# 2019面试题总结

1.索引

### 主键和唯一索引的区别？

在创建主键的同时会生成对应的唯一索引，主键在保证数据唯一性的同时不允许为 空，而唯一可以有一个为空数据项，一个表中只能有一个主键，但是一个主键可以 有多个字段，一个表中可以有多个唯一索引。

索引概述

1、索引的概念

索引就是为了提高数据的检索速度。

数据库的索引类似于书籍的索引。

在书籍中，索引允许用户不必翻阅完整个书就能迅速地找到所需要的信息。

在数据库中，索引也允许数据库程序迅速地找到表中的数据，

而不必扫描整个数据库.

2、索引的优点

　　1.创建唯一性索引，保证数据库表中每一行数据的唯一性

　　2.大大加快数据的检索速度，这也是创建索引的最主要的原因

　　3.减少磁盘IO（向字典一样可以直接定位）

3、索引的缺点

1.创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加

2.索引需要占用额外的物理空间

3.当对表中的数据进行增加、删除和修改的时候，

索引也要动态的维护，降低了数据的维护速度

4、索引的分类

　　1.普通索引和唯一性索引

　　 普通索引：CREATE INDEX mycolumn\_index ON mytable (myclumn)

　　 唯一性索引：保证在索引列中的全部数据是唯一的

　　 CREATE unique INDEX mycolumn\_index ON mytable (myclumn)

2. 单个索引和复合索引

　　 单个索引：对单个字段建立索引

　　 复合索引：又叫组合索引，在索引建立语句中同时包含多个字段名，

　　 最多16个字段

　　 CREATE INDEX name\_index ON userInfo(firstname,lastname)

3.顺序索引，散列索引,位图索引

2.高并发

互联网分布式架构设计，提高系统并发能力的方式，方法论上主要有两种：垂直扩展（Scale Up）与水平扩展（Scale Out）。

垂直扩展：提升单机处理能力。垂直扩展的方式又有两种：

（1）增强单机硬件性能，例如：增加CPU核数如32核，升级更好的网卡如万兆，升级更好的硬盘如SSD，扩充硬盘容量如2T，扩充系统内存如128G；

（2）提升单机架构性能，例如：使用Cache来减少IO次数，使用异步来增加单服务吞吐量，使用无锁数据结构来减少响应时间；

在互联网业务发展非常迅猛的早期，如果预算不是问题，强烈建议使用“增强单机硬件性能”的方式提升系统并发能力，因为这个阶段，公司的战略往往是发展业务抢时间，而“增强单机硬件性能”往往是最快的方法。

不管是提升单机硬件性能，还是提升单机架构性能，都有一个致命的不足：单机性能总是有极限的。所以互联网分布式架构设计高并发终极解决方案还是水平扩展。

水平扩展：只要增加服务器数量，就能线性扩充系统性能

详细的细节处理一般就是redis加锁，方法上锁，等方法处理完成之后再放开

相对于分布式来讲，高并发在解决的问题上会集中一些，其反应的是同时有多少量：比如在线直播服务，同时有上万人观看。

高并发可以通过分布式技术去解决，将并发流量分到不同的物理服务器上。但除此之外，还可以有很多其他优化手段：比如使用缓存系统，将所有的，静态内容放到CDN等；还可以使用多线程技术将一台服务器的服务能力最大化。

什么是分布式？

分布式更多的一个概念，是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段。该领域需要解决的问题极多，在不同的技术层面上，又包括：分布式文件系统、分布式缓存、分布式数据库、分布式计算等，一些名词如Hadoop、zookeeper、MQ等都跟分布式有关。从理念上讲，分布式的实现有两种形式：

水平扩展：当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供相当的服务；

垂直拆分：前端有多种查询需求时，一台机器扛不住，可以将不同的需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

什么是高并发？

相对于分布式来讲，高并发在解决的问题上会集中一些，其反应的是同时有多少量：比如在线直播服务，同时有上万人观看。

高并发可以通过分布式技术去解决，将并发流量分到不同的物理服务器上。但除此之外，还可以有很多其他优化手段：比如使用缓存系统，将所有的，静态内容放到CDN等；还可以使用多线程技术将一台服务器的服务能力最大化。

什么是多线程？

多线程是指从软件或者硬件上实现多个线程并发执行的技术，它更多的是解决CPU调度多个进程的问题，从而让这些进程看上去是同时执行（实际是交替运行的）。

这几个概念中，多线程解决的问题是最明确的，手段也是比较单一的，基本上遇到的最大问题就是线程安全。在JAVA语言中，需要对JVM内存模型、指令重排等深入了解，才能写出一份高质量的多线程代码。

总结一下：

●分布式是从物理资源的角度去将不同的机器组成一个整体对外服务，技术范围非常广且难度非常大，有了这个基础，高并发、高吞吐等系统很容易构建；

● 高并发是从业务角度去描述系统的能力，实现高并发的手段可以采用分布式，也可以采用诸如缓存、CDN等，当然也包括多线程；

● 多线程则聚焦于如何使用编程语言将CPU调度能力最大化。

3.分布式实现

分布式其实就是不同的的服务进行拆分，用多台服务器布置服务。Zookeeper和dubbo进行内联，zookeeper注册中心，dubbo内部rpc通讯，

分布式更多的一个概念，是为了解决单个物理服务器容量和性能瓶颈问题而采用的优化手段。该领域需要解决的问题极多，在不同的技术层面上，又包括：分布式文件系统、分布式缓存、分布式数据库、分布式计算等，一些名词如Hadoop、zookeeper、MQ等都跟分布式有关。从理念上讲，分布式的实现有两种形式：

水平扩展：当一台机器扛不住流量时，就通过添加机器的方式，将流量平分到所有服务器上，所有机器都可以提供相当的服务；

垂直拆分：前端有多种查询需求时，一台机器扛不住，可以将不同的需求分发到不同的机器上，比如A机器处理余票查询的请求，B机器处理支付的请求。

**1、纯负载均衡形式**

在集群前面，前置一个流量分发的组件进行流量分发，整个集群的机器提供无差别的服务，这在常见的 web 服务器中是最最常见的。目前比较主流的方式就是整个集群机器上云，根据实时的调用量进行云服务器弹性伸缩。常见的负载均衡有硬件层面的 F5、软件层面的 nginx 等。

**2、领导选举型**

整个集群的消息都会转发到集群的领导这里，是一种 master-slavers，区别只是这个 master 是被临时选举出来的，一旦 master 宕机，集群会立刻选举出一个新的领导，继续对外提供服务。使用领导选举型架构的典型的应用有 ElasticSearch，zookeeper。



**3、区块链型**

整个集群的每一个节点都可以进行记录，但是记录的内容要得到整个集群 N 个机器的认可才是合法的。典型的应用有 Bit Coin，以及 Hyperledger。



**4、master-slaver型**

整个集群以某台 master 为中枢，进行集群的调度。交互是这样，一般会把所有的管理类型的数据放到 master 上，而把具体的数据放到 slaver 上，实际进行调用的时候，client 先调用 master 获取数据所存放的 server 的 信息，再自行跟 slave 进行交互。典型的系统有 Hadoop。集群，HBase 集群，Redis 集群等。

**5、规则型一致性Hash**

这种架构类型一般出现在数据库分库分表的设计中。按照规则进行分库分表，在查询之前使用规则引擎进行库和表的确认，再对具体的应用进行访问。为什么要用一致性 Hash ？其实用什么都可以，只是对于这类应用来说一致性 Hash 比较常见而已。

4.Redis集群

redis是一个开源的key value存储系统，受到了广大互联网公司的青睐。redis3.0版本之前只支持单例模式，在3.0版本及以后才支持集群，我这里用的是redis3.0.0版本；

redis集群采用P2P模式，是完全去中心化的，不存在中心节点或者代理节点；

redis集群是没有统一的入口的，客户端（client）连接集群的时候连接集群中的任意节点（node）即可，集群内部的节点是相互通信的（PING-PONG机制），每个节点都是一个redis实例；

为了实现集群的高可用，即判断节点是否健康（能否正常使用），redis-cluster有这么一个投票容错机制：如果集群中超过半数的节点投票认为某个节点挂了，那么这个节点就挂了（fail）。这是判断节点是否挂了的方法；

那么如何判断集群是否挂了呢? -> 如果集群中任意一个节点挂了，而且该节点没有从节点（备份节点），那么这个集群就挂了。这是判断集群是否挂了的方法；

那么为什么任意一个节点挂了（没有从节点）这个集群就挂了呢？ -> 因为集群内置了16384个slot（哈希槽），并且把所有的物理节点映射到了这16384[0-16383]个slot上，或者说把这些slot均等的分配给了各个节点。当需要在Redis集群存放一个数据（key-value）时，redis会先对这个key进行crc16算法，然后得到一个结果。再把这个结果对16384进行求余，这个余数会对应[0-16383]其中一个槽，进而决定key-value存储到哪个节点中。所以一旦某个节点挂了，该节点对应的slot就无法使用，那么就会导致集群无法正常工作。

综上所述，每个Redis集群理论上最多可以有16384个节点。

二、集群搭建需要的环境

2.1 Redis集群至少需要3个节点，因为投票容错机制要求超过半数节点认为某个节点挂了该节点才是挂了，所以2个节点无法构成集群。

2.2 要保证集群的高可用，需要每个节点都有从节点，也就是备份节点，所以Redis集群至少需要6台服务器。因为我没有那么多服务器，也启动不了那么多虚拟机，所在这里搭建的是伪分布式集群，即一台服务器虚拟运行6个redis实例，修改端口号为（7001-7006），当然实际生产环境的Redis集群搭建和这里是一样的。

————————————————

版权声明：本文为CSDN博主「凉凉的西瓜」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/qq_42815754/article/details/82912130>

一、单节点实例  
　　单节点实例还是比较简单的，平时做个测试，写个小程序如果需要用到缓存的话，启动一个Redis还是很轻松的，做为一个key/value数据库也是可以胜任的。单节点部署参照：http://www.cnblogs.com/yiwangzhibujian/p/7053840.html 中的初级教程。

二、主从模式（master/slaver）  
　　主从模式的简介和配置，此处不再说，

2.1 搭建方式  
　　主从模式的搭建还是很简单的，官网的介绍就更加简单了，只需要在配置文件中加上一行配置：

slaveof 192.168.1.1 6379  
　　指明master的ip和端口号就可以了，实际上真的这么简单。

2.2 其他配置  
　　在redis.conf中，还有一些关于主从的其他配置，按需配置即可。

首先谈谈我对主从模式的必要性：

主从模式的一个作用是备份数据，这样当一个节点损坏（指不可恢复的硬件损坏）时，数据因为有备份，可以方便恢复。  
另一个作用是负载均衡，所有客户端都访问一个节点肯定会影响Redis工作效率，有了主从以后，查询操作就可以通过查询从节点来完成。  
　　对主从模式必须的理解（结论已经验证过，可以自行验证）：

一个Master可以有多个Slaves  
默认配置下，master节点可以进行读和写，slave节点只能进行读操作，写操作被禁止  
不要修改配置让slave节点支持写操作，没有意义，原因一，写入的数据不会被同步到其他节点；原因二，当master节点修改同一条数据后，slave节点的数据会被覆盖掉  
slave节点挂了不影响其他slave节点的读和master节点的读和写，重新启动后会将数据从master节点同步过来  
master节点挂了以后，不影响slave节点的读，Redis将不再提供写服务，master节点启动后Redis将重新对外提供写服务。  
master节点挂了以后，不会slave节点重新选一个master  
　　对有密码的情况说明一下，当master节点设置密码时：

客户端访问master需要密码  
启动slave需要密码，在配置中进行配置即可  
客户端访问slave不需要密码  
2.1 主从节点的缺点  
　　主从模式的缺点其实从上面的描述中可以得出：

master节点挂了以后，redis就不能对外提供写服务了，因为剩下的slave不能成为master  
　　这个缺点影响是很大的，尤其是对生产环境来说，是一刻都不能停止服务的，所以一般的生产坏境是不会单单只有主从模式的。所以有了下面的sentinel模式。

三、sentinel模式  
　　sentinel模式的简介和配置，此处不再说，参照下面的博客：Redis Sentinel模式详解https://www.cnblogs.com/yiwangzhibujian/p/7053848.html。

sentinel的中文含义是哨兵、守卫。也就是说既然主从模式中，当master节点挂了以后，slave节点不能主动选举一个master节点出来，那么我就安排一个或多个sentinel来做这件事，当sentinel发现master节点挂了以后，sentinel就会从slave中重新选举一个master。

对sentinel模式的理解：

sentinel模式是建立在主从模式的基础上，如果只有一个Redis节点，sentinel就没有任何意义  
当master节点挂了以后，sentinel会在slave中选择一个做为master，并修改它们的配置文件，其他slave的配置文件也会被修改，比如slaveof属性会指向新的master  
当master节点重新启动后，它将不再是master而是做为slave接收新的master节点的同步数据  
sentinel因为也是一个进程有挂掉的可能，所以sentinel也会启动多个形成一个sentinel集群  
当主从模式配置密码时，sentinel也会同步将配置信息修改到配置文件中，不许要担心。  
一个sentinel或sentinel集群可以管理多个主从Redis。  
sentinel最好不要和Redis部署在同一台机器，不然Redis的服务器挂了以后，sentinel也挂了  
sentinel监控的Redis集群都会定义一个master名字，这个名字代表Redis集群的master Redis。  
　　当使用sentinel模式的时候，客户端就不要直接连接Redis，而是连接sentinel的ip和port，由sentinel来提供具体的可提供服务的Redis实现，这样当master节点挂掉以后，sentinel就会感知并将新的master节点提供给使用者。

sentinel模式基本可以满足一般生产的需求，具备高可用性。但是当数据量过大到一台服务器存放不下的情况时，主从模式或sentinel模式就不能满足需求了，这个时候需要对存储的数据进行分片，将数据存储到多个Redis实例中，就是下面要讲的。

四、cluster模式  
　　sentinel模式的简介和配置，此处不再说，参照下面的博客：Redis cluster模式详解https://www.cnblogs.com/yiwangzhibujian/p/7053851.html`

cluster的出现是为了解决单机Redis容量有限的问题，将Redis的数据根据一定的规则分配到多台机器。对cluster的一些理解：

cluster可以说是sentinel和主从模式的结合体，通过cluster可以实现主从和master重选功能，所以如果配置两个副本三个分片的话，就需要六个Redis实例。  
因为Redis的数据是根据一定规则分配到cluster的不同机器的，当数据量过大时，可以新增机器进行扩容  
　　这种模式适合数据量巨大的缓存要求，当数据量不是很大使用sentinel即可。

5.jvm内存模型，调优，死锁

## jvm的内存结构

java虚拟机的内存结构分为堆(heap)和栈(stack),堆里面存放是对象实例也就是new出来的对象。栈里面存放的是基本数据类型以及引用数据类型的地址。

对于所谓的常量是存储在方法区的常量池里面。

jvm 内存结构？

堆： 逻辑上是连续,物理上可以处于不连续的内存空间中，

里面存储的是对象实例以及数组。可以细分为新生代，老生代。

通过-Xmx和-Xms控制大小。

虚拟机栈：基本数据类型，对象引用(地址，指针)。

本地方法栈（了解）:它与虚拟机栈发挥的作用差不多，区别在于虚拟机栈为java方法

的执行提供服务，而本地方法栈为虚拟机使用到的Native(本地)方法服务。

方法区：放了所加载的类的信息（名称、修饰符等）、类中的静态变量、

类中定义为final类型的常量、类中的Field信息、类中的方法信息

在Sun JDK中这块区域对应的为PermanetGeneration，又称为持久代，

默认为64M，可通过-XX:PermSize以及-XX:MaxPermSize来指定其大小

在服务器启动的时候报内存溢出是因为方法区太小，也就相当于持久代的内存太小。

通过-XX:PermSize以及-XX:MaxPermSize来指定其大小，可以解决这个问题。

常量池是方法区的一部分,用来存储常量信息。如String就存储在

常量池中。

计数器（了解）:通过该计数器的值来选取下一条要执行的字节码指令。

jvm的调优?

开启-Server模式，增大堆的大小，以及持久代的大小，从而

提高程序的运行效率，并且将初始堆大小和最大堆大小设置为

一样的值从而避免了堆增长会带来额外压力。持久代大小的设置

同理，也设置为初始大小和最大大小一样大。

6.MQ应用场景以及常见问题规避手段，幂等性解决方案

MQ,Message queue,消息队列，就是指保存消息的一个容器。

现在常用的MQ组件有activeMQ、rabbitMQ、rocketMQ、zeroMQ,当然近年来火热的kafka,从某些场景来说，也是MQ，当然kafka的功能更加强大，虽然不同的MQ都有自己的特点和优势，但是，不管是哪种MQ，都有MQ本身自带的一些特点

应用解耦（异步）

系统之间进行数据交互的时候，在时效性和稳定性之间我们都需要进行选择。基于线程的异步处理，能确保用户体验，但是极端情况下可能会出现异常，影响系统的稳定性，而同步调用很多时候无法保证理想的性能，那么我们就可以用MQ来进行处理。上游系统将数据投递到MQ，下游系统取MQ的数据进行消费，投递和消费可以用同步的方式处理，因为MQ接收数据的性能是非常高的，不会影响上游系统的性能，那么下游系统的及时率能保证吗？当然可以，不然就不会有下面的一个应用场景。

通知

这里就用到了前文一个重要的特点，发布订阅，下游系统一直在监听MQ的数据，如果MQ有数据，下游系统则会按照 先进先出 这样的规则， 逐条进行消费 ，而上游系统只需要将数据存入MQ里，这样就既降低了不同系统之间的耦合度，同时也确保了消息通知的及时性，而且也不影响上游系统的性能。

限流

上文有说了一个非常重要的特性，MQ 数据是只有一条数据在使用中。 在很多存在并发，而又对数据一致性要求高，而且对性能要求也高的场景，如何保证，那么MQ就能起这个作用了。不管多少流量进来，MQ都会让你遵守规则，排除处理，不会因为其他原因，导致并发的问题，而出现很多意想不到脏数据。

数据分发

MQ的发布订阅肯定不是只是简单的一对一，一个上游和一个下游的关系，MQ中间件基本都是支持一对多或者广播的模式，而且都可以根据规则选择分发的对象。这样上游的一份数据，众多下游系统中，可以根据规则选择是否接收这些数据，这样扩展性就很强了。

PS:上文中的上游和下游，在MQ更多的是叫做生产者（producer）和消费者（consumer）。

分布式事务

分布式事务是我们开发中一直尽量避免的一个技术点，但是，现在越来越多的系统是基于微服务架构开发，那么分布式事务成为必须要面对的难题，解决分布式事务有一个比较容易理解的方案，就是二次提交。基于MQ的特点，MQ作为二次提交的中间节点，负责存储请求数据，在失败的情况可以进行多次尝试，或者基于MQ中的队列数据进行回滚操作，是一个既能保证性能，又能保证业务一致性的方案，当然，这个方案的主要问题就是定制化较多，有一定的开发工作量。

https://blog.csdn.net/qq\_36236890/article/details/81174504

如何保证消息消费时的幂等性

其实消息重复消费的主要原因在于回馈机制（RabbitMQ是ack，Kafka是offset)，在某些场景中我们采用的回馈机制不同，原因也不同，例如消费者消费完消息后回复ack, 但是刚消费完还没来得及提交系统就重启了，这时候上来就pull消息的时候由于没有提交ack或者offset，消费的还是上条消息。

那么如何怎么来保证消息消费的幂等性呢？实际上我们只要保证多条相同的数据过来的时候只处理一条或者说多条处理和处理一条造成的结果相同即可，但是具体怎么做要根据业务需求来定，例如入库消息，先查一下消息是否已经入库啊或者说搞个唯一约束啊什么的，还有一些是天生保证幂等性就根本不用去管，例如redis就是天然幂等性。

还有一个问题，消费者消费消息的时候在某些场景下要放过消费不了的消息，遇到消费不了的消息通过日志记录一下或者搞个什么措施以后再来处理，但是一定要放过消息，因为在某些场景下例如spring-rabbitmq的默认回馈策略是出现异常就没有提交ack，导致了一直在重发那条消费异常的消息，而且一直还消费不了，这就尴尬了，后果你会懂的。

六、如何保证消息的可靠性传输？

由于笔者只使用和实践过RabbitMQ和Kafka，RocketMQ和ActiveMQ了解的不深，所以分析一下RabbitMQ和Kafka的消息可靠性传输的问题。、

（一）RabbitMQ

（1）生产者弄丢了数据

　　生产者将数据发送到RabbitMQ的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。此时可以选择用RabbitMQ提供的事务功能，就是生产者发送数据之前开启RabbitMQ事务（channel.txSelect），然后发送消息，如果消息没有成功被RabbitMQ接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。但是问题是，RabbitMQ事务机制一搞，基本上吞吐量会下来，因为太耗性能。

所以一般来说，如果你要确保说写RabbitMQ的消息别丢，可以开启confirm模式，在生产者那里设置开启confirm模式之后，你每次写的消息都会分配一个唯一的id，然后如果写入了RabbitMQ中，RabbitMQ会给你回传一个ack消息，告诉你说这个消息ok了。如果RabbitMQ没能处理这个消息，会回调你一个nack接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息id的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和cnofirm机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是confirm机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息RabbitMQ接收了之后会异步回调你一个接口通知你这个消息接收到了。

所以一般在生产者这块避免数据丢失，都是用confirm机制的。

（2）RabbitMQ弄丢了数据

就是RabbitMQ自己弄丢了数据，这个你必须开启RabbitMQ的持久化，就是消息写入之后会持久化到磁盘，哪怕是RabbitMQ自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。除非极其罕见的是，RabbitMQ还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建queue的时候将其设置为持久化的，这样就可以保证RabbitMQ持久化queue的元数据，但是不会持久化queue里的数据；第二个是发送消息的时候将消息的deliveryMode设置为2，就是将消息设置为持久化的，此时RabbitMQ就会将消息持久化到磁盘上去。必须要同时设置这两个持久化才行，RabbitMQ哪怕是挂了，再次重启，也会从磁盘上重启恢复queue，恢复这个queue里的数据。

而且持久化可以跟生产者那边的confirm机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者ack了，所以哪怕是在持久化到磁盘之前，RabbitMQ挂了，数据丢了，生产者收不到ack，你也是可以自己重发的。

哪怕是你给RabbitMQ开启了持久化机制，也有一种可能，就是这个消息写到了RabbitMQ中，但是还没来得及持久化到磁盘上，结果不巧，此时RabbitMQ挂了，就会导致内存里的一点点数据会丢失。

（3）消费端弄丢了数据

RabbitMQ如果丢失了数据，主要是因为你消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，那么就尴尬了，RabbitMQ认为你都消费了，这数据就丢了。

这个时候得用RabbitMQ提供的ack机制，简单来说，就是你关闭RabbitMQ自动ack，可以通过一个api来调用就行，然后每次你自己代码里确保处理完的时候，再程序里ack一把。这样的话，如果你还没处理完，不就没有ack？那RabbitMQ就认为你还没处理完，这个时候RabbitMQ会把这个消费分配给别的consumer去处理，消息是不会丢的。

（二）Kafka

（1）消费端弄丢了数据

唯一可能导致消费者弄丢数据的情况，就是说，你那个消费到了这个消息，然后消费者那边自动提交了offset，让kafka以为你已经消费好了这个消息，其实你刚准备处理这个消息，你还没处理，你自己就挂了，此时这条消息就丢咯。

大家都知道kafka会自动提交offset，那么只要关闭自动提交offset，在处理完之后自己手动提交offset，就可以保证数据不会丢。但是此时确实还是会重复消费，比如你刚处理完，还没提交offset，结果自己挂了，此时肯定会重复消费一次，自己保证幂等性就好了。

生产环境碰到的一个问题，就是说我们的kafka消费者消费到了数据之后是写到一个内存的queue里先缓冲一下，结果有的时候，你刚把消息写入内存queue，然后消费者会自动提交offset。

然后此时我们重启了系统，就会导致内存queue里还没来得及处理的数据就丢失了

（2）kafka弄丢了数据

这块比较常见的一个场景，就是kafka某个broker宕机，然后重新选举partiton的leader时。大家想想，要是此时其他的follower刚好还有些数据没有同步，结果此时leader挂了，然后选举某个follower成leader之后，他不就少了一些数据？这就丢了一些数据啊。

生产环境也遇到过，我们也是，之前kafka的leader机器宕机了，将follower切换为leader之后，就会发现说这个数据就丢了。

所以此时一般是要求起码设置如下4个参数：

给这个topic设置replication.factor参数：这个值必须大于1，要求每个partition必须有至少2个副本。

在kafka服务端设置min.insync.replicas参数：这个值必须大于1，这个是要求一个leader至少感知到有至少一个follower还跟自己保持联系，没掉队，这样才能确保leader挂了还有一个follower吧。

在producer端设置acks=all：这个是要求每条数据，必须是写入所有replica之后，才能认为是写成功了。

在producer端设置retries=MAX（很大很大很大的一个值，无限次重试的意思）：这个是要求一旦写入失败，就无限重试，卡在这里了。

（3）生产者会不会弄丢数据

如果按照上述的思路设置了ack=all，一定不会丢，要求是，你的leader接收到消息，所有的follower都同步到了消息之后，才认为本次写成功了。如果没满足这个条件，生产者会自动不断的重试，重试无限次。

六、如何保证消息的顺序性

因为在某些情况下我们扔进MQ中的消息是要严格保证顺序的，尤其涉及到订单什么的业务需求，消费的时候也是要严格保证顺序，不然会出大问题的。

先看看顺序会错乱的俩场景

rabbitmq：一个queue，多个consumer，这不明显乱了

kafka：一个topic，一个partition，一个consumer，内部多线程，这不也明显乱了

　　如何来保证消息的顺序性呢？

rabbitmq：拆分多个queue，每个queue一个consumer，就是多一些queue而已，确实是麻烦点；或者就一个queue但是对应一个consumer，然后这个consumer内部用内存队列做排队，然后分发给底层不同的worker来处理。

kafka：一个topic，一个partition，一个consumer，内部单线程消费，写N个内存queue，然后N个线程分别消费一个内存queue即可。

七、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时怎么解决？

（一）、大量消息在mq里积压了几个小时了还没解决

几千万条数据在MQ里积压了七八个小时，从下午4点多，积压到了晚上很晚，10点多，11点多

这个是我们真实遇到过的一个场景，确实是线上故障了，这个时候要不然就是修复consumer的问题，让他恢复消费速度，然后傻傻的等待几个小时消费完毕。这个肯定不能在面试的时候说吧。

一个消费者一秒是1000条，一秒3个消费者是3000条，一分钟是18万条，1000多万条，所以如果你积压了几百万到上千万的数据，即使消费者恢复了，也需要大概1小时的时间才能恢复过来。

一般这个时候，只能操作临时紧急扩容了，具体操作步骤和思路如下：

先修复consumer的问题，确保其恢复消费速度，然后将现有cnosumer都停掉。

新建一个topic，partition是原来的10倍，临时建立好原先10倍或者20倍的queue数量。

然后写一个临时的分发数据的consumer程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的10倍数量的queue。

接着临时征用10倍的机器来部署consumer，每一批consumer消费一个临时queue的数据。

这种做法相当于是临时将queue资源和consumer资源扩大10倍，以正常的10倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署架构，重新用原先的consumer机器来消费消息。

（二）、消息队列过期失效问题

假设你用的是rabbitmq，rabbitmq是可以设置过期时间的，就是TTL，如果消息在queue中积压超过一定的时间就会被rabbitmq给清理掉，这个数据就没了。那这就是第二个坑了。这就不是说数据会大量积压在mq里，而是大量的数据会直接搞丢。

这个情况下，就不是说要增加consumer消费积压的消息，因为实际上没啥积压，而是丢了大量的消息。我们可以采取一个方案，就是批量重导，这个我们之前线上也有类似的场景干过。就是大量积压的时候，我们当时就直接丢弃数据了，然后等过了高峰期以后，比如大家一起喝咖啡熬夜到晚上12点以后，用户都睡觉了。

这个时候我们就开始写程序，将丢失的那批数据，写个临时程序，一点一点的查出来，然后重新灌入mq里面去，把白天丢的数据给他补回来。也只能是这样了。

假设1万个订单积压在mq里面，没有处理，其中1000个订单都丢了，你只能手动写程序把那1000个订单给查出来，手动发到mq里去再补一次。

(三)、消息队列满了怎么搞？

如果走的方式是消息积压在mq里，那么如果你很长时间都没处理掉，此时导致mq都快写满了，咋办？这个还有别的办法吗？没有，谁让你第一个方案执行的太慢了，你临时写程序，接入数据来消费，消费一个丢弃一个，都不要了，快速消费掉所有的消息。然后走第二个方案，到了晚上再补数据吧。

7.一亿条数据分页查询优化方案  
 因为分页是全表扫描，如果想快就尽可能让分页走索引，越到后面就要跳过很多数据，所以数据量越大到后面的页码速度就越慢，如何让分页走索引，这才是优化的重点，初级方案是内连接，但是这不是最理想的，虽然走索引了，但是走的是内连接，就等于会产生临时表，效率略有提高，但是并没有提高多少，索引也是一张表，但是索引因为字段少，效率是非常高的，所以性能的提升来自于索引，

高级优化是水平分表，比如用户表有三个字段 id name sex，每张表容量是10000条，那么根据容量的飙升会产生user01 user02 user03 这样的表，分页时根据页数查询对应的分表，这样单表的数量比较小，即时走全表扫描也比较快，单表500万+数据量是mysql的瓶颈，如果把500w水平分割为10个表，那么单表的效率就比较高了，但是数据合并是个问题，通常使用中间件，初级方案shard-jdbc来自于当当网开发团队，中级方案Mycat，高级方案暂时不了解，主要配置得了解，shard\_jdbc能够实现的功能，以及怎么配置，mycat配置了哪些重要文件，比如，server.xml,schema.xml.rule.xml

8.设计模式

在软件程序的设计中应当遵循最基础的原则。

一、单一职责原则，一个类应当只负责一项或一类职责。

二接口隔离原则，一个类对另一个类的依赖应当建立在最小的接口上，使用多个隔离接口降低耦合。

三依赖倒转原则，高层不应该依赖其低层，应该两者都依赖抽象。不依赖具体但具体依赖于抽象。对于具体的稳定性，抽象的稳定性要比其高的多。对于Java来说，具体指的是详细的实现类，而抽象指的是抽象类和接口。即使用抽象类和接口，目的是为了制定规范，而不是为了去依赖具体。这样可以提高很大的可扩展性。

四里氏替换原则。所有涉及父类和子类相关方法的地方都应该全透明化。换句话说，也就是所有引用超类货基类的地方，都必须让子类透明化使用所有透明。使用继承时应当尽量减少子类重写父类的方法。事实告诉我们，继承实际上让两个类的耦合变得增强了。适当的使用聚合依赖组合来降低耦合。

五，开闭原则又称为ocp原则，是Java中最基础最基础的原则。对扩展开放对修改关闭。我们在写新功能的时候应该尽量考虑代码的扩展性，通过扩展来解决问题，而不是在原有的基础上修改代码，这样会带来很多新的问题。

六迪米特法则又称最小知道法则，类与类之间的交流应当保持在最小，广实现类中方法的逻辑有都复杂，都分装在自己类内部，对外只暴露出提供的公开方法。类与类之间关系越密切，耦合度越高。陌生的类最好不要以局部变量的方式出现在某一个类的内部。

七合成复用原则。让使用聚合合成来完成功能。而不是使用继承。聚合指的是通过构造方法set方法。方法形参的方式依赖某个类。合成就是new出来,一起出现，一起消亡。

总结。

单一类职责，最好保证类方法单一职责。

剥离组件最小化，用到什么聚合什么

依赖于抽象，而不依赖于具体，依赖于高层，而不是依赖于低层。

如何利用好继承的特性，提高代码的可复用性

对修改关闭对扩展开放。

最小知道原则，类和类之间的耦合越少越好，用好直接类，拒绝陌生类。

能用合成聚合代替的。就不用继承，降低耦合。

# 九，cloud的组件各作用。

Spring Cloud为开发人员提供了快速构建分布式系统的一些工具，既然是分布式，那就要有一个中心，我们叫做服务的注册和发现中心（Eureka），有了中心之后，我们就可以将各种服务往里面注册使得各个服务可以相互感知到。这个时候，我们往往需要服务之间进行交流与交互，那么服务消费者便出现（rest+ribbon），开发者越来越强迫症，所以基于ribbon的服务消费者feign（并且内置有断路器）出现了。服务多了，我们还要让服务健壮，所以一般会有两个以上一模一样的服务作为集群或负载均衡，然而不管如何做，某一服务都可能故障，所以便有了断路器（Hystrix），断路器的作用是当远程服务访问不到，可以改成访问本地的某一方法。好了，现在我们已经有了一个服务了，接下来就是要把所有的服务接口统一起来暴露出去给客户用了，这时候就有了路由网关（zuul），作用就是负载均衡统一服务端口和反向代理，注意到统一这两个字之后，我们就开始有想法，比如统一过滤，统一权限认证等等统一的东西都可以放在这里。然后开发者有发现一个问题，服务一多，改配置太麻烦了，需要有个东西来管理，最好还能在线修改配置，这时候分布式配置中心（Spring Cloud Config）就出现了，它实现了将所有服务的配置文件都抽取到一个统一的地方，这时候程序员又要发妖了，想要更改配置的时候，服务能够知道并且热更新配置，那么就需要一个消息传递工具——消息总线（Spring Cloud Bus），通过这个总线向其他服务传递消息。接下来，我们想要知道各个服务之间的调用关系间接得到服务之间的依赖，那么就需要服务追踪组件（zipkin ，SpringCloud Sleuth集成了zipkin）了。

# 十缓存穿透和雪崩解决方案。

缓存穿透、缓存击穿、缓存雪崩区别和解决方案 原创

每天进步一点点yes

2018-09-19 阅读数：35247

关注

一、缓存处理流程

前台请求，后台先从缓存中取数据，取到直接返回结果，取不到时从数据库中取，数据库取到更新缓存，并返回结果，数据库也没取到，那直接返回空结果。

二、缓存穿透

描述：

缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；

从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击

三、缓存击穿

描述：

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力

解决方案：

设置热点数据永远不过期。

加互斥锁，互斥锁参考代码如下：

说明：

1）缓存中有数据，直接走上述代码13行后就返回结果了

2）缓存中没有数据，第1个进入的线程，获取锁并从数据库去取数据，没释放锁之前，其他并行进入的线程会等待100ms，再重新去缓存取数据。这样就防止都去数据库重复取数据，重复往缓存中更新数据情况出现。

3）当然这是简化处理，理论上如果能根据key值加锁就更好了，就是线程A从数据库取key1的数据并不妨碍线程B取key2的数据，上面代码明显做不到这点。

四、缓存雪崩

描述：

缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是， 缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。

如果缓存数据库是分布式部署，将热点数据均匀分布在不同搞得缓存数据库中。

设置热点数据永远不过期。

# 11用程序思维讲解工作流。

1.部署流程的定义

2.启动流程的实例

3.查看我的任务列表

4.办理任务

5.查看流程（正在执行，还是已经结束）

6.查询历史任务

7.查询历史的流程实例

# 12list扩容。

ArrayList 的内部实现，其实是用一个对象数组进行存放具体的值，然后用一种扩容的机制，进行数组的动态增长。

其扩容机制可以理解为，如果元素的个数，大于其容量，则把其容量扩展为原来容量的1.5倍。调用add方法。

默认初始化构造出来的list，容量0，大小0。往其中添加一个元素，容量10，大小1。往其中添加11个元素。容量15，大小11。

ArrayList 几个重要属性之一

1，首先是默认初始值的大小：

private static final int DEFAULT\_CAPACITY = 10;,

2，接着是一个默认的空对象数组：

private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};

3，然后是ArrayList 实际数据存储的一个数组：

transient Object[] elementData;

4，elementData 的大小：

private int size;

# 13什么是面向对象？

面向过程和面向对象的优缺点：

面向过程

优点：性能上它是优于面向对象的，因为类在调用的时候需要实例化，开销过大。

缺点：不易维护、复用、扩展

用途：单片机、嵌入式开发、Linux/Unix等对性能要求较高的地方

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

缺点：性能比面向过程低

低耦合，简单的理解就是说，模块与模块之间尽可能的独立，两者之间的关系尽可能简单，尽量使其独立的完成成一些子功能，这避免了牵一发而动全身的问题。

# 14架构的主要考虑因素是什么？

开发架构，运行架构，存储架构，运维架构。

# 15，请解释以下名词，同步，异步，阻塞，非阻塞。

同步：执行一个操作之后，等待结果，然后才继续执行后续的操作。

异步：执行一个操作后，可以去执行其他的操作，然后等待通知再回来执行刚才没执行完的操作。

阻塞：进程给CPU传达一个任务之后，一直等待CPU处理完成，然后才执行后面的操作。

非阻塞：进程给CPU传达任务后，继续处理后续的操作，隔断时间再来询问之前的操作是否完成。这样的过程其实也叫轮询。

# 16设计模式，你在开发中用过什么？怎么用的？

日常使用比较多的，那就是单例模式。例如数据库连接池这些都是单例的。还有工厂模式。不管是mybatis还是hibernate都用到了很多工厂模式，我们使用最常见的sql session factory。

# 17设计模式之开闭原则。

对扩展开放，对修改关闭。

# 18。面向对象三大特征之封装如何符合这个特征？

. 封装：只对外界提供有用的属性和行为 外界不用关心类具体的实现过程

1、方法、继承都是封装的一种体现，private（私有的）也是封装的一个体现。被private修饰的成员变量只能在本类中使用，外部不能直接通过对象属性改变。获取和改变需要提供设置器（访问接口|访问方法）和访问器来进行操作。

特点：

1、可以隐藏内部细节，对外提供公共访问方式。

​ 2、提高安全性，提高代码的复用性。

2、这种多个封装的模板类统称之为javabean，模板类需要满足的一些规则

​ 一、类是公共的

​ 二、私有的属性

​ 三、提供公有的访问方式 set、get

​ 四、最好定义一个空构造

# 19。Spring源码走读。

# 20如何保证你写的bug少？

遵循总结的设计模式七大原则

良好的代码习惯，例如注释，例如单元测试

我写的bug本来就比别人少

# 21 io nio aio

Io分为网络io和磁盘io,是同步阻塞的。io面向流的。而nio和aio是面向缓冲区的。

Nio是同步的。aio是异步的。

# 22项目上高并发分布式缓存，如何使用？

一般我们都是三级缓存配置。一级本地缓存。与应用同生共死，常用有Ehcache。还包括浏览器存储的数据。二级缓存采用分布式缓存。我们常用的轻量级框架有redis。三级就是数据库集群了。

# 23Tomcat。Catalina。和start。文件的区别是什么？

tomcat 的真正启动是在 catalina.bat 设置并启动的。startup.bat 只是找到catalina.bat 然后执行catalina.bat 来启动tomat的。catalina除了启动配置之外，还有其他配置，比如关闭。调试debug,还有不同的启动tomcat的方法，比如run和start ，他们的区别是，是否在新窗口启动tomcat。

run是在本窗口中执行。

start是新开一个窗口执行。

# 24基础jvm。

java虚拟机的内存结构分为堆(heap)和栈(stack),堆里面存放是对象实例也就是new出来的对象。栈里面存放的是基本数据类型以及引用数据类型的地址。

对于所谓的常量是存储在方法区的常量池里面。

# 25hash map

1.Map是一个以键值对存储的接口。Map下有两个具体的实现，分别是HashMap和HashTable.

2.HashMap是线程非安全的，HashTable是线程安全的，所以HashMap的效率高于HashTable.

3.HashMap允许键或值为空，而HashTable不允许键或值为空.

# 26线程池

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。

假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

一个线程池包括以下四个基本组成部分：

1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；

广告

c++多线程编程【C语言+GO语言+区块链】

查看详情

js.tedu.cn

2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；

3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；

4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。

线程池不仅调整T1,T3产生的时间段，而且它还显著减少了创建线程的数目，看一个例子：

假设一个服务器一天要处理50000个请求，并且每个请求需要一个单独的线程完成。在线程池中，线程数一般是固定的，所以产生线程总数不会超过线程池中线程的数目，而如果服务器不利用线程池来处理这些请求则线程总数为50000。一般线程池大小是远小于50000。所以利用线程池的服务器程序不会为了创建50000而在处理请求时浪费时间，从而提高效率。

java线程池的工作原理和数据库连接池的差不多，因为每次重新创建线程

都是很耗资源的操作，所以我们可以建立一个线程池，这样当需要用到线程

进行某些操作时，就可以直接去线程池里面找到空闲的线程，这样就可以直接

使用，而不用等到用到的时候再去创建，用完之后可以把该线程重新放入线程池

供其他请求使用从而提高应用程序的性能。

线程池的核心流程:

1.构建一个 ThreadPoolExecutor 并指定默认要创建的线程的数量

2.通过 threadPool.execute()

去添加一个个要执行的线程即实现了Runable接口的java类

3.在实现了Runable接口的java类的run方法中写入具体的业务代码

线程池的业务场景：

我在工作的时候，当时一个同事给我提了一个需求，目前有大量的图片

需要处理生产缩略图并进行加水印，因为按照普通的处理方法一个个的

进行处理太慢了，问我有没有好的解决方案，这个时候我就想到了java中

的线程池，我构建了一个线程数为5个线程池，然后采用分段批量提取的

方式每500条为一组数据进行图片信息的提取，然后再把这些通过Threadpool的

execute方法交给线程池中的线程进行处理，即充分使用了CPU硬件资源又加快

了大数据情况下程序的处理效率。

我当时在工作的过程中，认识一个做电商的朋友，他们当时公司才起步，很多

技术都不成熟，所以就常常和我探讨一些技术问题，有次他向我请教一个问题，

问我如何才能提高网站的性能，我根据自己在项目中的经验以及自己以前阅读的

关于优化方面的资料给他提出了很多建议，如用lucene进行全文检索，用memcached

进行分布式缓存，以及通过spring定时器结合freeMarker模板引擎来生成静态

页面，由于要生成的页面的数量比较多，考虑到程序的性能，我建议他结合

java的线程池进行工作，这样就可以充分使用了CPU硬件资源又加快

# 30mybatis源码

# 31spring源码

# 32 谈谈aop ioc ,aop有哪些注解

Spring的概述

Spring 是完全面向接口的设计，降低程序耦合性，主要是事务控制并创建bean实例对象。在ssh整合时，充当黏合剂的作用。IOC(Inversion of Control) 控制反转/依赖注入，又称DI(Dependency Injection) (依赖注入)

IOC的作用：产生对象实例，所以它是基于工厂设计模式的

Spring IOC的注入

　　 通过属性进行注入，通过构造函数进行注入，

　　 注入对象数组 注入List集合

　　 注入Map集合 注入Properties类型

Spring IOC 自动绑定模式：

可以设置autowire按以下方式进行绑定

按byType只要类型一致会自动寻找，

按byName自动按属性名称进行自动查找匹配.

AOP 面向方面（切面）编程

AOP是OOP的延续，是Aspect Oriented Programming的缩写，

　　　　意思是面向方面(切面)编程。

　　 注：OOP(Object-Oriented Programming ) 面向对象编程

AOP 主要应用于日志记录，性能统计，安全控制,事务处理（项目中使用的）等方面。

Spring中实现AOP技术：

在Spring中可以通过代理模式来实现AOP

代理模式分为

静态代理：一个接口，分别有一个真实实现和一个代理实现。

动态代理：通过代理类的代理，接口和实现类之间可以不直接发生联系，而 可以在运行期（Runtime）实现动态关联。

动态代理有两种实现方式，可以通过jdk的动态代理实现也可以通过cglib

来实现而AOP默认是通过jdk的动态代理来实现的。jdk的动态代理必须要有

接口的支持，而cglib不需要，它是基于类的。

Spring AOP事务的描述：

在spring-common.xml里通过<aop:config>里面先设定一个表达式，设定对service里那些方法 如：对add\* ,delete\*,update\*等开头的

Aop日志控制，加解密，读写分离。

Ioc初始化，bean创建。

# 33在哪一些地方使用了工厂模式。

Ioc mybatis hibernate Sqlsession factory。

# 34.Spring MVC常用注解有哪些？

SpringMVC的常用注解：

@Service用于Service层的注解

@requestMapping 用于请求 url 映射。

@RequestBody 注解实现接收 http 请求的 json 数据，将 json 数据转换为 java 对象。

@ResponseBody 注解实现将 controller 方法返回对象转化为 json 响应给客户

SpringMVC常用注解及其介绍

前言：在介绍SpringMVC常用注解之前，有兴趣的可以先了解一下SpringMVC的工作流程。SpringMVC工作流程详解

OK，接下来进入本次主题。

在实际项目中，我们最常用的几个注解，包括 @Controller、@RestController、 @RequestMapping、@PathVariable、@RequestParam 以及 @RequestBody，此次主要介绍下这几个注解常用的使用方式和特点。

1.@Controller

在SpringMVC中，controller主要负责处理前端控制器(DispatcherServlet )发过来的请求，经过业务逻辑层处理之后封装层一个model，并将其返回给view进行展示。@controller注解通常用于类上，如果结合Thymeleaf模板使用的话，会返回一个页面。如果是前后端分离的项目，则使用@RestController，表明返回的是json格式数据。

2.@RestController

在介绍RestController之前，我们先点进去看一下：

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Controller

@ResponseBody

public @interface RestController {

@AliasFor(

annotation = Controller.class

)

String value() default "";

}

1

2

3

4

5

6

7

8

9

10

11

可以发现，@RestController注解里面包含了@Controller注解和@ResponseBody注解，@ResponseBody 注解是将返回的数据结构转换为 JSON 格式，所以说可以这么理解：@RestController = @Controller + @ResponseBody ，省了很多事，我们使用 @RestController 之后就不需要再使用 @Controller 了。

3.@RequestMapping

@RequestMapping 是一个用来处理请求地址映射的注解，它可以用于类上，也可以用于方法上。用于类上的注解会将一个特定请求或者请求模式映射到一个控制器之上，表示类中的所有响应请求的方法都是以该地址作为父路径；方法的级别上注解表示进一步指定到处理方法的映射关系。

该注解有6个属性，一般在项目中比较常用的有三个属性：value、method 和 produces。

value 属性：指定请求的实际地址，value 可以省略不写；

method 属性：指定请求的类型，主要有 GET、PUT、POST、DELETE，默认为 GET。

produces 属性：指定返回内容类型，如 produces = “application/json; charset=UTF-8”。

@RequestMapping 注解比较简单，举个例子：

@RestController

@RequestMapping(value = "/test", produces = "application/json; charset=UTF-8")

public class TestController {

@RequestMapping(value = "/get", method = RequestMethod.GET)

public String testGet() {

return "success";

}

}

1

2

3

4

5

6

7

8

9

这个很简单，启动项目在浏览器中输入 localhost:8080/test/get 测试一下即可。

四种不同的请求方式，都有相应的注解。不用每次在 @RequestMapping 注解中加 method 属性来指定，上面的 GET 方式请求可以直接使用 @GetMapping("/get") 注解，效果一样。相应地，PUT 方式、POST 方式和 DELETE 方式对应的注解分别为 @PutMapping、@PostMapping 和 DeleteMapping。

4.@PathVariable

@PathVariable 注解主要用来获取 URL 参数，Spring Boot 支持 Restfull 风格的 URL，比如一个 GET 请求携带一个参数 id，我们将 id 作为参数接收，可以使用 @PathVariable 注解。如下：

@GetMapping("/user/{id}")

public String testPathVariable(@PathVariable Integer id) {

System.out.println("获取到的id为：" + id);

return "success";

}

1

2

3

4

5

这里需要注意一个问题，如果想要 URL 中占位符中的 id 值直接赋值到参数 id 中，需要保证 URL 中的参数和方法接收参数一致，否则将无法接收。如果不一致的话，其实也可以解决，需要用 @PathVariable 中的 value 属性来指定对应关系。如下：

@RequestMapping("/user/{idd}")

public String testPathVariable(@PathVariable(value = "idd") Integer id) {

System.out.println("获取到的id为：" + id);

return "success";

}

1

2

3

4

5

对于访问的 URL，占位符的位置可以在任何位置，不一定非要在最后，比如这样也行：/xxx/{id}/user。另外，URL 也支持多个占位符，方法参数使用同样数量的参数来接收，原理和一个参数是一样的，例如：

@GetMapping("/user/{idd}/{name}")

public String testPathVariable(@PathVariable(value = "idd") Integer id, @PathVariable String name) {

System.out.println("获取到的id为：" + id);

System.out.println("获取到的name为：" + name);

return "success";

}

1

2

3

4

5

6

运行项目，在浏览器中请求：localhost:8080/test/user/2/zhangsan， 可以看到控制台输出如下信息：

获取到的id为：2

获取到的name为：zhangsan

所以它支持多个参数的接收。同样地，如果 URL 中的参数和方法中的参数名称不同的话，也需要使用 value 属性来绑定两个参数。

5.@RequestParam

@RequestParam 注解顾名思义，也是获取请求参数的，上面我们介绍了 @PathValiable 注解也是获取请求参数的，那么 @RequestParam 和 @PathVariable 有什么不同呢？

@PathValiable 是从 URL 模板中获取参数值， 即这种风格的 URL：

http://localhost:8080/user/{id}

1

@RequestParam 是从 Request 里获取参数值，即这种风格的 URL：

http://localhost:8080/user?id=1

1

对于@RequestParam 注解代码测试如下：

@GetMapping("/user") //注意这里请求路径的写法是不一样的

public String testRequestParam(@RequestParam Integer id) {

System.out.println("获取到的id为：" + id);

return "success";

}

1

2

3

4

5

同样的@RequestParam 注解的value 属性是比较常用的，其作用和@PathVariable注解的value属性是一样的。此外@RequestParam 注解还有两个属性比较常用：

required 属性：true 表示该参数必须要传，否则就会报 404 错误，false 表示可有可无。

defaultValue 属性：默认值，表示请求中没有同名参数时的默认值。

从 URL 中可以看出，@RequestParam 注解用于 GET 请求上时，接收拼接在 URL 中的参数。除此之外，该注解还可以用于 POST 请求，接收前端表单提交的参数，假如前端通过表单提交 username 和 password 两个参数，那我们可以使用 @RequestParam 来接收，用法和上面一样。

6.@RequestBody

RequestBody 注解用于接收前端传来的实体，接收参数也是对应的实体，比如前端通过 JSON 提交传来两个参数 username 和 password，此时我们需要在后端封装一个实体来接收。在传递的参数比较多的情况下，使用 @RequestBody 接收会非常方便。例如：

定义User类代码此处省略…

@PostMapping("/user")

public String testRequestBody(@RequestBody User user) {

System.out.println("获取到的username为：" + user.getUsername());

System.out.println("获取到的password为：" + user.getPassword());

return "success";

}

1

2

3

4

5

6

可以看出，@RequestBody 注解用于 POST 请求上，接收 JSON 实体参数。它和上面我们介绍的表单提交有点类似，只不过参数的格式不同，一个是 JSON 实体，一个是表单提交。在实际项目中根据具体场景和需要使用对应的注解即可。

# 35-Rest Controller，它是一个综合注解，它包含什么？

@RestController注解里面包含了@Controller注解和@ResponseBody注解，@ResponseBody 注解是将返回的数据结构转换为 JSON 格式，所以说可以这么理解：@RestController = @Controller + @ResponseBody ，省了很多事，我们使用 @RestController 之后就不需要再使用 @Controller 了。

# 36sql优化。

分

,建表,存储类型，索引类型，查询缓存,分库分表,使用中间件查

询缓存。

1.(重点)(必须说) SELECT语句中避免使用 \*，

尽量应该根据业务需求按字段进行查询

举例：如果表中有个字段用的是clob或者是blob这种大数据字段的话，

他们的查询应该根据业务需要来进行指定字段的查询，切记勿直接用\*

2.(重点) 删除重复记录(oracle)：

最高效的删除重复记录方法 ( 因为使用了ROWID)例子：

DELETE FROM EMP E WHERE E.ROWID > (SELECT MIN(X.ROWID)

FROM EMP X WHERE X.EMP\_NO = E.EMP\_NO);

3. 用>=替换>

如一个表有100万记录，一个数值型字段A，

A=0时，有30万条；

A=1时，有30万条；

A=2时，有39万条；

A=3时，有1万记录。

那么执行 A>2 与 A>=3 的效果就有很大的区别了，因为 A>2 时，

ORACLE会先找出为2的记录索引再进行比较，

而A>=3时ORACLE则直接找到=3的记录索引。

4.(重点)尽量多使用COMMIT

如对大数据量的分段批量提交

5. (重点)用NOT EXISTS 或（外连接+判断为空）方案 替换 NOT IN操作符

此操作是强列推荐不使用的，因为它不能应用表的索引。

推荐方案：用NOT EXISTS 或（外连接+判断为空）方案代替

6.(重点 必须说)LIKE操作符(大数据的全文检索使用luncene)(solr)

因为使用like不当，会导致性能问题，原因是like在左右两边都有

%的时候，不会使用索引。

如LIKE '%5400%' 这种查询不会引用索引，

而LIKE 'X5400%' 则会引用范围索引。

一个实际例子：

查询营业编号 YY\_BH LIKE '%5400%' 这个条件会产生全表扫描，

如果改成 YY\_BH LIKE 'X5400%' OR YY\_BH LIKE 'B5400%'

则会利用 YY\_BH 的索引进行两个范围的查询，性能肯定大大提

# 37。举个例子，对象被回收的过程。

对象基本上都是在jvm的堆区中创建，在创建对象之前，

会触发类加载（加载、连接、初始化），

当类初始化完成后，

根据类信息在堆中实例化类对象，

初始化非静态变量、非静态代码以及默认构造方法，

当对象使用完之后会在合适的时候被jvm垃圾收集器回收。

可达性分析算法

可达状态：当一个对象被创建后，若有一个以上的引用变量引用它，则这个对象在程序中处于可达状态。

可恢复状态：如果程序中某个对象不再有任何引用变量引用它，它就进入了可恢复状态。此时，系统的垃圾回收机制准备回收该对象所占用的内存，在回收该对象之前，系统会调用所有可恢复状态对象的finalize()方法进行资源清理。如果系统在调用finalize()方法时重新让一个引用变量引用该对象，则这个对象会再次变成可达状态；否则该对象将进入不可达状态。

不可达状态：当对象与所有引用变量的关联都被切断，且系统已经调用所有对象的finalize()方法后依然没有使该对象变成可达状态，那么这个对象将永久性地失去引用，最后变成不可达状态。只有当一个对象处于不可达状态时，系统才会真正回收该对象所占有的资源。

所以finalize方法是对象存活的最后一次机会，而且只会执行一次。可以将可恢复状态转变为可达状态。

# 38。联合索引和两个唯一索引哪个好？不考虑内存使用。

在条件一致的情况下，肯定是联合索引比较好。索引其实就是单独存了一张表，两次索引你就会经过两张表两次io。所以永远有一个最左匹配原则。理论上来说，如果执行顺序不对的话，他是不会走联合索引的。涉及到msql的加载过程。第一步语法解析和预处理。判断你的语法是否符合规范。同时也会检查表中你查询列是否存在？第二部查询优化。

经过前面的步骤生成的语法树被认为是合法的了，并且由优化器将其转化成查询计划，多数情况下，一条查询可以有很多种执行方式，最后都返回相同的结果，

优化器的作用就是找到这其中最好的执行计划。跟jvm有点相似。那边是jvm不保证你的代码执行顺序，但是保证你的执行结果是一致的,也就是说它会偷偷的优化你的代码,仅仅是在非多线程的情况下保证，要是多线程环境就无法保证了

# 39。怎么通过注解加载类的？

# 40。堆的结构。

堆的定义：

这里的堆是指一种数据结构（或数据结构属性），非指堆内存。堆属性用二叉树来体现，具有堆属性的数据结构才可被叫做为堆。堆的特点是任意的父节点都大于或者小于他们的子节点。大于的时候叫做大顶堆，小于的时候叫做小顶堆。

结构特点：

我们一般数组来表示一个堆数据。下标为n的元素在堆中父子节点下标分别是

父节点：（n-1）/2；

左子节点：2\*n+1；

右子节点：2\*n+2；

堆得构建一般按照以下步骤

1. 初始化数字大小为原数据大小

2. 把元素插入数组最后

3. 新插入元素个他的父节点比较，如果大于父节点则彼此交换位置

4.重复步骤3直到所有不存在子节点大于父节点

堆得删除的一般步骤：

1. 把堆的最后一个元素取出

2. 把堆的最后一个元素放到堆顶

3. 比较父节点跟子节点，如果有任意字节大于父节点则彼此交换位置

1. 重复步骤3知道没有子节点大于父节点

# 41.osi七层模型

第一层：物理层

第二层：数据链路层

第三层：网络层

第四层：传输层

第五层：会话层

第六层：表示层

第七层：应用层

# 42.java查看GC日志

自带的jconsole工具、jstat命令

终端或cmd命令行输入jconsole就会出现jdk自带的gui监控工具，可以根据内存使用情况间接了解内存使用和gc情况。

**例如:eclipse.ini中配置下面代码启动后会在同一目录下生成gc.log**

|  |  |
| --- | --- |
| 1  2  3 | -Xloggc:gc.log  -XX:+PrintGCTimeStamps  -XX:+PrintGCDetails |

Jstat -gcutil 端口 liunx总结GC统计

# 43.用最有效的方式计算2乘以8

因为将一个数左移n 位，就相当于乘以了2 的n 次方，那么，一个数乘以8 只要将其左移3 位  
即可，而位运算cpu 直接支持的，效率最高，所以，2 乘以8 等於几的最效率的方法是2 << 3

# 44.TCP/IP协议的三次握手四次挥手

第一次握手：建立连接时,客户端发送syn包(syn=j)到服务器,并进入SYN\_SEND状态,等待服务器确认；   
SYN：同步序列编号(Synchronize Sequence Numbers)  
第二次握手：服务器收到syn包,必须确认客户的SYN（ack=j+1）,同时自己也发送一个SYN包（syn=k）,即SYN+ACK包,此时服务器进入[SYN\_RECV](https://www.baidu.com/s?wd=SYN_RECV&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLmHn4mhfLnWIBuHfzPjc10ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjn4PHTsPj0s" \t "https://www.cnblogs.com/Qing-840/p/_blank)状态；   
第三次握手：客户端收到服务器的SYN＋ACK包,向服务器发送确认包ACK(ack=k+1),此包发送完毕,客户端和服务器进入ESTABLISHED状态,完成三次握手.

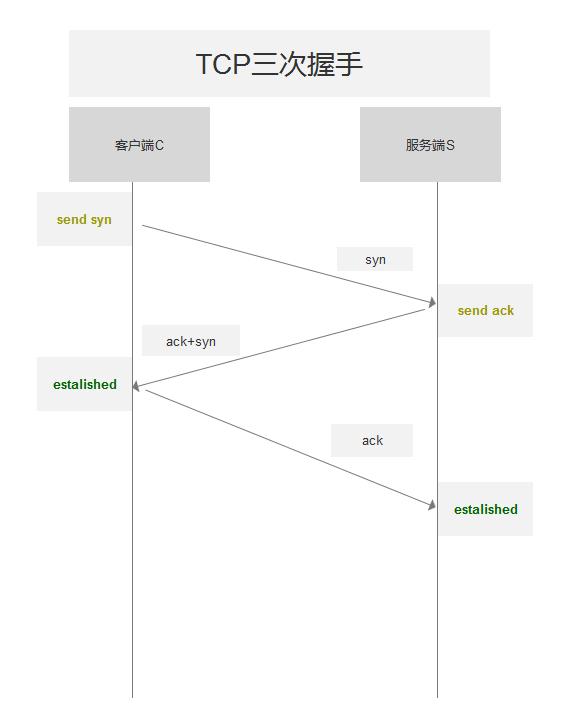
为什么建立连接需要三次握手？

首先非常明确的是两次握手是最基本的。第一次握手，客户端发了个连接请求消息到服务端，服务端收到信息后知道自己与客户端是可以连接成功的，但此时客户端并不知道服务端是否已经接收到了它的请求，所以服务端接收到消息后的应答，客户端得到服务端的反馈后，才确定自己与服务端是可以连接上的，这就是第二次握手。

客户端只有确定了自己能与服务端连接上才能开始发数据。所以两次握手肯定是最基本的。

看到这里，你或许会问，那么为什么需要第三次握手呢？我们来看一下，假设一下如果没有第三次握手，而是两次握手后我们就认为连接成功了，那么会发生什么？第三次握手是为了防止已经失效的连接请求报文段突然又传到服务端，因而产生错误。

譬如发起请求遇到类似这样的情况：客户端发出去的第一个连接请求由于某些原因在网络节点中滞留了导致延迟，直到连接释放的某个时间点才到达服务端，这是一个早已失效的报文，但是此时服务端仍然认为这是客户端的建立连接请求第一次握手，于是服务端回应了客户端，第二次握手。



如果只有两次握手，那么到这里，连接就建立了，但是此时客户端并没有任何数据要发送，而服务端还在傻傻的等候佳音，造成很大的资源浪费。所以需要第三次握手，只有客户端再次回应一下，就可以避免这种情况。

四次挥手

与建立连接的“三次握手”类似，断开一个TCP连接则需要“四次握手”。

第一次挥手：主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭方告诉被动关闭方：我已经不 会再给你发数据了(当然，在fin包之前发送出去的数据，如果没有收到对应的ack确认报文，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据。

第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号）。

第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了。

第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手。

————————————————

版权声明：本文为CSDN博主「ronety」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/ronety/article/details/89140053

# dubbo参数配置

dubbo-server.xml文件

## **一、多版本的支持**

如何发布服务，需要将需要暴露的服务接口发布出去供客户端调用，需要在java同级目录新建一个resources目录，然后将resoureces目录标记成Test Resoureces Root，然后在esources目录下新建MATE-INF.spring目录，在该目录下添加配置文件dubbo-server.xml文件

  dubbo的服务端配置文件如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"   xmlns:dubbo="http://code.alibabatech.com/schema/dubbo" xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springframework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp://code.alibabatech.com/schema/dubbo/dubbo.xsd">  <!--提供方信息，用于计算依赖关系-->  <dubbo:application name="dubbo-server" owner="mic"/>    <!--注册中心 暴露服务地址-->  <dubbo:registry address="zookeeper://192.168.126.129:2181"/>    <!--用dubbo协议在20880 端口暴露服务-->  <dubbo:protocol port="20880" name="dubbo"/>    <!--声明需要暴露的服务接口，指定协议为dubbo，设置版本号1.1.1-->  <dubbo:service interface="com.gupaoedu.dubbo.IGpHello"  ref="gpHelloService" protocol="dubbo" version="1.1.1"/>  <!--声明需要暴露的服务接口，指定协议为dubbo，设置版本号1.1.2-->  <dubbo:service interface="com.gupaoedu.dubbo.IDemoService"    ref="demoService" protocol="dubbo" version="1.1.2"/>    <!--和本地服务一样实现服务-->  <bean id="gpHelloService" class="com.gupaoedu.dubbo.GpHelloImpl"/>    <bean id="demoService" class="com.gupaoedu.dubbo.DemoService"/>  </beans> |

服务端的接口以及实现类

|  |  |
| --- | --- |
| 1 | public interface IGpHello {String sayHello(String msg);} |

|  |  |
| --- | --- |
| 1 | public interface IDemoService {String protocolDemo(String msg);} |

|  |  |
| --- | --- |
| 1  2  3  4 | public class GpHelloImpl implements IGpHello{  @Override  public String sayHello(String msg) {return "Hello:"+msg;}  } |

|  |  |
| --- | --- |
| 1  2  3  4 | public class GpHelloImpl2 implements IGpHello{  @Override  public String sayHello(String msg) {return "Hello,i'm server 2:"+msg;}  } |

编写Main方法，用spring容器来启动服务

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class Main {  public static void main(String[] args) throws IOException {  //默认情况下会使用spring容器来启动服务  com.alibaba.dubbo.container.Main.main(new String[]{"spring","log4j"});}  } |

服务启动过程会进行服务注册，启动监听端口，启动服务之后

在客户端通过远程调用访问服务端发布的服务，相应的客户端配置文件 dubbo-client.xml 如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springframework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp://code.alibabatech.com/schema/dubbo/dubbo.xsd">  <!--提供方信息-->  <dubbo:application name="dubbo-client" owner="mic"/>    <!--注册中心-->  <dubbo:registry id="zokeeper" address="zookeeper://192.168.126.129:2181?register=false" file="d:/dubbo-server"/>    <!--声明需要暴露的服务接口，指定版本号-->  <dubbo:reference id="gpHelloService" interface="com.gupaoedu.dubbo.IGpHello" registry="zookeeper" version="1.1.1"/>  </beans> |

其实我们可以在zookeeper的客户端可以发现，事实上已经发布了服务方已经发布了两个不同版本的服务，具体如下

1 dubbo%3A%2F%2F192.168.126.1%3A20880%2Fcom.gupaoedu.dubbo.IDemoService%3Fanyhost%3Dtrue%26application%3Ddubbo-server%26dubbo%3D2.5.3%26interface%3Dcom.gupaoedu.dubbo.IDemoService%26methods%3DprotocolDemo%26owner%3Dmic%26pid%3D22548%26revision%3D1.1.2%26side%3Dprovider%26timestamp%3D1530450331827%26version%3D1.1.22 dubbo%3A%2F%2F192.168.126.1%3A20880%2Fcom.gupaoedu.dubbo.IGpHello%3Fanyhost%3Dtrue%26application%3Ddubbo-server%26dubbo%3D2.5.3%26interface%3Dcom.gupaoedu.dubbo.IGpHello%26methods%3DsayHello%26owner%3Dmic%26pid%3D22548%26revision%3D1.1.1%26side%3Dprovider%26timestamp%3D1530450325703%26version%3D1.1.1

我们知道，这两个版本正是我们在服务发布方设置的不同版本号，同样的，在消费端，我们可以通过设置指定的版本号获取相应的版本服务，消费的代码如下



public class App{

public static void main( String[] args ) throws IOException, InterruptedException {

ClassPathXmlApplicationContext context=new ClassPathXmlApplicationContext("dubbo-client.xml")；

context.start();

IGpHello demoService = (IGpHello)context.getBean("gpHelloService");//获取远程服务代理

String hello = demoService.sayHello("world");//执行远程调用方法

System.out.println(hello);//显示调用结果 }

}



在控制台，我们可以看到同样的服务发布地址url。

[回到顶部](https://www.cnblogs.com/yxh1008/p/9251693.html" \l "_labelTop)

## ****二、主机绑定****

  什么叫主机绑定呢？主机绑定的流程是什么？

在发布一个dubbo服务的时候，会生成一个dubbo://ip:port的协议地址。这就是主机绑定过程，那么这个IP是如何生成的呢？大家可以通过ServiceConfig.java 中的 doExportUrlForlProtocol 方法中找到如下代码块。大致的逻辑是，首先从配置文件中获取，如果失败再尝试从本地网卡中获取host，如果这个也失败，会继续执行，直到找到合适的IP地址。



String name = protocolConfig.getName();if (name == null || name.length() == 0) {

name = "dubbo";

}

String host = protocolConfig.getHost();//从配置文件中获取hostif (provider != null && (host == null || host.length() == 0)) {

host = provider.getHost();

}

boolean anyhost = false;if (NetUtils.isInvalidLocalHost(host)) {

anyhost = true;

try {

host = InetAddress.getLocalHost().getHostAddress();//获取本机的host地址

} catch (UnknownHostException e) {

logger.warn(e.getMessage(), e);

}

if (NetUtils.isInvalidLocalHost(host)) {

if (registryURLs != null && registryURLs.size() > 0) {//如果还是没有获取到host地址

for (URL registryURL : registryURLs) {

try {

Socket socket = new Socket();

try {

SocketAddress addr = new InetSocketAddress(registryURL.getHost(), registryURL.getPort());

socket.connect(addr, 1000);

host = socket.getLocalAddress().getHostAddress();//3、

break;

} finally {

try {

socket.close();

} catch (Throwable e) {}

}

} catch (Exception e) {

logger.warn(e.getMessage(), e);

}

}

}

if (NetUtils.isInvalidLocalHost(host)) {//4、

host = NetUtils.getLocalHost();

}

}

}



[回到顶部](https://www.cnblogs.com/yxh1008/p/9251693.html" \l "_labelTop)

# 45.Redis五种数据结构原理分析

字符串String

列表List

哈希hash

集合Set

有序集合zset

关于Redis

redis是一个开源的使用C语言编写的一个kv存储系统，是一个速度非常快的非关系远程内存数据库。它支持包括String、List、Set、Zset、hash五种数据结构。除此之外，通过复制、持久化和客户端分片等特性，用户可以很方便地将redis扩展成一个能够包含数百GB数据和每秒处理上百万次的请求的系统。目前支持多种语言的api，方便用户使用。

redis同时也内置了事务、LUA脚本、复制等功能，提供两种持久化选项，一种是每隔一段时间将数据导入到磁盘(快照模式)，另一种是追加命令到日志中(AOF模式)。如果只是作为高效的内存数据库使用也可以关闭持久化功能。通过哨兵(sentinel)和自动分区(Cuuster)的方式可以提高redis服务器的高可用性。

与关系型数据库相比，redis的命令请求不需要经过查询分析器或查询优化器进行处理，也避免了更新数据时引起的随机读\写，这些慢操作。它直接读写内存中的数据，并且数据是按照一定的数据结构存储的。所以它的速度非常快。

五种数据结构

字符串(String)

       与其它编程语言或其它键值存储提供的字符串非常相似，键(key)------值(value) (字符串格式),字符串拥有一些操作命令，如：get set del 还有一些比如自增或自减操作等等。redis是使用C语言开发，但C中并没有字符串类型，只能使用指针或符数组的形式表示一个字符串，所以redis设计了一种简单动态字符串(SDS[Simple Dynamic String])作为底实现：

定义SDS对象，此对象中包含三个属性：

len buf中已经占有的长度(表示此字符串的实际长度)

free buf中未使用的缓冲区长度

buf[] 实际保存字符串数据的地方

所以取字符串的长度的时间复杂度为O(1)，另，buf[]中依然采用了C语言的以\0结尾可以直接使用C语言的部分标准C字符串库函数。

空间分配原则：当len小于IMB（1024\*1024）时增加字符串分配空间大小为原来的2倍，当len大于等于1M时每次分配 额外多分配1M的空间。

由此可以得出以下特性：

redis为字符分配空间的次数是小于等于字符串的长度N，而原C语言中的分配原则必为N。降低了分配次数提高了追加速度，代价就是多占用一些内存空间，且这些空间不会自动释放。

二进制安全的

高效的计算字符串长度(时间复杂度为O(1))

高效的追加字符串操作。

列表(List)

         redis对键表的结构支持使得它在键值存储的世界中独树一帜，一个列表结构可以有序地存储多个字符串，拥有例如：lpush lpop rpush rpop等等操作命令。在3.2版本之前，列表是使用ziplist和linkedlist实现的，在这些老版本中，当列表对象同时满足以下两个条件时，列表对象使用ziplist编码：

列表对象保存的所有字符串元素的长度都小于64字节

列表对象保存的元素数量小于512个

当有任一条件 不满足时将会进行一次转码，使用linkedlist。

而在3.2版本之后，重新引入了一个quicklist的数据结构，列表的底层都是由quicklist实现的，它结合了ziplist和linkedlist的优点。按照原文的解释这种数据结构是【A doubly linked list of ziplists】意思就是一个由ziplist组成的双向链表。那么这两种数据结构怎么样结合的呢？

ziplist的结构

         由表头和N个entry节点和压缩列表尾部标识符zlend组成的一个连续的内存块。然后通过一系列的编码规则，提高内存的利用率，主要用于存储整数和比较短的字符串。可以看出在插入和删除元素的时候，都需要对内存进行一次扩展或缩减，还要进行部分数据的移动操作，这样会造成更新效率低下的情况。

这篇文章对ziplist的结构讲的还是比较详细的：

https://blog.csdn.net/yellowriver007/article/details/79021049

linkedlist的结构

        意思为一个双向链表，和普通的链表定义相同，每个entry包含向前向后的指针，当插入或删除元素的时候，只需要对此元素前后指针操作即可。所以插入和删除效率很高。但查询的效率却是O(n)[n为元素的个数]。

了解了上面的这两种数据结构，我们再来看看上面说的“ziplist组成的双向链表”是什么意思？实际上，它整体宏观上就是一个链表结构，只不过每个节点都是以压缩列表ziplist的结构保存着数据，而每个ziplist又可以包含多个entry。也可以说一个quicklist节点保存的是一片数据，而不是一个数据。总结：

整体上quicklist就是一个双向链表结构，和普通的链表操作一样，插入删除效率很高，但查询的效率却是O(n)。不过，这样的链表访问两端的元素的时间复杂度却是O(1)。所以，对list的操作多数都是poll和push。

每个quicklist节点就是一个ziplist，具备压缩列表的特性。

在redis.conf配置文件中，有两个参数可以优化列表：

list-max-ziplist-size 表示每个quicklistNode的字节大小。默认为-2 表示8KB

list-compress-depth 表示quicklistNode节点是否要压缩。默认是0 表示不压缩

哈希(hash)

        redis的散列可以存储多个键 值 对之间的映射，散列存储的值既可以是字符串又可以是数字值，并且用户同样可以对散列存储的数字值执行自增操作或者自减操作。散列可以看作是一个文档或关系数据库里的一行。hash底层的数据结构实现有两种：

一种是ziplist，上面已经提到过。当存储的数据超过配置的阀值时就是转用hashtable的结构。这种转换比较消耗性能，所以应该尽量避免这种转换操作。同时满足以下两个条件时才会使用这种结构：

当键的个数小于hash-max-ziplist-entries（默认512）

当所有值都小于hash-max-ziplist-value（默认64）

另一种就是hashtable。这种结构的时间复杂度为O(1)，但是会消耗比较多的内存空间。

集合(Set)

         redis的集合和列表都可以存储多个字符串，它们之间的不同在于，列表可以存储多个相同的字符串，而集合则通过使用散列表（hashtable）来保证自已存储的每个字符串都是各不相同的(这些散列表只有键，但没有与键相关联的值)，redis中的集合是无序的。还可能存在另一种集合，那就是intset，它是用于存储整数的有序集合，里面存放同一类型的整数。共有三种整数：int16\_t、int32\_t、int64\_t。查找的时间复杂度为O(logN)，但是插入的时候，有可能会涉及到升级（比如：原来是int16\_t的集合，当插入int32\_t的整数的时候就会为每个元素升级为int32\_t）这时候会对内存重新分配，所以此时的时间复杂度就是O(N)级别的了。注意：intset只支持升级不支持降级操作。

intset在redis.conf中也有一个配置参数set-max-intset-entries默认值为512。表示如果entry的个数小于此值，则可以编码成REDIS\_ENCODING\_INTSET类型存储，节约内存。否则采用dict的形式存储。

有序集合(zset)

        有序集合和散列一样，都用于存储键值对：有序集合的键被称为成员（member),每个成员都是各不相同的。有序集合的值则被称为分值（score），分值必须为浮点数。有序集合是redis里面唯一一个既可以根据成员访问元素(这一点和散列一样),又可以根据分值以及分值的排列顺序访问元素的结构。它的存储方式也有两种：

是ziplist结构。

          与上面的hash中的ziplist类似，member和score顺序存放并按score的顺序排列

另一种是skiplist与dict的结合。

         skiplist是一种跳跃表结构，用于有序集合中快速查找，大多数情况下它的效率与平衡树差不多，但比平衡树实现简单。redis的作者对普通的跳跃表进行了修改，包括添加span\tail\backward指针、score的值可重复这些设计，从而实现排序功能和反向遍历的功能。

一般跳跃表的实现，主要包含以下几个部分：

表头（head）：指向头节点

表尾（tail）：指向尾节点

节点（node）：实际保存的元素节点，每个节点可以有多层，层数是在创建此节点的时候随机生成的一个数值，而且每一层都是一个指向后面某个节点的指针。

层（level）：目前表内节点的最大层数

长度（length）：节点的数量。

跳跃表的遍历总是从高层开始，然后随着元素值范围的缩小，慢慢降低到低层。

跳跃表的实现原理可以参考：https://blog.csdn.net/Acceptedxukai/article/details/17333673

前面也说了，有序列表是使用skiplist和dict结合实现的，skiplist用来保障有序性和访问查找性能，dict就用来存储元素信息，并且dict的访问时间复杂度为O(1)。

应用场景

redis一般应用场景

缓存会话（单点登录）

分布式锁，比如：使用setnx

各种排行榜或计数器

商品列表或用户基础数据列表等

使用list作为消息对列

秒杀，库存扣减等

五种类型的应用场景

String，redis对于KV的操作效率很高，可以直接用作计数器。例如，统计在线人数等等，另外string类型是二进制存储安全的，所以也可以使用它来存储图片，甚至是视频等。

hash，存放键值对，一般可以用来存某个对象的基本属性信息，例如，用户信息，商品信息等，另外，由于hash的大小在小于配置的大小的时候使用的是ziplist结构，比较节约内存，所以针对大量的数据存储可以考虑使用hash来分段存储来达到压缩数据量，节约内存的目的，例如，对于大批量的商品对应的图片地址名称。比如：商品编码固定是10位，可以选取前7位做为hash的key,后三位作为field，图片地址作为value。这样每个hash表都不超过999个，只要把redis.conf中的hash-max-ziplist-entries改为1024，即可。

list，列表类型，可以用于实现消息队列，也可以使用它提供的range命令，做分页查询功能。

set，集合，整数的有序列表可以直接使用set。可以用作某些去重功能，例如用户名不能重复等，另外，还可以对集合进行交集，并集操作，来查找某些元素的共同点

zset，有序集合，可以使用范围查找，排行榜功能或者topN功能。

总结

       本章介绍了redis的五种数据结构和它们使用的底层存储原理，为了达到节省内存和快速访问的目的每种数据结构可能有两种存储和访问结构，在必要的时候会由一种结构转换成另一种结构，但这个转换的过程会消耗系统性能和内存空间的，所以在使用的过程中需要注意这些配置参数，开发中尽量避免达到这些峰值，使得redis能够持续的提供高效的服务。

————————————————

版权声明：本文为CSDN博主「至臻于勤-大道至简」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/xpsallwell/article/details/84030285>

## **三、集群容错**

在分析dubbo的集群容错前，先了解什么是容错机制？容错机制指的是某种系统控制在一 定范围内的一种允许或包容犯错情况的发生，举个简单例 子，我们在电脑上运行一个程序，有时候会出现无响应的情况，然后系统会弹出一个提示框让我们选择，是立即结 束还是继续等待，然后根据我们的选择执行对应的操作， 这就是“容错”。

       在分布式架构下，网络、硬件、应用都可能发生故障，由 于各个服务之间可能存在依赖关系，如果一条链路中的其 中一个节点出现故障，将会导致雪崩效应。为了减少某一 个节点故障的影响范围，所以我们才需要去构建容错服务（这里要说明一点，容错机制仅仅是处理节点故障的一种机制）， 来优雅的处理这种中断的响应结果 。

Dubbo提供了6种容错机制，分别如下

1. failsafe 失败安全，可以认为是把错误吞掉（记录日 志）

2. failover(默认)   重试其他服务器； retries（2） ，缺省的重试次数，不包含第一次

3. failfast 快速失败， 失败以后立马报错

4. failback  失败后自动恢复。

5. forking  forks. 设置并行数

6. broadcast  广播，任意一台报错，则执行的方法报错

配置方式如下，通过cluster方式，配置指定的容错方案

<!--声明需要暴露的服务接口，指定版本号-->

<dubbo:reference id="gpHelloService" interface="com.gupaoedu.dubbo.IGpHello"

registry="zookeeper" version="1.1.1"

cluster="failover"/>

配置的优先级别

客户端会优于服务端，这里还可以细化，可以细化到方法级别

1. 方法级优先，接口级次之，全局配置再次之。

2. 如果级别一样，则消费方优先，提供方次之

其中，服务提供方配置，通过URL经由注册中心传递给消 费方

什么应该配置在客户端，什么应该配置在服务端，retires、loadBlance、cluster（客户端）、timeout（服务端）

以 timeout 为例，建议由服务提供方设置超时，因为一个方法需要执行多长 时间，服务提供方更清楚，如果一个消费方同时引用多个 服务，就不需要关心每个服务的超时设置。

[回到顶部](https://www.cnblogs.com/yxh1008/p/9251693.html" \l "_labelTop)

## ****四、服务降级****

降级的目的是为了保证核心服务可用。

降级可以有几个层面的分类： 自动降级和人工降级； 按 照功能可以分为：读服务降级和写服务降级；

1. 对一些非核心服务进行人工降级，在大促之前通过降级 开关关闭哪些推荐内容、评价等对主流程没有影响的功 能

2. 故障降级，比如调用的远程服务挂了，网络故障、或者 RPC服务返回异常。 那么可以直接降级，降级的方案比 如设置默认值、采用兜底数据（系统推荐的行为广告挂 了，可以提前准备静态页面做返回）等等

3. 限流降级，在秒杀这种流量比较集中并且流量特别大的 情况下，因为突发访问量特别大可能会导致系统支撑不 了。这个时候可以采用限流来限制访问量。当达到阀值 时，后续的请求被降级，比如进入排队页面，比如跳转 到错误页（活动太火爆，稍后重试等）

dubbo的降级方式： Mock 机制

实现步骤

1. 在client端创建一个TestMock类，实现对应IGpHello 的接口（需要对哪个接口进行 mock，就实现哪个）， 名称必须以Mock结尾

2. 在client端的xml配置文件中，添加如下配置，增加一 个mock属性指向创建的TestMock

3. 模拟错误（设置 timeout），模拟超时异常，运行测试代码即可访问到TestMock这个类。当服务端故障解除 以后，调用过程将恢复正常

服务端的类及实现类，以及配置文件，参考第一点的类及实现类以及配置文件，然后再客户端的配置文件的加上mock，具体见如下截图。

<!--声明需要暴露的服务接口，指定版本号-->

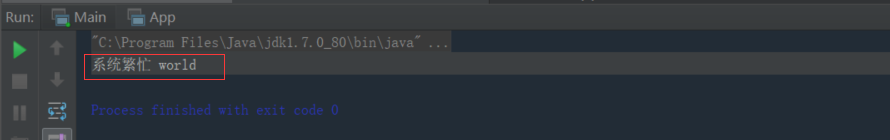
<dubbo:reference id="gpHelloService" interface="com.gupaoedu.dubbo.IGpHello"

registry="zookeeper" version="1.1.1" timeout="1" cluster="failover"

mock="com.gupaoedu.dubbo.TestMock"/>

根据Mock机制实现步骤

首先我们再客户端添加一个TestMock类，实现IGpHello接口,配置文件我已经添加好了，可以看到我这里的配置相对第一点客户端的配置添加了几个参数 timeout、failover、mock。这里主要是看mock和timeout配置，因为我们是要去验证mock机制，而failover是集群容错的配置，之所以我这里没有将这个参数去掉，是因为确定这里他们不会有配置冲突，配置好之后，先发布服务端的服务，然后运气客户端的App.java.在控制台我们可以看到打印结果，系统繁忙，这个结果说明，我们设置1秒超时，客户端远程调用服务超时，报错了，然后就走到了Mock里。



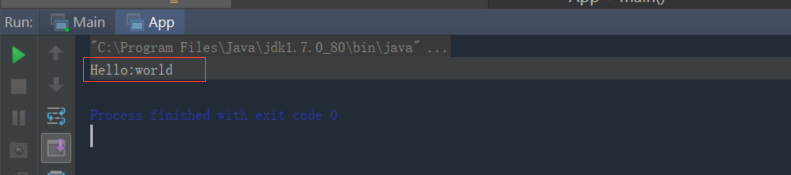
然后我们再验证，将超时间加大，设置为100，再运行，此时就不会报错，就会正常输出Hello world，输出结果如下

<!--声明需要暴露的服务接口，指定版本号-->

<dubbo:reference id="gpHelloService" interface="com.gupaoedu.dubbo.IGpHello"

registry="zookeeper" version="1.1.1" timeout="100" cluster="failover"

mock="com.gupaoedu.dubbo.TestMock"/>



# [Redis与Memcached的区别](http://www.cnblogs.com/wangkai1990/p/5212404.html)

观点一：

1、Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等；

2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储；

3、虚拟内存--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘；

4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10；

5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从；

6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）；

7、灾难恢复--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复；

8、Redis支持数据的备份，即master-slave模式的数据备份；

观点二：

Redis与Memcached的区别

如果简单地比较Redis与Memcached的区别，大多数都会得到以下观点：

1 Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。

2 Redis支持数据的备份，即master-slave模式的数据备份。

3 Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

在Redis中，并不是所有的数据都一直存储在内存中的。这是和Memcached相比一个最大的区别（我个人是这么认为的）。

Redis 只会缓存所有的key的信息，如果Redis发现内存的使用量超过了某一个阀值，将触发swap的操作，Redis根据“swappability = age\*log(size\_in\_memory)”计算出哪些key对应的value需要swap到磁盘。然后再将这些key对应的value持久化到磁 盘中，同时在内存中清除。这种特性使得Redis可以保持超过其机器本身内存大小的数据。当然，机器本身的内存必须要能够保持所有的key，毕竟这些数据 是不会进行swap操作的。

同时由于Redis将内存中的数据swap到磁盘中的时候，提供服务的主线程和进行swap操作的子线程会共享这部分内存，所以如果更新需要swap的数据，Redis将阻塞这个操作，直到子线程完成swap操作后才可以进行修改。

可以参考使用Redis特有内存模型前后的情况对比：

VM off: 300k keys, 4096 bytes values: 1.3G used

VM on: 300k keys, 4096 bytes values: 73M used

VM off: 1 million keys, 256 bytes values: 430.12M used

VM on: 1 million keys, 256 bytes values: 160.09M used

VM on: 1 million keys, values as large as you want, still: 160.09M used

当 从Redis中读取数据的时候，如果读取的key对应的value不在内存中，那么Redis就需要从swap文件中加载相应数据，然后再返回给请求方。 这里就存在一个I/O线程池的问题。在默认的情况下，Redis会出现阻塞，即完成所有的swap文件加载后才会相应。这种策略在客户端的数量较小，进行 批量操作的时候比较合适。但是如果将Redis应用在一个大型的网站应用程序中，这显然是无法满足大并发的情况的。所以Redis运行我们设置I/O线程 池的大小，对需要从swap文件中加载相应数据的读取请求进行并发操作，减少阻塞的时间。

redis、memcache、mongoDB 对比

从以下几个维度，对redis、memcache、mongoDB 做了对比，欢迎拍砖

1、性能

都比较高，性能对我们来说应该都不是瓶颈

总体来讲，TPS方面redis和memcache差不多，要大于mongodb

2、操作的便利性

memcache数据结构单一

redis丰富一些，数据操作方面，redis更好一些，较少的网络IO次数

mongodb支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富

3、内存空间的大小和数据量的大小

redis在2.0版本后增加了自己的VM特性，突破物理内存的限制；可以对key value设置过期时间（类似memcache）

memcache可以修改最大可用内存,采用LRU算法

mongoDB适合大数据量的存储，依赖操作系统VM做内存管理，吃内存也比较厉害，服务不要和别的服务在一起

4、可用性（单点问题）

对于单点问题，

redis，依赖客户端来实现分布式读写；主从复制时，每次从节点重新连接主节点都要依赖整个快照,无增量复制，因性能和效率问题，

所以单点问题比较复杂；不支持自动sharding,需要依赖程序设定一致hash 机制。

一种替代方案是，不用redis本身的复制机制，采用自己做主动复制（多份存储），或者改成增量复制的方式（需要自己实现），一致性问题和性能的权衡

Memcache本身没有数据冗余机制，也没必要；对于故障预防，采用依赖成熟的hash或者环状的算法，解决单点故障引起的抖动问题。

mongoDB支持master-slave,replicaset（内部采用paxos选举算法，自动故障恢复）,auto sharding机制，对客户端屏蔽了故障转移和切分机制。

5、可靠性（持久化）

对于数据持久化和数据恢复，

redis支持（快照、AOF）：依赖快照进行持久化，aof增强了可靠性的同时，对性能有所影响

memcache不支持，通常用在做缓存,提升性能；

MongoDB从1.8版本开始采用binlog方式支持持久化的可靠性

6、数据一致性（事务支持）

Memcache 在并发场景下，用cas保证一致性

redis事务支持比较弱，只能保证事务中的每个操作连续执行

mongoDB不支持事务

7、数据分析

mongoDB内置了数据分析的功能(mapreduce),其他不支持

8、应用场景

redis：数据量较小的更性能操作和运算上

memcache：用于在动态系统中减少数据库负载，提升性能;做缓存，提高性能（适合读多写少，对于数据量比较大，可以采用sharding）

MongoDB:主要解决海量数据的访问效率问题

# 线上问题的定位

每当项目在测试环境跑的时候正常，一旦上线生产，就各种问题出现。jvm问题导致tps降低 ，响应慢。cpu飙高，服务高峰期频繁挂起

其中需要考虑到的方向有很多

1、环境的差异，传统模式下的部署方式大都基于vm，出现环境的差异是不可避免的，如今的技术先进了很多，docker容器化足以支持环境的差异问题，不过这个也是一个比较优先的方案。

因此环境的差异化也是影响代码的要素之一

2、业务访问量差异，一般在测试环境访问量也就公司内部的测试人员在测试，生产上线则是测试环境的N倍数，使用了不恰当的代码，之前偶然协助一个同事定位线上问题的异常，一直定位不到是什么情况出现假死，后来仔细核查，是因为使用了不合理的线程代码，在大量并发的情况下才会出现假死。也是程序猿头疼的因素之一。

处理以上的方案可以从几个维度触发

1、利用分析工具找出性能瓶颈

2、代码内部实现层优化

3、架构层优化

这次的定位主要从分析工具定位，代码层跟架构层的优化需要因地制宜会比较好。

我们当时使用的是继承工具Jstack

A。jstack排查

1.查找进程 top

2.top -Hp 获取占用最高CPU的线程id

3. printf %x 线程id 转为十六进制

4. jstack -F <pid> | grep -a5 线程id（十六进制）

注： 存在账号权限限制

B。使用jmap排查

jmap排查 (需要下载 memory analysis 工具分析)

1、jmap -dump:format=b,file=文件名.bin [pid]

2、启动脚本：-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/equity/$1.bin

注：

1.存在账号权限限制

2.dump存在导致整个服务挂起导致服务不正常的风险

3、dump的文件比较大，存在占用一定的硬盘空间

后来，发现阿里出了一个神器阿尔萨斯arthas 极大的降低了运维排查的繁琐操作

C。arthas排查

1、快速开始 使用arthas-boot（推荐） 下载arthas-boot.jar，然后用java -jar的方式启动： wget https://alibaba.github.io/arthas/arthas-boot.jar java -jar arthas-boot.jar

命令：

a.Dashboard

b.thread -n 3 查看cpu最高的线程

c.jad javax.servlet.Servlet 反编译

d.Trace com.demo.Test doGet 查看慢的跟踪

e. ....

# 49.MySQL常用数据库引擎比较

****ISAM****：ISAM是一个定义明确且历经时间考验的数据表格管理方法，它在设计之时就考虑到数据库被查询的次数要远大于更新的次数。因此，ISAM执行读取操作的速度很快，而且不占用大量的内存和存储资源。ISAM的两个主要不足之处在于，它不支持事务处理，也不能够容错：如果你的硬盘崩溃了，那么数据文件就无法恢复了。如果你正在把ISAM用在关键任务应用程序里，那就必须经常备份你所有的实时数据，通过其复制特性，MYSQL能够支持这样的备份应用程序。

****MyISAM****：MyISAM是MySQL的ISAM扩展格式和缺省的数据库引擎。除了提供ISAM里所没有的索引和字段管理的大量功能，MyISAM还使用一种表格锁定的机制，来优化多个并发的读写操作，其代价是你需要经常运行OPTIMIZE TABLE命令，来恢复被更新机制所浪费的空间。MyISAM还有一些有用的扩展，例如用来修复数据库文件的MyISAMCHK工具和用来恢复浪费空间的 MyISAMPACK工具。MYISAM强调了快速读取操作，这可能就是为什么MySQL受到了WEB开发如此青睐的主要原因：在WEB开发中你所进行的大量数据操作都是读取操作。所以，大多数虚拟主机提供商和INTERNET平台提供商只允许使用MYISAM格式。MyISAM格式的一个重要缺陷就是不能在表损坏后恢复数据。

****InnoDB****：InnoDB数据库引擎都是造就MySQL灵活性的技术的直接产品，这项技术就是MYSQL+API。在使用MYSQL的时候，你所面对的每一个挑战几乎都源于ISAM和MyISAM数据库引擎不支持事务处理（transaction process）也不支持外来键。尽管要比ISAM和 MyISAM引擎慢很多，但是InnoDB包括了对事务处理和外来键的支持，这两点都是前两个引擎所没有的。如前所述，如果你的设计需要这些特性中的一者或者两者，那你就要被迫使用后两个引擎中的一个了。

****MEMORY:****MEMORY是MySQL中一类特殊的存储引擎。它使用存储在内存中的内容来创建表，而且数据全部放在内存中。这些特性与前面的两个很不同。每个基于MEMORY存储引擎的表实际对应一个磁盘文件。该文件的文件名与表名相同，类型为frm类型。该文件中只存储表的结构。而其数据文件，都是存储在内存中，这样有利于数据的快速处理，提高整个表的效率。值得注意的是，服务器需要有足够的内存来维持MEMORY存储引擎的表的使用。如果不需要了，可以释放内存，甚至删除不需要的表。MEMORY默认使用哈希索引。速度比使用B型树索引快。当然如果你想用B型树索引，可以在创建索引时指定。注意，MEMORY用到的很少，因为它是把数据存到内存中，如果内存出现异常就会影响数据。如果重启或者关机，所有数据都会消失。因此，基于MEMORY的表的生命周期很短，一般是一次性的。

在实际工作中，选择一个合适的存储引擎是一个比较复杂的问题。每种存储引擎都有自己的优缺点，不能笼统地说谁比谁好。

****InnoDB****：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据库，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback）。

****MyISAM****：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比较低，也可以使用。

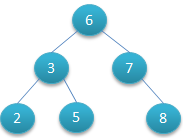
****MEMORY****：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

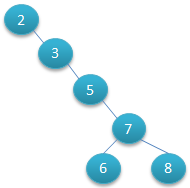
注意，同一个数据库也可以使用多种存储引擎的表。如果一个表要求比较高的事务处理，可以选择InnoDB。这个数据库中可以将查询要求比较高的表选择MyISAM存储。如果该数据库需要一个用于查询的临时表，可以选择MEMORY存储引擎。

# 50.BTree和B+Tree详解

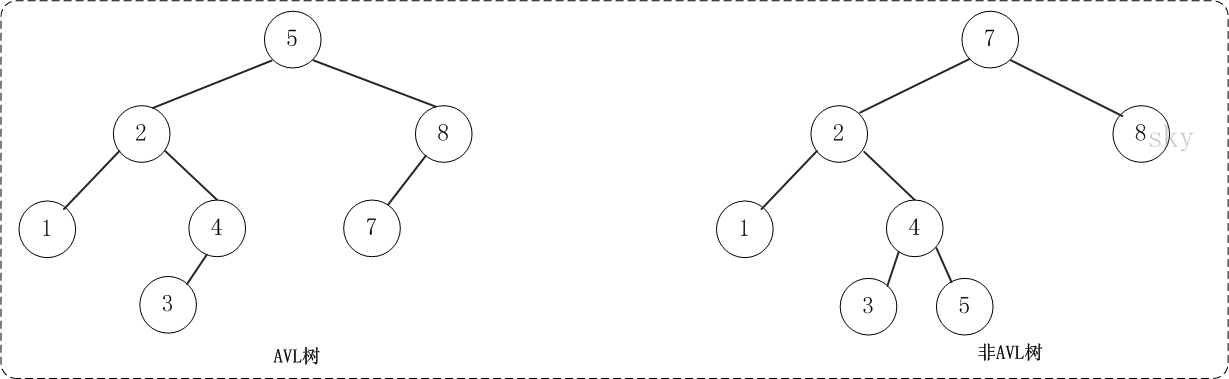
B+树索引是B+树在[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://www.cnblogs.com/vianzhang/p/_blank)中的一种实现，是最常见也是数据库中使用最为频繁的一种索引。B+树中的B代表平衡（balance），而不是二叉（binary），因为B+树是从最早的平衡二叉树演化而来的。在讲B+树之前必须先了解二叉查找树、平衡二叉树（AVLTree）和平衡多路查找树（B-Tree），B+树即由这些树逐步优化而来。

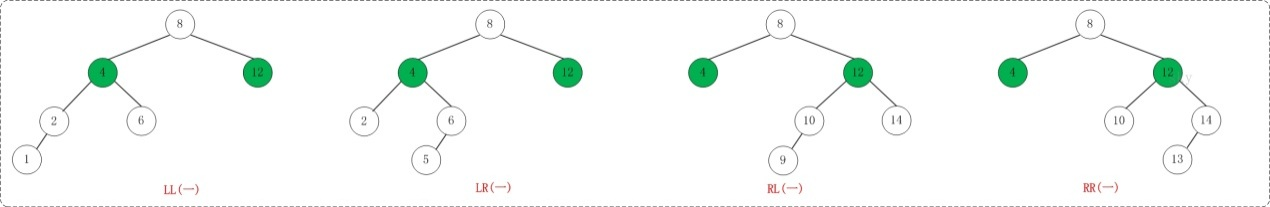
二叉查找树

二叉树具有以下性质：左子树的键值小于根的键值，右子树的键值大于根的键值。   
如下图所示就是一棵二叉查找树，   
   
对该二叉树的节点进行查找发现深度为1的节点的查找次数为1，深度为2的查找次数为2，深度为n的节点的查找次数为n，因此其平均查找次数为 (1+2+2+3+3+3) / 6 = 2.3次

二叉查找树可以任意地构造，同样是2,3,5,6,7,8这六个数字，也可以按照下图的方式来构造：   
   
但是这棵二叉树的查询效率就低了。因此若想二叉树的查询效率尽可能高，需要这棵二叉树是平衡的，从而引出新的定义——平衡二叉树，或称AVL树。

平衡二叉树（AVL Tree）

平衡二叉树（AVL树）在符合二叉查找树的条件下，还满足任何节点的两个子树的高度最大差为1。下面的两张图片，左边是AVL树，它的任何节点的两个子树的高度差<=1；右边的不是AVL树，其根节点的左子树高度为3，而右子树高度为1；   


如果在AVL树中进行插入或删除节点，可能导致AVL树失去平衡，这种失去平衡的二叉树可以概括为四种姿态：LL（左左）、RR（右右）、LR（左右）、RL（右左）。它们的示意图如下：   


这四种失去平衡的姿态都有各自的定义：   
LL：LeftLeft，也称“左左”。插入或删除一个节点后，根节点的左孩子（Left Child）的左孩子（Left Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

RR：RightRight，也称“右右”。插入或删除一个节点后，根节点的右孩子（Right Child）的右孩子（Right Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

LR：LeftRight，也称“左右”。插入或删除一个节点后，根节点的左孩子（Left Child）的右孩子（Right Child）还有非空节点，导致根节点的左子树高度比右子树高度高2，AVL树失去平衡。

RL：RightLeft，也称“右左”。插入或删除一个节点后，根节点的右孩子（Right Child）的左孩子（Left Child）还有非空节点，导致根节点的右子树高度比左子树高度高2，AVL树失去平衡。

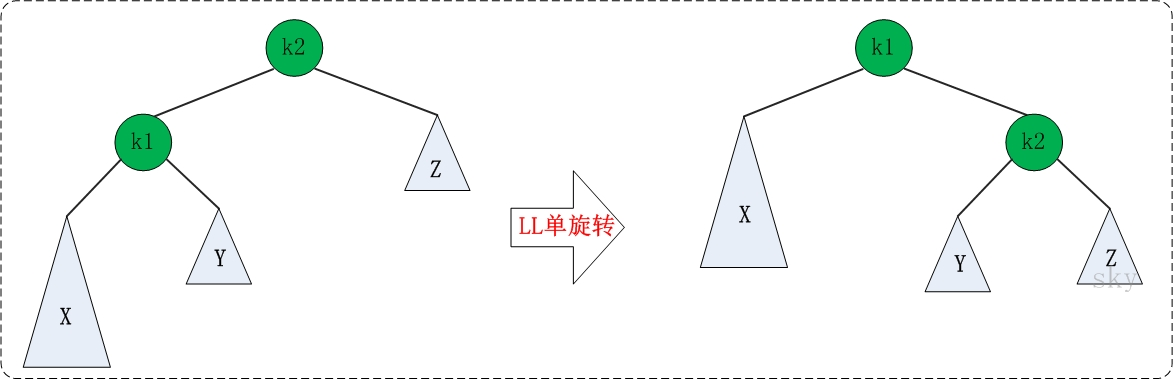
AVL树失去平衡之后，可以通过旋转使其恢复平衡。下面分别介绍四种失去平衡的情况下对应的旋转方法。

LL的旋转。LL失去平衡的情况下，可以通过一次旋转让AVL树恢复平衡。步骤如下：

将根节点的左孩子作为新根节点。

将新根节点的右孩子作为原根节点的左孩子。

将原根节点作为新根节点的右孩子。

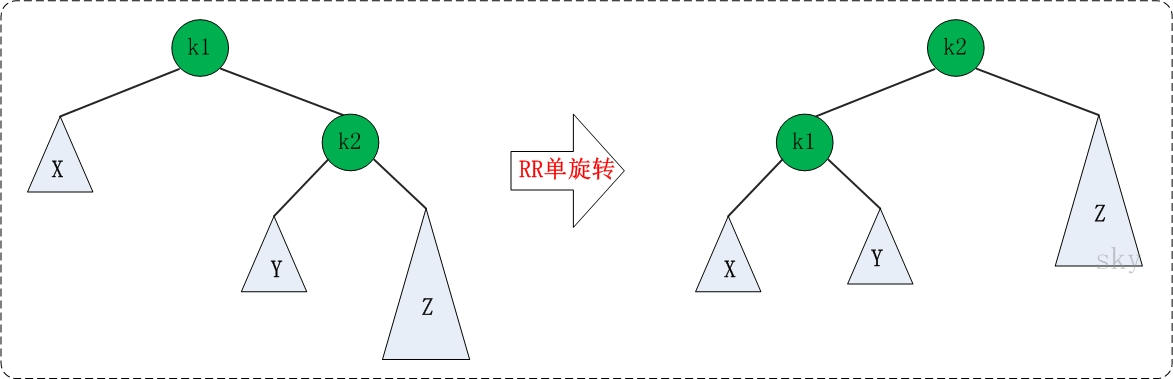
LL旋转示意图如下：   


RR的旋转：RR失去平衡的情况下，旋转方法与LL旋转对称，步骤如下：

将根节点的右孩子作为新根节点。

将新根节点的左孩子作为原根节点的右孩子。

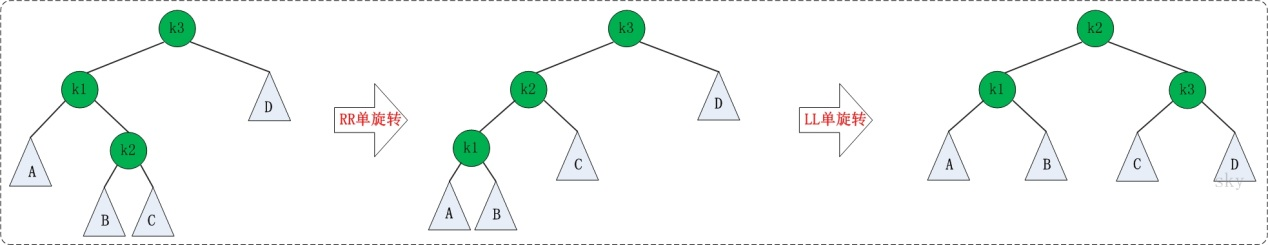
将原根节点作为新根节点的左孩子。

RR旋转示意图如下：   


LR的旋转：LR失去平衡的情况下，需要进行两次旋转，步骤如下：

围绕根节点的左孩子进行RR旋转。

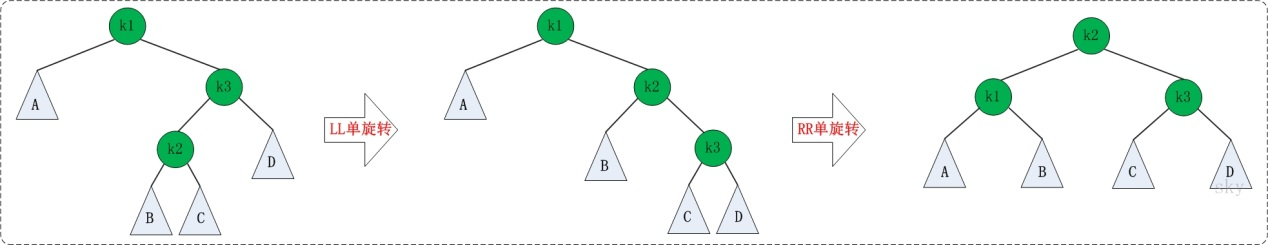
围绕根节点进行LL旋转。

LR的旋转示意图如下：   


RL的旋转：RL失去平衡的情况下也需要进行两次旋转，旋转方法与LR旋转对称，步骤如下：

围绕根节点的右孩子进行LL旋转。

围绕根节点进行RR旋转。

RL的旋转示意图如下：   


平衡多路查找树（B-Tree）

B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。

InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb\_page\_size将页的大小设置为4K、8K、16K，在[MySQL](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://www.cnblogs.com/vianzhang/p/_blank)中可通过如下命令查看页的大小：

mysql> show variables like 'innodb\_page\_size';

1

1

而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。InnoDB在把磁盘数据读入到磁盘时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data] ，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

一棵m阶的B-Tree有如下特性：   
1. 每个节点最多有m个孩子。   
2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。   
3. 若根节点不是叶子节点，则至少有2个孩子   
4. 所有叶子节点都在同一层，且不包含其它关键字信息   
5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）   
6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1   
7. ki(i=1,…n)为关键字，且关键字升序排序。   
8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：   


每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

模拟查找关键字29的过程：

根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】

比较关键字29在区间（17,35），找到磁盘块1的指针P2。

根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】

比较关键字29在区间（26,30），找到磁盘块3的指针P2。

根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】

在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

B+Tree

B+Tree是在B-Tree基础上的一种优化，使其更适合实现外存储索引结构，InnoDB存储引擎就是用B+Tree实现其索引结构。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

非叶子节点只存储键值信息。

所有叶子节点之间都有一个链指针。

数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：   


通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 \* 10^3 \* 10^3 = 10亿 条记录。

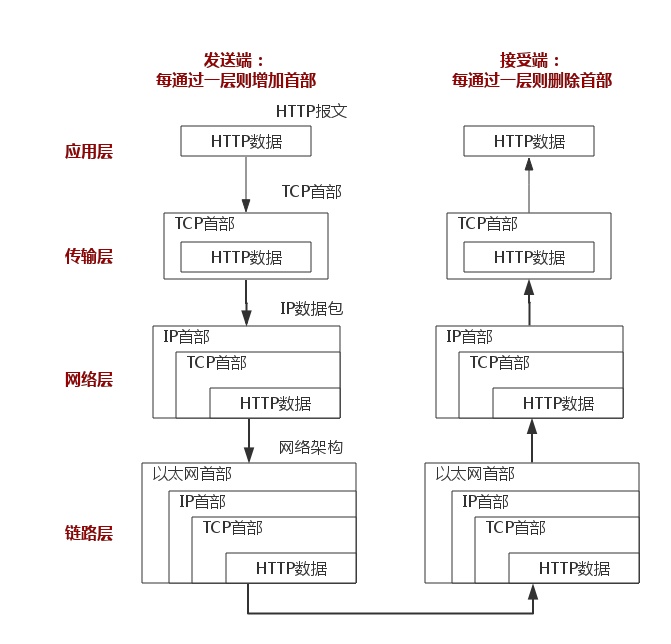
实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。[mysql](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "https://www.cnblogs.com/vianzhang/p/_blank)的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为聚集索引（clustered index）和辅助索引（secondary index）。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

# springboot启动过程及原理

* 首先，springboot有自己的启动入口类以及main方法，我们启动main方法的时候他就做了两步，第一步new了一个SpringApplication对象 ，第二步调用了run()方法，new SpringApplication()没做啥事情 ，主要加载了META-INF/spring.factories 下面定义的事件监听器接口实现类，最主要的就在run方法里了，他启动计时器，获取事件监听器SpringApplicationRunListener类型，并且执行starting()方法-，监听之后配置好spring环境，调用refreshContext(context) 这个方法，这个方法启动spring的代码加载了bean，还启动了内置web容器，扫描并且初始化单实列bean，选择容器类型，tomcat就是在这里启动的。
* 总结：1、run() 方法主要调用了spring容器启动方法扫描配置，加载bean到spring容器中
* 2、启动的内置Web容器

# 51.**[浏览器输入一个地址的过程分析](https://www.cnblogs.com/kongtongshu/p/11069603.html)**



* DNS解析过程,寻找对应的服务器ip地址 (**应用层**)  
  可能会有一次向外部DNS的请求  
  (参照 [DNS过程分析](https://www.cnblogs.com/fanfan259/p/10240670.html))
* 建立TCP连接,利用这个连接发送数据 (**传输层**)  
  三次握手
* 封装HTTP请求包，HTTP或HTTPS,添加相应的协议内容 (**应用层**)  
  HTTP头
* 封装TCP请求包，TCP或UDP,添加相应的协议内容（**传输层**）  
  TCP头
* 封装IP请求包，添加IP协议相应内容 （**网络层**）  
  IP头
* 封装ARP请求包，添加ARP协议相应内容 （**网络层**）  
  网卡发出去封装好的mac包，到达网关  
  MAC头
* 请求包打包完毕，通过网络，到达网络上,添加相应的以太网头 (**数据链路层**)  
  以太网头
* 请求包根据路由协议(相关路由表)，传输网络包 (**网络层**)
* 网络包根据RARP协议，找到目标服务器的IP地址 （**网络层**）
* 目标服务器发现MAC地址对的上，交给操作系统的网络层 （**网络层 传输层**）  
  网络层发现IP对的上，交给传输层TCP(TCP都会发送回复包,用于消息确认)  
  通过TCP包中的端口号，交给正在监听此端口号的进程  
  假设是Nginx,处理完成后，将相关数据打包，经过数据包的流转，回复给浏览器

应用层协议：DNS HTTP HTTPS  
传输层协议：TCP UDP  
网络层协议：IP ARP RARP

# 52.阿里巴巴java开发手册目录结构

1.编程规约

2.异常日志

3.单元测试

4安全规约

1. mysql数据库
2. 工程结构
3. 设计规约
4. 历史版本及专有名词解释

# 53.权限设计

根据角色授权的思想，我们需要涉及五张表(简单一写，没写约束，凑活看吧)

1）三张主表

a）用户表（user）

b） 角色表（role）

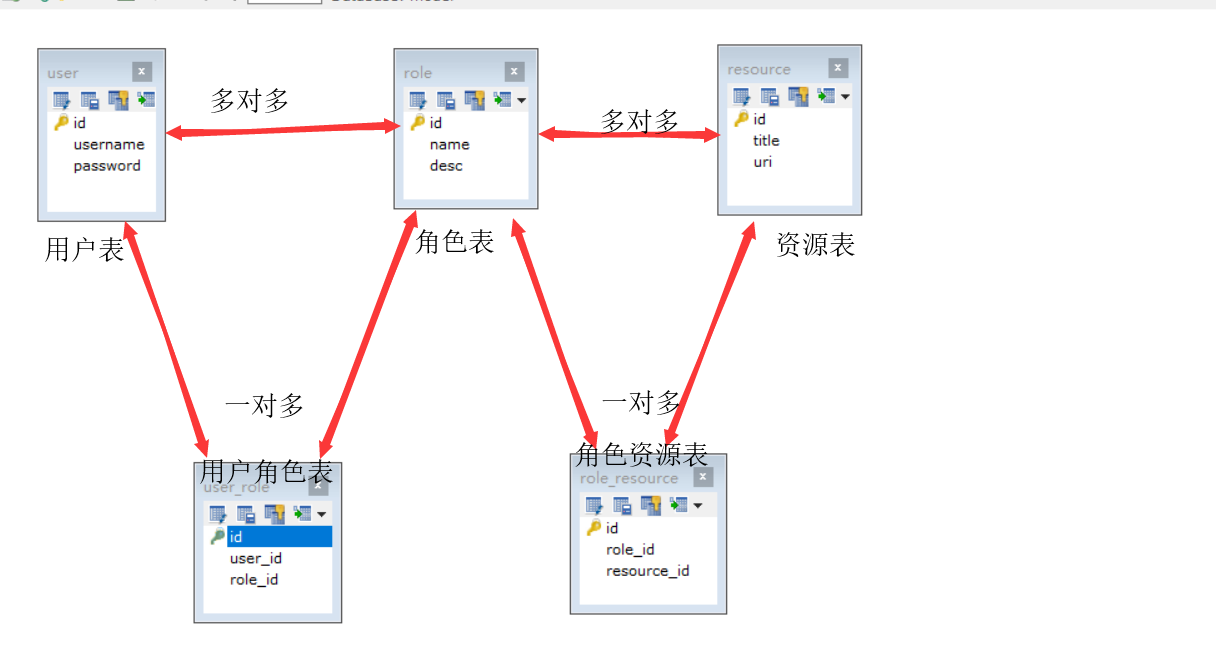
c） 资源表（module）[你也可以叫他权限表等等，反正就是代表着各种权限]

2）两张中间表

d）用户角色表（user\_role）

e）角色资源表（permission)

 db.sql



## **四：使用Shiro整合Spring进行管理权限**

### 1：Shiro简介

　　Apache Shiro是一个强大且易用的Java安全框架,执行身份验证、授权、密码学和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

### Shiro认证过程



1）：引入shiro的jar包

 shiro坐标

2）：创建类文件

　　2.1构建SecurityManager环境

 View Code

2.2主体提交认证请求

 提交认证

### 2：Shiro授权过程



1）类文件

1.1构建SecurityManager环境

 环境

1.2主体提交认证请求

 提交认证

### 3：关于Realm

Realm一般有三种，分别是，一般都使用自定义

1）IniRealm控制   .Ini 文件。

2）jdbcRealm控制 数据库文件

引入Mysql驱动包，引入数据源文件，设置jdbcurl以及用户名和密码

 View Code

主体提交验证

 验证

3）自定义realm

 MyJdbcRealm

### 4：关于md5加密

　　4.1环境搭建及使用

 main

4.2 MD5加密

 加密

# 54.保持登录状态的几种方式

## **（一）session机制保持会话**

### **存在的问题**

* 高并发情况下，会占用服务器大量内存
* 分布式（一个业务分成几个子业务，部署在多个服务器）或者集群（一个业务部署在多个服务器）的时候，session不能共享。

### **解决方案**

* 高并发的时候可以将session存储到redis，如果用户长时间没有访问，将session存储到redis，就减少了服务器的压力。
* 分布式或者集群的时候，先通过redis来判断用户状态也可以实现session共享.

## **（二）cookie机制保持会话**

### **使用的方法**

* 登录验证后，创建登录凭证（比如：用户id+登录时间+过期时间），将登录凭证进行加密（为了避免暴露信息），加密后写到浏览器的cookie，以后，每次请求都发送cookie，服务器根据对应的解密算法对其进行验证（或者将加密过的cookie内容存储到数据库，请求服务器的时候，服务器在数据库进行查找）。

### **存在的问题**

* 每次访问都提交cookie，增加请求量
* 其他访问可能需要cookie（比如说购物车的信息存放在cookie），浏览器对每个域存储的cookie的大小有限制，那么需要控制加密后的凭证。

## **（三）token机制保持会话**

### **使用方法**

* cookie 和session依赖于浏览器，如果客户端不是浏览器，那么需要手动添加token（和cookie类似，也是登录凭证），将token添加到http header或者做为参数添加到url。

### **存在的问题**

* 每次访问的时候手动添加token
* 和cookie 的方式一样增加了请求量

# 55.如何设计IO密集型多线程和CPU密集型多线程？

常见的任务分为两种：CPU密集型任务和IO密集型任务

CPU密集型任务（CPU-bound）：在一个任务中，主要做计算，CPU持续在运行，CPU利用率高，具有这种特点的任务称为CPU密集型任务。

IO密集型任务（IO-bound）：在一个任务中，大部分时间在进行I/O操作，由于I/O速度远远小于CPU，所以任务的大部分时间都在等待IO，CPU利用率低，具有这种特点的任务称为IO密集型任务。

所以我们在设计线程池时，应先对执行的任务有个大体分类，然后根据类型进行设置。一般而言，两种任务的线程数设置如下：

CPU密集型任务：线程个数为CPU核数。这几个线程可以并行执行，不存在线程切换到开销，提高了cpu的利用率的同时也减少了切换线程导致的性能损耗

IO密集型：线程个数为CPU核数的两倍。到其中的线程在IO操作的时候，其他线程可以继续用cpu，提高了cpu的利用率

————————————————

版权声明：本文为CSDN博主「XavierLe」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。

原文链接：<https://blog.csdn.net/LGM_lx/article/details/88898036>

# 56.进程之间的几种通讯方式

1.管道：速度慢，容量有限，只有父子进程能通讯

2.FIFO：任何进程间都能通讯，但速度慢

3.消息队列：容量受到系统限制，且要注意第一次读的时候，要考虑上一次没有读完数据的问题

4.信号量：不能传递复杂消息，只能用来同步

5.共享内存区：能够很容易控制容量，速度快，但要保持同步，比如一个进程在写的时候，另一个进程要注意读写的问题，相当于线程中的线程安全，当然，共享内存区同样可以用作线程间通讯，不过没这个必要，线程间本来就已经共享了同一进程内的一块内存

# 57.线程之间的几种通讯方式

### **锁机制：包括互斥锁、条件变量、读写锁**

互斥锁提供了以排他方式防止数据结构被并发修改的方法。   
读写锁允许多个线程同时读共享数据，而对写操作是互斥的。   
条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

wait/notify 等待

Volatile 内存共享

CountDownLatch 并发工具

CyclicBarrier 并发工具

### **信号量机制(Semaphore)**

包括无名线程信号量和命名线程信号量。

### **信号机制(Signal)**

类似进程间的信号处理。

线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

58**.乐观锁及悲观锁**

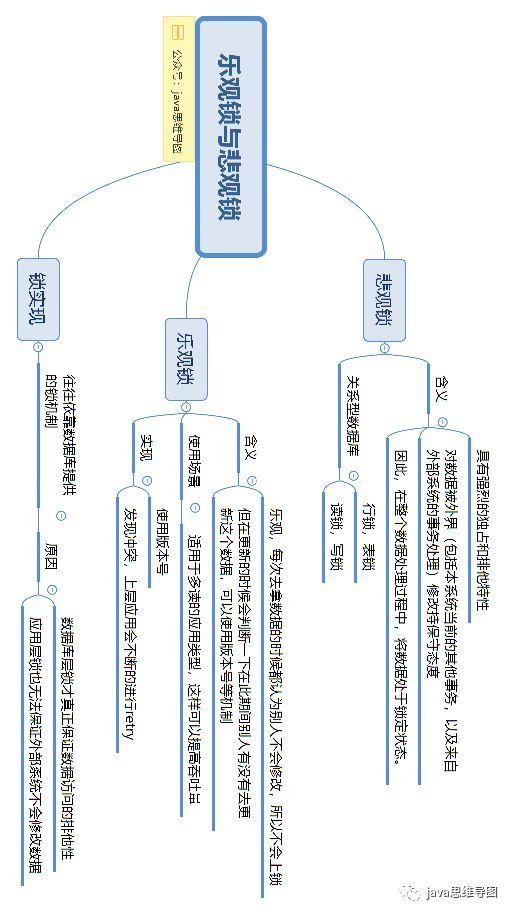
悲观锁(Pessimistic Lock)

顾名思义，就是很悲观，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。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

乐观锁(Optimistic Lock)

顾名思义，就是很乐观，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。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。



# 59.liunx批量修改

1.perl命令替换

格式: perl -p -i -e "s/修改的字段/新字段/g" 文件名（别的路径下需加好路径）

perl -p -i -e "s/123/888/g" test1.txt test2.txt

说明：将test1.txt和test2.txt中的123替换成888

说明：如果想替换所有的txt文件可以用\*.txt(perl -p -i -e "s/123/888/g" \*.txt )

## **2.sed命令下批量替换文件内容**

格式: sed -i "s/查找字段/替换字段/g" `grep 查找字段 -rl 路径` 文件名



说明：  把当前目录下text3.txt里的123都替换为444

# 60.Java 8 新特性

Java 8 (又称为 jdk 1.8) 是 Java 语言开发的一个主要版本。 Oracle 公司于 2014 年 3 月 18 日发布 Java 8 ，它支持函数式编程，新的 JavaScript 引擎，新的日期 API，新的Stream API 等。

## 新特性

Java8 新增了非常多的特性，我们主要讨论以下几个：

**Lambda 表达式** − Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中）。

**方法引用** − 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

**默认方法** − 默认方法就是一个在接口里面有了一个实现的方法。

**新工具** − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。

**Stream API** −新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。

**Date Time API** − 加强对日期与时间的处理。

**Optional 类** − Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。

**Nashorn, JavaScript 引擎** − Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

更多的新特性可以参阅官网：[What's New in JDK 8](http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html" \t "https://www.runoob.com/java/_blank)