需要背诵

1、方法区

2、抽象方法

3、创建对象时构造器的调用顺序是

4、接口和抽象类

5、Serializable与Externalizable的区别

6、final

7、类加载过程

8、java对象创建过程

9、GC Roots 有哪些

10、逃逸分析

11、zk选举机制

12、zk的监听器原理

13、zk集群中个服务器之间是怎样通信的

14、redis实现延时队列

15、磁盘IO操作

16、B+树和B树的对比

17、为什么InnoDB使用B+树而不是B树

18、spring为何要使用三级缓存来解决循环依赖

19、hash冲突

20、线程调度

21、JDK动态代理

22、CGLIB动态代理

23、数据库事务特性：原子性、一致性、隔离性、持久性

24、spring的事件

25、hash容量为2的指数：h%n=h&(n-1)

26、实现高质量的 equals()诀窍包括

27、springboot的动态配置

28、@Autowire、@Value、@PostConstruct等注解在什么时候被处理

30、创建对象时构造器的调用顺序

31、JDBC操作数据库步骤

32、Serializable与Externalizable的区别

33、final用法

34、API网关作用

35、虚拟机栈：局部变量表、操作数栈、动态链接、方法出口

36、方法区：类信息、常量、静态变量、即时编译器编译后的代码等数据

37、常量池、运行时常量池、串池

38、垃圾回收器

39、CMS过程

40、语法糖

41、类加载

42、对象创建

43、对象结构

44、逃逸分析

45、DispatcherServlet过程

46、bean生命周期理解

47、MyBatis的工作流程

48、@Autowired和@Resource区别

49、MyISAM和InnoDB的区别

50、MVCC（多版本并发控制）

51、MQ场景

52、MQ选择

53、QPS、TFS

54、SPI

55、ReentrantLock原理

56、CHAR和VARCHAR

57、怎样保证幂等性

58、时间轮

59、AQS 使用双向链表

60、ReentrantLock 的实现原理

61、怎么解决mysql+redis的数据一致性问题

1、方法区

类信息、常量、静态变量、即时编译器编译后的代码等数据

2、抽象方法

1）不可是静态，抽象方法需要被重写，静态方法不可以重写

2）不可是native，native是C实现的，抽象方法是没有实现的

3）不可被synchronized修饰

3、创建对象时构造器的调用顺序是

1）先初始化静态成员

2）然后调父类构造器

3）再初始化非静态成员

4）最后调自身构造器

4、接口和抽象类

1）接口可以继承接口，且支持多重继承

2）抽象类可以实现(implements)接口

3）抽象类可继承具体类也可以继承抽象类

5、Serializable与Externalizable的区别

1）Serializable与Externalizable的区别

2）Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制

6、final

1）被final修饰的方法，JVM会尝试将其内联（将方法体纳入编译范围内），以提高运行效率

2）被final修饰的常量，在编译阶段会存入常量池中

7、类加载过程

1）加载（产物为.class对象）

2）连接过程：验证、准备、解析

3）初始化

8、java对象创建过程

1）类加载检查

2）内存分配：指针碰撞（Serial，ParNew）、空闲列表（CMS）

3）初始化默认值

4）设置对象头

9、GC Roots 有哪些

1）在虚拟机栈（栈帧中的本地变量表）中引用的对象

2）在方法区中类静态属性引用的对象，譬如Java类的引用类型静态变量

3）在方法区中常量引用的对象，譬如字符串常量池的引用

4）在本地方法栈中JNI（即通常所说的Native方法法）引用的对象

5）Java虚拟机内部的引用，如基本数据类型对应的Class对象，一些常驻的异常对象（如NullPointExcepiton、OutOfMemoryError）等，还有系统类加载器

6）被同步锁（synchronized关键字）持有的对象

7）反映Java虚拟机内部情况的JMXBean、JVMTI中注册的回调、本地代码缓存等

10、逃逸分析

同步锁消除、标量替换（对象被拆分为若干标量）、栈上分配

11、zk选举机制

非第一次选举：

a）Epoch大的直接胜出

b）Epoch相同，事务id大的胜出

c）Epoch相同，事务id相同，服务器ID大的胜出

12、zk的监听器原理

1）main线程中创建zk客户端，这时会创建两个线程，一个负责网络连接通讯（connet），一个负责监听（listener）

2）通过connect线程将注册的监听事件发送给zk

3）在zk的注册监听事件列表中将注册的监听事件添加到注册监听器列表中

4）zk监听到有数据或者路径的变化，就会将这个消息发送给listener线程

5）Listener线程内部调用process()方法

13、zk集群中个服务器之间是怎样通信的

leader为 每 个Follower/Observer 都创建一个叫做 LearnerHandler 的实体

1）LearnerHandler 主要负责 Leader 和 Follower/Observer 之间的网络通讯，包括数据同步，请求转发和 proposal 提议的投票等。

2）Leader 服务器保存了所有 Follower/Observer 的 LearnerHandler

14、redis实现延时队列

使用sortedset，拿时间戳作为 score，消息内容作为 key 调用 zadd 来生产消息，消费者用 zrangebyscore 指令获取 N 秒之前的数据轮询进行处理

15、磁盘IO操作

磁盘用磁头来读写存储在盘片表面的位，而磁头连接到一个移动臂上，移动臂沿着盘片半径前后移动，可以将磁头定位到任何磁道上，这称之为寻道操作。一旦定位到磁道后，盘片转动，磁道上的每个位经过磁头时，读写磁头就可以感知到该位的值，也可以修改值。对磁盘的访问时间分为 寻道时间，旋转时间，以及传送时间。

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，因此为了提高效率，要尽量减少磁盘 I/O，减少读写操作。 为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字 节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的 局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此预读可以提高I/O效率。

页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（1024个字节或其整数倍），预读的长度一般为页的整倍数。主存和磁盘以页

为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位 置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

文件系统的设计者利用了磁盘预读原理，将一个结点的大小设为等于一个页（1024个字节或其整数倍），这样每 个结点只需要一次I/O就可以完全载入。那么3层的B树可以容纳1024*1024*1024差不多10亿个数据，如果换成二 叉查找树，则需要30层！假定操作系统一次读取一个节点，并且根节点保留在内存中，那么B树在10亿个数据中查 找目标值，只需要小于3次硬盘读取就可以找到目标值，但红黑树需要小于30次，因此B树大大提高了IO的操作效率。

16、B+树和B树的对比

B+ 树的优点在于：

- 由于B+树在非叶子结点上不包含真正的数据，只当做索引使用，因此在内存相同的情况下，能够存放更多的 key。
- B+树的叶子结点都是相连的，因此对整棵树的遍历只需要一次线性遍历叶子结点即可。而且由于数据顺序 排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。

B树的优点在于：

- 由于B树的每一个节点都包含key和value，因此我们根据key查找value时，只需要找到key所在的位置，就能找到value，但B+树只有叶子结点存储数据，索引每一次查找，都必须一次一次，一直找到树的最大深度处，也就是叶子结点的深度，才能找到value。

17、为什么InnoDB使用B+树而不是B树

- 出于对IO性能的考虑
- B树每个节点都存储数据，而B+树只有叶子节点才存储数据，所以在查询相同数据量的情况下，B树的IO会更频繁。因为索引本身存储在磁盘上，当数据量大时，就不能把整个索引全部加载到内存，只能逐一加载每一个磁盘页。更何况B树的索引中还保存了数据信息，导致B树的一个磁盘页保存的索引数量也比较少。即加载索引阶段还加载了许多用不到的数据。
- **遍历效率更高**：由于B+树的数据存储在叶子节点上，分支节点均为索引，方便扫库，只需要扫描一遍叶子即可，而且叶子节点形成链表，范围查询也比较方便。但B树在分支节点都保存着数据，要找到具体的顺序数据，就需要执行一次中序遍历来查询。
- **因为B树不管叶子节点还非叶子节点，都会保存数据，这样导致了非叶子节点中能保存的指针数量就变少，指针少的情况下还要保存大量数据，就只能增加树的高度，导致IO操作变多，查询性能变低**

18、spring为何要使用三级缓存来解决循环依赖

如果 Spring 选择二级缓存来解决循环依赖的话，那么就意味着所有 Bean 都需要在实例化完成之后就立马为其创建代理，而 Spring 的设计原则是在 Bean 初始化完成之后才为其创建代理。所以，Spring 选择了三级缓存。但是因为循环依赖的出现，导致了 Spring 不得不提前去创建代理，因为如果不提前创建代理对象，那么注入的就是原始对象，这样就会产生错误。

19、hash冲突

1）开发地址法：一直往下找一个不冲突的地址

2）再散列法：多个hash函数

3）拉链法：使用链表hashMap

4）公共溢出区

20、线程调度

1）上下文切换是计算密集型，需要一定CPU时间

2）线程初始化会为其分为堆栈空间，一般为512k或者1MB

3）线程过多导致引用很多对象，影响JVM的垃圾回收

21、JDK动态代理

1）拦截器实现InvocationHandler接口，重写invoke方法，method.invoke(被代理类对象, args)

2）Proxy.newProxyInstance

22、CGLIB动态代理

1）拦截器实现MethodInterceptor接口，重写intercept方法，methodProxy.invokeSuper(o, objects);

2）Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(impl.class);

enhancer.setCallback(拦截器对象);

impl proxy = (impl) enhancer.create();

proxy.method();

23、数据库事务特性：原子性、一致性、隔离性、持久性

JMM特性：原子性、可见性、有序性

24、spring的事件

1）事件继承ApplicationEvent

2）监听者实现ApplicationListener<事件>，或者使用@EventListener

3）调用applicationContext.publishEvent(event)

25、hash容量为2的指数：h%n=h&(n-1)

26、实现高质量的 equals()诀窍包括

1）if(this == o) {}

2）if(o == null) {}

3）if(getClass() != o.getClass()) {}

4）if (o instanceof Person) {

Person oPerson = (Person) o;

//5）最后判断属性是否相等

}

27、springboot的动态配置

BeanFactoryPostProcessor->BeanDefinitionRegistryPostProcessor->ConfigurationClassPostProcessor->springboot自动装配

28、@Autowire、@Value、@PostConstruct等注解在什么时候被处理

1）BeanPostProcessor->AutowiredAnnotationBeanPostProcessor->处理@Autowire/@Value

2）BeanPostProcessor->InitDestoryAnnotationBeanPostProcessor->处理@PostConstruct

29、BeanFactory和FactoryBean的区别

1）BeanFactory即bean工厂，提供了一套标准化的bean生命周期的流程

2）FactoryBean有三个方法，getObjectType()、isSingleton()、getObject()，留给用户自定义bean（可以代理，可以new等等）

30、创建对象时构造器的调用顺序

实例化静态成员–>调用父类构造器–>初始化非静态成员–>调用自身构造器

31、JDBC操作数据库步骤

加载驱动–>创建连接–>创建语句–>执行语句–>处理结果–>关闭资源（Result–>Statement–>Connection）

32、Serializable与Externalizable的区别

Externalizable允许指定特定的二进制格式

33、final用法

1）类不可以被集成、方法不可以被重写、变量不可以被改变

2）修饰的方法，JVM会尝试内联（将方法体纳入编译范围）

3）修饰的常量，编译期会进入常量池

34、API网关作用

授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等

35、虚拟机栈：局部变量表、操作数栈、动态链接、方法出口

36、方法区：类信息、常量、静态变量、即时编译器编译后的代码等数据

37、常量池、运行时常量池、串池

1）常量池：Class文件一部分，虚拟机指令根据这张常量表找到要执行的类名，方法名，参数类型、字面量等信息

2）运行时常量池：方法区的一部分，当该类被加载后，常量池信息就会放入运行时常量池，并把符号地址变为真实内存地址。

3）串池：存放字符串对象且里面的元素不重复

38、垃圾回收器

Serial　　ParNew（Serial多线程版）　Parallel Scavenge（吞吐量优先）

Serial Old　CMS（最短停顿时间）　　Parallnel old

39、CMS过程

初始标记STW->并发标记（用户线程可并发执行）->重新标记STW（并发标记的脏数据）->并发清除

40、语法糖

默认构造器、自动拆装箱、泛型擦除、可变参数、foreach循环、switch字符串(hashCode)、switch枚举（序号）、匿名内部类

41、类加载

加载->链接（验证、准备[类变量分配内存,常量编译期确定]、解析[符号引用替换为直接引用]）->初始化

42、对象创建

类加载->内存分配（指针碰撞、空闲列表）->初始化默认值->初始化对象头->初始化方法

43、对象结构

对象头、实例数据、对齐填充

44、逃逸分析

锁消除、标量替换、栈上分配

45、DispatcherServlet过程

1）前端控制器DisDispatcherServlet捕获请求

2）根据URI调用HandlerMapping获得Handler对象和Handler对应的拦截器，以HandlerExecutionChain对象形式返回

3）根据获得的handler选择合适的handlerAdapter，执行拦截器的preHandler()的方法

4）提取Request模型数据，填充handler入参，执行handler(Controller)，填充入参过程中，spring会做以下工作

a）HttpMessageConveter，将请求消息（json、xml）转化为对象

b）数据转换和格式化，String转interger等，或字符串格式化为日期

c）数据验证：长度、格式等，验证结果存储在BindingResult或者Error中

5）handler(Controller)完成后，向DisDispatcherServlet返回ModelAndView对象

6）根据ModelAndView对象选择ViewResolver返回给DisDispatcherServlet

7）ViewResolver渲染视图

8）视图负责将渲染结果返回给客户端

46、bean生命周期理解

1）使用beanDefubutionReader从配置或者xml中读取bean的定义信息

2）bean的定义信息经过一系列BeanFactoryPostProcessor，可以通过BeanFactoryPostProcessor来修改bean的定义信息（BeanDefinitionRegistryPostProcessor）

3）使用createBeaninstance实例化bean（分配内存）

4）属性赋值，使用populateBean给自定义属性赋值，使用invokeAwareMethods给容器对象赋值，当然也可以使用@Autowire注入

5）前置处理器：处理BeanPostProcessor的postProcessBeforeInitialization()

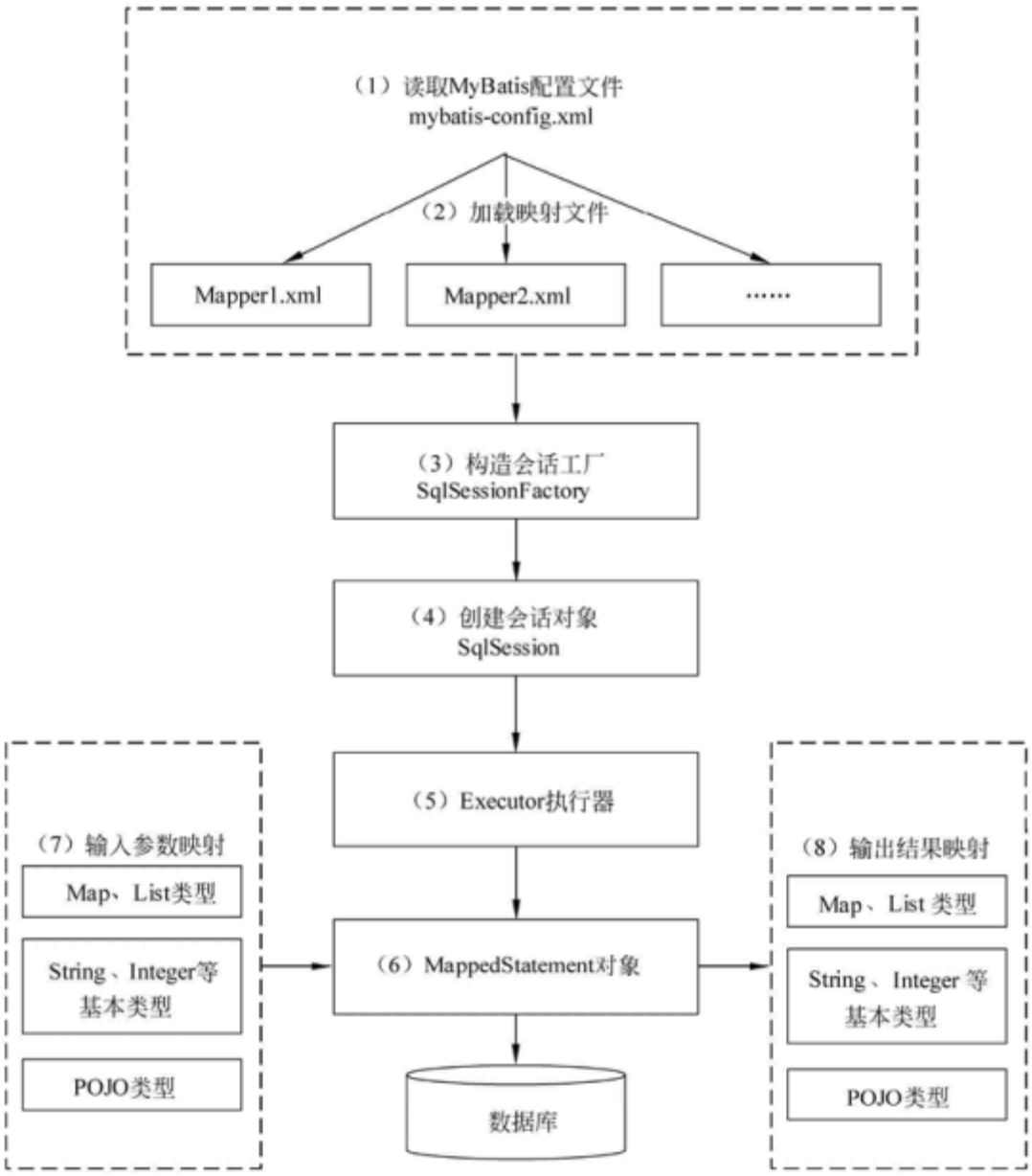
6）初始化bean，执行invokeInitMethods，并且判断是否实现了initializingBean接口，使用afterPropertiesSet给用户最后一次修改属性的机会，最后执行invokeInitMethod方法

7）后置处理器处理BeanPostProcessor的postProcessAfterInitialization()

8）完整bean对象

9）bean销毁（容器关闭的时候）

47、MyBatis的工作流程



1）读取 MyBatis 配置文件：mybatis-config.xml 为 MyBatis 的全局配置文件，配置了 MyBatis 的运行环境等信息，例如数据库连接信息。

2）加载映射文件。映射文件即 SQL 映射文件，该文件中配置了操作数据库的 SQL 语句，需要在 MyBatis 配置文件 mybatis-config.xml 中加载。mybatis-config.xml 文件可以加载多个映射文件，每个文件对应数据库中的一张表。

3）构造会话工厂：通过 MyBatis 的环境等配置信息构建会话工厂 SqlSessionFactory。

4）创建会话对象：由会话工厂创建 SqlSession 对象，该对象中包含了执行 SQL 语句的所有方法。

5）Executor 执行器：MyBatis 底层定义了一个 Executor 接口来操作数据库，它将根据 SqlSession 传递的参数动态地生成需要执行的 SQL 语句，同时负责查询缓存的维护。

6）MappedStatement 对象：在 Executor 接口的执行方法中有一个 MappedStatement 类型的参数，该参数是对映射信息的封装，用于存储要映射的 SQL 语句的 id、参数等信息。

7）输入参数映射：输入参数类型可以是 Map、List 等集合类型，也可以是基本数据类型和 POJO 类型。输入参数映射过程类似于 JDBC 对 preparedStatement 对象设置参数的过程。

8）输出结果映射：输出结果类型可以是 Map、 List 等集合类型，也可以是基本数据类型和 POJO 类型。输出结果映射过程类似于 JDBC 对结果集的解析过程。

48、@Autowired和@Resource区别

1）@Autowired支持使用@Primary决定装载顺序

2）@Autowired默认按照类型装配，可以试用@Qualifier按照名字装配

3）@Resource默认按照类型装配

49、MyISAM和InnoDB的区别

1）MyISAM磁盘上存储成三个文件（表定义、存储数据、索引文件），InnoDB存储两个文件（表结构、数据和索引为一个文件）

2）MyISAM叶子节点存储的是数据的地址，InnoDB叶子节点存储的是整行数据

3）MyISAM占用内存少

4）InnoDB支持事务

5）MyISAM是表所，InnoDB支持行锁

6）MyISAM不支持外键，InnoDB支持

50、MVCC（多版本并发控制）

MVCC相当于是为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。它是通过数据库记录中的隐式字段Undo日志、Read View来实现的。

解决的问题：

1）在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，从而提高数据库的

并发读写的处理能力。

2）能实现读一致性，从而解决脏读、幻读、不可重复读等不可重复读，但是不能解决数据更新丢失的问题。

3）采用乐观锁或者悲观锁用来解决写和写的冲突，从而最大程度地去提高数据库的并发性能。

4、springboot的自动装配理解？？

51、MQ场景

流量削峰、应用解耦、异步处理

52、MQ选择

1）数据量大、吞吐量要求比较高的场景一般采用Kafka；

2）对消息可靠性要求很高，甚至要求支持事务的场景，比如金融互联网，可以选择RocketMQ；

3）对于中小型公司来说，可以选择RabbitMQ，它利用erlang 语言本身的并发优势，性能好 在微秒级

53、QPS、TFS

QPS (每秒查询率)

例:
假如我们一天有10万pv(访问量),
公式 (100000 * 80%) / (86400*20%) = 4.62 QPS(峰值时间的每秒请求)

公式原理：每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间。

那我们还可以转一下公式算出我们需要的机器数量

** 机器：峰值时间的每秒请求 / 单台的QPS = 机器数量 **

TPS (吞吐量)

这个很好理解，简单来说就是在单位时间能处理的数量,我们都知道简单浏览器过程就是一个请求和响应的过程, 一般来说，在我们无并发的情况下,吞吐量还是响应时间的倒计时. 相反在我们的并发应用下我们这个就成为我们的机器的标准。

并发量

并发我们都听的很多，但是他还有个哥哥叫并行。

并发：一段时间访问的大量用户的请求

并行：同一时刻的大量用户的请求

并发是最能体现你的代码和机器的性能。

QPS每秒查询率

每秒查询率QPS是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

每秒查询率

因特网上，经常用每秒查询率来衡量域名系统服务器的机器的性能，其即为QPS。

对应fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

54、SPI

SPI(Service Provider Interface)是JDK内置的一种服务提供发现机制，可以用来启用框架扩展和替换组件,主要用于框架中开发

1）Java内置的SPI通过java.util.ServiceLoader类解析classPath和jar包的META-INF/services/目录 下的以接口全限定名命名的文件，并加载该文件中指定的接口实现类

2）Java SPI是一个服务提供接口对应一个配置文件，配置文件中存放当前接口的所有实现类，多个服务提供接口对应多个配置文件，所有配置都在services目录下；

Spring factories SPI是一个spring.factories配置文件存放多个接口及对应的实现类，以接口全限定名作为key，实现类作为value来配置，多个实现类用逗号隔开，仅resource/META-INF/spring.factories一个配置文件。

55、ReentrantLock原理

1）锁的竞争，ReentrantLock 是通过互斥变量，使用 CAS 机制来实现的。没有竞争到锁的线程，使用了 AbstractQueuedSynchronizer 这样一个队列同步器来存储。当锁被释放之后，会从 AQS 队列里面的头部唤醒下一个等待锁的线程

2）公平和非公平的特性，主要是体现在竞争锁的时候，是否需要判断 AQS 队列存在等待中的线程

3）锁的重入特性，在 AQS 里面有一个成员变量来保存当前获得锁的线程，当同一个线程下次再来竞争锁的时候，就不会去走锁竞争的逻辑，而是直接增加重入次数

56、CHAR和VARCHAR

CHAR存储固定长度字符串，不够的使用空格补充，删除时会自动删除多余的空格

VARCHAR存储变长的字符串

57、怎样保证幂等性

产生原因：

1）用户的重复提交或者用户的恶意攻击；

2）分布式系统中，为了避免数据丢失，采用的超时重试机制

解决办法：

1）使用数据库的唯一约束来实现幂等

2）使用 Redis 提供的 setNX 指令

3）使用状态机来实现幂等，所谓的状态机是指一条数据的完整运行状态的转换流程，比如 ，因为它的状态只会向前变更，所以多次修改同一条数据的时候，一旦状态发生变更，那么对这条数据修改造成的影响只会发生一次

4）基于 Token 机制或者增加去重表等方法

58、时间轮

59、AQS 使用双向链表

1）没有竞争到锁的线程加入到阻塞队列，并且阻塞等待的前提是，当前线程所在节点的前置节点是正常状态，这样设计是为了避免链表中存在异常线程导致无法唤醒后续线程的问题。线程阻塞之前需要判断前置节点的状态，如果没有指针指向前置节点，就需要从 Head 节点开始遍历，性能非常低。

2）处于锁阻塞的线程允许外部线程通过interrupt()方法触发唤醒并中断的，这个时候，被中断的线程的状态会修改成 CANCELLED。而被标记为 CANCELLED 状态的线程，是不需要去竞争锁的，但是它仍然存在于双向链表里面。后续的锁竞争中，需要把这个节点从链表里面移除，否则会导致锁阻塞的线程无法被正常唤醒。

3）为了避免线程阻塞和唤醒的开销，所以刚加入到链表的线程，首先会通过自旋的方式尝试去竞争锁。

60、ReentrantLock 的实现原理

第1个，锁的竞争，ReentrantLock 是通过互斥变量，使用 CAS 机制来实现的；

没有竞争到锁的线程，使用了 AbstractQueuedSynchronizer 这样一个队列同步器来存储，底层是通过双向链表来实现的。当锁被释放之后，会从 AQS 队列里面的头部唤醒下一个等待锁的线程。

第2个，公平和非公平的特性，主要是体现在竞争锁的时候，是否需要判断 AQS 队列存在等待中的线程。

第3个，锁的重入特性，在 AQS 里面有一个成员变量来保存当前获得锁的线程，当同一个线程下次再来竞争锁的时候，就不会去走锁竞争的逻辑，而是直接增加重入次数。

61、怎么解决mysql+redis的数据一致性问题

1）先更新mysql，再更新redis：如果更新mysql后，程序异常，造成数据不一致

2）先删除reids，再更新mysql：删除和更新不是原子操作

解决办法（最终一致性）：

基于RocketMQ的可靠性消息通信，来实现最终一致性（更新数据库、更新缓存（失败的请求写入MQ事务））

通过Canal组件，来监控MySQL中Binlog的日志，把更新后的数据同步到Redis中

62、springboot的约定优于配置体现

1）Spring BootStarter 启动依赖，它能帮我们管理所有 jar 包版本；

2）Spring Boot会自动内置Tomcat容器来运行 Web 应用，我们不需要再去单独做应用部署。

3）Spring Boot通过扫描约定路径下的 Spring.factories文件来识别配置类，实现 Bean 的自动装配。

4）Spring Boot会默认加载的配置文件 application.properties 等等。

63、为什么不建议使用过长的字段作为主键?

因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。

64、用非单调的字段作为主键在InnoDB中不是个好主意?

因为InnoDB数据文件本身是一颗B+Tree，主键索引顺序存储在磁盘上，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个

很好的选择。

65、CPU 资源过度消耗

1）阻塞导致上下文切换严重

2）CPU资源过渡消耗。若占用CPU的线程一直是同一个，需要dump线程日志；如果CPU利用率较高的线程不断切换，说明线程创建过多。

66、Seata框架

1）AT 模式，是一种基于本地事务+二阶段协议来实现的最终数据一致性方案，也是Seata 默认的解决方案

2）TCC 模式，TCC 事务是 Try、Confirm、Cancel 三个词语的缩写，简单理解就是把一个完整的业务逻辑拆分成三个阶段，然后通过事务管理器在业务逻辑层面根据每个分支事务的执行情况分别调用该业务的 Confirm 或者Cacel 方法。

3）Saga 模式，Saga 模式是 SEATA 提供的长事务解决方案，在 Saga 模式中，业务流程中每个参与者都提交本地事务，当出现某一个参与者失败则补偿前面已经成功的参与者。

4）XA 模式，XA 可以认为是一种强一致性的事务解决方法，它利用事务资源（数据库、消息服务等）对 XA 协议的支持，以 XA 协议的机制来管理分支事务的一种事务模式。

67、云原生

1）微服务；几乎每个云原生的定义都包含微服务，跟微服务相对的是单体应用，微服务有理论基础，那就是康威定律，指导服务怎么切分。微服务架构能实现服务解耦，内聚更强，变更更易；另一个划分服务的依据就是DDD。

2）容器化；Docker就是应用最为广泛的容器引擎，在大厂的基础设施中大量使用.容器化为微服务提供实施保障，起到应用隔离作用。而K8s 是容器编排系统，用于容器管理，容器间的负载均衡。

3）DevOps；DevOps是一个组合词，Dev + Ops，翻译过来就是开发和运维合体，实际上DevOps应该还包括测试。DevOps为云原生提供持续交付能力。

4）满足持续交付；持续交付又叫做CICD，相当于要实现在线不停机更新，这就要求开发版本和稳定版本并存，需要标准的流程和工具支撑。目前能够提供持续交付的工具也很多，比如Jenkins、Sonar等等。

68、线程池种类

1）CachedThreadPool，是一种可以缓存的线程池。无上限，现成存活60s

2）FiexdThreadPool，是一种固定线程数量的线程池。核心线程和最大线程数量都是一个固定的值。

3）SingleThreadExecutor只有一个工作线程的线程池

4）ScheduledThreadPool，具有延迟执行功能的线程池（可以定时调度）

5）WorkStealingPool，利用工作窃取的算法并行处理请求。

69、HashMap扩容：每次扩容印子*容量大小就扩容，扩容的大小是原来的2倍。

70、CountDownLatch和CyclicBarrier的区别

1）CountDownLatch的计数器只能使用一次。而CyclicBarrier的计数器可以使用reset() 方法重置。

2）CyclicBarrier能处理更为复杂的业务场景，比如计算发生错误，可以结束阻塞，重置计数器，重新执行程序

3）CyclicBarrier提供getNumberWaiting()方法，可以获得CyclicBarrier阻塞的线程数量，还提供isBroken()方法，可以判断阻塞的线程是否被中断，等等。

4）CountDownLatch会阻塞主线程，CyclicBarrier不会阻塞主线程，只会阻塞子线程。

71、死锁条件

死锁案例

线程1外部锁住了A资源，内部锁住了B资源（无法获取B资源）

线程2外部锁住了B资源，内部锁住了A资源（无法获取A资源）

死锁条件

1）互斥条件，共享资源a和b只能被一个线程占用；

2）请求和保持条件，线程T1已经获取共享资源a，在等待共享资源b的时候，不释放共享资源a；

3）不可抢占条件，其他线程不能强行抢占线程T1占有的资源；

4）循环等待条件，线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源，这形成了循环等待

主动释放线程占有的资源（超时释放），按照顺序申请资源防止死锁

72、FactoryBean使用

https://blog.csdn.net/weixin_42195284/article/details/109339203

73、@Transactional时效

1）如果@Transactional事务注解添加在不是public修饰的方法上，Spring的事务就会失效

2）如果事务方法所在的类没有加载到Spring IoC容器中，也就是说，事务方法所在的类没有被Spring管理，从而导致Spring无法实现代理，所以，Spring事务也会失效

3）如果事务方法抛出异常被 catch 处理了，导致 @Transactional 无法回滚而导致事务失效

4）如果同一个类中的两个方法分别为A和B，方法A上没有添加事务注解，方法B上添加了@Transactional事务注解，方法A调用方法B，那么，方法B的事务会失效

5）如果内部方法的事务传播类型为不支持事务的传播类型，那么，内部方法的事务在Spring中会失效

6）如果在@Transactional注解中rollbackFor参数标注了错误的异常类型，那么，Spring事务的回滚就无法识别，导致事务回滚失效

7）没有配置Spring的事务管理器，Spring的事务也不会生效

8）数据库本身不支持事务

74、Spring中实现异步调用的方式有哪些?

1）注解方式。在配置类加上@EnableAsync来启用异步注解，然后使用@Async注解标记需要异步执行的方法。返回值类型必须是java.util.concurrent.Future或其子类（Future、ListenableFuture、AsyncResult、CompletableFuture）

注意：@Async默认会使用SimpleAsyncTaskExecutor来执行，而这个线程池不会复用线程。所以，通常要使用异步处理，我们都会自定义线程池。

2）内置线程池方式。ThreadPoolTaskExecuto最为常用，只要当ThreadPoolTaskExecutor不能满足需求时，可以使用ConcurrentTaskExecutor。

3）自定义线程池方式。通过实现AsyncConfigurer接口或者直接继承AsyncConfigurerSupport类来自定义线程池

75、跨域如何解决

浏览器才不会发送预检请求

1）请求方法是GET 、HEAD其中任意一个

2）请求头中包含Accept、Accept-Language、Content-Language、Content-Type、DPR、Downlink、SaveData、Viewport.Width、Width字段。

3）Content-Type的值是text/plain 、multipart/form-data ,application/x-ww-form-urlencoded 中任意一个

解决跨域

1）spring项目，添加处理跨域的过滤器或者拦截器

2）spring boot项目，再支持跨域的方法上添加@CrossOrigin注解

3）spring boot项目配置类实现WebMvcConfigurer接口来实现跨域支持，重写addCorsMapping() 方法

76、空的java对象占用多少个字节

1）一个Java空对象，在开启压缩指针的情况下，占用12个字节。其中，Markword占8个字节、类元指针占4个字节。但是为了避免伪共享问题，JVM会按照8个字节的倍数进行填充，所以会在对齐填充区填充4个字节，变成16个字节长度。

2）那么在关闭压缩指针的情况下，Object默认会占用16个字节。其中，Markword占8个字节、类元指针占4个字节， 对齐填充占4个字节。16个字节正好是8的整数倍，因此不需要填充。

77、保证SimpleDateFormat线程安全

1）可以把SimpleDateFormat定义成非全局使用的局部变量，这样每个线程调用的时候都创建一个新的实例。

2）可以使用ThreadLocal，把SimpleDateFormat变成一个线程私有的对象。

3）定义SimpleDateFormat的时候，加上同步锁，这样就能够保证在同一时刻只允许一个线程操作

4）使用Java 8的新特性，在Java8中引入了一些线程安全的日期操作API，比如LocalDateTimer、DateTimeFormatter 等等

79、雪花算法：符号位（1Bit）、时间戳（41Bit）、机器码（10Bit）、序列号（12Bit）

1）分布式系统内不会产生ID碰撞，效率高

2）不需要依赖数据库等第三方系统，稳定性更高，可以根据自身业务分配bit位，非常灵活

3）生成ID的性能也非常高，每秒能生成26万个自增可排序的ID

缺点：机器时钟回拨，可能会导致ID重复

79、乐观锁适合于读多写少的情况，悲观锁适合于写多读少的情况（写线程竞争激烈，导致乐观锁重试，CPU占用率高）

80、分库分表场景

1）只分库不分表

当数据库的读写访问量过高，还有可能会出现数据库连接不够用的情况。这个时候我们就需要考虑分库，通过增加数据库实例的方式来获得更多的数据库连接，从而提升系统的并发性能

2）只分表不分库

当单表存储的数据量非常大的情况下，并且并发量也不高，数据库的连接也还够用。但是数据写入和查询的性能出现了瓶颈，这个时候就需要考虑分表了。将数据拆分到多张表中来减少单表存储的数据量，从而提升读写的效率。

3）既分库又分表

结合前面的两种情况，如果同时满足前面的两个条件，也就是数据连接也不够用，并且单表的数据量也很大，从而导致数据库读写速度变慢的情况，这个时候就要考虑既分库又分表。

81、读写分离是用来解决数据库的读性能瓶颈的

大多数互联网公司的业务场景，往往都是读多写少。当访问量过大情况下，数据库查询首先会成为瓶颈，这个时候，如果我们希望能够线性的提升数据库的查询性能，消除读写锁冲突，从而提升数据库的写性能，那么就可以考虑采用读、写分离架构。用一句话概括，读写分离是用来解决数据库的读性能瓶颈的

82、生产环境秒杀接口并发量过大如何处理

限流、缓存、增加服务器节点

83、分布式锁

Redis实现分布式锁

客户端命令：set user nx 10 ex 12

java命令：Boolean lock = redisTemplate.opsForValue.setIfAbsent("lock","10",3,TimeUnit.SECONDS);

分布式锁误解锁问题

https://blog.csdn.net/weixin_43715214/article/details/128213580

线程A：上锁、具体操作、服务器卡顿、锁时间到期自动释放

线程B：抢到锁、具体操作

线程A：反应过来手动释放锁，就把B的锁释放了

解决办法：使用UUID防止误删除

新问题：删除锁没有原子性操作：A判断是自己的锁，准备删除（还未删除却过期自动删除了），B抢到了锁。

解决办法：使用redis+LUA脚本实现判断和删除锁的原子性

传统分布式锁面临的问题

上面基于redis+lua实现的分布式锁可以满足大多数场景下的需求，可是仍然面临一些问题：

- 锁不可重入
- 可重试问题
- 超时释放问题（任务尚未完成，锁已经被释放）
- 主从一致性问题（redis的主从切换导致分布式锁失效）

Redission针对上述问题的解决

- 基于Redis的发布与订阅 以及Semaphore实现了锁的等待，就是没拿到锁我可以等待一段时间。
- 实现了可重入锁，基于Redis的Hash数据结构
- 为了死锁，给锁添加了过期时间
- 为了防止业务没有执行完，锁被释放，利用时间轮添加了续期机制（看门狗机制）
- 因为Redis是属于ap模型，在发生分区容错的时候，优先保证高可用，所以会牺牲一定的数据一致性。为了防止锁丢失，也提供了联锁的概念。

84、逃逸分析

锁消除、标量替换、栈上分配

85、为什么Redis选择使用跳表而不是红黑树来实现有序集合?

Redis 中的有序集合(zset) 支持的操作：

- 插入一个元素
- 删除一个元素
- 查找一个元素
- 有序输出所有元素
- 按照范围区间查找元素（比如查找值在 [100, 356] 之间的数据）

其中，前四个操作红黑树也可以完成，且时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。按照区间查找数据时，跳表可以做到 O(logn) 的时间复杂度定位区间的起点，然后在原始链表中顺序往后遍历就可以了，非常高效。

86、聚簇索引和非聚簇索引

https://blog.csdn.net/weixin_44842613/article/details/117122001

https://m.php.cn/faq/489392.html

https://blog.csdn.net/weixin_39270240/article/details/108571268

https://blog.csdn.net/weixin_43715214/article/details/127080931

一张表里最多**只有一个**主键索引，当然一个主键索引中可以包含多个字段。

为什么建议InnoDB表必须建主键?

在设计者设计InnoDB这种引擎的时候，ibd 文件必须要用一棵B+树来组织。当我们这个表里面有主键的情况下，首先主键会自带主键索引，显然我们就可以用这个主键索引来组织这张表的数据。当我们创建的表中没有主键索引，那么该存储引擎会帮我们挑选出一列不重复的数据，然后用这一列的索引数据来组织这一棵B+树。

但是如果这样的列如果不存在呢？换句话说就是不存在每个元素都不相同的列，怎么办?

MySQL会帮我们建立一列隐藏列，类似于我们oracle中使用的rowid一样，然后将其作为主键索引。

相信看到这里，优缺点大家已经一目了然了！这么简单的事情交给机器来做合适吗？机器万一挑错了怎么办？rowid那么长的一串字符会浪费多少空间？ 所以这种小事情在我们一开始建表的时候就应该自己做好（主键索引在一开始创建表的时候，就要自己指定出来）

为什么推荐使用整型的自增主键？为什么不用UUID?

整型的原因

- 整形比字符串（UUID）省空间
- 整形判断大小比字符串（UUID）效率要高（字符串是比较ASCALL码）

自增的原因

如果我们的主键是自增的，那我们插入下一个节点的时候，这个节点的索引值肯定是要比已存在的所有索引值要大的。我们的存储引擎是B+树的结构，那我们只需要在最右下方的一个数据页中，新增一个节点即可。但是如果我们设计的主键它不是自增的话，那我们插入下一个节点的时候，那这棵B+树很可能需要频繁的做平衡和节点分裂，非常浪费性能！

为什么非主键索引结构叶子节点存储的是主键值?

非主键索引（二级索引）**没有必要**放一整张表的数据，因为主键索引里面已经放了。找到主键索引然后再做一次“**回表**”操作就行了！

- 保证一致性
- 节省存储空间