Redis

1.Redis基本的概述问题

1. redis 和memcache 的区别

```
相同点:都是将数据缓存在内存中
区别:
支持存储的数据类型:
    redis 支持value存储的数据类型有5种  string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash (哈希类型)
    memcache 仅支持 字符串的存储
持久化:
    redis 支持持久化的存储
    memcache 存储基于LRU,不支持持久化,当出现宕机数据会丢失
线程:
    redis 采用的是单线程服务(意味着会有较多的阻塞情况),单线程IO复用模型
    memcache 采用的是多线程(意味着阻塞相对较少),非阻塞IO复用网络模型
```

2.Redis 的数据结构

结构类型	结构存储的值	结构的读写能力
String	可以是字符串，整数，浮点数	对整个字符串或者字符串的其中一部分进行操作，对整数和浮点数执行自增或者自减操作
list	一个链表，链表上每个节点都包含了一个字符串	从链表的俩端推入或者弹出元素：根据便移量对链表进行修剪：读取单个或多个元素，根据值查找或者移除元素
set	包含字符串的无序收集器，并且被包含的每个字符串都是独一无二各不相同的	添加、获取、移除单个元素，检查一个元素是否存在与集合中，计算交集，并集，差集，从集合里面随机获取元素
hash	包含键值对的无序散列表	添加、获取、移除单个键值对，获取所有键值对
zset	字符串成员与浮点数分值之间的有序映射，元素的排列顺序由分值的大小决定	添加、获取、删除单个元素，根据分值范围或者成员来获取元素

3.什么是一致性哈希

一致性哈希
一致性hash算法（DHT）可以通过减少影响范围的方式，解决增减服务器导致的数据散列问题，从而解决了分布式环境下负载均衡问题；
如果存在热点数据，可以通过增添节点的方式，对热点区间进行划分，将压力分配至其他服务器，重新达到负载均衡的状态。
Python模块--hash_ring，即Python中的一致性hash

4.redis中数据库默认是多少个db及作用?

默认支持16个数据库,在配置文件的databases可以进行修改
客户端与Redis建立连接后会自动选择0号数据库，不过可以随时使用SELECT命令更换数据库

5.如果redis中的某个列表中的数据量非常大,如果实现循环显示每一个值?

```
# 如果一个列表在redis中保存了10w个值，我需要将所有值全部循环并显示，请问如何实现?
# 一个一个取值，列表没有iter方法，但能自定义
def list_scan_iter(name,count=3):
    start = 0
    while True:
        result = conn.lrange(name, start, start+count-1)  # Lrange 返回列表中指定区间内的元素

        start += count
        if not result:
            break
        for item in result:
            yield item

for val in list_scan_iter('num_list'):
    print(val)
# 场景：投票系统，script-redis

#  ------------------------

# 通过scan_iter分片取，减少内存压力
scan_iter(match=None, count=None)增量式迭代获取redis里匹配的的值
# match，匹配指定key
# count，每次分片最少获取个数
    r = redis.Redis(connection_pool=pool)
    for key in r.scan_iter(match='PREFIX_*', count=100000):
        print(key)
```

2.Redis 主从同步

1.Redis 如何实现主从同步

```
#1.实现主从复制：环境准备
    '创建6379和6380配置文件'
    redis.conf：6379为默认配置文件，作为Master服务配置；
    redis_6380.conf：6380为同步配置，作为Slave服务配置；
    redis_6380.conf：6381为同步配置，作为Slave服务配置；
#2. '配置slaveof同步指令'
```

```
    在Slave对应的conf配置文件中，添加以下内容:
    slaveof 127.0.0.1 6379    #指明主的地址
    slaveof 127.0.0.1 6379    #指明主的地址
#3.写入数据，关闭主库6379
    检查从库主从信息，此时master_link_status:down
#4.关闭6380的从库身份
    redis-cli -p 6380
    info replication
    slaveof no one
#将6381设为6380的从库
    6382连接到6381:
    [root@db03 ~]# redis-cli -p 6381
    127.0.0.1:6381> SLAVEOF no one
    127.0.0.1:6381> SLAVEOF 127.0.0.1 6381
```

2.Redis主从同步的数据同步机制

```
原理:
1. 从服务器向主服务器发送 SYNC 命令。
2. 接到 SYNC 命令的主服务器会调用BGSAVE 命令，创建一个 RDB文件，并使用缓冲区记录接下来执行的所有写命令。
3. 当主服务器执行完 BGSAVE 命令时，它会向从服务器发送 RDB 文件，而从服务器则会接收并载入这个文件。
4. 主服务器将缓冲区储存的所有写命令发送给从服务器执行。
```

3.Redis持久化

1. 简述Redis有哪几种持久化策略以比较

```
由于Redis的数据都存放在内存中，如果没有配置持久化，redis重启后数据就全丢失了，于是需要开启redis的持久化功能，将数据保存到磁盘上，当redis重启后，可以从磁盘中恢复数据。
redis提供两种方式进行持久化，一种是RDB持久化,另外一种是AOF持久化。
```

```
# ------------------  RDB持久化 --------------
# 1. 原理
    RDB持久化是指在指定的时间间隔内将通过save命令将内存中的数据生成生成RDB快照文件
    RDB文件是经过压缩的二进制文件，这个文件被保存在硬盘中，redis可以通过这个文件还原数据库当时的状态。
# 2. 过程:
    Redis调用fork()，产生一个子进程。子进程把数据写到一个临时的RDB文件。当子进程写完新的RDB文件后，把旧的RDB文件替换掉
# 2. 优点
    RDB文件是一个很简洁的单文件，它保存了某个时间点的Redis数据，很适合用于做备份。你可以设定一个时间点对RDB文件进行归档，这样就能在需要的时候很轻易的把数据恢复到不同的版本。比起AOF，在数据量比较大的情况下，RDB的启动速度更快。
# 3. 缺点
    RDB容易造成数据的丢失。假设每5分钟保存一次快照，如果Redis因为某些原因不能正常工作，那么从上次产生快照到Redis出现问题这段时间的数据就会丢失了。RDB使用fork()产生子进程进行数据的持久化，如果数据比较大的话可能就会花费点时间，造成Redis停止服务几毫秒。如果数据量很大且CPU性能不是很好的时候，停止服务的时间甚至会到1秒。
```

```
# ------------------  AOF持久化 --------------------
# 1. 原理
    AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，生成AOF文件，重启Redis时，AOF里的命令会被重新执行一次，重建数据。
# 2. 过程
    Redis调用fork()，产生一个子进程。子进程把新的AOF写到一个临时文件里。主进程持续把新的变动写到内存里的buffer，同时也会把这些新的变动写到旧的AOF里，这样即使重写失败也能保证数据的安全。当子进程完成文件 的重写后，主进程会获得一个信号，然后把内存里的buffer追加到子进程生成的那个新AOF里
# 2. 优点
    比RDB可靠。你可以制定不同的fsync策略：不进行fsync、每秒fsync一次和每次查询进行fsync。默认是每秒fsync一次。这意味着你最多丢失一秒钟的数据。
# 3.缺点:
    在相同的数据集下，AOF文件的大小一般会比RDB文件大。
    日志重写：新文件上会写入能重建当前数据集的最小操作命令的集合。
```

```
小结
redis 持久化方式有哪些? 有什么区别?
rdb: 基于快照的持久化，速度更快，一般用作备份，主从复制也是依赖于rdb持久化功能
aof: 以追加的方式记录redis操作日志的文件。可以最大程度的保证redis数据安全，类似于mysql的binlog
```

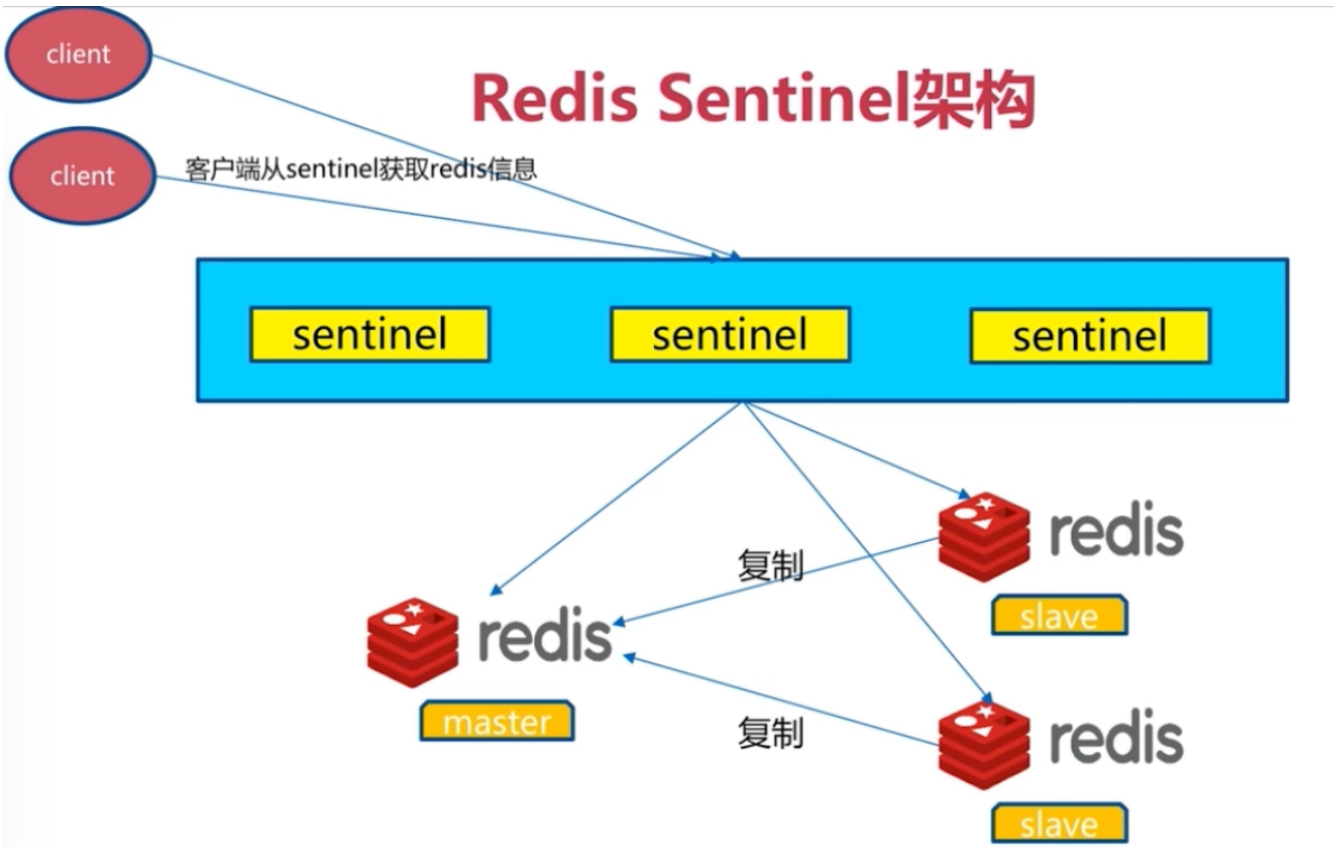
4.redis中sentinel(哨兵)

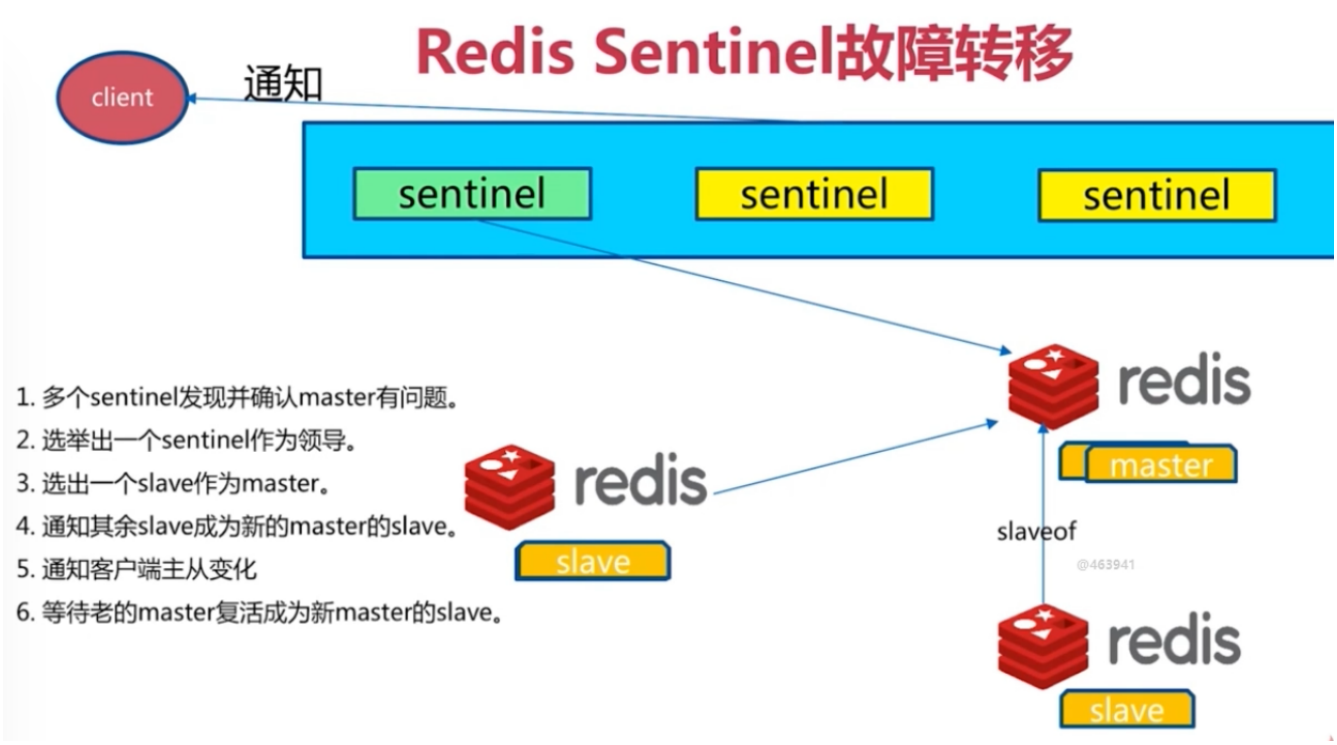
1. 使用背景

```
# Redis主从复制可将主节点数据同步给从节点，从节点此时有两个作用:
    一旦主节点宕机，从节点作为主节点的备份可以随时顶上来。
    扩展主节点的读能力，分担主节点读压力。
# 但是问题是:
    一旦主节点宕机，从节点上位，那么需要人为修改所有应用方的主节点地址（改为新的master地址），还需要命令所有从节点复制新的主节点
    那么这个问题，redis-sentinel就可以解决了
# 而redis-sentinel就是一个独立运行的进程，用于监控多个master-slave集群，自动发现master宕机，进行自动切换slave > master。
```

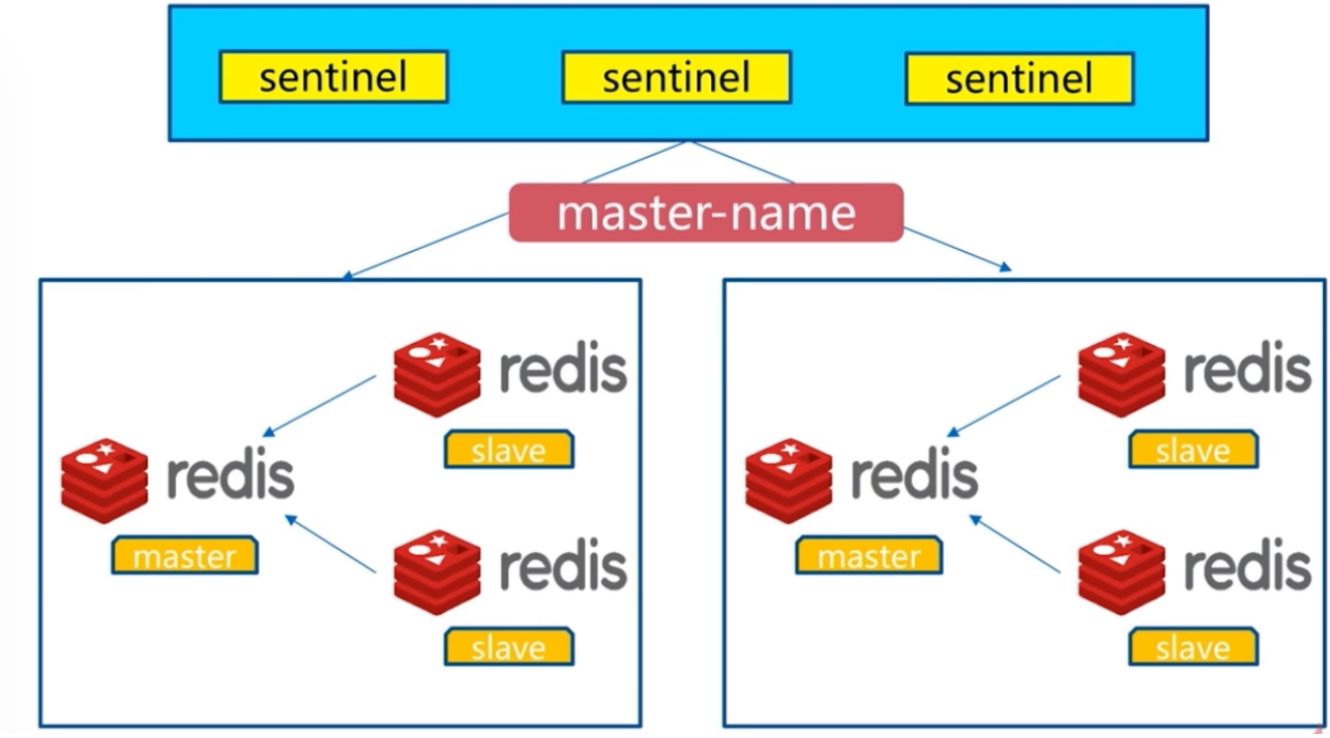
2.功能实现

```
#不时的监控redis是否良好运行，如果节点不可达就会对节点进行下线标识
#如果被标识的是主节点，sentinel就会和其他的sentinel节点“协商”，如果其他节点也人为主节点不可达，就会选举一个sentinel节点来完成自动故障转义
#在master-slave进行切换后，master_redis.conf、slave_redis.conf和sentinel.conf的内容都会发生改变，即master_redis.conf中会多一行slaveof的配置，sentinel.conf的监控目标会随之调换
```





1. 多个sentinel发现并确认master有问题。
2. 选举出一个sentinel作为领导。
3. 选出一个slave作为master。
4. 通知其余slave成为新的master的slave。
5. 通知客户端主从变化
6. 等待老的master复活成为新master的slave。



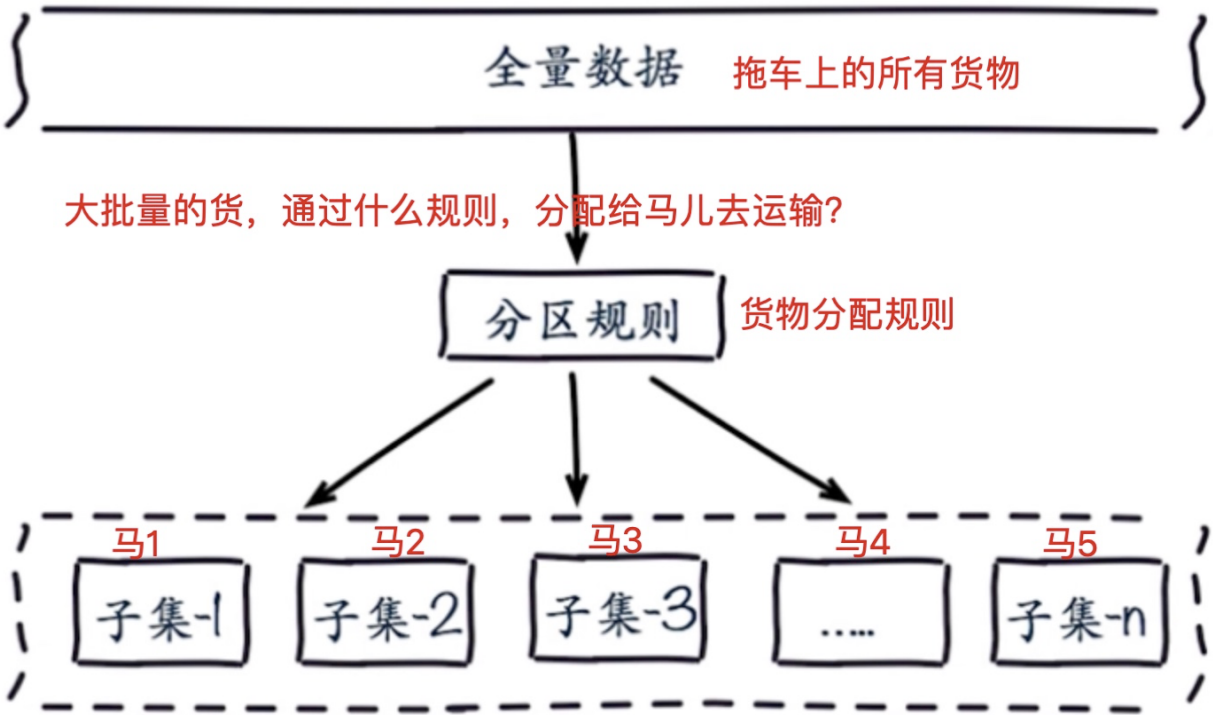
5.Redis - cluster（集群）

1.数量太大

```
一台服务器内存正常是16~256G，假如你的业务需要500G内存，
    核心思想都是将数据分片（sharding）存储在多个redis实例中，每一片就是一个redis实例。

各大企业集群方案：
    twemproxy由Twitter开源
    Codis由豌豆荚开发，基于GO和C开发
    redis-cluster官方3.0版本后的集群方案
```

2.数据分布原理图

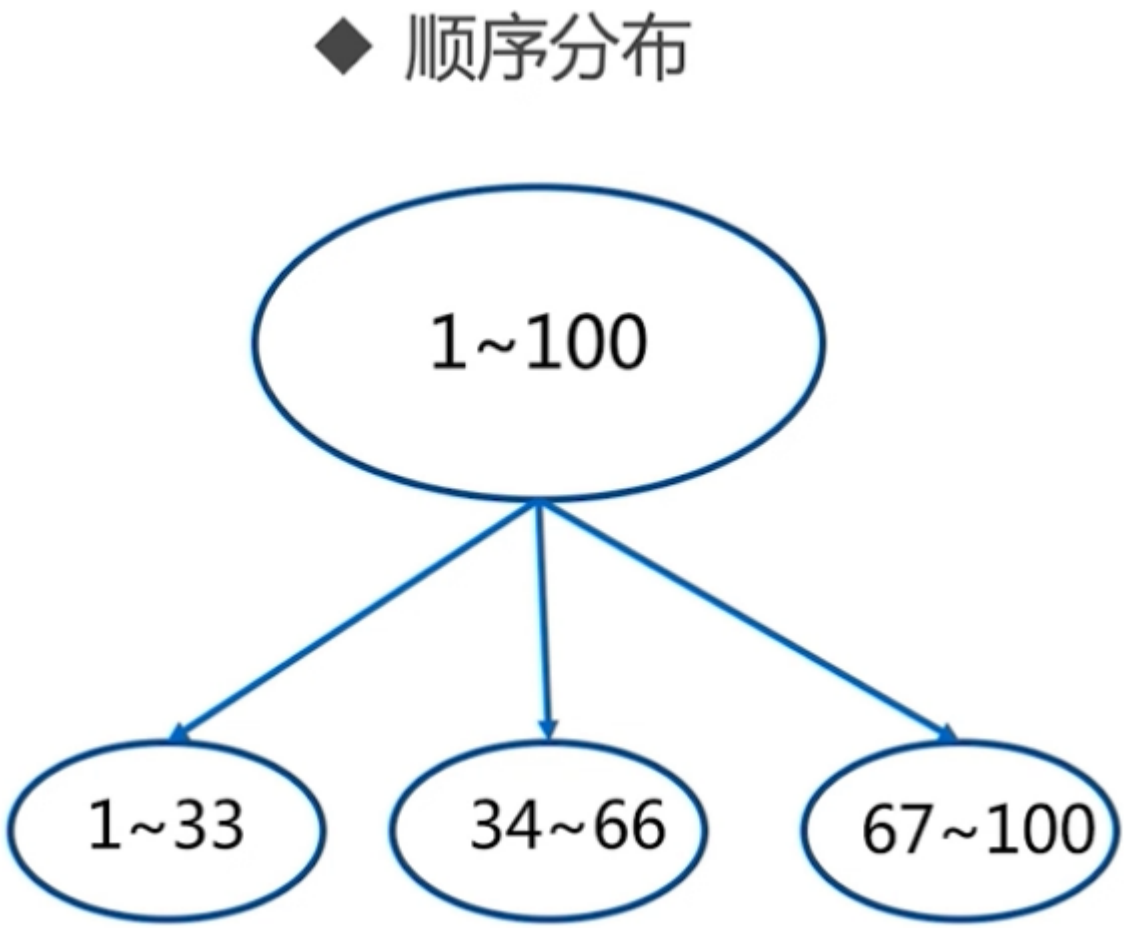


3.数据分布理论

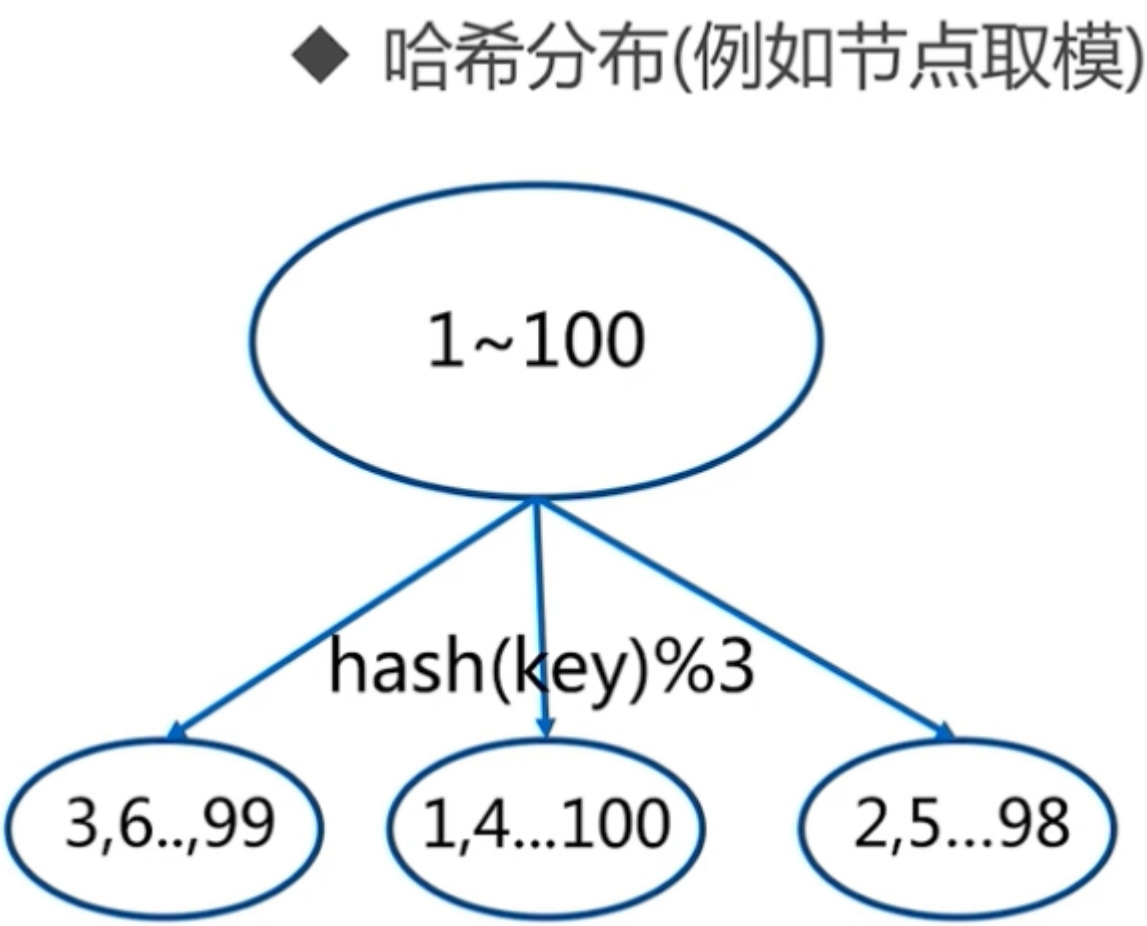
```
#分布式数据库首要解决把整个数据集按照分区规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整个数据的一个子集。

#Redis Cluster采用哈希分区规则，因此接下来会讨论哈希分区规则。
    节点取余分区
    一致性哈希分区
    虚拟槽分区(redis-cluster采用的方式)
```

3.数据分区

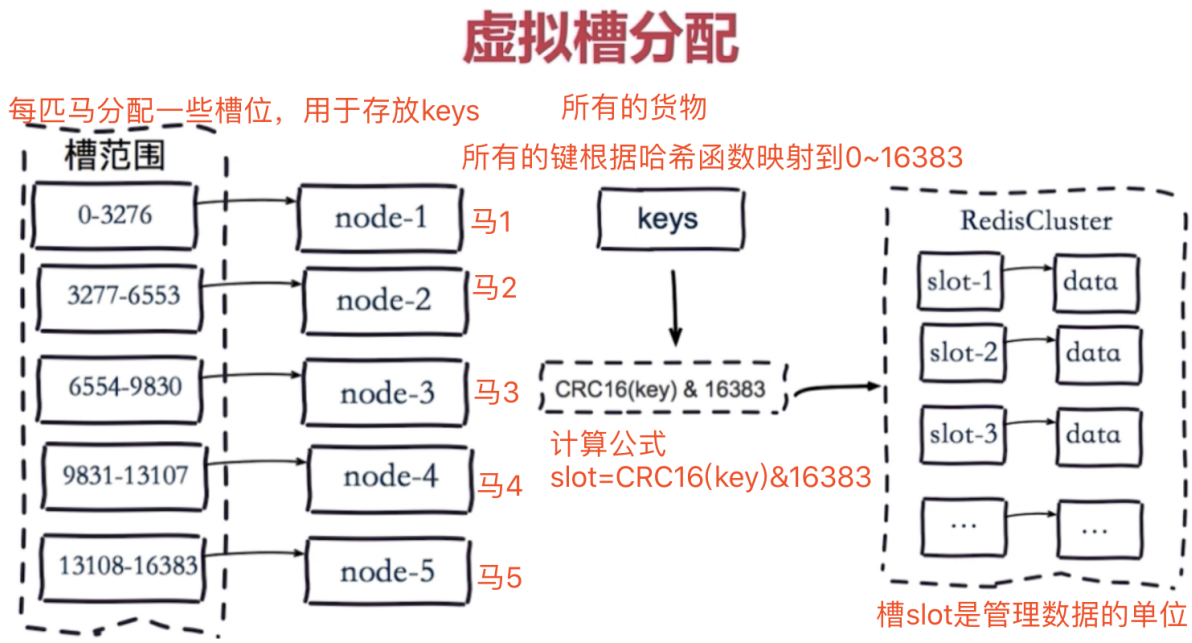


4.哈希分区



例如按照节点取余的方式，分三个节点
1~100的数据对3取余，可以分为三类
余数为0
余数为1
余数为2
那么同样的分4个节点就是hash(key)%4
节点取余的优点是简单，客户端分片直接是哈希+取余

5. 虚拟槽分区 ---`Redis Cluster`采用虚拟槽分区




```
#即使使用哨兵，redis每个实例也是全量数据存储，每个redis存储的内容都是完整的数据。
#为了最大化利用内存，可以采用cluster群集 -->分布式存储，即每台redis存储不同的内容。
    采用redis-cluster架构正是满足这种分布式存储要求的集群的一种体现。
    redis-cluster架构中，被设计成共有16384个hash slot。
    每个master分得一部分slot，其算法为：hash_slot = crc16(key) mod 16384 ，这就找到对应slot。
    采用hash slot的算法，实际上是解决了redis-cluster架构下，有多个master节点的时候，数据如何分布到这些节点上去。
    key是可用key，如果有{}则取{}内的作为可用key，否则整个可以是可用key。群集至少需要3主3从，且每个实例使用不同的配置文件。
```

6.Redis过期策略

1. Redis支持的过期策略

```
volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
no-enviction（驱逐）：禁止驱逐数据,不删除策略。当达到最大内存限制时，如果需要使用更多内存,则直接返回错误信息。
```

2.MySQL里有2000w数据, redis中只存20w的数据,如何保证 redis中都是热点数据?

方案：
限定 Redis 占用的内存，Redis 会根据自身数据淘汰策略，留下热数据到内存。所以，计算一下 50w 数据大约占用的内存，然后设置一下 Redis 内存限制即可，并将淘汰策略为volatile-lru或者allkeys-lru。

设置Redis最大占用内存：
打开redis配置文件，设置maxmemory参数，maxmemory是bytes字节类型

```
# In short... if you have slaves attached it is suggested that you set a lower# limit for maxmemory so that there is some free RAM on the system for slave# output buffers (but this is not needed if the policy is 'noeviction').## maxmemory <bytes>maxmemory 268435456
```

设置过期策略：
maxmemory-policy volatile-lru

7.Redis 实现队列

1.写代码,基于 redis的列表实现先进先出、后进先出队列、优先级队列

```
# 参看script-redis源码
from scrapy.utils.reqser import request_to_dict, request_from_dict

from . import picklecompat


class Base(object):
    """Per-spider base queue class"""

    def __init__(self, server, spider, key, serializer=None):
        """Initialize per-spider redis queue.

        Parameters
        ----------
        server : StrictRedis
            Redis client instance.
        spider : Spider
            Scrapy spider instance.
        key: str
            Redis key where to put and get messages.
        serializer : object
            Serializer object with ``loads`` and ``dumps`` methods.

        """
        if serializer is None:
            # Backward compatibility.
            # TODO: deprecate pickle.
            serializer = picklecompat
        if not hasattr(serializer, 'loads'):
            raise TypeError("serializer does not implement 'loads' function: %r"
                            % serializer)
        if not hasattr(serializer, 'dumps'):
            raise TypeError("serializer '%s' does not implement 'dumps' function: %r"
                            % serializer)
```

```
        self.server = server
        self.spider = spider
        self.key = key % {'spider': spider.name}
        self.serializer = serializer

    def _encode_request(self, request):
        """Encode a request object"""
        obj = request_to_dict(request, self.spider)
        return self.serializer.dumps(obj)

    def _decode_request(self, encoded_request):
        """Decode an request previously encoded"""
        obj = self.serializer.loads(encoded_request)
        return request_from_dict(obj, self.spider)

    def __len__(self):
        """Return the length of the queue"""
        raise NotImplementedError

    def push(self, request):
        """Push a request"""
        raise NotImplementedError

    def pop(self, timeout=0):
        """Pop a request"""
        raise NotImplementedError

    def clear(self):
        """Clear queue/stack"""
        self.server.delete(self.key)


class FifoQueue(Base):
    """Per-spider FIFO queue"""

    def __len__(self):
        """Return the length of the queue"""
        return self.server.llen(self.key)

    def push(self, request):
        """Push a request"""
        self.server.lpush(self.key, self._encode_request(request))

    def pop(self, timeout=0):
        """Pop a request"""
        if timeout > 0:
            data = self.server.brpop(self.key, timeout)
            if isinstance(data, tuple):
                data = data[1]
        else:
            data = self.server.rpop(self.key)
        if data:
            return self._decode_request(data)
```

```
class PriorityQueue(Base):
    """Per-spider priority queue abstraction using redis' sorted set"""

    def __len__(self):
        """Return the length of the queue"""
        return self.server.zcard(self.key)

    def push(self, request):
        """Push a request"""
        data = self._encode_request(request)
        score = -request.priority
        # We don't use zadd method as the order of arguments change depending on
        # whether the class is Redis or StrictRedis, and the option of using
        # kwargs only accepts strings, not bytes.
        self.server.execute_command('ZADD', self.key, score, data)

    def pop(self, timeout=0):
        """
        Pop a request
        timeout not support in this queue class
        """
        # use atomic range/remove using multi/exec
        pipe = self.server.pipeline()
        pipe.multi()
        pipe.zrange(self.key, 0, 0).zremrangebyrank(self.key, 0, 0)
        results, count = pipe.execute()
        if results:
            return self._decode_request(results[0])


class LifoQueue(Base):
    """Per-spider LIFO queue."""

    def __len__(self):
        """Return the length of the stack"""
        return self.server.llen(self.key)

    def push(self, request):
        """Push a request"""
        self.server.lpush(self.key, self._encode_request(request))

    def pop(self, timeout=0):
        """Pop a request"""
        if timeout > 0:
            data = self.server.blpop(self.key, timeout)
            if isinstance(data, tuple):
                data = data[1]
        else:
            data = self.server.lpop(self.key)

        if data:
```

```
            return self._decode_request(data)


# TODO: Deprecate the use of these names.
SpiderQueue = FifoQueue
SpiderStack = LifoQueue
SpiderPriorityQueue = PriorityQueue
```

2.如何基于 redis实现消息队列?

```
# 通过发布订阅模式的PUB、SUB实现消息队列
# 发布者发布消息到频道了，频道就是一个消息队列。
# 发布者:
import redis
conn = redis.Redis(host='127.0.0.1',port=6379)
conn.publish('104.9MH', "hahahahahaha")
# 订阅者:
import redis
conn = redis.Redis(host='127.0.0.1',port=6379)
pub = conn.pubsub()
pub.subscribe('104.9MH')
while True:
    msg= pub.parse_response()
    print(msg)
#对了，redis 做消息队列不合适
#业务上避免过度复用一个redis，用它做缓存、做计算，还做任务队列，压力太大，不好。
```

8.如何基于 redis实现发布和订阅?

```
#发布和订阅，只要有任务就给所有订阅者没人一份
#发布者:
import redis

conn = redis.Redis(host='127.0.0.1',port=6379)
conn.publish('104.9MH', "hahaha")

#订阅者:
import redis

conn = redis.Redis(host='127.0.0.1',port=6379)
pub = conn.pubsub()
pub.subscribe('104.9MH')

while True:
    msg= pub.parse_response()
    print(msg)
```

9.其他

1.什么是 codis?

Codis 是一个分布式 Redis 解决方案，对于上层的应用来说，连接到 Codis Proxy 和连接原生的 Redis Server 没有明显的区别
(不支持的命令列表)，上层应用可以像使用单机的 Redis 一样使用，Codis 底层会处理请求的转发，不停机的数据迁移等工作，
所有后边的一切事情，对于前面的客户端来说是透明的，可以简单的认为后边连接的是一个内存无限大的 Redis 服务.

2.什么是 twemproxy

是 Twtter 开源的一个 Redis 和 Memcache 代理服务器，主要用于管理 Redis 和 Memcached 集群，
减少与Cache 服务器直接连接的数量。

3.redis如何实现事务

```
import redis

pool = redis.ConnectionPool(host='10.211.55.4', port=6379)

conn = redis.Redis(connection_pool=pool)

# pipe = r.pipeline(transaction=False)
pipe = conn.pipeline(transaction=True)
# 开始事务
pipe.multi()

pipe.set('name', 'bendere')
pipe.set('role', 'sb')

# 提交
pipe.execute()

#注意：咨询是否当前分布式redis是否支持事务
```

4.redis中的 watch的命令的作用?

```
# 在Redis的事务中，WATCH命令可用于提供CAS(check - and -set)
# 功能。
# 假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，
# EXEC命令执行的事务都将被放弃，同时返回Null
# multi - bulk应答以通知调用者事务执行失败。
#
# 面试题：你如何控制剩余的数量不会出问题?
# 方式一：- 通过redis的watch实现
import redis

conn = redis.Redis(host='127.0.0.1', port=6379)

# conn.set('count',1000)
val = conn.get('count')
print(val)

with conn.pipeline(transaction=True) as pipe:
```

```
    # 先监视，自己的值没有被修改过
    conn.watch('count')

    # 事务开始
    pipe.multi()
    old_count = conn.get('count')
    count = int(old_count)
    print('现在剩余的商品有:%s', count)
    input("问媳妇让不让买? ")
    pipe.set('count', count - 1)

    # 执行，把所有命令一次性推送过去
    pipe.execute()
#方式二 - 数据库的锁
```

5.简述redis分布式锁和 medlock的实现机制

```
'''
在不同进程需要互斥地访问共享资源时，分布式锁是一种非常有用的技术手段。
有很多三方库和文章描述如何用Redis实现一个分布式锁管理器，但是这些库实现的方式差别很大
，而且很多简单的实现其实只需采用稍微增加一点复杂的设计就可以获得更好的可靠性。
用Redis实现分布式锁管理器的算法，我们把这个算法称为RedLock。

实现
- 写值并设置超时时间
- 超过一半的redis实例设置成功，就表示加锁完成。
- 使用：安装redlock-py
'''

from redlock import Redlock

dlm = Redlock(
    [
        {"host": "localhost", "port": 6379, "db": 0},
        {"host": "localhost", "port": 6379, "db": 0},
        {"host": "localhost", "port": 6379, "db": 0},
    ]
)

# 加锁，acquire
my_lock = dlm.lock("my_resource_name",10000)
if  my_lock:
    # J进行操作
    # 解锁,release
    dlm.unlock(my_lock)
else:
    print('获取锁失败')
# redis分布式锁?
# 不是单机操作，又多了一/多台机器
# redis内部是单进程、单线程，是数据安全的(只有自己的线程在操作数据)
```

```
'''A、B、C，三个实例(主)
```

```
1、来了一个'隔壁老王'要操作，且不想让别人操作，so，加锁；
    加锁：'隔壁老王'自己生成一个随机字符串，设置到A、B、C里(xxx=666)
2、来了一个'邻居老李'要操作A、B、C，一读发现里面有字符串，擦，被加锁了，不能操作了，等着吧~
3、'隔壁老王'解决完问题，不用锁了，把A、B、C里的key：'xxx'删掉；完成解锁
4、'邻居老李'现在可以访问，可以加锁了
# 问题：
1、如果'隔壁老王'加锁后突然挂了，就没人解锁，就死锁了，其他人干看着没法用咋办？
2、如果'隔壁老王'去给A、B、C加锁的过程中，刚加到A，'邻居老李'就去操作C了，加锁成功or失败？
3、如果'隔壁老王'去给A、B、C加锁时，C突然挂了，这次加锁是成功还是失败？
4、如果'隔壁老王'去给A、B、C加锁时，超时时间为5秒，加一个锁耗时3秒，此次加锁能成功吗？
# 解决
1、安全起见，让'隔壁老王'加锁时设置超时时间，超时的话就会自动解锁(删除key：'xxx')
2、加锁程度达到（1/2）+1个就表示加锁成功，即使没有给全部实例加锁；
3、加锁程度达到（1/2）+1个就表示加锁成功，即使没有给全部实例加锁；
4、不能成功，锁还没加完就过期，没有意义了，应该合理设置过期时间
# 注意
    使用需要安装redlock-py'''

from redlock import Redlock
dlm = Redlock(
    [
        {"host": "localhost", "port": 6379, "db": 0},
        {"host": "localhost", "port": 6379, "db": 0},
        {"host": "localhost", "port": 6379, "db": 0},
    ]
)
# 加锁，acquire
my_lock = dlm.lock("my_resource_name",10000)
if  my_lock:
    # 进行操作
    # 解锁,release
    dlm.unlock(my_lock)
else:
    print('获取锁失败')
# \通过sever.eval(self.unlock_script)执行一个lua脚本，用来删除加锁时的key
```

6.请基于redis设计一个商城商品计数器的实现方案?

```
import redis

conn = redis.Redis(host='192.168.1.41',port=6379)

conn.set('count',1000)

with conn.pipeline() as pipe:

    # 先监视，自己的值没有被修改过
    conn.watch('count')

    # 事务开始
    pipe.multi()
    old_count = conn.get('count')
    count = int(old_count)
```

```
    if count > 0:  # 有库存
        pipe.set('count', count - 1)

    # 执行，把所有命令一次性推送过去
    pipe.execute()
```


7.在项目中遇到redis性能问题?

我的项目中暂时没有遇到性能问题，不过我下来之后在学习Redis的时候，看到过一篇博客，写的很好，大概内容就是：

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3…

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

8.Redis的应用Key很大，如何取出特定的key呢?

```
# 通过scan_iter分片取，减少内存压力
scan_iter(match=None, count=None)增量式迭代获取redis里匹配的的值
# match，匹配指定key
# count，每次分片最少获取个数
    r = redis.Redis(connection_pool=pool)
    for key in r.scan_iter(match='PREFIX_*', count=100000):
        print(key)
```

9.如何高效的找到 redis中所有以 oldboy开头的key?

```
redis 有一个keys命令。
# 语法：KEYS pattern
# 说明：返回与指定模式相匹配的所用的keys。
该命令所支持的匹配模式如下：
1、?：用于匹配单个字符。例如，h?llo可以匹配hello、hallo和hxllo等；
2、*：用于匹配零个或者多个字符。例如，h*llo可以匹配hllo和heeeello等；
2、[]：可以用来指定模式的选择区间。例如h[ae]llo可以匹配hello和hallo，但是不能匹配hillo。同时，可以使用“/”符号来转义特殊的字符
# 注意
KEYS 的速度非常快，但如果数据太大，内存可能会崩掉，
如果需要从一个数据集中查找特定的key，最好还是用Redis的集合结构(set)来代替。
```

10.Redis缓存穿透、缓存雪崩和缓存击穿

缓存穿透：

- 缓存穿透，是指查询一个数据库一定不存在的数据。正常的使用缓存流程大致是，数据查询先进行缓存查询，如果key不存在或者key已经过期，再对数据库进行查询，并把查询到的对象，放进缓存。如果数据库查询对象为空，则不放进缓存
- 而每次查询都是空，每次又都不会进行缓存。假如有恶意攻击，就可以利用这个漏洞，对数据库造成压力，甚至压垮数据库。即便是采用UUID，也是很容易找到一个不存在的KEY，进行攻击。
- 如果从数据库查询的对象为空，也放入缓存，只是设定的缓存过期时间较短，比如设置为60秒。

缓存雪崩：

- 缓存雪崩，是指在某一个时间段，缓存集中过期失效。
- 产生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。
- 那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。
- 一般是采取不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

缓存击穿：

- 缓存击穿，是指一个key非常热点，在不停的扛着大并发，大并发集中对这一个点进行访问，当这个key在失效的瞬间，持续的大并发就穿破缓存，直接请求数据库，就像在一个屏障上凿开了一个洞。
- 其实，大多数情况下这种爆款很难对数据库服务器造成压垮性的压力。达到这个级别的公司没有几家的。所以，务实主义的小编，
- 对主打商品都是早早的做好了准备，让缓存永不过期。即便某些商品自己发酵成了爆款，也是直接设为永不过期就好了。