### 基础

hashMap 源码 实现原理 java8 对hashmap做的优化

1.存储方式：哈希散列表 以数组+链表的形式

数组存储key的hash值 后边链表存储value 如果hash值相同，会把value值放在链表的next下方

加入了红黑树 当链表长度大于8时 转换为红黑树结构

2.HashMap扩容，为什么是2的倍数

懒加载，构造完对象后，只要不put，不会扩容

初始值 是根据初始量（16）+负载因子（0.75）决定阀值 当数组长度大于12时 会申请新的数组

当put数据时，会判断集合长度 和阀值，大于就会调用resize进行扩容(<<1 乘以2倍)

元素table索引值也会变，从put方法看出，元素索引为器hash值的后N为确定，当扩容后，元素hash值的后N+1位确定，两种情况：

1. e.hash&size(扩容前数组长度) =0，不变
2. 为1 ，调整至2倍位置

资源浪费有关，在map中好多n-1&hash 运算，当n为奇数时，尾数永远为0，好多尾数为1的位置就永远为空

3.hashMap hashTable ConcurrentHashMap区别

hashMap 线程不安全

hashTable ConcurrentHashMap 线程安全

tashTable每步操作都加synchornized关键字 慢

后者分段锁 table每个数据加锁

4.高并发下 hashMap 容易造成链表回路问题

5.java4中修饰符范围

Private 私有的 当前类访问

Protect 受保护的 当前类 及 类的继承关系类

Defalut 包访问 相同包下

Public 公共的

6.Object类中方法

hashCode 获取对象哈希值

Equale 判断两个对象是否相同

toString 返回stinrg对象

getClass 发挥class对象

Wait 当前线程进入等待

Notify 随机让一个持有对象锁的线程获取操作权限

notifyAll 所有。。。。

Finalize 垃圾回收 匿名对象回收之前调用

1. 接口和抽象区别

接口 描述的是动作 方法都是abstract的 不能有构造器

实现接口的类必须把所有抽象方法都实现

只能被public修饰

抽象 描述根源 不能被实例化可以被继承 能有构造器 方法可以有具体实现方法

实现抽象的类 有未被实现的抽象方法也是抽象类

修饰符没有限制



1. Java 传值 传引用区别

基础数据类型 传值

其他类型对象 传引用

1.对象就是传引用

2.原始类型就是传值

3.String，Integer, Double等immutable类型因为没有提供自身修改的函数，每次操作都是新生成一个对象，所以要特殊对待。可以认为是传值。

9. arrayList 循环遍历过程中，删除会有问题?

普通for循环 可能有删除不干净问题

增强型for循环 报错 并发错误java.util.ConcurrentModificationException

ArrayList remove方法 底层有个fastRemove方法，有个modCount方法

使用iterator自带的remove方法

10.transactional 注解

@Transactional 注解只能应用到 public 可见度的方法上。 如果应用在protected、private或者 package可见度的方法上，也不会报错，不过事务设置不会起作用

默认情况下，Spring会对unchecked异常进行事务回滚；如果是checked异常则不回滚

再通俗一点：你写代码出现的空指针等异常，会被回滚，文件读写，网络出问题，spring就没法回滚了

检测为啥不回滚：

1、检查你方法是不是public的

2、你的异常类型是不是unchecked异常   
如果我想check异常也想回滚怎么办，注解上面写明异常类型即可

@Transactional(rollbackFor=Exception.class)

类似的还有norollbackFor，自定义不回滚的异常

3、数据库引擎要支持事务，如果是MySQL，注意表要使用支持事务的引擎，比如innodb，如果是myisam，事务是不起作用的

4、是否开启了对注解的解析

<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"/>

5、spring是否扫描到你这个包，如下是扫描到org.test下面的包

<context:component-scan base-package="org.test" ></context:component-scan>

6、检查是不是同一个类中的方法调用（如a方法调用同一个类中的b方法）   
7、异常是不是被你catch住了

### JVM

1. jvm内存结构

方法区 常量池 字段 方法信息 方法字节码

堆 对象、实例

栈 线程对应栈 方法对应栈帧

本地方法栈 调用c/c++ native方法

程序计数器 线程计数 保存线程的内存地址

1. Jvm方法栈工作过程 方法栈和本地方法栈 区别

每个方法被调用都会创建一个栈帧，用户存储局部变量、操作栈、动态连接、方法出口等信息，每个方法被调用直至执行完过程就对应一个栈帧在虚拟机入栈到出栈的过程

1. Jvm 栈 和堆 关系

堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。句柄

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候，才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

1. Java是否可以直接GC内存

System.gc

Runtim.getRuntion().ge()

1. Java类加载过程

类加载到jvm中，生命周期：加载、验证、准备、解析、初始化、使用、卸载

类加载过程 是加载、验证、准备、解析、初始化

加载：类加载器根据类的全限定名读取类的二进制字节流到jvm，并保存到运行时数据区的方法区，生成一个class对象实例

验证：验证jvm规范 格式 语义 操作等

准备：先为类中所有静态常量分配内存空间，为其设置一个初始值，分配内存并设置类比那辆初始值的阶段，这些内存在方法区中进行分配

解析：对类的字段，方法等东西进行转换

初始化：

主动引用

有且只有下面5种情况才会立即初始化类，称为主动引用：

new 对象时

读取或设置类的静态字段（除了 被final，已在编译期把结果放入常量池的 静态字段）或调用类的静态方法时；

用java.lang.reflect包的方法对类进行反射调用没初始化过的类时

初始化一个类时发现其父类没初始化，则要先初始化其父类

含main方法的那个类，jvm启动时，需要指定一个执行主类，jvm先初始化这个类

其他对类的引用 称为被动引用，加载类时不会进行初始化动作

子类继承父类时的初始化顺序

   1.首先初始化父类的static变量和块，按出现顺序

   2.初始化子类的static变量和块，按出现顺序

   3.初始化父类的普通变量，调用父类的构造函数

   4.初始化子类的普通变量，调用子类的构造函数

双亲委派原则 引导类加载器、扩展类加载器、系统类加载器、自定义类加载器

过程：一个类加载器收到类加载请求，它不会先去加载，把这个请求委派给父类加载器区完成，每一层类加载器都这样，因此所有加载请求都是传送到最顶层的启动类加载器。只有父类加载器反馈无法完成，子类加载器才会去加载

好处：有效保证一个类的全局唯一性，当程序中出现多个限定名相同的类时。类加载器在执行加载时，始终会加载其中一个类

### 多线程

1. Lock接口的实现类，使用场景

可重入锁 ReenTrantLock

1已加锁，不再重复加载

交互界面重复点击

2 该操作已经在执行，尝试等待一段时间，等待超时则不执行

1. 多线程实现方式
2. 继承thread 重写run方法 （实现了runnable接口）
3. 实现runnable
4. 实现Callable 重写call方法

用ExecutorService、Callable、Future实现由返回结果的多线程

Runnable和Callable区别：

1. Callable规定的方法是call()，Runnable规定的方法是run()。其中Runnable可以提交给Thread来包装下，直接启动一个线程来执行，而Callable则一般都是提交给ExecuteService来执行。   
   (2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得   
   (3)call方法可以抛出异常，run方法不可以   
   (4)运行Callable任务可以拿到一个Future对象，c表示异步计算的结果。

3.volitile 关键字作用 原理

保证数据可见性 有序性

可见性：加volitale关键字 变量会强制写入主内存中，读取会直接读取主内存

有序性：提供一层内层屏障，变量前后的数据不会重排

4.synchornized 用法 优缺点

修饰方法 代码块 静态方法 类

作用对象是非静态的，它取得的锁是对象

是一个静态方法或类，他取得的锁是对类，该类的所有对象是同一把锁

Synchronized 缺陷：

锁的粒度太大，当被synchronize修饰，一个线程获取到锁，其他线程只能一直等待，会有两种情况释放锁：

1）获取锁线程执行完，正常释放锁

2）线程发生异常，等jvm让线程释放锁

这个等待的过程影响效率

多线程执行时，读和写是有冲突的，但是读和读是没有冲突的，但采用synchronized修饰的，也会导致只能一个先读，等线程读取完才能下一个线程读。

区别：

Lock 是一个接口类 synchronized是java内置关键字

Lock是用户手动释放锁，不然有死锁，synchronized是系统自动释放对锁的占用

5.悲观锁 乐观所 优缺点 CAS 缺陷 解决

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作

依靠数据库提供的锁机制，将数据处于锁定状态，保证操作独占性

性能问题 慢

乐观锁：假定不会发生并发冲突，只有在提交操作时检查是否违反数据完整性

通过数据版本记录机制实现，对数据加一个版本字段控制，读取更新都会对版本号更新，校验提交的版本号是否大于数据库中当前的版本号

增加用户读取数据的次数

CAS 不太懂 compare and swap

1. ABC 三个线程 保证顺序执行

ReentrantLock 实现 lock.lock lock.unlock

1. 线程状态

初始 就绪 执行 阻塞 结束

**sleep()、wait()、yield()、join()的区别**

Join()：一种特殊的wait，当前运行线程调用另一个线程的join方法，当前线程进入阻塞状态直到另一个线程运行结束 加赛 调用哪个线程的join就先执行那个线程  
        yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行

sleep()方法是使线程停止一段时间的方法。在sleep 时间间隔期满后，线程不一定立即恢复执行。这是因为在那个时刻，其它线程可能正在运行而且没有被调度为放弃执行，除非“醒来”的线程具有更高的优先级，正在运行的线程因为其它原因而阻塞。

wait()是线程交互时，如果线程对一个同步对象x 发出一个wait()调用，该线程会暂停执行，被调对象进入等待状态，直到被唤醒或等待时间到。 (notify())

Wait() notify 是Object类提供的方法

1. ThreadLocal 了解 原理

线程本地变量

ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

每个运行的线程都会有一个类型为ThreadLocal.ThreadLocalMap的map,这个map就是用来存储与这个线程绑定的变量,map的key就是ThreadLocal对象,value就是线程正在执行的任务中的某个变量的包装类Entry.

get函数就是从当前线程的threadlocalmap中取出当前线程对应的变量的副本【注意，变量是保存在线程中的，而不是保存在ThreadLocal变量中】。当前线程中，有一个变量引用名字是threadLocals，这个引用是在ThreadLocal类中createmap函数内初始化的。每个线程都有一个这样的threadLocals引用的ThreadLocalMap，以ThreadLocal和ThreadLocal对象声明的变量类型作为参数。这样，我们所使用的ThreadLocal变量的实际数据，通过get函数取值的时候，就是通过取出Thread中threadLocals引用的map，然后从这个map中根据当前threadLocal作为参数，取出数据。

### 数据库

1. 常见数据库优化

Sql语句方面：

查询优化 避免全表扫描 在where orderby 字段列上加索引

避免在Where中对字段进行null判断

避免where中使用!= 或者 <> in not in 等操作符

Select 查询不要用\* 把查询字段名加上

避免在where中使用函数操作 和表达式操作 导致引擎放弃索引使用全表查询

1. 索引优缺点 什么字段可以建索引

优：提高系统性能 加大数据检索速度

缺： 创建和维护索引要消耗时间 随着量多而增加

占用物理空间

修改表字段时，索引也要动态维护，降低数据库维护速度

经常搜索的列上，加快搜索速度

主键列上 强制该列的唯一性

经常连接的列上 一些外键 加快连接速度

经常排序的列上

经常where中使用的列

很少使用的列 数据值很大的列 当前性能已经很好 都不需创建索引

索引 查询 遵循 最左前缀 原则 针对多列索引

1. 数据库连接池

分配 管理 释放数据连接

连接过程：

装载数据库驱动程序 DrviceManager

通过jdbc建立数据库连接 Connection Statement创建状态 resultSet 结果

访问数据库，执行sql

断开连接

最小连接--应用启动后随即打开的连接数以及后续最小维持的连接数。  
 最大连接数--应用能够使用的最多的连接数  
 连接增长数--应用每次新打开的连接个数

常见连接池：

dbcp:可以设置连接最大、最小、连接等待时间 稳定性可以 速度慢 并发下不太好 不提供监控连接池功能

C3p0:可以设置连接最大、最小、连接等待时间 稳定性好 并发下也很好 也不停工连接池监控

Proxool：稳定性有问题 但可以监控连接池

1. Durid 常用配置

连接池

dataSource = new DruidDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");//可不配 根据url识别

dataSource.setUsername("root");

dataSource.setPassword("11111111");

dataSource.setUrl("jdbc:mysql://127.0.0.1:3306/jspdemo");数据库url

dataSource.setInitialSize(5); //初始化连接数

dataSource.setMinIdle(1); //最小 连接数量

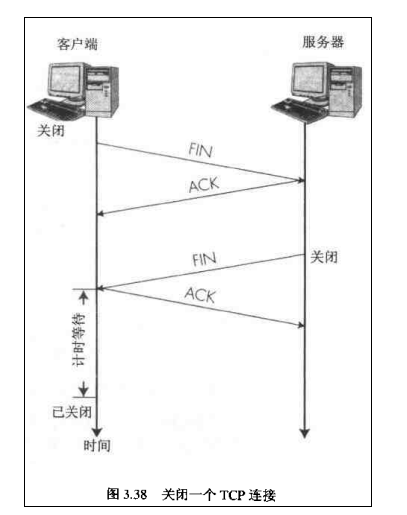
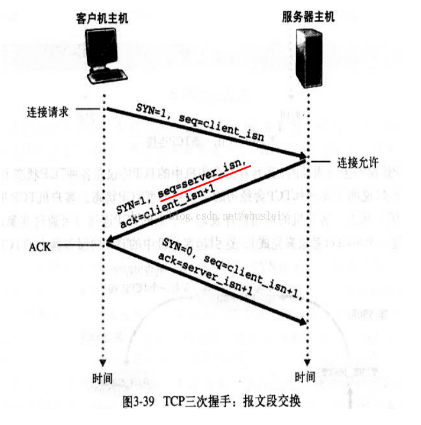
dataSource.setMaxActive(10);// 最大连接数量

dataSource.setFilters("stat");// for mysql 扩展插件 监控统计插件

dataSource.setPoolPreparedStatements(false);

### 计算机网络

1.Tcp udp区别



三次握手：

客户端client发送请求连接报文syn，server接受到连接后回复ack(响应)+syn报文，client接收到返回的报文后回复server发送ack报文，tcp连接就建立了。

为啥不是二次？

三次握手是为了解决网络延迟重复发送分组问题，client发送的报文没有丢失，在网络节点滞留，以至延误到连接释放后的某个时间到达server，本来是一个已经失效的连接，但server误以为是client发送的新请求，向client发送报文，同意连接，新开线程提供服务，造成服务器资源浪费

四次挥手：

Client发送中断请求，发送fin报文，server接收到后，但服务端要准备下，发送一个ack给client，client进入FIN\_WAI状态，等service准备好，发送FIN报文，client接收后，知道可以关闭连接，还是要通知servier可以关闭了(可以重传)，client进入TIME\_WAIT状态，等待一段时间后，没有回复，就关闭连接

1、TCP面向连接 控制传输协议（如打电话要先拨号建立连接）;UDP用户数据报协议是无连接的，即发送数据之前不需要建立连接

2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付

Tcp通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输。如丢包时的重发控制，还可以对次序乱掉的分包进行顺序控制。

3、UDP具有较好的实时性，工作效率比TCP高，适用于对高速传输和实时性有较高的通信或广播通信。

4.每一条TCP连接只能是点到点的;UDP支持一对一，一对多，多对一和多对多的交互通信

5、TCP对系统资源要求较多，UDP对系统资源要求较少。

1. 长连接 短连接

Http协议(应用层)的长短连接 实质是TCP协议(传输层)的长短连接 ip(网络层)

长连接：client方与server方先建立连接，连接建立后不断开，然后再进行报文发送和接收。这种方式下由于通讯连接一直存在。此种方式常用于P2P点对点的通信。

使用场景：监控系统、即时通信（用户登录、发送信息）、即时报价（后台数据变更）

短连接：Client方与server每进行一次报文收发交易时才进行通讯连接，交易完毕后立即断开连接。此方式常用于一点对多点通讯。

使用场景：Web http请求服务

4.连接池 是什么连接

数据库的连接用长连接，如果用短连接频繁的通信会造成socket错误，频繁的socket创建也是对资源的浪费。

### 分布式

1. 分布式事务控制 一点没接触过 放过

传统单库单表的系统 可以用spring transcational控制事务

分布式 数据分库 需要分布式事务控制

1. 分布式锁设计

高并发情况下的同一时间只能被同一个线程执行

分布式锁 CAP理论（任何一个分布式系统都无法同时满足一致性(consistency)、可用性(availability)和分区容错性(partition Tolerance)，最多只能同时满足两项）

1.通过数据库加锁实现 排他锁

数据库性能问题严重

会有很多问题 可能会出现锁表问题

不具备可重入、阻塞锁特性

2.基于缓存实现 redis 代码demo

1）使用setnx加锁，使用expire为锁加一个超时时间，超时则自动释放锁，锁的value值一个随机的UUID

2）获取锁时设置一个获取的超时时间，超过放弃获取锁

3）释放锁，通过UUID判断是否该锁。是的话，执行delete释放锁

3.基于zookeeper实现

1）创建一个目录mylock

2）线程A想获取锁，就先在mylock下创建临时顺序节点

3）获取mylock下所有子节点，获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获取锁

4）线程B获取所有节点，判断自己是不是最小节点，设置监听比自己次小的节点

5）线程A处理完，删除自己节点，线程B监听到时间，判断自己是不是最小节点，是就获取到锁

有个开源框架apache Curator

1. 分布式session设计
2. session复制 同步数据 将一台机器上session复制到其他机器上

机器少 网络流量小 实现简单 有延时性，网络开销

1. Session sticky 粘性session

当用户访问集群中某台机器后，强制指定后续请求都访问同一台机器

机器适中，稳定性没要求 实现简单 一台坏掉容易session丢失，单点故障

1. 缓存集中管理 将session放入某台机器，用户访问时先从缓存中拿数据

机器多 可靠性好 实现复杂、依赖缓存稳定性

1. Session cookie区别 联系

Session保存在服务器客户端不知道 cookie保存在客户端，服务器能知道其中信息

Session保存对象 cookie保存字符串

Session不区分路径，所用session在任何地方都能访问 cookie设置路径，同一网站不同路径下cookie互相不能访问

Session借助cookie正常工作

1. Dubbo组件 作用

Provider 暴露服务的服务提供方

Consumer 调用远程服务的服务消费方

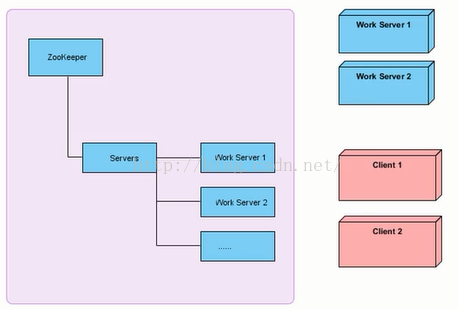
Registy 服务注册和发现的注册中心

Monitor 统计服务的调用次数 调用时间的监控中心

Container 服务运行容器

1. Zookper 负载均衡算法

Zookper 负载均衡原理：



每台WorkServer（后台服务器）启动的时候都会到Server创建临时节点。

        每台ClientServer启动的时候，都会到Server节点下面取得所有WorksServer节点，并通过一定算法取得一台并与之连接。

zookeeper实现负载均衡其实原理很简单，zookeeper 的数据存储类似于liunx的目录结构。首先建立servers节点，并建立监听器监视servers子节点的状态（用于在服务器增添时及时同步当前集群中服务器列表）。在每个服务器启动时，在servers节点下建立子节点worker server（可以用服务器地址命名）,并在对应的字节点下存入服务器的相关信息。这样，我们在zookeeper服务器上可以获取当前集群中的服务器列表及相关信息，可以自定义一个负载均衡算法，在每个请求过来时从zookeeper服务器中获取当前集群服务器列表，根据算法选出其中一个服务器来处理请求

1. Dubbo如何利用接口通信

底层是socket实现的

缓存

框架