**Lufax网站分布式缓存设计方案**

**更新记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **时间** | **内容** | **作者** |
| v0.1 | 2014-04-16 | 编写初稿 | 郑嘉庆 |
| v0.2 | 2014-04-22 | 添加第三章，增加对应用开发有帮助的redis功能描述 | 郑嘉庆 |
| v0.3 | 2014-04-30 | 第一章增加双写的HA方案；添加第三章，描述session-cache设计；添加第四章，描述list-cache设计 | 郑嘉庆 |
| v0.4 | 2014-05-08 | 增加2.4小结描述部署要求；附录增加sentinel.conf配置说明、jedis客户端sentinel的代码示例； | 郑嘉庆 |

目录

[1.概述 4](#_Toc387396257)

[1.1背景 4](#_Toc387396258)

[1.2Why Redis 4](#_Toc387396259)

[1.3名词解释 5](#_Toc387396260)

[2总体设计 6](#_Toc387396261)

[2.1HA方案选型 6](#_Toc387396262)

[2.1.1方案A：基于Sentinel的主从结构 6](#_Toc387396263)

[2.1.2方案B：基于Tedis的双写结构 9](#_Toc387396264)

[2.1.3方案折衷 10](#_Toc387396265)

[2.2容量规划和扩展性 11](#_Toc387396266)

[2.3持久化策略和数据恢复 13](#_Toc387396267)

[2.4物理结构部署要求 15](#_Toc387396268)

[3 Session-Cache设计 15](#_Toc387396269)

[3.1系统设计 15](#_Toc387396270)

[3.2数据结构 18](#_Toc387396271)

[3.3回退方案 19](#_Toc387396272)

[4 List-Cache设计 19](#_Toc387396273)

[4.1系统设计 19](#_Toc387396274)

[4.2数据结构 20](#_Toc387396275)

[4.3回退方案 21](#_Toc387396276)

[5.Redis核心功能 22](#_Toc387396277)

[5.1过期时间 22](#_Toc387396278)

[5.2事务 22](#_Toc387396279)

[5.3内存上限和剔除策略 23](#_Toc387396280)

[5.3批处理 24](#_Toc387396281)

[5.4运行监控 24](#_Toc387396282)

[5.5Java客户端 24](#_Toc387396283)

[附录 25](#_Toc387396284)

[附1，redis.conf配置参考 25](#_Toc387396285)

[附2，sentinel.conf配置参考 25](#_Toc387396286)

[附3，Jedis-Sentinel客户端配置参考 26](#_Toc387396287)

[附4，基准测试数据 29](#_Toc387396288)

# 1.概述

## 1.1背景

从2014年初开始，lufax.com的访问量呈快速上升趋势（2月份日UV约2万，4月初日UV约6万）。在大访问量下，现有的系统架构逐渐暴露了性能上的瓶颈，比如session的更新，列表页的查询等等。从提升系统性考虑， 准备在Q2引入分布式缓存，初期先供session和列表页查询使用。本方案将采用redis作为缓存服务。

## 1.2Why Redis

目前市面上比较热门的缓存主要有memcache和redis两种，这里选择redis主要考虑它提供了丰富的数据结构支持。从发展的角度看，redis除了单纯的string结构，还可以支持计数器、分布式队列、发布/订阅、集合的交并差集计算等应用场景。下面是关于memcache和redis各方面特性的比较，供参考：

|  |  |  |
| --- | --- | --- |
|  | Memcache | Redis |
| 性能 | 很好 | 也很好 |
| 数据类型 | Strings | Strings、lists、sets、sorted sets、hashes |
| Key过期时间 | 支持 | 支持 |
| 单个value最大值 | 1MB | 1GB |
| 事务 | 仅支持单个key的CAS | 支持，基于乐观锁 |
| 主从复制 | 不支持 | 支持 |
| 持久化 | 不支持 | RDB快照dump、AOF追加日志 两种方式 |
| 内存达到上限 | LRU踢出冷数据 | LRU踢出冷数据 |
| 线程模型 | 多线程 | 单线程 |
| 通信方式 | Tcp | Tcp |
| 网络库 | 使用libevent，较庞大 | 自己实现的轻量级网络库 |
| 扩容方案 | 客户端sharding | 服务端sharding，但还没有正式release；业内常用的是客户端sharding，或者基于twemproxy的中间层sharding |

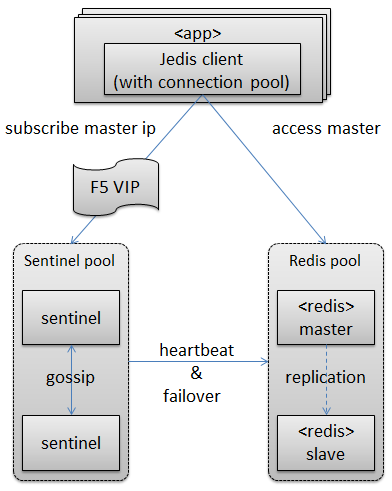
## 1.3名词解释

|  |  |
| --- | --- |
| 名词 | 解释 |
| Redis | REmote Directory Service，一个分布式KV服务，本方案的核心 |
| Sentinel | Redis官方提供的failover方案，支持监控、报警、自动故障切换 |
| Jedis | 一个Java版的Redis客户端，提供了连接池、Sentinel、Sharding等功能支持 |
| F5 | 一种硬件负载均衡，本方案主要利用其vip提供的loadbalance |
| Tedis | 淘宝开源的Redis客户端，与其它客户端的最大区别是它是通过双写方式保证可用性 |

# 2总体设计

## 2.1HA方案选型

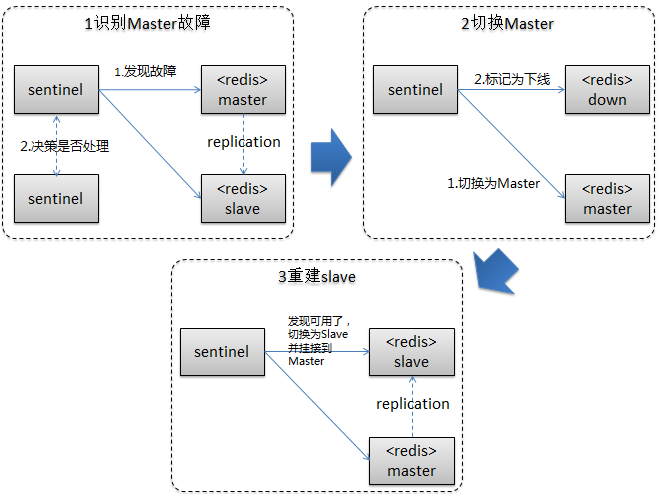
### 2.1.1方案A：基于Sentinel的主从结构



（图：系统总体结构）

如上图所示，整个方案涉及app、redis、sentinel三方，其职责和依赖关系如下：

* Redis为单Master单Slave结构， Master对外提供读写服务，Slave只作为热备。Master/Slave均开启RDB持久化策略；
* 由Sentinel提供failover机制，Sentinel通过秒级心跳监控所有Redis实例运行状态，当某实例故障时就自动执行failover，切换Master； （Sentinel是一个运行在特殊模式下的Redis实例）
* 由于failover会自动切换Master/Slave，所以app不能直接写死Master地址，而是通过订阅Sentinel拿到最新的Master地址，再根据地址访问具体的Redis。Sentinel自身为对等状态集群，通过F5 vip的方式保证Sentinel自身的failover；



（图：故障切换过程）

如上图所示，由sentinel控制的failover的过程可以分为三步：

第一步，Sentinel实例之间通过gossip协议同步彼此状态， 每个Sentinel实例对每个Redis都保持着每秒一次的心跳检测。当某个sentinel实例发现Master故障，会先请求其余Sentinel进行投票表决，如果过半的Sentinel同意执行failover，则Sentinel进入failover处理过程；

第二步，Sentinel通过info指令将原本的Slave动态调整为Master，将原有的Master从服务列表中删除，然后将变更信息推送到所有订阅的客户端（这里主要是各个app）。至此一次自动failover切换过程就结束了；

第三步，如果后续有管理员将故障的Master重新启动，Sentinel自动将其标示为Slave，并挂接到Master上；

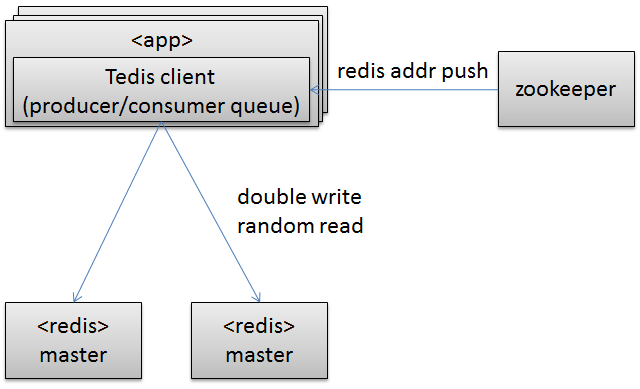
对于app来说，主要由Jedis客户端监听Sentinel的Master变更消息，当发生Master切换时，Jedis自动关闭现有连接池，并重新启动以新Master为host/port的连接池。当然，从Master故障到app感知到变更期间，访问redis的请求将抛出异常。这个过程的耗时，主要由如下6个子步骤累加决定：

|  |  |  |
| --- | --- | --- |
| 1 | Sentinel识别到Master故障 | 基于每秒一次的心跳，可以认为在1秒以内 |
| 2 | Sentinel第一次识别到故障后，将等待一段时间，如果在这期间Master没有任何正常的应答，会将Master标记为SDown状态（主观失败） | 等待的时间由sentinel.conf配置中的down-after-milliseconds参数决定，比如设置为5秒左右 |
| 3 | Sentinel之间投票表决的时间，当过半的Sentinel表决为故障，会将Master标记为ODown（客观失败） | 从本地测试看，两台Sentinel的表决时间在1秒内 |
| 4 | 将Slave切换为Master | 从本地测试看，约1+秒 |
| 5 | 通知客户端 | 基于subscribe方式，可以认为是准实时 |
| 6 | 将原有Master标记为Slave，并下线 | 从本地测试看，约5秒，但这个过程不是串行操作，不影响客户端感应 |

由此可以看出，对响应时间影响最大，且人为可控的是down-after-milliseconds的配置。（这个值设置太短可能会导致误判，比如短暂的网络抖动影响Sentinel接收正确的响应）

从本地测试看，如果配置为2秒，app通常在4+秒内完成切换，如果配置为5秒，app通常在6+秒完成切换。

### 2.1.2方案B：基于Tedis的双写结构



（图：系统总体结构）

该方案主要依赖淘宝的开源客户端Tedis，由Tedis提供双写随机读。其中两个Redis实例均作为Master，彼此之间互不通讯。相关组件的职责和依赖关系为：

* Redis实例均为Master，存储相同的数据；
* 应用系统通过Tedis访问Redis实例。对于写请求Tedis同时写两到两个Redis，对于读请求将随机挑选一个Redis读取； （在连接池方面，Tedis不像Jedis一样使用apache-commons连接池，而是自己实现了一个简单的生产者/消费者队列）
* zookeeper统一维护当前可用的Redis实例信息，当实例信息变更时主动推送给应用系统的Tedis客户端；（从开源版Tedis的设计看，这里zookeeper的定位有点鸡肋，仅用于人工地推送Redis节点信息，可以考虑去掉）

双写方案的故障切换由Tedis统一控制，其处理过程为：

* 如果对某实例写入/读取时异常，则进行2次重试。
* 如果重试结果任然异常，Tedis将该Redis实例标记为故障，后续的读写请求都不会走这个实例。同时Tedis启动一个后台监控线程，该线程每20秒对故障实例做一次检测（PING操作），如果故障实例能正常响应，则自动将该实例标记为可用。
* 应用系统之间的Tedis不相互通讯，即如果某实例故障，每台应用都会执行上述的故障切换逻辑。

### 2.1.3方案折衷

综合上述两个HA方案，其优缺点对比如下：

|  |  |  |
| --- | --- | --- |
| 特点 | 主从方案 | 双写方案 |
| 故障切换自动化 | 全自动处理 | 全自动处理 |
| 故障切换对应用的影响 | 切换期间存在若干秒的空档期 （具体依赖配置），期间应用系统无法正常访问Redis | 不影响，故障期间仍然有一个节点保持可用 |
| 恢复节点对应用的影响 | 不影响，sentinel自动将恢复的节点挂接到当前master上 | 恢复节点的数据无法和当前master节点保持同步，落到该节点的查询会读出旧数据 |
| 应用系统配置复杂度 | 仅配置Sentinel的VIP地址 | 如果不使用zk，需配置两台Master实例地址；使用zk则仅配置zk地址 |
| 开源技术支持 | 需要使用Redis 2.8版本以上；官方方案，相对较稳定； | 淘宝的开源方案，据了解主要应用在聚划算的某些业务场景。其开源版本已经近两年未更新，且没有完善的设计/使用文档；（代码不算复杂，可以自己修改和维护） |

从上述对比可以看出，两个方案的利弊主要体现在故障切换/节点恢复过程中对应用的影响，这里列举未来可能的应用场景进行对比分析：

|  |  |  |
| --- | --- | --- |
| 应用场景 | 主从方案 | 双写方案 |
| Session缓存 | 故障切换期间影响session check，页面上将跳转到登陆页面（用户重新登陆后仍跳到登陆页面）； | 节点恢复之后无法同步两节点的session状态，当前在线用户均会被跳到登陆页面，重新登陆之后恢复正常； |
| 列表页缓存 | 故障切换期间查询的列表内容为空、统计项为0； | 节点恢复之后由定时程序同时更新两节点状态，可保证数据同步； |
| 分布式锁 | 故障期间锁竞争失败； | 节点恢复之后无法同步两节点的锁状态，会导致锁争抢异常； |
| 其它数据结构（比如set集合、list列表等） | 故障期间无法操作； | 节点恢复之后无法同步两节点的集合内部数据，会导致部分数据丢失； |

综上所述，考虑双写方案难以保证数据的一致性，而主从方案若干秒的切换时间对可用性的影响还能接受，这里选择主从方案。

另外，如果整个Redis故障，可以在客户端设计一套容错策略，由具体业务场景指定如何应对，比如是直接抛错，还是改为读写DB。客户端容错策略不在本期实现，后续添加。

## 2.2容量规划和扩展性



（T\_SESSION表结构）

从存储空间看，根据现有数据库中T\_SESSION表的字段名称和字段类型统计，一笔session记录转成字符串之后最大占用252byte存储空间。算上redis自身数据结构的开销，一笔session的字符串记录存储到redis的内存中不会大于300byte，即按300byte计算。当前lufax网站每日uv为6+万，如果将session保存一天总共需要：

60000UV \* 300byte = 18000byte = 17.57MB

如果按照20倍增长估算，总共需要：

17.57MB \* 20 = 351.56MB

如果按照100倍增长估算，总共需要：

17.57MB \* 100 = 1.75GB

对于其它缓存需求，比如列表页缓存，从数据量和数据大小上都没达到session的量级，可以忽略不计。

从访问量看，目前网站PV约近100万，按高峰时段8小时算，均摊下来为

1000000PV / 8小时 / 60分/60秒 = 34.7 TPS

如果按照20倍增长估算，为

34.7TPS \* 20 = 694TPS

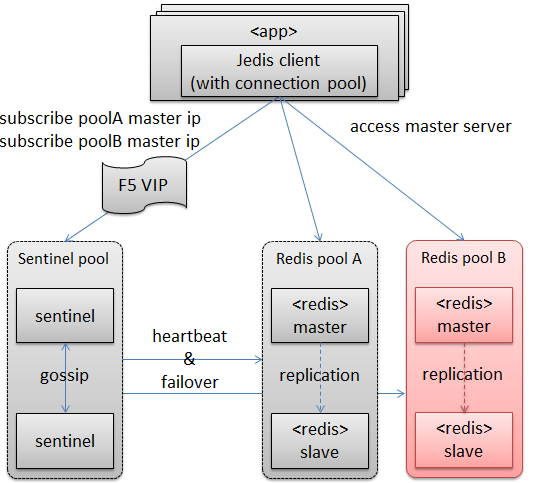
如果按照100倍增长估算，为

34.7TPS \* 100 = 3470 TPS

而从本地开发机的测试结果看，Redis单实例可以支持50000TPS的set/get操作。

从连接数看，redis官方宣称一个redis支持10k连接（未测试过）。多个应用同时连接不会有问题。

综上考虑，在内存空间和访问量上，单机Master足够支撑所有请求。如果发生意外的爆发式增长，或者需要按照功能划分独立Redis实例，可以通过对Redis做垂直拆分来分摊压力。垂直拆分不会改变现有整体结构，只增加一组Master/Slave即可，如下图所示：



（图：系统扩展结构）

\*注： 等到垂直拆分也无法满足容量需求的时候，相信官方的cluster也已经release了，可以直接采用。（目前cluster是alpha阶段，还不适合生产环境）

## 2.3持久化策略和数据恢复

Redis支持RDB快照和AOF追加日志两种持久化策略，开启持久化可以保证在机房整体故障后（如机房断电），仍然保存大部分缓存数据，不会对DB产生冲击。本方案采用RDB快照做持久化，下面是两种策略和数据可靠性的对比：

**RDB快照**： 当满足触发条件时，redis会将当前的内存数据全量dump到磁盘RDB文件中。触发条件通过redis.conf的save参数控制，格式类似于：如果多少秒内有多少个key发生了更新，就执行一次dump。比如：

Save 900 1 🡨如果15分钟内发生1次更新就执行dump

Save 300 10 🡨如果5分钟内发生10次更新就执行dump

Save 60 10000 🡨如果60秒内发生10000次更新就执行dump

Save参数可以叠加，当满足其中一个条件就会促发dump。当执行dump时，redis会fork一个子进程，将内存快照顺序写入一个临时文件，在写入完成后，通过一次原子性rename将临时文件更名为RDB快照文件。

**AOF追加日志**：就是将客户端提交的命令按顺序逐条追加到日志文件中。通过redis.conf中的appendfsync参数控制写入AOF文件的时机，有三种策略：

no 🡨从不主动写入磁盘，完全交由OS控制，对linux来说默认是30秒做一次fsync；

everysec 🡨每秒调用一次fsync写入磁盘，如果系统宕机将丢失最后一秒的更新；

always 🡨对每一个指令都做fsync，可以保证系统宕机时能保存所有更新，但性能很差；

由于AOF方式是不断的往AOF文件追加指令，势必造成AOF文件不断膨胀。对此redis提供了AOF rewrite机制：将AOF文件中同一个key的多个指令汇总成一条指令，从而将AOF文件维持在一个固定范围内。对于rewrite的时机，通过redis.conf的如下参数进行控制，两个参数必须同时满足才促发rewrite：

Auto-aof-rewrite-percentage 100 🡨当AOF文件达到上次rewrite后文件大小的百分比时触发

Auto-aof-rewrite-min-size 64mb 🡨当AOF文件达到指定大小时触发

从实现原理看，除了always策略下的AOF，其余策略均不能保证数据100%可靠。从目前的应用场景看，系统可以允许部分数据丢失：当缓存中没有对应数据时，就从DB重新查询。这里采用RDB作为持久化策略。

当Redis重启时，会自动从本地RDB文件加载数据，当数据加载完毕后，Redis才能对外提供服务。数据的加载时间请见附录的基准测试。

## 2.4物理结构部署要求

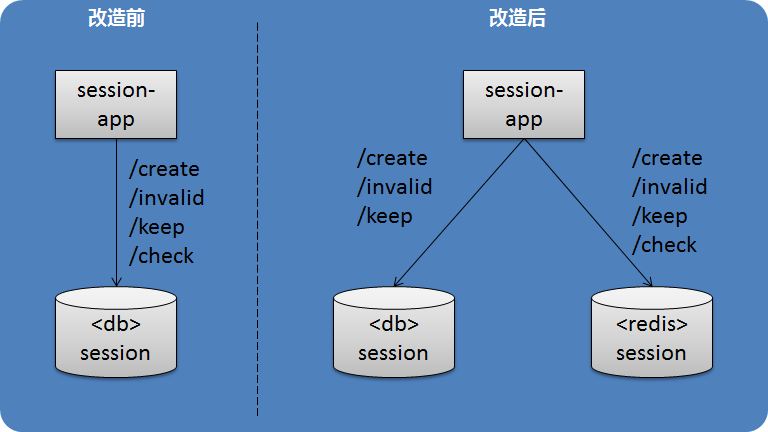
在物理部署结构上，有如下要求：

* 同一台物理机可以部署多个Redis实例，但同一组master/slave要求部署在不同物理机上，避免因物理机器故障导致一组master/slave同时不可用；
* 每个Redis实例建议至少分配16Gb内存空间；
* Sentinel实例只负责心跳检测和主从切换，访问压力较小，部署在较普通的虚拟机上即可；

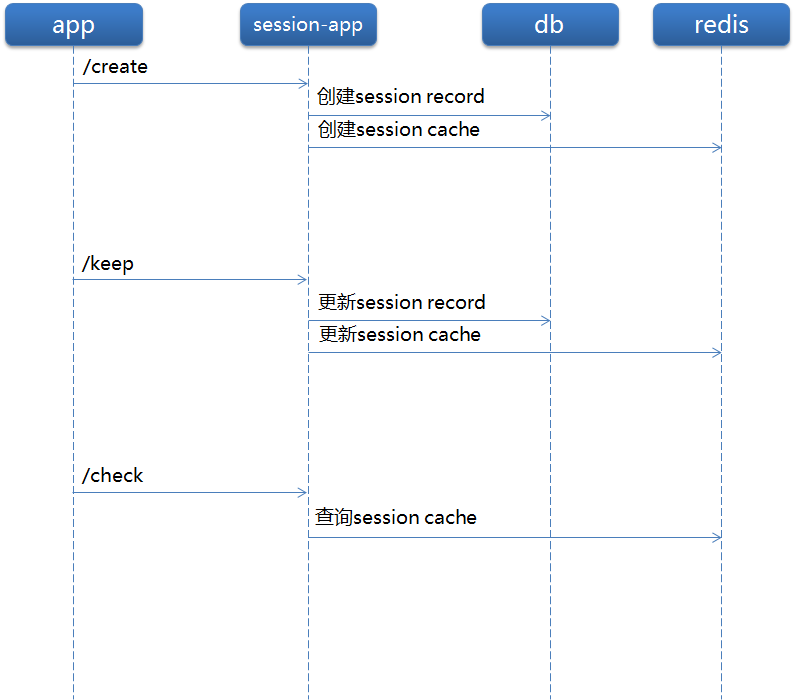
# 3 Session-Cache设计

## 3.1系统设计

目前系统的session存储依赖于DB，create、invelid、keep、check四个操作均需要访问DB。为缓解DB压力，可以将session同步存储到redis中。对于session的写操作需同时写DB和redis，对于纯读取的session check则直接从redis读取。



（图：session-app新老系统结构）



（图：session系统调用过程）

如上图所示，对于session的create、invelid、keep操作，由session-app先写db，再写redis。对于check，session-app直接通过redis读取。

## 3.2数据结构



（T\_SESSION表结构）

对于session的存储结构，可以将key设计为 userid + sessionid。这里的userid在当前是一个冗余字段，可以用于日后按用户维度做sharding。 value则采用redis的hash结构存储，这样客户端可以随意存储任意字段，而不必每次都获取整笔记录，同时也减去了序列化的开销。一笔session的数据结构类似于：

key = userid：sessionid

value = {

party\_no = xxx

start\_time = xxx

access\_time = xxx

end\_time = xxx

…

}

## 3.3回退方案

当前session-app带有回退功能，如果db/cache不可用，可以转为客户端session方式。

# 4 List-Cache设计

## 4.2数据结构

对于列表页面的缓存，主要涉及status=ONLINE/DONE的products表数据，页面展示需求为优先展示ONLINE，再展示DONE，支持按“当前投资金额”和“投资期限”升/降序排序。基于此需求，对ONLINE/DONE数据分别采用两个Sorted Sets存储，一个以“当前投资金额”排序，另一个以“投资期限“排序。数据结构类似于：

按当前投资金额排序的sorted sets：

Key = list：online：amount

Value = {

产品记录的json串， 当前投资金额；

产品记录的json串， 当前投资金额；

…

}

按投资期限排序的sorted sets：

Key = list：online：num\_of\_investments

Value = {

产品记录的json串， 投资期限；

产品记录的json串， 投资期限；

…

}

list-app通过zrange/zrevrange指令进行分页查询，查询指令类似于：

按“当前投资金额“逆序排列，查询第一页：

zrevrange “list : online : amount” 0 9

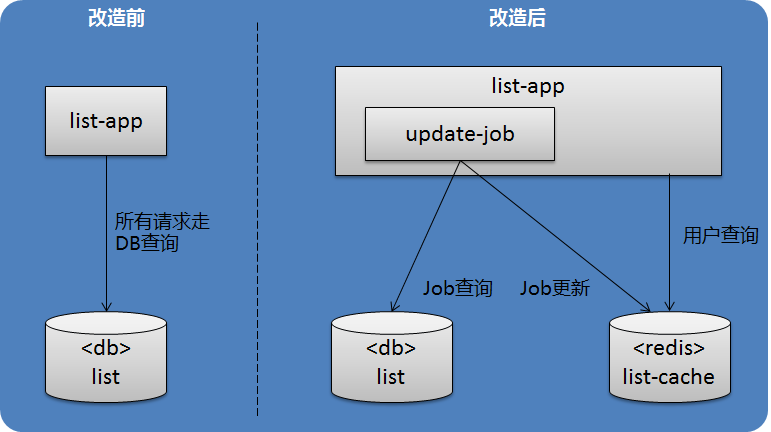
按“投资期限“升序排列，查询第二页：

zrange “list : online : num\_of\_investments” 10 19

另外，对于导航条上统计条目的缓存，直接采用Strings结构即可。

## 4.1系统设计

lufax.com上，首页（www.lufax.com）和产品列表（list.lufax.com/list/listing）是网站的重点页面。这两页面都涉及到产品列表的查询、统计，随着访问量的增长，现有每次用户访问都查询DB的方式，会对DB造成压力。改进方式是，由定时程序将数据库的列表信息同步到缓存中，用户访问时只从缓存查询。涉及的系统结构如下：



（图：list-app新老系统结构）

定时程序的处理逻辑如下：

1. 从DB查询当前列表信息；
2. 将列表写入Redis，Key为临时值，比如list : onlie : tmp；
3. 在写入完毕后，临时Key名Rename为正式名称，比如list : online；
4. 删除临时Key；

定时程序根据具体功能配置调度频率。比如对于ONLINE列表的缓存，由于对实时性要求较高，大概每1~3秒触发一次。而对于DONE列表的缓存，可以是1~5分钟触发一次。

## 4.3回退方案

在初次上线时，list-app需要设计开关用于控制是读取redis还是读取db。如果redis出现不可预知故障时，由运维切换开关，转为查询db。待运行稳定之后，可以去掉开关。

# 5.Redis核心功能

本章主要描述Redis的若干核心功能，主要供应用系统在程序设计上参考使用。

## 5.1过期时间

Redis支持对key设置过期时间，当key达到指定期限后redis将删除该key。该特性可用应用于：1）避免冷数据长期占用内存空间； 2）精确控制有明确存活期限的数据，比如session。

对于key的过期判断，redis采用主动删除和被动删除两个策略并行处理： 1）后台deamon每秒钟会做十次过期检查，每次随机挑选出100个key并删除其中的过期key，如果本次删除的过期key占比超过了25%，则再做一次； 2）当用户操作一个key时，判断该key是否过期。 这里可以看出，过期的key实际上不是立即被删除的，但这样可以保证大部分过期的key能在较短时间内被删除，同时不影响过期判断的效率。

过期时间的相关API：

|  |  |
| --- | --- |
| 指令 | 说明 |
| expire <key> <seconds> | 设置一个key的过期时间为XX秒 |
| ttl <key> | 获取一个key的过期时间，单位为秒。如果key不会过期返回-1，如果key不存在返回-2 |
| pexpire <key> <milliSeconds> | 同expire，过期时间精确到毫秒 |
| ptll <key> | 同ttl，时间精确到毫秒 |
| persist <key> | 取消key的过期时间，即不设置过期 |

## 5.2事务

Redis支持简单事务管理，当客户端开启事务后，Redis会将客户端提交的指令放入缓冲队列，直到事务提交时才一次性执行。期间如果有其它客户端更新了数据，则事务提交失败。

事务管理的相关API：

|  |  |
| --- | --- |
| 指令 | 说明 |
| watch | 跟踪一个key的状态变化，如果一个key在事务提交之前被其它人改动，事务提交会失败，相当于一个乐观锁。当事务提交或回滚时，自动取消该key的跟踪 |
| unwatch | 解除一个key的跟踪 |
| multi | 启动事务 |
| exec | 提交事务 |
| discard | 回滚事务 |

一个完善的事务处理模板如下：

**watch** some keys

**get** keys

check something

**multi**

**set** keys

**exec**

## 5.3内存上限和剔除策略

Redis支持设定内存上限，当内存达到上限后，可通过redis.conf的maxmemory-policy参数设置key的踢出策略，总共有：

**volatile-lru**：对有过期时间的key使用LRU算法踢除；

**allkeys-lru**：对所有key使用LRU算法踢除；

**volatile-random**：随机踢除有过期时间的key；

**allkey-random**：随机踢除任意key；

**boeviction**：不踢除原有的key，添加新的key会报错；

注：为了均衡性能，redis采用的只是近似的LRU算法，但对应用来说一般也足够了。这里我们采用allkeys-lru。内存达到上限后的性能损耗，可参考附录的基准测试。

## 5.3批处理

客户端通常是通过tcp长连接到redis服务器，采用一请求一应答的调用模式。即客户端每发送一个请求，都涉及到“发出请求、服务端处理、等待响应“的过程。从提升性能考虑，redis提供了批处理功能（pipeline），客户端可以将多个请求打包发送到redis，redis顺序处理并一次性返回结果。通过这个方式减少网络开销。如果应用程序设计一次处理多条指令，推荐采用pipeline。 Pipeline带来的性能提升可以参考附录的基准测试。

## 5.4运行监控

Redis没有提供监控系统，不过已经有很多第三方应用可以参考，比如基于python的Redis-live，基于ruby的Redis-stat，另外也可以将redis集成到ZABBIX，具体看运维的监控规范吧。

## 5.5Java客户端

现在市面上已经有若干Java客户端，这里没有对其做详细比较，直接选择Github上用户数最多的Jedis。Jedis基于Redis原生协议实现了所有Redis命令，同时也提供连接池、Sentinel客户端、Sharding客户端等高级功能。App将通过Jedis访问sentinel和redis。

Jedis采用/不用连接池的性能对比请见附录的基准测试。

# 附录

## 附1，redis.conf配置参考

${redis\_home}/redis.conf主要配置redis实例的相关特性，主要配置如下：

|  |  |  |
| --- | --- | --- |
| 配置项 | 推荐值 | 说明 |
| loglevel | Notice | 经测试，打印明细级别的日志会损失一定的性能，所以这里设置为Notice即可；另外redis的log自身不支持滚动日志，可以通过logrotate单独配置； |
| save | Save 900 1  Save 300 10  Save 60 1000 | RDB的dump策略，保证在频繁访问下故障，只丢失1000次更新； |
| maxmemory | 16Gb | 设置每个redis实例的内存上限，具体看运维安排，建议配置到16Gb或以上； |
| maxmemory-policy | Allkeys-lru | 内存达到上限后的清理策略，按LRU踢除； |
| Appendonly | No | 不开启AOF； |
|  |  |  |
|  |  |  |

## 附2，sentinel.conf配置参考

${redis\_home}/sentinel.conf用于配置对redis实例的监控行为，以监控一组session-cache的redis为例，配置如下（红色字体可根据实际情况修改）：

sentinel monitor session-master 172.17.20.50 6379 1 🡨指定master节点地址

sentinel known-slave session-master 172.17.20.1 6379 🡨指定slave节点地址，可以不配置，sentinel能自动识别

sentinel known-sentinel session-master 172.17.40.51 26379

7a7ee9a2466425b91014039fa09ba188352aef19 🡨其余sentinel实例地址，可以不配置，sentinel能自动识别  
sentinel down-after-milliseconds session-master 5000 🡨心跳检查持续多长时间没有响应则认为是故障，单位毫秒  
sentinel config-epoch session-master 1 🡨 每做一次切换会更新一个epoch版本，默认1即可

## 附3，Jedis-Sentinel客户端配置参考

Jedis客户端主要涉及连接池和Sentinel两部分的配置：

**连接池配置：**

Jedis连接池基于apache commons-pool，其配置项如下：

JedisPoolConfig poolingConfig = new JedisPoolConfig();

poolingConfig.setMaxTotal(20); // 维护一个Redis实例最大连接数  
poolingConfig.setMaxIdle(18); // 最大Idle连接数  
poolingConfig.setMinIdle(2); // 最小Idle连接数  
poolingConfig.setBlockWhenExhausted(true); // 连接池饱和策略，true：连接池满时将阻塞，阻塞时间由下面的MaxWaitMillis决定；false：连接池满时直接报错  
poolingConfig.setMaxWaitMillis(1000 \* 5); // borrowObject的最大的等待时间，如果超过等待时间，则抛出JedisConnectionException  
poolingConfig.setLifo(false); // borrowObject返回对象的机制，true：采用LIFO队列； false：采用FIFO队列  
poolingConfig.setTestOnBorrow(false); //在borrowObject时测试连接是否可用  
poolingConfig.setTestOnReturn(false);//在returnObject时测试连接是否可用  
poolingConfig.setTestWhileIdle(true); //当连接idle时测试连接是否可用  
poolingConfig.setTimeBetweenEvictionRunsMillis(1000 \* 30); //idle扫描的间隔时间  
poolingConfig.setNumTestsPerEvictionRun(-1); //每次idle扫描检查的连接数量，-1为不限制  
poolingConfig.setMinEvictableIdleTimeMillis(1000 \* 60); //一个连接停留在idle状态的最短时间，然后才能被idle object evitor扫描并清除

**Sentinel配置和使用：**

Set<String> sentinels = new HashSet<String>(); //添加sentinel实例地址，如果是vip则填写vip地址  
sentinels.add("172.17.40.52:26379");  
sentinels.add("172.17.40.51:26379");  
GenericObjectPoolConfig config = new GenericObjectPoolConfig(); //这里设置上述的连接池配置  
  
JedisSentinelPool pool = new JedisSentinelPool("session-master", sentinels, config);  
Jedis conn = null;  
try {

// 获取连接，执行各种操作  
      conn = pool.getResource();

} catch (Exception e) {

// 连接不可用时要归还故障连接  
try {  
 if (null != conn) {  
       pool.returnBrokenResource(conn);  
       }  
  } catch (Exception ee) { // 防止万一归还时异常，一般不会发生  
      log …  
  }

} finally {  
if (null != conn) { // 如果上面返回了故障连接，这里会是null  
 try {  
       pool.returnResource(conn);  
       } catch (Exception e) {  
        log …  
       }  
 }

}

## 附4，基准测试数据

**测试环境**

本次基准测试全部在开发网络中进行，采用标配开发机，客户端和服务端独立部署，具体参数为：

CPU： i7-3770（四核八线程， 3.4GHz）

内存：16GB （其中redis占用1G）

网卡：100Mbps

磁盘：7200RPM机械磁盘

OS： linux 2.6.32-279.el6.x86\_64

**持久化测试1**

测试目的：RDB/AOF两种策略下启动耗时的比较

测试方式：事先存储1G数据，查看Redis启动耗时

测试结论：RDB方式比AOF快1+倍

|  |  |  |
| --- | --- | --- |
|  | **RDB方式** | **AOF方式** |
| 加载本地RDB/AOF文件 | 约6+秒 | 约15+秒 |

**持久化测试2**

测试目的：纯内存/RDB/AOF三种策略下的性能对比

测试方式：分别在纯内存/RDB/AOF三种策略下，从客户端构造100连接，发起1000000次请求，每个请求体大小2byte，查看平均耗时

测试结论：AOF（always）策略下的写入性能明显下降，其它策略的性能没有明显差距

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **纯内存** | **RDB** | **AOF（everysec）** | **AOF（always）** |
| GET（TPS） | 62402.50 | 60912.10 | 62398.61 | 2193.26 |
| SET（TPS） | 63000.06 | 57759.73 | 63702.38 | 61180.79 |

**pipeline测试**

测试目的：开启pipeline后的性能提升

测试方式：从客户端构造100连接，发起1000000次请求，分别打开/关闭pipeline

测试结论：以10个请求为一组pipeline，性能可以提升约4倍

|  |  |  |
| --- | --- | --- |
|  | **不使用pipeline** | **每10个请求作为pipeline** |
| GET（TPS） | 61005.37 | 277392.50 |
| SET（TPS） | 58075.38 | 201369.31 |

**内存达到上限测试**

测试目的：内存达到上限后是否影响性能。

测试方式：限制Redis内存上限1GB，在Redis内存为空/满时，从客户端构造100连接，发起1000000次请求，计算平均处理时间。

测试结论：在内存达到上限后，性能没有明显变化

|  |  |  |
| --- | --- | --- |
|  | **内存为空** | **内存满** |
| GET（TPS） | 62189.02 | 62138.82 |
| SET（TPS） | 62162.00 | 61278.27 |

**Jedis开启连接池测试**

测试目的：使用Jedis连接池之后的性能提升

测试方式：构造250并发请求，对比不使用线程池、使用20个连接数的线程池的处理时间。

测试结论：使用线程池性能可以提升20倍

|  |  |  |
| --- | --- | --- |
|  | **不使用线程池** | **使用线程池** |
| 执行5次测试的耗时 | 1626ms, 1513ms, 1403ms, 1567ms, 1537ms | 74ms,   61ms,   72ms,   60ms,   70ms |
| 平均耗时 | 1529.2ms | 67.4ms |

**Jedis开启testOnBorrow测试**

测试目的：开启testOnBorrow/testWhileIdle之后的性能损耗

测试方式：分别在“全部关闭/开启testOnBorrow/开启 testWhileIdle”场景下，使用20连接的连接池，构造并发250并发线程，每线程做50次set，对比处理耗时

测试结论：开启testOnBorrow会造成80%的性能损耗，开启testWhildIdle没有明显性能损耗

|  |  |  |  |
| --- | --- | --- | --- |
|  | **关闭TestOnBorrow和TestWhileIdle** | **只开启TestOnBorrow** | **只开启TestWhileIdle** |
| 执行5次测试的耗时 | 601ms, 616ms, 604ms, 586ms, 612ms | 1108ms, 1102ms, 1120ms, 1113ms, 1100ms | 629ms, 590ms, 598ms, 601ms, 611ms |
| 平均耗时 | 603.8ms | 1108.6ms | 605.8ms |

空白空白空白空白空白空白空白空白空白空白空白空白空白空白空白空