北京航空航天大学

软件测试分析报告

Redis

SY1406108 陈志伟　　SY1406112 王珊珊

SY1406311 林 璐　　SY1406117 王志鹏

2015/06/06

版本变更历史

版本	提交日期	编制人	说明
Version 1.0	2015-05-17	陈志伟、林璐	初步完成测试分析报告的 1.0 版
Version2.0	2015-05-19	王志鹏	初步完成测试分析报告的 2.0 版
Version 3.0	2015-05-19	王珊珊	初步完成测试分析报告的 3.0 版
Version 4.0	2015-05-20	陈志伟	完成测试分析报告 4.0 版
最终版	2015-06-06	全体组员	完成测试分析报告最终版

目录

1 编写目的

本文档主要用来说明测试阶段的工作内容和测试结果。首先叙述了小组对测试工作的分析过程，给出了将要撰写的测试用例规格与对应的测试用例的对照表，对每个测试用例按照 RUCM4test 的标准进行了说明，最后给出了测试过程、测试数据和测试结果分析。

2 规范及对照表

2.1 设计测试用例

测试用例（Test Case）是指对一项特定的软件产品进行测试任务的描述，体现测试方案、方法、技术和策略，目的是能够将软件测试的行为转化成可管理的模式；同时测试用例也是将测试具体量化的方法之一，不同类别的软件，测试用例是不同的。

要使最终用户对软件感到满意，最有力的举措就是对最终用户的期望加以明确阐述，以便对这些期望进行核实并确认其有效性。测试用例反映了要核实的需求。然而，核实这些需求可能通过不同的方式并由不同的测试员来实施。

测试用例很重要，主要有以下几个方面：

测试用例构成了设计和制定测试过程的基础。

测试的“深度”与测试用例的数量成比例。由于每个测试用例反映不同的场景、条件或经由产品的事件流，因而，随着测试用例数量的增加，我们对产品质量和测试流程也就越有信心。

判断测试是否完全的一个主要评测方法是基于需求的覆盖，而这又是以确定、实施和/或执行的测试用例的数量为依据的。类似下面这样的说明：“95％ 的关键测试用例已得以执行和验证”，远比“我们已完成 95 % 的测试”更有意义。

测试工作量与测试用例的数量成比例。根据全面且细化的测试用例，可以更准确地估计测试周期各连续阶段的时间安排。

测试设计和开发的类型以及所需的资源主要都受控于测试用例。

测试用例通常根据它们所关联关系的测试类型或测试需求来分类，而且将随类型和需求进行相应地改变。最佳方案是为每个测试需求至少编制两个测试用例，一个测试用例用于证明该需求已经满足，通常称作正面测试用例；另一个测试用例反映某个无法接受、反常或意外的条件或数据，用于论证只有在所需条件下才能够满足该需求，这个测试用例称作负面测试用例。

总之，测试用例是测试工作的指导，是软件测试的必须遵守的准则，更是软件测试质量稳定的根本保障。

2.2 测试规格与测试用例对照表

依据测试需求规格说明书，小组选择了各模块中具有代表性的测试规格进行了测试用例的详细设计。测试规格与测试用例的对照表如下：

模块	测试规格	测试用例
服务器	启动服务器	启动服务器测试
	自定义服务器配置	自定义服务器配置正常测试
		自定义服务器配置异常测试
	服务器性能	服务器性能测试
RDB 持久化	同步回写 SAVE	同步回写 SAVE 功能正常测试
		同步回写 SAVE 功能阻塞测试
	异步回写 BGSAVE	异步回写 BGSAVE 功能正常测试
		异步回写 BGSAVE 功能异常测试
	载入数据	载入数据测试
AOF 持久化	命令同步	命令同步测试
	AOF 重写	AOF 重写测试
	AOF 文件还原	AOF 文件还原测试
客户端	命令请求处理	命令请求处理测试
	命令请求读取	命令请求读取测试
	命令请求执行	命令请求执行测试
	命令请求回复	命令请求回复测试

3 服务器模块

3.1 启动服务器测试

3.1.1 测试目标

本测试用例对应启动服务器测试规格，主要测试服务器正常启动时的处理方式。正常启动时，测试程序会直接输出服务器返回的 ping 值，并进行简单的字符串存储操作；但若测试程序抛出了异常，即可判断服务器启动异常。

3.1.2 测试用例

Test Case Specification		
Name	启动服务器测试	
Brief Description	测试 Redis 服务器是否正常启动	
Precondition	系统已安装 Redis 数据库	
Tester	测试员	
Dependency	None	
Test Setup	Name	准备测试文件 RedisExample.Java
	Description	测试员编写好用于测试的 Java 源文件
Basic Flow (Test Setup)	Steps	
	1	测试员在 eclipse 中配置好编写 Redis 测试程序所需的 jar 包；
	2	测试员编写 Java 测试源文件；
	3	测试员通过人工走查方式检查源文件；
	Postcondition (Test Oracle)	指定目录的 RedisExample.Java 源文件已存在； 测试代码没有逻辑和语法错误；
Basic Flow (Test Sequence)	Steps	
	1	测试员在 eclipse 中编译 Java 测试程序；
	2	测试员在 eclipse 中运行 Java 测试程序；
	3	测试程序使用指定的 IP 地址连接 redis 服务器；
	4	服务器返回 ping 值；
	5	测试程序 VALIDATES THAT 服务器正常启动；
	6	测试程序执行简单的字符串存储操作；
	Postcondition (Test Oracle)	测试程序输出服务器正常运行的提示； 测试程序输出存储的字符串； 服务器正常启动的情况被测试；
Specific Alternative Flows (Test Sequence)	RFS 5	
	1	测试程序抛出服务器连接异常；
	2	ABORT
	Postcondition (Test Sequence)	服务器启动异常；

3.1.3 测试代码

```
1  import redis.clients.jedis.Jedis;
2  public class RedisExample{
3      public static void main(String[] args) {
4          //Connecting to Redis server on localhost
5          Jedis jedis = new Jedis("localhost");
6
7          //check whether server is running or not
8          System.out.println("Server is running: "+jedis.ping());
9
10         jedis.set( "str", "Hello world!" ) ;
11         String output = jedis.get( "str") ;
12         System.out.println(output);
13 }
14 }
15
```

3.1.4 测试结果及分析

```
<terminated> RedisExample [Java Application] C:\Program Files\Java\jre8\bin\javaw.exe (2015年4月26日 上午11:40:41)
Server is running: PONG
Hello world!
```

期望结果：

输出服务器正在运行的提醒和 ping 值，输出从服务器取回的字符串。

实际结果及分析：

测试程序没有抛出异常，而是显示服务器正在运行，并输出服务器返回的 ping 值，同时测试程序还向服务器存储了一个简单的字符串，再用查询操作取得了这个字符串并进行了正确的输出。这些证明了服务器正常启动且正在运行，与预期结果一致，故测试通过。

3.2 自定义服务器配置正常测试

3.2.1 测试目标

本测试用例对应自定义服务器配置测试规格，主要测试在自定义配置参数合理的情况下，即测试需求能满足时，服务器启动时的处理方式。在启动时，服务器读取配置文件中的参数设置自身参数，并显示开始界面，即可判断自定义的服务器配置文件正常。

3.2.2 测试用例

Test Case Specification		
Name	自定义服务器配置正常测试	
Brief Description	测试在用户定义的配置正常的情况下对服务器的影响	
Precondition	系统已安装 Redis 数据库	
Tester	测试员	
Dependency	None	
Test Setup	Name	准备自定义的配置文件
	Description	测试员编写好用于测试的自定义配置文件
Basic Flow (Test Setup)	Steps	
	1	测试员编写 Redis 的配置文件；
	2	测试员给配置设置合理的参数；
	3	测试员将配置文件保存到指定目录；
	4	测试员通过人工走查方式检查配置文件；
	Postcondition (Test Oracle)	指定目录的自定义配置文件已存在； 配置文件符合 Redis 的配置文件相关规范；
Basic Flow (Test Sequence)	Steps	
	1	测试员在命令行中输入服务器可执行程序名和自定义的配置文件名；
	2	服务器读入配置文件中的相关配置参数的值；
	3	服务器 VALIDATES THAT 配置文件中的参数有效；
	4	服务器按配置文件中的参数的值设置自身的参数；
	Postcondition (Test Oracle)	服务器开始运行，在命令行中输出启动界面； 配置文件中的参数有效的情况被测试；

3.2.3 测试代码

- 自定义配置文件（显示将改变的参数）

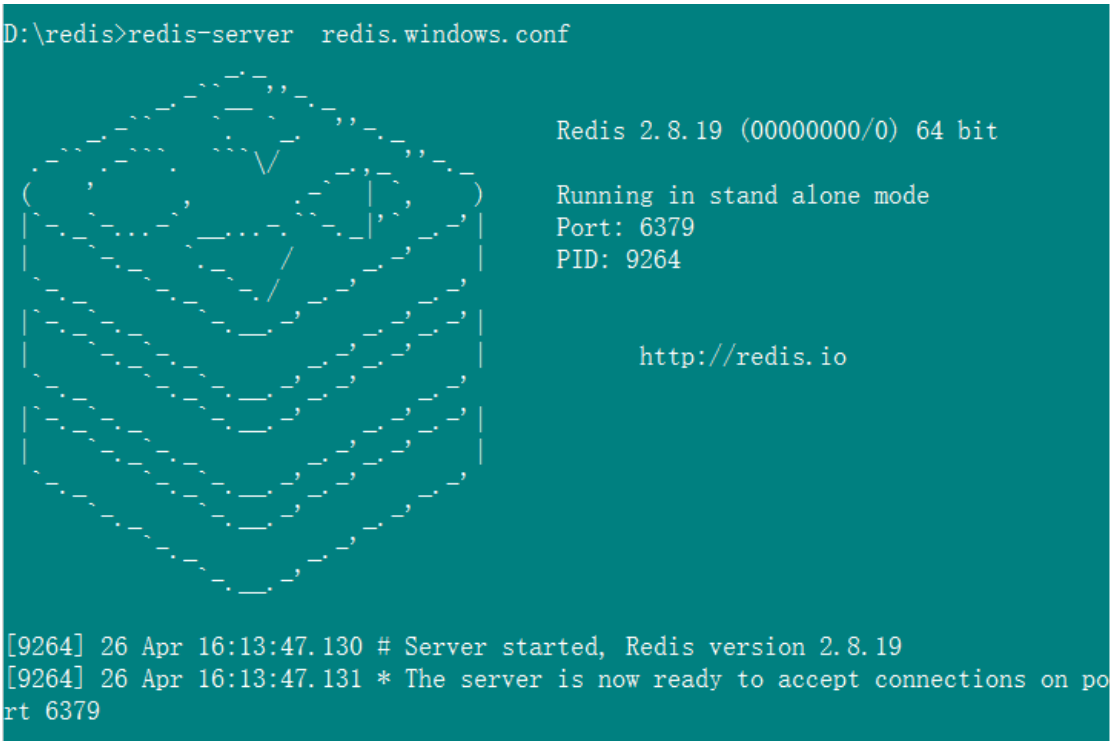
```
#
# maxheap <bytes>
maxheap 102400000
```

这是自定义配置文件的关键部分的截图。maxheap 表示堆栈的最大内存，在自定义的配置文件中，将修改 maxheap 的值，这里取了一个较小的 102,400,000byte。

- 测试指令

```
D:\redis>redis-server  redis.windows.conf
```

3.2.4 测试结果及分析



期望结果：

在配置参数设置合理的情况下，使用自定义的配置文件能让服务器正常启动运行。

实际结果及分析：

由上图中的结果可知，服务器已经正常启动运行，等待着客户端的连接，所以测试通过。

3.3 自定义服务器配置异常测试

3.3.1 测试目标

本测试用例也是对应自定义服务器配置测试规格，主要测试在自定义配置参数异常的情况下，即测试需求不能满足时，服务器启动时的处理方式。在启动时，服务器读取配置文件中的参数设置自身参数，由于测试设置异常导致服务器没法启动运行，故服务器退出给出出错的信息， 即可测试配置异常的情况。

3.3.2 测试用例

Test Case Specification	
Name	自定义服务器配置异常测试
Brief Description	测试在用户定义的配置异常的情况下对服务器的影响

Precondition	系统已安装 Redis 数据库	
Tester	None	
Dependency	None	
Test Setup	Name	准备自定义的配置文件
	Description	测试员编写好用于测试的自定义配置文件
Basic Flow (Test Setup)	Steps	
	1	测试员编写 Redis 的配置文件； 测试员给配置设置异常的参数；
	2	测试员将配置文件保存到指定目录；
	3	测试员通过人工走查方式检查配置文件；
	Postcondition (Test Oracle)	指定目录的自定义配置文件已存在； 配置文件符合 Redis 的配置文件相关规范；
Basic Flow (Test Sequence)	Steps	
	1	测试员在命令行中输入服务器可执行程序名和自定义的配置文件名；
	2	服务器读入配置文件中的相关配置参数的值；
	3	服务器 VALIDATES THAT 配置文件中的参数无效；
	4	服务器启动失败并给出配置文件中参数设置错误；
	5	ABORT
	Postcondition (Test Oracle)	配置文件中参数设置异常的情况被测试；

3.3.3 测试代码

（1）测试代码

- 自定义配置文件（显示改变的参数）

```
#
# maxheap <bytes>
maxheap 10240000000
```

- 测试指令

```
D:\redis>redis-server  redis.windows.conf
```

3.3.4 测试结果及分析

```
D:\redis>redis-server  redis.windows.conf
[10368] 26 Apr 16:36:23.219 #
The Windows version of Redis allocates a large memory mapped file for sharing
the heap with the forked process used in persistence operations. This file
will be created in the current working directory or the directory specified by
the 'heapdir' directive in the .conf file. Windows is reporting that there is
insufficient disk space available for this file (Windows error 0x70).

You may fix this problem by either reducing the size of the Redis heap with
the --maxheap flag, or by moving the heap file to a local drive with sufficient
space.
Please see the documentation included with the binary distributions for more
details on the --maxheap and --heapdir flags.

Redis can not continue. Exiting.
```

期望结果：

让服务器无法启动运行并退出，给出异常的原因。

实际结果及分析：

由上图中的结果可知，服务器启动运行失败。这是因为把配置文件中的maxheap 参数设置为一个较大的值，即 10,240,000,000bytes，由于堆栈的最大内存分配过大，导致磁盘没有足够的容纳空间，所以服务器启动失败，故测试通过。

3.4 服务器性能测试

3.4.1 测试目标

本测试用例对应服务器性能测试规格，主要测试在每个客户端对服务器请求数量一定的情况下，研究不同的数量的客户端，即并发数对服务器性能的影响。通过把每个客户端的请求数量设置为恒定的 1000 次，然后按一定的步长改变并发数，即增加客户端的数量，分别得到完成所有请求所用的时间，就可推算出对服务器性能的影响，从而完成测试过程。

3.4.2 测试用例

Test Case Specification		
Name	服务器性能测试	
Brief Description	模拟测试不同的并发数对服务器的性能影响	
Precondition	服务器已正常启动	
Tester	测试员	
Dependency	None	
Test Setup	Name	准备测试文件 Command.Java

	Description	测试员编写好用于测试的 Java 源文件
Basic Flow (Test Setup)	Steps	
	1	测试员编写 java 测试源文件；
	2	测试员将源文件保存到指定目录；
	3	测试员通过人工走查方式检查源文件；
	Postcondition (Test Oracle)	指定目录的 Java 源文件已存在； Java 代码没有逻辑和语法错误；
Basic Flow (Test Sequence)	Steps	
	1	测试员设置测试的并发数 N 的最小值为 0，最大值为 63000，步进为 1000；
	2	测试程序初始化并发数 N 为 0；
	3	DO
	4	测试程序调用性能测试工具向服务器以并发数N发送恒定数量的请求；
	5	测试工具给出测试结果；
	6	测试程序记录测试数据；
	7	测试程序把并发数N更改为N + 1000；
	8	UNTIL N > 63000
	9	测试员对测试程序所返回的测试数据进行分析；
	Postcondition (Test Oracle)	在请求数量一定的情况下，服务器在不同的并发数下的性能得到测试

3.4.3 测试代码

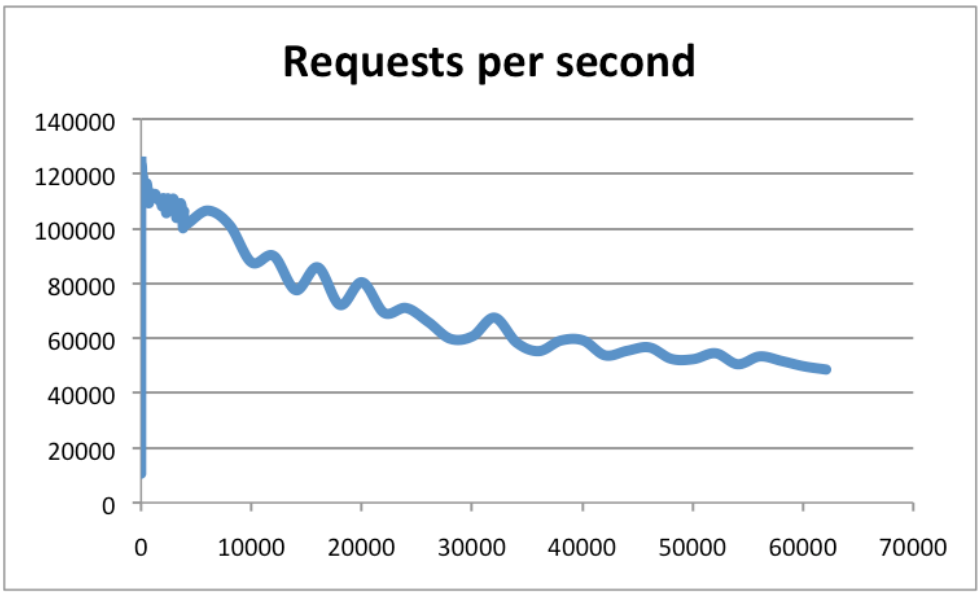
```
import java.io.*;

public class Command {
    public static void exeCmd(FileWriter fw, String commandStr, int c) {
        BufferedReader br = null;
        try {
            Process p = Runtime.getRuntime().exec(commandStr + c);
            br = new BufferedReader(new InputStreamReader(p.getInputStream()));
            String line = null;
            StringBuilder sb = new StringBuilder();
            while ((line = br.readLine()) != null) {
                sb.append(line + "\n");
                int a = line.indexOf("requests per second");
                if(a >= 0) {
                    //保存每一次的测试结果
                    String tmpStr = line.substring(0, a-1);
                    fw.write(c + "\t" + tmpStr + "\n");
                }
            }
            System.out.println(sb.toString());
            fw.flush();
        } catch (Exception e) {
            e.printStackTrace();
        }
        finally {
            if (br != null) {
                try {
                    br.close();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static void main(String[] args) {
        //创建一个FileWriter对象，用来存储测试数据
        FileWriter fw = null;
        try {
            fw = new FileWriter("data.txt");
            String commandStr = "D:\\redis\\redis-benchmark -h 127.0.0.1 -p 6379 -t set";
            for(int i=0; i<=63000; i+=1000) {
                Command.exeCmd(fw, commandStr + " -n " + i*1000 + " -c ", i);
            }
            fw.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

3.4.4 测试结果及分析



期望结果：

服务器的性能曲线应该先上升，然后到达峰值后开始下降。

实际结果及分析：

由上图可知，在并发数较小时服务器的性能上升得非常快，这主要是因为开始时客户端的个数较小，而每个客户端发送的请求数一定，故总的请求数也较小，此时服务器无法准确的估计出每秒处理请求数量的值。而到了客户端数量为100～150 这个范围时，服务器每秒处理请求的数量达到峰值。最后，当客户端数量达到 60000 时，服务此时的性能只有峰值性能的一半，所以 60000 为 Redis 系统的阈值。在实际中，Redis 每秒处理的请求数都是上万的，此时服务器一定不是运行于性能的峰值，但，虽然性能有所损害，每秒能服务的客户端却变得更多，这能显著的节省成本，性能和并发数的折中在实际中非常有意义，总之测试通过。

4 RDB 持久化模块

Redis 提供了两种持久化策略，RDB 和 AOF。RDB 是默认的，它定时创建数据库的完整磁盘镜像，即 dump.rdb 文件。创建镜像的时间间隔是可以设置的，假如每 5 分钟创建一次镜像，那么当系统崩溃时用户可能会丢失 5 分钟的数据。因此，**RDB 不是一个可靠性很高的方案，但是性能不错**。RDB 非常容易备份，用户直接将 dump.rdb 文件复制即可。为了提供更好的可靠性，Redis 支持 AOF，即将操作写入日志中（appendonly.aof 文件）。写日志的策略可以是每秒一次或每次操作一次，显然每秒一次意味着用户可能丢失 1 秒的数据，而**每次 AOF 操作的可靠性最高，但是性能最差**。日志文件可能会增长到非常大，因此 Redis 后台会执行 rewrite 操作整理日志。AOF 不适合备份。

Redis 推荐使用 RDB，以及在需要可靠性的时候用 RDB+AOF，不推荐单独使用 AOF。Redis 为了减少磁盘的负载，任何时刻都不会同时执行写镜像和写日志。

第四部分主要对 RDB 策略进行测试，第五部分主要对 AOF 持久化策略进行测试。

4.1 同步回写 SAVE 测试

4.1.1 测试目标

同步回写 SAVE 在 Redis 主进程中直接调用 rdbSave()函数，阻塞主进程，直到保存完成为止。在主进程阻塞期间，服务器不能处理客户端任何请求。

一般来说，在生产环境中很少用 SAVE 操作，因为会阻塞所有客户端的请求，保存数据库的任务通常由 BGSAVE 命令异步保存。然而，如果负责保存数据的后台子进程不幸出现问题时，SAVE 可以作为保存数据的最后手段来使用。

本部分测试在 window 环境下：

1) 当 Redis 开启 rdb 方式，数据变更后，手动进行 SAVE 保存的功能。
2) 验证其主进程被阻塞时，其他客户端发送请求，服务器报出异常的功能。

4.1.2 测试用例

4.1.2.1 同步回写 SAVE 功能正常测试

Test Case Specification		
Name	同步回写 SAVE 功能正常测试	
Brief Description	测试能否正确读取用户配置命令，服务器进行同步回写 SAVE 操作	
Precondition	Redis 在 windows 下正确安装	
Tester	测试员	
Dependency	None	
Test Setup	Name	配置 redis.conf 文件
	Description	配置 redis.conf 文件，以选择 RDB 持久化方式，并调用 SAVE 命令
Basic Flow (Test Setup)	Steps	
	1	测试员打开 redis.conf 文件
	2	测试员写入命令”**save “” ”**，不设置快照保存周期，而手动进行保存
	3	写入命令”dbfilename dump.rdb”，设置快照保存文件的文件名
	4	写入命令”dir ./”，设置备份文件放置路径
	5	写入命令”appendonly no”，选择非 AOF 方式，也即 RDB 方式
	Postcondition (Test Oracle)	配置完成，准备启动 Redis 进行快照 SAVE 测试
Basic Flow	Steps	

(Test Sequence)	1	打开 windows 下的 cmd 运行窗口
	2	输入”cd PATH-TO-REDIS”，切换到 Redis 文件夹下
	3	输入”redis-server.exe redis.conf”，启动 Redis
	4	输入”redis-cli.exe”，打开一个窗口运行客户端 A
	5	创建 Eclipse 项目，引入 jedis 客户端包
	6	编写测试程序 A-test，实现对 100 条数据的修改
	7	客户端 A 中输入”save”，手动选择 save 方式保存快照
	7	客户端 A 输入命令”get xxx”，获取被修改键的最新值
	Postcondition (Test Oracle)	最新的保存值恰是测试程序逻辑所得，手动同步保存 SAVE 的功能被测试

4.1.2.2 同步回写 SAVE 功能阻塞测试

Test Case Specification		
Name	同步回写 SAVE 功能阻塞测试	
Brief Description	测试进行同步回写操作时，主进程被阻塞，其他客户端发出请求时，服务器能否正常报错	
Precondition	Redis 在 windows 下可正常进行 RDB 持久化	
Tester	测试员	
Dependency	None	
Test Setup	Name	配置 redis.conf 文件
	Description	配置 redis.conf 文件，以选择 RDB 持久化方式，并调用 SAVE 命令
Basic Flow (Test Setup)	Steps	
	1	测试员打开 redis.conf 文件
	2	测试员写入命令”**save “” ”**， 不设置快照保存周期，而手动进行保存
	3	写入命令”dbfilename dump.rdb”，设置快照保存文件的文件名
	4	写入命令”dir ./”，设置备份文件放置路径
	5	写入命令”appendonly no”，选择非 AOF 方式，也即 RDB 方式
	Postcondition (Test Oracle)	配置完成，准备启动 Redis 进行快照 SAVE 测试
Basic Flow (Test Sequence)	Steps	
	1	打开 windows 下的 cmd 运行窗口
	2	输入”cd PATH-TO-REDIS”，切换到 Redis 文件夹下
	3	输入”redis-server.exe redis.conf”，启动 Redis
	4	创建 Eclipse 项目，引入 jedis 客户端包
	5	编写测试程序 A-test，实现对 1000000 条数据的修改
	6	编写测试程序 B-test，实现同时调用 save 命令和 set

		命令，以观察阻塞现象
	Postcondition (Test Oracle)	观察客户端接收的错误信息，服务器在 save 阻塞过程中拒绝客户端其他请求的报错功能被测试

4.1.3 测试代码

1) 同步回写 SAVE 功能正常测试
   a) 创建 Jedis 类，连接 redis 服务器，实现数据的读写方法：

```
package SE.lin;

import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;

public class RDBtester {

  JedisPool pool;
  Jedis jedis;

public void 14etup() {
     pool = new JedisPool(new JedisPoolConfig(),
“127.0.0.1”);

     jedis = pool.getResource();
     // jedis.auth(“password”);
  }

  public void testWriteData(String key, String value) {
     // ---添加数据---
     // jedis.set(“name1”, “value”);
     // System.out.println(jedis.get(“name1”));

     // ---修改数据---
     // 1.在原来基础上追加
     // jedis.append(“name1”, “b1”);
     // System.out.println(jedis.get(“name1”));

     // 2.直接覆盖
     jedis.set(key, value);
     // System.out.println(jedis.get(“name1”));
```

```
        // 删除key对应记录
        // jedis.del("name1");
        // System.out.println(jedis.get("name1"));
    }

    public String testReadData(String key) {
        return jedis.get(key);
    }

    public static void main(String[] args) {
        RDBtester tester1 = new RDBtester();
        int I = 0;
        while (I < 100) {
         I = I + 1;
            tester1.setUp();
            tester1.testWriteData("name" + I, "test" + i);
        }
    }
}
```

b) 实例化读数据线程，实现生成 100 个键值对：

```
package SE.lin;

import java.io.IOException;

public class BlockTester {
    int changeID = 0;

    public static void main(String args[]) {
        System.out.println("***SAVE测试开始***");
        // 创建thread1,写数据
        ThreadUseExtends thread1 = new ThreadUseExtends(0,
10);
     Thread1start();

        System.out.println("主线程将挂起1秒");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("回到主线程");
```

```
        if (thread1.isAlive()) {
            // 如果thread1还存在则杀掉他
            thread1.stop();
            System.out.println("thread1杀掉，写数据完成");
        } else {
            System.out.println("主线程没发现thread1，则它已顺
利结束");
        }
    }// main
}// BlockTester

class ThreadUseExtends extends Thread {
    int changeID;
    int step;

    ThreadUseExtends(int 16etup16d, int stp) {
        this.changeID = 16etup16d;
        this.step = stp;
    } // 构造函数

    public void run() {
        System.out.println("\t thread1:我是写数据线程");
        System.out.println("\t thread1:我将不断写数据");
        RDBtester tester1 = new RDBtester();
        while (changeID < 100) {
            changeID = changeID + 1;
            tester1.setUp();
            tester1.testWriteData("name" + changeID, "test"
+ changeID);
            try {
                sleep(step);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

2) 同步回写 SAVE 功能阻塞测试

一个线程调用 save 命令，一个线程循环调用 set 命令，二者同时进行 10 秒，则若 save 命令执行时间合适，set 命令循环调用过程中很可能被阻塞，此时可观察服务器报错信息。

```
Package SE.lin;
```

```
import java.io.IOException;

import redis.clients.jedis.exceptions.JedisException;

public class SaveBlocker {

  public static void main(String args[]) {
      System.out.println("***测试开始***");
      // 创建thread1,写数据
      SaveThread thread1 = new SaveThread(0);
      // 创建Thread2，读数据
      SaveThread thread2 = new SaveThread(1);
      // Thread thread2 = new Thread(new
ThreadUseRunnable(), "SecondThread");
      // 同时启动
      thread2.start();
      thread1.start();

      System.out.println("主线程将挂起10秒");
      try {
          Thread.sleep(10000);
      } catch (InterruptedException e) {
          return;
      }
      System.out.println("回到主线程");

      if (thread1.isAlive()) {
          // 如果thread1还存在则杀掉他
          thread1.stop();
          System.out.println("thread1杀掉，写数据完成");
      } else {
          System.out.println("主线程没发现thread1，则它已顺
利结束");
      }

      if (thread2.isAlive()) {
          thread2.stop();// 如果thread2还存在则杀掉他
          System.out.println("thread2杀掉，读数据完成");
      } else
          System.out.println("主线程没发现thread2,则它已顺
利结束!");

      System.out.println("程序结束按任意键继续!");
      try {
```

```
            System.in.read();
        } catch (IOException e) {
            System.out.println(e.toString());
        }

    }

}

/***
 * 不断写数据，以触发SAVE同步回写
 *
 * @author linlu
 *
 */
class SaveThread extends Thread {
    int func;
    RDBtester tester1 = new RDBtester();

    SaveThread(int funcid) {
        this.func = funcid;
        this.tester1.setUp();
    } // 构造函数

    public void run() {

        int I = 0;
        if (this.func == 0) {
            System.out.println("\t thread1:我是save线程");
            tester1.saveData();
        } else {
            System.out.println("\t thread1:我是发送请求线
程");
            while (true) {
                I = I + 1;
                try {
                    tester1.testWriteData("name1",
"blocktest" + i);
                } catch (JedisException e) {
                    System.out.println(e.getMessage());

                }
            }
        }
```

```
    }
```

4.1.4 测试结果及分析

1) 同步回写 SAVE 功能正常测试

jedis 测试程序修改 100 条键值对，在客户端手动输入 SAVE 命令，服务器保存成功信息如图所示：

```
[960636] 17 May 07:23:10.824 # Server started, Redis version 2.8.19
[960636] 17 May 07:23:10.825 * DB loaded from disk: 0.000 seconds
[960636] 17 May 07:23:10.826 * The server is now ready to accept connections on
port 6379
[960636] 17 May 07:23:45.257 * DB saved on disk
```

客户端得到成功反馈：

```
127.0.0.1:6379> save
OK
127.0.0.1:6379>
```

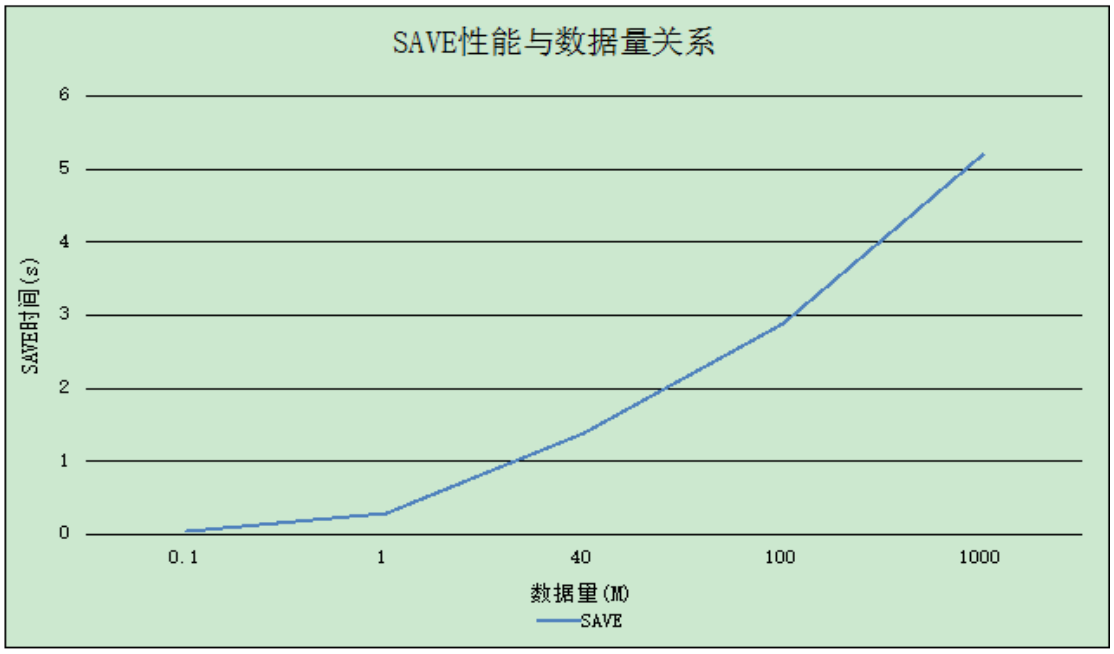
开启客户端，随机读取添加的键，正是新修改的值，如图所示:

```
127.0.0.1:6379> get name30
"test30"
127.0.0.1:6379> get name100
"test100"
```

由以上结果可知，同步回写 SAVE 方式保存的功能已成功实现。

2) 同步回写 SAVE 功能阻塞测试

为考察 save 的执行时间，以便选择合适的快照数据大小，首先进行 save 的性能测试：



由图可知，2 秒左右时间，对于测试命令的并发比较合适，因此我们选择 40M 左右数据进行修改和保存。

当 save 命令执行时，由于主进程被阻塞，服务器将对客户端的其他请求

报错，因此当调用”set name1 blocktestID”时，客户端收到错误信息：

```
***测试开始***
主线程将挂起10秒
        thread1:我是发送请求线程
        thread1:我是save线程
java.net.SocketTimeoutException: Read timed out
Exception in thread "Thread-0" redis.clients.jedis.exceptions.JedisConnectionException: java.net.SocketTimeoutException
        at redis.clients.jedis.Protocol.process(Protocol.java:74)
        at redis.clients.jedis.Protocol.read(Protocol.java:122)
        at redis.clients.jedis.Connection.getStatusCodeReply(Connection.java:152)
        at redis.clients.jedis.BinaryJedis.save(BinaryJedis.java:2615)
        at SE.lin.RDBtester.saveData(RDBtester.java:51)
        at SE.lin.SaveThread.run(SaveBlocker.java:73)
Caused by: java.net.SocketTimeoutException: Read timed out
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.read(SocketInputStream.java:150)
        at java.net.SocketInputStream.read(SocketInputStream.java:121)
        at java.net.SocketInputStream.read(SocketInputStream.java:107)
        at redis.clients.util.RedisInputStream.fill(RedisInputStream.java:110)
        at redis.clients.util.RedisInputStream.readByte(RedisInputStream.java:46)
        at redis.clients.jedis.Protocol.process(Protocol.java:59)
        ... 5 more
回到主线程
主线程没发现thread1，则它已顺利结束
thread2杀掉，读数据完成
程序结束按任意键继续！
```

当首先调用 save 命令的线程，并使发送其他请求（set 命令）的线程休眠 5 秒使 save 命令先执行完，此时服务器可正常处理其他请求：

```
***测试开始***
主线程将挂起10秒
        thread1:我是save线程
save cost:1913ms
        thread1:我是发送请求线程
回到主线程
主线程没发现thread1，则它已顺利结束
thread2杀掉，读数据完成
程序结束按任意键继续！
```

```
127.0.0.1:6379> get name1
"blocktest101055"
```

4.2 异步回写 BGSAVE 测试

4.2.1 测试目标

异步回写 BGSAVE 与 SAVE 的不同点在于，主进程会 fork 出一个子进程，子进程负责调用 rdbSave()函数，并在保存完成后向主进程发送信号，通知保存已完成。因为 rdbSave()执行期间，是被子进程调用的，所以 Redis 服务器在 BGSAVE 期间仍可以继续处理客户端的请求。

本部分测试在 window 环境下:

1) 当 Redis 开启 rdb 方式，配置每 m 秒发生 n 次变更进行 rdb 文件保存；
2) 验证其主进程被阻塞时，服务器仍能接受客户端命令，但是拒绝处理同时调用的 SAVE 或 BGSAVE 命令以避免竞争。

4.2.2 测试用例

4.2.2.1 异步回写 BGSAVE 功能正常测试

Test Case Specification		
Name	异步回写 BGSAVE 功能正常测试	
Brief Description	测试能否正确读取用户配置命令，服务器进行异步回写 BGSAVE 操作	
Precondition	Redis 在 windows 下正确安装	
Tester	测试员	
Dependency	None	
Test Setup	Name	配置 redis.conf 文件
	Description	配置 redis.conf 文件，以选择 RDB 持久化方式，并调用 SAVE 命令
Basic Flow (Test Setup)	Steps	
	1	测试员打开 redis.conf 文件
	2	测试员写入命令”**save 2 1**”，设置快照保存的策略为每 2 秒发生 1 次变更则进行保存，默认为 BGSAVE 方式
	3	写入命令”dbfilename dump.rdb”，设置快照保存文件的文件名
	4	写入命令”dir ./”，设置备份文件放置路径
	5	写入命令”appendonly no”，选择非 AOF 方式，也即 RDB 方式
	Postcondition (Test Oracle)	配置完成，准备启动 Redis 进行快照 BGSAVE 测试
Basic Flow (Test Sequence)	Steps	
	1	打开 windows 下的 cmd 运行窗口
	2	输入”cd PATH-TO-REDIS”，切换到 Redis 文件夹下
	3	输入”redis-server.exe redis.conf”，启动 Redis
	4	输入”redis-cli.exe”，打开一个窗口运行客户端 A
	5	创建 Eclipse 项目，引入 jedis 客户端包
	6	编写测试程序 A-test，实现每秒修改一次数据，运行 30 秒
	7	测试程序 A-test 运行完成，在客户端 A 输入命令”get xxx”，获取被修改键的最新值
	Postcondition (Test Oracle)	最新的保存值恰是测试程序逻辑所得，异步保存 BGSAVE 的每 2 秒发生 1 次变更则保存的功能被测试

4.1.2.2 异步回写 BGSAVE 功能异常测试

Test Case Specification	
Name	异步回写 BGSAVE 功能异常测试

Brief Description	测试进行异步回写操作时，其他客户端发出请求，服务器能否正常处理	
Precondition	Redis 在 windows 下可正常进行 RDB 持久化	
Tester	测试员	
Dependency	None	
Test Setup	Name	配置 redis.conf 文件
	Description	配置 redis.conf 文件，以选择 RDB 持久化方式，并调用 BGSAVE 命令
Basic Flow (Test Setup)	Steps	
	1	测试员打开 redis.conf 文件
	2	测试员写入命令”**save “” ”**，设置手动快照保存
	3	写入命令”dbfilename dump.rdb”，设置快照保存文件的文件名
	4	写入命令”dir ./”，设置备份文件放置路径
	5	写入命令”appendonly no”，选择非 AOF 方式，也即 RDB 方式
	Postcondition (Test Oracle)	配置完成，准备启动 Redis 进行快照 BGSAVE 测试
Basic Flow (Test Sequence)	Steps	
	1	打开 windows 下的 cmd 运行窗口
	2	输入”cd PATH-TO-REDIS”，切换到 Redis 文件夹下
	3	输入”redis-server.exe redis.conf”，启动 Redis
	4	创建 Eclipse 项目，引入 jedis 客户端包
	5	编写测试程序 A-test，实现 10000000L 数据的修改（大概 40M）
	6	编写测试程序 B-test，同时运行两个线程，分别调用 save 命令和其他命令
	7	编写测试程序 C-test，同时运行两个线程，分别调用两个 save 命令
	Postcondition (Test Oracle)	测试程序 B-test 正常执行，而 C-test 报错，异步回写 BGSAVE 过程中服务器处理其他客户端请求的功能被测试

4.2.3 测试代码

1) 异步回写 BGSAVE 功能正常测试
   a) 创建 Jedis 类，连接 redis 服务器，实现数据的读写方法：
   代码与 4.1.3(1)(a)相同。
   b) 实例化读数据线程，实现每秒修改一次数据；主线程中运行 30 秒：

```
package SE.lin;
import java.io.IOException;
public class BlockTester {
```

```
  public static void main(String args[]) {
      // 创建thread1,写数据
      ThreadUseExtends thread1 = new ThreadUseExtends();
      thread1.start();
      System.out.println("主线程将挂起30秒");
      try {
          Thread.sleep(30000);
      } catch (InterruptedException e) {
          return;
      }
      System.out.println("回到主线程");

      if (thread1.isAlive()) {
          // 如果thread1还存在则杀掉他
          thread1.stop();
          System.out.println("thread1杀掉，写数据完成");
      } else {
          System.out.println("主线程没发现thread1，则它已顺
利结束");
      }
  }// main
}// BlockTester

/***
 * 不断写数据，以触发快照保存
 *
 * @author linlu
 *
 */
class ThreadUseExtends extends Thread {
  ThreadUseExtends() {
  } // 构造函数
  public void run() {
      System.out.println("\t thread1:我是写数据线程");
      System.out.println("\t thread1:我将不断写数据");
      RDBtester tester1 = new RDBtester();
      int i = 0;
      while (true) {
          // 每1秒写一次数据
          i = i + 1;
          tester1.setUp();
          tester1.testWriteData("test" + i);
          try {
              sleep(1000);
```

```
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

2) 异步回写 BGSAVE 功能异常测试

```
package SE.lin;

import java.io.IOException;

import redis.clients.jedis.exceptions.JedisException;

public class SaveBlocker {

    public static void main(String args[]) {
        System.out.println("***测试开始***");
        // 创建thread1,写数据
        SaveThread thread1 = new SaveThread(0);
        // 创建Thread2，读数据
        SaveThread thread2 = new SaveThread(0);
        // Thread thread2 = new Thread(new
ThreadUseRunnable(), "SecondThread");
        // 同时启动
        thread2.start();
        thread1.start();

        System.out.println("主线程将挂起10秒");
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            return;
        }
        System.out.println("回到主线程");

        if (thread1.isAlive()) {
            // 如果thread1还存在则杀掉他
            thread1.stop();
            System.out.println("thread1杀掉，保存数据完成");
        } else {
            System.out.println("主线程没发现thread1，则它已顺
利结束");
```

```
        }

        if (thread2.isAlive()) {
            thread2.stop();// 如果thread2还存在则杀掉他
            System.out.println("thread2杀掉，保存数据完成");
        } else
            System.out.println("主线程没发现thread2,则它已顺
利结束!");

        System.out.println("程序结束按任意键继续!");
        try {
            System.in.read();
        } catch (IOException e) {
            System.out.println(e.toString());
        }

    }

}

/***
 * 向服务器发送save、或set请求
 *
 * @author linlu
 *
 */
class SaveThread extends Thread {
    int func;
    RDBtester tester1 = new RDBtester();

    SaveThread(int funcid) {
        this.func = funcid;
        this.tester1.setUp();
    } // 构造函数

    public void run() {

        int i = 0;
        int flag = 0;
        if (this.func == 0) {
            System.out.println("\t thread1:我是save线程");
            tester1.saveData();
        } else {
            // try {
```

```
                // sleep(5000);
                // } catch (InterruptedException e1) {
                // // TODO Auto-generated catch block
                // e1.printStackTrace();
                // }

                System.out.println("\t thread1:我是发送请求线程
");
                System.out.println("\t *服务器执行set命令之前：
name1的值为"
                        + tester1.testReadData("name1"));
                while (true) {
                    i = i + 1;
                    try {
                        tester1.testWriteData("name1",
"blocktest" + i);
                        if (flag == 0) {
                            System.out.println("\t *服务器执行set
命令之后：name1的值为"
                                    +
tester1.testReadData("name1"));
                            flag = 1;
                        }
                    } catch (JedisException e) {
                        System.out.println(e.getMessage());

                    }
                }
            }
        }
    }
```

4.2.4 测试结果及分析

1) 异步回写 BGSAVE 功能正常测试

在 jedis 测试程序一共运行 30 秒，每秒修改一次数据，redis 服务器端每两秒检查符合同步回写条件，则进行同步回写，服务器保存成功信息如图所示：

```
[921828] 17 May 00:57:51.480 # Server started, Redis version 2.8.19
[921828] 17 May 00:57:51.481 * DB loaded from disk: 0.000 seconds
[921828] 17 May 00:57:51.481 * The server is now ready to accept connections on
port 6379
[921828] 17 May 00:57:56.751 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:57:56.956 # fork operation complete
[921828] 17 May 00:57:56.966 * Background saving terminated with success
[921828] 17 May 00:57:59.040 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:57:59.344 # fork operation complete
[921828] 17 May 00:57:59.360 * Background saving terminated with success
[921828] 17 May 00:58:02.031 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:02.398 # fork operation complete
[921828] 17 May 00:58:02.413 * Background saving terminated with success
[921828] 17 May 00:58:05.023 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:05.400 # fork operation complete
[921828] 17 May 00:58:05.621 * Background saving terminated with success
[921828] 17 May 00:58:08.049 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:08.538 # fork operation complete
[921828] 17 May 00:58:08.549 * Background saving terminated with success
[921828] 17 May 00:58:11.067 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:11.432 # fork operation complete
[921828] 17 May 00:58:11.442 * Background saving terminated with success
[921828] 17 May 00:58:14.044 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:14.251 # fork operation complete
[921828] 17 May 00:58:14.261 * Background saving terminated with success
[921828] 17 May 00:58:17.061 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:17.520 # fork operation complete
[921828] 17 May 00:58:17.530 * Background saving terminated with success
[921828] 17 May 00:58:20.031 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:20.289 # fork operation complete
[921828] 17 May 00:58:20.299 * Background saving terminated with success
[921828] 17 May 00:58:23.000 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:23.192 # fork operation complete
[921828] 17 May 00:58:23.202 * Background saving terminated with success
[921828] 17 May 00:58:26.002 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:26.225 # fork operation complete
[921828] 17 May 00:58:26.237 * Background saving terminated with success
[921828] 17 May 00:58:29.037 * 1 changes in 2 seconds. Saving...
[921828] 17 May 00:58:29.325 # fork operation complete
[921828] 17 May 00:58:29.335 * Background saving terminated with success
```

开启客户端，读取被修改的键”name1”，正是最新修改的值，如图所示:

```
127.0.0.1:6379> get name1
"test30"
127.0.0.1:6379>
```

由以上结果可知，同步回写 SAVE 方式，可实现每 m 秒修改 n 次保存的功能。

2) 异步回写 BGSAVE 功能异常测试

对于非 save 和 bgsave 的普通客户端请求（如 set 命令），服务器能够正常响应：

```
***测试开始***  SaveBlocker [Java Application] C:\Program Files\Java\jdk1.7.0_21\bin\javaw.exe (2015年5月17日 上午9:34:01)
主线程将挂起10秒
        thread1:我是发送请求线程
        thread1:我是save线程
save cost:81ms
        *服务器执行set命令之前：name1的值为ghkhgfffffffffffffffsssssssseeeeeeeeeeeeeeeeee
        *服务器执行set命令之后：name1的值为blocktest1
回到主线程
主线程没发现thread1，则它已顺利结束
thread2杀掉，读数据完成
程序结束按任意键继续！
```

对于其他 save 命令，服务器报错：

```
***测试开始***
主线程将挂起10秒
        thread1:我是save线程
        thread1:我是save线程
save cost:388ms
Exception in thread "Thread-2" redis.clients.jedis.exceptions.JedisDataException: ERR Background save already in progress
        at redis.clients.jedis.Protocol.processError(Protocol.java:54)
        at redis.clients.jedis.Protocol.process(Protocol.java:61)
        at redis.clients.jedis.Protocol.read(Protocol.java:122)
        at redis.clients.jedis.Connection.getStatusCodeReply(Connection.java:152)
        at redis.clients.jedis.BinaryJedis.bgsave(BinaryJedis.java:2630)
        at SE.lin.RDBtester.saveData(RDBtester.java:51)
        at SE.lin.SaveThread.run(SaveBlocker.java:74)
回到主线程
主线程没发现thread1，则它已顺利结束
主线程没发现thread2,则它已顺利结束！
程序结束按任意键继续！
```

4.3 载入数据测试

4.3.1 测试目标

当 Redis 服务器启动时，执行载入函数 rdbLoad，读取备份.rdb 文件，并将文件中的数据库数据载入到内存中。

本部分测试当修改数据键值，并手动调用 save 命令将更改备份后，服务器关闭又重启后，是否能正确载入备份的数据到内存。

4.3.2 测试用例

Test Case Specification		
Name	载入数据测试	
Brief Description	测试 Redis 服务器是否能正确从.rdb 备份文件中恢复数据到内存中	
Precondition	Redis 服务器已启动，正在运行并未断电，已自动进行快照保存	
Tester	测试员	
Dependency	None	
Test Setup	Name	修改数据库值
	Description	修改某个数据库的值，以验证修改能被保存和恢复
Basic Flow (Test Setup)	Steps	
	1	测试员在运行窗口输入”redis-cli.exe”，打开一个窗口运行客户端 A
	2	输入”set name1 10”和”set name2 20”，设为测试键值
	3	输入”del name1”，删除测试键值对 name1

	4	输入”appen name2 100”，修改测试键值对 name2
	3	输入”save”，手动进行快照保存
	4	输入”shutdown”，停止 Redis 服务器
	Postcondition (Test Oracle)	数据库的一对键值被删除
Basic Flow (Test Sequence)	Steps	
	1	输入“redis-server.exe redis.conf”，重启服务器
	2	输入”redis-cli.exe”，再启动客户端
	3	输入”get name1”和”get name2”，获取之前被修改的数据
	Postcondition (Test Oracle)	name1 为 null，name2 为 20100，说明服务器重启后，重新从备份文件载入数据到内存成功

4.3.3 测试代码

本测试无需测试代码。

4.3.4 测试结果及分析

首先初始化键值，并修改之（name1 被删除，name2 被新值覆盖），并关闭服务器：

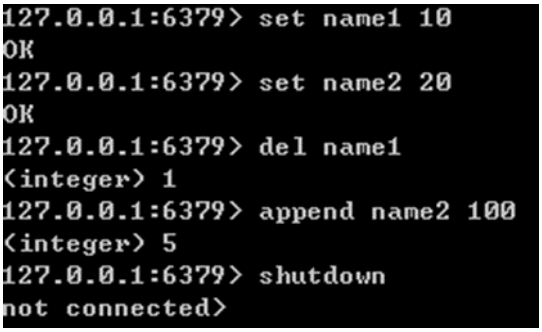


图 1 客户端修改、保存数据，关闭服务器

```
[928532] 17 May 01:55:58.444 # Server started, Redis version 2.8.19
[928532] 17 May 01:55:58.445 * DB loaded from disk: 0.000 seconds
[928532] 17 May 01:55:58.445 * The server is now ready to accept connections on
port 6379
[928532] 17 May 01:59:59.924 * 1 changes in 2 seconds. Saving...
[928532] 17 May 02:00:00.183 # fork operation complete
[928532] 17 May 02:00:00.194 * Background saving terminated with success
[928532] 17 May 02:00:05.995 * 1 changes in 2 seconds. Saving...
[928532] 17 May 02:00:06.666 # fork operation complete
[928532] 17 May 02:00:06.681 * Background saving terminated with success
[928532] 17 May 02:03:24.564 * 1 changes in 2 seconds. Saving...
[928532] 17 May 02:03:24.729 # fork operation complete
[928532] 17 May 02:03:24.738 * Background saving terminated with success
[928532] 17 May 02:03:32.839 * 1 changes in 2 seconds. Saving...
[928532] 17 May 02:03:33.078 # fork operation complete
[928532] 17 May 02:03:33.089 * Background saving terminated with success
[928532] 17 May 02:04:56.471 # User requested shutdown...
[928532] 17 May 02:04:56.471 * Saving the final RDB snapshot before exiting.
[928532] 17 May 02:04:56.565 * DB saved on disk
[928532] 17 May 02:04:56.565 # Redis is now ready to exit, bye bye...
```

图 2 服务器在此期间的响应

重启服务器之后，再读取该键值，为修改之后的值（因为是删除，因此返回为 null），说明数据在服务器重启时已正常载入：

```
not connected> get name1
(nil)
127.0.0.1:6379> get name2
"20100"
```

5 AOF 持久化模块

5.1 命令同步测试

5.1.1 测试目标

本测试用例是用来测试 Redis 的命令同步功能。以 AOF 持久化方式打开 Redis 服务器，在 Redis 客户端输入数据库修改写入命令，通过查看 AOF 文件，命令是否正确的以 AOF 网络通讯协议的格式保存下来,来判断 Redis 是否能够实现命令同步功能。

5.1.2 测试用例

Test Case Specification	
Name	命令同步测试
Brief Description	测试 Redis 将所有对数据库进行过写入的命令（及其参数）记录到 AOF 文件中
Precondition	Redis 服务器与客户端正常运行
Tester	Tester

Dependency	None	
Test Setup	Name	准备命令并打开 Redis 服务器和客户端
	Description	准备好向 Redis 写入的命令并以 aof 方式打开 Redis 服务器，打开 Redis 客户端等待命令写入
Basic Flow (Test Setup)	Steps	
	1	以管理员的方式打开 windows 下的 cmd 运行窗口
	2	切换到 Redis/src/bin 文件夹下
	3	输入”redis-server.exe --appendonly yes”启动命令参数，启动 Redis 服务器并且开启 aof 功能
	4	输入”redis-cli.exe”，打开一个窗口运行客户端
	5	准备好写入数据库的命令
	Postcondition (Test Oracle)	客户端命令行已经打开，处于等待数据库写入命令状态
Basic Flow (Test Sequence)	Steps	
	1	在客户端命令行里输入已经准备好的数据库写入命令
	2	切换到 Redis/src/bin 文件夹下
	3	输入”cat appendonly.aof” 查看 AOF 文件
	Postcondition (Test Oracle)	命令输入完成后，查看 AOF 文件命令被以 aof 网络通讯协议的格式保存下来，测试通过

5.1.3 测试代码

切换到 bin 文件夹下：

```
C:\Windows\system32>E:

E:\>cd downloads\redis-2.8\bin

E:\downloads\redis-2.8\bin>
```

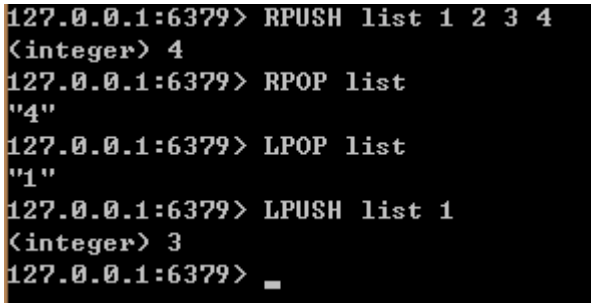
启动 redis 服务器：

```
E:\downloads\redis-2.8\bin>redis-server.exe --appendonly yes
```

启动客户端：

```
E:\downloads\redis-2.8\bin>redis-cli.exe
```

向客户端输入如下指令：



5.1.4 测试结果及分析

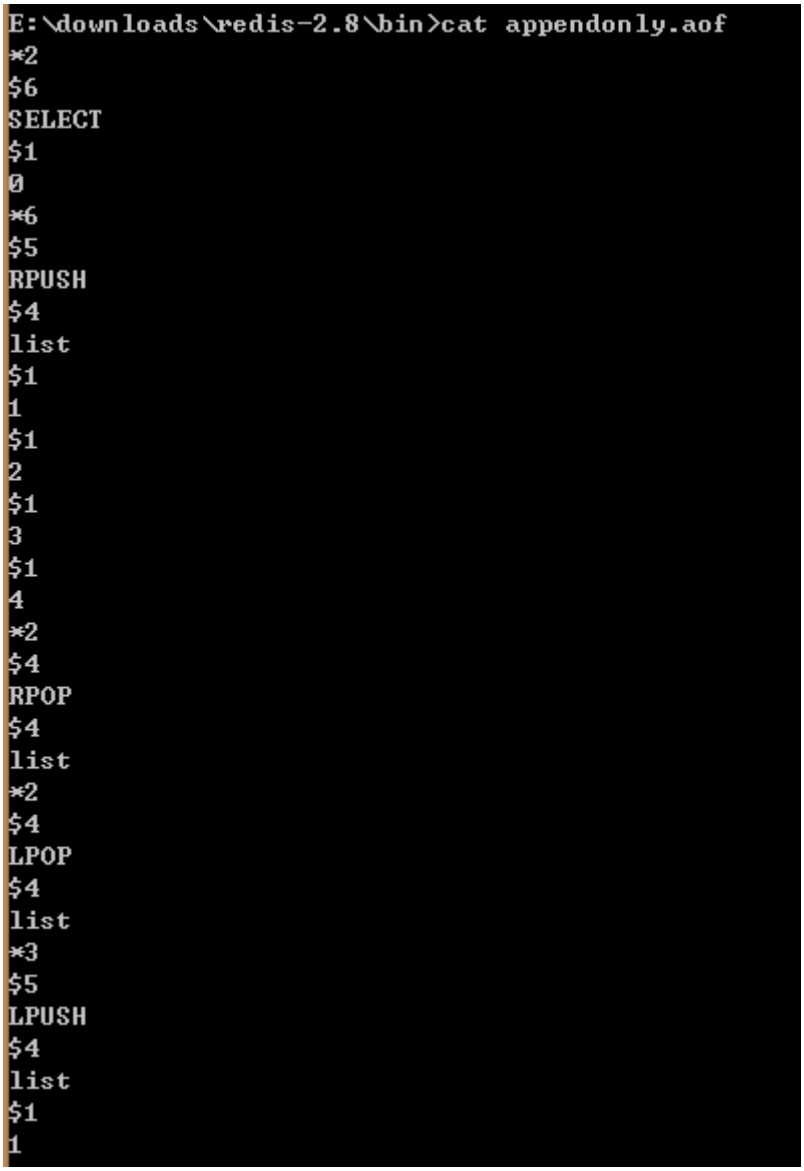
期望结果：

在 AOF 文件中向 Redis 输入的命令正确的以 AOF 网络通讯协议的格式保存下来

输入查看 AOF 文件的指令：

实际结果及分析：

在客户端输入的指令被保存为 AOF 网络通讯协议格式：



5.2 AOF 重写测试

5.2.1 测试目标

本测试用例是用来测试 Redis 的 AOF 重写功能，通过对比执行过 AOF 文件和重写后的 AOF 文件的数据库状态来判定 Redis 是否实现 AOF 重写功能。

5.2.2 测试用例

Test Case Specification	
Name	AOF 重写测试
Brief Description	测试根据数据库键的类型，使用适当的写入命令来重现键的当前值，以节省 AOF 文件的存储空间的功能

Precondition	Redis 服务器与客户端正常运行	
Tester	Tester	
Dependency	None	
Test Setup	Name	打开服务器和客户端
	Description	打开服务器和客户端命令行
Basic Flow (Test Setup)	Steps	
	1	以管理员的方式打开 windows 下的 cmd 运行窗口
	2	切换到 Redis/src/bin 文件夹下
	3	输入”redis-server.exe --appendonly yes”启动命令参数，启动 Redis 服务器并且开启 aof 功能
	4	输入”redis-cli.exe”，打开一个窗口运行客户端
	5	准备一段要写入到数据库中的命令
	Postcondition (Test Oracle)	命令行被打开，等待输入命令
Basic Flow (Test Sequence)	Steps	
	1	打开客户端，输入原本准备的数据库命令
	2	切换到 Redis/src/bin 文件夹下
	3	输入”cat appendonly.aof” 查看 AOF 文件
	4	在客户端输入”BGREWRITEAOF”指令进行重写
	5	输入”cat appendonly.aof” 查看 AOF 文件
	Postcondition (Test Oracle)	若后来的 AOF 文件为原 AOF 的简要缩写但是对 Redis 输入的指令意义又不变化，即为测试通过。

5.2.3 测试代码

切换到 bin 文件夹下：

```
C:\Windows\system32>E:

E:\>cd downloads\redis-2.8\bin

E:\downloads\redis-2.8\bin>
```

启动 redis 服务器：

```
E:\downloads\redis-2.8\bin>redis-server.exe --appendonly yes
```

启动客户端：

```
E:\downloads\redis-2.8\bin>redis-cli.exe
```

向客户端输入如下指令：

```
127.0.0.1:6379> RPUSH list 1 2 3 4
(integer) 4
127.0.0.1:6379> RPOP list
"4"
127.0.0.1:6379> LPOP list
"1"
127.0.0.1:6379> LPUSH list 1
(integer) 3
127.0.0.1:6379>
```

执行重写命令：

```
127.0.0.1:6379> BGREWRITEAOF
Background append only file rewriting started
127.0.0.1:6379>
```

5.2.4 测试结果及分析

期望结果：

AOF 能实现 AOF 重写对 AOF 文件的大小进行减负，并且 AOF 重写前与 AOF 重写后的对于 Redis 数据库改变的状态一致。

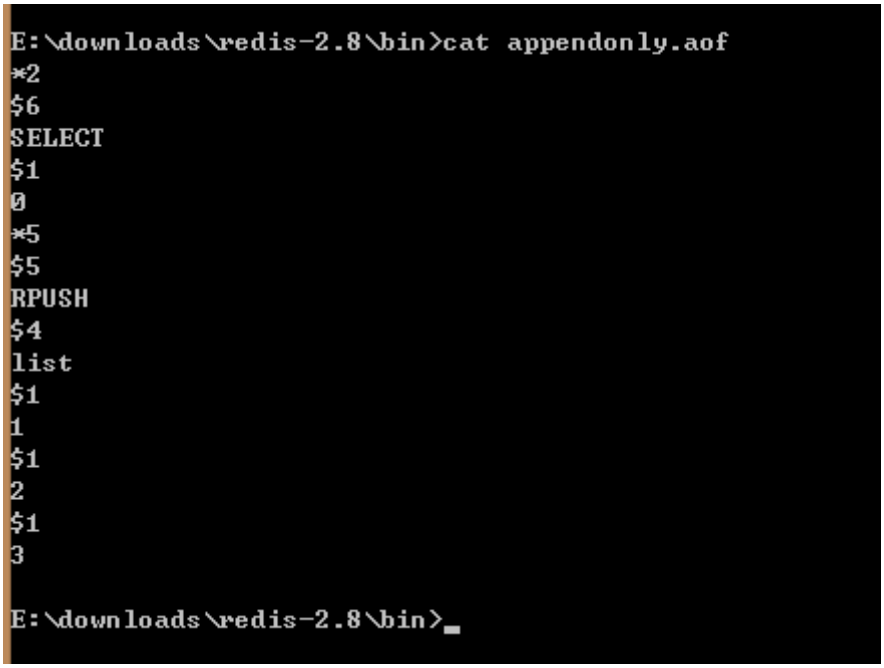
实际结果及分析：

AOF 重写不仅实现了 AOF 文件的减负，而且 AOF 文件对 Redis 指令对数据库的实际状态并没有改变。

重写前：

```
E:\downloads\redis-2.8\bin>cat appendonly.aof
*2
$6
SELECT
$1
0
*6
$5
RPUSH
$4
list
$1
1
$1
2
$1
3
$1
4
*2
$4
RPOP
$4
list
*2
$4
LPOP
$4
list
*3
$5
LPUSH
$4
list
$1
1
```

重写后：



5.3 AOF 文件还原测试

5.3.1 测试目标

本测试用例是用来测试 Redis 的 AOF 文件还原功能，通过将向数据库写入的命令和从 AOF 文件还原出的命令对比，如果相等可以测试出数据还原功能正常。

5.3.2 测试用例

Test Case Specification		
Name	AOF 文件还原测试	
Brief Description	测试读取 AOF 文件，并且将 AOF 网络通讯协议的格式还原为数据库写入命令，再执行该命令还原数据库的状态	
Precondition	Redis 服务器与客户端正常运行	
Tester	Tester	
Dependency	None	
Test Setup	Name	打开服务器和客户端
	Description	打开服务器和客户端命令行
Basic Flow (Test Setup)	Steps	
	1	以管理员的方式打开 windows 下的 cmd 运行窗口
	2	切换到 Redis/src/bin 文件夹下
	3	输入”redis-server.exe --appendonly yes”启动命令参数，启动 Redis 服务器并且开启 aof 功能
	4	输入”redis-cli.exe”，打开一个窗口运行客户端
	Postcondition	命令行被打开，等待输入命令

	(Test Oracle)	
Basic Flow (Test Sequence)	Steps	
	1	准备一段 Redis 写入命令自己单独记录
	2	打开客户端，输入数据库命令
	3	读取 AOF 文件
	4	根据还原函数 loadAppendOnlyFile 还原 AOF 文件为数据库命令
	Postcondition (Test Oracle)	对比开始写入的命令和从 AOF 文件还原出来的数据库命令，若两者相等则测试通过

5.3.3 测试代码

切换到 bin 文件夹下：

```
C:\Windows\system32>E:

E:\>cd downloads\redis-2.8\bin

E:\downloads\redis-2.8\bin>
```

启动 redis 服务器：

```
E:\downloads\redis-2.8\bin>redis-server.exe --appendonly yes
```

启动客户端：

```
E:\downloads\redis-2.8\bin>redis-cli.exe
```

向客户端输入如下指令：

```
127.0.0.1:6379> RPUSH list 1 2 3 4
(integer) 4
127.0.0.1:6379> RPOP list
"4"
127.0.0.1:6379> LPOP list
"1"
127.0.0.1:6379> LPUSH list 1
(integer) 3
127.0.0.1:6379>
```

使用还原函数代码：

```
robj ** loadAppendOnlyFile(char *filename) {


    // 打开 AOF 文件
    FILE *fp = fopen(filename,"r");

    struct redis_stat sb;
    int old_aof_state = server.aof_state;
    long loops = 0;

    // 检查文件的正确性
    if (fp && redis_fstat(fileno(fp),&sb) != -1 && sb.st_size == 0) {
        server.aof_current_size = 0;
        fclose(fp);
        return REDIS_ERR;
    }

    // 检查文件是否正常打开
    if (fp == NULL) {
        redisLog(REDIS_WARNING,"Fatal error: can't open the append log file for reading: %s",strerror(errno));
        exit(1);
    }

    /* Temporarily disable AOF, to prevent EXEC from feeding a MULTI
     * to the same file we're about to read.
     *
     * 暂时性地关闭 AOF ，防止在执行 MULTI 时，
     * EXEC 命令被传播到正在打开的 AOF 文件中。
     */
    server.aof_state = REDIS_AOF_OFF;

    fakeClient = createFakeClient();

    // 设置服务器的状态为：正在载入
    // startLoading 定义于 rdb.c
    startLoading(fp);

    while(1) {
```

```
        int argc, j;
        unsigned long len;
        robj **argv;
        char buf[128];
        sds argsds;
        struct redisCommand *cmd;

        /* Serve the clients from time to time
         *
         * 间隔性地处理客户端发送来的请求
         * 因为服务器正处于载入状态，所以能正常执行的只有 PUBSUB 等模块
         */
        if (!(loops++ % 1000)) {
            loadingProgress(ftello(fp));
            processEventsWhileBlocked();
        }

        // 读入文件内容到缓存
        if (fgets(buf,sizeof(buf),fp) == NULL) {
            if (feof(fp))
                // 文件已经读完，跳出
                break;
            else
                goto readerr;
        }

        // 确认协议格式，比如 *3\r\n
        if (buf[0] != '*') goto fmterr;

        // 取出命令参数，比如 *3\r\n 中的 3
        argc = atoi(buf+1);

        // 至少要有一个参数（被调用的命令）
        if (argc < 1) goto fmterr;

        // 从文本中创建字符串对象：包括命令，以及命令参数
        // 例如 $3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n
        // 将创建三个包含以下内容的字符串对象：
        // SET 、 KEY 、 VALUE
        argv = zmalloc(sizeof(robj*)*argc);
        for (j = 0; j < argc; j++) {
            if (fgets(buf,sizeof(buf),fp) == NULL) goto readerr;
```

```
        if (buf[0] != '$') goto fmterr;

        // 读取参数值的长度
        len = strtol(buf+1,NULL,10);
        // 读取参数值
        argsds = sdsnewlen(NULL,len);
        if (len && fread(argsds,len,1,fp) == 0) goto fmterr;
        // 为参数创建对象
        argv[j] = createObject(REDIS_STRING,argsds);
        cin>>argv[j]

        if (fread(buf,2,1,fp) == 0) goto fmterr; /* discard CRLF */
    }

}
```

5.3.4 测试结果及分析

期望结果：

通过将向数据库写入的命令和从 AOF 文件还原出的命令对比，两者应该相等即可：

执行此函数。

```
int main()
{
    loadAppendOnlyFile("appendonly.aof");


}
```

实际结果及分析：

向 Redis 输入的命令与从 aof 文件通过 loadAppendOnlyFile 函数还原出的指令相同。

```
127.0.0.1:6379> RPUSH list 1 2 3 4
(integer) 4
127.0.0.1:6379> RPOP list
"4"
127.0.0.1:6379> LPOP list
"1"
127.0.0.1:6379> LPUSH list 1
(integer) 3
127.0.0.1:6379>
```

```
RPUSH list 1 2 3 4
RPOP list
LPOP list
LPUSH list 1
Press any key to continue
```

6 客户端模块

6.1 命令请求处理测试

6.1.1 测试目标

本测试用例是用来测试客户端对用户输入命令的处理功能，通过在 redis 客户端中输入一系列正确的命令请求和异常的命令请求，分别查看 redis 客户端保存的协议格式，判断客户端是否能将各种命令请求正确转换成协议格式。

6.1.2 测试用例

Test Case Specification		
Name	命令请求处理测试	
Brief Description	测试客户端收到来自用户的命令请求，并将其转换为协议格式	
Precondition	Redis 服务器正在运行	
Tester	None	
Dependency	None	
Test Setup	Name	打开命令行
	Description	打开命令行界面以输入命令请求
Basic Flow (Test Setup)	Steps	
	1	打开操作系统的命令行界面
	2	打开命令行，进入 redis 源码路径
	Postcondition (Test Oracle)	命令行已经打开，处于等待启动客户端状态
Basic Flow (Test Sequence)	Steps	
	1	测试员进入命令行
	2	测试员输入命令”cd redis/src” 进入 redis 目录
	3	输入”./redis-cli” 启动 redis 客户端
	4	输入“set key value ”发送命令请求
	5	输入“get key” 发送命令请求
	6	输入一系列错误命令请求
	7	测试员查看客户端保存的命令协议格式
	Postcondition (Test Oracle)	命令输入完成，客户端保存的命令协议格式与预估格式相同，测试通过

6.1.3 测试代码

```
127.0.0.1:6379> set key value
```

```
127.0.0.1:6379> get key
```

```
127.0.0.1:6379> wrong key value
```

```
127.0.0.1:6379> get wrong
```

在源文件中输出保存命令协议的缓冲区：

```
1207        }
1208        printf("here is the querybuf: %s\n", c->querybuf);
1209        processInputBuffer(c);
```

6.1.4 测试结果及分析

期望结果：

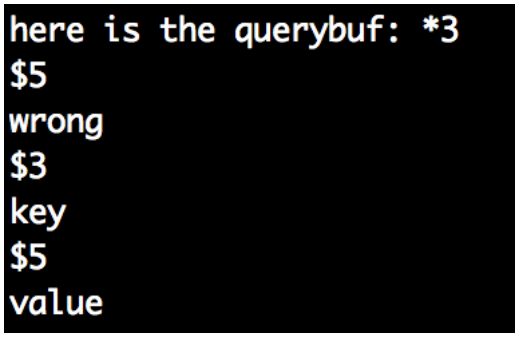
在客户端的输入缓冲区中应该存放输入命令转换成的协议格式。

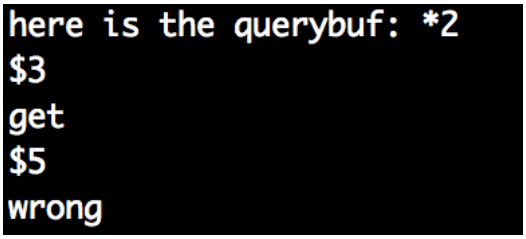
实际结果及分析：

客户端输入缓冲区保存的数据如下所示，无论是正确还是错误的命令，通信协议都能将命令成功转换为协议格式，并保存到客户端的输入缓冲区。

```
here is the querybuf: *3
$3
set
$3
key
$5
value
```

```
here is the querybuf: *2
$3
get
$3
key
```





6.2 命令请求读取测试

6.2.1 测试目标

本测试用例是用来测试服务器读取客户端请求的功能，通过在 redis 客户端中输入一系列正确的命令请求和异常的命令请求，分别查看 redis 客户端保存的 args 和 argv 属性，判断 redis 服务器是否正确读取并分析了 redis 客户端发送过来的协议格式。

6.2.2 测试用例

Test Case Specification		
Name	命令请求读取测试	
Brief Description	测试服务器是否能正确读取客户端的请求	
Precondition	Redis 服务器与客户端正在运行	
Tester	None	
Dependency	None	
Test Setup	Name	打开命令行
	Description	打开命令行界面以输入命令请求
Basic Flow (Test Setup)	Steps	
	1	打开操作系统的命令行界面
	2	打开命令行，进入 redis 源码路径
	Postcondition (Test Oracle)	命令行已经打开，处于等待输入命令状态
Basic Flow (Test Sequence)	Steps	
	1	测试员进入命令行
	2	测试员输入命令“cd redis/src” 进入 redis 目录

	3	输入”./redis-cli” 启动 redis 客户端
	4	输入“set key value ”发送命令请求
	5	输入“get key” 发送命令请求
	6	输入一系列错误命令请求
	7	测试员查看客户端状态的 argv 和 argc 属性
	Postcondition (Test Oracle)	命令输入完成，客户端状态中的 argv 与 argc 属性被成功保存，测试通过

6.2.3 测试代码

```
127.0.0.1:6379> set key value
```

```
127.0.0.1:6379> get key
```

```
127.0.0.1:6379> wrong key value
```

```
127.0.0.1:6379> get wrong
```

在源文件中添加代码，输出客户端的 argc 和 argv 属性：

```
140     printf("argc: %d\n", c->argc);
141     for (int i =0; i < c->argc; ++i) {
142         char *a = c->argv[i]->ptr;
143         printf("argv %i: %s\n", i, a);
144     }
```

6.2.4 测试结果

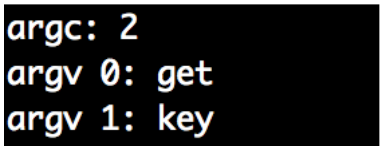
期望结果：

在客户端的 argc 中保存输入命令的参数个数，在 argv 中保存输入命令参数的名称。

实际结果及分析：

客户端中保存的 argc 和 argv 字段结果如下图所示，当命令输入正确时，客户端中可以保存正确的参数个数和名称，以供命令执行器查找命令并执行。而当输入命令错误时，客户端会抛出异常，不保存这些参数。

```
argc: 3
argv 0: set
argv 1: key
argv 2: value
```



6.3 命令请求执行测试

6.3.1 测试目标

本测试用例是用来测试命令执行器执行输入命令请求的功能，通过在 redis 客户端中输入一系列正确的命令请求和异常的命令请求，分别查看 redis 服务器保存在客户端的输出缓冲区里的命令执行结果，判断在命令请求异常的情况下，命令执行器能否成功识别；在命令请求正确的情况下，命令执行器能否正确执行命令并产生结果。

6.3.2 测试用例

Test Case Specification		
Name	命令请求执行测试	
Brief Description	测试命令执行器是否可以正确执行命令请求	
Precondition	Redis 服务器与客户端正常运行	
Tester	None	
Dependency	None	
Test Setup	Name	打开命令行
	Description	打开命令行界面以输入命令请求
Basic Flow (Test Setup)	Steps	
	1	打开操作系统的命令行界面
	2	打开命令行，进入 redis 源码路径
	Postcondition (Test Oracle)	命令行已经打开，处于等待输入命令状态
Basic Flow (Test Sequence)	Steps	
	1	测试员进入命令行
	2	测试员输入命令“cd redis/src” 进入 redis 目录
	3	输入”./redis-cli” 启动 redis 客户端
	4	输入“set key value ”发送命令请求
	5	输入“get key” 发送命令请求
	6	输入一系列错误命令请求
	7	命令执行器 VALIDATE THAT 客户端状态不为 null
	8	命令执行器 VALIDATE THAT 参数个数正确
	9	命令执行器 VALIDATE THAT 客户端通过身份验证
	10	测试员查看客户端状态的输出缓冲区的命令结果

	Postcondition (Test Oracle)	命令输入完成，客户端命令被成功执行，并保存到客户端状态的输出缓冲区
Specific Alternative Flows (Test Sequence)	RFS 1	
	1	命令执行器返回客户端状态为 null 的错误信息
	Postcondition (Test Oracle)	命令请求使客户端状态为 null 的情况被测试
	RFS 2	
	1	命令执行器返回参数个数不正确的错误信息
	Postcondition (Test Oracle)	参数个数不正确的命令请求被测试
	RFS 3	
	1	命令执行器返回客户端未通过身份验证的错误信息
	Postcondition (Test Oracle)	客户端未通过验证的请求被测试

6.3.3 测试代码

```
127.0.0.1:6379> set key value
```

```
127.0.0.1:6379> get key
```

```
127.0.0.1:6379> wrong key value
```

```
127.0.0.1:6379> get wrong
```

在源代码中添加输出缓冲区的代码：

```
printf("bufpos: %d\n", c->bufpos);
printf("buf: %s\n", c->buf);
```

6.3.4 测试结果

期望结果：

当客户端命令被成功执行，客户端状态的输出缓冲区应该保存输出命令执行结果的反馈。当客户端命令执行不成功时，客户端输出缓冲区应该保存出错信息。

实际结果及分析：

客户端中输出缓冲区 querybuf 中保存的数据如下所示，当命令执行成功时，保存+OK，否则保存了-ERR 出错信息，与预期结果相同。

```
bufpos: 5
buf: +OK
```

```
bufpos: 0
buf: +OK
```

```
bufpos: 0
buf: -ERR unknown command 'wrong'
```

6.4 命令请求回复测试

6.4.1 测试目标

本测试用例是用来测试服务器回复客户端请求的功能，通过在 redis 客户端中输入一系列正确的命令请求和异常的命令请求，分别查看命令行中打印出的命令回复结果，判断命令请求是否被成功回复。

6.4.2 测试用例

Test Case Specification		
Name	命令请求回复测试	
Brief Description	测试服务器是否能正确回复客户端的请求	
Precondition	Redis 服务器与客户端正在运行	
Tester	None	
Dependency	None	
Test Setup	Name	打开命令行
	Description	打开命令行界面以输入命令请求
Basic Flow (Test Setup)	Steps	
	1	打开操作系统的命令行界面
	2	打开命令行，进入 redis 源码路径
	Postcondition (Test Oracle)	命令行已经打开，处于等待输入命令状态
Basic Flow (Test Sequence)	Steps	
	1	测试员进入命令行
	2	测试员输入命令“cd redis/src” 进入 redis 目录
	3	输入”./redis-cli” 启动 redis 客户端
	4	输入“set key value ”发送命令请求
	5	输入“get key” 发送命令请求
	6	输入一系列错误命令请求
	7	查看命令行的输出结果
	Postcondition (Test Oracle)	命令输入完成，命令行中显示服务器返回的输出结果，测试通过。

6.4.3 测试代码

```
127.0.0.1:6379> set key value
```

```
127.0.0.1:6379> get key
```

```
127.0.0.1:6379> wrong key value
```

```
127.0.0.1:6379> get wrong
```

6.4.4 测试结果

期望结果：

当输入正确的命令时，客户端应该返回正确的结果；当输入不正确的命令时，客户端应该返回异常信息。

实际结果及分析：

当输入正确的命令时，如 set get ，可以获得期望的返回值；当输入错误的命令时，如 wrong 命令，则返回出错信息。说明命令请求回复可以正确执行。

```
127.0.0.1:6379> set key value
OK
127.0.0.1:6379> get key
"value"
127.0.0.1:6379> wrong key value
(error) ERR unknown command 'wrong'
127.0.0.1:6379> get wrong
(nil)
```

7 参考文献

[1] http://redis.io

[2] 黄健宏. Redis 设计与实现. 机械工业出版社[M]. 2014-06.

[3] 肖丁、吴建林等. 软件工程模型与方法[M]. 北京邮电大学出版社. 2008-03.

[4] http://my.oschina.net/hanruikai/blog/313724

[5] http://www.w3cschool.cc/redis/redis-benchmarks.html