Redis 学习笔记

Redis 是什么

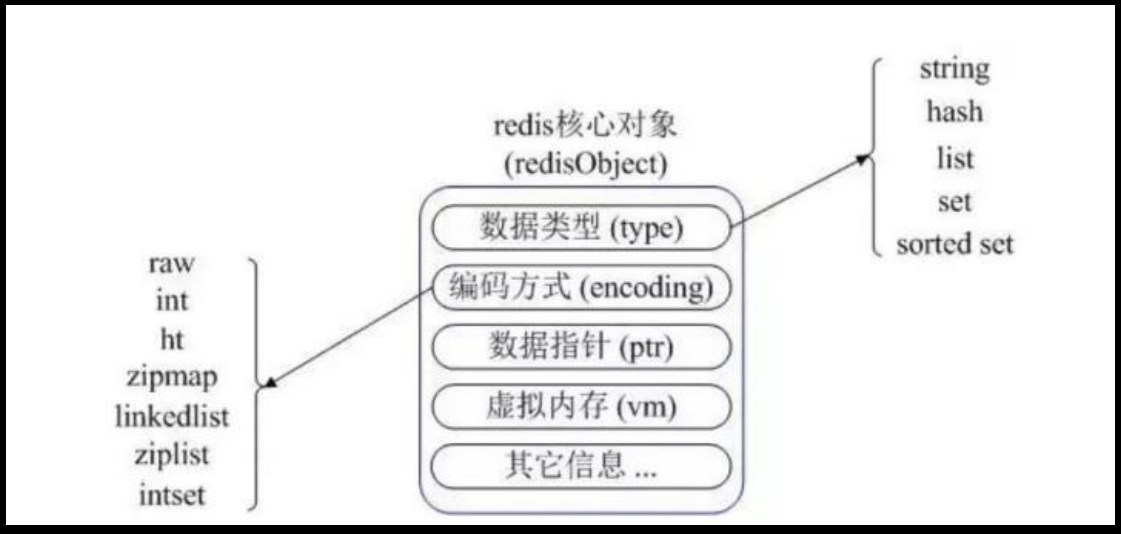
Redis 是 C 语言开发的一个开源的（遵从 BSD 协议）高性能键值对（key-value）的内存数据库，可以用作数据库、缓存、消息中间件等。它是一种 NoSQL（not-only sql，泛指非关系型数据库）的数据库。

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

zset 底层实现原理

和上面的集合对象相比，有序集合对象是有序的。与列表使用索引下标作为排序依据不同，有序集合为每个元素设置一个分数（score）作为排序依据。

有序集合的编码可以是 **ziplist 或者 skiplist**。

ziplist 编码的有序集合对象使用压缩列表作为底层实现，每个集合元素使用两个紧挨在一起的压缩列表节点来保存，第一个节点保存元素的成员，第二个节点保存元素的分值。并且压缩列表内的集合元素按分值从小到大的顺序进行排列，小的放置在靠近表头的位置，大的放置在靠近表尾的位置。

skiplist 编码的有序集合对象使用 zet 结构作为底层实现，一个 zset 结构同时包含一个字典和一个跳跃表：

```
typedef struct zset{
    //跳跃表
    zskiplist *zsl;
    //字典
    dict *dice;
} zset;
```

字典的键保存元素的值，字典的值则保存元素的分值；跳跃表节点的 object 属性保存元素的成员，跳跃表节点的 score 属性保存元素的分值。

查询：知道元素，查 hash 得到 score，然后根据 score 查询 skiplist，得到对应的 Object

这两种数据结构会通过指针来共享相同元素的成员和分值，所以不会产生重复成员和分值，造成内存的浪费。

说明：其实有序集合单独使用字典或跳跃表其中一种数据结构都可以实现，但是这里使用两种数据结构组合起来，原因是假如我们单独使用 字典，虽然能以 O(1) 的时间复杂度查找成员的分值，但是因为字典是以无序的方式来保存集合元素，所以每次进行范围操作的时候都要进行排序；假如我们单独使用跳跃表来实现，虽然能执行范围操作，但是查找操作有 O(1)的复杂度变为了 O(logN)。因此 Redis 使用了两种数据结构来共同实现有序集合。

编码转换

当有序集合对象同时满足以下两个条件时，对象使用 ziplist 编码：

1、保存的元素数量小于 128；

2、保存的所有元素长度都小于 64 字节。

不能满足上面两个条件的使用 skiplist 编码。

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。
  - @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

```java
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

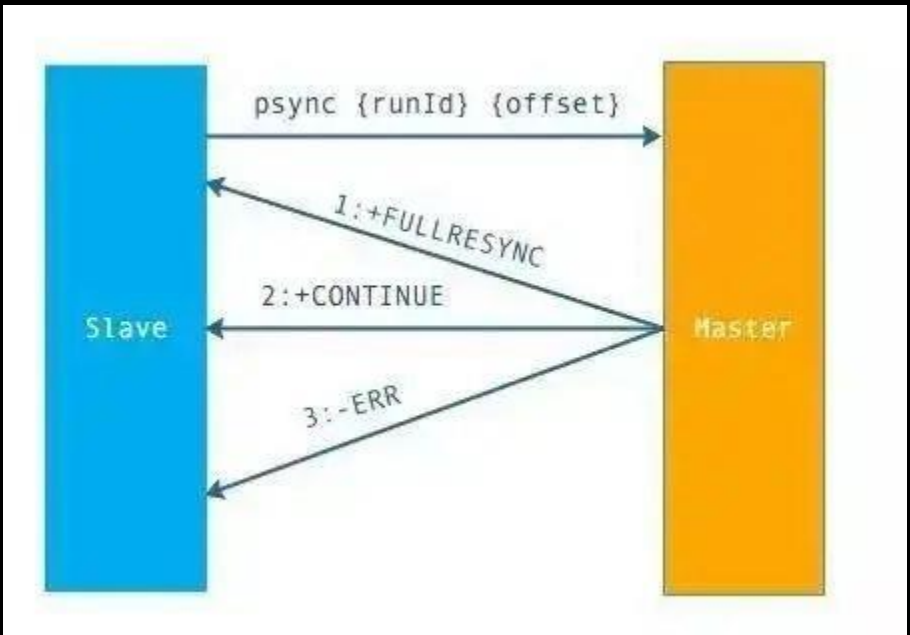
redis 单节点存在单点故障问题，为了解决单点问题，一般都需要对 redis 配置从节点，然后使用哨兵来监听主节点的存活状态，如果主节点挂掉，从节点能继续提供缓存功能

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。

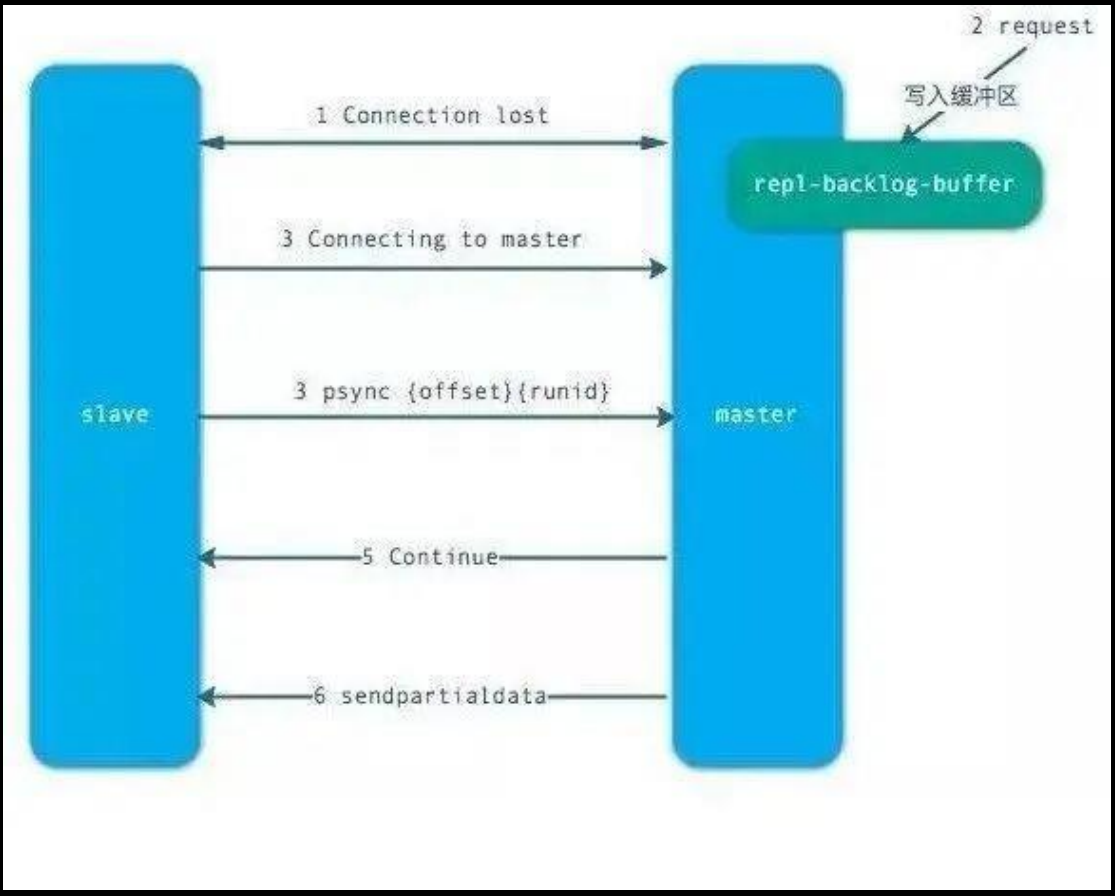
数据同步的过程：
redis2.8 之前使用 sync[runId][offset] 同步命令，redis2.8 之后使用 psync[runId][offset]命令。两者不同在于，sync 命令仅支持全量复制过程，psync 支持全量和部分复制。
几个概念：
1. runId：每个 redis 节点启动都会生成唯一的 uuid，每次 redis 重启后，runId 都会发生变化。
2. offset：主节点和从节点都各自维护自己的主从复制偏移量 offset，当主节点有写入命令时，offset=offset+命令的字节长度。从节点在收到主节点发送的命令后，也会增加自己的 offset，并把自己的 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。

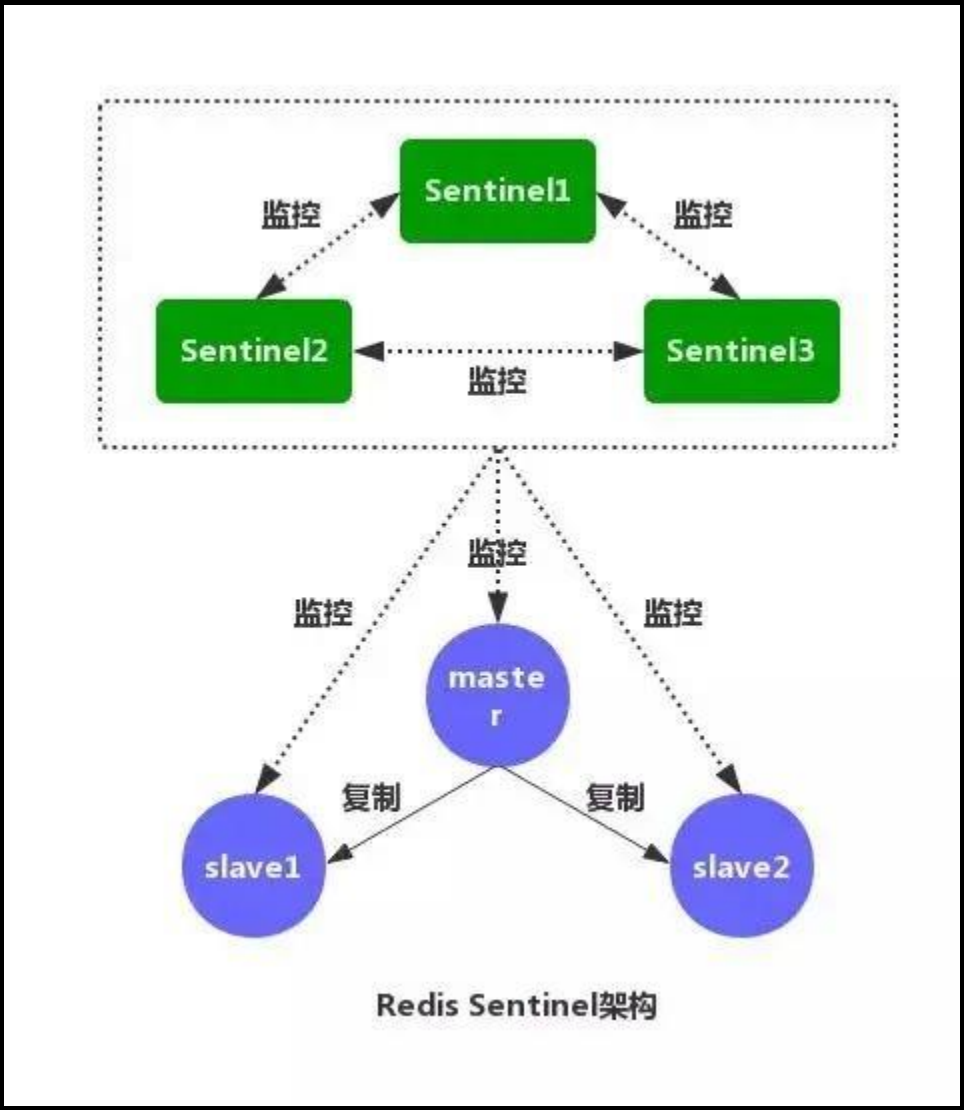
部分复制有以下几点说明：
1、部分复制主要是 Redis 针对全量复制的过高开销做出的一种优化措施，使用 psync[runId][offset]命令实现。当从节点正在复制主节点时，如果出现网络闪断或者命令丢失等异常情况时，从节点会向主节点要求补发丢失的命令数据，主节点的复制积压缓冲区将这部分数据直接发送给从节点，这样就可以保持主从节点复制的一致性。补发的这部分数据一般远远小于全量数据。
2、主从连接中断期间主节点依然响应命令，但因复制连接中断命令无法发送给从节点，不过主节点内的复制积压缓冲区依然可以保存最近一段时间的写命令数据。
3、当主从连接恢复后，由于从节点之前保存了自身已复制的偏移量和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

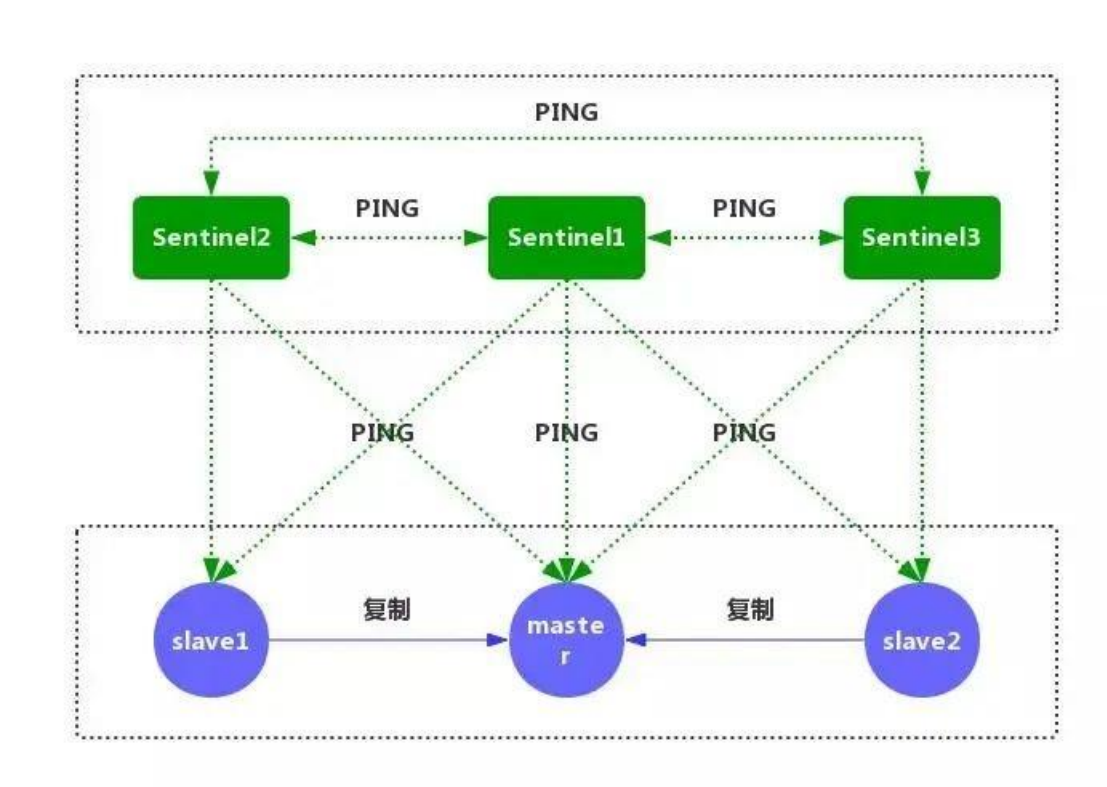


主要功能包括主节点存活检测、主从运行情况检测、自动故障转移、主从切换。
Redis Sentinel 最小配置是一主一从。

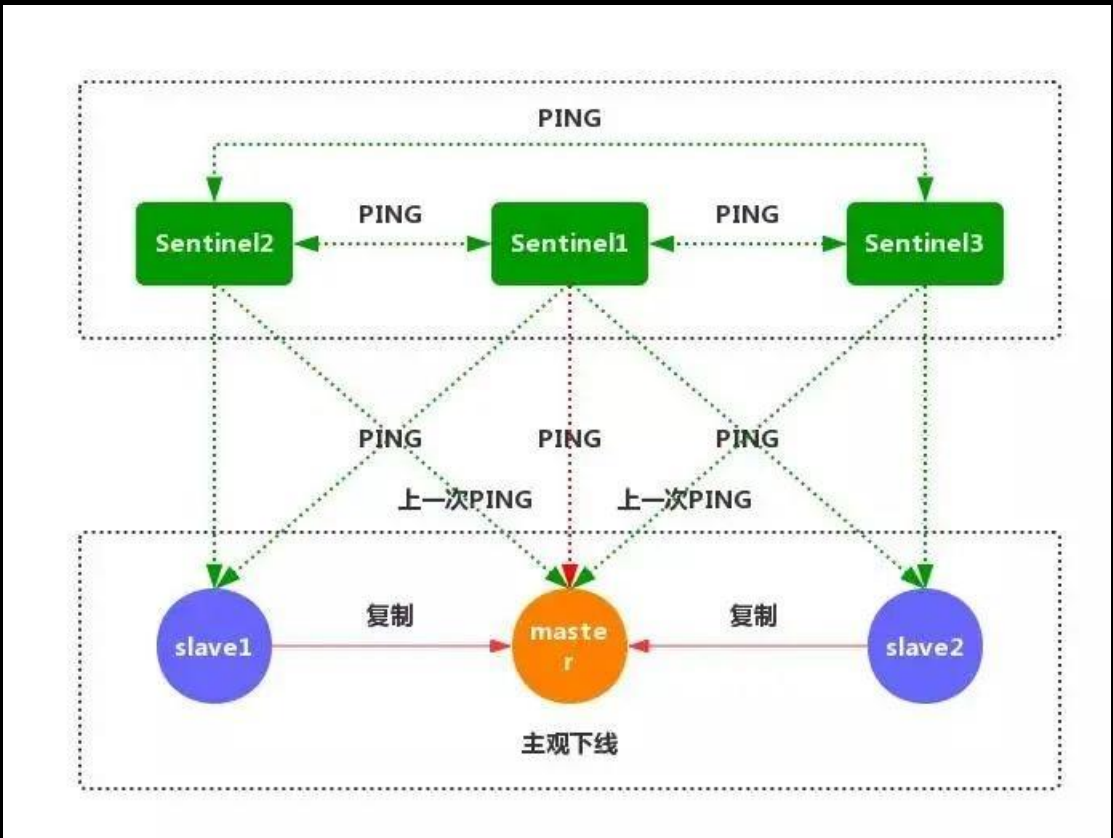
Redis 的 Sentinel 系统可以用来管理多个 Redis 服务器，该系统可以执行以下四个任务：
1、监控：不断检查主服务器和从服务器是否正常运行。
2、通知：当被监控的某个 redis 服务器出现问题，Sentinel 通过 API 脚本向管理员或者其他应用程序发出通知。
3、自动故障转移：当主节点不能正常工作时，Sentinel 会开始一次自动的故障转移操作，它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点，并且将其他的从节点指向新的主节点，这样人工干预就可以免了。
4、配置提供者：在 Redis Sentinel 模式下，客户端应用在初始化时连接的是 Sentinel 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：



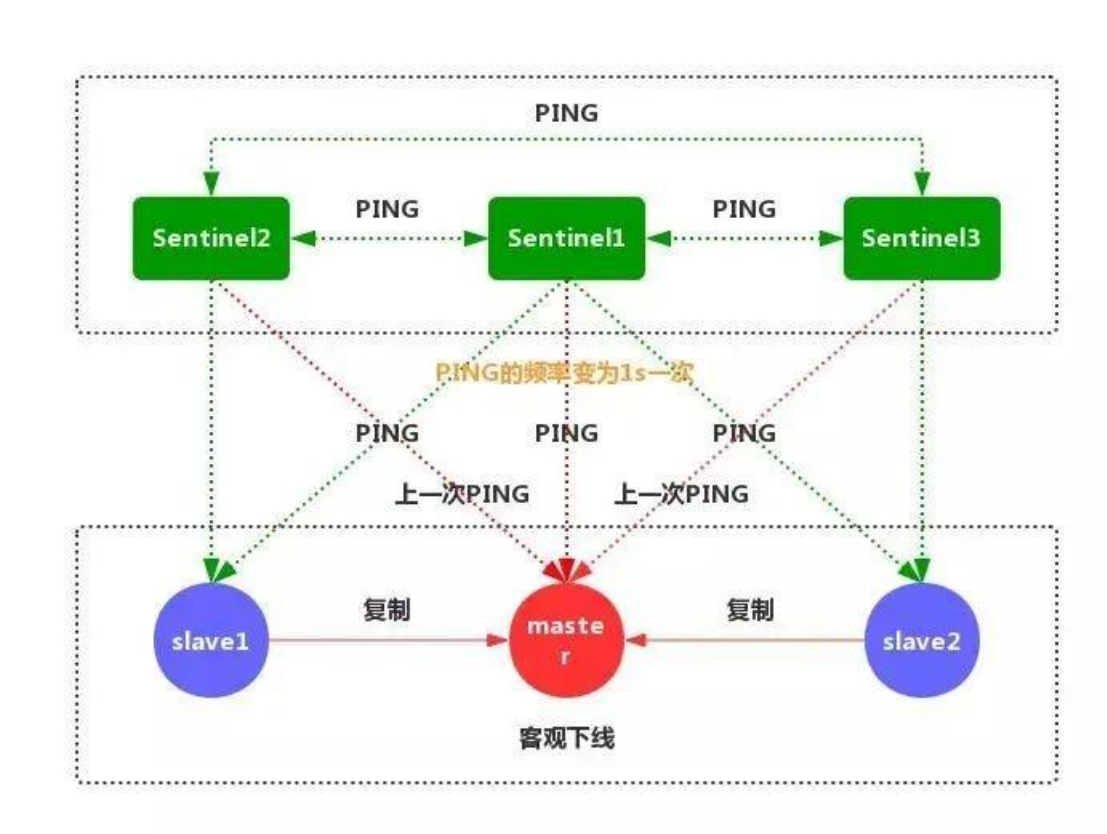
2. 如果一个实例距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 所指定的值，那么这个实例会被 Sentinel 标记为主观下线。



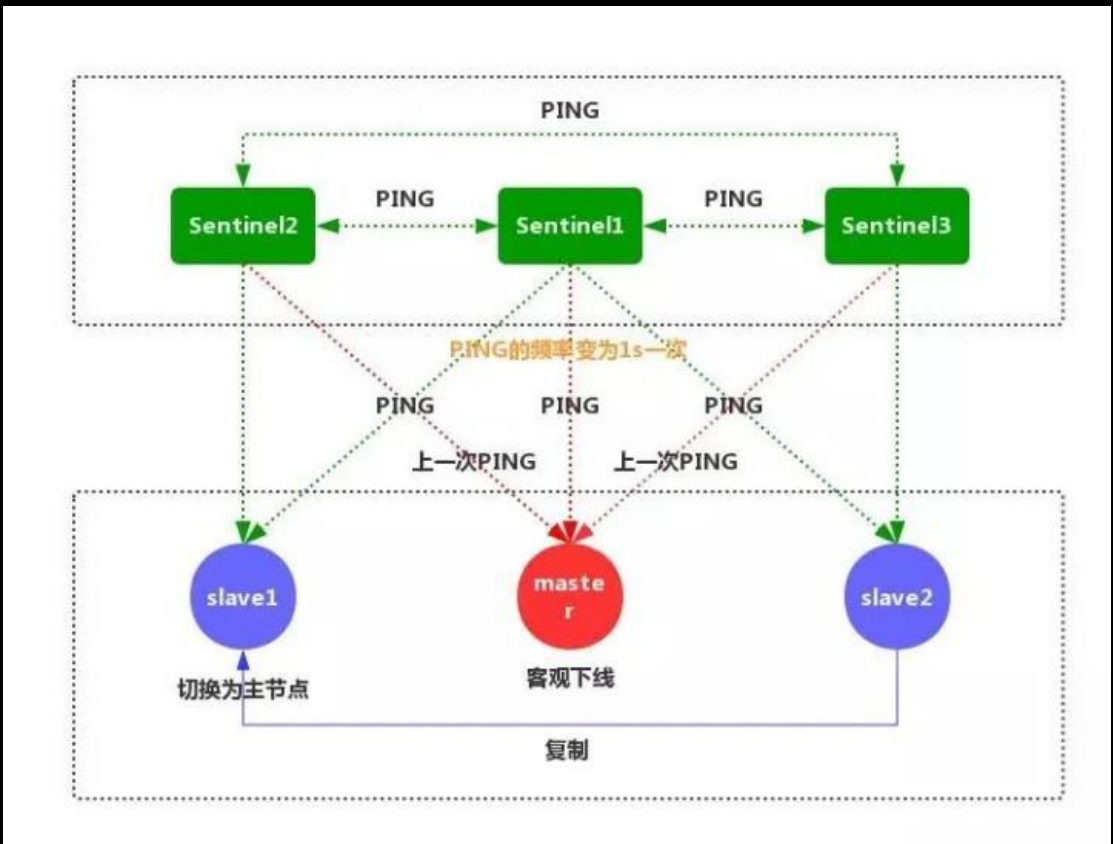
3. 如果一个主服务器被标记为主观下线，那么正在监视这个服务器的所有 Sentinel 节点，要以每秒一次的频率确认主服务器的确进入了主观下线状态。

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。

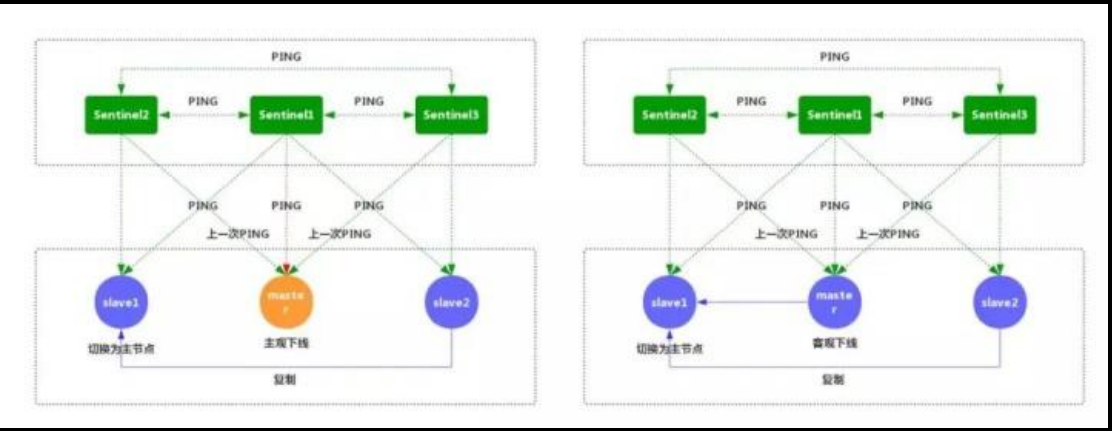


5. 一般情况下，每个 Sentinel 会以每 10 秒一次的频率向它已知的所有主服务器和从服务器发送 INFO 命令，当一个主服务器被标记为客观下线时，Sentinel 向下线主服务器的所有从服务器发送 INFO 命令的频率，会从 10 秒一次改为每秒一次。



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。

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

Redis 命令

获取键总数
语法: dbsize

查询键是否存在
语法: exists key [key ...]

查询键类型
语法: type key

查询 key 的生命周期（秒）
秒语法：ttl key
毫秒语法：pttl key
-1：永远不过期。

设置过期时间
秒语法：expire key seconds
毫秒语法：pexpire key milliseconds

设置永不过期
语法：persist key

更改键名称
语法：rename key newkey

获取键值
语法：get key

```
127.0.0.1:6379> get redis_key
"redis_vale"
```

值递增/递减
如果字符串中的值是数字类型的，可以使用 incr 命令每次递增，不是数字类型则报错。
语法：incr key

获取值长度
语法：strlen key

追加内容
语法：append key value

https://zhuanlan.zhihu.com/p/47692277

应用

SpringBoot + Redis

Redis 是完全开源免费的，遵守 BSD 协议，是一个高性能的 key-value 非关系性数据库(NoSql)。

Redis 与其他 key - value 缓存产品有以下三个特点:

1. Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
2. Redis 不仅仅支持简单的 key-value 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。
3. Redis **支持数据的备份**，即 master-slave 模式的数据备份

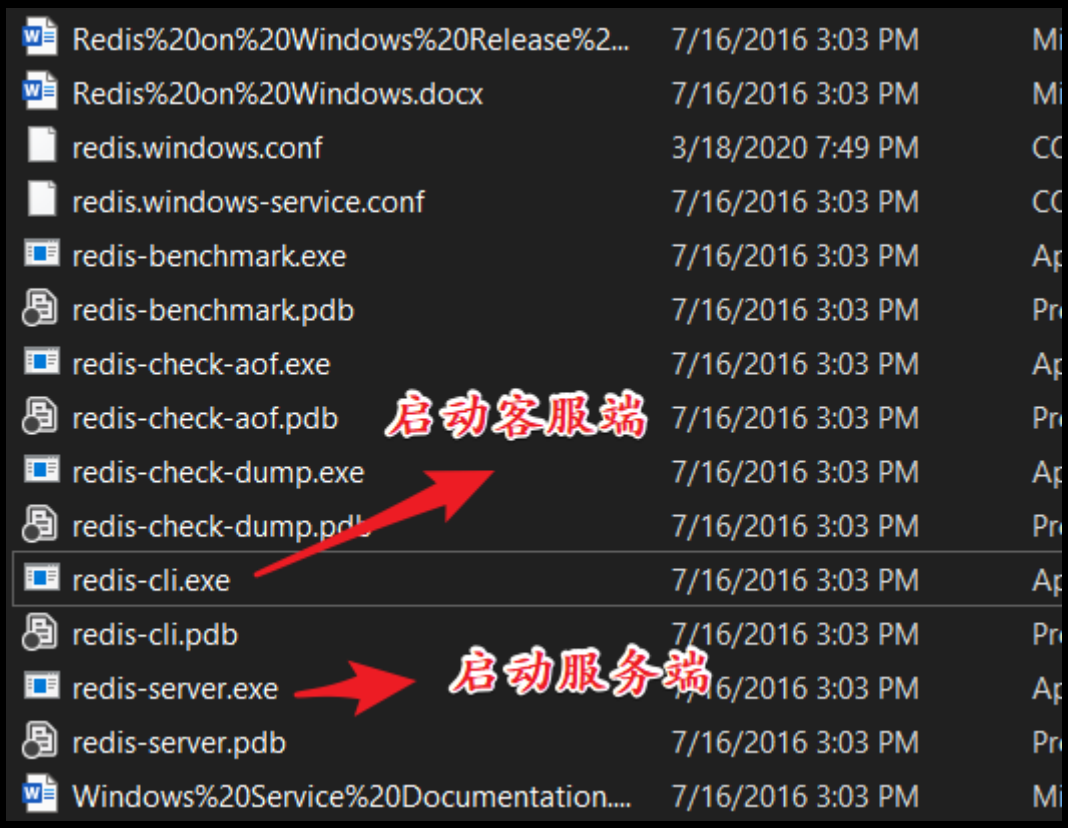
结构类型	结构存储的值	读写能力
String	可以是字符串、整数或者浮点数	对整个字符串或者字符串的其中一部分执行操作；对象和浮点数执行自增(increment)或者自减(decrement)
List	一个链表，链表上的每个节点都包含了一个字符串	从链表的两端推入或者弹出元素；根据偏移量对链表进行修剪(trim)；读取单个或者多个元素；根据值来查找或者移除元素
Set	包含字符串的无序收集器(unorderedcollection)，并且被包含的每个字符串都是独一无二的、各不相同	添加、获取、移除单个元素；检查一个元素是否存在于某个集合中；计算交集、并集、差集；从集合里卖弄随机获取元素
Hash	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(member)与浮点数分值(score)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(range)或者成员来获取元素

Redis 优势

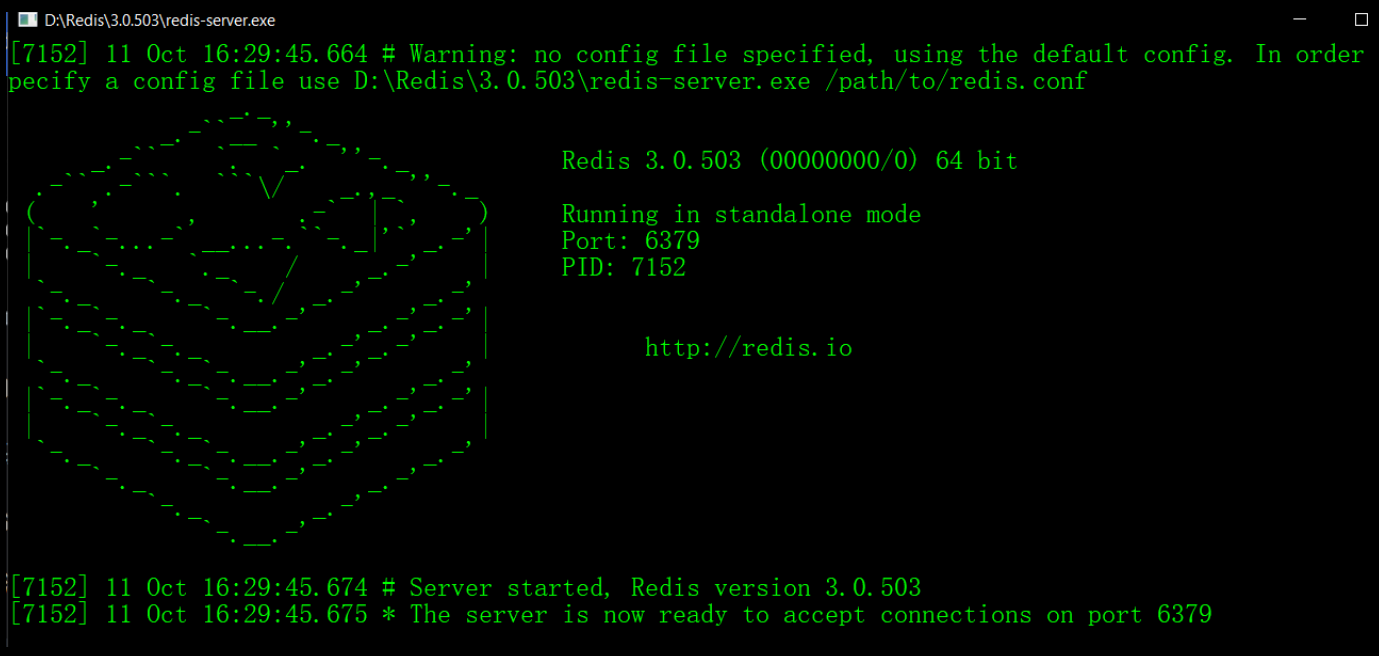
1. 性能极高 - Redis 能读的速度是 110000 次/s,写的速度是 81000 次/s 。
2. 丰富的数据类型 - Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。
3. **原子 - Redis 的所有操作都是原子性的**，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过 MULTI 和 EXEC 指令包起来。
4. 丰富的特性 - Redis 还支持 publish/subscribe, 通知, key 过期等等特性。

安装 Redis:

根据不同的需求下载 Linux 或 Windows 版本的，目前 Redis 官网只有 Linux 版本，但由于大多数开发者还是基于 windows 平台开发的，所有 GitHub 上的技术牛人基于 linux 平台下的 Redis 实现了 windows 版本，给 windows 开发带来了福音(https://github.com/MSOpenTech/redis/releases)。



redis.windows.conf 为 redis 配置文件，相关参数可以在这里配置，如：端口等



SpringBoot 中使用 Redis

添加 Redis 依赖包：

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

配置 Redis 数据库连接：

application.properties:

```
#redis 配置
#Redis 服务器地址
spring.redis.host=127.0.0.1
#Redis 服务器连接端口
spring.redis.port=6379
#Redis 数据库索引（默认为 0）
spring.redis.database=0
#连接池最大连接数（使用负值表示没有限制）
spring.redis.jedis.pool.max-active=50
#连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.jedis.pool.max-wait=3000
#连接池中的最大空闲连接
spring.redis.jedis.pool.max-idle=20
#连接池中的最小空闲连接
spring.redis.jedis.pool.min-idle=2
#连接超时时间（毫秒）
spring.redis.timeout=5000
```

编写 Redis 操作工具类

将 RedisTemplate 实例包装成一个工具类，便于对 redis 进行数据操作。

（这个应该是 k-V 的 String 类型）

```
@Component
public class RedisUtils {

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    /**
     * 读取缓存

     */
    public String get(final String key) {
        return redisTemplate.opsForValue().get(key);
    }

    /**
     * 写入缓存
     */
    public boolean set(final String key, String value) {
        boolean result = false;
        try {
            redisTemplate.opsForValue().set(key, value);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }

    /**
     * 更新缓存
     */
    public boolean getAndSet(final String key, String value) {
        boolean result = false;
        try {
            redisTemplate.opsForValue().getAndSet(key, value);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }

    /**
     * 删除缓存
     */
```

```
    public boolean delete(final String key) {
        boolean result = false;
        try {
            redisTemplate.delete(key);
            result = true;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return result;
    }
}
```

测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest
public class RedisTest {
    @Resource
    private RedisUtils redisUtils;

    /**
     * 插入缓存数据
     */
    @Test
    public void set() {
        redisUtils.set("redis_key", "redis_vale");
    }

    /**
     * 读取缓存数据
     */
    @Test
    public void get() {
        String value = redisUtils.get("redis_key");
        System.out.println(value);
    }
}
```

登录 redis-cli:

Set 之后:

```
127.0.0.1:6379> exists redis_key
(integer) 1
```

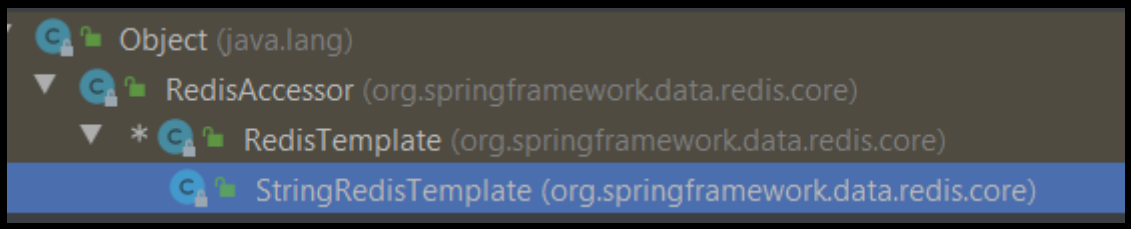
RedisTemplate

RedisTemplate 对五种数据结构分别定义了操作，如下所示：

1. 操作字符串：redisTemplate.opsForValue()
2. 操作 Hash：redisTemplate.opsForHash()
3. 操作 List：redisTemplate.opsForList()
4. 操作 Set：redisTemplate.opsForSet()
5. 操作 ZSet：redisTemplate.opsForZSet()

对于 string 类型的数据，Spring Boot 还专门提供了 **StringRedisTemplate** 类，而且官方也建议使用该类来操作 String 类型的数据

1. RedisTemplate 是一个泛型类，而 StringRedisTemplate 不是，后者只能对键和值都为 String 类型的数据进行操作，而前者则可以操作任何类型。
2. 两者的数据是不共通的，StringRedisTemplate 只能管理 StringRedisTemplate 里面的数据，RedisTemplate 只能管理 RedisTemplate 中的数据。



RedisTemplate 和 StringRedisTemplate 的配置 （Java 配置的方式）

```
@Configuration
public class RedisConfig {

    @Bean
    @ConditionalOnMissingBean(name = "redisTemplate")
    public RedisTemplate<String, Object> redisTemplate(
            RedisConnectionFactory redisConnectionFactory) {

        Jackson2JsonRedisSerializer<Object> jackson2JsonRedisSerializer = new
Jackson2JsonRedisSerializer<Object>(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);

        RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();
        template.setConnectionFactory(redisConnectionFactory);
```

```
        template.setKeySerializer(jackson2JsonRedisSerializer);
        template.setValueSerializer(jackson2JsonRedisSerializer);
        template.setHashKeySerializer(jackson2JsonRedisSerializer);
        template.setHashValueSerializer(jackson2JsonRedisSerializer);
        template.afterPropertiesSet();
        return template;
    }

    @Bean
    @ConditionalOnMissingBean(StringRedisTemplate.class)
    public StringRedisTemplate stringRedisTemplate(
            RedisConnectionFactory redisConnectionFactory) {
        StringRedisTemplate template = new StringRedisTemplate();
        template.setConnectionFactory(redisConnectionFactory);
        return template;
    }
}
```

面试题

缓存并发问题

这里的并发指的是多个 Redis 的客户端同时 set 值引起的并发问题。比较有效的解决方案就是把 set 操作放在队列中使其串行化，必须得一个一个执行。