

Redis基础

课程内容

- Redis简介
- 数据类型
- 常用指令
- Jedis
- 持久化

1. Redis 简介

1.1 NoSQL概念

1). 概念

NoSQL: 即 Not-Only SQL (泛指非关系型的数据库) , 作为关系型数据库的补充。 作用: 应对基于海量用户和海量数据前提下的数据处理问题。

他说这句话说的非常客气, 什么意思呢? 就是我们数据存储要用SQL, 但是呢可以不仅仅用SQL, 还可以用别的东西, 那别的东西叫什么呢? 于是他定义了一句话叫做NoSQL。这个意思就是说我们存储数据, 可以不光使用SQL, 我们还可以使用非SQL的这种存储方案, 这就是所谓的NoSQL。

2). 特征

可扩容, 可伸缩。SQL数据关系过于复杂, 扩容一下难度很高, 那我们Nosql 这种的, 不存关系, 所以它的扩容就简单一些。


大数据量下高性能。包数据非常多的时候, 它的性能高, 因为你不走磁盘IO, 你走的是内存, 性能肯定要比磁盘IO的性能快一些。

灵活的数据模型、高可用。他设计了自己的一些数据存储格式, 这样能保证效率上来说是比较高的, 最后一个高可用, 我们等到集群内部分再去它!

3) .常见 Nosql 数据库

目前市面上常见的Nosql产品：Redis、memcache、HBase、MongoDB

1.2 Redis概念

 **redis** [Commands](#) [Clients](#) [Documentation](#) [Community](#) [Download](#) [Modules](#) [Support](#) [Try Free](#)

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. Redis provides data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions, and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster. [Learn more →](#)

Try it

Ready for a test drive? Check this [interactive tutorial](#) that will walk you through the most important features of Redis.

Download it

[Redis 6.2.5 is the latest stable version.](#) Interested in release candidates or unstable versions? [Check the downloads page.](#)

Quick links

Follow day-to-day Redis on [Twitter](#) and [GitHub](#). Get help or help others by subscribing to [our mailing list](#), we are 5,000 and counting!

1.2.1 redis概念

概念：Redis 是用 C 语言开发的一个开源的高性能键值对（key-value）数据库。

特征：

- 1) . 数据间没有必然的关联关系；
- 2) . 内部采用单线程机制进行工作；
- 3) . 高性能。官方提供测试数据，50个并发执行100000 个请求,读的速度是110000 次/s,写的速度是81000次/s。
- 4) . 多数据类型支持

字符串类型 (string)

列表类型 (list)

散列类型 (hash)

集合类型 (set)

有序集合类型(zset/sorted_set)

- 5). 支持持久化, 可以进行数据灾难恢复

1.2.2 redis的应用场景

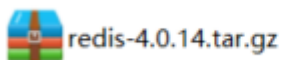
- 1). 为热点数据加速查询(主要场景)。如热点商品、热点新闻、热点资讯、推广类等高访问量信息等。
- 2). 即时信息查询。如各位排行榜、各类网站访问统计、公交到站信息、在线人数信息(聊天室、网站)、设备信号等。
- 3). 时效性信息控制。如验证码控制、投票控制等。
- 4). 分布式数据共享。如分布式集群架构中的 session 分离。
- 5). 消息队列。

1.3 Redis安装与启动

1.3.1 Redis安装

本课程所示, 均基于CenterOS7安装Redis。

- 1). 上传redis的安装包redis-4.0.14.tar.gz到Linux系统 /root目录下



- 2). 准备C语言编译环境

```
1 yum install gcc
```

- 3). 安装Redis

```
1 A. 切换目录/root
```

```
2    cd /root
3
4    B. 解压
5    tar -zxvf redis-4.0.14.tar.gz
6    cd redis-4.0.14
7
8    C. 编译C语言源码
9    make
10
11   D. 安装
12   make install PREFIX=/usr/local/redis
13
14   E. 进入安装目录，拷贝redis配置文件
15   cd /usr/local/redis/
16   ll
17   cp /root/redis-4.0.14/redis.conf .
```

```
[root@localhost redis]#
[root@localhost redis]# ll
总用量 60
drwxr-xr-x. 2 root root   134 8月 21 19:34 bin
-rw-r--r--. 1 root root 58765 8月 21 20:13 redis.conf
```

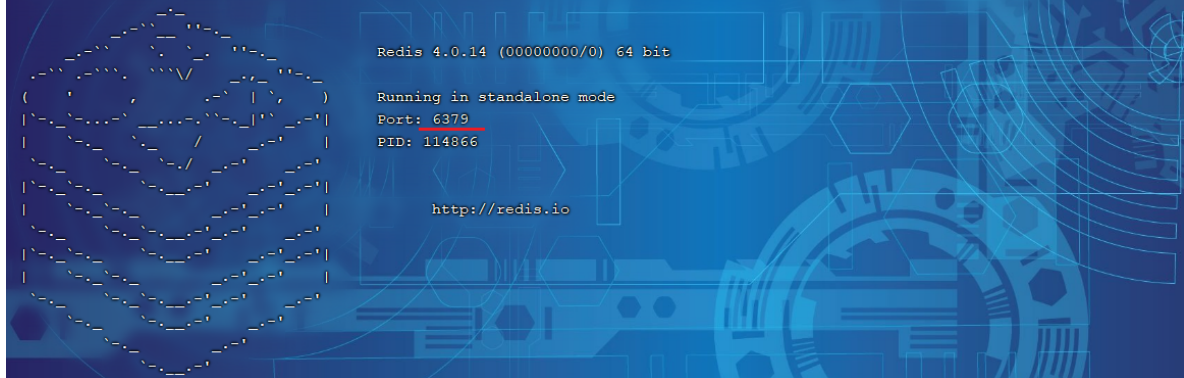
1.3.2 redis目录及文件

指令/文件	描述
bin/redis-server	服务器启动命令
bin/redis-cli	客户端启动命令
bin/redis-check-dump	RDB文件检查工具（快照持久化文件）
bin/redis-check-aof	AOF文件修复工具
redis.conf	redis核心配置文件

1.3.3 Redis服务器启动

```
1 cd /usr/local/redis/
2
3 bin/redis-server redis.conf
```

```
[root@localhost redis]#
[root@localhost redis]# bin/redis-server redis.conf
114866:C 27 Aug 18:53:05.289 # oOoOoOoOoOoOo Redis is starting oOoOoOoOoOoOo
114866:C 27 Aug 18:53:05.289 # Redis version=4.0.14, bits=64, commit=00000000, modified=0, pid=114866, just started
114866:C 27 Aug 18:53:05.289 # Configuration loaded
114866:M 27 Aug 18:53:05.290 * Increased maximum number of open files to 10032 (it was originally set to 1024).
```



1.3.4 Redis客户端启动

启动客户端

```
1 redis-cli [-h host] [-p port]
```

示例:

```
1 cd /usr/local/redis/
2
3 A. 连接指定IP, 指定端口号的redis
4 bin/redis-cli -h 192.168.200.200 -p 6379
5
6 B. 连接本地默认端口redis
7 bin/redis-cli
```

```
[root@localhost redis]#
[root@localhost redis]# bin/redis-cli
127.0.0.1:6379>
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
```

1.4 服务器端配置

1). 设置服务器以守护进程的方式运行，开启后服务器控制台中将打印服务器运行信息（同日志内容相同）

```
1 daemonize yes|no
```

no : 代表以非守护进程方式运行，启动之后会占用窗口；

yes : 代表以守护进程方式运行，后台运行，不占用前端窗口；

注意：设置为yes之后，重新启动，就不会占用前端窗口了；

2). 绑定主机地址

```
1 bind 127.0.0.1
```

如果绑定的是127.0.0.1,代表只能本机访问；如果所有的IP都可以访问，需要修改为 0.0.0.0

3). 设置服务器端口

```
1 port 6379
```

4). 设置服务器文件保存地址

```
1 dir path
```

2. 数据类型

2.1 数据类型介绍

2.1.1 Redis数据类型

数据类型	含义	特点
string	字符串类型	
hash	哈希类型	map格式
list	列表类型	linkedlist格式。支持重复元素
set	集合类型	不允许重复元素
sortedSet	有序集合类型	不允许重复元素，且元素有顺序

2.1.2 结构说明

在学习数据类型之前，我们先要明白数据类型到底是修饰什么的。我们知道redis自身是一个 Map，其中所有的数据都是采用 `key : value` 的形式存储。

对于这种结构来说，我们用来存储数据一定是一个值前面对应一个名称。我们通过名称来访问后面的值。按照这种形势，我们可以对出来我们的存储格式。前面这一部分我们称为key。后面的一部分称为value，而我们的数据类型，他一定是修饰value的。

数据类型指的是存储的数据的类型，也就是 `value` 部分的类型，`key` 部分永远都是字符串。

2.2 string数据类型操作

2.2.1 string类型

string类型中，value存储的就是字符串数据。

Redis 存储空间

key	value
name	itheima
age	101
名称	数据
key	value

2.2.2 string基本操作

1). 基础指令

①. 添加/修改数据

```
1 set key value
```

②. 获取数据

```
1 get key
```

③. 删除数据

```
1 del key
```

④. 判定性添加数据 (如果key不存在,则创建)

```
1 setnx key value
```

⑤. 添加/修改多个数据

```
1 mset key1 value1 key2 value2 ...
```


⑥. 获取多个数据

```
1 mget key1 key2 ...
```

⑦. 追加信息到原始信息后部 (如果原始信息存在就追加, 否则新建)

```
1 append key value
```

⑧. 设置过期时间

```
1 setex key seconds value
```

2). 拓展指令

①. 设置数值数据增加/减少

```
1 incr key
2 incrby key increment
```

②. 设置数值数据减少

```
1 decr key
2 decrby key increment
```

如:

```
set count 1
```

```
incr count -----> 自增操作, 每次增加1
```

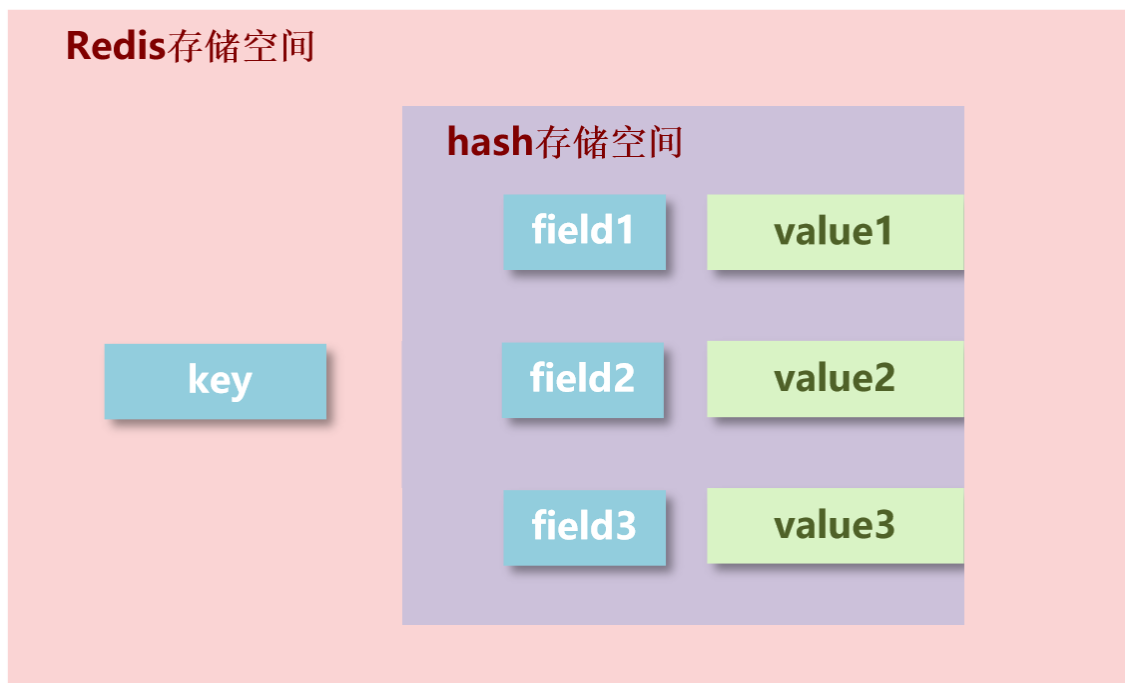
```
incr count 2 -----> 自增操作, 每次增加2
```

```
incr num -----> 如果num这个key不存在, 默认从0开始自增
```

2.3 hash数据类型操作

2.3.1 hash类型

hash类型：底层使用哈希表结构实现数据存储



2.3.2 hash基本操作

1). 基本操作

①. 添加/修改数据

```
1 hset key field value
```

②. 获取数据

```
1 hget key field
```

③. 删除数据

```
1 hdel key field1 [field2]
```

④. 添加/修改多个数据

```
1 hmset key field1 value1 field2 value2 ...
```

⑤. 获取多个数据

```
1 hmget key field1 field2 ...
```

⑥. 获取哈希表中字段的数量

```
1 hlen key
```

⑦. 获取哈希表中是否存在指定的字段

```
1 hexists key field
```

2). 拓展操作

①. 获取哈希表中所有的字段名

```
1 hkeys key
```

②. 获取哈希表中所有的字段值

```
1 hvals key
```

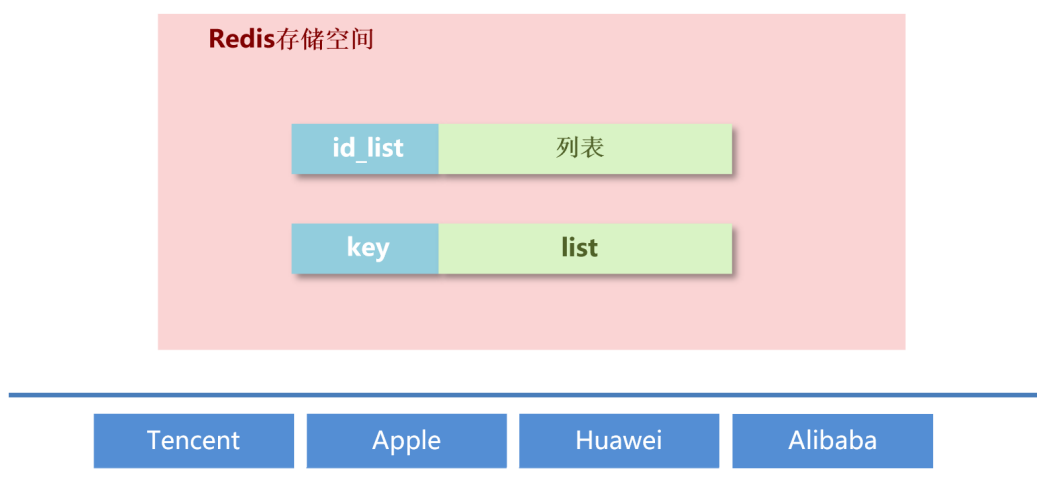
③. 一次性获取所有键值对

```
1 hgetall key
```

2.4 list数据类型操作

前面我们存数据的时候呢，单个数据也能存，多个数据也能存，但是这里面有一个问题，我们存多个数据用hash的时候它是没有顺序的。我们平时操作，实际上数据很多情况下都是有顺序的，那有没有一种能够用来存储带有顺序的这种数据模型呢，list就专门来干这事儿。

2.4.1 list类型



Redis的List数据类型是一个双向链表结构。Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）一个列表最多可以包含 $2^{32} - 1$ 个元素。

特点：

有序，可以重复；

2.4.2 list基本操作

①. 将元素加入列表左边

```
1 lpush key value1 [value2] .....
```

②. 将元素加入列表左边

```
1 rpush key value1 [value2] .....
```

③. 范围获取数据

```
1 lrange key start stop
```

④. 获取指定索引的元素

```
1 lindex key index
```

⑤. 获取元素个数

```
1 llen key
```

⑥. 删除列表最左边的元素，并将元素返回

```
1 lpop key
```

⑦. 删除列表最右边的元素，并将元素返回

```
1 rpop key
```

2.5 set数据类型操作

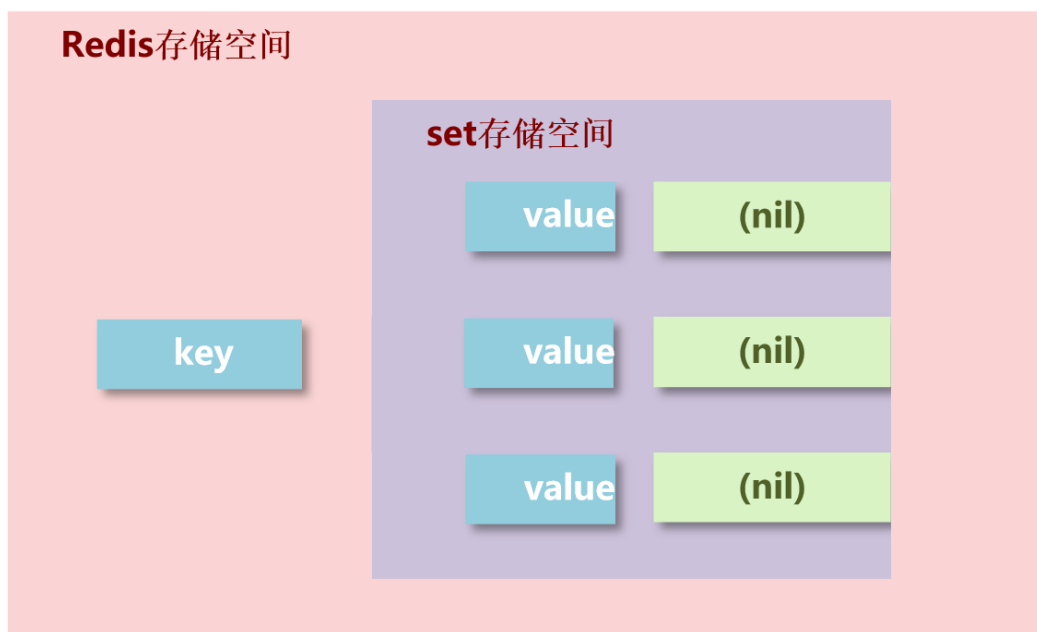
2.5.1 set类型

我们现在要存储大量的数据，并且要求提高它的查询效率。用list这种链表形式，它的查询效率是不高的，那怎么办呢？这时候我们就想，有没有高效的存储机制。其实前面咱讲Java的时候说过hash表的结构就非常的好，但是这里边我们已经有hash了，他做了这么一个设定，干嘛呢，他把hash的存储空间给改一下，右边你原来存数据改掉，全部存空，那你说数据放哪儿了？放到原来的filed的位置，

也就在这里边存真正的值，那么这个模型就是我们的set 模型。

set类型：与hash存储结构完全相同，仅存储键，不存储值（nil），并且值是不允许重复的。

看一下它的整个结构：



特点：

无序 ； 不可重复 ；

2.5.2 set基本操作

1). 基础操作

①. 添加数据

```
1 sadd key member1 [member2]
```

②. 获取全部数据

```
1 smembers key
```

③. 删除数据

```
1 srem key member1 [member2]
```

④. 获取集合数据总量

```
1 scard key
```

⑤. 判断集合中是否包含指定数据

```
1 sismember key member
```

⑥. 随机获取集合中指定数量的数据

```
1 srandmember key [count]
```

⑦. 随机获取集中的某个数据并将该数据移除集合

```
1 spop key [count]
```

2). 拓展操作

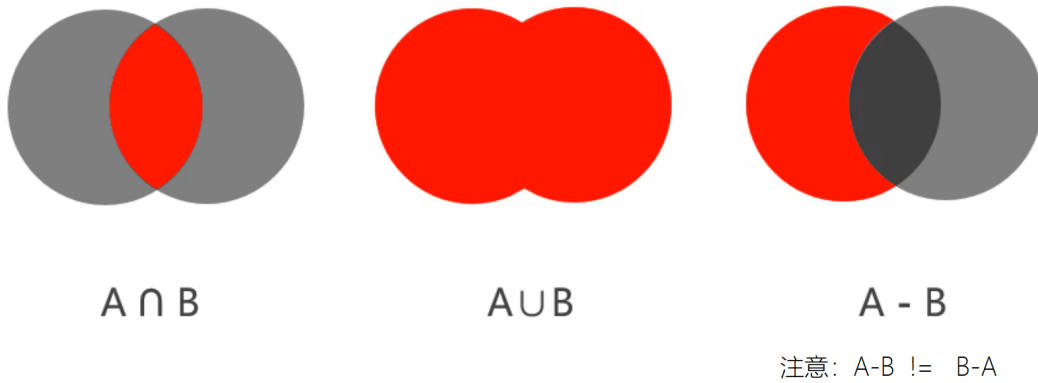
①. 求两个集合的交、并、差集

```
1 sinter key1 [key2 ...]  
2 sunion key1 [key2 ...]  
3 sdiff key1 [key2 ...]
```

②. 求两个集合的交、并、差集并存储到指定集合中

```
1 sinterstore destination key1 [key2 ...]  
2 sunionstore destination key1 [key2 ...]  
3 sdiffstore destination key1 [key2 ...]
```

通过下面一张图回忆一下交、并、差



2.6 zset数据类型操作

2.6.1 zset类型

zset, 也叫sortedSet, 是一个有序集合, 而set是无序的, zset怎么做到有序的呢? Redis 有序集合和集合一样也是 string 类型元素的集合, 且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。有序集合的成员是唯一的, 但分数(score)却可以重复。

2.6.2 zset基本操作

①. 添加元素

```
1 ZADD key score member [[score member] ...]
```

②. 获取元素个数

```
1 ZCARD key
```

③. 获取指定分数范围内的元素

分数从小到大排

```
1 ZRANGE key start end [WITHSCORES]
```


分数从大到小排

```
1 ZREVRANGE key start end [WITHSCORES]
```

④. 删除指定元素

```
1 zrem key value
```

3. 常用指令

3.1 key基本操作

删除指定key

```
1 del key
```

获取key是否存在

```
1 exists key
```

获取key的类型

```
1 type key
```

为指定key设置有效期

```
1 expire key seconds
```

获取key的有效时间

```
1    ttl key
```

切换key从时效性转换为永久性

```
1    persist key
```

查询key

```
1    keys pattern
```

查询模式规则：

*： 匹配任意数量的任意符号

?： 配合一个任意符号

3.3 数据库指令

redis为每个服务提供有16个数据库，编号从0到15，每个数据库之间的数据相互独立，在对应的数据库中划出一块区域，说他就是几，你就用几那块，同时，其他的这些都可以进行定义，一共是16个，这里边需要注意一点，他们这16个共用redis的内存。没有说谁大谁小，也就是说数字只是代表了一块儿区域，区域具体多大未知。这是数据库的一个分区的一个策略！

①． 切换数据库

```
1    select index
```

②． 心跳测试

```
1 ping
```

③. 数据总量

```
1 dbsize
```

④. 数据清除

```
1 flushdb : 清空当前数据库
2 flushall : 清空所有数据库
```

4. 应用程序操作Redis

4.1 Jedis

4.1.1 jedis介绍

一款java操作redis数据库的工具

4.1.2 jedis入门

1). pom.xml 引入依赖

```
1 <dependencies>
2     <dependency>
3         <groupId>redis.clients</groupId>
4         <artifactId>jedis</artifactId>
5         <version>2.9.0</version>
6     </dependency>
7
8     <dependency>
9         <groupId>junit</groupId>
```

```
10         <artifactId>junit</artifactId>
11         <version>4.12</version>
12         <scope>test</scope>
13     </dependency>
14 </dependencies>
```

2). 构建jedis对象

连接redis

```
1    Jedis jedis = new Jedis("localhost", 6379);
```

3). 代码实现

创建: com.itheima.redis.JedisDemo

```
1    import org.junit.Test;
2    import redis.clients.jedis.Jedis;
3
4    import java.util.List;
5    import java.util.Set;
6
7    public class JedisDemo {
8
9        //测试string数据类型
10        @Test
11        public void testString(){
12            //1. 获取连接对象Jedis
13            Jedis jedis = new Jedis("192.168.200.200",6379);
14
15            //2. 执行操作 ---> 与讲解的指令一一对应
16            jedis.set("name","itcast");
17            jedis.append("name","-is-very-good");
18            String name = jedis.get("name");
19
20            System.out.println(name);
21            jedis.close();
```

```

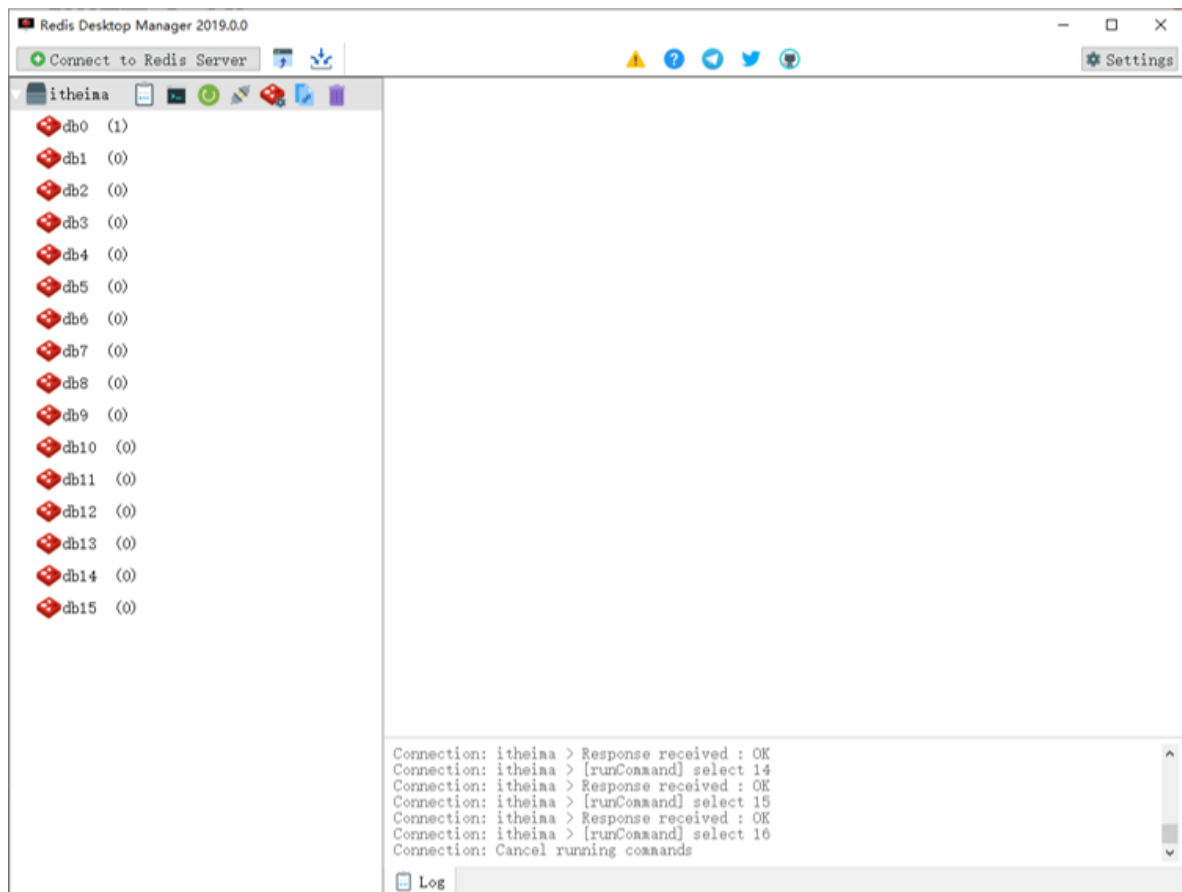
22     }
23
24
25     //测试Hash数据类型
26     @Test
27     public void testHash(){
28         //1. 获取连接对象Jedis
29         Jedis jedis = new Jedis("192.168.200.200",6379);
30
31         //2. 执行操作 ---> 与讲解的指令一一对应
32         jedis.hset("user","name","itcast");
33         jedis.hset("user","age","14");
34
35         String age = jedis.hget("user", "age");
36         System.out.println(age);
37         jedis.close();
38     }
39
40
41     //测试List数据类型
42     @Test
43     public void testList(){
44         //1. 获取连接对象Jedis
45         Jedis jedis = new Jedis("192.168.200.200",6379);
46
47         //2. 执行操作 ---> 与讲解的指令一一对应
48         jedis.lpush("names","Tom","Cat","Jerry","Lucy","Lee");
49         List<String> names = jedis.lrange("names", 0, -1);
50         for (String name : names) {
51             System.out.println(name);
52         }
53
54         jedis.close();
55     }
56
57
58     //测试Set数据类型

```

```
59     @Test
60     public void testSet() {
61         //1. 获取连接对象Jedis
62         Jedis jedis = new Jedis("192.168.200.200", 6379);
63
64         //2. 执行操作 ---> 与讲解的指令一一对应
65         jedis.sadd("dev", "java", "c", "php", ".Net", "go", "java");
66         Set<String> members = jedis.smembers("dev");
67
68         for (String member : members) {
69             System.out.println(member);
70         }
71         jedis.close();
72     }
73
74 }
```

4.1.3 可视化客户端

Redis Desktop Manager



4.2 SpringDataRedis

4.2.1 介绍

Spring Data Redis提供了从Spring应用程序轻松配置和访问Redis的功能。它提供了用于与存储交互的低级和高级抽象，使用户不必再关注基础设施。

4.2.2 入门程序

1) . pom.xml

```
1 <parent>
2 <groupId>org.springframework.boot</groupId>
3 <artifactId>spring-boot-starter-parent</artifactId>
4 <version>2.4.5</version>
```

```

5      <relativePath/> <!-- lookup parent from repository -->
6  </parent>
7
8
9  <dependencies>
10    <dependency>
11      <groupId>org.springframework.boot</groupId>
12      <artifactId>spring-boot-starter-data-redis</artifactId>
13    </dependency>
14
15    <dependency>
16      <groupId>org.springframework.boot</groupId>
17      <artifactId>spring-boot-starter-test</artifactId>
18    </dependency>
19
20    <dependency>
21      <groupId>junit</groupId>
22      <artifactId>junit</artifactId>
23      <scope>test</scope>
24    </dependency>
25
26    <dependency>
27      <groupId>org.projectlombok</groupId>
28      <artifactId>lombok</artifactId>
29    </dependency>
30  </dependencies>

```

2). application.yml

```

1  spring:
2    redis:
3      host: 192.168.200.200
4      port: 6379

```

3). 引导类


```

1  import org.springframework.boot.SpringApplication;
2  import org.springframework.boot.autoconfigure.SpringBootApplication;
3
4  @SpringBootApplication
5  public class RedisApplication {
6      public static void main(String[] args) {
7          SpringApplication.run(RedisApplication.class, args);
8      }
9  }
10

```

4). 基本操作

```

1  import org.junit.Test;
2  import org.junit.runner.RunWith;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.boot.test.context.SpringBootTest;
5  import org.springframework.data.redis.core.RedisTemplate;
6  import org.springframework.test.context.junit4.SpringRunner;
7
8  import java.util.List;
9  import java.util.Set;
10
11  @RunWith(SpringRunner.class)
12  @SpringBootTest
13  public class RedisTest {
14
15      @Autowired
16      private RedisTemplate redisTemplate;
17
18      //测试string数据类型
19      @Test
20      public void testString() {
21          redisTemplate.boundValueOps("name").set("itcast");
22
23          Object name = redisTemplate.boundValueOps("name").get();
24

```

```
24         System.out.println(name);
25     }
26
27
28     //测试Hash数据类型
29     @Test
30     public void testHash() {
31
32         redisTemplate.boundHashOps("user").put("name", "itheima");
33         redisTemplate.boundHashOps("user").put("age", "10");
34
35         Object o = redisTemplate.boundHashOps("user").get("age");
36         System.out.println(o);
37     }
38
39
40     //测试List数据类型
41     @Test
42     public void testList() {
43         redisTemplate.boundListOps("devs").leftPush("java");
44         redisTemplate.boundListOps("devs").leftPush("go");
45         redisTemplate.boundListOps("devs").leftPush("c");
46         redisTemplate.boundListOps("devs").leftPush("php");
47
48
49         List devs = redisTemplate.boundListOps("devs").range(0, -1);
50         for (Object dev : devs) {
51             System.out.println(dev);
52         }
53     }
54
55
56     //测试Set数据类型
57     @Test
58     public void testSet() {
59         redisTemplate.boundSetOps("names").add("Tom", "Dawn", "Jay", "Lee", "Tom");
60     }
```

```
61         Set names = redisTemplate.boundSetOps("names").members();
62         System.out.println(names);
63     }
64
65 }
```

5). 存储对象

A. 实体类

```
1  import lombok.AllArgsConstructor;
2  import lombok.Data;
3  import lombok.NoArgsConstructor;
4
5  @Data
6  @NoArgsConstructor
7  @AllArgsConstructor
8  public class User {
9
10     private Integer id;
11     private String name;
12     private Integer age;
13     private String desc;
14
15 }
```

B. RedisTemplate操作

```
1  @Test
2  public void testObject(){
3      User user = new User(1, "Tom", 10, "优秀");
4      redisTemplate.boundValueOps("user1").set(user);
5  }
```

当我们在将user保存到redis时，应用程序执行却报错了，报错信息如下：

```
org.springframework.data.redis.serializer.SerializationException: Cannot serialize; nested exception is org.springframework.core.serializer.support.Serializ
at org.springframework.data.redis.serializer.JdkSerializationRedisSerializer.serialize(JdkSerializationRedisSerializer.java:96)
at org.springframework.data.redis.core.AbstractOperations.rawValue(AbstractOperations.java:127)
at org.springframework.data.redis.core.DefaultValueOperations.set(DefaultValueOperations.java:235)
at org.springframework.data.redis.core.DefaultBoundValueOperations.set(DefaultBoundValueOperations.java:140)
at com.itheima.redis.RedisTest.testObject(RedisTest.java:72) <8 internal calls>
```

出现这个问题的主要原因，就是因为User对象无法被序列化，因为并没有实现序列化接口。因为一个对象，要想被存入内存，并且在一定的时机需要持久化到磁盘，就必须要实现序列化接口

Serializable。

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User implements Serializable {

    private Integer id;
    private String name;
    private Integer age;
    private String desc;

}
```

然后再次执行保存就可以成功了。

5. 持久化

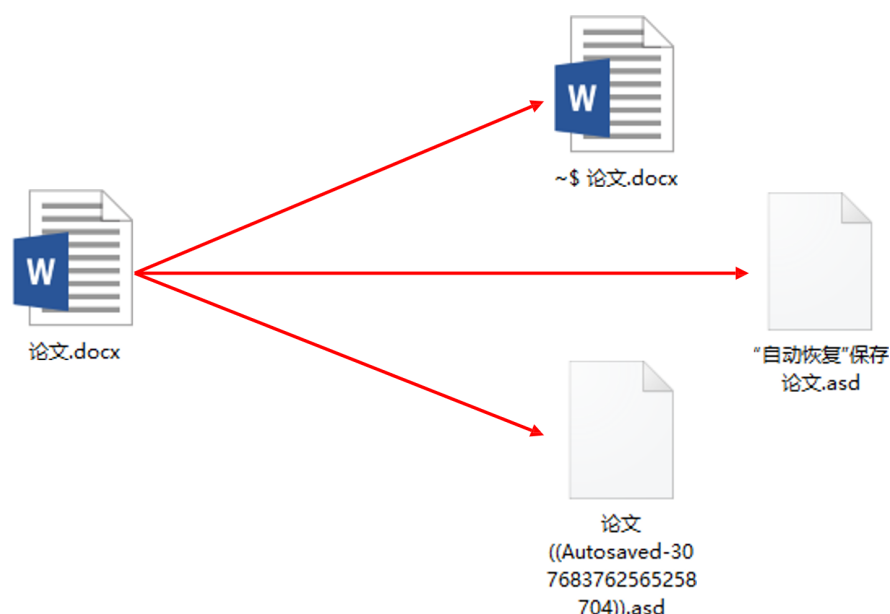
下面呢，进入到持久化的学习。这部分内容理解的东西多，操作的东西少。在这个部分，我们将讲解四部分内容：

5.1 持久化简介

5.1.1 场景-意外断电

不知道大家有没有遇见过，就是正工作的时候停电了，如果你用的是笔记本电脑还好，你有电池，但如果你用的是台式机呢，那恐怕就比较灾难了，假如你现在正在写一个比较重要的文档，如果你要使用的是word，这种办公自动化软件的话，他一旦遇到停电，其实你不用担心，因为它会给你生成一些其他的文件。

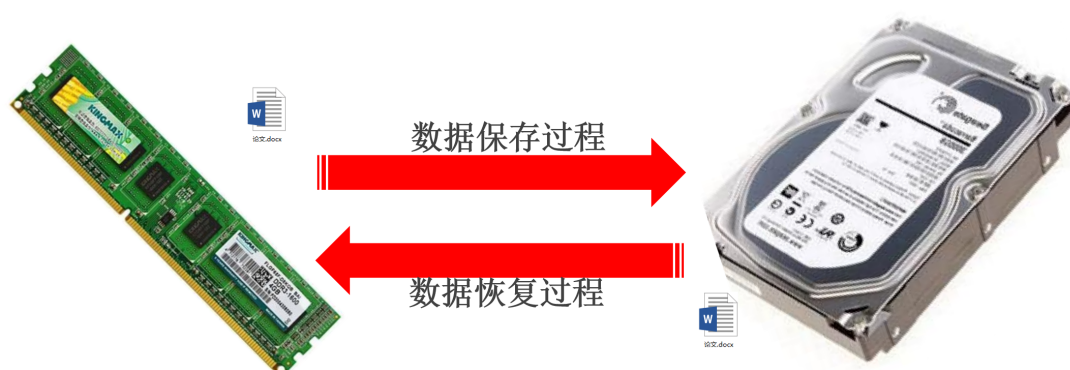
意外的断电



其实他们都在做一件事儿，帮你自动恢复，有了这个文件，你前面的东西就不再丢了。那什么是自动恢复呢？你要先了解他的整个过程。

我们说自动恢复，其实基于的一个前提就是他提前把你的数据给存起来了。你平常操作的所有信息都是在内存中的，而我们真正的信息是保存在硬盘中的，内存中的信息断电以后就消失了，硬盘中的信息断电以后还可以保留下来！

“自动备份”



我们将文件由内存中保存到硬盘中的这个过程，我们叫做数据保存，也就叫做持久化。但是把它保存下来不是你的目的，最终你还要把它再读取出来，它加载到内存中这个过程，我们叫做数据恢复，这就是我们所说的word为什么断电以后还能够给你保留文件，因为它执行了一个自动备份的过程，也就是通过自动的形式，把你的数据存储起来，那么有了这种形式以后，我们的数据就可以由内存到硬盘上实现保存。

5.1.2 什么是持久化

1) . 什么是持久化

利用永久性存储介质(磁盘)将数据进行保存,在特定的时间将保存的数据进行恢复的工作机制称为持久化。持久化用于防止数据的意外丢失,确保数据安全性。

2) . 持久化过程保存什么?

我们知道一点,计算机中的数据全部都是二进制,如果现在我要你给我保存一组数据的话,你有什么样的方式呢,其实最简单的就是现在长什么样,我就记下来就行了,那么这种是记录纯粹的数据,也叫做快照存储,也就是它保存的是某一时刻的数据状态。

还有一种形式,它不记录你的数据,它记录你所有的操作过程,比如说大家用idea的时候,有没有遇到过写错了`ctrl+z`撤销,然后`ctrl+y`还能恢复,这个地方它也是在记录,但是记录的是你所有的操作过程,那我想问一下,操作过程,我都给你留下来了,你说数据还会丢吗?肯定不会丢,因为你所有的操作过程我都保存了。这种保存操作过程的存储,用专业术语来说可以说是日志,这是两种不同的保存数据的形式啊。

```
10011001110000001
00101001011010110
10110011001110000
00100101001011011
```

数据(快照)

RDB

```
删除第3行
第4行末位添加字符x
删除第2到第4行
复制第3行粘贴到第5行
```

过程(日志)

AOF

总结一下:

第一种(RDB): 将当前数据状态进行保存,快照形式,存储数据结果,存储格式简单,关注点在数据。

第二种(AOF): 将数据的操作过程进行保存,日志形式,存储操作过程,存储格式复杂,关注点在数据的操作过程。

5.2 RDB

5.2.1 手动方式

直接通过指令 `save` 或 `bgsave` 来执行RDB持久化操作。

1). `save`

```
1 save
```

我们可以直接在客户端命令行中执行 `save` 指令来，来进行RDB持久化操作。但是这种样可能存在问题，并且耗费性能。

需要注意一个问题，如果有多个客户端各自要执行一个指令，把这些指令发送到redis服务器后，他们执行有一个先后顺序问题，假定就是按照1234的顺序放过去的话，那会是什么样的？记得redis是个单线程的工作模式，它会创建一个任务队列，所有的命令都会进到这个队列里边，在这儿排队执行，执行完一个消失一个，当所有的命令都执行完了，OK，结果达到了。

但是如果现在我们执行的时候`save`指令保存的数据量很大会有什么现象呢？

他会非常耗时，以至于影响到它在执行的时候，后面的指令都要等，所以说这种模式是不友好的，这是`save`指令对应的问题，当cpu执行的时候会阻塞redis服务器，直到他执行完毕，所以说我们不建议大家在线上环境用`save`指令。

`save`指令的执行会阻塞当前redis服务器，直到当前RDB过程完成为止，有可能会造成长时间的阻塞。线上环境不建议使用。

2). `bgsave`

上面我们讲到了当`save`指令的数据量过大时，单线程执行方式造成效率过低，那应该如何处理？此时我们可以使用：**`bgsave`**指令，bg其实是background的意思，后台执行的意思是手动启动后台保存操作，但不是立即执行

```
1 bgsave
```

当执行**bgsave**的时候，客户端发出**bgsave**指令给到redis服务器。注意，这个时候服务器马上回一个结果告诉客户端后台已经开始了，与此同时它会创建一个子进程，使用Linux的**fork**函数创建一个子进程，让这个子进程去执行**save**相关的操作，此时我们可以想一下，我们主进程一直在处理指令，而子进程在执行后台的保存，它会不会干扰到主进程的执行吗？

答案是不会，所以说他才是主流方案。子进程开始执行之后，它就会创建啊RDB文件把它存起来，操作完以后他会把这个结果返回，也就是说**bgsave**的过程分成两个过程，第一个是服务端拿到指令直接告诉客户端开始执行了；另外一个过程是一个子进程在完成后台的保存操作，操作完以后回一个消息。

bgsave命令是针对**save**阻塞问题做的优化。Redis内部所有涉及到RDB操作都采用**bgsave**的方式，上面讲解的**save**可以放弃使用。

5.2.2 自动方式

设置自动持久化的条件，满足限定时间范围内key的变化数量达到指定数量即进行持久化

```
1  save second changes
```

参数

second：监控时间范围

changes：监控key的变化量

范例：

```
1  save 900 1
2  save 300 10
3  save 60 10000
```

关于RDB的配置参数说明：


```
1  #设置本地数据库文件名，默认值为 dump.rdb
2  dbfilename filename
3
4  #设置存储.rdb文件的路径，通常设置成存储空间较大的目录中，目录名称data
5  dir path
6
7  #设置存储至本地数据库时是否压缩数据，默认yes，设置为no，节省 CPU 运行时间，但存储文件变大
8  rdbcompression yes|no
9
10 #设置读写文件过程是否进行RDB格式校验，默认yes，设置为no，节约读写10%时间消耗，单存在数据损坏的
    风险
11 rdbchecksum yes|no
```

5.2.3 RDB优缺点

RDB优点：

- RDB是一个紧凑压缩的二进制文件，存储效率较高
- RDB内部存储的是redis在某个时间点的数据快照，非常适合用于数据备份，全量复制等场景
- RDB恢复数据的速度要比AOF快很多
- 应用：服务器中每X小时执行bgsave备份，并将RDB文件拷贝到远程机器中，用于灾难恢复。

RDB缺点

- RDB方式无论是执行指令还是利用配置，无法做到实时持久化，具有较大的可能性丢失数据
- bgsave指令每次运行要执行fork操作创建子进程，要牺牲掉一些性能

5.3 AOF

5.3.1 AOF概念

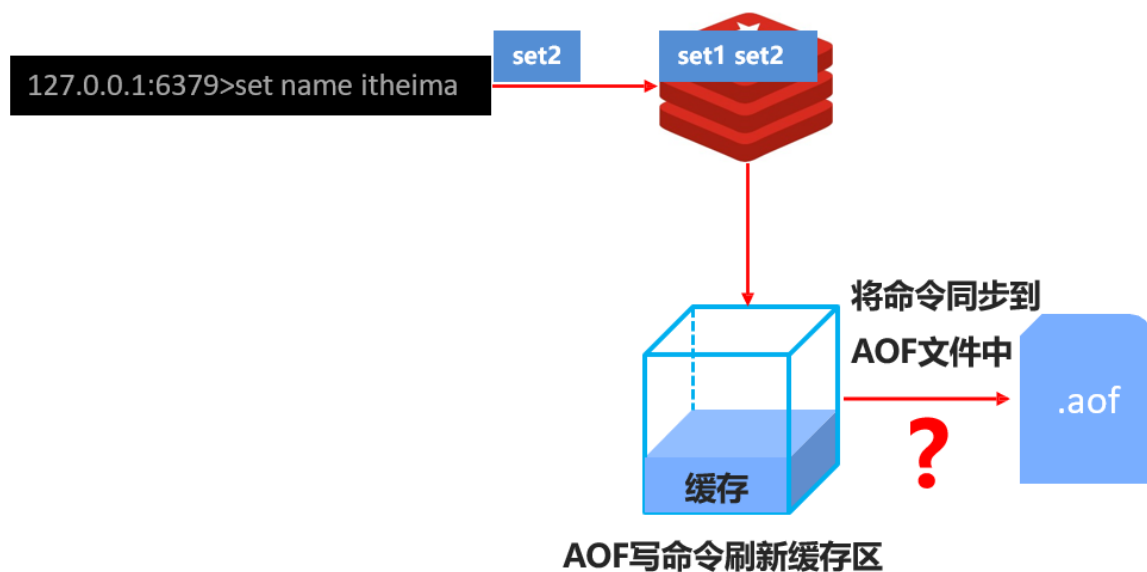
AOF (append only file)持久化：以独立日志的方式记录每次写命令，重启时再重新执行AOF文件中命令 达到恢复数据的目的。与RDB相比可以简单理解为由记录数据改为记录数据产生的变化

AOF的主要作用是解决了数据持久化的实时性，目前已经是Redis持久化的主流方式

启动AOF相关配置

```
1  #开启AOF持久化功能，默认no，即不开启状态
2  appendonly yes|no
3
4  #AOF持久化文件名，默认文件名为appendonly.aof，建议配置为appendonly-端口号.aof
5  appendfilename filename
6
7  #AOF写数据策略，默认为everysec
8  appendfsync always|everysec|no
```

5.3.2 AOF执行策略



AOF写数据三种策略 (appendfsync)

- **always** (每次)：每次写入操作均同步到AOF文件中数据零误差，性能较低，不建议使用。
- **everysec** (每秒)：每秒将缓冲区中的指令同步到AOF文件中，在系统突然宕机的情况下丢失1秒内的数据 数据准确性较高，性能较高，建议使用，也是默认配置
- **no** (系统控制)：由操作系统控制每次同步到AOF文件的周期，整体过程不可控

5.3.3 AOF重写

什么叫AOF重写？

随着命令不断写入AOF，文件会越来越大，为了解决这个问题，Redis引入了AOF重写机制压缩文件体积。AOF文件重写是将Redis进程内的数据转化为写命令同步到新AOF文件的过程。简单说就是将对同一个数据的若干个条命令执行结果转化成最终结果数据对应的指令进行记录。

AOF重写方式

1) . 手动重写

```
1  bgrewriteaof
```

2) . 自动重写

```
1  auto-aof-rewrite-min-size size
2  auto-aof-rewrite-percentage percentage
```

5.4 RDB与AOF区别

5.4.1 RDB与AOF对比 (优缺点)

持久化方式	RDB	AOF
占用存储空间	小（数据级：压缩）	大（指令级：重写）
存储速度	慢	快
恢复速度	快	慢
数据安全性	会丢失数据	依据策略决定
资源消耗	高/重量级	低/轻量级
启动优先级	低	高

5.4.2 RDB与AOF应用场景

- RDB与AOF的选择实际上是在做一种权衡，每种都有利有弊
- 如不能承受数分钟以内的数据丢失，对业务数据非常敏感，选用AOF
- 如能承受数分钟以内的数据丢失，且追求大数据集的恢复速度，选用RDB
- 灾难恢复选用RDB
- 双保险策略，同时开启 RDB和 AOF，重启后，Redis优先使用 AOF 来恢复数据，降低丢失数据的量