支付宝并发方案之“一锁二判三更新”

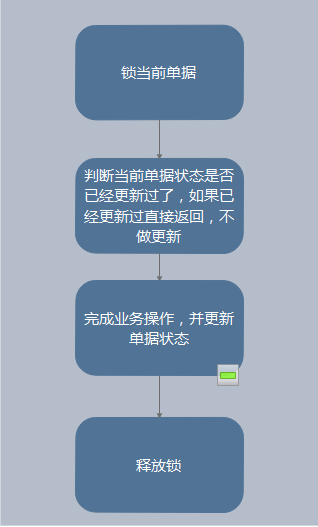
每年支付宝在双11和双12的活动中，都展示了绝佳的技术能力。这个能力不但体现在处理高TPS量的访问，更体现在几乎不会出错，不会出现重复支付的情况，这个是怎么做到的呢？

诚然，为了实现在高并发下仍不会出错的技术目标，支付宝下了很多功夫，比如幂等性的处理，分布式事务的使用等等，但是个人觉得其中最关键的一点就是“一锁二判三更新”这句看似毫不起眼的口诀。

何为“一锁二判三更新”？ 简单来说就是当任何一个并发请求过来的时候

1. 我们先锁定关联单据   
2. 然后判断关联单据状态，是否之前已经更新过对应状态了   
3. 如果基于第2步判断，之前并没有请求更新过对应状态，则本次请求可以更新并完成相关业务逻辑。   
如果之前已经有更新过状态了，则本次不能更新，也不能完成业务逻辑。

示意图：



|  |
| --- |
| //第1步锁当前支付单  PaymentInfo resultPaymentInfo = commonPayCoreService.queryPaymentForUpdate(createPaymentInfo.getId());  **if** (resultPaymentInfo.isFinalStatus()) {  //第2步，判断当前支付单状态，如果是终态，则直接返回  //不做任何更新  **return** resultPaymentInfo;  }  //第3步更新当前支付单状态到终态，并完成相关业务逻辑(支付成功)  payCoreService.updateRequestResult(payChannelResult); |

1.Java内存模型(堆、栈)

栈区：

栈分为java虚拟机栈和本地方法栈

①重点是Java虚拟机栈，它是线程私有的，生命周期与线程相同。

②每个方法执行都会创建一个栈帧，用于存放局部变量表，操作数栈，动态链接，方法出口等。每个方法从被调用，直到被执行完，对应着一个栈帧在虚拟机中从入栈到出栈的过程。

③ 通常说的栈就是指局部变量表部分，存放编译期间可知的8种基本数据类型，及对象引用和指令地址。局部变量表是在编译期间完成分配，当进入一个方法时，这个栈中的局部变量分配内存大小是确定的。

④ 会有两种异常StackOverFlowError和OutOfMemoryError。当线程请求栈深度大于虚拟机所允许的深度就会抛出StackOverFlowError错误；虚拟机栈动态扩展，当扩展无法申请到足够的内存空间时，抛出OutOfMemoryError。

⑤ 本地方法栈与虚拟机栈的区别是虚拟机栈为虚拟机执行Java方法(也就是字节码)服务，

而本地方法栈则为虚拟机使用到的Native方法服务。

堆区：

① 堆被所有线程共享区域，在虚拟机启动时创建，唯一目的存放对象实例。

② 堆区是GC的主要区域，通常情况下分为两个区块年轻代和老年代。更细一点年轻代又分为Eden区，放新创建对象，From survivor和To survivor保存gc后幸存下的对象，默认情况下各自占比8:1:1。

③ 会有异常OutOfMemoryError

方法区

① 被所有线程共享区域，用于存放已被虚拟机加载的类信息，常量，静态变量等数据，被Java虚拟机描述为堆的一个逻辑部分，习惯也叫它永久代(Permanent Generation)

② 垃圾回收很少光顾这个区域，不过也是需要回收的，主要针对常量池回收，类卸载。

③ 常量池具有一定的动态性，里面可以存放编译期生成的常量；运行期间的常量也可以添加进入常量池中，比如String的intern()方法。

[String的Intern方法详解](https://www.cnblogs.com/wxgblogs/p/5635099.html)

　　在 JAVA 语言中有8中基本类型和一种比较特殊的类型String。这些类型为了使他们在运行过程中速度更快，更节省内存，都提供了一种常量池的概念。常量池就类似一个JAVA系统级别提供的缓存。8种基本类型的常量池都是系统协调的，String类型的常量池比较特殊。它的主要使用方法有两种：

* 直接使用双引号声明出来的String对象会直接存储在常量池中。
* 如果不是用双引号声明的String对象，可以使用String提供的intern方法。intern 方法会从字符串常量池中查询当前字符串是否存在，若不存在就会将当前字符串放入常量池中

程序计数器

① 当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成的。

② Java虚拟机多线程是通过线程轮流切换并分配处理器执行时间的方式实现的。为了线程切换能恢复到正确的位置，每条线程都需要一个独立的程序计数器，所以它是线程私有的。

③ 唯一一块Java虚拟机没有规定任何OutOfMemoryError的区域

2.JVM类加载过程

加载->链接(验证->准备->解析)->初始化->使用->卸载

加载：将class文件字节码内容加载到内存中，并将这些静态数据转换成方法区中的运行时数据结构，在堆中生成一个代表这个类的java.lang.Class对象，作为方法区类数据的访问入口，这个过程需要类加载器参与。

链接：将Java类的二进制代码合并到JVM的运行状态之中的过程

验证：确保加载的类信息符合JVM规范，没有安全方面的问题

准备：正式为类变量(static变量)分配内存并且设置该类变量的初始值即0(如static int i = 5;这里只将i初始化为0，至于5的值将在初始化时赋值)，这里不包含用final修饰的static，因为final在编译的时候就会分配了，注意这里不会为实例化变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

解析：虚拟机常量池的符号引用替换为字节引用过程，符号引用就是一组符号来描述目标，可以是任何字面量，而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，接口方法解析。

初始化

① 初始化阶段是执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译器自动收藏类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生

② 当初始化一个类的时候，若发现其父类还没有进行过初始化，则需要先触发其父类的初始化。

③ 虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步

④ 当范围一个Java类的静态域时，只有真正声明这个域的类才会被初始化。

3.Java序列化和反序列化

序列化：将对象转换为二进制流的过程

反序列化：将二进制流恢复为对象的过程

4.Java IO流的理解

IO流简单来说就是Input和Output流，IO流主要是用来处理设备之间的数据传输，Java对于数据的操作都是通过流实现，而Java用于操作流的对象都在IO包中。

Java语言定义了许多类专门负责各种方式的输入或输出，这些类都被放在java.jo包中。其中:

所有输入流类都是抽象类InputStream(字节输入流)或者抽象类Reader(字符输入流)的子类；

所有输出流都是抽象类OutputStream(字节输出流)或者Writer(字符输出流)的子类

首先需要明白的是：流是干什么的？为了永久性保存数据

根据数据流向的不同分为输入流和输出流

根据处理数据类型的不同分为字符流和字节流

然后需要明白的是输入模式和输出模式是谁流向谁：

InputStream(字节输入流)和Reader(字符输入流)通俗的理解都是读(read)的

outputStream(字节输出流)和Writer(字符输出流)通俗的理解都是写(Writer)的。

字节流和字符流的用法几乎完全一样，区别在于字节流和字符流所操作的数据单元不同-字节流操作的数据单元是8位的字节，而字符流操作的数据单元是16位的字符。

字符流

字符流简介：

字符流的对象融合了编码表，也就是系统默认的编码表。我们的系统一般都是GBK编码。

字符流只用来处理文本数据，字节流用来处理媒体数据。

数据最常见的表现方式是文件，字符流用于操作文件的子类一般是FileReader和FileWriter。

字符流的读写

注意事项：

① 写入文件后必须要用flush()刷新

② 用完流后记得要关闭流

③ 使用流对象要抛出IO异常

5.线程安全和线程同步锁的了解

线程安全

指多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。

线程不安全

指不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。

6.Spring框架的IoC，AOP及底层实现机制

回答思路：

① 先用通俗易懂的话解释下何为IoC和AOP

② 各自实现原理

③ 自己的项目中如何使用

(1) IOC

许多应用都是通过彼此间的相互合作来实现业务逻辑的，如果类A要调用类B，以前我们都是在类A中，通过自身new一个类B，然后在调用类B的方法，现在我们把new类B的事情交给Spring来做，在我们调用的时候，容器会为我们实例化。

(2) IoC容器的初始化过程

资源定位，即定义bean的xml -> 载入 ->IoC容器注册，注册beanDefinition

IoC容器的初始化过程，一般不包含bean的依赖注入的实现，在spring IoC设计中，bean的注册和依赖注入是两个过程，依赖注入一般发生在应用第一次索取bean的时候，但是也可以在xml中配置，在容器初始化的时候，这个bean就完成了初始化。

(3) 三种注入方式：构造器、接口、set注入，我们常用的是set注入

(4) bean是如何创建 - 工厂模式

(5) 数据是如何注入 - 反射

(6) AOP

面向切面编程，在我们的应用中，经常需要做一些事情，但是这些事情与核心业务无关，比如，要记录所有update \* 方法的执行时间，操作人等信息，记录到日志。

(7) AOP的实现原理 - 代理

第一种：jdk动态代理(优先)

其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理，其核心的两个类是InvocationHandler和Proxy。

第二种：cglib代理

第三方代理技术，cglib代理，可以对任何类生成代理，代理的原理是对目标对象进行继承代理。如果目标对象被final修饰，那么该类无法被cglib代理。

这两种代理的区别是jdk动态代理需要目标对象实现接口，而cglib的动态代理则不需要。

7.消息队列的常用几种介绍下及应用场景

7.1.简介

消息队列中间件是分布式系统中重要的组件，主要解决应用耦合、异步消息、流量削峰等问题，实现高性能、高可用、可伸缩和最终一致性架构。使用较多的消息队列有ActiveMQ、RabbitMQ、ZeroMQ、Kafka、MetaMQ、RocketMQ。

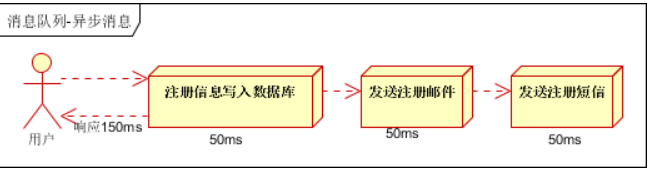
7.2.消息队列应用场景

以下介绍消息队列在实际应用中常用的使用场景：异步处理、应用解耦、流量削峰和消息通讯四个场景。

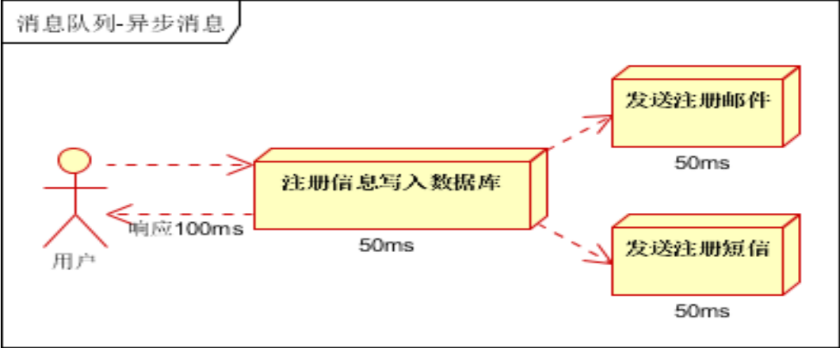
7.2.1.异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法两种：串行的方式和并行方式。

串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户。



并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端，与串行的差别是，并行的方式可以提高处理的时间。

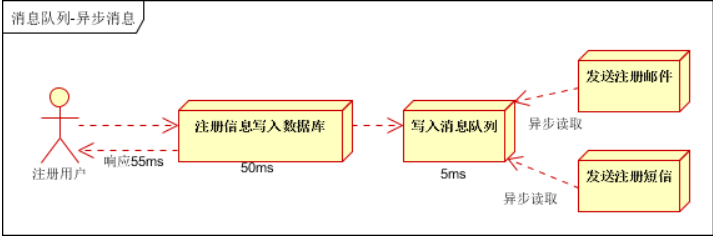


假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150ms，并行的时间可能是100ms。

因为CPU在单位时间内处理的请求是一定的，假设CPU1秒内吞吐量是100次。则串行方式1秒内CPU可处理的请求量是7次(1000/50)。并行方式处理的请求量是10次(1000/1000)。

小结：如以上案例描述，传统的方式系统的性能(并发量，吞吐量，响应时间)会有瓶颈。然后解决这个问题呢？

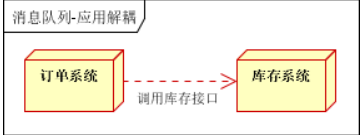
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：



按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，因此用户的响应时间可能是50ms。因此架构改变后，系统的吞吐量提高到每秒20QPS。比串行提高了3倍，比并行提高了2倍。

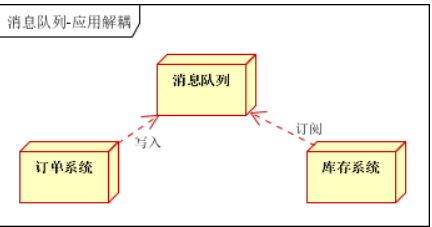
7.2.2.应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图：



**传统模式的特点：**

假如库存系统无法访问，则订单减存将失败，从而导致订单失败，订单系统与库存系统耦合。如何解决以上问题呢？引入应用消息队列后的方案，如下图：



订单系统：

用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功。

库存系统：

订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。

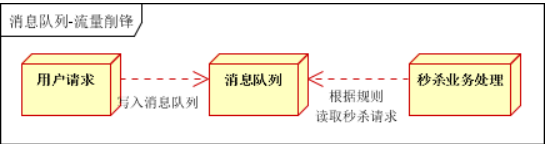
假如：在下单时库存系统不能正常使用。也不影响正常下单，因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

7.2.3.流量削峰

流量削峰也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛！

**应用场景**：秒杀活动，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。

可以控制活动的人数，可以缓解短时间内高流量压垮应用。



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面。

秒杀业务根据消息队列中的请求信息，再做后续处理。

7.2.4.日志处理

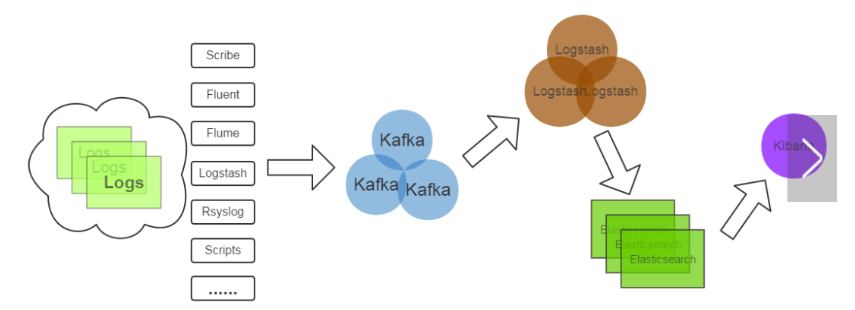
日志处理时指将消息队列用在日志处理中，比如Kafka的应用，解决大量日志传输的问题。

架构简化如下：



日志采集客户端，负责日志数据采集，定时写受写入Kafka队列；Kafka消息队列，负责日志数据的接收，存储和转发；日志处理应用;订阅并消费Kafka队列中的日志数据。

下面是新浪kafka日志处理应用案例：



Kafka:接收用户日志的消息队列；

Logstash：做日志解析，同一成JSON输出给Elasticsearch；

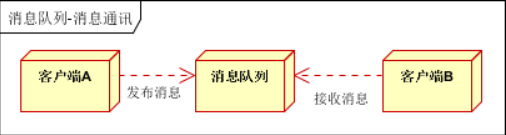
Elasticsearch：实时日志分析服务的核心技术，一个schemaless，实时的数据存储服务，通过index组织数据，兼具强大的搜索和统计功能；

kibana：基于Elasticsearch的数据可视化组件，超强的数据可视化能力是众多公司选择ElK stack的重要原因。

7.2.5.消息通讯

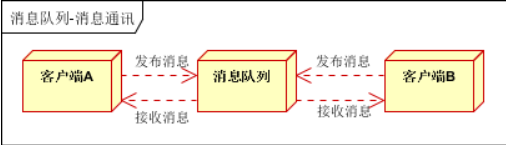
消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

**点对点通讯：**



客户端A和客户端B使用同一队列，进行消息通讯

**聊天室通讯**：

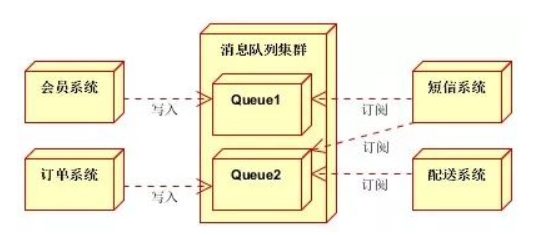


客户端A、客户端B，客户端N订阅同一主题，进行消息发布和接收，实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。模型为示意图，供参考。

7.3.消息中间件示例

① 电商系统



消息队列采用高可用，可持久化的消息中间件。比如ActiveMQ、RabbitMQ、RocketMQ

应用将主干逻辑处理完成后，写入消息队列。消息发送是否成功可以开启消息的确认模式。

(消息队列返回消息接收成功状态后，应用再返回，这样保障消息的完整性)

扩展流程(发短信，配送处理)订阅队列消息，采用推/拉的方式获取消息并处理；

消息将应用解耦的同时，带来数据一致性问题，可以采用最终一致性方式解决。比如主数据写入数据库，扩展应用根据消息队列，并结合数据库方式实现基于消息队列的后续处理。

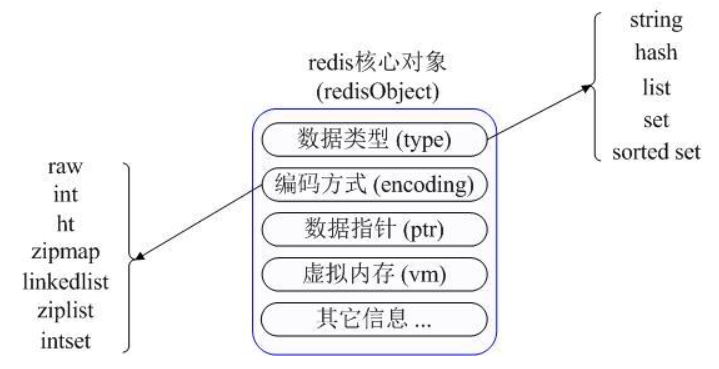
8.缓存的中间件介绍一二种，及使用场景

8.1.Redis常用数据类型

Redis最为常用的数据类型主要有以下五种：

String/Hash/List/Set/zset

下面我们先来逐一的分析下这五种数据类型的使用和内部实现方式



首先Redis内部使用一个redisObject对象来表示所有的key-value,redisObject最主要的信息如上图所示，type表示一个value对象具体是何种数据类型，encoding是不同数据类型在redis内部的存储方式，比如：type=String代表value存储的是一个普通字符串，那么对应的encoding可以是raw或者是int，如果是int则代表实际redis内部是按数值型存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如：“123”“456”这样的字符串。

这里需要特殊说明一下vm字段，只有打开了Redis的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的。通过上图我们可以发现Redis使用redisObject来表示所有的key/value数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给Redis不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用，我们随后会具体讨论。

8.2.各种数据类型应用和实现方式

8.2.1.String

String数据结构是简单的key-value类型，value其实不仅是数字，也可以是数字。

8.2.2.1.常用命令

set hello world 将间hello的值设置为world

get hello 获取存储在给定键中的值

del hello 删除这个键值对

incr key-name 将键存储的值加上1

decr key-name 将键存储的值减去1

8.2.2.2.应用场景

String是最常用的一种数据类型，普通的key/value存储都可以归为此类，即可以完全实现目前memcached的功能，并且效率更高。还可以享受Redis的定时持久化，操作日志即Replication等功能，除了提供与Memcached一样的get、set、incr、decr等操作外，Redis还提供了下面一些操作：

① 获取字符串长度

② 往字符串append内容

③ 设置和获取字符串的某一段内容

④ 设置及获取的某一位(bit)

⑤ 批量设置一系列字符串的内容

8.2.2.3.使用场景

常规key-value缓存应用，常规技术：微博数，粉丝数

8.2.2.4.实现方式

String在redis内部存储默认就是一个字符串，被redisObject所应用，当遇到incr,decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

8.2.2.Hash

8.2.2.1.常用命令

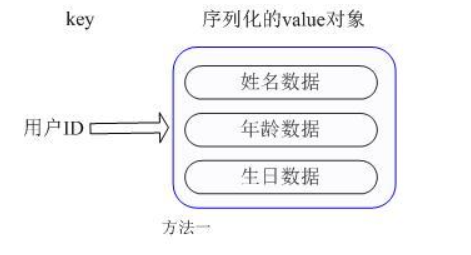
hget hset hgetall等

8.2.2.2.应用场景

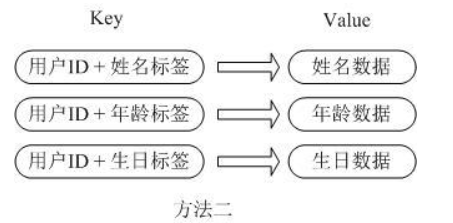
简单举个实例来描述下Hash的应用场景，比如要存储一个用户信息对象数据，包含以下信息：

用户ID为查找的key，存储的value用户对象包含姓名，年龄，生日等信息，

如果用普通的key/value结构来存储，主要有以下2种存储方式：

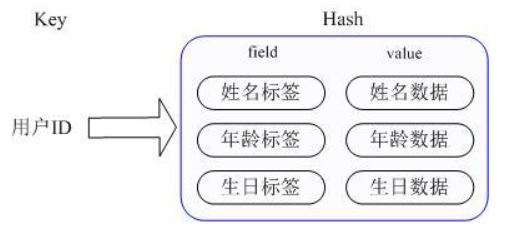


第一种方式将用户ID作为查找key，把其他信息封装成一个对象以序列化的方式存储，这种方式的缺点是，增加了序列化/反序列化的开销，并且在需要修改其中一项信息时，需要把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。



第二种方式是这个用户信息对象有多少成员就存成多少个key-value对，用用户ID+对应属性的名称作为唯一标识来取得对应属性的值，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

那么Redis提供的Hash很好的解决了这个问题，Redis的Hash实际是内部存储的value为一个HashMap，并提供了直接存取这个Map成员的接口，如下图：



也就是说，key仍然是用户ID，value是一个Map，这个Map的key是成员的属性名，value是属性值，这样对数据的修改和存取都可以直接通过其内部Map的Key(Redis里称内部Map的key为field)，也就是通过key(用户ID)+field(属性标签)就可以操作对应属性数据了，既不需要重复存储数据，也不会带来序列化和并发修改控制的问题，很好的解决了问题。

这里同时需要注意，Redis提供了接口(hgetall)可以直接取到全部的属性数据，但是如果内部Map的成员很多，那么涉及到遍历整个内部Map的操作，由于Redis单线程模型的缘故，这个遍历操作可能会比较耗时，而其他客户端的请求完全不响应，这点需要格外注意。

8.2.2.3.使用场景

存储部分变更数据，如用户信息等。

8.2.2.4.实现方式

上面已经说到Redis Hash对应value内部实际就是一个HashMap，实际这里会有2种不同实现，这个Hash的成员比较少时Redis为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的HashMap结构，对应的value redisObject的encoding为zipmap，当成员数量增大时会自动转成真正的HashMap，此时encoding为ht。

8.2.3.List

8.2.3.1.常用命令

lpush rpush lpop rpop lrange等

8.2.3.2.应用场景

Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现。

List就是链表，相信略有数据结构知识的人都应该能理解其结构，使用List结构，我们可以轻松地实现最新消息排行等功能。List的另一个应用就是消息队列，可以利用List的push操作，将任务存在List中，然后工作线程再用pop操作将任务取出进行执行。Redis还提供了操作List中某一段的api，可以直接查询、删除List中某一段的元素。

8.2.3.3.使用场景

消息队列系统

使用list可以构建队列系统，使用zset设置可以构建有优先级的队列系统。

比如：将Redis用作日志收集器

实际上还是一个队列，多个端点将日志信息写入Redis，然后一个worker同一将所有日志写到磁盘。

比如sina微博：

在Redis中我们的最新微博ID使用了常驻缓存，这是一直更新，但是我们做了限制不能超过5000个ID，因此我们的获取ID函数会一直询问Redis。只有在start/count参数超出了这个范围的时候，才会去访问数据库。

我们的系统不会像传统方式那样“刷新”缓存，Redis实例中的信息永远是一致的。SQL数据库(或是磁盘上的其他类型数据库)只是在用户需要获取“很远”的数据时才会被触发，而主页或第一个评论页是不会麻烦到硬盘上的数据库了。

8.2.3.4.实现方式

Redis list的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。Redis的list是每个子元素都是String类型的双向链表，可以通过push和pop操作从列表的头部或者尾部添加或者删除元素，这样List既可以作为栈，也可以作为队列。

8.2.4.set

8.2.4.1.常用命令

sadd spop smembers sunion

8.2.4.2.应用场景

Redis set对外提供的功能与list类似是一个列表的功能，特殊之处在于set是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在一个set集合内的重要接口，这个也是list所不能提供的。

set就是一个集合，集合的概念就是一堆不重复值的组合。利用Redis提供的set数据结构，可以存储一些集合性的数据。

案例：

在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。

set是集合，是string类型的无序集合，set是通过hashtable实现的，概念和数学中的集合基本类似，可以交集、并集、差集等，set中的元素是没有顺序的。

8.2.4.3.使用场景

获取某段时间所有数据去重值

这个是要Redis的set数据结构最合适了，只需要不断地将数据往set中扔就行了，set意为集合，所以会自动排重。

8.2.4.4.实现方式

set的内部实现是一个value永远为null的HashMap。实际就是通过计算hash的方式来快速排重的，这也是set能提供判断一个成员是否在集合内的原因。

8.2.5.zset

8.2.5.1.常用命令

zadd

增加一个或多个元素，如果该元素已经存在，更新它的score值，虽然有序集合有序，但它也是集合，不能重复元素，添加重复元素只会更新原有元素的score值

127.0.0.1:6379>zadd myZSet 1 zlh ---添加分数为1，值为zlh的zset集合

zcard

查看zset集合的成员个数。

127.0.0.1:6379>zcard myZSet ---输出zset的成员个数为3

8.2.5.2.使用场景

Redis zset的使用场景与set类似，区别是set不是自动有序的，而zset可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择set数据结构，比如Twitter的public timeline可以以发表时间作为score来存储，这样获取时就是自动按时间排好序的。

和set相比，zset增加了一个权重参数score，使得集合中的元素能够按score进行有序排序，比如一个存储全班同学成绩的zset，其集合value可以是同学的学号，而score就可以是其考试得分，这样在数据插入集合的时候，就已经进行了天然的排序。另外还可以用zset来做带权重的队列，比如普通消息的score为1，重要消息的score为2，然后工作线程可以选择按score的倒序来获取工作任务，让重要的任务优先执行。

8.2.5.3.实现方式

Redis zset的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，而跳跃表里存放的是所有的成员，排序依据是HashMap里存的score，使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

8.3.Redis实际应用场景

8.3.1.显示最新的项目列表

下面这个语句常用来显示最新项目，随着数据多了，

select \* from foo where ... order by time desc limit 10

在web应用中，“列出最新的回复”之类的查询非常普遍，这通常会带来可扩展性问题。这令人沮丧，因为项目本来就是按这个顺序被创建的，但要输出这个顺序却不得不进行排序操作。类似的问题就可以用Redis来解决。比如说，我们的一个Web应用想要列出用户贴出的最新20条评论。在最新的评论边上我们有一个“显示全部”的链接，点击后就可以获得更多的评论。

我们假设数据库中的每条评论都有一个唯一的递增的ID字段。我们可以使用分页来制作主页和评论页，使用Redis的模板，每次新评论发表时，会将它的ID添加到Redis列表：

LPUSH latest.comments<ID>

9.数据安全用到的加解密方法介绍一二及使用场景

安全算法主要包括摘要算法、对称加密算法、非对称加密算法、信息编码等。

总结：

摘要算法：MD5 SHA 不可逆，稳定

对称加密算法：DES算法、3DES算法、AES算法

非对称加密算法：RSA算法

数字签名算法：MD5withRSA SHA1whithRSA

9.1.摘要算法

单向散列函数的特点：

① 加密后密文的长度是定长的

② 如果明文不一样，那么散列后的结果一定不一样

③ 如果明文一样，那么加密后的密文一定一样(对相同数据加密，加密后的密文一样)

④ 所有的加密算法公开

⑤ 不可以逆推反算

9.1.1.MD5

1) 对字符串进行MD5加密可以得到128个比特位的摘要消息

2) 加密之后不能根据密文逆推出明文

9.1.2.SHA

SHA-1是基于MD4算法的，生成的摘要信息的长度为160位，由于生成的摘要信息更长，运算的过程更加复杂，在相同的硬件上，SHA-1的运行速度比MD5慢，但是也更为安全。

9.2.对称加密算法

1）加密/解密使用相同的密钥

2）加密和解密的过程是可逆的（明文-》密文-》明文）

9.3.非对称加密算法

1）使用公钥加密，使用私钥解密

2）公钥是公开的，私钥保密

3）加密处理安全，但是性能极差

9.4.数字签名

1.数字签名的应用场景

需要严格验证发送方身份信息情况

2.数字签名原理

1) 客户端处理

a.对“消息”进行hash得到“消息摘要”

b.发送方使用自己的私钥对“消息摘要”加密(数字签名)

c.把数字签名附着在“报文”的末尾一起发送给接收方

2) 服务端处理

a.对“消息”进行Hash得到“报文摘要”

b.使用公钥对“数字签名”解密

c.对结果进行匹配

10.数据库悲观锁和乐观锁的原理及应用

10.1.悲观锁

总是假设最坏的情况，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阻塞直到它拿到锁(共享资源每次只给一个线程使用，其他线程阻塞，用完后再把资源转让给其他线程)。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

10.2.乐观锁

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

10.3.适用场景

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

总结：

两种各有优缺点，读取频繁使用乐观锁，写入频繁使用悲观锁。

11.考核下数据库sql的能力，函数(group by，count(\*)之类)

where条件查询

group by分组与统计函数

max：求最大

min：求最小

sum：求和

avg：求平均

count:求总行数

select count(\*)：查询的是绝对的行数，就算某行全为null，也计算在内。

select count(列名)：查询的是该列不为null的所有行的行数

count(\*)和count(1)的区别：对于MyISAM引擎的表，是没有区别的，这种引擎内部有一计数器在维护着行数；如果是InnoDB的表，用count(\*)直接读行数，效率很低，因为会一行行数。

group by 有一个规则，就是select 后面的所有列中，没有使用聚合函数的列，必须出现在group by后面

如：我们对文字的分类id进行分组，查询每组分类的文章数

select articlecategory\_id,count(\*) from article group by articlecategory

查询每个类别下商品的平均价格

select cat\_id,avg(shop\_price) from goods group by cat\_id;

having子句可以让我们筛选成组后的各种数据，

where子句在聚合前先筛选记录，也就是说作用在group by和having子句前。

而having子句在聚合后对组记录进行筛选

order by(排序)是针对最终结果集，

所以order by 要放在where/group by/having后面。降序：desc 升序：asc

https://blog.csdn.net/fashion2014/article/details/78826299

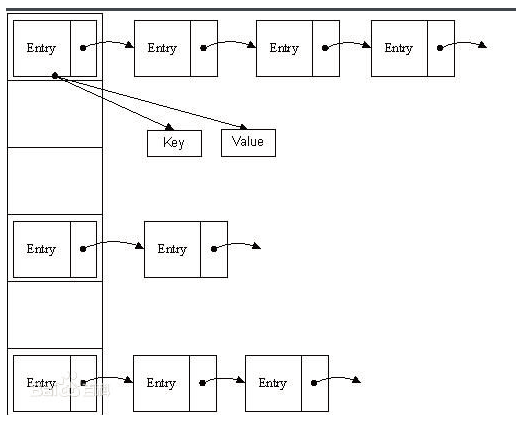
12.hashmap的实现原理

在JDK1.8之前，HashMap采用数组+链表实现，即使用链表处理冲突，同一hash值的节点都存储在一个表链表里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值一次查找的效率较低。

而JDK1.8中，HashMap采用数组+链表+红黑树实现，当链表长度超过阈值(8)时，将链表转换为红黑树，这样大大减少了查找时间。

下图中代表jdk1.8之前的HashMap结构，左边部分即代表哈希表，也成为哈希数组，

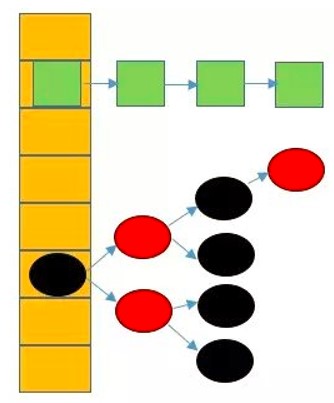
数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的key映射到了数组的同一位置处，就将其放入单链表中。

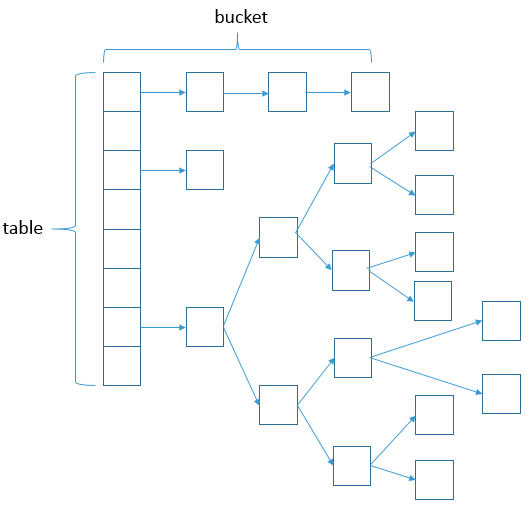


JDK1.8之前的HashMap都采用上图的结构，都是基于一个数组和多个单链表，hash值冲突的时候，就将对应节点以链表的形式存储。

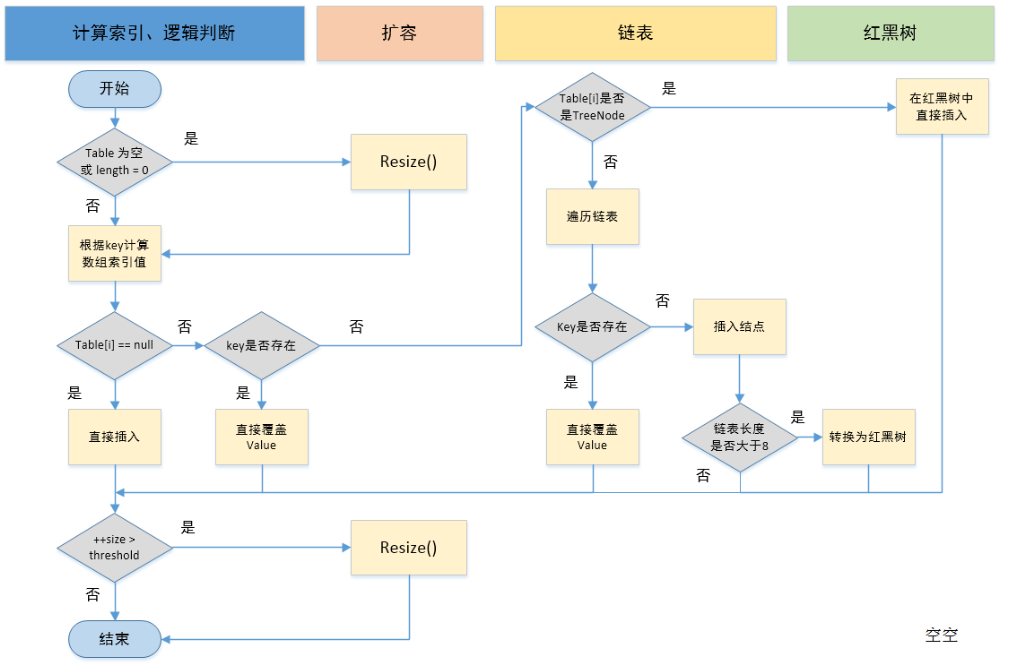
如果在一个链表中查找其中一个节点时，将会花费O(n)的查找时间，会有很大的性能损失。

到了JDK1.8,当同一个hash值的节点数不小于8时，不再采用单链表形式存储，而是采用红黑树，如下图所示。





说明：上图很形象的展示了HashMap的数据结构(数组+链表+红黑树)，桶中的结构可能是链表，也可能是红黑树，红黑树的引入是为了提高效率。



13.有几种线程池，他们的特点

线程池则是为了减少线程建立和销毁带来的性能消耗。

以web项目为例，有以下两种情况：

(1)每次过来一个请求，都要在服务端创建一个新线程来处理请求，请求处理完成销毁线程；

(2)每次过来一个请求，服务端在线程池中直接拿过一个空闲的线程来处理这个请求，处理

完成后还给线程池。

除了性能这个最重要的原因之外，线程池的使用可以帮助我们更合理的使用系统资源。

以web项目为例：

如果我们在服务端不使用线程池，而是无节制的来一个请求创建一个线程，系统资源将会很快被耗尽。而使用线程池的话，则可以防止这种情况发生，当然这要建立在正确合理的使用线程池的基础上，要固定线程的最大数以及等待队列的大小。

注：线程不允许使用Executors创建，而是通过ThreadPoolExecutor的方式创建，这样的处理方式能让编写代码的工程师更加明确线程池的运行规则，规避资源消耗的风险。

说明：Executors返回的线程池对象的弊端如下.

1) FixedThreadPool和SingleThreadPool：

允许的请求队列长度为Integer.MAX\_VALUE,可能会堆积大量的请求，从而导致OOM。

2) CachedThreadPool和ScheduledThreadPool：

允许的创建线程数量为Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM。

ThreadPoolExecutor有7个参数：

① corePoolSize

核心线程数，当有任务进来的时候，如果当前线程数还未达到corePoolSize个数，则创建核心线程，核心线程有几个特点：

1)当线程数未达到核心线程最大值的时候，新任务进来，即使有空闲线程，也不会复用，仍然新建核心线程。

2)核心线程一般不会被销毁，即使是空闲的状态，但是如果通过方法allowCoreThreadTimeOut(boolean value)设置为true时，超时也同样会被销毁；

3)生产环境首次初始化的时候，可以调用prestartCoreThread 方法来预先创建所有核心线程，避免第一次调用缓慢。

② maximumPoolSize

除了有核心线程外，有些策略是当核心线程完全无空闲的时候，还会创建一些临时的线程来处理任务，maximumPoolSize就是核心线程+临时线程的最大上限，临时线程有一个超时机制，超过了设置的空闲时间没事干，就会被销毁。

③ keepAliveTime

这个就是上面两个参数里所提到的超时时间，也就是线程的最大空闲时间，默认用于非核心线程，通过allowCoreThreadTimeOut(boolean value)方法设置后，也会用于核心线程。

④ unit

这个参数配合上面的keepAliveTime,指定超时的时间单位，秒、分、时等。

⑤ workQueue

等待执行的任务队列，如果核心线程没有空闲的了，新来的任务就会被放到这个等待队列中，这个参数其实一定程度上决定了线程池的运行策略，为什么这么说呢？因为队列分为有界队列和无界队列。

有界队列：队列的长度有上线，当核心线程满载的时候，新任务进来进入队列，当达到上限，又没有核心线程去及时取走处理，这个时候，就会创建临时线程(警惕临时线程无限增加的风险)

无界队列：队列没有上限的，当没有核心线程空闲的时候，新来的任务可以无止境的向队列中添加，而永远也不会创建临时线程。(警惕任务队列无限堆积的风险)

⑥ threadFactory

线程工厂，主要用来创建线程。

⑦ handler

当没有空闲的线程处理任务，并且等待队列已满(当然这只对有界队列有效)，再有新任务进来的话，就要做一些取舍了，而这个参数就是指定取舍策略的，有下面四种策略可以选择：

1) ThreadPoolExecutor.AbortPolicy:直接抛出异常，这是默认策略

2) ThreadPoolExecutor.DiscardPolicy:直接丢弃任务，但是不抛出异常

3) ThreadPoolExecutor.DiscardOldestPolicy:丢弃队列最前面的任务，然后将新来的任务加入等待队列。

4) ThreadPoolExecutor.CallerRunsPolicy:由线程池所在的线程处理该任务，比如在main函数中创建线程池，如果执行此策略，将有main线程来执行该任务。

13.1.newCachedThreadPool

① 底层

返回ThreadPoolExecutor实例，corePoolSize为0；

maximumPoolSize为Integer.MAX\_VALUE；

keepAliveTime为60L;

unit为TimeUnit.SECONDS；

workQueue为SynchronousQueue(同步队列)

② 通俗

当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可用线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

③ 适用

执行很多短期异步的小程序或者负载较轻的服务器

13.2.newFixedThreadPool

① 底层

返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，

corePoolSize为nThread，

maximumPoolSize为nThread；

keepAliveTime为0L(不限时)；

unit为：TimeUnit.MILLISECONDS；

WorkQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

② 通俗

创建可容纳固定数量线程的池子，每个线程的存活时间是无限的，当池子满了就不再添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

③ 适用

执行长期的任务，性能好很多

13.3.newSingleThreadExecutor

① 底层

FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；

unit为：TimeUnit.MILLISECONDS；

workQueue为：new LinkedBlockingQueue<Runnable>() 无界阻塞队列

② 通俗

创建只有一个线程的线程池，且线程的存活时间是无限的；

当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

③ 适用

一个任务一个任务执行的场景

13.4.NewScheduledThreadPool

① 底层

创建ScheduledThreadPoolExecutor实例，

corePoolSize为传递来的参数，

maximumPoolSize为Integer.MAX\_VALUE；

keepAliveTime为0；

unit为：TimeUnit.NANOSECONDS；

workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列

② 通俗

创建一个固定大小的线程池，线程池内线程存活时间无限制，

线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

③ 适用

周期性执行任务的场景

线程池任务执行流程：

当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。

当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行。

当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务。

当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理。

当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程

当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭

14.Spring的源码了解吗？简单介绍一下

15.SpringMVC的核心类是哪几个?

① 控制器核心类：

org.springframework.web.servlet.DispatcherServlet - 配置web.xml

② 加载配置文件核心类

org.springframework.web.context.ContextLoaderListener - spring的配置文件

③ 处理url映射核心类

org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping - 根据bean的名称请求一个bean spring的配置文件 /abc

④ 处理视图资源核心类：

org.springframework.web.servlet.view.ResourceBundleViewResolver

return hello

<context:component-scan base-package="cn.e3mall.service"/>用于替代上面的类

⑥ 方法动态调用核心类

org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver

16.在数据库中怎么实现设计乐观锁

17.数据库的隔离级别是什么

17.1.事务的基本要素(ACID)

① 原子性(Atomicity)

事务开始后所有操作，要么全部做完，要么全部不做，不可能停滞在中间环节。事务执行过程中出错，会回滚到事务开始前的状态，所有的操作就像没有发生一样，也就是事务是一个不可分割的整体。就像化学中学过的原子，是物质构成的基本单位。

② 一致性(Consistency)

事务开始前和结束后，数据库的完整性约束没有被破坏，比如A向B转账，不可能A扣了钱，B却没收到。

③ 隔离性(Isolation)

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如A正在从一张银行卡中取钱，在A取钱的过程结束前，B不能向这张卡转账。

④ 持久性(Durability)

事务完成后，事务对数据库的所有更新将保存到数据库，不能回滚。

17.2.事务的并发问题

① 脏读

事务A读取了事物B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据。

② 不可重复读

事务A多次读取同一数据，事务B在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果不一致。

③ 幻读

系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。

解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表。

17.3.MySQL事务隔离级别

|  |  |  |  |
| --- | --- | --- | --- |
| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| 读未提交(read uncommitted) | 是 | 是 | 是 |
| 提交读(read committed) | 否 | 是 | 是 |
| 可重复读(repeatable read) | 否 | 否 | 是 |
| 串行化(serializable) | 否 | 否 | 否 |

mysql默认的事务隔离级别是repeatable read

https://www.cnblogs.com/huanongying/p/7021555.html

18.数据库索引数据结构

MySQL索引使用的数据结构：B-Tree和B+Tree

MyISAM是MySQL5.5之前版本默认的存储引擎，从5.5之后，InnoDB开始成为MySQL默认的存储引擎。

MyISAM使用B-Tree实现主键索引、唯一索引和非主键索引。

InnoDB中非主键索引使用的是B-Tree数据结构，而主键索引使用的是B+Tree

B+树(适用于范围查找)

① 有K个字数的中间节点包含有k个元素(B树中是k-1个元素)，每个元素不保存数据，只用来做索引，所有数据都保存在叶子节点。

MyISAM

① 不支持事务，但是每次查询都是原子的；

② 支持表级锁，即每次操作是对整个表加锁；

③ 存储表的总行数；

Innodb

① 支持ACID的事务，支持事务的四种隔离级别；

② 支持行级锁及外键约束：因此可以支持写并发

③ 不存储总行数；

19.sql的优化方式

① 对查询进行优化，应尽量避免全表扫描，首先应考虑在where及order涉及的列上建立索引

② 应尽量避免在where子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

③ 应尽量避免在where子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num is null

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num=0

20.可重入锁和synchronized的区别

synchronized是Java内建的同步机制，所以也有人称其为intrinsic Locking，它提供了互斥的语义和可见性，当一个线程已经获取当前锁时，其他试图获取的线程只能等待或者阻塞在那里。

在Java5以前，synchronized是仅有的同步手段，在代码中，synchronized可以用来修饰方法，也可以使用在特定的代码块上，本质上synchronized方法等同于把方法全部语句用synchronized块包起来。

ReentrantLock,通常翻译为再入锁，是Java5提供的锁实现，它的语义和synchronized基本相同。再入锁通过代码直接调用lock()方法获取，代码书写也更加灵活。与此同时，ReentrantLock提供了很多实用的方法，能够实现很多synchronized无法做到的细节控制，比如可以控制fairness，也就是公平性，或者利用定义条件等。

|  |
| --- |
| ReentrantLock fairLock = new ReentrantLock(true); |

这里所谓的公平性是指在竞争场景中，当公平性为真时，会倾向于将锁赋予等待时间最久的线程。公平性是减少线程“饥饿”(个别线程长期等待锁，但始终无法获取)情况发生的一个办法。

如果使用synchronized，我们根本无法进行公平性的选择，其永远是不公平的，这也是主流操作系统线程调度的选择。通用场景中，公平性未必有想象中的那么重要，Java默认的调度策略很少会导致“饥饿”发生。与此同时，若要保证公平性则会引入额外开销，自然会导致一定的吞吐量下降。所以，只有当程序确实有公平性需要的时候，才有必要指定它。

但是，编码中也需要注意，必须要明确调用unlock()方法释放，不然就会一直持有该锁。

synchronized和ReentrantLock的性能不能一概而论，早期版本synchronized在很多场景下性能相差较大，在后续版本进行了较多改进，在低竞争场景中表现可能优于ReentrantLock.

21.简单介绍一下读写锁

类ReentrantLock具有完全互斥排他的效果，即同一时间只有一个线程在执行ReentrantLock.lock()方法后面的任务。这样做虽然保证了实例变量的线程安全性，但效率却是非常低下的。所以JDK中提供了一种读写锁ReentrantReadWriteLock类，使用它可以加快运行效率，在某些不需要操作实例变量的方法中，完全可以使用读写锁ReentrantReadWriteLock来提升该方法的代码运行速度。

读写锁表示也有两个锁，一个是读操作相关的锁，也称为共享锁；另一个是写操作相关的锁，也叫排他锁。也就是多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥。在没有线程Thread进行写入操作时，进行读取操作的多个Thread都可以获取读锁，而进行写入操作的Thread只有在获取写锁后才能进行写入操作。即多个Thread可以同时进行读取操作，但是同一时刻只允许一个Thread进行写入操作。

21.1.读写锁的使用：读读共享

类Service.java代码如下：

|  |
| --- |
| **public** **class** Service {  **private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();  **public** **void** read() {  **try** {  **try** {  lock.readLock().lock();  System.***out***.println("获取读锁" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());  Thread.*sleep*(10000);  } **finally** {  lock.readLock().unlock();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

两个线程类的代码如下所示：

|  |
| --- |
| **public** **class** ThreadA **extends** Thread{  **private** Service service;    **public** ThreadA(Service service) {  **super**();  **this**.service = service;  }    @Override  **public** **void** run() {  service.read();  }  } |

|  |
| --- |
| **public** **class** ThreadB **extends** Thread{  **private** Service service;    **public** ThreadB(Service service) {  **super**();  **this**.service = service;  }    @Override  **public** **void** run() {  service.read();  }  } |

文件Run.java代码如下：

|  |
| --- |
| **public** **class** Run {  **public** **static** **void** main(String[] args) {  Service service = **new** Service();  ThreadA a = **new** ThreadA(service);  a.setName("A");  ThreadB b = **new** ThreadB(service);  b.setName("B");  a.start();  b.start();  }  } |

程序运行后的结果

|  |
| --- |
| 获取读锁A 1536577644646  获取读锁B 1536577644646 |

从控制台中打印的时间来看，两个线程几乎同时进入lock()方法后面的代码。说明在此使用了lock.readLock()读锁可以提高程序运行效率，运行多个线程同时执行lock()方法后面的代码

21.2.读写锁的使用：写写互斥

类Service.java代码如下：

|  |
| --- |
| **public** **class** Service1 {  **private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();  **public** **void** read() {  **try** {  **try** {  lock.writeLock().lock();  System.***out***.println("获取写锁" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());  Thread.*sleep*(10000);  } **finally** {  lock.writeLock().unlock();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

程序运行后的结果：

|  |
| --- |
| 获取写锁A 1536582631546  获取写锁B 1536582641546 |

使用写锁代码lock.writeLock()的效果就是同一时间只允许一个线程执行lock()方法后面的代码。

21.3.读写锁的使用：读写互斥

类Service.java代码

|  |
| --- |
| **public** **class** Service2 {  **private** ReentrantReadWriteLock lock = **new** ReentrantReadWriteLock();  **public** **void** read() {  **try** {  **try** {  lock.readLock().lock();  System.***out***.println("获取读锁" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());  Thread.*sleep*(10000);  } **finally** {  lock.readLock().unlock();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }    **public** **void** write() {  **try** {  **try** {  lock.writeLock().lock();  System.***out***.println("获取写锁" + Thread.*currentThread*().getName() + " " + System.*currentTimeMillis*());  Thread.*sleep*(10000);  } **finally** {  lock.writeLock().unlock();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

运行类Run.java代码更改如下：

|  |
| --- |
| **public** **class** Run {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Service2 service = **new** Service2();  ThreadA a = **new** ThreadA(service);  a.setName("A");  a.start();  Thread.*sleep*(1000);  ThreadB b = **new** ThreadB(service);  b.setName("B");  b.start();  }  } |

程序运行后的结果

|  |
| --- |
| 获取读锁A 1536584824814  获取写锁B 1536584834815 |

此实验说明“读写”操作是互斥的，而且下一个示例说明“写读”操作也是互斥的。

即只要出现“写操作”的过程，就是互斥的。

21.4.读写锁的使用：写读互斥

|  |
| --- |
| **public** **class** Run {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Service2 service = **new** Service2();  ThreadB b = **new** ThreadB(service);  b.setName("B");  b.start();  Thread.*sleep*(1000);  ThreadA a = **new** ThreadA(service);  a.setName("A");  a.start();  }  } |

从控制台中打印的结果来看，“写读”操作也是互斥的。

“读写”、“写读”和“写写”都是互斥的；而“读读”是异步的，非互斥的。

22.JVM的内存区域划分

23.有几种垃圾回收器，他们的特点

① 标记-清除算法

它是最基础的收集算法。

原理：分为标记和清除两个阶段：首先标记出所有的需要回收的对象，在标记完成以后统一回收 所有被标记的对象。

特点：(1)效率问题，标记和清除的效率都不高；(2)空间的问题，标记清除以后会产生大量不连续的空间碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

地方：适合老年代进行垃圾回收，比如CMS收集器就是采用该算法进行回收的。

② 标记-整理算法

原理：分为标记和整理两个阶段：首先标记出所有需要回收的对象，让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

特点：不会产生空间碎片，但是整理会花一定的时间。

地方：适合老年代进行垃圾收集，

Parallel Old（针对parallel scanvange gc的） gc和Serial old收集器就是采用该算法进行回收。

③ 复制算法

原理

它先将可用的内存按容量划分为大小相同的两块，每次只是用其中的一块。当这块内存用完了，就将还存活着的对象复制到另一块上面，然后把已经使用过的内存空间一次清理掉。

特点：没有内存碎片，只要移动堆顶指针，按顺序分配内存。代价是将内存缩小为原来的一半。

地方：适合新生代区域进行垃圾回收。

Serial / ParNew / Parallel Scavenge收集器，就是采用该算法进行回收的。

复制算法改进思路：由于新生代都是朝生夕死的，所以不需要1:1划分内存空间，可以将内存划分为一块较大的Eden和两块较小的Survivor空间。每次使用Eden和其中一块Survivor。当回收的时候，将Eden和Survivor中还活着的对象一次性地复制到另一块Survivor空间上，最后清理掉Eden和刚才使用过的Survivor空间。其中Eden和Survivor的大小比例是8:1:1。

缺点是需要老年代进行分配担保，如果第二块的Survivor空间不够的时候，需要对老年代进行垃圾回收，然后存储新生代的对象，这些新生代当然会直接进入老年代。

23.1.Serial收集器(复制算法)

Serial收集器是最基本、发展历史最悠久的收集器。是单线程的收集器。它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集完成。Serial收集器依然是虚拟机运行在Client模式下默认新生代收集器，对于运行在Client模式下的虚拟机来说是一个很好的选择。优点是和其他收集器的单线程比，简单高效，对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

23.2.ParNew收集器(复制算法)

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The Worl、对象分配规则、回收策略等都与Serial 收集器完全一样。

ParNew收集器是许多运行在Server模式下的虚拟机中首选新生代收集器，其中有一个与性能无关但很重要的原因是，除Serial收集器之外，目前只有ParNew它能与CMS收集器配合工作。

23.3.Parallel Scavenge收集器(复制算法)

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)，虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

23.4.Serial Old收集器(标记-整理算法)

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在意给Client模式下的虚拟机使用。

如果在Server模式下，那么它主要还有两大用途：

① 在JDK1.5以及之前的版本中与Parallel Scavenge收集器搭配使用

② 作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

23.5.Parallel Old收集器(标记-整理算法)

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK1.6中才开始提供的，并行收集器，吞吐量优先。

23.6.CMS收集器(标记-清理算法)

CMS收集器是一种以获取最短回收停顿时间为目标的收集器。

CMS并非没有暂停，而是用两次短暂停来替代串行标记整理算法的长暂停，它的收集周期是这样：

初始标记 -> 并发标记 -> 重新标记 -> 并发清除

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”。

初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，

并发标记阶段就是进行GC Roots Tracing的过程，

而重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

缺点：

① 并发阶段占用CPU资源，拖慢用户程序，降低吞吐量，CMS默认启动的回收线程数是（CPU数量+3）/4，也就是当CPU在4个以上并发回收是垃圾收集线程不少于25%的CPU资源

② CMS收集器无法处理浮动垃圾。可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。由于CMS并发处理阶段用户线程还在运行着，伴随程序运行自然还会有新的垃圾产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分

③ 收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

23.7.G1收集器(标记-整理算法)

24.常用的几个并发容器

java collection framework提供了丰富的容器，有map、list、set、queue、deque。但是其存在一个不足：多数容器类都是非线程安全的，即使部分容器是线程安全的，由于适用synchronized进行锁控制，导致读/写均需进行锁操作，性能很低。

CAS

先比较目标对象现值是否和旧值一致，如果一致，则更新对象为新值；如果不一致，则表明对象已经被其他线程修改，直接返回。

24.1.ConcurrentHashMap

ConcurrentHashMap实现了HashTable的所有功能，线程安全，但却在检索元素时不需要锁定，因此效率更高。

ConcurrentHashMap的key 和 value都不允许null出现。原因在于ConcurrentHashMap不能区分出value是null还是没有map上，相对的HashMap却可以允许null值，在于其使用在单线程环境下，可以使用containKey（key）方法提前判定是否能map上，从而区分这两种情况

24.2.ConcurrentLinkedQueue

ConcurrentLinkedQueue使用链表作为数据结构，它采用无锁操作，可以认为是高并发环境下性能最好的队列。

ConcurrentLinkedQueue是非阻塞线程安全队列，无界，故不太适合做生产者消费者模式，而LinkedBlockingQueue是阻塞线程安全队列，可以做到有界，通常用于生产者消费者模式。

24.3.CopyOnWriteArrayList

CopyOnWriteArrayList提供高效地读取操作，使用在读多写少的场景。CopyOnWriteArrayList读取操作不用加锁，且是安全的；写操作时，先copy一份原有数据数组，再对复制数据进行写入操作，最后将复制数据替换原有数据，从而保证写操作不影响读操作。

24.4.ConcurrentSkipListMap

ConcurrentSkipListMap的实现就是实现了一个无锁版的跳表，主要是利用无锁的链表的实现来管理跳表底层，同样利用CAS来完成替换。

25.ConcurrentHashMap的实现原理是什么？

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多地方都会将其描述为分段锁。

ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

当执行put方法插入数据时，根据key的hash值，在Segment数组中找到相应的位置，如果相应位置的Segment还未初始化，则通过CAS进行赋值，接着执行Segment对象的put方法通过加锁机制插入数据。

1.8中放弃了Segment臃肿的设计，取而代之的是采用Node + CAS + Synchronized来保证并发安全进行实现。

put实现：

① 如果相应位置的Node还未初始化，则通过CAS插入相应的数据

② 如果相应位置的Node不为空，且当前该节点不处于移动状态，则对该节点加synchronized锁，如果该节点的hash不小于0，则遍历链表更新节点或插入新节点

26.红黑树

红黑树是一个平衡的二叉树，但不是一个完美的平衡二叉树。虽然我们希望一个所有查找都能在~lgN比较内结束，但是这样在动态插入中保持树的完美平衡代价太高，所以，我们稍微放松逛一下限制，希望找到一个能在对数时间内完成查找的数据结构-红黑树。

红黑树是在普通二叉树上，对每个节点添加一个颜色属性形成的，同时整个红黑二叉树需要同时满足一下五条性质。

红黑树需要满足的五条性质：

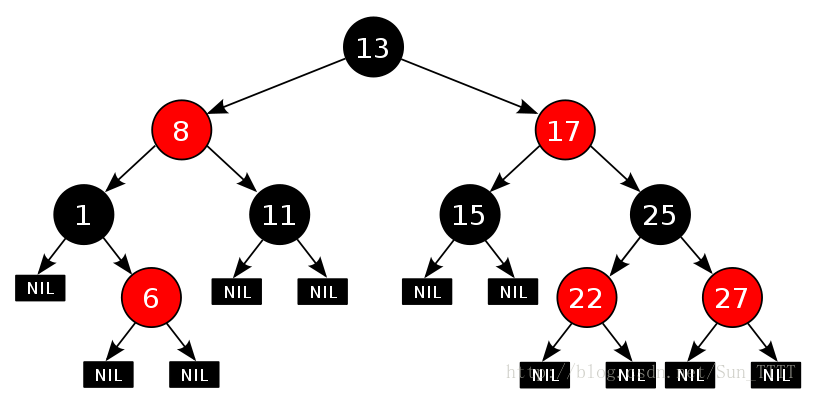
① 节点是红色或者是黑色

在树里面的节点不是红色就是黑色的，没有其他颜色

② 根节点是黑色

根节点总是黑色，它不能为红。

③ 每个叶节点(NIL或空节点)是黑色



这个图片就是一个红黑树，NIL节点是个空节点，并且是黑色的。

④ 每个红色节点的两个子节点都是黑色的(也就是说不存在两个连续的红色节点)

就是连续的两个节点不能是连续的红色，连续的两个节点的意思是父节点与子节点不能是连续的红色。

⑤ 从根节点到每一个NIL节点的路径中，都包含了相同数量的黑色节点。

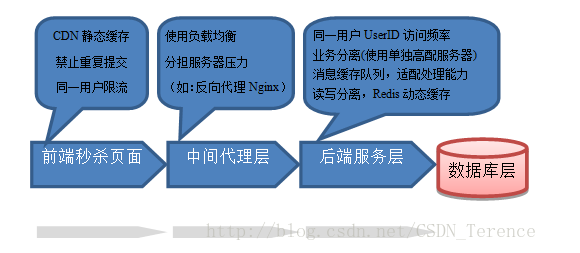
3.实现高并发秒杀方案

3.1.秒杀系统架构设计优化

一个常规的秒杀系统从前到后，依次有：

前端浏览器秒杀页面=》中间代理服务=》后端服务层=》数据库层

根据这个流程，一般优化设计思路：将请求拦截到上游，降低下游压力，在一个并发量大，实际需求小的系统中，应当尽量在前端拦截无效流量，降低下游服务器和数据库的压力，不然很可能造成数据库读写锁冲突，甚至导致死锁



前端方案

① 静态资源缓存：将活动页面上的所有可以静态的元素全部静态化，尽量减少动态元素；通过CDN缓存静态资源，来抗峰值。

② 禁止重复提交：用户提交之后按钮置灰，禁止重复提交

③ 用户限流：在某一时间段内只允许用户提交一次请求，比如可以采取IP限流

服务层

当用户量非常大的时候，拦截流量后的请求访问量还是非常大，此时仍需进一步优化。

1.业务分离:将秒杀业务系统和其他业务分离，单独放在高配服务器上，可以集中资源对访问请求抗压。

2.采用消息队列缓存请求：将大流量请求写到消息队列缓存，利用服务器根据自己的处理能力主动到消息缓存队列中抓取任务处理请求，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。

3.利用缓存应对读请求：对于读多写少业务，大部分请求是查询请求，所以可以读写分离，利用缓存分担数据库压力。

4.利用缓存应对写请求：缓存也是可以应对写请求的，可把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。

1.分布式事务处理

1.1.CAP定理

一个提供数据服务的存储系统无法同时满足

① 数据一致性(Consistency)

所有应用程序都能访问得到相同的数据

② 数据可用性(Availibility)

任何时候，任何应用程序都可以读写访问

③ 分区耐受性(Partition Tolerance)

 即使出现单个组件无法可用,操作依然可以完成

系统可以跨网络分区线性伸缩

在大型网站中，通常会选择强化分布式存储系统的可用性(A)和伸缩性(P)，而在某种程度上放弃一致性(C)。

数据最终一致：物理存储的数据可能是不一致，终端用户访问到的数据可能也是不一致的，但系统经过一个比较短的时间段的自我恢复和修正，数据最终会达到一致。

1.2.BASE定理

① Basically Available（基本可用）

分布式系统在出现故障时，允许损失部分可用功能，保证核心功能可用

② Soft state（软状态）

允许系统中存在中间状态，这个状态不影响系统可用性，这里指的是CAP中的不一致。

③ Eventually consistent（最终一致性）

最终一致是指经过一段时间后，所有节点数据都将会达到一致。

理论的核心思想是：**我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性**

1.3.两阶段提交(2PC)

二阶段提交协议主要分为来个阶段：准备阶段和提交阶段.

① 应用程序发起一个开始请求到TC(事务协调器)

② TC先将<prepare>消息写到本地日志，之后向所有的Si发起<prepare>消息。

为什么在执行任务前需要先写本地日志，只要是为了故障后恢复用。

③ Si收到<prepare>消息后，执行具体本机事务，但不会进行commit，

如果成功返回<yes>，不成功返回<no>。

同理，返回前都应把要返回的消息写到日志里，当作凭证。

④ TC收集所有执行器返回的消息，如果所有执行器都返回yes，那么给所有执行器发送commit消息，执行器收到commit后执行本地事务的commit操作；

如果有任何一个执行器返回no，那么给所有执行器发送abort消息，执行器收到abort消息后执行事务abort操作

1.3.1.两阶段提交例子

以A转账到B为例:

事务协调器先将<prepare>消息写到本地日志

TC给A的<prepare>消息是通知甲数据库相应账目扣款1w，

TC给给B的<prepare>消息是通知乙数据库相应账目增加1w.

A、B收到<prepare>消息后，执行具体本机事务，但不会进行commit，

如果成功返回<yes>，不成功返回<no>。返回前都应把要返回的消息写到日志里

事务协调器收集A和B返回的消息，如果A和B都返回yes，

那么事务协调器给A和B发送commit消息，

A和B收到commit后执行本地事务的commit操作；

如果有任何一个库返回no，那么事务协调器给两个库发送abort消息，

两个库收到abort消息后执行事务abort操作。

1.3.2.两阶段提交缺点

① 2PC协议中，如果出现协调者和参与者都挂了的情况，有可能导致数据不一致

② 两阶段提交涉及多次节点间的网络通信，通信时间太长

③ 事务时间相对于变长了，锁定的资源的时间也变长了，造成资源等待时间也增加好多

1.4.三阶段提交协议(3PC)

3PC最关键要解决的就是协调者和参与者同时挂掉的问题，所以3PC把2PC的准备阶段再次一分为二：这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。

在第一阶段，只是询问所有参与者是否可以执行事务操作，并不在本阶段执行事务操作。当协调者收到所有的参与者都返回yes时，在第二阶段才执行事务操作，然后在第三阶段在执行commit或者rollback.

1.4.1.3PC为什么比2PC好

如果挂掉的那台机器已经执行了commit，那么协调者可以从所有未挂掉的参与者的状态中分析出来，并执行commit，如果挂掉的那个参与者执行了rollback，那么协调器和其他的参与者执行的肯定也是rollback操作。

1.4.2.3PC存在的问题

在doCommit阶段，如果参与者无法及时接收到来自协调者的doCommit或者rebort请求时，会在等待超时之后，会继续进行事务的提交。

所以，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。

1.5.补偿事务TCC

TCC采用的是补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿(撤销)操作。它分为三个阶段：

① Try阶段主要是对业务系统做检测及资源预留

② Confirm阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行Confirm阶段时，默认Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。

③ Cancel阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

1.5.1.TCC例子

假如A要向B转账，思路大概是：

我们有一个本地方法，里面依次调用

① 首先在Try阶段，要先调用远程接口把A和B的钱给冻结起来

② 在Confirm阶段，执行远程调用的转账操作，转账成功进行解冻

③ 如果第2步执行成功，那么转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法(Cancel)

1.5.2.优缺点

优点：跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点：缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码。

1.6.使用消息队列来避免分布式事务

1.6.1.如何可靠保存凭证(消息)

有两种方法

1.6.1.1.业务与消息耦合的方式

A在完成扣款同时，同时记录消息数据，这个消息数据与业务数据保存在同一数据库实例里假设消息记录表表名为message

|  |
| --- |
| Begin transaction  update A set amount=amount-10000 where userId=1;  insert into message(userId, amount,status) values(1,10000,1);  End transaction  commit; |

上述事务能保证只要A账户里被扣了钱，消息一定能保存下来。

当上述事务提交成功后，我们通过实时消息服务MQ将此消息通知B，如果A发送失败，会进行重试发送，B处理成功后发送回复成功消息，A收到回复消息后删除该条消息数据。

1.6.1.2.业务与消息耦合方式

上述保存消息的方式使得消息数据和业务数据紧耦合在一起，从架构上看不够优雅，而且容易诱发其他问题。为了解耦，可以采用以下方式：

1）A在扣款事务提交之前，向实时消息服务请求发送消息，实时消息服务只记录消息数据，而不真正发送，只有消息发送成功后才会提交事务；

2)当A扣款事务被提交成功后，向实时消息服务确认发送，只有在得到确认发送指令后，实时消息服务才真正发送消息

3)当A扣款事务提交失败回滚后，向实时消息服务取消发送，在得到取消发送指令后，

该消息将不会被发送；

4）对于那些未确认的消息或者取消的消息，需要有一个消息状态确认系统定时去A系统查询这个消息的状态并进行更新。为什么需要这一步骤，举个例子：假设在第2步支付宝扣款事务被成功提交后，系统挂了，此时消息状态并未更新为“确认发送”，从而导致消息不能被发送。

优点：消息数据独立存储，降低业务系统与消息系统间的耦合。

缺点:一次消息发送需要两次请求，业务处理服务需要实现消息状态回查接口。

1.6.1.3.如何解决消息重复投递的问题

还有一个很重要的问题就是消息重复投递，以A转账到B为例，如果相同的消息被重复投递两次，那么我们B将会增加两万而不是一万了。

为什么相同的消息会被重复投递？比如B处理完消息msg后，发送处理成功的消息给A，正常情况下A应该要删除消息msg，但如果A这时候悲剧的挂了，重启后一看消息msg还在，就会继续发送消息msg。

解决方法很简单，在B这边增加消息应用状态(message\_apply)，通俗来说就是个账本。用于记录消息的消费情况，每次来一个消息，在真正执行之前，先去消息应用状态表中查询一遍，如果找到说明是重复消息，丢弃即可，如果没找到才执行，同时插入到消息应用状态表(同一事务)。

2.MySQL索引失效的情况(5种)

索引并不是时时都会生效，比如以下几种情况，将导致索引失效：

① 如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么少用or的原因)

注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引

② 对于多列索引，不是使用的第一部分(第一个)，则不会使用索引

③ like查询是以%开头

④ 如果列类型是字符串，那一定要在条件中将数据使用引号引用起来，否则不使用索引。

⑤ 如果mysql估计使用全表扫描要比使用索引快，则不使用索引

3.MySQL优化SQL查询的方法

① 对查询进行优化，应尽量避免全表扫描，

首先应考虑在where及order by 涉及的列上建立索引。

② 应尽量避免在 where 子句中使用 !=或<> 操作符，

否则将引擎放弃使用索引而进行全表扫描。

③ 应尽量避免在 where 子句中对字段进行 null值判断，

否则将导致引擎放弃使用索引而进行全表扫描，如：

　　select id from t where num is null

　　可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num=0

④ 应尽量避免在 where 子句中使用 or 来连接条件，

否则将导致引擎放弃使用索引而进行全表扫描。

⑤ 不要在 where 子句中的“=”【左边】进行函数、算术运算或其他表达式运算，

否则系统将可能无法正确使用索引。

⑥ 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

4.分布式锁的几种实现方式

一个方法在同一时间内只能被同一个线程执行

针对分布式锁的实现，目前比较常用的有以下几种方案：

4.1.基于数据库实现分布式锁

4.1.1.基于数据库表

要实现分布式锁，最简单的可能就是直接创建一张锁表，然后通过操作该表中的数据来实现。

当要锁住某个方法或资源时，就在该表中增加一条记录，想要释放锁的时候就删除这条记录。

创建这样一张数据库表：



当我们想要锁住某个方法时，执行以下SQL：

|  |
| --- |
| insert into methodLock(method,name,desc)values(‘method\_name’,‘desc’) |

因为对method\_name做了唯一约束，这里如果有多个请求同时提交到数据库的话，数据库会保证只有一个操作可以成功，那么我们就可以认为操作成功的那个线程获得了该方法的锁，可以执行方法体内容。

当方法执行完毕之后，想要释放锁的话，需要执行以下SQL：

|  |
| --- |
| delete from methodLock where method\_name=‘method\_name’ |

4.1.1.1.基于数据库表实现分布式的缺点

① 这把锁强依赖数据库的可用性，数据库是一个单点，一旦挂掉，会导致业务系统不可用。

② 这把锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得锁。

③ 这把锁只能是非阻塞的，因为数据的insert操作，一旦插入失败就会直接报错，没有获得锁的线程并不会进入排队队列，就想再次获得锁就要再次出发获得锁操作。

④ 这把锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁，因为数据中数据已经存在了。

4.1.1.2.解决方案

① 数据库是单点？搞两个数据库，数据之前双向同步。一旦挂掉快速切换到备库上。

② 没有失效时间？只要做一个定时任务，每隔一定时间把数据库中的超时数据清理一遍。

③ 非阻塞的？搞一个while循环，直到insert成功再返回成功。

④ 非重入的？在数据库表中加个字段，记录当前获得锁的机器的主机信息和线程信息，那么下次再获取锁的时候先查询数据库，如果当前机器的主机信息和线程信息在数据库可以查到的话，直接把锁分配给他就可以了。

4.2.基于数据库排他锁

除了可以通过增删操作数据表中的记录以外，还可以借助数据中自带的锁来实现分布式的锁。

还用刚刚创建的那张数据库表。可以通过数据库的排他锁来实现分布式锁。

基于MySql的InnoDB引擎，可以使用以下方法来实现加锁操作：

4.3.基于缓存实现分布式锁

相比基于数据库实现分布式锁的方案来说，基于缓存来实现在性能方面会表现得更好一些。

而且很多缓存是可以集群部署的，可以解决单点问题。

4.4.基于zookeeper实现分布式锁

基于zookeeper临时有序节点可以实现的分布式锁。

大致思想即为：每个客户端对某个方法加锁时，在zookeeper上的该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点，判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需要将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

4.4.1.解决方案

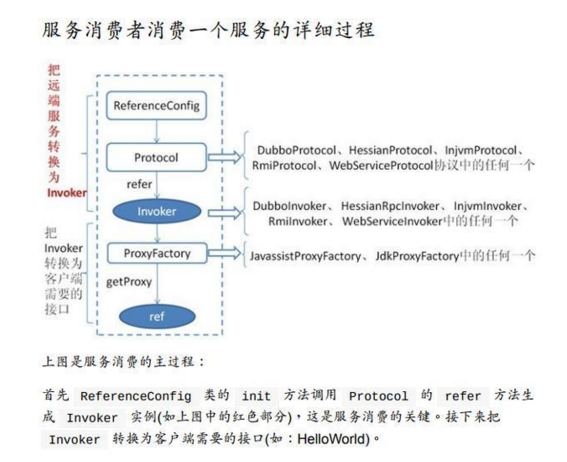
① 锁无法释放？使用Zookeeper可以有效的解决锁无法释放的问题，因为在创建锁的时候，客户端会在zk中创建一个临时节点，一旦客户端获取到锁之后突然挂掉(session连接断开)，那么这个临时节点就会自动删除，其他客户端就可以再次获得锁。

② 非阻塞锁？使用Zookeeper可以实现阻塞的锁，客户端可以通过zk中创建顺序节点，并且在节点上绑定监听器，一旦节点有变化，Zookeeper会通知客户端，客户端可以检查自己创建的节点是不是当前节点中序号最小的，如果是，那么自己就获取到锁，如果不一样就再创建一个临时的顺序节点，参与排队。

③ 单点问题？使用Zookeeper可以有效的解决单点问题，zk是集群部署的，只要集群中有半数以上的机器存活，就可以对外提供服务。

5.dubbo





6.zookeeper

6.1.Zookeeper是什么

分布、开源的应用程序协调服务，它是集群的管理者，监听者集群中各个节点的状态，根据节点的反馈进行下一步合理操作。

角色简介

Zookeeper角色分为三类：



6.2.Zookeeper基本原理

每个Server在工作过程中有三个状态：

① LOOKING：当前Server不知道leader是谁，正在搜寻。

② LEADING：当前Server即为选举出来的leader。

③ FOLLOWING：leader已经选举出来，当前Server与之同步。

6.2.1.选主流程

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的 Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。先介绍basic paxos流程：

Basic paxos：当前Server发起选举的线程,向所有Server发起询问,选举线程收到所有回复,计算zxid最大Server,并推荐此为leader，若此提议获得n/2+1票通过,此为leader，否则重复上述流程，直到leader选出。

Fast paxos:某Server首先向所有Server提议自己要成为leader，当其它Server收到提议以后，解决epoch和 zxid的冲突，并接受对方的提议，然后向对方发送接受提议完成的消息，重复这个流程，最后一定能选举出Leader。(即提议方解决其他所有epoch和 zxid的冲突,即为leader)。

广播：对某网络上所有主机发送数据包。

Paxos(帕克索斯)

Paxos(帕克索斯)算法的目的是为了解决分布式环境下一致性的问题

Paxos的两个组件

Proposer

提议发起者，处理客户端请求，将客户端的请求发送到集群中，以便决定这个值是否可以被批准。

Acceptor

提议批准者，负责处理接收到的提议，他们的回复就是一次投票。会存储一些状态来决定是否接收一个值