# 初步了解

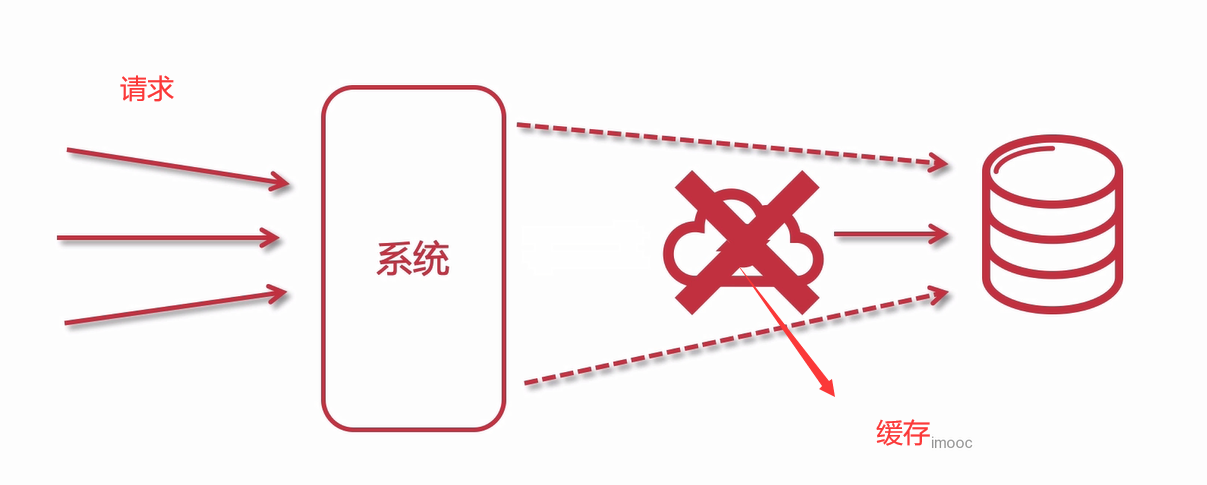
## 缓存相关

### 缓存穿透

查询的key再redis中不存在，对应的id在数据库也不存在。此时被非法用户进行攻击，大量的请求会去数据库(DB)造成宕机，从而影响整个系统。这种现象称之为 缓存穿透；

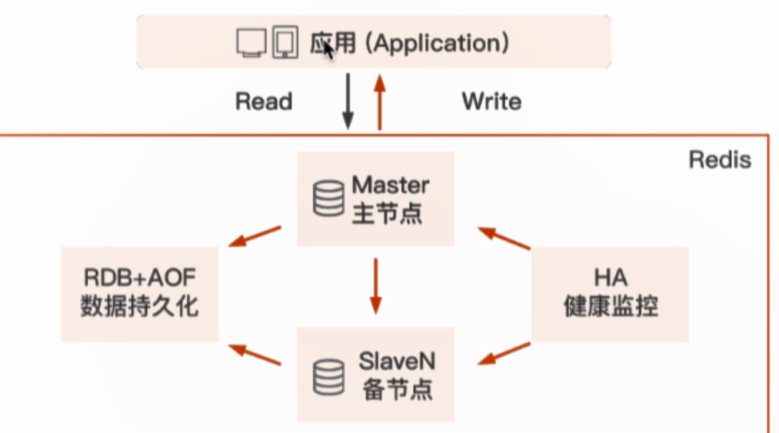
解决方法：缓存redis把空的数据也缓存到redis中，比如空字符串，空对象等；

### 缓存雪崩

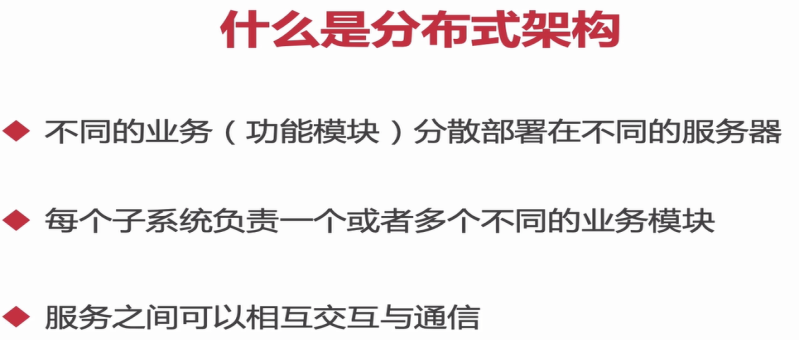


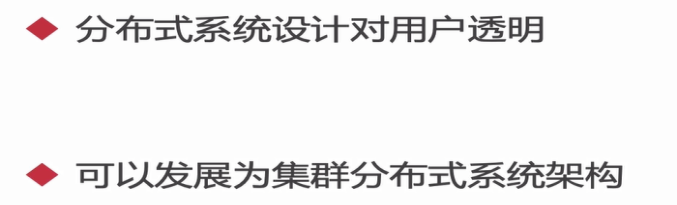
缓存雪崩一般只能缓解，不能杜绝；

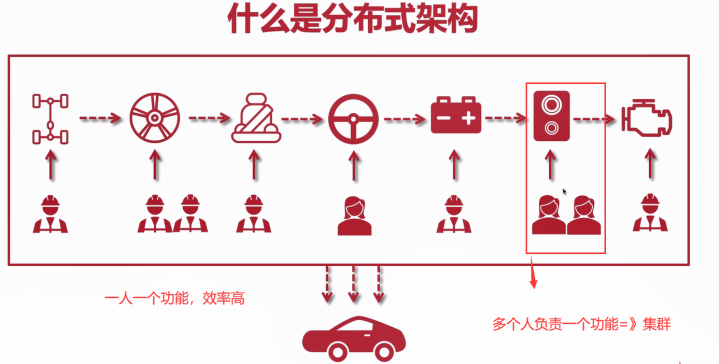
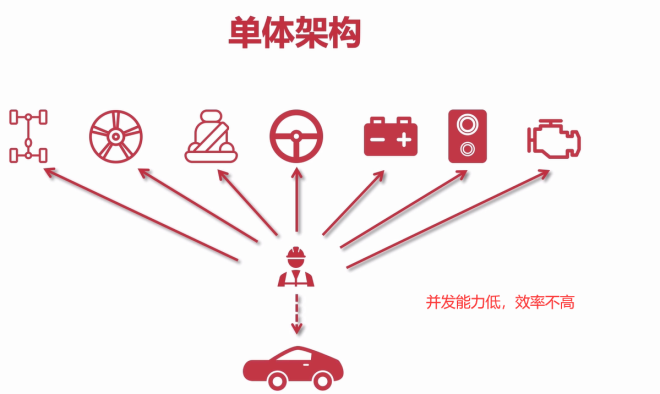
解决方式：缓存永不过期、过期时间错开(避免大量缓存同一时间过期)、多缓存结合(redis、Memcache)【先请求redis，没有就去Memcache】、采购第三方redis(比如阿里云)

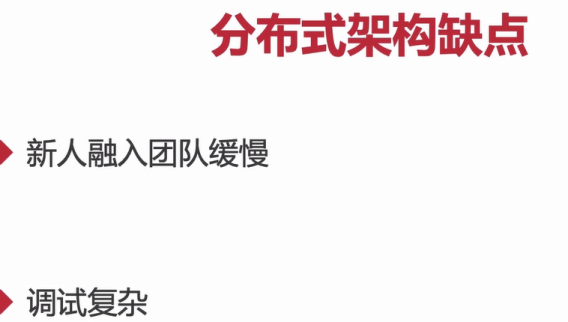
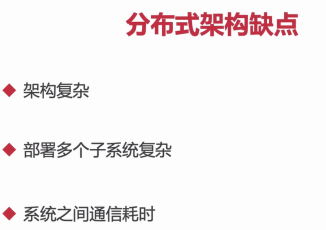
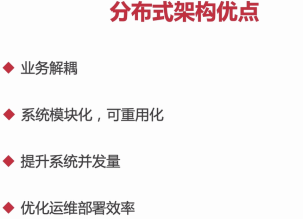


## 分布式架构

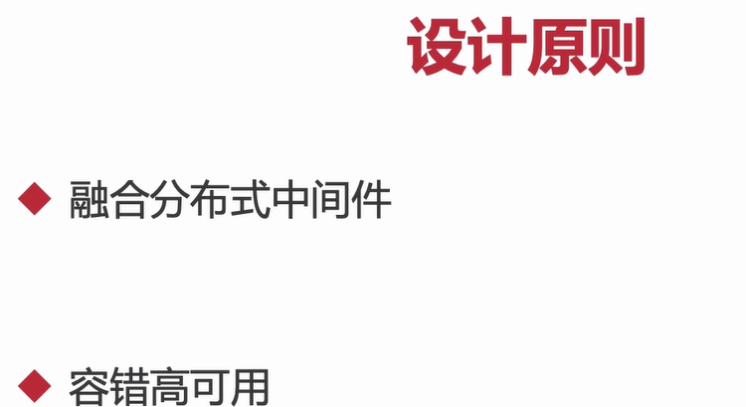
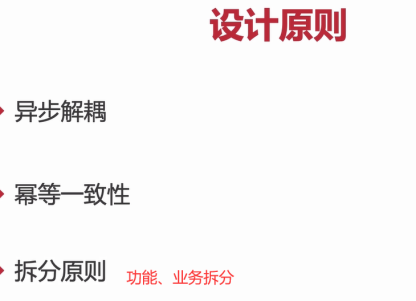




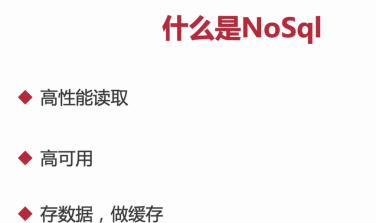
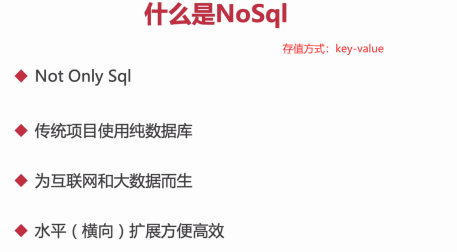




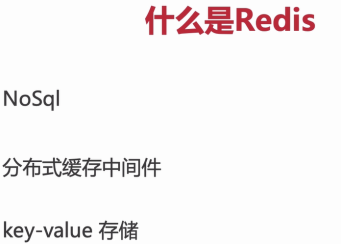
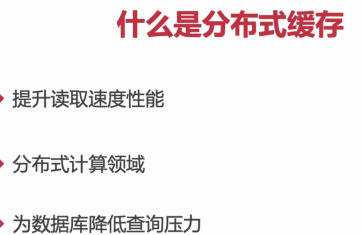
### 设计原则



## Nosql

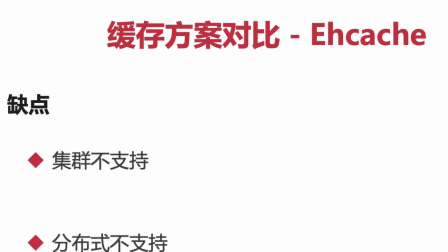
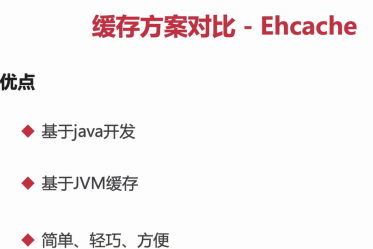


## 分布式缓存

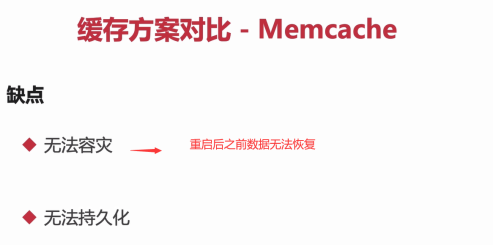


## 缓存方案对比

### Ehcache=》适合单应用



### Memcache



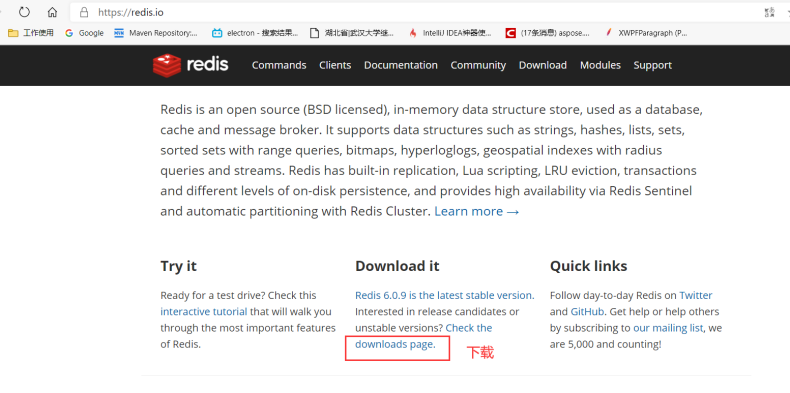
### Redis

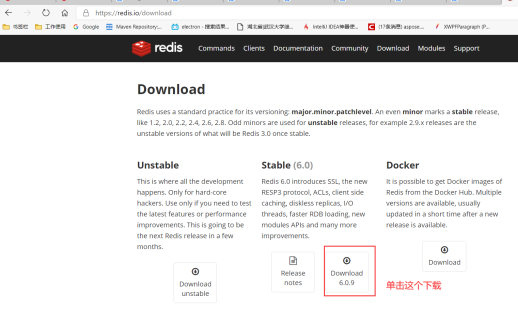


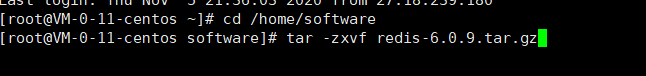
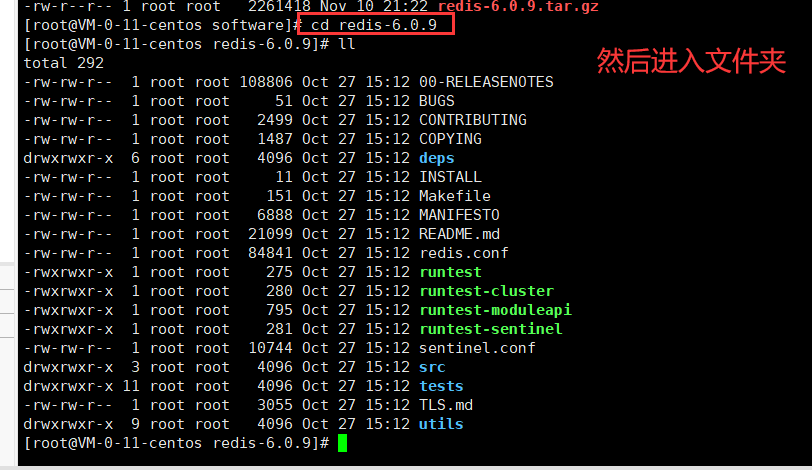
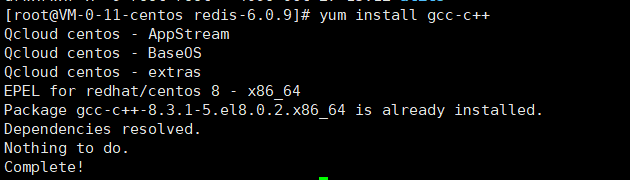
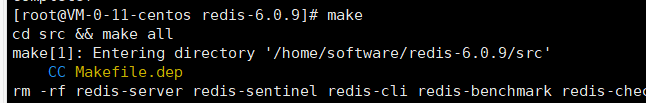
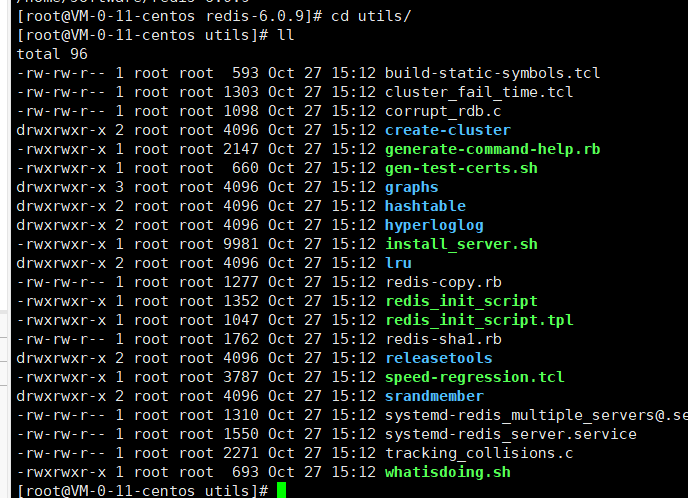
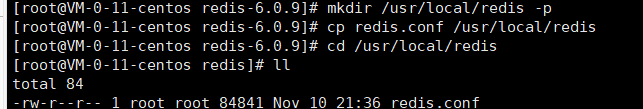
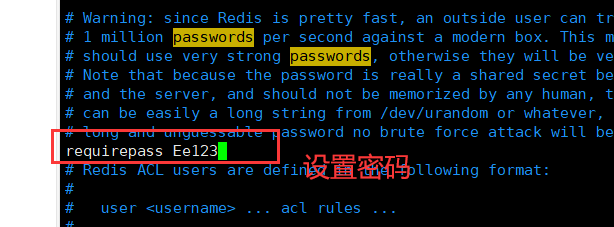
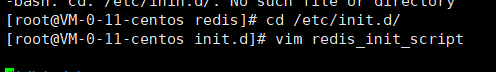
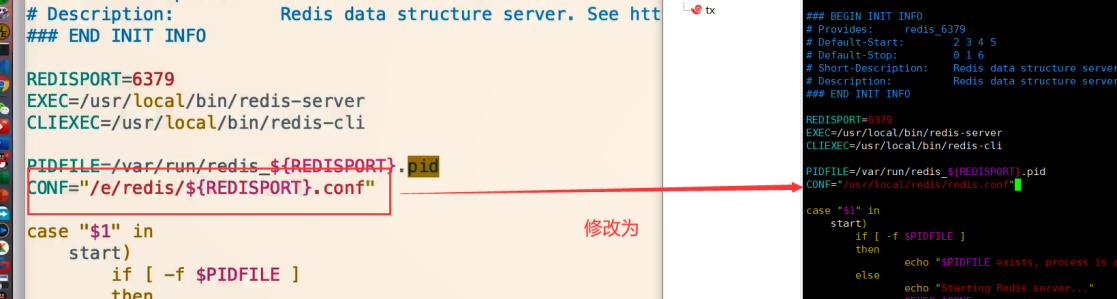
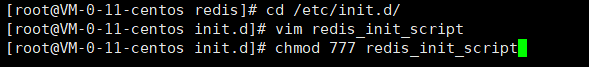
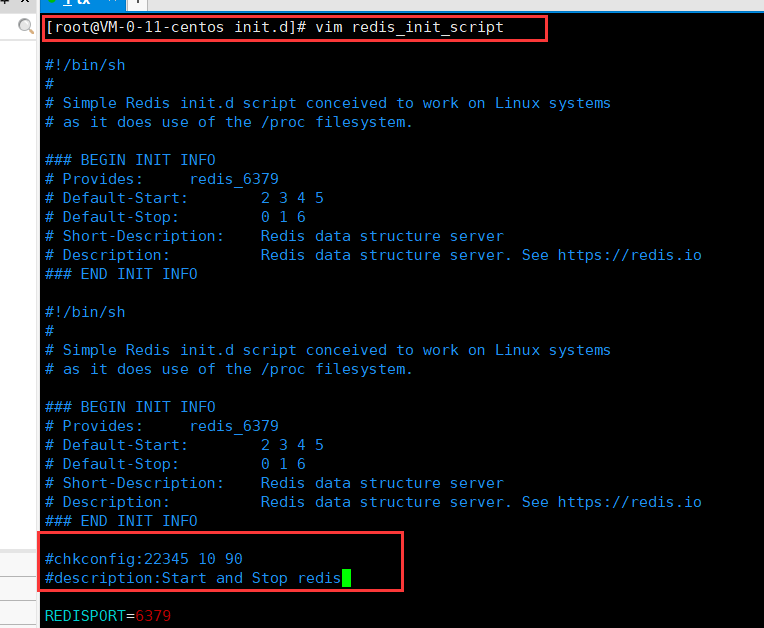
# Redis

## 安装Redis

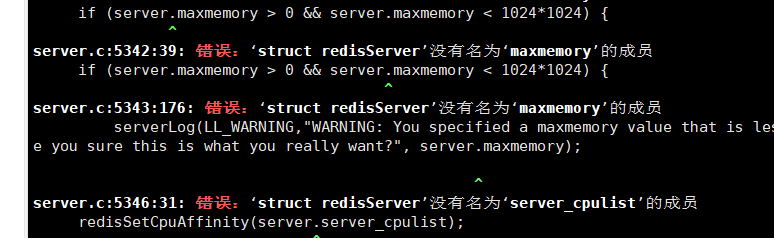
### Linux安装





* 进入到指定文件位置，解压
  + 
* 然后进入redis文件夹
  + 
* 安装redis之前需要安装一个依赖： yum install gcc-c++
  + 
* 然后安装redis
  + 先编辑：make
    - 
  + 然后安装：make install
* 配置redis
  + 先进入redis的utils文件夹下
    - 
  + 然后拷贝redis的启动脚本，拷贝到/etc/init.d/文件夹下:cp redis\_init\_script /etc/init.d/
  + 再把redis文件夹下的redis.conf拷贝到/usr/local/redis 下；然后进入/usr/local/redis 文件夹下 ，修改redis.conf => vim redis.conf
    - 
    - 周到 daemonize 修改no
      * 
      * 
      * requirepass Ee123
* 设置redis启动脚本
  + 
  + 
  + 给脚本运行权限
    - 
  + 运行redis=》 ./redis\_init\_scrip start
    - 
  + 设置redis开机启动：进入redis\_init\_script 添加【#chkconfig:22345 10 90
  + #description:Start and Stop redis
  + 】。填写完之后将此注册到服务器中=》chkconfig redis\_init\_script on
    - 
    - 

### 如果安装错误



1、安装gcc套装：

yum install cpp  
yum install binutils  
yum install glibc  
yum install glibc-kernheaders  
yum install glibc-common  
yum install glibc-devel  
yum install gcc  
yum install make  
2、升级gcc

yum -y install centos-release-scl

yum -y install devtoolset-9-gcc devtoolset-9-gcc-c++ devtoolset-9-binutils

scl enable devtoolset-9 bash

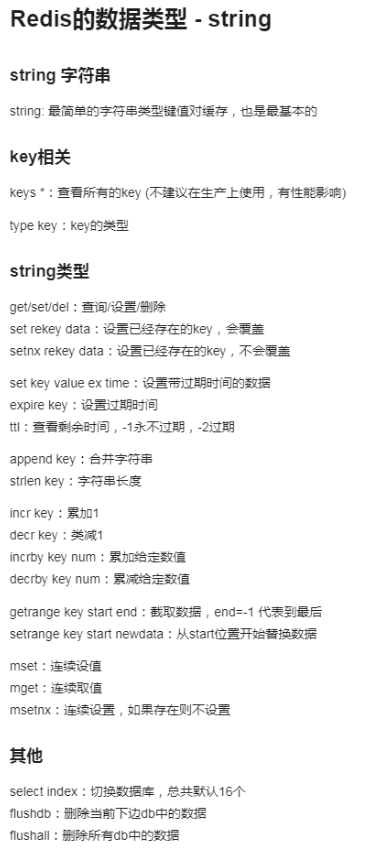
3、设置永久升级：

echo "source /opt/rh/devtoolset-9/enable" >>/etc/profile

## redis基础

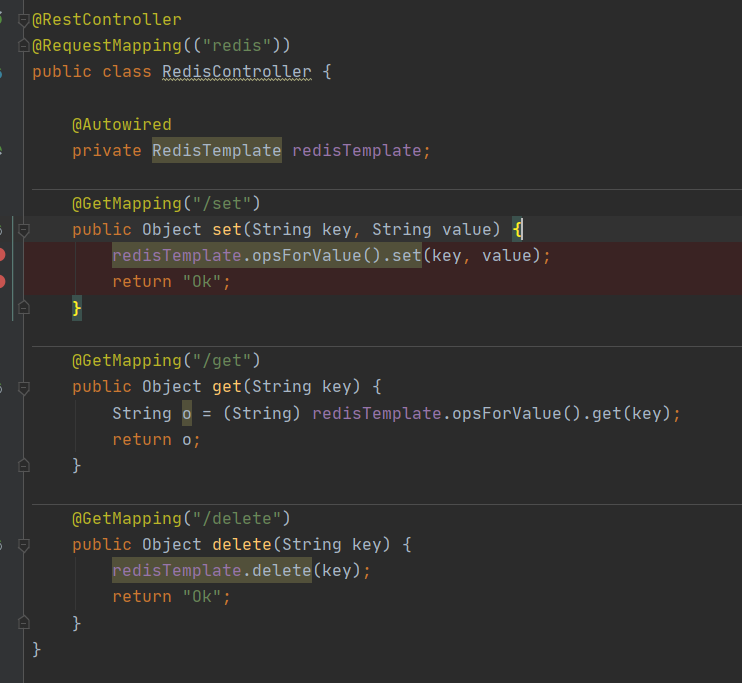


### Redis类型





### Redis 整合springboot



# Redis持久化

Redis数据一般放在内存，但是考虑突然断电等情况。会放在硬盘上面。等服务器重启之后会重新放在内存。

Aof和Rdb

<https://baijiahao.baidu.com/s?id=1654694618189745916&wfr=spider&for=pc>

## Rdb持久化(可接受在一段时间内丢失数据)

### 优缺点

优点：

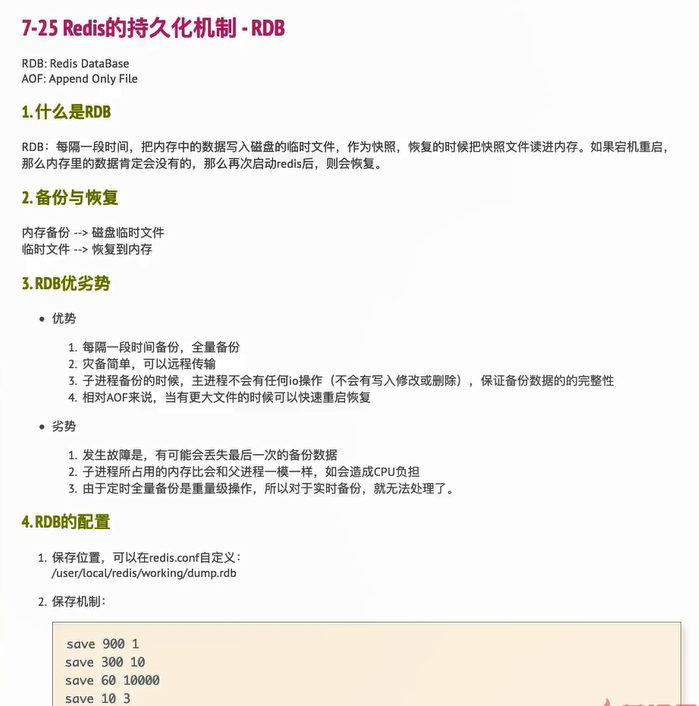
1 适合大规模的数据恢复。

2 如果业务对数据完整性和一致性要求不高，RDB是很好的选择。

缺点：

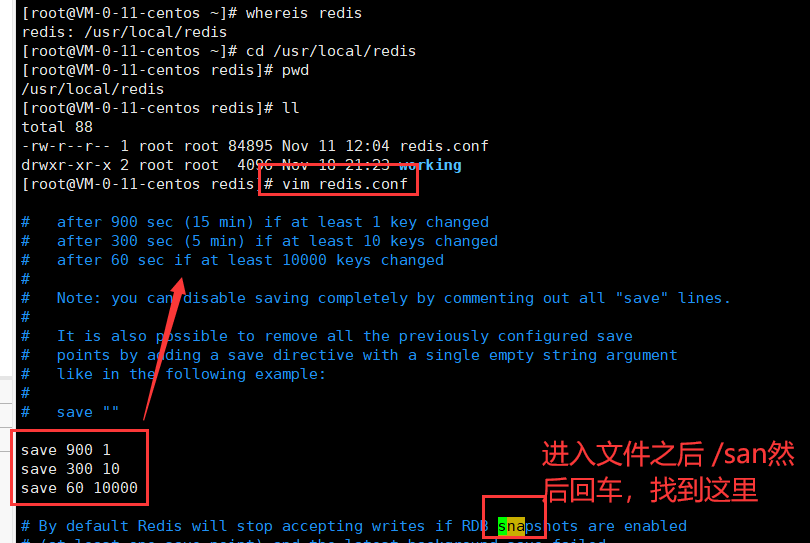
1 数据的完整性和一致性不高，因为RDB可能在最后一次备份时宕机了。

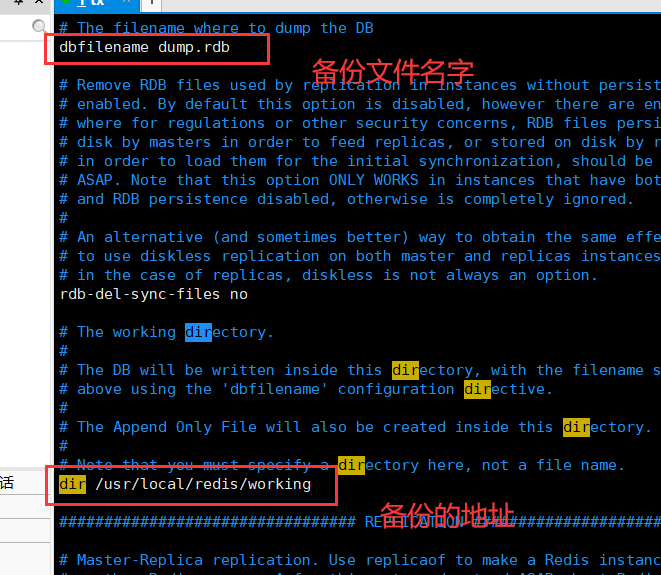
2 备份时占用内存，因为Redis 在备份时会独立创建一个子进程，将数据写入到一个临时文件（此时内存中的数据是原来的两倍哦），最后再将临时文件替换之前的备份文件。所以要考虑到大概两倍的数据膨胀性。

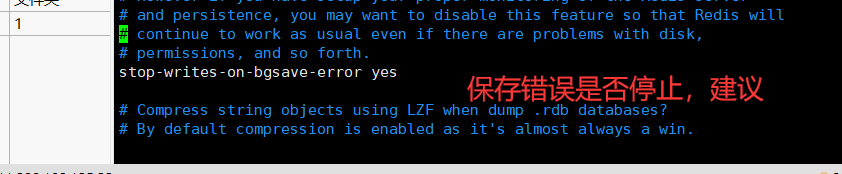


### 举例

下图说明：至少改一个key，900秒备份一次。至少改10个key 300秒备份一次。至少改10000key 60秒备份一次。







## Aof持久化(更高的缓存一致性)

### 优缺点

#### AOF优势：

1. . 该机制可以带来更高的数据安全性，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在效率上是最低的。至于无同步，无需多言，我想大家都能正确的理解它。
2. . 由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过redis-check-aof工具来帮助我们解决数据一致性的问题。
3. . 如果日志过大，Redis可以自动启用rewrite机制。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

#### AOF劣势：

1. . 对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，AOF在运行效率上往往会慢于RDB。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

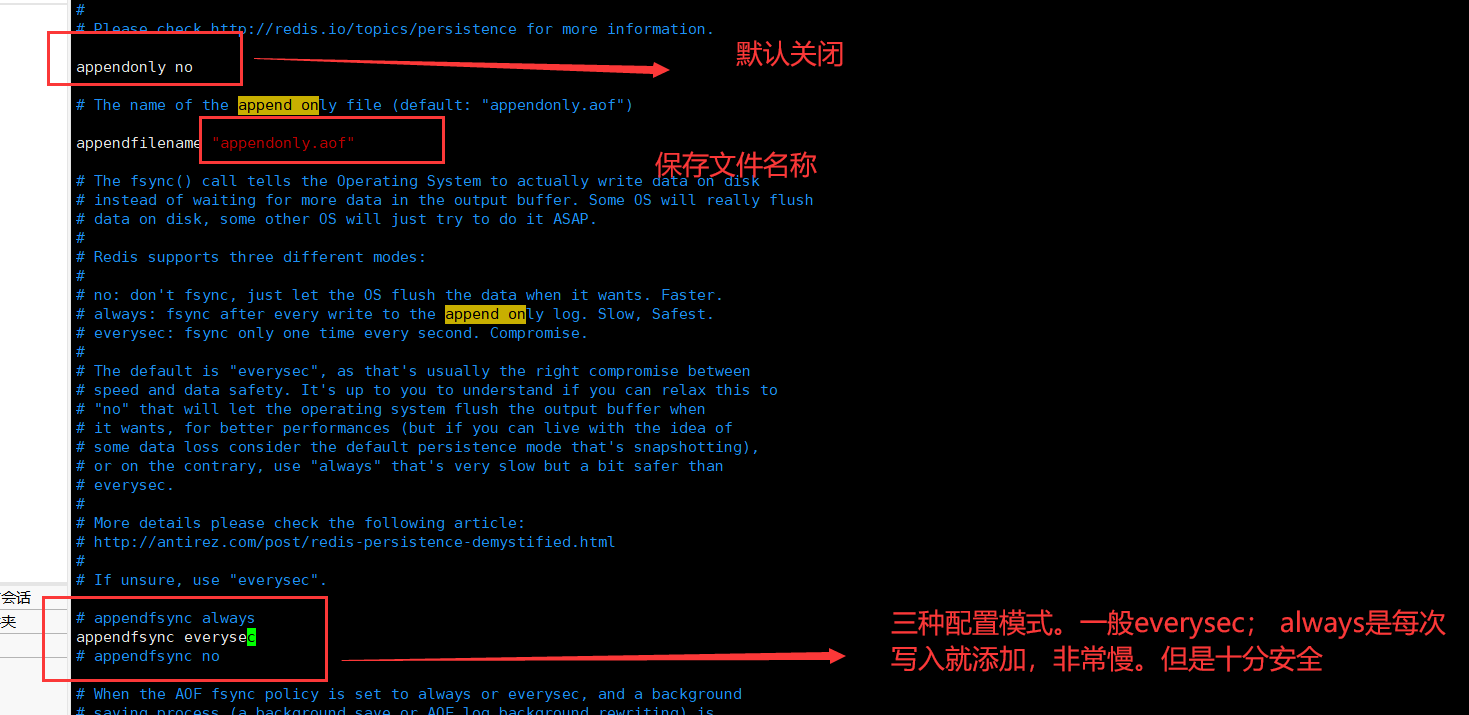
二者选择的标准，就是看系统是愿意牺牲一些性能，换取更高的缓存一致性（aof），还是愿意写操作频繁的时候，不启用备份来换取更高的性能，待手动运行save的时候，再做备份（rdb）。rdb这个就更有些 eventually consistent的意思了。

### 基础配置

appendfsync always：总是写入aof文件，并完成磁盘同步  
appendfsync everysec：每一秒写入aof文件，并完成磁盘同步

appendfsync no：写入aof文件，不等待磁盘同步。

可见，从持久化角度讲，always是最安全的。从效率上讲，no是最快的。而redis默认设置进行了折中，选择了everysec。



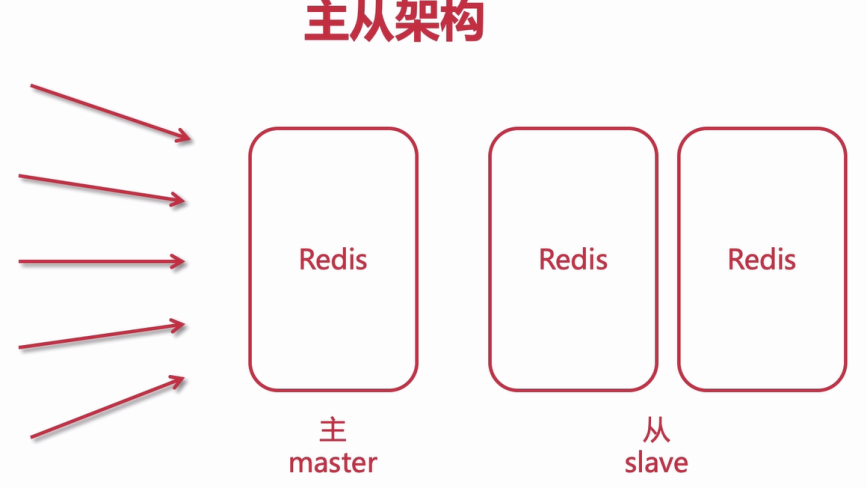
no-appendfsync-on-rewrite：

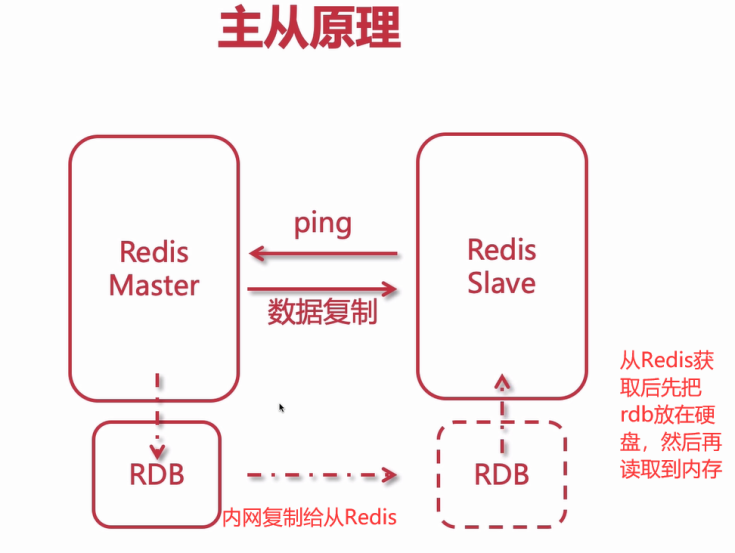
参数设置为no，是最安全的方式，不会丢失数据，但是要忍受阻塞的问题。如果设置为yes呢？这就相当于将appendfsync设置为no，这说明并没有执行磁盘操作，只是写入了缓冲区，因此这样并不会造成阻塞（因为没有竞争磁盘），但是如果这个时候redis挂掉，就会丢失数据。丢失多少数据呢？在linux的操作系统的默认设置下，最多会丢失30s的数据。

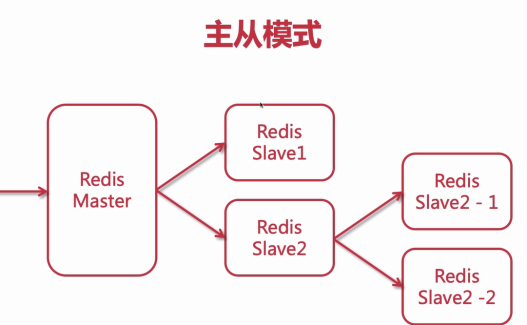
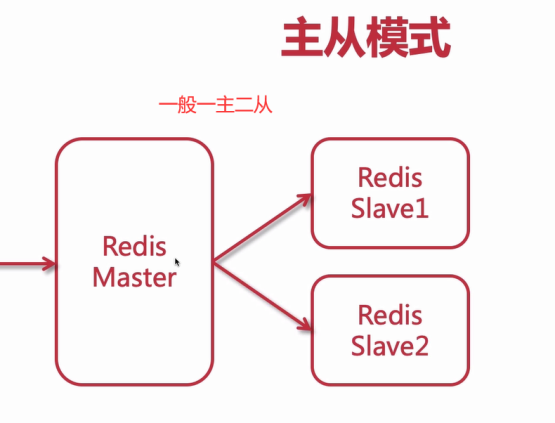
# 主从架构(就是读写分离)

## 基础

从节点负责读，主节点负责写。Master必须持久化，不然突然master断电，重启之后会情况slave的数据



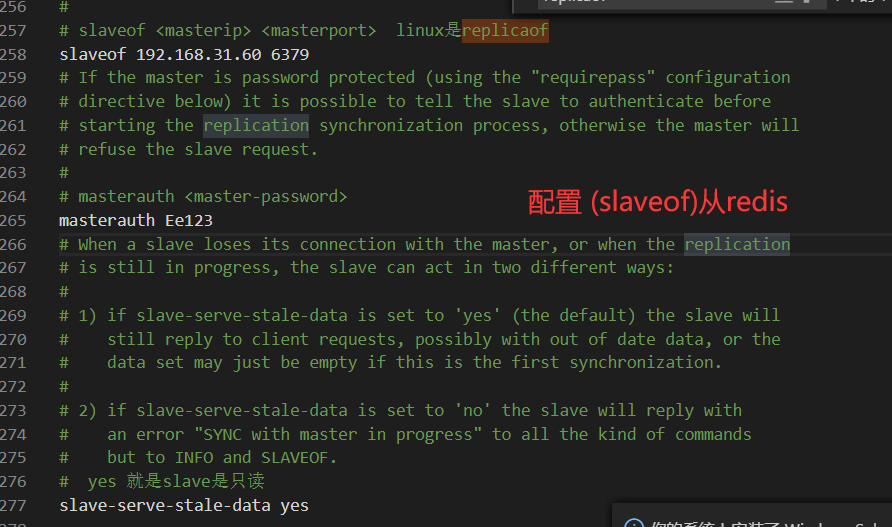


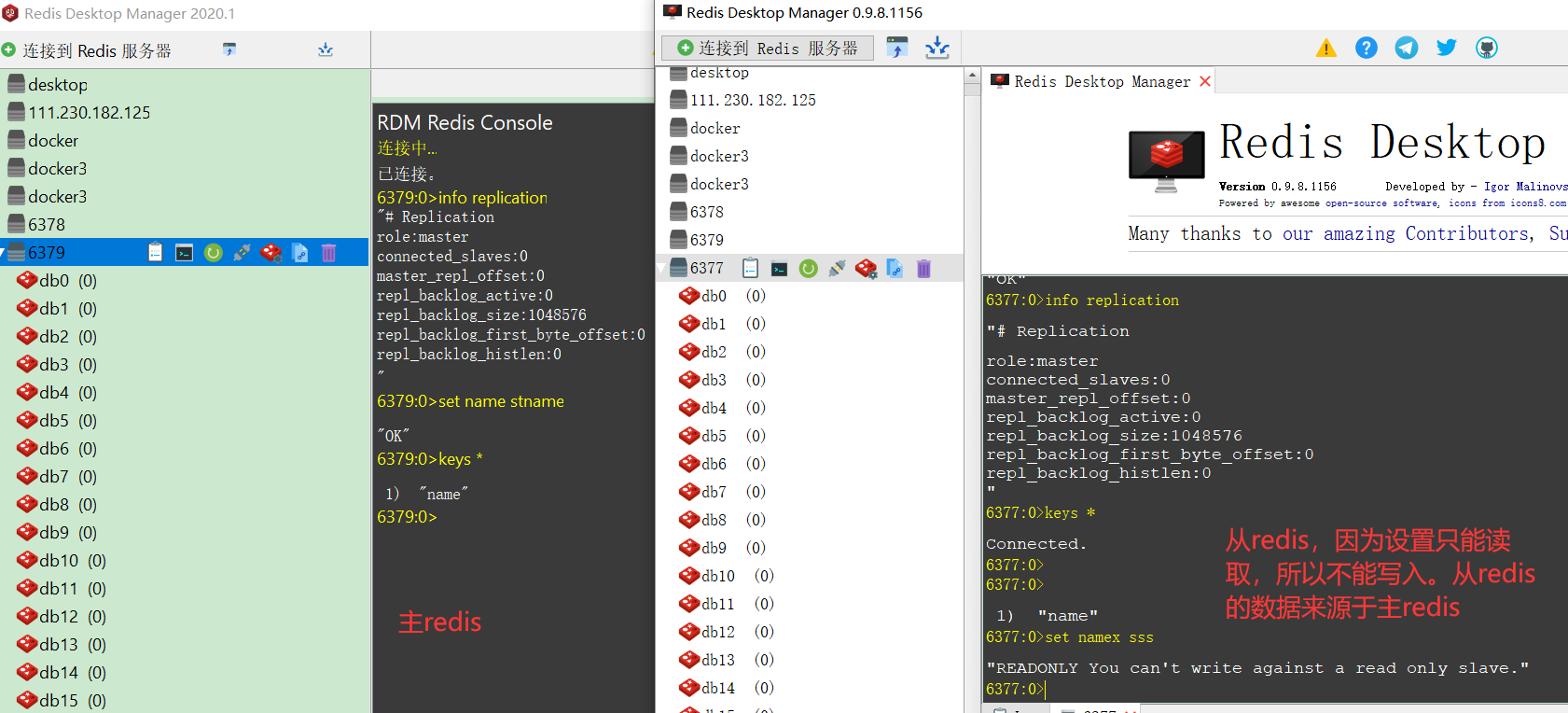


## 实际操作

多台linux系统上搭建redis，此处在自己的windows电脑上准备3台redis端口号分别是：6377、6378、6379。其中将6379作为master

然后进入6377、6378的redis.conf文件中进行配置(windows是redis.windows.conf)









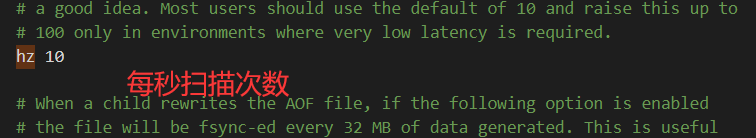
主redis的数据会立即同步到从redis、一旦主redis数据清空，从redis也会被清空

## 无磁盘化复制(目前暂时先不要用于生产环境)

当磁盘复制文件过慢，可以考虑使用

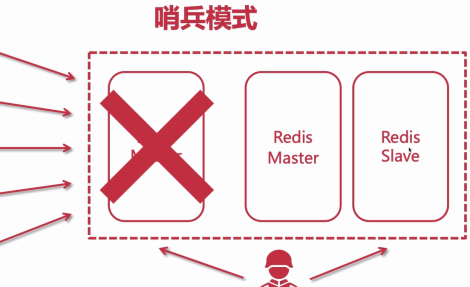
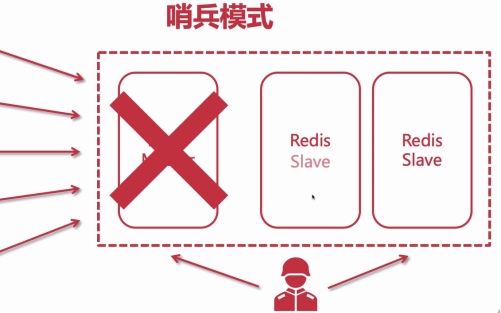


# Redis缓存过期机制

* 主动(定期删除):定时扫描，遇到过期的就删除
  + 
* 被动(惰性删除)：只有过期了，且被请求到才删除；（cup占用少）

# 哨兵模式(Sentinel)

主从模式中，当master挂掉。哨兵会将一个从(slave)redis变为master



关键文件 sentinel.conf;启动命令，redis-sentinel sentinel.conf

|  |
| --- |
| #绑定0.0.0.0解读无法访问的问题  bind 0.0.0.0  #当前Sentinel服务运行的端口  port 8099  #去监视一个名为mymaster的主redis实例，这个主实例的IP地址为本机地址127.0.0.1，端口号为6379，而将这个主实例判断为失效至少需要2个 Sentinel进程的同意，只要同意Sentinel的数量不达标，自动failover就不会执行  sentinel monitor mymaster 127.0.0.1 6379 2  #设置的密码  sentinel auth-pass mymaster Ee123  #指定了Sentinel认为Redis实例已经失效所需的毫秒数。当实例超过该时间没有返回PING，或者直接返回错误，那么Sentinel将这个实例标记为主观下线。只有一个 Sentinel进程将实例标记为主观下线并不一定会引起实例的自动故障迁移：只有在足够数量的Sentinel都将一个实例标记为主观下线之后，实例才会被标记为客观下线，这时自动故障迁移才会执行  sentinel down-after-milliseconds mymaster 10000  #指定了在执行故障转移时，最多可以有多少个从Redis实例在同步新的主实例，在从Redis实例较多的情况下这个数字越小，同步的时间越长，完成故障转移所需的时间就越长  sentinel parallel-syncs mymaster 1  #如果在该时间（ms）内未能完成failover操作，则认为该failover失败  sentinel failover-timeout mymaster 10000 |

## 设置一个demo

* 环境win10
* redis 2.8.2402
* 端口master：6379 ；slave：6378、6377
* Reids未设置密码

此次测试删除redis.windows-service.conf文件

【redis3.2也测试一次，并且没有删除redis.windows.conf，只配置redis.windows-service.con】

### redis.windows.conf

* Master
  + bind 0.0.0.0
  + port 6379
* slaveof1和 slaveof2
  + bind 0.0.0.0
  + port 6377
  + slaveof 127.0.0.1 6379

### sentinel.conf

* Master

|  |
| --- |
| # 这个是Redis6379配置内容，其他文件同理新增然后改一下端口即可，26380 26381  #当前Sentinel服务运行的端口  port 26379  bind 127.0.0.1  # 哨兵监听的主服务器  sentinel monitor mymaster 127.0.0.1 6379 2  # 3s内mymaster无响应，则认为mymaster宕机了  sentinel down-after-milliseconds mymaster 3000  #如果10秒后,mysater仍没启动过来，则启动failover  sentinel failover-timeout mymaster 10000  # 执行故障转移时， 最多有1个从服务器同时对新的主服务器进行同步  sentinel parallel-syncs mymaster 1 |

* Slaveof1

|  |
| --- |
| # 这个是Redis6379配置内容，其他文件同理新增然后改一下端口即可，26380 26381  #当前Sentinel服务运行的端口  port 26378  bind 127.0.0.1  # 哨兵监听的主服务器  sentinel monitor mymaster 127.0.0.1 6379 2  # 3s内mymaster无响应，则认为mymaster宕机了  sentinel down-after-milliseconds mymaster 3000  #如果10秒后,mysater仍没启动过来，则启动failover  sentinel failover-timeout mymaster 10000  # 执行故障转移时， 最多有1个从服务器同时对新的主服务器进行同步  sentinel parallel-syncs mymaster 1 |

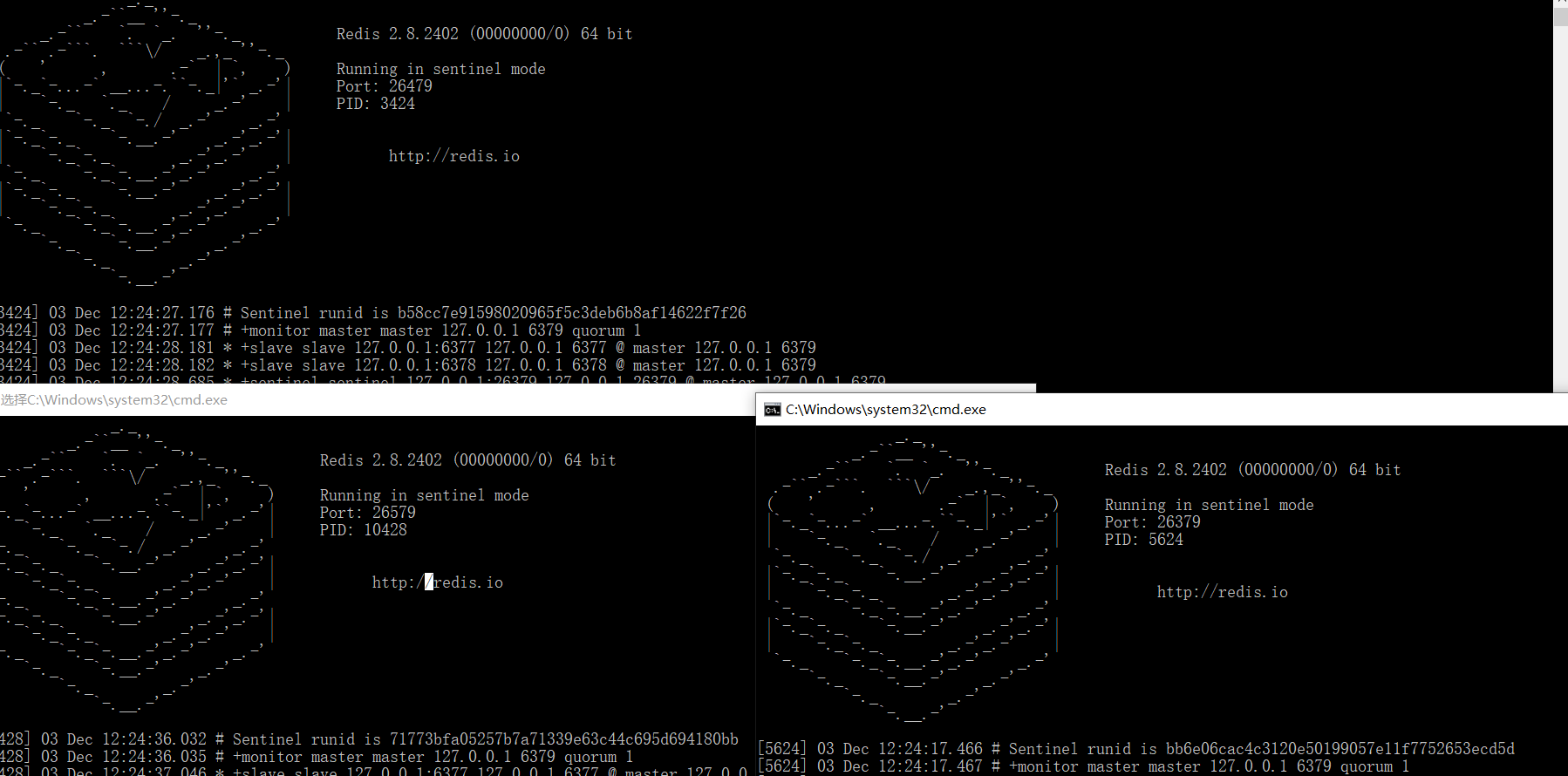
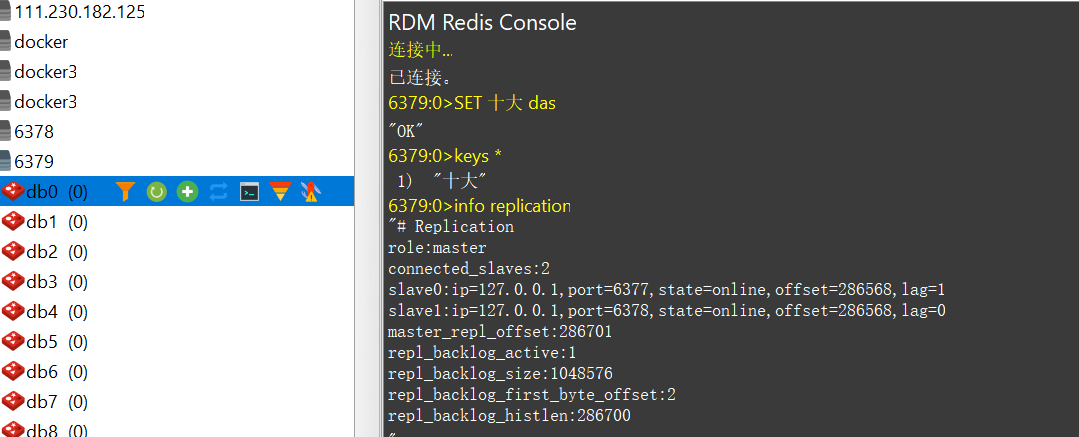
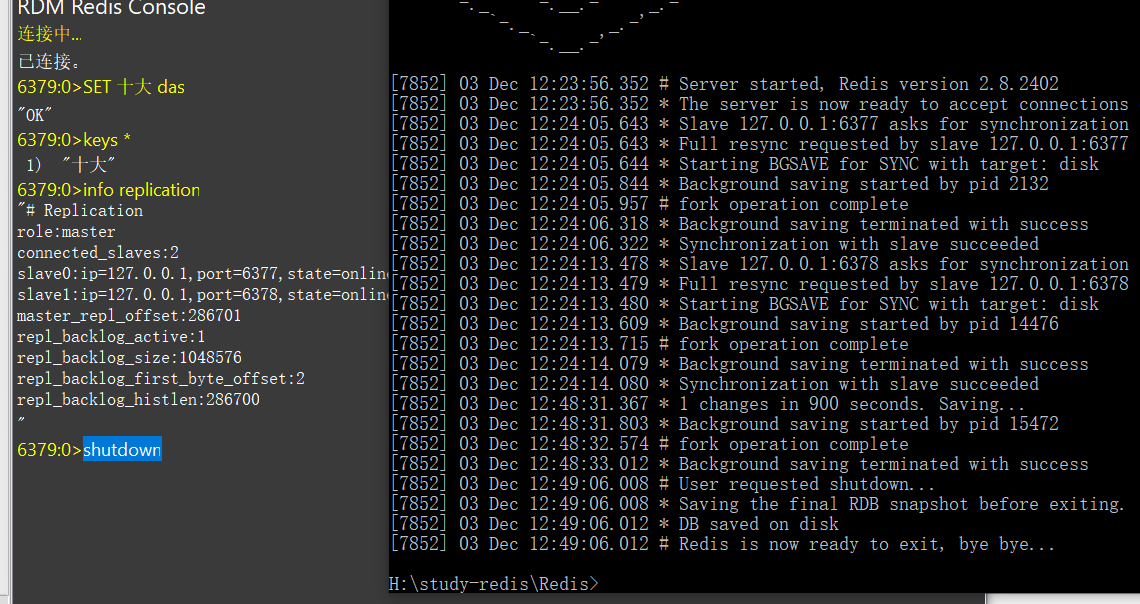
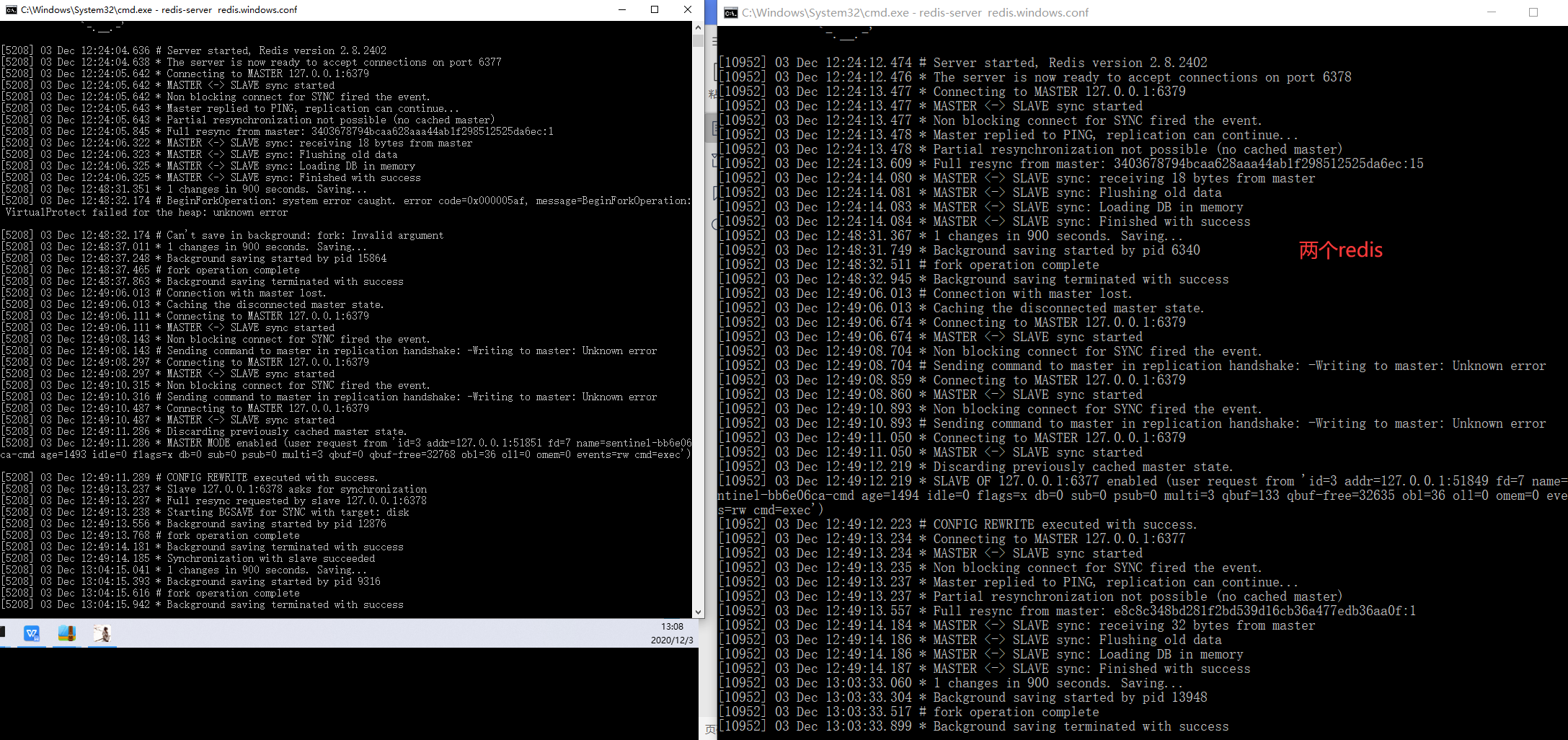
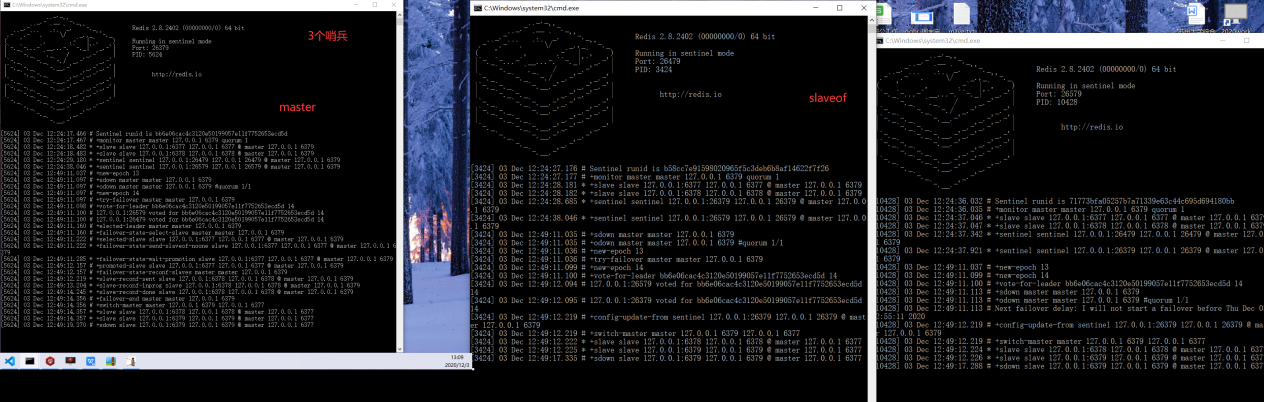
* Salverof2

|  |
| --- |
| # 这个是Redis6379配置内容，其他文件同理新增然后改一下端口即可，26380 26381  #当前Sentinel服务运行的端口  port 26377  bind 127.0.0.1  # 哨兵监听的主服务器  sentinel monitor mymaster 127.0.0.1 6379 2  # 3s内mymaster无响应，则认为mymaster宕机了  sentinel down-after-milliseconds mymaster 3000  #如果10秒后,mysater仍没启动过来，则启动failover  sentinel failover-timeout mymaster 10000  # 执行故障转移时， 最多有1个从服务器同时对新的主服务器进行同步  sentinel parallel-syncs mymaster 1 |

配置启动sentinel.conf文件 start.bat

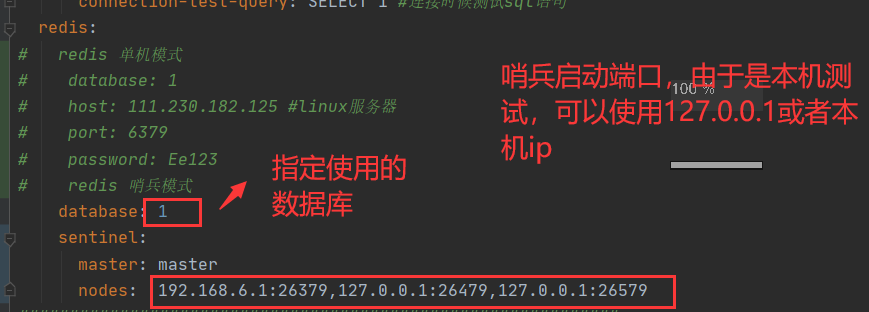
|  |
| --- |
| @echo off  redis-server.exe sentinel.conf --sentinel  @pause |

### 启动

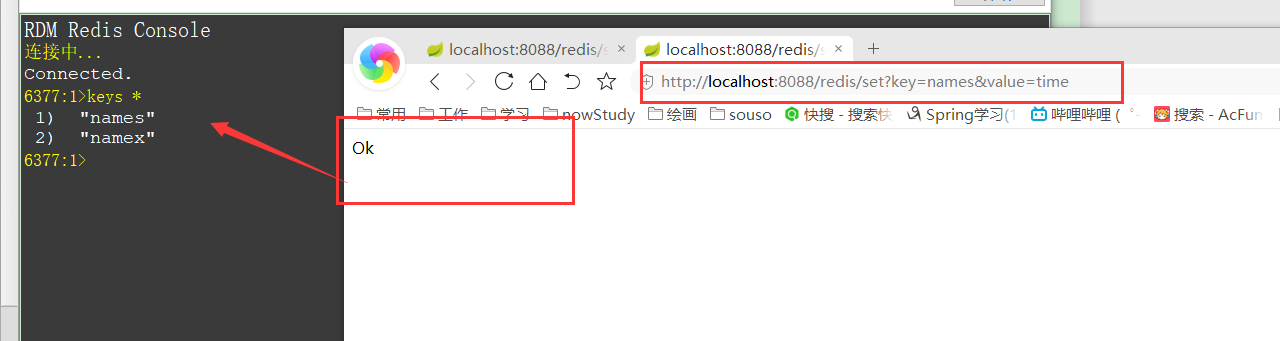
* 先启动所有redis，然后在启动sentinel.conf
  + 
  + 
* 输入：info replication。可以看到当前master一级slave的数量
  + 
* 然后再6379中停止：shutdown
  + 
* 可以看到，此时再其他的redis会进行选举新的master
  + 
  + 

## 在springboot中集成哨兵

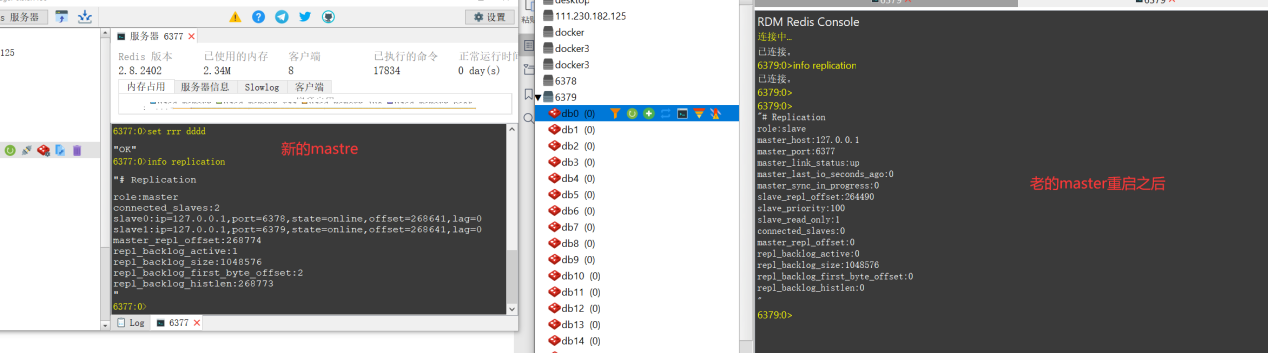
所有配置承接上面。



然后测试：

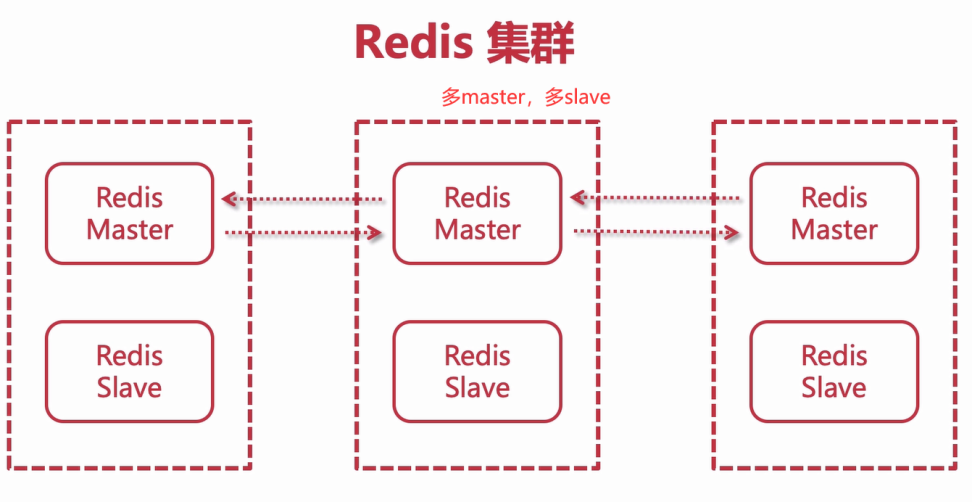


## 注意，每次重新选举新的master之后，重启老的master之后，老的redis不会变成master

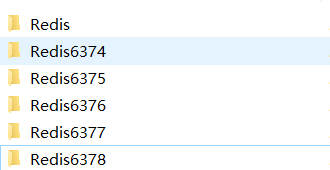


## redis-cluster集群=》windows[redis3.2才有，2.8没有]

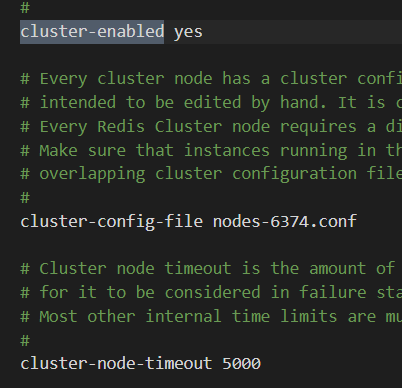
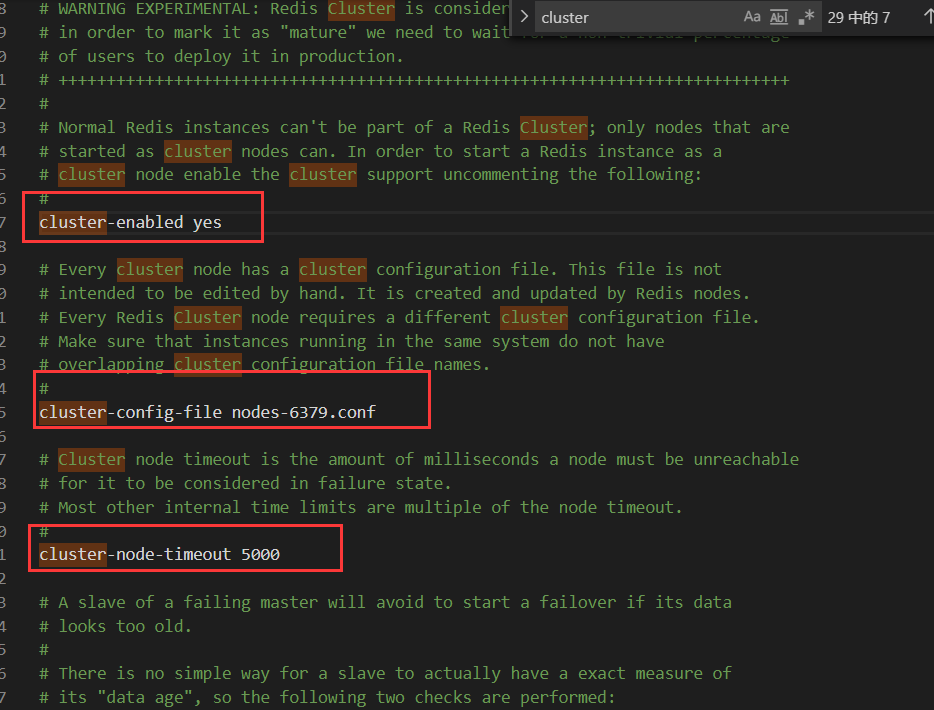
https://blog.csdn.net/hao495430759/article/details/80540407



### 低版本需要Ruby才能集群，高版本不需要；redis5.0.5不用ruby搭建集群。下载redis3.2(linux直接下载最高版本)，一共复制六个

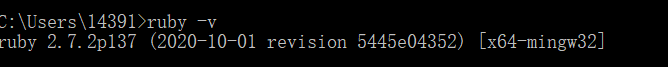
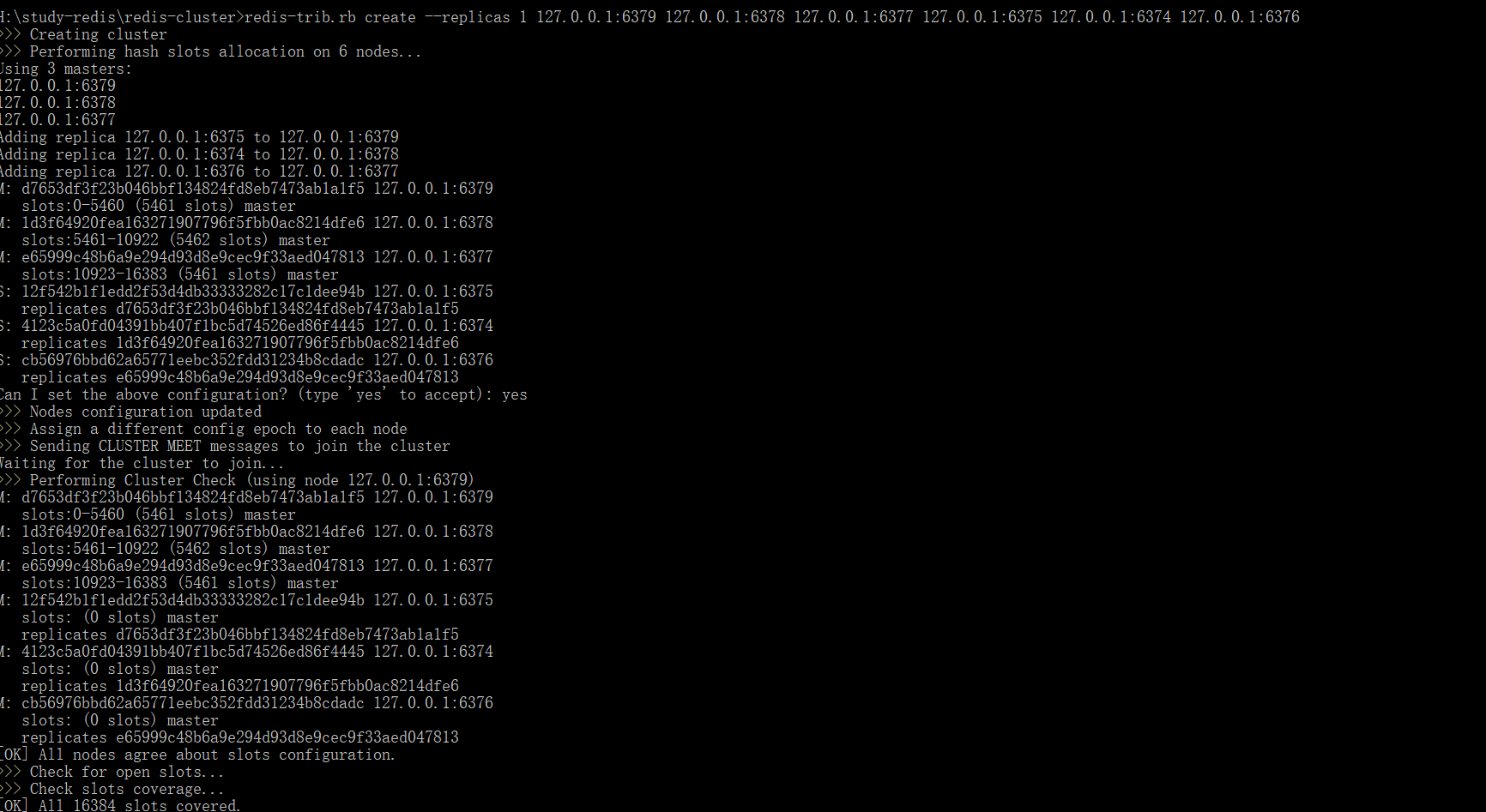


### 然后在每个redis文件夹中打开redis.conf(redis.windows.conf)中找到（不同端口，cluster-config-file的端口不同）

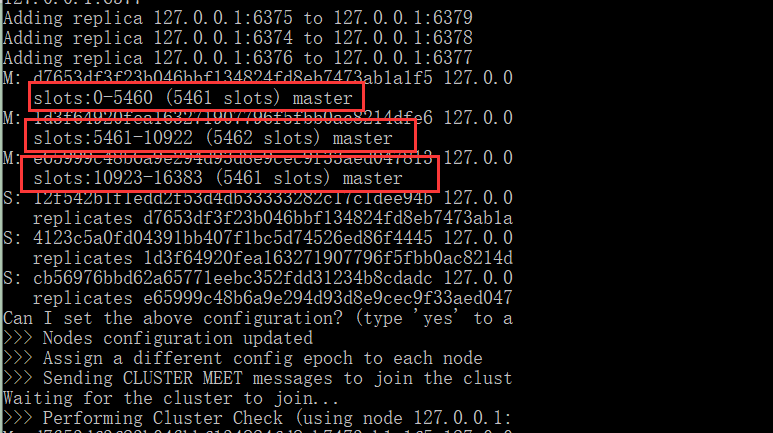


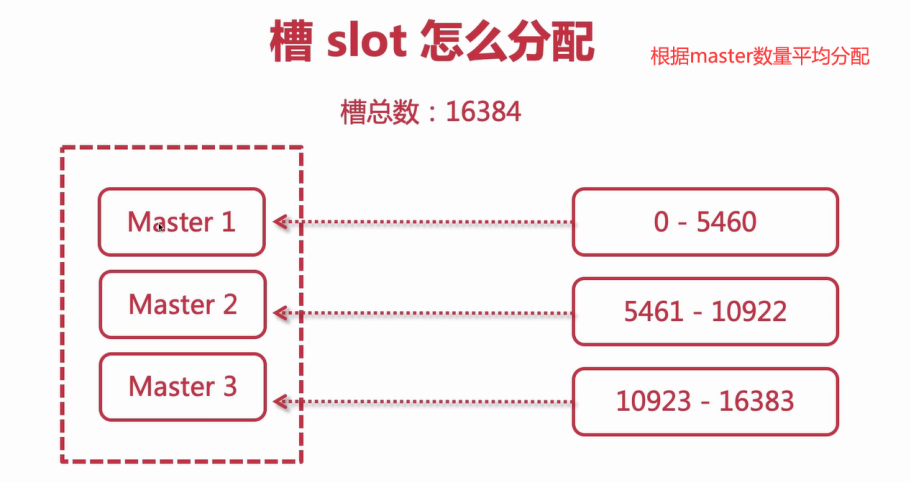
集群之前注意删除原来的数据=》\*.rdb文件和\*.aof文件

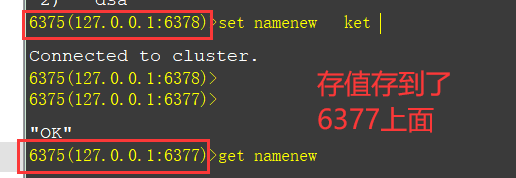
### 注意低版本需要配置ruby和redis-trib.rb文件

* Ruby：[Ruby 程序设计语言官方网站 (ruby-lang.org)](http://www.ruby-lang.org/zh_cn/)
  + 安装过程一直下一步
  + 
* redis-trib.rb：<https://github.com/beebol/redis-trib.rb>
  + 下载之后，放在redis文件一起
  + 
* 然后在当前文件夹下，cmd。然后输入 （–replicas 1 表示为集群中的每个主节点创建一个从节点）后面的端口号是启动redis的端口号：
  + redis-trib.rb create --replicas 1 127.0.0.1:6379 127.0.0.1:6378 127.0.0.1:6377 127.0.0.1:6375 127.0.0.1:6374 127.0.0.1:6376
  + 
  + 途会询问是否打印更多详细信息，输入yes即可，然后redis-trib 就会将这份配置应用到集群当中,让各个节点开始互相通讯
  + 

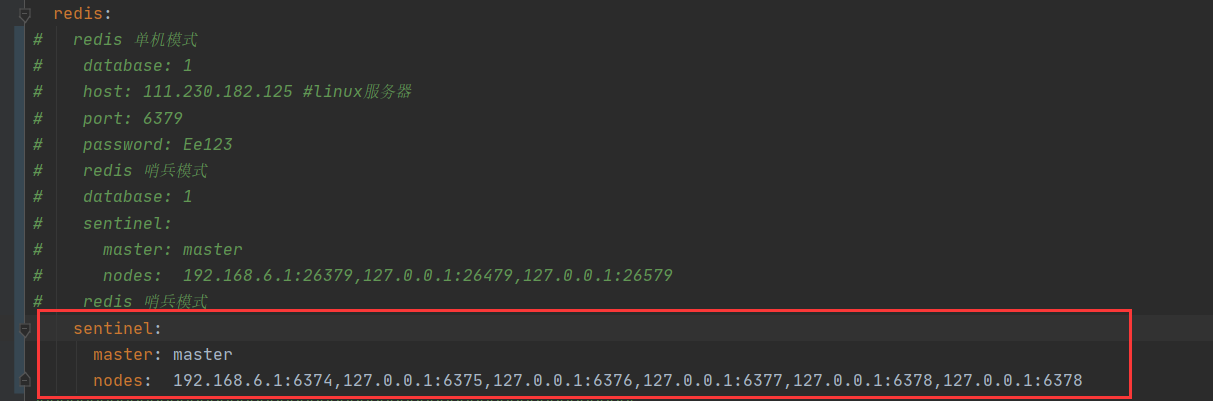
### 槽节点slot



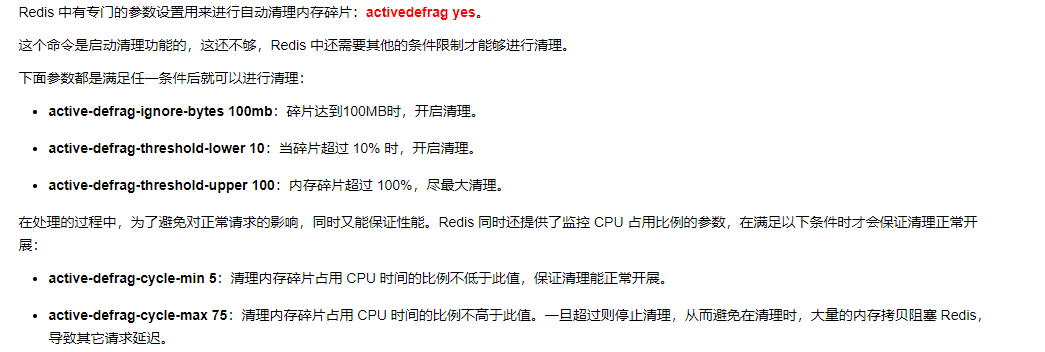




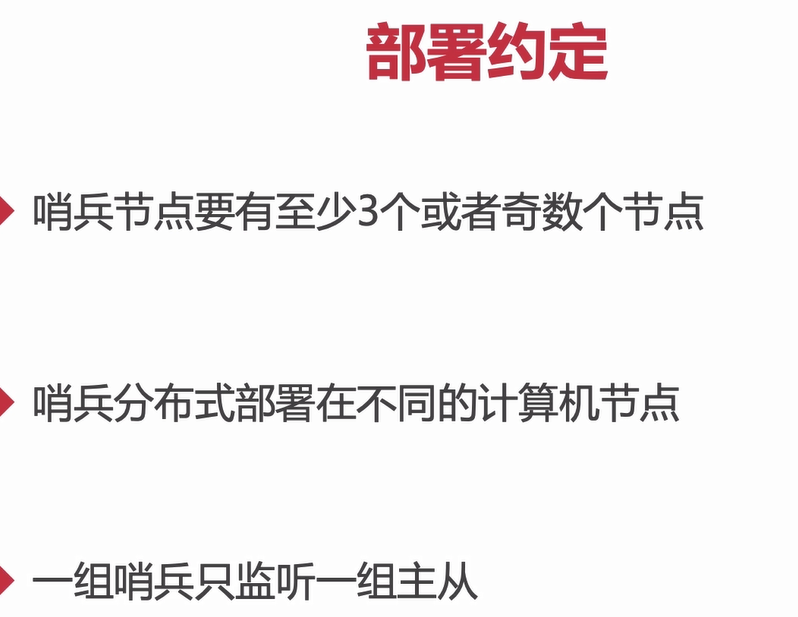
### 集成到springboot



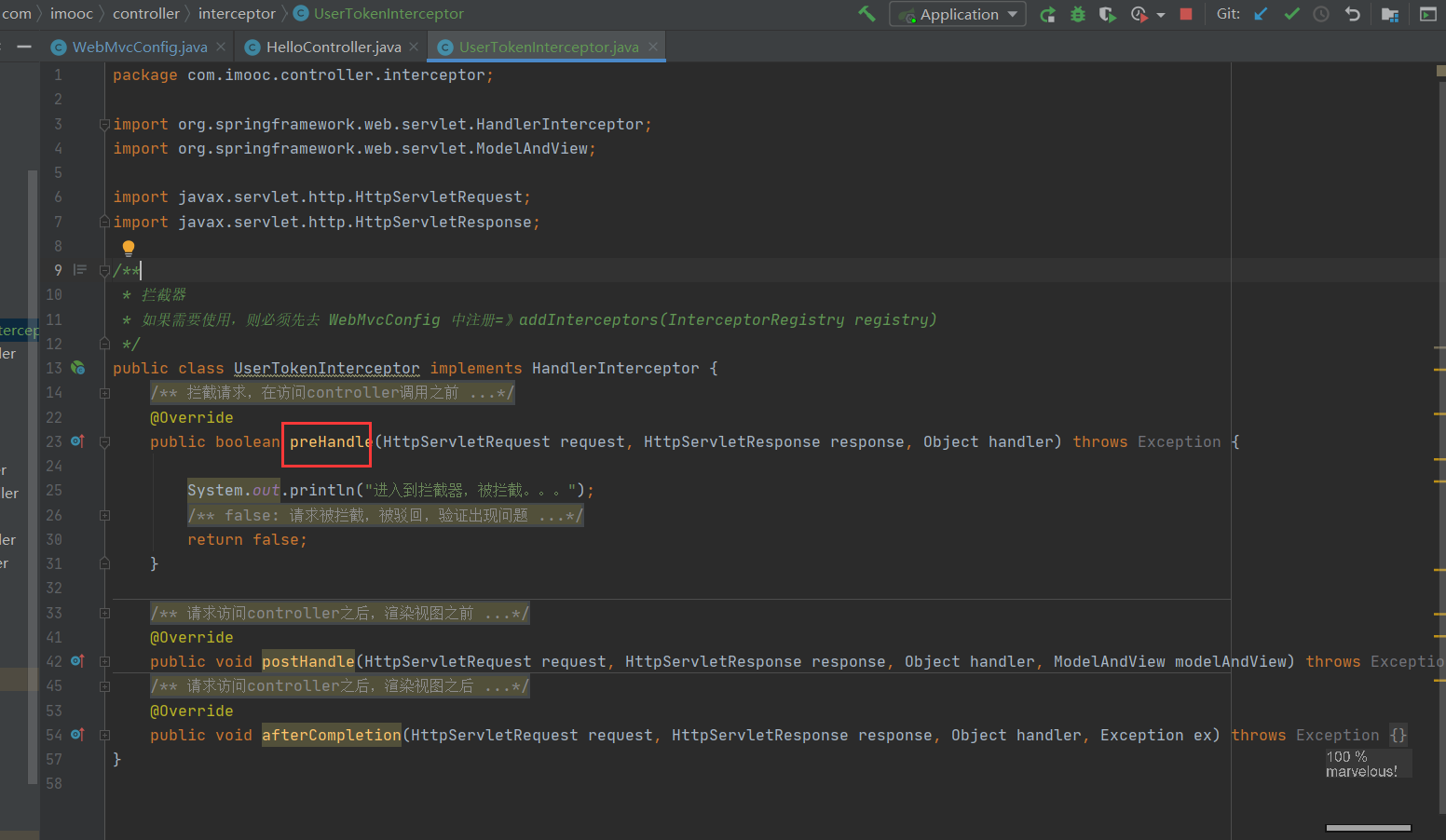
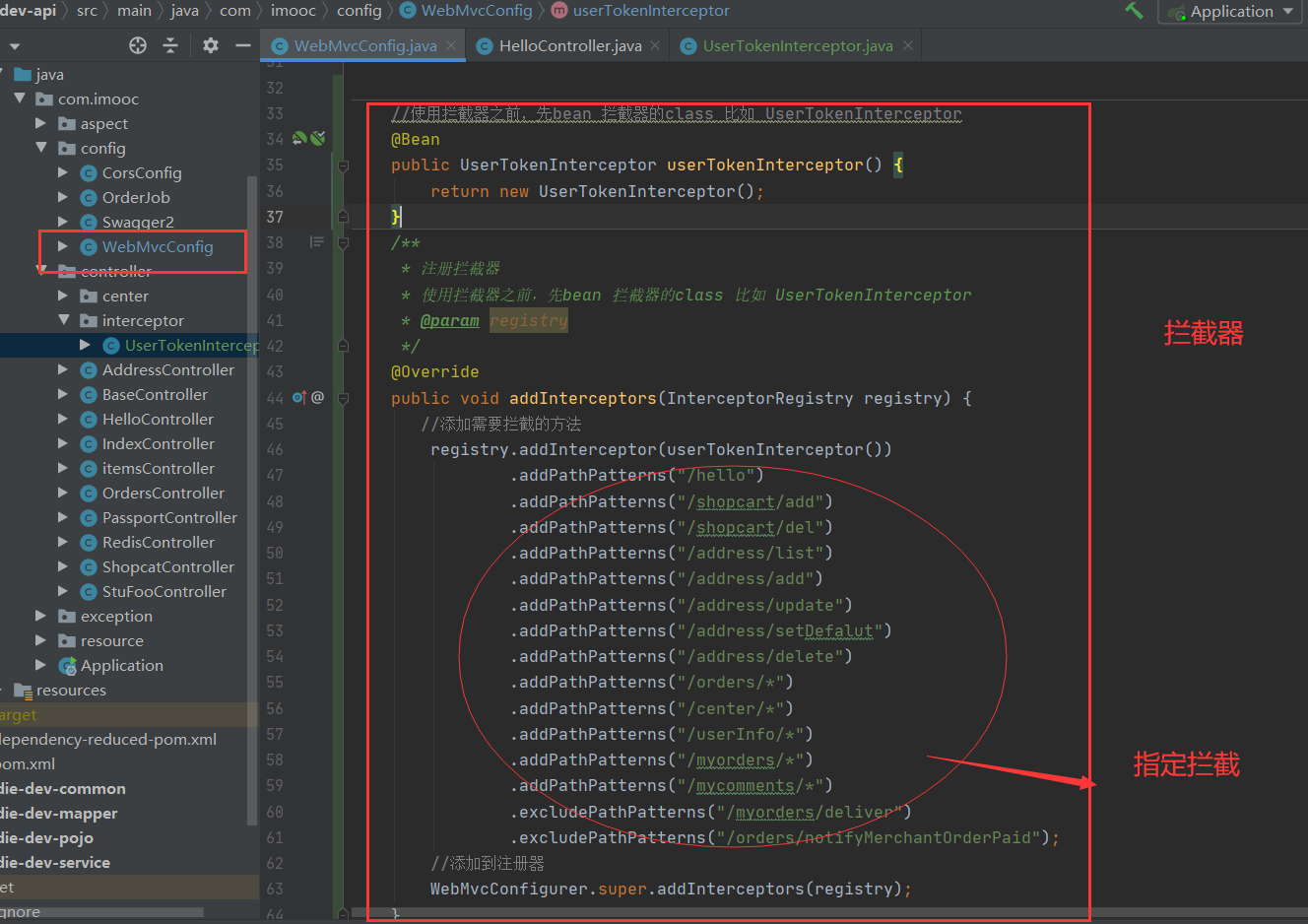
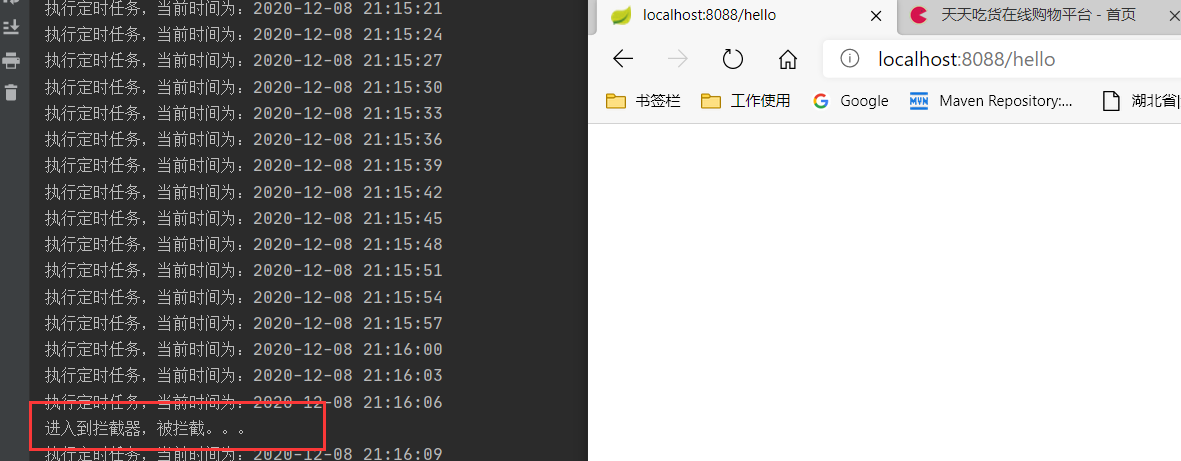
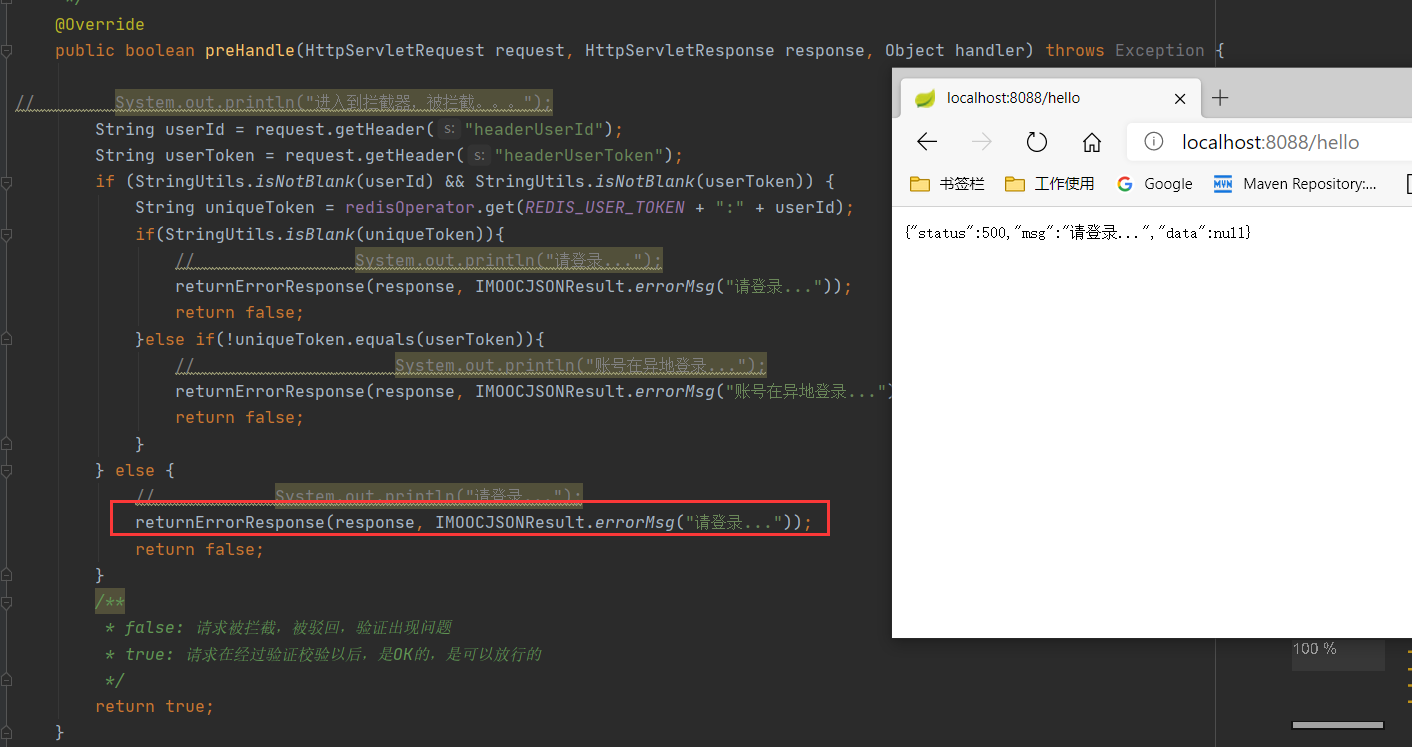
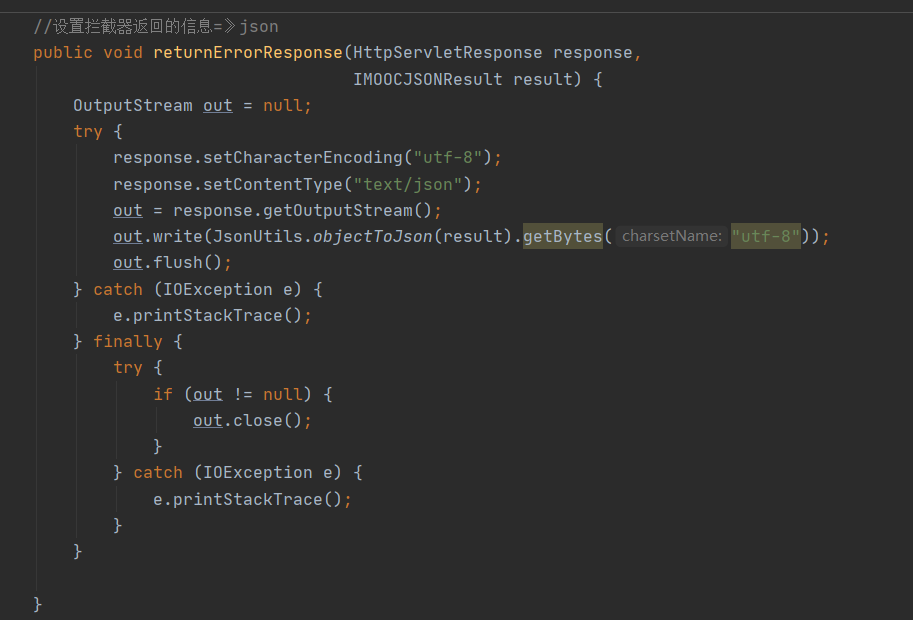
# 清理碎片内存



# 部署约定



# 分布式拦截器

* 首先创建一个控制器，作为拦截器，继承 implements HandlerInterceptor。然后添加preHandle、postHandle、afterCompletion
  + 
* 然后去WebMvcConfig进行注册
  + 
* 最后请求测试
  + 
* 完善拦截器：在UserTokenInterceptor添加一个方法，统一返回拦截器的对象为json
  + 
  + 

|  |
| --- |
| //设置拦截器返回的信息=》json      public void returnErrorResponse(HttpServletResponse response,                                      IMOOCJSONResult result) {          OutputStream out = null;          try {              response.setCharacterEncoding("utf-8");              response.setContentType("text/json");              out = response.getOutputStream();              out.write(JsonUtils.objectToJson(result).getBytes("utf-8"));              out.flush();          } catch (IOException e) {              e.printStackTrace();          } finally {              try {                  if (out != null) {                      out.close();                  }              } catch (IOException e) {                  e.printStackTrace();              }          }      } |

# 看到

