阿里、京东、蚂蚁等大厂面试真题解析

遇到过哪些设计模式?

Java死锁如何避免?

深拷贝和浅拷贝

如果你提交任务时，线程池队列已满，这时会发生什么

谈谈ConcurrentHashMap的扩容机制

Spring中Bean是线程安全的吗?

说说你常用的Linux基本操作命令

Maven中Package和Install的区别

项目及主要负责的模块

SpringCloud各组件功能，与Dubbo的区别

京东二面

说说类加载器双亲委派模型

泛型中extends和super的区别

并发编程三要素?

Spring用到了哪些设计模式

简述CAP理论

图的深度遍历和广度遍历

快排算法

TCP的三次握手和四次挥手

消息队列如何保证消息可靠传输

画出项目架构图，介绍自己所处的模块

蚂蚁一面

二叉搜索树和平衡二叉树有什么关系?

强平衡二叉树和弱平衡二叉树有什么区别

B树和B+树的区别，为什么Mysql使用B+树

epoll和poll的区别

简述线程池原理，FixedThreadPool用的阻塞队列是什么

sychronized和ReentrantLock的区别

sychronized的自旋锁、偏向锁、轻量级锁、重量级锁，分别介绍和联系

HTTPS是如何保证安全传输的

蚂蚁二面

设计模式有哪些大类，及熟悉其中哪些设计模式

volatile关键字，他是如何保证可见性，有序性

Java的内存结构，堆分为哪几部分，默认年龄多大进入老年代

Mysql的锁你了解哪些

ConcurrentHashMap 如何保证线程安全，jdk1.8 有什么变化

讲一下OOM以及遇到这种情况怎么处理的，是否使用过日志分析工具

Mysql索引原理

介绍一下亮点的项目

项目的并发大概有多高，Redis的瓶颈是多少

项目中遇到线上问题怎么处理的，说一下印象最深刻的

作者图灵学院：周瑜

阿里一面

说一下ArrayList和LinkedList区别

1. 首先，他们的底层数据结构不同，ArrayList底层是基于数组实现的，LinkedList底层是基于链表实现的
2. 由于底层数据结构不同，他们所适用的场景也不同，ArrayList更适合随机查找，LinkedList更适合删除和添加，查询、添加、删除的时间复杂度不同
3. 另外ArrayList和LinkedList都实现了List接口，但是LinkedList还额外实现了Deque接口，所以LinkedList还可以当做队列来使用

说一下HashMap的Put方法

先说HashMap的Put方法的大体流程：

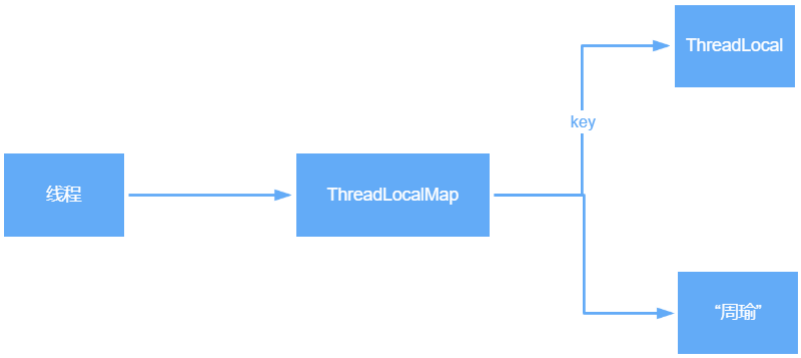
1. 根据Key通过哈希算法与与运算得出数组下标
2. 如果数组下标位置元素为空，则将key和value封装为Entry对象（JDK1.7中是Entry对象，JDK1.8中是Node对象）并放入该位置
3. 如果数组下标位置元素不为空，则要分情况讨论
    a. 如果是JDK1.7，则先判断是否需要扩容，如果要扩容就进行扩容，如果不用扩容就生成Entry对象，并使用头插法添加到当前位置的链表中
    b. 如果是JDK1.8，则会先判断当前位置上的Node的类型，看是红黑树Node，还是链表Node
        i. 如果是红黑树Node，则将key和value封装为一个红黑树节点并添加到红黑树中去，在这个过程中会判断红黑树中是否存在当前key，如果存在则更新value
        ii. 如果此位置上的Node对象是链表节点，则将key和value封装为一个链表Node并通过尾插法插入到链表的最后位置去，因为是尾插法，所以需要遍历链表，在遍历链表的过程中会判断是否

存在当前key，如果存在则更新value，当遍历完链表后，将新链表Node插入到链表中，插入到链表后，会看当前链表的节点个数，如果大于等于8，那么则会将该链表转成红黑树

iii. 将key和value封装为Node插入到链表或红黑树中后，再判断是否需要进行扩容，如果需要就扩容，如果不需要就结束PUT方法

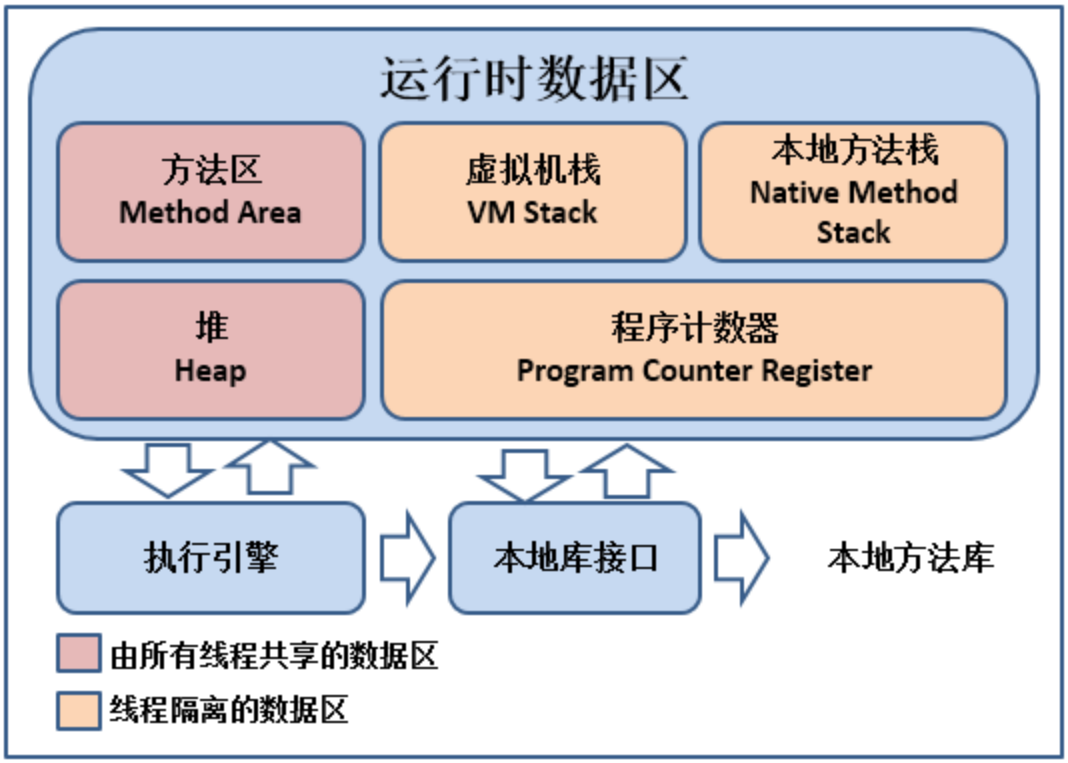
说一下ThreadLocal

1. ThreadLocal是Java中所提供的线程本地存储机制，可以利用该机制将数据缓存在某个线程内部，该线程可以在任意时刻、任意方法中获取缓存的数据
2. ThreadLocal底层是通过ThreadLocalMap来实现的，每个Thread对象（注意不是ThreadLocal对象）中都存在一个ThreadLocalMap，Map的key为ThreadLocal对象，Map的value为需要缓存的值
3. 如果在线程池中使用ThreadLocal会造成内存泄漏，因为当ThreadLocal对象使用完之后，应该要把设置的key，value，也就是Entry对象进行回收，但线程池中的线程不会回收，而线程对象是通过强引用指向ThreadLocalMap，ThreadLocalMap也是通过强引用指向Entry对象，线程不被回收，Entry对象也就不会被回收，从而出现内存泄漏，解决办法是，在使用了ThreadLocal对象之后，手动调用ThreadLocal的remove方法，手动清楚Entry对象
4. ThreadLocal经典的应用场景就是连接管理（一个线程持有一个连接，该连接对象可以在不同的方法之间进行传递，线程之间不共享同一个连接）



说一下JVM中，哪些是共享区，哪些可以作为gc root

1、堆区和方法区是所有线程共享的，栈、本地方法栈、程序计数器是每个线程独有的



2、什么是gc root，JVM在进行垃圾回收时，需要找到“垃圾”对象，也就是没有被引用的对象，但是直接找“垃圾”对象是比较耗时的，所以反过来，先找“非垃圾”对象，也就是正常对象，那么就需要从某些“根”开始去找，根据这些“根”的引用路径找到正常对象，而这些“根”有一个特征，就是它只会引用其他对象，而不会被其他对象引用，例如：栈中的本地变量、方法区中的静态变量、本地方法栈中的变量、正在运行的线程等可以作为gc root。

你们项目如何排查JVM问题

对于还在正常运行的系统：

1. 可以使用jmap来查看JVM中各个区域的使用情况
2. 可以通过jstack来查看线程的运行情况，比如哪些线程阻塞、是否出现了死锁
3. 可以通过jstat命令来查看垃圾回收的情况，特别是fullgc，如果发现fullgc比较频繁，那么就得进行调优了
4. 通过各个命令的结果，或者jvisualvm等工具来进行分析
5. 首先，初步猜测频繁发送fullgc的原因，如果频繁发生fullgc但是又一直没有出现内存溢出，那么表示fullgc实际上是回收了很多对象了，所以这些对象最好能在younggc过程中就直接回收掉，避免这些对象进入到老年代，对于这种情况，就要考虑这些存活时间不长的对象是不是比较大，导致年轻代放不下，直接进入到了老年代，尝试加大年轻代的大小，如果改完之后，fullgc减少，则证明修改有效
6. 同时，还可以找到占用CPU最多的线程，定位到具体的方法，优化这个方法的执行，看是否能避免某些对象的创建，从而节省内存

对于已经发生了OOM的系统：

1. 一般生产系统中都会设置当系统发生了OOM时，生成当时的dump文件（–XX:+HeapDumpOnOutOfMemoryError –XX:HeapDumpPath=/usr/local/base）
2. 我们可以利用jsisualvm等工具来分析dump文件
3. 根据dump文件找到异常的实例对象，和异常的线程（占用CPU高），定位到具体的代码
4. 然后再进行详细的分析和调试

总之，调优不是一蹴而就的，需要分析、推理、实践、总结、再分析，最终定位到具体的问题

如何查看线程死锁

1. 可以通过jstack命令来进行查看，jstack命令中会显示发生了死锁的线程
2. 或者两个线程去操作数据库时，数据库发生了死锁，这是可以查询数据库的死锁情况

```
1、查询是否锁表
show OPEN TABLES where In_use > 0;
2、查询进程
show processlist;
3、查看正在锁的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCKS;
4、查看等待锁的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_LOCK_WAITS;
```

线程之间如何进行通讯的

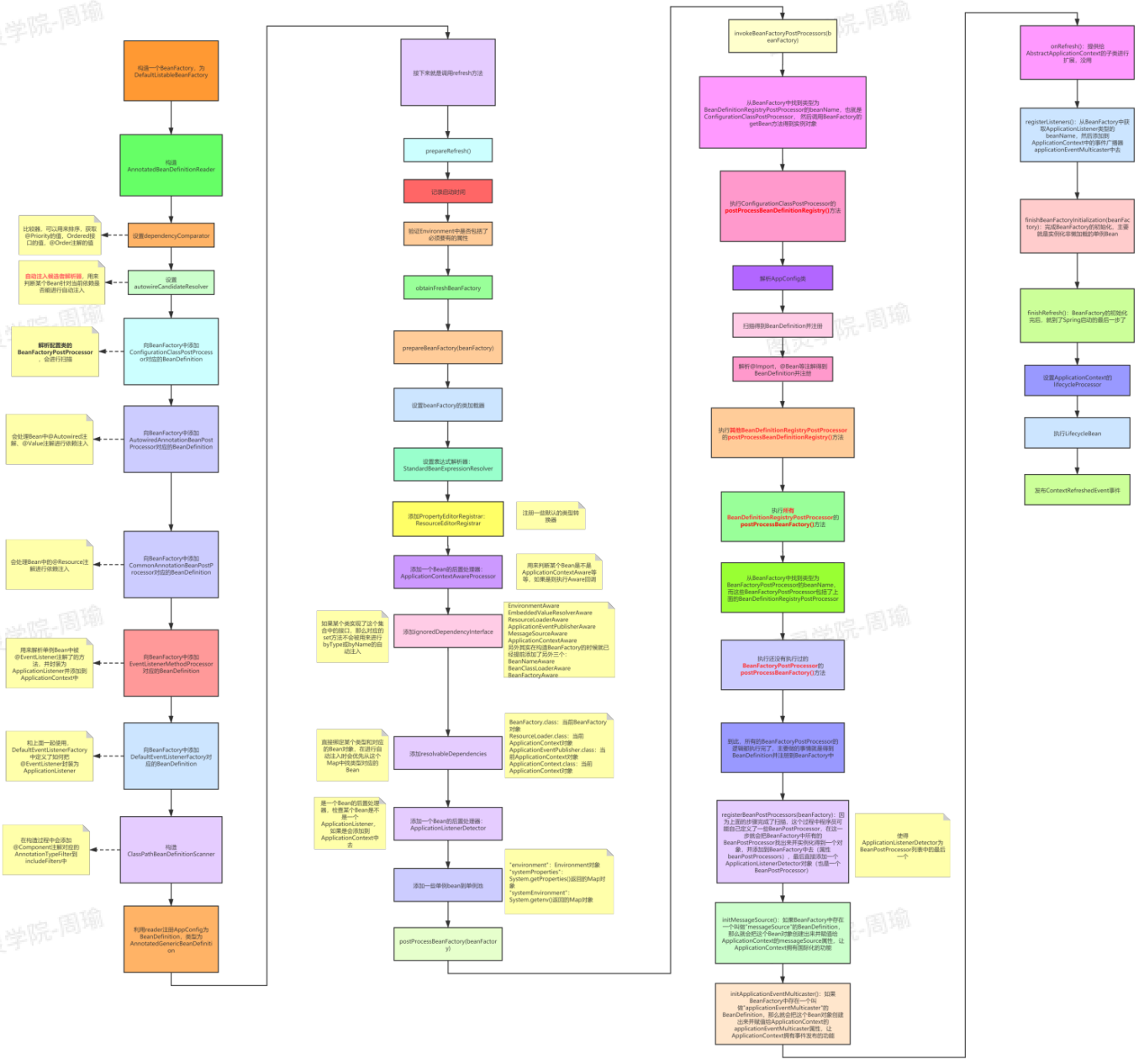
1. 线程之间可以通过共享内存或基于网络来进行通信
2. 如果是通过共享内存来进行通信，则需要考虑并发问题，什么时候阻塞，什么时候唤醒
3. 像Java中的wait()、notify()就是阻塞和唤醒
4. 通过网络就比较简单了，通过网络连接将通信数据发送给对方，当然也要考虑到并发问题，处理方式就是加锁等方式

介绍一下Spring，读过源码介绍一下大致流程

1. Spring是一个快速开发框架，Spring帮助程序员来管理对象
2. Spring的源码实现的是非常优秀的，设计模式的应用、并发安全的实现、面向接口的设计等
3. 在创建Spring容器，也就是启动Spring时：

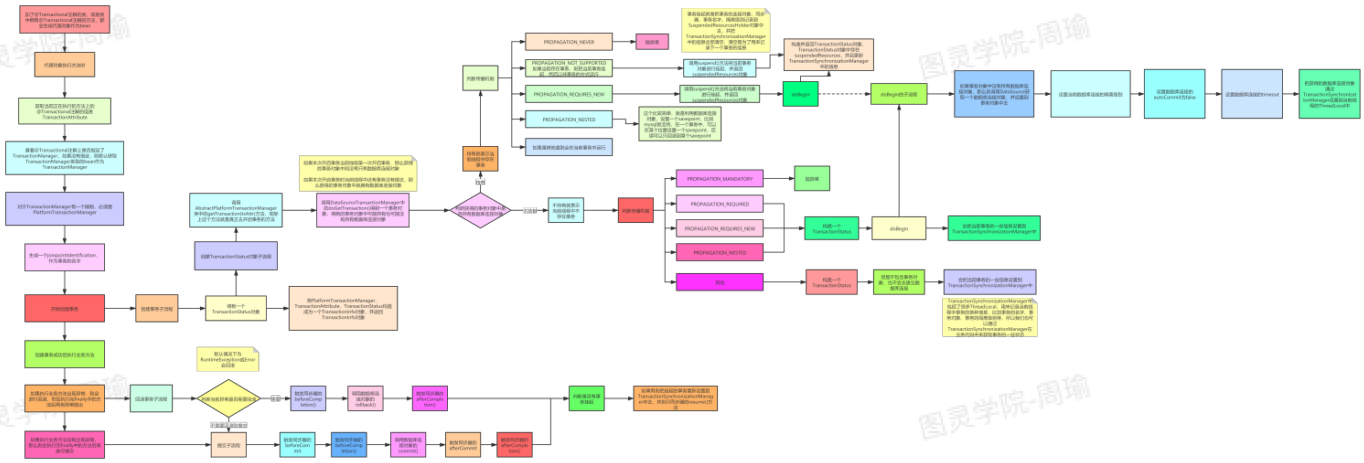
a. 首先会进行扫描，扫描得到所有的BeanDefinition对象，并存在一个Map中
b. 然后筛选出非懒加载的单例BeanDefinition进行创建Bean，对于多例Bean不需要在启动过程中去进行创建，对于多例Bean会在每次获取Bean时利用BeanDefinition去创建
c. 利用BeanDefinition创建Bean就是Bean的创建生命周期，这期间包括了合并BeanDefinition、推断构造方法、实例化、属性填充、初始化前、初始化、初始化后等步骤，其中AOP就是发生在初始化后这一步骤中

4. 单例Bean创建完了之后，Spring会发布一个容器启动事件
5. Spring启动结束
6. 在源码中会更复杂，比如源码中会提供一些模板方法，让子类来实现，比如源码中还涉及到一些BeanFactoryPostProcessor和BeanPostProcessor的注册，Spring的扫描就是通过BenaFactoryPostProcessor来实现的，依赖注入就是通过BeanPostProcessor来实现的
7. 在Spring启动过程中还会去处理@Import等注解



说一下Spring的事务机制

1. Spring事务底层是基于数据库事务和AOP机制的
2. 首先对于使用了@Transactional注解的Bean，Spring会创建一个代理对象作为Bean
3. 当调用代理对象的方法时，会先判断该方法上是否加了@Transactional注解
4. 如果加了，那么则利用事务管理器创建一个数据库连接
5. 并且修改数据库连接的autocommit属性为false，禁止此连接的自动提交，这是实现Spring事务非常重要的一步
6. 然后执行当前方法，方法中会执行sql
7. 执行完当前方法后，如果没有出现异常就直接提交事务
8. 如果出现了异常，并且这个异常是需要回滚的就会回滚事务，否则仍然提交事务
9. Spring事务的隔离级别对应的就是数据库的隔离级别
10. Spring事务的传播机制是Spring事务自己实现的，也是Spring事务中最复杂的
11. Spring事务的传播机制是基于数据库连接来做的，一个数据库连接一个事务，如果传播机制配置为需要新开一个事务，那么实际上就是先建立一个数据库连接，在此新数据库连接上执行sql



什么时候@Transactional失效

因为Spring事务是基于代理来实现的，所以某个加了@Transactional的方法只有是被代理对象调用时，那么这个注解才会生效，所以如果是被代理对象来调用这个方法，那么@Transactional是不会生效的。

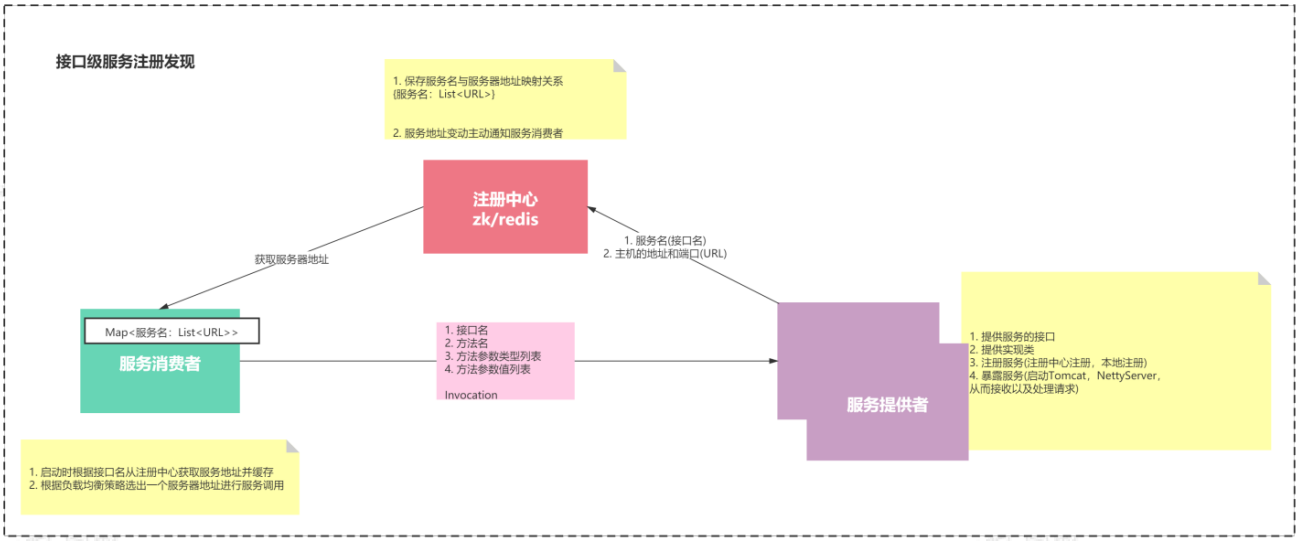
同时如果某个方法是private的，那么@Transactional也会失效，因为底层cglib是基于父子类来实现的，子类是不能重载父类的private方法的，所以无法很好的利用代理，也会导致@Transactianal失效

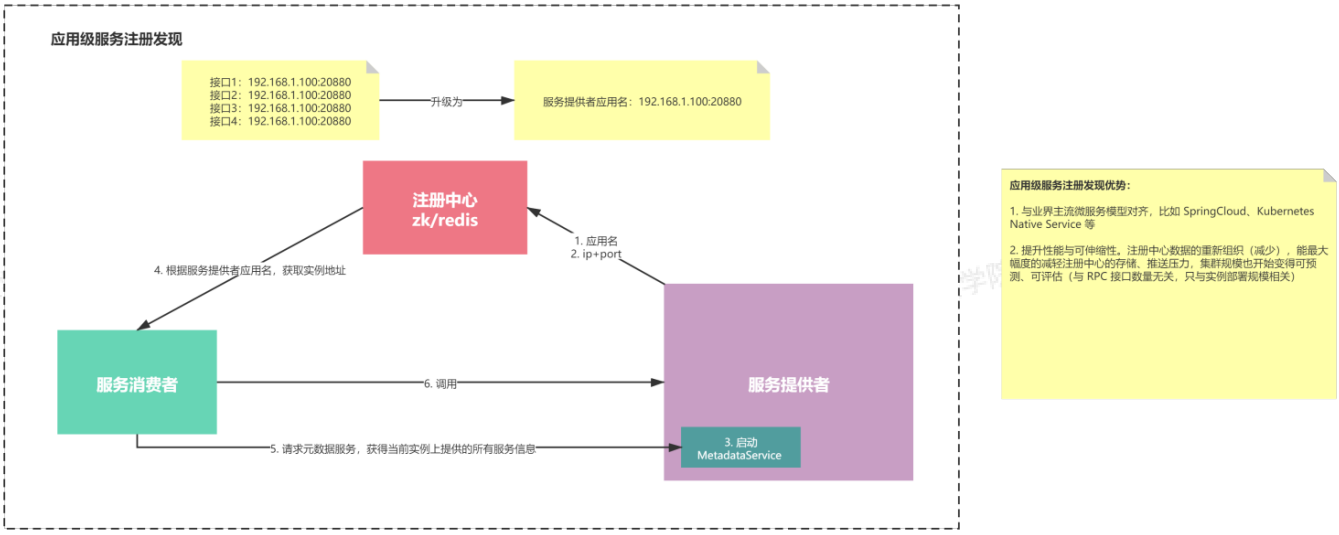
Dubbo是如何做系统交互的

Dubbo底层是通过RPC来完成服务和服务之间的调用的，Dubbo支持很多协议，比如默认的dubbo协议，比如http协议、比如rest等都是支持的，他们的底层所使用的技术是不太一样的，比如dubbo协议底层使用的是netty，也可以使用mina，http协议底层使用的tomcat或jetty。

服务消费者在调用某个服务时，会将当前所调用的服务接口信息、当前方法信息、执行方法所传入的入参信息等组装为一个Invocation对象，然后不同的协议通过不同的数据组织方式和传输方式将这个对象传送给服务提供者，提供者接收到这个对象后，找到对应的服务实现，利用反射执行对应的方法，得到方法结果后再通过网络响应给服务消费者。

当然，Dubbo在这个调用过程中还做很多其他的设计，比如服务容错、负载均衡、Filter机制、动态路由机制等等，让Dubbo能处理更多企业中的需求。





Dubbo的负载均衡策略

Dubbo目前支持：

1. 平衡加权轮询算法
2. 加权随机算法
3. 一致性哈希算法
4. 最小活跃数算法

https://www.yuque.com/renyong-jmovm/kb/gwu187

还读过哪些框架源码介绍一下你还熟悉的

这个问题比较广泛，你即可以说：HashMap、线程池等JDK自带的源码，也可以说Mybatis、Spring Boot、Spring Cloud、消息队列等开发框架或中间件的源码

阿里二面

Jdk1.7到Jdk1.8 HashMap 发生了什么变化(底层)?

1. 1.7中底层是数组+链表，1.8中底层是数组+链表+红黑树，加红黑树的目的是提高HashMap插入和查询整体效率
2. 1.7中链表插入使用的是头插法，1.8中链表插入使用的是尾插法，因为1.8中插入key和value时需要判断链表元素个数，所以需要遍历链表统计链表元素个数，所以正好就直接使用尾插法
3. 1.7中哈希算法比较复杂，存在各种右移与异或运算，1.8中进行了简化，因为复杂的哈希算法的目的就是提高散列性，来提供HashMap的整体效率，而1.8中新增了红黑树，所以可以适当的简化哈希算法，节省CPU资源

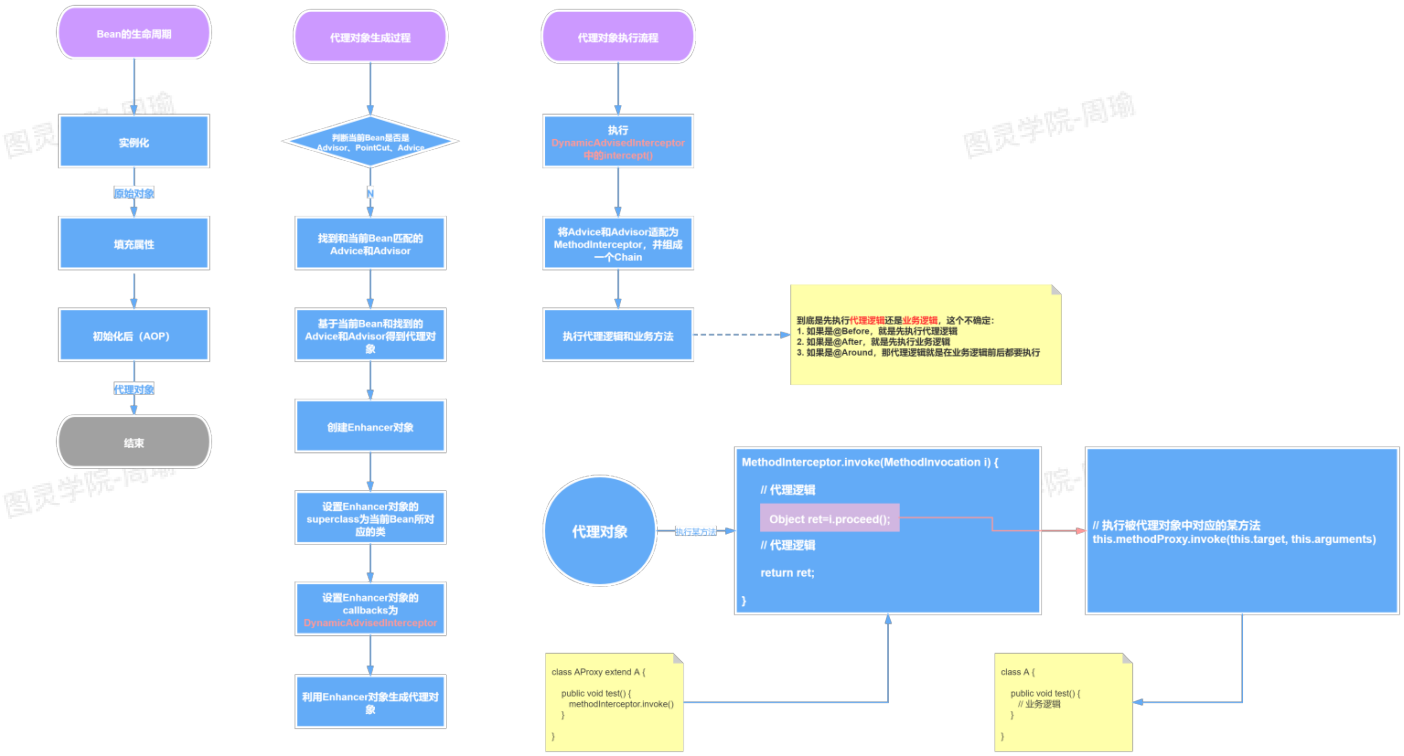
Jdk1.7到Jdk1.8 java虚拟机发生了什么变化?

1.7中存在永久代，1.8中没有永久代，替换它的是元空间，元空间所占的内存不是在虚拟机内部，而是本地内存空间，这么做的原因是，不管是永久代还是元空间，他们都是方法区的具体实现，之所以元空间所占的内存改成本地内存，官方的说法是为了和JRockit统一，不过额外还有一些原因，比如方法区所存储的类信息通常是比较难确定的，所以对于方法区的大小是比较难指定的，太小了容易出现方法区溢出，太大了又会占用了太多虚拟机的内存空间，而转移到本地内存后则不会影响虚拟机所占用的内存

如何实现AOP，项目哪些地方用到了AOP

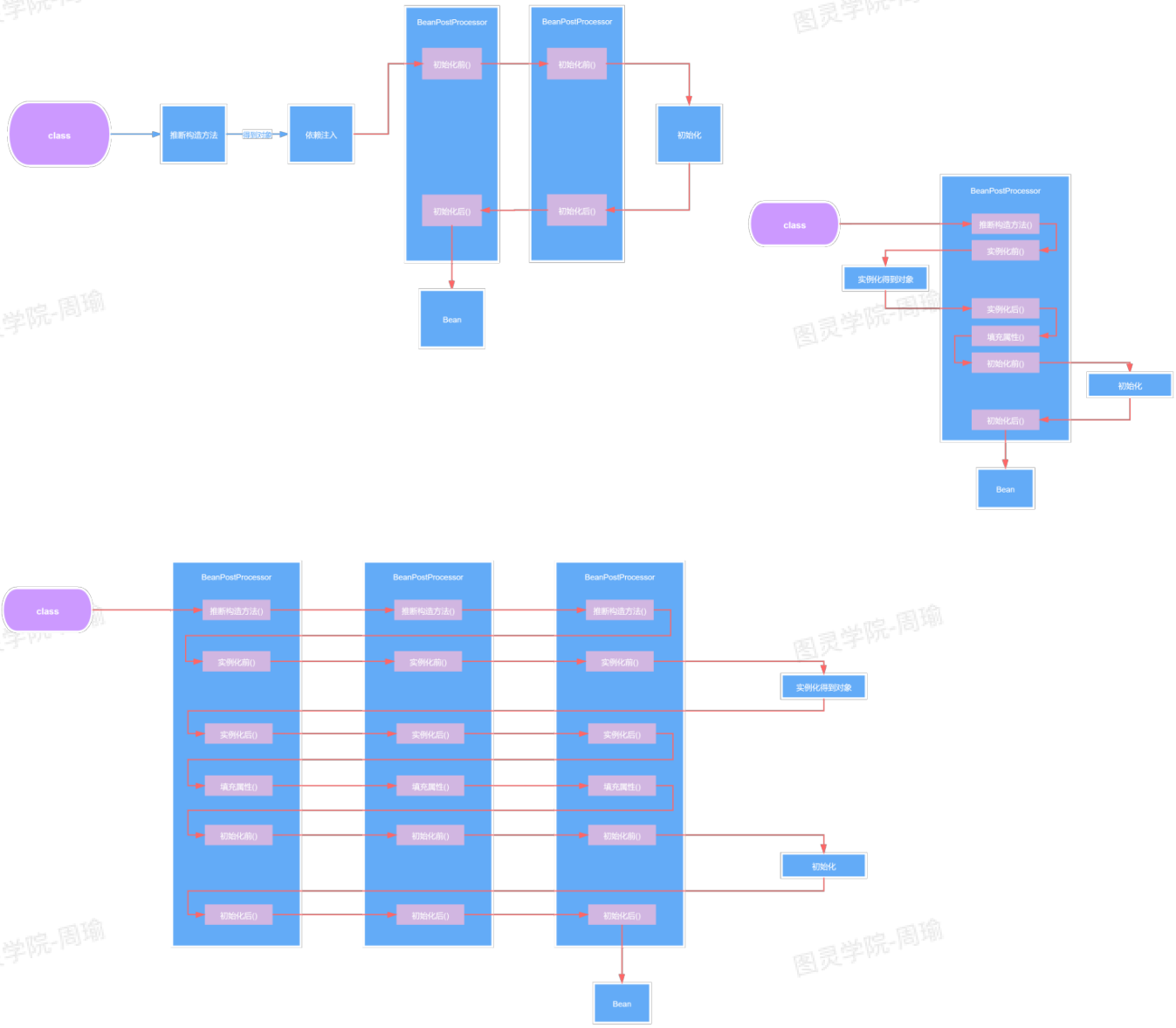
利用动态代理技术来实现AOP，比如JDK动态代理或Cglib动态代理，利用动态代理技术，可以针对某个类生成代理对象，当调用代理对象的某个方法时，可以任意控制该方法的执行，比如可以先打印执行时间，再执行该方法，并且该方法执行完成后，再次打印执行时间。

项目中，比如事务、权限控制、方法执行时长日志都是通过AOP技术来实现的，凡是需要对某些方法做统一处理的都可以用AOP来实现，利用AOP可以做到业务无侵入。



Spring中后置处理器的作用

Spring中的后置处理器分为BeanFactory后置处理器和Bean后置处理器，它们是Spring底层源码架构设计中非常重要的一种机制，同时开发者也可以利用这两种后置处理器来进行扩展。BeanFactory后置处理器表示针对BeanFactory的处理器，Spring启动过程中，会先创建出BeanFactory实例，然后利用BeanFactory处理器来加工BeanFactory，比如Spring的扫描就是基于BeanFactory后置处理器来实现的，而Bean后置处理器也类似，Spring在创建一个Bean的过程中，首先会实例化得到一个对象，然后再利用Bean后置处理器来对该实例对象进行加工，比如我们常说的依赖注入就是基于一个Bean后置处理器来实现的，通过该Bean后置处理器来给实例对象中加了@Autowired注解的属性自动赋值，还比如我们常说的AOP，也是利用一个Bean后置处理器来实现的，基于原实例对象，判断是否需要进行AOP，如果需要，那么就基于原实例对象进行动态代理，生成一个代理对象。



说说常用的SpringBoot注解，及其实现

1. @SpringBootApplication注解：这个注解标识了一个SpringBoot工程，它实际上是另外三个注解的组合，这三个注解是：
   a. @SpringBootConfiguration：这个注解实际就是一个@Configuration，表示启动类也是一个配置类
   b. @EnableAutoConfiguration：向Spring容器中导入了一个Selector，用来加载ClassPath下SpringFactories中所定义的自动配置类，将这些自动加载为配置Bean
   c. @ComponentScan：标识扫描路径，因为默认是没有配置实际扫描路径，所以SpringBoot扫描的路径是启动类所在的当前目录
2. @Bean注解：用来定义Bean，类似于XML中的<bean>标签，Spring在启动时，会对加了@Bean注解的方法进行解析，将方法的名字做为beanName，并通过执行方法得到bean对象
3. @Controller、@Service、@ResponseBody、@Autowired都可以说

说说你了解的分布式锁实现

分布式锁所要解决的问题的本质是：能够对分布在多台机器中的线程对共享资源的互斥访问。在这个原理上可以有很多的实现方式：

1. 基于Mysql，分布式环境中的线程连接同一个数据库，利用数据库中的行锁来达到互斥访问，但是Mysql的加锁和释放锁的性能会比较低，不适合真正的实际生产环境
2. 基于Zookeeper，Zookeeper中的数据是存在内存的，所以相对于Mysql性能上是适合实际环境的，并且基于Zookeeper的顺序节点、临时节点、Watch机制能非常好的来实现的分布式锁
3. 基于Redis，Redis中的数据也是在内存，基于Redis的消费订阅功能、数据超时时间，lua脚本等功能，也能很好的实现的分布式锁

Redis的数据结构及使用场景

Redis的数据结构有：

1. 字符串：可以用来做最简单的数据缓存，可以缓存某个简单的字符串，也可以缓存某个json格式的字符串，Redis分布式锁的实现就利用了这种数据结构，还包括可以实现计数器、Session共享、分布式ID
2. 哈希表：可以用来存储一些key-value对，更适合用来存储对象
3. 列表：Redis的列表通过命令的组合，既可以当做栈，也可以当做队列来使用，可以用来缓存类似微信公众号、微博等消息流数据
4. 集合：和列表类似，也可以存储多个元素，但是不能重复，集合可以进行交集、并集、差集操作，从而可以实现类似，我和某人共同关注的人、朋友圈点赞等功能
5. 有序集合：集合是无序的，有序集合可以设置顺序，可以用来实现排行榜功能

Redis集群策略

Redis提供了三种集群策略：

1. 主从模式：这种模式比较简单，主库可以读写，并且会和从库进行数据同步，这种模式下，客户端直接连主库或某个从库，但是但主库或从库宕机后，客户端需要手动修改IP，另外，这种模式也比较难进行扩容，整个集群所能存储的数据受到某台机器的内存容量，所以不可能支持特大数据量
2. 哨兵模式：这种模式在主从的基础上新增了哨兵节点，但主库节点宕机后，哨兵会发现主库节点宕机，然后在从库中选择一个库作为进的主库，另外哨兵也可以做集群，从而可以保证但某一个哨兵节点宕机后，还有其他哨兵节点可以继续工作，这种模式可以比较好的保证Redis集群的高可用，但是仍然不能很好的解决Redis的容量上限问题。
3. Cluster模式：Cluster模式是用得比较多的模式，它支持多主多从，这种模式会按照key进行槽位的分配，可以使得不同的key分散到不同的主节点上，利用这种模式可以使得整个集群支持更大的数据容量，同时每个主节点可以拥有自己的多个从节点，如果该主节点宕机，会从它的从节点中选举一个新的主节点。

对于这三种模式，如果Redis要存的数据量不大，可以选择哨兵模式，如果Redis要存的数据量大，并且需要持续的扩容，那么选择Cluster模式。

Mysql数据库中，什么情况下设置了索引但无法使用?

1. 没有符合最左前缀原则
2. 字段进行了隐式数据类型转化
3. 走索引没有全表扫描效率高

https://www.bilibili.com/video/BV1W64y1u761?from=search&seid=6062298215110905390

Innodb是如何实现事务的

Innodb通过Buffer Pool，LogBuffer，Redo Log，Undo Log来实现事务，以一个update语句为例：

1. Innodb在收到一个update语句后，会先根据条件找到数据所在的页，并将该页缓存在Buffer Pool中
2. 执行update语句，修改Buffer Pool中的数据，也就是内存中的数据
3. 针对update语句生成一个RedoLog对象，并存入LogBuffer中
4. 针对update语句生成undolog日志，用于事务回滚
5. 如果事务提交，那么则把RedoLog对象进行持久化，后续还有其他机制将Buffer Pool中所修改的数据页持久化到磁盘中
6. 如果事务回滚，则利用undolog日志进行回滚

聊聊你最有成就感的项目

1. 项目是做什么的
2. 用了什么技术
3. 你在项目中担任的职位
4. 收获了什么

自己最有挑战的项目、难点

1. 使用什么技术解决了什么项目难点
2. 使用什么技术优化了什么项目功能
3. 使用什么技术节省了多少成本

京东一面

遇到过哪些设计模式?

在学习一些框架或中间件的底层源码的时候遇到过一些设计模式：

1. 代理模式：Mybatis中用到JDK动态代理来生成Mapper的代理对象，在执行代理对象的方法时会去执行SQL，Spring中AOP、包括@Configuration注解的底层实现也都用到了代理模式

2. 责任链模式：Tomcat中的Pipeline实现，以及Dubbo中的Filter机制都使用了责任链模式
3. 工厂模式：Spring中的BeanFactory就是一种工厂模式的实现
4. 适配器模式：Spring中的Bean销毁的生命周期中用到了适配器模式，用来适配各种Bean销毁逻辑的执行方式
5. 外观模式：Tomcat中的Request和RequestFacade之间体现的就是外观模式
6. 模板方法模式：Spring中的refresh方法中就提供了给子类继承重写的方法，就用到了模板方法模式

Java死锁如何避免?

造成死锁的几个原因：

1. 一个资源每次只能被一个线程使用
2. 一个线程在阻塞等待某个资源时，不释放已占有资源
3. 一个线程已经获得的资源，在未使用完之前，不能被强行剥夺
4. 若干线程形成头尾相接的循环等待资源关系

这是造成死锁必须要达到的4个条件，如果要避免死锁，只需要不满足其中某一个条件即可。而其中前3个条件是作为锁要符合的条件，所以要避免死锁就需要打破第4个条件，不出现循环等待锁的关系。

在开发过程中：

1. 要注意加锁顺序，保证每个线程按同样的顺序进行加锁
2. 要注意加锁时限，可以针对所设置一个超时时间
3. 要注意死锁检查，这是一种预防机制，确保在第一时间发现死锁并进行解决

深拷贝和浅拷贝

深拷贝和浅拷贝就是指对象的拷贝，一个对象中存在两种类型的属性，一种是基本数据类型，一种是实例对象的引用。

1. 浅拷贝是指，只会拷贝基本数据类型的值，以及实例对象的引用地址，并不会复制一份引用地址所指向的对象，也就是浅拷贝出来的对象，内部的类属性指向的是同一个对象
2. 深拷贝是指，既会拷贝基本数据类型的值，也会针对实例对象的引用地址所指向的对象进行复制，深拷贝出来的对象，内部的属性指向的不是同一个对象

如果你提交任务时，线程池队列已满，这时会发生什么

1. 如果使用的无界队列，那么可以继续提交任务时没关系的
2. 如果使用的有界队列，提交任务时，如果队列满了，如果核心线程数没有达到上限，那么则增加线程，如果线程数已经达到了最大值，则使用拒绝策略进行拒绝

谈谈ConcurrentHashMap的扩容机制

1.7版本

1. 1.7版本的ConcurrentHashMap是基于Segment分段实现的

2. 每个Segment相对于一个小型的HashMap
3. 每个Segment内部会进行扩容，和HashMap的扩容逻辑类似
4. 先生成新的数组，然后转移元素到新数组中
5. 扩容的判断也是每个Segment内部单独判断的，判断是否超过阈值

1.8版本

1. 1.8版本的ConcurrentHashMap不再基于Segment实现
2. 当某个线程进行put时，如果发现ConcurrentHashMap正在进行扩容那么该线程一起进行扩容
3. 如果某个线程put时，发现没有正在进行扩容，则将key-value添加到ConcurrentHashMap中，然后判断是否超过阈值，超过了则进行扩容
4. ConcurrentHashMap是支持多个线程同时扩容的
5. 扩容之前也先生成一个新的数组
6. 在转移元素时，先将原数组分组，将每组分给不同的线程来进行元素的转移，每个线程负责一组或多组的元素转移工作

Spring中Bean是线程安全的吗?

Spring本身并没有针对Bean做线程安全的处理，所以：

1. 如果Bean是无状态的，那么Bean则是线程安全的
2. 如果Bean是有状态的，那么Bean则不是线程安全的

另外，Bean是不是线程安全，跟Bean的作用域没有关系，Bean的作用域只是表示Bean的生命周期范围，对于任何生命周期的Bean都是一个对象，这个对象是不是线程安全的，还是得看这个Bean对象本身。

说说你常用的Linux基本操作命令

1. 增删查改
2. 防火墙相关
3. ssh/scp
4. 软件下载、解压、安装
5. 修改权限

Maven中Package和Install的区别

1. Package是打包，打成Jar或War
2. Install表示将Jar或War安装到本地仓库中

项目及主要负责的模块

平时要多了解一下你目前在做的项目中的核心模块，核心功能的业务与使用到的技术

SpringCloud各组件功能，与Dubbo的区别

1. Eureka：注册中心，用来进行服务的自动注册和发现
2. Ribbon：负载均衡组件，用来在消费者调用服务时进行负载均衡
3. Feign：基于接口的申明式的服务调用客户端，让调用变得更简单
4. Hystrix：断路器，负责服务容错
5. Zuul：服务网关，可以进行服务路由、服务降级、负载均衡等
6. Nacos：分布式配置中心以及注册中心
7. Sentinel：服务的熔断降级，包括限流
8. Seata：分布式事务
9. Spring Cloud Config：分布式配置中心
10. Spring Cloud Bus：消息总线
11. ...

https://www.bilibili.com/video/BV1uy4y1W7tt?from=search&seid=94459244624220142

Spring Cloud是一个微服务框架，提供了微服务领域中的很多功能组件，Dubbo一开始是一个RPC调用框架，核心是解决服务调用间的问题，Spring Cloud是一个大而全的框架，Dubbo则更侧重于服务调用，所以Dubbo所提供的功能没有Spring Cloud全面，但是Dubbo的服务调用性能比Spring Cloud高，不过Spring Cloud和Dubbo并不是对立的，是可以结合起来一起使用的。

京东二面

说说类加载器双亲委派模型

JVM中存在三个默认的类加载器：

1. BootstrapClassLoader
2. ExtClassLoader
3. AppClassLoader

AppClassLoader的父加载器是ExtClassLoader，ExtClassLoader的父加载器是BootstrapClassLoader。

JVM在加载一个类时，会调用AppClassLoader的loadClass方法来加载这个类，不过在这个方法中，会先使用ExtClassLoader的loadClass方法来加载类，同样ExtClassLoader的loadClass方法中会先使用BootstrapClassLoader来加载类，如果BootstrapClassLoader加载到了就直接成功，如果BootstrapClassLoader没有加载到，那么ExtClassLoader就会自己尝试加载该类，如果没有加载到，那么则会由AppClassLoader来加载这个类。

所以，双亲委派指得是，JVM在加载类时，会委派给Ext和Bootstrap进行加载，如果没加载到才由自己进行加载。

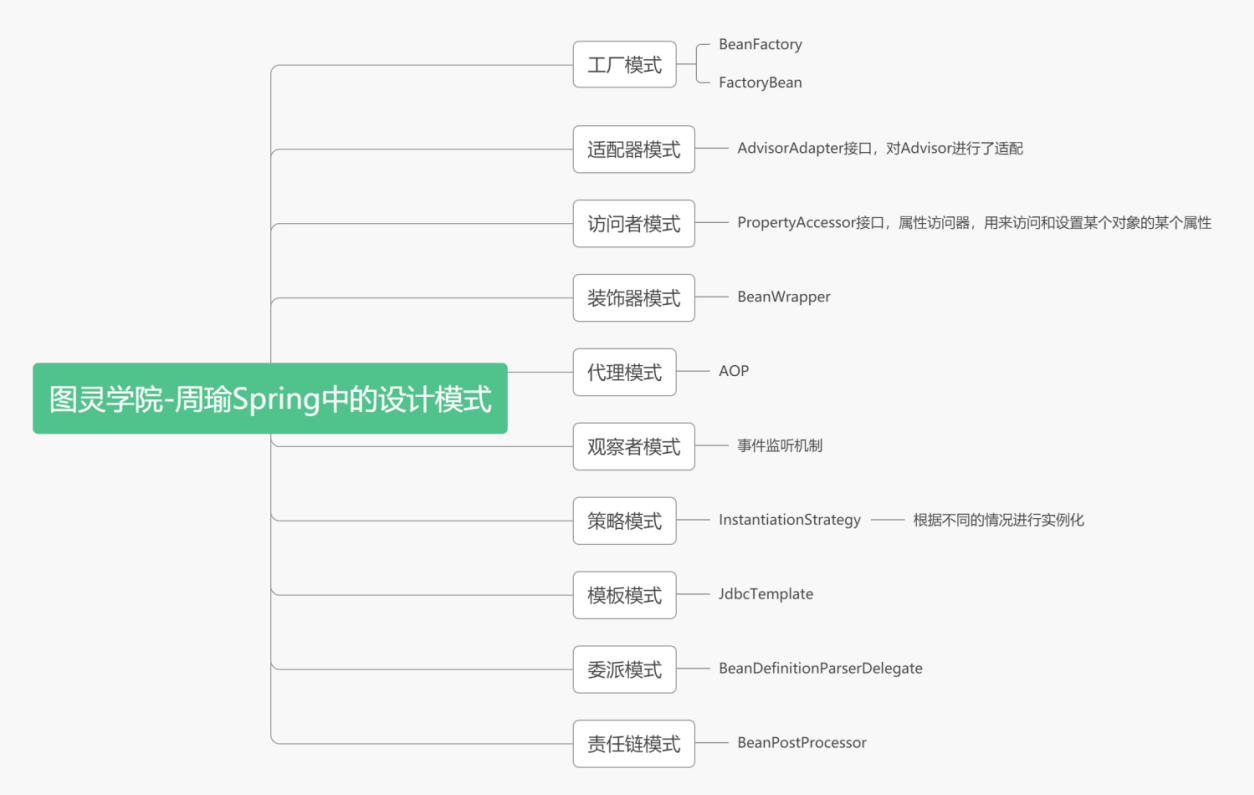
泛型中extends和super的区别

1. <? extends T>表示包括T在内的任何T的子类
2. <? super T>表示包括T在内的任何T的父类

并发编程三要素?

1. 原子性：不可分割的操作，多个步骤要保证同时成功或同时失败
2. 有序性：程序执行的顺序和代码的顺序保持一致
3. 可用性：一个线程对共享变量的修改，另一个线程能立马看到

Spring用到了哪些设计模式



简述CAP理论

CAP理论是分布式领域非常重要的一个理论，很多分布式中间件在实现时都需要遵守这个理论，其中：

1. C表示一致性：指的的是分布式系统中的数据的一致性
2. A表示可用性：表示分布式系统是否正常可用
3. P表示分区容器性：表示分布式系统出现网络问题时的容错性

CAP理论是指，在分布式系统中不能同时保证C和A，也就是说在分布式系统中要么保证CP，要么保证AP，也就是一致性和可用性只能取其一，如果想要数据的一致性，那么就需要损失系统的可用性，如果需要系统高可用，那么就要损失系统的数据一致性，特指强一致性。

CAP理论太过严格，在实际生产环境中更多的是使用BASE理论，BASE理论是指分布式系统不需要保证数据的强一致，只要做到最终一致，也不需要保证一直可用，保证基本可用即可。

图的深度遍历和广度遍历

1. 图的深度优先遍历是指，从一个节点出发，一直沿着边向下深入去找节点，如果找不到了则返回上一层找其他节点
2. 图的广度优先遍历只是，从一个节点出发，向下先把第一层的节点遍历完，再去遍历第二层的节点，直到遍历到最后一层

快排算法

快速排序算法底层采用了分治法。
基本思想是：

1. 先取出数列中的第一个数作为基准数
2. 将数列中比基准数大的数全部放在它的右边，比基准数小的数全部放在它的左边
3. 然后在对左右两部分重复第二步，直到各区间只有一个数

Java版算法实现

```
public class QuickSort {
    public static void quickSort(int[] arr,int low,int high){
        int i,j,temp,t;
        if(low>high){
            return;
        }
        i=low;
        j=high;
        //temp就是基准位
        temp = arr[low];

        while (i<j) {
            //先看右边，依次往左递减
            while (temp<=arr[j]&&i<j) {
```

```
                j--;
            }
            //再看左边，依次往右递增
            while (temp>=arr[i]&&i<j) {
                i++;
            }
            //如果满足条件则交换
            if (i<j) {
                t = arr[j];
                arr[j] = arr[i];
                arr[i] = t;
            }

        }
        //最后将基准为与i和j相等位置的数字交换
         arr[low] = arr[i];
         arr[i] = temp;
        //递归调用左半数组
        quickSort(arr, low, j-1);
        //递归调用右半数组
        quickSort(arr, j+1, high);
    }


    public static void main(String[] args){
        int[] arr = {10,7,2,4,7,62,3,4,2,1,8,9,19};
        quickSort(arr, 0, arr.length-1);
        for (int i = 0; i < arr.length; i++) {
            System.out.println(arr[i]);
        }
    }
```

TCP的三次握手和四次挥手

TCP协议是7层网络协议中的传输层协议，负责数据的可靠传输。

在建立TCP连接时，需要通过三次握手来建立，过程是：

1. 客户端向服务端发送一个SYN
2. 服务端接收到SYN后，给客户端发送一个SYN_ACK

3. 客户端接收到SYN_ACK后，再给服务端发送一个ACK

在断开TCP连接时，需要通过四次挥手来断开，过程是：

1. 客户端向服务端发送FIN
2. 服务端接收FIN后，向客户端发送ACK，表示我接收到了断开连接的请求，客户端你可以不发数据了，不过服务端这边可能还有数据正在处理
3. 服务端处理完所有数据后，向客户端发送FIN，表示服务端现在可以断开连接
4. 客户端收到服务端的FIN，向服务端发送ACK，表示客户端也会断开连接了

消息队列如何保证消息可靠传输

消息可靠传输代表了两层意思，既不能多也不能少。

1. 为了保证消息不多，也就是消息不能重复，也就是生产者不能重复生产消息，或者消费者不能重复消费消息
    a. 首先要确保消息不多发，这个不常出现，也比较难控制，因为如果出现了多发，很大的原因是生产者自己的原因，如果要避免出现问题，就需要在消费端做控制
    b. 要避免不重复消费，最保险的机制就是消费者实现幂等性，保证就算重复消费，也不会有问题，通过幂等性，也能解决生产者重复发送消息的问题
2. 消息不能少，意思就是消息不能丢失，生产者发送的消息，消费者一定要能消费到，对于这个问题，就要考虑两个方面
    a. 生产者发送消息时，要确认broker确实收到并持久化了这条消息，比如RabbitMQ的confirm机制，Kafka的ack机制都可以保证生产者能正确的将消息发送给broker
    b. broker要等待消费者真正确认消费到了消息时才删除掉消息，这里通常就是消费端ack机制，消费者接收到一条消息后，如果确认没问题了，就可以给broker发送一个ack，broker接收到ack后才会删除消息

画出项目架构图，介绍自己所处的模块

需要大家工作中积极的去了解项目架构

蚂蚁一面

二叉搜索树和平衡二叉树有什么关系?

平衡二叉树也叫做平衡二叉搜索树，是二叉搜索树的升级版，二叉搜索树是指节点左边的所有节点都比该节点小，节点右边的节点都比该节点大，而平衡二叉搜索树是在二叉搜索的基础上还规定了节点左右两边的子树高度差的绝对值不能超过1

强平衡二叉树和弱平衡二叉树有什么区别

强平衡二叉树AVL树，弱平衡二叉树就是我们说的红黑树。

1. AVL树比红黑树对于平衡的程度更加严格，在相同节点的情况下，AVL树的高度低于红黑树
2. 红黑树中增加了一个节点颜色的概念
3. AVL树的旋转操作比红黑树的旋转操作更耗时

B树和B+树的区别，为什么Mysql使用B+树

B树的特点：

1. 节点排序
2. 一个节点了可以存多个元素，多个元素也排序了

B+树的特点：

1. 拥有B树的特点
2. 叶子节点之间有指针
3. 非叶子节点上的元素在叶子节点上都冗余了，也就是叶子节点中存储了所有的元素，并且排好顺序

Mysql索引使用的是B+树，因为索引是用来加快查询的，而B+树通过对数据进行排序所以是可以提高查询速度的，然后通过一个节点中可以存储多个元素，从而可以使得B+树的高度不会太高，在Mysql中一个Innodb页就是一个B+树节点，一个Innodb页默认16kb，所以一般情况下一颗两层的B+树可以存2000万行左右的数据，然后通过利用B+树叶子节点存储了所有数据并且进行了排序，并且叶子节点之间有指针，可以很好的支持全表扫描，范围查找等SQL语句。

epoll和poll的区别

1. select模型，使用的是数组来存储Socket连接文件描述符，容量是固定的，需要通过轮询来判断是否发生了IO事件
2. poll模型，使用的是链表来存储Socket连接文件描述符，容量是不固定的，同样需要通过轮询来判断是否发生了IO事件
3. epoll模型，epoll和poll是完全不同的，epoll是一种事件通知模型，当发生了IO事件时，应用程序才进行IO操作，不需要像poll模型那样主动去轮询

简述线程池原理，FixedThreadPool用的阻塞队列是什么

线程池内部是通过队列+线程实现的，当我们利用线程池执行任务时：

1. 如果此时线程池中的数量小于corePoolSize，即使线程池中的线程都处于空闲状态，也要创建新的线程来处理被添加的任务。
2. 如果此时线程池中的数量等于corePoolSize，但是缓冲队列workQueue未满，那么任务被放入缓冲队列。
3. 如果此时线程池中的数量大于等于corePoolSize，缓冲队列workQueue满，并且线程池中的数量小于maximumPoolSize，建新的线程来处理被添加的任务。

4. 如果此时线程池中的数量大于corePoolSize，缓冲队列workQueue满，并且线程池中的数量等于maximumPoolSize，那么通过 handler所指定的策略来处理此任务。
5. 当线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数

FixedThreadPool代表定长线程池，底层用的LinkedBlockingQueue，表示无界的阻塞队列。

sychronized和ReentrantLock的区别

1. sychronized是一个关键字，ReentrantLock是一个类
2. sychronized会自动的加锁与释放锁，ReentrantLock需要程序员手动加锁与释放锁
3. sychronized的底层是JVM层面的锁，ReentrantLock是API层面的锁
4. sychronized是非公平锁，ReentrantLock可以选择公平锁或非公平锁
5. sychronized锁的是对象，锁信息保存在对象头中，ReentrantLock通过代码中int类型的state标识来标识锁的状态
6. sychronized底层有一个锁升级的过程

sychronized的自旋锁、偏向锁、轻量级锁、重量级锁，分别介绍和联系

1. 偏向锁：在锁对象的对象头中记录一下当前获取到该锁的线程ID，该线程下次如果又来获取该锁就可以直接获取到了
2. 轻量级锁：由偏向锁升级而来，当一个线程获取到锁后，此时这把锁是偏向锁，此时如果有第二个线程来竞争锁，偏向锁就会升级为轻量级锁，之所以叫轻量级锁，是为了和重量级锁区分开来，轻量级锁底层是通过自旋来实现的，并不会阻塞线程
3. 如果自旋次数过多仍然没有获取到锁，则会升级为重量级锁，重量级锁会导致线程阻塞
4. 自旋锁：自旋锁就是线程在获取锁的过程中，不会去阻塞线程，也就无所谓唤醒线程，阻塞和唤醒这两个步骤都是需要操作系统去进行的，比较消耗时间，自旋锁是线程通过CAS获取预期的一个标记，如果没有获取到，则继续循环获取，如果获取到了则表示获取到了锁，这个过程线程一直在运行中，相对而言没有使用太多的操作系统资源，比较轻量。

HTTPS是如何保证安全传输的

https通过使用对称加密、非对称加密、数字证书等方式来保证数据的安全传输。

1. 客户端向服务端发送数据之前，需要先建立TCP连接，所以需要先建立TCP连接，建立完TCP连接后，服务端会先给客户端发送公钥，客户端拿到公钥后就可以用来加密数据了，服务端到时候接收到数据就可以用私钥解密数据，这种就是通过非对称加密来传输数据
2. 不过非对称加密比对称加密要慢，所以不能直接使用非对称加密来传输请求数据，所以可以通过非对称加密的方式来传输对称加密的秘钥，之后就可以使用对称加密来传输请求数据了

3. 但是仅仅通过非对称加密+对称加密还不足以能保证数据传输的绝对安全，因为服务端向客户端发送公钥时，可能会被截取
4. 所以为了安全的传输公钥，需要用到数字证书，数字证书是具有公信力、大家都认可的，服务端向客户端发送公钥时，可以把公钥和服务端相关信息通过Hash算法生成消息摘要，再通过数字证书提供的私钥对消息摘要进行加密生成数字签名，在把没进行Hash算法之前的信息和数字签名一起形成数字证书，最后把数字证书发送给客户端，客户端收到数字证书后，就会通过数字证书提供的公钥来解密数字证书，从而得到非对称加密要用到的公钥。
5. 在这个过程中，就算有中间人拦截到服务端发出来的数字证书，虽然它可以解密得到非对称加密要使用的公钥，但是中间人是办法伪造数字证书发给客户端的，因为客户端上内嵌的数字证书是全球具有公信力的，某个网站如果要支持https，都是需要申请数字证书的私钥的，中间人如果要生成能被客户端解析的数字证书，也是要申请私钥的，所以是比较安全了。

蚂蚁二面

设计模式有哪些大类，及熟悉其中哪些设计模式

设计模式分为三大类：

1. 创建型
    a. 工厂模式（Factory Pattern）
    b. 抽象工厂模式（Abstract Factory Pattern）
    c. 单例模式（Singleton Pattern）
    d. 建造者模式（Builder Pattern）
    e. 原型模式（Prototype Pattern）
2. 结构型
    a. 适配器模式（Adapter Pattern）
    b. 桥接模式（Bridge Pattern）
    c. 过滤器模式（Filter、Criteria Pattern）
    d. 组合模式（Composite Pattern）
    e. 装饰器模式（Decorator Pattern）
    f. 外观模式（Facade Pattern）
    g. 享元模式（Flyweight Pattern）
    h. 代理模式（Proxy Pattern）
3. 行为型
    a. 责任链模式（Chain of Responsibility Pattern）
    b. 命令模式（Command Pattern）
    c. 解释器模式（Interpreter Pattern）
    d. 迭代器模式（Iterator Pattern）
    e. 中介者模式（Mediator Pattern）
    f. 备忘录模式（Memento Pattern）

g. 观察者模式（Observer Pattern）
h. 状态模式（State Pattern）
i. 空对象模式（Null Object Pattern）
j. 策略模式（Strategy Pattern）
k. 模板模式（Template Pattern）
l. 访问者模式（Visitor Pattern）

volatile关键字，他是如何保证可见性，有序性

1. 对于加了volatile关键字的成员变量，在对这个变量进行修改时，会直接将CPU高级缓存中的数据写回到主内存，对这个变量的读取也会直接从主内存中读取，从而保证了可见性
2. 在对volatile修饰的成员变量进行读写时，会插入内存屏障，而内存屏障可以达到禁止重排序的效果，从而可以保证有序性

Java的内存结构，堆分为哪几部分，默认年龄多大进入老年代

1. 年轻代
   a. Eden区（8）
   b. From Survivor区（1）
   c. To Survivor区（1）
2. 老年代

默认对象的年龄达到15后，就会进入老年代

Mysql的锁你了解哪些

按锁粒度分类：
1. 行锁：锁某行数据，锁粒度最小，并发度高
2. 表锁：锁整张表，锁粒度最大，并发度低
3. 间隙锁：锁的是一个区间

还可以分为：
1. 共享锁：也就是读锁，一个事务给某行数据加了读锁，其他事务也可以读，但是不能写
2. 排它锁：也就是写锁，一个事务给某行数据加了写锁，其他事务不能读，也不能写

还可以分为：
1. 乐观锁：并不会真正的去锁某行记录，而是通过一个版本号来实现的
2. 悲观锁：上面所的行锁、表锁等都是悲观锁

在事务的隔离级别实现中，就需要利用所来解决幻读

ConcurrentHashMap 如何保证线程安全，jdk1.8 有什么变化

讲一下OOM以及遇到这种情况怎么处理的，是否使用过日志分析工具

Mysql索引原理

介绍一下亮点的项目

项目的并发大概有多高，Redis的瓶颈是多少

项目中遇到线上问题怎么处理的，说一下印象最深刻的