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

62、springboot的约定优于配置体现

63、为什么不建议使用过长的字段作为主键?

64、用非单调的字段作为主键在InnoDB中不是个好主意?

65、CPU 资源过度消耗

66、Seata框架

67、云原生

68、线程池种类

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

71、死锁条件

72、FactoryBean使用

73、@Transactional失效

74、Spring中实现异步调用的方式有哪些?

75、跨域如何解决

76、空的java对象占用多少个字节

77、保证SimpleDateFormat线程安全

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

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...

80、分库分表场景

81、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

83、分布式锁

84、逃逸分析

85、为什么Redis选择使用跳表而不是红黑树来实现有序集合?

86、聚簇索引和非聚簇索引

87、慢sql排查

88、reentantLock 公平锁和非公平锁的实现区别

89、ThreadLocaL如何防止内存泄漏?

90、对象头理解

91、synchronized

92、ReentrantLock

93、用户态和内核态理解

94、说下跳表跟B+树的区别?

95、Redis和zk实现分布式锁区别:

96、缓存跟DB的一致性怎么解决

97、为什么Redis用头插，而HashMap用的是尾插

98、Redis有没有像HashMap一样的扩容机制

99、项目流量激增，如何处理

100、Redisson的联锁

101、核心线程池满后新任务是新建一个线程还是阻塞到队列中去

102、Mysql里面的RedoLog和BinLog，他们的作用在哪?

103、假如Myql的表很大，进行分页的时候，limit 1000000 加载很慢

104、为什么不建议用uuid作为主键

105、 为什么不推荐使用UUID

106、索引覆盖和索引下推

107、 如果一个查询既满足索引覆盖又满足索引下推的条件，通常优先选择索引覆盖。

108、导致MySQL在之前的版本中没有采用索引下推的原因:

109、synchronized 锁的加锁和解锁是重量级的，主要有以下几个原因:

110、 StampedLock底层原理

111、 在并发情况下，HashMap使用尾插入可能会导致以下问题:

112、 在并发情况下，HashMap使用头插入可能会导致以下问题

113、MySQL索引失效可能发生在以下场景中:

114、死锁条件

115、分库分表三种应用场景

116、CurrentHashMap

117、CPU利用率过高

118、Innodb 的事务实现原理

119、mysql的事务隔离级别

120、mysql怎么保证ACID

121、 stringTable、字符串常量池、常量池的区别

122、 stringTable和字符串常量池有和联系

123、mysql数据什么时候在内存中，什么时候在磁盘中

124、ForkAndJoin

125、 多线程为何不能自动传递上下文

126、 永久代发生内存不足主要是什么引起

127、NIO的缓冲区操作的是直接内存吗

128、HashMap 在 JDK 1.8 有什么改变

129、ThreadLocal原理

130、一致性 Hash 的缺点

131、怎么防止 SQL 注入

132、sql编译过程通常包括以下几个步骤

133、编译和解释有什么区别

134、使用 Mybatis 时，调用 DAO（Mapper）接口时是怎么调用到 SQL 的

135、FactoryBean 的使用场景

136、使用使用@Configuration、@Bean等注解来完成Bean的创建，为何还要使用FactoryBean

137、什么情况下对象不能被代理在Java中，有几种情况下对象不能被代理：

138、 要在 Spring IoC 容器构建完毕之后执行一些逻辑，怎么实现

139、要在 Spring IoC 容器构建完毕之后执行一些逻辑，怎么实现

140、 @Component注解所在的类也需要加入到spring容器中，那为何可以在容器初始化之后做一些事情呢

141、 MySQL 如何锁住一行数据

142、Spring 中的常见扩展点有哪些

143、MySQL 的可重复读是怎么实现的

144、MySQL 是否会出现幻读

145、MySQL 的 gap 锁

146、MySQL 是否会出现幻读

147、MySQL 的 gap 锁

148、分库分表的实现方案

149、explain 中每个字段的意思

150、explain 中的 type 字段有哪些常见的值

151、explain 中你通常关注哪些字段，为什么

152、运行时数据区

153、哪些场景需要打破双亲委派模式

154、线上服务器出现频繁 Full GC，怎么排查

155、定位问题常用哪些命令

156、介绍下 JVM 调优的过程

157、Redis 集群要增加分片，槽的迁移怎么保证无损

158、Redis 的 Hash 对象底层结构

159、Redis 中 Hash 对象的扩容流程

160、Redis 的 Hash 对象的扩容流程在数据量大的时候会有什么问题吗

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()

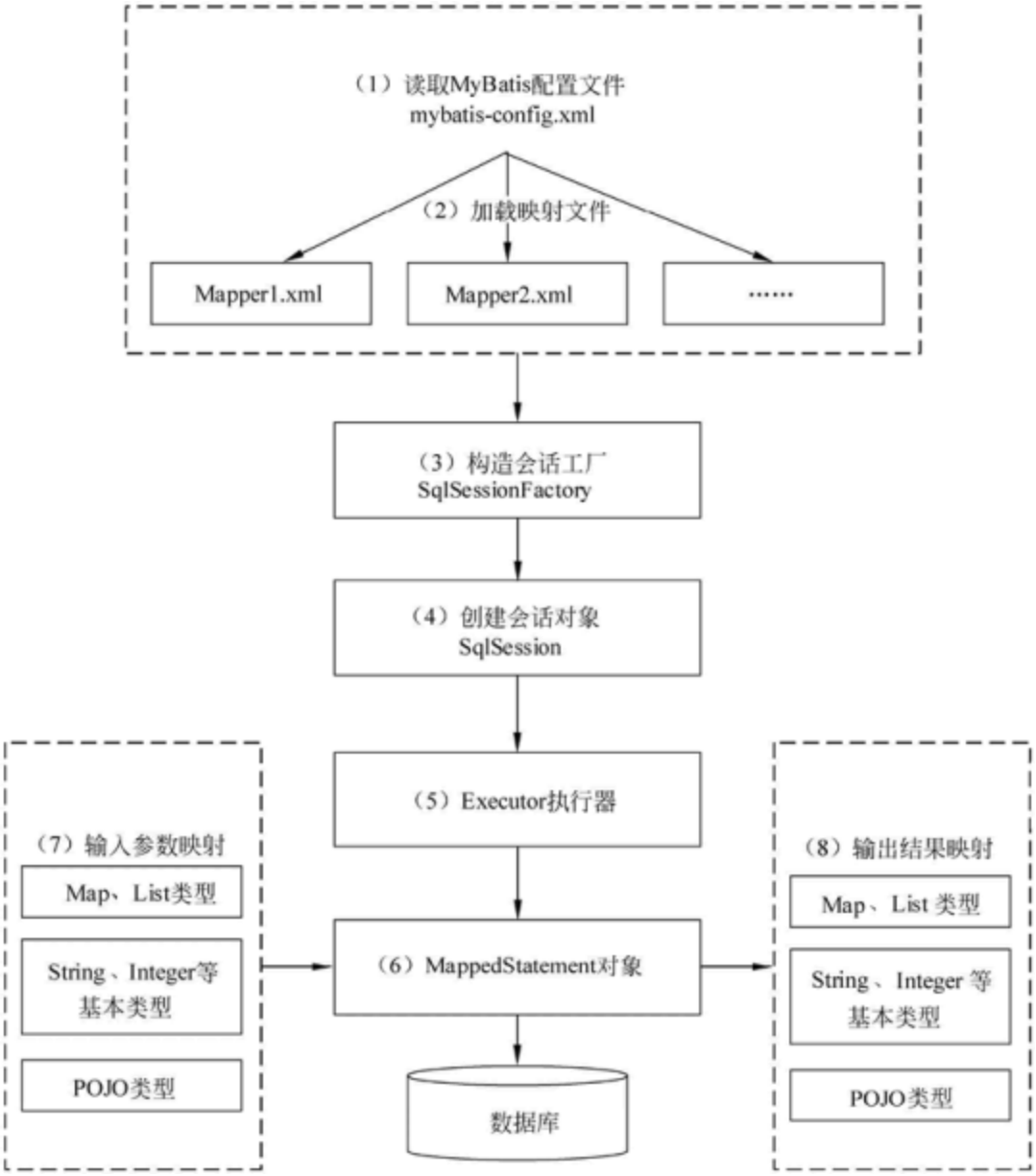
6）初始化bean，执行invokeInitMethods，并且判断是否实现了initializingBean接口，使用afterPropertiesSet给用户最后一次修改属性的机会，最后执行invokeInitMethod方法

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失效

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

87、慢sql排查

要排查MySQL的慢SQL问题，可以按照以下步骤进行：

1. 开启慢查询日志：在MySQL的配置文件中（通常是my.cnf或my.ini），找到slow_query_log参数，并将其设置为1，开启慢查询日志功能。同时，可以设置long_query_time参数来定义超过多少秒的查询才会被记录在慢查询日志中。
2. 查看慢查询日志：等待一段时间（根据long_query_time参数设置），然后使用文本编辑器打开慢查询日志文件，通常位于MySQL的数据目录下。查看其中记录的慢查询语句。
3. 分析慢查询语句：针对慢查询日志中的每个慢查询语句，可以使用MySQL自带的工具如mysqldumpslow或pt-query-digest来进行分析。这些工具可以帮助解析慢查询语句，提取出关键信息，如查询时间、执行计划等。
4. 优化慢查询语句：根据慢查询语句的分析结果，可以尝试优化查询语句的性能。一些常见的优化方法包括添加适当的索引、重写查询语句、调整数据库结构等。可以使用MySQL提供的EXPLAIN语句来查看查询语句的执行计划，进一步了解查询的执行方式和可能的瓶颈。
5. 监控系统资源：除了查询本身的优化，还应该监控MySQL所运行的服务器的系统资源，如CPU、内存、磁盘和网络等。如果系统资源出现瓶颈，可能会导致慢查询问题。可以使用各种系统监控工具来监控系统资源的使用情况，并根据需要进行调整。
6. 定期检查和优化：慢查询问题通常需要持续关注和优化。建议定期检查慢查询日志，查找和解决新出现的慢查询问题。此外，还可以考虑使用MySQL的性能监控工具来实时监控数据库性能，并根据需要进行调整和优化。

通过以上步骤，可以帮助您排查MySQL的慢查询问题，并对查询进行优化，提升数据库的性能和响应速度。

88、reentantLock 公平锁和非公平锁的实现区别

基于aqs实现的一个可重入锁，通过aqs中的state字段来维护是否获得锁或者重入了几次，并且这个字段需要用volatile来修饰，保证线程之间的可见性。如果是不同的线程，通过cas判断是否能拿到锁，如果是同一个线程重复抢占锁，那么state+1，这样就实现了可重入。

1）公平锁会去判断是不是等待队列
的第一个线程或者没有线程在等待，如果是，就能获取锁；如果有线程在等待，必须按照队列的顺序来获取锁。新的线程必须排在等待队列后面。这也是公平的由来，任何线程都必须按照等待队列的顺序来获取锁
2）非公平，只要锁释放，没有加入等待队列的线程能提前通过cas去抢占锁，可能拿到锁的时间节点比等待队列的线程更早

89、ThreadLocaL如何防止内存泄漏?

1）ThreadLocal里面的数据都是以entry的数组保存在我们的thread里的，其中entry的key为threadLocal对象，value是你设置的值。key为弱引用，GC时候回收；

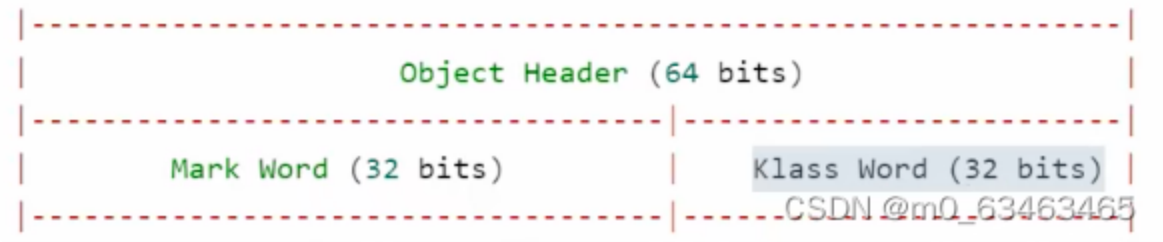
2）key被回收但是entry对象还存在，threadLocal通过各种机制，在使用的时候会去回收key=null的entry对象 ，比如当我去set的时候，会从当前冲突的位置向前或者向后找到第一个null的为止，并且把找到的所有的被GC回收掉的entry清除！！并且rehash，供其他对象使用，达到一个时间与空间的平衡；

3）因为如果没有hash冲突，自身是不能走清除的。所以我们一般使用的使用，用完要remove。

90、对象头理解

在JVM中，每个对象都是由三部分组成的：对象头、实例数据、数据填充。synchronized的锁的信息都是存储在对象头里。

对象头，分别为普通对象和数组对象的头信息



Object Header (96 bits)		
Mark Word(32bits)	Klass Word(32bits)	array length(32bits)

Mark Word结构，分别为32位和64位下的Mark Word

Mark Word (32 bits)				State
hashcode:25	age:4	biased_lock:0	01	Normal
thread:23	epoch:2	age:4	biased_lock:1	01	Biased
ptr_to_lock_record:30			00	Lightweight Locked
ptr_to_heavyweight_monitor:30			10	Heavyweight Locked
			11	Marked for GC

Mark Word (64 bits)					State
unused:25	hashcode:31	unused:1	age:4	biased_lock:0	01	Normal
thread:54	epoch:2	unused:1	age:4	biased_lock:1	01	Biased
ptr_to_lock_record:62					00	Lightweight Locked
ptr_to_heavyweight_monitor:62					10	Heavyweight Locked
					11	Marked for GC

1、Normal状态：此状态为普通状态，hashcode为对象的hashcode值，age代表垃圾回收的分代年龄，biased_lock表示是否为偏向锁，最后两位代表加锁状态。

2、Biased状态：此状态为偏向锁状态，thread指向获得偏向锁的线程，后3位为101表示对象为偏向锁状态。

3、Lightweight Locked状态：轻量级锁状态，ptr_to_lock_record指向栈帧的锁记录。

4、Heavyweight Locked状态：重量级锁，ptr_to_heavyweight_monitor指向Monitor。

注意：

1、当开启偏向锁时(默认开启)，创建一个对象，对象的Mark Word为偏向锁状态，偏向锁是默认延迟的，不会在程序启动时立即生效

2、当禁用偏向锁时，创建的对象为普通状态，即使该对象被synchronized修饰，也不会变为偏向锁状态

3、如果对象调用hashcode方法，会自动禁用偏向锁，是因为偏向锁的对象头中没办法存储hashcode。轻量级锁把Mark Word的值存放在栈帧中，重量级锁把Mark Word的值存放在Monitor中

91、synchronized

对象：对象头（Mark Word、Class Pointer）、实例数据、数据填充

Mark Word在默认情况下存储着对象的哈希码（HashCode）、GC分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等。

偏向锁：hashcode存放线程的ID

轻量级锁：线程内部开辟LockRecord空间存储锁对象，锁对象头存储线程的LockRecord内存地址。失败的线程自旋。

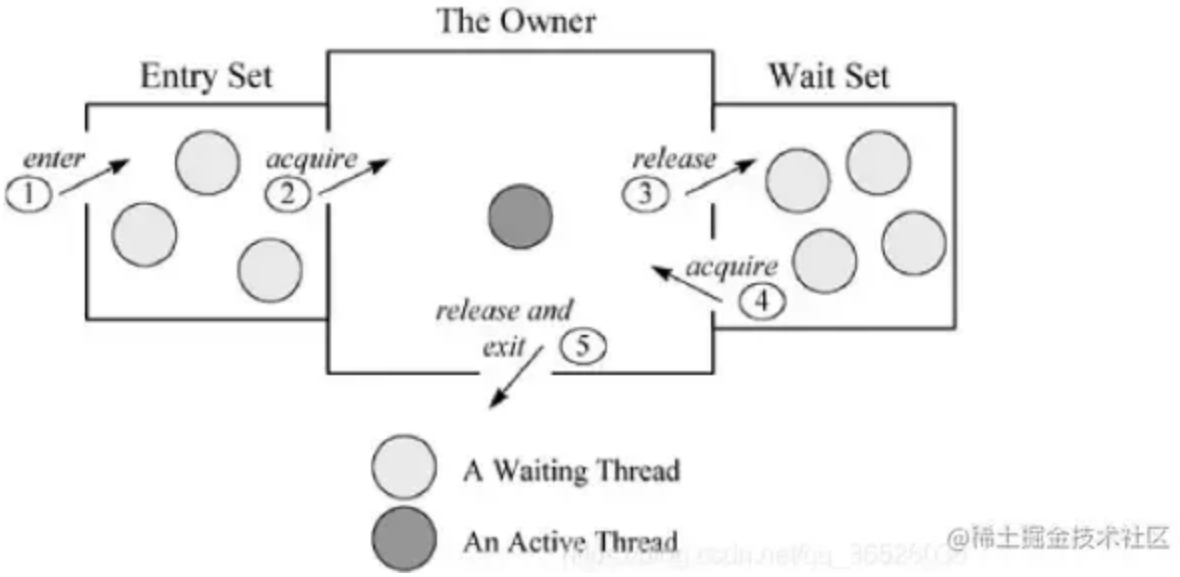
重量级锁：自旋失败升级为重量级锁，线程会由用户态转为内核态，申请互斥变量。

在Java中被synchronized关键字修饰的对象头且为重量级锁时，会关联一个Monitor对象，Monitor有count、Owner、EntryList、WaitSet等字段

- count用来记录线程进入加锁代码的次数
- owner记录当前持有锁的线程，即持有ObjectMonitor对象的线程
- EntryList是想要持有锁的线程的集合
- WaitSet 是加锁对象调用wait（）方法后，等待被唤醒的线程的集合

当多个线程访问同步代码块时：

- 首先线程会进入EntryList集合，然后当线程拿到Monitor对象时，进入owner区域，并把Monitor的owner设置为当前线程，_owner指向持有ObjectMonitor对象的线程，并把计数器count加1
- 若线程调用wait方法，将释放当前持有的monitor对象，同时owner变量恢复为null，count自减1，同时该线程进入WaitSet集合中等待被唤醒
- 当前线程执行完毕，也将释放monitor（锁）并复位count的值，以便其他线程进入获取monitor(锁)



92、ReentrantLock

1）底层是AQS，通过控制state完成一些锁特有的特性：重入、公平与非公平、读写锁

2）获取锁就是当前线程修改了AQS的vloatile成员state

3）获取锁失败就进入AQS的等待队列（FIFO的双向无环链表），进入到等待队列之后开始自旋，当前节点的waitStatus=-1之后LockSupport.park()挂起自己，等待唤醒

4）获取锁的线程释放锁（state进行减操作），唤醒head节点之后第一个未取消的等待节点

5）head节点之后第一个未取消的等待节点被唤醒，判断prev是否为head，是head的话尝试获取锁，将自己设置为head节点，将原来的老head移除等待队列

93、比较Lock和synchronized锁区别

1）原理上：

2）效率上：

3）使用场上：

（1）支持更灵活的同步代码块结构。使用synchronized关键字时，只能在同一个synchronized块结构中获取和释放控制。Lock接口允许实现更复杂的临界区结构（即控制的获取和释放不出现在同一个块结构中）。

（2）相比synchronized关键字，Lock接口提供了更多的功能。其中一个新功能是tryLock()方法的实现。这个方法试图获取锁，如果锁已经被其他线程获取，它将返回false并继续往下执行代码，使用synchronized关键字时，如果线程A试图执行一个同步代码块，而线程B已经在执行这个同步代码块，则线程A就会被挂起直到线程B运行完成这个同步代码块。使用锁的tryLock（）方法，通过返回值将得知是否有其他线程正在使用这个锁保护的代码块。

（3）Lock接口允许分离读和写操作，允许多个读线程和只有一个写线程。（实现ReadWriteLock接口）

（4）相比synchronized关键字，Lock接口具有更好的性能。

93、用户态和内核态理解

用户态和内核态是操作系统中的两个关键概念，用于描述程序在不同特权级别下执行的状态和所能访问的资源范围。下面是对用户态和内核态的简要解释：

1. 用户态（User Mode）：
    - 用户态是指程序在较低的特权级别下执行，只能访问有限的系统资源和执行受限的操作。
    - 在用户态下运行的程序无法直接访问操作系统内核的核心功能和底层硬件资源。
    - 用户态程序执行的是一般的应用逻辑，如运行应用程序、处理用户输入、进行计算等。
    - 用户态的程序运行速度较快，但对系统底层资源的直接控制能力有限。
2. 内核态（Kernel Mode）：
    - 内核态是操作系统的高特权级别，具有对系统资源和底层硬件的完全控制权。
    - 在内核态下运行的代码可以访问和操作操作系统的核心功能和底层硬件资源。
    - 内核态可以执行特权指令、管理内存、进行硬件设备的操作、调度进程等底层操作。
    - 内核态的代码具有更高的特权和更广泛的访问权限，但也需要谨慎操作，以确保系统的稳定性和安全性。

在操作系统中，用户态和内核态的切换是通过系统调用（System Call）来实现的。当用户态程序需要访问内核态的资源或执行需要特权的操作时，它会通过系统调用请求操作系统内核的帮助。操作系统会将进程切换到内核态，执行相应的操作，并将结果返回给用户态程序，然后再切换回用户态继续执行。

用户态和内核态的存在是为了提高系统的安全性和稳定性。通过将用户态和内核态分离，操作系统可以限制用户态程序的访问权限，防止其对系统资源的滥用或破坏。同时，操作系统可以对内核态的代码进行严格的控制和管理，确保系统的正常运行和保护核心数据。

总结来说，用户态和内核态是操作系统中不同特权级别下程序执行的状态。用户态下的程序受限于访问资源和执行操作的权限，而内核态下的程序具有更高的特权和广泛的资源访问权限。通过用户态和内核态的切换，操作系统实现了对系统资源的保护和管理，确保系统的安全性和稳定性。

线程切换为何涉及到用户态和内核态的转化

线程切换涉及到用户态和内核态的切换是因为线程的执行需要依赖操作系统的支持和调度。下面是一些原因解释为何线程切换需要设计到用户态和内核态的切换：

1. 特权级别和资源访问：
    - 内核态具有更高的特权级别，可以访问和操作系统底层的资源，如内存管理、硬件设备等。
    - 用户态只能访问有限的资源，受到操作系统的保护限制，无法直接访问底层资源。
    - 当线程需要访问内核态的资源或执行需要特权的操作时，需要切换到内核态。
2. 系统调用：
    - 系统调用是用户态程序请求操作系统内核提供服务和执行特权操作的方式。
    - 当线程需要执行系统调用，如文件操作、网络通信等，就需要切换到内核态执行相应的内核代码。
    - 内核态可以响应系统调用并执行相应的操作，然后将结果返回给用户态程序。
3. 异常和中断处理：
    - 在程序执行过程中，可能会发生各种异常情况，如空指针异常、除零异常等。
    - 当发生异常或硬件中断时，需要操作系统来处理并采取相应的措施，如异常处理、中断处理等。
    - 线程切换到内核态，操作系统可以捕获和处理异常或中断，确保系统的稳定性和安全性。
4. 调度和并发控制：
    - 操作系统需要进行线程调度和并发控制，以合理地分配系统资源和保证多个线程的公平执行。
    - 在内核态下，操作系统可以执行调度算法、管理线程的状态和优先级等。
    - 线程切换到内核态，可以让操作系统进行合适的调度和资源管理，提高系统的并发性能。

总之，线程切换涉及到用户态和内核态的切换是为了实现对系统资源的访问和管理、响应系统调用、处理异常和中断、以及进行线程调度和并发控制。用户态和内核态的切换是操作系统的一种机制，通过切换特权级别和访问权限，确保系统的正常运行、资源的合理分配和保护，以及多个线程的协调执行。

94、说下跳表跟B+树的区别?

跳表，其实就是一个链表结构。添加数据时都会随机一个层级，然后把每个数据层级相同的节 点以链表的方式进行链接。 查询的时候，从最外层的链表开始查询，思想跟二分法的思想很类似。

B+树，叶子节点存有真实数据，非叶子节点是这些真实节点的目录页，查询的时候，从根目录开始 查询，树的高度越高，查询的次数也就越多。跳表跟B+树都是用空间来换取时间，用额外的空间来保存链表或者目录页，来提升查询性能，

区别：

1）层高，B+树三层就能支持千万级别的数据，但是跳表，存储相同的数据量需要更高的层级，这也 是为什么我们innoDB索引用B+树而不用跳表的原因，InnoDb需要跟磁盘IO，层级越高，IO次数也就 越

多，但是Redis的zset是用的跳表，因为Redis是基于内存操作，也就没有磁盘IO的概念，所以跳 表反而更简单。

2）操作数据，跳表要比B+树快，因为B+树，在数据操作的时候，要去维护B+树，所以会有树的分裂 与合并，但是跳表是随机一个层次，实现相对简单。

跳表主要用在zset的有序数据集合类型。

95、Redis和zk实现分布式锁区别：

1）因为Zk是基于临时有序节点去获取锁，所以每个线程去抢占锁，都会去创建一个临时节点，当抢占锁的线程比较多的时候，对ZK集群的压力会比较大。而Redis则是通过判断key是否存在来获取锁。

2）Redis属于cap里面的ap，优先去保证高可用，所以在某些场景下，比如主从切换，可能导致数据丢失，导致锁丢失，而zk则是使用zap协议优先保证数据一致性，所以安全性方面，Redis要比ZK低，但是也有像联锁的方式去提升可靠性。

3）性能方面，Redis基于内存操作，所以性能比zk要高

96、缓存跟DB的一致性怎么解决

- 延时双删：所谓延时双删，就是在更新DB后，等待一段时间，再进行Redis删除！来等待其他的线程拿到的都是最新数据
- 采用锁机制，不让有并发在更新的时候，采取锁的机制，不让其他线程进行删除操作！但是会拖慢整个性能，违背了Redis的初衷
- 比如基于RocketMQ的可靠性消息通信，来实现最终一致性

97、为什么Redis用头插，而HashMap用的是尾插

1. Redis使用头插法（Head Insertion）： Redis是一种基于内存的键值对存储系统，使用的数据结构是链表。在链表中，每个节点都包含指向前一个节点和后一个节点的指针。当在Redis中插入新元素时，它会采用头插法，即将新元素插入到链表的头部。这种方式的好处是插入效率高，时间复杂度为O(1)，因为只需要修改头节点的指针即可，不需要移动其他节点。
2. 在jdk1.8之前是插入头部的，在jdk1.8中是插入尾部的。HashMap使用尾插法（Tail Insertion）：HashMap是一种基于哈希表的数据结构，用于存储键值对。在HashMap中，每个键值对被映射到一个桶（bucket），每个桶中可以存储多个键值对，通常使用链表或者红黑树来解决哈希冲突。当在HashMap中插入新的键值对时，它会采用尾插法，即将新键值对添加到桶的末尾。这种方式可以保持元素的插入顺序，使得遍历和迭代时能按照插入顺序进行。

1）HashMap用尾插，是因为在并发的场景下，可能会导致链表的死链。

2）Redis是单线程执行，所以不会有并发导致死链，头插法比尾插速度要快很多。所以用头插法即可

需要注意的是，Redis的链表数据结构和Java中的LinkedList并不完全相同，Redis的链表设计更加精简和高效，而Java的LinkedList则包含更多的功能和操作。Redis使用头插法是可以快速插入和删除节点，而HashMap使用尾插法是为了保持元素的插入顺序，并支持按照插入顺序进行遍历和迭代。这两种插入方式都有各自的优势，适用于不同的数据结构和设计需求。

98、Redis有没有像HashMap一样的扩容机制

Redis肯定也有扩容机制，因为如果没有扩容的话，会导致链表越来越长，从而降低查询性能。只不过Redis的扩容机制跟HashMap有点不一样，Redis会有2个hashTable，第二个table只有再扩容的时候使用，当第一个table的容量达到一定量，这个量正常是已有的数据是table大小的时候就会扩容，但是当有在进行持久化的时候，使用量是table容量的5倍的时候扩容扩容也不会一下子都扩容完成，因为一下子把所有的数据从第一个table移到到第二个table耗时太长。所以会采用渐进式rehash，分批次的将数据迁移到第二个table。然后把第一个table变量指向新table。第二个table赋空。

99、项目流量激增，如何处理

1）预估流量

2）全链路压测

3）找到链路瓶颈

4）加机器（增加资源）

5）限流、实时监控和告警

100、Redisson的联锁

联锁的目的是因为redis是属于ap模式的中间件，会存在数据丢失，那么锁就会失效。所以联锁主要做的一件事情就是尽可能的去保证数据不丢失，加锁会加在不同的独立集群机器。当满足一半成功就成功。其实主要思想就是把鸡蛋分散到不同的篮子，降低风险。 只要不是超过一半的失败，就是成功的。

101、核心线程池满后新任务是新建一个线程还是阻塞到队列中去

放到阻塞队列中。线程池中创建线程是需要获取mainlock的，也就是全局锁，全局锁的使用会影响并发效率。所以当核心线程池满后，会将新的任务阻塞到队列中去，与创建最大线程隔离开来，起一个缓冲

的作用。引入到阻塞队列中，是为了在执行execute()方法时，尽可能的避免对全局锁的获取。

102、Mysql里面的RedoLog和BinLog，他们的作用在哪?

1）RedoLog是innoDB里面的事务日志

2）binlog不属于存储引擎级别，不管是什么存储引擎都会有的，是Mysql服务自己去实现的。binlog其实就是一些二进制日志记录文件，主要目的是数据灾备以及主从节点之间的数据同步

InnoDB存储引擎为了提高数据交互性能，加了一个bufferpool内存缓存，如果我数据每次操作都同步给磁盘，就会很慢。

RedoLog和binlog实现的机制是2pc 二阶段提交，redoLog先是准备状态，然后再去写binLog，只有binLog写成功了，redolog才会真正的刷新到磁盘。只有当redoLog 跟binLog都有，才能保证数据一致性。否则，数据回滚。

103、假如Myql的表很大，进行分页的时候，limit 1000000 加载很慢

1）如果id是趋势递增的，那么每次查询都可以返回这次查询最大的ID，然后下次查询，加上大于上次最大id的条件

2）先limit出来主键ID，然后用主表跟查询出来的ID进行inner join 内连接，这样，也能一定上提速，因为减少了回表，查询ID只需要走聚集索引就行

3）缓存

104、为什么不建议用uuid作为主键

1）长度过长：UUID是一个128位的字符串，长度过长，导致在使用索引时会占用大量的存储空间，增加存储成本。而且在数据库的查询和排序中也会降低效率。

2）不容易读懂：UUID是一个由数字和字母组成的字符串，不容易被人类读懂，不利于直接进行数据分析和检查。

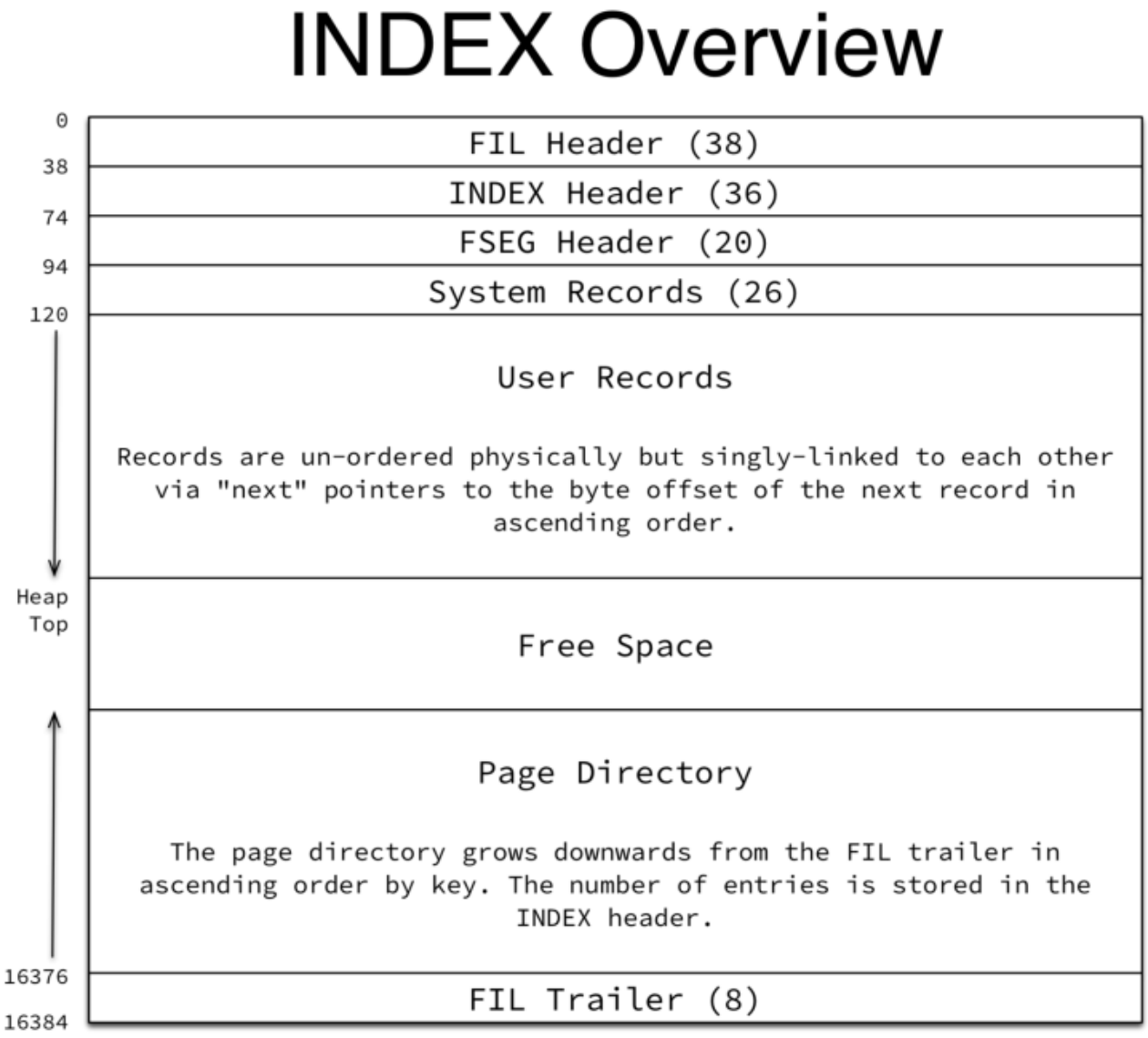
3）被人为改变的可能性：虽然UUID是通过特定算法随机生成的，但是并不能保证绝对随机性，并且UUID是公开的，不保密，如果被人为改变导致主键不唯一就会对数据造成灾难性的影响。

4）造成索引分裂：使用UUID作为主键会导致数据库索引分裂，这是因为UUID是随机分布的，分散地存储在磁盘中，索引也各自独立，存取速度变慢、效率下降。因此，建议将自增长递增数字或时间戳等简单且唯一的值作为主键，可以降低存储成本，便于数据管理与分析。同时，可以通过使用其他唯一约束

来保证数据的唯一性。

105、 为什么不推荐使用UUID

页是 InnoDB 管理的最小单位，常见的有 FSP_HDR，INODE, INDEX 等类型。页结构分为文件头(前38字节)，页数据和文件尾(后8字节)。每个数据页大小为16kb。



系统从磁盘中读取数据到内存时是以磁盘块（block）为基本单位，位于同一个磁盘块中的数据会被一次性读取出来。block大小空间往往没有16kb大，因此innodb每次io操作时都会将若干个地址连续的磁盘块的数据读入内存，从而实现整页读入内存。

MySQL写入数据时，会把数据存放到索引页中。使用UUID作为主键，新行的主键值不一定比之前的主键值大，所以innoDb无法做到总是把新行插入到索引的最后，而需要为新行寻找合适的位置来分配新的空间，这个过程会导致：

- 写入的目标页可能从缓存上移除了，或者还没有加载到缓存上，innodb写入之前需要先从磁盘找到目标页，会产生大量的随机IO
- 因为写入是乱序的，innoDb 要做频繁的分页操作，以便为行产生新的空间，页分裂导致移动大量的数据，一次插入最少需要修改三个页以上
- 频繁的页分裂，页会变得稀疏并被不规则的填充，最终会导致数据会有碎片
- 随机值（uuid和雪花id）载入到聚簇索引，有时候会需要做一次OPTIMEIZE TABLE来重建表并优化页的填充，这将又需要一定的时间消耗。

使用自增主键则可以避免上述问题：

- 自增主键值是顺序的，所以Innodb把每一条记录都存储在一条记录的后面。当达到页面的最大填充因子时候(innodb默认的最大填充因子是页大小的15/16,会留出1/16的空间留作以后的　　修改)，下一条记录就会写入新的页中；
- 数据按照顺序方式加载，主键页就会近乎于顺序的记录填满，提升了页面的最大填充率，不会有页的浪费；
- 新插入的行一定会在原有的最大数据行下一行，mysql定位和寻址很快，不会为计算新行的位置而做出额外的消耗；
- 减少了页分裂和碎片的产生

结论：使用innodb应该尽可能的按主键的自增顺序插入，并且尽可能使用单调的增加的聚簇键的值来插入新行

使用自增主键的问题

- 安全问题：如果采用自增主键，可能存在根据ID值爬取数据库记录，有安全风险
- 系统重构：系统重构或者与其他系统集成时，可能存在新老主键冲突
- 高并发负载，innodb在按主键进行插入的时候会造成明显的锁争用，主键的上界会成为争抢的热点，因为所有的插入都发生在这里，并发插入会导致间隙锁竞争、Auto_Increment锁机制会造成自增锁的抢夺,有一定的性能损失
- 自增主键有限，要考虑主键长度问题。

106、索引覆盖和索引下推

https://baijiahao.baidu.com/s?id=1716515482593299829&wfr=spider&for=pc

https://zhuanlan.zhihu.com/p/470255206

https://blog.csdn.net/m0_46761060/article/details/124196014

https://mp.weixin.qq.com/s?__biz=MzU3Mjk2NDc3Ng==&mid=2247483828&idx=1&sn=c21c6720a83240c2b899ed81ef57b912&chksm=fcc9ab73cbbe2265ac05ec67cca251bba661e995a00ad2acf2fabb69471bdc56987cdcf5b5e9&scene=27

MySQL的索引覆盖和索引下推是两种优化技术，用于提高查询性能和减少不必要的数据读取。它们在以下方面有所不同：

1. 索引覆盖（Index Covering）：
    - 索引覆盖是指通过创建合适的索引，查询可以直接从索引中获取所需的数据，而无需访问表的数据行。这样可以减少磁盘I/O操作和减少数据的传输量，提高查询的执行效率。
    - 索引覆盖适用于那些查询只需要返回索引列数据或包含索引列的查询结果的情况。当查询的列都包含在索引中时，MySQL可以直接从索引中读取所需数据，而不需要额外的表访问操作。
    - 索引覆盖对于大表和频繁查询的场景特别有效，可以显著减少查询的响应时间。
2. 索引下推（Index Condition Pushdown）：
    - 索引下推是指在执行查询时，MySQL将WHERE条件推送到存储引擎层进行过滤，减少从磁盘读取的数据量。它通过将过滤条件下推到存储引擎层进行处理，可以减少不必要的数据读取，提高查询性能。
    - 索引下推适用于那些包含过滤条件的查询。当查询中包含WHERE条件时，MySQL可以将这些条件下推到存储引擎层进行处理，并在存储引擎层使用索引进行过滤，减少需要读取的数据行。
    - 索引下推对于大表和复杂查询条件的场景特别有效，可以减少查询的数据量和读取的磁盘I/O操作。

总结：

- 索引覆盖是通过创建合适的索引，使得查询可以直接从索引中获取所需数据，减少表的访问操作。
- 索引下推是将查询的过滤条件下推到存储引擎层进行处理，减少不必要的数据读取。
- 索引覆盖适用于查询只需要返回索引列数据或包含索引列的查询结果的情况。
- 索引下推适用于包含过滤条件的查询，可以减少查询的数据量和磁盘I/O操作。
- 索引覆盖和索引下推都是优化查询性能的技术，可以根据具体场景选择适合的方法来提高查询效率。

107、 如果一个查询既满足索引覆盖又满足索引下推的条件，通常优先选择索引覆盖。

索引覆盖可以通过直接从索引中获取所需的数据，而无需访问表的数据行，减少了额外的磁盘I/O操作和数据传输量，从而提高查询的执行效率。索引覆盖适用于那些查询只需要返回索引列数据或包含索引列的查询结果的情况。

索引下推是将查询的过滤条件下推到存储引擎层进行处理，减少不必要的数据读取。它可以减少查询的数据量和磁盘I/O操作，提高查询性能。索引下推适用于包含过滤条件的查询。

虽然索引下推也可以减少不必要的数据读取，但是相比于索引覆盖，索引下推仍然需要读取表的数据行，可能会涉及更多的磁盘I/O操作。因此，如果一个查询既可以通过索引覆盖获得所需数据，又满足索引下推的条件，优先选择索引覆盖，因为它可以更有效地减少数据的读取量。

综上所述，当一个查询既满足索引覆盖又满足索引下推时，优先选择索引覆盖以获得更好的查询性能。

108、导致MySQL在之前的版本中没有采用索引下推的原因：

1. 技术限制：早期的MySQL版本的查询优化器设计并没有考虑到索引下推的实现。在索引下推的算法和逻辑方面需要进行较大的改动和优化，这需要投入大量的开发工作和测试。
2. 性能权衡：索引下推虽然可以减少不必要的数据读取，但它也会增加查询优化的复杂度和计算开销。在过去，为了保持查询的执行速度和稳定性，MySQL可能更关注其他优化方案，而不是立即引入索引下推。
3. 用户需求：在过去，对于大部分用户来说，索引覆盖已经可以满足查询性能的需求。索引覆盖可以避免回表操作，已经在许多场景中表现良好，所以MySQL可能认为优化索引覆盖的实现更为重要。

随着时间的推移和MySQL版本的更新，MySQL团队逐渐关注并实现了索引下推这一优化技术。从MySQL 5.6版本开始，索引下推被引入，并在后续版本中不断改进和优化。现在，MySQL中的索引下推已经成为一项重要的优化功能，可以在特定场景下提供更高效的查询性能。

总结起来，MySQL之前没有采用索引下推主要是由于技术限制、性能权衡以及用户需求的考虑。随着技术的发展和用户需求的变化，MySQL逐步引入了索引下推以提高查询性能

109、`synchronized` 锁的加锁和解锁是重量级的，主要有以下几个原因：

1. 互斥访问：`synchronized` 锁用于实现线程的互斥访问，即同一时间只有一个线程可以获取到锁，其他线程需要等待。为了保证互斥性，`synchronized` 锁需要维护锁的状态、等待队列、线

程的阻塞和唤醒等操作，这些操作都需要消耗额外的时间和系统资源，从而导致锁的加锁和解锁是重量级的。

2. 内存屏障和内核态切换：`synchronized` 锁的加锁和解锁过程中涉及到内存屏障的操作，以保证线程的可见性和顺序性。内存屏障会导致 CPU 缓存的刷新和同步操作，从而影响执行效率。此外，`synchronized` 锁的实现中，当无法获取到锁时，线程会进入阻塞状态，需要切换到内核态，这种状态切换也会带来额外的开销。
3. 锁的粒度和竞争：`synchronized` 锁是基于对象的，当多个线程同时竞争同一个对象的锁时，会导致线程之间的竞争和调度开销增加。此时，需要操作系统进行线程调度和阻塞唤醒的操作，以保证锁的互斥性和公平性。竞争和调度开销的增加也会导致锁的加锁和解锁成为重量级操作。

需要注意的是，尽管 `synchronized` 锁是重量级的，但在大部分情况下，它仍然是常用的同步机制，且具有可靠性和稳定性。对于绝大多数应用场景而言，`synchronized` 锁的性能已经足够满足需求。然而，在高并发场景或需要更高性能的情况下，可以考虑使用其他锁机制，如 `ReentrantLock`、`StampedLock` 等，它们提供了更灵活和高效的锁实现。

110、 StampedLock底层原理

`StampedLock` 的底层原理涉及到一种乐观读、悲观写的机制，以及基于版本号的检查和控制。下面是 `StampedLock` 的简要底层原理解释：

1. 内部数据结构：`StampedLock` 内部包含一个名为 `state` 的变量，它维护着锁的状态和版本号。其中低16位表示写锁的计数器，高16位表示读锁的计数器。
2. 乐观读锁：乐观读锁是 `StampedLock` 的特色之一。在获取乐观读锁时，不会阻塞其他线程的读写操作。它通过返回一个 `stamp` 值来标识读锁获取的版本号。线程在读取数据后，如果没有发生写操作，则认为数据是有效的。否则，需要将 `stamp` 值与当前的 `state` 进行比较，以验证数据的有效性。
3. 悲观写锁：悲观写锁是一种排他锁，用于保护临界区的写操作。获取悲观写锁时，如果当前没有任何读锁或写锁被占用，线程可以直接获取写锁，执行写操作。否则，线程会被阻塞，直到没有其他线程持有读锁或写锁。
4. 锁的转换：`StampedLock` 支持从读锁转换为写锁，以及从写锁转换为读锁。转换的过程是无阻塞的，只需要检查当前状态和版本号是否允许转换即可。如果转换成功，版本号会发生变化。
5. 无锁操作：在读锁获取时，如果没有写锁被持有，线程可以直接进行读操作，而无需进行任何阻塞。这种情况下，读锁获取是无锁操作。

总的来说，`StampedLock` 利用了乐观读锁和基于版本号的检查来提高读操作的并发性能，同时保留了悲观写锁的独占性质。通过细粒度的锁控制和无锁操作，`StampedLock` 在某些读多写少的场景中

可以提供更高的吞吐量和性能。然而，它也需要开发人员谨慎使用，避免过度使用乐观读锁而导致数据不一致的问题。

111、 在并发情况下，HashMap使用尾插入可能会导致以下问题：

1. 链表扩容问题：当多个线程同时插入元素到HashMap的同一个桶（bucket）时，由于采用尾插入方式，新的元素会被插入到桶的末尾。这样就可能导致多个线程在同一个桶上进行操作，从而触发链表的扩容操作。链表的扩容需要复制链表中的所有元素到新的链表中，这个过程是比较耗时的，同时也会占用更多的内存空间。
2. 链表长度不均衡问题：并发情况下，多个线程同时插入元素到HashMap的同一个桶时，由于尾插入的方式，新的元素会被插入到桶的末尾，导致链表的长度不均衡。如果某个桶上的链表长度过长，会增加查找、插入和删除的时间复杂度，降低HashMap的性能。
3. 链表结构竞争问题：由于多个线程同时在链表的尾部插入元素，可能导致竞争条件（Race Condition）。当多个线程同时执行插入操作时，它们可能会相互覆盖或丢失数据，导致链表结构出现问题，进而引发数据不一致性或丢失数据的情况。

为了解决并发情况下的问题，Java 8之后的HashMap引入了红黑树（Red-Black Tree）的机制。当链表长度超过阈值（默认为8）时，链表会自动转换为红黑树，以提高在较长链表上的查找、插入和删除操作的效率。红黑树的插入操作是通过比较和旋转的方式进行，相较于链表的尾插入，它能更好地保持平衡和性能。

综上所述，HashMap在并发情况下使用尾插入可能会导致链表扩容问题、链表长度不均衡问题以及链表结构竞争问题。为了解决这些问题，Java 8引入了红黑树机制来提高并发情况下HashMap的性能和稳定性。

112、 在并发情况下，HashMap使用头插入可能会导致以下问题

在并发情况下，HashMap使用头插入可能会导致以下问题：

1. 死锁问题：当多个线程同时插入元素到HashMap的同一个桶（bucket）时，由于采用头插入方式，新的元素会被插入到桶的头部。如果多个线程同时在同一个桶上进行插入操作，并且存在共享资源的互斥访问（例如锁），那么可能会出现死锁的情况。死锁指的是多个线程因为竞争资源而相互等待，导致程序无法继续执行。
2. 数据覆盖问题：并发情况下，多个线程同时插入元素到HashMap的同一个桶时，由于头插入的方式，新的元素会被插入到桶的头部，而原来的元素会被挤到链表的后面。这样就可能导致多个线程

同时在同一个桶上进行操作，从而覆盖了其他线程已经插入的元素，造成数据的覆盖问题。

3. 链表顺序问题：由于多个线程同时在链表的头部插入元素，可能导致链表中元素的顺序被打乱。在遍历链表或者进行其他依赖链表顺序的操作时，可能会出现预期之外的结果，导致程序逻辑错误。

为了避免上述问题，Java 8之后的HashMap引入了红黑树（Red-Black Tree）的机制，用于解决并发情况下的性能和稳定性问题。当链表长度超过阈值（默认为8）时，链表会自动转换为红黑树，以提高在较长链表上的查找、插入和删除操作的效率。红黑树采用平衡的数据结构，可以避免死锁问题和数据覆盖问题，同时保持元素的有序性。

综上所述，HashMap在并发情况下使用头插入可能会导致死锁问题、数据覆盖问题和链表顺序问题。为了解决这些问题，Java 8引入了红黑树机制来提高并发情况下HashMap的性能和稳定性。

113、 mysql的执行计划

MySQL的执行计划是指查询语句在执行过程中数据库优化器所生成的查询执行计划，用于指导数据库引擎如何执行查询并获取结果。执行计划可以展示查询语句的执行路径、执行顺序和使用的索引等信息，有助于开发人员和数据库管理员进行性能调优和优化。

执行计划返回的结果集包含了查询语句的执行计划信息，每一行代表一个执行计划节点，列则提供了有关节点的详细信息。下面是一些常见的列及其含义：

1. `id`：每个节点的唯一标识符，按照从上到下、从左到右的顺序递增。
2. `select_type`：查询类型，描述了节点的执行方式，例如`SIMPLE`（简单查询）、`PRIMARY`（主查询）等。
3. `table`：被访问的表名。
4. `type`：表示访问表的方式，常见的有`ALL`（全表扫描）、`index`（索引扫描）、`range`（范围扫描）等。
5. `possible_keys`：可能使用的索引。
6. `key`：实际使用的索引。
7. `key_len`：使用的索引长度。
8. `ref`：表示上一个表的连接列或常量，用于连接操作。
9. `rows`：估计的读取行数。
10. `Extra`：额外的信息，提供了执行计划中一些其他的细节，例如使用临时表、使用文件排序等。

通过分析执行计划的各个节点的信息，可以获得以下有关查询性能的信息：

- 执行顺序：执行计划从上到下，从左到右展示了查询语句的执行顺序，可以看出哪些操作先执行、哪些操作依赖于其他操作的结果。
- 访问方式：`type`列描述了访问表的方式，可以判断是否发生了全表扫描、索引扫描等操作，从而了解数据访问的效率。
- 使用索引：`possible_keys`和`key`列显示了查询优化器是否使用了索引，通过比较索引的使用

情况可以评估索引的效果。

- 估计的行数： `rows` 列提供了每个操作估计的读取行数，可以判断操作的数据量大小，从而判断查询语句的效率。
- 额外信息： `Extra` 列提供了一些额外的执行细节，例如是否使用了临时表、是否进行了文件排序等，这些信息对于性能优化有一定的指导意义。

通过仔细分析执行计划的结果集，可以了解查询语句的执行情况，识别潜在的性能问题，并采取相应的优化措施来提高查询效率。

在MySQL的执行计划中， `key` 列的值表示实际使用的索引。以下是一些常见的 `key` 值及其含义：

1. `NULL`：表示没有使用索引。
2. 索引名：具体的索引名称，表示使用了该索引进行数据访问。
3. `PRIMARY`：表示使用了主键索引进行数据访问。
4. `auto_increment`：表示使用了自增索引进行数据访问。
5. `const`：表示使用了常量索引，通常出现在使用了唯一索引或主键索引，并且WHERE条件中使用了等值查询。
6. `index_merge`：表示使用了多个索引合并进行数据访问。
7. `unique_subquery`：表示使用了唯一子查询索引进行数据访问。
8. `fulltext`：表示使用了全文索引进行数据访问。
9. `range`：表示使用了范围索引进行数据访问。
10. `index_subquery`：表示使用了子查询索引进行数据访问。
11. `index_merge_union`：表示使用了多个索引合并（并集）进行数据访问。
12. `index_merge_sort_union`：表示使用了多个索引合并（并集，并排序）进行数据访问。

Extra列提供了关于执行计划的附加信息。以下是一些常见的Extra值及其含义：

1. `Using index`：表示查询使用了覆盖索引，即从索引中直接获取所需的数据，而无需访问表的实际数据行。
2. `Using where`：表示查询中使用了WHERE条件进行过滤。
3. `Using temporary`：表示MySQL在执行查询时需要创建临时表来处理中间结果。
4. `Using filesort`：表示MySQL需要对结果进行排序操作。
5. `Using join buffer`：表示MySQL使用了连接缓存来处理连接操作。
6. `Distinct`：表示使用了DISTINCT关键字进行去重操作。
7. `Range checked for each record`：表示对于每条记录都进行了范围检查。
8. `Using index condition`：表示使用了索引条件进行过滤。
9. `Using where; Using index`：表示同时使用了WHERE条件过滤和索引进行数据访问。

10. `Using index for group-by`：表示使用了索引来执行GROUP BY操作。
11. `Using index for sort`：表示使用了索引来执行排序操作。
12. `Using index for order by`：表示使用了索引来执行ORDER BY操作。
13. `Using index for distinct`：表示使用了索引来执行DISTINCT操作。
14. Using index condition：表示使用了索引下推

113、MySQL索引失效可能发生在以下场景中：

1. 不使用索引列进行过滤：当查询中的WHERE子句没有使用索引列进行过滤时，MySQL无法有效利用索引，导致索引失效。例如，如果一个表有索引列A，但查询条件是基于未索引的列B进行过滤，那么索引将失效。
2. 对索引列进行函数操作：如果在查询条件中对索引列进行了函数操作，例如使用函数对索引列进行了计算、转换或截取等，这会导致索引失效。因为MySQL无法直接使用索引来匹配经过函数操作后的列值。
3. 非前缀索引的前缀匹配：当使用非前缀索引的前缀匹配时，索引也会失效。例如，如果一个表有一个字符串列，而索引只覆盖了该列的前几个字符，而查询使用了整个字符串进行匹配，那么索引将无法有效使用。
4. 表达式索引的使用限制：某些类型的索引，如全文索引、空间索引和JSON索引，具有特定的使用限制。如果在查询中使用这些索引时违反了限制条件，索引将失效。
5. 数据量过小导致全表扫描更快：当表中的数据量非常小（通常小于一定的阈值）时，执行全表扫描可能比使用索引更快，因此MySQL可能会选择放弃索引而执行全表扫描。
6. 统计信息不准确：MySQL使用统计信息来评估执行计划，并决定是否使用索引。如果统计信息不准确或过期，MySQL可能会做出错误的决策，导致索引失效。
7. 索引列类型不匹配：如果查询条件中的数据类型与索引列的数据类型不匹配，索引将无法使用。例如，如果索引列是字符串类型，而查询条件中使用了数值类型，索引将失效。

114、死锁条件

死锁是指多个进程或线程因为竞争资源而陷入无限等待的状态。死锁发生需要满足以下四个条件，它们被称为死锁的必要条件：

1. 互斥条件（Mutual Exclusion）：至少有一个资源被独占，即一次只能由一个进程或线程使用。如果一个资源已经被一个进程或线程占用，其他进程或线程必须等待。
2. 请求与保持条件（Hold and Wait）：一个进程或线程在持有至少一个资源的同时，又请求获取其他进程或线程当前占有的资源。如果一个进程或线程已经持有了某个资源，但又请求获取其他进程

或线程当前持有的资源，而其他进程或线程又在等待该进程或线程当前持有的资源，就会产生死锁。

3. 不可抢占条件（No Preemption）：已经分配给一个进程或线程的资源不能被强行抢占，只能由该进程或线程主动释放。即资源只能在使用完毕后由进程或线程自愿释放，其他进程或线程无法强制剥夺。
4. 循环等待条件（Circular Wait）：存在一组进程或线程P1、P2、...Pn，其中P1等待P2所持有的资源，P2等待P3所持有的资源，...，Pn等待P1所持有的资源。形成一个进程或线程的循环等待链，导致循环等待的产生。

115、分库分表三种应用场景

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。

116、CurrentHashMap

ConcurrentHashMap 是 Java 中并发编程中常用的线程安全的哈希表实现，它在多线程环境下提供了高效的并发操作。下面我将详细介绍 ConcurrentHashMap 的底层原理，并指出 Java 1.6 和 1.8 版本之间的不同点。

Java 1.6 版本中的 ConcurrentHashMap：

1. 数据结构：Java 1.6 版本的 ConcurrentHashMap 使用分段锁（Segment）的方式来实现并发控制。内部数据结构由一个 Segment 数组组成，每个 Segment 都是一个独立的哈希表，负责管理一部分数据。每个 Segment 有自己的锁，可以独立地对数据进行并发操作。
2. 并发度：ConcurrentHashMap 的并发度就是 Segment 的数量，默认为 16。这样多个线程可以同时访问不同的 Segment，提高了并发性能。

3. put 操作：在插入元素时，先根据 key 的哈希值找到对应的 Segment，然后在该 Segment 中进行插入操作。插入操作使用了锁机制，不同的 Segment 可以并发执行插入操作。
4. get 操作：在获取元素时，也需要先根据 key 的哈希值找到对应的 Segment，然后在该 Segment 中进行查找操作。查找操作不需要加锁，可以并发执行。
5. 扩容：在扩容时，每个 Segment 都会进行独立的扩容操作，不会对其他 Segment 产生影响。

Java 1.8 版本中的 ConcurrentHashMap：

1. 数据结构：Java 1.8 版本对 ConcurrentHashMap 进行了重构，使用了更复杂的数据结构。它不再使用分段锁，而是采用了一种称为 "Node + CAS" 的方式来实现并发控制。内部数据结构由一个数组和链表或红黑树组成。
2. 并发度：Java 1.8 版本中的 ConcurrentHashMap 并发度仍然是数组的长度，默认为 16。但每个数组元素不再是一个独立的哈希表，而是一个链表或红黑树的头节点。
3. put 操作：在插入元素时，先根据 key 的哈希值找到对应的数组元素，然后在链表或红黑树中进行插入操作。插入操作使用了 CAS（Compare and Swap）机制，保证了并发安全性。
4. get 操作：在获取元素时，同样根据 key 的哈希值找到对应的数组元素，然后在链表或红黑树中进行查找操作。查找操作也是通过遍历链表或红黑树来实现的，但由于链表长度较短或使用了红黑树的特性，查找效率较高。
5. 扩容：在扩容时，ConcurrentHashMap 会对整个数组进行扩容，而不是单独对某个 Segment 进行扩容。扩容过程中，数组元素会逐渐转换为红黑树，提高了查找效率。

总结：
Java 1.6 版本的 ConcurrentHashMap 使用分段锁机制，通过细粒度的锁来实现并发控制；而 Java 1.8 版本的 ConcurrentHashMap 则采用了更复杂的数据结构和 CAS 机制来提高并发性能。Java 1.8 版本中的 ConcurrentHashMap 在并发度、put 操作、get 操作和扩容方面有了较大的改进，提供了更高效、更安全的并发访问能力。

在 Java 8 版本的 ConcurrentHashMap 中，CAS（Compare and Swap）和 synchronized 锁都有各自的使用场景：

CAS 使用场景：

1. 并发更新操作：当多个线程并发地修改 ConcurrentHashMap 中的某个桶（bucket）时，会使用 CAS 操作来实现无锁的原子更新。CAS 操作允许多个线程同时尝试更新某个值，但只有一个线程会成功，其他线程需要重新尝试。这样可以避免使用锁带来的线程阻塞和上下文切换的开销，提高并发性能。

2. 扩容操作：当 ConcurrentHashMap 需要扩容时，使用 CAS 操作来保证并发的安全扩容。CAS 操作用于判断是否有其他线程正在进行扩容操作，如果没有，则当前线程可以执行扩容操作，否则需要重新尝试。

synchronized 锁使用场景：

1. 初始化操作：在 ConcurrentHashMap 进行初始化操作时，会使用 CAS。

2. 桶锁定：在 ConcurrentHashMap 中，每个桶（bucket）都对应一个锁。当某个线程需要对桶进行修改操作时，会首先通过 synchronized 锁定对应的桶，确保线程安全。

需要注意的是，Java 8 版本的 ConcurrentHashMap 引入了分段锁的机制，将原来的全局锁（Java 1.6）改为了细粒度的锁（Java 1.8）。这样做的目的是减小锁的粒度，提高并发性能。因此，在 Java 8 版本的 ConcurrentHashMap 中，并发更新操作和扩容操作会使用 CAS 操作，而初始化操作和桶锁定操作会使用 synchronized 锁。

117、CPU利用率过高

上下文切换过多、CPU资源过渡消耗（大量创建线程，有线程一直占用CPU）

1）CPU利用率过高的线程一直是同一个，说明程序中存在线程长期占用CPU没有释放的情况，这种情况直接通过jstack获得线程的Dump日志，定位到线程日志后就可以找到问题的代码。

2）CPU利用率过高的线程id不断变化，说明线程创建过多，需要挑选几个线程id，通过jstack去线程dump日志中排查。

3）最后有可能定位的结果是程序正常，只是在CPU飙高的那一刻，用户访问量较大，导致系统资源不够。

118、Innodb 的事务实现原理

1）原子性：是使用 undo log 来实现的，如果事务执行过程中出错或者用户执行了rollback，系统通过 undo log 日志返回事务开始的状态。

2）持久性：使用 redo log 来实现，只要 redo log 日志持久化了，当系统崩溃，即可通过redo log 把数据恢复。

3）隔离性：通过锁以及 MVCC,使事务相互隔离开。

4）一致性：通过回滚、恢复，以及并发情况下的隔离性，从而实现一致性。

119、mysql的事务隔离级别

1）读未提交，在这种隔离级别下，可能会产生脏读、不可重复读、幻读。

2）读已提交（RC），在这种隔离级别下，可能会产生不可重复读和幻读。

3）可重复读（RR），在这种隔离级别下，可能会产生幻读

4）串行化，在这种隔离级别下，多个并行事务串行化执行，不会产生安全性问题。

这四种隔离级别里面，只有串行化解决了全部的问题，但也意味着这种隔离级别的性能是最低的。

在Mysql里面，InnoDB引擎默认的隔离级别是RR（可重复读），因为它需要保证事务ACID特性中的隔离性特征。

120、mysql怎么保证ACID

1）原子性：是使用 undo log 来实现的，如果事务执行过程中出错或者用户执行了rollback，系统通过undo log 日志返回事务开始的状态。

2）持久性：使用 redo log 来实现，只要 redo log 日志持久化了，当系统崩溃，即可通过redo log 把数据恢复。

3）隔离性：通过锁以及 MVCC,使事务相互隔离开。

4）一致性：通过回滚、恢复，以及并发情况下的隔离性，从而实现一致性。

121、 stringTable、字符串常量池、常量池的区别

在编程中，字符串常量池、常量池和stringTable是不同概念，它们在不同的编程语言和上下文中有着不同的含义。下面是它们的一般解释和区别：

1. 字符串常量池（String Constant Pool）：字符串常量池是一种存储字符串常量的内存区域。它是在编译时或运行时创建的，用于存储程序中的字符串常量。在Java中，字符串常量池是位于堆内存的一部分，用于存储字符串字面值（通过双引号表示的字符串）。字符串常量池的目的是避免重复创建相同的字符串对象，提高内存利用率和性能。
2. 常量池（Constant Pool）：常量池是一种在编译时创建并存储常量的数据结构。它是在编译过程中生成的，并包含程序中使用的常量值、符号引用和其他编译时需要的信息。常量池可以包含各种类型的常量，例如整数、浮点数、字符串、类和方法的符号引用等。在Java中，常量池是位于类文件中的一部分，用于存储编译时生成的常量数据。
3. stringTable：stringTable是Java虚拟机（JVM）中的一个内部数据结构，用于存储字符串对象的引用。它是在运行时动态创建和维护的，用于优化字符串的处理和管理。JVM中的stringTable可以被看作是一个哈希表，它保存了已经创建的字符串对象的引用，以便在需要时可以重用相同的字符串对象。它的主要目的是减少字符串对象的内存占用和提高性能。

总结起来，字符串常量池是用于存储字符串常量的内存区域，常量池是用于存储各种常量的数据结构，而stringTable是JVM中的内部数据结构，用于优化字符串对象的处理和管理。它们在不同的编程语

言和上下文中有着不同的实现和用途。

122、 stringTable和字符串常量池有和联系

在Java中，stringTable（字符串表）是一种在运行时动态创建和维护的数据结构，用于存储字符串对象的引用。它与字符串常量池密切相关，可以说字符串常量池是stringTable的一部分。

Java的字符串常量池是存储字符串字面值（通过双引号表示的字符串）的区域，它位于堆内存中。当程序中出现字符串字面值时，Java编译器会将其添加到字符串常量池中，并在运行时重用相同的字符串对象。这样可以节省内存，避免重复创建相同的字符串对象。

而stringTable是JVM在运行时创建的哈希表，用于保存字符串对象的引用。当代码中使用字符串字面值创建字符串对象时，JVM会在字符串常量池中查找该字面值对应的字符串对象。如果找到了，就直接返回该对象的引用；如果没有找到，就在堆内存中创建新的字符串对象，并将其添加到字符串常量池中，并在stringTable中建立引用。这样，在后续的代码中，如果出现相同的字符串字面值，JVM会直接从stringTable中获取已存在的字符串对象的引用，而不是重复创建新的对象。

因此，可以说字符串常量池是存储字符串字面值的区域，而stringTable是JVM在运行时维护的用于优化字符串对象处理和重用的数据结构。字符串常量池和stringTable的关联在于，字符串常量池中的字符串对象在运行时会被添加到stringTable中进行管理和重用。

123、mysql数据什么时候在内存中，什么时候在磁盘中

在MySQL中，数据可以存在于内存中或磁盘中，具体取决于数据的访问模式和MySQL的配置。

1. 内存中的数据：MySQL使用缓冲池（Buffer Pool）来存储在内存中的数据。缓冲池是一个内存区域，用于缓存热点数据和频繁访问的数据页。当数据被读取或修改时，MySQL首先检查缓冲池中是否存在所需的数据页。如果数据页已经在缓冲池中，MySQL可以直接从内存中读取或修改数据，这样可以提高访问速度。因此，经常被访问的数据通常会在内存中。
2. 磁盘中的数据：当数据不在缓冲池中或缓冲池空间不足时，MySQL需要从磁盘读取数据。磁盘是数据的永久存储介质，在磁盘上存储的数据通常是不活跃或很少访问的数据。当数据被写入或更新时，MySQL通常会先将其写入到磁盘的日志文件中，然后再异步地将数据刷新到磁盘的数据文件中。

需要注意的是，MySQL还使用了一些技术来优化数据的访问和存储。例如，索引（Index）可以加速数据的查找和检索，部分索引可以保存在内存中。此外，MySQL还使用了预读（Prefetching）技术来预先读取可能会访问的数据页，以减少磁盘访问的延迟。

MySQL的数据存储和管理是一个复杂的过程，涉及到缓存、文件系统、磁盘IO等多个方面。具体的数据在内存中还是磁盘中的情况会根据数据的使用模式、MySQL配置和硬件资源等因素而异。通过合理

的配置和优化，可以提高MySQL的性能和数据访问效率。

124、ForkAndJoin

Fork/Join 框架是 Java7 提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

- ForkJoinTask：我们要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork() 和 join() 操作的机制，通常情况下我们不需要直接继承 ForkJoinTask 类，而只需要继承它的子类，Fork/Join 框架提供了以下两个子类：
  - RecursiveAction：用于没有返回结果的任务。
  - RecursiveTask ： 用于有返回结果的任务。
- ForkJoinPool ： ForkJoinTask 需要通过 ForkJoinPool 来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

Java | 复制代码

```java
public class ForkAndJoin {
    private static java.util.concurrent.Executors Executors;
    //获得执行ForkAndJoin任务的线程池
    private static final ForkJoinPool forkJoinPool = (ForkJoinPool)
            Executors.newWorkStealingPool();


    public static void main(String args[]) throws ExecutionException, InterruptedException {
        List<Integer> list = new ArrayList<>();
        for (int i = 1; i < 101; i++) {
            list.add(i);
        }
        ForkAndJoinRequest request = new ForkAndJoinRequest(0, list.size() - 1, list);
        forkJoinPool.submit(request);
        System.out.println(request.get());
    }


}

//定义request继承RecursiveTask，并实现compute方法
class ForkAndJoinRequest extends RecursiveTask<Integer> {

    private int start;
    private int end;
    private List<Integer> list;

    public ForkAndJoinRequest(int start, int end, List<Integer> list) {
        this.start = start;
        this.end = end;
        this.list = list;
    }

    @Override
    protected Integer compute() {
        int count = end - start;
        if (count <= 25) { //如果需要累加的数量小于等于25，则直接执行
            int result = 0;
            for (int i = start; i <= end; i++) {
                result += i;
            }
            return result;
        } else { //否则fork出其他request
```

```
44                int mid = (start + end) / 2;
45                ForkAndJoinRequest request1 = new ForkAndJoinRequest(start, mi
    d, list);
46                request1.fork(); //调用fork方法将自身放入等待队列中等待执行
47                ForkAndJoinRequest request2 = new ForkAndJoinRequest(mid + 1,
    end, list);
48                request2.fork();
49
50                //等待执行结果
51                return request1.join() + request2.join();
52
53            }
54        }
55    }
```

125、 多线程为何不能自动传递上下文

多线程之间不能自动传递上下文的原因是，每个线程都是独立执行的，它们具有自己的栈和执行上下文。当一个线程创建另一个线程时，新线程不会自动继承父线程的上下文信息。

这种设计决策是为了确保线程之间的隔离性和独立性。如果上下文自动传递给新线程，那么不同线程之间的操作将会相互干扰，导致难以预测和调试的问题。

此外，上下文信息可能包含敏感数据，如用户凭证、密码等。如果自动传递上下文，可能会导致潜在的安全风险。

为了在多线程环境中传递上下文信息，可以使用一些手动的机制，如ThreadLocal类。ThreadLocal允许在每个线程中创建一个独立的上下文实例，并使得线程中的代码能够访问和修改这个实例。通过ThreadLocal，可以将上下文信息与当前线程关联起来，使得在线程内的任何代码都可以轻松地访问到该信息，而不需要显式地传递它。

虽然这需要开发人员显式地管理上下文信息的创建、访问和销毁，但同时也提供了更大的灵活性和控制权，确保了线程之间的隔离和安全性。

126、 永久代发生内存不足主要是什么引起

永久代（Permanent Generation）是Java虚拟机内存模型中的一个区域，用于存储类的元数据、静态变量、常量池等信息。在较早的Java版本中，永久代是用于存放永久性的数据，例如类的定义信息、方法、字符串常量等。但从Java 8开始，永久代被元空间（Metaspace）所取代。

然而，如果你提到的是在早期版本的Java中，永久代发生内存不足，可能是由以下几个主要原因引起：

1. 类加载器泄漏（ClassLoader Leak）：如果在应用程序中存在类加载器泄漏，即加载的类无法被正确卸载，那么这些类和相关的元数据会一直保留在永久代中，导致永久代内存不断增长，最终耗尽内存。
2. 大量动态生成类：某些应用程序可能会在运行时动态生成大量的类，例如使用动态代理、字节码增强等技术。这些动态生成的类会被加载到永久代中，如果生成的类数量过多，会导致永久代内存耗尽。
3. 字符串常量池滥用：字符串常量池是永久代的一部分，如果应用程序中大量使用字符串并且频繁创建新的字符串，这些字符串可能会被添加到常量池中并保留在永久代中。如果字符串的创建过于频繁或者字符串过多，可能导致永久代内存不足。
4. 大量反射操作：反射操作涉及到动态生成类、调用私有方法等，这些操作会增加永久代的负担。如果应用程序中大量使用反射操作，可能会导致永久代内存耗尽。

需要注意的是，从Java 8开始，永久代被元空间所取代，元空间使用的是本地内存而不是Java堆内存，因此不存在永久代内存不足的问题。而是将类的元数据存储在本地内存中，根据系统的实际可用内存进行分配，避免了永久代内存不足的问题。

127、NIO的缓冲区操作的是直接内存吗

在 Java NIO 中，有两种类型的缓冲区：堆缓冲区（Heap Buffer）和直接缓冲区（Direct Buffer）。

堆缓冲区是由 JVM 在堆内存中分配的普通 Java 对象，数据存储在 Java 堆中。对于堆缓冲区的操作，需要经过一次内存复制，即将数据从堆缓冲区复制到内核缓冲区或者从内核缓冲区复制到堆缓冲区。

直接缓冲区是由 JVM 在堆外（直接内存）分配的，即数据存储在操作系统的内存中，不属于 Java 堆。直接缓冲区可以通过使用 ByteBuffer.allocateDirect() 方法来创建。对于直接缓冲区，数据可以直接在 JVM 和操作系统内存之间进行传输，避免了不必要的内存复制。

在某些场景下，直接缓冲区的性能可能比堆缓冲区更好，特别是在进行大量数据的 I/O 操作时，直接缓冲区可以减少数据复制的开销。

需要注意的是，尽管直接缓冲区可以避免一次内存复制，但其创建和释放的开销较大，因为涉及到 JVM 和操作系统之间的交互。因此，在选择使用堆缓冲区还是直接缓冲区时，需要根据具体的应用场景和性能需求进行评估和权衡。

128、HashMap 在 JDK 1.8 有什么改变

https://juejin.cn/post/6934332114280120327#heading-5

hashMap的put()方法：

1）判断键值对数组 table 是否为空或为 null，否则执行 resize() 进行扩容；

2）根据键值 key 计算 hash 值得到插入的数组索引 i，如果 table[i] == null，直接新建节点添加，转向 6，如果 table[i] 不为空，转向 3；

3）判断 table[i] 的首个元素是否和 key 一样，如果相同直接覆盖 value，否则转向 4，这里的相同指的是 hashCode 以及 equals；

4）判断 table[i] 是否为 treeNode，即 table[i] 是否是红黑树，如果是红黑树，则直接在树中插入键值对，否则转向 5；

5）遍历 table[i]，判断链表长度是否大于 8，大于 8 的话把链表转换为红黑树，在红黑树中执行插入操作，否则进行链表的插入操作；遍历过程中若发现 key 已经存在直接覆盖 value 即可；

6）插入成功后，判断实际存在的键值对数量 size 是否超过了负载 threshold，如果超过，进行扩容。

hashMap的resize()

1）原 table 数组的大小已经最大，无法扩容，则修改 threshold 的大小为 Integer.MAX_VALUE。产生的效果就是随你碰撞，不再扩容；

2）原 table 数组正常扩容，更新 newCap（新数组的大小） 与 newThr（新数组的负载）；

3）原 table 数组为 null ‖ length 为 0，则扩容使用默认值；

4）原 table 数组的大小在扩容后超出范围，将 threshold 的大小更改为 Integer.MAX_VALUE。

hashMap的rehash()

1.8主要对重新定位元素在哈希表中的位置

总结：

1）明白静态内部类 Node 的相关实现，清楚 HashMap 的底层实现是有关 Node 的 table 数组（哈希表）。

2）注意使用 HashMap 时最好使用不变的对象作为 key。

3）注意 HashMap 计算 key 的 hash 值时，使用了低位与高位异或的方式，返回最终的 hashcode。

4）了解 HashMap 中的定位方式：(n – 1) & hash。

5）在 HashMap 中使用链地址法解决冲突，并且当链表的节点个数大于 8 的时候，会转换为红黑树。（JDK 1.8 新特性）

6）JDK 1.8 中使用尾插法进行 put 与 resize，JDK 1.7 中使用头插法进行 put 与 resize。

7）JDK 1.8 中的 rehash 过程不用重新计算元素的哈希值，因为元素的位置只有两种情况：原位置 与 原位置 + 原本哈希表的长度。

8）清楚多线程环境下使用 HashMap 可能会造成的一种错误---形成环形链表。

129、ThreadLocal原理

ThreadLocal是Java中的一个线程局部变量，它提供了一种在多线程环境下保持变量的独立副本的机制。每个线程都可以独立地修改自己的副本，而不会影响其他线程的副本。

ThreadLocal的原理如下：

1）每个Thread对象都维护了一个ThreadLocalMap对象，该对象是一个键值对的哈希表，用于存储线程局部变量的值。

2）ThreadLocalMap的键是ThreadLocal对象，值是线程局部变量的副本。

3）当通过ThreadLocal的get()方法获取线程局部变量时，实际上是通过当前线程获取对应的ThreadLocalMap对象，然后根据ThreadLocal对象作为键来获取对应的值。

4）当通过ThreadLocal的set()方法设置线程局部变量时，也是通过当前线程获取对应的ThreadLocalMap对象，然后将ThreadLocal对象和要设置的值作为键值对存入ThreadLocalMap中。

5）当线程结束时，Thread对象会被垃圾回收，同时对应的ThreadLocalMap对象也会被回收，从而避免了内存泄漏。

通过ThreadLocal，每个线程都拥有自己独立的变量副本，可以在多线程环境下实现线程间的数据隔离。它常被用于实现线程安全的类，将线程共享的数据保存在ThreadLocal中，使得每个线程都能访问自己的数据副本，避免了线程之间的数据竞争和同步问题。

需要注意的是，虽然ThreadLocal可以提供线程间的数据隔离，但也可能导致内存泄漏的问题。如果在使用ThreadLocal的时候没有及时清理，即使线程结束，ThreadLocal中的值仍然存在于对应的ThreadLocalMap中，可能导致内存泄漏。因此，在使用ThreadLocal时，应该及时清理不再需要的线程局部变量，可以通过调用ThreadLocal的remove()方法来清理。

130、一致性 Hash 的缺点

一致性哈希（Consistent Hashing）是一种用于分布式系统中数据分片和负载均衡的算法，它将数据分散到节点上，并尽可能地保持节点数量的稳定。尽管一致性哈希具有许多优点，但也存在一些缺点：

1）数据倾斜：一致性哈希算法将数据分散到不同的节点上，但是在某些情况下，由于哈希函数的不均匀性或节点的动态变化，可能导致数据在节点上分布不均衡。某些节点可能负载过高，而其他节点负载较低。这可能导致性能不均衡，影响系统的整体吞吐量。

2）节点的加入和移除：在一致性哈希中，当一个节点加入或移除时，只会影响到少量的数据迁移。然而，在某些情况下，节点的加入或移除可能引起大规模的数据迁移，导致系统的不稳定性和延迟增加。

3）缓存命中率下降：一致性哈希的算法将数据均匀地散布在不同的节点上，这在一定程度上增加了缓存的命中率。然而，由于节点的变化或者数据倾斜的问题，某些数据可能需要重新计算哈希并且迁移到新的节点上，这可能导致缓存的命中率下降。

4）增加系统复杂性：一致性哈希算法需要维护哈希环、节点信息以及数据迁移等相关逻辑。这增加了系统的复杂性，需要更多的开发和维护工作。

尽管一致性哈希存在一些缺点，但在许多分布式系统中仍然被广泛应用。它可以提供较好的负载均衡性能和节点扩展性，但在设计和使用时需要仔细考虑其中的缺点，并采取相应的措施来解决或减轻这些问题。

131、怎么防止 SQL 注入

要防止SQL注入攻击，可以采取以下一些措施：

- 使用参数化查询（Prepared Statements）或预编译语句：使用参数化查询可以将用户提供的输入作为参数，而不是将其直接拼接到SQL语句中。这样可以防止恶意输入被解释为SQL代码。
- 使用ORM框架：使用对象关系映射（ORM）框架可以帮助自动处理SQL语句的生成和参数化查询，减少手动编写SQL的机会。
- 输入验证和过滤：对于用户输入的数据，进行验证和过滤，确保只接受预期的数据类型和格式。可以使用白名单过滤或正则表达式来限制输入的字符集合。
- 最小权限原则：数据库用户应该被授予最小的权限，仅限于执行必要的操作。这样可以限制攻击者对数据库的操作能力。
- 避免动态拼接SQL语句：尽量避免直接拼接用户输入的数据到SQL语句中，因为这会使得注入攻击更加容易。使用参数化查询和预编译语句可以解决这个问题。
- 对特殊字符进行转义：在拼接SQL语句时，对特殊字符（如单引号、双引号、分号等）进行转义处理，以防止它们被误解为SQL代码的一部分。
- 日志记录和监控：记录所有的SQL查询日志，并进行监控和审计。这样可以快速发现异常的SQL语句和潜在的注入攻击。
- 定期更新和维护数据库软件：及时应用数据库厂商发布的安全补丁和更新，以修复已知的安全漏洞。

综合采用上述措施可以有效降低SQL注入攻击的风险。同时，开发者应该具备安全意识，了解常见的安全漏洞和攻击技术，并对代码进行安全审计和测试，以保障应用的安全性。

132、sql编译过程通常包括以下几个步骤

- 词法分析（Lexical Analysis）：将SQL查询语句分解为词（Token），如关键字、标识符、运算符等。
- 语法分析（Syntax Analysis）：将词按照语法规则组织成语法树（Syntax Tree），检查语句是否符合SQL语法。
- 语义分析（Semantic Analysis）：验证查询语句的语义正确性，如表、列是否存在，权限验证等。
- 查询优化（Query Optimization）：在编译过程中，数据库会进行查询优化，目的是找到最优的执

行计划。优化器会考虑多个执行计划的选择，并估计每个执行计划的成本和效率，以选择最优的执行计划。

- 执行计划生成（Execution Plan Generation）：根据查询优化的结果，生成最终的执行计划。执行计划是一个具体的执行指令序列，描述了如何访问和处理数据库中的数据，以及执行各种操作（如扫描表、使用索引、连接等）。

133、编译和解释有什么区别

编译（Compilation）和解释（Interpretation）是两种不同的代码执行方式。

编译是将源代码转换为机器代码或字节码的过程。在编译过程中，编译器会对整个源代码进行静态分析、语法检查、优化等处理，并生成可执行的机器代码或字节码文件。这个生成的代码可以直接由计算机硬件或虚拟机执行，以达到更高的执行效率。

解释是逐行或逐块地将源代码转换为可执行的指令，并在运行时逐条执行。在解释过程中，解释器会解析源代码，逐条执行代码，并在执行时对代码进行动态解释和执行。　　解释过程没有生成额外的可执行文件，而是直接基于源代码进行解释执行。

主要区别如下：

1. 执行方式：编译器将源代码一次性转换为可执行的机器代码或字节码文件，而解释器在运行时逐条解释和执行源代码。
2. 执行效率：编译后的代码由机器直接执行，因此通常具有较高的执行效率。而解释器在运行时动态解释和执行源代码，执行效率相对较低。
3. 可移植性：编译生成的机器代码或字节码文件可以在不同的平台上执行，具有较好的可移植性。而解释器通常需要针对每个平台编写特定的解释器，因此可移植性相对较差。
4. 调试和错误处理：编译过程中可以进行静态分析和优化，但在出现错误时较难进行动态调试。而解释器在运行时逐条执行源代码，因此可以更容易地进行动态调试和错误处理。

需要注意的是，编译和解释不是绝对的二选一。实际上，现代的编程语言和执行环境通常会采用混合的执行方式。例如，某些语言会将源代码首先编译成中间代码（如Java的字节码），然后由虚拟机进行解释执行或即时编译成机器代码。这种混合的执行方式既兼具编译的高效性，又保留了解释的灵活性。

134、使用 Mybatis 时，调用 DAO（Mapper）接口时是怎么调用到 SQL 的

在使用 MyBatis时，调用DAO（Mapper）接口时，是通过动态代理（Dynamic Proxy）机制来实现将方法调用映射到对应的SQL语句的。

MyBatis通过为DAO接口生成代理对象，该代理对象实现了DAO接口的所有方法。当应用程序调用DAO接口的方法时，实际上是调用了代理对象的对应方法。

在代理对象的方法中，MyBatis会解析配置文件或注解中定义的SQL映射，找到与调用方法对应的SQL语句。然后，MyBatis将方法参数与SQL语句中的参数进行映射，生成完整的SQL语句。

接下来，MyBatis将生成的SQL语句传递给底层的JDBC驱动程序执行，执行结果会被返回给代理对象的方法。最后，代理对象将查询结果或执行状态返回给调用者。

通过这种方式，MyBatis将DAO接口的方法调用与SQL语句的映射关系解耦，使得开发者可以通过简单的方法调用来执行复杂的SQL查询和更新操作。

135、FactoryBean 的使用场景

FactoryBean的使用场景包括但不限于以下几个方面：

- 复杂对象的创建和初始化：当需要创建的Bean实例比较复杂，需要进行一些额外的初始化或配置操作时，可以使用FactoryBean来封装这些逻辑。例如，可以在FactoryBean中完成对象的初始化、依赖注入、连接池的初始化等操作。
- 整合第三方库：如果需要将第三方库中的对象纳入Spring容器的管理，可以通过FactoryBean来创建并管理这些对象。FactoryBean可以将第三方库的对象实例化，并将其纳入Spring容器，使得可以在应用中方便地使用。
- 动态代理：FactoryBean可以与Spring的AOP（面向切面编程）功能结合使用，用于创建代理对象。通过FactoryBean，可以在获取Bean实例时，返回一个经过AOP代理的实例，从而实现对Bean的拦截、增强等操作。
- 条件化创建： 在某些情况下，根据特定条件需要创建不同的Bean实例。FactoryBean可以根据条件动态选择创建不同的Bean实例，从而提供更灵活的Bean创建策略。
- 对象池管理：如果需要创建和管理多个实例，并对这些实例进行池化管理，可以使用FactoryBean来实现对象池的功能。
- 封装配置逻辑：FactoryBean可以将复杂的配置逻辑封装在一个工厂类中，提供统一的接口来获取Bean实例。这样可以简化配置文件的编写，并使得配置更加灵活和可扩展。
- 动态数据源切换：在一些多数据源的场景中，可以使用FactoryBean来动态创建数据源，并将其设置为特定的数据源，从而实现动态数据源切换。

总的来说，FactoryBean的使用场景涵盖了在Bean的创建、初始化、配置、管理等方面的多种情况。它提供了灵活的机制，使得开发者可以通过自定义的逻辑来创建和配置Bean实例，从而增强了Spring框架的可定制性和扩展性。

136、使用使用@Configuration、@Bean等注解来完成Bean的创建，为何还要使用FactoryBean

- 灵活性和定制能力： FactoryBean提供了更高级的灵活性和定制能力。通过自定义FactoryBean，你可以编写自己的逻辑来创建和配置对象，实现更复杂的创建过程，例如动态决定对象的实例化逻辑、根据条件选择不同的实现、在对象创建前后执行一些操作等。这使得FactoryBean在一些特殊和复杂的场景下更具优势。
- 抽象屏蔽： FactoryBean可以将底层的复杂逻辑抽象屏蔽起来，使得应用程序的其他部分不需要关注具体的实现细节。它提供了一种封装复杂对象创建过程的机制，让使用者只需关心获取Bean实例的逻辑，而不需要关心具体的创建和初始化过程。这样可以降低代码的耦合度，提高代码的可维护性和可扩展性。
- 懒加载和对象池： FactoryBean支持懒加载和对象池的功能。它可以延迟创建对象，只有在需要时才真正创建。同时，FactoryBean还可以管理对象的生命周期，实现对象的复用和对象池的管理。

137、什么情况下对象不能被代理在Java中，有几种情况下对象不能被代理：

- Final类： Final类是指被final修饰的类，它们不能被继承。由于代理是通过生成子类来实现的，所以无法为Final类生成代理。
- Final方法： Final方法是指被final修饰的方法，它们不能被子类重写。由于代理是通过生成子类来实现的，所以无法为Final方法生成代理。
- Private方法： Private方法是类中被private修饰的方法，它们不能被子类访问或重写。由于代理是通过生成子类来实现的，无法在子类中访问和重写Private方法
- Static方法： Static方法是类中被static修饰的方法，它们是属于类本身的方法，而不是实例方法。由于代理是基于实例的，无法为Static方法生成代理。
- 匿名内部类： 匿名内部类是在声明的同时实例化的类，由于其没有明确的类名，无法生成代理类。

需要注意的是，以上列出的情况是指Java的标准代理方式，即基于接口或基于类的代理（如JDK动态代理、CGLib代理）。在某些情况下，可以使用字节码操作库（如Byte Buddy、ASM）进行字节码级别的代理，以绕过上述限制，但这涉及到更底层的操作和更复杂的实现。

138、 要在 Spring IoC 容器构建完毕之后执行一些逻辑，怎么实现

在Spring中，如果您想在IoC容器构建完毕后执行一些逻辑，可以通过实现`ApplicationListener`接口或使用`@EventListener`注解来实现。

方法1：通过`ApplicationListener`接口

1. 创建一个类，实现`ApplicationListener<ContextRefreshedEvent>`接口。
2. 实现`onApplicationEvent(ContextRefreshedEvent event)`方法，在该方法中编写您想要执行的逻辑。

Java | 复制代码

```java
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.stereotype.Component;

@Component
public class MyApplicationListener implements ApplicationListener<ContextRefreshedEvent> {

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        // 在容器构建完毕后执行逻辑
        System.out.println("Spring容器构建完毕，可以执行一些逻辑了。");
    }
}
```

方法2：通过`@EventListener`注解

1. 在您的监听类上添加`@Component`注解，将其纳入Spring容器管理。
2. 创建一个方法，添加`@EventListener`注解，并指定事件类型为`ContextRefreshedEvent`，在该方法中编写您想要执行的逻辑。

```java
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.context.event.EventListener;
import org.springframework.stereotype.Component;

@Component
public class MyEventListener {

    @EventListener
    public void handleContextRefreshedEvent(ContextRefreshedEvent event) {
        // 在容器构建完毕后执行逻辑
        System.out.println("Spring容器构建完毕，可以执行一些逻辑了。");
    }
}
```

这样，在Spring IoC容器构建完成后，就会自动触发 `ContextRefreshedEvent` 事件，然后会执行 `ApplicationListener` 或 `@EventListener` 所指定的方法，并在其中执行您的逻辑。

139、要在 Spring IoC 容器构建完毕之后执行一些逻辑，怎么实现

1）@PostConstruct修饰的方法里面的注入的对象都是初始化成功的对象

2）ContextRefreshedEvent事件表示Spring容器已经初始化完成。通过实现ApplicationListener<ContextRefreshedEvent>接口，可以监听该事件并在Spring容器初始化之后执行相应的逻辑

3）CommandLineRunner、ApplicationRunner 接口是在容器启动成功后的最后一步回调

4）InitializingBean接口定义了一个afterPropertiesSet方法，它在Bean的属性设置完成后被调用。通过实现这个接口，可以在Spring容器初始化之后执行自定义的逻辑

140、 @Component注解所在的类也需要加入到spring容器中，那为何可以在容器初始化之后做一些事情呢

您的观察是正确的。`@Component`注解所标记的类也需要被纳入到Spring容器中进行管理。因此，您可能会疑惑为什么在`@Component`注解所在的类中的方法可以在容器初始化之后执行。

答案是，当Spring容器扫描并检测到一个被`@Component`注解标记的类时，它会创建该类的实例并将其纳入到容器管理中。在容器初始化过程中，会执行一系列的步骤，包括实例化Bean、依赖注入和

初始化等。在这个过程中，被`@Component`注解标记的类会被实例化和初始化，但是其中的方法不会立即被调用。

在Spring容器初始化过程完成后，容器会触发一些特定的事件，比如`ContextRefreshedEvent`事件，表示容器已经初始化完成。这时，容器会寻找所有实现了`ApplicationListener`接口的类，并调用其相应的事件处理方法。同样，使用`@EventListener`注解标记的方法也会被容器发现并执行。

因此，虽然`@Component`注解所在的类会在容器初始化期间被创建和初始化，但是类中的方法在容器初始化过程中不会立即调用。方法的调用通常会在容器初始化完成后，通过事件监听或其他方式进行触发。

总结起来，`@Component`注解所在的类需要在容器中进行管理，而容器初始化完成后执行的逻辑可以通过事件监听或其他机制触发并调用类中的方法。

141、 MySQL 如何锁住一行数据

MySQL提供了多种方式来锁住一行数据，以下是其中两种常用的方法：

1. 使用SELECT ... FOR UPDATE语句：
   可以使用SELECT ... FOR UPDATE语句来锁住一行数据。该语句会查询指定的数据，并将其行级锁定，确保其他事务无法修改或删除该行数据，直到当前事务提交或回滚。
   示例：

SQL 复制代码

```sql
START TRANSACTION;
SELECT * FROM table_name WHERE id = 1 FOR UPDATE;
-- 在事务中对数据进行操作
COMMIT;
```

上述示例中，通过在SELECT语句后添加FOR UPDATE子句来锁住id为1的行数据。其他事务如果尝试修改或删除该行数据，将会被阻塞等待。

2. 使用排他锁（Exclusive Lock）：
   另一种方式是使用排他锁来锁住一行数据。可以在事务中使用 `SELECT ... FOR UPDATE` 语句获取排他锁，也可以使用 `LOCK TABLES` 语句对表进行锁定。
   示例：

```sql
START TRANSACTION;
SELECT * FROM table_name WHERE id = 1 LOCK IN SHARE MODE;
-- 在事务中对数据进行操作
COMMIT;
```

上述示例中，使用 `LOCK IN SHARE MODE` 子句对id为1的行数据获取了排他锁。其他事务如果尝试获取该行数据的排他锁或修改该行数据，将会被阻塞等待。

请注意，锁住一行数据需要在事务中进行操作，并在事务结束时释放锁。同时，需要注意锁的粒度，以避免锁定过多的数据行导致性能问题或死锁。在使用锁时，应根据具体的业务需求和并发情况进行权衡和优化。

142、Spring 中的常见扩展点有哪些

在Spring框架中，有许多扩展点可以用来自定义和扩展应用程序的行为。以下是一些常见的Spring扩展点：

1. BeanPostProcessor：允许在容器实例化和配置Bean之后以及在销毁Bean之前进行自定义操作。
2. BeanFactoryPostProcessor：在容器实例化Bean之前对BeanFactory进行自定义操作，如修改Bean定义或注册新的Bean定义。
3. BeanFactoryAware和BeanPostProcessor：这些接口可以用来获取对BeanFactory的引用或修改BeanFactory的行为。
4. ApplicationContextInitializer：在ApplicationContext刷新之前初始化ApplicationContext的回调接口，可以进行一些自定义的初始化操作。
5. ApplicationListener：用于监听应用程序事件的接口，可以处理应用程序中的各种事件，如上下文刷新、Bean初始化等。
6. EnvironmentPostProcessor：用于在应用程序上下文刷新之前自定义修改Environment的策略。
7. ResourceLoaderAware：用于获取对ResourceLoader的引用，可以用于加载资源文件。
8. BeanDefinitionRegistryPostProcessor：用于在Bean定义注册之后对BeanDefinitionRegistry进行进一步的自定义操作，如动态注册Bean定义。
9. HandlerInterceptor：用于在请求处理过程中拦截和处理请求的接口，可以用来进行权限验证、日志记录等操作。
10. ConversionService：用于类型转换的接口，可以自定义类型转换器和类型转换规则。

这些扩展点可以通过实现相应的接口或者使用注解来进行自定义和扩展，从而实现对Spring框架的个性化定制和功能增强。通过利用这些扩展点，可以更好地适应不同的业务需求和场景，提高应用程序的灵

活性和可扩展性。

143、MySQL 的可重复读是怎么实现的

MySQL的可重复读是通过多版本并发控制（MVCC）机制来实现的。

在可重复读隔离级别下，MySQL使用了以下机制来实现数据的一致性和隔离性：

1. 快照读：当一个事务开始时，MySQL会为该事务创建一个快照，该快照代表了事务开始时数据库中数据的一个静态视图。事务在执行过程中只能看到该快照中的数据，而不会受到其他并发事务的影响。
2. 行级锁：MySQL使用行级锁来控制并发访问。事务在更新数据时会对涉及的行进行加锁，其他事务无法同时修改被锁定的行。这样可以保证在可重复读隔离级别下，一个事务读取的数据不会被其他事务修改。
3. Undo日志：当一个事务更新数据时，MySQL会在事务开始时记录一份数据的副本到undo日志中。如果其他事务需要读取被当前事务修改的数据，MySQL会从undo日志中恢复原始数据，保证读取的是事务开始时的数据快照。

通过使用快照读、行级锁和Undo日志，MySQL实现了可重复读隔离级别。这样即使其他并发事务修改了数据，当前事务仍然能够读取到一致的数据视图，保证了数据的一致性和隔离性。

144、MySQL 是否会出现幻读

是的，MySQL在某些情况下可能会出现幻读。

幻读是指在一个事务中，前后两次相同的查询语句返回了不同的结果，通常是由于其他事务对数据进行了插入、删除或更新操作导致的。

在可重复读隔离级别下，MySQL使用了MVCC机制来避免幻读的发生。MVCC通过在每一行数据中保存多个版本来实现。当一个事务开始时，MySQL会为该事务创建一个快照，并在该快照中记录事务开始时数据库中每一行数据的版本号。在事务执行过程中，只能看到事务开始时的快照中的数据，不受其他事务的修改影响。

然而，当其他事务在当前事务执行过程中插入或删除了符合当前事务查询条件的数据时，就会导致幻读的发生。因为根据事务开始时的快照，当前事务认为该行数据不存在或存在多次，与当前事务的查询结果不一致。

为了避免幻读，MySQL提供了更高级别的隔离级别，如可串行化隔离级别。在可串行化隔离级别下，MySQL使用了间隙锁（Gap Lock）机制，锁住了事务范围内的间隙，防止其他事务插入或删除符合查询条件的数据，从而避免幻读的发生。

需要注意的是，虽然可串行化隔离级别可以避免幻读，但也可能引入更多的锁冲突和性能开销，因此在实际应用中需要根据具体需求选择合适的隔离级别。

145、MySQL 的 gap 锁

MySQL的间隙锁（Gap Lock）是一种锁机制，用于在事务中锁定一个范围内的间隙（gap），防止其他事务在该范围内插入或删除数据。间隙锁可以避免幻读的发生。

具体来说，间隙锁是在索引上的间隙（两个索引记录之间的空白区域）上设置的，而不是在实际的数据记录上设置的。当事务使用SELECT ... FOR UPDATE或SELECT ... LOCK IN SHARE MODE语句时，MySQL会在满足查询条件的索引范围内设置间隙锁，阻止其他事务在该范围内插入或删除数据。

间隙锁的作用是保护查询结果的一致性。当一个事务使用间隙锁锁定了一个范围时，其他事务无法在该范围内插入或删除数据，从而确保了事务在查询过程中返回的数据的一致性。这样就避免了幻读的问题，即在同一事务中两次相同的查询返回了不同的结果。

需要注意的是，间隙锁的范围是基于索引的，而不是基于实际的数据记录。因此，当使用间隙锁时，需要确保查询条件涵盖了间隙范围，以便锁定正确的间隙。

间隙锁在MySQL的隔离级别为可串行化（SERIALIZABLE）时生效，默认情况下是开启的。但是，间隙锁会增加并发操作的开销，可能导致性能下降。因此，在实际应用中，需要权衡数据一致性和性能，并根据具体场景选择合适的隔离级别和锁机制。

146、MySQL 是否会出现幻读

是的，MySQL在某些情况下可能会出现幻读。

幻读是指在一个事务中，前后两次相同的查询语句返回了不同的结果，通常是由于其他事务对数据进行了插入、删除或更新操作导致的。

在可重复读隔离级别下，MySQL使用了MVCC机制来避免幻读的发生。MVCC通过在每一行数据中保存多个版本来实现。当一个事务开始时，MySQL会为该事务创建一个快照，并在该快照中记录事务开始时数据库中每一行数据的版本号。在事务执行过程中，只能看到事务开始时的快照中的数据，不受其他事务的修改影响。

然而，当其他事务在当前事务执行过程中插入或删除了符合当前事务查询条件的数据时，就会导致幻读的发生。因为根据事务开始时的快照，当前事务认为该行数据不存在或存在多次，与当前事务的查询结果不一致。

为了避免幻读，MySQL提供了更高级别的隔离级别，如可串行化隔离级别。在可串行化隔离级别下，MySQL使用了间隙锁（Gap Lock）机制，锁住了事务范围内的间隙，防止其他事务插入或删除符合查询

条件的数据，从而避免幻读的发生。

需要注意的是，虽然可串行化隔离级别可以避免幻读，但也可能引入更多的锁冲突和性能开销，因此在实际应用中需要根据具体需求选择合适的隔离级别。

147、MySQL 的 gap 锁

MySQL的主从同步是指将一个MySQL数据库实例（主库）的数据变动同步到其他MySQL数据库实例（从库）的过程。主从同步实现了数据的复制和备份，提高了系统的可用性和可靠性。

MySQL的主从同步原理如下：

1. 主库二进制日志（Binary Log）：主库将所有的数据变动操作（如插入、更新、删除）以二进制日志的形式记录下来，形成一个逐步增长的日志文件。
2. 从库复制线程：从库启动一个复制线程，连接到主库，并请求从主库获取二进制日志的内容。
3. 主库传输二进制日志：主库将二进制日志的内容传输给从库，通常是通过网络传输。
4. 从库重放日志：从库接收到二进制日志后，将日志内容重放到自己的数据库中，执行相同的数据变动操作，使得从库的数据与主库保持一致。
5. 从库追赶主库：从库会不断地拉取主库的二进制日志并应用，以保持与主库的数据同步。从库可以根据自身的情况设置拉取的频率，可以是实时或者定期。

需要注意的是，主从同步是异步的过程，主库和从库之间有一定的延迟。因此，在主从同步环境下，可能会存在主从数据的不一致性，但通常这个延迟是可以接受的。

主从同步的应用场景包括：

- 数据备份和灾难恢复：通过从库可以实现数据的备份，当主库出现故障时，可以快速切换到从库继续提供服务。
- 读写分离：通过将读操作分发到从库，可以提升系统的读取性能，减轻主库的负载压力。
- 数据分析和报表生成：从库可以用于执行数据分析任务，以免对主库的查询操作造成影响。

主从同步在MySQL中是一种常见且重要的数据库复制机制，可以提高系统的可用性、可靠性和性能。

148、分库分表的实现方案

分库分表是一种数据库水平拆分的策略，用于解决单一数据库的容量和性能限制。以下是几种常见的分库分表实现方案：

1. 垂直拆分：将一个大型数据库按照业务功能划分成多个独立的数据库，每个数据库包含部分表和相关的业务数据。例如，可以将用户表、订单表、商品表等分别放在不同的数据库中。这种方式适合于业务功能耦合度低的场景。

2. 水平拆分：将一个大型表按照某个关键字段（如用户ID、订单ID等）的取值范围划分成多个子表，每个子表存储一部分数据。例如，可以按照用户ID的哈希值进行取模操作，将数据均匀地分散到多个表中。这种方式适合于数据量较大且访问频率分散的场景。
3. 分区表：将一个大型表按照某个时间范围或其他条件进行分区，将数据分散存储在多个物理表中。例如，可以按照订单的创建时间将数据按月或按年进行分区。这种方式可以提高查询性能和数据管理的灵活性。
4. 数据库分片：将整个数据库集群分成多个独立的数据库实例，每个实例负责存储部分数据。通常使用分片键（如用户ID）将数据划分到不同的数据库实例中。这种方式可以水平扩展数据库容量和处理能力，但增加了数据一致性和事务管理的复杂性。

在实现分库分表时，需要考虑数据的分布均衡、查询路由、事务处理、数据迁移等问题，并选择适合业务需求和系统架构的分库分表方案。同时，需要借助数据库中间件或分布式数据库来提供数据访问和管理的抽象层，简化应用程序的开发和维护。一些常用的数据库中间件包括MySQL的MyCAT、OceanBase，以及阿里巴巴的TDDL、微软的Azure SQL Database等。

149、explain 中每个字段的意思

在MySQL中，EXPLAIN语句用于解释查询执行计划，并提供有关查询优化器如何执行查询的信息。EXPLAIN语句返回一组行，每一行对应查询执行计划中的一个步骤，每个字段提供了与该步骤相关的信息。下面是EXPLAIN语句中常见字段的含义：

1. id: 查询执行计划中每个步骤的唯一标识符，从大到小按顺序递减，表示查询的执行顺序。
2. select_type: 表示查询的类型，例如SIMPLE（简单查询）、PRIMARY（主查询）、SUBQUERY（子查询）等。
3. table: 表示查询操作涉及的表名。
4. partitions: 如果查询涉及到了分区表，该字段表示查询涉及的分区。
5. type: 表示访问表的方式，常见的取值包括：const（使用常量进行查询）、eq_ref（使用唯一索引进行等值连接）、ref（使用非唯一索引进行查询）、range（使用索引范围进行查询）、index（全表扫描索引）等。
6. possible_keys: 表示在该步骤中可能使用的索引列表。
7. key: 表示在该步骤中实际使用的索引。
8. key_len: 表示在该步骤中使用的索引长度。
9. ref: 表示在该步骤中使用的索引列或常量。
10. rows: 表示在该步骤中扫描的行数，即估计的查询返回的行数。
11. filtered: 表示该步骤扫描的行数在返回结果中的比例。
12. Extra: 提供额外的执行信息，如Using index（表示使用了覆盖索引）、Using temporary（表示使

用了临时表）、Using filesort（表示需要进行排序）等。

通过分析EXPLAIN语句的输出，可以了解查询的执行计划、表的访问方式、索引使用情况、数据扫描行数等信息，帮助优化查询性能和索引设计。

150、explain 中的 type 字段有哪些常见的值

在MySQL的EXPLAIN语句中，type字段表示访问表的方式，也称为访问类型。下面是常见的type字段取值及其含义：

1. const：使用常量进行查询，表示通过索引或常量条件只匹配一行数据。
2. eq_ref：使用唯一索引进行等值连接，表示通过唯一索引将两个表连接，通常在连接条件中使用主键或唯一索引。
3. ref：使用非唯一索引进行查询，表示通过非唯一索引进行查找，返回匹配的多行数据。
4. range：使用索引范围进行查询，表示通过索引范围扫描，返回指定范围内的数据。
5. index：全表扫描索引，表示全表扫描使用索引来遍历数据。
6. all：全表扫描，表示对整个表进行全表扫描，无需使用索引。
7. system：表示只有一行数据，这是const类型的特例。
8. NULL：表示无法使用索引或查询。

这些type字段的取值代表了不同的访问方式，其中使用索引的访问方式通常比全表扫描更高效，因为它可以减少磁盘I/O和数据的扫描量，提高查询性能。优化查询时，需要关注type字段的取值，确保选择合适的索引和查询方式来提高查询效率。

151、explain 中你通常关注哪些字段，为什么

在执行MySQL的EXPLAIN语句时，我通常会关注以下几个字段：

1. type：表示访问表的方式，这对于查询性能优化非常重要。我会检查type字段的取值，尽量选择能够使用索引的访问方式，避免全表扫描。
2. key：表示查询中使用的索引。我会检查是否使用了合适的索引，以及是否存在缺失的索引。缺失的索引可能导致查询性能较差。
3. rows：表示估计的扫描行数，即查询需要处理的行数。我会关注该值，如果估计的行数过大，可能需要优化查询或增加合适的索引。
4. Extra：表示额外的信息，例如使用了临时表、文件排序等。我会注意这些额外的操作是否对性能产生了负面影响。

通过关注这些字段，我可以评估查询的性能瓶颈，判断是否需要进行索引优化、查询重写或其他调整，以提高查询性能和效率。

152、运行时数据区

运行时数据区是指Java虚拟机在运行Java程序时所使用的内存区域，主要包括以下几个区域：

1. 方法区（Method Area）：用于存储类的结构信息，包括类的字段、方法信息、常量池等。在Java 8及之前的版本中，方法区也被称为永久代（Permanent Generation），而在Java 8及之后的版本中，被元数据区（Metaspace）所取代。
2. 堆（Heap）：用于存储对象实例和数组。所有通过new关键字创建的对象都会在堆中分配内存。堆是Java虚拟机管理的最大一块内存区域，被所有线程共享。
3. 虚拟机栈（VM Stack）：每个线程在运行时都会创建一个虚拟机栈，用于存储方法调用的局部变量、方法参数、返回值等信息。每个方法在执行时会创建一个栈帧，栈帧包含了方法的局部变量表、操作数栈、动态链接等信息。
4. 本地方法栈（Native Method Stack）：与虚拟机栈类似，但是用于执行Native方法（即使用非Java语言实现的方法）。
5. 程序计数器（Program Counter）：记录当前线程所执行的字节码指令的地址或索引。每个线程都有自己的程序计数器，用于线程切换后能够恢复到正确的执行位置。

除了以上几个主要的运行时数据区，还有一些其他的辅助数据区，如直接内存（用于NIO的ByteBuffer分配）、运行时常量池（在方法区或元数据区中存储运行时常量）、线程私有数据区等。

不同的运行时数据区在Java虚拟机中有不同的作用和使用方式，对于程序的执行和内存管理都起着重要的作用。

153、哪些场景需要打破双亲委派模式

在某些场景下，可能需要打破双亲委派模式，这通常发生在以下情况下：

1. 自定义类加载器：当需要自定义类加载器加载特定的类或资源时，可以打破双亲委派模式。自定义类加载器可以继承ClassLoader类，并覆写其中的加载方法，自行定义类加载的逻辑。
2. 动态更新类：在热部署或插件化开发中，可能需要动态更新已加载的类。为了避免双亲委派模式下父类加载器无法加载新版本的类，可以使用自定义的类加载器实现类的热替换。
3. 模块化开发：在模块化开发中，可能需要隔离不同模块的类加载环境，使其互不干扰。这时可以使用自定义的类加载器，每个模块使用独立的类加载器加载类，从而实现模块之间的隔离。

需要注意的是，打破双亲委派模式需要谨慎操作，并且了解其潜在的风险和副作用。在一般情况下，建议遵循双亲委派模式，利用好Java虚拟机的类加载机制，保证类加载的安全性和一致性。只有在特定的需求和场景下，才考虑打破双亲委派模式。

154、线上服务器出现频繁 Full GC，怎么排查

频繁的 Full GC 可能是系统性能或应用程序的问题。下面是一些排查频繁 Full GC 的常见步骤：

1. 监控工具：使用监控工具（如JVM监控工具、性能分析工具）来观察系统的内存使用情况、GC日志等信息，以确定是否存在内存问题。
2. 分析GC日志：查看GC日志，了解GC的频率、持续时间以及GC过程中的各种指标。特别关注Full GC的原因，如老年代空间不足、永久代空间不足等。
3. 内存泄漏检查：检查应用程序是否存在内存泄漏，即有大量对象无法被回收，导致内存占用逐渐增加。可以使用内存分析工具（如Eclipse Memory Analyzer）来分析堆内存快照，查找潜在的内存泄漏问题。
4. 内存设置：检查JVM的内存设置，包括堆内存大小、永久代大小等。如果堆内存设置过小，可能导致频繁的Full GC。可以根据应用程序的实际需求适当调整内存设置。
5. 代码优化：检查应用程序的代码，查找可能导致内存消耗过大或对象生命周期过长的问题。例如，及时释放不再使用的资源、避免创建过多临时对象等。
6. GC调优：根据实际情况调整GC算法和参数，以改善GC的性能。可以尝试调整新生代和老年代的比例、调整GC的触发条件、调整GC的线程数等。
7. 硬件资源：确保服务器的硬件资源足够满足应用程序的需求，包括CPU、内存、磁盘等。

以上是一些常见的排查频繁 Full GC 的方法，具体的排查步骤可能因系统环境和应用程序而异。在排查问题时，建议结合具体的场景和实际情况，综合分析并采取适当的措施。如有需要，也可以寻求专业的Java性能优化工程师的帮助。

155、定位问题常用哪些命令

在定位问题时，以下是一些常用的命令和工具：

1. top：用于监视系统的实时性能，包括CPU使用率、内存使用率、进程状态等。
2. ps：用于查看当前系统中运行的进程信息，如进程ID、CPU使用率、内存占用等。
3. netstat：用于查看网络连接和网络统计信息，包括当前打开的端口、连接状态等。
4. ping：用于检查网络连接是否正常，发送 ICMP 回显请求并接收回显应答。
5. traceroute（或 tracert）：用于跟踪数据包在网络中的路径，显示数据包从源到目的地的经过的路

由器。

6. lsof：用于查看系统打开的文件，包括进程所打开的文件、网络连接等。
7. strace：用于跟踪进程的系统调用和信号传递，可以查看进程的系统调用过程和返回结果。
8. jstack：用于生成 Java 进程的线程转储信息，可以用于分析线程状态、死锁等问题。
9. jstat：用于监视和收集 Java 虚拟机（JVM）的各种统计信息，包括堆内存使用、垃圾回收情况等。
10. jmap：用于生成 Java 进程的堆转储快照，可以用于分析内存泄漏、对象占用等问题。
11. tcpdump：用于抓取网络数据包，可以查看网络数据包的内容和传输情况。
12. Wireshark：网络分析工具，可以用于捕获和分析网络数据包。

以上是一些常用的命令和工具，可以根据具体的问题和需求选择合适的工具进行定位和分析。在使用这些命令和工具时，建议查阅其详细文档和使用说明，以便正确理解和使用。

156、介绍下 JVM 调优的过程

JVM（Java虚拟机）调优是通过优化Java应用程序在JVM上的执行来提高性能和资源利用率的过程。下面是JVM调优的一般过程：

1. 目标定义：首先需要明确调优的目标，例如提高应用程序的吞吐量、降低内存占用、减少垃圾回收等。
2. 监控和分析：使用性能分析工具（如JVM自带的JConsole、VisualVM，或者第三方工具如YourKit、JProfiler等）对应用程序进行监控和分析。收集关键指标，如CPU使用率、内存使用情况、垃圾回收行为等。
3. 定位性能瓶颈：根据监控数据和分析结果，确定应用程序中的性能瓶颈所在。可能的性能瓶颈包括CPU密集型计算、内存泄漏、频繁的垃圾回收等。
4. 代码优化：针对性能瓶颈进行代码优化，例如使用更高效的算法、减少内存占用、避免频繁的对象创建和销毁等。优化的方式可以涉及代码重构、使用合适的数据结构、避免不必要的计算等。
5. JVM参数调优：根据应用程序的需求和性能特点，调整JVM的参数。例如调整堆内存大小、设置垃圾回收器类型和参数、设置线程池大小等。这些参数的调整可以通过JVM启动参数或运行时参数进行设置。
6. 垃圾回收调优：根据监控数据和分析结果，优化垃圾回收行为。可以选择合适的垃圾回收器，调整垃圾回收器的参数，例如堆大小、垃圾回收阈值等。还可以通过对象的生命周期管理来减少垃圾回收的频率和开销。
7. 测试和验证：在调优过程中进行性能测试和验证，以确保优化后的应用程序达到预期的性能目标。可以使用负载测试工具模拟真实的应用负载，并进行性能指标的测量和对比分析。
8. 反馈和迭代：根据测试和验证的结果，收集反馈意见，并根据需要进行进一步的调优和优化。不断

迭代优化过程，逐步改进应用程序的性能和效率。

需要注意的是，JVM调优是一个复杂的过程，涉及到多个因素和调整参数的相互影响。在进行调优时，需要综合考虑应用程序的特点、硬件环境、负载情况等因素，并进行充分的测试和验证，以确保调优的有效性和稳定性。

157、Redis 集群要增加分片，槽的迁移怎么保证无损

在 Redis 集群中增加分片时，槽的迁移是必要的，以保证数据在新的分片上的均衡分布。为了保证槽迁移的过程中数据的无损和服务的可用性，Redis 提供了以下机制：

1. **哈希槽（Hash Slot）划分：** Redis 将数据分散存储在不同的哈希槽中，一个 Redis 集群共有 16384 个哈希槽。槽的迁移是基于这个划分的，通过将槽从一个节点迁移到另一个节点来实现分片的增加或减少。
2. **无损迁移原则：** Redis 集群在进行槽迁移时，会先将新节点加入集群，并进行预迁移准备工作。然后，槽的迁移会逐个进行，每个槽会依次从旧节点迁移到新节点。在迁移过程中，Redis 会保证数据的一致性，即同一个槽中的所有键值对要么都在旧节点上，要么都在新节点上，从而实现无损迁移。
3. **迁移状态的同步：** 在槽迁移期间，Redis 集群会使用 Gossip 协议来进行节点之间的信息同步。节点会互相通信，共享槽迁移的状态信息，确保集群中的所有节点都了解迁移过程，并根据需要进行相应的操作。
4. **数据同步机制：** 在槽迁移期间，旧节点会将属于迁移槽的数据转发到新节点，以保证数据的完整性。这样，当槽迁移完成后，新节点会拥有旧节点上的所有数据，保证数据的无损迁移。

需要注意的是，槽迁移是一个相对复杂的操作，对于大量数据和高并发场景，槽迁移可能会对集群的性能产生一定的影响。因此，在进行槽迁移时，建议采取合理的分批策略，避免一次性迁移过多的槽，以保证集群的稳定性和可用性。

总结起来，Redis 通过哈希槽划分、无损迁移原则、迁移状态同步和数据同步机制等方式，保证在集群增加分片时槽的迁移过程中数据的无损和服务的可用性。

158、Redis 的 Hash 对象底层结构

非常抱歉我之前的回答遗漏了 Redis Hash 对象的底层结构之一，即 ziplist（压缩列表）。

在 Redis 中，Hash 对象的底层结构可以是 ziplist 或者 hashtable，具体的选择取决于一些条件和配置。

1. **ziplist（压缩列表）：** ziplist 是 Redis 中的一种紧凑型数据结构，用于存储较小的哈希对象。它是一个连续的内存块，可以按照键值对的顺序存储多个字段和对应的值。ziplist 使用紧凑的编码方

式，可以节省内存空间。然而，由于其线性访问的特性，当字段数量较大或字段长度较长时，ziplist 的性能会受到影响。

2. **hashtable（哈希表）：** hashtable 是 Redis 中更通用的哈希对象底层结构。它采用哈希桶和链表或跳表来解决哈希冲突，并支持更高效的键值查找和操作。hashtable 适用于存储较大的哈希对象，它的性能在大多数情况下更稳定，但占用的内存空间相对较大。

Redis 在选择 ziplist 还是 hashtable 作为 Hash 对象的底层结构时，会根据一些因素进行自动选择。例如，当哈希对象的字段数量较少且字段长度较短时，Redis 倾向于使用 ziplist 来节省内存。而当字段数量较多或字段长度较长时，Redis 会选择 hashtable 以保证性能和稳定性。

需要注意的是，具体的选择和配置可能因不同的 Redis 版本和配置而有所不同。为了确切了解某个特定版本的 Redis 中的 Hash 对象底层结构选择策略，建议查阅对应版本的 Redis 官方文档或了解相关源代码实现。

159、Redis 中 Hash 对象的扩容流程

在 Redis 中，当 Hash 对象中的键值对数量超过一定阈值时，会触发 Hash 对象的扩容操作，以适应更多的键值对存储需求。下面是 Redis 中 Hash 对象的扩容流程：

1. **创建新的哈希表：** 当 Hash 对象需要扩容时，Redis 会创建一个更大的新哈希表，通常是当前哈希表大小的两倍。
2. **将键值对迁移至新哈希表：** Redis 会遍历旧哈希表中的所有键值对，并使用哈希函数将它们映射到新哈希表的对应位置。这个过程中，键值对会从旧哈希表中复制到新哈希表中。
3. **替换旧哈希表：** 在将键值对迁移至新哈希表后，Redis 会将旧哈希表替换为新哈希表，使得新哈希表成为 Hash 对象的底层数据结构。
4. **继续使用新哈希表：** 扩容完成后，Hash 对象会开始使用新的哈希表进行键值对的存储和查找操作。

需要注意的是，在扩容过程中，Redis 会使用渐进式 rehash 的方式进行，以避免扩容操作对系统性能的影响。具体而言，Redis 会将扩容操作分成多个步骤，每次只迁移一部分的键值对，而不是一次性将所有键值对迁移完毕。这样可以将扩容的负载均匀分散，减少对系统的影响。

在 rehash 过程中，旧哈希表和新哈希表会共存一段时间，直到旧哈希表中的所有键值对都被迁移到新哈希表中。期间，Redis 会通过哈希表的渐进式 rehash 机制来确保对 Hash 对象的读写操作正常进行。

总结起来，Redis 中的 Hash 对象扩容流程包括创建新哈希表、迁移键值对、替换旧哈希表和继续使用新哈希表。通过渐进式 rehash 的方式，Redis 在扩容过程中保证对系统性能的影响最小化。

160、Redis 的 Hash 对象的扩容流程在数据量大的时候会有什么问题吗

在 Redis 的 Hash 对象进行扩容时，当数据量大时可能会面临以下问题：

1. **内存占用：** 在扩容过程中，需要同时维护旧哈希表和新哈希表，这意味着需要额外的内存空间来存储两个哈希表的键值对。在数据量非常大的情况下，这可能会导致内存占用的显著增加，可能超过系统可用的内存资源。
2. **延迟增加：** 当数据量庞大时，迁移大量的键值对需要花费相当长的时间。在这个过程中，哈希表的扩容可能会导致系统的延迟增加。读写操作可能会受到影响，因为需要同时处理两个哈希表，并且在迁移期间可能会发生额外的哈希冲突和数据复制操作。
3. **网络带宽压力：** 在扩容过程中，键值对的迁移需要从旧哈希表传输到新哈希表。对于大型数据集，这可能会产生大量的网络数据传输，增加了网络带宽的负载。
4. **迁移失败的风险：** 扩容是一个复杂的操作，涉及到数据的迁移和重新分布。在极端情况下，例如网络故障或服务器崩溃，迁移过程可能会中断或失败。这可能导致数据不一致或丢失的风险。

为了应对这些问题，可以采取以下措施：

1. **合理规划：** 在设计和部署 Redis 时，应根据数据量的预估和系统资源情况进行合理规划，包括合理的硬件配置、内存分配和扩容策略。
2. **分批处理：** 当数据量非常大时，可以采用分批处理的方式进行扩容，即将数据分成多个较小的批次进行迁移。这可以减少单个扩容操作的压力和影响。
3. **网络优化：** 针对网络带宽压力，可以考虑优化网络连接和配置，如增加带宽、使用高速网络设备等，以提高迁移过程中的数据传输效率。
4. **备份和容灾策略：** 在进行扩容操作之前，应确保有可靠的数据备份和容灾策略。这可以防止在迁移过程中发生数据丢失或不一致的情况下，快速恢复数据。

综上所述，当 Redis 的 Hash 对象进行扩容时，特别是在面对大规模数据量时，需要谨慎规划和执行，以避免对系统性能、内存和网络带宽造成不可控的负面影响。