# NoSql之Redis

# NoSql入门和概述

## 入门概述

### 互联网时代背景下 大机遇，为什么用nosql

#### 1单机MySQL的美好年代

|  |
| --- |
| 在90年代，一个网站的访问量一般都不大，用单个数据库完全可以轻松应付。  在那个时候，更多的都是静态网页，动态交互类型的网站不多。    上述架构下，我们来看看数据存储的瓶颈是什么？  1.数据量的总大小 一个机器放不下时  2.数据的索引（B+ Tree）一个机器的内存放不下时  3.访问量(读写混合)一个实例不能承受   如果满足了上述1 or 3个，进化...... |

#### 2 Memcached(缓存)+MySQL+垂直拆分

|  |
| --- |
| 后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大的时候，多台web机器通过文件缓存不能共享，大量的小文件缓存也带了了比较高的IO压力。在这个时候，Memcached就自然的成为一个非常时尚的技术产品。     Memcached作为一个独立的分布式的缓存服务器，为多个web服务器提供了一个共享的高性能缓存服务，在Memcached服务器上，又发展了根据hash算法来进行多台Memcached缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端 |

#### 3 Mysql主从读写分离

|  |
| --- |
| 由于数据库的写入压力增加，Memcached只能缓解数据库的读取压力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了。 |

#### 4 分表分库+水平拆分+mysql集群

|  |
| --- |
| 在Memcached的高速缓存，MySQL的主从复制，读写分离的基础之上，这时MySQL主库的写压力开始出现瓶颈，而数据量的持续猛增，由于MyISAM使用表锁，在高并发下会出现严重的锁问题，大量的高并发MySQL应用开始使用InnoDB引擎代替MyISAM。   同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。 |

#### 5 MySQL的扩展性瓶颈

|  |
| --- |
| MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小，如果能把这些数据从MySQL省去，MySQL将变得非常的小。关系数据库很强大，但是它并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，表结构更改困难，正是当前使用MySQL的开发人员面临的问题。 |

#### 6 今天是什么样子？？

|  |
| --- |
|  |

#### 7 为什么用NoSQL

|  |
| --- |
| 为什么使用NoSQL ?  今天我们可以通过第三方平台（如：Google,Facebook等）可以很容易的访问和抓取数据。用户的个人信息，社交网络，地理位置，用户生成的数据和用户操作日志已经成倍的增加。我们如果要对这些用户数据进行挖掘，那SQL数据库已经不适合这些应用了, NoSQL数据库的发展也却能很好的处理这些大的数据。 |

### 是什么

|  |
| --- |
| NoSQL(NoSQL = Not Only SQL )，意即“不仅仅是SQL”，  泛指非关系型的数据库。随着互联网web2.0网站的兴起，传统的关系数据库在应付web2.0网站，特别是超大规模和高并发的SNS类型的web2.0纯动态网站已经显得力不从心，暴露了很多难以克服的问题，而非关系型的数据库则由于其本身的特点得到了非常迅速的发展。NoSQL数据库的产生就是为了解决大规模数据集合多重数据种类带来的挑战，尤其是大数据应用难题，包括超大规模数据的存储。  （例如谷歌或Facebook每天为他们的用户收集万亿比特的数据）。这些类型的数据存储不需要固定的模式，无需多余操作就可以横向扩展。 |

### 能干嘛

#### 易扩展

|  |
| --- |
| NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。  数据之间无关系，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。 |

#### 大数据量高性能

|  |
| --- |
| NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。  这得益于它的无关系性，数据库的结构简单。  一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，  在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，  是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了 |

#### 多样灵活的数据模型

|  |
| --- |
| NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，  增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦 |

#### 传统RDBMS VS NOSQL

|  |
| --- |
| RDBMS vs NoSQL  RDBMS  - 高度组织化结构化数据  - 结构化查询语言（SQL）  - 数据和关系都存储在单独的表中。  - 数据操纵语言，数据定义语言  - 严格的一致性  - 基础事务  NoSQL  - 代表着不仅仅是SQL  - 没有声明性查询语言  - 没有预定义的模式  -键 - 值对存储，列存储，文档存储，图形数据库  - 最终一致性，而非ACID属性  - 非结构化和不可预知的数据  - CAP定理  - 高性能，高可用性和可伸缩性 |

### 去哪下

* Redis
* Memcache
* Mongodb

### 怎么玩

* KV
* Cache
* Persistence
* ...

## 3V+3高

### 大数据时代的3V

* 海量Volume
* 多样Variety
* 实时Velocity

### 互联网需求的3高

* 高并发
* 高可扩
* 高性能

## 当下的NoSQL经典应用

### 当下的应用是sql和nosql一起使用

### 阿里巴巴中文站商品信息如何存放

#### 看看阿里巴巴中文网站首页 以女装/女包包为例

##### 架构发展历程

* 演变过程

|  |
| --- |
|  |

* 第5代

|  |
| --- |
|  |

* 第5代架构使命

|  |
| --- |
|  |

* ......

##### 和我们相关的，多数据源多数据类型的存储问题

|  |
| --- |
|  |

#### 商品基本信息

##### 名称、价格，出厂日期，生产厂商等

##### 关系型数据库：mysql/oracle目前淘宝在去O化(也即拿掉Oracle)，注意，淘宝内部用的Mysql是里面的大牛自己改造过的

* 为什么去IOE？

|  |
| --- |
| 2008年，王坚加盟阿里巴巴成为集团首席架构师，即现在的首席技术官。这位前微软亚洲研究院常务副院长被马云定位为：将帮助阿里巴巴集团建立世界级的技术团队，并负责集团技术架构以及基础技术平台搭建。  在加入阿里后，带着技术基因和学者风范的王坚就在阿里巴巴集团提出了被称为“去IOE”（在IT建设过程中，去除IBM小型机、Oracle数据库及EMC存储设备）的想法，并开始把云计算的本质，植入阿里IT基因。  王坚这样概括“去IOE”运动和阿里云之间的关系：“去IOE”彻底改变了阿里集团IT架构的基础，是阿里拥抱云计算，产出计算服务的基础。“去IOE”的本质是分布化，让随处可以买到的Commodity PC架构成为可能，使云计算能够落地的首要条件。 |

#### 商品描述、详情、评价信息(多文字类)

* 多文字信息描述类，IO读写性能变差
* 文档数据库MongDB中

#### 商品的图片

##### 商品图片展现类

##### 分布式的文件系统中

* 淘宝自己的TFS
* Google的GFS
* Hadoop的HDFS

#### 商品的关键字

* 搜索引擎，淘宝内用
* ISearch

#### 商品的波段性的热点高频信息

* 内存数据库
* tair、Redis、Memcache

#### 6 商品的交易、价格计算、积分累计

* 外部系统，外部第3方支付接口
* 支付宝

#### 总结大型互联网应用(大数据、高并发、多样数据类型)的难点和解决方案

##### 难点

* 数据类型多样性
* 数据源多样性和变化重构
* 数据源改造而数据服务平台不需要大面积重构

##### 解决办法

**阿里、淘宝干了什么？UDSL**

###### 是什么

|  |
| --- |
|  |

###### 什么样

|  |
| --- |
|  |

1. 映射

|  |
| --- |
|  |

1. API

|  |
| --- |
|  |

1. 热点缓存

|  |
| --- |
|  |

1. ......

## NoSQL数据模型简介

### 以一个电商客户、订单、订购、地址模型来对比下关系型数据库和非关系型数据库

#### 传统的关系型数据库你如何设计？

* **ER图(1:1/1:N/N:N,主外键等常见)**

|  |
| --- |
|  |

#### nosql你如何设计

##### 什么是BSON

|  |
| --- |
| BSON（）是一种类json的一种二进制形式的存储格式，简称Binary JSON，  它和JSON一样，支持内嵌的文档对象和数组对象 |

##### 用BSon画出构建的数据模型

|  |
| --- |
| {   "customer":{     "id":1136,     "name":"Z3",     "billingAddress":[{"city":"beijing"}],     "orders":[      {        "id":17,        "customerId":1136,        "orderItems":[{"productId":27,"price":77.5,"productName":"thinking in java"}],        "shippingAddress":[{"city":"beijing"}]        "orderPayment":[{"ccinfo":"111-222-333","txnid":"asdfadcd334","billingAddress":{"city":"beijing"}}],        }      ]    }  } |

#### 两者对比，问题和难点

##### 为什么上述的情况可以用聚合模型来处理

* 高并发的操作是不太建议有关联查询的，互联网公司用冗余数据来避免关联查询
* 分布式事务是支持不了太多的并发的

##### 想想关系模型数据库你如何查？

### 聚合模型

* KV键值
* Bson
* 列族

|  |
| --- |
| 顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。 |

* 图形

|  |
| --- |
|  |

## NoSQL数据库的四大分类

### KV键值：典型介绍

* 新浪：BerkeleyDB+redis
* 美团：redis+tair
* 阿里、百度：memcache+redis

### 文档型数据库(bson格式比较多)：典型介绍

* CouchDB
* MongoDB

|  |
| --- |
| MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。  MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。 |

### 列存储数据库

* Cassandra, HBase
* 分布式文件系统

### 图关系数据库

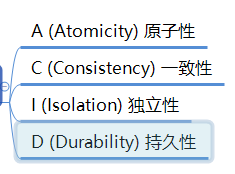
* 它不是放图形的，放的是关系比如:朋友圈社交网络、广告推荐系统
* 社交网络，推荐系统等。专注于构建关系图谱
* Neo4J, InfoGrid

### 四者对比

|  |
| --- |
|  |

## 在分布式数据库中CAP原理CAP+BASE

### 传统的ACID分别是什么



|  |
| --- |
| 关系型数据库遵循ACID规则  事务在英文中是transaction，和现实世界中的交易很类似，它有如下四个特性：  1、A (Atomicity) 原子性  原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做，事务成功的条件是事务里的所有操作都成功，只要有一个操作失败，整个事务就失败，需要回滚。比如银行转账，从A账户转100元至B账户，分为两个步骤：1）从A账户取100元；2）存入100元至B账户。这两步要么一起完成，要么一起不完成，如果只完成第一步，第二步失败，钱会莫名其妙少了100元。  2、C (Consistency) 一致性  一致性也比较容易理解，也就是说数据库要一直处于一致的状态，事务的运行不会改变数据库原本的一致性约束。  3、I (Isolation) 独立性  所谓的独立性是指并发的事务之间不会互相影响，如果一个事务要访问的数据正在被另外一个事务修改，只要另外一个事务未提交，它所访问的数据就不受未提交事务的影响。比如现有有个交易是从A账户转100元至B账户，在这个交易还未完成的情况下，如果此时B查询自己的账户，是看不到新增加的100元的  4、D (Durability) 持久性  持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。 |

### CAP

* C:Consistency（强一致性）
* A:Availability（可用性）
* P:Partition tolerance（分区容错性）

### CAP的3进2

|  |
| --- |
| CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。  而由于当前的网络硬件肯定会出现延迟丢包等问题，所以  分区容忍性是我们必须需要实现的。  所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。  ===============================================================================C:强一致性 A：高可用性 P：分布式容忍性   CA 传统Oracle数据库   AP 大多数网站架构的选择   CP Redis、Mongodb   注意：分布式架构的时候必须做出取舍。  一致性和可用性之间取一个平衡。多余大多数web应用，其实并不需要强一致性。  因此牺牲C换取P，这是目前分布式数据库产品的方向  ===============================================================================  一致性与可用性的决择  对于web2.0网站来说，关系数据库的很多主要特性却往往无用武之地  数据库事务一致性需求  　　很多web实时系统并不要求严格的数据库事务，对读一致性的要求很低， 有些场合对写一致性要求并不高。允许实现最终一致性。  数据库的写实时性和读实时性需求  　　对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多web应用来说，并不要求这么高的实时性，比方说发一条消息之 后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。  对复杂的SQL查询，特别是多表关联查询的需求  　　任何大数据量的web系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是SNS类型的网站，从需求以及产品设计角 度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL的功能被极大的弱化了。 |

### 经典CAP图

|  |
| --- |
| CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。  因此，根据 CAP 原理将 NoSQL 数据库分成了满足 CA 原则、满足 CP 原则和满足 AP 原则三 大类：  CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。  CP - 满足一致性，分区容忍必的系统，通常性能不是特别高。  AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。 |

### BASE

|  |
| --- |
| BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。  BASE其实是下面三个术语的缩写：      基本可用（Basically Available）      软状态（Soft state）      最终一致（Eventually consistent）  它的思想是通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观。为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法 |

### 分布式+集群简介

|  |
| --- |
| 分布式系统  分布式系统（distributed system）   由多台计算机和通信的软件组件通过计算机网络连接（本地网络或广域网）组成。分布式系统是建立在网络之上的软件系统。正是因为软件的特性，所以分布式系统具有高度的内聚性和透明性。因此，网络和分布式系统之间的区别更多的在于高层软件（特别是操作系统），而不是硬件。分布式系统可以应用在在不同的平台上如：Pc、工作站、局域网和广域网上等。  简单来讲：  1分布式：不同的多台服务器上面部署不同的服务模块（工程），他们之间通过Rpc/Rmi之间通信和调用，对外提供服务和组内协作。  2集群：不同的多台服务器上面部署相同的服务模块，通过分布式调度软件进行统一的调度，对外提供服务和访问。 |

# Redis入门介绍

## 入门概述

### 是什么

#### Redis:REmote DIctionary Server(远程字典服务器)

|  |
| --- |
|  |

#### 是完全开源免费的，用C语言编写的，遵守BSD协议，是一个高性能的(key/value)分布式内存数据库，基于内存运行并支持持久化的NoSQL数据库，是当前最热门的NoSql数据库之一,也被人们称为数据结构服务器

#### Redis 与其他 key - value 缓存产品有以下三个特点

* 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
* Redis不仅仅支持简单的key-value类型的数据，同时还提供list，set，zset，hash等数据结构的存储
* Redis支持数据的备份，即master-slave模式的数据备份

### 能干嘛

1. 内存存储和持久化：redis支持异步将内存中的数据写到硬盘上，同时不影响继续服务
2. 取最新N个数据的操作，如：可以将最新的10条评论的ID放在Redis的List集合里面
3. 模拟类似于HttpSession这种需要设定过期时间的功能
4. 发布、订阅消息系统
5. 定时器、计数器

### 去哪下

* <http://redis.io/>

|  |
| --- |
|  |

* <http://www.redis.cn/>

|  |
| --- |
|  |

### 怎么玩

#### 数据类型、基本操作和配置

#### 持久化和复制，RDB/AOF

#### 事务的控制

#### 复制

#### ......

## VMWare+VMTools千里之行始于足下

## Redis的安装

### Windows版安装

|  |
| --- |
| Window 下安装  下载地址：https://github.com/dmajkic/redis/downloads  下载到的Redis支持32bit和64bit。根据自己实际情况选择，将64bit的内容cp到自定义盘符安装目录取名redis。 如 C:\reids  打开一个cmd窗口 使用cd命令切换目录到 C:\redis 运行 redis-server.exe redis.conf 。  如果想方便的话，可以把redis的路径加到系统的环境变量里，这样就省得再输路径了，后面的那个redis.conf可以省略，  如果省略，会启用默认的。输入之后，会显示如下界面：  这时候另启一个cmd窗口，原来的不要关闭，不然就无法访问服务端了。  切换到redis目录下运行 redis-cli.exe -h 127.0.0.1 -p 6379 。  设置键值对 set myKey abc  取出键值对 get myKey |

### 重要提示：

由于企业里面做Redis开发，99%都是Linux版的运用和安装，

几乎不会涉及到Windows版，上一步的讲解只是为了知识的完整性，

Windows版不作为重点，同学可以下去自己玩，企业实战就认一个版：Linux

|  |
| --- |
|  |

### Linux版安装

#### 下载获得redis-3.0.4.tar.gz后将它放入我们的Linux目录/opt

#### /opt目录下，解压命令:tar -zxvf redis-3.0.4.tar.gz

#### 解压完成后出现文件夹：redis-3.0.4

|  |
| --- |
|  |

#### 进入目录:cd redis-3.0.4

|  |
| --- |
|  |

#### 在redis-3.0.4目录下执行make命令

* **运行make命令时故意出现的错误解析：**

##### 安装gcc

|  |
| --- |
| gcc是linux下的一个编译程序，是C程序的编译工具。  GCC(GNU Compiler Collection) 是 GNU(GNU's Not Unix) 计划提供的编译器家族，它能够支持 C, C++, Objective-C, Fortran, Java 和 Ada 等等程序设计语言前端，同时能够运行在 x86, x86-64, IA-64, PowerPC, SPARC 和 Alpha 等等几乎目前所有的硬件平台上。鉴于这些特征，以及 GCC 编译代码的高效性，使得 GCC 成为绝大多数自由软件开发编译的首选工具。虽然对于程序员们来说，编译器只是一个工具，除了开发和维护人员，很少有人关注编译器的发展，但是 GCC 的影响力是如此之大，它的性能提升甚至有望改善所有的自由软件的运行效率，同时它的内部结构的变化也体现出现代编译器发展的新特征。 |

能上网：yum install gcc-c++

不上网：

##### 二次make

##### jemalloc/jemalloc.h：没有那个文件或目录

**运行make distclean之后再make**

|  |
| --- |
|  |

##### Redis Test(可以不用执行)

|  |
| --- |
| 下载TCL的网址：  http://www.linuxfromscratch.org/blfs/view/cvs/general/tcl.html    安装TCL |

#### 如果make完成后继续执行make install

|  |
| --- |
|  |

#### 查看默认安装目录：usr/local/bin

|  |
| --- |
|  |

* redis-benchmark:性能测试工具，可以在自己本子运行，看看自己本子性能如何

服务启动起来后执行

* redis-check-aof：修复有问题的AOF文件，rdb和aof后面讲
* redis-check-dump：修复有问题的dump.rdb文件
* redis-cli：客户端，操作入口
* redis-sentinel：redis集群使用
* redis-server：Redis服务器启动命令

#### 启动

|  |
| --- |
|  |

##### 修改redis.conf文件将里面的daemonize no 改成 yes，让服务在后台启动

##### 将默认的redis.conf拷贝到自己定义好的一个路径下，比如/myconf

##### 启动

|  |
| --- |
|  |

##### 连通测试

|  |
| --- |
|  |

##### /usr/local/bin目录下运行redis-server，运行拷贝出存放了自定义conf文件目录下的redis.conf文件

#### 永远的helloworld

|  |
| --- |
|  |

#### 关闭

|  |
| --- |
|  |

1. 单实例关闭：redis-cli shutdown
2. 多实例关闭，指定端口关闭:redis-cli -p 6379 shutdown

## Redis启动后杂项基础知识讲解

##### 单进程

* 单进程模型来处理客户端的请求。对读写等事件的响应是通过对epoll函数的包装来做到的。Redis的实际处理速度完全依靠主进程的执行效率
* epoll是Linux内核为处理大批量文件描述符而作了改进的epoll，是Linux下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率。

##### 默认16个数据库，类似数组下表从零开始，初始默认使用零号库

|  |
| --- |
|  |

##### select命令切换数据库

##### dbsize查看当前数据库的key的数量

##### flushdb：清空当前库

##### Flushall；通杀全部库

##### 统一密码管理，16个库都是同样密码，要么都OK要么一个也连接不上

##### Redis索引都是从零开始

##### 为什么默认端口是6379

# Redis数据类型

## Redis的五大数据类型

|  |
| --- |
|  |

### string（字符串）

|  |
| --- |
| String（字符串）  string是redis最基本的类型，你可以理解成与Memcached一模一样的类型，一个key对应一个value。  string类型是二进制安全的。意思是redis的string可以包含任何数据。比如jpg图片或者序列化的对象 。  string类型是Redis最基本的数据类型，一个redis中字符串value最多可以是512M |

### hash（哈希，类似java里的Map）

|  |
| --- |
| Hash（哈希）  Redis hash 是一个键值对集合。  Redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象。  类似Java里面的Map<String,Object> |

### list（列表）

|  |
| --- |
| List（列表）  Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素导列表的头部（左边）或者尾部（右边）。  它的底层实际是个链表 |

### set（集合）

|  |
| --- |
| Set（集合）  Redis的Set是string类型的无序集合。它是通过HashTable实现实现的， |

### zset(sorted set：有序集合)

|  |
| --- |
| zset(sorted set：有序集合)  Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。  不同的是每个元素都会关联一个double类型的分数。  redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的,但分数(score)却可以重复。 |

## 哪里去获得redis常见数据类型操作命令

<http://redisdoc.com/>

|  |
| --- |
|  |

## Redis 键(key)

### 常用

|  |
| --- |
|  |

### 案例

1. keys \*
2. exists key的名字，判断某个key是否存在
3. move key db --->当前库就没有了，被移除了
4. expire key 秒钟：为给定的key设置过期时间
5. ttl key 查看还有多少秒过期，-1表示永不过期，-2表示已过期
6. type key 查看你的key是什么类型

## Redis字符串(String)

|  |
| --- |
|  |

### 常用

|  |
| --- |
|  |

### 单值单value

### 案例

1. set/get/del/append/strlen
2. Incr/decr/incrby/decrby,一定要是数字才能进行加减
3. getrange/setrange

|  |
| --- |
| getrange:获取指定区间范围内的值，类似between......and的关系  从零到负一表示全部    setrange设置指定区间范围内的值，格式是setrange key值 具体值 |

1. setex(set with expire)键秒值/setnx(set if not exist)

|  |
| --- |
| setex:设置带过期时间的key，动态设置。  setex 键 秒值 真实值  setnx:只有在 key 不存在时设置 key 的值。 |

1. mset/mget/msetnx

|  |
| --- |
| mset:同时设置一个或多个 key-value 对。  mget:获取所有(一个或多个)给定 key 的值。  msetnx:同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。 |

1. getset(先get再set)

|  |
| --- |
| getset:将给定 key 的值设为 value ，并返回 key 的旧值(old value)。  简单一句话，先get然后立即set |

## Redis列表(List)

### 常用

|  |
| --- |
|  |

### 单值多value

### 案例

1. lpush/rpush/lrange
2. lpop/rpop
3. lindex，按照索引下标获得元素(从上到下)

|  |
| --- |
| 通过索引获取列表中的元素 lindex key index |

1. Llen
2. lrem key 删N个value

|  |
| --- |
| \* 从left往right删除2个值等于v1的元素，返回的值为实际删除的数量   \*  LREM list3 0 值，表示删除全部给定的值。零个就是全部值 |

1. ltrim key 开始index 结束index，截取指定范围的值后再赋值给key

|  |
| --- |
| ltrim：截取指定索引区间的元素，格式是ltrim list的key 起始索引 结束索引 |

1. rpoplpush 源列表 目的列表

|  |
| --- |
| 移除列表的最后一个元素，并将该元素添加到另一个列表并返回 |

1. lset key index value

|  |
| --- |
|  |

1. linsert key before/after 值1 值2

|  |
| --- |
| 在list某个已有值的前后再添加具体值 |

1. 性能总结

|  |
| --- |
| 它是一个字符串链表，left、right都可以插入添加；  如果键不存在，创建新的链表；  如果键已存在，新增内容；  如果值全移除，对应的键也就消失了。  链表的操作无论是头和尾效率都极高，但假如是对中间元素进行操作，效率就很惨淡了。 |

## Redis集合(Set)

### 常用

|  |
| --- |
|  |

### 单值多value

### 案例

#### sadd/smembers/sismember

|  |
| --- |
|  |

#### scard，获取集合里面的元素个数

|  |
| --- |
| 获取集合里面的元素个数 |

#### srem key value 删除集合中元素

|  |
| --- |
|  |

#### srandmember key 某个整数(随机出几个数)

|  |
| --- |
| \*   从set集合里面随机取出2个   \*   如果超过最大数量就全部取出，   \*   如果写的值是负数，比如-3 ，表示需要取出3个，但是可能会有重复值。 |

#### spop key 随机出栈

|  |
| --- |
|  |

#### smove key1 key2 在key1里某个值 作用是将key1里的某个值赋给key2

|  |
| --- |
|  |

#### 数学集合类

1. 差集：sdiff

|  |
| --- |
| 在第一个set里面而不在后面任何一个set里面的项 |

1. 交集：sinter

|  |
| --- |
|  |

1. 并集：sunion

|  |
| --- |
|  |

## Redis哈希(Hash)

|  |
| --- |
|  |

### 常用

|  |
| --- |
|  |

### KV模式不变，但V是一个键值对

### 案例

1. hset/hget/hmset/hmget/hgetall/hdel

|  |
| --- |
|  |

1. Hlen
2. hexists key 在key里面的某个值的key
3. hkeys/hvals

|  |
| --- |
|  |

1. hincrby/hincrbyfloat

|  |
| --- |
|  |

1. Hsetnx

|  |
| --- |
| 不存在赋值，存在了无效。 |

## Redis有序集合Zset(sorted set)

|  |
| --- |
|  |

在set基础上，加一个score值。

之前set是k1 v1 v2 v3，

现在zset是k1 score1 v1 score2 v2

### 常用

|  |
| --- |
|  |

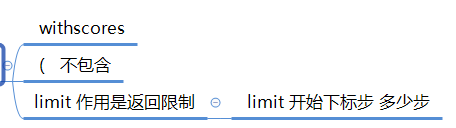
### 案例

1. zadd/zrange

|  |
| --- |
|  |

1. zrangebyscore key 开始score 结束score

|  |
| --- |
|  |



1. zrem key 某score下对应的value值，作用是删除元素

|  |
| --- |
|  |

1. zcard/zcount key score区间/zrank key values值，作用是获得下标值/zscore key 对应值,获得分数

|  |
| --- |
|  |

1. zrevrank key values值，作用是逆序获得下标值

|  |
| --- |
|  |

1. Zrevrange
2. zrevrangebyscore key 结束score 开始score

|  |
| --- |
|  |

# 解析配置文件 redis.conf

## 它在哪

1. 地址

|  |
| --- |
|  |

1. 为什么我将它拷贝出来单独执行？

## units单位

|  |
| --- |
| 1  配置大小单位,开头定义了一些基本的度量单位，只支持bytes，不支持bit      2  对大小写不敏感 |

## INCLUDES包含

|  |
| --- |
| 和我们的Struts2配置文件类似，可以通过includes包含，redis.conf可以作为总闸，包含其他 |

## GENERAL通用

1. daemonize 出厂默认为fasle，true表示以后台模式运行
2. pidfile
3. port
4. tcp-backlog

|  |
| --- |
| tcp-backlog  设置tcp的backlog，backlog其实是一个连接队列，backlog队列总和=未完成三次握手队列 + 已经完成三次握手队列。  在高并发环境下你需要一个高backlog值来避免慢客户端连接问题。注意Linux内核会将这个值减小到/proc/sys/net/core/somaxconn的值，所以需要确认增大somaxconn和tcp\_max\_syn\_backlog两个值  来达到想要的效果 |

1. timeout
2. bind
3. tcp-keepalive

|  |
| --- |
| 单位为秒，如果设置为0，则不会进行Keepalive检测，建议设置成60 |

1. Loglevel

|  |
| --- |
|  |

1. Logfile
2. syslog-enabled

|  |
| --- |
| 是否把日志输出到syslog中 |

1. syslog-ident

|  |
| --- |
|  |

1. syslog-facility

|  |
| --- |
|  |

1. Databases

默认16个库

## SNAPSHOTTING快照

|  |
| --- |
|  |

### Save

1. save 秒钟 写操作次数

|  |
| --- |
| RDB是整个内存的压缩过的Snapshot，RDB的数据结构，可以配置复合的快照触发条件，  默认  是1分钟内改了1万次，  或5分钟内改了10次，  或15分钟内改了1次。 |

1. 禁用

|  |
| --- |
| 如果想禁用RDB持久化的策略，只要不设置任何save指令，或者给save传入一个空字符串参数也可以 |

### stop-writes-on-bgsave-error

|  |
| --- |
| 如果配置成no，表示你不在乎数据不一致或者有其他的手段发现和控制 |

### Rdbcompression

|  |
| --- |
| rdbcompression：对于存储到磁盘中的快照，可以设置是否进行压缩存储。如果是的话，redis会采用  LZF算法进行压缩。如果你不想消耗CPU来进行压缩的话，可以设置为关闭此功能 |

### Rdbchecksum

|  |
| --- |
| rdbchecksum：在存储快照后，还可以让redis使用CRC64算法来进行数据校验，但是这样做会增加大约10%的性能消耗，如果希望获取到最大的性能提升，可以关闭此功能 |

### Dbfilename

dump.rdb

### dir

Config get dir 得到

## REPLICATION复制

## SECURITY安全

* 访问密码的查看、设置和取消

|  |
| --- |
|  |

## LIMITS限制

1. Maxclients

|  |
| --- |
| 设置redis同时可以与多少个客户端进行连接。默认情况下为10000个客户端。当你  无法设置进程文件句柄限制时，redis会设置为当前的文件句柄限制值减去32，因为redis会为自  身内部处理逻辑留一些句柄出来。如果达到了此限制，redis则会拒绝新的连接请求，并且向这  些连接请求方发出“max number of clients reached”以作回应。 |

1. Maxmemory

|  |
| --- |
| 设置redis可以使用的内存量。一旦到达内存使用上限，redis将会试图移除内部数据，移除规则可以通过maxmemory-policy来指定。如果redis无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”，  那么redis则会针对那些需要申请内存的指令返回错误信息，比如SET、LPUSH等。  但是对于无内存申请的指令，仍然会正常响应，比如GET等。如果你的redis是主redis（说明你的redis有从redis），那么在设置内存使用上限时，需要在系统中留出一些内存空间给同步队列缓存，只有在你设置的是“不移除”的情况下，才不用考虑这个因素 |

1. maxmemory-policy

|  |
| --- |
| （1）volatile-lru：使用LRU算法移除key，只对设置了过期时间的键  （2）allkeys-lru：使用LRU算法移除key  （3）volatile-random：在过期集合中移除随机的key，只对设置了过期时间的键  （4）allkeys-random：移除随机的key  （5）volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key  （6）noeviction：不进行移除。针对写操作，只是返回错误信息 |

* volatile-lru -> remove the key with an expire set using an LRU algorithm
* allkeys-lru -> remove any key according to the LRU algorithm
* volatile-random -> remove a random key with an expire set
* allkeys-random -> remove a random key, any key
* volatile-ttl -> remove the key with the nearest expire time (minor TTL)
* noeviction -> don't expire at all, just return an error on write operations

1. maxmemory-samples

|  |
| --- |
| 设置样本数量，LRU算法和最小TTL算法都并非是精确的算法，而是估算值，所以你可以设置样本的大小，  redis默认会检查这么多个key并选择其中LRU的那个 |

## APPEND ONLY MODE追加

### appendonly

默认 为no

|  |
| --- |
| aof与rdb文件同时存在，会读aof文件  aof文件损坏，可以用redis-check-aof --fix appendonly.aof命令修复 |

### appendfilename

### Appendfsync

|  |
| --- |
|  |

* always：同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好
* everysec：出厂默认推荐，异步操作，每秒记录 如果一秒内宕机，有数据丢失
* no

### no-appendfsync-on-rewrite：重写时是否可以运用Appendfsync，用默认no即可，保证数据安全性。

### auto-aof-rewrite-min-size：设置重写的基准值

### auto-aof-rewrite-percentage：设置重写的基准值

## 常见配置redis.conf介绍

|  |
| --- |
| **参数说明**  redis.conf 配置项说明如下：  1. Redis默认不是以守护进程的方式运行，可以通过该配置项修改，使用yes启用守护进程    daemonize no  2. 当Redis以守护进程方式运行时，Redis默认会把pid写入/var/run/redis.pid文件，可以通过pidfile指定    pidfile /var/run/redis.pid  3. 指定Redis监听端口，默认端口为6379，作者在自己的一篇博文中解释了为什么选用6379作为默认端口，因为6379在手机按键上MERZ对应的号码，而MERZ取自意大利歌女Alessia Merz的名字    port 6379  4. 绑定的主机地址    bind 127.0.0.1  5.当 客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能    timeout 300  6. 指定日志记录级别，Redis总共支持四个级别：debug、verbose、notice、warning，默认为verbose    loglevel verbose  7. 日志记录方式，默认为标准输出，如果配置Redis为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null    logfile stdout  8. 设置数据库的数量，默认数据库为0，可以使用SELECT <dbid>命令在连接上指定数据库id    databases 16  9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合    save <seconds> <changes>    Redis默认配置文件中提供了三个条件：    save 900 1    save 300 10    save 60 10000    分别表示900秒（15分钟）内有1个更改，300秒（5分钟）内有10个更改以及60秒内有10000个更改。    10. 指定存储至本地数据库时是否压缩数据，默认为yes，Redis采用LZF压缩，如果为了节省CPU时间，可以关闭该选项，但会导致数据库文件变的巨大    rdbcompression yes  11. 指定本地数据库文件名，默认值为dump.rdb    dbfilename dump.rdb  12. 指定本地数据库存放目录    dir ./  13. 设置当本机为slav服务时，设置master服务的IP地址及端口，在Redis启动时，它会自动从master进行数据同步    slaveof <masterip> <masterport>  14. 当master服务设置了密码保护时，slav服务连接master的密码    masterauth <master-password>  15. 设置Redis连接密码，如果配置了连接密码，客户端在连接Redis时需要通过AUTH <password>命令提供密码，默认关闭    requirepass foobared  16. 设置同一时间最大客户端连接数，默认无限制，Redis可以同时打开的客户端连接数为Redis进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis会关闭新的连接并向客户端返回max number of clients reached错误信息    maxclients 128  17. 指定Redis最大内存限制，Redis在启动时会把数据加载到内存中，达到最大内存后，Redis会先尝试清除已到期或即将到期的Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis新的vm机制，会把Key存放内存，Value会存放在swap区    maxmemory <bytes>  18. 指定是否在每次更新操作后进行日志记录，Redis在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis本身同步数据文件是按上面save条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为no    appendonly no  19. 指定更新日志文件名，默认为appendonly.aof     appendfilename appendonly.aof  20. 指定更新日志条件，共有3个可选值：    no：表示等操作系统进行数据缓存同步到磁盘（快）    always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）    everysec：表示每秒同步一次（折衷，默认值）    appendfsync everysec    21. 指定是否启用虚拟内存机制，默认值为no，简单的介绍一下，VM机制将数据分页存放，由Redis将访问量较少的页即冷数据swap到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析Redis的VM机制）     vm-enabled no  22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个Redis实例共享     vm-swap-file /tmp/redis.swap  23. 将所有大于vm-max-memory的数据存入虚拟内存,无论vm-max-memory设置多小,所有索引数据都是内存存储的(Redis的索引数据 就是keys),也就是说,当vm-max-memory设置为0的时候,其实是所有value都存在于磁盘。默认值为0     vm-max-memory 0  24. Redis swap文件分成了很多的page，一个对象可以保存在多个page上面，但一个page上不能被多个对象共享，vm-page-size是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page大小最好设置为32或者64bytes；如果存储很大大对象，则可以使用更大的page，如果不 确定，就使用默认值     vm-page-size 32  25. 设置swap文件中的page数量，由于页表（一种表示页面空闲或使用的bitmap）是在放在内存中的，，在磁盘上每8个pages将消耗1byte的内存。     vm-pages 134217728  26. 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4     vm-max-threads 4  27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启    glueoutputbuf yes  28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法    hash-max-zipmap-entries 64    hash-max-zipmap-value 512  29. 指定是否激活重置哈希，默认为开启（后面在介绍Redis的哈希算法时具体介绍）    activerehashing yes  30. 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件    include /path/to/local.conf |

# redis的持久化

## 总体介绍

* 官网介绍

|  |
| --- |
|  |

## RDB（Redis DataBase）

### 官网介绍

|  |
| --- |
|  |

### 是什么：

* 在指定的时间间隔内将内存中的数据集快照写入磁盘，也就是行话讲的Snapshot快照，它恢复时是将快照文件直接读到内存里
* Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。RDB的缺点是最后一次持久化后的数据可能丢失。

### Fork

* fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

### rdb 保存的是dump.rdb文件

### 配置位置

|  |
| --- |
|  |

### 如何触发RDB快照

#### 配置文件中默认的快照配置

|  |
| --- |
|  |

冷拷贝后重新使用

可以cp dump.rdb dump\_new.rdb

#### 命令save或者是bgsave

|  |
| --- |
|  |

* Save：save时只管保存，其它不管，全部阻塞
* BGSAVE：Redis会在后台异步进行快照操作，快照同时还可以响应客户端请求。可以通过lastsave命令获取最后一次成功执行快照的时间

#### 执行flushall命令，也会产生dump.rdb文件，但里面是空的，无意义

### 如何恢复

* 将备份文件 (dump.rdb) 移动到 redis 安装目录并启动服务即可
* CONFIG GET dir获取目录

### 优势

* 适合大规模的数据恢复
* 对数据完整性和一致性要求不高

### 劣势

* 在一定间隔时间做一次备份，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改
* fork的时候，内存中的数据被克隆了一份，大致2倍的膨胀性需要考虑

### 如何停止

* 动态所有停止RDB保存规则的方法：redis-cli config set save ""

注：一般不会这么做

### 小总结

|  |
| --- |
|  |

## AOF（Append Only File）

### 官网介绍

|  |
| --- |
|  |

### 是什么：

以日志的形式来记录每个写操作，将Redis执行过的所有写指令记录下来(读操作不记录)，只许追加文件但不可以改写文件，redis启动之初会读取该文件重新构建数据，换言之，redis重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

### Aof保存的是appendonly.aof文件

### 配置位置

|  |
| --- |
|  |

### AOF启动/修复/恢复

#### 正常恢复

* 启动：设置Yes 修改默认的appendonly no，改为yes
* 将有数据的aof文件复制一份保存到对应目录(config get dir)
* 恢复：重启redis然后重新加载

#### 异常恢复

* 启动：设置Yes 修改默认的appendonly no，改为yes
* 备份被写坏的AOF文件
* 修复： redis-check-aof --fix进行修复

|  |
| --- |
|  |

* 恢复：重启redis然后重新加载

### Rewrite

|  |
| --- |
|  |

#### 是什么：

AOF采用文件追加方式，文件会越来越大为避免出现此种情况，新增了重写机制,当AOF文件的大小超过所设定的阈值时，Redis就会启动AOF文件的内容压缩，只保留可以恢复数据的最小指令集.可以使用命令bgrewriteaof

#### 重写原理

AOF文件持续增长而过大时，会fork出一条新进程来将文件重写(也是先写临时文件最后再rename)，遍历新进程的内存中数据，每条记录有一条的Set语句。重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件，这点和快照有点类似

#### 触发机制

Redis会记录上次重写时的AOF大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍且文件大于64M时触发

注：生产上3g是起步

### 优势

* 每修改同步：appendfsync always 同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好
* 每秒同步：appendfsync everysec 异步操作，每秒记录 如果一秒内宕机，有数据丢失
* 不同步：appendfsync no 从不同步

### 劣势

* 相同数据集的数据而言aof文件要远大于rdb文件，恢复速度慢于rdb
* aof运行效率要慢于rdb,每秒同步策略效率较好，不同步效率和rdb相同

### 小总结

|  |
| --- |
|  |

## 总结(Which one)

1. 官网建议

|  |
| --- |
|  |

1. RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储
2. AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾.
3. Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大
4. 只做缓存：如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式.
5. 同时开启两种持久化方式

* 在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整.
* RDB的数据不实时，同时使用两者时服务器重启也只会找AOF文件。那要不要只使用AOF呢？作者建议不要，因为RDB更适合用于备份数据库(AOF在不断变化不好备份)，快速重启，而且不会有AOF可能潜在的bug，留着作为一个万一的手段。

1. 性能建议

|  |
| --- |
| 因为RDB文件只用作后备用途，建议只在Slave上持久化RDB文件，而且只要15分钟备份一次就够了，只保留save 900 1这条规则。  如果Enalbe AOF，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只load自己的AOF文件就可以了。代价一是带来了持续的IO，二是AOF rewrite的最后将rewrite过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少AOF rewrite的频率，AOF重写的基础大小默认值64M太小了，可以设到5G以上。默认超过原大小100%大小时重写可以改到适当的数值。  如果不Enable AOF ，仅靠Master-Slave Replication 实现高可用性也可以。能省掉一大笔IO也减少了rewrite时带来的系统波动。代价是如果Master/Slave同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个Master/Slave中的RDB文件，载入较新的那个。新浪微博就选用了这种架构 |

# Redis的事务

## 是什么

可以一次执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按顺序地串行化执行而不会被其它命令插入，不许加塞

官网：

|  |
| --- |
|  |

## 能干嘛

一个队列中，一次性、顺序性、排他性的执行一系列命令

## 怎么玩

|  |
| --- |
|  |

### 常用命令

|  |
| --- |
|  |

### case1：正常执行

|  |
| --- |
|  |

### Case2：放弃事务

|  |
| --- |
|  |

### Case3：全体连坐

|  |
| --- |
|  |

### Case4：冤头债主

**redis部分支持事务**

|  |
| --- |
|  |

### Case5：watch监控

#### 悲观锁/乐观锁/CAS(Check And Set)

* 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 |

* 乐观锁

|  |
| --- |
| 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，  乐观锁策略:提交版本必须大于记录当前版本才能执行更新 |

* CAS

#### 初始化信用卡可用余额和欠额

|  |
| --- |
|  |

#### 无加塞篡改，先监控再开启multi，保证两笔金额变动在同一个事务内

|  |
| --- |
|  |

#### 有加塞篡改

|  |
| --- |
|  |

监控了key，如果key被修改了，后面一个事务的执行失效

#### Unwatch

|  |
| --- |
|  |

#### 一旦执行了exec之前加的监控锁都会被取消掉了

#### 小结

* Watch指令，类似乐观锁，事务提交时，如果Key的值已被别的客户端改变，比如某个list已被别的客户端push/pop过了，整个事务队列都不会被执行
* 通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，EXEC命令执行的事务都将被放弃，同时返回Nullmulti-bulk应答以通知调用者事务执行失败

## 3阶段

1. 开启：以MULTI开始一个事务
2. 入队：将多个命令入队到事务中，接到这些命令并不会立即执行，而是放到等待执行的事务队列里面
3. 执行：由EXEC命令触发事务

## 3特性

1. 单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
2. 没有隔离级别的概念：队列中的命令没有提交之前都不会实际的被执行，因为事务提交前任何指令都不会被实际执行，也就不存在”事务内的查询要看到事务里的更新，在事务外查询不能看到”这个让人万分头痛的问题
3. 不保证原子性：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚

# Redis的发布订阅

## 是什么

### 进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

### 订阅/发布消息图

|  |
| --- |
|  |

## 命令

|  |
| --- |
|  |

## 案列

|  |
| --- |
| 先订阅后发布后才能收到消息，  1 可以一次性订阅多个，SUBSCRIBE c1 c2 c3  2 消息发布，PUBLISH c2 hello-redis  =============================================================  3 订阅多个，通配符\*， PSUBSCRIBE new\*  4 收取消息， PUBLISH new1 redis2015 |

# Redis的复制(Master/Slave)

## 是什么

* 官网

|  |
| --- |
|  |

* 行话：也就是我们所说的主从复制，主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主

## 能干嘛

1. 读写分离
2. 容灾恢复

## 怎么玩

### 配从(库)不配主(库)

### 从库配置：slaveof 主库IP 主库端口

* 每次与master断开之后，都需要重新连接，除非你配置进redis.conf文件
* info replication

|  |
| --- |
|  |

### 修改配置文件细节操作

#### 拷贝多个redis.conf文件

#### 开启daemonize yes

#### pid文件名字

|  |
| --- |
|  |

#### 指定端口log文件名字

|  |
| --- |
|  |

#### dump.rdb名字

|  |
| --- |
|  |
|  |

### 常用3招

#### 一主二仆

##### Init

##### 一个Master两个Slave

##### 日志查看

* 主机日志

|  |
| --- |
|  |

* 备机日志

|  |
| --- |
|  |

* info replication

|  |
| --- |
|  |

##### 主从问题演示

|  |
| --- |
| 1 切入点问题？slave1、slave2是从头开始复制还是从切入点开始复制?比如从k4进来，那之前的123是否也可以复制  2 从机是否可以写？set可否？  3 主机shutdown后情况如何？从机是上位还是原地待命  4 主机又回来了后，主机新增记录，从机还能否顺利复制？  5 其中一台从机down后情况如何？依照原有它能跟上大部队吗？      从头备份到尾    从机可以set？ No    主机down掉：    主机又活了：（从机原地待命）    从机down掉： （再活不是slave，除非写进配置文件） |

#### 薪火相传

* 上一个Slave可以是下一个slave的Master，Slave同样可以接收其他slaves的连接和同步请求，那么该slave作为了链条中下一个的master,可以有效减轻master的写压力
* 中途变更转向:会清除之前的数据，重新建立拷贝最新的
* slaveof 新主库IP 新主库端口

|  |
| --- |
|  |

#### 反客为主（手动）

SLAVEOF no one

使当前数据库停止与其他数据库的同步，转成主数据库

|  |
| --- |
|  |

## 复制原理

1. slave启动成功连接到master后会发送一个sync命令
2. Master接到命令启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，master将传送整个数据文件到slave,以完成一次完全同步
3. 全量复制：而slave服务在接收到数据库文件数据后，将其存盘并加载到内存中。
4. 增量复制：Master继续将新的所有收集到的修改命令依次传给slave,完成同步
5. 但是只要是重新连接master,一次完全同步（全量复制)将被自动执行

## 哨兵模式(sentinel)

### 是什么

反客为主的自动版，能够后台监控主机是否故障，如果故障了根据投票数自动将从库转换为主库

### 怎么玩(使用步骤)

#### 调整结构，6379带着80、81

#### 自定义的/myredis目录下新建sentinel.conf文件，名字绝不能错

#### 配置哨兵,填写内容

* sentinel monitor 被监控数据库名字(自己起名字) 127.0.0.1 6379 1

|  |
| --- |
|  |

* 上面最后一个数字1，表示主机挂掉后salve投票看让谁接替成为主机，得票数多少后成为主机

#### 启动哨兵

* redis-sentinel /myredis/sentinel.conf

|  |
| --- |
|  |

* 上述目录依照各自的实际情况配置，可能目录不同

#### 正常主从演示

#### 原有的master挂了

|  |
| --- |
|  |

#### 投票新选

|  |
| --- |
|  |

#### 重新主从继续开工,info replication查查看

|  |
| --- |
|  |

#### 问题：如果之前的master重启回来，会不会双master冲突？

|  |
| --- |
| 刚启动79，还为master是因为哨兵还没监控到。    等一会，就会看到，79变为slave |

### 一组sentinel能同时监控多个Master

## 复制的缺点

**复制延时**

|  |
| --- |
| 由于所有的写操作都是先在Master上操作，然后同步更新到Slave上，所以从Master同步到Slave机器有一定的延迟，当系统很繁忙的时候，延迟问题会更加严重，Slave机器数量的增加也会使这个问题更加严重。 |

# Redis的Java客户端Jedis

## Jedis常用操作

### 测试连通性

|  |
| --- |
| **public** **class** Demo01 {  **public** **static** **void** main(String[] args) {      //连接本地的 Redis 服务      Jedis jedis = **new** Jedis("127.0.0.1",6379);      //查看服务是否运行，打出pong表示OK      System.*out*.println("connection is OK==========>: "+jedis.ping());    }  } |

### 5+1

|  |
| --- |
| **package** com.atguigu.redis.test;  **import** java.util.\*;  **import** redis.clients.jedis.Jedis;  **public** **class** Test02  {  **public** **static** **void** main(String[] args)    {       Jedis jedis = **new** Jedis("127.0.0.1",6379);       //key       Set<String> keys = jedis.keys("\*");  **for** (Iterator iterator = keys.iterator(); iterator.hasNext();) {         String key = (String) iterator.next();         System.*out*.println(key);       }       System.*out*.println("jedis.exists====>"+jedis.exists("k2"));       System.*out*.println(jedis.ttl("k1"));       //String       //jedis.append("k1","myreids");       System.*out*.println(jedis.get("k1"));       jedis.set("k4","k4\_redis");       System.*out*.println("----------------------------------------");       jedis.mset("str1","v1","str2","v2","str3","v3");       System.*out*.println(jedis.mget("str1","str2","str3"));       //list       System.*out*.println("----------------------------------------");       //jedis.lpush("mylist","v1","v2","v3","v4","v5");       List<String> list = jedis.lrange("mylist",0,-1);  **for** (String element : list) {         System.*out*.println(element);       }       //set       jedis.sadd("orders","jd001");       jedis.sadd("orders","jd002");       jedis.sadd("orders","jd003");       Set<String> set1 = jedis.smembers("orders");  **for** (Iterator iterator = set1.iterator(); iterator.hasNext();) {         String string = (String) iterator.next();         System.*out*.println(string);       }       jedis.srem("orders","jd002");       System.*out*.println(jedis.smembers("orders").size());       //hash       jedis.hset("hash1","userName","lisi");       System.*out*.println(jedis.hget("hash1","userName"));       Map<String,String> map = **new** HashMap<String,String>();       map.put("telphone","13811814763");       map.put("address","atguigu");       map.put("email","abc@163.com");       jedis.hmset("hash2",map);       List<String> result = jedis.hmget("hash2", "telphone","email");  **for** (String element : result) {         System.*out*.println(element);       }       //zset       jedis.zadd("zset01",60d,"v1");       jedis.zadd("zset01",70d,"v2");       jedis.zadd("zset01",80d,"v3");       jedis.zadd("zset01",90d,"v4");         Set<String> s1 = jedis.zrange("zset01",0,-1);  **for** (Iterator iterator = s1.iterator(); iterator.hasNext();) {         String string = (String) iterator.next();         System.*out*.println(string);       }      }  } |

* 一个key
* 五大数据类型

### 事务提交

#### 日常

|  |
| --- |
| **package** com.atguigu.redis.test;  **import** redis.clients.jedis.Jedis;  **import** redis.clients.jedis.Response;  **import** redis.clients.jedis.Transaction;  **public** **class** Test03  {  **public** **static** **void** main(String[] args)    {       Jedis jedis = **new** Jedis("127.0.0.1",6379);         //监控key，如果该动了事务就被放弃       /\*3       jedis.watch("serialNum");       jedis.set("serialNum","s#####################");       jedis.unwatch();\*/         Transaction transaction = jedis.multi();//被当作一个命令进行执行       Response<String> response = transaction.get("serialNum");       transaction.set("serialNum","s002");       response = transaction.get("serialNum");       transaction.lpush("list3","a");       transaction.lpush("list3","b");       transaction.lpush("list3","c");         transaction.exec();       //2 transaction.discard();       System.*out*.println("serialNum\*\*\*\*\*\*\*\*\*\*\*"+response.get());      }  } |

#### 加锁

|  |
| --- |
| **public** **class** TestTransaction {  **public** **boolean** transMethod() {       Jedis jedis = **new** Jedis("127.0.0.1", 6379);  **int** balance;// 可用余额  **int** debt;// 欠额  **int** amtToSubtract = 10;// 实刷额度       jedis.watch("balance");       //jedis.set("balance","5");//此句不该出现，讲课方便。模拟其他程序已经修改了该条目       balance = Integer.*parseInt*(jedis.get("balance"));  **if** (balance < amtToSubtract) {         jedis.unwatch();         System.*out*.println("modify");  **return** **false**;       } **else** {         System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*transaction");         Transaction transaction = jedis.multi();         transaction.decrBy("balance", amtToSubtract);         transaction.incrBy("debt", amtToSubtract);         transaction.exec();         balance = Integer.*parseInt*(jedis.get("balance"));         debt = Integer.*parseInt*(jedis.get("debt"));         System.*out*.println("\*\*\*\*\*\*\*" + balance);         System.*out*.println("\*\*\*\*\*\*\*" + debt);  **return** **true**;       }    }    /\*\*     \* 通俗点讲，watch命令就是标记一个键，如果标记了一个键， 在提交事务前如果该键被别人修改过，那事务就会失败，这种情况通常可以在程序中     \* 重新再尝试一次。     \* 首先标记了键balance，然后检查余额是否足够，不足就取消标记，并不做扣减； 足够的话，就启动事务进行更新操作，     \* 如果在此期间键balance被其它人修改， 那在提交事务（执行exec）时就会报错， 程序中通常可以捕获这类错误再重新执行一次，直到成功。     \*/  **public** **static** **void** main(String[] args) {       TestTransaction test = **new** TestTransaction();  **boolean** retValue = test.transMethod();       System.*out*.println("main retValue-------: " + retValue);    }  } |

### 主从复制

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** InterruptedException    {       Jedis jedis\_M = **new** Jedis("127.0.0.1",6379);       Jedis jedis\_S = **new** Jedis("127.0.0.1",6380);         jedis\_S.slaveof("127.0.0.1",6379);         jedis\_M.set("k6","v6");       Thread.*sleep*(500);       System.*out*.println(jedis\_S.get("k6"));    } |

* 6379,6380启动，先各自先独立
* 主写
* 从读

## JedisPool

### 获取Jedis实例需要从JedisPool中获取

### 用完Jed案例见代码is实例需要返还给JedisPool

### 如果Jedis在使用过程中出错，则也需要还给JedisPool

### 案例见代码

* JedisPoolUtil

|  |
| --- |
| **package** com.atguigu.redis.test;  **import** redis.clients.jedis.Jedis;  **import** redis.clients.jedis.JedisPool;  **import** redis.clients.jedis.JedisPoolConfig;  **public** **class** JedisPoolUtil {    **private** **static** **volatile** JedisPool *jedisPool* = **null**;//被volatile修饰的变量不会被本地线程缓存，对该变量的读写都是直接操作共享内存。    **private** JedisPoolUtil() {}    **public** **static** JedisPool getJedisPoolInstance()   {  **if**(**null** == *jedisPool*)      {  **synchronized** (JedisPoolUtil.**class**)        {  **if**(**null** == *jedisPool*)           {             JedisPoolConfig poolConfig = **new** JedisPoolConfig();             poolConfig.setMaxActive(1000);             poolConfig.setMaxIdle(32);             poolConfig.setMaxWait(100\*1000);             poolConfig.setTestOnBorrow(**true**);    *jedisPool* = **new** JedisPool(poolConfig,"127.0.0.1");           }        }      }  **return** *jedisPool*;   }    **public** **static** **void** release(JedisPool jedisPool,Jedis jedis)   {  **if**(**null** != jedis)      {        jedisPool.returnResourceObject(jedis);      }   }  } |

* Demo5

jedisPool.getResource();

|  |
| --- |
| **package** com.atguigu.redis.test;  **import** redis.clients.jedis.Jedis;  **import** redis.clients.jedis.JedisPool;  **public** **class** Test01 {  **public** **static** **void** main(String[] args) {       JedisPool jedisPool = JedisPoolUtil.*getJedisPoolInstance*();       Jedis jedis = **null**;    **try**       {         jedis = jedisPool.getResource();         jedis.set("k18","v183");         } **catch** (Exception e) {         e.printStackTrace();       }**finally**{         JedisPoolUtil.*release*(jedisPool, jedis);       }    }  } |

### 配置总结all

|  |
| --- |
| JedisPool的配置参数大部分是由JedisPoolConfig的对应项来赋值的。  maxActive：控制一个pool可分配多少个jedis实例，通过pool.getResource()来获取；如果赋值为-1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhausted。  maxIdle：控制一个pool最多有多少个状态为idle(空闲)的jedis实例；  whenExhaustedAction：表示当pool中的jedis实例都被allocated完时，pool要采取的操作；默认有三种。   WHEN\_EXHAUSTED\_FAIL --> 表示无jedis实例时，直接抛出NoSuchElementException；   WHEN\_EXHAUSTED\_BLOCK --> 则表示阻塞住，或者达到maxWait时抛出JedisConnectionException；   WHEN\_EXHAUSTED\_GROW --> 则表示新建一个jedis实例，也就说设置的maxActive无用；  maxWait：表示当borrow一个jedis实例时，最大的等待时间，如果超过等待时间，则直接抛JedisConnectionException；  testOnBorrow：获得一个jedis实例的时候是否检查连接可用性（ping()）；如果为true，则得到的jedis实例均是可用的；  testOnReturn：return 一个jedis实例给pool时，是否检查连接可用性（ping()）；  testWhileIdle：如果为true，表示有一个idle object evitor线程对idle object进行扫描，如果validate失败，此object会被从pool中drop掉；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；  timeBetweenEvictionRunsMillis：表示idle object evitor两次扫描之间要sleep的毫秒数；  numTestsPerEvictionRun：表示idle object evitor每次扫描的最多的对象数；  minEvictableIdleTimeMillis：表示一个对象至少停留在idle状态的最短时间，然后才能被idle object evitor扫描并驱逐；这一项只有在timeBetweenEvictionRunsMillis大于0时才有意义；  softMinEvictableIdleTimeMillis：在minEvictableIdleTimeMillis基础上，加入了至少minIdle个对象已经在pool里面了。如果为-1，evicted不会根据idle time驱逐任何对象。如果minEvictableIdleTimeMillis>0，则此项设置无意义，且只有在timeBetweenEvictionRunsMillis大于0时才有意义；  lifo：borrowObject返回对象时，是采用DEFAULT\_LIFO（last in first out，即类似cache的最频繁使用队列），如果为False，则表示FIFO队列；  ==================================================================================================================  其中JedisPoolConfig对一些参数的默认设置如下：  testWhileIdle=true  minEvictableIdleTimeMills=60000  timeBetweenEvictionRunsMillis=30000  numTestsPerEvictionRun=-1 |