第二章：Redis高级

学习目标

目标1：能够说出redis中的数据删除策与略淘汰策略

目标2：能够说出主从复制的概念，工作流程以及场景问题及解决方案

目标3：能够说出哨兵的作用以及工作原理，以及如何启用哨兵

目标4：能够说出集群的架构设计，完成集群的搭建

目标5：能够说出缓存预热，雪崩，击穿，穿透的概念，能说出redis的相关监控指标

1.数据删除与淘汰策略

1.1 过期数据

1.1.1 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

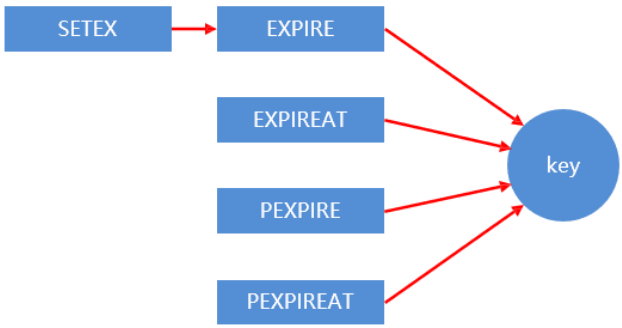
TTL返回的值有三种情况：正数，-1，-2

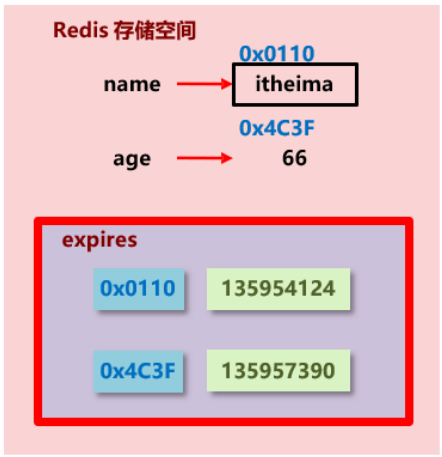
- **正数**：代表该数据在内存中还能存活的时间
- **-1**：永久有效的数据
- **2**：已经过期的数据 或被删除的数据 或 未定义的数据

删除策略就是针对已过期数据的处理策略，已过期的数据是真的就立即删除了吗？其实也不是，我们会有多种删除策略，是分情况的，在不同的场景下使用不同的删除方式会有不同效果，这也正是我们要将的数据的删除策略的问题

1.1.2 时效性数据的存储结构

在Redis中，如何给数据设置它的失效周期呢？数据的时效在redis中如何存储呢？看下图：





过期数据是一块独立的存储空间，Hash结构，field是内存地址，value是过期时间，保存了所有key的过期描述，在最终进行过期处理的时候，对该空间的数据进行检测， 当时间到期之后通过field找到内存该地址处的数据，然后进行相关操作。

1.2 数据删除策略

1.2.1 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

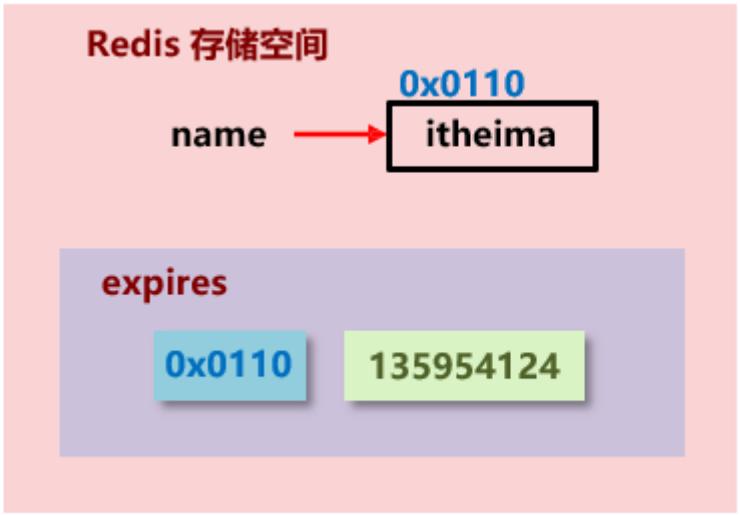
针对过期数据要进行删除的时候都有哪些删除策略呢?

- 1.定时删除
- 2.惰性删除
- 3.定期删除

1.2.2 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**：用处理器性能换取存储空间（拿时间换空间）

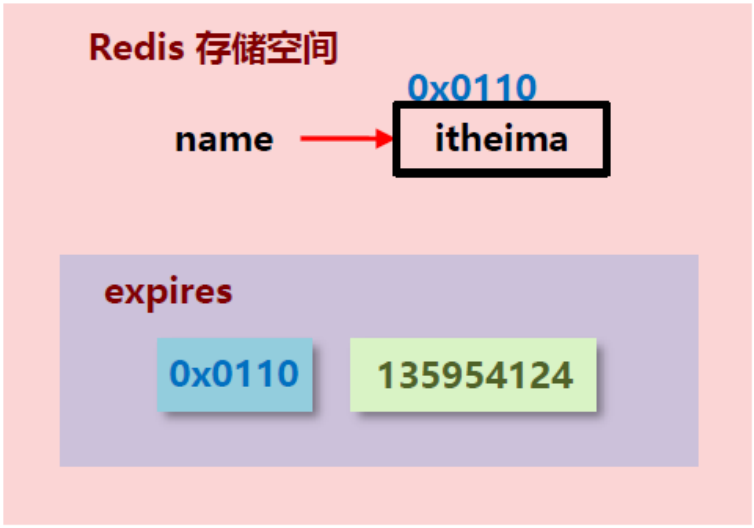


1.2.3 惰性删除

数据到达过期时间，不做处理。等下次访问该数据时，我们需要判断

1. 如果未过期，返回数据
2. 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）



1.2.4 定期删除

定时删除和惰性删除这两种方案都是走的极端，那有没有折中方案?

我们来讲redis的定期删除方案:

- 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，每次执行耗时：250ms/server.hz
- 对某个expires[*]检测时，随机挑选W个key检测

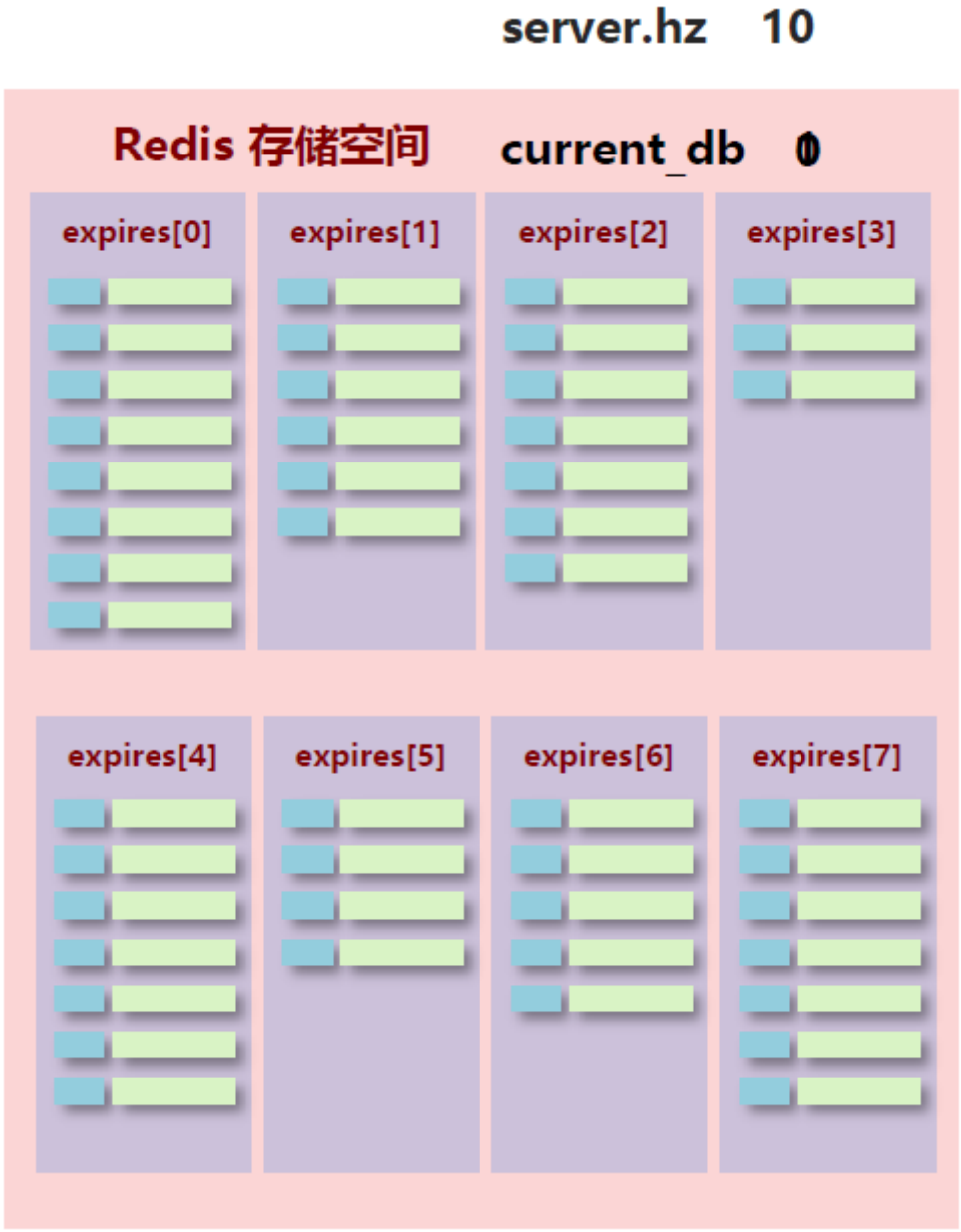
```
如果key超时，删除key

如果一轮中删除的key的数量>W*25%，循环该过程

如果一轮中删除的key的数量≤W*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



总的来说：定期删除就是周期性轮询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）

1.2.5 删除策略对比

1：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
```

2：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
```

3：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
```

1.3 数据淘汰策略（逐出算法）

1.3.1 淘汰策略概述

什么叫数据淘汰策略？什么样的应用场景需要用到数据淘汰策略？

当新数据进入redis时，如果内存不足怎么办？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， 如不能达到内存清理的要求，将出现错误信息如下

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

1.3.2 策略配置

影响数据淘汰的相关配置如下：

1：最大可使用内存，即占用物理内存的比例，默认值为0，表示不限制。生产环境中根据需求设定，通常设置在50%以上

```
maxmemory ?mb
```

2：每次选取待删除数据的个数，采用随机获取数据的方式作为待检测删除数据

```
maxmemory-samples count
```

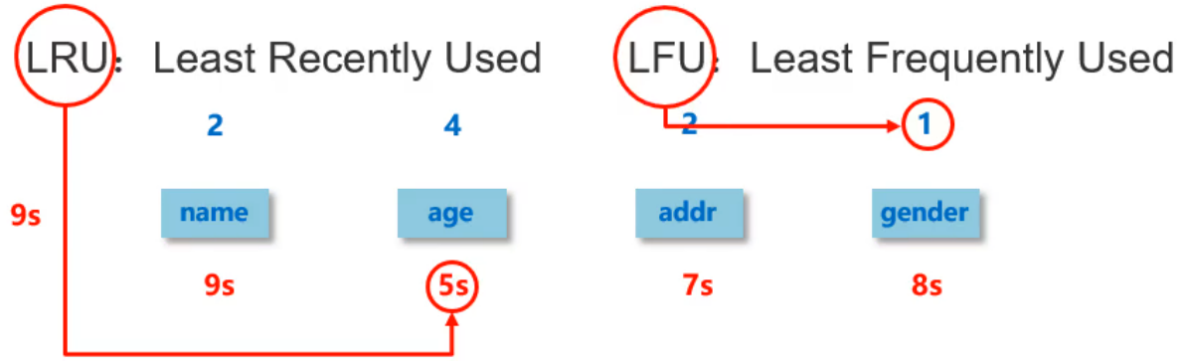
3：对数据进行删除的选择策略

```
maxmemory-policy policy
```

那数据删除的策略policy到底有几种呢？一共是**3类8种**

第一类：检测易失数据（可能会过期的数据集server.db[i].expires ）

```
volatile-lru：挑选最近最少使用的数据淘汰
volatile-lfu：挑选最近使用次数最少的数据淘汰
volatile-ttl：挑选将要过期的数据淘汰
volatile-random：任意选择数据淘汰
```



第二类：检测全库数据（所有数据集server.db[i].dict ）

```
allkeys-lru：挑选最近最少使用的数据淘汰
allkeLyRs-lfu：：挑选最近使用次数最少的数据淘汰
allkeys-random：任意选择数据淘汰，相当于随机
```

第三类：放弃数据驱逐

```
no-enviction（驱逐）：禁止驱逐数据(redis4.0中默认策略)，会引发OOM(Out Of Memory)
```

注意：这些策略是配置到哪个属性上？怎么配置？如下所示

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

数据淘汰策略配置依据

使用INFO命令输出监控信息，查询缓存 hit 和 miss 的次数，根据业务需求调优Redis配置

2.主从复制

2.1 主从复制简介

2.1.1 高可用

首先我们要理解互联网应用因为其独有的特性我们演化出的**三高**架构

- 高并发

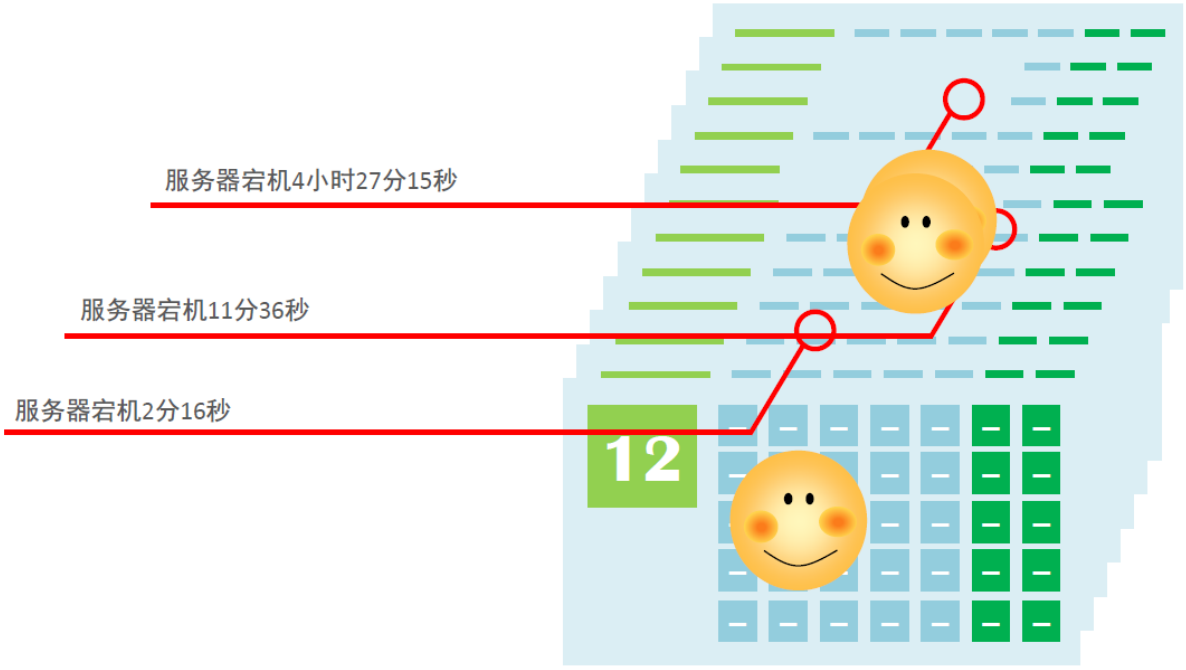
  > 应用要提供某一业务要能支持很多客户端同时访问的能力，我们称为并发，高并发意思就很明确了

- 高性能

  > 性能带给我们最直观的感受就是：速度快，时间短

- 高可用

可用性：一年中应用服务正常运行的时间占全年时间的百分比，如下图：表示了应用服务在全年宕机的时间



我们把这些时间加在一起就是全年应用服务不可用的时间，然后我们可以得到应用服务全年可用的时间

> 4小时27分15秒+11分36秒+2分16秒=4小时41分7秒=16867秒
>
> 1年=3652460*60=31536000秒
>
> 可用性=（31536000-16867）/31536000*100%=99.9465151%

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

2.1.2 主从复制概念

知道了三高的概念之后，我们想：你的“Redis”是否高可用？那我们要来分析单机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.

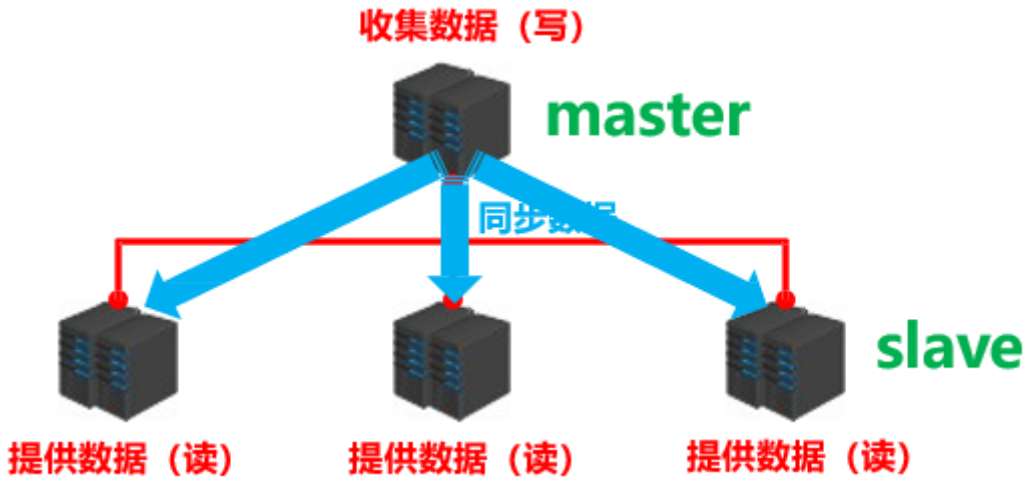
问题2.容量瓶颈

- 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。

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（master的数据复制到slave中）

这里我们可以来解释主从复制的概念：

概念：主从复制即将master中的数据即时、有效的复制到slave中

特征：一个master可以拥有多个slave，一个slave只对应一个master

职责：master和slave各自的职责不一样

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）
```

2.1.3 主从复制的作用

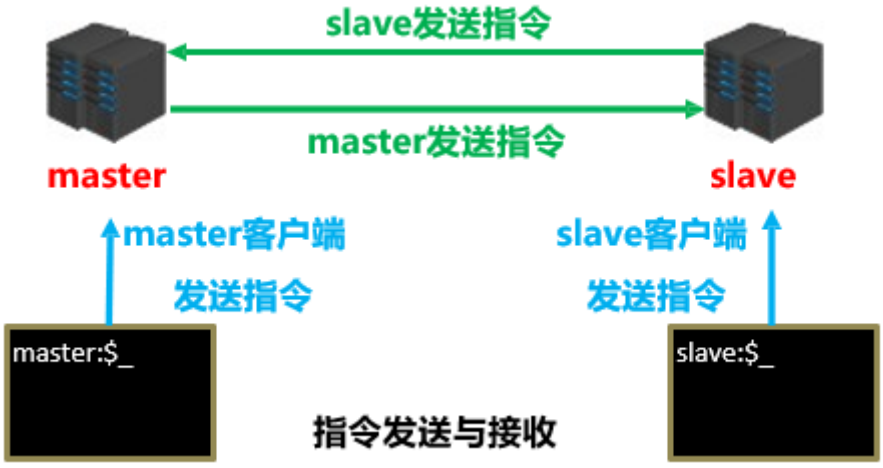
- 读写分离：master写、slave读，提高服务器的读写负载能力
- 负载均衡：基于主从结构，配合读写分离，由slave分担master负载，并根据需求的变化，改变slave的数 量，通过多个从节点分担数据读取负载，大大提高Redis服务器并发量与数据吞吐量
- 故障恢复：当master出现问题时，由slave提供服务，实现快速的故障恢复
- 数据冗余：实现数据热备份，是持久化之外的一种数据冗余方式
- 高可用基石：基于主从复制，构建哨兵模式与集群，实现Redis的高可用方案

2.2 主从复制工作流程

主从复制过程大体可以分为3个阶段

- 建立连接阶段（即准备阶段）
- 数据同步阶段
- 命令传播阶段（反复同步）

而命令的传播其实有4种，分别如下：



2.2.1 主从复制的工作流程（三个阶段）

2.2.1.1 阶段一：建立连接

建立slave到master的连接，使master能够识别slave，并保存slave端口号

流程如下：

1. 步骤1：设置master的地址和端口，保存master信息
2. 步骤2：建立socket连接
3. 步骤3：发送ping命令（定时器任务）
4. 步骤4：身份验证
5. 步骤5：发送slave端口信息

至此，主从连接成功！

当前状态：

slave：保存master的地址与端口

master：保存slave的端口

总体：之间创建了连接的socket

master和slave互联

接下来就要通过某种方式将master和slave连接到一起

方式一：客户端发送命令

```
slaveof masterip masterport
```

方式二：启动服务器参数

```
redis-server --slaveof masterip masterport
```

方式三：服务器配置（**主流方式**）

```
slaveof masterip masterport
```

slave系统信息

```
master_link_down_since_seconds
masterhost & masterport
```

master系统信息

```
uslave_listening_port(多个)
```

主从断开连接

断开slave与master的连接，slave断开连接后，不会删除已有数据，只是不再接受master发送的数据

```
slaveof no one
```

授权访问

master客户端发送命令设置密码

```
requirepass password
```

master配置文件设置密码

```
config set requirepass password
config get requirepass
```

slave客户端发送命令设置密码

```
auth password
```

slave配置文件设置密码

```
masterauth password
```

slave启动服务器设置密码

```
redis-server -a password
```

2.2.1.2 阶段二：数据同步

- 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

同步过程如下：

1. 步骤1：请求同步数据
2. 步骤2：创建RDB同步数据
3. 步骤3：恢复RDB同步数据
4. 步骤4：请求部分同步数据
5. 步骤5：恢复部分同步数据

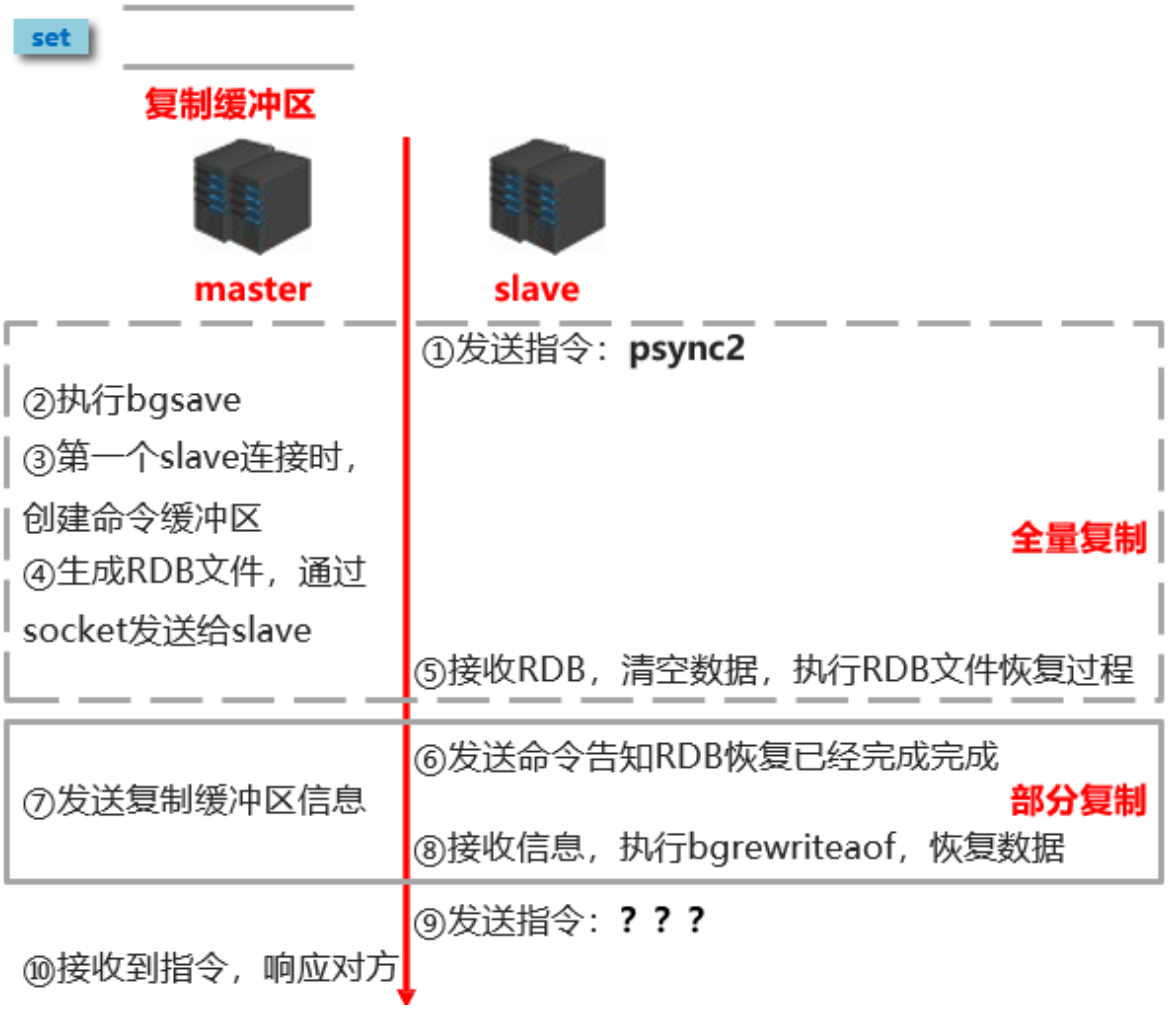
至此，数据同步工作完成！

当前状态：

slave：具有master端全部数据，包含RDB过程接收的数据

master：保存slave当前数据同步的位置

总体：之间完成了数据克隆



数据同步阶段master说明

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，影响业务正常执行

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 ?mb
```

3. 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

数据同步阶段slave说明

1. 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
```

2. 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
3. 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
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

2.2.1.3 阶段三：命令传播

- 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

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

这里我们主要来看部分复制，部分复制的三个核心要素

1. 服务器的运行 id（run id）
2. 主服务器的复制积压缓冲区
3. 主从服务器的复制偏移量

- 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
例如：fdc9ff13b9bbaab28db42b3d50f852bb5e3fcdce

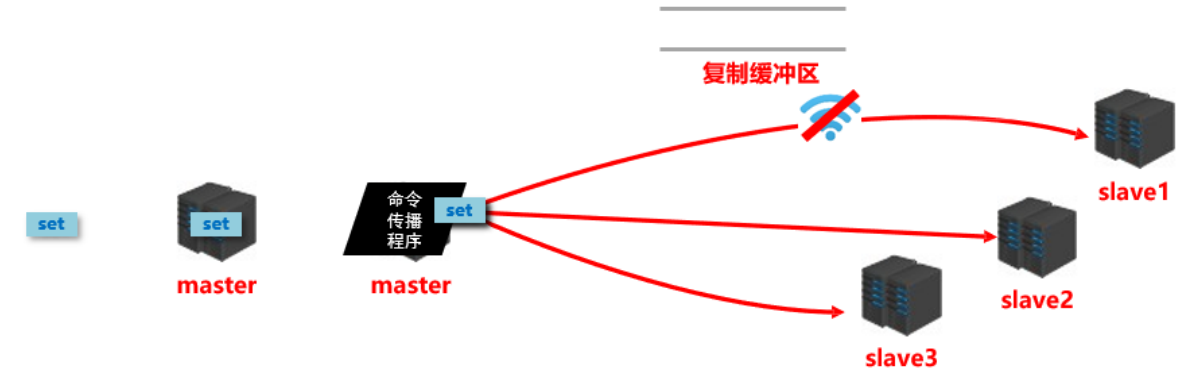
作用：运行id被用于在服务器间进行传输，识别身份
如果想两次操作均对同一台服务器进行，必须每次操作携带对应的运行id，用于对方识别

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

- 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
    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
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：

组成

- 偏移量

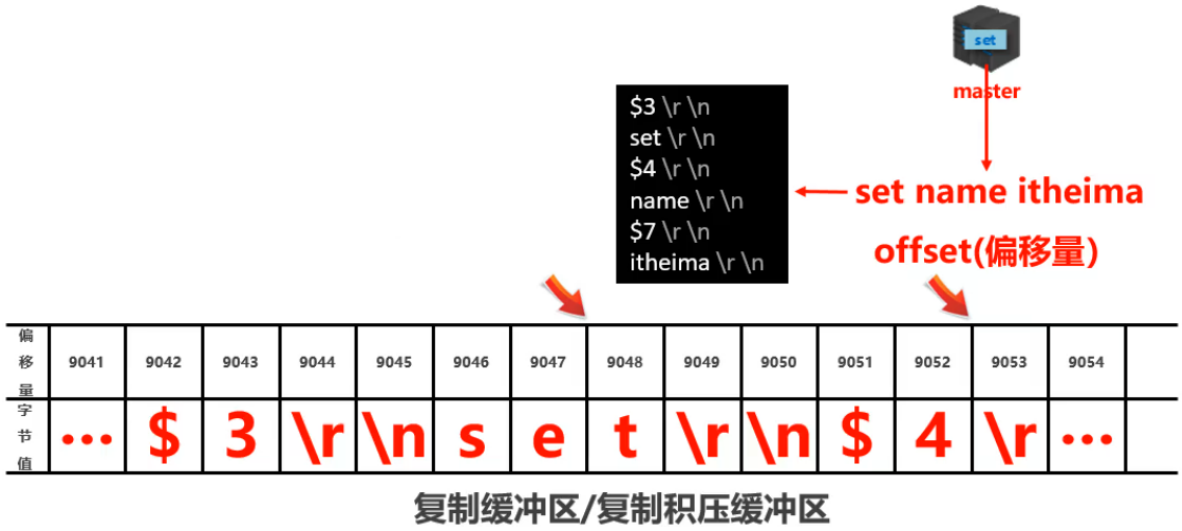
> 概念：一个数字，描述复制缓冲区中的指令字节位置
>
> 分类：
>
> - master复制偏移量：记录发送给所有slave的指令字节对应的位置（多个）
> - slave复制偏移量：记录slave接收master发送过来的指令字节对应的位置（一个）
>
> 作用：同步信息，比对master与slave的差异，当slave断线后，恢复数据使用
>
> 数据来源：
>
> - master端：发送一次记录一次
> - slave端：接收一次记录一次

- 字节值

工作原理

- 通过offset区分不同的slave当前数据传播的差异

- master记录已发送的信息对应的offset
- slave记录已接收的信息对应的offset



2.2.2 流程更新(全量复制/部分复制)

我们再次的总结一下主从复制的三个阶段的工作流程：

2.2.3 心跳机制

什么是心跳机制?

进入命令传播阶段候，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：

- 内部指令：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

- 内部指令：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：

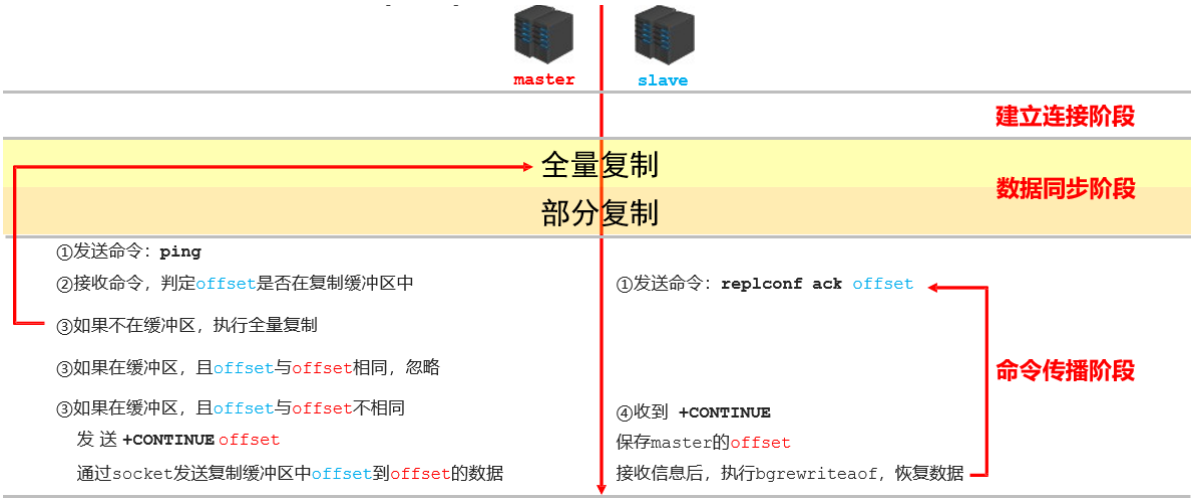
- 当slave多数掉线，或延迟过高时，master为保障数据稳定性，将拒绝所有信息同步

```
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8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REPLCONF ACK命令做确认

至此：我们可以总结出完整的主从复制流程：



2.3 主从复制常见问题

2.3.1 频繁的全量复制

- 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

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

- 第二种出现频繁全量复制的问题现象：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 ?mb
```

建议设置如下：

1.测算从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

2.3.2 频繁的网络中断

- 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REPLCONFACK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 seconds
```

该参数定义了超时时间的阈值（默认60秒），超过该值，释放slave

- 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-slave-period seconds
```

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

2.3.3 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
```

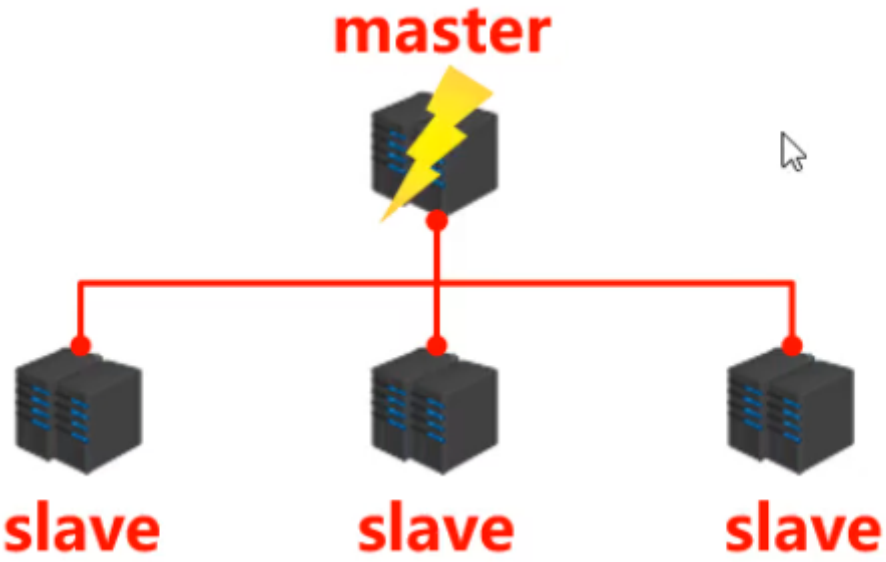
开启后仅响应info、slaveof等少数命令（慎用，除非对数据一致性要求很高）

3.哨兵模式

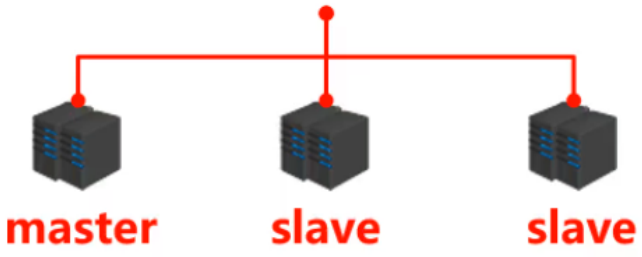
3.1 哨兵简介

3.1.1 哨兵概念

首先我们来看一个业务场景：如果redis的master宕机了，此时应该怎么办？



那此时我们可能需要从一堆的slave中重新选举出一个新的master，那这个操作过程是什么样的呢？这里面会有什么问题出现呢？



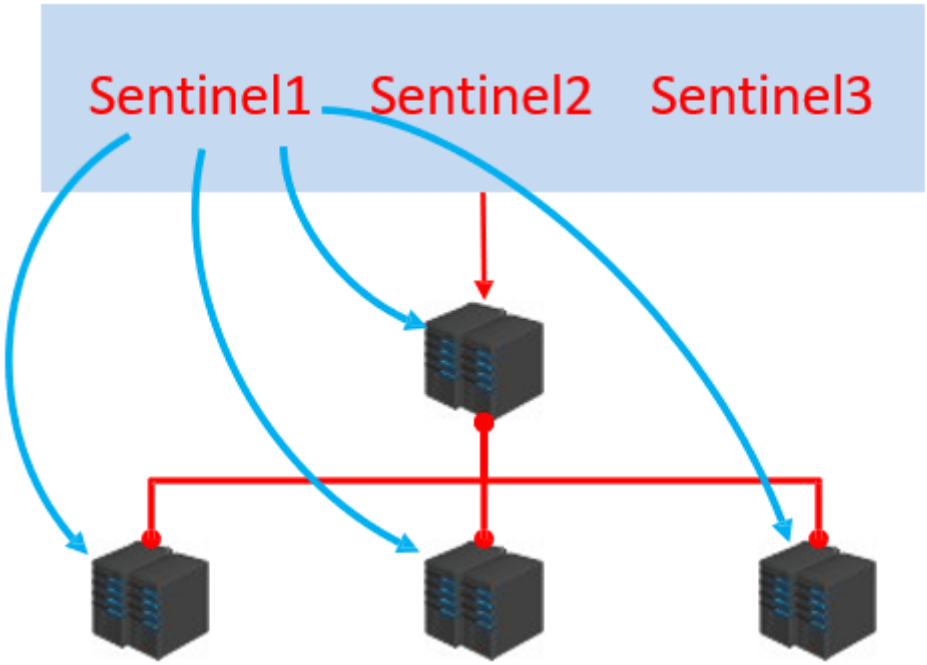
- 关闭master和所有slave
- 找一个slave作为master
- 修改其他slave的配置，连接新的主
- 启动新的master与slave
- 全量复制*N+部分复制*N

- 关闭期间的数据服务谁来承接?
- 找一个主？怎么找法?
- 修改配置后，原始的主恢复了怎么办?

要实现这些功能，我们就需要redis的哨兵，那哨兵是什么呢?

哨兵

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。



3.1.2 哨兵作用

哨兵的作用：

- 监控：监控master和slave

  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，只是不提供数据相关服务，通常哨兵的数量配置为单数

3.2 启用哨兵

配置哨兵

- 配置一拖二的主从结构（利用之前的方式启动即可）
- 配置三个哨兵（配置相同，端口不同），参看sentinel.conf

1：设置哨兵监听的主服务器信息， sentinel_number表示参与投票的哨兵数量

```
sentinel monitor master_name  master_host master_port  sentinel_number
```

2：设置判定服务器宕机时长，该设置控制是否进行主从切换

```
sentinel down-after-milliseconds master_name  million_seconds
```

3：设置故障切换的最大超时时

```
sentinel failover-timeout master_name million_seconds
```

4：设置主从切换后，同时进行数据同步的slave数量，数值越大，要求网络资源越高，数值越小，同步时间越长

```
sentinel parallel-syncs master_name sync_slave_number
```

- 启动哨兵

```
redis-sentinel filename
```

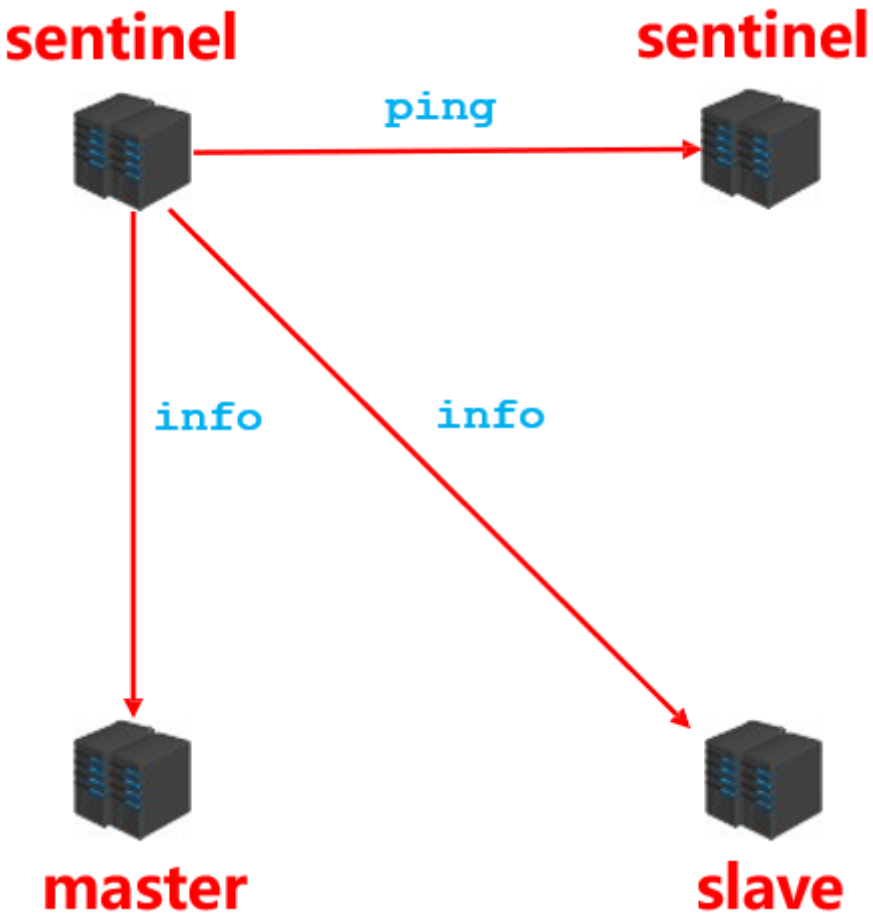
3.3 哨兵工作原理

哨兵在进行主从切换过程中经历三个阶段

- 监控
- 通知
- 故障转移

3.3.1 监控

用于同步各个节点的状态信息



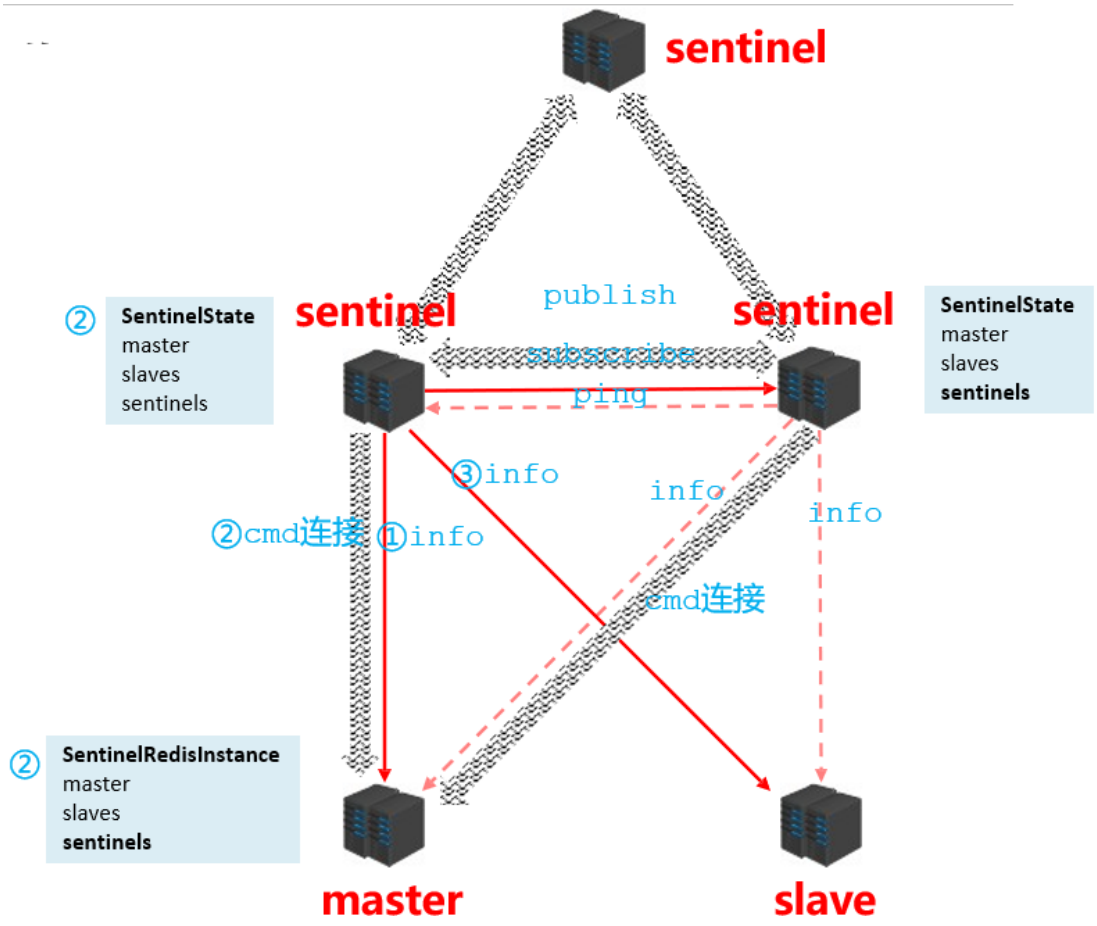
- 获取各个sentinel的状态（是否在线）
- 获取master的状态

```
master属性
  prunid
  prole：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）

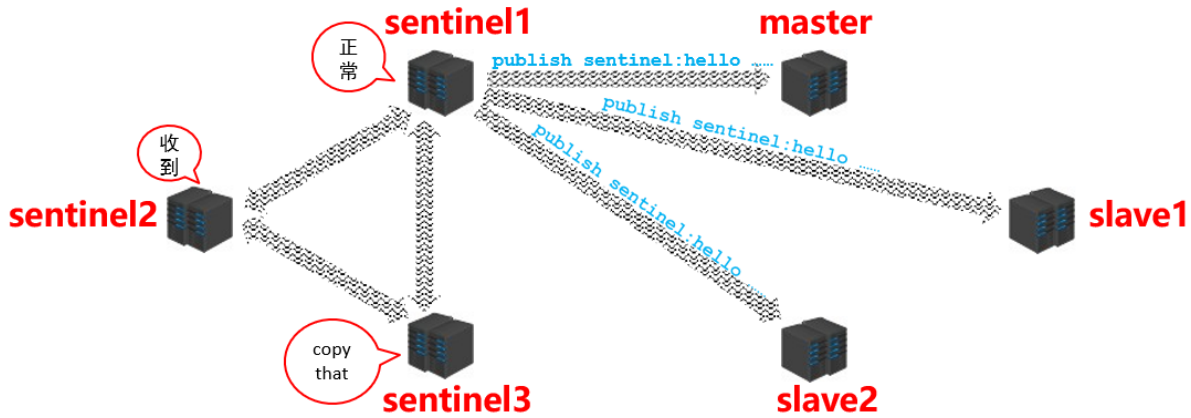
```
slave属性
  prunid
  prole：slave
  pmaster_host、master_port
  poffset
```

其内部的工作原理具体如下：



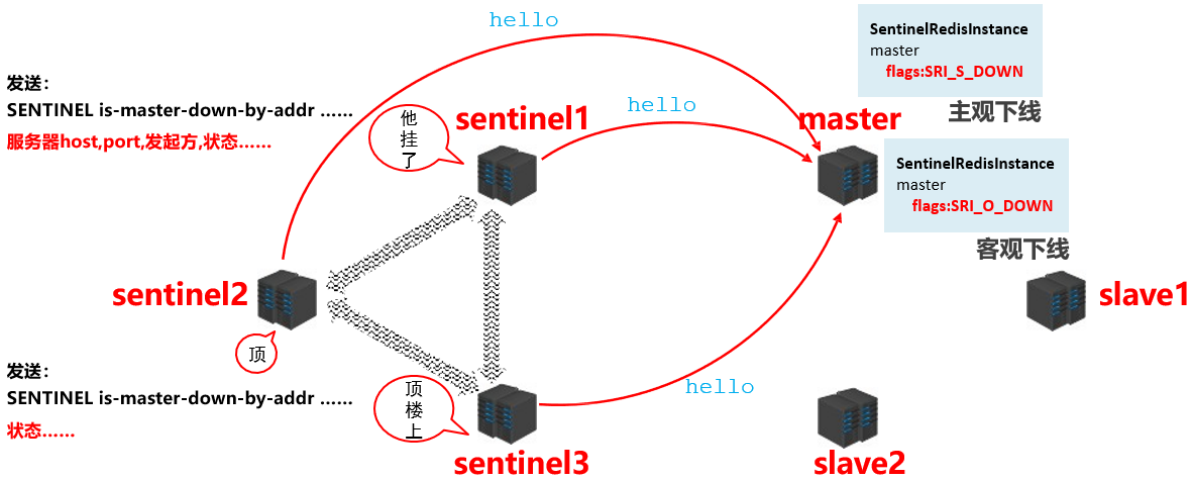
3.3.2 通知

sentinel在通知阶段要不断的去获取master/slave的信息，然后在各个sentinel之间进行共享，具体的流程如下：

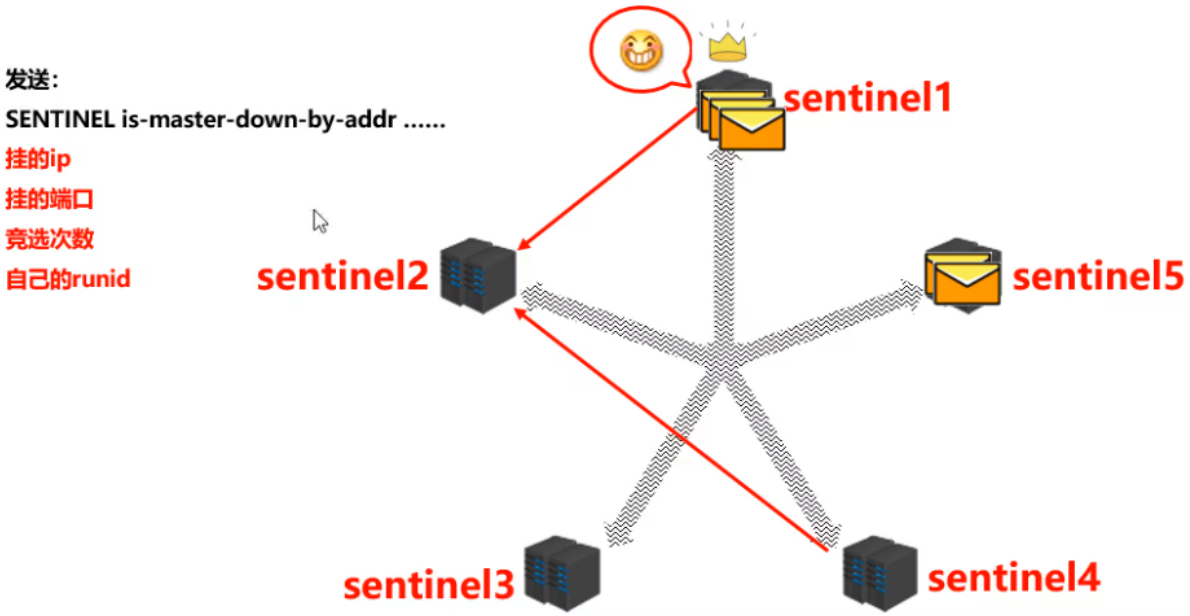


3.3.3 故障转移

当master宕机后sentinel是如何知晓并判断出master是真的宕机了呢？我们来看具体的操作流程



当sentinel认定master下线之后，此时需要决定更换master，那这件事由哪个sentinel来做呢？这时候sentinel之间要进行选举，如下图所示：



在选举的时候每一个人手里都有一票，而每一个人的又都想当这个处理事故的人，那怎么办？大家就开始抢，于是每个人都会发出一个指令，在内网里边告诉大家我要当选举人，比如说现在的sentinel1和sentinel4发出这个选举指令了，那么sentinel2既能接到sentinel1的也能接到sentinel4的，接到了他们的申请以后呢，sentinel2他就会把他的一票投给其中一方，投给谁呢？谁先过来我投给谁，假设sentinel1先过来，所以这个票就给到了sentinel1。那么给过去以后呢，现在sentinel1就拿到了一票，按照这样的一种形式，最终会有一个选举结果。对应的选举最终得票多的，那自然就成为了处理事故的人。需要注意在这个过程中有可能会存在失败的现象，就是一轮选举完没有选取，那就会接着进行第二轮第三轮直到完成选举。

接下来就是由选举胜出的sentinel去从slave中选一个新的master出来的工作，这个流程是什么样的呢？

首先它有一个在服务器列表中挑选备选master的原则

- 不在线的OUT
- 响应慢的OUT
- 与原master断开时间久的OUT
- 优先原则

  优先级

  offset

  runid

选出新的master之后，发送指令（ sentinel ）给其他的slave：

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

总结：故障转移阶段

1. 发现问题，主观下线与客观下线
2. 竞选负责人
3. 优选新master
4. 新master上任，其他slave切换master，原master作为slave故障恢复后连接

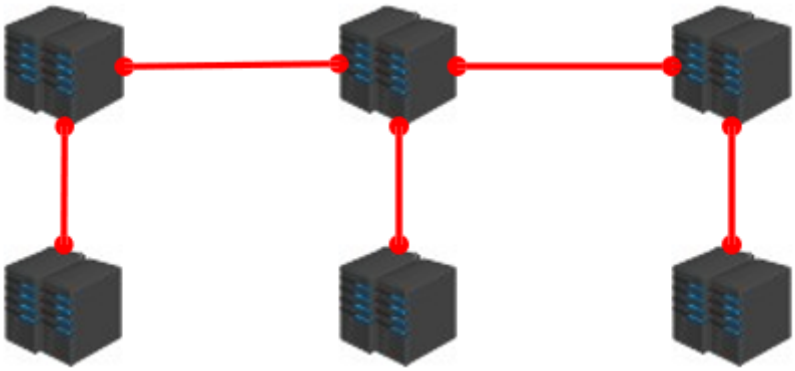
4.集群cluster

现状问题：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10万/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

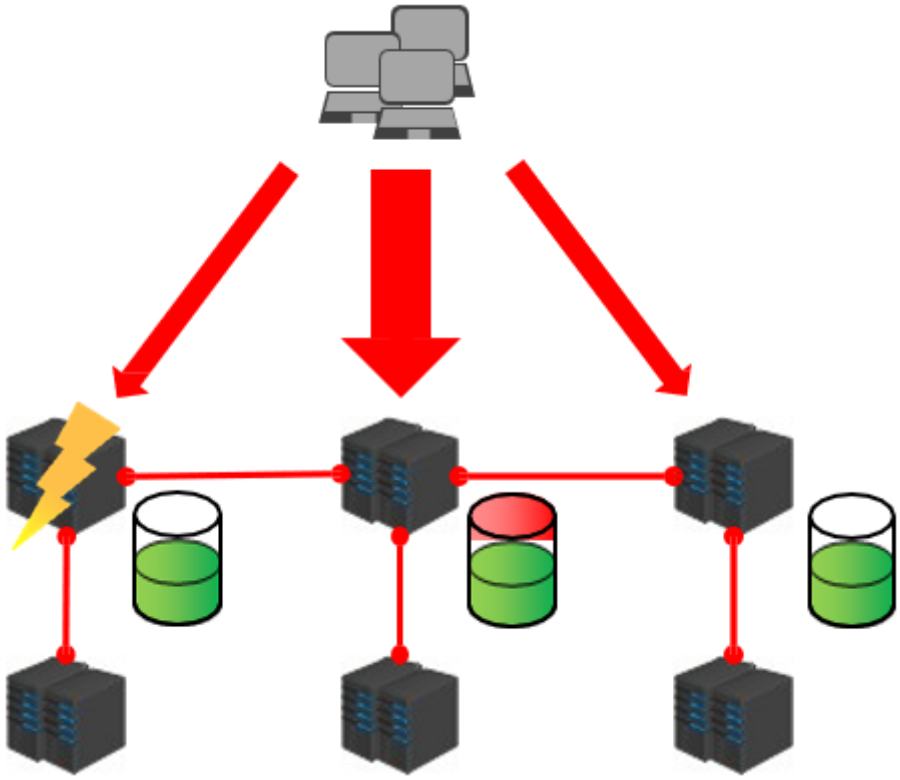
4.1 集群简介

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

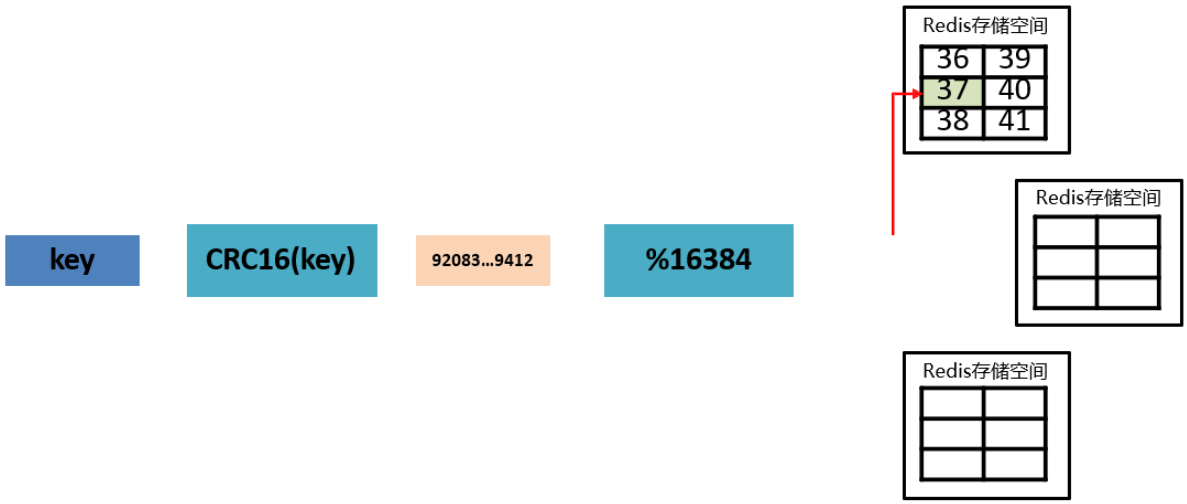


4.2 Cluster集群结构设计

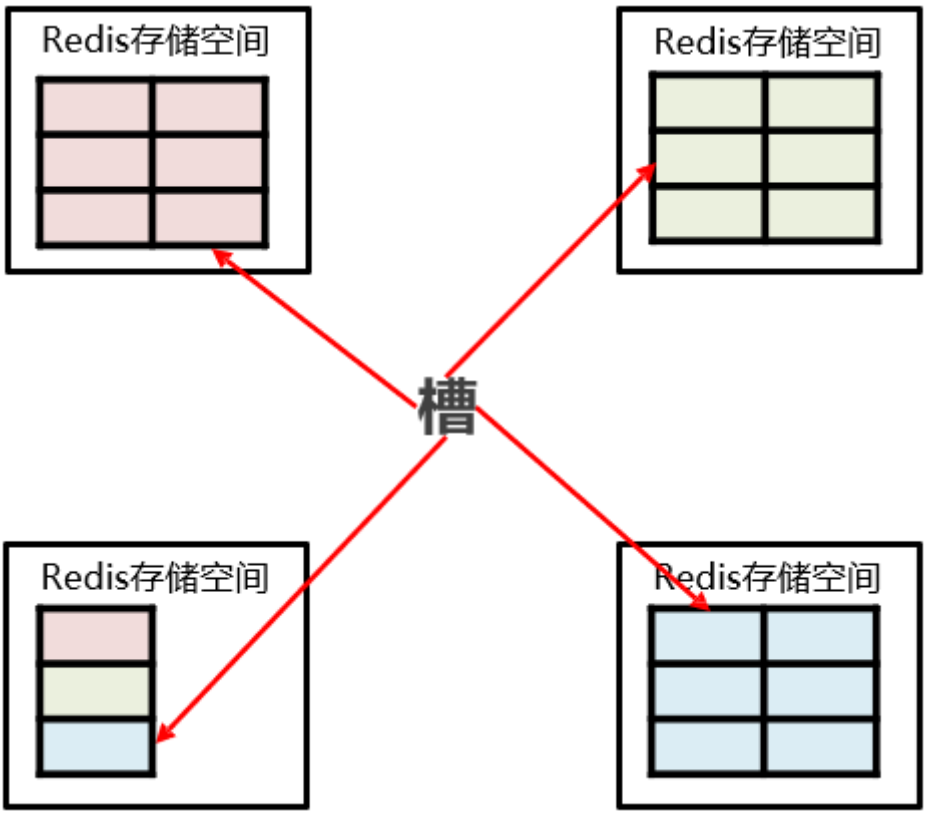
数据存储设计：

1. 通过算法设计，计算出key应该保存的位置
2. 将所有的存储空间计划切割成16384份，每台主机保存一部分

   注意：每份代表的是一个存储空间，不是一个key的保存空间
3. 将key按照计算出的结果放到对应的存储空间

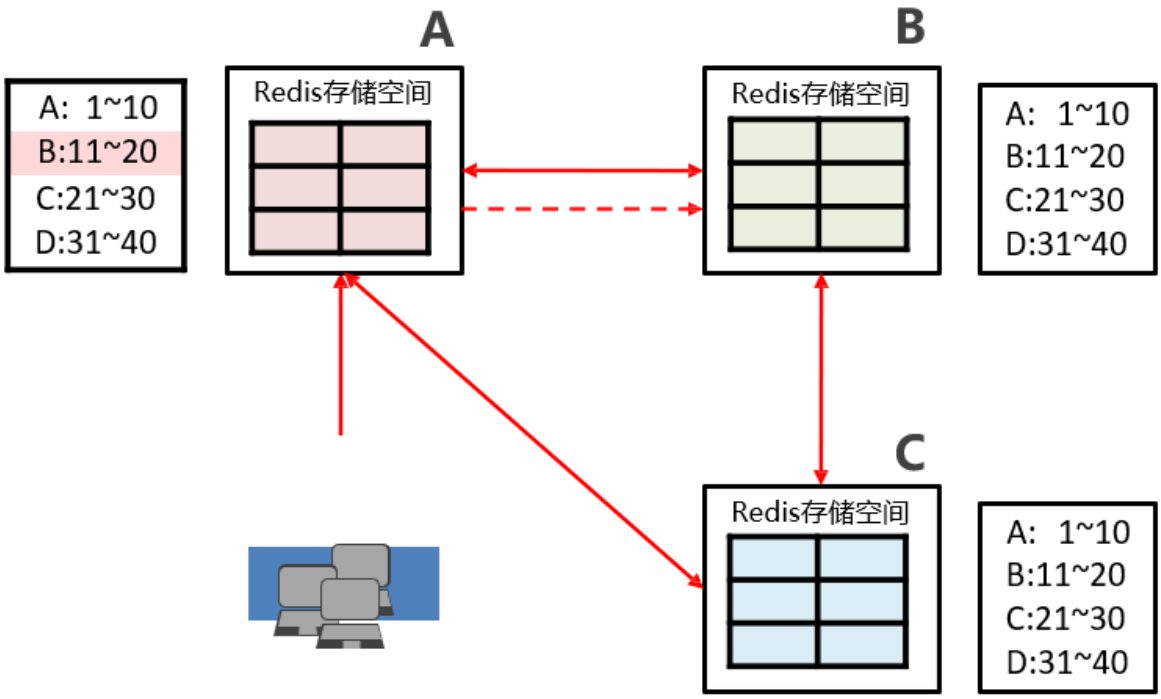


那redis的集群是如何增强可扩展性的呢？譬如我们要增加一个集群节点



当我们查找数据时，集群是如何操作的呢?

- 各个数据库相互通信，保存各个库中槽的编号数据
- 一次命中，直接返回
- 一次未命中，告知具体位置



4.3 Cluster集群结构搭建

首先要明确的几个要点:

- 配置服务器（3主3从）
- 建立通信（Meet）
- 分槽（Slot）

- 搭建主从（master-slave）

Cluster配置

- 是否启用cluster，加入cluster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 milliseconds
```

- master连接的slave最小数量

```
cluster-migration-barrier min_slave_number
```

Cluster节点操作命令

- 查看集群节点信息

```
cluster nodes
```

- 更改slave指向新的master

```
cluster replicate master-id
```

- 发现一个新节点，新增master

```
cluster meet ip:port
```

- 忽略一个没有solt的节点

```
cluster forget server_id
```

- 手动故障转移

```
cluster failover
```

集群操作命令：

- 创建集群

```
redis-cli --cluster create masterhost1:masterport1 masterhost2:masterport2
masterhost3:masterport3 [masterhostn:masterportn …] slavehost1:slaveport1
slavehost2:slaveport2 slavehost3:slaveport3 --cluster-replicas n
```

注意：master与slave的数量要匹配，一个master对应n个slave，由最后的参数n决定

master与slave的匹配顺序为第一个master与前n个slave分为一组，形成主从结构

- 添加master到当前集群中，连接时可以指定任意现有节点地址与端口

```
redis-cli --cluster add-node new-master-host:new-master-port now-host:now-port
```

- 添加slave

```
redis-cli --cluster add-node new-slave-host:new-slave-port master-host:master-port --cluster-slave --cluster-master-id masterid
```

- 删除节点，如果删除的节点是master，必须保障其中没有槽slot

```
redis-cli --cluster del-node del-slave-host:del-slave-port del-slave-id
```

- 重新分槽，分槽是从具有槽的master中划分一部分给其他master，过程中不创建新的槽

```
redis-cli --cluster reshard new-master-host:new-master:port --cluster-from src-master-id1, src-master-id2, src-master-idn --cluster-to target-master-id --cluster-slots slots
```

注意：将需要参与分槽的所有masterid不分先后顺序添加到参数中，使用，分隔

指定目标得到的槽的数量，所有的槽将平均从每个来源的master处获取

- 重新分配槽，从具有槽的master中分配指定数量的槽到另一个master中，常用于清空指定master中的槽

```
redis-cli --cluster reshard src-master-host:src-master-port --cluster-from src-master-id --cluster-to target-master-id --cluster-slots slots --cluster-yes
```

5.企业级解决方案

5.1 缓存预热

场景：“宕机”

服务器启动后迅速宕机

问题排查：

1.请求数量较高，大量的请求过来之后都需要去从缓存中获取数据，但是缓存中又没有，此时从数据库中查找数据然后将数据再存入缓存，造成了短期内对redis的高强度操作从而导致问题

2.主从之间数据吞吐量较大，数据同步操作频度较高

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：

1.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

2.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

3.热点数据主从同时预热

- 实施：

4.使用脚本程序固定触发数据预热过程

5.如果条件允许，使用了CDN（内容分发网络），效果会更好

总的来说：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！

5.2 缓存雪崩

场景：数据库服务器崩溃，一连串的场景会随之儿来

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.重启应用服务器无效

8.Redis服务器崩溃

9.Redis集群崩溃

10.重启数据库后再次被瞬间流量放倒

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请求，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

总而言之就两点：短时间范围内，大量key集中过期

解决方案

- 思路：

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

4.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

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

- 落地实践：

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

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！

总的来说：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。

5.3 缓存击穿

场景：还是数据库服务器崩溃，但是跟之前的场景有点不太一样

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数据的访问

总而言之就两点：单个key高热数据，key过期

解决方案：

1.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，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，对自然流量激增的数据延长过期时间或设置为永久性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，但是要注意也是性能瓶颈，慎重！

总的来说：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。

5.4 缓存穿透

场景：数据库服务器又崩溃了，跟之前的一样吗？

1.系统平稳运行过程中

2.应用服务器流量随时间增量较大

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URL访问

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

解决方案：

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30-60秒，最高5分钟

2.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）

2.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）

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，混淆到页面数据id中，发现访问key不满足规则，驳回数据访问

总的来说：缓存击穿是指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。

5.5 性能指标监控

redis中的监控指标如下：

- 性能指标：Performance

> 响应请求的平均时间:
>
> ```
> latency
> ```
>
> 平均每秒处理请求总数
>
> ```
> instantaneous_ops_per_sec
> ```
>
> 缓存查询命中率（通过查询总次数与查询得到非nil数据总次数计算而来）
>
> ```
> hit_rate(calculated)
> ```

- 内存指标：Memory

> 当前内存使用量
>
> ```
> used_memory
> ```
>
> 内存碎片率（关系到是否进行碎片整理）
>
> ```
> mem_fragmentation_ratio
> ```
>
> 为避免内存溢出删除的key的总数量
>
> ```
> evicted_keys
> ```
>
> 基于阻塞操作（BLPOP等）影响的客户端数量
>
> ```
> blocked_clients
> ```

- 基本活动指标：Basic_activity

> 当前客户端连接总数
>
> ```
> connected_clients
> ```

> 当前连接slave总数
>
> ```
> connected_slaves
> ```
>
> 最后一次主从信息交换距现在的秒
>
> ```
> master_last_io_seconds_ago
> ```
>
> key的总数
>
> ```
> keyspace
> ```

- 持久性指标：Persistence

> 当前服务器最后一次RDB持久化的时间
>
> ```
> rdb_last_save_time
> ```
>
> 当前服务器最后一次RDB持久化后数据变化总量
>
> ```
> rdb_changes_since_last_save
> ```

- 错误指标：Error

> 被拒绝连接的客户端总数（基于达到最大连接值的因素）
>
> ```
> rejected_connections
> ```
>
> key未命中的总次数
>
> ```
> keyspace_misses
> ```
>
> 主从断开的秒数
>
> ```
> master_link_down_since_seconds
> ```

要对redis的相关指标进行监控，我们可以采用一些用具：

- CloudInsight Redis
- Prometheus
- Redis-stat
- Redis-faina
- RedisLive
- zabbix

也有一些命令工具：

- benchmark

测试当前服务器的并发性能

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

范例1：50个连接，10000次请求对应的性能

```
redis-benchmark
```

范例2：100个连接，5000次请求对应的性能

```
redis-benchmark -c 100 -n 5000
```

序号	选项	描述	默认值
1	-h	指定服务器主机名	127.0.0.1
2	-p	指定服务器端口	6379
3	-s	指定服务器 socket	
4	-c	指定并发连接数	50
5	-n	指定请求数	10000
6	-d	以字节的形式指定 SET/GET 值的数据大小	2
7	-k	1=keep alive 0=reconnect	1
8	-r	SET/GET/INCR 使用随机 key, SADD 使用随机值	
9	-P	通过管道传输 <numreq> 请求	1
10	-q	强制退出 redis。仅显示 query/sec 值	
11	--csv	以 CSV 格式输出	
12	-l	生成循环，永久执行测试	
13	-t	仅运行以逗号分隔的测试命令列表。	
14	-I	Idle 模式。仅打开 N 个 idle 连接并等待。	

- redis-cli

  monitor：启动服务器调试信息

```
monitor
```

`slowlog`：慢日志

获取慢查询日志

```
slowlog [operator]
```

get： 获取慢查询日志信息

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
```