

优极限

“极限教育，挑战极限”

www.yjxxt.com

极限教育，挑战极限。优极限是一个让 95% 的学生年薪过 18 万的岗前培训公司，让我们的学员具备优秀的互联网技术和职业素养，勇攀高薪，挑战极限。公司位于上海浦东，拥有两大校区，共万余平。累计培训学员超 3 万名。我们的训练营就业平均月薪 19000，最高年薪 50 万。

核心理念：让学员学会学习，拥有解决问题的能力，拿到高薪职场的钥匙。

项目驱动式团队协作、一对一服务、前瞻性思维、教练式培养模型-培养你成为就业明星。首创的老学员项目联盟给学员充分的项目、技术支撑，利用优极限平台这根杠杆，不断挑战极限，勇攀高薪，开挂人生。

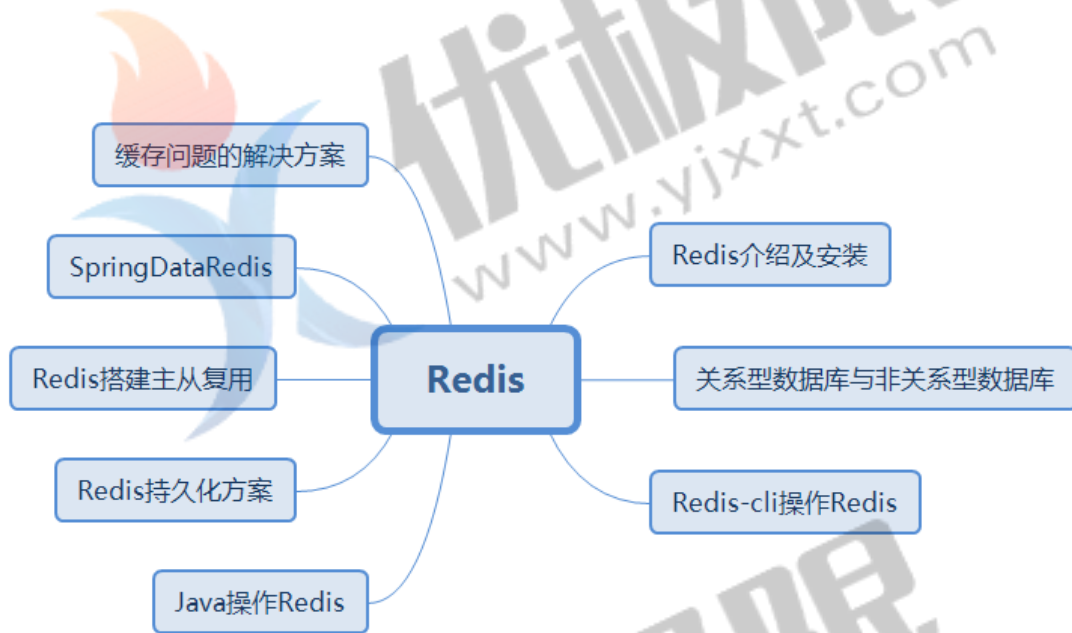
扫码关注优极限微信公众号：

（获取最新技术相关资讯及更多源码笔记）



Redis

学习目标



Redis的介绍及安装

Redis是什么？

Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如字符串（strings），散列（hashes），列表（lists），集合（sets），有序集合（sorted sets）与范围查询，bitmaps，hyperloglogs 和 地理空间（geospatial）索引半径查询。Redis 内置了复制（replication），LUA脚本（Lua scripting），LRU 驱动事件（LRU eviction），事务（transactions）和不同级别的磁盘持久化（persistence），并通过Redis哨兵（Sentinel）和自动分区（Cluster）提供高可用性（high availability）

性能

下面是官方的bench-mark数据：

测试完成了50个并发执行100000个请求。

设置和获取的值是一个256字节字符串。

结果:读的速度是110000次/s,写的速度是81000次/s

Redis历史简介

2008年，意大利一家创业公司Merzia的创始人Salvatore Sanfilippo为了避免MySQL的低性能，亲自定做一个数据库，并于2009年开发完成，这个就是Redis。

从2010年3月15日起，Redis的开发工作由VMware主持。

从2013年5月开始，Redis的开发由Pivotal赞助。

说明：Pivotal公司是由EMC和VMware联合成立的一家新公司。Pivotal希望为新一代的应用提供一个原生的基础，建立在具有领导力的云和网络公司不断转型的IT特性之上。Pivotal的使命是推行这些创新，提供给企业IT架构师和独立软件提供商。

支持语言

<ul style="list-style-type: none">ActionScriptCC++C#Clojure	<ul style="list-style-type: none">Common LispDartErlangGoHaskell	<ul style="list-style-type: none">HaxeIoJavaNode.jsLua	<ul style="list-style-type: none">Objective-CPerlPHPPure DataPython	<ul style="list-style-type: none">RRubyScalaSmalltalkTcl
---	--	--	---	--

支持的数据类型

string、hash、list、set、sorted set

安装

下载地址

<http://redis.io/>

上传至服务器

```
[root@localhost ~]# ll
总用量 201836
-rw-r--r--. 1 root root 1293 12月 20 18:14 anaconda-ks.cfg
-rw-r--r--. 1 root root 9653382 1月 19 14:11 apache-tomcat-8.5.37.tar.gz
-rw-r--r--. 1 root root 194042837 1月 18 16:10 jdk-8u202-linux-x64.tar.gz
drwxr-xr-x. 9 1001 1001 186 1月 19 14:35 nginx-1.14.2
-rw-r--r--. 1 root root 1015384 1月 19 14:33 nginx-1.14.2.tar.gz
-rw-r--r--. 1 root root 1959445 2月 1 11:26 redis-5.0.3.tar.gz
```

解压

```
tar zxvf redis-5.0.3.tar.gz
```

安装依赖

```
yum -y install gcc-c++ autoconf automake
```

```
总计 547 kB/s | 2.0 MB 00:00:03
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
 正在安装 : m4-1.4.16-10.el7.x86_64 1/6
 正在安装 : perl-Thread-Queue-3.02-2.el7.noarch 2/6
 正在安装 : perl-Test-Harness-3.28-3.el7.noarch 3/6
 正在安装 : perl-Data-Dumper-2.145-3.el7.x86_64 4/6
 正在安装 : autoconf-2.69-11.el7.noarch 5/6
 正在安装 : automake-1.13.4-3.el7.noarch 6/6
 验证中 : autoconf-2.69-11.el7.noarch 1/6
 验证中 : perl-Data-Dumper-2.145-3.el7.x86_64 2/6
 验证中 : perl-Test-Harness-3.28-3.el7.noarch 3/6
 验证中 : perl-Thread-Queue-3.02-2.el7.noarch 4/6
 验证中 : automake-1.13.4-3.el7.noarch 5/6
 验证中 : m4-1.4.16-10.el7.x86_64 6/6
已安装:
autoconf.noarch 0:2.69-11.el7 automake.noarch 0:1.13.4-3.el7
```

预编译

切换到解压目录

```
cd redis-5.0.3/  
make
```

```
LINK redis-server  
INSTALL redis-sentinel  
CC redis-cli.o  
LINK redis-cli  
CC redis-benchmark.o  
LINK redis-benchmark  
INSTALL redis-check-rdb  
INSTALL redis-check-aof  
  
Hint: It's a good idea to run 'make test' ;)  
make[1]: 离开目录“ /root/redis-5.0.3/src”
```

安装

创建安装目录

```
mkdir -p /usr/local/redis
```

不使用：make install (make install默认安装到/usr/local/bin目录下)

使用：如果需要指定安装路径，需要添加PREFIX参数

```
make PREFIX=/usr/local/redis/ install
```

```
Hint: It's a good idea to run 'make test' ;)  
  
INSTALL install  
INSTALL install  
INSTALL install  
INSTALL install  
INSTALL install  
make[1]: 离开目录“ /root/redis-5.0.3/src”
```

安装成功如图

```
[root@localhost bin]# pwd
/usr/local/redis/bin
[root@localhost bin]# ll
总用量 32672
-rwxr-xr-x. 1 root root 4367312 2月 1 11:35 redis-benchmark
-rwxr-xr-x. 1 root root 8092008 2月 1 11:35 redis-check-aof
-rwxr-xr-x. 1 root root 8092008 2月 1 11:35 redis-check-rdb
-rwxr-xr-x. 1 root root 4802672 2月 1 11:35 redis-cli
lrwxrwxrwx. 1 root root 12 2月 1 11:35 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 8092008 2月 1 11:35 redis-server
```

Redis-cli:客户端

Redis-server:服务器端

启动

安装的默认目标路径: /usr/local/redis/bin

```
[root@localhost ~]# cd /usr/local/redis/bin/
[root@localhost bin]# pwd
/usr/local/redis/bin
[root@localhost bin]# ll
总用量 8532
-rw-r--r--. 1 root root 18 4月 20 15:25 dump.rdb
-rwxr-xr-x. 1 root root 2076472 4月 20 15:22 redis-benchmark
-rwxr-xr-x. 1 root root 25168 4月 20 15:22 redis-check-aof
-rwxr-xr-x. 1 root root 56008 4月 20 15:22 redis-check-dump
-rwxr-xr-x. 1 root root 2206120 4月 20 15:22 redis-cli
lrwxrwxrwx. 1 root root 12 4月 20 15:22 redis-sentinel -> redis-server
-rwxr-xr-x. 1 root root 4358688 4月 20 15:22 redis-server
```

启动

```
./redis-server
```

```
[root@localhost bin]# ./redis-server
15630:C 24 Oct 09:36:46.722 # Warning: no config file specified, using the default config. In order to specify a config file use ./redis
15630:M 24 Oct 09:36:46.723 * Increased maximum number of open files to 10032 (it was originally set to 1024).

Redis 3.0.6 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 15630

http://redis.io

15630:M 24 Oct 09:36:46.755 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to
15630:M 24 Oct 09:36:46.755 # Server started, Redis version 3.0.6
15630:M 24 Oct 09:36:46.756 # WARNING overcommit_memory is set to 0! Background save may fail under low memory condition. To fix this i
and then reboot or run the command 'sysctl vm.overcommit_memory=1' for this to take effect.
15630:M 24 Oct 09:36:46.756 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency an
the command 'echo never > /sys/kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the
HP is disabled.
15630:M 24 Oct 09:36:46.756 * The server is now ready to accept connections on port 6379
```

默认为前台启动, 修改为后台启动

复制redis.conf至安装路径下

```
cp redis.conf /usr/local/redis/bin/
```

修改安装路径下的redis.conf, 将daemonize修改为yes


```
134 # By default Redis does not run as a daemon. Use 'yes' if you need it.
135 # Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
136 daemonize yes
```

启动时，指定配置文件路径即可

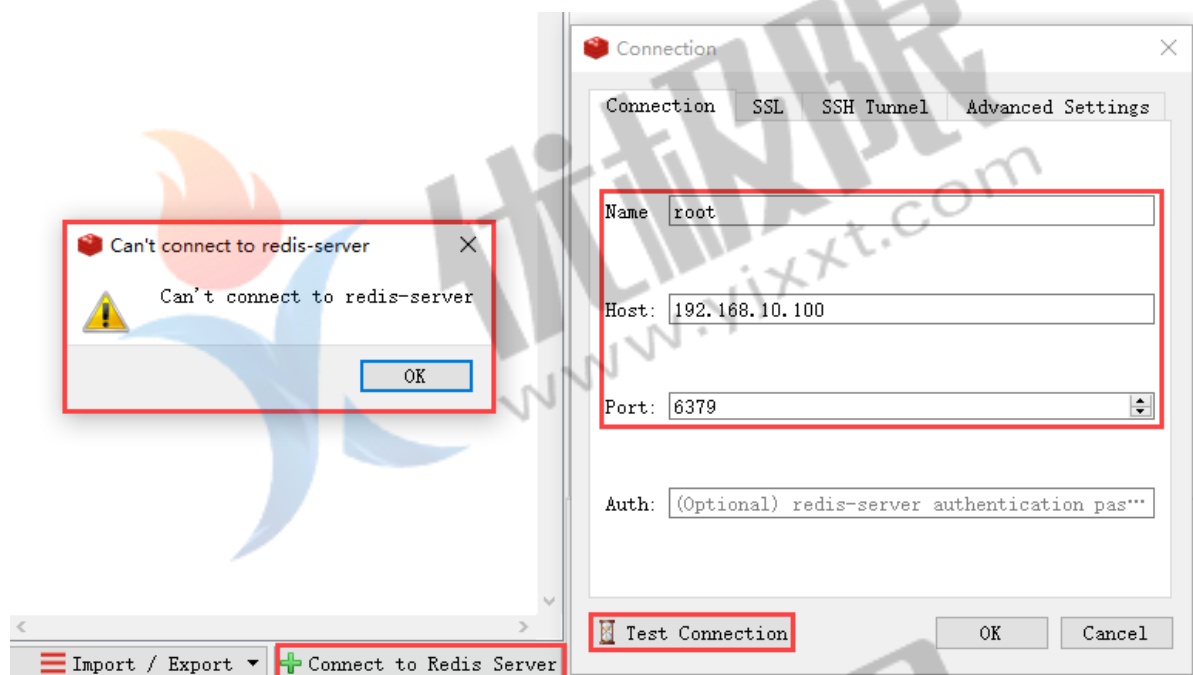
```
[root@localhost bin]# ./redis-server ./redis.conf
14505:C 02 Aug 11:17:04.228 # oOoOoOoOoOoOo Redis is starting oOoOoOoOoOoOo
14505:C 02 Aug 11:17:04.228 # Redis version=4.0.10, bits=64, commit=00000000, modified=0, pid=14505, just started
14505:C 02 Aug 11:17:04.228 # Configuration loaded
[root@localhost bin]# ps -ef | grep redis
root      14506      1   0 11:17 ?        00:00:00 ./redis-server 127.0.0.1:6379
root      14511  14416   0 11:17 pts/2    00:00:00 grep --color=auto redis
```

通过windows客户端访问

安装Redis客户端

 redis-desktop-manager-0.8.8.384.exe 应用程序 27,828 KB

建立连接->失败



修改配置文件redis.conf

注释掉 bind 127.0.0.1 可以使所有的ip访问redis，若是想指定多个ip访问，但并不是全部的ip访问，可以bind设置

```
66 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
67 # JUST COMMENT THE FOLLOWING LINE.
68 # ~~~~~
69 #bind 127.0.0.1
```

关闭保护模式，修改为no


```

84 # By default protected mode is enabled. You should disable it only if
85 # you are sure you want clients from other hosts to connect to Redis
86 # even if no authentication is configured, nor a specific set of interfaces
87 # are explicitly listed using the "bind" directive.
88 protected-mode no

```

添加访问认证

```

496 # Warning: since Redis is pretty fast an outside user can try up to
497 # 150k passwords per second against a good box. This means that you should
498 # use a very strong password otherwise it will be very easy to break.
499 #
500 requirepass root

```

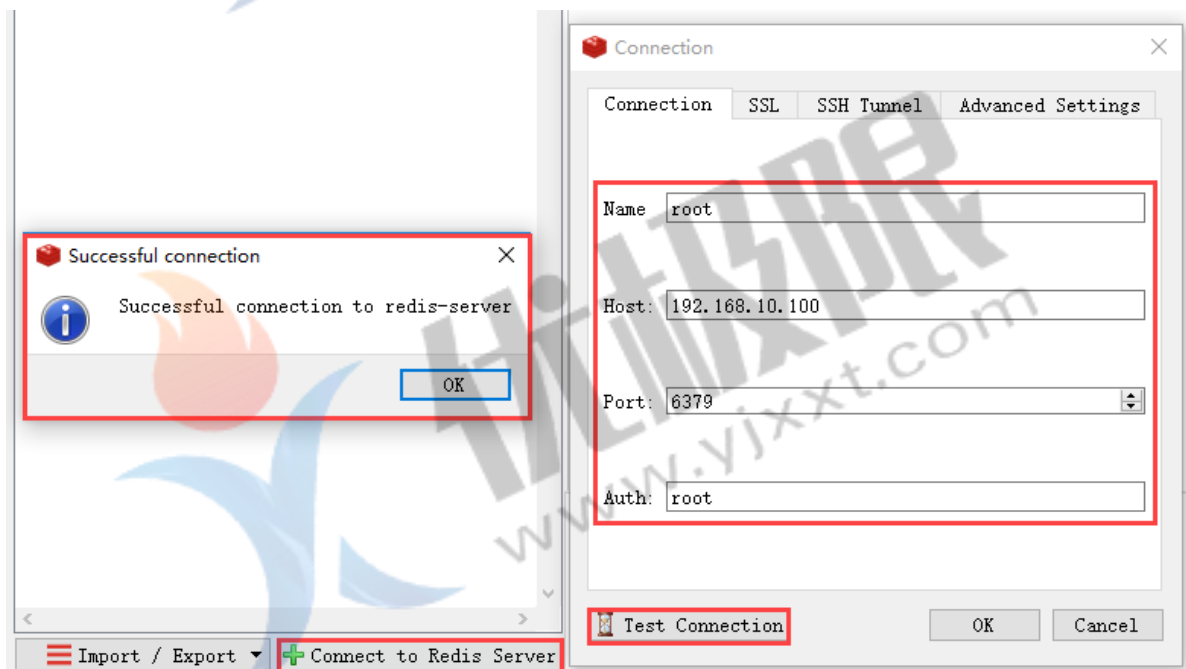
修改后kill -9 XXXX杀死redis进程，重启redis

```

[root@localhost bin]# ps -ef | grep redis
root      5688      1  0 16:15 ?        00:00:00 ./redis-server *:6379
root      5693    2219  0 16:16 pts/0    00:00:00 grep --color=auto redis
[root@localhost bin]# kill -9 5688
[root@localhost bin]# ./redis-server ./redis.conf

```

再次建立连接 -> 成功



我们可以修改默认数据库的数量 默认16

```

183 # Set the number of databases. The default database is DB 0, you can select
184 # a different one on a per-connection basis using SELECT <dbid> where
185 # dbid is a number between 0 and 'databases'-1
186 databases 16

```

修改database 32则默认为32个数据库

修改后kill -9 XXXX杀死redis进程，重启redis即可看到效果

```

[root@localhost bin]# ps -ef | grep redis
root      5688      1  0 16:15 ?        00:00:00 ./redis-server *:6379
root      5693    2219  0 16:16 pts/0    00:00:00 grep --color=auto redis
[root@localhost bin]# kill -9 5688
[root@localhost bin]# ./redis-server ./redis.conf

```

关系型数据库与非关系型数据库

关系型数据库

采用关系模型来组织数据的数据库，关系模型就是二维表格模型。一张二维表的表名就是关系，二维表中的一行就是一条记录，二维表中的一列就是一个字段。

优点

- 容易理解
- 使用方便，通用的sql语言
- 易于维护，丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大降低了数据冗余和数据不一致的概率

缺点

- 磁盘I/O是并发的瓶颈
- 海量数据查询效率低
- 横向扩展困难，无法简单的通过添加硬件和服务节点来扩展性能和负载能力，当需要对数据库进行升级和扩展时，需要停机维护和数据迁移
- 多表的关联查询以及复杂的数据分析类型的复杂sql查询，性能欠佳。因为要保证acid，必须按照三范式设计。

数据库

Oracle, Sql Server, MySql, DB2

非关系型数据库

非关系型，分布式，一般不保证遵循ACID原则的数据存储系统。键值对存储，结构不固定。

优点

- 根据需要添加字段，不需要多表联查。仅需id取出对应的value
- 适用于SNS（社会化网络服务软件。比如facebook，微博）
- 严格上讲不是一种数据库，而是一种数据结构化存储方法的集合

缺点

- 只适合存储一些较为简单的数据
- 不合适复杂查询的数据
- 不合适持久存储海量数据

数据库

- K-V: Redis, Memcache
- 文档: MongoDB
- 搜索: Elasticsearch, Solr
- 可扩展性分布式: HBase

比较

内容	关系型数据库	非关系型数据库
成本	有些需要收费 (Oracle)	基本都是开源
查询数据	存储于硬盘中, 速度慢	数据存于缓存中, 速度快
存储格式	只支持基础类型	K-V, 文档, 图片等
扩展性	有多表查询机制, 扩展困难	数据之间没有耦合, 容易扩展
持久性	适用持久存储, 海量存储	不适用持久存储, 海量存储
数据一致性	事务能力强, 强调数据的强一致性	事务能力弱, 强调数据的最终一致性

Redis-cli操作Redis

Redis-cli连接Redis

- h :用于指定ip
- p :用于指定端口
- a :用于指定认证密码

```
[root@localhost bin]# ./redis-cli -p 6379 -a root
```

PING命令返回PONG

```
127.0.0.1:6379> PING
PONG
```

指定database

```
127.0.0.1:6379> SELECT 3
OK
```

Redis-cli操作Redis

操作String

- set :添加一条String类型数据
- get :获取一条String类型数据
- mset :添加多条String类型数据
- mget :获取多条String类型数据

```
127.0.0.1:6379[3]> set username zhangsan
OK
127.0.0.1:6379[3]> mset address bj sex 1
OK
127.0.0.1:6379[3]> get username
"zhangsan"
127.0.0.1:6379[3]> mget username address sex
1) "zhangsan"
2) "bj"
3) "1"
```

操作hash

`hset`:添加一条hash类型数据

`hget`:获取一条hash类型数据

`hmset`:添加多条hash类型数据

`hmget`:获取多条hash类型数据

`hgetall`:获取指定所有hash类型数据

`hdel`:删除指定hash类型数据(一条或多条)

```
127.0.0.1:6379> hset userInfo name lisi
(integer) 1
127.0.0.1:6379> hmset userInfo age 20 sex 1
OK
127.0.0.1:6379> hget userInfo name
"lisi"
127.0.0.1:6379> hmget userInfo age sex
1) "20"
2) "1"
127.0.0.1:6379> hgetall userInfo
1) "name"
2) "lisi"
3) "age"
4) "20"
5) "sex"
6) "1"
127.0.0.1:6379> hdel userInfo name age
(integer) 2
```

操作list

`lpush`:左添加(头)list类型数据

`rpush`:右添加(尾)类型数据

`lrange`: 获取list类型数据start起始下标 end结束下标 包含关系

`llen`:获取条数

`lrem`:删除列表中几个指定list类型数据

```
127.0.0.1:6379> lpush students wangwu lisi
(integer) 2
127.0.0.1:6379> rpush students zhaoliu
(integer) 3
127.0.0.1:6379> lrange students 0 2
1) "lisi"
2) "wangwu"
3) "zhaoliu"
127.0.0.1:6379> llen students
(integer) 3
127.0.0.1:6379> lrem students 1 lisi
(integer) 1
```

操作set

`sadd`:添加set类型数据

`smembers`:获取set类型数据

`scard`:获取条数

`srem`:删除数据

```
127.0.0.1:6379> sadd letters aaa bbb ccc ddd eee
(integer) 5
127.0.0.1:6379> smembers letters
1) "bbb"
2) "aaa"
3) "ddd"
4) "ccc"
5) "eee"
127.0.0.1:6379> scard letters
(integer) 5
127.0.0.1:6379> srem letters aaa bbb
(integer) 2
```

操作sorted set

sorted set是通过分数值来进行排序的，分数值越大，越靠后。

`zadd`:添加sorted set类型数据

`zrange`:获取sorted set类型数据

`zcard`:获取条数

`zrem`:删除数据

`zadd`需要将Float或者Double类型分数值参数，放置在值参数之前

```
127.0.0.1:6379> zadd score 7 zhangsan 3 lisi 5 wangwu 6 zhaoliu 2 tianqi
(integer) 5
127.0.0.1:6379> zrange score 0 4
1) "tianqi"
2) "lisi"
3) "wangwu"
4) "zhaoliu"
5) "zhangsan"
127.0.0.1:6379> zcard score
(integer) 5
127.0.0.1:6379> zrem score lisi
(integer) 1
```

Redis中以层级关系、目录形式存储数据

```
127.0.0.1:6379> mset user:01 zhangsan
OK
127.0.0.1:6379> mget user:01
1) "zhangsan"
```



The image shows the Redis Desktop Manager interface. On the left, a tree view displays the database structure: db0 (1/0) contains a folder 'user (1)', which contains a key 'user:01'. Other databases db1 (0) and db2 (0) are also listed. On the right, the details for the 'user:01' key are shown, indicating it is a STRING type with a value of 'zhangsan' and a size of 8 bytes.

设置key的失效时间

Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候会被删除)：

EXPIRE <key> <ttl> :用于将键 key 的生存时间设置为 ttl 秒。

PEXPIRE <key> <ttl> :用于将键 key 的生存时间设置为 ttl 毫秒。

EXPIREAT <key> < timestamp> :用于将键 key 的过期时间设置为 timestamp 所指定的秒数时间戳。

PEXPIREAT <key> < timestamp > :用于将键 key 的过期时间设置为 timestamp 所指定的毫秒数时间戳。

TTL :获取的值为-1说明此 key 没有设置有效期，当值为-2时证明过了有效期。

方法一

```
127.0.0.1:6379> set code test EX 180
OK
127.0.0.1:6379> ttl code
(integer) 167
127.0.0.1:6379> ttl code
(integer) 165
```

方法二

```
127.0.0.1:6379> set code test
OK
127.0.0.1:6379> expire code 180
(integer) 1
127.0.0.1:6379> ttl code
(integer) 177
127.0.0.1:6379> ttl code
(integer) 175
```

方法三

第一个参数: `key`

第二个参数: `value`

第三个参数: `NX` 是不存在时才set, `XX` 是存在时才set

第四个参数: `EX` 是秒, `PX` 是毫秒

```
127.0.0.1:6379> set code test nx ex 180
OK
127.0.0.1:6379> ttl code
(integer) 163
127.0.0.1:6379> ttl code
(integer) 162
```

删除

`del`: 用于删除数据 (通用, 适用于所有数据类型)

`hdel`: 用于删除hash类型数据

```
127.0.0.1:6379[3]> del username
(integer) 1
127.0.0.1:6379[3]> hdel userInfo name
(integer) 1
127.0.0.1:6379[3]> del userInfo
(integer) 1
```

tips: 命令为java中方法名, 参数: 去除括号, 引号, 将逗号变空格即可

// 添加一条数据

```
jedis.set("username", "zhangsan");
```

```
jedis.set("age", "18");
```

去除括号，引号，将逗号变空格

// 添加多条数据 参数奇数为key 参数偶数为value

```
jedis.mset("address", "bj", "sex", "1");
```

```
127.0.0.1:6379[3]> set username zhangsan
OK
127.0.0.1:6379[3]> mset address bj sex 1
OK
```

zadd需要将Float或者Double类型参数，放置在值参数之前

```
127.0.0.1:6379> zadd score 7 zhangsan 3 lisi 5 wangwu 6 zhaoliu 2 tianqi
(integer) 5
127.0.0.1:6379> zrange score 0 4
1) "tianqi"
2) "lisi"
3) "wangwu"
4) "zhaoliu"
5) "zhangsan"
127.0.0.1:6379> zcard score
(integer) 5
127.0.0.1:6379> zrem score lisi
(integer) 1
```

```
Map<String, Double> scoreMembers = new HashMap<>();
scoreMembers.put("zhangsan", 7D);
scoreMembers.put("lisi", 3D);
scoreMembers.put("wangwu", 5D);
scoreMembers.put("zhaoliu", 6D);
scoreMembers.put("tianqi", 2D);
```

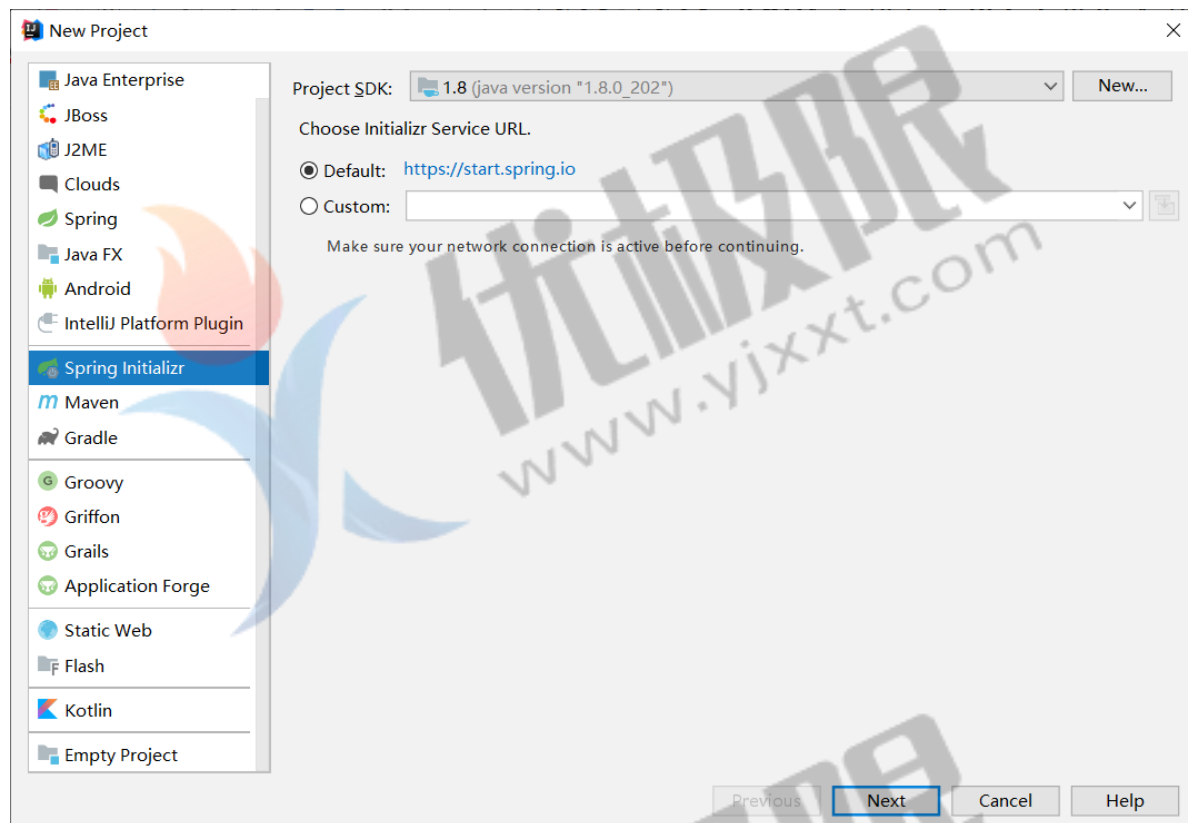
// 添加数据

```
jedis.zadd( key: "score", scoreMembers);
```

Java操作Redis

创建项目

创建项目



The 'New Project' dialog box shows a list of project types on the left. 'Spring Initializr' is selected. On the right, the 'Project SDK' is set to '1.8 (java version "1.8.0_202")'. The 'Choose Initializr Service URL' section has 'Default: https://start.spring.io' selected. A note below says 'Make sure your network connection is active before continuing.' At the bottom, there are 'Previous', 'Next', 'Cancel', and 'Help' buttons.

New Project

Project SDK: 1.8 (java version "1.8.0_202") New...

Choose Initializr Service URL.

☒ Default: <https://start.spring.io>

☐ Custom: ...

Make sure your network connection is active before continuing.

Previous Next Cancel Help



The 'New Project' dialog box shows the 'Project Metadata' section. Fields include Group (com.xxxx), Artifact (redisdemo), Type (Maven Project), Language (Java), Packaging (Jar), Java Version (8), Version (0.0.1-SNAPSHOT), Name (redisdemo), Description (Demo project for Spring Boot), and Package (com.xxxx.redisdemo). At the bottom, there are 'Previous', 'Next', 'Cancel', and 'Help' buttons.

New Project

Project Metadata

Group: com.xxxx

Artifact: redisdemo

Type: Maven Project (Generate a Maven based project archive.)

Language: Java

Packaging: Jar

Java Version: 8

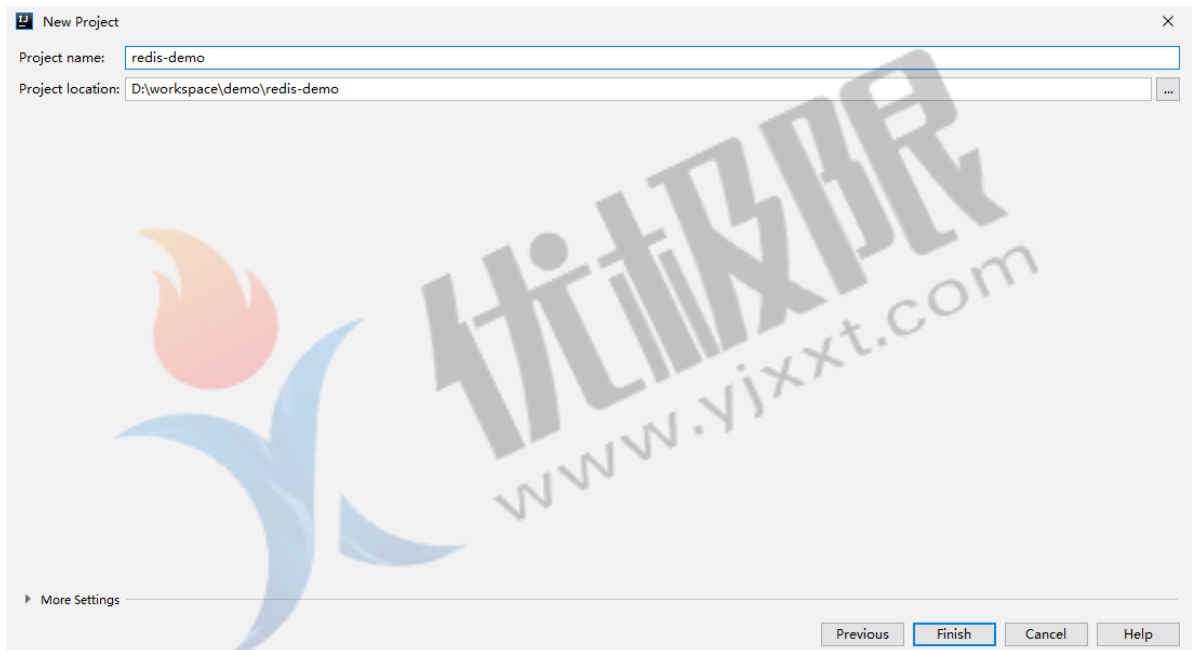
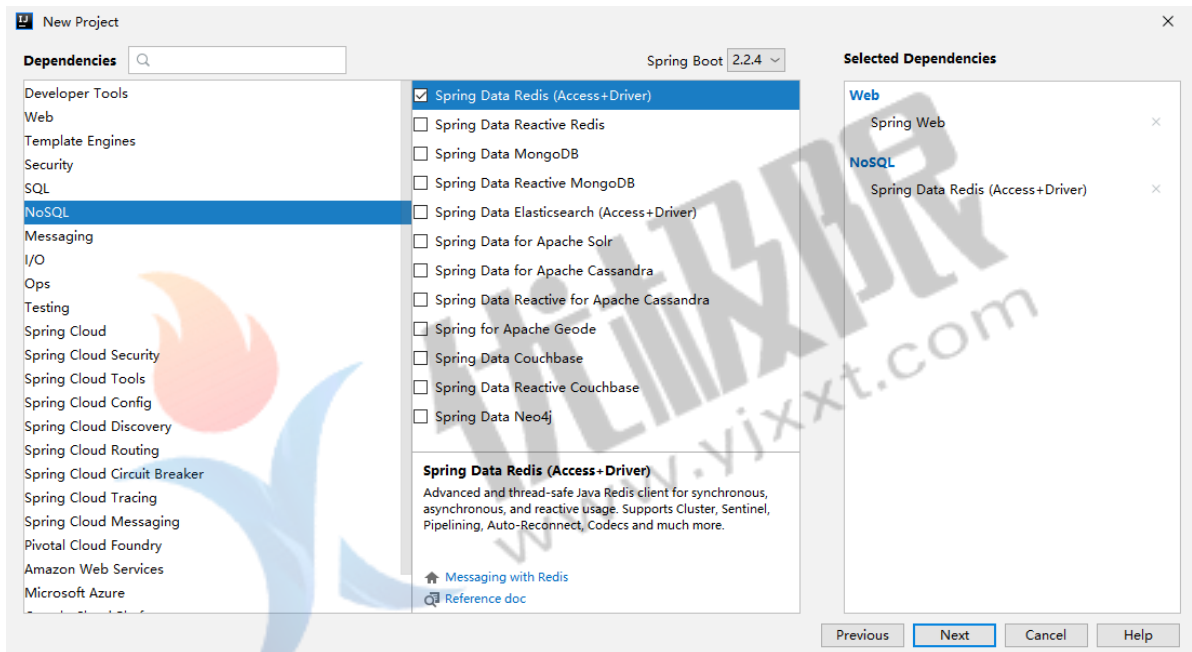
Version: 0.0.1-SNAPSHOT

Name: redisdemo

Description: Demo project for Spring Boot

Package: com.xxxx.redisdemo

Previous Next Cancel Help



添加依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>

  <groupId>com.xxxx</groupId>
  <artifactId>redisdemo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

```

<name>redisdemo</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <!-- spring data redis 组件 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    <!--
        1.x 的版本默认采用的连接池技术是 Jedis,
        2.0 以上版本默认连接池是 Lettuce,
        如果采用 Jedis, 需要排除 Lettuce 的依赖。
    -->
    <exclusions>
        <exclusion>
            <groupId>io.lettuce</groupId>
            <artifactId>lettuce-core</artifactId>
        </exclusion>
    </exclusions>
    </dependency>
    <!-- jedis 依赖 -->
    <dependency>
        <groupId>redis.clients</groupId>
        <artifactId>jedis</artifactId>
    </dependency>
    <!-- web 组件 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <!-- test 组件 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
</project>

```

配置文件

```

spring:
  redis:
    # Redis服务器地址
    host: 192.168.10.100
    # Redis服务器端口
    port: 6379
    # Redis服务器密码
    password: root
    # 选择哪个库, 默认0库
    database: 0
    # 连接超时时间
    timeout: 10000ms

```

```
jedis:
  pool:
    # 最大连接数, 默认8
    max-active: 1024
    # 最大连接阻塞等待时间, 单位毫秒, 默认-1ms
    max-wait: 10000ms
    # 最大空闲连接, 默认8
    max-idle: 200
    # 最小空闲连接, 默认0
    min-idle: 5
```

Java怎么连接Redis?

```
/**
 * 连接Redis
 */
@Test
public void initConn01() {
    // 创建jedis对象, 连接redis服务
    Jedis jedis = new Jedis("192.168.10.100", 6379);

    // 设置认证密码
    jedis.auth("root");

    // 指定数据库 默认是0
    jedis.select(1);

    // 使用ping命令, 测试连接是否成功
    String result = jedis.ping();
    System.out.println(result); // 返回PONG

    // 添加一条数据
    jedis.set("username", "zhangsan");

    // 获取一条数据
    String username = jedis.get("username");
    System.out.println(username);

    // 释放资源
    if (jedis != null)
        jedis.close();
}
```

通过Redis连接池获取连接对象并操作服务器

```
/**
 * 通过Redis连接池获取连接对象
 */
@Test
public void initConn02() {
    // 初始化redis客户端连接池
    JedisPool jedisPool = new JedisPool(new JedisPoolConfig(), "192.168.10.100",
    6379, 10000, "root");

    // 从连接池获取连接
    Jedis jedis = jedisPool.getResource();
```

```

// 指定数据库 默认是0
jedis.select(2);

// 使用ping命令, 测试连接是否成功
String result = jedis.ping();
System.out.println(result); // 返回PONG

// 添加一条数据
jedis.set("username", "zhangsan");

// 获取一条数据
String username = jedis.get("username");
System.out.println(username);

// 释放资源
if (jedis != null)
    jedis.close();
}

```

封装JedisUtil对外提供连接对象获取方法

```

@Configuration
public class RedisConfig {
    //服务器地址
    @Value("${spring.redis.host}")
    private String host;
    //端口
    @Value("${spring.redis.port}")
    private int port;
    //密码
    @Value("${spring.redis.password}")
    private String password;
    //超时时间
    @Value("${spring.redis.timeout}")
    private String timeout;
    //最大连接数
    @Value("${spring.redis.jedis.pool.max-active}")
    private int maxTotal;
    //最大连接阻塞等待时间
    @Value("${spring.redis.jedis.pool.max-wait}")
    private String maxWaitMillis;
    //最大空闲连接
    @Value("${spring.redis.jedis.pool.max-idle}")
    private int maxIdle;
    //最小空闲连接
    @Value("${spring.redis.jedis.pool.min-idle}")
    private int minIdle;

    @Bean
    public JedisPool redisPoolFactory() {
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
        //注意值的转变
        jedisPoolConfig.setMaxWaitMillis(Long.parseLong(maxWaitMillis.substring(0, maxWaitMillis.length() - 2)));
        //注意属性名
    }
}

```

```

        jedisPoolConfig.setMaxTotal(maxTotal);
        jedisPoolConfig.setMaxIdle(maxIdle);
        jedisPoolConfig.setMinIdle(minIdle);
        JedisPool jedisPool = new JedisPool(jedisPoolConfig, host, port,
Integer.parseInt(timeout.substring(0,
        timeout.length() - 2)), password);
        return jedisPool;
    }
}

```

Java操作Redis五种数据类型

连接与释放

```

@Autowired
private JedisPool jedisPool;

private Jedis jedis = null;

//初始化jedis对象实例
@Before
public void initConn(){
    jedis = jedisPool.getResource();
}

//释放资源
@After
public void closeConn(){
    if (jedis!=null){
        jedis.close();
    }
}

```

操作String

```

// 1.操作String
@Test
public void testString() {
    // 添加一条数据
    jedis.set("username", "zhangsang");
    jedis.set("age", "18");

    // 添加多条数据 参数奇数为key 参数偶数为value
    jedis.mset("address", "bj", "sex", "1");

    // 获取一条数据
    String username = jedis.get("username");
    System.out.println(username);

    // 获取多条数据
    List<String> list = jedis.mget("username", "age", "address", "sex");
    for (String str : list) {
        System.out.println(str);
    }

    // 删除
}

```



```
//jedis.del("username");  
}
```

操作hash

```
// 2.操作Hash  
@Test  
public void testHash() {  
    /*  
     * 添加一条数据  
     * 参数一: redis的key  
     * 参数二: hash的key  
     * 参数三: hash的value  
     */  
    jedis.hset("userInfo", "name", "lisi");  
  
    // 添加多条数据  
    Map<String, String> map = new HashMap<>();  
    map.put("age", "20");  
    map.put("sex", "1");  
    jedis.hmset("userInfo", map);  
  
    // 获取一条数据  
    String name = jedis.hget("userInfo", "name");  
    System.out.println(name);  
  
    // 获取多条数据  
    List<String> list = jedis.hmget("userInfo", "age", "sex");  
    for (String str : list) {  
        System.out.println(str);  
    }  
  
    // 获取Hash类型所有的数据  
    Map<String, String> userMap = jedis.hgetAll("userInfo");  
    for (Entry<String, String> userInfo : userMap.entrySet()) {  
        System.out.println(userInfo.getKey() + "--" + userInfo.getValue());  
    }  
  
    // 删除 用于删除hash类型数据  
    //jedis.hdel("userInfo", "name");  
}
```

操作list

```
// 3.操作list  
@Test  
public void testList() {  
    // 左添加(上)  
    // jedis.lpush("students", "Wang Wu", "Li Si");  
  
    // 右添加(下)  
    // jedis.rpush("students", "Zhao Liu");  
  
    // 获取 start起始下标 end结束下标 包含关系  
    List<String> students = jedis.lrange("students", 0, 2);  
    for (String stu : students) {
```

```

        System.out.println(stu);
    }

    // 获取总条数
    Long total = jedis.llen("students");
    System.out.println("总条数: " + total);

    // 删除单条 删除列表中第一次出现的Li Si
    // jedis.lrem("students", 1, "Li Si");

    // 删除多条
    // jedis.del("students");
}

```

操作set

```

// 4.操作set-无序
@Test
public void testSet() {
    // 添加数据
    jedis.sadd("letters", "aaa", "bbb", "ccc", "ddd", "eee");

    // 获取数据
    Set<String> letters = jedis.smembers("letters");
    for (String letter: letters) {
        System.out.println(letter);
    }

    // 获取总条数
    Long total = jedis.scard("letters");
    System.out.println(total);

    // 删除
    // jedis.srem("letters", "aaa", "bbb");
}

```

操作sorted set

```

// 5.操作sorted set-有序
@Test
public void testSortedSet() {
    Map<String, Double> scoreMembers = new HashMap<>();
    scoreMembers.put("zhangsan", 7D);
    scoreMembers.put("lisi", 3D);
    scoreMembers.put("wangwu", 5D);
    scoreMembers.put("zhao Liu", 6D);
    scoreMembers.put("tianqi", 2D);

    // 添加数据
    jedis.zadd("score", scoreMembers);

    // 获取数据
    Set<String> scores = jedis.zrange("score", 0, 4);
    for (String score: scores) {
        System.out.println(score);
    }
}

```

```

    }

    // 获取总条数
    Long total = jedis.zcard("score");
    System.out.println("总条数: " + total);

    // 删除
    //jedis.zrem("score", "zhangsan", "lisi");
}

```

Redis中以层级关系、目录形式存储数据

```

// Redis中以层级关系、目录形式存储数据
@Test
public void testdir(){
    jedis.set("user:01", "user_zhangsan");
    System.out.println(jedis.get("user:01"));
}

```

设置key的失效时间

Redis 有四个不同的命令可以用于设置键的生存时间(键可以存在多久)或过期时间(键什么时候会被删除):

EXPIRE <key> <ttl> :用于将键 key 的生存时间设置为 ttl 秒。

PEXPIRE <key> <ttl> :用于将键 key 的生存时间设置为 ttl 毫秒。

EXPIREAT <key> < timestamp> :用于将键 key 的过期时间设置为 timestamp 所指定的秒数时间戳。

PEXPIREAT <key> < timestamp > :用于将键 key 的过期时间设置为 timestamp 所指定的毫秒数时间戳。

TTL :获取的值为-1说明此 key 没有设置有效期, 当值为-2时证明过了有效期。

```

@Test
public void testExpire() {
    // 方法一:
    jedis.set("code", "test");
    jedis.expire("code", 180); // 180秒
    jedis.pexpire("code", 180000L); // 180000毫秒
    jedis.ttl("code"); // 获取秒

    // 方法二:
    jedis.setex("code", 180, "test"); // 180秒
    jedis.psetex("code", 180000L, "test"); // 180000毫秒
    jedis.pttl("code"); // 获取毫秒

    // 方法三:
    SetParams setParams = new SetParams();
    //不存在的时候才能设置成功
    // setParams.nx();
    // 存在的时候才能设置成功
    setParams.xx();
    //设置失效时间, 单位秒
    // setParams.ex(30);
    //查看失效时间, 单位毫秒
    setParams.px(30000);
}

```

```
jedis.set("code", "test", setParams);  
}
```

获取所有key&事务&删除

```
// 获取所有key  
@Test  
public void testAllKeys() {  
    // 当前库key的数量  
    System.out.println(jedis.dbSize());  
    // 当前库key的名称  
    Set<String> keys = jedis.keys("*");  
    for (String key: keys) {  
        System.out.println(key);  
    }  
}  
  
// 操作事务  
@Test  
public void testMulti() {  
    Transaction tx = jedis.multi();  
    // 开启事务  
    tx.set("tel", "10010");  
  
    // 提交事务  
    tx.exec();  
  
    // 回滚事务  
    tx.discard();  
}  
  
// 删除  
@Test  
public void testDelete() {  
    // 删除 通用 适用于所有数据类型  
    jedis.del("score");  
}
```

操作byte

SerializeUtil.java

```
package com.xxxx.util;  
  
import java.io.ByteArrayInputStream;  
import java.io.ByteArrayOutputStream;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
  
/**  
 * 序列化工具类  
 */  
public class SerializeUtil {  
  
    /**  
     * 将java对象转换为byte数组 序列化过程  
     */  
}
```

```

    */
    public static byte[] serialize(Object object) {
        ObjectOutputStream oos = null;
        ByteArrayOutputStream baos = null;
        try {
            // 序列化
            baos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(baos);
            oos.writeObject(object);
            byte[] bytes = baos.toByteArray();
            return bytes;
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    /**
     * 将byte数组转换为java对象 反序列化
     */
    public static Object unserialize(byte[] bytes) {
        if(bytes == null) return null;
        ByteArrayInputStream bais = null;
        try {
            // 反序列化
            bais = new ByteArrayInputStream(bytes);
            ObjectInputStream ois = new ObjectInputStream(bais);
            return ois.readObject();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}

```

User.java

```

package com.xxxx.entity;

import java.io.Serializable;

public class User implements Serializable {

    private static final long serialVersionUID = 9148937431079191022L;
    private Integer id;
    private String username;
    private String password;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {

```

```

        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
            "id=" + id +
            ", username='" + username + '\'' +
            ", password='" + password + '\'' +
            '}';
    }
}

```

JedisTest.java

```

// 操作byte
@Test
public void testByte() {
    User user = new User();
    user.setId(2);
    user.setUsername("zhangsan");
    user.setPassword("123");

    // 序列化
    byte[] userKey = SerializeUtil.serialize("user:" + user.getId());
    byte[] userValue = SerializeUtil.serialize(user);
    jedis.set(userKey, userValue);

    // 获取数据
    byte[] userResult = jedis.get(userKey);

    // 反序列化
    User u = (User) SerializeUtil.unserialize(userResult);
    System.out.println(u);
}

```

Redis搭建主从复用

Redis支持主从复用。数据可以从主服务器向任意数量的从服务器上同步，同步使用的是发布/订阅机制。Master Slave的模式，从Slave向Master发起SYNC命令。

可以是1 Master 多Slave，可以分层，Slave下可以再接Slave，可扩展成树状结构。

因为没有两台电脑，所以只能在一台机器上搭建两个Redis服务端。

这里使用单机来模拟redis 主从服务器，实现读写分离配置。

读写分离

创建三个目录（数据文件、日志文件、配置文件）

```
[root@localhost local]# mkdir -p /opt/redis/data
[root@localhost local]# mkdir -p /opt/redis/log
[root@localhost local]# mkdir -p /opt/redis/conf
[root@localhost local]# ls /opt/redis/
conf  data  log
```

复制redis.conf至/opt/redis/conf目录下

```
[root@localhost redis-4.0.10]# cp redis.conf /opt/redis/conf/redis-common.conf
[root@localhost redis-4.0.10]# ls /opt/redis/conf/
redis-common.conf
```

修改redis-common.conf公共配置文件

注释掉bind 127.0.0.1

```
66 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
67 # JUST COMMENT THE FOLLOWING LINE.
68 # ~~~~~
69 #bind 127.0.0.1
```

关闭保护模式，修改为no

```
84 # By default protected mode is enabled. You should disable it only if
85 # you are sure you want clients from other hosts to connect to Redis
86 # even if no authentication is configured, nor a specific set of interfaces
87 # are explicitly listed using the "bind" directive.
88 protected-mode no
```

注释公共配置端口

```
90 # Accept connections on the specified port, default is 6379 (IANA #815344).
91 # If port 0 is specified Redis will not listen on a TCP socket.
92 #port 6379
```

修改为后台启动

```
134 # By default Redis does not run as a daemon. Use 'yes' if you need it.
135 # Note that Redis will write a pid file in /var/run/redis.pid when daemonized.
136 daemonize yes
```

注释进程编号记录文件

```
156 # Creating a pid file is best effort: if Redis is not able to create it
157 # nothing bad happens, the server will start and run normally.
158 #pidfile /var/run/redis_6379.pid
```

注释公共配置日志文件

```
168 # Specify the log file name. Also the empty string can be used to force
169 # Redis to log on the standard output. Note that if you use standard
170 # output for logging but daemonize, logs will be sent to /dev/null
171 #logfile ""
```

注释公共配置数据文件、修改数据文件路径

在默认情况下, Redis 将数据库快照保存在名字为 `dump.rdb` 的二进制文件中。当然, 这里可以通过修改 `redis.conf` 配置文件来对数据存储条件进行定义, 规定在“N 秒内数据集至少有 M 个改动”这一条件被满足时,自动保存一次数据集。也可以通过调用 `save` 或 `bgsave` ,手动让Redis进行数据集保存操作

`dbfilename`和`dir`组合使用, `dbfilename`找`dir`路径生成数据文件

```
252 # The filename where to dump the DB
253 #dbfilename dump.rdb
254
255 # The working directory.
256 #
257 # The DB will be written inside this directory, with the filename specified
258 # above using the 'dbfilename' configuration directive.
259 #
260 # The Append Only File will also be created inside this directory.
261 #
262 # Note that you must specify a directory here, not a file name.
263 dir /opt/redis/data
```

添加从服务器访问主服务器认证

```
283 # If the master is password protected (using the "requirepass" configuration
284 # directive below) it is possible to tell the slave to authenticate before
285 # starting the replication synchronization process, otherwise the master will
286 # refuse the slave request.
287 #
288 # masterauth <master-password>
289 masterauth root
```

添加访问认证

```
496 # Warning: since Redis is pretty fast an outside user can try up to
497 # 150k passwords per second against a good box. This means that you should
498 # use a very strong password otherwise it will be very easy to break.
499 #
500 requirepass root
```

注释公共配置追加文件

根据需求配置是否打开追加文件选项

`appendonly yes` -> 每当Redis执行一个改变数据集的命令时 (比如 SET), 这个命令就会被追加到 AOF 文件的末尾。这样的话, 当Redis重新启时, 程序就可以通过重新执行 AOF文件中的命令来达到重建数据集的目的

```
667 # AOF and RDB persistence can be enabled at the same time without problems.
668 # If the AOF is enabled on startup Redis will load the AOF, that is the file
669 # with the better durability guarantees.
670 #
671 # Please check http://redis.io/topics/persistence for more information.
672
673 appendonly no
```

`appendfilename`和`dir`组合使用, 找`dir(/opt/redis/data)`路径生成数据文件

```
674 # The name of the append only file (default: "appendonly.aof")
675
676 #appendfilename "appendonly.aof"
```

从服务器默认是只读不允许写操作(不用修改)

```
312 # Note: read only slaves are not designed to be exposed to untrusted clients
313 # on the internet. It's just a protection layer against misuse of the instance.
314 # Still a read only slave exports by default all the administrative commands
315 # such as CONFIG, DEBUG, and so forth. To a limited extent you can improve
316 # security of read only slaves using 'rename-command' to shadow all the
317 # administrative / dangerous commands.
318 slave-read-only yes
```

添加3个服务的私有配置文件

`touch` 或者 `vi` 都可以创建空白文件

`touch` 直接创建空白文件, `vi` 创建并且进入编辑模式, `:wq` 创建成功, 否则不创建

```
[root@localhost conf]# pwd
/opt/redis/conf
[root@localhost conf]# ll
总用量 60
-rw-r--r--. 1 root root 58785 8月  2 17:42 redis-common.conf
[root@localhost conf]# touch redis-6379.conf
[root@localhost conf]# touch redis-6380.conf
[root@localhost conf]# touch redis-6381.conf
[root@localhost conf]# ll
总用量 60
-rw-r--r--. 1 root root    0 8月  2 17:51 redis-6379.conf
-rw-r--r--. 1 root root    0 8月  2 17:51 redis-6380.conf
-rw-r--r--. 1 root root    0 8月  2 17:51 redis-6381.conf
-rw-r--r--. 1 root root 58785 8月  2 17:42 redis-common.conf
```

redis-6379.conf

主备切换

主从节点redis.conf配置

参照 读写分离 的相应配置

修改sentinel-common.conf 哨兵公共配置文件

从redis解压目录下复制sentinel.conf至/opt/redis/conf/

```
cp sentinel.conf /opt/redis/conf/sentinel-common.conf
```

```
[root@localhost conf]# pwd
/opt/redis/conf
[root@localhost conf]# ll
总用量 80
-rw-r--r--. 1 root root 309 8月 2 18:01 redis-6379.conf
-rw-r--r--. 1 root root 440 8月 3 12:10 redis-6380.conf
-rw-r--r--. 1 root root 440 8月 3 12:08 redis-6381.conf
-rw-r--r--. 1 root root 58795 8月 2 18:58 redis-common.conf
-rw-r--r--. 1 root root 7606 8月 3 13:07 sentinel-common.conf
```

注释哨兵监听进程端口号

```
19 # port <sentinel-port>
20 # The port that this sentinel instance will run on
21 #port 26379
```

指示 Sentinel 去监视一个名为 master 的主服务器，这个主服务器的IP地址为 127.0.0.1，端口号为 6379，而将这个主服务器判断为失效至少需要1个(一般设置为2个)。Sentinel 同意（只要同意 Sentinel 的数量不达标，自动故障迁移就不会执行）。

这个要配局域网IP，否则远程连不上。

```
67 # Note: master name should not include special characters or spaces.
68 # The valid charset is A-z 0-9 and the three characters "-_.".
69 sentinel monitor mymaster 192.168.10.100 6379 2
```

设置master和slaves的密码

```
88 # sentinel auth-pass mymaster MySUPER--secret-0123passw0rd
89 sentinel auth-pass mymaster root
```

Sentinel 认为服务器已经断线所需的毫秒数


```
98 # Default is 30 seconds.
99 sentinel down-after-milliseconds mymaster 10000
```

若 sentinel 在该配置值内未能完成 failover 操作（即故障时 master/slave 自动切换），则认为本次 failover 失败。

```
130 # Default is 3 minutes.
131 sentinel failover-timeout mymaster 180000
```

关闭保护模式，修改为no

```
198 #关闭保护模式
199 protected-mode no
```

修改为后台启动

```
201 #修改为后台启动
202 daemonize yes
```

添加3个哨兵的私有配置文件

touch 或者 vi 都可以创建空白文件

touch 直接创建空白文件，vi 创建并且进入编辑模式，:wq 创建成功，否则不创建

```
[root@localhost conf]# pwd
/opt/redis/conf
[root@localhost conf]# ll
总用量 80
-rw-r--r--. 1 root root 309 8月 2 18:01 redis-6379.conf
-rw-r--r--. 1 root root 440 8月 3 12:10 redis-6380.conf
-rw-r--r--. 1 root root 440 8月 3 12:08 redis-6381.conf
-rw-r--r--. 1 root root 58795 8月 2 18:58 redis-common.conf
-rw-r--r--. 1 root root 7722 8月 3 13:20 sentinel-common.conf
[root@localhost conf]# touch sentinel-26379.conf
[root@localhost conf]# touch sentinel-26380.conf
[root@localhost conf]# touch sentinel-26381.conf
[root@localhost conf]# ll
总用量 80
-rw-r--r--. 1 root root 309 8月 2 18:01 redis-6379.conf
-rw-r--r--. 1 root root 440 8月 3 12:10 redis-6380.conf
-rw-r--r--. 1 root root 440 8月 3 12:08 redis-6381.conf
-rw-r--r--. 1 root root 58795 8月 2 18:58 redis-common.conf
-rw-r--r--. 1 root root 0 8月 3 13:24 sentinel-26379.conf
-rw-r--r--. 1 root root 0 8月 3 13:24 sentinel-26380.conf
-rw-r--r--. 1 root root 0 8月 3 13:24 sentinel-26381.conf
-rw-r--r--. 1 root root 7722 8月 3 13:20 sentinel-common.conf
```

sentinel-26379.conf

```
#引用公共配置
include /opt/redis/conf/sentinel-common.conf
#进程端口号
port 26379
#进程编号记录文件
pidfile /var/run/sentinel-26379.pid
#日志记录文件(为了方便查看日志,先注释掉,搭好环境后再打开)
logfile "/opt/redis/log/sentinel-26379.log"
```

复制 sentinel-26379.conf 的内容至 sentinel-26380.conf , sentinel-26381.conf 并且修改其内容, 将26379替换即可。

启动测试

启动3个redis服务

```
/usr/local/redis/bin/redis-server /opt/redis/conf/redis-6379.conf
/usr/local/redis/bin/redis-server /opt/redis/conf/redis-6380.conf
/usr/local/redis/bin/redis-server /opt/redis/conf/redis-6381.conf
```

```
[root@localhost ~]# /usr/local/redis/bin/redis-server /opt/redis/conf/redis-6379.conf
[root@localhost ~]# /usr/local/redis/bin/redis-server /opt/redis/conf/redis-6380.conf
[root@localhost ~]# /usr/local/redis/bin/redis-server /opt/redis/conf/redis-6381.conf
[root@localhost ~]# ps -ef | grep redis
root      2458      1   0 10:07 ?        00:00:00 /usr/local/redis/bin/redis-server *:6379
root      2463      1   0 10:07 ?        00:00:00 /usr/local/redis/bin/redis-server *:6380
root      2469      1   0 10:07 ?        00:00:00 /usr/local/redis/bin/redis-server *:6381
```

启动3个哨兵服务

```
/usr/local/redis/bin/redis-sentinel /opt/redis/conf/sentinel-26379.conf
/usr/local/redis/bin/redis-sentinel /opt/redis/conf/sentinel-26380.conf
/usr/local/redis/bin/redis-sentinel /opt/redis/conf/sentinel-26381.conf
```

```
[root@localhost ~]# ps -ef | grep redis
root      2458      1   0 10:07 ?        00:00:01 /usr/local/redis/bin/redis-server *:6379
root      2463      1   0 10:07 ?        00:00:01 /usr/local/redis/bin/redis-server *:6380
root      2469      1   0 10:07 ?        00:00:00 /usr/local/redis/bin/redis-server *:6381
root      2626    2576   0 10:11 pts/3    00:00:01 /usr/local/redis/bin/redis-sentinel *:26379 [sentinel]
root      2630    2539   0 10:11 pts/2    00:00:01 /usr/local/redis/bin/redis-sentinel *:26380 [sentinel]
root      2634    2497   0 10:11 pts/1    00:00:01 /usr/local/redis/bin/redis-sentinel *:26381 [sentinel]
```

查看主从状态

redis-6379


```
[root@localhost ~]# ps -ef | grep redis
root      2492      1  0 13:36 ?        00:00:00 /usr/local/redis/bin/redis-server *:6379
root      2497      1  0 13:36 ?        00:00:00 /usr/local/redis/bin/redis-server *:6380
root      2503      1  0 13:36 ?        00:00:00 /usr/local/redis/bin/redis-server *:6381
root      2698    2591  0 13:37 pts/3    00:00:00 /usr/local/redis/bin/redis-sentinel *:26379 [sentinel]
root      2702    2628  0 13:37 pts/4    00:00:00 /usr/local/redis/bin/redis-sentinel *:26380 [sentinel]
root      2706    2665  0 13:37 pts/5    00:00:00 /usr/local/redis/bin/redis-sentinel *:26381 [sentinel]
root      2711    2512  0 13:39 pts/1    00:00:00 /usr/local/redis/bin/redis-cli -p 6380 -a root
root      2718    2554  0 13:40 pts/2    00:00:00 /usr/local/redis/bin/redis-cli -p 6381 -a root
root      2720    2205  0 13:40 pts/0    00:00:00 grep --color=auto redis
[root@localhost ~]# kill -9 2492
```

已选举6380为主节点，从节点目前只有6381

```
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.10.100,port=6381,state=online,offset=45141,lag=1
master_replid:c46379a5770d36c04b3e4ea259928ed80388a44b
master_replid2:4a4d9b056b348c31ffe2231486893543518332a6
master_repl_offset:45141
second_repl_offset:42388
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:45141
```

重新启动6379节点，再次查看主从状态

发现6379已被发现且成为从节点

```
127.0.0.1:6380> info replication
# Replication
role:master
connected_slaves:2
slave0:ip=192.168.10.100,port=6381,state=online,offset=72802,lag=1
slave1:ip=192.168.10.100,port=6379,state=online,offset=72802,lag=0
master_replid:c46379a5770d36c04b3e4ea259928ed80388a44b
master_replid2:4a4d9b056b348c31ffe2231486893543518332a6
master_repl_offset:72802
second_repl_offset:42388
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:72802
```

6380之前不可以写操作，现在可以写操作，因为已成为主节点。

最后，公共配置文件修改为后台启动，私有配置文件打开日志记录文件，环境搭建成功。

SpringDataRedis

创建项目

Java

Java Enterprise

JBoss

Clouds

Spring

Java FX

Android

IntelliJ Platform Plugin

Spring Initializr

Maven

Gradle

Groovy

Grails

Application Forge

Kotlin

Static Web

Node.js and NPM

Flash

Empty Project

Project SDK: 1.8 (java version "1.8.0_231")

New...

Choose Initializr Service URL

☒ Default: <https://start.spring.io>

☐ Custom:

Make sure your network connection is active before continuing.

PreviousNextCancelHelp

Project Metadata

Group: com.xxxx

Artifact: springdataredis-demo

Type: Maven Project (Generate a Maven based project archive.)

Language: Java

Packaging: Jar

Java Version: 8

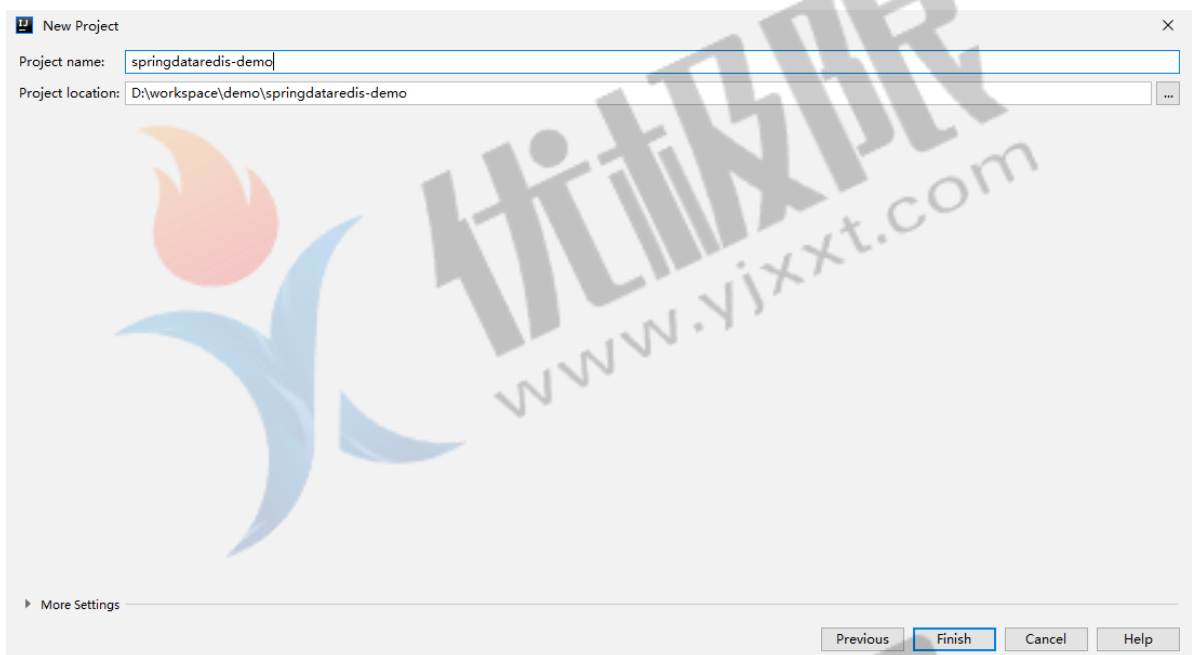
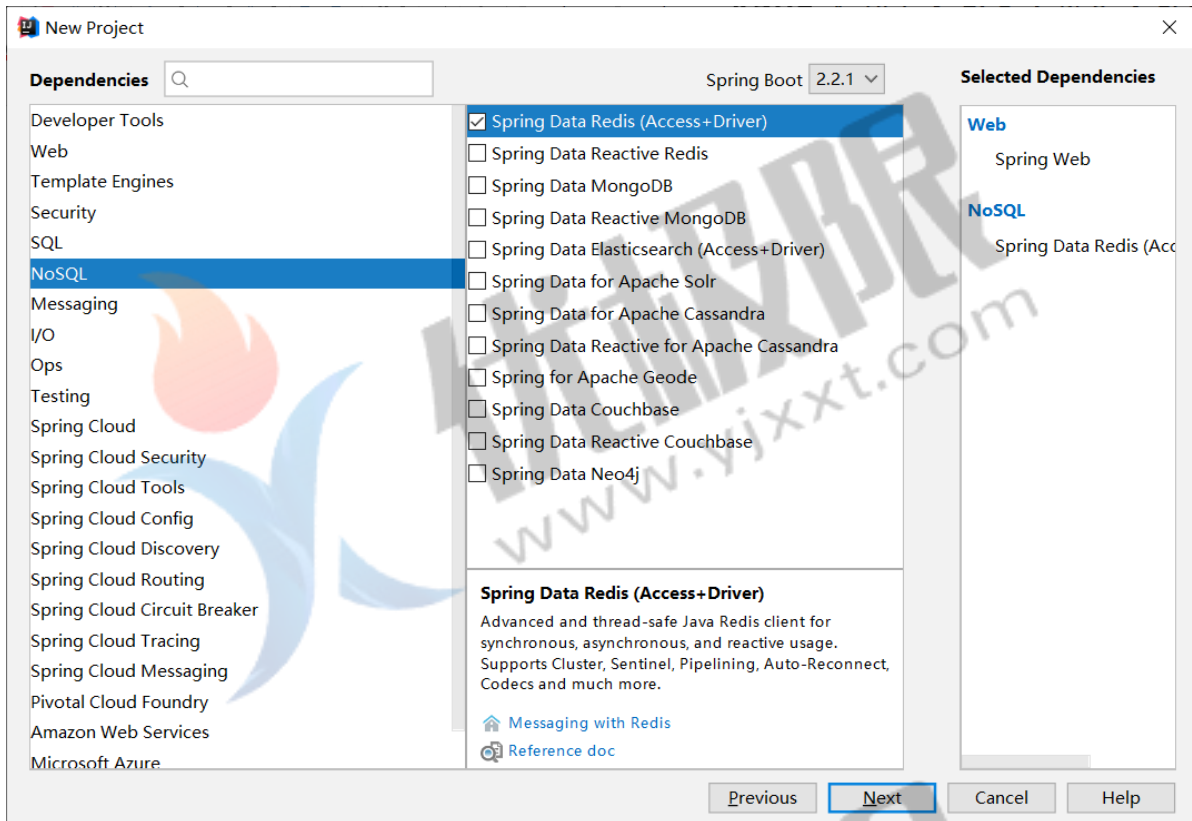
Version: 0.0.1-SNAPSHOT

Name: springdataredis-demo

Description: Demo project for Spring Boot

Package: com.xxxx.springdataredisdemo

PreviousNextCancelHelp



添加依赖

```
<dependencies>
  <!-- spring data redis 组件 -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
  </dependency>
  <!-- commons-pool2 对象池依赖 -->
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
  </dependency>
  <!-- web 组件 -->
  <dependency>
```



```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- test 组件 -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
</dependencies>
```

添加application.yml配置文件

```
spring:
  redis:
    # Redis服务器地址
    host: 192.168.10.100
    # Redis服务器端口
    port: 6379
    # Redis服务器端口
    password: root
    # Redis服务器端口
    database: 0
    # 连接超时时间
    timeout: 10000ms
    lettuce:
      pool:
        # 最大连接数，默认8
        max-active: 1024
        # 最大连接阻塞等待时间，单位毫秒，默认-1ms
        max-wait: 10000ms
        # 最大空闲连接，默认8
        max-idle: 200
        # 最小空闲连接，默认0
        min-idle: 5
```

Lettuce和Jedis的区别

Jedis 是一个优秀的基于 Java 语言的 Redis 客户端，但是，其不足也很明显：Jedis 在实现上是直接连接 Redis-Server，在多个线程间共享一个 Jedis 实例时是线程不安全的，如果需要在多线程场景下使用 Jedis，需要使用连接池，每个线程都使用自己的 Jedis 实例，当连接数量增多时，会消耗较多的物理资源。

Lettuce 则完全克服了其线程不安全的缺点：Lettuce 是基于 Netty 的连接 (StatefulRedisConnection) ，

Lettuce 是一个可伸缩的线程安全的 Redis 客户端，支持同步、异步和响应式模式。多个线程可以共享一个连接实例，而不必担心多线程并发问题。它基于优秀 Netty NIO 框架构建，支持 Redis 的高级功能，如 Sentinel，集群，流水线，自动重新连接和 Redis 数据模型。

测试环境测试环境是否搭建成功

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = SpringDataRedisApplication.class)
public class SpringDataRedisApplicationTests {

    @Autowired
    private RedisTemplate redisTemplate;
    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Test
    public void initconn() {
        ValueOperations<String, String> ops = stringRedisTemplate.opsForValue();
        ops.set("username", "lisi");
        ValueOperations<String, String> value = redisTemplate.opsForValue();
        value.set("name", "wangwu");
        System.out.println(ops.get("name"));
    }
}
```

自定义模板解决序列化问题

默认情况下的模板 `RedisTemplate<Object, Object>`，默认序列化使用的是 `JdkSerializationRedisSerializer`，存储二进制字节码。这时需要自定义模板，当自定义模板后又想存储 `String` 字符串时，可以使 `StringRedisTemplate` 的方式，他们俩并不冲突。

序列化问题：

要把 domain object 做为 key-value 对保存在 redis 中，就必须解决对象的序列化问题。Spring Data Redis 给我们提供了一些现成的方案：

`JdkSerializationRedisSerializer` 使用 JDK 提供的序列化功能。优点是反序列化时不需要提供类型信息(class)，但缺点是序列化后的结果非常庞大，是JSON格式的5倍左右，这样就会消耗 Redis 服务器的大量内存。

`Jackson2JsonRedisSerializer` 使用 Jackson 库将对象序列化为JSON字符串。优点是速度快，序列化后的字符串短小精悍。但缺点也非常致命，那就是此类的构造函数中有一个类型参数，必须提供要序列化对象的类型信息(class对象)。通过查看源代码，发现其只在反序列化过程中用到了类型信息。

`GenericJackson2JsonRedisSerializer` 通用型序列化，这种序列化方式不用自己手动指定对象的Class。

```
@Configuration
public class RedisConfig {
    @Bean
    public RedisTemplate<String, Object> redisTemplate(LettuceConnectionFactory
redisConnectionFactory){
        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();
        //为string类型key设置序列化器
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        //为string类型value设置序列化器
        redisTemplate.setValueSerializer(new
GenericJackson2JsonRedisSerializer());
        //为hash类型key设置序列化器
    }
}
```

```

        redisTemplate.setHashKeySerializer(new StringRedisSerializer());
        //为hash类型value设置序列化器
        redisTemplate.setHashValueSerializer(new
GenericJackson2JsonRedisSerializer());
        redisTemplate.setConnectionFactory(redisConnectionFactory);
        return redisTemplate;
    }
}

```

```

//序列化
@Test
public void testSerial(){
    User user = new User();
    user.setId(1);
    user.setUsername("张三");
    user.setPassword("111");
    ValueOperations<String, Object> value = redisTemplate.opsForValue();
    value.set("userInfo",user);
    System.out.println(value.get("userInfo"));
}

```

操作string

```

// 1.操作String
@Test
public void testString() {
    ValueOperations<String, Object> valueOperations =
redisTemplate.opsForValue();

    // 添加一条数据
    valueOperations.set("username", "zhangsan");
    valueOperations.set("age", "18");

    // redis中以层级关系、目录形式存储数据
    valueOperations.set("user:01", "lisi");
    valueOperations.set("user:02", "wangwu");

    // 添加多条数据
    Map<String, String> userMap = new HashMap<>();
    userMap.put("address", "bj");
    userMap.put("sex", "1");
    valueOperations.multiSet(userMap);

    // 获取一条数据
    Object username = valueOperations.get("username");
    System.out.println(username);

    // 获取多条数据
    List<String> keys = new ArrayList<>();
    keys.add("username");
    keys.add("age");
}

```

```

keys.add("address");
keys.add("sex");
List<Object> resultList = valueOperations.multiGet(keys);
for (Object str : resultList) {
    System.out.println(str);
}

// 删除
redisTemplate.delete("username");
}

```

操作hash

```

// 2. 操作Hash
@Test
public void testHash() {
    HashOperations<String, String, String> hashOperations =
        redisTemplate.opsForHash();

    /*
     * 添加一条数据
     * 参数一: redis的key
     * 参数二: hash的key
     * 参数三: hash的value
     */
    hashOperations.put("userInfo", "name", "lisi");

    // 添加多条数据
    Map<String, String> map = new HashMap();
    map.put("age", "20");
    map.put("sex", "1");
    hashOperations.putAll("userInfo", map);

    // 获取一条数据
    String name = hashOperations.get("userInfo", "name");
    System.out.println(name);

    // 获取多条数据
    List<String> keys = new ArrayList<>();
    keys.add("age");
    keys.add("sex");
    List<String> resultList = hashOperations.multiGet("userInfo", keys);
    for (String str : resultList) {
        System.out.println(str);
    }

    // 获取Hash类型所有的数据
    Map<String, String> userMap = hashOperations.entries("userInfo");
    for (Entry<String, String> userInfo : userMap.entrySet()) {
        System.out.println(userInfo.getKey() + "--" + userInfo.getValue());
    }

    // 删除 用于删除hash类型数据
    hashOperations.delete("userInfo", "name");
}

```

操作list

```
// 3.操作list
@Test
public void testList() {
    ListOperations<String, Object> listOperations = redisTemplate.opsForList();

    // 左添加(上)
    //      listOperations.leftPush("students", "Wang Wu");
    //      listOperations.leftPush("students", "Li Si");

    // 左添加(上) 把value值放到key对应列表中pivot值的左面, 如果pivot值存在的话
    //listOperations.leftPush("students", "Wang Wu", "Li Si");

    // 右添加(下)
    //      listOperations.rightPush("students", "Zhao Liu");

    // 获取 start起始下标 end结束下标 包含关系
    List<Object> students = listOperations.range("students", 0,2);
    for (Object stu : students) {
        System.out.println(stu);
    }

    // 根据下标获取
    Object stu = listOperations.index("students", 1);
    System.out.println(stu);

    // 获取总条数
    Long total = listOperations.size("students");
    System.out.println("总条数: " + total);

    // 删除单条 删除列表中存储的列表中几个出现的Li Si。
    listOperations.remove("students", 1, "Li Si");

    // 删除多条
    redisTemplate.delete("students");
}
```

操作set

```
// 4.操作set-无序
@Test
public void testSet() {
    SetOperations<String, Object> setOperations = redisTemplate.opsForSet();
    // 添加数据
    String[] letters = new String[]{"aaa", "bbb", "ccc", "ddd", "eee"};
    //setOperations.add("letters", "aaa", "bbb", "ccc", "ddd", "eee");
    setOperations.add("letters", letters);

    // 获取数据
    Set<Object> let = setOperations.members("letters");
    for (Object letter: let) {
        System.out.println(letter);
    }

    // 删除
```



```
setOperations.remove("letters", "aaa", "bbb");  
}
```

操作sorted set

```
// 5.操作sorted set-有序  
@Test  
public void testSortedSet() {  
    ZSetOperations<String, Object> zSetOperations = redisTemplate.opsForZSet();  
  
    ZSetOperations.TypedTuple<Object> objectTypedTuple1 =  
        new DefaultTypedTuple<Object>("zhangsan", 7D);  
    ZSetOperations.TypedTuple<Object> objectTypedTuple2 =  
        new DefaultTypedTuple<Object>("lisi", 3D);  
    ZSetOperations.TypedTuple<Object> objectTypedTuple3 =  
        new DefaultTypedTuple<Object>("wangwu", 5D);  
    ZSetOperations.TypedTuple<Object> objectTypedTuple4 =  
        new DefaultTypedTuple<Object>("zhaoliu", 6D);  
    ZSetOperations.TypedTuple<Object> objectTypedTuple5 =  
        new DefaultTypedTuple<Object>("tianqi", 2D);  
    Set<ZSetOperations.TypedTuple<Object>> tuples = new  
    HashSet<ZSetOperations.TypedTuple<Object>>();  
    tuples.add(objectTypedTuple1);  
    tuples.add(objectTypedTuple2);  
    tuples.add(objectTypedTuple3);  
    tuples.add(objectTypedTuple4);  
    tuples.add(objectTypedTuple5);  
  
    // 添加数据  
    zSetOperations.add("score", tuples);  
  
    // 获取数据  
    Set<Object> scores = zSetOperations.range("score", 0, 4);  
    for (Object score: scores) {  
        System.out.println(score);  
    }  
  
    // 获取总条数  
    Long total = zSetOperations.size("score");  
    System.out.println("总条数: " + total);  
  
    // 删除  
    zSetOperations.remove("score", "zhangsan", "lisi");  
}
```

获取所有key&删除

```
// 获取所有key  
@Test  
public void testAllKeys() {  
    // 当前库key的名称  
    Set<String> keys = redisTemplate.keys("*");  
    for (String key: keys) {  
        System.out.println(key);  
    }  
}
```



```
// 删除
@Test
public void testDelete() {
    // 删除 通用 适用于所有数据类型
    redisTemplate.delete("score");
}
```

设置key的失效时间

```
@Test
public void testEx() {
    ValueOperations<String, Object> valueOperations =
    redisTemplate.opsForValue();
    // 方法一：插入一条数据并设置失效时间
    valueOperations.set("code", "abcd", 180, TimeUnit.SECONDS);
    // 方法二：给已存在的key设置失效时间
    boolean flag = redisTemplate.expire("code", 180, TimeUnit.SECONDS);
    // 获取指定key的失效时间
    Long l = redisTemplate.getExpire("code");
}
```

```
@Test
public void testEx() {
    ValueOperations<String, Object> valueOperations = redisTemplate.opsForValue();
    // 插入一条数据
    valueOperations.set( k: "code", v: "abcd", l: 180, TimeUnit.);
}
```

TimeUnit	
HOURS	TimeUnit
SECONDS	TimeUnit
DAYS	TimeUnit
valueOf(String name)	TimeUnit
MICROSECONDS	TimeUnit
MILLISECONDS	TimeUnit
MINUTES	TimeUnit
NANOSECONDS	TimeUnit
values()	TimeUnit[]
valueOf(Class<T> enumType, String name)	T
class	

Dot, space and some other keys will also close this lookup and be inserted into editor >>

SpringDataRedis整合使用哨兵机制

application.yml

```
spring:
  redis:
    # Redis服务器地址
    host: 192.168.10.100
    # Redis服务器端口
    port: 6379
    # Redis服务器端口
    password: root
    # Redis服务器端口
    database: 0
    # 连接超时时间
```

```
timeout: 10000ms
lettuce:
  pool:
    # 最大连接数, 默认8
    max-active: 1024
    # 最大连接阻塞等待时间, 单位毫秒, 默认-1ms
    max-wait: 10000ms
    # 最大空闲连接, 默认8
    max-idle: 200
    # 最小空闲连接, 默认0
    min-idle: 5
  #哨兵模式
  sentinel:
    #主节点名称
    master: mymaster
    #节点
    nodes:
      192.168.10.100:26379,192.168.10.100:26380,192.168.10.100:26381
```

Bean注解配置

```
@Bean
public RedisSentinelConfiguration redisSentinelConfiguration(){
    RedisSentinelConfiguration sentinelConfig = new RedisSentinelConfiguration()
        // 主节点名称
        .master("mymaster")
        // 主从服务器地址
        .sentinel("192.168.10.100", 26379)
        .sentinel("192.168.10.100", 26380)
        .sentinel("192.168.10.100", 26381);
    // 设置密码
    sentinelConfig.setPassword("root");
    return sentinelConfig;
}
```

如何应对缓存穿透、缓存击穿、缓存雪崩问题

Key的过期淘汰机制

Redis可以对存储在Redis中的缓存数据设置过期时间, 比如我们获取的短信验证码一般十分钟过期, 我们这时候就需要在验证码存进Redis时添加一个key的过期时间, 但是这里有一个需要格外注意的问题就是: 并非key过期时间到了就一定会被Redis给删除。

定期删除

Redis 默认是每隔 100ms 就随机抽取一些设置了过期时间的 Key, 检查其是否过期, 如果过期就删除。为什么是随机抽取而不是检查所有key? 因为你如果设置的key成千上万, 每100毫秒都将所有存在的key检查一遍, 会给CPU带来比较大的压力。

惰性删除

定期删除由于是随机抽取可能会导致很多过期 Key 到了过期时间并没有被删除。所以用户在从缓存获取数据的时候, redis会检查这个key是否过期了, 如果过期就删除这个key。这时候就会在查询的时候将过期key从缓存中清除。

内存淘汰机制

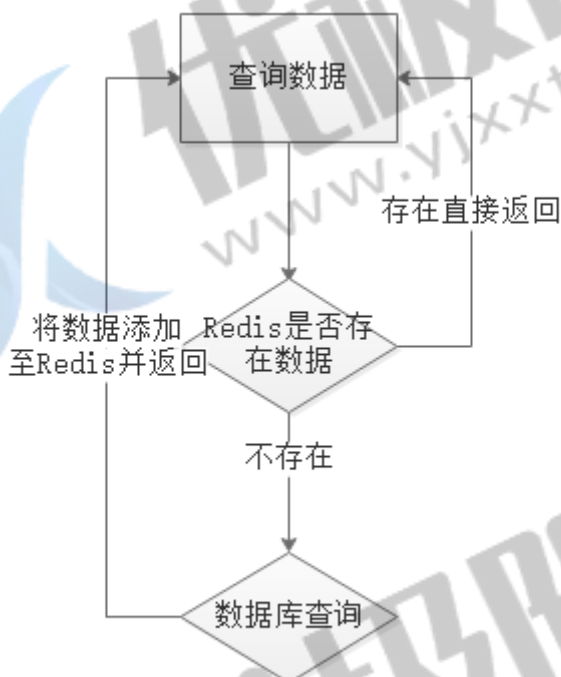
仅仅使用定期删除 + 惰性删除机制还是会留下一个严重的隐患：如果定期删除留下了很多已经过期的key，而且用户长时间都没有使用过这些过期key，导致过期key无法被惰性删除，从而导致过期key一直堆积在内存里，最终造成Redis内存块被消耗殆尽。那这个问题如何解决呢？这个时候Redis内存淘汰机制应运而生了。Redis内存淘汰机制提供了6种数据淘汰策略：

- `volatile-lru`：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰。
- `volatile-ttl`：从已设置过期时间的数据集中挑选将要过期的数据淘汰。
- `volatile-random`：从已设置过期时间的数据集中任意选择数据淘汰。
- `allkeys-lru`：当内存不足以容纳新写入数据时移除最近最少使用的key。
- `allkeys-random`：从数据集中任意选择数据淘汰。
- `no-eviction`（默认）：当内存不足以容纳新写入数据时，新写入操作会报错。

一般情况下，推荐使用 `volatile-lru` 策略，对于配置信息等重要数据，不应该设置过期时间，这样Redis就永远不会淘汰这些重要数据。对于一般数据可以添加一个缓存时间，当数据失效则请求会从DB中获取并重新存入Redis中。

缓存击穿

首先我们来看下请求是如何取到数据的：当接收到用户请求，首先先尝试从Redis缓存中获取到数据，如果缓存中能取到数据则直接返回结果，当缓存中不存在数据时从DB获取数据，如果数据库成功取到数据，则更新Redis，然后返回数据



定义：高并发的情况下，某个热门key突然过期，导致大量请求在Redis未找到缓存数据，进而全部去访问DB请求数据，引起DB压力瞬间增大。

解决方案：缓存击穿的情况下一般不容易造成DB的宕机，只是会造成对DB的周期性压力。对缓存击穿的解决方案一般可以这样：

- Redis中的数据不设置过期时间，然后在缓存的对象上添加一个属性标识过期时间，每次获取到数据时，校验对象中的过期时间属性，如果数据即将过期，则异步发起一个线程主动更新缓存中的数

据。但是这种方案可能会导致有些请求会拿到过期的值，就得看业务能否可以接受，

- 如果要求数据必须是新数据，则最好的方案则为热点数据设置为永不过期，然后加一个互斥锁保证缓存的单线程写。

缓存穿透

定义：缓存穿透是指查询缓存和DB中都不存在的数据。比如通过id查询商品信息，id一般大于0，攻击者会故意传id为-1去查询，由于缓存是不命中则从DB中获取数据，这将会导致每次缓存都不命中数据导致每个请求都访问DB，造成缓存穿透。

解决方案：

- 利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试
- 采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。
- 提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。
- 如果从数据库查询的对象为空，也放入缓存，只是设定的缓存过期时间较短，比如设置为60秒。

缓存雪崩

定义：缓存中如果大量缓存在一段时间内集中过期了，这时候会发生大量的缓存击穿现象，所有的请求都落在了DB上，由于查询数据量巨大，引起DB压力过大甚至导致DB宕机。

解决方案：

- 给缓存的失效时间，加上一个随机值，避免集体失效。如果Redis是集群部署，将热点数据均匀分布在不同的Redis库中也能避免全部失效的问题
- 使用互斥锁，但是该方案吞吐量明显下降了。
- 设置热点数据永远不过期。
- 双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点
 1. 从缓存A读数据库，有则直接返回
 2. A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程。
 3. 更新线程同时更新缓存A和缓存B。