

Redis

如何学好Redis

- 掌握数据结构
- 掌握常用特性
- 玩转配置文件 *.conf
- 掌握应用场景

Redis基础

一、Redis简介

1.1 出现问题的场景

1.12306经常崩

2.双11崩溃

3.北京奥运会订票崩溃

1.2 问题在哪？

1.2.1 问题

- 海量用户
- 高并发

1.2.2 问题源头 => 关系型数据库

问题

- io性能低下
- 数据关系复杂、扩展性能差，不便于大规模集群

解决

- 降低磁盘IO次数，越低越好 - 内存存储
- 去除数据间关系，越简单越好 - n不存储关系，仅存储数据

1.3 NoSQL

Nosql(非关系型数据库), not only sql => 关系数据库的补充

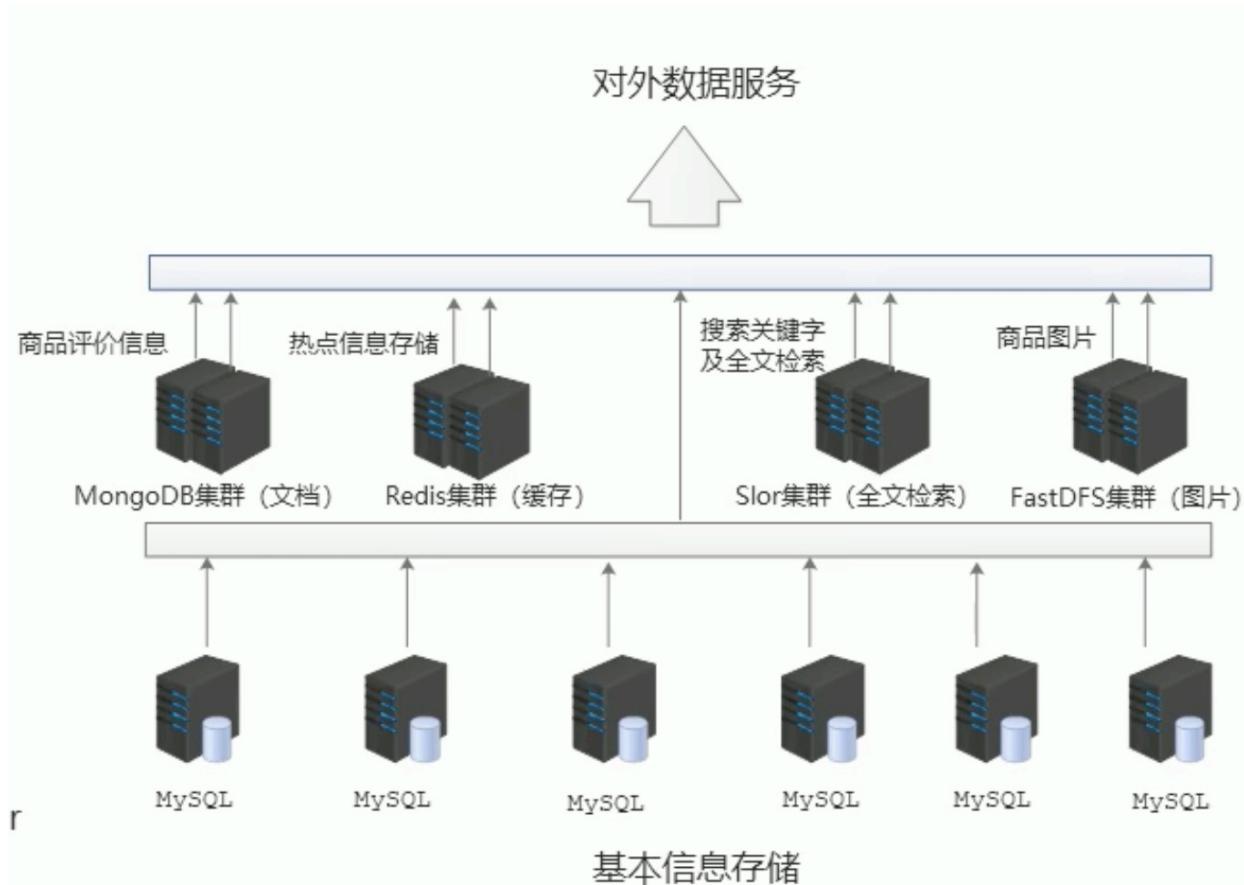
1.3.1 特征(相对SQL):

- 可扩容、可伸缩
- 大数据下的高性能
- 灵活的数据模型
- 高可用

1.3.2 常见NoSQL:

- Redis
- memcache
- HBase
- MongoDB

1.3.3 NoSQL应用场景=>电商



商品基本信息 => MySQL:

名称、价格、厂商

商品附加信息 => MongoDB:

描述、详情、评论

图片信息

分布式文件系统

搜索关键字

ES、Lucene、solr

热点信息 (redis、meache..、tair)

高频、波段性

1.4 Redis

1.4.1 概念:

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。

1.4.2 特征:

1. 数据间没有必然的关联关系
2. 内部采用单线程机制进行工作
3. 高性能。官方提供测试数据, 50个并发执行100000个请求, 读的速度是110000次/s,写的速度是81000次/s
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

持久化支持: 可以进行数据灾难恢复

1.4.3 Redis的应用场景

1. 为热点数据加速查询(主要场景) :

如热点商品、热点新闻、热点资讯、推广类等高访问量信息等、任务队列, 如秒杀、抢购、购票排队等

2. 即时信息查询(实时系统):

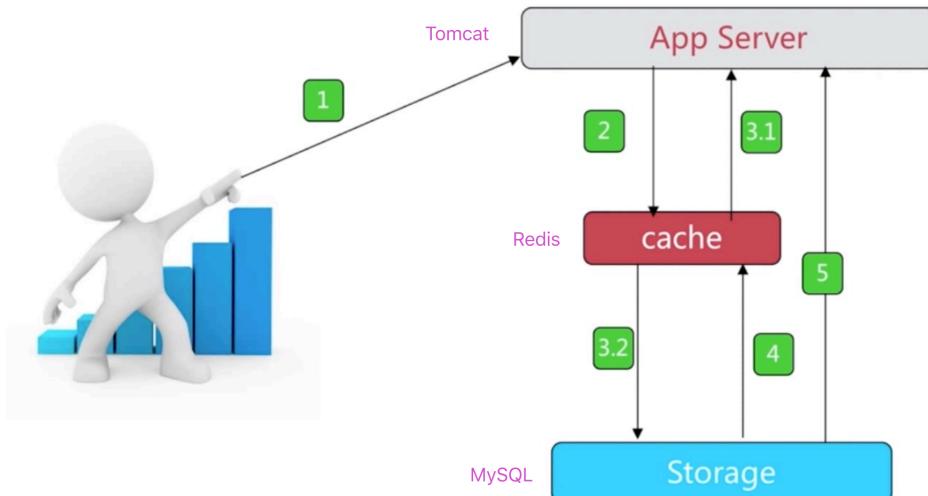
如各位排行榜、各类网站访问统计、公交到站信息、在线人数信息(聊天室、网站)、设备信号等、

排行榜

Top Questions			interesting	442 featured	hot	week	month
0 votes	1 answer	15 views	How to solve Android Error type 3?	android android-manifest	answered 1 min ago Harshad 75		
0 votes	0 answers	15 views	What does # mean in SASS?	html css sass	asked 5 mins ago Xeon 87		
0 votes	3 answers	23 views	Python insert vs Ruby insert	python arrays ruby	modified 9 mins ago Yu Hao 74.6k		
4 votes	2 answers	36 views	Why these two ways of creating a dictionary are both valid and give the same result? [on hold]	python dictionary	modified 18 mins ago thefourtheye 96.1k		
0 votes	0 answers	11 views	Not recognizing command line arguments	c	modified 5 mins ago Tim Biegeleisen 18.1k		
-1 votes	3 answers	35 views	Java Nested If/Else Conditionals	java if-statement	answered 18 mins ago That Guy Lionel 1		

3. 缓存系统：

缓存系统



- 原始业务功能设计
 - 秒杀
 - 618活动
 - 双11活动
 - 排队购票
- 运营平台监控到的突发高频访问数据
- 突发时政要闻，被强势关注围观
- 高频、复杂的统计数据
- 在线人数
- 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

4. 时效性信息控制:

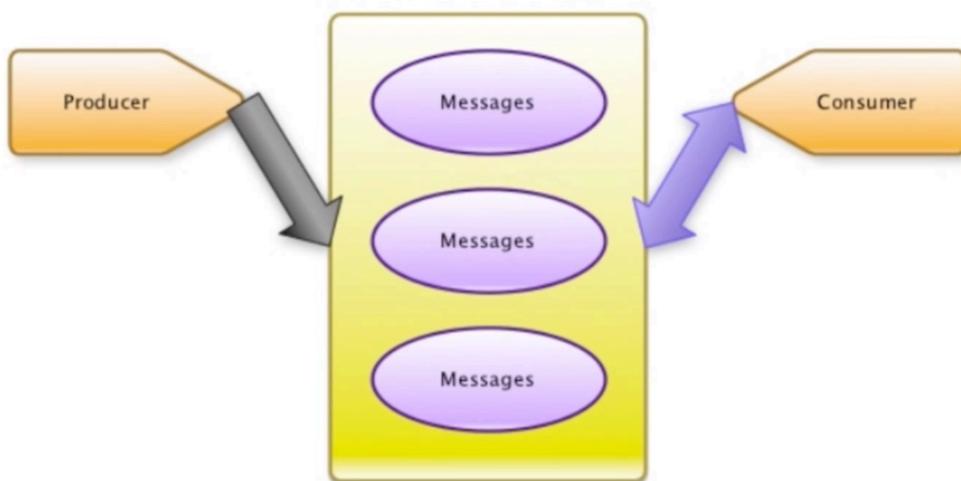
如验证码控制、投票控制等

5. 分布式数据共享:

如分布式集群架构中的session分离

6. 消息队列

消息队列系统



7. 计数器

计数器



8. 实时系统:

垃圾邮件处理、过滤器(布隆过滤器)

9. 社交网站:

微博、Twitter、...

二、Redis数据类型

1. String(String)
2. Hash(HashMap)
3. List(LinkList)
4. Set(HashSet)
5. Sort_Set(TreeSet)

学习目标-掌握17个解决方案：

- redis用于控制数据库表主键id,为数据库表主键提供生成策略,保障数据库表的主键唯一性
- redis控制数据的生命周期, 通过数据是否失效控制业务行为,适用于所有具有时效性限定控制的操作
- redis应用于各种结构型和非结构型高热度数据访问加速
- redis应用于购物车数据存储设计
- redis应用于抢购, 限购类、限量发放优惠卷、激活码等业务的数据存储设计
- redis应用于具有操作先后顺序的数据控制
- redis应用于最新消息展示
- redis应用于随机推荐类信息检索, 例如热点歌单推荐,热点新闻推荐, 热卖旅游线路,应用APP推荐, 大V推荐等
- redis应用于同类信息的关联搜索, 二度关联搜索, 深度关联搜索
- redis应用于同类型不重复数据的合并、取交集操作
- redis应用于同类型数据的快速去重
- redis应用于基于黑名单与白名单设定的服务控制
- redis应用于计数器组合排序功能对应的排名
- redis应用于定时任务执行顺序管理或任务过期管理
- redis应用于及时任务/消息队列执行管理
- redis应用于按次结算的服务控制
- redis应用于基于时间顺序的数据操作, 而不关注具体时间

2.1 String类型

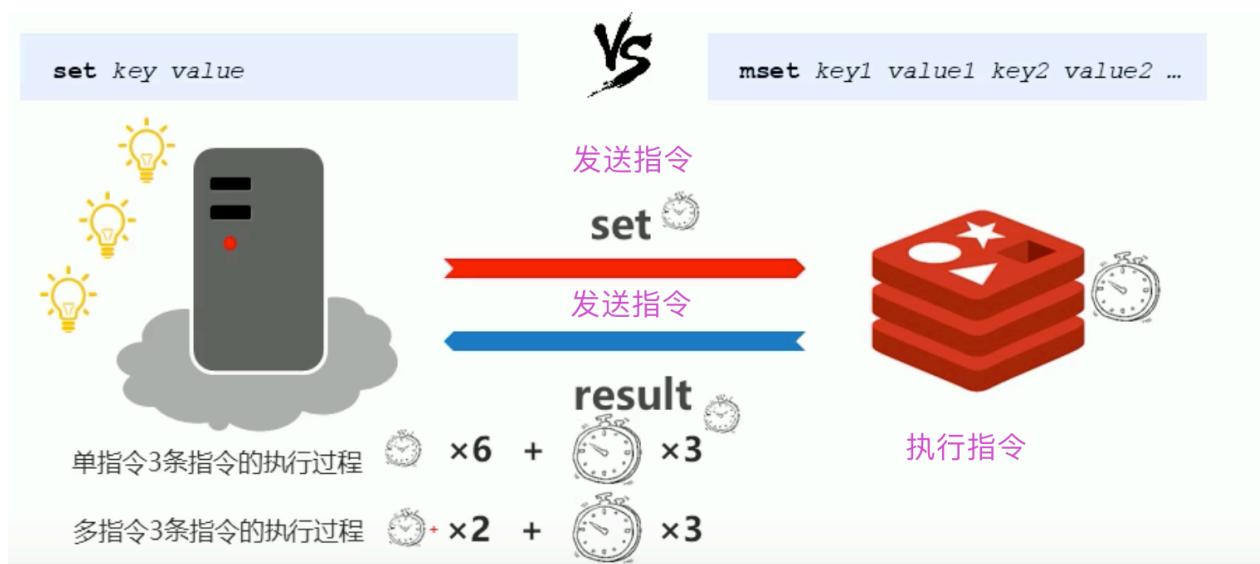
2.1.1 String类型数据的基本操作

1. 添加/修改数据: `set key value`
2. 获取数据 : `get key`
3. 删除数据: `del key`
4. 添加/修改多个数据: `mset key1 value1 key2 value2 ...`
5. 获取多个数据 : `mget key1 key2 ...`

6. 获取数据字符个数(字符串长度): `strlen key`

7. 追加信息到原始信息后部(如果原始信息存在就追加, 否则新建): `append key value`

如何选择 `mset` or `set`? => 从指令性能来分析



Key的设置规范:

数据库中的热点数据key命名惯例

表名	:	主键名	:	主键值	:	字段名
eg1:	order :	id	:	29437595	:	name
eg2:	equip :	id	:	390472345	:	type
eg3:	news :	id	:	202004150	:	title

2.1.2 String类型业务场景1

大型企业级应用中, 分表操作是基本操作, 使用多张表存储同类型数据, 但是对应的主键id必须保证唯一性, 不能重复。Oracle数据库具有sequence设定, 可以解决该问题, 但是MySQL数据库并不具有类似的机制, 那么如何解决?



String类型数据的扩展操作 (上述解决方案)

设置数值数据增加指定范围的值(值可为负)	设置数值数据减少指定范围的值
incr key	decr key
incrby key increment	decrby key incnemt
incrbyfloat key increment	

Tips: 可以通过 `help` 指令 查询指令用法

String作为数值操作

1.string在redis内部存储默认就是一个字符串，当遇到增减类操作incr, decr时会转成数值型进行计算。
2.redis所有的操作都是原子性的，采用单线程处理所有业务,命令是单个执行的，因此无需考虑并发带来的数据影响。

注意: 按数值进行操作的数据，如果原始数据不能转成数值,或超越了redis 数值上限范围，将报错。

如: 9223372036854775807 (java中long型数据最大值, Long.MAX_VALUE)

Tips 1:

- redis用于控制数据库表 主键id,为数据库表主键提供生成策略,保障数据库表的主键唯一性
- 此方案适用于所有数据库，支持数据库集群

2.1.3 String类型业务场景2

"最强女生"启动海选投票，只能通过微信投票，每个微信号每4小时只能投1票。

电商商家开启热门商品推荐，热门商品不能一直处于热门期，每种商品热门期维持3天，3天后自动取消热：[]。

新闻网站会出现热点新闻，热点新闻最大的特征是时效性，如何自动控制热点新闻的时效性。

解决方案：

设置数据具有指定的生命周期

`setex key seconds value`

`psetex key milliseconds value`

Tips 2: redis控制数据的生命周期，通过数据是否失效控制业务行为，适用于所有具有时效性限定控制的操作

2.1.4 String类型数据操作的注意事项

数据操作不成功的反馈与数据正常操作之间的差异:

1.表示运行结果是否成功:

(integer)0 → false == 失败

(integer)1 - → true == 成功

2.表示运行结果值:

(integer)3 → 3 == 3个

(integer)1 → 1 == 1个

- 3. 数据未获取到: (nil)等同于null
- 4. 数据最大存储量: 512MB
- 5. 数值计算最大范围(java中的long的最大值): 9223372036854775807

2.1.5 String类型应用场景

业务场景: 主页高频访问信息显示控制, 例如: 新浪微博大V, 主页显示、粉丝数、微博数量

解决方案1: 在redis中为大V用户设定用户信息, 以用户主键和属性值作为key,后台设定定时刷新策略即可

例如(注意key的命名规范):

```
user:id:35067 283 70:fans =>12210947  
user:id:35067 283 70:blogs =>6164  
user:id:35067 28370:focus =>83
```

解决方案2: 在redis中以json格式存储大V用户信息, 定时刷新(也可以使用hash类型)

```
userid:3506728370 => { id:35067 28370, name:春晚, fans:12210862, blogs:6164,  
focus:83 }
```

Tips 3: Redis应用于各种结构型和非结构型高热度数据访问加速

2.2 hash类型

2.2.1 hash引出

- 存储的困惑:
对象类数据的存储如果具有较频繁的更新需求操作会显得笨重
- 新的存储需求:
对一系列存储的数据进行编组, 方便管理, 典型应用存储对象信息
- 需要的存储结构:
1个存储空间保存多个键值对数据

2.2.2 hash数据结构:

底层使用哈希表结构实现数据存储

- hash存储结构优化:
 - 如果field数量较少, 存储结构优化为类数组结构
 - 如果field数量较多, 存储结构使用HashMap结构

2.2.3 hash类型数据基本操作

1. 添加/修改数据

```
hset key field value
```

2. 获取数据

```
hget key field  
hgetall key
```

3. 删除数据

```
hdel key field1 [ field2 ]
```

4. 添加/修改多个数据

```
hmset key field1 valne1 field2 value2..
```

5. 获取多个数据

```
hmget key field1 field2
```

6. 获取哈希表中字段的数量

```
hlen key
```

7. 获取哈希表中是否存在指定的字段

```
hexists key field
```

2.2.4 hash类型数据扩展操作

1. 获取哈希表中所有的字段名或字段值 => 区别 hgetall key 获取所有key&val

- `hkeys key`
- `hvals key`

2. 设置指定字段的数值数据增加指定范围的值

- `hincrby key field increment`
- `hinarbyfloat key field increment`

3. 设置hash field的值, 如果field不存在

- `hsetnx key field value`
- 应用于购物车访问了数量的数据, 如果存在, 则不改变数量.

2.2.5 hash类型数据操作的注意事项

1. hash类型下的value只能存储字符串, 不允许存储其他数据类型, 不存在嵌套现象(禁止套娃), 如果数据未获取到, 对应的值为(nil)
2. 每个hash可以存储232-1个键值对
3. hash类型十分贴近对象的数据存储形式, 并且可以灵活添加删除对象属性。但hash设计初衷不是为了存储大量对象而设计的, 切记不可滥用, 更不可以将hash作为对象列表使用
4. hgetall 操作可以获取全部属性, 如果内部field过多, 遍历整体数据效率就很会低, 有可能成为数据访问瓶颈

2.2.6 hash应用场景1

1. 业务场景: 电商网站购物车设计与实现
2. 业务分析: 仅分析购物车的redis存储模型
 - 添加、浏览、更改数量、删除、清空

- 购物车于数据库间持久化同步(不讨论)
- 购物车于订单间关系(不讨论)
- 提交购物车:读取数据生成订单
- 商家临时价格调整:隶属于订单级别
- 未登录用户购物车信息存储(不讨论)
- cookie存储

3.解决方案

- 以客户id作为key,每位客户创建一个hash存储结构存储对应的购物车信息
- 将商品编号作为field,购买数量作为value进行存储
- 添加商品:追加全新的field与value
- 浏览:遍历hash
- 更改数量:自增/自减, 设置value值
- 删除商品:删除field
- 清空:删除key

示例1:

```
//两个用户的购物车
127.0.0.1:6379> hmset u001 g01 100 g02 200
127.0.0.1:6379> hmset u002 g01 50 g02 300
```

示例1的设计是否加速了购物车的呈现?

当前仅仅是将数据存储到了redis中,并没有起到加速的作用,商品信息还需要二次查询数据库每条购物车中的商品记录保存成两条field:

- field1专用于保存购买数量
 - 命名格式:商品id:nums
 - 保存数据:数值
- field2专用于保存购物车中显示的信息,包含文字描述,图片地址,所属商家信息等
 - 命名格式:商品id:info
 - 保存数据: json

示例2:

```
127.0.0.1:6379> hmset u003 g01:nums 100 g01:info xxx
127.0.0.1:6379> hmset u004 g01:nums 200 g01:info xxx
127.0.0.1:6379> hgetall u003
1) "g01:nums"
2) "100"
3) "g01:info"
4) "xxx"
127.0.0.1:6379> hgetall u004
1) "g01:nums"
2) "200"
3) "g01:info"
4) "xxx"
```

示例2：出现问题 g01:info 信息重复

由于**field2**具有商品的公共信息，如文字描述、图片等等，每个用户购买相应产品会对应相同的商品公共信息，所以存在信息重复

将**field2**封装为独立的hash，格式为json

2.2.7 hash应用场景2

双11活动日，销售手机充值卡的商家对移动、联通、电信的30元、50元、100元商品推出抢购活动，每种商品抢购上限1000张



解决方案：

1. 以商家id作为key
2. 将参与抢购的商品id作为field
3. 将参与抢购的商品数量作为对应的value
4. 抢购时使用降值的方式控制产品数量
5. 实际业务中还有超卖等实际问题，这里不做讨论

Tips 5: redis 应用于抢购，限购类、限量发放优惠卷、激活码等业务的数据存储设计

2.3 list类型

2.3.1 list类型数据基本操作

- 添加/修改数据

```
lpush key value1 [value2] .....
rpush key value1 [value2].....
```

- 获取数据

```
lrange key start stop
lindex key index
llen key
```

- 获取并移除数据

```
lpop key
rpop key
```

- 代码演示

```
127.0.0.1:6379> lpush list1 1 2 3 4
(integer) 4
127.0.0.1:6379> rpush list1 5 6 7
(integer) 7
127.0.0.1:6379> lrange list1 0 7 // 0 -1
//4321567
```

2.3.2 list类型数据扩展操作

规定时间内获取并移除数据

```
blpop key1 [key2] timeout
brpop key1 [key2] timeout
//移除指定list的第一个元素，若当时list为空则等待timeout秒，timeout秒内若list内有值则取出，否则为空
```

2.3.3 应用场景1：

业务场景

微信朋友圈点赞，要求按照点赞顺序显示点赞好友信息
如果取消点赞，移除对应好友信息



取消，从中间拿走元素

解决方案：移除指定数据(取走指定元素)

```
lrem key count value
```

Tips 6:redis应用于具有操作先后顺序的数据控制

2.3.4 list类型数据操作注意事项

1. list中保存的数据都是string类型的，数据总容量是有限的，最多232-1个元素(4294967295)。
2. list具有索引的概念，但是操作数据时通常以队列的形式进行入队出队操作，或以栈的形式进行入栈出栈操作，获取全部数据操作结束索引设置为-1

3. list可以对数据进行分页操作,通常第一页的信息来自于list, 第2页及更多的信息通过数据库的形式加载

2.3.5 应用场景2

twitter、新浪微博、腾讯微博中个人用户的关注列表需要按照用户的关注顺序进行展示,粉丝列表需要将最近关注的粉丝列在前面

新闻、资讯类网站如何将最新的新闻或资讯按照发生的时间顺序展示?

企业运营过程中, 系统将产生出大量的运营数据,如何保障多台服务器操作日志的统一顺序输出?

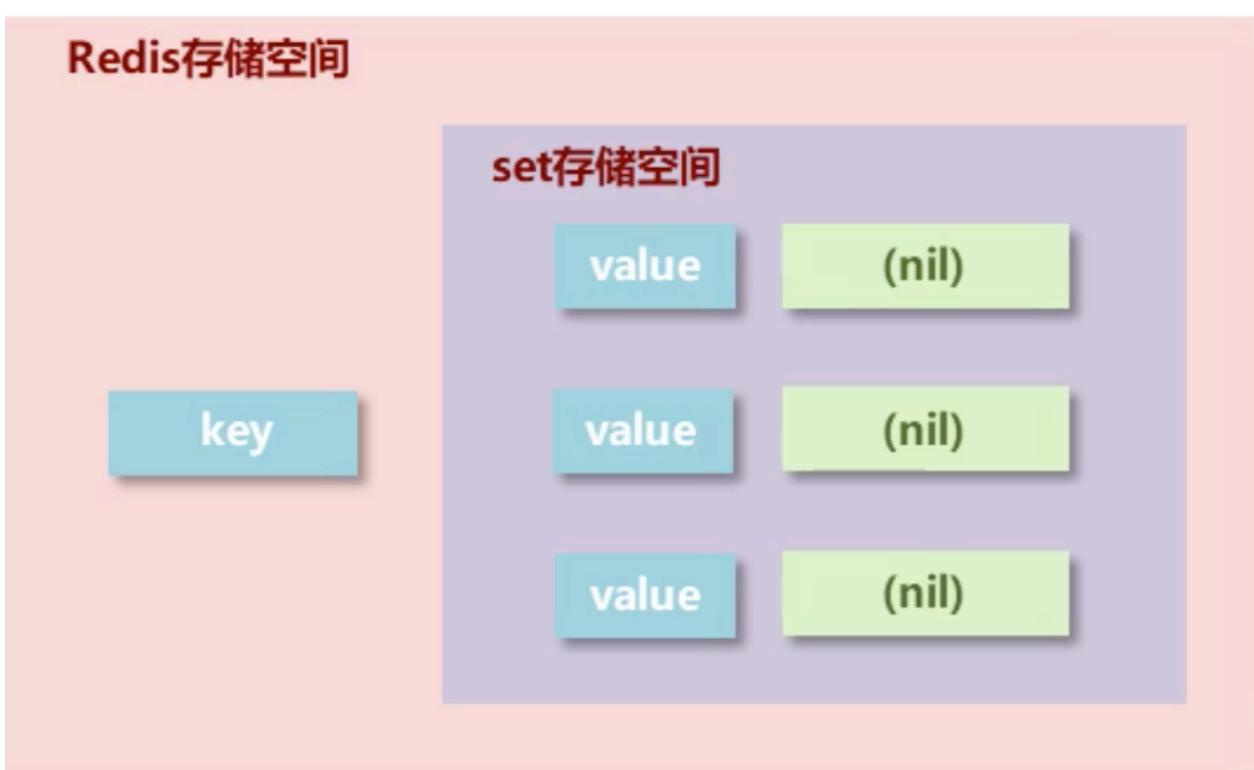
解决方案:

- 依赖list的数据具有顺序的特征对信息进行管理
- 使用队列模型解决多路信息汇总合并的问题
- 使用栈模型解决最新消息的问题

2.4 Set类型

2.4.1 为什么需要Set存储结构

- list是双向链表结构, 增删效率高, 查询慢
- 新的存储需求:存储大量的数据, 在查询方面提供更高的效率
- 需要的存储结构:能够保存大量的数据, 整的内部存储机制, 便于查询
- set类型:与hash存储结构完全相同, 仅存储键,不存储值(nil), 并且值是不允许重复的



2.4.2 Set类型数据的基本操作

- 添加数据
`sadd key member1 [member2]`
- 获取全部数据

- smembers key
- 删除数据
srem key member1 [member2]
- 获取集合数据总量
scard key
- 判断集合中是否包含指定数据
sismember key member

2.4.3 Set类型数据的扩展操作1:

业务场景1-推荐

每位用户首次使用今日头条时会设置3项爱好的内容，但是后期为了增加用户的活跃度、兴趣点，必须让用户对其他信息类别逐渐产生兴趣，增加客户留存度，如何实现？

业务分析：

- 系统分析出各个分类的最新或最热点信息条目并组织成set集合、
- 随机挑选其中部分信息
- 配合用户关注信息分类中的热点信息组织成展示的全信息集合

解决方案：

- 随机获取集合中指定数量的数据
 srandmember key [count]
- 随机获取集合中的某个数据并将该数据移出集合
 spop key

Tips 8: redis 应用于随机推荐类信息检索，例如热点歌单推荐，热点新闻推荐，热卖旅游线路，应用APP推荐，大V推荐等

2.4.4 Set类型数据的扩展操作2:

业务场景2-扩展延伸

- 脉脉为了促进用户间的交流，保障业务成单率的提升，需要让每位用户拥有大量的好友，事实上职场新人不具有更多的职场好友，如何快速为用户积累更多的好友？
- 新浪微博为了增加用户热度，提高用户留存性，需要微博用户在关注更多的人，以此获得更多的信息或热门话题，如何提高用户关注他人的总量？
- QQ新用户入网年龄越来越低，这些用户的朋友圈交际圈非常小，往往集中在一所学校甚至一个班级中，如何帮助用户快速积累好友用户带来更多的活跃度？
- 微信公众号是微信信息流通的渠道之一，增加用户关注的公众号成为提高用户活跃度的一种方式，如何帮助用户积累更多关注的公众号？
- 美团外卖为了提升成单量，必须帮助用户挖掘美食需求，如何推荐给用户最适合自己的美食？

解决方案:

1.求两个集合的交、并、差集

```
sinter key1 [key2]  
sunion key1 [key2]  
sdiff key1 [key2]
```

2.求两个集合的交、并、集并存储到指定集合中

```
sinterstore destination(目标集合) key1 [key2]  
sunionstore destination key1 [key2]  
sdiffstore destination key1 [key2]
```

3.将指定数据从原始集合中移动到目标集合中

```
smove source dest ination member
```

Tips 9: redis 应用于同类信息的关联搜索，二度关联搜索，深度关联搜索

显示共同关注(一度)、显示共同好友(一度)

由用户A出发， 获取到好友用户B的好友信息列表(一度)

由用户A出发， 获取到好友用户B的购物清单列表(二度)

由用户A出发, 获取到好友用户B的游戏充值列表(二度)

2.4.5 set类型数据操作的注意事项

1. set类型不允许数据重复，如果添加的数据在set中已经存在，将只保留一份
2. set 虽然与hash的存储结构相同，但是无法启用hash中存储值的空间

2.4.6 Set类型应用场景-权限校验

业务场景

集团公司共具有12000名员工，内部OA系统中具有700多个角色，3000多个业务操作，23000多种数据,每

位员工具有一个或多个角色，如何快速进行业务操作的权限校验？

解决方案

- 依赖set集合数据不重复的特征，依赖set集合hash存储结构特征完成数据过滤与快速查询
- 根据用户id获取用户所有角色
- 根据用户所有角色获取用户所有操作权限放入set集合
- 根据用户所有角色获取用户所有数据全选放入set集合

疑问=>校验工作，redis提供基础数据还是提供校验结果？

Tips 10: redis 应用于同类型不重复数据的合并操作

2.4.7 set类型应用场景-网站访问量统计

业务场景：

公司对旗下新的网站做推广,统计网站的PV (访问量),UV (独立访客),IP (独立IP):

- PV:网站被访问次数，可通过刷新页面提高访问量
- UV:网站被不同用户访问的次数，可通过cookie统计访问量，相同用户切换IP地址, UV不变
- IP:网站被不同IP地址访问的总次数，可通过IP地址统计访问量,相同IP不同用户访问，IP不变

解决方案：

- 利用set集合的数据去重特征，记录各种访问数据
- 建立string类型数据，利用incr统计日访问量(PV)
- 建立set模型，记录不同cookie数量(UV)
- 建立set模型，记录不同IP数量(IP)

Tips 11: Redis 应用于同类型数据的快速去重

2.4.8 Set类型应用场景-实现黑白名单

业务场景：

1.黑名单：

资讯类信息类网站追求高访问量,但是由于其信息的价值,往往容易被不法分子利用，通过爬虫技术,快速获取信息，个别特种行业网站信息通过爬虫获取分析后，可以转换成商业机密进行出售。例如第三方火车票、机票、酒店刷票代购软件,电商刷评论、刷好评。

同时爬虫带来的伪流量也会给经营者带来错觉，产生错误的决策，有效避免网站被爬虫反复爬取成为每个网站都要考虑的基本问题。在基于技术层面区分出爬虫用户后，需要将此类用户进行有效的屏蔽,这就是黑名单的典型应用。

ps:不是说爬虫一定做摧毁性的工作,有些小型网站需要爬虫为其带来一些流量。

2.白名单

对于安全性更高的应用访问，仅仅靠名单是不能解决安全问题的，此时需要设定可访问的用户群体，依赖白名单做更为苛刻的访问验证。

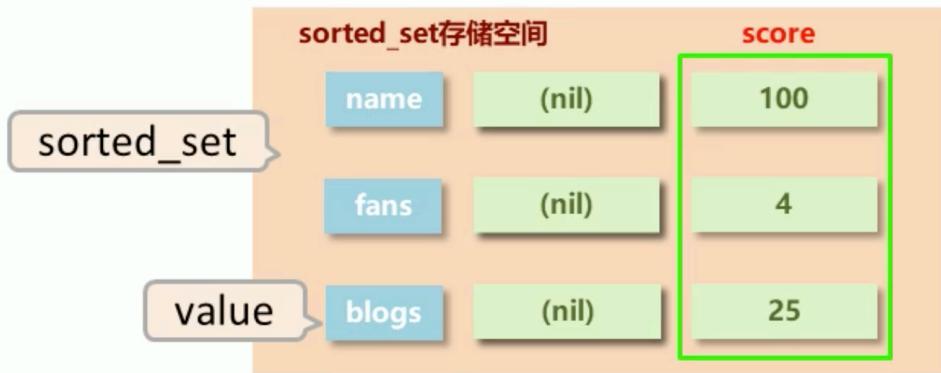
解决方案：

- 基于经营战略设定问题用户发现、鉴别规则
- 周期性更新满足规则的用户名单,加入set集合
- 用户行为信息达到后与黑名单进行比对,确认行为去向
- 黑名单过滤IP地址:应用于开放游客访问权限的信息源
- 黑名单过滤设备信息:应用于限定访问设备的信息源
- 黑名单过滤用户:应用于基于访问权限的信息源

2.5 Sorted_Set类型

2.5.1 为什么需要sorted_set？

- 新的存储需求:数据排序有利于数据的有效展示，需要提供1种可以根据自身特征进行排序的方式
- 需要的存储结构: 新的存储模型，可以保存可排序的数据
- sorted_set类型:在set的存储结构基础上添加可排序字段



2.5.2 sorted set类型数据的基本操作

- 添加数据

```
zadd key score1 member1 [score2 member2] //score理解为排序字段，不要理解为数据
```

- 获取全部数据

```
zrange key start stop [WITHSCORES]
zrevrange key start stop [WITHSCORES]
```

- 删除数据

```
zrem key member [member . . . ]
```

- 按条件获取数据

```
zrangebyscore key min max [WITHSCORES] [limit]
```

- zrangebyscore scores 50 99 limit 0 3 withscores //limit类似于MySQL

- zrevrangebyscore key max min [WITHSCORES]

- 条件删除数据

```
zremrangebyrank key start stop
zremrangebyscore key min max
```

注意：

min与max用于限定搜索查询的条件

start与stop用于限定查询范围,作用于索引, 表示开始和结束索引

offset与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

- 获取集合数据总量

```
zcard key
zcount key min max
```

- 集合交、并操作

```
zinterstore destination(目标集合) numkeys key [key...]
//交集的score值, 会累加各集合的score值
```

```
zunionstore destination numkeys key [key . . . ]
```

2.5.3 sorted set类型数据的扩展操作

业务场景：

- 票选广东十大杰出青年, 各类综艺选秀海选投票
- 各类资源网站TOP10(电影,歌曲, 文档,电商,游戏等)
- 聊天室活跃度统计

- 游戏好友亲密度

业务分析：

为所有参与排名的资源建立排序依据

解决方案：

- 获取数据对应的索引(排名):

`zrank key member`

`zrevrank key member`

- score值获取与修改

`zscore key member`

`zincrby key increment member`

Tips 13: redis 应用于计数器组合排序功能对应的排名

2.5.4 sorted set类型数据操作的注意事项

score保存的数据存储空间是64位，如果是整数范围是-9007199254740992~9007 199254740992

score保存的数据也可以是一个双精度的double值,基于双精度浮点数的特征，可能会丢失精度,使用时候要慎重

sorted set底层存储还是基于set结构的，因此数据不能重复，如果重复添加相同的数据，score值将被反

复覆盖，保留最后一次修改的结果

2.5.5 sorted set 类型应用场景 - 时效性任务管理

业务场景：

- 基础服务+增值服务类网站会设定各位会员的试用，让用户充分体验会员优势。
- 例如观影试用VIP、游戏VIP体验、云盘下载体验VIP、数据查看体验VIP。
- 当VIP体验到期后，如果有效管理此类信息。即便对于正式VIP用户也存在对应的管理方式。
- 网站会定期开启投票讨论,限时进行,逾期作废。如何有效管理此类过期信息。

解决方案：

对于基于时间线限定的任务处理，将处理时间记录为score值,利用排序功能区分处理的先后顺序记录下一个要处理的时间，当到期后处理对应任务,移除redis中的记录,并记录下一个要处理的时间，当新任务加入时，判定并更新当前下一个要处理的任务时间

为提升sorted set的性能,通常将任务根据特征存储成若干个sorted set。例如1小时内，天内，周内，年度等,操作时逐级提升,将即将操作的若干个任务纳入到1小时内处理的队列中

获取当前系统时间：`time`

Tips 14: redis 应用于定时任务执行顺序管理或任务过期管理

2.5.6 sorted set 类型应用场景-带有权重的任务管理

业务场景:

任务/消息权重设定应用:

当任务或者消息待处理, 形成了任务队列或消息队列时, 对于高优先级的任务要保障对其优先处理, 如何实现任务权重管理。

解决方案:

对于带有权重的任务, 优先处理权重高的任务, 采用score记录权重即可

多条件任务权重设定:

如果权重条件过多时, 需要对排序score值进行处理, 保障score值能够兼容2条件或者多条件, 例如外贸订单优先于国内订单, 总裁订单优先于员工订单, 经理订单优先于员工订单

因score长度受限, 需要对数据进行截断处理, 尤其是时间设置为小时或分钟级即可(折算后)
先设定订单类别, 后设定订单发起角色类别, 整体score长度必须是统一的, 不足位补0。第一排序规则首位不得是0

例如:

- 外贸101, 国内102, 经理004, 员工008。
- 员工下的外贸单score值为101008 (优先)
- 经理下的国内单score值为102004

Tips 15: redis 应用于即时任务/消息队列执行管理

2.6 数据类型实践案例1：按次结算的服务控制

业务场景:

人工智能领域的语义识别与自动对话将是未来服务业机器人应答呼叫体系中的重要技术, 百度自研用户评价

语义识别服务, 免费开放给企业试用, 同时训练百度自己的模型。

现对试用用户的使用行为进行限速, 限制每个用户每分钟最多发起10次调用

解决方案:

流程: 用户调动api => 计数器++, 生命周期执行, $x \geq 10$? 继续调用api: 停止调用api

1. 设计计数器, 记录调用次数, 用于控制业务执行次数。
2. 以用户id作为key, 使用次数作为value
3. 在调用前获取次数, 判断是否超过限定次数
4. 不超过次数的情况下, 每次调用计数+1, 业务调用失败, 计数-1
5. 为计数器设置生命周期为指定周期, 例如1秒/分钟, 自动清空周期内使用次数

解决方案改良:

1. 取消最大值的判定, 利用incr操作超过, 最大值抛出异常的形式替代每次判断是否大于最大值,
2. 判断是否为nil, ?设置为Max次数:计数+1
3. 业务调用失败, 计数-1
4. 遇到异常即++操作超过上限, 视为使用达到上限

Tips 16: redis应用于限时按次结算的服务控制

2.7 数据类型实践案例2：微信接收消息顺序控制

业务场景：

使用微信的过程中，当微信接收消息后，会默认将最近接收的消息置顶，当多个好友及关注的订阅号同时发

送消息时，该排序会不停的进行交替。同时还可以将重要的会话设置为置顶。一旦用户离线后，再次打开微

信时，消息该按照什么样的顺序显示？

解决方案：

- 依赖list的数据具有顺序的特征对消息进行管理,将list结构作为栈使用
- 对置顶与普通会话分别创建独立的list分别管理
- 当某个list中接收到用户消息后，将消息发送方的id从list的一侧加入list (此处设定左侧)
- 多个相同id发出的消息反复入栈会出现问题，在入栈之前无论是否具有当前id对应的消息，先删除对应id
- 推送消息时先推送置顶会话list,再推送普通会话list,推送完成的list清除所有数据
- 消息的数量，也就是微信用户对话数量采用计数器的思想另行记录,伴随list操作同步更新

Tips 17: redis 应用于基于时间顺序的数据操作，而不关注具体时间

三、Redis通用指令

3.1 key的通用指令

1.key特征

- key是一个字符串，通过key获取redis中保存的数据

2.key应该设计哪些操作？

- 对于key自身状态的相关操作，例如:删除，判定存在，获取类型等
- 对于key有效性控制相关操作,例如:有效期设定，判定是否有效，有效状态的切换等
- 对于key快速查询操作,例如:按指定策略查询key

3.1.1 key的基本操作

- 删除指定key
del key
- 获取key是否存在
exists key
- 获取key的类型
type key

3.1.2 key扩展操作：

1. 时效性控制

- 为指定key设置有效期

```

    expire key seconds
    pexpire key milli seconds
    expireat key timestamp
    pexpireat key milli seconds- timestamp

```

- 获取key的有效时间

```

    ttl key
    pttl key

```

- 切换key从时效性转换为永久性

```

    persist key

```

2. 查询模式

查询key	keys pattern
查询模式规则	
*	匹配任意数量的任意符号
?	匹配一个任意符号
[]	匹配一个指定符号
keys *	查询所有
keys it*	查询所有以it开头
keys *heima	查询所有以heima结尾
keys ??heima	查询所有前面两个字符任意，后面以heima结尾
keys user: ?	查询所有以user:开头，最后一个字符任意
keys u[st]er:1	查询所有以u开头，以er:1结尾，中间包含一个字母，s或t

3.1.3 key其他操作

- 为key改名

```

    rename key newkey
    renamenx key newkey

```

- 对所有key排序

```

    sort

```

- 其他key通用操作

```

    help @generic

```

3.2 数据库(db)的通用指令

1.为什么要有db => key的重复问题

- key是由程序员定义的
- redis在使用过程中,伴随着操作数据量的增加，会出现大量的数据以及对应的key
- 数据不区分种类、类别混杂在一起，极易出现重复或冲突

2.解决方案：

- redis为每个服务提供有16个数据库,编号从0到15
- 每个数据库之间的数据相互独立

3.2.1 db基本操作

- 其他操作

```
quit  
ping  
echo message
```

- 数据移动

```
move key db
```

- 数据清除

```
dbsize // 当前库里有多少key  
flushdb // 清理当前库  
flushall // 清理所有库
```

四、Jedis

4.1 Jedis入门

4.1.1 编程语言与redis

1. Java语言连接redis服务

Jedis(JDK8, 大于8不可以)

SpringData Redis

Lettuce

2. 其他

C、C++、C#、Erlang、Lua、Objective-C、Perl、PHP、Python、Ruby、Scala

4.1.2 环境搭建与HelloWorld

1. 环境搭建

```

<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.2.0</version>
</dependency>
<!--Junit-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>

```

2.HelloWorld

```

public class JedisDemo1 {
    public static void main(String[] args) {
        String testList = "test_list";
        String testString = "test_String";
        String testHash = "test_hash";
        //建立连接
        Jedis jedis = new Jedis("127.0.0.1",6379);
        //      jedis.del("test","test_list1");
        //关闭连接
        jedis.close();
    }
    public static void hashAdd(Jedis jedis,String test){
        jedis.hset(test,"f1","v1");
        jedis.hset(test,"f2","v2");
        jedis.hset(test,"f3","v3");
        Map<String, String> stringMaps = jedis.hgetAll(test);
        System.out.println(stringMaps);
    }
    public static void stringAdd(Jedis jedis,String test){
        jedis.set(test,"HelloWorld");
        System.out.println(jedis.get(test));
    }
    public static void listAdd(Jedis jedis,String test){
        jedis.lpush(test,"a","b","c");
        jedis.rpush(test,"d");
        List<String> strs = jedis.lrange(test,0,-1);
        System.out.println(strs);
    }
}

```

4.2 案例-服务调用次数控制

人工智能领域的语义识别与自动对话将是未来服务业机器人应答呼叫体系中的重要技术，百度自研用户评价语义识别服务，免费开放给企业试用，同时训练百度自己的模型。现对试用用户的使用行为进行限速，限制每个用户每分钟最多发起10次调用

1.案例要求：

- 设定A、B、C三个用户
- A用户限制10次/分调用，B用户限制30次/分调用，C用户不限制

2.需求分析：

- 设定一个服务方法，用于模拟实际业务调用的服务，内部采用打印模拟调用
- 在业务调用前服务调用控制单元，内部使用redis进行控制，参照之Z前的方案
- 对调用超限使用异常进行控制，异常处理设定为打印提示信息
- 主程序启动3个线程，分别表示3种不同用户的调用

3.代码演示：

```
/*
 * 模拟业务：限制用户调用API次数
 */
public class JedisDemo2 {
    public static void main(String[] args) {
        //多线程模拟多用户
        Mythread mt1 = new Mythread("初级用户", 10);
        Mythread mt2 = new Mythread("高级用户", 30);
        mt1.start();
        mt2.start();
    }
}

/**
 * 业务类
 */
class MyService {
    private String id;
    private int num;

    //构造函数，不同用户传入 => 不同的id、不同的调用次数
    public MyService(String id, int num) {
        this.id = id;
        this.num = num;
    }

    //控制单元
    public void service() {
        Jedis jedis = new Jedis("127.0.0.1", 6379);
        String value = jedis.get("compid:" + id);
        try {
            if (value == null) {
                //不存在，创建该值，初始值为最大整形-10，自增10次后数据溢出
                jedis.setex("compid:" + id, 30, Long.MAX_VALUE - num + "");
            }
        } catch (Exception e) {
            System.out.println("发生异常：" + e.getMessage());
        }
    }
}
```

```

        } else {
            //执行业务
            Long val = num - (Long.MAX_VALUE - jedis.incr("compid:" +
id));
            business(id, val);
        }
    } catch (Exception e) {
        System.out.println(id + " 调用次数达到上限, 请升级会员级别");
        return;
    } finally {
        jedis.close();
    }
}

//业务执行
public void business(String id, Long val) {
    System.out.println("用户 " + id + " MyService.....执行" + val + "次");
}

}

/**
 * 线程类
 */
class Mythread extends Thread {
    MyService ms;
    //构造函数, new一个线程对象 => 对应其用户性质、调用次数
    public Mythread(String id, int num) {
        this.ms = new MyService(id, num);
    }

    @Override
    public void run() {
        while (true) {
            ms.service();
            try {
                Thread.sleep(300L);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

4.3 Redis工具类制作

```

public class JedisPoolUtil {
    private static JedisPool jp;
    private static String host;
    private static int port;
    private static int maxTotal;

```

```

private static int maxIdle;

//静态代码块,初始化信息静态加载 => 改为配置文件方式加载连接信息
static {
    JedisPoolConfig jpc = new JedisPoolConfig();
    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"));
    //最大连接数
    jpc.setMaxTotal(maxTotal);
    //最大活动连接数
    jpc.setMaxIdle(maxIdle);
    jp = new JedisPool(jpc, host, port);
}

//设置静态方法, 方法在类加载时候初始化, 且唯一
public static Jedis getJedis() {
    return jp.getResource();
}
}

```

```

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

```

五、Linux安装Redis

5.1 redis.conf配置

基本配置

```

port 1111 # 配置端口号
daemonize yes # 是否后台运行 daemonize yes/no
logfile /var/log/redis.log # 日志文件位置
dbfilename dump.rdb # RDB持久化数据文件
dir /data/redis #持久化文件的位置

```

1.服务器端设定

- 设置服务器以守护进程的方式运行: `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`

2.日志配置

设置服务器以指定日志记录级别: `loglevel debug|verbose|notice|warning`

日志记录文件名: `logfile 端口号.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

3.客户端配置

- 设置同一时间最大客户端连接数, 默认无限制; 当客户端连接到达上限, Redis会关闭新的连接
- `maxclients 0`: 客户端闲置等待最大时长, 达到最大值后关闭连接。如需关闭该功能, 设置为0
- `timeout 300`

4.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/server- 端口号.conf`

5.2 其他见course: 46-50

Redis高级

六、持久化

6.1 持久化简介

1.什么是持久化

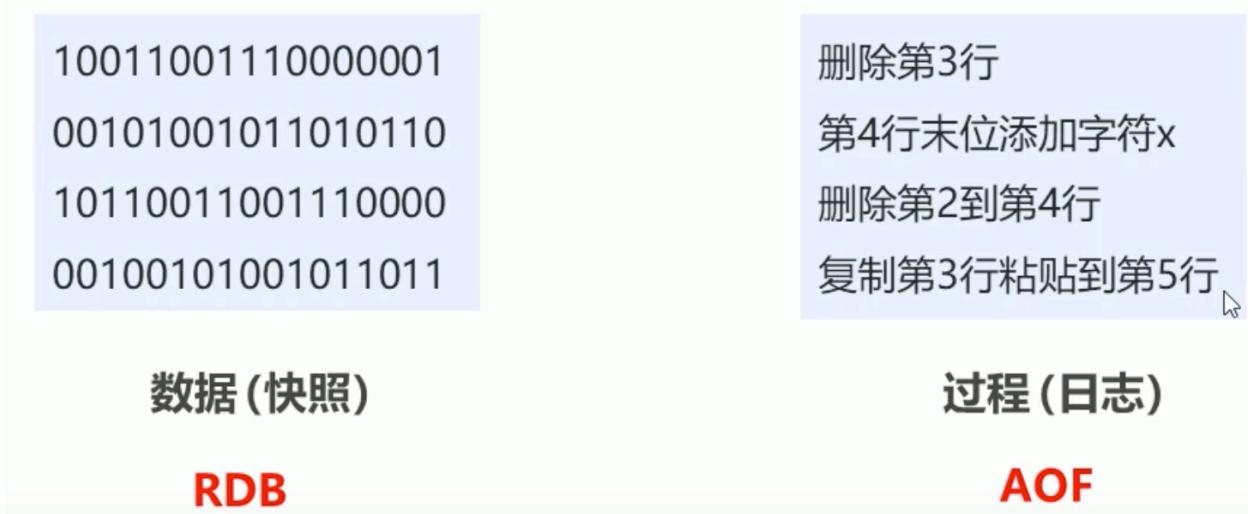
利用永久性存储介质将数据进行保存,在特定的时间将保存的数据进行恢复的工作机制称为持久化。

2.为什么要进行持久化

防止数据的意外丢失, 确保数据安全性

3.持久化过程保存什么

- 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
- 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



6.2 RDB

6.2.1 RDB启动方式—save指令

```
127.0.0.1:6379>save //保存到了/usr/local/redis-5.0.7/ 以及 messi433/
```

6.2.2 RDB启动方式—save指令相关配置

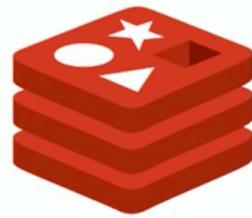
~ redis.conf/redis-6379.conf 下修改即可

- `dbfilename dump.rdb`
说明:设置本地数据库文件名, 默认值为dump.rdb
经验:通常设置为dump-端口号.rdb
- `dir`
说明:设置存储.rdb文件的路径
经验:通常设置成存储空间较大的目录中, 目录名称data
- `rdbcompression yes`
说明:设置存储至本地数据库时是否压缩数据,默认为yes,采用LZF压缩
经验:通常默认为开启状态,如果设置为no,可以节省CPU运行时间, 但会使存储的文件变大(巨大)
- `rdbchecksum yes`
说明:设置是否进行RDB文件格式校验,该校验过程在写文件和读文件过程均进行
经验:通常默认为开启状态, 如果设置为no,可以节约读写性过程约10%时间消耗, 但是存储一定的数据损坏风险
- `stop-writes-on-bgsave- erroryes`
说明:后台存储过程中如果出现错误现象, 是否停止保存操作
经验:通常默认为开启状态

6.2.3 save指令工作原理

1.单线程工作原理 => save指令的弊端

客户端1 127.0.0.1:6379>set key1 value1



客户端2 127.0.0.1:6379>set key2 value2

客户端3 127.0.0.1:6379>save

客户端4 127.0.0.1:6379>get key1

..... get save set set

单线程任务执行序列



注意: save指令的执行会阻塞当前Redis服务器, 直到当前RDB过程完成为止, 有可能会造成长时间阻塞, 线上环境不建议使用

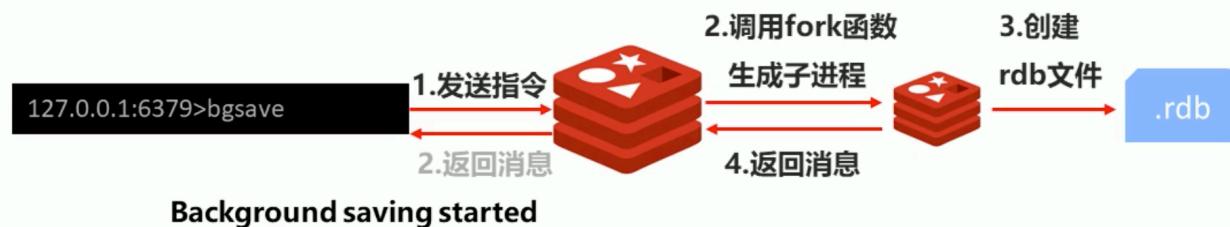
2. 数据量过大, 单线程执行方式造成效率过低如何处理?

- bgsave指令 =>后台执行

```
127.0.0.1:6379>bgsave //保存到了usr/local/redis-5.0.7/ 以及 messi433/  
Background saving started
```

6.2.4 bgsave执行工作原理

原理图:



日志文件:

mac下未找到日志文件

6.2.5 save配置

1. 反复执行保存指令, 忘记了怎么办? 不知道数据产生了多少变化, 何时保存?

- 配置: save second changes
- 作用: 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
- 参数:
 - second: 监控时间范围
 - changes: 监控key的变化量

- 位置：在conf文件中进行配置

- 范例：

```
save 900 1
save 300 10
save 60 10000 //通常为前大后小
```

2. 配置原理：



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

2.save配置中对于second与changes设置通常具有互补对应关系, 尽量不要设置成包含性关系

3.save配置启动后执行的是bgsave操作

6.2.6 RDB三种启动方式对比

方式	save指令	bgsave指令
读写	同步	异步
阻塞客户端指令	是	否
额外内存消耗	否	是
启动新进程	否	是

6.2.7 RDB特殊启动方式

- 全量复制
在主从复制中详细讲解
- 服务器运行过程中重启
debug reload
- 关闭服务器时指定保存数据
shutdown save

6.2.8 RDB优缺点

1.RDB优点

- RDB是一个紧凑压缩的二进制文件，存储效率较高
- RDB内部存储的是redis在某个时间点的数据快照，非常适合用于数据备份,全量复制等场景
- RDB恢复数据的速度要比AOF快很多
- 应用: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。

2.RDB缺点

- RDB方式无论是执行指令还是利用配置，无法做到实时持久化，具有较大的可能性丢失数据
- bgsave指令每次运行要执行fork操作创建子进程,要牺牲掉一些性能
- Redis的众多版本中未进行RDB文件格式的版本统一，有可能出现各版本服务之间数据格式无法兼容现象

6.3 AOF

6.3.1 AOF的引出

RDB存储的弊端

- 存储数据量较大，效率较低
基于快照思想，每次读写都是全部数据，当数据量巨大时,效率非常低
- 大数据量下的IO性能较低
- 基于fork创建子进程，内存产生额外消耗
- 宕机带来的数据丢失风险

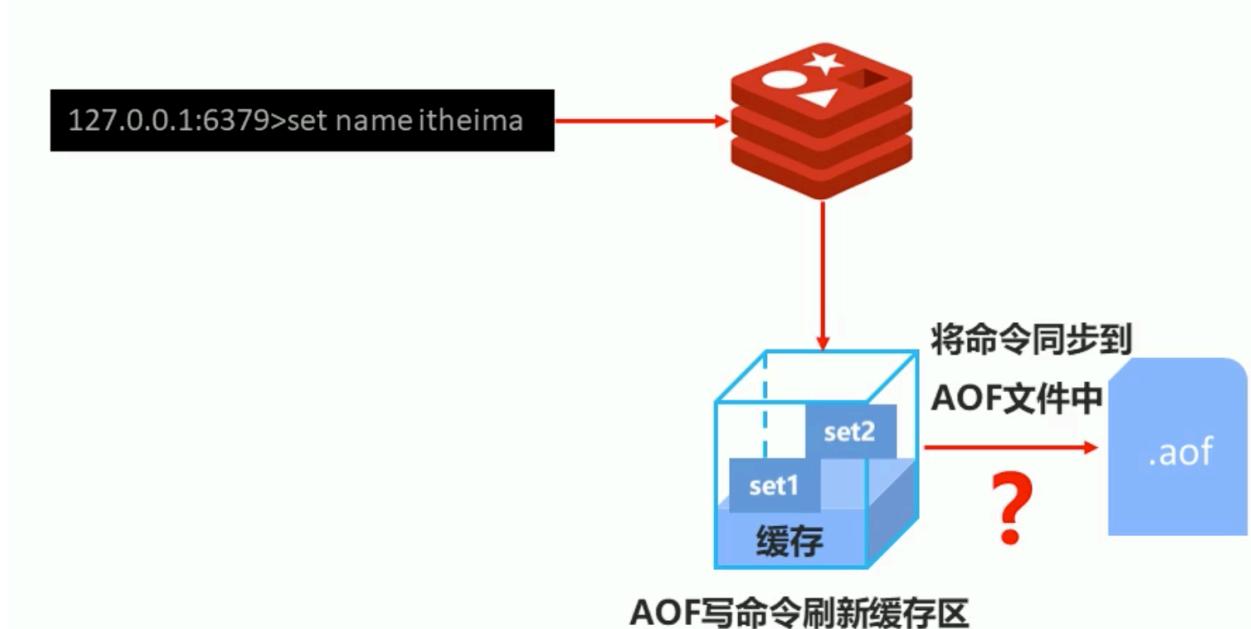
AOF的引出

- 不写全数据，仅记录部分数据
- 改记录数据为记录操作过程
- 对所有操作均进行记录，排除丢失数据的风险

AOF的概念

- 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文件中命令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

6.3.2 AOF写数据过程



AOF写数据三种策略(appendfsync)

- always(每次)
每次写入操作均同步到AOF文件中，数据零误差，性能较低，不建议使用。
- everysec (每秒)
每秒将缓冲区中的指令同步到AOF文件中，数据准确性较高,性能较高，建议使用，也是默认配置，在系统突然宕机的情况下丢失1秒内的数据
- no (系统控制)
由操作系统控制每次同步到AOF文件的周期，整体过程不可控

AOF功能开启

- 配置1: appendonly yes | no
作用：是否开启AOF持久化功能，默认为开启状态
- 配置2: appendfsync always | everysec | no
作用：AOF写数据策略
- 配置3: appendfilename filename
作用：AOF持久化文件名,默认文件名未appendonly.aof, 建议配置为appendonly 端口号.aof
- 配置4: dir
作用：AOF持久化文件保存路径，与RDB持久化文件保持一致即可

6.3.3 AOF写数据遇到的问题



如果连续执行如下指令该如何处理

```
127.0.0.1:6379>set name zs  
127.0.0.1:6379>set name ls  
127.0.0.1:6379>set name ww  
127.0.0.1:6379>incr num  
127.0.0.1:6379>incr num  
127.0.0.1:6379>incr num
```

```
127.0.0.1:6379>set name ww  
127.0.0.1:6379>set num 3
```

AOF重写

1. 随着命令不断写入AOF文件会越来越大，为了解决这个问题，Redis引入了AOF重写机制压缩文件体积。
2. 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。
3. 简单说就是将对同一个数据的若干个条命令执行结果转化成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,提高磁盘利用率
2. 提高持久化效率,降低持久化写时间, 提高IO性能
3. 降低数据恢复用时,提高数据恢复效率

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`del key1, hdel key2, srem key3, set key4111, set key 4222`等
3. 对同一数据的多条写命令合并为一条命令
如`lpush list1 a, lpush list1 b, lpush list1 c`可以转化为:`lpush list1 a b c`
4. 为防止数据量过大造成客户端缓冲区溢出, 对`list、set、hash、zset`等类型,每条指令最多写入64个元素

AOF重写方式:

1. 手动重写: `bgrewriteaof`
2. 自动重写
`auto-aof rewrite min -size size`
`auto-aof- rewrite-percentage percentage`

AOF手动重写——`bgrewriteaof`指令工作原理



AOF自动重写方式

- 自动重写触发条件设置

```
auto-aof-rewrite min-size size
auto-aof-rewrite percentage percent
```

- 自动重写触发比对参数(运行指令info Persistence获取具体信息)

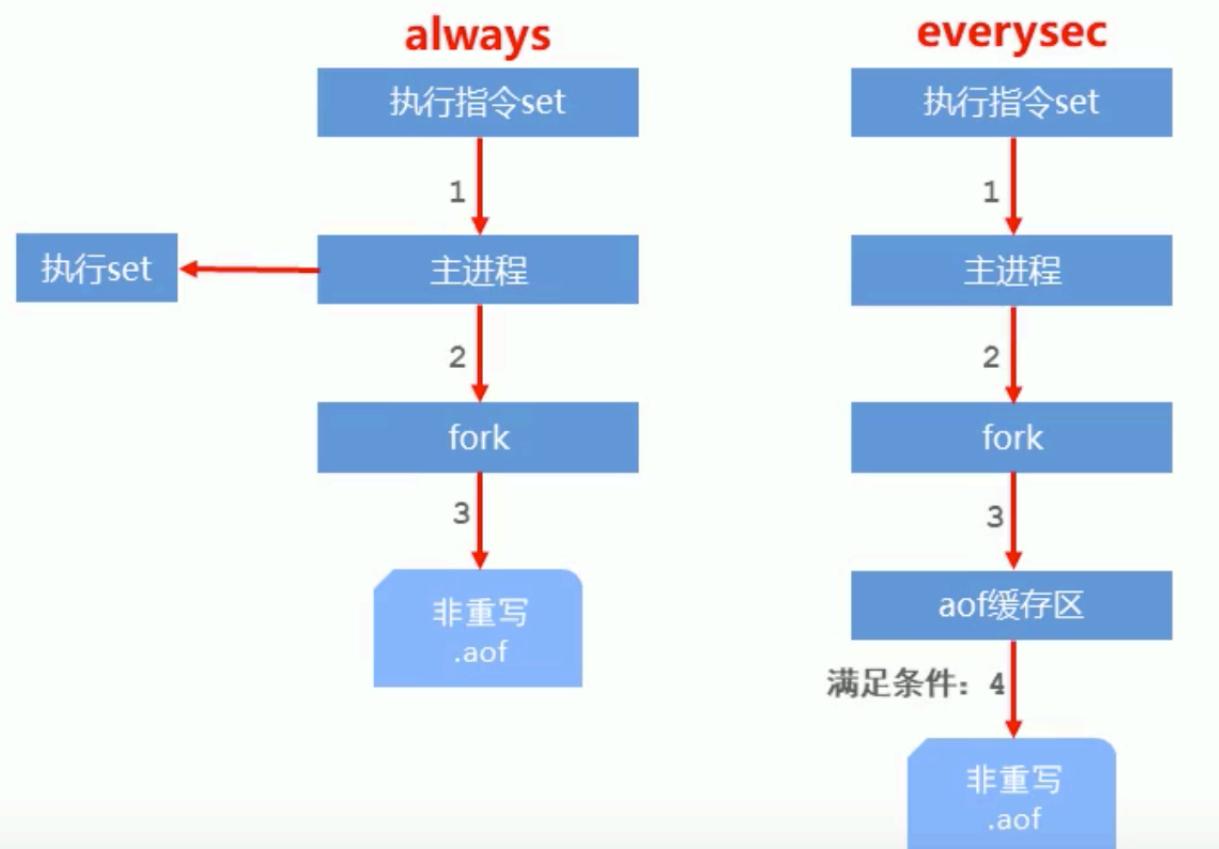
```
aof_current_size
aof_base_size
```

- 自动重写触发条件

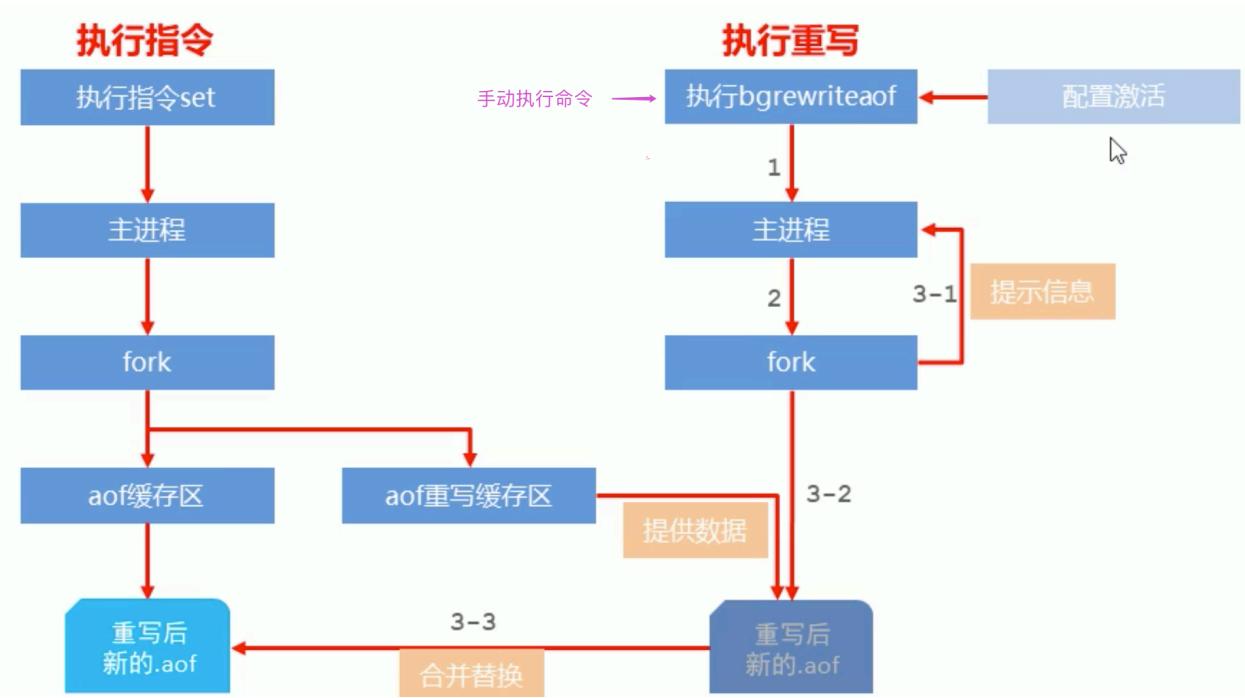
```
aof_current_size > auto-aof-rewrite-min-size
(aof_current_size - aof_base_size) / aof_base_size >= auto-aof-rewrite-percentage
```

6.3.4 AOF工作原理&重写流程

1. 工作原理:



2. 重写流程:



6.4 RDB与AOF的区别

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 (恢复数据)	慢 恢复指令
数据安全性	会丢失数据	依据策略决定
资源消耗	高/重量级	低/轻量级
启动优先级	低	高

RDB与AOF的选择之惑

- 对数据非常敏感，建议使用默认的AOF持久化方案
- AOF持久化策略使用everysecond,每秒钟fsync 1次。该策略redis仍可以保持很好的处理性能，当出现问题时，最多丢失0-1秒内的数据。
- 注意:由于AOF文件存储体积较大，且恢复速度较慢
- 数据呈现阶段有效性,建议使用RDB持久化方案
- 数据可以良好的做到阶段内无丢失(该阶段是开发者或运维人员手工维护的)，且恢复速度较快,阶段点数据恢复通常采用RDB方案
- 注意:利用RDB实现紧凑的数据持久化会使Redis降的很低
- RDB与AOF的选择实际上是在做一种权衡，每种都有利有弊
- 如不能承受数分钟以内的数据丢失，对业务数据非常敏感，选用AOF

- 如能承受数分钟以内的数据失，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，降低丢失数据的量

6.5 持久化应用场景

重要数据就持久化，否则没必要

- Tips1: redis用于控制数据库表主键i,为数据库表主键提供生成策略，保障数据库表的主键唯一性
- Tips3: redis应用于各种结构型和非结构型高热度数据访
- Tips4: redis 应用于购物车数据存储设计
- Tips5: redis 应用于抢购，限购类、限量发放优惠券、激活码等业务的数据存储设计
- Tips 6: redis 应用于具有操作先后顺序的数据控制
- Tips7: redis 应用于最新消息展示
- Tips9: redis 应用于同类信息的关联搜索，二度关联搜索，深度关联搜索
- Tips12: redis 应用于基于黑名单与白名单设定的服务控制
- Tips13: redis应用于计数器组合排序功能对应的排名
- Tips15: redis应用于即时任务/消息队列执行管理（通常使用RabbitMQ等框架）
- Tips16: redis 应用于按次结算的服务控制

七、Redis事务

7.1 Redis事务简介

7.1.1 Redis事务的引出

Redis执行指令过程中，多条连续执行的指令被干扰，打断，插队



7.1.2 Redis事务概念

- Redis事务就是一个命令执行的队列，将一系列预定义命令包装成一个整体(一个队列)。
- 当执行时，一次性按照添加顺序依次执行，中间不会被打断或者干扰。
- 一个队列中，一次性、顺序性、排他性的执行一列命令

7.2 事务的基本操作

7.2.1 基本命令

开启事务: `multi`

作用：设定事务的开启位置，此指令执行后，后续的所有指令均加入到事务中

执行事务: `exec`

作用：设定事务的结束位置，同时执行事务。与multi成对出现，成对使用

注意:加入事务的命令暂时进入到任务队列中，并没有立即执行，只有执行exec命令才开始执行

7.2.2 事务定义过程中发现出了问题，怎么办？

取消事务： `discard`

作用：终止当前事务的定义，发生在multi之后，exec之前

7.2.3 事务的注意事项

定义事务的过程中，命令格式输入错误怎么办？

语法错误：指命令书写格式有误

处理结果：如果定义的事务中所包含的命令存在语法错误，整体事务中所有命令均不会执行，包括那些语法正确的命令。

```
127.0.0.1:6379> multi
ok
127.0.0.1 :6379> set name itheima
QUEUED
127.0.0.1:6379> get name
QUEUED
127.0.0.1:6379>tes name itcast
(error)ERR unknown command tes
127.0.0.1 :6379>exec
(error)EXECABORT Transaction discarded because of previous errors
```

定义事务的过程中，命令执行出现错误怎么办？

运行错误：指命令格式正确，但无法正确的执行。例如对list进行incr操作

处理结果：能够正确运行的命令会执行，运行错误的命令不会被执行

```
127.0.0.1:6379> multi
oK
127.0.0.1:6379> set name itheima
QUEUED
127.0.0.1:6379> get name
QUEUED
127.0.0.1:6379> set name itcast
QUEUED
127.0.0.1:6379> get name
QUEUED
127.0.0.1:6379> lpush name a b c
QUEUED
127.0.0.1:6379> get name
QUEUED
127.0.0.1:6379> exec
ok
"itheima"
ok
"itcast"
(error)WRONGTYPE Operation against a key holding the wrong kind of value
itcast
```

注意:已经执行完毕的命令对应的数据不会自动回滚, 需要程序员自己在代码中实现回滚。

7.2.4 手动进行事务回滚(鸡肋)

记录操作过程中被影响的数据之前的状态

单数据: string

多数据: hash、list、set、zset

设置指令恢复所有的被修改的项

单数据:直接set (注意周边属性,例如时效)

多数据:修改对应值或整体克隆复制

7.3 事务-锁

7.3.1 基于特定条件的事务执行——锁

1.业务场景:

天猫双11热卖过程中, 对已经售罄的货物追加补货, 4个业务员都有权限进行补货。补货的操作可能是一系列的操作, 牵扯到多个连续操作, 如何保障不会重复操作?

2.业务分析:

多个客户端有可能同时操作同一组数据, 并且该数据一旦被操作修改后, 将不适用于继续操作
在操作之前锁定要操作的数据, 一发生变化, 终止当前操作

3.解决方案:

对key添加监视锁, 在执行exec前如果watch的key发生了变化, 终止事务执行, 必须要在事务之前开启watch

```
watch key1 [key.....]
```

4.取消对所有key的监视:

```
unwatch
```

Tips 18: redis应用基于状态控制的批量任务执行

7.3.2 基于特定条件的事务执行——分布式锁

1.业务场景

- 天猫双11热卖过程中, 对已经售罄的货物追加补货, 且补货完成。
- 客户购买热情高涨, 3秒内将所有商品购买完毕。
- 本次补货已经将库存全部清空, 如何避免最后一件商品不被多人同时购买? [超卖问题]

2.业务分析

使用watch监控一个key有没有改变已经不能解决问题, 此处要监控的是具体数据
虽然redis是单线程的, 但是多个客户端对同一数据同时进行操作时, 如何避免不被同时修改?

3.解决方案

- 使用setnx设置一个公共锁: `setnx lock-key value`
- 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

- 对于返回设置成功的，拥有控制权，进行下一步的具体业务操作
- 对于返回设置失败的，不具有控制权，排队或等待
- 操作完毕通过del操作释放锁

注意：上述解决方案是一种设计概念，依赖规范保障，具有风险性

Tips 19:redis 应用基于分布式锁对应的场景控制

7.3.3 基于特定条件的事务执行——分布式锁改良

1. 业务场景：

依赖分布式锁的机制，某个用户操作时对应客户端宕机，且此时已经获取到锁，如何解决？

某个用户获取到锁，但是他忘记了释放锁，怎么办？

2. 业务分析：

由于锁操作由用户控制加锁解锁，必定会存在加锁后未解锁的风险

需要解锁操作不能仅依赖用户控制，系统级别要给出对应的保底处理方案

3. 解决方案：

使用expire为锁key添加时间限定，到时不释放，放弃锁

```
expire lock-key second  
pexpire lock-key milliseconds
```

- 由于操作通常都是微妙或毫秒级，因此该锁定时间不宜设置过大，具体时间需要业务测试后确认
- 例如：持有锁的操作最长执行时间127ms,最短执行时间7ms。
- 测试百万次最长执行时间对应命令的最大耗时，测试百万次网络延迟平均耗时
- 锁时间设定推荐：最大耗时120%+平均网络延迟110%
- 如果业务最大耗时<<网络平均延迟，通常为2个数量级，取其中单个耗时较长即可

八、Redis删除策略

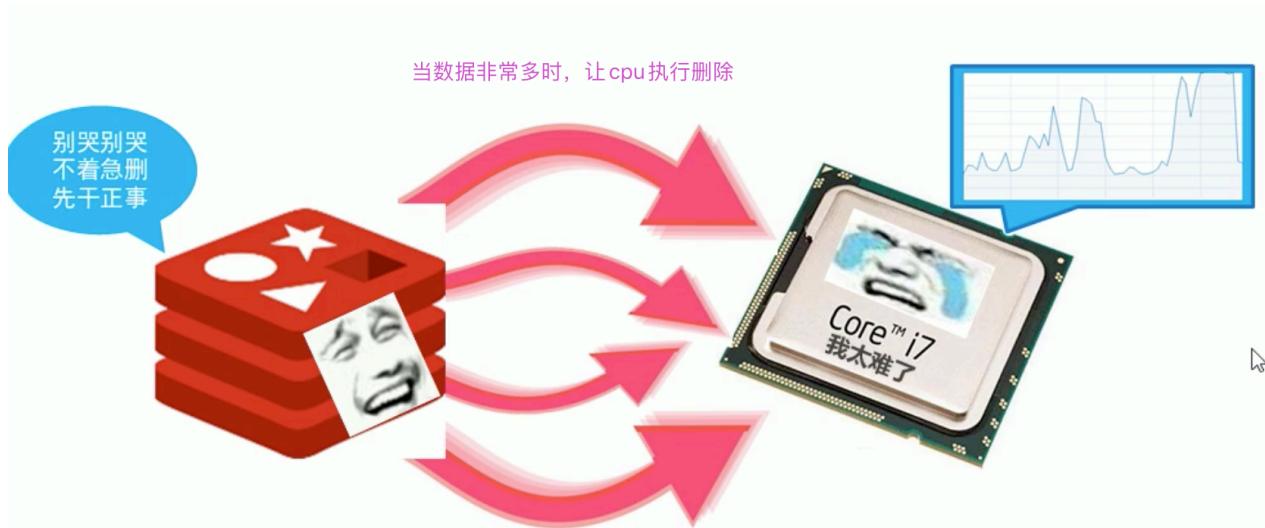
8.1 过期数据的概念

8.1.1 Redis中的数据特征

Redis是一种内存级数据库，所有数据均存放在内存中，内存中的数据可以通过TTL指令获取其状态

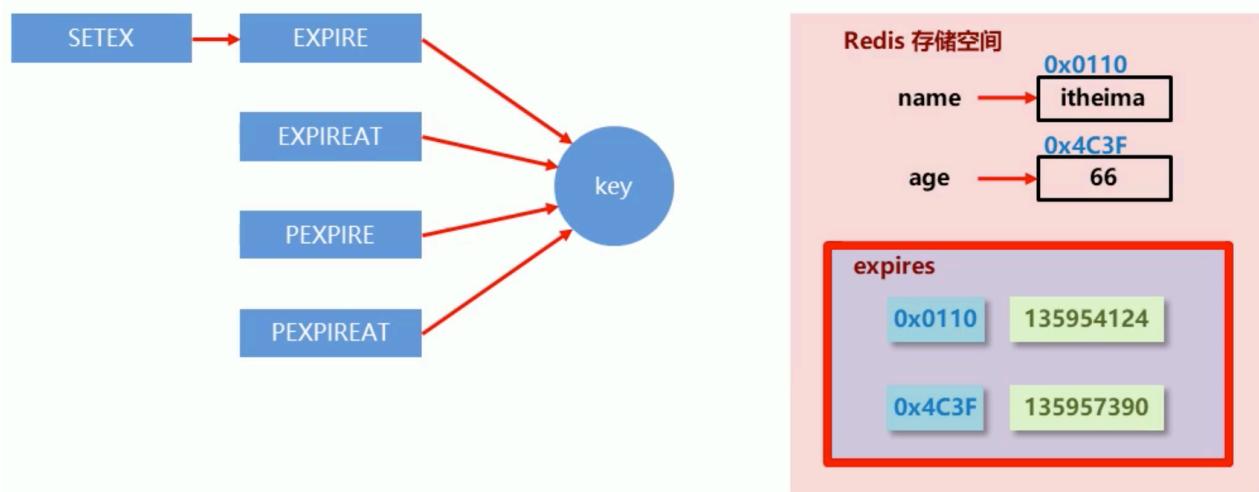
- XX：具有时效性的数据
- -1：永久有效的数据
- -2：已经过期的数据或被删除的数据或未定义的数据

过期的数据真的删除了吗？



8.2 过期数据

8.2.1 时效性数据的存储结构



8.2.2 数据删除策略

数据删除策略的目标

在内存占用与CPU占用之间寻找一种平衡，顾此失彼都会造成整体redis性能的下降，甚至引发服务器宕机或内存泄露

1. 定时删除

- 创建一个定时器，当key设置有过期时间，且过期时间到达时，由定时器任务立即执行对键的删除操作
- 优点：**节约内存，到时就删除，快速释放掉不必要的内存占用
- 缺点：**CPU压力很大，无论CPU此时负载量多高，均占用CPU，会影响redis服务器响应时间和指令吞吐量
- 总结：**处理器性能换取存储空间(拿时间换空间)

2. 惰性删除

- 数据到达过期时间，不做处理，等下次访问该数据时：

- 如果未过期，返回数据
- 发现已过期，删除，返回不存在
- 优点：节约CPU性能，发现必须删除的时候才删除
- 缺点：内存压力很大，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(拿时间换空间)

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2：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)

定期删除

-  两种方案都走极端，有没有折中方案？
- Redis启动服务器初始化时，读取配置server.hz的值，默认为10
 - 每秒钟执行server.hz次`serverCron()`
 - ↳ `databasesCron()`
 - ↳ `activeExpireCycle()`
 - `activeExpireCycle()`对每个`expires[*]`逐一进行检测，每次执行`250ms/server.hz`
 - 对某个`expires[*]`检测时，随机挑选W个key检测
 - 如果key超时，删除key
 - 如果一轮中删除的key的数量> $W \times 25\%$ ，循环该过程
 - 如果一轮中删除的key的数量≤ $W \times 25\%$ ，检查下一个`expires[*]`，0-15循环
 - W取值=ACTIVE_EXPIRE_CYCLE_LOOKUPS_PER_LOOP属性值
 - 参数`current_db`用于记录`activeExpireCycle()`进入哪个`expires[*]`执行
 - 如果`activeExpireCycle()`执行时间到期，下次从`current_db`继续向下执行



4. 删除策略比对

删除策略	内存占用	CPU占用	性能
定时删除	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
惰性删除	内存占用严重	延时执行，CPU利用率高	拿空间换时间
定期删除	内存定期随机清理	每秒花费固定的CPU资源维护内存	随机抽查，重点抽查

8.3 逐出算法

8.3.1 数据逐出概念

问题1：假设数据不是时效性的，全部都是长期存在的，那么数据如何清理？

问题2：当新数据进入redis时，如果内存不足怎么办？

- Redis使用内存存储数据，在执行每一个命令前，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为当前指令清理存储空

间。

- 清理数据的策略称为逐出算法。

注意:逐出数据的过程不是100%能够清理出足够的可使用的内存空间,如果不成功则反复执行。

当对所有数据尝试完毕后,如果不能达到内存清理的要求,将出现错误信息。

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

8.3.2 影响数据逐出的相关配置

- 最大可使用内存: `maxmemory`

占用物理内存的比例,默认值为0,不设置则不限制。生产环境中根据需求设定,通常设置在50%以上。

- 每次选取待删除数据的个数: `maxmemory-samples`

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

- 数据删除策略: `maxmemory -policy`

达到最大内存后的,对被挑选出来的数据进行删除的策略

- 检测易失数据(可能会过期的数据集`server.db[i].expires`)

① `volatile-lru`: 挑选最近最少使用的数据淘汰

② `volatile-lfu`: 挑选最近使用次数最少的数据淘汰

③ `volatile-ttl`: 挑选将要过期的数据淘汰

④ `volatile-random`: 任意选择数据淘汰

- 检测全库数据(所有数据集`server.db[j].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

- 放弃数据驱逐

⑧ `no-eviction` (驱逐): 禁止驱逐数据(redis4.0中默认策略),引发错误OOM (OutOf Memory)

```
maxmemory -policy volatile-lru
```

九、Redis高级数据类型

通常用来解决单一的数据业务

9.1 Bitmaps类型 (状态统计)

9.1.1 Bitmap基本使用

- 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, value只能是1或0: `setbit key offset value`

9.1.2 Bitmap扩展

业务场景

- 统计每天某一部电影是否被点播

- 统计每天有多少部电影被点播
- 统计每周/月/年有多少部电影被点播
- 统计年度哪部电影没有被点播



业务分析

`id`是5，就看第五位是否是1 看1的数量，即为每天点播数 假设这是某两天的点播数，用或操作求1

01010011	01010011	01010011	11011001
《非诚勿扰》	id:5	or 11011001	与操作3相反
offset:4		11011001	

- 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: 交 or: 并 not: 非 xor: 异或
```

- 统计指定key中1的数量

```
bitcount key [start end]
```

Tips21: redis应用信息状态统计

9.2 HyperLogLog (基数统计)

9.2.1 应用场景及基数

场景：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 ,
- 存储每个用户状态(bit)
- 全新的方案: 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的算法

基数示例:

- {1,3,5,7,5,7,8}
基数集: {1,3,5,7,8}
基数(个数): 5
- {1,1,1,1,1,7, 1}
基数集:{1,7}
基数: 2

9.2.2 HyperLogLog类型的基本操作

- 添加数据: pfadd key element [element .. .]
- 统计数据: pfcount key [key .. .]
- 合并数据: pfmerge des tkey sourcekey [sourcekey. ..]

Tips 22: redis应用于独立信息统计

9.2.3 HyperLogLog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
- 核心是基数估算算法,最终数值存在一定误差
- 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,无论合并之前数据多少

9.3 GEO

9.3.1 GEO的引出

场景: 火热的生活服务类软件

- 微信/陌陌
- 美团/饿了么
- 携程/马蜂窝
- 高德/百度



9.3.2 GEO类型的基本操作

- 添加坐标点:

```
geoadd key longitude latitude member [longitude latitude member . . . ]
```

- 获取坐标点:

```
geopos key member [member . . . ]
```

- 计算坐标点距离:

```
geodist key member1 member2 [unit]
```

- 根据坐标求范围内的数据:

```
georadius key longitude latitude radius[m|km|ft|mi] [withcoord] [withdist]
[withhash] [count count]
```

- 根据点求范围内数据:

```
georadiusbymember key member radius[m|km|ft|mi] [withcoord] [withdist]
[withhash] [count count]
```

- 获取指定点对应的坐标hash值:

```
geohash key member [member . . . ]
```

Tips 23: redis应用于地理位置计算

Redis集群

集群环境搭建

1.单机版伪分布式环境搭建: <https://blog.csdn.net/weberhuangxingbo/article/details/89488281>

2.虚拟机: <https://blog.51cto.com/14449521/2457827?source=dra>

十、主从复制

10.1 主从复制的简介

10.1.1 互联网三高架构

高并发

高性能

高可用

高可用的状态：服务器宕机时间极小

4小时27分15秒+11分36秒+2分16秒 = 4小时41分7秒 = 866467秒

1年 = 365*24*60*60 = 31536000

可用性 = $\frac{31536000 - 866467}{31536000} * 100\% = 97.252\%$

业界可用性目标5个9，即99.999%，即服务器年宕机时长低于315秒，约5.25分钟

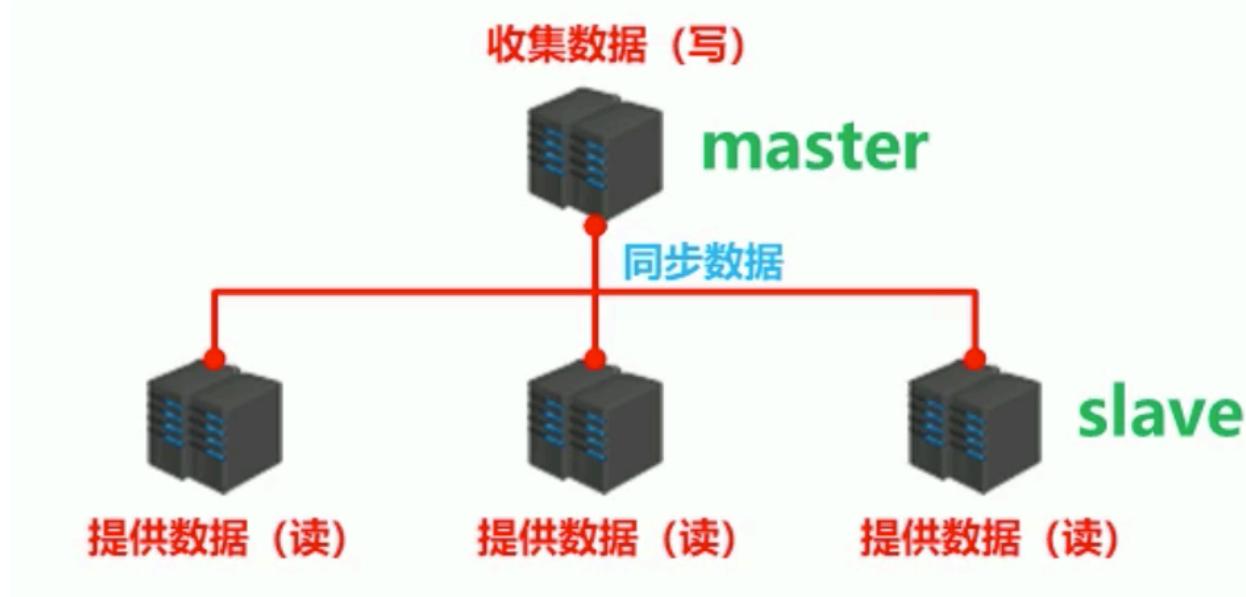
10.1.2 单机redis的风险与问题

- 问题1：机器故障 现象:硬盘故障、系统崩溃 本质:数据丢失，很可能对业务造成灾难性打击 结论:基本上会放弃使用redis.
- 问题2：容量瓶颈 现象:内存不足，从16G升级到64G，从64G升级到128G，无限升级内存 本质:穷，硬件条件跟不上 结论:放弃使用redis
- 结论:
 - 为了避免单点Redis服务器故障，准备多台服务器，互相连通。
 - 将数据复制多个副本保存在不同的服务器上，连接在一起，并保证数据是同步的。
 - 即使有其中一台服务器宕机，其他服务器依然可以继续提供服务，实现Redis的高可用，同时实现数据冗余备份。

10.1.3 主从复制概念

1.多台服务器连接方案

- 提供数据方: master 主服务器，主节点，主库，主客户端
- 接收数据方: slave 从服务器，从节点，从库，从客户端 需要解决的问题:数据同步
- 核心工作: master的数据复制到slave中

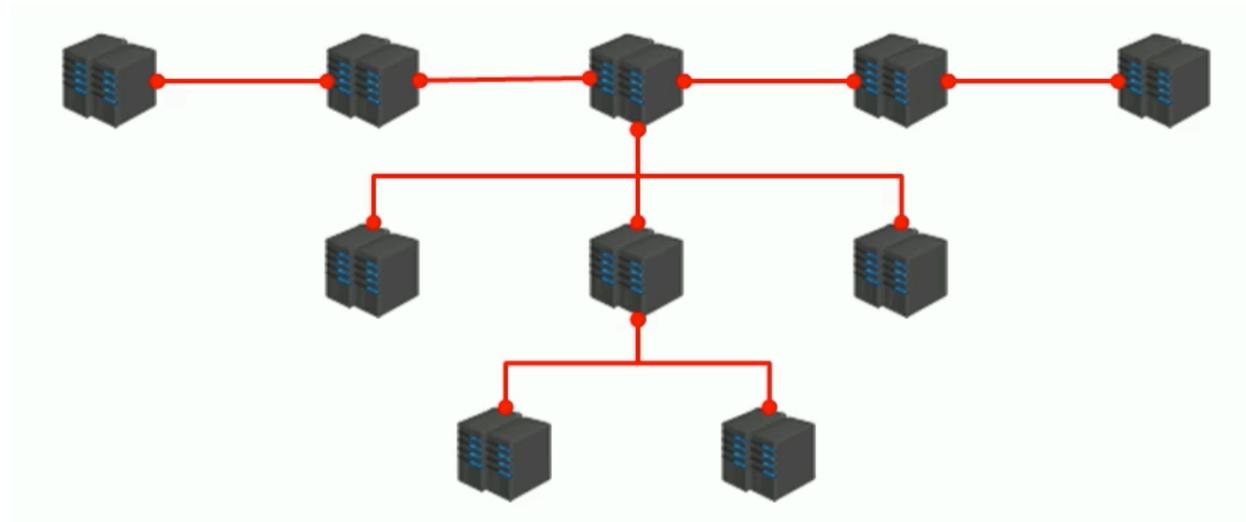


2. 主从复制

即将master中的数据即时、有效的复制到slave中，一个master可以拥有多个slave，一个slave只对应一个master

- master:
 - 写数据，执行写操作时，将出现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)

3. 高可用集群



10.1.4 主从复制的作用

- 读写分离：master写、slave读，提高服务器的读写负载能力
- 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
- 多个从节点分担数据读取负载，大大提高Redis服务器并发量与数据吞吐量

- 故障恢复：当master出现问题时，由slave提供服务，实现快速的故障恢复
- 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
- 高可用基石：基于主从复制，构建哨兵模式与集群，实现Redis的高可用方案

10.2 主从复制的工作流程

10.2.1 主从连接的准备工作

1.搭建环境

- 单机上，新加 `yourport.conf` 文件 => 修改port => 指定文件启动。
- 连接虚拟机上，模拟服务器(环境ubuntu18.04)

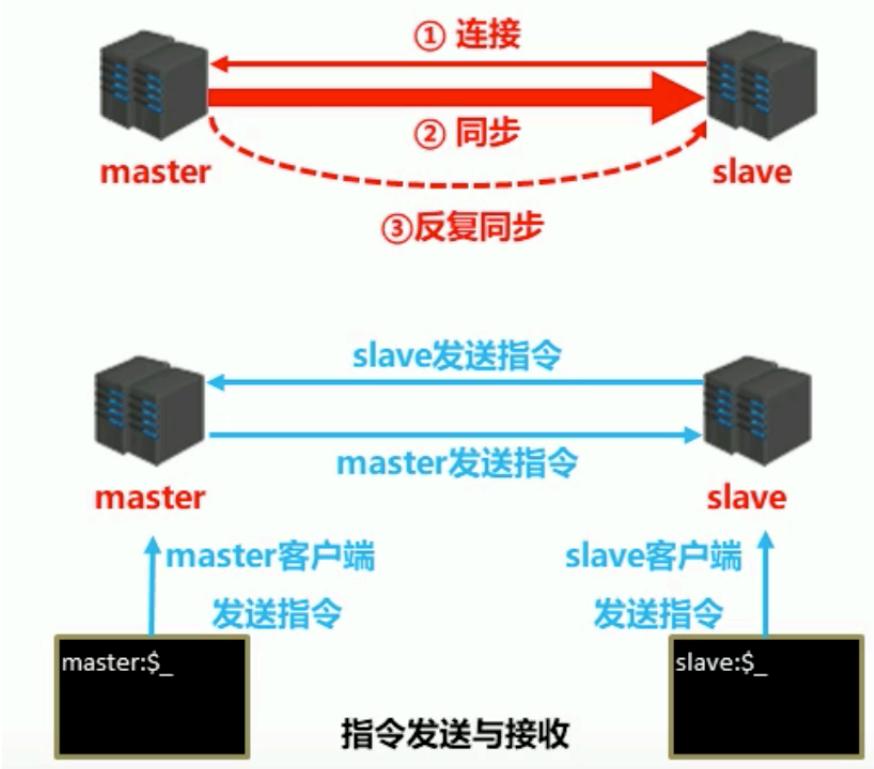
2.建立远程连接 =>修改 `.conf` 文件

```
/* 主从redis都建立远程连接*/
bind 127.0.0.1      //本地ip注掉
protected-mode no    //受保护模式关掉
```

10.2.2 工作流程总述

主从复制过程大体可以分为3个阶段：

1. 建立连接阶段(即准备阶段)
2. 数据同步阶段
3. 命令传播阶段



10.2.3 连接阶段的工作流程



10.3 主从连接—slave连接master

- 方式一：客户端发送命令 `slaveof`
- 方式二：启动服务器参数 `redis-server -slaveof`
- 方式三：服务器配置 `slaveof`

slave系统信息、master系统信息

```
master_ link_ down_ since_ seconds
slave_ listening_ port(多个)
masterhost
masterport
```

- 断开与master的连接 `slaveof no one`

10.3.1 授权访问

- master配置文件设置密码
`requirepass`
- slave客户端发送命令设置密码
`auth master`
- 客户端发送命令设置密码
`config set requirepass masterauth`
- slave配置文件设置密码
`config get requirepass`
- 启动客户端设置密码
`redis-cli -a`

10.4 数据同步阶段

- 在slave初次连接master后，复制master中的所有数据到slave

- 将slave的数据库状态更新成master当前的数据库状态

数据同步阶段的工作流程



10.4.1 数据同步阶段master说明

- 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，影响业务正常执行
- 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
- 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。

```
rep -backlog- size 1mb
```

- master单机内存占用主机内存的比例不应过大，建议使用50%-70%的内存，留下30%-50%的内存用于执行`bgsave`命令和创建复制缓冲区
- 为避免slave进行全量复制、部分复制时服务器响应阻塞或数据不同步，建议关闭此期间的对外服务

```
slave -serve-stale -data yes|no
```

- 数据同步阶段，master发送给slave信息可以理解master是slave的一个客户端，主动向slave发送命令
- 多个slave同时对master请求数据同步，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业务需求，适量错峰
- slave过多时，建议调整拓扑结构，由1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

10.5 命令传播阶段

- 当master数据库状态被修改后，导致主从服务器数据库状态不一致，此时需要让主从数据同步到一致的状态，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接收命令后执行命令

10.5.1 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

10.5.2 服务器运行ID (runid)

- 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可以生成多个运行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串
例如: `fdc9ff13b9bbaab28db42b3d50f852bb5e3fcde`
- 作用:运行id被用于在服务器间进行传输, 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的, 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, 可以查看节点的runid

10.5.3 复制缓冲区

概念:

- 复制缓冲区, 又名复制积压缓冲区, 是一个先进先出(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, 复制缓冲区默认数据存储空间大小是1M
- 由于存储空间大小是固定的, 当入队元素的数量大于队列长度时, 最先入队的元素会被弹出, 而新元素会被放入队列

由来:

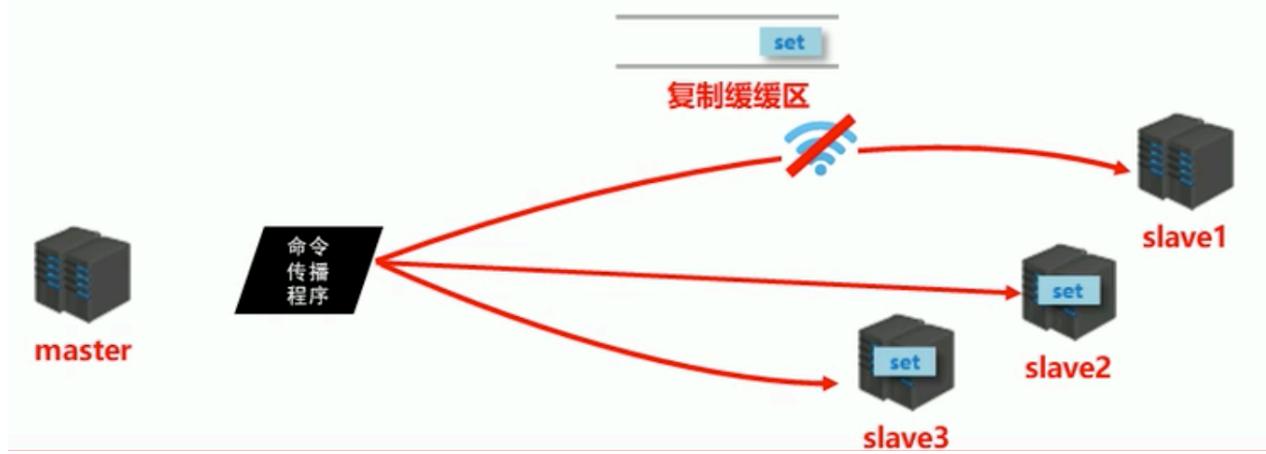
- 每台服务器启动时, 如果开启有AOF或被连接成为master节点, 即创建复制缓冲区

作用:

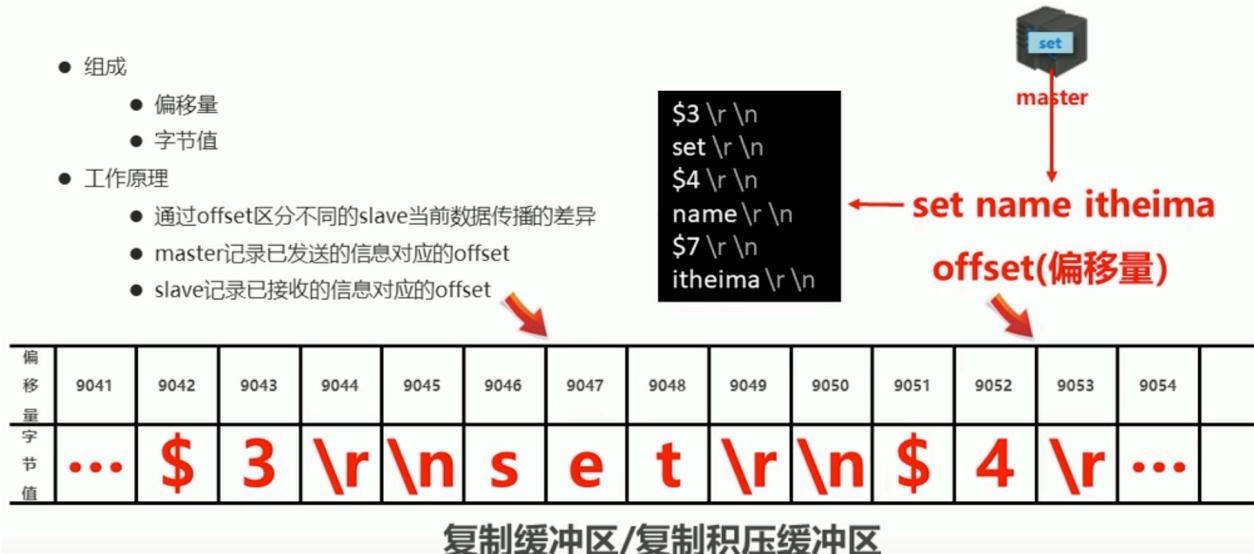
- 用于保存master收到的所有指令(仅影响数据变更的指令, 例如set, select)

数据来源:

- 当master接收到主客户端的指令时, 除了将指令执行, 会将该指令存储到缓冲区中

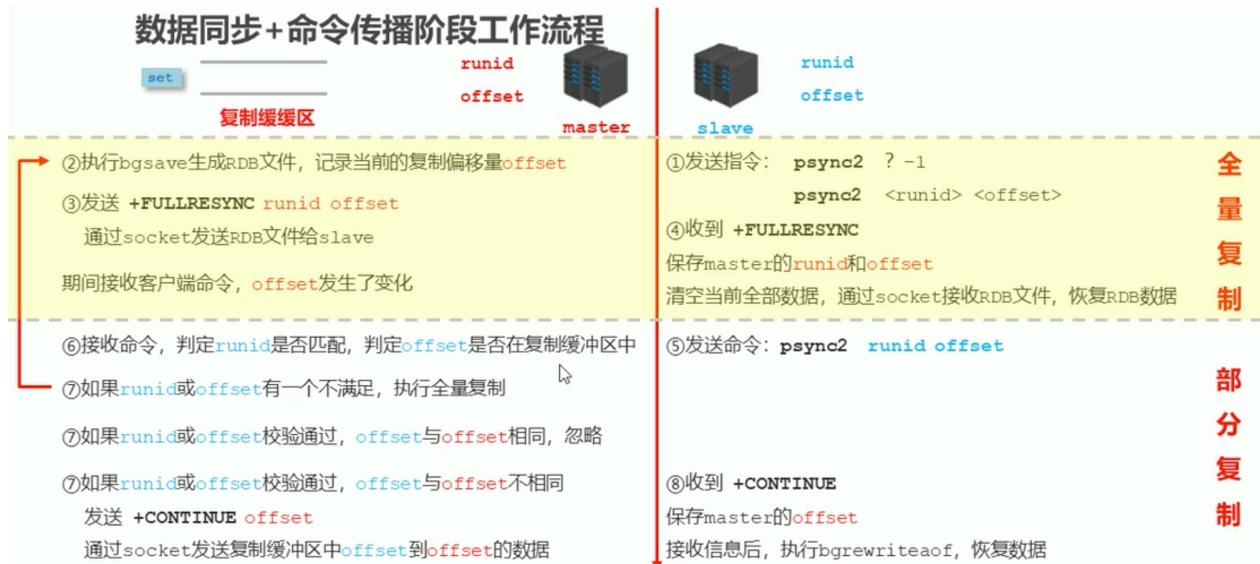


复制缓冲区内部工作原理



10.5.4 主从服务器复制偏移量(offset)

- 概念: 一个数字, 描述复制缓冲区中的指令字节位置
- 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-次
 - slave端:接收-次记录- -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



10.5.5 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, 实现双方连接保持在线

master 心跳:

- 指令: PING
- 周期: 由 `repl-ping-slave-period` 决定, 默认 10 秒
- 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 心跳任务

- 指令: `REPLCONF ACK {offset}`
- 周期: 1 秒
- 作用 1: 汇报 slave 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

心跳阶段注意事项

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


```
min_slaves_to_write 2
min_slaves_max_lag 8
```

 slave 数量少于 2 个, 或者所有 slave 的延迟都大于等于 10 秒时, 强制关闭 master 写功能, 停止数据同步
- slave 数量由 slave 发送 `REPLCONF ACK` 命令做确认
- slave 延迟由 slave 发送 `R EPLCONF ACK` 命令做确认



10.6 主从复制常见问题

10.6.1 频繁的全量复制(1)

伴随着系统的运行, `master` 的数据量会越来越大, 一旦 `master` 重启, `runid` 将发生变化, 会导致全部 `slave` 的全量复制操作

内部优化调整方案:

1. `master` 内部创建 `master_replid` 变量, 使用 `runid` 相同的策略生成, 长度 41 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` 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` => 可通过 `info` 命令可以查看该信息
4. 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`

10.6.2 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 => `repl-backlog-size`
- 建议设置如下:
 - 测算从 `master` 到 `slave` 的重连平均时长 `second`

- 获取master平均每秒产生写命令数据总量 `write_size_per_second`
- 最优复制缓冲区空间 = $2 * \text{second} * \text{write_size_per_second}$

10.6.3 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命令到master
 - 当slave接到了慢查询时(keys*, hgetall等), 会大量占用CPU性能
 - master每1秒调用复制定时函数`replicationCron()`, 比对slave发现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60秒), 超过该值, 释放slave

10.6.4 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 设定超时时间较短
 - ping 指令在网络中存在丢包
- 解决方案
 - 提高ping指令发送的频度 => `repl-ping-slaveperiod`
 - 超时时间 `repl-time` 的时间至少是ping指令频度的5到10倍, 否则 slave 很容易判定超时

10.6.5 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
 - 优化主从间的网络环境, 通常放置在同一个机房部署, 如使用阿里云等云服务器时要注意此现象
 - 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等少数命令(慎用, 除非对数据一致性要求很高)

十一、哨兵(主从切换)

11.1 哨兵简介

11.1.1 master宕机了怎么办?

1. 将宕机的master下线
2. 找一个slave作为master
3. 通知所有的slave连接新的master
4. 启动新的master与slave
5. 全量复制N +部分复制N

11.1.2 哨兵的引出

1. 谁来确认master宕机了
2. 找一个主?怎么找法?
3. 修改配置后，原始的主恢复了怎么办?

11.1.3 哨兵

概念：

哨兵(sentinel)是一个分布式系统，用于对主从结构中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

哨兵的工作 => 主从切换

哨兵在进行主从切换过程中经历三个阶段

- 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
- 通知(提醒)
 - 当被监控的服务器出现问题时，向其他(哨兵间，客户端)发送通知。
- 自动故障转移
 - 断开master与slave连接，选取一个slave作为master，将其他slave连接到新的master，并告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，通常哨兵配置数量为单数

11.2 哨兵结构搭建

学到的Linux命令

```

# 过滤带注释'#'的
cat sentinel.conf | grep -v "#" | grep -v "^\$"
# 过滤带注释'#'的 => 并将过滤后的结果复制为一份.conf文件
cat sentinel.conf | grep -v "#" | grep -v "^\$" > ./conf/sentinel-26379.conf
# 复制xxx.conf文件, 将文件中的内容xxx替换为xxx2
sed 's/26379/26381/g' sentinel-26379.conf > sentinel-26381.conf

```

11.2.1 配置哨兵配置文件

1. 将 `/redis.x.x.x` 目录下的 `sentinel.conf` 复制到 => `redis.x.x.x/conf/` (自己建立)

```

# 端口号, 哨兵通常前面加2
port 26379
daemonize no
pidfile "/var/run/redis-sentinel-26379.pid"
logfile "sentinel-26379.log"
dir "/usr/local/redis-5.0.7/data"
# 两台哨兵认为master挂了就真挂了
sentinel myid 383f810f632b52988eb774163ae7175e54937803
# 30s连接不上master则认为挂
sentinel deny-scripts-reconfig yes
#当master宕机时, 这里端口号将由master的端口号 => 新上位的master的端口号
sentinel monitor mymaster 127.0.0.1 6379 2
# slave 有3分钟时间充当master
sentinel config-epoch mymaster 1
sentinel leader-epoch mymaster 1
# Generated by CONFIG REWRITE
protected-mode no

```

2. 复制多份 `sentinel-端口号.conf`, 并修改

11.2.2 模拟master宕机 => slave上位 **

1. 启动master => slave => 哨兵

```
redis-sentinel sentinel-端口号.conf
```

2. 杀死master进程, 模拟master宕机

```
kill -9 pid
```

3. slave上位

4. 查看sentinel配置文件 => 端口号已修改

```
sentinel monitor mymaster 127.0.0.1 6380 2
```

5. 查看sentinel日志文件 => 查看slave上位流程

```

# 监控到master已经宕机
1948:X 13 Mar 2020 10:26:19.760 # +sdown master mymaster 127.0.0.1 6379
# 投票计数

```

```
1948:X 13 Mar 2020 10:26:19.866 # +new-epoch 1
# 投票选举 进程号为xxxx的上位
1948:X 13 Mar 2020 10:26:19.867 # +vote-for-leader
2d6333f04c42040d406d57546a9ea65e90d6bf7b1
1948:X 13 Mar 2020 10:26:20.216 # +config-update-from sentinel
2d6333f04c42040d406d57546a9ea65e90d6bf7b 127.0.0.1 26382 @ mymaster
127.0.0.1 6379
# 转换master => 6380为新上位的master
1948:X 13 Mar 2020 10:26:20.216 # +switch-master mymaster 127.0.0.1 6379
127.0.0.1 6380
# 其他slave重新从属新的master
1948:X 13 Mar 2020 10:26:20.217 * +slave slave 127.0.0.1:6381 127.0.0.1
6381 @ mymaster 127.0.0.1 6380
1948:X 13 Mar 2020 10:26:20.217 * +slave slave 127.0.0.1:6382 127.0.0.1
6382 @ mymaster 127.0.0.1 6380
# 6379 从属新的master
1948:X 13 Mar 2020 10:26:20.217 * +slave slave 127.0.0.1:6379 127.0.0.1
6379 @ mymaster 127.0.0.1 6380
# 6379 宕机了，无法slave
1948:X 13 Mar 2020 10:26:50.256 # +sdown slave 127.0.0.1:6379 127.0.0.1
6379 @ mymaster 127.0.0.1 6380
```

6. 重新启动6379 => 6379变成slave了

```
# Replication
role:slave
master_host:127.0.0.1
master_port:6380
```

7. 查看启动的.conf文件

```
# 连接的master，端口号、slaveof      自动修改了
replicaof 127.0.0.1 6380
```

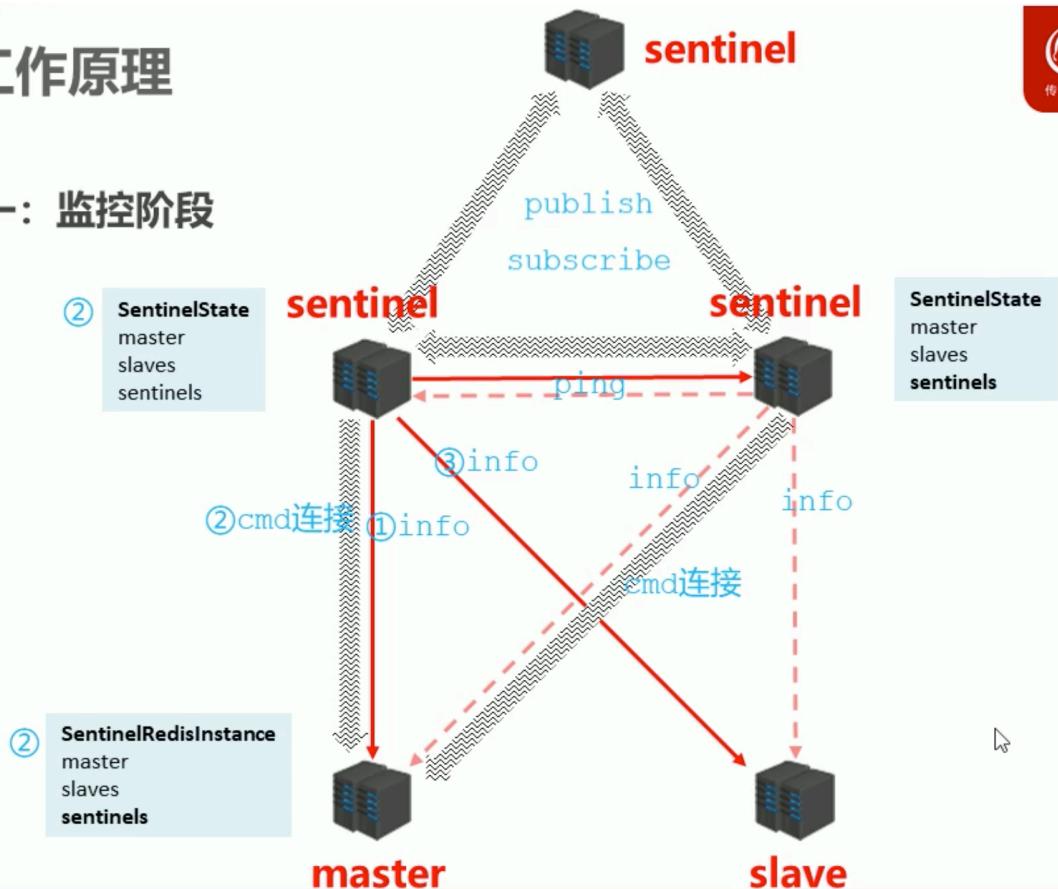
11.3 哨兵工作原理

11.3.1 监控阶段

哨兵工作原理

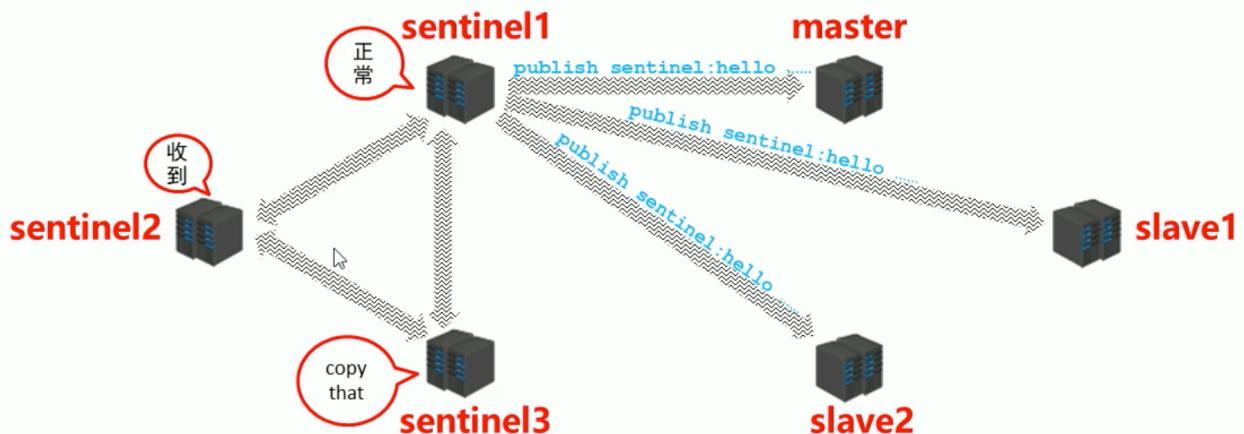


阶段一：监控阶段



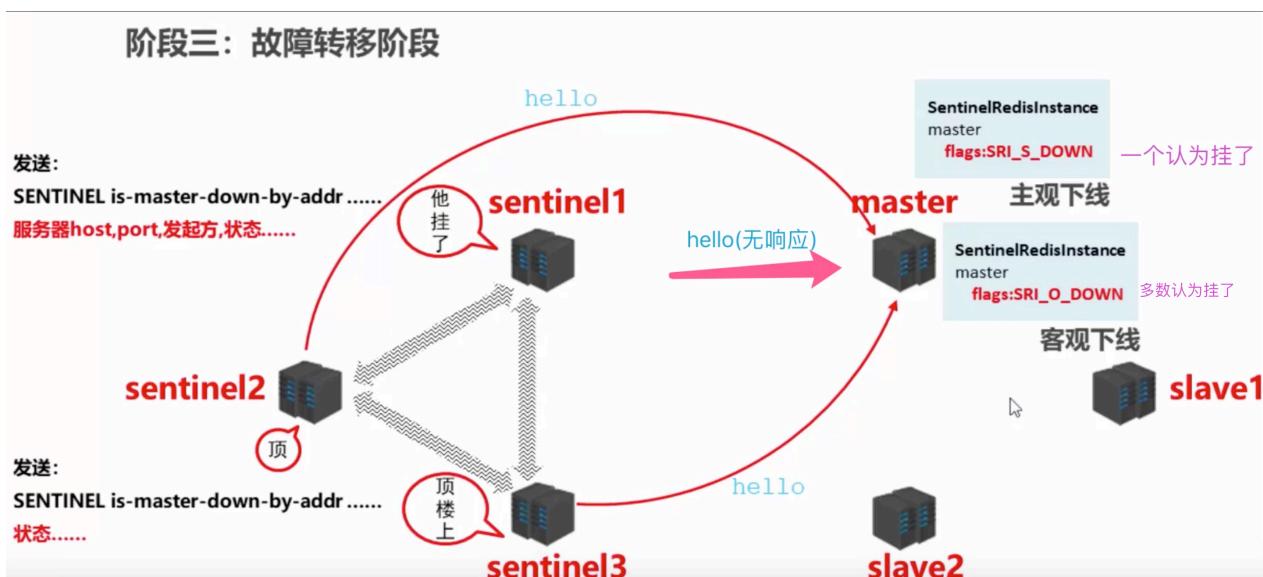
- 用于同步各个节点的状态信息
 - 获取各个sentinel的状态(是否在线)
 - 获取master的状态
 - master属性
 - runid
 - role: master
 - 各个slave的详细信息
 - 获取所有slave的状态(根据master中的slave信息)
 - slave属性
 - runid
 - role: slave
 - master host、master port
 - offset

11.3.2 通知阶段

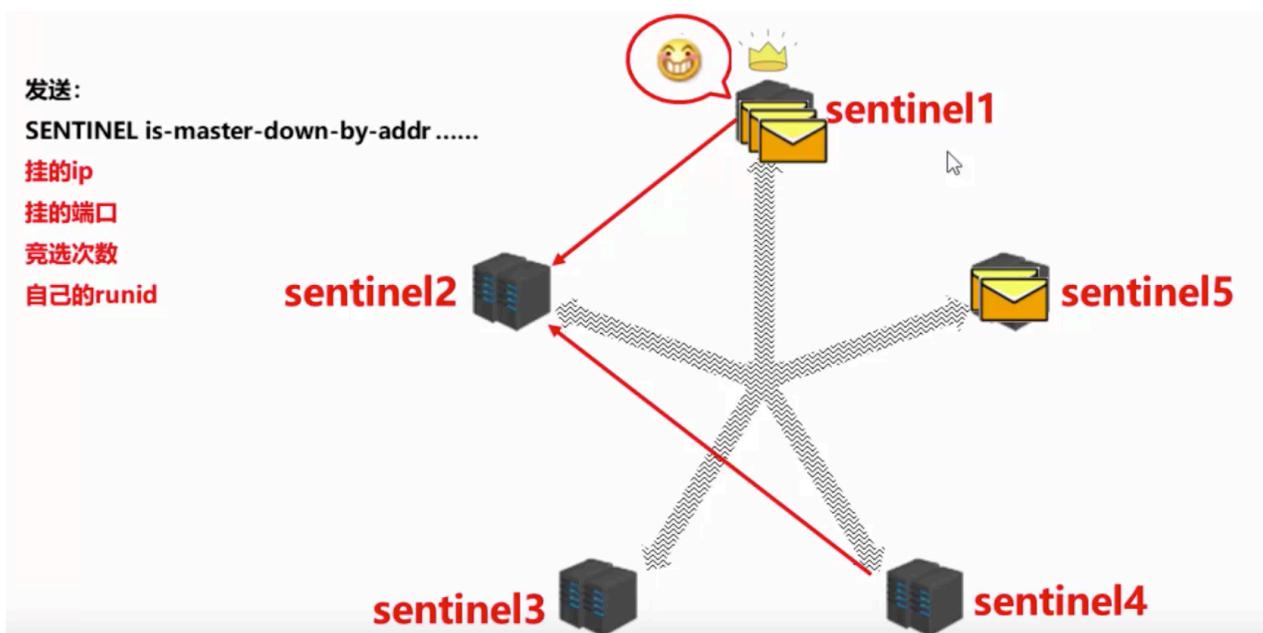


11.3.3 故障转移阶段

1. 大体原理：



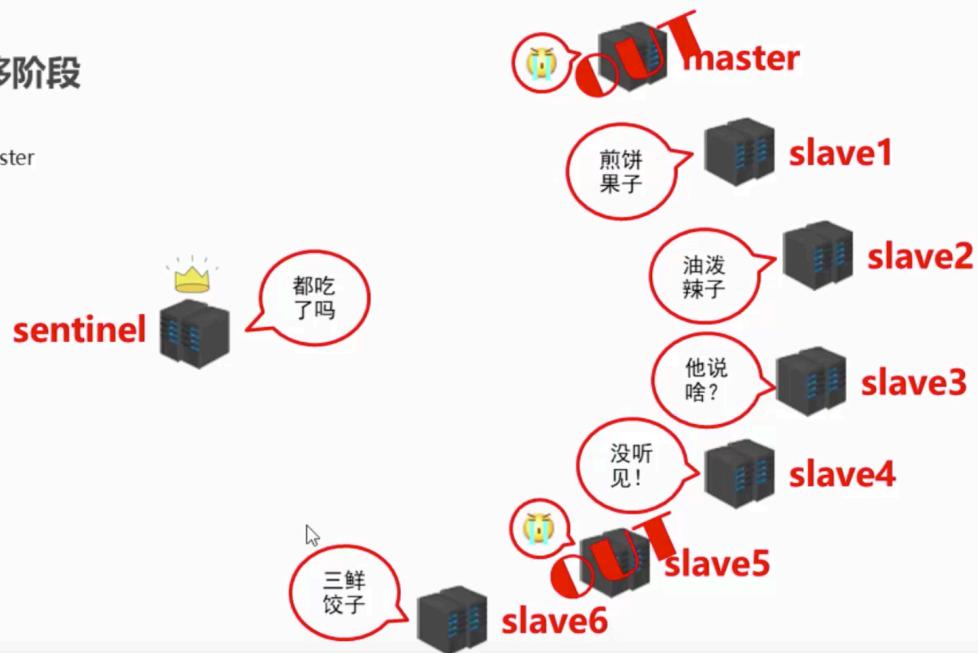
2. 挑选sentinel头头：



3. 挑选备选master：

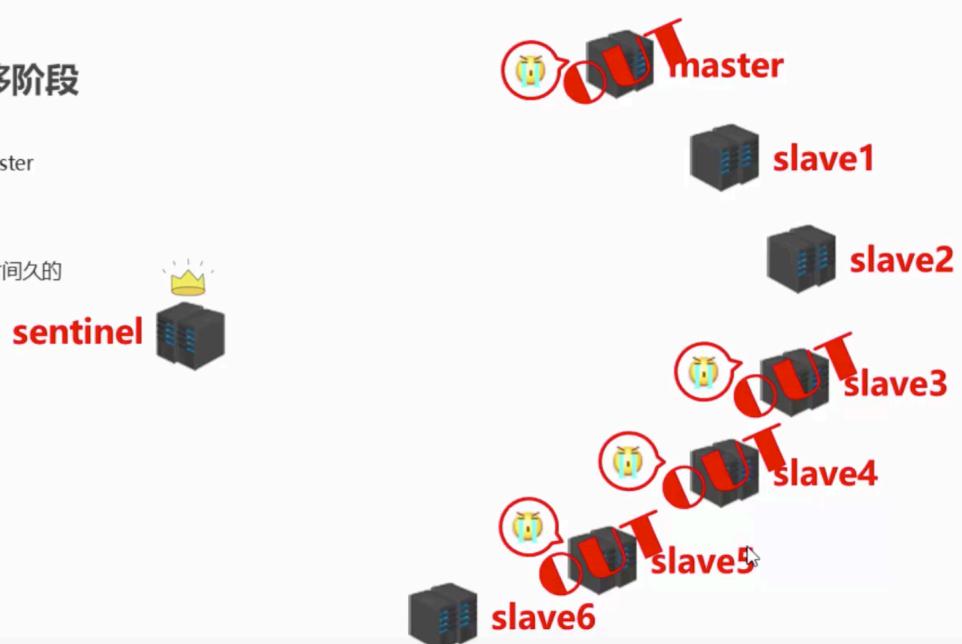
阶段三：故障转移阶段

- 服务器列表中挑选备选master
 - 在线的
 - 响应慢的



阶段三：故障转移阶段

- 服务器列表中挑选备选master
 - 在线的
 - 响应慢的
 - 与原master断开时间久的



阶段三：故障转移阶段

- 服务器列表中挑选备选master

- 在线的
- 响应慢的
- 与原master断开时间久的



阶段三：故障转移阶段

- 服务器列表中挑选备选master

- 在线的
- 响应慢的
- 与原master断开时间久的
- 优先原则
 - 优先级
 - offset
 - runid

- 发送指令 (sentinel)

- 向新的master发送slaveof no one
- 向其他slave发送slaveof 新masterIP端口



大体流程

- 监控
 - 同步信息
- 通知
 - 保持联通
- 故障转移
 - 发现问题
 - 竞选负责人
 - 优选新master
 - 新master上任，其他slave切换master，原master作为slave故障回复后连接

十二、集群

12.1 集群简介

12.1.1 为什么需要集群

业务发展过程中遇到的峰值瓶颈

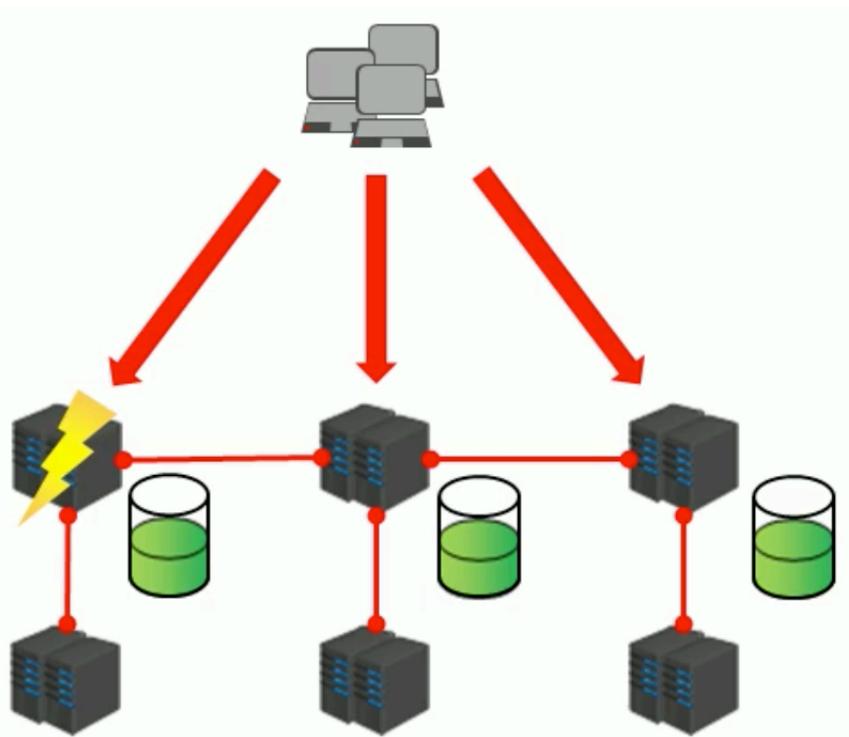
- redis提供的服务OPS可以达到10万/秒，当前业务OPS已经达到20万/秒
- 内存单机容量达到256G，当前业务需求内存容量1T
- 使用集群的方式可以快速解决上述问题

12.1.2 集群架构

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

12.1.3 集群作用

- 分散单台服务器的访问压力，实现负载均衡
- 分散单台服务器的存储压力，实现可扩展性
- 降低单台服务器宕机带来的业务灾难



12.2 搭建集群

<https://blog.csdn.net/gfk3009/article/details/102558445>

1. 在 `redis.x.x.x/cluster/` 配置 `xxx.conf`

```
# 其他同xxx.conf
# 单机测试一定要有绑定本地ip, 否则会出现 Waiting for the cluster to join 无限循环
bind 127.0.0.1
# 集群相关配置
cluster-enabled yes
cluster-config-file nodes-6379.conf
cluster-node-timeout 10000
```

2.输入命令 => 连接集群:

- Redis 5之前的命令:

```
o → src ./redis-trib.rb create --replica 1 127.0.0.1:6379 127.0.0.1:6380
127.0.0.1:6381 127.0.0.1:6382
```

```
WARNING: redis-trib.rb is not longer available!
You should use redis-cli instead.
```

```
All commands and features belonging to redis-trib.rb have been moved
to redis-cli.
```

```
In order to use them you should call redis-cli with the --cluster
option followed by the subcommand name, arguments and options.
```

```
Use the following syntax:
```

```
redis-cli --cluster SUBCOMMAND [ARGUMENTS] [OPTIONS]
```

Example:

```
redis-cli --cluster create 1 127.0.0.1:6379 127.0.0.1:6380 127.0.0.1:6381
127.0.0.1:6382
```

To get help about all subcommands, type:

```
redis-cli --cluster help
```

这里的1是指的 1个master连接一个slave

- Redis 5连接集群的命令()

```
o → ~ redis-cli --cluster create 127.0.0.1:6379 127.0.0.1:6380 127.0.0.1:6381
127.0.0.1:6382 127.0.0.1:6383 127.0.0.1:6384 --cluster-replicas 1
```

```
→ src redis-cli --cluster create 127.0.0.1:6379 127.0.0.1:6380
127.0.0.1:6381 127.0.0.1:6382 127.0.0.1:6383 127.0.0.1:6384 --cluster-
replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:6383 to 127.0.0.1:6379
Adding replica 127.0.0.1:6384 to 127.0.0.1:6380
Adding replica 127.0.0.1:6382 to 127.0.0.1:6381
```

```
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 0697a520a848c056f55a0a6f1d221e3894ea64ad 127.0.0.1:6379
    slots:[0-5460], [12706] (5461 slots) master
M: fa25857cccd5b869c93f4c5e726fed6fb9da28790 127.0.0.1:6380
    slots:[5461-10922] (5462 slots) master
M: 5b4d84abc7805f107bf375e8c3feaaacd9bd4c44 127.0.0.1:6381
    slots:[9189], [10923-16383] (5461 slots) master
S: b6ee9a0a70469bf89f651b62b2c67b0b167341c1 127.0.0.1:6382
    replicates 0697a520a848c056f55a0a6f1d221e3894ea64ad
S: 09766ffca233d7d5733f74883a44a00af0498568 127.0.0.1:6383
    replicates fa25857cccd5b869c93f4c5e726fed6fb9da28790
S: 2fc840164524b309caaa05fff302b8a11b09e7a7 127.0.0.1:6384
    replicates 5b4d84abc7805f107bf375e8c3feaaacd9bd4c44
Can I set the above configuration? (type 'yes' to accept):
```

- 出现的bug

```
# 连接至少6个节点 => 否则报错
*** ERROR: Invalid configuration for cluster creation.
*** Redis Cluster requires at least 3 master nodes.
*** This is not possible with 4 nodes and 1 replicas per node.
*** At least 6 nodes are required.
```

```
# .conf配置文件中不可以有slave从属信息，否则报错
*** FATAL CONFIG FILE ERROR ***
Reading the configuration file, at line 9
>>> 'slaveof 127.0.0.1 6379'
>>> replicaof directive not allowed in cluster mode
```

```
# 未连接的cluster的日志文件 node.xxx.conf
0697a520a848c056f55a0a6f1d221e3894ea64ad :6379@16379 myself, master - 0 0 1
connected 0-5460 12706
vars currentEpoch 1 lastVoteEpoch 0
```

```
# 集群节点不能为单数，否则出现如下错误
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Failed to send CLUSTER MEET command.
```

```

# 集群中key不为空、log文件、rdb、aof等文件存在时报错
[ERR] Node is not empty. Either the node already knows other nodes ...
解决方法：
1)、将需要新增的节点下aof、rdb等本地备份文件删除；
2)、同时将 newNode 的集群配置文件删除，即：删除你 redis.conf 里面 cluster-config-file 所在的文件；
3)、再次添加新节点如果还是报错，则登录 newNode， ./redis-cli -h x -p 对数据库进行清除：
172.168.63.201:7001> flushdb      #清空当前数据库

```

- 连接成功的完整信息

```

→ ~ redis-cli --cluster create 127.0.0.1:6379 127.0.0.1:6380 127.0.0.1:6381
127.0.0.1:6382 127.0.0.1:6383 127.0.0.1:6384 --cluster-replicas 1
>>> Performing hash slots allocation on 6 nodes...
Master[0] -> Slots 0 - 5460
Master[1] -> Slots 5461 - 10922
Master[2] -> Slots 10923 - 16383
Adding replica 127.0.0.1:6383 to 127.0.0.1:6379
Adding replica 127.0.0.1:6384 to 127.0.0.1:6380
Adding replica 127.0.0.1:6382 to 127.0.0.1:6381
>>> Trying to optimize slaves allocation for anti-affinity
[WARNING] Some slaves are in the same host as their master
M: 35f07e4051c91bc154b26290b7c7bad9e66cedfd 127.0.0.1:6379
    slots:[0-5460] (5461 slots) master
M: 2c5e5c790a790bfe85b0b7736618f3bd0533c59d 127.0.0.1:6380
    slots:[5461-10922] (5462 slots) master
M: 89526b8663925d27471456b9efd73c7d4865dd7f 127.0.0.1:6381
    slots:[10923-16383] (5461 slots) master
S: fae0b8421f41dfa50d4a0b1ce6f42a1121f56c1a 127.0.0.1:6382
    replicates 89526b8663925d27471456b9efd73c7d4865dd7f
S: cc74469ff209e4161971b908ef5a8474e5947e6c 127.0.0.1:6383
    replicates 35f07e4051c91bc154b26290b7c7bad9e66cedfd
S: e2d3b83bd0d309a1ba745bab82ce2846990ae1dd 127.0.0.1:6384
    replicates 2c5e5c790a790bfe85b0b7736618f3bd0533c59d
Can I set the above configuration? (type 'yes' to accept): yes
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
....
>>> Performing Cluster Check (using node 127.0.0.1:6379)
M: 35f07e4051c91bc154b26290b7c7bad9e66cedfd 127.0.0.1:6379
    slots:[0-5460] (5461 slots) master
    1 additional replica(s)
M: 89526b8663925d27471456b9efd73c7d4865dd7f 127.0.0.1:6381
    slots:[10923-16383] (5461 slots) master
    1 additional replica(s)
S: cc74469ff209e4161971b908ef5a8474e5947e6c 127.0.0.1:6383

```

```

slots: (0 slots) slave
replicates 35f07e4051c91bc154b26290b7c7bad9e66cedfd
M: 2c5e5c790a790bfe85b0b7736618f3bd0533c59d 127.0.0.1:6380
    slots:[5461-10922] (5462 slots) master
    1 additional replica(s)
S: e2d3b83bd0d309a1ba745bab82ce2846990ae1dd 127.0.0.1:6384
    slots: (0 slots) slave
    replicates 2c5e5c790a790bfe85b0b7736618f3bd0533c59d
S: fae0b8421f41dfa50d4a0b1ce6f42a1121f56c1a 127.0.0.1:6382
    slots: (0 slots) slave
    replicates 89526b8663925d27471456b9efd73c7d4865dd7f
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.

```

- 观察node.xxx.conf => 发现集群连接信息

```

89526b8663925d27471456b9efd73c7d4865dd7f 127.0.0.1:6381@16381 master - 0
1584104495023 3 connected 10923-16383
cc74469ff209e4161971b908ef5a8474e5947e6c 127.0.0.1:6383@16383 slave
35f07e4051c91bc154b26290b7c7bad9e66cedfd 0 1584104498115 5 connected
2c5e5c790a790bfe85b0b7736618f3bd0533c59d 127.0.0.1:6380@16380 master - 0
1584104497081 2 connected 5461-10922
e2d3b83bd0d309a1ba745bab82ce2846990ae1dd 127.0.0.1:6384@16384 slave
2c5e5c790a790bfe85b0b7736618f3bd0533c59d 0 1584104496057 6 connected
fae0b8421f41dfa50d4a0b1ce6f42a1121f56c1a 127.0.0.1:6382@16382 slave
89526b8663925d27471456b9efd73c7d4865dd7f 0 1584104495000 4 connected
35f07e4051c91bc154b26290b7c7bad9e66cedfd 127.0.0.1:6379@16379 myself, master -
0 1584104496000 1 connected 0-5460
vars currentEpoch 6 lastVoteEpoch 0

```

- 数据相关操作

```
redis-cli -c -p xxxx
```

12.3 主从下线与主从切换

Course 106

- 1.从slave下线 => 主标记其小弟下线, slave小弟上线即恢复
- 2.主master下线 => 从slave尝试连接master => 过了超时时间, 翻身农奴自己做master
- 3.原来的主master上线 => 成为原来小弟的小弟了

12.4 相关配置及操作

12.4.1 cluster 配置

- 设置加入cluster, 成为其中的节点

- cluster -enabled yes|no
- 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>
- 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 <milli seconds>
- master连接的slave最小数量
cluster -migration . -barrier < count>

12.4.2 Cluster节点的操作命令

- 查看集群节点信息
cluster nodes
- 进入一个从节点redis，切换其主节点
cluster replicate <master-id>
- 发现一个新节点，新增主节点
cluster meet ip :port
- 忽略一个没有solt的节点
cluster forget <id>
- 手动故障转移
cluster failover

企业级解决方案

十三、企业级解决方案

13.1 缓存预热

13.1.1 问题排查

- 请求数量较高
- 主从之间数据吞吐量较大，数据同步操作频度较高

13.1.2 解决方案

- 前置准备工作:
 - 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
 - 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:
 - 将统计结果中的数据分类，根据级别，redis优先加载级别较高的热点数据
 - 利用分布式多服务器同时进行数据读取，提速数据加载过程
- 实施:
 - 使用脚本程序固定触发数据预热过程
 - 如果条件允许，使用了CDN（内容分发网络），效果会更好

13.1.3 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！

13.2 缓存雪崩

13.2.1 数据库服务器崩溃(1)

- 系统平稳运行过程中，忽然数据库连接量激增
- 应用服务器无法及时处理请求
- 大量408, 500错误页面出现
- 客户反复刷新页面获取数据
- 数据库崩溃
- 应用服务器崩溃
- 重启应用服务器无效
- Redis服务器崩溃
- Redis集群崩溃
- 重启数据库后再次被瞬间流量放倒

13.2.2 问题排查

- 在一个较短的时间内，缓存中较多的key集中过期
- 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
- 数据库同时接收到大量的请求无法及时处理
- Redis大量请求被积压，开始出现超时现象
- 数据库流量激增，数据库崩溃
- 重启后仍然面对缓存中无数据可用
- Redis服务器资源被严重占用，Redis服务器崩溃
- Redis集群呈现崩塌，集群瓦解
- 应用服务器无法及时得到数据响应请求，来自客户端的请求数量越来越多，应用服务器崩溃
- 应用服务器，redis，数据库全部重启，效果不理想

13.2.3 问题分析

短时间范围内 => 大量key集中过期

13.2.4 解决方案

宏观

- 更多的页面静态化处理
- 构建多级缓存架构
 - Nginx缓存+redis缓存+ ehcache缓存
- 检测Mysql|严重耗时业务 进行优化
 - 对数据库的瓶颈排查:例如超时查询、耗时较高事务等
- 灾难预警机制
- 监控redis服务器性能指标

- CPU占用、CPU使用率
- 内存容量
- 查询平均响应时间
- 线程数
- 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

微观

- LRU与LFU切换
- 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90分钟，B类80分钟，C类70分钟
 - 过期时间使用固定时间+随机值的形式，稀释集中到期的key的数量
 - 超热数据使用永久key
- 定期维护(自动+人工)
 - 对即将过期数据做访问量分析，确认是否延时，配合访问量统计，做热点数据的延时
- 加锁(慎用!)

13.2.5 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(约40%)，配合其他策略一起使用，并监控服务器的运行数据，根据运行记录做快速调整。

13.3. 缓存击穿

类似于缓存雪崩，缓存雪崩是指的某些key，缓存击穿是指的某个key

13.3.1 问题排查

- Redis中某个key过期，该key访问巨大
- 多个数据请求从服务器直接压到Redis后，均未命中
- Redis在短时间内发起了大量对数据库中同一数据的访问

13.3.2 问题分析

- 单个key高热数据
- key过期

13.3.3 解决方案(术)

- 预先设定
 - 以电商为例，每个商家根据店铺等级，指定若干款主打商品，在购物节期间，加大此类信息key的过期时长
 - 注意：购物节不仅仅指当天，后续若干天，访问峰值呈现逐渐降低的趋势
- 现场调整
 - 监控访问量，对自然流量激增的数据延长过期时间或设置为永久性key

- 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，确保不秩
- 二级缓存
 - 设置不同的失效时间，保障不会被同时淘汰就行
- 加锁
 - 分布式锁，防止被击穿，但是要注意也是性能瓶颈，慎重！

13.3.4 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-数据的数据访问，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即时调整策略，毕竟单个key的过期监控难度较高，配合雪崩处理策略即可。

13.4 缓存穿透

13.4.1 数据库服务器崩溃(3)

- 系统平稳运行过程中，应用服务器流量随时间增量较大
- Redis服务器命中率随时间逐步降低
- Redis内存平稳，内存无压力
- Redis服务器CPU占用激增
- 数据库服务器压力激增，数据库崩溃

13.4.1 问题排查

- Redis中大面积出现未命中
- 出现非正常URL访问

13.4.2 问题分析

- 获取的数据在数据库中也不存在，数据库查询未得到对应数据
- Redis获取到null数据未进行持久化，直接返回
- 下次此类数据到达重复 上述过程
- 出现黑客攻击服务器

13.4.3 解决方案(术)

- 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30-60秒，最高5分钟
- 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)
- 实施监控
 - 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数据的占比

- 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10-50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)
- 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, 混淆到页面数据id中, 发现访问key不满足规则, 驳回数据访问

13.4.4 总结

- 缓存击穿访问了不存在的数据, 跳过了合法数据的redis数据缓存阶段, 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。

13.5 性能指标监控

13.5.1 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

13.5.2 指标描述

Name	Description
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), 命中越低, 压力越大
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的数量
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)

13.5.3 监控方式

工具

- Cloud Insight Redis
- Prometheus
- Redis- stat
- Redis- faina

- RedisLive
- zabbix

命令

- benchmark
- redis-cli
- monitor
- showlog

benchmark(压力测试工具)

命令: `redis-benchmark [-h] [-p] [-c] [-n <requests>] [-k]`

范例1: `redis -benchmark ...`

说明: 50个连接, 10000次请求对应的性能

范例2: `redis -benchmark -c 100 -n 5000`

说明: 100个连接, 5000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线, 单位:微妙`

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