# 编程

**调试技巧：**

（1）调试时，假设所有碰到的方法都与结果，跳过去后看是否产生需要的结果，产生了则该方法必然有关（可以跳过很多很多无关方法）。

（2）调试时，点击idea断点，可以设置断点触发条件。

**技术引入要关注的问题**：

每个技术都有优缺点，要考虑引入技术带来的好处和坏处，不要只看好处。

**高可用原理**

无论是java微服务，还是reids集群、kafka等高可用，无外乎两种方式及其组合：

（1）做集群，即做多个副本，每个都是可用的，设置同步机制保证一致性。且有机制保证当服务挂了后，将副本用起来。（如mysql读写分离机制，kafka的leader和follow的选举机制、zookeeper的选举机制）

（2）做分布式，即把整体拆成多个服务，这样某个服务挂了不影响其他服务。

# java数据结构（待完善）

**平衡二叉树**

二叉树的增强，通过保持平衡，降低查询的时间。

时间复杂度O(logN)。

**红黑树**

思想：平衡二叉树。

时间复杂度O(log n)

复杂的操作实现二叉树的平衡。

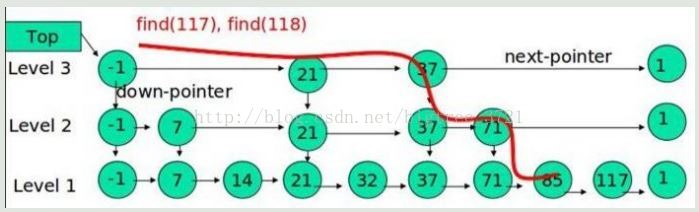
应用：在jdk8中，currentHashMap节点数大于8时，链表变为红黑树存储。

**跳表**

思想：空间换时间。

平均时间复杂度O(log n)。量越大对比其他结构速度越快。

印象：很多层链表构成。底层是全部数据。一层链表的上一层是其简化后的索引，直到最顶层，索引越短。。



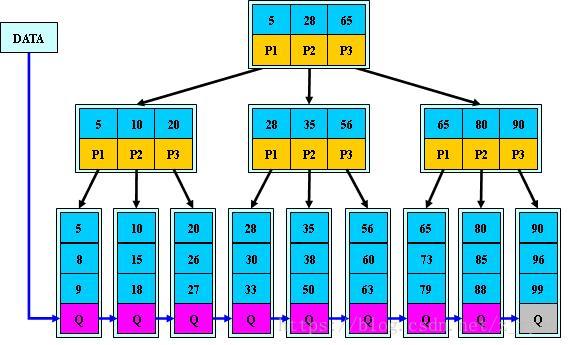
**b+树**

思想：多路平衡二叉树（多路，即每个节点多个索引指向多个路径）

时间复杂度O(log n)。

是一个内存存储数据的数据结构，

应用：如mysql的innodb引擎的索引，是b+树的形式存储的。



# java数据结构api

**TreeMap**

底层用红黑树实现

key相同，后面覆盖前面。

key可以是任意类型，根据hashcode来判断是否同一个key。

打印出来的顺序是key的顺序，如abc，可传入comparator，重写key的排序。

若要根据value排序。则将keyset（就是map的一个key，value节点）存入一个ArrayList，重写list的排序即可。

常用方法：tm.put(key,value);tm.remove(key);tm.get(key);tm.containsKey(key);

**ArrayList**

底层数组实现

ArrayList<keySet<String,int>> al = new ArrayList<keySet<String,int>>();

al.add(obj);al.clear();al.get(key);al.remove();

**WeakHashMap**

key、value可以为空，线程不安全。

当key失去强引用，将其放入弱引用队列，这样就知道了顺便将哪些value清理掉。

其性质适合做缓存（但是假如value被其他地方引用，key没用了把value清掉，其他地方会报错吧？）。

运用：做缓存，当key没用了，其机制会顺便把value清理掉。

# java常见排序（待完善）

**堆排序**

大量数据（百万级别数据）排序，内存占用相对少。

# Redis

**redis持久化方式**

两个，一个是一次保存所有数据到文件（rdb）。一个是将新数据追加到文件（aof）。

（1）RDB（redis dump binary）持久化（默认）

原理是将Reids在内存中的数据库记录定时dump到磁盘上=

优点：

只有一个文件，易备份、易恢复、启动比aof快、性能最大化（子进程负责持久化）。

缺点：

数据量大时，备份慢。若没持久化就宕机，则丢失数据。

（2）AOF（append only file）持久化

原理是将Reids的操作日志以追加的方式写入文件

优点：

策略多，每秒同步、每次修改同步、不同步。异步，性能高，最大限度避免数据丢失。

写入文件是追加模式，不损坏文件且结构清晰。

缺点：

aof文件大，恢复慢启动慢。每秒同步导致运行效率低于rdb。

**redis集群**

配置简单

**redis单线程模型**

与客户端建立连接后，用socket通信。专门有程序负责轮询所有客户端请求，并且不阻塞，将所有请求依次压到队列。专门有程序将一个请求处理完然后从队列拿另一个请求。

**为什么单线程性能还这么高**

因为redis不阻塞，所有请求都放队列。而专门处理请求的程序是存内存操作的，所以性能高。

**支持的类型**

string（字符串），hash（哈希），list（列表），set（集合）及sorted set

String：简单的key-value结构

hash：存对象，可操作对象值key=11,value = {“id”: 122, “name”:”xiaoming”}，每次可以修改其中的id或name

list：key = xx, value = [xxx,ffff,sss]有irange命令对应的api，实现高性能分页。

set：无序集合，自动去重。

1. 集群里面的全局去重。
2. 可以对两个set做交集并集，快速找到两set重复或不重复元素。

sorted set：自动去重，可排序。保存时需要传排序值。可以用命令zrevange board 0 3取出排名前四位。zrank board lisi 返回lisi的排名。总之有很多用法。

**1、常用操作:**

StringRedisTemplate redisTemplate;

（1）

同时绑定key和设置值

redisTemplate.opsForValue().set(key，value);

**2、特点：**

Redis单线程，若数据库有几千万的数据，如果用了 keys \*会很消耗资源，导致其他服务失败，服务宕机，所以要禁用keys命令（在配置文件中的rename-mand，将keys命令重命名成其他单词等）

五种数据结构：

String（Map<String,String>）、list(Map<String,List<Stirng>>)、set(Map<String,Set<String>>)、sort\_set、hash(Map<String,Map<String,String>>)

在分库分表中，redis可以做id的自增长

SpringBoot提供了redis客户端：RedisTemplate

**3、RedisTemplate和StringRedisTemplate区别：**

RedisTemplate使用的是JdkSerializationRedisSerializer，存入数据（key和value）会先序列化成字节数组再存入Redis数据库，可读性非常差。但是如果你的数据是复杂的对象类型，而取出的时候又不想做任何的数据转换，直接从Redis里面取出一个对象，那么使用RedisTemplate是更好的选择。

 StringRedisTemplate（或者RedisTemplate<String, String>）使用的是StringRedisSerializer，序列化成Stirng再存入如Redis数据库，可读性好。

**redis实现分布式锁（推荐使用zk实现，redis性能和复杂度和健壮性待提高）**

1. 第一种方式：

setNx设置一个key，值要唯一，设置成功返回ok，反之返回null。

操作成功释放锁：判断key的value是否是自己的值，是则删除否则不删。（类似mysql根据版本值实现乐观悲观锁）

缺点：redis挂掉或者主从复制没同步，会导致锁失效。

# Elasticsearch:

**elasticsearch集群、节点、索引、文档、类型、分片、副本：**

集群cluser即elasticsearch集群，默认名称elasticsearch。

节点node是一个elasticsearch的实例，本质是一个java进程，是集群的一个单独的服务器。

索引index，好比mysql的数据库概念。

类型type，好比mysql的表

文档document，好比mysql中的行

分片shard：通过主分片，将数据分布在集群上，每个分片内容一致。。一个index索引，可以有多个分片。

副本replica：给分片shard创建的副本数

**lucene是啥？倒排索引？全文检索？**

**es属于分布式类型的存储**

假设es分成3个shard（3个shard构成完整的数据），es集群是3台，即三个节点。每台都有同样的3个shard，数据是通过hash决定分配在哪个shard里面的。

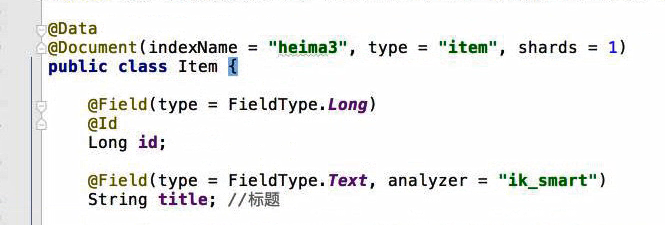
当写数据时：只能往primary shard里面写，而primary shard可能在三台机器上。es的3个shard，都各自有primary shar（主分片）d和replica shard（副分片）。且primary shard宕机后，从replica shard里面选举一个座位primary shard，所以三个分片的primary shard不一定在一台机器上，到这里可以看出是分布式的，而非集群。

当读数据时：可以从primary 和 replica 中读取。

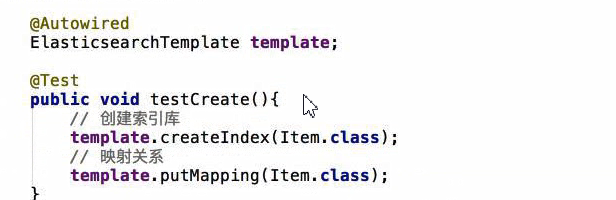
master节点：master节点负责取消掉宕机的节点上的primary shard标记和标记新的primary shard。

**Spring操作elastricsearch**

1、实体类



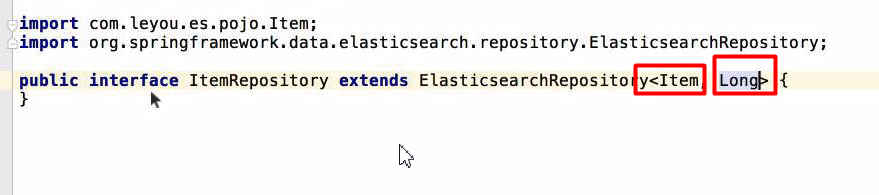
2、根据实体类的定义，创建索引：



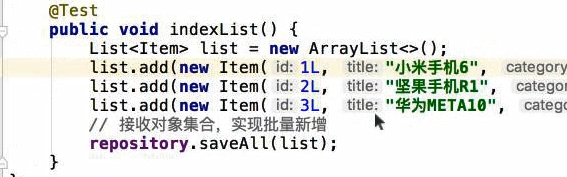
3、添加数据

spring提供了简单的增删改查接口（ElasticsearchTemplate做原生复杂查询才用）。

（1）、继承接口（传实体类，id类型）

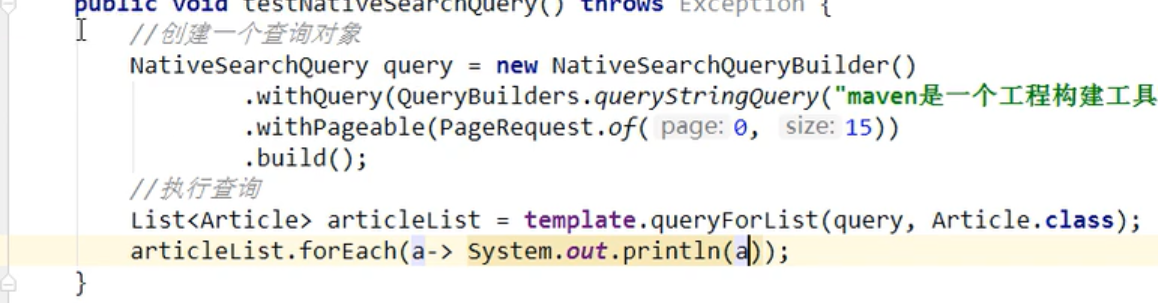


（2）、save方法添加或修改数据（根据id增加或修改）

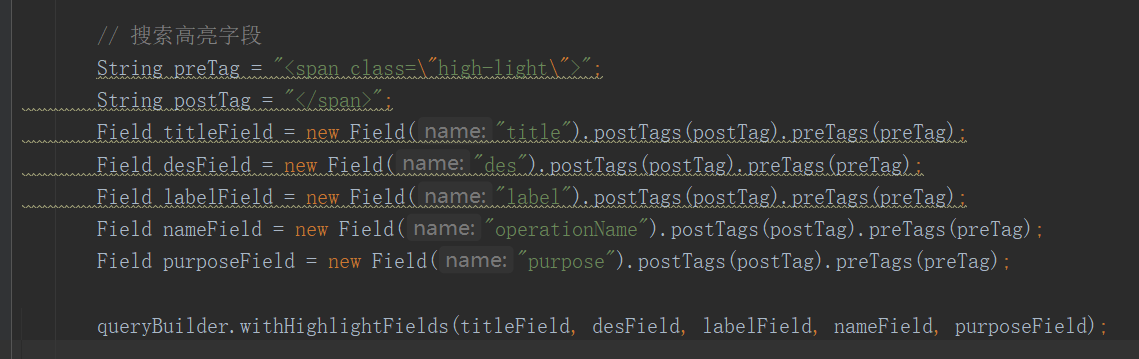


4、使用原生的查询（复杂查询。）

使用NativeSearchQueryBuilder类构造查询条件，使用template执行。



搜索字段高亮显示：



**ik分词器：**elasticsearch-analysis-ik-6.2.4：

如把“我是中国”，分成“我”、“是”、“中国”，估计是用于查找。

映射是定义文档的过程，文档包含哪些字段，这些字段是否保存，是否索引，是否分词等

**工具：**

kibana，操作原生的es。

ik分词器：分词

**创建映射字段（相当于创建）：**

PUT /索引库名/\_mapping/类型名称

{

"properties": {

"字段名": {

"type": "类型", //类型有许多：text(可以分词)、keyword（不能分词）、

"index": true，//true则该字段会被索引，false不索引，默认true

"store": false，//es会存一份源数据到\_source字段中，无须设置为true而多存一份数据

"analyzer": "分词器" //如"ik\_max\_word"

},

“字段名”：{

}，

……

}

}

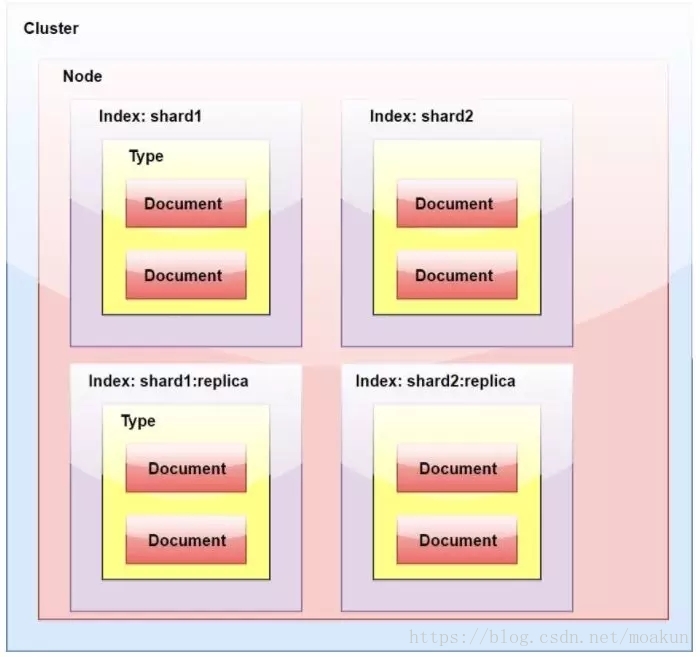
**聚合aggregations**

方便地对数据进行统计、分析。

Es的聚合包含多种数据类型，常用的有两种，一个叫桶，一个叫度量。

如把人按照国家分桶：中国桶、英国桶、美国桶。。。。。。

**结构：**cluster：集群，node：节点。shard：分片



# 多线程

多线程的代码中，任何一句都可能有线程同时执行。

**线程一些知识：**

1、创建一个线程的方法：

1）继承Thread类

2）实现Runnable接口

3）实现Callable 接口

2、像普通方法一样调用多线程类的方法：run();

3、其他：

Main（）方法执行后的线程名为main

Jdk1.7之后，可以Thread.sleep(10\_1000)，这样设置时间

4、线程生命周期：new（创建实例）、runnable（可运行）、running（运行中）、block（阻塞）、terminal（结束）

**常见创建线程**

继承Thread 创建线程

1、构造方法可以传ThreadGroup，统一管理组下线程,默认加入父线程的组。

2、有个构造方法能给线程指定虚拟机的栈长度。但是指定栈太长占用资源，导致能创建的线程变少，甚至程序崩溃。

3、Thread创建线程，线程名默认从Thread-0开始

Runnable创建线程

1. 可以用策略模式，实现runnable跑不同的需求。

Callable 接口：

1. 有返回值

**线程池ThreadPoolExecutor**

1、原理：

（1）、一开始无线程，有任务则创建核心线程，可重入锁（ReentrantLock）防止重复创建。

（2）、hashset存储核心线程。

（3）、核心线程无剩余，多余任务放阻塞队列

（4）、阻塞队列满了创临时线程执行。

（5）、队列满，无剩余线程执行，触发拒绝策略

2、常用方法

submit（new Runnable…） 方法增加线程

execute（new Runnable…）加入并启动线程

shutdown方法关闭线程池。

shutdown方法执行后，再调用execute方法会报错。

3、对象创建及参数

ThreadPoolExecutor threadPoolExecutor = ThreadPoolExecutor( int corePoolSize, int maximumPoolSize,long keepAliveTime, TimeUnit unit,BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler )

参数解析：

corePoolSize: 核心线程数量，可以类比正式员工数量，常驻线程数量。

maximumPoolSize: 最大的线程数量，公司最多雇佣员工数量。常驻+临时线程数量。

workQueue：多余任务等待队列，再多的人都处理不过来了，需要等着，在这个地方等。

keepAliveTime：非核心线程空闲时间，就是外包人员等了多久，如果还没有活干，解雇了。

TimeUnit：空闲时间的单位

threadFactory: 创建线程的工厂，在这个地方可以统一处理创建的线程的属性。每个公司对员工的要求不一样，恩，在这里设置员工的属性。

handler：线程池拒绝策略，什么意思呢?就是当任务实在是太多，人也不够，需求池也排满了，还有任务咋办?默认是不处理，抛出异常告诉任务提交者，我这忙不过来了。

workQueue我常用的是：java.util.concurrent.ArrayBlockingQueue

handler有四个选择：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出异常（默认）。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

**线程基本操作**

1. 启动一个线程的方法：start();

2、设置成守护线程：如main方法中启动一个线程t，然后t设置成守护线程（t.setDaemon( true )），则主线程main断掉前会关闭t。

3、t.join()方法： 效果是等t执行完再执行当前线程（join，加入，成为其中一员）。

Thread.currentThread().join();（等待当前线程自己结束，类似死循环）

4、t.interrupt()中断线程。

5、shutdown()方法：a中设置一个守护线程b。a.shutdown(2000)，2秒后去停止b，但是b才1秒就停止了，a这个方法不会等两秒,也是1秒结束。

**线程同步通信技术：**

用到共同数据（包括同步锁）的若干方法，定义在一个类上。体现高内聚、健壮性。

1. wait()和notify()、notifyAll

synchronized（Lock）{

lock.wait();//放弃cpu执行权。进入block（阻塞）状态，等其他地方调用Lock.notiy()通知停，才会继续执行。

}

但是：Lock对象的notify并不知道通知的谁，只有两个线程，互相wait和notify的话问题不大，多个线程会有问题。如多线程存在生产者p1，p2，应该是p1通知消费者继续消费，但是通知了p2的话，那p1，p2都会进入wait，没生产者生产产品，消费者也会wait等待产品，导致所有线程都wait。解决方法：使用notifyAll方法

**Wait和sleep的区别：**

我的理解：

wait（2000），等待的意思，等其他线程唤醒自己。原理是释放锁（wait需要用到monitor,只能在synchronized中使用），让其他线程执行,需要让其他人唤醒自己。

Sleep是Thread对象本身要休眠，不涉及锁的释放。

1、wait是所有对象的方法，但是sleep是Thread的方法。---基于xxx对象的方法

2、sleep不是基于monitor而wait基于monitor（wait需要在synchronized中执行）。--用在何处

3、sleep不会释放锁（monitor），但是wait会释放锁且将锁加入到monitor等待队列—原理

4、sleep不需要唤醒，但是wait需要。--其他

异常捕获：

使用try（）catch{ e.printStack }

**ThreadLocal类：**

static ThreadLocal<Integer> tl = new ThreadLocal();

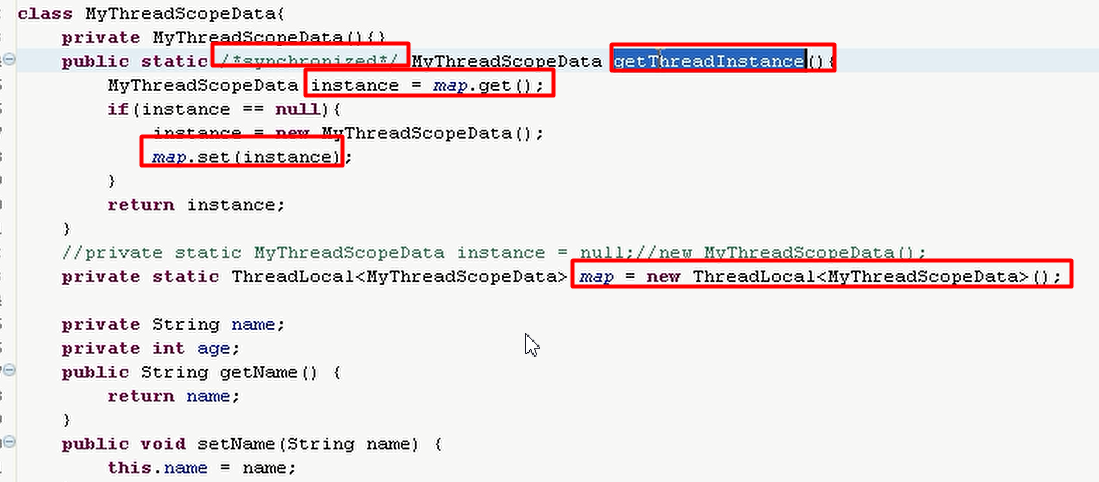
每个线程拷贝一份变量给它自己用。各个线程间不会互相干扰变量。

ThreadLocal.clear()可以在使用后更快释放内存。但是谁知道什么时候结束。

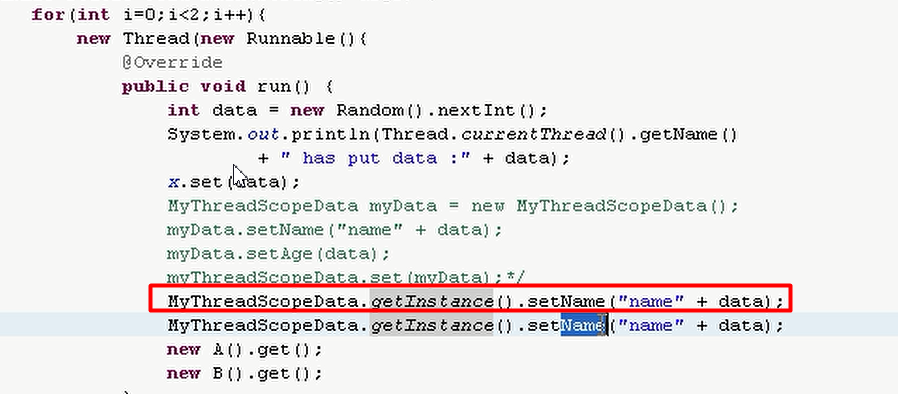
最佳实践：

自我感觉：里面存ThreadLocal类的map就是关键，每个线程有自己的map，所以都会自己创建一份MyThreadData保证互不干扰，此时单利模式只对自己有效，实际上每个线程有一个MyThreadData类的实例。

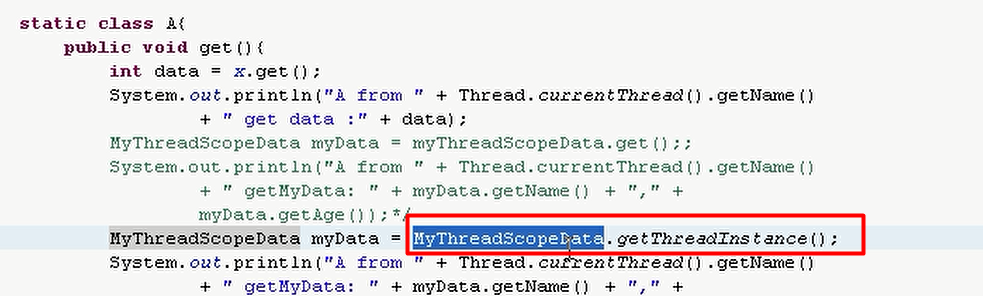
单例模式创建变量类，内置ThreadLocal对象，类型为变量类自身，保证单例模式下各自有一份数据：



存数据：用getInstance()获取单利对象



取数据：用getInstance()获取单利对象



**多线程同步工具：**

semaphore：控制同时访问自身的线程个数。

ciclicBarrier: 等线程集合好后一起执行。

countDownLatch：调用countDown方法则计数器减1，到0时开始执行。

Exchanger：用于两个线程间数据交换，等双方都拿着数据时才交换。

BlockingQueue: 阻塞队列，队列满时放数据的队列会阻塞，反之取数据的会阻塞。

各种同步集合类。

**使用线程池注意**

线程池要有自己的名字，不用系统自动生成的名字：

原因：子线程报错不会在主线程的catch中catch到的，报错信息第一句就是子线程名，要是系统分配的名字就完了，完全不知道哪里出错了。

# 线程锁

**公平锁概念：**

严格按照先来后到循序获取锁

**常用的锁对象：**

1. Lock接口—lock锁

Lock lock = new ReentrantLock（）-- ReentrantLock：可重入锁

try{

lock.lock()

}finally{

lock.unlock();

}

原理（公平锁思想）：

volatile修饰的表示锁状态的变量：

等待条件的队列a：

等待锁的队列b：

获取锁的一个线程节点用完后，唤醒其后面一个节点，依次进行、

某节点获取锁后

把a的首节点放入b的尾部，同时唤醒该线程，让其执行。

被唤醒线程的队列：

等待执行

1. ReadWriteLock接口--读写锁

使用：

ReadWriteLock r = new ReentrantReadWriteLock();

r.readLock().lock();

r.writeLock().lock();

try{

}finally{

r.readLock().unlock();

r.writeLock().unlock();

}

读写锁实现一个缓存系统：

private Map<String, Object> cache = new HashMap<>();

private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

public Object getData(String key) {

lock.readLock().lock();//上读锁

Object value = null;

try {

value = cache.get(key);

if (null == value) {

lock.readLock().unlock();//释放读锁否则无法上读锁

lock.writeLock().lock();//上读锁，去数据库读

try {

if(null == value){ //防止有线程获取锁后执行到这一步导致再次读库

value = "xxx";

}

} finally {

lock.writeLock().unlock();//去除写锁

}

lock.readLock().lock();//上读锁否则多线程出问题

}

} finally {

lock.readLock().unlock();

}

return value;

}

读写锁思想：只有写锁和写锁同时访问不互斥，能同时进行，其他互斥。

读写锁实现思路：用int变量记录当前状态：正在读、等待读、正在写、等待写。

（1）当在读时，允许其他线程读，拒绝写。

（2）正在写时，其他线程读和写都拒绝。

1. Condition类，类似传统线程的wait和notify，但是condition只能唤醒同一个condition对象。

用法（jdk有更好的例子）：

Lock lock = new ReentrantLock();

Condition condition1 = lock.newCondition();

Condition condition2 = lock.newCondition();

try {

while(flag){

condition.await();

condition2.signal();

…….

}

condition.signal();

condition2.await();

} catch (InterruptedException e) {

}finally {

lock.unlock();

}

原理（公平锁思想）：

等待条件的队列a：

等待锁的队列b：

获取锁的一个线程节点用完后，唤醒其后面一个节点，依次进行、

获取锁的节点调用signal方法

把a的首节点放入b的尾部，同时唤醒该线程，继续执行。

被唤醒线程的队列：

等待执行）

**Synchronized（锁对象）{代码。。。}**

**1、锁对象：**

(1)、可以是任意对象。(2)、同一锁对象的线程会同步，反之不同步。

原理：貌似是使用监视器锁（monitor，其实现依赖互斥锁），在要同步的代码块前后加类似开始（monitorenter）和结束（monitorexit）的标记。jvm试着用无锁实现，不行再一级级直到重锁（互斥锁）实现。

**2、同步方法：**

Synchronized修饰方法

Synchronized（this）{代码。。。}

锁对象是当前对象this

**3、静态同步方法**

Synchronized修饰静态方法

Synchronized（Test.class）{代码。。。}

静态方法在程序运行时就初始化了，比this对象早，所以其锁对象不是this。

锁对象：class文件对象

**多线程无锁设计：**

原则：

1. 不可变对象一定是安全的（如定义成final的对象）
2. 可变对象一定是不安全的。

例子：类和该类对象都定义成 final，不提供任何set方法。则该类是不可变对象，无须加锁。如String类，但是自己实现的不可变对象容易有问题，如包含list，list自己有set方法

**多线程Future设计模式实现思路：**

类似预定蛋糕，预定后可以做其他事，等人送蛋糕过来。

# java集合：

在迭代（遍历）过程中，对集合进行操作会出现问题而且难发现（在修改处用sleep可重现）。而且在多线程时更容易忽略这种问题。

**concurrentHashMap：**

jdk1.7中，用数据进行分段（16段），每个段加一个锁保证线程安全。

jdk1.8用volatile关键字、CAS操作和Synchronized关键字实现线程安全。java8的结构之前的不一样了，节点大于8个时用红黑树实现，数据结构与HashMap一样，即原来一个hash的key，对应一个链表，现在对应一个红黑树。

**HashMap、HashTable、HashSet：**

HashSet底层是用HashMap实现的，只是用了他的key而已，value不用，value随便搞个object，定义成static final不允许修改。

HashTable是key和value不能为null（底层代码有判断，是null就报异常）的HashMap；

**list和set**

Java集合分为value（list、set）、key-value（map）

List：

链式结构，有序可重复。

Set：

哈希表结构，无序，Set集合在调用add方法的时候，add方法会调用元素hashCode方法和equals方法，判断元素是否重复。所以必须重写hashcode和equals。

# 一些java使用

**Optional处理空指针：**

 Optional.ofNullable(text)//显示创建一个Optional壳

   .map(Test::print)

  .map(Test::print)

  .ifPresent(System.out::println)

# tomcat

**三种部署方式：**

1. 静态部署：

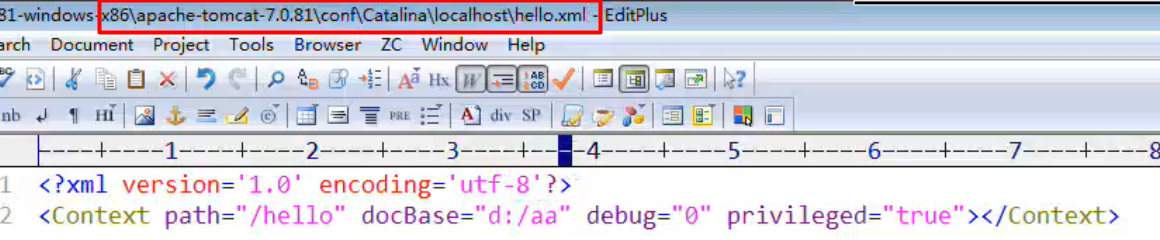
（1）、使用默认的host节点：直接放置项目在webapp目录。原理是默认的host节点名字为localhost，目录就是webapps目录

（2）、host节点中配置context节点:

注意path前面必须有“/”



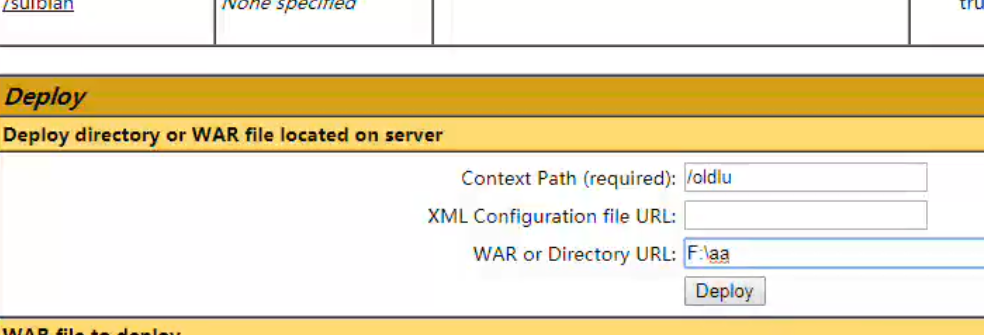
（3）、在localhost文件夹目录下放xml文件，文件名要跟path名称一样。放一个项目就创建一个xml文件。



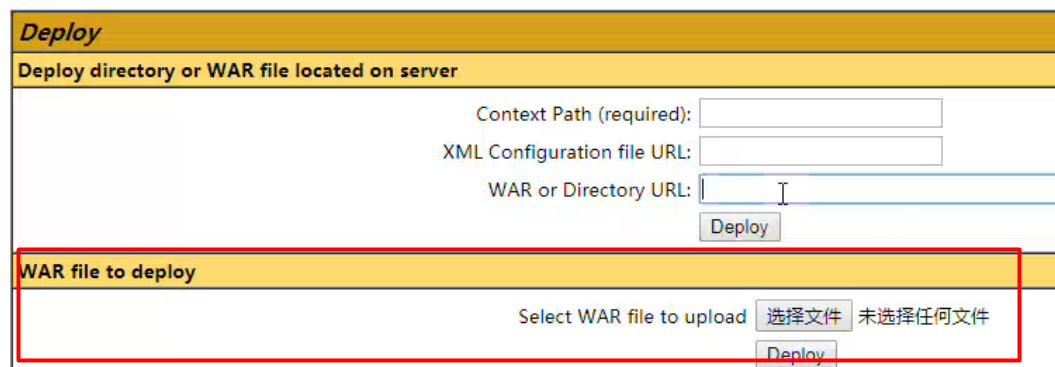
1. tomcat热部署（条件是没修改掉默认tomcat页面）：

访问localhost:8080，点击按钮进入tomcat的webapps管理界面，按提示配置好用户和角色。

（1）、项目部署



（2）、war包部署



**配置虚拟主机：**

server.xml文件下创建多个host节点，节点指向文件夹，文件夹中再放项目，即可配置不同域名映射不同项目。

如：



# jvm

**jvm调优**

目的：

减少垃圾收集的时间和次数。

**jvm调优实例：**

1、常规gc：

前提：当前linux账号能看该进程的gc信息。

jstat -gcutil 5671 1000

查看gc发生频率、时长，有异常则设置jvm参数。

2、下载dump文件并本地分析

注意：虚拟机参数-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=${目录}。可以在内存溢出时自动下载dump信息。

（1）、查看gc时间是否正常

jstat -gcutil 5671 1000 每1秒打印一次进程id为5671的gc信息。

（2）jmap生成堆dump文件并下载：

生成

jmap -F -dump:format=b,file=heapDump 5671

用xftp下载到本地

（3）用jdk6之后自带的jvisualVM工具打开并分析

--用OQL一层层查询其引用，看到是文件上传相关，应该是上传文件到云存储时没及时关闭。

（4）到文件上传的地方看代码，处理。

**linux观测内存占用方法：**

1、通过jstat（java统计）命令

（1）、jps 获取要优化的程序进程id 如234

（2）jstat -gc 234 1000 每隔一秒钟打印进程234的gc信息，以具体的值显示。（jstat -gcutil 234 1000则是以百分比显示）

2、java图形化观测工具

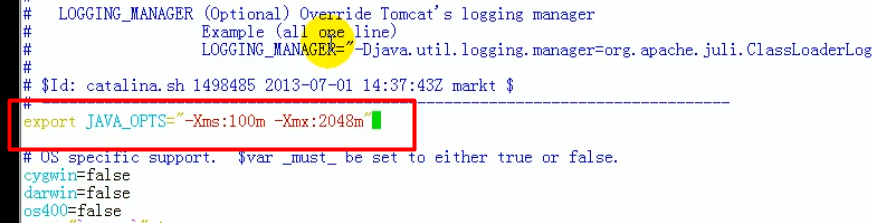
（1）、visualVM

**项目中配置jvm各个内存参数的方法：**

（1）cd usr/local/tomcat/bin

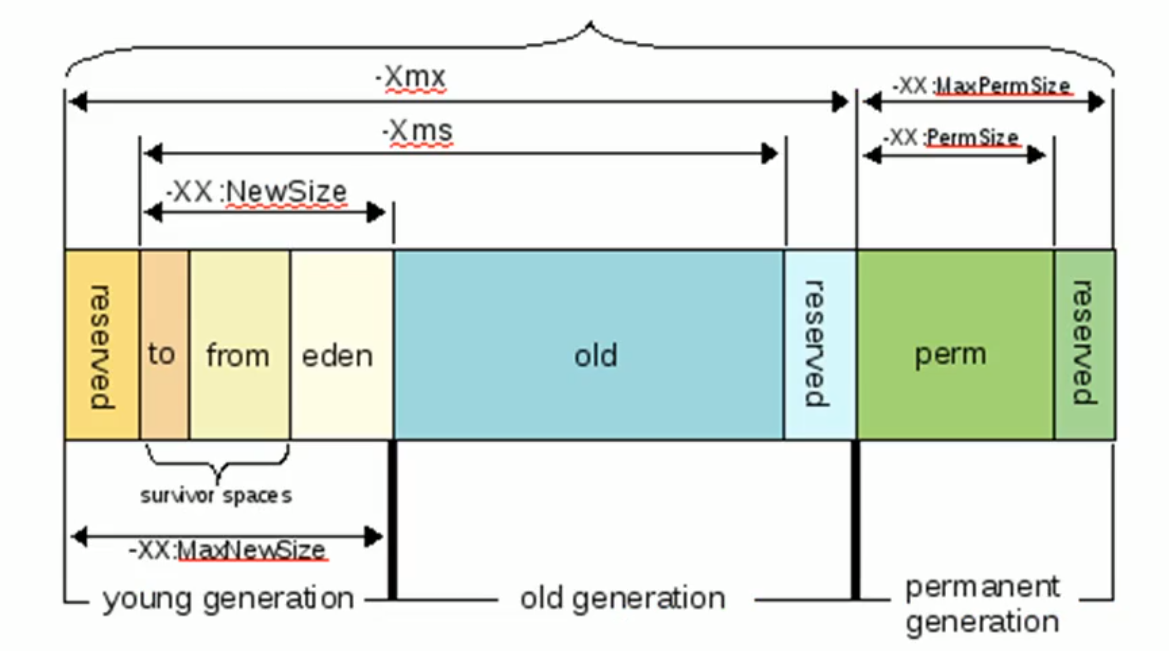
（2）vim catalina.sh

（3）设置配置参数（个内存大小、占比、垃圾收集算法等）



jdk5及以上且是双核的CPU，也许可以尝试这个参数: -XX:+UseParallelGC

**java进程内存模型**



个人总结：就三个 Xms，NewSize，PermSize。然后Xms – NewSize = 老年代内存大小。保留区是每个都有的块

**栈内存：**

如：。

public static void main( String args[]){

int a

select();

}

执行：main( 5 )

1、栈内存给执行main（）方法的线程分配一个栈空间。

2、main方法中变量a放于该栈的一个节点（栈先进后出）。

3、调用到selec方法，给select方法创建一个节点，压入栈中

栈内存满了后出现stackOverFlowError；

**堆内存：**

大概分三块： 年轻代（service0 + service1+NewSize）、老年代

gc清一次垃圾，eden的就进入srvice0，service0进入service1，service1进入old，当old满了之后，出现outOfMenmoryError。

**gc垃圾搜索算法：**

1. 引用计数器法:

有一个地方引用该对象则加计数器1，引用失效后减1，为0时可回收。缺点是无法解决循环引用问题，已废弃。

1. 根搜索算法：

创建对象引用链，某对象没有被引用链链接时，表示可回收。

**gc垃圾回收算法：**

1、标记—清除法：标记要回收的，要清除时进行清除。（效率不高，产生的空间不连续不适合分配到大内存）

2、复制算法：内存分两块，把存活的放到另一块中，清除剩下这块（内存利用率不高）。

3、标记—整理算法：把存活的往一端移动，清除存活边界以外的。（适合老年代）

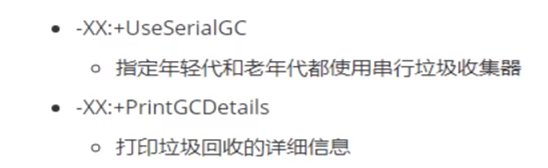
4、分代收集算法：新生代用复制算法，老年代用标记—整理算法。

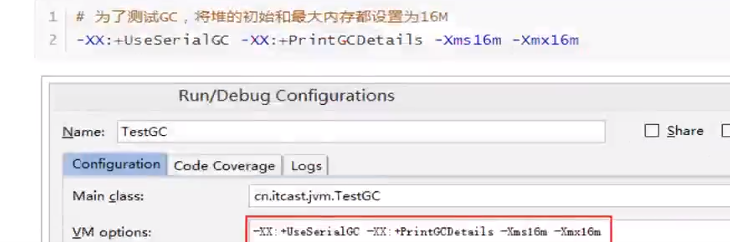
**各个垃圾收集器：**垃圾收集算法的落地

1、串行垃圾收集器（Serial）

单线程回收垃圾，回收时java应用线程都暂停。一般不用。

使用方法：



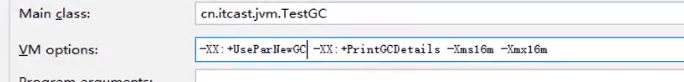


2、并行垃圾收集器

（1）、ParNew（多核CPU更好）

工作在年轻代上。多线程收集垃圾，并行能力强的服务器收集时间更快，只是期间也会暂停所有应用线程。

使用：



（2）、Parallel（多核CPU更好）

与ParNew差不多，只是加了和系统吞吐量相关的参数，更灵活高效。

# SpringCloud

服务中心（服务注册和使用）——Eureka（类zookeeper）

客户端负载均衡——Ribbon（类nginx）

断路器——Hystrix

服务网关——Zuul

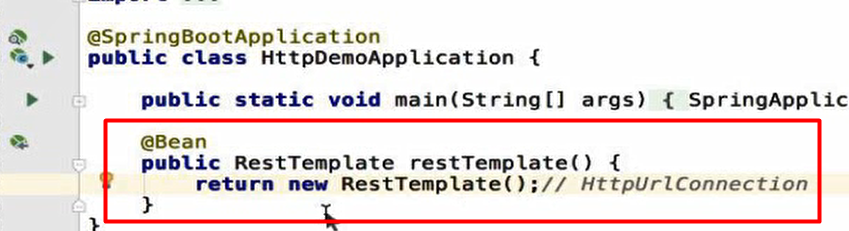
服务调用——feign（内部集成了ribbon）

**RestTemplate：**

一个调用http的工具类，默认json格式化。底层是对httpclient进行了封装。

使用：

1、注册为bean：

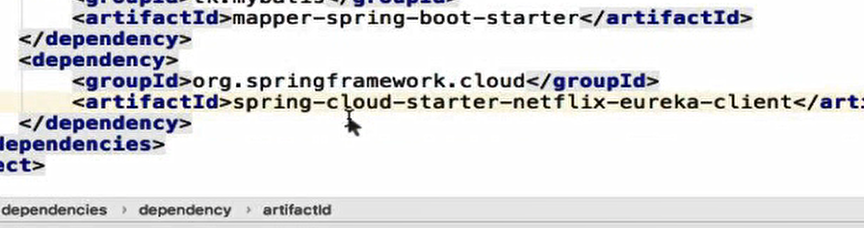


2、使用：

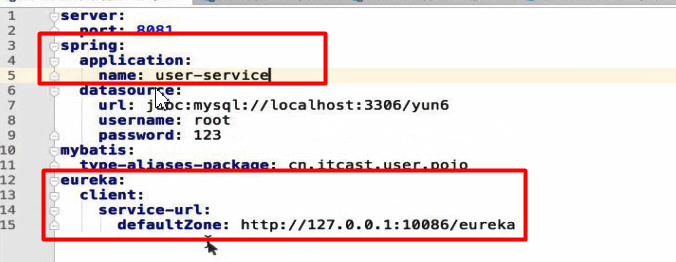


**客户端和服务端配置和应用流程：**

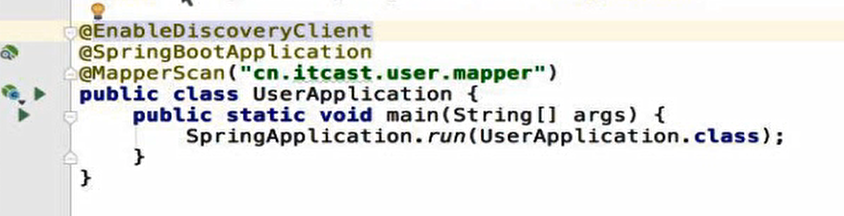
1. 添加依赖



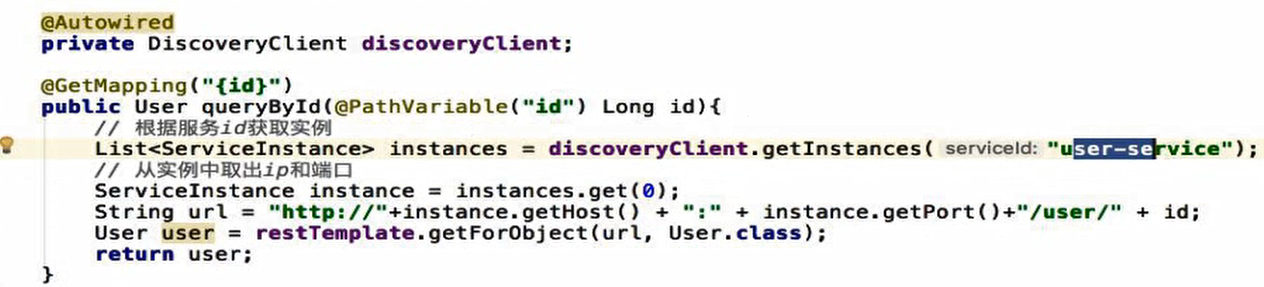
1. 配置服务中心地址和自身用户名



1. 启动类加注解@EnableDiscoveryClient（该注解支持Eureka、zk等三个客户端）



4、服务调用

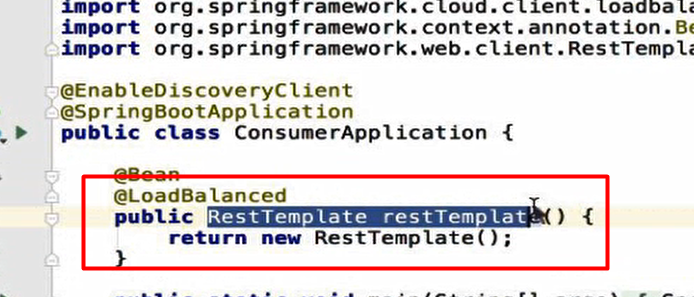


**负载均衡ribbon调用（整合euraka和ribbon后，消费者不用关系生产者端口和ip）：**

1. 要负载的客户端中引入配置

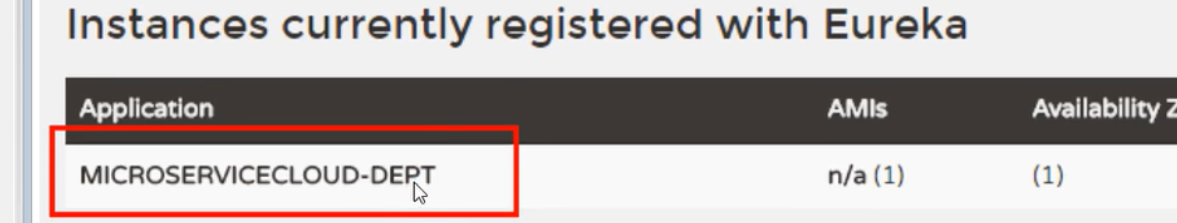


1. 添加@LoadBalanced注解（会在使用template时拦截，进行负载）

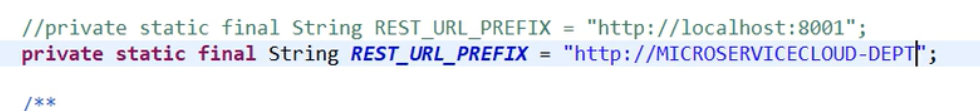


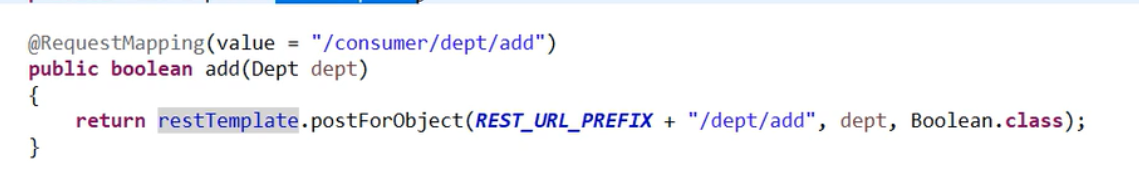
1. 使用:

euraka上查询提供者服务名



消费者使用该服务名





**Ribbon**客户端负载均衡**：**

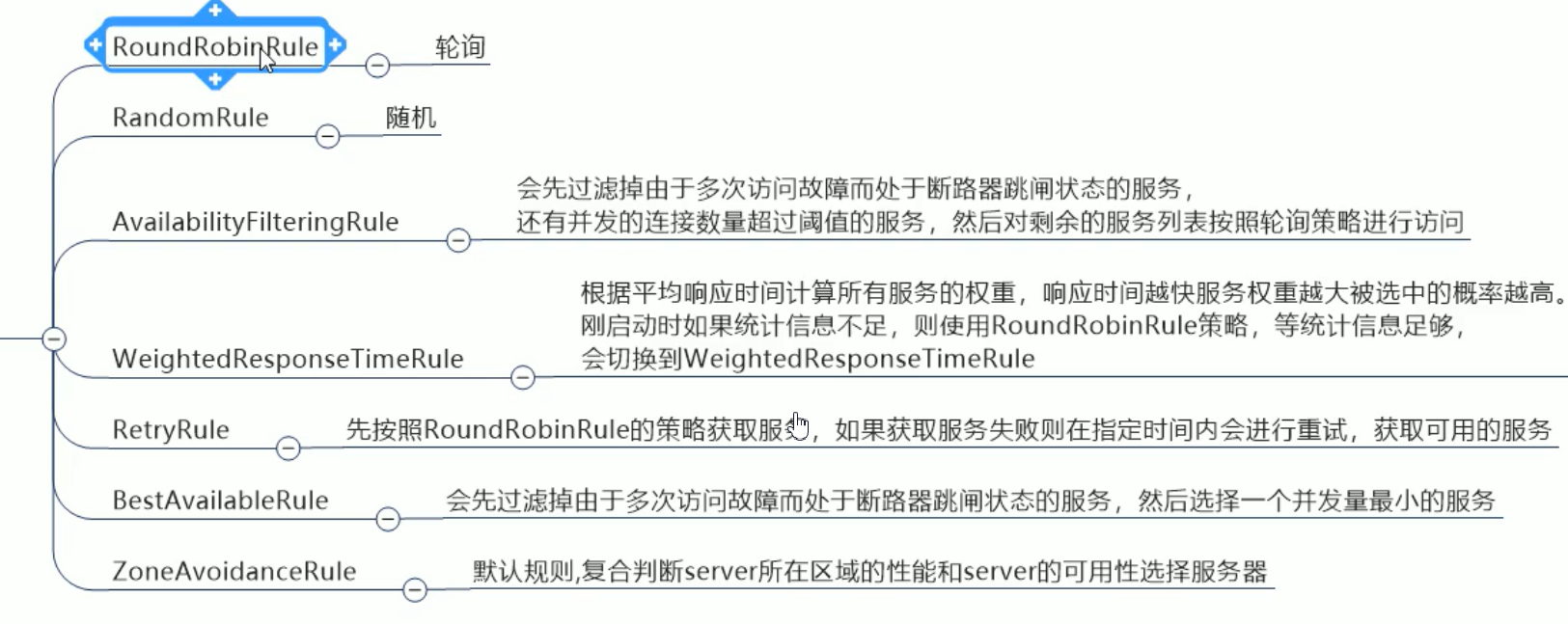
1、Load Balance负载均衡，简称LB

例如去买汉堡的人（客户），自动排到人少的窗口（服务）

集中式LB：客户端和服务端之间的负载工具，如nginx等

2、ribbon负载均衡算法

轮询（默认），其他如下：



**springcloud与dubbo比较：**

1、轮询（默认）



2、springcloud用http的rest方式，dubbo用rpc

3、springcloud更新力度强。

**eureka与zookeeper比较：**

1、zookeeper保证cp（一致性，分区容错性）

master节点故障后，选举时间太长（30-90s），导致服务长期不可用。

2、euraka保证ap（可用性，分区容错性）

节点都是平等的，某节点挂掉，用其他节点即可（可用性）。但是不保证节点都是新的（不保证一致性）

若15分钟内超85%的节点无正常心跳，则认为网络故障，此时：

（1）、Euraka不再从注册列表移除服务

（2）、继续接受新服务注册，但不同步到其他节点

（3）、网络稳定后进行同步节点

# springcloud下的权限方案?

我觉得其实也可以使用shiro框架，只要设置好分布式session即可。

# zookeeper

**zookeeper作为注册中心（如dubbo的注册中心）**

**zookeeper实现分布式事务**

服务a注册一个node节点，设置值为no，服务b执行完成后，修改node节点值为yes，此时zk会通知a，值已经改变为yes。若b不成功，a可以重试这套流程。

**zookeeper实现分布式锁**

a获取zk锁，b也获取zk锁时，zk可以保证只有一个服务能获取到锁。而且，在a释放锁后，zk自动把锁给b，省去了自己写代码监听锁的操作。

# Dubbo

**负载均衡策略**

1、随机（默认。给每个提供者权重）

2、轮询

3、最少活跃调用数，相同活跃数则随机。

4、一致性hash，相同参数总是发到同一提供者。

**dubbo的不同通信协议**

（1）dubbo协议（默认）

单一长连接、nio异步通信、hession协议（用于序列化）。

注：

长连接：建立后长期保持，期间可以发送多个请求。

短连接：发送请求后销毁。

nio异步通信：异步非阻塞通信，读写分别进行。

# 如何保证接口幂等性（重复发送、网络延迟等情况的处理）

1、每个请求必须有唯一的id。

2、处理完该请求，要标识其已经处理了。

3、每次接收请求，判断其是否已经处理。

如：协同的@Token防止重复提交。

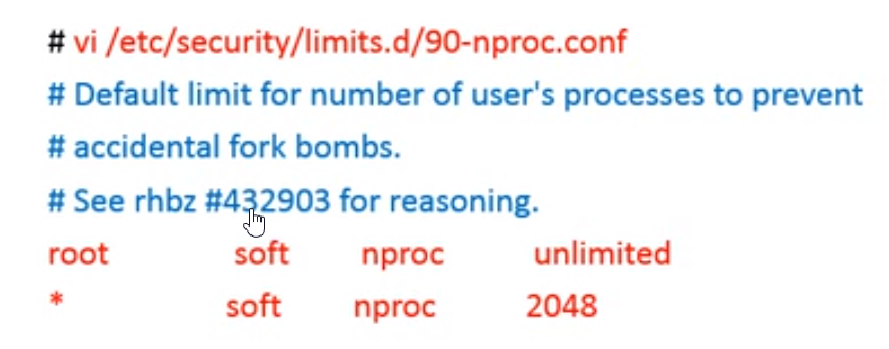
其他方法：可以用redis保存标识或者支付信息等，防止重复扣款。

# Linux

**配置用户最大线程数：**

如下图：root无限制，其他用户默认最大创建1024个。配置后重启系统才生效。

下图不代表全部版本的linux



注意：

1、尽量不要用root用户部署应用程序，root无线程个数限制，会耗尽资源无法登陆系统。

2、普通用户线程数，要看物理内存容量去配置，用ulimit -u命令查看操作系统允许普通用户设置的最大线程数。

# 设计模式

**分类：**



**设计模式的六大原则**

1、开闭原则

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

2、里氏代换原则

里氏代换原则面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3、依赖倒转原则

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。

5、迪米特法则（最少知道原则）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则

原则是尽量使用合成/聚合的方式，而不是使用继承。

**uml基础知识：**

<https://www.cnblogs.com/h2zZhou/p/6256725.html>

**设计模式理解参考：**

<https://blog.csdn.net/dengpeng0419/article/details/48110841>

**工厂模式：**

简单工厂模式、工厂模式、抽象工厂模式：

<https://blog.csdn.net/u012218309/article/details/81333631>

简单工厂模式可以根据key生成对象，方便统一控制（如切换云存储库）。

**策略模式：**

以鸟类举例，鸟的动作有唱、飞。策略模式让不同的鸟实现不同的唱、飞的动作。

--其实也可以重新定义一个新的构造方法，之前的不变。

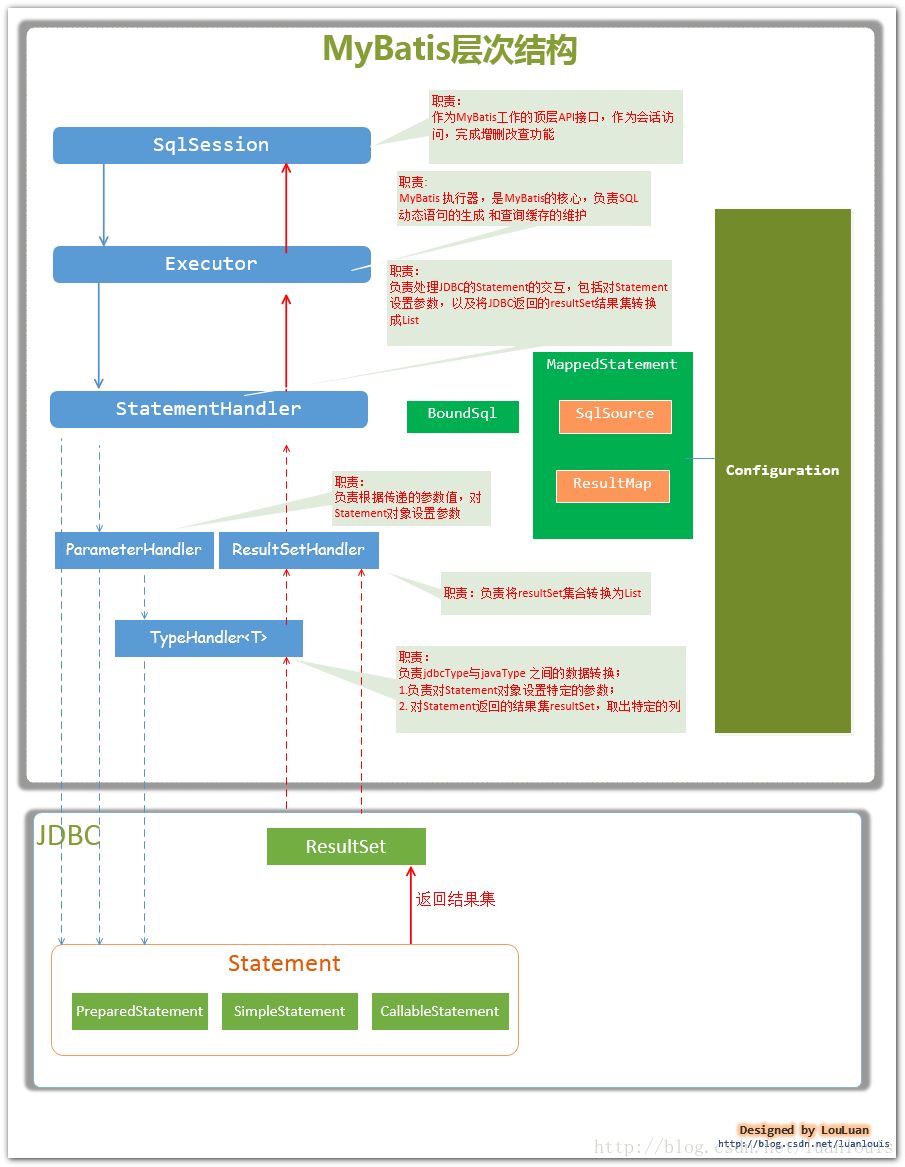


# Mybatis：

优势：方便的写sql和方便的转换sql结果成java对象。

理解：mybatis的 xxMapper.xml文件是xxDao层的实现类。

**流程图**



# NIO（new io）

**基本知识：**

1、除了boolean外，其他7个基本数据类型都有缓冲区，以Buffer结尾。

ByteBuffer，IntBuffer，ShortBuffer，LongBuffer，FloatBuffer，DoubleBuffer，CharBuffer

2、通过allocate（）方法获取缓冲区

3、核心方法：

put（）存入缓冲区，get（）从缓冲区获取

4、buffer的4个核心属性

capacity：总容量。声明后不能修改。

limit：界限后的数据不能进行读写。

position：当前操作位置

mark：记录position位置，用buf.reset（）方法让position回到该标记处。

**使用：**

下图表示put和get时各个属性位置

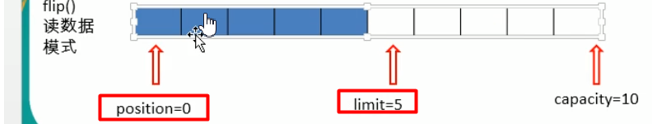
1、写模式：

（1）、buf.put（“abcde”）



2、读模式：

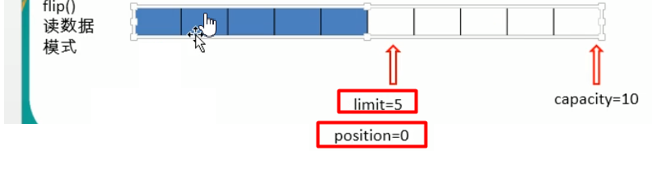
（1）、buf.flip（）



（2）、get（）获取数据

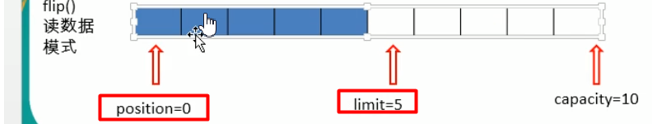
Byte temp = new Byte[ buf.limit() ]

buf.get( temp )



3、重复读

buf.rewind（）



4、buf.clear（）清空缓冲区，里面数据仍在，处于被遗忘状态（指针不指向数据了）。



**nio特点：**

面向缓冲区（有缓冲区，如buffer）

非阻塞io

选择器

**io特点：**

面向流（无缓冲直接传）

阻塞io

**直接缓冲区与非直接缓冲区：**

1、缓冲区申请方法

（1）、非直接缓冲区：

通过allocate（）分配缓冲区，建立在jvm内存中。

（2）、直接缓冲区：

getAllocateDirect（），建立在物理内存中（数据无须经过jvm与操作系统间的传输，大家都在物理内存读写）。

内存消耗大；文件合适写到磁盘，由操作系统控制；jvm对缓冲区的引用在垃圾回收器回收后才释放，占资源。

# spring源码

**一、spring的Aop源码：**

**aop全称**是Aspect-Oriented Programming，即面向切面编程。

**aop实现原理：**AOP的拦截功能是由java中的动态代理来实现的。说白了，就是在被代理类的基础上增加切面逻辑，生成增强的目标类（目标类的方法经过了增强，里面的方法除了被代理类的代码外，还嵌入了日志记录的代码，所以能实现日志记录）。

**那么spring如何生成这个增强的目标类呢（要么jdk动态代理要么cglib）？**

通过视频教程中的调试，spring的aop是在一个叫BeanPostProcessor接口完成的。该接口有许多实现类。spring在进行aop时，一个个调用其实现类，返回结果，完成动态代理，返回增强的目标类，实现动态代理。

切点：@PointCut

通知：@Before，@Around..

切面：切点和通知所在的类

**二、spring的ioc源码**

bean生命周期重要方法：

位于AbstractApplicationContext的refresh（）方法中。

**重要的方法：**

1、获取我们自定义bean的定义（AbstractBeanDefinition）的方法：

invokeBeanFactoryPostProcessors（beanFactory）

AbstractBeanDefinition定义：描述bean的属性，如是否抽象、单例等。

如何修改bean定义：实现BeanFactoryPostProcessor接口的postProcessorBeanFactory方法，可以设置bean定义的属性。

2、实例化各个bean的方法。

# spring各个技术概念

**二、动态代理：**

对方法的增强（只负责调用其方法，使用不同代理类，其方法也不同，增强了功能）。在运行的时候创建增强之后的对象（相当于生成一个拥有更多功能的对象）。

**jdk代理**

使用java.lang.reflect.Proxy的newProxyInstance方法实现动态代理。

与目标对象实现同一个接口，故能在接口的实现方法中，进行额外处理，实现aop。只能代理实现了接口的类，基于反射机制，生成代理接口的匿名类。

**cglib代理**

全称 code generator library，代码生成库。动态生成字节码对象（object.getClass()对象）

基于继承，继承被代理类，动态生成代理类的子类，故拥有相同方法，能在各个方法中进行额外操作。

# IOC、AOP：

，IOC（控制反转）是一种思想，DI（依赖注入）只是一种实现。IOC的实现原理是利用了JAVA的反射技术

AOP的实现原理是什么呢？——动态代理技术

目前动态代理技术主要分为Java自己提供的JDK动态代理技术和CGLIB技术。

Java自带的JDK动态代理技术是需要代理类实现接口（jdk动态代理会生成一个实现该接口的实例，里面调用被代理的实例的方法执行逻辑，并加上自己的增强逻辑）。

CGLIB是生成被代理类的子类。用字节码技术生成子类，子类中拦截父类调用，加入增强的逻辑。

# 一致性hash算法？

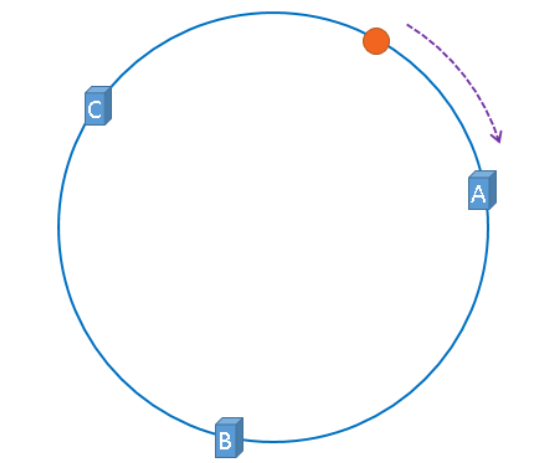
**解决了什么问题？**

解决取模算法变动机器个数时，一部分数据丢失的问题。

一致性hash算法是数据库分库分表、redis集群分配机器时用的，是解决取模算法固定机器缺陷的方案。

**方法：**

一致性hash算法，机器ip对2^32取模，把1到2^32-1看成环。数据对2^32取模，若数据的取模结果无机器与之匹配，则顺时针找到最近的机器，作为存储节点。



**优缺点**

优点：机器数量变化影响的数据变小。

缺点：数据倾斜问题。可能都映射到一台机器，故使用虚拟节点，机器ip+编号，一般32个ip+编号，来增加一个机器占用的节点，使数据尽可能平均分配）。

但是，变动机器时影响的数据如何处理？

--根据算法将需要变动的数据，移动到新节点？

# mysql

**常用函数？**

uuid()：获取一个uuid

avg、sum、max、min

concat( str1,str2… )

now()

**分页？**

用limit 偏移量,返回记录数

limit 5,10 偏移5，返回10条，即返回第 6到第15条的记录

## 数据库主从复制

**解决的问题？**

1、读写分离。主库写入时锁表，可读从库。

2、主库宕机，从库保证可用（集群思想）。

3、提高io性能。

**流程**（以binlog作为两库中介，保证数据一致）：

主库db的更新事件(update、insert、delete)被写到binlog

主库创建一个binlog dump thread，把binlog的内容发送到从库

从库启动并发起连接，连接到主库

从库启动之后，创建一个I/O线程，读取主库传过来的binlog内容并写入到relay log

从库启动之后，创建一个SQL线程，从relay log里面读取内容，从Exec\_Master\_Log\_Pos位置开始执行读取到的更新事件，将更新内容写入到自己的库

对于每一个主从复制的连接，都有三个线程。拥有多个从库的主库为每一个连接到主库的从库创建一个binlog输出线程，每一个从库都有它自己的I/O线程和SQL线程。

--从库的sql线程和io线程互相独立，在读写时，其io线程不受影响，保证性能。

--还有半复制等操作可以自己设置的，根据情况来。

**如何保证一致性？**

（1）半同步复制

等其中一个从库接收binlog并成功写入readylog后，才返回success给客户端。

优点：保证一致性

缺点：看主从网络带宽

（2）设置同步时间

可以用redis等中间件或数据库中间件。做转发，写转主库，读从库。写开始1s内来个读请求，则定向到主库读。避免从库未同步导致不一致。

**如何配置？（一般项目总监配置）**

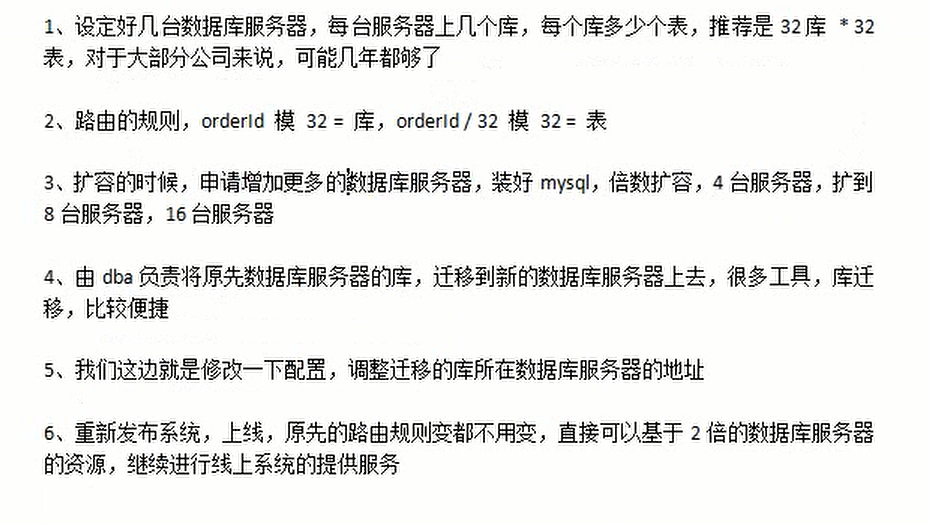
https://blog.csdn.net/change\_any\_time/article/details/79679509

**使用主从复制后，我们写的sql或者代码需要变化么？还是mysql自动切换主从库？**

有的系统是java代码select时自动切从库，总之这个不太清除。

## 分库分表动态扩容缩容方案：

注：总体思想，一开始就建32个数据库，后面可以分到32台服务器中，每台一个库即32\*1。或者8\*4。或者4\*8，开发时就1\*32。



# mq消息队列

**优点**：

解耦、异步，响应快（总时间=a+a发送mq时间）、削峰

**缺点**：

可用性降低，mq挂掉系统就挂了。

复杂度提高。引入消息重复发送、消息丢失问题。

分布式事务问题。本来abcd全成功才成功，实际上只有ab执行了，cd失败但是返回成功。

**各个队列优缺点**：

ActiveMQ RabbitMQ RocketMQ Kafka

**rabbitmq基本知识？**

分发方式：fanout（群发）、direct（某queue）、topic（模糊）

<https://www.jianshu.com/p/542e371c7f77>

**如何实现流量削峰**

问题情境：a（产生订单），b（库存、余额 …等数据库操作）。当a高峰期每秒1万，b数据库最多每秒处理2000条，会导致雪崩。

方法：变mq-server推模式为mq-client拉模式，每秒拉取自己能处理的最大值。同时优化b的代码和加多线程等等。

实现：

推模式：推模式（channel.basicConsume(QUEUE\_NAME, false, consumer）

拉模式：（channel.basicGet(QUEUE\_NAME, false);）

**如何保证高可用**：

1、rabbitmq高可用：

镜像集群模式：每个queue都会被自动同步到每个rabbitmq节点。任何节点宕机，消费者都可以到其他节点消费。

缺点：数据量特别大时，磁盘可能会挂。

如何开启：上网查

2、kafka高可用：

分布式+集群思想：

（1）创建topic 的分布式，即创建topic时，指定partition，假设是3个。

（2）每个partition自己作为leader，创建自己的副本follow，设置同步机制。

（3）当leader挂了，选举机制选举新的leader。

**如何保证幂等性（不重复消费？实际项目不需要都做幂等性）**

问题场景：当服务停机，没返回处理完成的信息，则mq认为未消费，服务重启后，mq再次发送同一个消息给消费者，导致重复消费。

数据库或者redis保存处理过的消息id，若存在则不处理。（但是放内存时高峰期数据量太大不好。其实放数据库也不需要查全部，查最新消费的即可）

**如何保证消息不丢失**

问题场景：（1）生产者发送的消息在网络传输上丢失了。（2）mq挂了，消费者没来得及消费。（3）开启了autoAck，消费者得到消息会自动发ack（ack确认的意思）给mq，但是没处理，消费者就挂了，mq认为已消费。

1. rabbitmq不丢失：

（1）rabbitmq发送保证不丢失

方法一：rabbitmq事务，不考虑，影响吞吐量。

方法二：confirm机制：把channel设置成confirm模式。生产者配置两个接口，代表成功（ack），失败（nack，not ack）。rabbitmq接收成功会回调成功接口，反正调失败接口。

（2）rabbitmq保证自身挂掉不丢失消息

方法：设置持久化

步骤a、在创建queue时设置成持久化的，保证持久化queue的源数据（但不持久化queue里的数据）。

步骤b、发送消息时将deliveryMode设置为2，持久化queue里的数据到磁盘。

a+b必须要同时用，配合confirm机制，在持久化到磁盘后才通知生产者ack。

（3）保证消费者不弄丢消息

将autoAck关闭，自己在成功后发送ack给mq。若mq没收到ack，则会自动发给其他消费者处理

**如何保证消息顺序性**

（1）rabbitmq集群中

情景：一个需要顺序消费的abc分别被三个消费者消费了，导致顺序不一致。

方法：给需要保证顺序的消息，开一个queue出来，只给一个消费者接收即可。

（2）kafka中

情景：kafka是分布式高可用架构，消费者从partition中，用多线程消费消息，会导致顺序不一致。（不开多线程每秒能处理的消息不多）

方法：给每个消费者线程弄个内存队列，把顺序执行的消息放同一个内存队列里面即可。

**积压几千万条queue怎么办**

若原来是3个消费者消费，则高峰期开30个临时消费者消费（数据库个数要增加？增加得考虑分库分表的取模问题了呀，不增加如何提升瓶颈？）。可以提前做好这个东西。

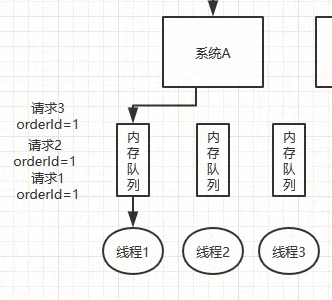
**消息数据过期怎么办**

一般不设置过期时间。（我觉得像订单这种，代码会判断的，queue就不设置过期了否则丢失数据）

# 分布式系统

**分布式系统如何保证请求的顺序**

用内存队列



**分布式系统的session方案**

1. 方案一tomcat+redis

简单，tomcat支持配置session到redis中，配置好即可。

缺点：严重依赖于web容器。若web容器迁移成jetty，要把jetty重新配置一份。

（2） 方案二spring session + redis（推荐）

该方案不依赖web容器、支持系统修改为springcloud微服务等。

**分布式系统的分布式锁各种方案**

使用zookeeper实现分布式锁

**分布式系统的分布式事务各种方案（面试就说不是我搞的，其他同事搞的，但是我知道流程）**

注意：

引入的话会增加10倍系统复杂性。

99%的bug是体验性的，而非数据出错。

99%的分布式接口不需要分布式事务，只需要监控（出错发邮件、短信等）、记录完整日志、事后定位排查修复

1. 方案一阿里的rocketMq+zookeeper

（2）方案二LCN框架解决分布式事务问题

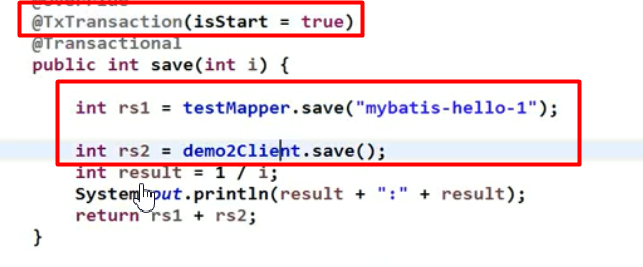
项目落地：

引入项目请参考<https://www.cnblogs.com/sxdcgaq8080/p/9776695.html>

1、pom引入项目TxManager

2、修改配置，改zk或eureka配置和redis配置

3、事务发起方加注解@TxTransaction（isStart = true）



1. 事务参与方—实现ITxTrasaction接口



原理（假设事务发起方为支付服务，接收方为积分服务）

1、支付服务创建事务分组（如分组内容是：1支付，2积分）到协调者（eureka或zookeeper）

2、支付调用积分服务时传分组id，积分收到分组id，执行代码后不会进行提交操作

3、支付执行成功，告诉协调者自己执行成功

4、协调者告诉积分服务，可以进行提交了

应该在分组的任务服务失败后，协调者会告诉其他服务不要提交，或者有机制让出问题的服务再试一下