1. 项目：
   1. SpringBoot项目
      1. 流程
         * 1. 如果是在SpringMVC上操作需要添加文件上传的Bean的配置才能在直接获取文件的输入流。但对于SpringBoot来说不需要配置文件上传下载的解析类了 因为已经注册好了
           2. 先实现一个文件上传下载的控制类 包括访问页面的方法 上传文件的方法以及下载文件的方法

后端用RandomAccessFile和MappedByteBuffer来写入文件

* + - * 1. 设置限制文件的大小
        2. 自定义可上传的文件的类型 通过自定的intercept实现拦截器
    1. 断点续传问题
       - 1. 核心逻辑

在分片上传的过程中，如果因为系统崩溃或者网络中断等异常因素导致上传中断，这时候客户端需要记录上传的进度。在之后支持再次上传时，可以继续从上次上传中断的地方进行继续上传。

为了避免客户端在上传之后的进度数据被删除而导致重新开始从头上传的问题，服务端也可以提供相应的接口便于客户端对已经上传的分片数据进行查询，从而使客户端知道已经上传的分片数据，从而从下一个分片数据开始继续上传。

* + - * 1. 流程

前端（客户端）需要根据固定大小对文件进行分片，请求后端（服务端）时要带上分片序号和大小

服务端创建conf文件用来记录分块位置，conf文件长度为总分片数，每上传一个分块即向conf文件中写入一个127，那么没上传的位置就是默认的0,已上传的就是Byte.MAX\_VALUE 127（这步是实现断点续传和秒传的核心步骤）

服务器按照请求数据中给的分片序号和每片分块大小（分片大小是固定且一样的）算出开始位置，与读取到的文件片段数据，写入文件

* + - * 1. （拓展）分布式文件上传下载

假设分片后上传到多个服务器端，或者某个服务器爆满其他服务器空闲

* + 1. Spring Boot问题
       - 1. Springboot是什么

是Spring开源组织下的子项目

是spring组件一站式解决方案

主要是简化了使用 spring 的难度 整合了 Mybatis + SpringMVC

* + - * 1. Spring Boot的好处

1. 根据在类路径中找到的工件自动配置应用程序

2. 提供生产中应用程序的非功能性功能 例如安全性或健康检查

3. 独立运行 简化配置 自动配置 无代码生成和XML配置 应用监控

* + - * 1. 核心配置文件 和区别

1. Application文件：用于自动化配置
2. Bootstrap文件：使用场景：
3. 使用Spring Cloud Config配置中心时 需要在bootstrap中添加连接到配置中心的配置属性来加载外部配置中心的配置信息
4. 一些固定的不能被覆盖的属性
5. 一些加密或解密的场景
   * + - 1. 开启SpringBoot特性有哪几种方式
6. 继承spring-boot-starter-parent项目
7. 导入spring-boot-dependencies项目依赖
   * + - 1. SpringBoot需要独立的容器运行吗

内置了Tomcat等容器

* + - * 1. SpringBoot自动配置原理是什么

注解@EnableAutoConfiguration @Configuration @ConditionalOnClass就是自动配置核心 首先是配置文件 其次是根据类路径下是否有这个类自动配置

* + - * 1. SpringBoot是怎么启动的

在Application类下 SpringApplication.run()

SpringBootApplication有三个Annotation

@EnableAutoConfiguration 通过@import将所有符合自动配置条件的bean定义都加载到IoC容器 搜索符合自动配置条件的功能需要借助于SpringFactoriesLoader提供的配置查找的功能 即根据 @EnableAutoConfiguration的完整类名作为查找的Key 获取对应的一组Configuration类

@Configuration 和JavaConfig形式的Spring loc容器的配置类使用的@Configuration一样 是其定义成一个JavaConfig配置类

@ComponentScan 对应XML配置中的元素 其功能就是自动扫描并加载符合条件的组件（比如@Component和@Repository等）或者bean定义，最终将这些bean定义加载到IoC容器

* + 1. Spring问题
       - 1. Spring的三级缓存

利用三级缓存解决Bean循环依赖问题

SingletonObjecs 完成初始化的单例对象的cache（一级缓存）

EarlySingletonObjecs 完成实例化但没有初始化的 提前曝光的单例对象的Cache（二级缓存）

SingletonFactories 进入实例化阶段的单例对象工厂的cache（三级缓存）

* + - * 1. Spring的事务的实现原理
        2. Spring设置事务隔离级别与数据库不一致会怎么样
        3. Spring的AOP和IoC
  1. 秒杀项目
     1. 流程
        + 1. 当用户从客户端，前端进入到程序当中，首先实现前端限流 过滤一些不必要的流量 比如重复提交的页面（需要隐藏秒杀地址 并且对按钮进行处理）
          2. 然后通过Nginx层进行限流 限制在单位时间内 同一个IP允许并发数 限制同一个IP单位时间请求数
          3. 如果此时是正常流量 则直接进入平时购买的Web服务集群 如果是高并发流量则再进入分布式令牌桶利用令牌桶算法限流
          4. 此时还剩下可以控制的n个线程进入秒杀 为了不影响在秒杀的同时 别的用户的正常购买行为 所以在这里进入秒杀的服务集群 秒杀下单
          5. 搭建秒杀订单系统 从数据库获取商品信息 价格 库存等 显示在订单上 并使用原子操作 扣库存防超卖 当订单生成 再调用支付接口 支付或者更新支付状态
     2. 功能分析
        + 1. 首先对于秒杀功能分析是读多写少 高并发而且会发生资源冲突的
          2. 针对这种情况进行解决 利用缓存实现读多写少 利用限流 负载均衡 缓存 异步 队列解决高并发 利用原子操作比如数据库的悲观锁和乐观锁解决资源冲突
          3. 进行系统的基本架构

A screenshot of a cell phone

Description automatically generated

* + - * 1. 悲观锁 select \* from 表名 for update 该用户不提交 其他人没法操作

乐观锁 在表里加一个version字段 通过一个版本号去控制

相比之下 悲观锁速度更快

* + - * 1. 限流算法

1.令牌桶

1. 限速10r/s 则按照100ms固定速率填充令牌 填满了则丢弃令牌

2. 请求的速率可以突发并且令牌桶允许这种突发

3. 当请求时 则获取令牌 获取到令牌的 处理请求 没有获取到令牌的则拒绝请求

2. 漏桶

1.按一定速率滴入水滴到桶如果流入过快超过了桶的容量则直接丢弃水滴

2. 按常量速度流出水滴

* + - * 1. Nginx配置

自身的一个漏桶限流的方式 对每一个IP进行限流 1r/s 表示1s一个

* + 1. 多并发问题
       - 1. 队列思路

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。强行将多线程变成单线程的感觉

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。

* + 1. Nginx问题
       - 1. Nginx好处：跨平台、能处理高并发连接，处理2-3w并发连接数，内存消耗少。处理静态文件很好，消耗内存少。并且内置健康检查，如果一个服务器宕机，再发送请求就不会发送到宕机的服务器上
         2. Nginx性能高的原因：异步非阻塞事件处理机制，运用了epoll模型，提供了一个队列，排队解决
         3. Nginx限流：就是限制用户请求速度，防止服务器受不了，基于漏桶流算法
         4. Nginx动静分离：让动态网站里的动态页面分局一定规则把不变的资源和经常变的资源区分开来，动静资源做好了拆分以后 ，根据静态资源的特点做缓存。然后动静分开走服务器。

1. 集合类：
2. 了解集合类吗？

集合类是java.util包里的 实现数据结构的内容 可以保存多个对象的类 也叫Java容器

1. String三个类的区别

String三个类分别是String StringBuffer StringBuider

String是final类 表示不可变的字符序列 实现了Serializable接口（序列化）和Comparable接口（可比较大小）String内部 效率最低

StringBuffer是可变类 效率低但线程安全 底层为16位的数组

**扩容原理：**

**如果添加的数过了初始长度16个 默认情况下会自动扩大为原来数组长度的两倍（左移一位） + 2 每次扩容时会新建更大的数组并将地址值返回给原数组，即每次扩容地址都在变**

StringBuiider 是可变类 效率高但线程不安全 扩容原理同上

**线程安全问题：**

**在StringBuider的代码中 append方法里count += len；这个操作不是一个原子操作，假设同时又两个线程同时运行到这里执行完加法算法后值返回的count值是一样的 而不是加了两次的值**

**抛ArrayIndexOutOfBoundsException异常的原因是在进行扩容操作时 假设同时又两个线程同时进行append操作，当调用完ensureCapacityInternal()方法后 count = 5 此时线程1的cpu时间片用完了 线程2继续执行 append上 使得count = 6了 线程1继续执行getChar()方法时，拿到的count就是6 则会报ArrayIndexOutOfBoundsException的异常**

**而StringBuffer的append()方法加了synchronized() 方法保证了线程安全**

1. String拼接怎么做到的？

用加号 底层实现是先调用String.valueOf(str1) 此时会构建一个StringBuider(str1)的构造方法 即先初始化一个 StringBuider 得到str1长度\*2 + 2长度的一个数组 再append(str1) 再append(str2) 再用StringBuider.toString返回结果

1. HashMap了解吗？

HashMap是Map的主要实现类 线程不安全 效率高 可以储存双列数据即：key-value对的数据

HashMap 和HashTable的区别：

* + - * 1. Hashtable是线程安全的 hashmap不是线程安全的
        2. Hashmap可以使用null作为key，hashtable不行
        3. Hashmap继承了AbstractMap，而hashtable继承Dictionary抽象类
        4. HashMap初始容量为16，Hashtable为11.HashMap扩容时翻倍，Hashtable \* 2 + 1
        5. 底层都是数组加链表
        6. 计算hash的方法不同，hashmap是对key的hashcode二次hash然后对数组长度取模，而hashtable直接使用key的hashcode然后取模

1. HashMap底层怎么实现的？

JDK7中 HashMap底层实现原理：

JDK7中 HashMap实现的方式是哈希表+链表的方式

实例化后 底层直接创建了一个16位【因为必须是2的次幂】的一维数组 Entry<K, V>[] table 哈希表

当调用put()函数时 首先调用key的hash()计算某哈希值 【如果key为null直接返回null对应的值】得到在bucket中的下标 然后在bucket中遍历所有的节点 判断 如果为空 则添加新的Entry[]节点成功 如果遍历到了以链表的状态存在一个或者多个数据 则判断 如果hash值不同 则添加成功 如果相同 调用key所在类的equals(key2)方法 如果相等 则添加失败 如果不同则添加成功

这种方法利用链表处理hash值相同的冲突 但是如果同一个hash值的元素较多时 就会使得查找效率很低

**扩容方式：**

**将长度变成之前的两倍利用resize() 并且transfer()数据 为避免碰撞过多 对每个Entry[]的链表结点判断是否需要rehash如果要就重新计算hash值 然后相同的根据hash值算bucket下标 用头插法做结点迁移**

JDK8中 HashMap底层实现原理：

JDK8中 HashMap实现的方式是哈希表+链表+红黑树的方式

相对于JDK7中 JDK8在链表长度超过阈值(8)时 将链表转换为红黑树 可以减少查找时间

实例化后 不创建数组 调用put()之后 如果Node[]没有初始化 则初始化 用resize()方法创建长度为16的Node[] 然后根据key的hash()方法得到在bucket中的下标 然后判断当前位置是否为Null 如果是空 则添加新节点并赋值 如果不为空 判断 如果hash值相同和key值相同就覆盖value旧的值 更新为当前值 **elseif** 如果是树结构 那么插入节点 **else**就遍历table[index]的链表 如果链表中存在 则替换旧的值 否则追加到末尾 如果链表长度大于等于8了 则执行转红黑树操作 如果不成立再执行扩容。全部执行完后如果当前存储的k-v对还是超过了threshold则再扩容

**扩容方式：**

**同样将长度变成原来的两倍 然后遍历hash table的每个不是null的bucket 如果没有链表 则放在新表的相同位置储存 如果有单链表 则需要遍历单链表 并将每个结点重新计算在新标的位置 进行复制 新表容量是旧表两倍 则将单链表拆成奇偶两队 使用尾插法 挂在新数组的散列位置 减少每个链表的长度增加查找效率**

红黑树的时间复杂度是O(logn)

哈希值作为树的分支变量，如果两个哈希值不等但指向同一个bucket 较大的那个会插入到右子树 如果哈希值相等 HashMap的Key值最好是实现了 comparable接口 这样可以按照一定的顺序进行插入

A close up of a map

Description automatically generated

1. HashMap 线程安全吗？

HashMap线程不安全

1. 数据覆盖

当两个线程同时执行put()操作时 两个key都指向同一个bucket 那么此时两个结点都会做头插法 此时两个线程获取到的头结点的地址是相同的 当一个线程更新完数据和头结点后 另一个线程所拥有的头结点就过时了 此时再插入就将原来那个新增的结点抹掉了

1. 扩容时导致死循环

只有JDK7中 扩容会存在死循环 因为JDK7中的扩容resize()时 用头插法 将原本的顺序做了反转 而JDK8中 改为了两队链表 并用尾插法 这种情况下 即使是多线程 也是从头结点再做一次尾插法

在JDK7中 现有HashMap中一链表 a -> b -> c -> null

两个线程同时扩容时 线程A先完成扩容操作 链表为c ->b -> a-> null线程B才开始扩容 当前结点e指向的是原来的链表的a结点 e.next指向的依旧是原来链表的b 用头插法将a插入到线程B的头结点 然后下一结点不为null 为b 则当前e就变成了b结点 可是线程A已经更新为c ->b -> a-> null 那么e.next就是a结点 执行头插法 b结点变为头结点 next指针指向的a 此时a.next为null结点 继续执行头插法 a变成头结点 next指向b 但是e.next为null 流程结束 此时行成环链表

1. 如何避免HashMap线程不安全
2. 将Map转为包装类 Collections.SynchronizeMap() 底层新增对象锁Synchronize
3. 使用ConcurrentHashMap类替代 使用上兼容了HashMap的用法 并且线程安全

JDK7中用了分段锁技术 segment + lock JDK8中 还是用了 Synchronized修饰符

1. 为什么HashMap的加载因子是0.75？

加载因子是 填入表中的元素的个数 / 散列表的长度

HashMap是一个牺牲空间利用率 避免hash冲突的数据结构 那么如果空间利用率过高 hash冲突的发生率就会变大

则利用拉链法（链地址法）去解决冲突问题：

拉链法将位置冲突的元素构成链表 如果发生冲突就放在链表+红黑树上

根据泊松分布公式 在理想情况下 在加载因子是0.75时 结点出现在hush桶的概率遵循平均0.5的泊松分布 所以当链表长度为8时 概率为0.000,000,006几乎为不可能事件

如果为0.8会有CPU Cache Missing概率成指数上升 0.6在空间上的成本太大了 0.75就是时间和空间成本的一种折中选择

1. 红黑树有哪些参数？

红黑树关键三个参数就是 treeify threshold / untreeify threshold / min treeify threshold

树化阈值 默认为8 转换为红黑树的链表结点数量

链表还原阈值 默认为6 转换为链表的红黑树元素数量

哈希表最小树形化容量 默认为64 当哈希表中的容量大于这个值时 表中的桶才进行树形化 否则会桶内元素过多会扩容而不是树形化

1. LinkedList和ArrayList的区别 使用场景 线程安全 解决方案

LinkedList是基于 链表的数据结构 ArrayList是基于动态数组的数据结构 对于随机访问get和set ArrayList要优于LinkedList 因为LinkedList要移动指针 而对于add和remove操作 在数量少的情况下两者效率差不多 在数量大的情况下 ArrayList优于LinkedList

线程安全

ArrayList和LinkedList都是线程不安全的

对于ArrayList的add方法 如果两个线程同时在执行elementDate[size++] = e的时候 线程1执行elsementDate[size]后 暂停线程1去执行线程2 此时size还没有+1 则线程2执行相同操作的时候会造成值覆盖

解决方法

在创建对象时 调用Collections.synchronizedList(new ArrayList<Object>())即可

对于LinkedList的add方法 直接调用linkLast(e)方法 即在尾部插入数据 而同时两个线程获取同一个尾部地址 插入数据时会覆盖所以线程不安全

解决方法

和ArrayList一样 在创建对象时 调用Collections.synchronizedList(new LinkedList<Object>())即可

1. 计算机网络
   1. TCP和UDP区别

TCP是面向连接的 点对点的 有流量控制 拥塞控制 可靠交付

UDP是无连接的 尽最大可能交付 没有拥塞控制 支持一对一 一对多 多对一和多对多交互通信

应用场景

TCP用于 整个数据要求准确无误的传输 比如微信聊天 文件传输

UDP用于对通讯质量不高 但速度需要快的时候 比如语音聊天 视频聊天

UDP和TCP的数据包有最大长度限制吗？

UDP包分为包头和正文 包头共有64位 分别为16位源端口 16位目的端口 16位UDP包长度和16位校验和 因此UDP正文最大长度是2\*\*16-1字节

* 1. TCP/IP 协议

Tcp在传输层 IP在网络互连层

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

* 1. 三次握手 四次挥手

TCP建立以及断开链连接的方式 建立连接即是在客户端和服务端服务器各存储一份对方的信息 并且TCP是双向传输对等的

**三次握手：**

第一次：客户端发 syn包（syn = j）——》服务端收 客户端进入SYN\_SEND状态 此时服务端得出 客户端发和服务端收功能正常

**SYN包 是 同步序列编码 Synchronize Sequence Number**

第二次：服务端 确认客户端的SYN（ack = j + 1）发syn包（syn = k）——》客户端收 服务器进入到SYN\_RECV状态 此时客户端得出 客户端发和服务端收正常 以及 服务端发 客户端收正常

第三次：客户端收到SYN + ACK包 发确认包ACK （ack = k+1）——》服务端收 客户端和服务端进入ESTABLISHED 完成三次握手 此时服务端得出 服务端发和客户端收正常 再加上第一次得出的结论 则 客户端发和服务端收正常 以及 服务端发 客户端收正常

**四次挥手：**

第一次：客户端发了一个FIN（fin = k） 用来关闭客户端到服务端的数据传输 客户端进入FIN\_WAIT\_1状态

第二次：服务端收到后 确认客户端的FIN 发送ACK（ack = k+1）给客户端 服务端进入CLOSE\_WAIT状态

第三次：服务器发送一个FIN（fin = j）给客户端 用来关闭服务端到客户端的数据传送 服务端进入LAST\_ACK状态

第四次：客户端收到FIN后 进入TIME\_WAIT状态 发送ACK（ack = j+ 1）给服务端确认 Server收到后进入CLOSED状态 客户端等待2MSL（最大报文存活时间）释放连接

* 1. 为什么是三次握手 四次挥手

因为在收到建立连接的请求报文后 可以把ACK和SYN放在一个报文里发送给客户端

而关闭时 服务端先发送确认 表明自己知道了要关闭传输 但是可能还有数据没有发送完 发送完后 再发送FIN去告诉客户端同意关闭 所以ACK和FIN一般都分开发送

* 1. **TCP流量控制 拥塞控制**

TCP流量控制：

如果发送方把数据发送得过快，接收方可能会来不及接收，这就会造成数据的丢失。所谓流量控制就是让发送方的发送速率不要太快，要让接收方来得及接收。

1. 利用滑动窗口实现流量控制

用动态变化rwnd值的窗口值控制接受数据的速率 每次只能接受多少字节 到了就超时重发旧的数据

1. 必须考虑传输速率

可以用PSH标志位执行push操作

TCP拥塞控制：

拥塞控制是防止过多的数据注入网络 这样使得网络中的路由器或链路不致过载

拥塞控制方法

慢开始 拥塞避免 快重传 快恢复四种

* 1. http报文结构

TCP首部 + TCP数据部分

首部 20字节固定（源端口 目标端口 序号 确认号 数据偏移（首部长度）标志位 窗口 校验和 紧急指针）

确认号就是Ack序号 占32位 当ACK标志位为1时 确认号字段才有效

标志位：

URG：紧急指针有效

ACK：确认号有效

PSH：接收方应尽快把报文交给应用层

RST：重置连接

SYN：发起一个新的连接

FIN：释放一个连接

* 1. https连接请求过程

1. 客户端发起HTTPS请求
2. 服务端的配置：即必须有一套数字证书 这套证书就是一对公钥和私钥
3. 服务端传送证书：公钥：包括很多关于证书信息
4. 客户端解析证书：解析信息 如果有问题会弹出警告窗 如果没问题生成一个随机值 并用证书对它加密 约定加密算法
5. 客户端传送加密信息：让服务器得到这个随机值 之后的通信就通过这个随机值进行加密
6. 服务端解密信息：用私钥解密后 得到了随机值后 把内容通过该值进行对称加密
7. 服务端传输加密后的信息
8. 客户端解密信息

这样 除了知道私钥的双方 其他人就算监听到了加密信息也没有用 保证了数据的安全传输

* 1. http和https的区别

1. http没有加密就是明文的 然后有了SSL协议后对HTTP加密后 有了https
2. http用80端口 https用443
3. http是无状态的 https是由ssl和http构建的加密传输 比http协议安全
   1. http状态码

200 成功201 创建成功301 重定向 404错误 500 服务器错误

* 1. http1.0和1.1的区别
     1. 缓存处理不同 HTTP1.1引入了更多缓存头来控制缓存策略
     2. 带宽优化以及网络连接的使用 http1.0有些浪费 http1.1引入了range头域 只请求部分资源 返回206
     3. 错误通知的管理 1.1中新增了24个错误相应状态码
     4. Host头处理 Http1.0 不支持虚拟主机 1.1支持
     5. 长连接 http1.1支持长连接 减少了创建 关闭连接的消耗和延时
  2. https加密过程
     1. 客户端请求服务器获取 证书公钥
     2. 客户端(SSL/TLS)解析证书（无效会弹出警告）
     3. 生成随机值
     4. 用证书加密 随机值生成 密钥
     5. 客户端将 密钥 发送给服务器
     6. 服务端 用 私钥 解密 秘钥 得到随机值
     7. 将信息和随机值混合在一起 进行对称加密
     8. 将加密的内容发送给客户端
  3. 非对称加密原理 对称加密原理 区别
     1. **加密解密的过程不同**： 非对称加蜜原理采用两个秘钥，公钥加密，私钥解密 而对称加密使用一个秘钥
     2. **加密速度不同**：对称加密解密的速度比较快，适合数据比较长时的使用。非对称加密和解密花费的时间长、速度相对较慢，只适合对少量数据的使用。
     3. **安全性不同**：对称加密的过程中无法确保密钥被安全传递，密文在传输过程中是可能被第三方截获的，如果密码本也被第三方截获，则传输的密码信息将被第三方破获，安全性相对较低。

非对称加密算法中私钥是基于不同的算法生成不同的随机数，私钥通过一定的加密算法推导出公钥，但私钥到公钥的推导过程是单向的，也就是说公钥无法反推导出私钥。所以安全性较高。

* 1. 将网址输入URL中 之后发生的所有事

DNS 解析:将域名解析成 IP 地址

TCP 连接：TCP 三次握手

发送 HTTP 请求

服务器处理请求并返回 HTTP 报文

浏览器解析渲染页面

断开连接：TCP 四次挥手

* 1. 长连接： 客户端和服务端建立连接后不进行断开，之后客户端再次访问这个服务器上的内容时，继续使用这一条连接通道。

短连接： 客户端和服务端建立连接，发送完数据后立马断开连接。 下次要取数据，需要再次建立连接。

1. Linux
   1. 命令
      1. Cat 连接文件 并打印输出
      2. Tail 输出文件最后几行
      3. Grep 搜索特定字符 配合正则表达式
      4. Top 显示各个进程的资源占用情况
      5. Df 显示指定磁盘文件的可用空间
      6. Awk 可以从文件或字符串中基于制定规则浏览和抽取信息
   2. 内核态和用户态怎么切换

用户态和内核态的区别

内核态：CPU可以访问内存的所有数据，包括外围设备，例如硬盘，网卡，CPU也可以将自己从一个程序切换到另一个程序。

用户态：只能受限的访问内存，且不允许访问外围设备，占用CPU的能力被剥夺，CPU资源可以被其他程序获取。

* + 1. 用户态-》内核态
       1. 系统调用
       2. 异常
       3. 外围设备中断
  1. Linux iNode是什么

操作系统读取硬盘时 一次性读取一个块 所有文件数据都储存在块中 现在就需要一个地方存放这个块的信息 比如创建者 创建日期 文件大小 读写权限 这个地方就是iNode

【就是mac的get info出来的那些信息】

* 1. 日志怎么查找关键字 太大怎么处理

用grep查找关键字

太大就重定向 如果不需要就定向到/dev/null

* 1. 在所有文件下查找字符串 ： find xargs grep -ri 字符串
  2. Linux线程同步的方法
     + 1. 互斥锁
       2. 条件变量
       3. 信号量

1. 数据库
   1. 事务

事务是逻辑上的一组操作 要么执行要么不执行

四大特性 ： 原子性 一致性 隔离性 持久性

* 1. 并发事务的影响
     1. 脏读 一个事务在操作 未执行时 另一个事务操作了原数据
     2. 丢失修改 一个事务修改了数据 未执行 另一个事务修改了同一数据 第一次修改丢失
     3. 不可重复读 一个事务修改时 另个事务读取 两次数据不一致
     4. 幻读 一个事务增删数据时 另个事务读取 两次数据不一样
  2. 事务的隔离级别 从低到高
     1. 读未提交：允许读取尚未提交的data
     2. 读已提交：允许读取已提交的data （不可能出现脏读）
     3. 可重复读：对同一字段的多次读取结果一致 排除本身被修改（不可能出现脏读和不可重复读）
     4. 可串行化：服从ACID隔离 所有事务依次执行 事物之间完全不可能产生干扰 （不可能出现脏读 不可重复读和幻读）

MySQL InnoDB存储引擎默认级别是 可重复读

* 1. 索引的优点
     1. 创建唯一索引即主键 可以保证每行数据的唯一性
     2. 大大检索速度 减少检索量
     3. 帮助服务器避免排序和临时表
     4. 加速表和表之间的联系
  2. 选择索引的三个原则
     1. 单行访问很慢 （如果服务器从储存中读取一个数据块是为了获取其中一行则浪费空间时间）则使用索引可以创建位置引 用以提高效率
     2. 按顺序访问范围数据时很快的
     3. 索引覆盖查询很快 因为不需要回表
  3. MySQL索引的主要使用结构
     1. 哈希索引：底层哈希表 在绝大多数需求为单条的记录查询的时候 用哈希索引 查询性能最快
     2. B+Tree索引：其余情况用B+Tree索引 正是因为这个二叉树算法，让查询速度快很多，二叉树的原理，就是取最中间的一个数，然后把大于这个数的往右边排，小于这个数的就向左排，每次减半，然后依次类推，每次减半，形成一个树状结构图
  4. B+树索引

B+树就是把所有数据存在最底层的叶子节点 而非叶子节点就当做索引

对于MyISAM储存引擎：非聚簇索引

B+Tree的叶节点的data域存放的数据记录的地址 在索引检索时 首先按B+Tree搜索索引 如果指定的Key存在 则取出data域的值 然后从data域的值为地址读取相应的数据记录

对于InnoDB引擎：聚簇索引

Data文件就是索引文件 其表数据文件本身就是按B+Tree组织的一个索引结构 树的叶节点data域保存了完整的数据

这个索引的key就是主键 因此InnoDB表数据文件本身就是主索引 其他索引为辅助索引 其data域存储主键的值

MyISAM和InnoDB的区别

* + - 1. MyISAM支持表锁 而InnoDB支持行锁和表锁
      2. MyISAM不支持外键 而InnoDB支持外键
      3. MyISAM查询具有原子性 不支持事务 而 InnoDB支持事务
      4. InnoDB支持MVCC 面对高并发事件 MVCC比单纯加锁要高效 MVCC只在读已提交和可重复读两个级别下工作 可以通过悲观锁和乐观锁实现
  1. 为什么索引能提高查询速度

MySQL的基本储存结构是页 1. 每个数据页中的记录可以组成一个单向链表

2.各个数据页可以组成一个双向链表

默认查询是 定位到记录所在的页需要遍历双向链表 找到所在的页 On

索引查询是 二分查找 O(logn) 底层结构式B+Tree

* 1. 最左前缀匹配原理

在索引中按一定顺序引用多列 叫联合索引

最左前缀原则是 如果查询的条件必须精确匹配索引的左边连续一列或几列 则此列要被用到

比如 联合索引 state/city/zipCode 能命中索引的只有 state 或者 state/city 或者 state/city/zipCode

* 1. 乐观锁 悲观锁 读锁 写锁

悲观锁：总是假设最坏的情况 即每次拿数据的时候都认为别人会修改 所以每次拿数据时都会给其上锁 则修改操作阻塞

在并发下：共享资源每次只会给一个线程使用 其余线程阻塞 用完后再把资源转让给其他资源

比如 行锁 表锁 读锁 写锁

乐观锁：总是假设最好的情况 即读的时候不会有人修改 所以不上锁 但会判断是否有更新（可以使用版本号机制和CAS算法实现）

在并发下：乐观锁适用于多读的应用类型，可提高吞吐量

比如 java.util.concurrent.atomic

乐观锁的实现方式：

* + - 1. 版本号机制 在数据表中加入一个数据版本号version 每次被修改就+1

<当线程A要操作时，需要读version值 提交更新时 判断当前version值是否被修改 不一致的话重新操作>

* + - 1. CAS算法 即比较和交换：可在没有线程被阻塞的情况下实现变量的同步

V 需要读写的地址值 A进行比较的值 B 准备写入的新值

While（1）：

If V == A: CAS 通过原子的方式用B更新V

Else: continue

自旋操作

CAS缺点

1. ABA问题 初次读取V为A 被修改为B后又改回为A 用CAS就无法判断这个

2. 循环时间长 开销大

3.只能保证一个共享变量的原子操作

* 1. 主键和索引的区别

区别: 1:主键是为了标识数据库记录唯一性,不允许记录重复,且键值不能为空,主键也是一个特殊索引

2:数据表中只允许有一个主键,但是可以有多个索引

3.使用主键会数据库会自动创建主索引,也可以在非主键上创建索引,方便查询效率

* 1. 大表优化方法：
     + 1. 限制数据的范围
       2. 读写分离 从主库写 从从库里读
       3. 垂直分区 根据数据库里的标的相关性进行拆分 即根据表列拆分 把一个列多的表 拆分成多个表
       4. 水平分区 保证表结构 按某种储存策略分片区 比如年份等

水平分区最好分库 可使用客户端代理 逻辑放jar包调用 或者中间件代理 应用和数据中间加代理层

* 1. 怎么给数据库表设定权限 怎么备份数据库

利用grant和revoke命令 可以在服务器 数据库 表 列等控制访问权限

表： grant select, insert, update, delete on bond\_db.\* to pats@’%’

备份数据库

mysqldump –u root –p 数据库名>备份文件名.sql

* 1. 给存在表加入新的一列

ALTER .表名.. ADD COLUMN .列名.

添加 索引 主键 index 等

ALTER 表名 ADD 唯一索引（UNIQUE）主键（PRIMARY KEY）index（INDEX）（‘列名’）

* 1. 两阶段提交的具体过程：

UPDATE语句的结果写入内存，同时将这个操作写入redo log，此时redo log处于prepare状态，并告知执行器随时可以提交事物。

执行器生成这个操作的binlog，并写入binlog日志。

执行器通知将之前处于prepare状态改为commit状态，更新完成。

1. JVM
   1. 垃圾回收了解吗

虚拟机自动执行释放一个对象的内存 在JVM有一个垃圾回收线程 当内存不足或者虚拟机空闲的时候 自动触发执行 去回收对象

GC算法有

* + - 1. 标记清除法
      2. 复制算法 ： Eden区域使用
      3. 标记整理法
      4. 分代收集
  1. 新生代 老年代为什么这么划分

新生代生命周期长 老年代生命周期短

* 1. 内存泄漏了解吗 怎么解决

当前对象不再被引用时所占用的内存不被释放 从而造成内存空间的浪费叫内存泄漏 Out of memory

解决方法：

* + - 1. 对于要在静态类中 使用非静态外部成员变量 使用弱引用来应用 避免内存泄漏
      2. 对于不再需要的对象 显示的直接赋值null
      3. 注意对象的生命周期
      4. 对于生命周期长的对象 将内部类改为静态 用弱引用调用外部类的成员变量
  1. 强引用、软引用、弱引用和虚引用

**如果一个对象具有强引用，那垃圾回收器绝不会回收它**。当内存空间不足时，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。

**如果一个对象只具有软引用，则内存空间充足时，垃圾回收器就不会回收它**；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

**弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期**。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

**虚引用顾名思义，就是形同虚设。与其他几种引用都不同，虚引用并不会决定对象的生命周期。**如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

1. Java基础
   1. 设计模式知道哪些
      1. 单例模式 保证被创建一次 节省系统开销
      2. 工厂模式 解耦代码
      3. 观察者模式 定义对象一对多依赖 当一个对象改变 依赖者都会收到通知且自动更新
      4. 外观模式 提供一个统一的接口 用来访问子系统一群接口
      5. 装饰器设计模式 被用于多个Java IO类中 要么直接添加功能 要么派生子类
   2. 单例模式 线程安全吗 怎么改成线程安全的

单例模式就是static某个类只有一个实例 不安全 加synchronized同步 利用object锁实现

* 1. 设计模式六大原则
     1. 单一职责原则 ：就一个类而言 应该只有一个引起它变化的原因
     2. 开闭原则：类 模块 函数都应该是可以拓展的 但不可修改的
     3. 里氏替换原则：【开闭原则的具体实现】所有应用基类的地方必须能透明的使用其子类的对象
     4. 依赖倒置原则：高层模块 底层模块 都应该依赖于抽象 细节依赖于抽象
     5. 接口隔离原则：一个类对另一个类的依赖应该建立在最小接口上
     6. 迪米特原则：一个软件的实体应当尽可能的少与其他实体发生相互作用
  2. Java多态了解

面对对象的三大特性：封装 继承 多态

多态的三个必要条件：要有继承 要有重写 父类引用指向子类的对象

多态的作用：消除类型之间的耦合关系

对多态的理解：多态实现了代码的通用性 抽象类和接口必须利用多态性 因为他们不能实体化

多态的好处：

* + - 1. 可替换性
      2. 可扩充性
      3. 接口性
      4. 灵活性
      5. 简化性
  1. 抽象类和接口的区别
     + 1. 抽象类可以有构造方法 接口不能有构造方法
       2. 抽象类有普通的成员变量 或包含非抽象的方法 接口不能有
       3. 接口修饰只能是public abstract
       4. 抽象类包含静态方法 接口不能包含静态方法
       5. 接口的静态成员变量只能是public static final类型
  2. 重写和重载的区别

重写Override 就是子类对父类的方法的覆盖

重载Overload是在一个类里面 方法名相同 但参数不同 返回类型也可以不同

Super关键字 当需要在子类中调用父类的被重写方法时 要使用super关键字

* 1. Object类方法

Object类是所有类的父类

主要有 clone方法 创建并返回对象的一个副本

Equals方法 指示某个其他对象是否与此对象相等

和==的区别是 ==在基本数据类型比较值 在引用数据类型比较存放地址 而equals在默认情况下也是比较的地址值 但是可以重写 比如String类的就是比较大小

Finalize方法 当垃圾回收器确定不存在对该对象的更多引用时 垃圾回收器调用此方法

getClass方法 返回一个对象的运行类

hashCode方法 返回对象的哈希值 原则如果equals返回true 那么hashcode必须相等 返回flase不一定 如果hashcode不相等 一定返回false 一个对象类型作为集合对象的元素时 必须拥有自己的equals和hashCode的设计

notify方法 唤醒此对象监视器上的单个线程

notifyAll方法 唤醒所有线程

toString方法 将该对象的字符串表示

wait方法 使得当前线程等待 知道调用notify方法 wait还有两个重载方法

registerNatives 对本地方法进行注册

* 1. Java运行时数据区 对象如何初始化（包括类加载机制）（图也是java内存分布）

A screenshot of a cell phone

Description automatically generated

首先类加载器从硬盘里加载出.class文件 被虚拟机加载的类信息 常量 静态变量 即 使编译器编译后的代码储存在方法区 创建对象时 对象引用储存在栈帧中的局部变量表中 具体的对象在堆中分配内存

类加载机制：

虚拟机把描述类的数据从class文件加载到内存并对数据进行检验，转换解析和初始化 最终可以形成被虚拟机直接使用的Java类型

* 1. Sleep yield和wait的区别

Sleep和yield都上在thread类下的 wait 在object类下

Wait和sleep的区别在于 wait用于线程间通信的 而sleep只是短时间暂停当前线程 更重要的时 当一个线程调用wait()方法时会释放它锁持有的对象的管程和锁 但是调用sleep时不会释放管程和锁

而yield方法 仅仅释放线程所占有的CPU资源 从而让其他线程有机会运行 但不能保证某个线程能获得CPU资源 就是只单纯释放

* 1. 介绍一下Java的可序列化

序列化就是将对象写入到IO流中 反序列化就是从IO流中恢复对象 其意义是序列化机制允许将实现序列化的Java对象转换为字节序列 🙆对象可以脱离程序的运行而独立存在

使用场景：所有在网络上传输的对象都必须是可序列化的 所有需要保存到磁盘的Java对象都是可序列化的

实现方式：实现Serializable接口或者Externalizable接口

Serializable接口

步骤一：创建一个ObjectOutputStream输出流

步骤二：调用ObjectOutputStream对象的writeObject输出可序列化对象

反之创建 ObjectInputSteam输入流

两者区别：

A screenshot of a cell phone

Description automatically generated

1. 关键字
   1. Volatile关键字
      1. 保证共享变量的可见性 当一个线程更新时 修改值会立刻更新 以保证其被读取
      2. 使得共享变量禁止进行重排序
      3. 无法保证原子性

通常volatile用作保存某个状态的boolean值

* 1. Volatile可见性

Java内存模型中定义的8种工作内存和主内存之间的原子操作

Lock unlock read load use assign store write

如果赋值了一个变量volatile后 该变量对对象的操作更严格 限制use之前不能被read和load assign之后必须紧跟store和write 将read-load-use和assign-store-write成为一个原子操作

* 1. Static关键字

内存空间只有一份的

修饰（属性 方法（main） 代码块（初始化块 初始化属性 初始化对象）内部类）

Static变量为类变量 即在任何实例都不变

* 1. Final关键字

修饰类 不能被继承 比如String类

修饰方法 不能被重写

修饰变量 此时该变量为常量

1. 多线程 并发
   1. Synchronized和lock有何异同

同：都是用来同步 解决线程安全问题的

异：Synchronized 自己释放同步监视器

Lock 需要手动启动lock以及结束时unlock

* 1. 线程怎么实现

JDK8.0前可使用 Runnable接口

实现Runnable接口的好处：

1. 不会有类的单继承性的局限性

2. 可以处理多个线程有共享数据的情况

JDK8.0后可使用Callable接口

实现Callnable接口的好处：

1. 可抛异常

2. 支持泛型和返回值 用FutherTask类获取返回值

* 1. 死锁以及其解决方式

所有线程都等待对方放弃的时候行成死锁（不会报错，仅阻塞）

解决方法：1. 减少对同步资源的定义

2. 建撒后嵌套

* 1. 线程状态 转变和区别

新建状态：创建新线程 分配相应的内存空间和其他资源 调用start()进入就绪状态

就绪状态：获取CPU执行权进入运行状态、

运行状态：如果CPU已经被占用 则进入阻塞状态 否则就执行完run方法进入死亡状态

阻塞状态：在运行状态下 调用join() yield() wait() sleep() suspend(过时)都可以进入阻塞状态 堵塞时 线程不能进入排队队列 只有当引起堵塞的原因被消除后 线程转入就绪状态

死亡状态：线程调用stop()方法destory() 或者run方法执行完后 线程即处于死亡状态 此时不具备继续运行的能力

* 1. JUC下常用库和区别

JUC中atomic包下运用了AtomicBoolean AtomicInteger等原子变量

同步工具类 闭锁 信号量等

并发容器：ConcurrentHashMap CopyOnWriteArrayList

CopyOnWriteArrayList读写分离 写操作在一个复制的数组上进行 读操作在原数组上进行 读写互不影响 写操作需要加锁 防止并发时写入数据丢失 写操作结束之后需要把原数组指向新的复制数组

JUC下的Executor框架的相关类 线程池的工厂类 Executors线程池的实现类 ThreadPoolExecutor

JUC下一些阻塞队列实现类 ArrayBlockingQueue

* 1. 线程池工作机制

JUC下的ThreadPoolExecutor线程池可以

* + - 1. 降低资源消耗
      2. 提升系统响应速度
      3. 提高线程的可管理度

工作原理：

A screenshot of a social media post

Description automatically generated

先判断**核心线程池**所有线程是和否都在执行任务 如果不是 则创建线程执行当前任务

如果所有线程都在执行任务 判断当前阻塞队列有没有满 如果没有满 则阻塞提交的任务

如果阻塞队列满了判断线程池的所有线程是否都在执行任务 如果没有 则创建一个新的线程来执行任务 否则按照饱和策略进行处理

* 1. 进程之间如何通信

进程间通信主要有：

管道：允许一个进程和另一个有共同祖先的进程通信

命名管道：允许无亲缘关系进程通信 并且在系统中有对应文件名 通过调用命令mkfifo或系统调用mkfifo创建

信号：用于通知接受进程有某种事件发生

消息队列：消息队列是消息的链接表

共享内存：使得多个进程可以访问同一块内存空间

内存映射：每一个使用内存映射的进程通过一个共享文件映射到自己的进程地址空间来实现它

信号量：主要作为进程间以及同一进程不同线程之间的同步手段

套接口socket：不同机器之间的线程通信

* 1. 阻塞 非阻塞 多路复用 异步

同步和异步的概念描述的是用户线程与内核的交互方式：同步是指用户线程发起IO请求后需要等待或者轮询内核IO操作完成后才能继续执行；而异步是指用户线程发起IO请求后仍继续执行（直接返回），当内核IO操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

阻塞和非阻塞的概念描述的是用户线程调用内核IO操作的方式：阻塞是指IO操作需要彻底完成后才返回到用户空间；而非阻塞是指IO操作被调用后立即返回给用户一个状态值，无需等到IO操作彻底完成。

一个IO操作可以分为两个步骤：发起IO请求和实际的IO操作

阻塞IO和非阻塞IO的区别在于第一步，发起IO请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞IO，如果不阻塞，那么就是非阻塞IO。

同步IO和异步IO的区别就在于第二个步骤是否阻塞，如果实际的IO读写阻塞请求进程（IO操作完成后才能返回），那么就是同步IO，因此阻塞IO、非阻塞IO、IO复用、信号驱动IO都是同步IO，如果不阻塞，而是操作系统做完IO两个阶段的操作再将结果返回给用户线程，那么就是异步IO。

IO多路复用，就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

Select poll epoll是IO多路复用的机制

Select函数 监视的文件描述符有3个 writefds readfds 和 exceptfds

调用后会全阻塞 在timeout 或者有数据可写可读时返回

返回后可通过遍历fdset找到已就绪的描述符

Poll使用一个pollfd的指针 pollfd包含了监视的事件以及发生的事件

Epoll（增强版的poll）Crete（创建） ctl（空或op操作） wait（等待的事件并返回）

* 1. 线程池有哪些

1. 数据结构基础
   1. 堆有两种 最大堆和最小堆

最大堆中 父节点的值比子节点的值都要大

最小堆中 父节点的值比子节点的值都要小

所以可以很快访问到最大值和最小值

在优先队列中常用

可以访问到用户最需要的值

* 1. 堆和树的区别

树中 left < root <right

堆中 root > left和right 或者root < left和right

树中占用内存大 因为 需要分配指针到子节点

树中必须保证有序才能保证复杂度 儿茶水壶搜索更快 堆中提取信息更快

* 1. LRU算法

检查最近最少使用的数据的算法 通常适用于内存淘汰策略中

原理 将所有数据按使用时间排序 筛选出LRU数据时 取排名靠后的即可

在Java中实现

* + - 1. 需要底层数据使用双向链表 方便在链表的任意位置进行删除 在链表尾进行添加
      2. 需要将链表按访问（使用）顺序排序
      3. 当数据量超过一定阈值的时候 需要删除LRU数据

在Java中LinkedHashMap实现了大部分需求 则实现LRUCache可直接继承于LinkedHashMap

A screenshot of a social media post

Description automatically generated

1. 反射机制相关：
   1. 反射了解吗？

反射机制就是在运行时 完成一些功能 比如 对任何一个类都可以获取属性方法构造器 对任何一个对象都可以调用任何一个方法和属性

包括 反射能够调用privae的构造器 属性 和方法

反射就是用Class类 从对象出发去实现这些功能

* 1. Spring中哪里用到了反射？

Spring通过反射 通过classStr 获取Class类对象 实例化对象 并放到容器中

Spring的IOC就是工厂模式+Java的反射机制 实现控制反转

* 1. 动态代理和静态代理的区别在哪儿？

代理模式是Java模式的一种 特征是代理类和委托类有同样的接口

代理类分为静态代理类和动态代理类

静态代理类是编译创建好的代码，在运行前，代理类的.class文件就已经生成好了

动态代理类是在运行时 运用反射机制动态创建而成的

* 1. 什么是JDK动态代理？

只需要JDK环境就可以实现动态代理 无需第三方库

实现通过重写JDK里的InvocationHandler接口的invoke方法 代理的只有接口 在运行时 动态修改字节码达到修改类的目的

1. 泛型：
   1. 了解泛型吗？

泛型是限制ArrayList或其他存储物品的数据类型 用于包装类 可以避免强转操作 通配符问号的使用 就是除了Null以外所有数据都不满足

* 1. 用于哪儿？

用于包装类，异常类和静态类都不能使用泛型

1. 排序
   1. 冒泡排序 ：两两交换 O(n2)
   2. 快速排序：分区间后 取一个值 将区间内的数分为大小两部分 再分区 再取一个值 再分为大小两部分 直到每个区间只有一个数 即排序成功 O(nlogn)
   3. 插入排序：从第二个数开始往前插 如果小于前面那个数 就交换位置O(n2)
   4. 归并排序：1. 分组从中间分开 全分为两两一组2. 比大小排序为小大的格式3.每两组之间从最小的开始比大小 如果小于就往新的数组里放 如果某一组放完了所有的数 另一组直接将剩下的全部放进去就可以 4.将分的组全部都放入新的数组后 完成 O(nlogn)
   5. 堆排序：123456用大顶堆 654321用小顶堆
      1. 将一个数组构成一个大顶堆或者小顶堆
      2. 将堆顶的元素和数组末尾元素交换 将最大的或者最小的数沉到数组末端
      3. 重新调整结构 是其满足堆定义 然后继续交换堆顶和末尾元素 重复 直到全部有序

O(nlogn)