LEARNING StackExchange.Redis

#stackexchange.redis

Table of Contents

About

Chapter 1: Getting started with StackExchange.Redis

Remarks

Installing

Binaries for StackExchange.Redis are available on Nuget, and the source is available on Github.

Common Tasks

- Profiling

Versions

Version	Release Date
1.0.187	2014-03-18

Examples

Basic Usage

```
using StackExchange.Redis;

// ...

// connect to the server
ConnectionMultiplexer connection = ConnectionMultiplexer.Connect("localhost");

// select a database (by default, DB = 0)
IDatabase db = connection.GetDatabase();

// run a command, in this case a GET
RedisValue myVal = db.StringGet("mykey");
```

Reuse Multiplexer Across Application

```
class Program
{
    private static Lazy<ConnectionMultiplexer> _multiplexer =
        new Lazy<ConnectionMultiplexer>(
        () => ConnectionMultiplexer.Connect("localhost"),
        LazyThreadSafetyMode.ExecutionAndPublication);

    static void Main(string[] args)
    {
        IDatabase db1 = _multiplexer.Value.GetDatabase(1);
        IDatabase db2 = _multiplexer.Value.GetDatabase(2);
    }
}
```

Configuration Options

Connect to Redis server and allow admin (risky) commands

```
ConfigurationOptions options = new ConfigurationOptions()
            {
                EndPoints = { { "localhost", 6379}},
                AllowAdmin = true,
                ConnectTimeout = 60*1000,
            };
ConnectionMultiplexer multiplexer = ConnectionMultiplexer.Connect(options);
```

or

```
ConnectionMultiplexer multiplexer =
    ConnectionMultiplexer.Connect("localhost:6379,allowAdmin=True,connectTimeout=60000");
```

Connect to Redis server via SSL

```
 ConfigurationOptions options = new ConfigurationOptions()
            {
                EndPoints = { { "localhost", 6380}},
                Ssl = true,
                Password = "12345"
            };
ConnectionMultiplexer multiplexer = ConnectionMultiplexer.Connect(options);
```

or

```
ConnectionMultiplexer multiplexer =
     ConnectionMultiplexer.Connect("localhost:6380,ssl=True,password=12345");
```

Read Getting started with StackExchange.Redis online: https://riptutorial.com/stackexchange-redis/topic/1/getting-started-with-stackexchange-redis

Chapter 2: Keys and Values

Examples

Setting Values

All values in Redis are ultimately stored as a `RedisValue` type:

```
//"myvalue" here is implicitly converted to a RedisValue type
//The RedisValue type is rarely seen in practice.
db.StringSet("key", "aValue");
```

Setting and getting an int

```
db.StringSet("key", 11021);
int i = (int)db.StringGet("key");
```

Or using StackExchange.Redis.Extensions:

```
db.Add("key", 11021);
int i = db.Get<int>("key");
```

Read Keys and Values online: https://riptutorial.com/stackexchange-redis/topic/11/keys-and-values

Chapter 3: Pipelining

Examples

Pipelining and Multiplexing

```
var multiplexer = ConnectionMultiplexer.Connect("localhost");
IDatabase db = multiplexer.GetDatabase();

// intialize key with empty string
await db.StringSetAsync("key", "");

// create transaction that utilize multiplexing and pipelining
ITransaction transacton = db.CreateTransaction();
Task<long> appendA = transacton.StringAppendAsync("key", "a");
Task<long> appendB = transacton.StringAppendAsync("key", "b");

if (await transacton.ExecuteAsync()) // sends "MULTI APPEND KEY a APPEND KEY b EXEC
// in single request to redis server
{
    // order here doesn't matter, result is always - "abc".
    // 'a' and 'b' append always together in isolation of other Redis commands
    // 'c' appends to "ab" because transaction is already executed successfully
    await appendA;
    await db.StringAppendAsync("key", "c");
    await appendB;
}

string value = db.StringGet("key"); // value is "abc"
```

Read Pipelining online: https://riptutorial.com/stackexchange-redis/topic/3217/pipelining

Chapter 4: Profiling

Remarks

StackExchange.Redis's profiling features are composed of the `IProfiler` interface, and the `ConnectionMultiplexer.RegisterProfiler(IProfiler)`, `ConnectionMultiplexer.BeginProfiling(object)`, `ConnectionMultiplexer.FinishProfiling(object)` methods.

Begin and Finish profiling take a context `object` so that related commands can be grouped together.

This grouping works by querying your `IProfiler` interface for a context object at the start of a command, before any threading shenanigans have happened, and associating that command with a any other commands that have the same context object. Begin must be called with the same context object so StackExchange.Redis knows to start profiling commands with that context object, and Finish is called to stop profiling and return the results.

Examples

Group all commands from set of threads together

```
class ToyProfiler : IProfiler
{
    public ConcurrentDictionary<Thread, object> Contexts = new ConcurrentDictionary<Thread,
object>();

    public object GetContext()
    {
        object ctx;
        if(!Contexts.TryGetValue(Thread.CurrentThread, out ctx)) ctx = null;

        return ctx;
    }
}


// ...

ConnectionMultiplexer conn = /* initialization */;
var profiler = new ToyProfiler();
var thisGroupContext = new object();

conn.RegisterProfiler(profiler);

var threads = new List<Thread>();

for (var i = 0; i < 16; i++)
{
    var db = conn.GetDatabase(i);

    var thread =
        new Thread(
            delegate()
            {
                var threadTasks = new List<Task>();

                for (var j = 0; j < 1000; j++)
                {
                    var task = db.StringSetAsync("" + j, "" + j);
                    threadTasks.Add(task);
                }

                Task.WaitAll(threadTasks.ToArray());
            }
        );

    profiler.Contexts[thread] = thisGroupContext;

    threads.Add(thread);
}

conn.BeginProfiling(thisGroupContext);

threads.ForEach(thread => thread.Start());
threads.ForEach(thread => thread.Join());

IEnumerable<IProfiledCommand> timings = conn.FinishProfiling(thisGroupContext);
```

At the end, timings will contain 16,000 IProfiledCommand objects - one for each command issued to redis.

Group commands based on issuing thread

```
ConnectionMultiplexer conn = /* initialization */;
var profiler = new ToyProfiler();

conn.RegisterProfiler(profiler);

var threads = new List<Thread>();

var perThreadTimings = new ConcurrentDictionary<Thread, List<IProfiledCommand>>();

for (var i = 0; i < 16; i++)
{
    var db = conn.GetDatabase(i);

    var thread =
        new Thread(
            delegate()
            {
                var threadTasks = new List<Task>();

                conn.BeginProfiling(Thread.CurrentThread);

                for (var j = 0; j < 1000; j++)
                {
                    var task = db.StringSetAsync("" + j, "" + j);
                    threadTasks.Add(task);
                }

                Task.WaitAll(threadTasks.ToArray());
                perThreadTimings[Thread.CurrentThread] =
conn.FinishProfiling(Thread.CurrentThread).ToList();
            }
        );

    profiler.Contexts[thread] = thread;

    threads.Add(thread);
}

threads.ForEach(thread => thread.Start());
threads.ForEach(thread => thread.Join());
```

`perThreadTimings` ends up with 16 entries of 1,000 IProfilingCommands, keyed by the Thread that issued them.

Read Profiling online: https://riptutorial.com/stackexchange-redis/topic/4/profiling

Chapter 5: Publish Subscribe

Examples

Basics

Once connected you can publish messages by calling the `ISubscriber.Publish` method:

```
// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();

// publish a message to the 'chat' channel
subscriber.Publish("chat", "This is a message")
```

Consumers can subscribe to the channel using the `ISubscriber.Subscribe` method. Messages sent by the publisher will be handled by the handler passed to this method.

```
// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();

// subscribe to a messages over the 'chat' channel
subscriber.Subscribe("chat", (channel, message) => {
    // do something with the message
    Console.WriteLine((string)message);
});
```

Complex Data (JSON)

You can broadcast more complex messages by serializing the payload before you publish it:

```
// definition of a message
public class ChatMessage
{
    public Guid Id { get; set; }
    public string User { get; set; }
    public string Text { get; set; }
}

// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();

var message = new ChatMessage
{
    Id = Guid.NewGuid(),
    User = "User 1234",
    Text = "Hello World!"
};

// serialize a ChatMessage
// this uses JIL to serialize to JSON
var json = JSON.Serialize(message);
```

```
// publish the message to the 'chat' channel
subscriber.Publish("chat", json)
```

The subscriber then needs to deserialize the message:

```
// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();

// subscribe to messages over the 'chat' channel
subscriber.Subscribe("chat", (channel, json) => {
    var message = JSON.Deserialize<ChatMessage>(json);

    // do something with the message
    Console.WriteLine($"{message.User} said {message.Text}");
});
```

Complex Data (Protobuf)

StackExchange.Redis also supports sending bytes over the pub/sub channel, here we use protobuf-net to serialize our message to a byte array before sending it:

```
// definition of a message (marked up with Protobuf attributes)
[ProtoContract]
public class ChatMessage
{
    [ProtoMember(1)]
    public Guid Id { get; set; }
    [ProtoMember(2)]
    public string User { get; set; }
    [ProtoMember(3)]
    public string Text { get; set; }
}

// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();

var message = new ChatMessage
{
    Id = Guid.NewGuid(),
    User = "User 1234",
    Text = "Hello World!"
};

using (var memoryStream = new MemoryStream())
{
    // serialize a ChatMessage using protobuf-net
    Serializer.Serialize(memoryStream, message);

    // publish the message to the 'chat' channel
    subscriber.Publish("chat", memoryStream.ToArray());
}
```

Again the subscriber needs to deserialize the message upon receipt:

```
// grab an instance of an ISubscriber
var subscriber = connection.GetSubscriber();
```

```
// subscribe to messages over the 'chat' channel
subscriber.Subscribe("chat", (channel, bytes) => {
    using (var memoryStream = new MemoryStream(bytes))
    {
        var message = Serializer.Deserialize<ChatMessage>(memoryStream);

        // do something with the message
        Console.WriteLine($"{message.User} said {message.Text}");
    }
});
```

Read Publish Subscribe online: https://riptutorial.com/stackexchange-redis/topic/1610/publish-subscribe

Chapter 6: Scan

Syntax

- `public IEnumerable<RedisKey> Keys(int database = 0, RedisValue pattern = default(RedisValue), int pageSize = CursorUtils.DefaultPageSize, long cursor = CursorUtils.Origin, int pageOffset = 0, CommandFlags flags = CommandFlags.None)`

Parameters

Parameter	Details
database	Redis database index to connect to
pattern	*Unsure*
pageSize	Number of items to return per page
cursor	*Unsure*
pageOffset	Number of pages to offset the results by
flags	*Unsure*

Remarks

The `Keys()` call will select either the `KEYS` or `SCAN` command based on the version of the Redis server. Where possible it will prefer the usage of `SCAN` which returns an `IEnumerable<RedisKey>` and does not block. `KEYS` on the other hand will block when scanning the key space.

Examples

Basic scanning of all keys on server

```
// Connect to a target server using your ConnectionMultiplexer instance
IServer server = conn.GetServer("localhost", 6379);

// Write out each key in the server
foreach(var key in server.Keys()) {
    Console.WriteLine(key);
}
```

Iterating using a cursor

```
// Connect to a target server using your ConnectionMultiplexer instance
IServer server = conn.GetServer("localhost", 6379);
```

```
var seq = server.Keys();
IScanningCursor scanningCursor = (IScanningCursor)seq;

// Use the cursor in some way...
```

Read Scan online: https://riptutorial.com/stackexchange-redis/topic/66/scan

Credits

S. No	Chapters	Contributors
1	Getting started with StackExchange.Redis	Adam Lear, Community, Kevin Montrose, Nilay Vishwakarma, Vladimir Dorokhov
2	Keys and Values	Arie Litovsky, Cigano Morrison Mendez, Kevin Montrose
3	Pipelining	Vladimir Dorokhov
4	Profiling	Jason Punyon, Kevin Montrose, Shog9
5	Publish Subscribe	Dean Ward
6	Scan	Joseph Vaughan