二、Java多线程

线程池的原理，为什么要创建线程池？创建线程池的方式；

线程的生命周期，什么时候会出现僵死进程；

说说线程安全问题，什么实现线程安全，如何实现线程安全；

创建线程池有哪几个核心参数？ 如何合理配置线程池的大小？

volatile、ThreadLocal的使用场景和原理；

ThreadLocal什么时候会出现OOM的情况？为什么？

synchronized、volatile区别、synchronized锁粒度、模拟死锁场景、原子性与可见性；

三、JVM相关

JVM内存模型，GC机制和原理；

GC分哪两种，Minor GC 和Full GC有什么区别？什么时候会触发Full GC？分别采用什么算法？

JVM里的有几种classloader，为什么会有多种？

什么是双亲委派机制？介绍一些运作过程，双亲委派模型的好处；

什么情况下我们需要破坏双亲委派模型；

常见的JVM调优方法有哪些？可以具体到调整哪个参数，调成什么值？

JVM虚拟机内存划分、类加载器、垃圾收集算法、垃圾收集器、class文件结构是如何解析的；

四、Java扩展篇

红黑树的实现原理和应用场景；

NIO是什么？适用于何种场景？

Java9比Java8改进了什么；

HashMap内部的数据结构是什么？底层是怎么实现的？（还可能会延伸考察ConcurrentHashMap与HashMap、HashTable等，考察对技术细节的深入了解程度）；

说说反射的用途及实现，反射是不是很慢，我们在项目中是否要避免使用反射；

说说自定义注解的场景及实现；

List 和 Map 区别，Arraylist 与 LinkedList 区别，ArrayList 与 Vector 区别；

五、Spring相关

Spring AOP的实现原理和场景？

Spring bean的作用域和生命周期；

Spring Boot比Spring做了哪些改进？ Spring 5比Spring4做了哪些改进；

如何自定义一个Spring Boot Starter？

Spring IOC是什么？优点是什么？

SpringMVC、动态代理、反射、AOP原理、事务隔离级别；

六、中间件篇

Dubbo完整的一次调用链路介绍；

Dubbo支持几种负载均衡策略？

Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做？

Dubbo启动的时候支持几种配置方式？

了解几种消息中间件产品？各产品的优缺点介绍；

消息中间件如何保证消息的一致性和如何进行消息的重试机制？

Spring Cloud熔断机制介绍；

Spring Cloud对比下Dubbo，什么场景下该使用Spring Cloud?

七、数据库篇

锁机制介绍：行锁、表锁、排他锁、共享锁；

乐观锁的业务场景及实现方式；

事务介绍，分布式事物的理解，常见的解决方案有哪些，什么事两阶段提交、三阶段提交；

MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么？

MySQL锁，悲观锁、乐观锁、排它锁、共享锁、表级锁、行级锁；

分布式事务的原理2阶段提交，同步异步阻塞非阻塞；

数据库事务隔离级别，MySQL默认的隔离级别、Spring如何实现事务、JDBC如何实现事务、嵌套事务实现、分布式事务实现；

SQL的整个解析、执行过程原理、SQL行转列；

八、Redis

Redis为什么这么快？redis采用多线程会有哪些问题?

Redis支持哪几种数据结构；

Redis跳跃表的问题；

Redis单进程单线程的Redis如何能够高并发?

Redis如何使用Redis实现分布式锁?

Redis分布式锁操作的原子性，Redis内部是如何实现的?

九、其他

【解答参照】网址：https://shimo.im/docs/LUYuXUGSX8wTOzY7/ 《阿里云 - 面试总结》

一、Java多线程

1、线程池的原理，为什么要创建线程池？创建线程池的方式；

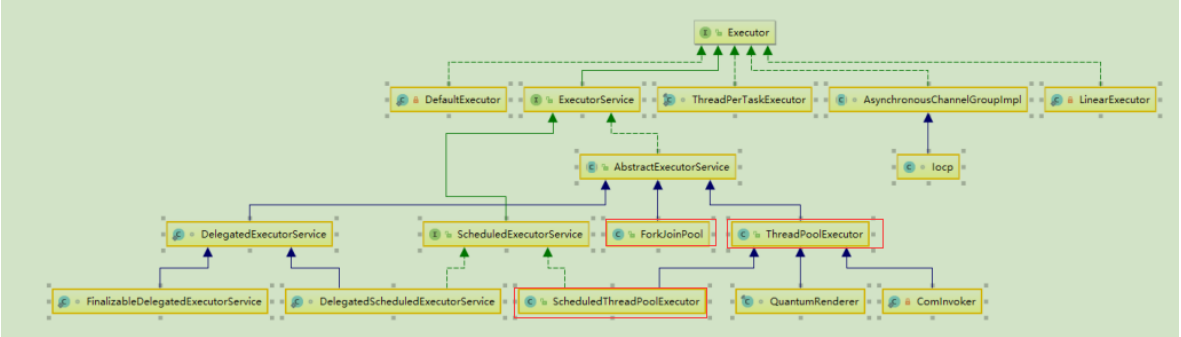
原理：

JAVA线程池原理详解一

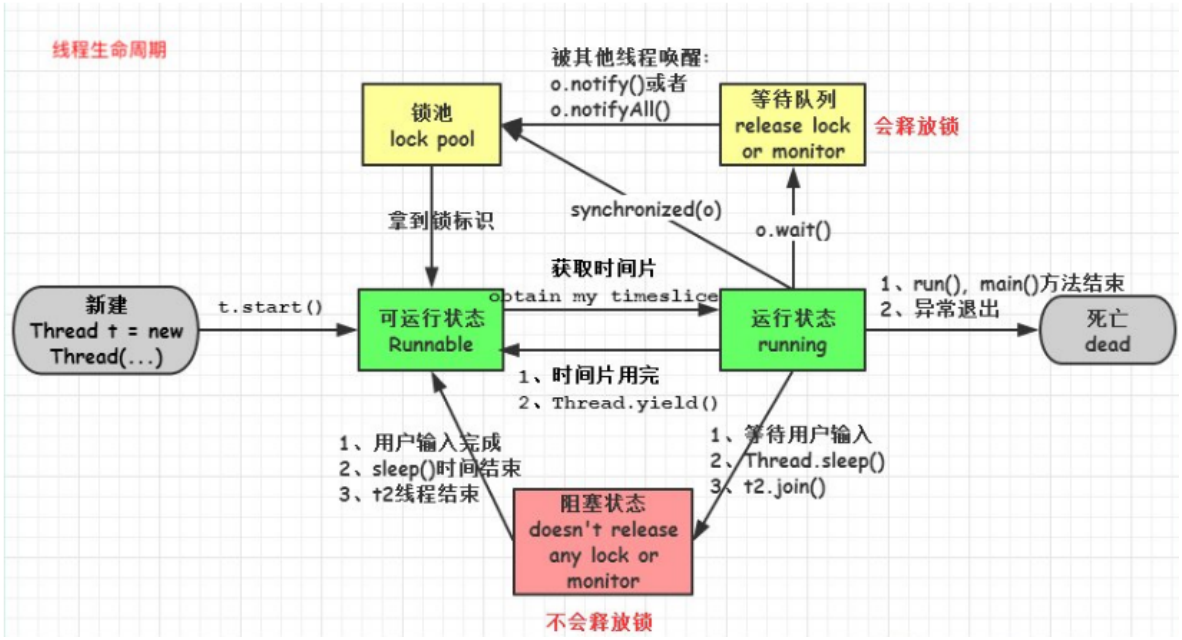
JAVA线程池原理详解二

创建线程池的几种方式：

ThreadPoolExecutor、ThreadScheduledExecutor、ForkJoinPool



2、线程的生命周期，什么时候会出现僵死进程；



僵死进程是指子进程退出时，父进程并未对其发出的SIGCHLD信号进行适当处理，导致子进程停留在僵死状态等待其父进程为其收尸，这个状态下的子进程就是僵死进程。

3、说说线程安全问题，什么是线程安全，如何实现线程安全；

线程安全 - 如果线程执行过程中不会产生共享资源的冲突，则线程安全。

线程不安全 - 如果有多个线程同时在操作主内存中的变量，则线程不安全

实现线程安全的三种方式

1）互斥同步

临界区：**syncronized、ReentrantLock**

信号量 semaphore

互斥量　mutex

2）非阻塞同步

CAS（Compare **And Swap)**

3）无同步方案

可重入代码

使用Threadlocal 类来包装共享变量，做到每个线程有自己的copy

线程本地存储

参考：https://blog.csdn.net/jackieeecheng/article/details/69779824

4、创建线程池有哪几个核心参数？ 如何合理配置线程池的大小？

1）核心参数

```
public ThreadPoolExecutor(int corePoolSize,  // 核心线程数量大小
                          int maximumPoolSize, // 线程池最大容纳线程数
                          long keepAliveTime, // 线程空闲后的存活时长
                          TimeUnit unit,
                        //缓存异步任务的队列 //用来构造线程池里的worker线程
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          //线程池任务满载后采取的任务拒绝策略
                          RejectedExecutionHandler handler)
```

2) 核心说明

1、当线程池中线程数量小于 corePoolSize 则创建线程，并处理请求。

2、当线程池中线程数量大于等于 corePoolSize 时，则把请求放入 workQueue 中,随着线程池中的核心线程们不断执行任务，只要线程池中有空闲的核心线程，线程池就从 workQueue 中取任务并处理。

3 、当 workQueue 已存满，放不下新任务时则新建非核心线程入池，并处理请求直到线程数目达到 maximumPoolSize（最大线程数量设置值）。

4、如果线程池中线程数大于 maximumPoolSize 则使用 RejectedExecutionHandler 来进行任务拒绝处理。

参考：http://gudong.name/2017/05/03/thread-pool-intro.html

3)线程池大小分配

线程池究竟设置多大要看你的线程池执行的什么任务了，CPU密集型、IO密集型、混合型，任务类型不同，设置的方式也不一样。

任务一般分为：CPU密集型、IO密集型、混合型，对于不同类型的任务需要分配不同大小的线程池。

3.1）CPU密集型

尽量使用较小的线程池，一般Cpu核心数+1

3.2）IO密集型

方法一：可以使用较大的线程池，一般CPU核心数 * 2

方法二：（线程等待时间与线程CPU时间之比 + 1）* CPU数目

3.3）混合型

可以将任务分为CPU密集型和IO密集型，然后分别使用不同的线程池去处理，按情况而定

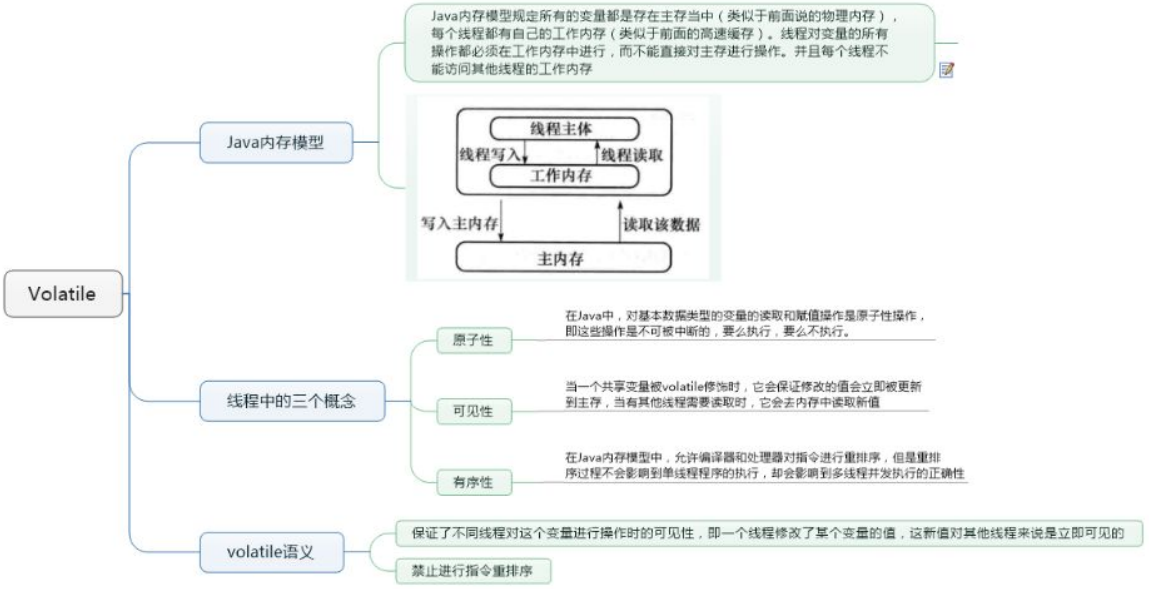
参考：https://www.cnblogs.com/cherish010/p/8334952.html

5、volatile、ThreadLocal的使用场景和原理；

volatile原理

volatile变量进行写操作时，JVM 会向处理器发送一条 Lock 前缀的指令，将这个变量所在缓存行的数据写会到系统内存。

Lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成。



volatile的适用场景

1）状态标志,如：初始化或请求停机

2）一次性安全发布，如：单列模式

3）独立观察，如：定期更新某个值

4） “volatile bean” 模式

5) 开销较低的“读 - 写锁”策略，如：计数器

参考：https://blog.csdn.net/hgc0907/article/details/79664102

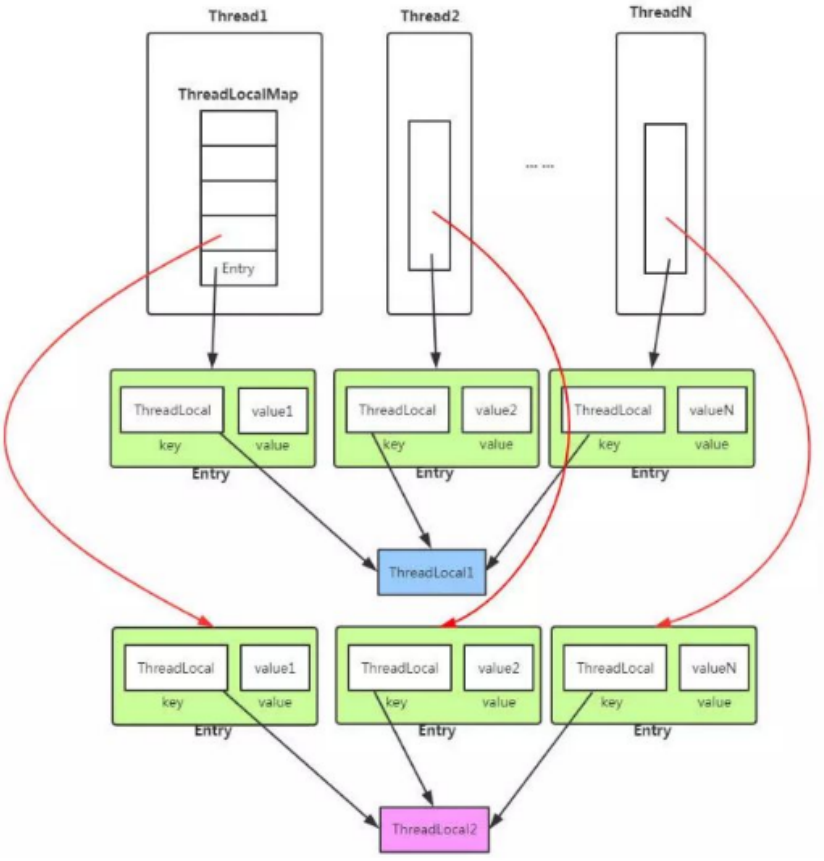
参考：https://www.ibm.com/developerworks/cn/java/j-jtp06197.html

ThreadLocal原理

ThreadLocal是用来维护本线程的变量的，并不能解决共享变量的并发问题。ThreadLocal是各线程将值存入该线程的map中，以ThreadLocal自身作为key，需要用时获得的是该线程之前存入的值。如果存入的是共享变量，那取出的也是共享变量，并发问题还是存在的。

参考：https://www.jianshu.com/p/ee8c9dccc953

参考：https://blog.csdn.net/xlgen157387/article/details/78297568



ThreadLocal结构内部

ThreadLocal的适用场景

场景：数据库连接、Session管理、

参考：https://www.jianshu.com/p/cadd53f063b9

6、ThreadLocal什么时候会出现OOM的情况？为什么？

ThreadLocal变量是维护在Thread内部的，这样的话只要我们的线程不退出，对象的引用就会一直存在。当线程退出时，Thread类会进行一些清理工作，其中就包含ThreadLocalMap，Thread调用**exit**方法如下：

```
/**
 * This method is called by the system to give a Thread
 * a chance to clean up before it actually exits.
 */
private void exit() {
    if (group != null) {
        group.threadTerminated(this);
        group = null;
    }
    /* Aggressively null out all reference fields: see bug 4006245 */
    target = null;
    /* Speed the release of some of these resources */
    threadLocals = null;
    inheritableThreadLocals = null;
    inheritedAccessControlContext = null;
    blocker = null;
    uncaughtExceptionHandler = null;
}
```

ThreadLocal在没有线程池使用的情况下，正常情况下不会存在内存泄露，但是如果使用了线程池的话，就依赖于线程池的实现，如果线程池不销毁线程的话，那么就会存在内存泄露。

参考：https://blog.csdn.net/xlgen157387/article/details/78297568

7、synchronized、volatile区别

1) **volatile**主要应用在多个线程对实例变量更改的场合，刷新主内存共享变量的值从而使得各个线程可以获得最新的值，线程读取变量的值需要从主存中读取；**synchronized**则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。另外，**synchronized**还会创建一个内存屏障，内存屏障指令保证了所有CPU操作结果都会直接刷到主存中（即释放锁前），从而保证了操作的内存可见性，同时也使得先获得这个锁的线程的所有操作

2) **volatile**仅能使用在变量级别；**synchronized**则可以使用在变量、方法、和类级别的。**volatile**不会造成线程的阻塞；**synchronized**可能会造成线程的阻塞，比如多个线程争抢**synchronized**锁对象时，会出现阻塞。

3) **volatile**仅能实现变量的修改可见性，不能保证原子性；而**synchronized**则可以保证变量的修改可见性和原子性，因为线程获得锁才能进入临界区，从而保证临界区中的所有语句全部得到执行。

4) **volatile**标记的变量不会被编译器优化，可以禁止进行指令重排；**synchronized**标记的变量可以被编译器优化。

参考：https://blog.csdn.net/xiaoming100001/article/details/79781680

8、synchronized锁粒度、模拟死锁场景；

synchronized：具有原子性，有序性和可见性

参考：https://www.jianshu.com/p/cf57726e77f2

粒度：对象锁、类锁

死锁场景，参考：https://blog.csdn.net/u013925989/article/details/50208839

二、JVM相关

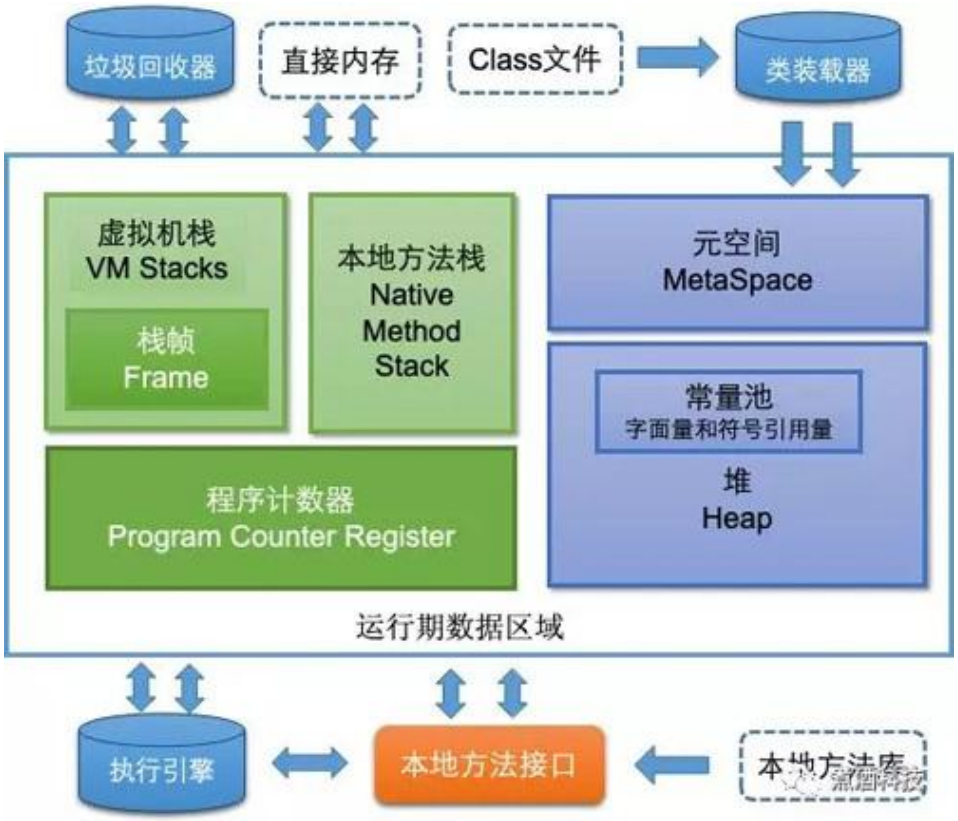
1、JVM内存模型，GC机制和原理；

内存模型

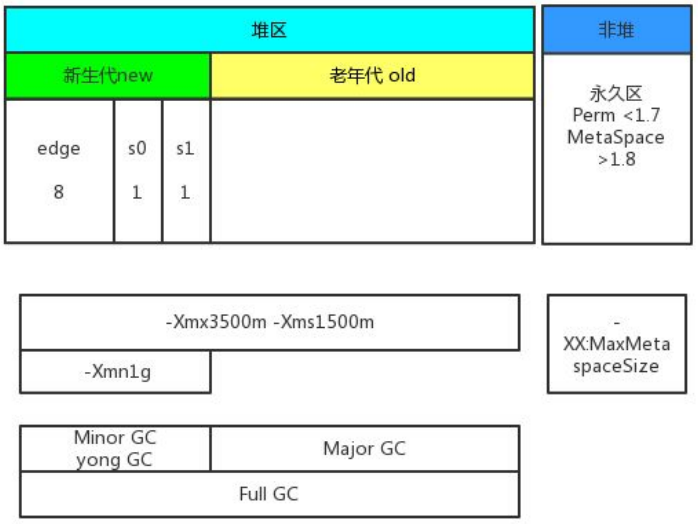
Jdk1.6及之前：有永久代, 常量池在方法区

Jdk1.7：有永久代，但已经逐步“去永久代”，常量池在堆

Jdk1.8及之后： 无永久代，常量池在元空间



2、GC分哪两种，Minor GC 和Full GC有什么区别？什么时候会触发Full GC？分别采用什么算法？



对象从新生代区域消失的过程，我们称之为 "minor GC"
对象从老年代区域消失的过程，我们称之为 "major GC"

Minor GC
清理整个YouGen的过程，eden的清理，S0\S1的清理都会由于MinorGC Allocation Failure(YoungGen区内存不足），而触发minorGC

Major GC
OldGen区内存不足，触发Major GC

Full GC
Full GC 是清理整个堆空间—包括年轻代和永久代

Full GC 触发的场景
1）System.gc
2）promotion failed (年代晋升失败,比如eden区的存活对象晋升到S区放不下，又尝试直接晋升到Old区又放不下，那么Promotion Failed,会触发FullGC)
3）CMS的Concurrent-Mode-Failure
由于CMS回收过程中主要分为四步: 1.CMS initial mark 2.CMS Concurrent mark 3.CMS remark 4.CMS Concurrent sweep。在2中gc线程与用户线程同时执行，那么用户线程依旧可能同时产生垃圾， 如果这个垃圾较多无法放入预留的空间就会产生CMS-Mode-Failure， 切换为SerialOld单线程做mark-sweep-compact。
4）新生代晋升的平均大小大于老年代的剩余空间 （为了避免新生代晋升到老年代失败）
当使用G1,CMS 时，FullGC发生的时候 是 Serial+SerialOld。
当使用ParalOld时，FullGC发生的时候是 ParallNew +ParallOld.

3、JVM里的有几种classloader，为什么会有多种?

启动类加载器：负责加载**JRE**的核心类库，如**jre**目标下的**rt**.jar,**charsets**.jar等

扩展类加载器：负责加载**JRE**扩展目录**ext**中**JAR**类包

系统类加载器：负责加载**ClassPath**路径下的类包

用户自定义加载器：负责加载用户自定义路径下的类包

为什么会有多种：

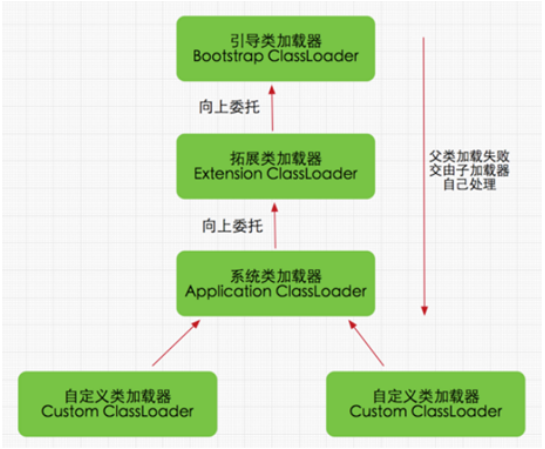
1）分工，各自负责各自的区块

2）为了实现委托模型

4、什么是双亲委派机制？介绍一些运作过程，双亲委派模型的好处；

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都不愿意干活，每次有活就丢给父亲去干，直到父亲说这件事我也干不了时，儿子自己想办法去完成，这不就是传说中的双亲委派模式。

动作过程



好处

沙箱安全机制：自己写的String.**class**类不会被加载，这样便可以防止核心API库被随意篡改

避免类的重复加载：当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次

5、什么情况下我们需要破坏双亲委派模型；

<待补充>

6、常见的JVM调优方法有哪些？可以具体到调整哪个参数，调成什么值?

调优工具

console，jProfile，VisualVM

Dump线程详细信息：查看线程内部运行情况
死锁检查

查看堆内类、对象信息查看：数量、类型等

线程监控
线程信息监控：系统线程数量。
线程状态监控：各个线程都处在什么样的状态下

热点分析
CPU热点：检查系统哪些方法占用的大量CPU时间
内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

内存泄漏检查

<待补充>

7、JVM虚拟机内存划分、类加载器、垃圾收集算法、垃圾收集器、class文件结构是如何解析的；

JVM虚拟机内存划分（重复）
类加载器（重复）

垃圾收集算法：标记-清除算法、复制算法、标记-整理算法、分代收集算法

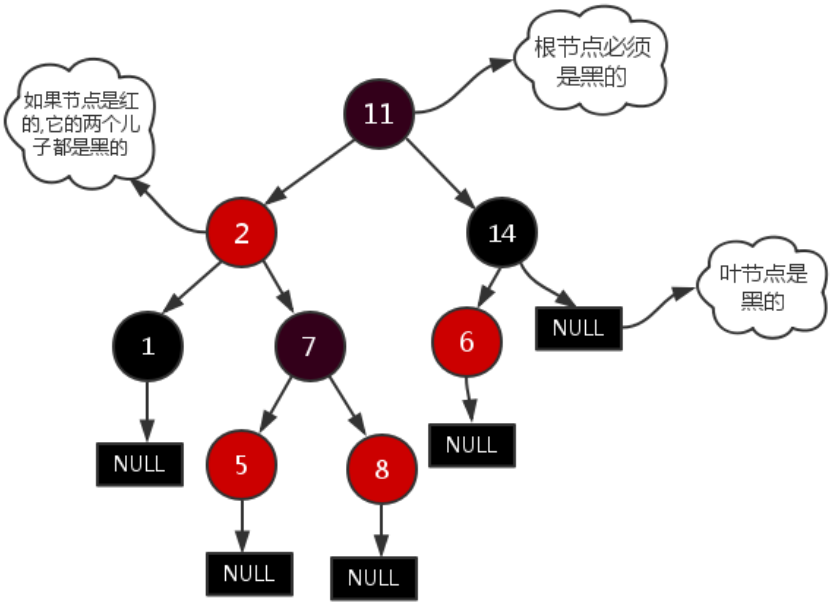
垃圾收集器： Serial收集器、ParNew收集器、**Parallel** Scavenge收集器、Serial **Old**收集器、**Parallel Old**收集器、CMS收集器、G1收集器、Z垃圾收集器

class文件结构是如何解析的
解悉过程：https://blog.csdn.net/sinat_38259539/article/details/78248454

```
ClassFile {
    u4 magic;                       魔数
    u2 minor_version;               副版本号
    u2 major_version;               主版本号
    u2 constant_pool_count;         常量池计数器
    cp_info constant_pool[constant_pool_count-1];   常量池
    u2 access_flags;                访问标志
    u2 this_class;                  类索引
    u2 super_class;                 父类索引
    u2 interfaces_count;            接口计数器
    u2 interfaces[interfaces_count];    接口表
    u2 fields_count;                字段计数器
    field_info fields[fields_count];    字段表
    u2 methods_count;               方法计数器
    method_info methods[methods_count];     方法表
    u2 attributes_count;            属性计数器
    attribute_info attributes[attributes_count];    属性表
}
```


三、Java扩展篇

1、红黑树的实现原理和应用场景；



红黑树(一棵自平衡的排序二叉树)五大特性：

1）每个结点要么是红的，要么是黑的。

2）根结点是黑的。

3）每个叶结点，即空结点是黑的。

4）如果一个结点是红的，那么它的俩个儿子都是黑的。

5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。

场景

1）广泛用于C++的STL中,map和set都是用红黑树实现的.

2）著名的linux进程调度Completely Fair Scheduler,用红黑树管理进程控制块,进程的虚拟内存区域都存储在一颗红黑树上,每个虚拟地址区域都对应红黑树的一个节点,左指针指向相邻的地址虚拟存储区域,右指针指向相邻的高地址虚拟地址空间.

3）IO多路复用epoll的实现采用红黑树组织管理sockfd，以支持快速的增删改查.

4）ngnix中,用红黑树管理**timer**,因为红黑树是有序的,可以很快的得到距离当前最小的定时器.

5）java中的TreeSet,TreeMap

2、NIO是什么？适用于何种场景?

（New IO）为所有的原始类型（boolean类型除外）提供缓存支持的数据容器，使用它可以提供非阻塞式的高伸缩性网络。

特性：**I/O**多路复用 + 非阻塞式**I/O**

NIO适用场景

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。例如总公司的一个中心服务器需要收集全国便利店各个收银机的交易信息，只需要少量线程按需处理维护的大量长期连接。

Jetty、Mina、Netty、ZooKeeper等都是基于NIO方式实现。

【NIO技术概览】

3、Java9比Java8改进了什么；

1）引入了模块系统，采用模块化系统的应用程序只需要这些应用程序所需的那部分JDK模块，而非是整个JDK框架了，减少了内存的开销。

2）引入了一个新的package:java.net.http，里面提供了对Http访问很好的支持，不仅支持Http1.1而且还支持HTTP2。

3）引入了jshell这个交互性工具，让Java也可以像脚本语言一样来运行，可以从控制台启动 jshell ，在 jshell 中直接输入表达式并查看其执行结果。

4）增加了**List**.of()、**Set**.of()、**Map**.of()和**Map**.ofEntries()等工厂方法来创建不可变集合

5）HTML5风格的Java帮助文档

6）多版本兼容 JAR 功能，能让你创建仅在特定版本的 Java 环境中运行库程序时选择使用的 class 版本。

7）统一 JVM 日志

可以使用新的命令行选项-Xlog 来控制JVM 上 所有组件的日志记录。该日志记录系统可以设置输出的日志消息的标签、级别、修饰符和输出目标等。

8）垃圾收集机制

Java 9 移除了在 Java 8 中 被废弃的垃圾回收器配置组合，同时把G1设为默认的垃圾回收器实现.因为相对于Parallel来说，G1会在应用线程上做更多的事情，而Parallel几乎没有在应用线程上做任何事情，它基本上完全依赖GC线程完成所有的内存管理。这意味着切换到G1将会为应用线程带来额外的工作，从而直接影响到应用的性能

9）I/O 流新特性

java.io.**InputStream** 中增加了新的方法来读取和复制 **InputStream** 中包含的数据。

readAllBytes：读取 **InputStream** 中的所有剩余字节。

readNBytes： 从 **InputStream** 中读取指定数量的字节到数组中。

transferTo：读取 **InputStream** 中的全部字节并写入到指定的 **OutputStream** 中 。

参考：

java8新特性

java9 新特性

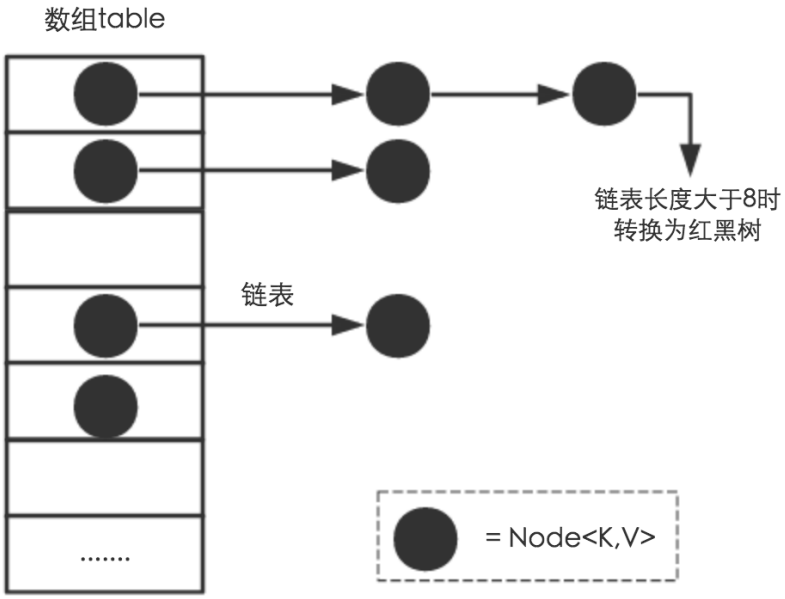
4、HashMap内部的数据结构是什么？底层是怎么实现的？

HashMap内部结构

jdk8以前：数组+链表

jdk8以后：数组+链表 （当链表长度到8时，转化为红黑树）

在并发的情况，发生扩容时，可能会产生循环链表，在执行get的时候，会触发死循环，引起CPU的100%问题，所以一定要避免在并发环境下使用**HashMap**。



5、延伸考察ConcurrentHashMap与HashMap、HashTable等，考察对技术细节的深入了解程度；

HashMap、HashTable、ConcurrentHashMap的原理与区别

老生常谈，HashMap的死循环

ConcurrentHashMap在jdk1.8中的改进

谈谈ConcurrentHashMap1.7和1.8的不同实现

深入分析ConcurrentHashMap1.8的扩容实现

深入浅出ConcurrentHashMap1.8

ConcurrentHashMap的红黑树实现分析

6、说说反射的用途及实现，反射是不是很慢，我们在项目中是否要避免使用反射；

一、用途

反射被广泛地用于那些需要在运行时检测或修改程序行为的程序中。

二、实现方式

```
Foo foo = new Foo();
```

第一种：通过Object类的getClass方法

```
Class cla = foo.getClass();
```

第二种：通过对象实例方法获取对象

```
Class cla = foo.class;
```

第三种：通过Class.forName方式

```
Class cla = Class.forName("xx.xx.Foo");
```

三、缺点

1）影响性能

反射包括了一些动态类型，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。

2）安全限制

使用反射技术要求程序必须在一个没有安全限制的环境中运行。

3）内部暴露

由于反射允许代码执行一些在正常情况下不被允许的操作（比如访问私有的属性和方 法），所以使用反射可能会导致意料之外的副作用 - - 代码有功能上的错误，降低可移植性。 反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

7、说说自定义注解的场景及实现；

利用自定义注解,结合SpringAOP可以完成权限控制、日志记录、统一异常处理、数字签名、数据加解密等功能。

实现场景（API接口数据加解密）

1）自定义一个注解，在需要加解密的方法上添加该注解

2）配置SringAOP环绕通知

3）截获方法入参并进行解密

4）截获方法返回值并进行加密

8、List 和 Map 区别

一、概述

List是存储单列数据的集合，Map是存储键和值这样的双列数据的集合，

List中存储的数据是有顺序，并且允许重复，值允许有多个**null**；

Map中存储的数据是没有顺序的，键不能重复，值是可以有重复的，key最多有一个**null**。

二、明细

List

1）可以允许重复的对象。

2）可以插入多个**null**元素。

3）是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4）常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Map

1）Map不是collection的子接口或者实现类。Map是一个接口。

2）Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

3）TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。

4）Map 里你可以拥有随意个 **null** 值但最多只能有一个 **null** 键。

5）Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）

Set（问题扩展）

1）不允许重复对象

2）无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或Comparable 维护了一个排序顺序。

3）只允许一个 **null** 元素

4）Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

三、场景（问题扩展）

1）如果你经常会使用索引来对容器中的元素进行访问，那么 List 是你的正确的选择。如果你已经知道索引了的话，那么 List 的实现类比如 ArrayList 可以提供更快速的访问,如果经常添加删除元素的，那么肯定要选择LinkedList。

2）如果你想容器中的元素能够按照它们插入的次序进行有序存储，那么还是 List，因为 List 是一个有序容器，它按照插入顺序进行存储。

3）如果你想保证插入元素的唯一性，也就是你不想有重复值的出现，那么可以选择一个 Set 的实现类，比如 HashSet、LinkedHashSet 或者 TreeSet。所有 Set 的实现类都遵循了统一约束比如唯一性，而且还提供了额外的特性比如 TreeSet 还是一个 SortedSet，所有存储于 TreeSet 中的元素可以使用 Java 里的 Comparator 或者 Comparable 进行排序。LinkedHashSet 也按照元素的插入顺序对它们进行存储。

4）如果你以键和值的形式进行数据存储那么 Map 是你正确的选择。你可以根据你的后续需要从 Hashtable、HashMap、TreeMap 中进行选择。

参考：List、Set、Map的区别

9、Arraylist 与 LinkedList 区别，ArrayList 与 Vector 区别;

1）数据结构

Vector、ArrayList内部使用数组，而LinkedList内部使用双向链表，由数组和链表的特性知：

LinkedList适合指定位置插入、删除操作，不适合查找；

ArrayList、Vector适合查找，不适合指定位置的插入删除操作。

但是ArrayList越靠近尾部的元素进行增删时，其实效率比LinkedList要高

2)线程安全

Vector线程安全，ArrayList、LinkedList线程不安全。

3）空间

ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

参考：ArrayList和LinkedList内部实现、区别、使用场景

四、Spring相关

1、Spring AOP的实现原理和场景;

AOP（Aspect Orient Programming），作为面向对象编程的一种补充，广泛应用于处理一些具有横切性质的系统级服务。

一、场景

事务管理、安全检查、权限控制、数据校验、缓存、对象池管理等

二、实现技术

AOP（这里的AOP指的是面向切面编程思想，而不是Spring AOP）主要的的实现技术主要有Spring AOP和AspectJ。

1）AspectJ的底层技术。

AspectJ的底层技术是静态代理，即用一种AspectJ支持的特定语言编写切面，通过一个命令来编译，生成一个新的代理类，该代理类增强了业务类，这是在编译时增强，相对于下面说的运行时增强，编译时增强的性能更好。

2）Spring AOP

Spring AOP采用的是动态代理，在运行期间对业务方法进行增强，所以不会生成新类，对于动态代理技术，Spring AOP提供了对JDK动态代理的支持以及CGLib的支持。

JDK动态代理只能为接口创建动态代理实例，而不能对类创建动态代理。需要获得被目标类的接口信息（应用Java的反射技术），生成一个实现了代理接口的动态代理类（字节码），再通过反射机制获得动态代理类的构造函数，利用构造函数生成动态代理类的实例对象，在调用具体方法前调用invokeHandler方法来处理。

CGLib动态代理需要依赖**asm**包，把被代理对象类的**class**文件加载进来，修改其字节码生成子类。

但是Spring AOP基于注解配置的情况下，需要依赖于AspectJ包的标准注解。

2、Spring bean的作用域和生命周期；

作用域

类别	说明
singleton	在Spring IoC容器中仅存在一个Bean实例，Bean以单例方式存在，默认值
prototype	每次从容器中调用Bean时，都返回一个新的实例，即每次调用getBean()时，相当于执行new XxxBean()
request	每次HTTP请求都会创建一个新的Bean，该作用域仅适用于WebApplicationContext环境
session	同一个HTTP Session 共享一个Bean，不同Session使用不同Bean，仅适用于WebApplicationContext 环境
globalSession	一般用于Portlet应用环境，该作用域仅适用于WebApplicationContext 环境

生命周期

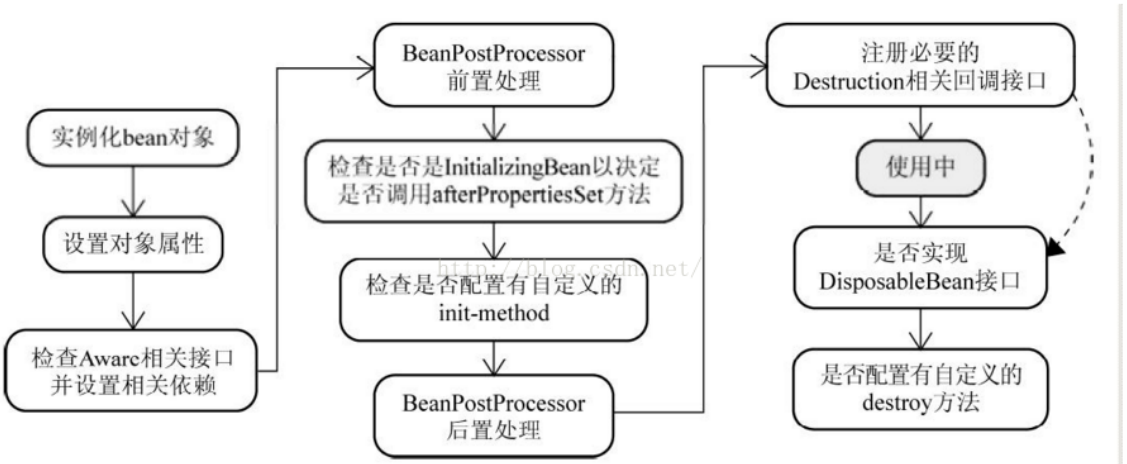


图4-10　Bean的实例化过程

3、Spring Boot比Spring做了哪些改进?

1）**Spring Boot**可以建立独立的**Spring**应用程序；

2）内嵌了如**Tomcat**，**Jetty**和**Undertow**这样的容器，也就是说可以直接跑起来，用不着再做部署工作了；

3）无需再像**Spring**那样搞一堆繁琐的**xml**文件的配置；

4）可以自动配置**Spring**。**SpringBoot**将原有的**XML**配置改为**Java**配置，将**bean**注入改为使用注解注入的方式(@Autowire)，并将多个**xml**、**properties**配置浓缩在一个**appliaction**.yml配置文件中。

5）提供了一些现有的功能，如量度工具，表单数据验证以及一些外部配置这样的一些第三方功能；

6）整合常用依赖（开发库，例如**spring-webmvc**、**jackson-json**、**validation-api**和**tomcat**等），提供的**POM**可以简化**Maven**的配置。当我们引入核心依赖时，**SpringBoot**会自引入其他依赖。

Spring 5比Spring4做了哪些改进;

【官网说明 】

Spring 4.x新特性

1. 泛型限定式依赖注入
2. 核心容器的改进
3. web开发增强
4. 集成Bean Validation 1.1（JSR-349）到SpringMVC
5. Groovy Bean定义DSL
6. 更好的Java泛型操作API
7. JSR310日期API的支持
8. 注解、脚本、任务、MVC等其他特性改进

Spring 5.x新特性

1. JDK8的增强
2. 核心容器的改进

3. 新的SpringWebFlux模块

4. 测试方面的改进

参考：

《Spring5官方文档》新功能

Spring4新特性——泛型限定式依赖注入

Spring4新特性——核心容器的其他改进

Spring4新特性——Web开发的增强

Spring4新特性——集成Bean Validation 1.1(JSR-349)到SpringMVC

Spring4新特性——Groovy Bean定义DSL

Spring4新特性——更好的Java泛型操作API

Spring4新特性——JSR310日期API的支持

Spring4新特性——注解、脚本、任务、MVC等其他特性改进

如何自定义一个Spring Boot Starter?

《自定义spring boot starter三部曲之一：准备工作》；

《自定义spring boot starter三部曲之二：实战开发》；

《自定义spring boot starter三部曲之三：源码分析spring.factories加载过程》；

Spring IOC是什么？优点是什么?

IoC文英全称Inversion of Control，即控制反转，我么可以这么理解IoC容器：“把某些业务对象的的控制权交给一个平台或者框架来同一管理，这个同一管理的平台可以称为IoC容器。”

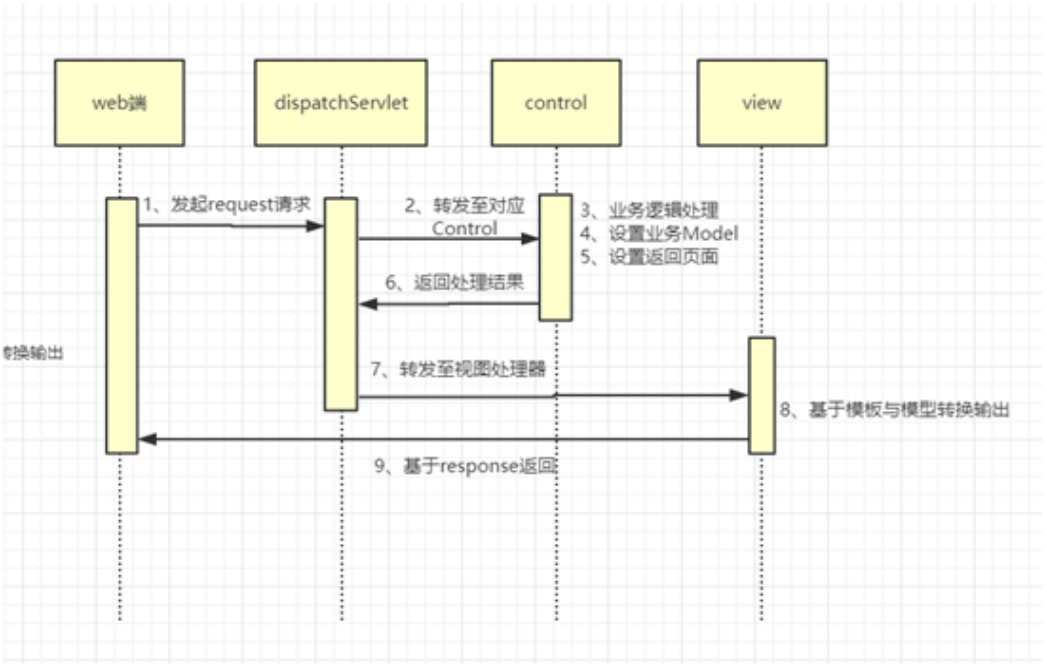
ioc的思想最核心的地方在于，资源不由使用资源的双方管理，而由不使用资源的第三方管理，这可以带来很多好处：

1）资源集中管理，实现资源的可配置和易管理

2）降低了使用资源双方的依赖程度，也就是我们说的耦合度

Spring IOC原理解读 面试必读

SpringMVC



动态代理
反射
AOP原理
Spring事务;

√: 可能出现　×: 不会出现

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

一、spring事务
什么是事务: 事务逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

二、事务特性（4种）:
原子性 （atomicity）:强调事务的不可分割.
一致性 （consistency）:事务的执行的前后数据的完整性保持一致.
隔离性 （isolation）:一个事务执行的过程中,不应该受到其他事务的干扰
持久性（durability） :事务一旦结束,数据就持久到数据库

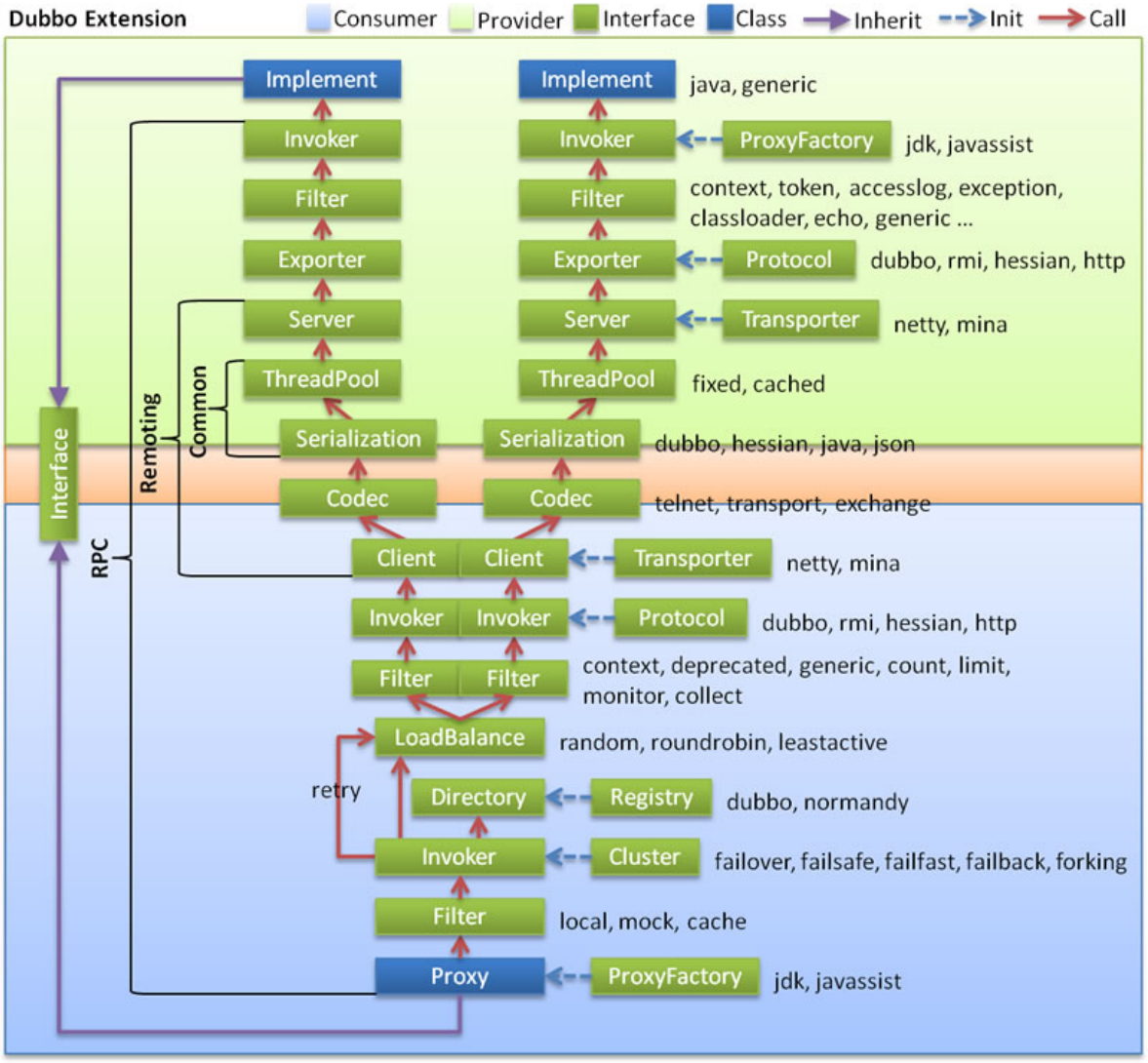
如果不考虑隔离性引发安全性问题:
脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致.
虚幻读 :一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

三、解决读问题: 设置事务隔离级别（5种）
DEFAULT 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.
未提交读（read uncommited） :脏读，不可重复读，虚读都有可能发生
已提交读 （read commited）:避免脏读。但是不可重复读和虚读有可能发生
可重复读 （repeatable read） :避免脏读和不可重复读.但是虚读有可能发生.
串行化的 （serializable） :避免以上所有读问题.
Mysql 默认:可重复读
Oracle 默认:读已提交

四、事务的传播行为
PROPAGION_XXX :事务的传播行为
* 保证同一个事务中
PROPAGATION_REQUIRED 支持当前事务，如果不存在 就新建一个(默认)
PROPAGATION_SUPPORTS 支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY 支持当前事务，如果不存在，抛出异常
* 保证没有在同一个事务中
PROPAGATION_REQUIRES_NEW 如果有事务存在，挂起当前事务，创建一个新的事务
PROPAGATION_NOT_SUPPORTED 以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER 以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED 如果当前事务存在，则嵌套事务执行

五、中间件篇
Dubbo完整的一次调用链路介绍;



参考：http://dubbo.apache.org/zh-cn/docs/dev/design.html

Dubbo支持几种负载均衡策略?

1）Random LoadBalance

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。

2）RoundRobin LoadBalance

轮询，按公约后的权重设置轮询比率。

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。

3）LeastActive LoadBalance

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。

4）ConsistentHash LoadBalance

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

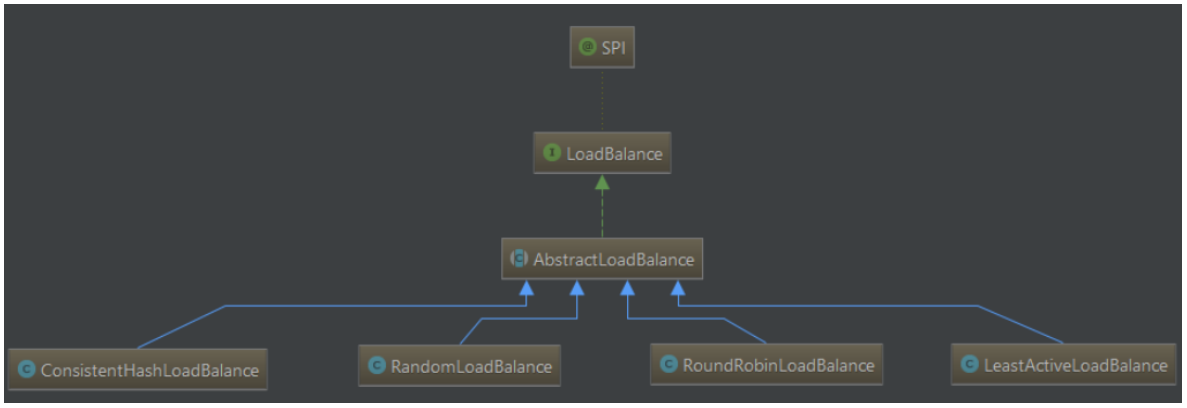
缺省只对第一个参数 Hash，如果要修改，请配置

```
<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" />
```

参考：http://dubbo.apache.org/zh-cn/docs/user/demos/loadbalance.html



原码分析：http://www.cnblogs.com/wyq178/p/9822731.html

Dubbo Provider服务提供者要控制执行并发请求上限，具体怎么做?

服务端并发限流：executes

客户端并发限流：actives

样例 1

限制 com.foo.BarService 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 com.foo.BarService 的 sayHello 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

样例 3

限制 com.foo.BarService 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```
或
```
<dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4
限制 com.foo.BarService 的 sayHello 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：
```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```
或
```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```
参考：http://dubbo.apache.org/zh-cn/docs/user/demos/concurrency-control.html

Dubbo启动的时候支持几种配置方式?

XML配置
http://dubbo.apache.org/zh-cn/docs/user/configuration/xml.html

属性配置
http://dubbo.apache.org/zh-cn/docs/user/configuration/properties.html

API配置
http://dubbo.apache.org/zh-cn/docs/user/configuration/api.html

注解配置
http://dubbo.apache.org/zh-cn/docs/user/configuration/annotation.html

了解几种消息中间件产品？各产品的优缺点介绍；

各种消息队列对比.pdf1MB

消息中间件如何保证消息的一致性

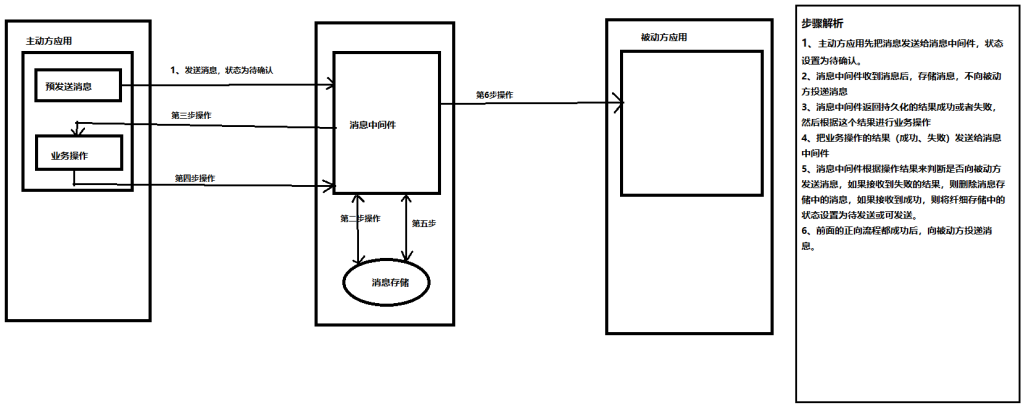
(1)主动方应用先把消息发送给消息中间件，消息状态标记为待确认；
(2)消息中间件收到消息之后，把消息持久化到消息存储中，但并不向被动方应用投递消息；
(3)消息中间件返回消息持久化结果（成功，或者失效），主动方应用根据返回结果进行判断如何处理业务操作处理；
①失败：放弃业务操作处理，结束（必须向上层返回失败结果）
②成功：执行业务操作处理
(4)业务操作完成后，把业务操作结果（成功/失败）发送给消息中间件；

(5)消息中间件收到业务操作结果后，根据结果进行处理；

①失败：删除消息存储中的消息，结束；

②成功：更新消息存储中的消息状态为·待发送（可发送）·，紧接着执行消息投递；

(6)前面的正向流程都成功后，向被动方应用投递消息；



如何进行消息的重试机制?

参考：Rocket重试机制，消息模式，刷盘方式

Spring Cloud熔断机制介绍;

在Spring Cloud框架里，熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败，就会启动熔断机制。熔断机制的注解是@HystrixCommand，Hystrix会找有这个注解的方法，并将这类方法关联到和熔断器连在一起的代理上。当前，@HystrixCommand仅当类的注解为@Service或@Component时才会发挥作用。

参考：http://www.cnblogs.com/lvgg/p/7843809.html

Spring Cloud对比下Dubbo，什么场景下该使用Spring Cloud?

两者所解决的问题域不一样：Dubbo的定位始终是一款RPC框架，而Spring **Cloud的目的是微服务架构下的一站式解决方案。**

Spring **Cloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。**

严格来说，这两种方式各有优劣。虽然在一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适。

核心要素	Dubbo	Spring Cloud
服务注册中心	Zookeeper、Redis	Spring Cloud Netflix Eureka
服务调用方式	RPC	REST API
服务网关	无	Spring Cloud Netflix Zuul
断路器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
分布式追踪系统	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream 基于Redis,Rabbit,Kafka实现的消息微服务
批量任务	无	Spring Cloud Task @51CTO博客

更多对比：http://blog.51cto.com/13954634/2296010

六、数据库篇

锁机制介绍：行锁、表锁、排他锁、共享锁；
乐观锁的业务场景及实现方式；
事务介绍，分布式事物的理解，常见的解决方案有哪些，什么是两阶段提交、三阶段提交；
MySQL记录binlog的方式主要包括三种模式？每种模式的优缺点是什么？

mysql复制主要有三种方式：基于SQL语句的复制(statement-based replication, SBR)，基于行的复制(row-based replication, RBR)，混合模式复制(mixed-based replication, MBR)。对应的，binlog的格式也有三种：STATEMENT，ROW，MIXED。

① STATEMENT模式（SBR）

每一条会修改数据的sql语句会记录到binlog中。优点是并不需要记录每一条sql语句和每一行的数据变化，减少了binlog日志量，节约IO，提高性能。缺点是在某些情况下会导致master-slave中的数据不一致（如sleep()函数， last_insert_id()，以及user-defined functions(udf)等会出现问题)

② ROW模式（RBR）

不记录每条sql语句的上下文信息，仅需记录哪条数据被修改了，修改成什么样了。而且不会出现某些特定情况下的存储过程、或function、或trigger的调用和触发无法被正确复制的问题。缺点是会产生大量的日志，尤其是**alter table**的时候会让日志暴涨。

③ MIXED模式（MBR）

以上两种模式的混合使用，一般的复制使用**STATEMENT**模式保存**binlog**，对于**STATEMENT**模式无法复制的操作使用**ROW**模式保存**binlog**，MySQL会根据执行的**SQL**语句选择日志保存方式。

MySQL锁，悲观锁、乐观锁、排它锁、共享锁、表级锁、行级锁；

乐观锁

用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “**version**” 字段来实现。当读取数据时，将**version**字段的值一同读出，数据每更新一次，对此**version**值加1。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的**version**值进行比对，如果数据库表当前版本号与第一次取出来的**version**值相等，则予以更新，否则认为是过期数据。

悲观锁

在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中**synchronized**很相似，共享锁（读锁）和排它锁（写锁）是悲观锁的不同的实现

共享锁（读锁）

共享锁又叫做读锁，所有的事务只能对其进行读操作不能写操作，加上共享锁后在事务结束之前其他事务只能再加共享锁，除此之外其他任何类型的锁都不能再加了。

排它锁（写锁）

若某个事物对某一行加上了排他锁，只能这个事务对其进行读写，在此事务结束之前，其他事务不能对其进行加任何锁，其他进程可以读取,不能进行写操作，需等待其释放。

表级锁

innodb 的行锁是在有索引的情况下,没有索引的表是锁定全表的

行级锁

行锁又分共享锁和排他锁,由字面意思理解，就是给某一行加上锁，也就是一条记录加上锁。

注意：行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。

更多参考：https://blog.csdn.net/yzj5208/article/details/81288633

分布式事务的原理2阶段提交，同步异步阻塞非阻塞；

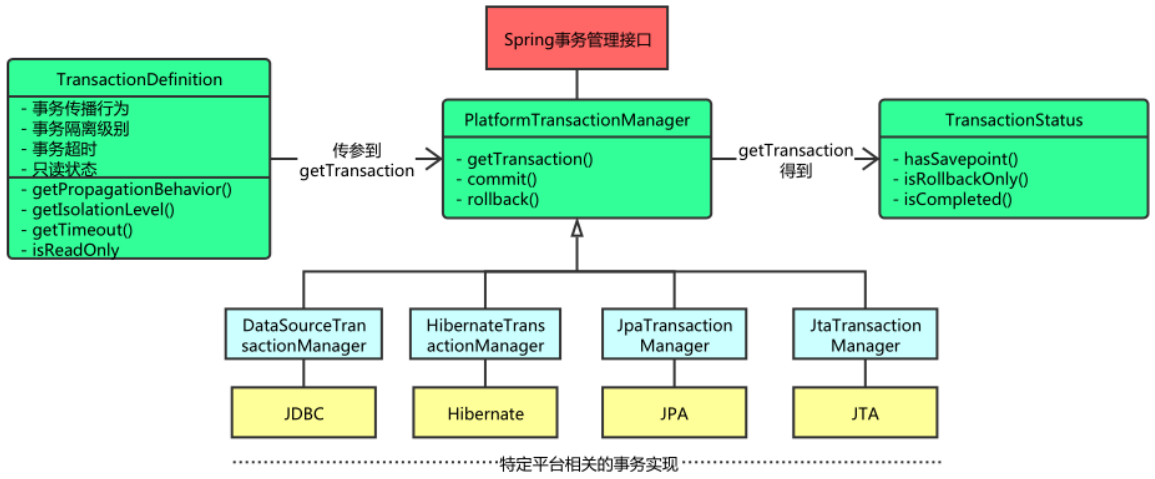
数据库事务隔离级别，MySQL默认的隔离级别

Mysql默认隔离级别：Repeatable **Read**

隔离级别	脏读	不可重复读	幻读
读未提交（Read uncommitted）	V	V	V
读已提交（Read committed）	X	V	V
可重复读（Repeatable read）	X	X	V
可串行化（Serializable）	X	X	X

Spring如何实现事务

参考：spring事务管理(详解和实例)



Spring 事物四种实现方式：

基于编程式事务管理实现

基于TransactionProxyFactoryBean的声明式事务管理

基于AspectJ的XML声明式事务管理

基于注解的声明式事务管理

参考：https://blog.csdn.net/zhuxinquan61/article/details/71075051

JDBC如何实现事务

在JDBC中处理事务，都是通过Connection完成的。

同一事务中所有的操作，都在使用同一个Connection对象。

①JDBC中的事务

Connection的三个方法与事务有关：

- setAutoCommit（boolean）:设置是否为自动提交事务，如果true（默认值为true）表示自动提交，也就是每条执行的SQL语句都是一个单独的事务，如果设置为false，那么相当于开启了事务了；**con.setAutoCommit(false) 表示开启事务。**
- **commit（）：提交结束事务。**
- **rollback（）：回滚结束事务。**

示例代码

```
try{
    con.setAutoCommit(false);//开启事务
```

```
    ......
    con.commit();//try的最后提交事务
} catch（） {
    con.rollback();//回滚事务
}
```

嵌套事务实现

spring 事务嵌套:外层事务TraB,内层事务TraA、TraC

场景1：

TraA、TraC @Transactional（默认REQUIRED）

TraB:

```
    traA.update(order1); (traA.update throw new RuntimeException();)
    traC.update(order2);
```

结果:内外层事务全部回滚;

场景2：

TraA、TraC @Transactional（默认REQUIRED）

TraB:

```
    traA.update(order1); (traA.update throw new RuntimeException();try catch
traC.update)
traC.update(order2);
```

结果:内外层事务全部不回滚，traA中try catch后的事务提交;

场景3：

TraA、TraC @Transactional（默认REQUIRED）

```
TraB:  try{(traA.update throw new RuntimeException();
          在外层TraB try catch TraA)
                    traA.update(order1);
                    }catch(){}
               traC.update(order2);
```

结果:内外层事务全部回滚,内层的异常抛出到外层捕获也会回滚;

场景4：

TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC @Transactional（默认REQUIRED）

TraB:

```
               traA.update(order1);    (traA.update throw new RuntimeException();)
               traC.update(order2);
```

结果:内层事务回滚,外层事务继续提交;

场景5:
TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC @Transactional（默认REQUIRED）
TraB:
traA.update(order1); (traA.update **throw new** RuntimeException();**try catch** traC.update)
traC.update(order2);
结果:内外层事务全部不回滚，traA中**try catch**后的事务提交,达到与场景2的同样效果;

场景6:
TraA @Transactional(propagation=Propagation.REQUIRES_NEW)、TraC @Transactional（默认REQUIRED）
TraB:
try{ (traA.update **throw new** RuntimeException();在外层TraB **try catch** TraA)
traA.update(order1);
} **catch**
traC.update(order2);
结果:内层事务回滚,外层事务不回滚;

更多参考：https://blog.csdn.net/yangchangyong0/article/details/51960143

分布式事务实现；

1）基于XA协议的两阶段提交（2PC）

XA 规范主要 定义了（全局）事务管理器（Transaction Manager）和（局部）资源管理器（Resource Manager）之间的接口。

2）两阶段提交

事务的提交分为两个阶段：

预提交阶段(Pre-**Commit** Phase)

决策后阶段（Post-Decision Phase）

3）补偿事务（TCC）

针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段

Try 阶段主要是对业务系统做检测及资源预留

Confirm 阶段主要是对业务系统做确认提交，Try 阶段执行成功并开始执行 **Confirm** 阶段时，默认**Confirm** 阶段是不会出错的。即：只要 Try 成功，**Confirm** 一定成功。

Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放

4）本地消息表（MQ 异步确保）

其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。

5）MQ 事务消息

有一些第三方的 MQ 是支持事务消息的，比如 RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交，但是市面上一些主流的 MQ 都是不支持事务消息的，比如 RabbitMQ 和 Kafka 都不支持。

6）Sagas 事务模型

该模型其核心思想就是拆分分布式系统中的长事务为多个短事务，或者叫多个本地事务，然后由 Sagas 工作流引擎负责协调，如果整个流程正常结束，那么就算是业务成功完成，如果在这过程中实现失败，那么Sagas工作流引擎就会以相反的顺序调用补偿操作，重新进行业务回滚。

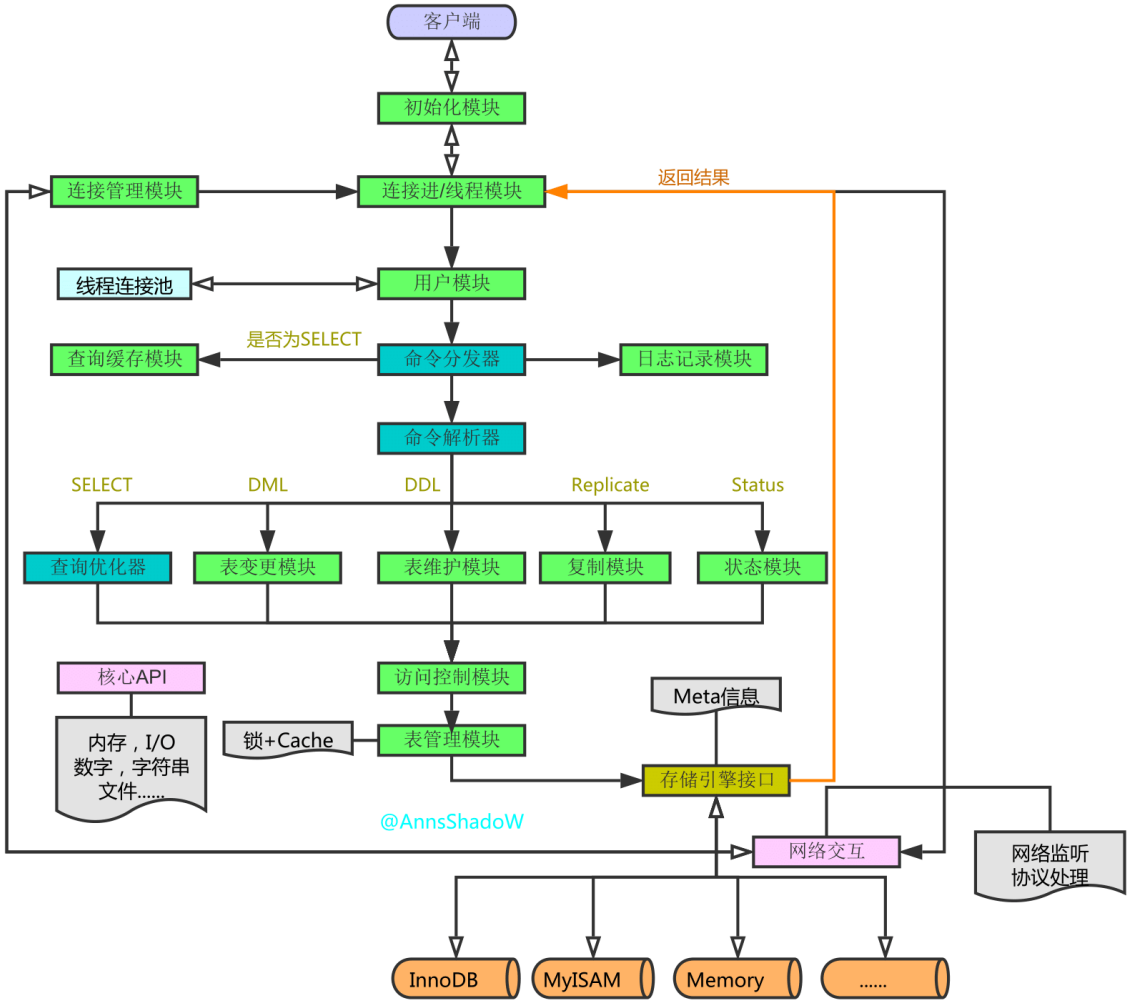
7）其他补偿方式

加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。在某些特殊的情况下，还会有 "人工补偿" 的，这也是最后一道屏障。

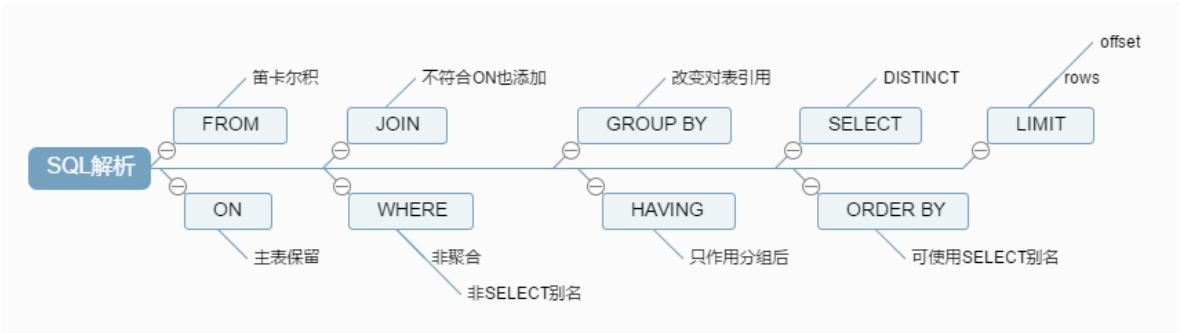
参考：https://www.javazhiyin.com/573.html

SQL的整个解析、执行过程原理、SQL行转列；

整体架构



SQL解析



行转列、列转行

参考：https://blog.csdn.net/jx_870915876/article/details/52403472

七、Redis

Redis为什么这么快?

(1) 绝大部分请求是纯粹的内存操作（非常快速）

(2) 采用单线程,避免了不必要的上下文切换和竞争条件

(3) 非阻塞IO - IO多路复用

redis采用多线程会有哪些问题?

1）单线程的问题

无法发挥多核CPU性能，单进程单线程只能跑满一个CPU核

可以通过在单机开多个Redis实例来完善

可以通过数据分片来增加吞吐量，问题（不支持批量操作、扩缩容复杂等）

2）多线程的问题

多线程处理可能涉及到锁

多线程处理会涉及到线程切换而消耗CPU

参考：阿里云Redis多线程性能提升思路解析

Redis支持哪几种数据结构；

String 、**List** 、**Set** 、**Hash** 、ZSet

Redis跳跃表的问题；

Redis只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群结点中用作内部数据结构

参考：https://blog.csdn.net/idwtwt/article/details/80233859

Redis是单进程单线程的，如何能够高并发?

采用多路 **I/O** 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗）

Redis如何使用Redis实现分布式锁?

参考：Redis分布式锁的正确实现方式

Redis分布式锁操作的原子性，Redis内部是如何实现的?

setnx

Incrby\Decrby

java终级面试题

Vector,ArrayList, LinkedList的区别是什么?

答：

1. Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。
2. List中的元素有序、允许有重复的元素，Set中的元素无序、不允许有重复元素。
3. Vector线程同步，ArrayList、LinkedList线程不同步。
4. LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。

5. ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省空间。

HashTable, HashMap，TreeMap区别?

答:

1. HashTable线程同步，HashMap非线程同步。
2. HashTable不允许<键,值>有空值，HashMap允许<键,值>有空值。
3. HashTable使用Enumeration，HashMap使用Iterator。
4. HashTable中hash数组的默认大小是11，增加方式的old*2+1，HashMap中hash数组的默认大小是16，增长方式一定是2的指数倍。
5. TreeMap能够把它保存的记录根据键排序，默认是按升序排序。

HashMap的数据结构

jdk1.8之前list + 链表

jdk1.8之后list + 链表（当链表长度到8时，转化为红黑树）

HashMap的扩容因子

默认0.75，也就是会浪费1/4的空间，达到扩容因子时，会将list扩容一倍，0.75 是时间与空间一个平衡值;

多线程修改HashMap

多线程同时写入，同时执行扩容操作，多线程扩容可能死锁、丢数据；可以对HashMap 加入同步锁Collections.synchronizedMap(hashMap)，但是效率很低，因为该锁是互斥锁，同一时刻只能有一个线程执行读写操作，这时候应该使用ConcurrentHashMap

LinkedHashMap

Java LinkedHashMap工作原理及实现

Java集合框架：LinkedHashMap

注意：在使用Iterator遍历的时候，LinkedHashMap会产生

`java.util.ConcurrentModificationException`。

扩展HashMap增加双向链表的实现，号称是最占内存的数据结构。支持iterator()时按Entry的插入顺序来排序(但是更新不算， 如果设置accessOrder属性为true，则所有读写访问都算)。实现上是在Entry上再增加属性before/after指针，插入时把自己加到Header Entry的前面去。如果所有读写访问都要排序，还要把前后Entry的before/after拼接起来以在链表中删除掉自己。

说说你知道的几个Java集合类：list、set、queue、map实现类

描述一下ArrayList和LinkedList各自实现和区别

Java基础篇(四):ArrayList和LinkedList内部实现、区别、使用场景

Java中的队列都有哪些，有什么区别

1. ArrayDeque, （数组双端队列）
2. PriorityQueue, （优先级队列）
3. ConcurrentLinkedQueue, （基于链表的并发队列）
4. DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）
5. ArrayBlockingQueue, （基于数组的并发阻塞队列）
6. LinkedBlockingQueue, （基于链表的FIFO阻塞队列）
7. LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）
8. PriorityBlockingQueue, （带优先级的无界阻塞队列）
9. SynchronousQueue （并发同步阻塞队列）

反射中，Class.forName和classloader的区别

java反射中，Class.forName和classloader的区别(代码说话)

Java7、Java8的新特性

java7,8的几个特性

Java数组和链表两种结构的操作效率，在哪些情况下(从开头开始，从结尾开始，从中间开始)，哪些操作(插入，查找，删除)的效率高

讲讲IO里面的常见类，字节流、字符流、接口、实现类、方法阻塞

Java IO流详解（二）——IO流的框架体系

讲讲NIO

NIO技术概览

缓冲区

虚拟内存&&内存空间的映射

三个channel使用

ServerSocketChannel||SocketChannel||FileChannel

Java NIO系列教程（八） SocketChannel

Java NIO系列教程（九） ServerSocketChannel

Java NIO系列教程（七） FileChannel

String 编码UTF-8 和GBK的区别

- GBK编码：是指中国的中文字符，其实它包含了简体中文与繁体中文字符，另外还有一种字符“gb2312”，这种字符仅能存储简体中文字符。
- UTF-8编码：它是一种全国家通过的一种编码，如果你的网站涉及到多个国家的语言，那么建议你选择UTF-8编码。

GBK和UTF8有什么区别?

UTF8编码格式很强大，支持所有国家的语言，正是因为它的强大，才会导致它占用的空间大小要比GBK大，对于网站打开速度而言，也是有一定影响的。

GBK编码格式，它的功能少，仅限于中文字符，当然它所占用的空间大小会随着它的功能而减少，打开网页的速度比较快。

什么时候使用字节流、什么时候使用字符流

什么时候使用字节流、什么时候使用字符流，二者的区别

递归读取文件夹下的文件，代码怎么实现

```
/**
 * 递归读取文件夹下的 所有文件
 *
 * @param testFileDir 文件名或目录名
 */
private static void testLoopOutAllFileName(String testFileDir) {
    if (testFileDir == null) {
        //因为new File(null)会空指针异常,所以要判断下
        return;
    }
    File[] testFile = new File(testFileDir).listFiles();
    if (testFile == null) {
        return;
    }
    for (File file : testFile) {
        if (file.isFile()) {
            System.out.println(file.getName());
        } else if (file.isDirectory()) {
            System.out.println("-------this is a directory, and its files are as follows:--------");
            testLoopOutAllFileName(file.getPath());
        } else {
            System.out.println("文件读入有误！");
        }
    }
}
```

Object.finalize

深入分析Object.finalize方法的实现原理

SynchronousQueue实现原理

SynchronousQueue实现原理

跳表SkipList

跳表（SkipList）及ConcurrentSkipListMap源码解析

Collections.sort排序算法

深入jdk——追踪Collections.sort 引发的bug（1）mergeSort

自定义类加载器

JVM——自定义类加载器

Java并发和并行

- 并发 : 是指两个或多个事件在同一时间间隔发生,在一台处理器上“同时”处理多个任务;
- 并行 : 是指两个或者多个事件在同一时刻发生,在多台处理器上同时处理多个任务。

怎么提高并发量，请列举你所知道的方案?

高并发解决方案——提升高并发量服务器性能解决思路

系统的用户量有多少？多用户并发访问时如何解决?

大型网站是怎样解决多用户高并发访问的

说说阻塞队列的实现：可以参考ArrayBlockingQueue的底层实现（锁和同步都行）

Java阻塞队列ArrayBlockingQueue和LinkedBlockingQueue实现原理分析

进程通讯的方式：消息队列，共享内存，信号量，socket通讯等

Linux进程间通信方式--信号，管道，消息队列，信号量，共享内存

用过并发包的哪些类

Excutors可以产生哪些线程池

为什么要用线程池

为什么要使用线程池

线程池的基础概念

core,maxPoolSize,keepalive

执行任务时

1. 如果线程池中线程数量 < core，新建一个线程执行任务;
2. 如果线程池中线程数量 >= core ,则将任务放入任务队列
3. 如果线程池中线程数量 >= core 且 < maxPoolSize，则创建新的线程;
4. 如果线程池中线程数量 > core ,当线程空闲时间超过了keepalive时，则会销毁线程;

由此可见线程池的队列如果是无界队列，那么设置线程池最大数量是无效的;

自带线程池的各种坑

1. Executors.newFixedThreadPool(10);

固定大小的线程池：

它的实现new ThreadPoolExecutor(10, 10, 0L, TimeUnit.MILLISECONDS,new LinkedBlockingQueue<Runnable>());

初始化一个指定线程数的线程池，其中corePoolSize == maximumPoolSize，使用LinkedBlockingQuene作为阻塞队列，当线程池没有可执行任务时，也不会释放线程。

由于LinkedBlockingQuene的特性，这个队列是无界的，若消费不过来，会导致内存被任务队列占满，最终oom；

2. Executors.newCachedThreadPool();

缓存线程池：

它的实现new ThreadPoolExecutor(0,Integer.MAX_VALUE,60L, TimeUnit.SECONDS,new SynchronousQueue<Runnable>());

初始化一个可以缓存线程的线程池，默认缓存60s，线程池的线程数可达到Integer.MAX_VALUE，即2147483647，内部使用SynchronousQueue作为阻塞队列；和newFixedThreadPool创建的线程池不同，newCachedThreadPool在没有任务执行时，当线程的空闲时间超过keepAliveTime，会自动释放线程资源，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销，因为线程池的最大值了Integer.MAX_VALUE，会导致无限创建线程；所以，使用该线程池时，一定要注意控制并发的任务数，否则创建大量的线程会导致严重的性能问题;

3. Executors.newSingleThreadExecutor()

单线程线程池：

同newFixedThreadPool线程池一样，队列用的是LinkedBlockingQueue无界队列，可以无限的往里面添加任务，直到内存溢出；

volatile关键字的用法：使多线程中的变量可见

Java并发编程：volatile关键字解析

线程的几种状态

线程在一定条件下，状态会发生变化。线程一共有以下几种状态：

1. 新建状态(New)：新创建了一个线程对象。
2. 就绪状态(Runnable)：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于“可运行线程池”中，变得可运行，只等待获取CPU的使用权。即在就绪状态的进程除CPU之外，其它的运行所需资源都已全部获得。
3. 运行状态(Running)：就绪状态的线程获取了CPU，执行程序代码。
4. 阻塞状态(Blocked)：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

阻塞的情况分三种：

a. 等待阻塞：运行的线程执行wait()方法，该线程会释放占用的所有资源，JVM会把该线程放入“等待池”中。进入这个状态后，是不能自动唤醒的，必须依靠其他线程调用notify()或notifyAll()方法才能被唤醒，
b. 同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入“锁池”中。
c. 其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5. 死亡状态(Dead)：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

常用的线程池模式以及不同线程池的使用场景

java线程池与五种常用线程池策略使用与解析

newFixedThreadPool此种线程池如果线程数达到最大值后会怎么办，底层原理。

多线程之间通信的同步问题，synchronized锁的是对象，衍伸出和synchronized相关很多的具体问题，例如同一个类不同方法都有synchronized锁，一个对象是否可以同时访问。或者一个类的static构造方法加上synchronized之后的锁的影响。

了解可重入锁的含义，以及ReentrantLock 和synchronized的区别

同步的数据结构，例如concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

atomicinteger和Volatile等线程安全操作的关键字的理解和使用

CAS和volatile关键字

通过volatile修饰的变量可以保证线程之间的可见性，但并不能保证字节码指令的原子执行，在多线程并发执行下，无法做到线程安全，得到正确的结果

1. 加锁(低效率)
2. cas

引用占小狼的简书：面试必问的CAS，要多了解

线程间通信，wait和notify

wait和notify的理解与使用

定时线程的使用

场景：在一个主线程中，要求有大量(很多很多)子线程执行完之后，主线程才执行完成。多种方式，考虑效率。

java线程池主线程等待子线程执行完成

进程和线程的区别

- 进程：

是具有一定独立功能的程序、它是系统进行资源分配和调度的一个独立单位，重点在系统调度和单独的单位，也就是说进程是可以独立运行的一段程序(比如正在运行的某个java程序)。

- 线程：

他是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源(一个线程只能属于一个进程，而一个进程可以有多个线程)。

什么叫线程安全？举例说明

- java中的线程安全是什么：

就是线程同步的意思，就是当一个程序对一个线程安全的方法或者语句进行访问的时候，其他的不能再对他进行操作了，必须等到这次访问结束以后才能对这个线程安全的方法进行访问

- 什么叫线程安全：

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，
就是线程安全的。
或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。
线程安全问题都是由全局变量及静态变量引起的。若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则就可能影响线程安全。
存在竞争的线程不安全，不存在竞争的线程就是安全的

并发、同步的接口或方法

Java并发编程的类、接口和方法

HashMap 是否线程安全，为何不安全。 ConcurrentHashMap，线程安全，为何安全。底层实现是怎么样的。

深入浅出ConcurrentHashMap1.8
谈谈ConcurrentHashMap1.7和1.8的不同实现
ConcurrentHashMap的红黑树实现分析
深入分析ConcurrentHashMap1.8的扩容实现
老生常谈，HashMap的死循环

J.U.C下的常见类的使用。 ThreadPool的深入考察； BlockingQueue的使用。（take，poll的区别，put，offer的区别）；原子类的实现。volatile的理解

java volatile关键字解惑

面试必问的volatile，你了解多少？

Tomcat并发

Tomcat的性能与最大并发配置

有个每秒钟5k个请求，查询手机号所属地的笔试题(记得不完整，没列出)，如何设计算法?请求再多，比如5w，如何设计整个系统?

高并发情况下，我们系统是如何支撑大量的请求的

- 尽量使用缓存，包括用户缓存，信息缓存等，多花点内存来做缓存，可以大量减少与数据库的交互，提高性能。
- 用jprofiler等工具找出性能瓶颈，减少额外的开销。
- 优化数据库查询语句，减少直接使用hibernate等工具的直接生成语句（仅耗时较长的查询做优化）。
- 优化数据库结构，多做索引，提高查询效率。
- 统计的功能尽量做缓存，或按每天一统计或定时统计相关报表，避免需要时进行统计的功能。
- 能使用静态页面的地方尽量使用，减少容器的解析（尽量将动态内容生成静态html来显示）。
- 解决以上问题后，使用服务器集群来解决单台的瓶颈问题。

1. HTML静态化

效率最高、消耗最小的就是纯静态化的html页面，所以尽可能使网站上的页面采用静态页面来实现，这个最简单的方法其实也是最有效的方法。但是对于大量内容并且频繁更新的网站，无法全部手动去挨个实现，于是出现了常见的信息发布系统CMS，像常访问的各个门户站点的新闻频道，甚至他们的其他频道，都是通过信息发布系统来管理和实现的，信息发布系统可以实现最简单的信息录入自动生成静态页面，还能具备频道管理、权限管理、自动抓取等功能，对于一个大型网站来说，拥有一套高效、可管理的CMS是必不可少的。

2. 图片服务器分离

对于Web服务器来说，不管是Apache、IIS还是其他容器，图片是最消耗资源的，于是有必要将图片与页面进行分离，这是基本上大型网站都会采用的策略，他们都有独立的图片服务器，甚至很多台图片服务器。这样的架构可以降低提供页面访问请求的服务器系统压力，并且可以保证系统不会因为图片问题而崩溃，在应用服务器和图片服务器上，可以进行不同的配置优化，比如apache在配置ContentType的时候可以尽量少支持，尽可能少的LoadModule，保证更高的系统消耗和执行效率。 这一实现起来是比较容易的一现，如果服务器集群操作起来更方便，如果是

独立的服务器，新手可能出现上传图片只能在服务器本地的情况下，可以在令一台服务器设置的IIS采用网络路径来实现图片服务器，即不用改变程序，又能提高性能，但对于服务器本身的IO处理性能是没有任何的改变。

3. 数据库集群和库表散列

大型网站都有复杂的应用，这些应用必须使用数据库，那么在面对大量访问的时候，数据库的瓶颈很快就能显现出来，这时一台数据库将很快无法满足应用，于是需要使用数据库集群或者库表散列。

4. 缓存

缓存一词搞技术的都接触过，很多地方用到缓存。网站架构和网站开发中的缓存也是非常重要。架构方面的缓存，对Apache比较熟悉的人都能知道Apache提供了自己的缓存模块，也可以使用外加的Squid模块进行缓存，这两种方式均可以有效的提高Apache的访问响应能力。

网站程序开发方面的缓存，Linux上提供的Memory Cache是常用的缓存接口，可以在web开发中使用，比如用Java开发的时候就可以调用MemoryCache对一些数据进行缓存和通讯共享，一些大型社区使用了这样的架构。另外，在使用web语言开发的时候，各种语言基本都有自己的缓存模块和方法，PHP有Pear的Cache模块，Java就更多了，.net不是很熟悉，相信也肯定有。

5. 镜像

镜像是大型网站常采用的提高性能和数据安全性的方式，镜像的技术可以解决不同网络接入商和地域带来的用户访问速度差异，比如ChinaNet和EduNet之间的差异就促使了很多网站在教育网内搭建镜像站点，数据进行定时更新或者实时更新。在镜像的细节技术方面，这里不阐述太深，有很多专业的现成的解决架构和产品可选。也有廉价的通过软件实现的思路，比如Linux上的rsync等工具。

6. 负载均衡

负载均衡将是大型网站解决高负荷访问和大量并发请求采用的终极解决办法。 负载均衡技术发展了多年，有很多专业的服务提供商和产品可以选择。

硬件四层交换

第四层交换使用第三层和第四层信息包的报头信息，根据应用区间识别业务流，将整个区间段的业务流分配到合适的应用服务器进行处理。　第四层交换功能就象是虚IP，指向物理服务器。它传输的业务服从的协议多种多样，有HTTP、FTP、NFS、Telnet或其他协议。这些业务在物理服务器基础上，需要复杂的载量平衡算法。在IP世界，业务类型由终端TCP或UDP端口地址来决定，在第四层交换中的应用区间则由源端和终端IP地址、TCP和UDP端口共同决定。

在硬件四层交换产品领域，有一些知名的产品可以选择，比如Alteon、F5等，这些产品很昂贵，但是物有所值，能够提供非常优秀的性能和很灵活的管理能力。Yahoo中国当初接近2000台服务器使用了三四台Alteon就搞定了。

集群如何同步会话状态

集群session一致性和同步问题

基于ZooKeeper的分布式Session实现

负载均衡的原理

六大Web负载均衡原理与实现

如果有一个特别大的访问量，到数据库上，怎么做优化（DB设计，DBIO，SQL优化，Java优化）

数据库SQL优化大总结之 百万级数据库优化方案

如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问题

如何提高服务器并发处理能力

假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。

如何查找 造成 性能瓶颈出现的位置，是哪个位置照成性能瓶颈。

你的项目中使用过缓存机制吗？有没用用户非本地缓存

Semphore、CountDownLatch、CyclicBarrier、Phaser

并发工具类（二）同步屏障CyclicBarrier

并发工具类（三）控制并发线程数的Semaphore

CLH队列

【Java并发编程实战】—— AQS(四)：CLH同步队列

JAVA并发编程学习笔记之CLH队列锁

产生死锁的必要条件

操作系统：死锁的产生、条件、和解锁

Java内存模型

设计模式

单例模式：饱汉、饿汉。以及饿汉中的延迟加载,双重检查

工厂模式、装饰者模式、观察者模式。

工厂方法模式的优点（低耦合、高内聚，开放封闭原则）

如何理解观察者模式?

列举出你说熟悉的设计模式，并对其中的一种的使用举一个例子。

JVM

User user = new User() 做了什么操作，申请了哪些内存?

1. new User(); 创建一个User对象，内存分配在堆上
2. User user; 创建一个引用，内存分配在栈上
3. = 将User对象地址赋值给引用

Java的内存模型以及GC算法

JVM内存模型与GC算法

jvm性能调优都做了什么

JVM性能调优

介绍JVM中7个区域，然后把每个区域可能造成内存的溢出的情况说明

介绍GC 和GC Root不正常引用

自己从classload 加载方式，加载机制说开去，从程序运行时数据区，讲到内存分配，讲到String常量池，讲到JVM垃圾回收机制，算法，hotspot。反正就是各种扩展

java classload 机制 详解

jvm 如何分配直接内存， new 对象如何不分配在堆而是栈上，常量池解析

JVM直接内存

浅谈HotSpot逃逸分析

触摸java常量池

数组多大放在 JVM 老年代（不只是设置 PretenureSizeThreshold ，问通常多大，没做过一问便知）

深入理解JVM（6）：Java对象内存分配策略

注意:PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效，Parallel Scavenge收集器不认识这个参数，Parallel Scavenge收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑ParNew加CMS的收集器组合。

老年代中数组的访问方式

GC算法，永久代对象如何GC，GC有环怎么处理

永久代GC的原因:

- 永久代空间已经满了
- 调用了System.gc()

注意： 这种GC是full GC 堆空间也会一并被GC一次

GC有环怎么处理

1. 根搜索算法

什么是根搜索算法

垃圾回收器从被称为GC Roots的点开始遍历遍历对象，凡是可以达到的点都会标记为存活，堆中不可到达的对象都会标记成垃圾，然后被清理掉。

GC Roots有哪些

- 类，由系统类加载器加载的类。这些类从不会被卸载，它们可以通过静态属性的方式持有对象的引用。

注意，一般情况下由自定义的类加载器加载的类不能成为GC Roots

- 线程，存活的线程
- Java方法栈中的局部变量或者参数
- JNI方法栈中的局部变量或者参数
- JNI全局引用
- 用做同步监控的对象

被JVM持有的对象，这些对象由于特殊的目的不被GC回收。这些对象可能是系统的类加载器，一些重要的异常处理类，一些为处理异常预留的对象，以及一些正在执行类加载的自定义的类加载器。但是具体有哪些前面提到的对象依赖于具体的JVM实现。

2. 如何处理

基于引用对象遍历的垃圾回收器可以处理循环引用，只要是涉及到的对象不能从GC Roots强引用可到达，垃圾回收器都会进行清理来释放内存。

谁会被GC，什么时候GC

Java JVM：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）

如果想不被GC怎么办

如果想在 GC 中生存 1 次怎么办

生存一次，释放掉对象的引用，但是在对象的finalize方法中重新建立引用，但是此方法只会被调用一次，所以能在GC中生存一次。

分析System.gc()方法

JVM源码分析之SystemGC完全解读

JVM 选项 -XX:+UseCompressedOops 有什么作用？为什么要使用?

当你将你的应用从 32 位的 JVM 迁移到 64 位的 JVM 时，由于对象的指针从 32 位增加到了 64 位，因此堆内存会突然增加，差不多要翻倍。这也会对 CPU 缓存（容量比内存小很多）的数据产生不利的影响。因为，迁移到 64 位的 JVM 主要动机在于可以指定最大堆大小，通过压缩 OOP 可以节省一定的内存。通过 -XX:+UseCompressedOops 选项，JVM 会使用 32 位的 OOP，而不是 64 位的 OOP。

写代码分别使得JVM的堆、栈和持久代发生内存溢出(栈溢出)

JVM内存溢出详解（栈溢出，堆溢出，持久代溢出以及无法创建本地线程）

为什么jdk8用metaspace数据结构用来替代perm?

JDK8：PermGen变更为MetaSpace详解

简单谈谈堆外内存以及你的理解和认识

JVM源码分析之堆外内存完全解读

threadlocal使用场景及注意事项

threadlocal原理及常用应用场景

JVM老年代和新生代的比例?

JVM-堆学习之新生代老年代持久带的使用关系

栈是运行时的单位，而堆是存储的单位。

栈解决程序的运行问题，即程序如何执行，或者说如何处理数据；堆解决的是数据存储的问题，即数据怎么放、放在哪儿。

在Java中一个线程就会相应有一个线程栈与之对应，这点很容易理解，因为不同的线程执行逻辑有所不同，因此需要一个独立的线程栈。而堆则是所有线程共享的。栈因为是运行单位，因此里面存储的信息都是跟当前线程（或程序）相关信息的。包括局部变量、程序运行状态、方法返回值等等；而堆只负责存储对象信息。

为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗?

1. 从软件设计的角度看，栈代表了处理逻辑，而堆代表了数据。这样分开，使得处理逻辑更为清晰。分而治之的思想。这种隔离、模块化的思想在软件设计的方方面面都有体现。
2. 堆与栈的分离，使得堆中的内容可以被多个栈共享（也可以理解为多个线程访问同一个对象）。这种共享的收益是很多的。一方面这种共享提供了一种有效的数据交互方式(如：共享内存)，另一方面，堆中的共享常量和缓存可以被所有栈访问，节省了空间。
3. 栈因为运行时的需要，比如保存系统运行的上下文，需要进行地址段的划分。由于栈只能向上增长，因此就会限制住栈存储内容的能力。而堆不同，堆中的对象是可以根据需要动态增长的，因此栈和堆的拆分，使得动态增长成为可能，相应栈中只需记录堆中的一个地址即可。
4. 面向对象就是堆和栈的完美结合。其实，面向对象方式的程序与以前结构化的程序在执行上没有任何区别。但是，面向对象的引入，使得对待问题的思考方式发生了改变，而更接近于自然方式的思考。当我们把对象拆开，你会发现，对象的属性其实就是数据，存放在堆中；而对象的行为（方法），就是运行逻辑，放在栈中。我们在编写对象的时候，其实即编写了数据结构，也编写的处理数据的逻辑。不得不承认，面向对象的设计，确实很美。

为什么不把基本类型放堆中呢?

因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象

本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

堆中存什么？栈中存什么？

堆中存的是对象。栈中存的是基本数据类型和堆中对象的引用。一个对象的大小是不可估计的，或者说是可以动态变化的，但是在栈中，一个对象只对应了一个4btye的引用（堆栈分离的好处：））。

为什么不把基本类型放堆中呢？因为其占用的空间一般是1~8个字节——需要空间比较少，而且因为是基本类型，所以不会出现动态增长的情况——长度固定，因此栈中存储就够了，如果把他存在堆中是没有什么意义的（还会浪费空间，后面说明）。可以这么说，基本类型和对象的引用都是存放在栈中，而且都是几个字节的一个数，因此在程序运行时，他们的处理方式是统一的。但是基本类型、对象引用和对象本身就有所区别了，因为一个是栈中的数据一个是堆中的数据。最常见的一个问题就是，Java中参数传递时的问题。

Java中的参数传递时传值呢？还是传引用？

要说明这个问题，先要明确两点：

1. 不要试图与C进行类比，Java中没有指针的概念
2. 程序运行永远都是在栈中进行的，因而参数传递时，只存在传递基本类型和对象引用的问题。不会直接传对象本身。

明确以上两点后。Java在方法调用传递参数时，因为没有指针，所以它都是进行传值调用（这点可以参考C的传值调用）。因此，很多书里面都说Java是进行传值调用，这点没有问题，而且也简化的C中复杂性。

但是传引用的错觉是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

堆和栈中，栈是程序运行最根本的东西。程序运行可以没有堆，但是不能没有栈。而堆是为栈进行数据存储服务，说白了堆就是一块共享的内存。不过，正是因为堆和栈的分离的思想，才使得Java的垃圾回收成为可能。

Java中，栈的大小通过-Xss来设置，当栈中存储数据比较多时，需要适当调大这个值，否则会出现java.lang.StackOverflowError异常。常见的出现这个异常的是无法返回的递归，因为此时栈中保存的信息都是方法返回的记录点

对象引用类型分为哪几类？

java中四种引用类型（对象的强、软、弱和虚引用）

讲一讲内存分代及生命周期。

管中窥豹——从对象的生命周期梳理JVM内存结构、GC、类加载、AOP编程及性能监控

什么情况下触发垃圾回收?

Java JVM 8：垃圾回收（GC 在什么时候，对什么东西，做了什么事情）

如何选择合适的垃圾收集算法?

深入理解垃圾收集器和收集器的选择策略

JVM给了三种选择：串行收集器、并行收集器、并发收集器 ，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前系统配置 进行判断。

吞吐量优先的并行收集器

如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。

典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

-XX:+UseParallelGC ：选择垃圾收集器为并行收集器。 此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。

-XX:ParallelGCThreads=20 ：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20

-XX:+UseParallelOldGC

-XX:+UseParallelOldGC ：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100

-XX:MaxGCPauseMillis=100 ：设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:MaxGCPauseMillis=100

-XX:+UseAdaptiveSizePolicy

-XX:+UseAdaptiveSizePolicy ：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器

如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。

典型配置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC

-XX:+UseConcMarkSweepGC ：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。

-XX:+UseParNewGC :设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection

-XX:CMSFullGCsBeforeCompaction ：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。

-XX:+UseCMSCompactAtFullCollection ：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

StringTable

探索StringTable提升YGC性能

JVM源码分析之String.intern()导致的YGC不断变长

JVM中最大堆大小有没有限制?

JVM调优常用参数配置

如何进行JVM调优？有哪些方法?

如何合理的规划一次jvm性能调优

如何理解内存泄漏问题？有哪些情况会导致内存泄露？如何解决?

详解java内存泄露和如何避免内存泄漏

开源框架

hibernate和ibatis的区别

Ibatis与Hibernate的区别

讲讲mybatis的连接池。

《深入理解mybatis原理》 Mybatis数据源与连接池

spring框架中需要引用哪些jar包，以及这些jar包的用途

springMVC的原理

Spring MVC工作原理 及注解说明

spring中beanFactory和ApplicationContext的联系和区别

Spring系列之beanFactory与ApplicationContext

spring注入的几种方式（循环注入）

Spring循环依赖的三种方式

spring如何实现事务管理的

springIOC

Spring IOC原理解读 面试必读

spring AOP的原理

Spring AOP 实现原理与 CGLIB 应用

spring AOP 两种代理方式

回答为什么要用什么方法这种问题的时候，通常首先要回答两个问题，第一个就是，我要做什么事情，第二个就是，不同方法的优劣是什么。

首先，我要做什么事情。

这里的回答比较简单，就是代理Java类/接口。那么，两者在完成这件事情上，有什么差别呢

JDK Proxy	Cglib Proxy
只能代理接口	以继承的方式完成代理，不能代理被final修饰的类

实际上，大部分的Java类都会以接口-实现的方式来完成，因此，在这个方面上，JDK Proxy实际上是比Cglib Proxy要更胜一筹的。因为如果一个类被final修饰，则Cglib Proxy无法进行代理。

其次，两种方法的优劣又在什么地方呢?

我们可以参考一下来自bytebuddy的数据，这个是在代理一个实现了具有18个方法的接口的类，时间单位为ns。

| JDK Proxy | Cglib Proxy
---|---|---
生成代理类时间 | 1'060.766 | 960.527
方法调用时间 | 0.008 | 0.003
来源 | JDK原生代码 | 第三方库，更新频率较低

不难看出，其实Cglib代理的性能是要远远好于JDK代理的。

其实从原理也能理解，直接通过类的方法调用，肯定要比通过反射调用的时间更短。但是从来源来看的话，一个是JDK原生代码，而另一个则是第三方的开源库。JDK原生代码无疑使用的人会更多范围也更广，会更佳稳定，而且还有可能在未来的JDK版本中不断优化性能。

而Cglib更新频率相对来说比较低了，一方面是因为这个代码库已经渐趋稳定，另一方面也表明后续这个库可能相对来说不会有大动作的优化维护。

对比完之后，再来回看这个问题，为什么要使用两种方式呢?

在功能上讲，实际上Cglib代理并不如JDK代理（如果大家都按接口-实现的方式来设计类）。但是从效率上将，Cglib远胜JDK代理啊！所以，为了提高效率，同时又保有在未来，当JDK代理的

性能也能够同样好的时候，使用更佳稳定靠谱的JDK代码，这种可能，于是采取了这种设计。

hibernate中的1级和2级缓存的使用方式以及区别原理（Lazy-Load的理解）

Hibernate的原理体系架构，五大核心接口，Hibernate对象的三种状态转换，事务管理。

Spring boot 热加载

Spring boot 热加载

Spring Boot设置有效时间和自动刷新缓存，时间支持在配置文件中配置

Spring Boot缓存实战 Redis 设置有效时间和自动刷新缓存，时间支持在配置文件中配置

Spring 如何保证 Controller 并发的安全?

springMVC一个Controller处理所有用户请求的并发问题

spring中用到哪些设计模式?

spring中用到哪些设计模式

Spring IOC 的理解，其初始化过程?

Spring IoC容器初始化过程学习

Spring的事务管理

Spring事务管理（详解+实例）

MyBatis缓存

《深入理解mybatis原理》 MyBatis缓存机制的设计与实现

MyBatis数据源与连接池

《深入理解mybatis原理》 Mybatis数据源与连接池

分布式

CAP原理和BASE理论

CAP原则(CAP定理)、BASE理论

分布式事务、分布式锁

常用的分布式事务解决方案介绍有多少种?

分布式锁的几种实现方式

分布式存储系统

redis

redis和memcache的区别;

用redis做过什么；

redis是如何持久化的：rdb和aof；

Redis数据类型

Redis五种数据类型介绍

redis集群如何同步；

Redis 复制与集群

redis的数据添加过程是怎样的：哈希槽；

redis集群实现（四） 数据的和槽位的分配

redis的淘汰策略有哪些；

Redis 内存淘汰机制

- volatile-lru -> 根据LRU算法删除带有过期时间的key。
- allkeys-lru -> 根据LRU算法删除任何key。
- volatile-random -> 根据过期设置来随机删除key, 具备过期时间的key。
- allkeys->random -> 无差别随机删, 任何一个key。
- volatile-ttl -> 根据最近过期时间来删除（辅以TTL）, 这是对于有过期时间的key
- noeviction -> 谁也不删，直接在写操作时返回错误。

redis有哪些数据结构；

redis的单线程模型

Redis 网络架构及单线程模型

redis 集群基础

- 所有的redis节点彼此互联(PING-PONG机制),内部使用二进制协议优化传输速度和带宽.
- 节点的fail是通过集群中超过半数的master节点检测失效时才生效.
- 客户端与redis节点直连,不需要中间proxy层.客户端不需要连接集群所有节点,连接集群中任何一个可用节点即可
- redis-cluster把所有的物理节点映射到[0-16383]slot上,cluster 负责维护node<->slot<->key.
- 如果存入一个值，按照redis cluster哈希槽的算法： CRC16('key')384 = 6782。 那么就会把这个key 的存储分配到对应的master上

redis Cluster主从模式

- 如果进群超过半数以上master挂掉，无论是否有slave集群进入fail状态,所以集群中至少应该有奇数个节点，所以至少有三个节点，每个节点至少有一个备份节点,
- redis cluster 为了保证数据的高可用性，加入了主从模式，一个主节点对应一个或多个从节点，主节点提供数据存取，从节点则是从主节点拉取数据备份，当这个主节点挂掉

后，就会有这个从节点选取一个来充当主节点，从而保证集群不会挂掉。

zookeeper

zookeeper是什么；

zookeeper哪里用到；

zookeeper的选主过程；

zookeeper集群之间如何通讯

zookeeper系列之通信模型

你们的zookeeper的节点加密是用的什么方式

分布式锁的实现过程；

kafka

传递保证语义：

- At most once：消息可能会丢，但绝不会重复传递。
- At least once：消息绝不会丢，但可能会重复传递。
- Exactly once： 每条消息只会被传递一次。

生产者的“Exactly once”语义方案

当生产者向Kafka发送消息，且正常得到响应的时候，可以确保生产者不会产生重复的消息。但是，如果生产者发送消息后，遇到网络问题，无法获取响应，生产者就无法判断该消息是否成功提交给了Kafka。根据生产者的机制，我们知道，当出现异常时，会进行消息重传，这就可能出现“At least one”语义。为了实现“Exactly once”语义，这里提供两个可选方案：

- 每个分区只有一个生产者写入消息，当出现异常或超时的情况时，生产者就要查询此分区的最后一个消息，用来决定后续操作是消息重传还是继续发送。
- 为每个消息添加一个全局唯一主键，生产者不做其他特殊处理，按照之前分析方式进行重传，由消费者对消息进行去重，实现“Exactly once”语义。

如果业务数据产生消息可以找到合适的字段作为主键，或是有一个全局ID生成器，可以优先考虑选用第二种方案。

消费者的“Exactly once”语义方案

为了实现消费者的“Exactly once”语义，在这里提供一种方案，供读者参考：消费者将关闭自动提交offset的功能且不再手动提交offset，这样就不使用Offsets Topic这个内部Topic记录其offset，而是由消费者自己保存offset。这里利用事务的原子性来实现“Exactly once”语义，我们将offset和消息处理结果放在一个事务中，事务执行成功则认为此消息被消费，否则事务回滚需要重新消费。当出现消费者宕机重启或Rebalance操作时，消费者可以从关系型数据库中找到对应的offset，然后调用KafkaConsumer.seek()方法手动设置消费位置，从此offset处开始继续消费。

ISR集合

ISR（In-SyncReplica）集合表示的是目前“可用”（alive）且消息量与Leader相差不多的副本集合，这是整个副本集合的一个子集。“可用”和“相差不多”都是很模糊的描述，其实际含义是ISR集合中的副本必须满足下面两个条件：

1. 副本所在节点必须维持着与ZooKeeper的连接。
2. 副本最后一条消息的offset与Leader副本的最后一条消息的offset之间的差值不能超出指定的阈值。

每个分区中的Leader副本都会维护此分区的ISR集合。写请求首先由Leader副本处理，之后Follower副本会从Leader上拉取写入的消息，这个过程会有一定的延迟，导致Follower副本中保存的消息略少于Leader副本，只要未超出阈值都是可以容忍的。如果一个Follower副本出现异常，比如：宕机，发生长时间GC而导致Kafka僵死或是网络断开连接导致长时间没有拉取消息进行同步，就会违反上面的两个条件，从而被Leader副本踢出ISR集合。当Follower副本从异常中恢复之后，会继续与Leader副本进行同步，当Follower副本“追上”（即最后一条消息的offset的差值小于指定阈值）Leader副本的时候，此Follower副本会被Leader副本重新加入到ISR中。

请说明什么是Apache Kafka?

Apache Kafka是由Apache开发的一种发布订阅消息系统，它是一个分布式的、分区的和重复的日志服务。

请说明什么是传统的消息传递方法?

传统的消息传递方法包括两种：

- 排队：在队列中，一组用户可以从服务器中读取消息，每条消息都发送给其中一个人。
- 发布-订阅：在这个模型中，消息被广播给所有的用户。

请说明Kafka相对传统技术有什么优势?

Apache Kafka与传统的消息传递技术相比优势之处在于：

- 快速:单一的Kafka代理可以处理成千上万的客户端，每秒处理数兆字节的读写操作。
- 可伸缩:在一组机器上对数据进行分区和简化，以支持更大的数据
- 持久:消息是持久性的，并在集群中进行复制，以防止数据丢失
- 设计:它提供了容错保证和持久性

在Kafka中broker的意义是什么?

在Kafka集群中，broker术语用于引用服务器。

Kafka服务器能接收到的最大信息是多少?

Kafka服务器可以接收到的消息的最大大小是1000000字节。

解释Kafka的Zookeeper是什么?我们可以在没有Zookeeper的情况下使用Kafka吗?

Zookeeper是一个开放源码的、高性能的协调服务，它用于Kafka的分布式应用。
不，不可能越过Zookeeper，直接联系Kafka broker。一旦Zookeeper停止工作，它就不能服务客户端请求。
Zookeeper主要用于在集群中不同节点之间进行通信
在Kafka中，它被用于提交偏移量，因此如果节点在任何情况下都失败了，它都可以从之前提交的偏移量中获取
除此之外，它还执行其他活动，如: leader检测、分布式同步、配置管理、识别新节点何时离开或连接、集群、节点实时状态等等。

解释Kafka的用户如何消费信息?

在Kafka中传递消息是通过使用sendfile API完成的。它支持将字节从套接口转移到磁盘，通过内核空间保存副本，并在内核用户之间调用内核。

解释如何提高远程用户的吞吐量?

如果用户位于与broker不同的数据中心，则可能需要调优套接口缓冲区大小，以对长网络延迟进行摊销。

解释一下，在数据制作过程中，你如何能从Kafka得到准确的信息?

在数据中，为了精确地获得Kafka的消息，你必须遵循两件事: 在数据消耗期间避免重复，在数据生产过程中避免重复。
这里有两种方法，可以在数据生成时准确地获得一个语义:

- 每个分区使用一个单独的写入器，每当你发现一个网络错误，检查该分区中的最后一条消息，以查看您的最后一次写入是否成功
- 在消息中包含一个主键(UUID或其他)，并在用户中进行反复制

解释如何减少ISR中的扰动?broker什么时候离开ISR?

ISR是一组与leaders完全同步的消息副本，也就是说ISR中包含了所有提交的消息。ISR应该总是包含所有的副本，直到出现真正的故障。如果一个副本从leader中脱离出来，将会从ISR中删除。

Kafka为什么需要复制?

Kafka的信息复制确保了任何已发布的消息不会丢失，并且可以在机器错误、程序错误或更常见些的软件升级中使用。

如果副本在ISR中停留了很长时间表明什么?

如果一个副本在ISR中保留了很长一段时间，那么它就表明，跟踪器无法像在leader收集数据那样快速地获取数据。

请说明如果首选的副本不在ISR中会发生什么?

如果首选的副本不在ISR中，控制器将无法将leadership转移到首选的副本。

有可能在生产后发生消息偏移吗?

在大多数队列系统中，作为生产者的类无法做到这一点，它的作用是触发并忘记消息。broker将完成剩下的工作，比如使用id进行适当的元数据处理、偏移量等。
作为消息的用户，你可以从Kafka broker中获得补偿。如果你注视SimpleConsumer类，你会注意到它会获取包括偏移量作为列表的MultiFetchResponse对象。此外，当你对Kafka消息进行迭代时，你会拥有包括偏移量和消息发送的MessageAndOffset对象。

kafka与传统的消息中间件对比

kafka与传统的消息中间件对比

KAFKA：如何做到1秒发布百万级条消息

KAFKA：如何做到1秒发布百万级条消息

kafka文件存储

Kafka文件存储机制那些事

据kafka官网吹，如果随机写入磁盘，速度就只有100KB每秒。顺序写入的话，7200转/s的磁盘就能达到惊人的600MB每秒！
操作系统对文件访问做了优化，文件会在内核空间分页做缓存（pageCache）。写入时先写入pageCache。由操作系统来决定何时统一写入磁盘。操作系统会使用顺序写入。
Kafka深入理解-1：Kafka高效的文件存储设计
kafka的log存储解析——topic的分区partition分段segment以及索引等

dubbo

默认使用的是什么通信框架，还有别的选择吗?

默认也推荐使用netty框架，还有mina。

服务调用是阻塞的吗?

默认是阻塞的，可以异步调用，没有返回值的可以这么做。

一般使用什么注册中心？还有别的选择吗?

推荐使用zookeeper注册中心，还有redis等不推荐。

默认使用什么序列化框架，你知道的还有哪些?

默认使用Hessian序列化，还有Duddo、FastJson、Java自带序列化。

服务提供者能实现失效踢出是什么原理?

服务失效踢出基于zookeeper的临时节点原理。

服务上线怎么不影响旧版本?

采用多版本开发，不影响旧版本。

如何解决服务调用链过长的问题?

可以结合zipkin实现分布式服务追踪。

说说核心的配置有哪些?

核心配置有 dubbo:service/ dubbo:reference/ dubbo:protocol/ dubbo:registry/ dubbo:application/ dubbo:provider/ dubbo:consumer/ dubbo:method/

dubbo推荐用什么协议?

默认使用dubbo协议。

同一个服务多个注册的情况下可以直连某一个服务吗?

可以直连，修改配置即可，也可以通过telnet直接某个服务。

画一画服务注册与发现的流程图

流程图见dubbo.io。

Dubbo集群容错怎么做?

读操作建议使用Failover失败自动切换，默认重试两次其他服务器。写操作建议使用Failfast快速失败，发一次调用失败就立即报错。

在使用过程中都遇到了些什么问题?

使用过程中的问题可以百度

dubbo和dubbox之间的区别?

dubbox是当当网基于dubbo上做了一些扩展，如加了服务可restful调用，更新了开源组件等。

你还了解别的分布式框架吗?

别的还有spring的spring cloud，facebook的thrift，twitter的finagle等。

dubbo重试雪崩

Dubbo超时机制导致的雪崩连接

TCP/IP

算法

使用随机算法产生一个数，要求把1-1000W之间这些数全部生成。（考察高效率，解决产生冲突的问题)

两个有序数组的合并排序

一个数组的倒序

计算一个正整数的正平方根

说白了就是常见的那些查找、排序算法以及各自的时间复杂度

二叉树的遍历算法

DFS,BFS算法

比较重要的数据结构，如链表，队列，栈的基本理解及大致实现。

排序算法与时空复杂度（快排为什么不稳定，为什么你的项目还在用）

逆波兰计算器

Hoffman 编码

查找树与红黑树

如何给100亿个数字排序?

如何给100亿个数字排序?

统计海量数据中出现次数最多的前10个IP

哈希分治法 - 统计海量数据中出现次数最多的前10个IP

排序算法时间复杂度

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性	复杂性
直接插入排序	O(n2)	O(n2)	O(n)	O(1)	稳定	简单
希尔排序	O(nlog2n)	O(n2)	O(n)	O(1)	不稳定	较复杂
直接选择排序	O(n2)	O(n2)	O(n2)	O(1)	不稳定	简单
堆排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(1)	不稳定	较复杂
冒泡排序	O(n2)	O(n2)	O(n)	O(1)	稳定	简单
快速排序	O(nlog2n)	O(n2)	O(nlog2n)	O(nlog2n)	不稳定	较复杂
归并排序	O(nlog2n)	O(nlog2n)	O(nlog2n)	O(n)	稳定	较复杂
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(n+r)	稳定	较复杂

判断链表中是否有环

判断链表中是否有环 ----- 有关单链表中环的问题

hash算法及常用的hash算法

常见hash算法的原理

查找算法

七大查找算法

设计与思想

重构过代码没有？说说经验；

一千万的用户实时排名如何实现；

海量积分数据实时排名处理

如何高效地获得玩家排名？

基于redis的zset实现排行榜功能

五万人并发抢票怎么实现；

Web大规模高并发请求和抢购的解决方案

有个每秒钟5k个请求，查询手机号所属地的笔试题(记得不完整，没列出)，如何设计算法?请求再多，比如5w，如何设计整个系统?

高并发情况下，我们系统是如何支撑大量的请求的

集群如何同步会话状态

集群服务器Session同步

负载均衡的原理

如果有一个特别大的访问量，到数据库上，怎么做优化（DB设计，DBIO，SQL优化，Java优化)

大数据量高并发的数据库优化

如果出现大面积并发，在不增加服务器的基础上，如何解决服务器响应不及时问题“。

如何提高服务器并发处理能力

假如你的项目出现性能瓶颈了，你觉得可能会是哪些方面，怎么解决问题。

三个方面解决性能问题的基本思路和方法

一个Web应用的性能瓶颈一般有哪些呢？通常采取什么手段解决呢？

如何查找 造成 性能瓶颈出现的位置，是哪个位置照成性能瓶颈。

五步定位性能瓶颈

性能测试如何定位瓶颈

你的项目中使用过缓存机制吗？有没用用户非本地缓存

使用缓存的9大误区（上）

使用缓存的9大误区（下）

网站缓存技术

Tomcat优化

Tomcat 调优及 JVM 参数优化

网络通信

http是无状态通信，http的请求方式有哪些，可以自己定义新的请求方式么。

socket通信，以及长连接，分包，连接异常断开的处理。

Socket TCP/IP协议数据传输过程中的粘包和分包问题

深入理解socket网络异常

socket通信模型的使用，AIO和NIO。

socket框架netty的使用，以及NIO的实现原理，为什么是异步非阻塞。

同步和异步，阻塞和非阻塞。

OSI七层模型，包括TCP,IP的一些基本知识

OSI七层协议模型和TCP/IP四层模型比较

http中，get post的区别

- get： 从服务器上获取数据，也就是所谓的查，仅仅是获取服务器资源，不进行修改。
- post: 向服务器提交数据，这就涉及到了数据的更新，也就是更改服务器的数据。
- 请求方式的区别: get 请求的数据会附加在URL之后,特定的浏览器和服务器对URL的长度有限制. post 更加安全数据不会暴漏在url上,而且长度没有限制.

HTTP报文内容

HTTP请求报文和HTTP响应报文

说说http,tcp,udp之间关系和区别。

说说浏览器访问http://www.taobao.com，经历了怎样的过程。

访问 www.taobao.com过程

HTTP协议、 HTTPS协议，SSL协议及完整交互过程；

HTTPS协议，SSL协议及完整交互过程

tcp的拥塞，快回传，ip的报文丢弃

https处理的一个过程，对称加密和非对称加密

head各个特点和区别

ping的原理

Ping过程 原理 详解(图)

ARP/RARP

TCP/IP协议详解笔记——ARP协议和RARP协议

DNS解析过程

DNS域名解析的过程

Http会话的四个过程

建立连接，发送请求，返回响应，关闭连接。

数据库MySql

MySql的存储引擎的不同

MySQL存储引擎之Myisam和Innodb总结性梳理

MySql参数

Mysql优化系列（1）--Innodb引擎下mysql自身配置优化

单个索引、联合索引、主键索引

Mysql主键索引、唯一索引、普通索引、全文索引、组合索引的区别

Mysql怎么分表，以及分表后如果想按条件分页查询怎么办(如果不是按分表字段来查询的话，几乎效率低下，无解)

如果按时间排序查询，使用limit n （不要使用limit m, n 页数多了之后效率低）然后记录最后一条的时间，下次从最后一条的时间开始查询

mysql 数据库 分表后 怎么进行分页查询？Mysql分库分表方案?

mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。

分表之后想让一个id多个表是自增的，效率实现

MySQL分表自增ID解决方案

分布式id生成算法

理解分布式id生成算法SnowFlake

MySql的主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离

Mysql主从同步的实现原理

MySQL索引

浅谈算法和数据结构: 十 平衡查找树之B树

MySQL索引背后的数据结构及算法原理

MySQL索引失效的几种情况

事务的四个特性，以及各自的特点（原子、隔离）等等，项目怎么解决这些问题

数据库的锁：行锁，表锁；乐观锁，悲观锁

mysql的锁--行锁，表锁，乐观锁，悲观锁

mysql for update语法实现行锁

MySQL 加锁处理分析

数据库事务的几种粒度；

理解事务 - MySQL 事务处理机制

MVCC

Mysql中的MVCC

聚簇索引

mysql索引原理之聚簇索引

关系型和非关系型数据库区别

- 关系型数据库：是指采用了关系模型(二维表格模型)来组织数据的数据库。
- 非关系型数据库：以键值对存储，且结构不固定.

MySql死锁排查

我的Mysql死锁排查过程（案例分析）

MySql优化

MySQL 对于千万级的大表要怎么优化?

Linux

介绍一下epoll

kill的用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）

硬链接和软链接的区别

grep的使用

进程间的通信，共享内存方式的优缺点

swap分区

Swap交换分区概念

Linux系统swappiness参数在内存与交换分区之间优化作用

overcommit_memory

取值为0，系统在为应用进程分配虚拟地址空间时，会判断当前申请的虚拟地址空间大小是否超过剩余内存大小，如果超过，则虚拟地址空间分配失败。因此，也就是如果进程本身占用的虚拟地址空间比较大或者剩余内存比较小时，fork、malloc等调用可能会失败。

取值为1，系统在为应用进程分配虚拟地址空间时，完全不进行限制，这种情况下，避免了fork可能产生的失败，但由于malloc是先分配虚拟地址空间，而后通过异常陷入内核分配真正的物理内存，在内存不足的情况下，这相当于完全屏蔽了应用进程对系统内存状态的感知，即malloc总是能成功，一旦内存不足，会引起系统OOM杀进程，应用程序对于这种后果是无法预测的

取值为2，则是根据系统内存状态确定了虚拟地址空间的上限，由于很多情况下，进程的虚拟地址空间占用远大小其实际占用的物理内存，这样一旦内存使用量上去以后，对于一些动态产生的

进程(需要复制父进程地址空间)则很容易创建失败，如果业务过程没有过多的这种动态申请内存或者创建子进程，则影响不大，否则会产生比较大的影响

[又一次内存分配失败(关于overcommit_memory)]

linux系统下查看CPU、内存负载情况

[linux系统下查看CPU、内存负载情况]