

目录

目录

必看

如何学习本项目

关于更新

如何贡献

关于本开源文档

如何赞赏

更新记录

一 备战面试

1.1 如何准备面试

1.1.1 如何获取大厂面试机会?

1.1.2 准备自己的自我介绍

1.1.3 搞清楚技术面可能会问哪些方向的问题

1.1.4 休闲着装即可

1.1.5 随身带上自己的成绩单和简历

1.1.6 如果需要笔试就提前刷一些笔试题

1.1.7 花时间一些逻辑题

1.1.8 准备好自己的项目介绍

1.1.9 提前准备技术面试

1.1.10 面试之前做好定向复习

1.1.11 面试之后记得复盘

1.2 程序员简历就该这样写

1.2.1 为什么说简历很重要?

1.2.2 关于简历你必须知道的点

1.2.3 写简历必须了解的两大法则

 STAR法则 (Situation Task Action Result)

 FAB 法则 (Feature Advantage Benefit)

1.2.4 项目经历怎么写?

1.2.5 专业技能该怎么写?

1.2.6 排版注意事项

1.2.7 其他的一些小tips

1.2.8 推荐的工具/网站

1.3 大部分程序员在面试前很关心的一些问题

1.3.1 我是双非/三本/专科学校的，我有机会进入大厂吗?

1.3.2 非计算机专业的学生能学好Java后台吗? 我能进大厂吗?

1.3.3 如何学好Java后端呢?

1.3.4 我没有实习经历的话找工作是不是特别艰难?

1.3.5 我该如何准备面试呢? 面试的注意事项有哪些呢?

1.3.6 我该自学还是报培训班呢?

1.3.7 没有项目经历/博客/Github开源项目怎么办?

 没有项目经验怎么办?

 没有博客怎么办?

 没有开源项目怎么办?

1.3.8 从招聘要求看大厂青睐什么样的人?

1.4 如何学习? 学会各种框架有必要吗?

1.4.1 我该如何学习?

1.4.2 学会各种框架有必要吗?

二 Java基础+集合+多线程+JVM

2.1 Java基础

1. 面向对象和面向过程的区别

2. Java 语言有哪些特点?

3. 关于 JVM JDK 和 JRE 最详细通俗的解答

JVM

JDK 和 JRE

4. Oracle JDK 和 OpenJDK 的对比
5. Java 和 C++的区别?
6. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同?
7. Java 应用程序与小程序之间有哪些差别?
8. 字符型常量和字符串常量的区别?
9. 构造器 Constructor 是否可被 override?
10. 重载和重写的区别
 - 重载
 - 重写
11. Java 面向对象编程三大特性: 封装 继承 多态
 - 封装
 - 继承
 - 多态
12. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?
13. 自动装箱与拆箱
14. 在一个静态方法内调用一个非静态成员为什么是非法的?
15. 在 Java 中定义一个不做事且没有参数的构造方法的作用
16. import java 和 javax 有什么区别?
17. 接口和抽象类的区别是什么?
18. 成员变量与局部变量的区别有哪些?
19. 创建一个对象用什么运算符? 对象实体与对象引用有何不同?
20. 什么是方法的返回值? 返回值在类的方法里的作用是什么?
21. 一个类的构造方法的作用是什么? 若一个类没有声明构造方法, 该程序能正确执行吗? 为什么?
22. 构造方法有哪些特性?
23. 静态方法和实例方法有何不同
24. 对象的相等与指向他们的引用相等, 两者有什么不同?
25. 在调用子类构造方法之前会先调用父类没有参数的构造方法, 其目的是?
26. == 与 equals(重要)
27. hashCode 与 equals (重要)
 - hashCode () 介绍
 - 为什么要有 hashCode
 - hashCode () 与 equals () 的相关规定
28. 为什么 Java 中只有值传递?
29. 简述线程、程序、进程的基本概念。以及他们之间关系是什么?
30. 线程有哪些基本状态?
- 31 关于 final 关键字的一些总结
- 32 Java 中的异常处理
 - Java 异常类层次结构图
 - Throwable 类常用方法
 - 异常处理总结
- 33 Java 序列化中如果有些字段不想进行序列化, 怎么办?
- 34 获取用键盘输入常用的两种方法
- 35 Java 中 IO 流
 - Java 中 IO 流分为几种?
 - 既然有了字节流, 为什么还要有字符流?
 - BIO,NIO,AIO 有什么区别?
36. 常见关键字总结:static,final,this,super
37. Collections 工具类和 Arrays 工具类常见方法总结
38. 深拷贝 vs 浅拷贝
- 参考
- 公众号

- 2.2.5 HashMap 和 Hashtable 的区别
 - 2.2.6 HashMap 和 HashSet区别
 - 2.2.7 HashSet如何检查重复
 - 2.2.8 HashMap的底层实现
 - JDK1.8之前
 - JDK1.8之后
 - 2.2.9 HashMap 的长度为什么是2的幂次方
 - 2.2.10 HashMap 多线程操作导致死循环问题
 - 2.2.11 ConcurrentHashMap 和 Hashtable 的区别
 - 2.2.12 ConcurrentHashMap线程安全的具体实现方式/底层具体实现
 - JDK1.7 (上面有示意图)
 - JDK1.8 (上面有示意图)
 - 2.2.13 comparable 和 Comparator的区别
 - Comparator定制排序
 - 重写compareTo方法实现按年龄来排序
 - 2.2.14 集合框架底层数据结构总结
 - Collection
 - Map
 - 2.2.15 如何选用集合?
- 2.3 多线程
- 2.3.1. 什么是线程和进程?
 - 何为进程?
 - 何为线程?
 - 2.3.2. 请简要描述线程与进程的关系,区别及优缺点?
 - 图解进程和线程的关系
 - 程序计数器为什么是私有的?
 - 虚拟机栈和本地方法栈为什么是私有的?
 - 一句话简单了解堆和方法区
 - 2.3.3. 说说并发与并行的区别?
 - 2.3.4. 为什么要使用多线程呢?
 - 2.3.5. 使用多线程可能带来什么问题?
 - 2.3.6. 说说线程的生命周期和状态?
 - 2.3.7. 什么是上下文切换?
 - 2.3.8. 什么是线程死锁?如何避免死锁?
 - 认识线程死锁
 - 如何避免线程死锁?
 - 2.3.9. 说说 sleep() 方法和 wait() 方法区别和共同点?
 - 2.3.10. 为什么我们调用 start() 方法时会执行 run() 方法,为什么我们不能直接调用 run() 方法?
 - 2.3.11 synchronized 关键字
 - 1.说一说自己对于 synchronized 关键字的了解
 - 2.说说自己是怎么使用 synchronized 关键字,在项目中用到了吗
 - 3.synchronized关键字最主要的三种使用方式
 - 4.讲一下 synchronized 关键字的底层原理
 - 5.说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化,可以详细介绍下这些优化吗
 - 6.谈谈 synchronized和ReentrantLock 的区别
 - 2.3.12 volatile关键字
 - 1. 讲一下Java内存模型
 - 2 并发编程的三个重要特性
 - 3. 说说 synchronized 关键字和 volatile 关键字的区别
 - 2.3.13 ThreadLocal
 - 1. ThreadLocal简介
 - 2. ThreadLocal示例
 - 3. ThreadLocal原理
 - 4. ThreadLocal 内存泄露问题
 - 2.3.14 线程池
 - 1. 为什么要用线程池?
 - 2. 实现Runnable接口和Callable接口的区别
 - 3. 执行execute()方法和submit()方法的区别是什么呢?

- 4. 如何创建线程池
- 5. ThreadPoolExecutor 类分析
 - ThreadPoolExecutor 构造函数重要参数分析
 - ThreadPoolExecutor 饱和策略
- 6. 一个简单的线程池Demo: Runnable + ThreadPoolExecutor
- 7. 线程池原理分析
- 2.3.15 Atomic 原子类
 - 1. 介绍一下Atomic 原子类
 - 2. JUC 包中的原子类是哪4类?
 - 3. 讲讲 AtomicInteger 的使用
 - 4. 能不能给我简单介绍一下 AtomicInteger 类的原理
- 2.3.16 AQS
 - 1. AQS 介绍
 - 2. AQS 原理分析
 - AQS 原理概览
 - AQS 对资源的共享方式
 - AQS底层使用了模板方法模式
 - 3. AQS 组件总结
- Reference
- 2.4 JVM
 - 2.4.1 介绍下 Java 内存区域(运行时数据区)
 - 程序计数器
 - Java 虚拟机栈
 - 本地方法栈
 - 堆
 - 方法区
 - 方法区和永久代的关系
 - 常用参数
 - 为什么要将永久代(PermGen)替换为元空间(MetaSpace)呢?
 - 运行时常量池
 - 直接内存
 - 2.4.2 说一下Java对象的创建过程
 - 2.4.3 对象的访问定位有哪两种方式?
 - 2.4.4 说一下堆内存中对象的分配的基本策略
 - 2.4.5 Minor Gc和Full GC 有什么不同呢?
 - 2.4.6 如何判断对象是否死亡?(两种方法)
 - 引用计数法
 - 可达性分析算法
 - 2.4.7 简单的介绍一下强引用,软引用,弱引用,虚引用
 - 强引用(StrongReference)
 - 软引用(SoftReference)
 - 弱引用(WeakReference)
 - 2.4.8 如何判断一个常量是废弃常量?
 - 2.4.9 如何判断一个类是无用的类?
 - 2.4.10 垃圾收集有哪些算法,各自的特点?
 - 标记-清除算法
 - 复制算法
 - 标记-整理算法
 - 分代收集算法
 - 2.4.11 HotSpot为什么要分为新生代和老年代?
 - 2.4.12 常见的垃圾回收器有那些?
 - Serial收集器
 - ParNew收集器
 - Parallel Scavenge收集器
 - Serial Old收集器
 - Parallel Old收集器
 - CMS收集器
 - G1收集器

2.4.13 类文件结构

介绍一下类文件结构吧!

2.4.14 类加载过程

知道类加载的过程吗?

那加载这一步做了什么?

知道哪些类加载器?

双亲委派模型知道吗? 能介绍一下吗?

双亲委派模型介绍

双亲委派模型实现源码分析

双亲委派模型带来了什么好处呢?

如果我们不想用双亲委派模型怎么办?

如何自定义类加载器?

三 计算机基础

3.1 计算机网络

3.1.1 OSI与TCP/IP各层的结构与功能,都有哪些协议?

应用层

运输层

网络层

数据链路层

物理层

总结一下

3.1.2 TCP 三次握手和四次挥手(面试常客)

TCP 三次握手漫画图解

为什么要三次握手

为什么要传回 SYN

传了 SYN,为啥还要传 ACK

为什么要四次挥手

3.1.2 TCP,UDP 协议的区别

3.1.3 TCP 协议如何保证可靠传输

3.1.4 ARQ协议

停止等待ARQ协议

连续ARQ协议

3.1.5 滑动窗口和流量控制

3.1.6 拥塞控制

3.1.7 在浏览器中输入url地址 => 显示主页的过程(面试常客)

3.1.8 状态码

3.1.9 各种协议与HTTP协议之间的关系

3.1.10 HTTP长连接,短连接

3.1.11 HTTP是不保存状态的协议,如何保存用户状态?

3.1.12 Cookie的作用是什么?和Session有什么区别?

3.1.13 HTTP 1.0和HTTP 1.1的主要区别是什么?

3.1.12 URI和URL的区别是什么?

3.1.13 HTTP 和 HTTPS 的区别?

建议

参考

3.2 数据结构

3.2.1 Queue

什么是队列

队列的种类

Java 集合框架中的队列 Queue

推荐文章

3.2.2 Set

什么是 Set

补充: 有序集合与无序集合说明

HashSet 和 TreeSet 底层数据结构

推荐文章

3.2.3 List

什么是List

List的常见实现类

ArrayList 和 LinkedList 源码学习

推荐阅读

3.2.4 Map

3.2.5 树

 1. 二叉树

 2. 完全二叉树

 3. 满二叉树

 4. 堆

 5. 二叉查找树 (BST)

 6. 平衡二叉树 (Self-balancing binary search tree)

 7. 红黑树

 8. B-, B+, B*树

 9. LSM 树

3.2.6 图

3.2.7 BFS及DFS

3.3 算法

3.3.1 几道常见的字符串算法题总结

 KMP 算法

 替换空格

3.3.2 最长公共前缀

3.3.3 回文串

 最长回文串

 验证回文串

 最长回文子串

 最长回文子序列

 括号匹配深度

 把字符串转换成整数

3.3.4 两数相加

 题目描述

 问题分析

 Solution

3.3.5 翻转链表

 题目描述

 问题分析

 Solution

3.3.6 链表中倒数第k个节点

 题目描述

 问题分析

 Solution

3.3.7 删除链表的倒数第N个节点

 问题分析

 Solution

3.3.8 合并两个排序的链表

3.3.9 剑指offer部分编程题

 斐波那契数列

3.3.10 跳台阶问题

 题目描述:

 问题分析:

 示例代码:

3.3.11 变态跳台阶问题

 题目描述:

 问题分析:

 示例代码:

 补充:

3.3.12 二维数组查找

 题目描述:

 问题解析:

 示例代码:

3.3.13 替换空格

题目描述:

问题分析:

示例代码:

3.3.14 数值的整数次方

题目描述:

问题解析:

示例代码:

3.3.15 调整数组顺序使奇数位于偶数前面

题目描述:

问题解析:

示例代码:

3.3.16 链表中倒数第k个节点

题目描述:

问题分析:

考察内容:

示例代码:

3.3.17 反转链表

考察内容:

示例代码:

3.3.18 合并两个排序的链表

题目描述:

问题分析:

考察内容:

示例代码:

3.3.19 用两个栈实现队列

题目描述:

问题分析:

考察内容:

示例代码:

3.3.20 栈的压入,弹出序列

题目描述:

题目分析:

考察内容:

示例代码:

3.4 操作系统

一 操作系统基础

1.1 什么是操作系统?

1.2 系统调用

二 进程和线程

2.1 进程和线程的区别

2.2 进程有哪几种状态?

2.3 进程间的通信方式

2.4 线程间的同步的方式

2.5 进程的调度算法

三 操作系统内存管理基础

3.1 内存管理介绍

3.2 常见的几种内存管理机制

3.3 快表和多级页表

快表

多级页表

总结

3.4 分页机制和分段机制的共同点和区别

3.5 逻辑(虚拟)地址和物理地址

3.6 CPU 寻址了解吗?为什么需要虚拟地址空间?

四 虚拟内存

4.1 什么是虚拟内存(Virtual Memory)?

4.2 局部性原理

4.3 虚拟存储器

4.4 虚拟内存的技术实现

4.5 页面置换算法

Reference

四 数据库面试题总结

4.1 MySQL

4.1.1 精品推荐

书籍推荐

文字教程推荐

相关资源推荐

视频教程推荐

常见问题总结

4.1.2 什么是MySQL?

4.1.3 存储引擎

一些常用命令

MyISAM和InnoDB区别

4.1.4 字符集及校对规则

4.1.5 索引

4.1.6 查询缓存的使用

4.1.7 什么是事务?

4.1.8 事物的四大特性(ACID)

4.1.9 并发事务带来哪些问题?

4.1.10 事务隔离级别有哪些?MySQL的默认隔离级别是?

4.1.11 锁机制与InnoDB锁算法

4.1.12 大表优化

限定数据的范围

读/写分离

垂直分区

水平分区

4.1.13 解释一下什么是池化设计思想。什么是数据库连接池?为什么需要数据库连接池?

4.1.14 分库分表之后,id 主键如何处理?

4.1.15 一条SQL语句在MySQL中如何执行的

4.1.16 MySQL高性能优化规范建议

4.1.17一条SQL语句执行得很慢的原因有哪些?

4.1.19 后端程序员必备: 书写高质量SQL的30条建议

4.2 Redis

4.2.1 redis 简介

为什么要用 redis/为什么要用缓存

为什么要用 redis 而不用 map/guava 做缓存?

4.2.2 redis 的线程模型

4.2.3 redis 和 memcached 的区别

4.2.4 redis 常见数据结构以及使用场景分析

String

Hash

List

Set

Sorted Set

4.2.5 redis 设置过期时间

4.2.6 redis 内存淘汰机制(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?)

4.2.7 redis 持久化机制(怎么保证 redis 挂掉之后再重启数据可以进行恢复)

4.2.8 redis 事务

4.2.9 缓存雪崩和缓存穿透问题解决方案

缓存雪崩

缓存穿透

4.2.10 如何解决 Redis 的并发竞争 Key 问题

4.2.11 如何保证缓存与数据库双写时的数据一致性?

参考

五 常用框架面试题总结

5.1 Spring面试题总结

5.1.1. 什么是 Spring 框架?

5.1.2 列举一些重要的Spring模块?

5.1.3 @RestController vs @Controller

5.1.4 Spring IOC & AOP

 谈谈自己对于 Spring IoC 和 AOP 的理解

 IoC

 AOP

 Spring AOP 和 AspectJ AOP 有什么区别?

5.1.5 Spring bean

 Spring 中的 bean 的作用域有哪些?

 Spring 中的单例 bean 的线程安全问题了解吗?

 @Component 和 @Bean 的区别是什么?

 将一个类声明为Spring的 bean 的注解有哪些?

 Spring 中的 bean 生命周期?

5.1.6 Spring MVC

 说自己对于 Spring MVC 了解?

 SpringMVC 工作原理了解吗?

5.1.7 Spring 框架中用到了哪些设计模式?

5.1.8 Spring 事务

 Spring 管理事务的方式有几种?

 Spring 事务中的隔离级别有哪几种?

 Spring 事务中哪几种事务传播行为?

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

5.1.9 JPA

 如何使用JPA在数据库中非持久化一个字段?

参考

公众号

5.2 MyBatis面试题总结

5.2.1 #{}和\${}的区别是什么?

5.2.2 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?

5.2.3 最佳实践中, 通常一个 Xml 映射文件, 都会写一个 Dao 接口与之对应, 请问, 这个 Dao 接口的工作原理是什么? Dao 接口里的方法, 参数不同时, 方法能重载吗?

5.2.4 Mybatis 是如何进行分页的? 分页插件的原理是什么?

 5.2.5 简述 Mybatis 的插件运行原理, 以及如何编写一个插件。

5.2.6 Mybatis 执行批量插入, 能返回数据库主键列表吗?

5.2.7 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?

5.2.8 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?

5.2.9 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。

5.2.10 Mybatis 是否支持延迟加载? 如果支持, 它的实现原理是什么?

5.2.11 Mybatis 的 Xml 映射文件中, 不同的 Xml 映射文件, id 是否可以重复?

5.2.12 Mybatis 中如何执行批处理?

5.2.13 Mybatis 都有哪些 Executor 执行器? 它们之间的区别是什么?

5.2.14 Mybatis 中如何指定使用哪一种 Executor 执行器?

5.2.15 Mybatis 是否可以映射 Enum 枚举类?

5.2.16 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?

5.2.17 简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系?

5.2.18 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?

5.3 Kafka面试题总结

5.3.1 Kafka 是什么? 主要应用场景有哪些?

5.3.2 和其他消息队列相比,Kafka的优势在哪里?

5.3.3 队列模型了解吗? Kafka 的消息模型知道吗?

 队列模型: 早期的消息模型

 发布-订阅模型:Kafka 消息模型

5.3.4 什么是Producer、Consumer、Broker、Topic、Partition?

5.3.5 Kafka 的多副本机制了解吗? 带来了什么好处?

5.3.6 Zookeeper 在 Kafka 中的作用知道吗?

5.3.7 Kafka 如何保证消息的消费顺序?

5.3.8 Kafka 如何保证消息不丢失

 生产者丢失消息的情况

消费者丢失消息的情况
Kafka 弄丢了消息

5.3.9 Kafka 如何保证消息不重复消费
Reference

5.4 Netty 面试题总结

5.4.1 Netty 是什么?

5.4.2 为什么要用 Netty?

5.4.3 Netty 应用场景了解么?

5.4.4 Netty 核心组件有哪些? 分别有什么作用?

- 1.Channel
- 2.EventLoop
- 3.ChannelFuture
- 4.ChannelHandler 和 ChannelPipeline

5.4.5 EventloopGroup 了解么?和 EventLoop 哪关系?

5.4.6 Bootstrap 和 ServerBootstrap 了解么?

5.4.7 NioEventLoopGroup 默认的构造函数会起多少线程?

5.4.8 Netty 线程模型了解么?

5.4.9 Netty 服务端和客户端的启动过程了解么?

- 服务端
- 客户端

5.4.10 什么是 TCP 粘包/拆包?有什么解决办法呢?

5.4.11 Netty 长连接、心跳机制了解么?

5.4.12 Netty 的零拷贝了解么?

参考

六 认证授权

6.1 认证授权面试题总结

6.1.1 认证 (Authentication) 和授权 (Authorization)的区别是什么?

6.1.2 什么是Cookie ? Cookie的作用是什么?如何在服务端使用 Cookie ?
什么是Cookie ? Cookie的作用是什么?
如何在服务端使用 Cookie 呢?

6.1.3 Cookie 和 Session 有什么区别? 如何使用Session进行身份验证?

6.1.4 如果没有Cookie的话Session还能用吗?

6.1.5 为什么Cookie 无法防止CSRF攻击, 而token可以?

6.1.6 什么是 Token?什么是 JWT?如何基于Token进行身份验证?

6.1.7 什么是OAuth 2.0?

6.1.8 什么是 SSO?

6.1.9 SSO与OAuth2.0的区别

参考

七 优质面经

五面阿里,终获offer

前言

一面(技术面)

二面(技术面)

三面(技术面)

四面(半个技术面)

五面(HR面)

总结

蚂蚁金服实习生面经总结

一面 (37 分钟左右)

二面 (33 分钟左右)

三面 (46 分钟)

HR 面

Bigo的Java面试, 我挂在了第三轮技术面上.....

背景

个人情况

Bigo面试

一面(微信电话面)

二面

三面

[总结](#)

[2020年字节跳动面试总结](#)

[基本条件](#)

[一面](#)

[二面](#)

[hr 面](#)

[总结](#)

[2019年蚂蚁金服、头条、拼多多的面试总结](#)

[准备过程](#)

[蚂蚁金服](#)

[一面](#)

[二面](#)

[三面](#)

[四面](#)

[五面](#)

[小结](#)

[拼多多](#)

[面试前](#)

[一面](#)

[二面](#)

[三面](#)

[小结](#)

[字节跳动](#)

[面试前](#)

[一面](#)

[二面](#)

[小结](#)

[总结](#)

逆风而行！从考研失败到收获到自己满意的Offer，分享一下自己的经历！

[个人情况](#)

[考研](#)

[准备春招](#)

[SHEIN面经分享](#)

[一面\(45min左右\)](#)

[二面 \(1h左右\)](#)

[三面 \(25min左右\)](#)

[四面 \(CTO面 时间很短，不到5分钟\)](#)

[五面 HR面](#)

[写在最后](#)

Java后端实习面经，电子科大大三读者投稿！看了之后感触颇深！很感动开心！

[关于我](#)

[准备面试](#)

[面试真题](#)

[数据结构与算法篇](#)

[计算机网络篇](#)

[操作系统篇](#)

[数据库篇](#)

[Java基础篇](#)

[写在最后](#)

八 微服务/分布式

九 真实大厂面试现场

[我和阿里面试官的一次邂逅\(上\)](#)

[自我介绍](#)

[项目介绍](#)

[消息队列](#)

[Redis](#)

[计算机网络](#)

[Java基础](#)

我和阿里面试官的一次邂逅(下)

操作系统

内存管理机制主要是做什么？

操作系统的内存管理机制了解吗？内存管理有哪几种方式？

分页机制和分段机制对比

逻辑地址和物理地址

进程和线程

多线程

为什么要使用多线程？

多线程死锁

从实现一个线程安全的单例模式看synchronized和volatile的使用

从CPU缓存模型聊到JMM(Java内存模型)

CPU缓存模型

JMM(Java内存模型)

synchronized关键字介绍

synchronized vs volatile

用过 CountDownLatch 么？什么场景下用的？

Netty

Netty 介绍

为什么要用 Netty？

Netty 应用场景

TCP 粘包/拆包以及解决办法

Netty线程模型

Netty 的零拷贝

Reference

十 开源项目推荐

Java教程类开源项目推荐

JavaGuide

CS-Notes

advanced-java

miaosha

architect-awesome

toBeTopJavaer

technology-talk

JavaFamily

JCSprout

fullstack-tutorial

附加5个不错的开源项目

Leetcode题解

1.CS-Notes

2.LeetCodeAnimation

3.leetcode

4.LeetCode-Solution-in-Good-Style

必看

如何学习本项目

提供了非常详细的目录，建议可以从头看是看一遍，如果基础不错的话也可以挑自己需要的章节查看。看的过程中自己要多思考，碰到不懂的地方，自己记得要勤搜索，需要记忆的地方也不要吝啬自己的脑子。

关于更新

《JavaGuide面试突击版》 预计一个月左右会有一次内容更新和完善，大家在我的公众号 **JavaGuide** 后台回复“面试突击” 即可获取最新版！另外，为了保证自己的辛勤劳动不被恶意盗版滥用，所以我添加了水印并且在一些内容注明版权，希望大家理解。



如何贡献

大家阅读过程中如果遇到错误的地方可以通过微信：JavaGuide1996 或者邮箱：koushuangbwcx@163.com 与我交流（ps：加过我微信的就不要重复添加了，这是另外一个账号，前一个已经满了）。

希望大家给我提反馈的时候可以按照如下格式：

我觉得2.3节Java基础的 2.3.1 这部分的描述有问题，应该这样描述：～巴拉巴拉～ 会更好！具体可以参考Oracle 官方文档，地址：~~~~。

为了提高准确性已经不必要的时间花费，希望大家尽量确保自己想法的准确性。

关于本开源文档

JavaGuide 目前已经 70k+ Star，目前已经是所有 Java 类别项目中 Star 数量第二的开源项目了。Star 虽然很多，但是价值远远比不上 Dubbo 这些开源项目，希望以后可以多出现一些这样的国产开源项目。国产开源项目！加油！奥利给！

随着越来越多的人参与完善这个项目，这个专注“Java知识总结+面试指南”项目的知识体系和内容的不断完善。JavaGuide 目前包括下面这两部分内容：

1. **Java 核心知识总结；**
2. **面试方向：**面试题、面试经验、备战面试系列文章以及面试真实体验系列文章

内容的庞大让JavaGuide 显得有一点臃肿。所以，我决定将专门为 Java 面试所写的文章以及来自读者投稿的文章整理成《JavaGuide面试突击版》系列，起这个名字也犹豫了很久，大家如果有更好的名字的话也可以向我建议。暂时的定位是将其作为 PDF 电子书，并不会像 JavaGuide 提供在线阅读版本。我之前也免费分享过PDF 版本的《Java面试突击》，期间一共更新了 3 个版本，但是由于后面难以同步和订正所以就没有再更新。《JavaGuide面试突击版》 pdf 版由于我工作流程的转变可以有效避免这个问题。

另外，这段时间，向我提这个建议的读者也不是一个两个，我自己当然也有这个感觉。只是自己一直没有抽出时间去做罢了！毕竟这算是一个比较耗费时间的工程。

这件事情具体耗费时间的地方是内容的排版优化（为了方便导出PDF生成目录），导出 PDF 我是通过 Typora 来做的。

如何赞赏

如果觉得本文档对你有帮助的话，欢迎加入我的知识星球。创建星球的目的主要是为了提高知识沉淀，微信群的弊端相比大家都了解。星球没有免费的原因是设立了门槛，提高进入读者的质量。我会在星球回答大家的问题，更新更多的大厂面试干货！

知识星球



Guide哥

送你一张星球优惠券

立减

¥ 23

可用于

JavaGuide读者圈

2020/06/17 12:00 至 2020/08/31 12:00

前 5000 名加入可用 ▶

长按二维码立抢优惠



我的知识星球的价格应该是我了解的圈子里面最低的，也就1顿饭钱吧！毕竟关注我的大部分还是学生，我打心底里希望自己分享的东西能对大家有帮助。



Guide哥

2020/2/26

余生是你J 提问： guide兄，最近学完ssm挺长一段时间了，也看了spring实战中相关IOC、AOP和springMVC部分，也有写一些管理系统的demo，但不知道下一步该学什么，是学springboot、Redis等；还是数据结构、设计模式，现在有在学着数据结构。本人19毕业机械转行，现在想法是学完找工作，希望guide兄指点。

不知道你接触 Docker 了没有，Git 用的如何？Git 和 Docker 都是非常有必要先学习的。

然后你就可以上手 Spring Boot，把 Spring Boot 好好学一下很有必要！学习 Spring Boot 的时候多实践一下多看看别人的优秀源码。学习 Spring Boot 的时候你就就会接触如何使用 Spring Boot 整合 Redis、Kafka 这些东西，到了这个时候再看一下 Redis、Kafka 这些东西。两者同步进行，我觉得也是木有毛病的啊，也算是带着目的性去学习 Spring Boot 和 Redis 这些了。

展开全部



查看详情 >

郭帅印、她、Always on!*、hz、KrisWu、南笙-北执、余生是你J、意无尽、阿嫖、皮皮西里的kk 等13人觉得很赞

Nature：相较于zuul，gateway用的少些吗哥

2020/2/27

余生是你J：感谢guide哥的指点👍 微服务的内容往后推推，学完spring boot后，做项目，然后巩固一下自己的基础，算法，设计模式这些👀

2020/2/27

Guide哥 回复 Nature：Spring Cloud Gateway 用的多点吧！。Zuul 2 虽然发布了，Spring 生态系统 还没有将他加进去。

2020/2/27

Guide哥 回复 余生是你J：赞！我的微信bwcx9393，老哥有问题可以直接微

Guide哥

创建124天



JavaGuide读者圈

JavaGuide 忠实读者交流社区。建知识星球的目的主要是做知识沉淀以及微信群的无门槛进入机制让群里的人鱼龙混... 展开



本星球已开放续费券功能

点击抢先体验 >

精华

45

机械工业黑皮书系列 B 站视频 《现代操作系统》：♂ 操作系统原理(北京大学)_哔哩哔哩 (゜-゜)つロ 干杯~-bilibili 《算法...

匿名用户 提问：请问对程序员兼职什么看法?最近想做兼职一直找不到门路

本文来自读者 Boyn 投稿！恭喜这位粉丝拿到了含金量极高的字节跳动实习 offer！赞！基本条件本人是底层 211 本科,现在才...

更新记录

V1.0-2020-03-07

第一版《JavaGuide面试突击版》正式完结发布！

V1.1-2020-03-13

修复问题：

- 每个章节都重复一遍目录，多滑了好多页
- 强烈要求加上版本号和发布日期，读者就知道自己的是什么版本了
- 2.1 Java基础部分 p36+p37文章链接失效
- 3.3 节 ThreadLocal 部分的一个笔误
- 水印过重，有一点影响阅读
- 文档名字开头加上版本表示示例：V1.1-JavaGuide面试突击版

增加/修改内容：

- 一备战面试部分：完善了“自我介绍”部分的内容并且增加技术面可能会问哪些方向的问题、如何学习等内容。

第三节常见框架部分增加了 Kafka 常见面试题

V2.0-2020-04-02

修复问题：

- 修复了部分错别字，这部分对整体阅读影响不大所以不做过多阐述。
- 增加了页码

增加/修改内容：

- Java基础知识部分自动拆装箱添加了一个参考文章。
- 提供了在线阅读版本：<https://snailclimb.gitee.io/javaguide-interview/#/>
- 计算机基础这一章节增加了：操作系统常见问题总结，这篇文章也更新在了公众号：[我和面试官之间关于操作系统的一场对弈！写了很久，希望对你有帮助！](#)

V3.0-2020-06-16

- 修复多处部分读者提到了笔误
 - 第九章- 真实大厂面试现场 增加了 [我和阿里面试官的一次邂逅\(下\)](#) (一篇花了Guide很多时间的文章，发在公众号上阅读不是蛮好，绝对干货~~~)
 - 增加万众期待的 **Netty** 常见面试题总结
 - 增加Java面试相关的开源项目
 - 增加算法类面试相关的开源项目
-

一 备战面试

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

不论是校招还是社招都避免不了各种面试、笔试，如何去准备这些东西就显得格外重要。不论是笔试还是面试都是有章可循的，我这个“有章可循”说的意思只是说应对技术面试是可以提前准备。我其实特别不喜欢那种临近考试就提前背啊记啊各种题的行为，非常反对！我觉得这种方法特别极端，而且在稍有一点经验的面试官面前是根本没有用的。建议大家还是一步一个脚印踏踏实实地走。

1.1 如何准备面试

1.1.1 如何获取大厂面试机会？

在讲如何获取大厂面试机会之前，先来给大家科普/对比一下两个校招非常常见的概念—春招和秋招。

1. 招聘人数：秋招多于春招；
2. 招聘时间：秋招一般7月左右开始，大概一直持续到10月底。**但是大厂（如BAT）都会早开始早**

结束，所以一定要把握好时间。春招最佳时间为3月，次佳时间为4月，进入5月基本就不会再有春招了（金三银四）。

3. **应聘难度**：秋招略大于春招；
4. **招聘公司**：秋招数量多，而春招数量较少，一般为秋招的补充。

综上，一般来说，秋招的含金量明显是高于春招的。

下面我就说一下我自己知道的一些方法，不过应该也涵盖了大部分获取面试机会的方法。

1. 关注大厂官网，随时投递简历（走流程的网申）；
2. 线下参加宣讲会，直接投递简历；
3. 找到师兄师姐/认识的人，帮忙内推（能够让你避开网申简历筛选，笔试筛选，还是挺不错的，不过也需要你的简历够棒）；
4. 博客发文被看中/Github优秀开源项目作者，大厂内部人员邀请你面试；
5. 求职类网站投递简历（不太推荐，适合海投）；

除了这些方法，我也遇到过这样的经历：有些大公司的一些部门可能暂时没招够人，然后如果你的亲戚或者朋友刚好在这个公司，而你正好又在寻求offer，那么面试机会基本上是有了，而且这种面试的难度好像一般还普遍比其他正规面试低很多。

1.1.2 准备自己的自我介绍

自我介绍一般是你和面试官的第一次面对面正式交流，换位思考一下，假如你是面试官的话，你想听到被你面试的人如何介绍自己呢？一定不是客套地说说自己喜欢编程、平时花了很多时间来学习、自己的兴趣爱好是打球吧？

我觉得一个好的自我介绍应该包含这几点要素：

1. 用简单的话说清楚自己主要的技术栈于擅长的领域；
2. 把重点放在自己在行的地方以及自己的优势之处；
3. 重点突出自己的能力比如自己的定位的bug的能力特别厉害；

从社招和校招两个角度来举例子吧！我下面的两个例子仅供参考，自我介绍并不需要死记硬背，记住要说的要点，面试的时候根据公司的情况临场发挥也是没问题的。另外，网上一般建议的是准备好两份自我介绍：一份对hr说的，主要讲能突出自己的经历，会的编程技术一语带过；另一份对技术面试官说的，主要讲自己会的技术细节和项目经验。

社招：

面试官，您好！我叫独秀儿。我目前有1年半的工作经验，熟练使用Spring、MyBatis等框架、了解Java 底层原理比如JVM调优并且有着丰富的分布式开发经验。离开上一家公司是因为我想在技术上得到更多的锻炼。在上一个公司我参与了一个分布式电子交易系统的开发，负责搭建了整个项目的基础架构并且通过分库分表解决了原始数据库以及一些相关表过于庞大的问题，目前这个网站最高支持 10 万人同时访问。工作之余，我利用自己的业余时间写了一个简单的 RPC 框架，这个框架用到了Netty进行网络通信，目前我已经将这个项目开源，在 Github 上收获了 2k 的 Star！说到业余爱好的话，我比较喜欢通过博客整理分享自己所学知识，现在已经是多个博客平台的认证作者。生活中我是一个比较积极乐观的人，一般会通过运动打球的方式来放松。我一直都非常想加入贵公司，我觉得贵公司的文化和技术氛围我都非常喜欢，期待能与你共事！

校招：

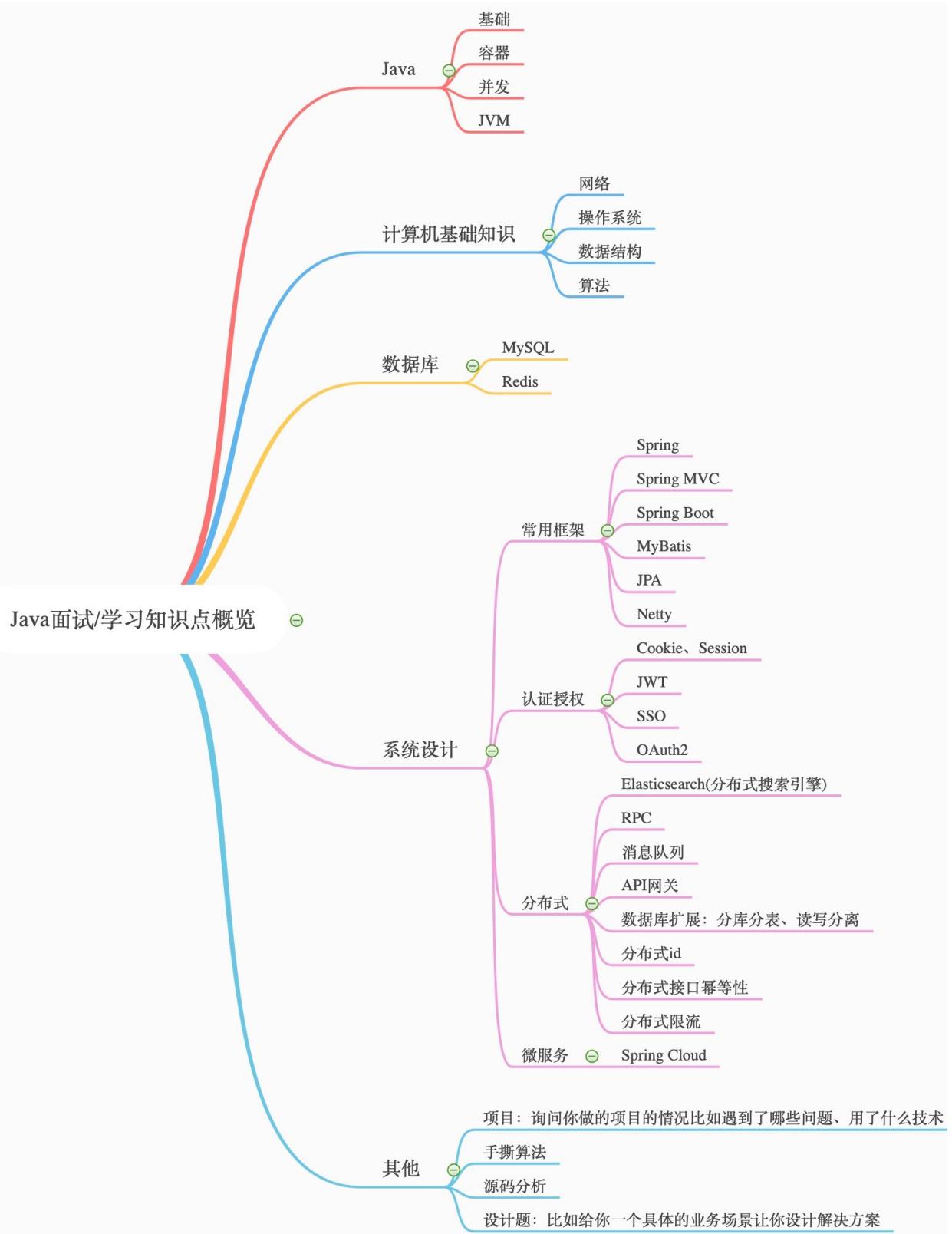
面试官，您好！我叫秀儿。大学时间我主要利用课外时间学习了 Java 以及 Spring、MyBatis等框架。在校期间参与过一个考试系统的开发，这个系统的主要用了 Spring、MyBatis 和 shiro 这三种框架。我在其中主要担任后端开发，主要负责了权限管理功能模块的搭建。另外，我在大学的时候参加过一次软件编程大赛，我和我的团队做的在线订餐系统成功获得了第二名的成绩。我还利用

自己的业余时间写了一个简单的 RPC 框架，这个框架用到了Netty进行网络通信，目前我已经将这个项目开源，在 Github 上收获了 2k的 Star！说到业余爱好的话，我比较喜欢通过博客整理分享自己所学知识，现在已经是多个博客平台的认证作者。生活中我是一个比较积极乐观的人，一般会通过运动打球的方式来放松。我一直都非常想加入贵公司，我觉得贵公司的文化和技术氛围我都非常喜欢，期待能与你共事！

1.1.3 搞清楚技术面可能会问哪些方向的问题

你准备面试的话首先要搞清技术面可能会被问哪些方向的问题吧！

我直接用思维导图的形式展示出来吧！这样更加直观形象一点，细化到某个知识点的话这张图没有介绍到，留个悬念，下篇文章会详细介绍。



上面思维导图大概涵盖了技术面试可能会设计的技术，但是你不需要把上面的每一个知识点都搞得很熟悉，要分清主次，对于自己不熟悉的技不要写在简历上，对于自己简单了解的技术不要说自己熟练掌握！

1.1.4 休闲着装即可

穿西装、打领带、小皮鞋？NO! NO! NO! 这是互联网公司面试又不是去走红毯，所以你只需要穿的简单大方就好，不需要太正式。

1.1.5 随身带上自己的成绩单和简历

校招的话，有的公司在面试前都会让你交一份成绩单和简历当做面试中的参考。

1.1.6 如果需要笔试就提前刷一些笔试题

平时空闲时间多的可以刷一下笔试题目（牛客网上有很多）。但是不要只刷面试题，不动手code，程序员不是为了考试而存在的。

1.1.7 花时间一些逻辑题

面试中发现有些公司都有逻辑题测试环节，并且都把逻辑笔试成绩作为很重要的一个参考。

1.1.8 准备好自己的项目介绍

如果有项目的话，技术面试第一步，面试官一般都是让你自己介绍一下你的项目。你可以从以下几个方向来考虑：

1. 对项目整体设计的一个感受（面试官可能会让你画系统的架构图）
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如：用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

1.1.9 提前准备技术面试

搞清楚自己面试中可能涉及哪些知识点、哪些知识点是重点。面试中哪些问题会被经常问到、自己该如何回答。（强烈不推荐背题，第一：通过背这种方式你能记住多少？能记住多久？第二：背题的方式的学习很难坚持下去！）

1.1.10 面试之前做好定向复习

所谓定向复习就是专门针对你要面试的公司来复习。比如你在面试之前可以在网上找找有没有你要面试的公司的面经。

举个栗子：在我面试 ThoughtWorks 的前几天我就在网上找了一些关于 ThoughtWorks 的技术面的一些文章。然后知道了 ThoughtWorks 的技术面会让我们在之前做的作业的基础上增加一个或两个功能，所以我提前一天就把我之前做的程序重新重构了一下。然后在技术面的时候，简单的改了几行代码之后写个测试就完事了。如果没有提前准备，我觉得 20 分钟我很大几率会完不成这项任务。

1.1.11 面试之后记得复盘

如果失败，不要灰心；如果通过，切勿狂喜。面试和工作实际上是两回事，可能很多面试未通过的人，工作能力比你强的多，反之亦然。我个人觉得面试也像是一场全新的征程，失败和胜利都是平常之事。所以，劝各位不要因为面试失败而灰心、丧失斗志。也不要因为面试通过而沾沾自喜，等待你的将是更美好的未来，继续加油！

1.2 程序员简历就该这样写

本篇文章除了教大家用Markdown如何写一份程序员专属的简历，后面还会给大家推荐一些不错的用来写Markdown简历的软件或者网站，以及如何优雅的将Markdown格式转变为PDF格式或者其他格式。

推荐大家使用Markdown语法写简历，然后再将Markdown格式转换为PDF格式后进行简历投递。

如果你对Markdown语法不太了解的话，可以花半个小时简单看一下Markdown语法说明：<http://www.markdown.cn>。

1.2.1 为什么说简历很重要？

一份好的简历可以在整个申请面试以及面试过程中起到非常好的作用。在不夸大自己能力的情况下，写出一份好的简历也是一项很棒的能力。为什么说简历很重要呢？

先从面试前来说：

- 假如你是网申，你的简历必然会经过HR的筛选，一张简历HR可能也就花费10秒钟看一下，然后HR就会决定你这一关是Fail还是Pass。
- 假如你是内推，如果你的简历没有什么优势的话，就算是内推你的人再用心，也无能为力。

另外，就算你通过了筛选，后面的面试中，面试官也会根据你的简历来判断你究竟是否值得他花费很多时间去面试。

所以，简历就像是我们的一个门面一样，它在很大程度上决定了你能否进入到下一轮的面试中。

再从面试中来说：

我发现大家比较喜欢看面经，这点无可厚非，但是大部分面经都没告诉你很多问题都是在特定条件下才问的。举个简单的例子：一般情况下你的简历上注明你会的东西才会被问到（Java、数据结构、网络、算法这些基础是每个人必问的），比如写了你会 redis，那面试官就很大概率会问你 redis 的一些问题。比如：redis的常见数据类型及应用场景、redis是单线程为什么还这么快、redis 和 memcached 的区别、redis 内存淘汰机制等等。

所以，首先，你要明确的一点是：你不会的东西就不要写在简历上。另外，你要考虑你该如何才能让你的亮点在简历中凸显出来，比如：你在某某项目做了什么事情解决了什么问题（只要有项目就一定有要解决的问题）、你的某一个项目里使用了什么技术后整体性能和并发量提升了很多等等。

面试和工作是两回事，聪明的人会把面试官往自己擅长的领域领，其他人则被面试官牵着鼻子走。虽说面试和工作是两回事，但是你要想要获得自己满意的 offer，你自身的实力必须要强。

1.2.2 关于简历你必须知道的点

1. 大部分公司的HR都说我们不看重学历（骗你的！），但是如果你的学校不出众的话，很难在一堆简历中脱颖而出，除非你的简历上有特别的亮点，比如：某某大厂的实习经历、获得了某某大赛的奖等等。
2. 大部分应届生找工作的硬伤是没有工作经验或实习经历，所以如果你是应届生就不要错过秋招和春招。一旦错过，你后面就极大可能会面临社招，这个时候没有工作经验的你可能就会面临各种碰壁，导致找不到一个好的工作。
3. 写在简历上的东西一定要慎重，这是面试官大量提问的地方；
4. 将自己的项目经历完美的展示出来非常重要。

1.2.3 写简历必须了解的两大法则

STAR法则（Situation Task Action Result）

- **Situation:** 事情是在什么情况下发生；
- **Task:** 你是如何明确你的任务的；
- **Action:** 针对这样的情况分析，你采用了什么行动方式；
- **Result:** 结果怎样，在这样的情况下你学习到了什么。

简而言之，STAR法则，就是一种讲述自己故事的方式，或者说，是一个清晰、条理的作文模板。不管是什
么，合理熟练运用此法则，可以轻松的对面试官描述事物的逻辑方式，表现出自己分析阐述问题的清
晰性、条理性和逻辑性。

FAB 法则（Feature Advantage Benefit）

- **Feature:** 是什么；
- **Advantage:** 比别人好在哪些地方；
- **Benefit:** 如果雇佣你，招聘方会得到什么好处。

简单来说，这个法则主要是让你的面试官知道你的优势、招了你之后对公司有什么帮助。

1.2.4 项目经历怎么写？

简历上有一两个项目经历很正常，但是真正能把项目经历很好的展示给面试官的非常少。对于项目经历大家
可以考虑从如下几点来写：

1. 对项目整体设计的一个感受
2. 在这个项目中你负责了什么、做了什么、担任了什么角色
3. 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
4. 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者
在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么
功能比如：用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

1.2.5 专业技能该怎么写？

先问一下你自己会什么，然后看看你意向的公司需要什么。一般HR可能并不太懂技术，所以他在筛选简历的时
候可能就盯着你专业技能的关键词来看。对于公司有要求而你不会的技能，你可以花几天时间学
习一下，然后在简历上可以写上自己了解这个技能。比如你可以这样写（下面这部分内容摘自我的简历，
大家可以根据自己的情况做一些修改和完善）：

- 计算机网络、数据结构、算法、操作系统等课内基础知识：掌握
- Java 基础知识：掌握
- JVM 虚拟机（Java内存区域、虚拟机垃圾算法、虚拟垃圾收集器、JVM内存管理）：掌握
- 高并发、高可用、高性能系统开发：掌握
- Struts2、Spring、Hibernate、Ajax、Mybatis、JQuery：掌握
- SSH 整合、SSM 整合、SOA 架构：掌握
- Dubbo：掌握
- Zookeeper：掌握
- 常见消息队列：掌握
- Linux：掌握
- MySQL常见优化手段：掌握
- Spring Boot +Spring Cloud +Docker：了解
- Hadoop 生态相关技术中的 HDFS、Storm、MapReduce、Hive、Hbase：了解
- Python 基础、一些常见第三方库比如OpenCV、wxpy、wordcloud、matplotlib：熟悉

1.2.6 排版注意事项

1. 尽量简洁，不要太花里胡哨；
2. 一些技术名词不要弄错了大小写比如MySQL不要写成mysql，Java不要写成java。这个在我看来还
是比较忌讳的，所以一定要注意这个细节；
3. 中文和数字英文之间加上空格的话看起来会舒服一点；

1.2.7 其他的一些小tips

1. 尽量避免主观表述，少一点语义模糊的形容词，尽量要简洁明了，逻辑结构清晰。
2. 如果自己有博客或者个人技术栈点的话，写上去会为你加分很多。
3. 如果自己的Github比较活跃的话，写上去也会为你加分很多。
4. 注意简历真实性，一定不要写自己不会的东西，或者带有欺骗性的内容
5. 项目经历建议以时间倒序排序，另外项目经历不在于多，而在于有亮点。
6. 如果内容过多的话，不需要非把内容压缩到一页，保持排版干净整洁就可以了。
7. 简历最后最好能加上：“感谢您花时间阅读我的简历，期待能有机会和您共事。”这句话，显的你会很有礼貌。

1.2.8 推荐的工具/网站

- 冷熊简历(MarkDown在线简历工具，可在线预览、编辑和生成PDF):<http://cv.ftqq.com/>
- Typora+Java程序员简历模板

1.3 大部分程序员在面试前很关心的一些问题

身边的朋友或者公众号的粉丝很多人都向我询问过：“我是双非/三本/专科学校的，我有机会进入大厂吗？”、“非计算机专业的学生能学好吗？”、“如何学习Java？”、“Java学习该学那些东西？”、“我该如何准备Java面试？”……这些问题。我会根据自己的一点经验对大部分人关心的这些问题进行答疑解惑。

希望这篇可以给已经在Java方向走了几年的朋友或者正在准备往Java后端方向发展的朋友们一点帮助。道理懂了如果没有实际行动，那这篇文章对你或许没有任何意义。

如果觉得内容不错的话，可以分享到朋友圈让你的朋友看到，感谢！

1.3.1 我是双非/三本/专科学校的，我有机会进入大厂吗？

我自己也是非985非211学校的，结合自己的经历以及一些朋友的经历，我觉得让我回答这个问题再好不过。

首先，我觉得学校歧视很正常，真的太正常了，如果要抱怨的话，你只能抱怨自己没有进入名校。但是，千万不要动不动说自己学校差，动不动拿自己学校当做自己进不了大厂的借口，学历只是筛选简历的很多标准中的一个而已，如果你够优秀，简历够丰富，你也一样可以和名校同学一起同台竞争。

企业HR肯定是更喜欢高学历的人，毕竟985, 211优秀人才比例肯定比普通学校高很多，HR团队肯定会优先在这些学校里选。这就好比相亲，你是愿意在很多优秀的人中选一个优秀的，还是愿意在很多普通的人中选一个优秀的呢？ 双非本科甚至是二本、三本甚至是专科的同学也有很多进入大厂的，不过比率相比于名校的低很多而已。从大厂招聘的结果上看，高学历人才的数量占据大头，那些成功进入BAT、美团，京东，网易等大厂的双非本科甚至是二本、三本甚至是专科的同学往往是因为具备丰富的项目经历或者在某个含金量比较高的竞赛比如ACM中取得了不错的成绩。一部分学历不突出但能力出众的面试者能够进入大厂并不是说明学历不重要，而是学历的软肋能够通过其他的优势来弥补。所以，如果你的学校不够好而你自己又想去大厂的话，建议你可以从这几点来做：①尽量在面试前最好有一个可以拿的出手的项目；②有实习条件的话，尽早出去实习，实习经历也会是你的简历的一个亮点（有能力在大厂实习最佳！）；③参加一些含金量比较高的比赛，拿不拿得到名次没关系，重在锻炼。

1.3.2 非计算机专业的学生能学好Java后台吗？我能进大厂吗？

当然可以！现在非科班的程序员很多，很大一部分原因是互联网行业的工资比较高。我们学校外面的培训班里面90%都是非科班，我觉得他们很多人学的都还不错。另外，我的一个朋友本科是机械专业，大一开始自学安卓，技术贼溜，在我看来他比大部分本科是计算机的同学学的还要好。参考Question1的回答，即使你是非科班程序员，如果你想进入大厂的话，你也可以通过自己的其他优势来弥补。

我觉得我们不应该因为自己的专业给自己划界限或者贴标签，说实话，很多科班的同学可能并不如你，你以为科班的同学就会认真听讲吗？还不是几乎全靠自己课下自学！不过如果你是非科班的话，你想要学好，那么注定就要舍弃自己本专业的一些学习时间，这是无可厚非的。

建议非科班的同学，首先要打好计算机基础知识基础：①计算机网络、②操作系统、③数据结构与算法，我个人觉得这3个对你最重要。这些东西就像是内功，对你以后的长远发展非常有用。当然，如果你想要进大厂的话，这些知识也是一定会被问到的。另外，“一定学好数据结构与算法！一定学好数据结构与算法！一定学好数据结构与算法！”，重要的东西说3遍。

1.3.3 如何学好Java后端呢？

对于学习路线的话，我说一条我比较推荐的，我相信照着这条学习路线来你的学习效率会非常高。下面提到的书籍以及相关学习视频都答主已经整理好，公众号JavaGuide后台回复关键“1”即可领取。

1. 掌握 Java 基础知识（可以看《Java 核心技术卷1》或者《Head First Java》这两本书在我看来都是入门Java的很不错的书籍），当然你也可以边看视频边看书学习（推荐黑马或者尚硅谷的视频）。记得多总结！打好基础！把自己重要的东西都记录下来。
2. 掌握多线程的简单实用（推荐《Java并发编程之美》或者《实战Java高并发程序设计》）。
3. （可选）如果你想进入大厂的话，我推荐你在学习完Java基础或者多线程之后，就开始每天抽出一点时间来学习算法和数据结构。为了提高自己的编程能力，你也可以坚持刷Leetcode。
4. 学习前端基础(HTML、CSS、JavaScript)，当然BootStrap、VUE等等前端框架你也可以了解一下。
5. 学习MySQL 的基本使用，基本的增删改查，SQL命令，索引、存储过程这些都学一下吧！
6. 建议学习J2ee框架之前可以提前花半天时间学习一下Maven的使用。（到处找Jar包，下载Jar包是真的麻烦费事，使用Maven可以为你省很多事情）
7. 学习Struts2(可不用学)、Spring、SpringMVC、Hibernate、Mybatis 等框架的使用，（可选）熟悉 Spring 原理（大厂面试必备），然后很有必要学习一下SpringBoot。我也遇到很多公司对于应届生直接上手SpringBoot，不过我还是推荐你把Spring、SpringMVC好好学一下。
8. 学习Linux的基本使用(常见命令、基本概念)
9. 学习Dubbo、Zookeeper、常见的消息队列（比如ActiveMq、RabbitMQ）的使用。（这些东西可以通过黑马最后一个分布式项目来学，边看视频，边自己做，查阅网上博客，效果更好）
10. 可以学习一下NIO和Netty，这样简历上也可以多点东西。
11. （可选），如果想去大厂，JVM 的一些知识也是必学的（Java内存区域、虚拟机垃圾算法、虚拟垃圾收集器、JVM内存管理）推荐《深入理解Java虚拟机：JVM高级特性与最佳实践（最新第二版）》，如果嫌看书麻烦的话，你也可以看我整理的文档，在下面有链接。

我上面主要概括一下每一步要学习的内容，对学习规划有一个建议。知道要学什么之后，如何去学呢？我觉得学习每个知识点可以考虑这样去入手：官网（大概率是英文，不推荐初学者看）、书籍（知识更加系统完全，推荐）、视频（比较容易理解，推荐，特别是初学的时候）、网上博客（解决某一知识点的问题的时候可以看看）。

这里给各位一个建议，看视频的过程中最好跟着一起练，要做笔记！！！最好可以边看视频边找一本书籍看，看视频没弄懂的知识点一定要尽快解决，如何解决？首先百度/Google，通过搜索引擎解决不了的话就找身边的朋友或者认识的一些人。

1.3.4 我没有实习经历的话找工作是不是特别艰难？

没有实习经历没关系，只要你有拿得出手的项目或者大赛经历的话，你依然有可能拿到大厂的 offer。笔者当时找工作的时候就没有实习经历以及大赛获奖经历，单纯就是凭借自己的项目经验撑起了整个面试。

如果你既没有实习经历，又没有拿得出手的项目或者大赛经历的话，我觉得在简历关除非你有其他特别的亮点，不然，你应该就会被刷。

1.3.5 我该如何准备面试呢？面试的注意事项有哪些呢？

下面是我总结的一些准备面试的Tips以及面试必备的注意事项：

1. 准备一份自己的自我介绍，面试的时候根据面试对象适当进行修改（突出重点，突出自己的优势在哪里，切忌流水账）；
2. 注意随身带上自己的成绩单和简历复印件；（有的公司在面试前都会让你交一份成绩单和简历当做面试中的参考。）
3. 如果需要笔试就提前刷一些笔试题，大部分在线笔试的类型是选择题+编程题，有的还会有简答题。（平时空闲时间多的可以刷一下笔试题目（牛客网上有很多），但是不要只刷面试题，不动手code，程序员不是为了考试而存在的。）另外，注意抓重点，因为题目太多了，但是有很多题目几乎次次遇到，像这样的题目一定要搞定。
4. 提前准备技术面试。搞清楚自己面试中可能涉及哪些知识点、那些知识点是重点。面试中哪些问题会被经常问到、自己该如何回答。（强烈不推荐背题，第一：通过背这种方式你能记住多少？能记住多久？第二：背题的方式的学习很难坚持下去！）
5. 面试之前做好定向复习。也就是专门针对你要面试的公司来复习。比如你在面试之前可以在网上找找有没有你要面试的公司的面经。
6. 准备好自己的项目介绍。如果有项目的话，技术面试第一步，面试官一般都是让你自己介绍一下你的项目。你可以从以下几个方向来考虑：①对项目整体设计的一个感受（面试官可能会让你画系统的架构图）；②在这个项目中你负责了什么、做了什么、担任了什么角色；③从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用；④项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如：用redis做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。
7. 提前知道有哪些技术问题常问：索引、隔离级别、HashMap源码分析、SpringMVC执行过程等等问题我觉得面试中实在太常见了，好好准备！后面的文章会我会分类详细介绍到那些问题最常问。
8. 提前熟悉一些常问的非技术问题：面试的时候有一些常见的非技术问题比如“面试官问你的优点是什么，应该如何回答？”、“面试官问你的缺点是什么，应该如何回答？”、“如果面试官问“你有什么问题问我吗？”时，你该如何回答”等等，对于这些问题，如何回答自己心里要有个数，别面试的时候出了乱子。
9. 面试之后记得复盘。面试遭遇失败是很正常的事情，所以善于总结自己的失败原因才是最重要的。如果失败，不要灰心；如果通过，切勿狂喜。

1.3.6 我该自学还是报培训班呢？

我本人更加赞同自学（你要知道去了公司可没人手把手教你了，而且几乎所有的公司都对培训班出生的有偏见。为什么有偏见，你学个东西还要去培训班，说明什么，同等水平下，你的自学能力以及自律能力一定是比不上自学的人的）。但是如果，你连每天在寝室坚持学上8个小时以上都坚持不了，或者总是容易半途而废的话，我还是推荐你去培训班。观望身边同学去培训班的，大多是非计算机专业或者是没有自律能力以及自学能力非常差的人。

另外，如果自律能力不行，你也可以通过结伴学习、参加老师的项目等方式来督促自己学习。

总结：去不去培训班主要还是看自己，如果自己能坚持自学就自学，坚持不下来就去培训班。如果要去培训班还要擦亮双眼，很多培训班现在都是为了圈钱，不道德！！！

1.3.7 没有项目经历/博客/Github开源项目怎么办？

从现在开始做！

没有项目经验怎么办？

如果实在没有实际项目让你去做，我觉得你可以通过下面几种方式：

1. 在网上找一个符合自己能力与找工作需求的实战项目视频或者博客跟着老师一起做。做的过程中，你要有自己的思考，不要浅尝辄止，对于很多知识点，别人的讲解可能只是满足项目就够了，你自己想多点知识的话，对于重要的知识点就要自己学会去往深出学。
2. Github或者码云上面有很多实战类别项目，你可以选择一个来研究，为了让自己对这个项目更加理解，在理解原有代码的基础上，你可以对原有项目进行改进或者增加功能。
3. 自己动手去做一个自己想完成的东西，遇到不会的东西就临时去学，现学现卖。

不光要做，还要改进，改善。另外，如果你的老师有相关 Java 后台项目的话，你也可以主动申请参与进来。

没有博客怎么办？

如果有自己的博客，也算是简历上的一个亮点。建议可以在掘金、Segmentfault、CSDN等技术交流社区写博客，当然，你也可以自己搭建一个博客（采用 Hexo+GitHub Pages 搭建非常简单）。写些什么？学习笔记、实战内容、读书笔记等等都可以。

没有开源项目怎么办？

多用 GitHub，用好 GitHub，上传自己不错的项目，写好 README 文档，在其他技术社区做好宣传。相信你也会收获一个不错的开源项目！

1.3.8 从招聘要求看大厂青睐什么样的人？

先从已经有两年左右开发经验的工程师角度来看： 我们来看一下阿里官网支付宝Java高级开发工程师的招聘要求，从下面的招聘信息可以看出，除去Java基础/集合/多线程这些，这些能力格外重要：

1. **底层知识比如jvm**：不只是懂理论更会实操；
2. **面向对象编程能力**：我理解这个不仅包括“面向对象编程”，还有SOLID软件设计原则，相关阅读：[《写了这么多年代码，你真的了解SOLID吗？》](#)（我司大佬的一篇文章）
3. **框架能力**：不只是使用那么简单，更要搞懂原理和机制！搞懂原理和机制的基础是要学会看源码。
4. **分布式系统开发能力**：缓存、消息队列等等都要掌握，关键是还要能使用这些技术解决实际问题而不是纸上谈兵。
5. **不错的sense**：喜欢和尝试新技术、追求编写优雅的代码等等。

发布时间:	2020-03-09	工作地点:	上海,杭州	工作年限:	三年以上
所属部门:	蚂蚁集团	学 历:	本科	招聘人数:	若干

岗位描述:

1. 深入发掘和分析业务需求，撰写技术方案和系统设计；
 2. 参与技术方案和系统设计评审；把握复杂系统的设计，确保系统的架构质量；
 3. 系统核心部分代码编写；疑难问题的解决；
 4. 对现存或未来系统进行宏观的思考，规划形成统一的框架、平台或组件；
 5. 指导和培训工程师，让团队成员在你的影响下取得成长；
 6. 为团队引入创新的技术、创新的解决方案，用创新的思路解决问题；
 7. 维护和升级现有软件产品和系统，快速定位并修复现有软件缺陷。
- 面向对象编程看似简单，实际做好的话很难

jvm调优加问题定位能力

岗位要求:

1. Java基础扎实，理解io、多线程、集合等基础框架，对JVM原理有一定的了解；熟悉面向对象设计开发；
2. 两年以上使用JAVA开发的经验，对于你用过的开源框架，能了解到它的原理和机制；
3. 熟悉分布式系统的设计和应用，熟悉分布式、缓存、消息、搜索\推荐等机制；能对分布式常用技术进行合理应用，解决问题；
4. 掌握Linux 操作系统和大型数据库；有较强的分析设计能力和方案整合能力；
5. 良好的沟通技能，团队合作能力，勤奋好学；
6. 我们希望你对互联网或J2EE应用开发的最新潮流有关注，喜欢去看及尝试最新的技术，追求编写优雅的代码，从技术趋势和思路上能影响技术团队；
7. 如果你觉得和以上要求不符，但你对这个岗位很感兴趣，并且确认你以往的其他经历或经验能给团队带来自己独特的价值，那么也欢迎投递简历；
8. 具有电子商务、金融行业背景的人优先。

再从应届生的角度来看： 我们还是看阿里巴巴的官网相关应届生 Java 工程师招聘岗位的相关要求。

研发工程师JAVA Software Engineer - Java

不需要你什么都懂，但是在某一

▶ 岗位描述 Job Description

领域一定要能独当一面

如果你想了解JAVA开发在阿里巴巴互联网生态系统中无与伦比的应用广度与深度；

如果你对基础技术感兴趣，你可以参与基础软件的设计、开发和维护，如分布式文件系统、缓存系统、Key/Value存储系统、数据库、Linux操作系统和Java优化等；

如果你热衷于高性能分布式技术，你可以参与高性能分布式服务端程序的系统设计，为阿里巴巴的产品提供强有力的后台支持，在海量的网络访问和数据处理中，设计并实施最强大的解决方案；

如果你喜欢研究搜索技术，你可以参与搜索引擎各个功能模块的设计和实现，构建高可靠性、高可用性、高可扩展性的体系结构，满足日趋复杂的业务需求；

如果你对电子商务产品技术感兴趣，你可以参与产品的开发和维护，完成从需求到设计、开发和上线等整个项目周期内的工作；

如果你对数据敏感，你可以参与海量数据处理和开发，通过sql、pl/sql、java进行etl程序开发，满足商业上对数据的开发需求；

如果你热衷于客户端开发，你可以参与为用户提供丰富而有价值的桌面或无线软件产品。

竞赛获奖（尤其是ACM）或者参与实际项目都会为简历加分很多

或许，你来自计算机专业，机械专业，甚至可能是学生物的；

但是，你酷爱着计算机以及互联网技术，热衷于解决挑战性的问题，追求极致的用户体验；

或许，你痴迷于数据结构和算法，热衷于ACM，常常为看到“accept”而兴奋的手足舞蹈；

或许，你熟悉Unix/Linux/Win32环境下编程，并有相关开发经验，熟练使用调试工具，并熟悉Perl，Python，shell等脚本语言；

或许，你熟悉网络编程和多线程编程，对TCP/IP，HTTP等网络协议有很深的理解，并了解XML和HTML语言；

或许，你热衷于数据库技术，能够熟练编写SQL脚本，有MySQL或Oracle应用开发经验；

或许，你并不熟悉Java编程语言，更精通C，C++，PHP，.NET等编程语言中的一种或几种，但你有良好和快速的学习能力；

有可能，你参加过大学生数学建模竞赛，“挑战杯”，机器人足球比赛等；

也有可能，你在学校的时候作为骨干参与学生网站的建设和开发；

这些，都是我们想要的。来吧，加入我们！

▶ 工作地点 Location

无锡市(Wuxi),上海市(Shanghai),成都市(Chengdu),深圳市(Shenzhen),北京市(Beijing),广州市(Guangzhou),杭州市(Hangzhou),南京市(Nanjing)

▶ 参加面试的城市或地区 Interview City or Region

远程(Remote Interviews)

视频面很常见，提前做好准备

申请岗位
Apply

结合阿里、腾讯等大厂招聘官网对于 Java 后端方向/后端方向的应届实习生的要求下面几点也提升你的个人竞争力：

1. 参加过竞赛（含金量超高的是 ACM）；
2. 对数据结构与算法非常熟练；
3. 参与过实际项目（比如学校网站）
4. 熟悉 Python、Shell、Perl 其中一门脚本语言；
5. 熟悉如何优化 Java 代码、有写出质量更高的代码的意识；
6. 熟悉 SOA 分布式相关的知识尤其是理论知识；
7. 熟悉自己所用框架的底层知识比如 Spring；
8. 有高并发开发经验；
9. 有大数据开发经验等等。

从来到大学之后，我的好多阅历非常深的老师经常就会告诫我们：“一定要有一门自己的特长，不管是技术还好还是其他能力”。我觉得这句话真的非常有道理！

刚刚也提到了要有一门特长，所以在这里再强调一点：公司不需要你什么都会，但是在某一方面你一定要有过于常人的优点。换言之就是我们不需要去掌握每一门技术（你也没精力去掌握这么多技术），而是需要去深入研究某一门技术，对于其他技术我们可以简单了解一下。

1.4 如何学习？学会各种框架有必要吗？

1.4.1 我该如何学习？



最最关键也是对自己最最重要的就是学习！看看别人分享的面经，看看我写的这篇文章估计你只需要10分钟不到。但这些东西终究是空洞的理论，最主要的还是自己平时的学习！

如何去学呢？我觉得学习每个知识点可以考虑这样去入手：

1. 官网（大概率是英文，不推荐初学者看）。
2. 书籍（知识更加系统完全，推荐）。
3. 视频（比较容易理解，推荐，特别是初学的时候。慕课网和哔哩哔哩上面有挺多学习视频可以看，只直接在上面搜索关键词就可以了）。
4. 网上博客（解决某一知识点的问题的时候可以看看）。

这里给各位一个建议，看视频的过程中最好跟着一起练，要做笔记！！！

最好可以边看视频边找一本书籍看，看视频没弄懂的知识点一定要尽快解决，如何解决？

首先百度/Google，通过搜索引擎解决不了的话就找身边的朋友或者认识的一些人。

1.4.2 学会各种框架有必要吗？

一定要学会分配自己时间，要学的东西很多，真的很多，搞清楚哪些东西是重点，哪些东西仅仅了解就够了。一定不要把精力都花在了学各种框架上，算法、数据结构还有计算机网络真的很重要！

另外，学习的过程中有一个可以参考的文档很重要，非常有助于自己的学习。我当初弄 JavaGuide：[h](https://github.com/Snailclimb/JavaGuide) <https://github.com/Snailclimb/JavaGuide> 的很大一部分目的就是因为这个。客观来说，相比于博客，JavaGuide 里面的内容因为更多人的参与变得更加准确和完善。

如果大家觉得这篇文章不错的话，欢迎给我来个三连（评论+转发+在看）！我会在下一篇文章中介绍如何从技术面试的角度准备面试？

二 Java基础+集合+多线程+JVM

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

2.1 Java基础

1. 面向对象和面向过程的区别

- **面向过程**：面向过程性能比面向对象高。因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix 等一般采用面向过程开发。但是，面向过程没有面向对象易维护、易复用、易扩展。
- **面向对象**：面向对象易维护、易复用、易扩展。因为面向对象有封装、继承、多态性的特性，所以可以设计出低耦合的系统，使系统更加灵活、更加易于维护。但是，面向对象性能比面向过程低。

参见 issue：[面向过程：面向过程性能比面向对象高？？](#)

这个并不是根本原因，面向过程也需要分配内存，计算内存偏移量，Java 性能差的主要原因并不是因为它是面向对象语言，而是 Java 是半编译语言，最终的执行代码并不是可以直接被 CPU 执行的二进制机械码。

而面向过程语言大多都是直接编译成机械码在电脑上执行，并且其它一些面向过程的脚本语言性能也并不一定比 Java 好。

2. Java 语言有哪些特点？

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；

7. 支持网络编程并且很方便（Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存；

修正（参见：[issue#544](#)）：C++11 开始（2011 年的时候），C++ 就引入了多线程库，在 windows、linux、macos 都可以使用 `std::thread` 和 `std::async` 来创建线程。参考链接：<http://www.cplusplus.com/reference/thread/thread/?kw=thread>

3. 关于 JVM JDK 和 JRE 最详细通俗的解答

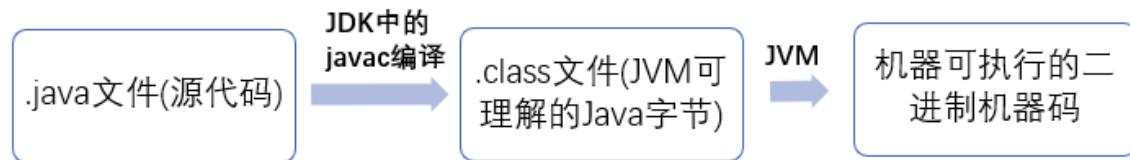
JVM

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows, Linux, macOS），目的是使用相同的字节码，它们都会给出相同的结果。

什么是字节码？采用字节码的好处是什么？

在 Java 中，JVM 可以理解的代码就叫做字节码（即扩展名为 `.class` 的文件），它不面向任何特定的处理器，只面向虚拟机。Java 语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型语言可移植的特点。所以 Java 程序运行时比较高效，而且，由于字节码并不针对一种特定的机器，因此，Java 程序无须重新编译便可在多种不同操作系统的计算机上运行。

Java 程序从源代码到运行一般有下面 3 步：



我们需要格外注意的是 `.class` → 机器码 这一步。在这一步 JVM 类加载器首先加载字节码文件，然后通过解释器逐行解释执行，这种方式的执行速度会相对比较慢。而且，有些方法和代码块是经常需要被调用的（也就是所谓的热点代码），所以后面引进了 JIT 编译器，而 JIT 属于运行时编译。当 JIT 编译器完成第一次编译后，其会将字节码对应的机器码保存下来，下次可以直接使用。而我们知道，机器码的运行效率肯定是高于 Java 解释器的。这也解释了我们为什么经常会说 Java 是编译与解释共存的语言。

HotSpot 采用了惰性评估（Lazy Evaluation）的做法，根据二八定律，消耗大部分系统资源的只有那一小部分的代码（热点代码），而这也正是 JIT 所需要编译的部分。JVM 会根据代码每次被执行的情况收集信息并相应地做出一些优化，因此执行的次数越多，它的速度就越快。JDK 9 引入了一种新的编译模式 AOT（Ahead of Time Compilation），它是直接将字节码编译成机器码，这样就避免了 JIT 预热等各方面的开销。JDK 支持分层编译和 AOT 协作使用。但是，AOT 编译器的编译质量是肯定比不上 JIT 编译器的。

总结：

Java 虚拟机（JVM）是运行 Java 字节码的虚拟机。JVM 有针对不同系统的特定实现（Windows, Linux, macOS），目的是使用相同的字节码，它们都会给出相同的结果。字节码和不同系统的 JVM 实现是 Java 语言“一次编译，随处可以运行”的关键所在。

JDK 和 JRE

JDK 是 Java Development Kit，它是功能齐全的 Java SDK。它拥有 JRE 所拥有的一切，还有编译器 (javac) 和工具（如 javadoc 和 jdb）。它能够创建和编译程序。

JRE 是 Java 运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java 虚拟机 (JVM)，Java 类库，java 命令和其他的一些基础构件。但是，它不能用于创建新程序。

如果你只是为了运行一下 Java 程序的话，那么你只需要安装 JRE 就可以了。如果你需要进行一些 Java 编程方面的工作，那么你就需要安装 JDK 了。但是，这不是绝对的。有时，即使您不打算在计算机上进行任何 Java 开发，仍然需要安装 JDK。例如，如果要使用 JSP 部署 Web 应用程序，那么从技术上讲，您只是在应用程序服务器中运行 Java 程序。那你为什么需要 JDK 呢？因为应用程序服务器会将 JSP 转换为 Java servlet，并且需要使用 JDK 来编译 servlet。

4. Oracle JDK 和 OpenJDK 的对比

可能在看这个问题之前很多人和我一样并没有接触和使用过 OpenJDK。那么 Oracle 和 OpenJDK 之间是否存在重大差异？下面我通过收集到的一些资料，为你解答这个被很多人忽视的问题。

对于 Java 7，没什么关键的地方。OpenJDK 项目主要基于 Sun 捐赠的 HotSpot 源代码。此外，OpenJDK 被选为 Java 7 的参考实现，由 Oracle 工程师维护。关于 JVM，JDK，JRE 和 OpenJDK 之间的区别，Oracle 博客帖子在 2012 年有一个更详细的答案：

问：OpenJDK 存储库中的源代码与用于构建 Oracle JDK 的代码之间有什么区别？

答：非常接近 - 我们的 Oracle JDK 版本构建过程基于 OpenJDK 7 构建，只添加了几个部分，例如部署代码，其中包括 Oracle 的 Java 插件和 Java WebStart 的实现，以及一些封闭的源代码派对组件，如图形光栅化器，一些开源的第三方组件，如 Rhino，以及一些零碎的东西，如附加文档或第三方字体。展望未来，我们的目的是开源 Oracle JDK 的所有部分，除了我们考虑商业功能的部分。

总结：

1. Oracle JDK 大概每 6 个月发一次主要版本，而 OpenJDK 版本大概每三个月发布一次。但这不是固定的，我觉得了解这个没啥用处。详情参见：<https://blogs.oracle.com/java-platform-group/update-and-faq-on-the-java-se-release-cadence>。
2. OpenJDK 是一个参考模型并且是完全开源的，而 Oracle JDK 是 OpenJDK 的一个实现，并不是完全开源的；
3. Oracle JDK 比 OpenJDK 更稳定。OpenJDK 和 Oracle JDK 的代码几乎相同，但 Oracle JDK 有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择 Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用 OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到 Oracle JDK 就可以解决问题；
4. 在响应性和 JVM 性能方面，Oracle JDK 与 OpenJDK 相比提供了更好的性能；
5. Oracle JDK 不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；
6. Oracle JDK 根据二进制代码许可协议获得许可，而 OpenJDK 根据 GPL v2 许可获得许可。

5. Java 和 C++的区别？

我知道很多人没学过 C++，但是面试官就是没事喜欢拿咱们 Java 和 C++ 比呀！没办法！！！就算没学过 C++，也要记下来！

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。

- Java 有自动内存管理机制，不需要程序员手动释放无用内存
- 在 C 语言中，字符串或字符数组最后都会有一个额外的字符‘\0’来表示结束。但是，Java 语言中没有结束符这一概念。这是一个值得深度思考的问题，具体原因推荐看这篇文章：
<https://blog.csdn.net/sszgg2006/article/details/49148189>

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

6. 什么是 Java 程序的主类 应用程序和小程序的主类有何不同？

一个程序中可以有多个类，但只能有一个类是主类。在 Java 应用程序中，这个主类是指包含 `main()` 方法的类。而在 Java 小程序中，这个主类是一个继承自系统类 `JApplet` 或 `Applet` 的子类。应用程序的主类不一定要求是 `public` 类，但小程序的主类要求必须是 `public` 类。主类是 Java 程序执行的入口点。

7. Java 应用程序与小程序之间有哪些差别？

简单说应用程序是从主线程启动(也就是 `main()` 方法)。applet 小程序没有 `main()` 方法，主要是嵌在浏览器页面上运行(调用 `init()` 或者 `run()` 来启动)，嵌入浏览器这点跟 flash 的小游戏类似。

8. 字符型常量和字符串常量的区别？

1. 形式上：字符常量是单引号引起的一个字符；字符串常量是双引号引起的若干个字符
2. 含义上：字符常量相当于一个整型值(ASCII 值)，可以参加表达式运算；字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小：字符常量只占 2 个字节；字符串常量占若干个字节（注意：`char` 在 Java 中占两个字节）

java 编程思想第四版：2.2.2 节

Java要确定每种基本类型所占存储空间的大小。它们的大小并不像其他大多数语言那样随机器硬件架构的变化而变化。这种所占存储空间大小的不变性是Java程序比用其他大多数语言编写的程序更具可移植性的原因之一。

基本类型	大小	最小值	最大值	包装器类型
boolean	—	—	—	Boolean
char	16-bit	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	—	—	—	Void

9. 构造器 Constructor 是否可被 override？

Constructor 不能被 `override` (重写)，但是可以 `overload` (重载)，所以你可以看到一个类中有多个构造函数的情况。

10. 重载和重写的区别

重载就是同样的一个方法能够根据输入数据的不同，做出不同的处理

重写就是当子类继承自父类的相同方法，输入数据一样，但要做出有别于父类的响应时，你就要覆盖父类方法

重载

发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同。

下面是《Java 核心技术》对重载这个概念的介绍：

4.6.1 重载

有些类有多个构造器。例如，可以如下构造一个空的 `StringBuilder` 对象：

```
StringBuilder messages = new StringBuilder();
```

或者，可以指定一个初始字符串：

```
StringBuilder todoList = new StringBuilder("To do:\n");
```

这种特征叫做重载 (overloading)。如果多个方法（比如，`StringBuilder` 构造器方法）有相同的名字、不同的参数，便产生了重载。编译器必须挑选出具体执行哪个方法，它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数，就会产生编译时错误，因为根本不存在匹配，或者没有一个比其他的更好。（这个过程被称为重载解析 (overloading resolution)。）

 **注释：**Java 允许重载任何方法，而不只是构造器方法。因此，要完整地描述一个方法，需要指出方法名以及参数类型。这叫做方法的签名 (signature)。例如，`String` 类有 4 个称为 `indexOf` 的公有方法。它们的签名是

```
indexOf(int)  
indexOf(int, int)  
indexOf(String)  
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说，不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

综上：重载就是同一个类中多个同名方法根据不同的传参来执行不同的逻辑处理。

重写

重写发生在运行期，是子类对父类的允许访问的方法的实现过程进行重新编写。

1. 返回值类型、方法名、参数列表必须相同，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类。
2. 如果父类方法访问修饰符为 `private/final/static` 则子类就不能重写该方法，但是被 `static` 修饰的方法能够被再次声明。
3. 构造方法无法被重写

综上：重写就是子类对父类方法的重新改造，外部样子不能改变，内部逻辑可以改变。

暖心的 Guide 哥最后再来个图标总结一下！

区别点	重载方法	重写方法

发生范围	同一个类	子类 中
参数列表	必须修改	一定不能修改
返回类型	可修改	一定不能修改
异常	可修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问修饰符	可修改	一定不能做更严格的限制（可以降低限制）
发生阶段	编译期	运行期

11. Java 面向对象编程三大特性: 封装 继承 多态

封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

关于继承如下 3 点请记住：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，**只是拥有**。
2. 子类可以拥有自己属性和方法，即子类可以对父类进行扩展。
3. 子类可以用自己的方式实现父类的方法。（以后介绍）。

多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在 Java 中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

12. String StringBuffer 和 StringBuilder 的区别是什么? String 为什么是不可变的?

可变性

简单的来说：String 类中使用 final 关键字修饰字符数组来保存字符串，`private final char value[]`，所以 String 对象是不可变的。

补充（来自[issue 675](#)）：在 Java 9 之后，String 类的实现改用 byte 数组存储字符串
`private final byte[] value`

而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中也是使用字符数组保存字符串 `char[] value` 但是没有用 final 关键字修饰，所以这两种对象都是可变的。

`StringBuilder` 与 `StringBuffer` 的构造方法都是调用父类构造方法也就是 `AbstractStringBuilder` 实现的，大家可以自行查阅源码。

AbstractStringBuilder.java

```
abstract class AbstractStringBuilder implements Appendable, CharSequence
{
    /**
     * The value is used for character storage.
     */
    char[] value;

    /**
     * The count is the number of characters used.
     */
    int count;

    AbstractStringBuilder(int capacity) {
        value = new char[capacity];
    }
}
```

线程安全性

`String` 中的对象是不可变的，也就可以理解为常量，线程安全。`AbstractStringBuilder` 是 `StringBuilder` 与 `StringBuffer` 的公共父类，定义了一些字符串的基本操作，如 `expandCapacity`、`append`、`insert`、`indexOf` 等公共方法。`StringBuffer` 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。`StringBuilder` 并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对 `String` 类型进行改变的时候，都会生成一个新的 `String` 对象，然后将指针指向新的 `String` 对象。`StringBuffer` 每次都会对 `StringBuffer` 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 `StringBuilder` 相比使用 `StringBuffer` 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结：

1. 操作少量的数据：适用 `String`
2. 单线程操作字符串缓冲区下操作大量数据：适用 `StringBuilder`
3. 多线程操作字符串缓冲区下操作大量数据：适用 `StringBuffer`

13. 自动装箱与拆箱

- 装箱：将基本类型用它们对应的引用类型包装起来；
- 拆箱：将包装类型转换为基本数据类型；

更多内容见：[深入剖析Java中的装箱和拆箱](#)

14. 在一个静态方法内调用一个非静态成员为什么是非法的？

由于静态方法可以不通过对象进行调用，因此在静态方法里，不能调用其他非静态变量，也不可以访问

非静态变量成员。

15. 在 Java 中定义一个不做事且没有参数的构造方法的作用

Java 程序在执行子类的构造方法之前，如果没有用 `super()` 来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用 `super()` 来调用父类中特定的构造方法，则编译时将发生错误，因为 Java 程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

16. `import java` 和 `javax` 有什么区别？

刚开始的时候 JavaAPI 所必需的包是 `java` 开头的包，`javax` 当时只是扩展 API 包来使用。然而随着时间的推移，`javax` 逐渐地扩展成为 Java API 的组成部分。但是，将扩展从 `javax` 包移动到 `java` 包确实太麻烦了，最终会破坏一堆现有的代码。因此，最终决定 `javax` 包将成为标准 API 的一部分。

所以，实际上 `java` 和 `javax` 没有区别。这都是一个名字。

17. 接口和抽象类的区别是什么？

1. 接口的方法默认是 `public`，所有方法在接口中不能有实现（Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。
2. 接口中除了 `static`、`final` 变量，不能有其他变量，而抽象类中则不一定。
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过 `extends` 关键字扩展多个接口。
4. 接口方法默认修饰符是 `public`，抽象方法可以有 `public`、`protected` 和 `default` 这些修饰符（抽象方法就是为了被重写所以不能使用 `private` 关键字修饰！）。
5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：

1. 在 JDK8 中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。
(详见 issue:<https://github.com/Snailclimb/JavaGuide/issues/146>)
2. jdk9 的接口被允许定义私有方法。

总结一下 jdk7~jdk9 Java 中接口概念的变化（[相关阅读](#)）：

1. 在 jdk 7 或更早版本中，接口里面只能有常量变量和抽象方法。这些接口方法必须由选择实现接口的类实现。
2. jdk8 的时候接口可以有默认方法和静态方法功能。
3. Jdk 9 在接口中引入了私有方法和私有静态方法。

18. 成员变量与局部变量的区别有哪些？

1. 从语法形式上看：成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 `public`、`private`、`static` 等修饰符所修饰，而局部变量不能被访问控制修饰符及 `static` 所修饰；但是，成员变量和局部变量都能被 `final` 所修饰。
2. 从变量在内存中的存储方式来看：如果成员变量是使用 `static` 修饰的，那么这个成员变量是属于类的，如果没有使用 `static` 修饰，这个成员变量是属于实例的。对象存于堆内存，如果局部变量类型为基本数据类型，那么存储在栈内存，如果为引用数据类型，那存放的是指向堆内存对象的引用或者是指向常量池中的地址。
3. 从变量在内存中的生存时间上看：成员变量是对象的一部分，它随着对象的创建而存在，而局部

变量随着方法的调用而自动消失。

- 成员变量如果没有被赋初值:则会自动以类型的默认值而赋值 (一种情况例外:被 final 修饰的成员变量也必须显式地赋值) , 而局部变量则不会自动赋值。

19. 创建一个对象用什么运算符?对象实体与对象引用有何不同?

new 运算符, new 创建对象实例 (对象实例在堆内存中) , 对象引用指向对象实例 (对象引用存放在栈内存中)。一个对象引用可以指向 0 个或 1 个对象 (一根绳子可以不系气球, 也可以系一个气球) ; 一个对象可以有 n 个引用指向它 (可以用 n 条绳子系住一个气球) 。

20. 什么是方法的返回值?返回值在类的方法里的作用是什么?

方法的返回值是指我们获取到的某个方法体中的代码执行后产生的结果! (前提是该方法可能产生结果) 。返回值的作用:接收出结果, 使得它可以用于其他的操作!

21. 一个类的构造方法的作用是什么?若一个类没有声明构造方法, 该程序能正确执行吗?为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

22. 构造方法有哪些特性?

- 名字与类名相同。
- 没有返回值, 但不能用 void 声明构造函数。
- 生成类的对象时自动执行, 无需调用。

23. 静态方法和实例方法有何不同

- 在外部调用静态方法时, 可以使用"类名.方法名"的方式, 也可以使用"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说, 调用静态方法可以无需创建对象。
- 静态方法在访问本类的成员时, 只允许访问静态成员 (即静态成员变量和静态方法) , 而不允许访问实例成员变量和实例方法; 实例方法则无此限制。

24. 对象的相等与指向他们的引用相等,两者有什么不同?

对象的相等, 比的是内存中存放的内容是否相等。而引用相等, 比较的是他们指向的内存地址是否相等。

25. 在调用子类构造方法之前会先调用父类没有参数的构造方法,其目的是?

帮助子类做初始化工作。

26. == 与 equals(重要)

== : 它的作用是判断两个对象的地址是不是相等。即, 判断两个对象是不是同一个对象(基本数据类型==比较的是值, 引用数据类型==比较的是内存地址)。

equals() : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况:

- 情况 1: 类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时, 等价于通过 "==" 比较这两个对象。

- 情况 2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true（即，认为这两个对象相等）。

举个例子：

```
public class test1 {
    public static void main(String[] args) {
        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b 为另一个引用, 对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找
        if (aa == bb) // true
            System.out.println("aa==bb");
        if (a == b) // false, 非同一对象
            System.out.println("a==b");
        if (a.equals(b)) // true
            System.out.println("aEQb");
        if (42 == 42.0) { // true
            System.out.println("true");
        }
    }
}
```

说明：

- String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的 equals 方法比较的是对象的值。
- 当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

27. hashCode 与 equals (重要)

面试官可能会问你：“你重写过 hashCode 和 equals 么，为什么重写 equals 时必须重写 hashCode 方法？”

hashCode () 介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个 int 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在 JDK 的 Object.java 中，这就意味着 Java 中的任何类都包含有 hashCode() 函数。

散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）

为什么要有 hashCode

我们先以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode：当你把对象加入 HashSet 时，HashSet 会先计算对象的 hashCode 值来判断对象加入的位置，同时也会与该位置其他已经加入的对象的 hashCode 值作比较，如果没有相符的 hashCode，HashSet 会假设对象没有重复出现。但是如果发现有相同 hashCode 值的对象，这时会调用 equals() 方法来检查 hashCode 相等的对象是否真

的相同。如果两者相同，`HashSet` 就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。（摘自我的 Java 启蒙书《Head first java》第二版）。这样我们就大大减少了 `equals` 的次数，相应就大大提高了执行速度。

通过我们可以看出：`hashCode()` 的作用就是获取哈希码，也称为散列码；它实际上是返回一个 `int` 整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。`hashCode()` 在散列表中才有用，在其它情况下没用。在散列表中 `hashCode()` 的作用是获取对象的散列码，进而确定该对象在散列表中的位置。

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. 两个对象相等，对两个对象分别调用 `equals` 方法都返回 `true`
3. 两个对象有相同的 `hashCode` 值，它们也不一定是相等的
4. 因此，`equals` 方法被覆盖过，则 `hashCode` 方法也必须被覆盖
5. `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

推荐阅读：[Java hashCode\(\) 和 equals\(\) 的若干问题解答](#)

28. 为什么 Java 中只有值传递？

[为什么 Java 中只有值传递？](#)

29. 简述线程、程序、进程的基本概念。以及他们之间关系是什么？

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享同一块内存空间和一组系统资源，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

程序是含有指令和数据的文件，被存储在磁盘或其他的数据存储设备中，也就是说程序是静态的代码。

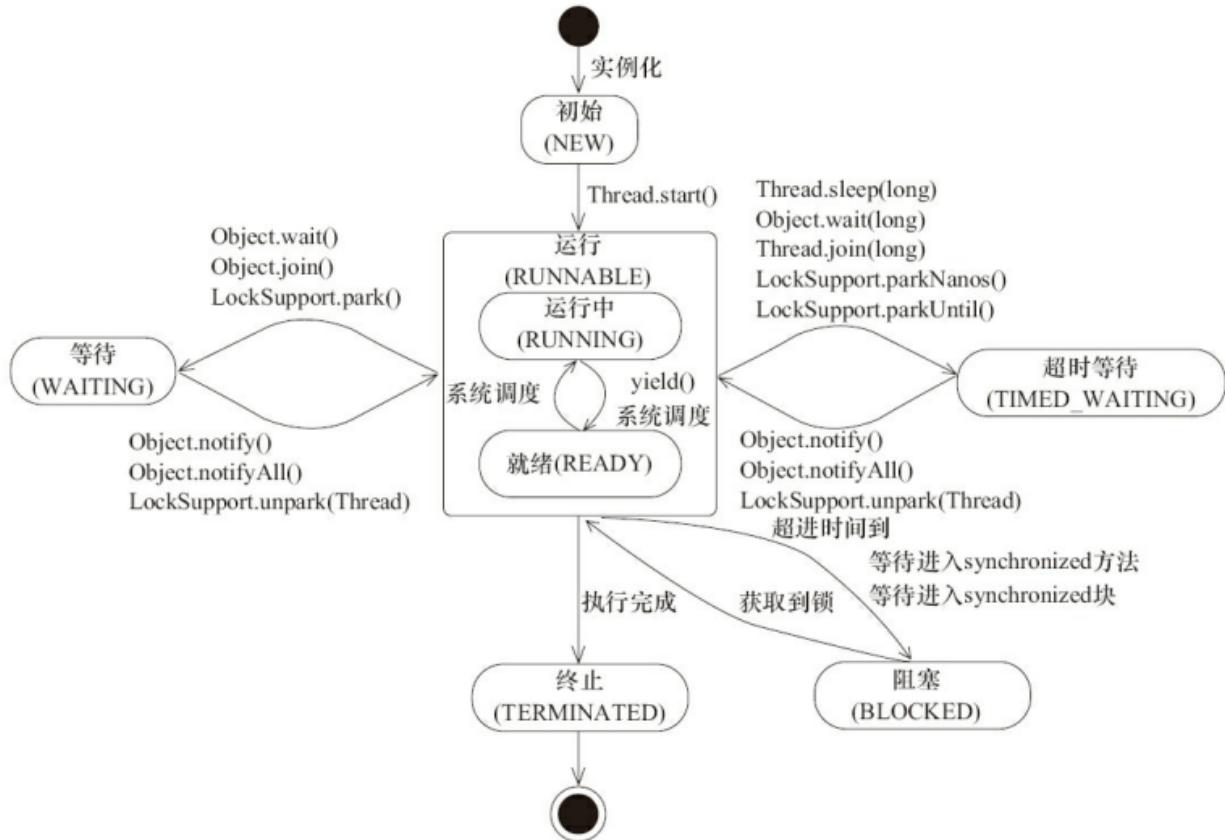
进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。简单来说，一个进程就是一个执行中的程序，它在计算机中一个指令接着一个指令地执行着，同时，每个进程还占有某些系统资源如 CPU 时间，内存空间，文件，输入输出设备的使用权等等。换句话说，当程序在执行时，将会被操作系统载入内存中。线程是进程划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。从另一角度来说，进程属于操作系统的范畴，主要是同一段时间内，可以同时执行一个以上的程序，而线程则是在同一程序内几乎同时执行一个以上的程序段。

30. 线程有哪些基本状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。

状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 <code>start()</code> 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

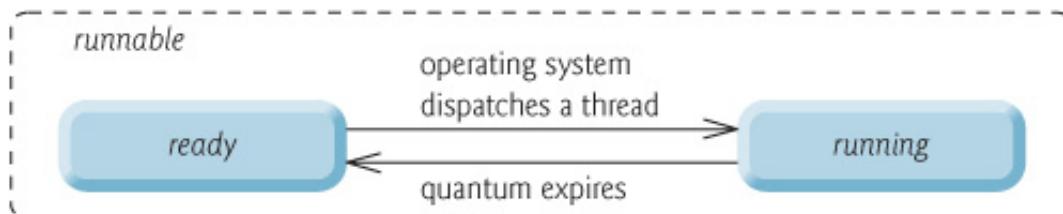
线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



由上图可以看出：

线程创建之后它将处于 NEW (新建) 状态，调用 `start()` 方法后开始运行，线程这时候处于 READY (可运行) 状态。可运行状态的线程获得了 cpu 时间片 (timeslice) 后就处于 RUNNING (运行) 状态。

操作系统隐藏 Java 虚拟机 (JVM) 中的 READY 和 RUNNING 状态，它只能看到 RUNNABLE 状态（图源：[HowToDoInJava: Java Thread Life Cycle and Thread States](#)），所以 Java 系统一般将这两个状态统称为 RUNNABLE (运行中) 状态。



当线程执行 `wait()` 方法之后，线程进入 WAITING (等待) 状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 TIME_WAITING(超时等待) 状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 TIMED_WAITING 状态。当超时时间到达后 Java 线程将会返回到 RUNNABLE 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 BLOCKED (阻塞) 状态。线程在执行 Runnable 的 `run()` 方法之后将会进入到 TERMINATED (终止) 状态。

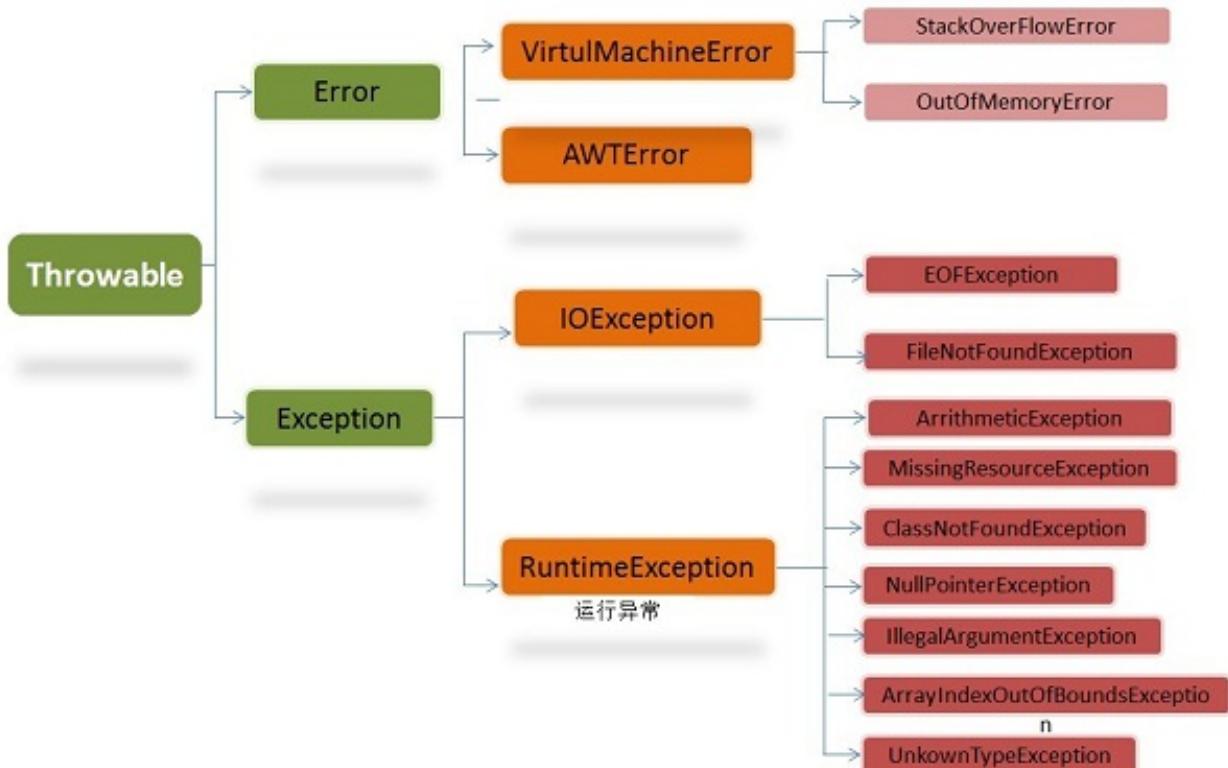
31 关于 final 关键字的一些总结

`final` 关键字主要用在三个地方：变量、方法、类。

- 对于一个 final 变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
- 当用 final 修饰一个类时，表明这个类不能被继承。final 类中的所有成员方法都会被隐式地指定为 final 方法。
- 使用 final 方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的 Java 实现版本中，会将 final 方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的 Java 版本已经不需要使用 final 方法进行这些优化了）。类中所有的 private 方法都隐式地指定为 final。

32 Java 中的异常处理

Java 异常类层次结构图



在 Java 中，所有的异常都有一个共同的祖先 `java.lang` 包中的 `Throwable` 类。`Throwable` 有两个重要的子类：`Exception`（异常）和 `Error`（错误），二者都是 Java 异常处理的重要子类，各自都包含大量子类。

Error（错误）：是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java 虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 `OutOfMemoryError`。这些异常发生时，Java 虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如 Java 虚拟机运行错误（Virtual MachineError）、类定义错误（`NoClassDefFoundError`）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java 中，错误通过 `Error` 的子类描述。

Exception（异常）：是程序本身可以处理的异常。`Exception` 类有一个重要的子类 `RuntimeException`。`RuntimeException` 异常由 Java 虚拟机抛出。`NullPointerException`（要访问的变量没有引用任何对象时，抛出该异常）、`ArithmaticException`（算术运算异常，一个整数除以 0 时，抛出该异常）和 `ArrayIndexOutOfBoundsException`（下标越界异常）。

注意：异常和错误的区别：异常能被程序本身处理，错误是无法处理。

Throwable 类常用方法

- `public String getMessage()`: 返回异常发生时的简要描述
- `public String toString()`: 返回异常发生时的详细信息
- `public String getLocalizedMessage()`: 返回异常对象的本地化信息。使用 `Throwable` 的子类覆盖这个方法，可以生成本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与 `getMessage()` 返回的结果相同
- `public void printStackTrace()`: 在控制台上打印 `Throwable` 对象封装的异常信息

异常处理总结

- `try` 块：用于捕获异常。其后可接零个或多个 `catch` 块，如果没有 `catch` 块，则必须跟一个 `finally` 块。
- `catch` 块：用于处理 `try` 捕获到的异常。
- `finally` 块：无论是否捕获或处理异常，`finally` 块里的语句都会被执行。当在 `try` 块或 `catch` 块中遇到 `return` 语句时，`finally` 语句块将在方法返回之前被执行。

在以下 4 种特殊情况下，`finally` 块不会被执行：

1. 在 `finally` 语句块第一行发生了异常。因为在其他行，`finally` 块还是会得到执行
2. 在前面的代码中用了 `System.exit(int)` 已退出程序。`exit` 是带参函数；若该语句在异常语句之后，`finally` 会执行
3. 程序所在的线程死亡。
4. 关闭 CPU。

下面这部分内容来自 issue:<https://github.com/Snailclimb/JavaGuide/issues/190>。

注意：当 `try` 语句和 `finally` 语句中都有 `return` 语句时，在方法返回之前，`finally` 语句的内容将被执行，并且 `finally` 语句的返回值将会覆盖原始的返回值。如下：

```
public static int f(int value) {  
    try {  
        return value * value;  
    } finally {  
        if (value == 2) {  
            return 0;  
        }  
    }  
}
```

如果调用 `f(2)`，返回值将是 0，因为 `finally` 语句的返回值覆盖了 `try` 语句块的返回值。

33 Java 序列化中如果有些字段不想进行序列化，怎么办？

对于不想进行序列化的变量，使用 `transient` 关键字修饰。

`transient` 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化；当对象被反序列化时，被 `transient` 修饰的变量值不会被持久化和恢复。`transient` 只能修饰变量，不能修饰类和方法。

34 获取用键盘输入常用的两种方法

方法 1：通过 Scanner

```
Scanner input = new Scanner(System.in);
String s = input.nextLine();
input.close();
```

方法 2：通过 BufferedReader

```
BufferedReader input = new BufferedReader(new
InputStreamReader(System.in));
String s = input.readLine();
```

35 Java 中 IO 流

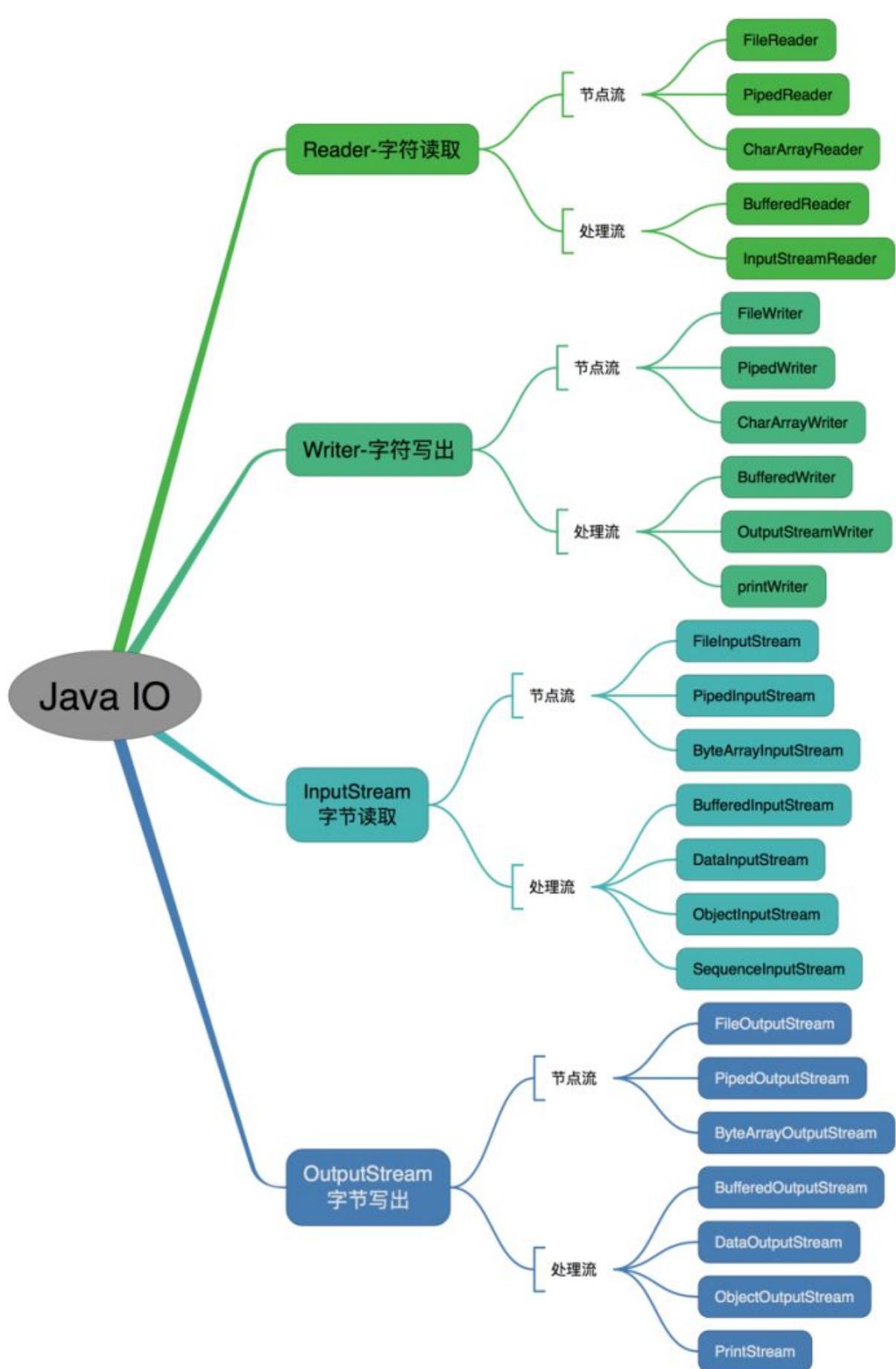
Java 中 IO 流分为几种？

- 按照流的流向分，可以分为输入流和输出流；
- 按照操作单元划分，可以划分为字节流和字符流；
- 按照流的角色划分为节点流和处理流。

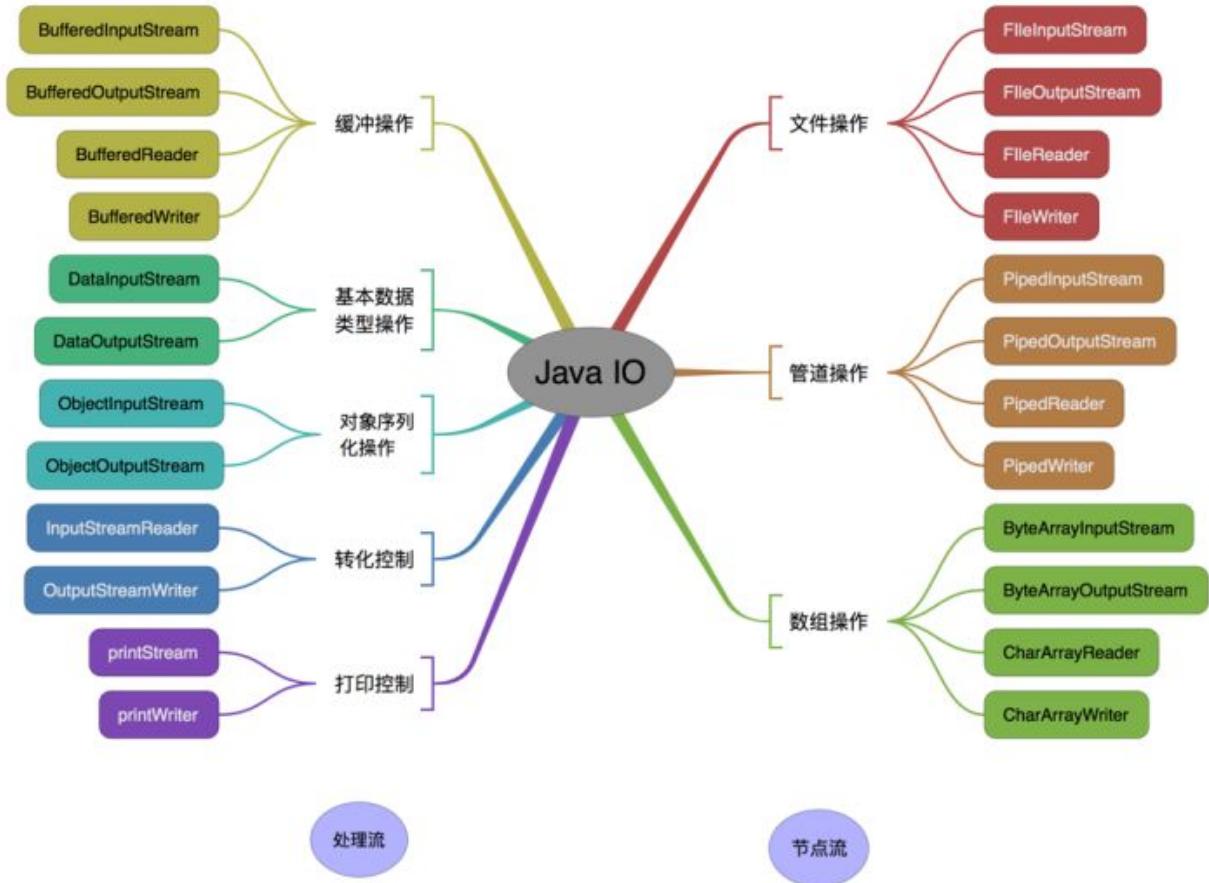
Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java I0 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

- `InputStream/Reader`: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
- `OutputStream/Writer`: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：



按操作对象分类结构图：



既然有了字节流,为什么还要有字符流?

问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？

回答：字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

BIO,NIO,AIO 有什么区别?

- **BIO (Blocking I/O)**: 同步阻塞 I/O 模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机 1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。
- **NIO (Non-blocking/New I/O)**: NIO 是一种同步非阻塞的 I/O 模型，在 Java 1.4 中引入了 NIO 框架，对应 java.nio 包，提供了 Channel, Selector, Buffer 等抽象。NIO 中的 N 可以理解为 Non-blocking，不单纯是 New。它支持面向缓冲的，基于通道的 I/O 操作方法。NIO 提供了与传统 BIO 模型中的 `Socket` 和 `ServerSocket` 相对应的 `SocketChannel` 和 `ServerSocketChannel` 两种不同的套接字通道实现，两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞 I/O 来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发
- **AIO (Asynchronous I/O)**: AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2，它是异步非阻塞的 I/O 模型。异步 I/O 是基于事件和回调机制实现的，也就是应用操作之后会直接返

回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步 IO 的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO 操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

36. 常见关键字总结:static,final,this,super

详见笔主的这篇文章：<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/basic/final,static,this,super.md>

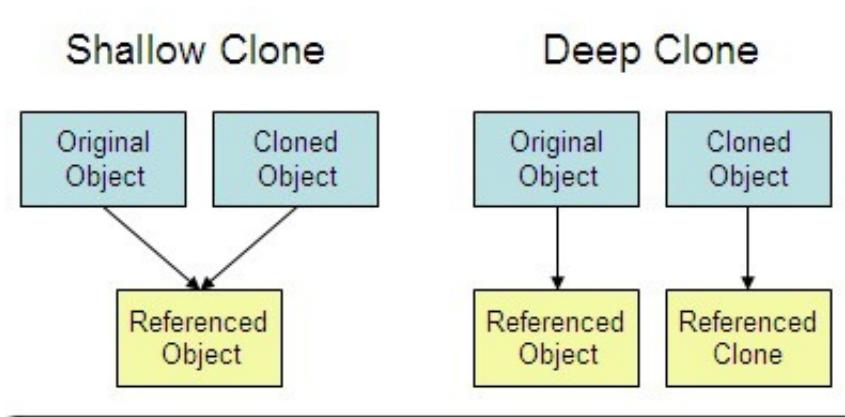
37. Collections 工具类和 Arrays 工具类常见方法总结

详见笔主的这篇文章：

<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/basic/Arrays,CollectionsCommonMethods.md>

38. 深拷贝 vs 浅拷贝

1. 浅拷贝：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。
2. 深拷贝：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。



参考

- <https://stackoverflow.com/questions/1906445/what-is-the-difference-between-jdk-and-jre>
- <https://www.educba.com/oracle-vs-openjdk/>
- <https://stackoverflow.com/questions/22358071/differences-between-oracle-jdk-and-openjdk?answertab=active#tab-top>

公众号

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《JavaGuide 面试突击版》：由本文档衍生的专为面试而生的《JavaGuide 面试突击版》版本公众号后台回复 "Java 面试突击" 即可免费领取！

Java 工程师必备学习资源：一些 Java 工程师常用学习资源公众号后台回复关键字“1”即可免费无套路获取。



2.2 Java集合

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

2.2.1 说说List,Set,Map三者的区别？

- **List(对付顺序的好帮手)：** List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象
- **Set(注重独一无二的性质)：** 不允许重复的集合。不会有多个元素引用相同的对象。
- **Map(用Key来搜索的专家)：** 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

2.2.2 ArrayList 与 LinkedList 区别？

- 1. 是否保证线程安全：`ArrayList` 和 `LinkedList` 都是不同步的，也就是不保证线程安全；
- 2. 底层数据结构：`ArrayList` 底层使用的是 `Object` 数组；`LinkedList` 底层使用的是双向链表 数据结构（JDK1.6之前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别，下面有介绍到！）
- 3. 插入和删除是否受元素位置的影响：
① `ArrayList` 采用数组存储，所以插入和删除元素的时间复杂度受元素位置的影响。比如：执行 `add(E e)` 方法的时候，`ArrayList` 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是 $O(1)$ 。但是如果要在指定位置 `i` 插入和删除元素的话 (`add(int index, E element)`) 时间复杂度就为 $O(n-i)$ 。因为在进行上述操作的时候集合中第 `i` 和第 `i` 个元素之后的($n-i$)个元素都要执行向后位/向前移一位的操作。
② `LinkedList` 采用链表存储，所以对于 `add(E e)` 方法的插入，删除元素时间复杂度不受元素位置的影响，近似 $O(1)$ ，如果是要在指定位置 `i` 插入和删除元素的话 (`(add(int index, E element))`) 时间复杂度近似为 $O(n)$ 因为需要先移动到指定位置再插入。
- 4. 是否支持快速随机访问：`LinkedList` 不支持高效的随机元素访问，而 `ArrayList` 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 `get(int index)` 方法)。
- 5. 内存空间占用：`ArrayList` 的空间浪费主要体现在在list列表的结尾会预留一定的容量空间，而`LinkedList`的空间花费则体现在它的每一个元素都需要消耗比`ArrayList`更多的空间（因

为要存放直接后继和直接前驱以及数据)。

补充内容:RandomAccess接口

```
public interface RandomAccess {  
}
```

查看源码我们发现实际上 `RandomAccess` 接口中什么都没有定义。所以，在我看来 `RandomAccess` 接口不过是一个标识罢了。标识什么？标识实现这个接口的类具有随机访问功能。

在 `binarySearch ()` 方法中，它要判断传入的list 是否 `RandomAccess` 的实例，如果是，调用 `indexedBinarySearch ()` 方法，如果不是，那么调用 `iteratorBinarySearch ()` 方法

```
public static <T>  
    int binarySearch(List<? extends Comparable<? super T>> list, T key)  
{  
    if (list instanceof RandomAccess || list.size()<BINARYSEARCH_THRESHOLD)  
        return Collections.indexedBinarySearch(list, key);  
    else  
        return Collections.iteratorBinarySearch(list, key);  
}
```

`ArrayList` 实现了 `RandomAccess` 接口，而 `LinkedList` 没有实现。为什么呢？我觉得还是和底层数据结构有关！`ArrayList` 底层是数组，而 `LinkedList` 底层是链表。数组天然支持随机访问，时间复杂度为 $O(1)$ ，所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素，时间复杂度为 $O(n)$ ，所以不支持快速随机访问。, `ArrayList` 实现了 `RandomAccess` 接口，就表明了他具有快速随机访问功能。`RandomAccess` 接口只是标识，并不是说 `ArrayList` 实现 `RandomAccess` 接口才具有快速随机访问功能的！

下面再总结一下 `list` 的遍历方式选择：

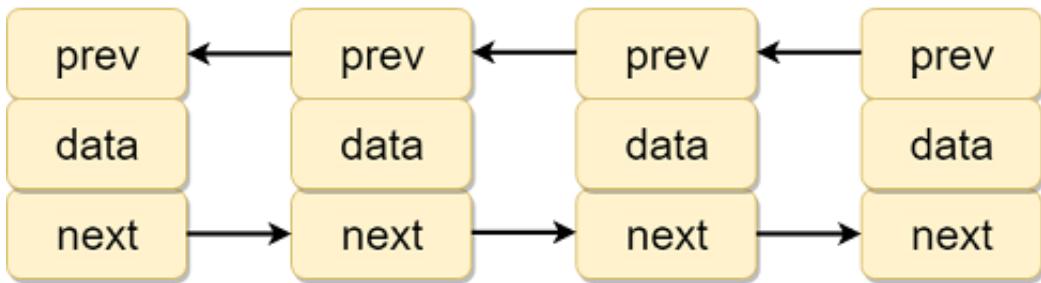
- 实现了 `RandomAccess` 接口的 `list`, 优先选择普通 `for` 循环，其次 `foreach`,
- 未实现 `RandomAccess` 接口的 `list`, 优先选择 `iterator` 遍历 (`foreach` 遍历底层也是通过 `iterator` 实现的,)，`list` 的数据，千万不要使用普通 `for` 循环

补充内容:双向链表和双向循环链表

双向链表：包含两个指针，一个 `prev` 指向前一个节点，一个 `next` 指向后一个节点。

| 另外推荐一篇把双向链表讲清楚的文章：<https://juejin.im/post/5b5d1a9af265da0f47352f14>

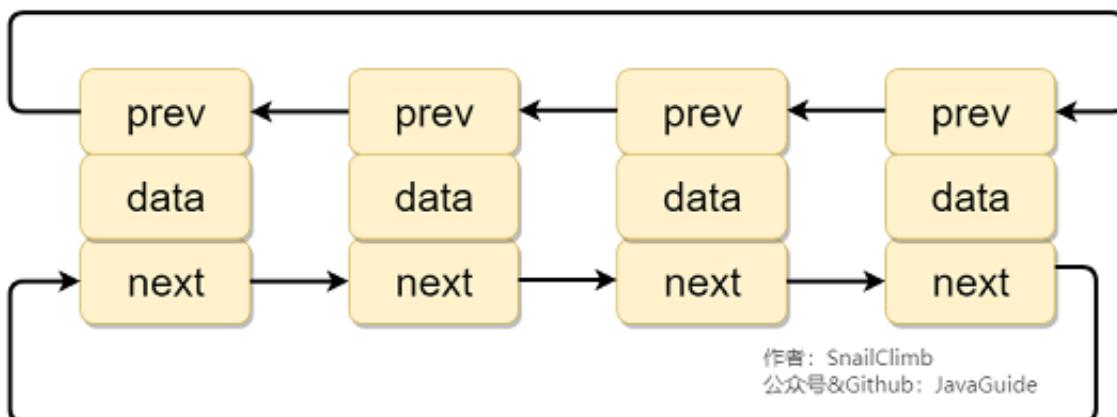
双向链表



作者: SnailClimb
公众号&Github: JavaGuide

双向循环链表: 最后一个节点的 next 指向head, 而 head 的prev指向最后一个节点, 构成一个环。

双向循环链表



作者: SnailClimb
公众号&Github: JavaGuide

2.2.3 ArrayList 与 Vector 区别呢?为什么要用ArrayList取代Vector呢?

`Vector`类的所有方法都是同步的。可以由两个线程安全地访问一个`Vector`对象、但是一个线程访问`Vector`的话代码要在同步操作上耗费大量的时间。

`ArrayList`不是同步的, 所以在不需要保证线程安全时建议使用`ArrayList`。

2.2.4 说一说 ArrayList 的扩容机制吧

详见笔主的这篇文章:[通过源码一步一步分析ArrayList 扩容机制](#)

2.2.5 HashMap 和 Hashtable 的区别

1. 线程是否安全: `HashMap` 是非线程安全的, `HashTable` 是线程安全的; `HashTable` 内部的方法基本都经过`synchronized` 修饰。 (如果你要保证线程安全的话就使用 `ConcurrentHashMap` 吧!) ;
2. 效率: 因为线程安全的问题, `HashMap` 要比 `HashTable` 效率高一点。另外, `HashTable` 基本被淘汰, 不要在代码中使用它;
3. 对Null key 和Null value的支持: `HashMap` 中, null 可以作为键, 这样的键只有一个, 可以有一个或多个键所对应的值为 null。。但是在 `HashTable` 中 put 进的键值只要有一个 null, 直接抛出 `NullPointerException`。

4. 初始容量大小和每次扩充容量大小的不同：①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的 $2n+1$ 。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的 `tableSizeFor()` 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小，后面会介绍到为什么是2的幂次方。
5. 底层数据结构：JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

HashMap 中带有初始容量的构造函数：

```

public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial
capacity: " +
                                         initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                         loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

```

下面这个方法保证了 HashMap 总是使用2的幂作为哈希表的大小。

```

/**
 * Returns a power of two size for the given target capacity.
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY
: n + 1;
}

```

2.2.6 HashMap 和 HashSet区别

如果你看过 `HashSet` 源码的话就应该知道：`HashSet` 底层就是基于 `HashMap` 实现的。（`HashSet` 的源码非常非常少，因为除了 `clone()`、`writeObject()`、`readObject()` 是 `HashSet` 自己不得不实现之外，其他方法都是直接调用 `HashMap` 中的方法。）

HashMap	HashSet
实现了Map接口	实现Set接口
存储键值对	仅存储对象
调用 <code>put ()</code> 向 map中添加元素	调用 <code>add ()</code> 方法向Set中添加元素
HashMap使用键 (Key) 计算 Hashcode	HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以 <code>equals()</code> 方法用来判断对象的相等性，

2.2.7 HashSet如何检查重复

当你把对象加入 `HashSet` 时，`HashSet` 会先计算对象的 `hashcode` 值来判断对象加入的位置，同时也会与其他加入的对象的 `hashcode` 值作比较，如果没有相符的 `hashcode`，`HashSet` 会假设对象没有重复出现。但是如果发现有相同 `hashcode` 值的对象，这时会调用 `equals ()` 方法来检查 `hashcode` 相等的对象是否真的相同。如果两者相同，`HashSet` 就不会让加入操作成功。（摘自我的 Java 启蒙书《Head first java》第二版）

`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. 两个对象相等，对两个 `equals` 方法返回 `true`
3. 两个对象有相同的 `hashcode` 值，它们也不一定是相等的
4. 综上，`equals` 方法被覆盖过，则 `hashCode` 方法也必须被覆盖
5. `hashCode()` 的默认行为是对堆上的对象产生独特值。如果没有重写 `hashCode()`，则该 class 的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

`=` 与 `equals` 的区别

1. `=` 是判断两个变量或实例是不是指向同一个内存空间 `equals` 是判断两个变量或实例所指向的内存空间的值是不是相同
2. `=` 是指对内存地址进行比较 `equals()` 是对字符串的内容进行比较
3. `=` 指引用是否相同 `equals()` 指的是值是否相同

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

2.2.8 HashMap的底层实现

JDK1.8之前

JDK1.8 之前 `HashMap` 底层是 数组和链表 结合在一起使用也就是 链表散列。`HashMap` 通过 `key` 的 `hashCode` 经过扰动函数处理过后得到 `hash` 值，然后通过 $(n - 1) \& hash$ 判断当前元素存放的位置（这里的 `n` 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 `hash` 值以及 `key` 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的 hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码:

JDK 1.8 的 hash方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

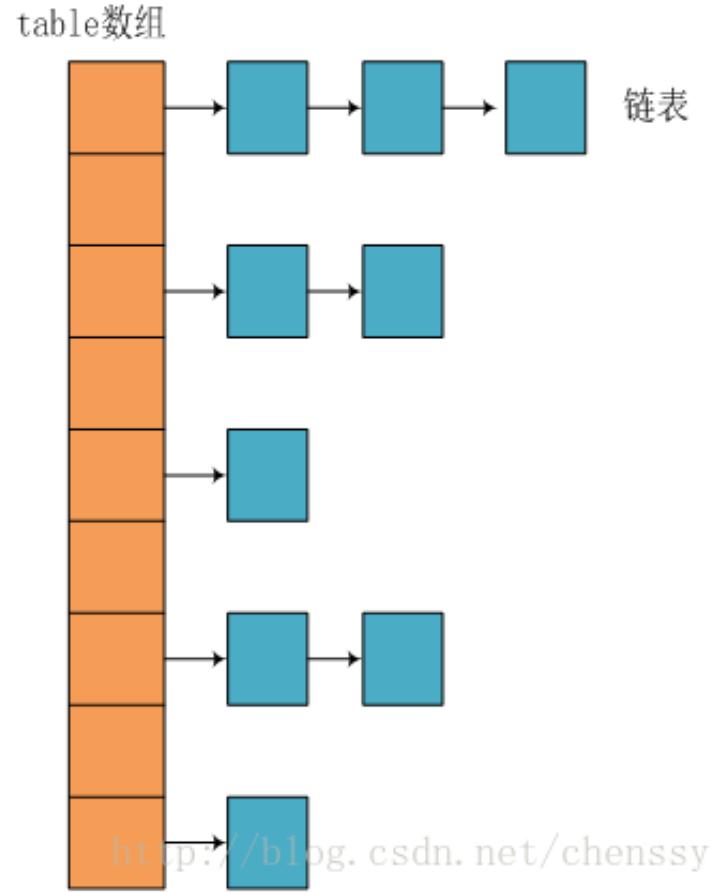
```
static final int hash(Object key) {  
    int h;  
    // key.hashCode(): 返回散列值也就是hashcode  
    // ^ : 按位异或  
    // >>>:无符号右移，忽略符号位，空位都以0补齐  
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}
```

对比一下 JDK1.7的 HashMap 的 hash 方法源码.

```
static int hash(int h) {  
    // This function ensures that hashCodes that differ only by  
    // constant multiples at each bit position have a bounded  
    // number of collisions (approximately 8 at default load factor).  
  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

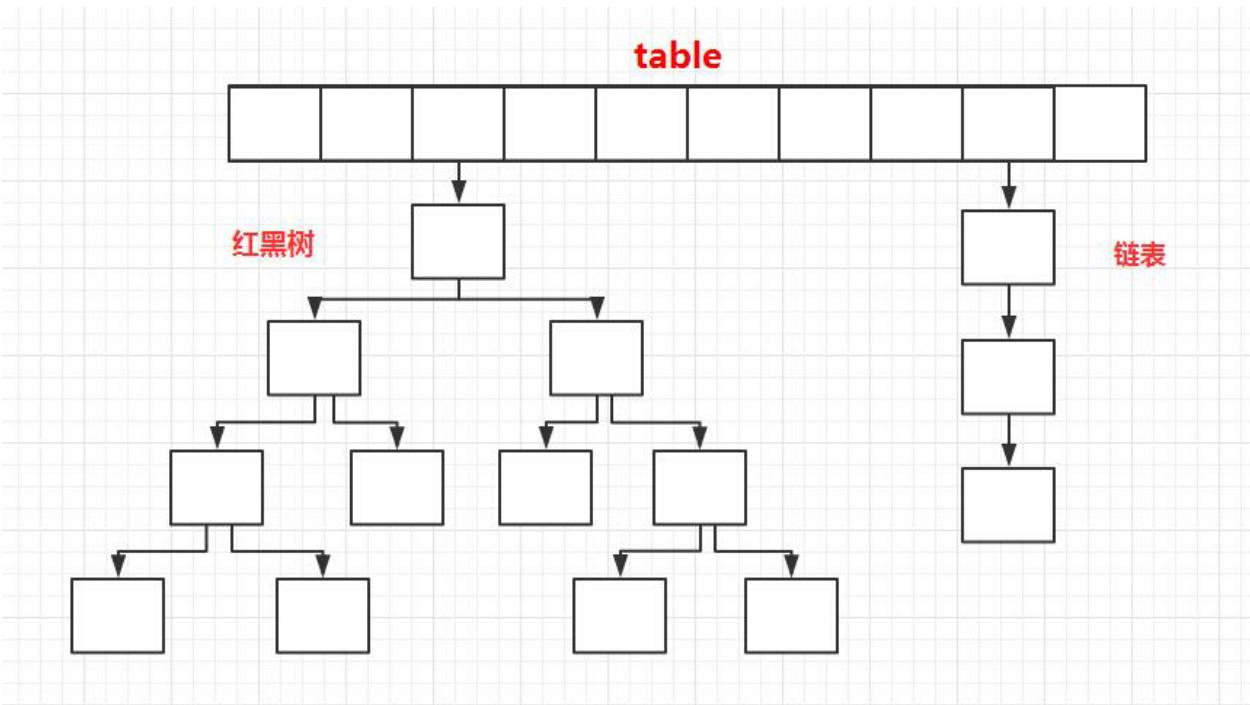
相比于 JDK1.8 的 hash 方法 ， JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。

所谓“拉链法”就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可。



JDK1.8之后

相比于之前的版本，JDK1.8之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。



TreeMap、TreeSet以及JDK1.8之后的HashMap底层都用到了红黑树。红黑树就是为了解决二叉查找树的缺陷，因为二叉查找树在某些情况下会退化成一个线性结构。

推荐阅读：

- 《Java 8系列之重新认识HashMap》 : <https://zhuanlan.zhihu.com/p/21673805>

2.2.9 HashMap 的长度为什么是2的幂次方

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之前还要先做对数组的长度取模运算，得到的余数才能用来存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ $(n - 1) \& hash$ ”。(n代表数组长度)。这也就解释了 HashMap 的长度为什么是2的幂次方。

这个算法应该如何设计呢？

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：“取余(%)操作中如果除数是2的幂次则等价于与其除数减一的与(&)操作（也就是说 $hash \% length = hash \& (length - 1)$ 的前提是 length 是2的 n 次方；）。”并且 采用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。

2.2.10 HashMap 多线程操作导致死循环问题

主要原因在于 并发下的Rehash 会造成元素之间会形成一个循环链表。不过，jdk 1.8 后解决了这个问题，但是还是不建议在多线程下使用 HashMap，因为多线程下使用 HashMap 还是会存在其他问题比如数据丢失。并发环境下推荐使用 ConcurrentHashMap 。

详情请查看：<https://coolshell.cn/articles/9606.html>

2.2.11 ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

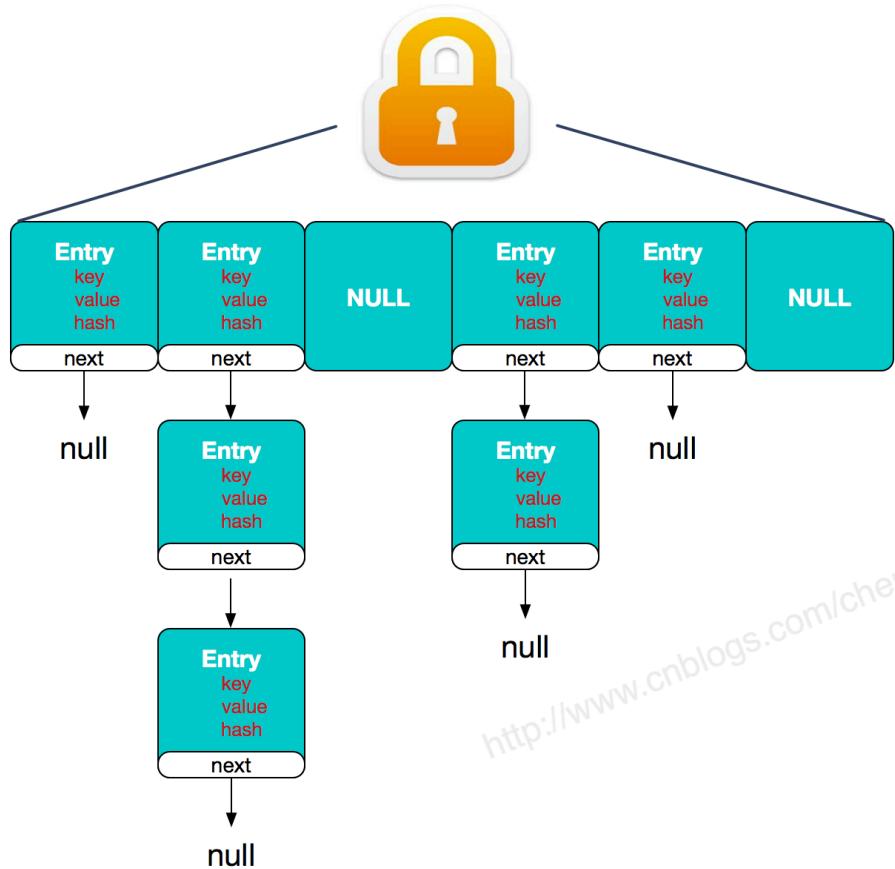
- **底层数据结构：** JDK1.7的 ConcurrentHashMap 底层采用 分段的数组+链表 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 数组+链表 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；
- **实现线程安全的方式（重要）：** ① 在JDK1.7的时候，ConcurrentHashMap（分段锁） 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized 锁做了很多优化） 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构，但是已经简化了属性，只是为了兼容旧版本；② **Hashtable(同一把锁)** : 使用 synchronized 来保证线程安全，效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用 put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。

两者的对比图：

图片来源：<http://www.cnblogs.com/chengxiao/p/6842045.html>

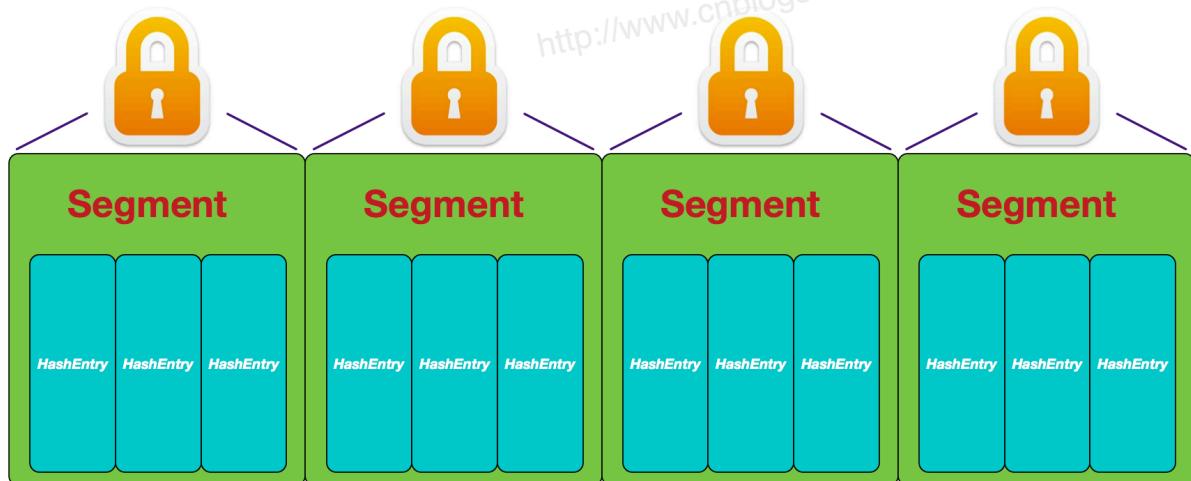
HashTable:

HashTable 全表锁

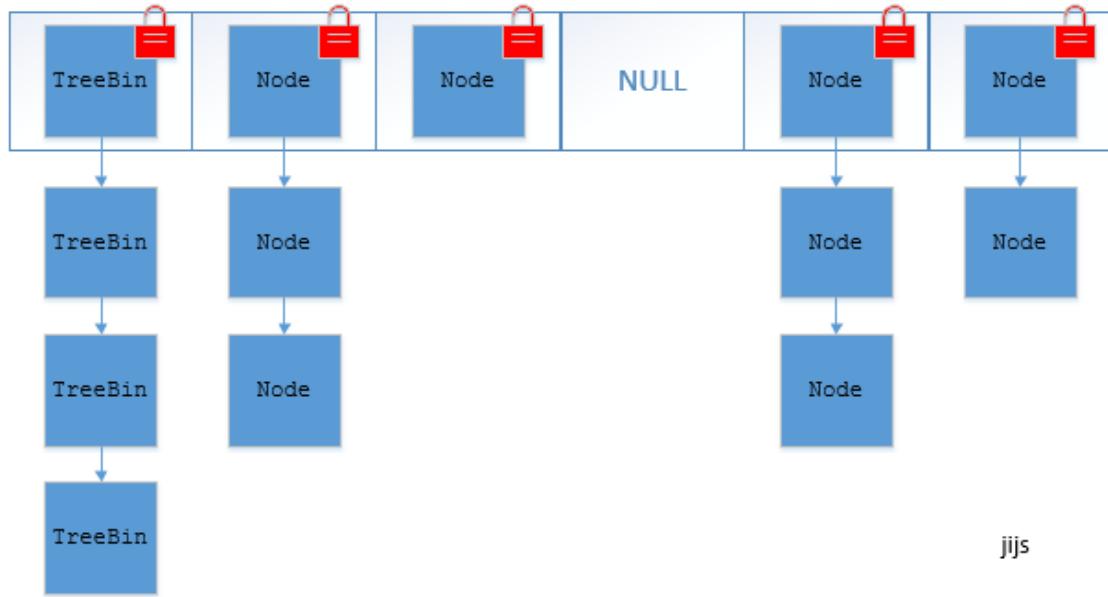


JDK1.7的ConcurrentHashMap：

ConcurrentHashMap 分段锁



JDK1.8的ConcurrentHashMap (TreeBin: 红黑二叉树节点 Node: 链表节点) :



2.2.12 ConcurrentHashMap线程安全的具体实现方式/底层具体实现

JDK1.7（上面有示意图）

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成。

Segment 实现了 ReentrantLock，所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

```
static class Segment<K,V> extends ReentrantLock implements Serializable
{
}
```

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment的锁。

JDK1.8（上面有示意图）

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。Java 8在链表长度超过一定阈值（8）时将链表（寻址时间复杂度为O(N)）转换为红黑树（寻址时间复杂度为O(log(N))）

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

2.2.13 comparable 和 Comparator的区别

- comparable接口实际上出自java.lang包 它有一个 `compareTo(Object obj)` 方法用来排

序

- comparator接口实际上是出自 java.util 包它有一个 compare(Object obj1, Object obj2) 方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写 compareTo() 方法或 compare() 方法，当我们需要对某一个集合实现两种排序方式，比如一个 song 对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写 compareTo() 方法和使用自制的 Comparator 方法或者以两个 Comparator 来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的 Collections.sort().

Comparator定制排序

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
arrayList.add(-1);
arrayList.add(3);
arrayList.add(3);
arrayList.add(-5);
arrayList.add(7);
arrayList.add(4);
arrayList.add(-9);
arrayList.add(-7);
System.out.println("原始数组:");
System.out.println(arrayList);
// void reverse(List list): 反转
Collections.reverse(arrayList);
System.out.println("Collections.reverse(arrayList):");
System.out.println(arrayList);

// void sort(List list), 按自然排序的升序排序
Collections.sort(arrayList);
System.out.println("Collections.sort(arrayList):");
System.out.println(arrayList);
// 定制排序的用法
Collections.sort(arrayList, new Comparator<Integer>() {
    @Override
    public int compare(Integer o1, Integer o2) {
        return o2.compareTo(o1);
    }
});
System.out.println("定制排序后: ");
System.out.println(arrayList);
```

Output:

原始数组:

```
[-1, 3, 3, -5, 7, 4, -9, -7]
Collections.reverse(arrayList):
[-7, -9, 4, 7, -5, 3, 3, -1]
Collections.sort(arrayList):
[-9, -7, -5, -1, 3, 3, 4, 7]
定制排序后:
[7, 4, 3, 3, -1, -5, -7, -9]
```

重写compareTo方法实现按年龄来排序

```
// person对象没有实现Comparable接口，所以必须实现，这样才不会出错，才可
以使treemap中的数据按顺序排列
// 前面一个例子的String类已经默认实现了Comparable接口，详细可以查看
String类的API文档，另外其他
// 像Integer类等都已经实现了Comparable接口，所以不需要另外实现了

public class Person implements Comparable<Person> {
    private String name;
    private int age;

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    /**
     * 按年龄升序排序
     */
    @Override
    public int compareTo(Person other) {
        if (this.age < other.age) {
            return -1;
        } else if (this.age > other.age) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

```

    * TODO重写compareTo方法实现按年龄来排序
    */
@Override
public int compareTo(Person o) {
    // TODO Auto-generated method stub
    if (this.age > o.getAge()) {
        return 1;
    } else if (this.age < o.getAge()) {
        return -1;
    }
    return age;
}
}

```

```

public static void main(String[] args) {
    TreeMap<Person, String> pdata = new TreeMap<Person, String>();
    pdata.put(new Person("张三", 30), "zhangsan");
    pdata.put(new Person("李四", 20), "lisi");
    pdata.put(new Person("王五", 10), "wangwu");
    pdata.put(new Person("小红", 5), "xiaohong");
    // 得到key的值的同时得到key所对应的值
    Set<Person> keys = pdata.keySet();
    for (Person key : keys) {
        System.out.println(key.getAge() + "-" + key.getName());
    }
}

```

Output:

```

5-小红
10-王五
20-李四
30-张三

```

2.2.14 集合框架底层数据结构总结

Collection

1. List

- **ArrayList**: Object数组
- **Vector**: Object数组
- **LinkedList**: 双向链表(JDK1.6之前为循环链表, JDK1.7取消了循环)

2. Set

- **HashSet (无序, 唯一)** : 基于 HashMap 实现的, 底层采用 HashMap 来保存元素
- **LinkedHashSet**: LinkedHashSet 继承于 HashSet, 并且其内部是通过 LinkedHashMap 来实现的。有点类似于我们之前说的 LinkedHashMap 其内部是基于 HashMap 实现一样, 不过还是有一点点区别的
- **TreeSet (有序, 唯一)** : 红黑树(自平衡的排序二叉树)

Map

- **HashMap**: JDK1.8之前HashMap由数组+链表组成的, 数组是HashMap的主体, 链表则是主要为了解决哈希冲突而存在的 (“拉链法”解决冲突)。JDK1.8以后在解决哈希冲突时有了较大的变化, 当链表长度大于阈值 (默认为8) 时, 将链表转化为红黑树, 以减少搜索时间
- **LinkedHashMap**: LinkedHashMap 继承自 HashMap, 所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外, LinkedHashMap 在上面结构的基础上, 增加了一条双向链表, 使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作, 实现了访问顺序相关逻辑。详细可以查看: [《LinkedHashMap 源码详细分析 \(JDK1.8\)》](#)
- **Hashtable**: 数组+链表组成的, 数组是 HashMap 的主体, 链表则是主要为了解决哈希冲突而存在的
- **TreeMap**: 红黑树 (自平衡的排序二叉树)

2.2.15 如何选用集合?

主要根据集合的特点来选用, 比如我们需要根据键值获取到元素值时就选用Map接口下的集合, 需要排序时选择TreeMap, 不需要排序时就选择HashMap, 需要保证线程安全就选用ConcurrentHashMap.当我们只需要存放元素值时, 就选择实现Collection接口的集合, 需要保证元素唯一时选择实现Set接口的集合比如TreeSet或HashSet, 不需要就选择实现List接口的比如ArrayList或LinkedList, 然后再根据实现这些接口的集合的特点来选用。

如果大家想要实时关注我更新的文章以及分享的干货的话, 可以关注我的公众号。

《JavaGuide 面试突击版》: 由本文档衍生的专为面试而生的《JavaGuide 面试突击版》版本公众号后台回复 “Java 面试突击” 即可免费领取!

Java 工程师必备学习资源: 一些 Java 工程师常用学习资源公众号后台回复关键字 “1” 即可免费无套路获取。



2.3 多线程

作者: Guide哥。

介绍: Github 70k Star 项目 [JavaGuide](#) (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

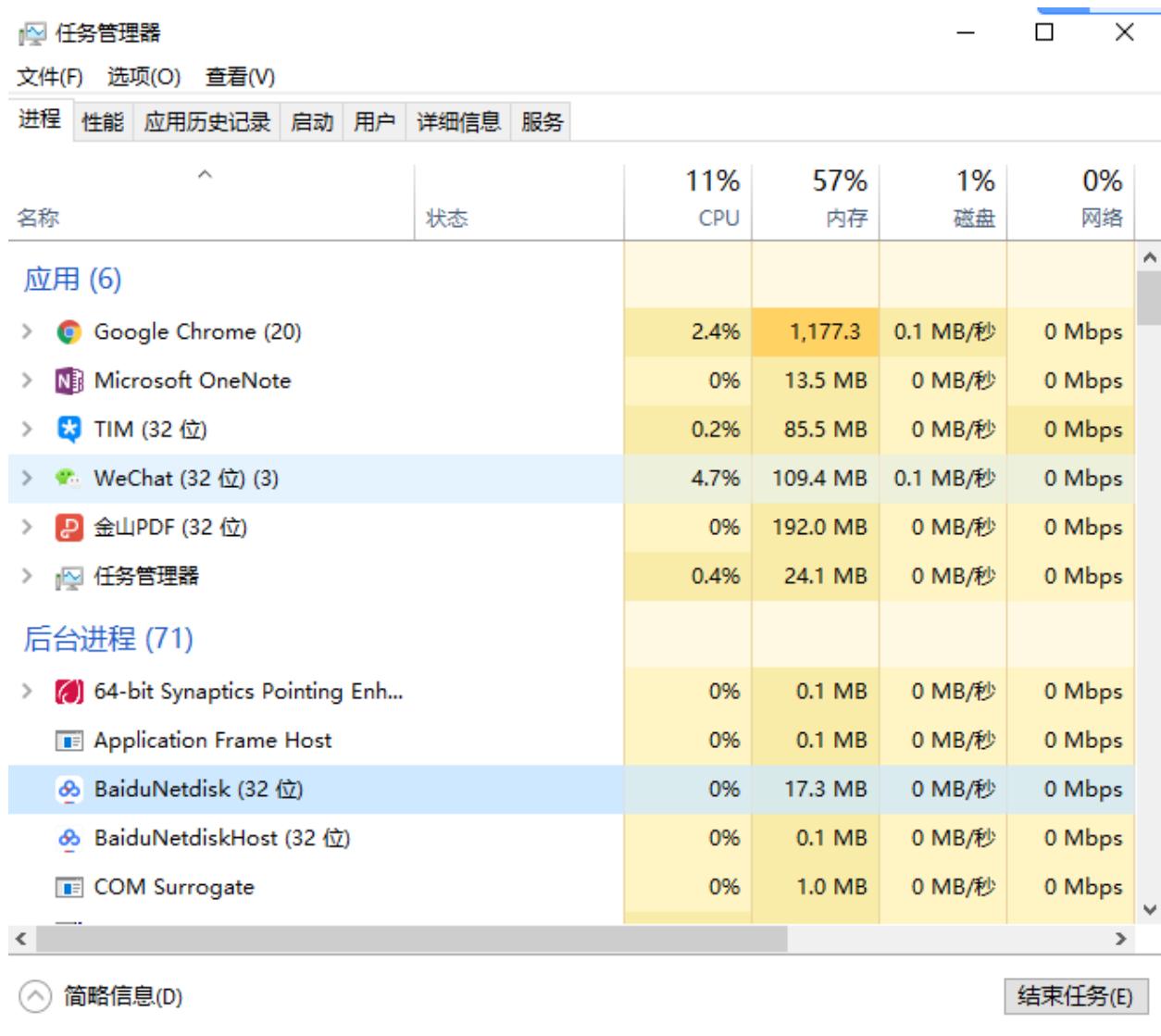
2.3.1. 什么是线程和进程?

何为进程?

进程是程序的一次执行过程，是系统运行程序的基本单位，因此进程是动态的。系统运行一个程序即是一个进程从创建，运行到消亡的过程。

在 Java 中，当我们启动 main 函数时其实就是启动了一个 JVM 的进程，而 main 函数所在的线程就是这个进程中的一个线程，也称主线程。

如下图所示，在 windows 中通过查看任务管理器的方式，我们就可以清楚看到 window 当前运行的进程 (.exe 文件的运行)。



The screenshot shows the Windows Task Manager with the 'Processes' tab selected. It displays a list of running applications and their resource usage. The columns are: Name, Status, CPU, Memory, Disk, and Network. The 'CPU' column is highlighted.

名称	状态	11% CPU	57% 内存	1% 磁盘	0% 网络
应用 (6)					
> Google Chrome (20)		2.4%	1,177.3	0.1 MB/秒	0 Mbps
> Microsoft OneNote		0%	13.5 MB	0 MB/秒	0 Mbps
> * TIM (32 位)		0.2%	85.5 MB	0 MB/秒	0 Mbps
> WeChat (32 位) (3)		4.7%	109.4 MB	0.1 MB/秒	0 Mbps
> 金山PDF (32 位)		0%	192.0 MB	0 MB/秒	0 Mbps
> 任务管理器		0.4%	24.1 MB	0 MB/秒	0 Mbps
后台进程 (71)					
> 64-bit Synaptics Pointing Enh...		0%	0.1 MB	0 MB/秒	0 Mbps
Application Frame Host		0%	0.1 MB	0 MB/秒	0 Mbps
BaiduNetdisk (32 位)		0%	17.3 MB	0 MB/秒	0 Mbps
BaiduNetdiskHost (32 位)		0%	0.1 MB	0 MB/秒	0 Mbps
COM Surrogate		0%	1.0 MB	0 MB/秒	0 Mbps

何为线程?

线程与进程相似，但线程是一个比进程更小的执行单位。一个进程在其执行的过程中可以产生多个线程。与进程不同的是同类的多个线程共享进程的堆和方法区资源，但每个线程有自己的程序计数器、虚拟机栈和本地方法栈，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多，也正因为如此，线程也被称为轻量级进程。

Java 程序天生就是多线程程序，我们可以通过 JMX 来看一下一个普通的 Java 程序有哪些线程，代码如下。

```
public class MultiThread {
    public static void main(String[] args) {
        // 获取 Java 线程管理 MXBean
        ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
        // 不需要获取同步的 monitor 和 synchronizer 信息，仅获取线程和线程堆
        // 栈信息
        ThreadInfo[] threadInfos = threadMXBean.dumpAllThreads(false,
        false);
        // 遍历线程信息，仅打印线程 ID 和线程名称信息
        for (ThreadInfo threadInfo : threadInfos) {
            System.out.println("[" + threadInfo.getThreadId() + "] " +
            threadInfo.getThreadName());
        }
    }
}
```

上述程序输出如下（输出内容可能不同，不用太纠结下面每个线程的作用，只用知道 main 线程执行 main 方法即可）：

```
[5] Attach Listener //添加事件
[4] Signal Dispatcher // 分发处理给 JVM 信号的线程
[3] Finalizer //调用对象 finalize 方法的线程
[2] Reference Handler //清除 reference 线程
[1] main //main 线程,程序入口
```

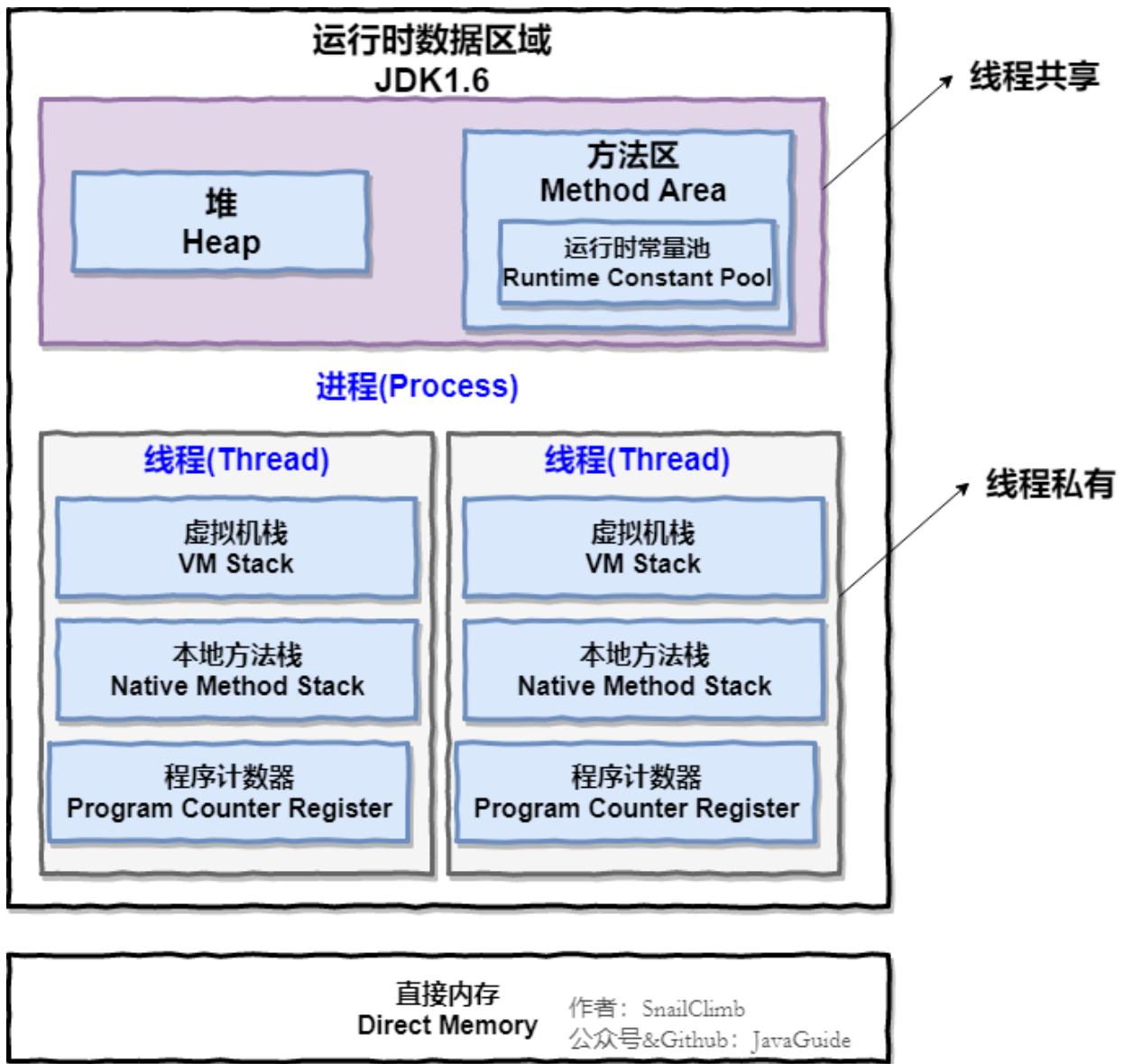
从上面的输出内容可以看出：一个 Java 程序的运行是 main 线程和多个其他线程同时运行。

2.3.2. 请简要描述线程与进程的关系,区别及优缺点?

从 JVM 角度说进程和线程之间的关系

图解进程和线程的关系

下图是 Java 内存区域，通过下图我们从 JVM 的角度来说一下线程和进程之间的关系。如果你对 Java 内存区域（运行时数据区）这部分知识不太了解的话可以阅读一下这篇文章：[《可能是把 Java 内存区域讲的最清楚的一篇文章》](#)



从上图可以看出：一个进程中可以有多个线程，多个线程共享进程的堆和方法区（JDK1.8之后的元空间）资源，但是每个线程有自己的程序计数器、虚拟机栈 和 本地方法栈。

总结：线程 是 进程 划分成的更小的运行单位。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。线程执行开销小，但不利于资源的管理和保护；而进程正相反

下面是该知识点的扩展内容！

下面来思考这样一个问题：为什么程序计数器、虚拟机栈和本地方法栈是线程私有的呢？为什么堆和方法区是线程共享的呢？

程序计数器为什么是私有的？

程序计数器主要有下面两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。

所以，程序计数器私有主要是为了线程切换后能恢复到正确的执行位置。

虚拟机栈和本地方法栈为什么是私有的？

- **虚拟机栈：**每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程。
- **本地方法栈：**和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了保证线程中的局部变量不被别的线程访问到，虚拟机栈和本地方法栈是线程私有的。

一句话简单了解堆和方法区

堆和方法区是所有线程共享的资源，其中堆是进程中最大的一块内存，主要用于存放新创建的对象（所有对象都在这里分配内存），方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。

2.3.3. 说说并发与并行的区别？

- **并发：**同一时间段，多个任务都在执行（单位时间内不一定同时执行）；
- **并行：**单位时间内，多个任务同时执行。

2.3.4. 为什么要使用多线程呢？

先从总体上来说：

- **从计算机底层来说：**线程可以比作是轻量级的进程，是程序执行的最小单位，线程间的切换和调度的成本远远小于进程。另外，多核 CPU 时代意味着多个线程可以同时运行，这减少了线程上下文切换的开销。
- **从当代互联网发展趋势来说：**现在的系统动不动就要求百万级甚至千万级的并发量，而多线程并发编程正是开发高并发系统的基础，利用好多线程机制可以大大提高系统整体的并发能力以及性能。

再深入到计算机底层来探讨：

- **单核时代：**在单核时代多线程主要是为了提高 CPU 和 IO 设备的综合利用率。举个例子：当只有一个线程的时候会导致 CPU 计算时，IO 设备空闲；进行 IO 操作时，CPU 空闲。我们可以简单地说这两者的利用率目前都是 50% 左右。但是当有两个线程的时候就不一样了，当一个线程执行 CPU 计算时，另外一个线程可以进行 IO 操作，这样两个的利用率就可以在理想情况下达到 100% 了。
- **多核时代：**多核时代多线程主要是为了提高 CPU 利用率。举个例子：假如我们要计算一个复杂的任务，我们只用一个线程的话，CPU 只会一个 CPU 核心被利用到，而创建多个线程就可以让多个 CPU 核心被利用到，这样就提高了 CPU 的利用率。

2.3.5. 使用多线程可能带来什么问题？

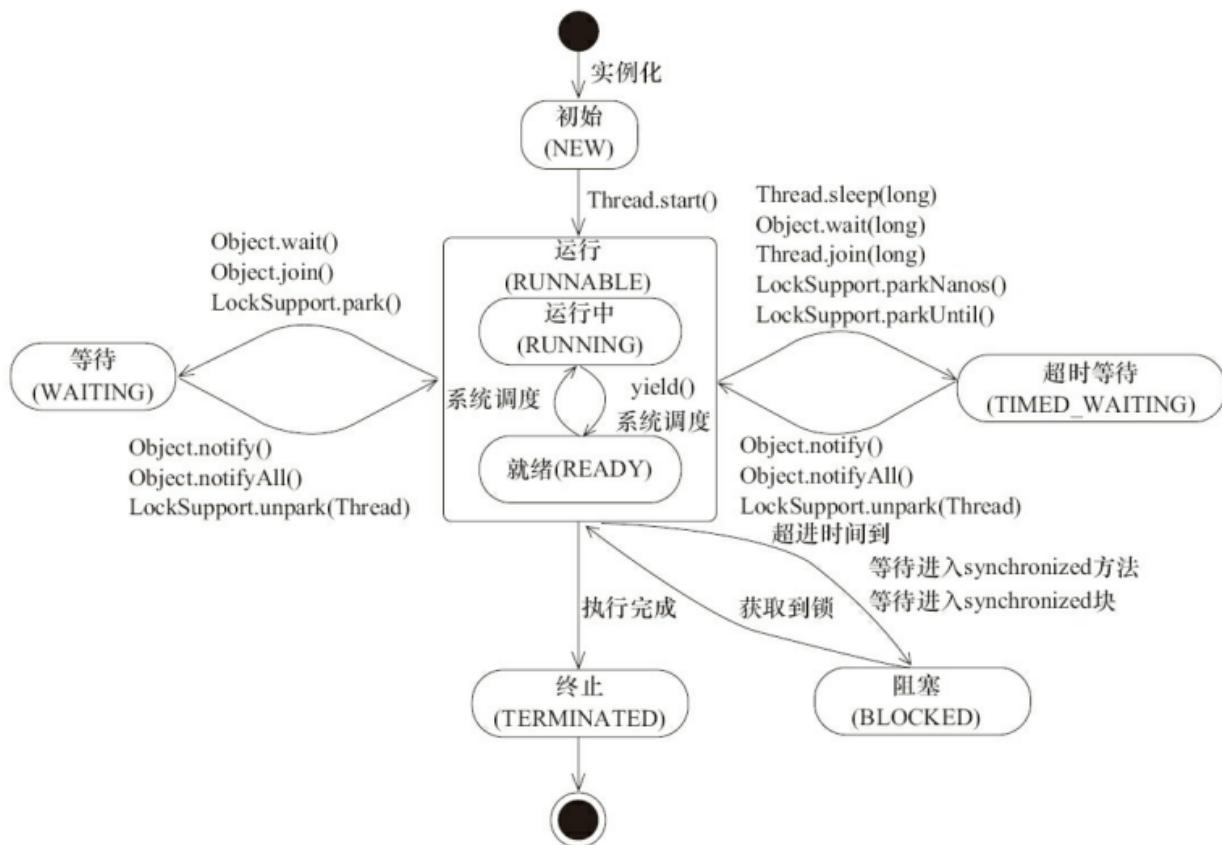
并发编程的目的就是为了能提高程序的执行效率提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：内存泄漏、上下文切换、死锁还有受限于硬件和软件的资源闲置问题。

2.3.6. 说说线程的生命周期和状态？

Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态（图源《Java 并发编程艺术》4.1.4 节）。

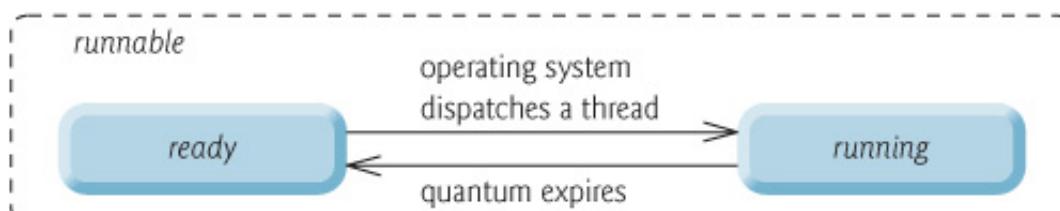
状态名称	说 明
NEW	初始状态，线程被构建，但是还没有调用 start() 方法
RUNNABLE	运行状态，Java 线程将操作系统中的就绪和运行两种状态笼统地称作“运行中”
BLOCKED	阻塞状态，表示线程阻塞于锁
WAITING	等待状态，表示线程进入等待状态，进入该状态表示当前线程需要等待其他线程做出一些特定动作（通知或中断）
TIME_WAITING	超时等待状态，该状态不同于 WAITING，它是可以在指定的时间自行返回的
TERMINATED	终止状态，表示当前线程已经执行完毕

线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换。Java 线程状态变迁如下图所示（图源《Java 并发编程艺术》4.1.4 节）：



由上图可以看出：线程创建之后它将处于 NEW（新建） 状态，调用 `start()` 方法后开始运行，线程这时候处于 READY（可运行） 状态。可运行状态的线程获得了 CPU 时间片（timeslice）后就处于 RUNNING（运行） 状态。

操作系统隐藏 Java 虚拟机（JVM）中的 RUNNABLE 和 RUNNING 状态，它只能看到 RUNNABLE 状态
(图源：[HowToDoInJava: Java Thread Life Cycle and Thread States](#))，所以 Java 系统一般将这两个状态统称为 RUNNABLE（运行中）状态。



当线程执行 `wait()` 方法之后，线程进入 `WAITING`（等待）状态。进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态，而 `TIME_WAITING`（超时等待）状态相当于在等待状态的基础上增加了超时限制，比如通过 `sleep (long millis)` 方法或 `wait (long millis)` 方法可以将 Java 线程置于 `TIMED_WAITING` 状态。当超时时间到达后 Java 线程将会返回到 `RUNNABLE` 状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到 `BLOCKED`（阻塞）状态。线程在执行 `Runnable` 的 `run()` 方法之后将会进入到 `TERMINATED`（终止）状态。

2.3.7. 什么是上下文切换？

多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。

概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换。

上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。

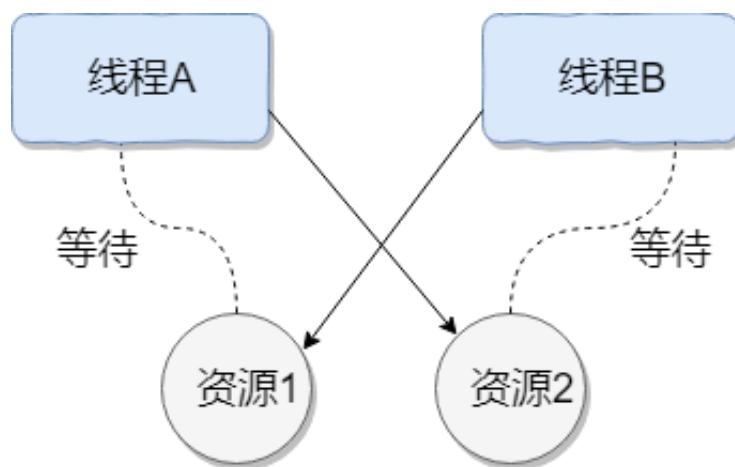
Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

2.3.8. 什么是线程死锁？如何避免死锁？

认识线程死锁

线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过一个例子来说明线程死锁，代码模拟了上图的死锁的情况（代码来源于《并发编程之美》）：

```
public class DeadLockDemo {  
    private static Object resource1 = new Object(); // 资源 1  
    private static Object resource2 = new Object(); // 资源 2
```

```
public static void main(String[] args) {
    new Thread(() -> {
        synchronized (resource1) {
            System.out.println(Thread.currentThread() + "get
resource1");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread() + "waiting get
resource2");
            synchronized (resource2) {
                System.out.println(Thread.currentThread() + "get
resource2");
            }
        }
    }, "线程 1").start();

    new Thread(() -> {
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread() + "waiting get
resource1");
            synchronized (resource1) {
                System.out.println(Thread.currentThread() + "get
resource1");
            }
        }
    }, "线程 2").start();
}
}
```

Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1
```

线程 A 通过 synchronized (resource1) 获得 resource1 的监视器锁，然后通过 Thread.sleep(1000); 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

学过操作系统的的朋友都知道产生死锁必须具备以下四个条件：

1. 互斥条件：该资源任意一个时刻只由一个线程占用。
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

如何避免线程死锁？

我上面说了产生死锁的四个必要条件，为了避免死锁，我们只要破坏产生死锁的四个条件中的其中一个就可以了。现在我们来挨个分析一下：

1. 破坏互斥条件：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
2. 破坏请求与保持条件：一次性申请所有的资源。
3. 破坏不剥夺条件：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
4. 破坏循环等待条件：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
new Thread(() -> {
    synchronized (resource1) {
        System.out.println(Thread.currentThread() + "get
resource1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource2");
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
        }
    }
})
```

```
}, "线程 2").start();
```

Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 1,5,main]waiting get resource2
Thread[线程 1,5,main]get resource2
Thread[线程 2,5,main]get resource1
Thread[线程 2,5,main]waiting get resource2
Thread[线程 2,5,main]get resource2
```

```
Process finished with exit code 0
```

我们分析一下上面的代码为什么避免了死锁的发生？

线程 1 首先获得到 resource1 的监视器锁，这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

2.3.9. 说说 **sleep()** 方法和 **wait()** 方法区别和共同点？

- 两者最主要的区别在于：sleep 方法没有释放锁，而 wait 方法释放了锁。
- 两者都可以暂停线程的执行。
- Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
- wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用 wait(long timeout) 超时后线程会自动苏醒。

2.3.10. 为什么我们调用 **start()** 方法时会执行 **run()** 方法，为什么我们不能直接调用 **run()** 方法？

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来！

new 一个 Thread，线程进入了新建状态；调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

总结：调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

2.3.11 synchronized 关键字

1. 说一说自己对于 **synchronized** 关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

2. 说自己是怎么使用 synchronized 关键字，在项目中用到了吗

3. synchronized 关键字最主要的三种使用方式

- **修饰实例方法：**作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
- **修饰静态方法：**也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管 new 了多少个对象，只有一份）。所以如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
- **修饰代码块：**指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

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。尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓存功能！

下面我以一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

双重校验锁实现对象单例（线程安全）

```
public class Singleton {  
  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {  
    }  
  
    public synchronized static Singleton getInstance() {  
        // 先判断对象是否已经实例过，没有实例化过才进入加锁代码  
        if (uniqueInstance == null) {  
            // 类对象加锁  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

```
    }  
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1. 为 uniqueInstance 分配内存空间
2. 初始化 uniqueInstance
3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1→3→2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

4.讲一下 synchronized 关键字的底层原理

synchronized 关键字底层原理属于 JVM 层面。

① synchronized 同步语句块的情况

```
public class SynchronizedDemo {  
    public void method() {  
        synchronized (this) {  
            System.out.println("synchronized 代码块");  
        }  
    }  
}
```

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 `javac SynchronizedDemo.java` 命令生成编译后的 .class 文件，然后执行 `javap -c -s -v -l SynchronizedDemo.class`。

```

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"

```

从上面我们可以看出：

synchronized 同步语句块的实现使用的是 **monitorenter** 和 **monitorexit** 指令，其中 **monitorenter** 指令指向同步代码块的开始位置，**monitorexit** 指令则指明同步代码块的结束位置。当执行 **monitorenter** 指令时，线程试图获取锁也就是获取 **monitor**(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 **monitorexit** 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

② synchronized 修饰方法的情况

```

public class SynchronizedDemo2 {
    public synchronized void method() {
        System.out.println("synchronized 方法");
    }
}

```

```

public test.SynchronizedDemo2();
descriptor: ()V
flags: ACC_PUBLIC
Code:
  stack=1, locals=1, args_size=1
  0: aload_0
  1: invokespecial #1           // Method java/lang/Object."<init>":()V
  4: return
LineNumberTable:
  line 3: 0

public synchronized void method();
descriptor: ()V
flags: ACC_PUBLIC, ACC_SYNCHRONIZED
Code:
  stack=2, locals=1, args_size=1
  0: getstatic    #2           // Field java/lang/System.out:Ljava/io/PrintStream;
  3: ldc          #3           // String synchronized 鑄規碼
  5: invokevirtual #4           // Method java/io/PrintStream.println:(Ljava/lang/String;)V
  8: return
LineNumberTable:
  line 5: 0
  line 6: 8
}
SourceFile: "SynchronizedDemo2.java"

```

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

5. 说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

关于这几种优化的详细信息可以查看笔主的这篇文章：<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/Multithread/synchronized.md>

6. 谈谈 synchronized 和 ReentrantLock 的区别

① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

② synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。

ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReentrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReentrantLock增加了一些高级功能。主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

- ReentrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实

现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

- ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。ReentrantLock默认情况是非公平的，可以通过 ReentrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。
- synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition()方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。在使用notify()/notifyAll()方法进行通知时，被通知的线程是由JVM选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法只会唤醒注册在该Condition实例中的所有等待线程。

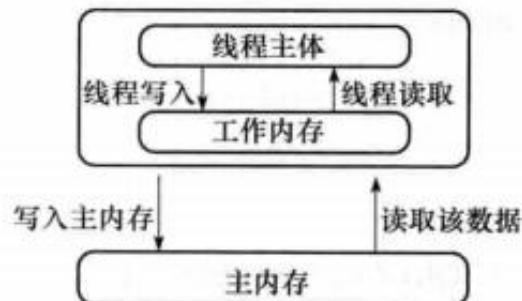
如果你想使用上述功能，那么选择ReentrantLock是一个不错的选择。

④ 性能已不是选择标准

2.3.12 volatile关键字

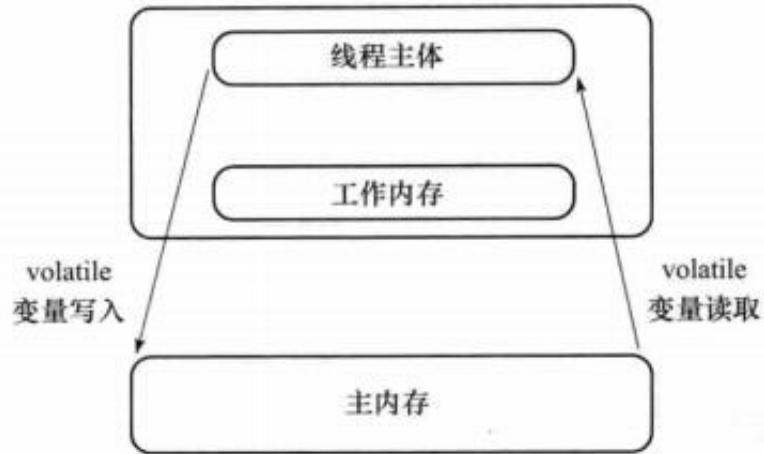
1. 讲一下Java内存模型

在JDK1.2之前，Java的内存模型实现总是从主存（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的Java内存模型下，线程可以把变量保存本地内存（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致。



要解决这个问题，就需要把变量声明为volatile，这就指示JVM，这个变量是不稳定的，每次使用它都要到主存中进行读取。

说白了，volatile关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。



2 并发编程的三个重要特性

1. **原子性**：一个的操作或者多次操作，要么所有的操作全部都得到执行并且不会收到任何因素的干扰而中断，要么所有的操作都执行，要么都不执行。`synchronized` 可以保证代码片段的原子性。
2. **可见性**：当一个变量对共享变量进行了修改，那么另外的线程都是立即可以看到修改后的最新值。`volatile` 关键字可以保证共享变量的可见性。
3. **有序性**：代码在执行的过程中的先后顺序，Java 在编译器以及运行期间的优化，代码的执行顺序未必就是编写代码时候的顺序。`volatile` 关键字可以禁止指令进行重排序优化。

3. 说说 `synchronized` 关键字和 `volatile` 关键字的区别

`synchronized`关键字和`volatile`关键字比较

- `volatile`关键字是线程同步的轻量级实现，所以`volatile`性能肯定比`synchronized`关键字要好。但是`volatile`关键字只能用于变量而`synchronized`关键字可以修饰方法以及代码块。
`synchronized`关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用`synchronized`关键字的场景还是更多一些。
- 多线程访问`volatile`关键字不会发生阻塞，而`synchronized`关键字可能会发生阻塞
- `volatile`关键字能保证数据的可见性，但不能保证数据的原子性。`synchronized`关键字两者都能保证。
- `volatile`关键字主要用于解决变量在多个线程之间的可见性，而 `synchronized`关键字解决的是多个线程之间访问资源的同步性。

2.3.13 ThreadLocal

1. ThreadLocal简介

通常情况下，我们创建的变量是可以被任何一个线程访问并修改的。如果想实现每一个线程都有自己的专属本地变量该如何解决呢？JDK中提供的`ThreadLocal`类正是为了解决这样的问题。

`ThreadLocal`类主要解决的就是让每个线程绑定自己的值，可以将`ThreadLocal`类形象的比喻成存放数据的盒子，盒子中可以存储每个线程的私有数据。

如果你创建了一个`ThreadLocal`变量，那么访问这个变量的每个线程都会有这个变量的本地副本，这也是`ThreadLocal`变量名的由来。他们可以使用`get ()` 和 `set ()` 方法来获取默认值或将其值更改为当前线程所存的副本的值，从而避免了线程安全问题。

再举个简单的例子：

比如有两个人去宝屋收集宝物，这两个共用一个袋子的话肯定会产生争执，但是给他们两个人分配一个袋子的话就不会出现这样的问题。如果把这两个人比作线程的话，那么ThreadLocal就是用来避免这两个线程竞争的。

2. ThreadLocal示例

相信看了上面的解释，大家已经搞懂 ThreadLocal 类是个什么东西了。

```
import java.text.SimpleDateFormat;
import java.util.Random;

public class ThreadLocalExample implements Runnable{

    // SimpleDateFormat 不是线程安全的，所以每个线程都要有自己独立的副本
    private static final ThreadLocal<SimpleDateFormat> formatter =
    ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyyMMdd HHmm"));

    public static void main(String[] args) throws InterruptedException {
        ThreadLocalExample obj = new ThreadLocalExample();
        for(int i=0 ; i<10; i++){
            Thread t = new Thread(obj, ""+i);
            Thread.sleep(new Random().nextInt(1000));
            t.start();
        }
    }

    @Override
    public void run() {
        System.out.println("Thread Name="
        "+Thread.currentThread().getName()+" default Formatter =
        "+formatter.get().toPattern());
        try {
            Thread.sleep(new Random().nextInt(1000));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        //formatter pattern is changed here by thread, but it won't
        reflect to other threads
        formatter.set(new SimpleDateFormat());

        System.out.println("Thread Name="
        "+Thread.currentThread().getName()+" formatter =
        "+formatter.get().toPattern());
    }
}
```

```
    }  
  
}
```

Output:

```
Thread Name= 0 default Formatter = yyyyMMdd HHmm  
Thread Name= 0 formatter = yy-M-d ah:mm  
Thread Name= 1 default Formatter = yyyyMMdd HHmm  
Thread Name= 2 default Formatter = yyyyMMdd HHmm  
Thread Name= 1 formatter = yy-M-d ah:mm  
Thread Name= 3 default Formatter = yyyyMMdd HHmm  
Thread Name= 2 formatter = yy-M-d ah:mm  
Thread Name= 4 default Formatter = yyyyMMdd HHmm  
Thread Name= 3 formatter = yy-M-d ah:mm  
Thread Name= 4 formatter = yy-M-d ah:mm  
Thread Name= 5 default Formatter = yyyyMMdd HHmm  
Thread Name= 5 formatter = yy-M-d ah:mm  
Thread Name= 6 default Formatter = yyyyMMdd HHmm  
Thread Name= 6 formatter = yy-M-d ah:mm  
Thread Name= 7 default Formatter = yyyyMMdd HHmm  
Thread Name= 7 formatter = yy-M-d ah:mm  
Thread Name= 8 default Formatter = yyyyMMdd HHmm  
Thread Name= 9 default Formatter = yyyyMMdd HHmm  
Thread Name= 8 formatter = yy-M-d ah:mm  
Thread Name= 9 formatter = yy-M-d ah:mm
```

从输出中可以看出，Thread-0已经改变了formatter的值，但仍然是thread-2默认格式化程序与初始化值相同，其他线程也一样。

上面有一段代码用到了创建 `ThreadLocal` 变量的那段代码用到了 Java8 的知识，它等于下面这段代码，如果你写了下面这段代码的话，IDEA会提示你转换为Java8的格式(IDEA真的不错！)。因为 `ThreadLocal`类在Java 8中扩展，使用一个新的方法 `withInitial()`，将Supplier功能接口作为参数。

```
private static final ThreadLocal<SimpleDateFormat> formatter = new  
ThreadLocal<SimpleDateFormat>(){  
    @Override  
    protected SimpleDateFormat initialValue()  
    {  
        return new SimpleDateFormat("yyyyMMdd HHmm");  
    }  
};
```

3. ThreadLocal原理

从 `Thread` 类源代码入手。

```
public class Thread implements Runnable {  
    ....  
    //与此线程有关的ThreadLocal值。由ThreadLocal类维护  
    ThreadLocal.ThreadLocalMap threadLocals = null;  
  
    //与此线程有关的InheritableThreadLocal值。由InheritableThreadLocal类维护  
    ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;  
    ....  
}
```

从上面 `Thread` 类 源代码可以看出 `Thread` 类中有一个 `threadLocals` 和一个 `inheritableThreadLocals` 变量，它们都是 `ThreadLocalMap` 类型的变量，我们可以把 `ThreadLocalMap` 理解为 `ThreadLocal` 类实现的定制化的 `HashMap`。默认情况下这两个变量都是 `null`，只有当前线程调用 `ThreadLocal` 类的 `set` 或 `get` 方法时才创建它们，实际上调用这两个方法的时候，我们调用的是 `ThreadLocalMap` 类对应的 `get()`、`set()` 方法。

`ThreadLocal` 类的 `set()` 方法

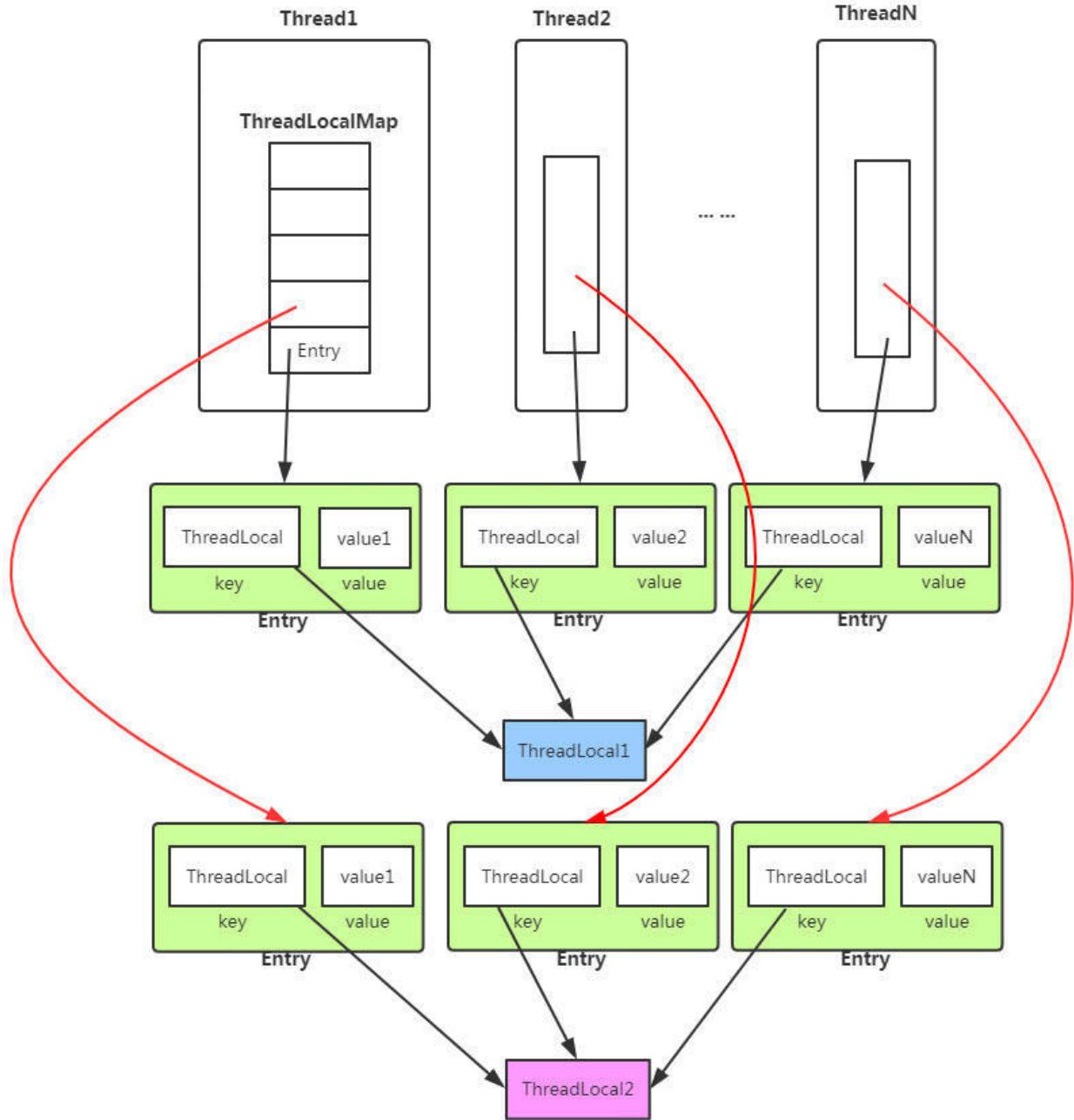
```
public void set(T value) {  
    Thread t = Thread.currentThread();  
    ThreadLocalMap map = getMap(t);  
    if (map != null)  
        map.set(this, value);  
    else  
        createMap(t, value);  
}  
ThreadLocalMap getMap(Thread t) {  
    return t.threadLocals;  
}
```

通过上面这些内容，我们足以通过猜测得出结论：最终的变量是放在了当前线程的 `ThreadLocalMap` 中，并不是存在 `ThreadLocal` 上，`ThreadLocal` 可以理解为只是 `ThreadLocalMap` 的封装，传递了变量值。`ThrealLocal` 类中可以通过 `Thread.currentThread()` 获取到当前线程对象后，直接通过 `getMap(Thread t)` 可以访问到该线程的 `ThreadLocalMap` 对象。

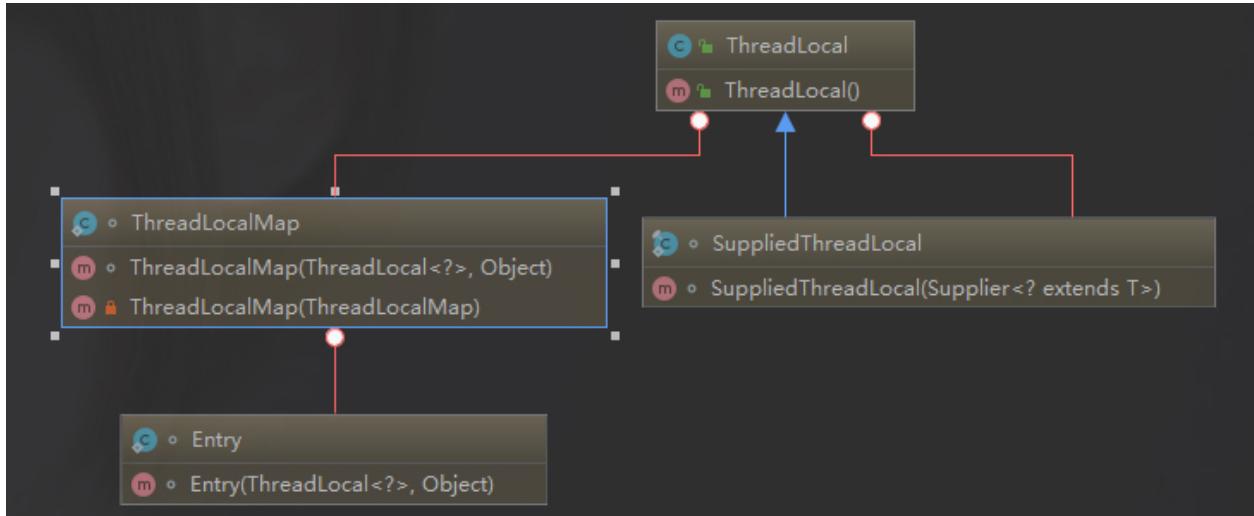
`ThreadLocal` 内部维护的是一个类似 `Map` 的 `ThreadLocalMap` 数据结构，`key` 为当前对象的 `Thread` 对象，值为 `Object` 对象。

```
ThreadLocalMap(ThreadLocal<?> firstKey, Object firstValue) {  
    ....  
}
```

比如我们在同一个线程中声明了两个 `ThreadLocal` 对象的话，会使用 `Thread` 内部都是使用仅有那个 `ThreadLocalMap` 存放数据的，`ThreadLocalMap` 的 key 就是 `ThreadLocal` 对象，value 就是 `ThreadLocal` 对象调用 `set` 方法设置的值。



`ThreadLocalMap` 是 `ThreadLocal` 的静态内部类。



4. ThreadLocal 内存泄露问题

`ThreadLocalMap` 中使用的 key 为 `ThreadLocal` 的弱引用,而 value 是强引用。所以,如果 `ThreadLocal` 没有被外部强引用的情况下,在垃圾回收的时候, key 会被清理掉,而 value 不会被清理掉。这样一来, `ThreadLocalMap` 中就会出现key为null的Entry。假如我们不做任何措施的话,value 永远无法被GC 回收,这个时候就可能会产生内存泄露。`ThreadLocalMap`实现中已经考虑了这种情况,在调用 `set()`、`get()`、`remove()` 方法的时候,会清理掉 key 为 null 的记录。使用完 `ThreadLocal`方法后 最好手动调用 `remove()` 方法

```

static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;

    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}

```

弱引用介绍:

如果一个对象只具有弱引用,那就类似于可有可无的生活用品。弱引用与软引用的区别在于:只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它 所管辖的内存区域的过程中,一旦发现了只具有弱引用的对象,不管当前内存空间足够与否,都会回收它的内存。不过,由于垃圾回收器是一个优先级很低的线程, 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列 (ReferenceQueue) 联合使用, 如果弱引用所引用的对象被垃圾回收, Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

2.3.14 线程池

1. 为什么要用线程池?

池化技术相比大家已经屡见不鲜了, 线程池、数据库连接池、Http 连接池等等都是对这个思想的应用。池化技术的思想主要是为了减少每次获取资源的消耗, 提高对资源的利用率。

线程池提供了一种限制和管理资源（包括执行一个任务）。每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java 并发编程的艺术》提到的来说一下使用线程池的好处：

- **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2. 实现Runnable接口和Callable接口的区别

Runnable 自 Java 1.0 以来一直存在，但 Callable 仅在 Java 1.5 中引入，目的就是为了来处理 Runnable 不支持的用例。Runnable 接口不会返回结果或抛出检查异常，但是 Callable 接口可以。所以，如果任务不需要返回结果或抛出异常推荐使用 Runnable 接口，这样代码看起来会更加简洁。

工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。

```
(Executors.callable (Runnable task) 或 Executors.callable (Runnable task,  
Object resule) )。
```

Runnable.java

```
@FunctionalInterface  
public interface Runnable {  
    /**  
     * 被线程执行，没有返回值也无法抛出异常  
     */  
    public abstract void run();  
}
```

Callable.java

```
@FunctionalInterface  
public interface Callable<V> {  
    /**  
     * 计算结果，或在无法这样做时抛出异常。  
     * @return 计算得出的结果  
     * @throws 如果无法计算结果，则抛出异常  
     */  
    V call() throws Exception;  
}
```

3. 执行execute()方法和submit()方法的区别是什么呢？

1. execute() 方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；
2. submit() 方法用于提交需要返回值的任务。线程池会返回一个 Future 类型的对象，通过这

一个 `Future` 对象可以判断任务是否执行成功，并且可以通过 `Future` 的 `get()` 方法来获取返回值，`get()` 方法会阻塞当前线程直到任务完成，而使用 `get (long timeout, TimeUnit unit)` 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

我们以 `AbstractExecutorService` 接口中的一个 `submit` 方法为例子来看看源代码：

```
public Future<?> submit(Runnable task) {  
    if (task == null) throw new NullPointerException();  
    RunnableFuture<Void> ftask = newTaskFor(task, null);  
    execute(ftask);  
    return ftask;  
}
```

上面方法调用的 `newTaskFor` 方法返回了一个 `FutureTask` 对象。

```
protected <T> RunnableFuture<T> newTaskFor(Runnable runnable, T  
value) {  
    return new FutureTask<T>(runnable, value);  
}
```

我们再来看看 `execute()` 方法：

```
public void execute(Runnable command) {  
    ...  
}
```

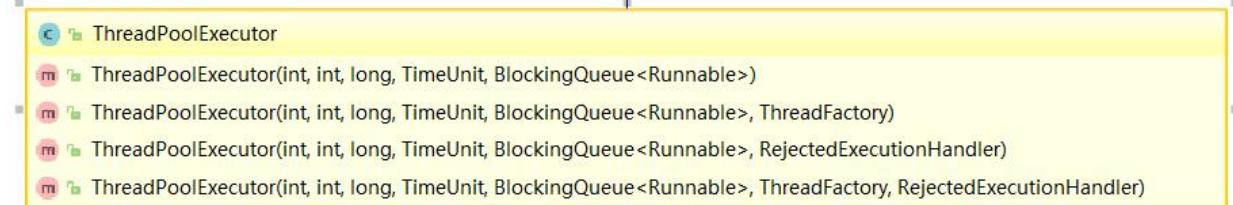
4. 如何创建线程池

《阿里巴巴Java开发手册》中强制线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

`Executors` 返回线程池对象的弊端如下：

- `FixedThreadPool` 和 `SingleThreadExecutor`： 允许请求的队列长度为 `Integer.MAX_VALUE`，可能堆积大量的请求，从而导致OOM。
- `CachedThreadPool` 和 `ScheduledThreadPool`： 允许创建的线程数量为 `Integer.MAX_VALUE`，可能会创建大量线程，从而导致OOM。

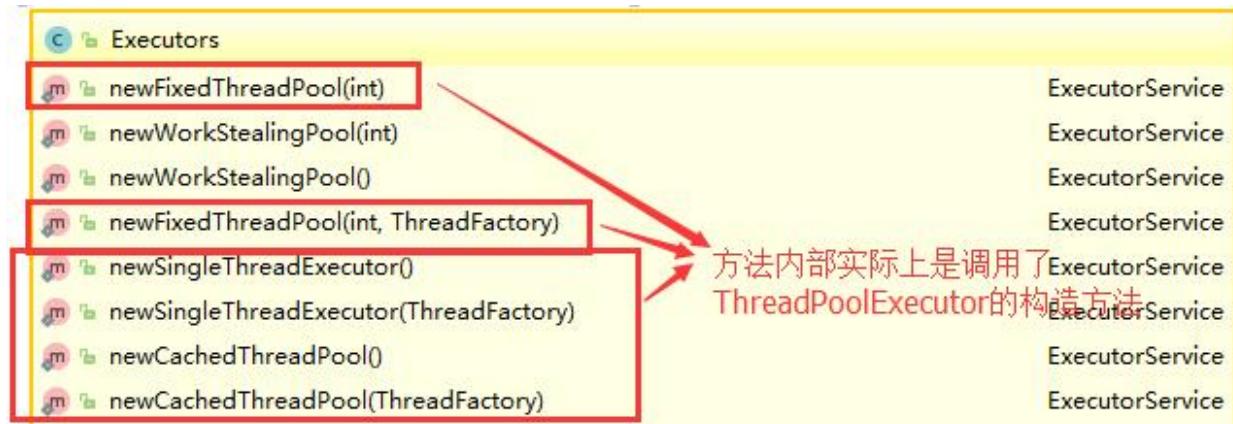
方式一：通过构造方法实现



方式二：通过Executor 框架的工具类Executors来实现 我们可以创建三种类型的 ThreadPoolExecutor：

- **FixedThreadPool**：该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
- **SingleThreadExecutor**：方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
- **CachedThreadPool**：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

对应Executors工具类中的方法如图所示：



5. ThreadPoolExecutor 类分析

ThreadPoolExecutor 类中提供的四个构造方法。我们来看最长的那个，其余三个都是在这个构造方法的基础上产生（其他几个构造方法说白点都是给定某些默认参数的构造方法比如默认制定拒绝策略是什么），这里就不贴代码讲了，比较简单。

```
/*
 * 用给定的初始参数创建一个新的ThreadPoolExecutor。
 */
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        keepAliveTime < 0 ||
        unit == null ||
        workQueue == null ||
        threadFactory == null ||
        handler == null)
        throw new IllegalArgumentException("参数错误");
    ...
}
```

```

        maximumPoolSize < corePoolSize ||  

        keepAliveTime < 0)  

        throw new IllegalArgumentException();  

    if (workQueue == null || threadFactory == null || handler ==  

        null)  

        throw new NullPointerException();  

    this.corePoolSize = corePoolSize;  

    this.maximumPoolSize = maximumPoolSize;  

    this.workQueue = workQueue;  

    this.keepAliveTime = unit.toNanos(keepAliveTime);  

    this.threadFactory = threadFactory;  

    this.handler = handler;  

}

```

下面这些对创建 非常重要，在后面使用线程池的过程中你一定会用到！所以，务必拿着小本本记清楚。

ThreadPoolExecutor 构造函数重要参数分析

ThreadPoolExecutor 3 个最重要的参数：

- **corePoolSize**：核心线程数线程数定义了最小可以同时运行的线程数量。
- **maximumPoolSize**：当队列中存放的任务达到队列容量的时候，当前可以同时运行的线程数量变为最大线程数。
- **workQueue**：当新任务来的时候会先判断当前运行的线程数量是否达到核心线程数，如果达到的话，新任务就会被存放在队列中。

ThreadPoolExecutor 其他常见参数：

1. **keepAliveTime**：当线程池中的线程数量大于 **corePoolSize** 的时候，如果这时没有新的任务提交，核心线程外的线程不会立即销毁，而是会等待，直到等待的时间超过了 **keepAliveTime** 才会被回收销毁；
2. **unit**：**keepAliveTime** 参数的时间单位。
3. **threadFactory**：executor 创建新线程的时候会用到。
4. **handler**：饱和策略。关于饱和策略下面单独介绍一下。

ThreadPoolExecutor 饱和策略

ThreadPoolExecutor 饱和策略定义：

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任务，**ThreadPoolTaskExecutor** 定义一些策略：

- **ThreadPoolExecutor.AbortPolicy**：抛出 **RejectedExecutionException** 来拒绝新任务的处理。
- **ThreadPoolExecutor.CallerRunsPolicy**：调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。

- `ThreadPoolExecutor.DiscardPolicy`: 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`: 此策略将丢弃最早的未处理的任务请求。

举个例子：Spring 通过 `ThreadPoolTaskExecutor` 或者我们直接通过 `ThreadPoolExecutor` 的构造函数创建线程池的时候，当我们不指定 `RejectedExecutionHandler` 饱和策略的话来配置线程池的时候默认使用的是 `ThreadPoolExecutor.AbortPolicy`。在默认情况下，`ThreadPoolExecutor` 将抛出 `RejectedExecutionException` 来拒绝新来的任务，这代表你将丢失对这个任务的处理。对于可伸缩的应用程序，建议使用 `ThreadPoolExecutor.CallerRunsPolicy`。当最大池被填满时，此策略为我们提供可伸缩队列。（这个直接查看 `ThreadPoolExecutor` 的构造函数源码就可以看出，比较简单的原因，这里就不贴代码了）

6. 一个简单的线程池Demo: Runnable + ThreadPoolExecutor

为了让大家更清楚上面的面试题中的一些概念，我写了一个简单的线程池 Demo。

首先创建一个 `Runnable` 接口的实现类（当然也可以是 `Callable` 接口，我们上面也说了两者的区别。）

MyRunnable.java

```
import java.util.Date;

/**
 * 这是一个简单的Runnable类，需要大约5秒钟来执行其任务。
 * @author shuang.kou
 */
public class MyRunnable implements Runnable {

    private String command;

    public MyRunnable(String s) {
        this.command = s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " Start.
Time = " + new Date());
        processCommand();
        System.out.println(Thread.currentThread().getName() + " End.
Time = " + new Date());
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        }
    }
}
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

@Override
public String toString() {
    return this.command;
}
}
```

编写测试程序，我们这里以阿里巴巴推荐的使用 `ThreadPoolExecutor` 构造函数自定义参数的方式来创建线程池。

`ThreadPoolExecutorDemo.java`

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExecutorDemo {

    private static final int CORE_POOL_SIZE = 5;
    private static final int MAX_POOL_SIZE = 10;
    private static final int QUEUE_CAPACITY = 100;
    private static final Long KEEP_ALIVE_TIME = 1L;
    public static void main(String[] args) {

        //使用阿里巴巴推荐的创建线程池的方式
        //通过ThreadPoolExecutor构造函数自定义参数创建
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            KEEP_ALIVE_TIME,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(QUEUE_CAPACITY),
            new ThreadPoolExecutor.CallerRunsPolicy());

        for (int i = 0; i < 10; i++) {
            //创建WorkerThread对象（WorkerThread类实现了Runnable 接口）
            Runnable worker = new MyRunnable(" " + i);
            //执行Runnable
            executor.execute(worker);
        }
    }
}
```

```
//终止线程池
executor.shutdown();
while (!executor.isTerminated()) {
}
System.out.println("Finished all threads");
}
```

可以看到我们上面的代码指定了：

1. `corePoolSize`: 核心线程数为 5。
2. `maximumPoolSize` : 最大线程数 10
3. `keepAliveTime` : 等待时间为 1L。
4. `unit`: 等待时间的单位为 `TimeUnit.SECONDS`。
5. `workQueue`: 任务队列为 `ArrayBlockingQueue`, 并且容量为 100;
6. `handler`: 饱和策略为 `CallerRunsPolicy`。

Output:

```
pool-1-thread-2 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-5 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-4 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-1 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-3 Start. Time = Tue Nov 12 20:59:44 CST 2019
pool-1-thread-5 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-3 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-4 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-1 End. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-1 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-4 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-3 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-5 Start. Time = Tue Nov 12 20:59:49 CST 2019
pool-1-thread-2 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-3 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-4 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-5 End. Time = Tue Nov 12 20:59:54 CST 2019
pool-1-thread-1 End. Time = Tue Nov 12 20:59:54 CST 2019
```

7. 线程池原理分析

承接 4.6 节，我们通过代码输出结果可以看出：线程池每次会同时执行 5 个任务，这 5 个任务执行完之后，剩余的 5 个任务才会被执行。大家可以先通过上面讲解的内容，分析一下到底是咋回事？（自己独立思考一会）

现在，我们就分析上面的输出内容来简单分析一下线程池原理。

为了搞懂线程池的原理，我们需要首先分析一下 `execute` 方法。在 4.6 节中的 Demo 中我们使用 `executor.execute(worker)` 来提交一个任务到线程池中去，这个方法非常重要，下面我们来看看它的源码：

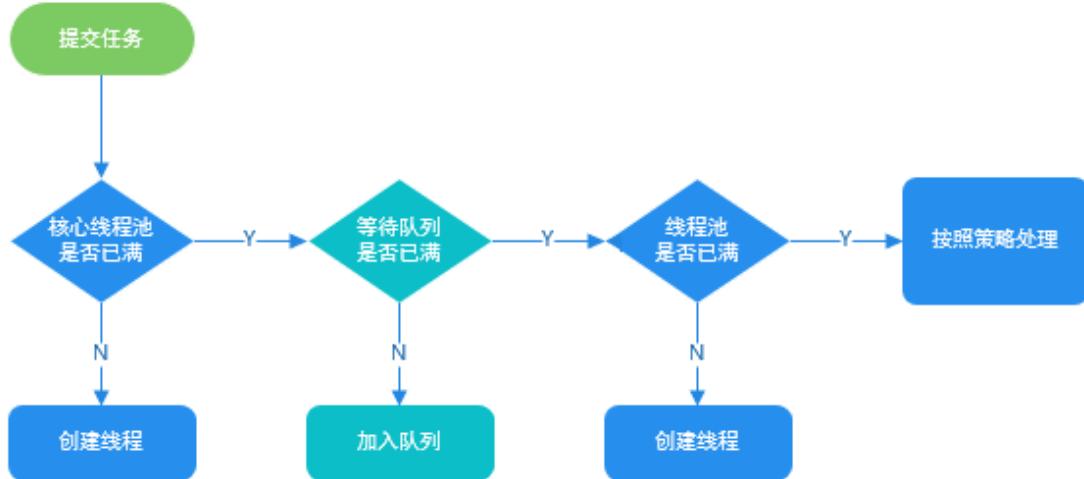
```
// 存放线程池的运行状态 (runState) 和线程池内有效线程的数量  
(workerCount)  
private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING,  
0));  
  
private static int workerCountOf(int c) {  
    return c & CAPACITY;  
}  
  
private final BlockingQueue<Runnable> workQueue;  
  
public void execute(Runnable command) {  
    // 如果任务为null，则抛出异常。  
    if (command == null)  
        throw new NullPointerException();  
    // ctl 中保存的线程池当前的一些状态信息  
    int c = ctl.get();  
  
    // 下面会涉及到 3 步 操作  
    // 1.首先判断当前线程池中之行的任务数量是否小于 corePoolSize  
    // 如果小于的话，通过addWorker(command, true)新建一个线程，并将  
    // 任务(command)添加到该线程中；然后，启动该线程从而执行任务。  
    if (workerCountOf(c) < corePoolSize) {  
        if (addWorker(command, true))  
            return;  
        c = ctl.get();  
    }  
    // 2.如果当前之行的任务数量大于等于 corePoolSize 的时候就会走到  
    // 这里  
    // 通过 isRunning 方法判断线程池状态，线程池处于 RUNNING 状态才  
    // 会被并且队列可以加入任务，该任务才会被加入进去  
    if (isRunning(c) && workQueue.offer(command)) {  
        int recheck = ctl.get();  
        // 再次获取线程池状态，如果线程池状态不是 RUNNING 状态就需要  
        // 从任务队列中移除任务，并尝试判断线程是否全部执行完毕。同时执行拒绝策略。  
        if (!isRunning(recheck) && remove(command))
```

```

        reject(command);
        // 如果当前线程池为空就新创建一个线程并执行。
    else if (workerCountOf(recheck) == 0)
        addWorker(null, false);
}
//3. 通过addWorker(command, false)新建一个线程，并将任务
//（command）添加到该线程中；然后，启动该线程从而执行任务。
//如果addWorker(command, false)执行失败，则通过reject()执行相应的
//拒绝策略的内容。
else if (!addWorker(command, false))
    reject(command);
}

```

通过下图可以更好的对上面这 3 步做一个展示，下图是我为了省事直接从网上找到，原地址不明。



现在，让我们回到 4.6 节我们写的 Demo，现在应该是很容易就可以搞懂它的原理了呢？

没搞懂的话，也没关系，可以看看我的分析：

我们在代码中模拟了 10 个任务，我们配置的核心线程数为 5、等待队列容量为 100，所以每次只可能存在 5 个任务同时执行，剩下的 5 个任务会被放到等待队列中去。当前的 5 个任务之行完成后，才会之行剩下的 5 个任务。

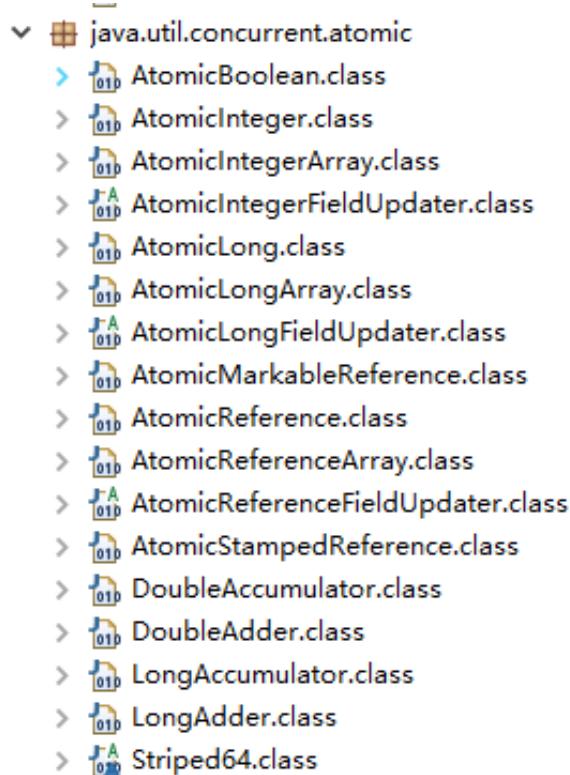
2.3.15 Atomic 原子类

1. 介绍一下Atomic 原子类

Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包 `java.util.concurrent` 的原子类都存放在 `java.util.concurrent.atomic` 下，如下图所示。



2. JUC 包中的原子类是哪4类？

基本类型

使用原子的方式更新基本类型

- AtomicInteger: 整形原子类
- AtomicLong: 长整形原子类
- AtomicBoolean: 布尔型原子类

数组类型

使用原子的方式更新数组里的某个元素

- AtomicIntegerArray: 整形数组原子类
- AtomicLongArray: 长整形数组原子类
- AtomicReferenceArray: 引用类型数组原子类

引用类型

- AtomicReference: 引用类型原子类
- AtomicStampedReference: 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference : 原子更新带有标记位的引用类型

对象的属性修改类型

- AtomicIntegerFieldUpdater: 原子更新整形字段的更新器
- AtomicLongFieldUpdater: 原子更新长整形字段的更新器
-

3. 讲讲 AtomicInteger 的使用

AtomicInteger 类常用方法

```
public final int get() //获取当前的值  
public final int getAndSet(int newValue)//获取当前的值，并设置新的值  
public final int getAndIncrement()//获取当前的值，并自增  
public final int getAndDecrement() //获取当前的值，并自减  
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值  
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期  
值，则以原子方式将该值设置为输入值 (update)  
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet  
设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

AtomicInteger 类的使用示例

使用 AtomicInteger 之后，不用对 increment() 方法加锁也可以保证线程安全。

```
class AtomicIntegerTest {  
    private AtomicInteger count = new AtomicInteger();  
    //使用AtomicInteger之后，不需要对该方法加锁，也可以实现线程安全。  
    public void increment() {  
        count.incrementAndGet();  
    }  
  
    public int getCount() {  
        return count.get();  
    }  
}
```

4. 能不能给我简单介绍一下 AtomicInteger 类的原理

AtomicInteger 线程安全原理简单分析

AtomicInteger 类的部分源码：

```

// setup to use Unsafe.compareAndSwapInt for updates (更新操作时提供“比较并替换”的作用)
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;

```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

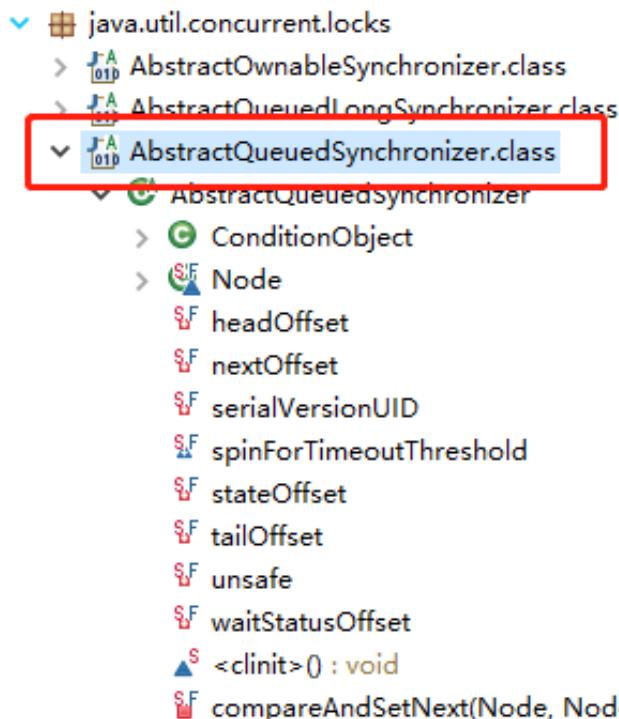
CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

关于 Atomic 原子类这部分更多内容可以查看我的这篇文章：[并发编程面试必备：JUC 中的 Atomic 原子类总结](#)

2.3.16 AQS

1. AQS 介绍

AQS的全称为 (AbstractQueuedSynchronizer) ，这个类在java.util.concurrent.locks包下面。



AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的ReentrantLock, Semaphore, 其他的诸如ReentrantReadWriteLock, SynchronousQueue, FutureTask等等皆是基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

2. AQS 原理分析

AQS 原理这部分参考了部分博客，在5.2节末尾放了链接。

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于AQS原理的理解”。下面给大家一个示例供大家参加，面试不是背题，大家一定要加入自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

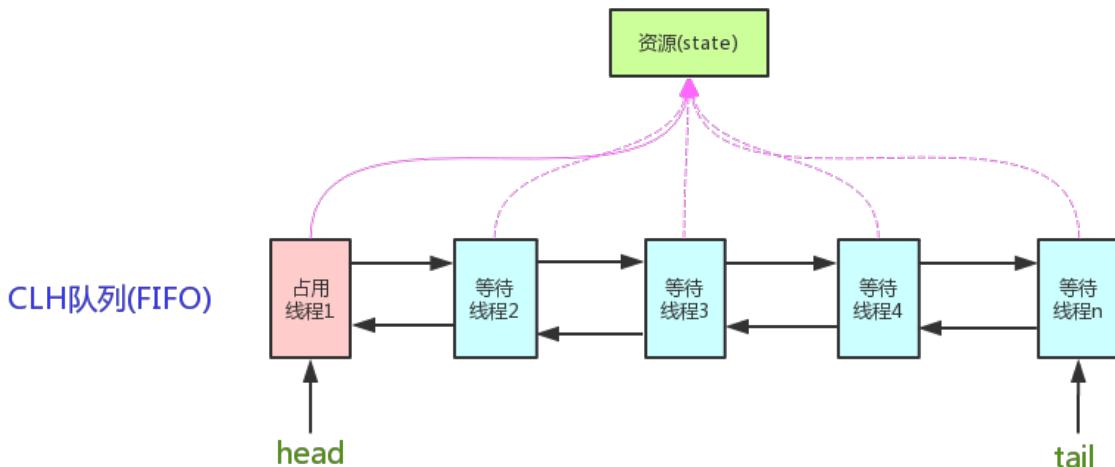
下面大部分内容其实在AQS类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

AQS 原理概览

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin, and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

看个AQS(AbstractQueuedSynchronizer)原理图：



AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

```
private volatile int state; //共享变量，使用volatile修饰保证线程可见性
```

状态信息通过protected类型的getState, setState, compareAndSetState进行操作

```

// 返回同步状态的当前值
protected final int getState() {
    return state;
}

// 设置同步状态的值
protected final void setState(int newState) {
    state = newState;
}

// 原子地 (CAS操作) 将同步状态值设置为给定值update如果当前同步状态的值等于expect (期望值)
protected final boolean compareAndSetState(int expect, int update) {
    return unsafe.compareAndSwapInt(this, stateOffset, expect,
        update);
}

```

AQS 对资源的共享方式

AQS 定义两种资源共享方式

- **Exclusive** (独占) : 只有一个线程能执行, 如ReentrantLock。又可分为公平锁和非公平锁:
 - 公平锁: 按照线程在队列中的排队顺序, 先到者先拿到锁
 - 非公平锁: 当线程要获取锁时, 无视队列顺序直接去抢锁, 谁抢到就是谁的
- **Share** (共享) : 多个线程可同时执行, 如Semaphore/CountDownLatch。Semaphore、CountDownLatch、CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式, 因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可, 至于具体线程等待队列的维护 (如获取资源失败入队/唤醒出队等), AQS已经在顶层实现好了。

AQS 底层使用了模板方法模式

同步器的设计是基于模板方法模式的, 如果需要自定义同步器一般的方式是这样 (模板方法模式很经典的一个应用) :

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。 (这些重写方法很简单, 无非是对共享资源state的获取和释放)
2. 将AQS组合在自定义同步组件的实现中, 并调用其模板方法, 而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别, 这是模板方法模式很经典的一个运用。

AQS 使用了模板方法模式, 自定义同步器时需要重写下面几个AQS提供的模板方法:

`isHeldExclusively()`//该线程是否正在独占资源。只有用到condition才需要去实现它。

`tryAcquire(int)`//独占方式。尝试获取资源，成功则返回true，失败则返回false。

`tryRelease(int)`//独占方式。尝试释放资源，成功则返回true，失败则返回false。

`tryAcquireShared(int)`//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

`tryReleaseShared(int)`//共享方式。尝试释放资源，成功则返回true，失败则返回false。

默认情况下，每个方法都抛出 `UnsupportedOperationException`。这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多少次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后（即state=0），会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现`tryAcquire-tryRelease`、`tryAcquireShared-tryReleaseShared`中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如`ReentrantReadWriteLock`。

推荐两篇 AQS 原理和相关源码分析的文章：

- <http://www.cnblogs.com/waterystone/p/4920797.html>
- <https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>

3. AQS 组件总结

- **Semaphore(信号量)-允许多个线程同时访问：** synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。
- **CountDownLatch (倒计时器)：** CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。
- **CyclicBarrier(循环栅栏)：** CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用 (Cyclic) 的屏障 (Barrier)。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。
CyclicBarrier默认的构造方法是 `CyclicBarrier(int parties)`，其参数表示屏障拦截的线程数量，每个线程调用`await()`方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

Reference

- 《深入理解 Java 虚拟机》
- 《实战 Java 高并发程序设计》
- 《Java并发编程的艺术》
- <http://www.cnblogs.com/waterystone/p/4920797.html>
- <https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html>
- <https://www.journaldev.com/1076/java-threadlocal-example>

2.4 JVM

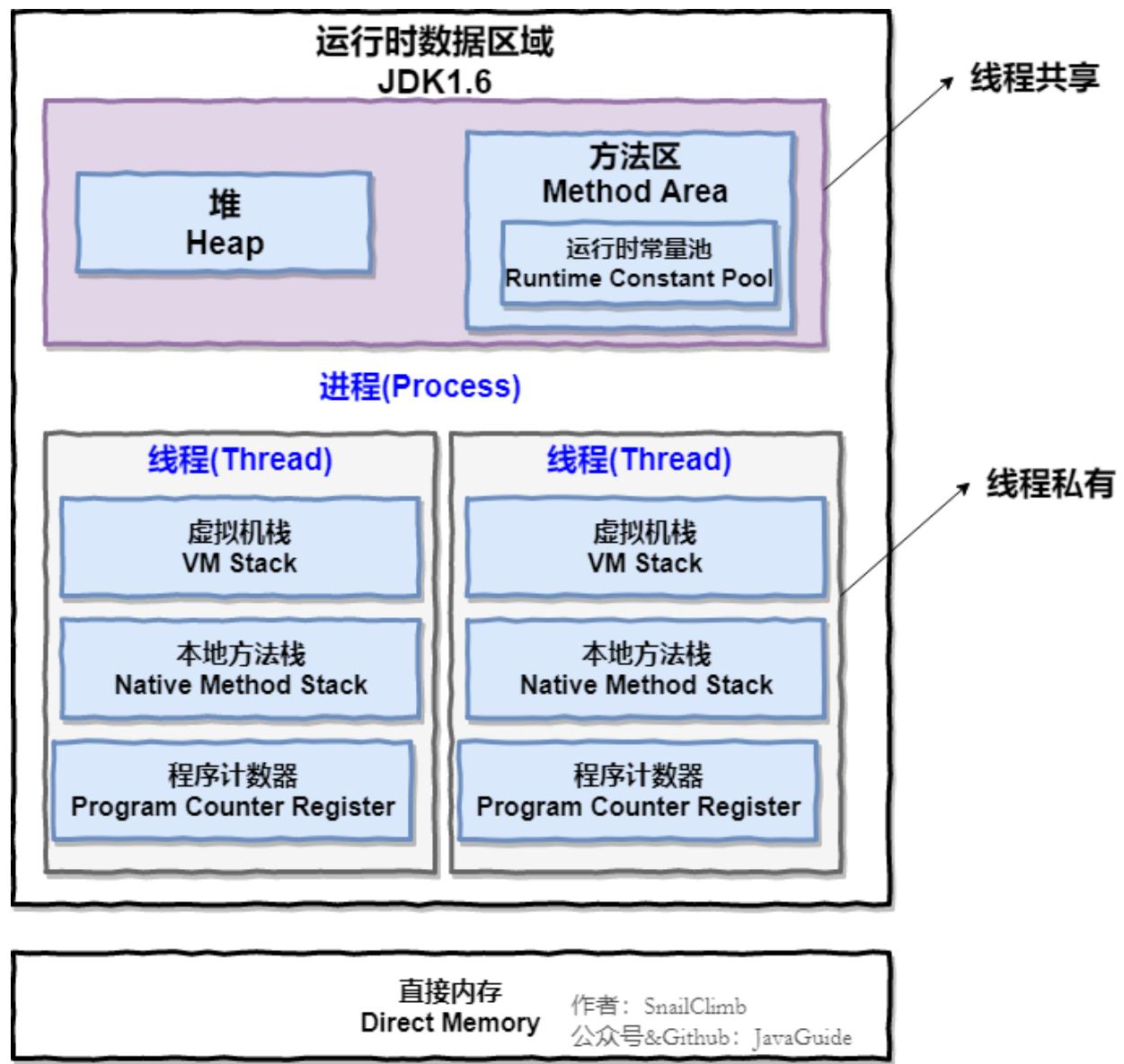
作者: Guide哥。

介绍: Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

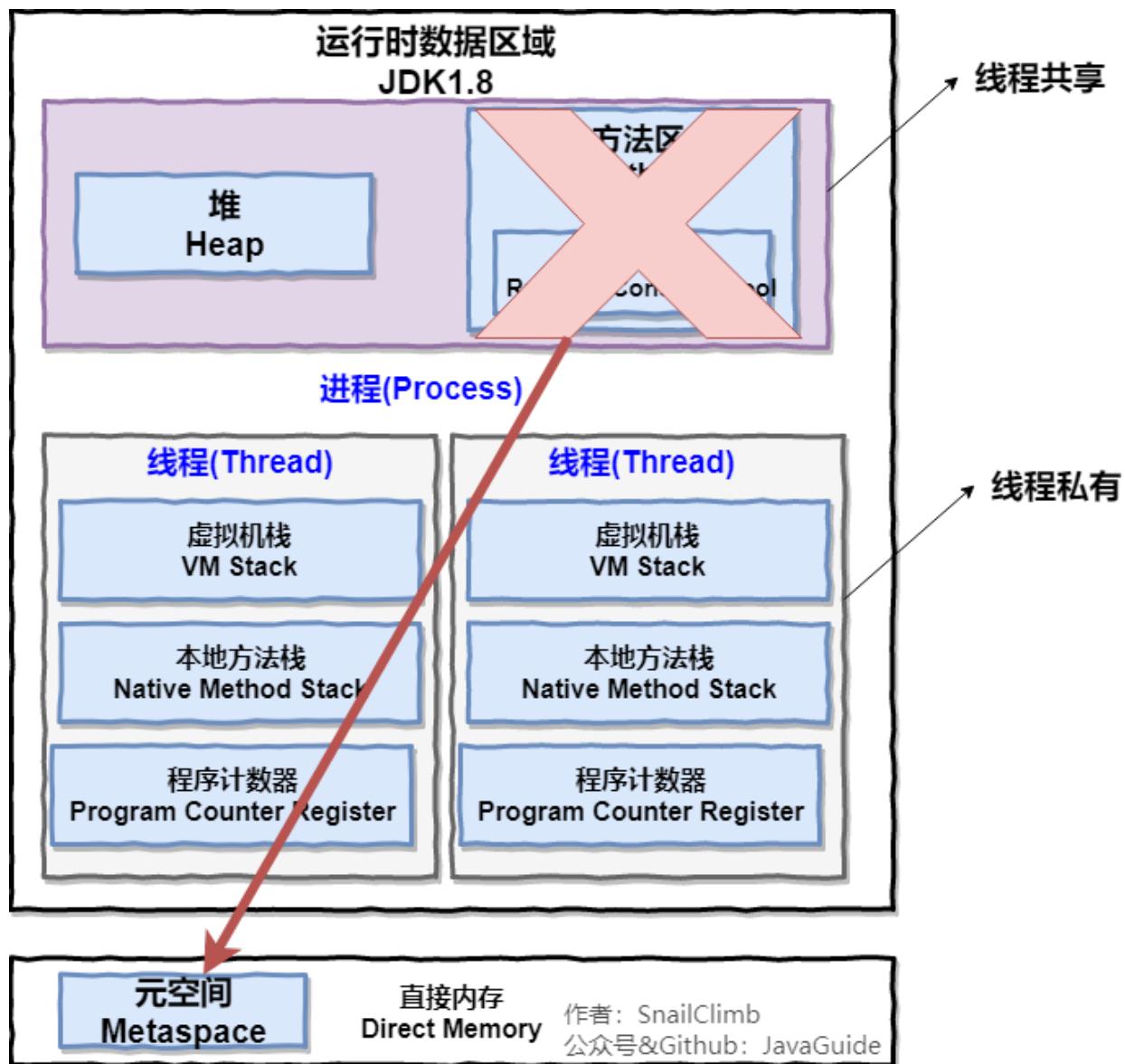
2.4.1 介绍下 Java 内存区域(运行时数据区)

Java 虚拟机在执行 Java 程序的过程中会把它管理的内存划分成若干个不同的数据区域。JDK. 1.8 和之前的版本略有不同, 下面会介绍到。

JDK 1.8之前:



JDK 1.8 :



线程私有的：

- 程序计数器
- 虚拟机栈
- 本地方法栈

线程共享的：

- 堆
- 方法区
- 直接内存(非运行时数据区的一部分)

程序计数器

程序计数器是一块较小的内存空间，可以看作是当前线程所执行的字节码的行号指示器。字节码解释器工作时通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等功能都需要依赖这个计数器来完成。

另外，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

从上面的介绍中我们知道程序计数器主要有两个作用：

1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

注意：程序计数器是唯一一个不会出现 `OutOfMemoryError` 的内存区域，它的生命周期随着线程的创建而创建，随着线程的结束而死亡。

Java 虚拟机栈

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期和线程相同，描述的是 Java 方法执行的内存模型，每次方法调用的数据都是通过栈传递的。

Java 内存可以粗略的区分为堆内存（Heap）和栈内存（Stack），其中栈就是现在说的虚拟机栈，或者说是虚拟机栈中局部变量表部分。（实际上，Java虚拟机栈是由一个个栈帧组成，而每个栈帧中都拥有：局部变量表、操作数栈、动态链接、方法出口信息。）

局部变量表主要存放了编译器可知的各种数据类型（`boolean`、`byte`、`char`、`short`、`int`、`float`、`long`、`double`）、对象引用（reference类型，它不同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关的位置）。

Java 虚拟机栈会出现两种异常：`StackOverflowError` 和 `OutOfMemoryError`。

- **StackOverflowError**：若 Java 虚拟机栈的内存大小不允许动态扩展，那么当线程请求栈的深度超过当前 Java 虚拟机栈的最大深度的时候，就抛出 `StackOverflowError` 异常。
- **OutOfMemoryError**：若 Java 虚拟机栈的内存大小允许动态扩展，且当线程请求栈时内存用完了，无法再动态扩展了，此时抛出 `OutOfMemoryError` 异常。

Java 虚拟机栈也是线程私有的，每个线程都有各自的 Java 虚拟机栈，而且随着线程的创建而创建，随着线程的死亡而死亡。

扩展：那么方法/函数如何调用？

Java 栈可用类比数据结构中栈，Java 栈中保存的主要内容是栈帧，每一次函数调用都会有一个对应的栈帧被压入 Java 栈，每一个函数调用结束后，都会有一个栈帧被弹出。

Java 方法有两种返回方式：

1. `return` 语句。
2. 抛出异常。

不管哪种返回方式都会导致栈帧被弹出。

本地方法栈

和虚拟机栈所发挥的作用非常相似，区别是：虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

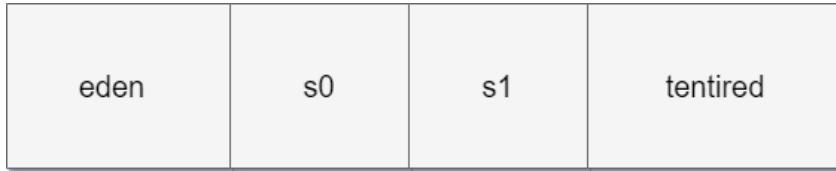
本地方法被执行的时候，在本地方法栈也会创建一个栈帧，用于存放该本地方法的局部变量表、操作数栈、动态链接、出口信息。

方法执行完毕后相应的栈帧也会出栈并释放内存空间，也会出现 `StackOverflowError` 和 `OutOfMemoryError` 两种异常。

堆

Java 虚拟机所管理的内存中最大的一块，Java 堆是所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

Java 堆是垃圾收集器管理的主要区域，因此也被称作**GC堆 (Garbage Collected Heap)** .从垃圾回收的角度，由于现在收集器基本都采用分代垃圾收集算法，所以Java堆还可以细分为：新生代和老年代：再细致一点有：Eden空间、From Survivor、To Survivor空间等。进一步划分的目的是更好地回收内存，或者更快地分配内存。



上图所示的 eden区、s0区、s1区都属于新生代，tentired 区属于老年代。大部分情况，对象都会首先在 Eden 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 s0 或者 s1，并且对象的年龄还会加 1(Eden区→Survivor 区后对象的初始年龄变为1)，当它的年龄增加到一定程度（默认为15岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

方法区

方法区与 Java 堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是它却有一个别名叫做 **Non-Heap (非堆)**，目的应该是与 Java 堆区分开来。

方法区也被称为永久代。很多人都会分不清方法区和永久代的关系，为此我也查阅了文献。

方法区和永久代的关系

《Java虚拟机规范》只是规定了有方法区这么个概念和它的作用，并没有规定如何去实现它。那么，在不同的 JVM 上方法区的实现肯定是不同的了。方法区和永久代的关系很像Java中接口和类的关系，类实现了接口，而永久代就是HotSpot虚拟机对虚拟机规范中方法区的一种实现方式。也就是说，永久代是HotSpot的概念，方法区是Java虚拟机规范中的定义，是一种规范，而永久代是一种实现，一个是标准一个是实现，其他的虚拟机实现并没有永久带这一说法。

常用参数

JDK 1.8 之前永久代还没被彻底移除的时候通常通过下面这些参数来调节方法区大小

```
-XX:PermSize=N //方法区(永久代)初始大小  
-XX:MaxPermSize=N //方法区(永久代)最大大小,超过这个值将会抛出  
OutOfMemoryError异常:java.lang.OutOfMemoryError: PermGen
```

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入方法区后就“永远存在”了。**

JDK 1.8 的时候，方法区（HotSpot的永久代）被彻底移除了（JDK1.7就已经开始了），取而代之是元空间，元空间使用的是直接内存。

下面是一些常用参数：

```
-XX:MetaspaceSize=N //设置Metaspace的初始 (和最小大小)  
-XX:MaxMetaspaceSize=N //设置Metaspace的最大大小
```

与永久代很大的不同就是，如果不指定大小的话，随着更多类的创建，虚拟机会耗尽所有可用的系统内存。

为什么要将永久代(PermGen)替换为元空间(MetaSpace)呢？

整个永久代有一个 JVM 本身设置固定大小上线，无法进行调整，而元空间使用的是直接内存，受本机可用内存的限制，并且永远不会得到java.lang.OutOfMemoryError。你可以使用 `-XX:MaxMetaspaceSize` 标志设置最大元空间大小，默认值为 `unlimited`，这意味着它只受系统内存的限制。`-XX: MetaspaceSize` 调整标志定义元空间的初始大小如果未指定此标志，则 Metaspace 将根据运行时的应用程序需求动态地重新调整大小。

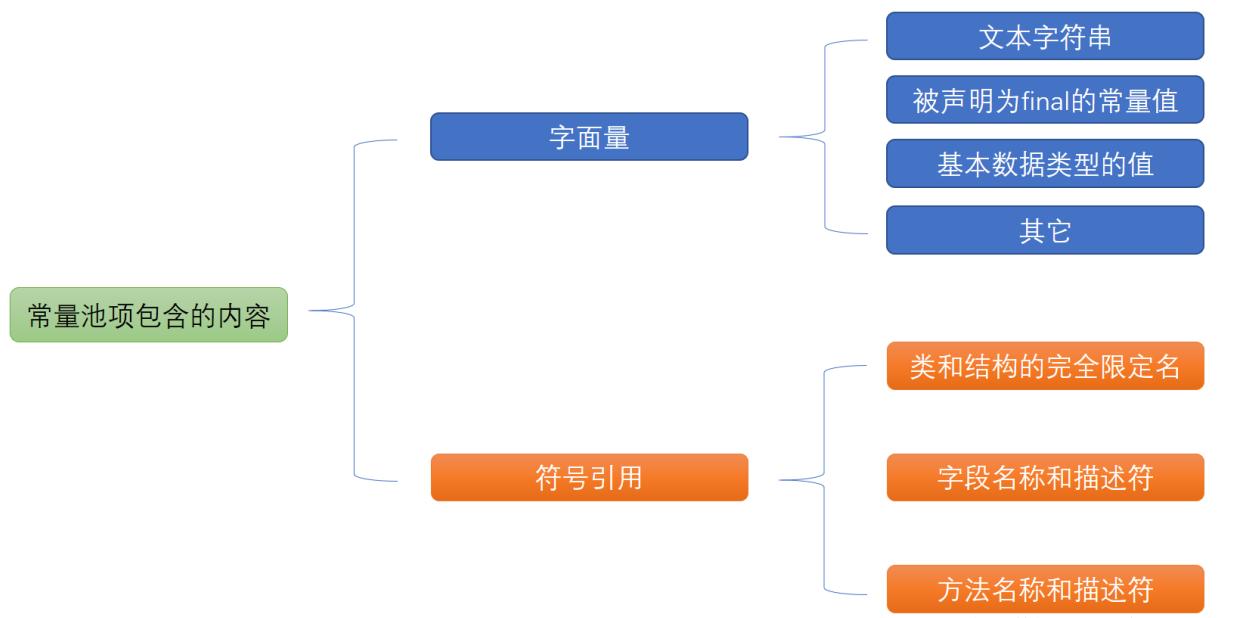
当然这只是其中一个原因，还有很多底层的原因，这里就不提了。

运行时常量池

运行时常量池是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有常量池信息（用于存放编译期生成的各种字面量和符号引用）

既然运行时常量池时方法区的一部分，自然受到方法区内存的限制，当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常。

JDK1.7及之后版本的 JVM 已经将运行时常量池从方法区中移了出来，在 Java 堆（Heap）中开辟了一块区域存放运行时常量池。



图片来源：<https://blog.csdn.net/wangbiao007/article/details/78545189>

直接内存

直接内存并不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用。而且也可能导致 OutOfMemoryError 异常出现。

JDK1.4 中新加入的 **NIO(New Input/Output)** 类，引入了一种基于通道（Channel）与缓存区（Buffer）的 I/O 方式，它可以直接使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。这样就能在一些场景中显著提高性能，因为避免了在 Java 堆和 Native 堆之间来回复制数据。

本机直接内存的分配不会收到 Java 堆的限制，但是，既然是内存就会受到本机总内存大小以及处理器寻址空间的限制。

2.4.2 说一下Java对象的创建过程

下图便是 Java 对象的创建过程，我建议最好是能默写出来，并且要掌握每一步在做什么。

Java 创建对象的过程



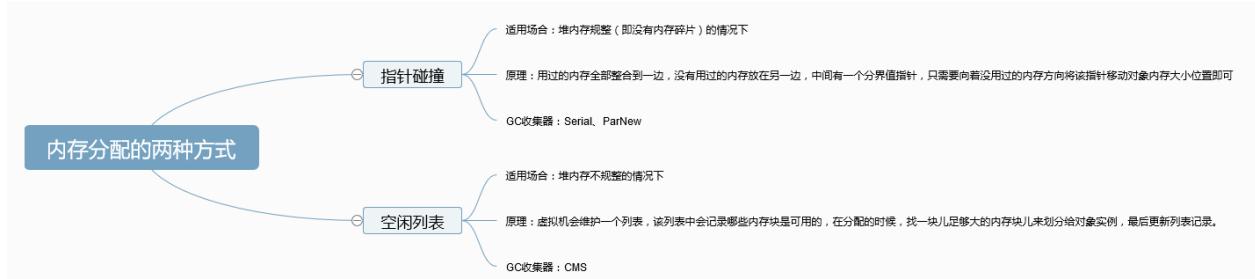
微信公众号：Java面试通关手册

①类加载检查：虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到这个类的符号引用，并且检查这个符号引用代表的类是否已被加载过、解析和初始化过。如果没有，那必须先执行相应的类加载过程。

②分配内存：在类加载检查通过后，接下来虚拟机将为新生对象分配内存。对象所需的内存大小在类加载完成后便可确定，为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。分配方式有“指针碰撞”和“空闲列表”两种，选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

内存分配的两种方式：（补充内容，需要掌握）

选择以上两种方式中的哪一种，取决于 Java 堆内存是否规整。而 Java 堆内存是否规整，取决于 GC 收集器的算法是“标记-清除”，还是“标记-整理”（也称作“标记-压缩”），值得注意的是，复制算法内存也是规整的



内存分配并发问题（补充内容，需要掌握）

在创建对象的时候有一个很重要的问题，就是线程安全，因为在实际开发过程中，创建对象是很频繁的事情，作为虚拟机来说，必须要保证线程是安全的，通常来讲，虚拟机采用两种方式来保证线程安全：

- **CAS+失败重试：** CAS 是乐观锁的一种实现方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性。
- **TLAB：** 为每一个线程预先在Eden区分配一块儿内存，JVM在给线程中的对象分配内存时，首先在 TLAB 分配，当对象大于 TLAB 中的剩余内存或 TLAB 的内存已用尽时，再采用上述的 CAS 进行内存分配

③初始化零值：内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值（不包括对象头），这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

④设置对象头： 初始化零值完成之后，虚拟机要对对象进行必要的设置，例如这个对象是那个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象头中。另外，根据虚拟机当前运行状态的不同，如是否启用偏向锁等，对象头会有不同的设置方式。

⑤执行 `init` 方法： 在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，`<init>` 方法还没有执行，所有的字段都还为零。所以一般来说，执行 `new` 指令之后会接着执行 `<init>` 方法，把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

2.4.3 对象的访问定位有哪两种方式？

建立对象就是为了使用对象，我们的Java程序通过栈上的 `reference` 数据来操作堆上的具体对象。对象的访问方式有虚拟机实现而定，目前主流的访问方式有①使用句柄和②直接指针两种：

1. 句柄： 如果使用句柄的话，那么 Java 堆中将会划分出一块内存来作为句柄池，`reference` 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息；

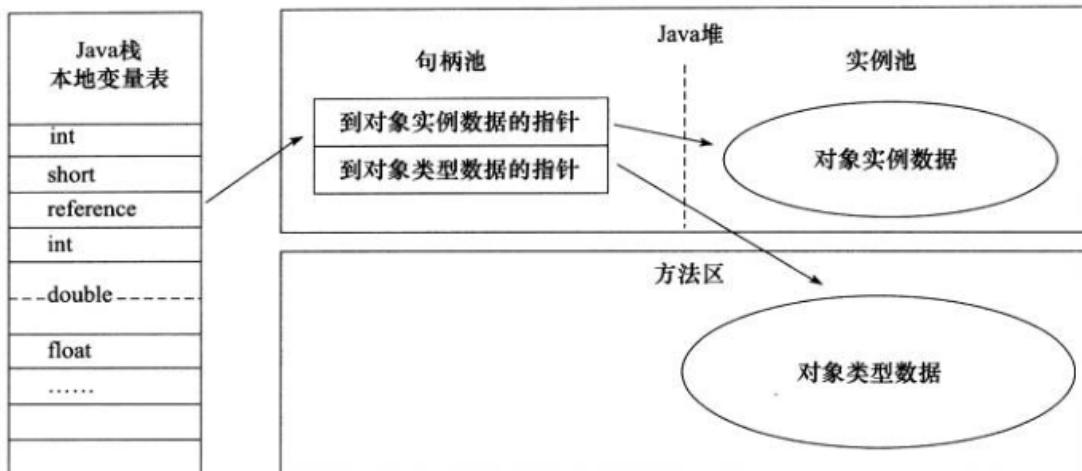


图 2-2 通过句柄访问对象

2. 直接指针： 如果使用直接指针访问，那么 Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而 `reference` 中存储的直接就是对象的地址。

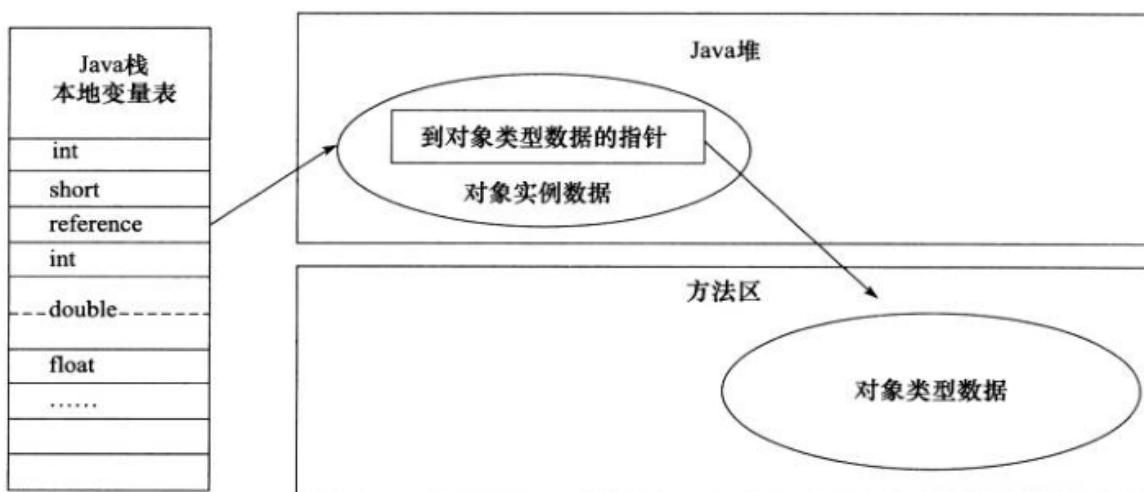
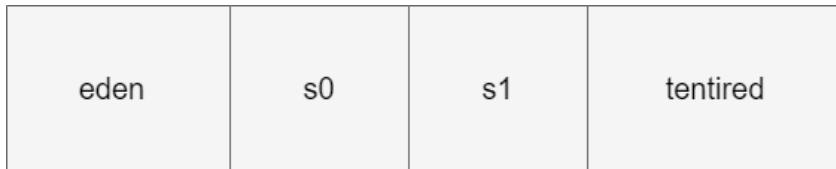


图 2-3 通过直接指针访问对象

这两种对象访问方式各有优势。使用句柄来访问的最大好处是 `reference` 中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而 `reference` 本身不需要修改。使用直接指针访问方式最大的好处就是速度快，它节省了一次指针定位的时间开销。

2.4.4 说一下堆内存中对象的分配的基本策略

堆空间的基本结构：



上图所示的 `eden` 区、`s0` 区、`s1` 区都属于新生代，`tentired` 区属于老年代。大部分情况，对象都会首先在 `Eden` 区域分配，在一次新生代垃圾回收后，如果对象还存活，则会进入 `s0` 或者 `s1`，并且对象的年龄还会加 1 (`Eden` 区 → `Survivor` 区后对象的初始年龄变为 1)，当它的年龄增加到一定程度（默认为 15 岁），就会被晋升到老年代中。对象晋升到老年代的年龄阈值，可以通过参数 `-XX:MaxTenuringThreshold` 来设置。

另外，大对象和长期存活的对象会直接进入老年代。



公众号：JavaGuide

2.4.5 Minor GC 和 Full GC 有什么不同呢？

大多数情况下，对象在新生代中 `eden` 区分配。当 `eden` 区没有足够空间进行分配时，虚拟机将发起一次 `Minor GC`。

- **新生代 GC (Minor GC)** : 指发生在新生代的垃圾收集动作，`Minor GC` 非常频繁，回收速度一般也比较快。
- **老年代 GC (Major GC/Full GC)** : 指发生在老年代的 GC，出现了 `Major GC` 经常会伴随至少一次的 `Minor GC` (并非绝对)，`Major GC` 的速度一般会比 `Minor GC` 的慢 10 倍以上。

2.4.6 如何判断对象是否死亡？(两种方法)

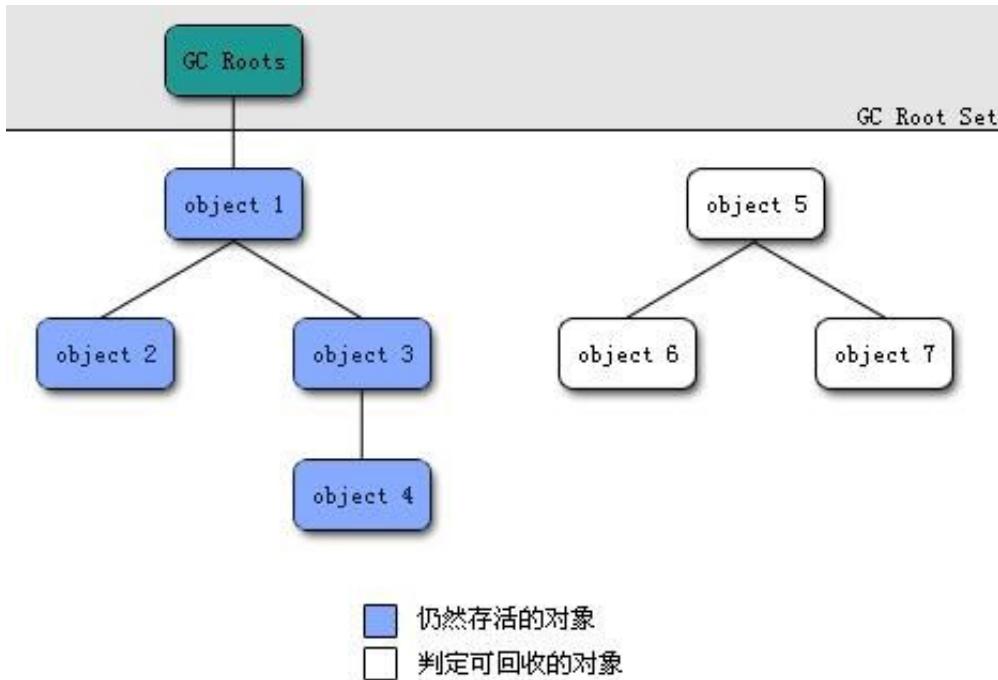
堆中几乎放着所有的对象实例，对堆垃圾回收前的第一步就是要判断哪些对象已经死亡（即不能再被任何途径使用的对象）。

引用计数法

给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加1；当引用失效，计数器就减1；任何时候计数器为0的对象就是不可能再被使用的。

可达性分析算法

这个算法的基本思想就是通过一系列的称为“**GC Roots**”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的。



2.4.7 简单的介绍一下强引用,软引用,弱引用,虚引用

无论是通过引用计数法判断对象引用数量，还是通过可达性分析法判断对象的引用链是否可达，判定对象的存活都与“引用”有关。

JDK1.2之前，Java中引用的定义很传统：如果reference类型的数据存储的数值代表的是另一块内存的起始地址，就称这块内存代表一个引用。

JDK1.2以后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用四种（引用强度逐渐减弱）

强引用(StrongReference)

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

软引用(SoftReference)

如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列 (ReferenceQueue) 联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中。

弱引用(WeakReference)

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列 (ReferenceQueue) 联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

4. 虚引用 (PhantomReference)

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

虚引用主要用来跟踪对象被垃圾回收的活动。

虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 (ReferenceQueue) 联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出 (OutOfMemory) 等问题的产生。

2.4.8 如何判断一个常量是废弃常量？

运行时常量池主要回收的是废弃的常量。那么，我们如何判断一个常量是废弃常量呢？

假如在常量池中存在字符串 "abc"，如果当前没有任何String对象引用该字符串常量的话，就说明常量 "abc" 就是废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

2.4.9 如何判断一个类是无用的类？

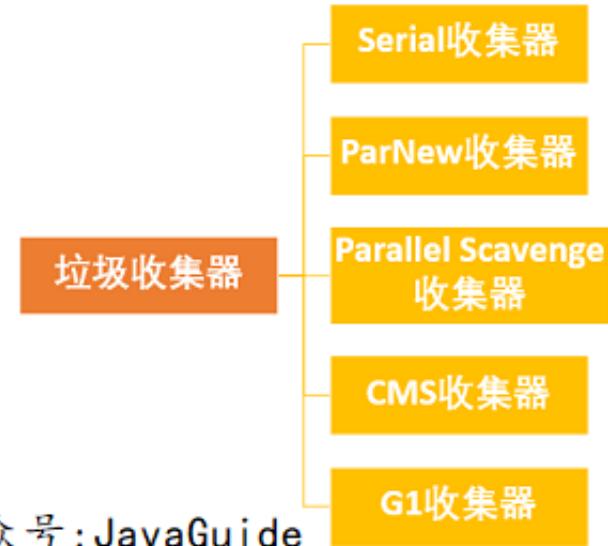
方法区主要回收的是无用的类，那么如何判断一个类是无用的类的呢？

判定一个常量是否是“废弃常量”比较简单，而要判定一个类是否是“无用的类”的条件则相对苛刻许多。类需要同时满足下面3个条件才能算是“无用的类”：

- 该类所有的实例都被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 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。

虚拟机可以对满足上述3个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样不使用了就会必然被回收。

2.4.10 垃圾收集有哪些算法，各自的特点？

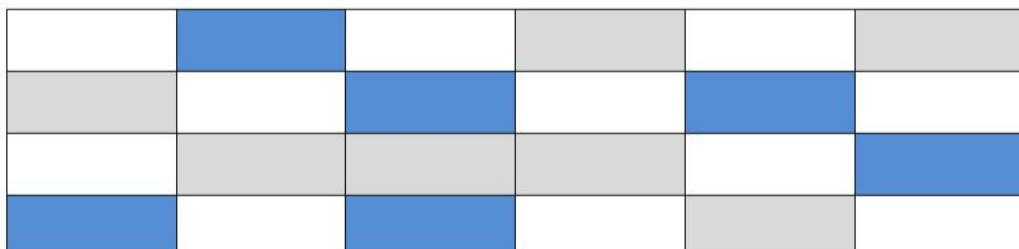


标记-清除算法

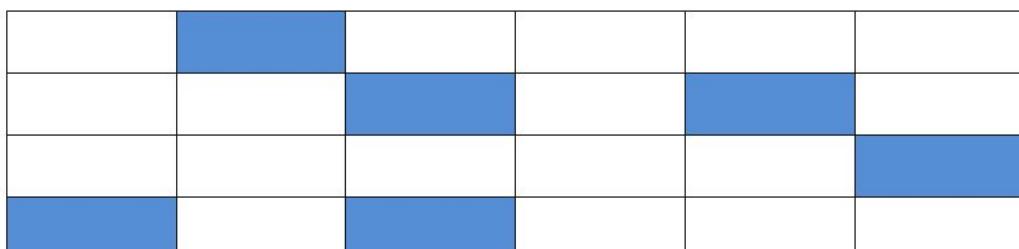
算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是最基础的收集算法，后续的算法都是对其不足进行改进得到。这种垃圾收集算法会带来两个明显的问题：

1. 效率问题
2. 空间问题（标记清除后会产生大量不连续的碎片）

内存整理前



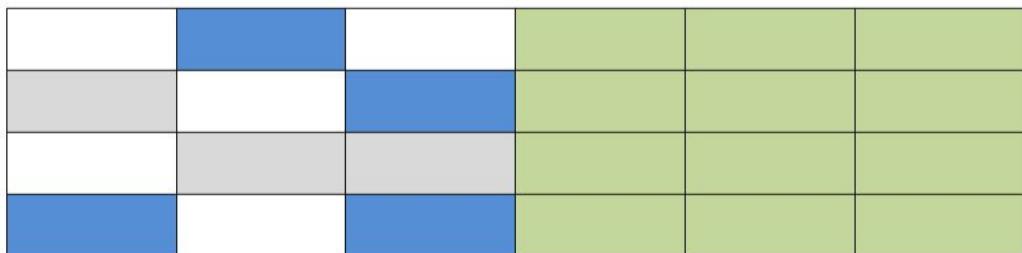
内存整理后



复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

内存整理前

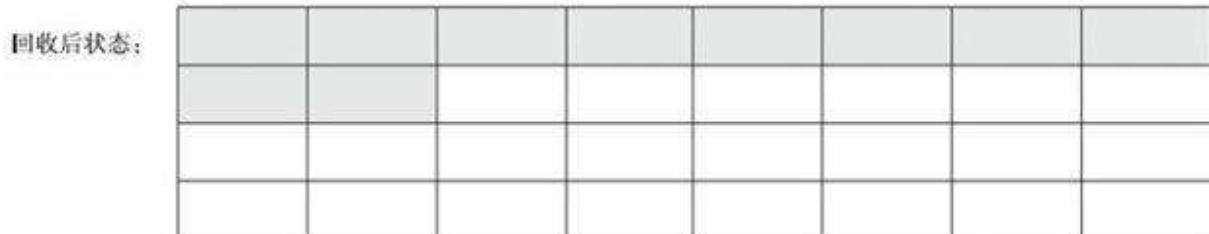
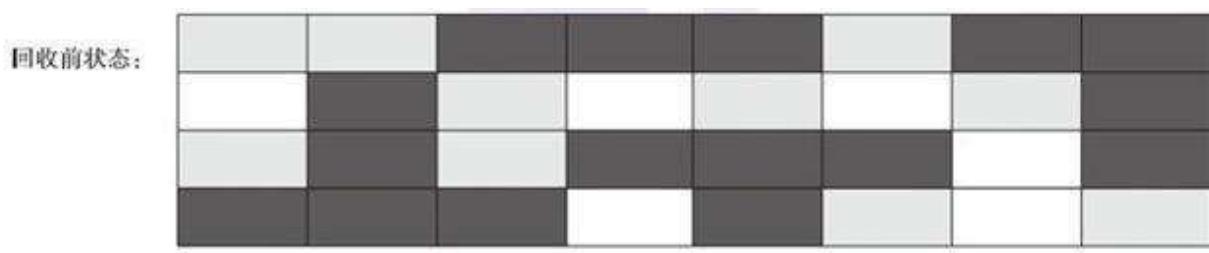


内存整理后



标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。



分代收集算法

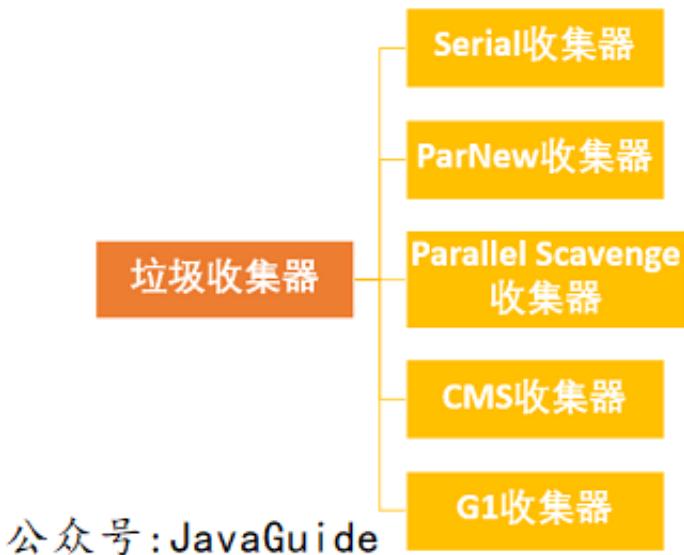
当前虚拟机的垃圾收集都采用分代收集算法，这种算法没有什么新的思想，只是根据对象存活周期的不同将内存分为几块。一般将java堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

2.4.11 HotSpot为什么要分为新生代和老年代？

主要是为了提升GC效率。上面提到的分代收集算法已经很好的解释了这个问题。

2.4.12 常见的垃圾回收器有那些？



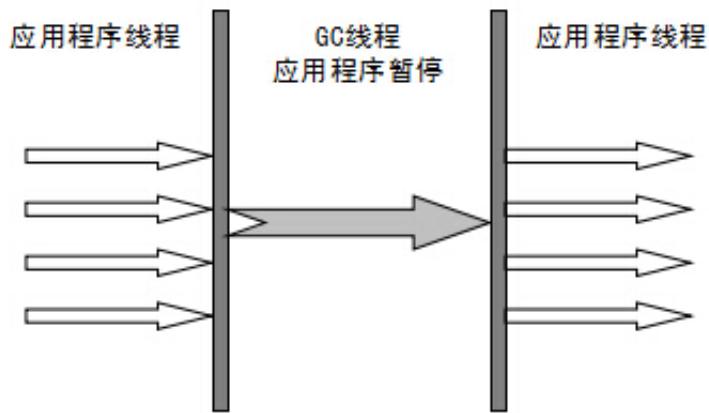
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们对各个收集器进行比较，但并非要挑选出一个最好的收集器。因为知道现在为止还没有最好的垃圾收集器出现，更加没有万能的垃圾收集器，我们能做的就是根据具体应用场景选择适合自己的垃圾收集器。试想一下：如果有一种四海之内、任何场景下都适用的完美收集器存在，那么我们的HotSpot虚拟机就不会实现那么多不同的垃圾收集器了。

Serial收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的“单线程”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程（"Stop The World"），直到它收集结束。

新生代采用复制算法，老年代采用标记-整理算法。



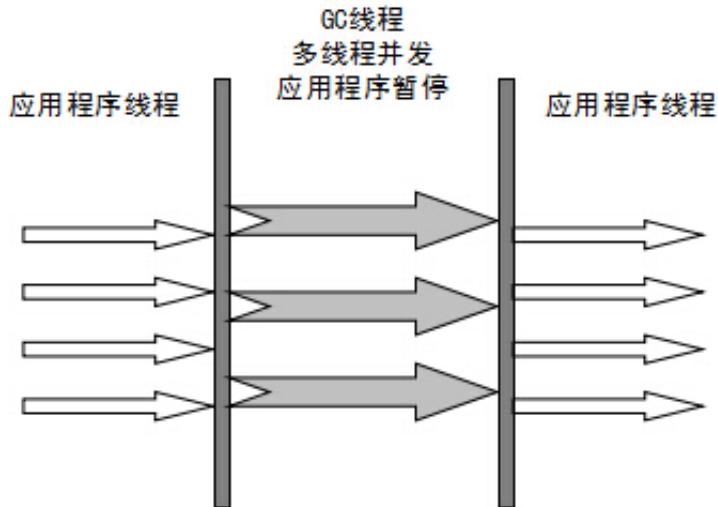
虚拟机的设计者们当然知道Stop The World带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是Serial收集器有没有优于其他垃圾收集器的地方呢？当然有，它简单而高效（与其他收集器的单线程相比）。Serial收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial收集器对于运行在Client模式下的虚拟机来说是个不错的选择。

ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和Serial收集器完全一样。

新生代采用复制算法，老年代采用标记-整理算法。



它是许多运行在Server模式下的虚拟机的首要选择，除了Serial收集器外，只有它能与CMS收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

并行和并发概念补充：

- **并行 (Parallel)**：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- **并发 (Concurrent)**：指用户线程与垃圾收集线程同时执行（但不一定是并行，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个CPU上。

Parallel Scavenge收集器

Parallel Scavenge 收集器类似于ParNew 收集器。那么它有什么特别之处呢？

-XX:+UseParallelGC

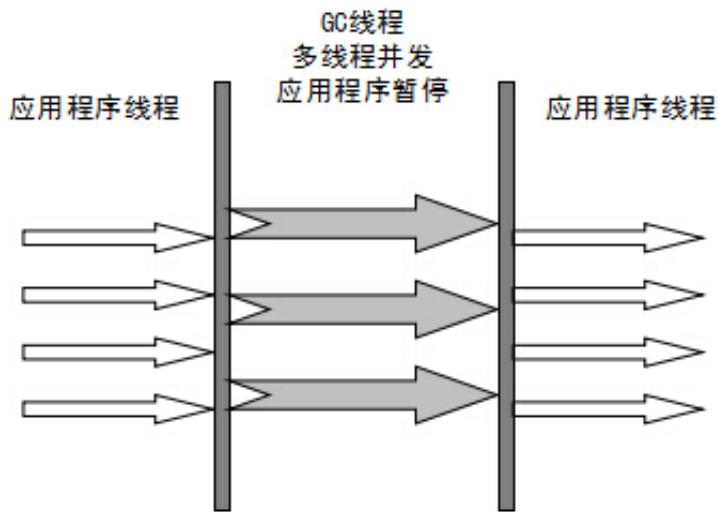
使用Parallel收集器+ 老年代串行

-XX:+UseParallelOldGC

使用Parallel收集器+ 老年代并行

Parallel Scavenge收集器关注点是吞吐量（高效率的利用CPU）。CMS等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是CPU中用于运行用户代码的时间与CPU总消耗时间的比值。Parallel Scavenge收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，手工优化存在的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。



Serial Old收集器

Serial收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在JDK1.5以及以前的版本中与Parallel Scavenge收集器搭配使用，另一种用途是作为CMS收集器的后备方案。

Parallel Old收集器

Parallel Scavenge收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及CPU资源的场合，都可以优先考虑 Parallel Scavenge收集器和Parallel Old收集器。

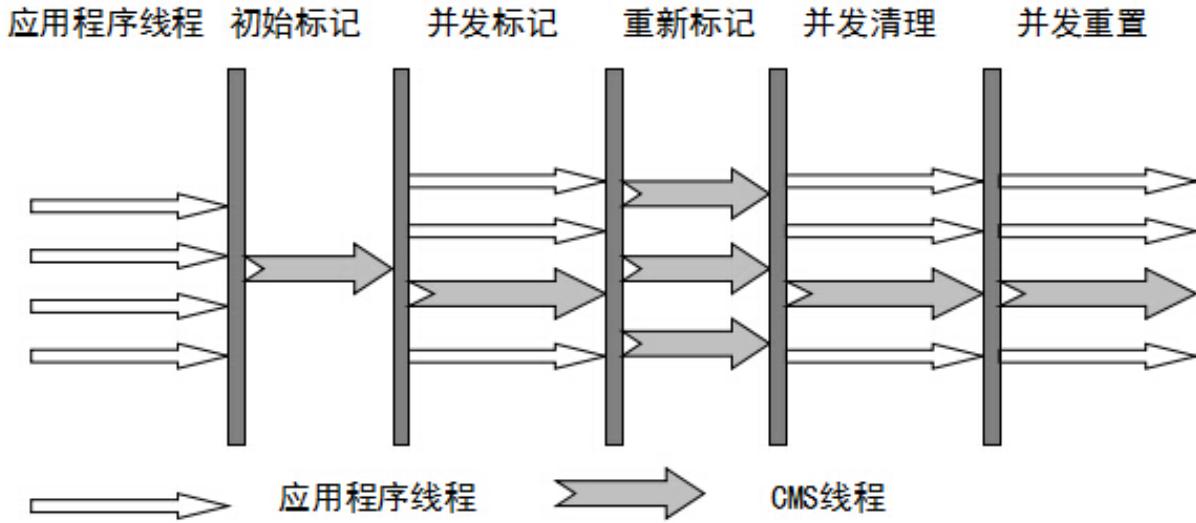
CMS收集器

CMS (Concurrent Mark Sweep) 收集器是一种以获取最短回收停顿时间为为目标的收集器。它而非常符合在注重用户体验的应用上使用。

CMS (Concurrent Mark Sweep) 收集器是HotSpot虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作。

从名字中的Mark Sweep这两个词可以看出，CMS收集器是一种“标记-清除”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- **初始标记：**暂停所有的其他线程，并记录下直接与root相连的对象，速度很快；
- **并发标记：**同时开启GC和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以GC线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。
- **重新标记：**重新标记阶段就是为了修正并发标记期间因为用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段的时间稍长，远远比并发标记阶段时间短
- **并发清除：**开启用户线程，同时GC线程开始对为标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：**并发收集、低停顿**。但是它有下面三个明显的缺点：

- 对CPU资源敏感；
- 无法处理浮动垃圾；
- 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

G1收集器

G1 (Garbage-First)是一款面向服务器的垃圾收集器，主要针对配备多颗处理器及大容量内存的机器。以极高概率满足GC停顿时间要求的同时，还具备高吞吐量性能特征。

被视为JDK1.7中HotSpot虚拟机的一个重要进化特征。它具备一下特点：

- **并行与并发：**G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU (CPU或者CPU核心) 来缩短Stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。
- **分代收集：**虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。
- **空间整合：**与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。
- **可预测的停顿：**这是G1相对于CMS的另一个大优势，降低停顿时间是G1 和 CMS 共同的关注点，但G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内。

G1收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记

- 最终标记
- 筛选回收

G1收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的Region(这也就是它的名字Garbage-First的由来)。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了GF收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。

2.4.13 类文件结构

介绍一下类文件结构吧！

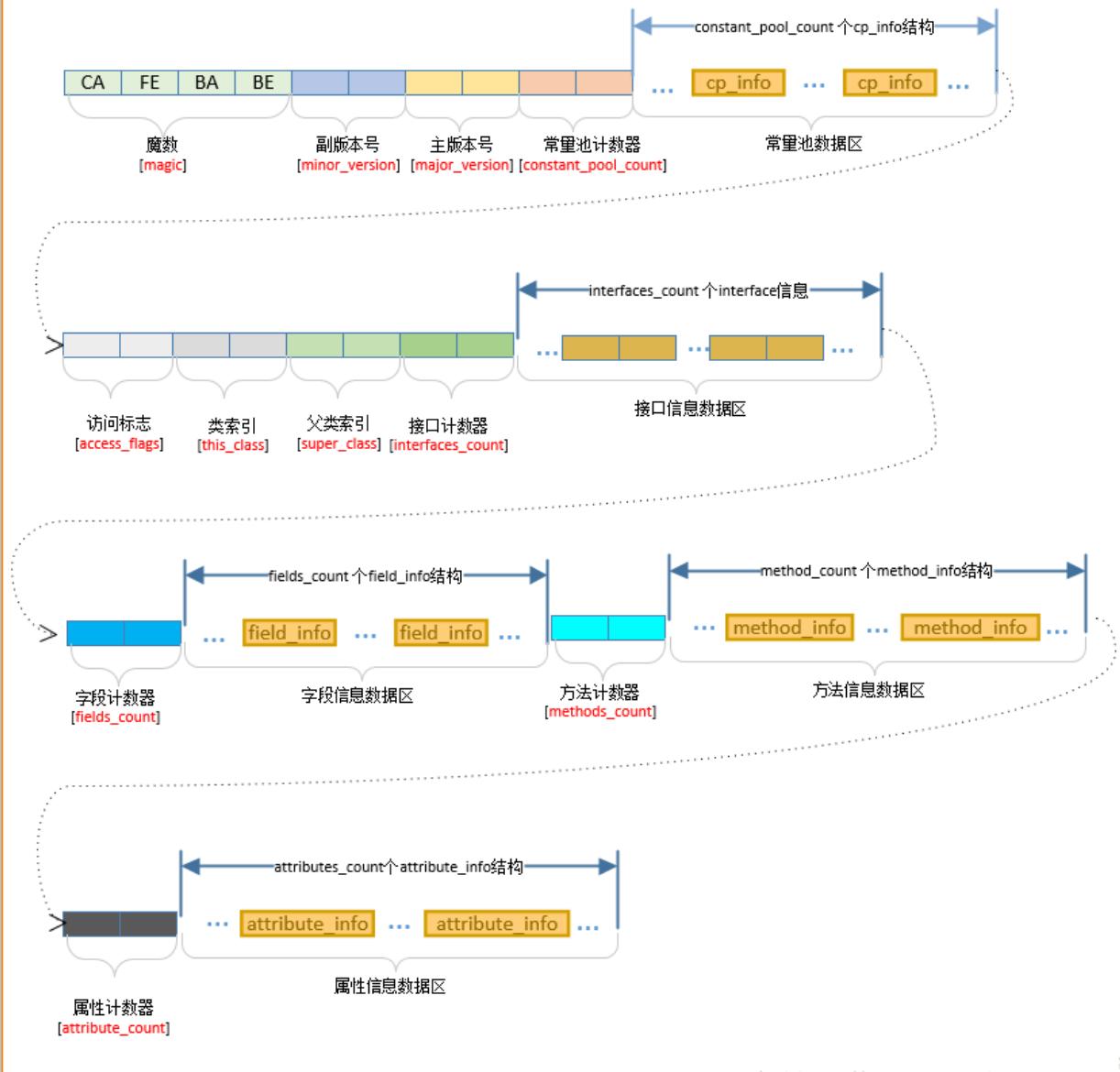
根据 Java 虚拟机规范，类文件由单个 ClassFile 结构组成：

```
ClassFile {
    u4          magic; //Class 文件的标志
    u2          minor_version; //Class 的小版本号
    u2          major_version; //Class 的大版本号
    u2          constant_pool_count; //常量池的数量
    cp_info     constant_pool[constant_pool_count-1]; //常量池
    u2          access_flags; //Class 的访问标记
    u2          this_class; //当前类
    u2          super_class; //父类
    u2          interfaces_count; //接口
    u2          interfaces[interfaces_count]; //一个类可以实现多个接口
    u2          fields_count; //Class 文件的字段属性
    field_info   fields[fields_count]; //一个类会可以有个字段
    u2          methods_count; //Class 文件的方法数量
    method_info  methods[methods_count]; //一个类可以有个多个方法
    u2          attributes_count; //此类的属性表中的属性数
    attribute_info attributes[attributes_count]; //属性表集合
}
```

Class文件字节码结构组织示意图（之前在网上保存的，非常不错，原出处不明）：

Class文件字节码结构组织示意图

注：被编译器编译成.class字节码文件的字节流以及其组织结构如下所示：



下面会按照上图结构按顺序详细介绍一下 Class 文件结构涉及到的一些组件。

1. 魔数：确定这个文件是否为一个能被虚拟机接收的 Class 文件。
2. Class 文件版本：Class 文件的版本号，保证编译正常执行。
3. 常量池：常量池主要存放两大常量：字面量和符号引用。
4. 访问标志：标志用于识别一些类或者接口层次的访问信息，包括：这个 Class 是类还是接口，是否为 public 或者 abstract 类型，如果是类的话是否声明为 final 等等。
5. 当前类索引,父类索引：类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名，由于 Java 语言的单继承，所以父类索引只有一个，除了 `java.lang.Object` 之外，所有的 java 类都有父类，因此除了 `java.lang.Object` 外，所有 Java 类的父类索引都不为 0。
6. 接口索引集合：接口索引集合用来描述这个类实现了那些接口，这些被实现的接口将按 `implements` (如果这个类本身是接口的话则是 `extends`) 后的接口顺序从左到右排列在接口索

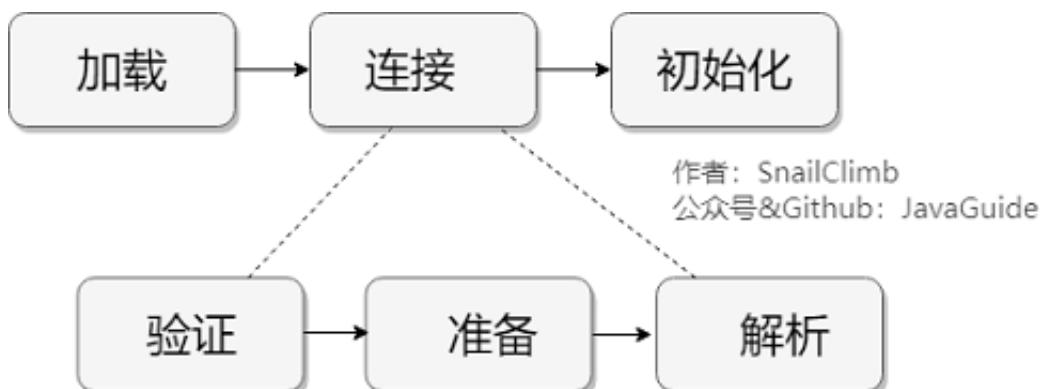
引集合中。

7. **字段表集合**：描述接口或类中声明的变量。字段包括类级变量以及实例变量，但不包括在方法内部声明的局部变量。
8. **方法表集合**：类中的方法。
9. **属性表集合**：在 Class 文件，字段表，方法表中都可以携带自己的属性表集合。

2.4.14 类加载过程

知道类加载的过程吗？

类加载过程：加载→连接→初始化。连接过程又可分为三步：验证→准备→解析。



那加载这一步做了什么？

类加载过程的第一步，主要完成下面3件事情：

1. 通过全类名获取定义此类的二进制字节流
2. 将字节流所代表的静态存储结构转换为方法区的运行时数据结构
3. 在内存中生成一个代表该类的 Class 对象，作为方法区这些数据的访问入口

虚拟机规范对上面这3点并不具体，因此是非常灵活的。比如：“通过全类名获取定义此类的二进制字节流”并没有指明具体从哪里获取、怎样获取。比如：比较常见的就是从 ZIP 包中读取（日后出现的 JAR、EAR、WAR 格式的基础）、其他文件生成（典型应用就是 JSP）等等。

一个非数组类的加载阶段（加载阶段获取类的二进制字节流的动作）是可控性最强的阶段，这一步我们可以去完成还可以自定义类加载器去控制字节流的获取方式（重写一个类加载器的 `loadClass()` 方法）。数组类型不通过类加载器创建，它由 Java 虚拟机直接创建。

类加载器、双亲委派模型也是非常重要的知识点，这部分内容会在后面的问题中单独介绍到。

加载阶段和连接阶段的部分内容是交叉进行的，加载阶段尚未结束，连接阶段可能就已经开始了。

知道哪些类加载器？

JVM 中内置了三个重要的 `ClassLoader`，除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`：

1. **BootstrapClassLoader(启动类加载器)**：最顶层的加载类，由 C++ 实现，负责加载 `%JAVA_HOME%/lib` 目录下的 jar 包和类或者或被 `-Xbootclasspath` 参数指定的路径中的所有类。
2. **ExtensionClassLoader(扩展类加载器)**：主要负责加载目录 `%JRE_HOME%/lib/ext` 目录下

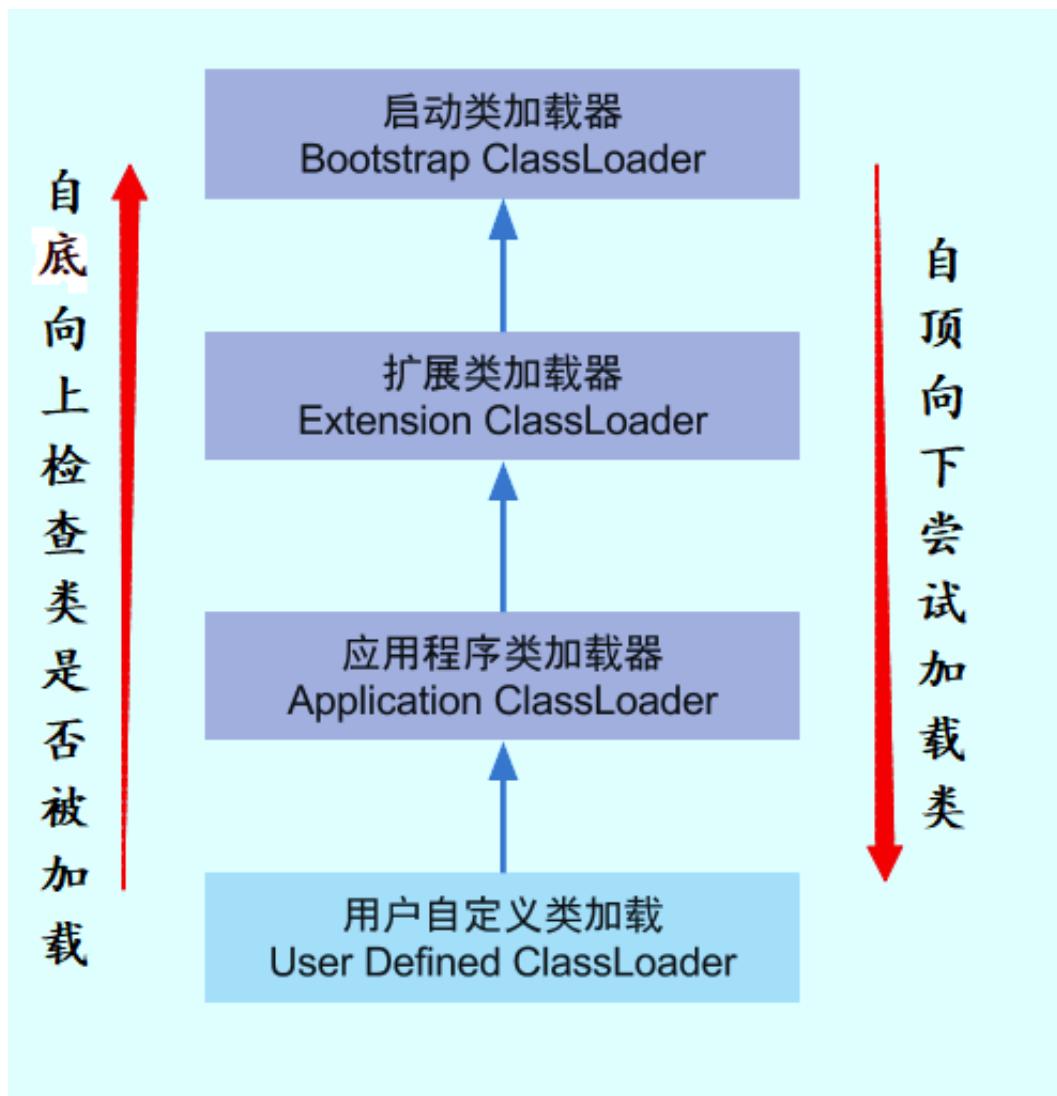
的jar包和类，或被 `java.ext.dirs` 系统变量所指定的路径下的jar包。

3. **AppClassLoader**(应用程序类加载器) :面向我们用户的加载器，负责加载当前应用classpath下的所有jar包和类。

双亲委派模型知道吗？能介绍一下吗？

双亲委派模型介绍

每一个类都有一个对应它的类加载器。系统中的 ClassLoader 在协同工作的时候会默认使用 **双亲委派模型**。即在类加载的时候，系统会首先判断当前类是否被加载过。已经被加载的类会直接返回，否则才会尝试加载。加载的时候，首先会把该请求委派该父类加载器的 `loadClass()` 处理，因此所有的请求最终都应该传送到顶层的启动类加载器 **BootstrapClassLoader** 中。当父类加载器无法处理时，才由自己来处理。当父类加载器为null时，会使用启动类加载器 **BootstrapClassLoader** 作为父类加载器。



每个类加载都有一个父类加载器，我们通过下面的程序来验证。

```
public class ClassLoaderDemo {  
    public static void main(String[] args) {  
        System.out.println("ClassLodarDemo's ClassLoader is " +  
ClassLoaderDemo.class.getClassLoader());  
        System.out.println("The Parent of ClassLodarDemo's ClassLoader  
is " + ClassLoaderDemo.class.getClassLoader().getParent());  
        System.out.println("The GrandParent of ClassLodarDemo's  
ClassLoader is " +  
ClassLoaderDemo.class.getClassLoader().getParent().getParent());  
    }  
}
```

Output

```
ClassLodarDemo's ClassLoader is  
sun.misc.Launcher$AppClassLoader@18b4aac2  
The Parent of ClassLodarDemo's ClassLoader is  
sun.misc.Launcher$ExtClassLoader@1b6d3586  
The GrandParent of ClassLodarDemo's ClassLoader is null
```

`AppClassLoader` 的父类加载器为 `ExtClassLoader` `ExtClassLoader` 的父类加载器为 `null`, `null` 并不代表 `ExtClassLoader` 没有父类加载器, 而是 `Bootstrap ClassLoader`。

其实这个双亲翻译的容易让别人误解, 我们一般理解的双亲都是父母, 这里的双亲更多地表达的是“父母这一辈”的人而已, 并不是说真的有一个 `Mather ClassLoader` 和一个 `Father ClassLoader`。另外, 类加载器之间的“父子”关系也不是通过继承来体现的, 是由“优先级”来决定。官方API文档对这部分的描述如下:

The Java platform uses a delegation model for loading classes. The basic idea is that every class loader has a "parent" class loader. When loading a class, a class loader first "delegates" the search for the class to its parent class loader before attempting to find the class itself.

双亲委派模型实现源码分析

双亲委派模型的实现代码非常简单, 逻辑非常清晰, 都集中在 `java.lang.ClassLoader` 的 `loadClass()` 中, 相关代码如下所示。

```
private final ClassLoader parent;  
protected Class<?> loadClass(String name, boolean resolve)  
    throws ClassNotFoundException  
{  
    synchronized (getClassLoadingLock(name)) {  
        // 首先, 检查请求的类是否已经被加载过  
        Class<?> c = findLoadedClass(name);  
        if (c == null) {
```

```

        long t0 = System.nanoTime();
        try {
            if (parent != null) { //父加载器不为空，调用父加载器
loadClass()方法处理
                c = parent.loadClass(name, false);
            } else { //父加载器为空，使用启动类加载器
BootstrapClassLoader 加载
                c = findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException e) {
            //抛出异常说明父类加载器无法完成加载请求
        }

        if (c == null) {
            long t1 = System.nanoTime();
            //自己尝试加载
            c = findClass(name);

            // this is the defining class loader; record the
stats
        }

sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);

sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
        sun.misc.PerfCounter.getFindClasses().increment();
    }
}
if (resolve) {
    resolveClass(c);
}
return c;
}
}
}

```

双亲委派模型带来了什么好处呢？

双亲委派模型保证了Java程序的稳定运行，可以避免类的重复加载（JVM 区分不同类型的方式不仅仅根据类名，相同的类文件被不同的类加载器加载产生的是两个不同的类），也保证了 Java 的核心 API 不被篡改。如果不用没有使用双亲委派模型，而是每个类加载器加载自己的话就会出现一些问题，比如我们编写一个称为 `java.lang.Object` 类的话，那么程序运行的时候，系统就会出现多个不同的 `Object` 类。

如果我们不想用双亲委派模型怎么办？

为了避免双亲委托机制，我们可以自己定义一个类加载器，然后重载 `loadClass()` 即可。

如何自定义类加载器？

除了 `BootstrapClassLoader` 其他类加载器均由 Java 实现且全部继承自 `java.lang.ClassLoader`。如果我们要自定义自己的类加载器，很明显需要继承 `ClassLoader`。

三 计算机基础

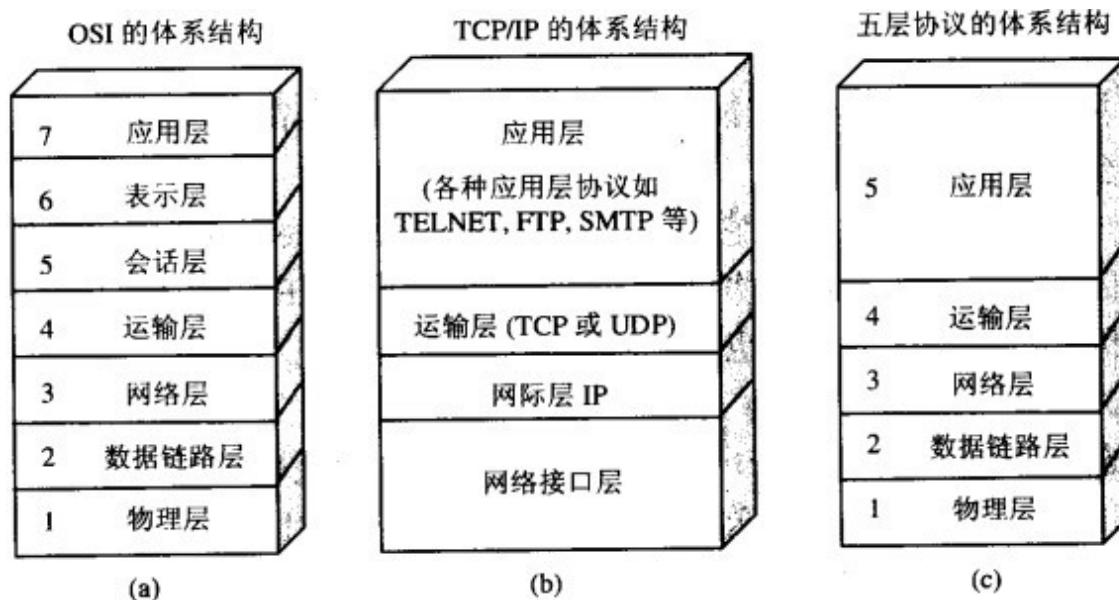
3.1 计算机网络

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#)（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

3.1.1 OSI与TCP/IP各层的结构与功能,都有哪些协议?

学习计算机网络时我们一般采用折中的办法，也就是中和 OSI 和 TCP/IP 的优点，采用一种只有五层协议的体系结构，这样既简洁又能将概念阐述清楚。



计算机网络体系结构：(a) OSI 的七层协议；(b) TCP/IP 的四层协议；(c) 五层协议

结合互联网的情况，自上而下地，非常简要的介绍一下各层的作用。

应用层

应用层(application-layer)的任务是通过应用进程间的交互来完成特定网络应用。应用层协议定义的是应用进程（进程：主机中正在运行的程序）间的通信和交互的规则。对于不同的网络应用需要不同的应用层协议。在互联网中应用层协议很多，如域名系统DNS，支持万维网应用的HTTP协议，支持电子邮件的SMTP协议等等。我们把应用层交互的数据单元称为报文。

域名系统

域名系统(Domain Name System缩写 DNS, Domain Name被译为域名)是因特网的一项核心服务，它作为可以将域名和IP地址相互映射的一个分布式数据库，能够使人更方便的访问互联网，而不用去记住能够被机器直接读取的IP数串。（百度百科）例如：一个公司的 Web 网站可看作是它在网上的门户，而域名就相当于其门牌地址，通常域名都使用该公司的名称或简称。例如上面提到的微软公司的域名，类似的还有：IBM 公司的域名是 www.ibm.com、Oracle 公司的域名是 www.oracle.com、Cisco公司的域名是 www.cisco.com 等。

HTTP协议

超文本传输协议 (HTTP, HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议。所有的 WWW (万维网) 文件都必须遵守这个标准。设计 HTTP 最初的目的是为了提供一种发布和接收 HTML 页面的方法。（百度百科）

运输层

运输层(transport layer)的主要任务就是负责向两台主机进程之间的通信提供通用的数据传输服务。应用进程利用该服务传送应用层报文。“通用的”是指并不针对某一个特定的网络应用，而是多种应用可以使用同一个运输层服务。由于一台主机可同时运行多个线程，因此运输层有复用和分用的功能。所谓复用就是指多个应用层进程可同时使用下面运输层的服务，分用和复用相反，是运输层把收到的信息分别交付上面应用层中的相应进程。

运输层主要使用以下两种协议：

1. 传输控制协议 TCP (Transmission Control Protocol) --提供面向连接的，可靠的数据传输服务。
2. 用户数据协议 UDP (User Datagram Protocol) --提供无连接的，尽最大努力的数据传输服务（不保证数据传输的可靠性）。

TCP 与 UDP 的对比见问题三。

网络层

在 计算机网络中进行通信的两个计算机之间可能会经过很多个数据链路，也可能还要经过很多通信子网。网络层的任务就是选择合适的网间路由和交换结点，确保数据及时传送。在发送数据时，网络层把运输层产生的报文段或用户数据报封装成分组和包进行传送。在 TCP/IP 体系结构中，由于网络层使用 IP 协议，因此分组也叫 IP 数据报，简称 数据报。

这里要注意：不要把运输层的“用户数据报 UDP ”和网络层的“ IP 数据报”弄混。另外，无论是哪一层的数据单元，都可笼统地用“分组”来表示。

这里强调指出，网络层中的“网络”二字已经不是我们通常谈到的具体网络，而是指计算机网络体系结构模型中第三层的名称。

互联网是由大量的异构 (heterogeneous) 网络通过路由器 (router) 相互连接起来的。互联网使用的网络层协议是无连接的网际协议 (Internet Protocol) 和许多路由选择协议，因此互联网的网络层也叫做网际层或IP层。

数据链路层

数据链路层(data link layer)通常简称为链路层。两台主机之间的数据传输，总是在一段一段的链路上传送的，这就需要使用专门的链路层的协议。在两个相邻节点之间传送数据时，数据链路层将网络层交下来的 IP 数据报组装成帧，在两个相邻节点间的链路上传送帧。每一帧包括数据和必要的控制信息（如同步信息，地址信息，差错控制等）。

在接收数据时，控制信息使接收端能够知道一个帧从哪个比特开始和到哪个比特结束。这样，数据链路层在收到一个帧后，就可从中提出数据部分，上交给网络层。控制信息还使接收端能够检测到所收到的帧中有误差错。如果发现差错，数据链路层就简单地丢弃这个出了差错的帧，以避免继续在网络中传送下去白白浪费网络资源。如果需要改正数据在链路层传输时出现差错（这就是说，数据链路层不仅要检错，而且还要纠错），那么就要采用可靠性传输协议来纠正出现的差错。这种方法会使链路层的协议复杂些。

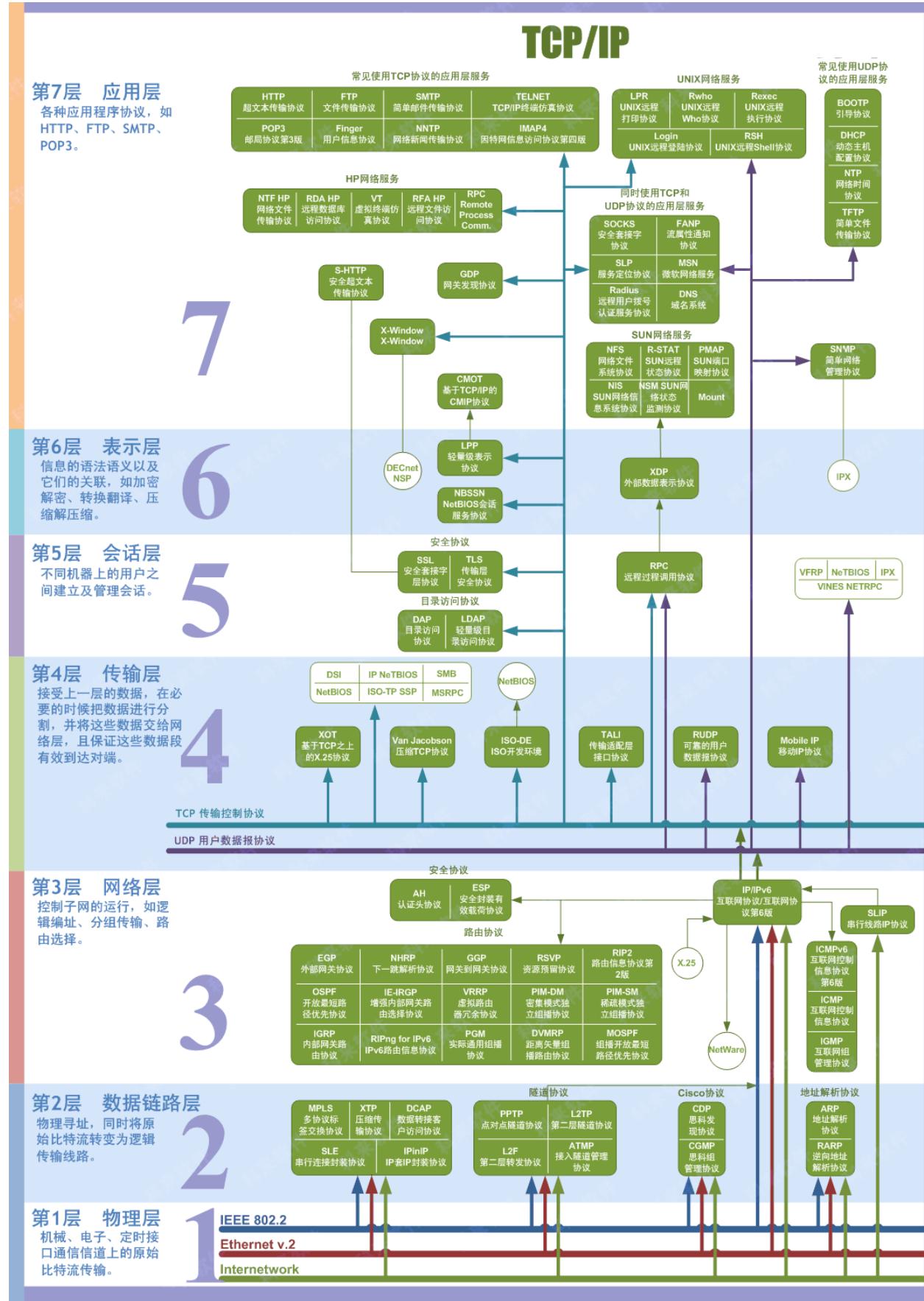
物理层

在物理层上所传送的数据单位是比特。物理层(physical layer)的作用是实现相邻计算机节点之间比特流的透明传送，尽可能屏蔽掉具体传输介质和物理设备的差异。使其上面的数据链路层不必考虑网络的具体传输介质是什么。“透明传送比特流”表示经实际电路传送后的比特流没有发生变化，对传送的比特流来说，这个电路好像是看不见的。

在互联网使用的各种协中最重要和最著名的就是 TCP/IP 两个协议。现在人们经常提到的TCP/IP并不一定单指TCP和IP这两个具体的协议，而往往表示互联网所使用的整个TCP/IP协议族。

总结一下

上面我们对计算机网络的五层体系结构有了初步的了解，下面附送一张七层体系结构图总结一下。图片来源：https://blog.csdn.net/yaopeng_2005/article/details/7064869

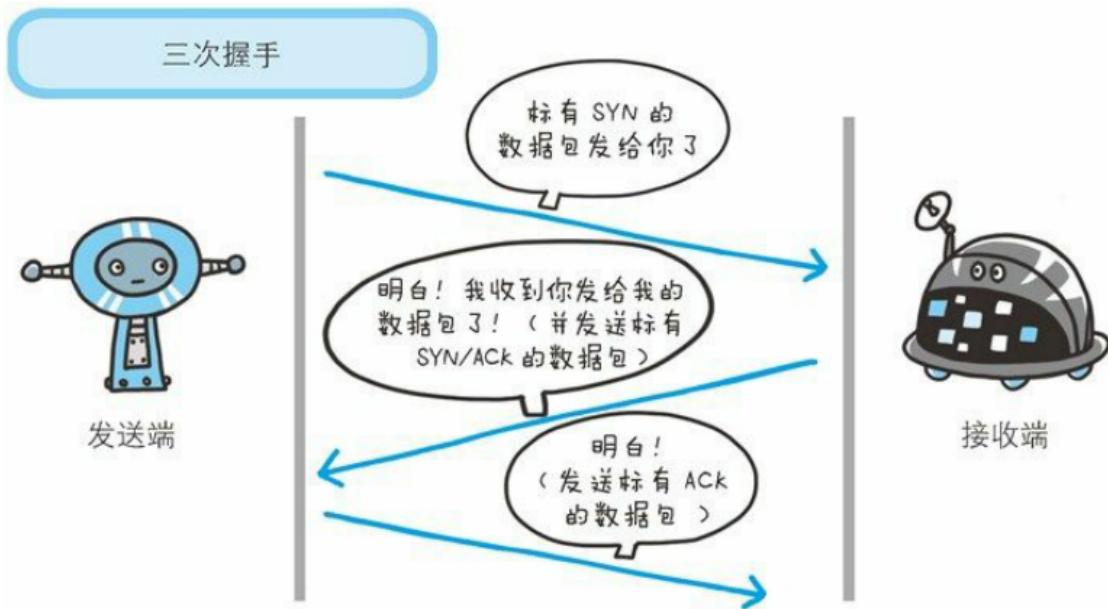


3.1.2 TCP 三次握手和四次挥手(面试常客)

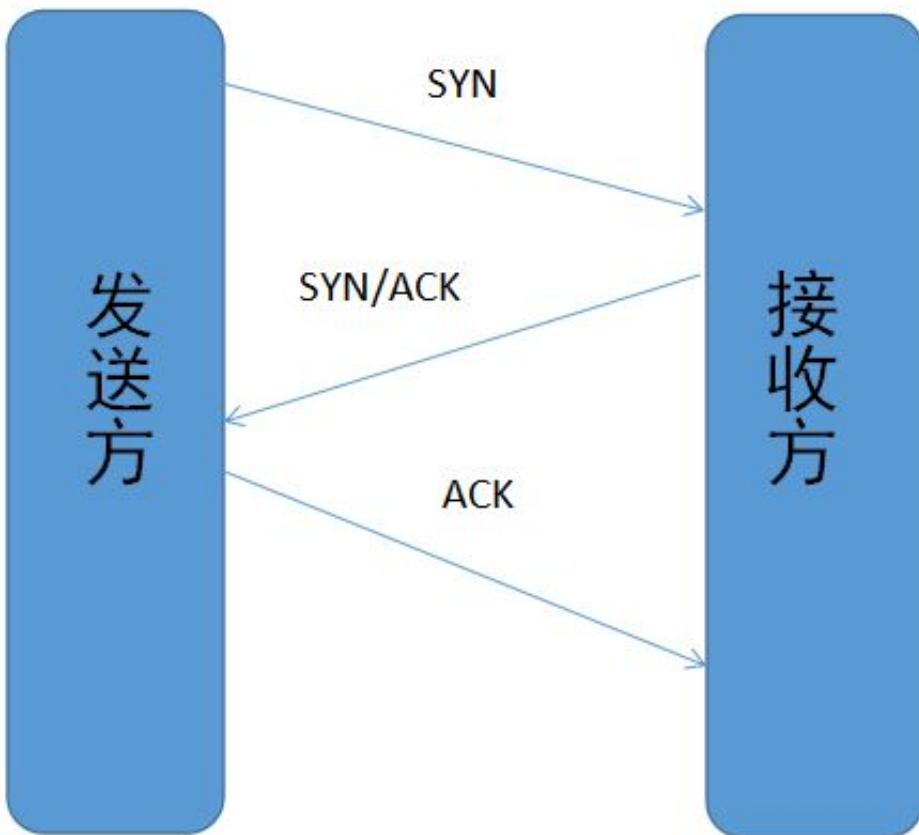
为了准确无误地把数据送达目标处，TCP协议采用了三次握手策略。

TCP 三次握手漫画图解

如下图所示，下面的两个机器人通过3次握手确定了对方能正确接收和发送消息(图片来源：《图解HTTP》)。



简单示意图：



- 客户端-发送带有 SYN 标志的数据包-一次握手-服务端
- 服务端-发送带有 SYN/ACK 标志的数据包-二次握手-客户端
- 客户端-发送带有带有 ACK 标志的数据包-三次握手-服务端

为什么要三次握手

三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收是正常的。

第一次握手：Client 什么都不能确认；Server 确认了对方发送正常，自己接收正常

第二次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：对方发送正常，自己接收正常

第三次握手：Client 确认了：自己发送、接收正常，对方发送、接收正常；Server 确认了：自己发送、接收正常，对方发送、接收正常

所以三次握手就能确认双发收发功能都正常，缺一不可。

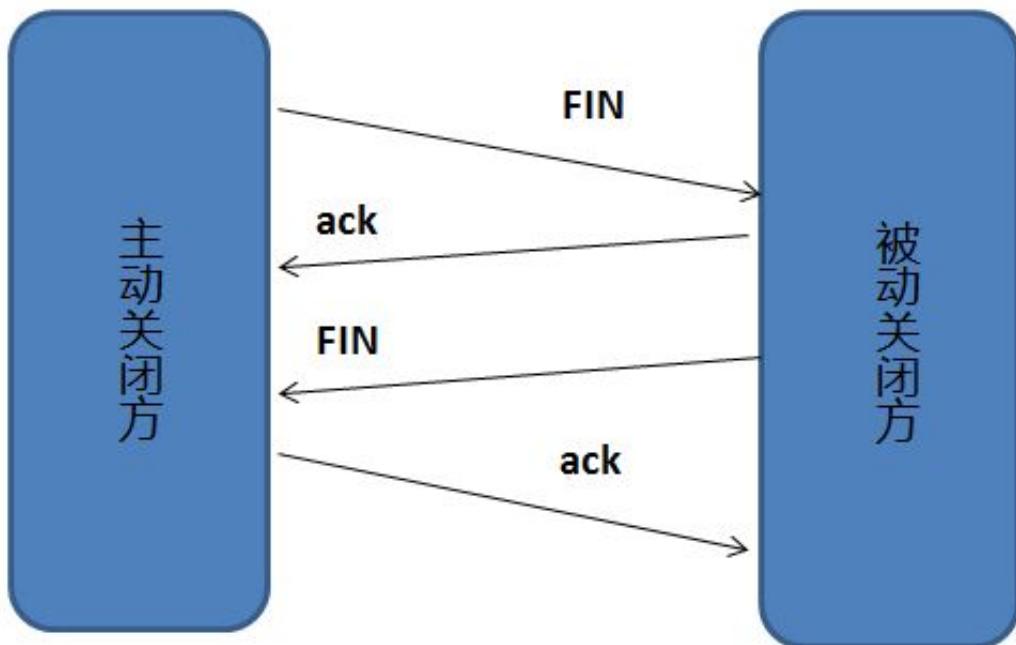
为什么要传回 SYN

接收端传回发送端所发送的 SYN 是为了告诉发送端，我接收到的信息确实就是你所发送的信号了。

SYN 是 TCP/IP 建立连接时使用的握手信号。在客户机和服务器之间建立正常的 TCP 网络连接时，客户机首先发出一个 SYN 消息，服务器使用 SYN-ACK 应答表示接收到了这个消息，最后客户机再以 ACK(Acknowledgement[汉译：确认字符，在数据通信传输中，接收站发给发送站的一种传输控制字符。它表示确认发来的数据已经接受无误。]) 消息响应。这样在客户机和服务器之间才能建立起可靠的TCP连接，数据才可以在客户机和服务器之间传递。

传了 SYN,为啥还要传 ACK

双方通信无误必须是两者互相发送信息都无误。传了 SYN，证明发送方到接收方的通道没有问题，但是接收方到发送方的通道还需要 ACK 信号来进行验证。



断开一个 TCP 连接则需要“四次挥手”：

- 客户端-发送一个 FIN, 用来关闭客户端到服务器的数据传送
- 服务器-收到这个 FIN, 它发回一个 ACK, 确认序号为收到的序号加1。和 SYN 一样，一个 FIN 将占用一个序号
- 服务器-关闭与客户端的连接，发送一个FIN给客户端
- 客户端-发回 ACK 报文确认，并将确认序号设置为收到序号加1

为什么要四次挥手

任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没有数据再发送的时候，则发出连接释放通知，对方确认后就完全关闭了TCP连接。

举个例子：A 和 B 打电话，通话即将结束后，A 说“我没啥要说的了”，B回答“我知道了”，但是 B 可能还会有要说的话，A 不能要求 B 跟着自己的节奏结束通话，于是 B 可能又巴拉巴拉说了一通，最后 B 说“我说完了”，A 回答“知道了”，这样通话才算结束。

上面讲的比较概括，推荐一篇讲的比较细致的文章：<https://blog.csdn.net/qzcsu/article/details/72861891>

3.1.2 TCP,UDP 协议的区别

类型	特点			性能		应用场景	首部字节
	是否面向连接	传输可靠性	传输形式	传输效率	所需资源		
TCP	面向连接	可靠	字节流	慢	多	要求通信数据可靠 (如文件传输、邮件传输)	20-60
UDP	无连接	不可靠	数据报文段	快	少	要求通信速度高 (如域名转换)	8个字节 (由4个字段组成)

UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如：QQ 语音、QQ 视频、直播等等

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP 不提供广播或多播服务。由于 TCP 要提供可靠的，面向连接的传输服务（TCP的可靠体现在TCP在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

3.1.3 TCP 协议如何保证可靠传输

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. 校验和：TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间，TCP的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送的速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）
6. 拥塞控制：当网络拥塞时，减少数据的发送。
7. ARQ协议：也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. 超时重传：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

3.1.4 ARQ协议

自动重传请求 (Automatic Repeat-reQuest, ARQ) 是OSI模型中数据链路层和传输层的错误纠正协议之一。它通过使用确认和超时这两个机制，在不可靠服务的基础上实现可靠的信息传输。如果发送方在发送后一段时间之内没有收到确认帧，它通常会重新发送。ARQ包括停止等待ARQ协议和连续ARQ协议。

停止等待ARQ协议

- 停止等待协议是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认（回复ACK）。如果过了一段时间（超时时间后），还是没有收到 ACK 确认，说明没有发送成功，需要重新发送，直到收到确认后再发下一个分组；
- 在停止等待协议中，若接收方收到重复分组，就丢弃该分组，但同时还要发送确认；

优点：简单

缺点：信道利用率低，等待时间长

1) 无差错情况：

发送方发送分组，接收方在规定时间内收到，并且回复确认。发送方再次发送。

2) 出现差错情况（超时重传）：

停止等待协议中超时重传是指只要超过一段时间仍然没有收到确认，就重传前面发送过的分组（认为刚才发送过的分组丢失了）。因此每发送完一个分组需要设置一个超时计时器，其重传时间应比数据在分组传输的平均往返时间更长一些。这种自动重传方式常称为 **自动重传请求 ARQ**。另外在停止等待协议中若收到重复分组，就丢弃该分组，但同时还要发送确认。**连续 ARQ 协议** 可提高信道利用率。发送维持一个发送窗口，凡位于发送窗口内的分组可连续发送出去，而不需要等待对方确认。接收方一般采用累积确认，对按序到达的最后一个分组发送确认，表明到这个分组位置的所有分组都已经正确收到了。

3) 确认丢失和确认迟到

- **确认丢失**：确认消息在传输过程丢失。当A发送M1消息，B收到后，B向A发送了一个M1确认消息，但却在传输过程中丢失。而A并不知道，在超时计时过后，A重传M1消息，B再次收到该消息后采取以下两点措施：1. 丢弃这个重复的M1消息，不向上层交付。2. 向A发送确认消息。（不会认为已经发送过了，就不再发送。A能重传，就证明B的确认消息丢失）。
- **确认迟到**：确认消息在传输过程中迟到。A发送M1消息，B收到并发送确认。在超时时间内没有收到确认消息，A重传M1消息，B仍然收到并继续发送确认消息（B收到了2份M1）。此时A收到了B第二次发送的确认消息。接着发送其他数据。过了一会，A收到了B第一次发送的对M1的确认消息（A也收到了2份确认消息）。处理如下：1. A收到重复的确认后，直接丢弃。2. B收到重复的M1后，也直接丢弃重复的M1。

连续ARQ协议

连续 ARQ 协议可提高信道利用率。发送方维持一个发送窗口，凡位于发送窗口内的分组可以连续发送出去，而不需要等待对方确认。接收方一般采用累计确认，对按序到达的最后一个分组发送确认，表明到这个分组为止的所有分组都已经正确收到了。

优点：信道利用率高，容易实现，即使确认丢失，也不必重传。

缺点：不能向发送方反映出接收方已经正确收到的所有分组的信息。比如：发送方发送了 5条 消息，中间第三条丢失（3号），这时接收方只能对前两个发送确认。发送方无法知道后三个分组的下落，而只好把后三个全部重传一次。这也叫 Go-Back-N（回退 N），表示需要退回来重传已经发送过的 N 个消息。

3.1.5 滑动窗口和流量控制

TCP 利用滑动窗口实现流量控制。流量控制是为了控制发送方发送速率，保证接收方来得及接收。接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

3.1.6 拥塞控制

在某段时间，若对网络中某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种情况就叫拥塞。拥塞控制就是为了防止过多的数据注入到网络中，这样就可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。拥塞控制是一个全局性的过程，涉及到所有的主机，所有的路由器，以及与降低网络传输性能有关的所有因素。相反，流量控制往往是点对点通信量的控制，是个端到端的问题。流量控制所要达到的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

为了进行拥塞控制，TCP 发送方要维持一个 **拥塞窗口(cwnd)** 的状态变量。拥塞控制窗口的大小取决于网络的拥塞程度，并且动态变化。发送方让自己的发送窗口取为拥塞窗口和接收方的接受窗口中较小的一个。

TCP的拥塞控制采用了四种算法，即 **慢开始**、**拥塞避免**、**快重传** 和 **快恢复**。在网络层也可以使路由器采用适当的分组丢弃策略（如主动队列管理 AQM），以减少网络拥塞的发生。

- **慢开始**：慢开始算法的思路是当主机开始发送数据时，如果立即把大量数据字节注入到网络，那么可能会引起网络阻塞，因为现在还不知道网络的符合情况。经验表明，较好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是由小到大逐渐增大拥塞窗口数值。cwnd初始值为1，每经过一个传播轮次，cwnd加倍。
- **拥塞避免**：拥塞避免算法的思路是让拥塞窗口cwnd缓慢增大，即每经过一个往返时间RTT就把发送放的cwnd加1。
- **快重传与快恢复**：在 TCP/IP 中，快速重传和恢复 (fast retransmit and recovery, FRR) 是一种拥塞控制算法，它能快速恢复丢失的数据包。没有 FRR，如果数据包丢失了，TCP 将会使用定时器来要求传输暂停。在暂停的这段时间内，没有新的或复制的数据包被发送。有了 FRR，如果接收机接收到一个不按顺序的数据段，它会立即给发送机发送一个重复确认。如果发送机接收到三个重复确认，它会假定确认件指出的数据段丢失了，并立即重传这些丢失的数据段。有了 FRR，就不会因为重传时要求的暂停被耽误。当有单独的数据包丢失时，快速重传和恢复 (FRR) 能最有效地工作。当有多个数据信息包在某一段很短的时间内丢失时，它则不能很有效地工作。

3.1.7 在浏览器中输入url地址 ->> 显示主页的过程(面试常客)

百度好像最喜欢问这个问题。

| 打开一个网页，整个过程会使用哪些协议

图解（图片来源：《图解HTTP》）：

过程	使用的协议
1. 浏览器查找域名的IP地址 (DNS查找过程: 浏览器缓存、路由器缓存、DNS 缓存)	DNS: 获取域名对应IP
2. 浏览器向web服务器发送一个HTTP请求 (cookies会随着请求发送给服务器)	
3. 服务器处理请求 (请求 处理请求 & 它的参数、cookies、生成一个HTML 响应)	<ul style="list-style-type: none"> TCP: 与服务器建立TCP连接 IP: 建立TCP协议时, 需要发送数据, 发送数据在网络层使用IP协议 OPSF: IP数据包在路由器之间, 路由选择使用OPSF协议 ARP: 路由器在与服务器通信时, 需要将ip地址转换为MAC地址, 需要使用ARP协议 HTTP: 在TCP建立完成后, 使用HTTP协议访问网页
4. 服务器发回一个HTML响应	
5. 浏览器开始显示HTML	

总体来说分为以下几个过程:

1. DNS解析
2. TCP连接
3. 发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 连接结束

具体可以参考下面这篇文章:

- <https://segmentfault.com/a/119000006879700>

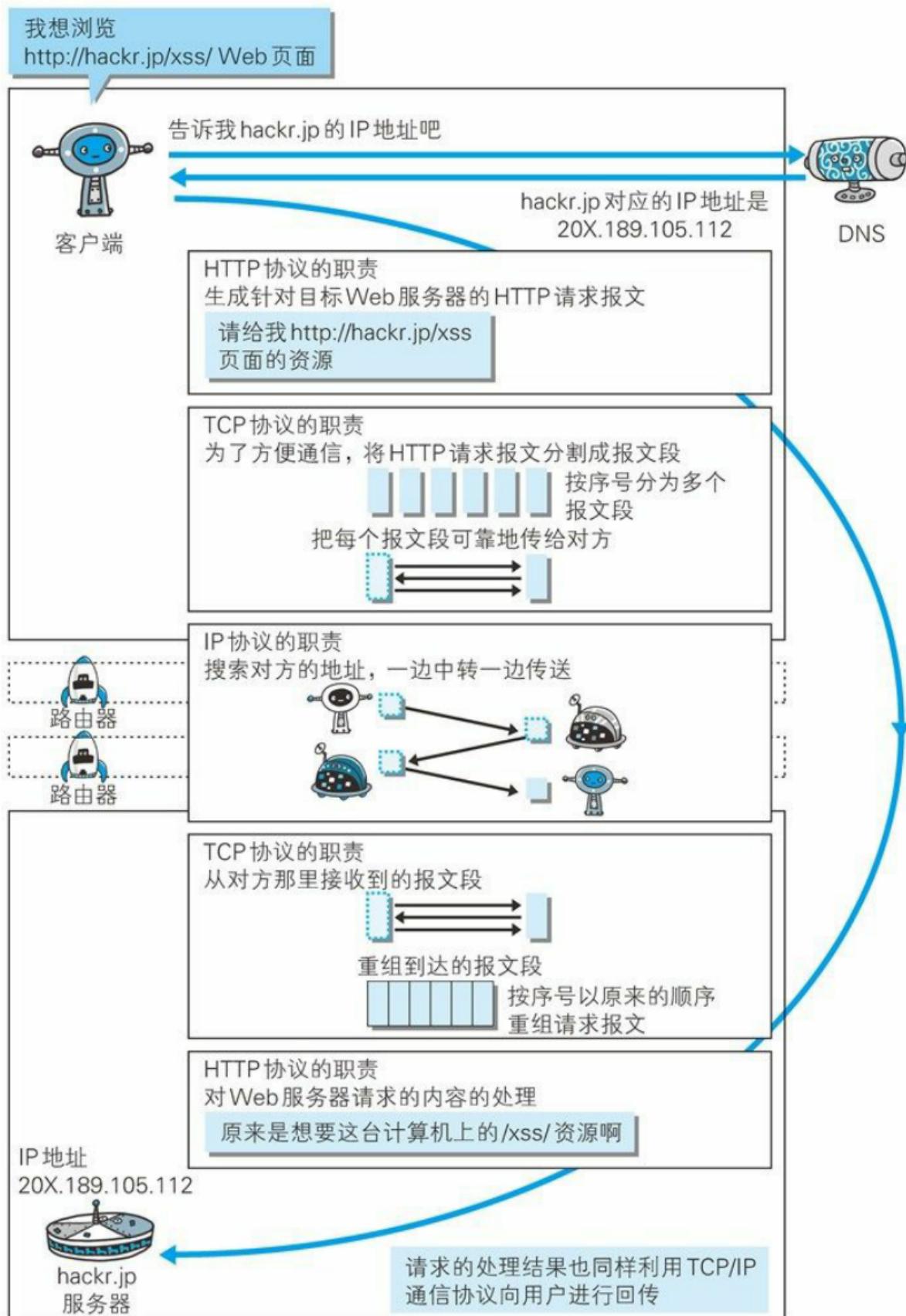
3.1.8 状态码

	类别	原因短语
1XX	Informational (信息性状态码)	接收的请求正在处理
2XX	Success (成功状态码)	请求正常处理完毕
3XX	Redirection (重定向状态码)	需要进行附加操作以完成请求
4XX	Client Error (客户端错误状态码)	服务器无法处理请求
5XX	Server Error (服务器错误状态码)	服务器处理请求出错

3.1.9 各种协议与HTTP协议之间的关系

一般面试官会通过这样的问题来考察你对计算机网络知识体系的理解。

图片来源：《图解HTTP》



3.1.10 HTTP长连接,短连接

在HTTP/1.0中默认使用短连接。也就是说，客户端和服务器每进行一次HTTP操作，就建立一次连接，任务结束就中断连接。当客户端浏览器访问的某个HTML或其他类型的Web页中包含有其他的Web资源（如JavaScript文件、图像文件、CSS文件等），每遇到这样一个Web资源，浏览器就会重新建立一个HTTP会话。

而从HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头加入这行代码：

```
Connection:keep-alive
```

在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，客户端再次访问这个服务器时，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接需要客户端和服务端都支持长连接。

HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

— 《HTTP长连接、短连接究竟是什么？》

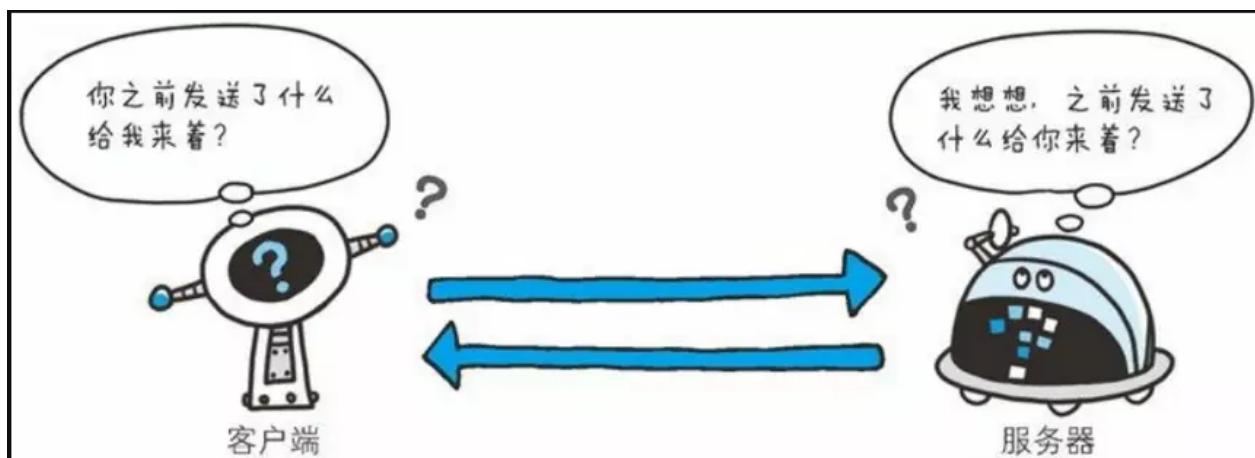
3.1.11 HTTP是不保存状态的协议,如何保存用户状态?

HTTP 是一种不保存状态，即无状态（stateless）协议。也就是说 HTTP 协议自身不对请求和响应之间的通信状态进行保存。那么我们保存用户状态呢？Session 机制的存在就是为了解决这个问题，Session 的主要作用就是通过服务端记录用户的状态。典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了（一般情况下，服务器会在一定时间内保存这个 Session，过了时间限制，就会销毁这个 Session）。

在服务端保存 Session 的方法很多，最常用的就是内存和数据库（比如是使用内存数据库redis保存）。既然 Session 存放在服务器端，那么我们如何实现 Session 跟踪呢？大部分情况下，我们都是通过在 Cookie 中附加一个 Session ID 来方式来跟踪。

Cookie 被禁用怎么办？

最常用的就是利用 URL 重写把 Session ID 直接附加在URL路径的后面。



3.1.12 Cookie的作用是什么?和Session有什么区别?

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

Cookie 一般用来保存用户信息 比如①我们在 Cookie 中保存已经登录过得用户信息，下次访问网站的时候页面可以自动帮你登录的一些基本信息给填了；②一般的网站都会有保持登录也就是说下次你再访问网站的时候就不需要重新登录了，这是因为用户登录的时候我们可以存放了一个 Token 在 Cookie 中，下次登录的时候只需要根据 Token 值来查找用户即可(为了安全考虑，重新登录一般要将 Token 重写)；③登录一次网站后访问网站其他页面不需要重新登录。**Session 的主要作用就是通过服务端记录用户的状态**。典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了。

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。

Cookie 存储在客户端中，而Session存储在服务器上，相对来说 Session 安全性更高。如果要在 Cookie 中存储一些敏感信息，不要直接写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

3.1.13 HTTP 1.0和HTTP 1.1的主要区别是什么？

| 这部分回答引用这篇文章 https://mp.weixin.qq.com/s/GICbiyJpINrHZ41u_4zT-A? 的一些内容。

HTTP1.0最早在网页中使用是在1996年，那个时候只是使用一些较为简单的网页上和网络请求上，而 HTTP1.1则在1999年才开始广泛应用于现在的各大浏览器网络请求中，同时HTTP1.1也是当前使用最为广泛的HTTP协议。 主要区别主要体现在：

1. **长连接**：在HTTP/1.0中，默认使用的是短连接，也就是说每次请求都要重新建立一次连接。HTTP 是基于TCP/IP协议的，每一次建立或者断开连接都需要三次握手四次挥手的开销，如果每次请求都要这样的话，开销会比较大。因此最好能维持一个长连接，可以用个长连接来发多个请求。HTTP 1.1起，默认使用长连接，默认开启Connection: keep-alive。HTTP/1.1的持续连接有非流水线方式和流水线方式。流水线方式是客户在收到HTTP的响应报文之前就能接着发送新的请求报文。与之相对应的非流水线方式是客户在收到前一个响应后才能发送下一个请求。
2. **错误状态响应码**：在HTTP1.1中新增了24个错误状态响应码，如409 (Conflict) 表示请求的资源与资源的当前状态发生冲突；410 (Gone) 表示服务器上的某个资源被永久性的删除。
3. **缓存处理**：在HTTP1.0中主要使用header里的If-Modified-Since, Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag, If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
4. **带宽优化及网络连接的使用**：HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206 (Partial Content)，这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3.1.12 URI和URL的区别是什么？

- URI(Uniform Resource Identifier) 是统一资源标志符，可以唯一标识一个资源。
- URL(Uniform Resource Location) 是统一资源定位符，可以提供该资源的路径。它是一种具体的 URI，即 URL 可以用来标识一个资源，而且还指明了如何 locate 这个资源。

URI的作用像身份证号一样，URL的作用更像家庭住址一样。URL是一种具体的URI，它不仅唯一标识资源，而且还提供了定位该资源的信息。

3.1.13 HTTP 和 HTTPS 的区别？

1. **端口**：HTTP的URL由“http://”起始且默认使用端口80，而HTTPS的URL由“https://”起始且默认使用端口443。

2. 安全性和资源消耗：HTTP协议运行在TCP之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS是运行在SSL/TLS之上的HTTP协议，SSL/TLS运行在TCP之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。所以说，HTTP安全性没有HTTPS高，但是HTTPS比HTTP耗费更多服务器资源。
- 对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有DES、AES等；
 - 非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有RSA、DSA等。

建议

非常推荐大家看一下《图解HTTP》这本书，这本书页数不多，但是内容很是充实，不管是用来系统的掌握网络方面的一些知识还是说纯粹为了应付面试都有很大帮助。下面的一些文章只是参考。大二学习这门课程的时候，我们使用的教材是《计算机网络第七版》（谢希仁编著），不推荐大家看这本教材，书非常厚而且知识偏理论，不确定大家能不能心平气和的读完。

参考

- https://blog.csdn.net/qq_16209077/article/details/52718250
- <https://blog.csdn.net/zixiaomuwu/article/details/60965466>
- https://blog.csdn.net/turn__back/article/details/73743641
- https://mp.weixin.qq.com/s/GICbiyJpINrHZ41u_4zT-A?

3.2 数据结构

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

下面只是简单地总结，给了一些参考文章，后面会对这部分内容进行重构。

欢迎大家自荐，数据结构这部分内容是很久之前写的，如果有老哥对这方面的总结不错的话，欢迎投稿联系。我的邮箱：koushuangbwcx@163.com。

3.2.1 Queue

什么是队列

队列是数据结构中比较重要的一种类型，它支持 FIFO，尾部添加、头部删除（先进队列的元素先出队列），跟我们生活中的排队类似。

队列的种类

- **单队列**（单队列就是常见的队列，每次添加元素时，都是添加到队尾，存在“假溢出”的问题也就是明明有位置却不能添加的情况）
- **循环队列**（避免了“假溢出”的问题）

Java 集合框架中的队列 Queue

Java 集合中的 Queue 继承自 Collection 接口，Deque, LinkedList, PriorityQueue, BlockingQueue 等类都实现了它。Queue 用来存放等待处理元素的集合，这种场景一般用于缓冲、并发访问。除了继承 Collection 接口的一些方法，Queue 还添加了额外的添加、删除、查询操作。

推荐文章

- [Java 集合深入理解（9）：Queue 队列](#)

3.2.2 Set

什么是 Set

Set 继承于 Collection 接口，是一个不允许出现重复元素，并且无序的集合，主要 HashSet 和 TreeSet 两大实现类。

在判断重复元素的时候，HashSet 集合会调用 hashCode() 和 equal() 方法来实现；TreeSet 集合会调用 compareTo 方法来实现。

补充：有序集合与无序集合说明

- 有序集合：集合里的元素可以根据 key 或 index 访问（List、Map）
- 无序集合：集合里的元素只能遍历。（Set）

HashSet 和 TreeSet 底层数据结构

HashSet 是哈希表结构，主要利用 HashMap 的 key 来存储元素，计算插入元素的 hashCode 来获取元素在集合中的位置；

TreeSet 是红黑树结构，每一个元素都是树中的一个节点，插入的元素都会进行排序；

推荐文章

- [Java集合--Set\(基础\)](#)

3.2.3 List

什么是List

在 List 中，用户可以精确控制列表中每个元素的插入位置，另外用户可以通过整数索引（列表中的位置）访问元素，并搜索列表中的元素。与 Set 不同，List 通常允许重复的元素。另外 List 是有序集合而 Set 是无序集合。

List的常见实现类

ArrayList 是一个数组队列，相当于动态数组。它由数组实现，随机访问效率高，随机插入、随机删除效率低。

LinkedList 是一个双向链表。它也可以被当作堆栈、队列或双端队列进行操作。LinkedList 随机访问效率低，但随机插入、随机删除效率高。

Vector 是矢量队列，和 ArrayList 一样，它也是一个动态数组，由数组实现。但是 ArrayList 是非线程安全的，而 Vector 是线程安全的。

Stack 是栈，它继承于 Vector。它的特性是：先进后出(FIFO, First In Last Out)。相关阅读：[java 数据结构与算法之栈（Stack）设计与实现](#)

ArrayList 和 LinkedList 源码学习

- [ArrayList 源码学习](#)
- [LinkedList 源码学习](#)

推荐阅读

- [java 数据结构与算法之顺序表与链表深入分析](#)

3.2.4 Map

- [集合框架源码学习之 HashMap\(JDK1.8\)](#)
- [ConcurrentHashMap 实现原理及源码分析](#)

3.2.5 树

1. 二叉树

[二叉树 \(百度百科\)](#)

- (1) **完全二叉树**—若设二叉树的高度为h，除第 h 层外，其它各层 (1~h-1) 的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。
- (2) **满二叉树**—除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。
- (3) **平衡二叉树**—平衡二叉树又被称为AVL树（区别于AVL算法），它是一棵二叉排序树，且具有以下性质：它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

2. 完全二叉树

[完全二叉树 \(百度百科\)](#)

完全二叉树：叶节点只能出现在最下层和次下层，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。

3. 满二叉树

[满二叉树 \(百度百科，国内外的定义不同\)](#)

国内教程定义：一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。也就是说，如果一个二叉树的层数为K，且结点总数是 $(2^k) - 1$ ，则它就是满二叉树。

4. 堆

[数据结构之堆的定义](#)

堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆；或者每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆。

5. 二叉查找树 (BST)

[浅谈算法和数据结构：七 二叉查找树](#)

二叉查找树的特点：

1. 若任意节点的左子树不空，则左子树上所有结点的 值均小于它的根结点的值；
2. 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
3. 任意节点的左、右子树也分别为二叉查找树；

4. 没有键值相等的节点 (no duplicate nodes) 。

6. 平衡二叉树 (Self-balancing binary search tree)

[平衡二叉树](#) (百度百科, 平衡二叉树的常用实现方法有红黑树、AVL、替罪羊树、Treap、伸展树等)

7. 红黑树

红黑树特点:

1. 每个节点非红即黑;
2. 根节点总是黑色的;
3. 每个叶子节点都是黑色的空节点 (NIL节点) ;
4. 如果节点是红色的, 则它的子节点必须是黑色的 (反之不一定) ;
5. 从根节点到叶节点或空子节点的每条路径, 必须包含相同数目的黑色节点 (即相同的黑色高度) 。

红黑树的应用:

[TreeMap](#)、[TreeSet](#)以及JDK1.8的[HashMap](#)底层都用到了红黑树。

为什么要用红黑树?

简单来说红黑树就是为了解决二叉查找树的缺陷, 因为二叉查找树在某些情况下会退化成一个线性结构。详细了解可以查看 [漫画: 什么是红黑树?](#) (也介绍到了二叉查找树, 非常推荐)

推荐文章:

- [漫画: 什么是红黑树?](#) (也介绍到了二叉查找树, 非常推荐)
- [寻找红黑树的操作手册](#) (文章排版以及思路真的不错)
- [红黑树深入剖析及Java实现](#) (美团点评技术团队)

8. B-, B+, B*树

[二叉树学习笔记之B树、B+树、B*树](#)

[《B-树, B+树, B*树详解》](#)

[《B-树, B+树与B*树的优缺点比较》](#)

B-树 (或B树) 是一种平衡的多路查找 (又称排序) 树, 在文件系统中有所应用。主要用作文件的索引。其中的B就表示平衡(Balance)

1. B+ 树的叶子节点链表结构相比于 B- 树便于扫库, 和范围检索。
2. B+树支持range-query (区间查询) 非常方便, 而B树不支持。这是数据库选用B+树的最主要原因之一。
3. B*树 是B+树的变体, B*树分配新结点的概率比B+树要低, 空间使用率更高;

9. LSM 树

[\[HBase\] LSM树 VS B+树](#)

B+树最大的性能问题是会产生大量的随机IO

为了克服B+树的弱点, HBase引入了LSM树的概念, 即Log-Structured Merge-Trees。

[LSM树由来、设计思想以及应用到HBase的索引](#)

3.2.6 图

3.2.7 BFS及DFS

- 《使用BFS及DFS遍历树和图的思路及实现》

3.3 算法

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

3.3.1 几道常见的字符串算法题总结

授权转载！

- 本文作者：[wwwxm](#)
- 原文地址：<https://www.weiweiblog.cn/13string/>

考虑到篇幅问题，我会分两次更新这个内容。本篇文章只是原文的一部分，我在原文的基础上增加了部分内容以及修改了部分代码和注释。另外，我增加了爱奇艺 2018 秋招 Java：求给定合法括号序列的深度 这道题。所有代码均编译成功，并带有注释，欢迎各位享用！

KMP 算法

谈到字符串问题，不得不提的就是 KMP 算法，它是用来解决字符串查找的问题，可以在一个字符串 (S) 中查找一个子串 (W) 出现的位置。KMP 算法把字符匹配的时间复杂度缩小到 $O(m+n)$ ，而空间复杂度也只有 $O(m)$ 。因为“暴力搜索”的方法会反复回溯主串，导致效率低下，而 KMP 算法可以利用已经部分匹配这个有效信息，保持主串上的指针不回溯，通过修改子串的指针，让模式串尽量地移动到有效的位置。

具体算法细节请参考：

- 字符串匹配的KMP算法：http://www.ruanyifeng.com/blog/2013/05/Knuth%20%93Morris%20%93Pratt_algorithm.html
- 从头到尾彻底理解KMP：https://blog.csdn.net/v_july_v/article/details/7041827
- 如何更好的理解和掌握 KMP 算法?：<https://www.zhihu.com/question/21923021>
- KMP 算法详细解析：<https://blog.sengxian.com/algorithms/kmp>
- 图解 KMP 算法：<http://blog.jobbole.com/76611/>
- 汪都能听懂的KMP字符串匹配算法【双语字幕】：<https://www.bilibili.com/video/av3246487/?from=search&seid=17173603269940723925>
- KMP字符串匹配算法1：<https://www.bilibili.com/video/av11866460?from=search&seid=12730654434238709250>

除此之外，再来了解一下BM算法！

BM算法也是一种精确字符串匹配算法，它采用从右向左比较的方法，同时应用到了两种启发式规则，即坏字符规则 和好后缀规则，来决定向右跳跃的距离。基本思路就是从右往左进行字符匹配，遇到不匹配的字符后从坏字符表和好后缀表找一个最大的右移值，将模式串右移继续匹配。

《字符串匹配的KMP算法》：http://www.ruanyifeng.com/blog/2013/05/Knuth%20%93Morris%20%93Pratt_algorithm.html

替换空格

剑指offer：请实现一个函数，将一个字符串中的每个空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

这里我提供了两种方法：①常规方法；②利用 API 解决。

```
//https://www.weiweiblog.cn/replacespace/
public class Solution {

    /**
     * 第一种方法：常规方法。利用String.charAt(i)以及
     * String.valueOf(char).equals(" ")
     * 遍历字符串并判断元素是否为空格。是则替换为"%20",否则不替换
     */
    public static String replaceSpace(StringBuffer str) {

        int length = str.length();
        // System.out.println("length=" + length);
        StringBuffer result = new StringBuffer();
        for (int i = 0; i < length; i++) {
            char b = str.charAt(i);
            if (String.valueOf(b).equals(" ")) {
                result.append("%20");
            } else {
                result.append(b);
            }
        }
        return result.toString();
    }

    /**
     * 第二种方法：利用API替换掉所用空格，一行代码解决问题
     */
    public static String replaceSpace2(StringBuffer str) {

        return str.toString().replaceAll("\\s", "%20");
    }
}
```

3.3.2 最长公共前缀

Leetcode：编写一个函数来查找字符串数组中的最长公共前缀。如果不存在公共前缀，返回空字符串 ""。

示例 1：

输入: ["flower", "flow", "flight"]
输出: "fl"

示例 2:

输入: ["dog", "racecar", "car"]
输出: ""
解释: 输入不存在公共前缀。

思路很简单! 先利用Arrays.sort(strs)为数组排序, 再将数组第一个元素和最后一个元素的字符从前往对比即可!

```
public class Main {  
    public static String replaceSpace(String[] strs) {  
  
        // 如果检查值不合法就返回空串  
        if (!checkStrs(strs)) {  
            return "";  
        }  
        // 数组长度  
        int len = strs.length;  
        // 用于保存结果  
        StringBuilder res = new StringBuilder();  
        // 给字符串数组的元素按照升序排序(包含数字的话, 数字会排在前面)  
        Arrays.sort(strs);  
        int m = strs[0].length();  
        int n = strs[len - 1].length();  
        int num = Math.min(m, n);  
        for (int i = 0; i < num; i++) {  
            if (strs[0].charAt(i) == strs[len - 1].charAt(i)) {  
                res.append(strs[0].charAt(i));  
            } else  
                break;  
        }  
        return res.toString();  
    }  
  
    private static boolean checkStrs(String[] strs) {  
        boolean flag = false;  
        if (strs != null) {  
            // 遍历strs检查元素值  
            for (int i = 0; i < strs.length; i++) {
```

```

        if (strs[i] != null && strs[i].length() != 0) {
            flag = true;
        } else {
            flag = false;
            break;
        }
    }
    return flag;
}

// 测试
public static void main(String[] args) {
    String[] strs = { "customer", "car", "cat" };
    // String[] strs = { "customer", "car", null }; // 空串
    // String[] strs = {};// 空串
    // String[] strs = null; // 空串
    System.out.println(Main.replaceSpace(strs)); // c
}
}

```

3.3.3 回文串

最长回文串

LeetCode：给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。在构造过程中，请注意区分大小写。比如“**Aa**”不能当做一个回文字符串。注意：假设字符串的长度不会超过 1010。

回文串：“回文串”是一个正读和反读都一样的字符串，比如“level”或者“noon”等等就是回文串。— 百度百科 地址：<https://baike.baidu.com/item/%E5%9B%9E%E6%96%87%E4%B8%B2/1274921?fr=aladdin>

示例 1：

输入：
"abccccdd"

输出：
7

解释：
我们可以构造的最长的回文串是"dccaccd"，它的长度是 7。

我们上面已经知道了什么是回文串？现在我们考虑一下可以构成回文串的两种情况：

- 字符出现次数为双数的组合
- 字符出现次数为双数的组合+一个只出现一次的字符

统计字符出现的次数即可，双数才能构成回文。因为允许中间一个数单独出现，比如“abcba”，所以如果最后有字母落单，总长度可以加 1。首先将字符串转变为字符数组。然后遍历该数组，判断对应字符是否在hashset中，如果不在就加进去，如果在就让count++，然后移除该字符！这样就能找到出现次数为双数的字符个数。

```
//https://leetcode-cn.com/problems/longest-palindrome/description/
class Solution {
    public int longestPalindrome(String s) {
        if (s.length() == 0)
            return 0;
        // 用于存放字符
        HashSet<Character> hashset = new HashSet<Character>();
        char[] chars = s.toCharArray();
        int count = 0;
        for (int i = 0; i < chars.length; i++) {
            if (!hashset.contains(chars[i])) {// 如果hashset没有该字符就保存进去
                hashset.add(chars[i]);
            } else {// 如果有，就让count++（说明找到了一个成对的字符），然后把该字符移除
                hashset.remove(chars[i]);
                count++;
            }
        }
        return hashset.isEmpty() ? count * 2 : count * 2 + 1;
    }
}
```

验证回文串

LeetCode：给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。说明：本题中，我们将空字符串定义为有效的回文串。

示例 1：

```
输入: "A man, a plan, a canal: Panama"
输出: true
```

示例 2：

输入: "race a car"

输出: false

```
//https://leetcode-cn.com/problems/valid-palindrome/description/
class Solution {
    public boolean isPalindrome(String s) {
        if (s.length() == 0)
            return true;
        int l = 0, r = s.length() - 1;
        while (l < r) {
            // 从头和尾开始向中间遍历
            if (!Character.isLetterOrDigit(s.charAt(l))) { // 字符不是字母和数
                l++;
            } else if (!Character.isLetterOrDigit(s.charAt(r))) { // 字符不是字
母和数字的情况
                r--;
            } else {
                // 判断二者是否相等
                if (Character.toLowerCase(s.charAt(l)) != Character.toLowerCase(s.charAt(r)))
                    return false;
                l++;
                r--;
            }
        }
        return true;
    }
}
```

最长回文子串

Leetcode: 最长回文子串 给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

示例 1:

输入: "babad"

输出: "bab"

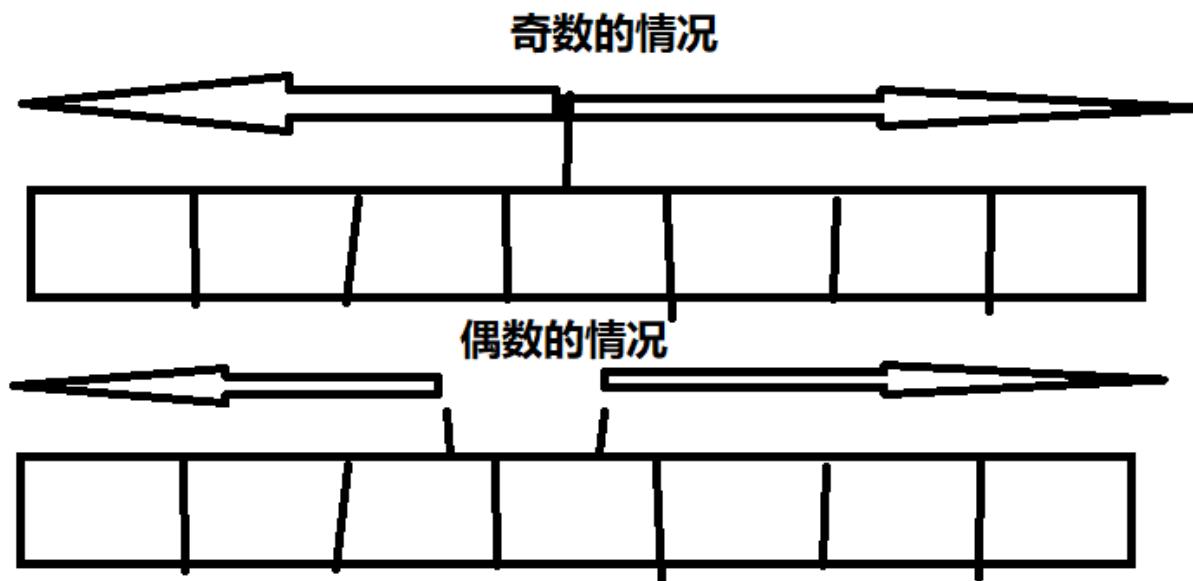
注意: "aba"也是一个有效答案。

示例 2:

输入: "cbbd"

输出: "bb"

以某个元素为中心，分别计算偶数长度的回文最大长度和奇数长度的回文最大长度。给大家大致花了个草图，不要嫌弃！



```
//https://leetcode-cn.com/problems/longest-palindromic-
substring/description/
class Solution {
    private int index, len;

    public String longestPalindrome(String s) {
        if (s.length() < 2)
            return s;
        for (int i = 0; i < s.length() - 1; i++) {
            PalindromeHelper(s, i, i);
            PalindromeHelper(s, i, i + 1);
        }
        return s.substring(index, index + len);
    }

    public void PalindromeHelper(String s, int l, int r) {
        while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
            l--;
            r++;
        }
        if (len < r - l - 1) {
            index = l + 1;
            len = r - l - 1;
        }
    }
}
```

```
    }
}
}
```

最长回文子序列

LeetCode: 最长回文子序列 给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。最长回文子序列和上一题最长回文子串的区别是，子串是字符串中连续的一个序列，而子序列是字符串中保持相对位置的字符序列，例如，“bbbb”可以是字符串“bbbab”的子序列但不是子串。

给定一个字符串s，找到其中最长的回文子序列。可以假设s的最大长度为1000。

示例 1：

```
输入：  
"bbbab"  
输出：  
4
```

一个可能的最长回文子序列为 “bbbb”。

示例 2：

```
输入：  
"cbbd"  
输出：  
2
```

一个可能的最长回文子序列为 “bb”。

动态规划： $dp[i][j] = dp[i+1][j-1] + 2$ if $s.charAt(i) = s.charAt(j)$ otherwise, $dp[i][j] = \max(dp[i+1][j], dp[i][j-1])$

```
class Solution {
    public int longestPalindromeSubseq(String s) {
        int len = s.length();
        int [][] dp = new int[len][len];
        for(int i = len - 1; i ≥ 0; i--){
            dp[i][i] = 1;
            for(int j = i+1; j < len; j++){
                if(s.charAt(i) == s.charAt(j))
                    dp[i][j] = dp[i+1][j-1] + 2;
                else
                    dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
            }
        }
    }
}
```

```
        return dp[0][len-1];
    }
}
```

括号匹配深度

爱奇艺 2018 秋招 Java：一个合法的括号匹配序列有以下定义：

1. 空串""是一个合法的括号匹配序列
2. 如果"X"和"Y"都是合法的括号匹配序列,"XY"也是一个合法的括号匹配序列
3. 如果"X"是一个合法的括号匹配序列,那么"(X)"也是一个合法的括号匹配序列
4. 每个合法的括号序列都可以由以上规则生成。

例如："","()", "()()", "((()))"都是合法的括号序列 对于一个合法的括号序列我们又有以下定义
它的深度：

1. 空串""的深度是0
2. 如果字符串"X"的深度是x,字符串"Y"的深度是y,那么字符串"XY"的深度为 $\max(x, y)$
3. 如果"X"的深度是x,那么字符串"(X)"的深度是x+1

例如："()()"的深度是1,"((()))"的深度是3。牛牛现在给你一个合法的括号序列,需要你计算出其深度。

输入描述：

输入包括一个合法的括号序列s,s长度 $length(2 \leq length \leq 50)$,序列中只包含'('和')'。

输出描述：

输出一个正整数,即这个序列的深度。

示例：

输入：

(())

输出：

2

思路草图：

从第一个字符开始向后遍历，碰到'('，count+1，否则count-1。用Max保存， $\text{max} = \text{Math.max}(\text{max}, \text{count})$ 。max是上次循环的保存的最大值。



((()))

代码如下：

```
import java.util.Scanner;

/**
 * https://www.nowcoder.com/test/8246651/summary
 *
 * @author Snailclimb
 * @date 2018年9月6日
 * @Description: TODO 求给定合法括号序列的深度
 */
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String s = sc.nextLine();
        int cnt = 0, max = 0, i;
        for (i = 0; i < s.length(); ++i) {
            if (s.charAt(i) == '(')
                cnt++;
            else
                cnt--;
            max = Math.max(max, cnt);
        }
        sc.close();
        System.out.println(max);
    }
}
```

把字符串转换成整数

剑指offer：将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函数。 数值为0或者字符串不是一个合法的数值则返回0。

```
//https://www.weiweiblog.cn/strtoint/
public class Main {

    public static int StrToInt(String str) {
        if (str.length() == 0)
            return 0;
        char[] chars = str.toCharArray();
        // 判断是否存在符号位
        int flag = 0;
        if (chars[0] == '+')
            flag = 1;
        else if (chars[0] == '-')
            flag = 2;
        int start = flag > 0 ? 1 : 0;
        int res = 0;// 保存结果
        for (int i = start; i < chars.length; i++) {
            if (Character.isDigit(chars[i])) {// 调用Character.isDigit(char)方法判断是否是数字，是返回True，否则False
                int temp = chars[i] - '0';
                res = res * 10 + temp;
            } else {
                return 0;
            }
        }
        return flag != 2 ? res : -res;
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String s = "-12312312";
        System.out.println("使用库函数转换: " + Integer.valueOf(s));
        int res = Main.StrToInt(s);
        System.out.println("使用自己写的方法转换: " + res);
    }
}
```

- 1. 两数相加
 - 题目描述
 - 问题分析
 - Solution
- 2. 翻转链表
 - 题目描述
 - 问题分析
 - Solution
- 3. 链表中倒数第k个节点
 - 题目描述
 - 问题分析
 - Solution
- 4. 删除链表的倒数第N个节点
 - 问题分析
 - Solution
- 5. 合并两个排序的链表
 - 题目描述
 - 问题分析
 - Solution

3.3.4 两数相加

题目描述

Leetcode: 给定两个非空链表来表示两个非负整数。位数按照逆序方式存储，它们的每个节点只存储单个数字。将两数相加返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

示例：

输入: (2 → 4 → 3) + (5 → 6 → 4)

输出: 7 → 0 → 8

原因: 342 + 465 = 807

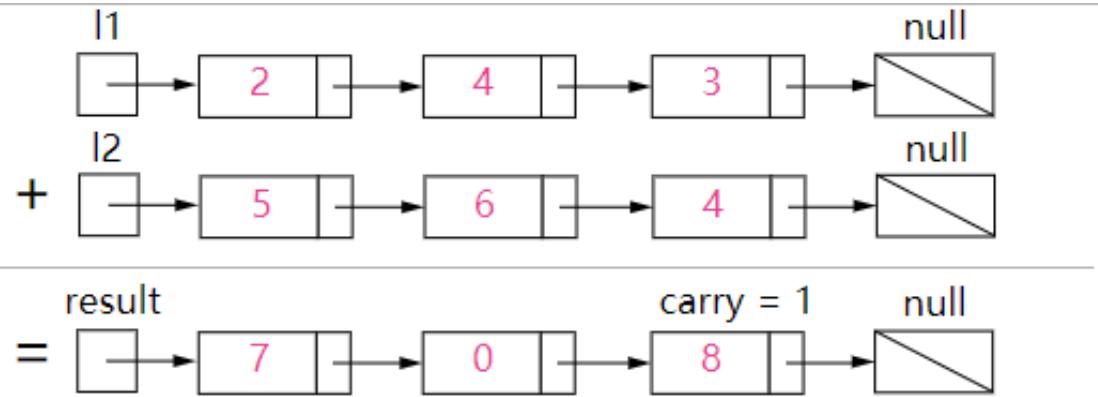
问题分析

Leetcode官方详细解答地址：

<https://leetcode-cn.com/problems/add-two-numbers/solution/>

要对头结点进行操作时，考虑创建哑节点dummy，使用`dummy→next`表示真正的头节点。这样可以避免处理头节点为空的边界问题。

我们使用变量来跟踪进位，并从包含最低有效位的表头开始模拟逐位相加的过程。



Solution

我们首先从最低有效位也就是列表 11和 12 的表头开始相加。注意需要考虑到进位的情况！

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
//https://leetcode-cn.com/problems/add-two-numbers/description/
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummyHead = new ListNode(0);
        ListNode p = l1, q = l2, curr = dummyHead;
        //carry 表示进位数
        int carry = 0;
        while (p != null || q != null) {
            int x = (p != null) ? p.val : 0;
            int y = (q != null) ? q.val : 0;
            int sum = carry + x + y;
            //进位数
            carry = sum / 10;
            //新节点的数值为sum % 10
            curr.next = new ListNode(sum % 10);
            curr = curr.next;
            if (p != null) p = p.next;
            if (q != null) q = q.next;
        }
        if (carry > 0) {
            curr.next = new ListNode(carry);
        }
        return dummyHead.next;
    }
}
```

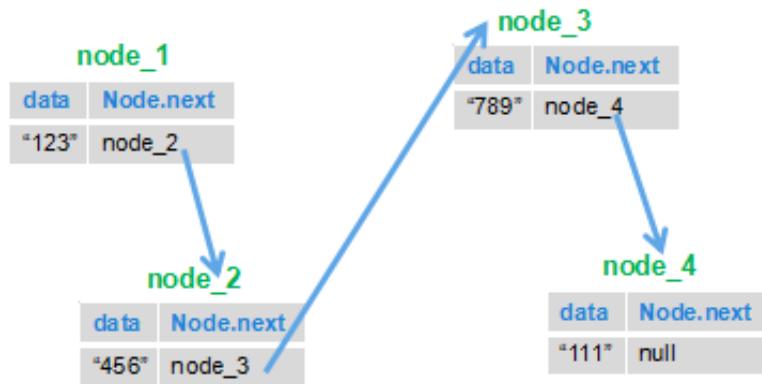
```
}
```

```
}
```

3.3.5 翻转链表

题目描述

剑指 offer: 输入一个链表，反转链表后，输出链表的所有元素。



问题分析

这道算法题，说直白点就是：如何让后一个节点指向前面一个节点！在下面的代码中定义了一个 next 节点，该节点主要是保存要反转到头的那个节点，防止链表“断裂”。

Solution

```
public class ListNode {  
    int val;  
    ListNode next = null;  
  
    ListNode(int val) {  
        this.val = val;  
    }  
}
```

```
/**  
 *  
 * @author Snailclimb  
 * @date 2018年9月19日  
 * @Description: TODO  
 */  
public class Solution {  
  
    public ListNode ReverseList(ListNode head) {
```

```

ListNode next = null;
ListNode pre = null;

while (head != null) {
    // 保存要反转到头的那个节点
    next = head.next;
    // 要反转的那个节点指向已经反转的上一个节点(备注:第一次反转的时候
    // 会指向null)
    head.next = pre;
    // 上一个已经反转到头部的节点
    pre = head;
    // 一直向链表尾走
    head = next;
}
return pre;
}

}

```

测试方法：

```

public static void main(String[] args) {

    ListNode a = new ListNode(1);
    ListNode b = new ListNode(2);
    ListNode c = new ListNode(3);
    ListNode d = new ListNode(4);
    ListNode e = new ListNode(5);
    a.next = b;
    b.next = c;
    c.next = d;
    d.next = e;
    new Solution().ReverseList(a);
    while (e != null) {
        System.out.println(e.val);
        e = e.next;
    }
}

```

输出：

```
5  
4  
3  
2  
1
```

3.3.6 链表中倒数第k个节点

题目描述

剑指offer：输入一个链表，输出该链表中倒数第k个结点。

问题分析

链表中倒数第k个节点也就是正数第($L-K+1$)个节点，知道了只一点，这一题基本就没问题！

首先两个节点/指针，一个节点 node1 先开始跑，指针 node1 跑到 $k-1$ 个节点后，另一个节点 node2 开始跑，当 node1 跑到最后时，node2 所指的节点就是倒数第k个节点也就是正数第($L-K+1$)个节点。

Solution

```
/*  
public class ListNode {  
    int val;  
    ListNode next = null;  
  
    ListNode(int val) {  
        this.val = val;  
    }  
}  
  
// 时间复杂度O(n),一次遍历即可  
// https://www.nowcoder.com/practice/529d3ae5a407492994ad2a246518148a?  
tpId=13&tqId=11167&tPage=1&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-  
interviews/question-ranking  
public class Solution {  
    public ListNode FindKthToTail(ListNode head, int k) {  
        // 如果链表为空或者k小于等于0  
        if (head == null || k <= 0) {  
            return null;  
        }  
        // 声明两个指向头结点的节点  
        ListNode node1 = head, node2 = head;  
        // 记录节点的个数  
        int count = 0;  
        // 记录k值，后面要使用
```

```

int index = k;
// p指针先跑，并且记录节点数，当node1节点跑了k-1个节点后，node2节点
开始跑,
// 当node1节点跑到最后时，node2节点所指的节点就是倒数第k个节点
while (node1 != null) {
    node1 = node1.next;
    count++;
    if (k < 1) {
        node2 = node2.next;
    }
    k--;
}
// 如果节点个数小于所求的倒数第k个节点，则返回空
if (count < index)
    return null;
return node2;
}
}

```

3.3.7 删除链表的倒数第N个节点

| Leetcode: 给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

给定一个链表：1→2→3→4→5，和 $n = 2$.

当删除了倒数第二个节点后，链表变为 1→2→3→5.

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

该题在 leetcode 上有详细解答，具体可参考 Leetcode.

问题分析

我们注意到这个问题可以容易地简化成另一个问题：删除从列表开头数起的第 $(L - n + 1)$ 个结点，其中 L 是列表的长度。只要我们找到列表的长度 L ，这个问题就很容易解决。



Solution

两次遍历法

首先我们将添加一个 **哑结点** 作为辅助，该结点位于列表头部。哑结点用来简化某些极端情况，例如列表中只含有一个结点，或需要删除列表的头部。在第一次遍历中，我们找出列表的长度 L 。然后设置一个指向哑结点的指针，并移动它遍历列表，直至它到达第 $(L - n)$ 个结点那里。我们把第 $(L - n)$ 个结点的 `next` 指针重新链接至第 $(L - n + 2)$ 个结点，完成这个算法。

```
/*
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
// https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/description/
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // 哑结点，哑结点用来简化某些极端情况，例如列表中只含有一个结点，或
        // 需要删除列表的头部
        ListNode dummy = new ListNode(0);
        // 哑结点指向头结点
        dummy.next = head;
        // 保存链表长度
        int length = 0;
        ListNode len = head;
        while (len != null) {
            length++;
            len = len.next;
        }
        length = length - n;
        ListNode target = dummy;
```

```

// 找到 L-n 位置的节点
while (length > 0) {
    target = target.next;
    length--;
}
// 把第 (L - n)个结点的 next 指针重新链接至第 (L - n + 2)个结点
target.next = target.next.next;
return dummy.next;
}
}

```

复杂度分析：

- **时间复杂度 $O(L)$** ：该算法对列表进行了两次遍历，首先计算了列表的长度 L 其次找到第 $(L - n)$ 个结点。操作执行了 $2L-n$ 步，时间复杂度为 $O(L)O(L)$ 。
- **空间复杂度 $O(1)$** ：我们只用了常量级的额外空间。

进阶——一次遍历法：

**链表中倒数第N个节点也就是正数第($L-N+1$)个节点。

其实这种方法就和我们上面第四题找“链表中倒数第k个节点”所用的思想是一样的。**基本思路就是：** 定义两个节点 node1、node2; node1 节点先跑，node1 节点跑到第 $n+1$ 个节点的时候，node2 节点开始跑。当 node1 节点跑到最后一个节点时，node2 节点所在的位置就是第 $(L-n)$ 个节点 (L 代表总链表长度，也就是倒数第 $n+1$ 个节点)

```

/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        // 声明两个指向头结点的节点
        ListNode node1 = dummy, node2 = dummy;

        // node1 节点先跑，node1 节点跑到第 n 个节点的时候，node2 节点开始跑
        // 当 node1 节点跑到最后一个节点时，node2 节点所在的位置就是第 (L-n)
        // 个节点，也就是倒数第 n+1 (L 代表总链表长度)
        while (node1 != null) {

```

```

        node1 = node1.next;
        if (n < 1 && node1 != null) {
            node2 = node2.next;
        }
        n--;
    }

    node2.next = node2.next.next;

    return dummy.next;
}

```

3.3.8 合并两个排序的链表

题目描述：

剑指offer:输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

问题分析：

我们可以这样分析：

1. 假设我们有两个链表 A,B;
2. A的头节点A1的值与B的头结点B1的值比较，假设A1小，则A1为头节点；
3. A2再和B1比较，假设B1小，则，A1指向B1；
4. A2再和B2比较 就这样循环往复就行了，应该还算好理解。

考虑通过递归的方式实现！

Solution:

递归版本：

```

/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}*/

```

```

//https://www.nowcoder.com/practice/d8b6b4358f774294a89de2a6ac4d9337?
tpId=13&tqId=11169&tPage=1&rp=1&ru=/ta/coding-interviews&qru=/ta/coding-
interviews/question-ranking
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        if(list1 == null){
            return list2;
        }
        if(list2 == null){
            return list1;
        }
        if(list1.val <= list2.val){
            list1.next = Merge(list1.next, list2);
            return list1;
        }else{
            list2.next = Merge(list1, list2.next);
            return list2;
        }
    }
}

```

3.3.9 剑指offer部分编程题

斐波那契数列

题目描述：

大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项。 $n \leq 39$

问题分析：

可以肯定的是这一题通过递归的方式是肯定能做出来，但是这样会有一个很大的问题，那就是递归大量的重复计算会导致内存溢出。另外可以使用迭代法，用fn1和fn2保存计算过程中的结果，并复用起来。下面我会把两个方法示例代码都给出来并给出两个方法的运行时间对比。

示例代码：

采用迭代法：

```

int Fibonacci(int number) {
    if (number <= 0) {
        return 0;
    }
    if (number == 1 || number == 2) {
        return 1;
    }
}

```

```

int first = 1, second = 1, third = 0;
for (int i = 3; i <= number; i++) {
    third = first + second;
    first = second;
    second = third;
}
return third;
}

```

采用递归：

```

public int Fibonacci(int n) {

    if (n <= 0) {
        return 0;
    }
    if (n == 1 || n == 2) {
        return 1;
    }

    return Fibonacci(n - 2) + Fibonacci(n - 1);

}

```

3.3.10 跳台阶问题

题目描述：

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

问题分析：

正常分析法： a.如果两种跳法，1阶或者2阶，那么假定第一次跳的是一阶，那么剩下的是n-1个台阶，跳法是f(n-1); b.假定第一次跳的是2阶，那么剩下的是n-2个台阶，跳法是f(n-2) c.由a, b假设可以得出总跳法为： $f(n) = f(n-1) + f(n-2)$ d.然后通过实际的情况可以得出：只有一阶的时候 $f(1) = 1$,只有两阶的时候可以有 $f(2) = 2$ 找规律分析法： $f(1) = 1, f(2) = 2, f(3) = 3, f(4) = 5,$ 可以总结出 $f(n) = f(n-1) + f(n-2)$ 的规律。但是为什么会出现这样的规律呢？假设现在6个台阶，我们可以从第5跳一步到6，这样的话有多少种方案跳到5就有多少种方案跳到6，另外我们也可以从4跳两步跳到6，跳到4有多少种方案的话，就有多少种方案跳到6，其他的不能从3跳到6什么的啦，所以最后就是 $f(6) = f(5) + f(4);$ 这样子也很好理解变态跳台阶的问题了。

所以这道题其实就是要解决斐波那契数列的问题。代码只需要在上一题的代码稍做修改即可。和上一题唯一不同的就是这一题的初始元素变为 1 2 3 5 8.....而上一题为1 1 2 3 5。另外这一题也可以用递归做，但是递归效率太低，所以我这里只给出了迭代方式的代码。

示例代码：

```

int jumpFloor(int number) {

```

```

if (number <= 0) {
    return 0;
}
if (number == 1) {
    return 1;
}
if (number == 2) {
    return 2;
}
int first = 1, second = 2, third = 0;
for (int i = 3; i <= number; i++) {
    third = first + second;
    first = second;
    second = third;
}
return third;
}

```

3.3.11 变态跳台阶问题

题目描述：

一只青蛙一次可以跳上1级台阶，也可以跳上2级.....它也可以跳上n级。求该青蛙跳上一个n级的台阶总共有多少种跳法。

问题分析：

假设 $n \geq 2$ ，第一步有 n 种跳法：跳1级、跳2级、到跳 n 级 跳1级，剩下 $n-1$ 级，则剩下跳法是 $f(n-1)$ 跳2级，剩下 $n-2$ 级，则剩下跳法是 $f(n-2)$ 跳 $n-1$ 级，剩下1级，则剩下跳法是 $f(1)$ 跳 n 级，剩下0级，则剩下跳法是 $f(0)$ 所以在 $n \geq 2$ 的情况下： $f(n)=f(n-1)+f(n-2)+\dots+f(1)$ 因为 $f(n-1)=f(n-2)+f(n-3)+\dots+f(1)$ 所以 $f(n)=2*f(n-1)$ 又 $f(1)=1$,所以可得 $f(n)=2^{(number-1)}$

示例代码：

```

int JumpFloorII(int number) {
    return 1 << --number; //2^(number-1)用位移操作进行，更快
}

```

补充：

java中有三种移位运算符：

1. “`<<`”：左移运算符，等同于乘2的 n 次方
2. “`>>`”：右移运算符，等同于除2的 n 次方
3. “`>>>`” 无符号右移运算符，不管移动前最高位是0还是1，右移后左侧产生的空位部分都以0来填充。与`>>`类似。例：`int a = 16; int b = a << 2; //左移2，等同于16 * 2的2次方，也就是16 * 4` `int c = a >> 2; //右移2，等同于16 / 2的2次方，也就是16 / 4`

3.3.12 二维数组查找

题目描述：

在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

问题解析：

这一道题还是比较简单的，我们需要考虑的是如何做，效率最快。这里有一种很好理解的思路：

矩阵是有序的，从左下角来看，向上数字递减，向右数字递增，因此从左下角开始查找，当要查找数字比左下角数字大时。右移 要查找数字比左下角数字小时，上移。这样找的速度最快。

示例代码：

```
public boolean Find(int target, int [][] array) {  
    //基本思路从左下角开始找，这样速度最快  
    int row = array.length-1;//行  
    int column = 0;//列  
    //当行数大于0，当前列数小于总列数时循环条件成立  
    while((row >= 0)&& (column< array[0].length)){  
        if(array[row][column] > target){  
            row--;  
        }else if(array[row][column] < target){  
            column++;  
        }else{  
            return true;  
        }  
    }  
    return false;  
}
```

3.3.13 替换空格

题目描述：

请实现一个函数，将一个字符串中的空格替换成“%20”。例如，当字符串为We Are Happy.则经过替换之后的字符串为We%20Are%20Happy。

问题分析：

这道题不难，我们可以通过循环判断字符串的字符是否为空格，是的话就利用append()方法添加追加“%20”，否则还是追加原字符。

或者最简单的方法就是利用： replaceAll(String regex, String replacement)方法了，一行代码就可以解决。

示例代码：

常规做法：

```

public String replaceSpace(StringBuffer str) {
    StringBuffer out=new StringBuffer();
    for (int i = 0; i < str.toString().length(); i++) {
        char b=str.charAt(i);
        if(String.valueOf(b).equals(" ")){
            out.append("%20");
        }else{
            out.append(b);
        }
    }
    return out.toString();
}

```

一行代码解决：

```

public String replaceSpace(StringBuffer str) {
    //return str.toString().replaceAll(" ", "%20");
    //public String replaceAll(String regex, String replacement)
    //用给定的替换替换与给定的regular expression匹配的此字符串的每个
    子字符串。
    //\ 转义字符. 如果你要使用 "\ 本身, 则应该使用 "\\ ". String类型
    中的空格用"\s"表示, 所以我这里猜测"\s"就是代表空格的意思
    return str.toString().replaceAll("\s", "%20");
}

```

3.3.14 数值的整数次方

题目描述：

给定一个double类型的浮点数base和int类型的整数exponent。求base的exponent次方。

问题解析：

这道题算是比较麻烦和难一点的一个了。我这里采用的是二分幂思想，当然也可以采用快速幂。更具剑指offer书中细节，该题的解题思路如下： 1.当底数为0且指数<0时，会出现对0求倒数的情况，需进行错误处理，设置一个全局变量； 2.判断底数是否等于0，由于base为double型，所以不能直接用==判断 3.优化求幂函数（二分幂）。当n为偶数， $a^n = (a^{n/2}) * (a^{n/2})$ ； 当n为奇数， $a^n = a^{[(n-1)/2]} * a^{[(n-1)/2]} * a$ 。时间复杂度 $O(\log n)$

时间复杂度： $O(\log n)$

示例代码：

```

public class Solution {
    boolean invalidInput=false;
    public double Power(double base, int exponent) {

```

```

//如果底数等于0并且指数小于0
//由于base为double型，不能直接用==判断
if(equal(base,0.0)&&exponent<0){
    invalidInput=true;
    return 0.0;
}
int absexponent=exponent;
//如果指数小于0，将指数转正
if(exponent<0)
    absexponent=-exponent;
//getPower方法求出base的exponent次方。
double res=getPower(base,absexponent);
//如果指数小于0，所得结果为上面求的结果的倒数
if(exponent<0)
    res=1.0/res;
return res;
}

//比较两个double型变量是否相等的方法
boolean equal(double num1,double num2){
    if(num1-num2>-0.000001&&num1-num2<0.000001)
        return true;
    else
        return false;
}

//求出b的e次方的方法
double getPower(double b,int e){
    //如果指数为0，返回1
    if(e==0)
        return 1.0;
    //如果指数为1，返回b
    if(e==1)
        return b;
    //e>>1相等于e/2，这里就是求 $a^n = (a^{n/2}) * (a^{n/2})$ 
    double result=getPower(b,e>>1);
    result*=result;
    //如果指数n为奇数，则要再乘一次底数base
    if((e&1)==1)
        result*=b;
    return result;
}
}

```

当然这一题也可以采用笨方法：累乘。不过这种方法的时间复杂度为 $O(n)$ ，这样没有前一种方法效率高。

```

// 使用累乘
public double powerAnother(double base, int exponent) {
    double result = 1.0;
    for (int i = 0; i < Math.abs(exponent); i++) {
        result *= base;
    }
    if (exponent ≥ 0)
        return result;
    else
        return 1 / result;
}

```

3.3.15 调整数组顺序使奇数位于偶数前面

题目描述：

输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于位于数组的后半部分，并保证奇数和奇数，偶数和偶数之间的相对位置不变。

问题解析：

这道题有挺多种解法的，给大家介绍一种我觉得挺好理解的方法：我们首先统计奇数的个数假设为n，然后新建一个等长数组，然后通过循环判断原数组中的元素为偶数还是奇数。如果是则从数组下标0的元素开始，把该奇数添加到新数组；如果是偶数则从数组下标为n的元素开始把该偶数添加到新数组中。

示例代码：

时间复杂度为O (n) ， 空间复杂度为O (n) 的算法

```

public class Solution {
    public void reOrderArray(int [] array) {
        //如果数组长度等于0或者等于1，什么都不做直接返回
        if(array.length==0||array.length==1)
            return;
        //oddCount: 保存奇数个数
        //oddBegin: 奇数从数组头部开始添加
        int oddCount=0,oddBegin=0;
        //新建一个数组
        int[] newArray=new int[array.length];
        //计算出（数组中的奇数个数）开始添加元素
        for(int i=0;i<array.length;i++){
            if((array[i]&1)==1) oddCount++;
        }
        for(int i=0;i<array.length;i++){
            //如果数为基数新数组从头开始添加元素
            //如果为偶数就从oddCount（数组中的奇数个数）开始添加元素

```

```

        if((array[i]&1)==1)
            newArray[oddBegin++]=array[i];
        else newArray[oddCount++]=array[i];
    }
    for(int i=0;i<array.length;i++){
        array[i]=newArray[i];
    }
}

```

3.3.16 链表中倒数第k个节点

题目描述：

输入一个链表，输出该链表中倒数第k个结点

问题分析：

一句话概括： 两个指针一个指针p1先开始跑，指针p1跑到k-1个节点后，另一个节点p2开始跑，当p1跑到最后时，p2所指的指针就是倒数第k个节点。

思想的简单理解： 前提假设：链表的结点个数(长度)为n。 规律一：要找到倒数第k个结点，需要向前走多少步呢？比如倒数第一个结点，需要走n步，那倒数第二个结点呢？很明显是向前走了n-1步，所以可以找到规律是找到倒数第k个结点，需要向前走n-k+1步。 算法开始：

1. 设两个都指向head的指针p1和p2，当p1走了k-1步的时候，停下来。p2之前一直不动。
2. p1的下一步是走第k步，这个时候，p2开始一起动了。至于为什么p2这个时候动呢？看下面的分析。
3. 当p1走到链表的尾部时，即p1走了n步。由于我们知道p2是在p1走了k-1步才开始动的，也就是说p1和p2永远差k-1步。所以当p1走了n步时，p2走的应该是在n-(k-1)步。即p2走了n-k+1步，此时巧妙的是p2正好指向的是规律一的倒数第k个结点处。这样是不是很好理解了呢？

考察内容：

链表+代码的鲁棒性

示例代码：

```

/*
//链表类
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}
*/

```

```

//时间复杂度O(n),一次遍历即可
public class Solution {
    public ListNode FindKthToTail(ListNode head, int k) {
        ListNode pre=null,p=null;
        //两个指针都指向头结点
        p=head;
        pre=head;
        //记录k值
        int a=k;
        //记录节点的个数
        int count=0;
        //p指针先跑，并且记录节点数，当p指针跑了k-1个节点后，pre指针开始
        跑,
        //当p指针跑到最后时，pre所指指针就是倒数第k个节点
        while(p!=null){
            p=p.next;
            count++;
            if(k<1){
                pre=pre.next;
            }
            k--;
        }
        //如果节点个数小于所求的倒数第k个节点，则返回空
        if(count<a) return null;
        return pre;
    }
}

```

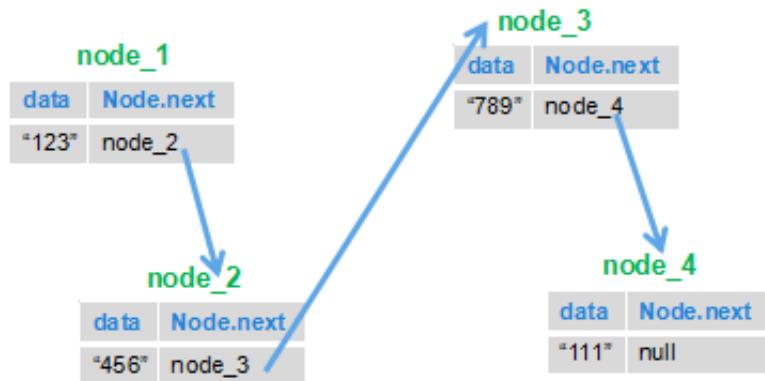
3.3.17 反转链表

题目描述：

输入一个链表，反转链表后，输出链表的所有元素。

问题分析：

链表的很常规的一道题，这一道题思路不算难，但自己实现起来真的可能会感觉无从下手，我是参考了别人的代码。思路就是我们根据链表的特点，前一个节点指向下一个节点的特点，把后面的节点移到前面来。就比如下图：我们把1节点和2节点互换位置，然后再将3节点指向2节点，4节点指向3节点，



这样以来下面的链表就被反转了。

考察内容：

链表+代码的鲁棒性

示例代码：

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
}/*
public class Solution {
public ListNode ReverseList(ListNode head) {
    ListNode next = null;
    ListNode pre = null;
    while (head != null) {
        //保存要反转到头来的那个节点
        next = head.next;
        //要反转的那个节点指向已经反转的上一个节点
        head.next = pre;
        //上一个已经反转到头部的节点
        pre = head;
        //一直向链表尾走
        head = next;
    }
    return pre;
}
```

```
}
```

3.3.18 合并两个排序的链表

题目描述：

输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

问题分析：

我们可以这样分析：

1. 假设我们有两个链表 A,B;
2. A的头节点A1的值与B的头结点B1的值比较，假设A1小，则A1为头节点；
3. A2再和B1比较，假设B1小，则，A1指向B1；
4. A2再和B2比较。.....就这样循环往复就行了，应该还算好理解。

考察内容：

链表+代码的鲁棒性

示例代码：

非递归版本：

```
/*
public class ListNode {
    int val;
    ListNode next = null;

    ListNode(int val) {
        this.val = val;
    }
} */
public class Solution {
    public ListNode Merge(ListNode list1,ListNode list2) {
        //list1为空，直接返回list2
        if(list1 == null){
            return list2;
        }
        //list2为空，直接返回list1
        if(list2 == null){
            return list1;
        }
        ListNode mergeHead = null;
        ListNode current = null;
        //当list1和list2不为空时
        while(list1!=null && list2!=null){
```

```

//取较小值作头结点
if(list1.val <= list2.val){
    if(mergeHead == null){
        mergeHead = current = list1;
    }else{
        current.next = list1;
        //current节点保存list1节点的值因为下一次还要用
        current = list1;
    }
    //list1指向下一个节点
    list1 = list1.next;
}else{
    if(mergeHead == null){
        mergeHead = current = list2;
    }else{
        current.next = list2;
        //current节点保存list2节点的值因为下一次还要用
        current = list2;
    }
    //list2指向下一个节点
    list2 = list2.next;
}
if(list1 == null){
    current.next = list2;
}else{
    current.next = list1;
}
return mergeHead;
}
}

```

递归版本：

```

public ListNode Merge(ListNode list1,ListNode list2) {
    if(list1 == null){
        return list2;
    }
    if(list2 == null){
        return list1;
    }
    if(list1.val <= list2.val){
        list1.next = Merge(list1.next, list2);
        return list1;
    }
}

```

```

    }else{
        list2.next = Merge(list1, list2.next);
        return list2;
    }
}

```

3.3.19 用两个栈实现队列

题目描述：

用两个栈来实现一个队列，完成队列的Push和Pop操作。队列中的元素为int类型。

问题分析：

先来回顾一下栈和队列的基本特点： 栈：后进先出（LIFO） 队列：先进先出 很明显我们需要根据JDK给我们提供的栈的一些基本方法来实现。先来看一下Stack类的一些基本方法：

Modifier and Type	Method and Description
boolean	<code>empty()</code> 测试此堆栈是否为空。
E	<code>peek()</code> 查看此堆栈顶部的对象，而不从堆栈中删除它。
E	<code>pop()</code> 删除此堆栈顶部的对象，并将该对象作为此函数的值返回。
E	<code>push(E item)</code> 将项目推送到此堆栈的顶部。
int	<code>search(Object o)</code> 返回一个对象在此堆栈上的基于1的位置。

既然题目给了我们两个栈，我们可以这样考虑当push的时候将元素push进stack1，pop的时候我们先把stack1的元素pop到stack2，然后再对stack2执行pop操作，这样就可以保证是先进先出的。（负[pop]负[pop]得正[先进先出]）

考察内容：

队列+栈

示例代码：

```

//左程云的《程序员代码面试指南》的答案
import java.util.Stack;

public class Solution {
    Stack<Integer> stack1 = new Stack<Integer>();
    Stack<Integer> stack2 = new Stack<Integer>();

    //当执行push操作时，将元素添加到stack1
    public void push(int node) {
        stack1.push(node);
    }
}

```

```

public int pop() {
    //如果两个队列都为空则抛出异常,说明用户没有push进任何元素
    if(stack1.empty()&&stack2.empty()){
        throw new RuntimeException("Queue is empty!");
    }
    //如果stack2不为空直接对stack2执行pop操作,
    if(stack2.empty()){
        while(!stack1.empty()){
            //将stack1的元素按后进先出push进stack2里面
            stack2.push(stack1.pop());
        }
    }
    return stack2.pop();
}
}

```

3.3.20 栈的压入,弹出序列

题目描述:

输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列1,2,3,4,5是某栈的压入顺序，序列4, 5,3,2,1是该压栈序列对应的一个弹出序列，但4,3,5,1,2就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）

题目分析:

这道题想了半天没有思路，参考了Alias的答案，他的思路写的也很详细应该很容易看懂。作者：
Alias <https://www.nowcoder.com/questionTerminal/d77d11405cc7470d82554cb392585106> 来源：牛客网

【思路】 借用一个辅助的栈，遍历压栈顺序，先讲第一个放入栈中，这里是1，然后判断栈顶元素是不是出栈顺序的第一个元素，这里是4，很显然 $1 \neq 4$ ，所以我们继续压栈，直到相等以后开始出栈，出栈一个元素，则将出栈顺序向后移动一位，直到不相等，这样循环等压栈顺序遍历完成，如果辅助栈还不为空，说明弹出序列不是该栈的弹出顺序。

举例：

入栈1,2,3,4,5

出栈4,5,3,2,1

首先1入辅助栈，此时栈顶 $1 \neq 4$ ，继续入栈2

此时栈顶 $2 \neq 4$ ，继续入栈3

此时栈顶 $3 \neq 4$ ，继续入栈4

此时栈顶 $4 = 4$ ，出栈4，弹出序列向后一位，此时为5，，辅助栈里面是1,2,3

此时栈顶 $3 \neq 5$ ，继续入栈5

此时栈顶5=5，出栈5，弹出序列向后一位，此时为3，，辅助栈里面是1,2,3

.... 依次执行，最后辅助栈为空。如果不为空说明弹出序列不是该栈的弹出顺序。

考察内容：

栈

示例代码：

```
import java.util.ArrayList;
import java.util.Stack;
//这道题没想出来，参考了Alias同学的答案：
https://www.nowcoder.com/questionTerminal/d77d11405cc7470d82554cb392585106
public class Solution {
    public boolean IsPopOrder(int[] pushA, int[] popA) {
        if(pushA.length == 0 || popA.length == 0)
            return false;
        Stack<Integer> s = new Stack<Integer>();
        //用于标识弹出序列的位置
        int popIndex = 0;
        for(int i = 0; i < pushA.length; i++) {
            s.push(pushA[i]);
            //如果栈不为空，且栈顶元素等于弹出序列
            while(!s.empty() && s.peek() == popA[popIndex]) {
                //出栈
                s.pop();
                //弹出序列向后一位
                popIndex++;
            }
        }
        return s.empty();
    }
}
```

3.4 操作系统

大家好，我是 Guide 哥！很多读者抱怨计算操作系统的知识点比较繁杂，自己也没有多少耐心去看，但是面试的时候又经常会遇到。所以，我带着我整理好的操作系统的常见问题来啦！这篇文章总结了一些我觉得比较重要的操作系统相关的问题比如**进程管理、内存管理、虚拟内存**等等。

文章形式通过大部分比较喜欢的面试官和求职者之间的对话形式展开。另外，Guide 哥也只是在大学的时候学习过操作系统，不过基本都忘了，为了写这篇文章这段时间看了很多相关的书籍和博客。如果文中有任何需要补充和完善的地方，你都可以在评论区指出。如果觉得内容不错的话，不要忘记点个好看哦！

我个人觉得学好操作系统还是非常有用的，具体可以看我昨天在星球分享的一段话：

对于为什么要了解操作系统的一点点看法：

操作系统中的很多思想、很多经典的算法，你都可以在我们日常开发使用的各种工具或者框架中找到它们的影子。

比如说我们开发的系统使用的缓存（比如 Redis）和操作系统的高速缓存就很像。CPU 中的高速缓存有很多种，不过大部分都是为了解决 CPU 处理速度和内存处理速度不对等的问题。我们还可以把内存看作外存的高速缓存，程序运行的时候我们把外存的数据复制到内存，由于内存的处理速度远远高于外存，这样提高了处理速度。同样地，我们使用的 Redis 缓存就是为了解决程序处理速度和访问常规关系型数据库速度不对等的问题。

高速缓存一般会按照局部性原理（2-8原则）根据相应的淘汰算法保证缓存中的数据是经常会被访问的。我们平常使用的 Redis 缓存很多时候也会按照2-8原则去做，很多淘汰算法都和操作系统中的类似。

既说了 2-8原则，那就不得不提命中率了，这是所有缓存概念都通用的。简单来说也就是你要访问的数据有多少能直接在缓存中直接找到。命中率高的话，一般表明你的缓存设计比较合理，系统处理速度也相对较快。

总结来说，我觉得学好操作系统能够提高自己思考的深度以及对技术的理解力。

等14人赞过

这篇文章只是对一些操作系统比较重要概念的一个概览，深入学习的话，建议大家还是老老实实地去看书。另外，这篇文章的很多内容参考了《现代操作系统》第三版这本书，非常感谢。

一 操作系统基础

面试官顶着蓬松的假发向我走来，只见他一手拿着厚重的 Thinkpad，一手提着他那淡黄的长裙。



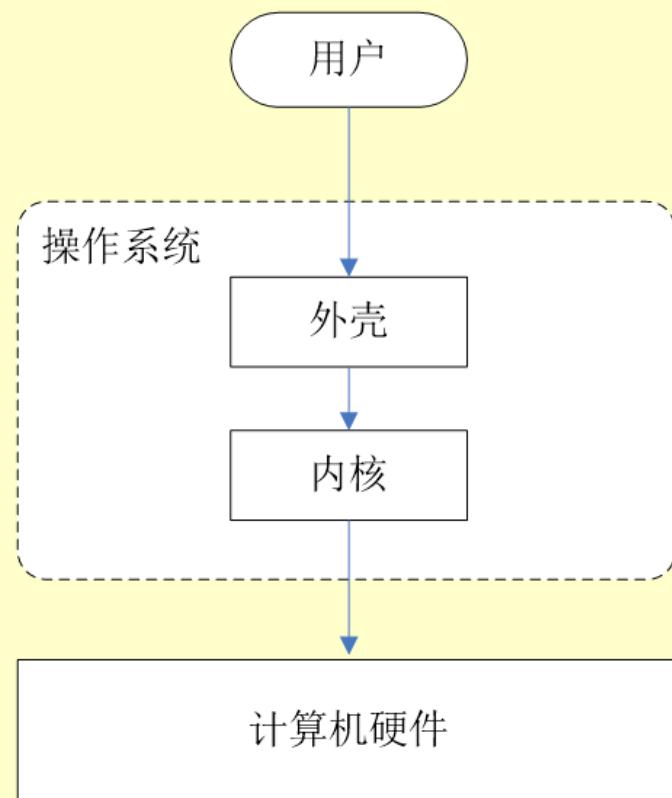
1.1 什么是操作系统？

面试官：先来个简单问题吧！什么是操作系统？

我：我通过以下四点向您介绍一下什么是操作系统吧！

1. 操作系统（Operating System，简称 OS）是管理计算机硬件与软件资源的程序，是计算机系统的内核与基石；
2. 操作系统本质上是运行在计算机上的软件程序；
3. 操作系统为用户提供一个与系统交互的操作界面；
4. 操作系统分内核与外壳（我们可以把外壳理解成围绕着内核的应用程序，而内核就是能操作硬件的程序）。

关于内核多插一嘴：内核负责管理系统的进程、内存、设备驱动程序、文件和网络系统等等，决定着系统的性能和稳定性。是连接应用程序和硬件的桥梁。内核就是操作系统背后黑盒的核心。



1.2 系统调用

面试官：什么是系统调用呢？能不能详细介绍一下。

我：介绍系统调用之前，我们先来了解一下用户态和系统态。



根据进程访问资源的特点，我们可以把进程在系统上的运行分为两个级别：

1. 用户态(user mode)：用户态运行的进程或可以直接读取用户程序的数据。
2. 系统态(kernel mode):可以简单的理解系统态运行的进程或程序几乎可以访问计算机的任何资源，不受限制。

说了用户态和系统态之后，那么什么是系统调用呢？

我们运行的程序基本都是运行在用户态，如果我们调用操作系统提供的系统态级别的子功能咋办呢？那就需要系统调用了！

也就是说在我们运行的用户程序中，凡是与系统态级别的资源有关的操作（如文件管理、进程控制、内存管理等），都必须通过系统调用方式向操作系统提出服务请求，并由操作系统代为完成。

这些系统调用按功能大致可分为如下几类：

- 设备管理。完成设备的请求或释放，以及设备启动等功能。
- 文件管理。完成文件的读、写、创建及删除等功能。
- 进程控制。完成进程的创建、撤销、阻塞及唤醒等功能。
- 进程通信。完成进程之间的消息传递或信号传递等功能。
- 内存管理。完成内存的分配、回收以及获取作业占用内存区大小及地址等功能。

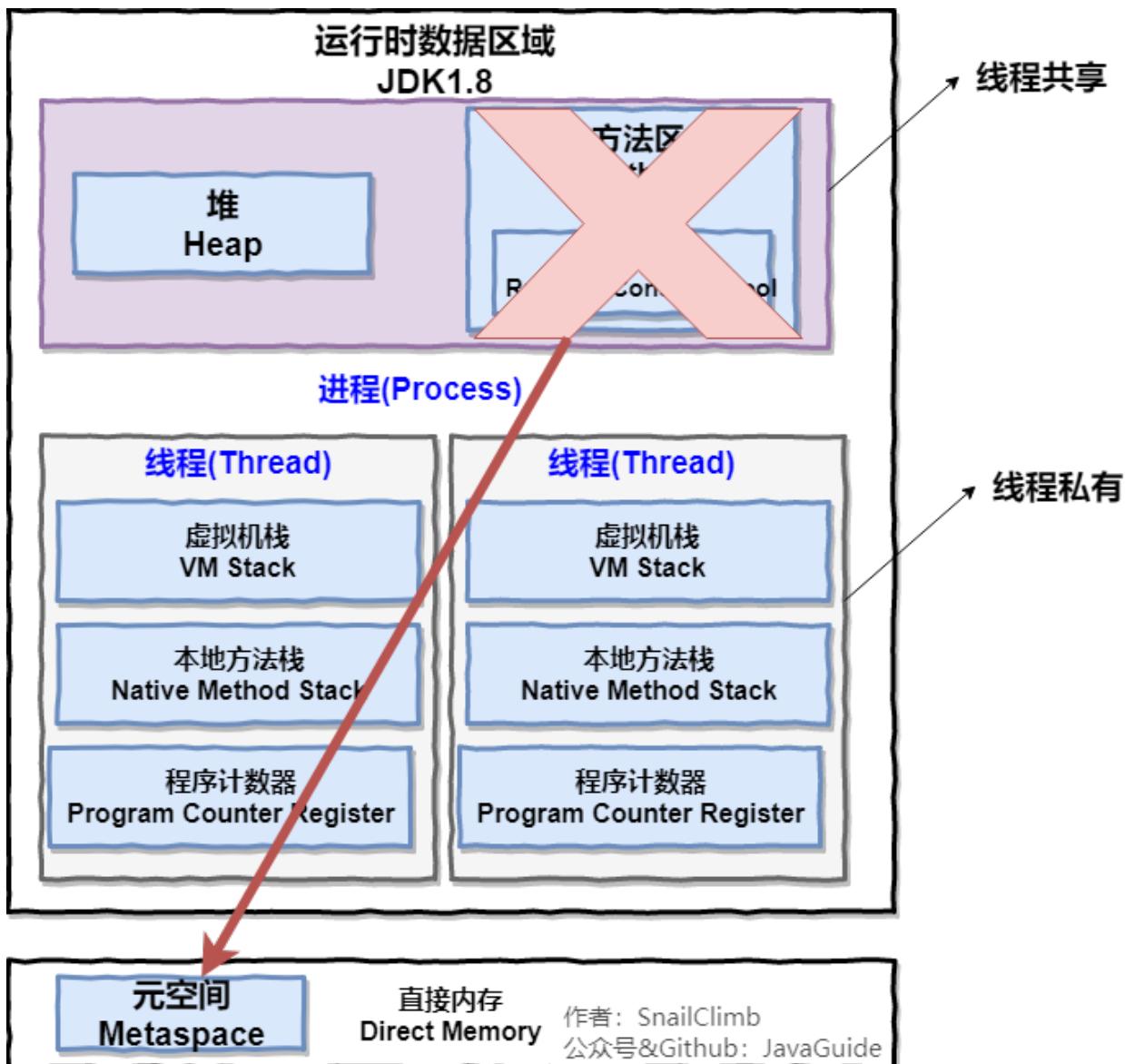
二 进程和线程

2.1 进程和线程的区别

面试官：好的！我明白了！那你再说一下：进程和线程的区别。

我：好的！下图是 Java 内存区域，我们从 JVM 的角度来说一下线程和进程之间的关系吧！

如果你对 Java 内存区域（运行时数据区）这部分知识不太了解的话可以阅读一下这篇文章：[《可能是把 Java 内存区域讲的最清楚的一篇文章》](#)



从上图可以看出：一个进程中可以有多个线程，多个线程共享进程的堆和方法区（JDK1.8之后的元空间）资源，但是每个线程有自己的程序计数器、虚拟机栈 和本地方法栈。

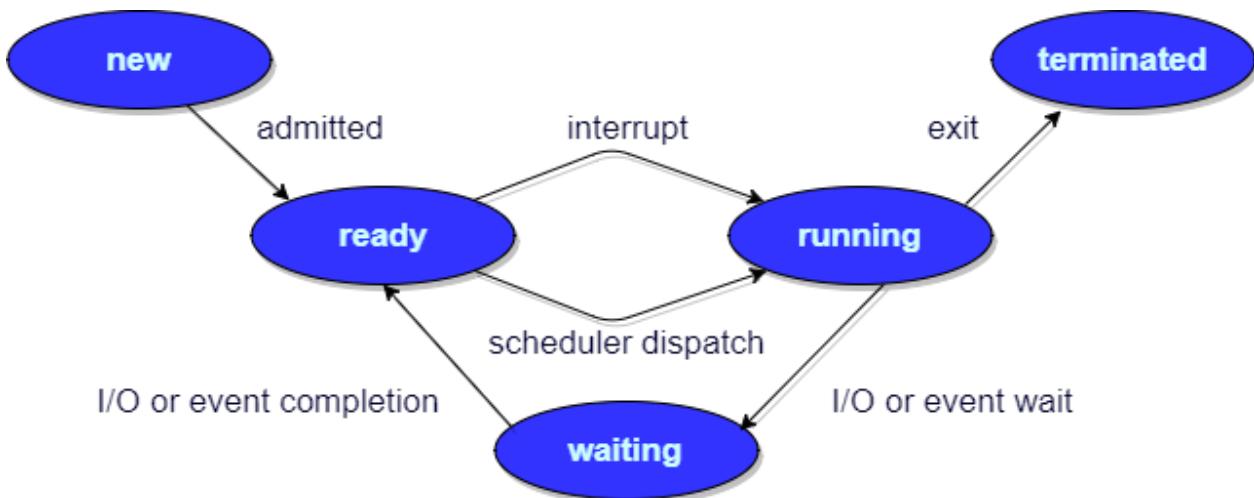
总结：线程是进程划分成的更小的运行单位，一个进程在其执行的过程中可以产生多个线程。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。线程执行开销小，但不利于资源的管理和保护；而进程正相反。

2.2 进程有哪几种状态？

面试官： 那你再说说进程有哪几种状态？

我： 我们一般把进程大致分为 5 种状态，这一点和线程很像！

- **创建状态(new)**：进程正在被创建，尚未到就绪状态。
- **就绪状态(ready)**：进程已处于准备运行状态，即进程获得了除了处理器之外的一切所需资源，一旦得到处理器资源（处理器分配的时间片）即可运行。
- **运行状态(running)**：进程正在处理器上运行（单核 CPU 下任意时刻只有一个进程处于运行状态）。
- **阻塞状态(waiting)**：又称为等待状态，进程正在等待某一事件而暂停运行如等待某资源为可用或等待 I/O 操作完成。即使处理器空闲，该进程也不能运行。
- **结束状态(terminated)**：进程正在从系统中消失。可能是进程正常结束或其他原因中断退出运行。



2.3 进程间的通信方式

面试官： 进程间的通信常见的的有哪几种方式呢？

我： 大概有 7 种常见的进程间的通信方式。

下面这部分总结参考了：[《进程间通信 IPC \(InterProcess Communication\)》](#) 这篇文章，推荐阅读，总结的非常不错。

1. **管道/匿名管道(Pipes)**：用于具有亲缘关系的父子进程间或者兄弟进程之间的通信。
2. **有名管道(Named Pipes)**：匿名管道由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道。有名管道严格遵循先进先出(first in first out)。有名管道以磁盘文件的方式存在，可以实现本机任意两个进程通信。
3. **信号(Signal)**：信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生；
4. **消息队列(Message Queuing)**：消息队列是消息的链表，具有特定的格式，存放在内存中并由消息队列标识符标识。管道和消息队列的通信数据都是先进先出的原则。与管道（无名管道：只存在于内存中的文件；命名管道：存在于实际的磁盘介质或者文件系统）不同的是消息队列存放在内核中，只有在内核重启（即，操作系统重启）或者显示地删除一个消息队列时，该消息队列才会被

真正的删除。消息队列可以实现消息的随机查询，消息不一定要以先进先出的次序读取，也可以按消息的类型读取。比 FIFO 更有优势。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5. **信号量(Semaphores)**：信号量是一个计数器，用于多进程对共享数据的访问，信号量的意图在于进程间同步。这种通信方式主要用于解决与同步相关的问题并避免竞争条件。
6. **共享内存(Shared memory)**：使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据的更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。可以说这是最有用的进程间通信方式。
7. **套接字(Sockets)**：此方法主要用于在客户端和服务器之间通过网络进行通信。套接字是支持 TCP/IP 的网络通信的基本操作单元，可以看做是不同主机之间的进程进行双向通信的端点，简单的说就是通信的两方的一种约定，用套接字中的相关函数来完成通信过程。

2.4 线程间的同步的方式

 面试官：那线程间的同步的方式有哪些呢？

 我：线程同步是两个或多个共享关键资源的线程的并发执行。应该同步线程以避免关键的资源使用冲突。操作系统一般有下面三种线程同步的方式：

1. **互斥量(Mutex)**：采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。比如 Java 中的 synchronized 关键词和各种 Lock 都是这种机制。
2. **信号量(Semaphores)**：它允许同一时刻多个线程访问同一资源，但是需要控制同一时刻访问此资源的最大线程数量
3. **事件(Event)**：Wait/Notify：通过通知操作的方式来保持多线程同步，还可以方便的实现多线程优先级的比较操作

2.5 进程的调度算法

 面试官：你知道操作系统中进程的调度算法有哪些吗？

 我：嗯嗯！这个我们大学的时候学过，是一个很重要的知识点！

为了确定首先执行哪个进程以及最后执行哪个进程以实现最大 CPU 利用率，计算机科学家已经定义了一些算法，它们是：

- **先到先服务(FCFS)调度算法**：从就绪队列中选择一个最先进入该队列的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
- **短作业优先(SJF)的调度算法**：从就绪队列中选出一个估计运行时间最短的进程为之分配资源，使它立即执行并一直执行到完成或发生某事件而被阻塞放弃占用 CPU 时再重新调度。
- **时间片轮转调度算法**：时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法，又称 RR(Round robin)调度。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。
- **多级反馈队列调度算法**：前面介绍的几种进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程。多级反馈队列调度算法既能使高优先级的作业得到响应又能使短作业（进程）迅速完成。因而它是目前被公认的一种较好的进程调度算法，UNIX 操作系统采取的便是这种调度算法。
- **优先级调度**：为每个流程分配优先级，首先执行具有最高优先级的进程，依此类推。具有相同优先级的进程以 FCFS 方式执行。可以根据内存要求，时间要求或任何其他资源要求来确定优先级。

三 操作系统内存管理基础

3.1 内存管理介绍

 面试官：操作系统的内存管理主要是做什么？

 我：操作系统的内存管理主要负责内存的分配与回收（`malloc` 函数：申请内存，`free` 函数：释放内存），另外地址转换也就是将逻辑地址转换成相应的物理地址等功能也是操作系统内存管理做的事情。

3.2 常见的几种内存管理机制

 面试官：操作系统的内存管理机制了解吗？内存管理有哪几种方式？

 我：这个在学习操作系统的时候有了解过。

简单分为**连续分配管理方式**和**非连续分配管理方式**这两种。连续分配管理方式是指为一个用户程序分配一个连续的内存空间，常见的如**块式管理**。同样地，非连续分配管理方式允许一个程序使用的内存分布在离散或者说不相邻的内存中，常见的如**页式管理**和**段式管理**。

1. **块式管理**：远古时代的计算机操作系统的内存管理方式。将内存分为几个固定大小的块，每个块中只包含一个进程。如果程序运行需要内存的话，操作系统就分配给它一块，如果程序运行只需要很小的空间的话，分配的这块内存很大一部分几乎被浪费了。这些在每个块中未被利用的空间，我们称之为碎片。
2. **页式管理**：把主存分为大小相等且固定的一页一页的形式，页较小，相对相比于块式管理的划分力度更大，提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址。
3. **段式管理**：页式管理虽然提高了内存利用率，但是页式管理其中的页实际并无任何实际意义。段式管理把主存分为一段段的，每一段的空间又要比一页的空间小很多。但是，最重要的是段是有实际意义的，每个段定义了一组逻辑信息，例如，有主程序段 MAIN、子程序段 X、数据段 D 及栈段 S 等。段式管理通过段表对应逻辑地址和物理地址。

 面试官：回答的还不错！不过漏掉了一个很重要的**段页式管理机制**。段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说**段页式管理机制**中段与段之间以及段的内部的都是离散的。

 我：谢谢面试官！刚刚把这个给忘记了～



3.3 快表和多级页表

 面试官：页表管理机制中有两个很重要的概念：快表和多级页表，这两个东西分别解决了页表管理中很重要的两个问题。你给我简单介绍一下吧！

 我：在分页内存管理中，很重要的两点是：

1. 虚拟地址到物理地址的转换要快。
2. 解决虚拟地址空间大，页表也会很大的问题。

快表

为了解决虚拟地址到物理地址的转换速度，操作系统在 **页表方案** 基础之上引入了 **快表** 来加速虚拟地址到物理地址的转换。我们可以把块表理解为一种特殊的高速缓冲存储器（Cache），其中的内容是页表的一部分或者全部内容。作为页表的 Cache，它的作用与页表相似，但是提高了访问速率。由于采用页表做地址转换，读写内存数据时 CPU 要访问两次主存。有了快表，有时只要访问一次高速缓冲存储器，一次主存，这样可加速查找并提高指令执行速度。

使用快表之后的地址转换流程是这样的：

1. 根据虚拟地址中的页号查快表；
2. 如果该页在快表中，直接从快表中读取相应的物理地址；
3. 如果该页不在快表中，就访问内存中的页表，再从页表中得到物理地址，同时将页表中的该映射表项添加到快表中；
4. 当快表填满后，又要登记新页时，就按照一定的淘汰策略淘汰掉快表中的一个页。

看完了之后你会发现快表和我们平时经常在我们开发的系统使用的缓存（比如 Redis）很像，的确是这样的，操作系统中的很多思想、很多经典的算法，你都可以在我们日常开发使用的各种工具或者框架中找到它们的影子。

多级页表

引入多级页表的主要目的是为了避免把全部页表一直放在内存中占用过多空间，特别是那些根本就不需要的页表就不需要保留在内存中。多级页表属于时间换空间的典型场景，具体可以查看下面这篇文章

- 多级页表如何节约内存：<https://www.polarxiong.com/archives/多级页表如何节约内存.html>

总结

为了提高内存的空间性能，提出了多级页表的概念；但是提到空间性能是以浪费时间性能为基础的，因此为了补充损失的时间性能，提出了快表（即 TLB）的概念。不论是快表还是多级页表实际上都利用到了程序的局部性原理，局部性原理在后面的虚拟内存这部分会介绍到。

3.4 分页机制和分段机制的共同点和区别

面试官：分页机制和分段机制有哪些共同点和区别呢？

我：



小朋友 你是否有很多问号

1. 共同点：

- 分页机制和分段机制都是为了提高内存利用率，较少内存碎片。
- 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是，每个页和段中的内存是连续的。

2. 区别：

- 页的大小是固定的，由操作系统决定；而段的大小不固定，取决于我们当前运行的程序。
- 分页仅仅是为了满足操作系统内存管理的需求，而段是逻辑信息的单位，在程序中可以体现为代码段，数据段，能够更好满足用户的需要。

3.5 逻辑(虚拟)地址和物理地址

面试官：你刚刚还提到了逻辑地址和物理地址这两个概念，我不太清楚，你能为我解释一下不？

我：em...好的嘛！我们编程一般只可能和逻辑地址打交道，比如在 C 语言中，指针里面存储的数值就可以理解成为内存里的一个地址，这个地址也就是我们说的逻辑地址，逻辑地址由操作系统决定。物理地址指的是真实物理内存中地址，更具体一点来说就是内存地址寄存器中的地址。物理地址是内存单元真正的地址。

3.6 CPU 寻址了解吗？为什么需要虚拟地址空间？

面试官：CPU 寻址了解吗？为什么需要虚拟地址空间？

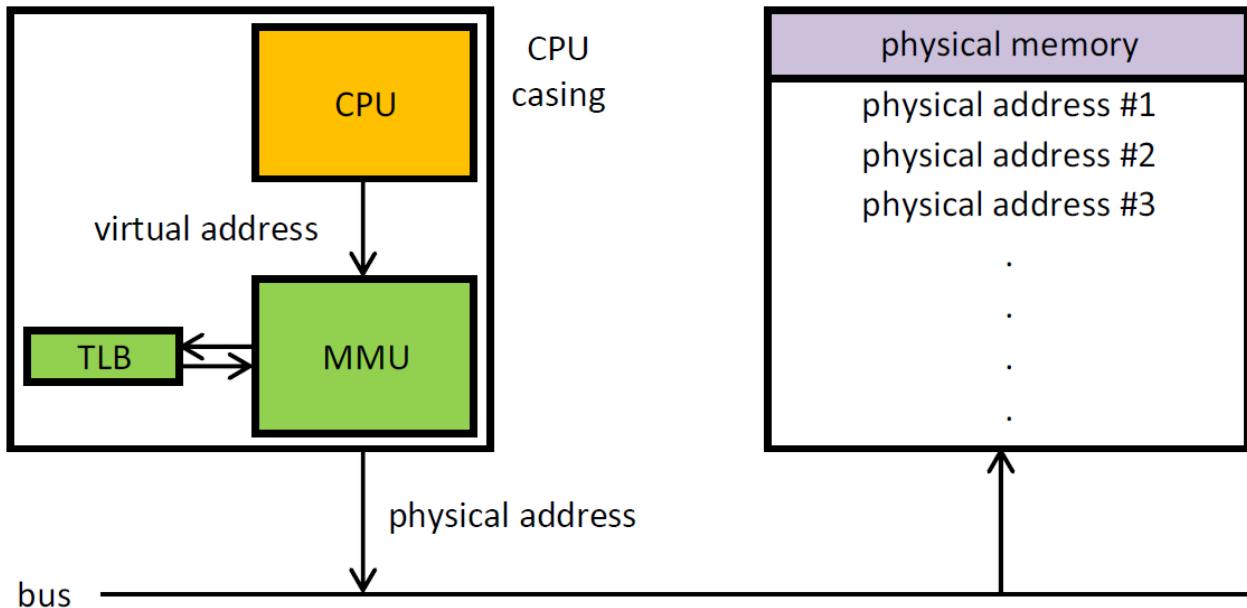
我：这部分我真不清楚！



于是面试完之后我默默去查阅了相关文档！留下了没有技术的泪水。。。

这部分内容参考了 Microsoft 官网的介绍，地址：[https://msdn.microsoft.com/zh-cn/library/windows/hardware/hh439648\(v=vs.85\).aspx](https://msdn.microsoft.com/zh-cn/library/windows/hardware/hh439648(v=vs.85).aspx)

现代处理器使用的是一种称为 **虚拟寻址(Virtual Addressing)** 的寻址方式。使用虚拟寻址，CPU 需要将虚拟地址翻译成物理地址，这样才能访问到真实的物理内存。实际上完成虚拟地址转换为物理地址转换的硬件是 CPU 中含有一个被称为 **内存管理单元 (Memory Management Unit, MMU)** 的硬件。如下图所示：



CPU: Central Processing Unit

MMU: Memory Management Unit

TLB: Translation lookaside buffer

为什么要有虚拟地址空间呢？

先从没有虚拟地址空间的时候说起吧！没有虚拟地址空间的时候，程序都是直接访问和操作的都是物理内存。但是这样有什么问题呢？

1. 用户程序可以访问任意内存，寻址内存的每个字节，这样就很容易（有意或者无意）破坏操作系统，造成操作系统崩溃。
2. 想要同时运行多个程序特别困难，比如你想同时运行一个微信和一个QQ音乐都不行。为什么呢？举个简单的例子：微信在运行的时候给内存地址1xxx赋值后，QQ音乐也同样给内存地址1xxx赋值，那么QQ音乐对内存的赋值就会覆盖微信之前所赋的值，这就造成了微信这个程序就会崩溃。

总结来说：如果直接把物理地址暴露出来的话会带来严重问题，比如可能对操作系统造成伤害以及给同时运行多个程序造成困难。

通过虚拟地址访问内存有以下优势：

- 程序可以使用一系列相邻的虚拟地址来访问物理内存中不相邻的大内存缓冲区。
- 程序可以使用一系列虚拟地址来访问大于可用物理内存的内存缓冲区。当物理内存的供应量变小时，内存管理器会将物理内存页（通常大小为4KB）保存到磁盘文件。数据或代码页会根据需要在物理内存与磁盘之间移动。
- 不同进程使用的虚拟地址彼此隔离。一个进程中的代码无法更改正在由另一进程或操作系统使用的物理内存。

四 虚拟内存

4.1 什么是虚拟内存(Virtual Memory)?

面试官：再问你一个常识性的问题！什么是虚拟内存(Virtual Memory)？

 我：这个在我们平时使用电脑特别是 Windows 系统的时候太常见了。很多时候我们使用点开了很多占内存的软件，这些软件占用的内存可能已经远远超出了我们电脑本身具有的物理内存。**为什么可以这样呢？** 正是因为**虚拟内存**的存在，通过**虚拟内存**可以让程序可以拥有超过系统物理内存大小的可用内存空间。另外，**虚拟内存**为每个进程提供了一个一致的、私有的地址空间，它让每个进程产生了一种自己在独享主存的错觉（每个进程拥有一片连续完整的内存空间）。这样会更加有效地管理内存并减少出错。

虚拟内存是计算机系统内存管理的一种技术，我们可以手动设置自己电脑的虚拟内存。不要单纯认为**虚拟内存**只是“使用硬盘空间来扩展内存”的技术。**虚拟内存的重要意义是它定义了一个连续的虚拟地址空间，并且把内存扩展到硬盘空间。**推荐阅读：[《虚拟内存的那点事儿》](#)

维基百科中有几句话是这样介绍虚拟内存的。

虚拟内存使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。与没有使用**虚拟内存**技术的系统相比，使用这种技术的系统使得大型程序的编写变得更容易，对真正的物理内存（例如 RAM）的使用也更有效率。目前，大多数操作系统都使用了**虚拟内存**，如 Windows 家族的“**虚拟内存**”；Linux 的“**交换空间**”等。From:<https://zh.wikipedia.org/wiki/虚拟内存>

4.2 局部性原理

 面试官：要想更好地理解**虚拟内存**技术，必须要知道计算机中著名的**局部性原理**。另外，**局部性原理**既适用于程序结构，也适用于数据结构，是非常重要的一个概念。

 我：**局部性原理**是**虚拟内存**技术的基础，正是因为程序运行具有**局部性原理**，才可以只装入部分程序到内存就开始运行。

以下内容摘自《计算机操作系统教程》第 4 章存储器管理。

早在 1968 年的时候，就有人指出我们的程序在执行的时候往往呈现**局部性规律**，也就是说在某个较短的时间段内，程序执行局限于某一小部分，程序访问的存储空间也局限于某个区域。

局部性原理表现在以下两个方面：

1. **时间局部性**：如果程序中的某条指令一旦执行，不久以后该指令可能再次执行；如果某数据被访问过，不久以后该数据可能再次被访问。产生**时间局部性**的典型原因，是由于在程序中存在着大量的循环操作。
2. **空间局部性**：一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，这是因为指令通常是顺序存放、顺序执行的，数据也一般是以向量、数组、表等形式簇聚存储的。

时间局部性是通过将近来使用的指令和数据保存到高速缓存存储器中，并使用高速缓存的层次结构实现。**空间局部性**通常是使用较大的高速缓存，并将预取机制集成到高速缓存控制逻辑中实现。**虚拟内存**技术实际上就是建立了“**内存—外存**”的两级存储器的结构，利用**局部性原理**实现高速缓存。

4.3 虚拟存储器

 面试官：都说了**虚拟内存**了。你再讲讲**虚拟存储器**把！

 我：

这部分内容来自：[王道考研操作系统知识点整理](#)。

基于局部性原理，在程序装入时，可以将程序的一部分装入内存，而将其他部分留在外存，就可以启动程序执行。由于外存往往比内存大很多，所以我们运行的软件的内存大小实际上是可以比计算机系统实际的内存大小大的。在程序执行过程中，当所访问的信息不在内存时，由操作系统将所需要的部分调入内存，然后继续执行程序。另一方面，操作系统将内存中暂时不使用的内容换到外存上，从而腾出空间存放将要调入内存的信息。这样，计算机好像为用户提供了一个比实际内存大的多的存储器——虚拟存储器。

实际上，我觉得虚拟内存同样是一种时间换空间的策略，你用 CPU 的计算时间，页的调入调出花费的时间，换来了一个虚拟的更大的空间来支持程序的运行。不得不感叹，程序世界几乎不是时间换空间就是空间换时间。

4.4 虚拟内存的技术实现

 面试官：虚拟内存技术的实现呢？

 我：虚拟内存的实现需要建立在离散分配的内存管理方式的基础上。虚拟内存的实现有以下三种方式：

1. **请求分页存储管理**：建立在分页管理之上，为了支持虚拟存储器功能而增加了请求调页功能和页面置换功能。请求分页是目前最常用的一种实现虚拟存储器的方法。请求分页存储管理系统中，在作业开始运行之前，仅装入当前要执行的部分段即可运行。假如在作业运行的过程中发现要访问的页面不在内存，则由处理器通知操作系统按照对应的页面置换算法将相应的页面调入到主存，同时操作系统也可以将暂时不用的页面置换到外存中。
2. **请求分段存储管理**：建立在分段存储管理之上，增加了请求调段功能、分段置换功能。请求分段存储管理方式就如同请求分页存储管理方式一样，在作业开始运行之前，仅装入当前要执行的部分段即可运行；在执行过程中，可使用请求调入中断动态装入要访问但又不在内存的程序段；当内存空间已满，而又需要装入新的段时，根据置换功能适当调出某个段，以便腾出空间而装入新的段。
3. **请求段页式存储管理**

这里多说一下？很多人容易搞混请求分页与分页存储管理，两者有何不同呢？

请求分页存储管理建立在分页管理之上。他们的根本区别是是否将程序全部所需的全部地址空间都装入主存，这也是请求分页存储管理可以提供虚拟内存的原因，我们在上面已经分析过了。

它们之间的根本区别在于是否将一作业的全部地址空间同时装入主存。请求分页存储管理不要求将作业全部地址空间同时装入主存。基于这一点，请求分页存储管理可以提供虚存，而分页存储管理却不能提供虚存。

不管是上面那种实现方式，我们一般都需要：

1. 一定容量的内存和外存：在载入程序的时候，只需要将程序的一部分装入内存，而将其他部分留在外存，然后程序就可以执行了；
2. 缺页中断：如果需执行的指令或访问的数据尚未在内存（称为缺页或缺段），则由处理器通知操作系统将相应的页面或段调入到内存，然后继续执行程序；
3. 虚拟地址空间：逻辑地址到物理地址的变换。

4.5 页面置换算法

 面试官：虚拟内存管理很重要的一个概念就是页面置换算法。那你说一下 页面置换算法的作用？常见的页面置换算法有哪些？

 我：

| 这个题目经常作为笔试题出现，网上已经给出了很不错的回答，我这里只是总结整理了一下。

地址映射过程中，若在页面中发现所要访问的页面不在内存中，则发生缺页中断。

缺页中断 就是要访问的页不在主存，需要操作系统将其调入主存后再进行访问。在这个时候，被内存映射的文件实际上成了一个分页交换文件。

当发生缺页中断时，如果当前内存中并没有空闲的页面，操作系统就必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。用来选择淘汰哪一页的规则叫做页面置换算法，我们可以把页面置换算法看成是淘汰页面的规则。

- **OPT 页面置换算法（最佳页面置换算法）**：最佳(Optimal, OPT)置换算法所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。但由于人们目前无法预知进程在内存下的若干页面中哪个是未来最长时间内不再被访问的，因而该算法无法实现。一般作为衡量其他置换算法的方法。
- **FIFO (First In First Out) 页面置换算法（先进先出页面置换算法）**：总是淘汰最先进入内存的页面，即选择在内存中驻留时间最长的页面进行淘汰。
- **LRU (Least Currently Used) 页面置换算法（最近最久未使用页面置换算法）**：LRU算法赋予每个页面一个访问字段，用来记录一个页面自上次被访问以来所经历的时间 T，当须淘汰一个页面时，选择现有页面中其 T 值最大的，即最近最久未使用的页面予以淘汰。
- **LFU (Least Frequently Used) 页面置换算法（最少使用页面置换算法）**：该置换算法选择在之前时期使用最少的页面作为淘汰页。

Reference

- 《计算机操作系统-汤小丹》第四版
- 《深入理解计算机系统》
- <https://zh.wikipedia.org/wiki/输入输出内存管理单元>
- <https://baike.baidu.com/item/快表/19781679>
- <https://www.jianshu.com/p/1d47ed0b46d5>
- <https://www.studytonight.com/operating-system>
- <https://www.geeksforgeeks.org/interprocess-communication-methods/>
- <https://juejin.im/post/59f8691b51882534af254317>
- 王道考研操作系统知识点整理：<https://wizardforcel.gitbooks.io/wangdaokaoyan-os/content/13.html>

四 数据库面试题总结

4.1 MySQL

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

4.1.1 精品推荐

书籍推荐

- 《SQL基础教程（第2版）》（入门级）
- 《高性能MySQL：第3版》（进阶）

文字教程推荐

- [SQL Tutorial](#) (SQL语句学习,英文)、[SQL Tutorial](#) (SQL语句学习,中文)、[SQL语句在线练习](#) (非常不错)
- [Github-MySQL入门教程 \(MySQL tutorial book\)](#) (从零开始学习MySQL, 主要是面向MySQL数据库管理系统初学者)
- [官方教程](#)
- [MySQL 教程 \(菜鸟教程\)](#)

相关资源推荐

- [中国5级行政区域mysql库](#)

视频教程推荐

基础入门：[与MySQL的零距离接触-慕课网](#)

MySQL开发技巧：[MySQL开发技巧（一）](#) [MySQL开发技巧（二）](#) [MySQL开发技巧（三）](#)

MySQL5.7新特性及相关优化技巧：[MySQL5.7版本新特性](#) [性能优化之MySQL优化](#)

[MySQL集群（PXC）入门](#) [MyCAT入门及应用](#)

常见问题总结

4.1.2 什么是MySQL?

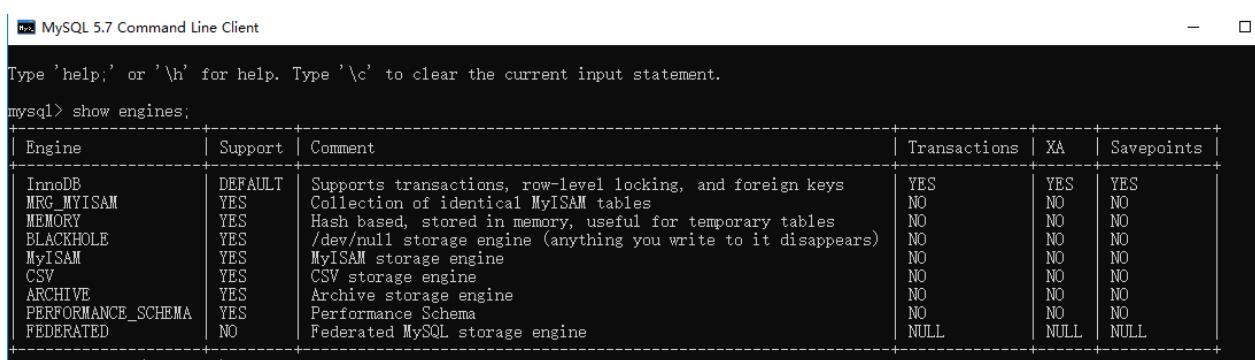
MySQL 是一种关系型数据库，在Java企业级开发中非常常用，因为 MySQL 是开源免费的，并且方便扩展。阿里巴巴数据库系统也大量用到了 MySQL，因此它的稳定性是有保障的。MySQL是开放源代码的，因此任何人都可以在 GPL(General Public License) 的许可下下载并根据个性化的需要对其进行修改。MySQL的默认端口号是3306。

4.1.3 存储引擎

一些常用命令

查看MySQL提供的所有存储引擎

```
mysql> show engines;
```



```
MySQL 5.7 Command Line Client

Type 'help,' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show engines;
+-----+-----+-----+-----+-----+-----+
| Engine | Support | Comment | Transactions | XA | Savepoints |
+-----+-----+-----+-----+-----+-----+
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES | YES | YES |
| MRG_MYISAM | YES | Collection of identical MyISAM tables | NO | NO | NO |
| MEMORY | YES | Hash based, stored in memory, useful for temporary tables | NO | NO | NO |
| BLACKHOLE | YES | /dev/null storage engine (anything you write to it disappears) | NO | NO | NO |
| MyISAM | YES | MyISAM storage engine | NO | NO | NO |
| CSV | YES | CSV storage engine | NO | NO | NO |
| ARCHIVE | YES | Archive storage engine | NO | NO | NO |
| PERFORMANCE_SCHEMA | YES | Performance Schema | NO | NO | NO |
| FEDERATED | NO | Federated MySQL storage engine | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+-----+
```

从上图我们可以查看出 MySQL 当前默认的存储引擎是 InnoDB，并且在 5.7 版本所有的存储引擎中只有 InnoDB 是事务性存储引擎，也就是说只有 InnoDB 支持事务。

查看 MySQL 当前默认的存储引擎

我们也可以通过下面的命令查看默认的存储引擎。

```
mysql> show variables like '%storage_engine%';
```

查看表的存储引擎

```
show table status like "table_name";
```

```
mysql> show table status like "tb_base";
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Name | Engine | Version | Row_format | Rows | Avg_row_length | Data_length | Max_data_length | Index_length | Data_free | Auto_increment | Create_time |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| tb_base | InnoDB | 10 | Dynamic | 0 | 0 | 16384 | 0 | 0 | 0 | 2 | 2019-05-05 15:58:24
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

MyISAM 和 InnoDB 区别

MyISAM 是 MySQL 的默认数据库引擎（5.5 版之前）。虽然性能极佳，而且提供了大量的特性，包括全文索引、压缩、空间函数等，但 MyISAM 不支持事务和行级锁，而且最大的缺陷就是崩溃后无法安全恢复。不过，5.5 版本之后，MySQL 引入了 InnoDB（事务性数据库引擎），MySQL 5.5 版本后默认的存储引擎为 InnoDB。

大多数时候我们使用的都是 InnoDB 存储引擎，但是在某些情况下使用 MyISAM 也是合适的比如读密集的情况下。（如果你不介意 MyISAM 崩溃恢复问题的话）。

两者的对比：

- 是否支持行级锁：MyISAM 只有表级锁(table-level locking)，而 InnoDB 支持行级锁(row-level locking)和表级锁，默认为行级锁。
- 是否支持事务和崩溃后的安全恢复：MyISAM 强调的是性能，每次查询具有原子性，其执行速度比 InnoDB 类型更快，但是不提供事务支持。但是 InnoDB 提供事务支持，外部键等高级数据库功能。具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。
- 是否支持外键：MyISAM 不支持，而 InnoDB 支持。
- 是否支持 MVCC：仅 InnoDB 支持。应对高并发事务，MVCC 比单纯的加锁更高效；MVCC 只在 READ COMMITTED 和 REPEATABLE READ 两个隔离级别下工作；MVCC 可以使用乐观(optimistic)锁和悲观(pessimistic)锁来实现；各数据库中 MVCC 实现并不统一。推荐阅读：[MySQL-InnoDB-MVCC 多版本并发控制](#)
-

《MySQL 高性能》上面有一句话这样写到：

不要轻易相信“MyISAM 比 InnoDB 快”之类的经验之谈，这个结论往往不是绝对的。在很多我们已知场景中，InnoDB 的速度都可以让 MyISAM 望尘莫及，尤其是用到了聚簇索引，或者需要访问的数据都可以放入内存的应用。

一般情况下我们选择 InnoDB 都是没有问题的，但是某些情况下你并不在乎可扩展能力和并发能力，也不需要事务支持，也不在乎崩溃后的安全恢复问题的话，选择 MyISAM 也是一个不错的选择。但是一般情况下，我们都是需要考虑到这些问题的。

4.1.4 字符集及校对规则

字符集指的是一种从二进制编码到某类字符符号的映射。校对规则则是指某种字符集下的排序规则。MySQL中每一种字符集都会对应一系列的校对规则。

MySQL采用的是类似继承的方式指定字符集的默认值，每个数据库以及每张数据表都有自己的默认值，他们逐层继承。比如：某个库中所有表的默认字符集将是该数据库所指定的字符集（这些表在没有指定字符集的情况下，才会采用默认字符集） PS：整理自《Java工程师修炼之道》

详细内容可以参考：[MySQL字符集及校对规则的理解](#)

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

4.1.5 索引

MySQL索引使用的数据结构主要有**BTree索引** 和 **哈希索引**。对于哈希索引来说，底层的数据结构就是哈希表，因此在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快；其余大部分场景，建议选择BTree索引。

MySQL的BTree索引使用的是B树中的B+Tree，但对于主要的两种存储引擎的实现方式是不同的。

- **MyISAM：** B+Tree叶节点的data域存放的是数据记录的地址。在索引检索的时候，首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其 data 域的值，然后以 data 域的值为地址读取相应的数据记录。这被称为“非聚簇索引”。
- **InnoDB：** 其数据文件本身就是索引文件。相比MyISAM，索引文件和数据文件是分离的，其表数据文件本身就是按B+Tree组织的一个索引结构，树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。这被称为“聚簇索引（或聚集索引）”。而其余的索引都作为辅助索引，辅助索引的data域存储相应记录主键的值而不是地址，这也是和MyISAM不同的地方。在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。 PS：整理自《Java工程师修炼之道》

更多关于索引的内容可以查看文档首页MySQL目录下关于索引的详细总结。

4.1.6 查询缓存的使用

执行查询语句的时候，会先查询缓存。不过，MySQL 8.0 版本后移除，因为这个功能不太实用

my.cnf加入以下配置，重启MySQL开启查询缓存

```
query_cache_type=1  
query_cache_size=600000
```

MySQL执行以下命令也可以开启查询缓存

```
set global query_cache_type=1;  
set global query_cache_size=600000;
```

如上，开启查询缓存后在同样的查询条件以及数据情况下，会直接在缓存中返回结果。这里的查询条件包括查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息。因此任何两个查询在任何字符上的不同都会导致缓存不命中。此外，如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、MySQL库中的系统表，其查询结果也不会被缓存。

缓存建立之后，MySQL的查询缓存系统会跟踪查询中涉及的每张表，如果这些表（数据或结构）发生变化，那么和这张表相关的所有缓存数据都将失效。

缓存虽然能够提升数据库的查询性能，但是缓存同时也带来了额外的开销，每次查询后都要做一次缓存操作，失效后还要销毁。因此，开启缓存查询要谨慎，尤其对于写密集的应用来说更是如此。如果开启，要注意合理控制缓存空间大小，一般来说其大小设置为几十MB比较合适。此外，还可以通过 `sql_cache` 和 `sql_no_cache` 来控制某个查询语句是否需要缓存：

```
select sql_no_cache count(*) from usr;
```

4.1.7 什么是事务？

事务是逻辑上的一组操作，要么都执行，要么都不执行。

事务最经典也经常被拿出来说例子就是转账了。假如小明要给小红转账1000元，这个转账会涉及到两个关键操作就是：将小明的余额减少1000元，将小红的余额增加1000元。万一在这两个操作之间突然出现错误比如银行系统崩溃，导致小明余额减少而小红的余额没有增加，这样就不对了。事务就是保证这两个关键操作要么都成功，要么都要失败。

4.1.8 事物的四大特性(ACID)



1. **原子性 (Atomicity)**：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
2. **一致性 (Consistency)**：执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；
3. **隔离性 (Isolation)**：并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
4. **持久性 (Durability)**：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

4.1.9 并发事务带来哪些问题？

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务（多个用户对同一数据进行操作）。并发虽然是必须的，但可能会导致以下的问题。

- 脏读 (Dirty read)**：当一个事务正在访问数据并且对数据进行了修改，而这种修改还没有提交到数据库中，这时另外一个事务也访问了这个数据，然后使用了这个数据。因为这个数据是还没有提交的数据，那么另外一个事务读到的这个数据是“脏数据”，依据“脏数据”所做的操作可能是不正确的。
- 丢失修改 (Lost to modify)**：指在一个事务读取一个数据时，另外一个事务也访问了该数据，那么在第一个事务中修改了这个数据后，第二个事务也修改了这个数据。这样第一个事务内的修改结果就被丢失，因此称为丢失修改。例如：事务1读取某表中的数据A=20，事务2也读取A=20，事务1修改A=A-1，事务2也修改A=A-1，最终结果A=19，事务1的修改被丢失。
- 不可重复读 (Unrepeatable read)**：指在一个事务内多次读同一数据。在这个事务还没有结束时，另一个事务也访问该数据。那么，在第一个事务中的两次读数据之间，由于第二个事务的修改导致第一个事务两次读取的数据可能不太一样。这就发生了在一个事务内两次读到的数据是不一样的情况，因此称为不可重复读。
- 幻读 (Phantom read)**：幻读与不可重复读类似。它发生在一个事务 (T1) 读取了几行数据，接着另一个并发事务 (T2) 插入了一些数据时。在随后的查询中，第一个事务 (T1) 就会发现多了些原本不存在的记录，就好像发生了幻觉一样，所以称为幻读。

不可重复读和幻读区别：

不可重复读的重点是修改比如多次读取一条记录发现其中某些列的值被修改，幻读的重点在于新增或者删除比如多次读取一条记录发现记录增多或减少了。

4.1.10 事务隔离级别有哪些?MySQL的默认隔离级别是?

SQL 标准定义了四个隔离级别：

- READ-UNCOMMITTED(读取未提交)**：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。
- READ-COMMITTED(读取已提交)**：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。
- REPEATABLE-READ(可重复读)**：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- SERIALIZABLE(可串行化)**：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	✗	√	√
REPEATABLE-READ	✗	✗	√
SERIALIZABLE	✗	✗	✗

MySQL InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ (可重读)。我们可以通过 `SELECT @@tx_isolation;` 命令来查看

```
mysql> SELECT @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
```

这里需要注意的是：与 SQL 标准不同的地方在于 InnoDB 存储引擎在 REPEATABLE-READ（可重读） 事务隔离级别下使用的是Next-Key Lock 锁算法，因此可以避免幻读的产生，这与其他数据库系统(如 SQL Server) 是不同的。所以说InnoDB 存储引擎的默认支持的隔离级别是 REPEATABLE-READ（可重读） 已经可以完全保证事务的隔离性要求，即达到了 SQL标准的 SERIALIZABLE(可串行化) 隔离级别。因为隔离级别越低，事务请求的锁越少，所以大部分数据库系统的隔离级别都是 READ-COMMITTED(读取提交内容) ，但是你要知道的是InnoDB 存储引擎默认使用 REPEAaTABLE-READ（可重读） 并不会有性能损失。

InnoDB 存储引擎在 分布式事务 的情况下一般会用到 SERIALIZABLE(可串行化) 隔离级别。

4.1.11 锁机制与InnoDB锁算法

MyISAM和InnoDB存储引擎使用的锁：

- MyISAM采用表级锁(table-level locking)。
- InnoDB支持行级锁(row-level locking)和表级锁,默认为行级锁

表级锁和行级锁对比：

- 表级锁： MySQL中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM和 InnoDB引擎都支持表级锁。
- 行级锁： MySQL中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

详细内容可以参考： MySQL锁机制简单了解一下：https://blog.csdn.net/qq_34337272/article/details/80611486

InnoDB存储引擎的锁的算法有三种：

- 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: record+gap 锁定一个范围，包含记录本身

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决Phantom Problem幻读问题
3. 当查询的索引含有唯一属性时，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生
5. 有两种方式显式关闭gap锁： (除了外键约束和唯一性检查外，其余情况仅使用record lock)
 - A. 将事务隔离级别设置为RC
 - B. 将参数innodb_locks_unsafe_for_binlog设置为1

4.1.12 大表优化

当MySQL单表记录数过大时，数据库的CRUD性能会明显下降，一些常见的优化措施如下：

限定数据的范围

务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内；

读/写分离

经典的数据库拆分方案，主库负责写，从库负责读；

垂直分区

根据数据库里面数据表的相关性进行拆分。例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说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。如下图所示，这样来说大家应该就更容易理解了。

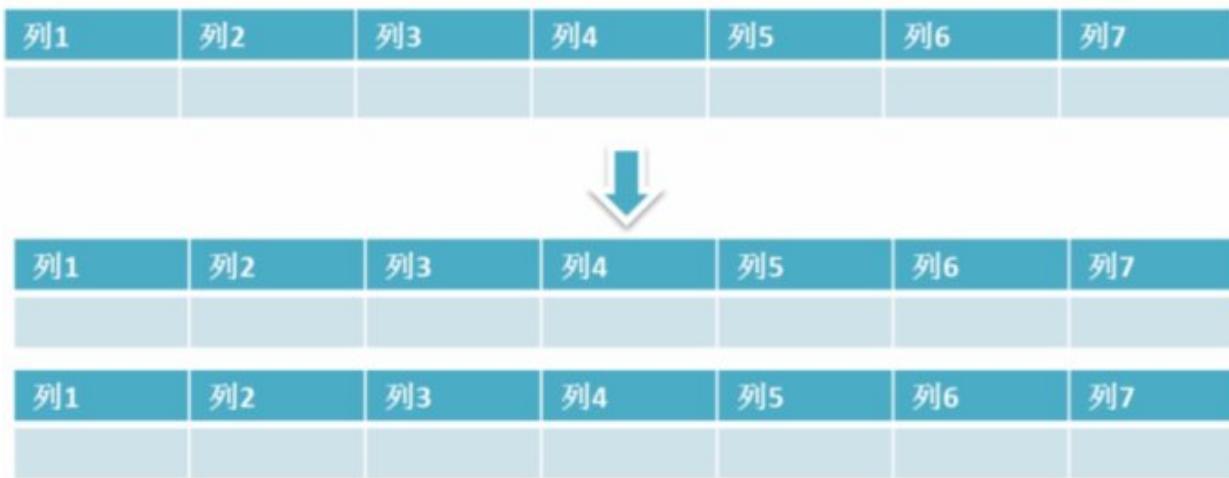


- **垂直拆分的优点：** 可以使得列数据变小，在查询时减少读取的Block数，减少I/O次数。此外，垂直分区可以简化表的结构，易于维护。
- **垂直拆分的缺点：** 主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外，垂直分区会让事务变得更加复杂；

水平分区

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。水平拆分可以支撑非常大的数据量。

水平拆分是指数据表行的拆分，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。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。



水平拆分可以支持非常大的数据量。需要注意的一点是：分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升MySQL并发能力没有什么意义，所以 **水平拆分最好分库**。

水平拆分能够 **支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决，跨节点Join性能较差，逻辑复杂**。《Java工程师修炼之道》的作者推荐 **尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度**，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络I/O。

下面补充一下数据库分片的两种常见方案：

- 客户端代理：分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC层来实现。当当网的 Sharding-JDBC、阿里的TDDL是两种比较常用的实现。
- 中间件代理：在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。我们现在谈的 Mycat、360的Atlas、网易的DDB等等都是这种架构的实现。

详细内容可以参考： MySQL大表优化方案：<https://segmentfault.com/a/1190000006158186>

4.1.13 解释一下什么是池化设计思想。什么是数据库连接池?为什么需要数据库连接池?

池化设计应该不是一个新名词。我们常见的如java线程池、jdbc连接池、redis连接池等就是这类设计的代表实现。这种设计会初始预设资源，解决的问题就是抵消每次获取资源的消耗，如创建线程的开销，获取远程连接的开销等。就好比你去食堂打饭，打饭的大妈会先把饭盛好几份放那里，你来了就直接拿着饭盒加菜即可，不用再临时又盛饭又打菜，效率就高了。除了初始化资源，池化设计还包括如下这些特征：池子的初始值、池子的活跃值、池子的最大值等，这些特征可以直接映射到java线程池和数据库连接池的成员属性中。这篇文章对**池化设计思想**介绍的还不错，直接复制过来，避免重复造轮子了。

数据库连接本质就是一个 socket 的连接。数据库服务端还要维护一些缓存和用户权限信息之类的 所以占用了一些内存。我们可以把数据库连接池看做是维护的数据库连接的缓存，以便将来需要对数据库的请求时可以重用这些连接。为每个用户打开和维护数据库连接，尤其是对动态数据库驱动的网站应用程序的请求，既昂贵又浪费资源。在连接池中，创建连接后，将其放置在池中，并再次使用它，因此不必建立新的连接。如果使用了所有连接，则会建立一个新连接并将其添加到池中。连接池还减少了用户必须等待建立与数据库的连接的时间。

4.1.14 分库分表之后,id 主键如何处理?

因为要是分成多个表之后，每个表都是从 1 开始累加，这样是不对的，我们需要一个全局唯一的 id 来支持。

生成全局 id 有下面这几种方式：

- **UUID**: 不适合作为主键，因为太长了，并且无序不可读，查询效率低。比较适合用于生成唯一的名字的标示比如文件的名字。
- **数据库自增 id**：两台数据库分别设置不同步长，生成不重复ID的策略来实现高可用。这种方式生成的 id 有序，但是需要独立部署数据库实例，成本高，还会有性能瓶颈。
- **利用 redis 生成 id**：性能比较好，灵活方便，不依赖于数据库。但是，引入了新的组件造成系统更加复杂，可用性降低，编码更加复杂，增加了系统成本。
- **Twitter的snowflake算法**：Github 地址：<https://github.com/twitter-archive/snowflake>。
- **美团的Leaf分布式ID生成系统**：Leaf 是美团开源的分布式ID生成器，能保证全局唯一性、趋势递增、单调递增、信息安全，里面也提到了几种分布式方案的对比，但也需要依赖关系数据库、Zookeeper等中间件。感觉还不错。美团技术团队的一篇文章：<https://tech.meituan.com/2017/04/21/mt-leaf.html>。
-

4.1.15 一条SQL语句在MySQL中如何执行的

[一条SQL语句在MySQL中如何执行的](#)

4.1.16 MySQL高性能优化规范建议

[MySQL高性能优化规范建议](#)

4.1.17 一条SQL语句执行得很慢的原因有哪些？

[腾讯面试：一条SQL语句执行得很慢的原因有哪些？---不看后悔系列](#)

4.1.19 后端程序员必备：书写高质量SQL的30条建议

[后端程序员必备：书写高质量SQL的30条建议](#)

4.2 Redis

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

4.2.1 redis 简介

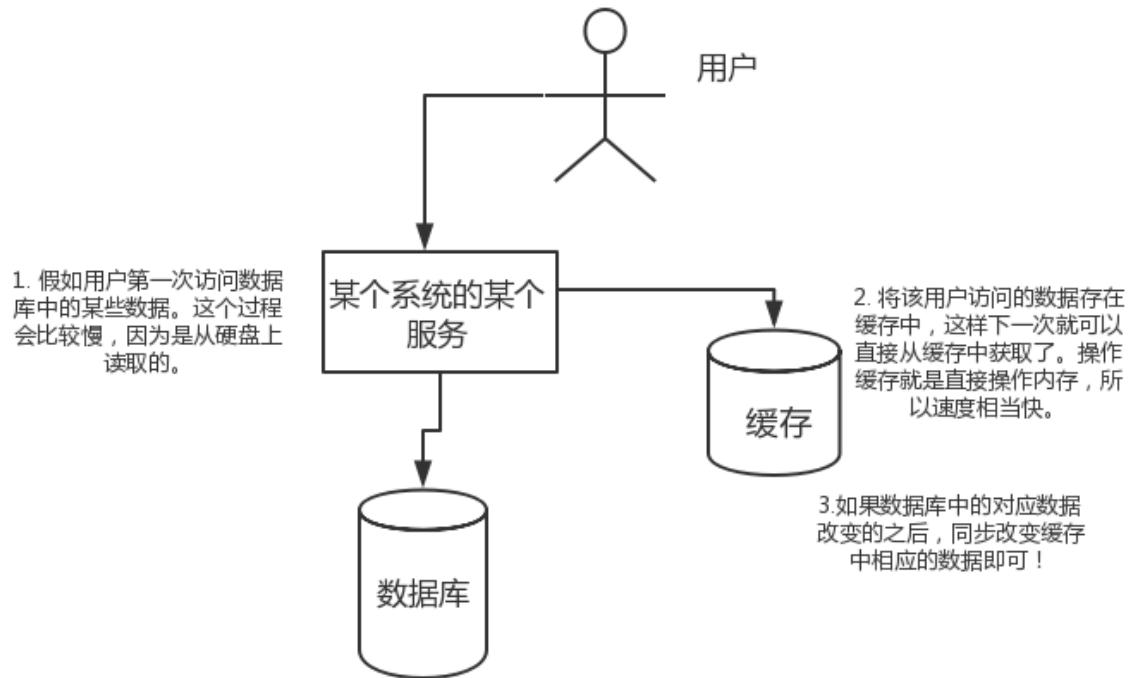
简单来说 redis 就是一个数据库，不过与传统数据库不同的是 redis 的数据是存在内存中的，所以读写速度非常快，因此 redis 被广泛应用于缓存方向。另外，redis 也经常用来做分布式锁。redis 提供了多种数据类型来支持不同的业务场景。除此之外，redis 支持事务、持久化、LUA脚本、LRU驱动事件、多种集群方案。

为什么要用 redis/为什么要用缓存

主要从“高性能”和“高并发”这两点来看待这个问题。

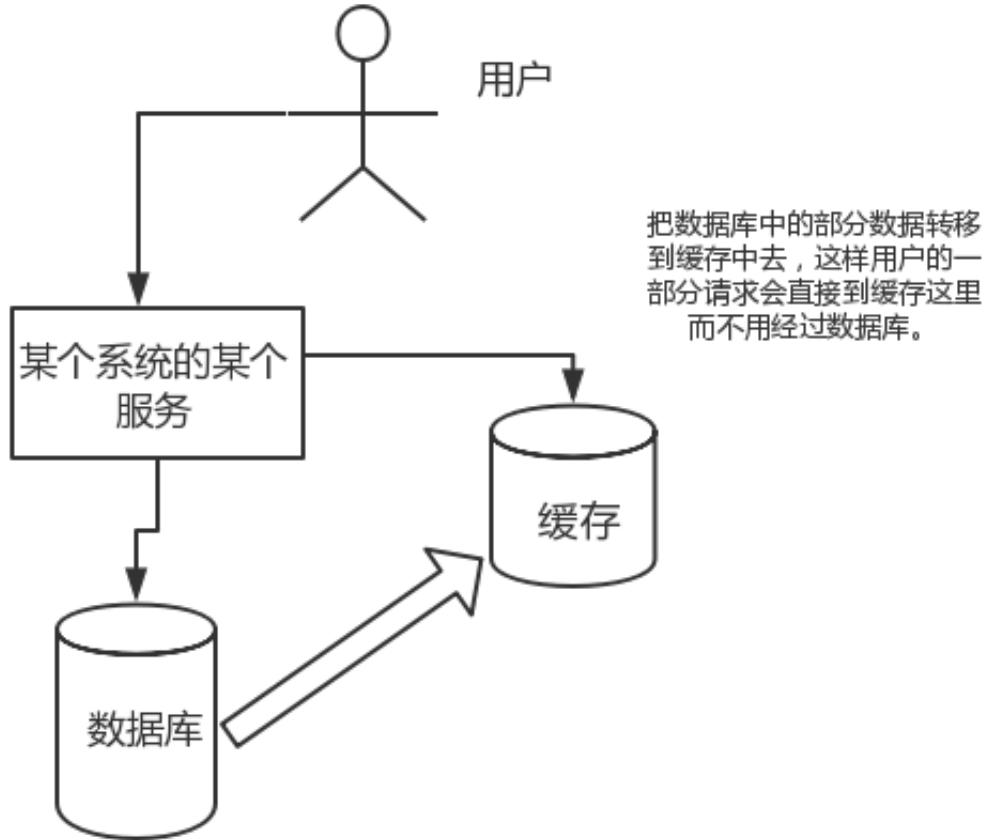
高性能：

假如用户第一次访问数据库中的某些数据。这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。如果数据库中的对应数据改变的之后，同步改变缓存中相应的数据即可！



高并发：

直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中去，这样用户的一部分请求会直接到缓存这里而不用经过数据库。



为什么要用 redis 而不用 map/guava 做缓存？

下面的内容来自 segmentfault 一位网友的提问，地址：<https://segmentfault.com/q/1010000009106416>

缓存分为本地缓存和分布式缓存。以 Java 为例，使用自带的 map 或者 guava 实现的是本地缓存，最主要的特点是轻量以及快速，生命周期随着 jvm 的销毁而结束，并且在多实例的情况下，每个实例都需要各自保存一份缓存，缓存不具有一致性。

使用 redis 或 memcached 之类的称为分布式缓存，在多实例的情况下，各实例共用一份缓存数据，缓存具有一致性。缺点是需要保持 redis 或 memcached 服务的高可用，整个程序架构上较为复杂。

4.2.2 redis 的线程模型

参考地址：<https://www.javazhiyin.com/22943.html>

redis 内部使用文件事件处理器 `file event handler`，这个文件事件处理器是单线程的，所以 redis 才叫做单线程的模型。它采用 IO 多路复用机制同时监听多个 socket，根据 socket 上的事件来选择对应的事件处理器进行处理。

文件事件处理器的结构包含 4 个部分：

- 多个 socket
- IO 多路复用程序
- 文件事件分派器
- 事件处理器（连接应答处理器、命令请求处理器、命令回复处理器）

多个 socket 可能会并发产生不同的操作，每个操作对应不同的文件事件，但是 IO 多路复用程序会监听多个 socket，会将 socket 产生的事件放入队列中排队，事件分派器每次从队列中取出一个事件，把该事件交给对应的事件处理器进行处理。

4.2.3 redis 和 memcached 的区别

对于 redis 和 memcached 我总结了下面四点。现在公司一般都是用 redis 来实现缓存，而且 redis 自身也越来越强大了！

1. **redis支持更丰富的数据类型（支持更复杂的应用场景）**：Redis不仅仅支持简单的k/v类型的数据，同时还提供list, set, zset, hash等数据结构的存储。memcache支持简单数据类型，String。
2. **Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用，而Memecache把数据全部存在内存之中。**
3. **集群模式：**memcached没有原生的集群模式，需要依靠客户端来实现往集群中分片写入数据；但是 redis 目前是原生支持 cluster 模式的。
4. **Memcached是多线程，非阻塞IO复用的网络模型；Redis使用单线程的多路 IO 复用模型。**

| 来自网络上的一张图，这里分享给大家！

对比参数	Redis	Memcached
类型	1、支持内存 2、非关系型数据库	1、支持内存 2、key-value键值对形式 3、缓存系统
数据存储类型	1、String 2、List 3、Set 4、Hash 5、Sort Set 【俗称ZSet】	1、文本型 2、二进制类型【新版增加】
查询【操作】类型	1、批量操作 2、事务支持【虽然是假的事务】 3、每个类型不同的CRUD	1、CRUD 2、少量的其他命令
附加功能	1、发布/订阅模式 2、主从分区 3、序列化支持 4、脚本支持【Lua脚本】	1、多线程服务支持
网络IO模型	1、单进程模式	2、多线程、非阻塞IO模式
事件库	自封装简易事件库AeEvent	贵族血统的LibEvent事件库
持久化支持	1、RDB 2、AOF	不支持

4.2.4 redis 常见数据结构以及使用场景分析

String

| 常用命令：set, get, decr, incr, mget 等。

String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。常规key-value缓存应用； 常规计数：微博数，粉丝数等。

Hash

| 常用命令： hget, hset, hgetall 等。

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，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。比如我们可以 hash 数据结构来存储用户信息，商品信息等等。比如下面我就用 hash 类型存放了我本人的一些信息：

```
key=JavaUser293847
value={
  "id": 1,
  "name": "SnailClimb",
  "age": 22,
  "location": "Wuhan, Hubei"
}
```

List

常用命令：lpush, rpush, lpop, rpop, lrange 等

list 就是链表，Redis list 的应用场景非常多，也是Redis最重要的数据结构之一，比如微博的关注列表，粉丝列表，消息列表等功能都可以用Redis的 list 结构来实现。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。

另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。

Set

常用命令： sadd, spop, smembers, sunion 等

set 对外提供的功能与list类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。

当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的接口，这个也是list所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。

比如：在微博应用中，可以将一个用户的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下：

```
sinterstore key1 key2 key3      将交集存在key1内
```

Sorted Set

常用命令： zadd, zrange, zrem, zcard 等

和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。

举例：在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维度的消息排行榜）等信息，适合使用 Redis 中的 Sorted Set 结构进行存储。

4.2.5 redis 设置过期时间

Redis中有个设置时间过期的功能，即对存储在 redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。如我们一般项目中的 token 或者一些登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间。

如果假设你设置了一批 key 只能存活1个小时，那么接下来1小时后，redis是怎么对这批key进行删除的？

定期删除+惰性删除。

通过名字大概就能猜出这两个删除方式的意思了。

- **定期删除**: redis默认是每隔 100ms 就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。注意这里是随机抽取的。为什么要随机呢？你想一想假如 redis 存了几十万个 key，每隔100ms就遍历所有的设置过期时间的 key 的话，就会给 CPU 带来很大的负载！
- **惰性删除** : 定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删除。假如你的过期 key，靠定期删除没有被删除掉，还停留在内存里，除非你的系统去查一下那个 key，才会被redis给删除掉。这就是所谓的惰性删除，也是够懒的哈！

但是仅仅通过设置过期时间还是有问题的。我们想一下：如果定期删除漏掉了很多过期 key，然后你也没及时去查，也就没走惰性删除，此时会怎么样？如果大量过期key堆积在内存里，导致redis内存块耗尽了。怎么解决这个问题呢？ **redis 内存淘汰机制**。

4.2.6 redis 内存淘汰机制(**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？**)

redis 配置文件 redis.conf 中有相关注释，我这里就不贴了，大家可以自行查阅或者通过这个网址查看：<http://download.redis.io/redis-stable/redis.conf>

redis 提供 6种数据淘汰策略：

1. **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
2. **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
3. **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
4. **allkeys-lru**: 当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key (这个是最常用的)
5. **allkeys-random**: 从数据集 (server.db[i].dict) 中任意选择数据淘汰
6. **no-eviction**: 禁止驱逐数据，也就是说当内存不足以容纳新写入数据时，新写入操作会报错。这个应该没人使用吧！

4.0版本后增加以下两种：

7. **volatile-lfu**: 从已设置过期时间的数据集 (server.db[i].expires) 中挑选最不经常使用的数据淘汰
8. **allkeys-lfu**: 当内存不足以容纳新写入数据时，在键空间中，移除最不经常使用的key

备注： 关于 redis 设置过期时间以及内存淘汰机制，我这里只是简单的总结一下，后面会专门写一篇文章来总结！

4.2.7 redis 持久化机制(怎么保证 redis 挂掉之后再重启数据可以进行恢复)

很多时候我们需要持久化数据也就是将内存中的数据写入到硬盘里面，大部分原因是为了之后重用数据（比如重启机器、机器故障之后恢复数据），或者是为了防止系统故障而将数据备份到一个远程位置。

Redis不同于Memcached的很重一点就是，Redis支持持久化，而且支持两种不同的持久化操作。Redis的一种持久化方式叫快照（snapshotting, RDB），另一种方式是只追加文件（append-only file, AOF）。这两种方法各有千秋，下面我会详细这两种持久化方法是什么，怎么用，如何选择适合自己的持久化方法。

快照 (snapshotting) 持久化 (RDB)

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。

快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置：

```
save 900 1          #在900秒(15分钟)之后，如果至少有1个key发生变化，  
Redis就会自动触发BGSAVE命令创建快照。  
  
save 300 10         #在300秒(5分钟)之后，如果至少有10个key发生变化，  
Redis就会自动触发BGSAVE命令创建快照。  
  
save 60 10000       #在60秒(1分钟)之后，如果至少有10000个key发生变化，  
Redis就会自动触发BGSAVE命令创建快照。
```

AOF (append-only file) 持久化

与快照持久化相比，AOF持久化的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF (append only file) 方式的持久化，可以通过appendonly参数开启：

```
appendonly yes
```

开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是appendonly.aof。

在Redis的配置文件中存在三种不同的 AOF 持久化方式，它们分别是：

```
appendfsync always    #每次有数据修改发生时都会写入AOF文件，这样会严重降  
低Redis的速度  
appendfsync everysec   #每秒钟同步一次，显示地将多个写命令同步到硬盘  
appendfsync no         #让操作系统决定何时进行同步
```

为了兼顾数据和写入性能，用户可以考虑 `appendfsync everysec` 选项，让 Redis 每秒同步一次 AOF 文件，Redis 性能几乎没受到任何影响。而且这样即使出现系统崩溃，用户最多只会丢失一秒之内产生的数据。当硬盘忙于执行写入操作的时候，Redis 还会优雅的放慢自己的速度以便适应硬盘的最大写入速度。

Redis 4.0 对于持久化机制的优化

Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项 `aof-use-rdb-preamble` 开启）。

如果把混合持久化打开，AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处是可以结合 RDB 和 AOF 的优点，快速加载同时避免丢失过多的数据。当然缺点也是有的，AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式，可读性较差。

补充内容：AOF 重写

AOF 重写可以产生一个新的 AOF 文件，这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样，但体积更小。

AOF 重写是一个有歧义的名字，该功能是通过读取数据库中的键值对来实现的，程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。

在执行 `BGREWRITEAOF` 命令时，Redis 服务器会维护一个 AOF 重写缓冲区，该缓冲区会在子进程创建新 AOF 文件期间，记录服务器执行的所有写命令。当子进程完成创建新 AOF 文件的工作之后，服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾，使得新旧两个 AOF 文件所保存的数据库状态一致。最后，服务器用新的 AOF 文件替换旧的 AOF 文件，以此来完成 AOF 文件重写操作。

更多内容可以查看我的这篇文章：

- [Redis 持久化](#)

4.2.8 redis 事务

Redis 通过 `MULTI`、`EXEC`、`WATCH` 等命令来实现事务(transaction)功能。事务提供了一种将多个命令请求打包，然后一次性、按顺序地执行多个命令的机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

在传统的关系式数据库中，常常用 ACID 性质来检验事务功能的可靠性和安全性。在 Redis 中，事务总是具有原子性 (Atomicity)、一致性 (Consistency) 和隔离性 (Isolation)，并且当 Redis 运行在某种特定的持久化模式下时，事务也具有持久性 (Durability)。

补充内容：

1. redis 同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚。（来自 [issue: 关于 Redis 事务不是原子性问题](#)）

4.2.9 缓存雪崩和缓存穿透问题解决方案

缓存雪崩

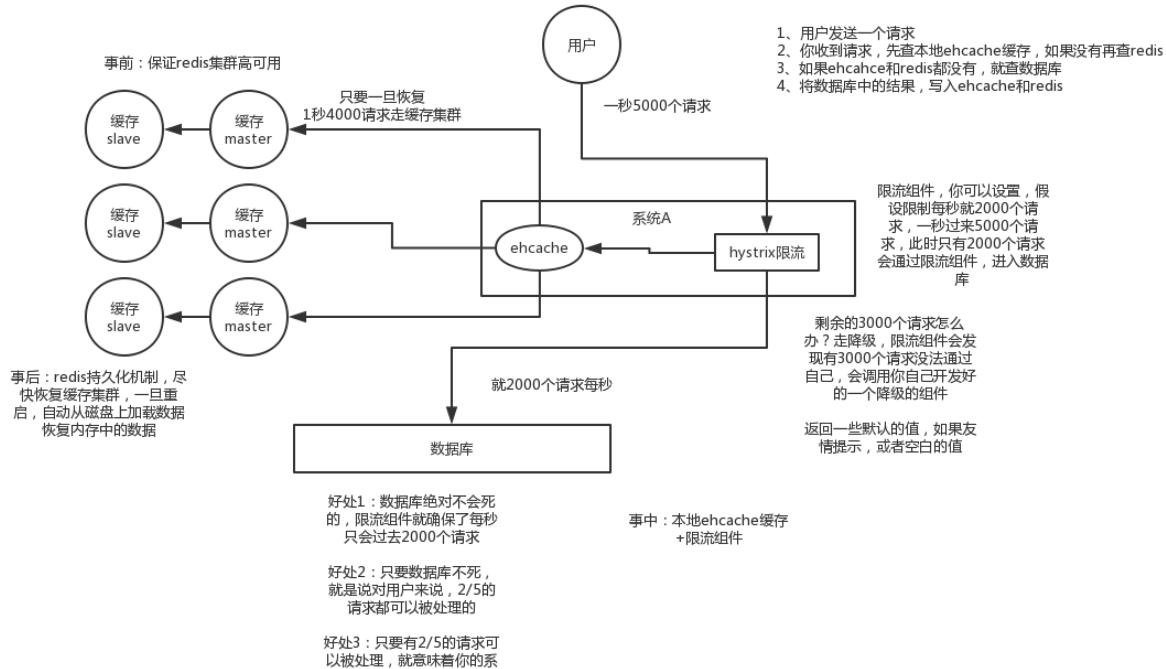
什么是缓存雪崩？

简介：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

有哪些解决办法？

(中华石杉老师在他的视频中提到过，视频地址在最后一个问题中有提到)：

- 事前：尽量保证整个 redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。
- 事中：本地ehcache缓存 + hystrix限流&降级，避免MySQL崩掉
- 事后：利用 redis 持久化机制保存的数据尽快恢复缓存

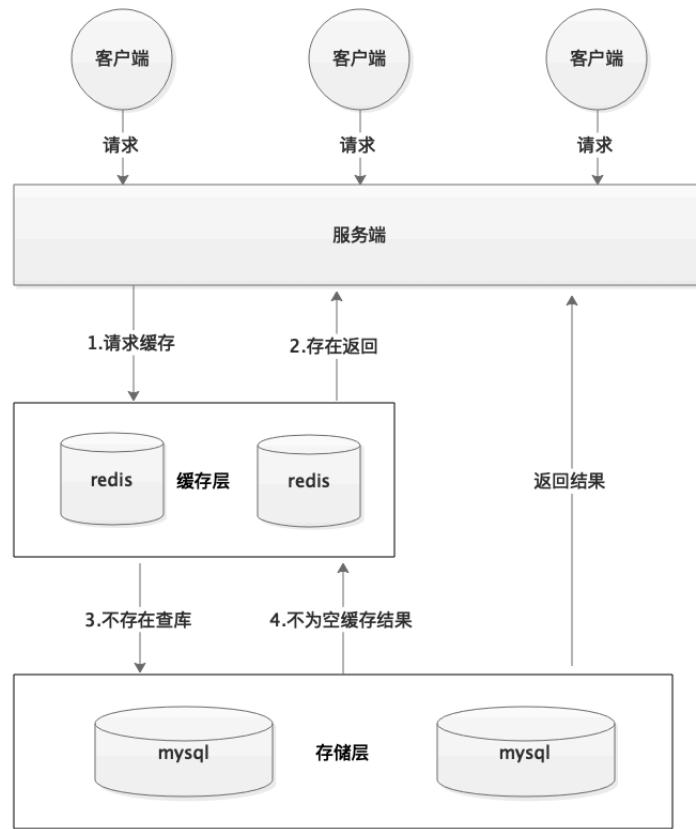


缓存穿透

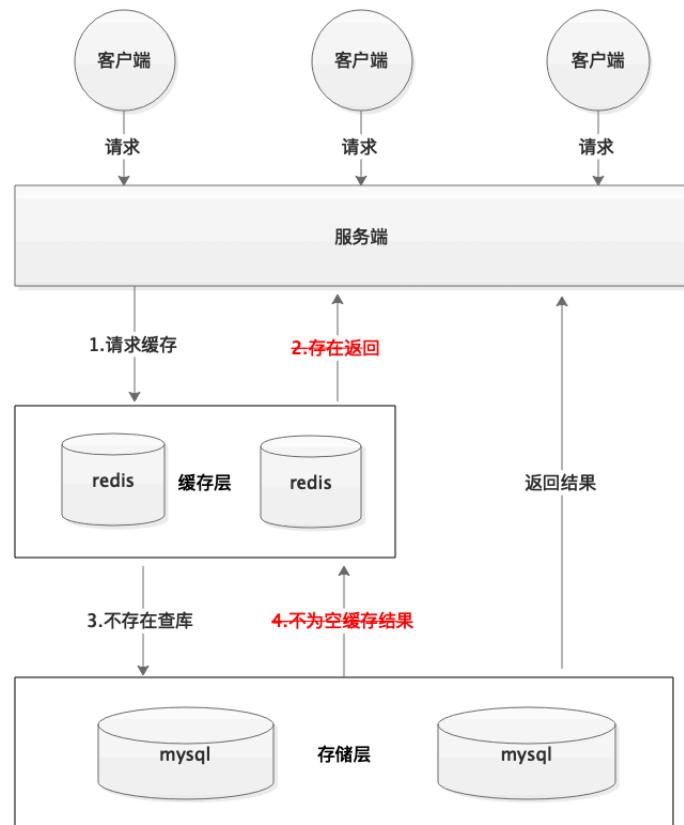
什么是缓存穿透？

缓存穿透说简单点就是大量请求的 key 根本不存在于缓存中，导致请求直接到了数据库上，根本没有经过缓存这一层。举个例子：某个黑客故意制造我们缓存中不存在的 key 发起大量请求，导致大量请求落到数据库。下面用图片展示一下(这两张图片不是我画的，为了省事直接在网上找的，这里说明一下)：

正常缓存处理流程：



缓存穿透情况处理流程：



一般MySQL 默认的最大连接数在 150 左右，这个可以通过 `show variables like '%max_connections%'` 命令来查看。最大连接数一个还只是一个指标，cpu，内存，磁盘，网络等无力条件都是其运行指标，这些指标都会限制其并发能力！所以，一般 3000 个并发请求就能打死大部分数据库了。

有哪些解决办法？

最基本的就是首先做好参数校验，一些不合法的参数请求直接抛出异常信息返回给客户端。比如查询的数据库 `id` 不能小于 0、传入的邮箱格式不对的时候直接返回错误消息给客户端等等。

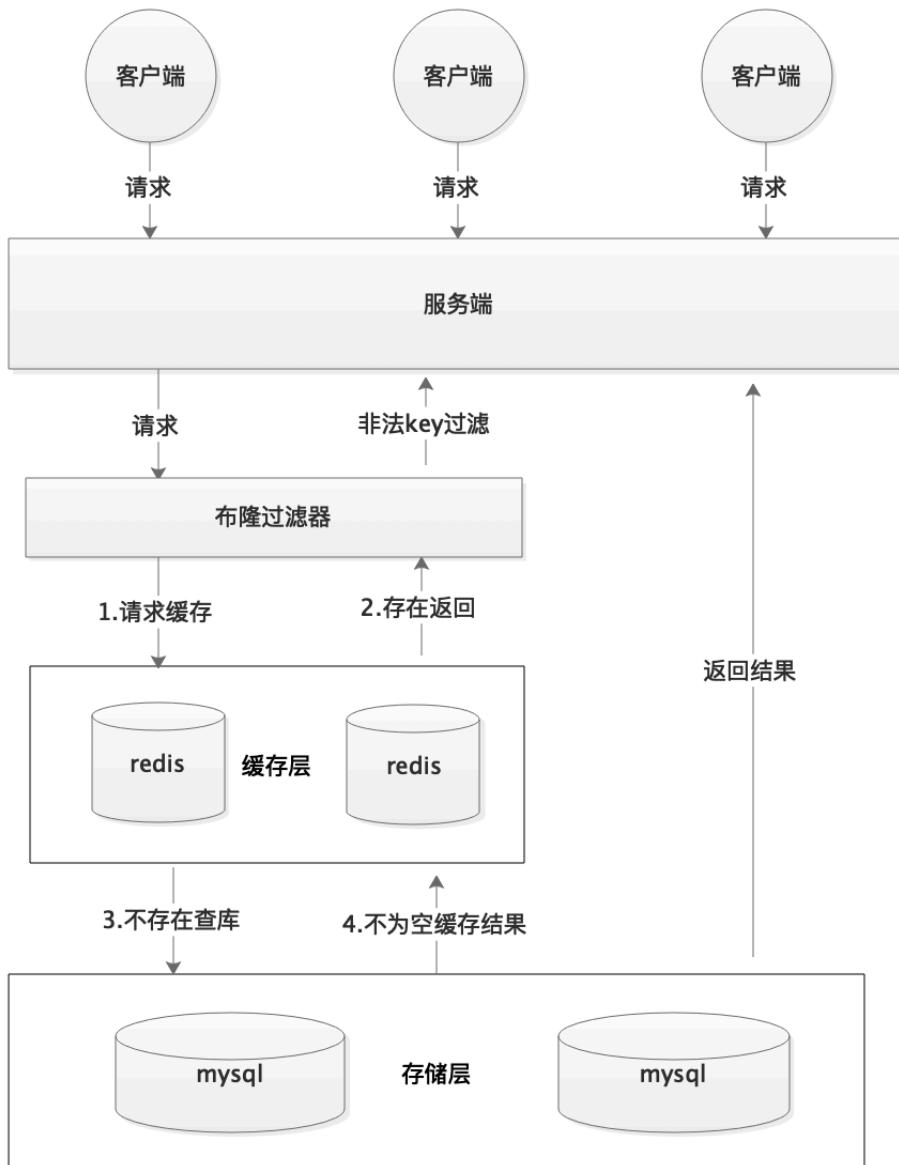
1) 缓存无效 key：如果缓存和数据库都查不到某个 `key` 的数据就写一个到 `redis` 中去并设置过期时间，具体命令如下：`SET key value EX 10086`。这种方式可以解决请求的 `key` 变化不频繁的情况，如果黑客恶意攻击，每次构建不同的请求`key`，会导致 `redis` 中缓存大量无效的 `key`。很明显，这种方案并不能从根本上解决此问题。如果非要用这种方式来解决穿透问题的话，尽量将无效的 `key` 的过期时间设置短一点比如 1 分钟。

另外，这里多说一嘴，一般情况下我们是这样设计 `key` 的：`表名:列名:主键名:主键值`。

如果用 Java 代码展示的话，差不多是下面这样的：

```
public Object getObjectInclNullById(Integer id) {
    // 从缓存中获取数据
    Object cacheValue = cache.get(id);
    // 缓存为空
    if (cacheValue == null) {
        // 从数据库中获取
        Object storageValue = storage.get(key);
        // 缓存空对象
        cache.set(key, storageValue);
        // 如果存储数据为空，需要设置一个过期时间(300秒)
        if (storageValue == null) {
            // 必须设置过期时间，否则有被攻击的风险
            cache.expire(key, 60 * 5);
        }
        return storageValue;
    }
    return cacheValue;
}
```

2) 布隆过滤器：布隆过滤器是一个非常神奇的数据结构，通过它我们可以非常方便地判断一个给定数据是否存在与海量数据中。我们需要的就是判断 `key` 是否合法，有没有感觉布隆过滤器就是我们想要找的那个“人”。具体是这样做的：把所有可能存在的请求的值都存放在布隆过滤器中，当用户请求过来，我会先判断用户发来的请求的值是否存在于布隆过滤器中。不存在的话，直接返回请求参数错误信息给客户端，存在的话才会走下面的流程。总结一下就是下面这张图（这张图片不是我画的，为了省事直接在网上找的）：



更多关于布隆过滤器的内容可以看我的这篇原创：[《不了解布隆过滤器？一文给你整的明明白白！》](#)，强烈推荐，个人感觉网上应该找不到总结的这么明明白白的文章了。

4.2.10 如何解决 Redis 的并发竞争 Key 问题

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，但是最后执行的顺序和我们期望的顺序不同，这样也就导致了结果的不同！

推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。（如果不存在 Redis 的并发竞争 Key 问题，不要使用分布式锁，这样会影响性能）

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。完成业务流程后，删除对应的子节点释放锁。

在实践中，当然是从以可靠性为主。所以首推Zookeeper。

参考：

- <https://www.jianshu.com/p/8bddd381de06>

4.2.11 如何保证缓存与数据库双写时的数据一致性？

一般情况下我们都是这样使用缓存的：先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。这种方式很明显会存在缓存和数据库的数据不一致的情况。

你只要用缓存，就可能会涉及到缓存与数据库双存储双写，你只要是双写，就一定会有数据一致性的
问题，那么你如何解决一致性问题？

一般来说，就是如果你的系统不是严格要求缓存+数据库必须一致性的话，缓存可以稍微的跟数据库偶
尔有不一致的情况，最好不要做这个方案，读请求和写请求串行化，串到一个内存队列里去，这样就可
以保证一定不会出现不一致的情况

串行化之后，就会导致系统的吞吐量会大幅度的降低，用比正常情况下多几倍的机器去支撑线上的一个
请求。

更多内容可以查看：<https://github.com/doocs/advanced-java/blob/master/docs/high-concurrency/redis-consistence.md>

参考： Java工程师面试突击第1季（可能是史上最好的Java面试突击课程） -中华石杉老师！公众号后台回复关键字“1”即可获取该视频内容。

参考

- 《Redis开发与运维》
- Redis 命令总结：<http://redisdoc.com/string/set.html>

五 常用框架面试题总结

5.1 Spring面试题总结

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原
创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

这篇文章主要是想通过一些问题，加深大家对于 Spring 的理解，所以不会涉及太多的代码！这篇文章
整理了挺长时间，下面的很多问题我自己在使用 Spring 的过程中也并没有注意，自己也是临时查阅了
很多资料和书籍补上的。网上也有一些很多关于 Spring 常见问题/面试题整理的文章，我感觉大部分
都是互相 copy，而且很多问题也不是很好，有些回答也存在问题。所以，自己花了一周的业余时间整
理了一下，希望对大家有帮助。

5.1.1. 什么是 Spring 框架？

Spring 是一种轻量级开发框架，旨在提高开发人员的开发效率以及系统的可维护性。Spring 官网：<https://spring.io/>。

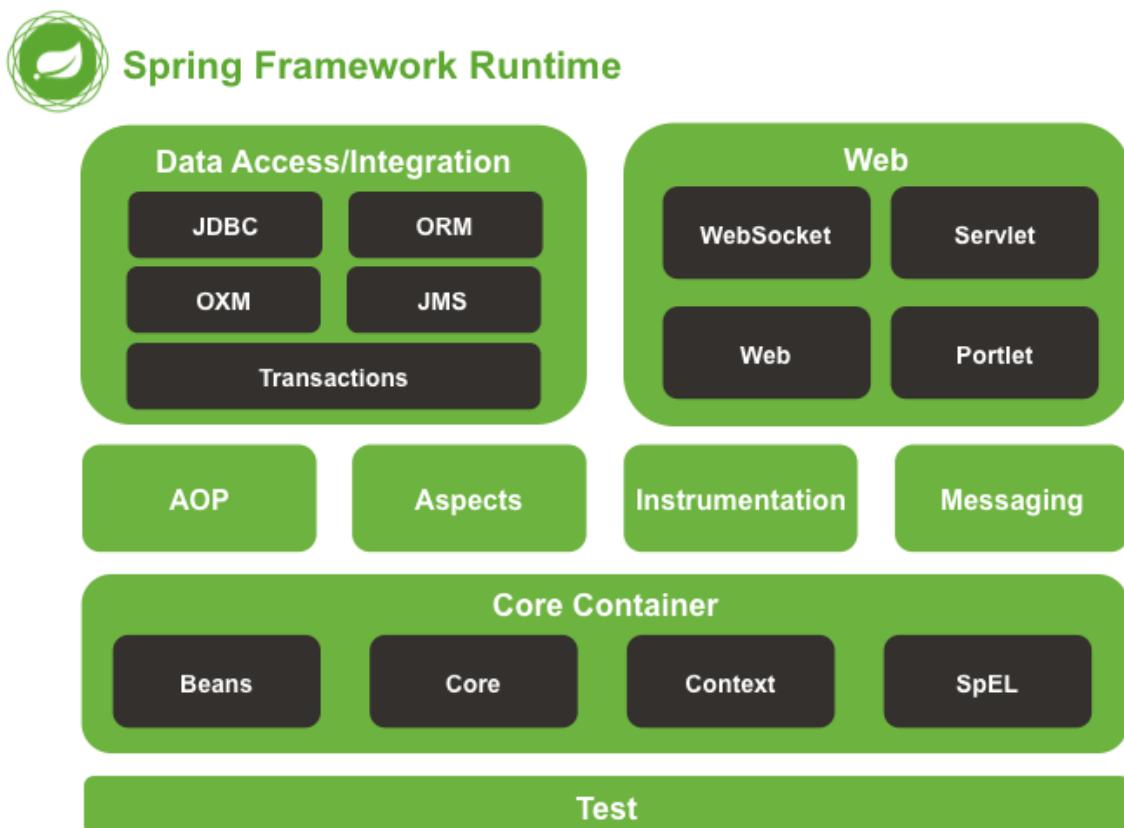
我们一般说 Spring 框架指的都是 Spring Framework，它是很多模块的集合，使用这些模块可以很方便地协助我们进行开发。这些模块是：核心容器、数据访问/集成、Web、AOP（面向切面编程）、工具、消息和测试模块。比如：Core Container 中的 Core 组件是 Spring 所有组件的核心，Beans 组件和 Context 组件是实现 IOC 和依赖注入的基础，AOP 组件用来实现面向切面编程。

Spring 官网列出的 Spring 的 6 个特征：

- **核心技术**：依赖注入(DI), AOP, 事件(events), 资源, i18n, 验证, 数据绑定, 类型转换, SpEL。
- **测试**：模拟对象, TestContext 框架, Spring MVC 测试, WebTestClient。
- **数据访问**：事务, DAO 支持, JDBC, ORM, 编组 XML。
- **Web 支持**：Spring MVC 和 Spring WebFlux Web 框架。
- **集成**：远程处理, JMS, JCA, JMX, 电子邮件, 任务, 调度, 缓存。
- **语言**：Kotlin, Groovy, 动态语言。

5.1.2 列举一些重要的 Spring 模块？

下图对应的是 Spring 4.x 版本。目前最新的 5.x 版本中 Web 模块的 Portlet 组件已经被废弃掉，同时增加了用于异步响应式处理的 WebFlux 组件。



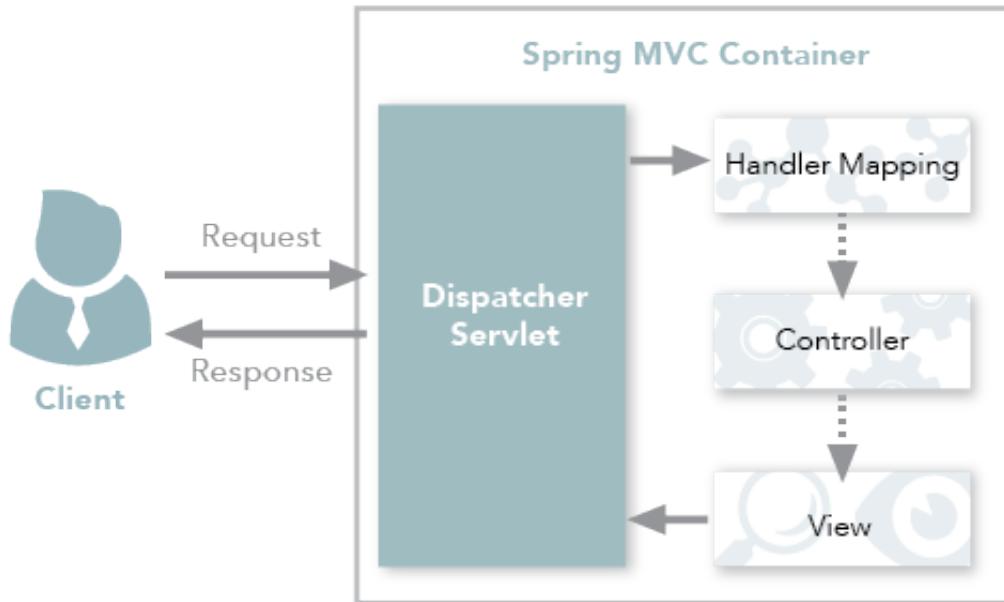
- **Spring Core**：基础，可以说 Spring 其他所有的功能都需要依赖于该类库。主要提供 IoC 依赖注入功能。
- **Spring Aspects**：该模块为与 AspectJ 的集成提供支持。
- **Spring AOP**：提供了面向切面的编程实现。
- **Spring JDBC**：Java 数据库连接。
- **Spring JMS**：Java 消息服务。
- **Spring ORM**：用于支持 Hibernate 等 ORM 工具。
- **Spring Web**：为创建 Web 应用程序提供支持。

- **Spring Test** : 提供了对 JUnit 和 TestNG 测试的支持。

5.1.3 @RestController vs @Controller

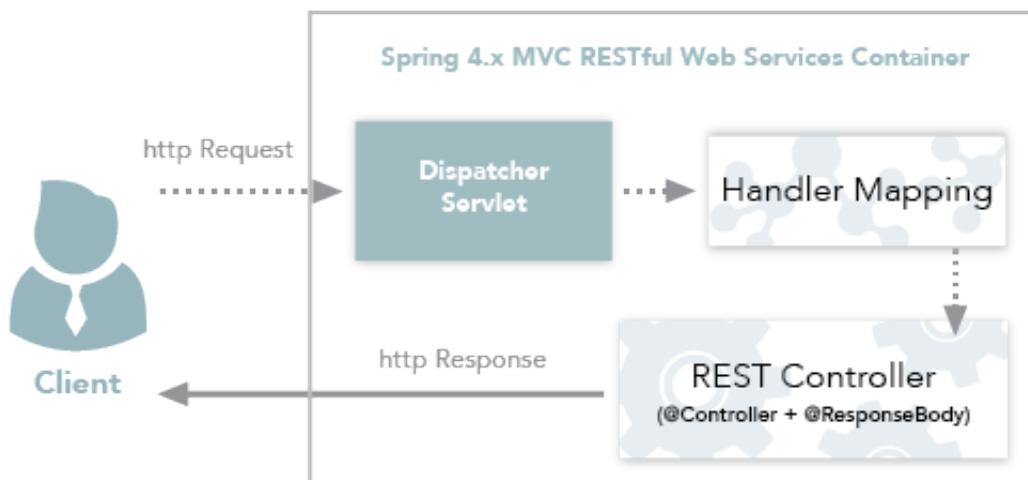
Controller 返回一个页面

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，对应于前后端不分离的情况。



@RestController 返回 JSON 或 XML 形式数据

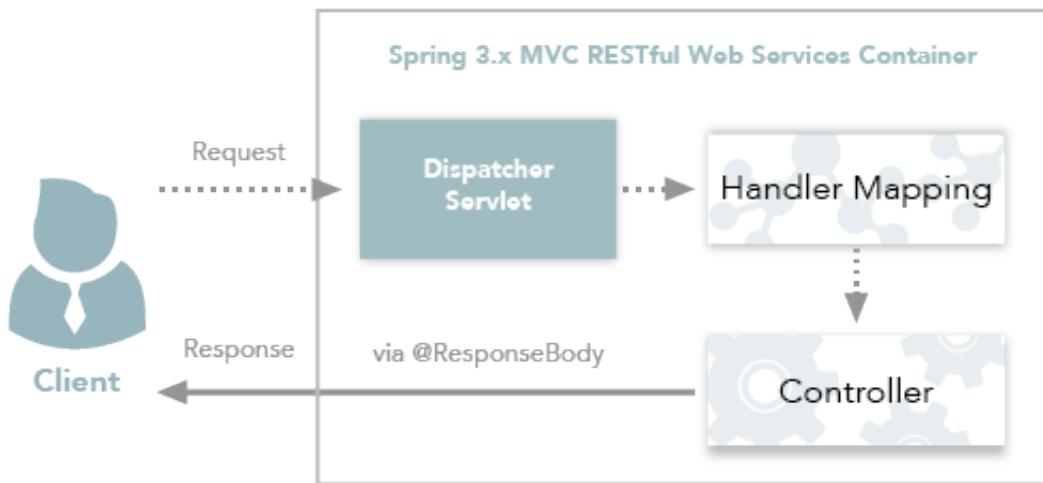
但 `@RestController` 只返回对象，对象数据直接以 JSON 或 XML 形式写入 HTTP 响应(Response) 中，这种情况属于 RESTful Web 服务，这也是目前日常开发所接触的最常用的情况（前后端分离）。



@Controller + @ResponseBody 返回 JSON 或 XML 形式数据

如果你需要在 Spring 4 之前开发 RESTful Web 服务的话，你需要使用 `@Controller` 并结合 `@ResponseBody` 注解，也就是说 `@Controller + @ResponseBody = @RestController` (Spring 4 之后新加的注解)。

`@ResponseBody` 注解的作用是将 `Controller` 的方法返回的对象通过适当的转换器转换为指定的格式之后，写入到HTTP响应(Response)对象的 body 中，通常用来返回 JSON 或者 XML 数据，返回 JSON 数据的情况比较多。



Reference:

- <https://dzone.com/articles/spring-framework-restcontroller-vs-controller> (图片来源)
- <https://javarevisited.blogspot.com/2017/08/difference-between-restcontroller-and-controller-annotations-spring-mvc-rest.html?m=1>

5.1.4 Spring IOC & AOP

谈谈自己对于 Spring IoC 和 AOP 的理解

IoC

IoC (Inverse of Control:控制反转) 是一种设计思想，就是 将原本在程序中手动创建对象的控制权，交由Spring框架来管理。 IoC 在其他语言中也有应用，并非 Spring 特有。 IoC 容器是 Spring 用来实现 IoC 的载体， IoC 容器实际上就是个Map (key, value) ,Map 中存放的是各种对象。

将对象之间的相互依赖关系交给 IoC 容器来管理，并由 IoC 容器完成对象的注入。这样可以很大程度上简化应用的开发，把应用从复杂的依赖关系中解放出来。 IoC 容器就像是一个工厂一样，当我们需要创建一个对象的时候，只需要配置好配置文件/注解即可，完全不用考虑对象是如何被创建出来的。在实际项目中一个 Service 类可能有几百甚至上千个类作为它的底层，假如我们需要实例化这个 Service，你可能要每次都要搞清这个 Service 所有底层类的构造函数，这可能会把人逼疯。如果利用 IoC 的话，你只需要配置好，然后在需要的地方引用就行了，这大大增加了项目的可维护性且降低了开发难度。

Spring 时代我们一般通过 XML 文件来配置 Bean，后来开发人员觉得 XML 文件来配置不太好，于是 SpringBoot 注解配置就慢慢开始流行起来。

推荐阅读：<https://www.zhihu.com/question/23277575/answer/169698662>

Spring IoC的初始化过程：



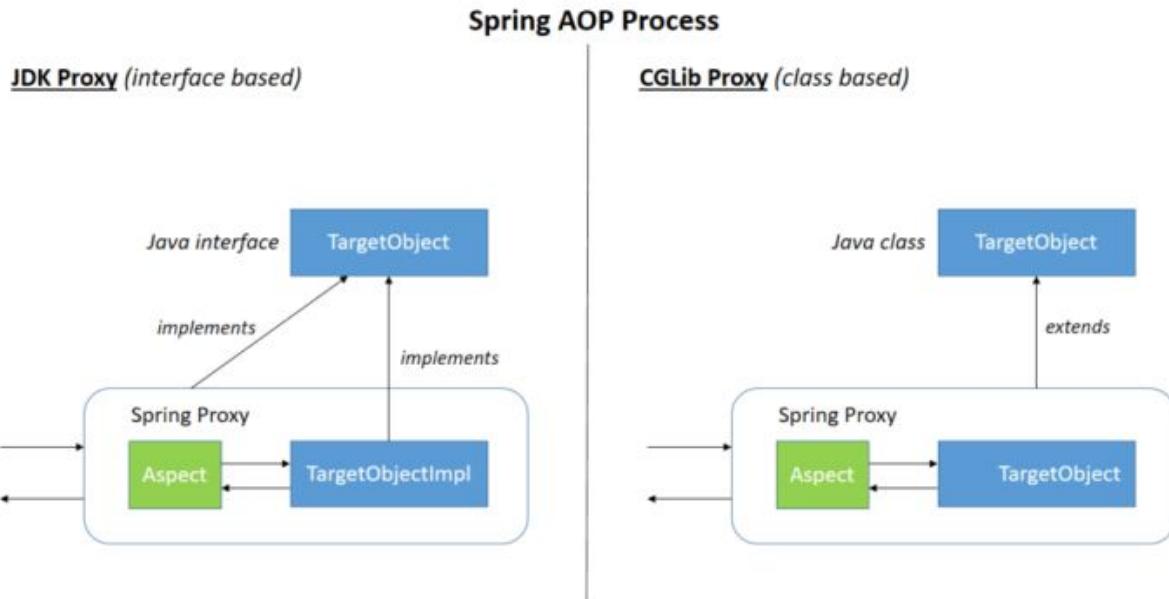
IoC源码阅读

- <https://javadoop.com/post/spring-ioc>

AOP

AOP(Aspect-Oriented Programming:面向切面编程)能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用JDK Proxy，去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理了，这时候 Spring AOP会使用Cglib，这时候Spring AOP会使用 Cglib 生成一个被代理对象的子类来作为代理，如下图所示：



当然你也可以使用 AspectJ，Spring AOP 已经集成了AspectJ，AspectJ 应该算的上是 Java 生态系统中最完整的 AOP 框架了。

使用 AOP 之后我们可以把一些通用功能抽象出来，在需要用到的地方直接使用即可，这样大大简化了代码量。我们需要增加新功能时也方便，这样也提高了系统扩展性。日志功能、事务管理等等场景都用到了 AOP 。

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 应该算的上是 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。

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

5.1.5 Spring bean

Spring 中的 bean 的作用域有哪些？

- singleton：唯一 bean 实例，Spring 中的 bean 默认都是单例的。
- prototype：每次请求都会创建一个新的 bean 实例。
- request：每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request 内有效。
- session：每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。
- global-session：全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。Portlet是能够生成语义代码(例如：HTML)片段的小型Java Web插件。它们基于portlet容器，可以像servlet一样处理HTTP请求。但是，与 servlet 不同，每个 portlet 都有不同的会话

Spring 中的单例 bean 的线程安全问题了解吗？

大部分时候我们并没有在系统中使用多线程，所以很少有人会关注这个问题。单例 bean 存在线程问题，主要是因为当多个线程操作同一个对象的时候，对这个对象的非静态成员变量的写操作会存在线程安全问题。

常见的有两种解决办法：

1. 在Bean对象中尽量避免定义可变的成员变量（不太现实）。
2. 在类中定义一个ThreadLocal成员变量，将需要的可变成员变量保存在 ThreadLocal 中（推荐的一种方式）。

@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侦测以及自动装配到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。

`@Bean`注解使用示例：

```
@Configuration  
public class AppConfig {  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

上面的代码相当于下面的 xml 配置

```
<beans>
    <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

下面这个例子是通过 `@Component` 无法实现的。

```
@Bean
public OneService getService(status) {
    case (status) {
        when 1:
            return new serviceImpl1();
        when 2:
            return new serviceImpl2();
        when 3:
            return new serviceImpl3();
    }
}
```

将一个类声明为Spring的 bean 的注解有哪些？

我们一般使用 `@Autowired` 注解自动装配 bean，要想把类标识成可用于 `@Autowired` 注解自动装配的 bean 的类，采用以下注解可实现：

- `@Component`：通用的注解，可标注任意类为 Spring 组件。如果一个 Bean 不知道属于哪个层，可以使用 `@Component` 注解标注。
- `@Repository`：对应持久层即 Dao 层，主要用于数据库相关操作。
- `@Service`：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao 层。
- `@Controller`：对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

Spring 中的 bean 生命周期？

这部分网上有很多文章都讲到了，下面的内容整理自：<https://yemengying.com/2016/07/14/spring-bean-life-cycle/>，除了这篇文章，再推荐一篇很不错的文章：<https://www.cnblogs.com/zrtqsk/p/3735273.html>。

- Bean 容器找到配置文件中 Spring Bean 的定义。
- Bean 容器利用 Java Reflection API 创建一个 Bean 的实例。
- 如果涉及到一些属性值 利用 `set()` 方法设置一些属性值。
- 如果 Bean 实现了 `BeanNameAware` 接口，调用 `setBeanName()` 方法，传入 Bean 的名字。
- 如果 Bean 实现了 `BeanClassLoaderAware` 接口，调用 `setBeanClassLoader()` 方法，传入 `ClassLoader` 对象的实例。
- 与上面的类似，如果实现了其他 `*.Aware` 接口，就调用相应的方法。
- 如果有和加载这个 Bean 的 Spring 容器相关的 `BeanPostProcessor` 对象，执行 `postProcessBeforeInitialization()` 方法
- 如果 Bean 实现了 `InitializingBean` 接口，执行 `afterPropertiesSet()` 方法。
- 如果 Bean 在配置文件中的定义包含 `init-method` 属性，执行指定的方法。

- 如果有和加载这个 Bean 的 Spring 容器相关的 BeanPostProcessor 对象，执行 postProcessAfterInitialization() 方法
- 当要销毁 Bean 的时候，如果 Bean 实现了 DisposableBean 接口，执行 destroy() 方法。
- 当要销毁 Bean 的时候，如果 Bean 在配置文件中的定义包含 destroy-method 属性，执行指定的方法。

图示：

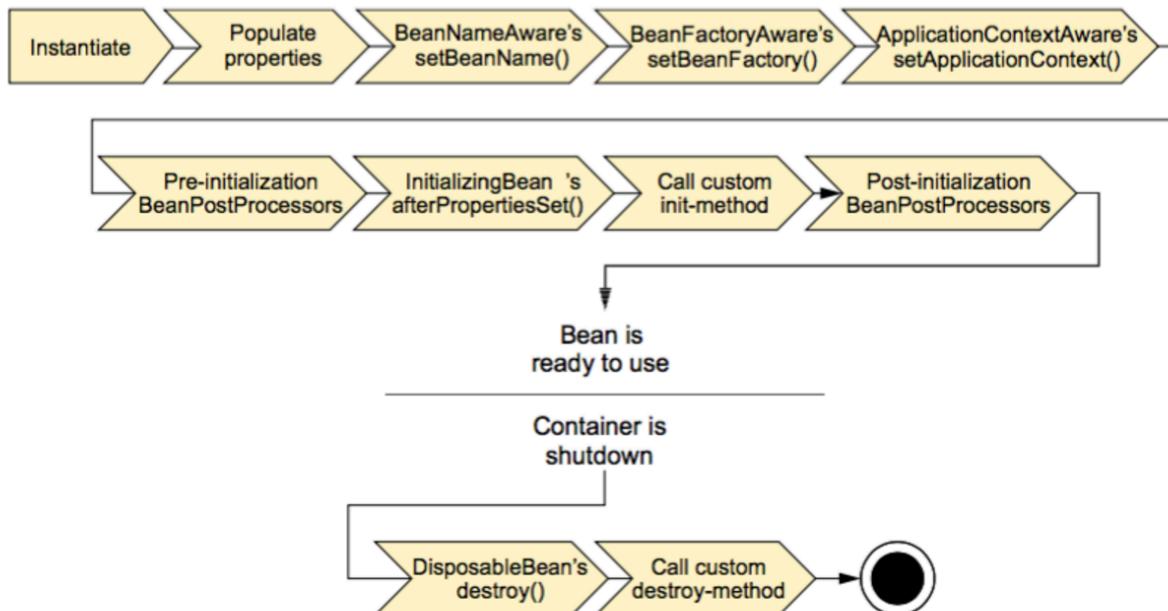
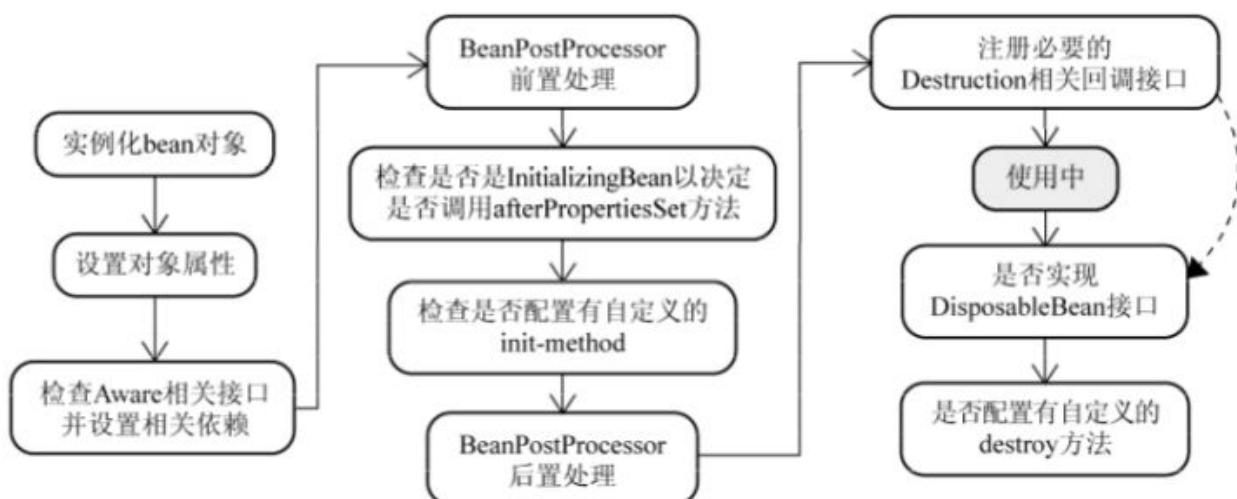


Figure 1.5 A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

与之比较类似的中文版本：



5.1.6 Spring MVC

说说自己对于 Spring MVC 了解？

谈到这个问题，我们不得不提之前 Model1 和 Model2 这两个没有 Spring MVC 的时代。

- Model1 时代：很多学 Java 后端比较晚的朋友可能并没有接触过 Model1 模式下的 JavaWeb

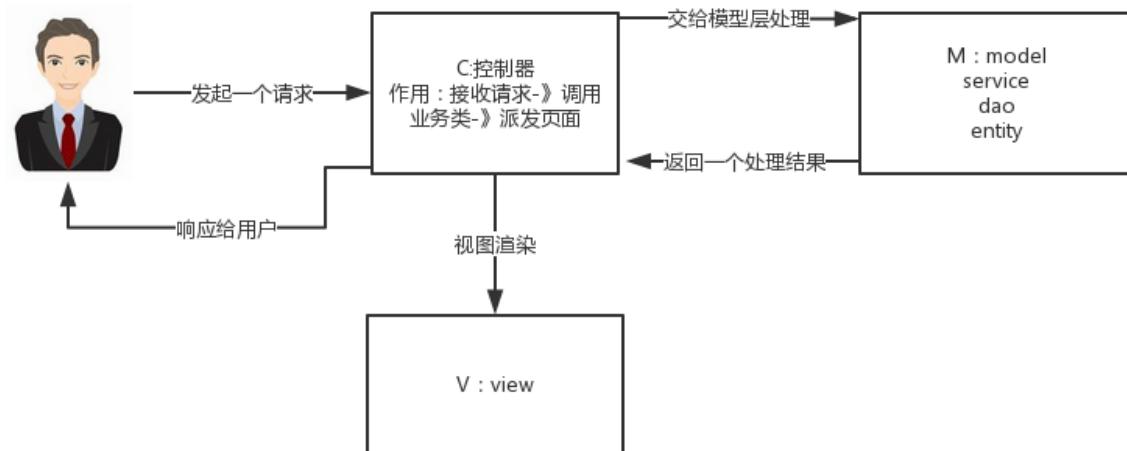
应用开发。在 Model1 模式下，整个 Web 应用几乎全部用 JSP 页面组成，只用少量的 JavaBean 来处理数据库连接、访问等操作。这个模式下 JSP 即是控制层又是表现层。显而易见，这种模式存在很多问题。比如①将控制逻辑和表现逻辑混杂在一起，导致代码重用率极低；②前端和后端相互依赖，难以进行测试并且开发效率极低；

- **Model2 时代**：学过 Servlet 并做过相关 Demo 的朋友应该了解“Java Bean(Model)+ JSP (View,) +Servlet (Controller) ”这种开发模式，这就是早期的 JavaWeb MVC 开发模式。
Model:系统涉及的数据，也就是 dao 和 bean。View：展示模型中的数据，只是用来展示。
Controller：处理用户请求都发送给，返回数据给 JSP 并展示给用户。

Model2 模式下还存在很多问题，Model2 的抽象和封装程度还远远不够，使用 Model2 进行开发时不可避免地会重复造轮子，这就大大降低了程序的可维护性和复用性。于是很多 JavaWeb 开发相关的 MVC 框架应运而生比如 Struts2，但是 Struts2 比较笨重。随着 Spring 轻量级开发框架的流行，Spring 生态圈出现了 Spring MVC 框架，Spring MVC 是当前最优秀的 MVC 框架。相比于 Struts2，Spring MVC 使用更加简单和方便，开发效率更高，并且 Spring MVC 运行速度更快。

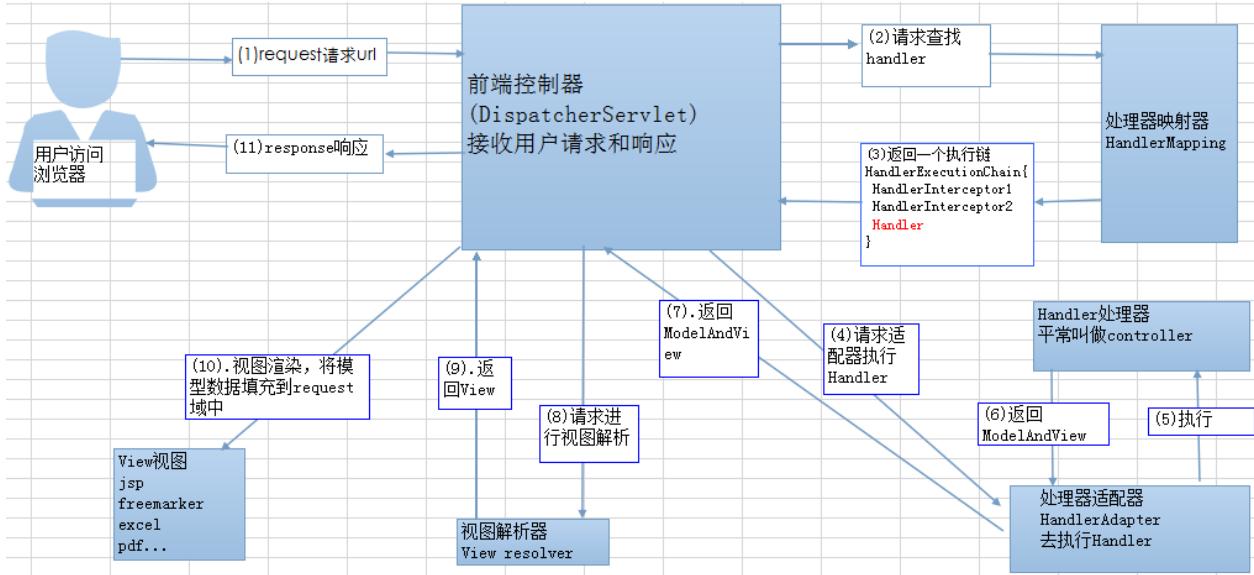
MVC 是一种设计模式，Spring MVC 是一款很优秀的 MVC 框架。Spring MVC 可以帮助我们进行更简洁的 Web 层的开发，并且它天生与 Spring 框架集成。Spring MVC 下我们一般把后端项目分为 Service 层（处理业务）、Dao 层（数据库操作）、Entity 层（实体类）、Controller 层（控制层，返回数据给前台页面）。

Spring MVC 的简单原理图如下：



SpringMVC 工作原理了解吗？

原理如下图所示：



上图的一个笔误的小问题：Spring MVC 的入口函数也就是前端控制器 `DispatcherServlet` 的作用是接收请求，响应结果。

流程说明（重要）：

1. 客户端（浏览器）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. 解析到对应的 `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` 会根据 `Handler` 来调用真正的处理器来处理请求，并处理相应的业务逻辑。
5. 处理器处理完业务后，会返回一个 `ModelAndView` 对象，`Model` 是返回的数据对象，`View` 是个逻辑上的 `View`。
6. `ViewResolver` 会根据逻辑 `View` 查找实际的 `View`。
7. `DispatcherServlet` 把返回的 `Model` 传给 `View`（视图渲染）。
8. 把 `View` 返回给请求者（浏览器）

5.1.7 Spring 框架中用到了哪些设计模式？

关于下面一些设计模式的详细介绍，可以看笔者前段时间的原创文章《面试官：“谈谈Spring中都用到了那些设计模式？”》。

- 工厂设计模式：Spring 使用工厂模式通过 `BeanFactory`、`ApplicationContext` 创建 bean 对象。
- 代理设计模式：Spring AOP 功能的实现。
- 单例设计模式：Spring 中的 Bean 默认都是单例的。
- 包装器设计模式：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。这种模式让我们可以根据客户的需求能够动态切换不同的数据源。
- 观察者模式：Spring 事件驱动模型就是观察者模式很经典的一个应用。
- 适配器模式：Spring AOP 的增强或通知(Advice)使用到了适配器模式、spring MVC 中也是用到了适配器模式适配 `Controller`。
-

5.1.8 Spring 事务

Spring 管理事务的方式有几种？

1. 编程式事务，在代码中硬编码。（不推荐使用）
2. 声明式事务，在配置文件中配置（推荐使用）

声明式事务又分为两种：

1. 基于XML的声明式事务
2. 基于注解的声明式事务

Spring 事务中的隔离级别有哪几种？

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

- **TransactionDefinition.ISOLATION_DEFAULT:** 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ 隔离级别 Oracle 默认采用的 READ_COMMITTED 隔离级别。
- **TransactionDefinition.ISOLATION_READ_UNCOMMITTED:** 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
- **TransactionDefinition.ISOLATION_READ_COMMITTED:** 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- **TransactionDefinition.ISOLATION_REPEATABLE_READ:** 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。
- **TransactionDefinition.ISOLATION_SERIALIZABLE:** 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

Spring 事务中哪几种事务传播行为？

支持当前事务的情况：

- **TransactionDefinition.PROPAGATION_REQUIRED:** 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。
- **TransactionDefinition.PROPAGATION_SUPPORTS:** 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。
- **TransactionDefinition.PROPAGATION_MANDATORY:** 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。（mandatory：强制性）

不支持当前事务的情况：

- **TransactionDefinition.PROPAGATIONQUIRES_NEW:** 创建一个新的事务，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED:** 以非事务方式运行，如果当前存在事务，则把当前事务挂起。
- **TransactionDefinition.PROPAGATION_NEVER:** 以非事务方式运行，如果当前存在事务，则抛出异常。

其他情况：

- **TransactionDefinition.PROPAGATION_NESTED:** 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于 TransactionDefinition.PROPAGATION_REQUIRED。

@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。

关于 `@Transactional` 注解推荐阅读的文章：

- [透彻的掌握 Spring 中@Transactional 的使用](#)

5.1.9 JPA

如何使用JPA在数据库中非持久化一个字段？

假如我们有下面一个类：

```
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 `secrect` 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
```

一般使用后面两种方式比较多，我个人使用注解的方式比较多。

参考

- 《Spring 技术内幕》
- <http://www.cnblogs.com/wmmykxz/p/8820371.html>
- <https://www.journaldev.com/2696/spring-interview-questions-and-answers>
- <https://www.edureka.co/blog/interview-questions/spring-interview-questions/>
- <https://www.cnblogs.com/clwydjgs/p/9317849.html>
- <https://howtodoinjava.com/interview-questions/top-spring-interview-questions-with-answers/>
- <http://www.tomaszezula.com/2014/02/09/spring-series-part-5-component-vs-bean/>
- <https://stackoverflow.com/questions/34172888/difference-between-bean-and-autowired>

公众号

如果大家想要实时关注我更新的文章以及分享的干货的话，可以关注我的公众号。

《Java面试突击》：由本文档衍生的专为面试而生的《Java面试突击》V2.0 PDF 版本[公众号后台回复 "Java面试突击"](#) 即可免费领取！

Java工程师必备学习资源：一些Java工程师常用学习资源[公众号后台回复关键字“1”](#) 即可免费无套路获取。



5.2 MyBatis面试题总结

本篇文章是JavaGuide收集自网络，原出处不明。

Mybatis 技术内幕系列博客，从原理和源码角度，介绍了其内部实现细节，无论是写的好与不好，我确实是用心写了，由于并不是介绍如何使用 Mybatis 的文章，所以，一些参数使用细节略掉了，我们的目标是介绍 Mybatis 的技术架构和重要组成部分，以及基本运行原理。

博客写的很辛苦，但是写出来却不一定好看，所谓开始很兴奋，过程很痛苦，结束很遗憾。要求不高，只要读者能从系列博客中，学习到一点其他博客所没有的技术点，作为作者，我就很欣慰了，我也读别人写的博客，通常对自己当前研究的技术，是很有帮助的。

尽管还有很多可写的内容，但是，我认为再写下去已经没有意义，任何其他小的功能点，都是在已经介绍的基本框架和基本原理下运行的，只有结束，才能有新的开始。写博客也积攒了一些经验，源码多了感觉就是复制黏贴，源码少了又觉得是空谈原理，将来再写博客，我希望是“精炼博文”，好读好懂美观读起来又不累，希望自己能再写一部开源分布式框架原理系列博客。

有胆就来，我出几道 Mybatis 面试题，看你能回答上来几道（都是我出的，可不是网上找的）。

5.2.1 #{}和\${}的区别是什么？

注：这道题是面试官面试我同事的。

答：

- \${} 是 Properties 文件中的变量占位符，它可以用于标签属性值和 sql 内部，属于静态文本替换，比如 \${driver} 会被静态替换为 com.mysql.jdbc.Driver。
- #{} 是 sql 的参数占位符，Mybatis 会将 sql 中的 #{} 替换为 ? 号，在 sql 执行前会使用 PreparedStatement 的参数设置方法，按序给 sql 的 ? 号占位符设置参数值，比如 ps.setInt(0, parameterValue), #{{item.name}} 的取值方式为使用反射从参数对象中获取 item 对象的 name 属性值，相当于 param.getItem().getName()。

5.2.2 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？

注：这道题是京东面试官面试我时问的。

答：还有很多其他的标

签，<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态 sql 的 9 个标签，trim|where|set|foreach|if|choose|when|otherwise|bind 等，其中为 sql 片段标签，通过 <include> 标签引入 sql 片段，<selectKey> 为不支持自增的主键生成策略标签。

5.2.3 最佳实践中，通常一个 Xml 映射文件，都会写一个 Dao 接口与之对应，请问，这个 Dao 接口的工作原理是什么？Dao 接口里的方法，参数不同时，方法能重载吗？

注：这道题也是京东面试官面试我时问的。

答：Dao 接口，就是人们常说的 Mapper 接口，接口的全限名，就是映射文件中的 namespace 的值，接口的方法名，就是映射文件中 MappedStatement 的 id 值，接口方法内的参数，就是传递给 sql 的参数。Mapper 接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为 key 值，可唯一定位一个 MappedStatement，举

例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到 namespace 为 com.mybatis3.mappers.StudentDao 下面 id = findStudentById 的 MappedStatement。在 Mybatis 中，每一个 <select>、<insert>、<update>、<delete> 标签，都会被解析为一个 MappedStatement 对象。

Dao 接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

Dao 接口的工作原理是 JDK 动态代理，Mybatis 运行时会使用 JDK 动态代理为 Dao 接口生成代理 proxy 对象，代理对象 proxy 会拦截接口方法，转而执行 MappedStatement 所代表的 sql，然后将 sql 执行结果返回。

5.2.4 Mybatis 是如何进行分页的？分页插件的原理是什么？

注：我出的。

答：Mybatis 使用 RowBounds 对象进行分页，它是针对 ResultSet 结果集执行的内存分页，而非物理分页，可以在 sql 内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

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。

举例：`select _ from student`，拦截 sql 后重写为：`select t._ from (select /* from student) t limit 0, 10`

5.2.5 简述 Mybatis 的插件运行原理，以及如何编写一个插件。

注：我出的。

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。

5.2.6 Mybatis 执行批量插入，能返回数据库主键列表吗？

注：我出的。

答：能，JDBC 都能，Mybatis 当然也能。

5.2.7 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？

注：我出的。

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。

5.2.8 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？

注：我出的。

答：第一种是使用 `<resultMap>` 标签，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。

5.2.9 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。

注：我出的。

答：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，其实就是一对一查询，只需要把 `selectOne()` 修改为 `selectList()` 即可；多对多查询，其实就是一对多查询，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 `<resultMap>` 标签内的 `<id>` 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，`<id>` 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 |
```

5.2.10 Mybatis 是否支持延迟加载？如果支持，它的实现原理是什么？

注：我出的。

答：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。

5.2.11 Mybatis 的 Xml 映射文件中，不同的 Xml 映射文件，id 是否可以重复？

注：我出的。

答：不同的 Xml 映射文件，如果配置了 `namespace`，那么 `id` 可以重复；如果没有配置 `namespace`，那么 `id` 不能重复；毕竟 `namespace` 不是必须的，只是最佳实践而已。

原因就是 `namespace+id` 是作为 `Map<String, MappedStatement>` 的 key 使用的，如果没有 `namespace`，就剩下 `id`，那么，`id` 重复会导致数据互相覆盖。有了 `namespace`，自然 `id` 就可以重复，`namespace` 不同，`namespace+id` 自然也就不同。

5.2.12 Mybatis 中如何执行批处理？

注：我出的。

答：使用 BatchExecutor 完成批处理。

5.2.13 Mybatis 都有哪些 Executor 执行器？它们之间的区别是什么？

注：我出的

答：Mybatis 有三种基本的 Executor 执行器，SimpleExecutor、ReuseExecutor、BatchExecutor。

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。

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<String, Statement>内，供下一次使用。简言之，就是重复使用 Statement 对象。

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。

作用范围：Executor 的这些特点，都严格限制在 SqlSession 生命周期范围内。

5.2.14 Mybatis 中如何指定使用哪一种 Executor 执行器？

注：我出的

答：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。

5.2.15 Mybatis 是否可以映射 Enum 枚举类？

注：我出的

答：Mybatis 可以映射枚举类，不单可以映射枚举类，Mybatis 可以映射任何对象到表的一列上。映射方式为自定义一个 TypeHandler，实现 TypeHandler 的 setParameter() 和 getResult() 接口方法。TypeHandler 有两个作用，一是完成从 javaType 至 jdbcType 的转换，二是完成 jdbcType 至 javaType 的转换，体现为 setParameter() 和 getResult() 两个方法，分别代表设置 sql 问号占位符参数和获取列查询结果。

5.2.16 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？

注：我出的

答：虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的，但是，被引用的 B 标签依然可以定义在任何地方，Mybatis 都可以正确识别。

原理是，Mybatis 解析 A 标签，发现 A 标签引用了 B 标签，但是 B 标签尚未解析到，尚不存在，此时，Mybatis 会将 A 标签标记为未解析状态，然后继续解析余下的标签，包含 B 标签，待所有标签解析完毕，Mybatis 会重新解析那些被标记为未解析的标签，此时再解析 A 标签时，B 标签已经存在，A 标签也就可以正常解析完成了。

5.2.17 简述 Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系？

注：我出的

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，`<parameterMap>` 标签会被解析为 `ParameterMap` 对象，其每个子元素会被解析为 `ParameterMapping` 对象。`<resultMap>` 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 `<select>`、`<insert>`、`<update>`、`<delete>` 标签均会被解析为 `MappedStatement` 对象，标签内的 sql 会被解析为 `BoundSql` 对象。

5.2.18 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？

注：我出的

答：Hibernate 属于全自动 ORM 映射工具，使用 Hibernate 查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而 Mybatis 在查询关联对象或关联集合对象时，需要手动编写 sql 来完成，所以，称之为半自动 ORM 映射工具。

面试题看似都很简单，但是想要能正确回答上来，必定是研究过源码且深入的人，而不是仅会使用的人或者用的很熟的人，以上所有面试题及其答案所涉及的内容，在我的 Mybatis 系列博客中都有详细讲解和原理分析。-----

5.3 Kafka面试题总结

5.3.1 Kafka 是什么？主要应用场景有哪些？

Kafka 是一个分布式流式处理平台。这到底是什么意思呢？

流平台具有三个关键功能：

1. 消息队列：发布和订阅消息流，这个功能类似于消息队列，这也是 Kafka 也被归类为消息队列的原因。
2. 容错的持久方式存储记录消息流：Kafka 会把消息持久化到磁盘，有效避免了消息丢失的风险。
3. 流式处理平台：在消息发布的时候进行处理，Kafka 提供了一个完整的流式处理类库。

Kafka 主要有两大应用场景：

1. 消息队列：建立实时流数据管道，以可靠地在系统或应用程序之间获取数据。
2. 数据处理：构建实时的流数据处理程序来转换或处理数据流。

5.3.2 和其他消息队列相比，Kafka 的优势在哪里？

我们现在经常提到 Kafka 的时候就已经默认它是一个非常优秀的消息队列了，我们也会经常拿它给 RocketMQ、RabbitMQ 对比。我觉得 Kafka 相比其他消息队列主要的优势如下：

1. 极致的性能：基于 Scala 和 Java 语言开发，设计中大量使用了批量处理和异步的思想，最高可以每秒处理千万级别的消息。
2. 生态系统兼容性无可匹敌：Kafka 与周边生态系统的兼容性是最好的没有之一，尤其在大数据和流计算领域。

实际上在早期的时候 Kafka 并不是一个合格的消息队列，早期的 Kafka 在消息队列领域就像是一个衣衫褴褛的孩子一样，功能不完备并且有一些小问题比如丢失消息、不保证消息可靠性等等。当然，这也和 LinkedIn 最早开发 Kafka 用于处理海量的日志有很大关系，哈哈哈，人家本来最开始就不是为了作为消息队列滴，谁知道后面误打误撞在消息队列领域占据了一席之地。

随着后续的发展，这些短板都被 Kafka 逐步修复完善。所以，Kafka 作为消息队列不可靠这个说法已经过时！

5.3.3 队列模型了解吗？Kafka 的消息模型知道吗？

题外话：早期的 JMS 和 AMQP 属于消息服务领域权威组织所做的相关的标准，我在 [JavaGuide](#) 的《消息队列其实很简单》这篇文章中介绍过。但是，这些标准的进化跟不上消息队列的演进速度，这些标准实际上已经属于废弃状态。所以，可能存在的情况是：不同的消息队列都有自己的一套消息模型。

队列模型：早期的消息模型

队列模型

作者：SnailClimb
公众号&Github：JavaGuide



使用队列（Queue）作为消息通信载体，满足生产者与消费者模式，一条消息只能被一个消费者使用，未被消费的消息在队列中保留直到被消费或超时。比如：我们生产者发送 100 条消息的话，两个消费者来消费一般情况下两个消费者会按照消息发送的顺序各自消费一半（也就是你一个我一个的消费。）

队列模型存在的问题：

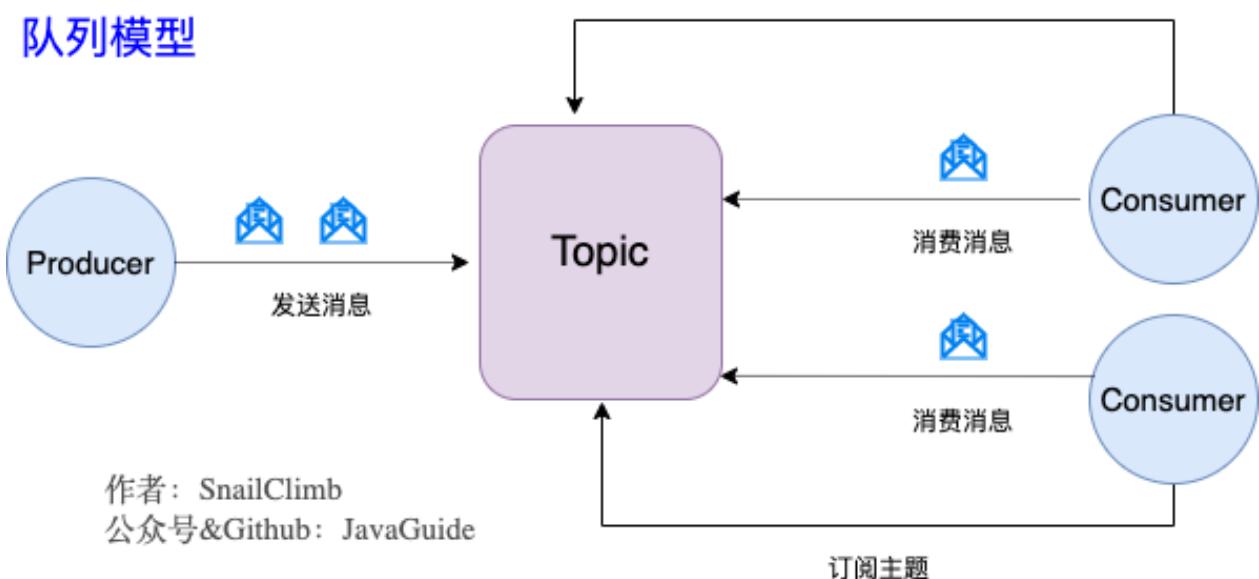
假如我们存在这样一种情况：我们需要将生产者产生的消息分发给多个消费者，并且每个消费者都能接收到完成的消息内容。

这种情况，队列模型就不好解决了。很多比较杠精的人就说：我们可以为每个消费者创建一个单独的队列，让生产者发送多份。这是一种非常愚蠢的做法，浪费资源不说，还违背了使用消息队列的目的。

发布-订阅模型：Kafka 消息模型

发布-订阅模型主要是为了解决队列模型存在的问题。

队列模型



发布订阅模型 (Pub-Sub) 使用主题 (Topic) 作为消息通信载体，类似于广播模式；发布者发布一条消息，该消息通过主题传递给所有的订阅者，在一条消息广播之后才订阅的用户则是收不到该条消息的。

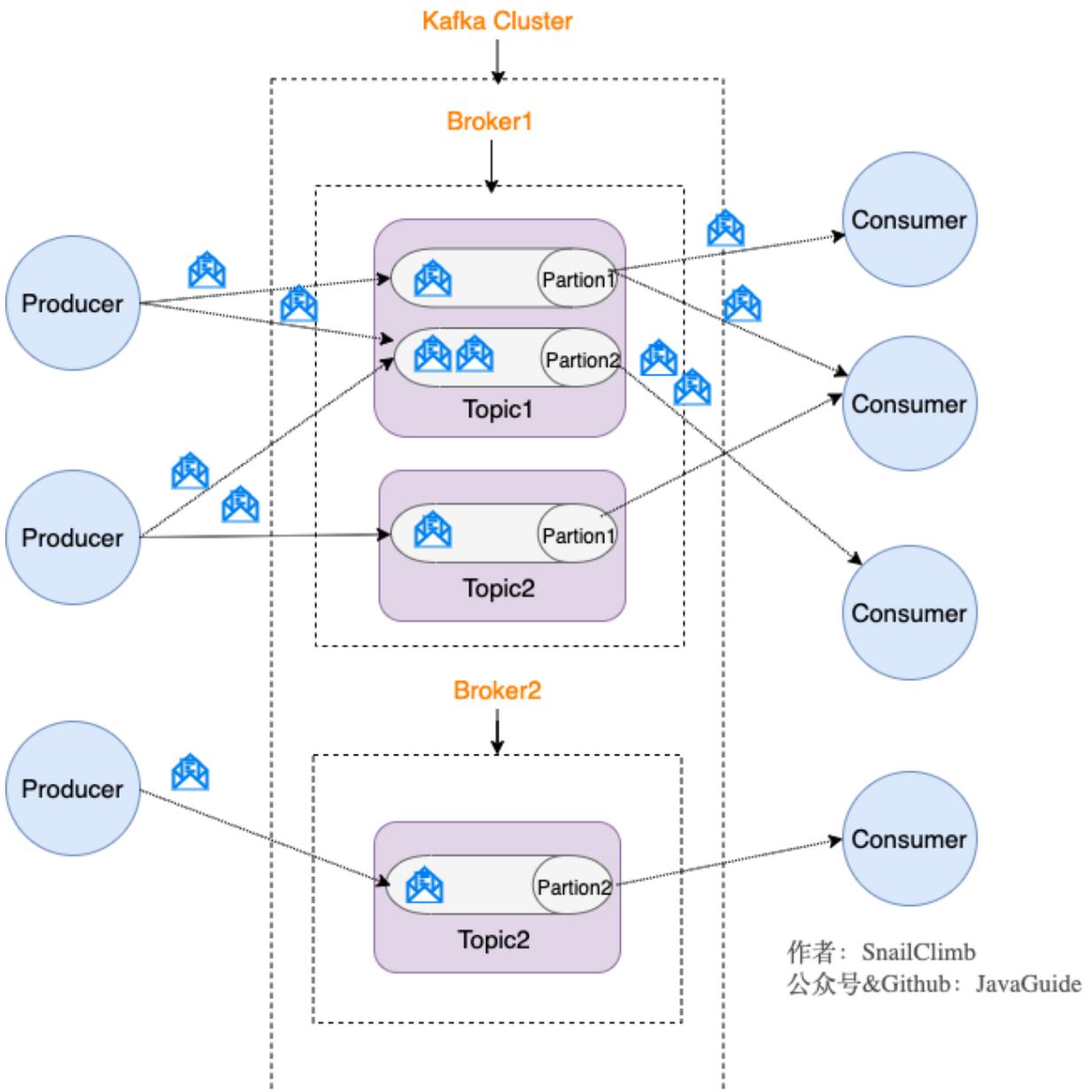
在发布 - 订阅模型中，如果只有一个订阅者，那它和队列模型就基本是一样的了。所以说，发布 - 订阅模型在功能层面上是可以兼容队列模型的。

Kafka 采用的就是发布 - 订阅模型。

RocketMQ 的消息模型和 Kafka 基本是完全一样的。唯一的区别是 Kafka 中没有队列这个概念，与之对应的是 Partition (分区)。

5.3.4 什么是Producer、Consumer、Broker、Topic、Partition?

Kafka 将生产者发布的消息发送到 Topic (主题) 中，需要这些消息的消费者可以订阅这些 Topic (主题)，如下图所示：



上面这张图也为我们引出了，Kafka 比较重要的几个概念：

1. **Producer (生产者)** : 产生消息的一方。
2. **Consumer (消费者)** : 消费消息的一方。
3. **Broker (代理)** : 可以看作是一个独立的 Kafka 实例。多个 Kafka Broker 组成一个 Kafka Cluster。

同时，你一定也注意到每个 Broker 中又包含了 Topic 以及 Partition 这两个重要的概念：

- **Topic (主题)** : Producer 将消息发送到特定的主题，Consumer 通过订阅特定的 Topic(主题) 来消费消息。
- **Partition (分区)** : Partition 属于 Topic 的一部分。一个 Topic 可以有多个 Partition，并且同一 Topic 下的 Partition 可以分布在不同的 Broker 上，这也就表明一个 Topic 可以横跨多个 Broker 。这正如我上面所画的图一样。

划重点：Kafka 中的 Partition (分区) 实际上可以对应成为消息队列中的队列。这样是不是更好理解一点？

5.3.5 Kafka 的多副本机制了解吗？带来了什么好处？

还有一点我觉得比较重要的是 Kafka 为分区 (Partition) 引入了多副本 (Replica) 机制。分区 (Partition) 中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。

生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。当 leader 副本发生故障时会从 follower 中选举出一个 leader，但是 follower 中如果有和 leader 同步程度达不到要求的参加不了 leader 的竞选。

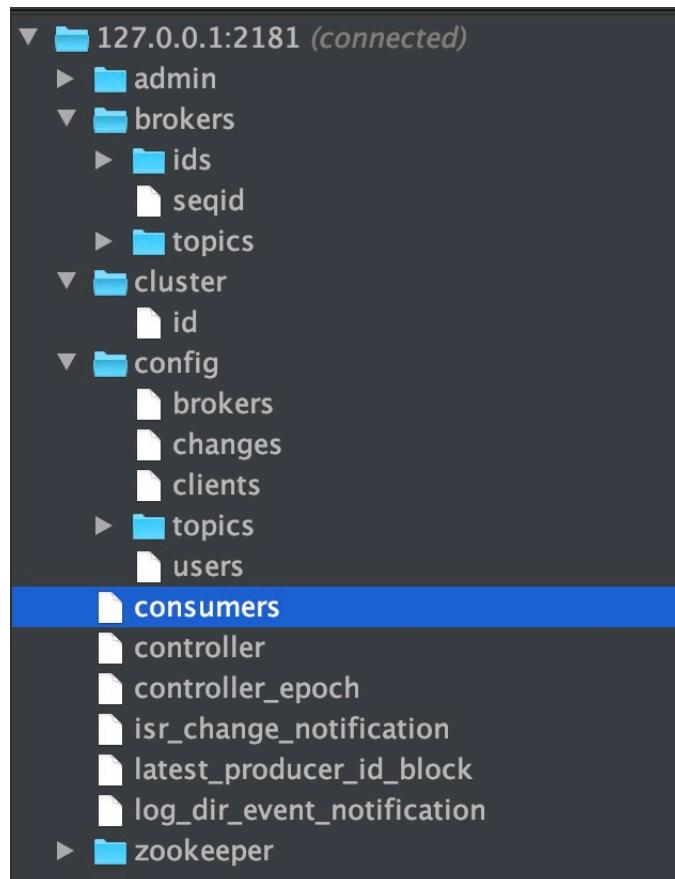
Kafka 的多分区 (Partition) 以及多副本 (Replica) 机制有什么好处呢？

1. Kafka 通过给特定 Topic 指定多个 Partition，而各个 Partition 可以分布在不同的 Broker 上，这样便能提供比较好的并发能力（负载均衡）。
2. Partition 可以指定对应的 Replica 数，这也极大地提高了消息存储的安全性，提高了容灾能力，不过也相应的增加了所需要的存储空间。

5.3.6 Zookeeper 在 Kafka 中的作用知道吗？

要想搞懂 zookeeper 在 Kafka 中的作用 一定要自己搭建一个 Kafka 环境然后自己进 zookeeper 去看一下有哪些文件夹和 Kafka 有关，每个节点又保存了什么信息。一定不要光看不实践，这样学来的也终会忘记！这部分内容参考和借鉴了这篇文章：<https://www.jianshu.com/p/a036405f989c>。

下图就是我的本地 Zookeeper，它成功和我本地的 Kafka 关联上（以下文件夹结构借助 idea 插件 Zookeeper tool 实现）。



ZooKeeper 主要为 Kafka 提供元数据的管理的功能。

从图中我们可以看出，Zookeeper 主要为 Kafka 做了下面这些事情：

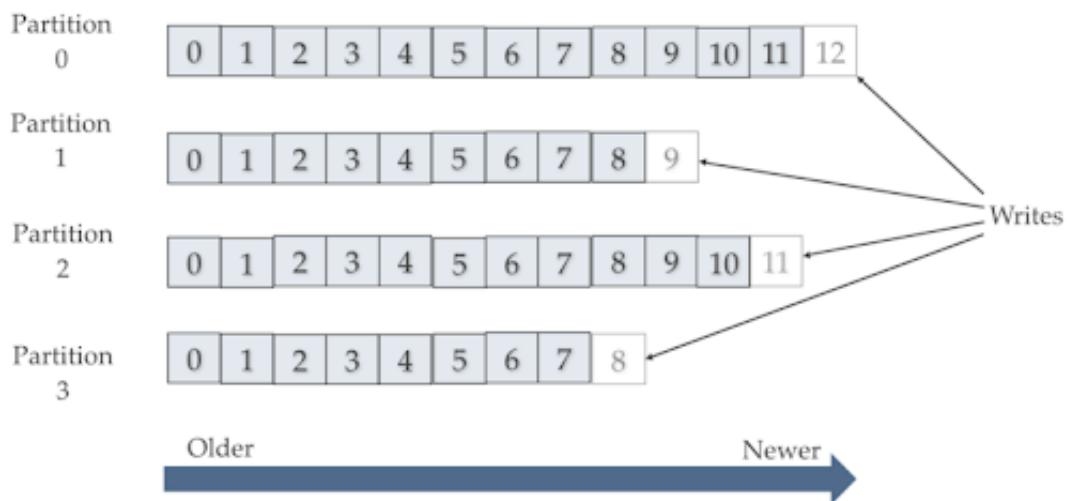
1. **Broker 注册**：在 Zookeeper 上会有一个专门用来进行 Broker 服务器列表记录的节点。每个 Broker 在启动时，都会到 Zookeeper 上进行注册，即到/brokers/ids 下创建属于自己的节点。每个 Broker 就会将自己的 IP 地址和端口等信息记录到该节点中去
 2. **Topic 注册**：在 Kafka 中，同一个 Topic 的消息会被分成多个分区并将其分布在多个 Broker 上，这些分区信息及与 Broker 的对应关系也都是由 Zookeeper 在维护。比如我创建了一个名字为 my-topic 的主题并且它有两个分区，对应到 zookeeper 中会创建这些文件夹：/brokers/topics/my-topic/Partitions/0、/brokers/topics/my-topic/Partitions/1
 3. **负载均衡**：上面也说过了 Kafka 通过给特定 Topic 指定多个 Partition，而各个 Partition 可以分布在不同的 Broker 上，这样便能提供比较好的并发能力。对于同一个 Topic 的不同 Partition，Kafka 会尽力将这些 Partition 分布到不同的 Broker 服务器上。当生产者产生消息后也会尽量投递到不同 Broker 的 Partition 里面。当 Consumer 消费的时候，Zookeeper 可以根据当前的 Partition 数量以及 Consumer 数量来实现动态负载均衡。
 4.

5.3.7 Kafka 如何保证消息的消费顺序？

我们在使用消息队列的过程中经常有业务场景需要严格保证消息的消费顺序，比如我们同时发了 2 个消息，这 2 个消息对应的操作分别对应的数据库操作是：更改用户会员等级、根据会员等级计算订单价格。假如这两条消息的消费顺序不一样造成的最终结果就会截然不同。

我们知道 Kafka 中 Partition(分区)是真正保存消息的地方，我们发送的消息都被放在了这里。而我们的 Partition(分区) 又存在于 Topic(主题) 这个概念中，并且我们可以给特定 Topic 指定多个 Partition。

Kafka Topic Partitions Layout



每次添加消息到 Partition(分区) 的时候都会采用尾加法, 如上图所示。Kafka 只能为我们保证 Partition(分区) 中的消息有序, 而不能保证 Topic(主题) 中的 Partition(分区) 的有序。

消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 (offset)。Kafka 通过偏移量 (offset) 来保证消息在分区内的顺序性。

所以，我们就有一种很简单的保证消息消费顺序的方法：1 个 Topic 只对应一个 Partition。这样当然可以解决问题，但是破坏了 Kafka 的设计初衷。

Kafka 中发送 1 条消息的时候，可以指定 topic, partition, key, data (数据) 4 个参数。如果你发送消息的时候指定了 Partition 的话，所有消息都会被发送到指定的 Partition。并且，同一个 key 的消息可以保证只发送到同一个 partition，这个我们可以采用表/对象的 id 来作为 key。

总结一下，对于如何保证 Kafka 中消息消费的顺序，有了下面两种方法：

1. 1 个 Topic 只对应一个 Partition。
2. (推荐) 发送消息的时候指定 key/Partition。

当然不仅仅只有上面两种方法，上面两种方法是我觉得比较好理解的，

5.3.8 Kafka 如何保证消息不丢失

生产者丢失消息的情况

生产者(Producer) 调用 send 方法发送消息之后，消息可能因为网络问题并没有发送过去。

所以，我们不能默认在调用 send 方法发送消息之后消息发送成功了。为了确定消息是发送成功，我们要判断消息发送的结果。但是要注意的是 Kafka 生产者(Producer) 使用 send 方法发送消息实际上是异步的操作，我们可以通过 get() 方法获取调用结果，但是这样也让它变为了同步操作，示例代码如下：

详细代码见我的这篇文章：[Kafka系列第三篇！10 分钟学会如何在 Spring Boot 程序中使用 Kafka 作为消息队列？](#)

```
SendResult<String, Object> sendResult = kafkaTemplate.send(topic, o).get();
if (sendResult.getRecordMetadata() != null) {
    logger.info("生产者成功发送消息到" +
    sendResult.getProducerRecord().topic() + "→" + sendResult.getProducerRecord().value().toString());
}
```

但是一般不推荐这么做！可以采用为其添加回调函数的形式，示例代码如下：

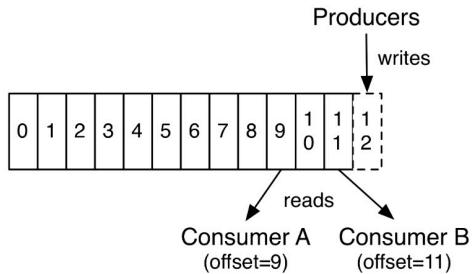
```
ListenableFuture<SendResult<String, Object>> future =
kafkaTemplate.send(topic, o);
future.addCallback(result -> logger.info("生产者成功发送消息到
topic:{} partition:{}的消息", result.getRecordMetadata().topic(),
result.getRecordMetadata().partition(),
ex -> logger.error("生产者发送消失败，原因：{}",
ex.getMessage()));
```

如果消息发送失败的话，我们检查失败的原因之后重新发送即可！

另外这里推荐为 Producer 的 retries (重试次数) 设置一个比较合理的值，一般是 3，但是为了保证消息不丢失的话一般会设置比较大一点。设置完成之后，当出现网络问题之后能够自动重试消息发送，避免消息丢失。另外，建议还要设置重试间隔，因为间隔太小的话重试的效果就不明显了，网络波动一次你3次一下子就重试完了

消费者丢失消息的情况

我们知道消息在被追加到 Partition(分区)的时候都会分配一个特定的偏移量 (offset)。偏移量 (offset) 表示 Consumer 当前消费到的 Partition(分区)的所在的位置。Kafka 通过偏移量 (offset) 可以保证消息在分区内的顺序性。



In fact, the only metadata retained on a per-consumer basis is the offset or position of that consumer in the log. This offset is controlled by the consumer: normally a consumer will advance its offset linearly as it reads records, but, in fact, since the position is controlled by the consumer it can consume records in any order it likes. For example a consumer can reset to an older offset to reprocess data from the past or skip ahead to the most recent record and start consuming from "now".

当消费者拉取到了分区的某个消息之后，消费者会自动提交了 offset。自动提交的话会有一个问题，试想一下，当消费者刚拿到这个消息准备进行真正消费的时候，突然挂掉了，消息实际上并没有被消费，但是 offset 却被自动提交了。

解决办法也比较粗暴，我们手动关闭自动提交 offset，每次在真正消费完消息之后之后再自己手动提交 offset。但是，细心的朋友一定会发现，这样会带来消息被重新消费的问题。比如你刚刚消费完消息之后，还没提交 offset，结果自己挂掉了，那么这个消息理论上就会被消费两次。

Kafka 弄丢了消息

我们知道 Kafka 为分区 (Partition) 引入了多副本 (Replica) 机制。分区 (Partition) 中的多个副本之间会有一个叫做 leader 的家伙，其他副本称为 follower。我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。生产者和消费者只与 leader 副本交互。你可以理解为其他副本只是 leader 副本的拷贝，它们的存在只是为了保证消息存储的安全性。

试想一种情况：假如 leader 副本所在的 broker 突然挂掉，那么就要从 follower 副本重新选出一个 leader，但是 leader 的数据还有一些没有被 follower 副本的同步的话，就会造成消息丢失。

设置 `acks = all`

解决办法就是我们设置 `acks = all`。`acks` 是 Kafka 生产者 (Producer) 很重要的一个参数。

`acks` 的默认值即为 1，代表我们的消息被 leader 副本接收之后就算被成功发送。当我们配置 `acks = all` 代表则所有副本都要接收到该消息之后该消息才算真正成功被发送。

设置 `replication.factor ≥ 3`

为了保证 leader 副本能有 follower 副本能同步消息，我们一般会为 topic 设置 `replication.factor ≥ 3`。这样就可以保证每个分区 (partition) 至少有 3 个副本。虽然造成了数据冗余，但是带来了数据的安全性。

设置 `min.insync.replicas > 1`

一般情况下我们还需要设置 `min.insync.replicas > 1`，这样配置代表消息至少要被写入到 2 个副本才算是被成功发送。`min.insync.replicas` 的默认值为 1，在实际生产中应尽量避免默认值 1。

但是，为了保证整个 Kafka 服务的高可用性，你需要确保 `replication.factor > min.insync.replicas`。为什么呢？设想一下加入两者相等的话，只要是有一个副本挂掉，整个分区就无法正常工作了。这明显违反高可用性！一般推荐设置成 `replication.factor = min.insync.replicas + 1`。

设置 `unclean.leader.election.enable = false`

Kafka 0.11.0.0 版本开始 `unclean.leader.election.enable` 参数的默认值由原来的 `true` 改为 `false`

我们最开始也说了我们发送的消息会被发送到 leader 副本，然后 follower 副本才能从 leader 副本中拉取消息进行同步。多个 follower 副本之间的消息同步情况不一样，当我们配置了 `unclean.leader.election.enable = false` 的话，当 leader 副本发生故障时就不会从 follower 副本中和 leader 同步程度达不到要求的副本中选择出 leader，这样降低了消息丢失的可能性。

5.3.9 Kafka 如何保证消息不重复消费

代办...

Reference

- Kafka 官方文档：<https://kafka.apache.org/documentation/>
- 极客时间-《Kafka核心技术与实战》第11节：无消息丢失配置怎么实现？

5.4 Netty 面试题总结

Netty 总算总结完了，Guide 也是长舒了一口气。有太多读者私信我让我总结 Netty 了，因为经常会在面试中碰到 Netty 相关的问题。

全文采用大家喜欢的与面试官对话的形式展开。如果大家觉得 Guide 总结的不错的话，不妨向好朋友推荐一下 JavaGuide，这是最好礼物，哈哈！

5.4.1 Netty 是什么？

 面试官：介绍一下自己对 Netty 的认识吧！小伙子。

 我：好的！那我就简单用 3 点来概括一下 Netty 吧！

1. Netty 是一个 基于 NIO 的 client-server(客户端服务器)框架，使用它可以快速简单地开发网络应用程序。
2. 它极大地简化并优化了 TCP 和 UDP 套接字服务器等网络编程，并且性能以及安全性等很多方面甚至都要更好。
3. 支持多种协议 如 FTP, SMTP, HTTP 以及各种二进制和基于文本的传统协议。

用官方的总结就是：Netty 成功地找到了一种在不妥协可维护性和性能的情况下实现易于开发，性能，稳定性和灵活性的方法。

除了上面介绍的之外，很多开源项目比如我们常用的 Dubbo、RocketMQ、Elasticsearch、gRPC 等等都用到了 Netty。

网络编程我愿意称中 Netty 为王。

5.4.2 为什么要用 Netty?

面试官：为什么要用 Netty 呢？能不能说一下自己的看法。

我：因为 Netty 具有下面这些优点，并且相比于直接使用 JDK 自带的 NIO 相关的 API 来说更加易用。

- 统一的 API，支持多种传输类型，阻塞和非阻塞的。
- 简单而强大的线程模型。
- 自带编解码器解决 TCP 粘包/拆包问题。
- 自带各种协议栈。
- 真正的无连接数据包套接字支持。
- 比直接使用 Java 核心 API 有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
- 安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。
- 社区活跃
- 成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty，比如我们经常接触的 Dubbo、RocketMQ 等等。
-

5.4.3 Netty 应用场景了解么？

面试官：能不能通俗地说一下使用 Netty 可以做什么事情？

我：凭借自己的了解，简单说一下吧！理论上来说，NIO 可以做的事情，使用 Netty 都可以做并且更好。Netty 主要用来做网络通信：

1. 作为 RPC 框架的网络通信工具：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务节点之间的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！
2. 实现一个自己的 HTTP 服务器：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为 Java 后端开发，我们一般使用 Tomcat 比较多。一个最基本的 HTTP 服务器可以处理常见的 HTTP Method 的请求，比如 POST 请求、GET 请求等等。
3. 实现一个即时通讯系统：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。
4. **实现消息推送系统**：市面上有很多消息推送系统都是基于 Netty 来做的。
5.

5.4.4 Netty 核心组件有哪些？分别有什么作用？

面试官：Netty 核心组件有哪些？分别有什么作用？

我：表面上，嘴上开始说起 Netty 的核心组件有哪些，实则，内心已经开始 mmp 了，深度怀疑这面试官是存心搞我啊！

1.Channel

Channel 接口是 Netty 对网络操作抽象类，它除了包括基本的 I/O 操作，如 bind()、connect()、read()、write() 等。

比较常用的 `Channel` 接口实现类是 `NioServerSocketChannel` (服务端) 和 `NioSocketChannel` (客户端)，这两个 `Channel` 可以和 BIO 编程模型中的 `ServerSocket` 以及 `Socket` 两个概念对应上。Netty 的 `Channel` 接口所提供的 API，大大地降低了直接使用 `Socket` 类的复杂性。

2. EventLoop

这么说吧！`EventLoop` (事件循环) 接口可以说是 Netty 中最核心的概念了！

《Netty 实战》这本书是这样介绍它的：

| `EventLoop` 定义了 Netty 的核心抽象，用于处理连接的生命周期中所发生的事件。



是不是很难理解？说实话，我学习 Netty 的时候看到这句话是没太能理解的。

说白了，`EventLoop` 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

那 `Channel` 和 `EventLoop` 直接有啥联系呢？

`Channel` 为 Netty 网络操作(读写等操作)抽象类，`EventLoop` 负责处理注册到其上的 `Channel` 处理 I/O 操作，两者配合参与 I/O 操作。

3. ChannelFuture

Netty 是异步非阻塞的，所有的 I/O 操作都为异步的。

因此，我们不能立刻得到操作是否执行成功，但是，你可以通过 `ChannelFuture` 接口的 `addListener()` 方法注册一个 `ChannelFutureListener`，当操作执行成功或者失败时，监听就会自动触发返回结果。

并且，你还可以通过 `ChannelFuture` 的 `channel()` 方法获取关联的 `Channel`

```

public interface ChannelFuture extends Future<Void> {
    Channel channel();

    ChannelFuture addListener(GenericFutureListener<? extends Future<?
super Void>> var1);
    .....

    ChannelFuture sync() throws InterruptedException;
}

```

另外，我们还可以通过 `ChannelFuture` 接口的 `sync()` 方法让异步的操作变成同步的。

4.ChannelHandler 和 ChannelPipeline

下面这段代码使用过 Netty 的小伙伴应该不会陌生，我们指定了序列化编解码器以及自定义的 `ChannelHandler` 处理消息。

```

b.group(eventLoopGroup)
    .handler(new ChannelInitializer<SocketChannel>() {
        @Override
        protected void initChannel(SocketChannel ch) {
            ch.pipeline().addLast(new
NettyKryoDecoder(kryoSerializer, RpcResponse.class));
            ch.pipeline().addLast(new
NettyKryoEncoder(kryoSerializer, RpcRequest.class));
            ch.pipeline().addLast(new KryoClientHandler());
        }
    });

```

`ChannelHandler` 是消息的具体处理器。他负责处理读写操作、客户端连接等事情。

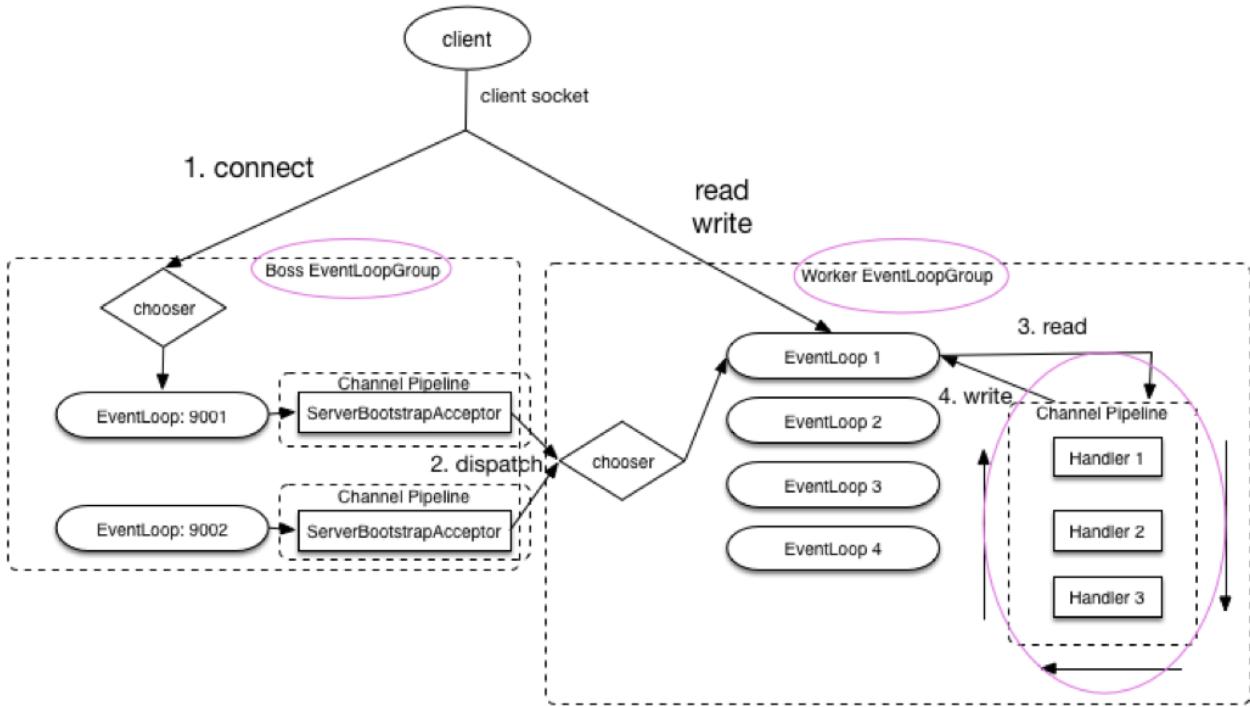
`ChannelPipeline` 为 `ChannelHandler` 的链，提供了一个容器并定义了用于沿着链传播入站和出站事件流的 API 。当 `Channel` 被创建时，它会被自动地分配到它专属的 `ChannelPipeline`。

我们可以在 `ChannelPipeline` 上通过 `addLast()` 方法添加一个或者多个 `ChannelHandler`，因为一个数据或者事件可能会被多个 Handler 处理。当一个 `ChannelHandler` 处理完之后就将数据交给下一个 `ChannelHandler`。

5.4.5 EventloopGroup 了解么？和 EventLoop 啥关系？

面试官：刚刚你也介绍了 `EventLoop`。那你再说说 `EventloopGroup` 吧！和 `EventLoop` 啥关系？

我：



EventLoopGroup 包含多个 **EventLoop** (每一个 **EventLoop** 通常内部包含一个线程) , 上面我们已经说了 **EventLoop** 的主要作用实际就是负责监听网络事件并调用事件处理器进行相关 I/O 操作的处理。

并且 **EventLoop** 处理的 I/O 事件都将在它专有的 **Thread** 上被处理, 即 **Thread** 和 **EventLoop** 属于 1 : 1 的关系, 从而保证线程安全。

上图是一个服务端对 **EventLoopGroup** 使用的大致模块图, 其中 **Boss EventloopGroup** 用于接收连接, **Worker EventloopGroup** 用于具体的处理 (消息的读写以及其他逻辑处理) 。

从上图可以看出: 当客户端通过 **connect** 方法连接服务端时, **bossGroup** 处理客户端连接请求。当客户端处理完成后, 会将这个连接提交给 **workerGroup** 来处理, 然后 **workerGroup** 负责处理其 I/O 相关操作。

5.4.6 Bootstrap 和 ServerBootstrap 了解么?

面试官 : 你再说说自己对 **Bootstrap** 和 **ServerBootstrap** 的了解吧!

我 :

Bootstrap 是客户端的启动引导类/辅助类, 具体使用方法如下:

```

EventLoopGroup group = new NioEventLoopGroup();
try {
    // 创建客户端启动引导/辅助类: Bootstrap
    Bootstrap b = new Bootstrap();
    // 指定线程模型
    b.group(group).
        .....
    // 尝试建立连接
    ChannelFuture f = b.connect(host, port).sync();
}

```

```
f.channel().closeFuture().sync();
} finally {
    // 优雅关闭相关线程组资源
    group.shutdownGracefully();
}
```

ServerBootstrap 客户端的启动引导类/辅助类，具体使用方法如下：

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2. 创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3. 给引导类配置两大线程组, 确定了线程模型
    b.group(bossGroup, workerGroup);

    .....
    // 6. 绑定端口
    ChannelFuture f = b.bind(port).sync();
    // 等待连接关闭
    f.channel().closeFuture().sync();
} finally {
    //7. 优雅关闭相关线程组资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
```

从上面的示例中，我们可以看出：

1. **Bootstrap** 通常使用 `connect()` 方法连接到远程的主机和端口，作为一个 Netty TCP 协议通信中的客户端。另外，**Bootstrap** 也可以通过 `bind()` 方法绑定本地的一个端口，作为 UDP 协议通信中的一端。
2. **ServerBootstrap** 通常使用 `bind()` 方法绑定本地的端口上，然后等待客户端的连接。
3. **Bootstrap** 只需要配置一个线程组— `EventLoopGroup`，而 **ServerBootstrap** 需要配置两个线程组— `EventLoopGroup`，一个用于接收连接，一个用于具体的处理。

5.4.7 NioEventLoopGroup 默认的构造函数会起多少线程？

面试官：看过 Netty 的源码了么？**NioEventLoopGroup** 默认的构造函数会起多少线程呢？

我：嗯嗯！看过部分。

回顾我们在上面写的服务器端的代码：

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

为了搞清楚 `NioEventLoopGroup` 默认的构造函数 到底创建了多少个线程，我们来看一下它的源码。

```
/*
 * 无参构造函数。
 * nThreads:0
 */
public NioEventLoopGroup() {
    //调用下一个构造方法
    this(0);
}

/*
 * Executor: null
 */
public NioEventLoopGroup(int nThreads) {
    //继续调用下一个构造方法
    this(nThreads, (Executor) null);
}

//中间省略部分构造函数

/*
 * RejectedExecutionHandler () : RejectedExecutionHandlers.reject()
 */
public NioEventLoopGroup(int nThreads, Executor executor, final
SelectorProvider selectorProvider, final SelectStrategyFactory
selectStrategyFactory) {
    //开始调用父类的构造函数
    super(nThreads, executor, selectorProvider,
selectStrategyFactory, RejectedExecutionHandlers.reject());
}
```

一直向下走下去的话，你会发现在 `MultithreadEventLoopGroup` 类中有相关的指定线程数的代码，如下：

```

// 从1, 系统属性, CPU核心数*2 这三个值中取出一个最大的
// 可以得出 DEFAULT_EVENT_LOOP_THREADS 的值为CPU核心数*2
private static final int DEFAULT_EVENT_LOOP_THREADS = Math.max(1,
SystemPropertyUtil.getInt("io.netty.eventLoopThreads",
NettyRuntime.availableProcessors() * 2));

// 被调用的父类构造函数, NioEventLoopGroup 默认的构造函数会起多少线
程的秘密所在
// 当指定的线程数nThreads为0时, 使用默认的线程数
DEFAULT_EVENT_LOOP_THREADS

protected MultithreadEventLoopGroup(int nThreads, ThreadFactory
threadFactory, Object... args) {
    super(nThreads == 0 ? DEFAULT_EVENT_LOOP_THREADS : nThreads,
threadFactory, args);
}

```

综上，我们发现 `NioEventLoopGroup` 默认的构造函数实际会起的线程数为 **CPU核心数*2**。

另外，如果你继续深入下去看构造函数的话，你会发现每个 `NioEventLoopGroup` 对象内部都会分配一组 `NioEventLoop`，其大小是 `nThreads`，这样就构成了一个线程池，一个 `NIOEventLoop` 和一个线程相对应，这和我们上面说的 `EventloopGroup` 和 `EventLoop` 关系这部分内容相对应。

5.4.8 Netty 线程模型了解么？

 面试官：说一下 Netty 线程模型吧！

 我：大部分网络框架都是基于 Reactor 模式设计开发的。

Reactor 模式基于事件驱动，采用多路复用将事件分发给相应的 Handler 处理，非常适合处理海量 IO 的场景。

在 Netty 主要靠 `NioEventLoopGroup` 线程池来实现具体的线程模型的。

我们实现服务端的时候，一般会初始化两个线程组：

1. `bossGroup`：接收连接。
2. `workerGroup`：负责具体的处理，交由对应的 Handler 处理。

下面我们来详细看一下 Netty 中的线程模型吧！

1. 单线程模型：

一个线程需要执行处理所有的 `accept`、`read`、`decode`、`process`、`encode`、`send` 事件。对于高负载、高并发，并且对性能要求比较高的场景不适用。

对应到 Netty 代码是下面这样的

使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 CPU 核心数 *2。

```

//1.eventGroup既用于处理客户端连接，又负责具体的处理。
EventLoopGroup eventGroup = new NioEventLoopGroup(1);
//2.创建服务端启动引导/辅助类: ServerBootstrap
ServerBootstrap b = new ServerBootstrap();
    boobtstrap.group(eventGroup, eventGroup)
//.....

```

2.多线程模型

一个 Acceptor 线程只负责监听客户端的连接，一个 NIO 线程池负责具体处理：

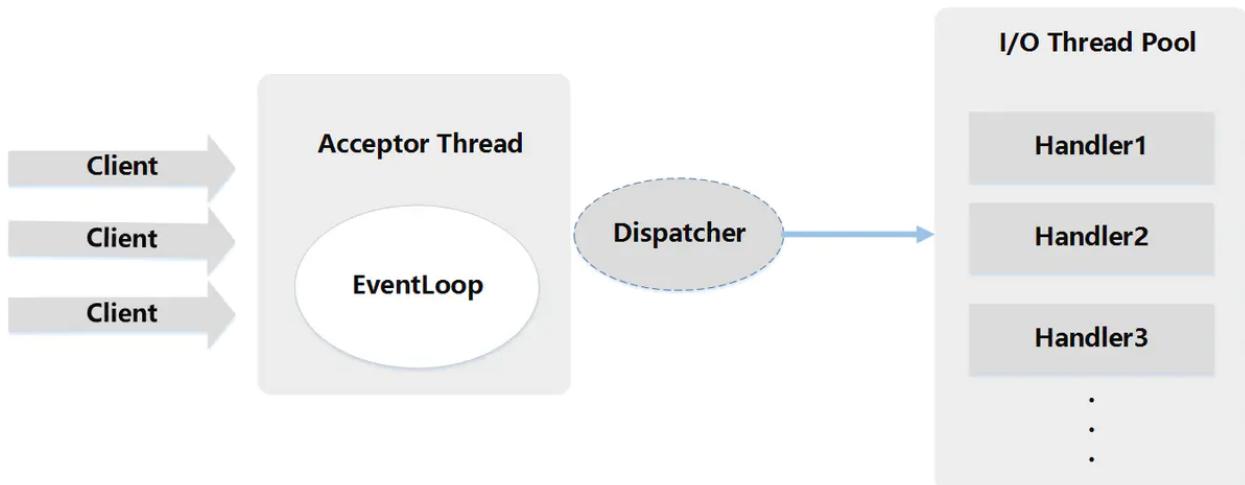
`accept`、`read`、`decode`、`process`、`encode`、`send` 事件。满足绝大部分应用场景，并发连接量不大的时候没啥问题，但是遇到并发连接大的时候就可能会出现问题，成为性能瓶颈。

对应到 Netty 代码是下面这样的：

```

// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....

```



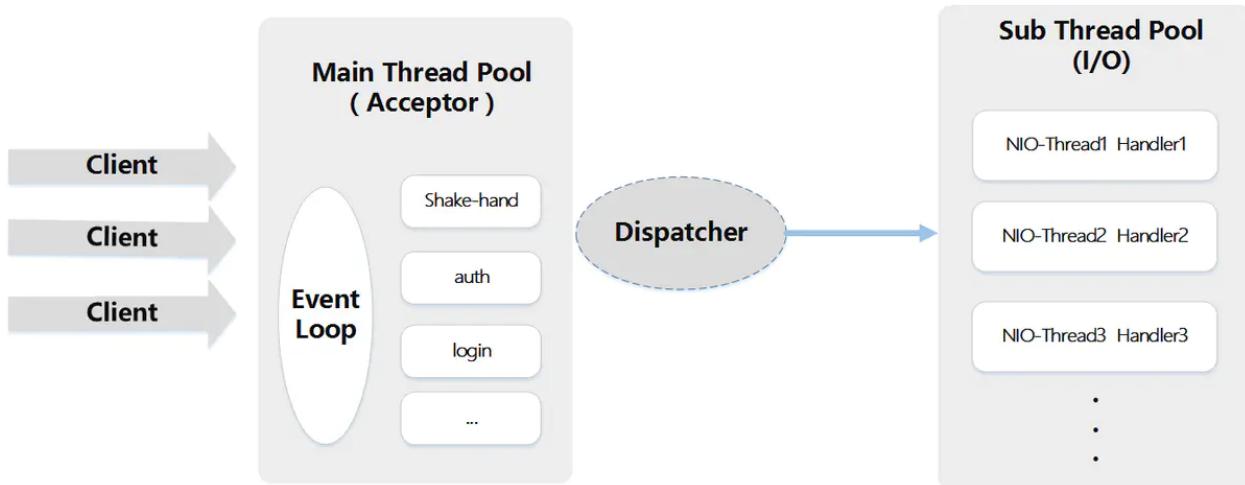
3.主从多线程模型

从一个 主线程 NIO 线程池中选择一个线程作为 Acceptor 线程，绑定监听端口，接收客户端连接的连接，其他线程负责后续的接入认证等工作。连接建立完成后，Sub NIO 线程池负责具体处理 I/O 读写。如果多线程模型无法满足你的需求的时候，可以考虑使用主从多线程模型。

```

// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....

```



5.4.9 Netty 服务端和客户端的启动过程了解么？

服务端

```

// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
        // (非必备)打印日志
        .handler(new LoggingHandler(LogLevel.INFO))
        // 4.指定 IO 模型
        .channel(NioServerSocketChannel.class)
        .childHandler(new ChannelInitializer<SocketChannel>
()
{
    @Override
    public void initChannel(SocketChannel ch) {
        ChannelPipeline p = ch.pipeline();
        //5.可以自定义客户端消息的业务处理逻辑
}

```

```

        p.addLast(new HelloServerHandler());
    }
});

// 6. 绑定端口, 调用 sync 方法阻塞知道绑定完成
ChannelFuture f = b.bind(port).sync();
// 7. 阻塞等待直到服务器Channel关闭(closeFuture()方法获取
// Channel 的CloseFuture对象, 然后调用sync()方法)
f.channel().closeFuture().sync();
} finally {
    //8. 优雅关闭相关线程组资源
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}

```

简单解析一下服务端的创建过程具体是怎样的：

1. 首先你创建了两个 `NioEventLoopGroup` 对象实例：`bossGroup` 和 `workerGroup`。

- `bossGroup`：用于处理客户端的 TCP 连接请求。
- `workerGroup`：负责每一条连接的具体读写数据的处理逻辑，真正负责 I/O 读写操作，交由对应的 Handler 处理。

举个例子：我们把公司的老板当做 `bossGroup`，员工当做 `workerGroup`，`bossGroup` 在外面接完活之后，扔给 `workerGroup` 去处理。一般情况下我们会指定 `bossGroup` 的线程数为 1（并发连接量不大的时候），`workGroup` 的线程数量为 CPU 核心数 *2。另外，根据源码来看，使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 CPU 核心数 *2。

2. 接下来 我们创建了一个服务端启动引导/辅助类：`ServerBootstrap`，这个类将引导我们进行服务端的启动工作。

3. 通过 `.group()` 方法给引导类 `ServerBootstrap` 配置两大线程组，确定了线程模型。

通过下面的代码，我们实际配置的是多线程模型，这个在上面提到过。

```

EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();

```

4. 通过 `channel()` 方法给引导类 `ServerBootstrap` 指定了 IO 模型为 `NIO`

- `NioServerSocketChannel`：指定服务端的 IO 模型为 NIO，与 BIO 编程模型中的 `ServerSocket` 对应
 - `NioSocketChannel`：指定客户端的 IO 模型为 NIO，与 BIO 编程模型中的 `Socket` 对应
5. 通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer`，然后指定了服务端消息的业务处理逻辑 `HelloServerHandler` 对象
6. 调用 `ServerBootstrap` 类的 `bind()` 方法绑定端口

客户端

```

//1. 创建一个 NioEventLoopGroup 对象实例
EventLoopGroup group = new NioEventLoopGroup();
try {
    //2. 创建客户端启动引导/辅助类: Bootstrap
    Bootstrap b = new Bootstrap();
    //3. 指定线程组
    b.group(group)
        //4. 指定 IO 模型
        .channel(NioSocketChannel.class)
        .handler(new ChannelInitializer<SocketChannel>() {
            @Override
            public void initChannel(SocketChannel ch) throws
Exception {
                ChannelPipeline p = ch.pipeline();
                // 5. 这里可以自定义消息的业务处理逻辑
                p.addLast(new HelloClientHandler(message));
            }
        });
    // 6. 尝试建立连接
    ChannelFuture f = b.connect(host, port).sync();
    // 7. 等待连接关闭 (阻塞, 直到Channel关闭)
    f.channel().closeFuture().sync();
} finally {
    group.shutdownGracefully();
}

```

继续分析一下客户端的创建流程：

1. 创建一个 `NioEventLoopGroup` 对象实例
2. 创建客户端启动的引导类是 `Bootstrap`
3. 通过 `.group()` 方法给引导类 `Bootstrap` 配置一个线程组
4. 通过 `channel()` 方法给引导类 `Bootstrap` 指定了 IO 模型为 `NIO`
5. 通过 `.childHandler()` 给引导类创建一个 `ChannelInitializer`，然后指定了客户端消息的业务处理逻辑 `HelloClientHandler` 对象
6. 调用 `Bootstrap` 类的 `connect()` 方法进行连接，这个方法需要指定两个参数：
 - `inetHost` : ip 地址
 - `inetPort` : 端口号

```
public ChannelFuture connect(String inetHost, int inetPort) {  
    return this.connect(InetSocketAddress.createUnresolved(inetHost,  
inetPort));  
}  
public ChannelFuture connect(SocketAddress remoteAddress) {  
    ObjectUtil.checkNotNull(remoteAddress, "remoteAddress");  
    this.validate();  
    return this.doResolveAndConnect(remoteAddress,  
this.config.localAddress());  
}
```

`connect` 方法返回的是一个 `Future` 类型的对象

```
public interface ChannelFuture extends Future<Void> {  
    .....  
}
```

也就是说这个方法是异步的，我们通过 `addListener` 方法可以监听到连接是否成功，进而打印出连接信息。具体做法很简单，只需要对代码进行以下改动：

```
ChannelFuture f = b.connect(host, port).addListener(future → {  
    if (future.isSuccess()) {  
        System.out.println("连接成功!");  
    } else {  
        System.err.println("连接失败!");  
    }  
}).sync();
```

5.4.10 什么是 TCP 粘包/拆包?有什么解决办法呢?

面试官：什么是 TCP 粘包/拆包？

我：TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。比如你多次发送：“你好，你真帅啊！哥哥！”，但是客户端接收到的可能是下面这样的：

```
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!  
message from client:你好,你真帅啊! 哥哥!  
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!  
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!  
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真  
！ 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!  
message from client:真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!  
！ 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真  
！ 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
```

面试官：那有什么解决办法呢？

我：

1. 使用 Netty 自带的解码器

- **LineBasedFrameDecoder** : 发送端发送数据包的时候，每个数据包之间以换行符作为分隔，**LineBasedFrameDecoder** 的工作原理是它依次遍历 **ByteBuf** 中的可读字节，判断是否有换行符，然后进行相应的截取。
- **DelimiterBasedFrameDecoder** : 可以自定义分隔符解码器，**LineBasedFrameDecoder** 实际上是一种特殊的 **DelimiterBasedFrameDecoder** 解码器。
- **FixedLengthFrameDecoder** : 固定长度解码器，它能够按照指定的长度对消息进行相应的拆包。
- **LengthFieldBasedFrameDecoder** :

2. 自定义序列化编解码器

在 Java 中自带的有实现 **Serializable** 接口来实现序列化，但由于它性能、安全性等原因一般情况下是不会被使用到的。

通常情况下，我们使用 Protostuff、Hessian2、json 序列方式比较多，另外还有一些序列化性能非常好的序列化方式也是很好的选择：

- 专门针对 Java 语言的：Kryo, FST 等等
- 跨语言的：Protostuff (基于 protobuf 发展而来) , ProtoBuf, Thrift, Avro, MsgPack 等等

由于篇幅问题，这部分内容会在后续的文章中详细分析介绍~~~

5.4.11 Netty 长连接、心跳机制了解么？

 面试官：TCP 长连接和短连接了解么？

 我：我们知道 TCP 在进行读写之前，server 与 client 之间必须提前建立一个连接。建立连接的过程，需要我们常说的三次握手，释放/关闭连接的话需要四次挥手。这个过程是比较消耗网络资源并且有时间延迟的。

所谓，短连接说的就是 server 端与 client 端建立连接之后，读写完成之后就关闭掉连接，如果下一次再要互相发送消息，就要重新连接。短连接的有点很明显，就是管理和实现都比较简单，缺点也很明显，每一次的读写都要建立连接必然会带来大量网络资源的消耗，并且连接的建立也需要耗费时间。

长连接说的就是 client 向 server 双方建立连接之后，即使 client 与 server 完成一次读写，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。长连接的可以省去较多的 TCP 建立和关闭的操作，降低对网络资源的依赖，节约时间。对于频繁请求资源的客户来说，非常适用长连接。

 面试官：为什么需要心跳机制？Netty 中心跳机制了解么？

 我：

在 TCP 保持长连接的过程中，可能会出现断网等网络异常出现，异常发生的时候，client 与 server 之间如果没有交互的话，它们是无法发现对方已经掉线的。为了解决这个问题，我们就需要引入 心跳机制。

心跳机制的工作原理是：在 client 与 server 之间在一定时间内没有数据交互时，即处于 idle 状态时，客户端或服务器就会发送一个特殊的数据包给对方，当接收方收到这个数据报文后，也立即发送一个特殊的数据报文，回应发送方，此即一个 PING-PONG 交互。所以，当某一段收到心跳消息后，就知道了对方仍然在线，这就确保 TCP 连接的有效性。

TCP 实际上自带的就有长连接选项，本身是也有心跳包机制，也就是 TCP 的选项：`SO_KEEPALIVE`。但是，TCP 协议层面的长连接灵活性不够。所以，一般情况下我们都是在应用层协议上实现自定义心跳机制的，也就是在 Netty 层面通过编码实现。通过 Netty 实现心跳机制的话，核心类是 `IdleStateHandler`。

5.4.12 Netty 的零拷贝了解么？

面试官：讲讲 Netty 的零拷贝？

我：

维基百科是这样介绍零拷贝的：

零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU 不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省 CPU 周期和内存带宽。

在 OS 层面上的 `Zero-copy` 通常指避免在 `用户态(User-space)` 与 `内核态(Kernel-space)` 之间来回拷贝数据。而在 Netty 层面，零拷贝主要体现在对于数据操作的优化。

Netty 中的零拷贝体现在以下几个方面

1. 使用 Netty 提供的 `CompositeByteBuf` 类，可以将多个 `ByteBuf` 合并为一个逻辑上的 `ByteBuf`，避免了各个 `ByteBuf` 之间的拷贝。
2. `ByteBuf` 支持 `slice` 操作，因此可以将 `ByteBuf` 分解为多个共享同一个存储区域的 `ByteBuf`，避免了内存的拷贝。
3. 通过 `FileRegion` 包装的 `FileChannel.transferTo` 实现文件传输，可以直接将文件缓冲区的数据发送到目标 `Channel`，避免了传统通过循环 `write` 方式导致的内存拷贝问题。

参考

- netty 学习系列二：NIO Reactor 模型 & Netty 线程模型：<https://www.jianshu.com/p/38b56531565d>
- 《Netty 实战》
- Netty 面试题整理(2)：<https://metatronx1.github.io/2019/10/22/Netty-面试题整理-二/>
- Netty (3) -源码 NioEventLoopGroup：<https://www.cnblogs.com/qdhxhz/p/10075568.html>
- 对于 Netty `ByteBuf` 的零拷贝(Zero Copy) 的理解：
[https://www.cnblogs.com/xys1228/p/6088805.html-----](https://www.cnblogs.com/xys1228/p/6088805.html)

六 认证授权

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide**（公众号同名）作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

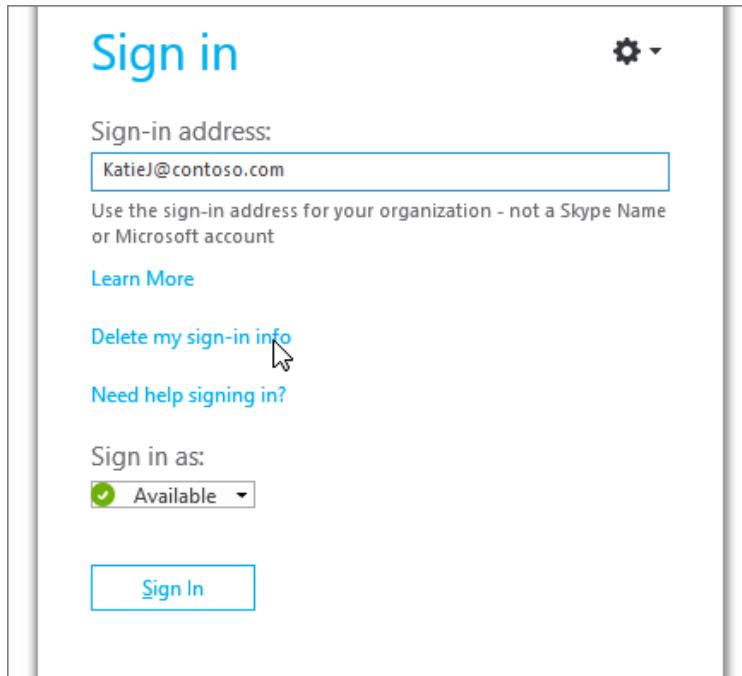
6.1 认证授权面试题总结

6.1.1 认证 (Authentication) 和授权 (Authorization)的区别是什么？

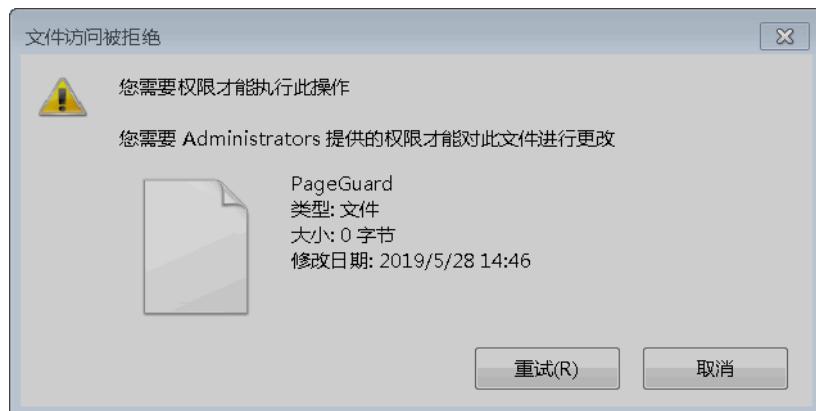
这是一个绝大多数人都会混淆的问题。首先先从读音上来认识这两个名词，很多人都会把它俩的读音搞混，所以我建议你先先去查一查这两个单词到底该怎么读，他们的具体含义是什么。

说简单点就是：

认证 (Authentication): 你是谁。



授权 (Authorization): 你有权限干什么。



稍微正式点（啰嗦点）的说法就是：

- **Authentication (认证)** 是验证您的身份的凭据（例如用户名/用户ID和密码），通过这个凭据，系统得以知道你就是你，也就是说系统存在你这个用户。所以，Authentication 被称为身份/用户验证。
- **Authorization (授权)** 发生在 **Authentication (认证)** 之后。授权嘛，光看意思大家应该就明白，它主要掌管我们访问系统的权限。比如有些特定资源只能具有特定权限的人才能访问比如 admin，有些对系统资源操作比如删除、添加、更新只能特定人才具有。

这两个一般在我们的系统中被结合在一起使用，目的就是为了保护我们系统的安全性。

6.1.2 什么是Cookie ? Cookie的作用是什么?如何在服务端使用Cookie ?

Name	Value
act	151
c_user	101
csrfToken	WHR4FdDOVC
datr	mcj7
ds_user_id	299
fbsr_124024574287414	e-w...5v
mid	WL3MngALAAE-H34
presence	EDvF3EtinF151805
rur	FTW
sb	pMu5WP...uvBDN4i...vDQZLSC
sessionid	IGSC209d92c9...06266a96125b488e
urle	"{"time": 151}
wd	2560x1334
xs	79%3A_KdtSal

什么是Cookie ? Cookie的作用是什么?

Cookie 和 Session都是用来跟踪浏览器用户身份的会话方式，但是两者的应用场景不太一样。

维基百科是这样定义 Cookie 的：Cookies是某些网站为了辨别用户身份而储存在用户本地终端上的数据（通常经过加密）。简单来说：Cookie 存放在客户端，一般用来保存用户信息。

下面是 Cookie 的一些应用案例：

1. 我们在 Cookie 中保存已经登录过的用户信息，下次访问网站的时候页面可以自动帮你登录的一些基本信息给填了。除此之外，Cookie 还能保存用户首选项，主题和其他设置信息。
2. 使用Cookie 保存 session 或者 token，向后端发送请求的时候带上 Cookie，这样后端就能取到session或者token了。这样就能记录用户当前的状态了，因为 HTTP 协议是无状态的。
3. Cookie 还可以用来记录和分析用户行为。举个简单的例子你在网上购物的时候，因为HTTP协议是没有状态的，如果服务器想要获取你在某个页面的停留状态或者看了哪些商品，一种常用的表现方式就是将这些信息存放在Cookie

如何在服务端使用 Cookie 呢？

这部分内容参考：<https://attacomsian.com/blog/cookies-spring-boot>，更多如何在Spring Boot中使用Cookie 的内容可以查看这篇文章。

1) 设置cookie返回给客户端

```

@GetMapping("/change-username")
public String setCookie(HttpServletRequest response) {
    // 创建一个 cookie
    Cookie cookie = new Cookie("username", "Jovan");
    //设置 cookie过期时间
    cookie.setMaxAge(7 * 24 * 60 * 60); // expires in 7 days
    //添加到 response 中
    response.addCookie(cookie);

    return "Username is changed!";
}

```

2) 使用Spring框架提供的`@CookieValue`注解获取特定的 cookie的值

```

@GetMapping("/")
public String readCookie(@CookieValue(value = "username", defaultValue =
"Atta") String username) {
    return "Hey! My username is " + username;
}

```

3) 读取所有的 Cookie 值

```

@GetMapping("/all-cookies")
public String readAllCookies(HttpServletRequest request) {

    Cookie[] cookies = request.getCookies();
    if (cookies != null) {
        return Arrays.stream(cookies)
            .map(c → c.getName() + "=" +
c.getValue()).collect(Collectors.joining(", "));
    }

    return "No cookies";
}

```

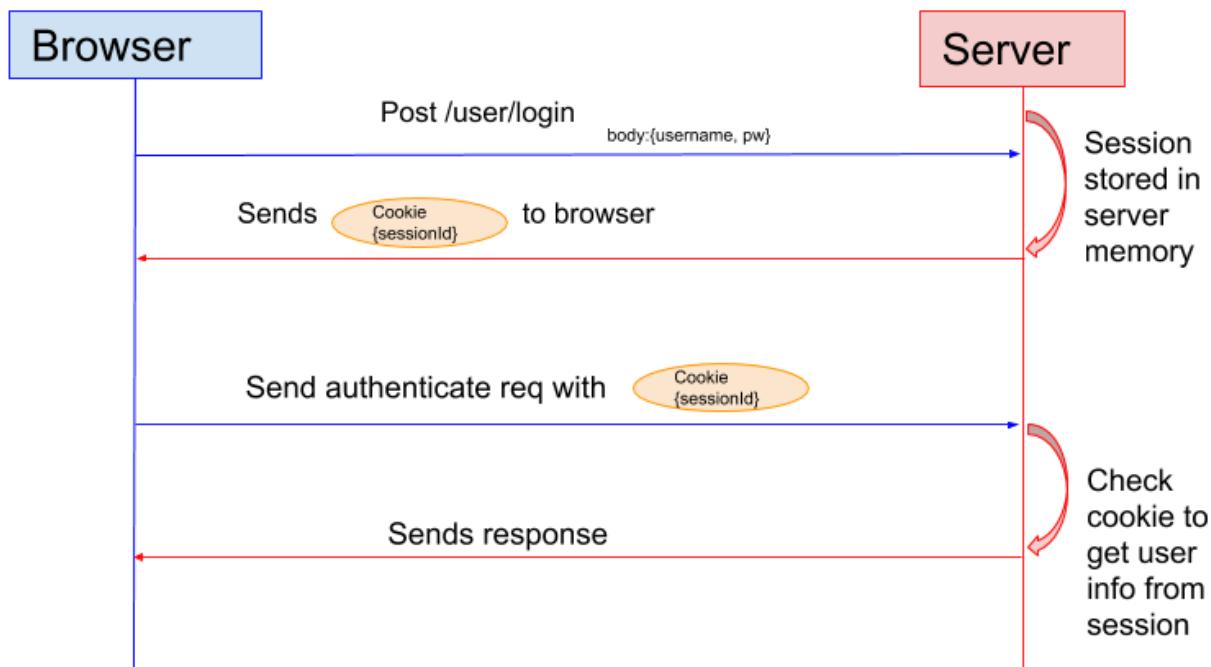
6.1.3 Cookie 和 Session 有什么区别？如何使用Session进行身份验证？

Session 的主要作用就是通过服务端记录用户的状态。典型的场景是购物车，当你要添加商品到购物车的时候，系统不知道是哪个用户操作的，因为 HTTP 协议是无状态的。服务端给特定的用户创建特定的 Session 之后就可以标识这个用户并且跟踪这个用户了。

Cookie 数据保存在客户端(浏览器端)，Session 数据保存在服务器端。相对来说 Session 安全性更高。如果使用 Cookie 的一些敏感信息不要写入 Cookie 中，最好能将 Cookie 信息加密然后使用到的时候再去服务器端解密。

那么，如何使用Session进行身份验证？

很多时候我们都是通过 SessionID 来实现特定的用户，SessionID 一般会选择存放在 Redis 中。举个例子：用户成功登陆系统，然后返回给客户端具有 SessionID 的 Cookie，当用户向后端发起请求的时候会把 SessionID 带上，这样后端就知道你的身份状态了。关于这种认证方式更详细的过程如下：



1. 用户向服务器发送用户名和密码用于登陆系统。
2. 服务器验证通过后，服务器为用户创建一个 Session，并将 Session信息存储起来。
3. 服务器向用户返回一个 SessionID，写入用户的 Cookie。
4. 当用户保持登录状态时，Cookie 将与每个后续请求一起被发送出去。
5. 服务器可以将存储在 Cookie 上的 Session ID 与存储在内存中或者数据库中的 Session 信息进行比较，以验证用户的身份，返回给用户客户端响应信息的时候会附带用户当前的状态。

使用 Session 的时候需要注意下面几个点：

1. 依赖Session的关键业务一定要确保客户端开启了Cookie。
2. 注意Session的过期时间

花了个图简单总结了一下Session认证涉及的一些东西。



- 1.生成保存SessionId并传递到前端
2.保存Session内容（比如当前用户的购物车的商品信息）

- 1.使用 Cookie 保存SessionId
2.请求后端的时候带上SessionId

另外，Spring Session提供了一种跨多个应用程序或实例管理用户会话信息的机制。如果想详细了解可以查看下面几篇很不错的文章：

- [Getting Started with Spring Session](#)
- [Guide to Spring Session](#)
- [Sticky Sessions with Spring Session & Redis](#)

6.1.4 如果没有Cookie的话Session还能用吗？

这是一道经典的面试题！

一般是通过 Cookie 来保存 SessionID，假如你使用了 Cookie 保存 SessionID 的方案的话，如果客户端禁用了 Cookie，那么 Session 就无法正常工作。

但是，并不是没有 Cookie 之后就不能用 Session 了，比如你可以将 SessionID 放在请求的 url 里面 https://javaguide.cn/?session_id=xxx。这种方案的话可行，但是安全性和用户体验感降低。当然，为了你也可以对 SessionID 进行一次加密之后再传入后端。

6.1.5 为什么Cookie 无法防止CSRF攻击，而token可以？

CSRF (Cross Site Request Forgery) 一般被翻译为 跨站请求伪造。那么什么是 跨站请求伪造 呢？说简单用你的身份去发送一些对你不友好的请求。举个简单的例子：

小壮登录了某网上银行，他来到了网上银行的帖子区，看到一个帖子下面有一个链接写着“科学理财，年盈利率过万”，小壮好奇的点开了这个链接，结果发现自己的账户少了10000元。这是怎么回事呢？原来黑客在链接中藏了一个请求，这个请求直接利用小壮的身份给银行发送了一个转账请求，也就是通过你的 Cookie 向银行发出请求。

```
<a src=http://www.mybank.com/Transfer?bankId=11&money=10000>科学理财，年盈利率过万</a>
```

上面也提到过，进行 Session 认证的时候，我们一般使用 Cookie 来存储 SessionId，当我们登陆后后端生成一个 SessionId 放在 Cookie 中返回给客户端，服务端通过 Redis 或者其他存储工具记录保存着这个 SessionId，客户端登录以后每次请求都会带上这个 SessionId，服务端通过这个 SessionId 来标示你这个人。如果别人通过 cookie 拿到了 SessionId 后就可以代替你的身份访问系统了。

Session 认证中 Cookie 中的 SessionId 是由浏览器发送到服务端的，借助这个特性，攻击者就可以通过让用户误点攻击链接，达到攻击效果。

但是，我们使用 token 的话就不会存在这个问题，在我们登录成功获得 token 之后，一般会选择存放在 local storage 中。然后我们在前端通过某些方式会给每个发到后端的请求加上这个 token，这样就不会出现 CSRF 漏洞的问题。因为，即使有个你点击了非法链接发送了请求到服务端，这个非法请求是不会携带 token 的，所以这个请求将是非法的。

需要注意的是不论是 Cookie 还是 token 都无法避免跨站脚本攻击 (Cross Site Scripting) XSS。

跨站脚本攻击 (Cross Site Scripting) 缩写为 CSS 但这会与层叠样式表 (Cascading Style Sheets, CSS) 的缩写混淆。因此，有人将跨站脚本攻击缩写为 XSS。

XSS 中攻击者会用各种方式将恶意代码注入到其他用户的页面中。就可以通过脚本盗用信息比如 cookie。

推荐阅读：

6.1.6 什么是 Token?什么是 JWT?如何基于Token进行身份验证?

我们在上一个问题中探讨了使用 Session 来鉴别用户的身份，并且给出了几个 Spring Session 的案例分享。我们知道 Session 信息需要保存一份在服务器端。这种方式会带来一些麻烦，比如需要我们保证保存 Session 信息服务器的可用性、不适合移动端（依赖Cookie）等等。

有没有一种不需要自己存放 Session 信息就能实现身份验证的方式呢？使用 Token 即可！JWT (JSON Web Token) 就是这种方式的实现，通过这种方式服务器端就不需要保存 Session 数据了，只用在客户端保存服务端返回给客户的 Token 就可以了，扩展性得到提升。

JWT 本质上就一段签名的 JSON 格式的数据。由于它是带有签名的，因此接收者便可以验证它的真实性。

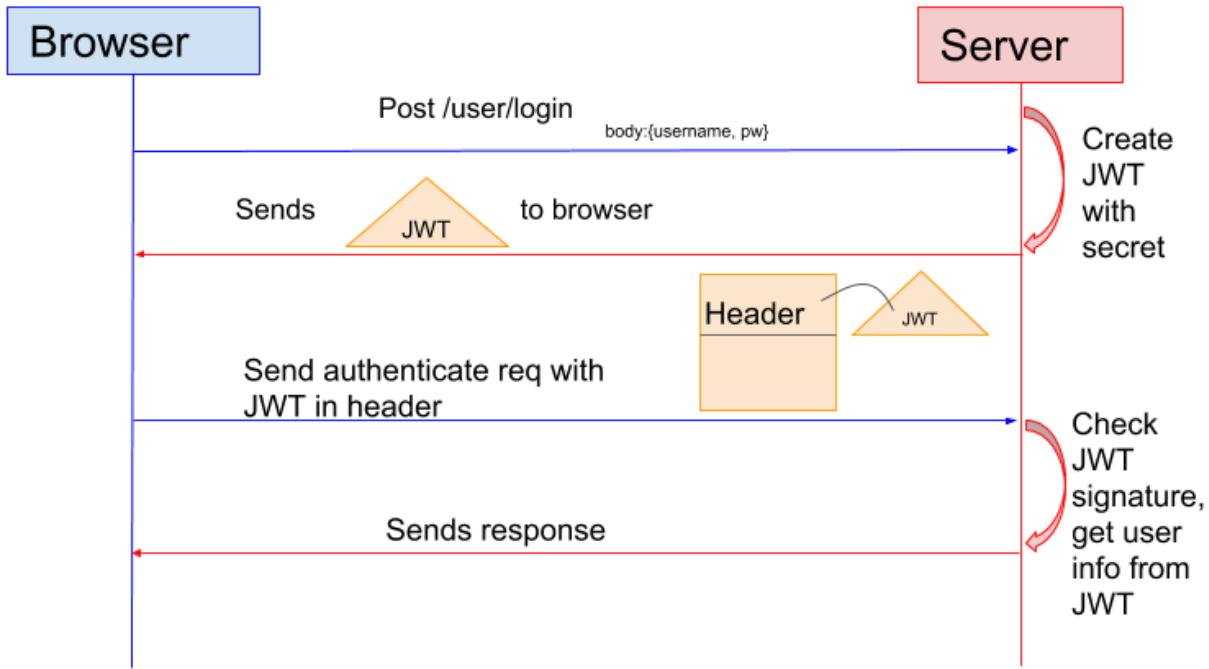
下面是 [RFC 7519](#) 对 JWT 做的较为正式的定义。

JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. —[JSON Web Token \(JWT\)](#)

JWT 由 3 部分构成：

1. Header :描述 JWT 的元数据。定义了生成签名的算法以及 Token 的类型。
2. Payload (负载) :用来存放实际需要传递的数据
3. Signature (签名) : 服务器通过 Payload、Header 和一个密钥(secret)使用 Header 里面指定的签名算法（默认是 HMAC SHA256）生成。

在基于 Token 进行身份验证的应用程序中，服务器通过 Payload、Header 和一个密钥(secret) 创建令牌 (Token) 并将 Token 发送给客户端，客户端将 Token 保存在 Cookie 或者 localStorage 里面，以后客户端发出的所有请求都会携带这个令牌。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP Header 的 Authorization 字段中：Authorization: Bearer Token。



1. 用户向服务器发送用户名和密码用于登陆系统。
2. 身份验证服务响应并返回了签名的 JWT，上面包含了用户是谁的内容。
3. 用户以后每次向后端发请求都在Header中带上 JWT。
4. 服务端检查 JWT 并从中获取用户相关信息。

推荐阅读：

- [JWT \(JSON Web Tokens\) Are Better Than Session Cookies](#)
- [JSON Web Tokens \(JWT\) 与 Sessions](#)
- [JSON Web Token 入门教程](#)
- [彻底理解Cookie, Session, Token](#)

6.1.7 什么是OAuth 2.0?

OAuth 是一个行业的标准授权协议，主要用来授权第三方应用获取有限的权限。而 OAuth 2.0是对 OAuth 1.0 的完全重新设计，OAuth 2.0更快，更容易实现，OAuth 1.0 已经被废弃。详情请见：[rfc6749](#)。

实际上它就是一种授权机制，它的最终目的是为第三方应用颁发一个有时效性的令牌 token，使得第三方应用能够通过该令牌获取相关的资源。

OAuth 2.0 比较常用的场景就是第三方登录，当你的网站接入了第三方登录的时候一般就是使用的 OAuth 2.0 协议。

另外，现在OAuth 2.0也常见于支付场景（微信支付、支付宝支付）和开发平台（微信开放平台、阿里开放平台等等）。

微信支付账户相关参数：

邮件中参数	API参数名	详细说明
APPID	appid	appid是微信公众账号或开放平台APP的唯一标识，在公众平台申请公众账号或者在开放平台申请APP账号后，微信会自动分配对应的appid，用于标识该应用。可在微信公众平台-->开发-->基本配置里面查看，商户的微信支付审核通过邮件中也会包含该字段值。
微信支付商户号	mch_id	商户申请微信支付后，由微信支付分配的商户收款账号。
API密钥	key	交易过程生成签名的密钥，仅保留在商户系统和微信支付后台，不会在网络中传播。商户妥善保管该Key，切勿在网络中传输，不能在其他客户端中存储，保证key不会被泄漏。商户可根据邮件提示登录微信商户平台进行设置。也可按以下路径设置：微信商户平台(pay.weixin.qq.com)-->账户中心-->账户设置-->API安全-->密钥设置
Appsecret	secret	AppSecret是APPID对应的接口密码，用于 获取接口调用凭证access_token 时使用。在微信支付中，先通过 OAuth2.0接口 获取用户openid，此openid用于微信内网页支付模式下单接口使用。可登录公众平台-->微信支付，获取AppSecret（需成为开发者且帐号没有异常状态）。

推荐阅读：

- [OAuth 2.0 的一个简单解释](#)
- [10 分钟理解什么是 OAuth 2.0 协议](#)
- [OAuth 2.0 的四种方式](#)
- [GitHub OAuth 第三方登录示例教程](#)

6.1.8 什么是 SSO？

SSO(Single Sign On)即单点登录说的是用户登陆多个子系统的其中一个就有权访问与其相关的其他系统。举个例子我们在登陆了京东金融之后，我们同时也成功登陆京东的京东超市、京东家电等子系统。

6.1.9 SSO与OAuth2.0的区别

OAuth 是一个行业的标准授权协议，主要用来授权第三方应用获取有限的权限。SSO解决的是一个公司的多个相关的自系统的之间的登陆问题比如京东旗下相关子系统京东金融、京东超市、京东家电等等。

参考

- <https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4>
- <https://www.varonis.com/blog/what-is-oauth/>
- <https://tools.ietf.org/html/rfc6749----->

七 优质面经

作者：Guide哥。

介绍：Github 70k Star 项目 [JavaGuide](#) (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

五面阿里,终获offer

作者: ppxyn。本文来自读者投稿, 同时也欢迎各位投稿, 对于不错的原创文章我根据你的选择给予现金(100-500)、付费专栏或者任选书籍进行奖励! 所以, 快提 pr 或者邮件的方式(邮件地址在主页)给我投稿吧! 当然, 我觉得奖励是次要的, 最重要的是你可以从自己整理知识点的过程中学习到很多知识。

前言

在接触 Java 之前我接触的比较多的是硬件方面, 用的比较多的语言就是C和C++。到了大三我才正式选择 Java 方向, 到目前为止使用Java到现在大概有一年多的时间, 所以Java算不上很好。刚开始投递的时候, 实习刚辞职, 也没准备笔试面试, 很多东西都忘记了。所以, 刚开始我并没有直接就投递阿里, 毕竟心里还是有一点点小害怕的。于是, 我就先投递了几个不算大的公司来练手, 就是想着刷刷经验而已或者说练练手(ps: 还是挺对不起那些公司的)。面了一个月其他公司后, 我找了我实验室的学长内推我, 后面就有了这5次面试。

下面简单的说一下我的这5次面试: 4次技术面+1次HR面, 希望我的经历能对你有所帮助。

一面(技术面)

1. 自我介绍 (主要讲自己会的技术细节, 项目经验, 经历那些就一语带过, 后面试官会问你的)。
2. 聊聊项目 (就是一个很普通的分布式商城, 自己做了一些改进), 让我画了整个项目的架构图, 然后针对项目抛了一系列的提高性能的问题, 还问了我做项目的过程中遇到了那些问题, 如何解决的, 差不读就这些吧。
3. 可能是我前面说了我会数据库优化, 然后面试官就开始问索引、事务隔离级别、悲观锁和乐观锁、索引、ACID、MVVC这些问题。
4. 浏览器输入URL发生了什么? TCP和UDP区别? TCP如何保证传输可靠性?
5. 讲下跳表怎么实现的? 哈夫曼编码是怎么回事? 非递归且不用额外空间(不用栈), 如何遍历二叉树
6. 后面又问了很多JVM方面的问题, 比如Java内存模型、常见的垃圾回收器、双亲委派模型这些
7. 你有什么问题要问吗?

二面(技术面)

1. 自我介绍 (主要讲自己会的技术细节, 项目经验, 经历那些就一语带过, 后面试官会问你的)。
2. 操作系统的内存管理机制
3. 进程和线程的区别
4. 说下你对线程安全的理解
5. volatile 有什么作用, synchronized和lock有什么区别
6. ReentrantLock实现原理
7. 用过CountDownLatch么? 什么场景下用的?
8. AQS底层原理。
9. 造成死锁的原因有哪些, 如何预防?
10. 加锁会带来哪些性能问题。如何解决?
11. HashMap、ConcurrentHashMap源码。HashMap是线程安全的吗? Hashtable呢? ConcurrentHashMap有了解吗?
12. 是否可以实习?
13. 你有什么问题要问吗?

三面(技术面)

1. 有没有参加过 ACM 或者其他竞赛，有没有拿过什么奖？（我说我没参加过 ACM，本科参加过数学建模竞赛，名次并不好，没拿过什么奖。面试官好像有点失望，然后我又赶紧补充说我和老师一起做过一个项目，目前已经投入使用。面试官还比较感兴趣，后面又和他聊了一下这个项目。）
2. 研究生期间，做过什么项目，发过论文吗？有什么成果吗？
3. 你觉得你有什么优点和缺点？你觉得你相比于那些比你更优秀的人欠缺什么？
4. 有读过什么源码吗？（我说我读过 Java 集合框架和 Netty 的，面试官说 Java 集合前几面一定问的差不多，就不问了，然后就问我 Netty 的，我当时很慌啊！）
5. 介绍一下自己对 Netty 的认识，为什么要用。说说业务中，Netty 的使用场景。什么是 TCP 粘包/拆包，解决办法。Netty 线程模型。Dubbo 在使用 Netty 作为网络通讯时候是如何避免粘包与半包问题？讲讲 Netty 的零拷贝？巴拉巴拉问了好多，我记得好几个我都没回答上来，心里想着凉凉了啊。
6. 用到了哪些开源技术、在开源领域做过贡献吗？
7. 常见的排序算法及其复杂度，现场写了快排。
8. 红黑树，B 树的一些问题。
9. 讲讲算法及数据结构在实习项目中的用处。
10. 自己的未来规划（就简单描述了一下自己未来的设想啊，说的还挺诚恳，面试官好像还挺满意的）
11. 你有什么问题要问吗？

四面(半个技术面)

三面面完当天，晚上 9 点接到面试电话，感觉像是部门或者项目主管。这个和之前的面试不大相同，感觉面试官主要考察的是你解决问题的能力、学习能力和团队协作能力。

1. 让我讲一个自己觉得最不错的项目。然后就巴拉巴拉的聊，我记得主要是问了项目是如何进行协作的、遇到问题是如何解决的、与他人发生冲突是如何解决的这些。感觉聊了挺久。
2. 出现 OOM 后你会怎么排查问题？
3. 自己平时是如何学习新技术的？除了 Java 还回去了解其他技术吗？
4. 上一段实习经历的收获。
5. NginX 如何做负载均衡、常见的负载均衡算法有哪些、一致性哈希的一致性是什么意思、一致性哈希是如何做哈希的
6. 你有什么问题问我吗？
7. 还有一些其他的，想不起来了，感觉这一面不是偏向技术来问。

五面(HR面)

1. 自我介绍（主要讲能突出自己的经历，会的编程技术一语带过）。
2. 你觉得你有什么优点和缺点？如何克服这些缺点？
3. 说一件大学里你自己比较有成就感的一件事情，为此付出了那些努力。
4. 你前面跟其他面试官讲过一些你做的项目吧？可以给我讲讲吗？你要考虑到我不是一个做技术的人，怎么让我也听得懂。项目中有什么问题，你怎么解决的？你最大的收获是什么？
5. 你目前有面试过其他公司吗？如果让你选，这些公司和阿里，你选哪个？（送分题，回答不好可能送命）
6. 你期望的工作地点是哪里？
7. 你有什么问题吗？

总结

1. 可以看出面试官问我的很多问题都是比较常见的问题，所以记得一定要提前准备，还要深入准备，不要回答的太皮毛。很多时候一个问题可能会牵扯出很多问题，遇到不会的问题不要慌，冷静分析，如果你真的回答不上来，也不要担心自己是不是就要挂了，很可能这个问题本身就比较难。

- 表达能力和沟通能力太重要了，一定要提前练一下，我自身就是一个不太会说话的人，所以，面试前我对于自我介绍、项目介绍和一些常见问题都在脑子里练了好久，确保面试的时候能够很清晰和简洁的说出来。
- 等待面试的过程和面试的过程真的好熬人，那段时间我压力也比较大，好在我私下找到学长聊了很多，心情也好了很多。
- 面试之后及时总结，面的好好的话，不要得意，尽快准备下一场面试吧！

我觉得我还算是比较幸运的，最后也祝大家都能获得心仪的Offer。

蚂蚁金服实习生面经总结

本文来自 Anonymous 的投稿，Guide哥 对原文进行了重新排版和一点完善。

一面 (37 分钟左右)

一面是上海的小哥打来的，3.12号中午确认的内推，下午就打来约时间了，也是唯一一个约时间的面试官。约的晚上八点。紧张的一比，人生第一次面试就献给了阿里。

幸运的一面的小哥特温柔。好像是个海归？口语中夹杂着英文。废话不多说，上干货：

面试官：先自我介绍一下吧！

我：巴拉巴拉....。

关于自我介绍：从 HR 面、技术面到高管面/部门主管面，面试官一般会让你先自我介绍一下，所以好好准备自己的自我介绍真的非常重要。网上一般建议的是准备好两份自我介绍：一份对 HR 说的，主要讲能突出自己的经历，会的编程技术一语带过；另一份对技术面试官说的，主要讲自己会的技术细节，项目经验，经历那些就一语带过。

面试官：我看你简历上写你做了个秒杀系统？我们就从这个项目开始吧，先介绍下你的项目。

关于项目介绍：如果有项目的话，技术面试第一步，面试官一般都是让你自己介绍一下你的项目。你可以从下面几个方向来考虑：

- 对项目整体设计的一个感受（面试官可能会让你画系统的架构图）
- 在这个项目中你负责了什么、做了什么、担任了什么角色
- 从这个项目中你学会了那些东西，使用到了那些技术，学会了那些新技术的使用
- 另外项目描述中，最好可以体现自己的综合素质，比如你是如何协调项目组成员协同开发的或者在遇到某一个棘手的问题的时候你是如何解决的又或者说你在这个项目用了什么技术实现了什么功能比如：用 redis 做缓存提高访问速度和并发量、使用消息队列削峰和降流等等。

我：我说了我是如何考虑它的需求（秒杀地址隐藏，记录订单，减库存），一开始简单的用 synchronized 锁住方法，出现了问题，后来乐观锁改进，又有瓶颈，再上缓存，出现了缓存雪崩，于是缓存预热，错开缓存失效时间。最后，发现先记录订单再减库存会减少行级锁等待时间。

一面面试官很耐心地听，并给了我一些指导，问我乐观锁是怎么实现的，我说是基于 sql 语句，在减库存操作的 where 条件里加剩余库存数>0，他说这应该不算是一种乐观锁，应该先查库存，在减库存的时候判断当前库存是否与读到的库存一样（可这样不是多一次查询操作吗？不是很理解，不过我没有反驳，只是说理解您的意思。事实证明千万别怼面试官，即使你觉得他说的不对）

面试官：我缓存雪崩什么情况下会发生？如何避免？

我：当多个商品缓存同时失效时会雪崩，导致大量查询数据库。还有就是秒杀刚开始的时候缓存里没有数据。解决方案：缓存预热，错开缓存失效时间

面试官：问我更新数据库的同时为什么不马上更新缓存，而是删除缓存？

我：因为考虑到更新数据库后更新缓存可能会因为多线程下导致写入脏数据（比如线程 A 先更新数据库成功，接下来要取更新缓存，接着线程 B 更新数据库，但 B 又更新了缓存，接着 B 的时间片用完了，线程 A 更新了缓存）

逼逼了将近 30 分钟，面试官居然用周杰伦的语气对我说：



我突然受宠若惊，连忙说谢谢，也正是因为第一次面试得到了面试官的肯定，才让我信心大增，二面稳定发挥。

面试官又曰：我看你还懂数据库是吧，答：略懂略懂。。。那我问个简单的吧！

我：因为这个问题太简单了，所以我忘记它是什么了。

面试官：你还会啥数据库知识？

我：我一听，问的这么随意的吗。。。都让我选题了，我就说我了解索引，慢查询优化，巴拉巴拉

面试官：等等，你说索引是吧，那你能说下索引的存储数据结构吗？

我：我心想这简单啊，我就说 B+树，还说了为什么用 B+树

面试官：你简历上写的这个 J.U.C 包是什么啊？（他居然不知道 JUC）

我：就是 java 多线程的那个包啊。。。

面试官：那你都了解里面的哪些东西呢？

我：哈哈哈！这可是我的强项，从 ConcurrentHashMap，ConcurrentLinkedQueue 说到 CountDownLatch，CyclicBarrier，又说到线程池，分别说了底层实现和项目中的应用。

面试官：我觉得差不多了，那我再问个与技术无关的问题哈，虽然这个问题可能不应该我问，就是你是如何考虑你的项目架构的呢？

我：先用最简单的方式实现它，再去发掘系统的问题和瓶颈，于是查资料改进架构。。。

面试官：好，那我给你介绍下我这边的情况吧

聊天结束

总结：一面可能是简历面吧，问的比较简单，我在讲项目中说出了我做项目时的学习历程和思考，赢得了面试官的好感，感觉他应该给我的评价很好。

二面（33 分钟左右）

然而开心了没一会，内推人问我面的怎么样啊？看我流程已经到大大 boss 那了。我一听二面不是主管吗？？？怎么直接跳了一面。于是瞬间慌了，赶紧（下床）学习准备二面。

隔了一天，3.14 的早上 10: 56 分，杭州的大大 boss 给我打来了电话，卧槽我当时在上毛概课，万恶的毛概课每节课都点名，我还在最后一排不敢跑出去。于是接起电话来怂恿地说不好意思我在上课，晚上可以面试吗？大大 boss 看来很忙啊，跟我说晚上没时间啊，再说吧！

于是又隔了一天，3.16 中午我收到了北京的电话，当时心里小失望，我的大大 boss 呢？？？接起电话来，就是一番狂轰乱炸。。。

第一步还是先自我介绍，这个就不多说了，提前准备好要说的重点就没问题！

面试官：我们还是从你的项目开始吧，说说你的秒杀系统。

我：一面时的套路。。。我考虑到秒杀地址在开始前不应暴露给用户。。。

面试官：等下啊，为什么要这样呢？暴露给用户会怎么样？

我：用户提前知道秒杀地址就可以写脚本来抢购了，这样不公平

面试官：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？

我：我考虑到了这方面，于是我自己写了个 LRU 缓存（划重点，这么多好用的缓存我为啥不用偏要自己写？就是为了让面试官上钩问我是怎么写的，这样我就可以逼逼准备好的内容了！），用这个缓存存储请求的 ip 和用户名，一个 ip 和用户名只能同时通过 3 个请求。

面试官：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。

我：这就是在为难我胖虎啊，我说这种情况跟真实用户操作太像了。。。我没法区别，不过我觉得可以通过地理位置信息或者机器学习算法来做吧。。。

面试官：好的这个问题就到这吧，你接着说

我：我把生成订单和减库存两条 sql 语句放在一个事务里，都操作成功了则认为秒杀成功。

面试官：等等，你这个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？

我：这面试官好变态啊，我只是个本科生？！？！我觉得应该要用分布式锁来实现吧。。。

面试官：有没有更轻量级的做法？

我：不知道了。后来查资料发现可以用消息队列来实现。使用消息队列主要能带来两个好处：(1) 通过异步处理提高系统性能（削峰、减少响应所需时间）；(2) 降低系统耦合性。关于消息队列的更多内容可以查看这篇文章：<https://snailclimb.gitee.io/javaguide/#/system-design/data-communication/message-queue>

后来发现消息队列作用好大，于是现在在学手写一个消息队列。

面试官：好的你接着说项目吧。

我：我考虑到了缓存雪崩问题，于是。。。

面试官：等等，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？

我：我不知道数据库会不会炸，反正我快炸了。当时说没考虑这么高的并发量，后来发现也是可以用消息队列来解决，对流量削峰填谷。

面试官：好项目聊（怼）完了，我们来说说别的，操作系统了解吧，你能说说 NIO 吗？

我：NIO 是。。。

面试官：那你知道 NIO 的系统调用有哪些吗，具体是怎么实现的？

我：当时复习 NIO 的时候就知道是咋回事，不知道咋实现。最近在补这方面的知识，可见 NIO 还是很重要的！

面试官：说说进程切换时操作系统都会发生什么？

我：不如杀了我，我最讨厌操作系统了。简单说了下，可能不对，需要答案自行百度。

面试官：说说线程池？

答：卧槽这我熟啊，把 Java 并发编程的艺术里讲的都说出来了，说了得有十分钟，自夸一波，毕竟这本书我看了五遍😂

面试官：好问问计网吧如果设计一个聊天系统，应该用 TCP 还是 UDP？为什么

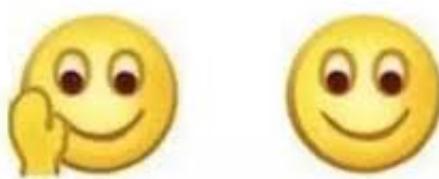
我：当然是 TCP！原因如下：

面试官：好的，你有什么要问我的吗？

我：我还有下一次面试吗？

面试官：应该。应该有的，一周内吧。还告诉我居然转正前要实习三个月？wtf，一个大三满课的本科生让我如何在八月底前实习三个月？

我：面试官再见



三面（46 分钟）

3.18 号，三面来了，这次又是那个大大 boss！

第一步还是先自我介绍，这个就不多说了，提前准备好要说的重点就没问题！

面试官：聊聊你的项目？

我：经过二面的教训，我迅速学习了一下分布式的理论知识，并应用到了我的项目（吹牛逼）中。

面试官：看你用到了 Spring 的事务机制，你能说下 Spring 的事务传播吗？

我：完了这个问题好像没准备，虽然之前刷知乎看到过。。。我就只说出来一条，面试官说其实这个有很多机制的，比如事务嵌套，内事务回滚外事务回滚都会有不同情况，你可以回去看看。

面试官：说说你的分布式事务解决方案？

我：我叭叭的照着资料查到的解决方案说了一通，面试官怎么好像没大听懂？？？

阿里巴巴之前开源了一个分布式 Fescar（一种易于使用，高性能，基于 Java 的开源分布式事务解决方案），后来，Ant Financial 加入 Fescar，使其成为一个更加中立和开放的分布式交易社区，Fescar 重命名为 Seata。Github 地址：<https://github.com/seata/seata>

面试官：好，我们聊聊其他项目，说说你这个 MapReduce 项目？MapReduce 原理了解过吗？

我：我叭叭地说了一通，面试官好像觉得这个项目太简单了。要不是没项目，我会把我的实验写上吗？？？

面试官：你这个手写 BP 神经网络是干了啥？

我：这是我选修机器学习课程时的一个作业，我又对它进行了扩展。

面试官：你能说说为什么调整权值时要沿着梯度下降的方向？

我：老大，你太厉害了，怎么什么都懂。我压根没准备这个项目。。。没想到会问，做过去好几个月了，加上当时一紧张就忘了，后来想起来大概是.....。

面试官：好我们问问基础知识吧，说说什么叫 xisuo？

我：？？？xisuo，您说什么，不好意思我没听清。（这面试官有点口音。。。）就是 xisuo 啊！xisuo 你不知道吗？。。。尴尬了十几秒后我终于意识到，他在说死锁！！！

面试官：假如 A 账户给 B 账户转钱，会发生 xisuo 吗？能具体说说吗？

我：当时答的不好，后来发现面试官又是想问分布式，具体答案参考这个：<https://blog.csdn.net/tylorchan2016/article/details/51039362>

面试官：为什么不考研？

我：不喜欢学术氛围，巴拉巴拉。

面试官：你有什么问题吗？

我：我还有下面吗。。。面试官说让我等，一周内答复。

等了十天，一度以为我凉了，内推人说我流程到 HR 了，让我等着吧可能 HR 太忙了，3.28 号 HR 打来了电话，当时在教室，我直接飞了出去。

HR 面

面试官：你好啊，先自我介绍下吧

我：巴拉巴拉....HR 面的技术面试和技术面的还是有所区别的！

面试官人特别好，一听就是很会说话的小姐姐！说我这里给你悄悄透露下，你的评级是 A 哟！



接下来就是几个经典 HR 面挂人的问题，什么难给我来什么，我看别人的 HR 面怎么都是聊聊天。。。

面试官：你为什么选择支付宝呢，你怎么看待支付宝？

我：我从个人情怀，公司理念，环境氛围，市场价值，趋势导向分析了一波（说白了就是疯狂夸支付宝，不过说实话我说的那些一点都没撒谎，阿里确实做到了。比如我举了个雷军和格力打赌 5 年 2000 亿销售额，大部分企业家关注的是利益，而马云更关注的是真的为人类为世界做一些事情，利益不是第一位的。）

面试官：明白了解，那你的优点我们都很明了了，你能说说你的缺点吗？

缺点肯定不能是目标岗位需要的关键能力！！！

总之，记住一点，面试官问你这个问题的话，你说一些不影响你这个职位工作需要的一些缺点。比如你面试后端工程师，面试官问你的缺点是什么的话，你可以这样说：自己比较内向，平时不太爱与人交流，但是考虑到以后可能要和客户沟通，自己正在努力改。

我：据说这是 HR 面最难的一个问题。。。我当时翻了好几天的知乎才找到一个合适的，也符合我的答案：我有时候会表现的不太自信，比如阿里的内推二月份就开始了，其实我当时已经复习了很久了，但是老是觉得自己还不行，不敢投简历，于是又把书看了一遍才投的，当时也是舍友怂恿一波才投的，面了之后发现其实自己也没有很差。（划重点，一定要把自己的缺点圆回来）。

面试官：HR 好像不太满意我的答案，继续问我还有缺点吗？

我：我说比较容易紧张吧，举了自己大一面实验室因为紧张没进去的例子，后来不断调整心态，现在已经好很多了。

接下来又是个好难的问题。

面试官：BAT 都给你 offer 了，你怎么选？

其实我当时好想说，BT 是什么？不好意思我只知道阿里。

我：哈哈哈哈开玩笑，就说了阿里的文化，支付宝给我们带来很多便利，想加入支付宝为人类做贡献！

最后 HR 问我实习时间，现在大几之类的问题，说肯定会给我发 offer 的，让我等着就好了，希望过两天能收到好的结果。



Bigo的Java面试，我挂在了第三轮技术面上.....

本文是鄙人薛某这位老哥的投稿，虽然面试最后挂了，但是老哥本身还是挺优秀的，而且通过这次面试学到了很多东西，我想这就足够了！加油！不要畏惧面试失败，好好修炼自己，多准备一下，后面一定会找到让自己满意的工作。

背景

前段时间家里出了点事，辞职回家待了一段时间，处理完老家的事情后就回到广州这边继续找工作，大概是国庆前几天我去面试了一家叫做Bigo(YY的子公司)，面试的职位是面向3-5年的Java开发，最终自己倒在了第三轮的技术面上。虽然有些遗憾和泄气，但想着还是写篇博客来记录一下自己的面试过程好了，也算是对广大程序员同胞们的分享，希望对你们以后的学习和面试能有所帮助。

个人情况

先说下LZ的个人情况。

17年毕业，二本，目前位于广州，是一个非常普通的Java开发程序员，算起来有两年多的开发经验。

其实这个阶段有点尴尬，高不成低不就，比初级程序员稍微好点，但也达不到高级的程度。加上现如今IT行业接近饱和，很多岗位都是要求至少3-5年以上开发经验，所以对于两年左右开发经验的需求其实是比较小的，这点在LZ找工作的过程中深有体会。最可悲的是，今年的大环境不好，很多公司不断的在裁员，更别说招人了，残酷的形势对于求职者来说更是雪上加霜，相信很多求职的同学也有所体会。所以，不到万不得已的情况下，建议不要裸辞！

Bigo面试

面试岗位：Java后台开发

经验要求：3-5年

由于是国庆前去面试Bigo的，到现在也有一个多月的时间了，虽然仍有印象，但也有不少面试题忘了，所以我只能尽量按照自己的回忆来描述面试的过程，不明白之处还请见谅！

一面(微信电话面)

bigo的第一面是微信电话面试，本来是想直接电话面，但面试官说需要手写算法题，就改成微信电话面。

- 自我介绍
- 先了解一下Java基础吧，什么是内存泄漏和内存溢出？（溢出是指创建太多对象导致内存空间不足，泄漏是无用对象没有回收）
- JVM怎么判断对象是无用对象？（根搜索算法，从GC Root出发，对象没有引用，就判定为无用对象）
- 根搜索算法中的根节点可以是哪些对象？（类对象，虚拟机栈的对象，常量引用的对象）

- 重载和重写的区别？（重载发生在同个类，方法名相同，参数列表不同；重写是父子类之间的行为，方法名好参数列表都相同，方法体内的程序不同）
- 重写有什么限制没有？
- Java有哪些同步工具？（synchronized和Lock）
- 这两者有什么区别？
- ArrayList和LinkedList的区别？（ArrayList基于数组，搜索快，增删元素慢，LinkedList基于链表，增删快，搜索因为要遍历元素所以效率低）
- 这两种集合哪个比较占内存？（看情况的，ArrayList如果有扩容并且元素没占满数组的话，浪费的内存空间也是比较多的，但一般情况下，LinkedList占用的内存会相对多点，因为每个元素都包含了指向前后节点的指针）
- 说一下HashMap的底层结构（数组 + 链表，链表过长变成红黑树）
- HashMap为什么线程不安全，1.7版本之前HashMap有什么问题（扩容时多线程操作可能会导致链表成环的出现，然后调用get方法会死循环）
- 了解ConcurrentHashMap吗？说一下它为什么能线程安全（用了分段锁）
- 哪些方法需要锁住整个集合的？（读取size的时候）
- 看你简历写着你了解RPC啊，那你说下RPC的整个过程？（从客户端发起请求，到socket传输，然后服务端处理消息，以及怎么序列化之类的大概讲了一下）
- 服务端获取客户端要调用的接口信息后，怎么找到对应的实现类的？（反射 + 注解吧，这里也不是很懂）
- dubbo的负载均衡有几种算法？（随机，轮询，最少活跃请求数，一致性hash）
- 你说的最少活跃数算法是怎么回事？（服务提供者有一个计数器，记录当前同时请求个数，值越小说明该服务器负载越小，路由器会优先选择该服务器）
- 服务端怎么知道客户端要调用的算法的？（socket传递消息过来的时候会把算法策略传递给服务端）
- 你用过redis做分布式锁是吧，你们是自己写的工具类吗？（不是，我们用redission做分布式锁）
- 线程拿到key后是怎么保证不死锁的呢？（给这个key加上一个过期时间）
- 如果这个过期时间到了，但是业务程序还没处理完，该怎么办？（额.....可以在业务逻辑上保证幂等性吧）
- 那如果多个业务都用到分布式锁的话，每个业务都要保证幂等性了，有没有更好的方法？
(额.....思考了下暂时没有头绪，面试官就说那先跳过吧。事后我了解到redission本身是有个看门狗的监控线程的，如果检测到key被持有的话就会再次重置过期时间)
- 你那边有纸和笔吧，写一道算法，用两个栈模拟一个队列的入队和出队。（因为之前复习的时候对这道题有印象，写的时候也比较快，大概是用了五分钟，然后就拍成图片发给了面试官，对方看完后表示没问题就结束了面试。）

第一面问的不算难，问题也都是偏基础之类的，虽然答得不算完美，但过程还是比较顺利的。几天之后，Bigo的hr就邀请我去他们公司参加现场面试。

二面

到Bigo公司后，一位hr小姐姐招待我到了一个会议室，等了大概半个小时，一位中年男子走了进来，非常的客气，说不好意思让我等那么久了，并且介绍了自己是技术经理，然后就开始了我们的交谈。

- 依照惯例，让我简单做下自我介绍，这个过程他也在边看我的简历。
- 说下你最熟悉的项目吧。（我就拿我上家公司最近做的一个电商项目开始介绍，从简单的项目描述，到项目的主要功能，以及我主要负责的功能模块，吧啦吧啦.....）
- 你对这个项目这么熟悉，那你根据你的理解画一下你的项目架构图，还有说下你具体参与了哪部分。（这个题目还是比较麻烦的，毕竟我当时离职的时间也挺长了，对这个项目的架构也是有些模糊。当然，最后还是硬着头皮还是画了个大概，从前端开始访问，然后通过nginx网关层，最后到具体的服务等等，并且把自己参与的服务模块也标示了出来）
- 你的项目用到了Spring Cloud GateWay，既然你已经有nginx做网关了，为什么还要用gateWay呢？（nginx是做负载均衡，还有针对客户端的访问做网关用的，gateWay是接入业务层做的网

关，而且还整合了熔断器Hystrix)

- 熔断器Hystrix最主要的作用是什么？（防止服务调用失败导致的服务雪崩，能降级）
- 你的项目用到了redis，你们的redis是怎么部署的？（额。。。好像是哨兵模式部署的吧。）
- 说一下你对哨兵模式的理解？（我对哨兵模式了解的不多，就大概说了下Sentinel监控之类的，还有类似ping命令的心跳机制，以及怎么判断一个master是下线之类。.....）
- 那你们为什么要用哨兵模式呢？怎么不用集群的方式部署呢？一开始get不到他的点，就说哨兵本身就是多实例部署的，他解释了一下，说的是redis-cluster的部署方案。（额.....redis的环境搭建有专门的运维人员部署的，应该是优先考虑高可用吧.....开始有点心慌了，因为我不知道为什么）
- 哦，那你是觉得集群没有办法实现高可用吗？（不....不是啊，只是觉得哨兵模式可能比较保证主从复制安全性吧.....我也不知道自己在说什么）
- 集群也是能保证高可用的，你知道它又是怎么保证主从一致性的吗？（好吧，这里真的不知道了，只能跳过）
- 你肯定有微信吧，如果让你来设计微信朋友圈的话，你会怎么设计它的属性成员呢？（嗯.....需要有用户表，朋友圈的表，好友表之类的吧）
- 嗯，好，你也知道微信用户有接近10亿之多，那肯定要涉及到分库分表，如果是你的话，怎么设计分库分表呢？（这个问题考察的点比较大，我答的其实一般，而且这个过程面试官还不断的进行连环炮发问，导致这个话题说了有将近20分钟，限于篇幅，这里就不再详述了）
- 这边差不多了，最后你写一道算法吧，有一组未排序的整形数组，你设计一个算法，对数组的元素两两配对，然后输出最大的绝对值差和最小的绝对值差的“对数”。（听到这道题，我第一想法就是用HashMap来保存，key是两个元素的绝对值差，value是配对的数量，如果有相同的就加1，没有就赋值为1，然后最后对map做排序，输出最大和最小的value值，写完后面试官说结果虽然是正确的，但是不够效率，因为遍历的时间复杂度成了 $O(n^2)$ ，然后提醒了我往排序这方面想。我灵机一动，可以先对数组做排序，然后首元素与第二个元素做绝对值差，记为num，然后首元素循环和后面的元素做计算，直到绝对值差不等于num位置，这样效率比起 $O(n^2)$ 快多了。）

面试完后，技术官就问我有什么要问他的，我就针对这个岗位的职责和项目所用的技术栈做了询问，然后就让我先等下，等他去通知三面的技术官。说实话，二面给我的感觉是最舒服的，因为面试官很亲切，面试的过程一直积极的引导我，而且在职业规划方面给了我很多的建议，让我受益匪浅，虽然面试时间有一个半小时，但却丝毫不觉得长，整个面试过程聊得挺舒服的，不过因为时间比较久了，很多问题我也记不清了。

三面

二面结束后半个小时，三面的技术面试官就开始进来了，从他的额头发量分布情况就能猜想是个大牛，人狠话不多，坐下后也没让我做自我介绍，直接开问，整个过程我答的也不好，而且面试官的问题表述有些不太清晰，经常需要跟他重复确认清楚。

- 对事务了解吗？说一下事务的隔离级别有哪些（我以比较了解的Spring来说，把Spring的四种事务隔离级别都叙述了一遍）
- 你做过电商，那应该知道下单的时候需要减库存对吧，假设现在有两个服务A和B，分别操作订单和库存表，A保存订单后，调用B减库存的时候失败了，这个时候A也要回滚，这个事务要怎么设计？（B服务的减库存方法不抛异常，由调用方也就是A服务来抛异常）
- 了解过读写分离吗？（额。。。大概了解一点，就是写的时候进主库，读的时候读从库）
- 你说读的时候读从库，现在假设有一张表User做了读写分离，然后有个线程在一个事务范围内对User表先做了写的处理，然后又做了读的处理，这时候数据还没同步到从库，怎么保证读的时候能读到最新的数据呢？（听完顿时有点懵圈，一时间答不上来，后来面试官说想办法保证一个事务中读写都是同一个库才行）
- 你的项目里用到了rabbitmq，那你说下mq的消费端是怎么处理的？（就是消费端接收到消息之后，会先把消息存到数据库中，然后再从数据库中定时跑消息）
- 也就是说你的mq是先保存到数据库中，然后业务逻辑就是从mq中读取消息然后再处理的是吧？（是的）
- 那你的消息是唯一的吗？（是的，用了唯一约束）

- 你怎么保证消息一定能被消费？或者说怎么保证一定能存到数据库中？（这里开始慌了，因为mq接入那一块我只是看过部分逻辑，但没有亲自参与，凭着自己对mq的了解就答道，应该是靠rabbitmq的ack确认机制）
- 好，那你整理一下你的消费端的整个处理逻辑流程，然后说说你的ack是在哪里返回的（听到这里我的心凉了一截，mq接入这部分我确实没有参与，硬着头皮按照自己的理解画了一下流程，但其实漏洞百出）
- 按照你这样画的话，如果数据库突然宕机，你的消息该怎么确认已经接收？（额.....那发送消息的时候就存放消息可以吧.....回答的时候心里千万只草泥马路过.....行了吧，没玩没了了。）
- 那如果发送端的服务是多台部署呢？你保存消息的时候数据库就一直报唯一性的错误？（好吧，你赢了。。。最后硬是憋出了一句，您说的是，这样设计确实不好。。。。）
- 算了，跳过吧，现在你来设计一个map，然后有两个线程对这个map进行操作，主线程高速增加和删除map的元素，然后有个异步线程定时去删除map中主线程5秒内没有删除的数据，你会怎么设计？
(这道题我答得并不好，做了下简单的思考就说可以把map的key加上时间戳的标志，遍历的时候发现小于当前时间戳5秒前的元素就进行删除，面试官对这样的回答明显不太满意，说这样遍历会影响效率，ps：对这道题，大佬们如果有高见可以在评论区说下！)

.....还有其他问题，但我只记住了这么多，就这样吧。

面完最后一道题后，面试官就表示这次面试过程结束了，让我回去等消息。听到这里，我知道基本上算是宣告结果了。回想起来，自己这一轮面试确实表现的很一般，加上时间拖得很长，从当天的2点半一直面试到6点多，精神上也尽显疲态。果然，几天之后，hr微信通知了我，说我第三轮技术面试没有通过，这一次面试以失败告终。

总结

以上就是面试的大概过程，不得不说，大厂的面试还是非常有技术水平的，这个过程中我学到了很多，这里分享下个人的一些心得：

- 1、**基础！基础！基础！**重要的事情说三遍，无论是什么阶段的程序员，基础都是最重要的。每个公司的面试一定会涉及到基础知识的提问，如果你的基础不扎实，往往第一面就可能被淘汰。
- 2、**简历需要适当的包装。**老实说，我的简历肯定是经过包装的，这也是我的工作年限不够，但却能获取Bigo面试机会的重要原因，所以适当的包装一下简历很有必要，不过切记一点，就是不能脱离现实，比如明明只有两年经验，却硬是写到三年。小厂还可能蒙混过关，但大厂基本很难，因为很多公司会在入职前做背景调查。
- 3、**要对简历上的技术点很熟悉。**简历包装可以，但一定要对简历上的技术点很熟悉，比如只是简单写过rabbitmq的demo的话，就不要写“熟悉”等字眼，因为很多的面试官会针对一个技能点问的很深入，像连环炮一样的深耕你对这个技能点的理解程度。
- 4、**简历上的项目要非常熟悉。**一般我们写简历都是需要对自己的项目做一定程序的包装和美化，项目写得好能给简历加很多分。但一定要对项目非常的熟悉，不熟悉的模块最好不要写上去。笔者这次就吃了大亏，我的简历上有个电商项目就写到了用rabbitmq处理下单，虽然稍微了解过那部分下单的处理逻辑，但由于没有亲自参与就没有做深入的了解，面试时在这一块内容上被Bigo三面的面试官逼得最后哑口无言。
- 5、**提升自己的架构思维。**对于初中级程序员来说，日常的工作就是基本的增删改查，把功能实现就完事了，这种思维不能说不好，只是想更上一层楼的话，业务时间需要提升下自己的架构思维能力，比如说如果让你接手一个项目的话，你会怎么考虑设计这个项目，从整体架构，到引入一些组件，再到设计具体的业务服务，这些都是设计一个项目必须要考虑的环节，对于提升我们的架构思维是一种很好的锻炼，这也是很多大厂面试高级程序员时的重要考察部分。

6、不要裸辞。这也是我最朴实的建议了，大环境不好，且行且珍惜吧，唉~~~~

总的来说，这次面试Bigo还是收获颇丰的，虽然有点遗憾，但也没什么后悔的，毕竟自己面试之前也是准备的很充分了，有些题目答得不好说明我还有很多技术盲区，不懂就是不懂，再这么吹也吹不出来。这也算是给我提了个醒，你还嫩着呢，好好修炼内功吧，毕竟菜可是原罪啊。

2020年字节跳动面试总结

本文来自读者 Boyn 投稿！恭喜这位粉丝拿到了含金量极高的字节跳动实习 offer！赞！

基本条件

本人是底层 211 本科，现在大三，无科研经历，但是有一些项目经历，在国内监控行业某头部企业做过一段时间的实习。想着投一下字节，可以积累一下面试经验和为春招做准备。投了简历之后，过了一段时间，HR 就打电话跟我约时间，在年后进行远程面。

说明一下，我投的是北京 office。

一面

面试官很和蔼，由于疫情的原因，大家都在家里面进行远程面试

开头没有自我介绍，直接开始问项目了，问了比如

- 常用的 Web 组件有哪些（回答了自己经常用到的 SpringBoot,Redis,Mysql 等等，字节这边基本没有用 Java 的后台，所以感觉面试官不大会问 Spring,Java 这些东西，反倒是对数据库和中间件比较感兴趣）
- Kafka 相关，如何保证不会重复消费，Kafka 消费组结构等等（这个只是凭着感觉和面试官说了，因为 Kafka 自己确实准备得不充分，但是心态稳住了）
- Mysql 索引，B+树（必考嗷同学们）

还有一些项目中的细节，这些因人而异，就不放上来了，提示一点就是要在项目中介绍一些亮眼的地方，比如用了什么牛逼的数据结构，架构上有什么特点，并发量大小还有怎么去 hold 住并发量

后面就是算法题了，一共做了两道

1. 判断平衡二叉树（这道题总体来说并不难，但是面试官在中间穿插了垃圾回收的知识，这就很难受了，具体的就是大家要判断一下对象在什么时候会回收，可达性分析什么时候对这个对象来说是不可达的，还有在递归函数中内存如何变化，这个是让我们来对这个函数进行执行过程的建模，只看栈帧大小变化的话，应该有是两个峰值，中间会有抖动的情况）
2. 二分查找法的变种题，给定 target 和一个升序的数组，寻找下一个比数组大的数。这道题也不难，靠大家对二分查找法的熟悉程度，当然，这边还有一个优化的点，可以看看[我的博客](#)找找灵感

完成了之后，面试官让我等一会有二面，大概 10 分钟左右吧，休息了一会就继续了

二面

二面一上来就是先让我自我介绍，当然还是同样的套路，同样的香脆

然后问了我一些关于 Redis 的问题，比如 zset 的实现（跳表，这个高频），键的过期策略，持久化等等，这些在大多数 Redis 的介绍中都可以找到，就不细说了

还有一些数据结构的问题,比如说问了哈希表是什么,给面试官详细说了一下 `java.util.HashMap` 是怎么实现(当然里面就穿插着红黑树了,多看看红黑树是有什么特点之类的)的,包括说为什么要用链地址法来避免冲突,探测法有哪些,链地址法和探测法的优劣对比

后面还跟我讨论了很久的项目,所以说大家的项目一定要做好,要有亮点的地方,在这里跟面试官讨论了很多项目优化的地方,还有什么不足,还有什么地方可以新增功能等等,同样不细说了

一边讨论的时候劈里啪啦敲了很多,应该是对个人的面试评价一类的

后面就是字节的传统艺能手撕算法了,一共做了三道

- 一二道是连在一起的.给定一个规则 $S_0 = \{1\}$ $S_1 = \{1, 2, 1\}$ $S_2 = \{1, 2, 1, 3, 1, 2, 1\}$ $S_n = \{S_{n-1}, n + 1, S_{n-1}\}$.第一个问题是他们的个数有什么关系(1 3 7 15... 2 的 n 次方-1,用位运算解决).第二个问题是给定数组个数下标 n 和索引 k ,让我们求出 $S_n(k)$ 所指的数,假如 $S_2(2) = 1$,我在做的时候没有什么好的思路,如果说有的话大家可以分享一下
- 第三道是下一个排列: <https://leetcode-cn.com/problems/next-permutation> 的题型,不过做了一些修改,数组大小 $10000 < n < 100000$,不能用暴力法,还有数字是在 1-9 之间会有重复

hr 面

一些偏职业规划的话题了,实习时间,项目经历,实习经历这些。

总结

基础很重要!这次准备到的 Redis, Mysql, JVM 原理等等都有问到了,(网络这一块没问,但是也是要好好准备的,对于后台来说,网络知识不仅仅是面试,还是以后工作的知识基础).当然自己也有准备不足的地方,比如 Kafka 等中间件,只会用不会原理是万万不行的.并且这些基础知识不能只靠背,面试官还会融合在项目里面进行串问

问到了不会的不要慌,因为面试官是在试探你的技术深度,有可能会针对某一个问题,问到你不会为止,所以你出现不会的问题是很正常的,心态把控住就行.

无论是做题,还是回答问题的时候,牢记你不是在考试,而是在交流,和面试官有互动和沟通是很重要的,你说的一些疏漏的地方,如果你及时跟面试官反馈,还是可以补救一下的

最重要的一点字节的面试就是算法一定要牢固,每一轮都会有手撕算法的,这个不用想,LeetCode+剑指 Offer 走起来就对了,心态很重要,算法题不一定都是你会的,要有一定的心理准备,遇到难题可以先冷静分析一波.而且写出 `Bug free` 的代码也是很重要的,我前面的几题算法因为在牛客网上进行面试,所以要运行出来.

最后祝大家在春招取得好的 Offer, 奥力给!

2019年蚂蚁金服、头条、拼多多的面试总结

作者: rhwayfun,原文地址: <https://mp.weixin.qq.com/s/msYty4vjjC0PvrwasRH5Bw>, JavaGuide 已经获得作者授权并对原文进行了重新排版。

文章有点长,请耐心看完,绝对有收获!不想听我BB直接进入面试分享:

- 准备过程

- 蚂蚁金服面试分享
- 拼多多面试分享
- 字节跳动面试分享
- 总结

说起来开始进行面试是年前倒数第二周，上午9点，我还在去公司的公交上，突然收到蚂蚁的面试电话，其实算不上真正的面试。面试官只是和我聊了下他们在做的事情（主要是做双十一这里大促的稳定性保障，偏中间件吧），说的很详细，然后和我沟通了下是否有兴趣，我表示有兴趣，后面就收到正式面试的通知，最后没选择去蚂蚁表示抱歉。

当时我自己也准备出去看看机会，顺便看看自己的实力。当时我其实挺纠结的，一方面现在部门也正需要我，还是可以有一番作为的，另一方面觉得近一年来进步缓慢，没有以前飞速进步的成就感了，而且业务和技术偏于稳定，加上自己也属于那种比较懒散的人，骨子里还是希望能够突破现状，持续在技术上有所精进。

在开始正式的总结之前，还是希望各位同仁能否听我继续发泄一会，抱拳！

我翻开自己2018年初立的flag，觉得甚是惭愧。其中就有一条是保持一周写一篇博客，奈何中间因为各种原因没能坚持下去。细细想来，主要是自己没能真正静下来心认真投入到技术的研究和学习，那么为什么会这样？说白了还是因为没有确定目标或者目标不明确，没有目标或者目标不明确都可能导致行动的失败。

那么问题来了，目标是啥？就我而言，短期目标是深入研究某一项技术，比如最近在研究mysql，那么深入研究一定要动手实践并且有所产出，这就够了么？还需要我们能够举一反三，结合实际开发场景想一想日常开发要注意什么，这中间有没有什么坑？可以看出，要进步真的不是一件简单的事，这种反人类的行为需要我们克服自我的弱点，逐渐形成习惯。真正牛逼的人，从不觉得认真学习是一件多么难的事，因为这已经形成了他的习惯，就喝早上起床刷牙洗脸那么简单。

扯了那么多，开始进入正题，先后进行了蚂蚁、拼多多和字节跳动的面试。

准备过程

先说说自己的情况，我2016先在蚂蚁实习了将近三个月，然后去了我现在的老东家，2.5年工作经验，可以说毕业后就一直老老实实的老东家打怪升级，虽说有蚂蚁的实习经历，但是因为时间太短，还是有点虚的。所以面试官看到我简历第一个问题绝对是这样的。

“哇，你在蚂蚁待过，不错啊”，面试官笑嘻嘻地问到。“是的，还好”，我说。“为啥才三个月？”，面试官脸色一沉问到。“哗啦啦解释一通。。。”，我解释道。“哦，原来如此，那我们开始面试吧”，面试官一本正经说到。

尼玛，早知道不写蚂蚁的实习经历了，后面仔细一想，当初写上蚂蚁不就给简历加点料嘛。

言归正传，准备过程其实很早开始了（当然这不是说我工作时老想着跳槽，因为我明白现在的老东家并不是终点，我还需要不断提升），具体可追溯到从蚂蚁离职的时候，当时出来也面了很多公司，没啥大公司，面了大概5家公司，都拿到offer了。

工作之余常常会去额外研究自己感兴趣的技术以及工作用到的技术，力求把原理搞明白，并且会自己实践一把。此外，买了N多书，基本有时间就会去看，补补基础，什么操作系统、数据结构与算法、MySQL、JDK之类的源码，基本都好好温习了（文末会列一下自己看过的书和一些好的资料）。**我深知基础就像“木桶效应”的短板，决定了能装多少水。**

此外，在正式决定看机会之前，我给自己列了一个提纲，主要包括Java要掌握的核心要点，有不懂的就查资料搞懂。我给自己定位还是Java工程师，所以Java体系是一定要做到心中有数的，很多东西没有常年的积累面试的时候很容易露馅，学习要对得起自己，不要骗人。

剩下的就是找平台和内推了，除了蚂蚁，头条和拼多多都是找人内推的，感谢蚂蚁面试官对我的欣赏，以后说不定会去蚂蚁咯😊。

平台：脉脉、GitHub、v2

蚂蚁金服



技术视点

- 一面
- 二面
- 三面
- 四面
- 五面
- 小结

一面

一面就做了一道算法题，要求两小时内完成，给了长度为N的有重复元素的数组，要求输出第10大的数。典型的TopK问题，快排算法搞定。

算法题要注意的是合法性校验、边界条件以及异常的处理。另外，如果要写测试用例，一定要保证测试覆盖场景尽可能全。加上平时刷刷算法题，这种考核应该没问题的。

二面

- 自我介绍下呗
- 开源项目贡献过代码么？（Dubbo提过一个打印accesslog的bug算么）
- 目前在部门做什么，业务简单介绍下，内部有哪些系统，作用和交互过程说下
- Dubbo踩过哪些坑，分别是怎么解决的？（说了异常处理时业务异常捕获的问题，自定义了一个异常拦截器）
- 开始进入正题，说下你对线程安全的理解（多线程访问同一个对象，如果不考虑额外的同步，调用对象的行为就可以获得正确的结果就是线程安全）
- 事务有哪些特性？（ACID）
- 怎么理解原子性？（同一个事务下，多个操作要么成功要么失败，不存在部分成功或者部分失败的情况）
- 乐观锁和悲观锁的区别？（悲观锁假定会发生冲突，访问的时候都要先获得锁，保证同一个时刻只有线程获得锁，读读也会阻塞；乐观锁假设不会发生冲突，只有在提交操作的时候检查是否有冲突）这两种锁在Java和MySQL分别是怎么实现的？（Java乐观锁通过CAS实现，悲观锁通过

synchronize实现。mysql乐观锁通过MVCC，也就是版本实现，悲观锁可以通过select... for update加上排它锁)

- HashMap为什么不是线程安全的？（多线程操作无并发控制，顺便说了在扩容的时候多线程访问时会造成死锁，会形成一个环，不过扩容时多线程操作形成环的问题再JDK1.8已经解决，但多线程下使用HashMap还会有一些其他问题比如数据丢失，所以多线程下不应该使用HashMap，而应该使用ConcurrentHashMap）怎么让HashMap变得线程安全？（Collections的synchronize方法包装一个线程安全的Map，或者直接用ConcurrentHashMap）两者的区别是什么？（前者直接在put和get方法加了synchronize同步，后者采用了分段锁以及CAS支持更高的并发）
- jdk1.8对ConcurrentHashMap做了哪些优化？（插入的时候如果数组元素使用了红黑树，取消了分段锁设计，synchronize替代了Lock锁）为什么这样优化？（避免冲突严重时链表多长，提高查询效率，时间复杂度从O(N)提高到O(logN)）
- redis主从机制了解么？怎么实现的？
- 有过GC调优的经历么？（有点虚，答得不是很好）
- 有什么想问的么？

三面

- 简单自我介绍下
- 监控系统怎么做的，分为哪些模块，模块之间怎么交互的？用的什么数据库？（MySQL）使用什么存储引擎，为什么使用InnoDB？（支持事务、聚簇索引、MVCC）
- 订单表有做拆分么，怎么拆的？（垂直拆分和水平拆分）
- 水平拆分后查询过程描述下
- 如果落到某个分片的数据很大怎么办？（按照某种规则，比如哈希取模、range，将单张表拆分为多张表）
- 哈希取模会有什么问题么？（有的，数据分布不均，扩容缩容相对复杂）
- 分库分表后怎么解决读写压力？（一主多从、多主多从）
- 拆分后主键怎么保证唯一？（UUID、Snowflake算法）
- Snowflake生成的ID是全局递增唯一么？（不是，只是全局唯一，单机递增）
- 怎么实现全局递增的唯一ID？（讲了TDDL的一次取一批ID，然后再本地慢慢分配的做法）
- Mysql的索引结构说下（说了B+树，B+树可以对叶子结点顺序查找，因为叶子结点存放了数据结点且有序）
- 主键索引和普通索引的区别（主键索引的叶子结点存放了整行记录，普通索引的叶子结点存放了主键ID，查询的时候需要做一次回表查询）一定要回表查询么？（不一定，当查询的字段刚好是索引的字段或者索引的一部分，就可以不用回表，这也是索引覆盖的原理）
- 你们系统目前的瓶颈在哪里？
- 你打算怎么优化？简要说下你的优化思路
- 有什么想问我么？

四面

- 介绍下自己
- 为什么要逆向？
- 怎么理解微服务？
- 服务治理怎么实现的？（说了限流、压测、监控等模块的实现）
- 这个不是中间件做的事么，为什么你们部门做？（当时没有单独的中间件团队，微服务刚搞不久，需要进行监控和性能优化）
- 说说Spring的生命周期吧
- 说说GC的过程（说了young gc和full gc的触发条件和回收过程以及对象创建的过程）
- CMS GC有什么问题？（并发清除算法，浮动垃圾，短暂停顿）
- 怎么避免产生浮动垃圾？（记得有个VM参数设置可以让扫描新生代之前进行一次young gc，但是因为gc是虚拟机自动调度的，所以不保证一定执行。但是还有参数可以让虚拟机强制执行一次young gc）
- 强制young gc会有什么问题？（STW停顿时间变长）

- 知道G1么？（了解一点）
- 回收过程是怎么样的？（young gc、并发阶段、混合阶段、full gc，说了Remember Set）
- 你提到的Remember Set底层是怎么实现的？
- 有什么想问的么？

五面

五面是HRBP面的，和我提前预约了时间，主要聊了之前在蚂蚁的实习经历、部门在做的事情、职业发展、福利待遇等。阿里面试官确实是具有一票否决权的，很看重你的价值观是否match，一般都比较喜欢皮实的候选人。HR面一定要诚实，不要说谎，只要你说谎HR都会去证实，直接cut了。

- 之前蚂蚁实习三个月怎么不留下来？
- 实习的时候主管是谁？
- 实习做了哪些事情？（尼玛这种也问？）
- 你对技术怎么看？平时使用什么技术栈？（阿里HR真的是既当爹又当妈，😂）
- 最近有在研究什么东西么
- 你对SRE怎么看
- 对待遇有什么预期么

最后HR还对我说目前稳定性保障部挺缺人的，希望我尽快回复。

小结

蚂蚁面试比较重视基础，所以Java那些基本功一定要扎实。蚂蚁的工作环境还是挺赞的，因为我面的是稳定性保障部门，还有许多单独的小组，什么三年1班，很有青春的感觉。面试官基本水平都比较高，基本都P7以上，除了基础还问了不少架构设计方面的问题，收获还是挺大的。

拼多多



- 面试前
- 一面
- 二面
- 三面
- 小结

面试前

面完蚂蚁后，早就听闻拼多多这个独角兽，决定也去面一把。首先我在脉脉找了一个拼多多的HR，加了微信聊了下，发了简历便开始我的拼多多面试之旅。这里要非常感谢拼多多HR小姐姐，从面试内推到offer确认一直都在帮我，人真的很nice。

一面

- 为啥蚂蚁只待了三个月？没转正？（转正了，解释了一通。。。）
- Java中的HashMap、TreeMap解释下？（TreeMap红黑树，有序，HashMap无序，数组+链表）
- TreeMap查询写入的时间复杂度多少？($O(\log N)$)

- HashMap多线程有什么问题? (线程安全, 死锁)怎么解决? (jdk1.8用了synchronize + CAS, 扩容的时候通过CAS检查是否有修改, 是则重试)重试会有什么问题么? (CAS (Compare And Swap)是比较和交换, 不会导致线程阻塞, 但是因为重试是通过自旋实现的, 所以仍然会占用CPU时间, 还有ABA的问题)怎么解决? (超时, 限定自旋的次数, ABA可以通过原理变量AtomicStampedReference解决, 原理利用版本号进行比较)超过重试次数如果仍然失败怎么办? (synchronize互斥锁)
- CAS和synchronize有什么区别? 都用synchronize不行么? (CAS是乐观锁, 不需要阻塞, 硬件级别实现的原子性; synchronize会阻塞, JVM级别实现的原子性。使用场景不同, 线程冲突严重时CAS会造成CPU压力过大, 导致吞吐量下降, synchronize的原理是先自旋然后阻塞, 线程冲突严重仍然有较高的吞吐量, 因为线程都被阻塞了, 不会占用CPU)
- 如果要保证线程安全怎么办? (ConcurrentHashMap)
- ConcurrentHashMap怎么实现线程安全的? (分段锁)
- get需要加锁么, 为什么? (不用, volatile关键字)
- volatile的作用是什么? (保证内存可见性)
- 底层怎么实现的? (说了主内存和工作内存, 读写内存屏障, happen-before, 并在纸上画了线程交互图)
- 在多核CPU下, 可见性怎么保证? (思考了一会, 总线嗅探技术)
- 聊项目, 系统之间是怎么交互的?
- 系统并发多少, 怎么优化?
- 给我一张纸, 画了一个九方格, 都填了数字, 给一个MN矩阵, 从1开始逆时针打印这MN个数, 要求时间复杂度尽可能低 (内心OS: 之前貌似碰到过这题, 最优解是怎么实现来着) 思考中。。。
- 可以先说下你的思路(想起来了, 说了什么时候要变换方向的条件, 向右、向下、向左、向上, 依此循环)
- 有什么想问我的?

二面

- 自我介绍下
- 手上还有其他offer么? (拿了蚂蚁的offer)
- 部门组织结构是怎样的? (这轮不是技术面么, 不过还是老老实实说了)
- 系统有哪些模块, 每个模块用了哪些技术, 数据怎么流转的? (面试官有点秃顶, 一看级别就很高) 给了我一张纸, 我在上面简单画了下系统之间的流转情况
- 链路追踪的信息是怎么传递的? (RpcContext的attachment, 说了Span的结构:parentSpanId + curSpanId)
- SpanId怎么保证唯一性? (UUID, 说了下内部的定制改动)
- RpcContext是在什么维度传递的? (线程)
- Dubbo的远程调用怎么实现的? (讲了读取配置、拼装url、创建Invoker、服务导出、服务注册以及消费者通过动态代理、filter、获取Invoker列表、负载均衡等过程 (哗啦啦讲了10多分钟), 我可以喝口水么)
- Spring的单例是怎么实现的? (单例注册表)
- 为什么要单独实现一个服务治理框架? (说了下内部刚搞微服务不久, 主要对服务进行一些监控和性能优化)
- 谁主导的? 内部还在使用么?
- 逆向有想过怎么做成通用么?
- 有什么想问的么?

三面

二面老大面完后就直接HR面了, 主要问了些职业发展、是否有其他offer、以及入职意向等问题, 顺便说了下公司的福利待遇等, 都比较常规啦。不过要说的是手上有其他offer或者大厂经历会有一定加分。

小结

拼多多的面试流程就简单许多，毕竟是一个成立三年多的公司。面试难度中规中矩，只要基础扎实应该不是问题。但不得不说工作强度很大，开始面试前HR就提前和我确认能否接受这样强度的工作，想来的老铁还是要做好准备

字节跳动



技术视点

- 面试前
- 一面
- 二面
- 小结

面试前

头条的面试是三家里最专业的，每次面试前有专门的HR和你约时间，确定OK后再进行面试。每次都是通过视频面试，因为都是之前都是电话面或现场面，所以视频面试还是有点不自然。也有人觉得视频面试体验很赞，当然萝卜青菜各有所爱。最坑的二面的时候对方面试官的网络老是掉线，最后很冤枉的挂了（当然有一些点答得不好也是原因之一）。所以还是有点遗憾的。

一面

- 先自我介绍下
- 聊项目，逆向系统是什么意思
- 聊项目，逆向系统用了哪些技术
- 线程池的线程数怎么确定？
- 如果是IO操作为主怎么确定？
- 如果计算型操作又怎么确定？
- Redis熟悉么，了解哪些数据结构？（说了zset） zset底层怎么实现的？（跳表）
- 跳表的查询过程是怎么样的，查询和插入的时间复杂度？（说了先从第一层查找，不满足就下沉到第二层找，因为每一层都是有序的，写入和插入的时间复杂度都是 $O(\log N)$ ）
- 红黑树了解么，时间复杂度？（说了是N叉平衡树， $O(\log N)$ ）
- 既然两个数据结构时间复杂度都是 $O(\log N)$ ，zset为什么不用红黑树（跳表实现简单，踩坑成本低，红黑树每次插入都要通过旋转以维持平衡，实现复杂）
- 点了点头，说下Dubbo的原理？（说了服务注册与发布以及消费者调用的过程）踩过什么坑没有？（说了dubbo异常处理的和打印accesslog的问题）
- CAS了解么？（说了CAS的实现）还了解其他同步机制么？（说了synchronize以及两者的区别，一个乐观锁，一个悲观锁）
- 那我们做一道题吧，数组A， 2^*n 个元素，n个奇数、n个偶数，设计一个算法，使得数组奇数下标位置放置的都是奇数，偶数下标位置放置的都是偶数
- 先说下你的思路（从0下标开始遍历，如果是奇数下标判断该元素是否奇数，是则跳过，否则从该位置寻找下一个奇数）
- 下一个奇数？怎么找？（有点懵逼，思考中。。。）
- 有思路么？（仍然是先遍历一次数组，并对下标进行判断，如果下标属性和该位置元素不匹配从

当前下标的下一个遍历数组元素，然后替换)

- 你这样时间复杂度有点高，如果要求O(N)要怎么做（思考一会，答道“定义两个指针，分别从下标0和1开始遍历，遇见奇数位是偶数和偶数位是奇数就停下，交换内容”）
- 时间差不多了，先到这吧。你有什么想问我的？

二面

- 面试官和蔼很多，你先介绍下自己吧
- 你对服务治理怎么理解的？
- 项目中的限流怎么实现的？（Guava ratelimiter，令牌桶算法）
- 具体怎么实现的？（要点是固定速率且令牌数有限）
- 如果突然很多线程同时请求令牌，有什么问题？（导致很多请求积压，线程阻塞）
- 怎么解决呢？（可以把积压的请求放到消息队列，然后异步处理）
- 如果不用消息队列怎么解决？（说了RateLimiter预消费的策略）
- 分布式追踪的上下文是怎么存储和传递的？（ThreadLocal + spanId，当前节点的spanId作为下个节点的父spanId）
- Dubbo的RpcContext是怎么传递的？（ThreadLocal）主线程的ThreadLocal怎么传递到线程池？
(说了先在主线程通过ThreadLocal的get方法拿到上下文信息，在线程池创建新的ThreadLocal并把之前获取的上下文信息设置到ThreadLocal中。这里要注意的线程池创建的ThreadLocal要在finally中手动remove，不然会有内存泄漏的问题)
- 你说的内存泄漏具体是怎么产生的？（说了ThreadLocal的结构，主要分两种场景：主线程仍然对ThreadLocal有引用和主线程不存在对ThreadLocal的引用。第一种场景因为主线程仍然在运行，所以还是有对ThreadLocal的引用，那么ThreadLocal变量的引用和value是不会被回收的。第二种场景虽然主线程不存在对ThreadLocal的引用，且该引用是弱引用，所以会在gc的时候被回收，但是对用的value不是弱引用，不会被内存回收，仍然会造成内存泄漏）
- 线程池的线程是不是必须手动remove才可以回收value？（是的，因为线程池的核心线程是一直存在的，如果不清理，那么核心线程的threadLocals变量会一直持有ThreadLocal变量）
- 那你说的内存泄漏是指主线程还是线程池？（主线程）
- 可是主线程不是都退出了，引用的对象不应该会主动回收么？（面试官和内存泄漏杠上了），沉默了一会。。。
- 那你说下SpringMVC不同用户登录的信息怎么保证线程安全的？（刚才解释的有点懵逼，一下没反应过来，居然回答成锁了。大脑有点晕了，此时已经一个小时过去了，感觉情况不妙。。。）
- 这个直接用ThreadLocal不就可以么，你见过SpringMVC有锁实现的代码么？（有点晕菜。。。）
- 我们聊聊mysql吧，说下索引结构（说了B+树）
- 为什么使用B+树？（说了查询效率高，O(logN)，可以充分利用磁盘预读的特性，多叉树，深度小，叶子结点有序且存储数据）
- 什么是索引覆盖？（忘记了。。。）
- Java为什么要设计双亲委派模型？
- 什么时候需要自定义类加载器？
- 我们做一道题吧，手写一个对象池
- 有什么想问我的么？（感觉我很多点都没答好，是不是挂了（结果真的是））

小结

头条的面试确实很专业，每次面试官会提前给你发一个视频链接，然后准点开始面试，而且考察的点都比较全。

面试官都有一个特点，会抓住一个值得深入的点或者你没说清楚的点深入下去直到你把这个点讲清楚，不然面试官会觉得你并没有真正理解。二面试官给了我一点建议，研究技术的时候一定要去研究产生的背景，弄明白在什么场景解决什么特定的问题，其实很多技术内部都是相通的。很诚恳，还是很感谢这位面试官大大。

总结

从年前开始面试到头条面完大概一个多月的时间，真的有点身心俱疲的感觉。最后拿到了拼多多、蚂蚁的offer，还是蛮幸运的。头条的面试对我帮助很大，再次感谢面试官对我的诚恳建议，以及拼多多的HR对我的啰嗦的问题详细解答。

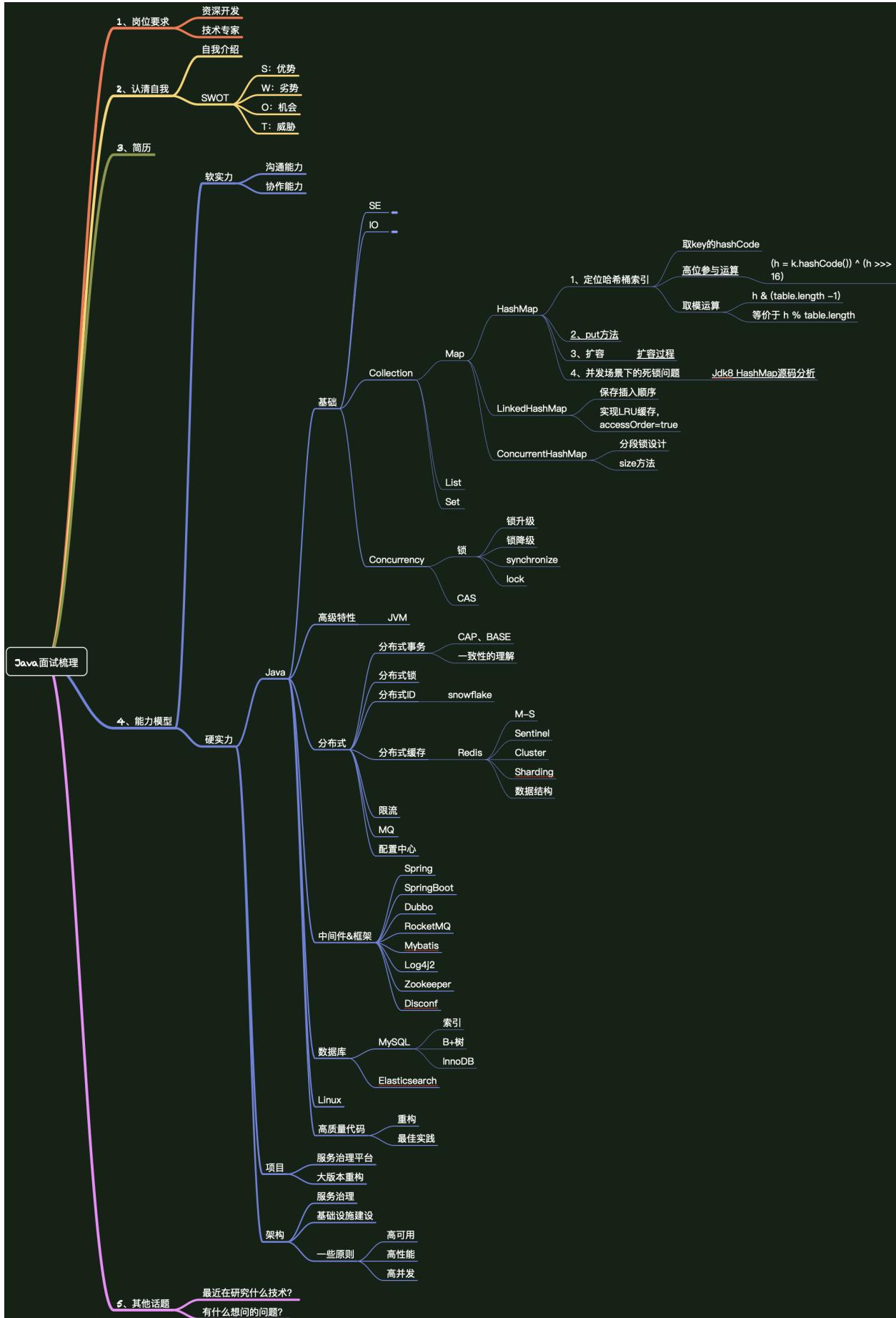
这里要说的是面试前要做好两件事：简历和自我介绍，简历要好好回顾下自己做的一些项目，然后挑几个亮点项目。自我介绍基本每轮面试都有，所以最好提前自己练习下，想好要讲哪些东西，分别怎么讲。此外，简历提到的技术一定是自己深入研究过的，没有深入研究也最好找点资料预热下，不打无准备的仗。

这些年看过的书：

《Effective Java》、《现代操作系统》、《TCP/IP详解：卷一》、《代码整洁之道》、《重构》、《Java程序性能优化》、《Spring实战》、《Zookeeper》、《高性能MySQL》、《亿级网站架构核心技术》、《可伸缩服务架构》、《Java编程思想》

说实话这些书很多只看了一部分，我通常会带着问题看书，不然看着看着就睡着了，简直是催眠良药。

最后，附一张自己面试前准备的脑图：



逆风而行！从考研失败到收获到自己满意的Offer,分享一下自己的经历！

个人情况

我本科是某双非一本，大学四年也没做过太多有成就的事情。和很多在校生一样，我也经历过很迷茫的时间段，倒腾过单片机。

当时还出于对黑客的崇拜，折腾过一个月的网络安全。反正什么都去接触一点，以此来消磨我无聊的时间，不过后面谈了女朋友就不无聊了，哈哈。

Guide哥：竟然有女朋友！

唯一感觉有收获的应该就是呆过 ACM 训练营，无奈自己太菜，拿的奖项都很小，蓝桥杯省一等奖这种水平。从大三开始，给自己明确了目标，还是老老实实学习一个领域的技术吧。当时从知乎上查看了有很多方向，前端，后端，大数据，人工智能。根据我自己的兴趣(好就业)给自己明确了Java后端开发的方向。

考研

当时出于想继续学习提升自己的目的，选择了考研。这个地方想说一点就是，到大三了一定要规划好自己将来要做什么考研，就业，考公务员等等，坚定自己的信心和决心！不要像我一样，在考研开始到结束的期间总会在某个时间段会心态上波动，觉得一整年的考研可能因此错过很多的机会，比如秋招。万一最后没考上研，就很尴尬了，毕业即失业？

尤其是自己考研期间复习不理想的时候，胡思乱想的东西就会越来越多。经常会找同学，朋友以及考上的学长谈心来调节自己的心态。这个地方特别想感谢我的女朋友，在我每次心态爆炸，迷茫想放弃的时候，都愿意花自己的时间陪我出去散心，虽然她也在备战考研。有机会的话，还是建议能找几个比较自律的研友，可以互相督促约束。

Guide哥：此处闪一下这位老哥的女朋友。

这一切都过来的时候，才会觉得自己当初的想法比较幼稚，天无绝人之路。既然选择了远方，便只顾风雨兼程。专心做好一件事就行，只要自己保持上进心，相信未来一定会越来越好，一切美好都将与你环环相扣。

好在我最后还是坚持的走完了考研的旅程，虽然结果不那么的美好，但是我觉得一切都是值得的，至少我的计算机基础，高数，英语在这一年里都得到了很大的提高。

准备春招

我从考研结束之后，就开始着手准备春招的内容，复习以前做过的项目和学习过的技术栈。由于时间比较紧，任务比较重。这个时候，我觉得可以面向面经来学习准备，我花了一个上午的时间去牛客网刷面经，

最终按照不同的模块整理了一份不重复的面试常见问题，接着一切的学习任务都围绕着这个面试题来展开复习，查阅相关的书籍资料。

总结了一下，需要准备的内容也就是：

1. 算法
2. 项目
3. 牛客网总结的常见面试知识点的复习。

算法的话，我的时间比较紧，复习的主要是《剑指offer》 + leetcode的top100。刚开始可以按分专题模块来刷，后面就可以随机练习。

项目的话，我觉得如果有机会能接触到真实的项目是非常好的，因为这一块当你面试的时候针对某些细节你可以自信的和面试官聊很多，如果要是自己包装的话，可能聊起来会觉得很虚。不过也没有关系，即使是你跟着网课学习的或者找的开源项目，我觉得首先得保证能完全吃透这个项目的细节，细到数据库的表各个字段的含义，项目中哪些功能在哪一个模块实现，为什么这样实现，有没有更好的实现方式了。这些我觉得都是你需要思考的问题，因为面试中会出现各种不同的情况，面对不同的面试官，问的问题也是千奇百怪的。

关于项目经历，我再补充一下，避免大家踩坑。

不管是网课的项目，还是开源的项目。你能发现，别人也能发现，怎么才能避免雷同，体现自己的特色，项目中真正具有你的思考在里面。我有如下建议送给你：

1. 可以替换其中的相关技术栈(比如 kafka 换成RocketMQ)，同时还需要准备自己选型这个技术栈的理由，一定要能够自圆其说。
2. 可以自己在这个项目的基础之上添加一些额外的功能。这些内容都算是你自己写的，也是自己思考的点，面试的时候可以自信的和面试官介绍。对于项目介绍的部分，我觉得可以主动突出自己的亮点和难点。比如常见的考察JVM相关的问题，可以通过"自己创造难点，遇到的问题"来将这个问题主动出来，将主动权握在自己的手中。比如我当时为了说明项目中解决的问题，在项目的读写分离部分是通过MyBatis的数据源的动态切换，这一模块中使用了Threadlocal来进行隔离，因此抛出由于团队人员在开发过程中忘记remove，最终导致项目上线后定期出现的oom问题，你可以聊你的解决方案以及定位问题的方法，接着面试官还有可能会考察ThreadLocal相关的问题，沿着这一条链路下来，可以思考着面试中面试官可能会问的这些问题，提前做好准备，让自己能够更有信心去准备面试。对于面试，一定需要记住提早开始面起来，不要像我一样"等待一切都准备好"再投简历开始面试，这样会错过很多的机会。面了2-3家之后就慢慢培养出感觉来，从一开始自我介绍都结结巴巴，到最后把握面试的过程，这个阶段是需要练习的，可以刚开始投递自己最不想去的公司，当成自己练习的过程。

好在自己准备的还算充分，感觉比较幸运的是在这个疫情笼罩加上互联网寒冬时期，各大互联网公司裁员的情况下，经历了几个月的反复准备让自己拿了一些的offer，最终也获得了自己比较满意的offer。面经部分，个人觉得SHEIN这家公司问的比较全面，涵盖了常见的题目。如下，仅供参考学习。

SHEIN面经分享

SHEIN是一家成立于2008年的快时尚出口跨境电商互联网公司，集商品设计、仓储供应链、互联网研发以及线上运营于一体。

一面(45min左右)

1. 自我介绍
2. 详细的聊了TCP三次握手四次挥手，以及各个环节可能出现的相关问题。
3. 有没有做过MySQL调优，MySQL的一些优化方法，还问到了MySQL选错索引的问题，整条MySQL执行会经过哪些过程。
4. HashMap和ConcurrentHashMap 1.7和1.8的变化。hash扩容为什么要扩大两倍，扩大3倍为什么不行。
5. 本地缓存GuavaCache 和 Redis的区别，为什么项目中采用了多级缓存的设计
6. 介绍常见的设计模式(这一块，我觉得结合jdk或者spring相关源码，或者自己的项目使用的设计模式聊比较好)
7. 为什么要使用SpringBoot，他能带来哪些好处。
8. 线程池你在项目中怎么使用的，线程池内部原理的流程是什么样的。
9. 阻塞队列有没有看过底层是怎么实现的
10. synchronize和ReentrantLock的区别，需要先介绍各自的底层实现。
11. 有没有什么想问他的。

二面 (1h左右)

二面问了挺久，总共一个半小时，基本围绕着简历来问，

1. 问了一些Java基础，HashMap, HashSet，重写了hashCode方法需不需要重写equal方法，如何解决哈希冲突的等等。
2. B+树，InnoDB与MyIsam的区别，还问了事务隔离级别读提交与可重复读的一些区别。
3. 接下来又问了Java并发知识点，Synchronized与ReentrantLock区别，可见性的问题，CAS，问到Unsafe是什么，原子类等等。
4. JVM问的比较多，程序计数器的作用，虚拟机栈里面的栈帧存放着什么，本地方法栈又是干什么用的，新生代与老年代，垃圾回收算法，垃圾收集器等等问题。
5. Spring问了IOC和AOP，这一块问的相对较少。
6. 问了很多基础之后才开始问项目，项目从第一个开始问，问的很细，难点在哪，怎么解决，点赞后站内信的通知异步是怎么实现的等等，问完第一个项目接着问第二个项目。
7. 问了netty如何使用的，nio相关问题，最后问到Linux的io, select, epoll这些。
8. HashMap存储了50w的数据，给出最快速的遍历方法
9. 有没有什么想问他的。

三面（25min左右）

三面问的技术问题就相对少了，主要问了跳表，Java并发的知识点，Linux的基础命令，Git的常规问题，JVM的回收算法介绍了下，还问了让我来介绍Git给不懂Git的人听，你会怎么跟他介绍。

四面（CTO面 时间很短，不到5分钟）

大概就随便和我聊了下，为什么想来南京，有没有参加秋招，本科期间代码量怎么样，我当时都还没开始聊起来，他就说大概就这些了。感觉有点虚，毕竟问的时间那么短，当时我还问了之前认识的一个老哥，他也面了CTO面，他也是5分钟左右，总体感觉CTO挺幽默的。

五面 HR面

主要介绍了公司的情况，薪酬待遇，问能不能提前去实习等等一些问题。

总体感觉shein的面试效率还是很高的，基本一天一面。HR的态度非常好，中间由于一些事情耽误，还鸽了一次技术面试，HR根据我的时间以及面试官的时间帮我额外安排了一次面试。对这家公司的映像非常好。值得一提的是感觉现在互联网上的资料太过于多，各大线上架构师等培训机构的出现，间接的促进了面试难度在逐年加大，有些问题不能不理解的单单去记忆背诵，以此来期望面试通过，这个方法肯定行不通。记得比较深刻的是有一场面试，我间接提了好几嘴自己对于HashMap, ConcurrentHashMap比较熟悉，面试官都不买账。包括后续问我对Java那一块比较熟悉除了集合部分（衰）。对于JVM的考察也不再是考察背诵垃圾回收算法以及常见的垃圾收集器，而是问为什么要按这个比例设定，如果不这样会导致什么问题等等。对于常见的排序和二叉树的时间复杂度被问到后，面试官希望你能够给他推导出来。所以，希望准备面试的小伙伴，

写在最后

还是要准备扎实的基础，不要靠直接背诵面试题这种方式来应付面试，方能以不变应万变。最后，吃水不忘挖井人，非常感谢Guide哥的帮助，Guide哥的公众号和github在我学习Java的道路上包括后续的准备面试的过程中对我的帮助都非常大。

Guide哥：这个彩虹屁很喜欢，哈哈！

Java后端实习面经，电子科大大三读者投稿！看了之后感触颇深！很感动开心！

大家好！我是Guide哥（这俗气的开头，Guide哥内心暗自BB）。

这篇文章是我的一位读者的投稿，为了方便称呼加上这位老哥的头像是哆啦A梦，我暂时称呼这位读者为哆啦A梦吧！哈哈！

那天我在朋友圈发了一个说说来恭喜一位校招成功进入网易的读者，然后哆啦A梦就评论说我的JavaGuide对他的帮助很大，他自己也成功入职了京东。每次看到这类消息，你可以脑补一下坐在屏幕前的傻笑的我，哈哈！然后，我就给哆啦A梦说，他可以分享一下自己的找工作的一些经验，结果第二天哆啦A梦就给我发了过来。看了之后，感觉写得真的很用心！下面的内容尤其对面试没有把握或者学习没有方向的人有很大帮助！



关于我

我现在是本科大三学生，在电子科大就读软件工程专业，在我大一大二的时候其实也并没有找到所谓的方向，将来想要从事什么岗位。只是一心想先学好学校的专业课程，工作就业的事以后再说。我就一直用自己在学校课程上取得的一点点成绩在麻痹自己，逃避就业的现实。其实大家也都非常清楚，现在高校里面讲授的内容很多都是偏向于底层的一些理论知识，并不会具体教你框架、怎么做项目、怎么样写代码、即使有很多实验课程也都是非常地老套和实际情况差距非常大。这就直接导致一个很大的问题：我的编程能力很差，没有一点自信。

由于我们学院特殊的安排，我们基本所有必修专业课程的学习都在大一和大二修完，大三上半学期有少量的专业选修课程和思政课。大三下整个学期都是要去企业完成6个月的实习。了解到很多优秀的学长在大三实习的时候就拿到了非常厉害的offer和优厚实习待遇，我当然是非常的心动，希望能够在大三下学期的时候能拿到一个不错的实习岗位。由于我个人是非常不愿意去做测试开发，算法开发的门槛又相对较高，然后就选择了Java这个方向。

准备面试

我其实在大二上半学期的时候修了Java这门课程，但是学校的Java课程是非常老套，和实际企业里的开发是完全脱节。在大三上半学期我当时就在网上找各种Java的学习路线，但我发现有很多学习路线看完都是“实力劝退”的感觉，因为内容太多太杂，对于一个想要入门开发的Javaer非常不友好。也是机缘巧合，在一个学长（很厉害的一个学长，目前在华科直博）推荐下，了解到JavaGuide这个开源项目，从那时起我才算是打开了新世界的大门。学习路线非常清楚，特别对于我们这种初学者的人来说非常友好，知识点的总结也在我后来面试过程帮了大忙。

看到身边的大佬们手拿多个大厂实习offer不知道怎么选时，一方面是非常羡慕，另一方面就是觉得自己是在还以前欠下的债，所以大三上整个学期我的压力都是挺大的，边学习Java的技术栈边准备面试。前前后后面试的公司有百度、成都SAP、京东（京东数科）、新浪微博等，最终也算是如愿以偿，马上准备入职京东。

至于我怎么准备的面试？我觉得很重要的一点就是根据自己写的简历和所投递岗位的JD有针对性地复习。在简历上最为重要的版块就是项目经历和技能清单这两块，这两部分直接决定了能不能拿到面试资格和面试官怎样提问。所以我当时就遇到了一种窘境，因为我是边学Java边面试，项目这部分可写的非常少，基本就没有。

我看各大公司的招聘需求：Java开发现在基本都是SSM、SpringBoot框架等等，当我学完了这部分之后，我就跟着学校老师那边做了一个Java后端的项目把学的框架练习了一遍，写在了简历上，之后我就对项目中的技术点进行复盘。

在当时我确实有着投机的心态，但是必须要有这样一个项目，否则我可能连面试的机会都没有，在参加了多次面试之后我的感受就是：作为实习生，项目这一方面重点在于面试官他要确认你是实实在在地做了，并且有你自己的思考和收获。面试的重点其实是在很多基础的问题上（面试题放在后面），在基础这部分，我反复地复习JavaGuide上面的基础知识点，在这里必须感谢JavaGuide，这可以说直接影响了我在面试中的表现。

面试真题

下面的面试题是来自百度、京东、新浪微博，我进行了一个总结，希望能帮到大家，划重点的部分表示反复被问到

数据结构与算法篇

- B树和B+树的区别
- 你了解哪些排序算法？算法的思想、时间复杂度、空间复杂度？
- LeetCode第1题及第15题：两数之和及三数之和问题

计算机网络篇

- TCP三次握手、四次挥手流程？为什么三次，为什么四次？
- TCP和UDP区别，有TCP为什么还要有UDP？
- TCP粘包和拆包问题有了解吗？
- TCP是怎样保持连接的？

操作系统篇

- 并发编程中死锁有了解吗？死锁产生的条件是什么？你在项目中是怎样解除避免和解除死锁的？
- 进程的都有哪些状态？怎么转换的？
- Linux下文件的操作命令

数据库篇

- 数据库范式了解吗？在你的项目中怎么运用的？会出现什么问题？
- 数据库索引了解吗？MySQL中索引底层是怎么实现的？
- MySQL中存储引擎InnoDB和MyISAM有什么区别？分别用于什么场景？
- 数据库事务有了了解吗？事务的隔离级别？你在项目中使用的隔离级别是什么？
- SQL优化有什么思路？
- 项目中使用到外键了吗？外键作用？使用外键要注意些什么问题？
- 除了MySQL数据库你还用到哪些数据库？Redis数据库和MySQL数据库的区别？
- 设计一个数据库表

Java基础篇

- 类和对象的区别？
- 讲讲static关键字和final关键字
- synchronized关键字是怎么用的？底层实现有了解吗？还有用过其他的锁吗？
- BIO、NIO、AIO区别有哪些？项目中有用到吗？Netty了解吗？
- 接口和抽象类的区别？什么时候用接口，什么时候用抽象类？接口可以继承接口吗？
- HashMap和HashTable的区别是什么？
- ConcurrentHashMap和HashMap的区别是什么？ConcurrentHashMap为什么线程安全？
- HashSet和HashMap的区别？HashSet是如何检查重复的？
- Java中线程的状态？join()、yield()方法是什么？
- Object类下有哪些方法？
- 字符串"123"转换成整型123的API是什么？整型123转换成字符串“123”的API又是什么？
- 创建线程有几种方式？分别是怎么做的？
- 线程池用过吗？如何创建一个线程池？其中各个参数的含义是什么？为什么要用线程池？coreSize？
- synchronized、ReentrantLock区别？
- CountDownLatch和Semaphore用过吗？他们的区别是什么？CountDownLatch应用场景？比如现在要让第5个线程等待前4个线程执行完毕再执行，具体怎么做？
- 使用synchronized来实现单缓冲区的生产者消费者模型？
- JVM有了解吗？JVM中参数-Xms和-Xmx是什么意思？
- 设计模式有了解过哪些？单例设计模式知道哪几种写法？策略设计模式了解吗？你在项目中用到了哪些设计模式？
- Spring中依赖注入有几种方式？怎么做的？
- Spring框架中有哪些组件了解吗？分别做什么的？
- SpringMVC的这种MVC模式了解吗？他的工作原理是什么？用到了哪些设计模式？（基本每轮面试都被问到）
- SpringMVC中要接受用户传来的参数要怎么做？REST的风格呢？
- Spring中bean的创建过程了解吗？
- SpringBoot和SpringMVC的区别和联系是什么？了解SpringBoot的启动流程吗？SpringBoot自动配置是如何实现的？

总结：其实我们看上面的问题，整体来说还是非常地基础，尤其对于实习生和应届生来说，基础是第一位的，就包括百度和京东的面试官都在面试最后给我强调基础的重要性

写在最后

以前觉得自己还小还早，告诉自己才大一大二，可是当突然把自己推向生活的洪流，我仿佛什么都做不了。有了这段找实习的经历，我觉得自己成长了不少，要勇敢地跳出自己的舒适圈，当自己不知道做什么的时候就去面试，让社会对你进行评价。

在这个过程中，我也眼看着很多好的机会从我身边流走，都是因为自己还不够优秀，虽然现在有幸拿到了实习机会，但我也时刻告诫自己要保持学习，沉淀自己，当有更好的机会来临时我能够抓的住。

在Java开发这条路上，我也算是刚刚入门，要学的还很多，作为JavaGuide的忠实粉丝，再次感谢JavaGuide! (Guide 哥故意加粗了一下，开心😊)

Guide哥注：生活要继续，学习也要继续。对我而言，JavaGuide 还有太多太多不足的地方，后面的日子会继续完善下去。

八 微服务/分布式

作者：Guide哥。

介绍：Github 70k Star 项目 **JavaGuide** (公众号同名) 作者。每周都会在公众号更新一些自己原创干货。公众号后台回复“1”领取Java工程师必备学习资料+面试突击pdf。

概览（看看自己能回答几题）：

1. 为什么要网关？
2. 你知道有哪些常见的网关系统？
3. 限流的算法有哪些？
4. 为什么要分布式 id ？
5. 分布式 id 生成策略有哪些？
6. 了解RPC吗？
7. 有哪些常见的 RPC 框架？
8. 如果让你自己设计 RPC 框架你会如何设计？
9. Dubbo 了解吗？
10. Dubbo 提供了哪些负载均衡策略？
11. 谈谈你对微服务领域的了解和认识！

答案地址：<https://t.zsxq.com/F6yrJiI>

这部分内容的答案更新在知识星球，欢迎加入我的知识星球。星球没有免费的原因是设立了门槛，提高进入读者的质量。我会在星球回答大家的问题，更新更多的大厂面试干货！后续会适当进行涨价。



九 真实大厂面试现场

我和阿里面试官的一次邂逅(上)

本文的内容都是根据读者投稿的真实面试经历改编而来，首次尝试这种风格的文章，花了几晚才总算写完，希望对你有帮助。

本文主要涵盖下面的内容：

1. 分布式商城系统：架构图讲解；
2. 消息队列相关：削峰和解耦；
3. Redis 相关：缓存穿透问题的解决；
4. 一些基础问题：
 - 网络相关：1. 浏览器输入 URL 发生了什么？2. TCP 和 UDP 区别？3. TCP 如何保证传输可靠性？
 - Java 基础：1. 既然有了字节流，为什么还要有字符流？2. 深拷贝 和 浅拷贝有啥区别呢？

下面是正文！

面试开始，坐在我前面的就是这次我的面试官吗？这发型看着根本不像程序员啊？我心里正嘀咕着，只听见面试官说：“小伙，下午好，我今天就是你的面试官，咱们开始面试吧！”。

自我介绍

面试官： 我也不用多说了，你先自我介绍一下吧，简历上有的就不要再说了哈。

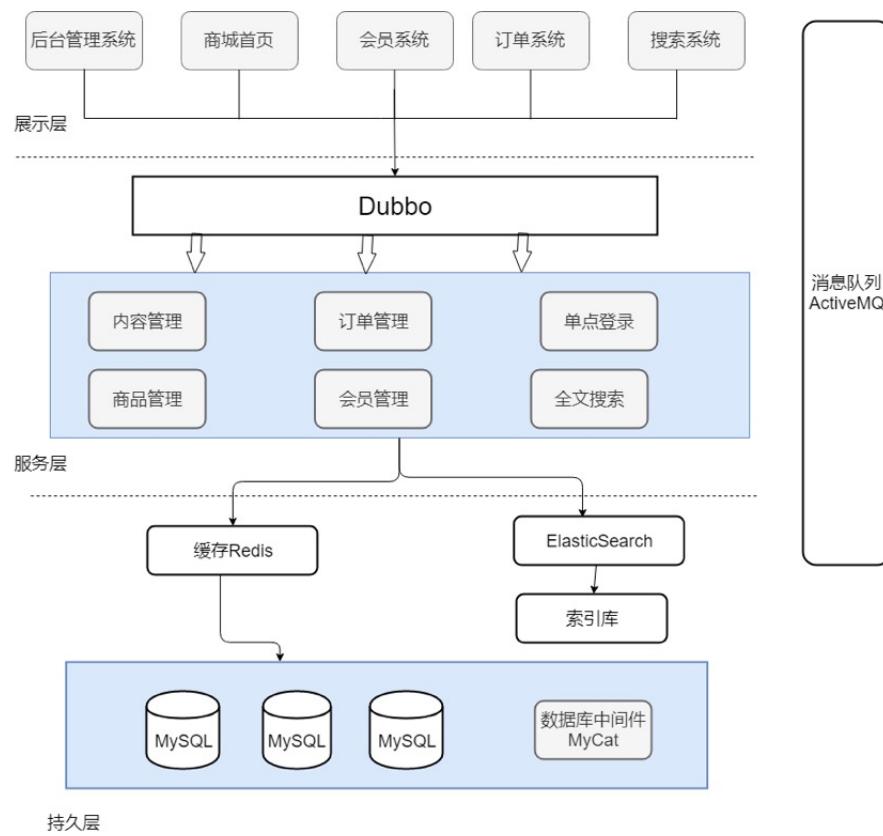
我： 内心 os：“果然如我所料，就知道会让我先自我介绍一下，还好我看了 [JavaGuide](#)，学到了一些套路。套路总结起来就是：最好准备好两份自我介绍，一份对 hr 说的，主要讲能突出自己的经历，会的编程技术一语带过；另一份对技术面试官说的，主要讲自己会的技术细节，项目经验，经历那些就一语带过。所以，我按照这个套路准备了一个还算通用的模板，毕竟我懒嘛！不想多准备一个自我介绍，整个通用的多好！”

面试官，您好！我叫小李子。大学时间我主要利用课外时间学习 Java 相关的知识。在校期间参与过一个某某系统的开发，主要负责数据库设计和后端系统开发，期间解决了什么问题，巴拉巴拉。另外，我自己在学习过程中也参照网上的教程写过一个电商系统的网站，写这个电商网站主要是为了能让自己接触到分布式系统的开发。在学习之余，我比较喜欢通过博客整理分享自己所学知识。我现在已经是某社区的认证作者，写过一系列关于 线程池使用以及源码分析的文章深受好评。另外，我获得过省级编程比赛二等奖，我将这个获奖项目开源到 Github 还收获了 2k 的 Star 呢？

项目介绍

面试官： 你刚刚说参考网上的教程做了一个电商系统？你能画个这个电商系统的架构图吗？

我： 内心 os：“这可难不倒我！早知道写在简历上的项目要重视了，提前都把这个系统的架构图画了好多遍了呢！”



做过分布式电商系统的一定很熟悉上面的架构图（目前比较流行的是微服务架构，但是如果你有分布式开发经验也是非常加分的！）。

面试官： 简单介绍一下你做的这个系统吧！

我： 我一本正经的对着我刚刚画的商城架构图开始了满嘴造火箭的讲起来：

本系统主要分为展示层、服务层和持久层这三层。表现层顾名思义主要就是为了用来展示，比如我们的后台管理系统的页面、商城首页的页面、搜索系统的页面等等，这一层都只是作为展示，并没有提供任何服务。

展示层和服务层一般是部署在不同的机器上来提高并发量和扩展性，那么展示层和服务层怎样才能交互呢？在本系统中我们使用 Dubbo 来进行服务治理。Dubbo 是一款高性能、轻量级的开源 Java RPC 框架。Dubbo 在本系统的主要作用就是提供远程 RPC 调用。在本系统中服务层的信息通过 Dubbo 注册给 ZooKeeper，表现层通过 Dubbo 去 ZooKeeper 中获取服务的相关信息。Zookeeper 的作用仅仅是存放提供服务的服务器的地址和一些服务的相关信息，实现 RPC 远程调用功能的还是 Dubbo。如果需要引用到某个服务的时候，我们只需要在配置文件中配置相关信息就可以在代码中直接使用了，就像调用本地方法一样。假如说某个服务的使用量增加时，我们只用为这单个服务增加服务器，而不需要为整个系统添加服务。

另外，本系统的数据库使用的是常用的 MySQL，并且用到了数据库中间件 MyCat。另外，本系统还用到 redis 内存数据库来作为缓存来提高系统的反应速度。假如用户第一次访问数据库中的某些数据，这个过程会比较慢，因为是从硬盘上读取的。将该用户访问的数据存在数缓存中，这样下一次再访问这些数据的时候就可以直接从缓存中获取了。操作缓存就是直接操作内存，所以速度相当快。

系统还用到了 Elasticsearch 来提供搜索功能。使用 Elasticsearch 我们可以非常方便的为我们的商城系统添加必备的搜索功能，并且使用 Elasticsearch 还能提供其它非常实用的功能，并且很容易扩展。

消息队列

面试官：我看你的系统里面还用到了消息队列，能说说为什么要用它吗？

我：

使用消息队列主要是为了：

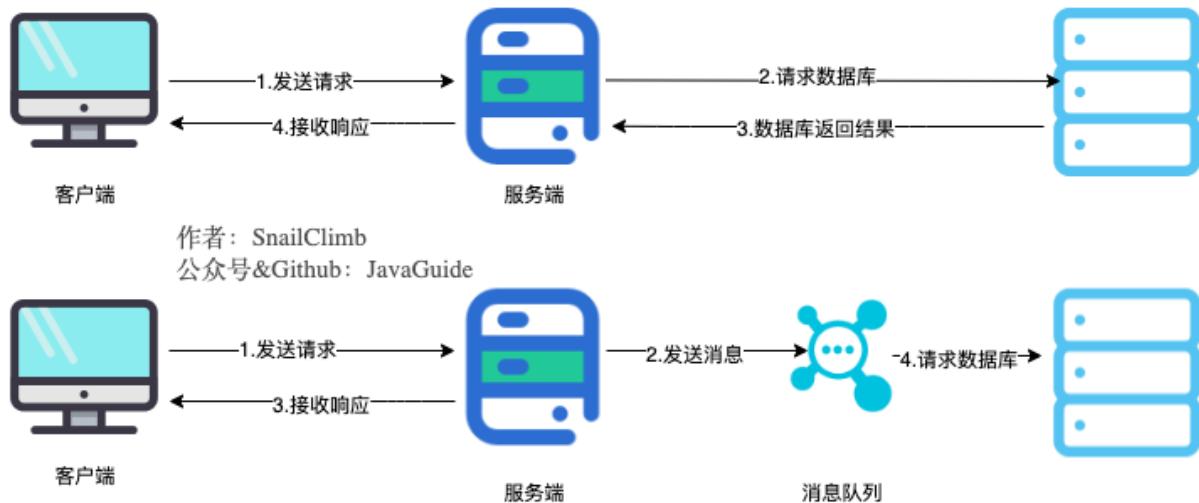
1. 减少响应所需时间和削峰。
2. 降低系统耦合性（解耦/提升系统可扩展性）。

面试官：你这说的太简单了！能不能稍微详细一点，最好能画图给我解释一下。

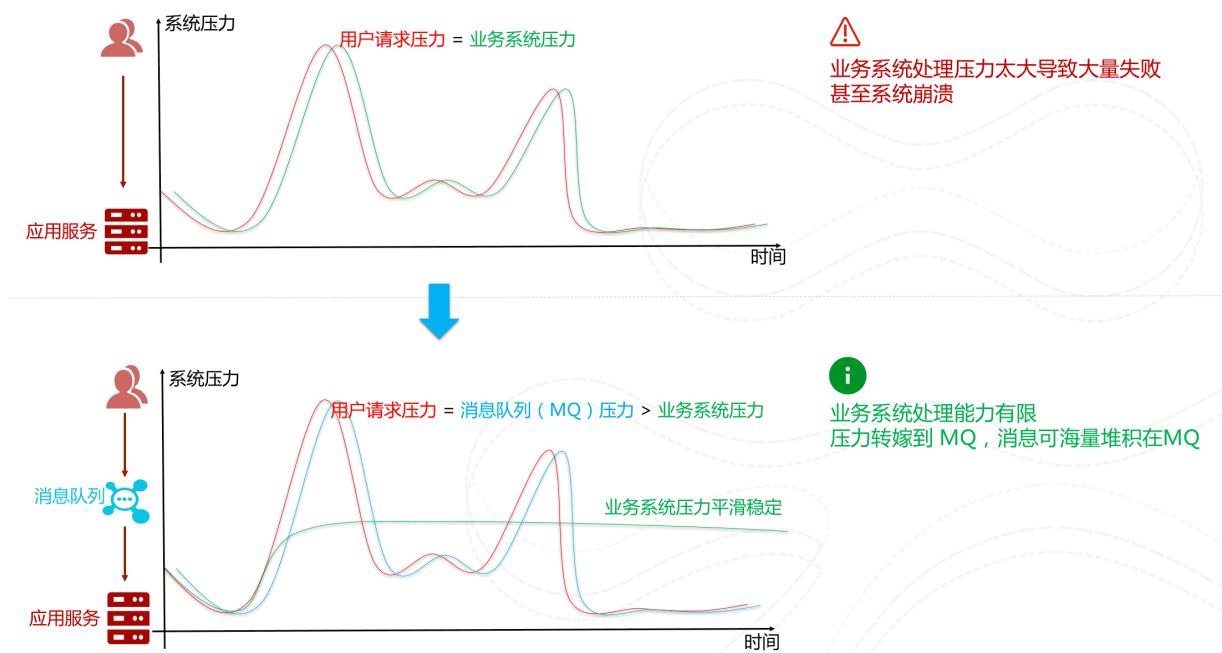
我：内心 os：“都 2019 年了，大部分面试者都能对消息队列的为系统带来的这两个好处倒背如流了，如果你想走的更远就要别别人懂的更深一点！”

当我们不使用消息队列的时候，所有的用户的请求会直接落到服务器，然后通过数据库或者缓存响应。假如在高并发的场景下，如果没有缓存或者数据库承受不了这么大的压力的话，就会造成响应速度缓慢，甚至造成数据库宕机。但是，在使用消息队列之后，用户的请求数据发送给了消息队列之后就可以立即返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库，不过要确保消息不被重复消费还要考虑到消息丢失问题。由于消息队列服务器处理速度快于数据库，因此响应速度得到大幅改善。

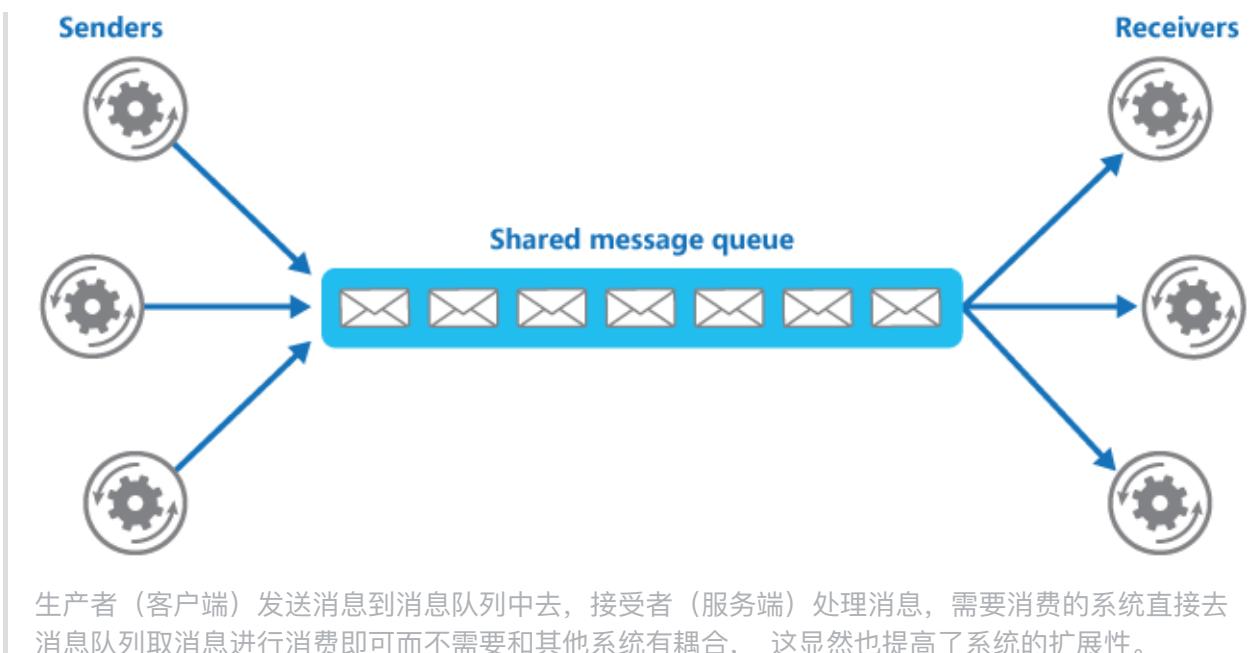
文字 is too 空洞，直接上图吧！下图展示了使用消息前后系统处理用户请求的对比（ps：我自己都被我画的这个图美到了，如果你也觉得这张图好看的话麻烦来个素质三连！）。



通过以上分析我们可以得出消息队列具有很好的削峰作用的功能—即通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。举例：在电子商务一些秒杀、促销活动中，合理使用消息队列可以有效抵御促销活动刚开始大量订单涌入对系统的冲击。如下图所示：



使用消息队列还可以降低系统耦合性。我们知道如果模块之间不存在直接调用，那么新增模块或者修改模块就对其他模块影响较小，这样系统的可扩展性无疑更好一些。还是直接上图吧：



生产者（客户端）发送消息到消息队列中去，接受者（服务端）处理消息，需要消费的系统直接去消息队列取消息进行消费即可而不需要和其他系统有耦合，这显然也提高了系统的扩展性。

面试官：你觉得它有什么缺点吗？或者说怎么考虑用不用消息队列？

我：内心 os: "面试官真鸡贼！这不是勾引我上钩么？还好我准备充分。"

我觉得可以从以下几个方面来说：

1. 系统可用性降低：系统可用性在某种程度上降低，为什么这样说呢？在加入 MQ 之前，你不用考虑消息丢失或者说 MQ 挂掉等等的情况，但是，引入 MQ 之后你就需要去考虑了！
2. 系统复杂性提高：加入 MQ 之后，你需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！
3. 一致性问题：我上面讲了消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了！

Redis

面试官：做项目的过程中遇到了什么问题吗？解决了吗？如果解决的话是如何解决的呢？

我：内心 os: "做的过程中好像也没有遇到什么问题啊！怎么办？怎么办？突然想到可以说我在使用 Redis 过程中遇到的问题，毕竟我对 Redis 还算熟悉嘛，把面试官往这个方向吸引，准没错。"

我在使用 Redis 对常用数据进行缓冲的过程中出现了缓存穿透问题。然后，我通过谷歌搜索相关的解决方案来解决的。

面试官：你还知道缓存穿透啊？不错啊！来说说什么是缓存穿透以及你最后的解决办法。

我：我先来谈谈什么是缓存穿透吧！

缓存穿透说简单点就是大量请求的 key 根本不存在于缓存中，导致请求直接到了数据库上，根本没有经过缓存这一层。举个例子：某个黑客故意制造我们缓存中不存在的 key 发起大量请求，导致大量请求落到数据库。

总结一下就是：

1. 缓存层不命中。
2. 存储层不命中，不将空结果写回缓存。
3. 返回空结果给客户端。

一般 MySQL 默认的最大连接数在 150 左右，这个可以通过 `show variables like '%max_connections%';` 命令来查看。最大连接数一个还只是一个指标，cpu，内存，磁盘，网络等物理条件都是其运行指标，这些指标都会限制其并发能力！所以，一般 3000 的并发请求就能打死大部分数据库了。

面试官： 小伙子不错啊！还准备问你：“为什么 3000 的并发能把支持最大连接数 4000 数据库压死？”想不到你自己就提前回答了！不错！

我： 别夸了！别夸了！我再来说说我知道的一些解决办法以及我最后采用的方案吧！您帮忙看看有没有问题。

最基本的就是首先做好参数校验，一些不合法的参数请求直接抛出异常信息返回给客户端。比如查询的数据库 id 不能小于 0、传入的邮箱格式不对的时候直接返回错误消息给客户端等等。

参数校验通过的情况还是会出现缓存穿透，我们还可以通过以下几个方案来解决这个问题：

1) **缓存无效 key**：如果缓存和数据库都查不到某个 key 的数据就写一个到 redis 中去并设置过期时间，具体命令如下：`SET key value EX 10086`。这种方式可以解决请求的 key 变化不频繁的情况，如何黑客恶意攻击，每次构建的不同的请求 key，会导致 redis 中缓存大量无效的 key。很明显，这种方案并不能从根本上解决此问题。如果非要用这种方式来解决穿透问题的话，尽量将无效的 key 的过期时间设置短一点比如 1 分钟。

另外，这里多说一嘴，一般情况下我们是这样设计 key 的：`表名:列名:主键名:主键值`。

2) **布隆过滤器**：布隆过滤器是一个非常神奇的数据结构，通过它我们可以非常方便地判断一个给定数据是否存在于海量数据中。我们需要的就是判断 key 是否合法，有没有感觉布隆过滤器就是我们想要找的那个“人”。

面试官： 不错不错！你还知道布隆过滤器啊！来给我谈一谈。

我： 内心 os：“如果你准备过海量数据处理的面试题，你一定对：“如何确定一个数字是否在于包含大量数字的数字集中（数字集很大，5 亿以上！）？”这个题目很了解了！解决这道题目就要用到布隆过滤器。”

布隆过滤器在针对海量数据去重或者验证数据合法性的时候非常有用。布隆过滤器的本质实际上是“位(bit)数组”，也就是说每一个存入布隆过滤器的数据都只占一位。相比于我们平时常用的 List、Map、Set 等数据结构，它占用空间更少并且效率更高，但是缺点是其返回的结果是概率性的，而不是非常准确的。

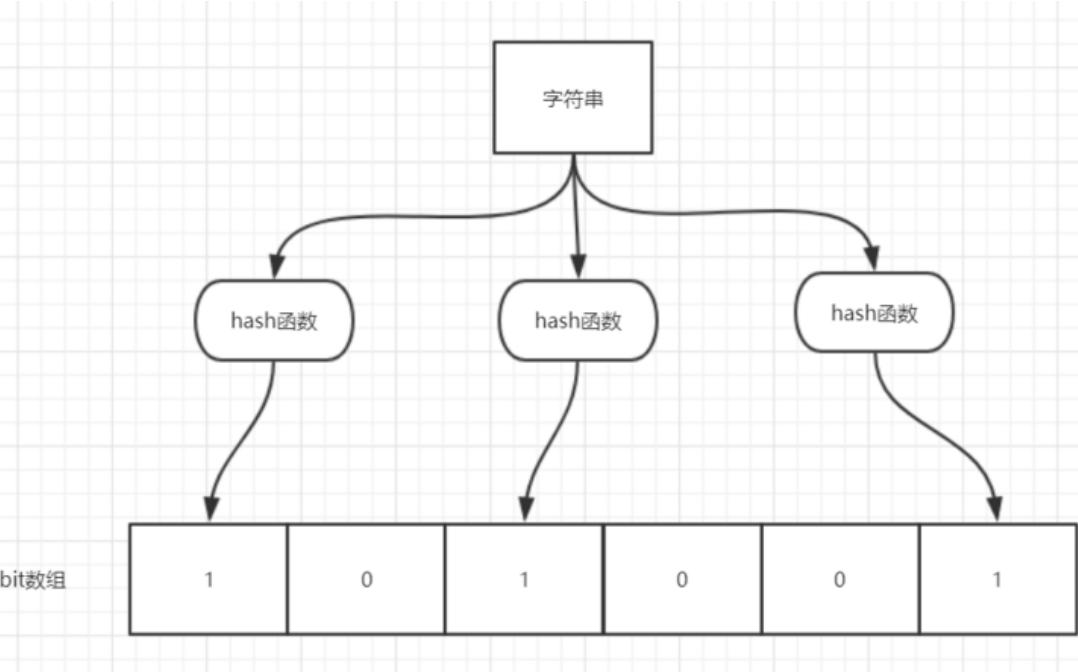
当一个元素加入布隆过滤器中的时候，会进行如下操作：

1. 使用布隆过滤器中的哈希函数对元素值进行计算，得到哈希值（有几个哈希函数得到几个哈希值）。
2. 根据得到的哈希值，在位数组中把对应下标的值置为 1。

当我们需要判断一个元素是否存在于布隆过滤器的时候，会进行如下操作：

1. 对给定元素再次进行相同的哈希计算；
2. 得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

举个简单的例子：



如图所示，当字符串存储要加入到布隆过滤器中时，该字符串首先由多个哈希函数生成不同的哈希值，然后在对应的位数组的下表的元素设置为 1（当位数组初始化时，所有位置均为 0）。当第二次存储相同字符串时，因为先前的对应位置已设置为 1，所以很容易知道此值已经存在（去重非常方便）。

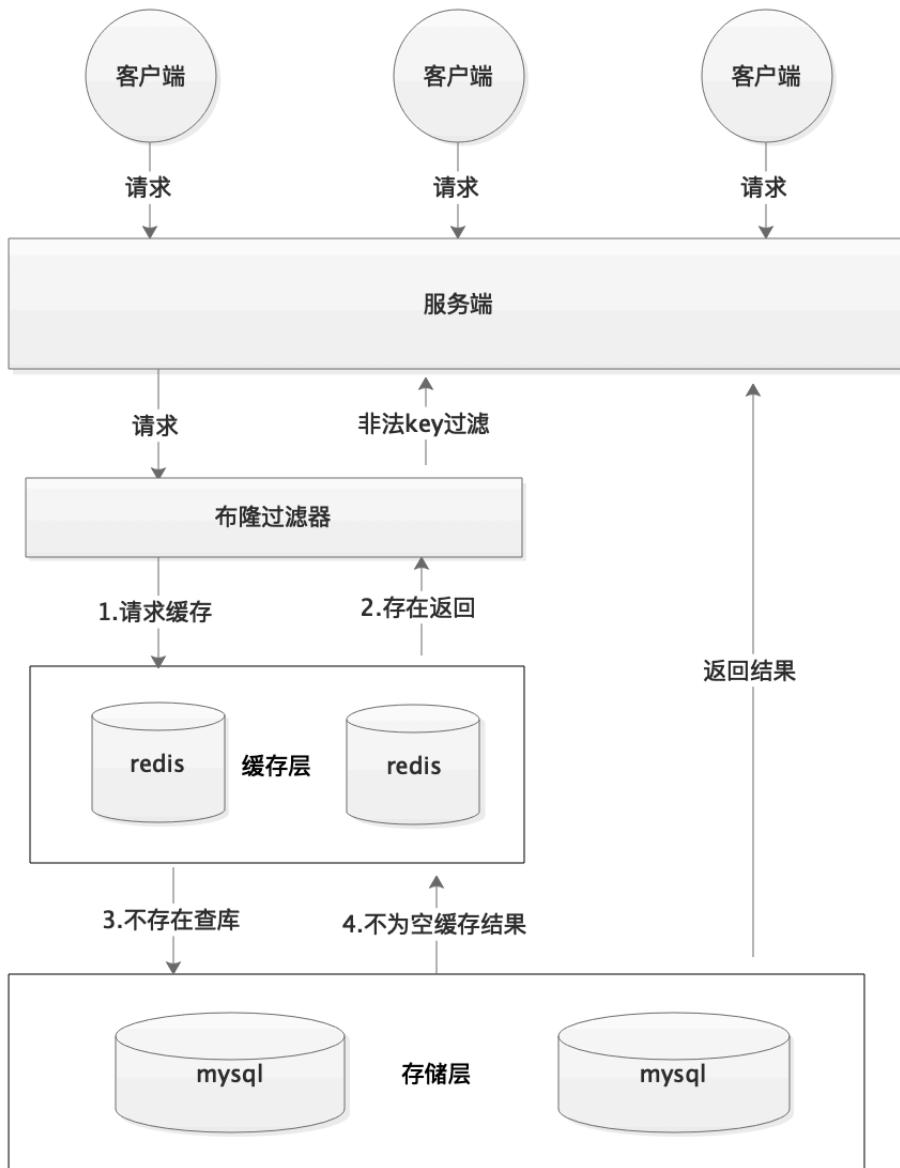
如果我们需要判断某个字符串是否在布隆过滤器中时，只需要对给定字符串再次进行相同的哈希计算，得到值之后判断位数组中的每个元素是否都为 1，如果值都为 1，那么说明这个值在布隆过滤器中，如果存在一个值不为 1，说明该元素不在布隆过滤器中。

不同的字符串可能哈希出来的位置相同，这种情况我们可以适当增加位数组大小或者调整我们的哈希函数。

综上，我们可以得出：布隆过滤器说某个元素存在，小概率会误判。布隆过滤器说某个元素不在，那么这个元素一定不在。

面试官：看来你对布隆过滤器了解的还挺不错的嘛！那你快说说你最后是怎么利用它来解决缓存穿透的。

我：知道了布隆过滤器的原理之后就很容易做了。我是利用 Redis 布隆过滤器做的。我把所有可能存在的请求的值都存放在布隆过滤器中，当用户请求过来，我会先判断用户发来的请求的值是否存在与布隆过滤器中。不存在的话，直接返回请求参数错误信息给客户端，存在的话才会走下面的流程。总结一下就是下面这张图(这张图片不是我画的，为了省事直接在网上找的)：



更多关于布隆过滤器的内容可以看我的这篇原创：[《不了解布隆过滤器？一文给你整的明明白白！》](#)，强烈推荐，个人感觉网上应该找不到总结的这么明明白白的文章了。

面试官：好了好了。项目就暂时问到这里吧！下面有一些比较基础的问题我简单地问一下你。
内心 os：难不成这家伙满口高并发，连最基础的东西都不会吧！

我：好的好的！没问题！

计算机网络

面试官：浏览器输入 URL 发生了什么？

我：内心 os：“很常问的一个问题，建议拿小本本记好了！另外，百度好像最喜欢问这个问题，去百度面试可要提前备好这道题的功课哦！相似问题：打开一个网页，整个过程会使用哪些协议？”。

图解（图片来源：《图解 HTTP》）：

过程	使用的协议
1. 浏览器查找域名的IP地址 (DNS查找过程: 浏览器缓存、路由器缓存、DNS 缓存)	DNS: 获取域名对应IP
2. 浏览器向web服务器发送一个HTTP请求 (cookies会随着请求发送给服务器)	
3. 服务器处理请求 (请求 处理请求 & 它的参数、cookies、生成一个HTML 响应)	<ul style="list-style-type: none"> • TCP: 与服务器建立TCP连接 • IP: 建立TCP协议时, 需要发送数据, 发送数据在网络层使用IP协议 • OPSF: IP数据包在路由器之间, 路由选择使用OPSF协议 • ARP: 路由器在与服务器通信时, 需要将ip地址转换为MAC地址, 需要使用ARP协议 • HTTP: 在TCP建立完成后, 使用HTTP协议
4. 服务器发回一个HTML响应	访问网页
5. 浏览器开始显示HTML	

总体来说分为以下几个过程:

1. DNS 解析
2. TCP 连接
3. 发送 HTTP 请求
4. 服务器处理请求并返回 HTTP 报文
5. 浏览器解析渲染页面
6. 连接结束

具体可以参考下面这篇文章:

- <https://segmentfault.com/a/1190000006879700>

面试官: TCP 和 UDP 区别?

我:

类型	特点			性能		应用场景	首部字节
	是否面向连接	传输可靠性	传输形式	传输效率	所需资源		
TCP	面向连接	可靠	字节流	慢	多	要求通信数据可靠 (如文件传输、邮件传输)	20-60
UDP	无连接	不可靠	数据报文段	快	少	要求通信速度高 (如域名转换)	8个字节 (由4个字段组成)

UDP 在传送数据之前不需要先建立连接，远地主机在收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但在某些情况下 UDP 确是一种最有效的工作方式（一般用于即时通信），比如：QQ 语音、QQ 视频、直播等等

TCP 提供面向连接的服务。在传送数据之前必须先建立连接，数据传送结束后要释放连接。TCP 不提供广播或多播服务。由于 TCP 要提供可靠的，面向连接的传输服务（TCP 的可靠体现在 TCP 在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源），这一难以避免增加了许多开销，如确认，流量控制，计时器以及连接管理等。这不仅使协议数据单元的首部增大很多，还要占用许多处理机资源。TCP 一般用于文件传输、发送和接收邮件、远程登录等场景。

面试官：TCP 如何保证传输可靠性？

我：

1. 应用数据被分割成 TCP 认为最适合发送的数据块。
2. TCP 给发送的每一个包进行编号，接收方对数据包进行排序，把有序数据传送给应用层。
3. 校验和：TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错，TCP 将丢弃这个报文段和不确认收到此报文段。
4. TCP 的接收端会丢弃重复的数据。
5. 流量控制：TCP 连接的每一方都有固定大小的缓冲空间，TCP 的接收端只允许发送端发送接收端缓冲区能接纳的数据。当接收方来不及处理发送方的数据，能提示发送方降低发送速率，防止包丢失。TCP 使用的流量控制协议是可变大小的滑动窗口协议。（TCP 利用滑动窗口实现流量控制）
6. 拥塞控制：当网络拥塞时，减少数据的发送。
7. ARQ 协议：也是为了实现可靠传输的，它的基本原理就是每发完一个分组就停止发送，等待对方确认。在收到确认后再发下一个分组。
8. 超时重传：当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。

面试官：我再来问你一些 Java 基础的问题吧！小伙子。

我：好的。（内心 os：“你尽管来！”）

Java基础

面试官：既然有了字节流，为什么还要有字符流？

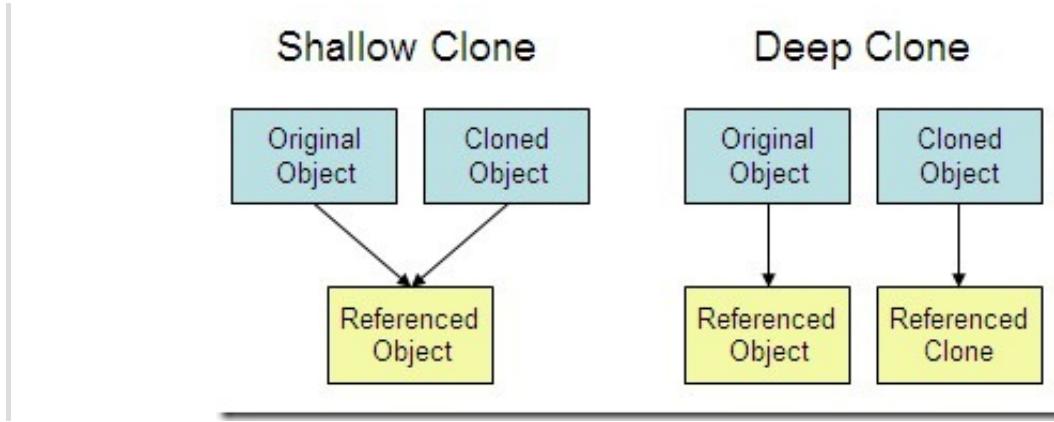
我：内心 os：“问题本质想问：不管是文件读写还是网络发送接收，信息的最小存储单元都是字节，那为什么 I/O 流操作要分为字节流操作和字符流操作呢？”

字符流是由 Java 虚拟机将字节转换得到的，问题就出在这个过程还算是非常耗时，并且，如果我们不知道编码类型就很容易出现乱码问题。所以，I/O 流就干脆提供了一个直接操作字符的接口，方便我们平时对字符进行流操作。如果音频文件、图片等媒体文件用字节流比较好，如果涉及到字符的话使用字符流比较好。

面试官：深拷贝 和 浅拷贝有啥区别呢？

我：

1. 浅拷贝：对基本数据类型进行值传递，对引用数据类型进行引用传递般的拷贝，此为浅拷贝。
2. 深拷贝：对基本数据类型进行值传递，对引用数据类型，创建一个新的对象，并复制其内容，此为深拷贝。



面试官： 好的！面试结束。小伙子可以的！回家等通知吧！

我： 好的好的！辛苦您了！

不知道这个系列大家喜欢不？喜欢的后续还会更新，不过我自己时间和能力有限，望大家理解！

2020-03-08

我和阿里面试官的一次邂逅(下)

本文主要内容如下：

操作系统：

1. 操作系统的内存管理机制了解吗？内存管理有哪几种方式？
2. 分页机制和分段机制有哪些共同点和区别呢？
3. 逻辑地址和物理地址
4. 进程和线程的区别

多线程：

1. 为什么要使用多线程？使用多线程可能带来什么问题？
2. 造成死锁的原因有哪些？如何避免线程死锁呢？
3. Java 内存模型了解吗？volatile 有什么作用？synchronized 和 volatile 的区别？
4. 用过 CountDownLatch 吗？什么场景下用的？CompletableFuture 呢？

Netty：

1. 介绍一下自己对 Netty 的认识，为什么要用
2. 通俗地说一下使用 Netty 可以做什么事情？
3. 什么是 TCP 粘包/拆包，解决办法。Dubbo 在使用 Netty 作为网络通讯时候是如何避免粘包与半包问题？
4. Netty 线程模型。
5. 讲讲 Netty 的零拷贝？

承接上一篇深受好评的文章：《【Java 大厂真实面试经历】我和阿里面试官的一次“邂逅”（附问题详解）》。时隔 n 个月，又一篇根据读者投稿的《5 面阿里，终获 offer》改编的“Java 大厂真实面试经历”文章来啦！希望这样形式的文章，你们能够喜欢，也希望你们可以从这篇文章中切实学到东西。



不同求职者的阿里面试经历因为面试官以及你的简历和能力的不同会有比较大的差异，但是在一些常见的问题上还是比较一致的。本篇文章的目的只是为了通过面试问答的形式，带着你去回顾和温习知识或者说是查漏补缺。

废话不说话！二面和三面开始了。面试官拿着一个厚重的 Thinkpad 走过来啦！他那稀疏的头发，犹豫的眼神，一看就知道是技术方面专家级别的的人物了。

操作系统

这部分的很多内容参考了《现代操作系统》第三版这本书。更多操作系统相关的面试题问题，见这篇文章：[《我和面试官之间关于操作系统的一场对弈！写了很久，希望对你有帮助！」](#)

内存管理机制主要是做什么？

面试官：操作系统的内存管理主要是做什么？

我：操作系统的内存管理主要负责内存的分配与回收（`malloc` 函数：申请内存，`free` 函数：释放内存），另外地址转换也就是将逻辑地址转换成相应的物理地址等功能也是操作系统内存管理做的事情。

操作系统的内存管理机制了解吗？内存管理有哪几种方式？

面试官：操作系统的内存管理机制了解吗？内存管理有哪几种方式？

我：这个在学习操作系统的时候有了解过。

简单分为**连续分配管理方式**和**非连续分配管理方式**这两种。连续分配管理方式是指为一个用户程序分配一个连续的内存空间，常见的如**块式管理**。同样地，非连续分配管理方式允许一个程序使用的内存分布在离散或者说不相邻的内存中，常见的如**页式管理**和**段式管理**。

1. **块式管理**：远古时代的计算机操作系统的内存管理方式。将内存分为几个固定大小的块，每个块中只包含一个进程。如果程序运行需要内存的话，操作系统就分配给它一块，如果程序运行只需要很小的空间的话，分配的这块内存很大一部分几乎被浪费了。这些在每个块中未被利用的空间，我们称之为碎片。
2. **页式管理**：把主存分为大小相等且固定的一页一页的形式，页较小，相对相比于块式管理的划分力度更大，提高了内存利用率，减少了碎片。页式管理通过页表对应逻辑地址和物理地址。
3. **段式管理**：页式管理虽然提高了内存利用率，但是页式管理其中的页实际并无任何实际意义。段式管理把主存分为一段段的，每一段的空间又要比一页的空间小很多。但是，最重要的是段是有实际意义的，每个段定义了一组逻辑信息，例如，有主程序段 MAIN、子程序段 X、数据段 D

及栈段 S 等。 段式管理通过段表对应逻辑地址和物理地址。

面试官：回答的还不错！不过漏掉了一个很重要的 **段页式管理机制**。段页式管理机制结合了段式管理和页式管理的优点。简单来说段页式管理机制就是把主存先分成若干段，每个段又分成若干页，也就是说 **段页式管理机制** 中段与段之间以及段的内部的都是离散的。

我：谢谢面试官！刚刚把这个给忘记了~



分页机制和分段机制对比

面试官：分页机制和分段机制有哪些共同点和区别呢？

我：



1. 共同点：

- 分页机制和分段机制都是为了提高内存利用率，较少内存碎片。
- 页和段都是离散存储的，所以两者都是离散分配内存的方式。但是，每个页和段中的内存是连续的。

2. 区别：

- 页的大小是固定的，由操作系统决定；而段的大小不固定，取决于我们当前运行的程序。
- 分页仅仅是为了满足操作系统内存管理的需求，而段是逻辑信息的单位，在程序中可以体现为代码段，数据段，能够更好满足用户的需要。

逻辑地址和物理地址

面试官：你刚刚还提到了**逻辑地址**和**物理地址**这两个概念，我不太清楚，你能为我解释一下不？

我：em...好的嘛！我们编程一般只可能和逻辑地址打交道，比如在 C 语言中，指针里面存储的数值就可以理解成为内存里的一个地址，这个地址也就是我们说的逻辑地址，逻辑地址由操作系统决定。物理地址指的是真实物理内存中地址，更具体一点来说就是内存地址寄存器中的地址。物理地址是

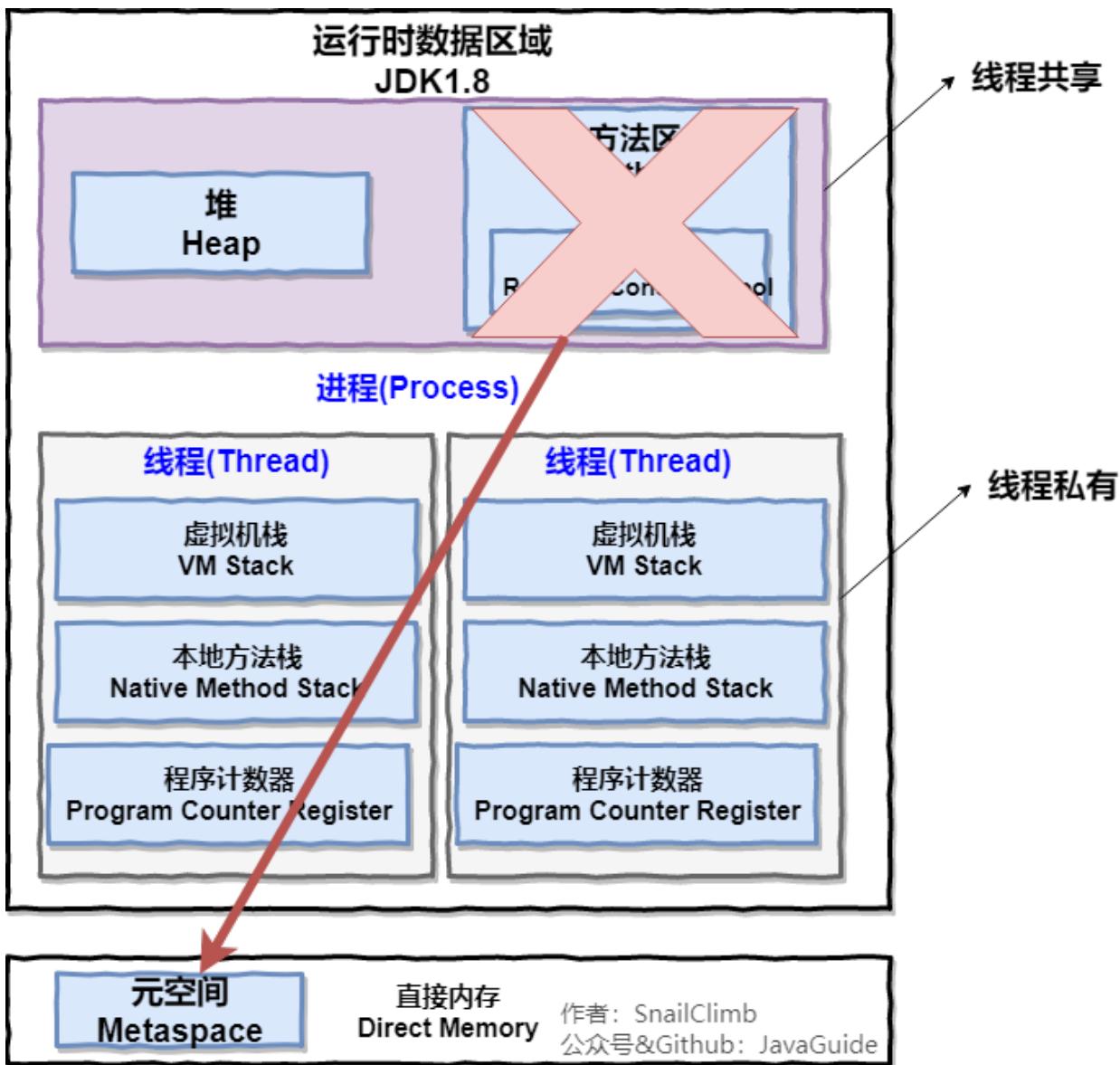
内存单元真正的地址。

进程和线程

面试官：好的！我明白了！那你再说一下： 进程和线程的区别。

我： 好的！ 下图是 Java 内存区域，我们从 JVM 的角度来说一下线程和进程之间的关系吧！

如果你对 Java 内存区域（运行时数据区）这部分知识不太了解的话可以阅读一下这篇文章：[《可能是把 Java 内存区域讲的最清楚的一篇文章》](#)



从上图可以看出：一个进程中可以有多个线程，多个线程共享进程的堆和方法区 (JDK1.8 之后的元空间) 资源，但是每个线程有自己的程序计数器、虚拟机栈 和 本地方法栈。

总结：线程是进程划分成的更小的运行单位，一个进程在其执行的过程中可以产生多个线程。线程和进程最大的不同在于基本上各进程是独立的，而各线程则不一定，因为同一进程中的线程极有可能会相互影响。线程执行开销小，但不利于资源的管理和保护；而进程正相反。

多线程

为什么要使用多线程？

 面试官：为什么要使用多线程？使用多线程可能带来什么问题？

 我：使用多线程目的就是为了能提高程序的执行效率提高程序运行速度。如果多线程使用不当，不仅不会提高程序的执行速度，可能会遇到很多问题，比如：线程不安全、内存泄漏、死锁等等。

多线程死锁

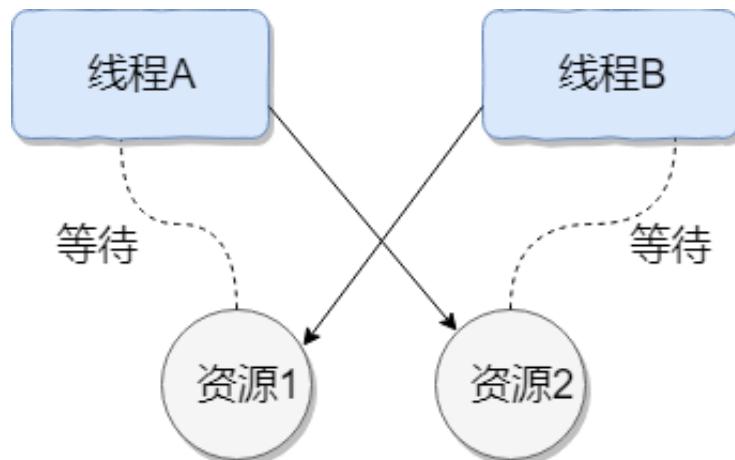
 面试官：那你说说造成线程死锁的原因有哪些吧？可以用代码给我演示一下不？

 我：我艹！有点难度啊！还好我看了《JavaGuide 面试突击版》，不然不是要 gg 了么！



线程死锁描述的是这样一种情况：多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



下面通过一个例子来说明线程死锁，代码模拟了上图的死锁的情况（代码来源于《并发编程之美》）：

```
public class DeadLockDemo {  
    private static Object resource1 = new Object(); // 资源 1  
    private static Object resource2 = new Object(); // 资源 2  
  
    public static void main(String[] args) {  
        new Thread(() -> {  
            synchronized (resource1) {  
                System.out.println("Thread A: Hold Resource 1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                synchronized (resource2) {  
                    System.out.println("Thread A: Hold Resource 1 and 2");  
                }  
            }  
        }).start();  
        new Thread(() -> {  
            synchronized (resource2) {  
                System.out.println("Thread B: Hold Resource 2");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
                synchronized (resource1) {  
                    System.out.println("Thread B: Hold Resource 1 and 2");  
                }  
            }  
        }).start();  
    }  
}
```

```

        System.out.println(Thread.currentThread() + "get
resource1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource2");
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
        }
    }
}, "线程 1").start();

new Thread(() -> {
    synchronized (resource2) {
        System.out.println(Thread.currentThread() + "get
resource2");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource1");
        synchronized (resource1) {
            System.out.println(Thread.currentThread() + "get
resource1");
        }
    }
}, "线程 2").start();
}
}

```

Output

```

Thread[线程 1,5,main]get resource1
Thread[线程 2,5,main]get resource2
Thread[线程 1,5,main]waiting get resource2
Thread[线程 2,5,main]waiting get resource1

```

线程 A 通过 `synchronized (resource1)` 获得 resource1 的监视器锁，然后通过 `Thread.sleep(1000);` 让线程 A 休眠 1s 为的是让线程 B 得到执行然后获取到 resource2 的监视器锁。线程 A 和线程 B 休眠结束了都开始企图请求获取对方的资源，然后这两个线程就会陷入互相等待的状态，这也就产生了死锁。上面的例子符合产生死锁的四个必要条件。

学过操作系统的朋友应该都知道产生死锁必须具备以下四个条件：

1. **互斥条件**：该资源任意一个时刻只由一个线程占用。
2. **请求与保持条件**：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. **不剥夺条件**：线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。
4. **循环等待条件**：若干进程之间形成一种头尾相接的循环等待资源关系。

面试官：那么问题来啦！如何避免线程线程死锁呢？如何让你上面写的代码变为不会产生死锁？

我：

我上面说了产生死锁的四个必要条件，为了避免死锁，我们只要破坏产生死锁的四个条件中的其中一个就可以了。现在我们来挨个分析一下：

1. **破坏互斥条件**：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。
2. **破坏请求与保持条件**：一次性申请所有的资源。
3. **破坏不剥夺条件**：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。
4. **破坏循环等待条件**：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

我们对线程 2 的代码修改成下面这样就不会产生死锁了。

```
new Thread(() -> {
    synchronized (resource1) {
        System.out.println(Thread.currentThread() + "get
resource1");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread() + "waiting get
resource2");
        synchronized (resource2) {
            System.out.println(Thread.currentThread() + "get
resource2");
        }
    }
}, "线程 2").start();
```

Output

```
Thread[线程 1,5,main]get resource1
Thread[线程 1,5,main]waiting get resource2
Thread[线程 1,5,main]get resource2
Thread[线程 2,5,main]get resource1
Thread[线程 2,5,main]waiting get resource2
Thread[线程 2,5,main]get resource2
```

```
Process finished with exit code 0
```

我们分析一下上面的代码为什么避免了死锁的发生？

线程 1 首先获得到 resource1 的监视器锁，这时候线程 2 就获取不到了。然后线程 1 再去获取 resource2 的监视器锁，可以获取到。然后线程 1 释放了对 resource1、resource2 的监视器锁的占用，线程 2 获取到就可以执行了。这样就破坏了破坏循环等待条件，因此避免了死锁。

从实现一个线程安全的单例模式看synchronized和volatile的使用

面试官：单例模式了解吗？你用双重检验+锁的方式实现一个吧！

我：好的好的！

双重校验锁实现对象单例（静态方法+synchronized 关键字）

```
public class Singleton {
    private static Singleton uniqueInstance;
    private Singleton() {
    }
    public static Singleton getInstance() {
        //先判断对象是否已经实例过，没有实例化过才进入加锁代码
        if (uniqueInstance == null) {
            //类对象加锁
            synchronized (Singleton.class) {
                //对象为空才去创建（懒加载）
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton(); //非原子操作。注意！！！
                }
            }
        }
        return uniqueInstance;
    }
}
```

面试官：可以简单解释一下上面的代码吗？

我：在上面的代码中，我们首先判断 `uniqueInstance` 是否为空，如果不为空直接返回。如果同时有多个线程都发现``uniqueInstance==null``为空的话，就会去创建这个对象，但是创建部分的代码块使用了 `synchronized` 关键字加锁，这样就保证了某一时刻只能有一个线程可以执行创建对象这部分代码块，也就保证了当前系统只存在一个 `Singleton` 对象。

面试官：但是，你上面写的代码在多线程下会出现问题的。你再检查一下你上面写的代码。

我：思考 🤔 许久....我还是没有发现问题呢！

面试官：我来给你说一下吧！`uniqueInstance = new Singleton()` 不是原子操作，这段代码可以简单分为下面三步执行：

1. 为 `uniqueInstance` 分配内存空间；
2. 初始化 `uniqueInstance`；
3. 将 `uniqueInstance` 指向分配的内存地址

由于但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1→3→2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 a 执行了 1 和 3，此时 线程 b 调用 `getUniqueInstance()` 后发现 `uniqueInstance` 不为空，因此返回 `uniqueInstance`，但此时 `uniqueInstance` 还未被初始化，所以就会导致空指针异常。

面试官：那你说说有没有解决办法？有没有想到多线程中哪个常用的关键字？

我：哦哦！我记起来了！使用 `volatile` 修饰变量就可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。我们只需要将上面的代码稍作修改，就可以在多线程环境下使用了！代码修改如下：

```
private volatile static Singleton uniqueInstance;
```

从CPU缓存模型聊到JMM(Java内存模型)

面试官：既然聊到了 `volatile` 关键字。那你说说自己对于 Java 内存模型（JMM）的了解吧！还有，`volatile` 除了防止 JVM 的指令重排，还有什么其他作用吗？

CPU缓存模型

我：面试官我给你讲，说到这个问题呢！我们先要从 CPU缓存模型 说起！

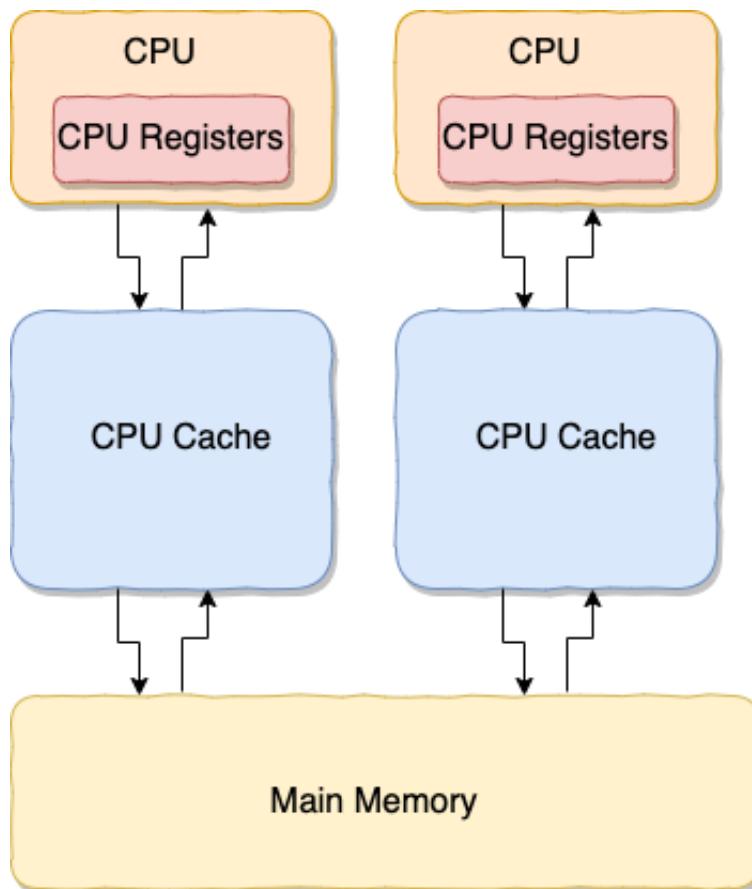
为什么要弄一个CPU高速缓存呢？

类比我们开发网站后台系统使用的缓存（比如 Redis）是为了解决程序处理速度和访问常规关系型数据库速度不对等的问题。`CPU缓存`则是为了解决CPU处理速度和内存处理速度不对等的问题。

我们甚至可以把 内存可以看作外存的高速缓存，程序运行的时候我们把外存的数据复制到内存，由于内存的处理速度远远高于外存，这样提高了处理速度。

总结：`CPU Cache` 缓存的是内存数据用于解决CPU处理速度和内存不匹配的问题，`内存缓存`的是硬盘数据用于解决硬盘访问速度过慢的问题。

为了更好地理解，我画了一个简单的CPU Cache示意图如下（实际上，现代的CPU Cache通常分为三层，分别叫L1,L2,L3 Cache）：



作者: Guide哥
公众号&Github: JavaGuide

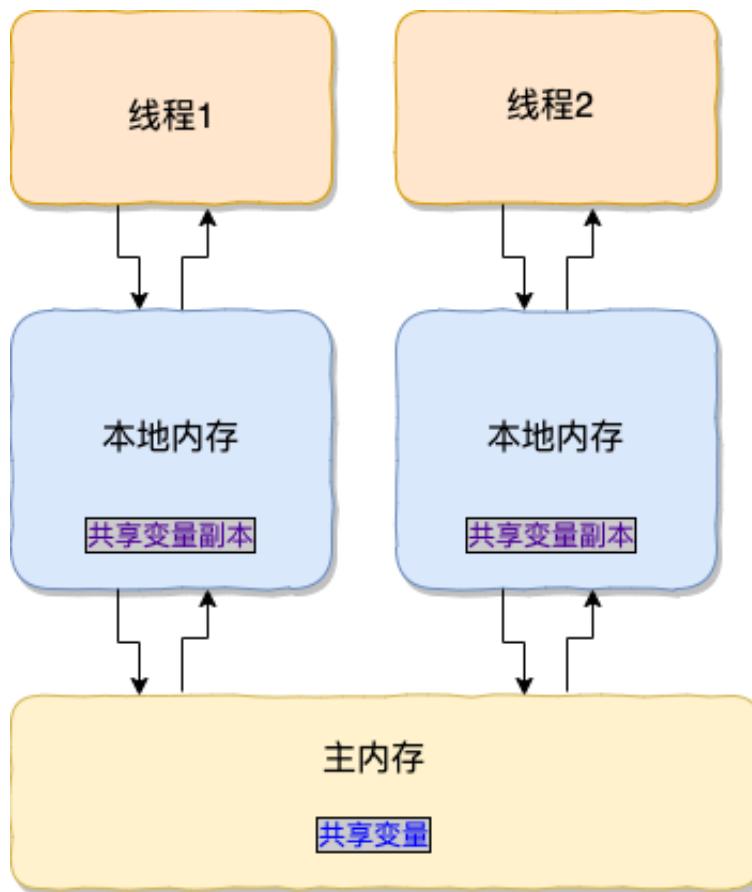
CPU Cache的工作方式:

先复制一份数据到 CPU Cache中，当CPU需要用到的时候就可以直接从CPU Cache中读取数据，当运算完成后，再将运算得到的数据写回Main Memory 中。但是，这样存在 内存缓存不一致性的问题！比如我执行一个 `i++`操作的话，如果两个线程同时执行的话，假设两个线程从CPU Cache中读取的`i=1`，两个线程做了`i++`运算完之后再写回 Main Memory 之后 `i=2`，而正确结果应该是 `i=3`。

CPU 为了解决内存缓存不一致性问题可以通过制定缓存一致协议或者其他手段来解决。

JMM(Java内存模型)

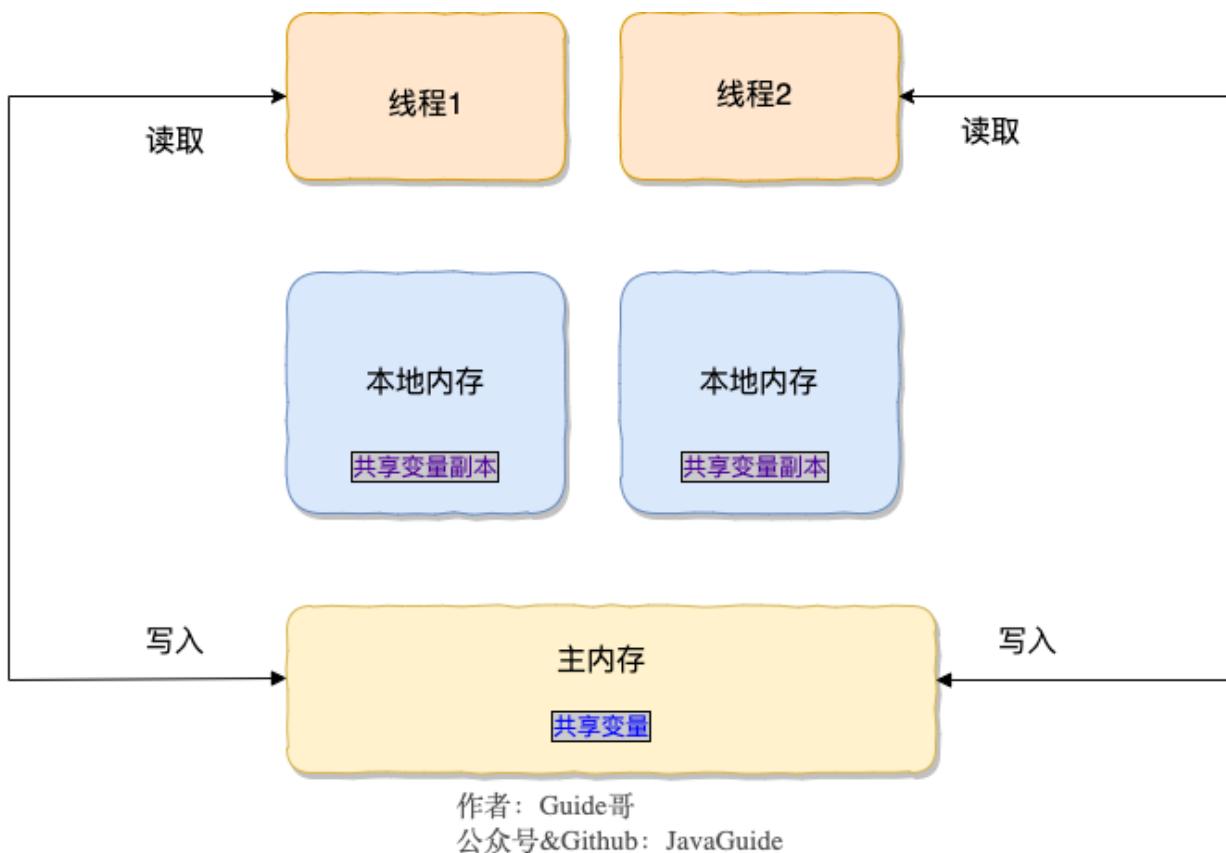
在 JDK1.2 之前，Java 的内存模型实现总是从主存（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存本地内存（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致。



作者: Guide哥
公众号&Github: JavaGuide

要解决这个问题，就需要把变量声明为 **volatile**，这就指示 JVM，这个变量是共享且不稳定的，每次使用它都到主存中进行读取。

所以，**volatile** 关键字 除了防止 JVM 的指令重排，还有一个重要的作用就是保证变量的可见性。



synchronized关键字介绍

面试官：synchronized关键字了解吗？

我：synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在Java早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的Mutex Lock来实现的，Java的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的synchronized效率低的原因。庆幸的是在Java 6之后Java官方对从JVM层面对synchronized较大优化，所以现在的synchronized锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

synchronized vs volatile

面试官：那你说说synchronized关键字和volatile关键字的区别吧！

我：synchronized关键字和volatile关键字是两个互补的存在，而不是对立的存在！

- volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。
- volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。
- volatile关键字主要用于解决变量在多个线程之间的可见性，而synchronized关键字解决的是多个线程之间访问资源的同步性。

用过CountDownLatch么？什么场景下用的？

面试官：用过CountDownLatch么？什么场景下用的？

 我 : `CountDownLatch` 的作用就是 允许 count 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。之前在项目中，有一个使用多线程读取多个文件处理的场景，我用到了 `CountDownLatch`。具体场景是下面这样的：

我们要读取处理6个文件，这6个任务都是没有执行顺序依赖的任务，但是我们需要返回给用户的时候将这几个文件的处理的结果进行统计整理。

为此我们定义了一个线程池和count为6的`CountDownLatch`对象。使用线程池处理读取任务，每一个线程处理完之后就将count-1，调用`CountDownLatch`对象的 `await()`方法，直到所有文件读取完之后，才会接着执行后面的逻辑。

伪代码是下面这样的：

```
public class CountDownLatchExample1 {  
    // 处理文件的数量  
    private static final int threadCount = 6;  
  
    public static void main(String[] args) throws InterruptedException {  
        // 创建一个具有固定线程数量的线程池对象（推荐使用构造方法创建）  
        ExecutorService threadPool = Executors.newFixedThreadPool(10);  
        final CountDownLatch countDownLatch = new  
        CountDownLatch(threadCount);  
        for (int i = 0; i < threadCount; i++) {  
            final int threadnum = i;  
            threadPool.execute(() -> {  
                try {  
                    //处理文件的业务操作  
                    .....  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                } finally {  
                    //表示一个文件已经被完成  
                    countDownLatch.countDown();  
                }  
            });  
        }  
        countDownLatch.await();  
        threadPool.shutdown();  
        System.out.println("finish");  
    }  
}
```

 面试官 : 有没有可以改进的地方呢？

 我 : 可以提示一下具体的改进方向不？

 面试官：Java 8 的新增加的一个多线程处理的类。

 我：是 `CompletableFuture` 吧！这个确实可以通过这个类来改进。Java8 的 `CompletableFuture` 提供了很多对多线程友好的方法，使用它可以很方便地为我们编写多线程程序，什么异步、串行、并行或者等待所有线程执行完任务什么的都非常方便。

```
CompletableFuture<Void> task1 =  
    CompletableFuture.supplyAsync(() -> {  
        //自定义业务操作  
    });  
.....  
CompletableFuture<Void> task6 =  
    CompletableFuture.supplyAsync(() -> {  
        //自定义业务操作  
    });  
.....  
CompletableFuture<Void>  
headerFuture = CompletableFuture.allOf(task1, ..., task6);  
  
try {  
    headerFuture.join();  
} catch (Exception ex) {  
    .....  
}  
System.out.println("all done. ");
```

 面试官：嗯嗯！大概意思说清楚了，不过代码还可以继续优化，当任务过多的时候，把每一个 task 都列出来不太现实，可以考虑通过循环来添加任务。

```
//文件夹位置  
List<String> filePaths = Arrays.asList(...)  
// 异步处理所有文件  
List<CompletableFuture<String>> fileFutures = filePaths.stream()  
    .map(filePath -> doSomething(filePath))  
    .collect(Collectors.toList());  
// 将他们合并起来  
CompletableFuture<Void> allFutures = CompletableFuture.allOf(  
    fileFutures.toArray(new CompletableFuture[fileFutures.size()]))  
);
```

Netty

Netty 介绍

 面试官：介绍一下自己对 Netty 的认识。

 我：简单用 3 点概括一下 Netty 吧！

1. Netty 是一个基于NIO 的 client-server(客户端服务器)框架，使用它可以快速简单地开发网络应用程序。
2. 它极大地简化并简化了 TCP 和 UDP 套接字服务器等网络编程，并且性能以及安全性等很多方面甚至都要更好。
3. 支持多种协议如 FTP, SMTP, HTTP 以及各种二进制和基于文本的传统协议。

用官方的总结就是：Netty 成功地找到了一种在不妥协可维护性和性能的情况下实现易于开发，性能，稳定性和灵活性的方法。

为什么要用 Netty？

 面试官：为什么要用？

 我：因为Netty 具有下面这些优点，并且相比于JDK自带的 NIO 相关 API 更加易用。

- 统一的API，支持多种传输类型，阻塞和非阻塞的。
- 简单而强大的线程模型。
- 自带编解码器解决 TCP 粘包/拆包问题。
- 自带各种协议栈。
- 真正的无连接数据包套接字支持。
- 比直接使用Java核心API有更高的吞吐量、更低的延迟、更低的资源消耗和更少的内存复制。
- 安全性不错，有完整的 SSL/TLS 以及 StartTLS 支持。
- 社区活跃
- 成熟稳定，经历了大型项目的使用和考验，而且很多开源项目都使用到了 Netty 比如我们经常接触的Dubbo、RocketMQ等等。
-

Netty 应用场景

 面试官：通俗地说一下使用 Netty 可以做什么事情？

 我：凭借自己的了解，简单说一下吧！理论上 NIO 可以做的事情，使用 Netty 都可以做并且更好。Netty 主要用来做网络通信：

1. **作为 RPC 框架的网络通信工具**：我们在分布式系统中，不同服务节点之间经常需要相互调用，这个时候就需要 RPC 框架了。不同服务指点的通信是如何做的呢？可以使用 Netty 来做。比如我调用另外一个节点的方法的话，至少是要让对方知道我调用的是哪个类中的哪个方法以及相关参数吧！
2. **实现一个自己的 HTTP 服务器**：通过 Netty 我们可以自己实现一个简单的 HTTP 服务器，这个大家应该不陌生。说到 HTTP 服务器的话，作为Java后端开发，我们一般使用 Tomcat 比较多。一个最基本的HTTP 服务器可以处理常见的 HTTP Method 的请求，比如 POST请求、GET 请求等等。
3. **实现一个即时通讯系统**：使用 Netty 我们可以实现一个可以聊天类似微信的即时通讯系统，这方面的开源项目还蛮多的，可以自行去 Github 找一找。
4. **实现消息推送系统**：市面上有很多消息推送系统都是基于 Netty 来做的。
5.

TCP 粘包/拆包以及解决办法

 面试官：什么是 TCP 粘包/拆包，解决办法？

 我 : TCP 粘包/拆包 就是你基于 TCP 发送数据的时候，出现了多个字符串“粘”在了一起或者一个字符串被“拆”开的问题。比如你多次发送：“你好，你真帅啊！哥哥！”，但是客户端接收到的可能是下面这样的：

```
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
message from client:你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥! 你好,你真帅啊! 哥哥!
```

解决办法：

1. Netty 自带的解码器
2. 自定义序列化编解码器

这篇文章中不详细分析TCP 粘包问题，后面会在我的《Netty实战+手写一个简单的RPC框架》中介绍到。

Netty线程模型

 面试官 : Netty线程模型

 我 : 大部分网络框架都是基于Reactor 模式设计开发的。

Reactor 模式基于事件驱动，采用多路复用将事件分发给相应的Handler处理，非常适合处理海量 IO 的场景。

在 Netty 主要靠 `NioEventLoopGroup` 线程池来实现具体的线程模型的。

我们实现服务端的时候，一般会初始化两个线程组：

1. `bossGroup` :接收连接。
2. `workerGroup` : 负责具体的处理，交由对应的Handler处理。

下面我们来详细看一下 Netty 中的线程模型吧！

1.单线程模型：

一个线程需要执行处理所有的accept、read、decode、process、encode、send事件。对于高负载、高并发，并且对性能要求比较高的场景不适用。

对应到 Netty 代码是下面这样的

使用 `NioEventLoopGroup` 类的无参构造函数设置线程数量的默认值就是 CPU 核心数 *2。

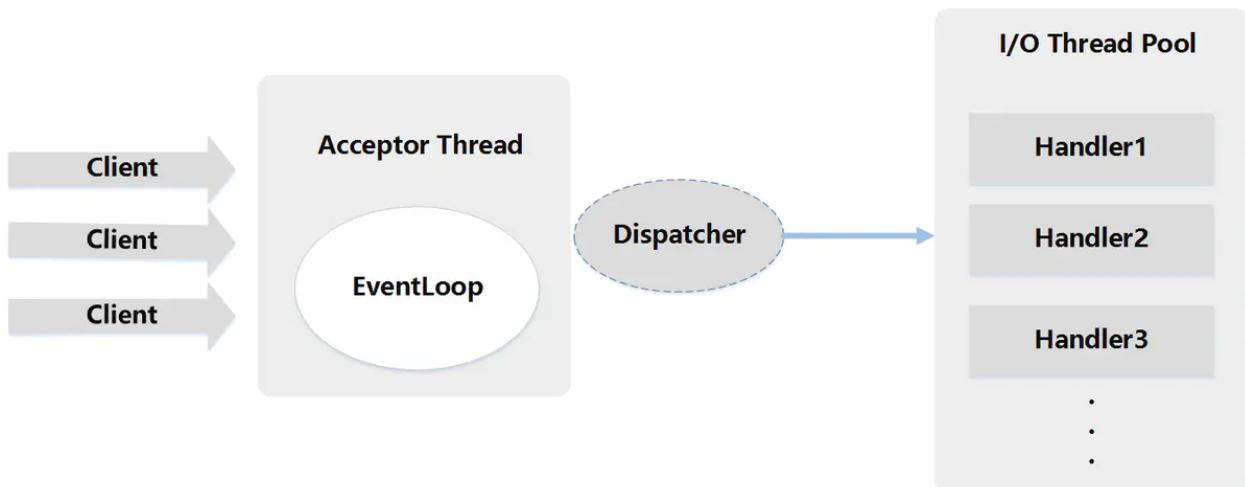
```
//1.eventGroup既用于处理客户端连接，又负责具体的处理。
EventLoopGroup eventGroup = new NioEventLoopGroup(1);
//2.创建服务端启动引导/辅助类：ServerBootstrap
ServerBootstrap b = new ServerBootstrap();
        bootstrap.group(eventGroup, eventGroup)
//.....
```

2.多线程模型

一个Acceptor线程只负责监听客户端的连接，一个NIO线程池负责具体处理：accept、read、decode、process、encode、send。满足绝大部分应用场景，并发连接量不大的时候没啥问题，但是遇到并发连接大的时候就可能会出现问题，成为性能瓶颈。

对应到Netty代码是下面这样的：

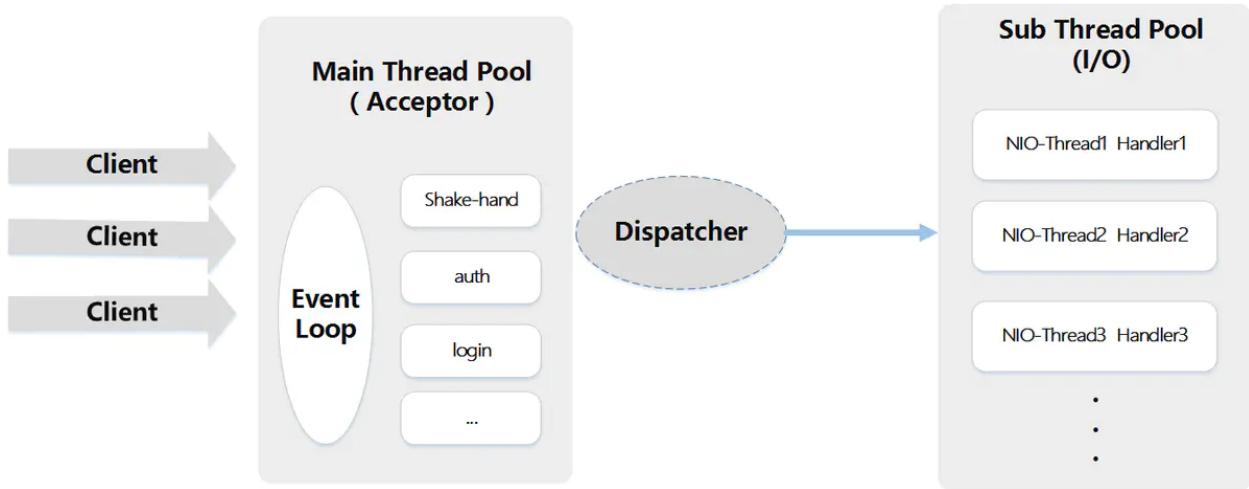
```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....
```



3. 主从多线程模型

从一个主线程NIO线程池中选择一个线程作为Acceptor线程，绑定监听端口，接收客户端连接的连接，其他线程负责后续的接入认证等工作。连接建立完成后，Sub NIO线程池负责具体处理I/O读写。如果多线程模型无法满足你的需求的时候，可以考虑使用主从多线程模型。

```
// 1.bossGroup 用于接收连接, workerGroup 用于具体的处理
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
try {
    //2.创建服务端启动引导/辅助类: ServerBootstrap
    ServerBootstrap b = new ServerBootstrap();
    //3.给引导类配置两大线程组,确定了线程模型
    b.group(bossGroup, workerGroup)
    //.....
```



Netty 的零拷贝

面试官：讲讲 Netty 的零拷贝？

我：

维基百科是这样介绍零拷贝的：

零复制（英语：Zero-copy；也译零拷贝）技术是指计算机执行操作时，CPU不需要先将数据从某处内存复制到另一个特定区域。这种技术通常用于通过网络传输文件时节省CPU周期和内存带宽。

在 OS 层面上的 `Zero-copy` 通常指避免在 `用户态(User-space)` 与 `内核态(Kernel-space)` 之间来回拷贝数据。而在 Netty 层面，零拷贝主要体现在对于数据操作的优化。

Netty 中的零拷贝体现在以下几个方面

下面的内容参考了这篇文章：<https://www.cnblogs.com/xys1228/p/6088805.html>

1. 使用 Netty 提供的 `CompositeByteBuf` 类，可以将多个 `ByteBuf` 合并为一个逻辑上的 `ByteBuf`，避免了各个 `ByteBuf` 之间的拷贝。
2. `ByteBuf` 支持 `slice` 操作，因此可以将 `ByteBuf` 分解为多个共享同一个存储区域的 `ByteBuf`，避免了内存的拷贝。
3. 通过 `FileRegion` 包装的 `FileChannel.transferTo` 实现文件传输，可以直接将文件缓冲区的数据发送到目标 `Channel`，避免了传统通过循环 `write` 方式导致的内存拷贝问题。

Reference

- 《计算机操作系统-汤小丹》第四版
- netty学习系列二：NIO Reactor模型 & Netty线程模型：<https://www.jianshu.com/p/38b56531565d>
- 《Netty实战》

十 开源项目推荐

Java教程类开源项目推荐

给大家推荐 15 个新手也能看懂的 Java 教程方向的开源项目。这些项目无论是对于你学习 Java 还是准备 Java 方向的面试都非常有帮助。

正如我第一个要推荐的开源项目 JavaGuide 说的那样：开源项目在于大家的参与，这才使得它的价值得到提升。

JavaGuide

- **简介**：一份涵盖大部分 Java 程序员所需要掌握的核心知识。
- **推荐理由**：JavaGuide 是我在大三的时候开源的一个项目。这个项目主要是记录一些我觉得比较重要的 Java 核心知识和一些常见的面试题。我觉得这个项目可以对大部分 Java 程序员都有帮助，不论是我们面试还是学习 Java 的话，都应该有很大的帮助。

CS-Notes

- **简介**：技术面试必备基础知识、Leetcode 题解、后端面试、Java 面试、春招、秋招、操作系统、计算机网络、系统设计。
- **推荐理由**：CS-Notes 是我要推荐的第二个项目。这个项目主要记录了技术面试必备的基础知识比如计算机网络、数据结构和算法，还有操作系统。除此之外，这个项目的 Leetcode 题解部分也十分给力。如果大家需要准备面试或者复习基础知识的话，一定不要错过这个项目。

advanced-java

- **简介**：互联网 Java 工程师进阶知识完全扫盲：涵盖高并发、分布式、高可用、微服务、海量数据处理等领域知识。
- **推荐理由**：这个项目大部分内容是由《Java 面试突击第一季》整理而来，视频地址我会放在评论区。如果你想要了解消息队列、分布式缓存、分布式搜索引擎、Dubbo 这些东西的话，很好，这个项目十分适合你！即使你的 Java 基础不太好，相信你也可以从中有很大的收获。

miaosha

- **简介**：秒杀系统设计与实现. 互联网工程师进阶与分析。
- **推荐理由**：大家听这个名字就应该知道这个项目和秒杀系统设计有关。是的，这个项目主要就是教你如何进行秒杀架构设计。秒杀架构设计师面试的时候面试官经常问到的东西。这个项目主要包括的技术有：jmeter 压测、消息队列 rabbitmq、分布式缓存 redis、mysql 主从复制、rpc 框架 dubbo 以及 nginx。

architect-awesome

1. **简介**：后端架构师技术图谱。
2. **推荐理由**：推荐的理由主要是它对于后端知识体系的涵盖特别全，如果大家在学习方向上有疑问的话不妨去看看这个项目。

toBeTopJavaer

1. **简介**：Java 工程师成神之路
2. **推荐理由**：阿里巴巴的一位老哥维护，这位老哥也是我一直以来的目标，非常优秀。这个项目几乎涵盖了 Java 工程师必备的所有知识，作者已经更新了部分内容，目前仍在继续更新中。

technology-talk

汇总java生态圈常用技术框架、开源中间件，系统架构、数据库、大公司架构案例、常用三方类库、项目管理、线上问题排查、个人成长、思考等知识

JavaFamily

1. 简介：【互联网一线大厂面试+学习指南】进阶知识完全扫盲。
2. 推荐理由：开源这个项目的老哥和我年龄一样大，非常优秀，同样是我学习的榜样！这个项目中有Java大厂面试指南方面的内容，作者目前已经把分布式缓存篇的所有文章的更新完了，需要这方面知识的老哥不妨去看看。

JCSprout

1. 简介：处于萌芽阶段的 Java 核心知识库。
2. 推荐理由：这个项目的作者算的是原创技术领域写得比较出名的一个了，我本人也是觉得他的文章对于大部分都很有帮助。这个项目就收录了他记录的一些 Java 核心知识比如 [如何优雅的使用和理解线程池、设计一个百万级的消息推送系统等等](#)。

fullstack-tutorial

1. 简介：后台技术栈/架构师之路/全栈开发社区，春招/秋招/校招/面试。
2. 推荐理由：包括的知识面比较广，除了 数据结构和算法这些基础知识，还包括 Java 后端、一点前端、一点 Python 内容。可以当作参考来看，内容比较杂。

附加5个不错的开源项目

1. [3y](#)：从Java基础、JavaWeb基础到常用的框架再到面试题都有完整的教程，几乎涵盖了Java后端必备的知识点
2. [JGrowing](#)：Java 成长路线，但学到不仅仅是 Java
3. [interview_internal_reference](#)：2019年最新总结，阿里，腾讯，百度，美团，头条等技术面试题目，以及答案，专家出题人分析汇总。
4. [effective-java-3rd-chinese:Effective Java中文版（第3版）](#)：Java 四大名著之一，本书一共包含90个条目，每个条目讨论Java程序设计中的一条规则。这些规则反映了最有经验的优秀程序员在实践中常用的一些有益的做法。
 1. [《OnJava8》](#)：又名《Java编程思想》第5版， Java 四大名著之一。

Leetcode题解

一个很明显的现象，现在大厂的应届生面试，甚至是社招面试都开始越来越重视算法了。为了能够应对，我们大部分人能做的就是刷 Leetcode 来积累做算法题的经验和套路。为了能够帮助我们更好的刷 Leetcode，Guide 精选了一些不错的基于 Java 题解的开源项目。

下面的项目是根据下面三个标准选出：

1. 项目是否还在继续维护更新。
2. 项目的质量如何，这一点可以从 star、issue 以及 pr 的数量侧面反映出来。
3. 是否是基于 Java 语言。

1.CS-Notes

这个开源项目不是单一关注算法的仓库，它是一个大的集合，包括了技术面试必备基础知识、Leetcode、计算机操作系统、计算机网络、系统设计等知识。

我和这个开源项目的原作者有过交流，是一名很优秀的 coder。



这个开源项目的算法部分包括4部分：

1. 剑指 Offer 题解：题目来自《何海涛. 剑指 Offer[M]. 电子工业出版社，2012.》
2. Leetcode 题解：从 Leetcode 中精选大概 200 左右的题目，去除了某些繁杂但是没有多少算法思想的题目，同时保留了面试中经常被问到的经典题目。
3. 算法：主要是一些基本的排序算法比如堆排序以及常见数据结构比如队列、栈的实现。
4. 笔试面试题库：跳转到牛客网的公司笔试面试真题。

2. LeetCodeAnimation

如果你想边看动画边学算法的话，LeetCodeAnimation 很适合你。因为，这个项目的目标是用动画的形式呈现解LeetCode题目的思路，目前这个浩大的工程只完成部分LeetCode题目。

仓库的更新的大部分算法题都是通过 Java 语言解答的，少部分是使用 C/C++解答。

0	十大经典排序算法动画与解析，看我就够了！（配代码完全版）	
1	两数之和	<p style="text-align: center;">1. Two Sum</p> <p>target = 9</p> <p>key record value</p> <p>看了个屁</p>
2	两数相加	<p style="text-align: center;">2. Add Two Numbers</p> <p>carried dummy Head Head</p> <p>@五分钟学算法</p>
3	无重复字符的最长子串	<p style="text-align: center;">Longest Substring Without Repeating Characters</p> <p>res = 3</p> <p>看了个屁</p>

3.leetcode

多种编程语言实现 LeetCode、《剑指 Offer (第 2 版)》、《程序员面试金典 (第 6 版)》题解。



4. LeetCode-Solution-in-Good-Style

这个项目是作者在学习《算法与数据结构》的时候，在 [LeetCode（力扣）](#) 上做的练习，刷题以 Java 语言为主。

作者在刷题的时候，非常考虑代码质量，他的很多问题的回答都被 Leetcode 官方精选，值得推荐！

日期	题目	难度	题解
4月1日	1111. 有效括号的嵌套深度	中等	嵌套深度 = 完成括号匹配问题实际使用的栈的最大高度
4月2日	289. 生命游戏	中等	「力扣」第289题：生命游戏（数组）
4月3日	8. 字符串转换整数 (atoi)	中等	尽量不使用库函数、一次遍历（Java）
4月4日	42. 接雨水	困难	暴力解法、优化、双指针、单调栈
4月5日	460. LFU缓存	困难	哈希表 + 双向链表（Java）
(同类问题)	146. LRU缓存机制	中等	哈希表 + 双向链表（Java）
4月6日	72. 编辑距离	困难	动态规划（Java）
4月7日	面试题 01.07. 旋转矩阵	中等	CSDN
4月8日	机器人的运动范围	中等	深度优先遍历、广度优先遍历
4月9日	22. 括号生成	中等	回溯算法（深度优先遍历）+ 广度优先遍历 + 动态规划
4月10日	151. 翻转字符串里的单词	中等	CSDN
4月11日	887. 鸡蛋掉落	困难	动态规划（只解释官方题解方法一）（Java）
4月12日	面试题 16.03. 交点	困难	使用直线斜截式方程，再判断交点是否有效（Java）
4月13日	355. 设计推特	中等	哈希表 + 链表 + 优先队列（经典多路归并问题）（Java）
4月14日	445. 两数相加 II	中等	
4月15日	542. 01矩阵	中等	
4月16日	56. 合并区间	中等	贪心算法（Java）
4月17日	55. 跳跃游戏	中等	
4月18日	11. 盛最多水的容器		
4月19日	466. 统计重复个数	困难	暴力解法 + 优化解法（Java代码）
4月20日	200. 岛屿数量	中等	DFS + BFS + 并查集（Python代码、Java代码）