Mysql分库分表（分库有什么好处）

***加密算法，用过哪些签名和加密算法***

单向加密

不可逆

双向加密

对称加密和非对称加密

签名和加密

***CA证书***

1.带有私钥的证书

由Public Key Cryptography Standards #12，PKCS#12标准定义，包含了公钥和私钥的二进制格式的证书形式，以pfx作为证书文件后缀名。

2.二进制编码的证书

证书中没有私钥，DER 编码二进制格式的证书文件，以cer作为证书文件后缀名。

3.Base64编码的证书

证书中没有私钥，BASE64 编码格式的证书文件，也是以cer作为证书文件后缀名。

由定义可以看出，只有pfx格式的数字证书是包含有私钥的，cer格式的数字证书里面只有公钥没有私钥。

*如何向证书发布机构去申请证书*





Sql注入

***Syncronized与reentrantLock区别***

Syncronized基于阻塞同步悲观的并发策略，线程获得的是独占锁，只能依靠阻塞来等待线程的释放，cpu转换线程阻塞会引起上下文切换，频繁切换上下文导致效率降低。

ReentrantLock基于冲突检测的乐观并发策略，如果共享数据被占用，产生了冲突，就不断的重试，直到成功为止。这种策略不需要挂起线程成为非阻塞同步。

两者都是可重入的，但是ReentrantLock

* 等待可中断：Syncronized产生的互斥锁，会一直阻塞，不能被中断
* 可实现公平锁：Syncronized的锁是非公平的
* 锁可以绑定多个条件：ReentrantLock对象可以同时绑定多个Condition对象（名曰：条件变量或条件队列），而在synchronized中，锁对象的wait（）和notify（）或notifyAll（）方法可以实现一个隐含条件，但如果要和多于一个的条件关联的时候，就不得不额外地添加一个锁，而ReentrantLock则无需这么做，只需要多次调用newCondition（）方法即可。而且我们还可以通过绑定Condition对象来判断当前线程通知的是哪些线程（即与Condition对象绑定在一起的其他线程）。

慢查询优化 索引字段的建立，性别 加索引和不加索引的区别

Jvm如何进行优化，怎么设置各个参数 ；垃圾回收算法和垃圾回收期如何选型和优化

***HashMap和ConcurrentHashMap的区别，链表查询和红黑树查询的复杂度 红黑数是如何查询的***

***Session和Cookie的区别***

***NIO原理和使用场景***

***SpringMVC原理和场景***

***多线程ABA问题如何解决***

***多线程安全问题如何解决***

产生原因：多个线程操作共享数据

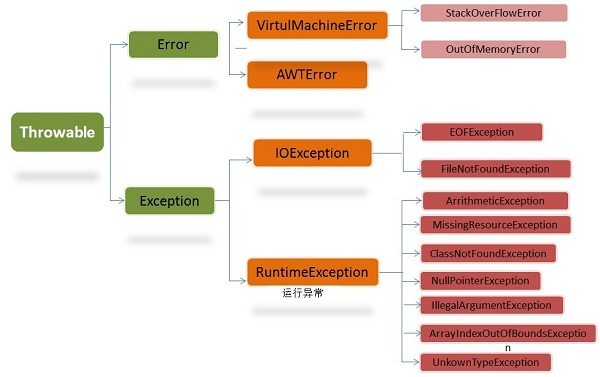
解决：加锁

***Spring框架中的缓存体系***

***如何保证分布式事务一致性，分布式事务实现方式***

***Java 重载和重写***

***Java异常的层次结构和继承关系***



Error：程序无法处理的错误，大多数错误与代码编写者执行的操作无关，而标识代码运行时JVM出现的问题。

Exception:程序本身可以处理的异常

* 运行时异常：NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。运行时异常的特点是Java编译器不会检查它。
* 非运行时异常（编译异常）：从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

***乐观锁和悲观锁***

***状态模式和策略模式的区别***

***内部类为什么会只加载一次***

***线程池构造类的5个参数及意义***

***ThreadExcutor的四种拒绝策略***

***单机上一个线程池正在处理服务如果忽然断电（正在处理和阻塞队列的请求如何处理）***

***使用无解阻塞队列会出现什么问题***

***Tcp三次握手 滑动窗口***

***http协议 get和post的区别***

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被Bookmark，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST么有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

HTTP协议中的两种发送请求的方法。HTTP的底层是TCP/IP。所以GET和POST的底层也是TCP/IP，也就是说，GET/POST都是TCP链接。GET和POST能做的事情是一样一样的。你要给GET加上request body，给POST带上url参数，技术上是完全行的通的

GET产生一个TCP数据包；POST产生两个TCP数据包。

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

也就是说，GET只需要汽车跑一趟就把货送到了，而POST得跑两趟，第一趟，先去和服务器打个招呼“嗨，我等下要送一批货来，你们打开门迎接我”，然后再回头把货送过去。

并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次

***高频请求接口，如何保证请求高效、顺序执行，并且避免瞬间压力导致的报警、瘫痪***

***Dubbo超时重试；dubbo超时时间设置***

***分布式事务和分布式锁（扣款不要出现负数）***

***分布式接口幂等性设计（不能重复扣款）***

只要调用接口成功，外部对接口的多次调用得到的结果是相同的。即执行多次和一次的效果是一样的。

以SQL为例，有下面三种场景，只有第三种场景需要开发人员使用其他策略保证幂等性：

SELECT col1 FROM tab1 WHER col2=2，无论执行多少次都不会改变状态，是天然的幂等。

UPDATE tab1 SET col1=1 WHERE col2=2，无论执行成功多少次状态都是一致的，因此也是幂等操作。

UPDATE tab1 SET col1=col1+1 WHERE col2=2，每次执行的结果都会发生变化，这种不是幂等的。

**保证幂等策略**

幂等需要通过唯一的业务单号来保证。也就是说相同的业务单号，认为是同一笔业务。使用这个唯一的业务单号来确保，后面多次的相同的业务单号的处理逻辑和执行效果是一致的。

下面以支付为例，在不考虑并发的情况下，实现幂等很简单：①先查询一下订单是否已经支付过，②如果已经支付过，则返回支付成功；如果没有支付，进行支付流程，修改订单状态为‘已支付’。

**防重复提交策略**

上述的保证幂等方案是分成两步的，第②步依赖第①步的查询结果，无法保证原子性的。在高并发下就会出现下面的情况：第二次请求在第一次请求第②步订单状态还没有修改为‘已支付状态’的情况下到来。既然得出了这个结论，余下的问题也就变得简单：把查询和变更状态操作加锁，将并行操作改为串行操作。

**乐观锁**

如果只是更新已有的数据，没有必要对业务进行加锁，设计表结构时使用乐观锁，一般通过version来做乐观锁，这样既能保证执行效率，又能保证幂等。例如：

UPDATE tab1 SET col1=1,version=version+1 WHERE version=#version#

不过，乐观锁存在失效的情况，就是常说的ABA问题，不过如果version版本一直是自增的就不会出现ABA的情况。（从网上找了一张图片很能说明乐观锁，引用过来，出自Mybatis对乐观锁的支持）

**防重表**

使用订单号orderNo做为去重表的唯一索引，每次请求都根据订单号向去重表中插入一条数据。第一次请求查询订单支付状态，当然订单没有支付，进行支付操作，无论成功与否，执行完后更新订单状态为成功或失败，删除去重表中的数据。后续的订单因为表中唯一索引而插入失败，则返回操作失败，直到第一次的请求完成（成功或失败）。可以看出防重表作用是加锁的功能。

**分布式锁**

这里使用的防重表可以使用分布式锁代替，比如Redis。订单发起支付请求，支付系统会去Redis缓存中查询是否存在该订单号的Key，如果不存在，则向Redis增加Key为订单号。查询订单支付已经支付，如果没有则进行支付，支付完成后删除该订单号的Key。通过Redis做到了分布式锁，只有这次订单订单支付请求完成，下次请求才能进来。相比去重表，将放并发做到了缓存中，较为高效。思路相同，同一时间只能完成一次支付请求。

**token令牌**

这种方式分成两个阶段：申请token阶段和支付阶段。

第一阶段，在进入到提交订单页面之前，需要订单系统根据用户信息向支付系统发起一次申请token的请求，支付系统将token保存到Redis缓存中，为第二阶段支付使用。

第二阶段，订单系统拿着申请到的token发起支付请求，支付系统会检查Redis中是否存在该token，如果存在，表示第一次发起支付请求，删除缓存中token后开始支付逻辑处理；如果缓存中不存在，表示非法请求。

实际上这里的token是一个信物，支付系统根据token确认，你是你妈的孩子。不足是需要系统间交互两次，流程较上述方法复杂。

**支付缓冲区**

把订单的支付请求都快速地接下来，一个快速接单的缓冲管道。后续使用异步任务处理管道中的数据，过滤掉重复的待支付订单。

优点是同步转异步，高吞吐。不足是不能及时地返回支付结果，需要后续监听支付结果的异步返回。

**幂等性接口的不足**

增加了额外控制幂等的业务逻辑，复杂化了业务功能；

把并行执行的功能改为串行执行，降低了执行效率。

因此除了业务上的特殊要求外，尽量不提供幂等的接口。

***分布式session设置，一致性***

缓存

***接口如何处理重复请求***

问题描述：用户下订单购买，因为各种原因（网络卡，快递点击等）重复提交2个或者以上一模一样的订单，由于是同时提交的，第一个订单执行扣款生成订单未完成时候，第二个已经进来了，导致付一笔钱购买了2次或多次商品

解决方案：

1、缓存lock，缓存此用户的操作行为，注意仅仅缓存操作的标志，下次进入判断此标志是否存在，存在即不进入数据库事务

2、应用程序application lock，和1相比，会阻塞其他用户的正常行为

3、模仿银行扣款机制，数据表建一个随机唯一标志，每次请求带上这个标志，操作的同时进行修改这个标志

4、应用程序生成唯一标志，数据库做字段的唯一索引

5、扣款为负数的事务进行回滚

6、使用事务的隔离级别

7、使用redis的incr控制用户的并发数，memcache的add也可以实现这种效果，memcached借助cas

不建议使用锁，使用不当，容易造成整个数据库挂掉

使用数据库唯一索引

每次request进来则往表里面插入数据

——成功,则可以继续操作（相当于获取锁）；

——失败,则说明有操作在进行。

基于缓存的计数器验证

每次request进来则新建一个以Id为key的计数器，然后＋1

如果>1（不能获得锁）: 说明有操作在进行，删除。

如果=1（获得锁）: 可以操作。

***如何限制一个号码一分钟内只能获取一次随机码***

我们在接到短信发送请求后，使用Redis的incr设置一个递增KEY（KEY由固定字符串+手机号码组成），并判断该KEY的数值，如果等于1，说明是第一个请求，我们将该KEY值有效期设置为一分钟；如果该KEY的数值大于1，说明是1分钟内的多次请求，这时我们直接返回短信获取频繁。

***如何设计一个对外服务接口的实现类，在1、2、3太主机上实现负载均衡和顺序轮询机制***

***一致Hash***

***YGC和FGC发生的具体场景***

1.YGC和FGC是什么

YGC ：对新生代堆进行gc。频率比较高，因为大部分对象的存活寿命较短，在新生代里被回收。性能耗费较小。

FGC ：全堆范围的gc。默认堆空间使用到达80%(可调整)的时候会触发fgc。以我们生产环境为例，一般比较少会触发fgc，有时10天或一周左右会有一次。

2.什么时候执行YGC和FGC

a.edn空间不足,执行 young gc

b.old空间不足，perm空间不足，调用方法System.gc() ，ygc时的悲观策略, dump live的内存信息时(jmap –dump:live)，都会执行full gc

***JVM老年代和新生代的比例***

***类加载过程，里面执行了哪些操作？***

加载方法不等于执行方法，初始化变量则会赋值

类加载顺序应为 加载静态方法-初始化静态变量-执行静态代码块

实例化时 先加载非静态方法-实例化非静态变量-执行构造代码块-执行构造函数

***海量数据解决方案***

1.使用缓存：

　　使用方式：1，使用程序直接保存到内存中。主要使用Map，尤其ConcurrentHashMap。

使用缓存框架。常用的框架：Ehcache，Memcache，Redis等。

　　最关键的问题是：什么时候创建缓存，以及其失效机制。

对于空数据的缓冲：最好用一个特定的类型值来保存，以区别空数据和未缓存的两种状态。

2.数据库优化：

　　1，表结构优化。

　　2，SQL语句优化，语法优化和处理逻辑优化。可记录各语句执行时间，有针对性的分析。

　　3，分区

　　4，分表

　　5，索引优化

　　6，使用存储过程代替直接操作

3.分离活跃数据

　　例如用户，可以分为活跃用户和不活跃用户。

4.批量读取和延迟修改

　　高并发情况可以将多个查询请求合并到一个。

　　高并发且频繁修改的可以暂存缓存中。

5.读写分离

　　上图，数据库服务器配置多个，配置主从数据库。写用主数据库，读用从数据库。

6.分布式数据库

　　将不同的表存放到不同的数据库中，然后再放到不同的服务器中。有些复杂问题，如：事务处理，多表查询。

7.NoSql和Hadoop

　　NoSql，not only SQL。没有关系型数据库那么多限制，比较灵活高效。

　　Hadoop，将一个表中的数据分层多块，保存到多个节点（分布式）。每一块数据都有多个节点保存（集群）。集群可以并行处理相同的数据，还可以保证数据的完整性。

二.高并发的解决方案

1.应用和静态资源分离。

　　将静态资源（js，css，图片等）放到专门的服务器中。

2.页面缓存

　　将应用生成的页面缓存起来可以节省大量cpu资源。

　　对于部分页面经常变换数据的，可以使用ajax来处理。

3.集群和分布式

　　集群，多台服务器具有相同的功能，主要起分流的作用。

　　分布式，将不同的业务放到不同的服务器中，处理一个请求可能需要多台服务器，进而提高一个请求的处理速度。

　　又分为静态资源集群和应用程序集群。后者较复杂，经常要考虑session同步等问题。

4.反向代理

　　客户端直接访问的服务器并不是直接提供服务的服务器，它从别的服务器获取资源，然后将结果返回给用户。

　　代理服务器和反向代理服务器：

　　代理服务器是代我们访获取资源，然后将结果返回。例如，访问外网的代理服务器。反向代理服务器是我们正常访问一台服务器的时候，服务器自己调用了别的服务器。

　　代理服务器我们主动使用，是为我们服务的，不需要有自己的域名；反向代理是服务器自己使用的，我们并不知道，有自己的域名。

5，CDN

　　CDN是一种特殊的集群页面缓冲服务器，和普通的集群的多台页面缓冲服务器相比主要区别是：其存放位置和分配请求方式不同。

　　CDN的服务器分布在全国各地，接收到请求后会将请求分配到最合适的CDN服务器节点来获取数据。其每一个CDN节点就是一个页面缓存服务器。

　　分配方式：并不是普通的负载均衡，而是专门的CDN域名解析服务器在解析域名的时候就分配好的，一般的做饭是：ISP那里使用CNAME将域名解析到一个特定的域名，然后再将解析到的那个域名用专门的CDN服务器解析（返回给浏览器，再访问）到相应的CDN节点。每个节点可能也集群了多台服务器。

不用循环数组求和