

Day22

Redis 数据类型

strings 类型及操作

set 操作

setnx 操作

setex 操作

setrange 操作

mset 操作

msetnx 操作

get 操作

getset 操作

getrange 操作

mget 操作

incr 操作

incrby 操作

decr 操作

decrby 操作

append 操作

strlen 操作

Redis 常用指令

key 操作分析

key 基本操作

key 扩展操作（时效性控制）

key 扩展操作（查询模式）

数据库指令

key 的重复问题

解决方案

数据库的基本操作

数据库扩展操作

Jedis

简介

准备工作

jar包导入

客户端连接 redis

Demo

Jedis简易工具类开发

基于连接池获取连接

加载配置信息

获取连接

作业

可视化客户端

Redis 高可用概述

持久化

复制

哨兵

集群

持久化简介

Redis 持久化

RDB 持久化

概述

触发条件

手动触发

自动触发

save 指令

save 指令相关配置

save 指令的问题

bgsave指令

bgsave指令相关配置

bgsave 指令工作原理

save配置自动执行

save配置工作原理

RDB 的启动时加载

RDB三种启动方式对比

RDB优点

RDB缺点

AOF 持久化

概念

AOF写数据过程

AOF执行策略

AOF 重写

什么叫AOF重写?

AOF 重写的作用

AOF 重写规则

AOF 重写方式

RDB 与 AOF 区别

RDB 与 AOF 应用场景

总结

Redis 数据类型

Redis 的作者 antirez 曾笑称 Redis 为一个数据结构服务器 (data structures server) , 这是一个非常准确的表述, Redis 的所有功能就是将数据以其固有的几种结构来保存, 并提供给用户操作这几种结构的接口。

Redis常用的有五种数据类型是: 字符串, 哈希, 列表, 集合, 有序集合。

strings 类型及操作

string 是最简单的类型, 你可以理解成与 Memcached 是一模一样的类型, 一个 key 对应一个 value, 其上支持的操作与 Memcached 的操作类似。但它的功能更丰富。

string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据, 比如 jpg 图片或者序列化的对象。从内部实现来看其实 string 可以看作 byte 数组, 最大上限是 1G 字节, 下面是

string 类型的定义:

```
1 struct sdshdr {
2     long len;
3     long free;
4     char buf[];
5 };
```

C 复制代码

其中:

- len 是 buf 数组的长度。
- free 是数组中剩余可用字节数
 - 由此可以理解为什么 string 类型是二进制安全的了, 因为它本质上就是个 byte 数组, 当然可以包含任何数据了
- buf 是个 char 数组用于存贮实际的字符串内容
 - 其实 char 和 c#中的 byte 是等价的, 都是一个字节。

另外 string 类型可以被部分命令按 int 处理。

比如 incr 等命令, 如果只用 string 类型, redis 就可以被看作加上持久化特性的 memcached。

当然 redis 对 string 类型的操作比 memcached 还是多很多的。

set 操作

设置 key 对应的值为 string 类型的 value。

例如我们添加一个 name= rushuni 的键值对, 可以这样做:

```
1 set name rushuni
```

JavaScript 复制代码

setnx 操作

设置 key 对应的值为 string 类型的 value。如果 key 已经存在, 返回 0, nx 是 not exist 的意思。例如我们添加一个 name= rushuni_new 的键值对, 可以这样做:

```
1 get name
2 setnx name rushuni_new
3 get name # 由于原来 name 有一个对应的值, 所以本次的修改不生效, 且返回码是 0。
```

Shell 复制代码

由于原来 name 有一个对应的值，所以本次的修改不生效，且返回码是 0。

setex 操作

setex 可以设置 key 对应的值为 string 类型的 value，并指定此键值对应的有效期。

例如我们添加一个 haircolor= red 的键值对，并指定它的有效期是 10 秒，可以这样做：

```
1 setex haircolor 10 red
2 get haircolor
3 get haircolor
```

Shell

复制代码

可见由于最后一次的调用是 10 秒以后了，所以取不到 haircolor 这个键对应的值。

setrange 操作

setrange 可以设置指定 key 的 value 值的子字符串。

例如我们希望将 rushuni 的 163 邮箱替换为 gmail 邮箱，那么我们可以这样做：

```
1 get name
2 setrange name 8 gmail.com
3 get name
```

Shell

复制代码

其中的 8 是指从下标为 （包含 8）的字符开始替换。

mset 操作

mset 可以一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置。

```
1 mset key1 rush1 key2 rush2
2 get key1
3 get key2
```

Shell

复制代码

msetnx 操作

msetnx 一次设置多个 key 的值，成功返回 ok 表示所有的值都设置了，失败返回 0 表示没有任何值被设置，但是不会覆盖已经存在的 key。

Shell | 复制代码

```
1 get key1
2 get key2
3 msetnx key2 rush1_new key3 rush2_new
4 get key2
5 get key3
```

可以看出如果这条命令返回 0，那么里面操作都会回滚，都不会被执行。

get 操作

获取 key 对应的 string 值,如果 key 不存在返回 nil。

例如我们获取一个库中存在的键 name，可以很快得到它对应的 value

Shell | 复制代码

```
1 get name
2 get name1
```

我们获取一个库中不存在的键 name1，那么它会返回一个 nil 以表时无此键值对

getset 操作

设置 key 的值，并返回 key 的旧值。

Shell | 复制代码

```
1 get name
2 getset name rushuni_new
3 get name
```

接下来我们看一下如果 key 不存的时候会什么样儿？

Shell | 复制代码

```
1 getset name1 aaa
```

可见，如果 key 不存在，那么将返回 nil

getrange 操作

getrange 可以获取指定 key 的 value 值的子字符串。

Shell | 复制代码

```
1 get name
2 getrange name 0 6
```

字符串左面下标是从 0 开始的

Shell | 复制代码

```
1 getrange name -7 -1
```

字符串右面下标是从-1 开始的

Shell | 复制代码

```
1 getrange name 7 100
```

当下标超出字符串长度时，将默认为是同方向的最大下标

mget 操作

一次获取多个 key 的值，如果对应 key 不存在，则对应返回 nil。

Shell | 复制代码

```
1 mget key1 key2 key3
```

key3 由于没有这个键定义，所以返回 nil。

incr 操作

对 key 的值做加加操作,并返回新的值。

注意 incr 一个不是 int 的 value 会返回错误，incr 一个不存在的 key，则设置 key 为 1

Shell | 复制代码

```
1 set age 20
2 incr age
3 get age
```

incrby 操作

incrby 同 incr 类似，加指定值，key 不存在时候会设置 key，并认为原来的 value 是 0

Shell | [复制代码](#)

```
1 get age
2 incrby age 5
3 get name
4 get age
```

decr 操作

对 key 的值做的是减减操作，decr 一个不存在 key，则设置 key 为-1

Shell | [复制代码](#)

```
1 get age
2 decr age
3 get age
```

decrby 操作

同 decr，减指定值。

Shell | [复制代码](#)

```
1 get age
2 decrby age 5
3 get age
```

decrby 完全是为了可读性，我们完全可以通过 incrby 一个负值来实现同样效果，反之一样。

Shell | [复制代码](#)

```
1 get age
2 incrby age -5
3 get age
```

append 操作

给指定 key 的字符串值追加 value,返回新字符串值的长度。

例如我们向 name 的值追加一个@163.com 字符串，那么可以这样做：

Shell | [复制代码](#)

```
1 append name @163.com
2 get name
```

strlen 操作

取指定 key 的 value 值的长度。


```
1 get name
2 strlen name
3 get age
4 strlen age
```

Redis 常用指令

key 操作分析

key应该有哪些操作？

- key是一个字符串，通过key获取redis中保存的数据
- 对于key自身状态的相关操作，例如：删除，判定存在，获取类型等
- 对于key有效性控制相关操作，例如：有效期设定，判定是否有效，有效状态的切换等
- 对于key快速查询操作，例如：按指定策略查询key

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
```

拓展操作

排序

Bash | 复制代码

```
1 lpush nums 12
2 lpush nums 11
3 lpush nums 13
4 lpush nums 10
5 sort nums
6 sort nums desc
7 1) "13"
8 2) "12"
9 3) "11"
10 4) "10"
```

改名

Bash | 复制代码

```
1 rename key newkey
2 renamenx key newkey
```

key 扩展操作（时效性控制）

为指定key设置有效期

Bash | 复制代码

```
1 expire key seconds
2 pexpire key milliseconds
3 expireat key timestamp
4 pexpireat key milliseconds-timestamp
```

获取key的有效时间

Bash | 复制代码

```
1 ttl key
2 pttl key
```

切换key从时效性转换为永久性

Bash | 复制代码

```
1 persist key
```

key 扩展操作（查询模式）

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 []匹配一个指定符号

```
1 keys *      查询所有
2 keys rush*  查询所有以rush开头
3 keys *rush  查询所有以rush结尾
4 keys ??rush 查询所有前面两个字符任意，后面以rush结尾
5 keys user:?  user:开头，最后一个字符任意
6 keys u[st]er:1  查询所有以u开头，以er:1结尾，中间包含一个字母，s或t
```

数据库指令

key 的重复问题

在这个地方我们来讲一下数据库的常用指令，在讲这个东西之前，我们先思考一个问题：

假如说你们十个人同时操作redis，会不会出现key名字命名冲突的问题。

一定会，为什么？因为你的key是由程序而定义的。你想写什么写什么，那在使用的过程中大家都在不停的加，早晚有一天他会冲突的。

redis在使用过程中，伴随着操作数据量的增加，会出现大量的数据以及对应的key。

那这个问题我们要不要解决？要！怎么解决呢？我们最好把数据进行一个分类，除了命名规范我们做统一以外，如果还能把它分开，这样是不是冲突的机率就会小一些了，这就是咱们下面要说的解决方案！

解决方案

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！

数据库的基本操作

切换数据库

```
1 select index
```


Bash

 复制代码

其他操作

```
1 ping
```

Bash


 复制代码

数据库扩展操作

数据移动

```
1 move key db
```


Bash

 复制代码

数据总量

```
1 dbsize
```


Bash

 复制代码

数据清除

```
1 flushdb flushall
```

Bash

 复制代码

Jedis

简介

在学习完 redis 后，我们现在就要用 Java 来连接 redis。

对于我们现在的数据来说，它是在我们的 redis 中，而最终我们是要做程序。

那么程序就要和我们的 redis 进行连接。干什么事情呢？

两件事：程序中有数据的时候，我们要把这些数据全部交给 redis 管理。同时，redis 中的数据还能取出来，回到我们的应用程序中。

在这个过程中，在 Java 与 redis 之间打交道的这个东西就叫做 Jedis。

简单说，Jedis 就是提供了 Java 与 redis 的连接服务的，里边有各种各样的 API 接口，你可以去调用它。

除了 Jedis 外，还有没有其他的这种连接服务呢？其实还有很多，了解一下：

Java 语言连接 redis 服务 Jedis (SpringData、Redis 、 Lettuce)

其它语言：C 、 C++ 、 C# 、 Erlang、 Lua 、 Objective-C 、 Perl 、 PHP 、 Python 、 Ruby 、 Scala

准备工作

jar包导入

```
1 <dependency>
2   <groupId>redis.clients</groupId>
3   <artifactId>jedis</artifactId>
4   <version>3.6.3</version>
5 </dependency>
```

Bash | 复制代码

客户端连接 redis


1. 连接 redis

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

Java | 复制代码

2. 操作 redis


Java

 复制代码

```
1 jedis.set("name", "rushuni");  
2 String value = jedis.get("name");
```

3. 关闭 redis 连接

Java

 复制代码

```
1 jedis.close();
```

Demo

```
1 package com.rushuni.redis_demo;
2
3 import redis.clients.jedis.Jedis;
4
5 import java.util.List;
6
7 /**
8  * @author rushuni
9  * @date 2021/08/07
10  */
11 public class RedisDemo {
12
13     public static void main(String[] args) {
14         // 1. 获取连接对象
15         Jedis jedis = new Jedis("localhost", 6379);
16         // 如果设置了密码
17         jedis.auth("rush123!");
18         // 2. 执行操作
19         jedis.set("name", "rushuni");
20         String value = jedis.get("name");
21         System.out.println(value);
22         jedis.lpush("list1", "a", "b", "c", "d");
23         List<String> list1 = jedis.lrange("list1", 0, -1);
24         for (String s: list1) {
25             System.out.println(s);
26         }
27         jedis.sadd("set1", "abc", "abc", "def", "poi", "cba");
28         Long len = jedis.scard("set1");
29         System.out.println(len);
30         // 3. 关闭连接
31         jedis.close();
32     }
33 }
```

Jedis简易工具类开发

前面我们做的程序还是有点儿小问题，就是我们的 Jedis 对象的管理是我们自己创建的，真实企业开发中是不可能让你去new一个的，那接下来咱们就要做一个工具类，简单来说，就是做一个创建Jedis的这样的一个工具。

基于连接池获取连接

JedisPool：Jedis提供的连接池技术

poolConfig:连接池配置对象

host:redis服务地址

port:redis服务端口号

JedisPool的构造器如下:

```
Java | 复制代码
1 public JedisPool(GenericObjectPoolConfig poolConfig, String host, int port) {
2     this(poolConfig, host, port, 2000, (String)null, 0, (String)null);
3 }
4 JedisPool pool = new JedisPool(new JedisPoolConfig(), "localhost", 6379);
```

4.2.2 封装连接参数

创建jedis的配置文件: jedis.properties

```
PowerShell | 复制代码
1 redis.host=127.0.0.1
2 redis.port=6379
3 redis.maxTotal=50
4 redis.maxIdle=10
```

加载配置信息

创建JedisUtils: com.rushuni.util.JedisUtils, 使用静态代码块初始化资源


```
1 package com.rushuni.redis_demo.util;
2
3 import redis.clients.jedis.Jedis;
4 import redis.clients.jedis.JedisPool;
5 import redis.clients.jedis.JedisPoolConfig;
6
7 import java.util.ResourceBundle;
8
9 /**
10  * jedis 工具类
11  * @author rushuni
12  * @date 2021/08/07
13  */
14 public class JedisUtils {
15
16     private static final JedisPool JEDIS_POOL;
17
18     static {
19         ResourceBundle bundle = ResourceBundle.getBundle("jedis");
20         int maxTotal = Integer.parseInt(bundle.getString("redis.maxTotal"));
21         int maxIdle = Integer.parseInt(bundle.getString("redis.maxIdle"));
22         String host = bundle.getString("redis.host");
23         int port = Integer.parseInt(bundle.getString("redis.port"));
24
25         //Jedis连接池配置
26         JedisPoolConfig jpc = new JedisPoolConfig();
27         jpc.setMaxTotal(maxTotal);
28         jpc.setMaxIdle(maxIdle);
29         JEDIS_POOL = new JedisPool(jpc, host, port);
30     }
31
32 }
```

获取连接

对外访问接口，提供jedis连接对象，连接从连接池获取，在JedisUtils中添加一个获取jedis的方法：
getJedis

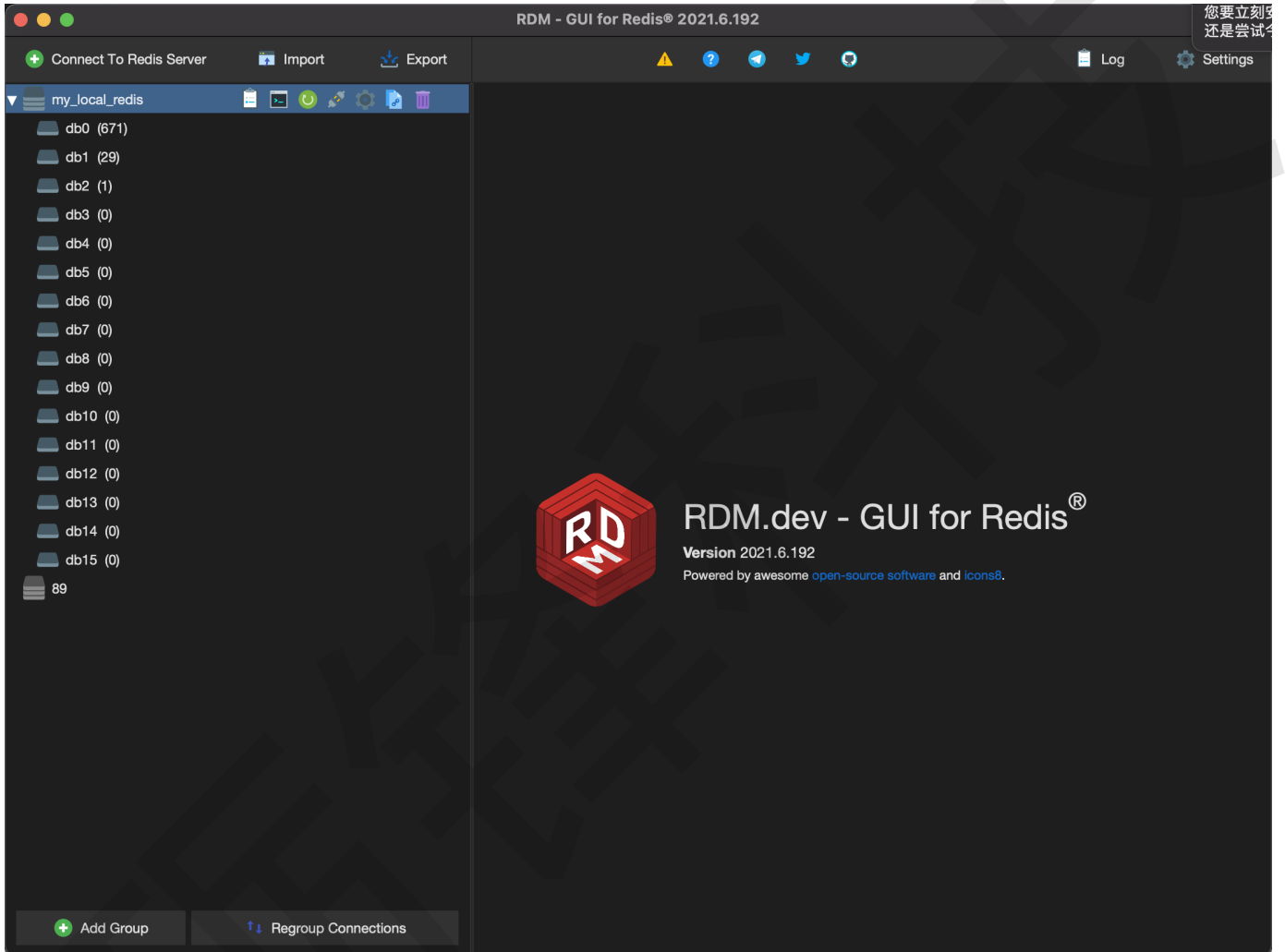
```
1 public static Jedis getJedis(){
2     return JEDIS_POOL.getResource();
3 }
```

作业

- 通过 jedis 完成 redis 五种数据类型的操作。

可视化客户端

- Redis Desktop Manager



Redis 高可用概述

我们知道，在web服务器中，高可用是指服务器可以正常访问的时间，衡量的标准是在多长时间内可以提供正常服务（99.9%、99.99%、99.999% 等等）。

但是在Redis语境中，高可用的含义似乎要宽泛一些，除了保证提供正常服务(如主从分离、快速容灾技术)，还需要考虑数据容量的扩展、数据安全不会丢失等。

持久化

持久化是最简单的高可用方法(有时甚至不被归为高可用的手段)，主要作用是数据备份，即将数据存储到硬盘，保证数据不会因进程退出而丢失。

复制

复制是高可用Redis的基础，哨兵和集群都是在复制基础上实现高可用的。

复制主要实现了数据的多机备份，以及对于读操作的负载均衡和简单的故障恢复。

缺陷：故障恢复无法自动化；写操作无法负载均衡；存储能力受到单机的限制。

哨兵

在复制的基础上，哨兵实现了自动化的故障恢复。

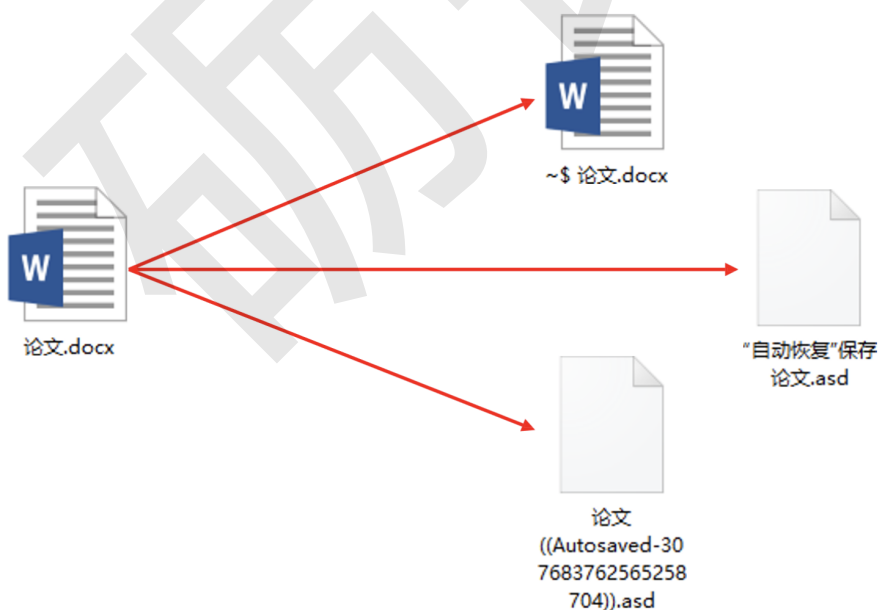
缺陷：写操作无法负载均衡；存储能力受到单机的限制。

集群

通过集群，Redis解决了写操作无法负载均衡，以及存储能力受到单机限制的问题，实现了较为完善的高可用方案。

持久化简介

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。



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！



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。

Redis 持久化

Redis 的持久化功能：Redis 是内存数据库，数据都是存储在内存中，为了避免进程退出导致数据的永久丢失，需要定期将 Redis 中的数据以某种形式(数据或命令)从内存保存到硬盘。

当下次 Redis 重启时，利用持久化文件实现数据恢复。

除此之外，为了进行灾难备份，可以将持久化文件拷贝到一个远程位置。

Redis持久化有两种方式：

- RDB 持久化
 - 前者将当前数据保存到硬盘

- AOF 持久化
 - 将每次执行的写命令保存到硬盘（类似于MySQL的binlog）；

```
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

由于 AOF 持久化的实时性更好，即当进程意外退出时丢失的数据更少，因此 AOF 是目前主流的持久化方式，不过 RDB 持久化仍然有其用武之地。

RDB 持久化

概述

RDB 持久化是将当前进程中的数据生成快照保存到硬盘(因此也称作快照持久化)，保存的文件后缀是 rdb。

当 Redis 重新启动时，可以读取 rdb文件恢复数据。

触发条件

RDB持久化的触发分为手动触发和自动触发两种。

手动触发

save 命令和 bgsave 命令都可以生成 RDB 文件。

save 命令会阻塞 Redis 服务器进程，直到 RDB 文件创建完毕为止，在 Redis 服务器阻塞期间，服务器不能处理任何命令请求。

```
1 save
```

Shell

复制代码

而 bgsave 命令会创建一个子进程，由子进程来负责创建 RDB 文件，父进程(即 Redis 主进程)则继续处理请求。

```
1 bgsave
```

Shell 复制代码

bgsave 命令执行过程中，只有 fork 子进程时会阻塞服务器，而对于 save 命令，通过下面讲解，我们将会知道，由于 save 整个过程都会阻塞服务器，因此 save 事实上已基本被废弃，线上环境要杜绝 save 的使用。

也就是说，在自动触发 RDB 持久化时，Redis 也会选择 bgsave 而不是 save 来进行持久化；

自动触发

自动触发最常见的情况是在配置文件中通过 save m n，指定当 m 秒内发生 n 次变化时，会触发 bgsave。

```
1 save m n
```

Shell 复制代码

例如，查看 redis 的配置文件，可以看到如下配置信息：

```
# You can set these explicitly by uncommenting the three following lines.
#
# save 3600 1
# save 300 100
# save 60 10000
```

其中 save 3600 1 的含义是：时间在 3600 秒内，如果 redis 数据发生了至少1次变化，则执行 bgsave；

save 300 100 和 save 60 10000同理。

当三个save条件满足任意一个时，都会引起bgsave的调用。

save 指令

手动执行一次保存操作

```
1 save
```

Bash 复制代码

save 指令相关配置

设置本地数据库文件名，默认值为 dump.rdb，通常设置为dump-端口号.rdb

```
1 dbfilename filename
```

Shell

复制代码

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

```
1 dir path
```

Shell

复制代码

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

```
1 rdbcompression yes|no
```

Shell

复制代码

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

```
1 rdbchecksum yes|no
```

Shell

复制代码

save 指令的问题

需要注意一个问题，如果现在有四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指令对应的一个问题，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。

bgsave指令

之前我们讲到了当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

手动启动后台保存操作，但不是立即执行

```
1 bgsave
```

Bash

复制代码

bgsave指令相关配置

后台存储过程中如果出现错误现象，是否停止保存操作，默认yes

```
1 stop-writes-on-bgsave-error yes|no
```

Bash

复制代码

其他

```
1 dbfilename filename
2 dir path
3 rdbcompression yes|no
4 rdbchecksum yes|no
```

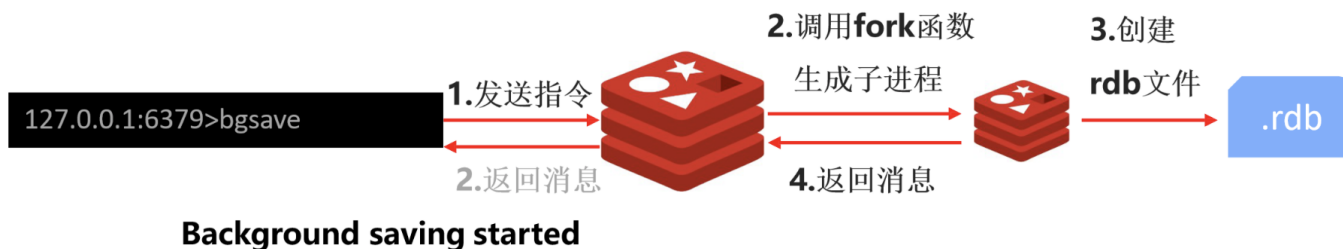
Bash

复制代码

bgsave 指令工作原理

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。



save配置自动执行

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
```

Shell

复制代码

参数

- second：监控时间范围
- changes：监控key的变化量

例子：

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

Shell

复制代码

其他相关配置：

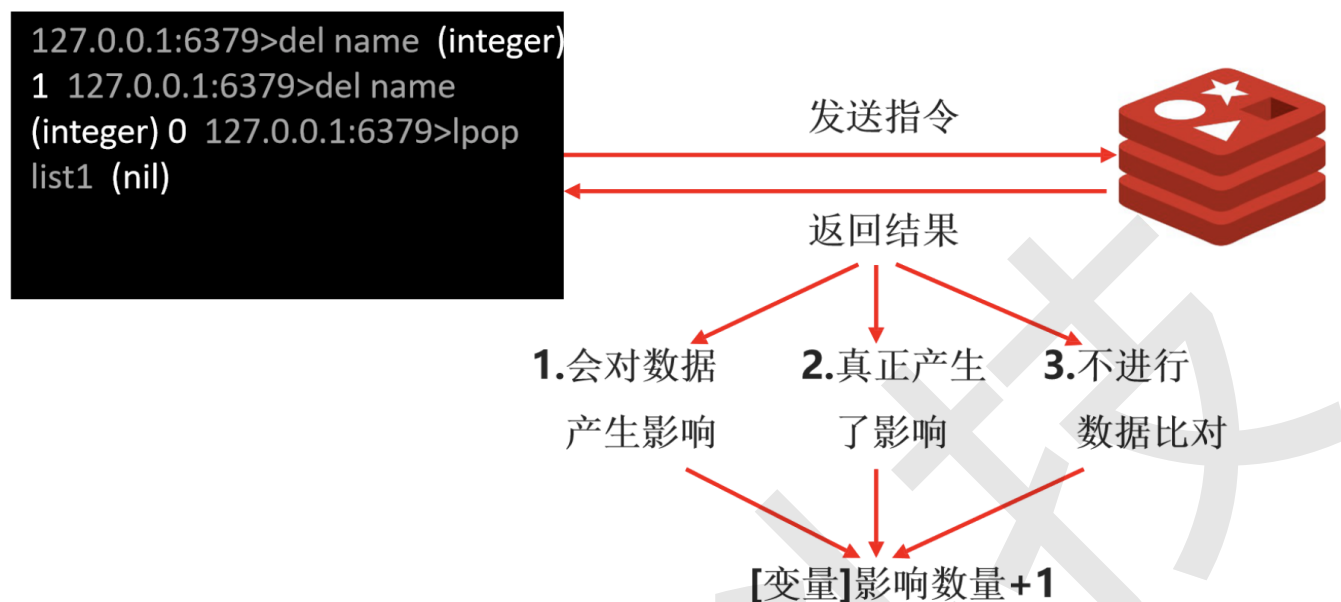
```
1 dbfilename filename
2 dir path
3 rdbcompression yes|no
4 rdbchecksum yes|no
5 stop-writes-on-bgsave-error yes|no
```

Shell

复制代码

save配置工作原理

RDB启动方式 ——save配置工作原理



注意： save配置要根据实际业务情况进行设置，频度过高或过低都会出现性能问题，结果可能是灾难性的 save配置启动后执行的是bgsave操作

RDB 的启动时加载

RDB 文件的载入工作是在服务器启动时自动执行的，并没有专门的命令。

但是由于 AOF 的优先级更高，因此当 AOF 开启时，Redis 会优先载入 AOF 文件来恢复数据。

只有当 AOF 关闭时，才会在 Redis 服务器启动时检测 RDB 文件，并自动载入。

服务器载入 RDB 文件期间处于阻塞状态，直到载入完成为止。

Redis 载入 RDB 文件时，会对 RDB 文件进行校验，如果文件损坏，则日志中会打印错误，Redis 启动失败。

RDB三种启动方式对比

方式	save指令	bgsave指令
读写	同步	异步
阻塞客户端指令	是	否
额外内存消耗	否	是
启动新进程	否	是

RDB特殊启动形式

- 服务器运行过程中重启

```
1 debug reload
```

Bash 复制代码

- 关闭服务器时指定保存数据

```
1 shutdown save
```

Shell 复制代码

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
- Redis 的众多版本中未进行 RDB 文件格式的版本统一，有可能出现各版本服务之间数据格式无法兼容现象

AOF 持久化

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，目前是 Redis 持久化的主流方式。

AOF写数据过程

启动AOF相关配置

开启AOF持久化功能，默认no，即不开启状态

```
1 appendonly yes|no
```

Shell

复制代码

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

```
1 appendfilename filename
```

Shell

复制代码

AOF 持久化文件保存路径，与 RDB 持久化文件的配置保持一致即可。

```
1 dir
```

Shell

复制代码

AOF 写数据策略，默认为everysec

```
1 appendfsync always|everysec|no
```

Shell

复制代码

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

AOF 重写

场景：AOF写数据遇到的问题，如果连续执行如下指令该如何处理

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 重写的作用

- 降低磁盘占用量，提高磁盘利用率
- 提高持久化效率，降低持久化写时间，提高IO性能
- 降低数据恢复用时，提高数据恢复效率

AOF 重写规则

- 进程内具有时效性的数据，并且数据已超时将不再写入文件
- 非写入类的无效指令将被忽略，只保留最终数据的写入命令
 - 如del key1、hdel key2、srem key3、set key4 111、set key4 222等
 - 如select指令虽然不更改数据，但是更改了数据的存储位置，此类命令同样需要记录
- 对同一数据的多条写命令合并为一条命令
 - 如 lpush list1 a、lpush list1 b、lpush list1 c 可以转化为：lpush list1 a b c。


为防止数据量过大造成客户端缓冲区溢出，对 list、set、hash、zset 等类型，每条指令最多写入 64 个元素

AOF 重写方式

- 手动重写

```
1 bgrewriteaof
```

Shell

 复制代码

AOF手动重写 —— bgrewriteaof指令工作原理



Background append only file rewriting started

- 自动重写触发条件设置

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

Shell

复制代码

- 自动重写触发比对参数（运行指令info Persistence获取具体信息）

```
1 aof_current_size
2 aof_base_size
```

Shell

复制代码

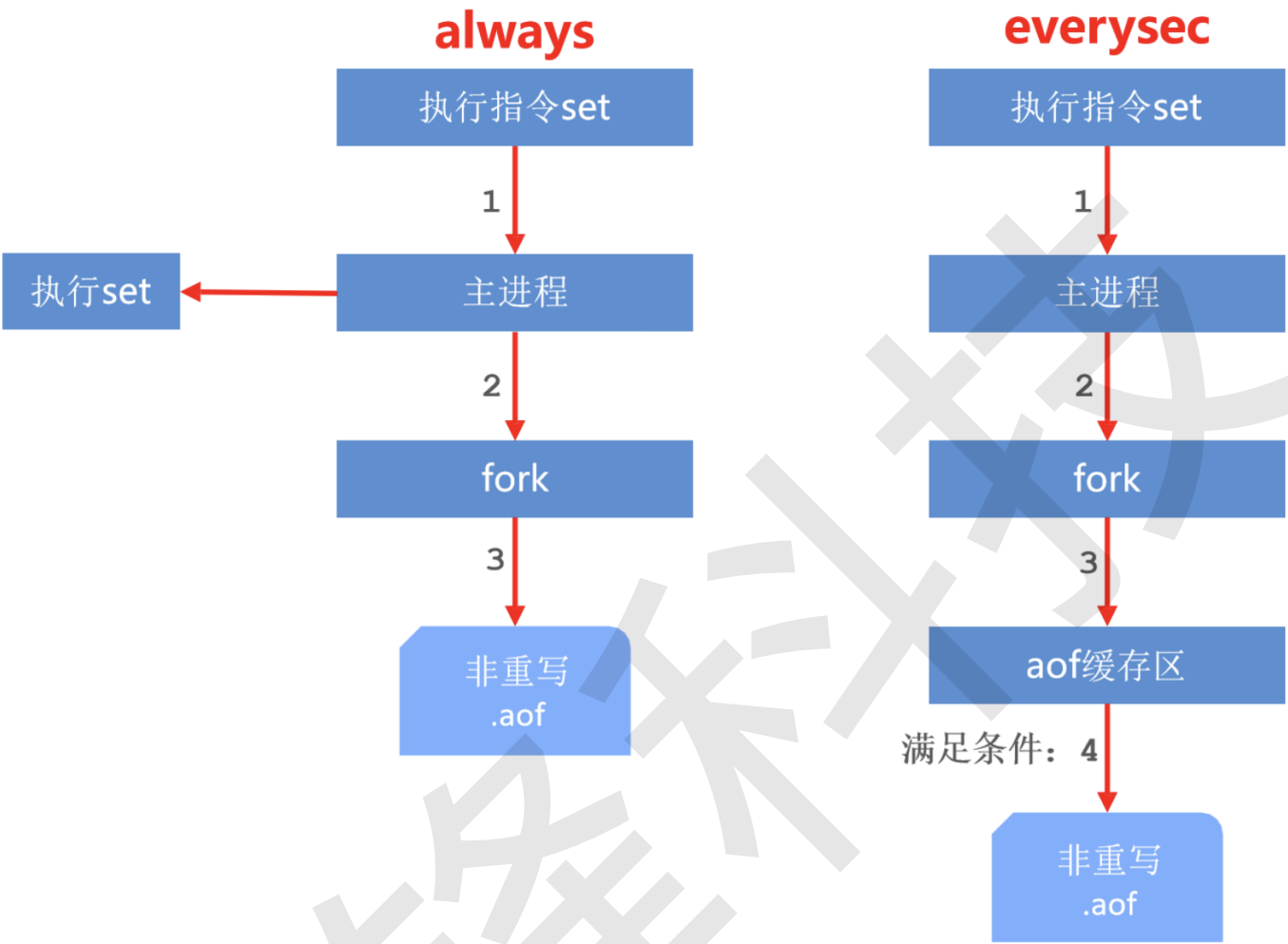
- 自动重写触发条件公式：

$aof_current_size > auto-aof-rewrite-min-size$

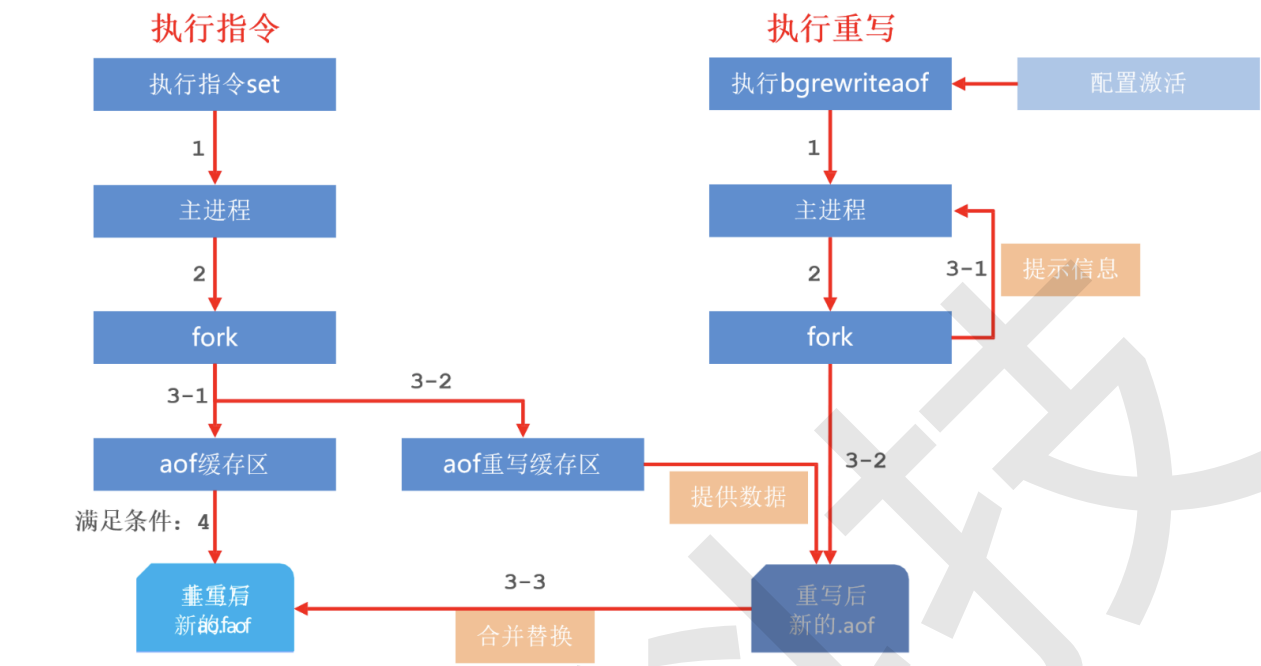
$$\frac{aof_base_size - aof_current_size}{aof_base_size} \geq auto-aof-rewrite-percentage$$

AOF工作流程及重写流程

AOF工作流程



AOF重写流程



RDB 与 AOF 区别

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

RDB 与 AOF 应用场景

RDB与AOF的选择之惑

- 对数据非常敏感，建议使用默认的 AOF 持久化方案

AOF 持久化策略使用 everysecond，每秒钟 fsync 一次。该策略 redis 仍可以保持很好的处理性能，当出现问题时，最多丢失0-1秒内的数据。

注意：由于AOF文件存储体积较大，且恢复速度较慢，因此

- 数据呈现阶段有效性，建议使用 RDB 持久化方案
- 数据可以良好的做到阶段内无丢失（该阶段是开发者或运维人员手工维护的），且恢复速度较快，阶段点数据恢复通常采用RDB方案

注意：利用 RDB 实现紧凑的数据持久化会使 Redis 性能降的很低，慎重。

总结

- 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