目　录

致谢

当前文档 《StackExchange.Redis 中文使用文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-10。

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，如果您在日常生活、工作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-docs-cn

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 中文使用文档

StackExchange.Redis 中文使用文档

Intro

翻译 StackExchange.Redis 的文档

原文文档在线地址： https://stackexchange.github.io/StackExchange.Redis/

Redis 简介

Redis是一个使用ANSI C编写的开源、支持网络、基于内存、可选持久性的键值对存储数据库。
从2015年6月开始，Redis的开发由Redis Labs赞助，而2013年5月至2015年6月期间，其开发由Pivotal赞助。在2013年5月之前，其开发由VMware赞助。
根据月度排行网站DB-Engines.com的数据显示，Redis是最流行的键值对存储数据库。
更多介绍可参考 https://zh.wikipedia.org/wiki/Redis

Redis官网 https://redis.io/

StackExchange.Redis 简介

StackExchange.Redis 是 Stackoverflow 开发的 Redis C# 客户端，是目前.net应用使用的最多的 redis 客户端，性能优越。

StackExchange.Redis中文使用文档

- Github: https://weihanli.github.io/StackExchange.Redis-docs-cn/

- Gitbook : https://www.gitbook.com/book/weihanli/stackexchange-redis-docs-cn/details
  - 点击阅读
  - 下载 PDF

目录

More

作者水平有限，若有疏漏或错误还望提醒，十分感谢。

您可以 提出问题 或者给我 发邮件。

Contact me:weihanli@outlook.com

基础

基本使用

StackExchange.Redis 中核心对象是在 `StackExchange.Redis` 命名空间中的 `ConnectionMultiplexer` 类，这个对象隐藏了多个服务器的详细信息。
因为 `ConnectionMultiplexer` 要做很多事，它被设计为在调用者之间可以共享和重用。
你不应该在执行每一个操作的时候就创建一个 `ConnectionMultiplexer` . 它完全是线程安全的，并准备好这种用法（多线程）。
在后续所有的例子中，我们假设你有一个 `ConnectionMultiplexer` 类的实例保存以重用。
但现在，让我们来先创建一个。 这是使用 `ConnectionMultiplexer.Connect` 或 `ConnectionMultiplexer.ConnectAsync` 完成的，传递配置字符串或 `ConfigurationOptions` 对象。
配置字符串可以采用逗号分隔的一系列节点的形式，所以让我们在默认端口（6379）上连接到本地机器上的一个实例：

```
using StackExchange.Redis;
...
ConnectionMultiplexer redis = ConnectionMultiplexer.Connect("localhost");
// ^^^ store and re-use this!!!
```

请注意， `ConnectionMultiplexer` 实现了 `IDisposable` 接口而且可以在不再需要的时候处理释放掉。
这是故意不展示使用 `using` 语句用法，因为你想要只是简单地使用一个 `ConnectionMultiplexer` 的情况是极少见的 ，因为想法是重用这个对象。

更复杂的情况可能涉及主/从设置；对于此用法，只需简单的指定组成逻辑redis层的所有所需节点（它将自动标识主节点）

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

如果它发现两个节点都是主节点，则可以可选地指定可以用于解决问题的仲裁密钥，然而幸运地是这样的条件是非常罕见的。

一旦你有一个 `ConnectionMultiplexer` ，你可能有3个主要想做的事：

- 访问一个 redis 数据库（注意，在集群的情况下，单个逻辑数据库可以分布在多个节点上）
- 使用 redis 的的 发布/订阅 功能
- 访问单独的服务器以进行维护/监视

使用 redis 数据库

访问redis数据库非常简单：

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

从 `GetDatabase` 返回的对象是一个成本很低的通道对象，不需要存储。
注意，redis支持多个数据库（虽然“集群”不支持），这可以可选地在调用 `GetDatabase` 中指定。
此外，如果您计划使用异步API，您需要 `Task.AsyncState` 有一个值，也可以指定：

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

一旦你有了 `IDatabase` ，它只是一个使用 redis API 的情况。
注意，所有方法都具有同步和异步实现。
根据微软的命名指导，异步方法都以 `...Async(...)` 结尾，并且完全是可以等待的 `await` 等。

最简单的操作也许是存储和检索值：

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

需要注意的是，这里的 `String...` 前缀表示 redis 的 String 类型，尽管它和 .NET 的字符串类型 都可以保存文本， 它们还是有较大差别的。
然而，redis 还允许键和值使用原始的二进制数据，用法和字符串是一样的：

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

覆盖所有redis数据类型的所有 [redis数据库命令]（http://redis.io/commands）的都是可以使用的。

使用 redis 发布/订阅

redis的另一个常见用法是作为 发布/订阅消息分发工具；
这也很简单，并且在连接失败的情况下， `ConnectionMultiplexer` 将处理重新订阅所请求的信道的所有细节。

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

同样，从 `GetSubscriber` 返回的对象是一个不需要存储的低成本的通道对象。
发布/订阅 API没有数据库的概念，但和之前一样，我们可以选择的提供一个异步状态。
注意，所有订阅都是全局的：它们不限于 `ISubscriber` 实例的生命周期。
redis中的 发布/订阅 功能使用命名的“channels”； channels 不需要事先在服务器上定义（这里有一个有趣的用法是利用每个用户的通知渠道类驱动部分的实时更新）
如在.NET中常见的，订阅采用回调委托的形式，它接受通道名称和消息：

```
1. sub.Subscribe("messages", (channel, message) => {
2.     Console.WriteLine((string)message);
3. });
```

另外（通常在一个单独的机器上的一个单独的进程），你可以发布到该通道：

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

这将（实际上瞬间）将“hello”写到订阅进程的控制台。 和之前一样，通道名和消息都可以是二进制的。

有关顺序和并发消息处理的使用文档说明，请参见 发布/订阅消息顺序 。

访问单独的服务器

出于维护目的，有时需要发出服务器特定的命令：

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

`GetServer` 方法会接受一个 `EndPoint` .aspx）终结点或者是一个可以唯一标识一个服务器的键/值对。

像之前介绍的那样， `GetServer` 方法返回的是一个不需要被存储的轻量级的通道对象，并且可以可选的指定 async-state（异步状态）。
需要注意的是，多个可用的节点也是可以的：

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

一个 `IServer` 实例是可以使用服务器命令的 Server commands，例如：

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

同步 vs 异步 vs 执行后不理

StackExchange.Redis有3种主要使用机制：

- 同步 - 适用于操作在方法返回到调用者之前完成（注意，尽管这可能阻止调用者，但它绝对不会阻止其他线程：StackExchange.Redis的关键思想是它积极地与并发调用者共享连接）
- Asynchronous - where the operation completes some time in the future, and a `Task` or `Task<T>` is returned immediately, which can later:
- 异步 - 操作在将来完成一些时间，并且立即返回一个 `Task` 或‘Task‘对象，也可以稍后再返回：
    - 是可以等待的（阻塞当前线程，直到响应可用） `.Wait()`
    - 可以增加一个后续的回调（TPL 中的 `ContinueWith` .aspx))
    - *awaited* 可等待的（这是简化后者的语言级特性，同时如果答复已经知道也立即继续）
- 执行后不理 - 适用于你真的对这个回复不感兴趣，并且乐意继续不管回应

同步的用法已经在上面的示例中展示了。这是最简单的用法，并且不涉及 TPL。

对于异步使用，关键的区别是方法名称上的 `Async` 后缀，以及（通常）使用“await”语言特性。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"
```

执行后不理 的用法可以通过所有方法上的可选参数 `CommandFlags flags` （默认值为 `null` ）来访问使用。

这种用法中，方法会立即方法一个默认值（所以通常返回一个 `String` 的方法总是返回 `null`，而一个通常返回一个 `Int64` 的方法总是返回 `0`）。
操作会在后台继续执行。这种情况的典型用例可能是增加页面浏览数量：

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

查看原文

配置

配置

因为有很多不同配置 redis 的方式，StackExchange.Redis 提供了一个丰富的配置模型，当调用 `Connect` （或 `ConnectAsync` ）时调用它。

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

这里的 `configuration` 可以是下面的任意一个：

- 一个 `ConfigurationOptions` 实例
- 一个代表配置的 `string`

后者是 基本上 是前者的标记化形式。

基本配置字符串
-

最简单的 配置示例只需要一个主机名：

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

这将使用默认的redis端口（6379）连接到本地计算机上的单个服务器。
附加选项只是简单地附加（逗号分隔）。 端口通常用冒号（ `:` ）表示。 配置选项在名称后面包含一个 `=` 。 例如：

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

下面显示了 `string` 和 `ConfigurationOptions` 表示之间的映射概述，但您可以轻松地在它们之间切换：

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

或者：

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

常见的用法是将 基础配置 细节存储在一个字符串中，然后在运行时应用特定的详细信息：

```
string configString = GetRedisConfiguration();
var options = ConfigurationOptions.Parse(configString);
options.ClientName = GetAppName(); // only known at runtime
options.AllowAdmin = true;
conn = ConnectionMultiplexer.Connect(options);
```

带密码的 Microsoft Azure Redis 示例

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

配置选项

`ConfigurationOptions` 对象具有许多的属性，所有这些都在智能提示中都有。

一些更常用的选项包括：

配置字符串	`ConfigurationOptions`	默认值
abortConnect={bool}	`AbortOnConnectFail`	`true` (Azure 上默认值为 `false`)
allowAdmin={bool}	`AllowAdmin`	`false`
channelPrefix={string}	`ChannelPrefix`	`null`
connectRetry={int}	`ConnectRetry`	`3`
connectTimeout={int}	`ConnectTimeout`	`5000`
configChannel={string}	`ConfigurationChannel`	`__Booksleeve_MasterChanged`

configCheckSeconds={int}	`ConfigCheckSeconds`	`60`
defaultDatabase={int}	`DefaultDatabase`	`null`
keepAlive={int}	`KeepAlive`	`-1`
name={string}	`ClientName`	`null`
password={string}	`Password`	`null`
proxy={proxy type}	`Proxy`	`Proxy.None`
resolveDns={bool}	`ResolveDns`	`false`
serviceName={string}	`ServiceName`	`null`
ssl={bool}	`Ssl`	`false`
sslHost={string}	`SslHost`	`null`
syncTimeout={int}	`SyncTimeout`	`1000`
tiebreaker={string}	`TieBreaker`	`__Booksleeve_TieBreak`
version={string}	`DefaultVersion`	`(3.0 in Azure, else 2.0)`
writeBuffer={int}	`WriteBuffer`	`4096`
ReconnectRetryPolicy={IReconnectRetryPolicy}	`ReconnectRetryPolicy`	重新连接重试策略

补充的只有代码才支持的选项:

> *ReconnectRetryPolicy (IReconnectRetryPolicy) - Default: ReconnectRetryPolicy = LinearRetry(ConnectTimeout);*

- 重连重试策略（`IReconnectRetryPolicy`）。默认设置：`ReconnectRetryPolicy = LinearRetry(ConnectTimeout);`

配置字符串中的令牌是逗号分隔的;任何没有 `=` 符号的都假定为redis服务器端点。
没有显式端口的端点将在未启用ssl的情况下使用6379，如果启用了ssl则使用6380。

以 `$` 开头的令牌被用来表示命令映射，例如： `$ config = cfg` 。

自动和手动配置

在许多常见的情况下，StackExchange.Redis将自动配置很多设置，包括服务器类型和版本，连接超时和主/从关系。
有时，在redis服务器上禁用了这些命令。 在这种情况下，可以提供更多的信息：

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

它相当于命令字符串：

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

重命名命令

redis的一个很不寻常的功能是可以禁用或重命名或禁用并重命名单个命令。

根据前面的例子，这是通过 `CommandMap` 来实现的，但不是传递一个 `HashSet <string>` 到 `Create()` （表示可用或不可用的命令），而是传递一个 `Dictionary < string>` 。

字典中未提及的所有命令都默认已启用且未重命名。

“null”或空白值记录命令被禁用。 例如：

```
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),
    // ...
}
```

以上代码等同于（在连接字符串中）：

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

Twemproxy

Twemproxy 是允许使用多个 redis 实例就像它是一个单个服务器，具有内置分片和容错（很像 redis 集群，但单独实现）的工具。
Twemproxy可用的功能集减少。
为了避免手动配置，可以使用 `Proxy` 选项：

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

Tiebreakers 和配置更改公告

通常 StackExchange.Redis 都会自动解析 主/从节点。然而，如果你不使用诸如 redis-sentinel 或 redis 集群之类的管理工具，有时你会有多个主节点（例如，在重置节点以进行维护时，它可能会作为主节点重新显示在网络上）。
为了帮助解决这个问题，StackExchange.Redis 可以使用 *tie-breaker* 的概念 - 这仅在检测到多个主节点时使用（不包括需要多个主节点的redis集群）。
为了与 BookSleeve 兼容，tiebreaker 使用默认的key为 `"__Booksleeve_TieBreak"` （总是在数据库0中）。
这用作粗略的投票机制，以帮助确定首选 主机，以便能够按正确的路由工作。

同样，当配置改变时（特别是主/从配置），连接的实例使得他们意识到新的情况（在可用的地方通过 `INFO`，`CONFIG` 等进行通知 ）是很重要的。
StackExchange.Redis 通过自动订阅可以在其上发送这样的通知的发布/订阅通道来实现。
由于类似的原因，这默认为 `"__Booksleeve_MasterChanged"` 。

这两个选项都可以通过 `.ConfigurationChannel` 和 `.TieBreaker` 配置属性来定制或禁用（设置为 `""` ）。

These settings are also used by the `IServer.MakeMaster()` method, which can set the tie-breaker in the database and broadcast 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.PublishReconfigure` method.

这些设置也由 `IServer.MakeMaster()` 方法使用，它可以设置数据库中的 tie-breaker 并广播配置更改消息。
配置消息也可以单独用于主/从变化，只需通过调用 `ConnectionMultiplexer.PublishReconfigure` 方法请求所有节点刷新其配置。

重新连接重试策略

当连接由于任何原因丢失时，StackExchange.Redis会自动尝试在后台重新连接。
它将继续重试，直到连接恢复。 它将使用 ReconnectRetryPolicy 来决定在重试之间应该等待多长时间。
ReconnectRetryPolicy可以是线性的（默认），指数的或者是一个自定义的重试策略。

举个例子：

```
config.ReconnectRetryPolicy = new ExponentialRetry(5000); // defaults maxDeltaBackoff to 10000 ms
//retry#    retry to re-connect after time in milliseconds
//1           a random value between 5000 and 5500
//2           a random value between 5000 and 6050
//3           a random value between 5000 and 6655
//4           a random value between 5000 and 8053
//5           a random value between 5000 and 10000, since maxDeltaBackoff was 10000 ms
//6           a random value between 5000 and 10000

config.ReconnectRetryPolicy = new LinearRetry(5000);
//retry#    retry to re-connect after time in milliseconds
//1           5000
//2           5000
```

```
14. //3            5000
15. //4            5000
16. //5            5000
17. //6            5000
```

查看原文

事件

事件

`ConnectionMultiplexer` 类型提供了许多事件可以用来理解被封装的底层是怎么工作的。这在记录日志时会特别有用。

- `ConfigurationChanged` - 当连接的配置从 `ConnectionMultiplexer` 内部发生修改时触发
- `ConfigurationChangedBroadcast` - 当经由发布/订阅接收到重新配置消息时引发；这通常是由于 `IServer.MakeMaster` 用于更改节点的复制配置，可以选择将这样的请求广播到所有客户端
- `ConnectionFailed` - 当连接由于无论任何原因失败时触发；请注意，在连接重新建立之前是不会再收到该连接的 `ConnectionFailed` 通知
- `ConnectionRestored` - 当重新建立到先前失败的节点的连接时触发
- `ErrorMessage` - 当redis服务器响应任何用户发起的具有错误消息的请求时触发；这种情况不包含将被报告给直接调用者的常规异常/故障的情况
- `HashSlotMoved` - 当“redis集群”指出 散列槽（hash-slot） 已在节点之间迁移时触发；请注意，请求通常会自动重新路由，因此用户不需要在这里做任何特殊操作
- `InternalError` - 当库在一些意想不到的方式失败时触发；这主要是为了调试目的，并且大多数用户应该不需要这个事件

需要注意，StackExchange.Redis 中的 发布/订阅 工作方式与事件非常相似，接收到消息时会调用接受一个 `Action<RedisChannel, RedisValue>` 类型回调方法的 `Subscribe` / `SubscribeAsync` 方法。

查看原文

同步执行

连续同步执行的危险

一旦遇到这样的问题，这里还有更多内容，然后发现了 一个适当恶劣的解决方法。
这篇文章没有列在索引中，但是为满足你的好奇心而保留了下来。

查看原文

键查找

`KEYS`, `SCAN`, `FLUSHDB` 这些在哪里？

这里是一些非常常见的常见问题：

> 似乎没有一个 `Keys(...)` 或者 `Scan(...)` 方法？ 如何查询数据库中存在哪些键？

或者

> 似乎没有一个 `Flush(...)` 方法？如何删除数据库中的所有的键？

很奇怪的是，这里的最后一个关键词是数据库。
因为StackExchange.Redis的目标是针对集群等场景，知道哪些命令针对 数据库 （可以是分布在多个节点上的逻辑数据库）以及哪些命令针对 服务器 是很重要的。
以下命令都针对单个服务器：

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

（我可能错过了至少一个）大多数这些将显得很明显，但前3行不那么明显：

- `KEYS` / `SCAN` 只列出当前服务器上的键； 而不是更广泛的逻辑数据库
- `FLUSHDB` / `FLUSHALL` 只删除当前服务器上的密钥;而不是更广泛的逻辑数据库
- `RANDOMKEY` 仅选择当前服务器上的密钥； 而不是更广泛的逻辑数据库

实际上，StackExchange.Redis 通过简单地随机选择目标服务器来欺骗 `IDatabase` API上的

`RANDOMKEY`，但这对其他服务器是不可能的。

那么如何使用它们呢？

最简单的：从服务器开始，而不是数据库。

```
// 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 ALL keys from database 0
server.FlushDatabase();
```

注意，与 `IDatabase` API（在 `GetDatabase()` 调用中已经选择了的目标数据库）不同，这些方法对数据库使用可选参数，或者默认为 `0`。

`Keys(...)` 方法值得特别一提：它并不常见，因为它没有一个 `*Async` 对应。 这样做的原因是，在后台，系统将确定使用最合适的方法（基于服务器版本的 `KEYS` VS `SCAN`），如果可能的话，将使用 `SCAN` 方法 一个 `IEnumerable<RedisKey>` 在内部执行所有的分页 - 所以你永远不需要看到游标操作的实现细节。
如果 `SCAN` 不可用，它将使用 `KEYS`，这可能导致服务器上的阻塞。 无论哪种方式，`SCAN` 和 `KEYS` 都需要扫描整个键空间，所以在生产服务器上应该避免 - 或者至少是针对从节点服务器。

所以我需要记住我连接到哪个服务器？ 这真糟糕！

不，不完全是。 你可以使用 `conn.GetEndPoints()` 来列出节点（所有已知的节点，或者在原始配置中指定的节点，这些不一定是相同的东西），并且使用 `GetServer()` 迭代找到想要的服务器（例如，选择一个从节点）。

查看原文

键与值

键，值和通道

在处理redis时，是键不是键之间有很重要的区别。
键是数据库中数据片段（可以是String，List，Hash或任何其他redis数据类型）的独一无二的名称。
键永远不会被解释为…好吧，任何东西：它们只是惰性名称。
此外，当处理集群或分片系统时，它是定义包含此数据的节点（或者如果有从节点的节点）的关键 - 因此键对于路由命令是至关重要的。

这与 值 形成对比； 值是单独（对于字符串数据）或分组对键的内容存储 。
值不影响命令路由 （注意：使用 SORT 命令时除非指定 BY 或者 GET，否则是很难解释的）

同样，为了操作的目的，值通常被redis翻译为：

- `incr` （和各种类似的命令）将String值转换为数值数据
- 排序可以使用数字或unicode规则解释值
- 和许多其他操作

关键是使用API需要理解什么是键，什么是值。
这反映在StackExchange.Redis API中，但是好消息是，大部分时间你根本不需要知道这一点。

当使用 发布/订阅 时，我们处理 *channels* ； channel 不会影响路由（因此它们不是密钥），但与常规值非常不同，因此要单考虑。

键

StackExchange.Redis 通过 `RedisKey` 类型表示键。
好消息是，可以从 `string` 和 `byte[]` 的隐式转换，允许使用文本和二进制密钥，没有任何复杂性。

例如， `StringIncrement` 方法使用一个 `RedisKey` 作为第一个参数，但是你不需要知道 ；

举个例子：

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

or

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

同样，有一些操作返回 键为 `RedisKey` - 再次，它依然可以自动隐式转换：

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

值

StackExchange.Redis 用 `RedisValue` 类型表示值。 与 `RedisKey` 一样，存在隐式转换，这意味着大多数时候你从来没有看到这种类型，例如：

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

然而，除了文本和二进制内容，值还可能需要表示类型化的原始数据 - 最常见的（在.NET术语中） `Int32` ， `Int64` ， `Double` 或 `Boolean` 。 因此， `RedisValue` 提供了比 `RedisKey` 更多的转换支持：

```
db.StringSet("mykey", 123); // this is still a RedisKey and RedisValue
...
int i = (int)db.StringGet("mykey");
```

请注意，虽然从基元类型到 `RedisValue` 的转换是隐式的，但是从 `RedisValue` 到基元类型的许多转换是显式的：这是因为如果数据没有合适的值，这些转换很可能会失败。

另外注意，当做数字 处理时，redis将不存在的键视为零； 为了与此一致，将空响应视为零：

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

如果您需要检测空状态，那么你就可以这样检查：

```
db.KeyDelete("abc");
var value = db.StringGet("abc");
bool isNil = value.IsNull; // this is true
```

或者更简单地，只是使用提供的 `Nullable <T>` 支持：

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

哈希

由于哈希中的字段名称不影响命令路由，它们不是键，但可以接受文本和二进制名称， 因此它们被视为用于API目的的值。

通道

发布/订阅 的通道名称由 `RedisChannel` 类型表示； 这与 `RedisKey` 大体相同，但是是独立处理的，因为虽然通道名是正当的第一类元素，但它们不影响命令路由。

脚本

redis中的脚本 有两项显著的特性：

- 输入必须保持键和值分离（在脚本内部分别成为 `KEYS` 和 `ARGV` ）
- 返回格式未预先定义：这将特定于您的脚本

正因为如此， `ScriptEvaluate` 方法接受两个独立的输入数组：一个用于键的 `RedisKey []` ，一个用于值的 `RedisValue []` （两者都是可选的，如果省略则假定为空）。 这可能是你实际需要在代码中键入 `RedisKey` 或 `RedisValue` 的少数几次之一，这只是因为数组变动规则：

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

（其中 `TransferScript` 是一些包含Lua的 `string` ，在这个例子中没有显示）

响应使用 `RedisResult` 类型（这是脚本专用的;通常API尝试尽可能直接清晰地表示响应）。 和前面一样， `RedisResult` 提供了一系列转换操作 - 实际上比 `RedisValue` 更多，因为除了可以转换为文本，二进制，一些基元类型和可空元素，响应也可以转换为 数组 ，例如：

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

结论

API中使用的类型是非常故意选择的，以区分redis *keys* 和 *values*。 然而，在几乎所有情况下，您不需要直接去参考所涉及的底层类型，因为提供了转换操作。

查看原文

管道与重用链接

管道和链接复用

延迟严重。现代计算机可以以惊人的速度搅动数据，并且高速网络（通常具有在重要服务器之间的多个并行链路）提供巨大的带宽，但是… 该延迟意味着计算机花费大量的时间等待数据 和 这是基于连续的编程越来越受欢迎的几个原因之一。

让我们考虑一些常规的程序代码：

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

在涉及的步骤方面，这看起来大致是这样：

```
[req1]                          # client: the client library constructs request 1
     [c=>s]                     # network: request one is sent to the server
          [server]              # server: the server processes request 1
                [s=>c]          # network: response one is sent back to the
client
                     [resp1]  # client: the client library parses response 1
                           [req2]
                                [c=>s]
                                     [server]
                                           [s=>c]
                                                [resp2]
```

现在让我们突出客户端正在做的一些事的时间点：

```
[req1]
     [====waiting=====]
                     [resp1]
                           [req2]
                                [====waiting=====]
                                                [resp2]
```

请记住，这是不成比例的 - 如果这是按时间缩放，它将是完全由 `waiting` 控制的。

管道线

因此，许多redis客户端允许你使用 *pipelining*， 这是在管道上发送多个消息而不等待来自每个的答复的过程 - 并且（通常）稍后当它们进入时处理答复。
在 .NET 中，可以启动但尚未完成，并且可能以后完成或故障的由TPL封装的操作可以由 TPL 通过 `Task` / `Task <T>` API 来实现。
本质上， `Task<T>` 表示 “ `T` 类型未来可能的值”（非泛型的 `Task` 本质尚是 `Task<void>` ）。你可以使用任意一种用法：

- 在稍后的代码块等待直到操作完成（ `.Wait()` ）
- 当操作完成时，调度一个后续操作（ `.ContinueWith(...)` 或 `await` ）

例如，要使用过程化（阻塞）代码来借助管道传递这两个 get 操作，我们可以使用：

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

注意，我在这里使用 `db.Wait` ，因为它会自动应用配置的同步超时，但如果你喜欢你也可以使用 `aPending.Wait()` 或 `Task.WaitAll(aPending,bPending);` 。
使用管道技术，我们可以立即将这两个请求发送到网络，从而消除大部分延迟。
此外，它还有助于减少数据包碎片：单独发送（等待每个响应）的20个请求将需要至少20个数据包，但是在管道中发送的20个请求可以通过少得多的数据包（也许只有一个）。

执行后不理

管道的一个特例是当我们明确地不关心来自特定操作的响应时，这允许我们的代码在排队操作在后台继续时立即继续。 通常，这意味着我们可以在单个调用者的连接上并发工作。 这是使用 `flags` 参数来实现的：

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

`FireAndForget` 标志使客户端库正常地排队工作，但立即返回一个默认值（因为 `KeyExpire` 返回一个 `bool` ，这将返回 `false` ，因为 `default(bool)` 是 `false` - 但是返回值是无意义的，应该忽略）。
这也适用于 `*Async` 方法：一个已经完成的 `Task <T>` 返回默认值（或者为 `void` 方法返回

一个已经完成的 `Task`)。

复用链接

管道是很好的，但是通常任何单个代码块只需要一个值（或者可能想要执行几个操作，但是依赖于彼此）。
这意味着我们仍然有一个问题，我们花大部分时间等待数据在客户端和服务器之间传输。
现在考虑一个繁忙的应用程序，也许是一个Web服务器。
这样的应用程序通常是并发的，所以如果你有20个并行应用程序请求都需要数据，你可能会想到旋转20个连接，或者你可以同步访问单个连接（这意味着最后一个调用者需要等待 延迟的所有其他19之前，甚至开始）。 或者折中一下，也许一个5个连接的租赁池 - 无论你怎么做，都会有很多的等待。
StackExchange.Redis不做这个； 相反，它做 很多 的工作，使你有效地利用所有这个空闲时间复用 一个连接。
当不同的调用者同时使用它时，它自动把这些单独的请求加入管道，所以不管请求使用阻塞还是异步访问，工作都是按进入管道的顺序处理的。
因此，我们可能有10或20个的“get a 和 b”包括此前的（从不同的应用程序请求）情景中，并且他们都将尽快到达连接。
基本上，它用完成其他调用者的工作的时间来填充 `waiting` 时间。

因此，StackExchange.Redis不提供的唯一redis特性（不会提供）是“阻塞弹出”（BLPOP，BRPOP 和 BRPOPLPUSH），因为这将允许单个调用者停止整个多路复用器，阻止所有其他调用者 。
StackExchange.Redis 需要保持工作的唯一其他时间是在验证事务的前提条件时，这就是为什么 StackExchange.Redis 将这些条件封装到内部管理的 `condition` 实例中。

在这里阅读更多关于事务。
如果你觉得你想“阻止出栈”，那么我强烈建议你考虑 发布 / 订阅 代替：

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

这实现了相同的目的，而不需要阻塞操作。 注意：

- 数据 不通过 发布 / 订阅 发送； 发布 / 订阅 API只用于通知处理器检查更多的工作
- 如果没有处理器，则新项目保持缓存在列表中； 工作不会丢失
- 只有一个处理器可以弹出单个值； 当消费者比生产者多时，一些消费者会被通知，然后发现没有

什么可做的

- 当你重新启动一个处理器，你应该假设 有工作，以便你处理任何积压的任务
- 但除此之外，语义与阻止出栈相同

StackExchange.Redis 的多路复用特性使得在使用常规的不复杂代码的同时在单个连接上达到极高的吞吐量成为可能。

并发

应当注意，管道/链接复用器/未来值 方法对于基于连续的异步代码也很好地起作用；例如你可以写：

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

查看原文

分析

分析

StackExchange.Redis公开了一些方法和类型来启用性能分析。 由于其异步和多路复用表现分析是一个有点复杂的主题。

接口

分析接口由 `IProfiler` , `ConnectionMultiplexer.RegisterProfiler(IProfiler)` , `ConnectionMultiplexer.BeginProfiling(object)` , `ConnectionMultiplexer.FinishProfiling(object)` 和 `IProfiledCommand` 。

你可以用一个 `ConnectionMultiplexer` 实例注册一个 `IProfiler` ，它不能被改变。你可以通过调用 `BeginProfiling(object)` 开始分析一个给定的上下文对象（例如，线程，Http请求等等），调用 `FinishProfiling(object)` 来完成。 `FinishProfiling(object)` 返回一个 `IProfiledCommand` 集合的对象，它包含了通过 `(Begin|Finish)Profiling` 调用和给定上下文配置好的 `ConnectionMultiplexer` 对象发送到 redis 的所有命令的时间信息。

应该使用什么“上下文”对象是应用程序来确定的。

可用时间

StackExchange.Redis显示有关以下内容的信息:

- 涉及到的redis服务器
- 正在查询的redis数据库
- redis命令运行
- 用于路由命令的标志
- 命令的初始创建时间
- 使命令进入队列所需的时间
- 命令入队后，发送命令需要多长时间

- 发送命令后，从redis接收响应需要多长时间
- 收到响应后处理响应所需的时间
- 如果命令是响应集群 ASK 或 MOVED 响应而发送的
    - 如果是，原始命令是什么

`TimeSpan` 有较高的精度，如果运行时支持。 `DateTime` 准确度如 `DateTime.UtcNow` 。

选择上下文

由于StackExchange.Redis的异步接口，分析需要外部协助将相关的命令组合在一起。 这是实现的
通过提供上下文对象，当你开始和结束profiling（通过 `BeginProfiling(object)` & `FinishProfiling(object)` 方法），当一个命令被发送（通过 `IProfiler` 接口的 `GetContext()` 方法）。

一个将许多不同线程发出的命令关联在一起的玩具示例：

```
1.  class ToyProfiler : IProfiler
2.  {
3.      public ConcurrentDictionary<Thread, object> Contexts = new
    ConcurrentDictionary<Thread, object>();
4.
5.      public object GetContext()
6.      {
7.          object ctx;
8.          if(!Contexts.TryGetValue(Thread.CurrentThread, out ctx)) ctx = null;
9.
10.         return ctx;
11.     }
12. }
13.
14. // ...
15.
16. ConnectionMultiplexer conn = /* initialization */;
17. var profiler = new ToyProfiler();
18. var thisGroupContext = new object();
19.
20. conn.RegisterProfiler(profiler);
21.
22. var threads = new List<Thread>();
23.
24. for (var i = 0; i < 16; i++)
25. {
26.     var db = conn.GetDatabase(i);
27.
```

```
    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);
```

最后， `timings` 将包含16,000个 `IProfiledCommand` 对象 - 每个发送给redis的命令对应一个对象。

如果相反，你像下面这样做：

```
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` 最终会有1000个 `IProfilingCommand` 的16个，键由 `Thread` 发出。

不再看玩具示例，这里是如何在一个MVC5应用程序中配置 StackExchange.Redis。

首先针对你的 `ConnectionMultiplexer` 对象注册以下 `IProfiler`：

```
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;
    }
}
```

然后，将以下内容添加到 Global.asax.cs 文件：

```
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);

        // do what you will with `timings` here
    }
}
```

这个实现将所有redis命令（包括 `async / await` -ed命令）与触发它们的http请求分组。

查看原文

发布订阅顺序

发布/订阅 消息顺序

当使用 发布/订阅 API 时，需要决定使用同一连接的消息应该是顺序处理 还是并行处理 。

顺序处理意味着你（很大程度上）不需要担心线程安全问题，并且这意味着你保持了事件的顺序。它们会完全按照（通过队列）接受的顺序来处理，但是结果这也意味着消息会延迟彼此。

另一种选择是 *concurrent*（并行） 处理。这使得工作的处理顺序 没有特定的保证 并且你的代码完全负责确保并发的消息不应该破坏内部的状态—但这样可以显著的更快，更加可以扩展。
如果消息间一般都不相关，这种处理方式特别好。

出于安全考虑， 默认处理方式是顺序处理。但是，强烈建议你尽可能的使用并行处理。这是一个简单的修改：

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

这不是一个配置 选项，因为这样做是否合适 完全 取决于订阅消息的代码。

查看原文

脚本

脚本

`IServer.ScriptLoad(Async)` 、 `IServer.ScriptExists(Async)` 、 `IServer.ScriptFlush(Async)` 、 `IDatabase.ScriptEvaluate` 和 `IDatabaseAsync.ScriptEvaluateAsync` 这些方法为基本的 Lua脚本 提供了支持。
这些方法暴露了向Redis提交和执行Lua脚本所需的基本命令。

通过 `LuaScript` 类可以获得更复杂的脚本。 `LuaScript` 类使得更容易准备和提交参数以及脚本，以及允许您使用清理代码后变量名称。

`LuaScript` 的使用示例：

```
    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` 类将 `@myVar` 形式的脚本中的变量重写为redis所需的合适的 `ARGV[someIndex]` 。
如果传递的参数是 `RedisKey` 类型，它将作为 `KEYS` 集合的一部分自动发送。

Any object that exposes field or property members with the same name as @-prefixed variables in the Lua script can be used as a parameter hash to `Evaluate` calls.

任何在Lua脚本中暴露的以 `@` 为前缀变量同名的字段或属性成员的对象都可以用作参数哈希 `Evaluate` 调用。

支持的成员类型如下：

- int(?)
- long(?)

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

为了避免在每次评估时重新传输Lua脚本到redis，`LuaScript` 对象可以通过 `LuaScript.Load(IServer)` 转换为 `LoadedLuaScript` 。`LoadedLuaScripts` 使用 `EVALSHA` 求值，并由 hash 引用。

`LoadedLuaScript` 的使用示例：

```
    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` 上的所有方法都有Async替代方法，并将提交到redis的实际脚本公开为 `ExecutableScript` 属性。

查看原文

超时

你是否正遇到网络或 CPU 的瓶颈？

验证客户端和托管redis-server的服务器上支持的最大带宽。如果有请求被带宽限制，则它们需要更长时间才能完成，从而可能导致超时。
同样，验证您没有在客户端或服务器框上获得CPU限制，这将导致请求等待CPU时间，从而超时。

有没有命令需要在 redis 服务器上处理很长时间？

可能有一些命令需要很长时间才能在redis服务器上处理，导致请求超时。
长时间运行的命令的很少例子有 mget有大量的键，键*或写得不好的lua脚本。
可以运行通过 SlowLog 命令查看是否有请求花费比预期更长的时间。

在这里 可以找到关于命令的更多细节。

在向Redis发出的几个小请求之前是否有大的请求超时？

错误消息中的参数“qs”告诉您有多少从客户端发送到服务器，但尚未处理响应的请求。
对于某些类型的加载，您可能会看到此值不断增长，因为 StackExchange.Redis 使用单个TCP连接，并且一次只能读取一个响应。
即使第一个操作超时，它也不会停止 向服务器发送/从服务器发送 数据，其他请求也会被阻塞，直到该操作完成。 从而，导致超时。
一个解决方案是通过确保redis-server缓存对于您的工作负载足够大并将大值分割为更小的块来最小化超时的可能性。

另一个可能的解决方案是在客户端中使用 ConnectionMultiplexer 对象池，并在发送新请求时选择“最小化加载”ConnectionMultiplexer。 这样可能会防止单个超时导致其他请求也超时。

在超时异常中，是否有很多 busyio 或 busyworker 线程？

让我们先了解一下 ThreadPool 增长的一些细节：

CLR ThreadPool有两种类型的线程 - “工作线程”和“I/O 完成端口”（也称为 IOCP）线程。

- 工作线程用于处理 `Task.Run(...)` 或 `ThreadPool.QueueUserWorkItem(...)` 方法时。当工作需要在后台线程上发生时，这些线程也被CLR中的各种组件使用。
- 当异步IO发生时（例如从网络读取），使用IOCP线程。

线程池根据需要提供新的工作线程或I / O完成线程（无任何调节），直到达到每种类型线程的“最小”设置。 默认情况下，最小线程数设置为系统上的处理器数。

一旦现有（繁忙）线程的数量达到“最小”线程数，ThreadPool将调节每500毫秒向一个线程注入新线程的速率。 这意味着如果你的系统需要一个IOCP线程的工作，它会很快处理这个工作。 但是，如果工作突发超过配置的“最小”设置，那么在处理一些工作时会有一些延迟，因为ThreadPool会等待两个事情之一发生：

```
1. 1. 现有线程可以自由处理工作
2. 2. 连续 500ms 没有现有线程空闲，因此创建一个新线程。
```

基本上，这意味着当忙线程数大于最小线程时，在应用程序处理网络流量之前，可能需要付出500毫秒的延迟。
此外，重要的是要注意，当现有线程保持空闲超过15秒（基于我记得），它将被清理，这个增长和收缩的循环可以重复。

如果我们看一个来自 StackExchange.Redis（build 1.0.450或更高版本）的示例错误消息，您将看到它现在会打印 ThreadPool 统计信息（请参阅下面的IOCP和WORKER详细信息）。

```
1. System.TimeoutException: Timeout performing GET MyKey, inst: 2, mgr: Inactive,
2. queue: 6, qu: 0, qs: 6, qc: 0, wr: 0, wq: 0, in: 0, ar: 0,
3. IOCP: (Busy=6,Free=994,Min=4,Max=1000),
4. WORKER: (Busy=3,Free=997,Min=4,Max=1000)
```

在上面的示例中，您可以看到，对于 IOCP 线程，有6个忙线程，并且系统配置为允许4个最小线程。 在这种情况下，客户端可能会看到两个500毫秒的延迟，因为6> 4。

请注意，如果 IOCP 或 WORKER 线程的增长受到限制，StackExchange.Redis 可能会超时。

同样需要注意的是 如果你使用的 .NET Core 版本使用的 `netstandard` 版本小于 2.0，IOCP 和 WORKER 线程将不会显示。

建议：

鉴于上述信息，建议将 IOCP 和 WORKER 线程的最小配置值设置为大于默认值的值。 我们不能给出一个大小适合所有指导这个值应该是什么，因为一个应用程序的正确价值将太高/低为另一个应用程序。
此设置也会影响复杂应用程序的其他部分的性能，因此您需要根据您的特定需求调整此设置。
一个好的起点是200或300，然后根据需要进行测试和调整。

如何配置这个设置：

- 在 ASP.NET 中，使用 machine.config 中 `<processModel>` 配置元素下的“minIoThreads”配置设置.aspx)。 根据微软的做法，你不能修改每个站点 web.config 中的这个值（即使你过去这样做是可以的），如果你这样改的话你所有的.NET 站点都会使用这个设置的值。
  请注意如果你设置 `autoconfig` 为 `false` 是不需要添加每一个属性的，仅需要添加 `autoconfig="false"` 并且覆盖原来的值就可以了：
  ```
  <processModel autoConfig="false" maxIoThreads="250" />
  ```

> 重要说明： 此配置元素中指定的值是为每个核 设置。例如，如果你有一个4核的机器，并希望你的 *minIthreads* 设置为200在运行时，你应该使用 `<processModel minIoThreads =“50”/>` 。

- 在 ASP.NET 之外，使用 ThreadPool.SetMinThreads(…).aspx)API。
- 在 .NET Core 中 添加环境变量 `COMPlus_ThreadPool_ForceMinWorkerThreads` 来覆盖默认的 `MinThreads` 设置，参考 Environment/Registry Configuration Knobs

查看原文

事务

Redis中的事务

Redis中的事务不像SQL数据库中的事务。
完整的文档在这里，这里稍微借用一下：

redis中的事务包括放置在 `MULTI` 和 `EXEC` 之间的一组命令（或者用于回滚的 `DISCARD` ）。
一旦遇到 `MULTI` ，该连接相关的命令不会被执行 - 它们会进入一个队列（并且 每一个命令调用者得到一个答复 `QUEUED` ）。

当遇到一个 `EXEC` 时，它们都被应用在一个单元中（即在操作期间没有其他连接获得时间去做任何事）。
如果看到 `DISCARD` 而不是 `EXEC` ，则一切都被抛弃。因为事务中的命令会排队，你不能在事务里面 操作。

例如，在SQL数据库中，您可以执行以下操作（伪代码 - 仅供说明）：

```
// assign a new unique id only if they don't already
// have one, in a transaction to ensure no thread-races
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();
}
```

如何在Redis中实现事务呢？

这在redis事务中是不可能的：一旦事务被打开你不能获取数据 - 你的操作被排队。 幸运的是，还有另外两个命令帮助我们： `WATCH` 和 `UNWATCH` 。

`WATCH {key}` 告诉Redis，我们对用于事务目的的特定的键感兴趣。
Redis会自动跟踪这个键，任何变化基本上都会使我们的事务回滚 - `EXEC` 和 `DISCARD` 一样（调用者可以检测到这一点，并从头开始重试）。
所以你可以做的是： `WATCH` 一个键，以正常的方式检查该键的数据，然后 `MULTI` / `EXEC` 你的更改。

如果，当你检查数据，你发现你实际上不需要事务，你可以使用 `UNWATCH` 来取消关注所有关注的键。
注意，关注的键在 `EXEC` 和 `DISCARD` 期间也被复位。 所以在*Redis*层，事务是从概念上讲的。

```
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 - 有一个 `MULTI` / `EXEC` 只跨越一个操作 - 但重要的是，我们现在也从其他所有连接跟踪对 `{custKey}` 的更改 - 如果任何人更改键 ，事务将被中止。

在 StackExchange.Redis 中如何实现事务？

说实话，StackExchange.Redis 使用多路复用器方法实现事务更复杂。
我们不能简单地让并发的调用者发出 `WATCH` / `UNWATCH` / `MULTI` / `EXEC` / `DISCARD` ：它会全部混在一起。

因此，StackExchange.Redis 提供了额外的抽象来使事情更简单的变得正常：*constraints*。

Constraints 基本上是预先测试涉及 `WATCH` ，一些测试，以及结果的检查。
如果所有约束都通过，则触发 `MULTI` / `EXEC` ， 否则触发 `UNWATCH` 。

这是以防止与其他调用者混合在一起的命令的方式完成的。 所以我们的例子变成了：

```
1. var newId = CreateNewId();
2. var tran = db.CreateTransaction();
3. tran.AddCondition(Condition.HashNotExists(custKey, "UniqueID"));
```

```
tran.HashSetAsync(custKey, "UniqueID", newId);
bool committed = tran.Execute();
// ^^^ if true: it was applied; if false: it was rolled back
```

注意，从 `CreateTransaction` 返回的对象只能访问 *async* 方法 - 因为每个操作的结果在 `Execute`（或 `ExecuteAsync` ）完成之前都不会知道。
如果操作不应用，所有的任务将被标记为已取消 - 否则，命令执行后，您可以正常获取每个的结果。

可用条件 的集合不是广泛的，而是涵盖最常见的情况；如果你还想看到其他条件，请与我联系（或更好的方式：提交 pull-request）。

借助 `When` 的内置操作

还应该注意的是，Redis 预期已经了许多常见的情况（特别是：密钥/散列 存在，如上所述），所以存在单次操作原子命令。

这些是通过 `When` 参数访问的 - 所以我们前面的例子可以也可以写成：

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

（这里， `When.NotExists` 导致使用 `HSETNX` 命令，而不是 `HSET` ）

Lua 脚本

你还应该知道，Redis 2.6及以上版本支持Lua脚本，用于在服务器端执行多个作为单个原子单元的操作的通用工具。由于在Lua脚本中没有服务于其他连接，它的行为很像一个事务，但没有 `MULTI` / `EXEC` 等这样复杂。
这也避免了在调用者和服务器之间的带宽和延迟等问题
，但是需要与脚本垄断服务器的持续时间之间权衡。

在Redis层（假设 `HSETNX` 不存在），这可以实现为：

```
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}
```

这可以在 StackExchange.Redis 中使用：

```
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 `ScriptEvaluate` 和 `ScriptEvaluateAsync` 的响应是可变的，这取决于你确切的脚本，响应可以被强制（类型）转换- 在这种情况下为 `bool` ）

查看原文