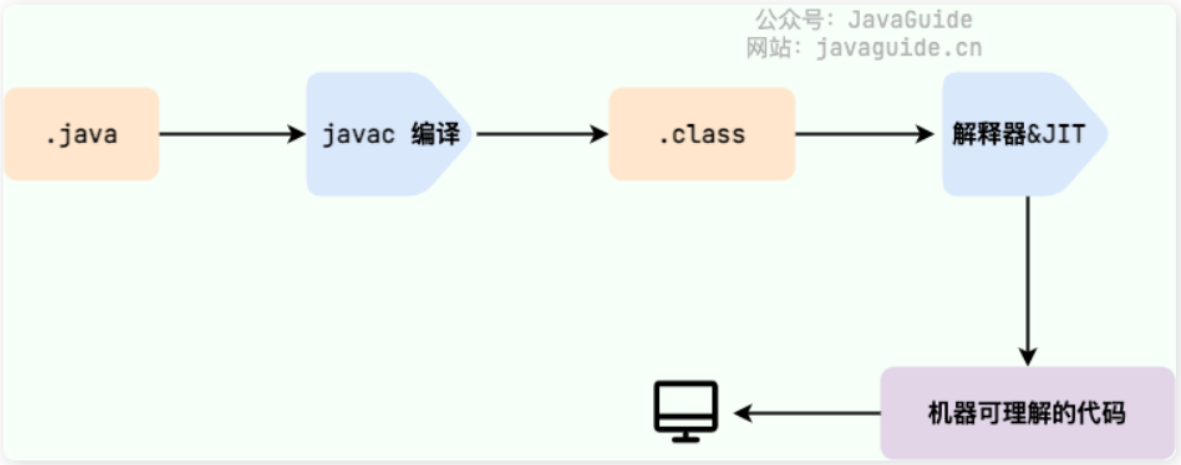
JAVA基础知识

JDK：包含JRE，同时还包含编译Java源码的编译器javac以及一些其他的工具如javadoc（文档注释工具），jdb（调试器）等等

JRE：Java运行时环境，仅包含Java应用程序的运行时环境和必要的类库，主要包括JVM和Java基础类库

JVM可以理解的代码就叫做字节码（即扩展名为.class的文件），它不面向任何特定的处理器，只面向虚拟机。

Java 程序从源代码到运行的过程如下图所示：



Java程序转变为机器代码的过程

.class到机器码这一步。JVM类加载器首先加载字节码文件，然后通过解释器逐行解释执行。JIT属于运行时编译，完成第一次编译后会将字节码对应的机器码保存下来，下次可以直接使用。Java时编译与解释共存的语言

移位运算符

<<：左移运算符，高位丢弃，低位补零，左移1位相当于乘以2（不溢出的情况下）。

>>：带符号右移，高位补符号位，低位丢弃，右移1位相当于除以2。符号位保持不变。

>>>:无符号右移，忽略符号位，空位都以零补齐。符号位跟着移位。

移位操作符实际上支持的只有int和long，编译器在对short、byte、char类型进行移位前，都会将其转换成int类型再操作

String

private且没有提供修改数组的办法保证本对象无法修改，final保证指向的地址不再发生改变，不会指向其他对象（final修饰保证其不会被继承），避免了子类破坏其不可变性

在堆中创建字符串对象，将其的引用放在字符串常量池中，字符串变量保存的是对字符串常量池中引用的引用

Java集合

comparable和comparator

对象只有实现了comparable接口collection类才能对其集合进行排序。

实现comparable接口只需实现compareTo方法即可。放在对象类的内部进行实现

comparator（比较器）接口要实现compare方法，一般作为参数传递给collection。compare方法的入参必须是对象

TreeMap

对集合内部元素根据键排序的能力和对集合内元素的搜索的能力

ArrayList

底层采用数组Object[]实现，支持随机访问，线程不安全。可以添加null值
初始时带数组容量则直接初始化数组。否则当添加第一个元素时才将容量变为初始容量10。
新容量的值为旧容量的值加上旧容量右移移位的值。约为1.5倍
如果新容量大于MAX_ARRAY_SIZE，则比较minCapacity和MAX_ARRAY_SIZE，如果大于，则为MAX_VALUE，否则为MAX_VALUE-8

LinkedList

HashMap

可以存储null值的key和value，但作为键只能有一个，作为值则可以有多个。
初始值为16，之后每次扩容容量变为原来的2倍
当链表长度大于等于阈值（默认为8）（将链表转换为红黑树之前会判断，如果当前数组长度小于64，会选择先进行数组扩容，）时，将链表转换为红黑树，以减少搜索时间。

ConcurrentHashMap

Node数组+链表/红黑树

JAVA并发

现在的Java线程本质就是操作系统的线程。
一个进程有多个线
程，多个线程共享进程的堆和方法区（元空间）资源，但每个进程有自己的程序计数器、虚拟机栈和本地方法栈
堆和方法区是所有线程共享的资源，堆是进程中最大的一块内存，用于存放新创建的对象，方法区用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

程序计数器

字节码解释器通过改变程序计数器来依次读取指令，程序计数器用于记录当前线程执行的位置

虚拟机栈

每个Java方法在执行之前会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应一个栈帧在Java虚拟机栈中入栈和出栈的过程。虚拟机栈为线程独享。
调用start方法方可启动线程并使线程进入就绪状态，直接执行run方法的话不会以多线程的方式执行

volatile

将变量声明为volatile，这就指示JVM，这个变量是共享且不稳定的，每次使用它都到主存中进行读取
防止JVM的指令重排序
保证变量的可见性，但不保证对变量操作的原子性

CAS算法

比较和交换，用于实现乐观锁，思想：用一个预期值和要更新的变量值进行比较，两值相等才会进行更新。这是一个原子操作，底层依赖于一条CPU的原子指令。会出现**ABA**问题，解决办法是在变量前加入版本号或时间戳

synchronized

主要解决的是多个线程之间访问资源的同步性，可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。早期属于重量级锁，后期引入了多项技术来减少锁的开销

修饰实例方法

给当前对象实例加锁，进入同步代码前要获得当前对象的锁

修饰静态方法

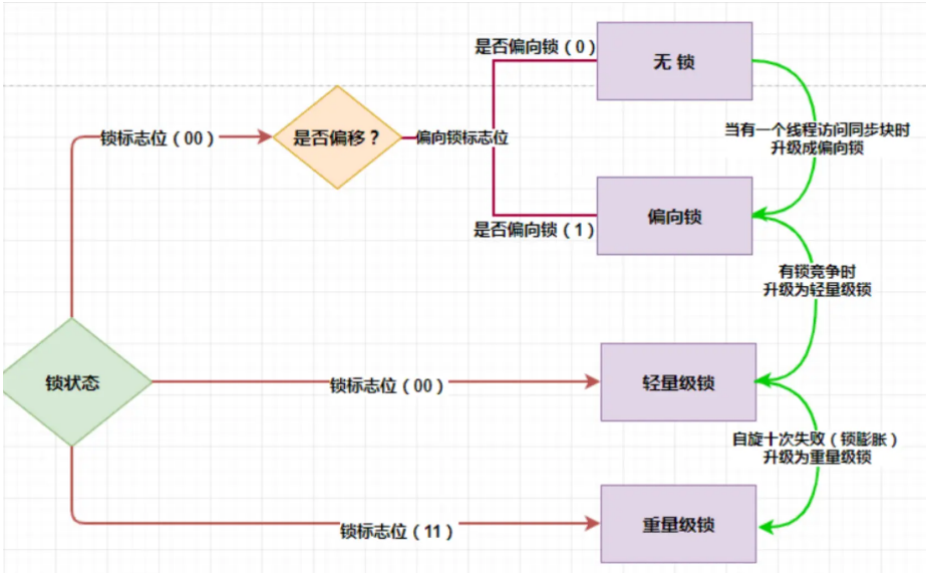
给当前类加锁，会作用于类的所有对象实例，进入同步代码前要获得当前class的锁
静态synchornized方法和非静态方法的访问不互斥。允许线程A调用实例对象的非静态方法，同时线程B调用该类的对象方法。类锁和实例对象锁不互斥

修饰代码块

锁指定对象/类。对括号里指定的对象/类加锁。

锁升级过程：

- **无锁**：没有开启偏向锁的状态，在JDK1.6之后锁是默认开启的，但是有一个偏向延迟，需要在JVM启动之后的多少秒之后才能开启，可以通过JVM参数进行配置，同时是否开启偏向锁也可以通过JVM参数进行配置。
- **偏向锁**：在偏向锁开启之后的状态，如果还没有一个线程拿到这个锁的话，这个状态叫做匿名偏向，当一个线程拿到偏向锁的时候，下次想要竞争锁只需要拿到线程ID根MarkWord当中存储的线程ID进行比较，如果线程ID相同则直接获取锁（相当于锁偏向这个线程），不需要进行CAS操作和将线程挂起的操作。
- **轻量级锁**：通过CAS操作实现，将对象的MarkWord存储到线程的虚拟机栈上，然后通过CAS将对象的MarkWord内容设置为指向Displaced Mark Word的指针，如果设置成功则获取锁，如果失败则表示有其他线程在获取锁，那么就需要在释放锁后将被挂起的线程唤醒。
- **重量级锁**：当有两个以上的线程获取锁的时候轻量级锁就会被升级成重量级锁，因为CAS如果没有成功的话始终都在自旋，进行while操作，但是在升级成重量级锁后，线程会被操作系统调度然后挂起，这样可以节约CPU资源。



ReentranLock

可重入且独占式锁，默认使用非公平锁。只能用在代码块
可中断性：线程在等待中可以被其他线程中断而提前结束等待。
设置超时时间：等待一段时间后如果还没获得锁，则放弃锁的获取。
公平锁和非公平锁
可重入性

ReentranReadWriteLock

可重入的读写锁

StampedLock

读写锁，不可重入

线程的创建方式

- 继承Thread类
  最直接的一种方式，用户自定义类继承Thread类，重写其run()方法，run()方法中定义了线程执行的具体任务。创建该类的实例后，通过调用start()方法启动线程。

```
class MyThread extends Thread {
    @Override
    public void run() {
        //线程执行的代码
    }
}

public static void main(String[] args) {
    MyThread t = new MyThread();
    t.start;
}
```

- 实现Runnable接口
  如果一个类已经继承了其他类，就不能再继承Thread类，此时可以实现Runnable接口，重写run()方法，然后将此Runnable对象作为参数传递给Thread类的构造器，创建Thread对象后调用其start方法启动线程。

```
class MyRunnable implements Runnable {
    @Override
    public void run() {
        //线程执行的代码
    }
}

public static void main(String[] args) {
    Thread t = new Thread(new MyRunnable());
    t.start();
}
```

  可以多个线程共享同一个目标对象，非常适合多个相同线程处理同一份资源的情况。

- 实现Callable接口和FutureTask
  Callable的call()方法可以有返回值并且可以抛出异常。要执行Callable任务，需将它包装进一个FutureTask，因为Thread类的构造器只接受Runnable参数，FutureTask实现了Runnable接口。

```
class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throw Exception {
        //线程执行的代码，这里返回一个整型结果
        return 1;
    }
}

public static void main(String[] args) {
    MyCallable task = new MyCallable();
    Future<Integer> futureTask = new Future<>(task);
    Thread t = new Thread(futureTask);
    t.start();

    try {
        Integer result = future.get();  //获取线程执行结果
        System.out.println(result);
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
    }
}
```

  可以获取线程的执行返回结果。
- 使用线程池
  将任务直接提交到线程池对象当中，由线程池对象负责线程的创建与启动。

线程池常见阻塞队列

Future类

当某一任务过于耗时时，可以将其交给子线程去执行，然后用future类获取执行结果

AQS（抽象队列同步器）

是一个抽象类，用于构建锁和同步器。**核心思想**：如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，就使用**CLH锁**实现线程阻塞等待以及被唤醒时锁分配机制。CLH锁是自旋锁的一种改进，是一个虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系，暂时获取不到锁的线程将被加入到该队列中。AQS将每条请求共享资源的线程封装成一个CLH队列锁的一个结点来实现锁的分配。在CLH队列锁中，一个结点表示一个线程，它保存着对线程的引用、当前结点在队列中的状态、前驱后继结点。

IO

MYSQL

存储引擎

SQL请求的执行流程

- 连接器：建立连接，管理连接、校验用户身份
- 查询缓存：查询语句如果命中缓存则直接返回，否则继续往下执行。MySQL8.0已删除该模块。
- 解析SQL：通过解析器对SQL查询语句进行词法分析、语法分析，然后构建语法树。
- 执行SQL
    - 预处理阶段：检查表或者字段是否存在
    - 优化阶段：基于查询成本考虑，选择查询成本最小的执行计划
    - 执行阶段：根据执行计划执行SQL语句，从存储引擎读取记录，返回给客户端

InnoDB

- 事务支持：InnoDB提供了对事务的支持，可以进行ACID属性的操作。MyISAM不支持事务。
- 并发性能：InnoDB采用行级锁的机制，可以提供更好的并发性能，MyISAM只支持表级锁，锁的粒度比较大。
- 崩溃恢复：InnoDB通过redolog日志实现了崩溃恢复，可以在数据库发生异常情况时，通过日志文件进行恢复，保持数据的持久性和一致性。MyISAM不支持崩溃恢复。

事务

逻辑上的一组操作，要么都执行，要么都不执行

并发事务带来的问题

- **脏读**：一个事务读取了另一个事务已修改但未提交的数据，后进行回滚，则该数据为脏数据
- **丢失修改**：第二个事务对数据的修改覆盖了第一个事务对数据的修改
- **不可重复读**：一个事务对数据的多次读取中间插入了另一个事务对数据的修改，导致读取的数据前后不一致
- **幻读**：一个事务读取了几行数据后另一事务插入了一些数据，导致前一事务发现了一些原本不存在的记录。

事务隔离级别如何实现

- 读未提交：直接读取最新数据即可，不做任何限制
- 可串行化：通过加读写锁的方式来避免并行访问
- 读提交和可重复读都是通过Read View实现，区别在于创建Read View的时机不同，读提交是在每个语句执行前都重新生成一个Read View，可重复读的隔离级别是在启动事务时生成一个Read View ，然后整个事务期间都在用这个Read View。均由MVCC多版本并发控制实现。

MYSQL锁

- **记录锁**：属于单个行记录上的锁
- **间隙锁**：锁定一个范围，不包括记录本身
- **临键锁**：锁定一个范围，包括记录本身，主要是为了解决幻读问题

如果两个范围查询的字段不是索引，就会触发全表扫描，全部索引都会加行级锁，第二条update执行时会被阻塞。

MVCC

日志

redo log（重做日志）

实现了事务中的持久性，主要用于掉电等故障恢复。
MySQL 中数据是以页为单位，你查询一条记录，会从硬盘把一页的数据加载出来，加载出来的数据叫数据页，会放入到 `Buffer Pool` 中。
后续的查询都是先从 `Buffer Pool` 中找，没有命中再去硬盘加载，减少硬盘 IO 开销，提升性能。
更新表数据的时候，也是如此，发现 `Buffer Pool` 里存在要更新的数据，就直接在 `Buffer Pool` 里更新。
然后会把“在某个数据页上做了什么修改”记录到重做日志缓存（`redo log buffer`）里，接着刷盘到 redo log 文件里。

刷盘策略

事务提交：当事务提交时，log buffer 里的 redo log 会被刷新到磁盘（可以通过 `innodb_flush_log_at_trx_commit` 参数控制，后文会提到）。
log buffer 空间不足时：log buffer 中缓存的 redo log 已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。
事务日志缓冲区满：InnoDB 使用一个事务日志缓冲区（transaction log buffer）来暂时存储事务的重做日志条目。当缓冲区满时，会触发日志的刷新，将日志写入磁盘。
Checkpoint（检查点）：InnoDB 定期会执行检查点操作，将内存中的脏数据（已修改但尚未写入磁盘的数据）刷新到磁盘，并且会将相应的重做日志一同刷新，以确保数据的一致性。
后台刷新线程：InnoDB 启动了一个后台线程，负责周期性（每隔 1 秒）地将脏页（已修改但尚未写入磁盘的数据页）刷新到磁盘，并将相关的重做日志一同刷新。
正常关闭服务器：MySQL 关闭的时候，redo log 都会刷入到磁盘里去。

binlog

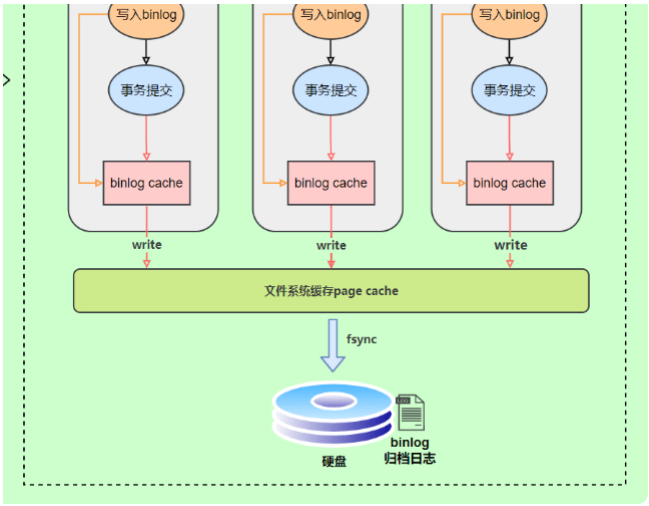
记录所有涉及更新数据的逻辑操作。有三种格式：

- **statement**：记录SQL语句原文，相当于记录逻辑操作。会有动态函数问题，比如uuid和now函数
- **row**：不仅是SQL语句原文，还包括操作的具体数据。记录下行数据最终被修改成什么样。
- **mixed**：MYSQL会判断这条sql语句是否会引起数据不一致，是，就用row格式，不是，则采用statement格式

写入机制

事务执行过程中，先把日志写到binlog cache中，事务提交时，把binlog cache写到binlog文件中。系统会给每个线程分配一块binlog cache。追加写，写满一个文件，就会创建新的文件继续写，不会覆盖以前的日志，保存的是全量的日志。

刷盘策略



write是把日志写入到文件系统的page cache中，并没有把数据持久化到磁盘。fsync才是把数据持久化到磁盘的操作

- 0：每次提交事务都只write，由系统判断什么时候执行fsync
- 1：每次提交事务都会fsync
- N：积累n个事务会进行一次fsync

两阶段提交

将redo log的写入拆成两个步骤prepare和commit。执行事务过程中写入redo log，但此时属于prepare阶段，当binlog写入完成后再执行redo log 的commit阶段

undo log 回滚日志

逻辑日志，记录的是SQL语句，比如说事务执行一条DELETE语句，那undo log就会记录一条相应的INSERT语句。同时，undo log的信息也会被记录到redo log中。用于撤销回退，保证了事务的原子性。在事务提交后就会被删除。

索引

优点：

- 使用索引可以加速数据的检索速度，减少IO次数
- 创建唯一性索引，可以保证数据库中的每一行数据的唯一性

缺点：

- 创建索引和维护索引需要耗费很多时间。当对表中的数据进行增删改查时，如果数据有索引，那么索引也需要动态的修改，会降低SQL执行效率。
- 索引需要物理文件存储，也需要一定时间。

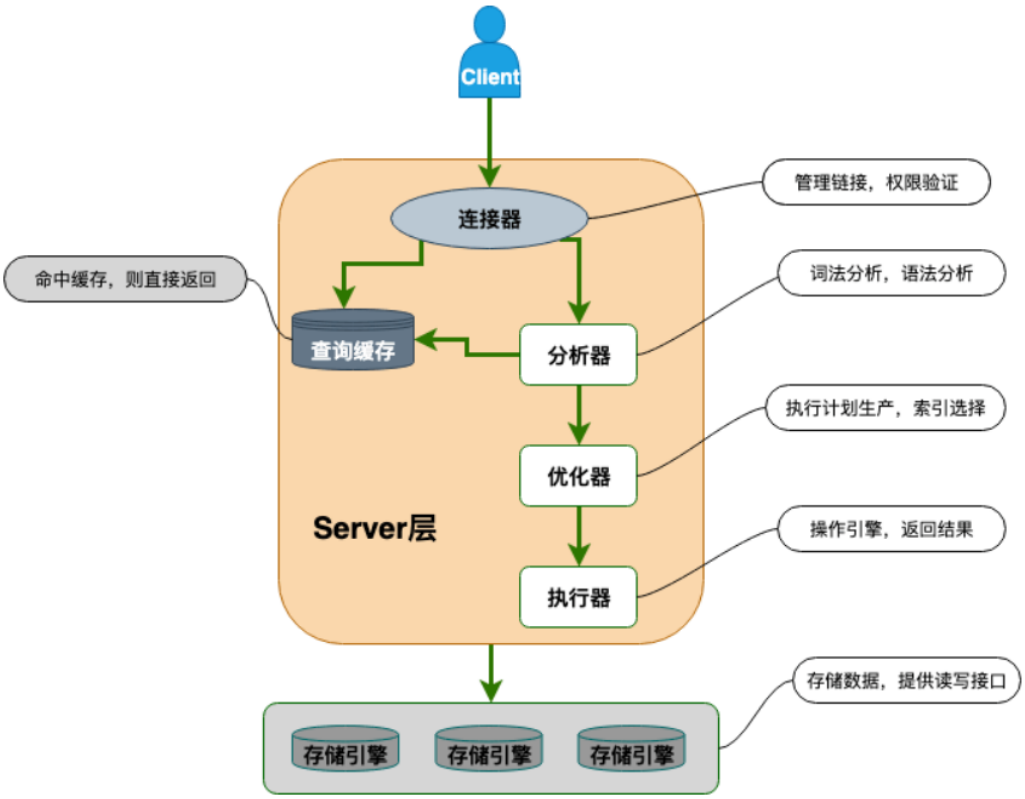
索引分类：

- 聚簇索引：索引结构和数据放在一起
- 非聚簇索引：索引结构和数据分开存放，叶子节点存放的可能是主键的值也可能是数据的位置。
- 二级索引：叶子节点存放的是主键的值，通过二级索引可以定位主键的位置
- 覆盖索引：即要查询的字段正好是索引的字段，可直接根据索引查到数据，无需回表查询
- 联合索引：使用表的多个字段创建索引，即为联合索引

索引下推：一项索引优化功能，它允许存储引擎在索引遍历过程中，执行部分 `WHERE` 字句的判断条件，直接过滤掉不满足条件的记录，从而减少回表次数，提高查询效率。

MYSQL语句的执行过程

- **连接器：** 身份认证和权限相关(登录 MySQL 的时候)。
- **查询缓存：** 执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。
- **分析器：** 没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。
- **优化器：** 按照 MySQL 认为最优的方案去执行。
- **执行器：** 执行语句，然后从存储引擎返回数据。 -



主从复制

使用binlog实现，复制的过程就是将binlog的数据从主库传输到从库上

- 写入binlog：主库写binlog日志，提交事务，并更新本地存储数据
- 同步binlog：把binlog复制到所有从库上，每个从库把binlog写到暂存日志中
- 回放binlog：回放binlog，并更新存储引擎中的数据

分库分表

分表策略

range范围分表：范围策略划分表，比如将表的主键id按照0~300万划分成一张表，300万~600万划分成一张表。**优点：**id一直在增大，有利于扩容。**缺点：**可能会有热点问题，因为订单id一直在增大，所以最近一段时间的请求都打在一张表中。

hash取模分片：对id进行hash取值后再取余。**优点：**分布比较均衡，不会出现明显的热点问题。**缺点：**不易扩容。

一致性hash：将数据和机器映射到同一个hash空间中，并按照顺时针的顺序存储数据到对应的机器中。

避免热点数据倾斜：先按照范围进行分库，再对库内的数据按照hash分片进行分表。

分页问题：解决思路

全局视野法：查到所有结果再在代码端进行分页
业务折衷法第一页全局查询，然后下一页和上一页在各个表中分别查询再在代码中进行汇总，不能跳页。

分布式shi'wu

JVM

Java内存区域详解

JAVA虚拟机栈：

每个线程独有，每个方法占一个栈帧，拥有局部变量表、操作数栈、动态链接、方法返回地址

方法区：

属于JVM运行时数据区域的一块逻辑区域，由各个线程共享。存储已被虚拟机加载的类信息、字段信息、方法信息、常量、静态变量、即时编译器编译后的代码缓存等数据。永久代是hotspot虚拟机对方法区的两种实现。

运行时常量池：

常量池表会在类加载后存放到方法区的运行时常量池中。常量池表用于存放class文件编译期生成的各种字面量和符号引用，

字符串常量池

通过=产生的字符串对象会存储于字符串常量池中。new出的字符串对象会在常量池中和堆中分别存储一份。
现存于Java堆中，同静态变量一起。对于编译期可以确定值的字符串，也就是常量字符串，jvm会将其存入字符串常量池，并且，字符串常量拼接得到的字符串常量在编译阶段也就已经被放入字符串常量池中，这个得益于编译器的优化。

垃圾回收

大多数情况下，对象在新生代中Eden区分配。大对象直接进入老年代。如果对象在Eden出生并经过第一次MinorGC后仍然能够存活，就被移动到survivor空间，增加到一定程度后会晋升到老年代。**空间分配担保，为了确保在MinorGC之前老年代本身还有容纳新生代所有对象的剩余空间**

Redis

基于内存，所以访问速度比磁盘快很多；基于Reactor模式设计开发了一套高效的事件处理模型，主要是单线程事件循环和IO多路复用；内置了多种优化后的数据类型和结构实现；通信协议实现简单且解析高效。
常用功能：

- 分布式锁
- 限流：使用RRateLimiter实现分布式限流，其底层实现就是基于Lua代码+令牌桶算法
- 消息队列：自带的list可以作为一个简单的消息队列使用。2.0引入了发布订阅功能。5.0新增一个Stream做消息队列，支持发布/订阅模式，按照消费者组进行消费。消息持久化。ACK机制，阻塞式获取消息
- 延时队列：内置了基于Sorted Set的实现的延时队列。过期事件监听或者内置的延时队列。主要使用延时队列，其中消息会被持久化，减少了丢消息的可能，消息不存在重复消费问题
- 分布式session

线程模型：一直是单线程模型，但是在4.0以后引入了多线程来执行一些大键值对的异步删除操作，6.0以后引入了多线程处理网络请求，用来提高网络读写性能。还有一些后台线程来执行一些比较耗时的模型如关闭文件、AOF刷盘、释放内存。使用IO多路复用技术来同时监听大量来自客户端的大量连接
过期字典：用来保存数据的过期时间，其键指向Redis中的某个key，值是一个longlong类型的整数，保存了过期时间。
使用跳表实现ZSet有序集合。

缓存穿透

大量请求的key是不合理的，不存在于缓存中也不存在于数据库中。解决办法：做参数校验，缓存无效key，布隆过滤器，接口限流。

Redis的数据类型和数据结构

- 五种基本数据类型：String，List，Set，Hash，Zset
- 数据结构：SDS（简单动态字符串），LinkedList（双向链表），Dict（字典），SkipList（跳表），IntSet（整数集合），ZipList（压缩列表），QuickList（快速列表）
-

String	List	Hash	Set	Zset
SDS	LinkedList/ZipList/QuickList	Dict、ZipList	Dict、Intset	ZipList、SkipList

事务

redis执行一条命令是单线程来处理的，不存在多线程安全的问题，所以是原子性。
可以使用lua脚本将多个命令写到一个脚本中，Redis会将整个脚本作为一个整体执行。当Redis执行Lua脚本时，会把lua脚本作为一个整体并把他当作一个任务加入到一个队列中，然后单线程按照队列的顺序依次执行这些任务，在执行过程中lua脚本是不会被其他命令或请求打断，因此可以保证每个任务的执行都是原子性的。
redis事务也可以保证多个操作的原子性，但事务执行发生错误了，就没办法保证原子性了。因为Redis没有提供回滚机制。

日志

持久化机制：1、AOF日志：每执行一条写操作命令，就将该命令以追加的形式写入到一个文件中；2、RDB快照：将某一时刻的内存数据以二进制的形式写入磁盘。

Spring

动态代理

一种在运行时创建代理对象的机制，主要用于在不修改原始类的情况下对方法调用进行拦截和增强

- **基于接口的代理（JDK动态代理）：** 要求目标对象必须实现至少一个接口，Java动态代理会创建一个实现了相同接口的代理类，然后在运行时动态生成该类的实例
- **基于类的代理（CGLIB动态代理）：** 不需要目标类实现接口，而是通过继承的方式创建代理类，可以在运行时动态生成一个目标类的子类。

静态代理：由程序员创建或者由指定工具创建，在代码编译时就确定了被代理的类是一个静态代理，静态代理通常只代理一个类。
动态代理：在代码运行期间，运用反射机制动态创建生成。动态代理代理的是一个接口下的多个实现类。

SpringBoot

- 简化开发：提供了一系列开箱即用的组件和自动配置，简化了项目的配置和开发过程，
- 自动化配置：通过自动配置功能，根据项目中的依赖关系和约定俗称的规则来配置应用程序，减少了配置的复杂性
- 相比spring的优势：
    - 自动化配置，通过约定优于配置的原则，很多常用配置可以自动完成
    - 快速的项目启动器，通过引入不同的starter，可以快速集成常用的框架和库，极大提高开发效率
    - 默认集成了多种内嵌服务器，无需额外配置