**Redis知识总结**

### 1.基础简介

#### 1.1 什么是redis

redis是一个Key-value存储系统。支持的value类型很多，包括string，list，set和zset。

为了保证效率，数据都是缓存在内存中。redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave主从同步。

#### 1.2 性能怎么样？

因为是内存数据库，性能较高，set操作每秒110000次，get每秒81000次。

#### 1.3 单台redis的存放数据必须比物理内存小

数据存储在内存带了高性能，但是也带来了不合理之处。数据并非全部都是活跃的，比如100万注册用户，只有5万活跃用户，一周访问一次的也只有15万次。如果这100万用户数据都放在内存中，RAM就被冷数据给浪费了。

在现在操作系统中，所有应用访问的数据都在内存中，但是数据量超出物理内存后，操作系统会智能把长期没有访问的数据交换到磁盘，为新的应用留出空间。因此现在操作系统给应用提供的并不是物理内存，而是虚拟内存（Virtual Memory）的概念。

基于相同考虑，Redis2中也增加了VM特性，让Redis数据突破了物理内存的限制。

#### 1.4 redis的VM是不是多余的？

既然操作系统已经实现虚拟内存  
 实际上确实有人利用系统自带的内存机制来处理冷热数据，也有比较成功的效果。

但是redis的作者antirez曾经总结自己为什么要实现VM的原因：第一是系统的vm机制粒度比较大，拥有一个page概念，比如一个page是4k，那么只要这4k数据中一个元素没有被访问，这也page也 不会被SWAP(释放到交换分区)。因此最大可能产生4k的无用内存，redis自己实现可以控制粒度大小。

而且，访问操作系统SWAP内存区域时block（阻塞）进程，造成不便。

#### 1.5 用什么方式来使用Redis？

redis是一个k-v存在，我们自然会想到使用set/get的方式来使用它，但是这并不是最好的方法。对于redis来说，节约内存是一个很重要的命题。

我们在使用它的时候，应该想尽办法来复用key，通过把几个key-value放入一个key中，value再作为一个set存入，这样同样512字节就会存放10-100倍的容量。

为了节约内存，建议使用hashset。

#### 1.6 redis的存储方式

Redis有两种存储方式，默认是snapshot（拍快照）方式，实现的方法是定时将内存的快照持久化到硬盘。缺点是持久化之后如果出现crash（宕机），就会丢失一段数据。

因此，作者增加了aof方式。aof=append only mode ，在吸入内存数据的同时将操作命令保存到日志文件，在一个并发更改上万的系统中，命令日志是一个非常庞大的数据，管理维护成本非常高，恢复重建时间会非常长，这样导致失去aof高可用性本意。

另外，redsi是一个内存数据结构模型，所有优势都是建立在对内存复杂数据结构高效原子操作上，aof是一个非常不协调的部分。

其实，aof主要实现数据可靠性以及高可用性。

在redis中有一个：Replication，由于复制基本没有延迟，这样达到了防止单点故障及实现了高可用。

#### 1.7 Redis如何支持集群

redis主从复制配置和使用都非常简单，通过主从复制可以允许多个slave server用用和master server相同的数据库副本。

主从复制的特点：

1.ster 可以拥有多个slave

2.slave不仅连通master，本身也和其他slave相连。

3.主从复制不会阻塞master

4.主从复制可以用来提高系统的可伸缩性。

5.可以用master禁用数据持久化，只需要注释掉master配置文件中所有slave配置，然后在slave上配置数据持久化。

主从复制过程：

当设置好slave服务器后，slave会建立和master的连接，然后发送sync命令。无论是第一次同步建立的连接还是连接断开后的重新连 接，master都会启动一个后台进程，将数据库快照保存到文件中，同时master主进程会开始收集新的写命令并缓存起来。后台进程完成写文件 后，master就发送文件给slave，slave将文件保存到磁盘上，然后加载到内存恢复数据库快照到slave上。接着master就会把缓存的命 令转发给slave。而且后续master收到的写命令都会通过开始建立的连接发送给slave。从master到slave的同步数据的命令和从 client发送的命令使用相同的协议格式。当master和slave的连接断开时slave可以自动重新建立连接。如果master同时收到多个 slave发来的同步连接命令，只会使用启动一个进程来写数据库镜像，然后发送给所有slave。

#### 1.8 简单实现redis的主从复制

假设有两台服务器，一台是windows系统（IP：192.168.3.82），一台是Linux操作系统（IP：192.168.3.90），在两个系统中都安装redis，window系统用cygwin（类UNIX模拟环境）工具安装。

命令为：

view sourceprint?

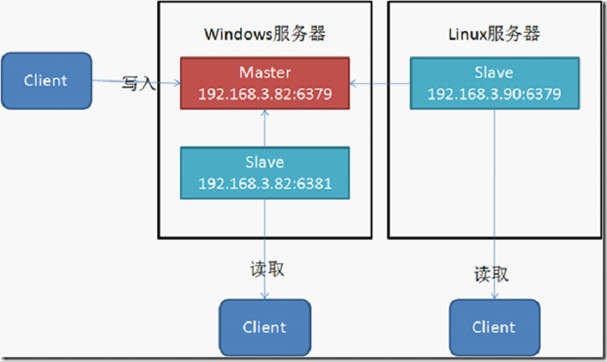
$ tar xzf redis-2.2.2.tar.gz

$ cd redis-2.2.2

$ make

可以通过”make test”命令判断是否安装成功。

这里我使用1个master以及2个slave（master在Windows下，一个slave在Windows下，一个slave在Linux下），基本流程是：



1. 在Windows服务器上创建两个目录，Demo1，Demo2，其中Demo1用来存放Master服务，Demo2用来存放Slave服务，

在Master服务中的配置文件修改：

view sourceprint?

bind 192.168.3.82

在Slave服务中的配置文件修改：

view sourceprint?

port 6381（服务端口号要分开）

bind 192.168.3.82

slaveof 192.168.3.82 6379 （设置master的Host以及Port）

2. 在Linux服务器上创建一个目录，Demo，Demo存放Slave服务，在服务中的配置文件修改：

view sourceprint?

bind 192.168.3.90

slaveof 192.168.3.82 6379（设置master的Host以及Port）

这样就完成了所有的配置。

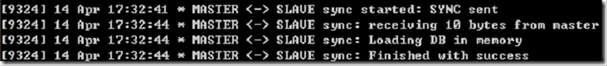
3. 现在运行这3个服务，通过命令：

view sourceprint?

./redis-server redis.conf

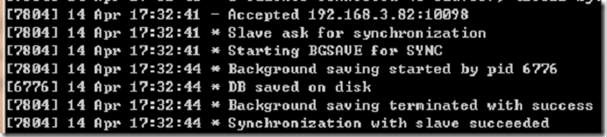
来启动redis服务。

注意到，当我启动master，然后启动一个slave的时候，可以发现slave上：



会发送一个SYNC请求，从Master上面进行相应，而且它支持自动重连，即当master掉线的情况下，它会处于等待请求的状态。

而Master上：



能够接受Slave的应答，并且开始持久化操作，说明在Slave每次去连接Master的时候，都会去持久化磁盘。

4. 现在开始写一个客户端程序，使用到ServiceStack.Redis.dll的.NET组件：

view sourceprint?

using ServiceStack.Redis;

static void Main(string[] args)

{

IRedisClientFactory factory = new RedisCacheClientFactory();

IRedisClient client = factory.CreateRedisClient(“192.168.3.82″, 6379);

client.Set<STRING>(“username”, “leepy”);

string username = client.Get<STRING>(“username”);

client.Save();

Console.WriteLine(“username: {0}”, username);

Console.ReadLine();

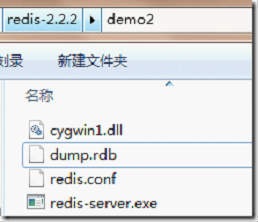
}

运行结果：



数据Set的时候，数据保存在内存中，当调用Save方法时候，将数据保存在磁盘中。

其中你会发现在3个服务目录中，都出现了dump.rdb，说明Master的文件都同步到Slave中去了。







从Redis源码中，可以发现rdb文件采用的是lzf压缩算法进行实现，默认lzf压缩算法是开启的。

#### 1.9 redis安装与应用

下载地址：http://redis.googlecode.com/files/redis-2.2.7.tar.gz

redis的默认端口为6379

#### 1.10 redis事务

redis对事务的支持比较简单，只能保证一个client法器的事务中的命令可以连续执行，而中间不会插入其他client的命令。

由于redis是单线程来处理所有client的请求的所以做到这点是很容易的。一般情况下redis在接受到一个client发来的命令后会立即处理并 返回处理结果，但是当一个client在一个连接中发出multi命令有，这个连接会进入一个事务上下文，该连接后续的命令并不是立即执行，而是先放到一 个队列中。当从此连接受到exec命令后，redis会顺序的执行队列中的所有命令。并将所有命令的运行结果打包到一起返回给client.然后此连接就 结束事务上下文。

一个十分罕见的问题是 当事务的执行过程中，如果redis意外的挂了。很遗憾只有部分命令执行了，后面的也就被丢弃了。当然如果我们使用的append-only file方式持久化，redis会用单个write操作写入整个事务内容。即是是这种方式还是有可能只部分写入了事务到磁盘。发生部分写入事务的情况 下，redis重启时会检测到这种情况，然后失败退出。可以使用redis-check-aof工具进行修复，修复会删除部分写入的事务内容。修复完后就 能够重新启动了。

#### 1.11 redis的持久化

redis是一个支持持久化的内存数据库，也就是说redis需要经常将内存中的数据同步到磁盘来保证持久化。redis支持两种持久化方式，一种是 Snapshotting（快照）也是默认方式，另一种是Append-only file（缩写aof）的方式。下面分别介绍：

Snapshotting  
 快照是默认的持久化方式。这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。可以通过配置设置自动做快照持久 化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置  
 save 900 1 #900秒内如果超过1个key被修改，则发起快照保存  
 save 300 10 #300秒内容如超过10个key被修改，则发起快照保存  
 save 60 10000  
下面介绍详细的快照保存过程  
 1.redis调用fork,现在有了子进程和父进程。  
 2. 父进程继续处理client请求，子进程负责将内存内容写入到临时文件。由于os的写时复制机制（copy on write)父子进程会共享相同的物理页面，当父进程处理写请求时os会为父进程要修改的页面创建副本，而不是写共享的页面。所以子进程的地址空间内的数 据是fork时刻整个数据库的一个快照。  
 3.当子进程将快照写入临时文件完毕后，用临时文件替换原来的快照文件，然后子进程退出。  
 client 也可以使用save或者bgsave命令通知redis做一次快照持久化。save操作是在主线程中保存快照的，由于redis是用一个主线程来处理所有 client的请求，这种方式会阻塞所有client请求。所以不推荐使用。另一点需要注意的是，每次快照持久化都是将内存数据完整写入到磁盘一次，并不 是增量的只同步脏数据。如果数据量大的话，而且写操作比较多，必然会引起大量的磁盘io操作，可能会严重影响性能。  
 另外由于快照方式是在一定间隔时间做一次的，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改。如果应用要求不能丢失任何修改的话，可以采用aof持久化方式。下面介绍  
 Append-only file  
 aof 比快照方式有更好的持久化性，是由于在使用aof持久化方式时,redis会将每一个收到的写命令都通过write函数追加到文件中(默认是 appendonly.aof)。

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉redis我们想要 通过fsync函数强制os写入到磁盘的时机。 有三种方式如下（默认是：每秒fsync一次）  
 appendonly yes //启用aof持久化方式  
 # appendfsync always //每次收到写命令就立即强制写入磁盘，最慢的，但是 保证完全的持久化，不推荐使用  
 appendfsync everysec //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐  
 # appendfsync no //完全依赖os，性能最好,持久化没保证  
 aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。收到此命令redis将使用与快照类似的方式将内存中的数据 以命令的方式保存到临时文件中，最后替换原来的文件。具体过程如下  
 1. redis调用fork ，现在有父子两个进程  
 2. 子进程根据内存中的数据库快照，往临时文件中写入重建数据库状态的命令  
 3.父进程继续处理client请求，除了把写命令写入到原来的aof文件中。同时把收到的写命令缓存起来。这样就能保证如果子进程重写失败的话并不会出问题。  
 4.当子进程把快照内容写入已命令方式写到临时文件中后，子进程发信号通知父进程。然后父进程把缓存的写命令也写入到临时文件。  
 5.现在父进程可以使用临时文件替换老的aof文件，并重命名，后面收到的写命令也开始往新的aof文件中追加。  
 需要注意到是重写aof文件的操作，并没有读取旧的aof文件，而是将整个内存中的数据库内容用命令的方式重写了一个新的aof文件,这点和快照有点类似。

### 2.应用问题

#### 2.1 redis与memcache相比的优势

#### 2.2 redis常见性能问题

#### 2.3 如何保证redis中的数据都是热点数据

#### 2.4 redis最适合的场景

### 3.新系统中对Redis应用逻辑

实际上就是spring-boot对redis的整合

#### 3.1 添加依赖

在pom.xml文件添加依赖

<!-- 提供了在srping应用中通过简单的配置访问redis服务 -->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

<version>${spring-data-redis.version}</version>

</dependency>

这个包对redis底层开发包进行了高度封装，提供了如下功能

1，连接池自动管理，提供RedisTemplate类

2，针对jedis或lettuce客户端api进行归类封装，封装为operation接口

<!-- Redis的客户端框架 -->

<dependency>

<groupId>io.lettuce</groupId>

<artifactId>lettuce-core</artifactId>

<version>${lettuce.version}</version>

</dependency>

1，Lettuce是连接Redis server的客户端程序。基于Netty的连接实例，可以在多个线程间并发访问，且线程安全，满足多线程环境的并发访问，同时是可伸缩的设计。

<!-- Redisson是架设在Redis基础上的一个Java驻内存数据存储-->

<dependency>

<groupId>org.redisson</groupId>

<artifactId>redisson</artifactId>

<version>${redisson.version}</version>

</dependency>

redission能够实现的功能很多，最重要的就是实现分布式锁

<!-- netty -->

netty是一个广泛应用的java网络编程框架，特点是：并发高，传输快，封装好。

提供异步，事件驱动的网络应用程序框架和工具。

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-common</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-codec</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-buffer</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-transport</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-handler</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

<groupId>io.netty</groupId>

<artifactId>netty-resolver-dns</artifactId>

<version>${netty-version}</version>

</dependency>

<dependency>

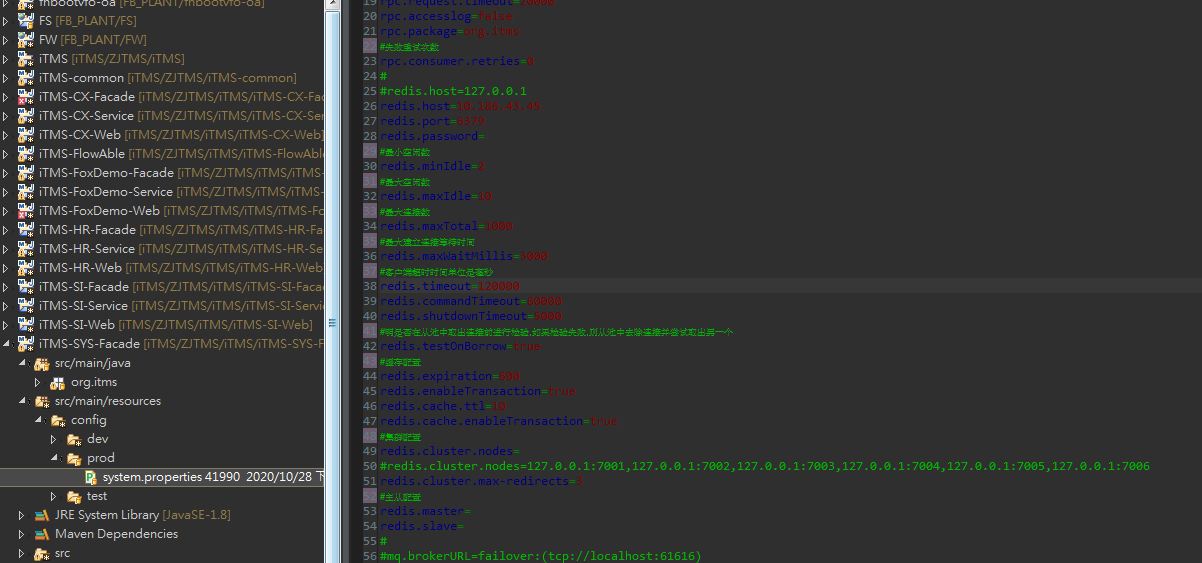
<groupId>io.netty</groupId>

<artifactId>netty-codec-dns</artifactId>

<version>${netty-version}</version>

</dependency>

#### 3.2 在配置中心里添加redis配置



#redis.host=127.0.0.1

redis.host=10.186.43.45

redis.port=6379

redis.password=

#最小空闲数

redis.minIdle=2

#最大空闲数

redis.maxIdle=10

#最大连接数

redis.maxTotal=1000

#最大建立连接等待时间

redis.maxWaitMillis=3000

#客户端超时时间单位是毫秒

redis.timeout=120000

redis.commandTimeout=60000

redis.shutdownTimeout=5000

#明是否在从池中取出连接前进行检验,如果检验失败,则从池中去除连接并尝试取出另一个

redis.testOnBorrow=true

#缓存配置

redis.expiration=600

redis.enableTransaction=true

redis.cache.ttl=10

redis.cache.enableTransaction=true

#集群配置

redis.cluster.nodes=

#redis.cluster.nodes=127.0.0.1:7001,127.0.0.1:7002,127.0.0.1:7003,127.0.0.1:7004,127.0.0.1:7005,127.0.0.1:7006

redis.cluster.max-redirects=3

#主从配置

redis.master=

redis.slave=

#### 3.3 redis配置类

在springBoot中引入redis后，自动帮我们在容器中生成一个RedisTemplate和StringRedisTemplate。但是这两个redis模板的泛型是<Object,Object>,写代码不方便，需要写很多类型转换的代码。我们需要一个泛型为<String,Oject>形式的RedisTemplate。并且这个模板没有设置数据存在Redis时，Key和value的序列化方法。

在我们系统中的配置类在iTMS-common中，路径为top.itms.core.config

类名：RedisConfig

类中有6个方法，具体如下：

**redisPoolConfig：**返回redis连接池配置，返回值类型GenericObjectPoolConfig （对象池）。通过PropertiesUtil类来读取.properties文件中

的对应配置。

**clientResources：**返回值类型ClientResources，destroyMethod（摧毁方法）=关闭。

**redisConnectionFactory：**返回值类型RedisConnectionFactory(Redis连接工厂)，

用上面两个方法返回值当成参数，整合Lettuce来进行连接池连接配置。

**redisTemplate：**返回值类型RedisTemplate，参数类型RedisConnectionFactory

通过连接工厂设置RedisTemplate，然后将其返回。

**redisCacheManager：**返回值类型CacheManager，通过RedisConnectionFactory，配置CacheManager（缓存管理器）。缓存管理器管理的是缓存中的数据。

**redisHelper ：**返回值类型RedisHelper(redis的工具类)

#### 3.4 redis工具类

了解完redis如何配置，接下来我们来看一下redis的工具类的实现。工具类的封装是为了写代码更方便一点。

在文件夹top.itms.core.support.cache下有redis的常用缓存辅助类-RedisHelper

我们来看一下辅助类中具体的实现。

RedisHelper 继承CacheManager

结合代码，我们开看CacheManager中定义的具体方法的接口

Logger ***logger*** = LogManager.*getLogger*();

用来输入错误日志

Object get(final String key);

通过key获取value

Object getFire(String key);

获取热点数据

Set<Object> getAll(String pattern, Integer expire);

批量获取数据

Set<Object> getAll(final String pattern);

通过一个set类型的key，获取一个set<Object>的value

void set(final String key, final Serializable value, int seconds);

把值放进缓存，expire(key, seconds);是设置缓存过期时间。

void set(final String key, final Serializable value);

把值放进缓存，过期时间是配置表中的默认数值

Boolean exists(final String key);

判断是否有该缓存

void del(final String key);

删除缓存

void delAll(final String pattern);

批量删除缓存

String type(final String key);

返回缓存类型

Boolean expire(final String key, final int seconds);

判断缓存是否在某段时间后失效

Boolean expireAt(final String key, final long unixTime);

判断缓存是否在某个时间段失效

Long ttl(final String key);

获取缓存过期时间，单位是秒

Object hget(String key, Serializable field);

获取指定键中的值

void hset(String key, Serializable field, Serializable value);

新增元素到指定键中

void hdel(String key, Serializable field);

批量删除map

Long incr(String key);

自增长map键的值

void setrange(String key, long offset, String value);

截取原有值的指定长度后添加新值在后面

String getrange(String key, long startOffset, long endOffset);

获取从指定位置开始，到指定位置为止的值

Object getSet(final String key, final Serializable value);

获取原来的值并重新赋新值

boolean setnx(String key, Serializable value);

没有值存在则添加

boolean lock(String key, String requestId, long seconds);

判断是否被锁

boolean unlock(String key, String requestId);

判断锁