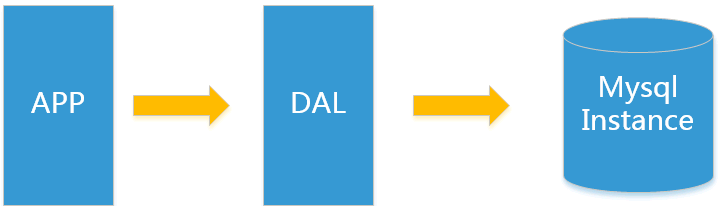
# Redis从浅入深学习笔记

## 一、NoSQL概述

***互联网时代背景下大机遇，为什么用NoSql？***

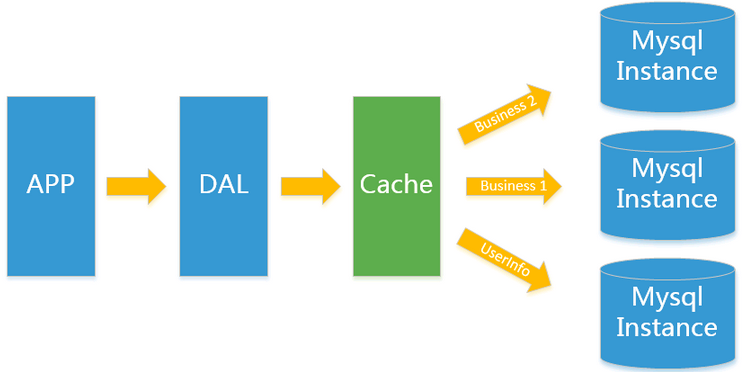
1、单机MySQL的美好年代。

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）数据量的总大小过大一个机器放不下；（2）数据的索引（B+ Tree）一个机器的内存放不下；（3）访问量(读写混合)一个实例不能承受。

2、Memcached(缓存)+MySQL+垂直拆分

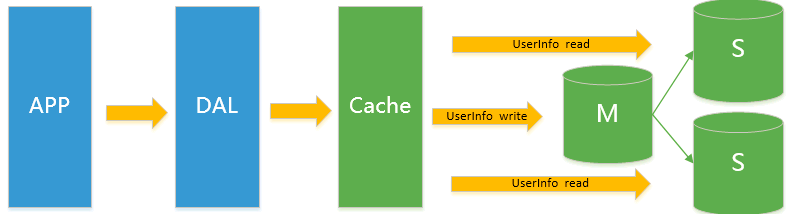
后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大的时候，多台web机器通过文件缓存不能共享，大量的小文件缓存也带了了比较高的IO压力。在这个时候，*Memcached就自然的成为一个非常时尚的技术产品*。



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

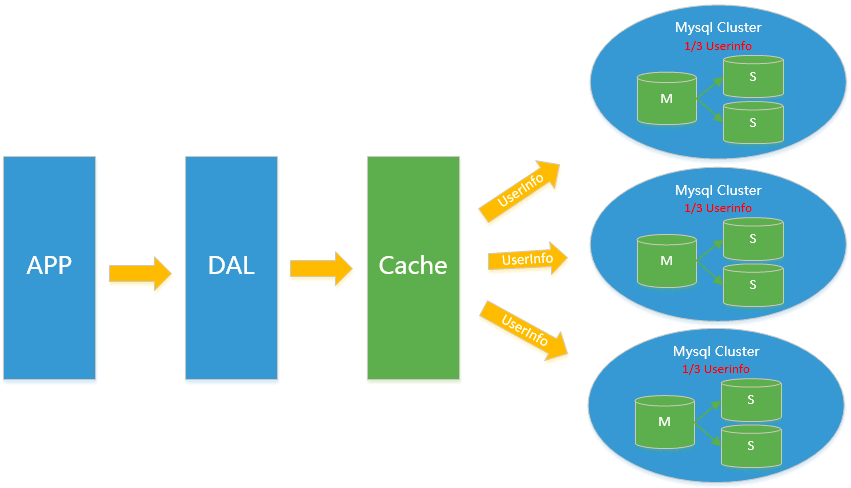
由于数据库的写入压力增加，*Memcached只能缓解数据库的读取压力*。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了。



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。

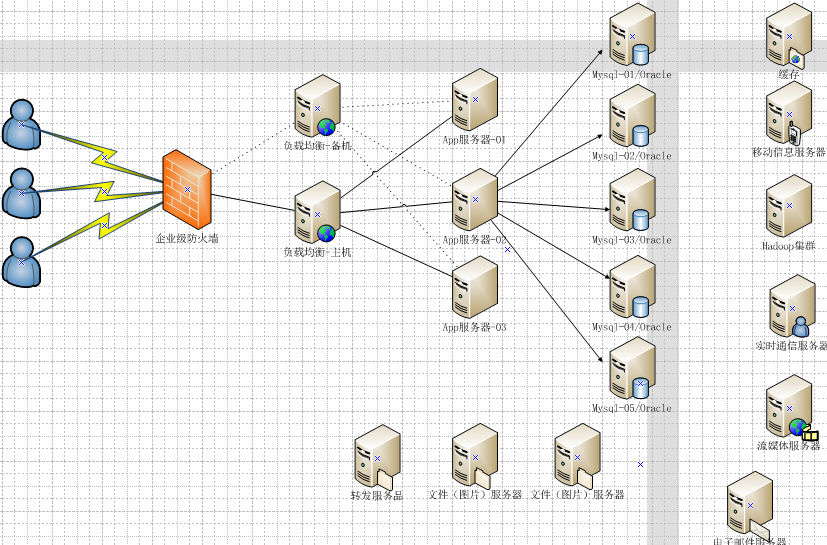
同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。



5、MySQL的扩展性瓶颈

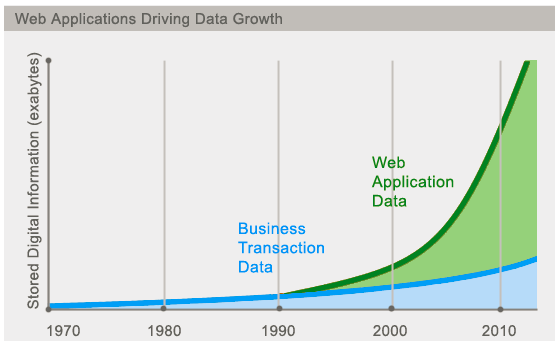
MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小，如果能把这些数据从MySQL省去，MySQL将变得非常的小。关系数据库很强大，但是它并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，表结构更改困难，正是当前使用MySQL的开发人员面临的问题。

6、今天是什么样子？？



7、为什么用NoSQL

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。



## 二、什么是NoSQL

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。

## 三、NoSQL能干什么

1、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。

2、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。

3、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。

4、传统RDBMS VS NOSQL

|  |  |
| --- | --- |
| RDBMS | NoSQL |
| 数据和关系都存储在单独的表中 | 代表着不仅仅是SQL |
| 数据操纵语言，数据定义语言 | 没有声明性查询语言 |
| 数据操纵语言，数据定义语言 | 没有声明性查询语言 |
| 基础事务 | 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 |

## 四、Redis、Memcached、MongoDB

Mongodb是文档型的非关系型数据库，其优势在于查询功能比较强大，能存储海量数据，缺点是比较消耗内存，Mongodb和Memcached不是一个范畴内的东西。

Memcached 是一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载。它通过在内存中缓存数据和对象来减少读取数据库的次数，从而提供动态、数据库驱动网站的速度。Memcached 的分布式不是在服务器端实现的，而是在客户端应用中实现的，即通过内置算法制定目标数据的节点，Memcached 的分布式是基于客户端的Key的hash来做均衡，是个伪分布式的系统。

Redis是一个key-value存储系统。和 Memcached 类似，它支持存储的value类型相对更多，包括string(字符串)、 list(链表)、set(集合)和zset(有序集合)。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，Redis支持各种不同方式的排序。与 Memcached 一样，为了保证效率，数据都是缓存在内存中。区别的是Redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件。

Memcached 和 Redis它们都是内存型数据库，数据保存在内存中，通过tcp直接存取，优势是速度快，并发高。若希望一专多能，支持丰富的数据类型，则Redis更合适，若仅仅用于高速缓存，则Memcache比较高效。

1. 性能，TPS方面Redis和Memcache差不多，要大于Mongodb，Redis 支持高并发量，官方宣传支持10万级别的并发读写。
2. 操作的便利性，Memcached 数据结构单一（key-value），Redis 支持的数据类型更多，数据操作方面，Redis更好一些，也可以在服务器端直接对数据进行丰富的操作,这样可以减少网络IO次数和数据体积，同时还提供String、Hash、List、Set和Sorted Set等数据结构的存储。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，可以限定内存使用大小，当数据超过阈值，则通过类似LRU的算法把内存中的最不常用数据保存到硬盘的页面文件中；可以对 key-value 设置过期时间（类似 Memcached）；Redis一个value最大支持512MB。Memcached 可以修改最大可用内存,采用 LRU 算法；Memcached单个key-value大小有限，一个value最大只支持1MB。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. 可靠性（持久化），Redis 具有持久化机制，可以定期将内存中的数据持久化到硬盘上，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。Memcached 不支持数据持久化，断电或重启后数据消失，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。
5. 数据一致性（事务支持），Memcached 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。
6. 数据分析，MongoDB 内置了数据分析的功能(mapreduce)，其他不支持。
7. 应用场景，Redis：适用于对读写效率要求都很高，数据处理业务复杂和对安全性要求较高的系统；数据量较小的更性能操作和运算，Redis 只使用单核，而 Memcached 可以使用多核，所以平均每一个核上Redis在存储小数据时比Memcached性能更高。Memcached：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。

由于Redis只使用单核，而Memcached可以使用多核，所以二者比较起来，平均每一个核上，Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis。虽然Redis最近也在存储大数据的性能上进行优化，但是比起Memcached，还是稍有逊色。结论是无论你使用哪一个，每秒处理请求的次数都不会成为瓶颈。

在内存使用效率上，如果使用简单的key-value存储，Memcached的内存利用率更高。而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached。当然，这和你的应用场景和数据特性有关。

如果你对数据持久化和数据同步有所要求，那么推荐你选择Redis。因为这两个特性Memcached都不具备。即使你只是希望在升级或者重启系统后缓存数据不会丢失，选择Redis也是明智的。

Redis相比Memcached来说，拥有更多的数据结构，并支持更丰富的数据操作。通常在Memcached里，你需要将数据拿到客户端来进行类似的修改再set回去。这大大增加了网络IO的次数和数据体积。在Redis中，这些复杂的操作通常和一般的GET/SET一样高效。所以，如果你需要缓存能够支持更复杂的结构和操作，那么Redis会是不错的选择。

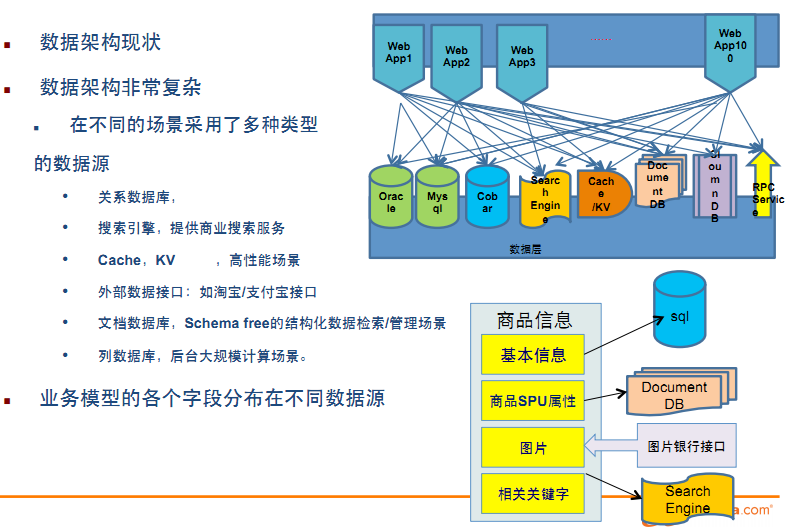
## 五、3V + 3高

大数据时代系统的三个痛点：海量（Volume）、多样（Variety）、实时（Velocity）！

互联网需求3高：高并发、高性能、高可扩展！

## 六、NoSQL的经典应用

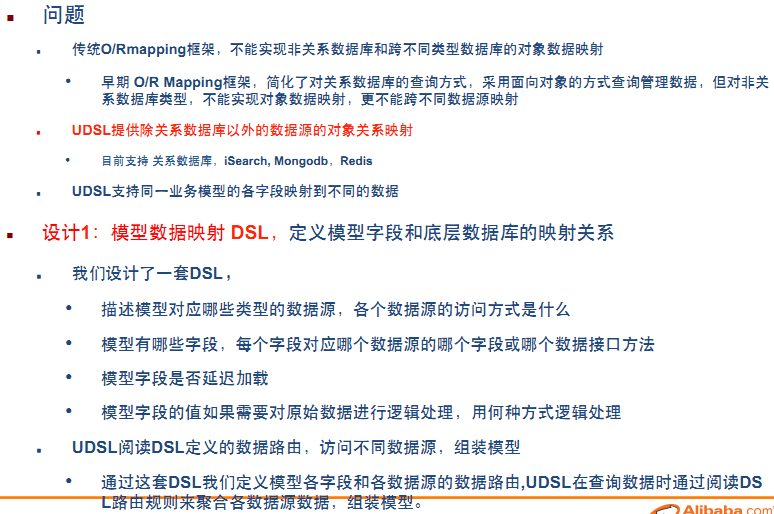
当下的应用是SQL和NoSQL结合一起使用，出现多数据源和多数据类型。



|  |  |
| --- | --- |
| 信息类型 | 存储方式 |
| 商品基本信息 | 名称、价格，出厂日期，生产厂商等趋冷、不变的信息存放在关系型数据库中。内部去IOE拥抱云计算、分布式 |
| 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，存储文档数据库MongDB中提高查询效率 |
| 商品的图片 | 分布式的文件系统中，淘宝的TFS，Google的GFS及Hadoop的HDFS |
| 商品的关键字 | 淘宝内用的搜索引擎使用， 基于ISearch构建站内搜索 |
| 商品的波段性的热点高频信息 | 例如：情人节的玫瑰、巧克力，使用内存数据库Tair、Redis、Memcache存储 |
| 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，如支付宝 |

大型互联网应用的难点：1、数据类型多样性；2、数据源多样性和变化重构；3、数据源改造而数据服务平台不需要大面积重构。其解决方案是：*EAI（企业应用集成）和统一数据平台服务层*。阿里、淘宝实现了统一数据服务层UDSL，在网站应用集群和底层数据源之间构建一层代理，统一数据层，主要工作为：

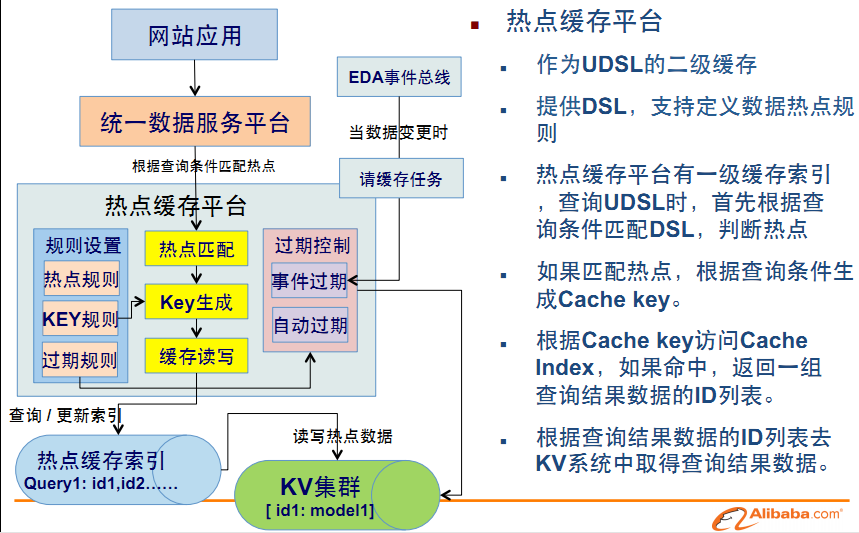
（1）、模型数据映射，实现业务模型各属性与底层不同类型数据源的模型映射，目前支持ISearch，Redis，MongoDB。



（2）、统一查询和更新API，提供了基于业务模型的统一查询和更新API，简化网站应用跨不同数据源的开发模式。



（3）、热点缓存



（4）、其他优化策略：字段延迟加载按需返回，异步并行查询数据（异步并行加载模型中来自不同数据源的字段），并发保护（拒绝访问频率过高的主机IP或IP段），过频高危查询（如会导致数据库崩溃的全表扫描）。

## 七、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：

1. 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)；
2. NoSQL如何设计？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。

{

"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"

}

}],

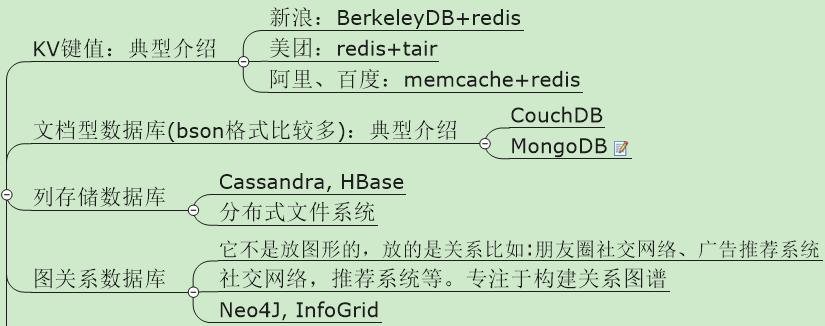
}]

}

}

采用传统数据库进行聚合操作是需要连接查询的，而*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，并且分布式事务支持不了太多的并发*，而采用BSON模型设计出来的结果其查询不需要连接大大提高性能。

## 八、NoSQL数据库的四大类



四大数据库种类对应四种聚合模型：KV键值对、BSON、列族、图形

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 分类 | 举例 | 典型场景 | 数据模型 | 优点 | 缺点 |
| 键值对 | Redis  Memcached | 内容缓存，用于处理大量数据频繁访问 | K-V通常用哈希表实现 | 查询速度快 | 数据无结构化，通常被当做字符串或二进制 |
| 列存储数据库 | Casssandra  HBase | 分布式文件系统 | 按列族存储，同一列数据在一起 | 查询速度快，扩展性强 | 功能相对局限 |
| 文档数据库 | MongoDB  CouchDB  Elasticsearch | web应用，与K-V类似，Value是结构化的，数据库能够了解Value内容 | K-V，V是结构化数据 | 数据结构要求不严格，表结构可变，无需提前自定义表结构 | 查询性能不高，而且缺乏统一的查询语法 |
| 图形数据库 | Neo4J  InfoGrid | 社交网络，推荐系统等，专注于构建关系图谱 | 图结构 | 利用图结构相关算法 | 需要整个图做计算才能得出需要信息，不太好做分布式集群方案 |

## 九、分布式数据库中的CAP、BASE理论

**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。

**分布式系统**是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 | 集群 |
| 不同的多台服务器上面部署不同的服务模块，它们之间通过Rpc/Rmi之间通信和调用，对外提供服务和组内协作。 | 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。 |

分布式一致性的提出

在分布式系统中要解决的一个重要问题就是数据的复制。在我们的日常开发经验中，相信很多开发人员都遇到过这样的问题：假设客户端C1将系统中的一个值K由V1更新为V2，但客户端C2无法立即读取到K的最新值，需要在一段时间之后才能读取到。这很正常，因为数据库复制之间存在延时。

分布式系统对于数据的复制需求一般都来自于以下两个原因：

1、为了增加系统的可用性，以防止单点故障引起的系统不可用

2、提高系统的整体性能，通过负载均衡技术，能够让分布在不同地方的数据副本都能够为用户提供服务

数据复制在可用性和性能方面给分布式系统带来的巨大好处是不言而喻的，然而数据复制所带来的一致性挑战，也是每一个系统研发人员不得不面对的。

所谓***分布一致性问题***是指在分布式环境中引入数据复制机制之后，不同数据节点之间可能出现的，并无法依靠计算机应用程序自身解决的数据不一致的情况。简单讲，数据一致性就是指在对一个副本数据进行更新的时候，必须确保也能够更新其他的副本，否则不同副本之间的数据将不一致。

那么如何解决这个问题？一种思路是"既然是由于延时动作引起的问题，那我可以将写入的动作阻塞，直到数据复制完成后，才完成写入动作"。没错，这似乎能解决问题，而且有一些系统的架构也确实直接使用了这个思路。但这个思路在解决一致性问题的同时，又带来了新的问题：写入的性能。如果你的应用场景有非常多的写请求，那么使用这个思路之后，后续的写请求都将会阻塞在前一个请求的写操作上，导致系统整体性能急剧下降。

总得来说，我们无法找到一种能够满足分布式系统所有系统属性的分布式一致性解决方案。因此如何既保证数据的一致性，同时又不影响系统运行的性能，是每一个分布式系统都需要重点考虑和权衡的。于是，一致性级别由此诞生：

1、强一致性，这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大

2、弱一致性，这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不久承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态

3、最终一致性，最终一致性是弱一致性的一个特例，系统会保证在一定时间内，能够达到一个数据一致的状态。这里之所以将最终一致性单独提出来，是因为它是弱一致性中非常推崇的一种一致性模型，也是业界在大型分布式系统的数据一致性上比较推崇的模型

**分布式环境的各种问题**

分布式系统体系结构从其出现之初就伴随着诸多的难题和挑战：

1、通信异常，从集中式向分布式演变的过程中，必然引入网络因素，由于网络本身的不可靠性，因此也引入了额外的问题。分布式系统需要在各个节点之间进行网络通信，因此每次网络通信都会伴随着网络不可用的风险，网络光纤、路由器或是DNS等硬件设备或是系统不可用都会导致最终分布式系统无法顺利完成一次网络通信。另外，即使分布式系统各个节点之间的网络通信能够正常进行，其延时也会大于单机操作。通常我们认为现代计算机体系结构中，单机内存访问的延时在纳秒数量级（通常是10ns），而正常的一次网络通信的延迟在0.1~1ms左右（相当于内存访问延时的105倍），如此巨大的延时差别，也会影响到消息的收发过程，因此消息丢失和消息延迟变得非常普遍

2、网络分区，当网络由于发生异常情况，导致分布式系统中部分节点之间的网络延时不断增大，最终导致组成分布式系统的所有节点中，只有部分节点之间能够正常通信，而另一些节点则不能----我们将这个现象称为网络分区。当网络分区出现时，分布式系统会出现局部小集群，在极端情况下，这些局部小集群会独立完成原本需要整个分布式系统才能完成的功能，包括对数据的事物处理，这就对分布式一致性提出了非常大的挑战

3、三态，上面两点我们已经了解到在分布式环境下，网络可能会出现各式各样的问题，因此分布式系统的每一次请求与响应，存在特有的三态概念，即成功、失败、超时。在传统的单机系统中，应用程序在调用一个函数之后，能够得到一个非常明确的响应：成功或失败。而在分布式系统中，由于网络是不可靠的，虽然在绝大部分情况下，网络通信也能够接受到成功或失败的响应，当时当网络出现异常的情况下，就可能会出现超时现象，通常有以下两种情况：

（1）由于网络原因，该请求并没有被成功地发送到接收方，而是在发送过程中就发生了消息丢失现象

（2）该请求成功地被接收方接收后，进行了处理，但是在将响应反馈给发送方的过程中，发生了消息丢失现象

当出现这样的超时现象时，网络通信的发起方是无法确定当前请求是否被成功处理的

4、节点故障，节点故障则是分布式环境下另一个比较常见的问题，指的是组成分布式系统的服务器节点出现的宕机或"僵死"现象，通常根据经验来说，每个节点都有可能出现故障，并且每天都在发生

**CAP理论**

一个经典的分布式系统理论。CAP理论告诉我们：一个分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容错性（P：Partition tolerance）这三个基本需求，最多只能同时满足其中两项。

1、一致性，在分布式环境下，一致性是指数据在多个副本之间能否保持一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一直的状态。

对于一个将数据副本分布在不同分布式节点上的系统来说，如果对第一个节点的数据进行了更新操作并且更新成功后，却没有使得第二个节点上的数据得到相应的更新，于是在对第二个节点的数据进行读取操作时，获取的依然是老数据（或称为脏数据），这就是典型的分布式数据不一致的情况。在分布式系统中，如果能够做到针对一个数据项的更新操作执行成功后，所有的用户都可以读取到其最新的值，那么这样的系统就被认为具有强一致性

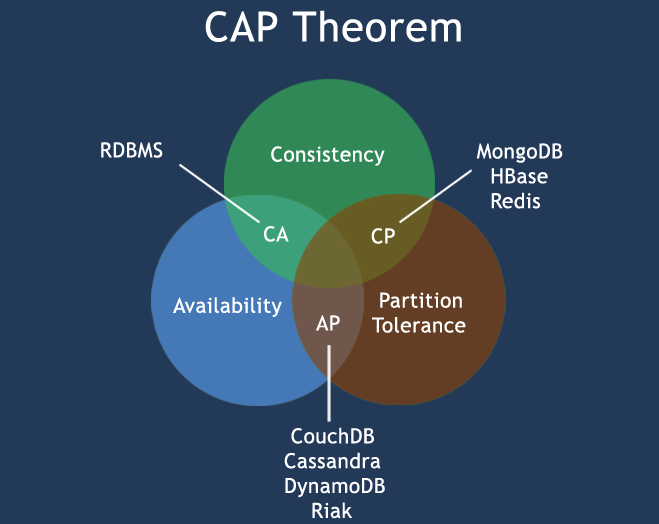
2、可用性，可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。这里的重点是"有限时间内"和"返回结果"。

"有限时间内"是指，对于用户的一个操作请求，系统必须能够在指定的时间内返回对应的处理结果，如果超过了这个时间范围，那么系统就被认为是不可用的。另外，"有限的时间内"是指系统设计之初就设计好的运行指标，通常不同系统之间有很大的不同，无论如何，对于用户请求，系统必须存在一个合理的响应时间，否则用户便会对系统感到失望。

"返回结果"是可用性的另一个非常重要的指标，它要求系统在完成对用户请求的处理后，返回一个正常的响应结果。正常的响应结果通常能够明确地反映出队请求的处理结果，即成功或失败，而不是一个让用户感到困惑的返回结果。

3、分区容错性，分区容错性约束了一个分布式系统具有如下特性：分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

**网络分区**是指在分布式系统中，不同的节点分布在不同的子网络（机房或异地网络）中，由于一些特殊的原因导致这些子网络出现网络不连通的状况，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干个孤立的区域。需要注意的是，组成一个分布式系统的每个节点的加入与退出都可以看作是一个特殊的网络分区。



既然一个分布式系统无法同时满足一致性、可用性、分区容错性三个特点，所以我们就需要抛弃一样：

|  |  |
| --- | --- |
| CAP的3进2 | 实例 |
| CA（强一致、高可用） | 传统单机数据库，可扩展性不强 |
| AP（高可用、分布式容忍） | 大多数网站架构的选择，NoSQL系统 |
| CP（强一致、分布式容忍） | Redis、Mongodb、ZK，网络问题会导致系统不可用 |

对于一个分布式系统而言，分区容错性是一个最基本的要求。因为既然是一个分布式系统，那么分布式系统中的组件必然需要被部署到不同的节点，否则也就无所谓分布式系统了，因此必然出现子网络。而对于分布式系统而言，网络问题又是一个必定会出现的异常情况，因此分区容错性也就成为了一个分布式系统必然需要面对和解决的问题。因此系统架构师往往需要把精力花在如何根据业务特点在C（一致性）和A（可用性）之间寻求平衡。

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。

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

1、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。

2、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。

3、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。

**BASE理论**

BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的缩写。BASE理论是对CAP中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于CAP定理逐步演化而来的。BASE理论的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。接下来看一下BASE中的三要素：

1、基本可用，基本可用是指分布式系统在出现不可预知故障的时候，允许损失部分可用性----注意，这绝不等价于系统不可用。比如：

（1）响应时间上的损失。正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障，查询结果的响应时间增加了1~2秒

（2）系统功能上的损失：正常情况下，在一个电子商务网站上进行购物的时候，消费者几乎能够顺利完成每一笔订单，但是在一些节日大促购物高峰的时候，由于消费者的购物行为激增，为了保护购物系统的稳定性，部分消费者可能会被引导到一个降级页面

2、软状态，软状态指允许系统中的数据存在中间状态，并认为该中间状态的存在不会影响系统的整体可用性，即允许系统在不同节点的数据副本之间进行数据同步的过程存在延时

3、最终一致性，最终一致性强调的是所有的数据副本，在经过一段时间的同步之后，最终都能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

总的来说，BASE理论面向的是大型高可用可扩展的分布式系统，和传统的事物ACID特性是相反的，它完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获得可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。但同时，在实际的分布式场景中，不同业务单元和组件对数据一致性的要求是不同的，因此在具体的分布式系统架构设计过程中，ACID特性和BASE理论往往又会结合在一起。

## 十、Redis入门介绍

**（1）Redis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采用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。

Redis与其他 key-value缓存产品有以下三个特点：

1. 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；
2. 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；
3. Redis支持数据的备份，即master-slave模式的数据备份；

**（2）Redis能干吗？**

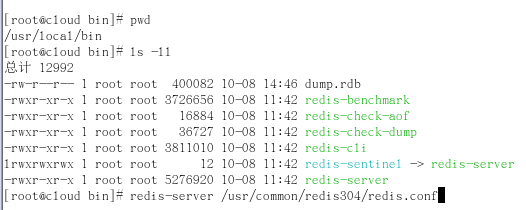
1. 内存存储和持久化：redis支持异步将内存中的数据写到硬盘上，同时不影响继续服务；
2. 取最新N个数据的操作，如：可以将最新的10条评论的ID放在Redis的List集合里面；
3. 模拟类似于HttpSession这种需要设定过期时间的功能；
4. 发布、订阅消息系统；
5. 定时器、计数器；

**（3）Redis安装**

1. 下载获得redis-3.0.4.tar.gz后将它放入我们的Linux目录/opt；
2. /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；
3. 进入目录:cd redis-3.0.4，在redis-3.0.4目录下执行make命令，如果make完成后继续执行make install；
4. 查看默认安装目录：usr/local/bin

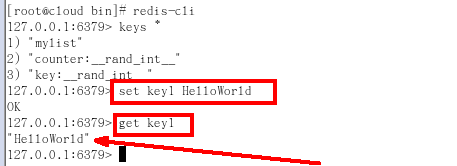
|  |  |
| --- | --- |
| 命令 | 说明 |
| Redis-benchmark | 性能测试工具，可以在自己本子运行，看看自己本子性能如何，需要redis服务启动后执行 |
| Redis-check-aof | 修复有问题的AOF文件 |
| Redis-check-dump | 修复有问题的dump.rdb文件 |
| Redis-cli | 客户端操作入口 |
| Redis-server | Redis服务器启动命令 |
| Redis-sentinel | redis集群使用 |

1. 启动Redis服务，首先修改redis.conf文件将里面的daemonize no 改成 yes，让服务在后台启动；然后将默认的redis.conf拷贝到自己定义好的一个路径下，比如/myconf；最后执行如下命令启动Redis服务。

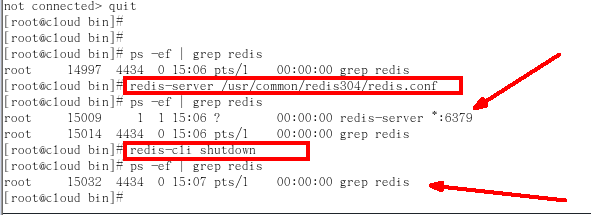


1. 联通性测试，通过redis-cli通过-h指定host，-p指定端口；





1. Redis服务关闭：单实例关闭（redis-cli shutdown）、多实例关闭（指定端口关闭:redis-cli -p 6379 shutdown）



Redis中默认16个数据库，类似数组下表从零开始，初始默认使用零号库，设置数据库的数量（databases 16），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。

|  |  |
| --- | --- |
| 命令 | 说明 |
| databases | 设置数据库的数量，Redis索引都是从零开始 |
| dbsize | 查看当前数据库的key的数量 |
| keys \* | 查看redis中所有键 |
| 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 |

## 十一、Redis[数据类型](Http://redisdoc.com/)

Redis中有五大数据类型 ：String、Hash、List、Set、Zset（Sorted Set）。

**（1）String类型**

String是redis最基本的类型，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：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*。

|  |  |
| --- | --- |
| 命令 | 说明 |
| SET key value | 设置指定key值 |
| GET key | 获取指定key的值 |
| GETRANGE key start end | 返回key中字符串值的子串 |
| GETSET key value | 将给定key值设为value并返回key的旧值 |
| GETBIT key offset | 对key所存储字符串值，获取指定偏移量上的位 |
| MGET key1 [key2...] | 获取所有给定(一个或多个)key的值 |
| SETBIT key offset value | 对Key存储的值，设置或清除指定偏移量上的位 |
| SETEX key seconds value | 将值关联到key并将key的过期时间设为seconds，单位毫秒 |
| SETNX key value | 只有在key不存在时设置key的值 |
| SETRANGE key offset value | 用value覆写给定key所存储的字符串值，从偏移量offset开始 |
| STRLEN key | 返回key所存储的值的长度 |
| MSET key value [key value...] | 批量设置一个或多个key-value对 |
| MSETNX key value [key value...] | 批量设置key-value当且仅当所有给定key都不存在 |
| PSETEX key milliseconds value | 设置Key-Value对并指定过期时间，单位为毫秒 |
| APPEND key value | 如果key存在并且存储的是字符串，则将value追加到原来值的末尾 |
| INCR key | 将key中存储的数字值增1 |
| INCRBY key increment | 将key中存储的数字值加上指定增量increment |
| INCRBYFLOAT key increment | 将key中存储的数字值加上指定浮点增量increment |
| DECR key | 将key中存储的数字值减1 |
| DECRBY key decrement | 将key中存储的数字值减去上指定量decrement |

**（2）Hash类型**

Redis Hash是一个string类型的field和value的映射表，但value为一个键值对，Hash特别适合用于存储对象，类似Java里面的Map<String,Object>。

|  |  |
| --- | --- |
| 命令 | 说明 |
| HDEL key field1 [field2...] | 删除 一个或多个哈希表字段 |
| HEXISTS key field | 查看key对应的哈希表中，指定的字段是否存在 |
| HGET key field | 获取key对应的哈希表中指定字段的值 |
| HMGET key field1 [field2...] | 获取key对应的哈希表中指定所有字段的值 |
| HSET key field value | 设置哈希表key中指定字段的值 |
| HMSET key key field1 value1 [field2 value2...] | 设置哈希表key中所有指定字段的值 |
| HSETNX key field value | 只有在字段field不存在哈希表中时才设置该值 |
| HICREBY key field increment | 为哈希表key中指定key的字段值增加增量increment |
| HICREBYFLOAT key field increment | 为哈希表key中指定key的字段值增加浮点增量increment |
| KKEYS key | 获取哈希表key中的所有键 |
| HVALS key | 获取哈希表key中的所有值 |
| HLEN key | 获取哈希表key的长度 |
| HSCAN key cursor [MATCH pattern] [COUNT count] | 迭代哈希表中的键值对 |

**（3）List类型**

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。

|  |  |
| --- | --- |
| 命令 | 说明 |
| BLPOP key1 [key2] timeout | 移出并获取列表的第一个元素，如果列表没有元素会阻塞列表直到等待时间超时或发现可弹出元素为止 |
| BRPOP key1 [key2] timeout | 移出并获取列表的最后一个元素，如果列表没有元素会阻塞列表直到等待时间超时或发现可弹出元素为止 |
| BRPOPLPUSH source destination timeout | 从列表中弹出一个值，将弹出的元素插入到另一个列表中并返回，如果列表没有元素会阻塞列表直到等待时间超时或发现可弹出元素为止 |
| LINDEX key index | 通过索引获取列表中的元素 |
| LINSERT key BEFORE|AFTER pivot value | 在列表的元素前或后插入元素 |
| LLEN key | 返回列表长度 |
| LPOP key | 移出并获取列表中的第一个元素 |
| LPUSH key value1 [value2...] | 将一个或多个值插入到列表头部 |
| LPUSHX key value | 将一个或多个值插入到已存在的列表头部 |
| LRANGE key start stop | 获取指定列表范围内的元素 |
| LREM key count value | 移除列表元素 |
| LSET key index value | 通过索引设置列表元素的值 |
| LTRIM key start stop | 对一个列表进行裁剪，保留列表指定区间内的元素，不在指定区间内的元素都将被删除 |
| RPOP key | 移除并获取列表最后一个元素 |
| RPOPLPUSH source destination | 移出列表的最后一个元素并将其追加到另一个列表并返回 |
| RPUSH key value1 [value2...] | 在列表中添加一个或多个值 |
| RPUSHX key value | 为已存在的列表添加值 |

**（4）Set类型**

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

|  |  |
| --- | --- |
| 命令 | 说明 |
| SADD key mem1 [mem2...] | 向集合中添加一个或多个成员 |
| SCARD key | 获取集合的成员数 |
| SDIFF key1 [key2...] | 返回给定所有集合的差集 |
| SDIFFSOTRE destination key1 [key2...] | 将给定所有集合的差集存储到destination中 |
| SINTER key1 [key2...] | 返回给定所有集合的交集 |
| SINTERSTORE destination key1 [key2...] | 将给定所有集合的交集存储到destination中 |
| SISMEMBER key member | 判断member元素是否是key的成员 |
| SMEMBERS key | 返回集合中的所有成员 |
| SMOVE source destination member | 将mem元素从source集合移动到destination中 |
| SPOP key | 移除并返回集合中的一个随机元素 |
| SRANDMEMBER key [count] | 返回集合中一个或多个随机成员 |
| SREM key mem1 [mem2...] | 移除集合中的一个或多个成员 |
| SUNION key1 [key2...] | 返回所有给定集合的并集 |
| SUNIONSTORE destination key1 [key2...] | 将所有给定集合的并集存储到destination集合中 |
| SSCAN key cursor [MATCH pattern] [COUNT count] | 迭代集合中的元素 |

**（5）Zset类型**

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*。

|  |  |
| --- | --- |
| 命令 | 说明 |
| ZADD key score1 mem1 [score2 mem2...] | 向有序集合中添加一个或多个成员或更新已存在成员的分数 |
| ZCARD key | 获取有序集合的成员数 |
| ZCOUNT key min max | 计算有序集合在指定区间分数的成员数 |
| ZLEXCOUNT key min max | 在有序集合中计算指定字典区间类成员数量 |
| ZRANGE key start stop [WITHSCORES] | 通过索引区间返回有序集合指定区间内的成员 |
| ZRANGEBYLEX key min max [LIMIT offset count] | 通过字典区间返回有序集合指定区间内的成员 |
| ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] | 通过分数返回有序集合指定区间内的成员 |
| ZRANK key member | 返回有序集合中指定成员的索引 |
| ZINCRBY key increment mem | 有序集合中对指定成员的分数加上增量increment |
| ZREM key mem1 [mem2...] | 移除有序集合中的一个或多个成员 |
| ZREMRANGEBYLEX key min max | 移除字典区间中有序集合的成员 |
| ZREMRANGEBYSCORE key min max | 移除分数区间内有序集合的成员 |
| ZREVRANGE key start stop [WITHSCORES] | 返回有序集中指索引区间内的成员，分数从高到底 |
| ZREVRANGEBYSCORE key max min [WITHSCORES] | 返回有序集中指定分数区间内的成员，分数由高到低排序 |
| ZREVRANK key mem | 返回有序集合中指定成员的排名，有序集成员按分数值递减排序 |
| ZSCORE key mem | 返回有序集合中成员的分数值 |
| ZINTERSTORE destination numkeys key1 [key2...] | 将给定所有有序集合的交集存储到destination中 |
| ZUNIONSTORE destination numkeys key1 [key2...] | 将给定所有有序集合的并集存储到destination中 |
| ZSCAN key cursor [MATCH pattern][COUNT count] | 迭代有序集合中的元素(包括元素成员和元素分值) |

**（6）keys命令**

|  |  |
| --- | --- |
| 命令 | 说明 |
| DEL key | 当key存在时删除key |
| DUMP key | 序列化给定key并返回序列化的值 |
| EXISTS key | 检查给定key是否存在 |
| EXPIRE key seconds | 为给定key设置过期时间，以秒为单位 |
| EXPIREAT key timestamp | 为给定key设置过期时间，时间参数为UNIX秒为时间戳 |
| PEXPIRE key milliseconds | 为给定key设置过期时间，以毫秒为单位 |
| PEXPIREAT key milliseconds-timestamp | 给定key设置过期时间，时间参数为UNIX毫秒时间戳 |
| KEYS pattern | 查找所有符合给定模式的key |
| MOVE key db | 将当前数据库的key移动到给定的数据库db中 |
| PERSIST key | 移除key的过期时间，key将永久保持 |
| PTTL key | 以毫秒为单位 返回key的剩余过期时间 |
| TTL key | 以秒为单位 返回key的剩余过期时间 |
| RANDOMKEY | 从当前数据库中随机返回一个key |
| RENAME key newkey | 修改key的名称 |
| RENAMENX | 仅当newKey不存在时，将key改名为newKey |
| TYPE key | 返回Key所存储的类型 |

## 十二、Redis配置文件

|  |  |
| --- | --- |
| 命令 | 说明 |
| 指定内存时的单位，大小写不敏感 | 1k => 1000 bytes 1kb => 1024 bytes  1m => 1000000 bytes 1mb => 1024\*1024 bytes  1g => 1000000000 bytes 1gb => 1024\*1024\*1024 bytes |
| include | 用于包含通用项，可被多个redis实例共享，例如：include /path/to/local.conf，放在配置文件最后会覆盖前面配置 |
| 网络配置 | |
| bind | 默认redis会监听当前服务器所有网口的redis请求连接，如：bind 192.168.1.100 10.0.0.1，bind 127.0.0.1 |
| port | 默认6379表示redis只监听来自该端口的连接，若为0则redis不会监听 |
| protected-mode | 默认为yes，在保护模式下redis只接收来自loopback口的请求，生产环境通常需要设置为no |
| tcp-backlog | 默认为511，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。 |
| timeout | 0，当客户端连接空闲超过N秒时关闭该连接，0表示关闭该功能 |
| tcp-keepalive | 默认为300单位秒，当客户端空闲时会尝试发送TCP ACK给客户端以保持连接，0则不会进行keepalive检测建议设置为60 |
| 通用配置 | |
| daemonize | no，默认redis作为前台进程运行 |
| supervised | no， If you run Redis from upstart or systemd, Redis can interact with your supervision tree. Options:  *supervised no* - no supervision interaction  *supervised upstart* - signal upstart by putting Redis into SIGSTOP mode  *supervised systemd* - signal systemd by writing READY=1 to $NOTIFY\_SOCKET  *supervised auto* - detect upstart or systemd method based on UPSTART\_JOB or NOTIFY\_SOCKET environment variables  Note: these supervision methods only signal "process is ready." They do not enable continuous liveness pings back to your supervisor. |
| pidfile | /var/run/redis\_6379.pid |
| loglevel | notice，可选：debug、verbose、notice、warning |
| logfile | 默认没有设置日志保存位置，会发送日志到/dev/null |
| syslog-enabled | no，配置是否将redis日志保存到syslog中 |
| syslog-ident | redis，指定syslog identity |
| syslog-facility | local0，指定syslog facility，Must be USER or between LOCAL0-LOCAL7 |
| databases | 16，设置数据库数量，默认在0号数据库，可以通过SELECT <dbid>切换数据库，dbid范围[0, databases-1] |
| 快照 Snapshot | |
| save | 格式：save 秒 写操作次数  用于持久化Redis内存中的数据，当时间和写次数都满足情况时才会进行持久化，save ""可以用于清除该命令之前的所有save配置。  save 900 1 900秒后至少有1次写入时进行持久化  save 300 10 300秒后至少有10次写入时进行持久化  save 60 10000 60秒后至少有1万次写入时进行持久化 |
| Stop-writes-on-bgsave-error | yes，当RDB快照持久化失败时，Redis就会停止处理所有的写请求，当设置为no表示不在乎数据不一致或者有其他的手段发现和控制 |
| rdbcompression | yes，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，这里性能的改善很小，一般开启。 |
| rdbchecksum | yes，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。一般是开启，通常是在空闲时段压缩。 |
| dbfilename | dump.rdb，快照文件名 |
| dir | ./，快照路径，config get dir可以获取 |
| Replication 与Redis集群模式有关，在Redis集群模式时讲解 | |
| 安全 | |
| requirepass | 设置验证面，默认不开启，开启后需要用户使用auth <PASSWORD>验证后才能进行Redis命令的执行 |
| graphic | |
| Limit限制 | |
| 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。 |
| 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 |
| 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 |
| 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配置 | |
| appendonly |  |
| 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，生产上至少为5G |
| auto-aof-rewrite-percentage | 设置重写的基准值，100%表示文件大小翻倍后重写 |

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，为redis-server运行命令时所在的文件夹！！

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

## 十三、Redis Database的持久化：RDB、AOF

RDB持久化在特定时间间隔进行point-in-time快照，而AOF持久化会记录每一次写操作，在服务器重启时会回放日志记录恢复Redis数据库。

**（1）RDB**

RDB在指定的时间间隔内将内存中的数据集快照写入磁盘（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(Copy-On-Write)，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快照保存在二进制dump.rdb文件中，配置save <seconds> <writes>使得RDB生效，dbfilename <filename>指定文件名，dir <dir>指定文件位置

**如何触发RDB快照？**

1、配置文件中默认配置：save 900 1等满足条件就会触发，进行冷拷贝(cp dump.db dump\_new.db)后可重新使用。

2、使用命令save或gbsave触发，save命令执行时只管保存，其它不管，全部阻塞（一般备份、恢复是凌晨做）；bgsave执行时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。

3、执行flushall命令，也会产生dump.rdb文件，但里面是空的，无意义，但会清空缓存，不常用！！

|  |  |
| --- | --- |
| 如何停止 | 动态所有停止RDB保存规则的方法：redis-cli config set save "" |
| 如何恢复快照？ | 将备份文件(dump.rdb)移动到redis安装目录并启动服务即可，通过CONFIG GET dir获取目录。 |
| RDB优势 | 1、适合大规模的数据恢复；2、对数据完整性和一致性要求不高；3、通过fork子进程方式进行持久化最大化redis性能，恢复时较AOF快 |
| RDB劣势 | 1、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；2、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，且数据集大时fork子进程非常耗时可能会导致Redis不能毫秒级响应客户； |

**（2）AOF**

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，相关配置如下：

|  |  |
| --- | --- |
| Append Only Mode配置 | |
| appendonly |  |
| 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，生产上至少为5G |
| auto-aof-rewrite-percentage | 设置重写的基准值，100%表示文件大小翻倍后重写 |

|  |  |
| --- | --- |
| AOF的启动、修复、恢复 | |
| 正常恢复 | 1. 修改默认的appendonly no，改为yes； 2. 将有数据的aof文件复制一份保存到对应目录(config get dir) 3. 恢复：重启redis然后重新加载 |
| 异常恢复 | 1. 修改默认的appendonly no，改为yes； 2. 备份被写坏的AOF文件； 3. Redis-check-aof --fix进行修复； 4. 恢复：重启redis然后重新加载； |



**Rewrite机制**

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。

|  |  |
| --- | --- |
| AOP优势 | 1、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；  2、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；  3、不同步：appendfsync no，从不同步； |
| AOF劣势 | 1、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；  2、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； |

**（3）总结**

RDB和AOF两者可以同时配置，当都存在是恢复时会使用AOF文件。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。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，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。

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。

同时开启两种持久化方式：

1、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。

2、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。

**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。

RDB --> AOF（解决RBD丢数过大问题） --> Mastart/Slave（解决AOF重写导致阻塞的问题）

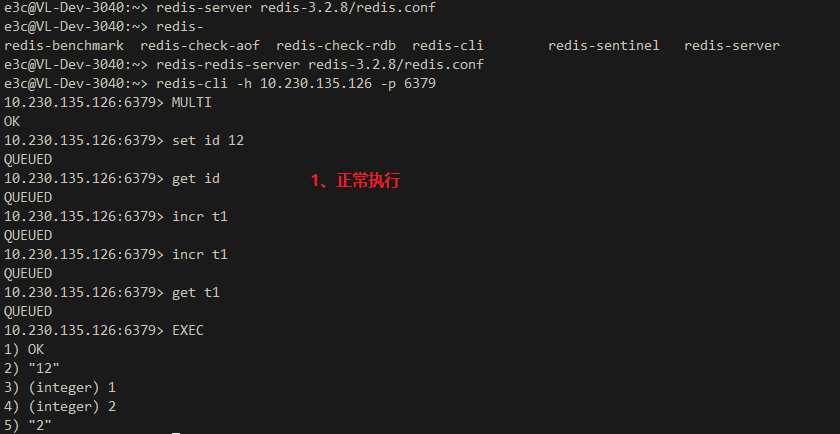
## 十四、Redis事务

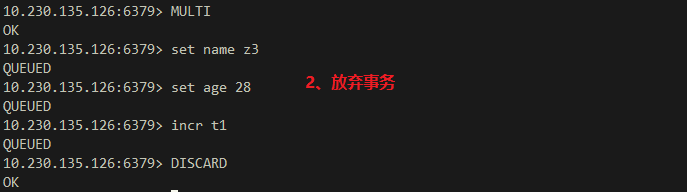
*一个Redis事务中中的所有命令会被序列化并顺序执行，这些命令的执行具有隔离性，即不允许其他命令的插入或加塞*。Redis事务具有原子性，事务中的命令要么全部执行要么不执行，EXEC命令用于触发Redis事务中所有命令的执行。当使用AOF机制时，Redis会通过一次系统调用将该事务持久化到磁盘上，在写入期间若Redis进程终止则会导致事务只被写入了部分，Redis在重启时会检测到错误并且启动失败，此时需要使用redis-check-aof工具修复AOF文件，即把不完整事务删除以确保Redis服务可以正常启动。

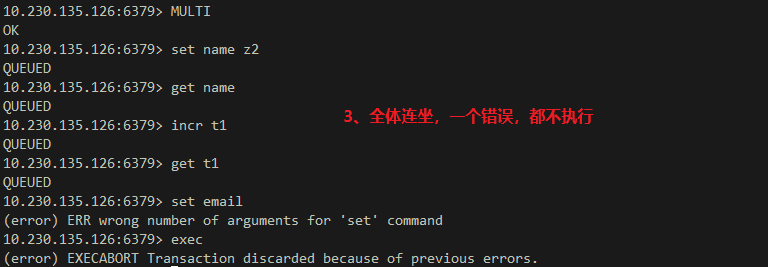
**能干什么**？一个队列中，一次性、顺序性、排他性的执行一系列命令。

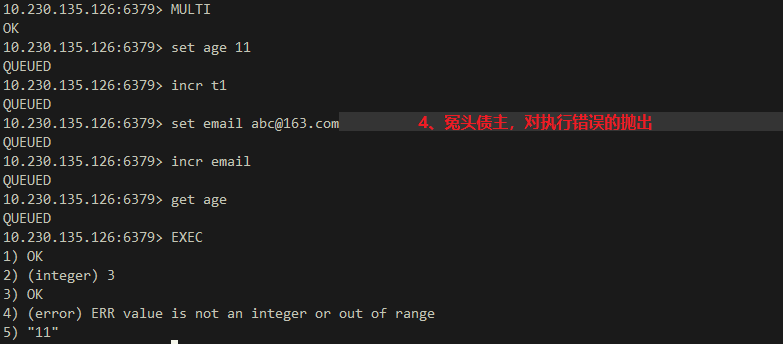
**怎么玩？**

|  |  |
| --- | --- |
| 命令 | 说明 |
| DISCARD | 取消事务，放弃执行事务块内的所有命令； |
| EXEC | 执行所有事务块内的命令 |
| MULTI | 标记一个事务的开始 |
| WATCH | key1 [key2...] 监视一个(或多个)key，如果在事务执行之前(或这些)key被其他命令所改动，那么事务将被打断。 |
| UNWATCH | 取消WATCH命令对所有key的监视 |









**Watch监控**

悲观锁(Pessimistic Lock)，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(Optimistic Lock)，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(Check And Set) 无锁算法](https://www.cnblogs.com/Mainz/p/3546347.html)，CPU指令，在大多数处理器架构，包括IA32、Space中采用的都是CAS指令，CAS的语义是“我认为V的值应该为A，如果是，那么将V的值更新为B，否则不修改并告诉V的值实际为多少”，CAS是项乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。CAS无锁算法的C实现如下：

在JDK1.5之前，如果不编写明确的代码就无法执行CAS操作，在JDK1.5中引入了底层的支持，在int、long和对象的引用等类型上都公开了CAS的操作，并且*JVM把它们编译为底层硬件提供的最有效的方法，在运行CAS的平台上，运行时把它们编译为相应的机器指令，如果处理器/CPU不支持CAS指令，那么JVM将使用自旋锁*。AtomicLong.incrementAndGet的实现用了乐观锁技术，调用了sun.misc.Unsafe类库里面的 CAS算法，用CPU指令来实现无锁自增。所以，AtomicLong.incrementAndGet的自增比用synchronized的锁效率倍增。

*服务端编程的3大性能杀手：1、大量线程导致的线程切换开销。2、锁。3、非必要的内存拷贝*。在高并发下,对于纯内存操作来说,单线程是要比多线程快的, 可以比较一下多线程程序在压力测试下cpu的sy和ni百分比。高并发环境下要实现高吞吐量和线程安全，两个思路：一个是用优化的锁实现，一个是lock-free的无锁结构。但非阻塞算法要比基于锁的算法复杂得多。开发非阻塞算法是相当专业的训练，而且要证明算法的正确也极为困难，不仅和具体的目标机器平台和编译器相关，而且需要复杂的技巧和严格的测试。虽然Lock-Free编程非常困难，但是它通常可以带来比基于锁编程更高的吞吐量。所以Lock-Free编程是大有前途的技术。它在线程中止、优先级倒置以及信号安全等方面都有着良好的表现。

CAS虽然很高效的解决原子操作，但是*CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操*作。

1. [ABA问题](http://blog.hesey.NET/2011/09/resolve-aba-by-atomicstampedreference.html)。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

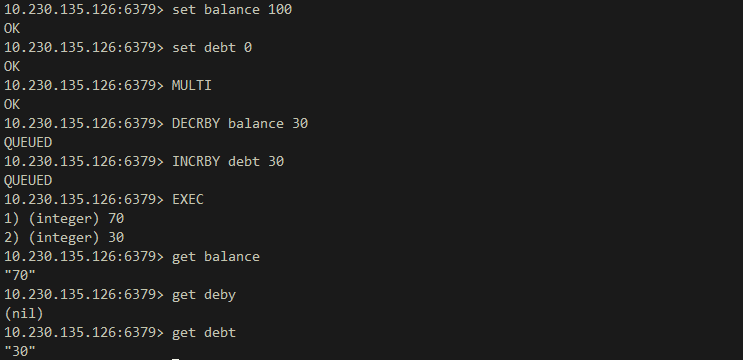
从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于ABA问题参考文档:

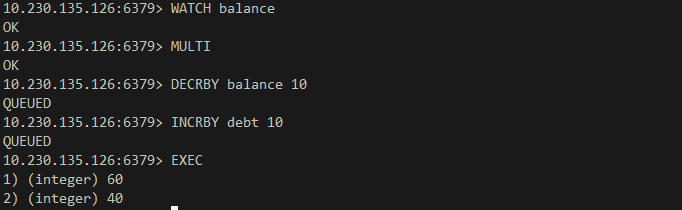
2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3. 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

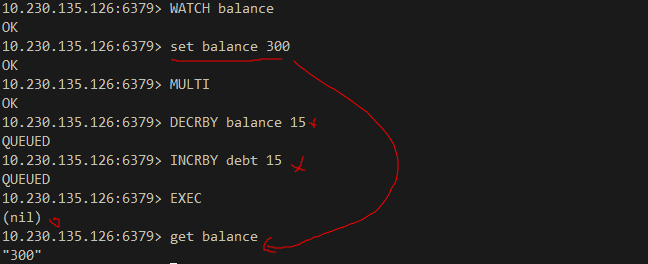
1、初始化信用卡可用余额和欠额



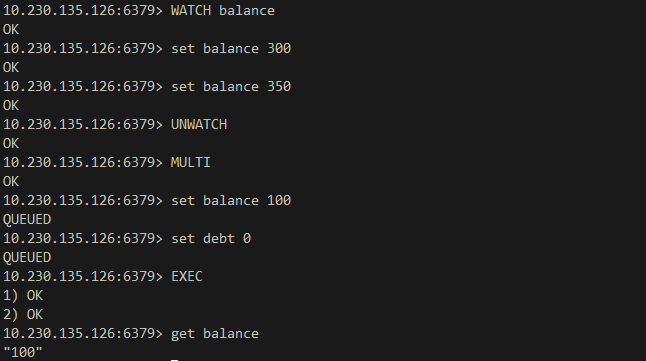
2、无加塞篡改，先监控再开启multi，保证两笔金额变动在同一个事务内



3、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



4、unwatch 取消对所有key的监控，如下图所示：检测到有数据被改动了，放弃本次的监控unwatch，重新watch监控数据是否被改动。



总结：

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

### Redis事务三阶段

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

### Redis事务三特性

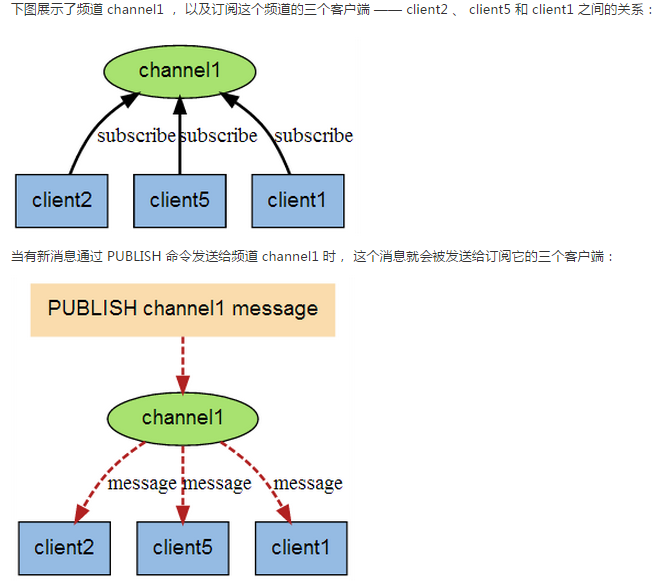
1、单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

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。

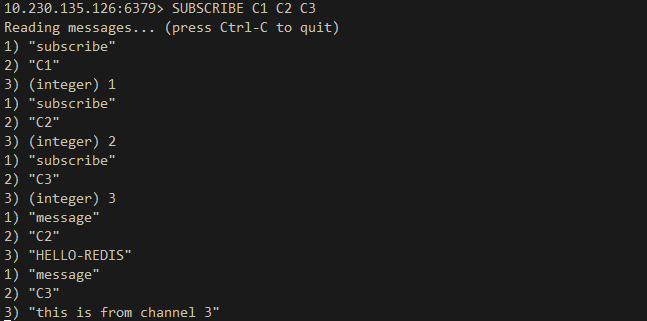
## 十五、Redis发布订阅消息

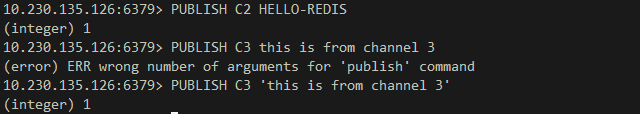
是什么？进程间的一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。实际生产会采用专业的消息中间件，如下图为订阅/发布示例。



|  |  |
| --- | --- |
| 命令 | 说明 |
| PSUBSCRIBE pattern [pattern…] | 订阅一个或多个符合给定模式的频道,  支持通配符\*， PSUBSCRIBE new\* |
| PUBSUB subcmd [arg …] | 查看订阅和发布系统的状态 |
| PUBLISH channel message | 将消息发送到指定的频道 |
| PUNSUBSCRIBE pattern [pattern…] | 退订所有给定模式的频道 |
| SUBSCRIBE channel [channel…] | 定于给定的一个或多个频道 |
| UNSUBSCRIBE channel [channel…] | 退订指定的频道 |

**案例：先订阅后发布后才能收到消息**





## 十六、Redis Master/Slave

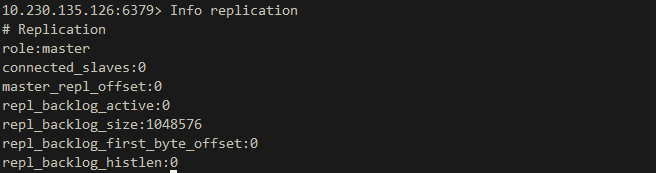
**是什么？**主从复制，主机数据更新后根据配置和策略，自动同步到备机的master/slaver机制，Master以写为主，Slave以读为主。Redis的Replication机制是异步的不会阻塞Master或Slave对外服务。

**能干什么？**读写分离、容灾恢复。

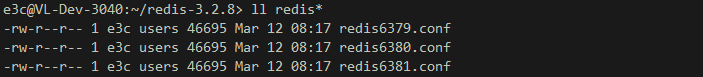
**怎么玩？**

1、配从(库)不配主(库)

2、从库配置：slaveof 主库IP 主库端口，命令行配置在每次与master断开之后，都需要重新连接，除非你配置进redis.conf文件

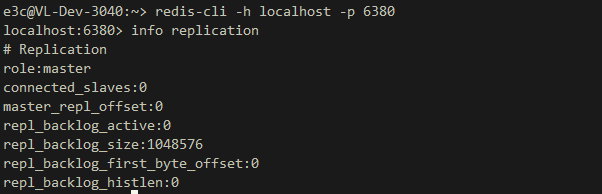


3、首先复制若干份配置文件，然后进行配置



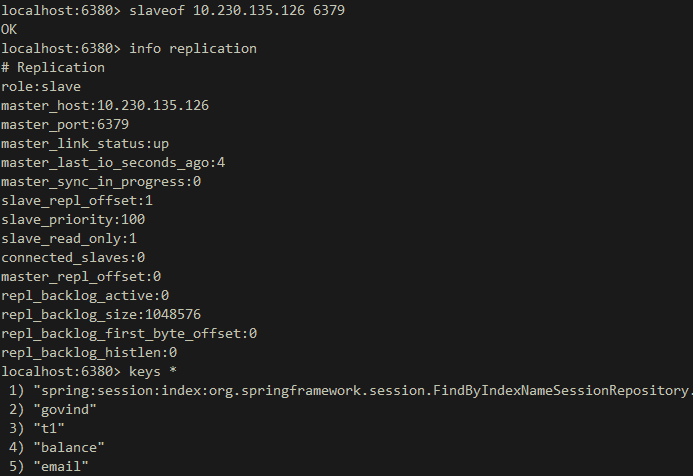
|  |  |
| --- | --- |
| 配置项 | 值 |
| daemonize | yes |
| pidfile | /var/run/redis6379.pid |
| port | 6379 |
| logfile | redis\_6379.log |
| dbfilename | dump6379.rdb |

4、一次通过 redis-server redis[6389-6380].conf启动三个Redis服务，初始启动时三个Redis服务器都是Master。

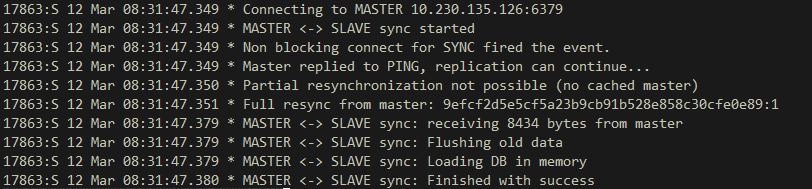


**一主两仆：一个Master，两个Slave**

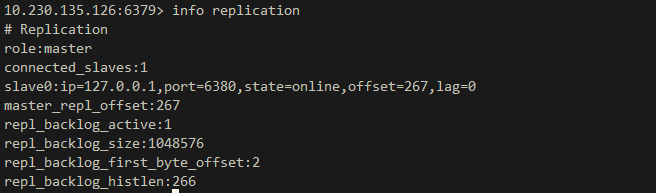
在两个从机器上分别执行命令：SLAVEOF 127.0.0.1 6379，可以从Slave机器上查看到Master机器上的数据



Slave机器日志如下



Master信息如下



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？ slaveof IP PORT 之后从机会完全复制Master上的数据；

2、从机是否可以写？set可否？ 不可以，从机只能读，不允许写操作！

3、主机shutdown后情况如何？从机是上位还是原地待命？主机shutdown后，从机原地待命，不会升级争抢Master角色，再Master重启后集群继续工作；

4、主机又回来了后，主机新增记录，从机还能否顺利复制？再Master重启后集群继续工作；

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？从机每次断开需要与Master节点重连后才能同步数据，除非将slaveof配置写入配置文件中；

**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**

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**

Slave启动成功连接到master后会发送一个sync命令，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。

* 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；
* 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；

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

**复制缺点：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。

**哨兵模式（Sentinel）**

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。一组sentinel能同时监控多个Master！

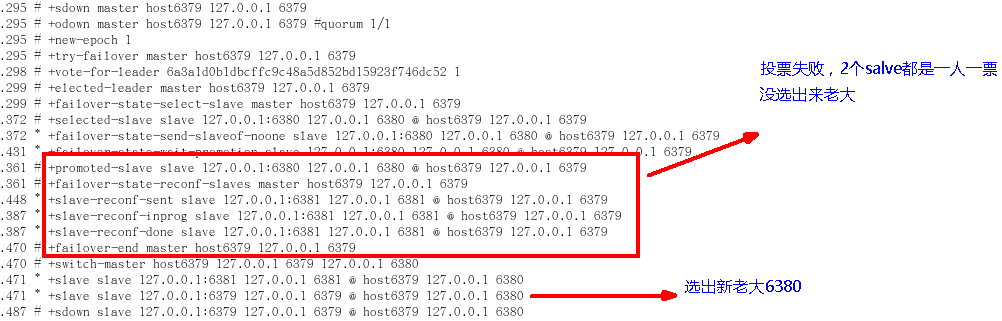
配置步骤

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

2、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，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；

3、启动哨兵：redis-sentinel redis304/sentinel.conf

4、原有Master挂掉，在Redis-cli中执行shutdown手动关闭服务，可以通过查看日志正在进行新Master的选举；



5、选取成功后，Redis主从结构确定可以继续工作；

6、如果之前的master重启回来，会不会双master冲突？Master回来之后，自动变为Slave。

## 十六、Jedis实战

private static void OperationOfFiveTypes*() {* Jedis jedis = new Jedis*(*"10.230.135.126", 6379*)*;  
 //查看服务是否运行，打出pong表示OK  
 *log*.info*(*"{}", jedis.ping*())*;  
  
 //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*)*;  
 *}  
}*public static void doTransaction*() {* Jedis jedis = new Jedis*(*"10.230.135.126", 6379*)*;  
  
 try *{* //监控key 如果改动了事务就被放弃  
 jedis.watch*(*"serialNum"*)*;  
 //jedis.set("serialNum", "ss#$");  
 //jedis.unwatch();  
  
 Transaction trans = jedis.multi*()*;  
 trans.set*(*"serialNum", "s002"*)*;  
 Response*<*String*>* response = trans.get*(*"serialNum"*)*;  
 trans.lpush*(*"list3", "a"*)*;  
 trans.lpush*(*"list3", "b"*)*;  
 trans.lpush*(*"list3", "c"*)*;  
  
 trans.exec*()*;  
 //trans.discard();  
 *log*.info*(*"serialNum: {}", response.get*())*;  
 trans.close*()*;  
 *}* catch *(*IOException e*) {* e.printStackTrace*()*;  
 *}  
}  
  
/\*\*  
 \* 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 boolean transactionAsLock*() {* Jedis jedis = new Jedis*(*"10.230.135.126", 6379*)*;  
 int balance; //可用余额  
 int amtToSubtract = 10;//实刷额度  
  
 jedis.watch*(*"balance"*)*;  
 //此句不该出现，模拟被其他程序修改的情况  
 //jedis.set("balance", "4");  
  
 balance = Integer.*valueOf(*jedis.get*(*"balance"*))*;  
 if *(*balance < amtToSubtract*) {* jedis.unwatch*()*;  
 *log*.info*(*"余额不足"*)*;  
 return false;  
 *}* else *{  
 log*.info*(*"开始事务：准备扣减"*)*;  
 Transaction transaction = jedis.multi*()*;  
 transaction.decrBy*(*"balance", amtToSubtract*)*;  
 transaction.incrBy*(*"debt", amtToSubtract*)*;  
 transaction.exec*()*;  
 *log*.info*(*"事务执行完成，balance={}, debt={}",jedis.get*(*"balance"*)*,jedis.get*(*"debt"*))*;  
 return true;  
 *}  
}*public static void masterSlave*()* throws InterruptedException *{* Jedis jedisMaster = new Jedis*(*"10.230.135.126", 6379*)*;  
 Jedis jedisSlave = new Jedis*(*"10.230.135.126", 6380*)*;  
  
 jedisSlave.slaveof*(*"10.230.135.126", 6379*)*;  
 jedisMaster.set*(*"K6", "V6"*)*;  
 Thread.*sleep(*2000*)*;  
 *log*.info*(*"K6={}", jedisSlave.get*(*"K6"*))*;  
*}*

**Redis连接池**

*/\*\*  
 \* JedisPoolConfig参数配置  
 \** ***@param*** *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。  
 \** ***@param*** *maxIdle：控制一个pool最多有多少个状态为idle(空闲)的jedis实例。  
 \** ***@param*** *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；  
 \** ***@param*** *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；  
 \** ***@param*** *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；  
 \*  
 \** ***@param*** *testOnReturn：return一个jedis实例给pool时，是否检查连接可用性ping()；  
 \*  
 \** ***@param*** *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；  
 \*  
 \** ***@param*** *timeBetweenEvictionRunsMillis：表示idle object evitor两次扫描之  
 \* 间要sleep的毫秒数；  
 \*  
 \** ***@param*** *numTestsPerEvictionRun：表示idle object evitor每次扫描的最多的对象数；  
 \*  
 \** ***@param*** *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；  
 \*  
 \** ***@param*** *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；  
 \*  
 \** ***@param*** *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；  
 \*  
 \*/*public class JedisPoolUtil *{* private JedisPoolUtil*(){}  
  
 /\*\*  
 \* 采用静态内部类实现单例模式  
 \*/* private static class JedisPoolHelper*{  
 /\*\*  
 \* 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 static volatile JedisPool *jedisPool*;  
 static *{* JedisPoolConfig config = new JedisPoolConfig*()*;  
 config.setMaxIdle*(*32*)*;  
 config.setMaxWaitMillis*(*100\*1000*)*;  
 config.setTestOnBorrow*(*true*)*;  
 config.setMaxTotal*(*1000*)*;  
 *jedisPool* = new JedisPool*(*config, "10.230.135.126",6379*)*;  
 *}  
 }* public static JedisPool getInstance*() {* return JedisPoolHelper.*jedisPool*;  
 *}* public static void release*(*Jedis jedis*) {* if *(*jedis != null*) {* jedis.close*()*;  
 *}  
 }  
}*