项目的最小支持版本：.netframework 4.0

将项目的框架改为4.0，点击项目属性改变

问：.NET Framework 4和.NET Framework 4 Client Profile的区别与联系

 最近接触了一些visual studio 2010的使用，发现出现了.NET Framework 4和.NET Framework 4 Client Profile。一开始总是弄不大明不二者之间有什么区别，因为和工作没啥直接关系，所以也就暂时压住了好奇，今天有空看了一下msdn里的介绍，给自己总结一下。

    我先自己定一下简称，这样叫起来比较方便。

.NET Framework 4 -〉 full版

.NET Framework 4 Client Profile -〉 client版

区别

client版比full版要小，装起来也快一点。这条虽然拿不上台面，但这确实也是区别。

client版主要针对单机桌面程序，这当然也包括windows form程序和WPF程序。full版除了包括client版的功能之外，更多还涉及一些server端的功能，比如asp.net程序。

如果需要一些特殊的开发工具支持的情况，还是得full版出马。

本人估计：以后恐怕.net framework也会分client版和server版。

联系

二者都可以在windows update里不断得到各自的更新。

根据开发时对最终用户的需求可以选择不同的版本进行编译。

问：对象浏览器

文档：Documentation

* [Basic Usage](https://github.com/StackExchange/StackExchange.Redis/blob/master/Docs/Basics.md) - getting started and basic usage

The central object in StackExchange.Redis is the ConnectionMultiplexer class in the StackExchange.Redis namespace; this is the object that hides away the details of multiple servers. Because the ConnectionMultiplexer does a lot, it is designed to be **shared and reused** between callers. You should not create a ConnectionMultiplexer per operation. It is fully thread-safe and ready for this usage. In all the subsequent examples it will be assumed that you have a ConnectionMultiplexer instance stored away for re-use. But for now, let's create one. This is done using ConnectionMultiplexer.Connect or ConnectionMultiplexer.ConnectAsync, passing in either a configuration string or a ConfigurationOptions object. The configuration string can take the form of a comma-delimited(逗号分割) series of nodes, so let's just connect to an instance on the local machine on the default port (6379):

using StackExchange.Redis;

...

ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("localhost");

// ^^^ store and re-use this!!!

Note that ConnectionMultiplexer implements IDisposable（可释放的） and can be disposed when no longer required. This is deliberately not showing using statement usage, because it is exceptionally rare that you would want to use aConnectionMultiplexer briefly, as the idea is to re-use this object.

A more complicated scenario（方案） might involve a master/slave setup; for this usage, simply specify all the desired nodes that make up that logical redis tier（等级） (it will automatically identify the master):

ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("server1:6379,server2:6379");

注：这里设置两个主和从

If it finds both nodes are masters, a tie-breaker（打乱等级） key can optionally be specified that can be used to resolve the issue, however such a condition is fortunately very rare.

Once you have a ConnectionMultiplexer, there are 3 main things you might want to do:

* access a redis database (note that in the case of a cluster, a single logical database may be spread over multiple nodes)
* make use of the [pub/sub](http://redis.io/topics/pubsub) features of redis
* access an individual server for maintenance（保持） / monitoring purposes

## Using a redis database

Accessing a redis database is as simple as:

IDatabase db = redis.GetDatabase(); //连接数据库

The object returned from GetDatabase is a cheap pass-thru （直通连接）object, and does not need to be stored. Note that redis supports multiple databases—多数据库连接 (although this is not supported on "cluster"注：在集群上还不支持多数据库连接); this can be optionally specified in the call to GetDatabase. Additionally, if you plan to make use of the asynchronous API and you require the [Task.AsyncState](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.asyncstate(v=vs.110).aspx) to have a value, this can also be specified:

int databaseNumber = ...

object asyncState = ...

IDatabase db = redis.GetDatabase(databaseNumber, asyncState);

Once you have the IDatabase, it is simply a case of using the [redis API](http://redis.io/commands). Note that all methods have both synchronous （同步）and asynchronous（异步） implementations（功能）. In line with Microsoft's naming guidance, the asynchronous methods all end ...Async(...), and are fully await-able etc.

The simplest operation would be to store and retrieve a value:

string value = "abcdefg";

db.StringSet("mykey", value);

...

string value = db.StringGet("mykey");

Console.WriteLine(value); // writes: "abcdefg"

Note that the String... prefix here denotes the [String redis type](http://redis.io/topics/data-types), and is largely separate to the [.NET String type](http://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx), although both can store text data. However, redis allows raw binary data for both keys and values （redis允许二进制作为key和value）- the usage is identical:

byte[] key = ..., value = ...;

db.StringSet(key, value);

...

byte[] value = db.StringGet(key);

The entire range of [redis database commands](http://redis.io/commands) covering all redis data types is available for use.

## Using redis pub/sub

Another common use of redis is as a [pub/sub message](http://redis.io/topics/pubsub) distribution tool; this is also simple, and in the event of connection failure, the ConnectionMultiplexer will handle all the details of re-subscribing to the requested channels.

ISubscriber sub = redis.GetSubscriber();

Again, the object returned from GetSubscriber is a cheap pass-thru object that does not need to be stored. The pub/sub API has no concept of databases, but as before we can optionally provide an async-state. Note that all subscriptions are global: they are not scoped to the lifetime of the ISubscriber instance. The pub/sub features in redis use named "channels"; channels do not need to be defined in advance on the server (an interesting use here is things like per-user notification channels, which is what drives parts of the realtime updates on [Stack Overflow](http://stackoverflow.com/)). As is common in .NET, subscriptions take the form of callback delegates which accept the channel-name and the message:

sub.Subscribe("messages", (channel, message) => {

Console.WriteLine((string)message);

});

Separately (and often in a separate process on a separate machine) you can publish to this channel:

sub.Publish("messages", "hello");

This will (virtually instantaneously) write "hello" to the console of the subscribed process. As before, both channel-names and messages can be binary.

Please also see [Pub / Sub Message Order](https://github.com/StackExchange/StackExchange.Redis/blob/master/Docs/PubSubOrder.md) for guidance on sequential versus concurrent message processing.

## Accessing individual servers

For maintenance purposes（大多数目的）, it is sometimes necessary to issue server-specific commands:

IServer server = redis.GetServer("localhost", 6379);

The GetServer method will accept an [EndPoint](http://msdn.microsoft.com/en-us/library/system.net.endpoint(v=vs.110).aspx) or the name/value pair that uniquely identify the server. As before, the object returned from GetServer is a cheap pass-thru object that does not need to be stored, and async-state can be optionally specified. Note that the set of available endpoints is also available:

EndPoint[] endpoints = redis.GetEndPoints();

From the IServer instance, the [Server commands](http://redis.io/commands#server) are available; for example:

DateTime lastSave = server.LastSave();

ClientInfo[] clients = server.ClientList();

## Sync vs Async vs Fire-and-Forget

There are 3 primary usage mechanisms with StackExchange.Redis:

Synchronous（同步的） - where the operation completes before the methods returns to the caller (note that while this may block （阻断）the caller, it absolutely does not block other threads—并不会阻断其他对redis的请求: the key idea in StackExchange.Redis is that it aggressively shares the connection between concurrent callers)

Asynchronous（异步） - where the operation completes some time in the future, and a Task or Task<T> is returned immediately, which can later:

be .Wait()ed (blocking the current thread until the response is available)

have a continuation callback added ([ContinueWith](http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.continuewith(v=vs.110).aspx) in the TPL)

be awaited (which is a language-level feature that simplifies the latter, while also continuing immediately if the reply is already known)

Fire-and-Forget - where you really aren't interested in the reply, and are happy to continue irrespective of the response

The synchronous usage is already shown in the examples above. This is the simplest usage, and does not involve the [TPL](http://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx).

For asynchronous usage, the key difference is the Async suffix（后缀） on methods, and (typically) the use of the await language feature. For example:

string value = "abcdefg";

await db.StringSetAsync("mykey", value);

...

string value = await db.StringGetAsync("mykey");

Console.WriteLine(value); // writes: "abcdefg"

The fire-and-forget usage is accessed by the optional CommandFlags flags parameter on all methods (defaults to none). In this usage, the method returns the default value immediately (so a method that normally returns a String will always returnnull, and a method that normally returns an Int64 will always return 0). The operation will continue in the background. A typical use-case of this might be to increment page-view counts:

db.StringIncrement(pageKey, flags: CommandFlags.FireAndForget);

* [Configuration](https://github.com/StackExchange/StackExchange.Redis/blob/master/Docs/Configuration.md) - options available when connecting to redis

Because there are lots of different ways to configure redis, StackExchange.Redis offers a rich configuration model, which is invoked （被行使）when calling Connect (or ConnectAsync)—调用Connect方法:

var conn = ConnectionMultiplexer.Connect(configuration);

The configuration here can be either:

* a ConfigurationOptions instance
* a string representing the configuration

The latter（后面的） is basically a tokenized （标记化的）form（形式） of the former.

## Basic Configuration Strings

The simplest configuration example is just the host name:

var conn = ConnectionMultiplexer.Connect("localhost"); //最简单的配置

This will connect to a single server on the local machine using the default redis port (6379). Additional options are simply appended (comma-delimited（逗号分隔的）). Ports are represented with a colon (:) as is usual. Configuration options include an = after the name. For example:

var conn = ConnectionMultiplexer.Connect("redis0:6380,redis1:6380,allowAdmin=true");

An overview of mapping between the string and ConfigurationOptions representation is shown below, but you can switch between them trivially（简单的）:

ConfigurationOptions options = ConfigurationOptions.Parse(configString);

or:

string configString = options.ToString();

A common usage is to store the basic details in a string, and then apply specific details at runtime:

string configString = GetRedisConfiguration();

var options = ConfigurationOptions.Parse(configString);

options.ClientName = GetAppName(); // only known at runtime

options.AllowAdmin = true;

conn = ConnectionMultiplexer.Connect(options);

Microsoft Azure（微软的云计算平台） Redis example with password

var conn = ConnectionMultiplexer.Connect("contoso5.redis.cache.windows.net,ssl=true,password=...");

## Configuration Options

The ConfigurationOptions object（配置对象） has a wide range of properties, all of which are fully documented in intellisense. Some of the more common options to use include:

| **Configuration string** | **ConfigurationOptions** | **Meaning** |
| --- | --- | --- |
| abortConnect={bool}一般设置为false | AbortOnConnectFail | If true, Connect will not create a connection while no servers are available |
| allowAdmin={bool} | AllowAdmin | Enables a range of commands that are considered risky |
| channelPrefix={string} | ChannelPrefix | Optional channel prefix for all pub/sub operations |
| connectRetry={int} | ConnectRetry | The number of times to repeat connect attempts during initialConnect—初始化连接时尝试连接的次数 |
| connectTimeout={int} | ConnectTimeout | Timeout (ms) for connect operations—断开连接的时间 |
| configChannel={string} | ConfigurationChannel | Broadcast channel name for communicating configuration changes |
| defaultDatabase={int} | DefaultDatabase | Default database index, from 0 to databases - 1 |
| keepAlive={int} | KeepAlive | Time (seconds) at which to send a message to help keep sockets alive—保持连接的时间 |
| name={string} | ClientName | Identification for the connection within redis |
| password={string} | Password | Password for the redis server |
| proxy={proxy type}使用的代理方式 | Proxy | Type of proxy in use (if any); for example "twemproxy" |
| resolveDns={bool} | ResolveDns | Specifies that DNS resolution should be explicit and eager, rather than implicit |
| serviceName={string} | ServiceName | Not currently implemented (intended for use with sentinel) |
| ssl={bool} | Ssl | Specifies that SSL encryption（加密） should be used |
| sslHost={string} | SslHost | Enforces a particular SSL host identity on the server's certificate |
| syncTimeout={int} | SyncTimeout | Time (ms) to allow for synchronous operations |
| tiebreaker={string} | TieBreaker | Key to use for selecting a server in an ambiguous master scenario |
| version={string} | DefaultVersion | Redis version level (useful when the server does not make this available) |
| writeBuffer={int} | WriteBuffer | Size of the output buffer—输出缓冲的大小 |

Tokens in the configuration string are comma-separated（逗号分隔的）; any without an = sign are assumed to be redis server endpoints—终端. Endpoints without an explicit port will use 6379 if ssl is not enabled, and 6380 if ssl is enabled. Tokens starting with $ are taken to represent command maps, for example: $config=cfg.

## Automatic and Manual Configuration

In many common scenarios（情况下）, StackExchange.Redis will automatically configure a lot of settings, including the server type and version, connection timeouts, and master/slave relationships. Sometimes, though, the commands for this have been disabled on the redis server. In this case, it is useful to provide more information:

ConfigurationOptions config = new ConfigurationOptions

{

EndPoints （终端）=

{

{ "redis0", 6379 },

{ "redis1", 6380 }

},

CommandMap = CommandMap.Create(new HashSet<string>

{ // EXCLUDE a few commands---排除掉一些命令，这些命令不能使用

"INFO", "CONFIG", "CLUSTER",

"PING", "ECHO", "CLIENT"

}, available: false),

KeepAlive = 180,

DefaultVersion = new Version(2, 8, 8),

Password = "changeme"

};

Which is equivalent （等价于）to the command string:

redis0:6379,redis1:6380,keepAlive=180,version=2.8.8,$CLIENT=,$CLUSTER=,$CONFIG=,$ECHO=,$INFO=,$PING=

## Renaming Commands

A slightly unusual feature of redis is that you can disable（禁止） and/or rename （重命名）individual（个人的） commands. As per （根据）the previous example, this is done via the CommandMap, but instead of passing a HashSet<string> to Create() (to indicate the available or unavailable commands), you pass a Dictionary<string,string>. All commands not mentioned in the dictionary are assumed to be enabled and not renamed. A null or blank value records that the command is disabled. For example:

var commands = new Dictionary<string,string> {

{ "info", null }, // disabled

{ "select", "use" }, // renamed to SQL equivalent for some reason

};

var options = new ConfigurationOptions {

// ...

CommandMap = CommandMap.Create(commands),

// ...

}

The above is equivalent to (in the connection string):

$INFO=,$SELECT=use

## Twemproxy—类似集群代理

[Twemproxy](https://github.com/twitter/twemproxy) is a tool that allows multiple redis instances（多redis实例） to be used as though（像） it were a single server, with inbuilt sharding and fault tolerance (much like redis cluster, but implemented separately). The feature-set（特征集） available to Twemproxy is reduced（简化的）. To avoid having to configure this manually, the Proxy option can be used:

var options = new ConfigurationOptions

{

EndPoints = { "my-server" },

Proxy = Proxy.Twemproxy //使用代理的配置

};

## Tiebreakers （附加属性）and Configuration Change Announcements

Normally StackExchange.Redis will resolve master/slave nodes automatically（正常情况下它会自动处理主从节点）. However, if you are not using a management tool such as redis-sentinel or redis cluster, there is a chance that occasionally（某些情况下） you will get multiple master nodes (for example, while resetting a node for maintenance it may reappear on the network as a master). To help with this, StackExchange.Redis can use the notion of a *tie-breaker* - which is only used when multiple masters are detected (not including redis cluster（redis集群）, where multiple masters are *expected（集群里是希望多master的）*). For compatibility with（兼容） BookSleeve, this defaults to the key named "\_\_Booksleeve\_TieBreak"(always in database 0). This is used as a crude（粗鲁的） voting mechanism to help determine the *preferred* master, so that work is routed （纠正）correctly.

Likewise, when the configuration is changed (especially the master/slave configuration), it will be important for connected instances to make themselves aware of the new situation (via INFO, CONFIG, etc - where available). StackExchange.Redis does this by automatically subscribing to a pub/sub channel upon which such notifications may be sent. For similar reasons, this defaults to "\_\_Booksleeve\_MasterChanged".

Both options can be customized or disabled (set to ""), via the .ConfigurationChannel and .TieBreaker configuration properties.

These settings are also used by the IServer.MakeMaster() method, which can set the tie-breaker in the database and broadcast （广播）the configuration change message. The configuration message can also be used separately to master/slave changes simply to request all nodes to refresh their configurations, via the ConnectionMultiplexer.PublishReconfiguremethod.

* [Keys, Values and Channels](https://github.com/StackExchange/StackExchange.Redis/blob/master/Docs/KeysValues.md) - discusses the data-types used on the API

In dealing with redis, there is quite an important distinction between keys and everything else. A key is the unique name of a piece of data (which could be a String, a List, Hash, or any of the other [redis data types](http://redis.io/topics/data-types)) within a database. Keys are never interpreted as... well, anything: they are simply inert names. Further - when dealing with clustered or sharded systems, it is the key that defines the node (or nodes if there are slaves) that contain this data - so keys are crucial（关键的） for routing commands.

This contrasts（对比） with values; values are the things that you store against keys - either individually (for String data) or as groups. Values do not affect command routing（安路线发送） (caveat（警告）: except for [the SORT command](http://redis.io/commands/sort) when BY or GET is specified, but that is really complicated to explain). Likewise, values are often interpreted by redis for the purposes of an operation:

* incr (and the various similar commands) interpret String values as numeric data
* sorting can interpret values using either numeric or unicode rules—根据数字和unicode规则排序
* and many others

The key point is that the API needs to understand what is a key and what is a value. This is reflected（反射的） in the StackExchange.Redis API, but the good news is that most of the time you don't need to know about this at all.

When using pub/sub, we are dealing with channels; channels do not affect routing (so they are not keys), but are quite distinct from regular values, so are considered separately.

## Keys

StackExchange.Redis represents keys by the RedisKey type. The good news, though, is that this has implicit conversions（隐式的会话） to and from both string and byte[], allowing both text and binary keys to be used without any complication. For example, the StringIncrement method takes a RedisKey as the first parameter, but *you don't need to know that*; for example:

string key = ...

db.StringIncrement(key);

or

byte[] key = ...

db.StringIncrement(key);

Likewise, there are operations that *return* keys as RedisKey - and again, it simply works:

string someKey = db.KeyRandom();

## Values

StackExchange.Redis represents values by the RedisValue type. As with RedisKey, there are implicit conversions（隐式的会话） in place which mean that most of the time you never see this type, for example:

**db.StringSet("mykey", "myvalue");**

However, in addition to text and binary contents, values can also need to represent typed primitive data - most commonly (in .NET terms) Int32, Int64, Double or Boolean. Because of this, RedisValue provides a lot more conversion support thanRedisKey:

db.StringSet("mykey", 123); // this is still a RedisKey and RedisValue

...

int i = (int)db.StringGet("mykey");

Note that while the conversions from primitives（原始的） to RedisValue are implicit（隐式的）, many of the conversions from RedisValue to primitives are explicit（显示的）: this is because it is very possible that these conversions will fail if the data does not have an appropriate value.

Note additionally that *when treated numerically*, redis treats a non-existent key as zero; for consistency with （一致）this, nil responses are treated as zero:

db.KeyDelete("abc");

int i = (int)db.StringGet("abc"); // this is ZERO

If you need to detect the nil condition, then you can check for that:

db.KeyDelete("abc");

var value = db.StringGet("abc");

bool isNil = value.IsNull; // this is true

or perhaps more simply, just use the provided Nullable<T> support:

db.KeyDelete("abc");

var value = (int?)db.StringGet("abc"); // behaves as you would expect

## Hashes

Since the field names in hashes do not affect command routing, they are not keys, but can take both text and binary names; thus they are treated as values for the purposes of the API.

## Channels

Channel names for pub/sub are represented by the RedisChannel type; this is largely identical to RedisKey, but is handled independently since while channel-names are rightly first-class elements, they do not affect command routing.

## Scripting

[Lua scripting in redis](http://redis.io/commands/EVAL) has two notable features:

* the inputs must keep keys and values separate (which inside the script become KEYS and ARGV, respectively)
* the return format is not defined in advance: it is specific to your script

Because of this, the ScriptEvaluate method accepts two separate input arrays: one RedisKey[] for the keys, oneRedisValue[] for the values (both are optional, and are assumed to be empty if omitted). This is probably one of the few times that you'll actually need to type RedisKey or RedisValue in your code, and that is just because of array variance rules:

var result = db.ScriptEvaluate(TransferScript,

new RedisKey[] { from, to }, new RedisValue[] { quantity });

(where TransferScript is some string containing Lua, not shown for this example)

The response uses the RedisResult type (this is unique to scripting; usually the API tries to represent the response as directly and clearly as possible). As before, RedisResult offers a range of conversion operations - more, in fact than RedisValue, because in addition to being interpreted as text, binary, primitives and nullable-primitives, the response can also be interpreted as arrays of such, for example:

string[] items = db.ScriptEvaluate(...);

## Conclusion

The types used in the API are very deliberately chosen（有意选择的） to distinguish redis *keys* from *values*. However, in virtually all cases you will not need to directly refer to the underlying types involved, as conversion operations are provided.

# Where are KEYS, SCAN, FLUSHDB etc?

Some very common recurring questions are:

There doesn't seem to be a Keys(...) or Scan(...) method? How can I query which keys exist in the database?

or

There doesn't seem to be a Flush(...) method? How can I remove all the keys in the database?

The key word （关键词）here, oddly （奇妙的）enough, is the last one: database. Because StackExchange.Redis aims to target scenarios（情况） such as cluster（集群）, it is important to know which commands target（针对） the database (the logical database that could be distributed over multiple nodes), and which commands target the server（针对服务器的）. The following commands all target a single server:--下面的命令都是针对服务器的

* KEYS / SCAN
* FLUSHDB / FLUSHALL
* RANDOMKEY
* CLIENT
* CLUSTER
* CONFIG / INFO / TIME
* SLAVEOF
* SAVE / BGSAVE / LASTSAVE
* SCRIPT (not to be confused with EVAL / EVALSHA)
* SHUTDOWN
* SLOWLOG
* PUBSUB (not to be confused with PUBLISH / SUBSCRIBE / etc)
* some DEBUG operations

(I've probably missed at least one) Most of these will seem pretty obvious, but the first 3 rows are not so obvious:

* KEYS / SCAN only list keys that are on the current server; not the wider logical database
* FLUSHDB / FLUSHALL only remove keys that are on the current server; not the wider logical database
* RANDOMKEY only selects a key that is on the current server; not the wider logical database

Actually, StackExchange.Redis spoofs（模仿） the RANDOMKEY one on the IDatabase API by simply selecting a target server at random（通过随机选择一个服务器）, but this is not possible for the others.

**So how do I use them?**

Simple: start from a server, not a database.

// get the target server

var server = conn.GetServer(someServer);

// show all keys in database 0 that include "foo" in their name

foreach(var key in server.Keys(pattern: "\*foo\*")) {

Console.WriteLine(key);

}

// completely wipe（清除） ALL keys from database 0

server.FlushDatabase();

Note that unlike the IDatabase API (where the target database has already been selected in the GetDatabase() call), these methods take an optional parameter for the database, or it defaults to 0.

The Keys(...) method deserves special mention: it is unusual in that it does not have an \*Async counterpart（没有对应的异步方法）. The reason for this is that behind the scenes, the system will determine the most appropriate method to use (KEYS vs SCAN, based on the server version), and if possible will use the SCAN approach to hand you back an IEnumerable<RedisKey> that does all the paging internally - so you never need to see the implementation details of the cursor operations. If SCAN is not available, it will use KEYS, which can cause blockages（阻塞） at the server. Either way, both SCAN and KEYS will need to sweep（扫描） the entire keyspace, so should be avoided on production servers - or at least, targeted at slaves.

**So I need to remember which server I connected to? That sucks!**

No, not quite. You can use conn.GetEndPoints() to list the endpoints (either all known, or the ones specified in the original configuration - these are not necessarily the same thing), and iterate with GetServer() to find the server you want (for example, selecting a slave).

DOCS

Timeout异常：

System.TimeoutException: Timeout performing HSET A0TM:3205:SourceInfo, inst: 1, mgr: ExecuteSelect, err: never, queue: 2, qu: 0, qs: 2, qc: 0, wr: 0, wq: 0, in: 0, ar: 0, clientName: DESKTOP-DN1NA98, IOCP: (Busy=0,Free=1000,Min=4,Max=1000), WORKER: (Busy=24,Free=32743,Min=4,Max=32767), Local-CPU: 27.45% (Please take a look at this article for some common client-side issues that can cause timeouts: https://github.com/StackExchange/StackExchange.Redis/tree/master/Docs/Timeouts.md)

在 StackExchange.Redis.ConnectionMultiplexer.ExecuteSyncImpl[T](Message message, ResultProcessor`1 processor, ServerEndPoint server) 位置 C:\Users\wukai\Downloads\StackExchange.Redis-1.1.608\StackExchange.Redis-1.1.608\StackExchange.Redis\StackExchange\Redis\ConnectionMultiplexer.cs:行号 2059

在 StackExchange.Redis.RedisBase.ExecuteSync[T](Message message, ResultProcessor`1 processor, ServerEndPoint server) 位置 C:\Users\wukai\Downloads\StackExchange.Redis-1.1.608\StackExchange.Redis-1.1.608\StackExchange.Redis\StackExchange\Redis\RedisBase.cs:行号 81

在 StackExchange.Redis.RedisDatabase.HashSet(RedisKey key, RedisValue hashField, RedisValue value, When when, CommandFlags flags) 位置 C:\Users\wukai\Downloads\StackExchange.Redis-1.1.608\StackExchange.Redis-1.1.608\StackExchange.Redis\StackExchange\Redis\RedisDatabase.cs:行号 226

在 MoliServer20.Publisher.Common.SourceInfoPublisher.PublishWorker(Object videoItemIds) 位置 E:\svn\_molitv\_server30\trunk\Tools\MoliServer20.Publisher\Common\SourceInfoPublisher.cs:行号 135

在 MoliServer20.Publisher.Common.Worker.Run(Object state) 位置 E:\svn\_molitv\_server30\trunk\Tools\MoliServer20.Publisher\Common\Worker.cs:行号 33

在 System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state, Boolean preserveSyncCtx)

在 System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback callback, Object state, Boolean preserveSyncCtx)

在 System.Threading.QueueUserWorkItemCallback.System.Threading.IThreadPoolWorkItem.ExecuteWorkItem()

## Are you getting network or CPU bound?

Verify what's the maximum bandwidth（带宽） supported on your client and on the server where redis-server is hosted. If there are requests that are getting bound by bandwidth, it will take longer for them to complete and thereby can cause timeouts. Similarly, verify you are not getting CPU bound on client or on the server box which would cause requests to be waiting for CPU time and thereby have timeouts.

## Are there commands taking long time to process on the redis-server?

There can be commands that are taking long time to process on the redis-server causing the request to timeout. Few examples of long running commands are mget with large number of keys, keys \* or poorly written lua script. You can run the SlowLog command to see if there are requests taking longer than expected. More details regarding the command can be found [here](http://redis.io/commands/slowlog)

## Was there a big request preceding （在…之前）several small requests to the Redis that timed out?

The parameter “qs”（这里的异常值为2） in the error message tells you how many requests were sent from the client to the server, but have not yet processed a response. For some types of load you might see that this value keeps growing, because StackExchange.Redis uses a single TCP connection （单一TCP连接）and can only read one response at a time. Even though the first operation timed out, it does not stop the data being sent to/from the server, and other requests are blocked until this is finished. Thereby, causing timeouts. One solution is to minimize the chance of timeouts by ensuring that your redis-server cache is large enough for your workload and splitting large values into smaller chunks. Another possible solution is to use a pool of ConnectionMultiplexer objects in your client, and choose the "least loaded" ConnectionMultiplexer when sending a new request. This should prevent a single timeout from causing other requests to also timeout.

## Are you seeing high number of busyio or busyworker threads in the timeout exception?

Let's first understand some details on ThreadPool Growth:

The CLR ThreadPool has two types of threads - "Worker" and "I/O Completion Port" (aka IOCP) threads.

* Worker threads are used when for things like processing Task.Run(…) or ThreadPool.QueueUserWorkItem(…) methods. These threads are also used by various components in the CLR when work needs to happen on a background thread.
* IOCP threads are used when asynchronous IO happens (e.g. reading from the network).

The thread pool （线程池）provides new worker threads or I/O completion threads on demand (without any throttling（扼杀）) until it reaches the "Minimum" setting for each type of thread. By default, the minimum number of threads is set to the number of processors on a system.

Once the number of existing (busy) threads hits the "minimum" number of threads, the ThreadPool （线程池）will throttle（扼杀） the rate at which is injects new threads to one thread per 500 milliseconds. This means that if your system gets a burst（突发） of work needing an IOCP thread, it will process that work very quickly. However, if the burst of work is more than the configured "Minimum" setting, there will be some delay in processing some of the work as（当） the ThreadPool waits for one of two things to happen ：

1. An existing thread becomes free to process the work

2. No existing thread becomes free for 500ms, so a new thread is created.

Basically, it means that when the number of Busy threads is greater than Min threads, you are likely paying a 500ms delay before network traffic is processed by the application. Also, it is important to note that when an existing thread stays idle （空闲的）for longer than 15 seconds (based on what I remember), it will be cleaned up and this cycle of growth and shrinkage can repeat.

If we look at an example error message from StackExchange.Redis (build 1.0.450 or later), you will see that it now prints ThreadPool statistics (see IOCP and WORKER details below).

System.TimeoutException: Timeout performing GET MyKey, inst: 2, mgr: Inactive,

queue: 6, qu: 0, qs: 6, qc: 0, wr: 0, wq: 0, in: 0, ar: 0,

IOCP: (Busy=6,Free=994,Min=4,Max=1000),

WORKER: (Busy=3,Free=997,Min=4,Max=1000)

In the above example, you can see that for IOCP thread there are 6 busy threads and the system is configured to allow 4 minimum threads. In this case, the client would have likely seen two 500 ms delays because 6 > 4.

Note that StackExchange.Redis can hit timeouts if growth of either IOCP or WORKER threads gets throttled.

Recommendation: Given the above information, it's recommend to set the minimum configuration value for IOCP and WORKER threads to something larger than the default value. We can't give one-size-fits-all guidance on what this value should be because the right value for one application will be too high/low for another application. This setting can also impact the performance of other parts of complicated applications, so you need to fine-tune this setting to your specific needs. A good starting place is 200 or 300, then test and tweak（稍稍调整） as needed.

How to configure this setting:

* In ASP.NET, use the ["minIoThreads" configuration setting](https://msdn.microsoft.com/en-us/library/vstudio/7w2sway1(v=vs.100).aspx) under the <processModel> configuration element in machine.config. If you are running inside of Azure WebSites, this setting is not exposed through the configuration options. You should be able to set this programmatically (see below) from your Application\_Start method in global.asax.cs.

**Important Note:** the value specified in this configuration element is a *per-core* setting. For example, if you have a 4 core machine and want your minIOThreads setting to be 200 at runtime, you would use <processModel minIoThreads="50"/>.

* Outside of ASP.NET, use the [ThreadPool.SetMinThreads(…)](https://msdn.microsoft.com/en-us/library/system.threading.threadpool.setminthreads(v=vs.100).aspx) API.