Redis 学习笔记

Redis 是什么

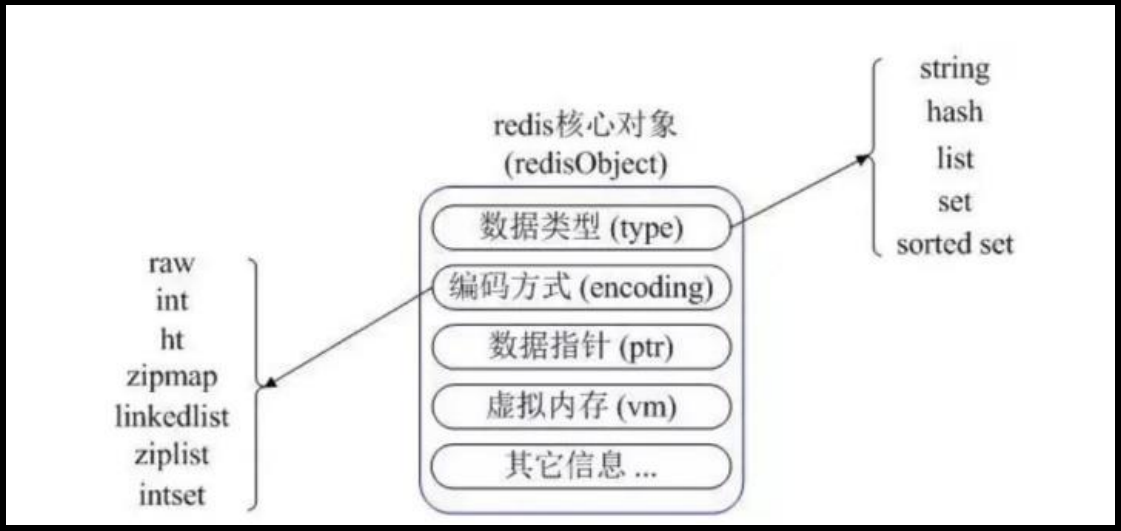
Redis 是 C 语言开发的一个开源的（遵从 BSD 协议）高性能键值对（key-value）的内存数据库，可以用作数据库、缓存、消息中间件等。它是一种 NoSQL（not-only sql，泛指非关系型数据库）的数据库。

特点

1、性能优秀，数据在内存中，读写速度非常快，支持并发 10W QPS；
2、单进程单线程，是**线程安全**的，采用 **IO 多路复用机制**；
3、丰富的数据类型，支持字符串（strings）、散列（hashes）、列表（lists）、集合（sets）、有序集合（sorted sets）等；
4、支持数据持久化。可以将内存中数据保存在磁盘中，重启时加载；
5、主从复制，哨兵，高可用；
6、可以用作分布式锁；
7、可以作为消息中间件使用，支持发布订阅

5 种数据类型

Redis 内部内存管理是如何描述这 5 种数据类型的：



redis 内部使用一个 redisObject 对象来表示所有的 key 和 value
redisObject 最主要的信息：

- type 表示一个 value 对象具体是何种数据类型
- encoding 是不同数据类型在 redis 内部的存储方式
  如：type=string 表示 value 存储的是一个普通字符串，那么 encoding 可以是 raw 或者 int。

5 类：

- string:
  最基本的类型

  string 类型是**二进制安全**的，意思是 redis 的 string 类型可以**包含任何数据**，比如 jpg 图片或者序列化的对象。string 类型的值最大能存储 512M。（另外如 int,bool,float 等）
- hash:
  键值（key-value）的集合
  redis 的 hash 是一个 string 的 key 和 value 的映射表
  Hash 特别适合存储对象。常用命令：**hget，hset，hgetall** 等。
- list:
  简单的字符串列表
  常用命令：lpush、rpush、lpop、rpop、lrange(获取列表片段)
  是一个**双向链表**，既可以支持反向查找和遍历
- set
  string 类型的**无序无重**集合
  是通过 hashtable 实现的
- zset: sorted set
  和 set 一样是 string 类型元素的集合
  常用命令：zadd、zrange、zrem、zcard
  可以通过用户额外提供一个优先级（score）的参数来为成员排序，并且是插入有序的，即自动排序。
  实现方式：Redis sorted set 的内部使用 HashMap 和跳跃表(skipList)来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score，使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

类型	简介	特性	场景
string（字符串）	二进制安全	可以包含任何数据，比如jpg图片或者序列化对象	---
Hash（字典）	键值对集合，即编程语言中的map类型	适合存储对象，并且可以像数据库中的update一个属性一样只修改某一项属性值	存储、读取、修改用户属性
List（列表）	链表（双向链表）	增删快，提供了操作某一元素的api	最新消息排行；消息队列
set（集合）	hash表实现，元素不重复	添加、删除、查找的复杂度都是O(1)，提供了求交集、并集、差集的操作	共同好友；利用唯一性，统计访问网站的所有Ip
sorted set（有序集合）	将set中的元素增加一个权重参数score，元素按score有序排列	数据插入集合时，已经进行了天然排序	排行榜；带权重的消息队列

redis 缓存使用

一般有两种方式：

- 直接通过 RedisTemplate 来使用
  默认情况下的模板只能支持 RedisTemplate<String, String>，也就是只能存入字符串，所以自定义模板很有必要

RedisCacheConfig.java

```java
@Configuration
@AutoConfigureAfter(RedisAutoConfiguration.class)
public class RedisCacheConfig {

    @Bean
    public RedisTemplate<String, Serializable> redisCacheTemplate(LettuceConnectionFact

        RedisTemplate<String, Serializable> template = new RedisTemplate<>();
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
        template.setConnectionFactory(connectionFactory);
        return template;
    }
}
```

测试类

```java
@RestController
@RequestMapping("/user")
public class UserController {

    public static Logger logger = LogManager.getLogger(UserController.class);

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Autowired
    private RedisTemplate<String, Serializable> redisCacheTemplate;

    @RequestMapping("/test")
    public void test() {
        redisCacheTemplate.opsForValue().set("userkey", new User(1, "张三", 25));
        User user = (User) redisCacheTemplate.opsForValue().get("userkey");
        logger.info("当前获取对象: {}", user.toString());
    }
```

- 使用 spring cache 集成 Redis（也就是注解的方式）

  缓存注解：

  - @Cacheable 根据方法的请求参数对其结果进行缓存
    - key：缓存的 key，可以为空，如果指定要按照 SPEL 表达式编写，如果不指定，则按照方法的所有参数进行组合。
    - value：缓存的名称，必须指定至少一个（如 @Cacheable (value='user') 或者@Cacheable(value={'user1','user2'})）
    - condition：缓存的条件，可以为空，使用 SPEL 编写，返回 true 或者 false，只有为 true 才进行缓存。

    用于查询和添加缓存，第一次查询的时候返回该方法返回值，并向 Redis 服务器保存数据。

    以后调用该方法先从 Redis 中查是否有数据，如果有直接返回 Redis 缓存的数据，而不执行方法里的代码。如果没有则正常执行方法体中的代码。
  - @CachePut 根据方法的请求参数对其结果进行缓存

和@Cacheable不同的是，它每次都会触发真实方法的调用

- @CacheEvict 根据条件对缓存进行清空
  - allEntries：是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存
  - beforeInvocation：是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存。缺省情况下，如果方法执行抛出异常，则不会清空缓存。

缓存问题

- 缓存和数据库数据一致性问题

  分布式环境下非常容易出现缓存和数据库间数据一致性问题，针对这一点，如果项目对缓存的要求是强一致性的，那么就不要使用缓存。我们只能采取合适的策略来降低缓存和数据库间数据不一致的概率，而无法保证两者间的强一致性。合适的策略包括合适的缓存更新策略，更新数据库后及时更新缓存、缓存失败时增加重试机制。

- Redis 雪崩

  一般缓存都是定时任务去刷新

  如果**同一时间缓存中的 key 大面积失效，瞬间 Redis 跟没有**一样，那这个数量级别的请求直接打到数据库几乎是灾难性的，你想想如果挂的是一个用户服务的库，那其他依赖他的库所有接口几乎都会报错，如果没做熔断等策略基本上就是瞬间挂一片的节奏，你怎么重启用户都会把你打挂。

  处理缓存雪崩：在批量往 Redis 存数据的时候，把每个 Key 的失效时间都加个随机值就好了，这样可以**保证数据不会再同一时间大面积失效**。

  ```
  setRedis（key，value，time+Math.random()*10000）；
  ```

- 缓存穿透

  缓存穿透是**指缓存和数据库中都没有的数据**，而用户（黑客）不断发起请求，

  举个栗子：我们数据库的 id 都是从 1 自增的，如果发起 id=-1 的数据，这样的不断攻击导致数据库压力很大，严重会击垮数据库。

  解决：

  - 在接口层增加**校验**，比如用户鉴权，参数做校验，不合法的校验直接 return，比如 id 做基础校验，id<=0 直接拦截。
  - 布隆过滤器（Bloom Filter）：利用高效的数据结构和算法快速判断出你这个 Key 是否在数据库中存在，不存在你 return 就好了

- 缓存击穿

指一个 Key 非常热点，在不停地扛着大量的请求，大并发集中对这一个点进行访问，当这个 Key 在失效的瞬间，持续的大并发直接落到了数据库上，就在这个 Key 的点上击穿了缓存。

和雪崩有点相似，不过其是大面积失效，击穿是一个热点 key 失效，作用原理都是大量请求直接砸到数据库

解决：

- 设置热点数据永不过期
- 加上互斥锁

```
public static String getData(String key) throws InterruptedException {
        //从Redis查询数据
        String result = getDataByKV(key);
        //参数校验
        if (StringUtils.isBlank(result)) {
            try {
                //获得锁
                if (reenLock.tryLock()) {
                    //去数据库查询
                    result = getDataByDB(key);
                    //校验
                    if (StringUtils.isNotBlank(result)) {
                        //插进缓存
                        setDataToKV(key, result);
                    }
                } else {
                    //睡一会再拿
                    Thread.sleep(100L);
                    result = getData(key);
                }
            } finally {
                //释放锁
                reenLock.unlock();
            }
        }
        return result;
    }
```

锁是开的

Redis 为何这么快

快：官方提供的数据可以达到 100000+的 QPS（每秒内的查询次数），这个数据不比 Memcached 差！

Redis 为什么是**单线程**的？

Redis 确实是**单进程单线程**的模型，因为 Redis 完全是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章的采用单线程的方案了

Redis 是单线程的，为什么还能这么快？

1. Redis 完全基于内存，绝大部分请求是纯粹的内存操作，非常快
2. 数据结构简单，对数据操作也简单
3. 采用单线程，**避免了不必要的上下文切换和竞争条件**，不存在多线程导致的 CPU 切换，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有死锁问题导致的性能消耗。
4. 使用**多路复用 IO 模型，非阻塞 IO**。

Redis 和 Memcached 的区别

1. **存储方式**上：memcache 会把数据全部存在内存之中，断电后会挂掉，数据不能超过内存大小。redis 有部分数据存在硬盘上，这样能保证数据的持久性。
2. **数据支持**类型上：memcache 对数据类型的支持简单，只支持简单的 key-value，，而 redis 支持五种数据类型。
3. 使用**底层模型**不同：它们之间底层实现方式以及与客户端之间通信的应用协议不一样。redis 直接自己构建了 VM 机制，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。
4. value 的大小：**redis 可以达到 1GB，而 memcache 只有 1MB**。

淘汰策略

六种淘汰策略：

策略	描述
volatile-lru	从已设置过期时间的KV集中优先对最近最少使用(less recently used)的数据淘汰
volitile-ttl	从已设置过期时间的KV集中优先对剩余时间短(time to live)的数据淘汰
volitile-random	从已设置过期时间的KV集中随机选择数据淘汰
allkeys-lru	从所有KV集中优先对最近最少使用(less recently used)的数据淘汰
allKeys-random	从所有KV集中随机选择数据淘汰
noeviction	不淘汰策略，若超过最大内存，返回错误信息

Redis4.0 加入了 LFU(least frequency use)淘汰策略，包括 volatile-lfu 和 allkeys-lfu，通过统计访问频率，将访问频率最少，即最不经常使用的 KV 淘汰。

持久化

redis 为了保证效率，数据缓存在了内存中，但是会**周期性的把更新的数据写入磁盘**或者把修改操作写入追加的记录文件中，以保证数据的持久化。

持久化策略有两种：

1. RDB(默认)：**快照形式**是直接把内存中的数据保存到一个 dump.rdb 的文件中，定时保存，保存策略。

   原理：当 Redis 需要做持久化时，Redis 会 fork 一个子进程，子进程将数据写到磁盘上一个临时 RDB 文件中。当子进程完成写临时文件后，将原来的 RDB 替换掉，这样的好处是可以 copy-on-write。

   优点：这种文件非常适合用于备份：比如，你可以在最近的 24 小时内，每小时备份一次，并且在每个月的每一天也备份一个 RDB 文件。这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。RDB 非常适合灾难恢复。

   缺点：如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不合适你。

2. AOF:把所有的对 Redis 的服务器进行修改的命令都存到一个文件里,命令的集合。(当 Redis 重启的时候，它会优先使用 AOF 文件来还原数据集，因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。)

   原理:使用 AOF 做持久化,每一个写命令都通过 write 函数追加到 appendonly.aof 中

   AOF 可以做到全程持久化，只需要在配置中开启 appendonly yes。这样 redis 每执行一个修改数据的命令，都会把它添加到 AOF 文件中，当 redis 重启时，将会读取 AOF 文件进行重放，恢复到 redis 关闭前的最后时刻。

   优点是会让 redis 变得**非常耐久**。可以设置不同的 fsync 策略，aof 的默认策略

是每秒钟 fsync 一次，在这种配置下，就算发生故障停机，也最多丢失一秒钟的数据。
缺点是对于相同的数据集来说，AOF 的文件体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB。

适用场景：
如果你非常关心你的数据，但仍然可以承受数分钟内的数据丢失，那么可以额只使用 RDB 持久。AOF 将 Redis 执行的每一条命令追加到磁盘中，处理巨大的写入会降低 Redis 的性能，不知道你是否可以接受。数据库备份和灾难恢复：定时生成 RDB 快照非常便于进行数据库备份，并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度快。当然了，redis 支持同时开启 RDB 和 AOF，系统重启后，redis 会优先使用 AOF 来恢复数据，这样丢失的数据会最少。

主从复制

redis 单节点存在单点故障问题，为了解决单点问题，一般都需要对 redis 配置从节点，然后使用哨兵来监听主节点的存活状态，如果主节点挂掉，从节点能继续提供缓存功能

主从配置结合哨兵模式能解决单点故障问题，提高 redis 可用性。从节点仅提供读操作，主节点提供写操作。对于读多写少的状况，可给主节点配置多个从节点，从而提高响应效率。

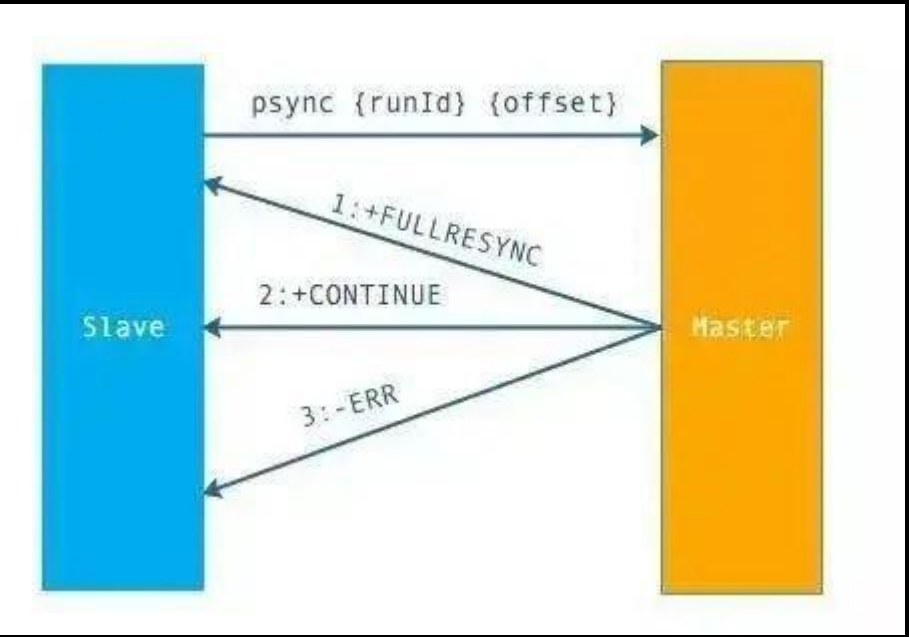
复制过程：
1、从节点执行 slaveof[masterIP][masterPort]，保存主节点信息
2、从节点中的定时任务发现主节点信息，建立和主节点的 socket 连接
3、从节点发送 Ping 信号，主节点返回 Pong，两边能互相通信
4、连接建立后，主节点将所有数据发送给从节点（数据同步）
5、主节点把当前的数据同步给从节点后，便完成了复制的建立过程。接下来，主节点就会持续的把写命令发送给从节点，保证主从数据一致性。

数据同步的过程：
redis2.8 之前使用 sync[runId][offset] 同步命令，redis2.8 之后使用 psync[runId][offset]命令。两者不同在于，sync 命令仅支持全量复制过程，psync 支持全量和部分复制。
几个概念：
1. runId：每个 redis 节点启动都会生成唯一的 uuid，每次 redis 重启后，runId

都会发生变化。

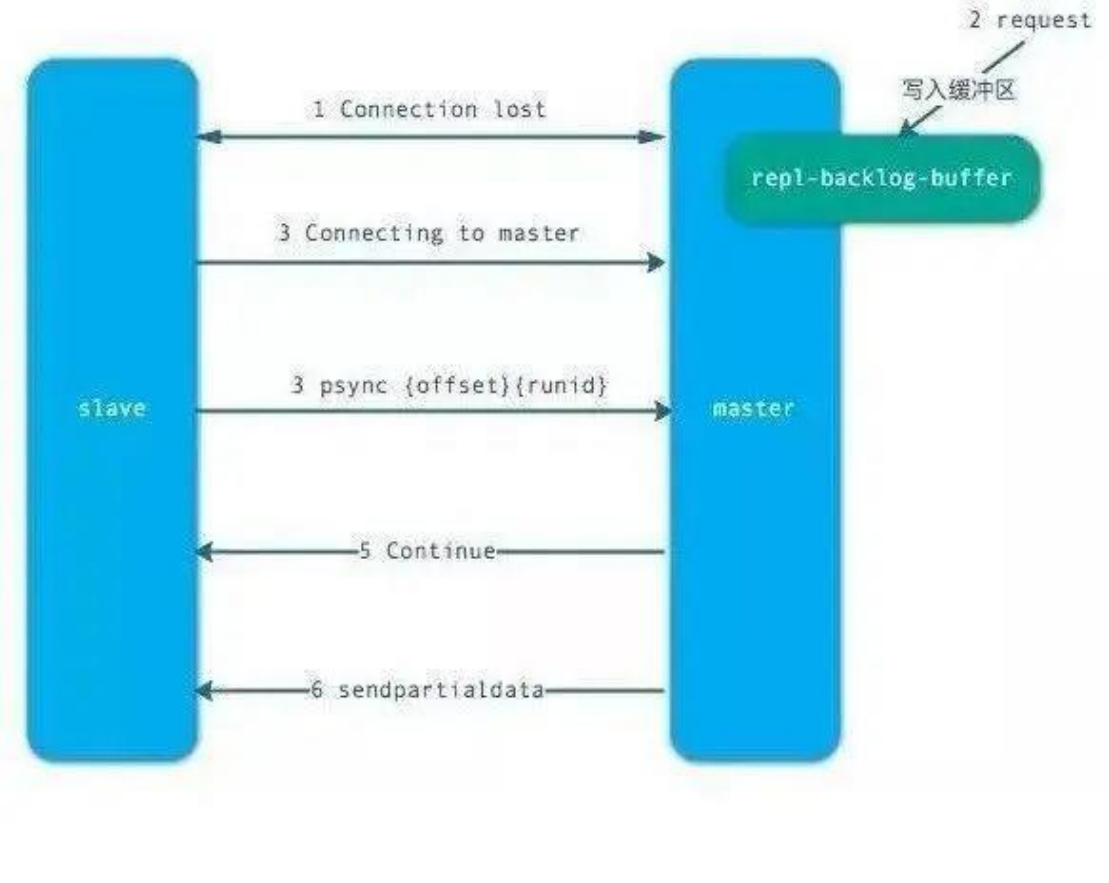
2. offset：主节点和从节点都各自维护自己的主从复制偏移量 offset，当主节点有写入命令时，offset=offset+命令的字节长度。从节点在收到主节点发送的命令后，也会增加自己的 offset，并把自己的 offset 发送给主节点。这样，主节点同时保存自己的 offset 和从节点的 offset，通过对比 offset 来判断主从节点数据是否一致。
3. repl_backlog_size：保存在主节点上的一个固定长度的先进先出队列，默认大小是 1MB。

1）主节点发送数据给从节点过程中，主节点还会进行一些写操作，这时候的数据存储在复制缓冲区中。从节点同步主节点数据完成后，主节点将缓冲区的数据继续发送给从节点，用于部分复制。
2）主节点响应写命令时，不但会把命名发送给从节点，还会写入复制积压缓冲区，用于复制命令丢失的数据补救。



上面是 psync 的执行流程：从节点发送 psync[runId][offset]命令，主节点有三种响应：（1）FULLRESYNC：第一次连接，进行全量复制 （2）CONTINUE：进行部分复制 （3）ERR：不支持 psync 命令，进行全量复制

全量复制的过程：



1、从节点发送 psync ? -1 命令（因为第一次发送，不知道主节点的 runId，所以为?，因为是第一次复制，所以 offset=-1）。
2、主节点发现从节点是第一次复制，返回 FULLRESYNC {runId} {offset}，runId 是主节点的 runId，offset 是主节点目前的 offset。
3、从节点接收主节点信息后，保存到 info 中。
4、主节点在发送 FULLRESYNC 后，启动 bgsave 命令，生成 RDB 文件（数据持久化）。
5、主节点发送 RDB 文件给从节点。到从节点加载数据完成这段期间主节点的写命令放入缓冲区。
6、从节点清理自己的数据库数据。
7、从节点加载 RDB 文件，将数据保存到自己的数据库中。
8、如果从节点开启了 AOF，从节点会异步重写 AOF 文件。

部分复制有以下几点说明：
1、部分复制主要是 Redis 针对全量复制的过高开销做出的一种优化措施，使用 psync[runId][offset]命令实现。当从节点正在复制主节点时，如果出现网络闪断或者命令丢失等异常情况时，从节点会向主节点要求补发丢失的命令数据，主节点的复制积压缓冲区将这部分数据直接发送给从节点，这样就可以保持主从节点复制的一致性。补发的这部分数据一般远远小于全量数据。
2、主从连接中断期间主节点依然响应命令，但因复制连接中断命令无法发送给从节点，不过主节点内的复制积压缓冲区依然可以保存最近一段时间的写命令数据。

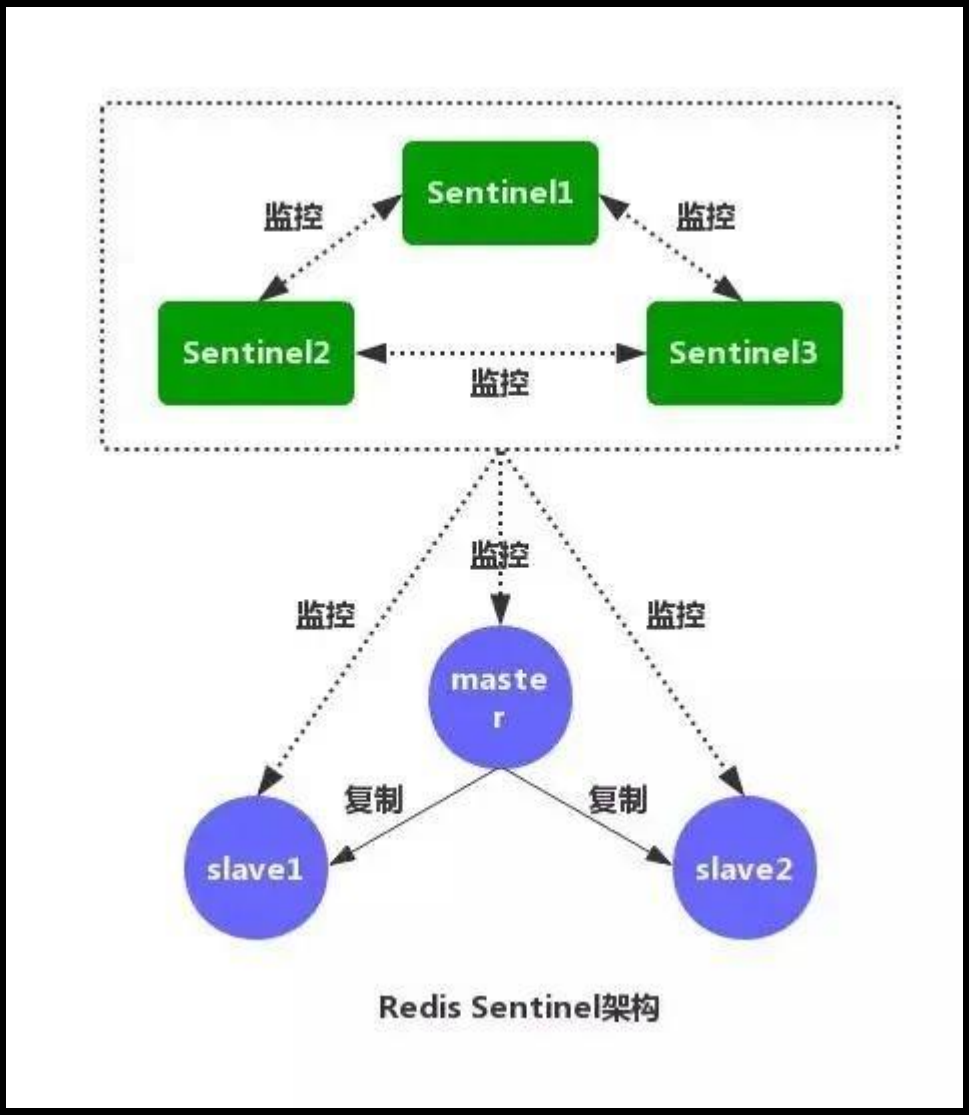
3、当主从连接恢复后，由于从节点之前保存了自身已复制的偏移量和主节点的运行 ID。因此会把它们当做 psync 参数发送给主节点，要求进行部分复制。
4、主节点接收到 psync 命令后首先核对参数 runId 是否与自身一致，如果一致，说明之前复制的是当前主节点；之后根据参数 offset 在复制积压缓冲区中查找，如果 offset 之后的数据存在，则对从节点发送+COUTINUE 命令，表示可以进行部分复制。因为缓冲区大小固定，若发生缓冲溢出，则进行全量复制。

主从复制会存在哪些问题？
1、一旦主节点宕机，从节点晋升为主节点，同时需要修改应用方的主节点地址，还需要命令所有从节点去复制新的主节点，整个过程需要人工干预。
2、主节点的写能力受到单机的限制。
3、主节点的存储能力受到单机的限制。
4、原生复制的弊端在早期的版本中也会比较突出，比如：redis 复制中断后，从节点会发起 psync。此时如果同步不成功，则会进行全量同步，主库执行全量备份的同时，可能会造成毫秒或秒级的卡顿。

解决方案：哨兵

哨兵

Redis Sentinel（哨兵）的架构图：



Redis Sentinel架构

主要功能包括主节点存活检测、主从运行情况检测、自动故障转移、主从切换。Redis Sentinel 最小配置是一主一从。

Redis 的 Sentinel 系统可以用来管理多个 Redis 服务器，该系统可以执行以下四个任务：

1、监控：不断检查主服务器和从服务器是否正常运行。

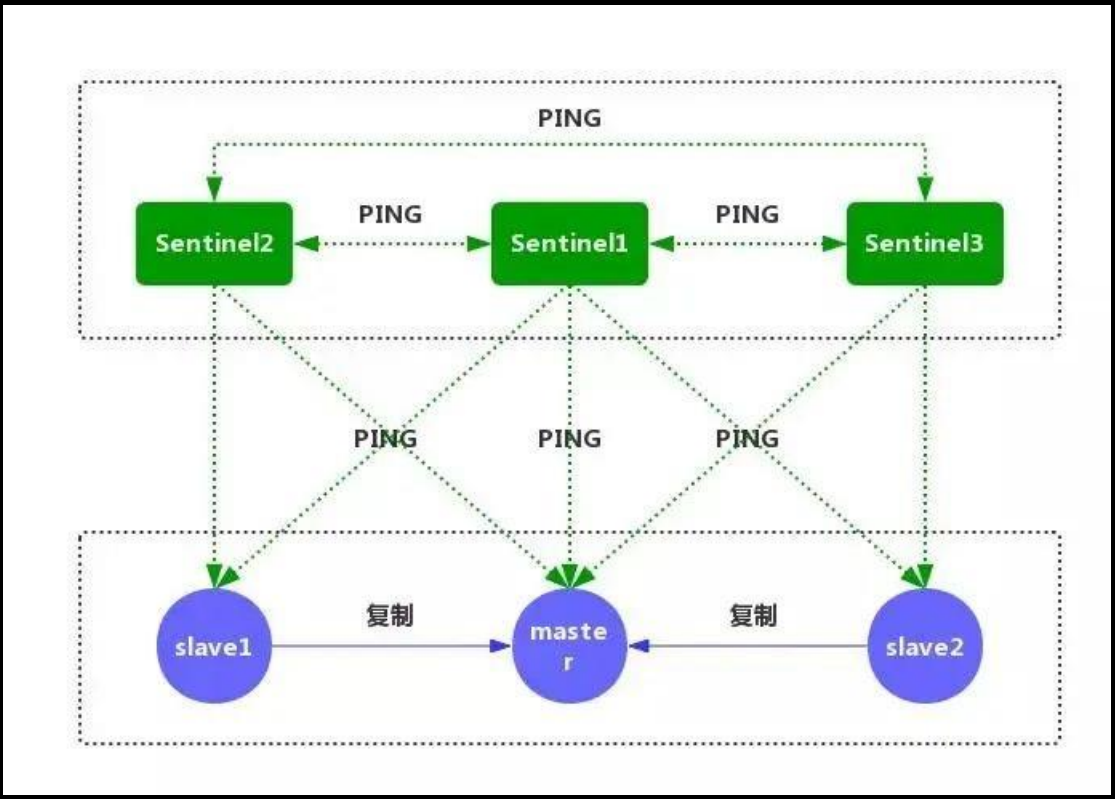
2、通知：当被监控的某个 redis 服务器出现问题，Sentinel 通过 API 脚本向管理员或者其他应用程序发出通知。

3、自动故障转移：当主节点不能正常工作时，Sentinel 会开始一次自动的故障转移操作，它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点，并且将其他的从节点指向新的主节点，这样人工干预就可以免了。

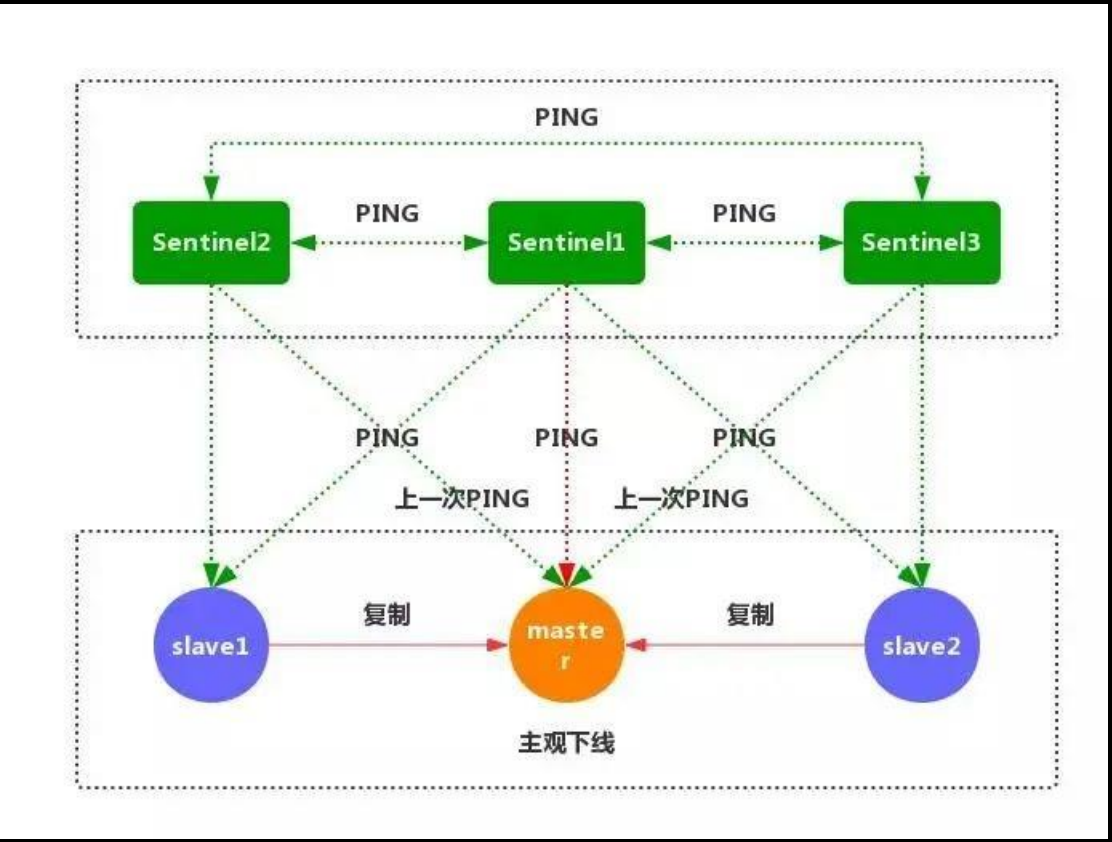
4、配置提供者：在 Redis Sentinel 模式下，客户端应用在初始化时连接的是 Sentinel 节点集合，从中获取主节点的信息。

哨兵的工作原理：

1. 每个Sentinel节点都需要定期执行以下任务:每个Sentinel以每秒一次的频率，向它所知的主服务器、从服务器以及其他的Sentinel实例发送一个PING命令:



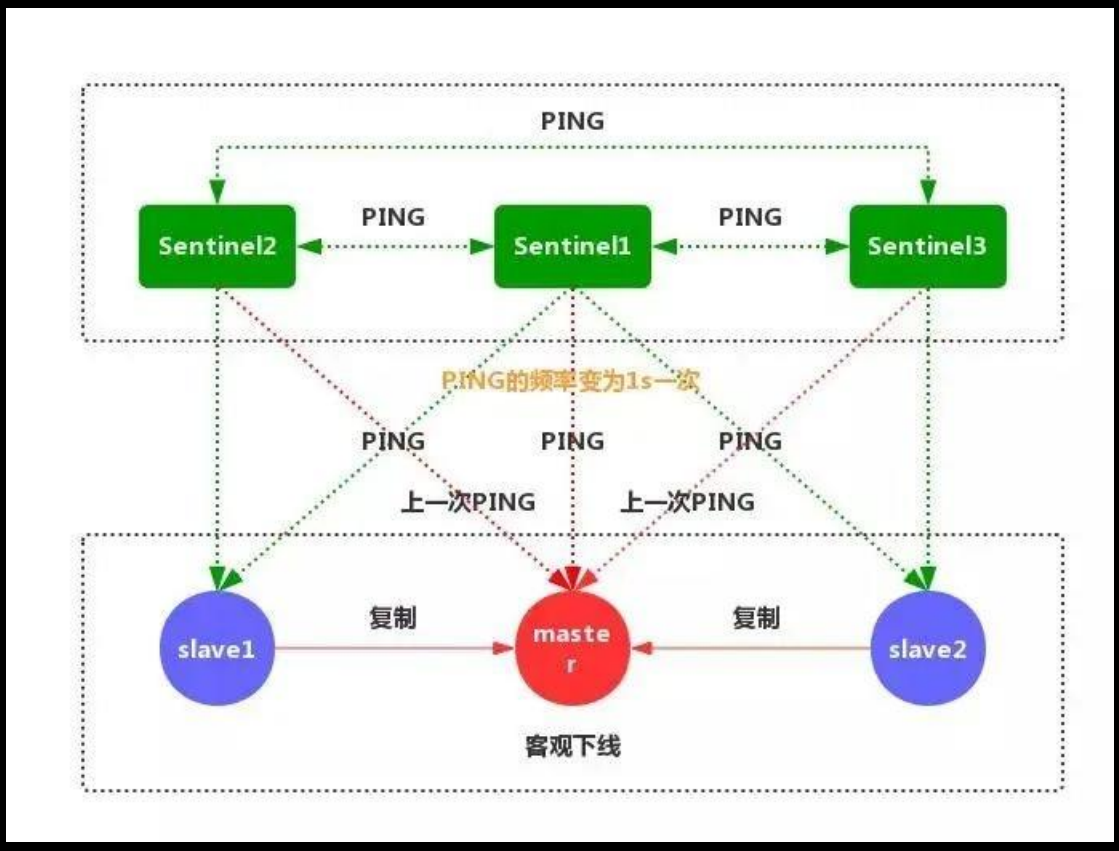
2. 如果一个实例距离最后一次有效回复PING命令的时间超过down-after-milliseconds所指定的值，那么这个实例会被Sentinel标记为主观下线。



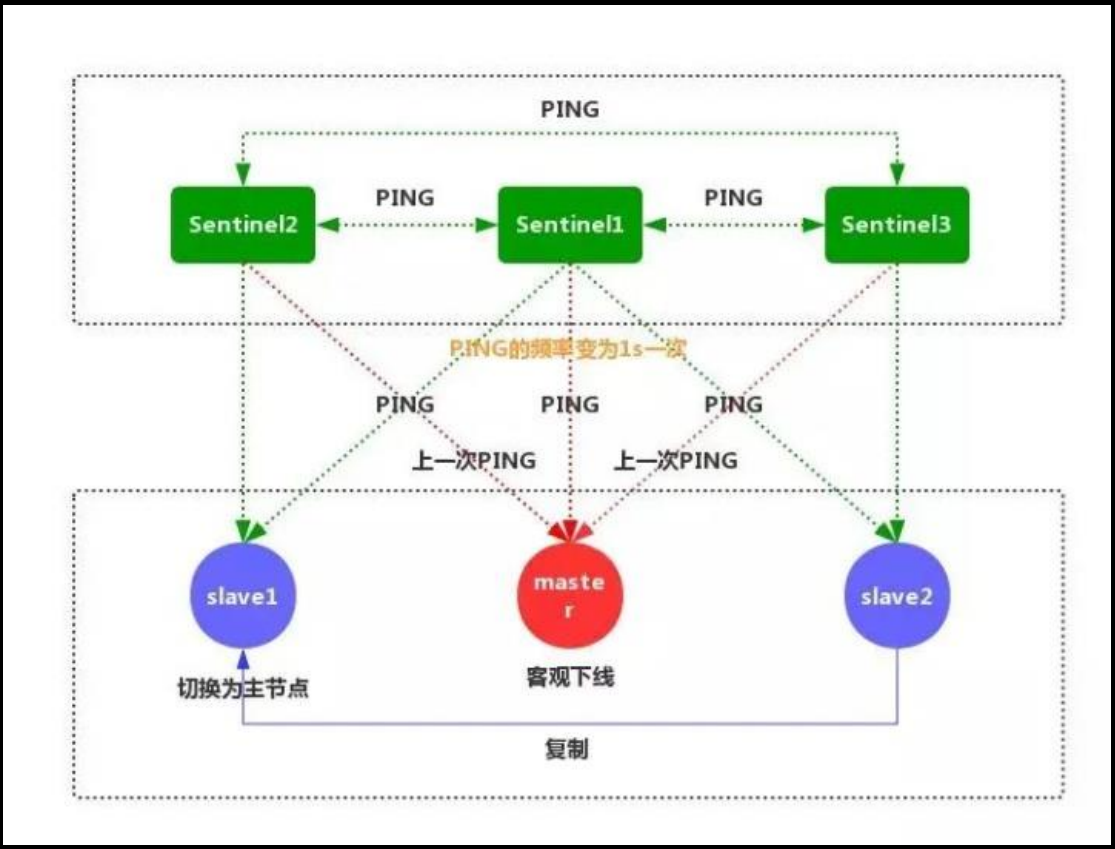
3. 如果一个主服务器被标记为主观下线，那么正在监视这个服务器的所有Sentinel

节点，要以每秒一次的频率确认主服务器的确进入了主观下线状态。

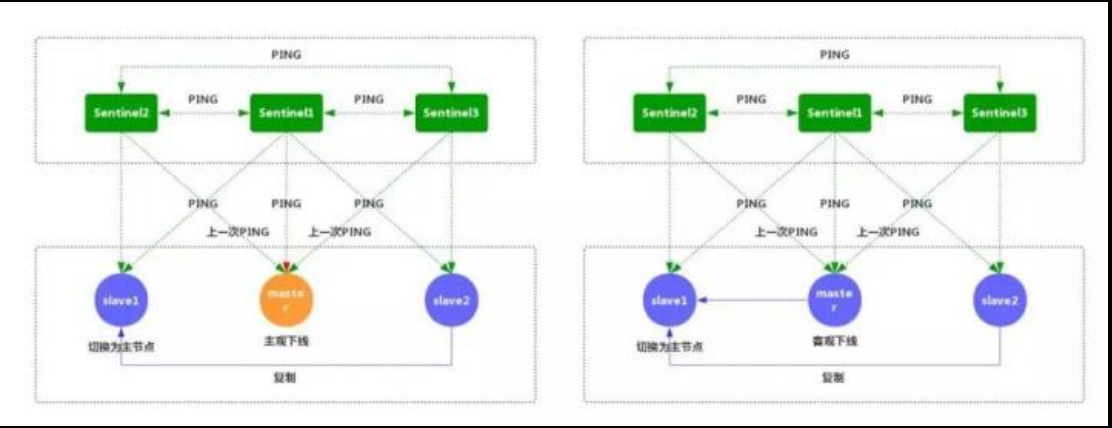
4. 如果一个主服务器被标记为主观下线，并且有足够数量的 Sentinel（至少要达到配置文件指定的数量）在指定的时间范围内同意这一判断，那么这个主服务器被标记为客观下线。



5. 一般情况下，每个 Sentinel 会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令，当一个主服务器被标记为客观下线时，Sentinel 向下线主服务器的所有从服务器发送 INFO 命令的频率，会从 10 秒一次改为每秒一次。



6. Sentinel 和其他 Sentinel 协商客观下线的主节点的状态，如果处于 SDOWN 状态，则投票自动选出新的主节点，将剩余从节点指向新的主节点进行数据复制。



7. 当没有足够数量的 Sentinel 同意主服务器下线时，主服务器的客观下线状态就会被移除。当主服务器重新向 Sentinel 的 PING 命令返回有效回复时，主服务器的主观下线状态就会被移除。

Redis 注解

注解的 condition 和 unless 的区别，condition 对传入值生效，unless 对结果

result 生效，使用了错误的 result 导致报空错误

视频课程<狂神聊 Redis>

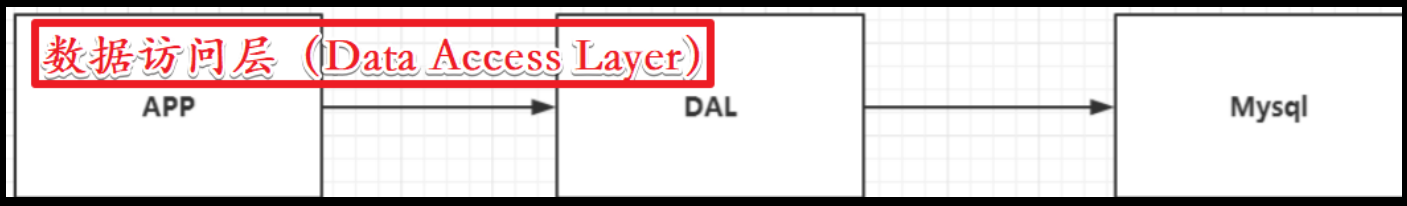
https://www.bilibili.com/video/BV1S54y1R7SB

学习方式：不是为了工作和面试，为了深入了解计算机技术，为了兴趣！

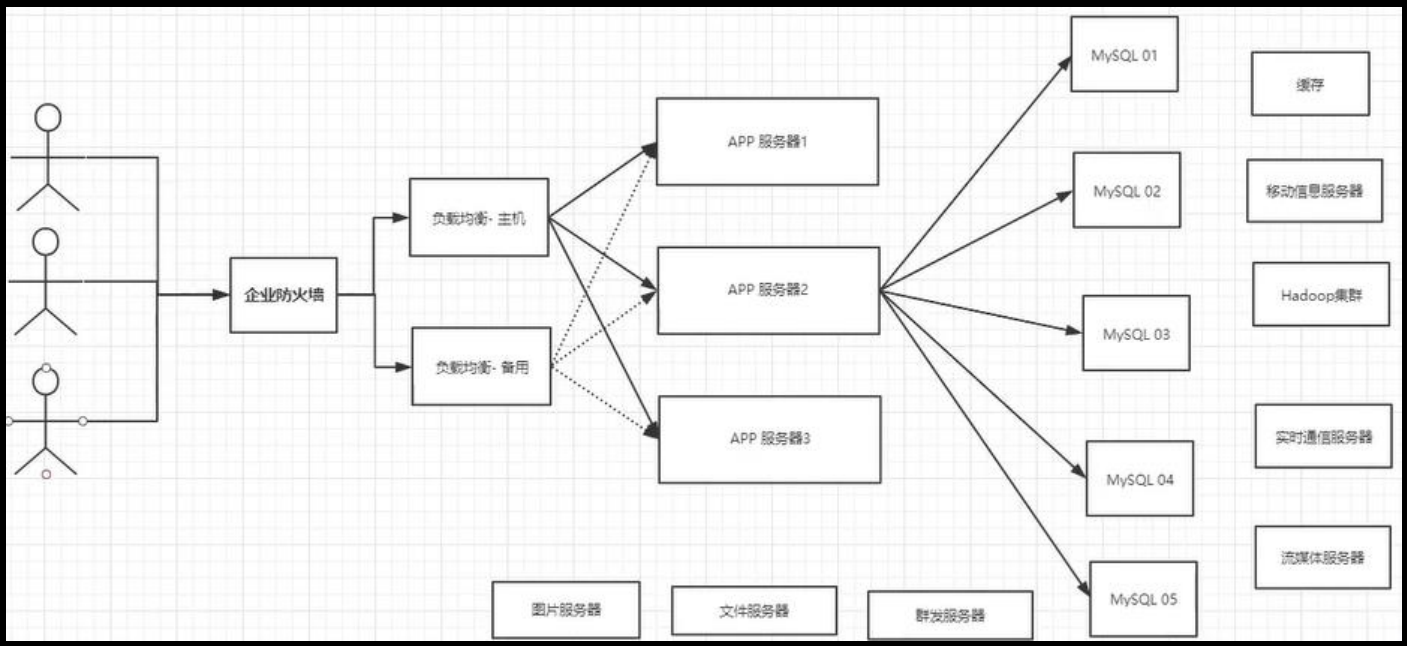
压力一定会越来越大，适者生存！一定要逼着自己学习，这是在这个社会生存的唯一法则！

NoSQL 概述

为什么要使用 NoSQL?



架构：



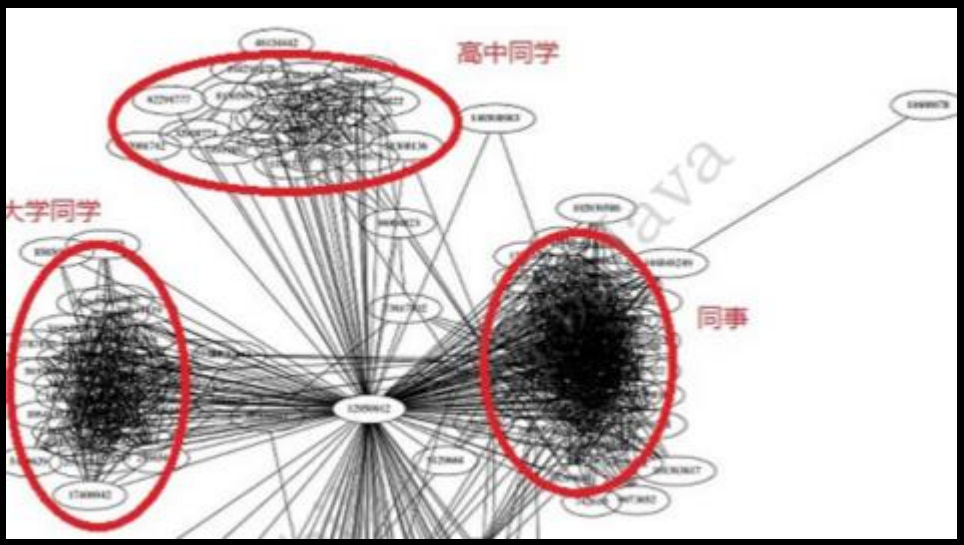
Nosql 特点：

1. 解耦
2. 扩展性

3. 大数据
4. 高性能
5. 数据类型多样

NoSQL 的四大分类

1. **键值对 KV**：redis
2. 文档型数据库：Bson 格式
   a) MongoDB：
      - 基于分布式文件存储的数据库，处理大量的文档
   b) ConthDB
3. **列存储**：
   a) HBase
   b) 分布式文件存储
4. 图形数据库：存储关系的数据库



分类	Examples举例	典型应用场景	数据模型	优点	缺点
键值（key-value）[3]	Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB	内容缓存，主要用于处理大量数据的高访问负载，也用于一些日志系统等等。[3]	Key 指向 Value 的键值对，通常用 hash table来实现[3]	查找速度快	数据无结构化，通常只被当作字符串或者二进制数据[3]
列存储数据库[3]	Cassandra, HBase, Riak	分布式的文件系统	以列簇式存储，将同一列数据存在一起	查找速度快，可扩展性强，更容易进行分布式扩展	功能相对局限
文档型数据库[3]	CouchDB, MongoDb	Web应用（与Key-Value类似，Value是结构化的，不同的是数据库能够了解Value的内容）	Key-Value对应的键值对，Value为结构化数据	数据结构要求不严格，表结构可变，不需要像关系型数据库一样需要预先定义表结构	查询性能不高，而且缺乏统一的查询语法。
图形(Graph)数据库[3]	Neo4J, InfoGrid, Infinite Graph	社交网络，推荐系统等。专注于构建关系图谱	图结构	利用图结构相关算法。比如最短路径寻址，N度关系查找等	很多时候需要对整个图做计算才能得出需要的信息，而且这种结构不太好做分布式的集群方案。[3]

Redis 概述

Redis（Remote Dictionary Server ），即远程字典服务

是一个开源的使用 ANSI C 语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value 数据库，并提供多种语言的 API

Redis 能干嘛？

1、内存存储、持久化，内存中是断电即失、所以说持久化很重要（rdb、aof）
2、效率高，可以用于高速缓存
3、发布订阅系统
4、地图信息分析
5、计时器、计数器（浏览量！）

Redis 推荐都是在 Linux 服务器上搭建的

默认端口：6379

测试性能

Redis-benchmark 是一个压力测试工具

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。	


```
# 测试：100 个并发连接 100000 请求
redis-benchmark -h localhost -p 6379 -c 100 -n 100000
```

Redis 基础知识

Redis 默认有 16 个数据库，默认使用 0 号。

可以使用 select 进行切换数据库！

Redis 为什么默认 16 个数据库？

注意：Redis 支持多个数据库，并且每个数据库的数据是**隔离的不能共享**，并且基于单机才有，如果是**集群就没有数据库的概念**

Redis 是一个字典结构的存储服务器，而实际上一个 Redis 实例提供了多个用来存储

数据的字典，客户端可以指定将数据存储在哪个字典中。这与我们熟知的在一个关系数据库实例中可以创建多个数据库类似，所以可以将其中的每个字典都理解成一个独立的数据库。

这些数据库更像是一种命名空间，而不适宜存储不同应用程序的数据。比如可以使用 0 号数据库存储某个应用生产环境中的数据，使用 1 号数据库存储测试环境中的数据，但不适宜使用 0 号数据库存储 A 应用的数据而使用 1 号数据库 B 应用的数据，不同的应用应该使用不同的 Redis 实例存储数据。

Flushdb:清空当前数据库
FLUSHALL:清空全部数据库

Redis 是单线程的。

Redis 基于内存操作，CPU 不是 redis 的性能瓶颈，而是机器内存和网络带宽，既然可以使用单线程，就使用单线程。

Redis 单线程为什么还这么快?
1、误区 1：高性能的服务器一定是多线程的?
2、误区 2：多线程（CPU 上下文会切换!）一定比单线程效率高!

CPU>内存>硬盘的速度

五大数据类型

字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）与范围查询，bitmaps，hyperloglogs 和 地理空间(geospatial)索引半径查询。

String

```
Keys
Flushdb
Flushall
EXISTS
```

```
 Move key dbNo 移除
 Get
 Expire key exp_sec 设置过期时间
 Type key: 查看类型
 APPEND key append_str: 追加字符串
 Strlen key 字符串长度
 Incr: 自增
 Decr: 自减
 Getrandge
 Setrange
 # setex (set with expire) # 设置过期时间
 # setnx (set if not exist) # 不存在在设置 （在分布式锁中会常常使用!）
mset k1 v1 k2 v2 k3 v3 # 同时设置多个值
mget k1 k2 k3 # 同时获取多个值
 msetnx k1 v1 k4 v4 # msetnx 是一个原子性的操作，要么一起成功，要么一起
 失败!
set user:1 {name:zhangsan,age:3} # 设置一个 user:1 对象 值为 json 字符来保存一个对
象!
 mset user:1:name zhangsan user:1:age 2
 getset db mongodb # 如果存在值，获取原来的值，并设置新的值
 getset db redis # 如果不存在值，则返回 nil
```

List

Redis 里面可以作为栈、队列、阻塞队列
所有的 list 命令都是用 l 开头的，Redis 不区分大小命令
List 可以存在重复值

LRANGE list 0 -1 获取 list

Lpush:从左边 push 进去
Rpush:从右边 push 进去

Lpop:从左边移除
Rpop:从右边移除

Lindex:从左边获取下标 index 的值
Rindex:从右边获取下标 index 的值

Lrem list num val: 从 list 中移除 num 个 val
没有 rrem

rpoplpush # 移除列表的最后一个元素，将他移动到新的列表中!
rpoplpush mylist myotherlist

ltrim mylist 1 2 # 通过下标截取指定的长度，这个 list 已经被改变了，截断了只剩下截取的元素！

lset 将列表中指定下标的值替换为另外一个值，更新操作
lset list 0 item # 如果不存在列表我们去更新就会报错 如果存在，更新当前下标的值

LINSERT mylist after world new

Set

Set: 值不能重复

命令开头都是 s

```
sadd myset "hello"
SMEMBERS myset # 查看指定 set 的所有值
SISMEMBER myset hello # 判断某一个值是不是在 set 集合中！
scard myset # 获取 set 集合中的内容元素个数！
srem myset hello # 移除 set 集合中的指定元素
SRANDMEMBER myset # 随机抽选出一个元素
SRANDMEMBER myset 2 # 随机抽选出指定个数的元素
spop myset # 随机删除一些 set 集合中的元素！
smove myset myset2 "kuangshen" # 将一个指定的值，移动到另外一个 set 集合！

SDIFF key1 key2 # 差集
SINTER key1 key2 # 交集 共同好友就可以这样实现
SUNION key1 key2 # 并集
```

Hash

```
set myhash field kuangshen
```

```
hset myhash field1 kuangshen # set 一个具体 key-vlaue
hget myhash field1 # 获取一个字段值
hmset myhash field1 hello field2 world # set 多个 key-val
hmget myhash field1 field2 # 获取多个字段值
hgetall myhash # 获取全部的数据（kvkv…格式显示）
hdel myhash field1 # 删除 hash 指定 key 字段！对应的 value 值也就消失了！
hlen myhash # 获取 hash 表的字段数量
HEXISTS myhash field1 # 判断 hash 中指定字段是否存在！
hkeys myhash # 只获得所有 key
hvals myhash # 只获得所有 value

HINCRBY myhash field3 1
```

hash 变更的数据 user name age,尤其是是用户信息之类的，经常变动的信息！ hash 更适合于对象的存储，String 更加适合字符串存储！

zset

有序集合：在 set 基础上增加了一个用于排序的值

Zadd zsetName sortBy value （根据 sortBy 排序）

```
zadd myset 1 one # 添加一个值，
zadd myset 2 two 3 three # 添加多个值
ZRANGE myset 0 -1
ZRANGEBYSCORE salary -inf +inf # 显示全部的用户 从小到大！
ZREVRANGE salary 0 -1 # 从大到进行排序！
ZRANGEBYSCORE salary -inf +inf withscores # 显示全部的用户并且附带成绩

zrem salary xiaohong # 移除有序集合中的指定元素
zcard salary # 获取有序集合中的个数
zcount myset 1 3 # 获取指定区间的成员数量！
```

Geospatial 地理位置

Redis 支持三种特殊数据类型

朋友的定位，附近的人，打车距离计算？

在 Redis3.2 版本就推出了！ 这个功能可以推算地理位置的信息，两地之间的距离，方圆几里的人！

6 个命令：

- **GEOADD**
- GEODIST
- GEOHASH
- GEOPOS
- GEORADIUS
- GEORADIUSBYMEMBER

```
    # getadd 添加地理位置
    # 规则：两极无法直接添加，我们一般会下载城市数据，直接通过 java 程序一次性导入！
    # 有效的经度从-180 度到 180 度。
    # 有效的纬度从-85.05112878 度到 85.05112878 度。
    # 当坐标位置超出上述指定范围时，该命令将会返回一个错误。
    # 127.0.0.1:6379> geoadd china:city 39.90 116.40 beijin
```

```
    # 参数 key 值（）
    127.0.0.1:6379> geoadd china:city 116.40 39.90 beijing
    (integer) 1
    127.0.0.1:6379> geoadd china:city 121.47 31.23 shanghai
    (integer) 1
    127.0.0.1:6379> geoadd china:city 106.50 29.53 chongqi 114.05 22.52
shengzhen
    (integer) 2
    127.0.0.1:6379> geoadd china:city 120.16 30.24 hangzhou 108.96 34.26 xian
    (integer) 2
```

GEOPOS china:city beijing # 获取指定的城市的经度和纬度！

GEODIST 两人之间的距离！

单位：

1. m 表示单位为米。
2. km 表示单位为千米。
3. mi 表示单位为英里。
4. ft 表示单位为英尺

```
    GEODIST china:city beijing shanghai km # 查看上海到北京的直线距离
```

georadius 以给定的经纬度为中心， 找出某一半径内的元素

我附近的人？ （获得所有附近的人的地址，定位！）通过半径来查询！

```
    GEORADIUS china:city 110 30 1000 km # 以 110, 30 这个经纬度为中心，寻找方圆 1000km 内的城市
```

GEORADIUSBYMEMBER 找出位于指定元素周围的其他元素！

```
    GEORADIUSBYMEMBER china:city beijing 1000 km
```

GEOHASH 命令 - 返回一个或多个位置元素的 Geohash 表示

该命令将返回 11 个字符的 Geohash 字符串！

```
# 将二维的经纬度转换为一维的字符串，如果两个字符串越接近，那么则距离越近！
127.0.0.1:6379> geohash china:city beijing chongqi
```

GEO 底层的实现原理其实就是 Zset！ 我们可以使用 Zset 命令来操作 geo

```
    ZRANGE china:city 0 -1 # 查看地图中全部的元
    zrem china:city beijing # 移除指定元素！
```

Hyperloglog

基数：简单来说，就是集合中不重复的元素的个数

Hyperloglog ： 基数统计的算法

UV(独立访客) UV 是指不同的、通过互联网访问、浏览一个网页的自然人。

优点：占用的内存是固定，2^64 不同的元素的技术，只需要废 12KB 内存！如果要从内存角度来比较的话 Hyperloglog 首选！

网页的 UV （一个人访问一个网站多次，但是还是算作一个人！）

传统的方式， set 保存用户的 id，然后就可以统计 set 中的元素数量作为标准判断 ！

这个方式如果保存大量的用户 id，就会比较麻烦！我们的目的是为了计数，而不是保存用户 id;

0.81% 错误率！ 统计 UV 任务，可以忽略不计的！

```
    127.0.0.1:6379> PFadd mykey a b c d e f g h i j # 创建第一组元素 mykey
    (integer) 1
    127.0.0.1:6379> PFCOUNT mykey # 统计 mykey 元素的基数数量
    (integer) 10
    127.0.0.1:6379> PFadd mykey2 i j z x c v b n m # 创建第二组元素 mykey2
    (integer) 1
    127.0.0.1:6379> PFCOUNT mykey2
    (integer) 9
    127.0.0.1:6379> PFMERGE mykey3 mykey mykey2 # 合并两组 mykey mykey2 => mykey3 并集
    OK
    127.0.0.1:6379> PFCOUNT mykey3 # 看并集的数量！
```

```
(integer) 15
```

Bitmap

统计用户信息，活跃，不活跃！ 登录 、 未登录！ 打卡，365 打卡！ 两个状态的，都可以使用 Bitmap！
位图

```
Setbit sign 0 1
Getbit sign 2
bitcount sign # 统计
```

事务

ACID 原子性 一致性 隔离性 持久性（Durability）

事务可以一次执行多个命令， 并且带有以下两个重要的保证：
1. 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
2. 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

Redis 事务本质：一组命令的集合！ 一个事务中的所有命令都会被序列化，在事务执行过程的中，会按照顺序执行！

Redis 事务没有没有隔离级别的概念！

redis 的事务：
1. **开启事务：multi**
2. **命令入队（.....）**
3. **执行事务：exec**
4. **取消事务：DISCARD（未提交的命令都不会执行）**

Redis 单条命令式保存原子性的，但是**事务不保证原子性！**
1. **检查型异常：命令入队的时候就报错。这时 exec 后，事务失败，所有的命令都不会执行**

2. 运行期异常：在 exec 提交后，真正执行的时候报错，如 incr 一个字符串。注意，这时，产生运行期异常的命令会失败，但是其他命令执行成功！

WATCH 监控

悲观锁：很悲观，认为什么时候都会出问题，无论做什么都会加锁！
乐观锁：很乐观，认为什么时候都不会出问题，所以不会上锁！ 更新数据的时候去判断一下，在此期间是否有人修改过这个数据（实现：版本控制、CAS…）

WATCH 可以实现乐观锁

测试多线程修改值 ，使用 watch 可以当做 redis 的乐观锁操作！

```
127.0.0.1:6379> watch money # 监视 money
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> DECRBY money 10
QUEUED
127.0.0.1:6379> INCRBY out 10
QUEUED
127.0.0.1:6379> exec # 执行之前，另外一个线程，修改了我们的值，这个时候，就会导致事务执行失
败！
(nil)
```

Watch 怎么实现的？

jedis

Java 可以使用 Jedis 来操作 Redis

是 Redis 官方推荐的 java 连接开发工具！ 使用 Java 操作 Redis 中间件！如果你要使用 java 操作 redis，那么一定要对 Jedis 十分的熟悉！

```
<dependencies>
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
<groupId>redis.clients</groupId>
<artifactId>jedis</artifactId>
<version>3.2.0</version>
</dependency>
<!--fastjson-->
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>fastjson</artifactId>
```

```
    <version>1.2.62</version>
    </dependency>
    </dependencies>

    public class TestPing {
    public static void main(String[] args) {
    // 1、 new Jedis 对象即可
    Jedis jedis = new Jedis("127.0.0.1",6379);
    // jedis 所有的命令就是我们之前学习的所有指令！所以之前的指令学习很重要！
    System.out.println(jedis.ping());
    }
    }
```

SpringBoot 整合

在 SpringBoot2.x 之后，原来使用的jedis 被替换为了 **lettuce**

Jedis 在实现上是直接连接的 Redis Server，如果在**多线程环境下是非线程安全**的。每个线程都去拿自己的 Jedis 实例，当连接数量增多时，资源消耗阶梯式增大，连接成本就较高了。

Lettuce 的连接是基于 Netty 的，Netty 是一个多线程、事件驱动的 I/O 框架。连接实例可以在多个线程间共享，当**多线程使用同一连接实例时，是线程安全**的。
所以，一个多线程的应用可以使用同一个连接实例，而不用担心并发线程的数量。
通过异步的方式可以让我们更好的利用系统资源，而不用浪费线程等待网络或磁盘 I/O。
所以 Lettuce 可以帮助我们充分利用异步的优势。

1. jedis ：采用的**直连**，多个线程操作的话，是不安全的，如果想要避免不安全的，使用 jedis pool 连接池！ 更像 BIO 模式
2. lettuce ： 采用 netty，实例可以再多个线程中进行共享，不存在线程不安全的情况！可以减少线程数据了，更像 NIO 模式

```
    <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>

    # 配置 redis
    spring.redis.host=127.0.0.1
    spring.redis.port=6379
```

两种方式使用：

1. redisTemplate
2. 注解

```
    @Autowired
    private RedisTemplate redisTemplate;
    @Test
    void contextLoads() {
        // redisTemplate 操作不同的数据类型，api 和我们的指令是一样的
        // opsForValue 操作字符串 类似 String
        // opsForList 操作 List 类似 List
        // opsForSet
        // opsForHash
        // opsForZSet
        // opsForGeo
        // opsForHyperLogLog
        // 除了进本的操作，我们常用的方法都可以直接通过 redisTemplate 操作，比如事务，
和基本的 CRUD
        // 获取 redis 的连接对象
        // RedisConnection connection =
        redisTemplate.getConnectionFactory().getConnection();
        // connection.flushDb();
        // connection.flushAll();
        redisTemplate.opsForValue().set("mykey","关注狂神说公众号");
        System.out.println(redisTemplate.opsForValue().get("mykey"));
    }
```

配置：

默认使用的配置 **RedisAutoConfiguration** 类

可以开发自己的配置 RedisConfig

可以封装自己的 Utils

Redis 配置文件详解

Redis.conf 文件

单位：

```
# 1k => 1000 bytes
# 1kb => 1024 bytes
# 1m => 1000000 bytes
# 1mb => 1024*1024 bytes
# 1g => 1000000000 bytes
# 1gb => 1024*1024*1024 bytes
#
# units are case insensitive so 1GB 1Gb 1gB are all the same.
```

包含

```
include /path/to/local.conf
include /path/to/other.conf
```

网络

```
bind 127.0.0.1 # 绑定的 ip
protected-mode yes # 保护模式
port 6379 # 端口设置
```

通用 GENERAL

```
daemonize yes # 以守护进程的方式运行，默认是 no，我们需要自己开启为 yes！
pidfile /var/run/redis_6379.pid # 如果以后台的方式运行，我们就需要指定一个 pid 文件！
```

```
# 日志
# Specify the server verbosity level.
# This can be one of:
# debug (a lot of information, useful for development/testing)
# verbose (many rarely useful info, but not a mess like the debug level)
# notice (moderately verbose, what you want in production probably) 生产环境
# warning (only very important / critical messages are logged)
loglevel notice
logfile "" # 日志的文件位置名
databases 16  # 数据库的数量，默认是 16 个数据库
always-show-logo yes  # 是否总是显示LOGO
```

快照

持久化， 在规定的时间内，执行了多少次操作，则会持久化到文件 .rdb. aof

redis 是内存数据库，如果没有持久化，那么数据断电及失！

```
# 如果900s内，如果至少有一个1 key进行了修改，我们及进行持久化操作
save 900 1
# 如果300s内，如果至少10 key进行了修改，我们及进行持久化操作
save 300 10
# 如果60s内，如果至少10000 key进行了修改，我们及进行持久化操作
save 60 10000
# 我们之后学习持久化，会自己定义这个测试！

stop-writes-on-bgsave-error yes    # 持久化如果出错，是否还需要继续工作！

rdbcompression yes # 是否压缩 rdb 文件，需要消耗一些cpu资源！

rdbchecksum yes # 保存rdb文件的时候，进行错误的检查校验！

dir ./  # rdb 文件保存的目录！
```


REPLICATION 复制

主从复制

SECURITY 安全

```
127.0.0.1:6379> config get requirepass   # 获取redis的密码
1) "requirepass"
2) ""
127.0.0.1:6379> config set requirepass "123456"   # 设置redis的密码
OK
127.0.0.1:6379> config get requirepass   # 发现所有的命令都没有权限了
(error) NOAUTH Authentication required.
127.0.0.1:6379> ping
(error) NOAUTH Authentication required.
127.0.0.1:6379> auth 123456  # 使用密码进行登录!
OK
127.0.0.1:6379> config get requirepass
1) "requirepass"
2) "123456"
```

可以在这里设置redis的密码，默认是没有密码!

限制 CLIENTS

```
maxclients 10000    # 设置能连接上redis的最大客户端的数量

maxmemory <bytes>  # redis 配置最大的内存容量

maxmemory-policy noeviction  # 内存到达上限之后的处理策略
    1、volatile-lru：只对设置了过期时间的key进行LRU（默认值）
    2、allkeys-lru ：  删除lru算法的key
    3、volatile-random：随机删除即将过期key
    4、allkeys-random：随机删除
    5、volatile-ttl ：  删除即将过期的
    6、noeviction ：  永不过期，返回错误
```

APPEND ONLY 模式 aof 配置

```
appendonly no    # 默认是不开启aof模式的，默认是使用rdb方式持久化的，在大部分所有的情况下，rdb完全够用!
appendfilename "appendonly.aof"  # 持久化的文件的名字

# appendfsync always    # 每次修改都会 sync。消耗性能
appendfsync everysec    # 每秒执行一次 sync，可能会丢失这1s的数据!
# appendfsync no        # 不执行 sync，这个时候操作系统自己同步数据，速度最快!
```

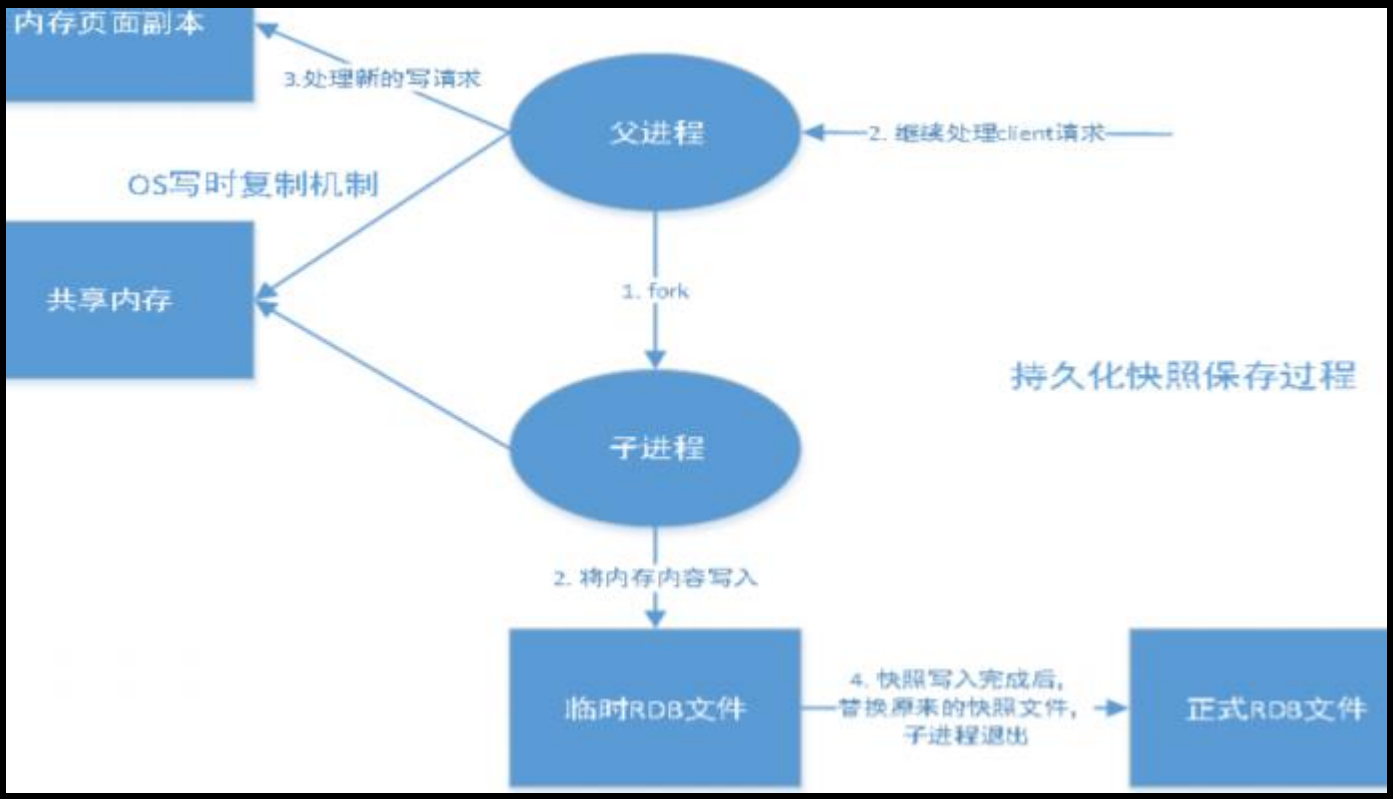
Redis 持久化

内存数据库，如果像 memchached 一样，就会断电即失。
Redis 是持久化的内存数据库

Redis 持久化相关的：
1. RDB
2. AOF

RDB

Redis Database

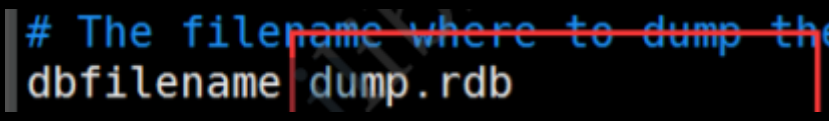


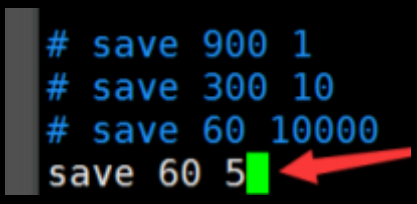
在指定的时间间隔内将内存中的数据集快照写入磁盘(**配置文件中的 save 时间 修改次数**)，也就是行话讲的 Snapshot 快照，它**恢复时是将快照文件直接读到内存里**。

Redis 会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何 IO 操作的。这就确保了极高的性能。如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那 RDB 方式要比 AOF 方式更加的高效。RDB 的缺点是最后一次持久化后的数据可能丢失。我们**默认的就是 RDB**，一般

情况下不需要修改这个配置！

rdb 保存的文件是 dump.rdb 都是在我们的配置文件中快照中进行配置的





触发机制：

1. save 的规则满足的情况下，会自动触发 rdb 规则
2. 执行 flushall 命令，也会触发我们的 rdb 规则！(flushdb 不会触发)
3. 退出 redis(SHUTDOWN)，也会产生 rdb 文件！

备份就自动生成一个 dump.rdb

如果需要恢复 rdb 文件，只需要将 rdb 文件放在我们 redis 启动目录就可以，redis 启动的时候会自动检查 dump.rdb 恢复其中的数据！

查看.rdb 存放的位置：

```
127.0.0.1:6379> config get dir
1) "dir"
2) "/usr/local/bin"  # 如果在这个目录下存在 dump.rdb 文件，启动就会自动恢复其中的数据
```

RDB 优点：

1、适合大规模的数据恢复！
2、对数据的完整性要不高！

缺点：

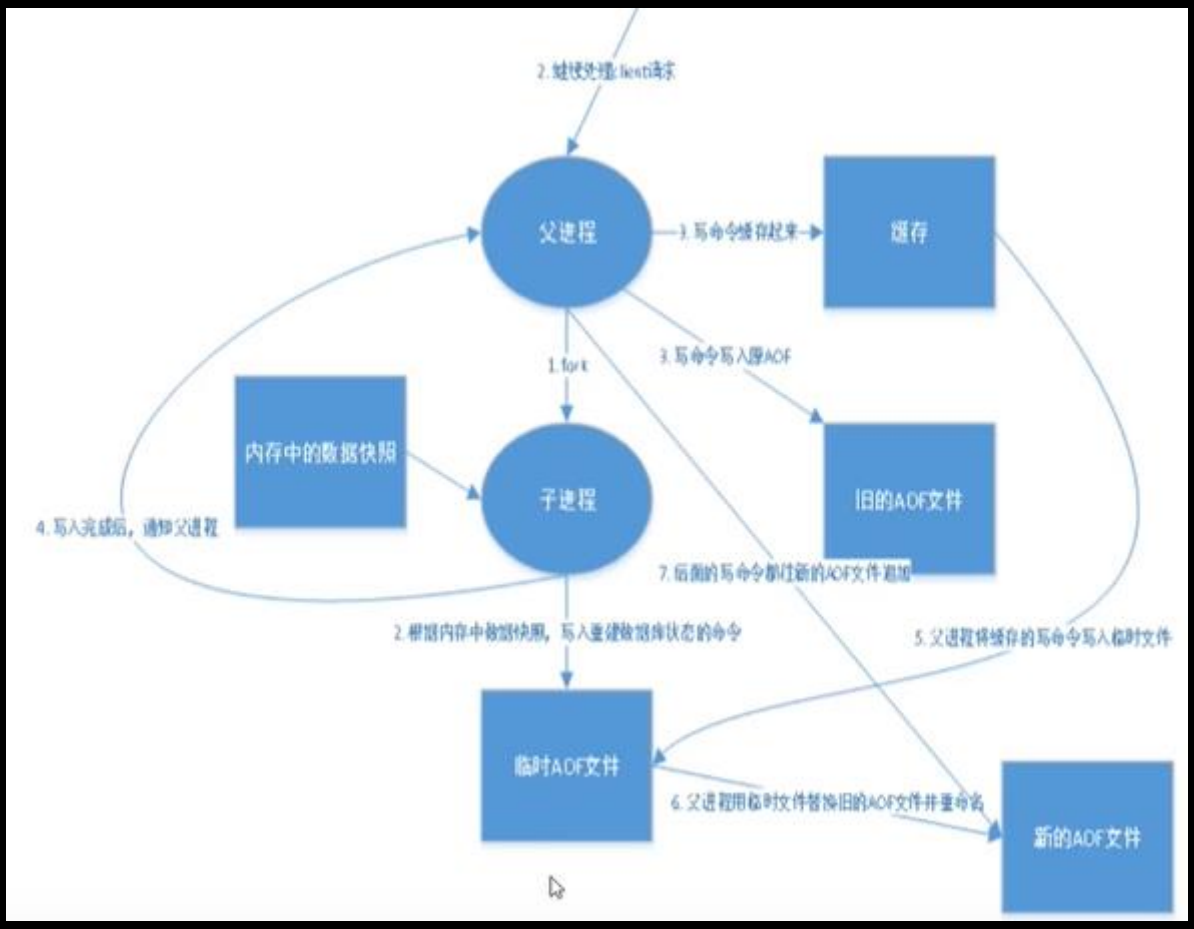
1、需要一定的时间间隔进程操作！如果 redis 意外宕机了，这个最后一次修改数据就没有的了！
2、fork 进程的时候，会占用一定的内容空间

Save 命令保存配置文件，不用重启

AOF

Append Only File

将我们的所有命令(写)都记录下来，history，恢复的时候就把这个文件全部在执行一遍！



以日志的形式来记录每个写操作，将 Redis 执行过的所有指令记录下来（读操作不记录），只许追加文件但不可以改写文件，redis 启动之初会读取该文件重新构建数据，换言之，redis 重启的话就根据日志文件的内容将写指令从前到后执行一次以完成数据的恢复工作

Aof 保存的是 appendonly.aof 文件

```
# Please check http://rec

appendonly no
```

默认是不开启的，我们需要手动进行配置！我们只需要将 appendonly 改为 yes 就开启了 aof

如果这个 aof 文件有错位，这时候 redis 是启动不起来
我们需要修复这个 aof 文件

redis 给我们提供了一个工具 **redis-check-aof --fix**

重写规则说明：
aof 默认就是文件的无限追加，文件会越来越大！

如果 aof 文件大于 64m，太大了！ fork 一个新的进程来将我们的文件进行重写！

```
no-appendfsync-on-rewrite no

# Automatic rewrite of the append only file.
# Redis is able to automatically rewrite the log file implicitly c
# BGREWRITEAOF when the AOF log size grows by the specified percen
#
# This is how it works: Redis remembers the size of the AOF file a
# latest rewrite (if no rewrite has happened since the restart, th
# the AOF at startup is used).
#
# This base size is compared to the current size. If the current s
# bigger than the specified percentage, the rewrite is triggered.
# you need to specify a minimal size for the AOF file to be rewrit
# is useful to avoid rewriting the AOF file even if the percentage
# is reached but it is still pretty small.
#
# Specify a percentage of zero in order to disable the automatic A
# rewrite feature.

auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

如果 aof 文件大于 64m，太大了！ fork 一个新的进程来将我们的文件进行重写！

优点和缺点：
优点：
1、每一次修改都同步，文件的完整会更加好！
2、每秒同步一次，可能会丢失一秒的数据
3、从不同步，效率最高的！
缺点：
1、相对于数据文件来说，aof 远远大于 rdb，修复的速度也比 rdb 慢！
2、Aof 运行效率也要比 rdb 慢，所以我们 redis 默认的配置就是 rdb 持久化

扩展：
1、RDB 持久化方式能够在指定的时间间隔内对你的数据进行快照存储
2、AOF 持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些

命令来恢复原始的数据，AOF 命令以 Redis 协议追加保存每次写的操作到文件末尾，Redis 还能对 AOF 文件进行后台重写，使得 AOF 文件的体积不至于过大。

3、只做缓存，如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化

4、同时开启两种持久化方式

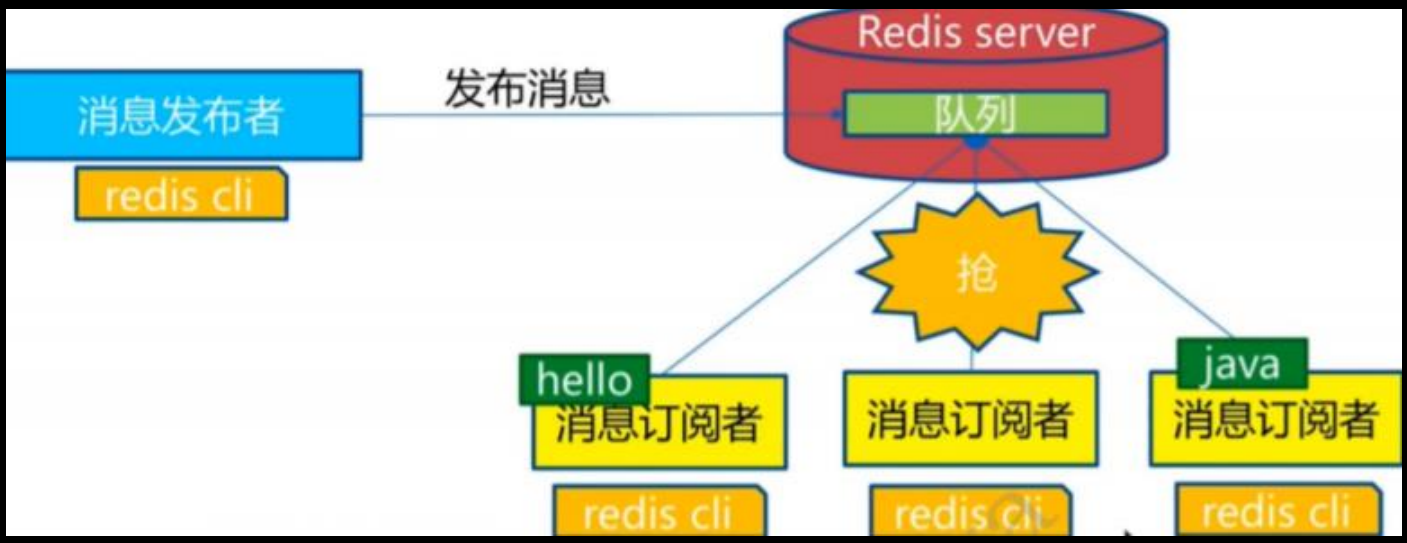
在这种情况下，当 redis 重启的时候会优先载入 AOF 文件来恢复原始的数据，因为在通常情况下 AOF 文件保存的数据集要比 RDB 文件保存的数据集要完整。RDB 的数据不实时，同时使用两者时服务器重启也只会找 AOF 文件，那要不要只使用 AOF 呢？作者建议不要，因为 RDB 更适合用于备份数据库（AOF 在不断变化不好备份），快速重启，而且不会有 AOF 可能潜在的 Bug，留着作为一个万一的手段。

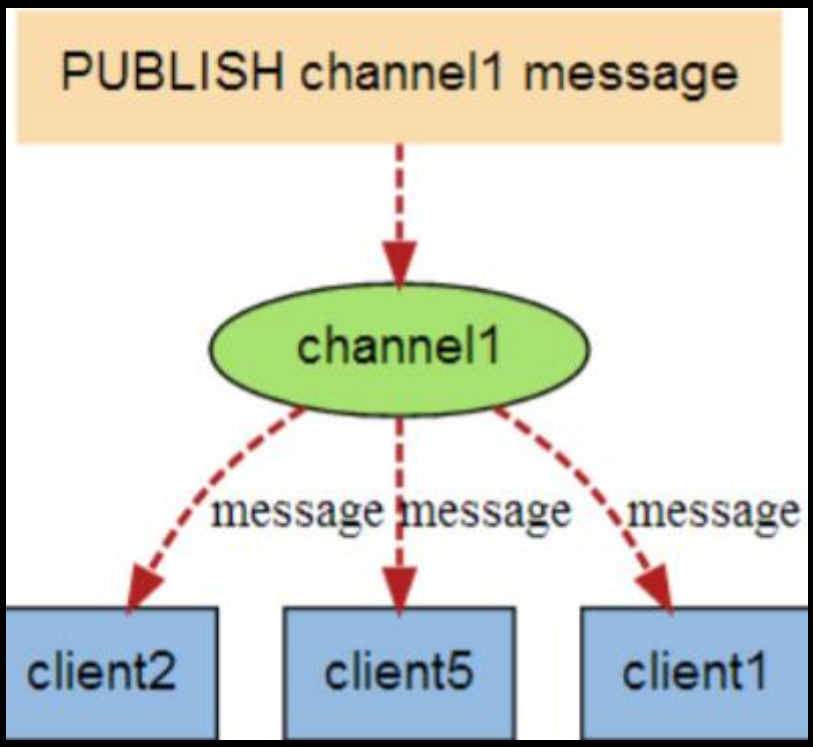
5、性能建议

1. 因为 RDB 文件只用作后备用途，建议**只在 Slave 上持久化 RDB 文件**，而且只要 15 分钟备份一次就够了，只保留 save 900 1 这条规则。
2. 如果 Enable AOF ，好处是在最恶劣情况下也只会丢失不超过两秒数据，启动脚本较简单只 load 自己的 AOF 文件就可以了，代价一是带来了持续的 IO，二是 AOF rewrite 的最后将 rewrite 过程中产生的新数据写到新文件造成的阻塞几乎是不可避免的。只要硬盘许可，应该尽量减少 AOF rewrite 的频率，AOF 重写的基础大小默认值 64M 太小了，可以设到 5G 以上，默认超过原大小 100%大小重写可以改到适当的数值。
3. 如果不 Enable AOF ，仅靠 Master-Slave Replication 实现高可用性也可以，能省掉一大笔 IO，也减少了 rewrite 时带来的系统波动。代价是如果 Master/Slave 同时倒掉，会丢失十几分钟的数据，启动脚本也要比较两个 Master/Slave 中的 RDB 文件，载入较新的那个，微博就是这种架构。

Redis 发布订阅

Pub/sub 发布-订阅


Redis server
消息发布者
发布消息
队列
redis cli
抢
hello
消息订阅者
java
消息订阅者
消息订阅者
redis cli
redis cli
redis cli


PUBLISH channel1 message
channel1
message
message
message
client2
client5
client1

序号	命令及描述
1	PSUBSCRIBE pattern [pattern ...] 订阅一个或多个符合给定模式的频道。
2	PUBSUB subcommand [argument [argument ...]] 查看订阅与发布系统状态。
3	PUBLISH channel message 将信息发送到指定的频道。
4	PUNSUBSCRIBE [pattern [pattern ...]] 退订所有给定模式的频道。
5	SUBSCRIBE channel [channel ...] 订阅给定的一个或多个频道的信息。
6	UNSUBSCRIBE [channel [channel ...]] 指退订给定的频道。

通过 PUBLISH 、SUBSCRIBE 和 PSUBSCRIBE 等命令实现发布和订阅功能。

```
SUBSCRIBE kuangshenshuo # 订阅一个频道 kuangshenshuo

PUBLISH kuangshenshuo "hello,kuangshen" # 发布者发布消息到频道！
```

订阅发布是在 pubsub.c 中实现的

微信：
通过 SUBSCRIBE 命令订阅某频道后，redis-server 里维护了一个字典，**字典的键就是一个个频道！，而字典的值则是一个链表**，链表中保存了所有订阅这个 channel 的客户端。

通过 PUBLISH 命令向订阅者发送消息，redis-server 会使用给定的频道作为键，在它所维护的 channel 字典中查找记录了订阅这个频道的所有客户端的链表，遍历这个链表，将消息发布给所有订阅者。

Pub/Sub 从字面上理解就是发布（Publish）与订阅（Subscribe），在 Redis 中，你可以设定对某一个 key 值进行消息发布及消息订阅，当一个 key 值上进行了消息发布

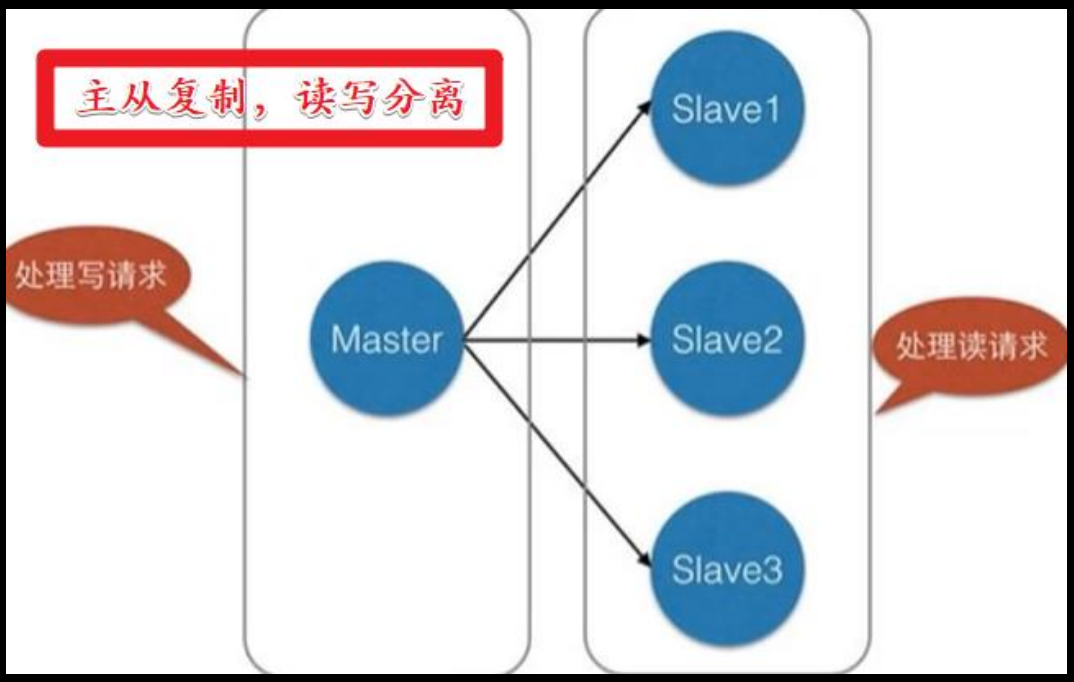
后，所有订阅它的客户端都会收到相应的消息。这一功能最明显的用法就是用作实时消息系统，比如普通的即时聊天，群聊等功能。

使用场景：
1、实时消息系统！
2、事实聊天！（频道当做聊天室，将信息回显给所有人即可！）
3、订阅，关注系统都是可以的！
稍微复杂的场景我们就会使用 消息中间件 MQ

Redis 主从复制

主从复制，是指将一台 Redis 服务器的数据，复制到其他的 Redis 服务器。前者称为**主节点(master/leader)**，后者称为**从节点(slave/follower)**；数据的**复制是单向的，只能由主节点到从节点。**

Master 以写为主，Slave 以读为主。



默认情况下，每台 Redis 服务器都是主节点

且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点

主从复制的作用主要包括：

1. 数据冗余：主从复制实现了数据的热备份，是持久化之外的一种数据冗余方式。
2. 故障恢复：当主节点出现问题时，可以由从节点提供服务，实现快速的故障恢复；实际上是一种服务的冗余。
3. 负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写服务，由从节点提供读服务（即写 Redis 数据时应用连接主节点，读 Redis 数据时应用连接从节点），分担服务器负载；尤其是在写少读多的场景下，通过多个从节点分担读负载，可以大大提高 Redis 服务器的并发量。
4. **高可用（集群）基石**：除了上述作用以外，主从复制还是哨兵和集群能够实施的基础，因此说主从复制是 Redis 高可用的基础

单台 Redis 最大使用内存不应该超过 20G

配置主从复制：

```
127.0.0.1:6379> info replication  # 查看当前库的信息
# Replication
role:master  # 角色  master
connected_slaves:0 #  没有从机
master_replid:b63c90e6c501143759cb0e7f450bd1eb0c70882a
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
repl_backlog_active:0
repl_backlog_size:1048576
repl_backlog_first_byte_offset:0
repl_backlog_histlen:0
```

复制 3 个配置文件，然后修改对应的信息

1、端口

2、pid 名字

3、log 文件名字

4、dump.rdb 名字

默认情况下，每台 Redis 服务器都是主节点； 我们一般情况下**只用配置从机**就好了！

```
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379   # SLAVEOF host 6379   找谁当自己的老大!
OK
127.0.0.1:6380> info replication
# Replication
role:slave  # 当前角色是从机
master_host:127.0.0.1   # 可以的看到主机的信息
master_port:6379
```

slaveOf

Info replication 改为 info 命令

真实的从主配置应该在**配置文件中配置**，这样的话是永久的，我们这里使用的是命令 slaveof，暂时的！

主机可以写，从机不能写只能读！主机中的所有信息和数据，都会自动被从机保存！

测试：主机断开连接，从机依旧连接到主机的，但是没有写操作，这个时候，主机如果回来了，从机依
旧可以直接获取到主机写的信息！

主从复制原理

Slave 启动成功连接到 master 后会发送一个 sync 同步命令
Master 接到命令，启动后台的存盘进程，同时收集所有接收到的用于修改数据集命令，在后台进程执行完毕之后，**master 将传送整个数据文件到 slave，并完成一次完全同步。**

1. **全量复制：** 而 slave 服务在接收到**数据库文件数据**后，将其存盘并加载到内存中。
2. **增量复制：** Master 继续将**新的所有收集到的[修改命令]依次传给 slave**，完成同步

但是只要是重新连接 master，一次完全同步（全量复制）将被自动执行！ 我们的数据一定可以在从机中看到！

主机宕机后可以手动改变主机：

如果主机断开了连接，我们可以使用 SLAVEOF no one 让自己变成主机！其他的节点就可以手动连接到最新的这个主节点（手动）！如果这个时候老大修复了，那就重新连接！

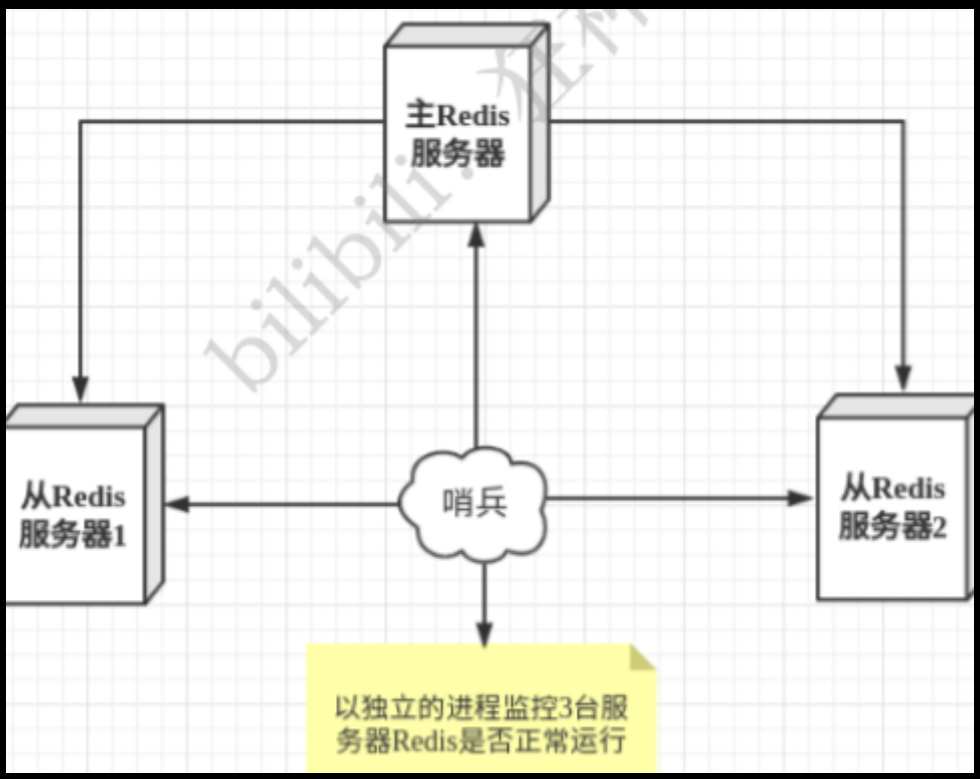
手动替换主节点的方式 不好 ， 自动：哨兵模式

哨兵模式

主从切换技术的方法是：当主服务器宕机后，需要手动把一台从服务器切换为主服务器，这就需要人工干预，费事费力，还会造成一段时间内服务不可用。这不是一种推荐的方式，
更多时候，我们优先考虑**哨兵模式**。Redis 从 2.8 开始正式提供了 Sentinel（哨兵）架构来解决这个问题

Sentinel 能够后台监控主机是否故障，如果故障了根据**投票数**自动将从库转换为主库

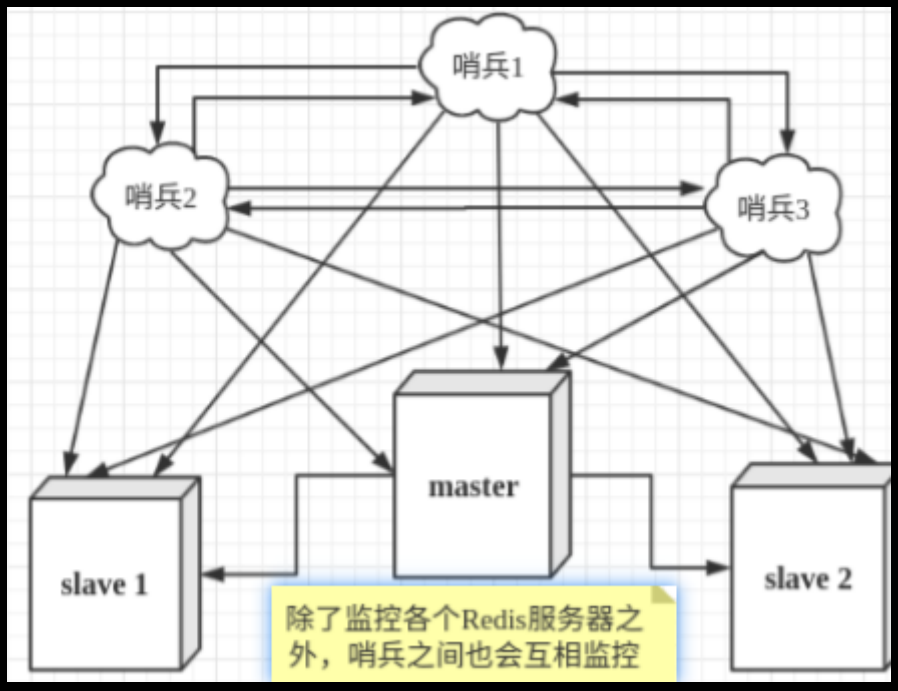
哨兵模式是一种特殊的模式，首先 Redis 提供了哨兵的命令，**哨兵是一个独立的进程**，作为进程，它会独立运行。其原理是**哨兵通过发送命令，等待 Redis 服务器响应，从而监控运行的多个 Redis 实例**



这里的哨兵有两个作用

1. 通过发送命令，让 Redis 服务器返回监控其运行状态，包括主服务器和从服务器。
2. 当哨兵监测到 master 宕机，会自动将 slave 切换成 master，然后通过发布订阅模式通知其他的从服务器，修改配置文件，让它们切换主机

一个哨兵有单点故障——多哨兵



假设主服务器宕机，哨兵 1 先检测到这个结果，系统并不会马上进行 failover 过程，仅仅是哨兵 1 主观的认为主服务器不可用，这个现象成为**主观下线**。当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行 failover[故障转移]操作。切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为**客观下线**

使用 sentinel:

1. 配置哨兵配置文件 sentinel.conf

```
# sentinel monitor 被监控的名称 host  port  1
sentinel monitor myredis 127.0.0.1 6379 1
```

   后面的这个数字 1，代表主机挂了，slave 投票看让谁接替成为主机，票数最多的，就会成为主机！

2. 启动哨兵

   redis-sentinel kconfig/sentinel.conf

failover 后，如果原来的主机此时回来了，只能归并到新的主机下，当做从机，这就是哨兵模式的规则！

优点：

1、哨兵集群，基于主从复制模式，所有的主从配置优点，它全有

2、 主从可以切换，故障可以转移，系统的可用性就会更好

3、哨兵模式就是主从模式的升级，手动到自动，更加健壮！

缺点：

1、Redis **不好在线扩容**，集群容量一旦到达上限，在线扩容就十分麻烦！

2、实现哨兵模式的配置其实是很麻烦的，里面有很多选择！

哨兵模式的全部配置：

```
    # Example sentinel.conf
    # 哨兵 sentinel 实例运行的端口 默认 26379
    port 26379
    # 哨兵 sentinel 的工作目录
    dir /tmp
    # 哨兵 sentinel 监控的 redis 主节点的 ip port
    # master-name 可以自己命名的主节点名字 只能由字母 A-z、数字 0-9 、这三个字符".-_"
组成。
    # quorum 配置多少个 sentinel 哨兵统一认为 master 主节点失联 那么这时客观上认为主节
点失联了
    # sentinel monitor <master-name> <ip> <redis-port> <quorum>
    sentinel monitor mymaster 127.0.0.1 6379 2
    # 当在 Redis 实例中开启了 requirepass foobared 授权密码 这样所有连接 Redis 实例的
客户端都要提供
    密码
    # 设置哨兵 sentinel 连接主从的密码 注意必须为主从设置一样的验证密码
    # sentinel auth-pass <master-name> <password>
    sentinel auth-pass mymaster MySUPER--secret-0123passw0rd
    # 指定多少毫秒之后 主节点没有应答哨兵 sentinel 此时 哨兵主观上认为主节点下线 默认
30 秒
    # sentinel down-after-milliseconds <master-name> <milliseconds>
    sentinel down-after-milliseconds mymaster 30000
    # 这个配置项指定了在发生 failover 主备切换时最多可以有多少个 slave 同时对新的
master 进行 同步，
    这个数字越小，完成 failover 所需的时间就越长，
    但是如果这个数字越大，就意味着越 多的 slave 因为 replication 而不可用。
    可以通过将这个值设为 1 来保证每次只有一个 slave 处于不能处理命令请求的状态。
    # sentinel parallel-syncs <master-name> <numslaves>
    sentinel parallel-syncs mymaster 1
    # 故障转移的超时时间 failover-timeout 可以用在以下这些方面:
    #1. 同一个 sentinel 对同一个 master 两次 failover 之间的间隔时间。
    #2. 当一个 slave 从一个错误的 master 那里同步数据开始计算时间。直到 slave 被纠正为向
正确的 master 那
    里同步数据时。
    #3.当想要取消一个正在进行的 failover 所需要的时间。
    #4.当进行 failover 时，配置所有 slaves 指向新的 master 所需的最大时间。不过，即使过
了这个超时，
    slaves 依然会被正确配置为指向 master，但是就不按 parallel-syncs 所配置的规则来了
    # 默认三分钟
   # sentinel failover-timeout <master-name> <milliseconds>
      sentinel failover-timeout mymaster 180000
    # SCRIPTS EXECUTION
    #配置当某一事件发生时所需要执行的脚本，可以通过脚本来通知管理员，例如当系统运行不正
常时发邮件通知
    相关人员。
    #对于脚本的运行结果有以下规则:
    #若脚本执行后返回 1，那么该脚本稍后将会被再次执行，重复次数目前默认为 10
    #若脚本执行后返回 2，或者比 2 更高的一个返回值，脚本将不会重复执行。
    #如果脚本在执行过程中由于收到系统中断信号被终止了，则同返回值为 1 时的行为相同。
    #一个脚本的最大执行时间为 60s，如果超过这个时间，脚本将会被一个 SIGKILL 信号终止，
之后重新执行。
```

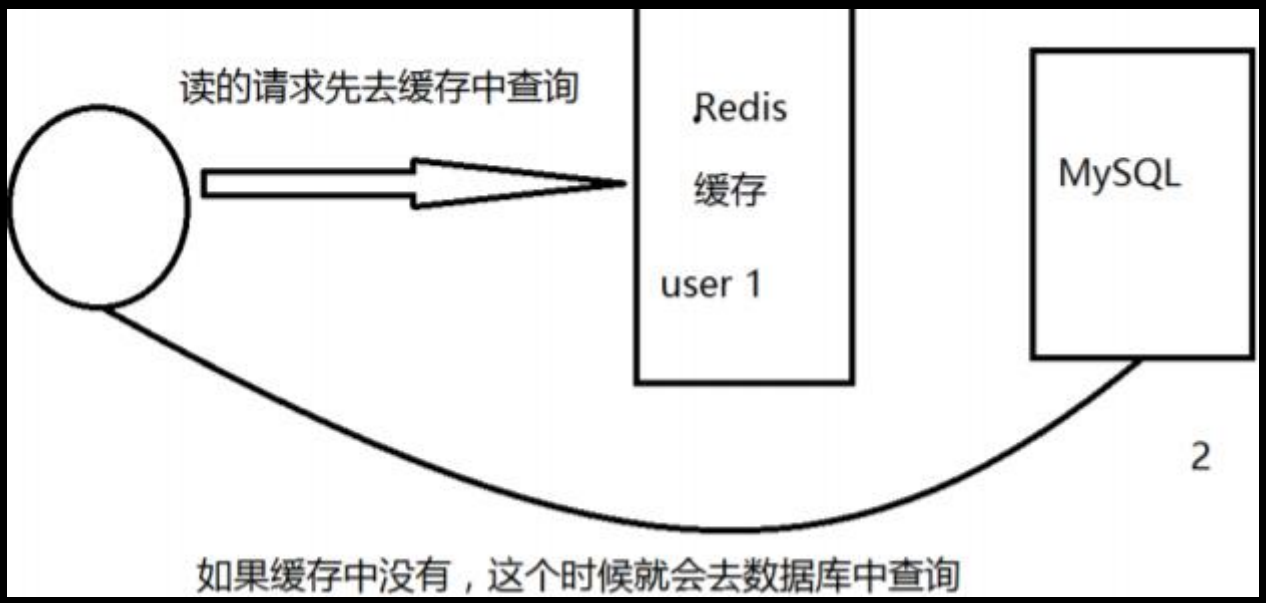
```
    #通知型脚本:当 sentinel 有任何警告级别的事件发生时（比如说 redis 实例的主观失效和客观失效等等),
    将会去调用这个脚本，这时这个脚本应该通过邮件，SMS 等方式去通知系统管理员关于系统不正常运行的信
    息。调用该脚本时，将传给脚本两个参数，一个是事件的类型，一个是事件的描述。如果 sentinel.conf 配
    置文件中配置了这个脚本路径，那么必须保证这个脚本存在于这个路径，并且是可执行的，否则 sentinel 无
    法正常启动成功。
    #通知脚本
    # shell 编程
    # sentinel notification-script <master-name> <script-path>
    sentinel notification-script mymaster /var/redis/notify.sh
    # 客户端重新配置主节点参数脚本
    # 当一个 master 由于 failover 而发生改变时，这个脚本将会被调用，通知相关的客户端关于 master 地址已
    经发生改变的信息。
    # 以下参数将会在调用脚本时传给脚本:
    # <master-name> <role> <state> <from-ip> <from-port> <to-ip> <to-port>
    # 目前<state>总是“failover”,
    # <role>是“leader”或者“observer”中的一个。
    # 参数 from-ip, from-port, to-ip, to-port 是用来和旧的 master 和新的 master(即旧的 slave)通
    信的
    # 这个脚本应该是通用的，能被多次调用，不是针对性的。
    # sentinel client-reconfig-script <master-name> <script-path>
    sentinel client-reconfig-script mymaster /var/redis/reconfig.sh # 一般都是由运维来配置!
```

Redis 缓存穿透和雪崩

服务的高可用问题

Redis 缓存的使用，极大的提升了应用程序的性能和效率，特别是数据查询方面。但同时，它也带来了一些问题。其中，最要害的问题，就是**数据的一致性问题**，从严格意义上讲，这个问题无解。**如果对数据的一致性要求很高，那么就不能使用缓存**

另外的一些典型问题就是，缓存穿透、缓存雪崩和缓存击穿。目前，业界也都有比较流行的解决方案



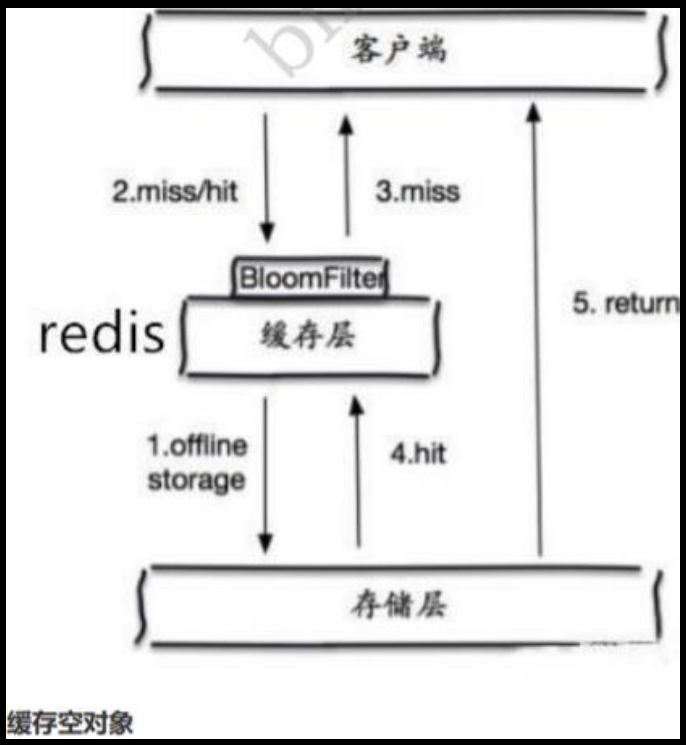
缓存穿透

缓存穿透的概念很简单，用户想要查询一个数据，发现 redis 内存数据库没有，也就是缓存没有命中，于是向持久层数据库查询。发现也没有，于是本次查询失败。当用户很多的时候，缓存都没有命中（秒杀!），于是都去请求了持久层数据库。这会给持久层数据库造成很大的压力，这时候就相当于出现了缓存穿透。

解决方案：

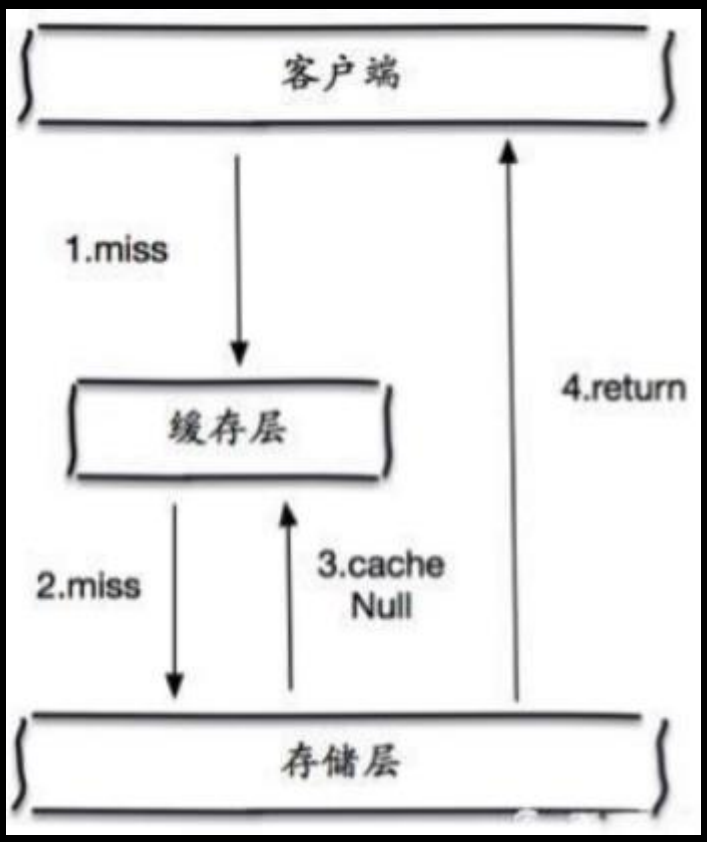
1. 布隆过滤器

   布隆过滤器是一种数据结构，对所有可能查询的参数以 hash 形式存储，在控制层先进行校验，不符合则丢弃，从而避免了对底层存储系统的查询压力；



缓存空对象

当存储层不命中后，**即使返回的空对象也将其缓存起来**，同时会设置一个过期时间，之后再访问这个数据将会从缓存中获取，保护了后端数据源



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。

解决方案：

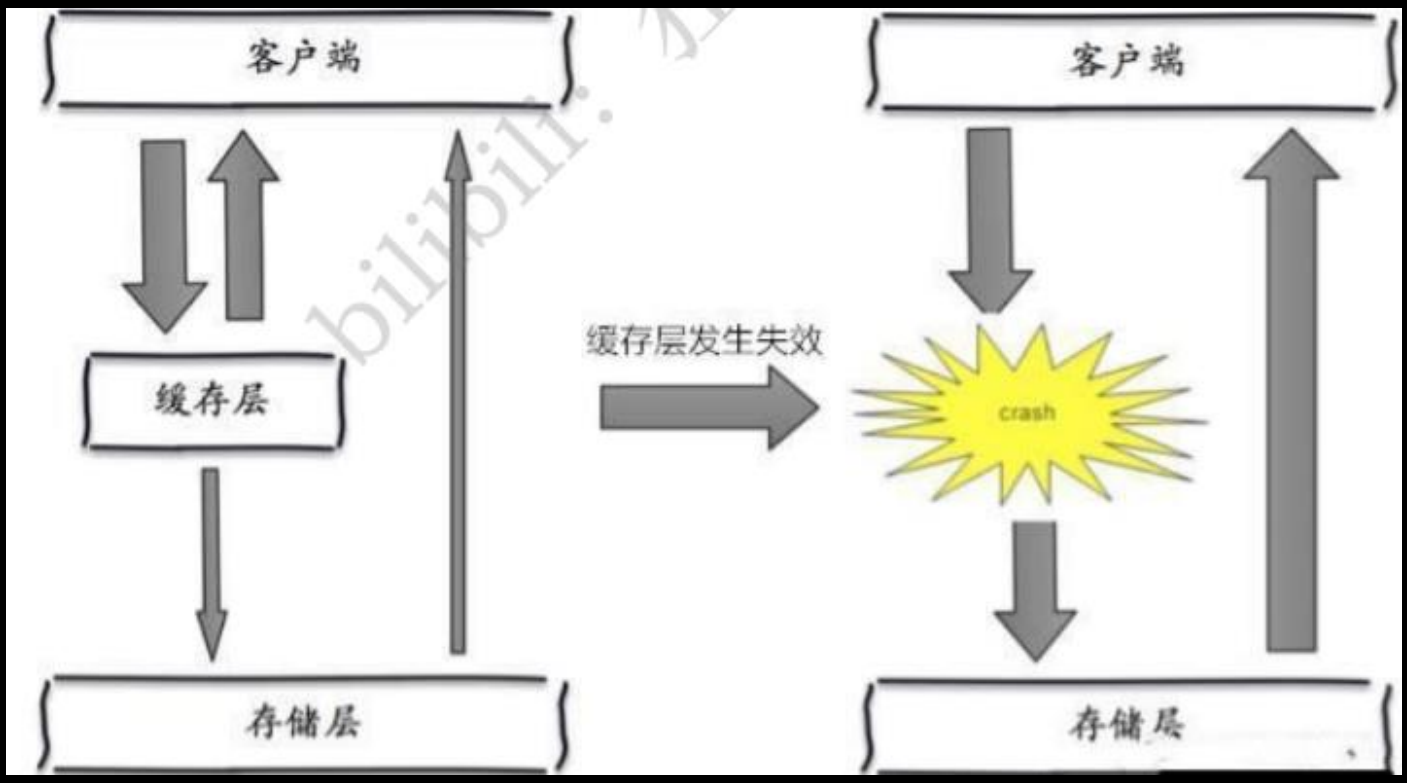
1. 设置热点数据永不过期
2. 加互斥锁：

   分布式锁：使用分布式锁，保证对于**每个 key 同时只有一个线程去查询后端服务，**其他线程没有获得分布式锁的权限，因此只需要等待即可。这种方式将高并发的压力转移到了分布式锁，因此对分布式锁的考验很大。

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。Redis 宕机

11.11 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。于是所有的请求都会达到存储层，存储层的调用量会暴增，造成存储层也会挂掉的情况。



其实集中过期，倒不是非常致命，比较致命的缓存雪崩，是缓存服务器某个节点宕机或断网。因为自然形成的缓存雪崩，一定是在某个时间段集中创建缓存，这个时候，数据库也是可以顶住压力的。无非就是对数据库产生周期性的压力而已。而缓存服务节点的宕机，对数据库服务器造成的压力是不可预知的，很有可能瞬间就把数据库压垮。

解决方案

1. redis 高可用

这个思想的含义是，既然 redis 有可能挂掉，那我多增设几台 redis，这样一台挂掉之后其他的还可以继续工作，其实就是搭建的集群。（异地多活！）

2. **限流降级**（在 SpringCloud 讲解过！）

这个解决方案的思想是，在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个 key 只允许一个线程查询数据和写缓存，其他线程等待。

3. 数据预热：数据加热的含义就是在正式部署之前，我先把可能的数据先预先访问一遍，这样部分可能大量访问的数据就会加载到缓存中。在即将发生大并发访问前手动触发加载缓存不同的 key，**设置不同的过期时间，让缓存失效的时间点尽量均匀**

其他知识

所有牛逼的人都有一段苦逼的岁月！但是你只要像 SB 一样的去坚持，终将牛逼！

敬畏之心可以使人进步

社会目前初级和中级程序员饱和、高级程序员重金难求，提升自己！

PV、TPS、QPS

- PV=page view 页面被浏览的次数

- TPS=transactions per second 每秒内的事务数，比如执行了 dml 操作
- QPS=queries per second 每秒内查询次数，比如执行了 select 操作，相应的 qps 会增加。
- RPS=requests per second
- RPS=并发数/平均响应时间

CAP 与 BASE 理论

大数据的 3V+3 高

1. 海量Volume
2. 多样Variety
3. 实时Velocity

1. 高并发
2. 高可拓
3. 高性能

Spring 事务？？？

社会目前初级和中级程序员饱和，高级程序员重金难求，提升自己！

Edwin Xu