1. **MySql**
2. **基本查询语句**

**File SORT**

**单路算法：一次磁盘IO,读数据到sort\_buffer，进行排序**

**多路算法：多次磁盘IO**

**sort\_buffer\_siz：缓冲区大小**

**max\_length\_for\_sort\_data：当查询字段大小之和小于它，排序字段不是text|BLOB就会使用单路排序**

**show variables like “sort\_buffer\_size”**

**set global sort\_buffer\_size=**

**慢查询日志开启**

**long\_query\_time:2**

**slow\_query\_log:1**

**slow\_query\_log\_path：””**

**mysqldumpslow:日志分析工具**

**mysqldumpslow --help**

**mysqldumpslow -s r -t 10 xxxx/xxxx/slow.log**

1. **Mysql的索引**

**定义：索引是一个排好序数据结构，目的是提高sql查询速度，查找+排序**

**优点：查找+排序，降低了磁盘IO消耗查询，降低了cpu消耗**

**缺点：索引占存储空间、查找快，但是插入和修改慢，需要经常变更**

**索引分类：单值索引、唯一索引、组合索引**

**基本语句**

**CREATE [unique] index indexname ON mytable(col\_name)**

**ALTER mytable ADD [UNIQUE] INDEX indexname ON (colname)**

**DROP INDEX indexname ON mytable**

**SHOW INDEX FROM table\_name\G**

**B+树 多路查找树?**

**B树节点，index1，index2 ，point1,point2,point3**

**默认主键索引，聚集索引，包含索引和数据，若无主键则mysql自带row\_id**

**Sql慢的原因：sql写的不合理；索引失效；关联join太多表**

**那些需要索引：**

**1、主键索引 2、频繁查询的字段 3、join的字段外键关系索引**

**4、频繁更新的不适合建立索引 5.where中经常用到的建索引**

**6、单值索引、组合索7、排序字段order by 8、统计分组字段group by**

**那些不要建索引：**

1. **记录少 2.经常增删改 3.某个列包含很多重复，枚举字段**

**EXPLAIN 查询计划 + sql**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **id** | **select\_type** | **table** | **type** | **possible\_keys** | **key** | **ref** | **rows** | **extra** |

**Id 语句加载顺序Id相同，顺序加载上到下；id不同，id大先执行**

**select\_type 查询的类型，普通、联合查询、嵌套查询、子查询**

**SIMPLE：简单的select查询，不包括子查询和union**

**PRIMARY :查询中包含任何复杂的子部分。最外层的标记为**

**SUBQUERY：select中或where中包含了子查询**

**DERIVED：from列表子查询标记为DERIVED衍生表执行子查询临时表**

**UNION：若第二个SELECT出现在UNION之后则被标记为UNION若UNION包含在from子查询中，外层SELECT被标记为DERIVED**

**UNION RESULT: 从UNION表获取结果的SELECT**

**type**

**system >const>eq\_ref>ref>range>index()>ALL(全表扫描)**

**system：表只有一条记录，const的特例**

**const：通过一次索引就找到了，用于主键索引，或者唯一索引，只匹配一行数据**

**eq\_ref：唯一性索引扫描，对于每个索引都只有一个记录匹配**

**ref: 非唯一索引扫描、返回匹配某个单值的所有行**

**rang：检索指定范围的行，使用索引选择行，范围between ，>,<**

**index： 全索引扫描**

**ALL：全表扫描**

**possible\_keys 现实可能用到的索引**

**key 实际查询的过程使用索引，null没有使用使用索引**

**理论可能用到索引，实际没有用， 索引失效**

**理论用不到，实际上用到了 索引覆盖了**

**key\_len 索引中使用字节的数**

**ref 索引那一列被使用了，并使用了什么值**

**rows 检索需要查询多少行**

**extra 额外信息**

**Using filesort mysql无法按照索引排序，使用外部排序（最左前缀）**

**Using temp mysql创建临时表，排序order by,group by（最左前缀）**

**Using index select数据列使用覆盖索引避免访问数据行直接从索引取**

**同时using where 使用索引查找**

**没有using where 没有使用索引**

**Using where、Using joinbuffer、Impossi where、Distinct 第一个匹配**

**避免索引失效**

**全值索引**

**最佳左前缀原则**

**不要在索引列上计算函数，类型转换**

**范围右边的索引失效**

**尽量使用覆盖索引**

**!= 和<>索引失效**

**Is null 和 is not null**

**Like中左边的%不走索引，非要用%XXXX%覆盖索引**

**mysql优化**

**1. 原则：尽量使用整型表示字符串**

**存储IP**

**INET\_ATON(str)，address to number**

**INET\_NTOA(number)，number to address**

**2. 原则：定长和非定长数据类型的选择**

**decimal不会损失精度，存储空间会随数据的增大而增大。double占用固定空间，较大数的存储会损失精度。非定长的还有varchar、text**

**定长char，非定长varchar、text（上限65535，其中varchar还会消耗1-3字节记录长度，而text使用额外空间记录长度）**

**4. 原则：尽可能选择小的数据类型和指定短的长度**

**5.原则：尽可能使用 not null**

**非null字段的处理要比null字段的处理高效些！且不需要判断是否为null。**

**6.原则：单表字段不宜过多**

**7. 存储引擎选择**

**InnoDB 支持事物acid，支持外键，支持行锁，适用于并发场景**

**MYISM 表锁，对增加和查询较快 ，并发性差**

**InnoDB只有在通过索引条件检索数据时使用行级锁，否则使用表锁，行锁变表锁**

**File sort的两种方式**

**单路排序：只进行一次磁盘IO，将数据读入sort\_buffer中一次排序**

**多路排序：进行2次磁盘IO，排序**

1. **Mysql存储引擎**

|  |  |  |
| --- | --- | --- |
| **对比** | **MyISAM** | **InnoDB** |
| **主外键** | **no** | **yes** |
| **事务** | **no** | **yes** |
| **行表锁** | **表锁** | **行锁（高并发）** |
| **缓存** | **缓存索引，不缓存数据** | **索引和数据内存要求高、内存会决定性影响性能** |
| **表空间** | **小** | **大** |
| **关注点** | **性能** | **事务** |
| **适合** | **经常插入、查询场景** | **增删改查** |

1. **mysql中的锁**

**锁的类型： 读锁（共享锁）、写锁（排他锁）、间隙锁gap**

**show open tables**

**lock table mytbl1 read, mytbl2 write**

**unlock tables**

**读锁（锁表）： session1自己可以读、不能写、不能读其他表，**

**session2可以读其他，可以读该表，但是写会阻塞**

**写锁（锁表）： session1，可以读写，不能读写其他表**

**session2 不可以读写该枷锁的表，会被阻塞**

**show status like ‘table%’**

**Table\_locks\_immediate 产生表级锁定的次数**

**Table\_locks\_waited 表级锁定等待的次数**

**锁的粒度**：**行锁、表锁**

**事务的ACID，**

**事务的问题， 丢失更新、脏读、不可重复读、幻读**

**事务的隔离级别 读未提交、 读已提交、可重复读、可序列化**

**Show variables like “tx\_isolation” 查看事务隔离级别**

**Set autocommit=0； 默认提交关闭**

**事务内**

1. **读自己的修改，读己之所写，别的事务是读不到你的修改**
2. **事务修改一行，未提交前，其他事务修改同一行会阻塞**
3. **事务修改一行，未提交前，其他事务修改另一行无阻塞**

**无索引行锁失效为表锁**

**索引失效性能下降行锁变表锁**

**间隙锁的危害：范围锁定然后修改，即使没有某个索引也会被锁定，其他的session你不能插入**

**锁定某一行，for update**

**Show status like “innodb\_row\_lock%“**

1. **Redis**

**1.Redis是单线程的为什么速度快**

**redis是基于内存的，内存的读写速度非常快；**

**redis是单线程的，省去了很多上下文切换线程的时间;**

**redis多路复用技术，可以处理并发的连接。非阻塞IO 内部实现采用epoll，采用了epoll+自己实现的简单的事件框架。epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性，绝不在io上浪费一点时间。**

1. Redis的持久化
2. Redis的实现原理
3. Redis的集群，哨兵模式
4. String的最大大小512m，Set的最大个数2的23次方-1
5. ActiveMQ
6. MQ的使用场景解决问题
7. MQ实现的基本原理
8. MQ的消息类型

点对点：队列（queue）

消息只能被一个消费者消费一次

多个消费者消费同一个队列，队列中的消息平均分配

发布订阅：主题（topic）

非持久：发布消息之前先订阅、无订阅的消息非消息丢弃

持久：先订阅，然后离线，发布者上线发布消息，订阅者上线可以获得历史消息

一个消息多个消费者消费

**Text、map，byte、stream、object**

**消息：消息头、消息体、消息扩展**

**消息头：ID、队列/主题、优先级、ttl、是否持久化**

**receive（）同步阻塞**

**lisener 异步**

**消息的可靠性：**

1. 持久 persistence

队列默认持久化、

主题topic的持久化消息，先订阅，订阅后会把所有未消费的消息发送到订阅者

1. 事务 transaction

ACID，MQ的事务

**对于生产者而言事务**  
**如果关闭事务false，，，，只要发送，就到队列之中**

**如果开启事务true，，，，发送、commit手动提交，到队列中**

**如果事务中某个消息失败，可以回滚**

**对于消费者而言事务**

**关闭事务：消息只消费一次**

**开启事务：没有commit，就认为未消费，会重复消费**

1. 签收 acknowldege （消费者层面避免重复消费）

**自动签收 auto**

**手动签收 client 如果开启了，没有手动签收会出现重复消费**

带副本的重复签收

事务级签收

开启事务，开启手动签收，事务commit，不签收，事务commit认为你签收，不会重复消费

开启事务，开启手动签收，事务不commit，签收，会产生重复消费

**只要你开启了事务必须commit**

**4.多节点集群**

broker是一个activeMQ是一个mq的实例

1. Springboot/Spring框架
2. Spring是如何管理bean的，bean的循环依赖
3. Spring的基本原理IOC 和AOP（动态代理的3个方式）的实现
4. SpringBoot为什么可以不需配置
5. Springboot处理异常的注解
6. 拦截器的使用
7. Mybatis框架
8. 基本使用
9. 原理
10. jvm
11. jvm的内存模型
12. jvm的类加载机制
13. **jvm的分为几个部分**

**内存： 堆、栈、程序计数器、本地方法栈、方法区、**

**类加载器**

1. **垃圾回收的几个算法（各个分代收集）**

**判断一个对象是否是垃圾？**

**引用计数法、可达性分析法（GCRoots）**

**引用计数存在循环引用，无法收集的问题。**

**GCRoots是 栈中的局部变量的引用对象、方法区中的静态引用、方法区的常量引用、本地方法栈中的引用**

**复制算法（新生代）、**

**标记清除（老年代）、**

**标记清楚整理/压缩（老年代）**

1. 垃圾回收器的种类
2. jvm的调优参数

标配参数 jps –l , jinfo –flag, java -version

-X参数 -Xint（解释执行） -Xcomp（编译执行） -Xmixed（混合模式）

-XX参数 布尔类型、kv类型

**1.如何查看运行java程序的参数？**

**jps –l**

**jinfo –flag PrintGCDetails java进程id**

**jinfo –flags java进程id**

1. **查看所有默认参数？**

**java -XX:+PrintFlagsInitial**

**java -XX:+PrintFlagsFinal**

**java -XX:+PrintCommandLineFlags –version**

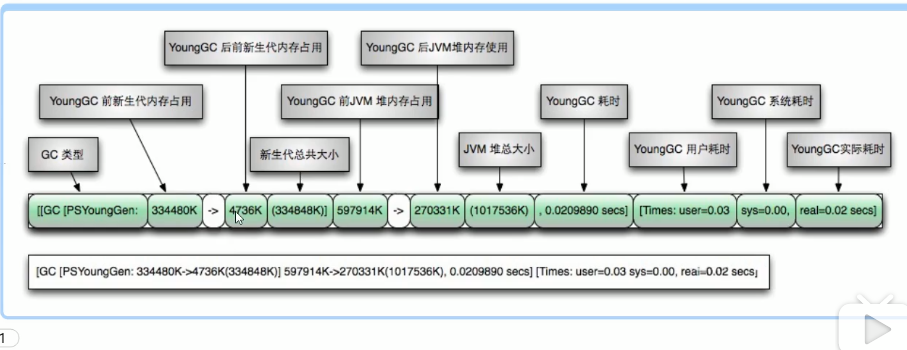
1. **常用参数**

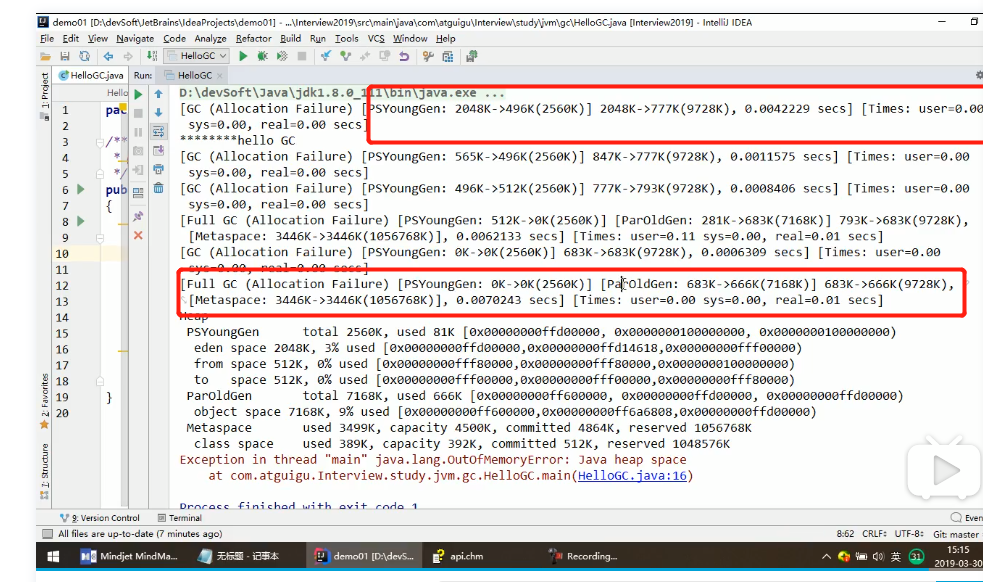
**-Xms(堆1/64)、-Xmx（堆最大1/4）、-Xss（栈大小1024kB）**

**-Xmn 年轻代大小**

**-XX:MetaSpaceSize 元空间大小**

**-XX:+PrintGCDetails 输出GC详细收集日志信息**

****

****

**-XX: SurvivorRatio =4 4;1;1 默认8：1：1 s from to**

**-XX: NewRatio=2新和老的比例1：2**

**-XX:MaxTenuringThreshold 年轻代到老年代，y到o要15次**

强引用：正常new的对象引用，就算OOM,也不会收，经常造成内存泄漏

软引用：系统内存充足不被回收，内存不够回收

SoftReference<T> soft = new SoftReference<>(ob1);

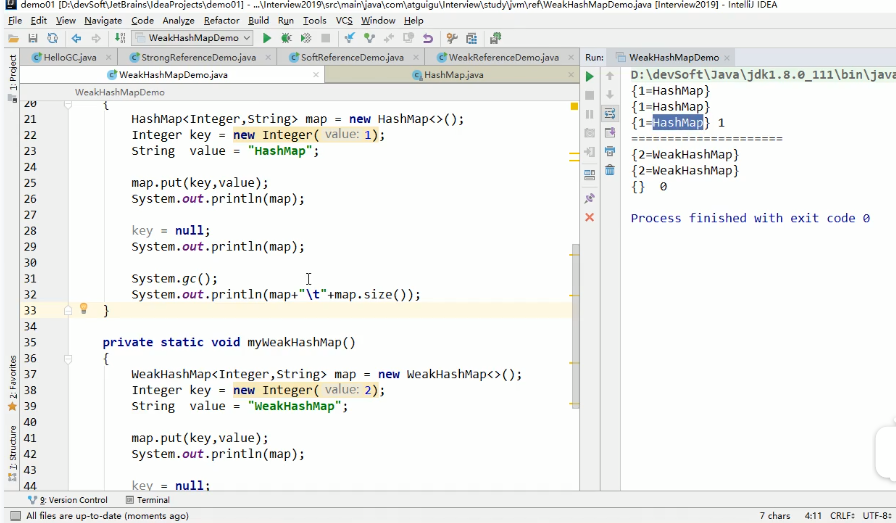
Soft.get()

弱引用：只要有ＧＣ就会被回收

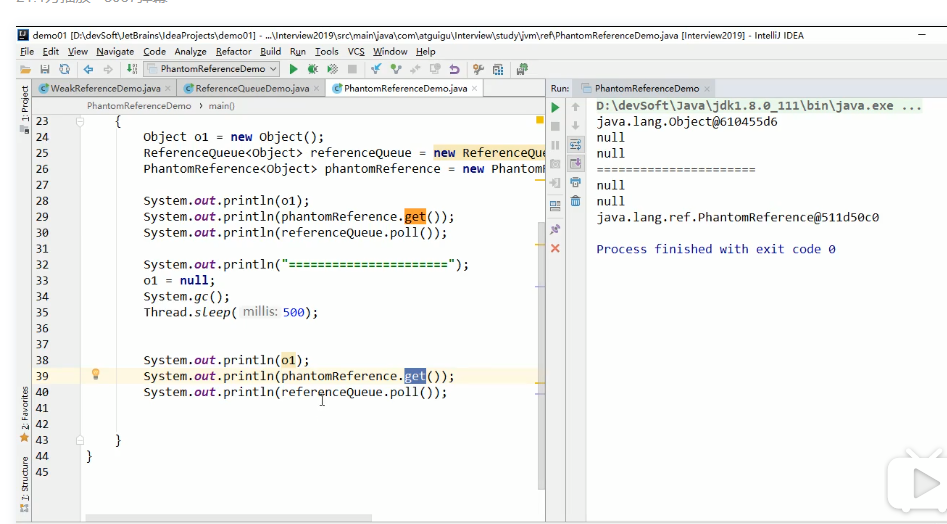
场景：加载图片，软或弱

Map<String ,SoftReference<BitMap>> soft=

WeakHashMap<T>



虚引用：无法get,无法单独使用，必须和ReferenceQueue使用，在回收在回收之前调用finalize方法



StackOverFlowError 递归调用栈溢出

OutofMemoryError：java heap space 堆溢出

OutofMemoryError：GC overhead limit exceeded

大量资源去做GC，但是GC的效果很差，回收了很少的内存

OutofMemoryError: Direct buffer memory

Nio技术导致的，写nio的时候

一使用ByteBuffer.allocate() 属于GC回收

二使用ByteBUffer.allocateDirect() 分配本地内存

OutofMemoryError: unable to cteate native thread

多线程不能继续创建



OutofMemoryError :MetaSpace



虚拟机加载类信息

常量池

静态变量

既是编译后的代码

**垃圾收集器**



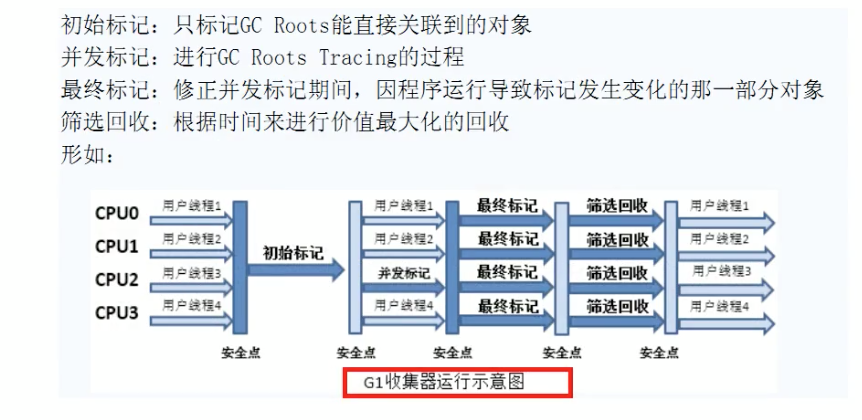


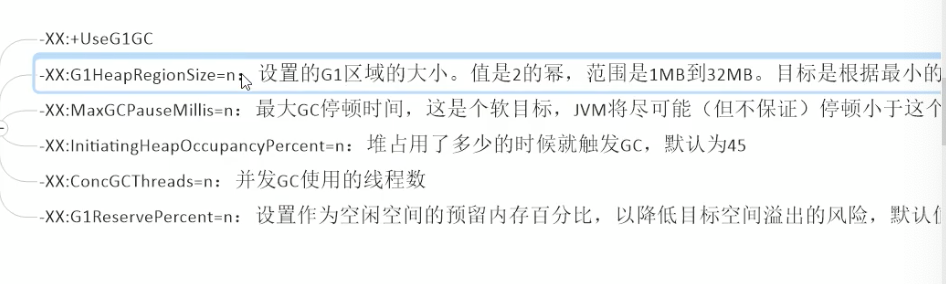
**G1垃圾回收器**

**面向服务端的垃圾收集器-server ，运用在多核，大内存，实现高吞吐量、尽量降低收集暂停时间短，将heap堆内存划为若干个regin块，每个块1m-32m，整体使用标记整理，局部复制**

**1.不会产生内存碎片（cms是并发标记清除，有碎片） 2.能够指定垃圾回收的时间**

**物理上不分新生代老年代，逻辑上分年轻代老年代**





1. 多线程与锁的机制
2. 线程创建的几种方法
3. 线程池使用及其参数
4. 锁的概念，Sychonized 和Lock
5. volatile
6. ThreadLocal
7. HashMap原理
8. 阻塞队列BlockQueue
9. CopyOnWriteArralist / ConcurrentHashMap 底层原理
10. CountDownLatch/。。。。。。
11. 如何选择线程个数
12. 异常处理
13. 异常的种类
14. 异常如何处理
15. 自定义异常
16. RPC
17. RPC的概念及RPC的实现
18. 分布式锁
19. 分布式锁的设计实现
20. IO/NIO/AIO

a.网络编程中的IO

b.网络编程中的NIO

c.网络编程中的AIO

**Springboot中的循环依赖**

**singleTonObjects 一级缓存，存储 单例对象初始化完成**

**singleTonFactories 二级缓存，存储的是创建bean的singleTonFactory,这个singleTonFactory有个getObject方法获取singleTon半成品对象**

**earlySingletonObjects 三级缓存 ，半成品对象**

**step1**

**第一次调用doGetBean（beanName）-> getSingleTon(beanName)**

**调用getSingleTon(beanName),获取singleTonObjects中的对象，若没有且正在创建bean，查询半成品earlySingletonObjects 中的对象，若earlySingletonObjects 中没有对象，则获取singleTonFactory中的singleTonFactory，若singleTonFactory不为空则获取这个中的对象**

**最后若singleTonObject为空返回null，不为空返回**

**第一次调用位能获取对象，则创建对象**

**Step2 doGetBean（beanName）-> getSingleTon(beanName,factory)**

**若第一次没有获取对象，则调用getSingleTon（）试图自己创建一个singleTonObject，**