1. **性能调优**

**性能优化如何理解：**

**性能基准**

**程序代码在部署的服务器环境下运行所占用的CPU和内存资源存在一定的标准，满足运行要求。在大量用户访问接口时服务器的基础硬件设施是否有显著的飙升，能否满足代码执行的基本要求。**

**什么是性能调优**

**使用较少的资源提供更好的服务。**

**调优策略：**

**1.使用空间换时间，内存、缓存就是这种情况，直接从内存中读取数据比直接从磁盘中读取数据效率更高**

**2.使用时间换空间，在处理大批量数据时，常常将数据进行分割，进行分批处理，比如大附件上传时**

**3.分而治之，把任务切分，分开执行，使用并发执行来提高效率**

**4.异步处理，有些任务执行的时间比较长，可以拆分业务，防止阻塞现象。常用的异步处理方式通过MQ消息中间件来实现**

**5.使用CDN技术，将用户需要的资源放在离用户近的地方**

**6.一切可扩展，业务模块化、服务化、良好的水平扩展能力**

**衡量性能**

**指标：**

**1.响应时间和吞吐量**

**应用程序完成数据传输的时间，用户请求HTTP到应用程序获取数据库数据到返回结果的时长，可以根据数据库表的体量来优化应用程序的响应时间。**

**吞吐量也是衡量数据传输性能的一个指标，是指单位时间内应用程序处理用户的请求次数。**

**2.平均负载**

**程序在执行过程的内核使用数量必须小于服务器的最大内核数量，超过最大内核数，一旦超过**

**最大内核数量，服务器将处于压力状态下。**

**3.错误率**

**HTTP传输失败的总百分比**

**4.GC率和暂停时间**

**GC暂停频率和使用时间**

**5.业务指标**

**应用程序的性能不一定完全取决于响应时间和错误率，跟具体的业务需求也要一定的关系**

**6.正常运行时间和服务运行状态**

**程序代码持续正常运行的时间，程序健壮性的衡量指标，随时监测服务的运行状态，也可以监测服务运行质量**

**7.日志大小**

**程序执行日志有一个缺点就是随着代码的执行，程序日志是累计增加的，当日志过大时会影响服务器运行的效率，因此我们需要随时关注日志大小。**

**目前通常的解决办法是使用LOGSTASH划分使用日志，并将它们发送并存储在SPLUNK、ELK或其他的日志管理工具中。**

**JVM调优**

**Jvm虚拟机内部剖析**

**1.类装载子系统**

**负责在运行时查找和加载类文件的类，JVM有多个类加载器：**

**引导类加载器：加载JAVA核心类库的类：java.lang.object、java.lang.string**

**扩展类加载器：加载JAVA扩展ext类库的类**

**系统类加载器：加载我们自己写的java类和main方法类、特殊情况下也会加载自定的类加载器**

**2.运行时数据区**

**堆、栈、方法区、程序计数器**

**堆区：JVM启动时创建，所有的实例对象都是在这里分配内存，这里也是GC(垃圾回收)重点清理的区域。如果内存不够分配给对象实例时，就会抛出内存溢出的异常。**

**栈：方法执行时会创建一个栈帧，栈帧用于存放局部变量、操作数栈、动态链接和方法出口等信息，基本变量和对象的应用变量都存放于此。**

**方法区：存放一些加载类的信息(包括类的版本、字段、方法和接口等描述信息)、常量、静态变量等信息**

**程序技术器：描述方法具体执行到什么位置**

**3.字节码执行引擎**

**将字节码分配给运行时数据区，执行引擎并读取字节码并逐段执行**

**垃圾收集器**

**垃圾收集器就是垃圾回收算法和内存回收的具体实现。**

**串行垃圾收集器：serial、serialOld**

**并行垃圾收集器：parNew、parallel old、parallel scavenge**

**并发垃圾收集器：CMS**

**G1垃圾收集器**

**新生代GC：是指发生在新生代的垃圾回收机制，因为JAVA对象大多都具备朝生夕灭的的特性、所以Minor GC一般比较频繁，一般回收速度也比较快**

**老年代GC：是指发生在老年代的垃圾回收，如果发生了major GC通常会伴随着至少一次的minor GC，老年代的垃圾回收通常比新生代的垃圾回收慢上10被左右**

**实战调优案例与解决方法**

**高性能硬件上的程序部署策略：**

**1.通过64位JDK来使用大内存**

**使用大内存的前提是有把握应用程序的FULL GC控制得足够低，至少低到不会影响用户的使用。控制FULL GC的频率的关键是看应用中绝大多数对象能否符合“朝生夕灭”的规律，即大多数对象的生存时间不能够太长**

**内存回收导致长时间的停顿**

**64位JDK的性能可能不如32位的JDK**

**需要保证程序稳定，因为程序一旦发生溢出，存储快照会过于庞大而无法分析**

**程序在64位JDK消耗的内存要比32位JDK的大**

**2.使用若干个32位虚拟机建立逻辑集群来利用硬件资源**

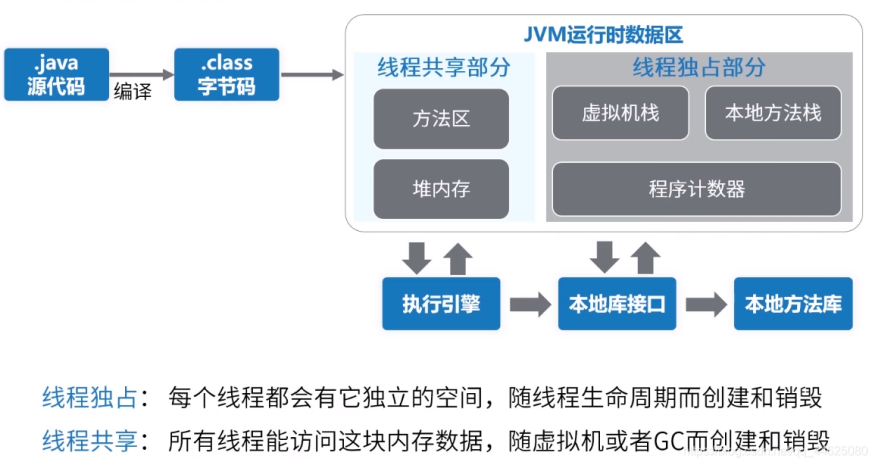
**在逻辑集群的环境下，要尽量避免节点竞争全局的资源，比如磁盘竞争、各个节点如果同时访问磁盘，会很容易导致IO异常**

**很难高效地使用的资源池，因为一般实在各个节点建立独立的连接池，这样可能导致一部分节点资源池满了，另一个节点资源池没满**

**各个节点会受到32位的内存限制**

**大量使用本地缓存的应用，在逻辑集群中会照成大量的内存浪费，因为每个节点都有一份缓存，可以考虑使用集中式缓存**

**Jvm运行时区**



**线程独占：每个线程都会有它独立的空间，随线程生命周期而创建和销毁**

**线程共享：所有线程能访问这块内存数据，随虚拟机或者GC创建和销毁**

**方法区：**

**JVM用来存储加载类的信息，常量、静态变量、编译后的代码等数据**

**堆内存：**

**老年代、新生代**

**JVM启动时存放对象的实例。垃圾回收器主要是管理堆内存。如果满了就会出现内存溢出异常**

**虚拟机栈：**

**没有线程在这个空间都有一个私有的空间**

**线程栈是由多个栈帧组成，一个线程包含一个或者多个方法，一个方法对应一个栈帧**

**栈帧的内容包括：局部变量表、操作数栈、动态链接、方法返回地址、附件信息等等**

**栈内存默认的大小是1M，超出则会报StackOverflowError异常**

**程序计术器：**

**记录当前线程执行字节码的位置，存储的字节码指令地址，如果执行Native方法，计数器为空**

**JAVA程序性能优化**

**优雅地创建对象**

**5种创建对象的方式：**

**1.使用new关键字**

**Student student = new Student();**

**Student student = new Student(“小明”);**

**2.使用Class类的newInstance方法**

**Student student = (Student) classs.forName(“com.dimple.NewObject.Student”)**

**.newInstance();**

**student.sayHello();**

**Student student = Student.class.newInstance();**

**这种方式，对象必须要有无参构造函数**

**3.使用Constructor的newInstance方法**

**Constructor<Student> constructor = Student.class.getConstructor(string.class);**

**Student constructorStudent = constructor.newInstance(“小明”);**

**4.使用clone的方法**

**Student clone = constructorStudent.clone();**

**clone.sayHello();**

**5.使用反序列化创建对象，这种方式需要实现Serializable接口**

**先序列化对象，在反序列化得到一个新的对象**

**ObjectOutputStream objectOutputStream = new ObjectOutputStream(new**

**FileOutputStream("data.o"));**

**objectOutputStream.writeObject(clone);**

**objectOutputStream.close();**

**ObjectInputStream objectInputStream = new ObjectInputStream(new**

**FileInputStream("data.o"));**

**Student studentSerialization = (Student) objectInputStream.readObject();**

**objectInputStream.close();**

**studentSerialization.sayHello();**

**注意对象的通用方法**

**类的设计陷阱**

**泛型需要注意的问题**

**·Java方法的哪些坑**

**Tomcat**

**线程模型分析**

**生产环境配置及调优**

**运行机制及框架**

**Mysql**

**探析Btree机制**

**执行计划深入分析**

**Mysql索引优化详解**

**慢查询分析与sql优化**

1. **应用框架源码解读**

**Spring IOC**

**Spring Framework体系结构**

**源码分析**

**BeanFactory源码分析**

**BeanDefinition源码分析**

**Bean生命周期**

**依赖实现**

**Spring AOP**

**AOP源码分析**

**Transaction源码分析**

**Spring Cache框架源码分析**

**Spring MVC**

**MVC简介与设计思想**

**SpringMVC组成**

**源码解读DispathServlet**

**Spring5新特性**

**容器增强**

**函数式编程**

**webFlux模块分析**

**Kotlin介绍**

**Testing改进**

**兼容性问题**

**Mysql**

**Mybatis组成**

**核心源码分析**

**手写mybatis框架**

**3、分布式**

**分布式架构思维**

**大型互联网架构演进过程**

**架构师因具备的分布式知识**

**主流分布式架构设计设计详解**

**架构开发基础**

**多线程开发**

**高性能NIO框架**

**架构核心服务层技术**

**服务的前世今生**

**深入理解通信协议**

**基于分布式解决RPC方案**

**Dubbo全解析**

**架构关键基础设施**

**分布式环境指挥官zookeeper**

**分布式消息通讯异步与MQ**

**分布式缓存**

**数据存储SQL&NOSQL**

**高并发分流技术Nginx**

**分布式解决方案**

**分布式解决方案**

**Session跨域共享实战**

**分布式事物解决方案实战**

**分布式锁解决方案实战**

**分布式单点登录实战SSO**

**分布式调度任务系统**

**分布式配置中心**

**4、微服务**

**SpringBoot**

**与微服务的区别与联系**

**快速构建SpringBoot工程**

**Springboot核心组件剖析**

**快速集成mybatis实战**

**快速集成dubbo及案例实战**

**快速集成redis及案例实战**

**构建swagger插件即实现API**

**管理及接口测试体系**

**SpringCloud**

**Zuul路由网关详解源码探析**

**Ribbon客户端负载均衡原理**

**Feign声明式服务调用方式**

**Hystrix服务熔断及服务降级**

**Eureka注册中心构件**

**Config配置服务中心**

**Svn、git快速构建**

**Sleuth调用链路跟踪**

**BUS消息总线技术**

**Docker虚拟化技术**

**介绍、安装及使用**

**Compose部署脚本**

**Service服务编排**

**Redis分布式集群部署**

**Docker file构建**

**通过maven插件打包镜像**

**部署及运行应用程序**

**Kubernetes编配**

**构建mysql集群实战**

**高可用springCloud微服务与docker集成实现动态扩容**

**5、团队协作开发**

**Git**

**Git基本工作原理**

**Git常用操作及问题处理**

**Maven**

**Maven重要指令**

**私服-nexus**

**Maven流行插件使用**

**手写自己的插件**

**架构师如何理解scope\lifecycle\phase\goal**

**Jenkins**

**Jenkins服务部署**

**Jenkins持续集成**

**Sonar**

**静态代码检查，代码更健壮代码走查**