

☆ 架构师☆

1. 大型网站在架构上应当考虑哪些问题

2. 网站前端优化的技术有哪些?

3. 应用服务器优化技术有哪些?

4. 如何防止表单重复提交

♡ 如何实现一个高并发的系统?

1) 前端优化

2) 中间层负载分发

3) 控制层(网关层)

4) 服务层

5) 数据库层

☆ 电商项目☆

所需技术点

1. Nginx 反向代理为什么可以提高网站性能

2. Nginx 和 Apache 各有什么优缺点?

3. Nginx 多进程模型是如何实现高并发的?

4. tomcat、Nginx、apache 区别?

5. HttpClient 是什么

6. HttpClient 的使用

7. 跨域问题

 5.1. 为什么有跨域问题?

 5.2. 解决跨域问题的方案

 5.3. cors解决跨域

 5.3.2.1. 简单请求

 5.3.2.2. 特殊请求

 5.3.3. 实现非常简单

FastDFS

 什么是分布式文件系统

 什么是FastDFS

 FastDFS的架构

 上传和下载流程

8. RabbitMQ 的优点

9. JWT

10. 结合Zuul的鉴权流程

电商项目面试问题

1. 简单介绍下你的项目

2. 描述下你的系统架构

3. 为什么使用分布式架构

4. 你承担这个项目的哪些模块

5. 这些模块的实现思路说一下

6. 项目中哪些功能设计到了大数据访问? 你是如何解决的?

7. 项目中遇到的问题及解决办法?

8. 做完这个项目有什么收获

9. 你的项目是用什么构建的? 多模块是如何划分的? 为什么要这么做

10. 你举得图片上传注意什么

11. 如何设计商品规格的

12. 如何实现跨系统调用的

13. 单点系统的设计思想你了解么? 它在系统中的作用是什么?

14. 订单ID是怎么生成的?

15. 多台tomcat之间的session是怎么同步的

16. 如何解决并发问题

17. LVS是什么

18. 你们生产环境的服务器有多少台?

19. 线上部署情况

20. 数据备份是怎么做的? 有没有读写分离

21. 你们服务器不止一台吧, 那么你们的session是怎么同步的?

- 22.你们用什么做支付的？请求超时如何处理？
- 23.返回超时却扣钱了怎么办？
- 24.你们怎么做退款功能的，要多长时间才能把钱退回给用户？
- 25.什么是 FreeMarker？
- 26.如何解决加载页面慢的问题
- 27.如果用户一直向购物车添加商品怎么办？
- 28.商品详情页静态页面如何处理价格问题
- 29.一个电商项目，在tomcat里面部署要打几个war包？
- 30.你说你用了redis缓存，你redis存的是什么格式的数据，是怎么存的？

秒杀系统面试总结

商品超卖问题

令牌桶算法

整体流程

秒杀系统流程图

秒杀模块

- 1.简单介绍下你的项目
- 2.缓存雪崩什么情况下会发生？如何避免？
- 3.更新数据库的同时为什么不马上更新缓存，而是删除缓存？
- 4.秒杀地址在开始前为什么不应暴露给用户？
- 5.那比如说啊，我现在是个黑客，我在秒杀开始时写好了脚本，运行一万个线程获取秒杀地址，这样是不是也不公平呢？
- 6.那我可不可以创建一个ip代理池和很多用户来抢购呢？假设我有很多手机号的账户。
- 7.我把减库存生成订单再在一个事务中，都操作成功则认为秒杀成功
- 8.订单表和商品库存表是在一个数据库的吧，那如果在不同的数据库中呢？
- 9.你有没有考虑到一种情况，假如说你的缓存刚刚失效，大量流量就来查缓存，你的数据库会不会炸？
- 10.你能说说NIO么
- 11.说说进程切换时操作系统都会发生什么？
- 12.设计一个聊天系统使用TCP还是UDP
- 13.说说你的分布式事务解决方案？

☆容器☆

1.java 容器都有哪些？

Java集合的快速失败机制“fail-fast”？

2.Collection 和 Collections 有什么区别？

comparable 和 comparator的区别？

Collection 和 Collections 有什么区别？

3.List、Set、Map 之间的区别是什么？

♡HashMap

说一下HashMap的实现原理？

为什么HashMap线程不安全？

♡说一下ConcurrentHashMap的实现原理

JDK1.7与JDK1.8的不同

♡HashMap的put方法的具体流程

hashMap的扩容操作是什么？

HashMap是怎么解决哈希冲突的？

为什么hashmap中的String、Integer这样的包装类型适合作为key？

HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标？

HashMap 和 Hashtable 有什么区别？

如何决定使用HashMap还是TreeMap？

hashCode()与equals的相关规定

== 和equals区别

♡Set

说一下HashSet的实现原理？

HashSet如何检查重复？HashSet是如何保证数据不可重复的？

♡ArrayList

如何实现数组和List之间的转换？

为什么ArrayList的elementData加上transient修饰？

♡LinkedList

ArrayList和LinkedList的区别是什么?

ArrayList 和 Vector 的区别是什么?

Array 和 ArrayList 有何区别?

Queue

BlockingQueue是什么?

在 Queue 中 poll()和 remove()有什么区别?

16.哪些集合类是线程安全的?

迭代器 Iterator 是什么?

Iterator 怎么使用? 有什么特点?

list遍历方式有哪些?

Iterator 和 ListIterator 有什么区别?

☆java基础☆

1.面向对象的四大特性

封装

继承

多态

抽象

instanceof关键字的作用

2. 什么是拆装箱?

3. 重载重写

4. final、 finally、 finalize 有什么区别?

6. 什么是 java 序列化? 什么情况下需要序列化?

7.Error与Exception有什么区别?

8. heap和stack有什么区别。

10. String s = new String("xyz");创建了几个String Object?

11.String 类的常用方法都有那些?

12.java 中的 Math.round(-1.5) 等于多少?

13.short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?

内部类 (Inner class)

14.接口和抽象类有什么区别?

15.java 中 IO 流分为几种?

16.BIO、 NIO、 AIO 有什么区别?

17.Files的常用方法都有哪些?

18.try {}里有一个return语句, 那么紧跟在这个try后的finally {}里的code会不会被执行, 什么时候被执行, 在return前还是后?

19.编程题: 用最有效率的方法算出2乘以8等于几?

20.char型变量中能不能存贮一个中文汉字?为什么?

21.值传递

22.运行时异常与一般异常有何异同?

23.String 、 StringBuffer和 StringBuilder的区别

24." == "和 equals 的区别是什么?

25.Equals、 hashCode

26.什么是反射?

27.动态代理是什么? 有哪些应用?

28.怎么实现动态代理?

29.如何实现对象克隆?

30.深拷贝和浅拷贝区别是什么?

31.throw 和 throws 的区别?

32.try-catch-finally 中哪个部分可以省略?

33. try-catch-finally 中, 如果 catch 中 return 了, finally 还会执行吗?

34.常见的异常类有哪些?

35.java中的两种异常类型是什么? 他们有什么区别?

☆JavaWeb☆

1. jsp 和 servlet 有什么区别?

2.jsp 有哪些内置对象? 作用分别是什么?

3.说一下 jsp 的 4 种作用域?

4.session 和 cookie 有什么区别?

购车session 实现思路

- 5.说一下 session 的工作原理?
- 6.如果客户端禁止 cookie 能实现 session 还能用吗?
- 7.spring mvc 和 struts 的区别是什么?
- 8、doGet()和doPost()区别?
 - 8.1 forward 和 redirect 的区别?
- 9、servlet的生命周期
- 10、如何现实servlet的单线程模式
- 11. forward 和redirect的区别?
- 5.什么情况下调用doGet()和doPost()?
- 6.JSP和Servlet有哪些相同点和不同点，他们之间的联系是什么?
- 7.请解释Filter和Listener的理解及作用? .
- 8.jdbc流程
- 9.Ajax

- 1.页面编码和被请求的资源编码如果不一致如何处理?
- 2.Ajax 的过程
- 3.简述异步加载
- 10.JS选择器
- 11.拦截器和过滤器有什么区别
- EJB
 - EJB与JAVA BEAN的区别?
 - 1、EJB容器提供的服务
 - 2、EJB的角色和三个对象
 - 2、EJB的几种类型
 - 3、bean 实例的生命周期
 - 4、激活机制
 - 5、remote接口和home接口主要作用
 - 6、客服端调用EJB对象的几个基本步骤

☆ Spring☆

- spring 由哪几部分组成?
- 1.为什么要使用Spring?
- 谈谈你对IOC的理解?
- 谈谈你对AOP的理解?
- 请解释一下 String AOP 里面的几个名词
- Spring通知有哪些类型?
- 3.Spring AOP 和 AspectJ AOP 有什么区别?
- 4.spring 常用的注入方式有哪些?
- Spring 中单例 bean 是线程安全的么?
- Spring 如何处理多线程并发问题?
- 6.spring 支持几种 bean 的作用域?
- 7.spring 自动装配 bean 有哪些方式?

创建Bean的三种方式

注解

- @Required 注解
- @Autowired 和@Resource区别?
- @Qualifier注解的作用
- @Component 和 @Bean 的区别是什么?

8.Spring 事务种类

●Spring 事务传播行为

●Spring 中的隔离级别

@Transactional(rollbackFor = Exception.class)注解了解吗?

14.BeanFactory和 ApplicationContext 的区别详解

15.Spring 有几种配置方式

●请解释Spring Bean 的生命周期

22.构造方法注入和设值注入有什么区别?

Spring 框架中有哪些不同类型的事件

24.FileSystemResource 和 ClassPathResource 有何区别

Spring 中用到了哪些设计模式?

Spring 如何解决循环依赖?

☆ Spring MVC ☆

2.优点

- 3.Spring MVC 工作流程
- 4.Spring MVC 和struts2的区别
- 5.基础传参
- 6.重定向 和转发
- 7.SpringMVC 用什么对象从后台像前台传递数据
- 8.怎么把ModelMap 里面的数据放入Session里面
- 9.@ResponseBody注解 与 Ajax
- 11.spring mvc 有哪些组件?
- 12.@RequestMapping 的作用是什么?
- 13. 如何解决Web页面乱码问题?
- 14.@RestController和 Controller
- 15.@Resource 和@Autowired 区别?
- 16.Spring MVC常用注解

☆ Spring boot ☆

- 1.什么是 spring boot?
 什么是Spring Boot的启动流程?
- 2.spring boot 核心配置文件是什么?
 Spring Boot 的核心注解是哪个? 它主要由哪几个注解组成的?
 自动配置原理是什么?
 Spring Boot 配置加载顺序?
- 3.什么是YAML?
 Spring Boot 是否可以使用 XML 配置 ?
 spring boot 核心配置文件是什么?
- 4.Spring boot 中的监视器是什么
- 4.如何在自定义端口上运行 Spring boot
- 5.什么是Spring Profiles?
- 6.请谈谈什么是RESTful

☆ Spring Cloud ☆

- 1.什么是 spring cloud?
- 2.spring cloud 断路器的作用是什么?
- 3.spring cloud 的核心组件有哪些?
- 4.服务注册和发现是什么意思?
- 5.负载均衡的意义是什么
- 6.什么是 Hystrix? 它如何实现容错
- 7.什么是 Netflix Fegin
- 8, Feign 的优缺点
- 9.什么是Spring Cloud Bus

☆ Hibernate ☆

- 1.hibernate 的执行流程
- 2.Hibernate 缓存、延迟加载

☆ Mybatis ☆

- 理解 Mybatis
- Mybatis 是如何解决 JDBC 的不足之处的
- Mybatis 的编程步骤
- 1. #{}和 \${}的区别是什么?
- 2.Mybatis分页
 - 1.mybatis 有几种分页方式?
 - 2.mybatis 逻辑分页和物理分页的区别是什么?
 - 3.简述 Mybatis 的插件运行原理, 以及如何编写一个插件。
 - 4.mybatis 分页插件的实现原理是什么?
- 3.延迟加载、缓存
 - 1.mybatis 是否支持延迟加载? 延迟加载的原理是什么?
 - 2.说一下 mybatis 的一级缓存和二级缓存?
- 4.批处理
 - 1.Mybatis 中如何执行批处理
 - 2.Mybatis 执行批量插入, 能返回数据库主键列表吗?

5.mybatis 和 hibernate 的区别有哪些?

6.Mybatis执行器

 1.mybatis 有哪些执行器 (Executor) ?

 2.Mybatis 中如何指定使用哪一种 Executor 执行器?

7.动态SQL

 Mybatis 动态 sql 是做什么的? 都有哪些动态 sql? 能简述一下动态 sql 的执行原理不?

8.Mybatis 是如何将 sql 执行结果封装为目标对象并返回的? 都有哪些映射形式?

9.Mybatis 有几种查询方式? 各有什么区别?

 1.MyBatis 实现一对一有几种方式

 2.Mybatis 能执行一对一、一对多的关联查询吗? 都有哪些实现方式, 以及它们之间的区别。

 3.如果要查询的表名和返回的实体Bean对象不一致, 那你是怎么处理的?

11.映射文件

 1.简述Mybatis 的XML 映射文件和Mybatis 内部数据结构之间的映射关系

 2.不同的Xml 映射文件, id 是否可以重复

 3.Xml 映射文件中, 除了常见的 select|insert|update|delete 标签之外, 还有哪些标签?

 4.Mybatis 映射文件中, 如果 A 标签通过 include 引用了 B 标签的内容, 请问, B 标签能否

 定义在 A 标签的后面, 还是说必须定义在 A 标签的前面?

12.Dao 接口的工作原理是什么?

13.Mybatis 是否可以映射 Enum 枚举类?

19.如何获取自动生成的 (主) 键值

20.resultType 和resultMap 区别

21.使用Mybatis 的mapper 接口调用时有哪些要求

22.jpa 和 hibernate 有什么区别?

23.为什么说 Mybatis 是半自动 ORM 映射工具? 它与全自动的区别在哪里?

23.如何使用JPA在数据库中非持久化一个字段?

☆设计模式☆

1.单例模式

2.观察者模式

3.装饰者模式

3.适配器模式

简单工厂和抽象工厂有什么区别?

☆JVM☆

1.tomcat 8 优化

 1.禁用AJP连接

 2.使用执行池 (线程池)

 3.调整运行模式

 4.调整堆的比例参数

运行时数据区哪些会Error哪些会GC

String Table

 改变

 String 不可变性

 String Pool (字符串常量池)

 为什么String Table 移动到堆中?

 String intern 的使用

△JDK1.8和JDK1.7VM内存区域的区别?

■JVM内存区

○程序计数器

 为什么将PC设置为线程私有的?

○虚拟机栈

 1.参数

 2.基本知识

 3.虚拟机栈包括哪几部分?

 1. 局部变量表

 2. 操作数栈

 3. 动态链接

 1.概念

 2.链接

 3.语言类型的区别

4. 方法返回地址

一个方法的结束，有两种退出方式：

5. 一些附加信息

4. 方法中定义局部变量是否线程安全？

为什么需要运行时常量池？



1. 基本参数

2. 创建一个对象的过程

3. MinorGC、MajorGC、FullGC区别

4. FullGC 触发条件一般分为以下五种情况。

5. 堆空间为什么要分代？

6. 内存分配策略

7. TLAB 是什么？

8. 逃逸分析



方法区

1. 基本参数

方法区的演进细节

运行时常量池

常量池

为什么需要常量池？

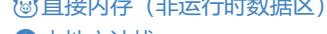
堆、栈、方法区的关系

为什么永久代要被元空间替代？

方法区的垃圾回收？

StringTable 为什么要调整位置？

静态变量存放在哪里？



直接内存（非运行时数据区）



本地方法栈

说一下JVM内存模型？

描述一下类加载过程？

简述类加载机制？

描述一下类加载器？

如何实现自定义加载器？

JVM的永久代中会发生垃圾回收么？

什么是双亲委派机制？

聊聊垃圾回收算法？

引用计数法和可达性分析算法？

垃圾回收器

serial 回收器

parallel Scavenge 回收器

CMS 回收器

参数

CMS 垃圾收集分为4个阶段

CMS 为什么无法处理浮动垃圾？

为什么 CMS 不在堆满时进行垃圾回收？

CMS 为什么不采用标记整理算法？

G1 垃圾收集器

特点

G1 垃圾收集器的优势

G1 垃圾收集器的缺点

G1 垃圾回收过程

G1 和 CMS 的区别？

垃圾回收器总结

☆java并发☆



基础

1. 并行和并发有什么区别？

2. 线程和进程的区别？

3. 守护线程是什么？

♡ 创建线程有哪几种方式？

什么是死锁?

怎么防止死锁?

说一下 **Runnable** 和 **Callable** 有什么区别?

线程的 run() 和 start() 方法有什么区别?

为什么要调用 start() 而不是 run()?

多线程锁的升级原理是什么?

为什么代码会重排序?

as-if-serial 规则和 happens-before 规则的区别

线程的基本状态

7. 状态之间的关系

Java 中用到的线程调度算法是什么?

线程的调度策略

什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)?

请说出与线程同步以及线程调度相关的方法。

11. 启动一个线程是用 run() 还是 start()?

10. 线程的 run() 和 start() 有什么区别?

8. sleep() 和 wait() 有什么区别?

9. notify() 和 notifyAll() 有什么区别?

你是如何调用 wait() 方法的? 使用 if 块还是循环? 为什么?

为什么线程通信的方法 wait(), notify() 和 notifyAll() 被定义在 Object 类里?

为什么 wait(), notify() 和 notifyAll() 这些方法必须在同步代码块中调用?

Thread 类中的 yield 方法有什么作用?

线程的 sleep() 方法和 yield() 方法有什么区别?

如何停止一个正在运行的线程?

Java 中 interrupted 和 isInterrupted 方法的区别?

Java 如何实现多线程之间的通讯和协作?

线程的同步

同步的范围越小越好。

为什么要使用同步?

如何实现同步?

请说出你所知道的线程同步的方法。

线程类的构造方法、静态块是被哪个线程调用的

线程池

饱和策略

如果你提交任务时, 线程池队列已满, 这时会发生什么

线程池都有哪些状态?

线程池中 submit() 和 execute() 方法有什么区别?

在 java 程序中怎么保证多线程的运行安全?

synchronized

三种使用方式

synchronized 底层实现原理?

为什么会有两个 monitorExit 呢?

synchronized 可重入的原理

什么是自旋

多线程中 synchronized 锁升级的原理是什么?

锁升级的目的?

当一个线程进入一个对象的 synchronized 方法 A 之后, 其它线程是否可进入此对象的 synchronized 方法 B?

synchronized 和 Lock 有什么区别?

synchronized 和 ReentrantLock 区别是什么?

volatile

volatile 关键字的作用

final

都哪些类是 final 的?

synchronized 和 volatile 的区别是什么?

Synchronized 和 ThreadLocal 的区别

ThreadLocal 是什么? 有哪些使用场景?

25.1 ThreadLocal 是怎么解决并发安全的

- 25.2Thread Local 需要注意什么
- 26.JVM 对 Java 原生锁做了哪些优化
- 27.为什么说 Synchronized 是非公平锁
- 28.什么是锁消除和锁粗化?
- 29.CAS
- 30.ABA问题、乐观锁的缺点
- 31.AQS 问题
- 32.ReentrantLock 是如何实现可重入性的?
- 33.如何让 Java 的线程彼此同步?
- 34.CountDownLatch、CyclicBarrier区别
- 35.Java中的线程池是如何实现的
- 36.创建线程池的几个核心构造参数?
- 37.线程池中线程是怎么创建的?
- 38.线程池的异同
- 39.线程池的数量该设置为多少合适?
- 40.如何在线程池中提交线程
- 41.什么是Java 内存模型
- 42.todo.锁的升、降级
- 43.volatle 有什么特点，为什么它能保证变量对所有的线程可见性
- 44.Java 内存模型定义的8 种内存间操作
- 45.volatle是否可以保证并发安全
- 46.说一下 atomic 的原理?

☆ RabbitMQ ☆

- 1.的使用场景有哪些?
- 2.有哪些重要的角色?
- 3.有哪些重要的组件?
- 4.rabbitmq 中 vhost 的作用是什么?
- 5.rabbitmq 的消息是怎么发送的?
- 6.rabbitmq 怎么保证消息的稳定性?
- 7.rabbitmq 怎么避免消息丢失?
- 8.要保证消息持久化成功的条件有哪些?
- 9.rabbitmq 持久化有什么缺点?
- 10rabbitmq 有几种广播类型?
- 11.rabbitmq 怎么实现延迟消息队列?
- 12.rabbitmq 集群有什么用?
- 13.rabbitmq 节点的类型有哪些?
- 14.rabbitmq 集群搭建需要注意哪些问题?
- 15.rabbitmq 每个节点是其他节点的完整拷贝吗? 为什么?
- 16.rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况?
- 17.rabbitmq 对集群节点停止顺序有要求吗?

☆ zookeeper ☆

- 1. zookeeper 是什么?
- 2.zookeeper 都有哪些功能?
- 3.zookeeper 有几种部署模式?
- 4.zookeeper 怎么保证主从节点的状态同步?
- 5.集群中为什么要要有主节点?
- 6.集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗?
- 7.说一下 zookeeper 的通知机制?

☆ Dubbo ☆

☆ MySQL ☆

关系型数据库和非关系型数据库的区别?

Mysql架构

3. 说一下MySQL执行一条查询语句的内部执行过程?

4.SQL Select 语句的执行顺序

优化器选择查询索引的影响因素?

3.数据库的三范式是什么?

♡ InnoDB和MyISAM的区别?

InnoDB和MyISAM 面试题

♡索引

- 索引的优缺点
- 索引的类型
- 主键索引和唯一索引的区别?
- 唯一索引和普通索引哪个性能更好?
- InnoDB中为什么主键索引高于唯一索引
- 常见索引存储算法
- 为什么B+Tree要比B-Tree的性能好?
- 什么是覆盖索引?
- 怎么验证 MySQL 的索引是否满足需求?**
- 如何查询一张表的所有索引?
- MySQL最多可以创建多少个索引?
- MySQL如何指定查询的索引?
- 在MySQL中指定了查询索引,为什么没有生效?
- MySQL会错选索引吗?
- 如何解决MySQL错选索引的问题?
- 如何优化身份证索引?
- 前缀索引
- 创建索引的时机?
- 索引优化口诀

♡什么叫回表查询?

为什么MySQL官方建议使用自增主键?

♡事务

- 1.说一下ACID是什么
- 丢失修改、脏读、不可重复读、幻读分别是什么?**
- 如何避免幻读?
- 说一下MySQL的隔离级别
- 如何设置MySQL的事务隔离级别?
- InnoDB默认的事务隔离级别是什么?如何修改?
- 如何手动操作事务?
- InnoDB如何开启手动提交事务?

并发事务有什么问题?如何解决?

MVCC是如何工作的?

♡可重复读隔离级别下MVCC如何工作?

♡锁、并发

- 说一下MySQL的行锁和表锁?**
- 说一下乐观锁和悲观锁?**
- 乐观锁有什么缺点(ABA)?
- MySQL都有什么锁?
- select for update 和lock in share mode的使用
- 什么是全局锁?它的应用场景?
- 使用全局锁会导致什么问题?
- 什么是共享锁?
- 什么是排他锁?
- 如何设置数据库为全局只读锁?
- FTWRL和set global readonly=true有什么区别?
- 如何实现表锁?

InnoDB存储引擎有几种锁算法?

InnoDB如何实现行锁?

共享锁和排他锁的区别?

谈谈如何优化锁?

♡死锁

- 什么情况下会造成死锁?
- 死锁的解决办法?
- 如何查看死锁?
- 如何避免死锁?
- InnoDB是如何对待死锁的?
- 如何开启死锁检测?

♡ MySQL高并发解决方案

聊聊数据库崩溃时事务的恢复机制?

♡ MySQL基础

1.如何获取当前数据库版本?

如何查看 MySQL 的空闲连接?

2.char 和 varchar 的区别是什么?

3.float 和 double 的区别是什么?

count(column) 和 count(*) 有什么区别?

4.什么是内连接、左连接、右连接有什么区别?

什么是临时表? 临时表什么时候删除?

内连接、外连接

MySQL 什么情况下导致自增主键不连续?

InnoDB 中自增主键能不能被持久化?

独立表空间和共享表空间

如何设置独立表空间?

如何进行表空间压缩?

说一下重建表的执行流程?

表的结构存储在哪里?

如果把一个 InnoDB 表的主键删掉, 是不是就没有主键, 就没办法进行回表查询了?

执行一个 update 语句以后, 我再去执行 hexdump 命令直接查看 ibd 文件内容, 为什么没有看到数据有改变呢?

MySQL 支持枚举吗? 如何实现? 它的用途是什么?

InnoDB 和 MyISAM 执行 select count(*) from t, 哪个效率更高? 为什么?

在 MySQL 中有对 count(*) 做优化吗? 做了哪些优化?

在 InnoDB 引擎中 count(*)、count(1)、count(主键)、count(字段) 哪个性能最高?

使用 delete 误删数据怎么找回?

delete 和 truncate 区别?

Flashback 恢复数据的原理是什么?

♡ MySQL数据表的基本操作

表的约束

索引

CRUD

单表查询

聚合查询

多表操作

子查询

♡什么是页?

♡日志

redo log 和 binlog 有什么区别?

什么是 crash-safe?

什么是脏页和干净页?

MySQL 刷脏页的速度很慢可能是什么原因?

如何控制 MySQL 只刷新当前脏页?

MySQL 的 WAL 技术是解决什么问题的?

为什么有时候会感觉 MySQL 偶尔卡一下?

redo log 和 binlog 是怎么关联的?

MySQL 怎么知道 binlog 是完整的?

MySQL 中可不可以只要 binlog, 不要 redo log?

MySQL 中可不可以只要 redo log, 不要 binlog?

为什么 binlog cache 是每个线程自己维护的, 而 redo log buffer 是全局共用的?

事务执行期间, 还未提交, 如果发生 crash, redo log 丢失, 会导致主备不一致呢?

在 MySQL 中用什么机制来优化随机读/写磁盘对 IO 的消耗?

redo log 和 bin log 面试题

有没有办法把 MySQL 的数据恢复到过去某个指定的时间节点? 怎么恢复?

♡ MySQL性能优化

性能指标

Mysql 优化举例

♡MySQL 的常见的优化手段有以下五种:

♡ MySQL 常见读写分离方案有哪些?

什么是 MySQL 多实例? 如何配置 MySQL 多实例?

介绍一下 Sharding-JDBC 的功能和执行流程?

♡ 怎样保证确保备库无延迟?

♡ MySQL 主从延迟的原因有哪些?

如何保证数据不被误删?

MySQL 服务器 CPU 飙升应该如何处理?

MySQL 毫无规律的异常重启, 可能产生的原因是什么? 该如何解决?

mysql 问题排查都有哪些手段?

MySQL 慢查询怎么解决?

查询长时间不返回可能是什么原因? 应该如何处理?

23. 垂直切分、水平切分、分库分表、读写分离

♡ 主从复制

主从复制的几种方式?

slave 是否可以进行写操作?

为什么需要多个 slave?

4. 主从复制中有 master, slave1, slave2, ... 等等这么多 MySQL 数据库, 那比如一个 JAVA WEB 应用到底应该连接哪个数据库?

5. 当 master 的二进制日志每产生一个事件, 都需要发往 slave, 如果我们有 N 个 slave, 那是发 N 次, 还是只发一次?

6. 当一个 select 发往 MySQL proxy, 可能这次由 slave-2 响应, 下次由 slave-3 响应, 这样的话, 就无法利用查询缓存了.

7. 随着应用的日益增长, 读操作很多, 我们可以扩展 slave, 但是如果 master 满足不了写操作了, 怎么办呢?

☆ Redis ☆

1. redis 是什么? 都有哪些使用场景?

redis 支持的数据类型有哪些?

♡ redis 和 memecache 有什么区别?

4. redis 为什么是单线程的? 使用队列技术将并发访问变成串行访问

♡ 什么是缓存穿透? 怎么解决?

缓存降级

♡ 什么是缓存雪崩? 如何解决?

什么是缓存预热?

♡ 什么是缓存击穿? (热点key重建优化问题)

缓存无底洞问题

缓存粒度问题

缓存预热

♡ redis 事务

什么是事务

为什么 Redis 事务不具备原子性?

Redis 事务相关命令有哪些?

8. 怎么保证缓存和数据库数据的一致性?

♡ redis 持久化

Redis 的持久化有哪两种方式?

AOF (Append-only file)

RDB (redis DataBase)

♡ redis 分布式锁?

1. 分布式锁的基本要素

2. 单机实例锁实现

3. 分布式系统如何实现分布式锁?

算法是异步的吗?

12. redis 如何做内存优化?

13. redis 淘汰策略有哪些?

redis 缓存失效策略?

14. redis 常见的性能问题有哪些? 该如何解决?

18. 一个字符串类型的值能存储最大容量是多少?

20. redis 的同步机制了解么

21. Pipeline 有什么好处, 为什么要用 pipeline?

♡ redis集群

redis 集群

讲讲主从复制

redis 主从架构

redisCluster

哨兵

Redis集群方案什么情况下会导致整个集群不可用？

说说redis 哈希槽的概念

Redis集群的主从复制模型是怎样的？

Redis集群会有写操作丢失吗？为什么？

Redis集群之间是如何复制的？

Redis集群默认选择0数据库，不支持选择。

⌚什么是慢查询？怎么配置？

29.Redis回收进程如何工作的？

30.Redis的内存用完了会发生什么？

31.一个 redis 实例最多存放多少keys？

32.MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？

34.假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

35.如果有大量的key需要设置同一时间过期，一般需要注意什么？

36.使用过Redis做异步队列么，你是怎么用的？

☆计算机网络☆

1.TCP和UDP的区别

1.1浏览器输入URL发生了什么？

2.说一下TCP三次握手

三次握手

tcp 为什么要三次握手，两次不行吗？为什么？

为什么连接建立需要三次握手，而不是两次握手？

这里会有人问，那么A 第三次发送给 B 的信号丢失了呢？

如果已经建立了连接，但是客户端突然出现故障了怎么办？(网络CLOSE)

三次握手连环炮

三次握手过程中可以携带数据么？

3.说一下 TCP 4次挥手？

1.为什么A要先进入TIME-WAIT状态，等待2MSL时间后才进入CLOSED状态？

2.为什么连接的时候是三次握手，关闭的时候却是四次握手？

3.状态为TIME_WAIT是不是所有执行主动关闭的socket都会进入TIME_WAIT状态呢？有没有什么情况使主动关闭的socket直接进入CLOSED状态呢？

等待2MSL的意义

为什么是四次挥手而不是三次？

4.说一下 tcp 粘包是怎么产生的？

为什么会发生TCP粘包、拆包呢？

TCP粘包/拆包的原因

5. forward 和 redirect 的区别？

6.计算机网络体系模型

7.如何实现跨域

8.说一下 JSONP 实现原理？

9.http 响应码 301 和 302 代表的是什么？有什么区别？

☆算法☆

1.B树和B+树的区别

2.你了解哪些排序算法么

3.解决hash 冲突的方法

1. 大型网站在架构上应当考虑哪些问题

分层

- 分层是处理任何复杂系统最常见的手段之一，将系统横向切分成若干个层面，每个层面只承担单一的职责，然后通过下层为上层提供的基础设施和服务以及上层对下层的调用来形成一个完整的复杂的系统。计算机网络的开放系统互联参考模型（OSI/RM）和Internet的TCP/IP模型都是分层结构，大型网站的软件系统也可以使用分层的理念将其分为持久层（提供数据存储和访问服务）、业务层（处理业务逻辑，系统中最核心的部分）和表示层（系统交互、视图展示）。需要指出的是：（1）分层是逻辑上的划分，在物理上可以位于同一设备上也可以在不同的设备上部署不同的功能模块，这样可以使用更多的计算资源来应对用户的并发访问；（2）层与层之间应当有清晰的边界，这样分层才有意义，才更利于软件的开发和维护。

分割

- 分割是对软件的纵向切分。我们可以将大型网站的不同功能和服务分割开，形成高内聚低耦合的功能模块（单元）。在设计初期可以做一个粗粒度的分割，将网站分割为若干个功能模块，后期还可以进一步对每个模块进行细粒度的分割，这样一方面有助于软件的开发和维护，另一方面有助于分布式的部署，提供网站的并发处理能力和功能的扩展。

分布式

- 除了上面提到的内容，网站的静态资源（JavaScript、CSS、图片等）也可以采用独立分布式部署并采用独立的域名，这样可以减轻应用服务器的负载压力，也使得浏览器对资源的加载更快。数据的存取也应该是分布式的，传统的商业级关系型数据库产品基本上都支持分布式部署，而新生的NoSQL产品几乎都是分布式的。当然，网站后台的业务处理也要使用分布式技术，例如查询索引的构建、数据分析等，这些业务计算规模庞大，可以使用Hadoop以及MapReduce分布式计算框架来处理。

集群

- 集群使得有更多的服务器提供相同的服务，可以更好的提供对并发的支持。

缓存

- 所谓缓存就是用空间换取时间的技术，将数据尽可能放在距离计算最近的位置。使用缓存是网站优化的第一定律。我们通常说的CDN、反向代理、热点数据都是对缓存技术的使用。

异步

- 异步是实现软件实体之间解耦合的又一重要手段。异步架构是典型的生产者消费者模式，二者之间没有直接的调用关系，只要保持数据结构不变，彼此功能实现可以随意变化而不互相影响，这对网站的扩展非常有利。使用异步处理还可以提高系统可用性，加快网站的响应速度（用Ajax加载数据就是一种异步技术），同时还可以起到削峰作用（应对瞬时高并发）。“能推迟处理的都要推迟处理”是网站优化的第二定律，而异步是践行网站优化第二定律的重要手段

冗余

- 各种服务器都要提供相应的冗余服务器以便在某台或某些服务器宕机时还能保证网站可以正常工作，同时也提供了灾难恢复的可能性。冗余是网站高可用性的重要保证。

2. 网站前端优化的技术有哪些？

- 浏览器访问优化
 - 减少HTTP请求数量：合并CSS、合并JavaScript、合并图片（CSS Sprite）

- 使用浏览器缓存：通过设置HTTP响应头中的Cache-Control和Expires属性，将CSS、JavaScript、图片等在浏览器中缓存，当这些静态资源需要更新时，可以更新HTML文件中的引用让浏览器重新请求新的资源
 - 启用压缩
 - CSS前置，JavaScript后置
 - 减少Cookie传输
- CDN加速：
 - CDN (Content Distribute Network) 的本质仍然是缓存，将数据缓存在离用户最近的地方，CDN通常部署在网络运营商的机房，不仅可以提升响应速度，还可以减少应用服务器的压力。当然，CDN缓存的通常都是静态资源。
 - 反向代理：
 - 反向代理相当于应用服务器的一个门面，可以保护网站的安全性，也可以实现负载均衡的功能，当然最重要的是它缓存了用户访问的热点资源，可以直接从反向代理将某些内容返回给用户浏览器。

3. 应用服务器优化技术有哪些？

1. **分布式缓存** 分布式缓存的本质就是内存中的哈希表，缓存主要用来存储那些读写比较高，变化很少的数据，这样应用从缓存中读取大大提高了访问效率。网站的数据访问符合二八定律，即80%的访问都集中在20%的数据上，如果能将这20%的数据缓存起来，那么系统的性能将得到显著的改善。

使用缓存的问题：

- 频繁修改的数据、数据不一致与脏读、缓存雪崩（可以采用分布式集群得以解决）、缓存预热、缓存穿透（可以采用布隆过滤算法解决）
2. **异步操作** 可以使用消息队列将调用异步化，通过异步化处理将短时间高并发产生的事件存储到消息队列中，从而起到削峰的作用。
 3. **使用集群**
 4. **代码优化**

多线程 解决线程安全问题，只要考虑以下方面

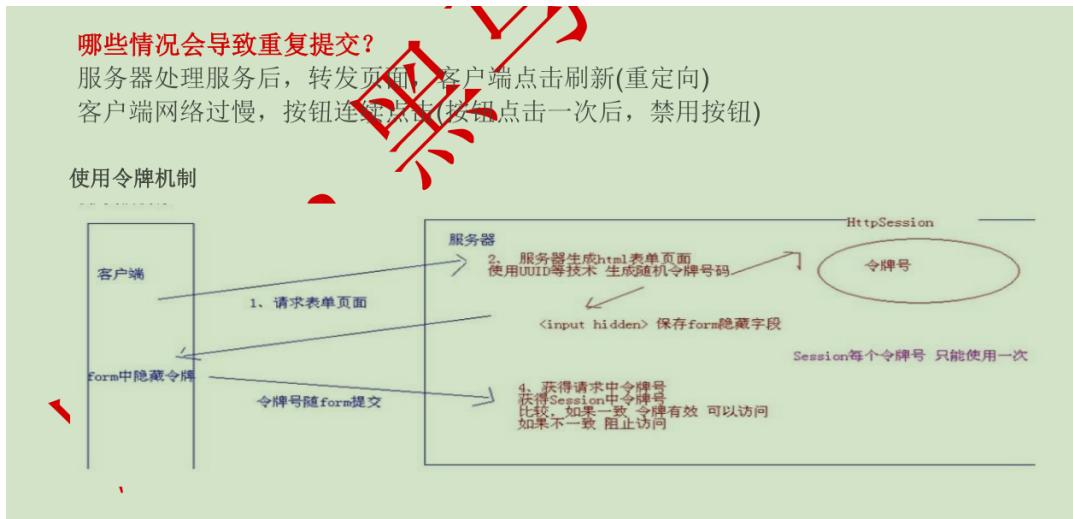
- 在方法内部创建对象，这样对象进入方法的线程创建，不会出现多线程同时访问一个对象问题。使用ThreadLocal 将对象与线程绑定也是很好的做法
- 资源在并发访问时使用合理的锁机制

非阻塞I/O 使用单线程和非阻塞I/O 是目前工人的比多线程的方式更能充分发挥服务器性能的应用模式，基于Node.js 构建的服务器就采用了这样的方式

资源复用

- 资源复用有两种方式 一是单例，而是对象池，我们使用的数据库连接池、线程池都是对对象池化技术，这是典型的用空间换取时间的策略
- 另一方面也实现对资源的复用，从而避免了不必要的创建和释放资源所带来的开销。

4. 如何防止表单重复提交



如何实现一个高并发的系统？

这道面试题涉及的知识点比较多，主要考察的是面试者的综合技术能力。高并发系统的设计手段有很多，主要体现在以下五个方面。

1) 前端优化

① 静态资源缓存：将活动页面上的所有可以静态的元素全部静态化，尽量减少动态元素；通过CDN、浏览器缓存，来减少客户端向服务器端的数据请求。② 禁止重复提交：用户提交之后按钮置灰，禁止重复提交。③ 用户限流：在某一时间段内只允许用户提交一次请求，比如，采取IP限流。

2) 中间层负载分发

可利用负载均衡，比如nginx等工具，可以将并发请求分配到不同的服务器，从而提高了系统处理并发的能力。nginx负载分发的五种方式：

① 轮询（默认） 每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器不能正常响应，nginx能自动剔除故障服务器。② 按权重（weight） 使用 weight 参数，指定轮询几率，weight 和访问比率成正比，用于后端服务器性能不均的情况，配置如下：

```
upstream backend {
    server 192.168.0.14 weight=10;
    server 192.168.0.15 weight=10;
}
```

③ IP哈希值(ip_hash) 每个请求按访问IP的哈希值分配，这样每个访客固定访问一个后端服务器，可以解决session共享的问题，配置如下：

```
upstream backend {
    ip_hash;
    server 192.168.0.14:88;
    server 192.168.0.15:80;
}
```

④ 响应时间(fair) 按后端服务器的响应时间来分配请求，响应时间短的优先分配，配置如下：

```
upstream backend {
    fair;
    server server1.com;
    server server2.com;
}
```

⑤ URL 哈希值 (url_hash) 按访问 url 的 hash 结果来分配请求，和 IP 哈希值类似。

```
upstream backend {  
    hash $request_uri;  
    server server1.com;  
    server server2.com;  
}
```

3) 控制层 (网关层)

限制同一个用户的访问频率，限制访问次数，防止多次恶意请求。

4) 服务层

① 业务服务器分离：比如，将秒杀业务系统和其他业务分离，单独放在高配服务器上，可以集中资源对访问请求抗压。② 采用 MQ（消息队列）缓存请求：MQ 具有削峰填谷的作用，可以把客户端的请求先导流到 MQ，程序在从 MQ 中进行消费（执行请求），这样可以避免短时间内大量请求，导致服务器程序无法响应的问题。③ 利用缓存应对读请求，比如，使用 Redis 等缓存，利用 Redis 可以分担数据库很大一部分压力。

5) 数据库层

① 合理使用数据库引擎 ② 合理设置事务隔离级别，合理使用事务 ③ 正确使用 SQL 语句和查询索引 ④ 合理分库分表 ⑤ 使用数据库中间件实现数据库读写分离 ⑥ 设置数据库主从读写分离

☆电商项目☆

所需技术点

1. Nginx 反向代理为什么可以提高网站性能

Nginx 会把 Request 在读取完整请求之前 Buffer 住，这样交给后端的就是一个完整的 HTTP 请求，从而提高后端的效率，而不是断断续续的传递（互联网的连接速度一般比较慢）。

同样 Nginx 把 Response Buffer 住，减轻后端的压力

举个例子：你有10个快递包裹，快递员一次把10个包裹送到你手上，你毫无压力。快递员一天分10次把10个包裹送你手上，你就崩溃了。Nginx 反向代理的作用就是你的收发室，收集了10个包裹在发给你

2.Nginx 和 Apache 各有什么优缺点？

Nginx

- 轻量级，比 apache 占用更少的内存及资源
- 高并发，Nginx 处理请求是异步非阻塞型，apache 是阻塞型的，高并发下 Nginx 能保持低资源低消耗 高性能
- 模块化设计，编写模块相对简单
- 社区活跃

Apache

- rewrite，比Nginx rewrite 强大
- 模块多
- bug少，相对于 Nginx
- 稳定

存在就是理由，需要性能的web 服务需要使用 Nginx，需要稳定，Apache

3.Nginx 多进程模型 是如何实现高并发的?

进程数和并发并不存在直接关系，取决于server 的工作方式

如果一个 server 采用一个进程负责一个 request 的方式，那么进程数就是并发数。那么显而易见的，就是会有很多进程在等待中。等什么？最多的就是等待网络传输。其缺点题主应该也感觉到了，此处不述。

而 nginx 的异步非阻塞工作方式正是利用了这点等待的时间。在需要等待的时候，这些进程就空闲出来待命了。因此表现为少数几个进程就解决了大量的并发问题。

apache 是如何利用的呢，简单来说：同样的 4 个进程，如果采用一个进程负责一个 request 的方式，那么，同时进来 4 个 request 之后，每个进程就负责其中一个，直至会话关闭。

期间，如果有第 5 个 request 进来了。就无法及时反应了，因为 4 个进程都没干完活呢，因此，一般有个调度进程，每当新进来了一个 request，就新开个进程来处理。

nginx 不这样，每进来一个 request，会有一个 worker 进程去处理。但不是全程的处理，处理到什么程度呢？处理到可能发生阻塞的地方，比如向上游（后端）服务器转发 request，并等待请求返回。

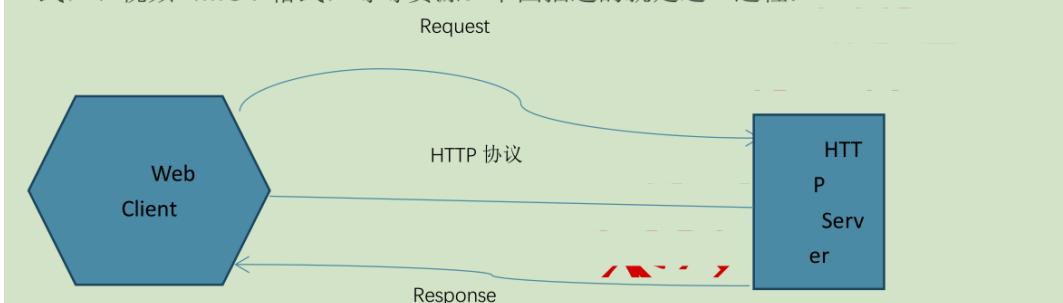
那么，这个处理的 worker 不会这么傻等着，他会在发送完请求后，注册一个事件：“如果 upstream 返回了，告诉我一声，我再接着干”。于是他就休息去了。此时，如果有 request 进来，他就可以很快再按这种方式处理。而一旦上游服务器返回了，就会触发这个事件，worker 才会来接手，这个 request 才会接着往下走。由于 web server 的工作性质决定了每个 request 的大部份生命都是在网络传输中，实际上花费在 server 机器上的时间片不多。这是几个进程就解决高并发的秘密所在。

webserver 刚好属于网络 io 密集型应用，不算是计算密集型。异步，非阻塞，使用 epoll，和大量细节处的优化。也正是 nginx 之所以然的技术基石。

4.tomcat、Nginx、apache 区别?

题主说的 Apache，指的应该是 Apache 软件基金会下的一个项目——Apache HTTP Server Project；Nginx 同样也是一款开源的 HTTP 服务器软件（当然它也可以作为邮件代理服务器、通用的 TCP 代理服务器）。

HTTP 服务器本质上也是一种应用程序——它通常运行在服务器之上，绑定服务器的 IP 地址并监听某一个 tcp 端口来接收并处理 HTTP 请求，这样客户端（一般来说是 IE, Firefox, Chrome 这样的浏览器）就能够通过 HTTP 协议来获取服务器上的网页（HTML 格式）、文档（PDF 格式）、音频（MP4 格式）、视频（MOV 格式）等等资源。下图描述的就是这一过程：



不仅仅是 Apache HTTP Server 和 Nginx，绝大多数编程语言所包含的类库中也都实现了简单的 HTTP 服务器方便开发者使用：

HttpServer (Java HTTP Server)

Python SimpleHTTPServer

使用这些类库能够非常容易的运行一个 HTTP 服务器，它们都能够通过绑定 IP 地址并监听 tcp 端口来提供 HTTP 服务。

Apache Tomcat 则是 Apache 基金会下的另外个项目，与 Apache HTTP Server 相比，Tomcat 能够动态的生成资源并返回到客户端。Apache HTTP Server 和 Nginx 都能够将某一个文本文件的内容通过 HTTP 协议返回到客户端，但是这个文本文件的内容是固定的——也就是说无论何时、任何人访问它得到的内容都是完全相同的，这样的资源我们称之为静态资源。动态资源则与之相反，在不同的时间、不同的客户端访问得到的内容是不同的，

例如：

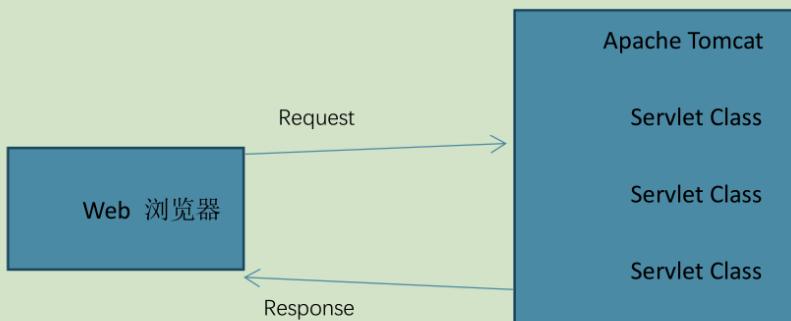
包含显示当前时间的页面

显示当前 IP 地址的页面

Apache HTTP Server 和 Nginx 本身不支持生成动态页面，但它们可以通过其他模块来支持（例如通过 Shell、PHP、Python 脚本程序来动态生成内容）。

如果想要使用 Java 程序来动态生成资源内容，使用这一类 HTTP 服务器很难做到。Java Servlet 技术以及衍生的 Java Server Pages 技术可以让 Java 程序也具有处理 HTTP 请求并且返回内容（由程序动态控制）的能力，Tomcat

正是支持运行 Servlet/JSP 应用程序的容器（Container）：



Tomcat 运行在 JVM 之上，它和 HTTP 服务器一样，绑定 IP 地址并监听 TCP 端口，同时还包含以下指责：

管理 Servlet 程序的生命周期

将 URL 映射到指定的 Servlet 进行处理

与 Servlet 程序合作处理 HTTP 请求——根据 HTTP 请求生成 HttpServlet Response 对象并传递给 Servlet 进行处理，将 Servlet 中的 HttpServlet Response 对象生成的内容返回给浏览器

虽然 Tomcat 也可以认为是 HTTP 服务器，但通常它仍然会**和 Nginx 配合在一起使用**：

动静态资源分离——运用 Nginx 的反向代理功能分发请求：

所有动态资源的请求交给 Tomcat，而静态资源的请求（例如图片、视频、CSS、JavaScript 文件等）则直接由 Nginx 返回到浏览器，这样能大大减轻 Tomcat 的压力。

负载均衡，当业务压力增大时，可能一个 Tomcat 的实例不足以处理，那么这时可以启动多个 Tomcat 实例进行水平扩展，而 Nginx 的负载均衡功能可以把请求通过算法分发到各个不同的实例进行处理

5.HttpClient 是什么

HttpClient不是一个浏览器。它是一个客户端的HTTP通信实现库。
HttpClient的目标是发送和接收HTTP报文。HttpClient不会去缓存内容，执行嵌入在HTML页面中的javascript代码，猜测内容类型，重新格式化请求/重定向URI，或者其它和HTTP运输无关的功能.它主要就是支持HTTP传输协议的.

6.HttpClient 的使用

我们知道，HTTP协议的连接方法有GET、POST、PUT和HEAD方式，在创建Method实例的时候可以更具具体的方法来创建。HttpClient的使用一般分如下几步：

- 1、创建HttpClient实例。
- 2、创建具体连接方法的实例。如POST方法创建PostMethod的实例，在实例化时从构造函数中传入待连接的URL地址。
- 3、对post的发送内容等信息进行配置
- 4、执行HttpClient的execute方法
- 5、如果返回的状态码正常，表明连接成功，可以读取response的内容

7.跨域问题

跨域：浏览器对于javascript的同源策略的限制。

以下情况都属于跨域：

跨域原因说明	示例
域名不同	www.jd.com 与 www.taobao.com
域名相同，端口不同	www.jd.com:8080 与 www.jd.com:8081
二级域名不同	item.jd.com 与 miaosha.jd.com

如果**域名和端口都相同，但是请求路径不同**，不属于跨域，如：

www.jd.com/item

www.jd.com/goods

http和https也属于跨域

而我们刚才是从 manage.leyou.com 去访问 api.leyou.com，这属于二级域名不同，跨域了。

5.1.为什么有跨域问题？

跨域不一定都会有跨域问题。

因为跨域问题是浏览器对于ajax请求的一种安全限制：**一个页面发起的ajax请求，只能是与当前页域名相同的路径**，这能有效的阻止跨站攻击。

因此：**跨域问题 是针对ajax的一种限制。**

但是这却给我们的开发带来了不便，而且在实际生产环境中，肯定会有许多台服务器之间交互，地址和端口都可能不同，怎么办？

5.2.解决跨域问题的方案

目前比较常用的跨域解决方案有3种：

- Jsonp

最早期的解决方案，利用script标签可以跨域的原理实现。

限制：

- 需要服务的支持
- 只能发起GET请求

- nginx反向代理

思路是：利用nginx把跨域反向代理为不跨域，支持各种请求方式

缺点：需要在nginx进行额外配置，语义不清晰

- CORS

规范化的跨域请求解决方案，安全可靠。

优势：

- 在服务端进行控制是否允许跨域，可自定义规则
- 支持各种请求方式

缺点：

- 会产生额外的请求

我们这里会采用cors的跨域方案。

5.3.cors解决跨域

5.3.1.什么是cors

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing）。

它允许浏览器向跨源服务器，发出[XMLHttpRequest](#)请求，从而克服了AJAX只能同源使用的限制。

CORS需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，IE浏览器不能低于IE10。

- 浏览器端：

目前，所有浏览器都支持该功能（IE10以下不行）。整个CORS通信过程，都是浏览器自动完成，不需要用户参与。

- 服务端：

CORS通信与AJAX没有任何差别，因此你不需要改变以前的业务逻辑。只不过，浏览器会在请求中携带一些头信息，我们需要以此判断是否允许其跨域，然后在响应头中加入一些信息即可。这一般通过过滤器完成即可。

5.3.2.原理有点复杂

浏览器会将ajax请求分为两类，其处理方案略有差异：简单请求、特殊请求。

5.3.2.1.简单请求

只要同时满足以下两大条件，就属于简单请求。：

- (1) 请求方法是以下三种方法之一：

- HEAD
- GET
- POST

(2) HTTP的头信息不超出以下几种字段：

- Accept
- Accept-Language
- Content-Language
- Last-Event-ID
- Content-Type: 只限于三个值 `application/x-www-form-urlencoded`、`multipart/form-data`、`text/plain`

当浏览器发现发起的ajax请求是简单请求时，会在请求头中携带一个字段：`origin`。

General

Request URL: `http://api.leyou.com/api/item/category/list?pid=0`
 Request Method: GET
 Status Code: 200
 Remote Address: 127.0.0.1:80
 Referrer Policy: no-referrer-when-downgrade

Response Headers (5)

Request Headers view source

Accept: application/json, text/plain, */*
 Accept-Encoding: gzip, deflate
 Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7
 Connection: keep-alive
 Host: api.leyou.com
Origin: http://manage.leyou.com
 Referer: http://manage.leyou.com/
 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36

Origin中会指出当前请求属于哪个域（协议+域名+端口）。服务会根据这个值决定是否允许其跨域。

如果服务器允许跨域，需要在返回的响应头中携带下面信息：

```
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
Content-Type: text/html; charset=utf-8
```

- Access-Control-Allow-Origin：可接受的域，是一个具体域名或者*（代表任意域名）
- Access-Control-Allow-Credentials：是否允许携带cookie，默认情况下，cors不会携带cookie，除非这个值是true

有关cookie：

要想操作cookie，需要满足3个条件：

- 服务的响应头中需要携带Access-Control-Allow-Credentials并且为true。
- 浏览器发起ajax需要指定withCredentials为true
- 响应头中的Access-Control-Allow-Origin一定不能为*，必须是指定的域名

5.3.2.2.特殊请求

不符合简单请求的条件，会被浏览器判定为特殊请求，例如请求方式为PUT。

预检请求

特殊请求会在正式通信之前，增加一次HTTP查询请求，称为“预检”请求（preflight）。

浏览器先询问服务器，当前网页所在的域名是否在服务器的许可名单之中，以及可以使用哪些HTTP动词和头信息字段。只有得到肯定答复，浏览器才会发出正式的XMLHttpRequest请求，否则就报错。

一个“预检”请求的样板：

```
OPTIONS /cors HTTP/1.1
Origin: http://manage.leyou.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: X-Custom-Header
Host: api.leyou.com
Accept-Language: en-US
Connection: keep-alive
User-Agent: Mozilla/5.0...
```

与简单请求相比，除了Origin以外，多了两个头：

- Access-Control-Request-Method：接下来会用到的请求方式，比如PUT
- Access-Control-Request-Headers：会额外用到的头信息

预检请求的响应

服务的收到预检请求，如果许可跨域，会发出响应：

```
HTTP/1.1 200 OK
Date: Mon, 01 Dec 2008 01:15:39 GMT
Server: Apache/2.0.61 (Unix)
Access-Control-Allow-Origin: http://manage.leyou.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: X-Custom-Header
Access-Control-Max-Age: 1728000
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Content-Length: 0
Keep-Alive: timeout=2, max=100
Connection: Keep-Alive
Content-Type: text/plain
```

除了Access-Control-Allow-Origin和Access-Control-Allow-Credentials以外，这里又额外多出3个头：

- Access-Control-Allow-Methods：允许访问的方式
- Access-Control-Allow-Headers：允许携带的头
- Access-Control-Max-Age：本次许可的有效时长，单位是秒，**过期之前的ajax请求就无需再次进行预检了**

如果浏览器得到上述响应，则认定为可以跨域，后续就跟简单请求的处理是一样的了。

5.3.3. 实现非常简单

虽然原理比较复杂，但是前面说过：

- 浏览器端都有浏览器自动完成，我们无需操心
- 服务端可以通过拦截器统一实现，不必每次都去进行跨域判定的编写。

事实上，SpringMVC已经帮我们写好了CORS的跨域过滤器：CorsFilter，内部已经实现了刚才所讲的判定逻辑，我们直接用就好了。

在leyou-gateway中编写一个配置类，并且注册CorsFilter：

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;

@Configuration
public class LeyouCorsConfig {
    @Bean
    public CorsFilter corsFilter() {
        //1.添加CORS配置信息
        CorsConfiguration config = new CorsConfiguration();
        //1) 允许的域，不要写*，否则cookie就无法使用了
        config.addAllowedOrigin("http://manage.leyou.com");
        //2) 是否发送Cookie信息
        config.setAllowCredentials(true);
        //3) 允许的请求方式
        config.addAllowedMethod("OPTIONS");
        config.addAllowedMethod("HEAD");
        config.addAllowedMethod("GET");
        config.addAllowedMethod("PUT");
        config.addAllowedMethod("POST");
        config.addAllowedMethod("DELETE");
        config.addAllowedMethod("PATCH");
        // 4) 允许的头信息
        config.addAllowedHeader("*");

        //2.添加映射路径，我们拦截一切请求
        UrlBasedCorsConfigurationSource configSource = new
UrlBasedCorsConfigurationSource();
        configSource.registerCorsConfiguration("/**", config);

        //3.返回新的CorsFilter.
        return new CorsFilter(configSource);
    }
}
```

FastDFS

什么是分布式文件系统

分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

通俗来讲：

- 传统文件系统管理的文件就存储在本机。
- 分布式文件系统管理的文件存储在很多机器，这些机器通过网络连接，要被统一管理。无论是上传或者访问文件，都需要通过管理中心来访问

什么是FastDFS

FastDFS是由淘宝的余庆先生所开发的一个轻量级、高性能的开源分布式文件系统。用纯C语言开发，功能丰富：

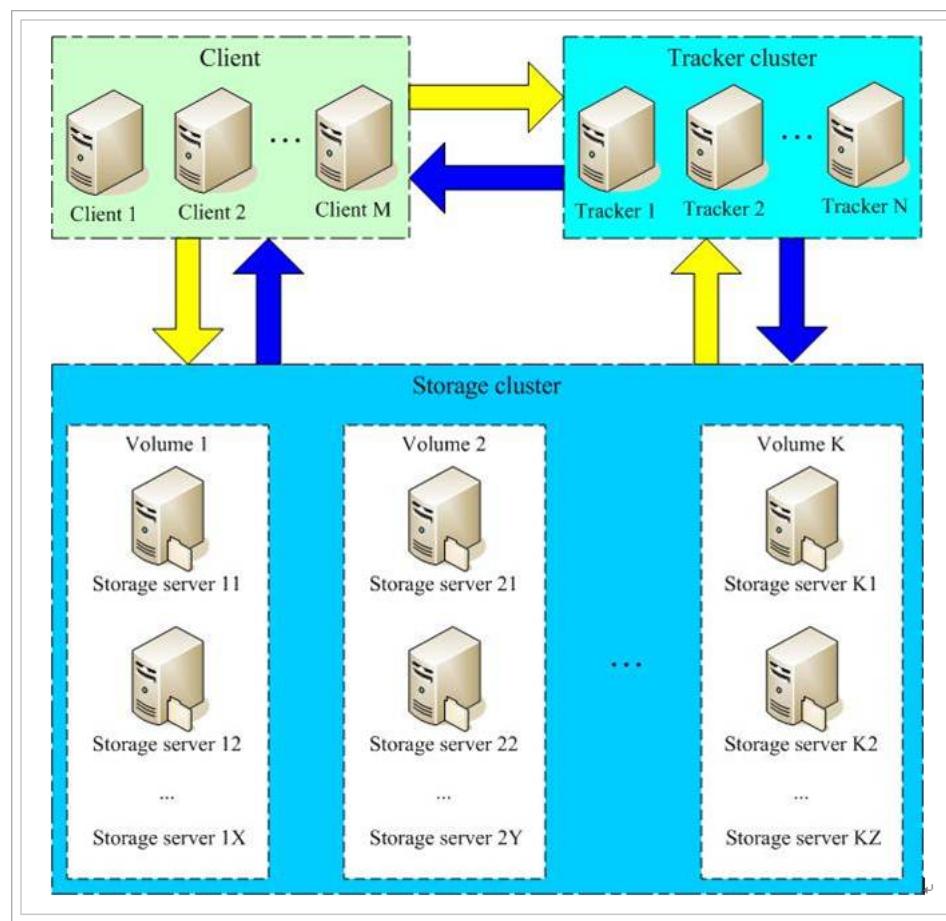
- 文件存储
- 文件同步
- 文件访问（上传、下载）
- 存取负载均衡
- 在线扩容

适合有大容量存储需求的应用或系统。同类的分布式文件系统有谷歌的GFS、HDFS（Hadoop）、TFS（淘宝）等。

FastDFS的架构

架构图

先上图：

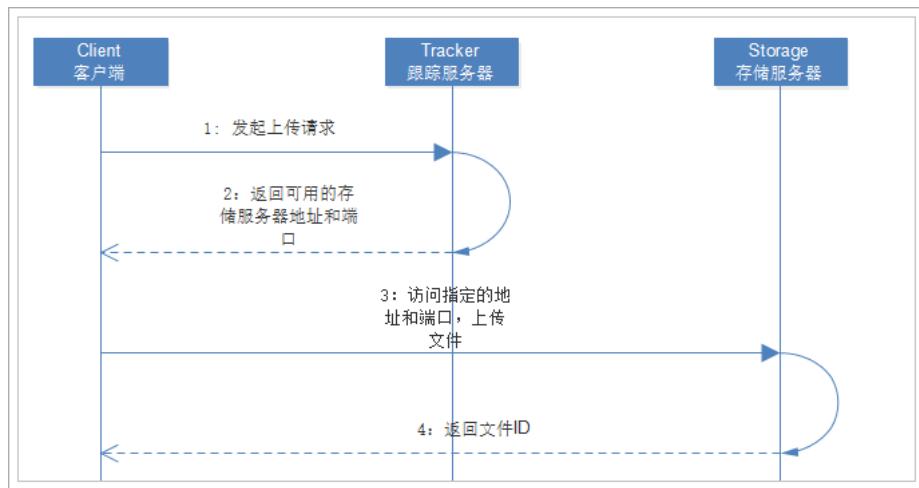


FastDFS两个主要的角色：Tracker Server 和 Storage Server。

- Tracker Server：跟踪服务器，主要负责调度storage节点与client通信，在访问上起负载均衡的作用，和记录storage节点的运行状态，是连接client和storage节点的枢纽。
- Storage Server：存储服务器，保存文件和文件的meta data（元数据），每个storage server会启动一个单独的线程主动向Tracker cluster中每个tracker server报告其状态信息，包括磁盘使用情况，文件同步情况及文件上传下载次数统计等信息
- Group：文件组，多台Storage Server的集群。上传一个文件到同组内的一台机器上后，FastDFS会将该文件即时同步到同组内的其它所有机器上，起到备份的作用。不同组的服务器，保存的数据不同，而且相互独立，不进行通信。
- Tracker Cluster：跟踪服务器的集群，有一组Tracker Server（跟踪服务器）组成。
- Storage Cluster：存储集群，有多个Group组成。

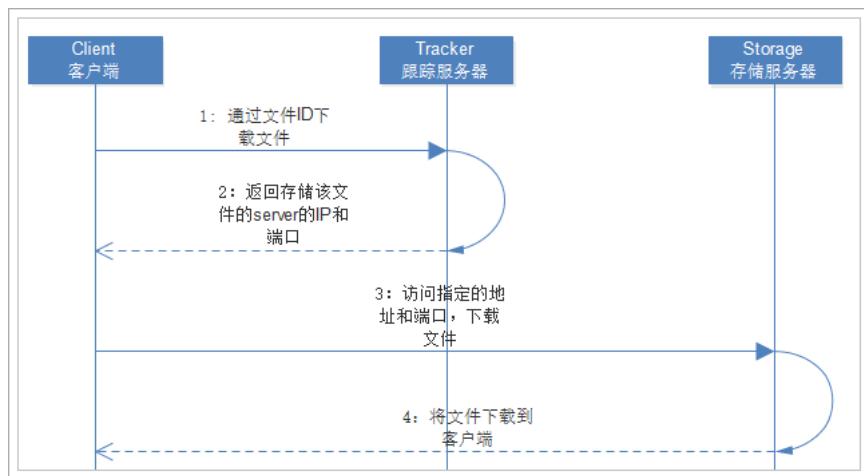
上传和下载流程

上传



1. Client 通过 Tracker server 查找可用的 Storage server。
2. Tracker server 向 Client 返回一台可用的 Storage server 的 IP 地址和端口号。
3. Client 直接通过 Tracker server 返回的 IP 地址和端口与其中一台 Storage server 建立连接并进行文件上传。
4. 上传完成, Storage server 返回 Client 一个文件 ID, 文件上传结束。

下载



1. Client 通过 Tracker server 查找要下载文件所在的 Storage server。
2. Tracker server 向 Client 返回包含指定文件的某个 Storage server 的 IP 地址和端口号。
3. Client 直接通过 Tracker server 返回的 IP 地址和端口与其中一台 Storage server 建立连接并指定要下载文件。
4. 下载文件成功。

8.RabbitMQ 的优点

1.2.1 基于 erlang 语言开发具有高并发的优点, 适合集群服务器。 ~~XXXX~~

1.2.2 健壮、稳定、易用、跨平台、支持多种语言、文档齐全。

1.2.3 有消息确认机制和持久化机制, 可靠性高。

1.2.4 开源

其他 MQ 的优势:

1.2.1 Apache ActiveMQ 曝光率最高, 但是可能会丢消息。

1.2.2 ZeroMQ 延迟很低、支持灵活拓扑, 但是不支持消息持久化和崩溃恢复。

9.JWT

JWT包含三部分数据：

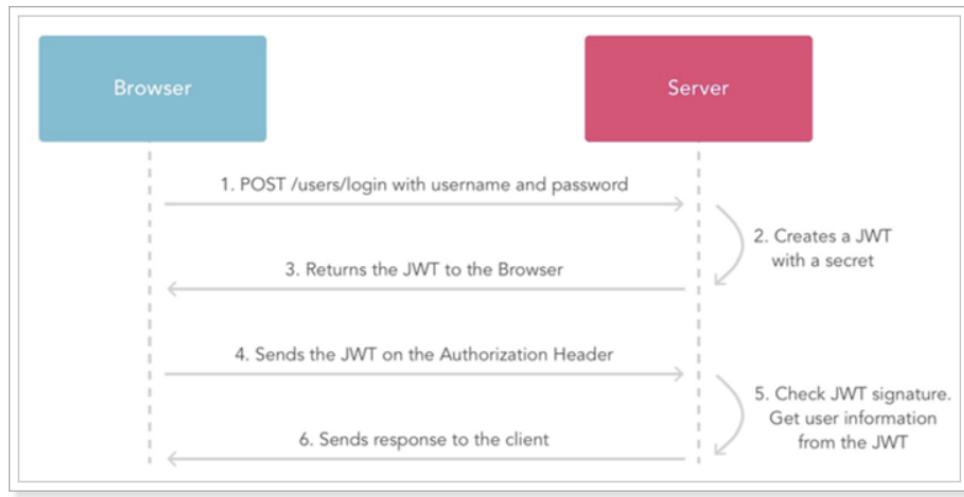
- Header：头部，通常头部有两部分信息：
 - 声明类型，这里是JWT我们会对头部进行base64编码，得到第一部分数据
- Payload：载荷，就是有效数据，一般包含下面信息：
 - 用户身份信息（注意，这里因为采用base64编码，可解码，因此不要存放敏感信息）
 - 注册声明：如token的签发时间，过期时间，签发人等这部分也会采用base64编码，得到第二部分数据
- Signature：签名，是整个数据的认证信息。一般根据前两步的数据，再加上服务的密钥（secret）（不要泄漏，最好周期性更换），通过加密算法生成。用于验证整个数据完整和可靠性



JWT+RSA非对称加密

1. 用户登录
2. 服务的认证，通过后根据私钥生成token
3. 将生成的token返回给浏览器
4. 用户每次请求携带token
5. 服务端利用公钥解读JWT签名，判断签名有效后，从payload中获取信息
6. 处理请求，返回相应结果

因为JWT签发的token 已经包含了用户的身份信息，并且每次请求都会携带，这样服务就无需保存用户信息，甚至无需去数据库查询，完全符合了Rest的无状态规范。



无状态登录流程

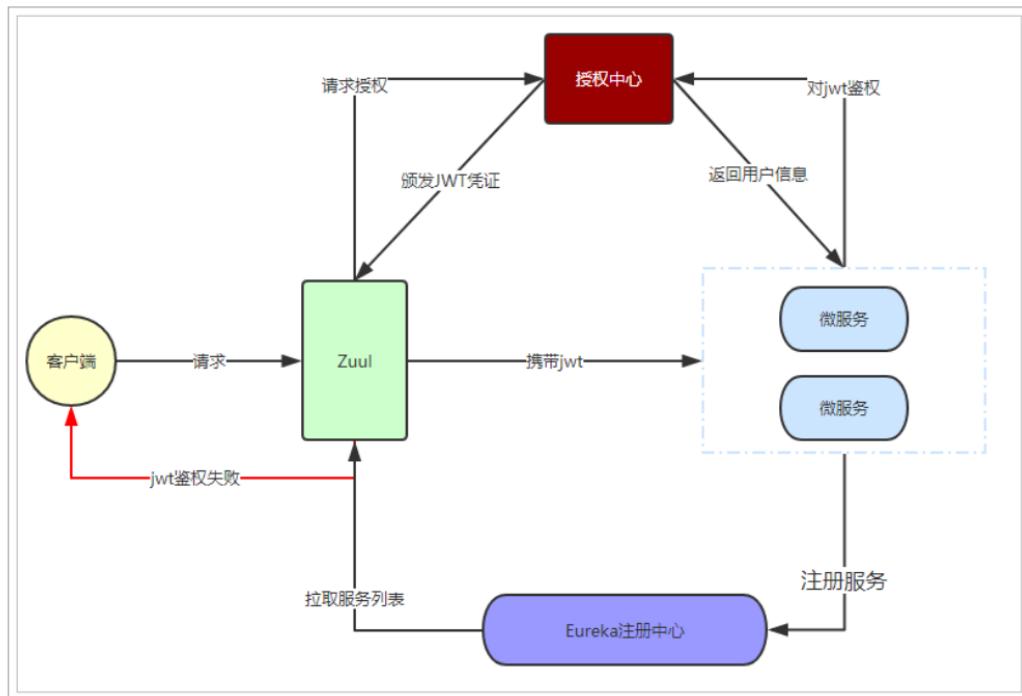
1. 当用户第一次请求服务时，服务端对用户进行信息认证（登录）
2. 认证通过，将用户信息进行加密形成token，返回客户端，作为登录凭证
3. 以后每次请求，客户端都携带认证的token
4. 服务端对token进行解密，判断是否有效。

10.结合Zuul的鉴权流程

我们逐步演进系统架构设计。需要注意的是：secret是签名的关键，因此一定要保密，我们放到鉴权中心保存，其它任何服务中都不能获取secret。

没有RSA加密时

在微服务架构中，我们可以把服务的鉴权操作放到网关中，将未通过鉴权的请求直接拦截，如图：



- 1、用户请求登录
- 2、Zuul将请求转发到授权中心，请求授权
- 3、授权中心校验完成，颁发JWT凭证
- 4、客户端请求其它功能，携带JWT
- 5、Zuul将jwt交给授权中心校验，通过后放行
- 6、用户请求到达微服务

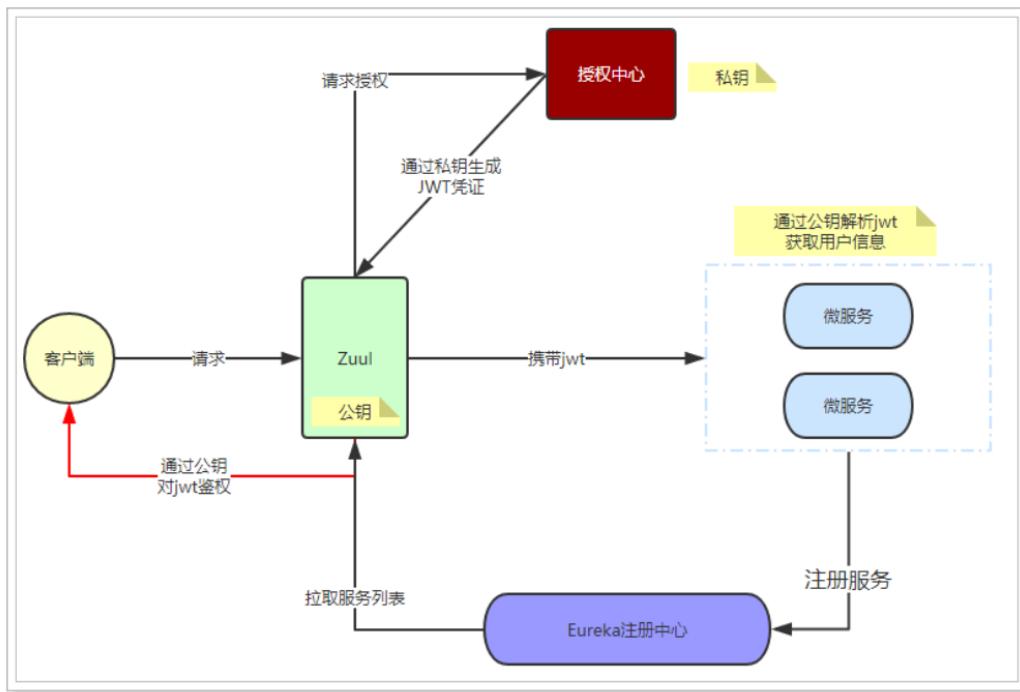
- 7、微服务将jwt交给鉴权中心，鉴权同时解析用户信息
- 8、鉴权中心返回用户数据给微服务
- 9、微服务处理请求，返回响应

发现什么问题了？

每次鉴权都需要访问鉴权中心，系统间的网络请求频率过高，效率略差，鉴权中心的压力较大。

结合RSA的鉴权

直接看图：



- 我们首先利用RSA生成公钥和私钥。私钥保存在授权中心，公钥保存在Zuul和各个信任的微服务
- 用户请求登录
- 授权中心校验，通过后用私钥对JWT进行签名加密
- 返回jwt给用户
- 用户携带JWT访问
- Zuul直接通过公钥解密JWT，进行验证，验证通过则放行
- 请求到达微服务，微服务直接用公钥解析JWT，获取用户信息，无需访问授权中心

电商项目面试问题

1. 简单介绍下你的项目

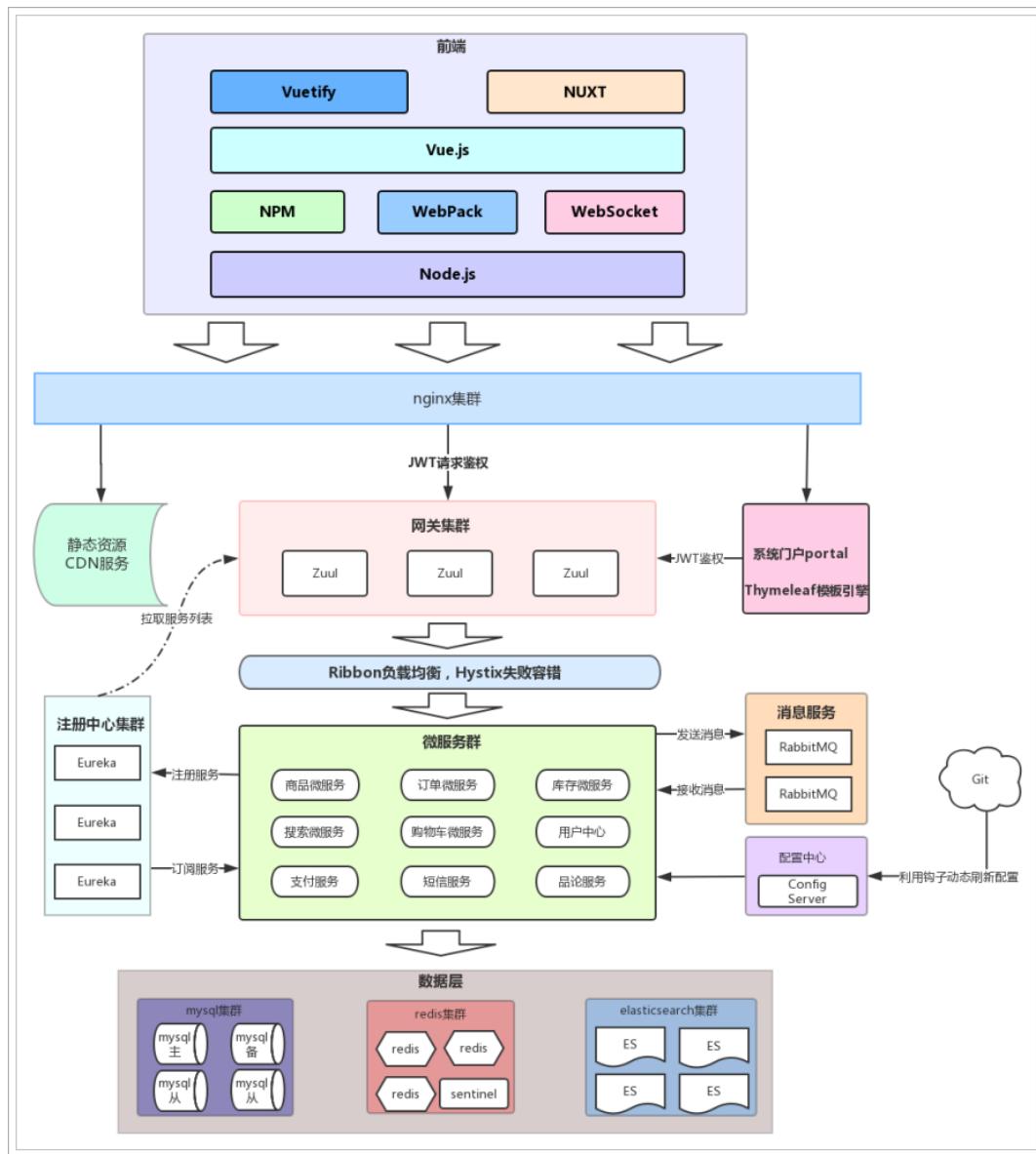
天天商城是一个全品类的 B2B2C 平台，类似淘宝、京东。商家可以申请入驻平台进行商品的销售，用户可以在商城 浏览商品、加入购物车、下单、以及参加秒杀等各种活动。

网站分为前台和后台。前台分为 单点登录、门户、搜索、商品详情页、购物车、下单、用户中心等模块。

系统有两个管理后台分为商家管理后台、运营管理后台。

- 商家管理后台为商家提供对商品的增加、删除、修改等功能。
- 运营管理后台审核商家的商品、入驻申请、商家统计等功能。

2.描述下你的系统架构



技术架构:

本项目使用 (Spring Boot + Spring Cloud + Mybatis) 框架开发。采用分布式的系统架构，前端使用Vue.js 等框架进行开发，前端请求经过 Nginx 转发。全部请求都要经过网关进行处理和权限的鉴定，不同权限的用户访问不同的微服务，比如商家管理系统和运维管理系统。请求再次由网关进行转发，使用Ribbon 负载均衡，并用Hystrix 进行熔断处理。微服务之间采用消息队列通信，持久层使用Mysql 和 Redis 进行数据存储。门户页面使用 Electricsearch 进行搜索。

功能架构:

后台管理系统：管理商品、订单、商品规格、用户管理等功能

前台系统：用户可以进行登录、搜索商品、下单、加入购物车

订单系统：提供下单、查询订单、修改订单、

搜索系统：提供商品的搜索功能

3.为什么使用分布式架构

1. 传统项目架构中，如果某个功能需要维护需要停掉整个项目进行维护，对于公司的损失很大。分布式项目采用模块化开发，维护一个模块不会影响其他模块的正常运行。
2. 处理海量数据时，传统架构比较乏力。分布式项目采用服务器集群、负载均衡、熔断等措施处理起来游刃有余。

4.你承担这个项目的哪些模块

1. 后台管理系统 商品的管理、商品规格参数管理、订单管理、会员管理、
2. 前台系统：使用HttpClient 和后台接口做交互
3. 登录系统：

5.这些模块的实现思路说一下

6.项目中哪些功能设计到了大数据访问？你是如何解决的？

7.项目中遇到的问题及解决办法？

redis 受物理内存限制，操作海量数据读写性能低； 采用redis 集群 和读写分离

HttpClient 耦合性高，后期我们采用 MQ

项目中用到了之前没有学过的技术； 私人时间主动学习

在开发环境中可以运行，放到服务器上不可以运行； 浏览器环境差异、服务器之间的差异、Redis 、Mysql 等版本差异

8.做完这个项目有什么收获

数据库方面：数据库设计是一个程序或软件设计的重要根基，好的设计减少了编程的复杂度，提高了性能和维护性。

团队合作方面：分工明确，不然两个人做了大量的重复工作，大大减少了开发效率。

命名规范：代码不是只有自己一个看的，当别人阅读自己写的代码时 见名知意

每完成一个模块都要进行多次测试，保证模块没有bug

遇到问题团队一起解决，一个人所想的东西是有限的，别人总会想到一些自己想不到的东西

9.你的项目是用什么构建的？多模块是如何划分的？为什么要这么做

我的这个项目使用 Maven 构建的，并使用了水平划分，这样层次清晰，代码可重用性高，易于维护。

水平划分 controller service dao pojo

垂直划分 订单模块、登录模块、商品模块等

优缺点：

- 垂直划分功能明确，层次不够清晰，代码重用性差
- 水平划分：层次清晰，代码重用性高，易于维护

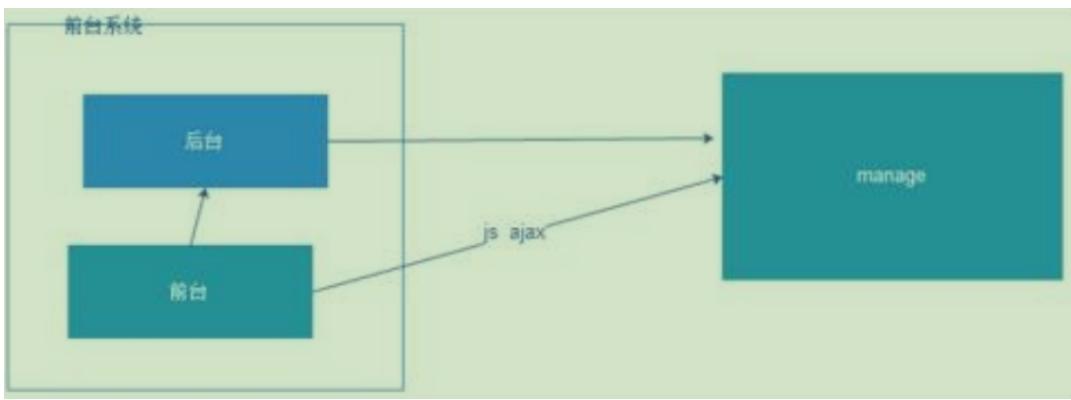
天天商城后台使用水平划分

10.你举得图片上传注意什么

对上传的文件做校验！

11.如何设计商品规格的

12.如何实现跨系统调用的



1.Ajax，走前台js，通过Jsonp来跨域

<http://www.cnblogs.com/yuzhongwusan/archive/2012/12/11/2812849.html>

13.单点系统的设计思想你了解么？他在系统中的作用是什么？

单点登录系统SSO(Single Sign On)在一个多系统共存的环境下，用户在一处登录后就可以不用在其他系统中登录，用户的一次登录能得到其他系统的信任。

实现单点登录就是如何解决存储信任和验证信任

第一种方式：cookie 实现本地存储，

用户首次登录，把cookie存储在本地中。用户再次登录检查携带的cookie是否和本地的一致。

缺点 不安全，不能实现跨域免登

第二种方式：在服务器存储

14.订单ID是怎么生成的？

用户ID+当前系统的时间戳

15.多台tomcat之间的session是怎么同步的

使用单点登录，使用redis存储session

16.如何解决并发问题

1. 前台首页静态化缓存服务器集群
2. 后台系统服务器集群缓存读写分离
3. 订单模块使用MQ进行流量限制

使用LVS提供的负载均衡技术实现高可用的服务器集群。

17.LVS是什么

使用LVS技术要达到的目标是：通过LVS提供的负载均衡技术和Linux操作系统实现一个高性能，高可用的服务器群集，它具有良好的可靠性、可扩展性和可操作性。从而以低廉的成本实现最优的服务性能。

LVS体系架构

使用LVS架设的服务器集群系统有三个部分组成：最前端的负载均衡层（Loader Balancer），中间的服务器群组层，用Server Array表示，最底层的数据共享存储层，用Shared Storage表示。在用户看来所有的应用都是透明的，用户只是在使用一个虚拟服务器提供的高性能服务。

18. 你们生产环境的服务器有多少台？

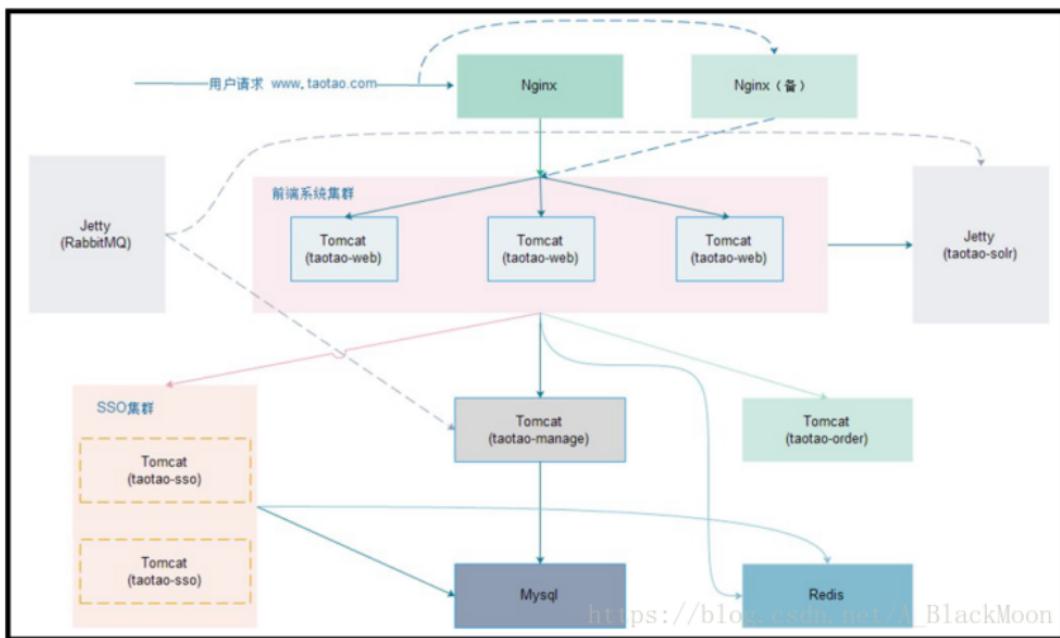
Nginx至少2台

Tomcat至少3台以上

数据库至少2台

Redis至少一台

19. 线上部署情况



20. 数据备份是怎么做的？有没有读写分离？

主从配置，一主多从，在主服务器写，从服务器读。

21. 你们服务器不止一台吧，那么你们的session是怎么同步的？

session存放在 redis 中，使用单点登录系统。

购物车实现思路

- 未登录：先写到 cookie 中，登录后写到数据库表中
- 已登录：直接写到数据库，而不写到 cookie

22. 你们用什么做支付的？请求超时如何处理？

支付宝接口，

1. 请求超时的话，一般重试 3 次，每次重试都要停顿一会，第一次停顿一秒 第二次停顿 2 秒，第三次停顿 3 秒
2. 给订单标识付款异常状态，写入日志
3. 写个定时任务，定时处理异常状态的订单

23. 返回超时却扣钱了怎么办？

你刚才不是说付款成功后支付宝会有数据返回吗？如果付款后易宝没有返回，或者返回超时了，但是钱又已经扣了，你怎么办？

1. 我们请求了支付宝，但是没有接收到响应，那么就认为该订单为没有支付成功，将订单设置为异常状态。

2. 使用定时处理任务
3. 做一个对账的任务，实时处理异常的订单

24.你们怎么做退款功能的，要多长时间才能把钱退回给用户？

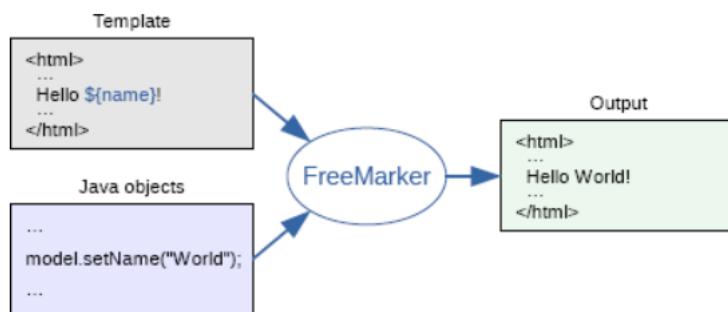
用户申请退款后，经过人工客服审核，审核通过将退款打给用户。用户付款的时候在数据库生成一个带有付款来源、付款金额、付款时间、付款订单号、钱款去向的表，当审核通过后，将从钱款去向里将付款金额转给付款来源。

时间由支付宝的处理

25.什么是 FreeMarker？

FreeMarker 是一款 模板引擎：即一种基于模板和要改变的数据，并用来生成输出文本(HTML网页，电子邮件，配置文件，源代码等)的通用工具。它不是面向最终用户的，而是一个Java类库，是一款程序员可以嵌入他们所开发产品的组件。

模板编写为FreeMarker Template Language (FTL)。它是简单的，专用的语言，不是像PHP那样成熟的编程语言。那就意味着要准备数据在真实编程语言中来显示，比如数据库查询和业务运算，之后模板显示已经准备好的数据。在模板中，你可以专注于如何展现数据，而在模板之外可以专注于要展示什么数据。



这种方式通常被称为 [MVC \(模型 视图 控制器\) 模式](#)，对于动态网页来说，是一种特别流行的模式。它帮助从开发人员(Java 程序员)中分离出网页设计师(HTML设计师)。设计师无需面对模板中的复杂逻辑，在没有程序员来修改或重新编译代码时，也可以修改页面的样式。

而FreeMarker最初的设计，是被用来在MVC模式的Web开发框架中生成HTML页面的，它没有被绑定到 Servlet或HTML或任意Web相关的东西上。它也可以用于非Web应用环境中。

FreeMarker 是 [免费的](#)，基于Apache许可证2.0版本发布。

26.如何解决加载页面慢的问题

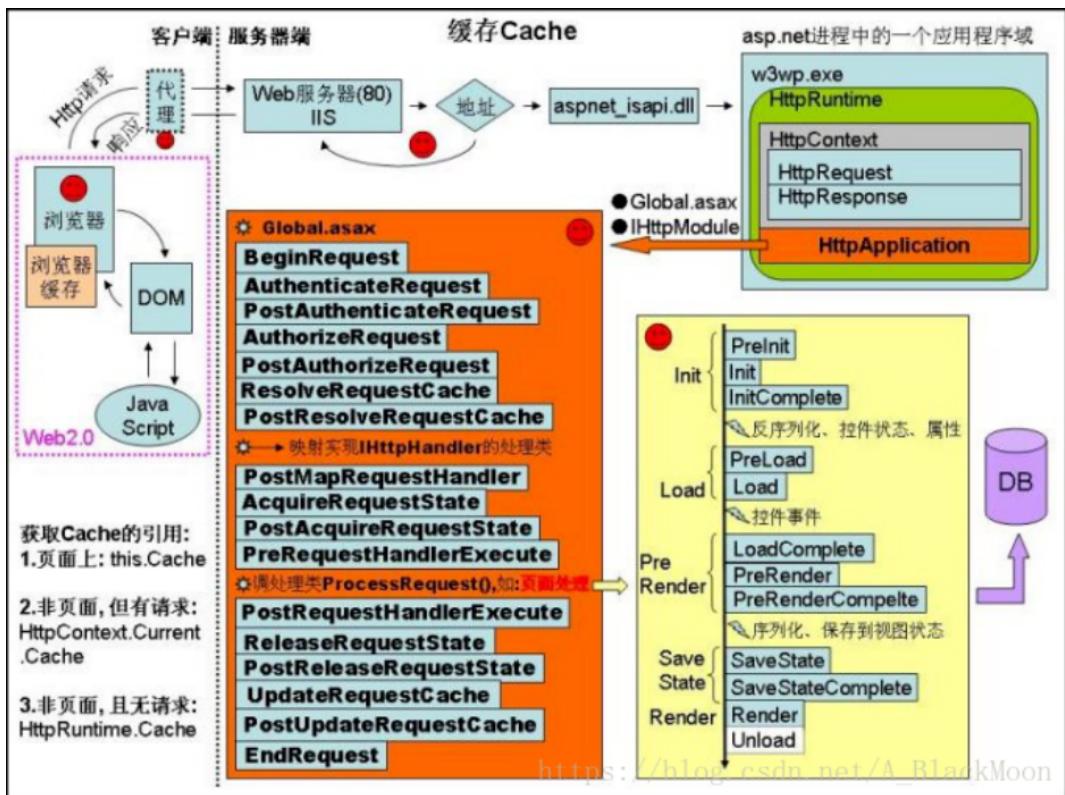
点一个链接访问到一个页面，这个页面上既有静态数据，又有动态数据（需要查数据库的），打开这个页面的时候就是很慢但是也能打开。怎么解决这个问题，怎么优化？

如果要静态页面的话 那就得用freemarker或者通过ajax异步,通过js操作异步刷新表单,通过js对返回结果组装成html。

缓存、动态页面静态化：

- 所谓缓存，是指将那些经常重复的操作结果暂时存放起来，在以后的执行过程中，只要使用前面的暂存结果即可。

那么在我们开发Web网站的过程中，**到底有多少工作可以采用用缓存呢？**或者说，**我们可以在哪些地方使用缓存呢？**见下图：下图是客户端浏览器和Web服务器之间的一次完整的通信过程，红色圆圈标示了可以采用缓存的地方。



首先, 最好的情况是客户端不发送任何请求直接就能获得数据, 这种情况下, 用于缓存的数据保存在客户端浏览器的缓存中.

其次, 在具有代理服务器的网络环境中, 代理服务器可以针对那些经常访问的网页制作缓存, 当局域网中第一台主机请求了某个网页并返回结果后, 局域网中的第二台主机再请求同一个网页, 这时代理服务器会直接返回上一次缓存的结果, 并不会向网络中的IIS服务器发送请求, 例如: 现在的连接电信和网通线路的加速器等. 但是代理服务器通常有自己专门的管理软件和管理系统, 作为网站开发人员对代理服务器的控制能力有限.

再次, 前面也说过, 当用户将请求地址发送到IIS服务器时, IIS服务器会根据请求地址选择不同的行为, 如: 对于*.aspx页面会走应用程序管道, 而对.html、.jpg等资源会直接返回资源,

那么我们可以把那些频繁访问的页面做成*.html文件, 这样用户请求*.html, 将不用再走应用程序管道, 因此会提升效率.

例如: 网站首页、或某些突发新闻或突发事件等, 可以考虑做成静态网页. 就拿“天气预报的发布页面”打比方: 天气预报的发布页面的访问用户非常多, 我们可以考虑将发布页做成静态的*.html网页, 之后在整个网站程序启动时, 在Global.asax的Application_Start事件处理器中, 创建子线程以实现每3个小时重新获取数据生成新的天气发布页面内容.

之后的asp.net的处理流程, 作为程序员我们是无法干涉的. 直到启动HttpApplication管道后, 我们才可以通过Global.asax或IHttpModule来控制请求处理过程, 在应用程序管道中适合做整页或用户控件的缓存. 如: 缓存热门页面, 我们可以自动缓存整个网站中访问量超过一定数值(阈值)的页面, 其中为了减小IO操作, 将缓存的页面放在内容中.

27.如果用户一直向购物车添加商品怎么办?

并且他添加一次你查询一次数据库? 互联网上用户那么多, 这样会对数据库造成很大压力你怎么办?

首先我们使用 redis 作为一个缓冲的内存数据库作为存储用户购物车的信息,当用户登陆以后,讲 redis 的信息写入关系型数据库.第二当用户是登录状态下,也是使用 redis 作为缓存,当用户做出关闭当前页面操作的时候,将 redis 的内容统一写入到数据库之中.为了提高 redis 的运行效率我还可以使用 redis 中的 hash 结构.来具体的找到某个字段,做小范围的修改.也可以减轻 redis 的压力.

购物车的设计方案:

购物车的实现不存在哪种方式更好,完全是根据公司和项目架构相关的,类似苏宁使用的是数据库存储,但是国美使用的就是Session,不同的软件架构和不同的业务需求对应的购物车存储也是不一样的

用数据库存你得给数据库造成多大的负担啊,而且对于购物车,这种需要实时操作的东西,数据库的访问量一大了,就容易出现并发错误,或者直接崩溃.

用Session确实效率很高,而且会话是针对各个连接的,所以便于管理,但是用Session也不是完美的,因为Session是有有效期的,根据服务器的设置不同而不一样长,如果你在购物的过程中Session超时了,那么购物车中的东西就会全没了.不知道你看过当当网的购物车没有,当你下线之后,再次上线,购物车中的东西还是存在的,这对于用户来说非常方便.所以如果你的服务器够强的话,你完全可以用一个静态变量来保存所有用户的购物车,比如用一个静态的Map,以IP作为Key,区分不同用户的购物车,这样就可以使用户在下线的情况下也可以保存购物车中的内容.这种方法实现过,只是没有用大量的并发访问测试其稳定性,但是一定是可行的。

采用存储过程将购物车**存储于数据库**相应表的方式,优点:数据稳定,不易丢失。缺点:效率低,增加数据库服务器负担。

变量 + Datatable 保存于客户端,优点:效率高,减轻数据库服务器负担。缺点:Session保存的变量容易丢失,但是一般情况下不会造成影响。

变量 + 购物车对象保存于客户端,这种方式以面向对象为指导思想,逻辑上具有一定的复杂性。优点:效率高,减轻数据库服务器负担,使用便捷。缺点:Session保存的变量容易丢失,但是一般情况下不会造成影响

未登录时可以加20个商品,登录后可以加50个。

28.商品详情页静态页面如何处理价格问题

京东商品详情页虽然仅是单个页面,但是其数据聚合源是非常多的,除了一些**实时性要求比较高**的如 价格、库存、服务支持等通过**AJAX异步加载**加载之外,

其他的数据都是在后端做**数据聚合**然后**拼装网页模板**的。整个京东有数亿商品,如果每次动态获取如上内容进行模板拼装,数据来源之多足以**造成性能无法满足要求**;最初的解决方案是生成静态页,但是静态页的最大的问题:

1. 无法迅速响应页面需求变更;
2. 很难做多版本线上对比测试。如上两个因素足以制约商品页的多样化发展,因此静态化技术不是很好的方案。

数据主要分为四种:

商品页基本信息、商品介绍(异步加载)、其他信息(分类、品牌、店铺等)、其他需要实时展示的数据(价格、库存等)。

而其他信息如分类、品牌、店铺是非常少的,完全可以放到一个占用内存很小的Redis中存储;

而商品基本信息我们可以**借鉴静态化技术将数据做聚合存储**,这样的好处是数据是原子的,而模板是随时可变的,吸收了静态页聚合的优点,弥补了静态页的多版本缺点;

另外一个非常严重的问题就是严重依赖这些相关系统，如果它们挂了或响应慢则商品页就挂了或响应慢；

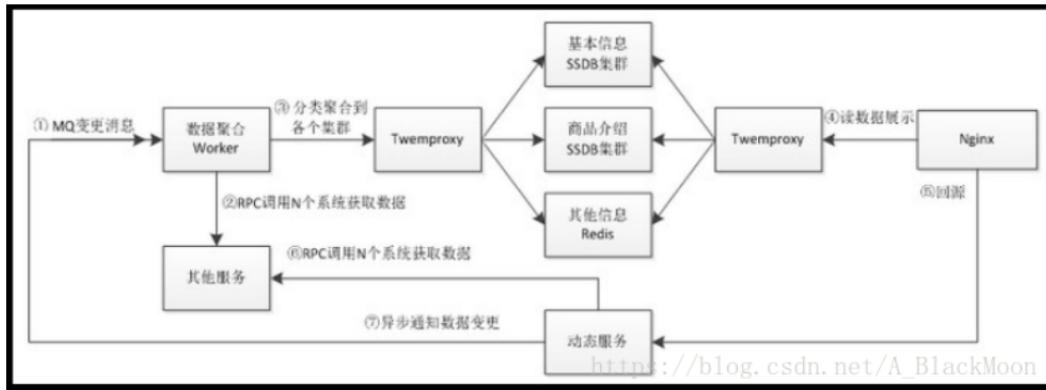
商品介绍我们也通过AJAX技术惰性加载（因为是第二屏，只有当用户滚动鼠标到该屏时才显示）；

而实时展示数据通过AJAX技术做异步加载

1、接收商品变更消息，做商品基本信息的聚合，即从多个数据源获取商品相关信息如图片列表、颜色尺码、规格参数、扩展属性等等，聚合为一个大的JSON数据做成数据闭环，以key-value存储；因为是闭环，即使依赖的系统挂了我们商品页还是能继续服务的，对商品页不会造成任何影响；

2、接收商品介绍变更消息，存储商品介绍信息；

3、介绍其他信息变更消息，存储其他信息



技术选型

MQ可以使用如Apache ActiveMQ；

Worker/动态服务可以通过如Java技术实现；

RPC可以选择如alibaba Dubbo；

KV持久化存储可以选择SSDB（如果使用SSD盘则可以选择SSDB+RocksDB引擎）或者ARDB（LMDB引擎版）；

缓存使用Redis；

SSDB/Redis分片使用如Twemproxy，这样不管使用Java还是Nginx+Lua，它们都不关心分片逻辑；

前端模板拼装使用Nginx+Lua；

数据集群数据存储的机器可以采用RAID技术或者主从模式防止单点故障；

因为数据变更不频繁，可以考虑SSD替代机械硬盘。

核心流程

1、首先我们监听商品数据变更消息；

2、接收到消息后，数据聚合Worker通过RPC调用相关系统获取所有要展示的数据，此处获取数据的来源可能非常多而且响应速度完全受制于这些系统，可能耗时几百毫秒甚至上秒的时间；

3、将数据聚合为JSON串存储到相关数据集群；

4、前端Nginx通过Lua获取相关集群的数据进行展示；商品页需要获取基本信息+其他信息进行模板拼装，即拼装模板仅需要两次调用（另外因为其他信息数据量少且对一致性要求不高，因此我们完全可以缓存到Nginx本地全局内存，这样可以减少远程调用提高性能）；当页面滚动到商品介绍页面时异步调用商品介绍服务获取数据；

5、如果从聚合的SSDB集群/Redis中获取不到相关数据；则回源到动态服务通过RPC调用相关系统获取所有要展示的数据返回（此处可以做限流处理，因为如果大量请求过来的话可能导致服务雪崩，需要采取保护措施），此处的逻辑和数据聚合Worker完全一样；然后发送MQ通知数据变更，这样下次访问时就可以从聚合的SSDB集群/Redis中获取数据了。

基本流程如上所述，主要分为Worker、动态服务、数据存储和前端展示；因为系统非常复杂，只介绍动态服务和前端展示、数据存储架构；Worker部分不做实现。

29.一个电商项目，在tomcat里面部署要打几个war包？

分布式系统，有几个独立的系统就需要打几个war包

30.你说你用了redis缓存，你redis存的是什么格式的数据，是怎么存的？

1 Key

Key 不能太长，比如1024字符，但antirez也不喜欢太短如"u:1000:pwd"，要表达清楚意思才好。他私人建议用":"分隔域，用"."作为单词间的连接，如"comment:1234:reply.to"。

Keys，返回匹配的key，支持通配符如 "keys a*"、"keys a?c"，但不建议在生产环境大数据量下使用。

2 String

最普通的key-value类型，说是String，其实是任意的byte[]，比如图片，最大512M。所有常用命令的复杂度都是O(1)，普通的Get/Set方法，可以用来做Cache，存Session，为了简化架构甚至可以替换掉Memcached。

3 Hash

Key-HashMap结构，相比String类型将这整个对象持久化成JSON格式，Hash将对象的各个属性存入Map里，可以只读取/更新对象的某些属性。

这样有些属性超长就让它一边呆着不动，另外不同的模块可以只更新自己关心的属性而不会互相并发覆盖冲突。

另一个用法是土法建索引。比如User对象，除了id有时还要按name来查询。

可以有如下的数据记录：

```
(String) user:101 -> {"id":101,"name":"calvin"...}  
(String) user:102 -> {"id":102,"name":"kevin"...}  
(Hash) user:index-> "calvin"->101, "kevin" -> 102
```

底层实现是hash table，一般操作复杂度是O(1)，要同时操作多个field时就是O(N)，N是field的数量。

4 List

List是一个双向链表，支持双向的Pop/Push，江湖规矩一般从左端Push，右端Pop——LPush/RPop，而且还有Blocking的版本BLPop/BRPop，客户端可以阻塞在那直到有消息到来，所有操作都是O(1)的好孩子，可以当Message Queue来用。

在消息队列中，并没有JMS的ack机制，如果消费者把job给Pop走了又没处理完就死机了怎么办？

解决方法之一是加多一个sorted set，分发的时候同时发到list与sorted set，以分发时间为score，用户把job做完了之后要用ZREM消掉sorted set里的job，并且定时从sorted set中取出超时没有完成的任务，重新放回list。

另一个做法是为每个worker多加一个的list，弹出任务时改用RPopLPush，将job同时放到worker自己的list中，完成时用LREM消掉。

如果集群管理(如zookeeper)发现worker已经挂掉，就将worker的list内容重新放回主list。

5 Set

Set就是Set，可以将重复的元素随便放入而Set会自动去重，底层实现也是hash table。

SAdd/SRem/SIsMember/SCard/SMove/SMembers，各种标准操作。除了SMembers都是O(1)。

SInter/SInterStore/SUnion/SUnionStore/SDiff/SDiffStore，各种集合操作。交集运算可以用来显示在线好友(在线用户 交集 好友列表)，共同关注(两个用户的关注列表的交集)。

O(N)，并集和差集的N是集合大小之和，交集的N是小的那个集合的大小*2。

6 Sorted Set

有序集，元素放入集合时还要提供该元素的分数。

ZRange/ZRevRange，按排名的上下限返回元素，正数与倒数。

ZRangeByScore/ZRevRangeByScore，按分数的上下限返回元素，正数与倒数。

ZRemRangeByRank/ZRemRangeByScore，按排名/按分数的上下限删除元素。

ZCount，统计分数上下限之间的元素个数。

ZRank/ZRevRank，显示某个元素的正倒序的排名。

ZScore/ZIncrby，显示元素的分数/增加元素的分数。

ZAdd/Add)/ZRem(Remove)/ZCard(Count)，ZInsertStore(交集)/ZUnionStore(并集)，Set操作，与正牌Set相比，少了IsMember和差集运算。

Sorted Set的实现是hash table(element->score，用于实现ZScore及判断element是否在集合内)，和skip list(score->element,按score排序)的混合体。

skip list有点像平衡二叉树那样，不同范围的score被分成一层一层，每层是一个按score排序的链表。

ZAdd/ZRem是O(log(N))，ZRangeByScore/ZRemRangeByScore是O(log(N)+M)，N是Set大小，M是结果/操作元素的个数。

可见，原本可能很大的N被很关键的Log了一下，1000万大小的Set，复杂度也只是几十不到。

当然，如果一次命中很多元素M很大那谁也没办法了。

秒杀系统面试总结

商品超卖问题

在商品表中添加一个版本号字段，减库存业务时，通过商品id获取商品的版本号，在减库存操作时先判断查询的verison 是否一致，一致则减库存成功。写入秒杀订单。

```
update set seckill_goods stock = stock - 1, version = version + 1  
where id = #{goodsId} and stock > 0 and version = #{version}
```

令牌桶算法

限流是对某一时间窗口内的请求数进行限制，保持系统的可用性和稳定性，防止因流量暴增而导致的系统运行缓慢或宕机。常用的限流算法有令牌桶和漏桶，而Google开源项目Guava中的RateLimiter使用的就是令牌桶控制算法。

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流

- 缓存：缓存的目的是提升系统访问速度和增大系统处理容量
- 降级：降级是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行
- 限流：限流的目的是通过对并发访问/请求进行限速，或者对一个时间窗口内的请求进行限速来保护系统，一旦达到限制速率则可以拒绝服务、排队或等待、降级等处理

我们经常在调别人的接口的时候会发现有限制，比如微信公众平台接口、百度API Store、聚合API等等这样的，对方会限制每天最多调多少次或者每分钟最多调多少次

我们自己在开发系统的时候也需要考虑到这些，比如我们公司在上传商品的时候就做了限流，因为用户每一次上传商品，我们需要将商品数据同到到美团、饿了么、京东、百度、自营等第三方平台，这个工作量是巨大，频繁操作会拖慢系统，故做限流。

以上都是题外话，接下来我们重点看一下令牌桶算法

整体流程

- 前端页面采用隐藏秒杀地址和使用随机秒杀地址来防止用户恶意刷接口，点击秒杀时会让用户输入数字计算验证码，输入正确后进入后端业务逻辑。

后端业务逻辑分为生产者和消费者。

生产者

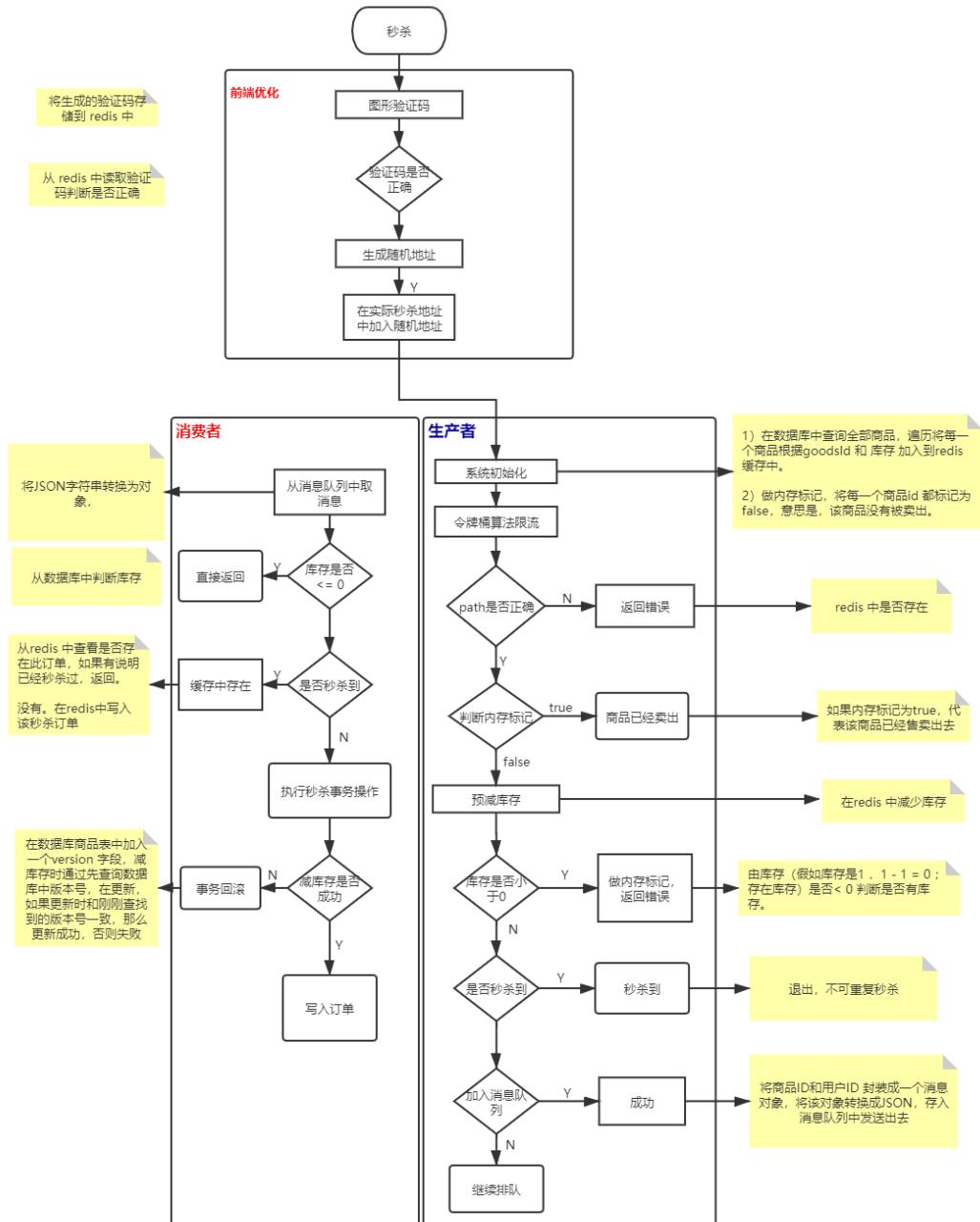
1. 生产者首先进行预加载，查询所有的商品列表，把每一个商品根据商品ID 和 库存容量加入到redis 中，其次做一个内存标记，使用hashMap，将每一个商品id 都标记为 false。
2. 使用令牌桶算法对其进行限流，保证系统的稳定性，防止流量暴增导致系统不可用
3. 判断刚刚随机生成的路径是否正确，然后在判断该商品是否已经卖出根据内存标记，如果都满足那么在redis中预减库存（因为初始化时，每一个商品Id和库存都存入redis中）
4. 判断库存是否小于 0，如果不小于0说明减库存成功，
5. 接着判断是否秒杀成功，从redis 中根据用户id 和商品id查询，如果存在说明该用户已经秒杀，不能重复秒杀
6. 加入MQ，将用户id 和商品ID封装成对象序列化成JSON 字符串。

消费者

1. 从队列中数据，将JSON字符串转换为对象，取出用户ID 和商品Id
2. 从数据库中判断库存是否 <= 0,如果小于 0 说明库存为0直接返回。
3. 从缓存中查看该订单是否秒杀到，如果秒杀到直接返回。没有秒杀到在缓存中写入该秒杀订单

4. 执行秒杀事务，先判断减库存是否成功，如果成功写入秒杀订单。

秒杀系统流程图



秒杀模块

1. 简单介绍下你的项目

我的这个秒杀系统采用 redis + Rabbit MQ 实现的，前端页面通过隐藏秒杀地址、秒杀加验证码分散用户的请求，对接口进行防刷限制，防止恶意攻击。一开始sql语句没有加入库存 > 0 限制，导致商品超卖问题产生，后来在改进sql的基础上加了乐观锁（每次秒杀时会根据订单id查询数据库中的版本号，如果当前版本号和数据库中的一样则进行减库存操作，否则进行循环再一次判断，设置最大冲突次数为5，超过则判定失败。）又有瓶颈，给页面加缓存、对象缓存，发生缓存雪崩（为键设置不同的过期时间），缓存预热，错开缓存失效时间。最后发现记录订单在减库存会减少行级锁等待时间。

2.缓存雪崩什么情况下会发生？如何避免？

1是秒杀开始时缓存中还没有缓存，解决办法 缓存预热。2.当多个商品缓存同时过期失效，此时会同时查数据，导致数据库宕机。解决办法 错开缓存失效时间。

3.更新数据库的同时为什么不马上更新缓存，而是删除缓存？

主要有几下原因，

1. 如果先更新数据库在删除缓存会导致脏数据的产生，在更新数据库时，其他线程会认为缓存中有数据不去访问数据库，这样造成了缓存不一致问题。
2. 对于一些访问量不大的订单，没有必要把这些订单放到缓存中浪费时间

4.秒杀地址在开始前为什么不应暴露给用户？

如果用户直到秒杀地址会提前写好脚本进行抢购了，这样不公平

5.那比如说啊，我现在是个黑客，我在秒杀开始时写好了脚本，运行一万个线程获取秒杀地址，这样是不是也不公平呢？

用redis 缓存这个用户id 和ip 地址，一个用户和ip 只能访问访问3次

6.那我可不可以创建一个 ip 代理池和很多用户来抢购呢？假设我有很多手机号的账户。

这时就要利用redis geo 存储地理位置判断每个用户的地理位置距离，如果地理位置相同且超过一定界限后认为此用户存在刷单嫌疑，禁止他购买。

7.我把减库存生成订单再在一个事务中，都操作成功则认为秒杀成功

8.订单表和商品库存表是在一个数据库的吧，那如果在不同的数据库中呢？

第一种方法：使用分布式锁

实现思路：使用唯一约束，可以保证多次提交只有一次成功，而成功的这次就代表了获取到锁

第二种方法：使用消息队列

实现思路：通过异步处理提高系统性能（削峰、减少响应时间）、降低系统耦合性

9.你有没有考虑到一种情况，假如说你的缓存刚刚失效，大量流量就来查缓存，你的数据库会不会炸？

在数据库前使用消息队列进行削峰，保证打到数据库的流量保持在可允许的范围内

10.你能说说 NIO么

当客户端连接时，服务端会通过ServerSocketChannel 得到 SocketChannel。

Selector 进行监听 select 方法 返回有事件发生的通道个数

将 SocketChannel 注册到 Selector 上，一个Selector 可以注册多个 SocketChannel

注册完成返回一个 SelectionKey，和 Selector 关联

如果有事件发生，进一步得到 各个 selectionKey

再通过 selectionKey 反向获取 SocketChannel

通过得到的channel 完成业务处理。

11. 说说进程切换时操作系统都会发生什么？

12. 设计一个聊天系统使用 Tcp 还是 UDP

使用TCP协议和客户端保持连接（长连接）从客户登录到下线，在客户端与服务器之间一直保持一个TCP连接，两者之间可以随时相互通信，信息的传输是可靠的。这对于小规模的公司内部即时通讯可以使用。**但对于大规模的应当采用UDP进行网络传输。**因为大量的TCP长连接，网关承受不了。

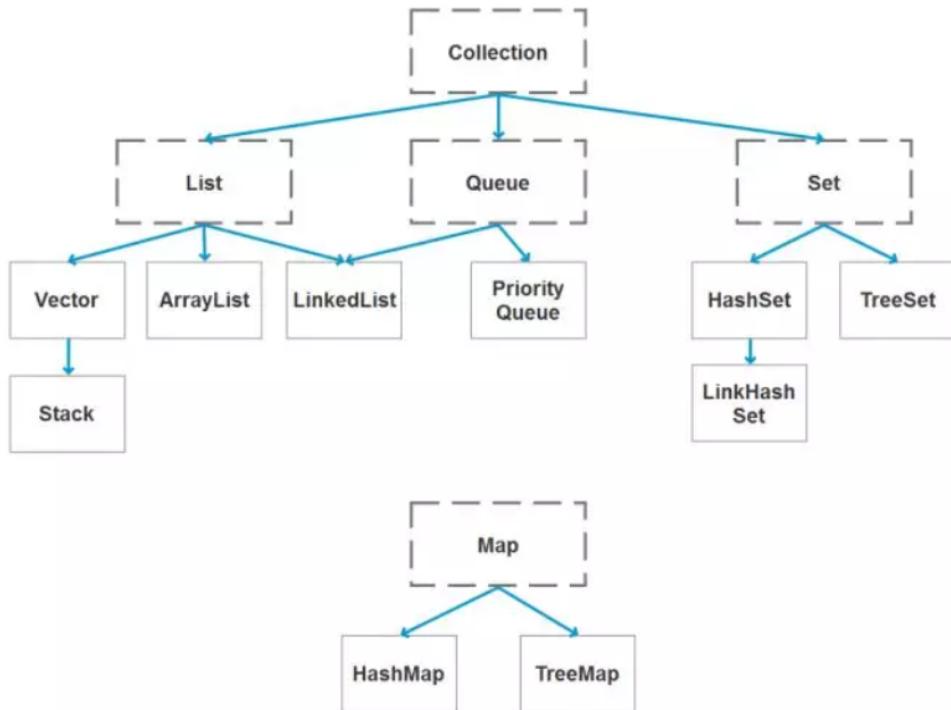
腾讯采用了上层服务来保证可靠传输。（如果客户端使用UDP协议发出消息后，服务器收到该消息，需要使用UDP协议发回一个应答消息，用以保证信息的无遗漏传输）

总体来说：小规模TCP 大规模UDP

13. 说说你的分布式事务解决方案？

☆ 容器 ☆

1. java 容器都有哪些？



Java集合的快速失败机制 “fail-fast”？

是java集合的一种错误检测机制，当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。

例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 `ConcurrentModificationException` 异常，从而产生fail-fast机制。

原因：迭代器在遍历时直接访问集合中的内容，并且在遍历过程中使用一个 `modCount` 变量。

集合在被遍历期间如果内容发生变化，就会改变`modCount`的值。每当迭代器使用

`hashNext()/next()`遍历下一个元素之前，都会检测`modCount`变量是否为`expectedModCount`值，是的话就返回遍历；否则抛出异常，终止遍历。

解决办法：

1. 在遍历过程中，所有涉及到改变modCount值得地方全部加上synchronized。
2. 使用CopyOnWriteArrayList来替换ArrayList

2. Collection 和 Collections 有什么区别？

- java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。
 - Collection接口在Java类库中有很多具体的实现。
 - Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。
- Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

comparable 和 comparator的区别？

- comparable来自java.lang包。他有一个compareTo (Object obj) 方法用来排序
- comparator 来自java.util 包，他有一个compare (Object obj1, Object obj2) 方法用来排序

Collection 和 Collections 有什么区别？

- java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。
- Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

3. List、Set、Map 之间的区别是什么？

比较	List	Set	Map
继承接口	Collection	Collection	
常见实现类	AbstractList(其常用子类有ArrayList、LinkedList、Vector)	AbstractSet(其常用子类有HashSet、LinkedHashSet、TreeSet)	HashMap、HashTable
常见方法	add()、remove()、clear()、get()、contains()、size()	add()、remove()、clear()、contains()、size()	put()、get()、remove()、clear()、containsKey()、containsValue()、keySet()、values()、size()
元素	可重复	不可重复(用 equals() 判断)	不可重复
顺序	有序	无序(实际上由HashCode决定)	
线程安全	Vector线程安全		Hashtable线程安全

list 有序，存入和输出的顺序有序，元素可以重复，底层可以由ArrayList、LinkedList双向链表和Vector实现。

set 无序，存入和输出的顺序无序，元素不可以重复。底层由HashSet、LinkedHashSet、和 TreeSet 实现

Map 是一个键值对集合，存储键和值之间的映射。key 是唯一的。

- **HashMap**: JDK 1.8 之前 HashMap 由数组+链表组成，数组是主体，链表是为了解决hash冲突而存在的（使用拉链法）。当链表长度大于8时，由链表转换为红黑树，减少搜索次数

- **LinkedHashMap** : LinkedHashMap 继承自HashMap。所以它的底层实现和HashMap相同。LinkedHashMap 在上面的结构上增加了一个双向链表，可以保证插入有序
- **HashTable** : 数组+链表组成，数组是HashMap的主体，链表则是解决hash冲突的
- **TreeMap** 由红黑树组成

♡ HashMap

说一下 HashMap 的实现原理？

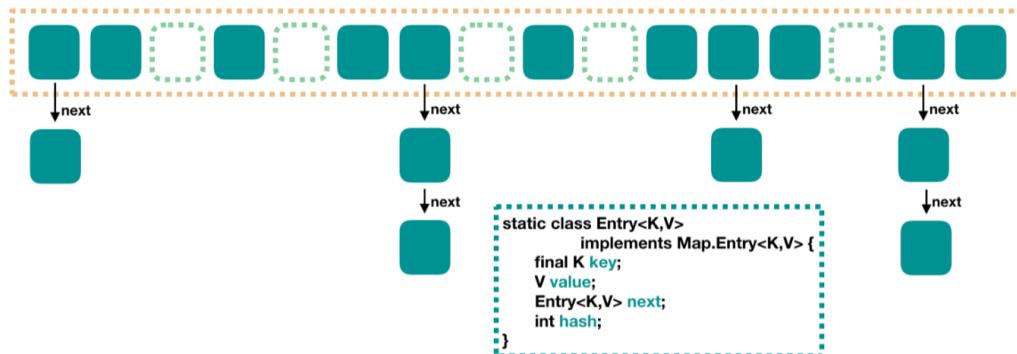
HashMap概述：HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

HashMap是一个“链表散列”的数据结构，即数组和链表的结合体。

JDK7

- 当我们往HashMap中put元素时，首先根据key的hashCode重新计算hash值，根据hash值得到这个元素在数组中的位置(下标)，如果该数组在该位置上已经存放了其他元素，那么在这个位置上的元素将以链表的形式存放，**新加入的放在链头，最先加入的放入链尾**。如果数组中该位置没有元素，就直接将该元素放到数组的该位置上。
- 采用数组加链表实现

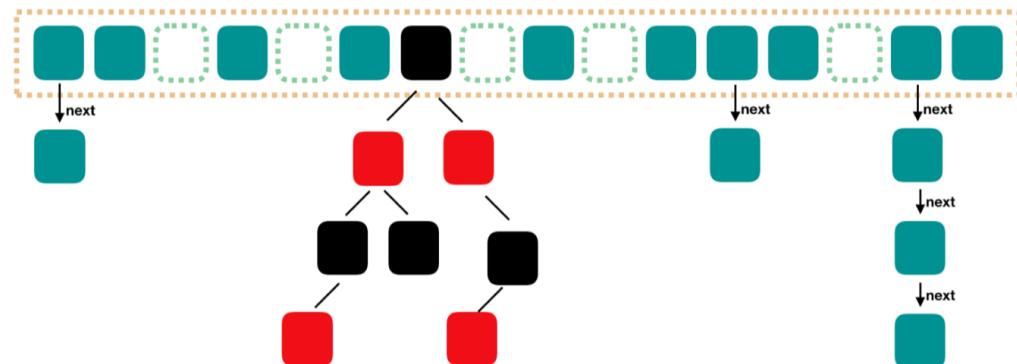
Java7 HashMap 结构



JDK8

- 需要注意Jdk 1.8中对HashMap的实现做了优化，当链表中的节点数据超过八个之后，该链表会转为红黑树来提高查询效率，从原来的O(n)到O(logn)
- 采用数组加+链表+红黑树实现

Java8 HashMap 结构



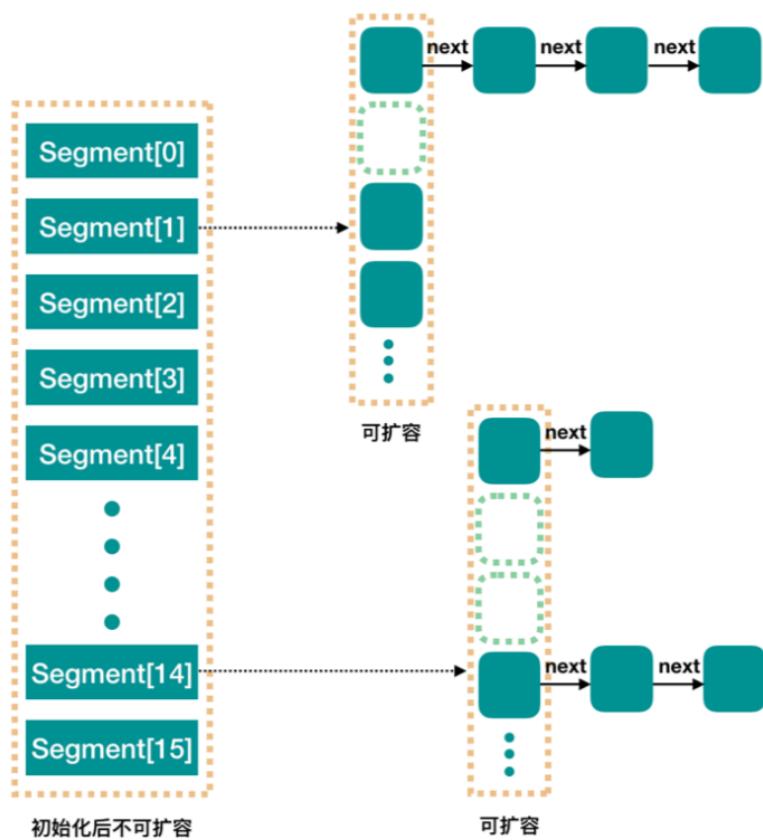
为什么HashMap线程不安全？

- 同时put碰撞导致数据丢失
 - 同时put 扩容导致数据丢失
 - 在多线程同时扩容的时候会造成链表的互相指向，导致死循环
 - 死循环造成CPU 100% 在JDK7 之前存在

说一下 ConcurrentHashMap 的实现原理

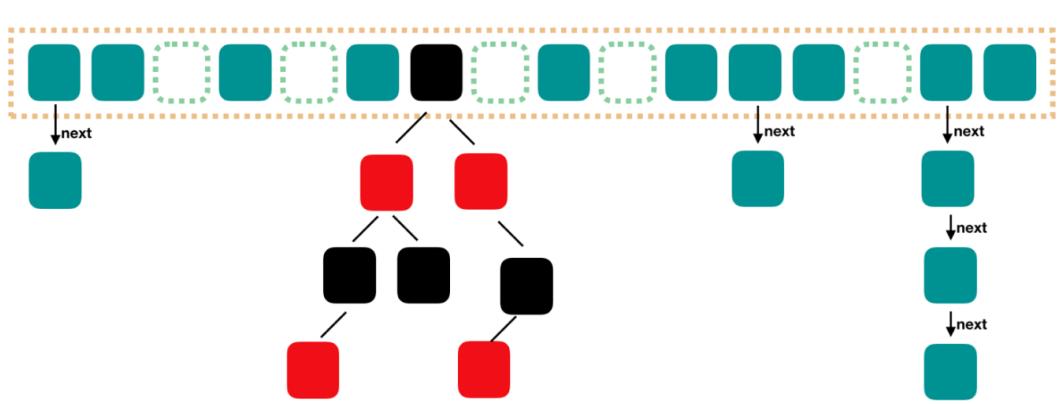
JDK7

- 最外层是多个segment，每个segment的底层数据结构是与HashMap类似，仍然是数组 + 链表组成的拉链法
 - 每个segment独立上**ReentrantLock** 锁，每个segment之间互不影响，提高了并发效率
 - ConcurrentHashMap默认有16个segments，所以最多可支持16个线程并发写（操作分别分布在不同segment上）这个默认值可以在初始化的时候设置为其他值，但是一旦初始化后，是不可以扩容的。



JDK8

- Java 1.8 采用**CAS** 加 **synchronized** 实现



JDK1.7 与 JDK1.8 的不同

数据结构不同 • Java 1.7 中采用 cegment 结构，默认只有 16 个，并发度低 • Java 1.8 中采用链表加红黑树结构，提高了并发性

Hash碰撞 • Java 1.7 中采用拉链法，时间复杂度是 $O(n)$ • Java 1.8 中采用链表加红黑树算法，时间复杂度是 $O(\log n)$

保证并发安全不同 Java 1.7 采用的 cegement 分段锁 继承自 ReentrantLock 可重入锁 Java 1.8 采用 CAS 加 synchronized 实现

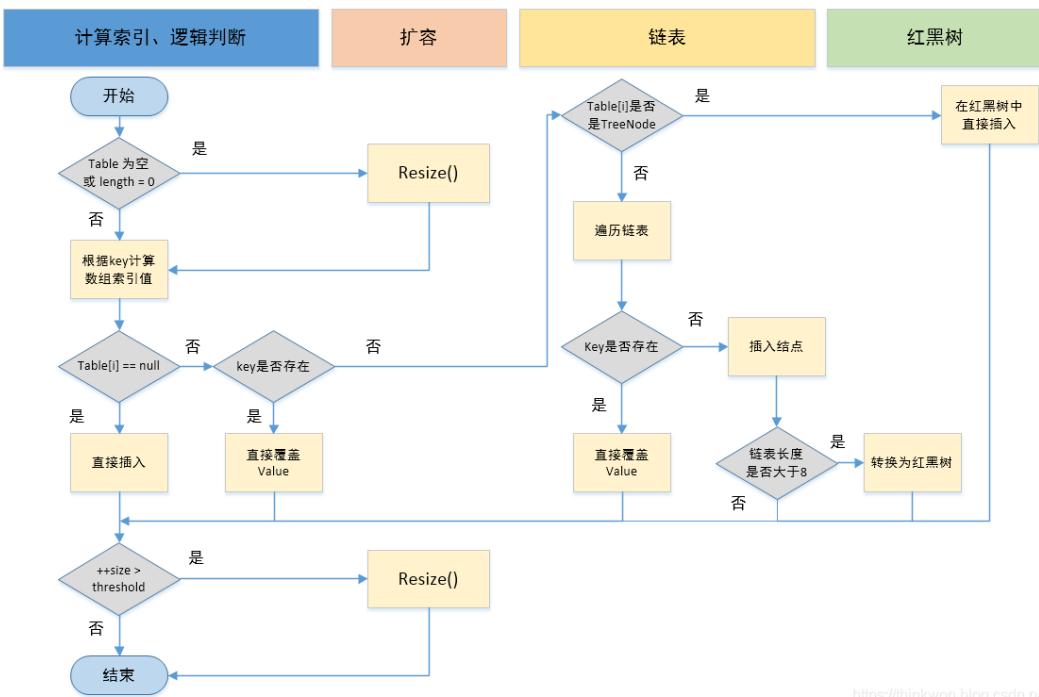
不同	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数： <code>inflateTable()</code>	直接集成到了扩容函数 <code>resize()</code> 中
hash值计算方式	扰动处理 = 9 次扰动 = 4 次位运算 + 5 次异或运算	扰动处理 = 2 次扰动 = 1 次位运算 + 1 次异或运算
存放数据的规则	无冲突时，存放数组；冲突时，存放链表	无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8：树化并存放红黑树
插入数据方式	头插法（先讲原位置的数据移到后 1 位，再插入数据到该位置）	尾插法（直接插入到链表尾部/红黑树）
扩容后存储位置的计算方式	全部按照原来方法进行计算（即 <code>hashCode -> 扰动函数 -> (h & length - 1)</code> ）	按照扩容后的规律计算（即 扩容后的位置 = 原位置 or 原位置 + 旧容量）

💡 HashMap 的 put 方法的具体流程

首先计算 key 的 hash 值。

- `key.hashCode()` 与 `key.hashCode() >>> 16` 进行异或操作。
- 高 16 位补 0，一个数和 0 异或不变，所以 hash 函数大概作用就是
- 高 16bit 不变，低 16bit 和高 16bit 做一个异或，**目的是减少碰撞**

putVal 方法执行流程图



<https://thinkwon.blog.csdn.net>

```

public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}

static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

//实现Map.put和相关方法
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 步骤①: tab为空则创建
    // table未初始化或者长度为0, 进行扩容
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // 步骤②: 计算index, 并对null做处理
    // (n - 1) & hash 确定元素存放在哪个桶中, 桶为空, 新生成结点放入桶中(此时, 这个
    // 结点是放在数组中)
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // 桶中已经存在元素
    else {
        Node<K,V> e; K k;
        // 步骤③: 节点key存在, 直接覆盖value
        // 比较桶中第一个元素(数组中的结点)的hash值相等, key相等
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            // 将第一个元素赋值给e, 用e来记录
            e = p;
        // 步骤④: 判断该链为红黑树
        // hash值不相等, 即key不相等; 为红黑树结点
        // 如果当前元素类型为TreeNode, 表示为红黑树, putTreeVal返回待存放的node,
        // e可能为null
        else if (p instanceof TreeNode)
            // 放入树中
    }
}

```

```

        e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // 步骤⑤: 该链为链表
        // 为链表结点
        else {
            // 在链表最末插入结点
            for (int binCount = 0; ; ++binCount) {
                // 到达链表的尾部

                // 判断该链表尾部指针是不是空的
                if ((e = p.next) == null) {
                    // 在尾部插入新结点
                    p.next = newNode(hash, key, value, null);
                    // 判断链表的长度是否达到转化红黑树的临界值, 临界值为8
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        // 链表结构转树形结构
                        treeifyBin(tab, hash);
                    // 跳出循环
                    break;
                }
                // 判断链表中结点的key值与插入的元素的key值是否相等
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    // 相等, 跳出循环
                    break;
                // 用于遍历桶中的链表, 与前面的e = p.next组合, 可以遍历链表
                p = e;
            }
        }
        // 判断当前的key已经存在的情况下, 再来一个相同的hash值、key值时, 返回新来的
        // value这个值
        if (e != null) {
            // 记录e的value
            v oldValue = e.value;
            // onlyIfAbsent为false或者旧值为null
            if (!onlyIfAbsent || oldValue == null)
                // 用新值替换旧值
                e.value = value;
            // 访问后回调
            afterNodeAccess(e);
            // 返回旧值
            return oldValue;
        }
    }
    // 结构性修改
    ++modCount;
    // 步骤⑥: 超过最大容量就扩容
    // 实际大小大于阈值则扩容
    if (++size > threshold)
        resize();
    // 插入后回调
    afterNodeInsertion(evict);
    return null;
}

```

hashMap 的扩容操作是什么？

1. 在 JDK1.8 中， resize() 方法在 hashMap 中的键值对大于阈值时初始化，就调用 resize 方法进行扩容
2. 每次扩容，都是 2 倍
3. 扩容后的 Node 对象的位置要么在原位置，要么移动到偏移量两倍的位置。

在 putVal() 中，我们看到在这个函数里面使用到了 2 次 resize() 方法， resize() 方法表示的在进行第一次初始化时会对其进行扩容，或者当该数组的实际大小大于其临界值时（**第一次为 12**），这个时候在扩容的同时也会伴随的桶上面的元素进行重新分发，这也是 JDK1.8 版本的一个优化的地方，在 1.7 中，扩容之后需要重新去计算其 Hash 值，根据 Hash 值对其进行分发，但在 1.8 版本中，则是根据在同一个桶的位置中进行判断 (e.hash & oldCap) 是否为 0，重新进行 hash 分配后，该元素的位置要么停留在原始位置，要么移动到原始位置 + 增加的数组大小这个位置上

```
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table; // oldTab 指向 hash 桶数组
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) { // 如果 oldCap 不为空的话，就是 hash 桶数组不为空
        if (oldCap >= MAXIMUM_CAPACITY) { // 如果大于最大容量了，就赋值为整数最大的阀值
            threshold = Integer.MAX_VALUE;
            return oldTab; // 返回
        } // 如果当前 hash 桶数组的长度在扩容后仍然小于最大容量 并且 oldCap 大于默认值 16
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                  oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold 双倍扩容阀值 threshold
    }
    // 旧的容量为 0，但 threshold 大于零，代表有参构造有 cap 传入， threshold 已经被初始化成最小 2 的 n 次幂
    // 直接将该值赋给新的容量
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    // 无参构造创建的 map，给出默认容量和 threshold 16, 16 * 0.75
    else { // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    // 新的 threshold = 新的 cap * 0.75
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
? (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    // 计算出新的数组长度后赋给当前成员变量 table
    @SuppressWarnings({"rawtypes", "unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; // 新建 hash 桶数组
    table = newTab; // 将新数组的值复制给旧的 hash 桶数组
    // 如果原先的数组没有初始化，那么 resize 的初始化工作到此结束，否则进入扩容元素重排逻辑，使其均匀的分散
    if (oldTab != null) {
        // 遍历新数组的所有桶下标
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
```

```

    if ((e = oldTab[j]) != null) {
        // 旧数组的桶下标赋给临时变量e，并且解除旧数组中的引用，否则就数组
无法被GC回收
        oldTab[j] = null;
        // 如果e.next==null，代表桶中就一个元素，不存在链表或者红黑树
        if (e.next == null)
            // 用同样的hash映射算法把该元素加入新的数组
            newTab[e.hash & (newCap - 1)] = e;
        // 如果e是TreeNode并且e.next!=null，那么处理树中元素的重排
        else if (e instanceof TreeNode)
            ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
        // e是链表的头并且e.next!=null，那么处理链表中元素重排
        else { // preserve order
            // loHead,loTail 代表扩容后不用变换下标，见注1
            Node<K,V> loHead = null, loTail = null;
            // hiHead,hiTail 代表扩容后变换下标，见注1
            Node<K,V> hiHead = null, hiTail = null;
            Node<K,V> next;
            // 遍历链表
            do {
                next = e.next;
                if ((e.hash & oldCap) == 0) {
                    if (loTail == null)
                        // 初始化head指向链表当前元素e，e不一定是链表的第一个元素，初始化后loHead
                        // 代表下标保持不变的链表的头元素
                        loHead = e;
                    else
                        // loTail.next指向当前e
                        loTail.next = e;
                    // loTail指向当前的元素e
                    // 初始化后，loTail和loHead指向相同的内存，所以当
loTail.next指向下一个元素时，
                        // 底层数组中的元素的next引用也相应发生变化，造成
loHead.next.next.....
                        // 跟随loTail同步，使得loHead可以链接到所有属于该链
表的元素。
                        loTail = e;
                }
                else {
                    if (hiTail == null)
                        // 初始化head指向链表当前元素e，初始化后hiHead
代表下标更改的链表头元素
                        hiHead = e;
                    else
                        hiTail.next = e;
                    hiTail = e;
                }
            } while ((e = next) != null);
            // 遍历结束，将tail指向null，并把链表头放入新数组的相应下标，形成新的映射。
            if (loTail != null) {
                loTail.next = null;
                newTab[j] = loHead;
            }
            if (hiTail != null) {
                hiTail.next = null;
                newTab[j + oldCap] = hiHead;
            }
        }
    }
}

```

```

        }
    }
}
}
return newTab;
}

```

HashMap 是怎么解决哈希冲突的？

1. 使用链地址法（使用散列表）来链接拥有相同hash值的数据。
2. 使用2次扰动函数来降低hash冲突的频率，使得数据分布更均匀
3. 引入红黑树进一步降低遍历的时间复杂度，使得遍历更快。

为什么 hashmap 中的String、Integer 这样的包装类型适合作为key？

String、Integer 等包装类的特性可以保证hash值的不可更改性和准确计算性，能够有效的减少发生碰撞的几率。

- 都是final类型，即不可变性，保证key的不可更改性，不会获取hash值不同的情况
- 内部已经重写了equals、hashcode等方法，遵循了hashmap内部的规范，不容易出现hash值计算错误的情况

HashMap为什么不直接使用hashCode()处理后的哈希值直接作为table的下标？

HashMap的容量范围是在16（初始化默认值）~ 2^{30} 通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，从而导致经过hashcode计算出的哈希值可能不在数组大小范围内，进而无法匹配存储位置。

那怎么解决呢？

- HashMap自己实现了自己的hash方法，通过两次扰动使得自己的高位哈希值与低位哈希值进行异或运算，降低哈希碰撞概率也使得数据分布更均匀。
- 在保证数组长度为2的幂次方的时候，使得hash（）运算后的值与(&)运算数组长度-1；来获取数组下标的方式进行存储，

这样一来比取余操作更有效率，

二来也是因为只有当数组长度为2的幂次方时， $h \& (length-1)$ 才等价于 $h \% length$

三解决了哈希值与数组大小不匹配的问题

HashMap 和 Hashtable 有什么区别？

- hashMap去掉了HashTable 的contains方法，但是加上了containsValue（）和containsKey（）方法。
- hashTable同步的，而HashMap是非同步的，效率上比hashTable要高。
- hashMap允许空键值，而hashTable不允许。
- 初始容量和每次扩容大小不同

	HashMap	HashTable
初始容量	16(未指定容量)，指定容量将其扩充为2的幂次方	11
扩容	$2n$	$2n+1$

如何决定使用 HashMap 还是 TreeMap?

- 对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。
- 然而，假如你需要对一个**有序的key集合进行遍历**，TreeMap是更好的选择。

基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

hashCode () 与equals 的相关规定

1. 如果两个对象相等，则hashcode一定也是相同的
2. 如果两个对象相等，两个equals方法返回true
3. 如果两个对象的hashcode相等，那么他们的值也不定相等
4. equals方法被覆盖，hashcode方法也被覆盖

== 和equals 区别

如果是基本数据类型 == 和equals 含义相同都是判断两个元素是否相等。

如果是引用类型

- == 表示的两个对象的地址是否相等
- equals 表示的是两个对象的内容是否相等

♡ Set

说一下 HashSet 的实现原理?

- HashSet底层由HashMap实现
- HashSet的值存放于HashMap的key上
- HashMap的value统一为PRESENT

HashSet 如何检查重复? HashSet是如何保证数据不可重复的?

结合hash值和equals 方法进行比较。

HashSet 的key 是唯一的，调用了Hashmap的 put () 方法。

♡ ArrayList

- 非线程安全的列表，底层用数组实现，支持随机访问。
- 插入删除慢 :删除元素的时候需要做一次元素复制操作，如果复制的元素过多，那么性能就不是很好。
- 每次扩容只会增加50%

如何实现数组和 List 之间的转换?

- List转换成为数组：调用ArrayList的toArray方法。
- 数组转换成为List：调用Arrays的asList方法。

为什么 ArrayList 的 elementData 加上 transient 修饰?

ArrayList 中的数组定义如下：

```
private transient Object[] elementData;
```

再看一下 ArrayList 的定义：

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

可以看到 ArrayList 实现了 Serializable 接口，这意味着 ArrayList 支持序列化。transient 的作用是说不希望 elementData 数组被序列化，重写了 writeObject 实现：

```
private void writeObject(java.io.ObjectOutputStream s) throws
java.io.IOException{
    /* Write out element count, and any hidden stuff*
       int expectedModCount = modCount;
       s.defaultWriteObject();
    /* Write out array length*
       s.writeInt(elementData.length);
    /* Write out all elements in the proper order.*/
       for (int i=0; i<size; i++)
           s.writeObject(elementData[i]);
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

每次序列化时，先调用 defaultWriteObject() 方法序列化 ArrayList 中的非 transient 元素，然后遍历 elementData，只序列化已存入的元素，这样既加快了序列化的速度，又减小了序列化之后的文件大小。

♡ LinkedList

- 非线程安全的列表，底层使用双向列表实现。不支持随机访问。
- 插入删除快，

ArrayList 和 LinkedList 的区别是什么？

数据结构 ArrayList 是动态数组实现的，LinkedList 是基于双向链表实现的

随机访问效率 ArrayList 比 LinkedList 的随机访问效率要高。

增加删除效率 在非首尾的增加和删除效率，LinkedList 的效率要比 ArrayList 高，因为 ArrayList 的增删操作要影响数组内的其他数据的下标

内存空间占用 LinkedList 要比 ArrayList 更占内存。因为 LinkedList 除了存储数据，还需要存储两个引用，一个指向下一个元素，一个指向前一个元素。

线程安全 他们都是线程不安全的。

总的来说：删除和更新使用 LinkedList 更快一些，查找操作使用 ArrayList 更快。

ArrayList 和 Vector 的区别是什么？

他们都实现了 List 接口，都是有序集合。

- Vector 是同步的，而 ArrayList 不是。然而，**如果你寻求在迭代的时候对列表进行改变，你应该使用 CopyOnWriteArrayList。**
- ArrayList 比 Vector 快，它因为有同步，不会过载。
- ArrayList 更加通用，因为我们可以使用 Collections 工具类轻易地获取同步列表和只读列表。

Array 和 ArrayList 有何区别？

- Array 可以容纳 **基本类型和对象**，而 ArrayList 只能容纳对象。
- Array 是指定大小的，而 ArrayList 大小是固定的。

- Array没有提供ArrayList那么多功能，比如addAll、removeAll和iterator等。

Queue

BlockingQueue是什么？

当添加一个元素，如果队列满则阻塞住，当移除一个元素，如果队列为空，则进行阻塞。

主要用来实现生产者和消费者

在 Queue 中 poll() 和 remove() 有什么区别？

poll() 和 remove() 都是从队列中取出一个元素，

1. poll() 在获取元素失败的时候会返回空

poll()

获取并移除此队列的头，如果此队列为空，则返回 null。

返回：

队列的头，如果此队列为空，则返回 null

2. 但是 remove() 失败的时候会抛出异常。

remove()

获取并移除此队列的头。此方法与 [poll](#) 唯一的不同在于：此队列为空时将抛出一个异常。

返回：

队列的头

抛出：

[NoSuchElementException](#) – 如果此队列为空

16.哪些集合类是线程安全的？

- vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。
- statck：堆栈类，先进后出。
- hashtable：就比hashmap多了个线程安全。
- enumeration：枚举，相当于迭代器。

迭代器 Iterator 是什么？

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为“轻量级”对象，因为创建它的代价小。

Iterator 怎么使用？有什么特点？

Java中的Iterator功能比较简单，并且只能单向移动：

(1) 使用方法iterator()要求容器返回一个Iterator。第一次调用Iterator的next()方法时，它返回序列的第一个元素。注意：iterator()方法是java.lang.Iterable接口被Collection继承。

(2) 使用next()获得序列中的下一个元素。

(3) 使用hasNext()检查序列中是否还有元素。

(4) 使用remove()将迭代器新返回的元素删除。

Iterator是Java迭代器最简单的实现，为List设计的ListIterator具有更多的功能，它可以从两个方向遍历List，也可以从List中插入和删除元素。

list遍历方式有哪些？

for 循环遍历

迭代器遍历，iterator。 iterator是面向对象的一个设计模式，目的是屏蔽不同集合的特点，统一遍历集合的接口。

foreach 循环遍历

Iterator 和 ListIterator 有什么区别？

- Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。
- Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。
- ListIterator实现了Iterator接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

☆java基础☆

1. 面向对象的四大特性

封装

- 隐藏对象的属性及实现细节，仅对外提供公共的接口，将变化隔离，便于使用，提高安全性和复用性

继承

- 提供的代码的复用性，继承是多态的前提

多态

- 父类或接口定义的引用变量可以指向父类或实现类的具体对象。提高了程序的扩展性

抽象

- 将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面，抽象只关注对象有哪些属性和行为，并不关注行为的细节是什么

instanceof关键字的作用

instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例，用法为：

```
boolean result = obj instanceof Class
```

2. 什么是拆装箱？

```
public class Test4 {
    public static void main(String[] args) {
        Integer a=127;
        Integer b=127;
        // -128 -- +127 之间的都是相等的
        System.out.println(a==b); //true
        Integer c=129;
        Integer d=129;
        System.out.println(c==d); //false
    }
}
```

3. 重载重写

overload (重载)

- 发生在同一个类中
- 方法名必须相同，参数列表必须不同
- 与访问修饰符、返回值类型无关
- 构造器可以被重载

重载的作用：实现方法名的复用

override (重写)

- 发生在子类中
- 方法名、参数列表、返回值类型必须相同
- 访问修饰符不能低于父类的

4. final、finally、finalize 有什么区别？

final

- final可以修饰类、变量、方法
- 修饰类表示该类不能被继承、修饰方法表示该方法不能被重写、修饰变量表示该变量是一个常量不能被重新赋值。如果变量是对象则对象的引用不能变，其值可以改变

finally

- finally一般作用在try-catch代码块中，在处理异常的时候，通常我们将一定要执行的代码放在finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

finalize

- finalize是一个方法，属于Object类的一个方法，而Object类是所有类的父类，该方法一般由垃圾回收器来调用，当我们调用System的gc()方法的时候，由垃圾回收器调用finalize()，回收垃圾。
- 一旦垃圾回收器准备好释放对象占用的空间，将首先调用其finalize() 方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。

6. 什么是 java 序列化？什么情况下需要序列化？

简单说就是为了保存在内存中的各种对象的状态（也就是实例变量，不是方法），并且可以把保存的对象状态再读出来。虽然你可以用你自己的各种各样的方法来保存object states，但是Java给你提供一种应该比你自己好的保存对象状态的机制，那就是序列化。

什么情况下需要序列化：？

- a) 当你想把的内存中的对象状态保存到一个文件中或者数据库中时候;
- b) 当你想用套接字在网络上传送对象的时候;
- c) 当你想通过RMI传输对象的时候;

7. Error与Exception有什么区别?

Error表示 系统级 的错误和程序不必处理的异常，比如说内存溢出

Exception 表示需要捕捉或者需要程序进行处理的异常。

8. heap和stack有什么区别。

栈是一种线形集合，其添加和删除元素的操作应在同一段完成。

栈按照后进先出的方式进行处理。

10. String s = new String("xyz");创建了几个String Object?

- 两个对象，一个是“xyz”，
- 一个是指向“xyz”的引用对象s。

11.String 类的常用方法都有那些?

- indexOf(): 返回指定字符的索引。
- charAt(): 返回指定索引处的字符。
- replace(): 字符串替换。
- trim(): 去除字符串两端空白。
- split(): 分割字符串，返回一个分割后的字符串数组。
- getBytes(): 返回字符串的 byte 类型数组。
- length(): 返回字符串长度。
- toLowerCase(): 将字符串转成小写字母。
- toUpperCase(): 将字符串转成大写字符。
- substring(): 截取字符串。
- equals(): 字符串比较。

12.java 中的 Math.round(-1.5) 等于多少?

等于 -1，因为在数轴上取值时，中间值 (0.5) 向右取整，所以正 0.5 是往上取整，负 0.5 是直接舍弃。

13.short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?

s1是short型，s1+1是int型,不能显式转化为short型。

short s1 = 1; s1 += 1正确。

内部类 (Inner class)

每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响

使用内部类最大的优点就在于它能够非常好的解决多重继承的问题,使用内部类还能够为我们带来如下特性:

- (1)、内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。
- (2)、在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。
- (3)、创建内部类对象的时刻并不依赖于外围类对象的创建。
- (4)、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。
- (5)、内部类提供了更好的封装，除了该外围类，其他类都不能访问。

- 用static修饰一个内部类时（嵌套类），这个类相当于是一个外部定义的类，所以static的内部类中可声明static成员
- static内部类不能使用外部类的非static成员变量

14. 接口和抽象类有什么区别？

- 实现：抽象类的子类使用 extends 来继承；接口必须使用 implements 来实现接口。
- 构造函数：抽象类可以有构造函数；接口不能有。
- main 方法：抽象类可以有 main 方法，并且我们能运行它；接口不能有 main 方法。
- 实现数量：类可以实现很多个接口；但是只能继承一个抽象类。
- 访问修饰符：接口中的方法默认使用 public 修饰；抽象类中的方法可以是任意访问修饰符。

15. java 中 IO 流分为几种？

按功能来分：输入流（input）、输出流（output）。

按类型来分：字节流和字符流。

字节流和字符流的区别是：字节流按 8 位传输以字节为单位输入输出数据，字符流按 16 位传输以字符为单位输入输出数据。

16. BIO、NIO、AIO 有什么区别？

- BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。
- NIO：New IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。
- AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO，异步 IO 的操作基于事件和回调机制。

17. Files的常用方法都有哪些？

- Files.exists()：检测文件路径是否存在。
- Files.createFile()：创建文件。
- Files.createDirectory()：创建文件夹。
- Files.delete()：删除一个文件或目录。
- Files.copy()：复制文件。
- Files.move()：移动文件。
- Files.size()：查看文件个数。
- Files.read()：读取文件。
- Files.write()：写入文件。

18.try {}里有一个return语句，那么紧跟在这个try后的finally {}里的code会不会被执行，什么时候被执行，在return前还是后？

会执行，在return前执行。

19.编程题：用最有效率的方法算出2乘以8等于几？

2 << 3

20.char型变量中能不能存贮一个中文汉字？为什么？

答：是能够定义成为一个中文的，因为java中以unicode编码，一个char占16个字节，所以放一个中文是没问题的

21.值传递

当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？

- 是值传递。Java 编程语言只由值传递参数。
- 当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的内容可以在被调用的方法中改变，但对象的引用是永远不会改变的。

22.运行时异常与一般异常有何异同？

异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误。

java编译器要求方法必须声明抛出可能发生的非运行时异常，但是并不要求必须声明抛出未被捕获的运行时异常。

23.String、StringBuffer和StringBuilder的区别

- String 和 StringBuffer、StringBuilder 的区别在于 String 声明的是不可变的对象，每次操作都会生成新的 String 对象，然后将指针指向新的 String 对象，
- 而 StringBuffer、StringBuilder 可以在原有对象的基础上进行操作，所以在经常改变字符串内容的情况下最好不要使用 String。
- StringBuffer 和 StringBuilder 最大的区别在于，StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的，但 StringBuilder 的性能却高于 StringBuffer，所以在单线程环境下推荐使用 StringBuilder，多线程环境下推荐使用 StringBuffer。

24.” == “和 equals 的区别是什么？

== 解读

对于基本类型和引用类型 == 的作用效果是不同的，如下所示：

- 基本类型：比较的是值是否相同；
- 引用类型：比较的是引用是否相同；

```
String x = "string";
String y = "string";
String z = new String("string");
System.out.println(x==y); // true
System.out.println(x==z); // false
System.out.println(x.equals(y)); // true
System.out.println(x.equals(z)); // true
```

代码解读：因为 x 和 y 指向的是同一个引用，所以 == 也是 true，而 new String()方法则重写开辟了内存空间，所以 == 结果为 false，而 equals 比较的一直是值，所以结果都为 true。

总结：

== 对于基本类型来说是值比较，对于引用类型来说是比较的是引用；

而 equals 默认情况下是引用比较，只是很多类重写了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等。

25.Equals、hashcode

1. 两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？

- 两个对象相同 hashcode 一定相同
- hashcode相同，两个对象不一定相同

26.什么是反射？

反射主要是指程序可以访问、检测和修改它本身状态或行为的一种能力

Java反射：

在Java运行时环境中，对于任意一个类，能否知道这个类有哪些属性和方法？对于任意一个对象，能否调用它的任意一个方法

Java反射机制主要提供了以下功能：

- 在运行时判断任意一个对象所属的类。
- 在运行时构造任意一个类的对象。
- 在运行时判断任意一个类所具有的成员变量和方法。
- 在运行时调用任意一个对象的方法。

27.动态代理是什么？有哪些应用？

动态代理：

当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新类。这个代理类并不是定义好的，是动态生成的。具有解耦意义，灵活，扩展性强。

动态代理的应用：

- Spring的AOP
- 加事务
- 加权限
- 加日志

28.怎么实现动态代理？

首先必须定义一个接口，还要有一个InvocationHandler(将实现接口的类的对象传递给它)处理类。再有一个工具类Proxy(习惯性将其称为代理类，因为调用他的newInstance()可以产生代理对象,其实他只是一个产生代理对象的工具类)。利用到InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

29.如何实现对象克隆？

有两种方式：

- 1) 实现Cloneable接口并重写Object类中的clone()方法；
- 2) 实现Serializable接口，通过对对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下：

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class MyUtil {

    private MyUtil() {
        throw new AssertionError();
    }

    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T obj) throws Exception
    {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bout);
        oos.writeObject(obj);

        ByteArrayInputStream bin = new
        ByteArrayInputStream(bout.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bin);
        return (T) ois.readObject();

        // 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义
        // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放
    }
}
```

下面是测试代码：

```
import java.io.Serializable;

class Person implements Serializable {
    private static final long serialVersionUID = -9102017020286042305L;

    private String name;      // 姓名
    private int age;          // 年龄
    private Car car;          // 座驾
```

```

public Person(String name, int age, Car car) {
    this.name = name;
    this.age = age;
    this.car = car;
}

//getter setter...
//toString()
}

```

```

class Car implements Serializable {
    private static final long serialVersionUID = -5713945027627603702L;

    private String brand;          // 品牌
    private int maxSpeed;         // 最高时速

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }

    //getter setter...
    //toString()
}

```

```

class CloneTest {

    public static void main(String[] args) {
        try {
            Person p1 = new Person("郭靖", 33, new Car("Benz", 300));
            Person p2 = MyUtil.clone(p1);    // 深度克隆
            p2.getCar().setBrand("BYD");
            // 修改克隆的Person对象p2关联的汽车对象的品牌属性
            // 原来的Person对象p1关联的汽车不会受到任何影响
            // 因为在克隆Person对象时其关联的汽车对象也被克隆了
            System.out.println(p1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

注意：基于**序列化和反序列化**实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，**这项检查是编译器完成的，不是在运行时抛出异常**，

这种方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

30.深拷贝和浅拷贝区别是什么？

- 浅拷贝只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅拷贝（例：assign()）
- 深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse()和JSON.stringify()，但是此方法无法复制函数类型）

31.throw 和 throws 的区别?

- throws是用来声明一个方法可能抛出的所有异常信息， throws是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。
- 而throw则是指抛出的一个具体的异常类型。

32.try-catch-finally 中哪个部分可以省略?

答：catch 可以省略

原因：

更为严格的说法其实是：**try只适合处理运行时异常**， try+catch适合处理运行时异常+普通异常。

也就是说，如果你只用try去处理普通异常却不加以catch处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用catch显示声明以便进一步处理。

而运行时异常在编译时没有如此规定，所以catch可以省略，你加上catch编译器也觉得无可厚非。

理论上，编译器看任何代码都不顺眼，都觉得可能有潜在的问题，所以你即使对所有代码加上try，代码在运行期时也只不过是在正常运行的基础上加一层皮。

但是你一旦对一段代码加上try，就等于显示地承诺编译器，对这段代码可能抛出的异常进行捕获而非向上抛出处理。

如果是普通异常，编译器要求必须用catch捕获以便进一步处理；如果运行时异常，捕获然后丢弃并且+finally扫尾处理，或者加上catch捕获以便进一步处理。

至于加上finally，则是在不管有没有捕获异常，都要进行的“扫尾”处理。

33. try-catch-finally 中，如果 catch 中 return 了， finally 还会执行吗？

答：会执行，在 return 前执行。

34.常见的异常类有哪些？

- NullPointerException：当应用程序试图访问空对象时，则抛出该异常。
- SQLException：提供关于数据库访问错误或其他错误信息的异常。
- IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。
- NumberFormatException：当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。
- FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。
- IOException：当发生某种I/O异常时，抛出此异常。此类是失败或中断的I/O操作生成的异常的通用类。
- ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。
- ArrayStoreException：试图将错误类型的对象存储到一个对象数组时抛出的异常。
- IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。
- ArithmeticException：当出现异常的运算条件时，抛出此异常。例如，一个整数“除以零”时，抛出此类的一个实例。
- NegativeArraySizeException：如果应用程序试图创建大小为负的数组，则抛出该异常。
- NoSuchElementException：无法找到某一特定方法时，抛出该异常。
- SecurityException：由安全管理器抛出的异常，指示存在安全侵犯。
- UnsupportedOperationException：当不支持请求的操作时，抛出该异常。

- `Runtime``Exception``RuntimeException`: 是那些可能在Java虚拟机正常运行期间抛出的异常的超类。

35java中的两种异常类型是什么？他们有什么区别？

- 受检查异常 (checked) 必须使用 `throws` 语句在方法或者构造函数上声明
- 不受检查异常 (unchecked) 不需要在方法和构造函数上声明

☆JavaWeb☆

1. jsp 和 servlet 有什么区别？

1. jsp经编译后就变成了Servlet. (JSP的本质就是Servlet, JVM只能识别java的类，不能识别JSP的代码，Web容器将JSP的代码编译成JVM能够识别的java类)
2. jsp更擅长表现于页面显示，servlet更擅长于逻辑控制。
3. Servlet中没有内置对象，**Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse 对象以及HttpServletRequest对象得到。**
4. Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

2.jsp 有哪些内置对象？作用分别是什么？

JSP有9个内置对象：

- **request:** 封装客户端的请求，其中包含来自GET或POST请求的参数；
 - 它包含了有关浏览器请求的信息，并且提供了几个用于获取**cookie, header, 和session**数据的有用的方法。
 - 是 `HttpServletRequest` 类的实例
 - 作用域： `request` (用户请求期)
- **response:** 封装服务器对客户端的响应；
 - 并提供了几个用于设置**送回** 浏览器的响应的方法（如**cookies,头信息等**）
 - 是 `HttpServletResponse` 类的实例
 - 作用域 `page` (页面执行期)
- **pageContext:** 通过该对象可以获取其他对象；
 - 它是用于方便存取各种范围的名字空间、servlet相关的对象的API，并且包装了**通用的servlet相关功能的方法**。
- **session:**
 - 指的是客户端与服务器的一次会话，从客户端连到服务器的一个 `WebApplication` 开始，直到客户端与服务器断开连接为止；
 - Session可以存储用户的状态信息
 - 它是 `HttpSession` 的实例
 - 作用域 `session` (会话期)
- **application:**
 - 实现了用户数据的共享，可存放全局变量。
 - 它开始于服务器启动，直到服务器关闭，在此期间此对象一直存在。这样在用户的前后连接或不同用户之间的连接中，可以对此对象的同一属性进行操作；
 - 在任何地方对此对象属性的操作，都将影响到其他用户对此的访问。
 - 服务器的启动和关闭决定了 `application` 对象的生命；
 - 作用域 `application`。

- **out:**
 - 输出服务器响应的输出流对象；
 - out 对象是 JspWriter 类的实例，是向客户端输出内容常用的对象
 - 作用域 page
- **config:** Web应用的配置对象；作用域 page
- **page:**
 - JSP页面本身（相当于Java程序中的this）；
 - page 对象代表了正在运行的由JSP 文件产生的类对象。
 - 作用域 page
- **exception:**
 - 封装页面抛出异常的对象。
 - 当一个页面在运行过程中发生了例外，就产生这个对象。
 - 如果一个JSP 页面要应用此对象，就必须把isErrorPage 设为true，否则无法编译。
 - 作用域 page

3.说一下 jsp 的 4 种作用域？

JSP中的四种作用域包括**page、request、session和application**，具体来说：

- **page** 代表与一个页面相关的对象和属性。
- **request** 代表与Web客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个Web组件；需要在页面显示的临时数据可以置于此作用域。
- **session** 代表与某个**用户与服务器建立的一次会话相关的对象和属性**。跟某个用户相关的数据应该放在用户自己的session中。
- **application** 代表与整个Web应用程序相关的对象和属性，它实质上是**跨越整个Web应用程序，包括多个页面、请求和会话的一个全局作用域**。

4.session 和 cookie 有什么区别？

Session

- 由于**HTTP协议是无状态的协议**，所以服务端需要记录用户的状态时，就需要用某种机制来识别具体的用户，这个机制就是**Session**
- 典型的场景比如**购物车**，当你点击下单按钮时，由于HTTP协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的Session，**用于标识这个用户**，并且跟踪用户，这样才知道购物车里面有几本书。
- 这个Session是**保存在服务端的，有一个唯一标识**。
- 在**服务端保存Session**的方法很多，**内存、数据库、文件都有**。集群的时候也要考虑Session的转移，在大型的网站，一般会有专门的Session服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的，使用一些缓存服务比如**Memcached之类的来放 Session**。

Cookie

- **思考一下服务端如何识别特定的客户？**
- 这个时候Cookie就登场了。每次HTTP请求的时候，客户端都会发送相应的Cookie信息到服务端。实际上大多数的应用都是用 Cookie 来实现Session跟踪的，第一次创建Session的时候，服务端会在HTTP协议中告诉客户端，**需要在 Cookie 里面记录一个Session ID，以后每次请求把这个会话ID发送到服务器**，我就知道你是谁了。
- **有人问，如果客户端的浏览器禁用了 Cookie 怎么办？**
 - 一般这种情况下，会使用一种叫做**URL重写的技术**来进行会话跟踪，即每次HTTP交互，URL后面都会被附加一个诸如 sid=xxxxx 这样的参数，服务端据此来识别用户。

- Cookie其实还可以用在一些方便用户的场景下，设想你某次登陆过一个网站，下次登录的时候不想再次输入账号了，怎么办？
 - 这个信息可以写到Cookie里面，访问网站的时候，网站页面的脚本可以读取这个信息，就自动帮你把用户名给填了，能够方便一下用户。这也是Cookie名称的由来，给用户的一点甜头。

总结

- Session**是在服务端保存的一个数据结构**，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中；
- Cookie是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现Session的一种方式。

图解

Cookie与Session的区别	
Cookie	Session
存储在客户端	存储在服务器端
两种类型 ※有生命周期 ※无生命周期	两种实现方式 ※依赖于cookie ※url重写
父路径不能访问子路径的cookie	同一个session 的窗口共享一个 session
典型应用： ※3个月不用再登陆 ※购物车	典型应用： ※用户登陆 ※购物车也可以用session 实现。
不可靠	可靠

1. cookie 数据存储在客户端上，session 数据存储在服务器上
2. cookie 不安全，别人可以分析放在本地中的cookie 并进行cookie欺骗，需要考虑安全的应用请使用session
3. session 会在一定时间中保存到服务器，当访问增多，会消耗服务器的性能，如果考虑减轻服务器性能方面考虑 cookie
4. 单个cookie 在客户端的限制是3 K，也即是说站点在客户端保存的cookie 不能超过3 K
5. session 适合分布式登录，同一个用户在访问一个网站期间，所有的session在任何一个地方都可以访问到。而cookie 设置了路径参数，同一个网站下的不同路径互相是访问不了的，cookie 只能是子路径访问父路径的cookie

购物车session 实现思路

session 是一种保存上下文的信息的机制，它是针对每一个用户的，变量的值保存在服务器端，通过 SessionID 来区分不同的客户，session 是以cookie 或 URL 重写为基础的。

默认使用 cookie 来实现，系统会为其创建一个名为**JSESSIONID 的输出 cookie**，我们叫做 **session cookie**，以区分 persistent cookie (我们通常所说的cookie) 。

注意 **session cookie是存储于浏览器内存中的**，并不是写到硬盘上的，这也就是我们刚才看到的 JSESSIONID，我们通常情况下是看不到 JSESSIONID 的，但是当我们把浏览器的 cookie 禁止后，web 服务器会采用 URL 重写的方式传递 Sessionid 我们就可以在地址栏看到

sessionid= KWJHUG6JJM65HS2K6之类的字符串。

明白了原理，我们就可以很容易的分辨出 persistent cookie和 session cookie的区别了，网上那些关于两者安全性的讨论也就一目了然了， session cookie针对某一次会话而言，会话结束 session cookie也就随着消失了，而**persistent cookie只是存在于客户端硬盘上的一段文本（通常是加密的）**，而且可能会遭到 cookie欺骗以及针对 cookie的跨站脚本攻击，自然不如 session cookie安全了

通常 **session cookie是不能跨窗口使用的**，当你新开了一个浏览器窗口进入相同页面时，系统会赋予你一个新的 sessionid这样我们信息共享的目的就达不到了。

此时我们可以先把 sessionid保存在 **persistent cookie中**，然后在新窗persistent cookie的结合我们就实现了跨窗口的 session tracking会话跟踪)

5.说一下 session 的工作原理？

其实session是一个存在服务器上的类似于一个散列表格的文件。

里面存有我们需要的信息，在我们需要用的时候可以从里面取出来。**类似于一个大号的map**

里面的键存储的是**用户的sessionid**，用户向服务器发送请求的时候会带上这个sessionid。这时就可以从中取出对应的值了。

6.如果客户端禁止 cookie 能实现 session 还能用吗？

Cookie与 Session，一般认为是两个独立的东西，

- Session采用的是在服务器端保持状态的方案，
- 而Cookie采用的是在客户端保持状态的方案。

为什么禁用Cookie就不能得到Session呢？

- 因为Session是用Session ID来确定当前对话所对应的服务器Session，而Session ID是通过Cookie来传递的，禁用Cookie相当于失去了Session ID，也就得不到Session了。

假定用户关闭Cookie的情况下使用Session，其实现途径有以下几种：

1. 设置php.ini配置文件中的“session.use_trans_sid = 1”，或者编译时打开打开了“--enable-trans-sid”选项，让PHP自动跨页传递Session ID。
2. 手动通过URL传值、隐藏表单传递Session ID。
3. 用文件、数据库等形式保存Session ID，在跨页过程中手动调用。

7.spring mvc 和 struts 的区别是什么？

1. 拦截机制的不同

Struts2

- 是类级别的拦截，**每次请求就会创建一个Action**，和Spring整合时Struts2的ActionBean注入作用域是**原型模式prototype**，然后通过setter，getter吧request数据注入到属性。
- Struts2中，一个Action对应一个request，response上下文，在接收参数时，可以通过属性接收，这说明属性参数是让多个方法共享的。**Struts2中Action的一个方法可以对应一个url，而其类属性却被所有方法共享**，这也就无法用注解或其他方式标识其所属方法了，只能设计为多例。

SpringMVC

- **是方法级别的拦截，一个方法对应一个Request上下文**，所以方法直接基本上是独立的，独享request，response数据。
- 而每个方法同时又何一个url对应，参数的传递是直接注入到方法中的，是方法所独有的。处理结果通过ModelMap返回给框架。在Spring整合时，**SpringMVC的Controller Bean默认**

单例模式Singleton, 所以默认对所有的请求, 只会创建一个Controller, 有应为没有共享的属性, 所以是线程安全的, 如果要改变默认的作用域, 需要添加@Scope注解修改。

Struts2有自己的拦截Interceptor机制, **SpringMVC这是用的是独立的Aop方式**, 这样导致Struts2的配置文件量还是比SpringMVC大。

2.底层框架的不同

- Struts2采用**Filter** (StrutsPrepareAndExecuteFilter) 实现, Filter在容器启动之后即初始化; 服务停止以后销毁, 晚于Servlet。
- SpringMVC (DispatcherServlet) 则采用**Servlet实现**。Servlet是在调用时初始化, 先于Filter调用, 服务停止后销毁。

3.性能方面

- Struts2是类级别的拦截, 每次请求对应实例一个新的Action, 需要加载所有的属性值注入,
- SpringMVC实现了零配置, 由于SpringMVC基于方法的拦截, 有加载一次单例模式bean注入。所以, SpringMVC开发效率和性能高于Struts2。

4.配置方面

spring MVC和Spring是无缝的。从这个项目的管理和安全上也比Struts2高。

8、doGet()和doPost()区别?

doGet()

- 安全性差。因为是直接将数据显示在地址栏中, 浏览器有缓冲, 可记录用户信息。
- 只能提交256个字符 (1024字节)
- 数据传输载体是URL (提交方式能form也能任意URL链接)
- **get方式有四种:**
 - 1) 直接在URL地址栏中输入URL。
 - 2) 网页中的超链接。
 - 3) form中method为get。
 - 4) form中method为空时, 默认是get提交。

doPost()

- 安全性高。因为post方式提交数据时是采用的HTTP post机制, 是将表单中的字段与值放置在HTTP HEADER内一起传送到ACTION所指的URL中, 用户是看不见的。
- Post适合发送大量的数据。
- post只知道有一种: form中method属性为post。

通常我们使用的都是**doPost方法**, 你只要在servlet中让这两个方法互相调用就行了, 例如在doGet方法中这样写

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    doPost(request, response);
}
```

8.1 forward 和 redirect 的区别?

Forward和Redirect代表了两种请求转发方式：直接转发和间接转发。

直接转发方式 (Forward)，客户端和浏览器只发出一次请求，Servlet、HTML、JSP或其它信息资源，由第二个信息资源响应该请求，在请求对象request中，保存的对象对于每个信息资源是共享的。

间接转发方式 (Redirect) 实际是**两次HTTP请求**，服务器端在响应第一次请求的时候，让浏览器再向另外一个URL发出请求，从而达到转发的目的。

举个通俗的例子：

直接转发就相当于：“A找B借钱， B说没有， B去找C借， 借到借不到都会把消息传递给A”；

间接转发就相当于：“A找B借钱， B说没有， 让A去找C借”。

```
"forward:user.do?name=method4"  
"redirect:http://www.uu456.com"
```

9、servlet的生命周期

1. Servlet 通过调用 init () 方法进行初始化。

init 方法被设计成只调用一次。它在第一次创建 Servlet 时被调用，在后续每次用户请求时不再调用。

Servlet 是单例的 多个用户同时访问存在线程安全问题

解决办法：尽量不要在Servlet 中定义成员变量。即使定义了成员变量，也不要对其修改值

2. Servlet 调用 service() 方法来处理客户端的请求。

每次访问Servlet时，Service方法都会被调用一次。

3. Servlet 通过调用 destroy() 方法终止（结束）。

destroy() 方法只会被调用一次，在 Servlet 生命周期结束时被调用。destroy() 方法可以让您的 Servlet 关闭数据库连接、停止后台线程、把 Cookie 列表或点击计数器写入到磁盘，并执行其他类似的清理活动。

10、如何现实servlet的单线程模式

```
<%@ page isThreadSafe="false"%>
```

11. forward 和redirect的区别?

forward(转发)：

- 是服务器请求资源，服务器直接访问目标地址的URL，把那个URI的相应内容读取过来，然后把这些内容在发给浏览器。对于浏览器来说不知道内容从哪来的，因为跳转过程是在服务器实现的。所以它的地址还是原来的地址

redirect(重定向)

- 是服务端根据逻辑发送一个状态码，告诉浏览器重新去请求那个地址，所以地址栏显示的是新的URL

转发是服务器行为，重定向是客户端行为

区别：

1. 数据共享来说

- forward: 转发页面和转发到的页面可以共享request里面的数据
- redirect 不能共享数据

2. 从地址栏来说 forward 不变 redirect 变

3. 从运用地方来说

forward: 一般用于用户登录的时候，根据角色转发到相应的模块

redirect: 一般用于用户注销登录时返回主页面或跳转到其它的网站等

4. 从效率来说

forward 高于 direct

5. 什么情况下调用doGet()和doPost()？

Jsp页面中的form标签里的method属性为get时调用doGet(), 为post时调用doPost()。

6.JSP和Servlet有哪些相同点和不同点，他们之间的联系是什么？

答：JSP是Servlet技术的扩展，本质上是Servlet的简易方式，更强调应用的外表表达。JSP编译后是“类servlet”。

Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。

JSP侧重于视图，Servlet主要用于控制逻辑。

7. 请解释Filter和Listener的理解及作用？、

8.jdbc流程

1. 加载JDBC 驱动程序

```
Class.forName("com.mysql.jdbc.Driver");
```

2. 提供JDBC 连接的URL

2. 获取数据库连接

方法: static Connection getConnection (String url, String user, String password)

参数

url: 指定连接的路径

语法: jdbc:mysql://i地址(域名):端口号/数据库名称

例子: jdbc:mysql://localhost:3306/db3

细节: 如果连接的是本机mysql服务器，并且mysql服务默认端口是3306，则url可以简写为: jdbc:mysql:///数据库名称

user: 用户名

password: 密码

3. 创建数据库的连接

```
//获取 connection对象  
Connection conn = DriverManager.getConnection(  
    url: "jdbc:mysql://localhost:3306/ db3",  
    user: "root",password: "root");
```

4. 创建一个Statement

要执行SQL语句，必须获得 `java.sql.Statement` 实例，
`Statement` 实例分为以下3种类型：
1、执行静态SQL语句。通常通过 `Statement` 实例实现。
2、执行动态SQL语句。通常通过 `PreparedStatement` 实例实现。
3、执行数据库存储过程。通常通过 `CallableStatement` 实例实现。

```
//获取执行SQL的对象  
Statement stmt = conn.createStatement();  
String sql = "select * from tb_user";  
PreparedStatement pstmt = conn.createStatement(sql);
```

5. 执行sql语句

`Statement` 接口提供了三种执行SQL语句的方法：
`executeQuery`、`executeUpdate` 和 `execute`
1、`ResultSet executeQuery (String sqlstring)` 执行查询数据库的SQL语句，返回一个结果集（`ResultSet`）对象。
2、`int executeUpdate (String sqlstring)` 用于执行 `INSERT`、`UPDATE` 或 `DELETE` 语句以及 SQL DDL 语句，如：`CREATE TABLE` 和 `DROP TABLE` 等
3、`execute (sql String)`：用于执行返回多个结果集、多个更新计数或二者组合的语句。具体实现的代码：

6. 处理结果

```
ResultSet st= stmt.executeQuery(sql);
```

7. 关闭连接，与创建顺序相反

9.Ajax

Ajax并不算是一种新的技术，全称是 asynchronous javascript and xml 可以说是已有技术的组合，主要用来实现客户端与服务器端的异步通信效果，实现页面的局部刷新

1. 页面编码和被请求的资源编码如果不一致如何处理？

对于ajax请求传递的参数，如果是get请求方式，参数如果传递中文，在有些浏览器会乱码，不同的浏览器对参数编码的处理方式不同，所以对于get请求的参数需要使用 `encodeURIComponent` 函数对参数进行编码处理，后台开发语言都有相应的解码api。对于post请求不需要进行编码

2. Ajax 的过程

1. 创建 `Xml http request` 对象，也就是创建一个异步调用对象
2. 创建一个新的HTTP请求，并指定该HTTP请求的方法、URL及验证信息
3. 设置响应HTTP请求状态变化的函数
4. 发送HTTP请求
5. 获取异步调用返回的数据
6. 使用 JavaScript 和 DoM 实现局部刷新

3. 简述异步加载

1. 异步加载的方案： 动态插入 `script` 标签
2. 通过 `ajax` 去获取 `js` 代码，然后通过 `eval` 执行
3. `script` 标签上添加 `defer` 或者 `async` 属性
4. 创建并插入 `iframe`, 让它异步执行 `js`

10.JS选择器

原生 JS 选择器有 `getElementById`、`getElementsByName`、`getElementsByTagName` 和 `getElementsByClassName` 这四个

1. `getElementById`(通过 ID 获取元素)

用法:`document.getElementById("Id");``Id` 为要获取的元素的 `id` 属性值。

2. `getElementsByName`(通过 name 属性获取元素)

用法:`document.getElementsByName("Name");``Name` 为要获取元素的 `name` 属性值,这个方法一般适用于提交表单数据,当元素为 `form`、`img`、`iframe`、`applet`、`embed`、`object` 的时候设置 `name` 属性时,会自动在 `Document` 对象中创建以该 `name` 属性值命名的属性。所以可以通过 `document.domName` 引用相应的 dom 对象

3. `getElementsByTagName`(通过元素名称获取元素)

用法:`document.getElementsByTagName(TagName);``TagName` 为要获取元素的标签名称,当 `TagName` 为*的时候表示获取所有的元素,`document` 也可以换成 `DOM` 元素,但是这样就只能获取到该 `DOM` 元素后面的子集元素。

4. `getElementsByClassName`(通过 CSS 类来获取元素)

用法:`document.getElementsByClassName(ClassName);``ClassName` 为要获取元素的 `CSS` 类名称,如果要同时获取多个的话,在每个 `CSS` 类后面用空格隔开。如 `document.getElementsByClassName("class2 class1")` 就会获取到 `class1` 和 `class2` 样式的元素,`document` 也可以换成 `DOM` 元素,这样也是只能获取到该 `DOM` 元素后面的子集元素。

11.拦截器和过滤器有什么区别

1. 拦截器是基于java反射机制，过滤器是基于函数回调.
2. 拦截器不依赖于 Servlet 容器，过滤器依赖 Servlet 容器
3. 拦截器只能对 action 请求起作用，过滤器几乎可以对所有的请求起作用
4. 拦截器可以访问 action 上下文、值栈里的对象，过滤器不能
5. 在 action 的声明周期中，拦截器可以多次调用，过滤器只能在初始化的时候被调用一次

拦截器：在面向切面编程的情况下，就是在你的一个 service 或者一个方法前调用一个方法，或者在方法后调用一个方法。例如 动态代理就是拦截器的简单实现，在你调用方法前打印字符串（或者做其他业务逻辑），也可以在你调用方法后打印字符串，甚至在你抛出异常的时候做业务逻辑的操作。

EJB

EJB与JAVA BEAN的区别?

答:EJB与JAVA BEAN是SUN的不同组件规范, EJB是在容器中运行的, 分步式的, 而JAVA BEAN主要是一种可利用的组件, 主要在客户端UI表现上。

1、EJB容器提供的服务

主要提供声明周期管理、代码产生、持续性管理、安全、事务管理、锁和并发管理等服务。

2、EJB的角色和三个对象

EJB角色主要包括Bean开发者 应用组装者 部署者 系统管理员 EJB容器提供者 EJB服务器提供者

三个对象是Remote (Local) 接口、Home (LocalHome) 接口, Bean类

2、EJB的几种类型

会话 (Session) Bean , 实体 (Entity) Bean 消息驱动的 (Message Driven) Bean

会话Bean又可分为有状态 (Stateful) 和无状态 (Stateless) 两种

实体Bean可分为Bean管理的持续性 (BMP) 和容器管理的持续性 (CMP) 两种

3、bean 实例的生命周期

对于Stateless Session Bean、Entity Bean、Message Driven Bean一般存在缓冲池管理, 而对于Entity Bean和Statefull Session Bean存在Cache管理, 通常包含创建实例, 设置上下文、创建EJB Object (create) 、业务方法调用、remove等过程, 对于存在缓冲池管理的Bean, 在create之后实例并不从内存清除, 而是采用缓冲池调度机制不断重用实例, 而对于存在Cache管理的Bean则通过激活和去激活机制保持Bean的状态并限制内存中实例数量。

4、激活机制

以Statefull Session Bean 为例：其Cache大小决定了内存中可以同时存在的Bean实例的数量，根据MRU或NRU算法，实例在激活和去激活状态之间迁移，激活机制是当客户端调用某个EJB实例业务方法时，如果对应EJB Object发现自己没有绑定对应的Bean实例则从其去激活Bean存储中（通过序列化机制存储实例）回复（激活）此实例。状态变迁前会调用对应的ejbActive和ejbPassivate方法。

5、remote接口和home接口主要作用

remote接口定义了业务方法，用于EJB客户端调用业务方法

home接口是EJB工厂用于创建和移除查找EJB实例

6、客服端调用EJB对象的几个基本步骤

一、设置JNDI服务工厂以及JNDI服务地址系统属性

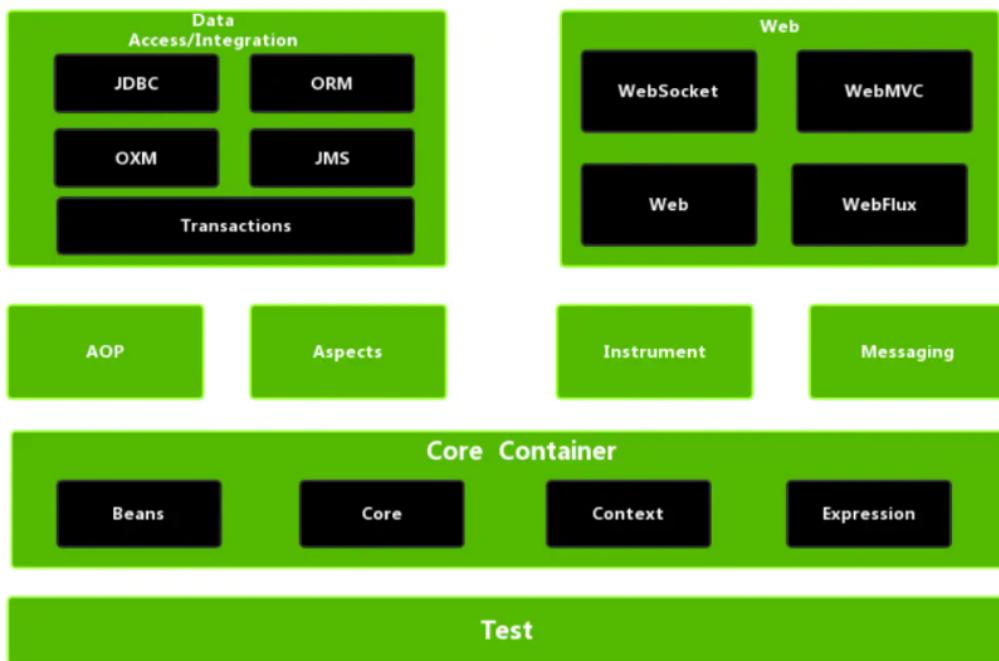
二、查找Home接口

三、从Home接口调用Create方法创建Remote接口

四、通过Remote接口调用其业务方法

☆Spring☆

spring 由哪几部分组成？



spring5组成架构

spring 由核心容器（Core Container）、数据访问/集成部分（Data Access/Integration）、AOP、Aspects、工具（instrument）、Web、报文发送（Messaging）、Test等模块。

（一）Core Container

1) Spring-core IOC 和 DI 的基本实现。

2) spring-beans BeanFactory 工厂使用IOC 对应用程序配置，BeanFactory 实例化并不会自动实例化 Bean，只有当 bean 使用时 BeanFactory 才会对其进行实例化和依赖关系的装配

3) spring-context Spring的上下文，即IOC容器。它扩展了 BeanFactory，添加了bean 的声明周期、框架事件体系等

4) spring-expression Spring表达式语言，是统计表达式el 的扩展模块。

(二) AOP

spring-aop 面向切面编程，spring中的aop使用了动态代理：基于接口的jdk实现，基于类的Cglib继承实现。

spring-aspects 集成AspectJ，主要是为Spring Aop提供多种AOp 语法

spring-instrument 提供一些类级的工具支持和ClassLoader级的实现

(三) 数据访问集成

Spring-jdbc 用来简化JDBC 编程 主要实现类JDBCTemplate

spring-tx 事务支持，

spring-orm 对象关系映射，集成ORM 框架

spring-oxm 对象XML映射

spring-jms 消息服务

spring-messaging 集成一些基础的报文发送服务

(四) WEB

spring-web 最基础的web支持，主要建立在核心容器上，通过 servlet 和 listener初始化IOC 容器，也包括一些web相关服务

spring-webmvc 实现了Spring-MVC 的web 应用

spring-websocket 主要于web前端的全双工通信协议

spring-webflux 是一个全新的非阻塞式的 Reactive Web 框架，可以用来异步、非阻塞、事件驱动的服务。

(Test)

1) spring-test: spring测试，提供junit与mock测试功能

2) spring-context-support: spring额外支持包，比如邮件服务、视图解析等。

1.为什么要使用Spring?

- IOC 控制反转，实现了高内聚低耦合
- AOP 面向切面编程，将业务逻辑与系统的服务分离
- 支持事务管理，spring提供了统一的事务管理接口，无需手动编程
- 支持单元测试
- 支持集成其他框架（Mybatis、Hibernate）
- 异常处理 spring 提供统一异常处理机制

谈谈你对IOC的理解？

IOC (inverse of control)

- IOC 字面意思是控制反转，将本来程序中需要我们自己创建的对象交给容器来管理，当我们使用的时候，容器会使用工厂模式自动为我们创建，
- spring的IOC有三种注入方式：构造器注入、setter注入、接口注入。
- 对象与对象之间松耦合，有利于功能的复用

谈谈你对AOP的理解？

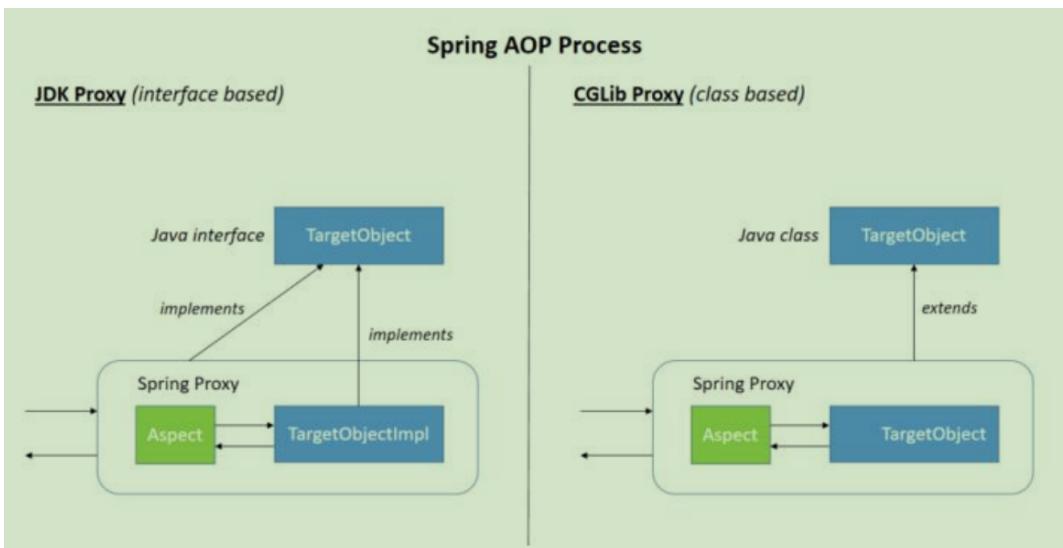
AOP的出现是对OOP的一种补充和完善，OOP允取开发者纵向的关系，导致了大量的重复代码和高耦合。AOP作为面向对象的一种补充，允许开发者横向的开发，将那些重复代码封装成一个模块，形成切面。降低系统耦合性，有利于未来的维护性。比如日志、权限认证、事务管理这些用到了AOP思想。

静态代理为Aspectj

- Aspectj 静态代理具有更好的性能，在编译阶段生成AOP 代理类，因为称为编译器增强。他会在编译器将切面织入到字节码中，运行时就是增强之后的AOP 对象

动态代理分为基于接口的 JDK动态代理和基于类的 Cglib动态代理。

- 因为Cglib是通过继承的方式做的动态代理，所以被代理类不能是final



请解释一下 Spring AOP 里面的几个名词

- (1) 切面 (Aspect)：被抽取的公共模块，可能会横切多个对象。在Spring AOP中，切面可以使用通用类（基于模式的风格）或者在普通类中以 @AspectJ 注解来实现。
- (2) 连接点 (Join point)：指方法，在Spring AOP中，一个连接点总是代表一个方法的执行。
- (3) 通知 (Advice)：在切面的某个特定的连接点 (Join point) 上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型，并维护一个以连接点为中心的拦截器链。
- (4) 切入点 (Pointcut)：切入点是指我们要对哪些Join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add、search。
- (5) 引入 (Introduction)：（也被称为内部类型声明 (inter-type declaration)）。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。
- (6) 目标对象 (Target Object)：被一个或者多个切面 (aspect) 所通知 (advise) 的对象。也有人把它叫做 被通知 (advised) 对象。既然Spring AOP是通过运行时代理实现的，这个对象永远是一个被代理 (proxied) 对象。
- (7) 织入 (Weaving)：指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

切入点 (pointcut) 和连接点 (join point) 匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。切入点使得定位通知 (advice) 可独立于OO层次。例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。

Spring通知有哪些类型？

https://blog.csdn.net/qq_32331073/article/details/80596084

- (1) 前置通知 (Before advice) : 在某连接点 (join point) 之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。
- (2) 返回后通知 (After returning advice) : 在某连接点 (join point) 正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。
- (3) 抛出异常后通知 (After throwing advice) : 在方法抛出异常退出时执行的通知。
- (4) 后通知 (After (finally) advice) : 当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
- (5) 环绕通知 (Around Advice) : 包围一个连接点 (join point) 的通知，如方法调用。这是最强大的一种通知类型。环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和Boss4，都只提供环绕通知。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice before advice target method 执行 around after advice after advice afterReturning

②有异常情况下的执行顺序：

around before advice before advice target method 执行 around after advice after advice afterThrowing:异常发生 java.lang.RuntimeException: 异常发生

3.Spring AOP 和 AspectJ AOP 有什么区别？

Spring AOP 属于运行时增强，而 AspectJ 是编译时增强。Spring AOP 基于代理(Proxying)，而 AspectJ 基于字节码操作(Bytecode Manipulation)。

Spring AOP 已经集成了 AspectJ，AspectJ 应该算是 Java 生态系统中最完整的 AOP 框架了。

AspectJ 相比于 Spring AOP 功能更加强大，但是 Spring AOP 相对来说更简单，如果我们的切面比较少，那么两者性能差异不大。

但是，当切面太多的话，最好选择 AspectJ，它比 Spring AOP 快很多。

4.spring 常用的注入方式有哪些？

Spring通过DI（依赖注入）实现IOC（控制反转），常用的注入方式主要有三种：

1. 构造方法注入
2. setter注入
3. 基于接口的注入

```
1.接口注入
public interface Injection{
    public void injectionName(String name);
}
为对象注入name属性
public class User implements Injection{
    private String name;
    public void injectName(String name){
        this.name = name;
    }
}
```

```
2.构造器注入
public class User{
    private String name;
    public User(String name){
        this.name = name;
    }
}
```

```
3.set注入
public class User{
    private String name;
    public void setname(String name){
        this.name = name;
    }
}
```

Spring 支持构造器注入和setter 方法注入：

构造器注入，通过 `<constructor-args>` 元素完成注入

setter 方法注入，通过 `<setters>` 元素完成注入（开发中常用）

Spring 中单例 bean 是线程安全的么？

Spring 框架并没有对单例 bean 进行任何多线程的封装处理。实际上大部分 Spring bean 并没有可变的状态（比如 service 和 Dao 类），所以某种程度上 Spring bean 是线程安全的。

如果你的 Bean 有多种状态的话（比如 View Model 对象），就需要自行保证线程安全，最浅显的解决办法是将单例模式改为原型模式。

Spring 如何处理多线程并发问题？

Spring 使用 ThreadLocal 保证多线程并发安全问题。

同步机制采用时间换空间，仅提供一份变量，不同线程在访问时需要获取锁，没有获取锁的线程排队等待。

ThreadLocal 采用空间换时间的方式，为每一个线程提供一个独立的变量副本，从而隔离了多个线程之间的访问冲突。从而没有必要对该变量进行同步。

6. Spring 支持几种 bean 的作用域？

当通过 Spring 容器创建一个 Bean 实例时，不仅可以完成 Bean 实例的实例化，还可以为 Bean 指定特定的作用域。Spring 支持如下 5 种作用域：

- singleton：单例模式，在整个 Spring IoC 容器中，使用 `singleton` 定义的 Bean 将只有一个实例
- prototype：原型模式，每次通过容器的 `getBean` 方法获取 `prototype` 定义的 Bean 时，都将产生一个新的 Bean 实例
- request：对于每次 HTTP 请求，使用 `request` 定义的 Bean 都将产生一个新实例，即每次 HTTP 请求将产生不同的 Bean 实例。只有在 Web 应用中使用 Spring 时，该作用域才有效
- session：对于每次 HTTP Session，使用 `session` 定义的 Bean 都将产生一个新实例。同样只有在 Web 应用中使用 Spring 时，该作用域才有效
- globalSession：每个全局的 HTTP Session，使用 `globalSession` 定义的 Bean 都将产生一个新实例。典型情况下，仅在使用 portlet context 的时候有效。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

其中比较常用的是 `singleton` 和 `prototype` 两种作用域。对于 `singleton` 作用域的 Bean，每次请求该 Bean 都将获得相同的实例。容器负责跟踪 Bean 实例的状态，负责维护 Bean 实例的生命周期行为；如果一个 Bean 被设置成 `prototype` 作用域，程序每次请求该 id 的 Bean，Spring 都会新建一个 Bean 实例，然后返回给程序。在这种情况下，Spring 容器仅仅使用 `new` 关键字创建 Bean 实例，一旦创建成功，容器不在跟踪实例，也不会维护 Bean 实例的状态。

如果不指定 Bean 的作用域，Spring 默认使用 `singleton` 作用域。

Java在创建Java实例时，需要进行内存申请；销毁实例时，需要完成垃圾回收，这些工作都会导致系统开销的增加。因此，prototype作用域Bean的创建、销毁代价比较大。而singleton作用域的Bean实例一旦创建成功，可以重复使用。因此，除非必要，否则尽量避免将Bean被设置成prototype作用域。

7.spring 自动装配 bean 有哪些方式？

(一) Spring 框架在XML 配置中有5种自动装配.

- **no** 默认的不进行装配，通过手工设置 ref 属性进行装配
- **byName** 通过 bean 的名称进行自动装配，如果一个bean的 property 与另一个bean 的 name 相同，就进行自动装配
- **byType** 通过参数的数据类型进行装配
- **constructor** 利用构造函数进行装配，并且构造函数的参数通过byType进行装配
- **autodetect** 自动探测，如果有构造方法，通过constructor的方式自动装配，否则使用 byType 方式自动装配

(二) 基于注解的方式

@Autowired

- 在使用此注解之前需要先在配置文件中进行配置，开启注解配置 <context:annotation-config />
- 在启动Spring IOC 时，容器自动装载了一个 AutowiredAnnotationBeanProcessor 后置处理器，当容器扫描到@.Autowired、@Resource或@Inject 时，就会在IOC 容器中自动查找需要的bean，并装配给对象属性

创建Bean的三种方式

1. 使用默认构造函数创建

在spring 配置文件中使用 bean 标签，配以id 和class 属性，且没有其他属性和标签时

```
<bean id="accountService" class =  
"com.ncst.service.impl.AccountServiceImpl"></bean>
```

2. 使用普通工厂中的方法创建对象（工厂方法模式）

```
<bean id="instanceFactory" class="com.itheima.factory.InstanceFactory">  
</bean>  
<bean id="accountService" factory-bean="instanceFactory"  
factory-method="getAccountService"></bean>
```

3. 使用工厂中的静态方法创建对象（简单工厂模式）

```
<bean id="accountService" class="com.ncst.factory.StaticFactory"  
factory-method="getAccountService"></bean>
```

注解

@Required 注解

这个注解表明 bean 的属性必须在配置的时候设置，通过一个bean 定义的显式配置或者自动装配。若@Required 注解的bean 属性未被设置，容器抛出BeanInitializationException

```
public class Employee {  
    private String name;  
    @Required  
    public void setName(String name){  
        this.name=name;  
    }  
    public string getName(){  
        return name;  
    }  
}
```

RequiredAnnotationBeanPostProcessor 是Spring 中的后置处理器用来验证被@Required 注解的bean 是否被正确设置了

@Autowired 和@Resource 区别？

1. @Autowired 是按照类型装配注入的， 默认情况下他要求依赖对象必须存在（required = true）
2. @Resource 默认是按照名称来装配注入的， 只有找不到与名称匹配的 bean 才会按照类型装配注入。

@Qualifier注解的作用

解决有两个相同类型的bean 使用@Autowired 时不能区分到底使用的是哪种类型， 使用了 @Qualifier 注解 告诉Spring 容器（按照 byName类型）到底要装配哪个bean

@Component 和 @Bean 的区别是什么？

1. 作用对象不同： @Component 注解作用于类， 而 @Bean 注解作用于方法。
2. @Component 通常是通过类路径扫描来自动检测以及自动装配到Spring容器中（我们可以使用@ComponentScan 注解定义要扫描的路径从中找出标识了需要装配的类自动装配到 Spring 的bean 容器中）。

@Bean 注解通常是在标有该注解的方法中定义产生这个 bean, @Bean 告诉了Spring这是某个类的示例， 当我需要用它的时候还给我。

3. @Bean 注解比 Component 注解的自定义性更强， 而且很多地方我们只能通过 @Bean 注解来注册bean。 比如当我们引用第三方库中的类需要装配到 Spring 容器时，则只能通过 @Bean 来实现。

8.Spring 事务种类

Spring 事务的本质其实就是数据库对事务的支持， 没有数据库的事务支持， Spring 是无法提供事务功能的。

（1）Spring 事务的种类

Spring 支持编程式事务管理和声明式事务管理两种方式：

编程式事务管理 transactionTemplate

- 我们需要在代码中调用beginTransaction()、 commit()、 rollback()等事务管理相关的方法

声明式事务管理

- 声明式事务管理建立在AOP 之上， 其本质是通过AOP 功能， 对方法前后进行拦截， 将事务处理的功能编织到拦截的方法中， 也即是目标方法开始之前加入一个事务，在执行目标方法后根据情况提交或者回滚事务。

- 声明式事务的优点是不需要在业务逻辑代码中掺杂事务管理的代码，只需要配置相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到逻辑中。
- 声明式管理优于编程式事务管理，这正符合Spring倡导的非侵入式的开发方式

● Spring 事务传播行为

当多个事务同时存在的时候，Spring 如何处理这些事务的行为

♡ propagation(传播)

PROPAGATION_REQUIRED: 如果当前存在事务，就加入该事务；如果当前没有事务，就创建一个新事务（这个是最常用的配置）

PROPAGATION_SUPPORTS: 支持当前事务。如果当前存在事务，就加入事务；不存在以非事务执行。♡ supports(支持的)

PROPAGATION_MANDATORY: 强制执行事务。如果当前存在事务，就加入事务；不存在就抛出异常。♡ mandatory (强制的)

PROPAGATIONQUIRES_NEW 创建新事务。无论是否存在事务，都新建事务

PROPAGATION_NOT_SUPPORTED 以非事务的方式执行操作，如果当前存在事务，则将当前事务挂起

PROPAGATION_NEVER 以非事务的方式执行，如果当前存在事务，则抛出异常

PROPAGATION_NESTED 如果存在事务，则按嵌套事务执行。如果当前没有事务，则按 REQUIRED属性执行 ♡ nested (嵌套的)

● Spring 中的隔离级别

ISOLATION_DEFAULT 使用数据库默认隔离级别

ISOLATION_READ_UNCOMMITTED 读未提交，允取另一个事务可以看到这个事务未提交的数据。《幻读、脏读、不可重复读》

ISOLATION_READ_COMMITTED 读已提交，保证一个事务修改的数据提交后，其他事物才可以读取，而且能看到该事务对已有记录的更新《幻读、不可重复读》

ISOLATION_REPEATABLE_READ: 可重复读，保证要给事务修改的数据提交后才能被另一个事务读取，但是不能看到该事务对已有记录的更新《幻读》

ISOLATION_SERIALIZABLE: 一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

- 脏读：一个事务读到另一个事务未提交的更新数据。
- 幻读：事务A 按照一定条件进行数据读取，期间事务B 插入了相同搜索条件的新数据，事务A再次按照原先条件进行读取时，发现了事务B 新插入的数据 称为幻读
- 不可重复读：比方说在同一个事务中先后执行两条一模一样的select语句，期间在此次事务中没有执行过任何DDL语句，但先后得到的结果不一致，这就是不可重复读。

@Transactional(rollbackFor = Exception.class)注解了解吗？

我们知道：Exception分为运行时异常 RuntimeException和非运行时异常。

事务管理对于企业应用来说是至关重要的，即使出现异常情况，它也可以保证数据的一致性。

当 @Transactional 注解作用于类上时，该类的所有 public 方法将都具有该类型的事务属性，同时，我们也可以在方法级别使用该标注来覆盖类级别的定义。

如果类或者方法加了这个注解，那么这个类里面的方法抛出异常，就会回滚，数据库里面的数据也会回滚。

在 @Transactional 注解中如果不配置 rollbackFor 属性，那么事物只会在遇到 RuntimeException 的时候才会回滚。加上 rollbackFor=Exception.class，可以让事物在遇到非运行时异常时也回滚。

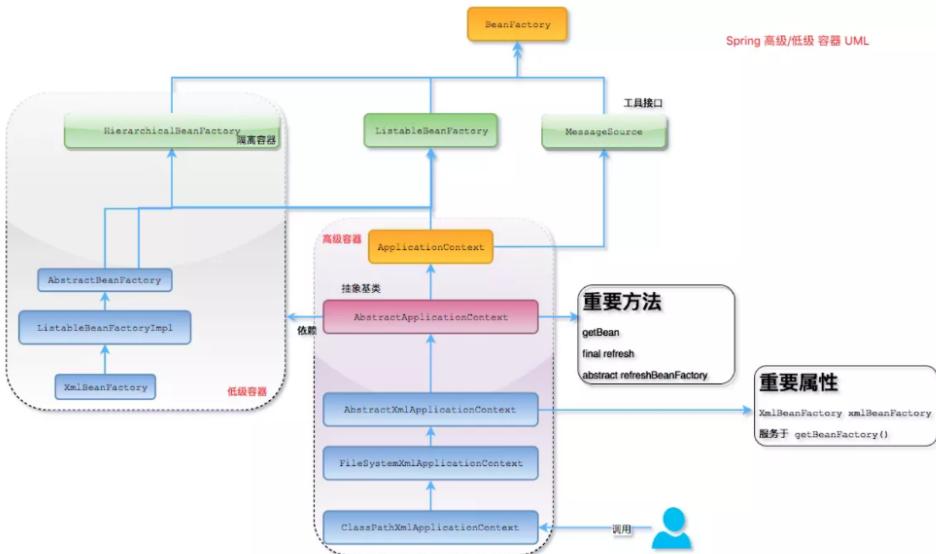
14. BeanFactory 和 ApplicationContext 的区别详解

(一) BeanFactory

- 是一个 Bean 工厂，使用简单工厂模式，是 Spring IoC 的顶级接口，可以理解为含有 Bean 集合的工厂类，作用是管理 Bean，包括实例化、定位、配置对象及建立这些对象的依赖。
- BeanFactory 实例化后并不会自动实例化 Bean，只有当 Bean 被使用时才实例化与装配依赖关系，属于延迟加载，适合多例模式
- BeanFactory 简单粗暴可以理解为一个 HashMap，key 是 beanName，value 是 bean 的实例。通常只提供 put（注册）和 get（获取）两个功能。称为低级容器

(二) ApplicationContext

- ApplicationContext 继承于 BeanFactory 接口，扩展了 BeanFactory 的功能，提供了支持国际化的文本消息，统一的资源文件读取方式，事件传播以及应用层的特别配置等。
- 容器会在初始化时对配置的 Bean 进行预实例化，Bean 的依赖注入在容器初始化时就已经完成，属于立即加载，适合单例模式，一般推荐使用。
- ApplicationContext 可以被称为高级容器



15. Spring 有几种配置方式

1. 基于 XML 的配置
2. 基于注解的配置
3. 基于 Java 的配置

请解释 Spring Bean 的生命周期

首先说一下 Servlet 的生命周期：实例化、初始化 init、接受请求 service、销毁 destroy

Spring Bean 的生命周期

(1) 实例化 Bean

对于BeanFactory 容器，当客户容器请求一个尚未初始化的bean 时或初始化bean 需要注入另一个尚未初始化的依赖时，容器就会调用 createBean 进行实例化。

对于ApplicationContext 容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

(2) 设置对象属性 (依赖注入)

实例化后对象被封装到 BeanWrapper对象中，紧接着，Spring根据BeanDefinition 中的信息以及通过 BeanWrapper 提供的设置属性的接口完成依赖注入

(3) 处理Aware 接口

接着Spring 会检测该对象是否实现了 XXXAware 接口，并将相关的XXXAware 实例注入给 Bean

1. 如果这个Bean 实现了 BeanNameAware 接口，将会调用 setBeanName(String beanId) 方法，(此处传递的就是Spring 配置文件中 Bean 的ID 值)
2. 如果这个Bean 实现了 BeanFactoryAware 接口，将会调用 setBeanFactory () 方法，传递的是Spring 工厂自身
3. 如果这个Bean 实现了 ApplicationContextAware 接口，会调用 setApplicationContext (ApplicationContext) 方法，传入Spring 上下文。

(4) BeanPostProcessor 前置处理

如果想对 Bean 进行一些自定义的处理，那么可以让Bean 实现此接口。

(5) InitializationBean 与 init-method

如果在配置文件中配置了init-method，则会自动调用其配置的初始化方法

(6) BeanPostProcessor 后置处理

如果这个Bean 实现了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization (Object obj, String s) 方法。由于这个方法是在初始化结束后调用的，所以可以被用于缓存或内存技术，

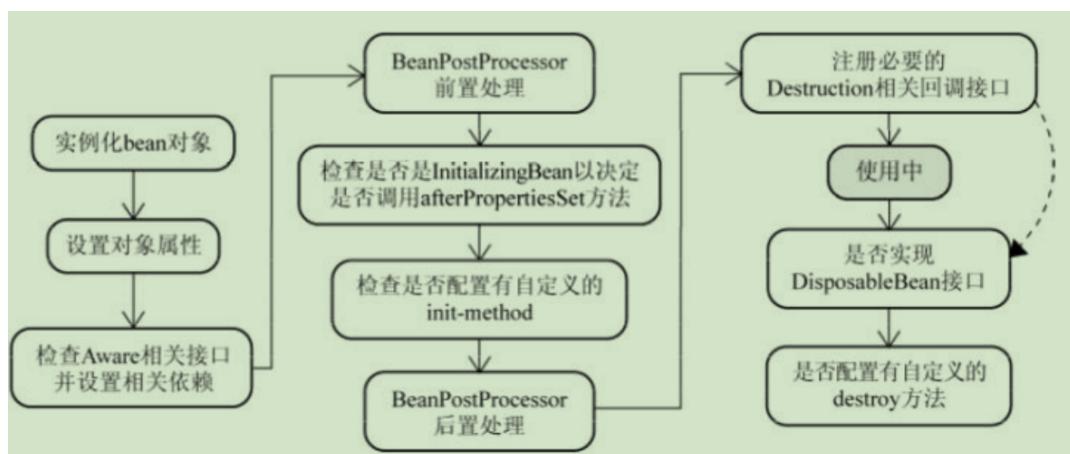
开始使用

(7) DisposableBean

- 当Bean 不需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean这个接口，会调用其实现的destory()方法。

(8) destory-method

- 如果这个Bean 的Spring 配置中配置了destory-method 属性，会自动调用其配置销毁方法



22. 构造方法注入和设值注入有什么区别？

22、构造方法注入和设值注入有什么区别？请注意以下明显的区别：

1. 在设值注入方法支持大部分的依赖注入，如果我们仅需要注入int、string和long型的变量，我们不要用设值的方法注入。对于基本类型，如果我们没有注入的话，可以为基本类型设置默认值。在构造方法注入不支持大部分的依赖注入，因为在调用构造方法中必须传入正确的构造参数，否则的话为报错。
2. 设值注入不会重写构造方法的值。如果我们对同一个变量同时使用了构造方法注入又使用了设置方法注入的话，那么构造方法将不能覆盖由设值方法注入的值。很明显，因为构造方法尽在对象被创建时调用。
3. 在使用设值注入时有可能还不能保证某种依赖是否已经被注入，也就是说这时对象的依赖关系有可能是不完整的。而在另一种情况下，构造器注入则不允许生成依赖关系不完整的对象。
4. 在设值注入时如果对象A和对象B互相依赖，在创建对象A时Spring会抛出 `sObjectCurrentlyInCreationException` 异常，因为在 B 对象被创建之前 A 对象是不能被创建的，反之亦然。所以 Spring 用设值注入的方法解决了循环依赖的问题，因对象的设值方法是在对象被创建之前被调用的。

Spring 框架中有哪些不同类型的事件

Spring 提供了以下5种标准的事件：

- (1) 上下文更新事件 (ContextRefreshedEvent)：在调用`ConfigurableApplicationContext`接口中的`refresh()`方法时被触发。
- (2) 上下文开始事件 (ContextStartedEvent)：当容器调用`ConfigurableApplicationContext`的`Start()`方法开始/重新开始容器时触发该事件。
- (3) 上下文停止事件 (ContextStoppedEvent)：当容器调用`ConfigurableApplicationContext`的`Stop()`方法停止容器时触发该事件。
- (4) 上下文关闭事件 (ContextClosedEvent)：当`ApplicationContext`被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。
- (5) 请求处理事件 (RequestHandledEvent)：在Web应用中，当一个http请求 (request) 结束触发该事件。

如果一个bean实现了`ApplicationListener`接口，当一个`ApplicationEvent` 被发布以后，bean会自动被通知。

24. FileSystemResource 和 ClassPathResource 有何区别

在 FileSystemResource 中需要给出 spring-config.xml 文件在你项目中的相对路径或者绝对路径。在 ClassPathResource 中 spring 会在 ClassPath 中自动搜寻配置文件，所以要把 ClassPathResource 文件放在 ClassPath 下。

如果将 spring-config.xml 保存在了 src 文件夹下的话，只需给出配置文件的名称即可，因为 src 文件夹是默认。

简而言之，ClassPathResource 在环境变量中读取配置文件，FileSystemResource 在配置文件中读取配置文件。

Spring 中用到了哪些设计模式？

工厂模式：BeanFactory 就是工厂模式的实现，Spring 通过 BeanFactory 创建 Bean

单例模式：Spring 默认 Bean 是单例的

适配器模式：

- Spring 的 AOP 中，通知（advice）通过适配器实现的，AdvisorAdapter
- HandlerAdapter Spring MVC 前端控制器发来的请求经过 HandlerAdapter 处理器适配后调用具体的 Controller

包装器模式

代理模式：Spring AOP 中 JDKDynamicAopProxy 和 Cglib2AopProxy 使用到了代理模式

观察者模式：Spring 中的监听器，ApplicationListener

策略模式：（定义一系列的算法，把它们一个个的封装起来，并且使它们可以相互替换。在实例化对象时用到）

模板方法模式：JdbcTemplate、RestTemplate、JmsTemplate、JpaTemplate

Spring 如何解决循环依赖？

Spring 通过内部维护三个 map

☆ Spring MVC ☆

1、什么是 SpringMVC？

SpringMVC 是 spring 的一个模块，基于 MVC 的一个框架，无需中间整合层来整合。

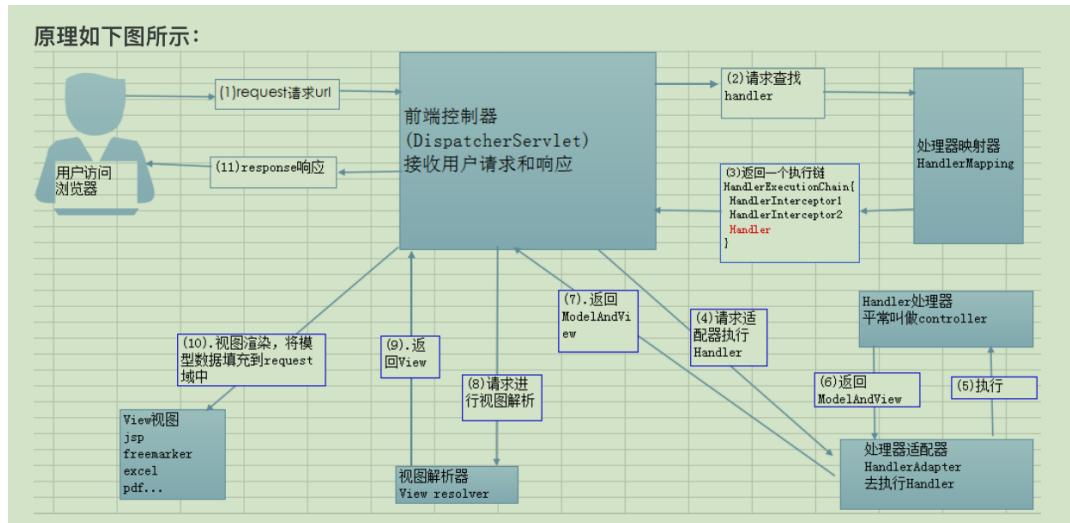
2. 优点

2、Spring MVC 的优点：

1. 它是基于组件技术的。全部的应用对象，无论控制器和视图，还是业务对象之类的都是 Java 组件，并且和 Spring 提供的其他基础结构紧密集成。
2. 不依赖于 Servlet API（目标虽是如此，但是在实现的时候确实是依赖于 Servlet 的）
3. 可以任意使用各种视图技术，而不仅仅局限于 JSP
4. 支持各种请求资源的映射策略
5. 它应是易于扩展的

3. Spring MVC 工作流程

Spring MVC运行流程图：



Spring运行流程描述：

1. 客户端发送请求，直接请求到DispatcherServlet
2. DispatcherServlet 根据请求信息调用 HandlerMapping，解析请求对应的handler
3. 解析对应的 handler (也就是我们平常说的Controller 控制器)后，开始由 HandlerAdapter 适配器处理
4. HandlerAdapter 根据对应的handler 来调用真正的处理器请求，并处理相应的业务逻辑
5. 处理完业务逻辑后，返回一个 ModelAndView 对象 Model 是数据上的对象，view是逻辑上的 view
6. viewResolver 根据view 查找实际的view
7. DispatcherServlet 会把 model 传给 View (渲染)
8. 把 view 传给浏览器 (请求者)

5、SpringMvc 的控制器是不是单例模式,如果是,有什么问题,怎么解决?

是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

4. Spring MVC 和struts2的区别

①springmvc的入口是一个servlet即前端控制器，而struts2入口是一个filter过滤器。

②springmvc是基于方法开发，传递参数是通过方法形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

③Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request对象内容进行解析成方法形参，将响应数据和页面封装成ModelAndView对象，最后又将模型数据通过request对象传输到页面。Jsp视图解析器默认使用jstl。

5.基础传参

12、我想在拦截的方法里面得到从前台传入的参数,怎么得到?

答:直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样

13、如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象?

直接在方法中声明这个对象,SpringMvc 就自动会把属性赋值到这个对象里面。

6.重定向 和转发

15、SpringMVC 怎么样设定重定向和转发的?

在返回值前面加"forward:"就可以让结果转发,譬如"forward:user.do?name=method4" 在返回值前面加"redirect:"就可以让返回值重定向,譬如 "redirect:<http://www.baidu.com>"

7.SpringMVC 用什么对象从后台像前台传递数据

16、SpringMvc 用什么对象从后台向前台传递数据的?

答:通过 ModelMap 对象,可以在这个对象里面用 put 方法,把对象加到里面,前台就可以通过 el 表达式拿到。

8.怎么把ModelMap 里面的数据放入Session里面

可以在类上面加上@SessionAttributes 注解,里面包含的字符串就是要放入 session 里面的 key

9.@ResponseBody注解与 Ajax

19、SpringMvc 怎么和 AJAX 相互调用的?

通过 Jackson 框架就可以把 Java 里面的对象直接转化成 Js 可以识别的 Json 对象。具体步骤如下:

1.加入 Jackson.jar

2.在配置文件中配置 json 的映射

3.在接受 Ajax 方法里面可以直接返回 Object,List 等,但方法前面要加上 @ResponseBody 注解

11.spring mvc 有哪些组件?

Spring MVC的核心组件:

1. DispatcherServlet: 中央控制器, 把请求给转发到具体的控制类
2. Controller: 具体处理请求的控制器
3. HandlerMapping: 映射处理器, 负责映射中央处理器转发给controller时的映射策略
4. ModelAndView: 服务层返回的数据和视图层的封装类
5. ViewResolver: 视图解析器, 解析具体的视图
6. Interceptors : 拦截器, 负责拦截我们定义的请求然后做处理工作

12. @RequestMapping 的作用是什么？

RequestMapping是一个用来处理请求地址映射的注解，可用于类或方法上。用于类上，表示类中的所有响应请求的方法都是以该地址作为父路径。

RequestMapping注解有六个属性，下面我们把她分成三类进行说明。

1. value, method:

value：指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

method：指定请求的method类型， GET、POST、PUT、DELETE等；

2. consumes, produces

consumes：指定处理请求的提交内容类型（Content-Type），例如application/json, text/html；

produces：指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；

3. params, headers

params：指定request中必须包含某些参数值是，才让该方法处理。

headers：指定request中必须包含某些指定的header值，才能让该方法处理请求。

13. 如何解决Web页面乱码问题？

解决 post 请求乱码问题

```
<filter>
    <filter-name>encodingFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.CharacterEncodingFilter
    </filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>

    <init-param>
        <param-name>forceEncoding</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

解决get 方法的乱码问题

修改 tomcat 配置文件添加编码与工程编码一致，如下：

```
<Connector URIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>
```

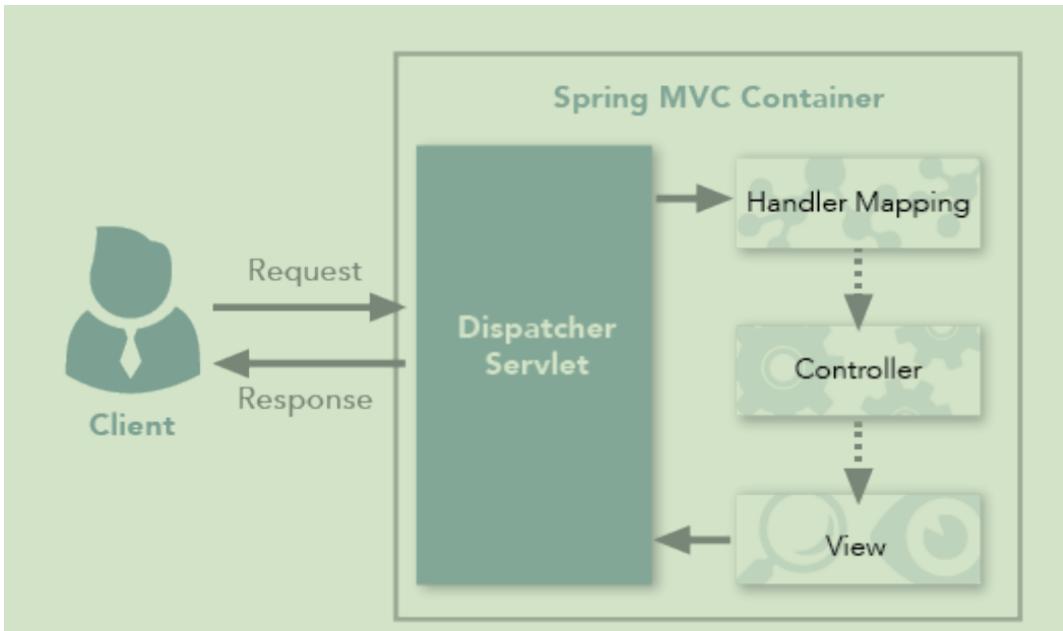
另外一种方法对参数进行重新编码：

```
String userName = new String(request.getParameter("userName").getBytes("ISO8859-1"), "utf-8")
```

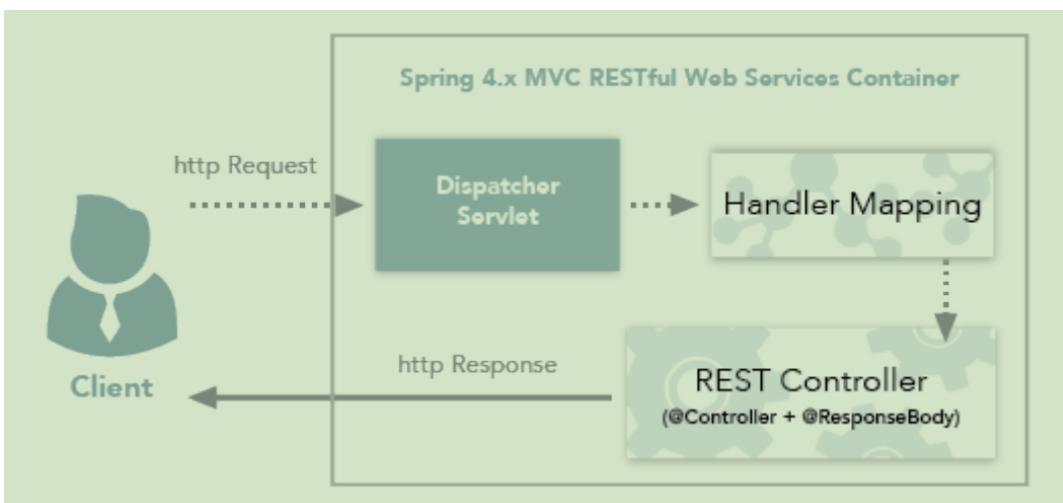
ISO8859-1 是 tomcat 默认编码，需要将 tomcat 编码后的内容按 utf-8 编码

14.@RestController和 Controller

Controller 返回一个页面单独使用 @Controller 不加 @ResponseBody 的话一般使用在要返回一个视图的情况，这种情况属于比较传统的**Spring MVC 的应用，对应于前端不分离的情况。**



@RestController 返回 JSON 或 XML 形式数据但 @RestController 只返回对象，对象数据直接以 JSON 或 XML 形式写入 HTTP 响应(Response)中，这种情况属于 RESTful Web 服务，这也是目前日常开发所接触的最常用的情况（**前后端分离**）。



15.@Resource 和@Autowired 区别？

- @Autowired 是按照ByType类型注入的，默认情况下它要求注入的对象必须存在。如果允取空值，那么设置required属性为false。可以配合使用@Qualifier 注解按照ByName 类型注入。
- @Resource 默认是按照Byname类型注入的。@Resource 有两个属性name 和type 使用哪个就用哪种形式注入。如果不指定name 和type 默认使用ByName 注入。

16. Spring MVC常用注解

1. @RequestBody :

- 直接获取请求体的内容，直接使用得到的是 key = value & key = value 。。结构的数据。不适用 Get 请求的方式
 - required: 是否必须有请求体。默认值是ture。当取值为true时，get请求方式就会报错。如果取值为 false，get请求得到是null

```
/*
 * 获取到请求体的内容
 * @return
 */
@RequestMapping("/testRequestBody")
public String testRequestBody(@RequestBody String body){
    System.out.println("执行了...");
    System.out.println(body);
    return "success";
}
```

username=hehe&age=20

2. @RequestParam

- 把请求中指定名称的参数给控制器中的形参赋值
 - value: 请求参数的名称
 - required : 请求参数中是否必须提供此参数。默认是 true 。表示必须提供。
- 当所传参数不同可以使用此注解，但是一旦使用所传参数必须和RequestParam 中的name保持相同名称。

3. @PathVariable.

- 用于绑定 url 中的占位符，例如 url 中 /find/{id} , 这个 id 就是 url 占位符。
 - value: 用于指定url中占位符名称
 - required: 是否必须提供占位符

```
/*
 * PathVariable注解
 * @return
 */
@RequestMapping(value="/testPathVariable/{sid}")
public String testPathVariable(@PathVariable(name="sid") String id){
    System.out.println("执行了...");
    System.out.println(id);
    return "success";
}
```

4 . @RequestHeader

- 用于获取请求消息头。
 - value: 提供消息头名称
 - required: 是否必须有此消息头

```
@RequestMapping(value="/testRequestHeader")
public String testRequestHeader(@RequestHeader(value="Accept") String header,
```

5. @ReponseBody

- 在使用 @RequestMapping后，返回值通常解析为跳转路径，但是加上 @ResponseBody 后返回结果不会被解析为跳转路径，而是直接写入 HTTP response body 中。比如异步获取 json 数据，加上 @ResponseBody 后，会直接返回 json 数据。

6.@RequestMapping

- 用来处理请求地址映射的注解，可用于类或方法上。用于类上代表所有方法的请求都以该地址作为父路径

value： 指定请求的实际地址，指定的地址可以是URI Template 模式（后面将会说明）；

method： 指定请求的method类型， GET、 POST、 PUT、 DELETE等；

consumes： 指定处理请求的提交内容类型（Content-Type），例如application/json, text/html；

produces： 指定返回的内容类型，仅当request请求头中的(Accept)类型中包含该指定类型才返回；

params： 指定request中必须包含某些参数值是，才让该方法处理。

headers： 指定request中必须包含某些指定的header值，才能让该方法处理请求。

☆Spring boot☆

1.什么是 spring boot?

在Spring框架这个大家族中，产生了很多衍生框架，比如 Spring、SpringMvc框架等，Spring的核心内容在于控制反转(IOC)和依赖注入(DI),所谓控制反转并非是一种技术，而是一种思想，在操作方面是指在spring配置文件中创建，依赖注入即为由spring容器为应用程序的某个对象提供资源，比如引用对象、常量数据等。

SpringBoot是一个框架，一种全新的编程规范，他的产生**简化了框架的使用**，所谓**简化是指简化了Spring众多框架中所需的大量且繁琐的配置文件**，所以 SpringBoot是一个服务于框架的框架，服务范围是简化配置文件。

为什么要用 spring boot?

- Spring Boot使编码变简单
- Spring Boot使配置变简单
- Spring Boot使部署变简单
- Spring Boot使监控变简单
- Spring的不足

什么是Spring Boot的启动流程?

第一部分进行SpringApplication的初始化模块，配置一些基本的环境变量、资源、构造器、监视器。

第二部分实现了具体的启动方案：启动流程的监听模块、加载配置环境模块、核心的创建上下文环境模块。

第三部分是自动化配置模块，该模块作为Springboot自动配置核心

启动

- 每个Spring boot 项目在启动时都需要调用SpringApplication.run() 启动。该方法需要使用 @SpringBootApplication 注解。**其中包含**
 - @EnableAutoConfiguration： SpringBoot根据所声明的依赖对Spring 框架进行自动配置。

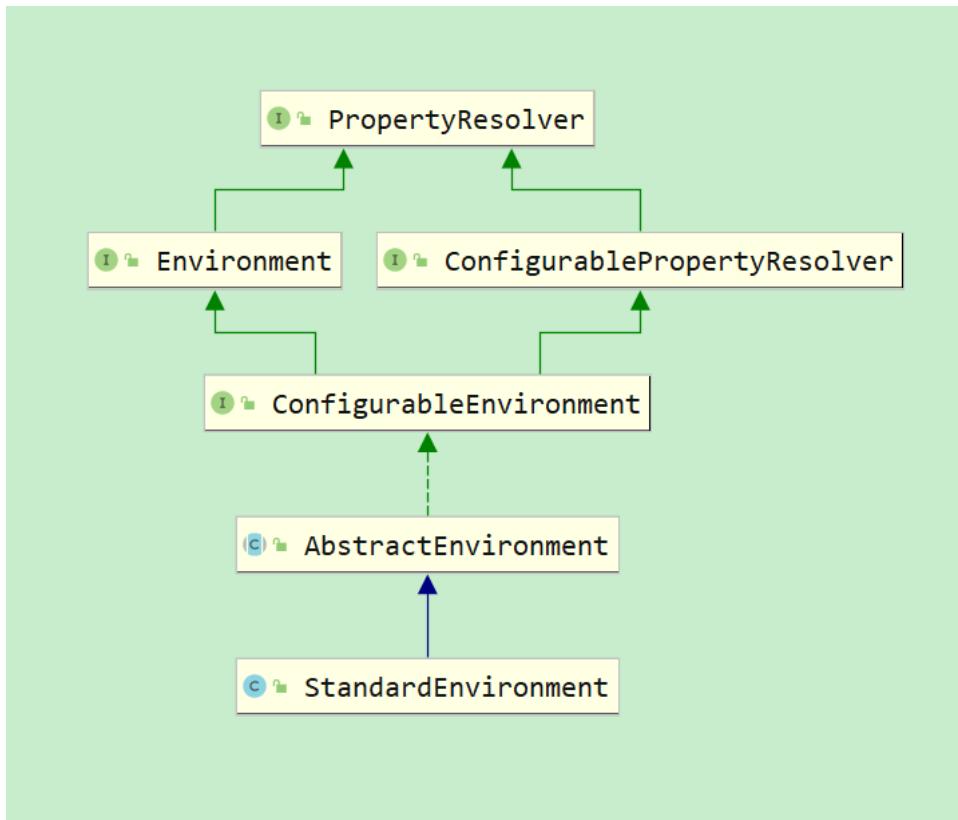
- @SpringBootConfiguration 装配所有bean事务，提供一个Spring的上下文环境。
- @ComponentScan 组件扫描，可以自动发现和装配Bean，默认扫描SpringApplication 所在包路径下文件，通常把启动类放到根路径下。

首先进入run()方法，run 方法创建一个SpringApplication 实例，在构造方法内，调用了一个 initialize 方法。该方法主要给SpringApplication对象赋一些初值。

```

public ConfigurableApplicationContext run(String... args) {
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.start();
    ConfigurableApplicationContext context = null;
    FailureAnalyzers analyzers = null;
    configureHeadlessProperty();
    //1. 创建应用监听器并开始监听
    SpringApplicationRunListeners listeners = getRunListeners(args);
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
DefaultApplicationArguments(
            args);
        //2. 加载SpringBoot配置环境
        ConfigurableEnvironment environment =
prepareEnvironment(listeners,
            applicationArguments);
        Banner printedBanner = printBanner(environment);
        //4. 创建run 方法的返回对象。
        context = createApplicationContext();
        analyzers = new FailureAnalyzers(context);
        prepareContext(context, environment, listeners,
applicationArguments,
            printedBanner);
        refreshContext(context);
        afterRefresh(context, applicationArguments);
        listeners.finished(context, null);
        stopwatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopwatch);
        }
        return context;
    }
    catch (Throwable ex) {
        handleRunFailure(context, listeners, analyzers, ex);
        throw new IllegalStateException(ex);
    }
}

```



可以看出，*Environment最终都实现了PropertyResolver接口，我们平时通过environment对象获取配置文件中指定Key对应的value方法时，就是调用了propertyResolver接口的getProperty方法

3. 配置环境(Environment)加入到监听器对象中(SpringApplicationRunListeners)

4. 创建run方法的返回对象：ConfigurableApplicationContext(应用配置上下文)，我们可以看一下创建方法：

```

protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            contextClass = Class.forName(this.webEnvironment
                ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
        }
        catch (ClassNotFoundException ex) {
            throw new IllegalStateException(
                "Unable to create a default ApplicationContext, "
                + "please specify an ApplicationContextClass",
                ex);
        }
    }
    return (ConfigurableApplicationContext)
        Beanutils.instantiate(contextClass);
}
  
```

方法会先获取显式设置的应用上下文(applicationContextClass)，如果不存在，再加载默认的环境配置（通过是否是web environment判断），默认选择AnnotationConfigApplicationContext注解上下文（通过扫描所有注解类来加载bean），最后通过BeanUtils实例化上下文对象，并返回，ConfigurableApplicationContext类图如下：

Spring boot 的启动，主要创建了配置环境（environment）、事件监听、应用上下文（applicationContext），并基于以上条件，在容器中实例化我们所需要的bean，至此，Spring boot启动的程序已经构造完成。

2.spring boot 核心配置文件是什么？

Spring Boot提供了两种常用的配置文件：

- properties文件，
- yml文件 yml文件更年轻，也有很多的坑，可谓成也萧何败萧何，yml通过空格来确定层级关系，使配置文件结构跟清晰，但也会因为微不足道的空格而破坏了层级关系。

Spring Boot 的核心注解是哪个？它主要由哪几个注解组成的？

启动类上面的注解是@SpringBootApplication，它也是 Spring Boot 的核心注解，主要组合包含了以下 3 个注解：

@SpringBootConfiguration：组合了 @Configuration 注解，实现配置文件的功能。

@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能：@SpringBootApplication(exclude = {
DataSourceAutoConfiguration.class })。

@ComponentScan：Spring组件扫描。

自动配置原理是什么？

- 注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是自动配置的核心，
- @EnableAutoConfiguration 给容器导入META-INF/spring.factories 里定义的自动配置类。
- 筛选有效的自动配置类。
- 每一个自动配置类结合对应的 xxxProperties.java 读取配置文件进行自动配置功能

Spring Boot 配置加载顺序？

在 Spring Boot 里面，可以使用以下几种方式来加载配置。

- 1) properties文件；
- 2) YAML文件；
- 3) 系统环境变量；
- 4) 命令行参数；

什么是 YAML？

YAML 是一种人类可读的数据序列化语言。它通常用于配置文件。与属性文件相比，如果我们要在配置文件中添加复杂的属性，YAML 文件就更加结构化，而且更少混淆。可以看出 YAML 具有分层配置数据。

Spring Boot 是否可以使用 XML 配置？

Spring Boot 推荐使用 Java 配置而非 XML 配置，但是 Spring Boot 中也可以使用 XML 配置，通过 @ImportResource 注解可以引入一个 XML 配置。

spring boot 核心配置文件是什么？

spring boot 核心的两个配置文件：

- bootstrap (.yml 或者 .properties): bootstrap 由父 ApplicationContext 加载的，比 application 优先加载，配置在应用程序上下文的引导阶段生效。一般来说我们在 Spring Cloud Config 或者 Nacos 中会用到它。且 bootstrap 里面的属性不能被覆盖；
- application (.yml 或者 .properties): 由 ApplicationContext 加载，用于 spring boot 项目的自动化配置。

3.Spring boot 中的监视器是什么

actuator 执行机构

Spring boot actuator 是 spring 启动框架中的重要功能之一。Spring boot 监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。有几个指标必须在生产环境中进行检查和监控。即使一些外部应用程序可能正在使用这些服务来向相关人员触发警报消息。监视器模块公开了一组可直接作为 HTTP URL 访问的 REST 端点来检查状态。

4.如何在自定义端口上运行 Spring boot

为了在自定义端口上运行 Spring Boot 应用程序，您可以在 application.properties 中指定端口。

server.port = 8090

5.什么是Spring Profiles?

Spring Profiles 允许用户根据配置文件(dev, test, prod 等)来注册 bean。因此，当应用程序在开发中运行时，只有某些 bean 可以加载，而在 PRODUCTION 中，某些其他 bean 可以加载。假设我们的要求是 Swagger 文档仅适用于 QA 环境，并且禁用所有其他文档。这可以使用配置文件来完成。Spring Boot 使得使用配置文件非常简单。

16、如何使用 Spring Boot 实现异常处理？

Spring 提供了一种使用 ControllerAdvice 处理异常的非常有用的方法。我们通过实现一个 ControllerAdvice 类，来处理控制器类抛出的所有异常。

6.请谈谈什么是RESTful

REST 这个词是 Roy Thomas Fielding 在 2000 年的博士论文中提出的。

REST (Representational State Transfer) 即表现层状态转化

资源 (Resources)

- 表现层指的是“资源”的“表现层”，所谓资源就是网络上的一个实体，或者说是网络上的一个具体信息。它可以是一张图片、一首歌曲、一种服务，总之是一个具体的实在。你可以用一个 URI 指向它，每种资源对应一个特定的 URI。要获取资源访问对应的 URI 即可，URI 是每一个资源独一无二的地址或独一无二的标识符。所谓上网就是与互联网上的一系列的“资源”互动，调用它的 URI。

表现层 (Representation)

- 资源是一种信息实体，它可以有多种外在表现形式。我们把资源呈现出来的形式叫做它的表现层 (representation)。比如，文本可以用 txt 格式表现，也可以用 HTML 格式、XML 格式、JSON 格式表现，甚至可以采用二进制格式；图片可以采用 JPG 格式表现，也可以使用 PNG 格式表现。
- URI 只代表资源的实体，不代表它的形式。
- 严格说的，有些网址最后的 .HTML 后缀名是不必要的，因为这个后缀名表示格式，属于表现层的范畴，而 URI 代表的是资源的位置，它的具体表现形式，应该在 HTTP 请求的头信息用 Accept 和 Content-Type 字段指定，这两个才是表现层的描述。

状态转化 (State Transfer)

- 访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。
- HTTP 协议是一个无状态协议，这意味着所有状态都保存在服务器端，因此客户端想要操作数据，必须通过某种手段，让服务器发生状态转化 (State Transfer)。而这种转化是建立在表现层之上的，所以就是表现层状态转化。
- 客户端用到的只能是 HTTP 协议，具体来说就是四个表示方式的动词：POST（新建资源、更新资源）、GET（获取资源）、PUT（更新资源）、DELETE（删除资源）。

总的来说

- 每一个 URI 代表一种资源
- 客户端和服务器之间传递这种资源的某种表现层
- 客户端通过四个 HTTP 动词，对服务端资源进行操作，实现表现层状态转化

☆ Spring Cloud ☆

1. 什么是 spring cloud?

从字面理解，Spring Cloud 就是致力于分布式系统、云服务的框架。

Spring Cloud 是整个 Spring 家族中新的成员，是最近云服务火爆的必然产物。

Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具，例如：

- 配置管理
- 服务注册与发现
- 断路器
- 智能路由
- 服务间调用
- 负载均衡
- 微代理
- 控制总线
- 一次性令牌
- 全局锁
- 领导选举
- 分布式会话
- 集群状态
- 分布式消息

使用 Spring Cloud 开发人员可以开箱即用的实现这些模式的服务和应用程序。这些服务可以任何环境下运行，包括分布式环境，也包括开发人员自己的笔记本电脑以及各种托管平台。

2.spring cloud 断路器的作用是什么？

断路器（Hystrix）可以防止一个应用程序多次试图执行一个操作，即很可能失败，导致整个系统瘫痪。

当某个服务单元发生故障之后，通过断路器的故障监控（类似保险丝），像调用放返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间得不到释放，避免了故障在分布式系统中的蔓延。

3.spring cloud 的核心组件有哪些？

① 服务发现——Netflix Eureka

一个RESTful服务，用来定位运行在AWS地区（Region）中的中间层服务。由两个组件组成：Eureka服务器和Eureka客户端。Eureka服务器用作服务注册服务器。Eureka客户端是一个java客户端，用来简化与服务器的交互、作为轮询负载均衡器，并提供服务的故障切换支持。Netflix在其生产环境中使用的是另外的客户端，它提供基于流量、资源利用率以及出错状态的加权负载均衡。

② 客服端负载均衡——Netflix Ribbon

Ribbon，主要提供客户侧的软件负载均衡算法。Ribbon客户端组件提供一系列完善的配置选项，比如连接超时、重试、重试算法等。Ribbon内置可插拔、可定制的负载均衡组件。

③ 断路器——Netflix Hystrix

断路器可以防止一个应用程序多次试图执行一个操作，即很可能失败，允许它继续而不等待故障恢复或者浪费CPU周期，而它确定该故障是持久的。断路器模式也使应用程序能够检测故障是否已经解决。如果问题似乎已经得到纠正，应用程序可以尝试调用操作。

④ 服务网关——Netflix Zuul

类似nginx，反向代理的功能，不过netflix自己增加了一些配合其他组件的特性。

⑤ 分布式配置——Spring Cloud Config

这个还是静态的，得配合Spring Cloud Bus实现动态的配置更新。

4.服务注册和发现是什么意思？

当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。Eureka服务注册和发现可以在这种情况下提供帮助。由于所有服务都在Eureka服务器上注册并通过调用Eureka服务器完成查找，因此无需处理服务地点的任何更改和处理。

5.负载均衡的意义是什么

可以改善跨计算机，计算机集群，网络连接，中央处理单元或磁盘驱动等多种计算

负载均衡指在优化资源使用，最大化吞吐量，最小化响应时间，防止任何单一资源的过载

使用多个组件进行负载均衡而不是单个组件可能会通过冗余来提高高可用性和高可靠性。

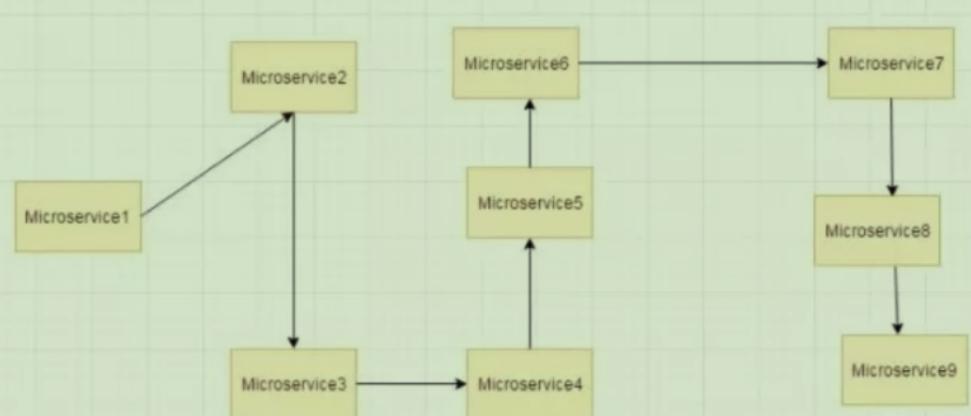
6.什么是 Hystrix？它如何实现容错

Hystrix 是一个延迟和容错库，指在隔离远程系统，服务和第三方的访问点，当出现不可避免的故障时，停止级联故障并在复杂的分布式系统中时下你弹性。

当服务出现了故障后，就使用我们预先定义的回退方法

通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。

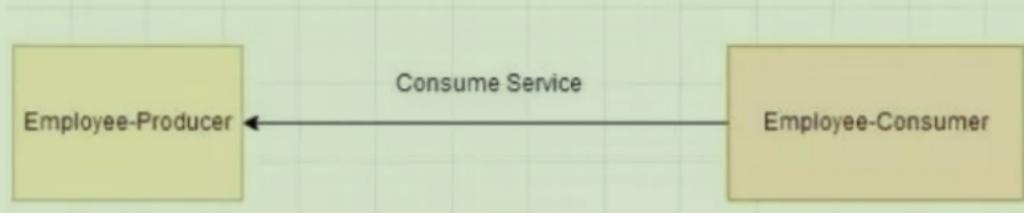
思考以下微服务



假设如果上图中的微服务 9 失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。

随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达 1000. 这是 hystrix 出现的地方，我们将使用 Hystrix 在这种情况下的 Fallback 方法功能。我们有两个服务 employee-consumer 使用由 employee-producer 公开的服务。

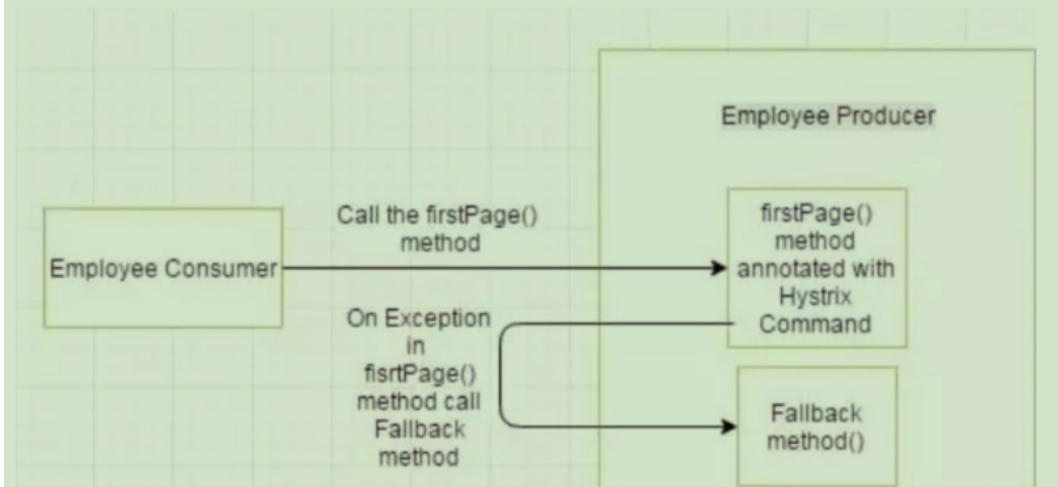
简化图如下所示



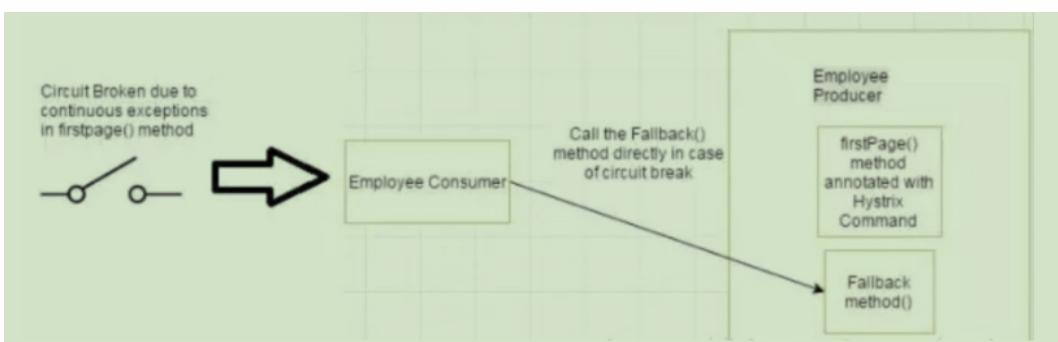
现在假设由于某种原因，employee-producer 公开的服务会抛出异常。我们在这种情况下使用 Hystrix 定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。

6、什么是 Hystrix 断路器?我们需要它吗?

由于某些原因, employee-consumer 公开服务会引发异常。在这种情况下使用 Hystrix 我们定义了一个回退方法。如果在公开服务中发生异常, 则回退方法返回一些默认值。



如果 firstPage method() 中的异常继续发生, 则 Hystrix 电路将中断, 并且员工使用者将一起跳过 firstPage 方法, 并直接调用回退方法。断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间, 并导致异常恢复。可能发生的情况是, 在负载较小的情况下, 导致异常的问题有更好的恢复机会。



7.什么是 Netflix Feign

Feign 是受到 Retrofit, JAXRS-2.0 和 WebSocket 启发的 java 客户端联编程序。Feign 的第一个目标是将约束分母的复杂性统一到 http apis, 而不考虑其稳定性。在 employee-consumer 的例子中, 我们使用了 employee-producer 使用 REST 模板公开的 REST 服务。

但是我们必须编写大量代码才能执行以下步骤

- 使用功能区进行负载平衡。
- 获取服务实例, 然后获取基本 URL。
- 利用 REST 模板来使用服务。前面的代码如下

8, Feign 的优缺点

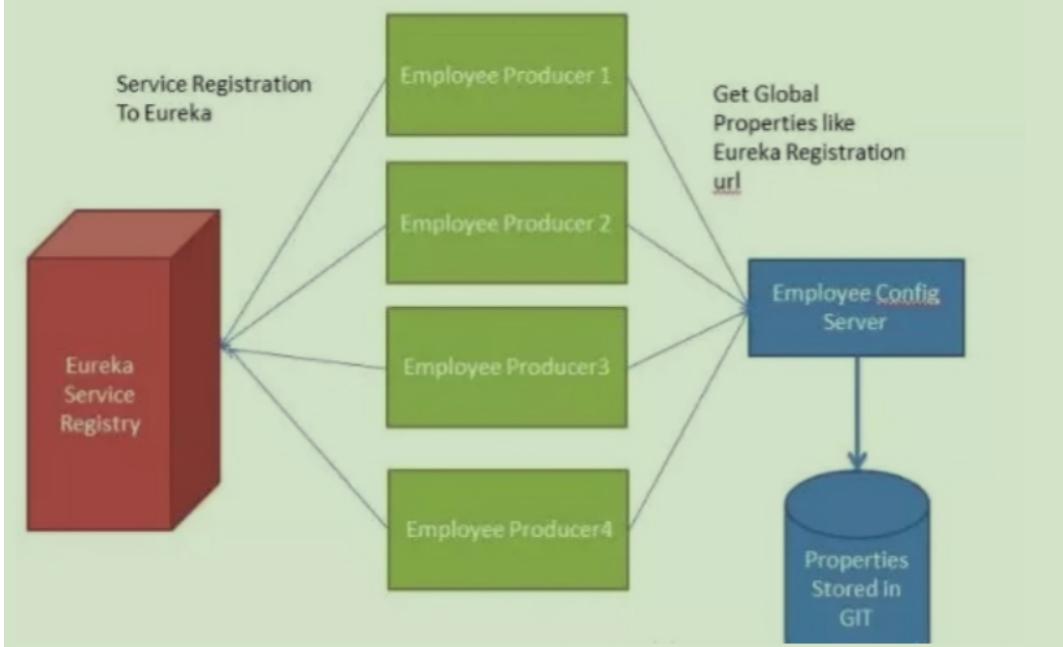
优点 使用Spring Cloud Feign继承特性的优点很明显，可以将接口的定义从Controller中剥离，同时配合Maven私有仓库就可以轻易地实现接口定义的共享，实现在构建期的接口绑定，从而有效减少服务客户端的绑定配置。这么做虽然可以很方便地实现接口定义和依赖的共享，不用再复制粘贴接口进行绑定，但是这样的做法使用不当的话会带来副作用。

缺点 由于接口在构建期间就建立起了依赖，那么接口变动就会对项目构建造成影响，可能服务提供方修改了一个接口定义，那么会直接导致客户端工程的构建失败。所以，如果开发团队通过此方法来实现接口共享的话，建议在开发评审期间严格遵守面向对象的开闭原则，尽可能地做好前后版本的兼容，防止牵一发而动全身的后果，增加团队不必要的维护工作量

9.什么是Spring Cloud Bus

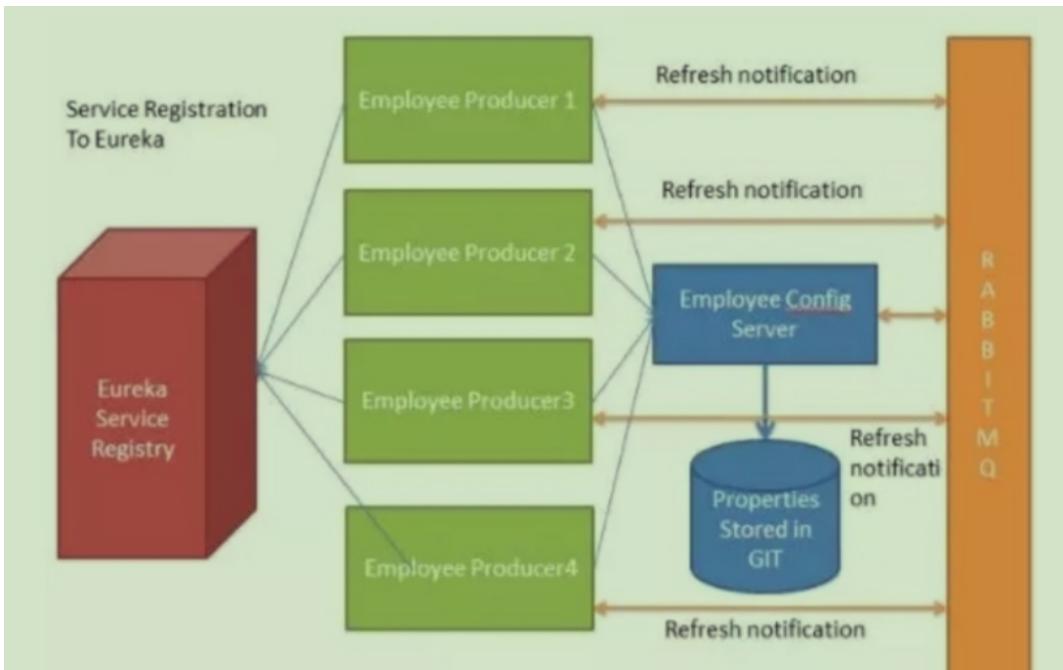
考虑以下情况:我们有多个应用程序使用 Spring Cloud Config 读取属性，而 Spring Cloud Config 从 GIT 读取这些属性。

下面的例子中多个员工生产者模块从 Employee Config Module 获取 Eureka 注册的财产。



如果假设 GIT 中的 Eureka 注册属性更改为指向另一台 Eureka 服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。

还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个 url。例如，如果 Employee Producer1 部署在端口 8080 上，则调用 `http:// localhost:8080 / refresh`。同样对于 Employee Producer2 `http:// localhost:8081 / refresh` 等等。这又很麻烦。这就是 Spring Cloud Bus 发挥作用的地方。



Spring Cloud Bus 提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新 Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。

☆ Hibernate ☆

1. hibernate 的执行流程

通过 Configuration.configure() 读取并解析 hibernate.cfg.xml 配置文件；
由 hibernate.cfg.xml 中的<mapping resource="com/xx/User.hbm.xml" />
读取并解析映射信息；
通过 config.buildSessionFactory() 创建 SessionFactory；
sessionFactory.openSession() 打开 session；
session.beginTransaction() 创建事务 transaction；
persistent operate (持久化操作)；
session.getTransaction().commit() 提交事务；
关闭 session；
关闭 SessionFactory。

2. Hibernate 缓存、延迟加载

Hibernate 一级缓存又称为“Session 的缓存”。Session 内置不能被卸载，Session 的缓存是事务范围的缓存（Session 对象的生命周期通常对应一个数据库事务或者一个应用事务）。一级缓存中，持久化类的每个实例都具有唯一的 OID；

Hibernate 二级缓存又称为“SessionFactory 的缓存”。由于 SessionFactory 对象的生命周期和应用程序的整个过程对应，因此 Hibernate 二级缓存是进程范围或者集群范围的缓存，有可能出现并发问题，因此需要采用适当的并发访问策略，该策略为被缓存的数据提供了事务隔离级别。第二级缓存是可选的，是一个可配置的插件，默认下 SessionFactory 不会启用这个插件。

Session 的延迟加载实现要解决两个问题：正常关闭连接、确保请求中访问的是同一个 session。Hibernate session 就是 java.sql.Connection 的一层高级封装，一个 session 对应了一个 Connection。http 请求结束后正确的关闭 session（过滤器实现了 session 的正常关闭），延迟加载必须保证是同一个 session（session 绑定在 ThreadLocal）。

☆ Mybatis ☆

理解 Mybatis

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手工设置参数以及抽取结果集。MyBatis 使用简单的 XML 或注解来配置和映射基本体，将接口和 Java 的 POJOs(Plain Old Java Objects, 普通的 Java 对象)映射成数据库中的记录。

Mybatis 是如何解决 JDBC 的不足之处的

① 数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

解决：在 SqlMapConfig.xml 中配置数据链接池，使用连接池管理数据库链接。

② Sql 语句写在代码中造成代码不易维护，实际应用 sql 变化的可能较大，sql 变动需要改变 java 代码。

解决：将 Sql 语句配置在 XXXXmapper.xml 文件中与 java 代码分离。

③ 向 sql 语句传参数麻烦，因为 sql 语句的 where 条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis 自动将 java 对象映射至 sql 语句。

④ 对结果集解析麻烦，sql 变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成 pojo 对象解析比较方便。

解决：Mybatis 自动将 sql 执行结果映射至 java 对象。

Mybatis 的编程步骤

- ① 创建 SqlSessionFactory
- ② 通过 SqlSessionFactory 创建 SqlSession
- ③ 通过 sqlSession 执行数据库操作
- ④ 调用 session.commit() 提交事务
- ⑤ 调用 session.close() 关闭会话

1. #{} 和 \${} 的区别是什么？

- #{} 是预编译处理，\${} 是字符串替换；
- Mybatis 在处理#{} 时，会将 sql 中的#{} 替换为?号，调用 PreparedStatement 的 set 方法来赋值；
- Mybatis 在处理\${} 时，就是把 \${} 替换成变量的值；
- 使用#{} 可以有效的防止 SQL 注入，提高系统安全性。

2. Mybatis 分页

1. mybatis 有几种分页方式？

- 数组分页
- sql 分页
- **拦截器分页(物理分页)**：分页插件的基本原理是使用 Mybatis 提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的 sql，然后重写 sql，根据 dialect 方言，添加对应的物理分页语句和物理分页参数。
- **RowBounds 分页(逻辑分页)**：针对 ResultSet 结果集执行的内存分页，而非物理分页

2. mybatis 逻辑分页和物理分页的区别是什么？

- 物理分页速度上并不一定快于逻辑分页，逻辑分页速度上也并不一定快于物理分页。
- 物理分页总是优于逻辑分页：没有必要将属于数据库端的压力加诸到应用端来，就算速度上存在优势，然而其它性能上的优点足以弥补这个缺点。

3.简述 Mybatis 的插件运行原理，以及如何编写一个插件。

答：Mybatis 仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor 这 4 种接口的插件，**Mybatis 使用 JDK 的动态代理**，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这 4 种接口对象的方法时，就会进入拦截方法，具体就是 InvocationHandler 的 invoke() 方法，当然，只会拦截那些你指定需要拦截的方法。

实现 Mybatis 的 Interceptor 接口并复写 intercept() 方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

4.mybatis 分页插件的实现原理是什么？

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数

3.延迟加载、缓存

1.mybatis 是否支持延迟加载？延迟加载的原理是什么？

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，

association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

2.说一下 mybatis 的一级缓存和二级缓存？

Mybatis 首先去缓存中查询结果集，如果没有则查询数据库，如果有则从缓存取出返回结果集就不是数据库。Mybatis 内部存储缓存使用一个 HashMap，key 为 hashCode+sqlId+Sql 语句。value 为从查询出来映射生成的 java 对象

Mybatis 的二级缓存即查询缓存，它的作用域是一个 mapper 的 namespace，即在同一个 namespace 中查询 sql 可以从缓存中获取数据。二级缓存是可以跨 SqlSession 的。

一级缓存：基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。

默认不打开二级缓存，要开启二级缓存，**使用二级缓存属性类需要实现Serializable序列化接口**（可用来保存对象的状态），可在它的映射文件中配置；

对于缓存数据更新机制，当某一个作用域（一级缓存 Session/二级缓存Namespaces）的进行了 C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

4.批处理

1.Mybatis 中如何执行批处理

使用BatchExecutor 完成批处理

2.Mybatis 执行批量插入，能返回数据库主键列表吗？

答：能， JDBC 都能， Mybatis 当然也能。

5.mybatis 和 hibernate 的区别有哪些？

(1) Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句。

(2) Mybatis直接编写原生态sql，可以严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。但是灵活的前提是**mybatis无法做到数据库无关性**，如果需要实现支持多种数据库的软件，则需要自定义多套sql映射文件，工作量大。

(3) Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用hibernate开发可以节省很多代码，提高效率。

6.Mybatis执行器

1.mybatis 有哪些执行器（Executor）？

Mybatis有三种基本的执行器（Executor）：

1. **SimpleExecutor**: 每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。
2. **ReuseExecutor**: 执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，**用完后，不关闭Statement对象**，而是放置于Map内，供下一次使用。简言之，就是重复使用Statement对象。
3. **BatchExecutor**: 执行update（没有select， JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待**统一执行**（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

2.Mybatis 中如何指定使用哪一种 Executor 执行器？

在 Mybatis 配置文件中，可以指定默认的 ExecutorType 执行器类型，也可以手动给 DefaultSqlSessionFactory 的创建 SqlSession 的方法传递 ExecutorType 类型参数。.

7.动态SQL

Mybatis 动态 sql 是做什么的？都有哪些动态 sql？能简述一下动态 sql 的执行原理不？

答：Mybatis 动态 sql 可以让我们在 Xml 映射文件内，以标签的形式编写动态 sql，完成逻辑判断和动态拼接 sql 的功能，Mybatis 提供了 9 种动态 sql 标签
trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用 OGNL 从sql 参数对象中计算表达式的值，根据表达式的值动态拼接 sql，以此来完成动态 sql 的功能。

8.Mybatis 是如何将 sql 执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用 `label`，逐一定义列名和对象属性名之间的映射关系。

第二种是使用sql 列的别名功能，将列别名书写为对象属性名，比如 `T_NAME AS NAME`，对象属性名一般是 `name`，小写，但是列名不区分大小写，**Mybatis 会忽略列名大小写**，智能找到与之对应对象属性名，你甚至可以写成 `T_NAME AS NaMe`，Mybatis 一样可以正常工作。

有了列名与属性名的映射关系后，Mybatis 通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。.

9.Mybatis 有几种查询方式？各有什么区别？

1.MyBatis 实现一对一有几种方式

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次,通过在 `resultMap` 里面配置 `association` 节点配置一对一的类就可以完成;嵌套查询是先查一个表,根据这个表里面的结果的外键 `id`,去再另外一个表里面查询数据,也是通过 `association` 配置,但另外一个表的查询通过 `select` 属性配置。

2.Mybatis 能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别。

能，Mybatis 不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实是一对一查询，只需要把 `selectOne()` 修改为 `selectList()` 即可；多对多查询，其实是一对多查询，只需要把 `selectOne()` 修改为 `selectList()` 即可。

关联对象查询，有两种实现方式，一种是单独发送一个 sql 去查询关联对象，赋给主对象，然后返回主对象。

另一种是使用嵌套查询，嵌套查询的含义为使用 `join` 查询，一部分列是 A 对象的属性值，另外一部分列是关联对象 B 的属性值，好处是只发一个 sql 查询，就可以把主对象和其关联对象查出来。

那么问题来了，`join` 查询出来 100 条记录，如何确定主对象是 5 个，而不是 100 个？其去重复的原理是标签内的子标签，指定了唯一确定一条记录的 `id` 列，Mybatis 根据列值来完成 100 条记录的去重复功能，可以有多个，代表了联合主键的语意。

同样主对象的关联对象，也是根据这个原理去重复的，尽管一般情况下，只有主对象会有重复记录，关联对象一般不会重复。

举例：下面 `join` 查询出来 6 条记录，一、二列是 `Teacher` 对象列，第三列为 `Student` 对象列，Mybatis 去重复处理后，结果为 1 个老师 6 个学生，而不是 6 个老师 6 个学生。
`t_id t_name
s_id | 1 | teacher | 38 | | 1 | teacher | 39 | | 1 | teacher | 40 | | 1 | teacher | 41 | | 1 |
teacher | 42 | | 1 | teacher | 43 |`

3.如果要查询的表名和返回的实体Bean对象不一致,那你是怎么处理的？

在MyBatis里面最主要最灵活的一个映射对象的ResultMap,在它里面可以映射键值对,默认里面有id节点,result节点,它可以映射表里面的列名和对象里面的字段名. 并且在一对一,一对多的情况下结果集也一定要用ResultMap

11.映射文件

1.简述Mybatis 的XML 映射文件和Mybatis 内部数据结构之间的映射关系

答： Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。

在 Xml映射文件中， 标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。

标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。

每一个

```
Entity(name="USER")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "ID")
    private Long id;
    @Column(name="USER_NAME")
    private String userName;
    @Column(name="PASSWORD")
    private String password;
    private String secrect;
}
```

如果我们想让 secret 这个字段不被持久化，也就是不被数据库存储怎么办？我们可以采用下面几种方法：

```
static String transient1; // not persistent because of static
final String transient2 = "Satish"; // not persistent because of final
transient String transient3; // not persistent because of transient
@Transient
String transient4; // not persistent because of @Transient
```

☆设计模式☆

1. 单例模式

一个应用程序中，某个类的实例对象只有一个，无法通过构造函数 new 出来，因为构造器被 private 修饰，一般通过 getInstance() 的方法来获取他们的实例

实现一：静态内部类

```

public class Singleton {
    private Singleton() {
    }

    private static class Holder {
        private static Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        return Holder.instance;
    }
}

```

实现二：延迟加载 / “懒汉模式”

- 延迟加载就是调用get()方法时实例才被创建（先不急着实例化出对象，等要用的时候才给你创建出来。不着急，故又称为“懒汉模式”），常见的实现方法就是在get方法中进行new实例化。

```

public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance==null){
            singleton=new Singleton();
        }
        return singleton;
    }
}

```

“懒汉模式”的优缺点：

- **优点：**实现起来比较简单，当类SingletonTest被加载的时候，静态变量static的instance未被创建并分配内存空间，当getInstance方法第一次被调用时，初始化instance变量，并分配内存，因此在某些特定条件下会节约了内存。
- **缺点：**在多线程环境中，这种实现方法是完全错误的，根本不能保证单例的状态。

实现三：线程安全的“懒汉模式”

```

public class Singleton {

    // 将自身实例化对象设置为一个属性，并用static修饰
    private static Singleton instance;

    // 构造方法私有化
    private Singleton() {}

    // 静态方法返回该实例，加synchronized关键字实现同步
    public static synchronized Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}

```

- 优点：在多线程情形下，保证了“懒汉模式”的线程安全。
- 缺点：众所周知在多线程情形下，**synchronized方法通常效率低**，显然这不是最佳的实现方案。

实现四：饿汉式写法/立即加载

- 立即加载就是使用类的时候已经将对象创建完毕（不管以后会不会使用到该实例化对象，先创建了再说。很着急的样子，故又被称为“饿汉模式”），常见的实现办法就是直接new实例化。

```
public class Singleton {
    // 将自身实例化对象设置为一个属性，并用static、final修饰
    private static final Singleton instance = new Singleton();

    // 构造方法私有化
    private Singleton() {}

    // 静态方法返回该实例
    public static Singleton getInstance() {
        return instance;
    }
}
```

“饿汉模式”的优缺点：

优点：实现起来简单，没有多线程同步问题。

缺点：当类SingletonTest被加载的时候，会初始化static的instance，静态变量被创建并分配内存空间，从这以后，这个static的instance对象便一直占着这段内存（即便你还没有用到这个实例），当类被卸载时，静态变量被摧毁，并释放所占有的内存，因此在某些特定条件下会**耗费内存**。

实现五：双重检验锁线程安全

```
public class Singleton {
    // 首先，也是先堵死 new Singleton() 这条路
    private Singleton() {}

    //volatile保证可见性
    private static volatile Singleton instance=null;

    public static Singleton getInstance() {
        if (instance==null){
            //加锁
            synchronized (Singleton.class){
                // 这一次判断也是必须的，不然会有并发问题
                if (instance==null){
                    instance=new Singleton();
                }
            }
        }
        return instance;
    }
}
```

- 单例模式的最佳实现方式。内存占用率高，效率高，线程安全，多线程操作原子性。
- 双重检查，指的是两次检查 instance 是否为 null。
- volatile 在这里是需要的，希望能引起读者的关注。

2. 观察者模式

- 在对象之间定义一个的依赖，这样一来，当一个对象改变状态，依赖它的对象都会收到通知，并自动更新

```
public interface Person {  
    void getMessage(String s);  
}  
  
public class Laowang implements Person {  
    String name = "老王";  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：" + s);  
    }  
}  
  
public class XiaoLi implements Person {  
    String name = "小李";  
  
    @Override  
    public void getMessage(String s) {  
        System.out.println(name + "接到了小美打过来的电话，电话内容是：-？？" + s);  
    }  
}  
  
public class XiaoMei {  
    List<Person> list = new ArrayList<>();  
  
    public void addPerson(Person person) {  
        list.add(person);  
    }  
  
    public void notifyPerson() {  
        for (Person person : list) {  
            person.getMessage("你们过来吧，谁先过来谁就能陪我一起玩儿游戏！");  
        }  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        XiaoMei xiaoMei = new XiaoMei();  
  
        xiaoMei.addPerson(new Laowang());  
        xiaoMei.addPerson(new XiaoLi());  
  
        xiaoMei.notifyPerson();  
    }  
}
```

测试结果

E:\java\jdk1.8.0_121\bin\java.exe ...

老王接到了小美打过来的电话，电话内容是：你们过来吧，谁先过来谁就能陪我一起玩儿游戏！

小李接到了小美打过来的电话，电话内容是：-？？你们过来吧，谁先过来谁就能陪我一起玩儿游戏！

3.装饰者模式

- 动态的将新功能附加到对象上。在对象功能扩展方面，它比继承更有弹性，装饰者模式也体现了开闭原则(ocp)
- 可以设计出多个不同的具体装饰类，创造出多个不同行为的组合。

你去星巴克想要点一杯饮料，比如咖啡，有时候你还想往咖啡里加些调料比如牛奶、摩卡之类的，第二天你还去星巴克，想要喝点茶，这时候想要往茶里加点调料比如枸杞，这时就需要装饰者模式了

```
//饮料类
public abstract class Beverage {
    String description = "Unkonw!!";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}

//调料类
public abstract class CondimentDecorate extends Beverage {
    @Override
    public abstract String getDescription();
}
```

咖啡系列饮料

```
//浓缩咖啡
public class Espresso extends Beverage {
    public Espresso() {
        description = "浓缩咖啡";
    }

    @Override
    public double cost() {
        return 2;
    }
}

//咖啡里面加的调料 牛奶
public class Milk extends CondimentDecorate {
    private Beverage beverage;

    public Milk(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + "加牛奶";
    }
}
```

```

@Override
public double cost() {
    return beverage.cost() + 1;
}

//咖啡里面加的调料 摩卡
public class Mocha extends CondimentDecorator {

    private Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription()+"加 Mocha";
    }

    @Override
    public double cost() {
        return beverage.cost()+ 1.2;
    }
}

```

茶系列饮料

```

public class Tea extends Beverage {
    public Tea() {
        description = "茶";
    }

    @Override
    public double cost() {
        return 2;
    }
}

//往茶里加枸杞
public class Gouqi extends CondimentDecorator {
    private Beverage beverage;

    public Gouqi(Beverage beverage) {
        this.beverage = beverage;
    }

    @Override
    public String getDescription() {
        return beverage.getDescription() + "加枸杞";
    }

    @Override
    public double cost() {
        return beverage.cost() + 2;
    }
}

```

测试类

```
public class Test {
    public static void main(String[] args) {
        System.out.println("=====Coffee=====");
        Beverage espresso=new Espresso();
        espresso=new Mocha(espresso);
        espresso=new Milk(espresso);

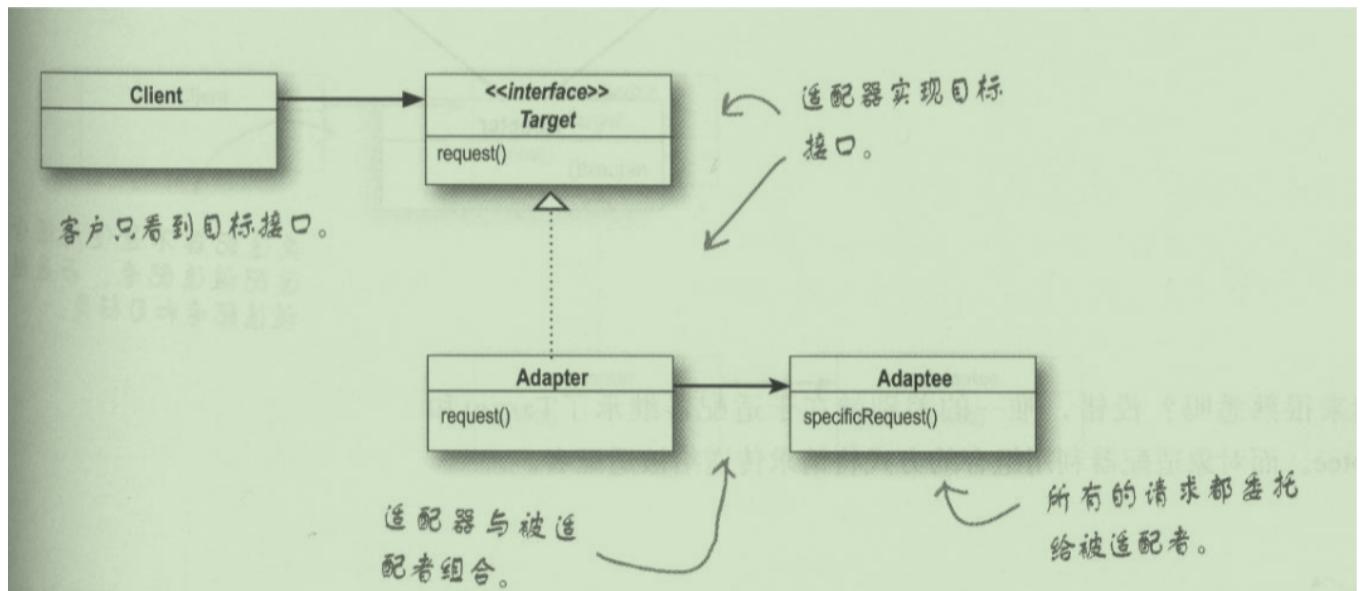
        System.out.println(espresso.getDescription()+"$"+espresso.cost());

        System.out.println("=====Tea=====");
        Beverage tea=new Tea();
        tea=new Gouqi(tea);
        System.out.println(tea.getDescription()+"$"+tea.cost());
    }
}
```

```
=====Coffee=====
浓缩咖啡加 MoCha加牛奶$4.2
=====Tea=====
茶加枸杞$4.0
```

3.适配器模式

- 作为两个不兼容接口之间的桥梁。



简单工厂和抽象工厂有什么区别？

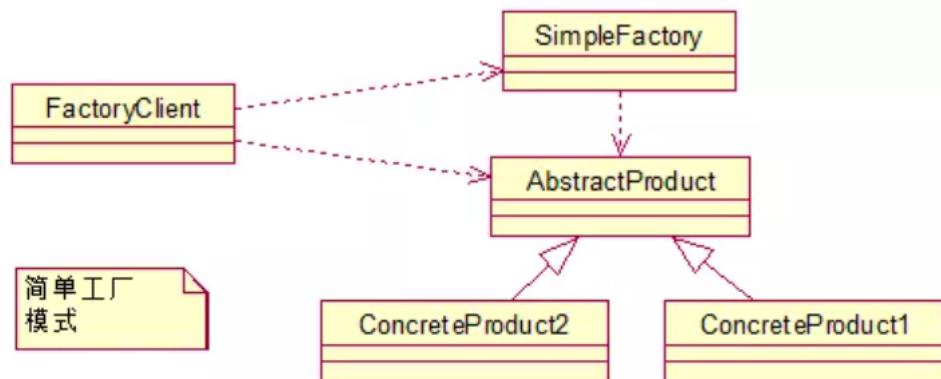
简单工厂模式：

- 这个模式本身很简单而且使用在业务较简单的情况下。一般用于小项目或者具体产品很少扩展的情况下（这样工厂类才不用经常更改）。

它由三种角色组成：

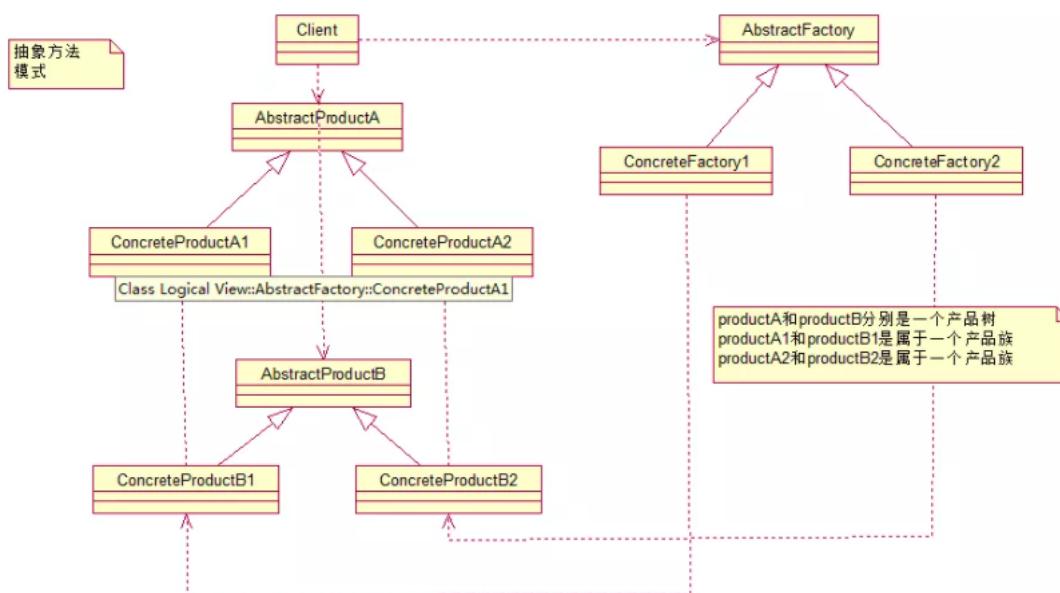
- 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑，根据逻辑不同，产生具体的工厂产品。如例子中的Driver类。
- 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。由接口或者抽象类来实现。如例中的Car接口。
- 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现，如例子中的Benz、Bmw类。

来用类图来清晰的表示它们之间的关系：



抽象工厂模式：

先来认识下什么是产品族：位于不同产品等级结构中，功能相关联的产品组成的家族。



可以说，它和工厂方法模式的区别就在于需要创建对象的复杂程度上。而且抽象工厂模式是三个里面最为抽象、最具一般性的。抽象工厂模式的用意为：给客户端提供一个接口，可以创建多个产品族中的产品对象。

而且使用抽象工厂模式还要满足一下条件：

1. 系统中有多个产品族，而系统一次只可能消费其中一族产品
2. 同属于同一个产品族的产品以其使用。来看看抽象工厂模式的各个角色（和工厂方法的如出一辙）：
 - 抽象工厂角色：这是工厂方法模式的核心，它与应用程序无关。是具体工厂角色必须实现的接口或者必须继承的父类。在java中它由抽象类或者接口来实现。
 - 具体工厂角色：它含有和具体业务逻辑有关的代码。由应用程序调用以创建对应的具体产品的对象。在java中它由具体的类来实现。
 - 抽象产品角色：它是具体产品继承的父类或者是实现的接口。在java中一般有抽象类或者接口来实现。
 - 具体产品角色：具体工厂角色所创建的对象就是此角色的实例。在java中由具体的类来实现。



1.tomcat 8 优化

1.禁用AJP连接

默认开启 AJP 服务，并且占用端口 8009

什么时AJP?

- AJP (Apache JServer Protocol) AJPv13协议是面向包的
- WEB 服务器和 servlet 容器通过 HTTP 连接来交互，为了节省 socket创建的昂贵代价，web 服务器会尝试维护一个永久TCP连接到 servlet容器，并且在多个请求和响应周期过程中重用连接。

不过因为现在一般使用 nginx + tomcat 架构，所以用不到 AJP协议。

2.使用执行池（线程池）

在 tomcat 中每一个用户都是一个线程，所以可以使用线程池提高性能

```
<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"
maxThreads="500"
minSpareThreads="50"
prestartMinSpareThreads="true"
maxQueueSize="100"/>
```

maxThreads: 最大并发数，默认设置 200，一般建议在 500 ~ 1000，根据硬件设施和业务来判断

minSpareThreads: Tomcat 初始化时创建的线程数，默认设置 25

prestartMinSpareThreads: 在 Tomcat 初始化的时候就初始化 **minSpareThreads** 的参数值，如果不等于true，**minSpareThreads** 的值就没啥效果了

maxQueueSize, 最大的等待队列数，超过则拒绝请求

```
<!--在Connector中设置executor属性指向上面的执行器-->
<Connector executor="tomcatThreadPool" port="8080" protocol="HTTP/1.1"
    connectionTimeout="20000"
    redirectPort="8443" />
```

3.调整运行模式

tomcat的运行模式有3种：

1. bio 默认的模式，性能非常低下，没有经过任何优化处理和支持。
2. nio nio(new I/O)，是Java SE 1.4及后续版本提供的一种新的I/O操作方式(即java.nio包及其子包)。
 - Java nio是一个基于缓冲区、并能提供非阻塞I/O操作的Java API，因此nio也被看成是non-blocking I/O的缩写。
 - 它拥有比传统I/O操作(bio)更好的并发运行性能。
3. apr 安装起来最困难，但是从操作系统级别来解决异步的IO问题，大幅度的提高性能。

推荐使用nio，不过，在tomcat8中有最新的nio2，速度更快，建议使用nio2.

4.调整堆的比例参数

运行时数据区哪些会Error哪些会GC

运行时数据区	是否存在Error	是否存在GC
程序计数器	否	否
虚拟机栈	是	否
本地方法栈	是	否
方法区	是 (OOM)	是
堆	是	是

String Table

改变

Jdk7 从方法区移动到堆空间。

jdk 9 String 的由 char[] 数组 更改为 byte[] 数组，节约了一些空间

```
// jdk 1.9 之前  
private final char value[];  
// jdk 1.9之后  
private final byte[] value
```

String 不可变性

- 当对字符串进行赋值时，需要重新指定内存区域赋值，不能使用原有 value 赋值
- 当现有字符串进行连接操作时，当使用String 的replace () 方法修改指定字符或字符串时，需要指定内存区域进行赋值，不能使用原有value赋值
- 通过字面量的方式（区别于new）给一个字符串赋值，此时的字符串值声明在字符串常量池中

String Pool (字符串常量池)

字符串常量池是一个固定大小的HashTable ,默认大小长度为1009 。

常量池类似一个Java系统级别的缓存 。8种基本数据结构类型的常量池都是系统协调的

- 直接使用 " " 双引号声明出来的String 对象会直接存储在常量池中

```
String str = " hello JVM"; //存储在常量池中
```

- 如果不适用双引号，可以使用String.intern()方法。

为什么String Table 移动到堆中？

因为永久代默认比较小

永久代的垃圾回收频率低

String intern 的使用

jdk1.6 , 将这个字符串对象尝试放入串池

- 如果池中存在，则并不会放入，返回已有的串池中的地址
- 如果没有，会把对象复制一份，放入串池，并返回串池中的对象地址

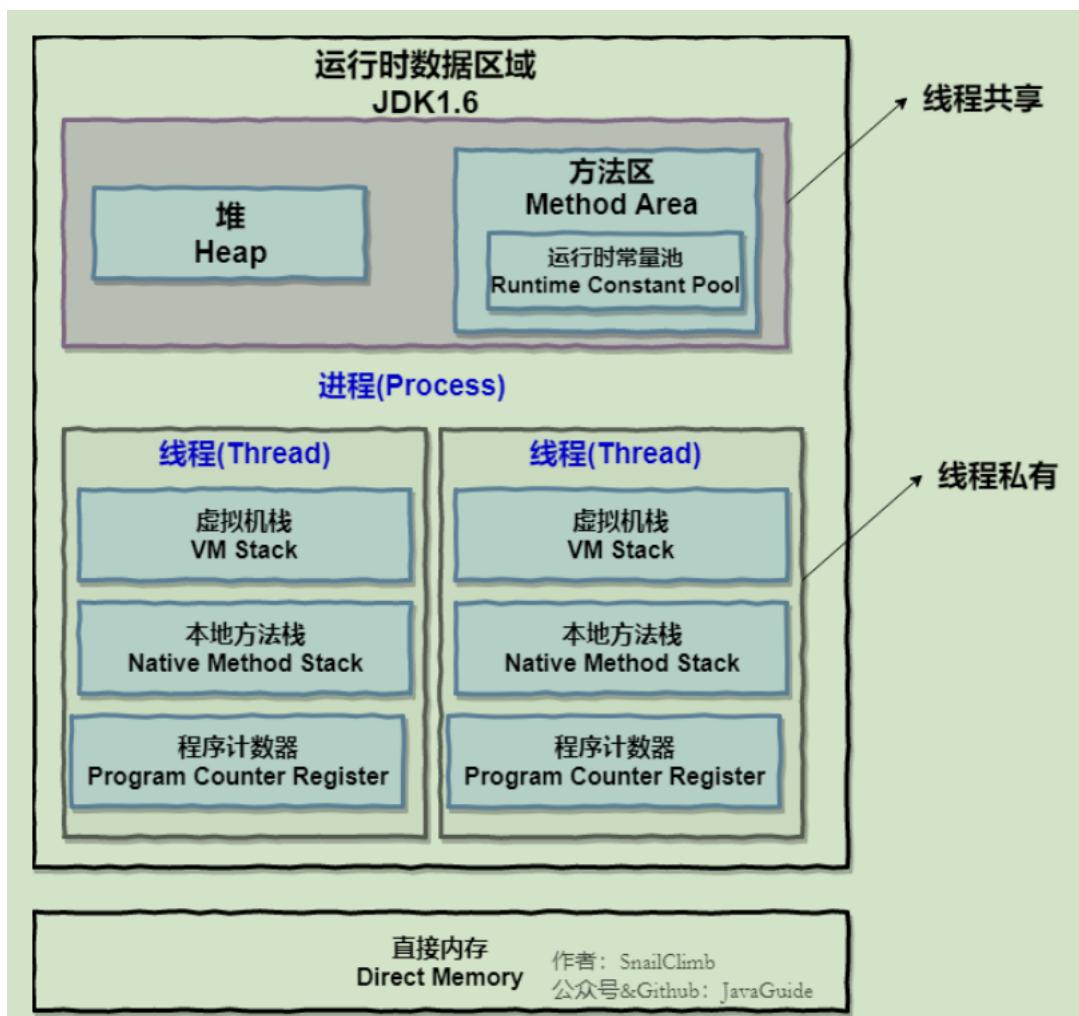
jdk1.7，将这个字符串尝试放入池中

- 如果池中存在，则并不会放入，返回已有的串池中的地址
- 如果没有，**会把对象的引用地址复制一份**，放入串池，并返回串池中的引用地址。

⚠️JDK1.8和JDK1.7JVM内存区域的区别？

- JDK1.8 和 JDK1.7 内存区域的区别是方法区不在放在堆空间中，而是放到本地内存中，解决了方法区报OOM的风险

🚩JVM内存区

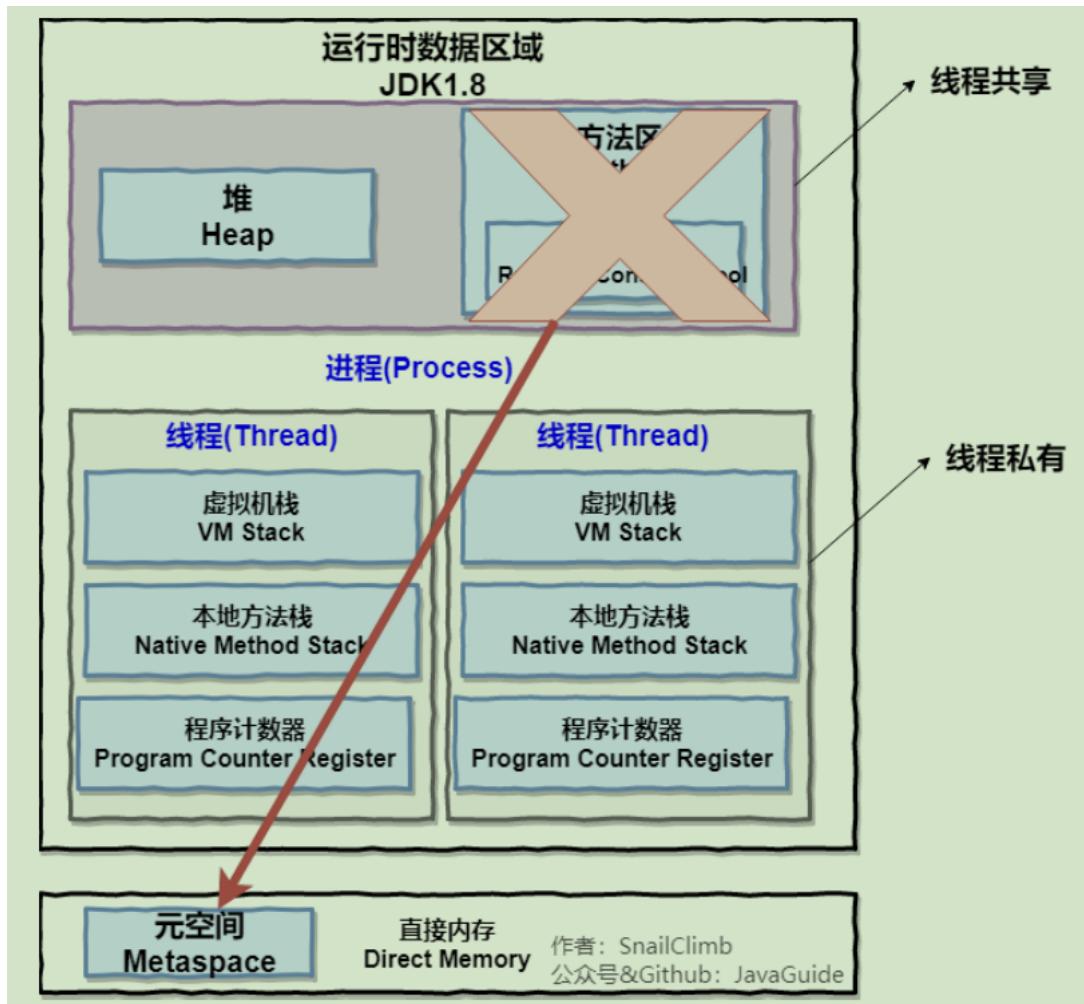


线程私有

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享

- 堆
- 方法区
- 直接内存 (非运行时数据区的一部分)



○ 程序计数器

属于线程私有的

用来存储指向下一步指令的地址，即将要执行的指令代码，由执行引擎读取下一步指令。

为什么将PC设置为线程私有的？

- 因为程序运行过程中CPU会不停地进行任务切换，这样必然会发生中断或恢复。通过PC记录各个线程正在执行的当前指令的字节码指令地址。这样就可以保证每个线程之间的独立计算，互不干扰。

○ 虚拟机栈

1. 参数

-Xss: 设置栈内存初始大小

2. 基本知识

- 线程私有，会发生StackOverflowError
- 生命周期和线程一样，线程结束了，该虚拟机栈也结束了

3. 虚拟机栈包括哪几部分？

虚拟机栈是用栈帧 (stack frame) 存储的

局部变量表、操作数栈、动态链接、方法返回地址、附加信息。

1. 局部变量表

- 定义为一个数字数组，用于存储方法参数和定义在方法体内部的局部变量。
- 这些数据类型包括：基本数据类型、对象引用和 returnAddress 类型。

2. 操作数栈

- 用于保存计算过程的中间结果，同时作为计算过程中变量临时的存储空间

3. 动态链接

1.概念

- 所有的变量和方法引用都作为符号引用保存到 class 文件的常量池里。
- 描述一个方法调用了另外其他方法时，就是通过常量池中指向方法的符号引用来表示的，
- **动态链接的作用就是为了将这些符号引用转换为调用方法的直接引用。**

2.链接

- 静态链接：被调用的方法在编译期可知，且运行保持长期不变。
- 动态链接：被调用的方法在编译期无法确定下来

3.语言类型的区别

- **静态类型语言**
 - 对类型的检查是在编译期
 - 判断变量自身的类型信息；
- **动态类型语言**
 - 对类型的检查是在运行期
 - 判断变量值的类型信息

Java: String info = "mogu blog"; (Java是静态类型语言的，会先编译就进行类型检查)

JS: var name = "shkstart"; var name = 10; (运行时才进行检查)

4. 方法返回地址

作用：存放调用方法的程序寄存器（PC）的值。

一个方法的结束，有两种退出方式：

- 正常退出：调用者的程序计数器的值作为返回地址，及调用该方法指令下一条指令的地址。
- 异常退出：返回地址通过异常表确定，栈帧中一般不会保存这些信息。

5. 一些附加信息

栈帧中还允许携带与 Java 虚拟机实现相关的一些附加信息。例如：对程序调试提供支持的信息。

4. 方法中定义局部变量是否线程安全？

- 如果只有一个线程才可以操作此数据，则线程必是安全的。
- 如果由多个线程操作，则此数据是共享数据，如果不考虑共享机制，则为线程不安全。

总的来说：如果对象在内部产生，并在内部消亡，没有返回到外部，那么它是线程安全的，反之是线程不安全的。

为什么需要运行时常量池？

因为在不同的方法里，都可能调用常量或者方法，所以只需要存储一份即可，节省了空间。

常量池的作用：为了提供一些符号和常量，便于指令的识别。



1. 基本参数

jdk1.7中包含年轻代、老年代、永久代。

jdk1.8 堆中移除永久代，永久代更名为元空间。位于本地内存中

- 年轻代与老年代的比例是 1 : 2;
- eden: from : to = 8:1:1;

survivor区的年龄到达默认值（15）就会被转移到老年代。或者年轻代过来的对象大于survivor区内存空间一半以上时，也直接送到老年代。

堆常用参数：

- -Xms：初始堆内存 **默认情况下**：物理电脑内存大小/64
- -Xmx：最大堆内存 **默认情况下**：物理电脑内存大小/4
- -Xmn：设置新生代的大小。

1. 为什么要废弃方法区？

因为在jdk1.7之前方法区在堆内存中，经常发生内存不足导致内存泄漏

2. 创建一个的对象的过程

1. new 出来的对象先放到堆空间的eden区
2. 当 eden 区填满后，此时触发 MinorGC，利用复制算法，将 eden 区存活的对象，复制到 survivor 区的 from 区，当 from 区满复制到 to 区域，将 from 区的对象进行销毁。
 - 此时将 to 区 改变成 from 区，from 区 改变成 to 区。永远让 to 区为空。回收一次给对象的年龄计数器 +1.
 - 当对象在survivor中的年龄大于默认值 15 时 移送到老年代。或者年轻代的对象大于survivor 区内存空间一半的时候，直接移送到老年代。
 - **复制之后有交换，谁空谁是to**
3. 在老年代，当内存不足时触发 MajorGC，使用标记清除或者标记整理算法。因为老年代存活对象相对来说较多，需要清除的对象较少，
4. 若老年代进行MajorGC 后依然无法保存对象就会报OOM 异常。
5. 若新创建的对象在eden 、老年代都放不下便会触发FullGC。

3. MinorGC、MajorGC、FullGC区别

MinorGC 指eden区满后，触发。会触发STW

MajorGC 发生在老年代，一次MajorGC 经常会触发一次MinorGC。

- 因为当老年代空间不足时会先触发一次MinorGC，如果空间还不足，便会触发MajorGC。MajorGC比MinorGC 慢10倍以上。如果MajorGC 后空间还不足就会报OOM。

4. FullGC触发条件一般分为以下五种情况.

- 调用System.gc();
- 老年代空间不足
- 永久代/方法区空间不足
- 通过MinorGC进入老年代的平均大小大于老年代的可用内存
- 由Eden 区转向 survivor区复制时对象大小大于 to 所占内存，把对象转移到老年代，老年代的可用内存也不够这个对象时触发 FullGC。

5.堆空间为什么要分代？

其实不分代完全可以，分代主要是为了提升GC性能，大部分对象都是朝生夕死的，把新创建的对象放到一个区域，当GC时把那些存放朝生夕死的对象的空间进行整体回收，这样就腾出来很大的空间。

6.内存分配策略

- 优先分配到Eden区
 - 将较长的字符串或者数组放到年轻代。
- 大对象直接分配到老年代
- 长期存活的对象分配到老年代
- 动态对象年龄判断
 - 如果survivor区相同年龄的所有对象大小总和大于 survivor空间的一半，无需等待是否到达指定默认年龄（15），直接进入老年代。

7.TLAB是什么？

(Thread Local Allocation Buffer) 为每个线程单独分配一个缓冲区

由于对象实例的创建在JVM非常频繁，因此在并发环境下从堆区中划分内存空间是线程不安全的，为避免多个线程操作同一地址，需要使用加锁等机制，进而影响分配速度。

8.逃逸分析

堆并非分配对象的唯一选择

堆外存储技术。

- 经过逃逸分析后发现，一个对象没有逃逸出方法的话，那么就有可能被优化成栈上分配。
- 这样就无需在堆上分配内存，也无法进行垃圾回收。

开发中尽量使用局部变量，防止逃逸分析。

方法区

1.基本参数

JDK7:

- -xx:Permsize 来设置永久代的初始化分配空间。默认为20.75M
- -XX:Maxpermisize 来设置永久代最大可分配空间。32位机器默认64M，64位机器默认82M。

JDK8

- -XX:MetaspaceSize 设置元空间大小
- 对于一个64位的服务器JVM来说其默认的-XX:MetaspaceSize 为21M。一旦触发这个水位线，将会触发FullGC，并卸载没用的类。
- 建议将-XX:MetaspaceSize 设置为一个较高的值，防止频繁GC

方法区主要存放的是Class，而堆中存放的是实例化的对象。

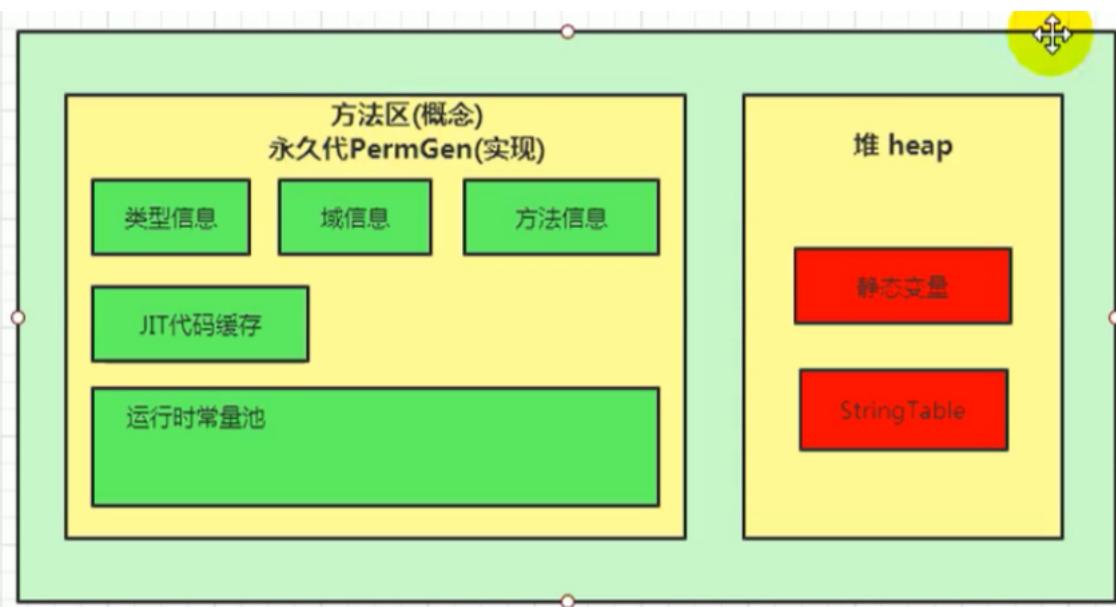
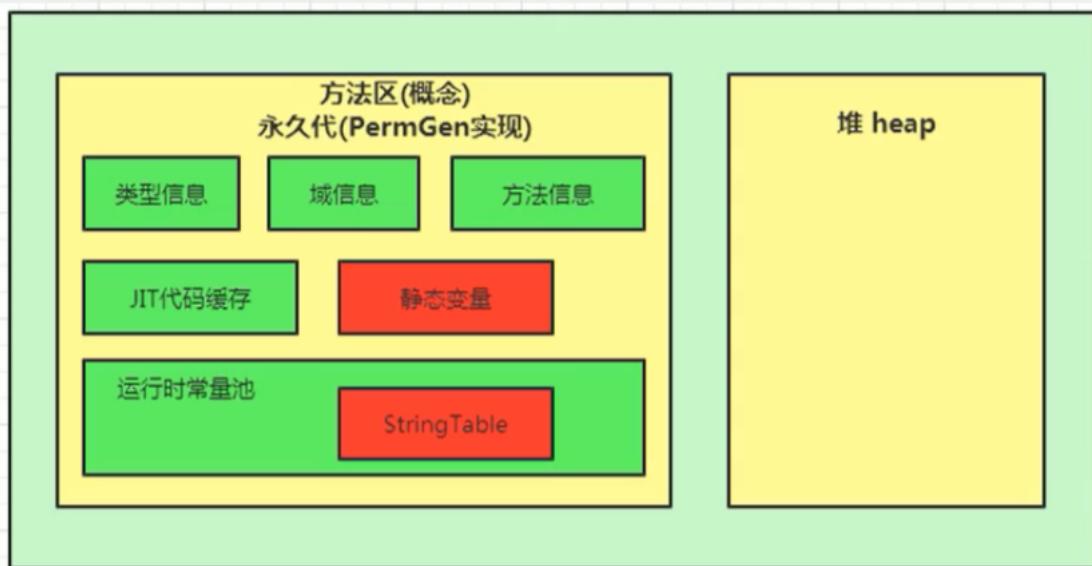
线程共享

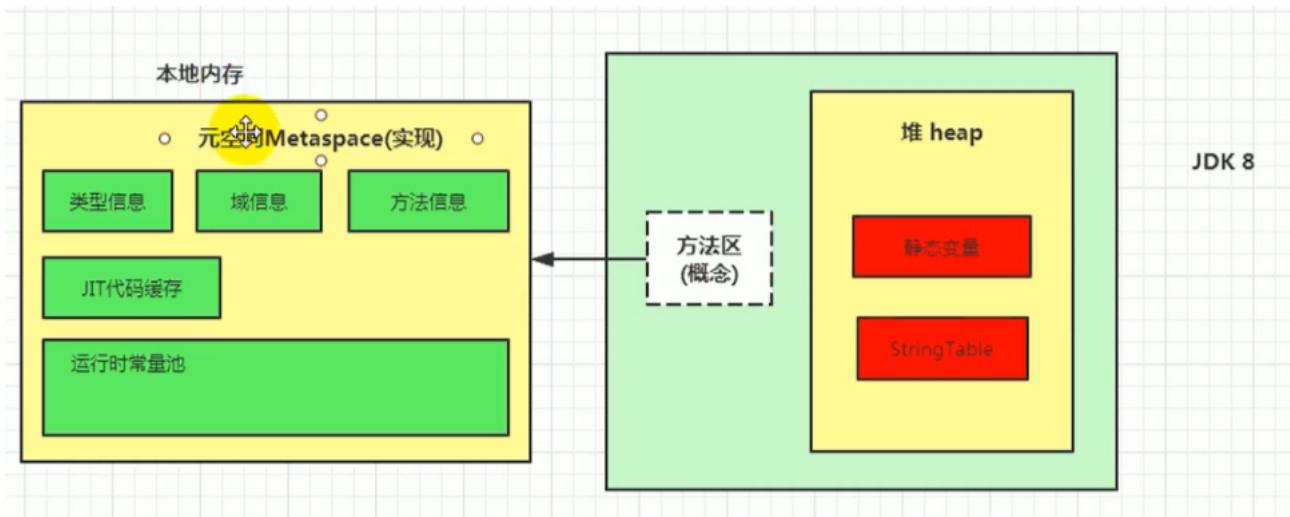
随JVM启动的时候被创建，JVM关闭被销毁

方法区满报 OutofMemoryError

方法区的演进细节

JDK1.6及以前	有永久代，静态变量存储在永久代上
JDK1.7	有永久代，但已经逐步“去永久代”，字符串常量池，静态变量移除，保存在堆中
JDK1.8	无永久代，类型信息，字段，方法，常量保存在本地内存的元空间，但字符串常量池、静态变量仍然在堆中。





运行时常量池

运行时常量池是方法区/元空间的一部分，

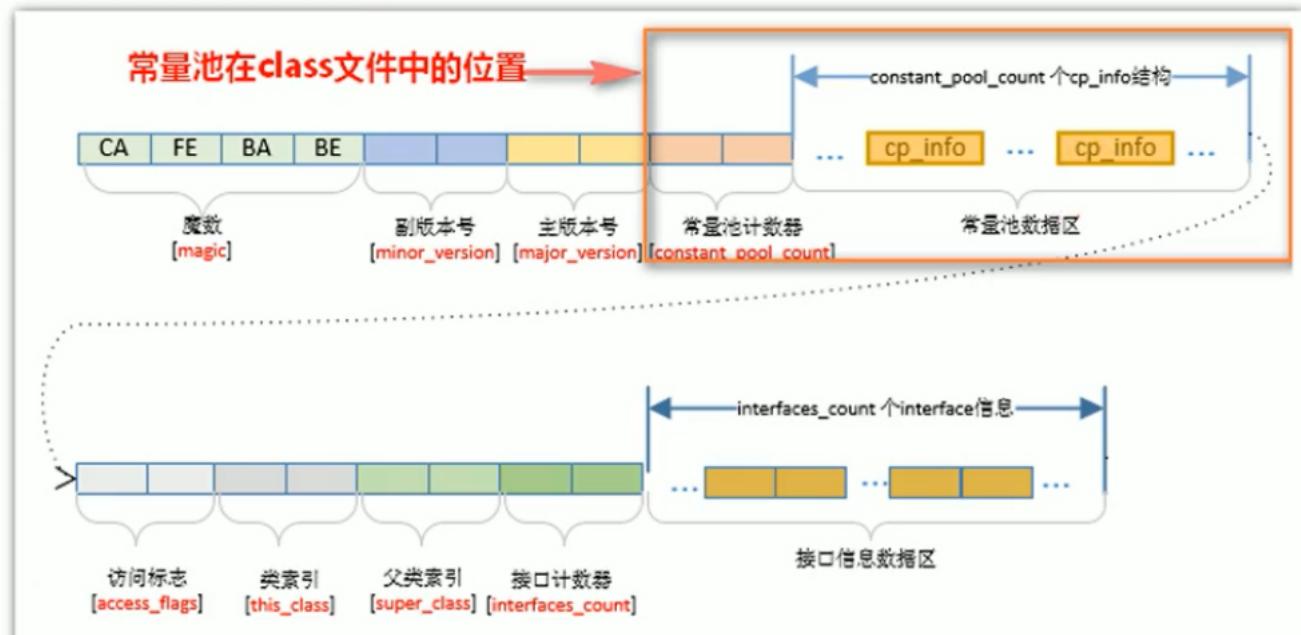
常量池表是 class 文件的一部分，用于存放编译期生成的各种字面量和符号引用，这部分内容在类加载后存放到方法区的运行时常量池中。

JVM 为每个已加载的类型（类或接口）都维护一个常量池。池中的数据如同数组，通过索引访问。

运行时常量池包含多种不同的常量，包括编译期明确的**数值字面量**，也包括运行期解析后才能获得的**方法或者字段引用**，此时不再是常量池的**符号地址**了，这里换成的**真实地址**。

运行池常量池相对于 class 常量池的另一个重要特征是**具备动态性**。

常量池



常量池可以看作是一张表，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、字面量等类型。

一个字节码文件除了包含版本信息、字段、方法、以及接口等描述信息外，还包含一项常量池表：**包括字面量和对类型、域、方法的符号引用**。

常量池中包含什么？

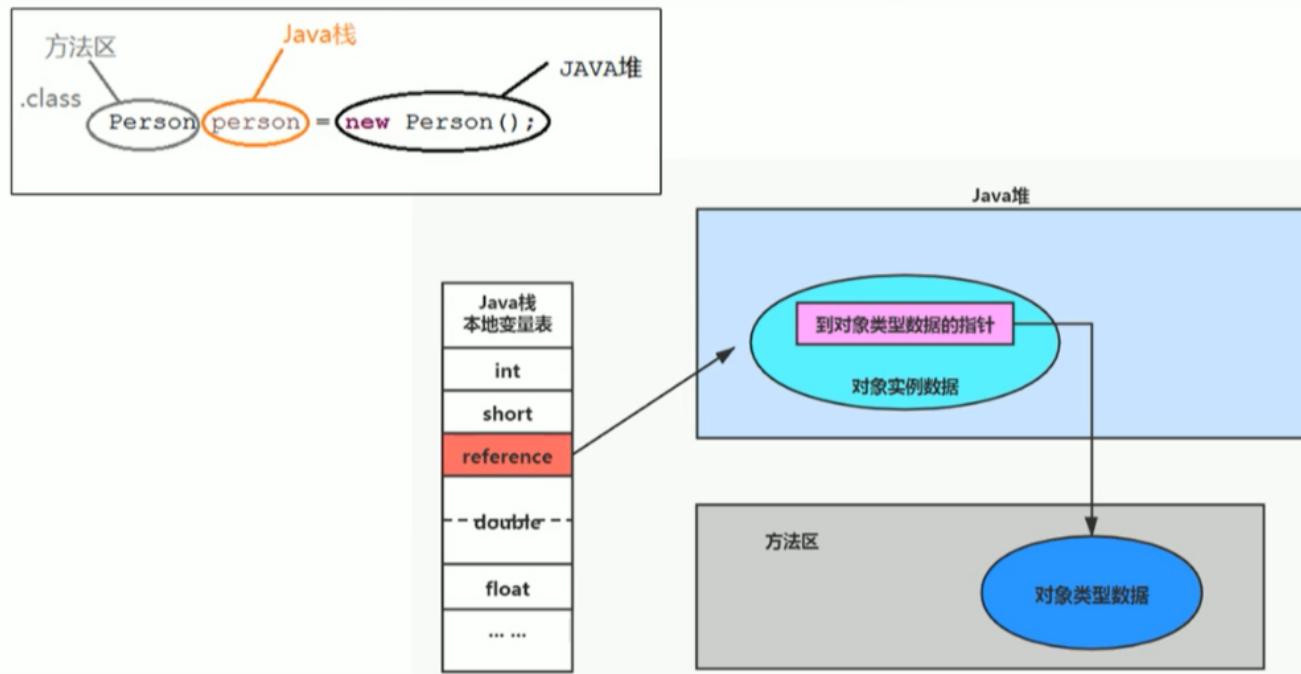
- 数量值、字符串值、类引用、字段引用、方法引用

为什么需要常量池?

- 一个Java文件经过编译后生成.class字节码文件，而java中的字节码需要数据支持，通常这种数据会很大以至于不能直接存放到字节码文件里，换另一种方式存到常量池，这个字节码包含了指向常量池的引用。
- 在动态链接的时候会用到常量池的引用。

堆、栈、方法区的关系

下面就涉及了对象的访问定位



- `Person`: 存放在元空间，也可以说方法区
- `person`: 存放在Java栈的局部变量表中
- `new Person()`: 存放在Java堆中

为什么永久代要被元空间替代?

- 因为之前永久代存放在堆中，很容易出现内存溢出。
- 元空间把数据分配到本地内存中，仅受本地内存大小限制。
- 对永久代调优很困难

方法区的垃圾回收?

方法区的垃圾收集主要分为两部分：

常量池中废弃的常量

- 只要常量池中常量没有被任何地方引用，就可以回收。

不在使用的类型。满足以下三种条件允许被回收，

- 该类所有的实例都已被回收，也就是java堆中不存在该类及任何派生子类的实例
- 加载该类的类加载器已经被回收
- 该类对应的`java.lang.Class`对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

在大量使用反射、动态代理、CGLib等字节码框架，动态生成SPI以及oSGi这类频繁自定义类加载器的场景中，通常都需要Java虚拟机具备类型卸载的能力，以保证不会对方法区造成过大的内存压力。

StringTable为什么要调整位置？

- 因为永久代的回收效率很低，在FullGC时才会触发。而FullGC在老年代空间不足和永久代空间不足才会触发。
- 这样就导致StringTable回收效率不高，而我们开发中有大量的字符串被创建、回收效率低，导致永久代内存不足。**放到堆里能及时回收内存。**

静态变量存放在哪里？

从JDK7起，静态变量存放在堆中。

直接内存（非运行时数据区）

线程共享

本地方法栈

线程私有，用C语言实现 native 方法。

说一下JVM内存模型？

分为方法区、堆、程序计数器、虚拟机栈、本地方法栈。其中方法区、堆、是线程共享的。程序计数器、虚拟机栈、本地方法栈是线程私有的。

方法区

- 在jdk8称为元空间，存放在内存中，不受堆内存大小限制。
- 方法区很少发生GC，会对常量池和类型信息进行回收。方法区用来存储类型信息、域信息、方法信息、JIT代码缓存、运行时常量池。
- 方法区的运行时常量池用来存放静态编译产生的字面量和符号引用

堆

- 新创建的对象都会放在堆中，在jdk8中，堆分为年轻代和老年代，移除永久代。
- 年轻代分为eden和2个survivor区，年轻代的比例是8：1：1。
- 年轻代与老年代的比例是1：2
- 垃圾回收主要发生在堆中

程序计数器

- 程序计数器用来存放正在执行的Java方法的JVM指令地址。
- 程序计数器是内存区域唯一不会发生OOM的内存区域。

虚拟机栈

- 它的生命周期和线程相同。它有多个栈帧组成，每个栈帧包含一个指向方法区中运行时常量池的符号引用。
- 每个栈帧中存储了局部变量表、操作数栈、方法返回地址、附加信息、动态链接。
- 动态链接的作用是将符号引用转换为直接引用。
- 操作数栈的作用是用来存储计算的中间结果
- 局部变量表用来存储方法参数和定义在方法内的局部变量。

本地方法栈

- 本地方法栈和虚拟机栈相似，只不过它的实现是native方法。

描述一下类加载过程？

类加载过程分为加载、链接、准备、解析、初始化、使用、卸载

类加载：通过一个类的全限定类名获取类的二进制流，将这个字节流所代表的静态存储结构转化为方法区的运行时数据区。在内存中生成一个代表这个类的java.lang.Class 对象，作为方法区这个类的访问入口。

链接: 验证class文件是否符合虚拟机要求，保证被加载类的正确性。

准备: 为类变量分配内存，并且设置该类变量的默认初始值

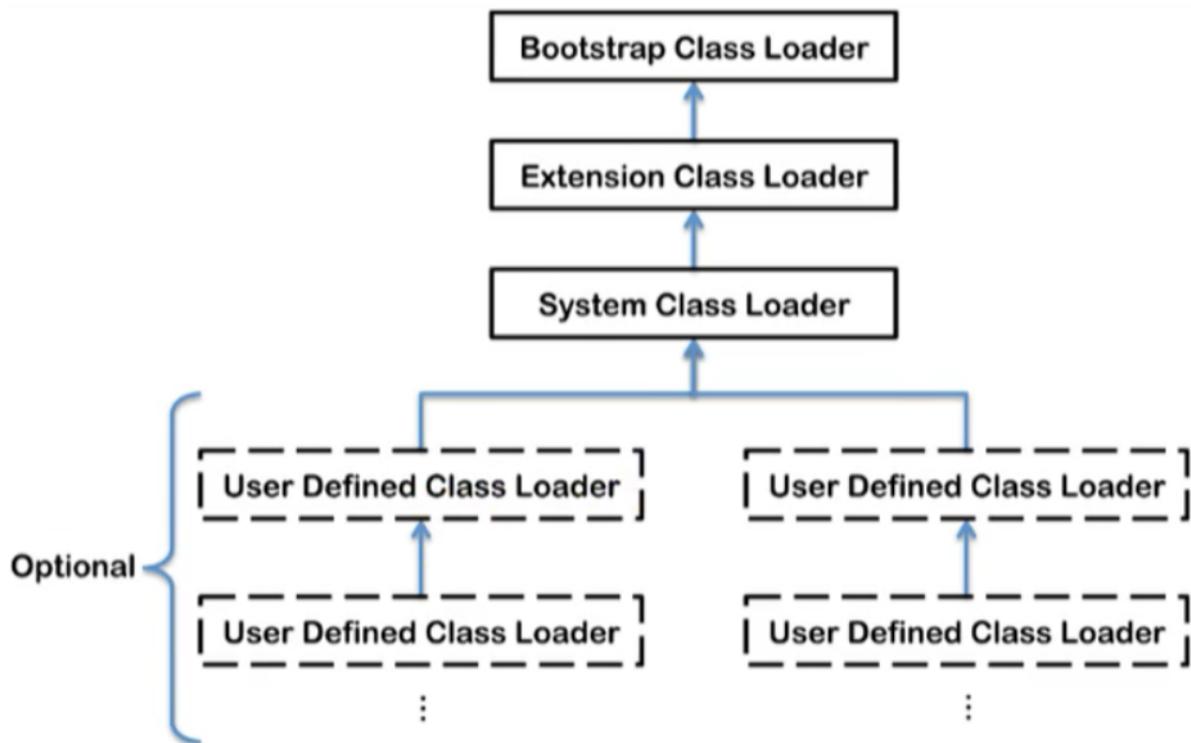
解析: 将常量池中符号引用转化为直接引用（直接指向目标的指针、相对偏移量、或者一个间接定位到目标的句柄）。

初始化: 给类变量显示赋值，给静态代码块赋值。

简述类加载机制？

JVM把描述类的数据从 class文件加载到内存，并对其进行数据检验，解析，初始化，最终可以形成虚拟机直接使用的 java 类型

描述一下类加载器？



启动类加载器、扩展类加载器、应用类加载器、用户类自定义加载器

启动类加载器

- 使用C\C++语言实现的，嵌套在JVM内部，无法被java程序直接引用

扩展类加载器

- 父类加载器是启动类加载器
- 由java实现，用于加载java扩展库
- 用户创建的jar放在此目录下，也会由扩展类加载器加载
- 从java.ext.dirs系统属性所指定的目录加载类库，或从jdk安装目录jre/lib/ext子目录下加载类库

应用类加载器：

- 父类加载器为扩展类加载器
- 他负责加载环境变量ClassPath或系统属性java.class.path指定路径下的类库。
- 是程序中默认的类加载器，一般的java类都是由它来加载的

用户类自定义加载器: 通过继承java.lang.ClassLoader类的方式实现。

如何实现自定义加载器？

有时我们想要**实现自己的类加载器**，但是不想打破双亲委派机制，那么我们可以

1. 继承自ClassLoader
2. 重写 findclass 方法，在findclass 里获取类的字节码，并调用ClassLoader中的defineClass方法加载类，获取class 对象。

有时我们想要**实现自己的类加载器**，打破双亲委派机制，那么我们可以

- 重写loadClass 方法

JVM的永久代中会发生垃圾回收么？

永久代不会发生垃圾回收，当永久代满时会发生FullGC

什么是双亲委派机制？

当一个类收到类加载请求时，它会判断父类是否可以被加载，依次向上委派。如果父类加载不了，则让其子类加载。

聊聊垃圾回收算法？

标记-清除算法

- 标记清除算法分为标记和清除，标记阶段从根节点开始遍历，标记所用被引用的对象，
- 清除阶段会把那些没有标记的节点进行清除
- 标记清除算法的效率不高，会产生碎片

标记-复制算法

- 将内存区域分成两块，每次只使用一块，在GC时将存活的对象复制到另一个区域，清除存在垃圾区域的全部空间，交换两个空间的位置，重复进行此操作
- 解决了碎片的产生，内存空间只可以使用一半，适合新生代垃圾收集。
- 对于G1这种存在大量 Region 的GC，复制而不是移动。意味着GC需要维护region之间对象的引用关系，不管是内存占用还是时间开销都不小。

标记-整理算法

- 标记整理算法分为两部分，
- 第一阶段进行对象的可达标记，第二阶段将所有存活的对象压缩到内存的一端，之后清理边界外的所有空间。
- 适合于老年代，存活对象时间长，回收频率低。
- 缺点是会造成STW，效率低于标记复制算法。

引用计数法和可达性分析算法？

引用计数算法：

- 为每个对象设置一个引用计数器，对于对象A，任何对象引用了A，A的引用计数器 + 1，只要引用计数器减为0，即可被垃圾回收。
- 优点实现简单，
- 缺点会造成循环引用。

可达性分析算法

- 以根节点为起点，从上到下进行搜索，判断对象是否可达，把可达的对象标记起来。
- 回收时，将不在引用链上的对象进行回收

垃圾回收器

serial 回收器

- 串行回收，采用复制算法，适用于年轻代，会造成STW，client 端的默认新生代垃圾收集器

serial old 回收器：串行回收，会造成STW，使用标记压缩算法，作为老年代CMS 的后备垃圾收集方案

parNew 回收器

- 采用并行方式回收垃圾，采用复制算法。
- 是server 端下面默认的新生代垃圾回收器。
- 目前和CMS搭配使用

对于新生代，回收频率频繁，使用并行可以提高回收效率

对于老年代，回收频率不频繁，使用串行比较高效（节省了切换线程所消耗的资源）

parallel Scavenge回收器

在java8 默认是此垃圾收集器

- 采用复制算法，注重吞吐量优先，具有自适应调节功能，适用于新生代
- 年轻代的大小、Eden、survivor 区的比例、晋升老年代的对象年龄等参数会自动调整，为了达到堆大小、吞吐量、停顿时间的平衡点

parallel Old 在java8 默认是此垃圾收集器

- 老年代版本，采用标记压缩算法

CMS回收器

参数

使用率阈值：jdk 5 之前默认值为68 即老年代为92%

CMS (concurrent mark sweep)

优点：并发收集、低延迟

缺点：产生碎片、无法处理浮动垃圾

- 是一款真正意义上的并发收集器，让垃圾收集线程与用户线程同时工作。
- 采用标记-清除算法，CMS 关注点是尽可能缩短垃圾收集时用户线程停顿的时间
- 作为老年代垃圾收集器只能与ParNew 和 serial 配合实用

CMS 垃圾收集分为4个阶段

初始标记 (STW) : 标记出GCRoots 能直接关联到的对象，由于直接关联对象比较小，这里的速度非常快。

并发标记：从GCRoots 直接关联的对象遍历整个对象图的过程，这个过程耗时长，但是不需要占用用户线程，和垃圾收集器线程一起运行。

重新标记 (STW) : 由于并发标记程序一直运行，会产生新的对象，因为修正并发标记期间产生变动的那一部分对象的变更记录，这个阶段比初始标记时间长，比并发标记时间短。

并发清除：清理掉标记阶段已死亡的对象。

CMS为什么无法处理浮动垃圾？

因为在并法标记过程中，程序的工作线程和垃圾收集器是同时运行或者交叉运行的，在并发标记过程中，如果产生新的垃圾对象，CMS将无法对这些垃圾对象进行标记，导致垃圾不能得到及时的回收，只能等到下一次GC

为什么CMS不在堆满时进行垃圾回收？

CMS并不会等到堆空间满时触发垃圾回收，因为在并行标记过程中，程序是一直运行的，此时并发标记也需要内存空间，当在垃圾回收时内存空间不足时会报 Concurrent Mode Failure。此时会启动后备方案，使用serial old收集器代替老年代收集，这样停顿时间会大大增加

CMS为什么不采用标记整理算法？

因为标记整理算法会占用用户线程。

G1 垃圾收集器

特点

- G1 (garbage first) **垃圾优先**
- 在**延迟可控**的情况下获得**尽可能高的吞吐量**

G1垃圾回收器的优势

1.并发与并行

- 并行性：G1回收期间，可以多个GC线程同时工作，有效利用多线程计算能力，此时用户线程STW
- 并发性：G1拥有与用户线程交替执行的能力，部分工作可以和线程同时执行。因此一般来说，不会在整个回收阶段完全阻塞应用程序的情况。

2.分代收集

- 从分代看：G1依然属于分代型垃圾收集器，分为年轻代、老年代、年轻代依然存在Eden和两个 survivor区。
- 从堆看：他不要求整个Eden区、survivor区是连续的，也不设置固定大小和固定数量。
- 将堆分为若干个空间这些空间包含逻辑上的年轻代和老年代
- G1垃圾收集器同时兼顾年轻代和老年代

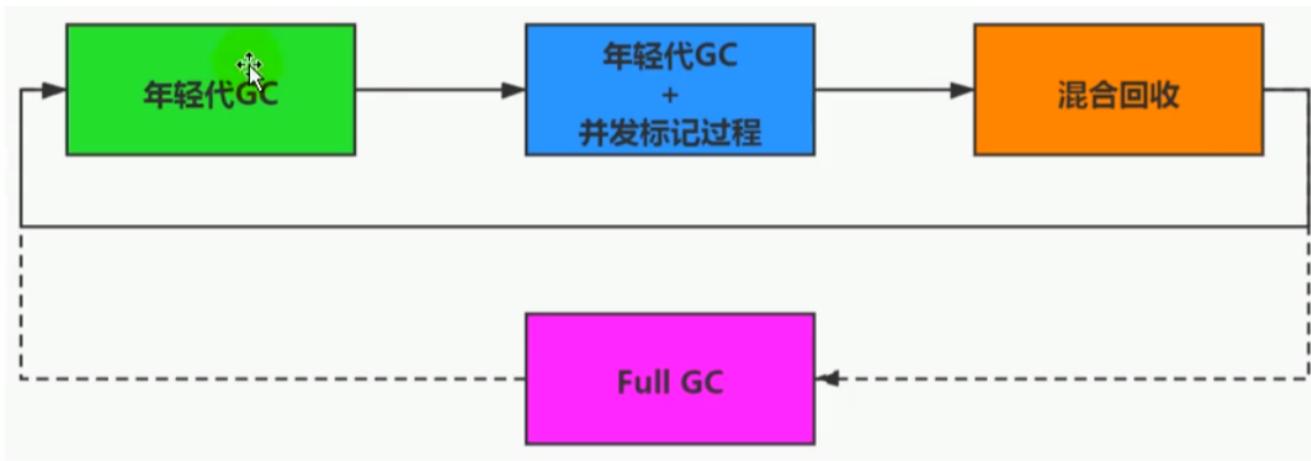
3.可预测的停顿时间模型

- G1除了追求低停顿外，还能建立可预测的停顿时间模型，这能让使用者明确指定一个长度为M 毫秒的时间片段内，消耗在垃圾收集时间上不超过 N 毫秒。
- 由于分区的原因G1 可以选择部分区域进行回收，这样缩小了回收范围，因此对于全局停顿情况的发生具有较好的控制
- **G1追踪各个Region里面的垃圾堆积的大小**（回收所获得空间大小以及回收所需的经验值），在后台维护一个优先列表，每次根据允许的收集时间，**优先回收价值最大的Region**。

G1垃圾收集器的缺点

- 在程序的运行过程中，G1垃圾收集器的内存占用和程序运行时额外负载都要比CMS高

G1垃圾回收过程



顺时针，young gc -> young gc + concurrent mark-> Mixed GC顺序，进行垃圾回收。

Young GC ==> YoungGC + 并发标记过程 ==> 混合回收 ==> FullGC

G1和CMS的区别？

CMS基于标记-清除算法，会产生内存碎片、若干次GC后进行依次碎片整理

G1将内存划分为一个个的Region，内存的回收是以Region为单位，Region之间是复制算法。避免产生内存碎片。有助于程序长时间运行。分配大对象不会因无法找到连续合适的存储空间而触发下一次GC，尤其是当分配堆特别大时，G1的优势特别明显。

G1适合大内存应用场景、CMS适合小内存应用

垃圾回收器总结

垃圾收集器	分类	作用位置	使用算法	特点	适用场景
Serial	串行运行	作用于新生代	复制算法	响应速度优先	适用于单CPU环境下的client模式
ParNew	并行运行	作用于新生代	复制算法	响应速度优先	多CPU环境Server模式下与CMS配合使用
Parallel	并行运行	作用于新生代	复制算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
Serial Old	串行运行	作用于老年代	标记-压缩算法	响应速度优先	适用于单CPU环境下的Client模式
Parallel Old	并行运行	作用于老年代	标记-压缩算法	吞吐量优先	适用于后台运算而不需要太多交互的场景
CMS	并发运行	作用于老年代	标记-清除算法	响应速度优先	适用于互联网或B/S业务
G1	并发、并行运行	作用于新生代、老年代	标记-压缩算法、复制算法	响应速度优先	面向服务端应用

最小化实用内存和并行开销：Serial GC

最大化的应用吞吐量：Parreal Scavenge GC

最小化GC的中断和停顿时间 CMS GC

☆java并发☆

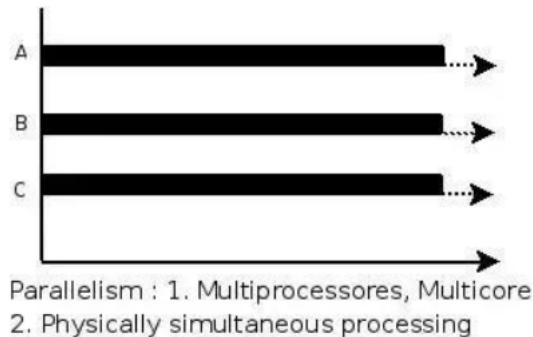
⌚ 基础

1. 并行和并发有什么区别？

并行

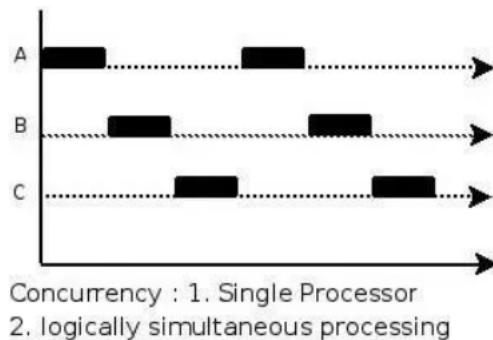
- 是指两个或者多个事件在 **同一时刻** 发生；
- 并行的多个任务之间是不互相抢占资源的、

- 其实决定并行的因素不是CPU的数量，而是CPU的核心数量，比如一个CPU多个核也可以并行。



并发

- 是指两个或多个事件在 **同一时间间隔** 发生。
- 并发的多个任务之间是互相抢占资源的。
- 在操作系统中，是指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行。**



只有在多CPU或者一个CPU多核的情况下，才会发生并行。否则，看似同时发生的事情，其实都是并发执行的。

2. 线程和进程的区别？

简而言之，进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切换次数，从而效率更高。

线程是进程的一个实体，是cpu调度和分派的基本单位，是比程序更小的能独立运行的基本单位。

同一进程中的多个线程之间可以并发执行。

3. 守护线程是什么？

守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程。

创建线程有哪几种方式？

```
public static void main(String[] args) throws ExecutionException, InterruptedException {
    FutureTask<String> task = new FutureTask<>(new MyCallable());
    new Thread(task).start();
    System.out.println(task.get());

    Thread myRunnable = new Thread(new MyRunnable());
    myRunnable.start();

    Thread myThread = new Thread(new MyThread());
    myThread.start();
}
```

- 继承 Thread 类并重写 run 方法。实现简单，但不符合里氏替换原则，不可以继承其他类。

```
/**
 * 继承 Thread 方法，重写 run() 方法
 */
static class MyThread extends Thread{
    @Override
    public void run() {
        System.out.println("呦西，好吃的不得了");
    }
}
```

- 实现 Runnable 接口并重写 run 方法。避免了单继承局限性，编程更加灵活，实现解耦。

```
/**
 * 实现Runnable 接口，重写 run() 方法
 */
static class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("太君土豆哪里去挖？");
    }
}
```

- 实现 Callable 接口并重写 call 方法。可以获取线程执行结果的返回值，并且可以抛出异常

```
/**
 * 实现Callable() 方法，并且有返回值。
 * 需要使用FutureTask<V> 在外部封装一下获取返回值
 */
static class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        return "我是小鬼子！";
    }
}
```

① 继承Thread类创建线程类

- 定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。
- 创建Thread子类的实例，即创建了线程对象。
- 调用线程对象的start()方法来启动该线程。

② 通过Runnable接口创建线程类

- 定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
- 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
- 调用线程对象的start()方法来启动该线程。

③ 通过Callable和Future创建线程

- 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
- 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
- 使用FutureTask对象作为Thread对象的target创建并启动新线程。
- 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值。

什么是死锁？

- 死锁指的是两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞现象。若无外力的作用，他们将无法推进下去

是操作系统层面的一个错误，是**进程死锁的简称**，最早在 1965 年由 Dijkstra 在研究银行家算法时提出的，它是计算机操作系统乃至整个并发程序设计领域最难处理的问题之一。

怎么防止死锁？

死锁的四个必要条件：

- 互斥条件：**
 - 进程对所分配到的资源**不允许其他进程进行访问**，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源
- 请求和保持条件：**
 - 进程获得一定的资源之后，又对其他资源发出请求，但是该资源可能被其他进程占有，此事请求阻塞，但又对自己获得的资源保持不放
- 不可剥夺条件：**
 - 是指进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用完后自己释放
- 环路等待条件：**
 - 是指进程发生死锁后，若干进程之间形成一种头尾相接的循环等待资源关系

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而**只要上述条件之一不满足，就不会发生死锁**。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。

所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。

此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

说一下 runnable 和 callable 有什么区别？

相同点

- 都是接口，都采用Thread.start()启动，都可以编写多线程

不同点

- Runnable接口中的run()方法无返回值。Callable接口中的call()方法有返回值，是个泛型和future、FutureTask配合可以获取异步执行的结果。
 - Runnable接口只抛出运行时异常，且无法捕获处理。Callable接口call()方法允取抛出异常信息，可以捕获异常
- callable接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主线程。

线程的run()和start()方法有什么区别？

- start()方法用于启动线程，真正的实现了多线程运行，调用start()无需等待run()方法体执行完毕，可以继续执行其他代码。此时线程处于就绪状态，并没有运行。然后通过调用run()方法来完成其运行状态，run()方法运行结束，线程终止。
- start()只能调用一次，而run()可以调用多次。
- 如果直接调用run()那么就是调用一个普通函数而已。

为什么要调用start() 而不是run()?

new一个thread线程进入新建状态。调用start()方法，会启动一个线程并使线程进入就绪状态，当分配到时间片后就可以运行了。start()会执行线程的相应准备工作，然后自动执行run()方法中的内容，这是真正的多线程工作。

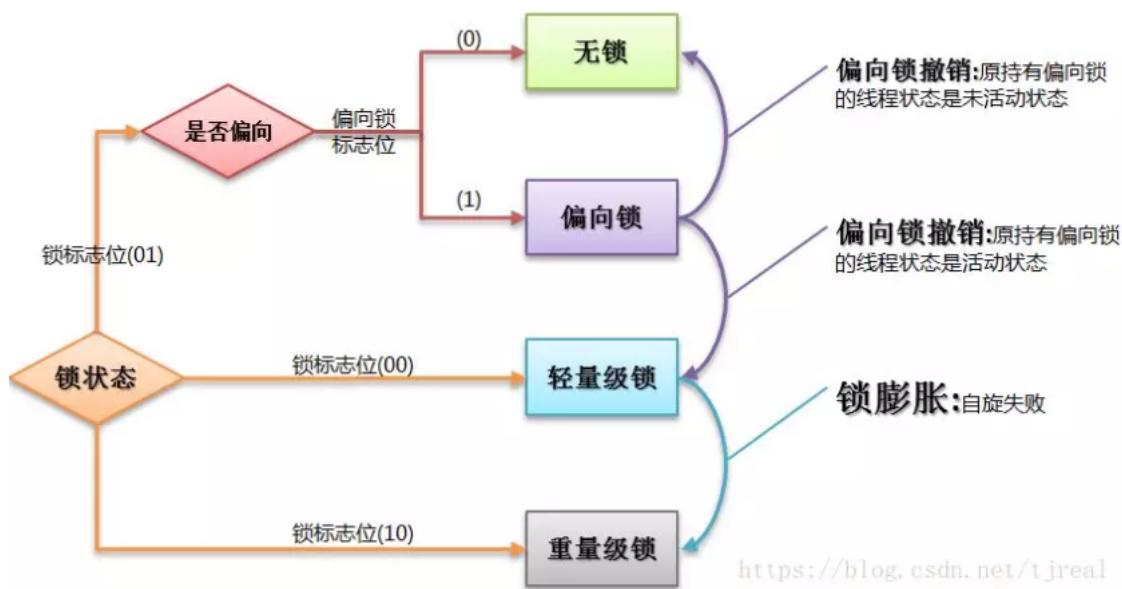
而直接执行run()方法，run方法只不过是一个普通的函数而已，并不会在线程中去执行它。

调用start()方法会让线程进入就绪状态，而run()方法只不过是一个普通的函数而已

多线程锁的升级原理是什么？

在Java中，锁共有4种状态，级别从低到高依次为：无状态锁，偏向锁，轻量级锁和重量级锁状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级。

锁升级的图示过程：



为什么代码会重排序？

在执行程序时，为提高性能，处理器和编译器常常会对指令重排序。满足以下两种条件

- 在单线程环境下不能改变程序运行的结果
- 存在依赖关系的不运行重排序

as-if-serial规则和happens-before规则的区别

- as-if-serial语义保证单线程内程序的执行结果不被改变，
- happens-before保证正确同步的多线程程序的执行结果不受改变。

都是为了不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

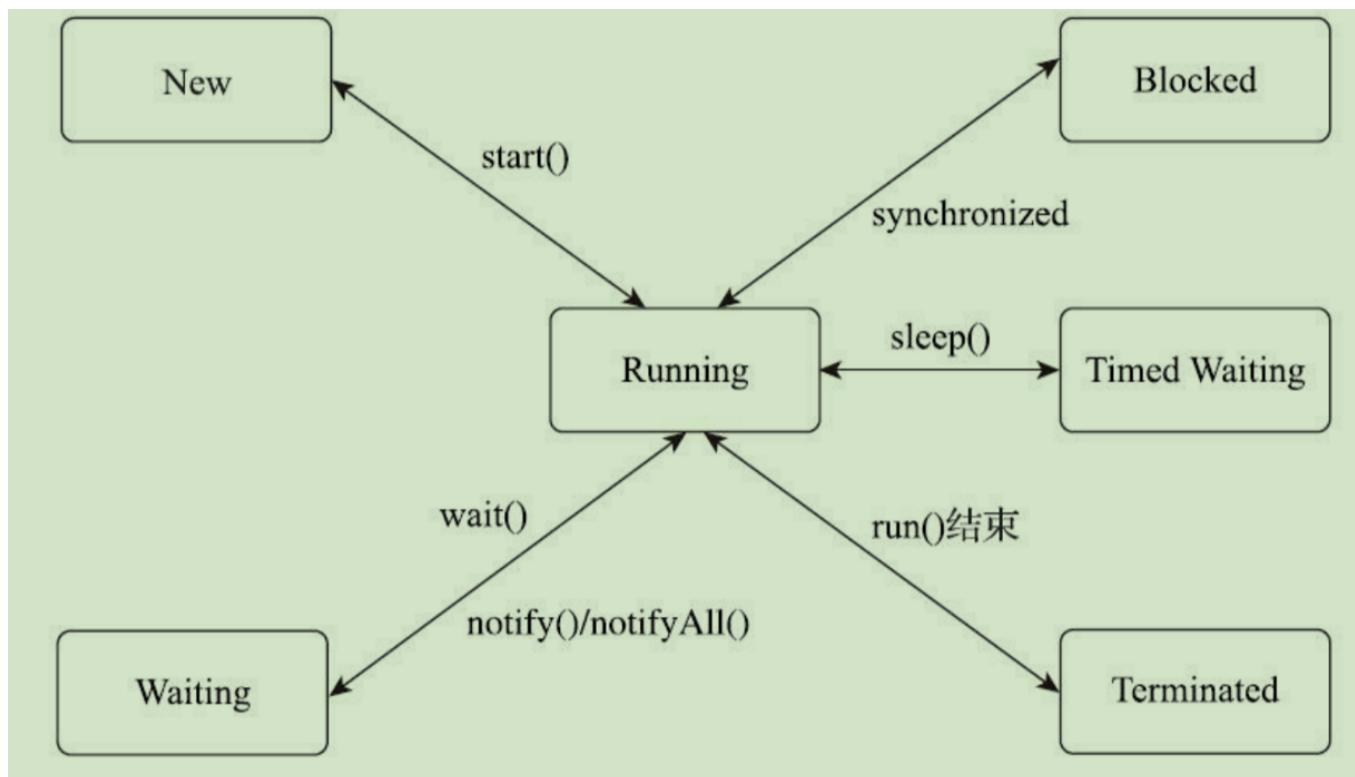
线程的基本状态

线程通常都有五种状态，创建、可运行、运行、阻塞和死亡。

1. **新建(new)**：新创建了一个线程对象。
2. **可运行(runnable)**：线程对象创建后，当调用线程对象的 start() 方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。

3. **运行(running)**: 可运行状态(runnable)的线程获得了cpu时间片 (timeslice) , 执行程序代码。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；
 4. **阻塞(block)**: 处于运行状态中的线程由于某种原因，暂时放弃对 CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被 CPU 调用以进入到运行状态。
- 阻塞的情况分三种：** (一). 等待阻塞：运行状态中的线程执行 **wait()**方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态； (二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态； (三). 其他阻塞: 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。
5. **死亡(dead)**: 线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

7.状态之间的关系



Java 中用到的线程调度算法是什么？

计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得CPU 的使用权才能执行指令。所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在运行池中，会有多个处于就绪状态的线程在等待 CPU，JAVA 虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配 CPU 的使用权。

有两种调度模型：**分时调度模型和抢占式调度模型**。

分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

线程的调度策略

线程调度器选择优先级最高的线程运行，但是，如果发生以下情况，就会终止线程的运行：

- (1) 线程体中调用了 yield 方法让出了对 cpu 的占用权利

- (2) 线程体中调用了 sleep 方法使线程进入睡眠状态
- (3) 线程由于 IO 操作受到阻塞
- (4) 另外一个更高优先级线程出现
- (5) 在支持时间片的系统中，该线程的时间片用完

什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。

时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。

线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

请说出与线程同步以及线程调度相关的方法。

1. wait() 使一个线程处于阻塞（等待）状态，并且释放所持有的对象的锁。
2. sleep() 使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException 异常
3. notify() 唤醒一个处于等待的线程，并不会指定唤醒哪个线程，而是由JVM确定唤醒哪个线程
4. **notifyAll() 唤醒所有等待的线程，让他们竞争，只有获得锁的线程进入等待状态。**

11.启动一个线程是用run()还是start()？

- 启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM调度并执行。这并不意味着线程就会立即运行。
- run()方法可以产生必须退出的标志来停止一个线程。

10.线程的 run() 和 start() 有什么区别？

每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，方法run()称为线程体。通过调用Thread类的start()方法来启动一个线程。

start()

- 方法来启动一个线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；这时此线程是处于就绪状态，并没有运行。然后通过此Thread类调用方法run()来完成其运行状态，这里方法run()称为线程体，它包含了要执行的这个线程的内容，Run方法运行结束，此线程终止。然后CPU再调度其它线程。

run()

- 方法是在本线程里的，只是线程里的一个函数，而不是多线程的。如果直接调用run(),其实就相当于是调用了一个普通函数而已，直接调用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

8.sleep() 和 wait() 有什么区别？

sleep():

- 不释放锁，是Thread 的静态方法
- 用于暂停线程执行。
- sleep() 执行完后会立即苏醒。
- 不考虑线程的优先级

wait():

- 释放锁，是Object类的方法
- 通常用于线程间通信。
- wait() 调用后，线程不会自动苏醒，而是需要调用notify或者notifyAll方法

9.notify() 和 notifyAll() 有什么区别？

- 如果线程调用了对象的 wait() 方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。
- 当有线程调用了对象的 **notifyAll()方法（唤醒所有 wait 线程）** 或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的线程便会进入该对象的锁池中，**锁池中的线程会去竞争该对象锁**。也就是说，调用了 notify 后只要一个线程会由等待池进入锁池，而 notifyAll 会将该对象等待池内的所有线程移动到锁池中，等待锁竞争。
- 优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，**唯有线程再次调用 wait()方法，它才会重新回到等待池中**。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

你是如何调用 wait() 方法的？使用 if 块还是循环？为什么？

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

```
synchronized (monitor) {
    // 判断条件谓词是否得到满足
    while(!locked) {
        // 等待唤醒
        monitor.wait();
    }
    // 处理其他的业务逻辑
}
```

为什么线程通信的方法 wait(), notify() 和 notifyAll() 被定义在 Object 类里？

java 中任何对象都可以作为锁，并且 wait(), notify() 等方法用于等待对象的锁和唤醒对象的锁线程，在 java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定在 Object 中。

为什么 wait(), notify() 和 notifyAll() 这些方法必须在同步代码块中调用？

当一个线程需要调用对象的 wait() 或者 notify() 方法的时候，**这个线程必须拥有该对象的锁**，接着它就会**释放这个对象的锁并进入等待状态** 直到其他线程调用这个对象的 **notify()** 方法。

Thread 类中的 yield 方法有什么作用？

使当前线程从运行状态转为就绪状态。

线程的 sleep() 方法和 yield() 方法有什么区别？

- sleep 不考虑优先级。yield 方法考虑优先级
- sleep 执行进入阻塞状态，yield 执行后进入就绪状态
- sleep 抛出 InterruptedException，yield 不会抛出异常
- sleep 具有更好的移植性

如何停止一个正在运行的线程？

1. 使用 interrupt 中断线程

Java 中 interrupted 和 isInterrupted 方法的区别?

interrupt: 用于中断线程。

interrupted: 是静态方法, 查看当前中断信号是true 还是false并且清除中断信号。如果一个线程被中断了第一次调用 interrupted 返回true, 第二次和后面的返回false。

isInterrupted: 查看当前中断信号是true 还是false

♡ Java 如何实现多线程之间的通讯和协作?

经典的生产者、消费者模式。

java 中通信协作的两种方式

- synchronized加锁的线程的Object类的wait()、notify()、notifyAll()
- ReentrantLock类加锁的线程的Condition的 await()、signal()、signalAll()

通过管道进行线程间通信：字节流、字符流

线程的同步

♡ 同步的范围越小越好。

为何要使用同步?

java允许多线程并发控制, 当多个线程同时操作一个可共享的资源变量时(如数据的增删改查), 将会导致数据不准确, 相互之间产生冲突, 因此加入同步锁以避免在该线程没有完成操作之前, 被其他线程的调用, 从而保证了该变量的唯一性和准确性。

如何实现同步?

1. 使用synchronized关键字

即有synchronized关键字修饰的方法(所有访问状态变量的方法都必须进行同步), 此时充当锁的对象为调用同步方法的对象。在调用该方法前, 需要获得内置锁, 否则就处于阻塞状态。如下代码:

```
public class Test{
    private int count = 0;

    public synchronized int add(){
        count += 1;
        return count;
    }

    public synchronized int delete(){
        count -= 1;
        return count;
    }
}
```

2. 同步代码块

即有synchronized关键字修饰的语句块。锁的粒度更细, 并且充当锁的对象不一定是this, 也可以是其它对象, 使用起来更加灵活。

```
public class Test{
    private int count = 0;

    public int add(){
```

```

        synchronized(this){
            count += 1;
        }
        return count;
    }

    public int delete(){
        synchronized(this){
            count -= 1;
        }
        return count;
    }
}

```

同步是一种高开销的操作，因此应该尽量减少同步的内容。通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。下面看几个例子：

3. 使用volatile 实现线程同步
4. 使用Reentrantlock

请说出你所知道的线程同步的方法。

- wait():使一个线程处于等待状态，并且释放所持有的对象的lock。
- sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException异常。
- notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且不是按优先级。
- notifyAll():唤醒所有处入等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

线程类的构造方法、静态块是被哪个线程调用的

线程类的构造方法、静态块是被new 这个线程类所在的线程所调用的，而run 方法里面的代码是被线程自身所调用的。

举个例子：假设 Thread2 中 new 了Thread1， main 函数中 new 了 Thread2，那么：

- (1) Thread2 的构造方法、静态块是 main 线程调用的， Thread2 的 run()方法是Thread2 自己调用的
- (2) Thread1 的构造方法、静态块是 Thread2 调用的， Thread1 的 run()方法是Thread1 自己调用的

♡ 线程池

FixedThreadPool

固定大小的线程池，初始化时指定线程池的大小。当向线程池中添加线程时，首先判断是否小于 corePoolSize，如果满足条件则像线程池中添加线程。当线程预热完成后，向队列中添加线程。底层实现是使用LinkedBlockedQueue 无界队列。会反复从队列中获取任务执行。

SingleThreadPool

核心线程池为1的线程数，底层也是使用LinkedBlockingQueue实现的。当预热完成后从队列中获取任务

CacheThreadPool

核心线程池数量为0，底层是使用一个无容量的 SynchronousQueue 实现的，它的最大线程池数量为Integer.Max_Value 。当线程池有空闲线程时，主线程执行offer() 操作，线程池执行poll() 操作，表示匹配成功，主线程把任务交给空闲线程执行。如果此时没有空闲线程，那么会新建一个新线程任务。再去与主线程匹配。当线程池空闲线程等待 60 秒无操作就会自动终止。

ScheduledThreadPool

底层是使用delayQueue 实现的，封装了一个优先级队列。将time 小的放到前面。

- 创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

饱和策略

当线程池满后一共有4种策略

- 直接抛出异常 AbortPolicy
- 只用调用者所在线程来运行任务 CallerRunsPolicy
- 丢弃队列里最近一个任务，并执行当前任务 DiscardOldestPolicy
- 不处理，丢弃掉 DiscardPolicy

如果你提交任务时，线程池队列已满，这时会发生什么

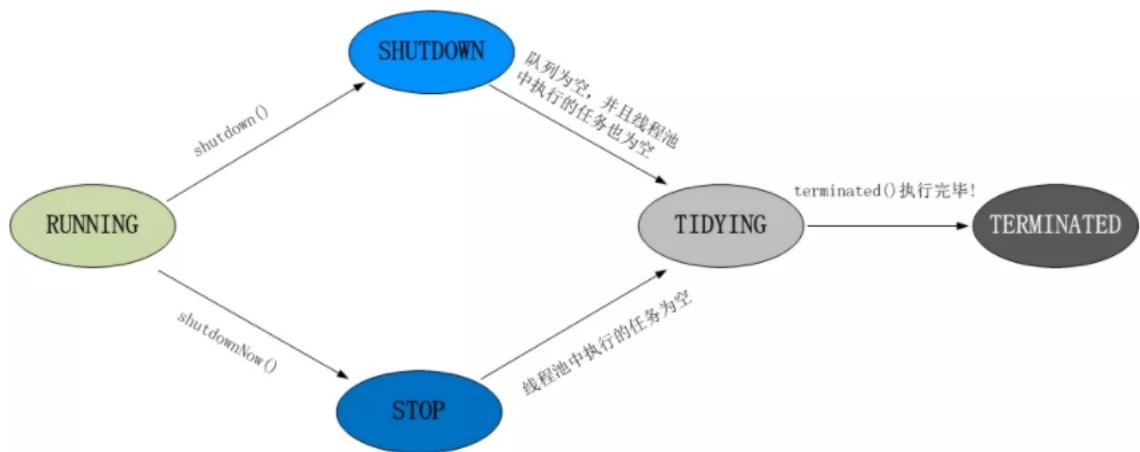
这里区分一下：

- (1) 如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务
- (2) 如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy

线程池都有哪些状态？

线程池有5种状态：Running、ShutDown、Stop、Tidying、Terminated。

线程池各个状态切换框架图：



线程池中 submit() 和 execute() 方法有什么区别？

- 接收的参数不一样
- submit有返回值，而execute没有
- submit方便Exception处理

在 java 程序中怎么保证多线程的运行安全？

方法一：使用安全类，比如 java.util.concurrent 下的类，使用原子类AtomicInteger

方法二：使用自动锁 synchronized。

方法三：使用手动锁 Lock。

```

Lock lock = new ReentrantLock();
lock.lock();
try {
    System.out.println("获得锁");
} catch (Exception e) {
    // TODO: handle exception
} finally {
    System.out.println("释放锁");
    lock.unlock();
}

```

synchronized

三种使用方式

分类	具体分类	被锁的对象	伪代码
方法	实例方法	类的实例对象	//实例方法，锁住的是该类的实例对象 public synchronized void method() { }
	静态方法	类对象	//静态方法，锁住的是类对象 public static synchronized void method1() { }
代码块	实例对象	类的实例对象	//同步代码块，锁住的是该类的实例对象 synchronized (this) { }
	class对象	类对象	//同步代码块，锁住的是该类的类对象 synchronized (SynchronizedDemo.class) { }
	任意实例对象Object	实例对象Object	//同步代码块，锁住的是配置的实例对象 //String对象作为锁 String lock = ""; synchronized (lock) { }

如图，synchronized可以用在**方法**上也可以使用在**代码块**中，其中方法是实例方法和静态方法分别锁的是该类的实例对象和该类的对象。而使用在代码块中也可以分为三种，具体的可以看上面的表格。这里的需要注意的是：**如果锁的是类对象的话，尽管new多个实例对象，但他们仍然是属于同一个类依然会被锁住，即线程之间保证同步关系。**

总结：synchronized关键字加到 static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁。synchronized关键字加到实例方法上是给对象实例上锁。

synchronized 底层实现原理？

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

synchronized 是由JVM 实现的一种互斥同步的方式，通过字节码会发现被 synchronized 修饰的程序段会被 **monitorenter** 和 **monitorexit** 两个字节码指令包住。

```
public void method();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
    0: aload_0
    1: dup
    2: astore_1
    3: monitorenter           // Field java/lang/System.out:Ljava/io/PrintStream;
    4: getstatic   #2          // String Method 1 start
    7: ldc         #3          // Method java/io/PrintStream.println:(Ljava/lang/String;)V
    9: invokevirtual #4
   12: aload_1
   13: monitorexit            // Field java/lang/System.out:Ljava/io/PrintStream;
   14: goto      22
   17: astore_2
   18: aload_1
   19: monitorexit
   20: aload_2
   21: athrow
   22: return
  Exception table:
    from   to   target type
      4    14    17  any
     17    20    17  any
LineNumberTable:
  line 5: 0
  line 6: 4
  line 7: 12
  line 8: 22
StackMapTable: number_of_entries = 2
  frame_type = 255 /* full_frame */
  offset_delta = 17
  locals = [ class test/SynchronizedDemo, class java/lang/Object ]
  stack = [ class java/lang/Throwable ]
  frame_type = 250 /* chop */
  offset_delta = 4
}
SourceFile: "SynchronizedDemo.java"
```

为什么会有两个monitorexit呢？

防止同步代码块中线程在异常退出时，锁没有得到释放，这样必然造成死锁。因此最后一个确保在异常情况下，锁也可以得到释放，避免死锁。

synchronized可重入的原理

底层维护了一个计数器，当线程获取锁时，计数器加一，释放锁时计数器减一。当计数器为0时，表明该锁未被任何线程所持有，其他线程可以竞争获取锁。

什么是自旋

很多synchronized里面的代码都是一些简单的代码，执行时间非常快，此时的等待线程加锁可能是一种不太值得的操作，因为线程涉及到用户态和内核态切换的问题。所以让等待锁的线程不要被阻塞，在synchronized边界做循环，这就是自旋。

如果做了多次自选没有获取到锁，在阻塞是一个不错的策略。

多线程中 synchronized 锁升级的原理是什么？

在锁的对象头中有一个threadId字段，第一次访问时ThreadId 为空，JVM让其拥有偏向锁，并将ThreadId设置为线程ID，再次进入时会判断线程ID 和ThreadId 是否一致，如果一致那么直接使用此对象。

不一致将偏向锁升级为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数后还没有获取到则将轻量级锁升级为重量级锁。

锁升级的目的？

锁升级降低锁带来的性能消耗。

当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B？

不可以，其他线程只能访问该对象的非同步方法，同步方法则不同进入。因为非静态方法上的synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入A 方法，那么说明对象锁已经被取走，那么试图进入B 方法的线程就只能等锁池中等待对象的锁。

synchronized 和 Lock 有什么区别？

- 首先synchronized是java内置关键字，在jvm层面，**Lock**是个java类；
- synchronized无法判断是否获取锁的状态，**Lock**可以判断是否获取到锁；
- synchronized 会自动释放锁(a 线程执行完同步代码会释放锁； b 线程执行过程中发生异常会释放锁)，**Lock需在 finally中手工释放锁（unlock()方法释放锁）**，否则容易造成线程死锁；
- 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而**Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了**；
- synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平（两者皆可）；
- Lock锁适合**大量同步的代码的同步问题**，synchronized锁适合**代码少量的同步问题**。

synchronized 和 ReentrantLock 区别是什么？

synchronized是和if、else、for、while一样的**关键字**，**ReentrantLock是类**，这是二者的本质区别。

既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

- ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
- ReentrantLock可以获取各种锁的信息
- ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的：

- ReentrantLock底层调用的是Unsafe的park方法加锁，
- synchronized操作的应该是对象头中mark word。

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

- 普通同步方法，锁是当前实例对象
- 静态同步方法，锁是当前类的class对象
- 同步方法块，锁是括号里面的对象

♡ volatile

volatile关键字的作用

volatile 主要体现在以下两方面，重排序和内存可见性。volatile提供了happens-before原则保证，确保一个线程的修改对另一个线程是可见的

♡ final

都哪些类是final的？

基本数据类型包装类：

Boolean, Character, Short, Integer, Long, Float, Double, Byte, Void

字符串类

String StringBuffer StringBuilder

synchronized 和 volatile 的区别是什么？

- volatile本质是在告诉 JVM 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
- **volatile仅能使用在变量级别**； synchronized则可以使用在变量、方法、和类级别的。
- volatile仅能实现变量的修改可见性，不能保证原子性；而synchronized则可以保证变量的**修改可见性和原子性**。
- **volatile不会造成线程的阻塞**； synchronized可能会造成线程的阻塞。
- **volatile标记的变量不会被编译器优化**； synchronized标记的变量可以被编译器优化

Synchronized 和 ThreadLocal 的区别

相同点 都用于解决多线程并发访问，防止任务在共享资源上产生冲突

不同点：

- **Synchronized** 用于实现同步机制，是利用锁的机制使变量或代码块在某一时刻只能被一个线程访问，是一种 **以时间换空间** 的方式
- **ThreadLocal** 为每一个线程都提供了变量的副本，使得每个线程在某一个时间访问到的并不是同一个对象，根除了对变量的共享，是一种 **以空间换时间**的方式

♡ ThreadLocal 是什么？有哪些使用场景？

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。

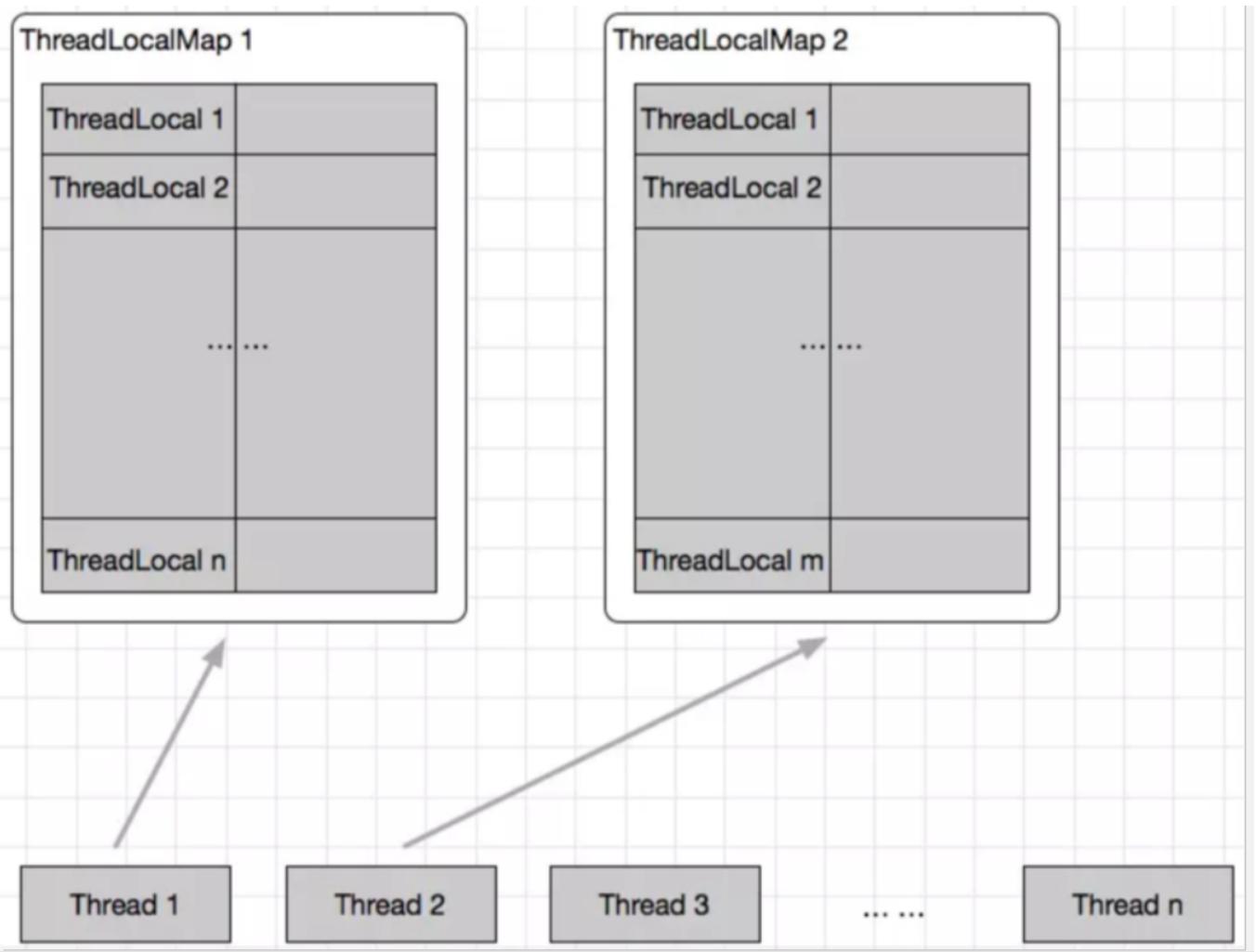
Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

25.1 ThreadLocal 是怎么解决并发安全的

ThreadLocal **为每一个线程维护变量的副本，把共享数据可见性范围限制在同一个线程内**

实现原理：在ThreadLocal 里存在一个Map 用于存储每一个线程的变量的副本

一图胜千言



25.2 Thread Local 需要注意什么

要注意remove:

ThreadLocal 的实现是基于一个所谓的 **ThreadLocalMap**，在其中，它的key 是一个弱引用。

通常弱引用都会和引用队列配合清理机制使用，但是 ThreadLocal 没有这么做

这意味着，废弃项目的回收依赖于显示地触发，否则就要等待线程结束，进而回收相应 ThreadLocal，这就是很多OOM 的来源。所以都会建议应用一定要自己负责 remove，并且不要和线程池配合，因为worker 线程往往是不会退出的。

26. JVM 对 Java 原生锁做了哪些优化

在 Java 6 之前，Monitor 的实现完全依赖底层操作系统的互斥锁来实现，也就是我们刚才在问题二中所阐述的获取/释放锁的逻辑。

由于 Java 层面的线程与操作系统的原生线程有映射关系，如果要将一个

线程进行阻塞或唤起都需要操作系统的协助，这就需要从用户态切换到内核态来执行，这种切换代价十分昂贵，很耗处理器时间，现代 JDK 中做了大量的优化。一种优化是使用自旋锁，即在把线程进行阻塞操作之前先让线程自旋等待一段时间，可能在等待期间其他线程已经解锁，这时就无需再让线程执行阻塞操作，避免了用户态到内核态的切换。

现代 JDK 中还提供了三种不同的 Monitor 实现，也就是三种不同的 锁：

- 偏向锁(Biased Locking)
- 轻量级锁
- 重量级锁

这三种锁使得 JDK 得以优化 Synchronized 的运行，当 JVM 检测 到不同的竞争状况时，会自动切换到适合的锁实现，这就是锁的升级、降级。

- 当没有竞争出现时，默认会使用偏向锁。

JVM 会利用 CAS 操作，在对象头上的 Mark Word 部分设置线程 ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁，因为在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

- 如果有另一线程试图锁定某个被偏斜过的对象，JVM 就撤销偏斜锁，切换到轻量级锁实现。
- 轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁。

27.为什么说 Synchronized 是非公平锁

非公平主要表现在获取锁的行为上，并非是按照申请锁的时间前后，给等待线程分配锁，每当锁被释放后，任何一个线程都有机会竞争到锁，这样的目的是为了提高执行性能，缺点是可能会产生线程饥饿线像。

28.什么是锁消除和锁粗化？

锁消除：指虚拟机即时编译 (JIT) 在运行时，对一些代码上要求同步，但被检查到可能存在共享数据竞争的锁进行消除。主要根据逃逸分析

锁粗化：原则上，同步块的作用范围要尽量小。如果一系列的操作都对同一个对象反复加锁和解锁，甚至加锁操作在循环体内，频繁的进行互斥同步也会导致不必要的性能损耗。

锁粗化实质就是增大锁的作用域

29.CAS

什么是 CAS，它有什么特性？

Synchronized 显然是一个悲观锁，因为它的并发策略是悲观的：不管是否会产生竞争，任何的数据操作都必须要加锁、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略。先进行操作，如果没有其他线程征用数据，那操作就成功了；如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种乐观的并发策略的许多实现不需要线程挂起，所以被称为非阻塞同步。乐观锁的核心算法是 CAS(Compareand Swap，比较并交换)，它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。CAS 具有原子性，它的原子性由 CPU 硬件指令实现保证，即使用 JNI 调用 Native 方法调用由 C++ 编写的硬件级别指令，JDK 中提供了 Unsafe 类执行这些操作。

30.ABA问题、乐观锁的缺点

乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，但它也有缺点：

1. 乐观锁只能保证一个共享变量的原子操作。如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。
2. 长时间自旋可能导致开销大。假如 CAS 长时间不成功而一直自旋，会给 CPU 带来很大的开销。
3. ABA 问题。CAS 的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是 A，后来被一条线程改为 B，最后又被改成了 A，则 CAS 认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入版本号，每次变量更新都把版本号加一。

31.AQS 问题

AQS(Abstract Queued Synchronizer)类是一个用来构建锁和同步器的框架，各种Lock包中的锁(如ReentrantLock、ReadWriteLock)，以及其他如 Semaphore、CountDownLatch，甚至早期的FutureTask 等，都是基于AQS构建的

1. AQS 在内部定义了一个 volatile int state 变量，表示同步状态：当线程调用 lock 方法时，如果 state=0，说明没有任何线程占有共享资源的锁，可以获得锁并将 state=1；如果 state=1，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待。
2. AQS 通过 Node 内部类构成的一个双向链表结构的同步队列，来完成线程获取锁的排队工作，当有线程获取锁失败后，就被添加到队列末尾。
 - Node 类是对要访问同步代码的线程的封装，包含了线程本身及其状态叫 waitStatus(有五种不同取值，分别表示是否被阻塞，是否等待唤醒，是否已经被取消等)，每个 Node 结点关联其 prev 结点和 next 结点，方便线程释放锁后快速唤醒下一个在等待的线程，是一个 FIFO 的过程。
 - Node 类有两个常量，SHARED 和 EXCLUSIVE，分别代表共享模式和独占模式。所谓共享模式是一个锁允许多条线程同时操作(信号量 Semaphore 就是基于 AQS 的共享模式实现的)，独占模式是同一个时间段只能有一个线程对共享资源进行操作，多余的请求线程需要排队等待(如 ReentrantLock)。
3. AQS 通过内部类 ConditionObject 构建等待队列(可有多个)，当 Condition 调用 wait() 方法后，线程将会加入等待队列中，而当 Condition 调用 signal() 方法后，线程将从等待队列转移到同步队列中进行锁竞争。
4. AQS 和 Condition 各自维护了不同的队列，在使用 Lock 和 Condition 的时候，其实就是两个队列的互相移动。

32. ReentrantLock 是如何实现可重入性的？

ReentrantLock 内部自定义了同步器 Sync(Sync 既实现了 AQS，又实现了 AOS，而 AOS 提供了一种互斥锁持有的方式)，其实质就是加锁的时候通过 CAS 算法，将线程对象放到一个双向链表中，每次获取锁的时候，看下当前维护的那个线程 ID 和当前请求的线程 ID 是否一样，一样就可重入了。

33. 如何让 Java 的线程彼此同步？

JUC 同步器的三个主要成员：CountDownLatch、CyclicBarrier 和 Semaphore

CountDownLatch 叫倒计数，允许一个或多个线程等待某些操作完成。看几个场景：

- 跑步比赛，裁判需要等到所有的运动员(“其他线程”)都跑到终点(达到目标)，才能去算排名和颁奖。
- 模拟并发，我需要启动 100 个线程去同时访问某一个地址，我希望它们能同时并发，而不是一个一个的去执行。

用法：CountDownLatch 构造方法指明计数数量，被等待线程调用 countDown 将计数器减 1，等待线程使用 await 进行线程等待。一个简单的例子：

CyclicBarrier 叫循环栅栏，它实现让一组线程等待至某个状态之后再全部同时执行，而且当所有等待线程被释放后，CyclicBarrier 可以被重复使用。CyclicBarrier 的典型应用场景是用来等待并发线程结束。CyclicBarrier 的主要方法是 await()，await() 每被调用一次，计数便会减少 1，并阻塞住当前线程。当计数减至 0 时，阻塞解除，所有在此 CyclicBarrier 上面阻塞的线程开始运行。

在这之后，如果再次调用 await()，计数就又会变成 N-1，新一轮重新开始，这便是 Cyclic 的含义所在。CyclicBarrier.await() 带有返回值，用来表示当前线程是第几个到达这个 Barrier 的线程。

```
public class TestCyclicBarrier{  
    private CyclicBarrier cyclicbarrier= new CyclicBarrier(5);  
  
    public static void main(String [] args){  
        new TestCyclicBarrier().begin();  
    }  
  
    public void begin(){  
        for(int i =0;i< 5; i++){  
            new Thread(new Student()).start();  
        }  
    }  
    static class Student implements Runnable{  
        @Override  
        public void run(){  
            try{  
                Thread.sleep(2000); //正在前往饭店的路上  
                cyclicbarrier.await(); //到了就等着，等其他人都到了  
            }catch(Exception e){  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Semaphore，Java 版本的信号量实现，用于控制同时访问的线程个数，来达到限制通用资源访问的目的，其原理是通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

```
public class TestSemaphore {  
    public static void main(String[] args) {  
        //5台机器 即5个许可证  
        Semaphore semaphore = new Semaphore(5);  
        for (int worker = 0; worker < 8; worker++) {  
            new Worker(worker, semaphore).start();  
        }  
    }  
  
    static class Worker extends Thread {  
        private int num;  
        private Semaphore semaphore;
```

```

public Worker(int num, Semaphore semaphore) {
    this.num = num;
    this.semaphore = semaphore;
}

@Override
public void run() {
    try {
        semaphore.acquire(); // 抢许可
        System.out.println(num+"");
        Thread.sleep(2000);
        System.out.println("释放");
        semaphore.release(); // 释放许可
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}

```

如果 Semaphore 的数值被初始化为1，那么一个线程就可以通过 acquire 进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比

如互斥锁是有持有者的，而对于 Semaphore 这种计数器结构，虽然有类似功能，但其实不存在真正意义的持有者，除非我们进行扩展包装。

34.CountDownLatch、CyclicBarrier区别

CyclicBarrier: 生活中的例子，咱们三个人明天中午食堂碰面，都到齐后，商量下火锅吃油碟还是麻酱

作用不同: CyclicBarrier 要等固定数量的线程都到达了栅栏位置才能继续执行，而 CountDownLatch 只需等待数字到0.

CountDownLatch 用于事件，而 CyclicBarrier 用于线程

可用性不同: CountDownLatch 在倒数到0时出发门闩打开后，不能再次使用了。除非创建新的实例。而 CyclicBarrier 可以重复使用

35.Java中的线程池是如何实现的

- 在 Java 中，所谓的线程池中的“线程”，其实是被抽象为了一个静态 内部类 Worker，它基于 AQS 实现，存放在线程池的 HashSet workers 成员变量中；
- 而需要执行的任务则存放在成员变量 workQueue(BlockingQueue workQueue)中。
这样，整个线程池实现的基本思想就是：从 workQueue 中不断取出 需要执行的任务，放在 Workers 中进行处理。

36. 创建线程池的几个核心构造参数？

Java 中的线程池的创建其实非常灵活，我们可以通过配置不同的参数，创建出行为不同的线程池，这几个参数包括：

- corePoolSize：线程池的核心线程数。
- maximumPoolSize：线程池允许的最大线程数。
- keepAliveTime：超过核心线程数时闲置线程的存活时间。
- workQueue：任务执行前保存任务的队列，保存由 execute 方法提交的 Runnable 任务。

37. 线程池中线程是怎么创建的？

显然不是的。线程池默认初始化后不启动 Worker，等待有请求时才启动。

每当我们调用 execute() 方法添加一个任务时，线程池会做如下判断：

- 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
- 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
- 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
- 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。

当一个线程完成任务时，它会从队列中取下一个任务来执行。当一个线程无事可做，超过一定的时间(keepAliveTime)时，线程池会判断。

如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

38. 线程池的异同

SingleThreadExecutor 线程池

- 只有一个核心线程在工作，相当于单线程执行所有任务。
- 如果这唯一一个线程因为异常结束，那么会有一个新的线程来替代它。
- 此线程保证所有任务的执行顺序按照任务的提交顺序执行

- corePoolSize: 1，只有一个核心线程在工作。
- maximumPoolSize: 1。
- keepAliveTime: 0L。
- workQueue: new LinkedBlockingQueue<Runnable>()，其缓冲队列是无界的。

FixedThreadPool 线程池

- 是一个固定大小的线程池，只有核心线程
- 每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。
- 线程池一旦达到最大值就保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程
- 多用于一些很稳定、很固定的正规并发线程，多用于服务器

- corePoolSize: nThreads
- maximumPoolSize: nThreads
- keepAliveTime: 0L
- workQueue: new LinkedBlockingQueue<Runnable>() , 其缓冲队列是无界的。

CachedThreadPool 线程池

- 无界线程池，如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行的任务）线程，当任务数增加时，此线程又可以智能的添加新线程来处理任务
- 线程池的大小完全依赖于操作系统能够创建的最大线程大小。
- SynchronousQueue 是一个缓冲区为 1 的阻塞队列
- 通常用于执行一些生存期很短的异步性任务，因此一些面向连接的daemon型 Server 中用的不多

- corePoolSize: 0
- maximumPoolSize: Integer.MAX_VALUE
- keepAliveTime: 60L
- workQueue: new SynchronousQueue<Runnable>() , 一个是缓冲区为 1 的阻塞队列。

ScheduledThreadPool 线程池

- 核心线程池固定，大小无限的线程池。
- 此线程支持定时以及周期性执行任务的需求
- 创建一个周期性执行的线程池
- 如果闲置，非线程池会在 DEFAULT_KEEPALIVE_MILLIS 时间内回收

- corePoolSize: corePoolSize
- maximumPoolSize: Integer.MAX_VALUE
- keepAliveTime: DEFAULT_KEEPALIVE_MILLIS
- workQueue: new DelayedWorkQueue()

39.线程池的数量该设置为多少合适？

- ◆ CPU密集型（加密、计算hash等）：最佳线程数为CPU核心数的1-2倍左右。
- ◆ 耗时IO型（读写数据库、文件、网络读写等）：最佳线程数一般会大于cpu核心数很多倍，以JVM线程监控显示繁忙情况为依据，保证线程空闲可以衔接上，参考Brain Goetz推荐的计算方法：
- ◆ 线程数=CPU核心数*（1+平均等待时间/平均工作时间）

40.如何在线程池中提交线程

线程池最常用的提交任务的方法有两种:

1. execute(): ExecutorService.execute 方法接收一个例, 它用来执行一个任务:

```
ExecutorService.execute(Runnable runnable)
```

2. submit(): ExecutorService.submit() 方法返回的是 Future 对象。可以用

isDone() 来查询 Future 是否已经完成, 当任务完成时, 它具有一个结果, 可以调用 get() 来获取结果。也可以不用 isDone() 进行检查就直接调用 get(), 在这种情况下, get() 将阻塞, 直至结果准备就绪。

```
FutureTask task = ExecutorService.submit(Runnable runnable);
FutureTask<T> task = ExecutorService.submit(Runnable runnable, T Result);
FutureTask<T> task = ExecutorService.submit(Callable<T> callable);
```

41.什么是Java 内存模型

Java 的内存模型定义了程序中各个变量的访问规则, 即在虚拟机中将 变量存储到内存和从内存中取出这样的底层细节。此处的变量包括实例字段、静态字段和构成数组对象的元素, 但是不包括局部变量和方法参数, 因为这些是线程私有的, 不会被共享, 所以不存在竞争问题。

Java 中各个线程是怎么彼此看到对方的变量的呢?Java 中定义了主内存与工作内存的概念:

所有的变量都存储在主内存, 每条线程还有自己的工作内存, 保存了被 该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作(读取、赋值)都必须在工作内存中进行, 不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量, 线程间变量值的传递需要通过主内存。

42todo.锁的升、降级

43.volatile 有什么特点, 为什么它能保证变量对所有的线程可见性

volatile 是 Java 虚拟机提供的轻量级的同步机制。

当一个变量被volatile 修饰

- 保证变量被所有线程的可见性。当一条线程修改了这个变量的值, 新的值对于其他线程是可以立即得知的。
- 禁止指令重排序优化: 普通变量仅仅能保证在该方法执行过程中, 得到正确结果, 但是不能保证程序代码的执行顺序。

44.Java 内存模型定义的8 种内存间操作

lock 和 unlock

- 把一个变量标识为一条线程独占的状态。
- 把一个处于锁定状态的变量释放出来，释放之后的变量才能被其他线程锁定。

read 和 write

- 把一个变量值从主内存传输到线程的工作内存，以便 load。
- 把 store 操作从工作内存得到的变量的值，放入主内存的变量中。

load 和 store

- 把 read 操作从主内存得到的变量值放入工作内存的变量副本中。
- 把工作内存的变量值传送到主内存，以便 write。

use 和 assign

- 把工作内存变量值传递给执行引擎。
- 将执行引擎值传递给工作内存变量值。

volatile 的实现基于这 8 种内存间操作，保证了一个线程对某个 volatile 变量的修改，一定会被另一个线程看见，即保证了可见性。

45.volatile是否可以保证并发安全

不可以。虽然 volatile 保证了一致性但是 java 里面的运算并非是原子操作，导致了 volatile 在并发下是不安全的。

46.说一下 atomic 的原理？

Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

Atomic系列的类中的核心方法都会调用unsafe类中的几个本地方法。我们需要先知道一个东西就是Unsafe类，**全名为：sun.misc.Unsafe**，这个类包含了大量的对C代码的操作，包括很多直接内存分配以及原子操作的调用，而它之所以标记为非安全的，是告诉你这个里面大量的方法调用都会存在安全隐患，需要小心使用，否则会导致严重的后果，例如在通过 unsafe分配内存的时候，如果自己指定某些区域可能会导致一些类似C++一样的指针越界到其他进程的问题。

☆ RabbitMQ ☆

1.的使用场景有哪些？

①. 跨系统的异步通信，所有需要异步交互的地方都可以使用消息队列。就像我们除了打电话（同步）以外，还需要发短信，发电子邮件（异步）的通讯方式。

②. 多个应用之间的耦合，由于消息是平台无关和语言无关的，而且语义上也不再是函数调用，因此更适合作为多个应用之间的松耦合的接口。基于消息队列的耦合，不需要发送方和接收方同时在线。在企业应用集成（EAI）中，文件传输，共享数据库，消息队列，远程过程调用都可以作为集成的方法。

③. 应用内的同步变异步，比如订单处理，就可以由前端应用将订单信息放到队列，后端应用从队列里依次获得消息处理，高峰时的大量订单可以积压在队列里慢慢处理掉。由于同步通常意味着阻塞，而大量线程的阻塞会降低计算机的性能。

- ④. 消息驱动的架构 (EDA) , 系统分解为消息队列, 和消息制造者和消息消费者, 一个处理流程可以根据需要拆成多个阶段 (Stage) , 阶段之间用队列连接起来, 前一个阶段处理的结果放入队列, 后一个阶段从队列中获取消息继续处理。
- ⑤. 应用需要更灵活的耦合方式, 如发布订阅, 比如可以指定路由规则。
- ⑥. 跨局域网, 甚至跨城市的通讯 (CDN行业) , 比如北京机房与广州机房的应用程序的通信。

2.有哪些重要的角色?

RabbitMQ 中重要的角色有：生产者、消费者和代理：

- 生产者：消息的创建者，负责创建和推送数据到消息服务器；
- 消费者：消息的接收方，用于处理数据和确认消息；
- 代理：就是 RabbitMQ 本身，用于扮演“快递”的角色，本身不生产消息，只是扮演“快递”的角色。

3.有哪些重要的组件?

- ConnectionFactory (连接管理器)：应用程序与Rabbit之间建立连接的管理器，程序代码中使用。
- Channel (信道)：消息推送使用的通道。
- Exchange (交换器)：用于接受、分配消息。
- Queue (队列)：用于存储生产者的消息。
- RoutingKey (路由键)：用于把生成者的数据分配到交换器上。
- BindingKey (绑定键)：用于把交换器的消息绑定到队列上。

4.rabbitmq 中 vhost 的作用是什么?

vhost 可以理解为虚拟 broker，即 mini-RabbitMQ server。其内部均含有独立的 queue、exchange 和 binding 等，但最重要的是，其拥有独立的权限系统，可以做到 vhost 范围的用户控制。当然，从 RabbitMQ 的全局角度，vhost 可以作为不同权限隔离的手段（一个典型的例子就是不同的应用可以跑在不同的 vhost 中）。

5.rabbitmq 的消息是怎么发送的?

首先客户端必须连接到 RabbitMQ 服务器才能发布和消费消息，客户端和 rabbit server 之间会创建一个 tcp 连接，一旦 tcp 打开并通过了认证（认证就是你发送给 rabbit 服务器的用户名和密码），你的客户端和 RabbitMQ 就创建了一条 amqp 信道 (channel)，信道是创建在“真实”tcp 上的虚拟连接，amqp 命令都是通过信道发送出去的，每个信道都会有一个唯一的 id，不论是发布消息，订阅队列都是通过这个信道完成的。

6.rabbitmq 怎么保证消息的稳定性?

- 提供了事务的功能。
- 通过将 channel 设置为 confirm (确认) 模式。

7.rabbitmq 怎么避免消息丢失?

- 消息持久化
- ACK确认机制
- 设置集群镜像模式
- 消息补偿机制

8.要保证消息持久化成功的条件有哪些?

1. 声明队列必须设置持久化 durable 设置为 true.
2. 消息推送投递模式必须设置持久化，deliveryMode 设置为 2 (持久) 。
3. 消息已经到达持久化交换器。
4. 消息已经到达持久化队列。

以上四个条件都满足才能保证消息持久化成功。

9.rabbitmq 持久化有什么缺点？

持久化的缺点就是降低了服务器的吞吐量，因为使用的是磁盘而非内存存储，从而降低了吞吐量。可尽量使用 ssd 硬盘来缓解吞吐量的问题。

10.rabbitmq 有几种广播类型？

三种广播模式：

1. fanout: 所有bind到此exchange的queue都可以接收消息（纯广播，绑定到RabbitMQ的接受者都能收到消息）；
2. direct: 通过routingKey和exchange决定的那个唯一的queue可以接收消息；
3. topic:所有符合routingKey(此时可以是一个表达式)的routingKey所bind的queue可以接收消息；

11.rabbitmq 怎么实现延迟消息队列？

1. 通过消息过期后进入死信交换器，再由交换器转发到延迟消费队列，实现延迟功能；
2. 使用 RabbitMQ-delayed-message-exchange 插件实现延迟功能。

12.rabbitmq 集群有什么用？

集群主要有以下两个用途：

- 高可用：某个服务器出现问题，整个 RabbitMQ 还可以继续使用；
- 高容量：集群可以承载更多的消息量

13.rabbitmq 节点的类型有哪些？

- 磁盘节点：消息会存储到磁盘。
- 内存节点：消息都存储在内存中，重启服务器消息丢失，性能高于磁盘类型。

14.rabbitmq 集群搭建需要注意哪些问题？

- 各节点之间使用“--link”连接，此属性不能忽略。
- 各节点使用的 erlang cookie 值必须相同，此值相当于“秘钥”的功能，用于各节点的认证。
- 整个集群中必须包含一个磁盘节点。

15.rabbitmq 每个节点是其他节点的完整拷贝吗？为什么？

不是，原因有以下两个：

1. **存储空间的考虑**：如果每个节点都拥有所有队列的完全拷贝，这样新增节点不但没有新增存储空间，反而增加了更多的冗余数据；
2. **性能的考虑**：如果每条消息都需要完整拷贝到每一个集群节点，那新增节点并没有提升处理消息的能力，最多是保持和单节点相同的性能甚至是更糟。

16.rabbitmq 集群中唯一一个磁盘节点崩溃了会发生什么情况？

如果唯一磁盘的磁盘节点崩溃了，不能进行以下操作：

- 不能创建队列
- 不能创建交换器
- 不能创建绑定
- 不能添加用户
- 不能更改权限
- 不能添加和删除集群节点

唯一磁盘节点崩溃了，集群是可以保持运行的，但你不能更改任何东西。

17.rabbitmq 对集群节点停止顺序有要求吗？

RabbitMQ 对集群的停止的顺序是有要求的，应该先关闭内存节点，最后再关闭磁盘节点。如果顺序恰好相反的话，可能会造成消息的丢失。

☆zookeeper☆

1. zookeeper 是什么？

zookeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 google chubby 的开源实现，是 hadoop 和 hbase 的重要组件。

它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

2.zookeeper 都有哪些功能？

- 集群管理：监控节点存活状态、运行请求等。
- 主节点选举：主节点挂掉了之后可以从备用的节点开始新一轮选主，主节点选举说的就是这个选举的过程，使用 zookeeper 可以协助完成这个过程。
- 分布式锁：zookeeper 提供两种锁：独占锁、共享锁。独占锁即一次只能有一个线程使用资源，共享锁是读锁共享，读写互斥，即可以有多线程同时读同一个资源，如果要使用写锁也只能有一个线程使用。zookeeper 可以对分布式锁进行控制。
- 命名服务：在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。

3.zookeeper 有几种部署模式？

zookeeper 有三种部署模式：

- 单机部署：一台集群上运行；
- 集群部署：多台集群运行；
- 伪集群部署：一台集群启动多个 zookeeper 实例运行。

4.zookeeper 怎么保证主从节点的状态同步？

zookeeper 的核心是原子广播，这个机制保证了各个 server 之间的同步。实现这个机制的协议叫做 zab 协议。

zab 协议有两种模式，分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，zab 就进入了恢复模式，当领导者被选举出来，且大多数 server 完成了和 leader 的状态同步以后，恢复模式就结束了。状态同步保证了 leader 和 server 具有相同的系统状态。

5.集群中为什么要有主节点？

在分布式环境中，有些业务逻辑只需要集群中的某一台机器进行执行，其他的机器可以共享这个结果，这样可以大大减少重复计算，提高性能，所以就需要主节点。

6.集群中有 3 台服务器，其中一个节点宕机，这个时候 zookeeper 还可以使用吗？

可以继续使用，单数服务器只要没超过一半的服务器宕机就可以继续使用。

7.说一下 zookeeper 的通知机制？

客户端会对某个 znode 建立一个 watcher 事件，当该 znode 发生变化时，这些客户端会收到 zookeeper 的通知，然后客户端可以根据 znode 变化来做出业务上的改变。

☆ Dubbo ☆

1、Dubbo 中 zookeeper 做注册中心，如果注册中心集群都挂掉，发布者和订阅者之间还能通信么？

可以通信的，启动 dubbo 时，消费者会从 zk 拉取注册的生产者的地址接口等数据，缓存在本地。每次调用时，按照本地存储的地址进行调用；注册中心对等集群，任意一台宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务恢复；挂掉是不要紧的，但前提是你要调用新的服务，则是不能办到的。

(2) 健壮性：

- 监控中心宕掉不影响使用，只是丢失部分采样数据
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
- **注册中心对等集群，任意一台宕掉后，将自动切换到另一台**
- **注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯**
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

☆ MySQL ☆

关系型数据库和非关系型数据库的区别？

(一) 非关系型数据库的优势

性能：NOSQL 是基于键值对的形式进行数据存储的，不要经过SQL 层层解析，所以性能非常高

可扩展性：因为存储的是键值对，数据之间没有耦合性，所以容易水平扩展

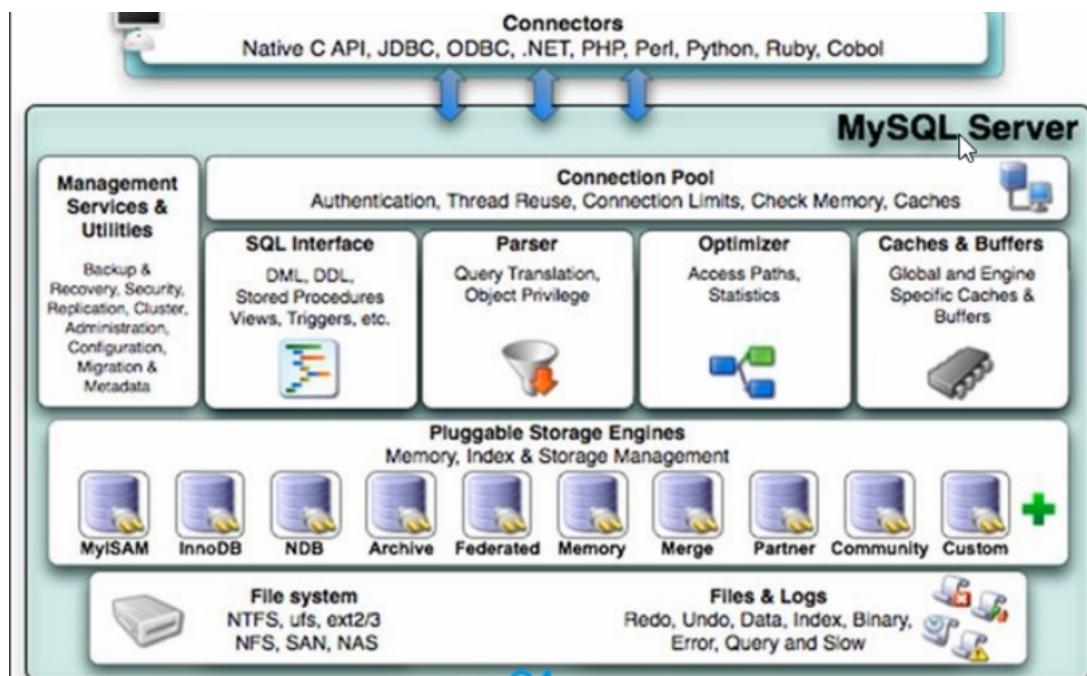
(二) 关系型数据库的优势

复杂查询 可以使用SQL 语句在一个表和多个表之间做复杂的数据查询

事务支持：对于安全性很高的数据得以实现

Redis 用一些原子操作来曲线救国，比如使用 set nx 。当数据在库中存在，就不会去添加

Mysql架构



MySQL插件式的存储引擎架构将 **查询处理** 和其它的**系统任务** 以及 **数据存储** 相分离。

可以根据业务需求选择合适的存储引擎。

整体分为连接层、服务层、引擎层、存储层

服务层

- **解析器** SQL命令将通过解析器进行验证和解析
- **查询优化器** SQL 语句在查询之前会先使用查询优化器进行优化
- **cache&buffer** 查询缓存：如果查询缓存有命中的结果那么那么查询语句直接从缓存中取出。

3. 说一下MySQL执行一条查询语句的内部执行过程？

1. 首先客户端通过连接器连接到MySQL服务器
2. 连接器权限验证通过后，先查询是否有缓存，如果有缓存（之前执行过此语句）则直接返回缓存中的数据，如果没有缓存进入分析器。
3. 分析器会对SQL 进行语法分析和词法分析，判断SQL 是否正确，如果查询语法错误，直接返回客户端错误信息，否则进入优化器
4. **优化器对查询语句进行优化处理**，例如一个表中有多个索引，优化器判断哪个索引更好
5. 优化器执行完成就进入执行器，执行器开始执行语句，如果查询到满足条件的所有数据，返回数据。

4.SQL Select 语句的执行顺序

select--from--where--group by--having--order by

from ---> where ---> group by ---> having ---> 计算所有的表达式 ---> order by--->select 输出

from 从哪个数据库表检索数据

where 过滤表中的条件

group by 将过滤的条件进行分组

having 对上面已经分组的数据进行过滤

order by 按什么样的顺序排序

select 查看结果集中的哪个列，或列的计算结果

优化器选择查询索引的影响因素?

优化器的目的是使用最小代价选择最优的执行方案，影响优化器选择索引的因素如下：

- 扫描行数，扫描的行数越少，执行的代价越少，执行的效率越高
- 是否使用了临时表
- 是否排序

3.数据库的三范式是什么？

- 第一范式：属性不可分
- 第二范式：唯一性，一个数据库表中只能保存一种数据，不可以把多种数据保存到同一张数据库表中。**(所有非主属性必须完全依赖于主码)**
- 第三范式：独立性（每个字段独立依赖于主键字段），消除传递依赖

第一范式

院系	高级职称人数	
	教授	副教授
计算机	5	8
通信	5	9
自动化	6	6

上表就不符合第一范式，正确的差分如下，**满足第一范式**

院系	教授	副教授
计算机	5	8
通信	5	9
自动化	6	6

第二范式

唯一性，一张表中只能保存依赖于主键的那一部分数据。所有非主属性必须依赖于主码。

比如关系模型（职工号，姓名，职称，项目号，项目名称）就不满足第二范式。

正确表示为 职工表（职工号，姓名，职称）；项目表（项目号，项目名称）

第三范式

每一个属性跟主键存在直接关系，而不存在间接关系

比如student表（学号，姓名，年龄，所在院校，院校地址）应该拆分为

student表（学号，姓名，年龄）院校表（院校名称，院校地址）

♡ InnoDB和MyIASM的区别？

InnoDB 引擎：

- mysql 5.1 后默认的数据库引擎，提供了对数据库 acid 事务的支持，
- 并且还提供了**行级锁和外键**的约束，它的设计的目标就是**处理大数据容量的数据库系统**。
- MySQL 运行的时候，InnoDB 会在内存中建立缓冲池，用于缓冲数据和索引。
- 但是该引擎是不支持全文搜索，同时启动也比较慢，它是不会保存表的行数的，所以当进行 select count(*) from table 指令的时候，需要**进行扫描全表**。

- 由于锁的粒度小，写操作是不会锁定全表的，所以在并发度较高的场景下使用会提升效率的。
- 由于InnoDB不支持hash索引，但是在某些情况hash索引的效率很高，InnoDB引擎会监控表上索引的查找，如果观察到建立hash索引可以提高性能，则自动建立hash索引。

MyISAM引擎：

- 不提供事务的支持，也不支持行级锁和外键。
- 因此当执行插入和更新语句时，即执行写操作的时候需要**锁定这个表**，所以会导致效率会降低。
- 不过和InnoDB不同的是，**MyISAM引擎是保存了表的行数**，于是当进行select count(*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。
- 所以，如果表的读操作远多于写操作时，并且不需要事务的支持的，可以将MyISAM作为数据库引擎的首选。

InnoDB和MyISAM面试题

♥一张自增表里面总共有7条数据，删除了最后2条数据，重启mysql数据库，又插入了一条数据，此时id是几？

表类型如果是MyISAM，那id就是8。

- 因为MyISAM会把自增主键的最大ID记录到数据库文件里面，重启MySQL后，自增主键的最大ID也不会丢失。

表类型如果是InnoDB，那id就是6。

- 因为如果不重启，这条记录就是8，但是如果重启MySQL的话，这条记录就是6，因为InnoDB把自增主键的最大ID记录存在内存中，所以重启会使最大ID丢失

♥索引

索引的优缺点

(一) 优点

- 快速访问数据表中的特定信息，提高检索速度
- 创建唯一索引，保证数据表中每一行数据的唯一性
- 加速表与表之间的连接
- 使用分组和排序进行数据检索时，可以减少查询中分组和排序的时间

(二) 缺点

- 虽然显著提高了查询速度，但是降低了更新表的速度。
- 建立索引会占用磁盘文件的索引文件

索引的类型

(一) 按逻辑分类

主键索引

- 一张表只有一个主键索引，不允许重复，不允许为空

唯一索引

- 数据列不允许重复，允取为空，一张表可以有多个唯一索引，但是一个唯一索引只能包含一列，比如身份证号码、卡号

普通索引

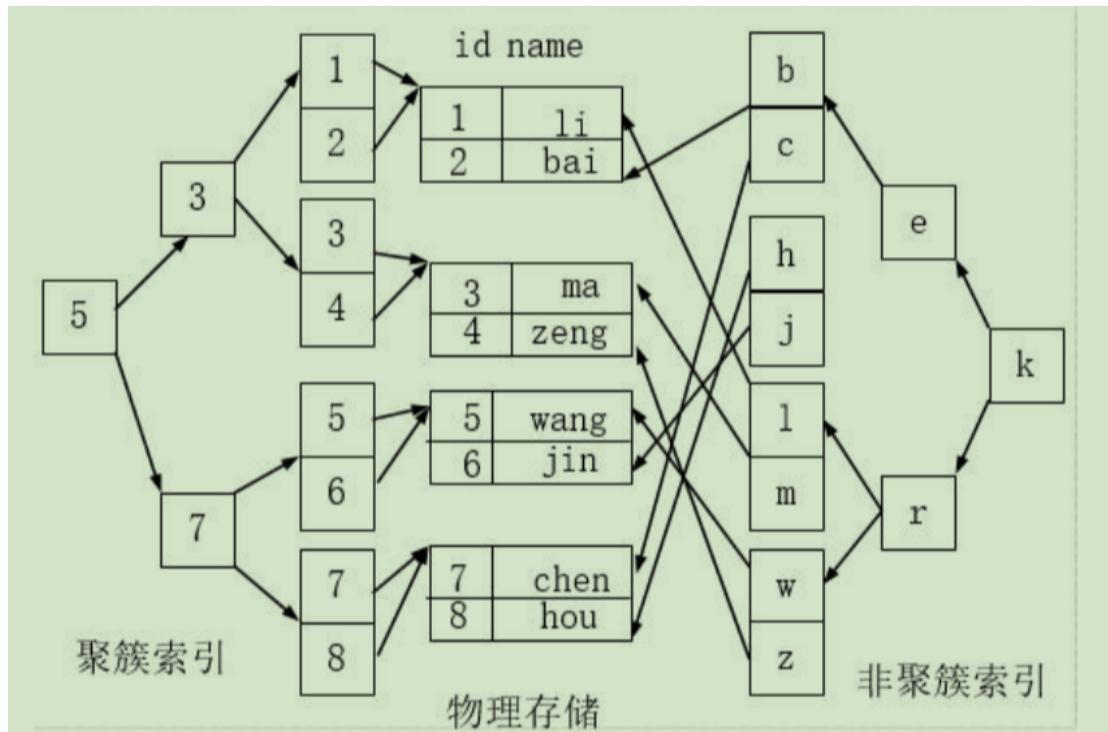
- 一张表可以有多个普通索引，一个普通索引可以包含多个字段，允取数据重复，允取空值

全文索引

(二) 按物理分类

1) 聚簇索引

- 聚簇索引并不是一种单独的索引类型，而是一种存储方式
- 聚簇表示 数据行 和 相邻的键值聚簇的存储在一起



聚簇索引的好处

按照聚簇索引排列顺序，查询显示一定范围的时候，由于数据是紧密相连的，数据库不用从多个数据块中提取数据，节省了大量的IO操作

由于数据物理存储排序方式只能有一种，所以每个表中只能有一个聚簇索引**一般情况下该表的主键是聚簇索引**

为了充分利用聚簇索引特点，InnoDB表的主键尽量选择有序的ID，而不建议用无序的ID，如UUID这种

举例子

- 聚集索引类似于新华字典中**用拼音去查找汉字**，拼音检索表于书记顺序都是按照 a~z 排列的，就像相同的逻辑顺序于物理顺序一样，当你需要查找 a,ai 两个读音的字，或是想一次寻找多个傻(sha)的同音字时，也许向后翻几页，或紧接着下一行就得到结果了。

2) 非聚簇索引

- 非聚集索引类似在新华字典上**通过偏旁部首来查询汉字**，检索表也许是按照横、竖、撇来排列的，但是由于正文中是 a~z 的拼音顺序，所以就类似于逻辑地址于物理地址的不对应。同时适用的情况就在于分组，大数目的不同值，频繁更新的列中，这些情况即不适合聚集索引。

主键索引和唯一索引的区别？

- 主键索引不能重复且不能为空，唯一索引不能重复，可以为空
- 一张表只有一个主键索引，但是可以有多个唯一索引
- 主键索引的查询效率高于唯一 索引。

唯一索引和普通索引哪个性能更好？

对于查询操作来说：

- 普通索引和唯一索引性能相近都是从索引树中进行查询

对于更新操作来说

- 唯一索引要比普通索引慢，因为唯一索引需要把数据读取到内存中，在内存中对数据的唯一校验，所以执行起来慢。

InnoDB中为什么主键索引高于唯一索引

因为普通索引的查询多执行一次检索操作

比如 select * from t where id = 10; 只需要搜索id的这颗B+树。

普通索引 select * from t where tt = 5 会先查询索引树，得到id的值后去搜索id的B+树，因此多执行了一次检索，所以执行效率比主键索引低。

常见索引存储算法

(一) Hash 索引《Heap、Memory引擎支持》

1. hash 索引 仅仅能满足 “=”、“IN”、“<=>”查询，不能使用范围查询，因为经过相应的hash 算法**处理之后的 hash 值的大小关系并不能和 hash 运算前完全一样**
2. hash 索引**无法被用来避免数据排序操作**，因为hash值的大小关系并不一定和hash 运算前的键值完全一样。
3. hash 索引不能利用部分索引键查询，对于组合索引。hash 索引是在计算 hash 值的时候是将组合索引键合并后在一起计算hash 值，而不是单独计算hash 值，所以通过组合索引计算一个或几个索引键进行查询的时候，hash 索引也无法被利用。
4. **hash 索引在任何时候都不能避免全表扫描**，因为**不同索引存在相同的hash 值**，所以即使获取到某个hash 键值的数据，也无法从 hash 索引中直接完成查询，需要回表查询数据。
5. hash 索引遇到**大量的 hash 值相等时性能不一定B+树索引高**。

(二) 有序数组存储法

- 按顺序存储，优点是可以使用二分法快速找到数据，缺点是更新效率，适合静态数据存储

(三) 搜索树

- 以树的方式进行存储，查询性能好，更新速度快

(四) B-Tree索引

- B-Tree 索引关键字和数据都放在一起，叶子节点可以看作是外部节点，不包含任何信息
- 越靠近根节点查询速度越快

(五) B+Tree 索引

- InnoDB 引擎默认使用B+树索引。它会实时监控表上索引的情况，如果认为hash 索引可以提高效率，则自动在内从中**自适应哈希索引缓冲区**中建立哈希索引。InnoDB 默认开启自适应哈希索引。MySQL 会利用index key 的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个hash索引能够加快等值查询。
- B+树索引的关键字检索效率比较平均

为什么B+Tree 要比B-Tree 的性能好？

1. 因为B+树每个非叶子节点不存放数据，这样每个叶子节点容纳的元素比B-Tree多，树高比B-Tree小，这样带来的好处就是减少磁盘访问次数
2. 尽管B+树找到一个记录所需要的比较次数比B-Tree 多，但是一次磁盘访问的时间相当于成百上千次内存访问比较时间。
3. B+树叶子节点指针都是连在一起的，方便顺序遍历，这也是很多数据库和文件系统使用B+树的原因

什么是覆盖索引？

- 查询的数据列只从索引中获取，不必去读取数据行。MySQL 可以利用索引返回select 列表中的字段，而不必根据索引再次读取数据文件，**换句话说查询列要被所建的索引覆盖**
- 索引是高效找到行的一个方法，但是一般数据库也能使用索引找到一个列的数据，因此不必读取行。毕竟索引叶子节点存储了他们索引的数据；**当通过索引就可以读到想要的数据，就没有必要读取行了。**

- 一个索引包含了满足查询结果的数据就叫做覆盖索引

怎么验证 MySQL 的索引是否满足需求？

使用 explain 查看 SQL 是如何执行查询语句的，从而分析你的索引是否满足需求。

explain 语法：explain select * from table where type=1。

如何查询一张表的所有索引？

show index from t ;查询表 t 上面所有的索引

MySQL 最多可以创建多少个索引？

16个

MySQL 如何指定查询的索引？

在MySQL 中可以使用 force index 强行选择一个索引

```
select * from t force index(index_t);
```

在 MySQL 中指定了查询索引，为什么没有生效？

因为MySQL 会根据优化器自己选择索引，如果 force index 指定的索引出现在候选索引列上，这个时候MySQL 不会判断扫描的行数多少直接使用指定的索引，如果没在候选索引列上，即使 force index 指定了索引列也不会生效。

MySQL 会错选索引吗？

MySQL 会错选索引，比如 k 索引的速度更快，但是 MySQL 并没有使用而是采用了 v 索引，这种就叫错选索引，因为索引选择是 MySQL 的服务层的优化器来自动选择的，但它在复杂情况下也和人写程序一样出现缺陷。

如何解决 MySQL 错选索引的问题？

1. 删除错选的索引，只留下对的索引
2. 使用 force index 指定索引
3. 修改SQL查询语句引导MySQL 使用我们期望的索引

如何优化身份证索引？

在中国因为前 6 位代表的是地区，所以很多人的前六位都是相同的，如果我们使用前缀索引为 6 位的话，性能提升也并不是很明显，但如果设置的位数过长，那么占用的磁盘空间也越大，数据页能放下的索引值就越少，搜索效率也越低。针对这种情况优化方案有以下两种：

- 使用身份证倒序存储，这样设置前六位的意义就很大了；
- 使用 hash 值，新创建一个字段用于存储身份证的 hash 值。

前缀索引

- 前缀索引也叫局部索引，比如给身份证的前10位添加索引，类似这种给某列部分信息添加的方式叫做前缀索引

(一) 优点

- 有效减小索引文件的大小，让每个索引保存更多的索引值，从而提高索引的查询速度。

(二) 缺点

- 不能在 order by 或者 group by 中触发前缀索引，也不能把它们用于覆盖索引

(三) 什么时候使用前缀索引？

当字符串本身可能比较长，而且前几个字符就开始不同，适合使用

创建索引的时机？



索引优化口诀

全职匹配我最爱，最左前缀要遵守；

带头大哥不能死，中间兄弟不能断；

索引列上少计算，范围之后全失效；

LIKE 百分写最右，覆盖索引没有“*”；

不等空值还有 OR，索引影响要注意；

VAR 引号不可丢，SQL 优化有诀窍。

什么叫回表查询？

普通索引查询到主键索引后，回到主键索引树搜索的过程，称为回表查询

为什么MySQL 官方建议使用自增主键？

因为自增主键是连续的，在插入过程中尽量减少页分裂，即使进行页分裂，也会分裂很少一部分。并且自增主键也能减少数据的移动，每次插入都是插入到最后。所以自增主键性能是最高的。

自增主键优点

- 数据存储空间很小
- 性能最好
- 较少页分裂

缺点

- 数据量过大，可能会超出自增长取值范围
- 无法满足分布式存储，分库分表情况下无法合并表
- 主键有自增规律，容易被破解

♥ 事务

1. 说一下ACID是什么

Atomicity (原子性)

- 一个事务中的所有操作，要么全都执行，要么全都不执行

Consistency (一致性)

- 数据库在事务执行的前后都保持一致性状态，所有事务对一个数据的读取都是相同的

Isolation (隔离性)

- 一个事务所做的修改在提交前对其他事务是不可见的

Durability (持久性) dang'ji

- 一旦事务提交所做的修改将会永久保存在数据库中。

《只要满足一致性，事务的执行结果才是正确的》

丢失修改、脏读、不可重复读、幻读分别是什么？

丢失修改：T1和T2同时修改一个数据，T2覆盖了T1 修改的数据，导致了T1丢失修改。

脏读：T1修改一个数据，T2读取T1 的数据，此时T1撤销修改操作。T2读到了不存在的数据。

不可重复读：T2读到了T1的数据，T1对数据进行了修改。T2再次读取，则数据不一致

幻读：T1读取数据，T2插入数据，此时T1再次读取发现数据不一致，造成幻读。

如何避免幻读？

- 使用间隙锁的方式来避免幻读。间隙锁锁定了行与行之间的间隙，能够阻塞新插入的操作；间隙锁降低并发度，可能导致死锁。
- 使用MVCC 机制解决幻读

说一下MySQL的隔离级别

MySQL 的事务隔离是在 MySQL. ini 配置文件里添加的，在文件的最后添加：

```
transaction-isolation = REPEATABLE-READ
```

未提交读：事务中的修改即使没有提交，对其他事务也是可见《写数据只会锁住相应的行》

提交读：不能读取到未提交的数据

可重复读：数据多次读取到值保持一致 (**MySQL默认隔离级别**) 《写数据锁住整个表》

可串行化：强制事务串行执行《读写数据都会锁住整个表》

隔离级别	脏读	不可重复读	幻影读
未提交读	√	√	√
提交读	×	√	√
可重复读	×	×	√
可串行化	×	×	×

如何设置MySQL的事务隔离级别？

MySQL 事务隔离级别 MySQL.cnf 文件里设置的（默认目录 /etc/my.cnf），在文件的文末添加配置：

```
transaction-isolation = REPEATABLE-READ
```

InnoDB 默认的事务隔离级别是什么？如何修改？

可重复读

```
MySQL> set global transaction isolation level read committed; // 设置全局事务隔离级别为 read committed  
MySQL> set session transaction isolation level read committed; // 设置当前会话事务隔离级别为 read committed
```

如何手动操作事务？

使用 begin 开启事务； rollback 回滚事务； commit 提交事务。具体使用示例如下：

```
begin;  
insert person(uname,age) values('laowang',18);  
rollback;  
commit;
```

InnoDB 如何开启手动提交事务？

InnoDB 默认是自动提交事务的，每一次 SQL 操作（非 select 操作）都会自动提交一个事务，如果要手动开启事务需要设置 `set autocommit=0` 禁止自动提交事务，相当于开启手动提交事务。

并发事务有什么问题？如何解决？

- 加锁：在读取数据前，对其加锁，阻止其他事务对数据进行修改
- 使用MVCC（Multi-Version Concurrency Control）

MVCC 是如何工作的？

InnoDB的MVCC是通过在每行记录后面保存两个隐藏的列来实现。这两个列保存了行的创始时间和行的过期时间（删除时间）。当然存储的并不是真实的时间而是版本号。没开始一个新的事务，系统版本号自动递增，事务开始时刻系统的版本号作为事务的版本号，用来查询到的每行记录的版本号进行比较。

♡ 可重复读隔离级别下MVCC如何工作？

- SELECT：InnoDB 会根据以下条件检查每一行记录：第一，InnoDB 只查找版本早于当前事务版本的数据行，这样可以确保事务读取的行要么是在开始事务之前已经存在要么是事务自身插入或者修改过的。第二，行的删除版本号要么未定义，要么大于当前事务版本号，这样可以确保事务读取到的行在事务开始之前未被删除。
- INSERT：InnoDB 为新插入的每一行保存当前系统版本号作为行版本号。
- DELETE：InnoDB 为删除的每一行保存当前系统版本号作为行删除标识。
- UPDATE：InnoDB 为插入的一行新纪录保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为删除标识保存这两个版本号，使大多数操作都不用加锁。它不足之处是每行记录都需要额外的存储空间，需要做更多的行检查工作和一些额外的维护工作。

锁、并发

说一下 MySQL 的行锁和表锁？

MyISAM 只支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

- 表级锁：开销小，加锁快，不会出现死锁。**锁定粒度大，发生锁冲突的概率最高，并发量最低。**
- 行级锁：**开销大，加锁慢，会出现死锁。** 锁力度小，发生锁冲突的概率小，并发度最高。

说一下乐观锁和悲观锁？

(一) 悲观锁(Pessimistic Lock): ❤ Pessimistic (悲观的)

- 每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候**都会上锁**，这样别人想拿这个数据就会阻止，直到这个锁被释放。
- 通常所说的一锁二查三更新指的就是悲观锁。
- **使用select . . . for update 或 select . . . lock in share mode 实现悲观锁**

select for update

- 当数据库执行 select . . . for update 时会获取被 select 中的数据行的行锁，因此其他并发执行的 select . . . for update 如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。
- select . . . for update 获取的行锁会在事务的结束时自动释放，因此必须在事务中使用。
- select for update **所有扫描过的行都被锁上**，这一点很容易造成问题。如果在MySQL中使用悲观锁务必要确定使用索引，而不是全表扫描。

(二) 乐观锁(Optimistic Lock): ❤ Optimistic (乐观的)

- 每次去拿数据的时候都认为别人不会修改，所以不上锁，
- 但是在提交更新的时候会判断一下在此期间别人有没有去更新这个数据。

实现乐观锁

- 数据库的乐观锁需要自己实现，在表里面添加一个 version 字段。
- 当读数据时，将version 字段的值一同读出，数据更新一次，version 值加1。
- 当我们提交更新的时候，判断数据库对应记录的当前版本信息与第一次取出来的version 值进行比对。
- 如果数据库表当前版本号与第一次取出来的version 相等，则予以更新，否则认为是过期数据。

```
select version from t where id = #{id}; //先查到数据库中的version  
update t  
set value = 2, version = version +1  
where id = #{id} and version = #{version}; //更新时看是否和刚刚查询到的version相同，相同则更新成功，不同则更新失败
```

(三) 总结

悲观锁和乐观锁是数据库用来保证数据在并发状态下防止数据丢失的两种方法。两者有一定的区别如下几点：

- **响应速度**：如果需要非常高的响应速度，建议采用乐观锁，成功就执行，不成功就失败，不需要等待其他并发去释放锁
- **冲突频率**：如果发生冲突的频率非常高，建议使用悲观锁
- **重试代价**：如果重试代价大，建议采用悲观锁

乐观锁有什么缺点 (ABA) ?

- 缺点就是产生ABA 问题。ABA问题指的就是一个变量A 初次读取时是A 值，并且在准备赋值的时候检查它仍然是A 值，会误以为没有被修改过会正常执行修改操作，实际上这段时间它的值可能被改了其他值，之后再改回A值。

MySQL都有什么锁？

行锁 开销大、加锁慢、出现死锁。锁的粒度小，发生冲突的概率最低，并发度最高

表锁 开销小，加锁快。不会出现死锁。锁的粒度大，发生锁冲突的频率最高，并发度最低。

页面锁 开销、加锁时间和锁粒度位于表锁和行锁之间；会出现死锁；并发一般

select for update 和 lock in share mode的使用

1. 使用悲观锁。MySQL有两种实现方法

```
select ... for update  
select ... lock in share mode
```

select ... for update 实现方式

```
set autocommit = 0; //关闭自动提交  
begin; //开启事务  
select * from user where id = 1 for update; 查询信息  
update user set name = '张三'; //修改信息  
commit ; //提交事务
```

执行select ... for update 时，一般的 select 查询则不受影响

lock in share mode 实现方式

```
set autocommit = 0; //关闭自动提交  
begin; //开启事务  
select * from user where id =1 lock in share mode;//锁定查询的字段  
update user set name = '张三'; //修改信息  
commit ; //提交事务
```

二者区别

- 前者属于共享锁，允许其他事务添加共享锁，不允许其他事物修改或者加排他锁
- 后者属于排他锁，不允许其他事物添加共享锁和排他锁，也不允许修改

2. 使用乐观锁

采用带版本号（version）更新，相当于CAS思想

什么是全局锁？它的应用场景？

全局锁就是对整个数据库实例加锁，它的应用场景是全数据库逻辑备份。这个命令使整个库为只读状态。使用该命令之后，数据更新语句、数据定义语句、更新类事务的提交语句等操作都会被阻塞。

使用全局锁会导致什么问题？

如果在主库备份，在备份期间不能更新，业务停摆，所有更新业务处于等待状态。

如果在从库备份，在备份期间不能执行主库同步的bin log，导致主从延迟

什么是共享锁？

共享锁又称读锁（read lock），是读取操作创建的锁。允取其他用户并发读取数据，不允许任何事务对数据进行修改（获取数据的排他锁）。直到以释放所有共享锁。如果事务对读锁进行修改操作，很可能造成死锁。

什么是排他锁？

排他锁也叫写锁（write lock）。若某个事务对某一行加上了排他锁，只能这个事务对其进行读写，在此事务结束之前，其他事物不能对其加任何锁。其他进程可以读取，不能进行写操作，需要等待锁释放。

排他锁是悲观锁的一种实现。

如何设置数据库为全局只读锁？

flush tables with read lock; (FTWRL)

FTWRL 和 set global readonly=true 有什么区别？

FTWRL 当断开客户端后，整个数据库会取消只读。

set global readonly=true 会一直保持只读状态

如何实现表锁？

MySQL 里标记锁有两种：表级锁、元数据锁（meta data lock）简称 MDL。表锁的语法是 lock tables t read/write。可以用 unlock tables 主动释放锁，也可以在客户端断开的时候自动释放。lock tables 语法除了会限制别的线程的读写外，也限定了本线程接下来的操作对象。

对于 InnoDB 这种支持行锁的引擎，一般不使用 lock tables 命令来控制并发，毕竟锁住整个表的影响面还是太大。

MDL：不需要显式使用，在访问一个表的时候会被自动加上。

MDL 的作用：保证读写的正确性。

在对一个表做增删改查操作的时候，加 MDL 读锁；当要对表做结构变更操作的时候，加 MDL 写锁。

读锁之间不互斥，读写锁之间，写锁之间是互斥的，用来保证变更表结构操作的安全性。

MDL 会直到事务提交才会释放，在做表结构变更的时候，一定要小心不要导致锁住线上查询和更新。

InnoDB 存储引擎有几种锁算法？

- Record Lock 单个行记录上的锁
- Gap Lock 间隙锁，锁定一个范围，不包括记录本身
- Next-Key lock 锁定一个范围，包括本身

InnoDB 如何实现行锁？

行级锁是 MySQL 中粒度最小的一种锁，大大减少了数据库操作的冲突。

InnoDB 的行级锁有共享锁和排他锁两种。

- 共享锁允许事务读一行记录，不允许任何其他线程对该记录进行修改。
- 排他锁允许可当前事务删除、更新一条记录，其他线程不能操作该记录

共享锁和排他锁的区别？

共享锁： select。。。 lock in share mode； MySQL 会对结果集中每行添加共享锁，前提是当前线程中没有对该结果集中的任何行使用排他锁，否则申请阻塞

排他锁： select * from t where id = 1 for update；其中 id 字段必须有索引，MySQL 会对查询结果集中每行添加排他锁，在事务操作中，任何添加、删除都会添加排他锁。前提是当前没有线程对该结果集中的任何行使用排他锁或共享锁，否则申请阻塞

谈谈如何优化锁？

- 精心设计索引，尽量使用索引访问数据，使锁更加精确，从而减少锁冲突的机会
- 选择合理的事务大小，小事务发生的锁冲突的概率最小
- 给结果集加锁时，一次性的请求足够级别的锁。比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时在获取排他锁，这样容易产生死锁。
- 不同的程序访问一组表时，应尽量约定以相同的顺序访问。对于一个表而言，尽可能以固定的顺序存取表中的行，这样大大降低死锁的机会
- 尽量使用相等条件访问数据，这样可以避免间隙锁对并发插入的影响
- 不要申请超过实际需要的锁级别
- 除非必须，不要加锁，MySQL 的MVCC 可以实现事务的查询不用加锁，优化事务性能。MVCC只在读提交 (committed read) 和可重复读 (repeatable read) 两种隔离级别中使用
- 对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能。

死锁

什么情况下会造成死锁？

死锁：两个或两个以上的进程在执行过程中，因争夺资源而造成一种互相等待的现象，若人为不干预，他们将无法推进下去。

表级锁不会发生死锁，所以**死锁只要针对的是InnoDB引擎**

死锁的解决办法？

(一) 查出线程杀死

```
SELECT trx_MySQL_thread_id FROM information_schema.INNODB_TRX;
```

(二) 设置锁的超时时间

```
set innodb_lock_wait_timeout=1000; --设置当前会话 InnoDB 行锁等待超时时间，单位秒。
```

如何查看死锁？

```
show engine innodb status;//查看最近一次死锁
```

InnoDB Lock Monitor 打开锁监控，每隔 15s 输出一次日志。使用完毕后建议关闭，否则影响数据库性能。

如何避免死锁？

- 为单个InnoDB 表执行多个并发写入时造成死锁；可以在事务开始时使用 select ... for update 语句获取锁，即使这些语句是在之后才执行的
- 如果要更新数据时，应该使用排他锁。
- 如果事务需要修改或锁定多个表，则应在每个事务中以相同的顺序使用加锁语句。在应用中，如果不同的程序会并发存取多个表，应尽量约定以相同顺序访问表，这样可以大大降低产生死锁的机会
- 通过 select ... lock in share mode 获取读锁后，如果当前事务需要对该数据进行更新操作，则很有可能造成死锁。
- 改变事务隔离级别

InnoDB是如何对待死锁的？

默认使用设置死锁超时时间来让死锁超时的策略，默认innodb_lock_wait_timeout 设置为 50s。

如何开启死锁检测？

设置 innodbdeadlockdetect 设置为 on 可以主动检测死锁，在 Innodb 中这个值默认就是 on 开启的状态。

♡ MySQL高并发解决方案

需求分析：互联网每天大量数据读取、写入、并发性高

解决方式：

- 水平分库分表，由单点分配到多点数据库上，从而降低单点数据库压力
- 集群方案 解决DB宕机带来的单点数据库服务不可用
- 读写分离策略：极大提高了读取数据的并发量

聊聊数据库崩溃时事务的恢复机制？

(一) Undo log

Undo log 为了实现事务的原子性，在MySQL 数据库InnoDB 引擎中来实现MVCC。

- 事务的原子性：要不全部完成，要不全部未完成。
- Undo log 为了保证原子性，在操作任何事务之前，先将数据备份到Undo log 上面，如果发生回滚，那么系统可以利用 Undo log 将数据恢复到事务之前的状态。

之所以能保证原子性，主要是因为如下特点：

- 更新记录前记录Undo log
- 为了保持持久性，必须先将数据的事务提交写到磁盘，只要事务成功提交，数据必然已经持久化
- Undo log 先于数据持久化到磁盘，如果系统崩溃，Undo log 是完整的，可以用来回滚事务

缺陷：每次提交事务之前都要将数据和Undo log 写入到磁盘，导致大量的IO，性能低。

(二) Redo Log

Redo log 记录的是新数据的备份，在事务提交前只要将redo log 持久化即可。不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是 redo log 已经持久化。系统可根据 redo log 的内容将所有数据恢复到事务开启前的状态。

♡ MySQL基础

1.如何获取当前数据库版本？

使用 select version() 获取当前 MySQL 数据库版本。

如何查看 MySQL 的空闲连接？

在 MySQL 的命令行中使用 `show processlist;` 查看所有连接，其中 Command 列显示为 Sleep 的表示空闲连接，如下图所示：

mysql> show processlist;								
Id	User	Host	db	Command	Time	State	Info	
4	event_scheduler	localhost	NULL	Daemon	25930	Waiting on empty queue	NULL	
12	root	localhost	NULL	Query	0	starting	show processlist	
13	root	222.90.210.42:29695	NULL	Sleep	6		NULL	

3 rows in set (0.00 sec)

2.char 和 varchar 的区别是什么？

(一) char(n) :

- 存储方式：对英文字符（ASCII）占用一个字节，对一个汉字占用2个字节

- 固定长度类型，比如订阅 char(10)，当你输入"abc"三个字符的时候，它们占的空间还是 10 个字节，其他 7 个是空字节。
- **优点：效率高；**
- 缺点：占用空间；
- 适用场景：存储密码的 md5 值，固定长度的，使用 char 非常合适。

(二) varchar(n) :

- 存数方式：对每个引文占用2个字节，汉字也是2个字节
- 可变长度，存储的值是每个值占用的字节再加上一个**用来记录其长度的字节的长度**。
- **所以，从空间上考虑 varcahr 比较合适；**
- 从效率上考虑 char 比较合适，二者使用需要权衡。

3.float 和 double 的区别是什么？

- float 最多可以存储 8 位的十进制数，并在内存中占 4 字节。
- double 最多可以存储 16 位的十进制数，并在内存中占 8 字节。

count(column) 和 count(*) 有什么区别？

count 在 InnoDB 中是一行一行读取，然后累计计数的。

统计结果不一致。

count (column) 不会统计列值为空的数据

count (*) 统计所有信息

4.什么是内连接、左连接、右连接有什么区别？

内连接

内连接关键字：inner join；左连接：left join；右连接：right join。

内连接是把匹配的关联数据显示出来；左连接是左边的表全部显示出来，右边的表显示出符合条件的数据；右连接正好相反。

什么是临时表？临时表什么时候删除？

临时表可以手动删除 ❤ temporary (临时表)

```
DROP TEMPORARY TABLE IF EXISTS temp_tb
```

临时表只在当前链接可见，当关闭连接时，MySQL会自动删除临时表并释放空间

```
CREATE TEMPORARY TABLE tmp_table (
    NAME VARCHAR (10) NOT NULL ,
    time date NOT NULL
);

select * from tmp_table;
```

创建数据库

```
create database 数据库名称;
show databases; //查看数据库
```

修改数据库

```
alter database 数据库名称 default character set 编码方式 collate 编码方式_bin;
eg: alter database tb_user default character set utf8 collate utf8_bin;
```

删除数据库

```
drop database 数据库名;
```

数据类型

度,D表示的是小数点后的长度。比如,插入一个浮点数 3.14

插入数据库后,显示的结果为 3.14。

timestamp

2.2.3 日期与时间类型

year date time datetime

为了方便在数据库中存储日期和时间,MySQL 提供了表示日期和时间的数据类型,分别是 YEAR、DATE、TIME、DATETIME 和 TIMESTAMP。表 2-3 列举了这些 MySQL 中日期和时间数据类型所对应的字节数、取值范围、日期格式以及零值。

表 2-3 MySQL 日期和时间类型

数据类型	字节数	取值范围	日期格式	零值
YEAR	1	1901~2155	YYYY	0000
DATE	4	1000-01-01~9999-12-31	YYYY-MM-DD	0000-00-00
TIME	3	-838:59:59~838:59:59	HH:MM:SS	00:00:00
DATETIME	8	1000-01-01 00:00:00~9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	0000-00-00 00:00:00
TIMESTAMP	4	1970-01-01 00:00:01~2038-01-19 03:14:07	YYYY-MM-DD HH:MM:SS	0000-00-00 00:00:00

从表 2-3 中可以看出,每种日期和时间类型的取值范围都是不同的。需要注意的是,如果插入的数值不合法,系统会自动将对应的零值插入数据库中。

创建数据表

```
create table 表名
{
    字段 1, 数据类型,
    字段 2, 数据类型,
}
```

内连接、外连接

内连接:又称简单连接,或自然连接,内连接使用比较运算符对两个表中的数据进行比较,并列出与连接条件匹配的数据行,组合成新的记录,也就是说在内连接查询中,只有满足条件的记录才能出现在查询结果中

```
select 查询字段 from 表1 [INNER] join 表2 on 表1.关系字段=表2.关系字段
```

eg:

```
select employee.name,department.dname  
from department join employee on department.did=employee.did;
```

使用 where 语句

```
select employee.name,department.dname  
from department,employee  
where department.did=employee.did;
```

自连接

如果在一个连接查询中，涉及的两个表是同一个表，这种查询称为**自连接**查询。

自连接是一种特殊的内连接，它是指相互连接的表在物理上为同一个表，但逻辑上分为两个表，

例如要查询王红所在部门有哪些员工，就可以使用自连接查询。

```
select p1.* from employee p1 join employee p2 on p1.did=p2.did where p2.name='王红';
```

外连接

```
select 所查字段  
from 表 1 left|right [outer] join 表 2 on 表 1.关系字段=表 2.关系字段  
where 条件
```

MySQL 什么情况下导致自增主键不连续？

唯一主键冲突、事务回滚

InnoDB 中自增主键能不能被持久化？

MySQL8之前不可以， MySQL 8 之后会把自增主键保存到 redo log 中，当MySQL 重启会从redo log 中恢复。

独立表空间和共享表空间

共享表空间：数据库的所有表数据、索引文件全部放在一个文件中，默认这个共享表空间在文件路径 data 目录下。

独立表空间 每一个表都会以独立的文件方式进行存储

二者区别是，如果把表放进共享区间，即使表删除了，空间也不会删除，所以表依然很大，而独立表空间如果删除表就会清除空间

如何设置独立表空间？

独立表空间是由参数 innodbfileper_table 控制的，把它设置成 ON 就是独立表空间了，从 MySQL 5.6.6 版本之后，这个值就默认是 ON 了。

如何进行表空间压缩？

使用重建表的方式可以收缩表空间

```
alter table t engine = innodb  
  
optimize table t  
  
truncate table t
```

说一下重建表的执行流程？

1. 建立一个临时文件，扫描表 t 主键的所有数据页
2. 用数据页中表 t 的记录生成 B+Tree，存储到临时文件中
3. 生成临时文件的过程中，将所有对 t 的操作记录在一个日志文件（row log）中
4. 临时文件生成后，将日志文件中的操作应用到临时文件中，得到一个逻辑数据上与表 t 相同的数据文件
5. 用临时文件替换表 t 的数据文件。

表的结构存储在哪里？

表的结构定义占用的存储空间比较小，在MySQL 8 之前，表结构存储在 .frm 为后缀的文件里，MySQL 8 之后，允许把表结构的定义信息存在系统数据表中

如果把一个 InnoDB 表的主键删掉，是不是就没有主键，就没办法进行回表查询了？

可以回表查询，如果把主键删掉了，那么 InnoDB 会自己生成一个长度为 6 字节的 rowid 作为主键。

执行一个 update 语句以后，我再去执行 hexdump 命令直接查看 ibd 文件内容，为什么没有看到数据有改变呢？

可能是因为 update 语句执行完成后，InnoDB 只保证写完了 redo log、内存，可能还没来得及将数据写到磁盘。

MySQL 支持枚举吗？如何实现？它的用途是什么？

MySQL 支持枚举，它的实现方式如下：

```
create table t(
    sex enum('boy','girl') default 'unknown'
);
```

枚举的作用是预定义结果值，当插入数据不在枚举值范围内，则插入失败，提示错误 Data truncated for column 'xxx' at row n。

InnoDB 和 MyISAM 执行 select count(*) from t，哪个效率更高？为什么？

MyISAM 效率最高，因为 MyISAM 内部维护了一个计数器，直接返回总条数，而 InnoDB 要逐行统计。

在 MySQL 中有对 count(*) 做优化吗？做了哪些优化？

count(*) 在不同的 MySQL 引擎中的实现方式是不相同的，在没有 where 条件的情况下：

- MyISAM 引擎会把表的总行数存储在磁盘上，因此在执行 count(*) 的时候会直接返回这个行数，执行效率很高；
- InnoDB 引擎中 count(*) 就比较麻烦了，需要把数据一行一行的从引擎中读出来，然后累计基数。

但即使这样，在 InnoDB 中，MySQL 还是做了优化的，我们知道对于 count() 这样的操作，遍历任意索引树得到的结果，在逻辑上都是一样的，因此，MySQL 优化器会找到最小的那颗索引树来遍历，这样就能在保证逻辑正确的前提下，**尽量少扫描数据量，从而优化了 count() 的执行效率**。

在 InnoDB 引擎中 count(*)、count(1)、count(主键)、count(字段) 哪个性能最高？

count(字段) < count(主键 id) < count(1) ≈ count(*) 题目解析：

- 对于 count(主键 id) 来说，InnoDB 引擎会遍历整张表，把每一行的 id 值都取出来，返回给 server 层。server 层拿到 id 后，判断是不可能为空的，就按行累加。
- 对于 count(1) 来说，InnoDB 引擎遍历整张表，但不取值。server 层对于返回的每一行，放一个数字“1”进去，判断是不可能为空的，按行累加。
- 对于 count(字段) 来说，如果这个“字段”是定义为 not null 的话，一行行地从记录里面读出这个字段，判断不能为 null，按行累加；如果这个“字段”定义允许为 null，那么执行的时候，判断到有可能是 null，还要把值取出来再判断一下，不是 null 才累加。

- 对于 count(*) 来说，并不会把全部字段取出来，而是专门做了优化，不取值，直接按行累加。

使用 delete 误删数据怎么找回？

可以用 Flashback 工具通过闪回把数据恢复回来。

delete 和 truncate 区别？

- delete 后面可以跟where 子句，通过指定的条件表达式删除满足条件的部分记录，truncate 则删除全部记录。
- delete 删除表中数据后，再次像表中添加记录时，自动增长的值为删除时该字段的最大值 + 1；而 truncate 自动增加的默认初始值重新从1开始；
- delete 删除会记录日志信息，truncate 不会记录日志信息
- truncate 执行效率比delete 快

Flashback 恢复数据的原理是什么？

Flashback 恢复数据的原理是修改 binlog 的内容，拿回原库重放，从而实现数据找回。

MySQL 数据表的基本操作

查看数据表

```
show create table 表名;
describe 表名; //或者简写desc 表名
```

修改数据表

```
alter table 旧表名 rename [TO] 新表名;
```

修改字段名

```
alter table change 表名 旧字段名 新字段名 新数据类型
eg: alter table change tb_user name user_name varchar(20);
```

修改字段的数据类型

```
alter table 表名 modify 字段名 数据类型;
eg:alter table tb_user modify user_name int(20);
```

添加字段

```
alter table 表名 add 新字段名 数据类型[约束条件][first|after 已存在字段名]
eg: alter table tb_user add age int(10);
```

删除字段

```
alter table 表名 drop 字段名;
```

修改字段的排列位置

```
alter table 表名 modify 字段名 1 数据类型 first|after 字段名 2 ;
```

删除数据表

```
drop table 表名;
```

表的约束

单字段主键

```
字段名 数据类型 primary key  
eg: create table tb_test(id int primary key,name varchar(20),grade float);
```

多字段主键

```
primary key(字段名 1,字段名 2, ...字段名 n)  
eg:  
create table tb_test(  
stu_id int,  
course_id int,  
name varchar(20),  
grade float,  
primary key(stu_id,course_id)  
);
```

唯一约束

```
字段名 数据类型 unique;  
eg:  
create table tb_test1(  
id int primary key,  
stu_id int unique,  
name varchar(20) not null  
);
```

默认约束

```
字段名 数据类型 default 默认值;  
eg:  
create table tb_test2(  
id int primary key auto_increment,  
stu_id int unique,  
grade float default 0  
);
```

索引

普通索引

- 普通索引是由KEY或INDEX定义的索引，他是MySQL中的基本索引类型，可以创建在任何数据类型中，

唯一性索引

- 唯一性索引由unique定义的索引，该索引所在字段的值必须是唯一的。

全文索引

- 只有MyISAM存储引擎支持全文索引

创建索引

1. 创建表的时候创建索引

```
create table 表名 (字段名 数据类型[完整性约束条件],  
                    字段名 数据类型[完整性约束条件],  
                    ...  
                    字段名 数据类型[UNIQUE|FULLTEXT|SPATIAL] INDEX|KEY[别名] (字段名 1[长度]) [ASC|DESC] )  
  
eg:  
create table tb_index1(  
    id int primary key auto_increment,  
    stu_id int unique,  
    grade float default 0,  
    index iddd (id)  
);
```

2. 在已经存在的表上创造索引,使用 create index

```
create [UNIQUE|FULLTEXT|SPATIAL] index 索引名 on 表名 (字段名 [(长度)] [ASC|DESC])  
  
eg:  
create unique index unique_idx on tb_index1(id);
```

3. 在已经存在的表上创造索引,使用 alter table

```
alter table 表名 add [UNIQUE|FULLTEXT|SPATIAL] index 索引名 (字段名 [(长度)] [ASC|DESC])
```

删除索引

1. 使用 alter table 删除索引

```
alter table 表名 drop index 索引名  
  
eg:  
alter table tb_index1 drop index unique_idx;
```

2. 使用 drop index 删除索引

```
drop index 索引名 on 表名 ;  
  
eg:  
drop index unique_idx on tb_index1;
```

CRUD

1. 添加, 指定字段名

```
insert into 表名(字段名 1 , 字段名 2, ...) values(值 1, 值 2, ...);
```

添加, 不指定字段名

```
insert into 表名 values(值 1, 值 2, ...); //添加顺序必须和表中定义的字段顺序一样
```

其他用法

```
insert into 表名 set 字段名 1 =值 1[, 字段名 2 =值 2, ...];
eg:
insert into tb_user set name ='李四', age= 18;
```

添加多条记录

```
insert into 表名[(字段名 1 , 字段名 2, ...)] values
(值 1, 值 2, ...),
...
(值 1, 值 2, ...),
```

eg:

```
insert into tb_user values
('王五',15),
('李四',19),
('赵二麻子',21);
```

2. 更新数据

```
update 表名 set 字段名 1= 值 1[,字段名 2=值 2, ...][where 条件表达式];//
update 表名 set 字段名 1= 值 1[,字段名 2=值 2, ...];//更新全部数据
```

3. 删除数据

```
delete from 表名 [where 条件表达式];
delete from 表名 ;// 删除所有数据
```

truncate

```
truncate [table] 表名; //删除所有数据

eg: truncate table tb_uesr;
```

delete 和truncate 区别?

- delete 后面可以跟where 子句，通过指定的条件表达式删除满足条件的部分记录， truncate 则删除全部记录。
- delete 删除表中数据后，再次像表中添加记录时，自动增长的值为删除时该字段的最大值 + 1；而 truncate 自动增加的默认初始值重新从1开始；

单表查询

select 语句

```
select [distinct] * | {字段名 1, 字段名 2, 字段名 3, ...}
from 表名
[where 条件表达式 1]
[group by 字段名 [having 条件表达式 2]]
[order by 字段名 [ASC | DESC]]
[limit [offset] 记录数]
```

IN 关键词

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 字段名 [NOT] IN (元素 1, 元素 2, ...)
```

eg:查询student 表中id 为 1, 2, 3的记录
select * from student where id in(1,2,3);

between and

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 字段名 [NOT] between 值 1 and 值 2
区间: [a,b] 包括a 和b
```

空值:某些列可能为空值(NULL), 空值不同于0, 也不同于空字符串

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 字段名 IS [NOT] NULL
```

distinct: 去重

```
select distinct gender from student;
```

☆当**distinct** 关键字作用于多个字段

```
select distinct gender ,name from student;
```

只有distinct 关键字后指定的多个字段值都相同，才会被认做是重复记录

like

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 字段名 [NOT] LIKE '匹配字符串'
```

%: 任意长度

_: 单个字符

通配符: "\%" --> %

AND

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 条件表达式 1 AND 条件表达式 2 [...条件表达式 n]
```

OR

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
  from 表名
  where 条件表达式 1 OR 条件表达式 2 [...条件表达式 n]
```

OR 和AND 同时使用注意优先级

AND 的优先级高于 OR

```
select * from student where gender ='女' OR gender ='男' AND grade =100;
```

先执行gender ='男' AND grade =100 满足这个条件或者 gender ='女' 这个条件则输出

聚合查询

COUNT()

SUM()

AVG()

MAX()

MIN()

```
select count(*) from student;
```

order by:默认升序 ASC

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
from 表名
order by 字段名 1[ASC | DESC], 字段名 2[ASC | DESC]
```

eg:

```
select * from student order by grade;
```

group by: 分组查询

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
from 表名
group by 字段名 1,字段名 2,... [having 条件表达式]
```

eg:

```
mysql> select *from student group by gender;
+----+-----+-----+
| id | name  | grade | gender |
+----+-----+-----+
| 10 | 雍正 |    33 | NULL   |
|  2 | 李四 |    45 | 女     |
|  1 | 王五 |    23 | 男     |
+----+-----+-----+
3 rows in set (0.00 sec)
```

group by 和聚合函数一起使用

```
mysql> select count(*),gender from student group by gender;
+-----+-----+
| count(*) | gender |
+-----+-----+
|      1 | NULL   |
|      4 | 女     |
|      5 | 男     |
+-----+-----+
3 rows in set (0.00 sec)
```

group 和having 一起使用

```
mysql> select sum(grade),gender from student group by gender having sum(grade)<300;
+-----+-----+
| sum(grade) | gender |
+-----+-----+
|      33 | NULL   |
|    159 | 女     |
|     80 | 男     |
+-----+-----+
3 rows in set (0.00 sec)
```

limit限制查询的数量

```
select * | 字段名 1, 字段名 2, 字段名 3, ...
from 表名
limit [offset,] 记录数
```

offset表示偏移量，默认为0；

如果偏移量为0则从查询的第一条记录开始

偏移量为1 则表示从查询结果中的第二条记录开始，以此类推

取别名

```
select * from 表名 [AS] 别名;
```

多表操作

外键：为了保证数据的完整性，将两表之间的数据建立关系。

创建外键

```
alter table 表名 add constraint FK_ID foreign key(外键字段名) references 外键表名 (主键字段名);
```

eg

```
mysql> alter table student add constraint FK_ID foreign key(gid) references grade (id);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

查看创建的外键

```
mysql> show create table student;
+ Table    | Create Table
+ student | CREATE TABLE `student` (
  `sid` int(4) NOT NULL,
  `sname` varchar(36) DEFAULT NULL,
  `gid` int(4) NOT NULL,
  PRIMARY KEY (`sid`),
  KEY `FK_ID` (`gid`),
  CONSTRAINT `FK_ID` FOREIGN KEY (`gid`) REFERENCES `grade` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

注意：

- 建立外建的表必须是InnoDB引擎，不能是临时表。MySQL中只有InnoDB支持外键
- 定义外键名时，不能加引号，如 constraint 'FK_ID' 或constraint "FK_ID"都是错误的；

删除外键约束

```
alter table 表名 drop foreign key 外键名
```

eg:

```
mysql> alter table student drop foreign key FK_IDS;
```

操作关联表

添加数据

student

```
mysql> insert into student values(1,'周润发',1),  
(2,'刘德华',1),(3,'周润发',2),(4,'刘德华',2);
```

grade

```
mysql> insert into grade values(1,'软件一班'),(2,'软件二班');
```

```
mysql> select * from grade;
```

id	name
1	软件一班
2	软件二班

2 rows in set (0.00 sec)

```
mysql> select * from student;
```

sid	sname	gid
1	周润发	1
2	刘德华	1
3	周润发	2
4	刘德华	2

4 rows in set (0.00 sec)

注意:

- 在上述语句中，添加的主键ID为1，2，由于student表的外键和grade表的主键关联，因此在为student添加数据的时候，gid只能是1或2

删除数据

- 具有关系的表中删除数据时，一定要先删除从表中的数据，然后在删除主表中的数据。

子查询

子查询：指一个查询语句嵌套在另一个查询语句内部的查询。在执行查询语句时，首先执行子查询中的语句，然后将返回的结果作为外层查询的过滤条件，在子查询中通常可以使用IN、exists、any、all操作符。

带IN关键字的子查询：使用IN关键字进行子查询时，内层查询语句仅返回一个数据列，这个数据列的值将供外层语句进行比较操作

查询存在年龄为20岁的员工的部门

```
select * from department where id in(select did from employee where age=20);
```

带 EXISTS 关键字的子查询：EXISTS 关键字后面的参数可以是任意一个子查询，这个子查询的作用相当于测试，他不产生任何数据，只返回TRUE 或FALSE，当返回值为TRUE 时，外层查询才会执行。

EXISTS 关键字比 IN 关键字的运行效率高：因为 EXISTS 允许用户使用相关子查询以排除一个表能够与另一个表成功连接的所有记录。 EXISTS 用到了连接，能够发挥已建好索引的作用，IN 不能使用索引

带any 关键字的子查询：any 关键字表示，满足其中任意一个条件，它允许创建一个表达式对子查询的返回值列表进行比较，只要满足内层子查询的任意一个比较条件，就返回一个结果作为外层查询条件

```
select * from department where did > any (select did from employee);
did dname
2 媒体部
3 研发部
5 人事部
```

首先子查询会将employee 表中的所有did 查寻出来，分别为1、1、2、4，然后将department 表中did 的值与之进行比较，只要大于employee.did 中的任意一个值，就是符合条件的查询结果，由于department 表中的媒体部，研发部，人事部 的did都大于employee 表中的did (did=1)，因此输出结果为媒体部、研发部、和人事部

带 ALL 关键字的子查询：ALL 关键字的子查询返回的结果需要同时满足所有的内层查询条件

```
select * from department where did > all(select did from employee);
```

带比较运算符的子查询 如< >= 、 = 、 !=

```
select * from department where did = (select did from employee where name ='赵四');
```

什么是页？

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。主存和磁盘以页为单位交换数据。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次磁盘 IO 就可以完全载入。

日志

redo log 和 binlog 有什么区别？

redo log (重做日志) 和 binlog (归档日志) 都是 MySQL 的重要的日志，它们的区别如下：

- redo log 是物理日志，记录的是“在某个数据页上做了什么修改”。
- binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这一行的 c 字段加 1 ”。
- redo log 是 InnoDB 引擎特有的；binlog 是 MySQL 的 Server 层实现的，所有引擎都可以使用。
- redo log 是循环写的，**空间固定会用完**；**binlog 是可以追加写入的**。“追加写”是指 binlog 文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

最开始 MySQL 里并没有 InnoDB 引擎，MySQL 自带的引擎是 MyISAM，但是 MyISAM 没有 crash-safe 的能力，binlog 日志只能用于归档。而 InnoDB 是另一个公司以插件形式引入 MySQL 的，既然只依靠 binlog 是没有 crash-safe 能力的，所以 InnoDB 使用另外一套日志系统，也就是 redo log 来实现 crash-safe 能力。

什么是 crash-safe？

crash-safe 是指发生宕机等意外情况下，服务器重启后数据依然不会丢失的情况。

什么是脏页和干净页？

MySQL 为了操作的性能优化，会把数据更新**先放入内存中，之后再统一更新到磁盘**。当内存数据和磁盘数据内容不一致的时候，我们称这个内存页为脏页；内存数据写到磁盘后，内存的数据和磁盘上的内容就一致了，我们称为“干净页”。

MySQL 刷脏页的速度很慢可能是什么原因？

在 MySQL 中单独刷一个脏页的速度是很快的，如果发现刷脏页的速度很慢，说明触发了 MySQL 刷脏页的“连坐”机制，MySQL 的“连坐”机制是指当 MySQL 刷脏页的时候如果发现相邻的数据页也是脏页也会一起刷掉，而这个动作可以一直蔓延下去，这就是导致 MySQL 刷脏页慢的原因了。

如何控制 MySQL 只刷新当前脏页？

在 InnoDB 中设置 `innodbflushneighbors` 这个参数的值为 0，来规定 MySQL 只刷当前脏页，MySQL 8 这个值默认是 0。

MySQL 的 WAL 技术是解决什么问题的？

A.防止误删除，找回数据用的 B.容灾恢复，为了还原异常数据用的 C.事务处理，为了数据库的稳定性 D.为了降低 IO 成本
答：D

题目解析：WAL 技术的全称是 Write Ahead Logging（中文：预写式日志），是先写日志，再写磁盘的方式，因为每次更新都写磁盘的话 IO 成本很高，所以才有了 WAL 技术。

为什么有时候会感觉 MySQL 偶尔卡一下？

如果偶尔感觉 MySQL 卡一下，可能是 MySQL 正在刷脏页，正在把内存中的更新操作刷到磁盘中。

redo log 和 binlog 是怎么关联的？

它们有一个共同的数据字段，叫 XID。崩溃恢复的时候，会按顺序扫描 redo log：

- 如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；
- 如果碰到只有 prepare、而没有 commit 的 redo log，就拿着 XID 去 binlog 找对应的事务。

MySQL 怎么知道 binlog 是完整的？

- statement 格式的 binlog，完整的标识是最后有 COMMIT 关键字。
- row 格式的 binlog，完整的标识是最后会有一个 XID event 关键字。

MySQL 中可不可以只要 binlog，不要 redo log？

不可以，binlog 没有崩溃恢复的能力。

MySQL 中可不可以只要 redo log，不要 binlog？

不可以，原因有以下两个：

- redo log 是循环写不能保证所有的历史数据，这些历史数据只能在 binlog 中找到；
- binlog 是高可用的基础，高可用的实现原理就是 binlog 复制。

为什么 binlog cache 是每个线程自己维护的，而 redo log buffer 是全局共用的？

因为 binlog 是不能“被打断的”，一个事务的 binlog 必须连续写，因此要整个事务完成后，再一起写到文件里。而 redo log 并没有这个要求，中间有生成的日志可以写到 redo log buffer 中，redo log buffer 中的内容还能“搭便车”，其他事务提交的时候可以被一起写到磁盘中。

事务执行期间，还未提交，如果发生 crash，redo log 丢失，会导致主备不一致呢？

不会，因为这时候 binlog 也还在 binlog cache 里，没发给备库，crash 以后 redo log 和 binlog 都没有了，从业务角度看这个事务也没有提交，所以数据是一致的。

在 MySQL 中用什么机制来优化随机读/写磁盘对 IO 的消耗？

redo log 是用来节省随机写磁盘的 IO 消耗，而 change buffer 主要是节省随机读磁盘的 IO 消耗。redo log 会把 MySQL 的更新操作先记录到内存中，之后再统一更新到磁盘，而 change buffer 也是把关键查询数据先加载到内存中，以便优化 MySQL 的查询。

redo log 和 bin log 面试题

- A.redo log 是 InnoDB 引擎特有的，它的固定大小的
- B.redo log 日志是不全的，只有最新的一些日志，这和它的内存大小有关
- C.redo log 可以保证数据库异常重启之后，数据不丢失
- D.binlog 是 MySQL 自带的日志，但它并不能保证数据库异常重启之后数据不丢失。
- E.binlog 日志是追加写的，后面的日志并不会覆盖前面的日志

有没有办法把 MySQL 的数据恢复到过去某个指定的时间节点？怎么恢复？

可以恢复，只要你备份了这段时间的所有 binlog，同时做了全量数据库的定期备份，比如，一天一备，或者三天一备，这取决于你们的备份策略，这个时候你就可以把之前备份的数据库先还原到测试库，从备份的时间点开始，将备份的 binlog 依次取出来，重放到你要恢复数据的那个时刻，这个时候就完成了数据到指定节点的恢复。比如，今天早上 9 点的时候，你想把数据恢复成今天早上 6:00:00 的状态，这个时候你可以先取出今天凌晨（00:01:59）备份的数据库文件，还原到测试库，再从 binlog 文件中依次取出 00:01:59 之后的操作信息，重放到 6:00:00 这个时刻，这就完成了数据库的还原。

♡ MySQL 性能优化

性能指标

① **TPS (Transaction Per Second)** 每秒事务数，即数据库每秒执行的事务数。

② **QPS (Query Per Second)** 每秒请求次数，也就是数据库每秒执行的 SQL 数量，包含 INSERT、SELECT、UPDATE、DELETE 等。

③ **IOPS (Input/Output Operations per Second)** 每秒处理的 I/O 请求次数。

Mysql 优化举例

1. 当只要一行数据时，使用 limit
2. 选择正确的存储引擎： MyISAM 适合大量查询的请求，使用表锁，功能单一； InnoDB 适合写请求多的业务，支持事务，行锁
3. Not Exist 代替 Not In： **EXISTS 关键字比 IN 关键字的运行效率高**：因为 EXISTS 允许用户使用相关子查询以排除一个表能够与另一个表成功连接的所有记录。 **EXISTS 用到了连接**，能够发挥已建好索引的作用，**IN 不能使用索引**
4. 对操作符的优化，尽量不采用不利用索引的操作符
5. Mysql 分库分表：**垂直切分和水平切分**
6. 避免使用 select *，列出需要查询的字段。
7. 为搜索字段创建索引。

♡ MySQL 的常见的优化手段有以下五种：

① 查询优化

- 避免 SELECT *，只查询需要的字段。
- 小表驱动大表，即小的数据集驱动大的数据集，比如，当 B 表的数据集小于 A 表时，用 in 优化 exist，两表执行顺序是先查 B 表，再查 A 表，查询语句：select * from A where id in (select id from B)。
- 一些情况下，可以使用连接代替子查询，因为使用 join 时，MySQL 不会在内存中创建临时表。

② 优化索引的使用

- 尽量使用主键查询，而非其他索引，因为主键查询不会触发回表查询。
- 不做列运算，把计算都放入各个业务系统实现
- 查询语句尽可能简单，大语句拆小语句，减少锁时间
- 不使用 select * 查询
- or 查询改写成 in 查询
- 不用函数和触发器
- 避免 %xx 查询
- 少用 join 查询
- 使用同类型比较，比如 '123' 和 '123'、123 和 123
- 尽量避免在 where 子句中使用 != 或者 <> 操作符，查询引用会放弃索引而进行全表扫描
- 列表数据使用分页查询，每页数据量不要太大
- 用 exists 替代 in 查询
- 避免在索引列上使用 is null 和 is not null
- 尽量使用主键查询
- 避免在 where 子句中对字段进行表达式操作
- 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型

③ 表结构设计优化

- 使用可以存下数据最小的数据类型。
- 使用简单的数据类型，int 要比 varchar 类型在 MySQL 处理简单。
- 尽量使用 tinyint、smallint、mediumint 作为整数类型而非 int。
- 尽可能使用 not null 定义字段，因为 null 占用 4 字节空间。
- 尽量少用 text 类型，非用不可时最好考虑分表。
- 尽量使用 timestamp，而非 datetime。
- 单表不要有太多字段，建议在 20 个字段以内。

④ 表拆分

当数据库中的数据非常大时，查询优化方案也不能解决查询速度慢的问题时，我们可以考虑拆分表，让每张表的数据量变小，从而提高查询效率。a) **垂直拆分**：是指数据表列的拆分，把一张列比较多的表拆分为多张表，比如，用户表中一些字段经常被访问，将这些字段放在一张表中，另外一些不常用的字段放在另一张表中，插入数据时，使用事务确保两张表的数据一致性。垂直拆分的原则：

- 把不常用的字段单独放在一张表；
- 把 text, blob 等大字段拆分出来放在附表中；
- 经常组合查询的列放在一张表中。

b) **水平拆分**：指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。

通常情况下，我们使用取模的方式来进行表的拆分，比如，一张有 400W 的用户表 users，为提高其查询效率我们把其分成 4 张表 users1, users2, users3, users4，然后通过用户 ID 取模的方法，同时查询、更新、删除也是通过取模的方法来操作。

⑤ 读写分离

一般情况下对数据库而言都是“读多写少”，换言之，数据库的压力多数是因为大量的读取数据的操作造成的，我们可以采用数据库集群的方案，使用一个库作为主库，负责写入数据；其他库为从库，负责读取数据。这样可以缓解对数据库的访问压力。

♡ MySQL 常见读写分离方案有哪些？

MySQL 常见的读写分离方案，如下列表：

1) 应用层解决方案 可以通过应用层对数据源做路由来实现读写分离，比如，使用 SpringMVC + MyBatis，可以将 SQL 路由交给 Spring，通过 AOP 或者 Annotation 由代码显示的控制数据源。优点：路由策略的扩展性和可控性较强。缺点：需要在 Spring 中添加耦合控制代码。

2) 中间件解决方案 通过 MySQL 的中间件做主从集群，比如：Mysql Proxy、Amoeba、Atlas 等中间件都能符合需求。优点：与应用层解耦。缺点：增加一个服务维护的风险点，性能及稳定性待测试，需要支持代码强制主从和事务。

什么是 MySQL 多实例？如何配置 MySQL 多实例？

MySQL 多实例就是在同一台服务器上启用多个 MySQL 服务，它们监听不同的端口，运行多个服务进程，它们相互独立，互不影响的对外提供服务，便于节约服务器资源与后期架构扩展。多实例的配置方法有两种：

- 一个实例一个配置文件，不同端口；
- 同一配置文件(my.cnf)下配置不同实例，基于 MySQL 的 d_multi 工具。

介绍一下 Sharding-JDBC 的功能和执行流程？

Sharding-JDBC 在客户端对数据库进行水平分区的常用解决方案，也就是保持表结构不变，根据策略存储数据分片，这样每一片数据被分散到不同的表或者库中，Sharding-JDBC 提供以下功能：

- 分库分表
- 读写分离
- 分布式主键生成

Sharding-JDBC 的执行流程：当业务代码调用数据库执行的时候，先触发 Sharding-JDBC 的分配规则对 SQL 语句进行解析、改写之后，才会对改写的 SQL 进行执行和结果归并，然后返回给调用层。

怎样保证确保备库无延迟？

1. 每次从库执行查询请求前，先判断 secondsbehindmaster 是否已经等于0。如果不等于0.要等到这个参数变为0才能执行查询操作。secondsbehindmaster 是用来衡量主备延迟时间的长短
2. 对比 **位点** 确保主备无延迟。MasterLogFile 和 ReadMasterLogPos，表示的是读到的主库的最新位点，RelayMasterLogFile 和 ExecMasterLog_Pos，表示的是备库执行的最新位点；
3. 对比 GTID 集合确保主备无延迟。AutoPosition=1，表示这对主备关系使用了 GTID 协议；RetrievedGtidSet，是备库收到的所有日志的 GTID 集合；ExecutedGtid_Set，是备库所有已经执行完成的 GTID 集合。

MySQL 主从延迟的原因有哪些？

主从延迟可以根据 MySQL 提供的命令判断，比如，在从服务器使用命令：`show slave status;`，其中 SecondsBehindMaster 如果为 0 表示主从复制状态正常。导致主从延迟的原因有以下几个：

- 主库有大事务处理；
- 主库做大量的增、删、改操作；
- 主库对大表进行字段新增、修改或添加索引等操作；
- 主库的从库太多，导致复制延迟。从库数量一般 3-5 个为宜，要复制的节点过多，导致复制延迟；
- 从库硬件配置比主库差，导致延迟。查看 Master 和 Slave 的配置，可能因为从库的配置过低，执行时间长，由此导致的复制延迟时间长；
- 主库读写压力大，导致复制延迟；
- 从库之间的网络延迟。主从库网卡、网线、连接的交换机等网络设备都可能成为复制的瓶颈，导致复制延迟，另外跨公网主从复制很容易导致主从复制延迟。

如何保证数据不被误删？

保证数据不被误删的方法如下列表：

- 权限控制与分配（数据库和服务器权限）
- 避免数据库账号信息泄露，在生产环境中，业务代码不要使用明文保存数据库连接信息；

- 重要的数据库操作，通过平台型工具自动实施，减少人工操作；
- 部署延迟复制从库，万一误删除时用于数据回档，且从库设置为 read-only；
- 确认备份制度及时有效；
- 启用 SQL 审计功能，养成良好 SQL 习惯；
- 启用 sqlsafeupdates 选项，不允许没 where 条件的更新/删除；
- 将系统层的 rm 改为 mv；
- 线上不进行物理删除，改为逻辑删除（将 row data 标记为不可用）；
- 启用堡垒机，屏蔽高危 SQL；
- 降低数据库中普通账号的权限级别；
- 开启 binlog，方便追溯数据。

MySQL 服务器 CPU 飙升应该如何处理？

使用 `show full processlist;` 查出慢查询，为了缓解数据库服务器压力，先使用 `kill` 命令杀掉慢查询的客户端，效果如下：

mysql> show full processlist;								
Id	User	Host	db	Command	Time	State	Info	
4	event_scheduler	localhost	NULL	Daemon	164972	Waiting on empty queue	NULL	
25	root	localhost	learndb	Query	0	starting	show full processlist	
28	root	111.21.173.130:65451	learndb	Sleep	340		NULL	

3 rows in set (0.00 sec)

然后再去项目中找到执行慢的 SQL 语句进行修改和优化。

MySQL 毫无规律的异常重启，可能产生的原因是什么？该如何解决？

可能是积累的长连接导致内存占用太多，被系统强行杀掉导致的异常重启，因为在 MySQL 中长连接在执行过程中使用的临时内存对象，只有在连接断开的时候才会释放，这就会导致内存不断飙升，解决方案如下：

- 定期断开空闲的长连接；
- 如果是用的是 MySQL 5.7 以上的版本，可以定期执行 `mysqlresetconnection` 重新初始化连接资源，这个过程会释放之前使用的内存资源，恢复到连接刚初始化的状态

mysql 问题排查都有哪些手段？

- 使用 `show processlist` 命令查看当前所有连接信息。
- 使用 `explain` 命令查询 SQL 语句执行计划。
- 开启慢查询日志，查看慢查询的 SQL。

MySQL 慢查询怎么解决？

`slow_query_log`

- 这个参数设置为ON，可以捕获执行时间超过一定数值的SQL语句。

`long_query_time`

- 当SQL语句执行时间超过此数值时，就会被记录到日志中，建议设置为1或者更短。

`slow_query_log_file`

- 记录日志的文件名。

`log_queries_not_using_indexes`

- 这个参数设置为ON，可以捕获到所有未使用索引的SQL语句，尽管这个SQL语句有可能执行得挺快。

查询长时间不返回可能是什么原因？应该如何处理？

- 1) 查询字段没有索引或者没有触发索引查询
- 2) I/O 压力大，读取磁盘速度变慢。
- 3) 内存不足
- 4) 网络速度慢
- 5) 查询出的数据量过大，可以采用多次查询或其他的方法降低数据量
- 6) 死锁，一般碰到这种情况的话，大概率是表被锁住了，可以使用 `show processlist;` 命令，看看 SQL 语句的状态，再针对不同的状态做相应的处理。

23.垂直切分、水平切分、分库分表、读写分离

垂直切分

即将表按照功能模块、关系密切程度划分出来，部署到不同的库上。例如，我们会建立定义数据库 `workDB`、商品数据库 `payDB`、用户数据库 `userDB`、日志数据库 `logDB` 等，分别用于存储项目数据定义表、商品定义表、用户数据表、日志数据表等。

水平切分

水平切分：当一个表中的数据量过大时，我们可以把该表的数据按照某种规则，例如 `userID` 散列，进行划分，然后存储到多个结构相同的表，和不同的库上。例如，我们的 `userDB` 中的用户数据表中，每一个表的数据量都很大，就可以把 `userDB` 切分为结构相同的多个 `userDB`: `part0DB`、`part1DB` 等，再将 `userDB` 上的用户数据表 `userTable`，切分为很多 `userTable`: `userTable0`、`userTable1` 等，然后将这些表按照一定的规则存储到多个 `userDB` 上。

单库多表

随着用户数量的增加，~~user~~ 表的数据量会越来越大，当数据量达到一定程度的时候对 `user` 表的查询会渐渐的变慢，从而影响整个 DB 的性能。如果使用 `mysql`，还有一个更严重的问题是，当需要添加一列的时候，`mysql` 会锁表，期间所有的读写操作只能等待。

可以将 `user` 进行水平的切分，产生两个表结构完全一样的 `user_0000`,`user_0001` 等表，~~user_0000 + user_0001 + ...~~ 的数据刚好是一份完整的数据。

多库多表

~~随着数据量增加也许单台 DB 的存储空间不够，随着查询量的增加单台数据库服务器已经没办法支撑。这个时候可以再对数据库进行水平区分。~~

~~分库分表规则举例：~~

通过分库分表规则查找到对应的表和库的过程。如分库分表的规则是 `user_id` 除以 4 的方式，当用户新注册了一个账号，账号 `id` 的 123，我们可以通过 `id` 除以 4 的方式确定此账号应该保存到 `User_0003` 表中。当用户 123 登录的时候，我们通过 123 除以 4 后确定记录在 `User_0003` 中。

Mysql读写分离

在实际的应用中，绝大部分情况都是读远大于写。Mysql 提供了读写分离的机制，所有的写操作都必须对应到 Master，读操作可以在 Master 和 Slave 机器上进行，Slave 与 Master 的结构完全一样，一个 Master 可以有多个 Slave，甚至 Slave 下还可以挂 Slave，通过此方式可以有效的提高 DB 集群的每秒查询率。

所有的写操作都是先在 Master 上操作，然后同步更新到 Slave 上，所以从 Master 同步到 Slave 机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave 机器数量的增加也会使这个问题更加严重。
此外，可以看出 Master 是集群的瓶颈，当写操作过多，会严重影响到 Master 的稳定性，如果 Master 挂掉，整个集群都将不能正常工作。
所以，1. 当读压力很大的时候，可以考虑添加 Slave 机器的分式解决，但是当 Slave 机器达到一定的数量就得考虑分库了。 2. 当写压力很大的时候，就必须得进行分库操作。

♡ 主从复制

主从复制的几种方式？

同步复制

- master 的数据存在变化，必须等 slave1、slave2、。。。slaven 同步完成才能返回。

异步复制

- 如同AJAX请求一样。master 只要完成自己的数据库操作即可，不必关心 slave 是否收到 binlog 文件，是否完成操作。是 MySQL 的默认配置。

半同步复制

- master 只保证 slaves 中的一个操作成功，就返回，其他 slave 不管。这个功能是由 google 为 MySQL 引入的。

slave 是否可以进行写操作？

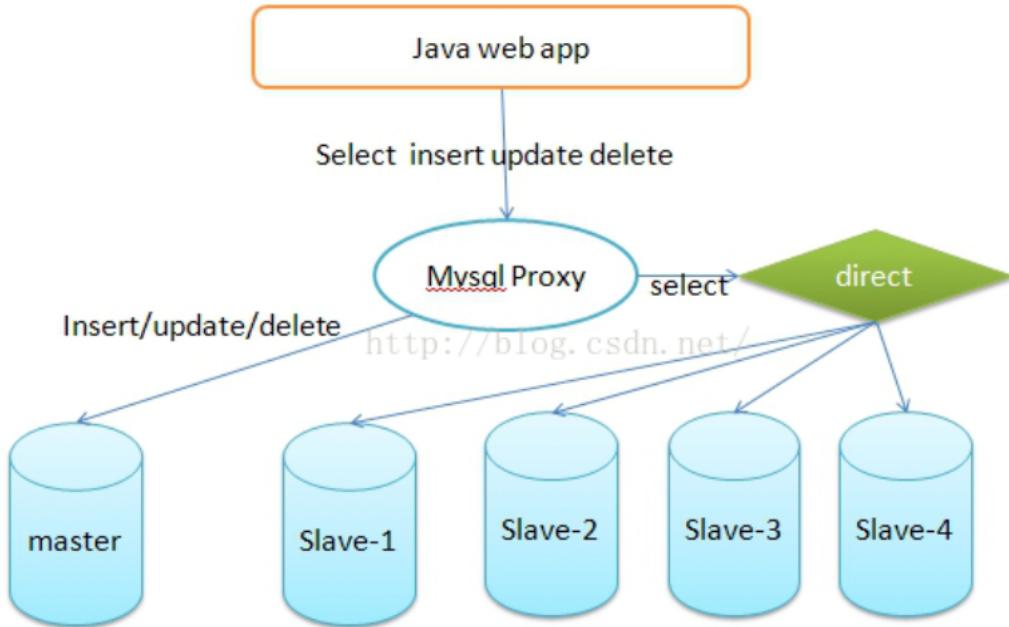
slave 不可以进行写操作，因为 slave 无法通知 master，这样导致 master 和 slave 数据不一致。

为什么需要多个 slave？

1. 异地容灾：比如 master 放在北京，因为地址导致挂点，位于上海的 slave 顶上去成为 master。
2. 高可用。
3. 读写分离，将读请求分担到多个 slave 上，分担负载

4. 主从复制中有 master, slave1, slave2, ... 等等这么多 MySQL 数据库，那比如一个 JAVA WEB 应用到底应该连接哪个数据库？

使用简单的轮询算法，找一个组件（haproxy）来完成。



5.当 master 的二进制日志每产生一个事件，都需要发往 slave，如果我们有 N 个 slave,那是发 N 次，还是只发一次？

复制n次。可以使用多级复制的概念。

slave1 作为 master 的从节点。 slave2、 slave3、 slave4作为slave1的从节点。同时， slave1不在负责查询。

6. 当一个 select 发往 MySQL proxy，可能这次由 slave-2 响应，下次由 slave-3 响应，这样的话，就无法利用查询缓存了。

可以使用一个共享式缓存，redis 缓存

7. 随着应用的日益增长，读操作很多，我们可以扩展 slave，但是如果 master 满足不了写操作了，怎么办呢？

分库（垂直拆分）分表（水平拆分）

☆ Redis ☆

1.redis 是什么？都有哪些使用场景？

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库，并提供多种语言的API。

redis 有哪些功能？

- 数据缓存功能
- 分布式锁的功能
- 支持数据持久化
- 支持事务
- 支持消息队列

Redis 使用场景：

- 数据高并发的读写
- 海量数据的读写
- 对扩展性要求高的数据

(1) 、会话缓存 (Session Cache)

最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？幸运的是，随着Redis这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

(2)、全页缓存 (FPC)

除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

(3)、队列

Redis在内存存储引擎领域的一大优点是提供list和set操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对list的push/pop操作。如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

(4)、排行榜/计数器

Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变得非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户-我们称之为“user_scores”，我们只需要像下面一样执行即可：当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：ZRANGE user_scores 0 10 WITHSCORES Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

(5)、发布/订阅

最后（但肯定不是最重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！

redis 支持的数据类型有哪些？

string

- 格式: set key value
- string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象。
- string类型是Redis最基本的数据类型，一个键最大能存储512MB。

list

- redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）
- 格式: lpush name value
在key对应list的头部添加字符串元素
- 格式: rpush name value
在key对应list的尾部添加字符串元素
- 格式: lrem name index
key对应list中删除count个和value相同的元素
- 格式: llen name
返回key对应list的长度

hash

- 格式: hmset name key1 value1 key2 value2
- Redis hash是一个键值(key=>value)对集合。

- Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。

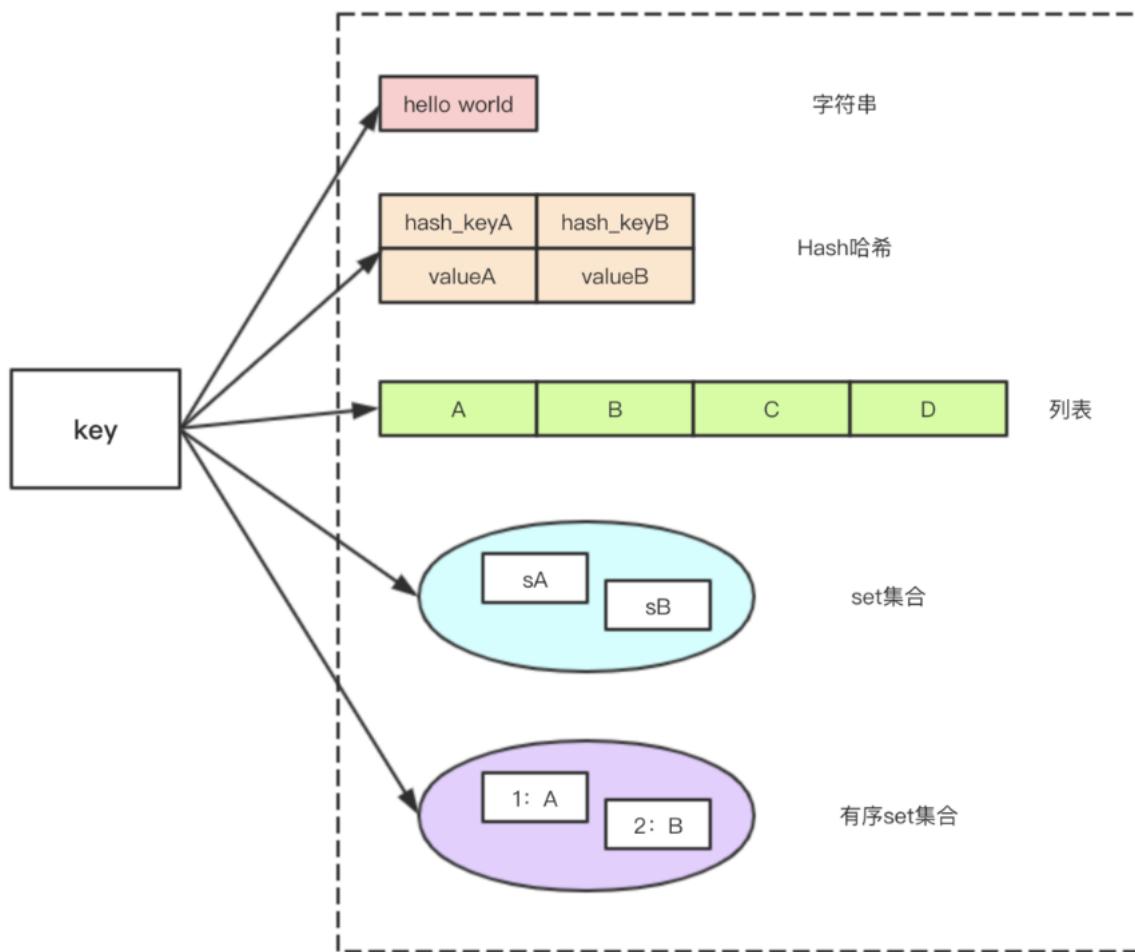
set

- 格式: sadd name value
- Redis的Set是string类型的无序集合。
- 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。

zset

- (sorted set: 有序集合) 不允许重复
- 格式: zadd name score value
- 不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。
- zset的成员是唯一的,但分数(score)却可以重复。

加分项：另外redis还对这几种数据结构做了扩展，如GEO对位置计算，hyperLogLog做统计，bitmaps: redis底层存储value值都是存储的二进制数据，redis提供bitmaps（位图）可以直接访问或修改底层存储的二进制数据



♡ redis 和 memecache 有什么区别?

- memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
- redis的速度比memcached快很多
- redis可以持久化其数据

4.redis 为什么是单线程的？使用队列技术将并发访问变成串行访问

因为 **cpu 不是 Redis 的瓶颈**，Redis 的瓶颈最有可能是**机器内存或者网络带宽**。既然单线程容易实现，而且 cpu 又不会成为瓶颈，那就顺理成章地采用单线程的方案了。

关于 Redis 的性能，官方网站也有，普通笔记本轻松处理每秒几十万的请求。

而且单线程并不代表就慢 nginx 和 nodejs 也都是高性能单线程的代表。

♡什么是缓存穿透？怎么解决？

缓存穿透：表示恶意用户频繁的请求缓存不存在的数据，以致这些请求短时间内落在数据库上，导致数据库性能急剧下降，最终影响服务整体的性能。

解决方案：

1. 布隆过滤：将所有可能存在的数据存到一个bitMap中，不存在的数据就会进行拦截
2. 对查询结果为空的情况也进行缓存（不管数据是否存在，还是系统故障），缓存时间设置短一点，不超过5分钟。

空对象：内存空间占用大（设置较短的过期时间）、数据不一致（利用消息系统清除缓存）

布隆过滤：数据命中不高、数据相对固定、实时性低（数据集较大的场景）、代码维护复杂、占用空间少

表11-3 缓存空对象和布隆过滤器方案对比

解决缓存穿透	适用场景	维护成本
缓存空对象	<ul style="list-style-type: none">• 数据命中不高• 数据频繁变化实时性高	<ul style="list-style-type: none">• 代码维护简单• 需要过多的缓存空间• 数据不一致
布隆过滤器	<ul style="list-style-type: none">• 数据命中不高• 数据相对固定实时性低	<ul style="list-style-type: none">• 代码维护复杂• 缓存空间占用少

缓存降级

缓存降级就是当缓存失效或者服务挂掉时，我们也不去访问数据库。直接访问内存部分数据缓存或者返回默认数据。

♡什么是缓存雪崩？如何解决？

缓存雪崩：在短时间有大量缓存失效，如果这段时间有大量的请求发生同样会导致数据库发生宕机。

解决办法：

1. **做二级缓存**，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期
2. 计算数据缓存节点的时候采用一致性 hash 算法，这样在节点数量发生改变时不会存在大量的缓存数据需要迁移的情况发生。
3. **设置不同的过期时间**，让缓存失效的时间点尽量均匀。
4. 如果缓存数据库是分布式部署，将热点数据均匀分布在不同缓存数据库中。
5. 在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。
6. **数据预热**，在系统上线前，将相关直接加载到缓存系统上
7. **定时更新二级缓存**对时效性不高的缓存，容器启动初始化加载，采用定时任务更新或移除缓存
 - 事前：redis 高可用 主从+哨兵 redis cluster 避免全盘崩溃
 - 事中：hystrix 限流降级，避免MySQL被打死
 - 事后：redis 持久化，快速恢复缓存数据

什么是缓存预热？

缓存预热指的是系统上线后，将相关的数据加载到缓存中。避免刚上线时大量的请求打过来，导致系统宕机。

♡什么是缓存击穿？(热点key重建优化问题)

缓存击穿指的是一个key **非常热点**，在不停的扛着高并发，大并发集中在这个点进行访问，当这个key在失效的瞬间，持续的大并发导致缓存击穿，直接去请求数据库，就像一个屏障上开了一个洞。

解决办法

1. 互斥锁

此方法只允许一个线程重建缓存，其他线程等待重建缓存的线程执行完，重新从缓存获取数据即可

2. 永远不过期 “永远不过期”包含两层意思：①从缓存层面来看，确实没有设置过期时间，所以不会出现热点key过期后产生的问题，也就是“物理”不过期。②从功能层面来看，为每个value设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

表11-5 两种热点key的解决方法

解决方案	优点	缺点
简单分布式锁	<ul style="list-style-type: none">• 思路简单• 保证一致性	<ul style="list-style-type: none">• 代码复杂度增大• 存在死锁的风险• 存在线程池阻塞的风险
“永远不过期”	基本杜绝热点 key 问题	<ul style="list-style-type: none">• 不保证一致性• 逻辑过期时间增加代码维护成本和内存成本

3. 缓存屏障

该方法类似于方法一：使用CountDownLatch和atomicInteger.compareAndSet()方法实现轻量级锁

```
class MyCache{  
  
    private ConcurrentHashMap<String, String> map;  
  
    private CountDownLatch countDownLatch;  
  
    private AtomicInteger atomicInteger;  
  
    public MyCache(ConcurrentHashMap<String, String> map, CountDownLatch countDownLatch,  
                  AtomicInteger atomicInteger) {  
        this.map = map;  
        this.countDownLatch = countDownLatch;  
        this.atomicInteger = atomicInteger;  
    }  
  
    public String get(String key){  
  
        String value = map.get(key);  
        if (value != null){  
            System.out.println(Thread.currentThread().getName()+"\t 线程获取value值  
value="+value);  
            return value;  
        }  
        // 如果没获取到值  
        // 首先尝试获取token，然后去查询db，初始化缓存；  
        // 如果没有获取到token，超时等待  
        if (atomicInteger.compareAndSet(0, 1)){  
            System.out.println(Thread.currentThread().getName()+"\t 线程获取token");  
            return null;  
        }  
  
        // 其他线程超时等待  
        try {  
            countDownLatch.await();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        System.out.println(Thread.currentThread().getName()+"\t 线程没有获取token, 等待
中。。。");
        countDownLatch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    // 初始化缓存成功, 等待线程被唤醒
    // 等待线程等待超时, 自动唤醒
    System.out.println(Thread.currentThread().getName()+"\t 线程被唤醒, 获取value
="+map.get("key"));
    return map.get(key);
}

public void put(String key, String value){

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    map.put(key, value);

    // 更新状态
    atomicInteger.compareAndSet(1, 2);

    // 通知其他线程
    countDownLatch.countDown();
    System.out.println();
    System.out.println(Thread.currentThread().getName()+"\t 线程初始化缓存成功! value
="+map.get("key"));
}
}

class MyThread implements Runnable{

    private MyCache myCache;

    public MyThread(MyCache myCache) {
        this.myCache = myCache;
    }

    @Override
    public void run() {
        String value = myCache.get("key");
        if (value == null){
            myCache.put("key", "value");
        }
    }
}

public class CountDownLatchDemo {
    public static void main(String[] args) {

        MyCache myCache = new MyCache(new ConcurrentHashMap<>(), new CountDownLatch(1), new
AtomicInteger(0));
    }
}
```

```
MyThread myThread = new MyThread(myCache);

ExecutorService executorService = Executors.newFixedThreadPool(5);
for (int i = 0; i < 5; i++) {
    executorService.execute(myThread);
}
}
```

缓存无底洞问题

2010年，Facebook的Memcache节点已经达到了3000个，承载着TB级别的缓存数据。但开发和运维人员发现了一个问题，为了满足业务要求添加了大量新Memcache节点，但是发现性能不但没有好转反而下降了，当时将这种现象称为缓存的“无底洞”现象。

为什么会发生这个问题呢？

由于数据量和访问量的持续增长，造成需要大量的节点进行水平扩展，导致键值分布在不同的节点上，相比于单机批量操作只涉及一次网络操作，分布式批量操作会涉及多次网络时间

IO优化思路：

- 命令本身的优化，例如优化SQL语句等。
- 减少网络通信次数。
- 降低接入成本，例如客户端使用长连/连接池、NIO等。

网络通信次数

- 客户端n次get： n次网络+n次get命令本身。
- 客户端1次pipeline get： 1次网络+n次get命令本身。
- 客户端1次mget： 1次网络+1次mget命令本身。

1.串行命令

由于n个key是比较均匀地分布在Redis Cluster的各个节点上，因此无法使用mget命令一次性获取，所以通常来讲要获取n个key的值，最简单的方法就是逐次执行n个get命令，这种操作时间复杂度较高，它的**操作时间 = n次网络时间 + n次命令时间**，网络次数是n。很显然这种方案不是最优的，但是实现起来比较简单，如图11-9所示。

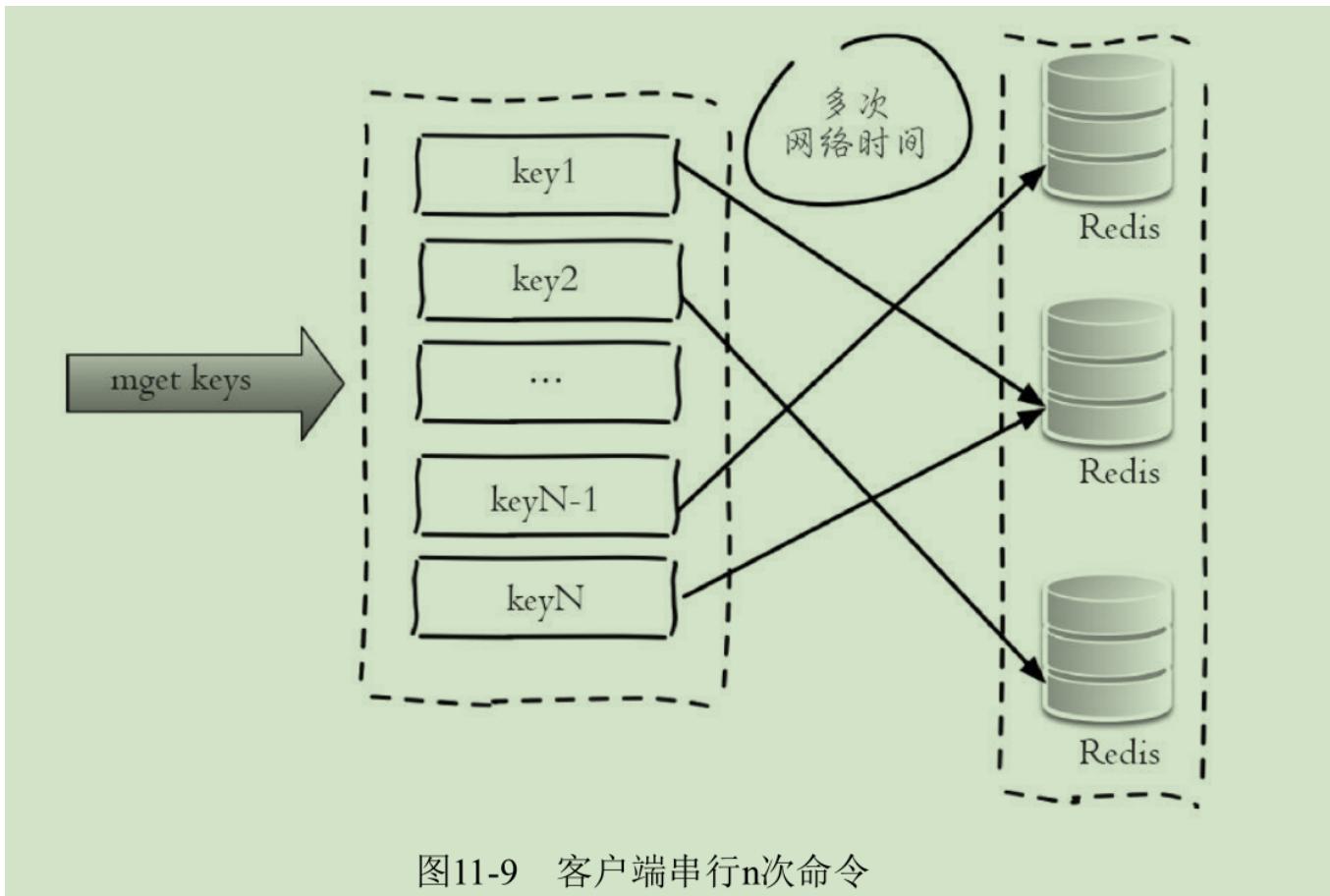


图11-9 客户端串行n次命令

2. 串行IO

Redis Cluster使用CRC16算法计算出散列值，再取对16383的余数就可以算出slot值，同时Smart客户端会保存 **slot** 和 **节点** 的对应关系，有了这两个数据就可以将属于同一个节点的key进行归档，得到每个节点的 key 子列表，之后对每个节点执行 mget 或者 Pipeline 操作，它的操作时间 = **node次网络时间 + n次命令时间**，网络次数是node的个数，整个过程如图11-10所示，很明显这种方案比第一种要好很多，但是如果节点数太多，还是有一定的性能问题。

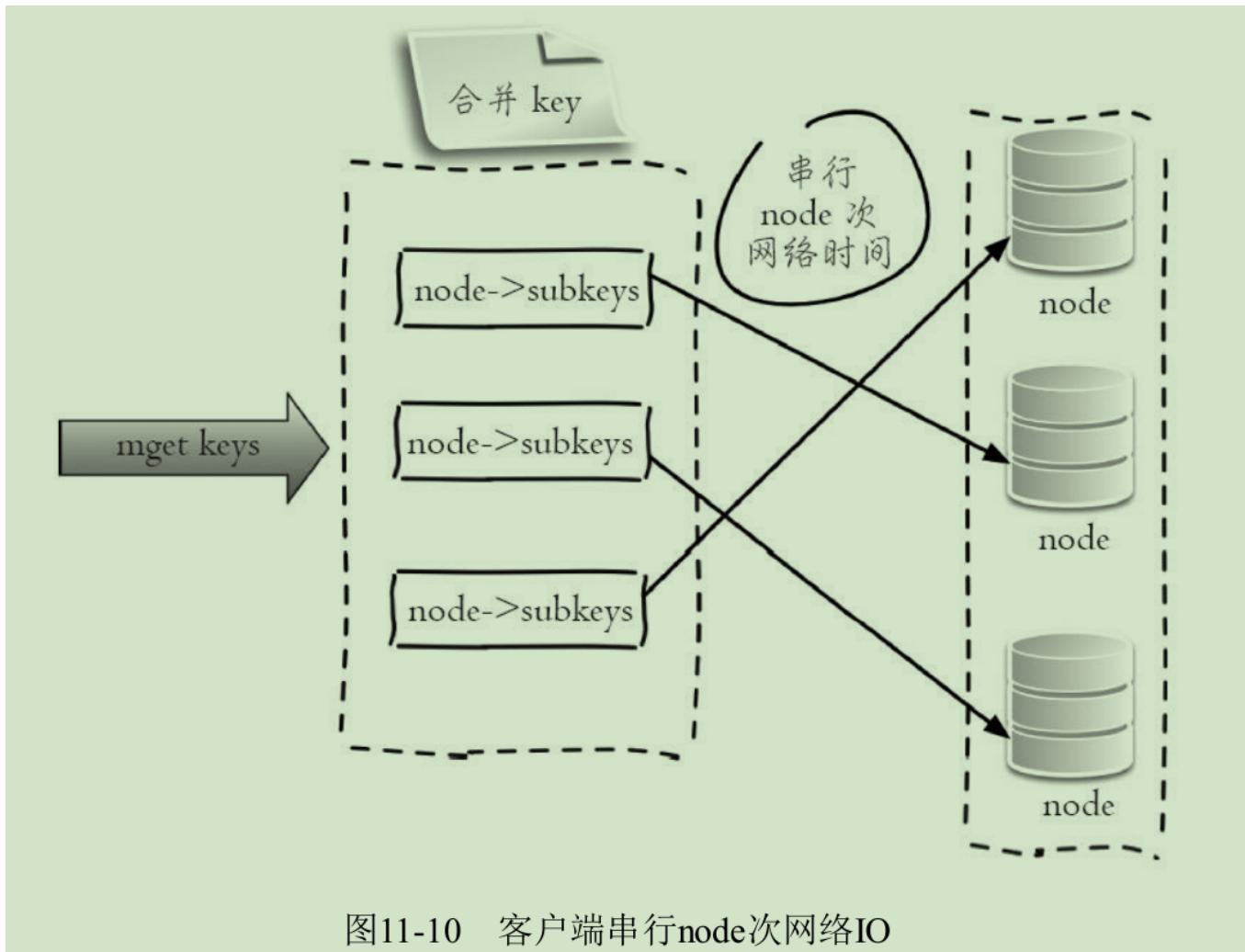


图11-10 客户端串行node次网络IO

3.并行IO 此方案是将方案2中的最后一步改为多线程执行，网络次数虽然还是节点个数，但由于使用多线程网络时间变为 $O(1)$ ，这种方案会增加编程的复杂度。它的操作时间为：

```
max_slow(node 网络时间)+n 次命令时间
```

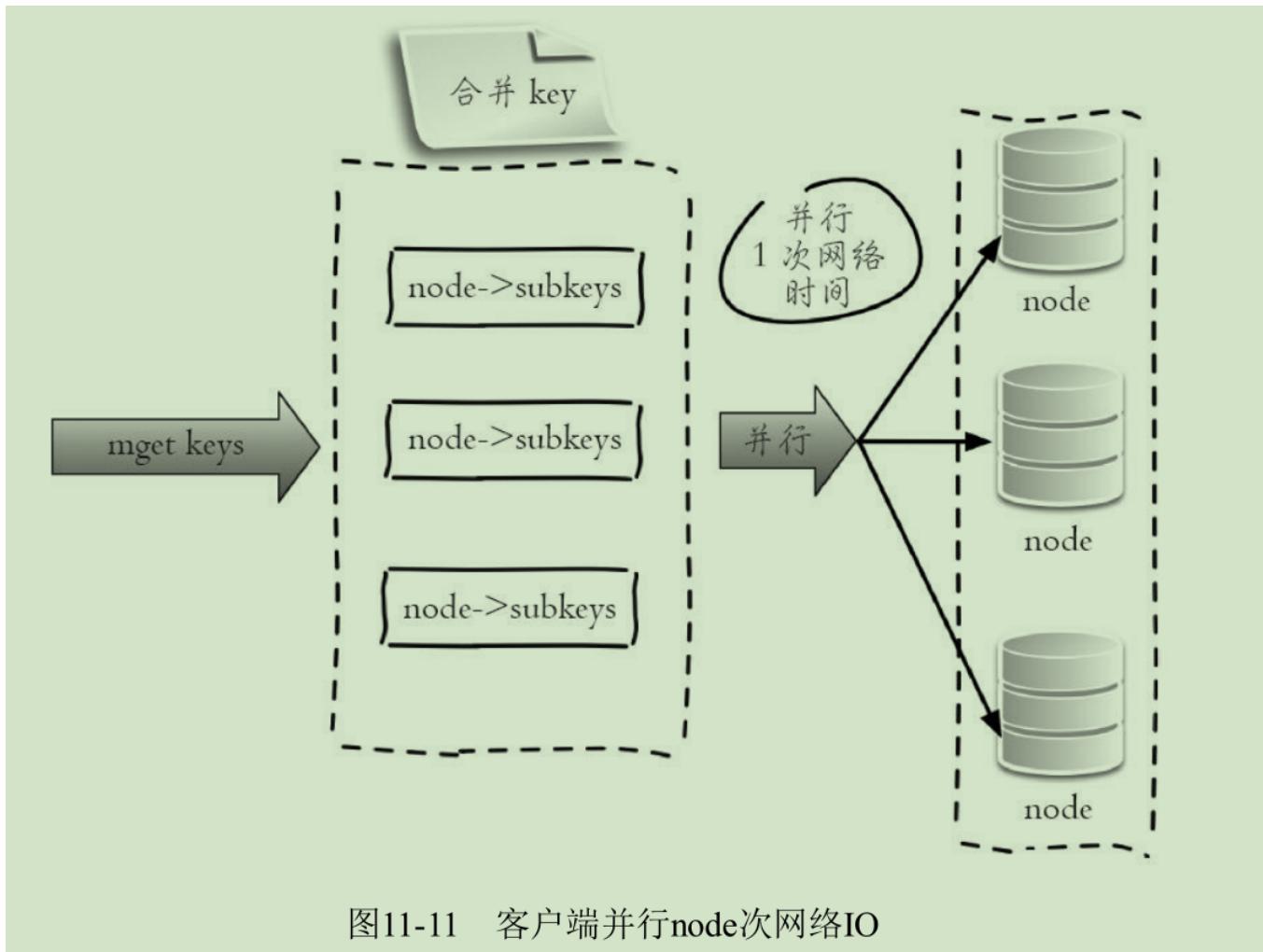


图11-11 客户端并行node次网络IO

缓存粒度问题

通俗来讲，缓存粒度问题就是我们在使用缓存时，是将所有数据缓存还是缓存部分数据？

数据类型	通用性	空间占用（内存空间+网络码率）	代码维护
全部数据	高	大	简单
部分数据	低	小	较为复杂

缓存预热

1. 在系统上线前将热点数据缓存到Redis中。

redis事务

什么是事务

redis 事务是一组命令的集合，是redis的最小执行单位，一个事务要么都执行，要不都不执行。主要有以下三个重要特征：

- 批量操作在发送EXEC命令前被放入队列缓存
- 收到EXEC命令后进入事务执行，事务中任意命令执行失败，其余的命令依然执行
- 在事务执行过程中，其他客户端提交的命令请求不会插入到事务执行序列中

redis 事务的原理是先将属于一个事务的命令发送给redis，然后依次执行这些命令

为什么Redis事务不具备原子性?

单个Redis命令的执行是原子性的，但是Redis不支持回滚操作。事务可以理解为一个打包的批量执行脚本，但批量指令并发原子性操作，一旦中间某条指令失败，不会导致之前的指令回滚，而是继续执行后续指令。

Redis事务相关命令有哪些?

- **DISCARD** 取消事务，放弃执行事务块内的所有命令 ❤ **discard (抛弃)**
- **EXEC** 执行所有事务块内的命令 ❤ **exec(执行)**
- **MULTI** 标记一个事务的开始 ❤ **multi(多)**
- **WATCH** 监视一个（或多个）key，如果事务执行之前这个（或这些）key被其他命令所改动，那么事务将被打断
- **UNWATCH** 取消WATCH命令对所有key的监视

8.怎么保证缓存和数据库数据的一致性?

将不一致分为三种情况：

1. 数据库有数据，缓存没有数据；
2. 数据库有数据，缓存也有数据，数据不相等；
3. 数据库没有数据，缓存有数据。

缓存模式

- 首先尝试从缓存读取，读到数据则直接返回；如果读不到，就读数据库，并将数据写到缓存，并返回
 - 需要更新数据时，先更新数据库，然后把缓存里对应的数据失效掉（删除）
1. 对于第一种，在读数据的时候，会自动把数据库的数据写到缓存，因此不一致自动消除。
 2. 对于第二种，数据最终变成了不相等，但他们之前在某一个时间点一定是相等的（不管你使用懒加载还是预加载的方式，在缓存加载的那一刻，它一定和数据库一致）。这种不一致，一定是由于你更新数据所引发的。前面我们讲了更新数据的策略，先更新数据库，然后删除缓存。因此，不一致的原因，一定是数据库更新了，但是删除缓存失败了。
 3. 对于第三种，情况和第二种类似，你把数据库的数据删了，但是删除缓存的时候失败了。

最终的结论是，需要解决的不一致，产生的原因是更新数据库成功，但是删除缓存失败。

解决方案

1. 对删除缓存进行重试，数据的一致性要求越高，重试的越快
2. 定期全量更新，简单的说就是定期把缓存全部清掉，然后在全量加载
3. 给所有缓存一个失效期

针对并发情况

并发不高

- 读：redis -> 没有，读 MySQL -> 把MySQL数据写回redis；有直接从redis中读取
- 写：写MySQL -> 成功，再写redis

并发高

- 读：redis -> 没有，读 MySQL -> 把MySQL数据写回redis；有直接从redis中读取
- 写：异步先写入redis缓存，直接返回；定期将数据保存到MySQL中，可以做到多次更新，一次保存

❤ redis持久化

Redis 的持久化有哪两种方式？

- RDB (Redis Database) : 指定的时间间隔能对你的数据进行快照存储。定时持久机制，宕机可能会丢失最后一次持久化之后的数据
- AOF (Append Only File) : 每一个收到的写命令都通过write函数追加到文件中。

AOF (Append-only file)

以独立日志的方式记录每次写命令，重启时再重新执行 AOF 文件中的命令达到数据恢复的目的

优点

1. 数据安全：实时性高，
 - 触发机制（根据配置文件配置项） no: 表示等操作系统进行数据缓存同步到磁盘（快，持久化没保证） always: 同步持久化，每次发生数据变更时，立即记录到磁盘（慢，安全） everysec: 表示每秒同步一次（默认值很快，但可能会丢失一秒以内的数据）
2. 通过 append 模式写文件，即使中途服务宕机，可以通过 redis-check-aof 工具解决数据一致性问题。
3. AOF 机制的 rewrite 模式。（AOF文件没被rewrite之前（文件过大时会对命令进行合并重写），可以删除其中的某些命令（比如误操作的flushall））

缺点

1. AOF 文件比 RDB 文件大，且恢复速度慢
2. 数据集大时，比RDB 启动效率低

RDB (redis DataBase)

指用数据集快照的方式（半持久话模式），记录 redis 数据库所有的键值对，在某个时间点将数据写入到一个临时文件，待持久化结束后，用这个临时文件替换上次持久化好的文件。

优点

1. 只有一个持久化文件 dump.rdb 方便持久化
2. 容灾性好，一个文件可以保存到安全的磁盘
3. 性能最大化，fork 子进程来完成写操作，让主进程继续执行命令，所以是 IO 最大化。（使用单独子进程来进行持久化，主进程不进行任何IO 操作，保证了 redis 的高性能）。
4. 相对于数据集大时，比 AOF 的启动效率更高

缺点

数据安全性低： RDB 会每隔一段时间进行一次持久化，如果持久化之间 redis 发生故障，会发生数据丢失。

♡ redis 分布式锁？

1. 分布式锁的基本要素

- 安全特性：相互排斥。在任何时候只有一个客户端可以被锁定，不会发生2个客户端同时被锁定。
- 活性A：无死锁。无论何时，锁定资源的客户端崩溃或者网络被分裂总是可以获取锁的。
- 活性B：容错，只要大多数redis 节点启动，客户端就可以获取并释放锁。

2. 单机实例锁实现

为什么基于故障转换（master-slave模型，master 故障时，slave 升级为master）的实现是不够的？

- 实用redis 锁定资源的最简单办法是在redis 实例中创建一个密钥，通常实用redis 过期功能实现在客户端崩溃后锁的自动释放。

该方法存在单点故障问题，（master宕机、slave竞选成为master，通过此方法无法实现互斥安全的特性）因为redis 复制是异步的！

互斥安全问题：

- 客户端A 获取 master 设备中的锁定
- 将密钥写入slave节点前 master 崩溃
- slave 升级为 master
- 客户端 B 使用同样的密钥获取客户端 A 本以获取到的资源锁。

如何解决互斥安全问题

- SET resource_name my_random_value NX PX 30000 **set nx**(set if not exists)
- NX(只有键不存在才会设置成功)、PX (设置键的过期时间：单位毫秒)
- 该键设置为一个随机值，该随机值在所有客户端和所有对资源的锁定请求中必须唯一。 (否则会出现互斥问题)

基于随机值访问：适合于非分布式系统是安全的

- 基于随机值访问的目的是为了以安全的方式释放锁，使用随机值，通过脚本告诉redis：只有当 **密钥存在且存储在密钥中的值恰好与我期望的值（随机值）一样时** 才能移除该密钥 (执行del key 操作)
- 该过程主要作用是为了防止一个客户端移除其他客户端创建的锁。

例如：客户端 A 当前获取到了锁，但是客户端 A 并不是因为崩溃而超时，只是被阻塞超时，例如进行数据库操作时被阻塞，所以等到客户单 A 阻塞结束后，他会认为他还持有锁，然后会通过 DEL 命令尝试释放锁，**但是实际上该锁此时是客户端 B 所有的，而客户端 A 并不知道，这样就会造成客户端 A 将客户端 B 的锁删除而造成客户端 B 锁失效，并且后续客户端 B 可能还会重复该操作，此时锁实际上已经失去了作用，而如果加入随机数后，客户端 A 删除的时候会校验该随机数，如果发现该随机数与客户端 A 设置的不一致时则表示该锁已经不是 A 所有的，则 A 不会执行 DEL 操作， B 仍然会持有锁继而保持互斥安全，而这也是为什么要保证随机数唯一的原因。** ↵

简述

- Redis 分布式锁其实就是在系统里面占一个“坑”，其他程序也要占“坑”的时候，占用成功了就可以继续执行，失败了就只能放弃或稍后重试。
- 占坑一般使用 setnx(set if not exists)指令，只允许被一个程序占有，使用完调用 del 释放锁。

3. 分布式系统如何实现分布式锁？

Red Lock

- 尝试从N个独立的Redis 实例中获取锁
- 计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且大多数 ($N/2+1$) 实例上获取了锁。
- 如果获取锁失败，每个实例释放锁。

set nx expire

在上述算法的分布式版本中，我们假设我们有N个redis-master节点，这些节点完全独立，所以我们不使用复制或任何其他隐式协调系统。我们已经描述了如何在单个实例中安全的获取和释放锁，我们理所当然的认为该算法将使用此方法在单个示例中获取并释放锁。在我们的例子中，我们设置N=5，这是一个合理的值，所以我们需要在不同的计算机或虚拟机上运行5个redis-master节点，以确保他们以最独立的方式失败（单个节点挂掉不会影响其他节点）。

为了获得锁，客户端执行以下操作：

1. 它以毫秒为单位获取当前时间；
2. 它试图依次获取所有 N 个实例中的锁，在所有实例中使用**相同的密钥名和随机值**。

在步骤2中，当在每个实例中设置锁时，客户端使用与锁自动释放时间相比较小的超时以获取它，例如：**如果自动释放时间是10秒，则超时可能在5至50毫秒范围内**。这样当某个redis挂掉后可以防止客户端长时间阻塞在尝试与该redis实例创建连接的状态，如果redis实例不可用，那么我们应该尽快尝试与下一个redis实例通讯。

3. 客户机通过从当前时间中**扣除步骤1中获得的时间戳**来计算获取锁定所花费的时间。当且仅当客户机能获得大多数redis实例的锁定（至少N/2+1个redis实例的锁定），并且获取该锁的总时间小于锁有效时间，则认为该锁被获取；
4. 如果获得锁，则其**有效时间**被认为是**初始有效时间**减去**经过时间**，该经过时间如步骤3中计算的那样。（因为如果超过该时间那么获取的第一个redis实例的锁将会超时自动释放），该锁将有可能不满足[至少N/2+1个redis实例的锁定]条件；
5. 如果客户端由于某种原因未能获得锁（或者无法锁定N/2+1个实例或者获得的锁有效时间为负），**他将尝试解锁所有实例**（即使他认为他没有获取到锁，方便后续其他客户端重新获取锁）；

算法是异步的吗？

上述算法依赖于这样一个假设：即虽然跨进程没有同步时钟，但每个进程中的本地时间仍以大约相同的速率步进，与锁的自动释放时间相比误差较小。这个假设与真实世界的计算机非常相似，每台计算机都有一个本地时钟，我们通常可以依靠不同的计算机来产生很小的时钟漂移。

在这一点上，我们需要更好地指定我们的互斥规则：**即锁的有效期只有持有锁的客户端在有效时间**（有效时间为步骤3中获得的）减去某个时间（仅几毫秒以补偿不同主机之间的时钟漂移）。

12.redis 如何做内存优化？

尽可能使用**散列表 (hashes)**，散列表（是说散列表里面存储的数少）使用的内存非常小，所以你应该尽可能的将你的数据模型抽象到一个散列表里面。

比如你的web系统中有一个用户对象，不要为这个用户的名称，姓氏，邮箱，密码设置单独的key,而是应该把这个用户的所有信息存储到一张散列表里面。

13.redis 淘汰策略有哪些？

- volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰。
- volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰。
- volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰。
- allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰。
- allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰。
- no-eviction（驱逐）：禁止驱逐数据，永不回收数据。

使用策略规则：

- 1、如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru
- 2、如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random

redis缓存失效策略？

(一) 定时删除:

- 每个设置过期时间的key都需要创建一个定时器，到期时间就会立即删除。
- 该策略立即删除过期的数据，对内存友好，但是会占用大量CPU时间去处理过期的数据，从而影响缓存的响应时间和吞吐量

(二) 惰性删除:

- 只有当访问一个key时，才会判断key是否过期，过期则清除。
- 该策略可以最大化节省CPU资源，却对内存非常不友好。极端情况下可能出现大量过期的key没有被再次访问，可是又不会清除，占用大量内存。

(三) 定期过期策略

- 每隔一定的时间，会扫描一定数量的数据库的 expires 字典中一定数量的key，并清除其中已经过期的key。
- 该策略是前两者的一个折中的方案。
- 通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

14. redis 常见的性能问题有哪些？该如何解决？

- 主服务器写内存快照，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以主服务器最好不要写内存快照。
- Redis 主从复制的性能问题，为了主从复制的速度和连接的稳定性，主从库最好在同一个局域网内。

18.一个字符串类型的值能存储最大容量是多少？

512M

20.redis 的同步机制了解么

redis 可以使用主从复制、从从复制，第一次同步时，主节点做一次，并同时将后续修改操作记录到内存 BUFFER 中，待完成后将RDB文件全量同步到复制节点，复制节点接受完成后将 RDB 镜像加载到内存。加载完成后，在通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

21.Pipeline有什么好处，为什么要用pipeline？

可以将多次IO 往返的时间缩减为一次，前提是pipeline 执行指令之间没有因果相关性。使用redis-benchmark 进行压测时可以发现影响redis的QPS 峰值的一个重要因素是 pipeline批次执行指令的数目。

redis集群

redis 集群

- 1.Redis集群是一个由多个节点组成的分布式服务集群，它具有复制、高可用和分片特性
- 2.Redis的集群没有中心节点，并且带有复制和故障转移特性，这可用避免单个节点成为性能瓶颈，或者因为某个节点下线而导致整个集群下线
- 3.集群中的主节点负责处理槽（储存数据），而从节点则是主节点的复制品
- 4.Redis集群将整个数据库分为16384个槽，数据库中的每个键都属于16384个槽中的其中一个
- 5.集群中的每个主节点都可以负责0个至16384个槽，当16384个槽都有节点在负责时，集群进入上线状态，可以执行客户端发送的数据命令
- 6.主节点只会执行和自己负责的槽有关的命令，当节点接收到不属于自己处理的槽的命令时，它将会处理指定槽的节点的地址返回给客户端，而客户端会向正确的节点重新发送
- 7.如果需要完整地分片、复制和高可用特性，并且要避免使用代理带来的性能瓶颈和资源消耗，那么可以选择使用Redis 集群；

如果只需要一部分特性（比如只需要分片，但不需要复制和高可用等），那么单独选用twemproxy、Redis的复制和Redis Sentinel中的一个或多个

讲讲主从复制

1. 从节点执行 slaveof 命令。与master 节点建立连接，此时还没有建立连接
2. 从节点内部的定时任务发现有主节点的信息，开始使用 socket 连接主节点
3. 连接建立成功后，从节点发送 ping 命令，希望得到pong 命令响应，否则进行重试
4. 如果此时主节点设置了权限，那么就进行权限验证，如果验证失败复制终止
5. 权限验证通过后，进行数据同步，这个步骤是最耗时的，主节点将所有数据全部发送给从节点
6. 当主节点把当前数据同步给从节点后，便完成了复制的建立流程。接下来主节点会把持续的写命令发送给从节点，保证主从数据一致性。

redis 主从架构

redis 主从架构可以进行主从复制（冗余备份）、读写分离

主从复制

- 使用这种架构可以保证系统的高可用，当master 出现故障发生宕机时，可以使用 从库替代master 进行故障恢复。

读写分离

- 主从架构中，可以关闭主服务器的持久化功能，只让从服务器进行持久化，这样可以提高主服务器的处理性能。从服务器设置为只读模式，这样可以避免从服务器的数据被修改。

redisCluster

Rediscluster集群模式 基本回答：Rediscluster是一个高可用集群，它基于分片（对key进行crc16，然后对16384取余）的原理，可以把他理解为是由多组哨兵集群组成，但是它不依赖哨兵

哨兵

主要作用就是启动哨兵节点对主节点进行监控，如果半数以上ping主节点不同，则认为主节点宕掉，他们会选举出一个从节点作为主机点进行故障转移。同时将这个变化通知给客户端。

Redis集群方案什么情况下会导致整个集群不可用？

答：有A, B, C三个节点的集群，在没有复制模型的情况下，如果节点B失败了，那么整个集群就会以为缺少5501-11000这个范围的槽而不可用

说说redis 哈希槽的概念

redis 没有使用一致性哈希，而是引入了哈希槽的概念，redis集群中有16384个哈希槽所谓哈希槽就是每个key 通过crc16校验后对 16384 取模来决定放置哪个槽，集群的每个节点负责一部分的 hash 槽

Redis集群的主从复制模型是怎样的？

答：为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型，每个节点都会有N-1个复制品。

Redis集群会有写操作丢失吗？为什么？

答：Redis并不能保证数据的强一致性，这意味着在实际中集群在特定的条件下可能会丢失写操作。

Redis集群之间是如何复制的？

答：异步复制

Redis集群默认选择0数据库，不支持选择。

apple 什么是慢查询？怎么配置？

执行命令操作时，如果某条指令超过了设置的阈值，就说明该条命令为慢指令。

配置参数

showlog-log-slower-than：超过这个阈值就是慢查询，**单位为微秒，默认为10000**

slowlog-max-len 慢查询的日志列表最大长度，当慢查询日志列表处于最大长度的时候，最早插入的一个命令将从列表中移除。

29.Redis回收进程如何工作的？

Redis检查内存使用情况，如果大于maxmemory的限制，则根据设定好的策略进行回收。

30.Redis的内存用完了会发生什么？

答：如果达到设置的上限，Redis的写命令会返回错误信息（但是读命令还可以正常返回。）或者你可以将Redis当缓存来使用配置淘汰机制，当Redis达到内存上限时会冲刷掉旧的内容。

31.一个redis实例最多存放多少keys？

理论上redis可以处理多达2³²的keys。redis的存储容量取决于可用内存值

32.MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据？

使用allkeys-lru淘汰机制，淘汰掉最少使用的数据

34.假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用keys会导致线程阻塞、线上服务会停顿，直到指令执行完毕，服务才会恢复。

最佳选择是使用scan。scan可以无阻塞的取出指定模式的key列表，花费时间要比keys长

35.如果有大量的key需要设置同一时间过期，一般需要注意什么？

答：如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象，有可能会出现缓存雪崩。一般需要在时间上加一个随机值，使得过期时间分散一些。

36.使用过Redis做异步队列么，你是怎么用的？

答：一般使用list结构作为队列，lpush生产消息，rpop消费消息。当rpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢？

- list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？

- 使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

如果对方追问pub/sub有什么缺点？

- 在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如RabbitMQ等。

如果对方追问redis如何实现延时队列？

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。

到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

1.TCP和UDP的区别

TCP

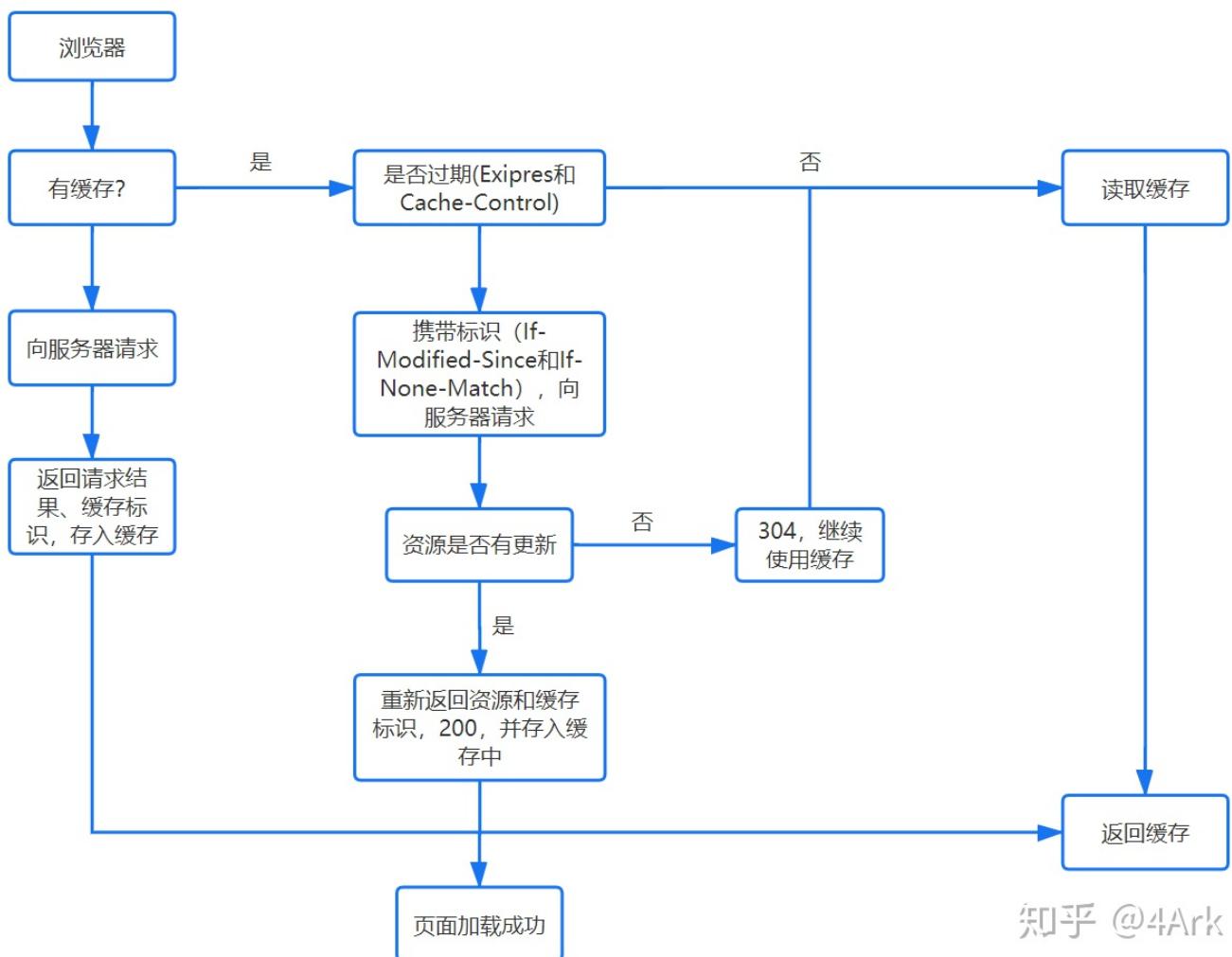
- 面向连接的传输控制协议，面向字节流
- 使用流量控制和拥塞控制的可靠传输
- 点到点的连接
- 对系统资源要求较多

UDP

- 基于数据报
- 无连接的不可靠传输
- 支持一对一，一对多，多对多，多对一的交互通信
- 具有较好的实时性
- 对系统资源要求较少

1.1 浏览器输入URL发生了什么？

1. URL 解析
2. DNS 查询
3. TCP 连接
4. 服务器处理请求
5. 客户端接收响应
6. 渲染页面

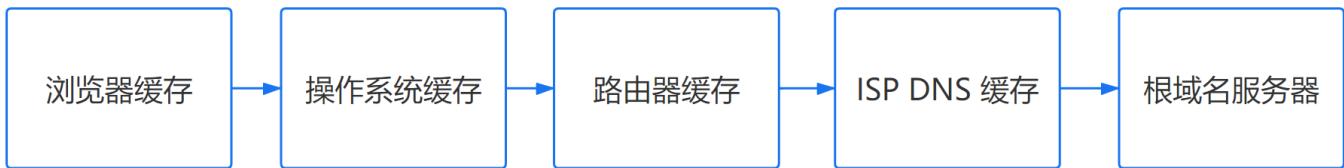


知乎 @4Ark

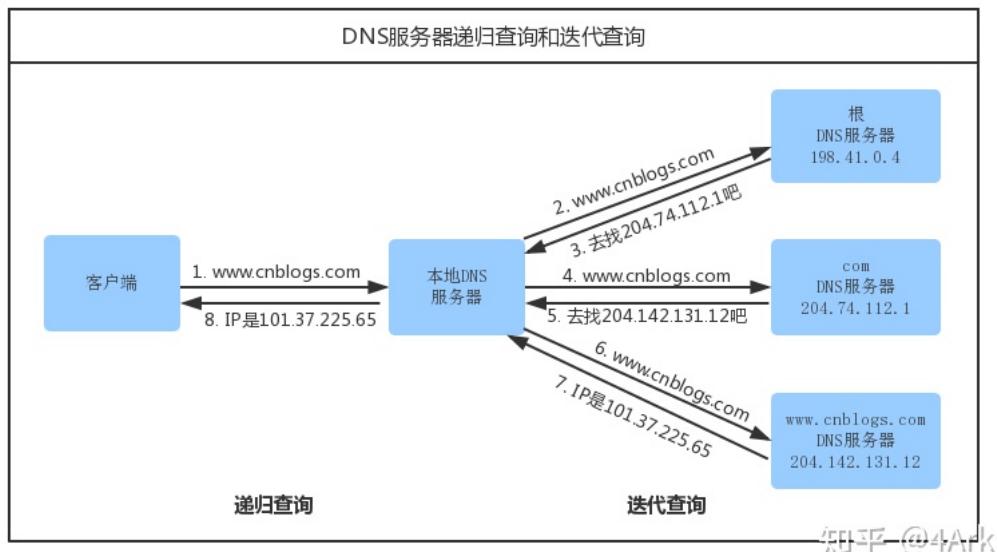
1) URL解析

- URL 解析出要请求的域名、服务器名、端口号、文件路径等信息，用于构造HTTP 请求

2) DNS 查询



- 检查浏览器的DNS 缓存
- 若浏览器中未找到该域名的 ip，则查找 **操作系统的DNS缓存** 对应的 hosts 文件即域名与ip地址的映射关系
- 若在操作系统中没有找到则 **递归** 查找 **本地DNS服务器缓存**
- 若本地DNS服务器缓存没有找到则查询 **路由器DNS 缓存**
- 若本地DNS 服务器没有找到则 **迭代** 查找 **根DNS 服务器**



优化

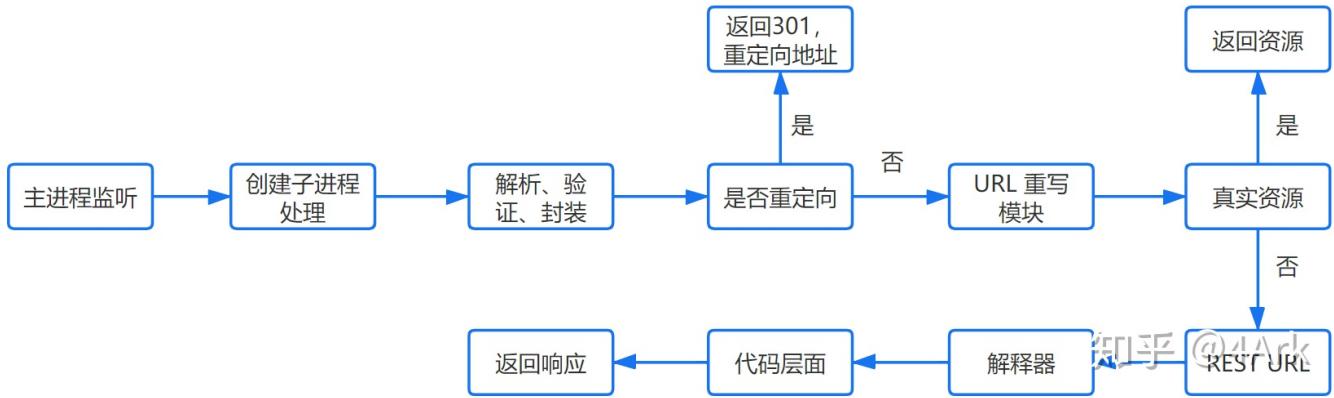
优化：prefetch 预获取，比如使用了cdn的域名

```

<meta charset="utf-8"/>
<meta name="renderer" content="webkit"/>
<meta http-equiv="X-UA-Compatible" content="IE=Edge"/>
<link rel="dns-prefetch" href="//g.alicdn.com"/>
<link rel="dns-prefetch" href="//img.alicdn.com"/>
<link rel="dns-prefetch" href="//gm.mmstat.com"/>
<link rel="dns-prefetch" href="//al.d.taobao.com"/>
<link rel="dns-prefetch" href="//bar.tmall.com"/>
<link href="//img.alicdn.com/tfs/TB1X1F3RpXXXXc6XXXXXXXXXXXX-16-16.png" rel="shortcut icon" type="image/x-icon"/>
  
```

3) 客户端与服务器端建立连接 通过三次握手

4) 服务器处理请求，并进行重定向、URL 重写

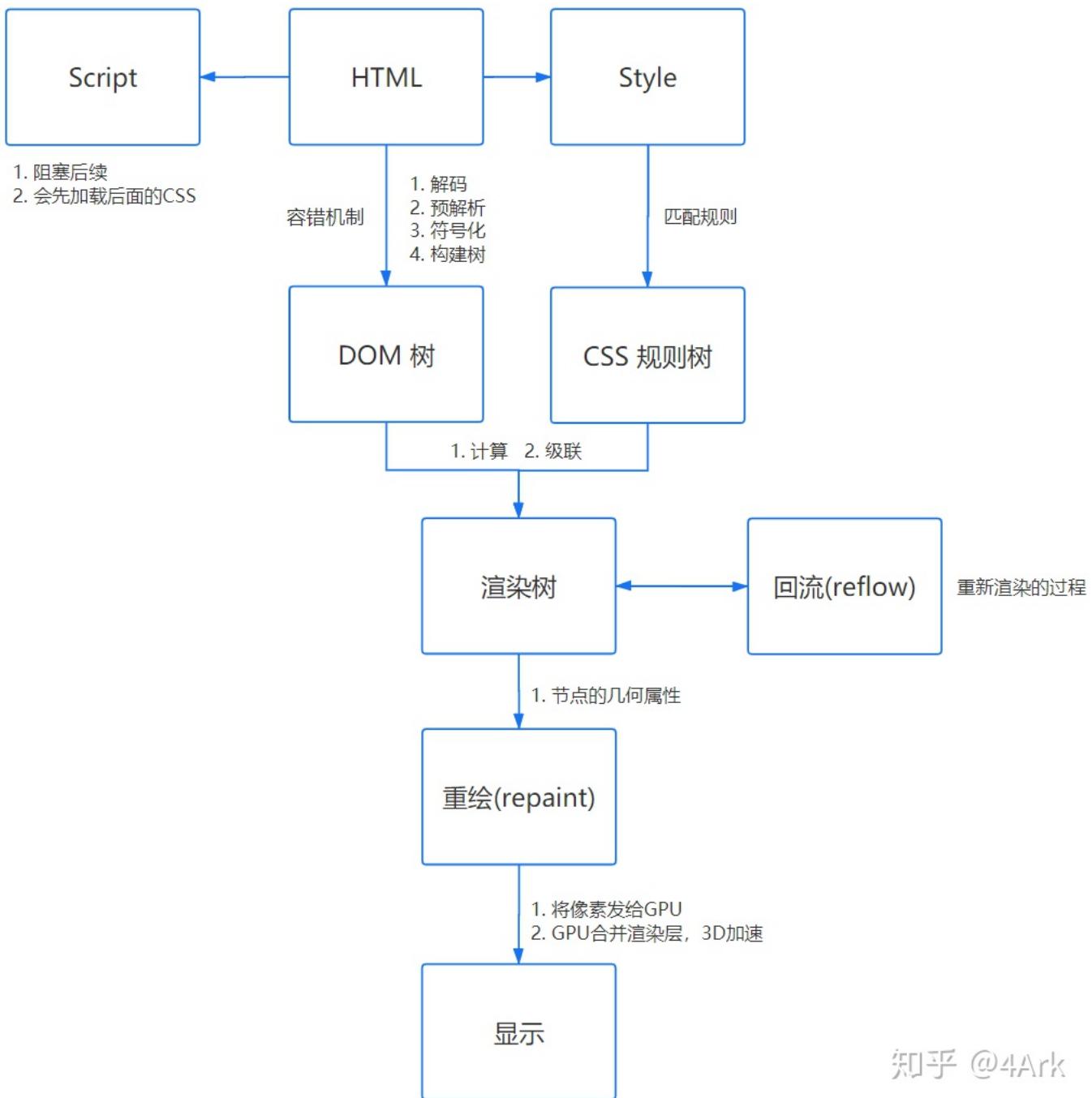


5) 客户端接收响应 对资源进行分析

6) 浏览器渲染页面

客户端拿到服务器传来的文件，找到HTML 和MIME文件，通过 MIME文件，浏览器知道页面渲染引擎来处理HTML文件

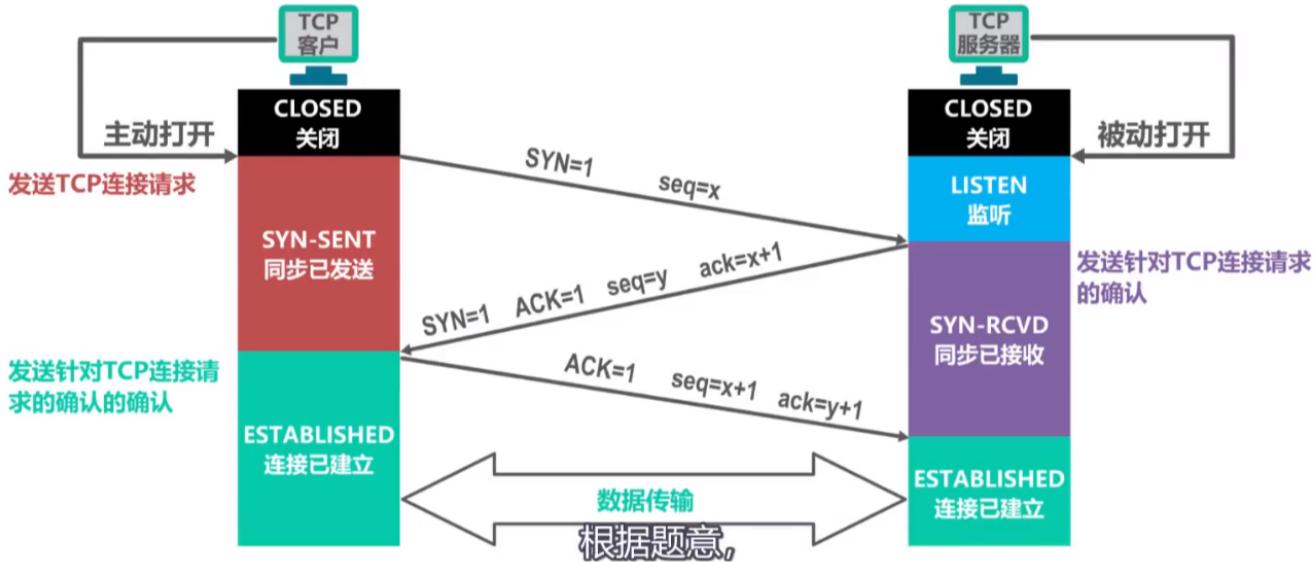
1. 浏览器会解析HTML源码，创建DOM树：在DOM 树中每一个HTML标签都有一个对应的结点，并且一个文本也有一个对应的文本结点
2. 浏览器解析css 代码，计算出最终的样式布局，形成css-tree
3. 渲染树：DOM树 和 css 规则树 合并的过程
4. 浏览器根据渲染树把页面绘制到屏幕上



知乎 @4Ark

2.说一下TCP三次握手

三次握手



第一次握手：客户端 A 主动打开连接 将 $SYN = 1$, $ACK = 0$, $seq = x$ 发送给服务器端 B, 请求发送后便进入 SYN-SENT 状态。

- $SYN = 1$, $ACK = 0$ 表示该报文段为请求报文
- x 为本次 TCP 通信的字节流的初始序号。

TCP 规定 $SYN = 1$ 的报文段不能有数据部分，但要消耗一个序号

第二次握手：服务端 B 收到连接请求报文后，如果同意连接，则会发送一个应答： $SYN = 1$, $ACK = 1$, $seq = y$, $ack = x + 1$ ；该应答发送完成后便进入 SYN-RCVD 状态

- $SYN = 1$, $ACK = 1$ 表示该报文段为连接同意的应答报文
- $seq = y$ 表示服务器 B 作为发送者，发送字节流的初始序号
- $ack = x + 1$ 表示服务器希望下一个数据报发送序号从 $x + 1$ 的字节开始

第三次握手：当客户端收到连接同意的应答后，还要向服务器 B 发送一个确认报文段，表示服务器端 B 发来的同意连接应答成功收到。 $ACK = 1$, $seq = x + 1$, $ack = y + 1$ ；此时客户端发完这个报文后进入 ESTABLISHED (Established: 建立) 状态，服务端收到这个应答后同样也进入 Established 状态。此时连接建立完成。

tcp 为什么要三次握手，两次不行吗？为什么？

为了实现可靠数据传输，TCP 协议的通信双方，**都必须维护一个序列号**，以标识发送出去的数据包中，哪些是已经被对方收到的。

三次握手的过程即是通信双方相互告知序列号起始值，并确认对方已经收到了序列号起始值的必经步骤。

为什么连接建立需要三次握手，而不是两次握手？

防止失效的连接请求报文段被服务器端接收，从而产生错误。

失效连接请求：若客户端发送的连接请求丢失，客户端等待应答超时后会再次发送连接请求，此时上一个连接请求是失败的！

这里会有人问，那么 A 第三次发送给 B 的信号丢失了呢？

TCP 不会为没有数据的 ack 超时重传，此时服务器端 B 会超时重传自己的 syn 同步号，直到收到 A 的 ack 为止

如果已经建立了连接，但是客户端突然出现故障了怎么办？(网络CLOSE)

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

三次握手连环炮

1. 三次握手的第一个包，即A发给B的SYN中途被丢，没有到达B会怎么样？ A会周期性超时重传，直到收到B的确认
2. 三次握手的第二个包，即B发给A的SYN+ACK中途被丢，没有到达A会怎么样？ B会周期性超时重传，直到收到A的确认
3. 三次握手的第三个包，即A发给B的ACK中途被丢，没有到达B会怎么样？

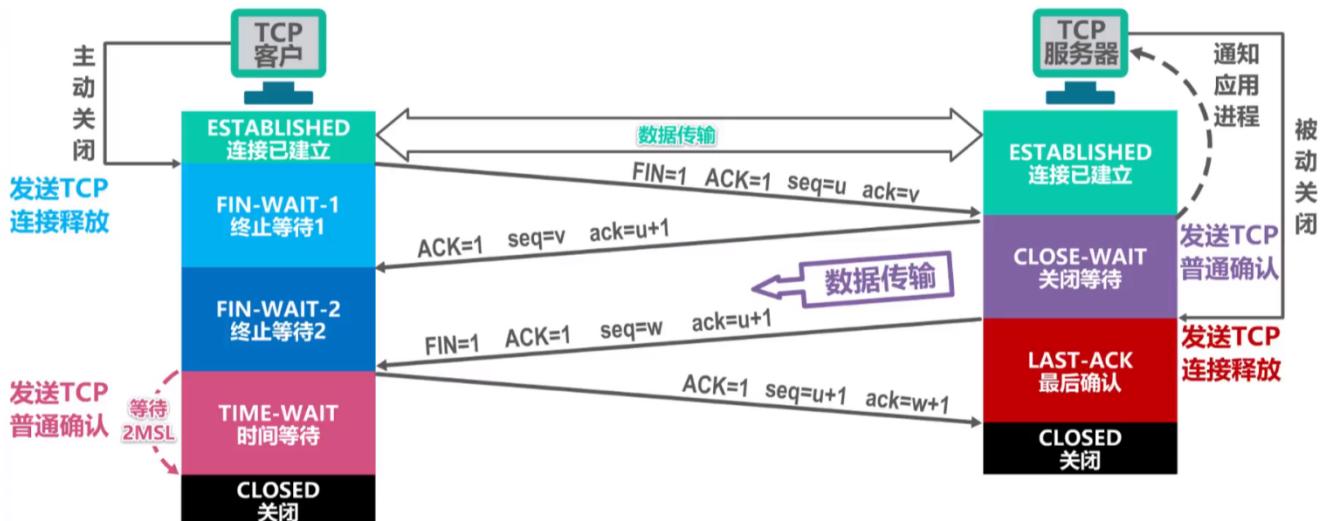
A发完ACK,单方面认为TCP为Established状态，而B显然认为TCP为Active状态：

- a.假定此时双方都没有数据发送，B会周期性超时重传，直到收到A的确认，收到之后B的TCP连接也为Established状态，双向可以发包。
- b.假定此时A有数据发送，B收到A的Data + ACK，自然会切换为established 状态，并接受A的数据。
- c.假定B有数据发送，数据发送不了，会一直周期性超时重传SYN+ACK，直到收到A的确认才可以发送数据.

三次握手过程中可以携带数据么？

第三次握手的时候，可以携带。前两次握手不能携带数据。

3.说一下 TCP 4次挥手？



MSL(Maximum Segment Lifetime)意思是最长报文段寿命，RFC793建议为2分钟。

TCP连接的释放一共需要四步，因此称为四次挥手，前两次挥手用于断开一个方向的连接，后两次挥手用于断开另一个方向的连接。

第一次挥手：若客户端 A 认为数据发送完成，则它需要向 B 发送连接释放请求。该请求只有报文头，头中携带主要参数：FIN = 1, seq = u 此时A 将进入 FIN-WAIT-1状态。

- FIN = 1 表示该报文是一个连接释放请求
- seq = u , u - 1是A 向 B 发送的最后一个字节的序号

第二次挥手：服务器端 B 收到连接释放请求后，会通知响应的应用程序，告诉他们 A 向 B 这个方向的连接已经释放。此时 B进入CLOSE-WAIT 状态，并向 A 发送连接释放的应答，其报文头含ACK = 1, seq = v, ack = u + 1

- ACK = 1：除TCP连接请求报文段以外，TCP 通信过程中所有数据包的ACK 都为 1，表示应答。
- seq = v : v - 1是服务器 B 向 A 发送的最后一个字节的序号

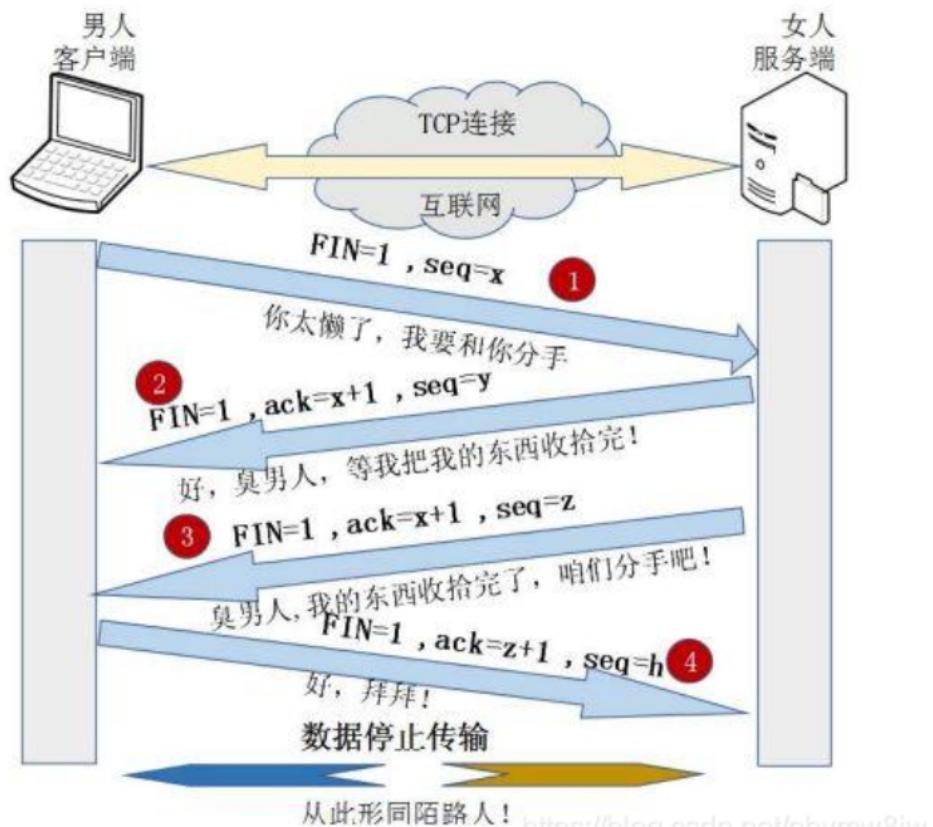
- $ack = u + 1$: 表示希望收到从 $u + 1$ 个字节开始的报文段，并且已经成功接收到了前 u 个字节。

A 收到该应答，进入FIN-WAIT-2 状态，等待B 发送连接释放请求。

第二次挥手完成后，A 到 B 方向的连接已经释放，B 不会在接收数据，A 也不会在发送数据。但B 到 A 方向的连接仍然存在，B 可以继续向 A 发送数据。

第三次挥手：当 B 向 A 发完所有数据后，向 A 发送连接释放请求，请求头：FIN = 1, ACK = 1, seq = w, ack = u + 1 B 变进入LAST-ACK 状态。

第四次挥手：A 收到释放请求后，向B 发送确认应答，此时A 进入TIME-WAIT状态。该状态会持续2MSL时间。若该时间段内没有B 的重发请求，就进入CLOSED状态，撤销TCB。当B收到确认应答后，也进入CLOSED状态，撤销TCB。



1.为什么A要先进入TIME-WAIT状态，等待2MSL时间后才进入CLOSED状态？

如果网络是不可靠的，有可能最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。

在Client发送出最后的ACK回复，但该ACK可能丢失。

Server如果没有收到ACK，将不断重复发送FIN片段。

所以Client不能立即关闭，它必须确认Server接收到了该ACK。Client会在发送出ACK之后进入到TIME_WAIT状态。Client会设置一个计时器，等待2MSL的时间。如果在该时间内再次收到FIN，那么Client会重发ACK并再次等待2MSL。所谓的2MSL是两倍的MSL(Maximum Segment Lifetime)。MSL指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

2.为什么连接的时候是三次握手，关闭的时候却是四次握手？

因为当Server端收到Client端的SYN连接请求报文后，**可以直接发送SYN+ACK报文**。其中ACK报文是用来应答的，SYN报文是用来同步的。**但是关闭连接时**，当Server端收到FIN报文时，很可能并不会立即关闭SOCKET，所以只能先回复一个ACK报文，告诉Client端，“你发的FIN报文我收到了”。**只有等到我Server端所有的报文都发送完了**，我才能发送FIN报文，因此不能一起发送。故需要四步握手

3. 状态为TIME_WAIT是不是所有执行主动关闭的socket都会进入TIME_WAIT状态呢？有没有什么情况使主动关闭的socket直接进入CLOSED状态呢？

主动关闭的一方在发送最后一个 ack 后就会进入 TIME_WAIT 状态 停留2MSL (max segment lifetime) 时间

这个是TCP/IP必不可少的，也就是“解决”不了的。也就是TCP/IP设计者本来是这么设计的 **主要有两个原因** 1。防止上一次连接中的包，迷路后重新出现，影响新连接（经过2MSL，上一次连接中所有的重复包都会消失）2。可靠的关闭TCP连接在主动关闭方发送的最后一个 ack(fin)，有可能丢失，这时被动方会重新发 fin，如果这时主动方处于 CLOSED 状态，就会响应 rst 而不是 ack。所以 主动方要处于 TIME_WAIT 状态，而不能是 CLOSED 。

等待2MSL的意义

如果不等待会怎样？

如果不等待，客户端直接跑路，当服务端还有很多数据包要给客户端发，且还在路上的时候，若客户端的端口此时刚好被新的应用占用，那么就接收到了无用数据包，造成数据包混乱。所以，最保险的做法是等服务器发来的数据包都死翘翘再启动新的应用。

那，照这样说一个 MSL 不就够了嘛，为什么要等待 2 MSL？

- 1 个 MSL 确保四次挥手主动关闭方最后的 ACK 报文最终能达到对端
- 1 个 MSL 确保对端没有收到 ACK 重传的 FIN 报文可以到达

这就是等待 2MSL 的意义。

为什么是四次挥手而不是三次？

因为服务端在接收到 FIN，往往不会立即返回 FIN，必须等到服务端所有的报文都发送完了，才能发 FIN。因此先发一个 ACK 表示已经收到客户端的 FIN，延迟一段时间才发 FIN。这就造成了四次挥手。

如果是三次挥手会有什么问题？

等于说服务端将 ACK 和 FIN 的发送合并为一次挥手，这个时候长时间的延迟可能会导致客户端误以为 FIN 没有到达客户端，从而让客户端不断的重发 FIN。

4. 说一下 tcp 粘包是怎么产生的？

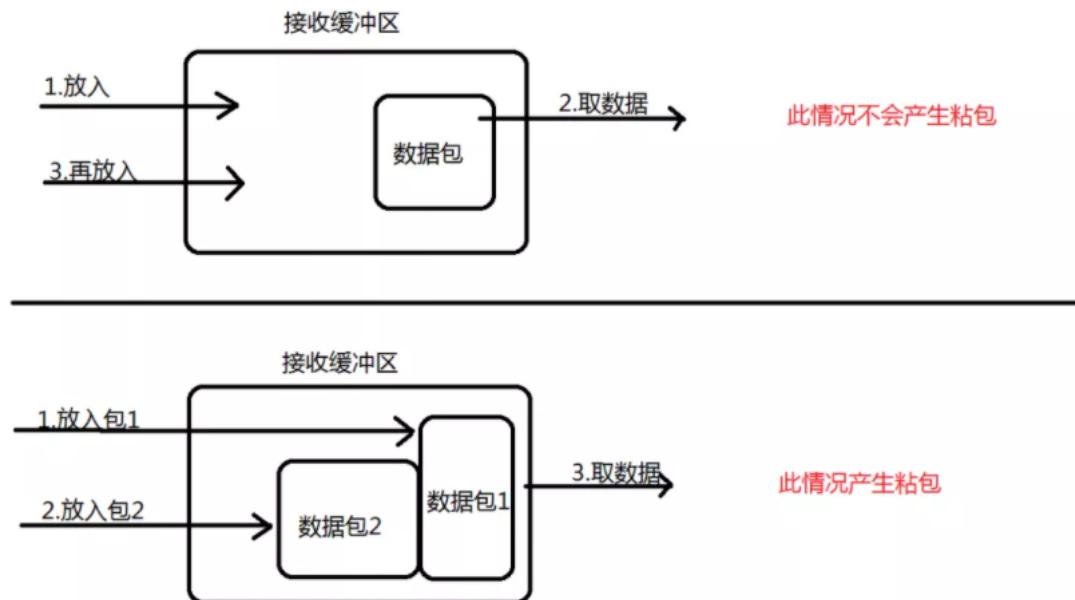
①. 发送方产生粘包

采用TCP协议传输数据的客户端与服务器经常是保持一个长连接的状态（一次连接发一次数据不存在粘包），双方在连接不断开的情况下，可以一直传输数据；但当发送的数据包过于的小时，那么TCP协议默认的会启用Nagle算法，将**这些较小的数据包进行合并发送**（缓冲区数据发送是一个堆压的过程）；这个合并过程就是在发送缓冲区中进行的，也就是说数据发送出来它已经是粘包的状态了。



②. 接收方产生粘包

接收方采用TCP协议接收数据时的过程是这样的：数据到底接收方，从网络模型的下方传递至传输层，传输层的TCP协议处理是将其放置**接收缓冲区**，然后由应用层来主动获取（C语言用recv、read等函数）；这时会出现一个问题，就是我们在程序中调用的读取数据函数不能及时的把缓冲区中的数据拿出来，而下一个数据又到来并有一部分放入的缓冲区末尾，等我们读取数据时就是一个粘包。（放数据的速度 > 应用层拿数据速度）



为什么会产生TCP粘包、拆包呢？

TCP 是以流的方式处理数据，一个完整的包可能会被TCP 拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP粘包/拆包的原因

应用程序写入的字节大小大于套接字发送的缓冲区的大小，会发生拆包。

应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，发生粘包。

5. forward 和 redirect 的区别？

Forward和Redirect代表了两种请求转发方式：直接转发和间接转发。

直接转发方式 (Forward)，客户端和浏览器只发出一次请求，Servlet、HTML、JSP或其它信息资源，由第二个信息资源响应该请求，在请求对象request中，保存的对象对于每个信息资源是共享的。

间接转发方式 (Redirect) 实际是**两次HTTP请求**，服务器端在响应第一次请求的时候，让浏览器再向另外一个URL发出请求，从而达到转发的目的。

举个通俗的例子：

直接转发就相当于：“A找B借钱，B说没有，B去找C借，借到借不到都会把消息传递给A”；

间接转发就相当于：“A找B借钱，B说没有，让A去找C借”。

6.计算机网络体系模型

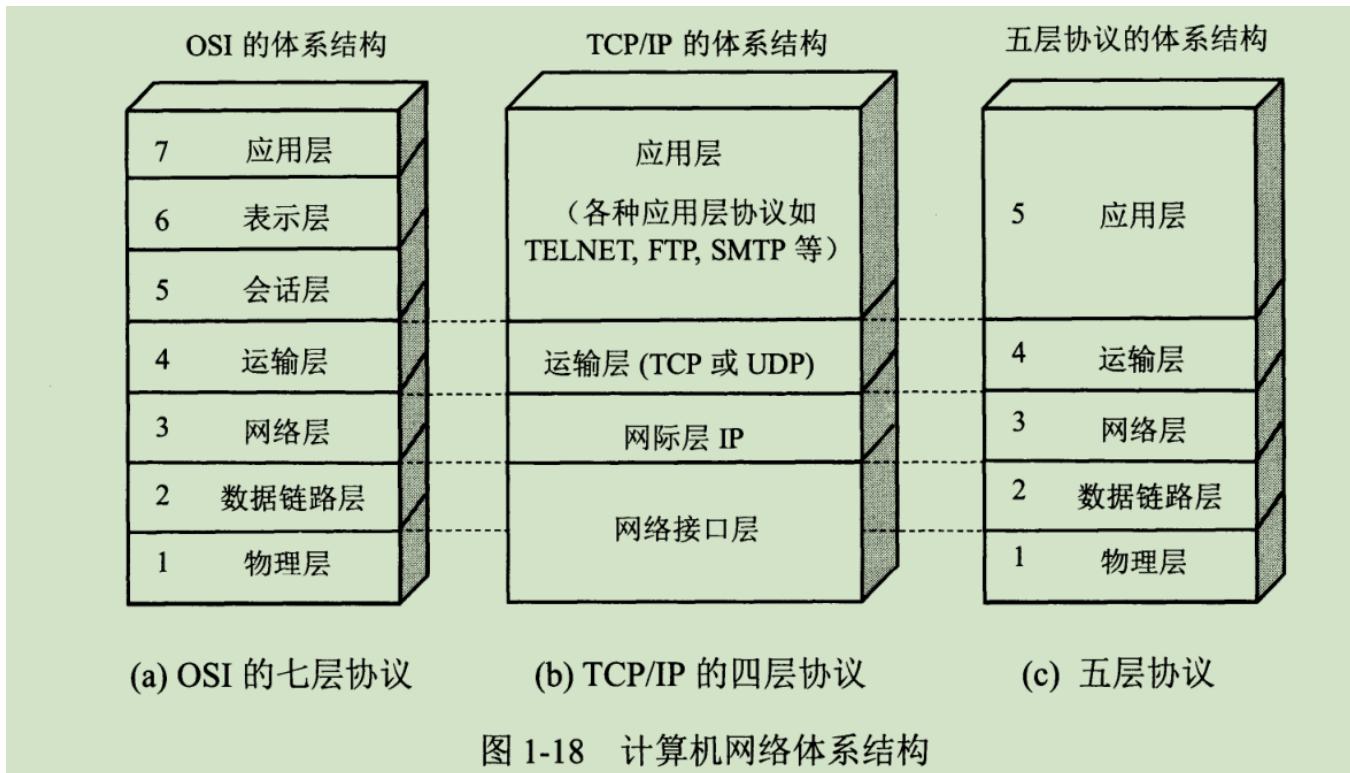
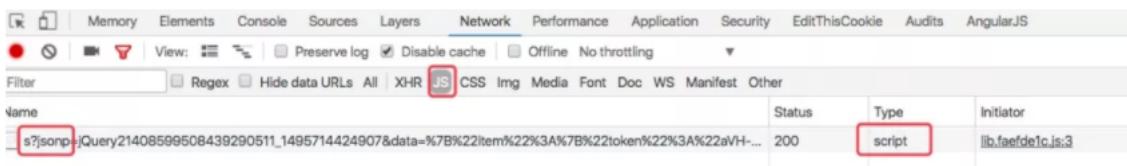


图 1-18 计算机网络体系结构

7.如何实现跨域

方式一：JSONP跨域

JSONP (JSON with Padding) 是数据格式JSON的一种“使用模式”，可以让网页从别的网域要数据。根据 XMLHttpRequest 对象受到同源策略的影响，而利用元素的这个开放策略，网页可以得到从其他来源动态产生的JSON数据，而这种使用模式就是所谓的 JSONP。用JSONP抓到的数据并不是JSON，而是任意的JavaScript，用 JavaScript解释器运行而不是用JSON解析器解析。所有，通过Chrome查看所有JSONP发送的Get请求都是js类型，而非XHR。



缺点：

- 只能使用Get请求
- 不能注册success、error等事件监听函数，不能很容易的确定JSONP请求是否失败
- JSONP是从其他域中加载代码执行，容易受到跨站请求伪造的攻击，其安全性无法确保

方式二：CORS

Cross-Origin Resource Sharing (CORS) 跨域资源共享是一份浏览器技术的规范，提供了 Web 服务从不同域传来沙盒脚本的方法，以避开浏览器的同源策略，确保安全的跨域数据传输。现代浏览器使用CORS在API容器如XMLHttpRequest来减少HTTP请求的风险来源。与 JSONP 不同，CORS 除了 GET 要求方法以外也支持其他的 HTTP 要求。服务器一般需要增加如下响应头的一种或几种：

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
```

跨域请求默认不会携带Cookie信息，如果需要携带，请配置下述参数：

```
"Access-Control-Allow-Credentials": true  
// Ajax设置  
"withCredentials": true
```

方式三：代理

同源策略是针对浏览器端进行的限制，可以通过服务器端来解决该问题

DomainA客户端（浏览器） ==> DomainA服务器 ==> DomainB服务器 ==> DomainA客户端（浏览器）

8.说一下JSONP 实现原理？

jsonp 即 json+padding，动态创建script标签，利用script标签的src属性可以获取任何域下的js脚本，通过这个特性(也可以说漏洞)，服务器端不在返货json格式，而是返回一段调用某个函数的js代码，在src中进行了调用，这样实现了跨域。

9.http 响应码 301 和 302 代表的是什么？有什么区别？

答：301，302 都是HTTP状态的编码，都代表着某个URL发生了转移。

区别：

- 301 redirect: 301 代表永久性转移(Permanently Moved)。
- 302 redirect: 302 代表暂时性转移(Temporarily Moved)。

☆算法☆

1.B树和B+树的区别

B树，B即balance 平衡的意思，B树的搜索从根节点开始，进行有序的二分查找，如果命中则结束，查找结果有可能在非叶子结点命中结束。

B+树 同样搜索从根节点开始，区别是所有关键字只存在叶子结点的链表中（也叫稠密索引），且链表的关键字是有序的。非叶子结点相当于叶子结点的索引也叫稀疏索引。B+树更适合文件索引。

2.你了解哪些排序算法么

快排、归并

快排：在序列中选择一个轴点元素（pivot），假设每次选择0位置为轴点元素，利用pivot 将序列分割成2个子序列。

- 将小于pivot 的放到pivot 的左面（前面），
- 将大于pivot 的放到轴点元素的后面（右面）。
- 等于pivot 的元素放哪边都可以。（这就决定了快排是不稳定排序算法）。

对子序列重复1 2 操作，直到不能在分割（子序列只剩下一个元素）

时间复杂度 $O(n \log n)$ ，最坏情况轴点左右元素分布不均匀时间复杂度为 $O(n^2)$ ；

空间复杂度 $\log n$

不稳定排序

归并

1. 不断将当前序列平均分割成2个子序列，直到不能在分割（序列中只剩一个元素）
2. 不断将2个子序列合并成一个有序序列，直到最终只剩下一个有序序列。

时间复杂度 $O(n \log n)$

空间复杂度 $O(n)$

3.解决hash冲突的方法

1. 开放地址法
2. 链地址法
3. 再哈希法
4. 建立一个公共溢出区