StackExchange.Redis文档翻译

书栈(BookStack.CN)

目　录

致谢

当前文档 《StackExchange.Redis文档翻译》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-06-07。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/StackExchange.Redis-Chinese-Doc

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

StackExchange.Redis 文档翻译

StackExchange.Redis 文档翻译

ServiceStack.Redis从4.0版本开始收费使用，好在好的东西不存在没有开源免费的；
大名鼎鼎的Stack Overflow就提供了它的Redis客户端库：StackExchange.Redis免费给我们使用。
看到网上很少有关于StackExchange.Redis的翻译，个人本着学习共享精神翻译了一下，
翻译应该有一些错误(有些地方我都感觉好像不是用英文的人写的，可能是我的英语水平本就不够，所以请读者原谅)，
读者可以综合原文阅读，如果发现错误可以提交一个PR给我，我会合并改正。

目录

来源(书栈小编注)

> *https://github.com/carldai0106/StackExchange.Redis-Chinese-Doc*

基本用法

基本用法

ConnectionMultiplexer 类是StackExchange.Redis的中枢对象，它在StackExchange.Redis名称空间中；
这个对象封装了很多基础服务对象的详细信息。由于 **ConnectionMultiplexer** 做了很多底层处理，它在调用者之间被设计为共享和重用。你不应该为每一个操作都创建一个 **ConnectionMultiplexer** 对象。该对象是完全线程安全的。在随后所有的示例中，**ConnectionMultiplexer** 被假定成一个存储起来且可供重用的对象。
现在，让我们来创建一个。我们可以使用 **ConnectionMultiplexer.Connect** 或者 **ConnectionMultiplexer.ConnectAsync** 并且传递一个配置字符串或者 **ConfigurationOptions** 对象来完成创建。配置字符串可以被逗号分隔成一系列的节点形式，让我们在本地机器上使用默认端口 **6379** 来连接到一个实例。

```
using StackExchange.Redis;
...
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("localhost");
// ^^^ 存储并重用该对象
```

注意：**ConnectionMultiplexer** 实现了 **IDisposable** 接口，当不在需要的时候，我们可以释放它。但是，我故意不显示的使用 **using** 语句。由于该对象是一个很耗费资源的对象，因此最好是重用该对象。

一个更复杂的场景可能涉及到主/从服务的设定。对于这种用法，只需要简单的设定一个连接字符串，该字符串包含主从服务器。(它能够自动的识别出主服务器)

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

如果发现两个节点都是主节点(主服务器)，它能够通过打破平衡(权衡决策)指定那个是主服务器，从而解决这个问题。但是这种情况是非常罕见的。

如果你有了一个 **ConnectionMultiplexer** 对象，那么有3个事情是你可能想要做的：

- 访问redis数据库(注意：在集群情况下，一个单一的逻辑数据库可能分布在多个节点当中)
- 使用redis的发布/订阅功能
- 以维护和监控为目的，访问一个独立的服务器

使用redis数据库

访问redis数据库就是这样简单：

```
IDatabase db = redis.GetDatabase();
```

GetDatabase 方法返回的对象是一个廉价的直通对象，并不需要存储。注意redis支持多数据库(尽管这不是支持集群)；在调用 **GetDatabase** 时可以任意的指定调用的是那个数据库。还有，如果你计划使用异步API,那你需要为 Task.AsyncState.aspx) 指定一个值，也可以这样指定：

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

一旦你拥有 **IDatabase** 对象，那么我们就可以简单的调用Redis API。注意所有的方法都有同步和异步实现。这符合微软的命名规范，异步方法全部以 Async结尾，并且全部都是可等待的。

最简单的操作是存储并且取回一个值：

```
string value = "abcdefg";
db.StringSet("mykey", value);
...
string value = db.StringGet("mykey");
Console.WriteLine(value); // writes: "abcdefg"
```

注意：String前缀表示的是Redis的String类型，并且它与.Net的Sting类型在很大程度上是不同的。尽管这两者都能存储文本数据。然而，redis允许Key和Value都为原生的字节数据。示例如下：

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

Redis数据库命令所覆盖的Redis数据类型都是可用的。

使用Redis发布/订阅功能

Redis另一个常用的功能是作为发布/订阅消息的分发工具；这也是很简单的，
在连接失败时，**ConnectionMultiplexer** 将会处理所有的重订阅细节。

```
ISubscriber sub = redis.GetSubscriber();
```

GetSubscriber 返回了一个不需要存储的廉价对象。 发布/订阅API没有数据库的概念，但正如之前所提到的，我们可以提供一个异步状态(async-state)。注意：所有的订阅都是全局的。它们不局限于 **ISubscriber** 的生命周期。发布订阅功能在Redis中叫做通道”channels”；通道不需要预先定义在服务器上(一个令人关注的使用是每个用户的通知通道，例如：Stack Overflow 实时驱动更新部分)。这在.NET中是很常见的，订阅采用匿名函数回调的方式来处理发布消息：

```
sub.Subscribe("messages", (channel, message) => {
    Console.WriteLine((string)message);
});
```

你可以单独的发布一个消息到通道中：

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

这会将”hello”(实时的)发布到订阅了该消息的控制台中。正如前面所提到的，通道名称和消息都可以使用字节类型。

在顺序性和消息并发处理方面，请参考发布/订阅 消息顺序

访问独立的服务器

出于维护为目的，有时候需要发出特定于某个服务器的命令：

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

GetServer 方法将会接受一个 **EndPoint** 对象或服务器端有唯一标识的键值对对象。和之前一样，**GetServer** 方法返回一个不需要存储的廉价对象。并且异步状态(async-state)是可被选择指定的。注意：可用的终结点的集合也是可用的：

```
1. EndPoint[] endpoints = redis.GetEndPoints();
```

来自 **IServer** 的服务器端命令都是可用的；例如：

```
1. DateTime lastSave = server.LastSave();
2. ClientInfo[] clients = server.ClientList();
```

同步 VS 异步 VS 即发即弃

在StackExchange.Redis中，有3个主要的使用机制：

- 同步 - 在操作完成之前方法会返回给调用者(注意：尽管这可能会阻塞调用者，但是决不会阻塞其他线程；StackExchange.Redis的关键理念就是它积极共享并发调用方之间的连接)
- 异步 - 操作将在未来的某个时间完成，并且以 **Task** 或者 **Task** 类型立即返回；可以用以下方式实现：
    - 使用 **Wait** 方法 (阻塞当前线程直到响应可用)
    - 使用 **ContinueWith.aspx)** 方法(添加一个延续性的回调函数)
    - 使用 **await** 运算符 (这是一个简化之后的语言级功能)
- 即发即弃 - 对答复不感兴趣并且乐于忽略响应

从上面我们看到了同步使用的例子，这是最简单的使用，并且不涉及TPL.aspx)。

对于异步使用，关键的区别是方法后缀 **Async**，并且使用 **awiat** 运算符。例如：

```
1. string value = "abcdefg";
2. await db.StringSetAsync("mykey", value);
3. ...
4. string value = await db.StringGetAsync("mykey");
5. Console.WriteLine(value); // writes: "abcdefg"
```

在所有的方法中，使用即发即弃访问是通过可选参数 **CommandFlags flags** (默认是传入该参数)来实现的。这样使用时，方法会立即返回默认值(因此通常返回一个字符串的方法会一直返回 **null**，而返回一个 **Int64** 方法会一直返回0)。该操作将会在后台继续执行。一个典型的用例是页面点击率统计：

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

配置

配置

配置Redis有很多不同的方式，StackExchange.Redis提供了一个丰富的配置模型，我们可在调用 **Connect** 或者 **ConnectAsync** 方法时传入配置：

```
1. var conn = ConnectionMultiplexer.Connect(configuration);
```

在这里参数configure可以是：

- **ConfigurationOptions** 实例配置
- 字符串方式配置

后面一种也是前面一种的标记形式。

基本配置 - 通过字符串配置

最简单的配置实例就是以主机名来配置：

```
1. var conn = ConnectionMultiplexer.Connect("localhost");
```

这会连接到本机上的单个服务器，默认使用Redis的缺省端口：6379。还有一些选项以逗号分隔的方式附加上去。端口通常用一个冒号(:)来表示。配置选项的名字后跟随了一个=号。如下所示：

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

我们可以自由的在 **string** 和 **ConfigurationOptions** 两者之间相互转换，如下所示：

```
1. ConfigurationOptions options = ConfigurationOptions.Parse(configString);
```

或者

```
1. string configString = options.ToString();
```

最常用的方式是将基本信息存储在一个字符串中，然后在运行时应用这个特定的基本信息：

```
1. string configString = GetRedisConfiguration();
2. var options = ConfigurationOptions.Parse(configString);
```

```
options.ClientName = GetAppName(); // 仅仅在运行时才知道
options.AllowAdmin = true;
conn = ConnectionMultiplexer.Connect(options);
```

在微软Azure上使用Redis并附上密码的例子：

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

配置选项

ConfigurationOptions 对象具有很多属性，所有选项的说明都已在智能提示里面。下面表格是一些最常用的选项描述：

Configuration string	ConfigurationOptions	描述
abortConnect={bool}	AbortOnConnectFail	如果是 true，当没有可用的服务器时 **Connect** 不会创建连接
allowAdmin={bool}	AllowAdmin	如果是 true，开启一些被认为是有风险的命令
channelPrefix={string}	ChannelPrefix	所有 pub/sub 操作的可选通道前缀
connectRetry={int}	ConnectRetry	在初始化 **Connect** 时，连接重试次数
connectTimeout={int}	ConnectTimeout	连接超时设置，时间单位是ms
configChannel={string}	ConfigurationChannel	设置广播通道名称
configCheckSeconds={int}	ConfigCheckSeconds	时间（秒）检查配置；如果支持的话，它可以对交互式套接字进行持久连接
defaultDatabase={int}	DefaultDatabase	默认数据库索引，从 **0** 到 **databases - 1**
keepAlive={int}	KeepAlive	如果在指定时间(seconds)内没有活动，那么发送一条信息来帮助socket保持连接
name={string}	ClientName	唯一名称，用来识别Redis里面的连接
password={string}	Password	Redis服务器密码
proxy={proxy type}	Proxy	使用的代理类型 （如果有的话）； 例如 “twemproxy”
resolveDns={bool}	ResolveDns	指定DNS解析方式是显示而不是隐式
serviceName={string}	ServiceName	当前没有实现
ssl={bool}	Ssl	指定使用SSL加密
sslProtocols={enum}	SslProtocols	使用加密连接时支持的Ssl / Tls版本。使用’	‘ 提供多个值。
sslHost={string}	SslHost	强制SSL主机识别，需要使用服务器端证书
syncTimeout={int}	SyncTimeout	异步超时设置(ms)
tiebreaker={string}	TieBreaker	主要是在一个模糊不清的主机之间选择出一个作为主服务器
version={string}	DefaultVersion	Redis 版本级别 （该选项是非常有用的，当服务器不可用时）
writeBuffer={int}	WriteBuffer	输出缓存区的大小

其他仅用于代码的选项：

- `ReconnectRetryPolicy(IReconnectRetryPolicy)` - 默认值：`ReconnectRetryPolicy = LinearRetry(ConnectTimeout);`

在配置字符串中的标记都是以逗号分隔的；任何没有 `=` 符号的都被假定为Redis的服务终端。如果没有开启SSL，并且终端没有指定一个明确的端口，那么将使用6379作为端口；如果开启了SSL，那么6380将作为端口。以 `$` 开始的标记会被当做命令来映射；例如：**`$config=cfg`** 。

自动配置与手动配置

在很多常见的情况下，`StackExchange.Redis` 将会自动的配置多个设置选项，包括服务器类型和版本，连接超时和主/从关系配置。可是有时候在Redis服务器这个命令是被禁止的。在这种情况下，提供更多的信息是非常有用的：

```
ConfigurationOptions config = new ConfigurationOptions
{
    EndPoints =
    {
        { "redis0", 6379 },
        { "redis1", 6380 }
    },
    CommandMap = CommandMap.Create(new HashSet<string>
    { // 排除几个命令
        "INFO", "CONFIG", "CLUSTER",
        "PING", "ECHO", "CLIENT"
    }, available: false),
    KeepAlive = 180,
    DefaultVersion = new Version(2, 8, 8),
    Password = "changeme"
};
```

上面的配置等同于下面的字符串配置：

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

重命名命令

在Redis中有些不常用的功能，那就是你能禁用或者重命名单个命令。正如前面的所展示的，这是通过 **`CommandMap`** 来实现的，而不是通过 **`HashSet\`** 来 **`Create()`**（用这个来指示可用和不可用的命令），可以传递一个 **`Dictionary\`**。所有不在字典里面命令都被假定为开启且未被重命名。**`null`** 或者空值记录的命令表示是被禁用的。例如：

```
var commands = new Dictionary<string,string> {
        { "info", null }, // 禁用
        { "select", "use" }, // 由于某种原因重命名为等效的SQL
};
var options = new ConfigurationOptions {
    // ...
    CommandMap = CommandMap.Create(commands),
    // ...
}
```

上面的配置等同于下面的字符串配置(在连接字符串中):

```
$INFO=,$SELECT=use
```

Twemproxy

Twemproxy是一个允许多个Redis实例使用起来像是单个服务一样的工具，它内置分片和容错能力(这很像Redis集群，但是它是单独实现的)。Twemproxy简化了功能设置的可用性。为了避免手动配置，**Proxy** 选项可以这样配置:

```
var options = new ConfigurationOptions
{
    EndPoints = { "my-server" },
    Proxy = Proxy.Twemproxy
};
```

打破僵局(Tiebreakers或者权衡决策)和配置更改公告

通常StackExchange.Redis会自动的解决主/从节点问题。然而可能你没有使用像 **Redis集群** 或 **Redis-Sentinel** 那样的管理工具，你可能会碰到这样的一个场合: 同时具有多个主节点(例如: 当我们以维护为目的而重新设置一个节点时，它可能作为一个主节点重新出现在网络上)。为了解决这个问题，StackExchange.Redis可以用打破僵局(权衡决策)这一概念，这个仅适用于多个主机被发现的情况。为了兼容 **BookSleeve**，默认键名是 **“__Booksleeve_TieBreak”**(一直是在索引为0的数据库中)。这用来作为一个简单的投票机制，帮助决策那个是首选主节点，使之以正确路由工作。

同样的，当配置发生更改的时候(尤其是在主/从配置的时候)，这对于连接实例来说将是非常重要的，这可以使它们自己意识到有新的情况发生(通过 **IFNO,CONFIG** 等等)。StackExchange.Redis通过自动订阅来 **pub/sub** 发送这样的通知。由于类似的原因，默认是键名是 **“__Booksleeve_MasterChanged”**。

这两个选项可以被自定义或者禁用(设置为“”)，可以通过 **.ConfigurationChannel** 和 **.TieBreaker** 来配置属性。

这些设置也可以通过 **IServer.MakeMaster()** 来配置，可以在数据库中设置打破僵局(tie-breaker)和广播配置更改的消息。通过 **ConnectionMultiplexer.PublishReconfigure** 方法，配置消息也可以单独使用主/从更新来请求所有的节点刷新它们的配置。

连接重试策略

当连接由于某种原因丢失时，StackExchange.Redis 会自动在后台尝试连接。它会不断的重试，直到连接重新恢复。它将使用ReconnectRetryPolicy来决定在重试之间应该等待多长时间。ReconnectRetryPolicy可以是线性(默认)，指数或自定义重试策略。

例如:

```
config.ReconnectRetryPolicy = new ExponentialRetry(5000); // defaults maxDeltaBackoff to 10000 ms
//重试      # 重试以毫秒为间隔重新连接
//1           随机值，在 5000 到 5500 之间
//2           随机值，在 5000 到 6050 之间
//3           随机值，在 5000 到 6655 之间
//4           随机值，在 5000 到 8053 之间
//5           随机值，在 5000 到 10000 之间，由于 maxDeltaBackoff 是 10000 ms
//6           随机值，在 5000 到 10000 之间

config.ReconnectRetryPolicy = new LinearRetry(5000);
//重试  #    重试以毫秒为间隔重新连接
```

```
12. //1           5000
13. //2           5000
14. //3           5000
15. //4           5000
16. //5           5000
17. //6           5000
```

管道和多路复用器

管道和多路复用器

延迟情况是难以忍受的。现代计算机能以惊人的速度生成数据，并且高速互联网(经常是在重要的服务器之间有多个并行连接)提供了极大的带宽，但是这可恶的延迟意味着电脑花了大量时间等待数据。基于延续的编程变得越来越流行的几个原因之一。让我们考虑一些规则的程序代码：

```
1. string a = db.StringGet("a");
2. string b = db.StringGet("b");
```

按照这些关联的步骤，这看起来像：

```
1.     [req1]                        # 客户端 : 客户端库构造出一个请求1
2.          [c=>s]                   # 网络   : 请求1被发送到服务器
3.               [server]            # 服务器 : 服务器处理请求1
4.                     [s=>c]        # 网络   : 响应1被发送回客户端
5.                          [resp1]  # 客户端 : 客户端库解析响应数据1
6.                                [req2]
7.                                     [c=>s]
8.                                          [server]
9.                                                [s=>c]
10.                                                    [resp2]
```

现在让我们突出客户端处理的部分：

```
1. [req1]
2.      [====waiting=====]
3.                      [resp1]
4.                            [req2]
5.                                 [====waiting=====]
6.                                                 [resp2]
```

记住这是不可测量的，如果是用时间来衡量，那么会一直等待下去(一直耗费时间在等待处理)。

管道

由于这个原因，很多Redis客户端允许你利用管道，处理发送多个消息而无需等待每一个的回复，并且当消息进来的时候，回复的处理将会延后。在.NET中，一个操作可以被初始化且尚未完成；在完成后或者发生错误后由TPL.aspx)通过 Task.aspx)/Task\.aspx) 的API封装。本质上，**Task** 表示的是一个”将来可能的T类型的值”(非泛型 **Task** 实际上是 **Task**)。你可以二选其一：

- **.Wait()** (阻塞执行，直到任务完成)
- **.ContinueWith(…)**或者**await** (创建一个在目标任务完成时异步执行的延续任务)

例如：下面是Redis客户端利用管道的示例代码：

```
var aPending = db.StringGetAsync("a");
var bPending = db.StringGetAsync("b");
var a = db.Wait(aPending);
var b = db.Wait(bPending);
```

注意：在这里我使用了 **db.Wait** 因为他会自动的应用同步超时配置，如你你喜欢的话，你也可以使用 **aPending.Wait()** 或者 **Task.WaitAll(aPending, bPending)**；使用管道允许我们在网络中立即得到两个请求，从而消除大部分的延时。此外，它也可以帮助我们减少包碎片：20个请求单独的发送(等待每个响应)至少需要20个包，但是在管道中发送20个请求只需要少数几个包(甚至只需要一个包)。

即发即弃

一个特别的管道案例是当我们不关心操作的响应，允许代码继续执行且排队操作是在后台处理的时候。这通常意味着我们能把并发工作放在来自一个单独调用的连接中。我们可以使用 **flags** 参数来实现：

```
// 可调期限
db.KeyExpire(key, TimeSpan.FromMinutes(5), flags: CommandFlags.FireAndForget);
var value = (string)db.StringGet(key);
```

FireAndForget 标记会使客户端库去正常的排队工作，但是会立即返回一个默认值(**KeyExpire** 会返回一个 **bool** 类型，这将返回 **false**，因为默认值是 **false** - 然而返回的是毫无意义的值，我们应该忽略)。***Async** 方法也会返回一个已完成的 **Task** 作为默认值(或者一个已完成的 **Task** 作为 **void** 返回)。

多路复用(Multiplexing)

使用管道处理技术是非常好的，但是我们经常单独使用阻塞代码仅去取一个单独的值(或者可能只执行一些操作，这取决于各自的需要)。这意味着我们仍然有这样一个问题：我们花费大量的时间去等待数据从客户端传输到服务器端。现在我们考虑一个繁忙的应用，这可能是一个web服务。这类应用通常都具有高并发性，当你有20个并行应用请求所有需要的数据，你可能想旋转(spinning up)这20个连接，或者你可以同步访问一个单独连接(这意味着最后的调用者需要等待前面19个全部执行完成才开始)。或者作为一个妥协方式，也许是个出租5个连接的连接池—不管你怎么做，都会有大量的等待操作。**StackExchange.Redis** 不需要那样做；反而，它为你做了大量的工作，通过多路复用单个连接，使你可以有效的利用空余的时间。当不同的调用方同时访问时，它会自动使用管道分离访问请求，所以无论使用阻塞方式或者异步方式访问，这些工作都是被管道处理的。因此我们可以有10或者20个先前的"get a and b"的场景(来自不同应用的请求)，并且它们会尽快的取得连接。从本质上讲，它填补了 **waiting** 时间与其他调用方的工作。

因此，StackExchange.Redis不会提供(并将永远不会提供)"阻塞弹出(blocking pops)"(BLPOP, BRPOP 以及 BRPOPLPUSH) - 因为这将允许一个单独的调用方拖延整个多路复用器，进而阻塞所有的调用方。
StackExchange.Redis 需要保持的工作是为了验证某个事务的前提条件，这就是为什么StackExchange.Redis封装了这样的条件在内部管理 **Condition** 实例。更多事务信息。如果你想要"阻塞弹出(blocking pops)"，那么我强烈建议你考虑使用发布/订阅功功能：

```
sub.Subscribe(channel, delegate {
    string work = db.ListRightPop(key);
    if (work != null) Process(work);
});
//...
db.ListLeftPush(key, newWork, flags: CommandFlags.FireAndForget);
sub.Publish(channel, "");
```

注意：无需阻塞操作即可达到相同的目的有：

- 数据不是通过发布/订阅发送的；发布/订阅API仅被用来通知工人来检查更多的工作
- 如果没有工人，那么新项仍然在缓冲列表中，工作不会执行
- 仅有一个工人能弹出一个值；当消费者多于生产者，一些消费者将得到通知然后发现没有什么可做的
- 当你重新启动工人，你应该假设有积压工作可以处理
- 除此之外，对于阻塞弹出的语义是相同的

并发

应该注意的是管道/多路复用器/future-value 等方式与基于
延续的异步代码也是做得非常好的；例如：

```
string value = await db.StringGet(key);
if (value == null) {
    value = await ComputeValueFromDatabase(...);
    db.StringSet(key, value, flags: CommandFlags.FireAndForget);
}
return value;
```

键，值以及通道

键、值以及通道

在对待Redis时候，键和其他的事物之间有个相当重要的区别。键是在数据库中一段数据的唯一标识(可能String，List，Hash或者其他的Redis数据类型)。键是没有任何实质意义，就是一个简单的名字。进一步说：当处理集群或者分片系统时，它就是定义在包含数据的节点上的Key，所以对于命令传送来说key是至关重要的。

值是相对于键来存储的。要么是单个(String数据)要么一组组的。值不会影响命令的传送(注意：除了SORT命令，并且该命令与BY或者GET组合使用的时候，但是这个真的很难解释说明；详细可以去看Redis的SORT命令文档)。值通常会被Redis以操作为目的来解读：

- **inc** (各种类似的命令)解读字符串值作为数字数据
- 排序：值要么以数值或以Unicode规则来排序
- 还有很多其他的

关键点在于API需要明白什么是Key和什么是Value。这个涉及到StackExchange.Redis的API，但是大多数时候你根本不需要知道这个。

当你在使用发布/订阅时，我们是使用通道来处理的；通道不影响到命令传送(所以它们不是Keys)，但是这与常规值是区别巨大的，所以应该分别考虑。

Keys

StackExchange.Redis中键的类型是 **RedisKey**。不过好在它会隐式的从 **string** 和 **byte[]** 转换，允许使用文本和二进制键。例如：**StringIncrement** 方法将 **RedisKey** 作为第一个参数，但是你不需要自己去做转换，例如：

```
1. string key = ...
2. db.StringIncrement(key);
```

或者

```
1. byte[] key = ...
2. db.StringIncrement(key);
```

同样的，有一些方法能够返回 **RedisKey** 类型的key；如下所示：

```
1. string someKey = db.KeyRandom();
```

Values

StackExchange.Redis中的值的类型是 **RedisValue**。与 **RedisKey** 一样，它也可以隐式转换；这意味着你大多数时候都看不到该类型，例如：

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

然而，除了文本和二进制内容外，值还可以被表示为其他原生类型，如：**Int32**，**Int64**，**Double**，**Boolean**。正因为如此，**RedisValue** 提供了大量的隐式转换支持，而 **RedisKey** 则没有这么多：

```
db.StringSet("mykey", 123); // 这仍旧是一个RedisKey 和 RedisValue
...
int i = (int)db.StringGet("mykey");
```

注意：从原生类型到 **RedisValue** 的转换都是隐式的，但是从 **RedisValue** 转换为原生类型是显示转换：如果数据没有一个适当的值，这些转换很有可能会失败。

注意：当值是数值类型时，Redis去获取一个不存在(no-existent)的键值时，该值会以数值0返回；为了一致性，nil响应被视作为0：

```
db.KeyDelete("abc");
int i = (int)db.StringGet("abc"); // 这个会被作为0返回
```

如果你需要检测nil条件，那么你可以这样做：

```
db.KeyDelete("abc");
var value = db.StringGet("abc");
bool isNil = value.IsNull; // 这个表示真
```

或者这样做更简单，使用 **Nullable**：

```
db.KeyDelete("abc");
var value = (int?)db.StringGet("abc"); // 正如你所期望的，会返回空
```

Hashes

由于哈希表中的字段名不会影响到命令的传送，他们也不是key，但是文本和二进制可以作为名字使用；因此它们被视作为值。

Channels

发布/订阅所使用的管道名字的类型是 **RedisChannel**；大致上来说和 **RedisKey** 相同，但是被独立处理的，因为管道名称是头等公民，它们不会影响命令的传输。

Scripting

Redis中的Lua脚本有两个值得注意的特性：

- 输入必须保证键值分离(在脚本内会分别变成 **KEYS** 和 **ARGV**)
- 返回的格式没有被预先定义：这取决于你的脚本

由于这个原因，**ScriptEvaluate** 方法接受两个单独的输入数组：一个是作为Key的 **RedisKey[]**，一个是作为Value的

RedisValue[]（两个都是可选的，如果省略则被假定为空）。这可能是少数几次中的一次你真的需要在你的代码中输入 **RedisKey**或**RedisValue**，而这只是因为数组变量的规则决定的。

```
1. var result = db.ScriptEvaluate(TransferScript,
2.     new RedisKey[] { from, to }, new RedisValue[] { quantity });
```

TransferScript 是一些包含Lua的字符串，在这个例子中不会展示

响应使用类型的是 **RedisResult**（这是脚本所独有的，通常API会试图尽可能直接和清晰的表示响应）。之前，**RedisResult** 提供了一系列的转换操作，甚至超过了 **RedisValue**，因为除了被解释为文本，二进制，原生类型以及可空类型，响应也可以被解释为数组，例如：

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

总结

API中使用的类型是非常刻意的被选择的，以便区分Redis的Key和Value。然而，几乎在所有情况下，你不需要直接引用所涉及的基础类型，默认提供了转换操作。

事务

Redis中的事务

Redis的事务是与SQL数据库不同的。详细了解请参考文档，转述如下：

Redis的事务：先以 **MULTI** 开始一个事务，然后将多个命令入队到事务中， 最后由 **EXEC** 命令触发事务。当碰到命令：**MULTI** (标记一个事务块的开始)，在该连接上的命令不会执行：它们会排队(调用方会得到每个队列的回复)。当遇到命令：**EXEC**(执行所有事务块内的命令)，它们被应用到一个单独的单元中(比如：没有其它连接操作之间的那个时间段)。如果是命令 **DISCARD**(取消事务，放弃执行事务块内的所有命令) 而不是 **EXEC**，那么所有的操作都会不执行(回滚)。因为命令是在事务里面排队的，所以你不能改变内部事务。

> 注意：*Redis* 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

> 一个事务从开始到执行会经历以下三个阶段：

- 开始事务。
- 命令入队。
- 执行事务。

Redis 事务命令

- DISCARD 取消事务，放弃执行事务块内的所有命令。
- EXEC 执行所有事务块内的命令。
- MULTI 标记一个事务块的开始。
- UNWATCH 取消 WATCH 命令对所有 key 的监视。
- WATCH key [key …] 监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

例如：在SQL数据库中你可能回做如下操作：

```
// 仅在它没有唯一ID的时候，分配一个唯一的ID。确保事务中没有线程竞争
var newId = CreateNewUniqueID(); // optimistic
using(var tran = conn.BeginTran())
{
    var cust = GetCustomer(conn, custId, tran);
    var uniqueId = cust.UniqueID;
    if(uniqueId == null)
    {
        cust.UniqueId = newId;
        SaveCustomer(conn, cust, tran);
    }
    tran.Complete();
}
```

在Redis中是怎么做的？

在Redis事务中这简直是不可能的是：一旦事务被开启，你不能去获取数据 — 你的操作是排队执行的。幸运的是，有另外两个命令可以帮助我们：**WATCH**和**UNWATCH**。

WATCH{key} 命令用于监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断且回滚。**EXEC** 所做的和 **DISCARD** 一样(调用方一开始就能发现并重试)。那么你能做的是：使用命令：**WATCH** 某个键，以正常的方式，来检查给定键的数据，然后使用 **MULTI/EXEC** 命令执行你的改变。当你检查数据的时候，你会发现你实际上不需要事务，你可以用 **UNWATCH** 命令用于取消 WATCH 命令对所有 key 的监视。注意：在使用 **EXEC** 和 **DISCARD** 的时候，观察键也是可以重置的(如果执行EXEC 或者DISCARD，则不需要手动执行UNWATCH)。所以在Redis层，这只是概念上的：

```
1. WATCH {custKey}
2. HEXISTS {custKey} "UniqueId"
3. (check the reply, then either:)
4. MULTI
5. HSET {custKey} "UniqueId" {newId}
6. EXEC
7. (or, if we find there was already an unique-id:)
8. UNWATCH
```

这可能看起来很奇怪：只有跨越单个操作时才可以使用 **MULTI/EXEC** 命令，但重要的是我们现在也可以使用 **{custKey}** 从所有其它的连接中来跟踪变更：如果其他人更改这个Key，那么事务会被终止。

在StackExchange.Redis又该怎么做？

更复杂的事实是StackExchange.Redis使用的是多路复用器的方式。

我们不能只让并发调用方发布 **WATCH / UNWATCH / MULTI / EXEC / DISCARD**：这应该是混合在一起的。所以一个额外的抽象被给出：另外会让使事情更简单准确：约束。约束是预定义测试包括 **WATCH** 某种类型的测试并对结果进行检查。如果所有的约束都通过了，那么要么是以 **MULTI / EXEC** 发布(从事务开始，到执行整个事务块)；要么是以 **UNWATCH** 发布(取消 WATCH 命令对所有 key 的监视)。阻止命令于其它调用方被混合在一起；所以例子可以是：

```
1. var newId = CreateNewId();
2. var tran = db.CreateTransaction();
3. tran.AddCondition(Condition.HashNotExists(custKey, "UniqueID"));
4. tran.HashSetAsync(custKey, "UniqueID", newId);
5. bool committed = tran.Execute();
6. // ^^^ 如果真：该命令会被执行；如果假：那么会回滚。
```

注意：从 **CreateTransaction** 返回的对象最后都是调用异步方法来执行命令(Execute方法最终也是调用ExecuteAsync，具体可以看源码)：由于不知道每个操作的结果，除非在 **Execute** 或 **ExecuteAsync** 操作完成后。如果操作没有被执行，所有的 **Task** 将被标记为取消，否则在命令执行后你可以获取每个正常的结果。

通过 When 的内置操作

还应该注意的是，Redis已经为我们预料到了许多常见的场景(特别是：key/hash的存在，就像上面一样)，还有单操作(single-operation)原子命令的存在。
通过 **When** 来访问，所以前面的示例也可以这样来实现：

```
1. var newId = CreateNewId();
```

```
2. bool wasSet = db.HashSet(custKey, "UniqueID", newId, When.NotExists);
```

注意：**When.NotExists** 会使用命令 **HSETNX** 而不会使用 **HSET**

Lua

你应该记住Redis 2.6及以上的版本支持Lua脚本，它可以描述为：一个常用的工具，使多个操作在服务器端的以一个原子单元执行。在使用Lua脚本的时候，由于不需要服务于其它的连接，所以它的行为更像是一个事务处理，但是没有 **MULTI / EXEC** 那么复杂。这也避免了诸如调用方和服务器端之间带宽和延迟的问题。但是代价是在脚本执行的时候独占了服务器。

在Redis层(假设 **HSETNX** 不存在)我们可以有如下实现：

```
1. EVAL "if redis.call('hexists', KEYS[1], 'UniqueId') then return redis.call('hset', KEYS[1], 'UniqueId', ARGV[1]) else
   return 0 end" 1 {custKey} {newId}
```

在 StackExchange.Redis 是这样使用的：

```
1. var wasSet = (bool) db.ScriptEvaluate(@"if redis.call('hexists', KEYS[1], 'UniqueId') then return redis.call('hset',
   KEYS[1], 'UniqueId', ARGV[1]) else return 0 end",
2.         new RedisKey[] { custKey }, new RedisValue[] { newId });
```

注意：来自 **ScriptEvaluate** 和 **ScriptEvaluateAsync** 的响应是可变的，这依赖于你所写的脚本。响应结果可以被强制转换，在这个例子中是被转换为 **bool** 类型。

事件

事件

ConnectionMultiplexer 类型公开了多个事件，可以用来了解正在发生的事件。这是非常有用的特别是在记录日志的时候：

- ConfigurationChanged 当 **ConnectionMultiplexer** 里面的连接配置被更改后触发
- ConfigurationChangedBroadcast 通过发布/订阅功能接受到一个重新配置的消息的时候；这通常是由于使用 **IServer.MakeMaster** 更改了一个节点的复制配置，也可以选择广播某个请求给所有的客户。
- ConnectionFailed 当连接失败的时候；注意：对于该连接你不会收到 **ConnectionFailed** 的通知，直到连接重新建立。
- ConnectionRestored 当重新建立连接到之前失败的那个节点的时候
- ErrorMessage 当用户发起的请求的时候，Redis服务器给出一个错误消息的响应；这是除了常规异常/故障之外，立即报告给调用方的。
- HashSlotMoved 当 “Redis集群” 表示 hash-slot 已经被迁移到节点之间的时候，注意：请求通常会被自动重新路由，所以用户不会在这里要求做任何指定的事情。
- InternalError 当Redis类库内部执行发生了某种不可预期的失败的时候；这主要是为了用于调试,大多数用户应该不需要这个事件。

注意：StackExchange.Redis 实现的 pub/sub 工作原理类似于事件，**Subscribe / SubscribeAsync** 接受一个 Action 的回调，当信息被接收到的时候该回调会被调用。

发布/订阅 消息顺序

发布/订阅 消息顺序

当使用 pub/sub API的时候，你需要做一个决定：那就是对于来自同一个连接的消息是应该按顺序处理还是应该并行处理。

按顺序处理意味着你不需要关心线程安全，并且保持了事件的顺序；消息会以完全相同的顺序接收处理(通过队列)，因此，这意味着消息能够被相互延迟。

另外一种选择是并发处理。使用并发处理 不能保证 工作处理的有序性，并且你的代码要对并行消息完全负责确保它不会破坏内部状态；使得处理速度更快并且扩展性伸缩性更好。如果消息是互不相干的，那么选择这种方式处理是特别好的。

为了安全，默认是 有序处理；但是，强烈建议你尽可能的使用并发处理。示例如下：

```
multiplexer.PreserveAsyncOrder = false;
```

这不是一个配置选项的原因是：这样做是否合理，这完全取决于你的订阅消息的代码。

KEYS，SCAN，FLUSHDB 等命令在哪里？

KEYS，SCAN，FLUSHDB 等等，这些命令在哪里？

一些常见的重复性的问题是：

> 好像没有看到如：*Key(…)* 或者 *Scan(…)* 方法？我该怎么才能查询哪些*key*是在数据库中的？

或者

> 好像也没有 *Flush(…)* 方法？我该怎样才能移除掉所有在数据库中的*key*？

The key word here, oddly enough, is the last one: database. 这句话真不知道该怎么翻译

奇怪的是这里的关键字的最后一个是：数据库。？？？？

由于StackExchange.Redis 服务的目标场景(或者宗旨)是集群服务，重要的是要知道命令所面向的数据库(可分布在多个节点的逻辑数据库)，还有命令所面向的服务器。下面这些命令都是面向单个服务器的：

- KEYS / SCAN
- FLUSHDB / FLUSHALL
- RANDOMKEY
- CLIENT
- CLUSTER
- CONFIG / INFO / TIME
- SLAVEOF
- SAVE / BGSAVE / LASTSAVE
- SCRIPT (不要与 EVAL / EVALSHA 混淆)
- SHUTDOWN
- SLOWLOG
- PUBSUB (不要与 PUBLISH / SUBSCRIBE 等命令混淆)
- 一些 DEBUG 操作

(我可能遗忘了多个命令没有列举) 其中大多数会显得很明显，但是前三行就不那么明显：

- KEYS / SCAN 不是在所有的逻辑数据库中，而是仅列出当前服务器的key;
- FLUSHDB / FLUSHALL 不是在所有的逻辑数据库中，而是仅移除当前数据库中的key;
- RANDOMKEY 不是在所有的逻辑数据库中，而是仅在当前数据库中选出一个key;

事实上，StackExchange.Redis 在使用 **IDatabase** API的时候，欺骗了 **RANDOMKEY** 命令，实际上它是以一个随机的方式选择了一台目标服务器。但这对其它的来说是不可能的。

那么我该怎么使用它们？

让我们先从一个服务器开始，而不是从一个数据库开始。

```
1. // 取得目标服务器
2. var server = conn.GetServer(someServer);
3.
4. // 在索引为0的数据库中展示出所有的key，这个key的名字必须匹配 *foo*
```

```
foreach(var key in server.Keys(pattern: "*foo*")) {
    Console.WriteLine(key);
}

// 从索引为0的数据库中清除所有的key
server.FlushDatabase();
```

注意：这个和 **IDatabase** API是不同的(在调用 **GetDatabase()** 方法的时候目标数据库已被选择)，可以给这些方法传一个可选参数来选择数据库，默认是0。

值得特别注意的是 **Keys(…)** 方法：该方法没有一个对应的 ***Async** 异步方法。原因是在后台运作，由系统确定出一个最合适的方法来使用(**KEYS** vs **SCAN**，基于服务器的版本)，并且如果可能的话将会使用 **SCAN** 方法来处理你回传的 **IEnumerable** 参数，内部会对该参数分页处理：所以你绝不会看到游标操作的详细实现。如果 **SCAN** 是不可用的，那么将会使用 **KEYS**，这个可能会在服务器导致阻塞。无论哪种方式，**SCAN** 和 **KEYS** 都需要扫描整个键空间，所以应该避免在生产服务器上使用；或者至少值针对从服务器。

那么我需要记住我所连接的服务器吗？ 那简直糟透了！

不完全是这样的，你可以使用 **coon.GetEndPoints()** 方法列出端点(要么是所有已知的，要么是在原始配置中指定的：这未必是同一个事情)，还可以迭代 **GetServer()** 方法去找到你想要的服务器(例如：选择一个从服务器)。

性能分析

性能分析

StackExchange.Redis 公开了少量的方法和类型来开启性能分析。由于其异步性和多路复用行为，性能分析是一个有点复杂的话题。

接口

性能分析接口是由这些组成的：**IProfiler**，**ConnectionMultiplexer.RegisterProfiler(IProfiler)**，**ConnectionMultiplexer.BeginProfiling(object)**， **ConnectionMultiplexer.FinishProfiling(object)** 还有**IProfiledCommand**。

你可以用 **ConnectionMultiplexer** 的实例来注册一个 **IProfiler** 接口，注册后它不能被更改。通过调用 **BeginProfiling(object)** 方法开始分析一个给定的上下文(例如：Thread，HttpRequest等等)，然后调用 **FinishProfiling(object)** 方法完成分析；**FinishProfiling(object)** 方法返回一个 **IProfiledCommand** 的集合，该集合包含计时信息的所有命令都被发送到Redis；使用给定的上下文参数，通过已配置的 **ConnectionMultiplexer** 对像来调用 **(Begin|Finish)Profiling (**也就是**BeginProfiling & FinishProfiling)** 方法。

在具体的应用中什么样的 “上下文” 对象应该使用。

Available Timings

StackExchange.Redis公共的信息有：

- 涉及的Redis服务器
- 对Redis数据库的查询
- 运行的Redis命令
- 路由命令的使用标志
- 命令的初始化创建时间
- 用了多长时间来排队命令
- 在排队之后，用了多长时间来发送命令
- 在命令被发送后，用了多长时间接受来自Redis的响应
- 在接受响应后，用了多长时间来处理响应
- 如果命令被发送以回应一个集群 ASK 或 MOVED 的响应
  - 如果这样，那么原始的命令的 **TimeSpan** 是高精确度的， 如果运行时支持。**DateTime** 和 **DateTime.UtcNow** 精确度是一样的。

选择上下文

由于StackExchange.Redis的异步接口，分析需要外部协助来组织相关的命令。开始分析和结束分析都是通过给定的上下文对象来实现的(通过 **BeginProfiling(object) & FinishProfiling(object)** 方法实现)，通过 **IProfiler** 接口的 **GetContext** 方法取得上下文对象。

下面是一个从很多不同的线程发出相关命令的示例：

```
1. class ToyProfiler : IProfiler
2. {
```

```
    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();

//注册实现了IProfiler接口的对象
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);
}

//分析开始
conn.BeginProfiling(thisGroupContext);

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

```

```
//分析结束，并且返回了含定时信息的所有命令集合
IEnumerable<IProfiledCommand> timings = conn.FinishProfiling(thisGroupContext);
```

在结束后，**timings** 包含了16,000个 **IProfiledCommand** 对象：每一个命令都会被发送到Redis。

替代方案，你可以按照如下做：

```
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 最终会包含16项1,000个 **IProfilingCommand** 记录，以线程作为键来获取perThreadTimings集合中的值来发送它们。

让我们忘记玩具示例，这里展示的是一个在MVC5应用中配置StackExchange.Redis的示例：

首先注册 **IProfiler** 接口，而不是 **ConnectionMultiplexer** ：

```
public class RedisProfiler : IProfiler
{
    const string RequestContextKey = "RequestProfilingContext";

    public object GetContext()
    {
        var ctx = HttpContext.Current;
        if (ctx == null) return null;

        return ctx.Items[RequestContextKey];
    }

    public object CreateContextForCurrentRequest()
    {
        var ctx = HttpContext.Current;
        if (ctx == null) return null;

        object ret;
        ctx.Items[RequestContextKey] = ret = new object();

        return ret;
    }
}
```

那么，添加下面的代码到你的Global.asax.cs文件中:

```
protected void Application_BeginRequest()
{
    var ctxObj = RedisProfiler.CreateContextForCurrentRequest();
    if (ctxObj != null)
    {
        RedisConnection.BeginProfiling(ctxObj);
    }
}

protected void Application_EndRequest()
{
    var ctxObj = RedisProfiler.GetContext();
    if (ctxObj != null)
    {
        var timings = RedisConnection.FinishProfiling(ctxObj);

        // 在这里你可以使用`timings`做你想做的
    }
}
```

这些实现会组织所有的Redis命令，包括 **async/await** 并随着http请求初始化它们。

脚本

脚本

我们通过 **IServer.ScriptLoad(Async)**, **IServer.ScriptExists(Async)**, **IServer.ScriptExists(Async)**, **IDatabase.ScriptEvaluate**, 还有 **IDatabaseAsync.ScriptEvaluateAsync** 方法来执行 **Lua脚本**，使用这些方法提交执行Lua脚本到Redis。

可以使用 **LuaScript** 类来实现更复杂Lua脚本。**LuaScript** 类使脚本的编写和参数的提交更加简单，并且允许你使用更清晰的变量名。

LuaScript 的示例如下:

```
const string Script = "redis.call('set', @key, @value)";

using (ConnectionMultiplexer conn = /* init code */)
{
    var db = conn.GetDatabase(0);

    var prepared = LuaScript.Prepare(Script);
    db.ScriptEvaluate(prepared, new { key = (RedisKey)"mykey", value = 123 });
}
```

LuaScript 类重写了脚本中形式为: @myVar的变量，使之以符合 **ARGV[someIndex]** 的需要。如果传递的参数是 **RedisKey** 类型，它会自动的作为 **KEYS** 集合的一部分发送。

任何对象的公开字段或者属性成员可以在Lua脚本中以@为前缀(使用与公开成员相同的名字，例如: 类里面有个name的属性，那么变量就是@name)的变量来使用，且被作为 **Evaluate** 调用的Hash参数。
成员类型可以是:

- int(?)
- long(?)
- double(?)
- string
- byte[]
- bool(?)
- RedisKey
- RedisValue

为了避免重新发送Lua脚本到Redis，通过调用 **LuaScript.Load(IServer)** 方法可以将 **LuaScript** 对象转换为 **LoadedLuaScript**。**LoadedLuaScript** 可以执行 **EVALSHA** 命令(在脚本比较长的情况下，如果每次调用脚本都需要将整个脚本传给Redis会占用较多的带宽。为了解决这个问题，Redis提供了EVALSHA命令，允许开发者通过脚本内容的SHA1摘要来执行脚本，该命令的用法和EVAL一样，只不过是将脚本内容替换成脚本内容的SHA1摘要。)。

LoadedLuaScript 的示例如下:

```
 const string Script = "redis.call('set', @key, @value)";

using (ConnectionMultiplexer conn = /* init code */)
{
```

```
    var db = conn.GetDatabase(0);
    var server = conn.GetServer(/* appropriate parameters*/);

    var prepared = LuaScript.Prepare(Script);
    var loaded = prepared.Load(server);
    loaded.Evaluate(db, new { key = (RedisKey)"mykey", value = 123 });
}
```

LuaScript 和 **LoadedLuaScript** 的所有方法都有异步方法(*Async)的实现，我们可以调用Evaluate/EvaluateAsync方法把 **ExecutableScript** 属性的值(Lua脚本)提交到Redis。