶段，负责资源的检查和预留，Confirm操作作为二阶段提交操作，执行真正的业务，Cancel是预留资源的取消；
1.业务操作分两阶段(try阶段和comfirm阶段)
2.一阶段失败到达二阶段触发空回滚
3.一阶段超时但并未失败，在2的基础上要拒绝延时到达的try请求（防倒悬设置）
4.幂等控制，业务重复提交结果相同
5.业务数据可见性控制，try阶段的资源预留是否对其他事务可见
6.资源预留时的并发修改处理



http://www.cnblogs.com/jajian/p/10014145.html

5 分布式一致性协议raft，paxos了解吗

paxos：



注意：不是K值越大，就可以获得决议通过，当AcceptV确定下来后不会再发生改变。

https://www.jdon.com/artichect/raft.html （raft）

https://www.jdon.com/artichect/paxos.html （paxos）

6 微服务架构下，如果有一个订单系统，一个库存系统，怎么保证事务？(**)

1) XA协议的2PC提交
2) TCC模式（Try-Confirm-Cancel）
3) 消息队列MQ
4) 对账
5) GTS分布式事务中间件框架（TCC+消息队列）
（1和2两者都借助于事务协调器）

Attention：非关系型数据库不支持2PC&3PC方案
ps:两阶段提交2PC和三阶段提交3PC:
2PC:解决分布式事务的原子性问题，分为准备阶段和提交阶段，二阶段提交的算法思路可以概括为：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。
问题：1.同步阻塞问题。参与者执行事务锁定共享资源知道最重commit 2.单点故障，协调者的依赖性强，一阶段故障可以重选，二阶段会阻塞一阶段参与者已经占用的公共资源 3.数据不一致，二阶段commit没有全部完成
3PC:1、引入超时机制。同时在协调者和参与者中都引入超时机制。同时能在发生单点故障时，继续达成一致2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。也是为了减少同步阻塞的发生范围。
2PC和3PC都没有解决最终数据一致性的问题。

7 Zookeeper中的ZAB协议，选主算法

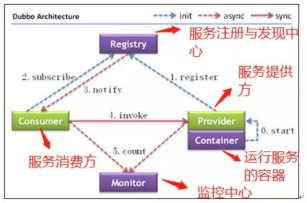
Zookeeper Atomic Broadcast ZooKeeper原子广播协议，支持 崩溃恢复 和 原子广播 协议
消息广播：类似于2PC两阶段提交，Leader接收到写请求生成proposal，proposal信息复制广播到zk-follower订阅的队列，follower从队列中取到proposal信息回复ack，leader收到半数以上zk-follower的ack后发送commit通知
ZAB 协议确保那些已经在Leader提交的事务最终会被所有服务器提交。（Leader完成了全部的提交事务）
ZAB 协议确保丢弃那些只在 Leader 提出/复制，但没有提交的事务。（Leader未提交，所有Follower都不会提交）
崩溃恢复（依赖于选举算法）
能够确保提交已经被 Leader 提交的事务，同时丢弃已经被跳过的事务。
Leader 选举算法能够保证新选举出来的Leader服务器拥有集群总所有机器编号（即 ZXID 最大）的事务，Leader接收到写请求后会生成提议proposal，同时为proposal生成全局唯一的递增的ID，谁拥有最大值的ZXID则意味着谁执行到了最新的proposal，知识越多的人越有机会成为Leader

8 介绍下PAXOS协议

9 介绍下Zookeeper的ZAB协议

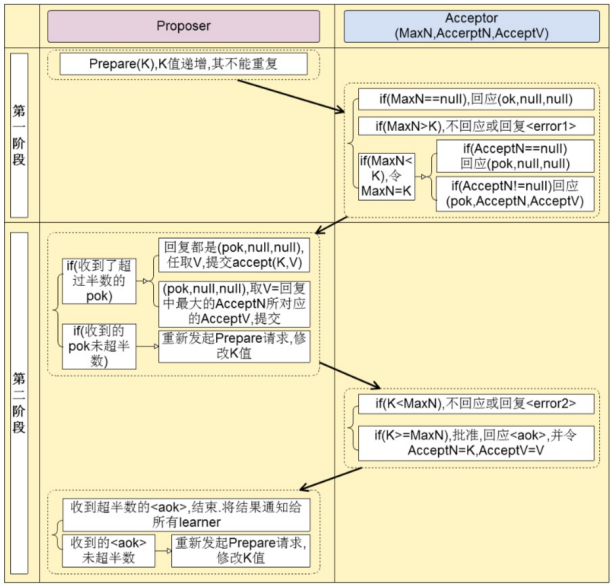
10 怎么进行服务注册发现 zk实现具体说说

服务生产者将自己提供的服务注册到Zookeeper中心，服务的消费者在进行服务调用的时候先到Zookeeper中查找服务，获取到服务生产者的详细信息之后，再去调用服务生产者的内容与数据。



11 分布式的paxos和raft算法了解么

paxos：多个proposer发请提议（每个提议有id+value），acceptor接受最新id的提议并把之前保留的提议返回。当超过半数的accetor返回某个提议时，此时要求value修改为propeser历史上最大值，propeser认为可以接受该提议，于是广播给每个acceptor，acceptor发现该提议和自己保存的一致，于是接受该提议并且learner同步该提议。



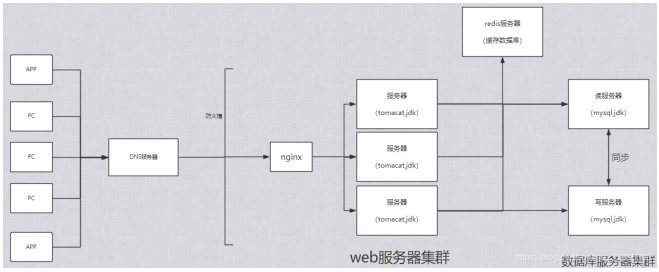
raft：raft要求每个节点有一个选主的时间间隔，每过一个时间间隔向master发送心跳包，当心跳失败，该节点重新发起选主，当过半节点响应时则该节点当选主机，广播状态，然后继续下一轮选主

12 怎样理解分布式服务

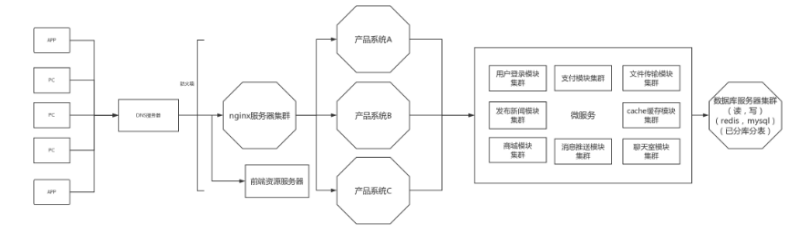
分布式是系统部署服务一种方式，将服务按照功能业务独立部署在对应的集群上，服务之间通过RPC方式完成沟通

微服务相对于分布式而言，粒度更小，每个服务承担指责更加单一，可以单独部署运行

13 典型的分布式架构图，从前端负载均衡到中间件，以及后端数据库，整个流程



微服务化



https://blog.csdn.net/qq_37856300/article/details/83714182

消息队列

1 Kafka怎么保证数据可靠性？

2 kafka和其他消息队列的对比

https://blog.csdn.net/lrxcmwy2/article/details/82846417

	Kafka	RabbitMQ	ActiveMQ	Redis
消息回溯	支持，无论是否被消费都会保留，可设置策略进行过滤删除（基于消息超时或消息大小）	不支持，一旦被确认消费就会标记删除	不支持	可设置消息过期时间，自动过期
API完备性	高	高	中	高
单机吞吐量	十万级	万级	万级	十万级
首次部署难度	中	低	低	低
消息堆积	支持	支持（内存堆积达到特定阈值可能会影响性能）	支持（有上线，当消息过期或存储设备溢出时，会终结它）	支持（有上线，服务器容量不够会容易崩溃造成数据丢失）
消息持久化（数据持久化到硬盘）	支持	支持	不支持	支持
多语言支持	支持，Java优先	语言无关	支持，Java优先	支持
消息优先级设置（某些消息被优先处理）	不支持	支持	支持	支持
消息延迟（消息被发送之后，并不想让消费者立刻拿到消息，而是等待特定时间后，消费者才能拿到这个消息进行消费）	不支持	支持	支持	支持
消费模式（①推模式：由消息中间件主动地将消息推送给消费者；②拉模式：由消费者主动向消息中间件拉取消息）	拉模式	推模式+拉模式	推模式+拉模式	拉模式
消息追溯	不支持	支持（有插件，可进行界面管理）	支持（可进行界面管理）	支持
常用场景	日志处理、大数据等	金融支持机构	降低服务之间的耦合	共享Cache、共享Session
运维管理	有多种插件可进行监控，如Kafka Management	有多种插件可进行监控，如rabbitmq_management(不利于做二次开发和维护)	无	有多种插件可进行监控，如zabbix

IO

1 IO模型有哪些（IO操作分为两个阶段，等待数据准备完毕，从内核拷贝数据到对应的进程当中）

a)BIO（数据准备和拷贝过程中进程或者线程都处于阻塞的状态）

b)NIO（仅在数据拷贝阶段堵塞，通过轮询的方式完成与数据准备的通信，轮询过程为非阻塞的状态）

c)IO多路复用（事件驱动IO，通过select/epoll不断轮询负责的socket，当socket有数据到达通知相应的线程处理，对于连接数高的服务适合使用，使用Reactor设计模式）

d)信号驱动IO（当数据报准备好的时候，给我发送一个信号，对SIGIO信号进行捕捉，并且调用我的信号处理函数来获取数据报）

e)异步IO（数据准备和数据拷贝阶段均不阻塞，基于异步通知实现）

划分方案2（程序执行的同步和异步【是否存在通知】，读取写入数据的阻塞和非阻塞）

a)同步阻塞io，bio是同步阻塞io，对应java中最普通的io操作

b)同步非阻塞io，nio是同步非阻塞io，对应java中的nio操作

c)异步阻塞io，io多路复用就是一种异步阻塞io，select系统调用是阻塞的，reactor线程的执行是异步的

d)异步非阻塞io

3 BIO和NIO的应用场景

（1）NIO适合处理连接数目特别多，但是连接比较短（轻操作）的场景，Jetty，Mina，ZooKeeper等都是基于java nio实现。服务器需要支持超大量的长时间连接。比如10000个连接以上，并且每个客户端并不会频繁地发送太多数据。

（2）BIO方式适用于连接数目比较小并且一次发送大量数据的场景，这种方式对服务器资源要求比较高，并发局限于应用中。

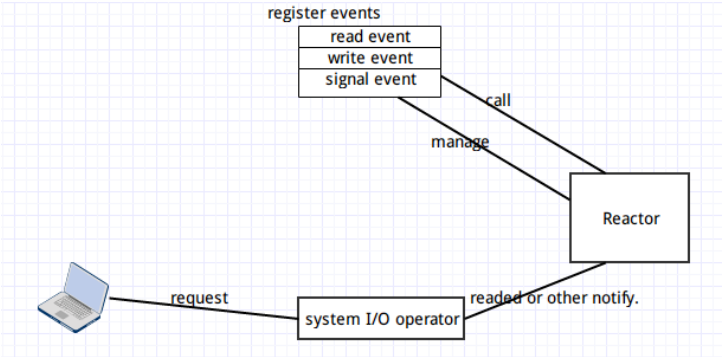
https://www.cnblogs.com/zedosu/p/6666984.html（BIO和NIO的理解）

4 什么叫做io多路复用

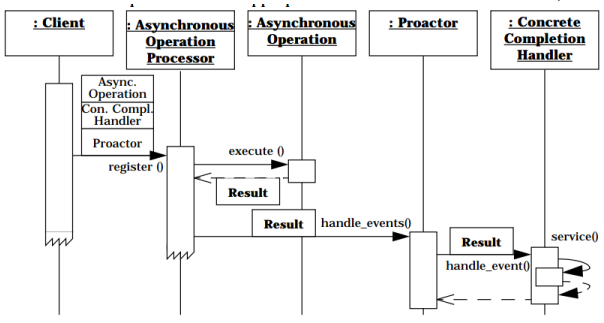
同一个线程内同时处理多个io请求,select函数通知线程执行数据拷贝读取任务前，线程可以做其他的事情，减少了资源浪费。但是每个io请求本身还是阻塞的，在select函数上阻塞

5 io的两种设计模式（Reactor同步IO，Proactor异步IO）

Reactor反应堆设计模式：用户线程注册事件处理器HandlerEvent到Reactor线程上，Reactor线程负责调用内核select函数检查socket状态，socket数据到达，为可读状态时，reactor线程通知用户线程的回调接口对相关的socket发起读取请求，拷贝相关数据到对应的线程上进行业务处理



Proactor主动器设计模式，用户线程将异步操作AsynchronousOperation，主动器Proactor，以及完成处理器CompletionHandler注册到同步操作处理器AsynchronousProcessor，异步操作处理器AsynchronousProcessor提供io操作的api，用户线程调用api继续执行其他的任务，异步操作处理器使用单独的内核线程执行异步IO操作，异步io操作完成后取出Proactor和CompletionHandler，将io操作结果和completionhandler一同转发给Proactor，Proactor回调完成处理器的handle_event方法



4 Netty对于nio的实现

netty是nio的封装，

5 nio的io多路复用(用一个线程去管理多个socket IO通信)

在多路复用IO模型中，会有一个线程（Java中的Selector）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

BIO&IO：多socket多线程，每个线程绑定一个socket，read无资源或者write过程中会持续进行，可能存在阻塞等待的情况

NIO：采用一个线程管理多个socketIO通信的方式，借助Selector的轮询的方式监督每个socket,socket在读等待或者写的间隙可以参与其他的工作，不阻塞

AIO：read和write请求采用缓冲区异步实现，异步非阻塞IO

6 说一下NIO的类库或框架

讲了netty，写过服务端和客户端的demo，没有在生产中实践。

1 channelhandler负责请求就绪时的io响应。

2 bytebuf支持零拷贝，通过逻辑buff合并实际buff。

3 eventloop线程组负责实现线程池，任务队列里就是io请求任务，类似线程池调度执行。

4 acceptor接收线程负责接收tcp请求，并且注册任务到队列里。

7 netty的线程模型是什么样的

网络WEB

1 https和http区别，有没有用过其他安全传输手段？

HTTP：超文本传输协议，http是一个客户端和服务器端请求和应答的标准，是TCP协议上的应用

HTTPS：安全套接字层超文本传输协议，在http下加入ssl验证层协议。

安全性不同：http(明文)<https

端口不同：http(80),https(443)

证书申请：https需要申请证书

2 tcp&udp

1.基于连接（tcp）与无连接（udp）；

2.首部开销，tcp20字节，udp8字节

3.Tcp是点对点的，UDP则是1对多，多对一等

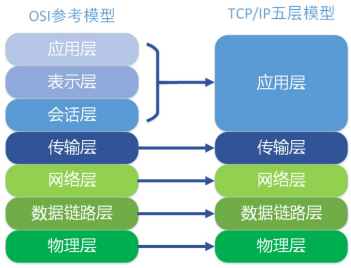
4.流模式与数据报模式；TCP(字节流) UDP（数据报文）

5.TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证。

应用层协议	应用	传输层协议
SMTP	电子邮件	TCP
TELNET	远程终端接入	
HTTP	万维网	
FTP	文件传输	
DNS	名字转换	UDP
TFTP	文件传输	
RIP	路由选择协议	
BOOTP，DHCP	IP 地址配置	
SNMP	网络管理	
NFS	远程文件服务器	
专用协议	IP 电话	

tcp三次握手：SYN_SEND&SYN_RECV&ESTABLISHED

3 OSI七层模型&TCP/IP五层模型



4 Cookie和Session的区别（Cookie客户端浏览器缓存内容，Session服务端标记会话的内容）

位于用户的计算机上，用来维护用户计算机中的信息，直到用户删除。比如我们在网页上登录某个软件时输入用户名及密码时如果保存为cookie，则每次我们访问的时候就不需要登录网站了。我们可以在浏览器上保存任何文本，而且我们还可以随时随地的去阻止它或者删除。我们同样也可以禁用或者编辑cookie，但是有一点需要注意不要使用cookie来存储一些隐私数据，以防隐私泄露

session称为会话信息，位于web服务器上，主要负责访问者与网站之间的交互，当访问浏览器请求http地址时，将传递到web服务器上并与访问信息进行匹配，当关闭网站时就表示会话已经结束，网站无法访问该信息了，所以它无法保存永久数据，我们无法访问以及禁用网站

5 Cors(跨域请求), cxrf漏洞(跨站请求模拟，cookie共享)，Xss攻击类型：

Xss:反射型（请求在URL当中）、DOM-based 型客户端（Dom解析产生）、存储型（数据存储）

解决方案：DOM-based型，关键字过滤，html字符编码 存储型：html，js编码

Cors:对跨域的域名限制

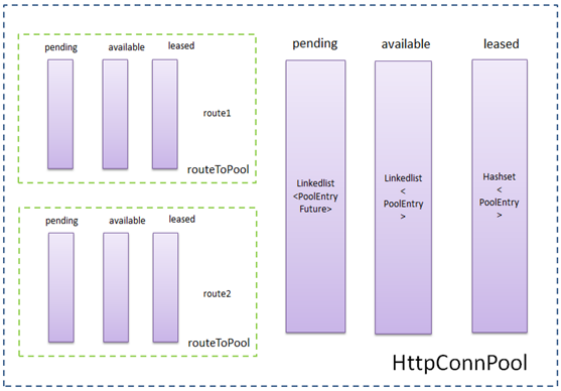
Csrf:加refer验证，token，图形验证码

6 HTTP（Client）连接池实现原理

原因：降低延迟，提高并发，http需要简历tcp链接，三次握手，四次挥手

https://blog.csdn.net/umke888/article/details/54881946

连接管理器&请求相关配置&重试策略



PoolEntry<HttpRoute, ManagedHttpClientConnection> -- 连接池中的实体（HttpRoute是route路由信息&ManagedHttpClientConnection是httpClient链接）
（1）LinkedList<PoolEntry> available -- 存放可用连接PoolEntry
（2）HashSet<PoolEntry> leased -- 存放已经被使用的PoolEntry
（3）LinkedList<PoolEntryFuture> pending -- 存放等待获取连接的线程的Future，类似与线程池中的等待队列，await()阻塞
routeToPool -- 每个路由对应的pool（包含以上（1），（2），（3）），针对不同的路由能有自己不同的配置，例如每个路由的最大链接数都是隔离的
ps:简单的来说routeToPool里面的链接是route私有的，routeToPool外面的链接是公用的
1、使用连接池时，要正确释放连接需要通过读取输入流 或者 instreamclose()方式；
2、如果已经释放连接，response.close()直接返回，否则会关闭连接；
3、httpClient.close()会关闭连接管理器，并关闭其中所有连接，谨慎使用。

7 HTTP协议

参考51题目

https://www.cnblogs.com/ranyonsue/p/5984001.html

HTTP协议是Hyper Text Transfer Protocol（超文本传输协议)&属于TCP协议族当中&属于应用层的面向对象的协议&工作于客户端-服务端架构为上

URI和URL的区别
URI，是uniform resource identifier，统一资源标识符，用来唯一的标识一个资源。
URL是uniform resource locator，统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源
状态码：
1xx：指示信息--表示请求已接收，继续处理
2xx：成功--表示请求已被成功接收、理解、接受
3xx：重定向--要完成请求必须进行更进一步的操作
4xx：客户端错误--请求有语法错误或请求无法实现
5xx：服务器端错误--服务器未能实现合法的请求

8 负载均衡

◎ 依序Round Robin 轮询
◎ 比重Weighted Round Robin
◎ 流量比例Traffic
◎ 使用者端User
◎ 应用类别Application
◎ 联机数量Session
◎ 服务类别Service
◎ 自动分配Auto Mode

9 正向代理（客户端代理）和反向代理（服务器端代理） 正向代理 shadowsocks 反向代理 nginx

正向代理，代理客户端，服务端不知道实际发起请求的客户端； 代理客户访问原始服务。shadowsocks代替客户访问达到翻墙目的。
反向代理，代理服务端，客户端不知道实际提供服务的服务端； 代理原始服务为客户服务。nginx
正向代理，为在防火墙内的局域网客户端提供访问Internet的途径；
反向代理，将防火墙后面的服务器提供给Internet访问；

10 CDN实现原理

CDN的全称是Content Delivery Network，即内容分发网络
普通：url请求——浏览器域名解析——IP访问服务器——服务主机返回数据（反向相同）
CDN：url请求——浏览器域名解析（CDN调整解析函数获取CNAME）——CNAME负载均衡访问最近的Cache缓存服务器——有缓存数据直接返回，无通过CNameDNS解析访问IP服务器——服务主机返回数据——缓存服务器缓存同时返回数据

11 怎么提升系统的QPS和吞吐量

集群+负载均衡&增加缓存&系统拆分&分库分表 垂直拆分+水平拆分&异步化+MQ

12 DNS的实现原理

（两阶段，DNS客户端上报DNS服务器域名解析获取IP，浏览器根据Ip发起http链接）

DNS:域名系统，将主机名，域名转化为IP
①主机上运行DNS的客户端
②浏览器抽域名字段，就是访问的主机名，提交DNS应用的客户端
③DNS客户机端向DNS服务器端发送一份查询报文
④DNS客户端收到域名对应IP的响应报文，发给浏览器
⑤浏览器向IP地址定位的HTTP服务器发起TCP连接
所有DNS请求和回答报文使用的UDP数据报经过端口53发送，根DNS服务器，顶级DNS服务器，权威DNS服务器。

13 拥塞窗口讲一讲，为什么要用慢启动算法（慢慢尝试增大发送量，空值合理的发送速率）

拥塞窗口：“拥塞窗口”就是“拥塞避免”的窗口，它是一个装在发送端的可滑动窗口，窗口的大小是不超过接收端确认通知的窗口。“慢速启动”是在连接建立后，每收到一个来自收端的确认，就控制窗口增加一个段值大小，当窗口值达到“慢速启动”的限值后，慢速启动便停止工作，避免了网络发生拥塞
慢启动算法：避免出现网络拥塞，慢启动算法就是在主机刚开始发送数据报的时候先探测一下网络的状况，如果网络状况良好，发送方每发送一次文段都能正确的接受确认报文段。那么就从小到大的增加拥塞窗口的大小，即增加发送窗口的大小。
https://baike.baidu.com/item/%E6%8B%A5%E5%A1%9E%E7%AA%97%E5%8F%A3/4399307?fr=aladdin

14 http报文头部是什么（不同版本处理方式不同）

1.0，1.1，2.0的区别：
1）连接方面
HTTP1.0使用非持久连接,即在非持久连接下,一个tcp连接只传输一个Web对象。
HTTP1.1默认使用持久连接 一个连接中可以传输多个对象，减少了建立和关闭连接的消耗和延迟。
2）缓存方面
HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准
HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略带宽优化及网络连接的使用

3）状态码
在HTTP1.1中新增了24个错误状态响应码
4）带宽优化
HTTP 1.1 可以先发header验证再发送body数据。正常验证返回100，否则401
5）Host头
HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，url无主机名，但是物理服务器上可以存在多个虚拟主机，共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，精准访问虚机
HTTP1.1 VS HTTP2.0
（1）多路复用：
在HTTP/1.1协议中，浏览器客户端在同一时间针对同一域名的请求有一定数据限制。超过限制数目的请求会被阻塞。
HTTP2.0使用多路复用的技术，做到同一个连接并发处理多个请求，并发量增加
当然HTTP1.1也可以建立多个TCP连接，来支持处理更多并发的请求，但是创建TCP连接本身也是有开销的。TCP连接有一个预热和保护的过程，因此对应瞬时并发的连接，服务器的响应就会变慢。最好使用一个建立好的连接，并且这个连接可以支持瞬时并发的请求。
（2）首部压缩：
HTTP1.1不支持header数据的压缩，HTTP2.0使用HPACK算法对header的数据进行压缩，这样数据体积小了，在网络上传输就会更快。
（3）服务器推送：
当我们对支持HTTP2.0的web server请求数据的时候，服务器会顺便把一些客户端需要的资源一起推送到客户端，免得客户端再次创建连接发送请求到服务器端获取。这种方式非常合适加载静态资源。

15 web请求的过程，讲了浏览器到http服务器的过程，再讲了mvc的请求处理过程。
DNS域名解析-TCP连接建立-发起request请求-后端服务器处理request返回处理结果-response
请求收到-断开连接

16 http get和post有什么区别
参数传递方式不同，url或者body
传参长度限制，get有限制，post无显示
get一般用于获取，post一般用于提交，作用域不同
编码方式不同，get需要通过url编码
都是http请求
Restful API：
GET用于查询，PUT、POST、DELETE用于修改,X-HTTP-Method-Override添加标注
使用名词而不使用动词
在HTTP请求的head体里定义序列化类型
请求的集合应设定好过滤条件、排序、字段、分页

17 什么是DoS、DDoS、DRDoS攻击？如何防御？
DoS：拒绝服务
DDoS：分布式拒绝服务，采用肉鸡的方式
DRDOS：分布反射式拒绝服务，伪装目标计算机广播，回应数据全部回应到目标计算机

18 http链接的keepAlive参数的含义
Keep-Alive功能避免了建立或者重新建立连接（http1.0默认关闭，http2.0默认关闭）

LINUX命令

1 linux怎么查看系统负载情况？
uptime/top/w/vmstat等
https://blog.csdn.net/qq_36357820/article/details/76606113
https://blog.csdn.net/sycflash/article/details/6643492

具体功能模块设计

1 如果有几十亿的白名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能。
明确问题，白名单需要判断的就是存在或者不存在，转化为去重的问题
高并发的查询：布隆过滤器（bitmap位图） 晚上的更新：双内存切换，减少切换过程中发生的问题

2 新浪微博是如何实现把微博推给订阅者
不懂这题的意思，更像是说协同过滤的意思

3 Google是如何在一秒内把搜索结果返回给用户的。
CND缓存&客户端缓存&分词索引&搜索框架内存级别的存储

4 12306网站的订票系统如何实现，如何保证票不被超卖。
和5题目类似

5 如何实现一个秒杀系统，保证只有几位用户能买到某件商品。
倒计时&nginx限流&独立服务（安全隔离）&队列解耦&延迟交付
1 下单接口延迟暴露，页面开启倒计时
2 nginx限流，减少后端处理的并发量
3 后端服务单独部署，从nginx到域名到服务集群，与线上其他业务隔离
4 队列解耦下单过程，下单成功（立即返回成功结果）和生成订单（异步消息推送）分离，针对进入后端的请求，采用redis自减控制下单成年公个数，redis记录下单成功用户id，value未订单生成状态
5 用户方面，收到下单成功通知延迟等待订单生成
6 订单服务订阅生成订单消息，生成订单成功之后更新redis中用户value订单状态

6 10G的log里面每一行都保存着一个url，内存只有250M
1.首先，考虑将10G的log文件划分为多个小于250M的文件，这样每个小文件就可以一次性载入内存了。
2.当小文件可以一次性载入内存后，可以直接grep搜索，也可以对文件内容排序后，然后二分查找。
或者
复杂的预处理：（1）文件分割（2）分割数据逐条hash再分隔 ，变成有规律的层状查询结构

7 如何设计一个流控功能？
监控指数：A:接口级别的-并发数，QPS、B:整体的流量分接口的负载降级
配置方式：分布式应用程序协调服务对接口扣流控参数统一集中管理推送。
限流算法：令牌桶
是否适应削峰填谷的策略。

8 写一个转账（从账户A到账户B的转账一笔资金）的伪代码，需要考虑事务。如何解决死锁的问题？如何解决热点账户的问题？
```
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
UPDATE Accounts
SET CurrentBalance=CurrentBalance-Amount  WHERE AccountID=A;
```

UPDATE Accounts
SET CurrentBalance=CurrentBalance+Amount WHERE AccountID=B;
if error then ROLLBACK;
COMMIT;
第一招就是 减少事务时间，具体方法就是，减少事务的数据库操作，讲一些不太重要的操作进行分解。
第二招 优化网络,,其实就是减少了应用到数据库的交换机，给高并发应用配置资源
第三招 异步 记录交易流水 走消息队列
第四招 将一个热点账户拆分为多个热点账户

9 实现一小段用户登录验证小逻辑

1）需要考虑登录安全与用户信息安全
2）用户信息是维护在文件中
3）若需要支持10000+同时在线登录，依然将用户信息维护在文件中，如何实现？

10 双11这样的并发流量如何确保服务的可用性

11 问如果量级扩大1000倍，你会怎么做？有哪些优化措施？高性能&高可用措施？

12 场景，同时给10万个人发工资，怎么样设计并发方案，能确保在1分钟内全部发完

配置线程池，根据每个线程每分钟能执行发工资的个数配置线程池大小。
考虑共享账号的问题，发工资使用同一个账号时对账号数据进行切割处理，分段锁减少并发数。
另外还要考虑线程开销问题，要在资源满足最大支持线程数创建线程池

系统安全

1 服务降级（过载保护）&熔断策略（熔断器）的设计

1. 服务熔断（下游服务）故障引起，而服务降级一般是从整体负荷考虑；
2.管理目标的层次不太一样，熔断其实是一个框架级的处理，每个微服务都需要（无层级之分），降级与业务相关
3.实现方式不太一样

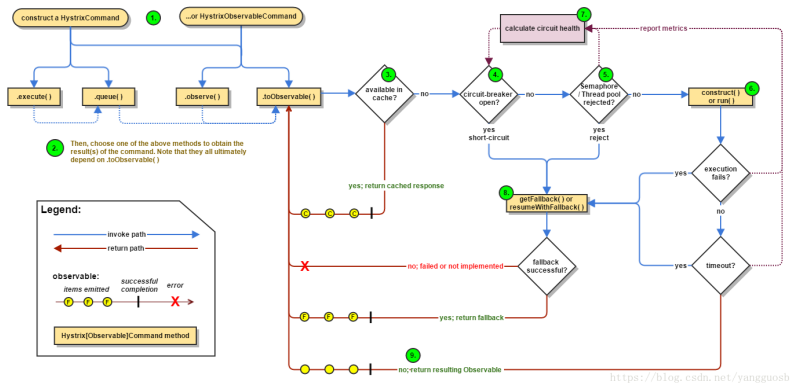
hystrix

1 从hystrix一路问到原理->自己如何实现->如何优化->响应流编程(reactive streams);

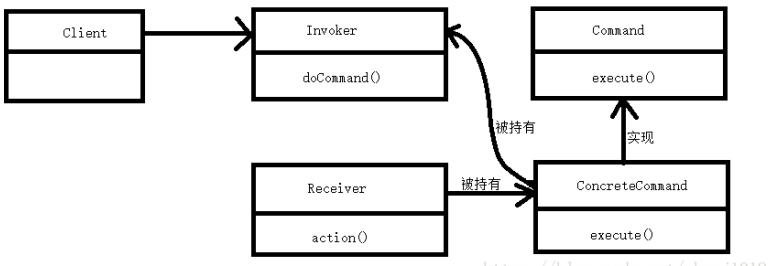
Hystrix提供了熔断、隔离、Fallback、cache、监控等功能，能够在一个、或多个依赖同时出现问题时保证系统依然可用。

- 使用命令模式将所有外部服务的调用包装在HystrixCommand或HystrixObservableCommand对象中，并将对象放在单独线程中执行；
- 每个调用都维护着一个线程池（或信号量），线程池被耗尽则拒绝请求（而不是让请求排队）。
- 记录请求成功，失败，超时和线程拒绝。
- 服务错误百分比超过了阈值，熔断器开关自动打开，一段时间内停止对该服务的所有请求。
- 请求失败，被拒绝，超时或熔断时执行降级逻辑。
- 近实时地监控指标和配置的修改。

Spring cloud Hystrix:
服务降级：
服务熔断：
线程和信号隔离：
请求缓存：
请求合并：
服务监控：



命令模式：



https://blog.csdn.net/zhwyj1019

```
//接收者
public class Receiver {
 public void action() {
  //真正的业务逻辑
 }
}
//抽象命令
interface Command {
 void execute();
}
//具体的命令实现类
public class ConcreteCommand implement Command {
 private Receiver receiver;
```

```
 public ConcreteCommand(Receiver receiver) {
  this.receiver = receiver;
 }
 @Override
 public void execute() {
  this.receiver.action();
 }
}
//客户端调用者
public class Invoker {
 private Command command;
 public void setCommand(Command command) {
  this.command = command;
 }
 public void execute() {
  this.command.action();
 }
}
//客户端
public class Client {
public static void main(String[] args) {
 Receiver receiver = new Receiver();
 Command command = new ConcreteCommand(receiver);
 Invoker invoker = new Invoker();
 invoker.setCommand(command);
 invoker.action(); //客户端通过调用者来执行命令
 }
}
```

RxJava:发布订阅者模型，观察者模式

链式编程模式

```
        //创建一个上游 Observable:
        Observable<Integer> observable = Observable.create(new ObservableOnSubscribe
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exceptio
                emitter.onNext(1);
                emitter.onNext(2);
                emitter.onNext(3);
                emitter.onComplete();
            }
        });
        //创建一个下游 Observer
        Observer<Integer> observer = new Observer<Integer>() {
            @Override
            public void onSubscribe(Disposable d) {
                Log.d(TAG, "subscribe");
            }

            @Override
            public void onNext(Integer value) {
                Log.d(TAG, "" + value);
            }

            @Override
            public void onError(Throwable e) {
                Log.d(TAG, "error");
            }

            @Override
            public void onComplete() {
                Log.d(TAG, "complete");
            }
        };
        //建立连接
        observable.subscribe(observer);
```

Observable：被观察者

ObservableEmitter：Emitter是发射器的意思，那就很好猜了，这个就是用来发出事件的，它可以发出三种类型的事件，通过调用emitter的onNext(T value)、onComplete()和onError(Throwable error)就可以分别发出next事件、complete事件和error事件。

Observer：观察者，实现onNext(T value)、onComplete()和onError(Throwable error)方法，与ObservableEmitter方法一一对应，一个发送一个接收

HytrixCommand提供两种方法：

HytrixCommand的excute和queue分别对应同步和异步，excute直接返结果，queue放回future对象

```java
public class CommandHelloWorld extends HystrixCommand<String> {

    private final String name;

    public CommandHelloWorld(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        // a real example would do work like a network call here
        return "Hello " + name + "!";
    }
}
```

通过上面实现的CommandHelloWorld，既可以实现请求的同步执行也可以实现异步执行。

同步执行示例：

```java
String s = new CommandHelloWorld("World").execute();
```

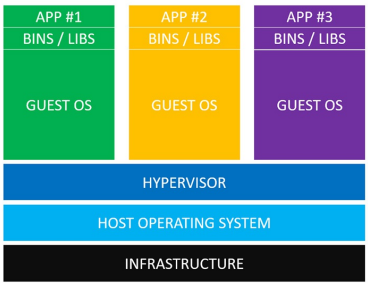
异步执行示例：

```java
Future<String> fs = new CommandHelloWorld("World").queue();
String s = fs.get();
```

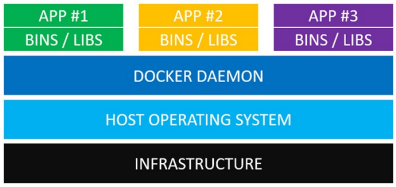
HytrixObservableCommand提供两种方法：
observe()：对应Hot-Observable 发布端发布消息，订阅端加入后订阅当时发布以后的消息
toObservable()：对应Cold-Observable 无订阅端发布端不发布消息，订阅端加入后需要消费发布端发布的全部消息包括历史消息
HytrixCommand&HytrixObservableCommand的区别：
（1）HytrixCommand提供了同步和异步两种方案，HytrixObservableCommand只有异步的方式
（2）HystrixCommand的run方法是用内部线程池的线程来执行的，而HystrixObservableCommand则是由调用方(例如Tomcat容器)的线程来执行的
（3）HytrixObservableCommand获取能发射多次的Observable，通过OnNext完成事件的多次发送

docker

1 Docker和虚拟机的区别



虚拟机（从下往上）：基础设施电脑PC，用户操作系统，虚拟机管理系统，三个虚拟机应用



Docker（从下往上）：基础设施电脑PC，用户操作系统，docker守护进程，三个应用
Docker守护进程可以直接与主操作系统进行通信，为各个Docker容器分配资源；
它还可以将容器与主操作系统隔离，并将各个容器互相隔离
区别：
docker不需要再每个应用简历独立的操作系统，占用内存，磁盘空间小
虚拟机是运行环境的彻底隔离
虚拟机应用：云服务比如阿里云 docker应用：前端后端数据库系统的隔离

2 讲一下docker的实现原理

我的项目

字体加密

1 字体加密产生背景，需要解决的问题

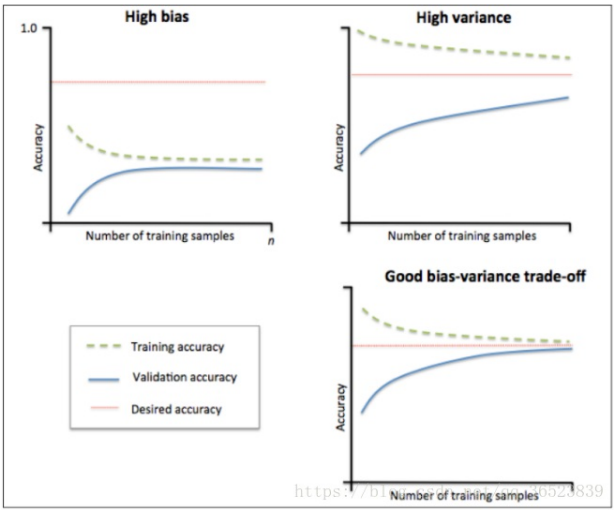
背景：业务层面上用户隐私的泄露，技术层面上对于服务器压力的降低
已有解决方案：传统的后端反抓取策略面对趋近于真实用户的机器行为不能有效判别，同时存在误伤
新方案的目标：应对web网站爬虫在网页源码级别的加密的解决方案

2 字体加密的实现方案

字体文件的组成：编码，图元（多个数组组成，每个数组代表了直线或者贝塞尔曲线）
字体构建层面：通过nodejs开源包解源文件微软雅黑字体文件，抽取所需要的文字域，修改原始图元编码为自定义的编码区间并打乱图元顺序，针对图元数组做长宽位置的微调，添加无用曲线修改绘图数组的大小
页面应用层面上：服务器端缓存定量套数的字体文件，定时更新，浏览器调用每次随机抽取缓存字体库中的其中一套字体，保证每次访问编码的随机性

3 字体加密的实现效果

用户浏览器浏览行为基本不受影响，文字会有细微的微调，爬虫爬取同一页面每次关键数据域的编码都会发生变化



先来看看如何解析学习曲线图：

要看深刻了解上面的图形意义，你需要了解偏差（bias）、方差（variance）对于训练模型的意义，可以参考这里，当你了解后，我们来看看上面的图形代表的意义：（横坐标表示训练样本的数量，纵坐标表示准确率）

1：观察左上图，训练集准确率与验证集准确率收敛，但是两者收敛后的准确率远小于我们的期望准确率（上面那条红线），所以由图可得该模型属于欠拟合（underfitting）问题。由于欠拟合，所以我们需要增加模型的复杂度，比如，增加特征、增加树的深度、减小正则项等等，此时再增加数据量是不起作用的。

2：观察右上图，训练集准确率高于期望值，验证集则低于期望值，两者之间有很大的间距，误差很大，对于新的数据集模型适应性较差，所以由图可得该模型属于过拟合（overfitting）问题。由于过拟合，所以我们降低模型的复杂度，比如减小树的深度、增大分裂节点样本数、增大样本数、减少特征数等等。

3：一个比较理想的学习曲线图应当是：低偏差、低方差，即收敛且误差小。

7 什么是深度学习，它与机器学习算法之间有什么联系

深度学习是机器学习的一个子领域，它关心的是参照神经学科的理论构建神经网络

8 如何处理一个不平衡的数据集

收集更多的数据，使其达到平衡

使用重复采样

使用不同的算法

9 你如何确保你的模型没有过拟合（参考第6题）

参数太多，模型复杂度过高

获取额外数据进行交叉验证

7.如何评估你的机器学习模型的有效性

训练集和测试集，或者使用给交叉验证方法

混淆矩阵具体参数的判断，ROC曲线

https://baijiahao.baidu.com/s?id=1621804908085622890&wfr=spider&for=pc

杂项

1 集群、负载均衡、分布式、数据一致性的区别与关系

集群：一组相互独立的计算机，高性能，性价比，可伸缩

负载均衡：合理分配任务到集群的每一台服务器

分布式：不同的业务分布在不同的节点（每个节点都可能是集群）

在分布式集群中存在的数据一致性问题，单节点采用锁的方式 分布式则是采用Paxos算法（类似与一种选举投票算法）

2 最后谈谈Redis、Kafka、 Dubbo，各自的设计原理和应用场景

Redis:缓存数据库

Kafka:消息队列

Dubbo:整合了服务注册发现，容错负载均衡策略的RPC框架

3 面向对象（行为动作和属性的封装叫做对象，对象的分组叫做类）

把数据及对数据的操作方法放在一起，作为一个相互依存的整体——对象。对同类对象抽象出其共性，形成类。类中的大多数数据，只能用本类的方法进行处理。类通过一个简单的外部接口与外界发生关系，对象与对象之间通过消息进行通信。程序流程由用户在使用中决定。

4 面向过程（步骤函数依次执行）

自顶向下顺序执行，逐步求精；其程序结构是按功能划分为若干个基本模块，这些模块形成一个树状结构；各模块之间的关系尽可能简单，在功能上相对独立；每一模块内部均是由顺序、选择和循环三种基本结构组成；其模块化实现的具体方法是使用子程序。程序流程在写程序时就已决定。

5 如何搭建一个高可用系统

1，主备/集群模式，防止单点

2，限流，削峰，防止后端压力过大

3，熔断机制，类似与限流

4，容灾机制，多机房/异地部署

6 哪些设计模式可以增加系统的可扩展性

工厂模式 构造函数的切面编程

观察者模式：（Subject内部添加Observer成员变量）很方便增加观察者，方便系统扩展

模板方法模式：（抽象类封装模板方法，实现类进行抽象方法的重写实现）很方便的实现不稳定的扩展点，完成功能的重用

适配器模式：可以很方便地对适配其他接口

代理模式(AOP)：可以很方便在原来功能的基础上增加功能或者逻辑

责任链模式：可以很方便得增加拦截器/过滤器实现对数据的处理，比如struts2的责任链

策略模式：通过新增策略从而改变原来的执行策略

装饰模式：二次封装

7 介绍设计模式，如模板模式，命令模式，策略模式，适配器模式、桥接模式、装饰模式，观察者模式，状态模式，访问者模式。

状态模式：状态注入到实体类，实体类执行方法时按照状态执行

访问者模式：

8 抽象能力，怎么提高研发效率。

任务执行，分类是否可分解，分解后是否可并发

具体方案，造轮子还是用轮子还是装饰轮子

扩展性探究

9 什么是高内聚低耦合，请举例子如何实现

高内聚（单一责任原则） 低耦合（模块之间的依赖相关性少）
分层扩展，继承和多态，就是保证高内聚低耦合的一种设计手段。

10 什么情况用接口，什么情况用消息（接口同步，消息异步）

接口的特点是同步调用，接口实时响应，阻塞等待（实时）
消息的特点是异步处理，非实时响应，消息发送后则返回，消息队列可以削峰（非实时）

11 如果AB两个系统互相依赖，如何解除依赖（引入第三方依赖，类似于ICO）

借助第三方C，环路依赖

12 如何写一篇设计文档，目录是什么

背景&设计思路&技术选型&整体框架设计图&分模块详解&性能与扩展

简要文档功能背景说明&技术选型&整体技术设计图&分模块详解&性能和扩展

13 什么场景应该拆分系统，什么场景应该合并系统

拆分：集群性能跟不上，业务场景复杂
合并：跨进程访问消耗高，进程间一致性处理复杂
拆分：集群方式不能解决性能问题（按性能点拆分，爬虫系统爬取器网络IO大，数据解析结构化CPU消耗高），业务场景复杂（按照业务拆分，庞大的系统按照业务功能拆分）

合并：系统间通过跨进程访问的性能损耗过高，可以将系统合并成一个系统，减少跨进程访问的消耗

14 系统和模块的区别，分别在什么场景下使用

系统：完整功能，完整的生命周期，系统由模块组成
模块：不能单独工作，单一方面问题解决的单元

15 如何保证消息的一致性

实时：ack同步 非实时：最终一致性校验：对账：

16 有没有处理过线上问题？出现内存泄露，CPU利用率飙高，应用无响应时如何处理的。

内存泄露问题：
1）jmap定位是否是堆内还是堆外内存溢出
2）堆内查看GC情况， jstat -gc pid，jstat -gcutil pid统计GC信息
2.1）jmap dump堆内存分析，查看内存泄露在哪个类以及调用链情况
3）堆外内存泄露采用google的开源gperftools定时打印内存监控使用日志
3.1）找到占用资源大的JNI方法
3.2）btrace定位JNI方法的调用链（AOP编程）

CPU利用率飙高问题定位：
1）通过top命令找到CPU占用高的进程
2）通过PS命令定位CPU飙高的线程ps H -eo user,pid,ppid,tid,time,%cpu,cmd –sort=%cpu
3）获取线程id对应的16进制编码
4）jstack grep对应编码找到对应的线程，定位问题

应用无响应问题：
1）判断请求是否到达
2）追溯后端处理的整个调用链，找到耗时位置
3）定位最终问题

17 开发中有没有遇到什么技术问题？

字体加密，爬虫插件，ID加密

18 如何学习一项新技术，比如如何学习Java的，重点学习什么

明确学习目的，为什么要学习这样新技术
构建学习流程图，明确学习步骤
划分小目标，规划小的里程碑
坚持克服惯性，客服懒惰

19 有关注哪些新的技术

arthas线上监控

20 工作任务非常多非常杂时如何处理

优先级排序，按照重要紧急两要素四象限处理。重要紧急>重要不紧急=紧急不重要>不重要不紧急

21 项目出现延迟如何处理

22 和同事的设计思路不一样怎么处理

23 如何保证开发质量

24 职业规划是什么？短期，长期目标是什么

25 团队的规划是什么

26 能介绍下从工作到现在自己的成长在那里

27 项目中用的中间件的理解(Dubbo、MQ、Redis、kafka、zk)

Dubbo(RPC+容错负载+服务注册发现)
MQ（消息队列，同步解耦）
Redis（缓存数据库）
zk（协调管理中心，集群实现）

28 高并发架构的设计思路

服务器架构：单一到集群，再到分布式服务
数据库：主从分离，集群分布式，Nosql数据库
数据解耦：消息队列
负载均衡：nginx
缓存配置：redis,memcache
数据库索引：elasticsearch

29 未来三年职业成长规划

找到立足点，证明自己，提升自己，学习身边的人事，让自己更加优秀
技术层面上能Cover住项目，能再小团队作业中脱颖而出

30 谈谈你对SOA和微服务的理解，以及分布式架构从应用层面涉及到的调整和挑战

面向服务的架构（SOA）
微服务相比于SOA更加精细，微服务更多的以独立的进程的方式存在，互相之间并无影响；
微服务提供的接口方式更加通用化，例如HTTP RESTful方式，各种终端都可以调用，无关语言、平台限制；
微服务更倾向于分布式去中心化的部署方式，在互联网业务场景下更适合；
分布式架构从应用层面问题:数据一致性问题，数据同步问题，分布式事务问题

31 在阿里有了解过什么中间件吗？实现原理？与其他开源消息队列有什么特点？

32 API接口与SDK接口的区别（API是提供给别人的接口）。

SDK相当于开发集成工具环境，API就是数据接口。在SDK环境下调用API数据。

33 流量控制相关问题

漏桶，令牌桶，计数器

34 如何获取一个本地服务器上可用的端口

netstat查看端口使用情况

35 数据库TPS是多少，是否进行测试过

每秒事务数

36 什么是著名的拜占庭将军问题

这个问题的本质就是如何让众多完全平等的节点针对某一状态达成共识。PoW工作量证明

37 xml，json，protocalbuffer序列化方式的比较

XML、JSON、PB比较

最近公司要把原来的项目从php生成xml该成用python生成pb(Protocol Buffer)结构，以前没接触pb，从网上查了一些。

我现在来总结一下：

	XML	JSON	PB
数据结构支持	复杂结构	简单结构	复杂结构
数据保存方式	文本	文本	二进制
数据保存大小	大	一般	小
解析效率	慢	一般	快
语言支持程度	非常多	多	C++/Java/Python/golang
开发难度？繁琐？	繁琐	简单	简单
学习成本	低	低	低
适用范围	数据交换	数据交换	数据交换
读取性	好	一般	差