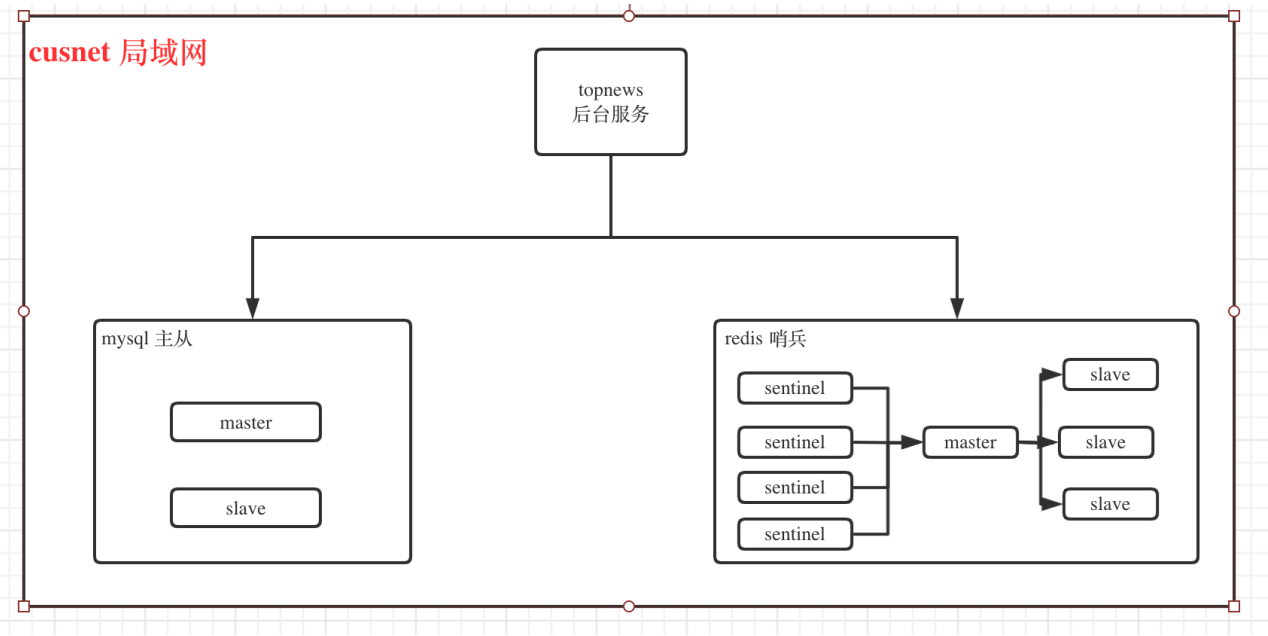
1. redis 哨兵
    1.1 python 操作
    1.2 集成到项目
    1.3 flask 项目运行在容器中
2. redis 集群
    2.1 集群搭建
    2.2 python 操作
    2.3 项目集成
3. 缓存架构
4. 缓存粒度
5. 项目缓存设计介绍
6. 缓存过期
7. 缓存淘汰

1. redis 哨兵

1.1 python 操作

1.2 集成到项目

1.3 flask 项目运行在容器中




```
docker run -it --network=cusnet -p 8000:8000 --name flask-server -w /project -v
/Users/mering/PycharmProjects/py38topnews:/project -e FLASK_APP=app.main -e
FLASK_ENV=development flask flask run -h 0.0.0.0 -p 8000
```

> `-p` 8000:8000 表示端口映射
>
> `--name` flask-server 表示容器的名字
>
> `-w` /project 是指容器运行时的工作目录
>
> `-v` /Users/mering/PycharmProjects/py38topnews:/project 表示目录映射，我们将项目的目录映射到容器的中 /project 下
>
> `-e` FLASK_APP=app.main 表示给容器内部设置环境变量，我们设置了两个环境变量
>
> `flask run -h 0.0.0.0 -p 8000` 表示在容器中指定的命令
>
> 注意：`/Users/mering/PycharmProjects/py38topnews` 要替换成自己的项目的路径

```yaml
version: "3.8"
services:
  flask-server:
    image: flask:latest
    container_name: flask-server
    network_mode: cusnet
    ports:
      - "8000:8000"
    volumes:
      - type: bind
        source: /Users/mering/PycharmProjects/py38topnews
        target: /project
    working_dir: /project
    environment:
      FLASK_APP: "app.main"
      FLASK_ENV: development
    restart: always
    command: flask run -h 0.0.0.0 -p 8000
```

2. redis 集群

1. 基本介绍

- 多个节点共同保存数据
- 作用
  - 扩展存储空间
  - 提高吞吐量, 提高写的性能
- 和单机的不同点
  - **不再区分数据库, 只有0号库**, 单机默认0-15
  - **不支持事务/管道/多值操作**
- 特点
  - 要求至少 **三主三从**
  - 要求必须开启 **AOF持久化**
  - 自动选择集群节点进行存储
  - **默认集成哨兵**, 自动故障转移

2.1 集群搭建

```
cluster-config-file nodes.conf
cluster-node-timeout 5000
# 开启集群
cluster-enabled yes
# 开启AOF 及相关配置
appendonly yes
```

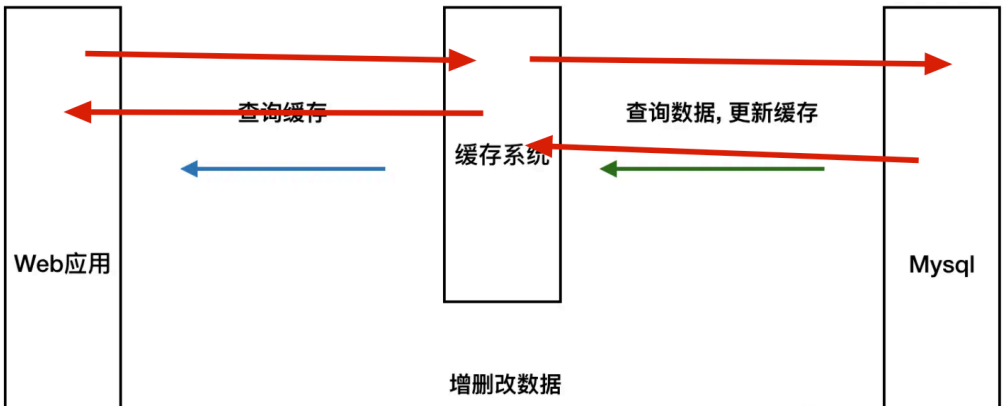
```
#!/bin/bash

REPLICAS=1
HOSTS=""

# 这里的名称是后面我们要创建的 6 个容器的名称
for HOSTNAME in "flask-redis-cluster-1" "flask-redis-cluster-2" "flask-redis-cluster-3"
do
IP=$(getent hosts $HOSTNAME | awk '{ print $1 }')
HOSTS="$HOSTS $IP:6379"
done

echo "yes"| redis-cli --cluster create $HOSTS --cluster-replicas 1

while true;
do
sleep 10
done
```

2.2 python 操作

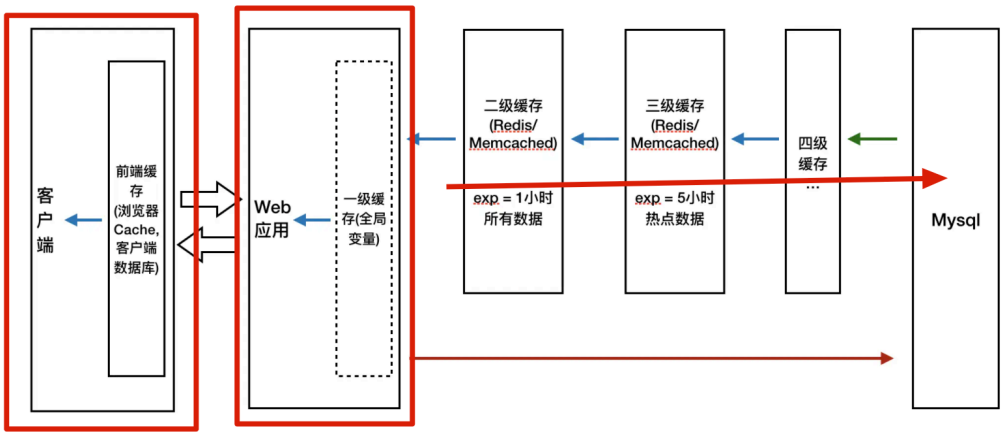
```
Python 3.8.2 (default, Apr 23 2020, 14:22:33)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.19.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from rediscluster import RedisCluster

In [2]: master_nodes=[
   ...:
   ...: {'host':'flask-redis-cluster-1','port':6379},
   ...: {'host':'flask-redis-cluster-2','port':6379},
   ...: {'host':'flask-redis-cluster-3','port':6379},
   ...: ]

In [3]:

In [3]: client = RedisCluster(startup_nodes=master_nodes)

In [4]: client.set('name','zhangsan')
Out[4]: True

In [5]: client.get('name')
Out[5]: b'zhangsan'

In [6]:
```

2.3 项目集成

3. 缓存架构



- 多级缓存



4. 缓存粒度

缓存数据对象

- 一条数据库记录 hash
- 优点: **可以多次复用**
- 场景: 用户/文章数据

```
# 用户的基本信息
user = User.query.filter_by(id=1).first()
user -> User对象
{
    'user_id':1,
    'user_name': 'python',
    'age': 28,
    'introduction': ''
}
```

缓存数据集合

- 数据库查询的结果集 set zset
- 场景: 文章/关注列表
- 项目中主要对 **数据集合+数据对象** 进行缓存, 优点 **复用性强, 节省内存**
- 使用数据集合存储数据对象键的形式也称为 **自定义redis二级索引**

5. 项目缓存设计介绍

- 缓存设计的基本思路:
  - 缓存来源于数据库, 应该根据 **项目的数据库结构** 来设计缓存
  - 数据库结构 和 缓存的关系: **基础数据表 -> 数据对象, 关系表 -> 数据集合** list set zset
  - **先设计数据对象**, 几乎所有页面都依赖基础数据表 hash
  - 数据集合的设计标准 **需要分析页面的具体使用形式**, 根据页面的使用形式 来确定 是否需要设计缓存及 缓存的数据格式

6. 缓存过期

- 常规的过期策略通常有以下三种：

- **定时过期**

  每个设置过期时间的key都创建一个定时器，到过期时间就会立即清除。该策略可以立即清除过期的数据，对内存很友好；但是会占用大量的CPU资源进行计时和处理过期数据，从而影响缓存的响应时间和吞吐量。

- **惰性过期**

  只有当访问一个key时，才会判断该key是否已过期，过期则清除(返回nil)。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。

- **定期过期**

  每隔一定的时间，扫描数据库中一部分设置了有效期的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

Redis的过期策略

Redis中同时使用了惰性过期和定期过期两种过期策略。

- 定期过期：默认是每100ms检测一次，遇到过期的key则进行删除，这里的检测并不是顺序检测，而是随机检测。
- 惰性过期：当我们去读/写一个key时，会触发Redis的惰性过期策略，直接删除过期的key

检测异步分 key，如果过期就删除，通过多次检测来溢出过期的 key

7. 缓存淘汰

2.1 LRU （Least recently used 最后使用时间策略）

- LRU算法根据数据的历史访问记录来进行淘汰数据，**优先淘汰最近没有使用过的数据**。
- 基本思路
  - 新数据插入到列表头部；
  - 每当缓存命中（即缓存数据被访问），则将数据移到列表头部；
  - 当列表满的时候，将列表尾部的数据丢弃。
- 存在的问题
  - 单独按照最后使用时间来进行数据淘汰，可能会将一些使用频繁的数据删除，如下例中数据A虽然最后使用时间比数据B早，但是其使用次数较多，后续再次使用的可能性也更大

数据	最后使用时间	使用次数
数据A	2020-03-15	100
数据B	2020-03-16	2

热点数据被移除掉
请求全部落在数据库上，导致数据库奔溃

2.2 LFU (Least Frequently Used 最少使用次数策略)

对于未来的热点数据，始终不能被缓存下来，
请求压力还是落在数据库上

- redis 4.x 后支持LFU策略
- 它是基于“如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小”的思路，**优先淘汰使用率最低的数据**

- allkeys-lfu: 当内存不足以容纳新写入数据时，在键空间中，优先移除使用次数最少的key。
- volatile-lfu: 当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，优先移除使用次数最少的key。
- allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，优先移除最近没有使用过的key。
- volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，优先移除最近没有使用过的key。
- allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。
- volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。

> 手撕 LRU，LFU