# 基本

Redis是一个键值的内存数据库，在c#中使用redis，目前采用的是StackExchange.Redis.dll

Redis是个内存数据库，同时它是单线程的

## 支持的结构

Redis提供了对5种结构的操纵

* String

可能是字符串，整数或者浮点数

支持对整个字符串的读写或者对其一部分进行读写；对整数和浮点数进行自增或自减

* List（列表）

一个双向链表，链表上每个节点包含一个字符串

支持从链表两端推入或弹出元素，根据偏移量对链表进行修剪，读取单个或多个元素，根据值查找或移除元素

* Set（集合）

元素是字符串的无序集合。要求每个字符串都不能重复

添加，获取，移除单个元素，检查一个元素是否存在于集合，计算交叉并集，从集合随机获取元素

* Hash（散列）

包含键值对的无序列表

添加，获取，移除某个键值对；获取所有键值对

* ZSET(有序集合)

字符串成员与浮点数分值之间的有序集合，元素的排序顺序由分值决定

添加，获取，移除单个键值对；获取所有键值对；添加，获取，移除单个元素；根据分值范围或成员来获取元素

# 使用方向

* 可以当做内存数据库处理，就需要用到redis提供的指令来操作redis的5种结构
* 可以当做消息中间件来处理，就可以用到redis提供的订阅发布（不推荐使用，实际上需要用到更加专业的消息中间件来做这个处理，例如MQ）。主要受限于两个方面：

1. 和redis的稳定性有关，在旧版的redis中，如果一个客户端订阅了消息，但是客户端的接收速率赶不上发布速率，那么消息的堆积会导致redis运行越来越缓慢最终崩溃。最新版的redis不会出现这个问题，因为会自动断开不符合 client-output-buffer-limit pubsub 配置项的客户端
2. 还有一个原因是数据传输的可靠性，任何网络操作都可能遇到断线，如果已经订阅成功，那么在连接断开后，客户端会尝试重连；但是如果是在订阅操作执行中发生了连接断开，那么订阅就无法建立，此后发布的数据就都不能接受。

# 客户端使用

## 命令模式

* Sync 同步模式，会直接阻塞调用者
* Async异步模式，采用的是Task方式
* Fire-and-Forget，发送命令而不理会方法的返回值，调用了一个命令后会立刻返回，返回值都是虚假的默认值，它不理会命令最终的执行结果。

后面两种模式实际就是使用了管道，将命令封装在管道中。

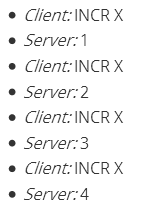
## 管道

### 作用

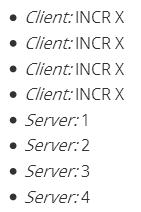
Redis中的管道（PipeLine）特性：简述一下就是，Redis如何从客户端一次发送多个命令，服务端到客户端如何一次性响应多个命令。  
 Redis使用的是客户端-服务器模型和请求/响应协议的TCP服务器,这就意味着一个请求要有以下步骤才能完成：1、客户端向服务器发送查询命令，然后通常以阻塞的方式等待服务器相应。2、服务器处理查询命令，并将相应发送回客户端。这样便会通过网络连接，如果是本地回环接口那么就能特别迅速的响应，但是如果走外网，甚至外网再做一系列的层层转发，那就显的格外蛋疼。无论网络延时是多少，那么都将占用整体响应的时间。这样一来如果一次发送1个命令，网络延时为100ms，我们不得不做。那么如果1次发1000个命令，那么网络延时100\*1000ms就很难容忍啦。  
 针对与上面的问题，Redis在2.6版本以后就都提供管道（Pipeline）功能。他可以使客户端在没有读取旧的响应时，处理新的请求。这样便可以向服务器发送多个命令，而不必等待答复，直到最后一个步骤中读取答复。这被称为管线（PipeLine）；实际就是在客户端对请求进行了缓存，上层调用直接将请求推入缓存，底层会在之后批量将请求给到redis，由redis最后统一应答。

客户端和服务端的交互模式：

一般如下图，每一次都需要等待应答后才进行下一次的请求



管道如下图，最后才会进行统一应答

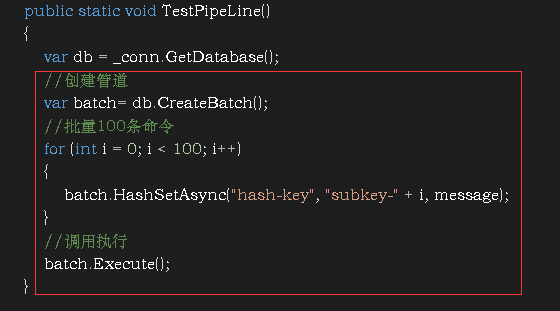


### 使用

在redis中使用命令pipeline，当传入参数true或不传入参数时，客户端会自动使用MULTI和EXEC包裹所有要执行的命令（就是开启一个事务）；如果传入参数时false，那么就只是包裹了所有要执行的命令，仅仅和服务进行一次数据交互就批量执行了命令，而不是开启事务。

最后通过指令pipeline命令获取到的对象调用execute指令来执行命令

.net中实例：



在使用异步模式或者使用Fire-and-Forget模式，都会自动的使用管道。

### 缺陷

* 鉴于Pipepining发送命令的特性，Redis服务器是以队列来存储准备执行的命令，而队列是存放在有限的内存中的，所以不宜一次性发送过多的命令。如果需要大量的命令，可分批进行，效率不会相差太远。
* 由于pipeline的原理是收集需执行的命令，到最后才一次性执行。所以无法在中途立即查得数据的结果（需待pipelining完毕后才能查得结果），这样会使得无法立即查得数据进行条件判断（比如判断是非继续插入记录），只有在命令都执行完后统一收到结果

### 相关

<http://www.cnblogs.com/nick-huang/p/5785304.html>

<http://www.3fwork.com/b803/000692MYM024048/>

<https://redis.io/topics/pipelining>

## 事务

### 作用

Redis事务很简单，满足一致性和隔离性，但是不满足原子性和持久性

Redis只是保证了客户端的事务不受干扰，一个事务执行完毕后，redis才会处理其他命令。

* 原子性

不满足，redis的任务当中任何 命令失败，不会导致之前成功的命令回滚；也不会阻止后续命令执行

* 持久性

事务本身只是保证一组命令顺序执行，不提供任何额外持久化功能，持久化由redis服务本身来提供

* 隔离性和一致性

事务执行过程中，不执行其他命令，等事务所有命令执行完成后才处理其他命令，满足隔离性。

Redis事务保证一致性

### 使用

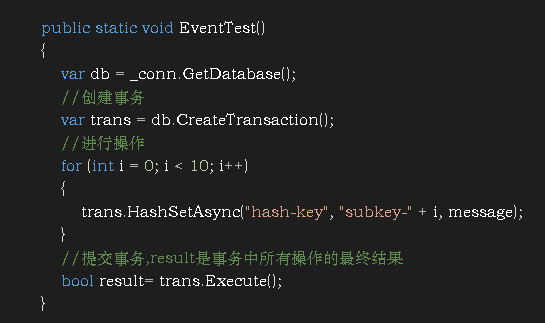
在命令中，通过multi命令开启事务，然后通过exec执行事务，两个命令之间的所有命令作为任务命令顺序执行。

可以使用命令watch，在使用watch对键进行监视后，一直到exec执行前，如果被监视的键有更新，替换或删除操作，那么exec操作时事务会失败并抛出异常

可以使用命令discard，来进行对multi命令后的

为了提高效率，redis客户端会在最后，一次性（通过exec命令）将命令提交给redis服务执行（这是事务的作用之一，能够有效减少服务端和客户端之间的交互次数）

.NET中使用事务



### 注意

1. 执行命令的操作需为异步操作。
2. 在事物中执行的命令，都不会直接看到结果，故此结果也不能用于下面代码做判断，因为当前的异步命令在Execute()之前是不会对数据库产生任何影响的。

### 缺陷

* 事务不具备一致性，因此不会回滚，事务中有一个操作出错不会影响后续操作执行也不会回滚之前已完成的操作
* 如果事务中有查询命令，那么查询结果不会立刻返回，而是和所有命令结果一起在事务执行完的时候返回

解决方案：

1. 可以使用多个库，读写分离，查询库只用于查询，更新库只用于事务写操作
2. 不使用redis使用

* 事务每条指令都和redis服务进行交互

这个问题一般redis客户端API都会进行优化

解决方案：

1. 对于所有get/set操作，可以使用mget/mset指令
2. 对于不同类型的更新操作，使用lua脚本打包后一次发给服务

### 相关

<http://www.cnblogs.com/me115/p/4326375.html>

<http://www.3fwork.com/b803/000692MYM024048/>

<https://redis.io/topics/pipelining>

## 锁

Redis使用watch命令代替对数据库加锁，它只会对数据进行监视而不会阻止对数据的操作，作用类似 乐观锁。

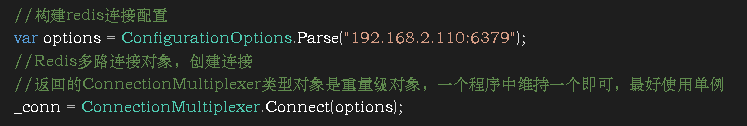
Redis的排它锁的调用对象不是进程，线程，而是不同机器上的redis客户端。使用指令SETNX来构建排它锁。

性能上来说乐观锁开销小，但是如果是使用乐观锁要达到悲观锁 的功能（即如果监视对象改变就重新使用乐观锁），根据不同的需求可能效率比悲观锁更低（因为可能不停的重复使用乐观锁）

# .NET使用redis相关

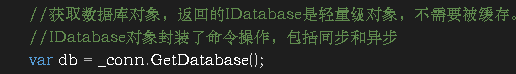
## 创建连接

创建连接只需要两步：创建配置对象和创建连接对象：



## 使用命令

使用命令之前需要创建一个数据库对象，这个对象封装了所有命令的同步和异步方法，它是个可以随时创建的对象



### 具体操作

Redis主要是通过提供命令操作具体的5种数据结构:string,list,set,hash,zset；对它们都有对应的同步和异步的get和set方法用来进行存储和读取。

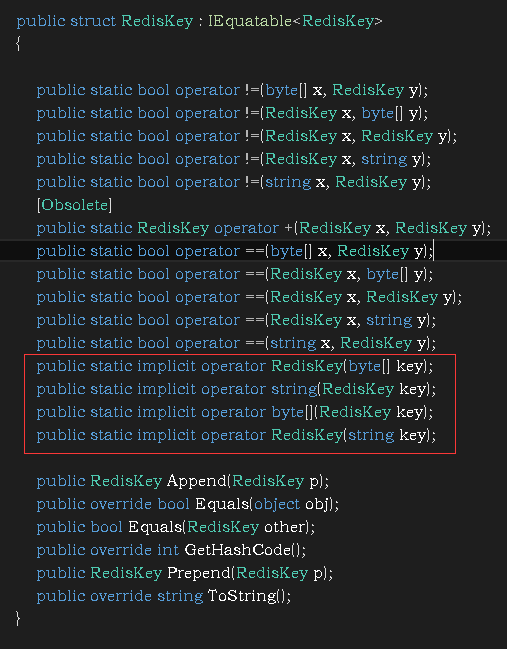
在API中，命令方法的所有参数都是以下几种种结构：

RedisKey，RedisValue和HashEntry,SortedSetEntity

如果是订阅发布模式，还需要用到结构：

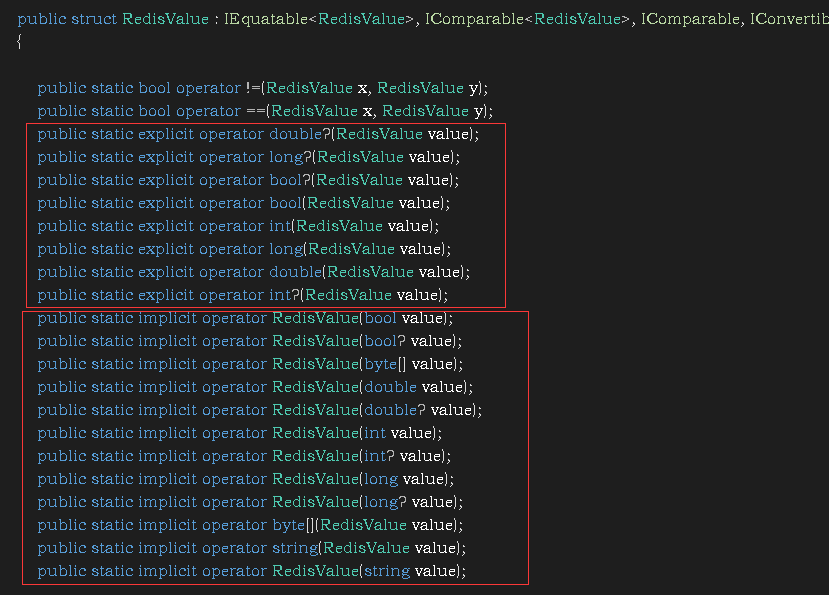
RedisChannel

* RedisKey是Redis中的Key：



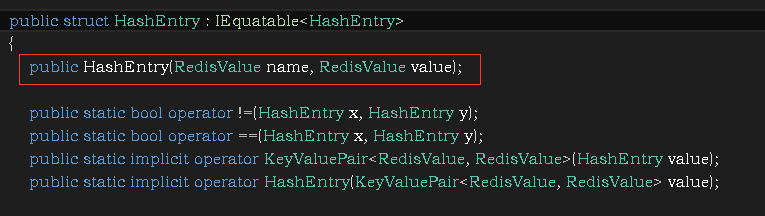
它提供了与string和byte数组的相互之间的隐式转换，即Redis的Key只能支持string和字节数组对象

* RedisValue



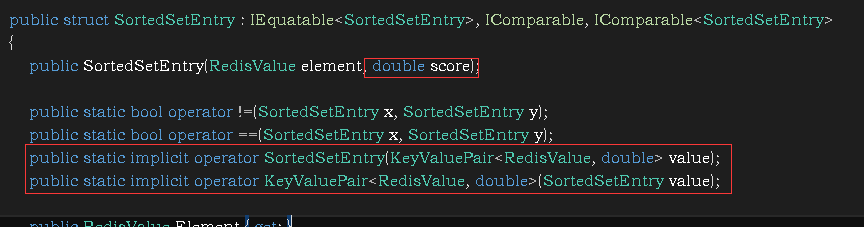
它是redis的值的提现，它封装了与bool,bool?,byte[],double,double?,int,int?long,long?string,string?之间的相互的显式和隐式转换，即它能够支持常用的整型和浮点型，字符串，字节数组。

* HashEntry



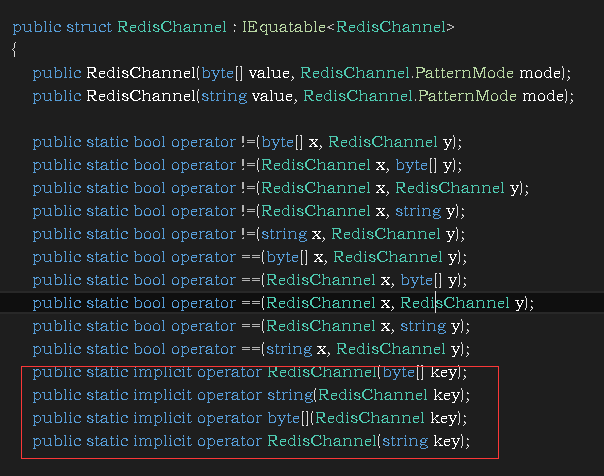
这个对象用于Redis的Hash结构，封装Hash的Subkey和Value存入redis

* SortedSetEntity



这个对于用于Redis的有序集合，封装了member和sorce，但是可以看到sorce只能是浮点数类型，有序集合对象会根据sorce的值进行排序

* RedisChannel



RedisChannel只能接受byte数组和string

# 数据持久化

数据持久化由redis服务提供（需要进行配置），分为快照持久化和只追加文件，可以分别使用和组合使用

* 快照持久化

快照是Redis通过创建快照来获取存储在内存中的数据在某个时间点上的副本，创建快照后，用户可以对快照进行备份；备份可以赋值到其他服务器也可以留在本地以供重启使用。

根据配置，快照会被写入到指定的文件中，并存储在设定的路径上。如果这个过程中redis，系统或硬件中任一出现崩溃，那么redis会丢失最近一次快照之后到崩溃时所有内存中的数据。

因此快照适用于容许丢失部分数据的情况。

* AOF持久化

AOF持久化会将被执行的写命令写到AOF文件的末尾，来记录数据发生的变化。因此redis只需要从头到尾执行难一次AOF包含的所有写命令，就能恢复数据。

# 其他

键名尽量简短（包括redis键和散列，有序集合的子键）

Redis中如果key值相同，那么对key做的任何操作都会替换掉key原有的value，一般在添加数据时，如果是新添加的key则返回true；如果是修改原有的key的value则返回false

相关资料：

<http://www.cnblogs.com/bnbqian/p/4962855.html>

## 使用要点

* 控制键名，Redis常用作热数据的第二数据存储，当主数据库中数据删除时，要注意级联删除Redis中的数据。否则可能导致Redis中存储大量无效数据，而因为Redis是非关系型的，所以除非键的命名有一套成熟的规则，否则很难清除这些数据
* 控制键名长度，Redis是内存数据库，键名越长意味着内存开销越大
* 使用合适的数据结构，就效率而言，sortedSet最昂贵，如果不需要排序，可以使用hash代替；如果不需要用到set特性，使用list代替；如果进行了大量的string存储，可以考虑将所有string存储到hash中进行分组。
* 使用scan时别使用键，scan是一个基于游标的迭代器，每次调用都返回一个新游标

相关

<http://www.csdn.net/article/2014-09-29/2821930-5-key-takeaways-for-developing-with-redis>

# 降低内存占用

* 短结构

Redis为列表，集合，散列和有序集合提供一组配置选项，这些选项能够让Redis以更节约空间的方式存储长度较短的结构（“短结构”）。这些数据每次读取要经过 解码；每次写入要经过编码。

* 压缩列表

压缩列表是体积较小的列表（list），散列（hash）和有序集合（zset）三种对象的非结构化表示，通常redis使用双向链表表示列表，散列表表示散列，散列表加跳跃表表示有序集合的做法不同，压缩列表以序列化方式存储数据（以此来节省内存空间）。

可以通过限制每个节点的容量或者元素数量来进行压缩

例如：list-max-ziplist-entries 512 表示列表最多512元素就进行压缩

List-max-ziplist-value 64 表示列表容量最多64M就进行压缩

* 集合的整数集合编码

体积较小的集合也有自己的紧凑表示：如果整数包含的所有成员都可以解释为十进制数，

这些整数又处于平台的有符号整数范围内，就可以使用整数集合表示。

整数集合的配置只能通过限制容量大小来进行压缩。

因为对于短结构数据的每次操作都要经过编码或解码，甚至还要对内存中数据移动，因此如果一个长度大的压缩列表可能会给性能带来负面影响。

经测试，压缩列表的长度限制在500~2000个元素，每个元素体积限制在128字节以下是合理的；一般限制最大1024个元素，每个元素不超过64字节。

* 分片结构

分片本质上是基于某些规则将数据分成更小的部分，然后根据数据所属的部分来决定数据发送到哪个位置。

* 分片散列

对散列进行分片首先需要选择一个方法对数据进行划分。可以把散列的键作为一个信息源，并使用散列函数为键计算出一个数字散列值（即计算出一个RedisKey）。然后进行存储

也就是实际上，是把在一个HASH中存储的键值对存放到不同的hash中，让每个hash容量尽量小，然后针对这些hash进行短结构压缩，来达到节省空间的目的。

集合的分片原理类似

* 打包存储二进制位和字节

如果被存储的是一些简短并且长度固定的连续ID，就有比分片更节省内存的存储方式。

# 效率

## IO吞吐

因为Redis是一个内存数据库，所以实际在使用Redis的过程中，效率瓶颈不在于Redis服务端的操作（1.基本所有操作都是内存操作，非常快；2.使用了非阻塞的IO操作），效率瓶颈集中在网络IO上，因此客户端提高操作Redis的效率，就需要从网络下手（例如使用pipelining，或者使用多个连接去处理）

通过实际测试，在本地稳定的网络环境中测试：

每一包数据10KB，中通1W包

批量提交（使用管道或异步）连接局域网redis，函数执行过程非常迅速(30毫秒左右，仅仅将命令填入底层缓存)，但是通过抓包从数据流量分析，全部数据被发送到redis耗时6秒左右。即6秒内发送了1W条命令总共10万KB的数据，每秒1666包，每秒16.666MB的数据

单条提交，连接局域网redis， 耗时6000毫秒左右，对于上层来说，同步比异步的时间开销大很多，但是实际数据发送到redis花的时间差距不大（主要是因为在局域网内进行测试，网络影响小）；连接连接本机器redis，耗时750-1000毫秒。

可以看到在内网环境中，网络影响不大的情况下，批量和单条的写入量都是16.6M/s;但是在不走网络的情况下，效率可以提高6-8倍。

相关

<https://www.zhihu.com/question/19764056>

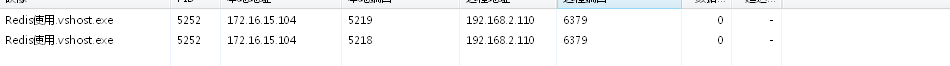
<http://blog.csdn.net/blackcomet/article/details/40349457>

<http://blog.csdn.net/nightelve/article/details/16854223>

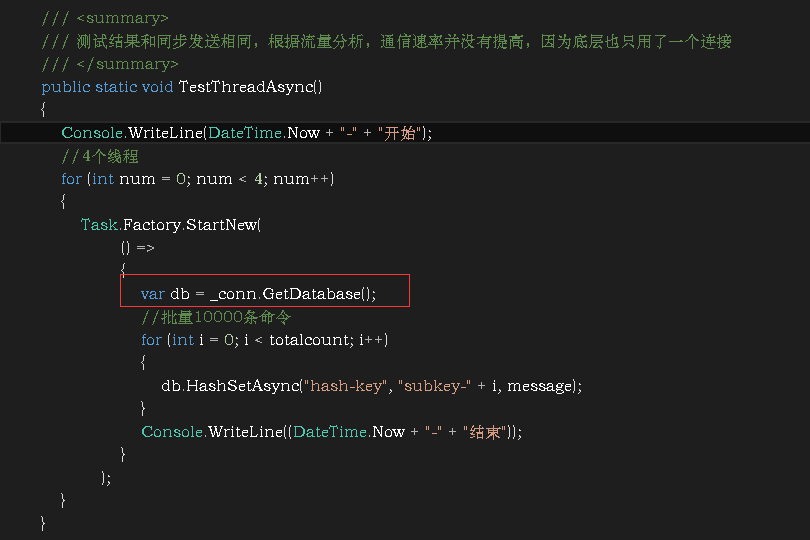
## 多线程/多连接

像之前所说，redis效率的瓶颈主要在网络方面，为了提高效率，可以使用多连接 发送redis数据，提高网络发送效率。

需要注意的是，在.net中使用客户端组件,使用一个ConnectionMultiplexer对象，底层实际只有2个TCP连接：



此后即使反复多次调用GetDatabase函数来获取数据提交对象，实际底层并不会新建连接而是公用现有连接，所以实际通信效率并不会提高：



因此，如果需要提高效率，就需要建立多个和redis服务的连接，也就是需要创建多个ConnectionMultiplexer对象（每次创建这个对象都会建立2个TCP连接）：

