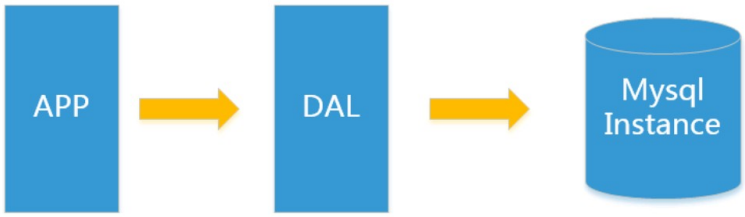
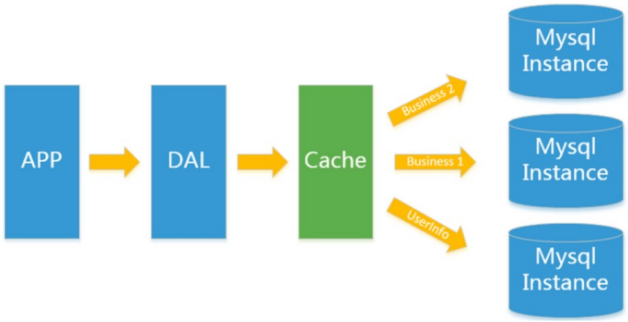
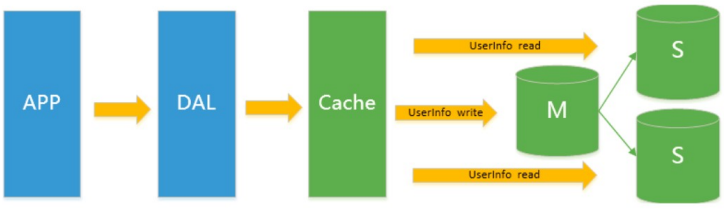
一、NoSql入门和概述 @CreateByShadow

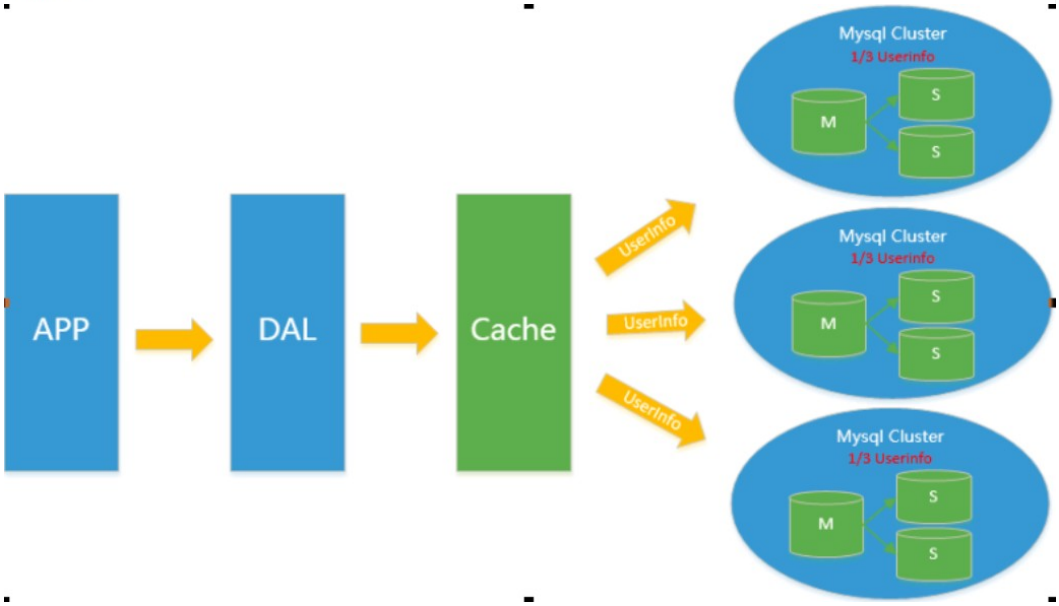
1. 入门概述

- 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
>
> 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。
>
> 
>
>
> 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。

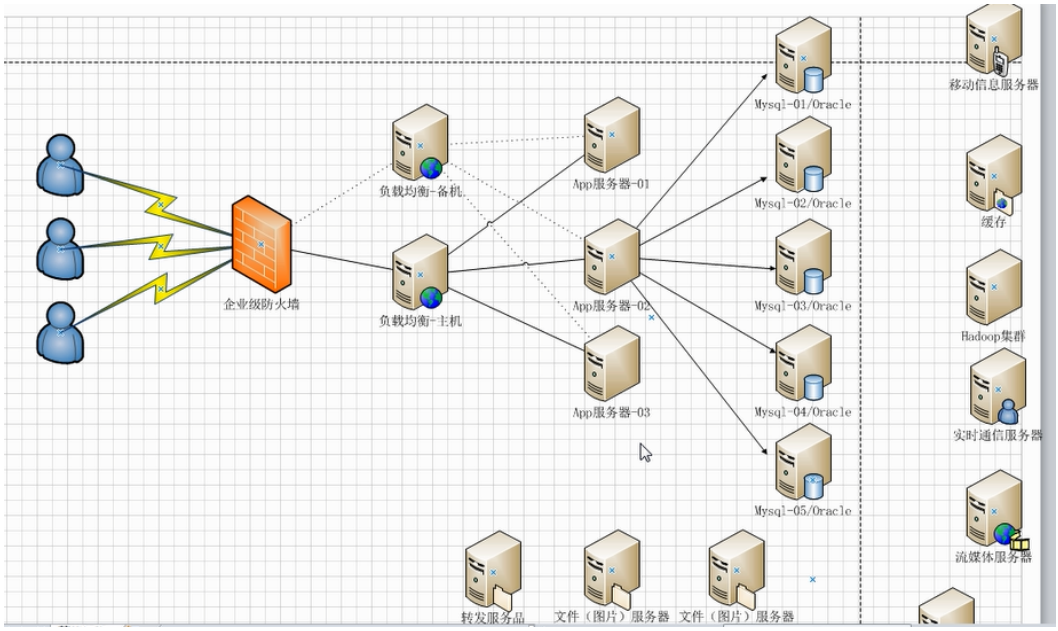
同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但性能也不能很好满足互联网的要求，只是在高可靠性上提供了非常大的保证。



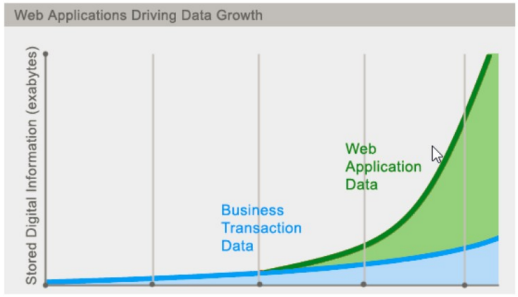
5. MySQL的扩展性瓶颈

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

1. 易扩展

> NoSQL数据库种类繁多，但是有个共同的特点都是去掉关系数据库的关系模型特性。
>
> 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无需实现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
>
> -- 高度组织化结构化数据
>
> -- 结构化查询语句（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
    1. Redis
    2. Memcache
    3. Mongdb
- 怎么玩
    1. KV
    2. Cache
    3. Persistence
    4. ....

2. 3V + 3高

- 大数据时代的3V
    1. **海量Volume**
    2. **多样Variety**
    3. **实时Velocity**
- 互联网需求的3高
    1. **高并发**
    2. **高可扩**
    3. **高性能**

3. 当下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问题

4. NoSQL数据模型简介

- 聚合模型
    1. KV键值
    2. Bson（binary Json）
    3. 列族
    4. 图形

5. NoSQL数据库的四大分类

- KV键值：典型介绍
    1. 新浪：BerkeleyDB + redis
    2. 美团：redis + tair

3. 阿里、百度：memcache + redis

- 文档型数据库（bson格式比较多）：典型介绍
    1. CouchDB
    2. MongDB
- 列存储数据库
    1. Cassandra、HBase
    2. 分布式文件系统
- 图关系数据库
    1. 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
    2. Neo4J、InfoGrid
- 四者对比

分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值（key-value）	Tokyo、Cabinet/Tyrant、Redis、Voldemort、Oracle BDB	内容缓存，主要用于处理大量社交的高访问负载，也用于一些日子系统等等	Key指向Value的键值对，通常用hashtable来实现	查找速度快	数据无结构化，通常只被当做字符串或者二进制数据
列存储数据库	Cassandra、HBase、Riak	分布式的文件系统	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相对局限
文档型数据库	CouchDB、MongDB	Web应用（Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容）	Key-Value对应的键值对，Value为结构化数据	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	查询性能不高，而且缺乏统一的查询语法
图形（Graph）数据库	Neo4J、InfoGrid、Infinite Graph	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。比如最短路径寻址，N度关系查找等	很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案

6. 在分布式数据库中CAP原理CAP+BASE

- **传统的ACID分别是什么**
  1. **A（Atomicity）原子性**
  2. **C（Consistency）一致性**

3. **I（Isolation）隔离性**
4. **D（Durability）持久性**

- **CAP**

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

- CAP的3进2
- 经典CAP图
- BASE是什么

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。

二、解析配置文件redis.conf

1. 文件位置

1.1 Windows版

1.2 Linux版

2. Units 单位

> Redis 内部单位说明

```
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
# 大小写不敏感
# units are case insensitive so 1GB 1Gb 1gB are all the same.
```

3. INCLUDES 包含

> redis.conf 可以作为总闸，包含其他的配置文件

```
################################## INCLUDES ###################################

# Include one or more other config files here.  This is useful if you
# have a standard template that goes to all Redis servers but also need
# to customize a few per-server settings.  Include files can include
# other files, so use this wisely.
#
# Notice option "include" won't be rewritten by command "CONFIG REWRITE"
# from admin or Redis Sentinel. Since Redis always uses the last processed
# line as value of a configuration directive, you'd better put includes
# at the beginning of this file to avoid overwriting config change at runtime.
#
# If instead you are interested in using includes to override configuration
# options, it is better to use include as the last line.
#
# include /path/to/local.conf
# include /path/to/other.conf
```

4. GENERAL 、NETWORK通用

4.1 daemonize

> 是否开启守护进程
>
> 默认情况下，Redis不作为守护进程运行。如果需要，请使用“yes”
>
> 注意，当被守护时，Redis将在/var/run/redis.pid中写入一个pid文件

```
# By default Redis does not run as a daemon. Use 'yes' if you need it.
# Note that Redis will write a pid file in /var/run/redis.pid when
daemonized.
daemonize no
```

4.2 pidfile

> pidfile文件

```
# If a pid file is specified, Redis writes it where specified at startup
# and removes it at exit.
#
# When the server runs non daemonized, no pid file is created if none is
# specified in the configuration. When the server is daemonized, the pid
file
# is used even if not specified, defaulting to "/var/run/redis.pid".
#
# Creating a pid file is best effort: if Redis is not able to create it
# nothing bad happens, the server will start and run normally.
pidfile /var/run/redis_6379.pid
```

4.3 port

> 端口绑定
>
> 接受指定端口号上的连接，默认6379
>
> 如果 port 指定 为 0，Redis将不会进行tcp socket监听

```
# Accept connections on the specified port, default is 6379 (IANA #815344).
# If port 0 is specified Redis will not listen on a TCP socket.
port 6379
```

4.4 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的值
>
> 来达到想要的效果

```
# TCP listen() backlog.
#
# In high requests-per-second environments you need an high backlog in
order
# to avoid slow clients connections issues. Note that the Linux kernel
# will silently truncate it to the value of /proc/sys/net/core/somaxconn so
# make sure to raise both the value of somaxconn and tcp_max_syn_backlog
# in order to get the desired effect.
tcp-backlog 511
```

4.5 timeout

> 在客户端空闲N秒后关闭连接（0表示禁用）

```
# Close the connection after a client is idle for N seconds (0 to disable)
timeout 0
```

4.6 bind

> 地址绑定

```
# By default, if no "bind" configuration directive is specified, Redis
listens
```

```
# for connections from all the network interfaces available on the server.
# It is possible to listen to just one or multiple selected interfaces
using
# the "bind" configuration directive, followed by one or more IP addresses.
#
# Examples:
#
# bind 192.168.1.100 10.0.0.1
# bind 127.0.0.1 ::1
#
# ~~~ WARNING ~~~ If the computer running Redis is directly exposed to the
# internet, binding to all the interfaces is dangerous and will expose the
# instance to everybody on the internet. So by default we uncomment the
# following bind directive, that will force Redis to listen only into
# the IPv4 loopback interface address (this means Redis will be able to
# accept connections only from clients running into the same computer it
# is running).
#
# IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
# JUST COMMENT THE FOLLOWING LINE.
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bind 127.0.0.1
```

4.7 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

```
# TCP keepalive.
#
# If non-zero, use SO_KEEPALIVE to send TCP ACKs to clients in absence
# of communication. This is useful for two reasons:
#
# 1) Detect dead peers.
# 2) Take the connection alive from the point of view of network
#    equipment in the middle.
#
# On Linux, the specified value (in seconds) is the period used to send
ACKs.
# Note that to close the connection the double of the time is needed.
# On other kernels the period depends on the kernel configuration.
#
# A reasonable value for this option is 300 seconds, which is the new
# Redis default starting with Redis 3.2.1.
tcp-keepalive 300
```

4.8 loglevel

> 指定日志级别：
>
> - debug
> - verbose
> - notice
> - warning

```
# Specify the server verbosity level.
# This can be one of:
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably)
# warning (only very important / critical messages are logged)
loglevel notice
```

4.9 logfile

> 指定日志文件

```
# Specify the log file name. Also the empty string can be used to force
# Redis to log on the standard output. Note that if you use standard
# output for logging but daemonize, logs will be sent to /dev/null
logfile ""
```

4.10 syslog-enabled

> 是否开启系统日志

```
# To enable logging to the system logger, just set 'syslog-enabled' to yes,
# and optionally update the other syslog parameters to suit your needs.
# syslog-enabled no
```

4.11 syslog-ident

> 指定系统日志标识

```
# Specify the syslog identity.
# syslog-ident redis
```

4.12 syslog-facility

> 指定syslog设备。必须是用户或介于LOCAL0-LOCAL7之间

```
# Specify the syslog facility. Must be USER or between LOCAL0-LOCAL7.
# syslog-facility local0
```

4.13 databases

> 设置数据库数量，通过 SELECT index 切换库

```
# Set the number of databases. The default database is DB 0, you can select
# a different one on a per-connection basis using SELECT <dbid> where
# dbid is a number between 0 and 'databases'-1
databases 16
```

5. SNAPSHOTTING 快照（见下面持久化）

5.1 save

- save 秒钟 写操作次数

  > 默认三种（或关系）：
  >
  > - 一分钟修改1万次
  > - 五分钟修改10次

    - 十五分钟修改1次
- 禁用
    - 不设置任何 save 指令
    - save ""
- 手动 save，马上更新快照

5.2 stop-write-on-gbsave-error

> 后台保存，停止写操作

```
# By default Redis will stop accepting writes if RDB snapshots are enabled
# (at least one save point) and the latest background save failed.
# This will make the user aware (in a hard way) that data is not persisting
# on disk properly, otherwise chances are that no one will notice and some
# disaster will happen.
#
# If the background saving process will start working again Redis will
# automatically allow writes again.
#
# However if you have setup your proper monitoring of the Redis server
# and persistence, you may want to disable this feature so that Redis will
# continue to work as usual even if there are problems with disk,
# permissions, and so forth.
stop-writes-on-bgsave-error yes
```

5.3 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。

```
# Compress string objects using LZF when dump .rdb databases?
# For default that's set to 'yes' as it's almost always a win.
# If you want to save some CPU in the saving child set it to 'no' but
# the dataset will likely be bigger if you have compressible values or
keys.
rdbcompression yes
```

5.4 rdbchecksum

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

```
# Since version 5 of RDB a CRC64 checksum is placed at the end of the file.
# This makes the format more resistant to corruption but there is a
performance
# hit to pay (around 10%) when saving and loading RDB files, so you can
disable it
# for maximum performances.
#
# RDB files created with checksum disabled have a checksum of zero that
will
# tell the loading code to skip the check.
rdbchecksum yes
```

5.5 dbfilename

> 存储文件名称

```
# The filename where to dump the DB
dbfilename dump.rdb
```

5.6 dir

> 文件目录

```
# The working directory.
#
# The DB will be written inside this directory, with the filename specified
# above using the 'dbfilename' configuration directive.
#
# The Append Only File will also be created inside this directory.
#
# Note that you must specify a directory here, not a file name.
dir ./
```

6. REPLICATION 复制（备份）

7. SECURITY 安全

7.1 requirepass

> 设置密码

```
# Require clients to issue AUTH <PASSWORD> before processing any other
# commands.  This might be useful in environments in which you do not trust
# others with access to the host running redis-server.
#
# This should stay commented out for backward compatibility and because
most
# people do not need auth (e.g. they run their own servers).
#
# Warning: since Redis is pretty fast an outside user can try up to
# 150k passwords per second against a good box. This means that you should
# use a very strong password otherwise it will be very easy to break.
#
# requirepass foobared
```

7.2 auth

> 输入密码

```
auth foobared
```

7.3 config 客户端指令获取设置值

> config get requirepass 获取密码
>
> config set requirepass "xxx" 设置密码
>
> config get dir 获取路径
>
> config get [配置文件里的key；bind port...] 获取配置值
>
> config set [配置文件里的key；bind port...] 设置配置值

8. LIMITS （CLIENTS 、MEMORY MANAGEMENT）限制

8.1 maxclients

> 配置最大客户端支持数

```
# Set the max number of connected clients at the same time. By default
# this limit is set to 10000 clients, however if the Redis server is not
# able to configure the process file limit to allow for the specified limit
# the max number of allowed clients is set to the current file limit
# minus 32 (as Redis reserves a few file descriptors for internal uses).
#
# Once the limit is reached Redis will close all the new connections
sending
# an error 'max number of clients reached'.
#
# maxclients 10000
```

8.2 maxmemory

> 最大内存

```
# Set a memory usage limit to the specified amount of bytes.
# When the memory limit is reached Redis will try to remove keys
# according to the eviction policy selected (see maxmemory-policy).
#
# If Redis can't remove keys according to the policy, or if the policy is
# set to 'noeviction', Redis will start to reply with errors to commands
# that would use more memory, like SET, LPUSH, and so on, and will continue
# to reply to read-only commands like GET.
#
# This option is usually useful when using Redis as an LRU or LFU cache, or
to
# set a hard memory limit for an instance (using the 'noeviction' policy).
#
# WARNING: If you have replicas attached to an instance with maxmemory on,
# the size of the output buffers needed to feed the replicas are subtracted
# from the used memory count, so that network problems / resyncs will
# not trigger a loop where keys are evicted, and in turn the output
# buffer of replicas is full with DELs of keys evicted triggering the
deletion
# of more keys, and so forth until the database is completely emptied.
#
# In short... if you have replicas attached it is suggested that you set a
lower
# limit for maxmemory so that there is some free RAM on the system for
replica
# output buffers (but this is not needed if the policy is 'noeviction').
#
# maxmemory <bytes>
```

8.3 maxmemory-policy

> 最大内存策略：配置 redis 数据缓存过期策略

```
# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
# is reached. You can select among five behaviors:
# 使用LRU（least RecenTly Used：最近最少使用）算法移除key，只对设置了过期时间的键
```

```
# volatile-lru -> Evict using approximated LRU among the keys with an
expire set.
# 使用LRU（least Recently Used：最近最少使用）算法移除key
# allkeys-lru -> Evict any key using approximated LRU.
# 使用LFU(Least Frequently Used：最近访问频率最小的元素)算法移除key，只对设置了过期
时间的键
# volatile-lfu -> Evict using approximated LFU among the keys with an
expire set.
# 使用LFU(Least Frequently Used：最近访问频率最小的元素)算法移除key
# allkeys-lfu -> Evict any key using approximated LFU.
# 随机移除key，只对设置过期时间的
# volatile-random -> Remove a random key among the ones with an expire set.
# 随机移除任何key
# allkeys-random -> Remove a random key, any key.
# 移除那些TTL(time to leave)值最小的key，即那些最近要过期的的key
# volatile-ttl -> Remove the key with the nearest expire time (minor TTL)
# 永不过期,针对写操作，只返回错误信息
# noeviction -> Don't evict anything, just return an error on write
operations.
#
# LRU means Least Recently Used
# LFU means Least Frequently Used
#
# Both LRU, LFU and volatile-ttl are implemented using approximated
# randomized algorithms.
#
# Note: with any of the above policies, Redis will return an error on write
#       operations, when there are no suitable keys for eviction.
#
#       At the date of writing these commands are: set setnx setex append
#       incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd
#       sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby
#       zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby
#       getset mset msetnx exec sort
#
# The default is:
# 默认永不过期
# maxmemory-policy noeviction
```

8.4 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并选择其中的LRU的那个

```
# LRU, LFU and minimal TTL algorithms are not precise algorithms but
approximated
# algorithms (in order to save memory), so you can tune it for speed or
# accuracy. For default Redis will check five keys and pick the one that
was
# used less recently, you can change the sample size using the following
# configuration directive.
#
# The default of 5 produces good enough results. 10 Approximates very
closely
# true LRU but costs more CPU. 3 is faster but not very accurate.
#
# maxmemory-samples 5
```

9. APPEND ONLY MODE 追加（见下面持久化）

9.1 appendonly

> 是否开启aof，默认 no，开始改为 yes

9.2 appendfilename

> aof文件名称，默认 appendonly.aof

9.3 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:
  > 不同步

9.4 no-appendfsync-on-rewrite

> 重写时是否可以运用appendfsync，默认是no即可，保证数据安全性

9.5 auto-aof-rewrite-min-size

> 设置重写的基准值大小

9.6 auto-aof-rewrite-percentage

> 设置重写的基准值百分比

10. 常见配置redis.conf 介绍（总结）

```
# 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

# 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

# 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

# 绑定的主机地址
bind 127.0.0.1

# 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

# 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

# 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

# 设置数据库的数量，默认数据库为0，可以使用SELECT 命令在连接上指定数据库id
databases 16

# 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
# 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。

# 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

# 指定本地数据库文件名，默认值为dump.rdb
dbfilename dump.rdb

# 指定本地数据库存放目录
dir ./

# 设置当本机为slave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

# 当master服务设置了密码保护时，slave服务连接master的密码
masterauth

# 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 命令提供密码，默认关闭
requirepass foobared

# 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

# 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

# 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

# 指定更新日志文件名，默认为appendonly.aof
appendfilename appendonly.aof

# 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
```

```
#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

# 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

# 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

#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

# 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

# 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

# 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

# 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

# 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

# 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
```

三、Redis的持久化（RDB、AOF）

1. RDB

1. 官网介绍

> 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

> ### RDB advantages
>
> - RDB is a very compact single-file point-in-time representation of your Redis data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.
> - RDB is very good for disaster recovery, being a single compact file that can be transferred to far data centers, or onto Amazon S3 (possibly encrypted).
> - RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest. The parent instance will never perform disk I/O or alike.
> - RDB allows faster restarts with big datasets compared to AOF.
>
> ### RDB disadvantages
>
> - RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
> - RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

2. 是什么

> 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。

3. 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。

4. RDB保存的是dump.rdb文件

```
# The filename where to dump the DB
dbfilename dump.rdb
```

5. 配置位置

```
################################ SNAPSHOTTING
################################
#
# Save the DB on disk:
#
#   save <seconds> <changes>
#
#   Will save the DB if both the given number of seconds and the given
#   number of write operations against the DB occurred.
#
#   In the example below the behaviour will be to save:
#   after 900 sec (15 min) if at least 1 key changed
#   after 300 sec (5 min) if at least 10 keys changed
#   after 60 sec if at least 10000 keys changed
#
```

```
#   Note: you can disable saving completely by commenting out all "save"
lines.
#
#   It is also possible to remove all the previously configured save
#   points by adding a save directive with a single empty string argument
#   like in the following example:
#
#   save ""

save 900 1
save 300 10
save 60 10000

# By default Redis will stop accepting writes if RDB snapshots are enabled
# (at least one save point) and the latest background save failed.
# This will make the user aware (in a hard way) that data is not persisting
# on disk properly, otherwise chances are that no one will notice and some
# disaster will happen.
#
# If the background saving process will start working again Redis will
# automatically allow writes again.
#
# However if you have setup your proper monitoring of the Redis server
# and persistence, you may want to disable this feature so that Redis will
# continue to work as usual even if there are problems with disk,
# permissions, and so forth.
stop-writes-on-bgsave-error yes

# Compress string objects using LZF when dump .rdb databases?
# For default that's set to 'yes' as it's almost always a win.
# If you want to save some CPU in the saving child set it to 'no' but
# the dataset will likely be bigger if you have compressible values or
keys.
rdbcompression yes

# Since version 5 of RDB a CRC64 checksum is placed at the end of the file.
# This makes the format more resistant to corruption but there is a
performance
# hit to pay (around 10%) when saving and loading RDB files, so you can
disable it
# for maximum performances.
#
# RDB files created with checksum disabled have a checksum of zero that
will
# tell the loading code to skip the check.
rdbchecksum yes

# The filename where to dump the DB
dbfilename dump.rdb

# The working directory.
#
# The DB will be written inside this directory, with the filename specified
# above using the 'dbfilename' configuration directive.
#
# The Append Only File will also be created inside this directory.
#
# Note that you must specify a directory here, not a file name.
```

```
dir ./
```

6. 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，其他不管，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，但是里面是空的，无意义

7. 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，使用 config get dir 获取目录

8. 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

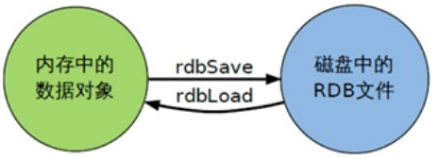
9. 劣势

- 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

10. 如何停止

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

11. 小总结



- RDB是一个非常紧凑的文件
- RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能.
- 与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些.

- 数据丢失风险大
- RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级不能相应客户端请求

2. AOF

1.官网介绍

> AOF advantages
>
> - Using AOF Redis is much more durable: you can have different fsync policies: no fsync at all, fsync every second, fsync at every query. With the default policy of fsync every second write performances are still great (fsync is performed using a background thread and the main thread will try hard to perform writes when no fsync is in progress.) but you can only lose one second worth of writes.
> - The AOF log is an append only log, so there are no seeks, nor corruption problems if there is a power outage. Even if the log ends with an half-written command for some reason (disk full or other reasons) the redis-check-aof tool is able to fix it easily.
> - Redis is able to automatically rewrite the AOF in background when it gets too big. The rewrite is completely safe as while Redis continues appending to the old file, a completely new one is produced with the minimal set of operations needed to create the current data set, and once this second file is ready Redis switches the two and starts appending to the new one.
> - AOF contains a log of all the operations one after the other in an easy to understand and parse format. You can even easily export an AOF file. For instance even if you flushed everything for an error using a FLUSHALL command, if no rewrite of the log was performed in the meantime you can still save your data set just stopping the server, removing the latest command, and restarting Redis again.
>
> AOF disadvantages
>
> - AOF files are usually bigger than the equivalent RDB files for the same dataset.
> - AOF can be slower than RDB depending on the exact fsync policy. In general with fsync set to *every second* performance is still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.
> - In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to not reproduce exactly the same dataset on reloading. These bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is fine. However, these kind of bugs are almost impossible with RDB persistence. To make this point more clear: the Redis AOF works by incrementally updating an existing state, like MySQL or MongoDB does, while the RDB snapshotting creates everything from scratch again and again, that is conceptually more robust. However - 1) It should be noted that every time the AOF is rewritten by Redis it is recreated from scratch starting from the actual data contained in the data set, making resistance to bugs stronger compared to an always appending AOF file (or one rewritten reading the old AOF instead of reading the data in memory). 2) We have never had a single report from users about an AOF corruption that was detected in the real world.

2. 是什么

> 以日志的形式来记录每个**写操作**，将redis 执行过的所有鞋指令记录下来（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

3. AOF保存的是appenonly.aof文件

4. 配置位置

```
############################## APPEND ONLY MODE
###################################

# By default Redis asynchronously dumps the dataset on disk. This mode is
# good enough in many applications, but an issue with the Redis process or
# a power outage may result into a few minutes of writes lost (depending on
# the configured save points).
#
# The Append Only File is an alternative persistence mode that provides
# much better durability. For instance using the default data fsync policy
# (see later in the config file) Redis can lose just one second of writes
in a
# dramatic event like a server power outage, or a single write if something
# wrong with the Redis process itself happens, but the operating system is
# still running correctly.
#
# AOF and RDB persistence can be enabled at the same time without problems.
# If the AOF is enabled on startup Redis will load the AOF, that is the
file
```

```
# with the better durability guarantees.
#
# Please check http://redis.io/topics/persistence for more information.

appendonly no

# The name of the append only file (default: "appendonly.aof")

appendfilename "appendonly.aof"

# The fsync() call tells the Operating System to actually write data on
disk
# instead of waiting for more data in the output buffer. Some OS will
really flush
# data on disk, some other OS will just try to do it ASAP.
#
# Redis supports three different modes:
#
# no: don't fsync, just let the OS flush the data when it wants. Faster.
# always: fsync after every write to the append only log. Slow, Safest.
# everysec: fsync only one time every second. Compromise.
#
# The default is "everysec", as that's usually the right compromise between
# speed and data safety. It's up to you to understand if you can relax this
to
# "no" that will let the operating system flush the output buffer when
# it wants, for better performances (but if you can live with the idea of
# some data loss consider the default persistence mode that's
snapshotting),
# or on the contrary, use "always" that's very slow but a bit safer than
# everysec.
#
# More details please check the following article:
# http://antirez.com/post/redis-persistence-demystified.html
#
# If unsure, use "everysec".

# appendfsync always
appendfsync everysec
# appendfsync no

# When the AOF fsync policy is set to always or everysec, and a background
# saving process (a background save or AOF log background rewriting) is
# performing a lot of I/O against the disk, in some Linux configurations
# Redis may block too long on the fsync() call. Note that there is no fix
for
# this currently, as even performing fsync in a different thread will block
# our synchronous write(2) call.
#
# In order to mitigate this problem it's possible to use the following
option
# that will prevent fsync() from being called in the main process while a
# BGSAVE or BGREWRITEAOF is in progress.
#
# This means that while another child is saving, the durability of Redis is
# the same as "appendfsync none". In practical terms, this means that it is
# possible to lose up to 30 seconds of log in the worst scenario (with the
# default Linux settings).
```

```
#
# If you have latency problems turn this to "yes". Otherwise leave it as
# "no" that is the safest pick from the point of view of durability.

no-appendfsync-on-rewrite no

# Automatic rewrite of the append only file.
# Redis is able to automatically rewrite the log file implicitly calling
# BGREWRITEAOF when the AOF log size grows by the specified percentage.
#
# This is how it works: Redis remembers the size of the AOF file after the
# latest rewrite (if no rewrite has happened since the restart, the size of
# the AOF at startup is used).
#
# This base size is compared to the current size. If the current size is
# bigger than the specified percentage, the rewrite is triggered. Also
# you need to specify a minimal size for the AOF file to be rewritten, this
# is useful to avoid rewriting the AOF file even if the percentage increase
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic AOF
# rewrite feature.

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb

# An AOF file may be found to be truncated at the end during the Redis
# startup process, when the AOF data gets loaded back into memory.
# This may happen when the system where Redis is running
# crashes, especially when an ext4 filesystem is mounted without the
# data=ordered option (however this can't happen when Redis itself
# crashes or aborts but the operating system still works correctly).
#
# Redis can either exit with an error when this happens, or load as much
# data as possible (the default now) and start if the AOF file is found
# to be truncated at the end. The following option controls this behavior.
#
# If aof-load-truncated is set to yes, a truncated AOF file is loaded and
# the Redis server starts emitting a log to inform the user of the event.
# Otherwise if the option is set to no, the server aborts with an error
# and refuses to start. When the option is set to no, the user requires
# to fix the AOF file using the "redis-check-aof" utility before to restart
# the server.
#
# Note that if the AOF file will be found to be corrupted in the middle
# the server will still exit with an error. This option only applies when
# Redis will try to read more data from the AOF file but not enough bytes
# will be found.
aof-load-truncated yes

# When rewriting the AOF file, Redis is able to use an RDB preamble in the
# AOF file for faster rewrites and recoveries. When this option is turned
# on the rewritten AOF file is composed of two different stanzas:
#
#   [RDB file][AOF tail]
#
# When loading Redis recognizes that the AOF file starts with the "REDIS"
# string and loads the prefixed RDB file, and continues loading the AOF
```

```
# tail.
aof-use-rdb-preamble yes
```

5. 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 xxx.aof 进行修复
- 恢复
  - 重新redis然后重新加载

6. rewrite

6.1 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文件的大小超过锁设定的阈值时，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

6.2 重写原理

AOF文件持续增长而过大时，会fork出一条新进程来讲文件重写（也是先写临时文件最后在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。

6.3 触发机制

Redis会记录上一次重写时的AOF大小，默认配置是当AOF文件大小是上传rewrite后大小的一倍且文件大于64M时触发

```
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

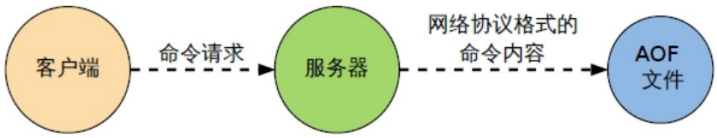
7. 优势

- 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较好
- 没秒同步：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

8. 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，每秒同步策略效果较好，不同步效果和rdb相同

9. 小总结



- AOF文件时一个只进行追加的日志文件
- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写
- AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂，对文件进行分析也很轻松

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积
- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB

3. 使用RDB还是AOF?

- 官网建议
- 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，reids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

  > 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的数据不时实，同时使用两者时服务器重启也只会找AOF文件。为何不单独使用AOF呢？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

四、Redis的事务

1. 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他命令插入，不许加塞。

2. 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

3. 怎么玩

3.1 常用命令

序号	命令及描述
1	DISCARD 取消事务，放弃执行事务块内的所有命令。
2	EXEC 执行所有事务块内的命令。
3	MULTI 标记一个事务块的开始。
4	UNWATCH 取消 WATCH 命令对所有 key 的监视。
5	WATCH key [key ...] 监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

3.2 执行事务可能出现的情况

> - 正常执行
> - 放弃事务
> - 全体连坐 [在入队时就已经报错了，EXEC时队列中的命令全部执行失败]
> - 冤头债主 [入队时没有错误，EXEC时执行失败，只会影响当前执行命令]
> - watch监控 [监控某个字段，CAS乐观锁思想]
>
>> 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
>>
>> ```
>> WATCH balance
>>
>> MUTIL
>>
>> set k1 v1
>>
>> EXEC
>> ```

4. 三阶段

> - 开启：以MUTIL开始一个事务
> - 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
> - 执行：由EXEC命令触发事务

5. 三特性

> - 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

- 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"这个让人头痛的问题
- 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。

五、Redis的发布订阅

1. 命令

序号	命令及描述
1	PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	SUBSCRIBE channel [channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

2. 案例

先订阅后发布 才能收到消息

- 可以一次性订阅多个，SUBSCRIBE c1 c2 c3

消息发送，PUBLISH c2 helloworld

- 订阅多个，通配符*，SUBSCRIBE new*

发送消息，PUBLISH new1 hello

六、Redis的复制(Master/Slave)

1. 是什么

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，Slaver以读为主**

2. 能干嘛

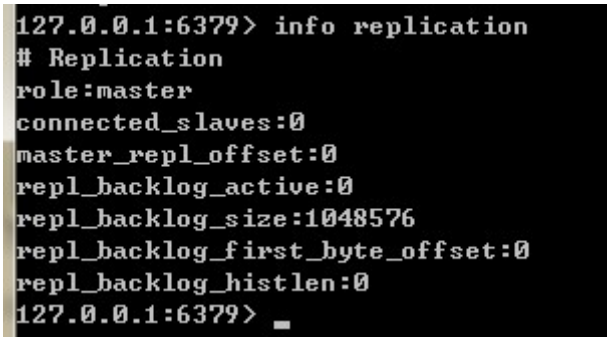
- 读写分离

- 容灾备份

3. 怎么玩

- **配从（库）不配主（库）**
- **从库配置：slaveof 主库ip 主库port**

  > **每次与master断开后，都需要重新连接，除非你配置进redis.conf文件，查看使用命令：**
  >
  > **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
- 常用三招
  - 一主二仆（一台主机两台备机）

    > 1. 主机宕机后，备机待命，主机后来后，恢复
    > 2. 备机宕机后，需要重新建立连接
    > 3. 备机全部依赖主机，主机压力大

  - 薪火相传（主机 -> 备机1-> 备机2...）

    > 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salveof 新主库ip 新主库port**

  - 反客为主

    > slaveof no one 使当前数据库停止与其他数据库的同步，转为主数据库

4. 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一次传给slave，完成同步**

> 但是只要是重新连接master，一次完全同步（全量同步）将被自动执行

5. 哨兵模式（sentinel）

5.1 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自动将从库转换为主库

5.2 怎么玩

- 自定义新建 sentinel.conf文件
- 配置哨兵，填写内容

```
port 26379
daemonize yes
logfile "26379.log"
dir "./"
sentinel monitor mymaster 192.168.250.132 6379 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 15000
sentinel auth-pass mymaster 123
bind 192.168.250.132 127.0.0.1
```

```
sentinel monitor <master-name> <ip> <redis-port> <quorum>
#告诉sentinel去监听地址为ip:port的一个master，这里的master-name可以自定义，quorum是一个数字，指明当有多少个sentinel认为一个master失效时，master才算真正失效

sentinel auth-pass <master-name> <password>
#设置连接master和slave时的密码，注意的是sentinel不能分别为master和slave设置不同的密码，因此master和slave的密码应该设置相同。

sentinel down-after-milliseconds <master-name> <milliseconds>
#这个配置项指定了需要多少失效时间，一个master才会被这个sentinel主观地认为是不可用的。 单位是毫秒，默认为30秒

sentinel parallel-syncs <master-name> <numslaves>
#这个配置项指定了在发生failover主备切换时最多可以有多少个slave同时对新的master进行 同步，这个数字越小，完成failover所需的时间就越长，但是如果这个数字越大，就意味着越 多的slave因为replication而不可用。可以通过将这个值设为 1 来保证每次只有一个slave 处于不能处理命令请求的状态。

sentinel failover-timeout <master-name> <milliseconds>
#failover-timeout 可以用在以下这些方面：
#1. 同一个sentinel对同一个master两次failover之间的间隔时间。
#2. 当一个slave从一个错误的master那里同步数据开始计算时间。直到slave被纠正为向正确的master那里同步数据时。
#3.当想要取消一个正在进行的failover所需要的时间。
#4.当进行failover时，配置所有slaves指向新的master所需的最大时间。不过，即使过了这个超时，slaves依然会被正确配置为指向master，但是就不按parallel-syncs所配置的规则来了。
```

5.3 一组sentinel能同时监控多个master

6. 复制的缺点

七、Redis的Java客户端Jedis

查看redis是否启动的方式：

- ps -ef | grep redis
- lsof -i :6379
- netstat -tnlp