# 简历-附加：

项目名称：**ABS银盒项目**

项目概述：项目均基于ssm进行开发，使用Spring Eureka实现服务注册和发现；通过feign+hytrix进行包装服务的调用，使用了Ribong进行负债均衡；使用ras+aes完成数据加密传输；通过Sping Config +Git进行了无宕机配置更新；使用MongoDB+log4j管理日志；通过ribbitmq实现延迟任务。重构了老系统的中心项目。项目集成同步GC系统信息、用友平台信息、监管报送信息的推送。

负责功能：

1. 负责客户数据对接服务、账户模块。
2. 项目的投决审批、估值尽调、未结业务管理、采购报表；
3. 科目设置、财务核算、凭证管理、日终清算等

# 讲一讲项目的亮点，项目单机和分布式：

1：单机的考虑是用了什么什么组件 中间件，提升了多少了多少qps

完全没有考虑分布式场景，分布式场景 多机一定比单机慢。那么我们为什么还有分布式多机处理。

# 为什么离职：

为了自身职业的发展吧？已经不适合在这个公司了。

1：公司的流动性有点大，毕竟外包公司，对人员的照顾也没有本部的那么好

2：项目在一期上线的时候，干了五个多月，最后二期就要调往别的部门，流动性不喜欢

1：在该公司干了时间有三年了，感觉要换个环境看看自己人生的路

2：该公司因为发版过于频繁。

3：该公司因阿里内部成立中台小组，数峰需要进行解散操作。

离职的根本原因？

钱少事多离家远，位低权轻责任重

为什么离职？

基于个人原因、市场环境和职业规划等因素

去年看中了博彦的项目管理管理，分割清清除，可以尝试不同的项目环境，接手不同的项目。然后所学有所有所用。

但是这样的也有不好地方，可能时间久了没有归属感。

想找一个稳定的甲方

# 参考网站：

## 需要学习：

### 底层的协议 跟rpc htpp

### 用了shrdingjdbc 那么有事务了

## Java面试资料清单：

<https://blog.csdn.net/zj15527620802/article/details/83384839>

## Java面试宝典：

<https://blog.csdn.net/chen_2890/article/details/83928763>

## 廖师兄-Spring-Cloud实战讲解

https://cloud.tencent.com/developer/article/1336092

<https://cloud.tencent.com/developer/article/1336095>

## java面试总结

<https://blog.csdn.net/chen_2890/article/details/84312126>

## 常被问道的十个Java面试题

<http://www.importnew.com/30956.html>

没有啥东西

## ★★★经典Java面试题收集(一) - 简书

https://www.jianshu.com/p/c01eb6e46226

# Spring Cloud总结：

eureka 微服务的注册和发现 ribbon 微服务负载均衡客户端 hystrix服务降级容错 feign简化两者及集成 zuul服务级切面拦截 config集中式配置服务 bus消息中心（提供消息中间件集成） stream这个用于消息中间件的解耦，目前支持kafka,rabbitmq sleuth 微服务跟踪

## 服务注册和发现：

<https://blog.csdn.net/forezp/article/details/73017664>

@EnableEurekaServer 服务端

@EanblDiscoveryClient 客户端

### 原理核心DiscoveryClient：

注册服务实例；服务租约；取消租约；查询实例列表

在DiscoveryClient类有一个服务注册的方法register()，该方法是通过Http请求向Eureka Client注册。，它被InstanceInfoReplicator 类的run()方法调用，其中InstanceInfoReplicator实现了Runnable,接口其中有一个initScheduledTasks()方法。该方法主要开启了获取服务注册列表的信息，如果需要向Eureka Server注册，则开启注册，同时开启了定时向Eureka Server服务续约的定时任务

### 存储：

Eureka Server 的数据存储分了两层：数据存储层和缓存层。数据存储层记录注册到 Eureka Server 上的服务信息，缓存层是经过包装后的数据，可以直接在 Eureka Client 调用时返回。我们先来看看数据存储层的数据结构。

Eureka Server 的数据存储层是双层的 ConcurrentHashMap，

第一层的 key=spring.application.name，也就是客户端实例注册的应用名；value 为嵌套的 ConcurrentHashMap。

第二层的 key=instanceId，也就是服务的唯一实例 ID，value 为 Lease 对象，Lease 对象存储着这个实例的所有注册信息，包括 ip 、端口、属性等。

第一层缓存：readOnlyCacheMap，本质上是 ConcurrentHashMap，依赖定时从 readWriteCacheMap 同步数据，默认时间为 30 秒。

readOnlyCacheMap ： 是一个 CurrentHashMap 只读缓存，这个主要是为了供客户端获取注册信息时使用，其缓存更新，依赖于定时器的更新，通过和 readWriteCacheMap 的值做对比，如果数据不一致，则以 readWriteCacheMap 的数据为准。

第二层缓存：readWriteCacheMap，本质上是 Guava 缓存。

readWriteCacheMap：readWriteCacheMap 的数据主要同步于存储层。当获取缓存时判断缓存中是否没有数据，如果不存在此数据，则通过 CacheLoader 的 load 方法去加载，加载成功之后将数据放入缓存，同时返回数据。

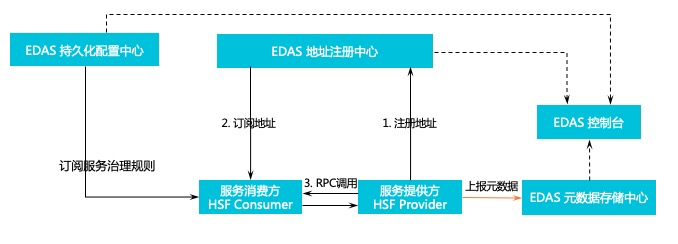
### 七种服务注册/发现

#### consul\_etcd3\_sofa\_redis\_zk\_nacos\_eruka\_EDAS

#### EDAS-HSF:

高速服务框架HSF （High-speed Service Framework），是在阿里巴巴广泛使用的分布式RPC服务框架。

EDAS



#### dobdo:zk

#### redis实现

流程如下: 1. 校验地址是否合法

2. 获取 Redis 的实例，然后将地址注册到当前 Redis 上面。

3. 发送注册成功的通知

|  |
| --- |
| @Override  public void register(InetSocketAddress address) {  // 校验地址是否合法  NetUtil.validAddress(address);  String serverAddr = NetUtil.toStringAddress(address);  // 获取 Redis 的实例  Jedis jedis = jedisPool.getResource();  try {  // 将地址注册到当前 Redis 上面。  jedis.hset(getRedisRegistryKey(), serverAddr, ManagementFactory.getRuntimeMXBean().getName());  // 发送注册成功的通知  jedis.publish(getRedisRegistryKey(), serverAddr + "-" + RedisListener.REGISTER);  } finally {  jedis.close();  }  } |

## 负载均衡

不是独立部署，在每一个基础微服务中。

<https://blog.csdn.net/wudiyong22/article/details/80829808>

@LoadBalanced 注解了解 Ribbon 负载均衡

### ribbon负载均衡原理：

LoadBalancerClient（RibbonLoadBalancerClient是实现类）在初始化的时候（execute方法），会通过ILoadBalance（BaseLoadBalancer是实现类），启动schedule（斯凯球）获取服务信息，过滤器默认只过滤出与消费端相同zone的服务，如果要保证高可用可配置ZoneAffinityServerListFilter过滤器，Eureka注册中心获取服务注册列表，并且每10s一次向EurekaClient发送“ping”，来判断服务的可用性，如果服务的可用性发生了改变或者服务数量和之前的不一致，则从注册中心更新或者重新拉取。LoadBalancerClient有了这些服务注册列表，就可以根据具体的IRule来进行负载均衡。

### 策略规则：(默认的)轮询

随机 (Random)

(默认的)轮询 (RoundRobin)

一致性哈希 (ConsistentHash)

哈希 (Hash)

加权（Weighted）

ILoadBalance 负载均衡器

1：在BaseLoadBalancer类下，BaseLoadBalancer的构造函数，该构造函数开启了一个PingTask任务setupPingTask();

PingTask源码，即new一个Pinger对象，并执行runPinger()方法。

2：查看Pinger的runPinger()方法，最终根据 pingerStrategy.pingServers(ping, allServers)来获取服务的可用性，如果该返回结果，如之前相同，则不去向EurekaClient获取注册列表，如果不同则通知ServerStatusChangeListener或者changeListeners发生了改变，进行更新或者重拉

CAP中的AP：与zookeeper的CP（一致、可靠）

### nginx、slb、f5

## hystrix服务降级容错

### hystrix原理

@EanbleCircuitBreaker开启

@ HyrtrixCommand(fallbackMethod =’testBack’) 注解标识调用方法

两种模式：信号量模式和线程池

原理分析：

1：创建一个HyrtrixCommand(返回单结果) 或者HrstrixObservableCommand (返回多结果)，采用了命令模式 来实现对服务调用的操作的封装。

调用着（Invoker）与操作者（Receiver）通过对Command命令接口实现了解耦，这两个都是对Command的进一步抽象定义。

为什么命令对象，callback进行了登记回调方法，系统奔溃也会执行这句分，因为一个命令对象和原先的请求有着不同的生命周期，请求者可能已经不在了，命令是活动的。可以调用undo和redo来进行撤销和重调。

2：命令执行

HyrtrixCommand的同步执行execute()和异步执行queue()colb有订阅在发布;

HrstrixObservableCommand返回多个消息体的observe()Hot，toObservable的ColdObservabl;

ps：hot是不管有没有订阅都会发布；colb是要有才会发布 p139页

大量的使用了，RxJava，观察者-订阅者模式。Observabl是“”、“被观察者”， 对应的subscriber可以理解为订阅者、观察者。Observal每次发送事件，会调用subscriber的onNext()方法，完成调用onComplete(),错误onError(),

3：命中缓存，有缓存返回 Observable对象

4：没有命中，检查断路是否打开，打开直接调用fallback处理，通俗叫服务降级。

5：没有打开断路的话，检测信号量或者线程池满，不行就跳\*fallback。这里的线程池，每个服务有自己专门的线程池。采用了 “舱壁模式”

6：HyrtrixCommand的run返回单一结果。和HrstrixObservableCommand的construct发射多个结果。出异常或者错误会跳\*fallback

7：计算断路器健康度，所有信息（成功、失败、超时、拒绝、）根据统计数据进行“熔断/短路”，恢复期间后还是不行，继续“熔断/短路”

### 断路器原理：存在与HystrixCircuitBreaker P144页

allRequest(),每个hystrix命令都通过他判断是否已经被执行；

isOpen() 返回当前断路器是否打开；

markSuccess 闭合断路器

### HystrixDashboard仪表盘

## Feign：ribbon+hystrix+Web

feign简化两者且加了Web

@enableFeignClients

@EnableDiscoveryClien

@FeignCLient(“testClient”)来制定服务名来绑定服务

ribbon.ConnextTimeout=500 会自动转换为 testClient. ribbon.ConnextTimeout=500

禁用hrytrix见P211页

## zuul服务级切面拦截：cas登陆系统

包含了ribbon和hrytrix的依赖

两个核心功能：面向服务、请求过滤

面向服务：屏蔽了各个微服务直接的调用细节

请求过滤：检查请求头部

## config集中式配置服务：动态读取git配置

## bus消息中心：kafka,rabbitmq sleuth

### stream这个用于消息中间件的解耦

## sleuth：链路追踪

dome：

logback-spring.xml 配置 见P395页

# K8s容器总结：

## 长连接：

一开始这个项目发布到服务器上面，只能支持一万的长连接。并且接口都访问不了。top命令看了linux的参数，发现CPU很内存都还很空。那么就不是linux的问题。查了很多资料，发现springboot项目默认的启动容器是tomcat,而tomcat默认支持1W的连接数量。超过就会拒绝。既然问题出在tomcat，那么现在就有两个方案。1、调大tomcat的连接数量。2、容器换成jetty。对于需要保持数十万的长连接，jetty无疑更适合作为启动容器。

会不会是linux本身有什么限制。导致只能维持1.6W＋的长连接呢？  
ulimit -a 参看linux的各种参数限制。

这个参数跟长连接的数量及其相似，初步猜测是不是这个参数的问题。  
ulimit -l unlimited 把这个参数调成了无限制。现在再来测试长连接的数量。  
已经可以达到5W+。因为测试工具的原因。没有做更高的压测，但是初步观察。保持10万的长连接应该是支持的。

原文链接：<https://blog.csdn.net/hgswsdn/article/details/106939964>

阿里的做法：

统一接入网关（ACCGW），可以理解成一个前置的 Nginx，是蚂蚁基于 Nginx 二次开发的一套组件，在内部我们叫做 Spanner，它在接入架构中主要负责非业务的那一部分逻辑处理，主要包括 SSL 的卸载，MMTP 的协议解析，数据的压缩、解压缩，客户端 TCP 长连接的维持，接入流量的总控，数据包的路由以及客户端日志的接入。API 网关、PUSH 推送、数据同步等组件，都在它的庇荫之下。

## 总结

自动化容器部署和复制。  
实时弹性收缩容器规模。  
容器编排成组，并提供容器间的负载均衡。

kubectl:客户端命令行工具，作为整个系统的操作入口。

kube-apiserver:以REST API服务形式提供接口，作为整个系统的控制入口。

kube-controller-manager:执行整个系统的后台任务，包括节点状态状况、Pod个数、Pods和Service的关联等。

kube-scheduler:负责节点资源管理，接收来自kube-apiserver创建Pods任务，并分配到某个节点。

etcd:负责节点间的服务发现和配置共享。

kube-proxy:运行在每个计算节点上，负责Pod网络代理。定时从etcd获取到service信息来做相应的策略。

kubelet:运行在每个计算节点上，作为agent，接收分配该节点的Pods任务及管理容器，周期性获取容器状态，反馈给kube-apiserver。

DNS：一个可选的DNS服务，用于为每个Service对象创建DNS记录，这样所有的Pod就可以通过DNS访问服务了。

## K8s常用命令：

### kubectl get - 列出可用资源

* 列出所有的pod  ： kubectl get pods --all-namespaces
* 列出所有的job :  kubectl get job --all-namespaces

### kubectl describe - 显示有关资源的详细信息

* kubectl describe pod nvjob-lnrxj -n default
* -n default 是指定namespace为default里的pod，是语法中的flag

### kubectl logs - 从 Pod 中的容器打印日志

* 这个特殊一点哦，这个不用指定TYPE，因为kubeclt logs 默认就是pod类型，所以 kubectl  logs  pod 会报错，"Error from server (NotFound): pods "pod" not found"
* kubectl logs calijob  -n calib

### kubectl exec - 在 Pod 中的容器执行命令

* 和log一样，不用指明type,默认是pod ： kubectl exec <pod\_name>  -n <namespace> date

### kubectl delete - 在 Pod 中的容器执行命令

* kubectl delete pod cali-2 -n calib
* 批量删除namespace 是calib中 状态为Error的所有pod：kubectl get pods -n calib | grep Error | awk '{print $1}' | xargs kubectl delete pod -n calib（注意 “Error”，“Completed”状态得首字母都是大写哦）

## doker命令：

### 容器生命周期管理

* [run](https://www.runoob.com/docker/docker-run-command.html)
* [start/stop/restart](https://www.runoob.com/docker/docker-start-stop-restart-command.html)
* [kill](https://www.runoob.com/docker/docker-kill-command.html)
* [rm](https://www.runoob.com/docker/docker-rm-command.html)
* [pause/unpause](https://www.runoob.com/docker/docker-pause-unpause-command.html)
* [create](https://www.runoob.com/docker/docker-create-command.html)
* [exec](https://www.runoob.com/docker/docker-exec-command.html) 进入容器

### 容器操作

* [ps](https://www.runoob.com/docker/docker-ps-command.html)
* [inspect](https://www.runoob.com/docker/docker-inspect-command.html)
* [top](https://www.runoob.com/docker/docker-top-command.html)
* [attach](https://www.runoob.com/docker/docker-attach-command.html) 进入容器
* [events](https://www.runoob.com/docker/docker-events-command.html)
* [logs](https://www.runoob.com/docker/docker-logs-command.html)
* [wait](https://www.runoob.com/docker/docker-wait-command.html)
* [export](https://www.runoob.com/docker/docker-export-command.html)
* [port](https://www.runoob.com/docker/docker-port-command.html)

### 容器rootfs命令

* [commit](https://www.runoob.com/docker/docker-commit-command.html)
* [cp](https://www.runoob.com/docker/docker-cp-command.html)
* [diff](https://www.runoob.com/docker/docker-diff-command.html)

### 镜像仓库

* [login](https://www.runoob.com/docker/docker-login-command.html)
* [pull](https://www.runoob.com/docker/docker-pull-command.html)
* [push](https://www.runoob.com/docker/docker-push-command.html)
* [search](https://www.runoob.com/docker/docker-search-command.html)

### 本地镜像管理

* [images](https://www.runoob.com/docker/docker-images-command.html)
* [rmi](https://www.runoob.com/docker/docker-rmi-command.html)
* [tag](https://www.runoob.com/docker/docker-tag-command.html)
* [build](https://www.runoob.com/docker/docker-build-command.html)
* [history](https://www.runoob.com/docker/docker-history-command.html)
* [save](https://www.runoob.com/docker/docker-save-command.html)
* [load](https://www.runoob.com/docker/docker-load-command.html)
* [import](https://www.runoob.com/docker/docker-import-command.html)

### info|version

* [info](https://www.runoob.com/docker/docker-info-command.html)
* [version](https://www.runoob.com/docker/docker-version-command.html)

## 管道：

### Azure Pipelines

海康内部搭建

# 目录结构：

数据结构

红黑树（5点特性）

思维拓展

一堆字符串求出现次数最多的字符串

Java

Java 基础部分

Java 集合框架

ArrayList 与 LinkedList

Hashtable、Hashmap 与 ConcurrentHashmap

Java 并发编程

线程状态

新建、可运行（运行和就绪）、阻塞、无限期等待、限期等待、死亡

实现线程三种方式（继承Thread，实现 Runnable 接口，实现 Callable 接口）

并发两个关键问题

线程通信

线程同步

Java内存模型（JMM），解决并发中的可见性，原子性，有序性问题

Java IO

序列化

AIO、BIO、NIO

Java 虚拟机

运行时数据区

数据库

操作系统

进程和线程的区别

资源、调度、开销、通信 4个方面来对比

线程间通信

进程通信

直接通信

间接通信

计算机网络

传输层

为什么三次握手，两次不行吗

四次挥手 TimeWait 作用

TCP可靠传输

分布式

分布式通信协议

http

https

RESTful API

分布式锁

项目

Cookie 和 Session 的原理

# 基础：

排序算法

选择排序

冒泡排序：

int i, j, temp;

for (i = 0; i < len - 1; i++)

for (j = 0; j < len - 1 - i; j++)

if (arr[j] > arr[j + 1]) {

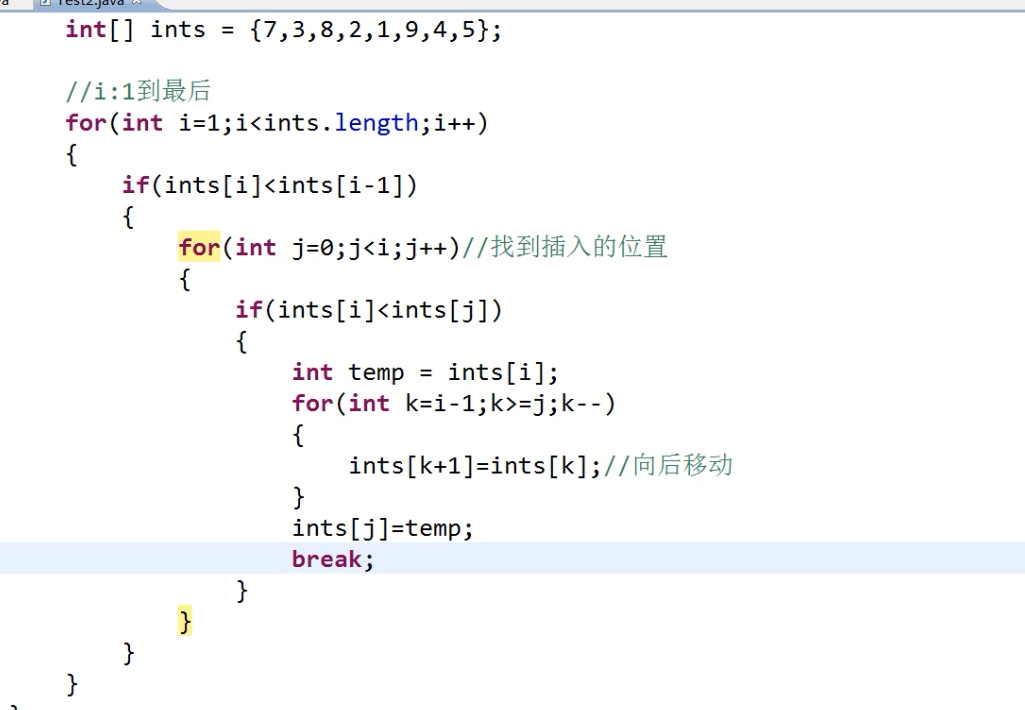
temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

插入排序：



快速排序

快速排序（普通，二路，三路）

归并排序

希尔排序

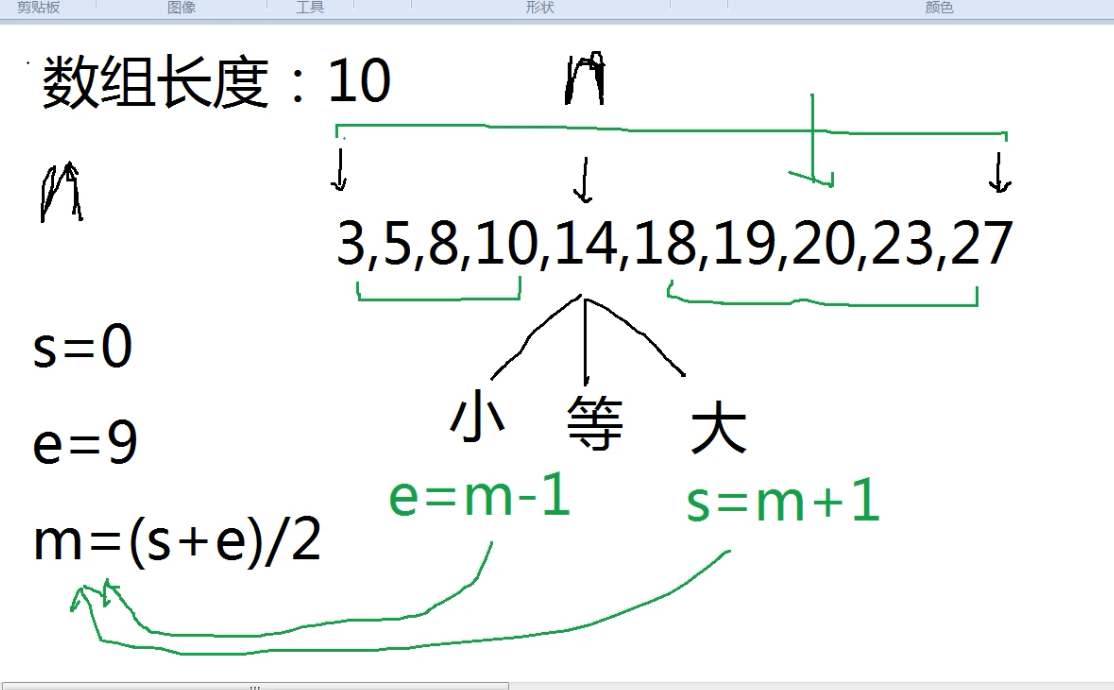
堆排序

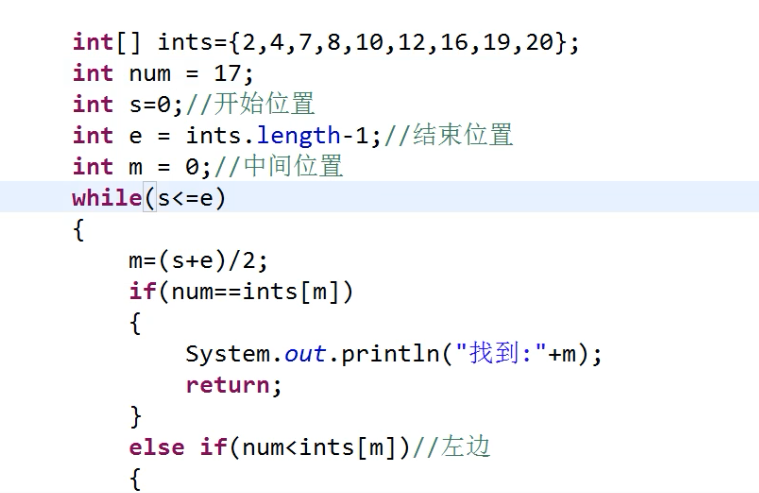
计数排序

桶排序

基数排序

二分查找：





Java 中的排序工具

原文：<https://blog.csdn.net/zj15527620802/article/details/83384839>

Equals：Integer比较常问的

-128到127

滨江：

## 第一章、初认识JAVA

### 8、Java有什么核心优势让其流行?

面向对象的编程思想,可以跨平台运行,分布式的开发模式,可靠,安全,多线程等

### 9、Java的三个版本是什么?

JavaME(迷你版,主要开发桌面应用),JavaSE(标准版),JavaEE(企业版)

10、Java是哪个公司发明的? 现在这家公司处境如何? 为什么?

Sun公司发明的，现已被甲骨文公司收购，因为在Sun公司在与微软的竞争中不敌微软，最终由于业绩问题被甲骨文公司以74美元收购。

### 11、Java的跨平台是通过什么实现的?

因为.java会被编译成.class文件,而负责执行.class文件的是java虚拟机,只要在不同的操作系统上安装正确的的虚拟机,就可以执行相同的.class文件,而其他语言没有JVM虚拟机,所以就无法实现跨平台操作.这个JVM虚拟机是java自己设计的计算机,相当于在你的电脑中又模拟了一个计算机.

### 12、Java语言释型还是编译型?还是两者都是?

两者都是

### 13、JDK、JRE、JVM的全称是?

JDK:英文全称叫Java Development Kit,他是JAVA开发的工具包(SDK),在JDK安装目录中有一个jre文件,其中包含俩个文件夹,一个是lib,一个是bin,bin中就是(JVM)java的虚拟机,而lib中就是JVM工作需要的类库,JVM和lib结合起来就是jre.

        JRE:英文全称叫Java Runtime Environment (java的运行环境),包含JVM标准实现及Java核心类库,JRE是java的运行环境,并不是一个开发环境,所以没有包含任何开发工具

        JVM:英文全称为Java Virtual Machine,是JAVA虚构出来的一个计算机,用于执行.class文件

### 14、JDK包含JRE，这句话对吗?

对

### 15、方法的重载和重写

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

16、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

## 第二章、数据类型和运算符、命名

### 33、如和将10进制转换为2进制

十进制转成十六进制： Integer.toHexString(int i)

十进制转成八进制Integer.toOctalString(int i)

十进制转成二进制Integer.toBinaryString(int i)

### 34、位、字节、字符的概念及其技术范围

1个字(word)是16位=2个字节

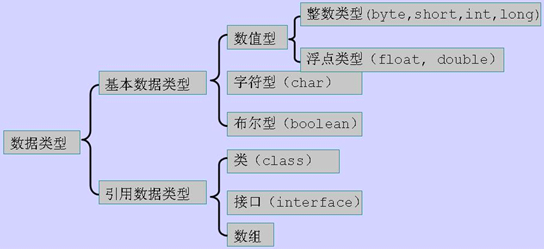
1个字节(byte)是8位=8bit(比特),

1个bit=1位

### 35、标识符的作用

java标识符是用来给类、对象、方法、变量、接口和自定义数据类型命名的

### 39、★★四类八种byte shot int long 类型分别占几个字节，代表的范围多大





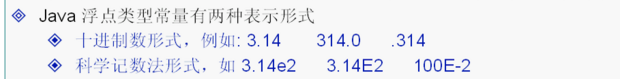
### 40、整型常数一般默认是什么类型

Int

### 41、float 和 double



### 42、科学计数法标识3.14



### 43、浮点常量默认是什么类型

Double

### 47、boolean类型占用空间是一位还是一个字节

就boolean类型没有给出具体的占用字节数，因为对虚拟机来说根本就不存在 boolean 这个类型，boolean类型在编译后会使用其他数据类型来表示，

### 49、常量的声明使用哪个关键字

Final

### 54、引用类型是占用几个字节的

引用本身是保留在栈中的 引用所指的对象，是存放在堆中的

62、什么情况下，加号会被虐成字符串连接付

在有” ”的情况下

### 正则表达式

<https://www.jb51.net/tools/zhengze.html>

### 为什么阿里巴巴禁止开发人员使用is Success作为变量名？

可以看到，由于不同的序列化工具，在进行序列化的时候使用到的策略是不一样的，所以，对于同一个类的同一个对象的序列化结果可能是不同的。

### 为什么阿里巴巴禁止把SimpleDateFormat定义成static变量?

SimpleDateFormat中的format方法在执行过程中，会使用一个成员变量calendar来保存时间。这其实就是问题的关键。

由于我们在声明SimpleDateFormat的时候，使用的是static定义的。那么这个SimpleDateFormat就是一个共享变量，随之，SimpleDateFormat中的calendar也就可以被多个线程访问到。

### 为什么阿里巴巴禁止开发人员修改serialVersionUID字段的值？

serialVersionUID是用来验证版本一致性的。所以在做兼容性升级的时候，不要改变类中serialVersionUID的值。

特别说明一下，由于本文标题并不完全能表达本文的全部内容，这里再强调一下：serialVersionUID 既然是验证版本一致性的，在做版本升级的时候（非兼容性升级）

，记得要修改这个字段的值哦，这样可以避免序列化混乱。

## 第三章、控制语句

### 70、三种控制结构是什么

顺序 条件 循环

### 73、switc语句的功能是否可以使用if else多选择结构来代替？如果是，为什么还要switch

首先要看一个问题，if 语句适用范围比较广，只要是 boolean 表达式都可以用 if 判断；而 switch 只能对基本类型进行数值比较。两者的可比性就仅限在两个基本类型比较的范围内。

说到基本类型的数值比较，那当然要有两个数。然后重点来了——

if 语句每一句都是独立的，看下面的语句：

if (a == 1) ...

else if (a == 2) ...

这样 a 要被读入寄存器两次，1 和 2 分别被读入寄存器一次。于是你是否发现其实 a 读两次是有点多余的，在你全部比较完之前只需要一次读入寄存器就行了，其余都是额外开销。但是 if 语句必须每次都把里面的两个数从内存拿出来读到寄存器，它不知道你其实比较的是同一个 a。

因为特定的规则，他一开始就知道你要比 a，于是 a 一次性读取，相比 if 节约了很多开销。然后根据大量的实际程序测试（不考虑不同的编译器优化程度差异，假设都是最好的优化），那么Switch语句击中第三个选项的时间跟if/else if语句击中第三个选项的时间相同。

击中第一，第二选项的速度if语句快，击中第四以及第四之后的选项的速度switch语句快。

### 74、switch中表达式的结果需要什么类型？

在java的1.7之后的jdk版本，java中的switch里面表达式的类型可以是string类型。（Jdk版本一定要1.7之后，1.5可能只能是int或者char类型的）

### 85、关键字continue、break和return的区别：

continue：跳出本次循环继续下一次循环

break： 跳出循环体，继续执行循环外的函数体

return: 跳出整个函数体，函数体后面的部分不再执行

### 89、java中参数传递是使用值传递还是引用传递

http://blog.csdn.net/panyongcsd/article/details/52371203

Java 语言的参数传递是值传递

### 90、方法定义中，形式参数和实际参数有什么区别；

1 形式参数：就是在定义函数或过程的时候命名的参数。通俗讲就是一个记号。2 实际参数：就是在执行时，调用函数或过程时，传递给函数或过程的参数。通俗讲就是实际值。3 参数嘛，就是一个可变量，它随着使用者的不同而发生变化。举个例子，在中学的时候学过sin(x)函数，这里的x就是形式参数，当你需要求1的正弦值时，你会使用sin(1)，这里的1就是实际参数。4 形参和实参间的关系：两者是在调用的时候进行结合的，通常实参会将取值

91、方法定义中，使用的是形参还是实参；

定义中使用的是形参

### 92、定义形参，必须要写变量类型吗；

是的

### 93、实参类型是否必须和形参类型所匹配

是

### 94、什么是方法重载

子类方法与父类同名，根据不同参数做出不同的判断，返回不同的参数

方法重载是让类以统一的方式处理不同类型数据的一种手段。调用方法时通过传递给它们的不同个数和类型的参数来决定具体使用哪个方法，这就是多态性。

### 97、递归的优缺点

1. 简洁2.在树的前序，中序，后序遍历算法中，递归的实现明显要比循环简单得多。

缺点：1.递归由于是函数调用自身，而函数调用是有时间和空间的消耗的：每一次函数调用，都需要在内存栈中分配空间以保存参数、返回地址以及临时变量，而往栈中压入数据和弹出数据都需要时间。->效率

2.递归中很多计算都是重复的，由于其本质是把一个问题分解成两个或者多个小问题，多个小问题存在相互重叠的部分，则存在重复计算，如fibonacci斐波那契数列的递归实现。->效率

3.调用栈可能会溢出，其实每一次函数调用会在内存栈中分配空间，而每个进程的栈的容量是有限的，当调用的层次太多时，就会超出栈的容量，从而导致栈溢出。->性能

## 第四章、数组

### 109:、数组类型可是任意类型吗？

引用类型，

### 111、数组的长度可以遍布？数组的长度使用哪个属性？

不可变。第一种是在数组初始化的时候长度申请的足够大，这样做会造成内存空间的浪费，一般不推荐使用。第二种是用java中提供的集合的方式存储数据，如List,Set和Map类型的对象存储数据，一方面这些类型的对象的长度都是动态增长的，另一方面这些类中提供了很多便于操作数据的方法，因此在对所需存储数据的多少不确定的时候，第二种方法比第一种方法更优秀。

Length

**为什么阿里巴巴要求谨慎使用ArrayList中的subList方法？**

也就是说subList 返回是一个视图，那么什么叫做视图呢？这个方法返回了一个SubList，这个类是ArrayList中的一个内部类。SubList这个类中单独定义了set、get、size、add、remove等方法。

因为SubList只是ArrayList的内部类，他们之间并没有集成关系，故无法直接进行强制类型转换。

为什么阿里巴巴禁止在 foreach 循环里进行元素的 remove/add 操作?

modCount是ArrayList中的一个成员变量。它表示该集合实际被修改的次数。

expectedModCount 是 ArrayList中的一个内部类——Itr中的成员变量。expectedModCount表示这个迭代器期望该集合被修改的次数。其值是在ArrayList.iterator方法被调用的时候初始化的。只有通过迭代器对集合进行操作，该值才会改变。

Itr是一个Iterator的实现，使用ArrayList.iterator方法可以获取到的迭代器就是Itr类的实例。

★★可以看到，它只修改了modCount，并没有对expectedModCount做任何操作。但是使用普通的for循环还是可以的，因为普通for循环并没有用到Iterator的遍历， 所以压根就没有进行fail-fast的检验。

解决：原生的循环

使用Java 8中提供的filter过滤

或者可以调整为iterator.remove(userName);

使用增强for循环其实也可以：删除完毕跳出;如果，我们非常确定在一个集合中，某个即将删除的元素只包含一个的话， 比如对Set进行操作，那么其实也是可以使用增强for循环的，只要在删除之后，立刻结束循环体，不要再继续进行遍历就可以了，也就是说不让代码执行到下一次的next方法。

**为什么阿里巴巴不建议在for循环中使用“+”进行字符串拼接？**

我们使用形如以上形式的代码，分别测试下五种字符串拼接代码的运行时间。得到结果如下：

+cost :5119

StringBuilder :3

StringBuffer:4

concat cost: 3626

StringUtil.jion:25726

## 第五章、面向对象

### 136、简述面向对象和面向过程的区别和联系

1：考虑问题的时候不应该考虑第一步我干嘛，第二部我干嘛，而应该考虑作为整个问题有哪些类

2：每种类的对象有属性和方法

3：类和类之间有什么关系

### 137、对象和类是什么关系

对象用计算机语言对问题域事务的描述，对象通过属性和方法来分别对应事务所具备的静态属性和动态属性。 人

类是用于描述统一类型对象的一种抽象概念，类中定义了这个对象所具有的属性。人类

类可以看成一类对象的模板，对象可以看成这个类的一个实例。 实例化一个对象

### 145、构造方法的作用

1.为了初始化成员属性，而不是初始化对象，初始化对象是通过new关键字实现的

2.通过new调用构造方法初始化对象，编译时根据参数签名来检查构造函数，称为静态联编和编译多态参数签名：参数的类型，参数个数和参数顺序）

3.创建子类对象会调用父类构造方法但不会创建父类对象，只是调用父类构造方法初始化父类成员属性；

### Java中有几种类型的流？

答：字节流和字符流。字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。在 java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是两种对称性（输入和输出的对称性，字节和字符的对称性）；二是两种设计模式（适配器模式和装潢模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

### 为什么阿里巴巴建议开发者谨慎使用继承?

面向对象的复用技术

每个人在刚刚学习继承的时候都会或多或少的有这样一个印象：继承可以帮助我实现类的复用。所以，很多开发人员在需要复用一些代码的时候会很自然的使用类的继承的方式，因为书上就是这么写的（老师就是这么教的）。但是，其实这样做是不对的。长期大量的使用继承会给代码带来很高的维护成本。

前面提到复用，这里就简单介绍一下面向对象的复用技术。

复用性是面向对象技术带来的很棒的潜在好处之一。如果运用的好的话可以帮助我们节省很多开发时间，提升开发效率。但是，如果被滥用那么就可能产生很多难以维护的代码。

作为一门面向对象开发的语言，代码复用是Java引人注意的功能之一

## 第六章、设计模式

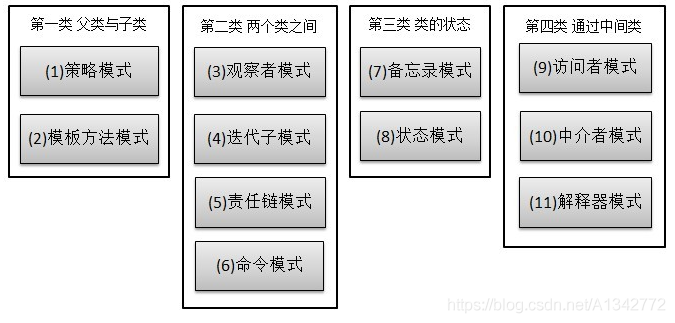
### 简介：详情见 【设计模式23.md】

按照三只类型可以分为：

创建类型5：工厂、抽象工厂（先定义抽象，在实现工厂）、单例（双重检查）、建造（顺序构建）、原型（拷贝类，提升性能）

结构类型7：适配器（A继承B实现C接口）、装饰器（继承演变）、代理（中间步骤交给代理来完成）、桥接（两个维度+建造）、外观（代理+建造）、组合（树形结构）、享元（有些类通过一个大类来存储，重复查询 导致大量内存溢出）

关系类型11：



# 算法：

## o(1), o(n), o(logn), o(nlogn)

表示对应算法的时间复杂度，是算法的时空复杂度的表示。不仅仅用于表示时间复杂度，也用于表示空间复杂度。

O后面的括号中有一个函数，指明某个算法的耗时/耗空间与数据增长量之间的关系。其中的n代表输入数据的量。

比如时间复杂度为O(n)，就代表数据量增大几倍，耗时也增大几倍。比如常见的遍历算法。再比如时间复杂度O(n^2)，就代表数据量增大n倍时，耗时增大n的平方倍，这是比线性更高的时间复杂度。比如冒泡排序，就是典型的O(n^2)的算法，对n个数排序，需要扫描n×n次，2次循环。

        再比如O(logn)，当数据增大n倍时，耗时增大logn倍（这里的log是以2为底的，比如，当数据增大256倍时，耗时只增大8倍，是比线性还要低的时间复杂度）。二分查找就是O(logn)的算法，每找一次排除一半的可能，256个数据中查找只要找8次就可以找到目标。

  O(nlogn)同理，就是n乘以logn，当数据增大256倍时，耗时增大256\*8=2048倍。这个复杂度高于线性低于平方。归并排序就是O(nlogn)的时间复杂度。

        O(1)就是最低的时空复杂度了，也就是耗时/耗空间与输入数据大小无关，无论输入数据增大多少倍，耗时/耗空间都不变。 哈希算法就是典型的O(1)时间复杂度，无论数据规模多大，都可以在一次计算后找到目标（不考虑冲突的话）

## c语言网络编程-内核：select，poll，epoll。

从事服务端开发，少不了要接触网络编程。epoll 作为 Linux

下高性能网络服务器的必备技术至关重要，nginx、Redis、Skynet 和大部分游戏服务器都使用到这一多路复用技术。

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

### 支持一个进程所能打开的最大连接数

select：单个进程所能打开的最大连接数有FD\_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是3232，同理64位机器上FD\_SETSIZE为3264），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。

poll：poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的。

epoll：虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接。

### FD剧增后带来的IO效率问题

select：因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。

poll：同上

epoll：因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

### 消息传递方式

select：内核需要将消息传递到用户空间，都需要内核拷贝动作

poll：同上

epoll：epoll通过内核和用户空间共享一块内存来实现的。

版权声明：本文为CSDN博主「panamera12」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/wteruiycbqqvwt/article/details/90299610原文链接：<https://blog.csdn.net/wteruiycbqqvwt/article/details/90299610>

### epoll 原理：红黑树+双链表数据+回调机制

通过在Linux内核中申请一个简易的文件系统(文件系统一般用什么数据结构实现？B+树)。把原先的select/poll调用分成了3个部分

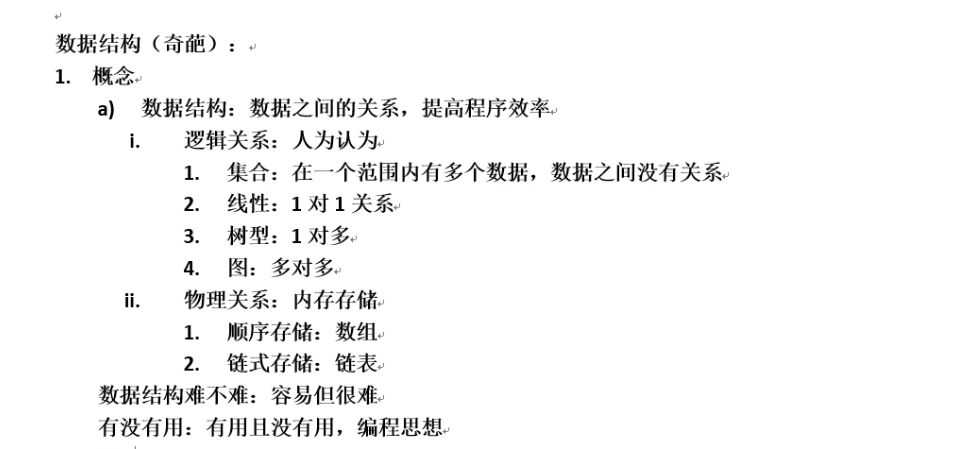
1）调用epoll\_create()建立一个epoll对象(在epoll文件系统中为这个句柄对象分配资源)

2）调用epoll\_ctl向epoll对象中添加这100万个连接的套接字

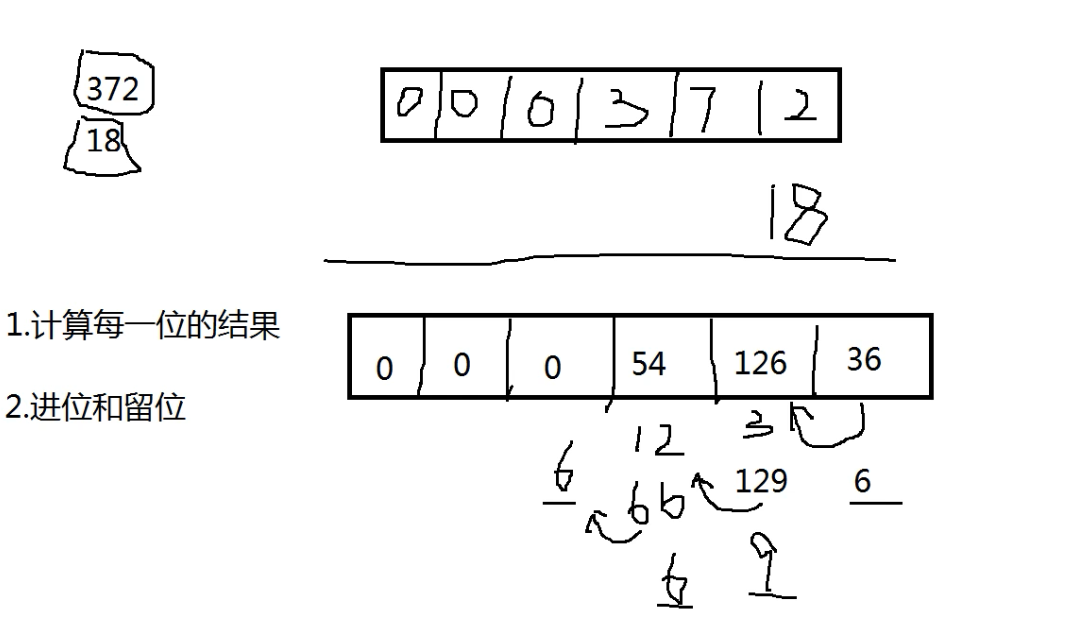
3）调用epoll\_wait收集发生的事件的连接

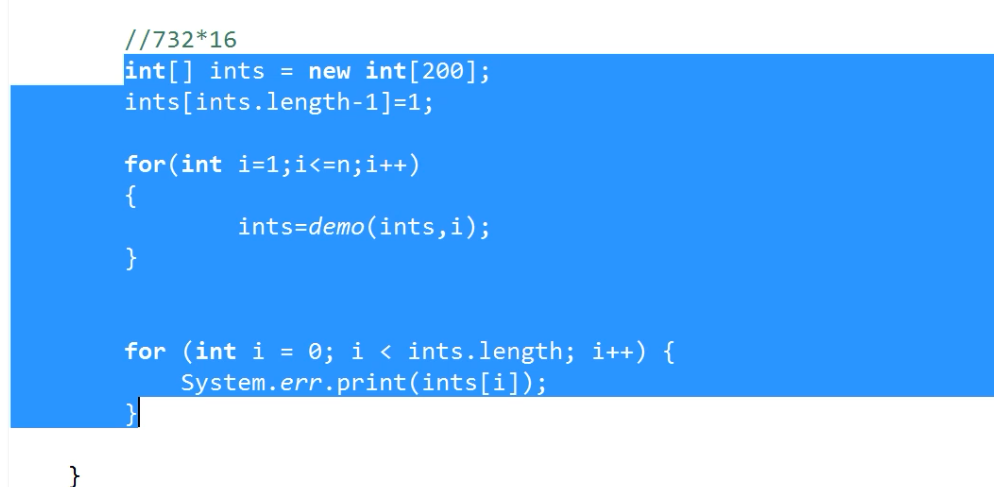
## 编程思想：

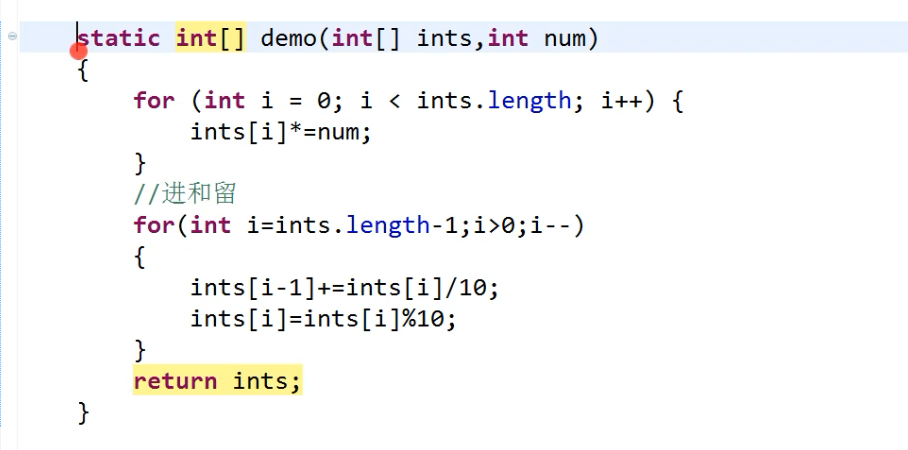
问人问题的时候，不会告诉你这个代码怎么写这么写，而是给你思路。大型公司的领导阶级才需要进行编写，小型公司只想你快速上手，第一天就可以开始干活，写的更多是业务代码。



第一题：阶乘，如果成出来超过了 int double bigdecimal ，那么需要证明表达，用数组：







## 垃圾回收算法：

最基本的垃圾收集算法有四种：mark-sweep 标记-清除算法 标记-压缩算法(mark-compact),复制算法(copying)以及引用计数算法(reference counting).

### mark-sweep 标记-清除算法

https://www.jianshu.com/p/b0f5d21fe031

在标记阶段，collector从mutator根对象开始进行遍历，对从mutator根对象可以访问到的对象都打上一个标识，一般是在对象的header中，将其记录为可达对象。

而在清除阶段，collector对堆内存(heap memory)从头到尾进行线性的遍历，如果发现某个对象没有标记为可达对象-通过读取对象的header信息，则就将其回收。

### 标记-压缩算法(mark-compact)：老年代

https://www.jianshu.com/p/698eb5e1ccb9 有点深奥

内存碎片一直是非移动垃圾回收器(指在垃圾回收时不进行对象的移动)的一个问题，比如说在前面的标记-清除垃圾回收器就有这样的问题。而标记-压缩垃圾回收算法能够有效的缓解这一问题。

算法原理

既然叫标记-压缩算法，那么它也分为两个阶段，一个是标记(mark)，一个是压缩(compact). 其中标记阶段跟标记-清除算法中的标记阶段是一样的

而对于压缩阶段，它的工作就是移动所有的可达对象到堆内存的同一个区域中，使他们紧凑的排列在一起，从而将所有非可达对象释放出来的空闲内存都集中在一起，通过这样的方式来达到减少内存碎片的目的。

在压缩阶段，由于要移动可达对象，那么需要考虑移动对象时的顺序，一般分为下面三种：

任意顺序 - 即不考虑原先对象的排列顺序，也不考虑对象间的引用关系，随意的移动可达对象，这样可能会有内存访问的局部性问题。

线性顺序 - 在重新排列对象时，会考虑对象间的引用关系，比如A对象引用了B对象，那么就会尽可能的将A，B对象排列在一起。

滑动顺序 - 顾名思义，就是在重新排列对象时，将对象按照原先堆内存中的排列顺序滑动到堆的一端。

现在大多数的垃圾收集算法都是按照任意顺序或滑动顺序去实现的

Two-Finger 算法

Two-Finger算法来自Edwards, 它在压缩阶段移动对象时是任意顺序移动的，它最适用于处理包含固定大小对象的内存区域。由于Mark阶段都是跟标记-清除算法一致的，这里我们只关注Compact阶段。

Two-Finger算法是一个Two Passes算法，即需要遍历堆内存两次，第一次遍历是将堆末尾的可达对象移动到堆开始的空闲内存单元去，第二次遍历则需要修改可达对象的引用，因为一些可达对象已经被移动到别的地址，而原先引用它们的对象还指向着它们移动前的地址。

在这两次遍历过程中，首尾两个指针分别从堆的头尾两个位置向中间移动，直至两个指针相遇，由于它们的运动轨迹酷似两根手指向中间移动的轨迹，因此称为Two Finger算法。

优点：

解决内存碎片问题。

缺点：压缩阶段，由于移动了可用对象，需要去更新引用。

### 复制算法(copying) ：新生代

新生对象被分配到A块中未使用的内存当中。当A块的内存用完了， 把A块的存活对象复制到B块。

清理A块所有对象。

新生对象被分配到B块中未使用的内存当中。当B块的内存用完了， 把B块的存活对象复制到A块。

清理B块所有对象。

循环1。

这种算法简单高效，但是内存代价极高，有效内存只为总内存的一半，会浪费掉50%的空间。所以这种算法只是纸面算法，不具备可用性，一般来说都会使用优化的复制算法。

这种方法不涉及到对象的删除，只是把可用的对象从一个地方拷贝到另一个地方，因此适合大量对象回收的场景，比如新生代的回收

因为年轻代中的对象基本都是朝生夕死的(80%以上)，所以在年轻代的垃圾回收算法使用的是复制算法，

复制算法的基本思想就是将内存分为两块，每次只用其中一块，当这一块内存用完，就将还活着的对象复制到另外一块上面。复制算法不会产生内存碎片。

过程： Eden+S0(From Survivor区)可分配新生对象；

对Eden+S0(From Survivor区)进行垃圾收集，Eden存活对象复制到S1(To Survivor区),From Survivor区仍然存活的对象,根据他们的年龄决定去向,年龄达到阈值的去年老代中,未达到阈值的去To Survivor区,此时Eden+S0(From Survivor区)相当于是被清空了,对象都被移到了其他区,该回收的也被回收了。一次新生代GC结束。

From Survivor区和To Survivor区交换,Eden+S1(此时的S1相当于From Survivor区)可分配新生对象；

对Eden+S1进行垃圾收集，存活对象复制到S0(此时的S0相当于To Survivor区)。清理Eden+S1(From Survivor区)。二次新生代GC结束。

循环1。

https://blog.csdn.net/kzadmxz/article/details/96574203

在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用“复制算法”，只需要付出少量存活对象的复制成本就可以完成收集。

在老年代中，因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

### 引用计数算法(reference counting)：

https://www.cnblogs.com/WJ5888/p/4359783.html

引用计数算法作为垃圾收集器最早的算法，有其优势，也有其劣势，虽然现在的JVM都不再采用引用计数算法进行垃圾回收【例如Sun的Java hotspot采用了火车算法进行垃圾回收】，

但这种算法也并未被淘汰，在著名的单进程高并发缓存Redis中依然采用这种算法来进行内存回收【后绪会以Redis作为例子，说明该算法】

直白一点，就是对于创建的每一个对象都有一个与之关联的计数器，这个计数器记录着该对象被使用的次数，垃圾收集器在进行垃圾回收时，

对扫描到的每一个对象判断一下计数器是否等于0，若等于0，就会释放该对象占用的内存空间,同时将该对象引用的其他对象的计数器进行减一操作

两种实现方式

侵入式与非侵入性，引用计数算法的垃圾收集一般有侵入式与非侵入式两种，侵入式的实现就是将引用计数器直接根植在对象内部，用C++的思想进行解释就是，

在对象的构造或者拷贝构造中进行加一操作，在对象的析构中进行减一操作，非侵入式恩想就是有一块单独的内存区域，用作引用计数器

算法的优点

使用引用计数器,内存回收可以穿插在程序的运行中，在程序运行中，当发现某一对象的引用计数器为0时，可以立即对该对象所占用的内存空间进行回收，

这种方式可以避免FULL GC时带来的程序暂停，如果读过Redis 1.0的源码，可以发现Redis中就是在引用计数器为0时，对内存进行了回收

算法的劣势

采用引用计数器进行垃圾回收，最大的缺点就是不能解决循环引用的问题，例如一个父对象持有一个子对象的引用，子对象也持有父对象的引用，这种情况下

，父子对象将一直存在于JVM的堆中，无法进行回收,代码示例如下所示(引用计数器无法对a与b对象进行回收):

### 当然Redis中还有LRU算法，实现内存淘汰策略

LRU算法：

https://www.cnblogs.com/wyq178/p/9976815.html

最近使用的页面数据会在未来一段时期内仍然被使用,已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用

活跃的用户做一个排名

## 数据结构与算法

排序算法

选择排序

冒泡排序

插入排序

快速排序

快速排序（普通，二路，三路）

归并排序

希尔排序

堆排序

计数排序

https://www.jianshu.com/p/ff1797625d66

桶排序

基数排序

二分查找

Java 中的排序工具

# 数据库：

## 为什么阿里巴巴禁止使用count(列名) 或count(常量) 来替代count(\*)？

那么列名、 常量 和 \*这三个条件中，常量 是一个固定值，肯定不为NULL。\*可以理解为查询整行，所以肯定也不为NULL，那么就只有列名的查询结果有可能是NULL了。

存储引擎

## 索引（衡量标准：IO渐进复杂度）

Hash（冲突，无法做范围查询）

FullText（全文搜索，优化：前缀索引）

R-Tree（空间索引，附近三公里的数据怎么查询）

B+ Tree（B Tree：每个key、data信息存在结点，B+ Tree存在叶子节点）

https://www.cnblogs.com/liuning8023/archive/2012/11/17/2775472.html

## 哪些情况下不会使用索引

全表扫描更快

条件中有or

不是复合索引的第一部分

like以%开始

## 事务

<https://zhuanlan.zhihu.com/p/69380112>

<https://www.cnblogs.com/jywy/p/11058067.html>

https://www.jianshu.com/p/dd6a340e50b2

### 简介ACID

Oracle隔离机制： 读提交（默认） 串行读

Mysql默认的事务隔离级别是**可重复读(Repeatable Read)**

事务所提供的安全保证，通常由首字母缩略词ACID来描述——原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。 A原子+I隔离 =C一致

原子性：

中止出错的事务，并撤销该事务进行的所有变更。叫做可中止性更通俗易懂。

一致性

应用程序可以用原子性和隔离性来实现一致性

如果做一个财务系统，账户A转钱到账户B，那么账户A中减少的钱与账户B中增加的钱必须相等。这并不是数据库可以保证的事情，而是应用程序来保证的

原子性，隔离性、持久性都是数据库的属性，而一致性是应用程序的属性。应用程序可以用原子性和隔离性来实现一致性，所以ACID中的C就是用来凑缩写单词的。

隔离性：

线程执行同时执行的事务不能相互冒犯。这意味着，每个事务都可以认为它是唯一在整个数据库上运行的事务。数据库确保多个事务提交后，结果与它们按顺序一个接一个地运行是一样的，尽管实际上它们可能是并发执行的。

持久性：

数据备份 存储 复制镜像

* RU读未提交（READ UNCOMMITTED）
* RC读已提交READ COMMITTED
* RR 可重复读 （REPEATABLE READ）
* S 串行化 （SERIALIZABLE）



### RC读已提交READ COMMITTED：这种最弱的隔离级别

脏读：

读未提交，就是脏毒，字面意思理解，别人都没有提交你这么能读到，根本没有事务可言；

未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。读到了并一定最终存在的数据，这就是脏读。

先前的写入尚未提交，后面的写入又覆盖了这个尚未提交的值，会怎么样呢？这就叫做**脏写。**

1. 从数据库读取时，只能看到已提交的数据（不**脏读**）。
2. 写入数据库时，只会覆盖已经提交的写入数据（不**脏写**）。

**为了防止脏读，**每次写入前，数据库都会记住旧值。 当前事务尚未提交时，其他事务的读取都会拿到旧值。当前事务提交后，其他事务才能读取到新值。

**为了防止脏写，**数据库一般用**行锁。**当事务想要修改特定的行时，必须先获得该行的锁。一次只有一个事务可持有任何给定行的锁。如果另一个事务要写入同一行，就必须等到第一个事务提交或回滚后。

### RR-可重复度-快照隔离

是这个问题最常见的解决方案

**读偏差：**

账户A和账户B，每个存有500元，一共1000元。现在发起一笔转账，从账户A转100元到账户B。发起转账交易后，小明马上查询两个账户的余额。不幸的是，对账户B的查询发生在转账事务提交前（显示500元），对账户A的查询发生在转账事务提交后（显示400元）—— 似乎有100元不翼而飞！

它对长时间运行的只读查询（如备份和分析）非常有用。做法是，每个事务都从数据库的**一致快照**中读取——也就是说，**事务始终可以看到事务开始时数据库中的所有数据**。即使这些数据随后被其他事务更改，这个事务也只能看到该特定时间点的旧数据。

快照隔离的实现通常使用**写锁**来防止脏写（与读已提交的行锁其实是一样的）：写入型事务会阻止另一个事务修改同一个对象，但是读取不需要任何锁定。从性能的角度来看，快照隔离做到了：**读不阻塞写，写不阻塞读**。

可是这与**读已提交**有啥子区别呢？

区别在于，**读已提交为每个查询保留单独的快照，而快照隔离为每个事务保留相同的快照。**在快照隔离级别下，如果我们发起多个长时间的查询请求，数据库系统往往需要为每个被查询的行保留N多个不同的版本，用一个唯一的、自增的**事务ID**来标识。因为正在进行的多个事务需要看到数据库在不同时刻的状态。这允许了数据库在处理长时间查询时，可以同时处理写入操作，且写入操作不会影响查询结果的正确性，且快照隔离也不会产生什么太大的额外开销

https://zhuanlan.zhihu.com/p/55819387

### 丢失更新==乐观锁：

**读已提交**和**快照隔离**级别，可以保证**只读事务在并发写入时**可以看到什么，却忽略了两个事务并发写入的问题。如果两个写入事务同时执行，则其中一个的修改可能会丢失，因为第二个写入的内容并没有包括第一个事务的修改——这就是**丢失更新**。

**1、原子写：**

**2、for update显式锁定：**

**3、比较并设置（CAS, Compare And Set）：**

### 物化冲突-幻读

如果幻读的问题是没有对象可以加锁

一个事务的写入改变了另一个事务的查询结果的正确性，被称为**幻读**

现在，要创建预订的事务可以锁定这个新创建的表了。在锁定之后，可以检查重叠的预订并像以前一样插入新的预订。这个表不用来存储任何与预订相关的信息，它完全就是一排锁，用于防止并发写入时的幻读。实践中，想要弄清楚如何物化冲突可能很难也很容易出错，这种做法也不优雅。如果没有其他办法，物化冲突应被视为最后的手段。

幻读是针对数据**插入（INSERT）**操作来说的。假设事务A对某些行的内容作了更改，但是还未提交，此时事务B插入了与事务A更改前的记录相同的记录行，并且在事务A提交之前先提交了，而这时，在事务A中查询，会发现好像刚刚的更改对于某些数据未起作用，但其实是事务B刚插入进来的，让用户感觉很魔幻，感觉出现了幻觉，这就叫幻读。

### 2PL最强的隔离级别，串行化-两阶段锁定：共享锁、互斥锁

只要没有写入，就允许多个事务同时读取同一行。但只要有写入（修改或删除），就独占访问权限。换句话说，写入不仅会阻塞其他写入，也会阻塞读——这是和快照隔离之间的关键区别。

**第一阶段（只加锁）：**事务开始前，进程尝试对所有此事务需要的行进行加锁，按顺序一次锁一行，查询就加共享锁，修改就加互斥锁。若第一阶段**全部加锁成功**，就开始**第二阶段（只解锁）：**执行更新然后**释放所有锁**。若在第一阶段某个进程加锁时发生冲突，则该进程**释放它所有加锁的行，**然后**重试**第一阶段。

两阶段提交用到了两种锁，**共享锁**（也称为**读锁，**允许多个事务同时读取同一行）或**互斥锁**（也称为**写锁，**只要有写入就独占访问权限），这是它们的经典使用场景。

两阶段锁定真正实现了**串行化**性质，它可以防止之前讨论的所有并发问题，也是性能损耗最大的选择，尤其是它可能会更频繁地导致死锁出现。事务由于死锁而被中止后只能重试，这意味着巨大的开销。

### 2PC两阶段提交Two-phaseCommit

在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持一致性而设计的一种算法(Algorithm)。

所谓的两个阶段是指：第一阶段：准备阶段(投票阶段)和第二阶段：提交阶段（执行阶段）。

缺点：

缺点：

1、同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

2、单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。（如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题）

3、数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

4、二阶段无法解决的问题：协调者再发出commit消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

### 三阶段提交（Three-phase commit）

1、引入超时机制。同时在协调者和参与者中都引入超时机制。  
2、在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

3PC主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

### 分布式事务如何保持一致性：

<https://www.cnblogs.com/luxiaoxun/p/8832915.html>

### **异步确保最终一致性：**

它的核心思想是将需要分布式处理的任务通过消息或者日志的方式来异步执行，消息或日志可以存到本地文件、数据库或消息队列，再通过业务规则进行失败重试，它要求各服务的接口是幂等的。

### MQ（事务消息）

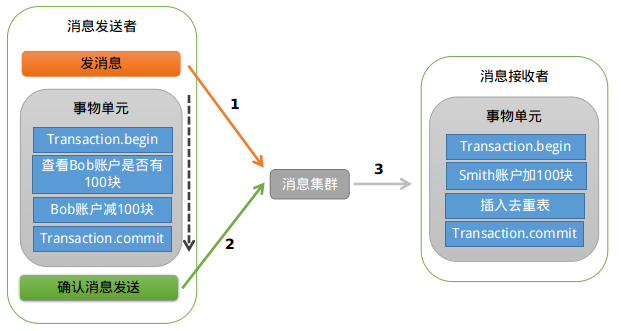
举个例子，Bob向Smith转账，那我们到底是先发送消息，还是先执行扣款操作？

好像都可能会出问题。如果先发消息，扣款操作失败，那么Smith的账户里面会多出一笔钱。反过来，如果先执行扣款操作，后发送消息，那有可能扣款成功了但是消息没发出去，Smith收不到钱。除了上面介绍的通过异常捕获和回滚的方式外，还有没有其他的思路呢？

下面以阿里巴巴的RocketMQ中间件为例，分析下其设计和实现思路。

RocketMQ第一阶段发送Prepared消息时，会拿到消息的地址，第二阶段执行本地事物，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。细心的读者可能又发现问题了，如果确认消息发送失败了怎么办？RocketMQ会定期扫描

消息集群中的事物消息，这时候发现了Prepared消息，它会向消息发送者确认，Bob的钱到底是减了还是没减呢？如果减了是回滚还是继续发送确认消息呢？RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。如下图：



各大知名的电商平台和互联网公司，几乎都是采用类似的设计思路来实现“最终一致性”的。这种方式适合的业务场景广泛，而且比较可靠。不过这种方式技术实现的难度比较大。目前主流的开源MQ（ActiveMQ、RabbitMQ、Kafka）均未实现对事务消息的支持，所以需二次开发或者新造轮子。比较遗憾的是，RocketMQ事务消息部分的代码也并未开源，需要自己去实现。

## 分布式事务Seata: RPC协议

简介：

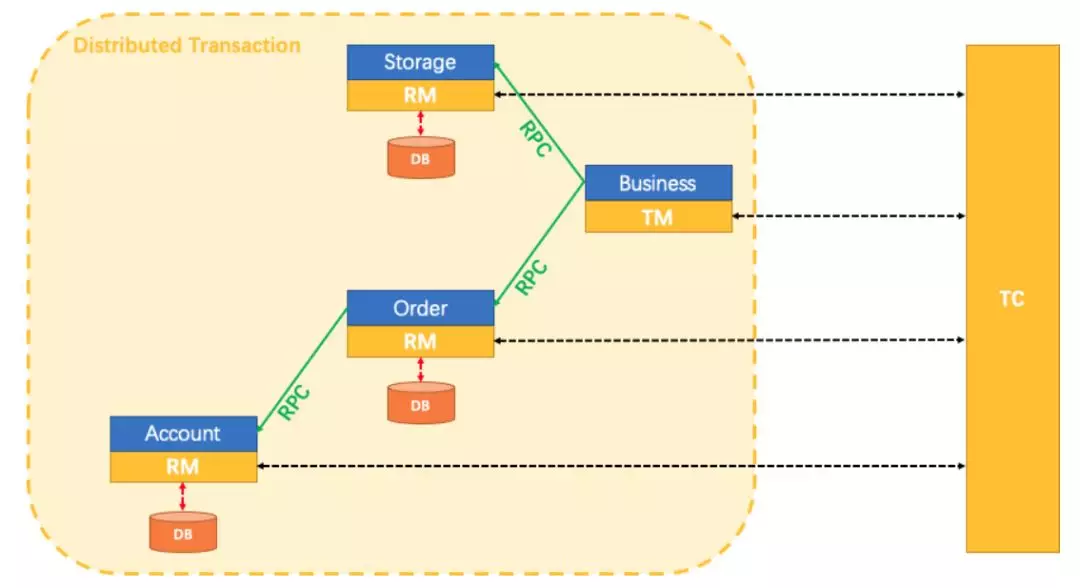
★★ 源码：https://zhuanlan.zhihu.com/p/87100741

### 历史 阿里14年起 19年改

1. TXC：Taobao Transaction Constructor，阿里巴巴中间件团队自 2014  
   年起启动该项目，以满足应用程序架构从单一服务变为微服务所导致的分布式事务问题。
2. GTS：Global Transaction Service，2016 年 TXC 作为阿里中间件的产品，更名为 GTS 发布。
3. FESCAR：2019 年开始基于 TXC/GTS 开源 FESCAR。
4. 后更名为Seata

社区里也有一些开源的分布式解决方案的框架，比如ByteTCC、LCN，但是这些框架没有一个权威的组织在维护

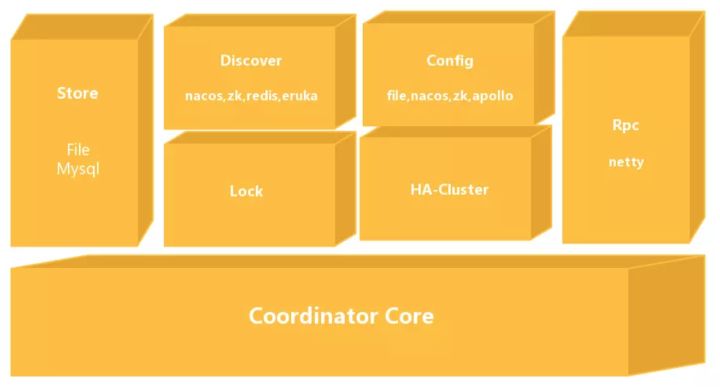
* Transaction Coordinator(TC)：维护全局和分支事务的状态，驱动全局事务提交与回滚。
* Transaction Manager™：定义全局事务的范围：开始、提交或回滚全局事务。
* Resource Manager(RM)：管理分支事务处理的资源，与 TC通信以注册分支事务并报告分支事务的状态，并驱动分支事务提交或回滚。



### Transaction Coordinator (TC)：

事务协调器，维护全局事务的运行状态，负责协调并驱动全局事务的提交或回滚。

整体的模块图如上所示:



#### Coordinator Core:

在最下面的模块是事务协调器核心代码，主要用来处理事务协调的逻辑，如分支的注册, commit, rollback 等协调活动。

当 TC 启动时, 先恢复本机的 Session, 然后启动 RPC Server, 最后注册自己的地址到注册中心, 这些我们前面已经介绍过了, 除此之外, TC 还会启动几个后台线程, 这些线程保证了 TC 的协调工作能够在发生错误时, 最终能顺利完成。

后台任务分别是回滚重试, 提交重试, 异步提交, 超时检测, 删除没用的 AT 模式 undo log。

#### Discover: 服务注册/发现模块，用于将 Server 地址暴露给我们 Client。

订阅的过程如下: 1. 获取当前 clusterName 名字 2. 判断当前 cluster 是否已经获取过了，如果获取过就从map中取。 3. 从 Redis 拿到地址数据，将其转换成我们所需要的数据。 4. 将数据变动的 Listener 注册到 Redis

我们知道 Server 是将自己注册到注册中心, 然后 Client 订阅更新, 并得到 Server 的列表, 最后通过负载均衡选择一个 Server 进行连接。当连接建立成功后, Server 会保存所有的连接, 在需要进行分支回滚和提交时, 从所有 RM 的连接记录中, 找到对应 RM 的所有连接, 它会首先寻找最原始的 RM 节点, 如果该节点宕机了, 它会找到该 RM 的其他节点, 然后发送分支提交请求。

#### **Config: 只有随机和轮训,用来存储和查找我们服务端的配置。**

getShort/getInt/Long/Boolean/Config()：通过dataId来获取对应的值。

putConfig：用于添加配置。

removeConfig：删除一个配置。

add/remove/get ConfigListener：添加/删除/获取 配置监听器，一般用来监听配置的变更。

目前为止有四种方式获取 Config：File(文件获取), Nacos, Apollo, ZK，etcd。在 Seata 中首先现在项目 resources 下保存一个 registry.conf 文件，在该文件中配置具体使用 Config 接口哪个实现类。

#### Lock: 读已提交隔离，锁模块，用于给 Seata 提供全局锁的功能。

Seata 中可以保证写操作的互斥性，而读的隔离级别一般是读未提交，但是提供了达到读已提交隔离的手段。

* acquireLock：用于对我们的 BranchSession 加锁，这里虽然是传的分支事务 Session，实际上是对分支事务操作的数据行加锁，成功返回 true。
* isLockable：根据事务 ID，资源 ID，锁住的Key来查询是否已经加锁。
* releaseLock: 释放分支事务的所有锁。
* releaseGlobalSessionLock： 释放全局事务的所有分支事务的锁。
* cleanAllLocks：清除所有的锁。

一种是基于内存的锁(Session 存储模式为 File 时使用), 一种是基于 DB 的(Session 存储模式为 DB 时使用)，它们都实现了 Locker 接口

#### Store: 存储模块，用来将我们的数据持久化，防止重启或者宕机数据丢失。

RM 可能会发生频繁回滚那么其完全无法应对高并发的场景。

在 Seata 中默认提供了文件方式的存储，下面我们定义我们存储的数据为 Session，而我们的TM创造的全局事务数据叫 GlobalSession，RM 创造的分支事务叫 BranchSession，一个 GlobalSession 可以拥有多个 BranchSession。我们的目的就是要将这么多 Session 存储下来。

Seata 中目前有 2 种实现方案, 一种是基于文件的, 一种是基于 DB 的

##### File：

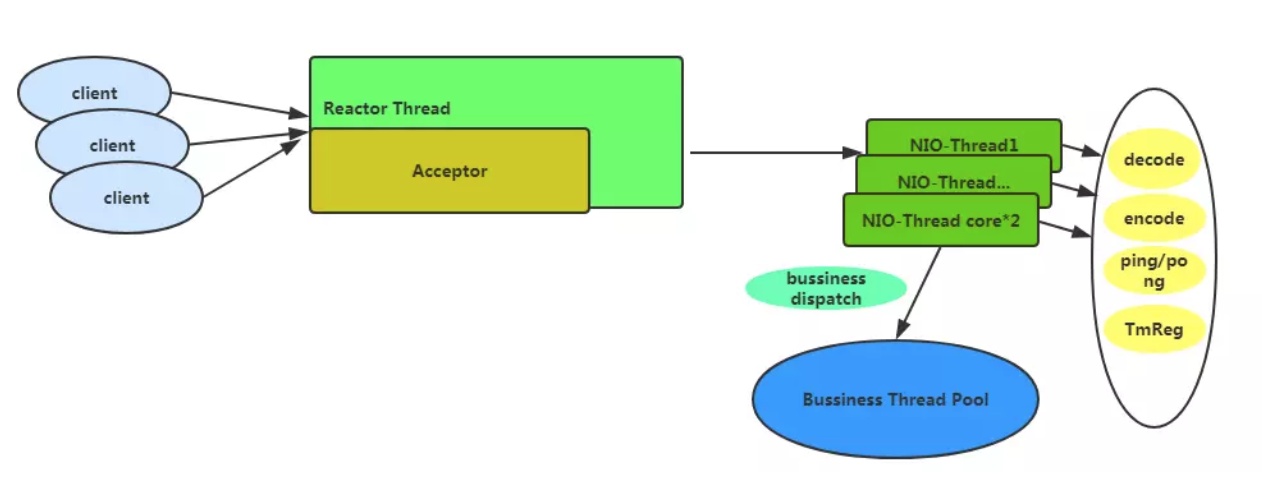
基于文件的实现是 FileTransactionStoreManager, 它可以使用同步刷盘或异步刷盘的策略，每当有 Session 的状态的更新时，它都会将变化的内容存储起来。为了防止存储文件的无限增殖，当达到一定条件时，它会另打开一个文件从头开始记录，并将之前的文件保存起来。这里有一个非常巧妙的设计，就是该方案既能保证所有超时事务不丢，只有已完成的事务被清除，同时文件的大小也得到了控制。

如果 TC 宕机，重启时只要先读取 historyFullFile，再读取 currentDataFile 就能恢复所有数据。

##### db：

基于 DB 的实现相较于基于文件的实现就显得朴实无华，logStore 实际上就是一个 DAO 层的接口，对应了数据的 CRUD，在重启恢复时只不过是按照条件遍历 DB 中的所有数据，进行 Session 恢复。

#### Rpc: 用于和其他端通信。



如果采用默认的基本配置, 那么会有一个 Acceptor 线程用于处理客户端的链接，会有 cpu\*2 数量的 NIO-Thread，在这些 NIO-Thread 线程中不会做业务太重的事情，只会做一些速度比较快的事情，比如编解码，心跳事件，和 TM 注册。一些比较费时间的业务操作将会交给业务线程池，默认情况下业务线程池配置为最小线程为 100，最大为 500。

**关于 Netty 的使用基础, 我们这里就不详细介绍了, 简单说就是对于每个连接都会绑定上数据的 handler, 它会按照责任链的原则, 顺着 handler 的绑定顺序, 处理数据, 这里简单看下它都绑定了什么 handler:**

#### HA-Cluster: 高可用集群，目前还没开源。

为 Seata 提供可靠的高可用功能。

### Transaction Manager (TM)

首先 TM 在启动的时候会去连接 TC Server, 然后然后通过该 TM Client 与 TC 模块进行通讯。在 TM 模块中最核心的接口就是 GlobalTransaction, 里面包含了全局事务的创建, 提交, 回滚过程, 其实质就是向 TC 发送 RPC 请求。

初始化：

#### TransactionManager：

它调用 transactionManager 发送消息, 然后将涉及到的全局事务 XID 保存起来.

TransactionManager 才是真正做实事的, 消息的发送工作都在这里完成。好了, 至此我们知道了哪个接口管理着全局事务的记录, 哪个接口真正进行 RPC 调用, 那么谁才是这些接口的真正调用者呢? Seata 使用了模板方法模式来进行这部分工作

#### TransactionalTemplate：模板方法：

1. 看看当前是不是已经在一个分布式事务中了, 如果是, 则复用现存的全局事务, 否则创建新的 - 什么时候会出现已经存在全局事务的情况呢? 假设 A 调用了 B, A 创建了全局事务 GT1, B 碰巧也执行了上述的模板, 这时候 B 就不会创建新的全局事务, 而是使用 GT1, 这实际上是前面提到的事物的传播

2. 如果是自己创建的全局事务, 则发 RPC 开始事务, 如果不是自己创建的则什么都不干 3. 执行真正的业务逻辑

4. 如果发生了异常, 如果自己创建全局事务, 才负责回滚, 否则就只管异常外抛

5. 如果没发生异常, 如果自己创建全局事务, 才负责提交, 否则就什么都不做

6. 清理工作

#### GlobalTransactional切面

#### TM 怎么将事务信息是怎么传递

实际上 Seata 基于 Spring 切面, 已经帮我们做了这些事, 我们只需要使用 GlobalTransactional 注解就够了,

那么还有一个问题, 事务信息是怎么传递的呢, TM 怎么将全局事务 XID 传递给 RPC 的提供者的呢? 这部分, 根据 RPC 框架的不同, 需要不同的实现, 但是本质上都是一样的, 拦截 RPC 的调用过程, 在 RPC 请求中加一个隐藏属性来存储 XID, RPC 的提供方从请求中获取到该隐藏属性, 然后存储在事务 Context 的 ThreadLocal 中, 我们以

#### Dubbo 为例

Dubbo 是通过一个 Filter 的概念来表示 AOP 特性的，Filter 的注入基于 Dubbo 的 SPI, 而 Seata 这里所做的就是实现了一个 Dubbo Filter, 把事务 Context 和 RPCContext 中的数据做一下绑定。其他 RPC 框架的支持方案基本类似,

官方demo：

### Seata 管理分布式事务的典型生命周期：

* TM 要求 TC 开始新的全局事务，TC 生成表示全局事务的 XID。
* XID 通过微服务的调用链传播。XID 通过微服务的调用链传播。
* RM 在 TC 中将本地事务注册为 XID 的相应全局事务的分支。RM 在 TC 中将本地事务注册为 XID 的相应全局事务的分支。
* TM 要求 TC 提交或回滚 XID 的相应全局事务。TM 要求 TC 提交或回滚 XID 的相应全局事务。
* TC 驱动 XID 的相应全局事务下的所有分支事务，完成分支提交或回滚。TC 驱动 XID 的相应全局事务下的所有分支事务，完成分支提交或回滚。

### AT事务模式:集团的 TXC==事务模式

★★★：<https://chenjiayang.me/2019/05/02/seata-tcc/>

Seata 是阿里近期开源的分布式事务框架，地址：<https://github.com/seata/seata>。框架包括了集团的 TXC（云版本叫 GTS）, TXC 在 Seata 中又叫 AT 模式意为补偿方法是框架自动生成的，对用户完全屏蔽，用户可以向使用本地事务那样使用分布式事务，缺点是仅支持关系型数据库（目前支持 MySQL）,引入 Seata AT 的服务需要本地建表存储 rollback\_info，隔离级别默认 RU 适用场景有限。

### TCC事务模式:蚂蚁金服的 TCC（Try-Confirm-Cancel）

<https://www.bytesoft.org/tcc-intro/>

蚂蚁金服向 Seata 贡献了自己的 TCC 实现，据说已经演化了十多年，大量应用在在金融、交易、仓储等领域。

#### TCC事务机制简介

关于TCC（Try-Confirm-Cancel）的概念，最早是由Pat Helland于2007年发表的一篇名为《Life beyond Distributed Transactions:an Apostate’s Opinion》的论文提出。在该论文中，TCC还是以Tentative-Confirmation-Cancellation作为名称；正式以Try-Confirm-Cancel作为名称的，可能是Atomikos（Gregor Hohpe所著书籍《Enterprise Integration Patterns》中收录了关于TCC的介绍，提到了Atomikos的Try-Confirm-Cancel，并认为二者是相似的概念）。

国内最早关于TCC的报道，应该是InfoQ上对阿里程立博士的一篇采访。经过程博士的这一次传道之后，TCC在国内逐渐被大家广为了解并接受。相应的实现方案和开源框架也先后被发布出来，ByteTCC就是其中之一。

TCC事务机制相对于传统事务机制（X/Open XA Two-Phase-Commit），其特征在于它不依赖资源管理器(RM)对XA的支持，而是通过对（由业务系统提供的）业务逻辑的调度来实现分布式事务。

对于业务系统中一个特定的业务逻辑S，其对外提供服务时，必须接受一些不确定性，即对业务逻辑执行的一次调用仅是一个临时性操作，调用它的消费方服务M保留了后续的取消权。如果M认为全局事务应该rollback，它会要求取消之前的临时性操作，这将对应S的一个取消操作；而当M认为全局事务应该commit时，它会放弃之前临时性操作的取消权，这对应S的一个确认操作。

每一个初步操作，最终都会被确认或取消。因此，针对一个具体的业务服务，TCC事务机制需要业务系统提供三段业务逻辑：初步操作Try、确认操作Confirm、取消操作Cancel。

#### 初步操作（Try）

TCC事务机制中的业务逻辑（Try），从执行阶段来看，与传统事务机制中业务逻辑相同。但从业务角度来看，却不一样。TCC机制中的Try仅是一个初步操作，它和后续的确认一起才能真正构成一个完整的业务逻辑。可以认为

|  |  |
| --- | --- |
| 1 | [传统事务机制]的业务逻辑 = [TCC事务机制]的初步操作（Try） + [TCC事务机制]的确认逻辑（Confirm）。 |

TCC机制将传统事务机制中的业务逻辑一分为二，拆分后保留的部分即为初步操作（Try）；而分离出的部分即为确认操作（Confirm），被延迟到事务提交阶段执行。  
TCC事务机制以初步操作（Try）为中心的，确认操作（Confirm）和取消操作（Cancel）都是围绕初步操作（Try）而展开。因此，Try阶段中的操作，其保障性是最好的，即使失败，仍然有取消操作（Cancel）可以将其不良影响进行回撤。

#### **确认操作（Confirm）**

确认操作（Confirm）是对初步操作（Try）的一个补充。当TCC事务管理器决定commit全局事务时，就会逐个执行初步操作（Try）指定的确认操作（Confirm），将初步操作（Try）未完成的事项最终完成。

#### 取消操作（Cancel）

取消操作（Cancel）是对初步操作（Try）的一个回撤。当TCC事务管理器决定rollback全局事务时，就会逐个执行初步操作（Try）指定的取消操作（Cancel），将初步操作（Try）已完成的事项全部撤回。

在传统事务机制中，业务逻辑的执行和事务的处理，是在不同的阶段由不同的部件来完成的：业务逻辑部分访问资源实现数据存储，其处理是由业务系统负责；事务处理部分通过协调资源管理器以实现事务管理，其处理由事务管理器来负责。二者没有太多交互的地方，所以，传统事务管理器的事务处理逻辑，仅需要着眼于事务完成（commit/rollback）阶段，而不必关注业务执行阶段。

而在TCC事务机制中的业务逻辑处理和事务处理，其关系就错综复杂：业务逻辑（Try/Confirm/Cancel）阶段涉及所参与资源事务的commit/rollback；全局事务commit/rollback时又涉及到业务逻辑（Try/Confirm/Cancel）的执行。其中关系，本站将另行撰文详细介绍，敬请关注！

#### 常用类源码：

Seata 中 TCC 模式的源码并不复杂，主要集中于：

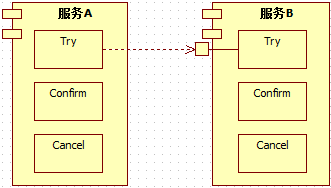
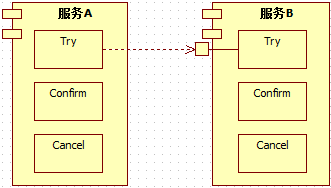
| module | class | 功能 |
| --- | --- | --- |
| seata-spring | GlobalTransactionalInterceptor.class | 全局事务切面逻辑，包括注册全局事务，拿到 xid |
| seata-spring | TccActionInterceptor.class | TCC 参与方切面逻辑 |
| seata-tcc | TCCResourceManager.class | 解析 TCC Bean，保存 TCC Resources，便于后续回调 |
| seata-tcc | ActionInterceptorHandler.class | TCC 分支事务注册实现 |
| seata-server | DefaultCoordinator.class、FileTransactionStoreManager.class | 主要是 TC 的实现、事务存储等实现 |

#### 流程demo：

TCC全局事务必须基于RM本地事务来实现全局事务

TCC服务是由Try/Confirm/Cancel业务构成的，其Try/Confirm/Cancel业务在执行时，会访问资源管理器（Resource Manager，下文简称RM）来存取数据。这些存取操作，必须要参与RM本地事务，以使其更改的数据要么全部commit，要么全部rollback。

这一点不难理解，考虑一下如下场景：



假设图中的服务B没有基于RM本地事务（以RDBS为例，可通过设置auto-commit为true来模拟），那么一旦[B:Try]操作中途执行出错，TCC事务框架后续决定回滚全局事务时，该[B:Cancel]则需要判断[B:Try]中哪些操作已经写到DB、哪些操作还没有写到DB：如果[B:Try]业务有5个写库操作，[B:Cancel]业务则需要逐个判断这5个操作是否生效，并将生效的操作执行反向操作。

不幸的是，由于[B:Cancel]业务也有n（0<=n<=5）个反向的写库操作，此时一旦[B:Cancel]也中途出错，则后续的[B:Cancel]执行任务更加繁重。因为，相比第一次[B:Cancel]操作，后续的[B:Cancel]操作还需要判断先前的[B:Cancel]操作的n（0<=n<=5）个写库中哪几个已经执行、哪几个还没有执行，这就涉及到了幂等性问题。

然而，对幂等性的保障，很可能也需要涉及额外的写库操作，该写库操作又会因为没有RM本地事务的支持而存在类似问题。。。

可想而知，如果不基于RM本地事务，TCC事务框架是无法有效的管理TCC全局事务的

demo：

### 单纯的存储层 2PC

### demo-tcc：

#### Seata TCC 参与方

Seata 中的 TCC 模式要求 TCC 服务的参与方在接口上增加 @TwoPhaseBusinessAction 注解，注明 TCC 接口的名称（全局唯一），TCC 接口的 confirm 和 cancel 方法的名称，用于后续框架反射调用，下面是一个 TCC 接口的案例：

public interface TccAction {

@TwoPhaseBusinessAction(name = "yourTccActionName", commitMethod = "confirm", rollbackMethod = "cancel")

public boolean try(BusinessActionContext businessActionContext, int a, int b);

public boolean confirm(BusinessActionContext businessActionContext);

public boolean cancel(BusinessActionContext businessActionContext);

}

## MongoDB：C++编写

### 为什么用用MongoDb：大批量数据、不好用

### 简介：

由于MongoDB独特的数据处理方式，可以将热点数据加载到内存，故而对查询来讲，会非常快（当然也会非常消耗内存）；

同时由于采用了BSON的方式存储数据，故而对JSON格式数据具有非常好的支持性以及友好的表结构修改性，文档式的存储方式，数据友好可见；

数据库的分片集群负载具有非常好的扩展性以及非常不错的自动故障转移（大赞）。

缺点：

数据库的查询采用了特有的查询方式，有一定的学习成本（不高）；

索引不咋滴；

锁只能提供到collection级别，还做不到行级锁；

没有事务机制（不能回滚啊）；

学习资料肯定没有MySQL的多

如果业务中存在大量复杂的事务逻辑操作，则不要用MongoDB数据库

### MongoDB能为我解决哪些问题

一般来讲，我会将MySQL中的部分表迁移到MongoDB中，主要是涉及到车辆历史轨迹以及温湿度数据等机器采集到的数据，

而订单数据、客户数据等信息，仍然放到MySQL数据库中，主要是因为这两类数据实时采集，实时更新，会随着时间的推移，项目的扩大（PAAS服务），造成非常巨大的数据量，而一般MySQL在单表数据量超过500万后，性能就会下降的比较快，虽然可以通过分表的方式进行处理，但是随着时间的增长，仍然会给我带来比较大的麻烦（如查询等），这样，就不如将其放到MongoDB中存储，查询什么的都会比较方便，不过需要注意根据片键分片哦。

### MongoDB与Hadoop的区别

MongoDB侧重于对数据进行操作的应用系统，而Hadoop则侧重于对数据进行分析统计的应用。   
MongoDB能够满足对数据库读写性能具有极高要求的应用场景（很消耗memory的），一般这些应用的响应延迟会要求控制在10ms以下，甚至更低。而Hadoop由于每一次的读写操作会包含大量数据（Hadoop更适合少次操作大批量数据的场景），通过聚集分析处理大量数据，这种分析一般都会走MapReduce，会造成很高的延迟（数分钟到数小时不等）

### Mongodb与redis相比较：

mongoDB 源码语言是C++,redis也是C或C++,

mongodb 文件存储是BSON格式类似JSON，或自定义的二进制格式。

mongodb与redis性能都很依赖内存的大小，mongodb 有丰富的数据表达、索引；最类似于关系数据库，支持丰富的查询语言，redis数据丰富，较少的IO ，这方面mongodb优势明显。

mongodb不支持事物，靠客户端自身保证，redis支持事物，比较弱，仅能保证事物中的操作按顺序执行，这方面 redis优于mongodb。

mongodb对海量数据的访问效率提升，redis 较小数据量的性能及运算,这方面 mongodb性能优于redis .monbgodb 有mapredurce功能，提供数据分析，redis 没有 ，这方面 mongodb优于redis 。

## ElasterSearch：-JAVA编写

### 为什么要使用Elasticsearch ：分词、搜索

1.响应时间

MySQL

背景：

小刘在做测试时，发现当数据库中的文档数仅仅上万条时，关键词查询就比较慢了。如果一旦到企业级的数据，响应速度就会更加不可接受。

原因：

在数据库做模糊查询时，如LIKE语句，它会遍历整张表，同时进行字符串匹配。

例如，当小刘在数据库查询“市场”时，数据库会在每一条记录去匹配“市场”这两字是否出现。实际上，并不是所有记录都包含“市场”，所以做了很多无用功。

这两个步骤都不高效，而且随着数据量的增大，消耗的资源和时间都会线性的增长。

Elasticsearch

提升：

小刘使用了云搜索服务后，发现这个问题被很好解决，TB级数据在毫秒级就能返回检索结果，很好地解决了痛点。

原因：

而Elasticsearch是基于倒排索引的

### 为什么比mysql快：

为什么 Elasticsearch/Lucene 检索可以比 mysql 快了。Mysql 只有 term dictionary 这一层，是以 b-tree 排序的方式存储在磁盘上的。检索一个 term 需要若干次的 random access 的磁盘操作。而 Lucene 在 term dictionary 的基础上添加了 term index 来加速检索，term index 以树的形式缓存在内存中。从 term index 查到对应的 term dictionary 的 block 位置之后，再去磁盘上找 term，大大减少了磁盘的 random access 次数。

### term dictionary、term index 、postingList

假设我们有很多个 term，比如：

**Carla,Sara,Elin,Ada,Patty,Kate,Selena**

如果按照这样的顺序排列，找出某个特定的 term 一定很慢，因为 term 没有排序，需要全部过滤一遍才能找出特定的 term。排序之后就变成了：

**Ada,Carla,Elin,Kate,Patty,Sara,Selena**

这样我们可以用二分查找的方式，比全遍历更快地找出目标的 term。这个就是 term dictionary。有了 term dictionary 之后，可以用 logN 次磁盘查找得到目标。但是磁盘的随机读操作仍然是非常昂贵的（一次 random access 大概需要 10ms 的时间）。所以尽量少的读磁盘，有必要把一些数据缓存到内存里。但是整个 term dictionary（Mysql 只有 term dictionary 这一层） 本身又太大了，无法完整地放到内存里。于是就有了 term index。term index 有点像一本字典的大的章节表。

在存储的时候，无论父文档还是子文档，对于 Lucene 来说都是文档，都会有文档 Id。但是对于嵌套文档来说，可以保存起子文档和父文档的文档 id 是连续的，而且父文档总是最后一个。有这样一个排序性作为保障，那么有一个所有父文档的 posting list 就可以跟踪所有的父子关系。也可以很容易地在父子文档 id 之间做转换。把父子关系也理解为一个 filter，那么查询时检索的时候不过是又 AND 了另外一个 filter 而已。前面我们已经看到了 Elasticsearch 可以非常高效地处理多 filter 的情况，充分利用底层的索引。

使用了嵌套文档之后，对于 term 的 posting list 只需要保存父文档的 doc id 就可以了，可以比保存所有的数据点的 doc id 要少很多。如果我们可以在一个父文档里塞入 50 个嵌套文档，那么 posting list 可以变成之前的 1/50。

### elsaterSearch和MongoDB的对比：

mongodb的目标是：“取代oracle和db2”（财务总监时上市说的）。和RDBMS是竞争关系。  
2、es的大部分场景是：“一个常见的设置是使用其它数据库作为主要的数据存储，使用 Elasticsearch 做数据检索”（2.X官方文档里说的）。和RDBMS是辅助关系。

1、es偏向于检索、查询、数据分析，适用于OLAP系统。mongodb偏向于大数据规模下的CRUD，适用于对事务要求不强的OLTP系统。

mongodb的分片有hash和range两种方式，es只有hash一种。

es是天生分布式，主副分片自动分配和复制，开箱即用。mongodb的分布式是由“前置查询路由+配置服务+shard集合”，需要手动配置集群服务。

es全文检索有强大的分析器且可以灵活组合，查询时智能匹配。mongodb的全文检索字段个数有限制。

es所有字段自动索引，mongodb的字段需要手动索引。

es非实时有数据丢失窗口。mongodb实时理论上无数据丢失风险。

## Redis缓存穿透、缓存击穿、缓存雪崩区别和解决方案

### Redis为什么好：

redis好是可以提高服务端的性能、满足条件可以去redis取数据；避免与数据库交互；对高访问量；数据量大 有提升；redis 在台服务器的情况下，数据更好的进行存储，所以用到了redis，比memcached好，存储量大，支持的数据类型多。

### 并发问题：

1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。  
2.服务器角度，利用setnx实现锁。

### redis持久化的几种方式

　　1、快照（snapshots）

　　　　缺省情况情况下，Redis把数据快照存放在磁盘上的二进制文件中，文件名为dump.rdb。你可以配置Redis的持久化策略，例如数据集中每N秒钟有超过M次更新，就将数据写入磁盘；或者你可以手工调用命令SAVE或BGSAVE。

工作原理

Redis forks.

子进程开始将数据写到临时RDB文件中。

当子进程完成写RDB文件，用新文件替换老文件。

这种方式可以使Redis使用copy-on-write技术。

　　2、AOF快照模式并不十分健壮，当系统停止，或者无意中Redis被kill掉，最后写入Redis的数据就会丢失。这对某些应用也许不是大问题，但对于要求高可靠性的应用来说，

　　Redis就不是一个合适的选择。  
　　Append-only文件模式是另一种选择。  
　　你可以在配置文件中打开AOF模式

　　3、虚拟内存方式（被废弃

　　4、diskstore方式（被废弃

### Redis 数据类型

String：字符串

Hash：字典

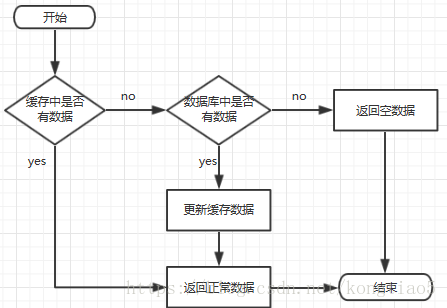
List：链表

Set：集合

Sorted Set：有序集合

### 缓存处理流程

数据库中取，数据库取到更新缓存，并返回结果，数据库也没取到，那直接返回空结果。



   缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

### 缓存穿透

**描述：**缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

**解决方案：**

接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

### 缓存击穿

      描述：

      缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

      解决方案：

设置热点数据永远不过期。

加互斥锁，互斥锁参考代码如下：



   说明：

          1）缓存中有数据，直接走上述代码13行后就返回结果了

         2）缓存中没有数据，第1个进入的线程，获取锁并从数据库去取数据，没释放锁之前，其他并行进入的线程会等待100ms，再重新去缓存取数据。这样就防止都去数据库重复取数据，重复往缓存中更新数据情况出现。

          3）当然这是简化处理，理论上如果能根据key值加锁就更好了，就是线程A从数据库取key1的数据并不妨碍线程B取key2的数据，上面代码明显做不到这点。

### 缓存雪崩

      描述：

      缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，        缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

     解决方案：

缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。

设置热点数据永远不过期。

## Oracle：

### 执行计划-分析表的访问方式

 TABLE ACCESS FULL（全表扫描）

 TABLE ACCESS BY ROWID（通过ROWID的表存取）

 TABLE ACCESS BY INDEX SCAN（索引扫描

### 执行计划-索引扫描方式

Index Unique Scan(唯一扫描)每次 只返回一条记录

情景：unique primary key

Index Range Scan(索引范围扫描)

情景：索引列使用 范围操作符 > <

组合索引，查询部分

非唯一索引列 任何查询

Index Full Scan(索引全扫描)——查询数据必须从索引列 全部直接得到

Index Fast Full Scan(索引快速扫描)——类似Index Full Scan，但是它查询的数据 不进行排序

Index Skip Scan（索引跳跃扫描）情景：复合索引的前导列(索引中第一列) 没有在查询语句中出现

## Mysql：C++编写

### 两种模式：MyISAM、 InnoDB

MyISAM（表级锁 不支持事务，全文索引，只读应用）

.frm / .myd / .myi

非聚集索引

InnoDB（行级锁，最大支持并发，事务，表更新和查询都相当的频繁）

.frm / .ibd

聚集索引，以主键为索引来组织数据

Hash B-tree B+tree？

### 事务隔离机制-为什么是可重复读

是基于binlog复制的！这里不想去搬binlog的概念了，就简单理解为binlog是一个记录数据库更改的文件吧～  
*binlog有几种格式？*  
OK，三种，分别是

* statement:记录的是修改SQL语句
* row：记录的是每行实际数据的变更
* mixed：statement和row模式的混合

那Mysql在5.0这个版本以前，binlog只支持STATEMENT这种格式！而这种格式在**读已提交(Read Commited)**这个隔离级别下主从复制是有bug的，因此Mysql将**可重复读(Repeatable Read)**作为默认的隔离级别！

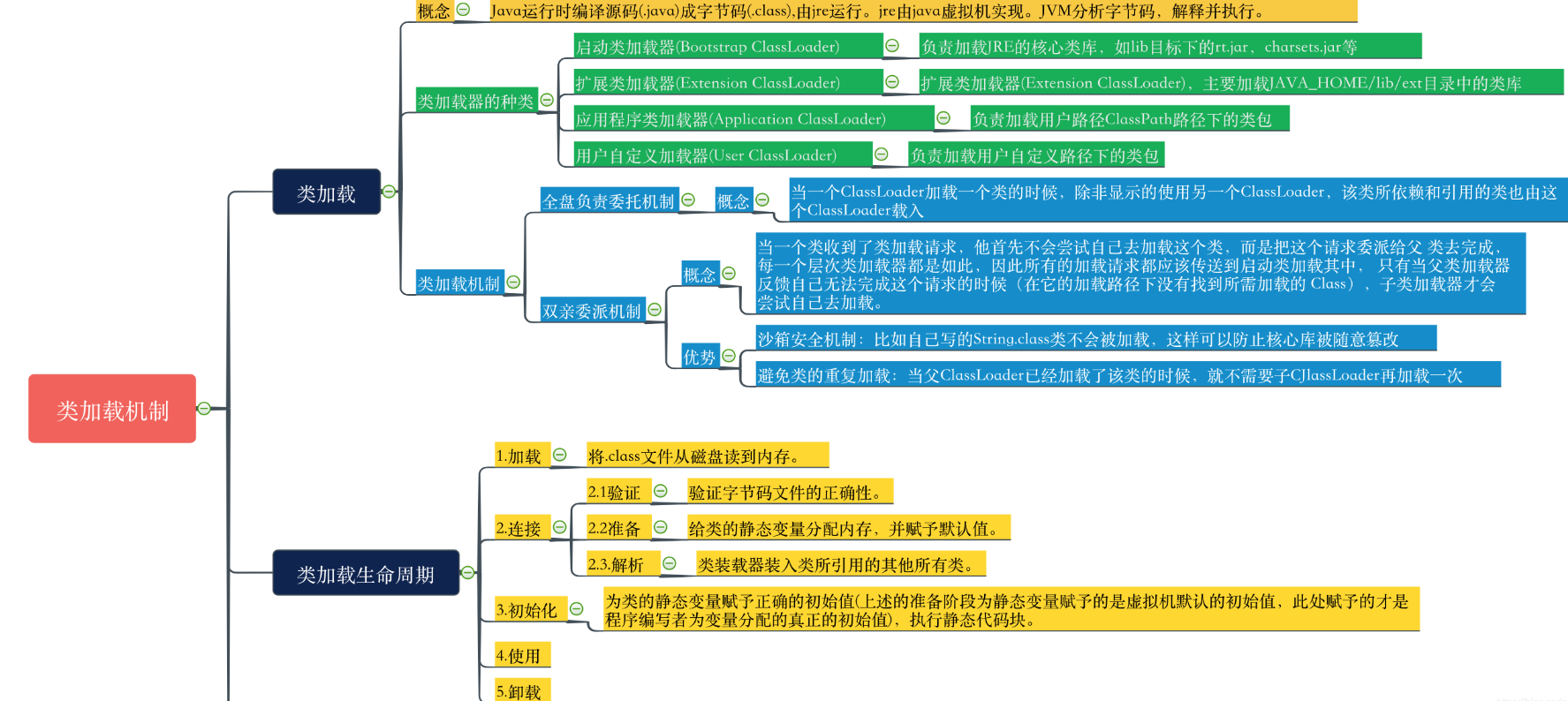
### 为什么在互联网项目为什么将隔离级别设为**读已提交(Read Commited)**！

ok，我们先明白一点！项目中是不用**读未提交(Read UnCommitted)**和**串行化(Serializable)**两个隔离级别，原因有二

* 采用**读未提交(Read UnCommitted)**,一个事务读到另一个事务未提交读数据，这个不用多说吧，从逻辑上都说不过去！
* 采用**串行化(Serializable)**，每个次读操作都会加锁，快照读失效，一般是使用mysql自带分布式事务功能时才使用该隔离级别！(笔者从未用过mysql自带的这个功能，因为这是XA事务，是强一致性事务，性能不佳！互联网的分布式方案，多采用最终一致性的事务解决方案！

# JVM：

**双亲委派=类加载原理：**



## JVM虚拟机：

两个调用方法：System.gc() 或Runtime.getRuntime().gc()

 -Xms / -Xmx — 堆的初始大小 / 堆的最大大小

 -Xmn — 堆中年轻代的大小

 -XX:-DisableExplicitGC — 让System.gc()不产生任何作用

 -XX:+PrintGCDetails — 打印GC的细节

 -XX:+PrintGCDateStamps — 打印GC操作的时间戳

 -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小

 -XX:NewRatio — 可以设置老生代和新生代的比例

 -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布

 -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

 -XX:TargetSurvivorRatio：设置幸存区的目标使用率

### 描述一下JVM加载class文件的原理机制？

答：JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。  
由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。  
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

* Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；
* Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；
* System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中加载类，是用户自定义加载器的默认父加载器。

## 一般使用那些jvm：

jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.8 默认垃圾收集器（帕洛拉-斯凯文具）Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9 默认垃圾收集器G1

## 1：jmap

然能够查看每个类的使用情况为什么不能在生产环境执行；

如果内存120G，执行需要2个小时，

如果在秒杀 等两个小时。。。。

那为什么不晚上执行，晚上的是没有那个并发量

那怎么回答？

测试环境压测执行；

slb复制执行；

影响性能： 内存越大，时间越久

## 2：java VisulVm

在目录 jdk目录下/bin 里面 有执行文件

可以远程执行查看

为什么不能随便拷贝？会使生产环境卡死

为什么生产环境不能直接连接？

连接会导致端口被暴露，引发安全问题

会导致jvm分析 性能下降

什么时候可以拿到文件，内存泄漏直接导致项目挂了

tcp Copy吧生产的请求复制到测试环境，50年薪必备

## 3：arthas阿尔塔斯-boot阿里提供的观察工具

集合了 top jheap heapdump（jmap） 三个软件

thread -b 直接查看死锁

影响性能：10-15%

反编译：为什么代码我自己写的 还要反编译

因为实际的代码库是有多个人来进行合成的。可能A依赖于B的代码，但是版本不对，所以导致本来输出1，一直输出2

# 线程：

## 为什么要多线程：

1.提高运行效率

就如上面提到的，引入多线程可以提高程序运行效率，合理利用cpu资源。但是如果是在单核cpu中，多线程就没有用了吗？

并不是。这就是引入多线程的第二个目的了。

2.防止阻塞

同样如果我们使用单线程执行一个任务，如果碰到I/O操作，后面的任务就要被阻塞，这时候cpu就处于空闲状态。

如果是多线程的情况，就可以各自执行不同的操作，及时I/O阻塞，也不会影响另一个线程中的cpu计算操作。

Springboot：默认200线程 10000连接

## Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?

答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

## 线程的sleep()方法和yield()方法有什么区别？

答：  
① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；  
② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；  
③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；  
④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

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

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

## 请说出与线程同步以及线程调度相关的方法。

答：

* wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
* sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
* notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
* notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

补充：Java 5通过Lock接口提供了显式的锁机制（explicit lock），增强了灵活性以及对线程的协调。Lock接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了newCondition()方法来产生用于线程之间通信的Condition对象；此外，Java 5还提供了信号量机制（semaphore），信号量可以用来限制对某个共享资源进行访问的线程的数量。在对资源进行访问之前，线程必须得到信号量的许可（调用Semaphore对象的acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用Semaphore对象的release()方法）。

## 编写多线程程序有几种实现方式？

答：Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

**synchronized关键字的用法？**

答：synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。在第60题的例子中已经展示了synchronized关键字的用法。

## 举例说明同步和异步。

答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

## 启动一个线程是调用run()还是start()方法？

答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

## 什么是线程池（thread pool）？

ExecutorService线程池

线程池：

假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

一个线程池包括以下四个基本组成部分：

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。  
Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors里面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

* newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
* newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
* newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
* newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

第60题的例子中演示了通过Executors工具类创建线程池并使用线程池执行线程的代码。如果希望在服务器上使用线程池，强烈建议使用newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

## 线程池原理：

https://blog.csdn.net/sihai12345/article/details/82870009

ThreadPoolExecutor执行execute方法有4种情况：

1）如果当前运行的线程少于corePoolSize，则创建新的线程来执行任务。

2）如果运行的线程等于或者多余corePoolSize，则将任务加入BlockingQueue中，在等待队列中，等待有新的线程可以运行。

3）如果BlockingQueue队列满了，且没有超过maxPoolSize，则创建新的线程来处理任务。

4）如果创建的线程超过maxPoolSize，任务会拒绝，并调用RejectExecutionHandler.rejectedExecution()方法。

————————————————

pool.execute(new Runnable(){

@Override

public void run() {

System.out.println("使用execute提交任务.");

}

});

try {

if (executor.awaitTermination(3, TimeUnit.SECONDS)) {

System.out.println("task finished");

} else {

System.out.println("task time out,will terminate");

for (Future<?> f : futures) {

if (!f.isDone()) {

f.cancel(true);

}

}

}

} catch (InterruptedException e) {

System.out.println("executor is interrupted");

} finally {

executor.shutdown();

}

## 线程的基本状态以及状态之间的关系？

说明：其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

## synchronized 原理：

<https://blog.csdn.net/zbuger/article/details/51030772>

<https://www.jianshu.com/p/46d092bb737d>

https://www.jianshu.com/p/afa5296a4832

种内建的synchronized语法：1、synchronized语句；2、synchronized方法。对于synchronized语句当Java源代码被javac编译成bytecode的时候，会在同步块的入口位置和退出位置分别插入monitorenter和monitorexit字节码指令。而synchronized方法则会被翻译成普通的方法调用和返回指令如:invokevirtual、areturn指令，在VM字节码层面并没有任何特别的指令来实现被synchronized修饰的方法，而是在Class文件的方法表中将该方法的access\_flags字段中的synchronized标志位置1，表示该方法是同步方法并使用调用该方法的对象或该方法所属的Class在JVM的内部对象表示Class做为锁对象。

简单来说在JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的Mutex Lock来实现的，但是由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行，这种切换的代价是非常昂贵的；然而在现实中的大部分情况下，同步方法是运行在单线程环境（无锁竞争环境）如果每次都调用Mutex Lock那么将严重的影响程序的性能。不过在jdk1.6中对锁的实现引入了大量的优化，如锁粗化（Lock Coarsening）、锁消除（Lock Elimination）、轻量级锁（Lightweight Locking）、偏向锁（Biased Locking）、适应性自旋（Adaptive Spinning）等技术来减少锁操作的开销。

锁粗化（Lock Coarsening）： 也就是减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁。

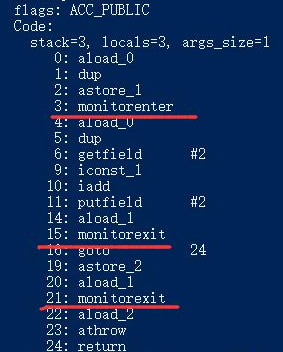
锁消除（Lock Elimination）： 通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本地Stack上进行对象空间的分配（同时还可以减少Heap上的垃圾收集开销）。

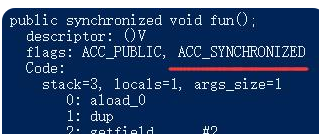
VM中创建对象时会在对象前面加上两个字大小的对象头，在32位机器上一个字为32bit，根据不同的状态位Mark World中存放不同的内容，如上图所示在轻量级锁中，Mark Word被分成两部分，刚开始时LockWord为被设置为HashCode、最低三位表示LockWord所处的状态，初始状态为001表示无锁状态。Klass ptr指向Class字节码在虚拟机内部的对象表示的地址。Fields表示连续的对象实例字段

  Monitor Record是线程私有的数据结构，每一个线程都有一个可用monitor record列表，同时还有一个全局的可用列表；那么这些monitor record有什么用呢？每一个被锁住的对象都会和一个monitor record关联（对象头中的LockWord指向monitor record的起始地址，由于这个地址是8byte对齐的所以LockWord的最低三位可以用来作为状态位），同时monitor record中有一个Owner字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。如下图所示为Monitor Record的内部结构：



MDove：别着急，让我们先看一下\_owner，它指向持有ObjectMonitor对象的线程。当多个线程同时访问一段同步代码时，会先存放到 \_EntryList 集合中，接下来当线程获取到对象的monitor时，就会把\_owner变量设置为当前线程。同时count变量+1。如果线程调用wait() 方法，就会释放当前持有的monitor，那么\_owner变量就会被置为null，同时\_count减1，并且该线程进入 WaitSet集合中，等待下一次被唤醒





可以看到：字节码中并没有monitorenter指令和monitorexit指令，取得代之的是ACC\_SYNCHRONIZED标识，JVM通过ACC\_SYNCHRONIZED标识，就可以知道这是一个需要同步的方法，进而执行上述同步的过程，也就是\_count加1，这些过程。

小A：哦，原来是这样。一个是用了指令，一个是用的标识呀~对了，我听说synchronized的性能特别低是这样么？

**总结：jdk1.6后对锁进行优化，偏向锁（无线程竞争，不调用操作系统函数），重量锁（线程竞争，调用操作系统函数）**

## 简述synchronized 和java.util.concurrent.locks.Lock的异同？

答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

https://blog.csdn.net/sinat\_26279177/article/details/80646301

## 为什么阿里巴巴禁止使用Executors创建线程池?

https://blog.csdn.net/chenssy/article/details/103535589

FixedThreadPool和SingleThreadExecutor => 允许的请求队列长度为Integer.MAX\_VALUE，可能会堆积大量的请求，从而引起OOM异常

CachedThreadPool => 允许创建的线程数为Integer.MAX\_VALUE，可能会创建大量的线程，从而引起OOM异常

## volatile

1：保证线程的可见性，且volatile是轻量级别的锁，无法代替synchronize。

2：禁止指令重重排：

CPU在执行的时候，原来他是一步步顺序执行的，现在是第一个指令执行一半，第二个指令可以开始执行，这种叫流水线式的执行。在这种新的架构的设计的基础之上想充分的利用这一点，那么要求你的编译器吧你的源码编译完指令之后可能进行一个指令的重新排序。

单例中双重检查锁机制，要不要加volatile？

答案是要加的，实际情况中，很难出现，所以不加volatile也不会有问题，不加volatile的问题就会出现指令重排序上。

第一个线程 instance= new Mgr06();经过我们的编译器编译之后，指令分为了三步。

1：给指令申请内存

2：给成员变量初始化

3：把这块内存的内容赋值给instance

既然有这个值了你在另一个线程里头上先去检查，你会发现这个值已经有了，你根本就不会进入锁的那部分代码。？？？？

加了 volatile ，指令重排序不允许不允许存在了，所以在这个时候，一定是保证你初始化完了之后才会赋值给你这个变量

对象作为锁的时候不让他发生改变，加final

cas

## 什么CPU缓存L1、L2、L3

缓存图

L0寄存器

L1cpu内核

L2cpu内核

L3 共享缓存行 多个cpu共享

## 2-3：什么是缓存行，为什么会有缓存行

CPU速度太快， 去内存（主存），cpu执行速度是内存的100倍，所以会在cpu和内存直接加入多种缓存，

为什么是三级：工业化测试定制的

## 4-5-6：缓存行的大小是多少？Disruptor框架如何使用缓存行填充提高效率的。

64直接，1：解析1.7jdk源码中 有点类就有写到；

2：著名的高速dipust消息队列，内部就是构造了7个对象，凑成了8个对象 64直接，速度提升四倍

为什么要凑成7个？

相近的内存 分割，如果是只有一个对象地址，64 一个L1. 会存放两个L1的 X，X发生改变，会通知其他L1所以效率会慢。

5：缓存行如何影响Java编程

有些特殊的类，填充到 8的倍数

## 7：什么是CPU的乱序执行

Cpu执行的时候，会在这个线程，执行一下，那个线程的内容执行一下，因为速度太快，这样做是为了提升cpu的使用效率。

## 8：如何证明CPU的乱序执行？

肯定是乱序的 用java中的内容来看

**new 一个对象，指令解析**

class T{

int m = 8；

}

T t = newT();

申请一块空间 new

初始化成员变量 dup double设置为0.0 int 为0之类

进行赋值 构造方法 invokespecial

astore—1 建立关联 t=8

return

## 9：JVM如何用内存屏障禁止重排序

hotsopt 偷懒 定义了，读写读， 底层是lock。addl

volatile内部也是使用(汇编翻译) lock.addl来实现内存屏障，因为cpu去内存读只有一条通道，如果使用了 lock.add

## 10：DCL单例，双重检查单列，不禁止重排序会造成什么严重的后果

前面知道是cpu是可以乱序执行的，我们new一个对象，可以分为五个指令，这个五个指令 3赋值4关联是可以支持乱序的，如果是不用volatile进行隔离，多个线程的访问这个 单列的时候会拿到不同的半成品初始化对象，虽然结果可能是一样，但是内部的code已经不一样。那就表示这两个对象不是同一对象，违反了对象

## cas 无锁优化，自旋

## 线程顺序：

<https://www.cnblogs.com/myseries/p/11575757.html>

## AQS：

20200819-AQS源码解析.mq

<https://blog.csdn.net/mulinsen77/article/details/84583716>

<https://zhuanlan.zhihu.com/p/151270503>

超时如何控制：

https://blog.csdn.net/wonking666/article/details/76552019

# 源码：

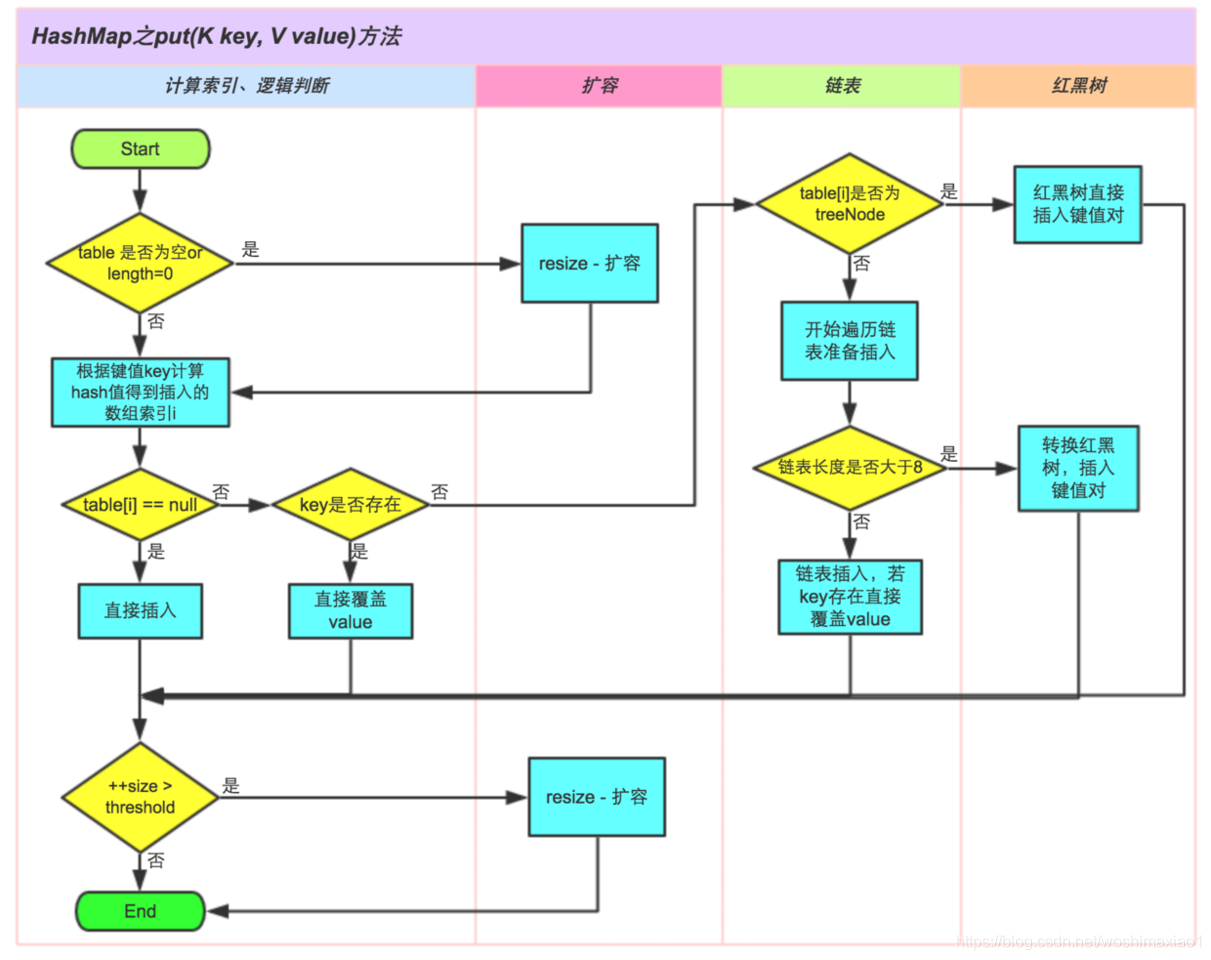
## 集合原理：

### HASHMAP：

<https://blog.csdn.net/woshimaxiao1/article/details/83661464>

1. **什么是哈希表**
2. **哈希冲突**
3. HashMap的实现原理
4. 为何HashMap的数组长度一定是2的次幂？
5. 重写equals方法需同时重写hashCode方法
6. JDK1.8中HashMap的性能优化

get方法的实现相对简单，key(hashcode)–>hash–>indexFor–>最终索引位置，找到对应位置table[i]，再查看是否有链表，遍历链表，通过key的equals方法比对查找对应的记录。



### ArrayList和linkedList的区别

**Array（数组）是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。**

Array获取数据的时间复杂度是O(1),但是要删除数据却是开销很大，因为这需要重排数组中的所有数据, (因为删除数据以后, 需要把后面所有的数据前移)

* 时间复杂度

| **操作** | **数组** | **链表** |
| --- | --- | --- |
| 随机访问 | O(1) | O(N) |
| 头部插入 | O(N) | O(1) |
| 头部删除 | O(N) | O(1) |
| 尾部插入 | O(1) | O(1) |
| 尾部删除 | O(1) | O(1) |

同样查找, 时间复杂度都是O(N), 但是数组要比链表快

因为数组的连续内存, 会有一部分或者全部数据一起进入到CPU缓存, 而链表还需要在去内存中根据上下游标查找, CPU缓存比内存块太多

数据大小固定, 不适合动态存储, 动态添加, 内存为一连续的地址, 可随机访问, 查询速度快

链表代销可变, 扩展性强, 只能顺着指针的方向查询, 速度较慢

### 阐述ArrayList、Vector、LinkedList的存储性能和 特性。

**ArrayList读慢 不安全，LinkedList读慢插快不安全 vector老人家**

答：ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。Vector属于遗留容器（Java早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已经不推荐使用，但是由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现

### 为什么阿里巴巴建议初始化HashMap的容量大小? 合理利用空间 防止扩容

Java开发手册建议创建HashMap时设置初始化容量， 但是多少合适呢? 0.75 总/0.75+1

>首先回忆HashMap的数据结构，我们都知道有序数组存储数据，对数据的索引效率都很高，但是插入和删除就会有性能瓶颈（回忆ArrayList），

链表存储数据，要一次比较元素来检索出数据，所以索引效率低，但是插入和删除效率高（回忆LinkedList），两者取长补短就产生了哈希散列这种存储方式，

也就是HashMap的存储逻辑.

而负载因子表示一个散列表的空间的使用程度，有这样一个公式：initailCapacity\*loadFactor=HashMap的容量。所以负载因子越大则散列表的装填程度越高，

也就是能容纳更多的元素，元素多了，链表大了，所以此时索引效率就会降低。

反之，负载因子越小则链表中的数据量就越稀疏，此时会对空间造成烂费，但是此时索引效率高

hashmap源码中，我们知道hashmap不指定大小的时候，初始容量 capacity值为16。负载因子为0.75，当我们put操作的时候，会先看下容量是否达到capacity\*3/4的扩容条件。而每次扩容都会进行rehash。每次的 rehash都是需要重建哈希表，很费时间和性能。当我们在已知集合大小的情况下，初始化容量大小那就不会进行rehash，性能提升。 反之就容量就默认就好了</p>

当我们初始化定义容量大小的时候，大小定义多少合适了呢？

就定义为集合大小吗？当然不行，如果我们定义为集合大小，还是可能出现扩容，rehash。所以我们需要计算一下，大小多少合适，合理，减少扩容，避免rehash。

initCapacity=已知需要存储元素大小/负载因子+1。

为什么加1呢？

因为每次扩容的条件是存储元素每次达到capacity\*3/4就扩容，减一当然就不会扩容了。再加一个就要扩容了

## Spring源码:

### Bean实例生命周期如下：

Spring对bean进行实例化，默认bean是单例；

Spring对bean进行依赖注入；

如果bean实现了BeanNameAware接口，spring将bean的id传给setBeanName()方法；

如果bean实现了BeanFactoryAware接口，spring将调用setBeanFactory方法，将BeanFactory实例传进来；

如果bean实现了ApplicationContextAware接口，它的setApplicationContext()方法将被调用，将应用上下文的引用传入到bean中；

如果bean实现了BeanPostProcessor接口，它的postProcessBeforeInitialization方法将被调用；

如果bean实现了InitializingBean接口，spring将调用它的afterPropertiesSet接口方法，类似的如果bean使用了init-method属性声明了初始化方法，该方法也会被调用；

如果bean实现了BeanPostProcessor接口，它的postProcessAfterInitialization接口方法将被调用；

此时bean已经准备就绪，可以被应用程序使用了，他们将一直驻留在应用上下文中，直到该应用上下文被销毁；

若bean实现了DisposableBean接口，spring将调用它的distroy()接口方法。同样的，如果bean使用了destroy-method属性声明了销毁方法，则该方法被调用；

### Srping单例如何解决互相依赖

circulationB使用setter依赖circulationA, 因此createBeanInstance使用默认的空参数构造实例化, 完成之后加入三级缓存并在populateBean中属性进行初始化, 此时需要实例化circulationA. 当使用构造函数创建circulationA需要依赖circulationB, 则也会暂时停下来并去创建circulationB, 由于在缓存中拿到circulationB即完成circulationA实例化. 再次返回circulationB的populateBean方法. 此时circulationA 和 circulationB 加载完成. 由此可以类推, 如果只是通过属性 或者 setter方法进行循环依赖 spring可以完美解决.

field属性注入循环依赖

@Service

public class A1 {

@Autowired

private B1 b1;

}

@Service

public class B1 {

@Autowired

public A1 a;

}

### 分析单例bean创建流程:

由refresh()为入口切入

1)、AbstractBeanFactory.getBean为入口 并委托 AbstractBeanFactory.doGetBean创建  
2)、AbstractBeanFactory.doGetBean 会首先从AbstractBeanFactory.getSingleton中获取缓存的bean对象, 如果不存在则调用抽象方法createBean, 即子类实现的AbstractAutowireCapableBeanFactory.createBean方法  
3)、AbstractAutowireCapableBeanFactory.createBean方法触发doCreateBean依次调用以下方法实现bean创建过程

createBeanInstance: 实例化bean, 如果需要依赖其他对象则首先创建其他对象(发生循环依赖的地方)

addSingletonFactory: 将实例化bean加入三级缓存

populateBean: 初始化bean, 如果需要依赖其他对象则首先创建其他对象(发生循环依赖的地方) –读取autuweire的map

initializeBean

registerDisposableBeanIfNecessary

4)、AbstractAutowireCapableBeanFactory.autowireConstructor使用构造函数进行实例化

5)、最终调用ConstructorResolver.autowireConstructor 和 ConstructorResolver.resolveConstructorArguments 进行实例化已经解析构造参数

6)、调用BeanDefinitionValueResolver.resolveValueIfNecessary 和 BeanDefinitionValueResolver.resolveReference 模版类解析构造参数

### SpringBoot 如何实现自动注入

**@EnableAutoConfiguration**

AutoConfigurationImportSelector实现了ImportSelector

这里就会扫描具有MEAT-INF/spring.factories文件的jar包，得到所有的配置类：

在spring-web-xxx-release.jar包中有一个已经实现了ServletContextListener接口的类ContextLoader

this.contextLoader.initWebApplicationContext(event.getServletContext())方

### SpringAOP加强bean的设计模式？

<https://juejin.cn/post/6844903591526793223>

注解解析器：AutowiredAnnotationBeanPostProcessor

Spring容器启动时，AutowiredAnnotationBeanPostProcessor被注册到容器；

扫描代码，如果带有@Autowired注解，则将依赖注入信息封装到InjectionMetadata中（见扫描过程）；

创建bean时（实例化对象和初始化），会调用各种BeanPostProcessor对bean初始化，AutowiredAnnotationBeanPostProcessor负责将相关的依赖注入进来；

refresh方法中registerBeanPostProcessors(beanFactory)完成了对AutowiredAnnotationBeanPostProcessor的注册，当执行finishBeanFactoryInitialization(beanFactory)方法对非延迟初始化的单例bean进行初始化时，会执行到AbstractAutowireCapableBeanFactory类的doCreateBean方法，在这个方法中有如下这么一段代码。

同样，在doCreateBean方法中执行populateBean方法实现对属性的注入。

### Aop动态代理有哪几种方式：

有jdk代理和 cglib两种，我们一般开发都是面向接口开发，也符合jdk只能针对实现了接口的生成代理，jdk动态代理就是拦截器invocationHanlder+反射机制来代理接口代理，

和cglib在以前版版本的jdk里面的确会比 jdk代理会快点，现在随着 1.678 的优化，速度已经比cglib快了，而且cglib代理默认是针对类的 生成了一个子类，覆盖其中方法，采用的是继承，这样导致有的final类或方法，是无法继承的。

aop的cglib动态代理实现机制？

1. \* Jdk代理生成的代理类只有一个，因而其编译速度是非常快的；而由于被代理的目标类是动态传入代理类中的，Jdk代理的执行效率相对来说低一点，这也是Jdk代理被称为动态代理的原因；
2. \* Cglib代理需要为每个目标类生成相应的子类，因而在实际运行过程中，其可能会生成非常多的子类，过多的子类始终不是太好的，因为这影响了虚拟机编译类的效率；但由于在调用过程中，代理类的方法是已经静态编译生成了的，因而Cglib代理的执行效率相对来说高一些。
3. 自定义注解实现方法：SupportedAnnotationTypes

<https://www.jianshu.com/p/46d092bb737d>

JDK动态代理

利用拦截器(拦截器必须实现InvocationHanlder)加上反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

2、CGLiB动态代理

利用ASM开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

如果目标对象没有实现了接口，必须采用CGLIB库，Spring会自动在JDK动态代理和CGLIB之间转换。

JDK动态代理只能对实现了接口的类生成代理，而不能针对类。

2）CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法，并覆盖其中方法实现增强，但是因为采用的是继承，所以该类或方法最好不要声明成final，  
对于final类或方法，是无法继承的。

### CGlib比JDK快？

1）使用CGLib实现动态代理，CGLib底层采用ASM字节码生成框架，使用字节码技术生成代理类，在jdk6之前比使用Java反射效率要高。唯一需要注意的是，CGLib不能对声明为final的方法进行代理，因为CGLib原理是动态生成被代理类的子类。

2）在jdk6、jdk7、jdk8逐步对JDK动态代理优化之后，在调用次数较少的情况下，JDK代理效率高于CGLIB代理效率，只有当进行大量调用的时候，jdk6和jdk7比CGLIB代理效率低一点，但是到jdk8的时候，jdk代理效率高于CGLIB代理，总之，每一次jdk[版本升级](https://www.baidu.com/s?wd=%E7%89%88%E6%9C%AC%E5%8D%87%E7%BA%A7&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)，jdk代理效率都得到提升，而CGLIB代理消息确有点跟不上步伐。

当Bean实现接口时，Spring就会用JDK的动态代理。

2）当Bean没有实现接口时，Spring使用CGlib是实现。

JDK环境就可以进行代理，使用条件

）实现InvocationHandler

2）使用Proxy.newProxyInstance产生代理对象

3）被代理的对象必须要实现接口

# 中间件：

## 为什么用mq？

A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃…A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统压根儿不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。

就是一个系统或者一个模块，调用了多个系统或者模块，互相之间的调用很复杂，维护起来很麻烦。但是其实这个调用是不需要直接同步调用接口的，如果用 MQ 给它异步化解耦。

异步：A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求。如果使用 MQ，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms。

Ribitmq：用的人最多 不方便维护 erlang开发

四种模式：direct, topic ,Headers和fanout。fanout就是广播模式

RocketMQ：阿里开发，java编写，吞吐量高

# 框架：

## 什么是DAO模式？

答：DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

## ★★★事务的ACID是指什么？

答：

* 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；
* 一致性(Consistent)：事务结束后系统状态是一致的；
* 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；
* 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

补充：关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失更新）。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

不可重复读（Unrepeatable Read）：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，B事务执行回滚，那么A读取到的数据就是脏数据。

幻读（Phantom Read）：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

第1类丢失更新：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

| **时间** | **取款事务A** | **转账事务B** |
| --- | --- | --- |
| T1 | 开始事务 |  |
| T2 |  | 开始事务 |
| T3 | 查询账户余额为1000元 |  |
| T4 |  | 查询账户余额为1000元 |
| T5 |  | 汇入100元修改余额为1100元 |
| T6 |  | 提交事务 |
| T7 | 取出100元将余额修改为900元 |  |
| T8 | 撤销事务 |  |
| T9 | 余额恢复为1000元（丢失更新） |  |

第2类丢失更新：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元将余额修改为900元 |
| T6 |  | 提交事务 |
| T7 | 汇入100元将余额修改为1100元 |  |
| T8 | 提交事务 |  |
| T9 | 查询账户余额为1100元（丢失更新） |  |

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。  
直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

| **隔离级别** | **脏读** | **不可重复读** | **幻读** | **第一类丢失更新** | **第二类丢失更新** |
| --- | --- | --- | --- | --- | --- |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITTED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

Oracle不支持脏读，Read Committed默认

Mysql 默认repeatable read 可重复读

Oracle中的隔离级别及实现机制：

Oracle数据库支持READ COMMITTED 和 SERIALIZABLE这两种事务隔离级别。所以Oracle不支持脏读，即Oracle中不允许一个会话读取其他事务未提交的数据修改结果，从而防止了由于事务回滚发生的读取不正确。

Oracle回滚段，在修改数据记录时，会把这些记录被修改之前的结果存入回滚段或撤销段中。Oracle读取操作不会阻碍更新操作，更新操作也不会阻碍读取操作，这样在Oracle中的各种隔离级别下，读取操作都不会等待更新事务结束，更新操作也不会因为另一个事务中的读取操作而发生等待，这也是Oracle事务处理的一个优势所在。

## spring如何控制事务：

datasouretransationManager或者hibernatetr...

## 为什么阿里巴巴禁止工程师直接使用日志系统(Log4j、Log back) 中的API？

https://blog.csdn.net/hollis\_chuang/article/details/84560482

前面提到过一个重要的原因，就是为了在应用中屏蔽掉底层日志框架的具体实现。这样的话，

即使有一天要更换代码的日志框架，只需要修改jar包，最多再改改日志输出相关的配置文件就可以了。这就是解除了应用和日志框架之间的耦合。

# 问到的：

Http2.0的特点

## Session安全：如何共享

两个服务器如何保证安全：白名单或者内外；或者证书加密；

## Cookie确保安全：

改头文件；签名验证；老jar包是否安全；

## HashMap和Hashtable的区别

hashMap的底层结构：HashMap和Hashtable的底层实现都是数组+链表结构实现的

两者最主要的区别在于Hashtable是线程安全，而HashMap则非线程安全

Hashtable的实现方法里面都添加了synchronized关键字来确保线程同步，因此相对而言HashMap性能会高一些，我们平时使用时若无特殊需求建议使用HashMap，在多线程环境下若使用HashMap需要使用Collections.synchronizedMap()方法来获取一个线程安全的集合（Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理）

HashMap可以使用null作为key，而Hashtable则不允许null作为key

虽说HashMap支持null值作为key，不过建议还是尽量避免这样使用，因为一旦不小心使用了，若因此引发一些问题，排查起来很是费事

HashMap以null作为key时，总是存储在table数组的第一个节点上

HashMap是对Map接口的实现，HashTable实现了Map接口和Dictionary抽象类

HashMap的初始容量为16，Hashtable初始容量为11，两者的填充因子默认都是0.75

HashMap扩容时是当前容量翻倍即:capacity\*2，Hashtable扩容时是容量翻倍+1即:capacity\*2+1

两者计算hash的方法不同

Hashtable计算hash是直接使用key的hashcode对table数组的长度直接进行取模

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

HashMap计算hash对key的hashcode进行了二次hash，以获得更好的散列值，然后对table数组长度取摸

看成Hash 和Map

Hash散列将一个任意长度通过某种（hash函数算法：java中是移位）转化成一个固定值，然后通过固定值定位到map上，然后通过value存储到这个mao中

Hashmap table：数据结构：数组+链表

### 1.8JDK下Hashmap有什么区别：

JDK 1.8对HashMap进行了比较大的优化，底层实现由之前的“数组+链表”改为“数组+链表+红黑树”，本文就HashMap的几个常用的重要方法和JDK 1.8之前的死循环问题展开学习讨论。JDK 1.8的HashMap的数据结构如下图所示，当链表节点较少时仍然是以链表存在，当链表节点较多时（大于8）会转为红黑树。

### **伸缩性的不足：**

每当hashmap扩容的时候需要重新去 add entry 对象，需要重新Hash，然后放入我们新的entry table 数组里面，如果你们的工作中你知道你hashmap 需要存多少值，几千或者几万的时候，最好就是先制定他们的扩容大学，防止在put 的时候再次进行扩荣很多次

## SpringMVC与Struts2的主要区别？

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

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

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

## Ioc和aop

Ioc里面的设计模式是单利模式观察者模式、模板方法模式，Aop 的切面是根据动态代理来实现的。

依赖注入：从客户端获得依赖关系的控制不再存在于这些类中。它存由底层的注入器 / DI 框架来处理。

观察者模式：当主体发生变化时，控制从观察者传递到主体。

模板方法模式：控制发生在定义模板方法的基类中，而不是实现算法步骤的子类中。

◆控制反转——Spring通过一种称作控制反转IoC的技术促进了松耦合。当应用了IoC一个对象依赖的其它对象会通过被动的方式传递进来而不是这个对象自己创建或者查找依赖

对象。你可以认为IoC与JNDI相反——不是对象从容器中查找依赖而是容器在对象初始化

时不等对象请求就主动将依赖传递给它。

◆面向切面——Spring提供了面向切面编程的丰富支持允许通过分离应用的业务逻辑与系统

级服务例如审计auditing和事务管理进行内聚性的开发。应用对象只实现它们应

该做的——完成业务逻辑——仅此而已。它们并不负责甚至是意识其它的系统级关注点例

如日志或事务支持。

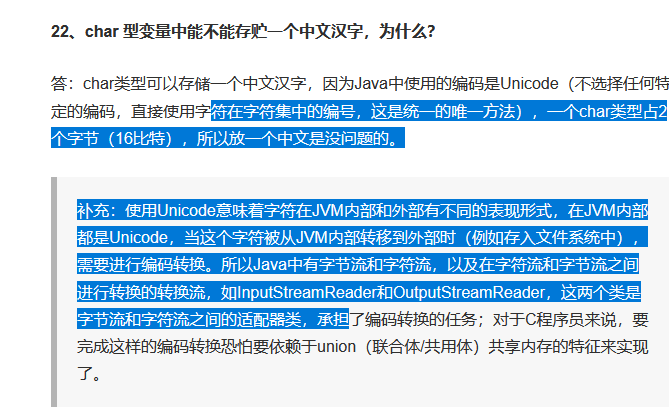
## 华为的面试题- "为什么不能根据返回类型来区分重载"，快说出你的答案吧！

因为调用时不能指定类型信息，编译器不知道你要调用哪个函数。  
例如：

float max(int a, int b);

int max(int a, int b);

当调用max(1, 2);时无法确定调用的是哪个，单从这一点上来说，仅返回值类型不同的重载是不应该允许的。



## 断言assert：

开发和测试时开启，断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。断言的使用如下面的代码所示：

## Java中如何实现序列化，有什么意义？

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。  
要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第29题）

## Java代码列出一个目录下所有的文件？

import java.io.File;

class Test12 {

public static void main(String[] args) {

File f = new File("/Users/nnngu/Downloads");

for(File temp : f.listFiles()) {

if(temp.isFile())System.out.println(temp.getName());

}

}

}

## 行级锁-表级锁：

1、行级锁(tx锁，也叫事务锁)在操作DML（create，insert，update）语句时，oracle会自动加上行级锁，在select \* from table for update 【of column】【nowait|wait 3】时，oracle也会自动加锁

    建议（1）一般在for update 时加nowait，这样就不用等待其他事务执行了，一判断有事务，立马抛出错误。

    建议（2）一般的of column都是在多表操作的时候，能锁定其中一个表的某些列，如果还是同一张表的话，并不能锁定某些列

2、表级锁（tm锁）

*表级锁包含了行级锁，（1），（2）也是通过行级锁来达到表级锁的，他俩没有严格的分类，表级锁是oracle直接扫整个表来判断事务了，而不用去扫描表里的每一行去判断事务了，这个在一些时候是非常有用的*

表级锁：

页级:引擎 BDB。

表级:引擎 MyISAM ， 理解为锁住整个表，可以同时读，写不行

行级:引擎 INNODB ， 单独的一行记录加锁

表级，直接锁定整张表，在你锁定期间，其它进程无法对该表进行写操作。如果你是写锁，则其它进程则读也不允许

行级,，仅对指定的记录进行加锁，这样其它进程还是可以对同一个表中的其它记录进行操作。

页级，表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。

## 索引-主键索引和唯一索引有什么区别：

**唯一索引**  
唯一索引不允许两行具有相同的索引值。  
如果现有数据中存在重复的键值，则大多数数据库都不允许将新创建的唯一索引与表一起保存。当新数据将使表中的键值重复时，数据库也拒绝接受此数据。例如，如果在 employee 表中的职员姓氏(lname) 列上创建了唯一索引，则所有职员不能同姓。

**主键索引**  
主键索引是唯一索引的特殊类型。  
数据库表通常有一列或列组合，其值用来唯一标识表中的每一行。该列称为表的主键。  
在数据库关系图中为表定义一个主键将自动创建主键索引，主键索引是唯一索引的特殊类型。主键索引要求主键中的每个值是唯一的。当在查询中使用主键索引时，它还允许快速访问数据。

它们的一些比较：  
(1)对于主健/unique constraint ， oracle/sql server/mysql等都会自动建立唯一索引；  
(2)主键不一定只包含一个字段，所以如果你在主键的其中一个字段建唯一索引还是必要的；  
(3)主健可作外健，唯一索引不可；  
(4)主健不可为空，唯一索引可；  
(5)主健也可是多个字段的组合；  
(6)主键与唯一索引不同的是：  
a.有not null属性；  
b.每个表只能有一个。

## Equal重写时候为什么要重写HashCode：

好了，到此hashCode()该介绍的我们都说了，还有一点要说的如果我们提供的是一个数值类型的变量的话，那么我们可以调用Arrays.hashCode()来计算它的散列码，这个散列码是由数组元素的散列码组成的。接下来我们回归到我们之前的问题，重写equals方法时也必须重写hashCode方法。在Java API文档中关于hashCode方法有以下几点规定（原文来自java深入解析一书）。

在java应用程序执行期间，如果在equals方法比较中所用的信息没有被修改，那么在同一个对象上多次调用hashCode方法时必须一致地返回相同的整数。如果多次执行同一个应用时，不要求该整数必须相同。

如果两个对象通过调用equals方法是相等的，那么这两个对象调用hashCode方法必须返回相同的整数。

如果两个对象通过调用equals方法是不相等的，不要求这两个对象调用hashCode方法必须返回不同的整数。但是程序员应该意识到对不同的对象产生不同的hash值可以提供哈希表的性能。

通过前面的分析，我们知道在Object类中，hashCode方法是通过Object对象的地址计算出来的，因为Object对象只与自身相等，所以同一个对象的地址总是相等的，计算取得的哈希码也必然相等，对于不同的对象，由于地址不同，所获取的哈希码自然也不会相等。因此到这里我们就明白了，如果一个类重写了equals方法，但没有重写hashCode方法，将会直接违法了第2条规定，这样的话，如果我们通过映射表(Map接口)操作相关对象时，就无法达到我们预期想要的效果。如果大家不相信, 可以看看下面的例子（来自java深入解析一书）

## 在进行数据库编程时，连接池有什么作用？

答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在Java开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：C3P0、Proxool、DBCP、BoneCP、Druid等。

补充：在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

## NIO和IO的区别：

NIO即New IO，这个库是在JDK1.4中才引入的。NIO和IO有相同的作用和目的，但实现方式不同，NIO主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

下表总结了Java IO和NIO之间的主要区别：

|  |  |
| --- | --- |
| **IO** | **NIO** |
| 面向流 | 面向缓冲 |
| 阻塞IO | 非阻塞IO |
| 无 | 选择器 |

BIO：一个连接一个线程，客户端有连接请求时服务器端就需要启动一个线程进行处理。线程开销大。

伪异步IO：将请求连接放入线程池，一对多，但线程还是很宝贵的资源。

NIO：一个请求一个线程，但客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO：一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，

BIO是面向流的，NIO是面向缓冲区的；BIO的各种流是阻塞的。而NIO是非阻塞的；BIO的Stream是单向的，而NIO的channel是双向的。

NIO的特点：事件驱动模型、单线程处理多任务、非阻塞I/O，I/O读写不再阻塞，而是返回0、基于block的传输比基于流的传输更高效、更高级的IO函数zero-copy、IO多路复用大大提高了Java网络应用的可伸缩性和实用性。基于Reactor线程模型。

在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生，事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。如在Reactor中实现读：注册读就绪事件和相应的事件处理器、事件分发器等待事件、事件到来，激活分发器，分发器调用事件对应的处理器、事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

## 7.NIO的组成？

Buffer：与Channel进行交互，数据是从Channel读入缓冲区，从缓冲区写入Channel中的

flip方法 ： 反转此缓冲区，将position给limit，然后将position置为0，其实就是切换读写模式

clear方法 ：清除此缓冲区，将position置为0，把capacity的值给limit。

rewind方法 ： 重绕此缓冲区，将position置为0

DirectByteBuffer可减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，不可控，通常会用内存池来提高性能。直接缓冲区主要分配给那些易受基础系统的本机I/O 操作影响的大型、持久的缓冲区。如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer，由JVM进行管理。

Channel：表示 IO 源与目标打开的连接，是双向的，但不能直接访问数据，只能与Buffer 进行交互。通过源码可知，FileChannel的read方法和write方法都导致数据复制了两次！

Selector可使一个单独的线程管理多个Channel，open方法可创建Selector，register方法向多路复用器器注册通道，可以监听的事件类型：读、写、连接、accept。注册事件后会产生一个SelectionKey：它表示SelectableChannel 和Selector 之间的注册关系，wakeup方法：使尚未返回的第一个选择操作立即返回，唤醒的

原因是：注册了新的channel或者事件；channel关闭，取消注册；优先级更高的事件触发（如定时器事件），希望及时处理。

Selector在Linux的实现类是EPollSelectorImpl，委托给EPollArrayWrapper实现，其中三个native方法是对epoll的封装，而EPollSelectorImpl. implRegister方法，通过调用epoll\_ctl向epoll实例中注册事件，还将注册的文件描述符(fd)与SelectionKey的对应关系添加到fdToKey中，这个map维护了文件描述符与SelectionKey的映射。

fdToKey有时会变得非常大，因为注册到Selector上的Channel非常多（百万连接）；过期或失效的Channel没有及时关闭。fdToKey总是串行读取的，而读取是在select方法中进行的，该方法是非线程安全的。

Pipe：两个线程之间的单向数据连接，数据会被写到sink通道，从source通道读取

NIO的服务端建立过程：Selector.open()：打开一个Selector；ServerSocketChannel.open()：创建服务端的Channel；bind()：绑定到某个端口上。并配置非阻塞模式；register()：注册Channel和关注的事件到Selector上；select()轮询拿到已经就绪的事件

## RabbitMq由什么东西组成：

它的优势是什么？  
异步有两个优势：  
1）对用户来说，比同步时更加快捷，用户体验非常好。（让用户以为自己抢到了，欺骗ing   ）  
2）对系统访问压力来说，异步因为没有没有真正执行，不会造成某时刻对系统的访问压力剧增。而是放入队列，

## rinbbon和fegin的协议是什么协议：

## 基于token认证-JSON Web Token-jwt

token原理：

使用用户名和密码请求登录接口

登录接口验证用户名和密码

登录接口生成一个uuid作为token，将用户信息作为值，然后保存到redis缓存中jedis.set(token, user);

登录接口返回用户信息和token

浏览器将token保存到本地

当请求其它接口时就携带token值

接口根据token去缓存中查，如果找到了就调用接口，如果找不到报token错误(一般通过拦截器来实现检查)

## 高并发解决方案总结

ali：1：缓存 2负载 3机器X2 5前后端分离

1缓存 ~~2集群~~ 3负载 ~~4cdn~~ 5前后端 ~~6分库分表~~ 7 读写分离

1.使用缓存技术(常用)

如Redis

2.负载均衡，比如Nginx、SLB、F5、K8s

3.集群

4.CDN加速进行内容分发（不建议）

5.静态资源动静分离，比如Nginx

6.数据库读写分离

7.分表分库

## 什么是分布式？

分布式更多的一个概念，**是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段**。该领域需要解决的问题极多，在不同的技术层面上，又包括：

、分布式在不同的技术层面上，包括以下内容：

分布式文件系统

分布式缓存

分布式数据库

分布式计算等

一些名词如Hadoop、zookeeper、MQ等都跟分布式有关。从理念上讲，分布式的实现有两种形式：

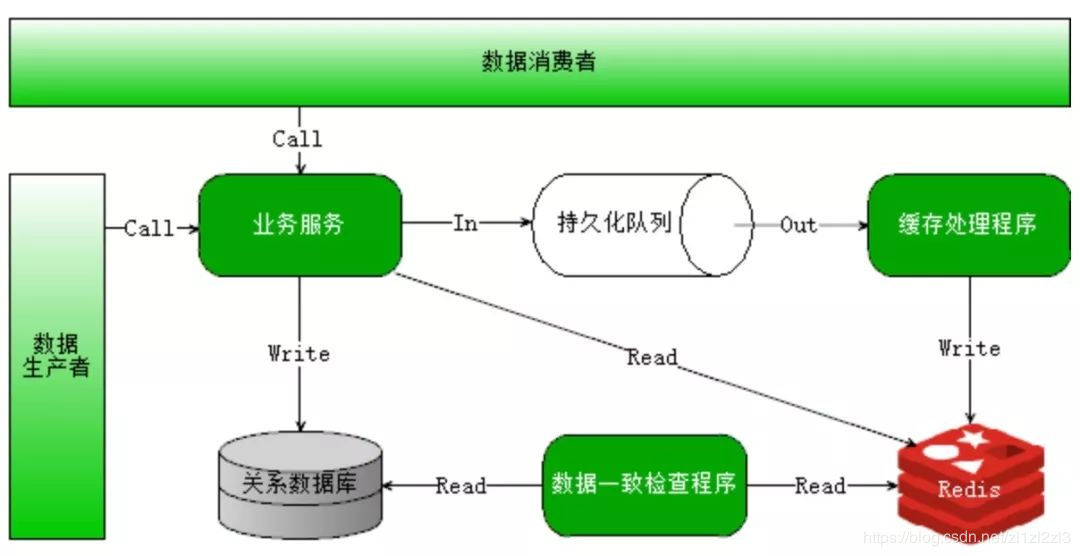
**水平扩展：**当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供相当的服务；

**垂直拆分：**前端有多种查询需求时，一台机器扛不住，可以将不同的需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

## 什么是高并发？

**相对于分布式来讲，高并发在解决的问题上会集中一些，其反应的是同时有多少量**：比如在线直播服务，同时有上万人观看。

高并发可以通过分布式技术去解决，将并发流量分不到不同的物理服务器上。但除此之外，还可以有很多其他优化手段：比如使用缓存系统，将所有的，静态内容放到CDN等；还可以使用多线程技术将一台服务器的服务能力最大化。



## 什么是多线程？

**多线程是指从软件或者硬件上实现多个线程并发执行的技术**，它更多的是解决CPU调度多个进程的问题，从而让这些进程看上去是同时执行（实际是交替运行的）。

这几个概念中，**多线程解决的问题是最明确的，手段也是比较单一的，基本上遇到的最大问题就是线程安全**。在JAVA语言中，需要对JVM内存模型、指令重排等深入了解，才能写出一份高质量的多线程代码。

## 内存的五大类型：

  1、栈区（stack）—   由编译器自动分配释放   ，存放函数的参数值，局部变量的值等。其    
  操作方式类似于[**数据结构**](http://lib.csdn.net/base/datastructure)中的栈。    
  2、堆区（heap）   —   一般由程序员分配释放，   若程序员不释放，程序结束时可能由OS回    
  收   。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。    
  3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的    
  全局变量和静态变量在一块区域，   未初始化的全局变量和未初始化的静态变量在相邻的另    
  一块区域。   -   程序结束后由系统释放。    
  4、文字常量区   —常量字符串就是放在这里的。   程序结束后由系统释放    
  5、程序代码区—存放函数体的二进制代码。

灵魂35问：

线程顺序：

https://www.cnblogs.com/myseries/p/11575757.html

## 构建项目管道：

Azure Pipelines