# **1. 基础**

## **1.1 入门**

海量用户场景

高并发

12306

双11

罪魁祸首---关系型数据库

性能瓶颈：磁盘IO

扩展瓶颈：数据关系复杂，扩展性差，不便于大规模集群

解决思路：

降低磁盘IO次数，越低越好 --- 内部存储

去除数据间的关系，越简单越好 --- 不存储关系，仅存储数据

【Nosql的思路就是这样】

NoSQL：Not-Only SQL（泛指非关系型的数据库），作为关系型数据库的补充。

作用： 应对基于海量用户和数据前提下的数据处理。

特征：

可扩容，可伸缩

大数据量下的高性能

灵活的数据模型

高可用

常见的Nosql数据库：

Redis

memcache

HBase

MongoDB

一个电商的场景：

商品的基本信息：名称、价格、厂商 --- MySQL

商品的附加信息：描述、详情、评论 --- MongoDB

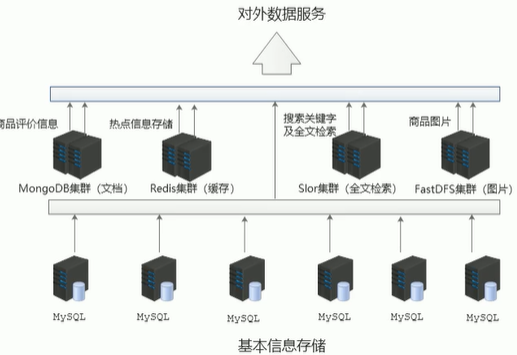
图片信息： 分布式文件系统

搜索关键字：ES、Lucene、solr

热点信息：

高频

波段性  --- Redis、memcache、tair



### 1.1.1 简介

Redis（Remote Dictionary Server）是用C语言开发的一个开源的高性能键值对（key-value）数据库。

特征：

1. 数据之间没有必然的关联关系

2. 内部采用单线程机制工作

3. 高性能

4. 多数据类型支持

字符串类型：string

列表类型： list

散列类型： hash

集合类型： set

有序集合类型： sorted\_set

5. 持久化支持。可以进行数据灾难恢复

应用：

1. 为热点数据加速查询，如热点商品、热点新闻、推广类信息等高访问信息等。

2. 任务队列，如秒杀、抢购、购票排队等

3. 即时信息查询，如排行榜、访问统计、公交到站信息、在线人数、设备信号等

4. 时效性信息控制，如验证码、投票控制等。

5. 分布式的数据共享，如分布式集群架构中的session分离

6. 消息队列

7. 分布式锁

### 1.1.2 下载和安装

linux

windows

https://github.com/microsoftarchive/redis/releases

Redis-x64-3.2.100.zip   绿色免安装

redis-benchmark.exe 性能测试工具

redis-check-aof.exe AOF文件修复工具

redis-check-dump.exe RDB文件检查工具（快照持久化文件）

redis-cli.exe       命令行客户端

redis-server.exe    服务端启动命令

redis.windows.conf  核心配置文件

redis.windows-service.conf

端口默认 6379

### 1.1.3 基本操作

```java

信息添加

set key value

$ set name aa

信息查询，如果不存在，返回nil

get key

$ get name

清除

$ clear

帮助

$ help set

$ help @string

退出命令行

$ quit

$ exit

<ESC>

```

## 1.2 数据类型

业务数据的特殊性

1. 原始业务功能设计

秒杀

618

双11

排队购票 12306

2. 运营平台监控到的突发高频访问数据

突发热点新闻

3. 高频、复杂的统计数据

在线人数

投票排行

4. 附加功能

系统功能优化或升级

单服务器升级集群

session管理

token管理

针对以上情况，提供5种常用数据类型

key永远是字符串

记录操作结果，1成功，0失败

### 1. string

如果value以整数形式展示，可以作为数字操作使用

基本操作

```java

$ set key value

$ get key

$ del key

添加/修改多个数据【multiple】

$ mset key1 value1 key2 value2

获取多个数据

$ mget key1 key2

获取数据字符长度

$ strlen key

追加信息（存在就追加，否则新建）

$ append key value

```

单数据操作和多数据操作

假如多个指令一次性执行耗时很久，就必然要对指令进行切割，进行多次执行，防止阻塞。

选择如何执行要看实际场景。

扩展操作

比如针对Mysql多个分表的主键生成

解决方案：

```java

设置数值增加指定范围的值

$ incr key

$ incrby key increment

$ incrbyfloat key increment

设置数值减少指定范围的值

$ decr key

$ decrby key increment

```

increment 可以是负数

string默认是字符串，遇到增减类操作incr\decr时会转成数值型进行计算。

Redis所有的操作都是原子性的，采用单线程处理所有业务，命令是一个个执行，因此无需考虑并发的问题。

注意：

按照数值进行操作的数据，如果数据不能转成数值，或者超过了redis数值上限，报错

【java long的最大值 Long.MAX\_VALUE】

Tips：

redis用于控制数据库表主键id，保证了主键的唯一性。

这个方案适用于所有的数据库，支持数据库集群。

扩展操作

热点数据的时效性控制

```java

设定数据的生命周期

$ setex key seconds value

$ psetex key milliseconds value

```

Tips：

redis控制数据的生命周期，通过数据是否失效控制业务，适用于时效性限定的操作

注意事项：

1.表示运行结果是否成功

0 --》 false 失败

1 --》 true  成功

2. 表示运行结果值

3 --> 3个

1 --》 1个

3. 数据未获取到

(nil) 等同于 null

4. 数据最大存储大小

512M

5. 数值计算最大范围

java的long.MAX\_VALUE

应用：

weibo的粉丝数量、微博数量

$ set user:id:12434:fans 1232

$ set user:id:12231:blogs 2342

$ set user:id:12312:focus 23

也可以用json格式

$ set user:id:12132 {id:q2312,name:春,fans:21312,blogs:3123,focus:243}

Tips:

redis应用于各种结构型和非结构型高热度数据访问加速

key的设置约定：

1. 数据表的热点数据key命名规范

表名:主键名:主键值:字段名

order:id:3242342:name

order:id:3242342:type

order:id:3242342:title

### 2. hash

对一系列存储的数据进行编组，方便管理，存储对象信息。

一个存储空间保存多个键值对数据。

底层使用哈希表结构实现数据存储。

存储结构优化：

如果field数量少，底层存储优化为类数组结构

如果field数量多，底层存储用hashMap结构

基本操作

```java

添加、修改

$ hset key field value

获取

$ hget key field

$ hgetall key

删除

$ hdel key field1 [field2 ...]

多个操作

$ hmset key field1 value1 field2 value2 ...

$ hmget key field1 field2 ...

获取字段数量

$ hlen key

是否存在指定的字段

$ hexists key field

```

扩展操作

```java

获取所有字段或者值

$ hkeys key

$ hvals key

设置指定字段的数值增加指定范围的值

$ hincrby key field increment

$ hincrbyfloat key field increment

存在field就跳过，否则添加键值对

$ hsetnx key field value

```

注意事项：

1. value只能存储字符串，不允许其它数据类型，不能嵌套。数据不存在，返回(nil)

2. 每个hash可以存储2^32-1个键值对

3. 贴近对象的存储形式，可以灵活删除对象属性和值。不可滥用，不可以将hash当做对象列表使用

4. hgetall可以获取全部属性，如果field过多，效率变低，可能称为性能瓶颈

string存储json和hash存储对象的选择根据场景选择。

### 3. list

存储多个数据，对数据进入的顺序进行区分

底层使用双向链表结构实现。

基本操作：

```java

添加、修改

$ lpush key value1 [value2 ...]

$ rpush key value1 [value2 ...]

获取数据

$ lrange key start end

$ lindex key index

$ llen key

获取并移除

$ lpop key

$ rpop key

$ lrange list1 0 -1

$ lrange list1 0 2

```

扩展操作：

```java

阻塞获取

规定时间内获取并删除 block

$ blpop key1 [key2] timeout

$ brpop key1 [key2] timeout

移除指定数据

$ lrem key count value

```

应用：

具有先后顺序的数据控制

注意事项：

1. list保存的都是string类型，容量上限2^32-1

2. 具有索引的概念，通常以队列的形式或者栈的形式操作

3. 获取全部数据的结束索引 -1

4. 可以对数据进行分页操作，第一页存list，后面连数据库查询

### 4. set

存储大量数据，高效的内部存储机制，便于查询。

与hash存储结构完全相同，仅存储键，不存储值(nil)，并且值不允许重复。

基本操作

```java

添加

$ sadd key member1 [member2 ...]

获取

$ smembers key

删除

$ srem key member1 [member2 ...]

获取总量

$ scard key

判断是否包含

$ sismember key member

```

扩展操作

```java

随机获取指定数量的数据

$ srandmember key [count]

随机获取某个数据并移除集合

$ spop key

```

可以应用于随机推荐类信息检索

扩展操作

```java

交、并、差

$ sinter key1 [key2]

$ sunion key1 [key2]

$ sdiff key1 [key2]

求交、并、差之后存储到指定集合

$ sinterstore destination key1 [key2]

$ sunionstore destination key1 [key2]

$ sdiffstore destination key1 [key2]

将指定数据从原始集合中移动到指定目标集合

$ smove source destination member

```

可用于同类信息的关联检索，二度关联搜索，深度关联搜索

显式共同关注

共同好友，兴趣等

redis提供基础数据还是检验结果？

推荐只提供基础数据，检验放到代码里进行。

Tip:

redis应用于同类型不重复数据的合并操作

应用：

统计网站

PV 访问量，可以通过刷新页面

UV 独立访客，可以通过cookie统计访问量，相同用户切换ip，UV不变

IP 独立IP，可以通过IP地址统计访问量，相同IP不同用户访问，IP不变

网站防爬虫

做ip黑白名单

Tips:

redis应用于同类型数据的快速去重。

redis应用于黑名单和白名单设定的服务控制。

### 5. sorted\_set

保存可排序的数据。

在set的存储基础上添加可排序的字段

默认排序从小到大

基本操作

```java

添加数据

$ zadd key score1 member1 [score2 member2 ...]

获取全部数据

$ zrange key start end [WITHSCORES]

$ zrevrange key start end [WITHSCORES]

删除数据

$ zrem key member1 [member2 ...]

$ zrange key 0 -1

$ zrange key 0 -1 withscores

按条件获取数据

$ zrangebyscore key min max [WITHSCORES] [LIMIT]

$ zrevrangebyscore key max min [WITHSCORES]

按条件删除数据

$ zremrangebyrank key start end

$ zremrangebyscore key min max

$ zrangebyscore key 2 99 limit 0 3 withscores

获取集合数据总量

$ zcard key

$ zcount key min max

集合交、并

$ zinterstore destination numkeys k1 k2 [k3...]

$ zunionstore destination numkeys k1 k2 [k3...]

还有其它参数，求MAX\MIN

$ help zinterstore

$ help zunionstore

```

扩展操作

```java

获取数据对于的索引(排名)

$ zrank key member

$ zrevrank key member

score的获取和修改

$ zscore key member

$ zincrby key increment member

redis应用于计数器组合排序对应的排名

```

注意：

1. min,max用于查询条件的范围参数

2. start,end用于查询索引范围参数

3. offset,count用于限定查询范围，作用于查询结果，表示开始位置和数据总量

注意事项

1. score的存储空间是64位，范围有限制

2. score也可以是一个双精度的double值，可能会丢失精度，使用慎重

3. sorted\_set和set一样，数据不能重复

获取当前系统时间

$ time

应用场景：

过期VIP通知

任务队列/消息队列权重设定应用

Tips:

redis应用于定时任务执行顺序管理或者任务过期管理【通过score设置，score可以是时间戳等】

### 实践案例

1. API调用次数限制，比如一个ip每小时5000次调用

redis应用于限时按次结算的服务控制

2. 微信消息排序

redis应用于基于时间排序的数据操作

## 1.3 通用命令

### key通用指令

```java

删除指定key

$ del key

获取key是否存在

$ exists key

获取key类型

$ type key

为指定key设置有效期

$ expire key seconds

$ pexpire key milliseconds

$ expireat key timestamp

$ pexpireat key milliseconds-timestamp

获取key的有效时间(-1 -2 1 2)

$ ttl key

$ pttl key

切换key从时效性转为永久性

$ persist key

查看key

$ keys pattern

$ keys \*

查看模式规则

\* 匹配任意数量的任意字符

? 匹配一个任意符号

[] 匹配一个指定符号

$ keys \*

$ keys aa\*

$ keys \*zz

$ keys ??aa

$ keys aa:?

$ keys aa[bc]ee:1

修改key名称

$ rename key newkey

对所有key排序

$ sort

其它key通用操作

$ help @generic

$ help set

```

### 数据库通用指令

redis为每个服务提供16个数据库，编号从0-15

每个数据库之间的数据互相独立。

```java

切换数据库

$ select index

$ quit

$ ping

$ echo message

数据移动

$ move key db

数据清除

$ dbsize

$ flushdb

$ flushall

```

## 1.4 Jedis操作

java连接redis

jedis、SpringData Redis、letture

其它语言有自己的实现

### 基础操作

```java

import org.junit.Test;

import redis.clients.jedis.Jedis;

import java.util.List;

import java.util.Map;

public class JedisTest {

    @Test

    public void testHash() {

        // 1. 连接redis

        Jedis jedis = new Jedis("127.0.0.1", 6379);

        // 2. 操作redis

        jedis.hset("hash1", "k1", "v1");

        jedis.hset("hash1", "k2", "v2");

        Map<String, String> hash1 = jedis.hgetAll("hash1");

        for (Map.Entry<String, String> entry : hash1.entrySet()) {

            System.out.println(entry.getKey() +":" +entry.getValue());

        }

        System.out.println(jedis.hlen("hash1"));

        // 3. 关闭连接

        jedis.close();

    }

    @Test

    public void testList() {

        // 1. 连接redis

        Jedis jedis = new Jedis("127.0.0.1", 6379);

        // 2. 操作redis

        jedis.lpush("list1", "a", "b", "c");

        jedis.rpush("list1", "x");

        List<String> list1 = jedis.lrange("list1", 0, -1);

        list1.stream().forEach(System.out::print);

        System.out.println(jedis.llen("list1"));

        // 3. 关闭连接

        jedis.close();

    }

    @Test

    public void testString() {

        // 1. 连接redis

        Jedis jedis = new Jedis("127.0.0.1", 6379);

        // 2. 操作redis

        jedis.set("name", "whh");

        String v1 = jedis.get("name");

        System.out.println(v1);

        // 3. 关闭连接

        jedis.close();

    }

}

```

```java

import redis.clients.jedis.Jedis;

import redis.clients.jedis.exceptions.JedisDataException;

import java.util.concurrent.TimeUnit;

public class RedisMain1 {

    public static void main(String[] args) {

        ServiceThread t = new ServiceThread("初级用户", 10);

        t.start();

    }

    static class Service {

        private String name;

        private int c;

        Service(String name, int c) {

            this.name = name;

            this.c = c;

        }

        void service() {

            Jedis jedis = new Jedis("127.0.0.1", 6379);

            String key = "user:" + name;

            String value = jedis.get(key);

            try {

                if (value == null) {

                    jedis.setex(key, 20, Long.MAX\_VALUE-c+"");

                } else {

                    long l1 = jedis.incr(key);

                    f1(key, c-(Long.MAX\_VALUE-l1));

                }

            } catch (JedisDataException ex) {

                System.out.println("到达使用次数, 升级VIP");

                return;

            } finally {

                jedis.close();

            }

        }

        void f1(String key, long val) {

            System.out.println(key + "执行业务第" + val + "次");

            try {

                TimeUnit.SECONDS.sleep(1);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

    static class ServiceThread extends Thread {

        Service service;

        String name;

        int c;

        ServiceThread(String name, int c) {

            service = new Service(name, c);

        }

        @Override

        public void run() {

            while (true) {

                service.service();

            }

        }

    }

}

```

### 连接池

JedisPool：

连接池

poolConfig

host

port

```java

import redis.clients.jedis.Jedis;

import redis.clients.jedis.JedisPool;

import redis.clients.jedis.JedisPoolConfig;

import java.util.ResourceBundle;

public class JedisUtil {

    private static JedisPool pool = null;

    private static String host;

    private static int port;

    private static int maxTotal;

    private static int maxIdle;

    static {

        ResourceBundle rb = ResourceBundle.getBundle("redis");

        host = rb.getString("redis.host");

        port = Integer.parseInt(rb.getString("redis.port"));

        maxTotal = Integer.parseInt(rb.getString("redis.maxTotal"));

        maxIdle = Integer.parseInt(rb.getString("redis.maxIdle"));

        JedisPoolConfig config = new JedisPoolConfig();

        config.setMaxTotal(maxTotal);

        config.setMaxIdle(maxIdle);

        pool = new JedisPool(config, host, port);

    }

    static Jedis getPool() {

        return pool.getResource();

    }

    public static void main(String[] args) {

        Jedis jedis = getPool();

        jedis.set("k1", "v1");

        System.out.println(jedis.get("k1"));

    }

}

```

redis.properties

**redis.host**=**127.0.0.1  
redis.port**=**6379  
redis.maxTotal**=**30  
redis.maxIdle**=**10**

### 可视化客户端

Redis Desktop Manager

## centos7安装redis

```

$ wget http://download.redis.io/releases/redis-4.0.0.tar.gz

$ tar -xvf redis-4.0.0.tar.gz

$ cd redis-4.0.0

$ make

$ make install

$ cd src

$ redis-server

$ redis-cli

$ redis-server --port 6380

$ redis-cli -p 6380

$ redis-server redis-6380.conf

$ redis-cli -h 127.0.0.1

$ redis-cli -h 127.0.0.1 -p 6380

```

# 2. 高级

## 2.1 持久化

自动备份

内存 <---> 硬盘

保存和恢复

是么是持久化？

利用永久性存储介质将数据进行保存，在特定的时间将保存的时间进行恢复的工作机制称为持久化。

为什么要进行持久化？

防止数据的意外丢失，确保数据安全。

持久化保存什么？

1. 将当前数据状态进行保存，快照形式，存储数据结果，存储格式简单，关注点在数据 【RDB】

2. 将数据的操作过程进行保存，日志形式，存储操作过程，存储格式复杂，关注点在数据的操作过程 【AOF】

### RDB

redis用户可以随时发RDB命令来保存数据快照。

RDB启动方式

```java

手动执行一次保存操作

$ save

./data/dump.rdb

```

save命令相关配置

1. dbfilename dump.rdb

设置本地数据库文件名，默认dump.rdb，通常设置为dump-端口号.rdb

2. dir

存储rdb文件的路径，通常设置成存储空间大的目录里，默认安装目录的data

3. rdbcompression yes

设置存储到本地数据库时是否压缩数据，默认yes，采用LZF压缩。

通常默认是开启，如果设置为no，可以节省CPU性能，但是压缩文件变大（巨大）。

4. rdbchecksum yes

设置是否进行rdb文件格式校验，校验过程在读写文件时均进行。

通常默认是开启，如果设置为no，可以节省读写10%的性能损耗，但是有存储损坏数据的风险

```java

./conf/redis-6379.conf

port 6379

daemonize yes

logfile "6379.log"

dir /redis-4.0.0/data

dbfilename dump-6379.rdb

rdbcompression yes

rdbchecksum yes

$ kill -s 9 pid

```

redis服务重新启动后会根据配置从rdb文件中恢复数据。

save指令工作原理

save指令的执行会阻塞当前redis服务器，直到当前RDB过程执行完毕，有可能会造成长时间阻塞，线上环境不建议使用。【redis是单线程任务执行序列】

针对上面的情况，提供bgsave命令

```java

手动启动后台保存，但不是立即执行

$ bgsave

Background saving started

```

bgsave指令工作原理

指令发送到redis服务端后，立刻返回Background saving started，但是指令不会立即执行，redis会调用系统的fork函数生成子进程来创建rdb文件。

可以查看.log文件观察bgsave指令的执行情况。

注意：

bgsave是对save的优化。Redis内部所有涉及到RDB的操作都采用bgsave的方式，save可以放弃使用。

bgsave相关配置

```

./conf/redis-6379.conf

port 6379

daemonize yes

logfile "6379.log"

dir /redis-4.0.0/data

dbfilename dump-6379.rdb

rdbcompression yes

rdbchecksum yes

stop-writes-on-bgsave-error yes

```

stop-writes-on-bgsave-error yes

后台存储过程中出现错误，是否停止保存操作，通常默认为开启。

RDB自动保存

```java

在conf文件中配置

save second changes

满足限定时间范围内key的变化数量达到指定数量即进行持久化。

second：监控时间范围

changes：监控key的变化量

比如

save 900 1

save 300 10

save 60 10000

```

变化量增加有几个场景：

1. 会对数据产生影响

2. 真正产生了影响

3. 不进行数据比对

注意：

save配置要根据实际业务情况进行设置，频度过高或者过低都会出现性能问题，结果可能是灾难性的。

save配置中对于second和changes设置通常具有互补关系，尽量不要设置为包含关系。

save配置启动后执行的是bgsave操作

RDB启动方式对比

方式        save        bgsave

读写        同步        异步

阻塞客户端指令  是      否

额外消耗内存    否      是

启动新进程      否      是

RDB特殊启动方式

```java

1. 全量复制

在主从复制中

2. 服务器运行过程中重启

$ debug reload

3. 关闭服务时指定保存数据

$ shutdown save

```

RDB优点

1. RDB是一个紧凑压缩的二进制文件，存储效率高

2. RDB内部存储是redis在某个时间点的数据快照，非常适合数据备份，全量数据复制等场景

3. RDB恢复数据的速度要比AOF快很多

4. 服务器中每X小时执行bgsave备份，并将RDB文件拷贝到远程机器中，用于灾难恢复。

RDB缺点

1. RDB方式无论是执行还是配置，无法做到实时持久化，有较大丢数据的可能。

2. bgsave每次运行要fork创建子进程，有性能损耗

3. 不同版本rdb文件格式不统一，可能出现不同版本数据不兼容

### AOF

针对RDB存储的弊端，有以下解决思路：

1. 不写全量数据，仅记录部分

2. 改记录数据为记录操作过程

3. 对所有的操作均进行记录，排除丢失数据的风险

AOF概念

AOF（append only file）持久化：

以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中的命令以达到恢复数据的目的。与RDB相比可以简单描述为改记录数据为记录数据产生的过程。

AOF的主要作用是解决了数据持久化的实时性，目前已经是redis持久化的主流方式。

AOF写数据全过程：

client的命令  --> Redis服务器，接收到指令后，先不执行，首先将命令缓存到“AOF写命令刷新缓存区” -->然后根据不同的写策略将命令写到 .aof文件中。

AOF写数据3种策略（appendfsync）:

1. always

每次写入操作均同步到aof中，数据零误差，性能较低，不建议使用

2. everysec

每秒将缓存区的指令同步到aof中，数据准确性较高，性能较高，建议使用，也是默认配置

3. no

由操作系统控制每次同步aof文件的周期，整体过程不可控

AOF功能开启

```java

配置

appendonly yes|no

作用： 是否开启AOF，默认不开启

appendfsync always|everysec|no

作用：AOF写数据策略

appendfilename filename

作用：默认文件名appendonly.aof，建议配置为appendonly-端口.aof

dir

作用：aof文件路径，与rdb文件保持一致

```

AOF重写

aof文件会越来越大，redis引入了AOF重写机制压缩文件体积。AOF文件重写是将redis进程内的数据转化为写命令同步到新aof文件的过程。

简单就是对同一个数据的若干条命令执行结果转化为最终结果数据对应的指令进行记录。

AOF重写作用

1. 降低磁盘占用量，提供磁盘利用率

2. 提高持久化效率，降低持久化写时间，提供IO性能

3. 降低数据恢复用时，提高恢复效率

AOF重写规则

1. 进程内已超时的数据不再写入文件

2. 忽略无效指令，重写时使用进程内数据直接生成，这样新的aof文件只保留最终数据的写入命令

如set k1 1, set k1 2 --> set k1 2

3. 对同一数据的多条写合并为一条命令

如 lpush k1 1, lpush k1 2 --> lpush k1 1 2

为防止数据量过大造成客户端缓存溢出，对list,set,hash,zset等类型，限制每条指令最多写入64个元素。

AOF重写方式

1. 手动重写

```

$ bgrewriteaof

```

bgrewriteaof指令工作原理

指令发送到redis服务端后，立刻返回Background append only file rewriting started。但是指令不会立即执行，redis会调用系统的fork函数生成子进程来重写aof文件。

2. 自动重写

```配置

auto-aof-rewrite-min-size size

auto-aof-rewrite-percentage percentage

```

自动重写触发比对参数（运行执行info Persistence 获取）

$ info

$ info Persistence

aof\_current\_size

aof\_base\_size

自动触发条件

aof\_current\_size > auto-aof-rewrite-min-size

(aof\_current\_size - aof\_base\_size) / aof\_base\_size >= auto-aof-rewrite-percentage

重写后台都是调用bgrewriteaof来实现。

### RDB和AOF区别

持久化方式              RDB             AOF

占用存储空间        小（数据级：压缩）    大（指令级：重写）

存储速度            慢                  快

恢复速度            快                  慢

数据安全性          会丢失数据           依据写策略决定

资源消耗            高/重量级            低/轻量级

启动优先级          低                   高

### RDB和AOF的选择

1. 对数据敏感，建议使用默认的AOF方案

如果策略用everysec，有较好的性能，出现问题时，最多丢失0-1s内的数据

由于aof文件很大，恢复速度慢

2. 数据呈现阶段有效性，建议使用RDB方案

数据可以良好的做到阶段内无丢失（开发或者运维维护），恢复速度快

利用rdb实现紧凑的持久化会降低redis性能

3. 综合对比

rdb和aof的选择根据实际业务场景选用

如果不能承受数据丢失，选用aof

如果能承受时间段内的数据丢失，且追求大数据集的恢复速度，选用rdb

灾难恢复选用rdb

双保险策略，同时开启rdb和aof，重启后，redis优先使用aof来恢复数据，降低丢失数据的风险

### 持久化应用场景

视业务来定是否需要进行持久化。

## 2.2 事务

场景：执行指令的过程中国，多条连续执行的指令被干扰、打断、插队。

Redis事务就是执行一个命令队列，将一系列命令包装成一个整体。当执行时，一次性按照队列的顺序依次执行，中间不会被打断或者干扰。

一句话：一个队列中，一次性、顺序性、排他性的执行一些列指令。

事务的边界

multi <----> exec

事务的基本操作：

```java

开启事务

$ multi

作用：设定事务的开启，执行后，后续所有指令加入到事务中

执行事务

$ exec

作用：设定事务的结束位置，同时执行事务。和multi配合使用

取消事务

$ discard

作用：终止当前事务定义，发生在multi之后，exec之前。

```

注意：

加入事务的命令没有立即执行，只有exec之后才开始执行。

事务的工作流程

服务端接收到指令，先判断是否处在开启事务状态，如果有就把命令放入到事务队列中

注意事项：

1. 定义事务过程中，命令输入错误

如果事务队列中包含错误的指令，整体事务所有命令都不执行。

2. 定义事务过程中，命令执行错误

正确的命令会执行，运行错误的命令不执行

注意：已经执行完毕的命令的结果是不会回滚的，需要程序自己在代码中实现回滚

### 锁

基于特定条件的事务执行。

```

对key添加监视锁，exec前如果key发生了变化，终止事务运行

$ watch key1 [key2 ...]

取消对所有key的监视

$ unwatch

```

注意：

必须在开启事务之前watch。

Tips:

redis可以应用基于状态控制的批量任务执行。

### 分布式锁

基于特定条件的事务执行

```

使用setnx设置一个公共锁

$ setnx lock-key value

$ set num 10

$ setnx lock-num 1

$ incrby num -1

$ del lock-num

```

利用setnx命令的返回值特征，有值则返回设置失败，无值则返回设置成功

1. 对于设置成功的，对key有操作权

2. 对于设置失败的，对key没有操作权

操作完毕后，通过del操作释放锁

分布式锁是一种设计概念，依赖规范保障，具有风险性。

Tips:

Redis应用基于分布式锁对应的场景控制。

在分布式锁机制下，一个用户获取锁之后宕机了，如何解决？

存在加锁后未解锁的风险，所以加锁操作不能仅依赖用户控制，系统级别也要有保底的处理方案。

### 分布式锁改良

```

使用expire为锁key添加时间限定

$ expire lock-key second

$ pexpire lock-key milliseconds

$ set num 10

$ setnx lock-num 1

$ expire lock-num 10

$ get name

$ del lock-num

```

由于操作都是很快的，因此锁的设定时间不宜过大，具体时间根据业务测试后确认。

## 2.3 删除策略

【只针对具有时效性的数据】

redis是一种内存型的数据库，所有数据都存在内存中，内存中的数据状态可以通过TTL指令获取状态：

1. XX : 具有时效型的数据

2. -1 : 永久有效的数据

3. -2 : 已经过期的数据或者被删除的数据或者未定义的数据

问题：

过期的数据真的被删除了吗？

过期数据

设置时效的数据到达失效期后，并没有被立即删除。根据删除策略不同有不同的处理。

时效性数据的存储结构

setex --> expire    ---> key

          expireat  ---> key

          pexpire   ---> key

          pexpireat ---> key

Redis存储空间

key1     --->        value1 (0x0110)

key2     --->        value2 (0x3423)

redis会开辟一个expires空间，用来保存时效性key对应的value内存地址对应的过期时间

expires

0x0110  ---->  12565656

0x3423  ---->  15454532

数据删除策略的目标：

在内存占用和CPU占用之间寻找一种平衡，顾此失彼都会造成redis性能的下降，甚至引发宕机或者内存泄漏。

### 数据删除策略

有3种：

#### 1. 定时删除

创建一个定时器，当有key的过期时间到达时，由定时器任务立即执行对过期数据的删除。

优点：节省内存，到时就删除，快速释放不必要的内存占用

缺点：CPU压力很大，无论CPU此时负载多高，均占用CPU，会影响redis服务响应时间和指令吞吐量

总结：用处理器性能换取存储空间（时间换空间）

#### 2. 惰性删除

数据过期后，不做处理。等下次访问时

如果未过期，返回数据

如果已经过期，删除，返回不存在 （exprieIfNeeded() <---> get key）

优点：节约CPU性能，发现必须删除的时候才删除

缺点：内存压力大，出现长期占用内存的数据

总结：用存储空间换取处理器性能（空间换时间）

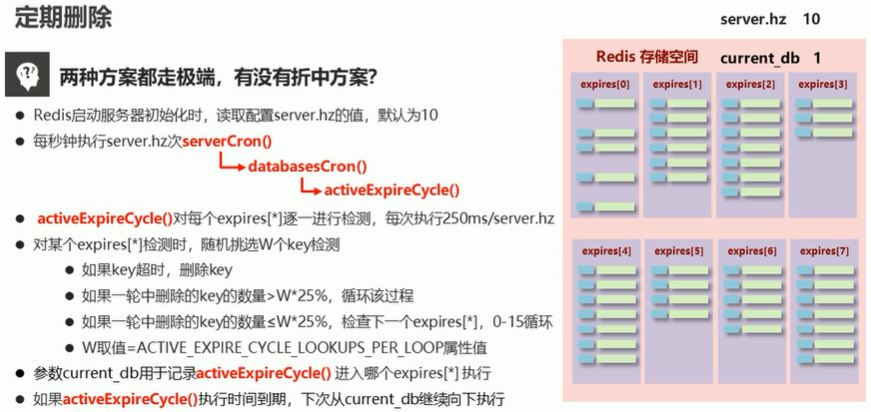
#### 3. 定期删除

周期性轮询redis库中的时效性数据，采用随机抽取的策略，利用过期数据占比的方式控制删除频度。

特点1：CPU性能占用设置有峰值，检测频度可自定义设置

特点1：内存压力不是很大，长期占用内存的冷数据会被持续清理

总结：周期性抽查存储空间（随机抽查，重点抽查）



$ info server

server.hz 10

current\_db 0

#### 删除策略对比

1.定时：节约内存，无占用。不分时段占用CPU资源，频度高。拿时间换空间。

2.惰性：内存占用严重。延时执行，CPU利用率高。拿空间换时间。

3.定期：内存定期随机清理。每秒花费固定CPU资源维护内存。随机抽查，重点抽查。

### 逐出算法（淘汰算法）

【8种策略】

问题：

当新数据进入redis时，如果内存不足怎么办？

在执行一个命令之前，都会调用freeMemoryIfNeeded()检测内存是否充足。如果内存不满足新加入数据的最低存储要求，redis要临时删除一些数据来清理存储空间。清理数据的策略称为逐出算法（淘汰算法）。

注意：

逐出数据的过程不是100%能够清理出足够可使用的空间。如果不成功则返回执行。当对所有数据尝试完毕后，仍然不能达到内存清理的要求，报错:

(error) OOM command not allowed when used memory>'maxmemory'

影响数据逐出的配置

1. 最大可使用内存

maxmemory

占用物理内存的比例，默认是0，表示不限制。生产环境根据需求设定，通常设置50%以上。

2. maxmemory-samples

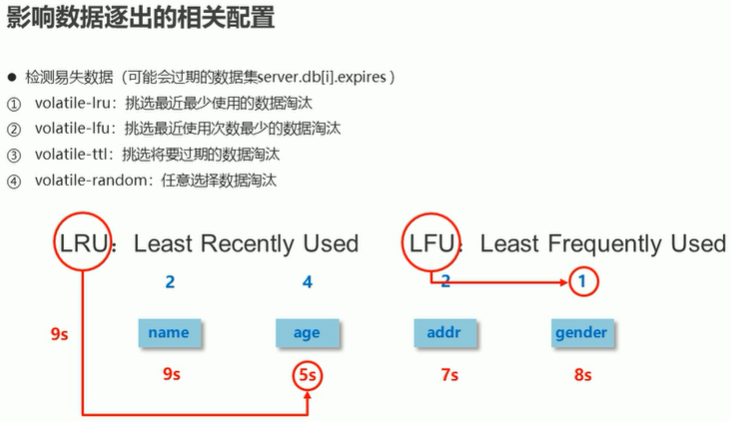
每次选取待删除数据的个数

选取数据时并不会全库扫描，导致严重的性能损耗，降低读写性能。因此采用随机获取数据的方式作为待检测数据

3. maxmemory-policy

删除策略

到达最大内存后，对被选出来的数据进行删除的策略



检测全库数据（所有数据集server.db[i].dict）

5. allkeys-lru： 挑选最近最少使用的数据淘汰

6. allkeys-lfu： 挑选最近使用次数最少的数据淘汰

7. allkeys-random： 任意选择数据淘汰

放弃数据淘汰

8. no-enviction（驱逐）： 禁止驱逐数据（Redis4.0默认策略），会引发OOM。

maxmemory-policy volatile-lru

数据逐出策略配置依据

使用info输出监控信息，查询缓存 hit 和 miss 次数，根据业务需求调优redis配置。

### redis.conf

服务器基础配置

```

设置服务器以守护进程的方式启动

daemonize yes|no

绑定主机地址

bind 127.0.0.1

设置服务器端口号

port 6379

设置数据库数量

databases 16

```

日志配置

```

设置服务器以指定日志记录级别

loglevel debug|verboase|notice|warning

日志记录文件名

logfile port.log

```

注意：

日志级别开发期设置为verbose即可，生产环境中配置为notice，简化日志输出量，降低写日志io的频度。

客户端配置

```

设置同一时间最大客户端连接数，默认无限制。当达到限制上限，redis会关闭新的连接

maxclients 0

客户端闲置等待最大时长，达到后关闭连接。如果需要关闭该功能，设置为0

timeout 300

```

多服务器快捷配置

```

导入并加载指定配置文件信息，用于快速创建redis公共配置较多的redis实例配置文件，便于维护

include /path/port.conf

```

### 高级数据类型

Bitmaps

存储需求：

一些纯状态的值，1\0，用8位00000000 string就可以存储，不过需要对位进行操作

基础操作

```

获取指定key对应偏移量上的bit值

$ getbit key offset

设置指定key对应偏移量上的bit值，vlaue只能是1或者0

$ setbit key offset value

```

扩展操作

```

对指定key按位进行交、并、非、异或操作，并将结果保存到destkey中

$ bitop op destkey key1 [key2 ...]

and

or

not

xor

统计指定key中1的数量

$ bitcount key [start end]

```

Tips

redis应用于信息状态统计

HyperLogLog

作用：统计不重复数据的数量

例如：

统计独立UV

原始方案：set

存储每个用户的id（字符串）

改进方案：Bitmaps

存储每个用户的状态（bit）

全新方案：HyperLogLog

基数：

基数是数据集去重后元素个数

HyperLogLog是用来做基数统计的。运用了LogLog算法。

基本操作

```

添加数据

$ pfadd key element [element1...]

统计数据

$ pfcount key [key1...]

合并数据

$ pfmerge destkey sourcekey [sourcekey1...]

```

Tips

redis应用于数据统计

相关说明

用于进行基数统计，不是集合，不保存数据，只记录数量而不是具体数据

核心是基于估算算法，最终数值存在一定的误差

误差范围：基数估计的结果是一个带有0.81%标准错误的近似值

耗空间极小，每个hyperloglog key占用了12k的内存用于标记基数

pfadd命令不是一次性分配12k内存使用，会随着基数的增加内存逐渐增大

pfmerge命令合并后占用的存储空间为12k，无论合并之前的数据量多少

GEO

地理位置数据

基本操作

```

添加坐标点

$ geoadd key longitude latitude member [key longitude latitude member]

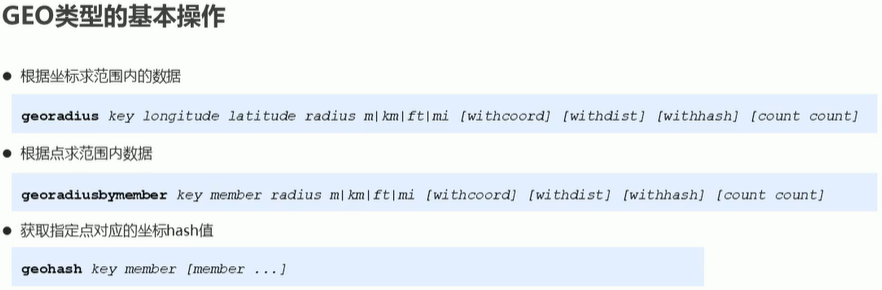
获取坐标点

$ geopos key member [member...]

计算坐标点距离

$ geodist key member1 member2 [unit]

```



Tips

Redis应用于地理位置计算

## 2.4 集群

### 1. 主从复制

简介

互联网“3高”架构

高并发

高性能

高可用

(一年总时长(s) - 全年宕机时长(s)) / 一年总时长(s) = 可用性

业界可用性目标5个9，即99.999%，即服务器全年宕机总时长低于315s。

单机redis有很大风险

1. 机器故障

2. 容量瓶颈

为了避免单点服务器的风险，提供多台服务，数据互通，并保证数据是同步的。即使其中一台宕机，其它机器也能提供服务，实现高可用，同时数据冗余备份。

主从复制：

将master数据即时、有效复制到slave中。

特征：一个master可用拥有多个slave，一个slave只对应一个master。

职责：

master：

写数据。

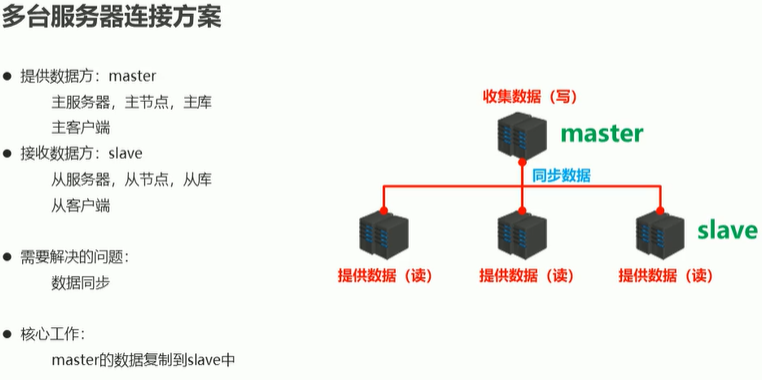
执行写时，将变化的数据自动同步到slave。

读数据（可忽略）。

slave：

读数据。

写数据（禁止）。



主从复制的作用：

1. 读写分离，master写，slave读，提供服务器的读写负载能力

2. 负载均衡，基于主从结构，配合读写分离，由slave分担master负载，并根据需求的变化，改变slave的数量，通过多个节点分担读取负载，大大提供了并发量和数据吞吐量。

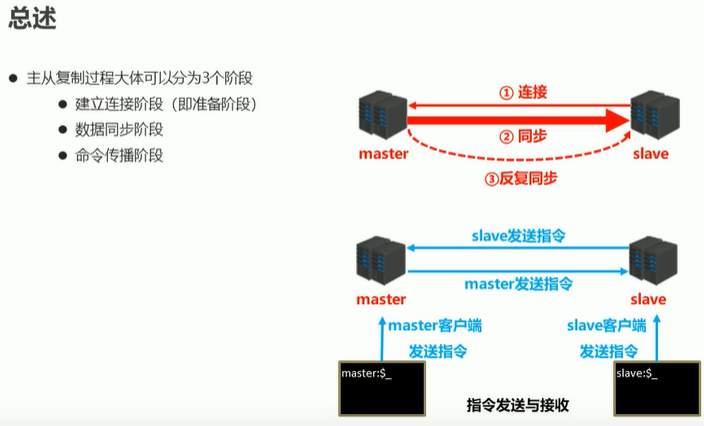
3. 故障恢复，当master故障时，由slave提供服务，实现快速的故障恢复。

4. 数据冗余，实现数据热备份，是持久化之外的一种数据冗余方式

5. 高可用基石，基于主从复制，构建哨兵模式集群，实现高可用方案。

工作流程

【分为3个阶段】



阶段1：建立连接阶段

建立slave到master的连接，使master能识别到slave，并保存slave的端口号。



建立slave到master连接的命令

方式1：客户端发送命令

$ slaveof masterip masterport

方式2：启动服务器参数

$ redis-server -slaveof masterip masterport

方式3：服务器配置

slaveof masterip masterport

slave系统信息

$ info

master\_link\_down\_since\_seconds

master\_host

master\_port

master系统信息

$ info

slave\_listening\_port (多个)

主从断开连接

客户端发送命令

$ slaveof no one

授权访问

master配置文件设置密码

requirepass pwd

master客户端发送命令设置密码

$ config set requirepass <pwd>

$ config get requirepass

slave客户端发送命令设置密码

$ auth <pwd>

slave配置文件设置密码

masterauth <pwd>

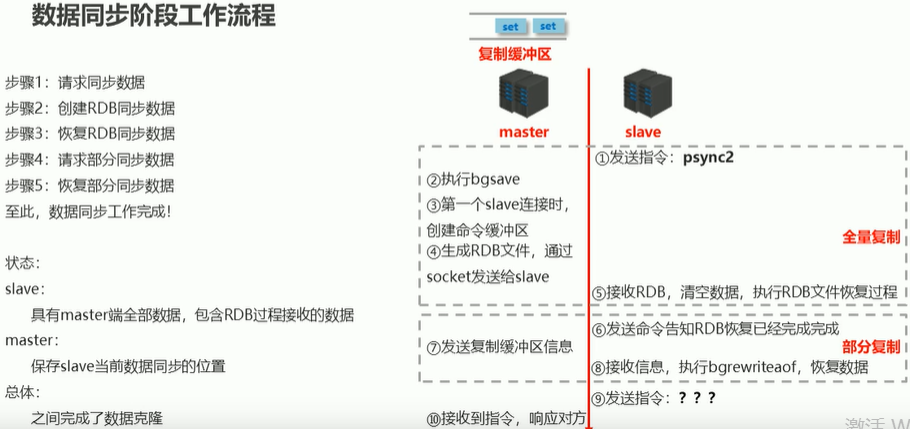
启动客户端设置密码

$ redis-cli -a <pwd>

阶段2：数据同步阶段工作流程

在slave初次连接master后，复制master所有数据到slave

将slave的数据库状态更新成master当前数据库状态



master数据同步期间的注意事项

1. 如果master数据量巨大，数据同步阶段应避开流量高峰期，避免造成master阻塞，影响业务执行

2. 复制缓冲区大小设定不合理，会导致数据溢出。如进行全量复制周期太长，进行部分复制时发现数据已经存在丢失的情况，必须进行第二次全量复制，致使slave陷入死循环状态。

repl-backlog-size 1mb

3. master单机内存占主机内存的比例不应过大，建议使用50%-70%的内存，留下30%-50%的内存用来进行bgsave和创建复制缓冲区。

slave数据同步期间的注意事项

1. 为了避免slave进行全量复制、部分复制时服务器响应阻塞或者数据不同步，建议关闭此期间的对外服务

slave-serve-stale-data yes|no

2. 数据同步阶段，master发给slave的信息可以理解master是slave的一个客户端，主动向slave发送命令

3. 多个slave同时对master请求数据，master发送的rdb文件增多，会对带宽造成巨大冲击，如果master带宽不足，因此数据同步需要根据实际业务场景，适量错峰。

4. slave过多时，建议调整拓扑结构，由一主多从结构变为树状结构，中间的节点既是master也是slave。注意使用树状结构时，由于层级深度，导致深度越高的slave与最顶层master间数据同步延迟较大，数据一致性变差，应谨慎选择。

阶段3：命令传播阶段

当master数据库状态被修改后，导致主从状态不一致，此时需要同步，同步的动作称为命令传播

master将接收到的数据变更命令发送给slave，slave接收后执行

命令传播阶段的部分复制

命令传播阶段出现了断网现象

网络闪断闪连    忽略

短时间网络中断  部分复制

长时间网络中断  全量复制

部分复制的三个核心要素

服务器的运行id（run id）

主服务器的复制积压缓冲区

主从服务器的复制偏移量

服务器运行ID（run id）

概念：

服务器运行ID是每一台服务器每次运行的身份识别码，一台服务器多次运行生成多个运行id。

组成：

运行id由40位字符组成，是一个随机的十六进制字符

fdc343fggregefgsfsdfs3r3224sfsdfsd

作用：

运行id被用于在服务器间进行传输时识别身份。

如果想两次操作均对同一台服务器进行，必须每次操作携带对应的运行id，用于对方识别。

实现方式：

运行id在每台服务器启动时自动生成，master在首次连接slave时，会将自己的运行id发送给slave，slave保存此id，通过info server 命令，可以查看slave的runid。

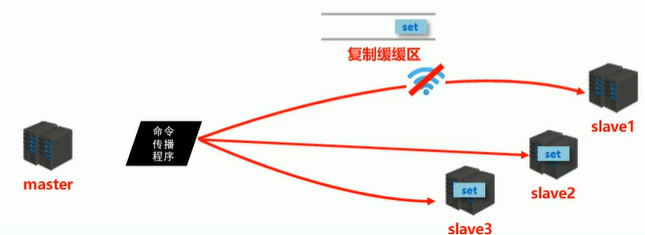
$ info server

run\_id

复制缓冲区

概念：

又名复制积压缓冲区，是一个FIFO的队列。用于存储服务器执行过的命令，每次传播命令，master都会将传播的命令记录下来，并存储在复制缓冲区中。





复制缓冲区默认存储大小1M，由于存储空间大小是固定的，当入队元素大于队列长度时，最先入队的元素被弹出，新的元素放入队列

由来：

每台服务器启动时，如果开启有aof或者被连接称为master节点，即创建复制缓冲区

作用：

用来保存master收到的所有指令（仅影响数据变更的指令，例如set,select...）

数据来源：

当master接收到主客户端指令时，除了运行指令，还将指令存储到缓冲区中。

复制偏移量（offset）

概念：

一个数字，记录复制缓冲区中指令的字节位置

分类：

master复制偏移量：记录发送给所有slave的指令字节对应的位置（多个）

slave复制偏移量：记录slave接收master发送过来的指令字节对应的位置（一个）

数据来源：

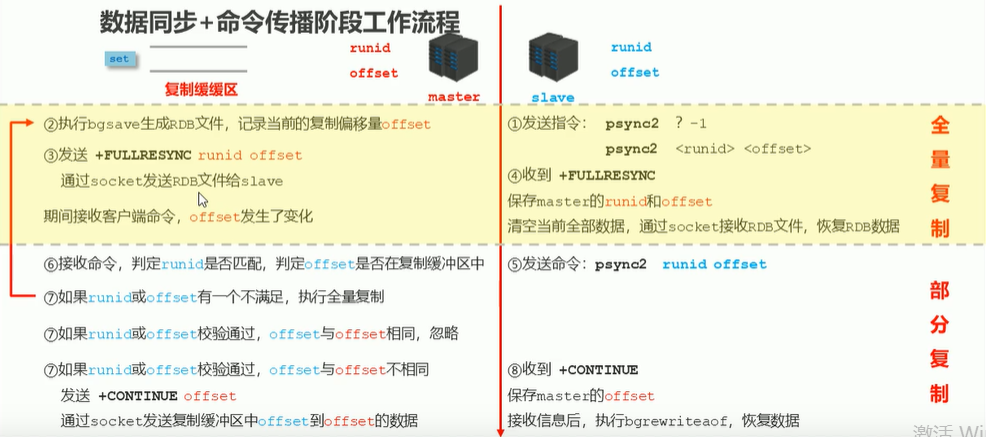
master端：发送一次记录一次

slave端：接收一次记录一次

作用：

同步信息，比对master与slave的差异，当slave断线后，恢复数据使用。

数据同步 + 命令传播阶段工作流程



心跳机制

进入命令传播阶段，master与slave间需要进行信息交换，使用心跳机制进行维护，实现双方保持在线。

master心跳

指令：PING

周期：由repl-ping-slave-period决定，默认10s

作用：判断slave是否在线

查询：INFO replication  获取slave最后一次连接时间间隔，lag项维持在0或1视为正常

slave心跳

指令：REPLCONF ACK {offset}

周期：1s

作用1：汇报自己的复制偏移量，获取最新的数据变更指令

作用2：判断master是否在线

心跳注意事项

1. 当slave多数掉线，或延迟过高时，master为了保障数据稳定性，将拒绝所有信息同步操作

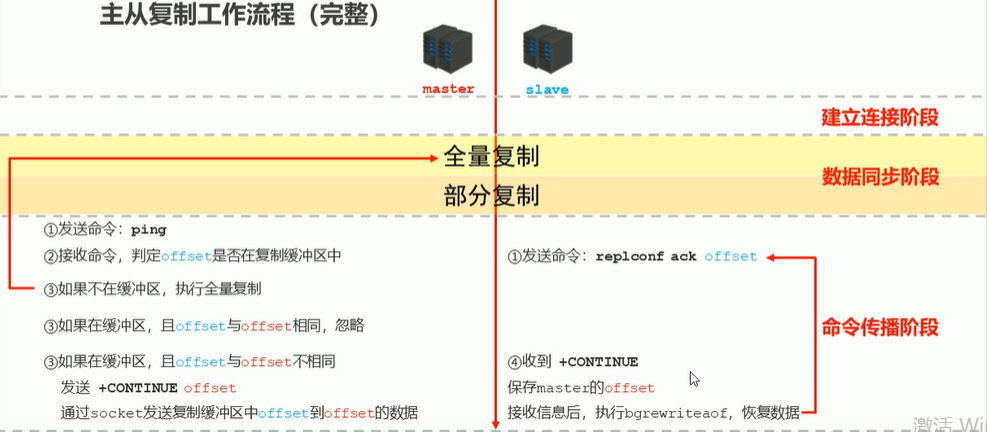
min-slaves-to-write 2

min-slaves-max-lag 10

slave数量少于2个，或者所有slave的延迟都大于等于10s时，强制关闭master写功能，停止数据同步。

2. slave数量由slave发送REPLCONF ACK命令做确认

3. slave延迟由slave发送REPLCONF ACK命令做确认



主从复制常见问题

1. 频繁的全量复制（1）

随着系统运行，master的数据越来越大，一旦master重启，runid将发生变化，会导致全部slave的全量复制操作。

内部优化方案：

1. master内部创建master\_replid变量，使用runid相同的策略生成，长度4位，并发送给所有slave

2. 在master关闭时执行命令 shutdown save，进行RDB持久化，将runid与offset保存到rdb文件中

repl-id repl-offset

通过

$ redis-check-rdb dump-6379.rdb

命令可以查看该信息。

3. master重启后加载rdb文件，恢复数据

重启后，将rdb文件中保存的repl-id与repl-offset加载到内存中

master\_repl\_id = repl

master\_repl\_offset = repl-offset

通过info命令可以查看该信息

作用：

本机保存上次runid，重启后恢复该值，使所有slave认为还是之前的master。

2. 频繁的全量复制（2）

问题现象：

网络环境不佳，出现网络中断，slave不提供服务

问题原因：

复制缓冲区过小，断网后slave的offset越界，触发全量复制

最终结果：

slave反复进行全量复制

解决方案：

修改复制缓冲区大小

repl-backlog-size 100M

建议设置如下：

1. 测算出从master到slave的重连平均时长 second

2. 获取master平均每秒产生写命令数据总量 write\_size\_per\_second

3. 最优复制缓冲区空间 = 2 \* second \* write\_size\_per\_second

3. 频繁的网络中断（1）

问题现象：

master的cpu占用过高或slave频繁断开连接

问题原因：

slave每1秒发送REPLCONF ACK命令道master

当slave有执行慢查询时（keys \*, hgetall等），会大量占用cpu性能

master每1秒调用复制定时函数（replicationCron()），发现slave长时间没有进行响应

最终结果：

master各种资源（输出缓冲区，带宽、连接等）被严重占用

解决方案：

通过设置合理的超时时间，确认是否释放slave

repl-timeout

该参数定义了超时时间的阈值（默认是60s），超过该值，释放slave。

4. 频繁的网络中断（2）

问题现象：

slave与master连接断开

问题原因：

master发送ping指令频度较低

master设置超时时间较短

ping指令在网络中存在丢包

解决方案：

提供ping指令发送的频度

repl-ping-slave-period

超时时间 repl-time的时间至少是ping指令频度的5-10倍，否则slave很容易判定超时。

5. 数据不一致

问题现象：

多个slave获取相同数据不同步

问题原因：

网络信息不同步，数据发送有延迟

解决方案：

1. 优化主从间的网络环境，通常放置在同一个机房部署，如使用云服务器时要注意此现象。

2. 监控主从节点延迟（通过offset判断），如果slave延迟过大，暂时屏蔽程序对该slave的数据访问

slave-serve-stale-data yes|no

开启后仅响应info、slaveof等少数指令（慎用，除非对数据一致性要求很高）

### 2. 哨兵模式

背景：

宕机的master下线

找一个slave作为master

通知所有的slave连接新的master

启动新的master与slave

全量复制\*N + 部分复制\*N

问题：

谁来确认master宕机了

找一个master？谁来找？怎么找？

修改配置后，原始的主恢复了怎么办？

哨兵简介

哨兵（sentinel）是一个分布式系统，用于对主从结构的每台服务器进行监控，当出现故障时通过投票机制选择新的master并将所有slave连接到新的master。

哨兵的作用

1.监控

不断的检查master和slave是否正常运行

master存活检测，master与slave运行情况检测

2.通知（提醒）

当被监控的服务器出现问题时，向其它（哨兵间、客户端）发送通知

3.自动故障转移

断开master与slave连接，选取一个slave作为master，将其它slave连接到新的master，并通知客户端新的服务器地址

注意：

哨兵也是一台redis服务器，只是不提供数据服务

哨兵配置数量通常为单数

配置哨兵

配置一拖二的主从结构

配置3个哨兵（配置相同，端口不同），参考sentinel.conf

启动哨兵

$ redis-sentinel sentinel-端口号.conf

查看配置文件

$ cat sentinel.conf | grep -v "#" | grep -v "^$"

$ cat sentinel.conf > ./conf/sentinel-26380.conf

$ sed 's/26380/23456/g' sentinel-26379.conf > sentinel-26380.conf

哨兵工作原理

哨兵在进行主从切换过程经历了三个阶段

1. 监控阶段

同步信息

2. 通知阶段

保持联通

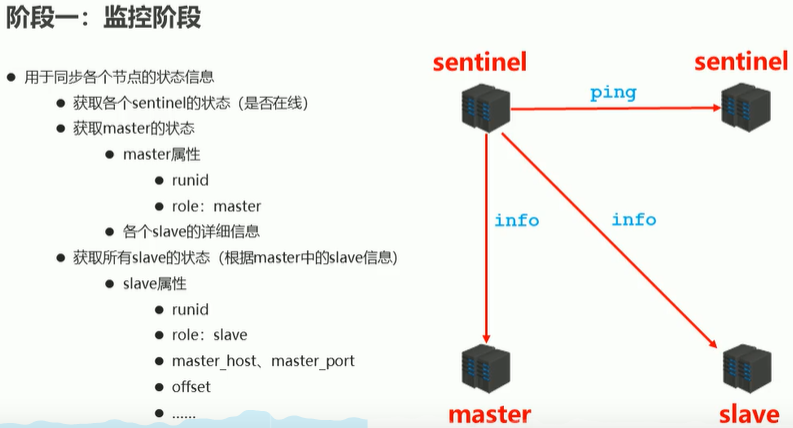
3. 故障转移阶段

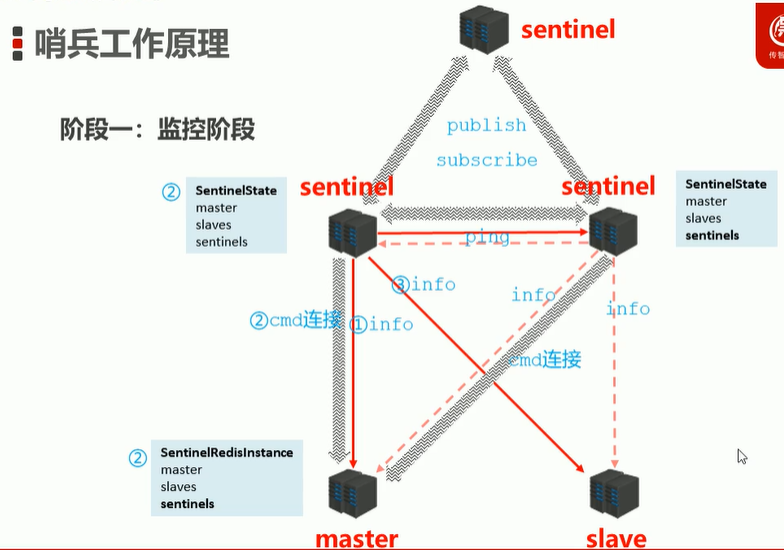
发现问题

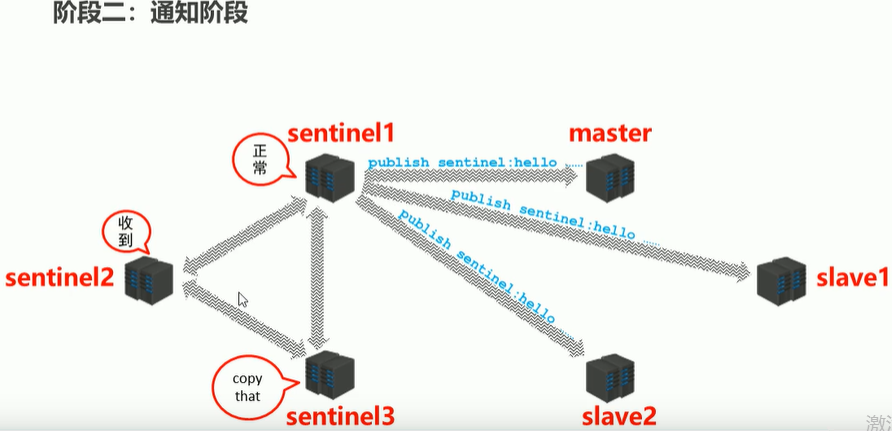
竞选负责人

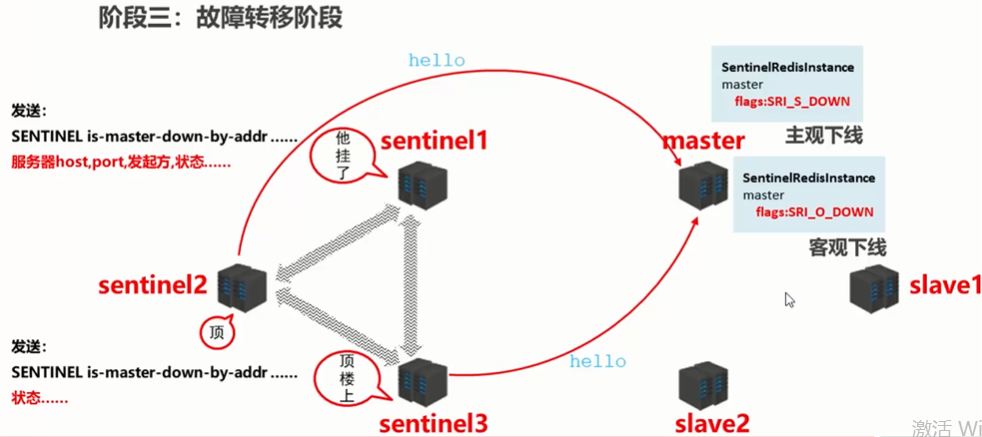
优选新master

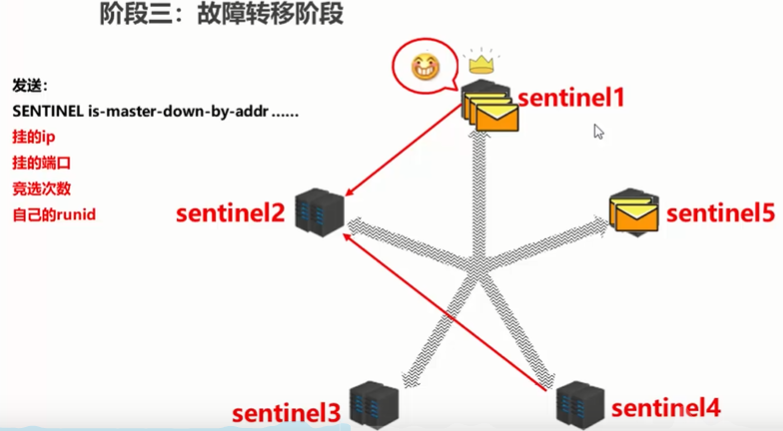
新master上任，其它slave切换master，原master作为slave故障恢复后连接











选举阶段根据信息到达的先后顺序来给定选票。

一轮没有选出来就进行下一轮。



### 3. 集群

简介

集群就是使用网络将若干台计算机联通起来，并提供统一的管理方式，使其对外呈现单机的服务效果。

集群的作用

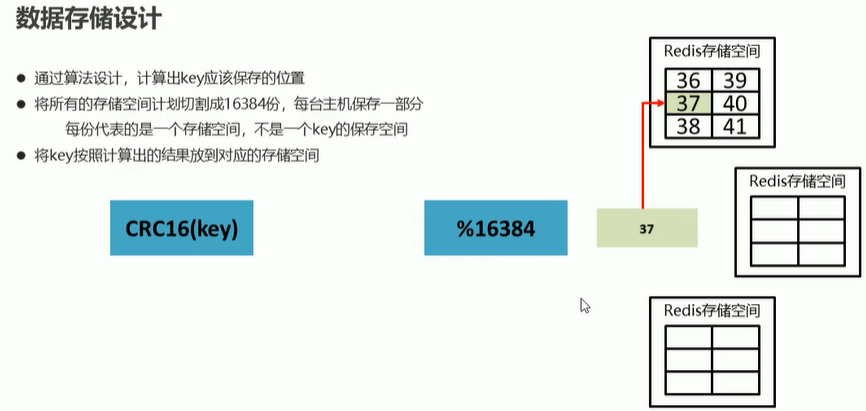
1. 分散单台服务器的访问压力，实现负载均衡

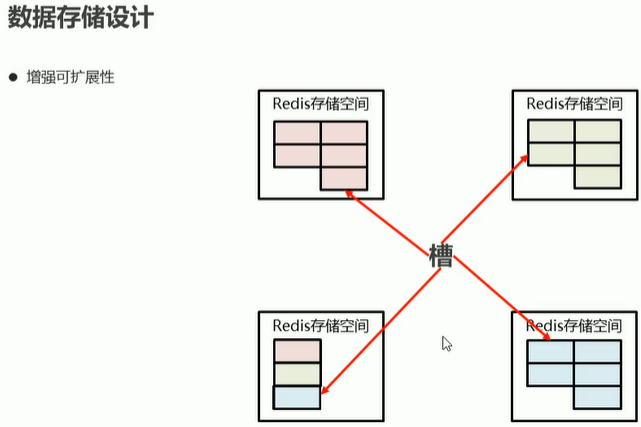
2. 分散单台服务器的存储压力，实现可扩展性

3. 降低单台服务器宕机带来的业务灾难

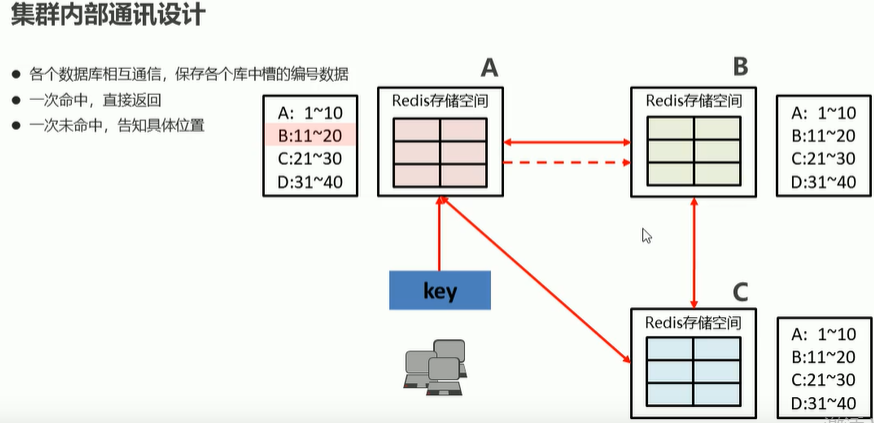
redis集群结构设计

数据存储





数据存储的槽动态分配



最多找2次就能确定数据存储槽的位置

cluster集群结构搭建

配置增加

```

cluster-enabled yes

cluster-config-file node-1.conf

cluster-node-timeout 100000

```

需要安装ruby

$ redis-trib.rb

$ redis-cli -c

$ cluster nodes

cluster配置

```

1. 设置加入cluster，使其称为其中的节点

cluster-enabled yes|no

2. cluster配置文件名，该文件属于自动生成，仅用于快速查找文件并查询文件内容

cluster-config-file <filename>

3. 节点服务响应超时时间，用于判定该节点是否下线或切换为从节点

cluster-node-timeout <milliseconds>

4. master连接的slave最小数量

cluster-migration-barrier <count>

```

cluster节点操作命令

```

1. 查看集群节点信息

$ cluster nodes

2. 进入一个从节点redis，切换其主节点

$ cluster replicate <master-id>

3. 发现一个新节点，新增主节点

$ cluster meet ip:port

4. 忽略一个没有solt的节点

$ cluster forget <id>

5. 手动故障转移

$ cluster failover

```

可以选择不同的cluster方案。

# 3. 应用

3.1 企业级解决方案

## 1. 缓存预热

问题排查：

1. 请求数量高

2. 主从之间的数据吞吐量大，数据同步操作频度高

解决方案：

前置准备工作：

1. 日常例行统计数据访问记录，统计访问频度较高的热点数据

2. 利用LRU数据删除策略，构建数据留存队列，例如storm与kafka配合

准备工作：

3. 将统计集结果中的数据分类，根据级别，redis优先加载级别较高的热点数据

4. 利用分布式多服务器同时进行数据读取，提供数据加载过程

实施：

1. 利用脚本程序固定触发数据预热过程

2. 如果条件允许，使用CDN（内容分发网络），效果更好

总结：

缓存预热就是系统启动前，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据。

## 2. 缓存雪崩

现象：

数据库服务器崩溃（1）

1. 系统平稳运行过程中，忽然数据库连接量激增

2. 应用服务器无法及时处理请求

3. 大量408、500错误页面出现

4. 客户反复刷新页面获取数据

5. 数据库崩溃

6. 应用服务器崩溃

7. redis服务器崩溃

8. redis集群崩溃

9. 重启数据库后再次被瞬间流量放到

问题排查：

1. 在一个较短的时间内，缓存中较多的key集中过期

2. 此周期内请求访问过期的数据，redis未命中，redis向数据库获取数据

3. 数据库同时接收到大量的请求无法及时处理

4. redis大量请求被积压，开始出现超时现象

5. 数据库流量激增，数据库崩溃

6. 重启后仍然面对缓存中无数据可用

7. redis服务器资源被严重占用，redis服务器崩溃

8. redis集群呈现崩塌，集群瓦解

9. 应用服务器无法及时得到数据响应，来自客户端的请求数量越来越多，应用服务器崩溃

10. 应用服务器，redis，数据库全部重启，效果不理想

问题分析

1. 短时间范围内

2. 大量key集中过期

解决方案（道）

1. 更多的页面静态化处理

2. 构建多级缓存架构

nginx缓存 + redis缓存 + ehcache缓存

3. 检测mysql数据库耗时业务进行优化

对数据库的瓶颈进行排查：例如超时查询，耗时较高的事务等

4. 灾难预警机制

监控redis的性能指标

cpu占用，cpu使用率

内存容量

查询平均响应时间

线程数

5. 限流、降级

短时间范围内牺牲一些客户体验，限制一部分请求访问，降低应用服务器压力，待业务低速运转后再逐步放开访问

解决方案（术）

1. LRU与LFU切换

2. 数据有效期策略调整

根据业务数据有效期进行分类错峰，A类90min, B类80min..

过期时间使用固定时间+随机值的形式，稀释集中到期的key数量

3. 超热数据使用永久key

4. 定期维护（自动 + 人工）

对即将过期数据做访问量统计分析，确认是否延时，配合访问量统计，做热点数据的延时

5. 加锁

慎用！

总结：

缓存雪崩就是瞬间过期数据量太大，导致对数据库服务器造成压力。如果能够有效避免过期时间集中，可以有效解决雪崩现象的出现（越40%），配合其它策略一起使用，并监控服务器的运行数据，根据运行记录做快速调整。

## 3. 缓存击穿

数据库服务器崩溃（2）

1. 系统平稳运行过程中

2. 数据库连接量瞬间激增

3. redis服务器无大量key过期

4. redis内存平稳，无波动

5. redis服务器cpu正常

6. 数据库崩溃

问题排查

1. redis中的某个key过期，该key访问量巨大

2. 多个数据请求从服务器直接压到redis后，均未命中

3. redis在短时间内发起了大量对数据库中同一个数据的访问

问题分析

1. 单个key高热数据

2. key过期

解决方案（术）

1. 预先设定

以电商为例，每个商家根据店铺等级，指定若干主打商品，在购物期间，加大此类信息key的过期时长

注意：购物节不仅仅指当天，以及后续若干天，访问峰值呈现逐渐降低的趋势

2. 现场调整

监控访问量，对自然流量激增的后台数据延长过期时间或者设为永久性key

3. 后台刷新数据

启动定时任务，高峰期来临之前，刷新数据有效期，确保不丢失

4. 二级缓存

设置不同的失效时间，保障不会被同时淘汰就行

5. 加锁

分布式锁，防止被击穿，但是要注意性能瓶颈，慎重！

总结：

缓存击穿就是单个高热数据过期的瞬间，数据访问量较大，未命中redis后，发起了大量对同一数据的数据库访问，导致对数据库服务器造成压力。应对策略应该在业务数据分析与预防方面进行，配合运行监控测试与及时调整策略，毕竟单个key的过期监控难度较高，配合雪崩处理策略即可。

## 4. 缓存穿透

数据库服务器崩溃（3）

1. 系统平稳运行过程中

2. 应用服务器流量随着时间增量较大

3. redis服务器命中率随时间逐步降低

4. redis内存平稳，内存无压力

5. redis服务器cpu占用激增

6. 数据库服务器压力激增

7. 数据库崩溃

问题排查

1. redis中大面积出现未命中

2. 出现非正常访问url访问

问题分析

1. 获取的数据在数据库中也不存在，数据库查询未得到对应的数据

2. redis获取到null数据未进行持久化，直接返回

3. 下次此类数据到达重复上述过程

4. 出现黑客攻击服务器

解决方案（术）

1. 缓存null

对查询结果为null的数据进行缓存（长期使用，定期清理），设定短时限，例如30-60s，最高5min

2. 黑白名单策略

提前预热各种分类数据id对应的bitmaps，id作为bitmaps的offset，相当于设置了数据白名单。当加载正常数据时，放行，加载异常数据时直接拦截（效率偏低）

使用布隆过滤器（有关布隆过滤器的命中问题对当前状况可以忽略）

3. 实时监控

实时监控redis命中率（业务正常范围时，通常会有一个波动值）与null值的比例

非活动时段波动：通常检测3-5倍，超过5倍纳入重点排查对象

活动时段波动：通常检测5-10倍，超过50倍纳入重点排查对象

根据倍数不同，启动不同的排查流程。然后使用黑名单进行防控（运营）

4. key加密

问题出现后，临时启动防灾业务key，对key进行业务层传输加密服务，设定校验程序，过来的key校验

例如每天随机分配60个加密串，挑选2到3个，混淆到页面id中，发现访问ey不满足规则，驳回数据访问

总结

缓存击穿访问了不存在的数据，跳过了合法数据的redis数据缓存，每次访问数据库，导致对数据库服务器造成压力。通常此类数据的出现量是一个较低的值，当出现此类情况以毒攻毒，并及时报警。应对策略应该在临时预案防范方面多做文章。

无论是黑名单还是白名单，都是对整体系统的压力，警报解除后尽快移除。

# 4. 性能指标监控

监控指标

性能指标：Performance

内存指标：Memory

基本活动指标：Basic activity

持久性指标：Persistence

错误指标：Error

性能指标：Performance

latency         redis响应一个请求的时间

instantaneous\_ops\_per\_sec   平均每秒处理请求总数

hit rate(calculated)    缓存命中率(计算出来的)

内存指标：Memory

used\_memory     已使用内存

mem\_fragmentation\_ratio 内存碎片率

evicted\_keys    由于最大内存限制被移除的key数量

blocked\_clients 由于BLPOP、BRPOP、or BRPOPLPUSH而被阻塞的客户端

基本活动指标：Basic activity

connected\_clients        客户端连接数

connected\_slaves         slave数量

master\_last\_io\_seconds\_ago  最近一次主从交互之后的秒数

keyspace        数据库中的key值总数

持久性指标：Persistence

rdb\_last\_save\_time  最后一次持久化保存到磁盘的时间戳

rdb\_changes\_since\_last\_save   自最后一次持久化以来数据库的更改数

错误指标：Error

rejected\_connections    由于达到maxclient限制而被拒绝的连接数

keyspace\_misses         key值查找失败（没有命中）次数

master\_link\_down\_since\_seconds  主从断开的持续时间（以秒为单位）

监控方式

工具

cloud Insight Redis

Prometheus

Redis-stat

Redis-faina

RedisLive

zabbix

命令

benchmark

redis cli

monitor

slowlog

benchmark

```

$ redis-benchmark [-h ] [-p ] [-c ] [-n <requests>] [-k ]

$ redis-benchmark

50个连接，10000次请求对应的性能

$ redis-benchmark -c 100 -n 50000

100个连接，50000次请求对应的性能

```

monitor

```

$ monitor

打印服务器调试信息

```

slowlog

```

$ slowlog [operator]

get: 获取慢查询日志

len: 获取慢查询日志条目数

reset:  重置慢查询日志

相关配置

slowlog-log-slower-than 1000 # 设置慢查询的时间下限，单位：微秒

slowlog-max-len 100 # 设置慢查询命令对应的日志显式长度，单位：命令数

```