ASP.NET Core

Session

Redis

www.bzMe.work

参考：

<https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1>

<https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1>

<https://www.runoob.com/redis/redis-install.html>

**一.常用的数据缓存方式**

HTTP是无状态的协议。 不采取其他步骤的情况下，HTTP 请求是不保留用户值或应用状态的独立消息。 如果你想要缓存用户的登录信息就会很困难。本文介绍了几种保留请求期间用户数据和应用状态的方法。可以使用几种方法存储状态。

|  |  |
| --- | --- |
| 存储方法 | 存储机制 |
| [Cookie](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#cookies) | HTTP Cookie（可能包括使用服务器端应用代码存储的数据） |
| [Session State](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#session-state) | HTTP Cookie 和服务器端应用代码 |
| [TempData](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#tempdata) | HTTP Cookie 或会话状态 |
| [Query Strings](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#query-strings) | HTTP 查询字符串 |
| [Hidden Fields](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#hidden-fields) | HTTP 窗体字段 |
| [HttpContext.Items](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#httpcontextitems) | 服务器端应用代码 |
| [Cache](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#cache) | 服务器端应用代码 |
| [Dependency Injection](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#dependency-injection) | 服务器端应用代码 |
|  |  |

**1.Cookie**

Cookie存储所有请求的数据。因为Cookie是随每个请求发送的，所以它们的大小应该保持在最低限度。 理想情况下，仅标识符应存储在Cookie中(例如用户名称,Email地址)，而数据则由后端的应用存储。大多数浏览器Cookie大小限制为4096个字节。每个域仅有有限数量的Cookie可用。

由于Cookie易被篡改，因此它们必须由服务器进行验证。 客户端(前端)上的Cookie能被用户删除或者过期。 但是，Cookie通常是客户端上最持久的数据暂留形式。

Cookie通常用于个性化设置，其中的内容是为已知用户定制的。大多数情况下，仅标识用户，但不对其进行身份验证。Cookie可以存储用户名、帐户名或唯一的用户ID（例如GUID）。 然后，可以使用Cookie 访问用户的个性化设置，例如首选的网站背景色。

由于Cookie把数据缓存在客户端，发布Cookie和处理隐私问题时，请留意[欧盟一般数据保护条例 (GDPR)](https://ec.europa.eu/info/law/law-topic/data-protection)。有关详细信息，请参阅 [ASP.NET Core 中的一般数据保护条例 (GDPR) 支持](https://docs.microsoft.com/zh-cn/aspnet/core/security/gdpr?view=aspnetcore-3.1)。

**2.Session**

会话状态(Session State)是在用户浏览Web应用时用来存储用户数据的ASP.NET Core方案。

会话状态使用服务端(后端)应用维护的存储来保存客户端所有请求的数据。会话数据由缓存支持并被视为临时数据，站点应在没有会话数据的情况下继续运行。关键应用程序数据应存储在用户数据库中(例如单据业务)，并仅作为性能优化缓存在会话中(例如缓存在本地Redis数据库中，不用每次都通过WebApi等方式从服务器上读取)。

[SignalR](https://docs.microsoft.com/aspnet/signalr/index)应用不支持会话，因为[SignalR中心](https://docs.microsoft.com/zh-cn/aspnet/core/signalr/hubs?view=aspnetcore-3.1)可能独立于HTTP上下文(HttpContext，其中包括当前的Http会话的Request、Response、Session、Cokie等会话资源，这称之为HTTP上下文)执行。例如，当中心打开的长轮询请求超出请求的HTTP上下文的生存期时，可能发生这种情况。

ASP.NET Core通过向客户端提供包含会话ID的Cookie来维护会话状态(即SessionID和Cookie数据会被一起发送给客户端，这就是我们为什么能在服务端的HTTP上下文修改Cookie数据后客户端能反映修改后数据的原因)，该会话ID与每个请求一起发送到服务端应用(即当浏览器发起一个请求的时候，会将SessionID回传回服务端，这就是为什么会话是独立于浏览器的原因)。服务端应用使用会话ID来获取会话数据。会话状态具有以下行为：

（1）由于会话Cookie是特定于浏览器的，因此不能跨浏览器共享会话Cookie。

（2）浏览器会话结束时会话Cookie被删除。

（3）如果收到过期的会话Cookie，则创建使用相同会话Cookie的新会话。

（4）不会保留空会话：会话中必须设置了至少一个值以保存所有请求的会话，否则会话未保留时，为每个新的请求会生成新会话ID(这意味着你想要得到一个持久的会话，必须为当前会话设置至少一个值，否则当你刷新页面或者跳转页面的时候都会生成新的会话ID，因为系统认为没有值的会话不需要保留)。

（5）应用在上次请求后保留会话的时间有限。应用设置会话超时，或者使用20分钟的默认值。会话状态适用于存储特定于特定会话的用户数据，但该数据无需永久的会话存储。

（6）调用[ISession.Clear](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession.clear)或者会话过期时，会删除会话数据。

（7）没有默认机制告知客户端浏览器已关闭或者客户端上的会话Cookie被删除或过期的应用代码。

（8）ASP.NET Core MVC 和 Razor Pages 模板包括对一般数据保护条例(GDPR)的支持。默认情况下，会话状态Cookie不标记为“基本”，因此，除非站点访问者允许跟踪，否则会话状态不起作用。有关详细信息，请参阅 [在 ASP.NET Core中的常规数据保护法规 (GDPR) 支持](https://docs.microsoft.com/zh-cn/aspnet/core/security/gdpr?view=aspnetcore-3.1#tempdata-provider-and-session-state-cookies-arent-essential)。(这是为了保护用户隐私，首次访问页面时会提醒你是否愿意接受开启Cookie，缓存用户认为可能是敏感的数据)

警告：请勿将敏感数据存储在会话状态中。用户可能不会关闭浏览器并清除会话Cookie。某些浏览器会保留所有浏览器窗口中的有效会话Cookie。会话可能不限于单个用户-下一个用户可能继续使用同一会话 Cookie浏览应用。

内存中缓存提供程序在应用驻留的服务器内存中存储会话数据。在服务器场方案中：

（1）使用粘性会话将每个会话加入到单独服务器上的特定应用实例。默认情况下，[Azure应用服务](https://azure.microsoft.com/services/app-service/)使用[应用程序请求路由(ARR)](https://docs.microsoft.com/zh-cn/iis/extensions/planning-for-arr/using-the-application-request-routing-module)强制实施粘性会话。然而，粘性会话可能会影响可伸缩性，并使 Web应用更新变得复杂。更好的方法是使用Redis或SQL Server分布式缓存，它们不需要粘性会话。有关详细信息，请参阅[ASP.NET Core 中的分布式缓存](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1)。

（2）通过[IDataProtector](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.dataprotection.idataprotector)加密会话Cookie。必须正确配置数据保护，以在每台计算机上读取会话 Cookie。有关详细信息，请参阅[ASP.NET Core 数据保护](https://docs.microsoft.com/zh-cn/aspnet/core/security/data-protection/introduction?view=aspnetcore-3.1)和[密钥存储提供程序](https://docs.microsoft.com/zh-cn/aspnet/core/security/data-protection/implementation/key-storage-providers?view=aspnetcore-3.1)。

**(1)** **配置会话状态**

[Microsoft.AspNetCore.App metapackage](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/metapackage-app?view=aspnetcore-3.1)中包含的[Microsoft.AspNetCore.Session](https://www.nuget.org/packages/Microsoft.AspNetCore.Session/)包提供中间件来管理会话状态。若要启用会话中间件，Startup必须包含：

（1）任一[IDistributedCache](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.caching.distributed.idistributedcache)内存缓存。IDistributedCache实现用作会话后备存储(即将当前会话备份在后端存储中，例如Redis服务器中)。有关详细信息，请参阅[ASP.NET Core 中的分布式缓存](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1)。

（2）对ConfigureServices中[AddSession](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.sessionservicecollectionextensions.addsession)的调用。

（3）对Configure中[UseSession](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.builder.sessionmiddlewareextensions.usesession#Microsoft_AspNetCore_Builder_SessionMiddlewareExtensions_UseSession_Microsoft_AspNetCore_Builder_IApplicationBuilder_)的调用。

以下代码演示如何使用IDistributedCache的默认内存中实现设置内存中会话提供程序：

|  |
| --- |
| public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  services.AddDistributedMemoryCache();  services.AddSession(options =>  {  // Set a short timeout for easy testing.  options.IdleTimeout = TimeSpan.FromSeconds(10);  options.Cookie.HttpOnly = true;  // Make the session cookie essential 启用会话Cookie  options.Cookie.IsEssential = true;  });  services.AddMvc()  .SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);  }  public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  else  {  app.UseExceptionHandler("/Error");  app.UseHsts();  }  app.UseHttpsRedirection();  app.UseStaticFiles();  app.UseSession();  app.UseHttpContextItemsMiddleware();  app.UseMvc();  }  } |

中间件的顺序很重要。在前面的示例中，在UseMvc之后调用UseSession时会发生 InvalidOperationException 异常。有关详细信息，请参阅[中间件排序](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/middleware/index?view=aspnetcore-3.1#order)。

配置会话状态后，[HttpContext.Session](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.httpcontext.session)可用。

调用UseSession以前无法访问HttpContext.Session。

在应用已经开始写入到响应流之后，不能创建有新会话Cookie的新会话。此异常记录在Web服务器日志中但不显示在浏览器中。

**(2)** **以异步方式加载会话状态**

只有在 [TryGetValue](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession.trygetvalue)、[Set](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession.set)或 [Remove](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession.remove)方法之前显式调用[ISession.LoadAsync](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession.loadasync)方法，ASP.NET Core 中的默认会话提供程序才会从基础[IDistributedCache](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.caching.distributed.idistributedcache)后备存储以异步方式加载会话记录。如果未先调用LoadAsync，则会同步加载基础会话记录，这可能对性能产生大规模影响。

若要让应用强制实施此模式，如果未在TryGetValue、Set或Remove之前调用LoadAsync方法，那么使用引起异常的版本包装[DistributedSessionStore](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.session.distributedsessionstore)和 [DistributedSession](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.session.distributedsession)实现。在服务容器中注册的已包装的版本。

**(3)** **会话选项**

若要替代会话默认值，请使用SessionOptions。

|  |  |
| --- | --- |
| [Cookie](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.builder.sessionoptions.cookie)  (会话Cookie) | 确定用于创建Cookie的设置。  [名称](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.name)默认为[SessionDefaults.CookieName](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.session.sessiondefaults.cookiename)(.AspNetCore.Session)。  [路径](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.path)默认 [SessionDefaults.CookiePath](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.session.sessiondefaults.cookiepath)(/)。  [SameSite](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.samesite)默认为[SameSiteMode.Lax](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.samesitemode) (1)。  [HttpOnly](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.httponly)默认为true。  [IsEssential](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.isessential)默认为false。 |
| [IdleTimeout](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.builder.sessionoptions.idletimeout)  (会话超时时间) | IdleTimeout 显示放弃其内容前，内容可以空闲多长时间。每个会话访问都会重置超时。此设置仅适用于会话内容，不适用于Cookie。默认为20分钟。 |
| [IOTimeout](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.builder.sessionoptions.iotimeout)  (存储超时) | 允许从存储加载会话或者将其提交回存储的最大时长。此设置可能仅适用于异步操作。 可以使用[InfiniteTimeSpan](https://docs.microsoft.com/zh-cn/dotnet/api/system.threading.timeout.infinitetimespan)禁用超时。默认值为1分钟。 |
|  |  |

会话使用Cookie跟踪和标识来自单个浏览器的请求。

默认情况下，此 Cookie名为.AspNetCore.Session，并使用路径/。由于Cookie默认值不指定域，因此它不提供页上的客户端脚本（因为 [HttpOnly](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.cookiebuilder.httponly) 默认为 true）。若要替换Cookie会话默认值，请使用 SessionOptions：

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.Configure<CookiePolicyOptions>(options =>  {  options.CheckConsentNeeded = context => true;  options.MinimumSameSitePolicy = SameSiteMode.None;  });  services.AddDistributedMemoryCache();  services.AddMvc()  .SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);  services.AddSession(options =>  {  options.Cookie.Name = ".AdventureWorks.Session";  options.IdleTimeout = TimeSpan.FromSeconds(10);  options.Cookie.IsEssential = true;  });  } |

应用使用 IdleTimeout 属性确定放弃服务器缓存中的内容前，内容可以空闲多长时间。此属性独立于 Cookie 到期时间。通过会话中间件传递的每个请求都会重置超时。

会话状态为“非锁定”。如果两个请求同时尝试修改同一会话的内容，则后一个请求替代前一个请求。 Session 是作为一个连贯会话实现的，这意味着所有内容都存储在一起。 两个请求试图修改不同的会话值时，后一个请求可能替代前一个做出的会话更改。

**(4)** **设置和获取会话值**

使用 [HttpContext.Session](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.httpcontext.session)从Razor Pages [PageModel](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.mvc.razorpages.pagemodel)类或MVC [控制器](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.mvc.controller)类访问会话状态。此属性是 [ISession](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession) 实现。

ISession 实现提供用于设置和检索整数和字符串值的若干扩展方法。 项目引用 [Microsoft.AspNetCore.Http.Extensions](https://www.nuget.org/packages/Microsoft.AspNetCore.Http.Extensions/)包时，扩展方法位于 [Microsoft.AspNetCore.Http](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http) 命名空间中（添加 using Microsoft.AspNetCore.Http; 语句获取对扩展方法的访问权限）。这两个包均包括在 [Microsoft.AspNetCore.App元包](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/metapackage-app?view=aspnetcore-3.1)中。ISession 扩展方法：

Get(ISession, String)

GetInt32(ISession, String)

GetString(ISession, String)

SetInt32(ISession, String, Int32)

SetString(ISession, String, String)

以下示例在 Razor Pages页中检索IndexModel.SessionKeyName键（示例应用中的\_Name）的会话值：

|  |
| --- |
| @page  @using Microsoft.AspNetCore.Http  @model IndexModel  ...  Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName) |

以下示例显示如何设置和获取整数和字符串：

|  |
| --- |
| public class IndexModel : PageModel  {  public const string SessionKeyName = "\_Name";  public const string SessionKeyAge = "\_Age";  const string SessionKeyTime = "\_Time";  public string SessionInfo\_Name { get; private set; }  public string SessionInfo\_Age { get; private set; }  public string SessionInfo\_CurrentTime { get; private set; }  public string SessionInfo\_SessionTime { get; private set; }  public string SessionInfo\_MiddlewareValue { get; private set; }  public void OnGet()  {  // Requires: using Microsoft.AspNetCore.Http;  if (string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))  {  HttpContext.Session.SetString(SessionKeyName, "The Doctor");  HttpContext.Session.SetInt32(SessionKeyAge, 773);  }  var name = HttpContext.Session.GetString(SessionKeyName);  var age = HttpContext.Session.GetInt32(SessionKeyAge); |

必须对所有会话数据进行序列化以启用分布式缓存方案，即使是在使用内存中缓存的时候。

提供最小的字符串和数字序列化程序（请参阅[ISession](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.isession)的方法和扩展方法）。用户必须使用另一种机制（例如 JSON）序列化复杂类型。添加以下扩展方法以设置和获取可序列化的对象：

|  |
| --- |
| public static class SessionExtensions  {  public static void Set<T>(this ISession session, string key, T value)  {  session.SetString(key, JsonConvert.SerializeObject(value));  }  public static T Get<T>(this ISession session, string key)  {  var value = session.GetString(key);  return value == null ? default(T) :  JsonConvert.DeserializeObject<T>(value);  }  } |

以下示例演示如何使用扩展方法设置和获取可序列化的对象：

|  |
| --- |
| // Requires you add the Set and Get extension method mentioned in the topic.  if (HttpContext.Session.Get<DateTime>(SessionKeyTime) == default(DateTime))  {  HttpContext.Session.Set<DateTime>(SessionKeyTime, currentTime);  } |

**3.TempData**

ASP.NET Core 公开 Razor Pages [TempData](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.razorpages.pagemodel.tempdata#Microsoft_AspNetCore_Mvc_RazorPages_PageModel_TempData) 或控制器 [TempData](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.controller.tempdata#Microsoft_AspNetCore_Mvc_Controller_TempData)。在另一个请求读取数据之前，此属性将读取此数据。 [Keep(String)](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.itempdatadictionary.keep) 和 [Peek(string)](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.itempdatadictionary.peek) 方法可用于检查数据，而无需在请求结束时删除。 [Keep()](https://docs.microsoft.com/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.itempdatadictionary.keep)将标记字典中的所有项以进行保留。当多个请求需要数据时，TempData 非常有助于进行重定向。TempData提供程序使用Cookie或会话状态(Session)实现TempData。

**(1)** **TempData示例**

考虑创建客户的以下页面：

|  |
| --- |
| public class CreateModel : PageModel  {  private readonly RazorPagesContactsContext \_context;  public CreateModel(RazorPagesContactsContext context)  {  \_context = context;  }  public IActionResult OnGet()  {  return Page();  }  [TempData]  public string Message { get; set; }  [BindProperty]  public Customer Customer { get; set; }  public async Task<IActionResult> OnPostAsync()  {  if (!ModelState.IsValid)  {  return Page();  }  \_context.Customer.Add(Customer);  await \_context.SaveChangesAsync();  Message = $"Customer {Customer.Name} added";  return RedirectToPage("./IndexPeek");  }  } |

以下页面显示 TempData["Message"]：

|  |
| --- |
| @page  @model IndexModel  <h1>Peek Contacts</h1>  @{  if (TempData.Peek("Message") != null)  {  <h3>Message: @TempData.Peek("Message")</h3>  }  }  @\*Content removed for brevity.\*@ |

在前面的标记中，在请求结束时，不会删除 TempData["Message"]，因为正在使用Peek。刷新页面将显示 TempData["Message"](因为值存储在Session中，所以刷新页面不会丢失)。

以下标记类似于前面的代码，但使用Keep在请求结束时保留数据：

|  |
| --- |
| @page  @model IndexModel  <h1>Contacts Keep</h1>  @{  if (TempData["Message"] != null)  {  <h3>Message: @TempData["Message"]</h3>  }  TempData.Keep("Message");  }  @\*Content removed for brevity.\*@ |

在 IndexPeek 和 IndexKeep 页面之间导航不会删除TempData["Message"]。

以下代码显示TempData["Message"]，但请求结束时，将删除TempData["Message"]：

|  |
| --- |
| @page  @model IndexModel  <h1>Index no Keep or Peek</h1>  @{  if (TempData["Message"] != null)  {  <h3>Message: @TempData["Message"]</h3>  }  }  @\*Content removed for brevity.\*@ |

**(2)** **TempData提供程序**

基于cookie的TempData提供程序默认用于存储cookie中的TempData。

使用由 [Base64UrlTextEncoder](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.webutilities.base64urltextencoder) 编码的 [IDataProtector](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.dataprotection.idataprotector) 对 Cookie 数据进行加密，然后进行分块。 因为Cookie进行了分块，所以ASP.NET Core 1.x 中的单个Cookie大小限制不适用。未压缩 Cookie数据，因为压缩加密的数据会导致安全问题，如[CRIME](https://wikipedia.org/wiki/CRIME_(security_exploit))和[BREACH](https://wikipedia.org/wiki/BREACH_(security_exploit))攻击。有关基于Cookie的TempData提供程序的详细信息，请参阅[CookieTempDataProvider](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.mvc.viewfeatures.cookietempdataprovider)。

**(3)** **选择TempData提供程序**

选择 TempData提供程序涉及几个注意事项，例如：

（1）应用是否已使用会话状态？如果是，使用会话状态TempData提供程序对应用没有额外的成本（除了数据的大小）。

（2）应用是否只对相对较小的数据量（最多500个字节）使用TempData？如果是，Cookie TempData 提供程序将为每个携带TempData的请求增加较小的成本。如果不是，会话状态TempData提供程序有助于在使用TempData前，避免在每个请求中来回切换大量数据。

（3）应用是否在多个服务器上的服务器场中运行？如果是，无需其他任何配置，即可在数据保护外使用 Cookie TempData提供程序（请参阅[ASP.NET Core 数据保护](https://docs.microsoft.com/zh-cn/aspnet/core/security/data-protection/introduction?view=aspnetcore-3.1)和[密钥存储提供程序](https://docs.microsoft.com/zh-cn/aspnet/core/security/data-protection/implementation/key-storage-providers?view=aspnetcore-3.1)）。

备注

大多数Web 客户端（如 Web 浏览器）针对每个Cookie的最大大小和/或Cookie总数强制实施限制。 使用Cookie TempData提供程序时，请验证应用未超过这些限制。考虑数据的总大小。解释加密和分块导致的Cookie大小增加。

**(4)** **配置TempData提供程序**

默认情况下启用基于Cookie的TempData提供程序。

若要启用基于会话的TempData提供程序，请使用[AddSessionStateTempDataProvider](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.dependencyinjection.mvcviewfeaturesmvcbuilderextensions.addsessionstatetempdataprovider)扩展方法：

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.Configure<CookiePolicyOptions>(options =>  {  options.CheckConsentNeeded = context => true;  options.MinimumSameSitePolicy = SameSiteMode.None;  });  services.AddMvc()  .SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2)  .AddSessionStateTempDataProvider();  services.AddSession();  }  public void Configure(IApplicationBuilder app, IHostingEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  app.UseDatabaseErrorPage();  }  else  {  app.UseExceptionHandler("/Error");  app.UseHsts();  }  app.UseHttpsRedirection();  app.UseStaticFiles();  app.UseCookiePolicy();  app.UseSession();  app.UseMvc();  } |

中间件的顺序很重要。在前面的示例中，在UseMvc之后调用UseSession时会发生 InvalidOperationException 异常。有关详细信息，请参阅[中间件排序](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/middleware/index?view=aspnetcore-3.1#order)。

**4.查询字符串**

可以将有限的数据从一个请求传递到另一个请求，方法是将其添加到新请求的查询字符串中。 这有利于以一种持久的方式捕获状态，这种方式允许通过电子邮件或社交网络共享嵌入式状态的链接。 由于URL查询字符串是公共的，因此请勿对敏感数据使用查询字符串。

除了意外的共享，在查询字符串中包含数据还会为[跨站点请求伪造(CSRF)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF))攻击创造机会，从而欺骗用户在通过身份验证时访问恶意网站。 然后，攻击者可以从应用中窃取用户数据，或者代表用户采取恶意操作。 任何保留的应用或会话状态必须防止CSRF攻击。有关详细信息，请参阅[预防跨网站请求伪造 (XSRF/CSRF) 攻击](https://docs.microsoft.com/zh-cn/aspnet/core/security/anti-request-forgery?view=aspnetcore-3.1)。

**5.隱藏字段**

数据可以保存在隐藏的表单域中，并在下一个请求上回发。这在多页窗体中很常见。由于客户端可能篡改数据，因此应用必须始终重新验证存储在隐藏字段中的数据。

**6.HttpContext.Items**

处理单个请求时，使用[HttpContext.Items](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.http.httpcontext.items)集合存储数据。处理请求结束后，放弃集合的内容。通常使用Items集合允许组件或中间件在请求期间在不同时间点操作且没有直接传递参数的方法时进行通信。在下面示例中，[中间件](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/middleware/index?view=aspnetcore-3.1)将isVerified添加到Items集合。

|  |
| --- |
| app.Use(async (context, next) =>  {  // perform some verification  context.Items["isVerified"] = true;  await next.Invoke();  }); |

然后，在管道中，另一个中间件可以访问isVerified的值：

|  |
| --- |
| app.Run(async (context) =>  {  await context.Response.WriteAsync($"Verified:{context.Items["isVerified"]}");  }); |

对于只供单个应用使用的中间件，string键是可以接受的。应用实例间共享的中间件应使用唯一的对象键以避免键冲突。以下示例演示如何使用中间件类中定义的唯一对象键：

|  |
| --- |
| public class HttpContextItemsMiddleware  {  private readonly RequestDelegate \_next;  public static readonly object HttpContextItemsMiddlewareKey = new Object();  public HttpContextItemsMiddleware(RequestDelegate next)  {  \_next = next;  }  public async Task Invoke(HttpContext httpContext)  {  httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";  await \_next(httpContext);  }  }  public static class HttpContextItemsMiddlewareExtensions  {  public static IApplicationBuilder  UseHttpContextItemsMiddleware(this IApplicationBuilder builder)  {  return builder.UseMiddleware<HttpContextItemsMiddleware>();  }  } |

其他代码可以使用通过中间件类公开的键访问存储在HttpContext.Items中的值：

|  |
| --- |
| HttpContext.Items  .TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey,  out var middlewareSetValue);  SessionInfo\_MiddlewareValue =  middlewareSetValue?.ToString() ?? "Middleware value not set!"; |

此方法还有避免在代码中使用关键字符串的优势。

综上，HttpContext.Items适用于Http请求上下文中各中间件之间互通数据。

**7.缓存**

缓存是存储和检索数据的有效方法。应用可以控制缓存项的生存期。

缓存数据未与特定请求、用户或会话相关联。 请注意不要缓存可能由其他用户请求检索的特定于用户的数据。有关详细信息，请参阅[响应缓存在ASP.NET Core](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/response?view=aspnetcore-3.1)。

**8.依赖关系注入**

使用[依赖关系注入](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1)可向所有用户提供数据：

（1）定义一项包含数据的服务中间件。例如，定义一个名为MyAppData的类：

|  |
| --- |
| public class MyAppData  {  // Declare properties and methods  } |

（2）将服务类添加到Startup.ConfigureServices：

|  |
| --- |
| public void ConfigureServices(IServiceCollection services)  {  services.AddSingleton<MyAppData>(); // 将数据服务加入到DI容器中  } |

（3）使用数据服务类：

|  |
| --- |
| public class IndexModel : PageModel  {  // 类实例化时，参数中一旦引用了数据服务，系统就会自动注入数据服务对象  public IndexModel(MyAppData myService)  {  // Do something with the service  // Examples: Read data, store in a field or property  }  } |

常见错误

（1）“在尝试激活‘Microsoft.AspNetCore.Session.DistributedSessionStore’时无法为类型‘Microsoft.Extensions.Caching.Distributed.IDistributedCache’解析服务。”

这通常是由于不能配置至少一个IDistributedCache实现而造成的。 有关详细信息，请参阅 [ASP.NET Core 中的分布式缓存](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1) 和 [缓存在内存中 ASP.NET Core](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/memory?view=aspnetcore-3.1)。

（2）在会话中间件保存会话失败的事件中（例如，如果后备存储不可用），中间件记录异常而请求继续正常进行。这会导致不可预知的行为。

例如，用户将购物车存储在会话中。用户将商品添加到购物车，但提交失败。应用不知道有此失败，因此它向用户报告商品已添加到购物车，但事实并非如此。

检查此类错误的建议方法是完成将应用写入到该会话后，

从应用代码调用 await HttpContext.Session.CommitAsync();。

如果后备存储不可用，则CommitAsync引发异常。

如果 CommitAsync 失败，应用可以处理异常。

在与数据存储不可用的相同的条件下，HttpContext.Session.LoadAsync引发异常。

**9.SignalR和会话状态**

SignalR应用不应使用会话状态来存储信息。SignalR应用可以将每个连接状态存储在中心的 Context.Items 中。

**二.分布式缓存**

本文主要讲解用Redis数据库作为分布式缓存，因此先讲解Redis数据库。

**1.Redis简介**

<https://blog.csdn.net/xgangzai/category_9289426.html>

<https://www.runoob.com/redis/redis-tutorial.html>

<https://redis.io/> (官方)

<http://www.redis.cn/> (中文)

官方对Redis的介绍如下：

“Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster.”

“Redis 是一个开源（BSD许可）的，内存中的数据结构存储系统，它可以用作数据库、缓存和消息中间件。它支持多种类型的数据结构，如[字符串（strings）](http://www.redis.cn/topics/data-types-intro.html#strings)，[散列（hashes）](http://www.redis.cn/topics/data-types-intro.html#hashes)，[列表（lists）](http://www.redis.cn/topics/data-types-intro.html#lists)， [集合（sets）](http://www.redis.cn/topics/data-types-intro.html#sets)， [有序集合（sorted sets）](http://www.redis.cn/topics/data-types-intro.html#sorted-sets)与范围查询，[bitmaps](http://www.redis.cn/topics/data-types-intro.html#bitmaps)，[hyperloglogs](http://www.redis.cn/topics/data-types-intro.html#hyperloglogs)和[地理空间（geospatial）](http://www.redis.cn/commands/geoadd.html)索引半径查询。 Redis内置了[复制（replication）](http://www.redis.cn/topics/replication.html)，[LUA脚本（Lua scripting）](http://www.redis.cn/commands/eval.html)，[LRU驱动事件（LRU eviction）](http://www.redis.cn/topics/lru-cache.html)，[事务（transactions）](http://www.redis.cn/topics/transactions.html)和不同级别的[磁盘持久化（persistence）](http://www.redis.cn/topics/persistence.html)，并通过 [Redis哨兵（Sentinel）](http://www.redis.cn/topics/sentinel.html)和自动[分区（Cluster）](http://www.redis.cn/topics/cluster-tutorial.html)提供高可用性（high availability）。”

简而言之，Redis是一个开源（BSD许可）的，使用C语言编写，既可以基于内存也可以持久化的一个高性能的Key-Value数据库。相较于其它Key-Value数据库，Redis具备以下优势：

(1) 性能极高

因为是基于字典(Dictionary)的Key-Value数据库，数据存在内存中,存取时间复杂度是O(1)，这带来极高的速度，例如微软.NET的数据字典，在内存中读写100万数据大概300毫秒，对于Key-Value数据库来说，性能瓶颈主要发生在对数据进行持久化和原子一致性操作上，Redis读的速度大约是110000次/s,写的速度是81000次/s。

(2) 丰富的数据类型

Redis支持Strings(字符串), Lists(列表), Hashes(哈希), Sets(集合)及

Ordered Sets(有序集合)数据类型操作。

(3) 支持事务

Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。

(4) 丰富的特性

Redis可用于缓存，消息，通知，Key过期自动删除等。

**2.Redis应用场景**

**1.缓存**

后端：

对于分布式微服务架构，常根据模块将服务划分在后端不同的容器中，以WebApi的形式向应用提供服务。

当前端应用需要大量的读写数据的时候，为了避免每次接口请求都要去查询数据库，可以把一些数据缓存到Redis中，这样是直接从内存中获取数据，大幅提高响应速度。

前端：

1. Session缓存

对于前端应用，用户登陆的一些信息存到Session中，并缓存到Redis中。

Redis根据Session ID设置过期时间以自动删除，保持与Session的生命周期同步。

(2) 购物车缓存

对于前端购物，可将购物车中的商品ID、商品名称、购买数量生成哈希，以用户ID作为Key，

缓存到Redis，这样购物车信息就不会随用户登录退出删除。

1. Top N List缓存

如果想要根据Cookie中记录的用户信息，或者前端购物应用的用户信息，跟踪用户最近常访问的商品以判断其个人喜好；或者统计站点最热的商品、点击率最高的商品、活跃度最高的用户等等。如果将这些数据频繁地写入关系数据库，将给数据库带来极大的压力。可以构造一个列表，以用户ID作为Key，缓存到Redis。

**2.队列**

Redis中提供了list接口，这个list提供了lpush和rpop，这两个方法具有原子性，可以插入队列元素和弹出队列元素。参考：<http://www.redis.net.cn/order/3584.html>。

在排队系统中，例如用户购票、大厅排队，如果想要限定用户的次序，采用先进先出的原则，那么采用Redis队列属性和lpush lpop两个函数即可解决。

**3.单点登录**

同一用户在多端（例如手机、平板、电脑等设备）多浏览器（例如IE、Firefox、Google Chrome等）中发起会话的时候，为了只允许用户单点登录系统，此时可以采用Redis限制用户。例如：

多用户单账号系统(一个用户只有一个账号，用户即账号)：

|  |  |  |
| --- | --- | --- |
| 用户ID（账号） | 活动的会话ID | 会话超时时间(小时) |
| 001 | 530ec70e-4918-ca61-83f8-1e2dcecb89be | 2 |
|  |  |  |

这种系统中，处理单点登录的规则有两种：

(1)如果用户已经登录，则在其它设备上登录时不允许登录。

(2)如果用户已经登录，则在其它设备上登录时，提醒是否允许登录当前用户，让用户在其它设备下线，例如QQ即采用这种登录模式。此时，在当前设备登录的时候，将活动的会话ID替换即可，使得其它正在线上的设备因为在Redis中找不到了其活动的会话ID，从而清理当前会话数据后退出。

多用户多账号系统(一个用户有多个账号)：

|  |  |  |  |
| --- | --- | --- | --- |
| 用户ID | 账号 | 活动的会话ID | 会话超时时间(小时) |
| 001 | A | 530ec70e-4918-ca61-83f8-1e2dcecb89be | 2 |
|  | B | 93e213ae-3245-2415-af2e-7c75d6015cca | 2 |
|  | C | 66a7340d-8294-a940-fe8d-7f976d099ede | 2 |
|  |  |  |  |

这种系统中，处理单点登录的规则同样有两种：

(1)如果用户的某个账号已经登录，则在其它设备上登录时不允许再用此账号登录。

(2)如果用户的某个账号已经登录，则在其它设备上登录时，提醒是否允许登录当前账号，让用户的其它活动账号在其它设备下线。此时，在当前设备登录的时候，将其它账号的活动会话ID清除即可，使得其它正在线上的设备因为在Redis中找不到了其活动的会话ID，从而清理当前会话数据后退出。

注意：

(1)客户端每次登录的时候除了在Redis中记录当前活动的会话ID以外，还要同步会话超时时间，以保持与客户端的会话超时时间一致，这样当超时以后使得Redis能自动清理用户登录信息，防止垃圾数据出现。

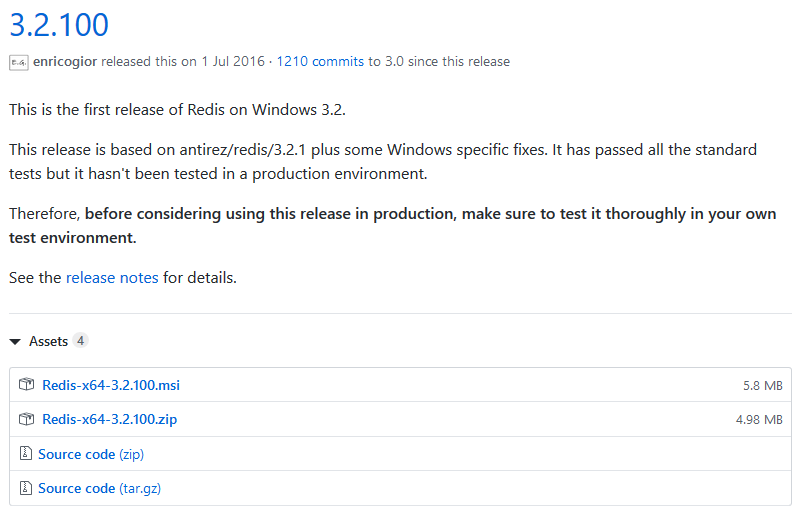
(2)在分布式微服务架构中，客户端应用可能部署在多个容器中，因此用户的会话数据应缓存到中心数据库中，这样才能起到单点登录控制目的。

**3.安装Redis**

**(1)下载Redis**

Windows版本：<https://github.com/microsoftarchive/redis/releases>。

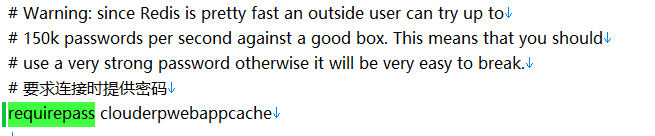
Linux版本：<http://redis.io/download>



本文下载的是Windows版本的Redis

(2)启动Redis

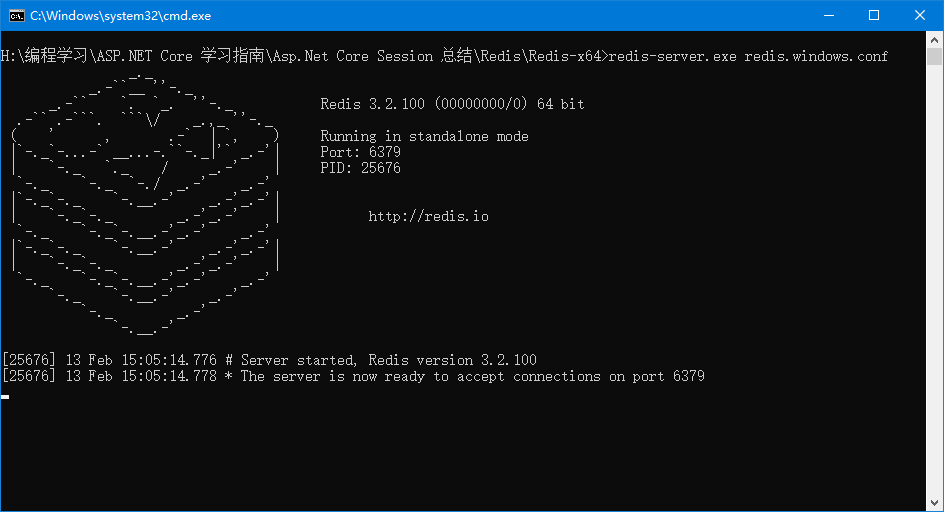
修改启动配置文件redis.windows.conf查找“requirepass”去掉注释符“#”，启用连接时要求密码：



编写一个批处理文件，输入：

|  |
| --- |
| redis-server.exe redis.windows.conf  pause |

另存为“RedisStartup.bat”，然后双击启动Redis服务器，启动后如下，默认监听6379端口以接受客户端连接：



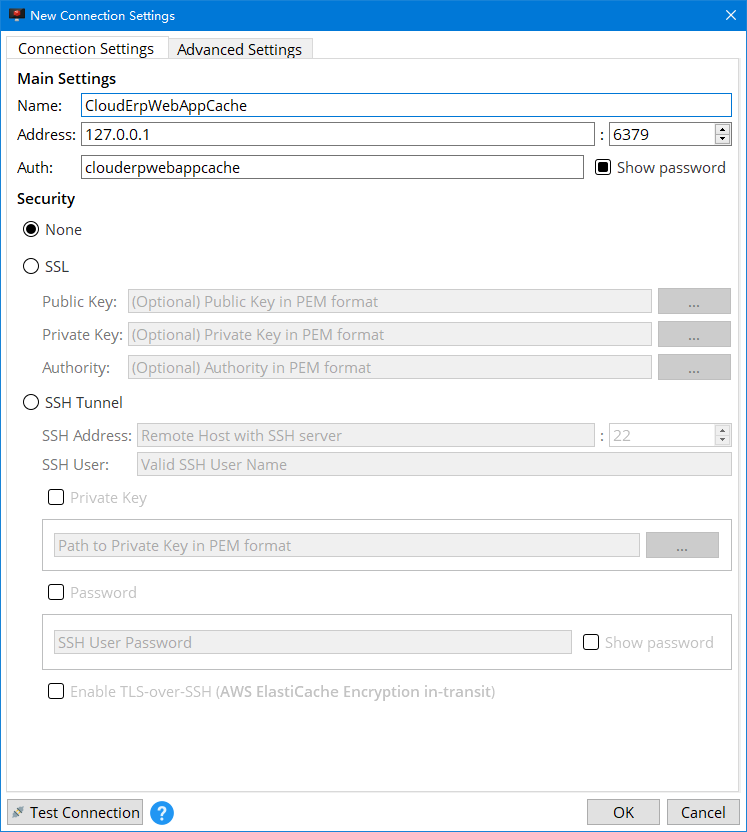
(3)下载Redis管理工具Redis Desktop Manager

工具名称：Redis Desktop Manager

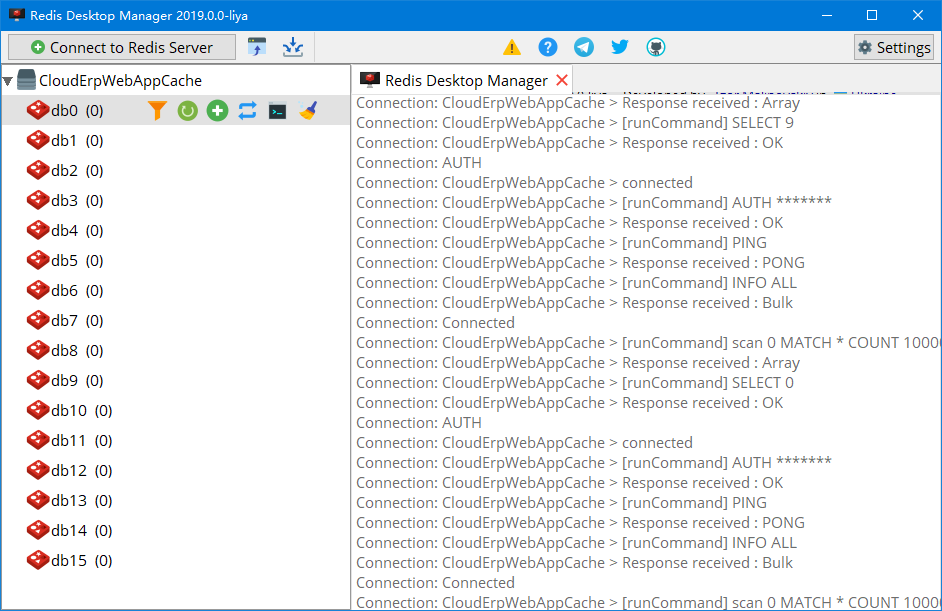
下载地址：<https://redisdesktop.com/>

源码地址：<https://github.com/uglide/RedisDesktopManager>

下载完成后，启动“Redis Desktop Manager”新建一个连接，连接名称任意设置，如下：



注意前面在配置文件中设置的连接密码：clouderpwebappcache 单击“OK”后：



Redis为我们提供了15个数据库，我们可以在任意数据库中读写数据。

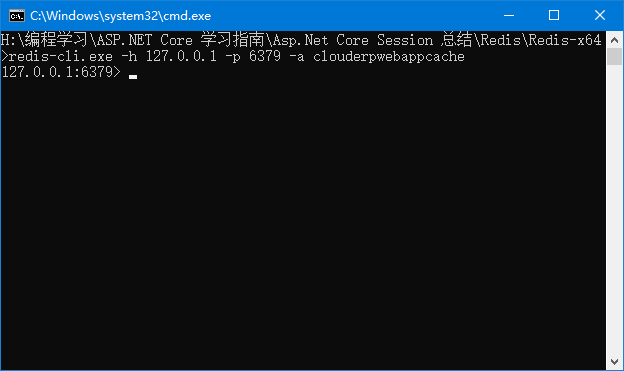
Redis数据库安装完成。

**4.配置Redis**

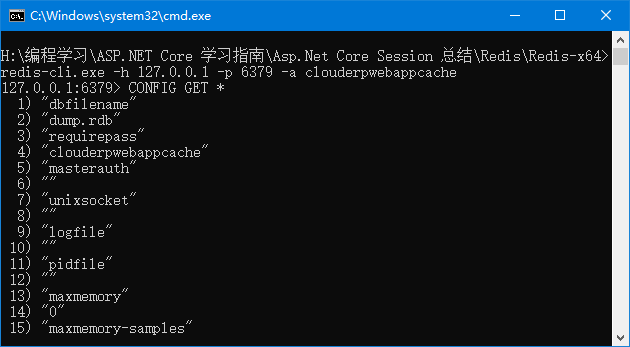
在Redis所在目录中新建一批处理文件，输入：

|  |
| --- |
| redis-cli.exe -h 127.0.0.1 -p 6379 -a clouderpwebappcache  pause |

另存为“RedisConnection.bat”，然后双击连接到Redis服务器，连接后如下：



现在已经连接到Redis服务器了，可以对数据库进行操作了，例如输入命令config get \* 查看所有配置项目：

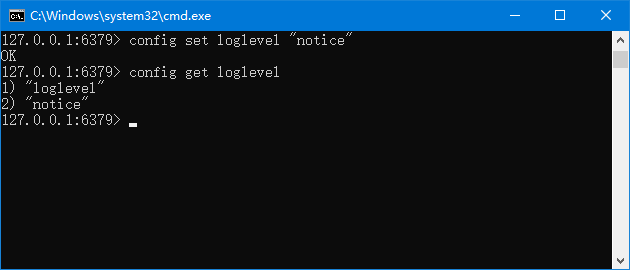


Redis提供的配置项默认清单如下：

|  |  |
| --- | --- |
| 配置项目 | 值 |
| dbfilename | dump.rdb |
| requirepass |  |
| masterauth |  |
| unixsocket |  |
| logfile |  |
| pidfile |  |
| maxmemory | 0 |
| maxmemory-samples | 5 |
| timeout | 0 |
| auto-aof-rewrite-percentage | 100 |
| auto-aof-rewrite-min-size | 67108864 |
| hash-max-ziplist-entries | 512 |
| hash-max-ziplist-value | 64 |
| list-max-ziplist-size | -2 |
| list-compress-depth | 0 |
| set-max-intset-entries | 512 |
| zset-max-ziplist-entries | 128 |
| zset-max-ziplist-value | 64 |
| hll-sparse-max-bytes | 3000 |
| lua-time-limit | 5000 |
| slowlog-log-slower-than | 10000 |
| latency-monitor-threshold | 0 |
| slowlog-max-len | 128 |
| port | 6379 |
| tcp-backlog | 511 |
| databases | 16 |
| repl-ping-slave-period | 10 |
| repl-timeout | 60 |
| repl-backlog-size | 1048576 |
| repl-backlog-ttl | 3600 |
| maxclients | 10000 |
| watchdog-period | 0 |
| slave-priority | 100 |
| min-slaves-to-write | 0 |
| min-slaves-max-lag | 10 |
| hz | 10 |
| cluster-node-timeout | 15000 |
| cluster-migration-barrier | 1 |
| cluster-slave-validity-factor | 10 |
| repl-diskless-sync-delay | 5 |
| tcp-keepalive | 0 |
| cluster-require-full-coverage | yes |
| no-appendfsync-on-rewrite | no |
| slave-serve-stale-data | yes |
| slave-read-only | yes |
| stop-writes-on-bgsave-error | yes |
| daemonize | no |
| rdbcompression | yes |
| rdbchecksum | yes |
| activerehashing | yes |
| protected-mode | yes |
| repl-disable-tcp-nodelay | no |
| repl-diskless-sync | no |
| aof-rewrite-incremental-fsync | yes |
| aof-load-truncated | yes |
| maxmemory-policy | noeviction |
| loglevel | notice |
| supervised | no |
| appendfsync | everysec |
| appendonly | no |
| dir | H:\\\xb1\xe0\xb3\xcc\xd1\xa7\xcf\xb0\\ASP.NET Core \xd1\xa7\xcf\xb0\xd6\xb8\xc4\xcf\\Asp.Net Core Session \xd7\xdc\xbd\xe1\\Redis\\Redis-x64 |
| save | jd 900 jd 300 jd 60 |
| client-output-buffer-limit | normal 0 0 0 slave 268435456 67108864 60 pubsub 33554432 8388608 60 |
| unixsocketperm | 0 |
| slaveof |  |
| notify-keyspace-events |  |
| bind | 127.0.0.1 |
|  |  |

上表的每个配置项的说明参见“redis.windows.conf”配置文件中的描述。

可以通过config set命令修改配置项，例如：



常用的重要配置项如下表所示：

|  |  |
| --- | --- |
| 配置项目 | 说明 |
| daemonize no | Redis 默认不是以守护进程的方式运行，可以通过该配置项修改，使用 yes 启用守护进程（Windows 不支持守护线程，配置为 no ） |
| pidfile /var/run/redis.pid | 当 Redis 以守护进程方式运行时，Redis 默认会把 pid 写入 /var/run/redis.pid 文件，可以通过 pidfile 指定 |
| port 6379 | 指定 Redis 监听端口，默认端口为 6379，作者在自己的一篇博文中解释了为什么选用 6379 作为默认端口，因为 6379 在手机按键上 MERZ 对应的号码，而 MERZ 取自意大利歌女 Alessia Merz 的名字 |
| bind 127.0.0.1 | 绑定的主机地址 |
| timeout 300 | 当客户端闲置多长时间后关闭连接，如果指定为0，表示关闭该功能 |
| loglevel notice | 指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 notice |
| logfile stdout | 日志记录方式，默认为标准输出，如果配置 Redis 为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给 /dev/null |
| databases 16 | 设置数据库的数量，默认数据库为0，可以使用SELECT 命令在连接上指定数据库id |
| save <seconds> <changes> | 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合。  Redis 默认配置文件中提供了三个条件：  save 900 1  save 300 10  save 60 10000  分别表示：  900 秒（15 分钟）内有 1 个更改，  300 秒（5 分钟）内有 10 个更改  60 秒内有 10000 个更改。 |
| rdbcompression yes | 指定本地数据库文件名，默认值为 dump.rdb |
| dir ./ | 指定本地数据库存放目录 |
| slaveof <masterip> <masterport> | 设置当本机为 slav 服务时，设置 master 服务的 IP 地址及端口，在 Redis 启动时，它会自动从 master 进行数据同步 |
| masterauth <master-password> | 当 master 服务设置了密码保护时，slav 服务连接 master 的密码 |
| requirepass foobared | 设置 Redis 连接密码，如果配置了连接密码，客户端在连接 Redis 时需要通过 AUTH <password> 命令提供密码，默认关闭 |
| maxclients 128 | 设置同一时间最大客户端连接数，默认无限制，Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis 会关闭新的连接并向客户端返回 max number of clients reached 错误信息 |
| maxmemory <bytes> | 指定 Redis 最大内存限制，Redis 在启动时会把数据加载到内存中，达到最大内存后，Redis 会先尝试清除已到期或即将到期的 Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis 新的 vm 机制，会把 Key 存放内存，Value 会存放在 swap 区 |
| appendonly no | 指定是否在每次更新操作后进行日志记录，Redis 在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis 本身同步数据文件是按上面 save 条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为 no |
| appendfilename appendonly.aof | 指定更新日志文件名，默认为 appendonly.aof |
| appendfsync everysec | 指定更新日志条件，共有3个可选值：  (1)no：表示等操作系统进行数据缓存同步到磁盘（快）  (2)always：表示每次更新操作后手动调用fsync()将数据写到磁盘（慢，安全）  (3)everysec：表示每秒同步一次（折中，默认值） |
| vm-enabled no | 指定是否启用虚拟内存机制，默认值为 no，简单的介绍一下，VM 机制将数据分页存放，由 Redis 将访问量较少的页即冷数据 swap 到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析 Redis 的 VM 机制） |
| vm-swap-file /tmp/redis.swap | 虚拟内存文件路径，默认值为 /tmp/redis.swap，不可多个 Redis 实例共享 |
| vm-max-memory 0 | 将所有大于 vm-max-memory 的数据存入虚拟内存，无论 vm-max-memory 设置多小，所有索引数据都是内存存储的(Redis 的索引数据 就是 keys)，也就是说，当 vm-max-memory 设置为 0 的时候，其实是所有 value 都存在于磁盘。默认值为 0 |
| vm-page-size 32 | Redis swap 文件分成了很多的 page，一个对象可以保存在多个 page 上面，但一个 page 上不能被多个对象共享，vm-page-size 是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page 大小最好设置为 32 或者 64bytes；如果存储很大大对象，则可以使用更大的 page，如果不确定，就使用默认值 |
| vm-pages 134217728 | 设置 swap 文件中的 page 数量，由于页表（一种表示页面空闲或使用的 bitmap）是在放在内存中的，，在磁盘上每 8 个 pages 将消耗 1byte 的内存。 |
| vm-max-threads 4 | 设置访问swap文件的线程数,最好不要超过机器的核数,如果设置为0,那么所有对swap文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为4 |
| glueoutputbuf yes | 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启 |
| hash-max-zipmap-entries 64  hash-max-zipmap-value 512 | 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法 |
| activerehashing yes | 指定是否激活重置哈希，默认为开启（后面在介绍 Redis 的哈希算法时具体介绍） |
| include /path/to/local.conf | 指定包含其它的配置文件，可以在同一主机上多个Redis实例之间使用同一份配置文件，而同时各个实例又拥有自己的特定配置文件 |
|  |  |

**5.Redis类型**

Redis支持五种数据类型：

string(字符串)

hash(哈希)

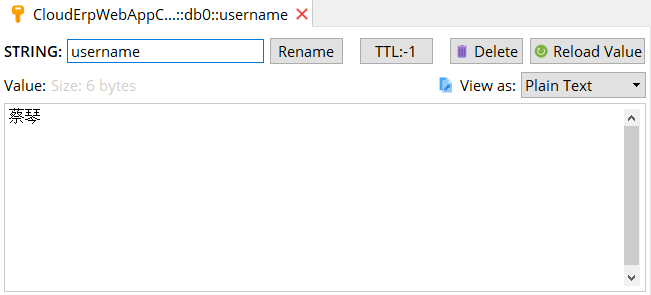
list(列表)

set(集合)

zset(sorted set, 有序集合)

**(1)** **string(字符串)**

string是redis最基本的类型。string类型是二进制安全的，意思是redis的string可以包含任何数据，例如jpg图片或者序列化的对象。string类型的值最大能存储512MB，这意味着你可以将任何序列化对象以字符串形式存储，例如Json字符串，这在缓存数据时非常有用。

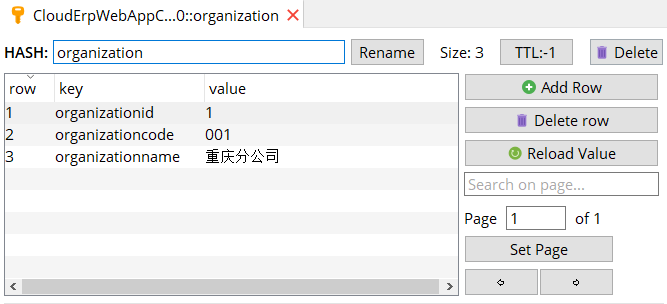


命令：

|  |
| --- |
| set username 蔡琴 // 新增  set username 张珊 // 修改  del username // 删除  get username // 查询 |

**(2)** **hash(哈希)**

hash是一个键值对(key-value)集合，即数据字典。hash特别适合用于存储对象。每个hash可以存储232 -1键值对，即一个集合的元素数量可以多达40多亿。

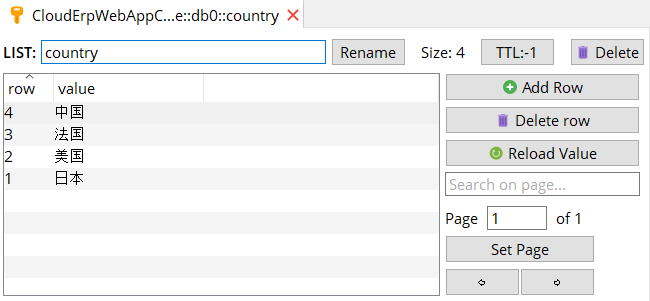


在上例中创建了一个名为organization的哈希，其key-value对是一个Organization模型的成员。由此我们可以看出hash适合用来存储对象。命令：

|  |
| --- |
| hsetnx organization organizationname 重庆分公司 // 新增  hset organization organizationname北京分公司 // 修改  hdel organization organizationname // 删除  hget organization organizationname // 查询  hscan organization 0 count 10000 // 查询(按范围)  hlen organization // 长度 |

**(3)** **list(列表)**

list是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部(lpush)或者尾部(rpush)。列表最多可存储232 – 1个元素(4294967295，每个列表可存储40多亿)。

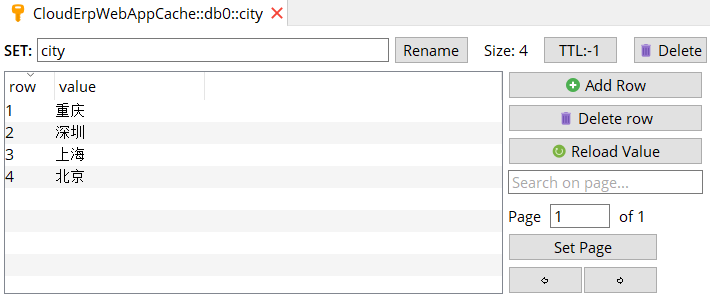


命令：

|  |
| --- |
| lpush country中国 // 新增  lset country 0 中国 // 修改  lset country 0 // 删除  lrange country 0 0 // 查询  lrange country 0 10 // 查询(按范围)  llen country // 长度 |

**(4)** **set(集合)**

set是string类型的无序集合。集合是通过哈希表实现的，所以添加、删除、查找的复杂度都是 O(1)。集合中最大的成员数为232-1(4294967295，每个集合可存储40多亿个成员)。



命令：

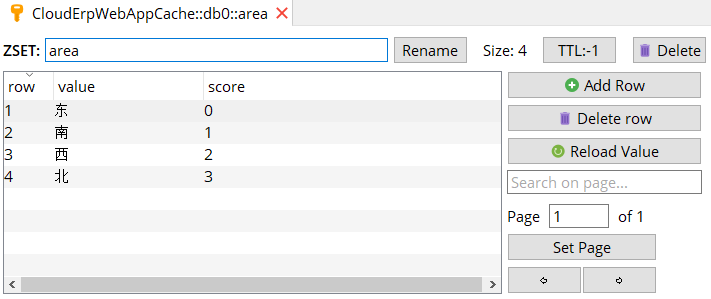
|  |
| --- |
| sadd city北京 // 新增  sadd city 上海 // 修改(没有修改命令，因此修改必须先删除后新增)  srem city 北京 // 删除  sscan city 1 // 查询  sscan city 0 count 10000 // 查询(按范围)  smembers city // 查询所有  scard city // 长度 |

注意：根据集合内元素的唯一性，相同值反复插入时后面的值会被忽略。

**(5)** zs**et(sorted set, 有序集合)**

zset和set一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis是通过分数来为集合中的成员进行从小到大排序的。zset的成员是唯一的,但分数(score)却可以重复。有序集合中最大的成员数为232-1(4294967295，每个有序集合可存储40多亿个成员)。



命令：

|  |
| --- |
| zadd area 0 东 // 新增  zadd area 0 西 // 修改(没有修改命令，因此修改必须先删除后新增)  zrem area 东// 删除  zrange area 0 0 withscores // 查询  zrange area 0 3 withscores // 查询(按范围)  zcard area // 长度 |

注意：根据集合内元素的唯一性，相同值反复插入时后面的值会被忽略。

**小结：Redis**类型的应用场景

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 简介 | 特点 | 场景 |
| String(字符串) | 二进制安全 | 可以包含任何数据,例如jpg图片或者序列化的对象,一个键最大能存储512M | 所有 |
| Hash(字典) | 键值对集合,即编程语言中的Map类型(java)或Dictionary类型(c#) | 适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(而Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去) | 存储、读取、修改用户属性 |
| List(列表) | 链表(双向链表) | 增删快,提供了操作某一段元素的API | 1. 最新消息排行等功能(比如朋友圈的时间线) 2. 消息队列 |
| Set(集合) | 哈希表实现,元素不重复 | 1. 添加、删除,查找的复杂度都是O(1) 2. 为集合提供了求交集、并集、差集等操作 | 1. 共同好友 2. 利用唯一性,统计访问网站的所有独立ip 3. 好友推荐时,根据tag求交集,大于某个阈值就可以推荐 |
| Sorted Set(有序集合) | 将Set中的元素增加一个权重参数score,元素按score有序排列 | 数据插入集合时,即进行了排序 | 1. 排行榜 2. 带权重的消息队列 |
|  |  |  |  |

**6.Redis命令**

Redis命令用于在Redis服务器上执行操作，以管理数据库中的而对象。

更多命令请参考：<https://redis.io/commands>

**(1)** **key(键)**

|  |  |
| --- | --- |
| 命令 | 描述 |
| DEL key | 功能：删除已存在的key，不存在则忽略。  返回：被删除的key的数量。 |
| DUMP key | 功能：序列化给定key  返回：被序列化的值。 |
| EXISTS key | 功能：检查给定key是否存在。  返回：若key存在返回1，否则返回0。 |
| EXPIRE key seconds | 功能：为给定key设置过期时间，以秒计。  返回： 设置成功返回1。当key不存在或者不能为key设置过期时间时(例如在低2.1.3版本的Redis中你尝试更新key的过期时间)返回0。 |
| EXPIREAT key timestamp | 功能： EXPIREAT的作用和EXPIRE类似，都用于为key设置过期时间。不同 EXPIREAT命令接受的时间参数是UNIX 时间戳(unix timestamp)。  返回： 设置成功返回1。 当key不存在或者不能为key设置过期时间时(例如在低于2.1.3版本的Redis中你尝试更新key的过期时间)返回0 。 |
| KEYS pattern | 功能：查找所有符合给定模式(pattern)的key。  返回：符合给定模式的key列表(Array)。 |
| MOVE key db | 功能：将当前数据库的key移动到给定的数据库db当中。  返回：移动成功返回1，失败则返回0。 |
| PERSIST key | 功能：移除key的过期时间，key将持久保持。  返回：当过期时间移除成功时，返回1。如果key不存在或key没有设置过期时间，返回0。 |
| PTTL key | 功能：以毫秒为单位返回key的剩余的过期时间。  返回：当key不存在时，返回-2。当key存在但没有设置剩余生存时间时，返回-1。否则，以毫秒为单位，返回key的剩余生存时间。  注意：在Redis2.8以前，当key不存在，或者key没有设置剩余生存时间时，命令都返回-1。 |
| TTL key | 功能：以秒为单位，返回给定key的剩余生存时间(TTL, time to live)。  返回：当key不存在时，返回-2。当key存在但没有设置剩余生存时间时，返回-1。否则，以秒为单位，返回key的剩余生存时间。  注意：在Redis 2.8以前，当key不存在，或者key没有设置剩余生存时间时，命令都返回-1。 |
| RANDOMKEY | 功能：从当前数据库中随机返回一个key 。  返回：当数据库不为空时，返回一个key。当数据库为空时，返回nil (windows 系统返回null)。 |
| RENAME key newkey | 功能：修改 key 的名称  返回： 改名成功时返回OK，失败时候返回一个错误。当OLD\_KEY\_NAME和NEW\_KEY\_NAME相同，或者OLD\_KEY\_NAME不存在时，返回一个错误。 当NEW\_KEY\_NAME已经存在时，RENAME 命令将覆盖旧值。 |
| RENAMENX key newkey | 功能：仅当newkey不存在时，将key改名为newkey。  返回：修改成功时，返回1。如果NEW\_KEY\_NAME已经存在，返回0 。 |
| TYPE key | 功能：返回key所储存的值的类型。  返回：返回key的数据类型，数据类型有：  none (key不存在)  string (字符串)  hash (哈希表)  list (列表)  set (集合)  zset (有序集合) |
|  |  |

**(2)** **string(字符串)**

|  |  |
| --- | --- |
| 命令 | 描述 |
| SET key value | 功能：设置指定key的值。  返回：在Redis 2.6.12以前版本，SET命令总是返回OK。从 Redis 2.6.12版本开始，SET在设置操作成功完成时，才返回 OK。 |
| GET key | 功能：获取指定key的值。  返回：返回key的值，如果key不存在时，返回nil(null)。如果key不是字符串类型，那么返回一个错误。 |
| GETRANGE key start end | 功能：返回key中字符串值的子字符。  返回：截取得到的子字符串。 |
| GETSET key value | 功能：将给定key的值设为value，并返回key的旧值(old value)。  返回：返回给定key的旧值。当key没有旧值时，即key不存在时，返回nil(null)。当key存在但不是字符串类型时，返回一个错误。 |
| GETBIT key offset | 功能：对key所储存的字符串值，获取指定偏移量上的位(bit)。返回：字符串值指定偏移量上的位(bit)。当偏移量OFFSET比字符串值的长度大，或者key不存在时，返回0。 |
| MGET key1 [key2..] | 功能：获取所有(一个或多个)给定key的值。  返回：一个包含所有给定key的值的列表。 |
| SETBIT key offset value | 功能：对key所储存的字符串值，设置或清除指定偏移量上的位(bit)。  返回：指定偏移量原来储存的位。 |
| SETEX key seconds value | 功能：将值value关联到key，并将key的过期时间设为seconds (以秒为单位)。  返回：设置成功时返回 OK 。 |
| SETNX key value | 功能：只有在key不存在时设置key的值。  返回：设置成功，返回1。设置失败，返回0。 |
| SETRANGE key offset value | 功能：用value参数覆写给定key所储存的字符串值，从偏移量 offset开始。  返回：被修改后的字符串长度。 |
| STRLEN key | 功能：返回key所储存的字符串值的长度。  返回：字符串值的长度。当key不存在时，返回0。 |
| MSET key value [key value ...] | 功能：同时设置一个或多key-value对。  返回：总是返回OK 。 |
| MSETNX key value [key value ...] | 功能：同时设置一个或多个key-value对，当且仅当所有给定key 都不存在。  返回：当所有key 都成功设置，返回1。如果所有给定key都设置失败(至少有一个key已经存在)，那么返回0。 |
| PSETEX key milliseconds value | 功能：这个命令和SETEX命令相似，但它以毫秒为单位设置key的生存时间，而不是像SETEX命令那样，以秒为单位。  返回：设置成功时返回OK。 |
| INCR key | 功能：将key中储存的数字值增一。  返回：执行INCR命令之后key的值。 |
| INCRBY key increment | 功能：将key所储存的值加上给定的增量值(increment)。  返回：加上指定的增量值之后，key的值。 |
| INCRBYFLOAT key increment | 功能：将key所储存的值加上给定的浮点增量值(increment)。  返回：执行命令之后key的值。 |
| DECR key | 功能：将key中储存的数字值减一。  返回：执行命令之后key的值。 |
| DECRBY key decrement | 功能：将key所储存的值减去给定的减量值(decrement)  返回：减去指定减量值之后，key的值。 |
| APPEND key value | 功能：如果key已经存在并且是一个字符串，APPEND命令将指定的value追加到该key原来值(value)的末尾。  返回：追加指定值之后，key中字符串的长度。 |
|  |  |

**(3)** **hash(哈希)**

|  |  |
| --- | --- |
| 命令 | 描述 |
| HDEL key field1 [field2] | 功能：删除一个或多个哈希表字段。  返回：被成功删除字段的数量，不包括被忽略的字段。 |
| HEXISTS key field | 功能：查看哈希表key中，指定的字段是否存在。  返回：如果哈希表含有给定字段，返回1。如果哈希表不含有给定字段，或key不存在，返回0。 |
| HGET key field | 功能：获取存储在哈希表中指定字段的值。  返回：返回给定字段的值。如果给定的字段或key不存在时，返回 nil(null)。 |
| HGETALL key | 功能：获取在哈希表中指定key的所有字段和值  返回：以列表形式返回哈希表的字段及字段值。若key不存在，返回空列表。 |
| HINCRBY key field increment | 功能：为哈希表key中的指定字段的整数值加上增量increment。  返回：执行HINCRBY命令之后，哈希表中字段的值。 |
| HINCRBYFLOAT key field increment | 功能：为哈希表key中的指定字段的浮点数值加上增量increment。  返回：执行Hincrbyfloat命令之后，哈希表中字段的值。 |
| HKEYS key | 功能：获取所有哈希表中的字段  返回：包含哈希表中所有域(field)列表。当key不存在时，返回一个空列表。 |
| HLEN key | 功能：获取哈希表中字段的数量。  返回：哈希表中字段的数量。当key不存在时，返回0。 |
| HMGET key field1 [field2] | 功能：获取所有给定字段的值。  返回：一个包含多个给定字段关联值的表，表值的排列顺序和指定字段的请求顺序一样。 |
| HMSET key field1 value1 [field2 value2 ] | 功能：同时将多个field-value(域-值)对设置到哈希表ke中。  返回：如果命令执行成功，返回OK。 |
| HSET key field value | 功能：将哈希表key中的字段field的值设为value。  返回：如果字段是哈希表中的一个新建字段，并且值设置成功，则返回1。如果哈希表中域字段已经存在且旧值已被新值覆盖，返回0。 |
| HSETNX key field value | 功能：只有在字段field不存在时，设置哈希表字段的值。  返回：设置成功，返回1。如果给定字段已经存在且没有操作被执行，返回0。 |
| HVALS key | 功能：获取哈希表中所有值。  返回：一个包含哈希表中所有域(field)值的列表。当key不存在时，返回一个空表。 |
| HSCAN key cursor [MATCH pattern] [COUNT count] | 功能：迭代哈希表中的键值对。  返回： |
|  |  |

**(4)** **list(列表)**

|  |  |
| --- | --- |
| 命令 | 描述 |
| BLPOP key1 [key2 ] timeout | 功能：移出并获取列表的第一个元素，如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  返回：如果列表为空，返回一个nil(null)。否则，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 key ，第二个元素是被弹出元素的值。 |
| BRPOP key1 [key2 ] timeout | 功能：移出并获取列表的最后一个元素，如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  返回：假如在指定时间内没有任何元素被弹出，则返回一个nil(null)和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的key，第二个元素是被弹出元素的值。 |
| BRPOPLPUSH source destination timeout | 功能：从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它；如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。  返回：假如在指定时间内没有任何元素被弹出，则返回一个nil(null)和等待时长。反之，返回一个含有两个元素的列表，第一个元素是被弹出元素的值，第二个元素是等待时长。 |
| LINDEX key index | 功能：通过索引获取列表中的元素。  返回：列表中下标为指定索引值的元素。如果指定索引值不在列表的区间范围内，返回nil(null)。 |
| LINSERT key BEFORE|AFTER pivot value | 功能：在列表的元素前或者后插入元素。  返回：如果命令执行成功，返回插入操作完成之后，列表的长度。 如果没有找到指定元素，返回-1。 如果key不存在或为空列表，返回 0。 |
| LLEN key | 功能：获取列表长度。  返回：列表的长度。 |
| LPOP key | 功能：移出并获取列表的第一个元素。  返回：列表的第一个元素。当列表key不存在时，返回nil(null)。 |
| LPUSH key value1 [value2] | 功能：将一个或多个值插入到列表头部。  返回：执行LPUSH命令后，列表的长度。 |
| LPUSHX key value | 功能：将一个值插入到已存在的列表头部。  返回：LPUSHX命令执行之后，列表的长度。 |
| LRANGE key start stop | 功能：获取列表指定范围内的元素。LRANGE返回列表中指定区间内的元素，区间以偏移量START和END指定。  其中0表示列表的第一个元素，1表示列表的第二个元素，以此类推。你也可以使用负数下标，以-1表示列表的最后一个元素，-2表示列表的倒数第二个元素，以此类推。  返回：一个列表，包含指定区间内的元素。 |
| LREM key count value | 功能：移除列表元素。LREM根据参数COUNT的值，移除列表中与参数VALUE相等的元素。  COUNT的值可以是以下几种：  (1)count > 0 : 从表头开始向表尾搜索，移除与VALUE相等的元素，数量为COUNT 。  (2)count < 0 : 从表尾开始向表头搜索，移除与VALUE相等的元素，数量为COUNT的绝对值。  (3)count = 0 : 移除表中所有与VALUE相等的值。  返回：被移除元素的数量。列表不存在时返回0。 |
| LSET key index value | 功能：通过索引设置列表元素的值。  返回：操作成功返回Ok，否则返回错误信息。 |
| LTRIM key start stop | 功能：对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。LTRIM对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。 下标0表示列表的第一个元素，以1表示列表的第二个元素，以此类推。你也可以使用负数下标，以-1表示列表的最后一个元素，-2表示列表的倒数第二个元素，以此类推。  返回：命令执行成功时，返回Ok。 |
| RPOP key | 功能：移除列表的最后一个元素，返回值为移除的元素。  返回：被移除的元素。当列表不存在时，返回nil(null)。 |
| RPOPLPUSH source destination | 功能：移除列表的最后一个元素，并将该元素添加到另一个列表并返回。  返回：被弹出的元素。 |
| RPUSH key value1 [value2] | 功能：在列表中添加一个或多个值。  返回：执行RPUSH操作后，列表的长度。 |
| RPUSHX key value | 功能：为已存在的列表添加值。  返回：执行RPUSHX操作后，列表的长度。 |
|  |  |

**(5)** **set(集合)**

|  |  |
| --- | --- |
| 命令 | 描述 |
| SADD key member1 [member2] | 功能：向集合添加一个或多个成员。  返回：被添加到集合中的新元素的数量，不包括被忽略的元素。 |
| SCARD key | 功能：获取集合的成员数。  返回：集合的数量。当集合key不存在时，返回0。 |
| SDIFF key1 [key2] | 功能：返回给定所有集合的差集。  返回：包含差集成员的列表。 |
| SDIFFSTORE destination key1 [key2] | 功能：返回给定所有集合的差集并存储在destination中。  返回：结果集中的元素数量。 |
| SINTER key1 [key2] | 功能：返回给定所有集合的交集。  返回：交集成员的列表。 |
| SINTERSTORE destination key1 [key2] | 功能：返回给定所有集合的交集并存储在destination中。  返回：结果集中的元素数量。 |
| SISMEMBER key member | 功能：判断member元素是否是集合key的成员。  返回：如果成员元素是集合的成员，返回1。 如果成员元素不是集合的成员，或key不存在，返回0。 |
| SMEMBERS key | 功能：返回集合中的所有成员。  返回：集合中的所有成员。 |
| SMOVE source destination member | 功能：将member元素从source集合移动到destination集合。  返回：如果成员元素被成功移除，返回1。 如果成员元素不是source 集合的成员，并且没有任何操作对destination集合执行，那么返回0。 |
| SPOP key | 功能：移除并返回集合中的一个随机元素。  返回：被移除的随机元素。当集合不存在或是空集时，返回nil(null)。 |
| SRANDMEMBER key [count] | 功能：返回集合中一个或多个随机数。  返回：只提供集合key参数时，返回一个元素；如果集合为空，返回 nil(null)。如果提供了count参数，那么返回一个数组；如果集合为空，返回空数组。 |
| SREM key member1 [member2] | 功能：移除集合中一个或多个成员。  返回：被成功移除的元素的数量，不包括被忽略的元素。 |
| SUNION key1 [key2] | 功能：返回所有给定集合的并集。  返回：并集成员的列表。 |
| SUNIONSTORE destination key1 [key2] | 功能：所有给定集合的并集存储在destination集合中。  返回：结果集中的元素数量。 |
| SSCAN key cursor [MATCH pattern] [COUNT count] | 功能：迭代集合中的元素  返回：数组列表。 |
|  |  |

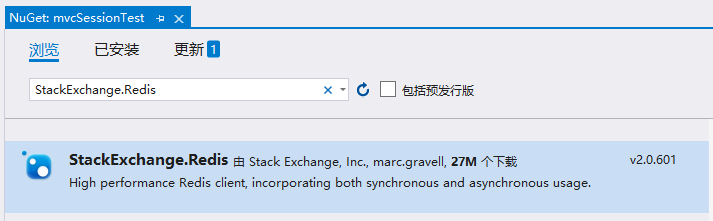
**(6)** z**set(sorted set, 有序集合)**

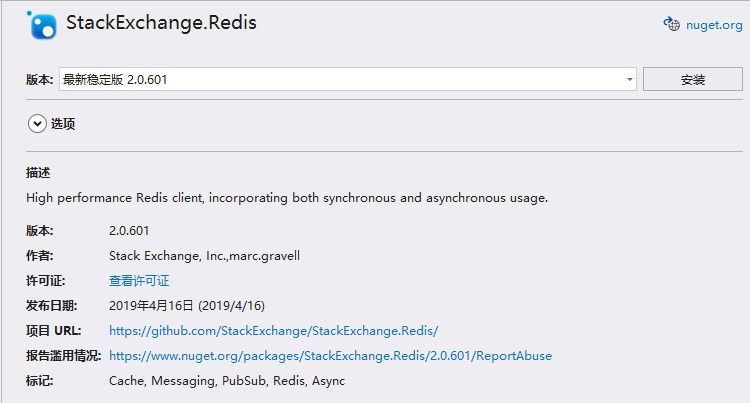
|  |  |
| --- | --- |
| 命令 | 描述 |
| ZADD key score1 member1 [score2 member2] | 功能：向有序集合添加一个或多个成员，或者更新已存在成员的分数。  返回：被成功添加的新成员的数量，不包括那些被更新的、已经存在的成员。 |
| ZCARD key | 功能：获取有序集合的成员数。  返回：当key存在且是有序集类型时，返回有序集的数量。当key不存在时，返回0。 |
| ZCOUNT key min max | 功能：计算在有序集合中指定区间分数的成员数。  返回：分数值在min和max之间的成员的数量。 |
| ZINCRBY key increment member | 功能：有序集合中对指定成员的分数加上增量increment。  返回：member成员的新分数值，以字符串形式表示。 |
| ZINTERSTORE destination numkeys key [key ...] | 功能：计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合key中。  返回：保存到目标结果集的的成员数量。 |
| ZLEXCOUNT key min max | 功能：在有序集合中计算指定字典区间内成员数量。  返回：指定区间内的成员数量。 |
| ZRANGE key start stop [WITHSCORES] | 功能：通过索引区间返回有序集合指定区间内的成员。Zrange 返回有序集中，指定区间内的成员。 其中成员的位置按分数值递增(从小到大)来排序。 具有相同分数值的成员按字典序(lexicographical order )来排列。  如果你需要成员按值递减(从大到小)来排列，请使用ZREVRANGE命令。下标参数start和stop都以0为底，也就是说，以0表示有序集第一个成员，以1表示有序集第二个成员，以此类推。  你也可以使用负数下标，以-1表示最后一个成员，-2表示倒数第二个成员，以此类推。  返回：指定区间内，带有分数值(可选)的有序集成员的列表。 |
| ZRANGEBYLEX key min max [LIMIT offset count] | 功能：通过字典区间返回有序集合的成员。  返回：指定区间内的元素列表。 |
| ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT] | 功能：通过分数返回有序集合指定区间内的成员。  返回：指定区间内，带有分数值(可选)的有序集成员的列表。 |
| ZRANK key member | 功能：返回有序集合中指定成员的索引。  返回：如果成员是有序集key的成员，返回member 的排名。如果成员不是有序集key的成员，返回nil(null)。 |
| ZREM key member [member ...] | 功能：移除有序集合中的一个或多个成员。  返回：被成功移除的成员的数量，不包括被忽略的成员。 |
| ZREMRANGEBYLEX key min max | 功能：移除有序集合中给定的字典区间的所有成员。  返回：被移除成员的数量。 |
| ZREMRANGEBYRANK key start stop | 功能：移除有序集合中给定的排名区间的所有成员。  返回：指定区间内，带有分数值(可选)的有序集成员的列表。 |
| ZREMRANGEBYSCORE key min max | 功能：移除有序集合中给定的分数区间的所有成员。  返回：被移除成员的数量。 |
| ZREVRANGE key start stop [WITHSCORES] | 功能：返回有序集中指定区间内的成员，通过索引，分数从高到低。  返回：指定区间内，带有分数值(可选)的有序集成员的列表。 |
| ZREVRANGEBYSCORE key max min [WITHSCORES] | 功能：返回有序集中指定分数区间内的成员，分数从高到低排序。  返回：指定区间内，带有分数值(可选)的有序集成员的列表。 |
| ZREVRANK key member | 功能：返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序。  返回：如果成员是有序集key的成员，返回成员的排名。如果成员不是有序集key的成员，返回nil(null)。 |
| ZSCORE key member | 功能：返回有序集中，成员的分数值。  返回：成员的分数值，以字符串形式表示。 |
| ZUNIONSTORE destination numkeys key [key ...] | 功能：计算给定的一个或多个有序集的并集，并存储在新的key中。  返回：保存到destination的结果集的成员数量。 |
| ZSCAN key cursor [MATCH pattern] [COUNT count] | 功能：迭代有序集合中的元素(包括元素成员和元素分值)。  返回：返回的每个元素都是一个有序集合元素，一个有序集合元素由一个成员(member)和一个分值(score)组成。 |
|  |  |

**7.在Asp.Net Core中使用Redis缓存Session**

**(1)** **安装StackExchange.Redis**

在NuGet包管理中输入：StackExchange.Redis，如下图所示：

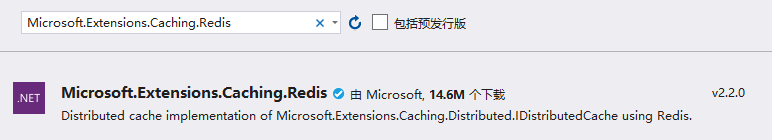


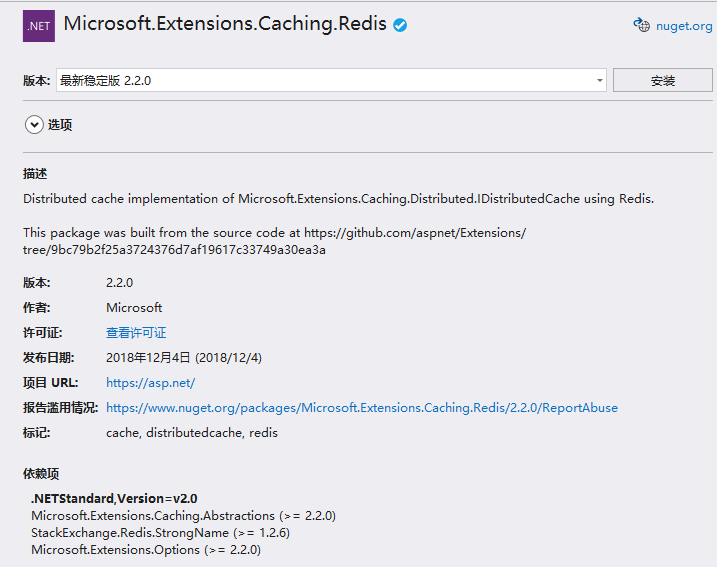


源码路径：<https://github.com/StackExchange/StackExchange.Redis/>

**(2)** **安装Microsoft.Extensions.Caching.Redis**

在NuGet包管理中输入：Microsoft.Extensions.Caching.Redis，如下图所示：



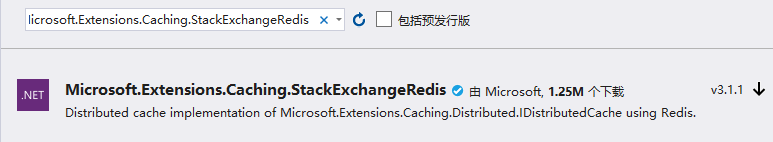


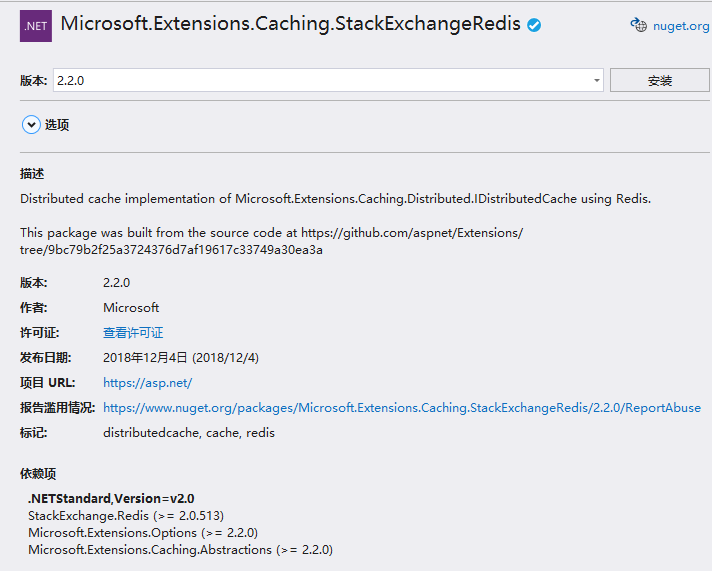
源码路径：<https://github.com/aspnet/Extensions/tree/9bc79b2f25a3724376d7af19617c33749a30ea3a>

注意：从Microsoft.Extensions.Caching.Redis 2.2.0开始，

微软已经正式将其更名为：Microsoft.Extensions.Caching.StackExchangeRedis

在NuGet包管理中输入：Microsoft.Extensions.Caching.StackExchangeRedis，如下图所示：





源码路径：<https://github.com/aspnet/Extensions/tree/9bc79b2f25a3724376d7af19617c33749a30ea3a>

注意：同为2.2.0版本

Microsoft.Extensions.Caching.Redis和

Microsoft.Extensions.Caching.StackExchangeRedis

的依赖项目完全不一样

Microsoft.Extensions.Caching.Redis 依赖StackExchange.Redis.StrongName (>= 1.2.6)

Microsoft.Extensions.Caching.StackExchangeRedis 依赖 StackExchange.Redis (>= 2.0.513)

这说明从2.2.0开始StackExchangeRedis正式命名为：StackExchange.Redis

而微软也正式和StackExchangeRedis保持一致，因此应该选择：

Microsoft.Extensions.Caching.StackExchangeRedis

下载并安装完成后如下：



**(3)** **配置Redis连接字符串**

打开appsettings.json文件，增加Redis数据库连接串：

|  |
| --- |
| {  "Logging": {  "LogLevel": {  "Default": "Warning"  }  },  "ConnectionStrings": {  "RedisCacheConn": "127.0.0.1:6379,allowAdmin=true,password=clouderpwebappcache,defaultdatabase=0"  },  "AllowedHosts": "\*"  } |

**(4)** **配置Redis缓存服务**

在Startup.cs文件的的ConfigureServices(IServiceCollection services)中增加Redis缓存服务：

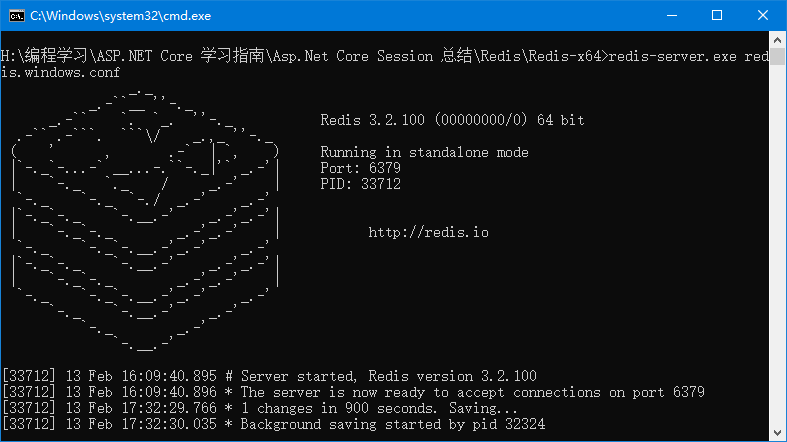
|  |
| --- |
| // 增加Redis缓存服务  services.AddStackExchangeRedisCache(options =>  {  options.Configuration = Configuration.GetConnectionString("RedisCacheConn");  options.InstanceName = "RedisCache";  }); |

**(5)** **测试Redis缓存服务**

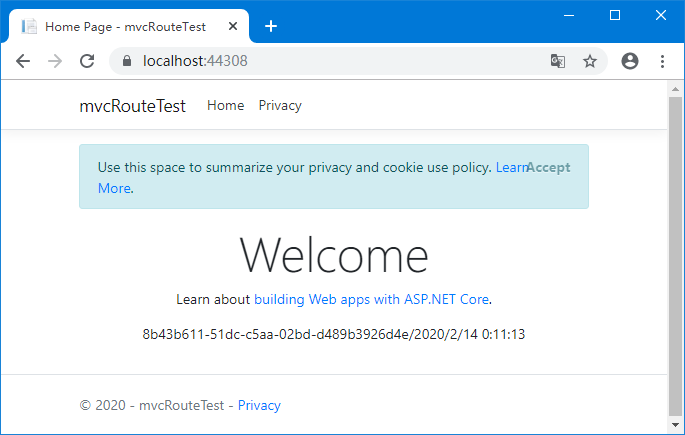
在HomeController控制器的中输入：

|  |
| --- |
| public IActionResult Index()  {  ViewBag.SessionID = HttpContext.Session.Id + "/" + DateTime.Now;  // https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/app-state?view=aspnetcore-3.1#session-state  // 会话状态具有的行为：不会保留空会话：会话中必须设置了至少一个值以保存所有请求的会话。会话未保留时，为每个新的请求生成新会话ID。  // 这意味着你想要得到一个持久的会话，必须为当前会话设置至少一个值，否则当你刷新页面或者跳转页面的时候都会生成新的会话ID，因为系统认为没有值的会话不需要保留。  // 因此，屏蔽下面的代码，每次F5刷新页面时你都会看到一个新的会话ID。  HttpContext.Session.SetString("Time", DateTime.Now.ToString());    return View();  } |

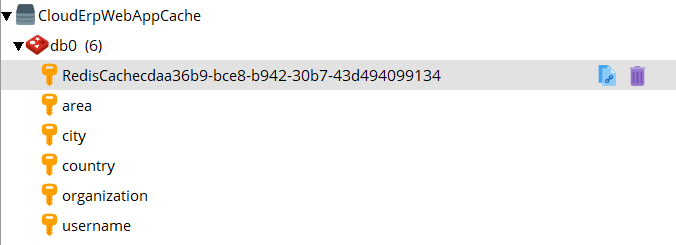
确保Redis服务器正在运行中：

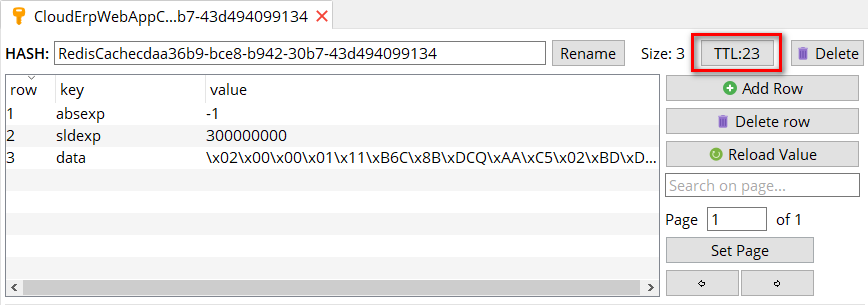


以调试模式启动App：

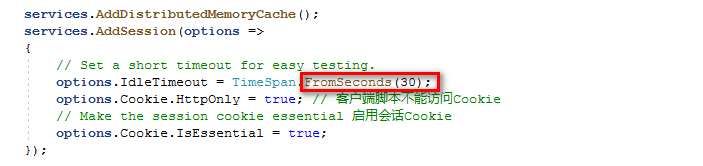


此时用Redis Desktop Manager打开Redis我们会看到会话已经缓存到Redis中：

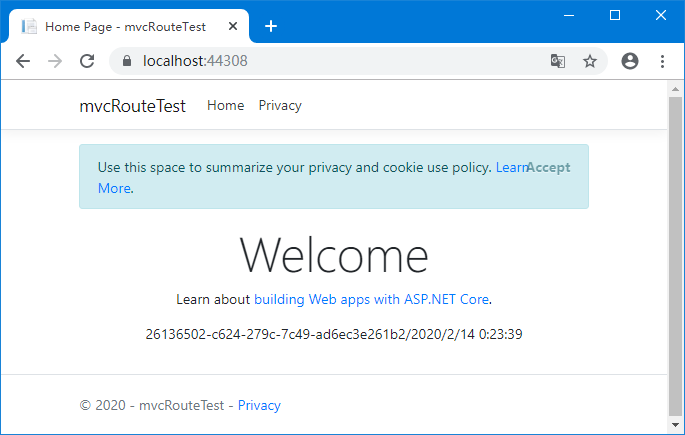


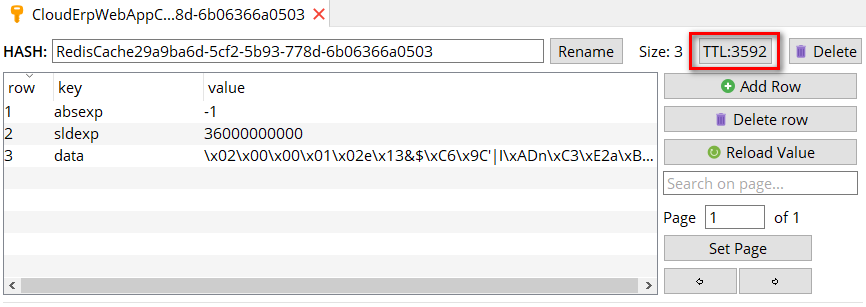


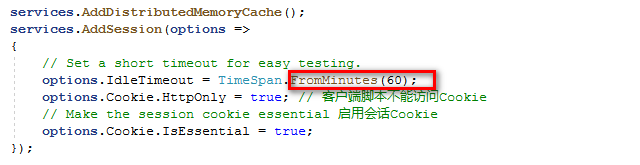
此时我们注意到缓存的会话Key的过期时间是30秒，和Session的超时时间一致：



现在把Session在无响应的情况下的过期时间改成1小时，后重新启动App：

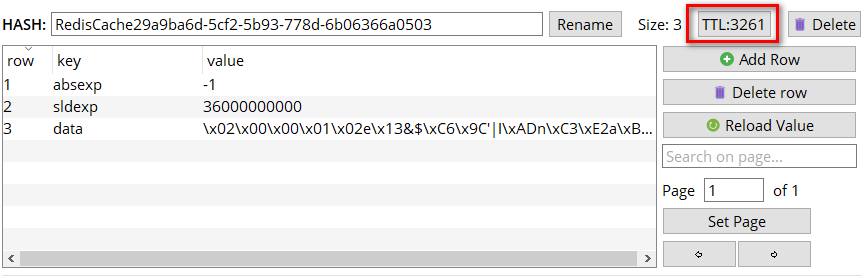




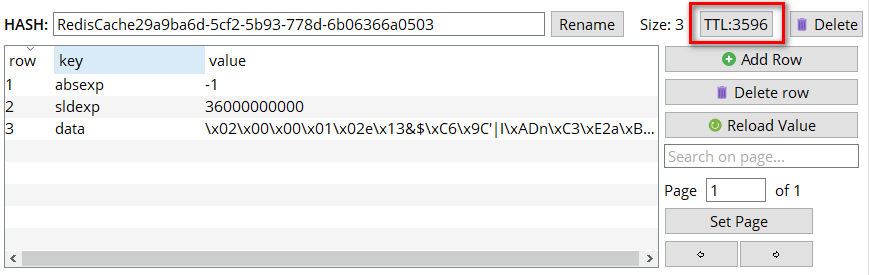


说明Redis缓存的会话的过期时间和客户端的实际的会话过期时间是保持一致的(注意：误差几秒是因我查看不及时的原因)。

过一段时间以后，查看Redis中会话的TLL：



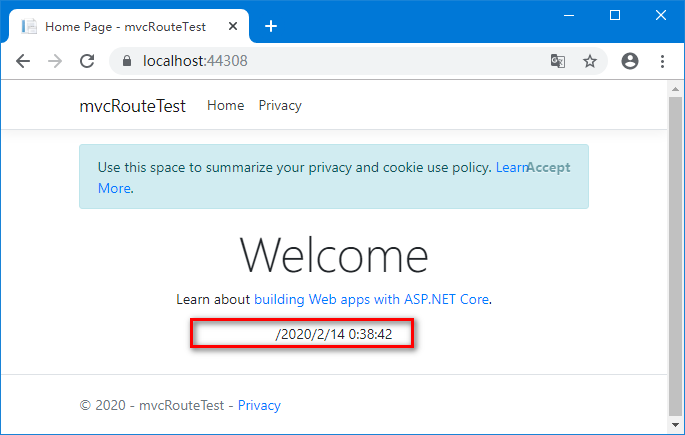
现在在浏览器中按F5刷新一下页面，让会话处于活动状态，再看Redis中会话的TLL：



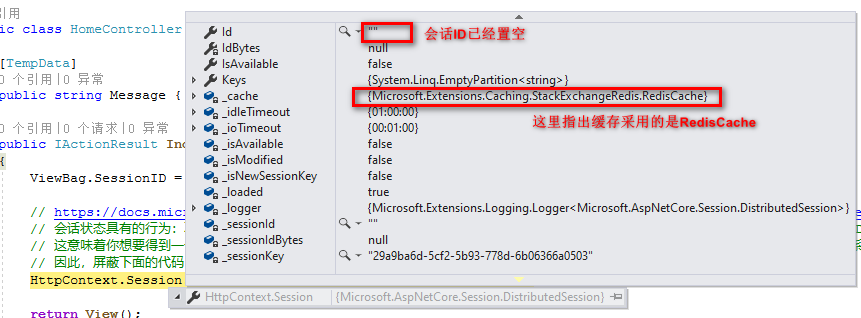
此时我们看到TTL又恢复到了3600秒，说明刷新页面的时候客户端同步了Redis中会话的TTL。

现在验证在后端缓存断开的情况下会不会抛出异常。

关闭Redis服务器，然后在浏览器中按F5刷新页面：



没有抛出错误，但是会话ID已经不存在，出现异常：



结论：采用数据库缓存会话的风险在于,一旦数据库关闭或崩溃就会导致当前会话无效，同时每刷新一次页面都会连接数据库，如果连接时间超时时间是30秒，则页面要等待30秒连接失败以后才能显示出来，这是不可接受的。

**8.Redis通信协议**

参考: <https://redis.io/topics/protocol> (官方)

**(1)** **简介**

Redis客户端使用一种称为RESP协议(REdis Serialization Protocol，Redis序列化协议)与Redis服务器通信。虽然该协议是专门为Redis设计的，但它可以用于其他客户机-服务器(client-server)软件项目。

RESP协议主要是从以下方面进行考虑后设计的结果：

|  |
| --- |
| 易于实现  快速解析  人类可读 |

ESP可以序列化不同的数据类型，例如整数(integers)、字符串(strings)、数组(arrays)，还有一个特定的错误(errors)类型。请求以字符串数组(arrays of strings)的形式从客户端发送到Redis服务器，这些字符串表示要执行的命令和参数。Redis使用特定于命令的数据类型(command-specific data type)进行响应。

RESP是二进制安全的(binary-safe)(这意味着可以传输二进制数据)，不需要处理从一个进程传输到另一个进程的批量数据(bulk data)，因为它使用前缀长度(prefixed-length)来传输批量数据(bulk data)。

注意:这里列出的协议仅用于客户机-服务器通信（client-server communication）。Redis集群(Redis Cluster)使用不同的二进制协议(binary protocol)在节点之间交换消息。

**(2)** **网络层**(Networking layer)

客户端连接到Redis服务器，在6379端口上创建TCP连接(以用于传输数据到服务器)。

虽然RESP在技术上不特定于TCP，但在Redis上下文中，该协议仅用于TCP连接(或类似Unix sockets的面向流的连接)。

**(3) 请求响应模型**(Request-Response model)

Redis接受由不同参数(arguments)组成的命令(commands)。服务器接收到命令后，将对其进行处理并将应答(reply)发送回客户端。

这是最简单的模型，但有两个例外:

(1)Redis支持管道(pipele)操作(本文档后面将介绍)。因此，客户端可以同时发送多个命令(multiple commands)，然后等待稍后的回复。

(2)当Redis客户端订阅了发布/订阅频道(Pub/Sub channel),协议语义变化,成为推动协议(push protocol),也就是说,客户不再需要发送命令,因为服务器会自动发送新消息到客户端。

排除上述两个例外，Redis协议是一个简单的请求-响应协议(request-response protocol)。

**(4) RESP协议描述**(RESP protocol description)

RESP协议在Redis 1.2中引入，但在Redis 2.0中成为与Redis服务器通信的标准方式。这是你应该在你的Redis客户端实现的协议(也就是说要与Redis服务器通信数据必须采用RESP协议)。

RESP实际上是一个序列化协议(serialization protocol)，它支持以下数据类型(data types):

|  |
| --- |
| 简单字符串(Simple Strings);  错误(Errors);  整数(Integers);  大容量字符串(Bulk Strings);  数组(Arrays)。 |

RESP在Redis中用作请求-响应协议(request-response protocol)的方式如下:

（1）客户端请求(request)：客户端将命令封装成由大容量字符串组成的RESP数组（RESP Array of Bulk Strings）发送到Redis服务器（也就是说客户端发送的是一个数组，数组的内容是由大容量字符串组成的二进制数据）

（2）服务器响应(response)：服务器根据收到的命令(command)采用不同的RESP类型进行响应。在RESP中，某些数据的响应类型取决于第一个字节：

|  |
| --- |
| 对于简单字符串(Simple Strings)，应答的第一个字节是"+"。  对于错误(Errors)，应答的第一个字节是"-"。  对于整数(Integers)，应答的第一个字节是":"。  对于大容量字符串(Bulk Strings)，应答的第一个字节是"$"。  对于数组(Arrays)，应答的第一个字节是"\*”。 |

此外，RESP还可以使用稍后指定的特殊大容量字符串(Bulk Strings)或数组(Array)变体来表示空值(Null)。在RESP中，协议的不同部分总是用"\r\n"(CRLF)终止(terminated)。

**(5) RESP简单字符串**(RESP Simple Strings)

简单字符串的编码方式如下：

|  |
| --- |
| 一个"+"字符，后跟一个不能包含CR或LF字符的字符串(不允许换行)，然后由CRLF(即"\r\n")结束。 |

简单字符串用于传输开销最小的非二进制安全字符串。例如，许多Redis命令回复成功时只有"OK"，作为一个RESP简单字符串编码为以下5个字节:

|  |
| --- |
| +OK\r\n |

为了发送二进制安全(binary-safe)的字符串，请使用RESP大容量字符串(Bulk Strings)替代。

当Redis使用一个简单的字符串进行应答时，客户端应该向调用者返回"+"后面的字符串，不包括最后的CRLF字符。

**(6) RESP错误**(RESP Errors)

RESP为错误设计了一个特定的数据类型。实际上，错误与RESP简单字符串完全相同，但是第一个字符是"-"字符，而不是"+"字符。

简单字符串和RESP中的错误之间的真正区别是，客户端将错误视为异常。它的基本格式为:

|  |
| --- |
| -Error message\r\n |

错误回复只在发生错误时发送，例如当您试图对错误的数据类型执行操作时，或者当命令不存在时，等等。当接收到错误回复时，客户端应引发异常。下面是错误回复的例子:

|  |
| --- |
| -ERR unknown command 'foobar'  -WRONGTYPE Operation against a key holding the wrong kind of value |

"-"后面的第一个单词，直到第一个空格或换行，表示返回的错误类型。这只是Redis使用的约定，不属于RESP错误格式的一部分。

例如，ERR是通用错误，而WRONGTYPE是更具体的错误，它表示客户端试图对错误的数据类型执行操作。这称为错误前缀(Error Prefix)允许客户端理解服务器返回的错误类型，而不需要依赖于所给出的确切消息，因为消息可能会随着时间的推移而变化。

客户端可以根据不同的错误返回不同类型的异常，或者直接将错误消息作为字符串提供给调用者来提供捕获错误的通用方法。

但是，这样的特性不应该被认为是至关重要的，因为它很少有用，而且有限的客户端实现可能只返回一般的错误条件，例如false。

**(7) RESP整数**(RESP Integers)

整数类型只是一个以CRLF结尾的字符串，表示一个整数，前面有一个":"字符。例如，":0\r\n"或":1000\r\n"是整数。整数的编码方式如下：

|  |
| --- |
| 一个":"字符，后跟一个有符号的64位整数，然后由CRLF结束。 |

许多Redis命令返回RESP整数，例如[INCR](https://redis.io/commands/incr), [LLEN](https://redis.io/commands/llen) 和 [LASTSAVE](https://redis.io/commands/lastsave)。

对于返回的整数没有特殊意义，它只是代表[INCR](https://redis.io/commands/incr)命令的一个增量数，[LASTSAVE](https://redis.io/commands/lastsave)命令的一个UNIX时间，等等。但是，返回的整数保证在有符号的64位整数范围内(a signed 64 bit integer)。

整数应答也广泛用于返回真或假。例如，[EXISTS](https://redis.io/commands/exists)或[SISMEMBER](https://redis.io/commands/sismember)之类的命令将返回1表示真，0表示假。其他命令，如[SADD](https://redis.io/commands/sadd)、[SREM](https://redis.io/commands/srem)和[SETNX](https://redis.io/commands/setnx)，如果实际执行了操作，则返回1，否则返回0。

以下命令将以整数应答: [SETNX](https://redis.io/commands/setnx), [DEL](https://redis.io/commands/del), [EXISTS](https://redis.io/commands/exists), [INCR](https://redis.io/commands/incr), [INCRBY](https://redis.io/commands/incrby), [DECR](https://redis.io/commands/decr), [DECRBY](https://redis.io/commands/decrby), [DBSIZE](https://redis.io/commands/dbsize), [LASTSAVE](https://redis.io/commands/lastsave), [RENAMENX](https://redis.io/commands/renamenx), [MOVE](https://redis.io/commands/move), [LLEN](https://redis.io/commands/llen), [SADD](https://redis.io/commands/sadd), [SREM](https://redis.io/commands/srem), [SISMEMBER](https://redis.io/commands/sismember), [SCARD](https://redis.io/commands/scard).

**(9) RESP大容量字符串**(RESP Bulk Strings)

大容量字符串用于表示单个二进制安全字符串，长度不超过512MB。 (二进制安全是指不对任何字节进行特殊标识、转码等，也就是按字节处理，原样输出)。

大容量字符串的编码方式如下：

|  |
| --- |
| 一个"$"字符，后跟组成字符串的字节数，然后由CRLF结束。  实际的字符串数据。  CRLF终结符。 |

因此字符串"foobar"被编码如下:

|  |
| --- |
| $6\r\nfoobar\r\n |

一个空字符串(empty string)时:

|  |
| --- |
| $0\r\n\r\n |

大容量字符串也可以使用表示空值的特殊格式来表示值不存在。在这种特殊的格式中，长度是-1，没有数据，所以Null表示为:

|  |
| --- |
| $-1\r\n |

这称为**Null Bulk String**。客户端API不应该返回空字符串(empty string)，而应该返回nil对象(null对象)，当服务器使用Null Bulk String进行应答时。例如，Ruby应该返回'nil'，而C应该返回NULL，C#应返回null等等。

**(10) RESP数组**(RESP Arrays)

这里再次强调，客户端使用RESP数组向Redis服务器发送命令。

类似地，某些使用RESP数组向客户端返回元素集合的Redis命令是应答类型。一个例子是返回列表元素的[LRANGE](https://redis.io/commands/lrange)命令。

RESP数组使用以下格式发送数据给服务器:

|  |
| --- |
| 一个"\*"字符作为第一个字节，后跟一个十进制数表示数组中的元素数量，然后由CRLF结束。  用RESP类型表示的每个元素。 |

所以，空数组是这样的:

|  |
| --- |
| \*0\r\n |

例如，当数组的元素由2两个RESP Bulk Strings ："foo"和"bar"组成的时候，数组被编码为:

|  |
| --- |
| \*2\r\n$3\r\nfoo\r\n$3\r\nbar\r\n |

可以看到，在\*<count>CRLF部分为数组加上前缀之后，后面组成数组元素的其他RESP类型只是一个接一个地连接起来。例如，一个有三个整数的数组编码如下:

|  |
| --- |
| \*3\r\n:1\r\n:2\r\n:3\r\n |

因此数组的元素可以包含混合类型(mixed types)，即元素不必具有相同的RESP类型。例如，一个包含四个整数和一个大容量字符串的列表可以编码如下:

|  |
| --- |
| \*5\r\n  :1\r\n  :2\r\n  :3\r\n  :4\r\n  $6\r\n  foobar\r\n |

(为了清晰起见，回复也被分成了几行)

服务器发送的第一行是\*5\r\n，以便指定接下来的5个响应。然后，发送构成多个批量应答项(Multi Bulk reply)的每个应答。

空数组(Null Array)的概念也存在，它是指定空值的另一种方法(通常使用Null Bulk String，但由于历史原因，我们有两种格式)。

例如，当[BLPOP](https://redis.io/commands/blpop)命令超时时，它返回一个空数组，其计数为-1，如下例所示:

|  |
| --- |
| \*-1\r\n |

当Redis返回一个空数组时，客户端API应该返回一个空对象(null object)，而不是一个空数组(Null Array)。这对于区分空列表和不同的条件(例如BLPOP命令的超时条件)是必要的。

在RESP中可以使用数组的数组(即数组中包含数组)。例如，两个数组的数组编码如下:

|  |
| --- |
| \*2\r\n  \*3\r\n  :1\r\n  :2\r\n  :3\r\n  \*2\r\n  +Foo\r\n  -Bar\r\n |

(为了便于阅读，该格式也被分成了多行).

上面的RESP数据类型编码一个双元素数组，该数组包含三个整数1、2、3和一个由简单字符串和错误构成的数组。

**(11) RESP数组中的空元素**(Null elements in RESP Arrays)

数组的单个元素可以为空(Null)。这在Redis的回复中使用，以表明这些元素是缺失的，而不是空字符串(empty strings)。当与GET模式(GET pattern)选项一起使用时，当指定的键丢失时，排序命令可能会发生这种情况。一个数组回复包含一个空元素的例子:

|  |
| --- |
| \*3\r\n  $3\r\n  foo\r\n  $-1\r\n  $3\r\n  bar\r\n |

第二个元素是Null。客户端应该返回如下内容:

|  |
| --- |
| ["foo",null,"bar"] |

注意，这并不是前面几节中提到的一个例外，只是一个进一步指定协议的示例。

**(12) 向Redis服务器发送命令**(Sending commands to a Redis Server)

现在您已经熟悉了RESP序列化格式（RESP serialization format），编写一个Redis客户端库的实现将会很容易。这对你用编程语言(例如C/C++/C#/Java等)实现一个Redis客户端至关重要。现在进一步强调客户端和服务器之间的交互是如何工作的:

|  |
| --- |
| A client sends the Redis server a RESP Array consisting of just Bulk Strings.  A Redis server replies to clients sending any valid RESP data type as reply.  客户端向Redis服务器发送一个由大容量字符串组成的RESP数组。  Redis服务器向客户端发送的任何有效的RESP数据类型作为应答。 |

例如，一个典型的相互交互是这样的：

客户端发送命令**LLEN mylist**，以获得存储在mylist中的列表长度，服务器使用整数应答，如下面的示例所示(C:是客户端，S:服务器)。

|  |
| --- |
| C: \*2\r\n  C: $4\r\n  C: LLEN\r\n  C: $6\r\n  C: mylist\r\n  S: :48293\r\n |

为了简单起见，我们通常用换行(newlines)来分隔协议的不同部分，但是实际的交互是客户端将\*2\r\n$4\r\nLLEN\r\n$6\r\nmylist\r\n作为一个整体发送给服务器。

**(13) 多个命令和管道**(Multiple commands and pipelining)

客户端可以使用相同的连接来发出多个命令(multiple commands)。

Redis支持管道(即流水线作业)，因此客户端可以通过一个写操作发送多个命令，而不需要在发出下一个命令之前读取上一个命令的服务器应答。所有的回复都可以在最后看到。

有关更多信息，请查看我们关于管道的页面：[about Pipelining](https://redis.io/topics/pipelining)。

**(14) 内联命令**(Inline Commands)

有时你只有telnet在手，你需要发送一个命令到Redis服务器。虽然Redis协议易于实现，但在交互式会话中使用它并不理想，而且Redis-cli可能并不总是可用。出于这个原因，Redis还以一种专门为人类设计的方式接受命令，这种方式被称为内联命令(inline command)。简单地说内联命令就是直接命令，只需要在终端输入Redis支持的命令即能和服务器进行交互通信。

下面是一个使用内联命令的服务器/客户端聊天示例(S:代表服务器，C:代表客户端)

|  |
| --- |
| C: PING  S: +PONG |

下面是另一个内联命令返回整数的例子:

|  |
| --- |
| C: EXISTS somekey  S: :0 |

基本上，您只需在telnet会话中编写空格分隔的参数。由于没有在统一请求协议中使用以\*开头的命令，所以Redis能够检测此条件并解析您的命令。

**(15) 用于Redis协议的高性能解析器**(High performance parser for the Redis protocol)

虽然Redis协议非常易于阅读和实现，但它的实现性能类似于二进制协议。

RESP使用带前缀的长度(prefixed lengths)来传输大量数据(transfer bulk data)，因此不需要扫描负载(payload)以查找特殊字符(例如JSON)，也不需要引用需要发送到服务器的负载。

对于Bulk和Multi Bulk length，可以使用代码进行处理，每个字符执行一个操作，同时扫描CR字符，如下所示的C代码，读取字符串的长度:

|  |
| --- |
| #include <stdio.h>  int main()  {  const char\* p = "$123\r\n";  int len = 0;  p++; // 跳过前缀  while (\*p != '\r') {  len = (len \* 10) /\*将位数扩大\*/ + (\*p - '0')/\*分解出数字\*/;  p++;  }  /\* Now p points at '\r', and the len is in bulk\_len. \*/  printf("%d\n", len); // 输出： 123  return 0;  } |

识别出第一个CR后，可以与下面的LF一起跳过，而不进行任何处理。然后，可以使用不以任何方式检查有效负载的单个读操作读取批量数据。最后，将剩余的CR和LF字符丢弃，不进行任何处理。

虽然Redis协议在性能上可以与二进制协议相媲美，但在大多数非常高级的语言中实现起来却要简单得多，从而减少了客户端软件中的bug。

<https://reservation.weihanli.xyz/>

<https://github.com/gbraad/DistributedCache>

<https://github.com/shichaohu/SYSD.Service.DistributedCache/tree/master/SYSD.Service.DistributedCache>

<https://www.cnblogs.com/szlblog/p/9045209.html>

<https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/memory?view=aspnetcore-3.1>

与缓存标记帮助程序共享的属性

enabled， expires-on， expires-after， expires-sliding， vary-by-header，vary-by-query， vary-by-route， vary-by-cookie， vary-by-user， vary-by priority。

分布式缓存标记帮助程序继承自与缓存标记帮助程序相同的类。 有关这些属性的说明，请参阅[缓存标记帮助程序](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/views/tag-helpers/built-in/cache-tag-helper?view=aspnetcore-3.1)。

|  |  |  |
| --- | --- | --- |
| 属性类型 | 示例 | 默认 |
| String | my-distributed-cache-unique-key-101 |  |

name 是必需的。 name 属性用作每个存储的缓存实例的键。 分布式缓存标记帮助程序分配缓存键时只以属性 name 上的键为基础，这点与缓存标记帮助程序不同，后者基于 Razor 页面中的 Razor 页面名称和位置为每个实例分配缓存键。示例:

|  |
| --- |
| <distributed-cache name="my-distributed-cache-unique-key-101">  Time Inside Cache Tag Helper: @DateTime.Now  </distributed-cache> |

ASP.NET Core 中内置了 [IDistributedCache](https://docs.microsoft.com/dotnet/api/microsoft.extensions.caching.distributed.idistributedcache) 的两个实现。 一个是基于 SQL Server，另一个是基于 Redis。 有关这些实现的详细信息，请参阅 [ASP.NET Core 中的分布式缓存](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed?view=aspnetcore-3.1)。 这两种实现都需要在 Startup 中设置 IDistributedCache 的实例。没有专门与使用 IDistributedCache 的任何特定实现相关的标记属性。