# 常用的面试

# 1.微服务

理解:

微：

服务的粒度要细，即服务要细化到API

服务：

复杂度可控(专注单一的职能)

技术选型灵活

独立部署，服务间通信使用rpc框架，不用管服务的编程语言、操作系统

容错

服务治理:

问题(服务化面临的挑战):

服务间依赖关系复杂

服务之间的负载均衡

服务监控

解决:

服务治理

dubbo

方案:

服务注册和发现

服务监控:dubbo-admin,内存中统计调用次数、关系,为扩容和降级提供依据

集群容错

负载均衡

微服务的优缺点分别是什么?说下你在项目开发中碰到的坑

优点:

每一个服务足够内聚,代码容易理解

开发效率提高,一个服务只做一件事

微服务能够被小团队单独开发

微服务是松耦合的,是有功能意义的服务

可以用不同的语言开发,面向接口编程

易于与第三方集成

微服务只是业务逻辑的代码,不会和HTML,CSS或者其他界面组合

开发中,两种开发模式

前后端分离

全栈工程师

可以灵活搭配,连接公共库/连接独立库

缺点:

分布式系统的负责性

多服务运维难度,随着服务的增加,运维的压力也在增大

系统部署依赖

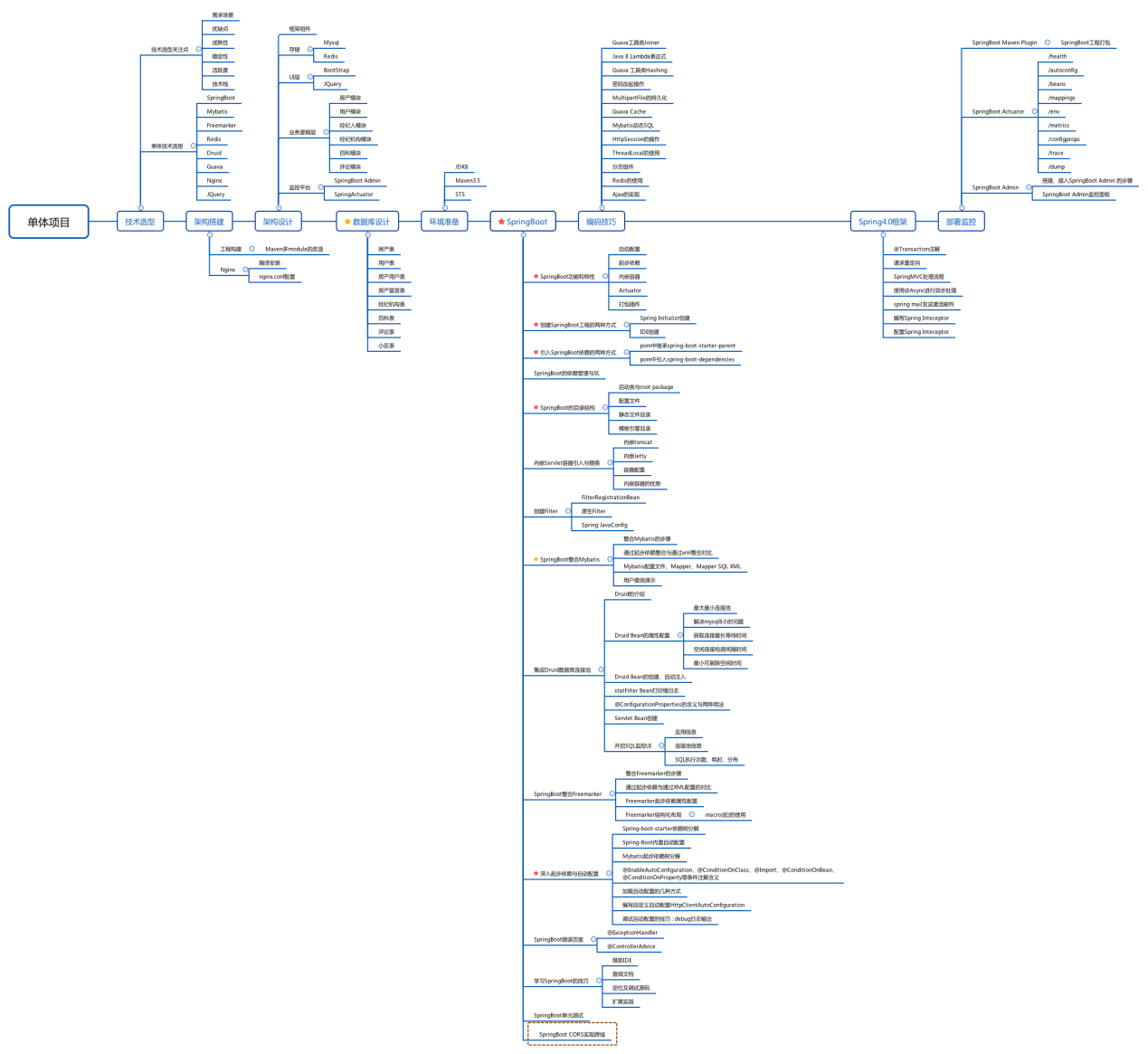
服务间通信成本

数据一致性

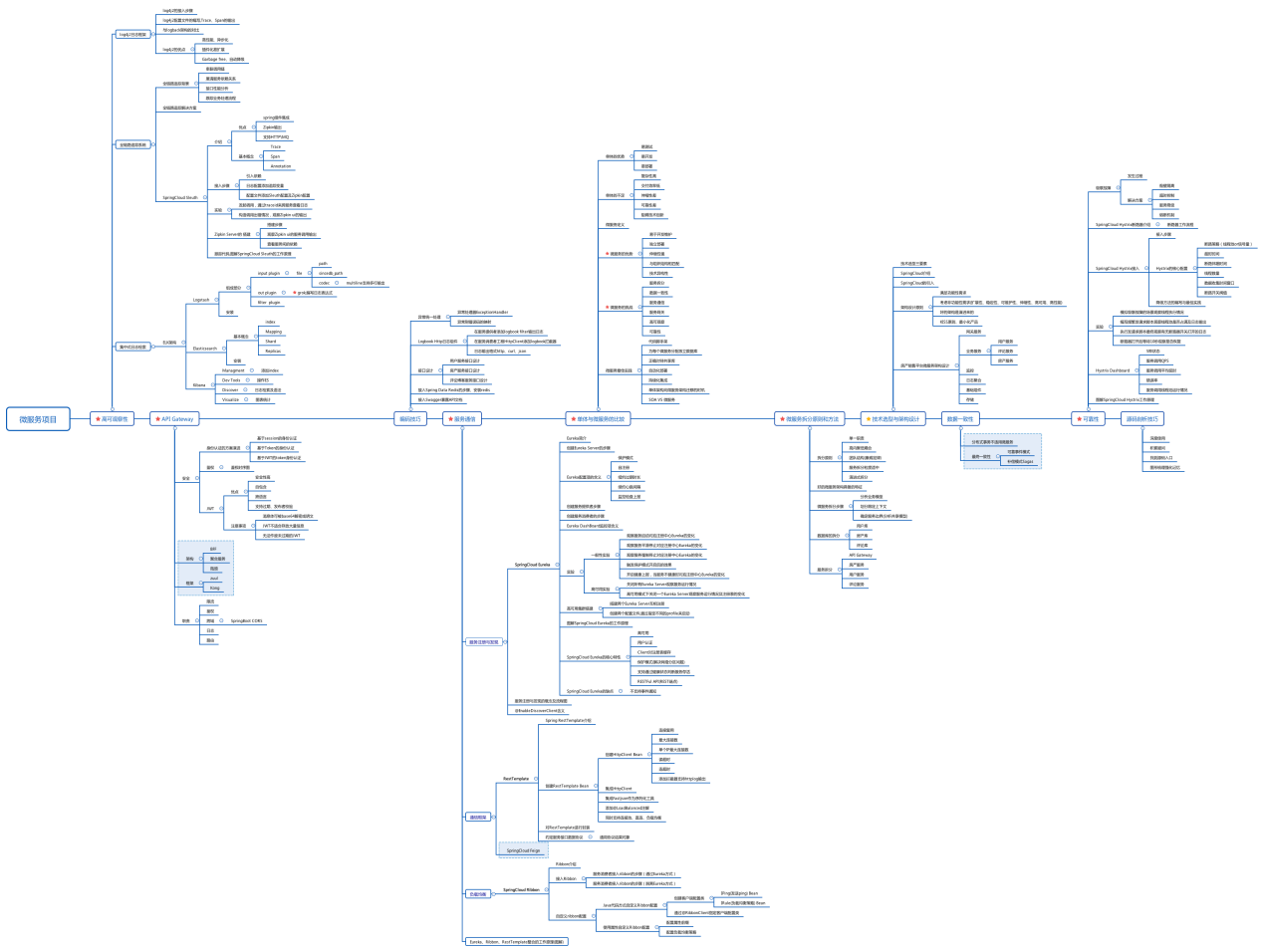
系统集成测试

性能监控

单体知识点



微服务知识点



# 2.SpringCloud

## 2.1.理解:

微服务的一整套解决方案，基于SpringBoot，组件轻量、丰富、开箱即用、解耦合

## 2.2.组件:

⒈eureka注册中心，解决url硬编码，微服务的注册和发现<-->zookeeper

⒉ribbon:负载均衡,默认@LoadBalanced轮询法

⒊hystrix:容错保护，避免故障蔓延，断路器、线程隔离，服务降级、熔断。

⒋zuul:分布式服务网关，负责内容的分发和过滤 服务的无状态性<-->nginx

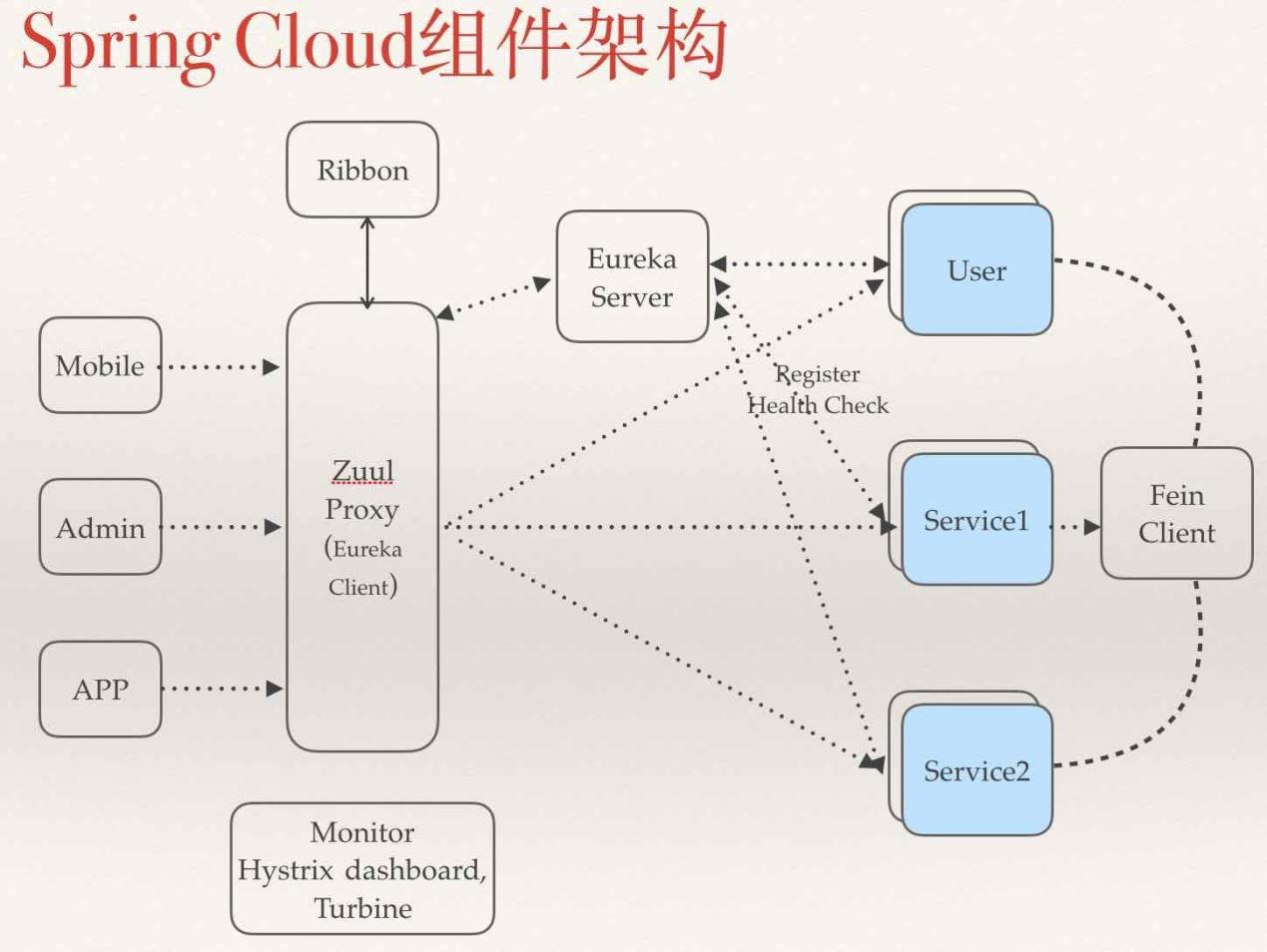
Zuul:请求转发,一般用做网关

Nginx:负载均衡,请求转发

⒌config:统一配置文件，svn、git、本地文件存储，默认git

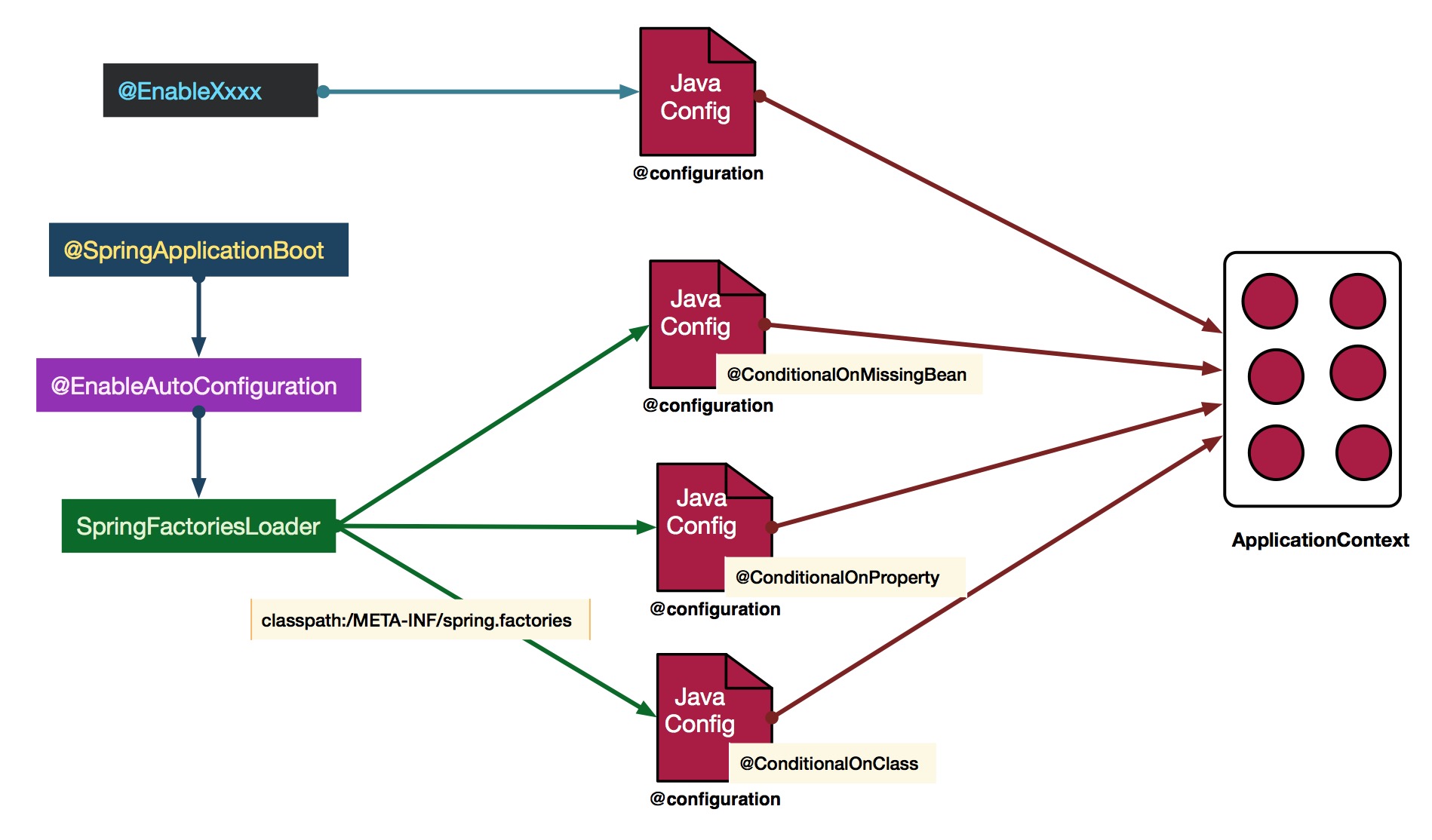
⒍feign:声明式的rest调用<-->duubo [feɪn]

⒎spring-cloud-sleuth分布式链路跟踪[sluːθ]



## 常用的starter

## SpringBoot自动配置的原理?



## 2.3.SpringBoot的理解

快速开发的脚手架

简化java配置

特点:约定优于配置,内置默认的配置

SpringBoot的常用注解:

@SpringBootApplication

@Configuration

条件注解

@Bean

导入资源文件

@PropertySource(value={‘classpath:jdbc.properties’})

@ComponentScan(basePackes=”com.leelovejava”)

@AutoConfigure

@Transactional

SpingBoot的原理:

组合注解 条件注解

ConditionalOnBean ConditionalOnMissingBean ConditionalOnClass

SpringBoot的tomcat打包:

1.maven packing war

2.spring-boot-starter-tomcat的scope为provided

3.application extends SpringBootServletInitialize [ɪ'nɪʃ(ə)laɪz]

@Overide builder方法

builder.sources(Application.class);

## Spring Boot 2.X 有什么新特性？与 1.X 有什么区别？

* 配置变更
* JDK 版本升级
* 第三方类库升级
* 响应式 Spring 编程支持
* HTTP/2 支持
* 配置属性绑定

## CAP理论

## 项目中有哪些服务？

## 2.4.SpringBoot和SpringCloud的区别?

微服务框架

SpingCloud基于SpingBoot

专注点:SpingBoot简化java配置,快速开发

SpringCloud微服务的一整套解决方案

嵌入式组件开发:

内嵌tomcat、eureka

springcloud如何实现服务的注册和发现

服务在发布时 指定对应的服务名（服务名包括了IP地址和端口） 将服务注册到注册中心（eureka或者zookeeper）

这一过程是springcloud自动实现 只需要在main方法添加@EnableDisscoveryClient  同一个服务修改端口就可以启动多个实例

调用方法：传递服务名称通过注册中心获取所有的可用实例 通过负载均衡策略调用（ribbon和feign）对应的服务

Eureka和ZooKeeper都可以提供服务注册与发现的功能,请说说两个的区别

1.ZooKeeper保证的是CP,Eureka保证的是AP

ZooKeeper在选举期间注册服务瘫痪,虽然服务最终会恢复,但是选举期间不可用的

Eureka各个节点是平等关系,只要有一台Eureka就可以保证服务可用,而查询到的数据并不是最新的

自我保护机制会导致

Eureka不再从注册列表移除因长时间没收到心跳而应该过期的服务

Eureka仍然能够接受新服务的注册和查询请求,但是不会被同步到其他节点(高可用)

当网络稳定时,当前实例新的注册信息会被同步到其他节点中(最终一致性)

Eureka可以很好的应对因网络故障导致部分节点失去联系的情况,而不会像ZooKeeper一样使得整个注册系统瘫痪

2.ZooKeeper有Leader和Follower角色,Eureka各个节点平等

3.ZooKeeper采用过半数存活原则,Eureka采用自我保护机制解决分区问题

4.Eureka本质上是一个工程,而ZooKeeper只是一个进程

Ribbon和Feign的区别：

Ribbon和Feign都是用于调用其他服务的，不过方式不同。

1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

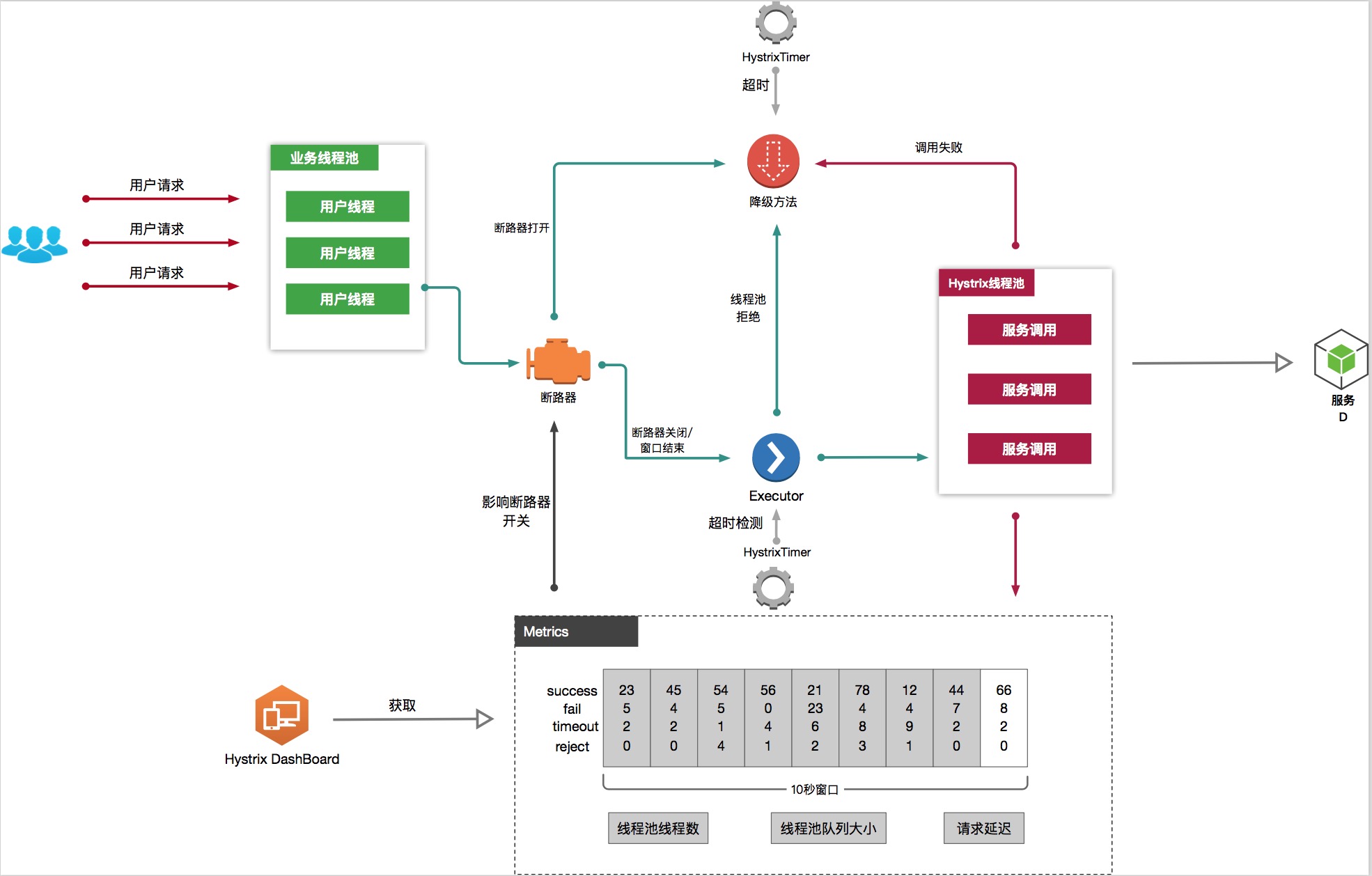
2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，模拟http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，

不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致

Hystrix的作用和原理?



什么是服务熔断?什么是服务降级

在复杂的分布式系统中,微服务之间的相互调用,有可能出现各种各样的原因导致服务的阻塞,在高并发场景下,服务的阻塞意味着线程的阻塞,导致当前线程不可用,服务器的线程全部阻塞,导致服务器崩溃,由于服务之间的调用关系是同步的,会对整个微服务系统造成服务雪崩

为了解决某个微服务的调用响应时间过长或者不可用进而占用越来越多的系统资源引起雪崩效应就需要进行服务熔断和服务降级处理。

所谓的服务熔断指的是某个服务故障或异常一起类似显示世界中的“保险丝"当某个异常条件被触发就直接熔断整个服务，而不是一直等到此服务超时。

服务熔断就是相当于我们电闸的保险丝,一旦发生服务雪崩的,就会熔断整个服务,通过维护一个自己的线程池,当线程达到阈值的时候就启动服务降级,如果其他请求继续访问就直接返回fallback的默认值

# 3.大数据

## 3.1.起源:

Goole三篇论文

Goole FS->HDFS

MapReduce

BigTable->hbase

## 3.2.架构

1. 分布式存储hdfs

2.分布式计算

MapReduce、Storm(离线、流计算)

Spark(实时、内存计算)

## 3.3.体系:

爬虫:python、jsoup

资源调度:YARN

数据仓库:hive、 hbase

数据挖掘:python、mahout

数据迁移：pig、sqoop

数据分析：可视化建模、构建用户头像

数据清洗：将不规整数据转化为规整数据

机器学习、人工智能(贝叶斯算法)、机器语言(scala)

消息队列:Kafka、RocketMQ、ZeroMQ、ActiveMQ、RabbitMQ

数据可视化

**Spark面试题**

**Spark项目：**

**一人一档项目,**

# 4.Spring：

Spring5.0新特性：

1. .基准升级,如servlet3
2. 使用jdk8、jdk9新特性

Spring 接口中的默认方法

基于 Java8 反射增强的内部代码改进

在框架代码中使用函数式编程 - lambda表达式 和 stream流

1. 响应式编程
2. 函数式web框架
3. Kotlin支持
4. 移除框架支持,如goava

## 1.概念:

Springs是一个设计层面,轻量级的控制反转(IOC)和切面(AOP)非侵入式的java企业级的开源框架,解决的是业务逻辑逻层和其他层的松耦合问题,可以整合一些框架

两个核心：AOP、IOC(DI)

解耦合(高内聚 低耦合)

## 2.注入方式：

set

构造方法

注解(@resource-先byType再byName @autowired-先byName再byType)

## 3.bean的生存周期(scope):

singleton单例

prototype:原型

request

session

global session

## 4.aop：

### 4.1.理解

面向切面编程,是一种编程思想,是对OOP的补充,将系统中非核心的业务提取出来,进行单独处理,比如事务、日志、安全等

### 4.2.实现原理：

动态代理、静态织入

### 4.3.应用：

订单系统中优惠活动使用aop织入到支付接口、动态切换数据源、日志，降低耦合度

### 4.4概念

#### ⑴横向关注点:

对哪些方法进行拦截，拦截后怎么处理，这些关注点称之为横向关注点.例如:打印日志

#### ⑵Aspect(切面):

切入到指定类指定方法的代码

切面就是对横切关注点的抽象

@Aspect

public class SleepHelper {}

#### ⑶JointPoint(切点):

切入到哪些类、哪些方法

被拦截到的点(程序执行过程中明确的点)，一般是方法的调用

#### ⑷Advice(通知):

AOP执行的动作

AOP在特定的切入点上执行的增强处理，有before,after,afterReturning,afterThrowing,around

@Before("pointCut()")

public void doBefore(JoinPoint joinPoint){

System.out.println("AOP Before Advice...");

}

#### ⑸Pointcut(切入点):

对连接点进行拦截的定义(带有通知的连接点)，在程序中主要体现为书写切入点表达式

@Pointcut("execution(\* com.java.springcode.service..\*.\*(..))")

public void pointCut(){}

#### ⑹AOP代理：

AOP框架创建的对象，代理就是目标对象的加强。

cglib创建的动态代理对象比JDK的动态代理对象的性能高，但是创建对象的时间长.对于单例无需频繁创建对象,用cglib合适。Cglib采用动态创建子类的方法，对于final、private方法无法处理.

Spring中的AOP代理使用JDK、 CGLIB代理，前者基于接口，后者基于子类

#### ⑺目标对象(target)

代理的目标对象

#### ⑻织入（weave）

将切面应用到目标对象并导致代理对象创建的过程

#### ⑼引入（introduction）

在不修改代码的前提下，引入可以在运行期为类动态地添加一些方法或字段

### 4.5要求:

启用@AsjectJ支持 <aop:aspectj-autoproxy />

## 5.SpringMVC的执行流程

1.dispatcherServlet接收用户请求

2.dispatcherServlet查询对应的请求映射器handlerMapping(@RequestMapping)，

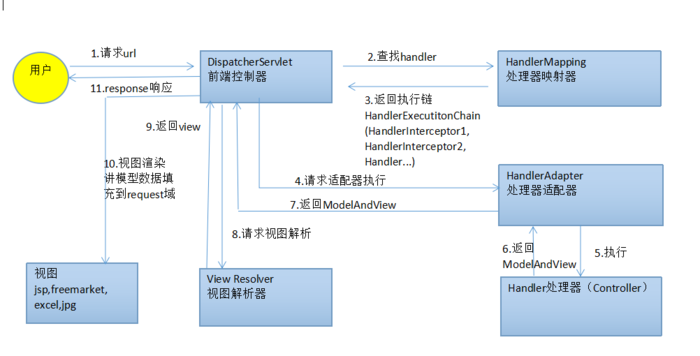
找到处理请求的controller，负责对请求进行真正的处理工作

3.dispatcherServlet请求提交给目标controller

4.controller进行业务逻辑处理，返回一个modeAndView

5.dispatcher查询到一个或多个viewResolver视图解析器，找到modelAndView对象指定的视图对象

6.视图对象负责渲染返回客户端



## 6.Sruts2和SpringMVC的区别？

Struts2 SpringMVC

机制 入口是filter 入口是servlet

性能 基于类，多例 基于方法的设计，单例，优越

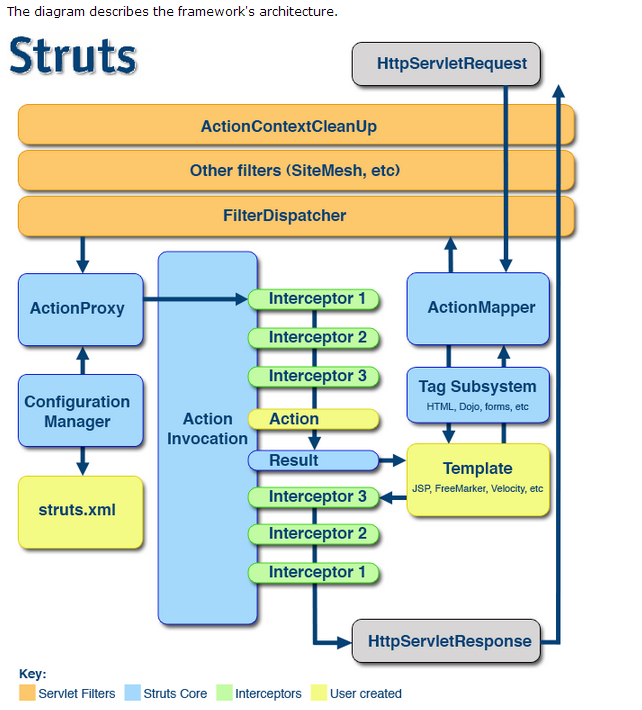
拦截器实现机制 自己的intercepter 独立的aop

数据 存储值栈、OGNL存取 request、moderAndView、表达式SpEL

设计思想 oop servlet扩展

过滤器和拦截器的区别?

1. 拦截器是基于java的[反射机制](https://www.baidu.com/s?wd=%E5%8F%8D%E5%B0%84%E6%9C%BA%E5%88%B6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)的，而过滤器是基于[函数](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)[回调](https://www.baidu.com/s?wd=%E5%9B%9E%E8%B0%83&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)。  
 2. 拦截器不依赖与servlet容器，过滤器依赖与servlet容器。   
 3. 拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用。  
 4. 拦截器可以访问action[上下文](https://www.baidu.com/s?wd=%E4%B8%8A%E4%B8%8B%E6%96%87&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)、值栈里的对象，而过滤器不能访问。   
 5. 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器[初始化](https://www.baidu.com/s?wd=%E5%88%9D%E5%A7%8B%E5%8C%96&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)时被调用一次



## 7.Spring中的设计模式

简单工厂:BeanFactory,根据传入一个唯一的标识来获得Bean对象

工厂方法:AbstractFactoryBean,createInstance()由子类负责具体创建对象

单例（Singleton）:bean的scope默认为Singleton

适配器（Adapter）:HandlerAdapter

代理（Proxy）:为其他对象提供一种代理以控制对这个对象的访问.AOP实现了动态代理,cglib、JDK

模板方法（Template Method）:定义一个操作中的算法的骨架，而将一些步骤延迟到子类中

JdbcTemplate的execute方法

责任链模式:HandlerExecutionChain,SpringMVC的拦截器链

chain[tʃeɪn]

## 8.Spring事务传播属性：

1. PROPAGATION\_REQUIRED:支持当前事务，如果当前没有事务，就新建一个事务.默认

2. PROPAGATION\_SUPPORTS:支持当前事务，如果当前没有事务，就以非事务方式执行。

　　3. PROPAGATION\_MANDATORY:支持当前事务，如果当前没有事务，就抛出异常。

　　4. PROPAGATION\_REQUIRES\_NEW:新建事务，如果当前存在事务，把当前事务挂起。

　　5. PROPAGATION\_NOT\_SUPPORTED:以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

　　6. PROPAGATION\_NEVER:以非事务方式执行，如果当前存在事务，则抛出异常。

## 9.Spring事务隔离级别：

　　1. ISOLATION\_DEFAULT:默认使用数据库默认的事务隔离级别.

　　2. ISOLATION\_READ\_UNCOMMITTED： [读未提交].允许另外一个事务可以看到这个事务未提交的数据。

　　 3. ISOLATION\_READ\_COMMITTED： [读已提交].保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据

　　 4. ISOLATION\_REPEATABLE\_READ： [可重复读]

　　 5. ISOLATION\_SERIALIZABLE :[读序列化],读操作会隐式获取共享锁，可以保证不同事务间的互斥。

问题：

脏读:

另一个事务修改了数据，但尚未提交，而本事务中的SELECT会读到这些未被提交的数据。

不重复读:

同一个事务执行过程中，另外一个事务提交了新数据，本事务先后两次读到的数据结果会不一致。

幻读:

两个事务同时提交了新数据，本事务再更新时，select更新后的数据

锁

悲观锁:

每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁.

传统的数据库,使用此机制,如行锁、表锁、读锁、写锁,适用于冲突多

乐观锁:

每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制

适用多读,省去了锁的开销，加大了系统的整个吞吐量

## Spring原理:

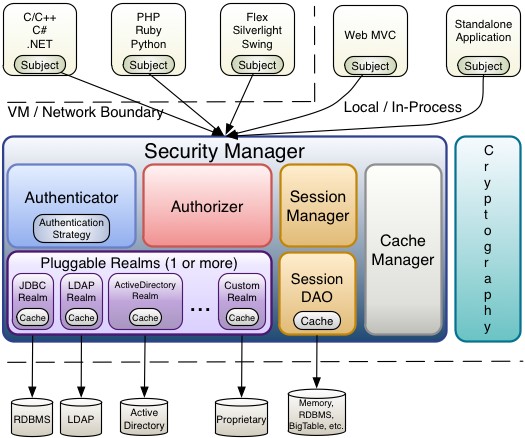
# 5.Shiro

## 概念:

安全框架,简化验证、授权的框架

功能:认证、授权、加密、会话管理、与Web集成、缓存

## 组件:



SecurityManager安全认证管理者,核心. 相当于SpringMVC中的DispatcherServlet或者Struts2中的FilterDispatcher

subject:主体,代表了当前“用户”

authenticator:身份认证/登录，验证用户是不是拥有相应的身份；

authorizer授权器,权限验证，验证某个已认证的用户是否拥有某个权限；

即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限

realm:数据源,获取安全数据（如用户、角色、权限）

sessionDao数据访问对象

cacheManager:缓存控制器,来管理如用户、角色、权限等的缓存的；因为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能

sessionManager:会话管理,用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话

crptography:密码凭证(加密算法、散列):保护数据的安全性，如密码加密存储到数据库，而不是明文存储

rememberMe:记住我

## 授权方式：

1.编程式

@RequiresPermissions("item:edit")

2.注解式

hasRole方法，返回值为boolean类型（判断当前用户是否有一个角色)

isPermitted方法，返回值类型为boolean（判断当前用户是否拥有一个权限）

3.jsp标签

判断是否有角色abc

<shiro:hasRole name="abc">

拥有权限资源abc

<shiro:hasPermission name="abc">

## 表设计

用户、角色、权限、用户-角色、用户-权限

## 权限模块设计

---------

# 6.dubbo

## ⒈概念:

高性能的rpc框架，资源调度、服务治理的管理工具，可以统计系统之间的调用关系和调用次数

## ⒉角色:

服务提供者、服务消费者、注册中心、监控中心

## ⒊软负载均衡算法:

随机

轮询

最小活跃调用数(慢的提供者收到更少请求)

一致性hash(相同的请求发到同一提供者,宕机平摊到其他提供者)

Nginx的负载均衡算法

## ⒋序列化协议:

dubbo、hession、http、rmi、webservice、thrift

java序列化

实现 Serializable 接口的对象，可以序列化为本地文件

Java序列化是指把Java对象保存为二进制字节码的过程

Java反序列化是指把二进制码重新转换成Java对象的过程

## ⒌集群容错模式:

失败自动切换，当出现失败，重试其它服务器

快速失败，只发起一次调用，失败立即报错

失败安全，出现异常时，直接忽略

失败自动恢复，后台记录失败请求，定时重发

并行调用多个服务器，只要一个成功即返回

广播调用所有提供者，逐个调用，任意一台报错则报错

## ⒍mq和rpc的适用场景:

RPC:

客户端明确调用的服务器

调用需要立即得到返回结果

架构简单,同一个请求的关注者只有一个

MQ:

发送者并不明确谁是消费者

发送者并不关心谁来消费消息

消息的发送和处理是异步的

同一个请求的关注者不只一个

---------

# 7.Zookeeper

## 1.概念:

分布式集群协调服务、数据发布与订阅、分布式事务锁、配置中心

## 2.角色，作用?：

Leader:群首,负责相应所有对Zookeeper状态变更的请求

follower(追随者):相应本服务器上的读请求,处理leader的提议,在leader提交该提议时在本地进行提交

observer:观察者，不参加选举、不响应提议

提高读取的吞吐量,场景:集群的读取负载高 客户端多到跨机房

## 3.节点类型：

持久、持久顺序、临时、临时顺序

## 4.ZooKeeper和Eureka对比

分布式架构设计的CAP原理:

consistency->C->一致性 [kən'sɪst(ə)nsɪ]

availability->A->可用性 [ə,veɪlə'bɪlətɪ]

partition tolerance->P->分区容错性 [pɑː'tɪʃ(ə)n] ['tɒl(ə)r(ə)ns]

Zookeeper:

基于cp(数据一致性、容错性),服务不可用情况:正在选主、集群半数机器不可用,对于服务消费者来说,能消费才是最重要的

作为注册中心，其实配置是不经常变动的，只有发版和机器出故障时会变。对于不经常变动的配置来说,CP是不合适的

SpringCloud支持zookeeper作注册中心

Eureka:

基于AP(可用性、容错性),牺牲一致性,保证数据的可用性,返回旧数据和缓存数据

运行多个实例,解决单点问题

Peer to Peer对等通信,去中心化的架构

2.0闭源,最新版的springCloud使用eureka1.9.2

总结:

理论上Eureka是更适合作注册中心。

现实环境中大部分项目可能会使用ZooKeeper，那是因为集群不够大，并且基本不会遇到用做注册中心的机器一半以上都挂了的情况

---------

# 8.MQ

## ⒈常用的消息队列比较(kafka、ActiveMQ、RabbitMQ、RocketMQ、ZeroMQ都有什么优点，缺点啊??)

持久化消息:

持久化消息主要是指：MQ down或者MQ所在的服务器down了，消息不会丢失的机制

zeroMq不支持,activeMq和rabbitMq都支持

核心技术：

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统、社区:

RabbitMq最好，ActiveMq次之，ZeroMq最差

高并发:

RabbitMQ最高

https://blog.csdn.net/oMaverick1/article/details/51331004

kafaka:

优点:

分布式可高可扩展、高性能(单机吞吐量10w)

容错

日志

缺点:

不支持AMQP事务处理(通讯协议:自行设计的基于TCP层的协议)

重复消息(消息可能发送多次)

消息乱序

复杂性(需要zookeeper 集群的支持,Topic通常需要人工来创建，部署和维护较一般消息队列成本更高)

集群:需三台zookeeper

## ⒉如何避免MQ重复消费

1.正常情况下,消费者在消费消息时候，消费完毕后，会发送一个确认信息给消息队列，消息队列就知道该消息被消费了，就会将该消息从消息队列中删除,

只是不同的消息队列发送的确认信息形式不同,例如RabbitMQ是发送一个ACK确认消息，RocketMQ是返回一个CONSUME\_SUCCESS成功标志，

kafka实际上有个offset的概念,就是每一个消息都有一个offset，kafka消费过消息后，需要提交offset，让消息队列知道自己已经消费过了

2.产生原因:

网络传输等故障.确认信息没有传送到消息队列，导致消息队列不知道自己已经消费过该消息了，再次将该消息分发给其他的消费者

3.解决:根据业务场景

⑴拿消息去数据库insert,消息做一个唯一主键,重复消息后,会主键冲突

⑵redis的set,无需处理,set几次结果都是一样,幂等性

⑶引入第三方介质,比如说redis,消息分配全局id,只要消费过消息,写入到redis,开始消费前先去redis查询有无消费记录

MQTT重复推送:

问题排查:人员到店信息推送,查看日志,后台只推送一条,前台控制台打印mqtt连接异常,重新连接

解决办法:后台返回数据中,增加一个唯一标识,前台根据唯一标识，是否消费过

## ⒊RabbitMQ

### ⑴.理解

消息队列是应用程序和应用程序中之间的通信方法.rabbitMQ是一个开源的，可复用的企业消息系统,支持主流的操作系统和多种开发语言的支持

### ⑵好处:

异步处理 消息通讯 流量削峰 应用接口 日志处理

### ⑶项目中应用:

超市收银系统的扫描枪,扫描时将条形码发送到队列,取条形码

### ⑷Spring整合

#### ①配置pom.xml

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit</artifactId>

<version>1.4.6.RELEASE</version>

</dependency>

#### ②配置properties文件

#rabbitmq

rabbit.host=192.168.1.75

rabbit.username=rabbitmq

rabbit.password=123456

rabbit.port=5672

rabbit.exchange=mq-exchange

exchange:交换机,

.

三种模式(订阅、路由、通配符)

消息发送到交换机,经过队列绑定交换机后,到达队列

订阅:一个消息被多个消费者读取

路由:生产者发送消息添加路由键,消费者绑定队列到交换机时，需要添加路由键,才能接受消息

通配符:思想和路由模式一样,路由键支持模糊匹配,例如:#匹配一个或多个,\*

#### ③配置xml,spring-context-rabbitmq.xml

<!-- 1.连接服务配置(rabbitMQ 连接工厂)-->

<rabbit:connection-factory id="connectionFactory" host="${rabbit.host}" username="${rabbit.username}"

password="${rabbit.password}" port="${rabbit.port}" />

<!--2.MQ的管理，包括队列和交换器等-->

<rabbit:admin connection-factory="connectionFactory"/>

<!-- 3.queue 队列声明-->

<rabbit:queue durable="true" auto-delete="false" exclusive="false" name="offer\_rec"/>

<!-- 4.exchange queue binging key 绑定 (定义交换机 topic~exchange，手动绑队列)-->

<rabbit:direct-exchange name="mq-exchange" durable="true" auto-delete="false">

<rabbit:bindings>

<rabbit:binding queue="offer\_rec" key="offer\_rec\_key"/>

</rabbit:bindings>

</rabbit:direct-exchange>

<!-- spring amqp默认的是jackson 的一个插件,目的将生产者生产的数据转换为json存入消息队列-->

<bean id="jsonMessageConverter" class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter"></bean>

<!-- 5.spring template声明(定义模板 rabbit:template)-->

<rabbit:template reply-timeout="0" exchange="mq-exchange" id="rabbitTemplate" connection-factory="connectionFactory"

message-converter="jsonMessageConverter"/>

#### ④发送消息

@Autowired

private RabbitTemplate rabbitTemplate;

#### ⑤调用

rabbitTemplate.convertAndSend(QueueUtil.offer\_rec, MapUtil.pakMap(OptionUtil.save, bizType));

## Kafka

使用场景?

优点?

遇到的问题，如何解决?

# 9.http/Tcp

## 9.1HTTP状态码

200:请求成功

301:资源（网页等）被[永久转移]到其它URL(Permanently Moved),重定向,旧资源不在

302:资源（网页等）被[暂时性转移]到其它URL(Temporarily Moved),重定向,旧资源在

400:请求无效Bad request

403:服务器理解请求客户端的请求，但是拒绝执行此请求Forbidden,例:method不匹配

404:请求的资源（网页等）不存在

500:内部服务器错误

100-199 信息，服务器收到请求，需要请求者继续执行操作

200-299 成功，操作被成功接收并处理

300-399 重定向，需要进一步的操作以完成请求

400-499 客户端错误，请求包含语法错误或无法完成请求

500-599 服务器错误，服务器在处理请求的过程中发生了错误

## 9.2.Tcp和UDP的区别？

TCP UDP

可靠性 是 否

有序性 是 否

速度 慢 快

头大小 大 轻

运用场景:

tcp适用高可靠,对网络的传输时间要求不高,如金融领域

udp适用于快速、高速的传输,如娱乐和游戏

## 9.3.TCP理解:

传输控制协议

## 9.4.TCP的网络七层

链路、网络、传输、应用

表示层、会话层、物理层

## 9.5.TCP三次分手四次握手

三次握手(建立连接)、四次分手(终止连接)-传输层

https://blog.csdn.net/github\_33736971/article/details/51553321

三次握手:

SYN(类型,值为0、1,开关)

第一次握手:

客户端发送SYN（SEQ=x）(SYN同步、SEQ序列号)报文给服务器端，进入SYN\_SEND状态。

\*Client什么都不能确认;Server确认对方发送正常

>你好,我要和你建立连接,你听到没有,我的打招呼

第二次握手:

服务器端收到SYN报文，回应一个SYN （SEQ=y）ACK(ACK=x+1）报文，进入SYN\_RECV状态。

\*Client确认:自己发送、接收正常,对方发送、接收正常

\*Server确认:自己接收正常,对方发送正常

>你好,我收到了你的招呼,我向你打招呼,你收到没有

第三次握手:

客户端收到服务器端的SYN报文，回应一个ACK(ACK=y+1）报文，进入Established状态。

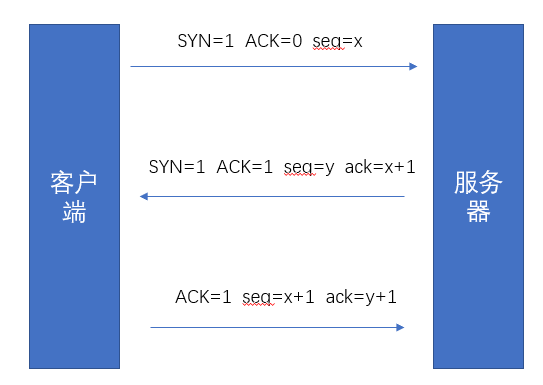
\*Client确认:自己发送、接收正常,对方发送、接收正常

\*Server确认:自己发送、接收正常,对方发送、接收正常

>我收到了你的招呼

目的:建立可靠的通信通道

三次握手确认双方都能收发功能呢正常,完成后,TCP客户端和服务器端成功地建立连接，可以开始传输数据了



四次分手:

FIN(类型,值为0、1,开关)

第一次分手：

Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

>A你要的东西,我收到了,我要断开这次连接

第二次分手：

Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

>B好,我知道你要断开这次连接了

第三次分手：

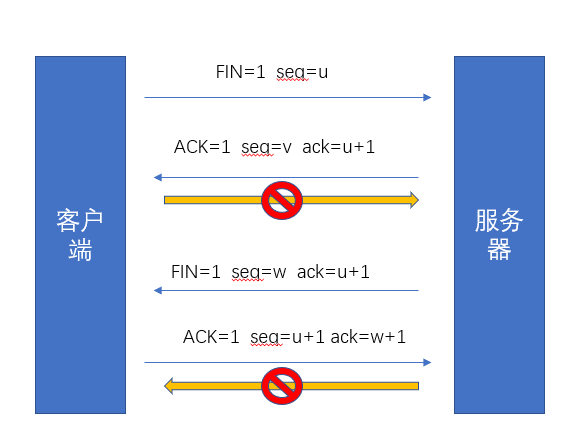
Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

>A我要绝交

第四次分手：

Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手

>好,我知道你要和我绝交了



# 10.netty

## 10.1.挑战：

分布式带来的服务之间调用开销，要用多个socket连接，但有时连接用不到，导致资源浪费、线程之间切换的开销。

netty通过[多路复用]的方式，让一个线程处理多个socket。

## 10.2.理解

通讯框架、nio异步非阻塞、事件驱动

好处:并发高(多路复用IO)、传输快(零拷贝)、封装好

## 10.3.核心组件:

管道channel

概念:

数据传输流,传入（入站）或者传出（出站）数据的载体

状态：

打开、连接、关闭

主要操作：

读、写、连接、绑定

注意:

使用完毕后，调用close方法，释放管道占用的资源

fature:

异步操作的占位符，执行完回填

future.isSuccess()

事件:

打开连接、关闭连接

ChannelHandler

## 10.4.拆包粘包：

### 10.4.1问题描述:

一个完整的包可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送

TCP拆包:

服务端分两次读取到了两个数据包，第一次读取到了完整的D1包，和D2包的部分内容，第二次读取到了D2包的剩余内容

TCP粘包:

服务端一次接收到了两个数据包，D1和D2是粘合在一起

### 10.4.2发生原因:

1、要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。

2、待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

3、要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。

4、接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包

### 10.4.3.解决办法：

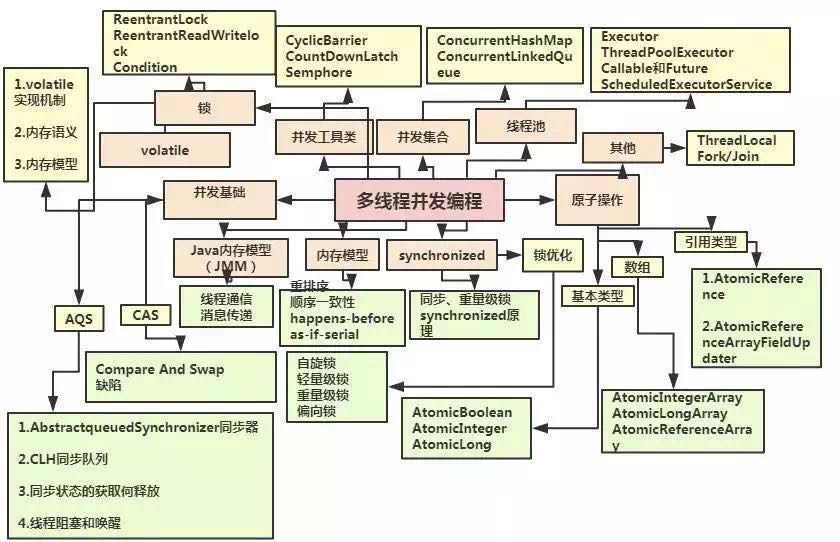
消息固定长度(FixedLengthFrameDecoder类、不够用0填充)

包尾使用换位符进行分割

消息分为消息头、消息体

解码器：Google的Protobuf

# 11.多线程



## 1.理解

一个程序有多条执行路径

## 2.运用场景：

异步处理，数据量比较大(导入/导出)，分布式计算

实例：国调项目中数据批量导入

## 3.线程和进程的区别？

一个程序至少有一个进程，一个进程至少有一个线程

线程是系统执行的最小单位,同一个进程里的线程资源共享

进程资源不同享

## 4.同步 synchronied

同步方法 同步代码快 静态同步方法

## 5.线程安全类

hashMap collecton vetchor StringBuffer

## 6.锁

### 1>.对象锁、类锁的区别?

对象锁:实例方法之间的同步

类锁:静态方法

### 2>.分布式锁:zookeeper、redis

zk步骤:

1.建立一个节点，假如名为：lock 。节点类型为持久节点（PERSISTENT）

2.每当进程需要访问共享资源时，会调用分布式锁的lock()或tryLock()方法获得锁，这个时候会在第一步创建的lock节点下建立相应的顺序子节点，

节点类型为临时顺序节点（EPHEMERAL\_SEQUENTIAL），通过组成特定的名字name+lock+顺序号。

3.在建立子节点后，对lock下面的所有以name开头的子节点进行排序，判断刚刚建立的子节点顺序号是否是最小的节点，假如是最小节点，则获得该锁对资源进行访问。

4.假如不是该节点，就获得该节点的上一顺序节点，并给该节点是否存在注册监听事件。同时在这里阻塞。等待监听事件的发生，获得锁控制权。

5.当调用完共享资源后，调用unlock（）方法，关闭zk，进而可以引发监听事件，释放该锁。

实现的分布式锁是严格的按照顺序访问的并发锁

### 3>.死锁

死锁产生原因：

同步嵌套

避免死锁:

阻止循环等待条件,将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）

检测线程是否有锁:

java.lang.Thread中方法holdsLock()，返回true如果当且仅当前线程拥有某个具体对象的锁

乐观锁和悲观锁，应用场景有哪些？什么情况下会发生死锁，怎么处理死锁？**美团**

**多线程 重点**

## 7.多线程状态

**新建**、**就绪**、**运行**、**阻塞**、**死亡**

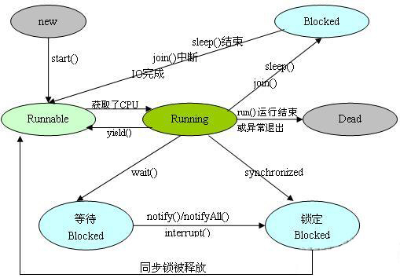
new() start() run() run()完成/stop()/destroy()/停止Thread.currentThread().interrupt()

新建-----> 就绪-----> 运行 -----> 死亡

notify()/notifyAll() join()、wait()、sleep()

|\_\_ \_\_|

阻塞



## 8.创建线程

1>.继承java.lang.thread，重写run()

对象.start()

2>.实现java.lang.Runnable()接口，重写run()-推荐

Thread t = new Thread(对象)

t.start()

使用代理

## 9.Thread类中的start()和run()方法的区别?

start()方法被用来启动新创建的线程，而且start()内部 调用了run()方法

调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程

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，这意味着可以用jvm调度并执行，这不意味着线程会立即执行

run()方法可以产生必须退出的标志来停止一个线程

## 10.sleep yield wait的区别

wait()方法是Object的方法，sleep()是Thread的方法

wait()方法必须写在synchronized里面

wait()方法在等待的时候会释放对象锁，sleep睡过去也抱着对象锁不放

yield()让当前正在执行的线程暂停，该方法不会阻塞线程，而是将线程转入可运行状态

## **11.线程池? 重点**

**[java常用的几种线程池比较](https://www.cnblogs.com/aaron911/p/6213808.html)**

<https://www.cnblogs.com/aaron911/p/6213808.html>

1).常用线程池:druid、DBCP、c3p0

newCachedThreadPool

newFixedThreadPool

newSingleThreadExecutor

newScheduleThreadPool

2).为什么用?使用场景?风险？

风险:

死锁

资源不足

并发错误

线程泄露

3).参数: **美团**

corePoolSize:线程池数

maxPoolSize:最大线程池数

keepAliveTime:线程池空闲时间

allowCoreThreadTimeout:是否允许核心线程空闲退出，默认值为false

queueCapacity:任务队列容量

workQueue:阻塞队列，用来存储等待执行的任务

底层实现:

http://www.cnblogs.com/dongguacai/p/6030187.html

---------

# 数据库

SQL(mysql oracle sqlServer)

NoSQL(redis mongodb hbase)

## 1.mysql oracle sqlServer的区别?

Oracle:用户来管理数据库，诞生早、结构严谨、高可用、高性能，应用广泛

MySQL:开源、简便易用,不支持事务操作、子查询 、外键、存储过程和视图等功能

SQL SERVER：集成了MS公司的各类产品及资源，提供了强大的可视化界面、高度集成的管理开发工具，在快速构建商业智能（BI）方面颇有建树

## 2.mysql的索引的类型?如何建索引、联合索引?实现原理?索引失效情况?索引优化?

优点:

提高查询效率

缺点:

降低更新表的效率

占用磁盘空间的索引文件

索引类型

A.普通索引

B.唯一索引

C.主键索引

D.组合索引

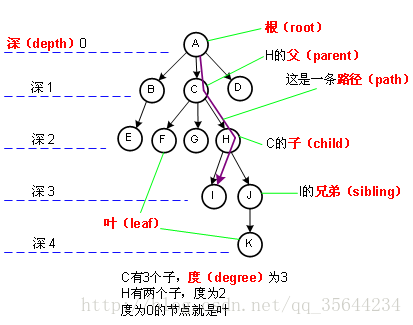
E.全文索引

实现原理:

二叉树算法b+tree

https://blog.csdn.net/qq\_35644234/article/details/53013738

二叉树的数据结构图:



创建索引:

A.普通索引

ALTER TABLE 表名 ADD INDEX `索引名` (`字段名`)

B.联合索引

ALTER TABLE 表名 ADD INDEX `索引名` (`字段名1`,`字段名2`,`字段名3`)

索引失效:

值为null

优化:

A.避免索引失效

①.索引列排序

order by 不会使用索引

②.like查询以%开头

like 字段名% 不会使索引失效,cancat解决

③.索引包含有null值的列

④.使用or、!=

⑤.where 查询条件使用函数

⑥.列数据类型为string,未用引号

B.使用短索引

C.数据过多的列

## 3.数据库的高并发?

服务层面

配置mysql性能优化参数

系统层面

优化数据表结构、字段类型、字段索引、分表，分库、读写分离等等

数据库层面

a.优化SQL语句，合理使用字段索引

b.根据主键查询

c.大数据表删除,用truncate table代替delete

d.合理使用索引

e.查询时,使用列名(select \*)

f.尽量少嵌套子查询,使用关联

g.避免在 where 条件中进行函数计算

h.减少对表的查询

i.尽量多使用 COMMIT

k.使用 DECODE 函数来减少处理时间

l.删除重复记录

m.sql 语句用大写的

o.用 IN 来替换 OR

代码层面

使用缓存和NoSQL数据库方式存储，如MongoDB/Memcached/Redis来缓解高并发下数据库查询的压力

5000条SQL批量插入，性能最佳?

高并发下的数据错乱问题？

## 4.事务?

### ⑴ 分类:

①隐式事务

②自动提交事务：针对单个数据库操作

③显式事务：针对多个数据库操作

A- 开始事务 begin transaction

B- 提交事务 commit transaction

C- 回滚事务 rollback transaction

### ⑵、特点：ACID

A- 原子性，事务是最小的、不可分割的逻辑单元

C- 一致性，事务前后的数据应保持一致

I- 隔离性，事务之间是相互独立的

D- 永久性，事务提交后对数据库的影响是永久的，一旦提交则无法回滚

分布式事务

原理

阿里的分布式事务GTS

可以与MySQL和PostgreSQL数据源,Dubbo等RPC框架,MQ消息队列等中间件配合使用

支持SpringBoot

跨数据库、跨服务、消息队列、混合的分布式事务,多种事务模式

## 5.Spring事务的传播特性?

## 6.数据库访问隔离级别?

Read uncommitted

读未提交,问题:脏读

Read committed

读提交,Sql Server、Oracle等大部分db默认,问题:不可重复读

Repeatable read

重复读,mysql默认,问题:幻读

Serializable

事务串行化顺序执行

## 7.redis的理解?应用场景?数据类型?数据库对应的应用场景?优化?

概念:

开源的高性能键值对数据库

优点:

多种数据类型存储(字符串 散列 列表 集合 有序集合类型)

内存存储与持久化

功能丰富

简单稳定,支持事务

应用场景:发送短信验证码,将手机号码作为key,短信验证码为value,过期时间为5分钟

会话缓存 队列 排行榜/计数 注册中心

项目中的应用:

存储用户的信息

网红系统的网红排行榜

存储产品分类

数据类型:

string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)

redis4.0新增stream队列,模仿kafka

优化:

a.合理使用数据类型

b.内存预分配

c.持久化机制,两种方式 定时快照方式(snapshot)和基于语句追加方式(aof)

根据业务开启或关闭,关闭节省内存

## 8.redis的api?设置生存时间?

客户端jedis、SpingDataRedis

redis-cli

设置生存时间:

exprie key time(秒)

查询生存时间:

til key time(秒)

redisTemplate

hasKey(key) 是否存在key

opsForValue().get(key) 获取key

opsForValue().set(key,value) 设置key

opsForValue().set(key,value,EXP\_TIMES,TimeUnit) 设置过期时间

opsForHash() hash()

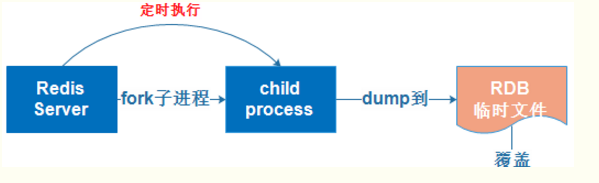
## 9.为什么redis是单线程?(redis为什么快?)

redis是内存数据库,内存不是瓶颈,瓶颈在网络IO

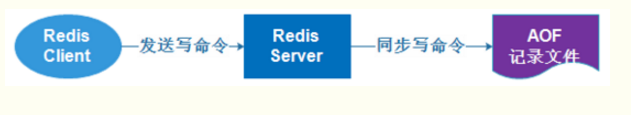
单线程的好处,不用考虑锁 并发 线程切换的开销,代码简洁,执行效率高

## Redis持久化:

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。



AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。



推荐RDB,只有一个文件、性能最大化

## 10.redis和memcached的区别?

(1) 数据支持类型:memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型

(2) 速度:redis的速度比memcached快很多

(3) 存储方式：redis可以持久化其数据

12.分页?取前10条?

MySQL:

select \* 表名 limit pageIndex,(pageIndex-1)\*pageSize

Oracle:

SELECT \* FROM

(SELECT ROWNUM rn,t.\* FROM

(SELECT ename,sal FROM emp ORDER BY sal) t --排序查询所有结果

WHERE ROWNUM<=10) --限制查询的上限（pageIndex\*pageSize）

WHERE rn>5 --限制查询的下限(pageIndex-1)\*pageSize

## 13.存储过程的优缺点?手写

优点

a.模块化，按功能进行封装

b.重用性好，易于维护

c.执行效率高，编译好之后存储在数据库中

d.减少网络流量的使用

e.安全性高

缺点

数据库移植性差

创建

CREATE OR REPLACE PROCEDURE 过程名

AS

声明部分

BEGIN

逻辑处理部分

END;

调用

BEGIN

过程名

END;

## 14.游标的类型?手写查询表的游标?

游标是系统为用户开设的一个数据缓冲区，存放SQL语句的执行结果,作用是从临时存储从数据库中提取的数据块

类型:

隐示

显示

ref

优点：

a .对从表中检索出的数据进行操作非常灵活。

b.允许应用程序对查询语句select 返回的行结果集中每一行进行相同或不同的操作。

c .提供对基于游标位置而对表中数据进行删除或更新的能力。

缺点：

游标速度较慢

显示游标:

declare cursor cur\_rs is select \* from student;

sinfo student%rowtype;

begin

open cur\_rs;

loop

fetch cur\_rs into sinfo;

exit when cur\_rs%%notfound;

dbms\_output.put\_line(sinfo.sname);

end loop;

exception when others then

dbms\_output.put\_line(sqlerrm);

end;

## 15.jdbc的执行过程?

①.加载数据库驱动

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

②.创建连接

Connection con =

DriverManager.getConnection(url , username , password ) ;

③.创建一个Statement

④.执行SQL语句

⑤.处理结果

⑥.关闭JDBC对象

## 16.数据库连接池的理解?

连接池的实现是以空间换时间

优点:效率高，安全性稳定性好，节约资源

## 17.差集?交集?并集?

交集intersect

并集union

差集except

## 18.union和union all的区别?

同:合并结果集

union:去除重复值，会排序，效率低

union all:不会去除重复值

## 19.数据库插入字段和修改字段？

添加字段:

alter table 表名 add 字段名 数据类型 [是否允许非空]

修改字段:

alter table 表名 modify 字段名 字段类型 [是否允许非空]

## 20.数据库锁?

悲观锁:假设数据肯定会冲突，所以在数据开始读取的时候就把数据锁定住。

乐观锁:认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，

如果发现冲突了，则让用户返回错误的信息，让用户决定如何去做。

表级锁、行级锁、页面锁的区别?

表级锁 : 开销小，加琐快。不会出现死锁；锁粒度大，发出琐冲突的概率最高，并发度最低。

行级锁:开锁大，加琐慢；会出现死琐；琐力度最小，发生琐冲突概率最低，并发度也最高。

页面锁:锁力度界于表锁和行锁之间，并发度一般

## 21.数据库当前时间

mysql:

now()

select current\_date; 获取当前 日期

select current\_time; 获取当前 时间

select current\_timestamp; 获取时间戳 就是日期时间

sqlserver:

select getdate()

oracle:

select to\_char(SYSDATE，'MM-DD-YYYY HH24:MI:SS') "now" from dual;

## 22.外连接中的左外连接、右外连接和全外连接有什么特点?

左外连接：A left [outer] join B on 等值条件

右外连接：A right [outer] join B on 等值条件

完全外连接：A full [outer] join B on 等值条件

## 23.数据库去重?

distinct

left join B on ... where B.column is null

查询重复的数据:

having count(column) > 1

24.数据库的存储引擎？(Mysql 中 MyISAM 和 InnoDB 的区别有哪些？)

MyISAM不支持事务,适合查询多，支持全文索引,查询效率高,5.1之前默认

InnoDB: 默认,适用于读写频繁,支持事务，行锁，外键

Hbase

1. .项目中使用?存储什么数据?
2. 二级索引

3).rowKey设计

4）.过滤器

---------

# 13.JAVA

## java的length、length()、size()的区别?

length:数组的属性

length():字符串String

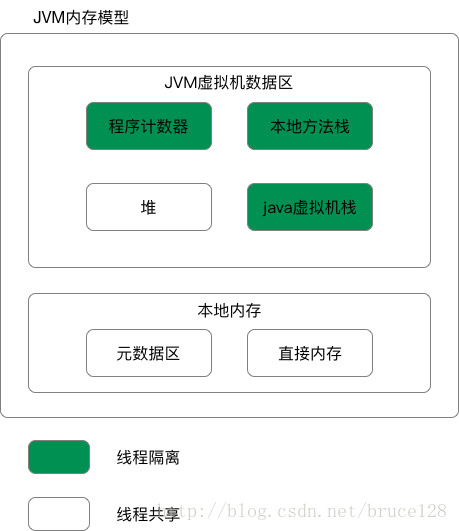
size():集合的元素

## JVM(重点)

### 内存模型

https://blog.csdn.net/bruce128/article/details/79357870

JVM



1. 程序计数器

2. Java虚拟机栈

3. 本地方法栈

4. 堆

5. 方法区

**元数据区取代了永久代,本质一致,** 是对JVM规范中方法区的实现

**元空间与永久代之间最大的区别:** **元数据空间并不在虚拟机中，而是使用本地内存**

**元数据:描述数据的数据**

**程序计数器**:当前线程正在执行的字节码的行号指示器

如果线程正在执行的是一个Java方法，PC值为正在执行的虚拟机字节码指令的地址

如果线程正在执行的是Native方法，PC值为空(未定义)

PC就是一块内存区域。存放着下一条要执行的指令的地址

**Java虚拟机栈:**

*Java方法运行过程的内存模型,由栈帧组成,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.每个线程对应一个java虚拟机栈,生命周期和线程同步, 每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程*

动态扩展内存。

如果线程请求的栈深度太大,则抛出StackOverflowError

如果动态扩展时没有足够的大小,则抛出OutOfMemoryError

**本地方法栈:**

与java虚拟机栈功能相似,为native方法服务

**堆:**

存放对象实例

* 老年代 ： 三分之二的堆空间
* 年轻代 ： 三分之一的堆空间
  + eden区： 8/10 的年轻代空间
  + survivor0 : 1/10 的年轻代空间
  + survivor1 : 1/10 的年轻代空间

**方法区:**

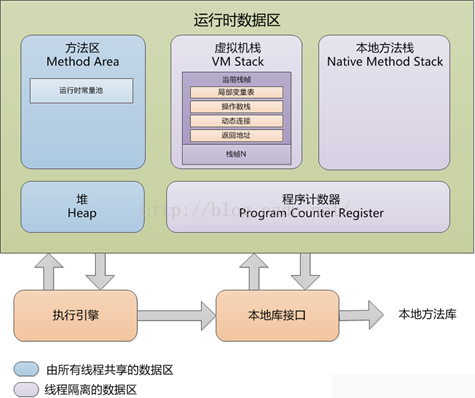
存放虚拟机加载的类信息、常量、静态常量等

元数据区

直接内存(Direct Memory):

本地方法库直接在java堆之外申请的内存区域

好处就是避免了在java堆和native堆直接同步数据的步骤.并不是由JVM来管理



### 垃圾回收

#### ⒈垃圾收集算法

⑴标记-清除算法(清理再压缩)(Mark-Sweep)

标记阶段标记出所有需要回收的对象，清除阶段回收被标记的对象所占用的空间

缺点:内存碎片化

⑵复制算法(Copying)

按内存容量将内存划分为等大小的两块。每次只使用其中一块，当这一块内存满后将尚存活的对象复制到另一块上去，把已使用的内存清掉

问题:可用内存被压缩到了原本的一半。且存活对象增多的话，Copying算法的效率会大大降低。

⑶标记-整理算法(Mark-Compact)

将存活对象移向内存的一端。然后清除端边界外的对象

⑷分代收集算法(Generational Collection)

根据对象存活的不同生命周期将内存划分为不同的域,GC堆划分为老生代(Tenured/Old Generation)和新生代(Young Generation)

新生代的特点是每次垃圾回收时都有大量垃圾需要被回收->Copying

老生代的特点是每次垃圾回收时只有少量对象需要被回收->Mark-Compact

#### ⒉垃圾回收器:

HotSpot、G1

#### ⒊线上发送频繁 full gc如何处理??cpu使用率过高怎么办?

如何定位问题?如何解决?说一下解决思路和处理方法。

定位:查看gc日志

原因:程序没释放资源,导致一有请求就会有大对象进入old区,在业务高峰期old区一会就被打满，所以一直进行fgc

解决:程序释放资源,例如流flush()、close()

## 集合

集合的分类?

Collection接口是集合类的顶级接口，子接口有Set和List。

Set中唯一无序。

List有序不唯一，根据索引访问的方式。

Map是Java.util包中的另一个接口。Map包含了key-value对。Map不能包含重复的key，但是可以包含相同的value。

Iterator，所有的集合类，都实现了Iterator接口，这是一个用于遍历集合中元素的接口，主要包含以下三种方法：

1.hasNext()是否还有下一个元素。

2.next()返回下一个元素。

3.remove() 删除当前元素

List和set的区别?

1、List

有序不唯一

查询速度快:基于数组,根据索引查询

插入删除慢:插入或删除数据时，会伴随着后面数据的移动

2、Set

唯一无序

3、Map

键值对、键唯一、值不唯一

ArrayList和LinkList的区别?

说出ArrayList,Vector, LinkedList的存储性能和特性？[重点]

A. ArrayList和Vector都是使用数组方式存储数据，它们都允许直接按序号查询元素，但是插入元素要涉及数组元素移动等内存操作，所以查询 数据快而插入删除慢 ['vekt?] Vector

B. Vector由于使用了synchronized方法，通常性能上较ArrayList差，

C. LinkedList使用双向链表实现存储，按序号查询数据需要前向或后向遍历，但是插入数据时只要记录前后项，所以插入删除较快。

### 13.4.1.HashMap和HashTable的区别?

HashMap HashTable

线程安全 否 是

继承父类 AbstractMap Dictionary

是否有contains方法 否,containsValue和containsKey 三个方法 是

是否允许null值 一个null键,多个null值 不允许null键和null值

遍历方式 Iterator Iterator、Enumeration

hash值 重新计算hash值 对象的hashCode

默认容量 16 11

内部底层数组的容量为2的等次幂 一定 不要求

扩容的倍数 2 2倍+1

HashMap的实现原理?

数据结构:数组+链表

get()

1.bucket里的第一个节点，直接命中；

2.如果有冲突，则通过key.equals(k)去查找对应的entry

若为树，则在树中通过key.equals(k)查找，O(logn)；

若为链表，则在链表中通过key.equals(k)查找，O(n)

get()

1.对key的hashCode()做hash，然后再计算index;

2.如果没碰撞直接放到bucket里；

3.如果碰撞了，以链表的形式存在buckets后；

4.如果碰撞导致链表过长(大于等于TREEIFY\_THRESHOLD)，就把链表转换成红黑树；

5.如果节点已经存在就替换old value(保证key的唯一性)

6.如果bucket满了(超过load factor\*current capacity)，就要resize(扩容)

resize()

新的数组代替已有的容量小的数组,重新计算hash

hashMap的原理，红黑树是什么，hash冲突怎么解决 **美团**

### 13.4.2.HashMap的源码分析

Jdk1.8

## 13.5.反射：

### 13.5.1.理解:

在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；

对于任意一个对象，都能够调用它的任意一个方法和属性

动态获取类的信息、动态调用对象的方法

### 13.5.2.方法:

invoke:调用类中的方法，invoke(class, method)

new instance:静态方式实例化对象

### 13.5.3.应用：

hibernet实体类的映射

---------

# 14.设计模式

## 单例模式

### 1、饿汉式（静态常量）[可用]

public class Singleton {

private final static Singleton INSTANCE = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return INSTANCE;

}

}

优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。

缺点：在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费

### 2、饿汉式（静态代码块）[可用]

public class Singleton {

private static Singleton instance;

static {

instance = new Singleton();

}

private Singleton() {}

public Singleton getInstance() {

return instance;

}

}

这种方式和上面的方式其实类似，只不过将类实例化的过程放在了静态代码块中，也是在类装载的时候，就执行静态代码块中的代码，初始化类的实例。优缺点和上面是一样的。

### 3、懒汉式(线程不安全)[不可用]

public class Singleton {

private static Singleton singleton;

private Singleton() {}

public static Singleton getInstance() {

if (singleton == null) {

singleton = new Singleton();

}

return singleton;

}

}

这种写法起到了Lazy Loading的效果，但是只能在单线程下使用。如果在多线程下，一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。所以在多线程环境下不可使用这种方式

### 4、懒汉式(线程安全，同步方法)[不推荐用]

public class Singleton {

private static Singleton singleton;

private Singleton() {}

public static synchronized Singleton getInstance() {

if (singleton == null) {

singleton = new Singleton();

}

return singleton;

}

}

解决上面第三种实现方式的线程不安全问题，做个线程同步就可以了，于是就对getInstance()方法进行了线程同步。

缺点：效率太低了，每个线程在想获得类的实例时候，执行getInstance()方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接return就行了。方法进行同步效率太低要改进。

### 5、懒汉式(线程安全，同步代码块)[不可用]

public class Singleton {

private static Singleton singleton;

private Singleton() {}

public static Singleton getInstance() {

if (singleton == null) {

synchronized (Singleton.class) {

singleton = new Singleton();

}

}

return singleton;

}

}

由于第四种实现方式同步效率太低，所以摒弃同步方法，改为同步产生实例化的的代码块。但是这种同步并不能起到线程同步的作用。跟第3种实现方式遇到的情形一致，假如一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。

### 6、双重检查[推荐用]

public class Singleton {

private static volatile Singleton singleton;

private Singleton() {}

public static Singleton getInstance() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

Double-Check概念对于多线程开发者来说不会陌生，如代码中所示，我们进行了两次if (singleton == null)检查，这样就可以保证线程安全了。这样，实例化代码只用执行一次，后面再次访问时，判断if (singleton == null)，直接return实例化对象。

优点：线程安全；延迟加载；效率较高

### 7、静态内部类[推荐用]

public class Singleton {

private Singleton() {}

private static class SingletonInstance {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonInstance.INSTANCE;

}

}

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证初始化实例时只有一个线程。不同的地方在饿汉式方式是只要Singleton类被装载就会实例化，

没有Lazy-Loading的作用，而静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化时，调用getInstance方法，才会装载SingletonInstance类，从而完成Singleton的实例化。

类的静态属性只会在第一次加载类的时候初始化，所以在这里，JVM帮助我们保证了线程的安全性，在类进行初始化时，别的线程是无法进入的。

优点：避免了线程不安全，延迟加载，效率高。

### 8、枚举[推荐用]

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

借助JDK1.5中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。可能是因为枚举在JDK1.5中才添加，所以在实际项目开发中，很少见人这么写过。

优点

系统内存中该类只存在一个对象，节省了系统资源，对于一些需要频繁创建销毁的对象，使用单例模式可以提高系统性能。

缺点

当想实例化一个单例类的时候，必须要记住使用相应的获取对象的方法，而不是使用new，可能会给其他开发人员造成困扰，特别是看不到源码的时候。

适用场合

• 需要频繁的进行创建和销毁的对象；

• 创建对象时耗时过多或耗费资源过多，但又经常用到的对象；

• 工具类对象；

• 频繁访问数据库或文件的对象。

Java数据类型:

基本数据类型:

整数:byte、short、int、long

浮点:float、double

字符类型:char

布尔类型:boolean

引用数据类型:

类、接口、数组

访问级别:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier | 当前类 | Package | 子类 | 其他包 |
| public (公有的) | Y | Y | Y | Y |
| Protected(受保护) | Y | Y | Y | N |
| default(默认) | Y | Y | N | N |
| Private(私有的) | Y | N | N | N |

## UML

# 15.服务器

## 15.1.Nginx

配置:

静态资源配置

跨域

负载均衡算法:

高可用:

nginx+keepalived

### 15.1.1.常用状态码

200：服务器成功返回网页

301 （永久移动） Permanently Moved.重定向, 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302 （临时移动） Temporarily Moved,重定向,服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

403：Forbidden服务器拒绝请求。Nginx:没有访问目录的权限,没有列出目录的权限

404：请求的网页不存在

405：方法不允许Method not allowed

499：客户端主动断开了连接。Nginx

500：服务器遇到错误，无法完成请求。

502：Bad Geteway服务器作为网关或代理，从上游服务器收到无效响应。nginx

503 - 服务不可用.Nginx

504：服务器作为网关或代理，但是没有及时从上游服务器收到请求。

这些状态码被分为五大类：

100-199 用于指定客户端应相应的某些动作。

200-299 用于表示请求成功。

300-399 用于已经移动的文件并且常被包含在定位头信息中指定新的地址信息。

400-499 用于指出客户端的错误。

500-599 用于支持服务器错误。

200 （成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。

201 （已创建） 请求成功并且服务器创建了新的资源。

202 （已接受） 服务器已接受请求，但尚未处理。

203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 （无内容） 服务器成功处理了请求，但没有返回任何内容。

205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206 （部分内容） 服务器成功处理了部分 GET 请求。

300 （多种选择） 针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。

301 （永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。

302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

303 （查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。

304 （未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。

305 （使用代理） 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。

307 （临时重定向） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

400 （错误请求） 服务器不理解请求的语法。

401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 （禁止） 服务器拒绝请求。

404 （未找到） 服务器找不到请求的网页。

405 （方法禁用） 禁用请求中指定的方法。

406 （不接受） 无法使用请求的内容特性响应请求的网页。

407 （需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。

408 （请求超时） 服务器等候请求时发生超时。

409 （冲突） 服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。

410 （已删除） 如果请求的资源已永久删除，服务器就会返回此响应。

411 （需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。

412 （未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。

413 （请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。

414 （请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。

415 （不支持的媒体类型） 请求的格式不受请求页面的支持。

416 （请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。

417 （未满足期望值） 服务器未满足"期望"请求标头字段的要求。

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

501 （尚未实施） 服务器不具备完成请求的功能。 例如，服务器无法识别请求方法时可能会返回此代码。

502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。

504 （网关超时） 服务器作为网关或代理，但是没有及时从上游服务器收到请求。

505 （HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP 协议版本。

## 15.2. Tomcat

### Tomcat优化

# 16.搜索

## elasticsearch

<https://blog.csdn.net/yangwenbo214/article/details/77802331>

https://blog.csdn.net/napoay/article/details/78307269

1. 如何防止脑裂?

**Elasticsearch.yml**

**参数:**

**discovery.zen.minimum\_master\_nodes**

=N/2+1 N是急群中节点的数量

**discovery.zen.ping.timeout:默认3秒,**一个节点在假设集群中的另一个节点响应失败的情况时等待多久.网速慢,调大值

1. 优缺点
2. query和filter的区别?
3. 描述master节点选举算法

选主是ZenDiscovery模块负责的，主要包含Ping（节点之间通过这个RPC来发现彼此）和Unicast（单播模块包含一个主机列表以控制哪些节点需要ping通）这两部分

1. 在并发情况下，Elasticsearch如果保证读写一致

## Solr

## luence

# 17.前端

js闭包

读取函数内部变量 变量保存在内存中

vuejs

组件化 声明式渲染 客户端路由 状态管理 构建工具 指令 拦截器

渐进式理解:

主张最少,轻量,只做该做的事情.没必要一开始就用vue全家桶,根据业务场景选择合适的组件

Angularjs是强主张

双向数据绑定的理解:

数据模型module和视图view,数据变化自动同步view,无需操作dom

组件:

vue-router 路由

vuex状态管理

网络请求:vue-resource/axios

框架:

element-ui

mint-ui

# 18.docker

docker概念:

镜像

容器:镜像的一个运行实例

仓库:集中存放镜像的地方

docker好处:

1. 秒级启动
2. 对资源利用率高,一台主机同时运行数千个docker容器
3. 性能好，开销小

安装

yum install -y docker

阿里云加速

镜像:

1.获取镜像:

docker pull redis:3.0

2.查看镜像

docker images

3.删除镜像

-f 表示强制删除,正在运行的

docker rmi id –f

4.搜索镜像

docker search name

例如:docker search redis

容器:

1. 查看容器列表

docker ps

docker ps –a

1. 创建容器

docker create -p 16379:6379 --name redis redis:3.0

1. 启动

docker start 容器id(前几位)

停止

docker stop 容器名或容器id

docker kill 容器名或容器id

删除

dokcer rmi 容器名

进入容器

docker exec

# 算法/数据结构

手写链表等常用数据结构

二分查找 冒泡排序算法

# 项目问题:

1).在项目中充当什么角色?

2).项目中遇到最复杂的问题，如何解决?

3).如何排查技术问题?

fullGC过于频繁的原因和解决方案

<https://blog.csdn.net/wangshuminjava/article/details/80907140>

4).项目技术架构图,手绘

# 架构设计(高级)

如何让你设计一个秒杀系统,如何设计?

如何设计一个消息中间件,你会从哪些方面来考虑?核心的架构以及数据结构设计?

如果让你来负责一个电商双十一大促系统,你会如何来考虑和设计?

我们公司有这样一个业务场景xxxx,我给你画个图,YYYY,就根据这样的一个场景以及面临的问题,如果让你来设计这个系统,你会如何考虑?

建议:了解面试公司的业务场景,大概用户量

关注公司的官方社区或者公众号,看看技术分享

针对公司的具体业务设计能力,没有思路,结合常规设计思考套用

# 人生

职业规划?

如何学习?

如何解决问题?