Red Hat Single Sign-On 7.6

服务器开发人员指南

用于 Red Hat Single Sign-On 7.6

Last Updated: 2023-03-02

Red Hat Single Sign-On 7.6 服务器开发人员指南

用于 Red Hat Single Sign-On 7.6

法律通告

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

本指南包括开发人员自定义 Red Hat Single Sign-On 7.6 的信息

目录

使开源包含更多 4

第 1 章 前言 5

第 2 章 ADMIN REST API 6
- 2.1. 使用 CURL 的示例 6
- 2.2. 其他资源 7

第 3 章 THEMES 8
- 3.1. ME 类型 8
- 3.2. 配置主题 8
- 3.3. 默认主题 9
- 3.4. 创建主题 9
- 3.5. 部署它们 18
- 3.6. ME 选择器 19
- 3.7. ME 资源 19
- 3.8. 区域设置选择器 19
- 3.9. 其他资源 20

第 4 章 自定义用户属性 21
- 4.1. 注册页面 21
- 4.2. 帐户管理控制台 21
- 4.3. 其他资源 22

第 5 章 IDENTITY BROKERING API 23
- 5.1. 检索外部 IDP 令牌 23
- 5.2. 客户端发起的帐户链接 23

第 6 章 服务提供商接口(SPI) 27
- 6.1. 实施 SPI 27
- 6.2. 使用可用供应商 30
- 6.3. 注册供应商实现 31
- 6.4. 使用 JAKARTA EE 32
- 6.5. JAVASCRIPT 供应商 34
- 6.6. 可用的 SPI 38

第 7 章 USER STORAGE SPI 40
- 7.1. 供应商接口 40
- 7.2. 供应商功能接口 42
- 7.3. 型号接口 43
- 7.4. 打包和部署 44
- 7.5. 配置技术 51
- 7.6. 添加/删除用户并查询功能接口 54
- 7.7. 增强外部存储 59
- 7.8. 导入实施策略 61
- 7.9. 用户缓存 64
- 7.10. 使用 JAKARTA EE 66
- 7.11. REST 管理 API 68
- 7.12. 从较早的用户联合 SPI 迁移 70
- 7.13. 基于流的接口 72

第 8 章 VAULT SPI 74
- 8.1. VAULT 供应商 74

8.2. 从 VAULT 消耗值 74

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。有关更多详情，请参阅我们的首席技术官 Chris Wright 提供的消息。

第 1 章 前言

在某些示例列表中，要在一个行中显示哪些内容不适合可用页面宽度中。这些行已经损坏。行末尾的 '\' 表示在该页面中引入了一个中断，并缩进以下行：因此：

> Let's pretend to have an extremely \
> long line that \
> does not fit
> This one is short

确实是：

> Let's pretend to have an extremely long line that does not fit
> This one is short

第 2 章 ADMIN REST API

Red Hat Single Sign-On 附带了功能齐全的 Admin REST API，具有管理控制台提供的所有功能。

若要调用 API，您需要获取具有适当权限的访问令牌。服务器管理指南中介绍了所需的权限。

您可以使用 Red Hat Single Sign-On 为应用程序启用身份验证来获取令牌，请参阅保护应用程序和服务指南。您还可以使用直接访问权限授权来获取访问令牌。

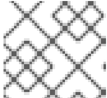
2.1. 使用 CURL 的示例

2.1.1. 使用用户名和密码进行身份验证

流程

1. 使用用户名 **admin 和密码**，为 realm **master** 中的用户获取访问令牌：

```
curl \
  -d "client_id=admin-cli" \
  -d "username=admin" \
  -d "password=password" \
  -d "grant_type=password" \
  "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```



注意

默认情况下，此令牌在 1 分钟后过期

结果将是一个 JSON 文档。

2. 通过提取 **access_token** 属性的值来调用所需的 API。

3. 通过在 API 请求的 **Authorization** 标头中包含值来调用 API。
以下示例演示了如何获取 master 域的详情：

```
curl \
  -H "Authorization: bearer eyJhbGciOiJSUz..." \
  "http://localhost:8080/auth/admin/realms/master"
```

2.1.2. 使用服务帐户进行身份验证

要使用 **client_id** 和 **client_secret** 对 Admin REST API 进行身份验证，请执行此流程。

流程

1. 确保配置了客户端，如下所示：

- **client_id** 是 属于域 master的机密客户端
- **client_id** 启用了 **Service Accounts Enabled** 选项
- **client_id** 有自定义"Audience"映射程序

  - 包含的客户端 Audience: **security-admin-console**

2. 检查 **client_id** 在 "Service Account Roles" 选项卡中分配了角色 'admin'。

```
curl \
  -d "client_id=<YOUR_CLIENT_ID>" \
  -d "client_secret=<YOUR_CLIENT_SECRET>" \
  -d "grant_type=client_credentials" \
  "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```

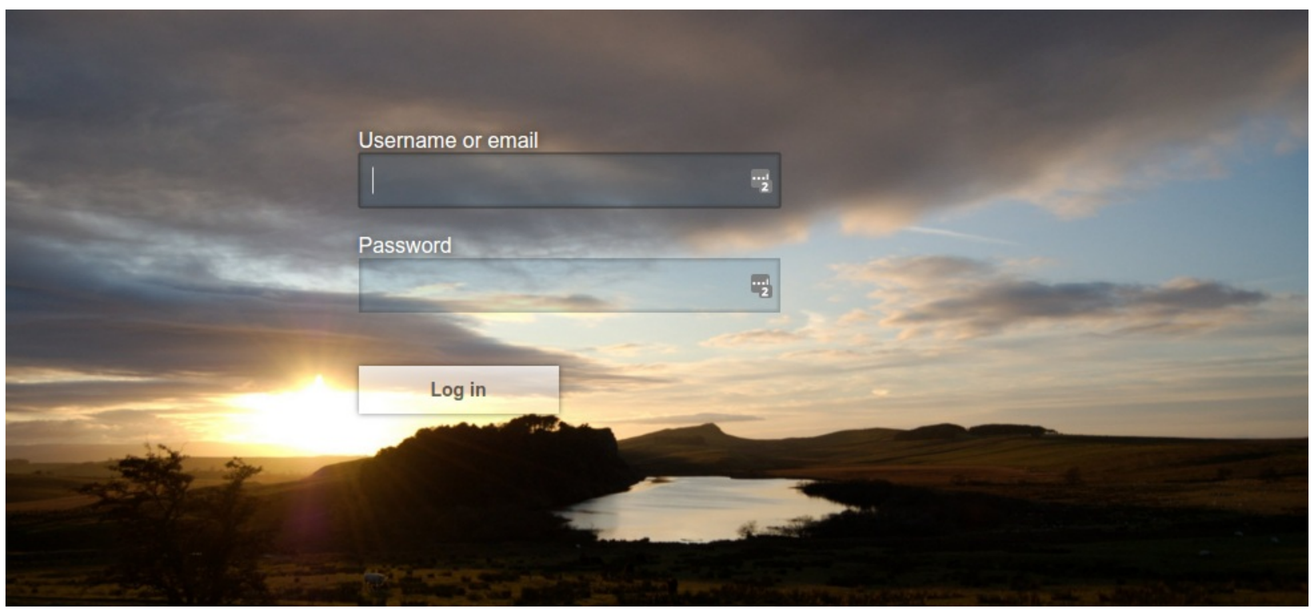
2.2. 其他资源

- 服务器管理指南
- 保护应用程序和服务指南
- API 文档

第 3 章 THEMES

Red Hat Single Sign-On 为网页和电子邮件提供支持。这允许自定义面向最终用户的外观和感觉，以便它们可以与您的应用程序集成。

图 3.1. 带有 sunrise 示例的登录页面



3.1. ME 类型

此主题可以提供一个或多个类型，以自定义 Red Hat Single Sign-On 的不同方面。可用的类型有：

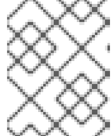
- 帐户 - 帐户管理
- Admin - Admin Console
- 电子邮件 - 电子邮件
- 登录 - 登录表单
- Welcome - 欢迎页面

3.2. 配置主题

除 welcome 外的所有主题类型都通过 Admin Console 配置。

流程

1. 登录 Admin 控制台。
2. 从左上角的下拉菜单中选择您的域。
3. 从菜单中点 **Realm Settings**。
4. 点 **Themes** 选项卡。



注意

要为 **master** Admin 控制台设置主题，您需要设置 **master** 域的 Admin Console 主题。

5. 要查看 Admin Console 刷新该页面的更改。
6. 通过编辑 **standalone.xml**、**standalone-ha.xml** 或 **domain.xml** 更改欢迎主题。
7. 将 **welcomeTheme** 添加到me 元素，例如：

```
<theme>
    ...
    <welcomeTheme>custom-theme</welcomeTheme>
    ...
</theme>
```

8. 重启服务器以使更改生效。

3.3. 默认主题

Red Hat Single Sign-On 在服务器的根目录中捆绑了默认 **主题**。要简化升级，不应直接编辑捆绑的它们。相反，请创建自己的主题来扩展其中一个捆绑主题。

3.4. 创建主题

主题包括：

- HTML 模板(自由标记模板)
- 镜像
- 消息捆绑包
- 风格表
- 脚本
- me 属性

除非计划替换每个页面，否则您应该扩展另一个主题。您最有可能想扩展 Red Hat Single Sign-On 主题，但如果您对页面的外观有很大更改，您还可能考虑对基础进行扩展。其基础主题主要由 HTML 模板和消息捆绑包组成，而 Red Hat Single Sign-On theme 主要包含镜像和样式表。

在扩展主题时，您可以覆盖单个资源（模板、风格表等）。如果您决定覆盖 HTML 模板，在升级到新版本时您可能需要更新自定义模板。

在创建主题时，最好禁用缓存，因为这样可以直接从主题目录中编辑主题资源，而无需重启 Red Hat Single Sign-On。

流程

1. 编辑 **standalone.xml**。

2. 对于 主题，将 **staticMaxAge** 设置为 **-1**，将 **cacheTemplates** 和 **cacheThemes** 设置为 **false** ：

```
<theme>
    <staticMaxAge>-1</staticMaxAge>
    <cacheThemes>false</cacheThemes>
    <cacheTemplates>false</cacheTemplates>
    ...
</theme>
```

3. 在 主题 目录中创建一个目录。

   目录的名称成为主题的名称。例如，要创建一个名为 **mytheme** 的主题，请创建目录 **themes/mytheme**。

4. 在me 目录中，为您要提供的每种类型创建一个目录。

   例如，要将登录类型添加到 **mytheme theme** 中，请创建目录 **themes/mytheme/login**。

5. 对于每个类型，创建一个文件 **theme.properties**，允许为主题设置一些配置。

   例如，要配置主题主题/mytheme/login 以扩展基础主题并导入一些常见资源，请创建文件 **themes/mytheme/login /theme.properties** 及以下内容：

```
parent=base
import=common/keycloak
```

   您现在已创建了支持登录类型的主题。

6. 登录到 Admin Console 以签出您的新主题

7. 选择您的域

8. 从菜单中点 **Realm Settings**。

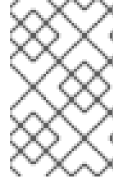
9. 点 **Themes** 选项卡。

10. 对于 **Login Theme**，选择 **mytheme** 并点 **Save**。

11. 打开域的登录页面。

    您可以通过应用程序登录或打开帐户管理控制台(/realms/{realm name}/account)来执行此操作。

12. 要查看更改父主题的效果，请在me .properties 中设置 parent=keycloak 并刷新登录页面。



注意

确保在生产环境中重新启用缓存，因为它会对性能有严重影响。

3.4.1. me 属性

me 属性在主题目录中的 < ;THEME TYPE>/theme.properties 中设置。

- 父项 - 要扩展的主题
- import - 从另一主题导入资源
- 样式 - 空格分隔的样式列表包括
- locales - 以逗号分隔的支持区域列表

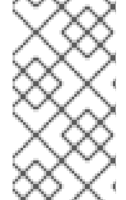
有一个属性列表可用于更改用于某些元素类型的 cs 类。如需这些属性的列表，请查看对应类型 keycloak theme 中的 theme.properties 文件(themes/keycloak/<THEME TYPE>/theme.properties)。

您还可以添加自己的自定义属性，并从自定义模板使用它们。

在这样做时，您可以使用以下格式替换系统属性或环境变量：

- ${some.system.property} - for system properties
- ${env.ENV_VAR} - 用于环境变量。

如果系统属性或环境变量没有使用 ${foo:defaultValue} 找到，也可以提供默认值。



注意

如果没有提供默认值，且没有对应的系统属性或环境变量，则不会替换任何内容，您会在模板中以 格式结束。

以下是可能的问题示例：

```
javaVersion=${java.version}

unixHome=${env.HOME:Unix home not found}
windowsHome=${env.HOMEPATH:Windows home not found}
```

3.4.2. 将风格表添加到主题

您可以在主题中添加一个或多个样式表。

流程

1. 在 < THEME TYPE>/resources/css 目录中创建一个文件。
2. 将此文件添加到me .properties 中的 styles 属性。

   例如，要将 风格.cs 添加到 mytheme 中，请使用以下内容创建 themes/mytheme/login/resources/css/styles.css ：

```
.login-pf body {
    background: DimGrey none;
}
```

3. 编辑 它们/mytheme/login/theme.properties 并添加：

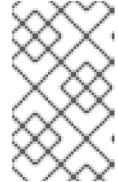
```
styles=css/styles.css
```

4. 要查看更改，请打开您的域的登录页面。

   您将注意到应用的唯一样式是来自您的自定义风格表。

5. 为了包括来自父主题的样式，请从该主题加载 样式。编辑 themes/mytheme/login/theme.properties 并更改 样式 以：

```
styles=web_modules/@fontawesome/fontawesome-free/css/icons/all.css
web_modules/@patternfly/react-core/dist/styles/base.css web_modules/@patternfly/react-
core/dist/styles/app.css node_modules/patternfly/dist/css/patternfly.min.css
node_modules/patternfly/dist/css/patternfly-additions.min.css css/login.css css/styles.css
```



注意

要从父风格表中覆盖样式，请确保您的风格表已列出最后。

3.4.3. 将脚本添加到主题

您可以将一个或多个脚本添加到主题。

流程

1. 在 < THEME TYPE>/resources/js 目录中创建一个文件。

2. 将该文件添加到me .properties 中的 scripts 属性。

   例如，要将 script.js 添加到 mytheme 中，请使用以下内容创建 themes/mytheme/login/resources/js/script.js ：

```
alert('Hello');
```

然后编辑它们 /mytheme/login/theme.properties 并添加：

```
scripts=js/script.js
```

3.4.4. 将镜像添加到主题

要使镜像提供给主题，请将它们添加到主题的 < ;THEME TYPE>/resources/img 目录中。它们可以从样式表内使用，或者直接在 HTML 模板中使用。

例如，要在 mytheme 中添加一个镜像，将镜像复制到 themes/mytheme/login/resources/img/image.jpg 中。

然后您可以使用以下自定义风格表：

```
body {
   background-image: url('../img/image.jpg');
   background-size: cover;
}
```

或者直接在 HTML 模板中使用以下内容，将以下内容添加到自定义 HTML 模板：

```
<img src="${url.resourcesPath}/img/image.jpg">
```

3.4.5. 消息

模板中的文本从消息捆绑包加载。扩展另一主题的主题将继承父消息捆绑包中的所有消息，您可以通过向您的主题添加 < THEME TYPE>/messages/messages_en.properties 来覆盖单个消息。

例如，在登录表单上，使用 Your Username for mytheme create file themes/mytheme/login/messages/messages_en.properties 替换为以下内容：

```
usernameOrEmail=Your Username
```

在使用消息值时，{0} 和 {1} 等消息值会被替换为参数。例如，{0} 中的 {0} 被替换为该域的名称。

这些消息捆绑包的文本可以被特定域的值覆盖。特定于 realm 的值可通过 UI 和 API 进行管理。

3.4.6. 在域中添加语言

先决条件

- 要为域启用国际化，请查看 服务器管理指南。

流程

1. 在主题的目录中创建文件 <THEME TYPE>/messages/messages_<LOCALE>.properties。

2. 将此文件添加到 < THEME TYPE>/theme.properties 中的 locales 属性。要使语言提供给用户 登录、帐户和 电子邮件，主题必须支持该语言，因此您需要为这些主题类型添加您的语言。

   例如：要将 Norwegian 转换添加到 mytheme theme create file themes/mytheme/login/messages/messages_no.properties 中以下内容：

   ```
   usernameOrEmail=Brukernavn
   password=Passord
   ```

   如果省略了消息的翻译，将使用英语。

3. 编辑 它们/mytheme/login/theme.properties 并添加：

   ```
   locales=en,no
   ```

4. 为 帐户 添加相同的 电子邮件 类型。要执行此操作，请创建它们 /mytheme/account/messages/messages_no.properties 和 themes/mytheme/email/messages_no.properties。这些文件为空将导致使用英语消息。

5. 将 themes/mytheme/login/theme.properties 复制到 themes/mytheme/account/theme.properties 以及

themes/mytheme/email/theme.properties。

6. 为语言选择器添加翻译。这是通过向英语翻译添加消息来完成的。要执行此操作，请将以下内容添加到 themes/mytheme/account/messages/messages_en.properties 和 themes/mytheme/login/messages/messages_en.properties:

```
locale_no=Norsk
```

默认情况下，消息属性文件应使用 ISO-8859-1 进行编码。也可以使用特殊标头指定编码。例如，使用 UTF-8 编码：

```
# encoding: UTF-8
usernameOrEmail=....
```

其他资源

- 如需有关如何选择当前区域的详细信息，请参阅 Locale Selector。

3.4.7. 添加自定义身份提供程序图标

Red Hat Single Sign-On 支持为自定义身份提供程序添加图标，该图标显示在登录屏幕中。

流程

1. 在 login theme.properties 文件中定义图标类（例如 themes/mytheme/login/theme.properties）使用键模式 kcLogoIdP-<alias& gt;。

2. 对于带有别名 myProvider 的身份提供程序，您可以在自定义主题的 me.properties 文件中添加一行。例如：

```
kcLogoIdP-myProvider = fa fa-lock
```

所有图标都位于 PatternFly4 的官方网站上。社交供应商的图标已在基本登录主题属性中定义（它们/密钥/登录/theme.properties），您可以激励自己。

3.4.8. 创建自定义 HTML 模板

Red Hat Single Sign-On 使用 Apache Freemarker 模板来生成 HTML。您可以通过创建 < THEME TYPE>/<TEMPLATE>.ftl 来覆盖您的主题中的单个模板。有关使用的模板列表，请查看 themes/base/<THEME TYPE>。

流程

1. 将模板从基础主题复制到您自己的主题。

2. 应用您需要的修改。

   例如，要为 mytheme theme 创建自定义登录表单，请将 themes/base/login/login.ftl 复制到他们 es/mytheme/login，并在编辑器中打开它。

   第一行(<#import …>)后，添加 <h1>HELLO WORLD!</h1& gt;，如下所示：

```
<#import "template.ftl" as layout>
<h1>HELLO WORLD!</h1>
...
```

3. 备份修改后的模板。当升级到新版本的 Red Hat Single Sign-On 时，您可能需要更新自定义模板，以对原始模板应用更改（如果适用）。

其他资源

- 有关如何编辑模板的详情，请参阅 FreeMarker Manual。

3.4.9. 电子邮件

要编辑电子邮件的主题和内容，如密码恢复电子邮件，请在主题 的电子邮件 类型中添加消息捆绑包。每个电子邮件都有三个信息。对于主题，一个用于纯文本正文，另一个用于 html 正文。

要查看所有可用电子邮件，请查看 主题/基础/电子邮件/messages/messages_en.properties。

例如，要更改 mytheme 的密码恢复电子邮件，请创建主题主题创建 主题/mytheme/email/messages/messages_en.properties 以及以下内容：

```
passwordResetSubject=My password recovery
passwordResetBody=Reset password link: {0}
passwordResetBodyHtml=<a href="{0}">Reset password</a>
```

3.5. 部署它们

通过复制主题目录或可作为存档部署，可将主题目录部署到 Red Hat Single Sign-On 中。在开发过程中，您可以将主题 复制到 主题 目录，但在生产环境中，您可能需要考虑使用 存档。存档 使主题的版本控制副本变得更加简单，尤其是当您拥有多个红帽单点登录（如群集）实例时。

流程

1. 要将主题部署为存档，请使用主题资源创建 JAR 存档。

2. 将文件 META-INF/keycloak-themes.json 添加到可列出存档中可用的文件以及每个主题提供的类型。

   例如，mytheme theme create mytheme.jar 中包含其内容：

   - META-INF/keycloak-themes.json
   - theme/mytheme/login/theme.properties
   - theme/mytheme/login/login.ftl
   - theme/mytheme/login/resources/css/styles.css
   - theme/mytheme/login/resources/img/image.png
   - theme/mytheme/login/messages/messages_en.properties
   - theme/mytheme/email/messages/messages_en.properties

本例中的 META-INF/keycloak-themes.json 的内容是：

```
{
  "themes": [{
    "name" : "mytheme",
    "types": [ "login", "email" ]
  }]
}
```

个存档可以包含多个主题，每个主题都可以支持一个或多个类型。

要将存档部署到 Red Hat Single Sign-On，请将其添加到 Red Hat Single Sign-On 的 standalone/deployments/ 目录中，它会被自动加载。

3.6. ME 选择器

默认情况下，使用为 realm 配置的主题，但客户端无法覆盖登录主题。可以通过 Theme Selector SPI 更改此行为。

例如，可以通过查看用户代理标头来为桌面和移动设备选择不同的主题。

要创建自定义me 选择器，您需要实施 ThemeSelectorProviderFactory 和 ThemeSelectorProvider。

3.7. ME 资源

当在 Red Hat Single Sign-On 中实施自定义提供程序时，通常需要添加额外的模板、资源和消息捆绑包。

加载额外主题资源的最简单方法是在me -resources/resources 和 messages 捆绑包中使用模板在 me-resources/ templates 资源 中创建 JAR。

如果您希望更灵活地加载可通过 ThemeResourceSPI 实现的模板和资源。通过实施 主题 ResourceProviderFactory 和 ThemeResourceProvider，您可以精确决定如何加载模板和资源。

3.8. 区域设置选择器

默认情况下，使用实现 LocaleSelectorProvider 接口的 DefaultLocaleSelectorProvider 选择区域设置。禁用国际化时，英语是默认语言。启用国际化后，区域设置会根据 服务器管理指南 中描述的逻辑解析。

此行为可通过 LocaleSelectorSPI 通过实现 LocaleSelectorProvider 和 LocaleSelectorProvider Factory 来更改。

LocaleSelectorProvider 接口具有一个方法，它 可以解析 Locale，它必须返回一个为 RealmModel 和 nullable UserModel 的区域设置。实际请求可从 KeycloakSession#getContext 方法中找到。

自定义实现可以扩展 DefaultLocaleSelectorProvider，以便重复使用默认行为的部分。例如，要忽略 Accept-Language 请求标头，自定义实现可能会扩展默认供应商，覆盖其 getAcceptLanguageHeaderLocale，并返回 null 值。因此，区域选择将回退到域的默认语言。

3.9. 其他资源

- 有关创建和部署自定义提供程序的详情，请参阅 服务提供商接口。

第 4 章 自定义用户属性

您可以使用自定义主题在注册页面和帐户管理控制台中添加自定义用户属性。

4.1. 注册页面

在注册页面中，使用此流程输入自定义属性。

流程

1. 将模板 themes/base/login/register.ftl 复制到您的自定义主题的登录类型。

2. 在编辑器中打开副本。

   例如，要在注册页面中添加移动号，请在表单中添加以下片断：

```
<div class="form-group">
  <div class="${properties.kcLabelWrapperClass!}">
      <label for="user.attributes.mobile" class="${properties.kcLabelClass!}">Mobile
number</label>
  </div>

  <div class="${properties.kcInputWrapperClass!}">
      <input type="text" class="${properties.kcInputClass!}" id="user.attributes.mobile"
name="user.attributes.mobile" value="${(register.formData['user.attributes.mobile']!'')}"/>
  </div>
</div>
```

3. 确保输入 html 元素的名称以 user.attributes 开头。在上面的示例中，属性将由名为 mobile 的 Red Hat Single Sign-On 存储。

4. 要查看更改，请确保您的域正在使用您的登录主题主题并打开注册页面。

4.2. 帐户管理控制台

在帐户管理控制台中，使用此流程管理用户配置集页面中的自定义属性。

流程

1. **将模板 themes/base/account/account.ftl 复制到自定义主题的帐户类型。**

2. **在编辑器中打开副本。**

   作为示例，在帐户页面中添加一个移动号，将以下代码片段添加到表单中：

```
<div class="form-group">
  <div class="col-sm-2 col-md-2">
     <label for="user.attributes.mobile" class="control-label">Mobile number</label>
  </div>

  <div class="col-sm-10 col-md-10">
     <input type="text" class="form-control" id="user.attributes.mobile"
name="user.attributes.mobile" value="${(account.attributes.mobile!'')}"/>
  </div>
</div>
```

3. **确保输入 html 元素的名称以 user.attributes 开头。**

4. **要查看更改，请确保您的域正在使用帐户的主题，并在帐户管理控制台中打开用户配置文件页面。**

4.3. 其他资源

- **有关如何创建自定义 主题，请参阅主题。**

第 5 章 IDENTITY BROKERING API

红帽单点登录可将身份验证委派给用于登录的父 IDP。典型的例子是您希望用户能够通过社交供应商（如 Facebook 或 Google）登录。您还可以将现有帐户链接到代理的 IDP。本节介绍了应用程序可使用的一些 API，因为它与身份代理相关。

5.1. 检索外部 IDP 令牌

红帽单点登录允许您使用外部 IDP 存储来自身份验证流程的令牌和响应。为此，您可以使用 IDP 设置页面中的 Store Token 配置选项。

应用程序代码可以检索这些令牌和响应，以拉取额外的用户信息，或者安全地调用外部 IDP 上的请求。例如，应用程序可能希望使用 Google 令牌在其他 Google 服务和 REST API 上调用。要检索特定身份提供程序的令牌，您需要按照以下方式发送请求：

```
GET /auth/realms/{realm}/broker/{provider_alias}/token HTTP/1.1
Host: localhost:8080
Authorization: Bearer <KEYCLOAK ACCESS TOKEN>
```

应用程序必须通过 Red Hat Single Sign-On 进行身份验证，并获得了访问令牌。此访问令牌需要设置 broker 客户端级角色 read-token。这意味着用户必须具有此角色的角色映射，客户端应用必须在其范围内拥有该角色。在这种情况下，假设您在 Red Hat Single Sign-On 中访问受保护的服务，您需要在用户身份验证期间发送由 Red Hat Single Sign-On 发布的访问令牌。在代理配置页面中，您可以通过打开 Stored Tokens Readable 交换机，自动将此角色分配给新导入的用户。

这些外部令牌可以通过提供程序再次登录，或使用发起的帐户链接 API 来重新建立这些外部令牌。

5.2. 客户端发起的帐户链接

有些应用程序希望与 Facebook 等社交供应商集成，但不希望提供通过这些社交供应商登录的选项。Red Hat Single Sign-On 提供基于浏览器的 API，应用程序可以使用该 API 将现有用户帐户链接到特定的外部 IDP。这称为客户端发起的帐户链接。连接的帐户只能由 OIDC 应用程序启动。

工作方式是，应用程序将用户浏览器转发到 Red Hat Single Sign-On 服务器上的 URL，请求该用户的帐户链接到特定的外部供应商（即 Facebook）。服务器发起使用外部提供程序的登录。浏览器登录外部提供程序，并重新重定向到服务器。服务器建立链接，并通过确认重新重定向到应用程序。

在客户端应用程序启动这个协议前，客户端应用程序必须满足一些前提条件：

- 必须为 admin 控制台中的用户域配置并启用所需的身份提供程序。
- 该用户帐户必须已经通过 OIDC 协议以现有用户身份登录
- 用户必须具有 account.manage-account 或 account.manage-account-links 角色映射。
- 应用必须被授予在其访问令牌内这些角色的范围
- 应用必须有权访问其访问令牌，因为它需要其中的信息来生成重定向 URL。

若要启动登录，应用必须构造 URL，并将用户浏览器重定向到此 URL。URL 如下所示：

```
/{auth-server-root}/auth/realms/{realm}/broker/{provider}/link?client_id={id}&redirect_uri={uri}&nonce=
{nonce}&hash={hash}
```

下面是每个路径和查询参数的描述：

provider

这是您在管理控制台的 Identity Provider 部分中定义的外部 IDP 的供应商别名。

client_id

这是应用程序的 OIDC 客户端 id。在管理控制台中将应用程序注册为客户端时，您必须指定这个客户端 ID。

redirect_uri

这是您在建立帐户链接后要重定向到的应用程序回调 URL。它必须是有效的客户端重定向 URI 模式。换句话说，它必须与您在 admin 控制台中注册客户端时定义的有效 URL 模式之一匹配。

nonce

这是应用程序必须生成的随机字符串

hash

这是一个 Base64 URL 编码哈希。此哈希通过 Base64 URL 编码 非 + token.getSessionState（） + token.getIssuedFor （） + provider 的 SHA_256 哈希。令牌变量从 OIDC 访问令牌获取。

基本上，您要访问的随机非数、用户会话 ID、客户端 ID 和身份提供程序别名。

以下是生成 URL 以便建立帐户链接的 Java Servlet 代码示例。

```
   KeycloakSecurityContext session = (KeycloakSecurityContext)
httpServletRequest.getAttribute(KeycloakSecurityContext.class.getName());
   AccessToken token = session.getToken();
   String clientId = token.getIssuedFor();
   String nonce = UUID.randomUUID().toString();
   MessageDigest md = null;
   try {
     md = MessageDigest.getInstance("SHA-256");
   } catch (NoSuchAlgorithmException e) {
     throw new RuntimeException(e);
   }
   String input = nonce + token.getSessionState() + clientId + provider;
   byte[] check = md.digest(input.getBytes(StandardCharsets.UTF_8));
   String hash = Base64Url.encode(check);
   request.getSession().setAttribute("hash", hash);
   String redirectUri = ...;
   String accountLinkUrl = KeycloakUriBuilder.fromUri(authServerRootUrl)
                   .path("/auth/realms/{realm}/broker/{provider}/link")
                   .queryParam("nonce", nonce)
                   .queryParam("hash", hash)
                   .queryParam("client_id", clientId)
                   .queryParam("redirect_uri", redirectUri).build(realm, provider).toString();
```

为什么包含此哈希？我们这样做，使得身份验证服务器可以保证知道客户端应用程序启动请求，而没有随机要求将用户帐户链接到特定提供程序的其他恶意应用程序。身份验证服务器将首先检查用户是否通过检查登录时设置了 SSO cookie 来登录。然后，它将尝试基于当前的登录重新生成哈希，并将其与应用程序发送的哈希匹配。

帐户链接后，auth 服务器将重新重定向到 redirect_uri。如果提供链接请求时遇到问题，则 auth 服务器可能会重新重定向到 redirect_uri。浏览器可能只是在错误页面中结束，而不重定向到应用程序。如果存在错误条件，并且 auth 服务器足以重新重定向到客户端应用，则会将额外的 错误查询参数 附加到 redirect_uri 中。

警告

虽然此 API 可确保应用程序启动请求，但它不会完全阻止 CSRF 对这个操作进行攻击。应用程序仍负责保护 CSRF 攻击目标。

5.2.1. 刷新外部令牌

如果您使用由登录到供应商（例如 Facebook 或 GitHub 令牌）生成的外部令牌，您可以通过重新初始化帐户链接 API 来刷新此令牌。

第 6 章 服务提供商接口(SPI)

Red Hat Single Sign-On 是在不需要自定义代码的情况下覆盖大多数用例，但我们也希望可以自定义。要达到此目的，Red Hat Single Sign-On 有许多可实施您自己的提供程序的服务提供商接口(SPI)。

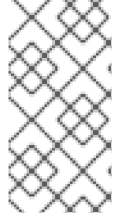
6.1. 实施 SPI

要实施 SPI，您需要实施其提供商工厂和提供程序接口。您还需要创建服务配置文件。

例如，要实施 Theme Selector SPI，您需要实施 ThemeSelectorProviderFactory 和 ThemeSelectorProvidery，并提供文件 META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory.

ThemeSelectorProviderFactory 示例：

```
package org.acme.provider;

import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory {

    @Override
    public ThemeSelectorProvider create(KeycloakSession session) {
        return new MyThemeSelectorProvider(session);
    }

    @Override
    public void init(Config.Scope config) {
    }

    @Override
    public void postInit(KeycloakSessionFactory factory) {
    }

    @Override
    public void close() {
    }

    @Override
    public String getId() {
        return "myThemeSelector";
    }
}
```



注意

Red Hat Single Sign-On 创建一个单一供应商工厂实例，以便存储多个请求的状态。提供商实例通过在每个请求的工厂上调用 create 来创建，因此它们应该是轻量级的对象。

ThemeSelectorProvider 示例：

```
package org.acme.provider;

import ...

public class MyThemeSelectorProvider implements ThemeSelectorProvider {

    public MyThemeSelectorProvider(KeycloakSession session) {
    }


    @Override
    public String getThemeName(Theme.Type type) {
        return "my-theme";
    }

    @Override
  public void close() {
    }
}
```

服务配置文件示例(META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory):

```
org.acme.provider.MyThemeSelectorProviderFactory
```

您可以通过 standalone.xml、standalone-ha.xml 或 domain.xml 配置您的供应商。

例如，将以下内容添加到 standalone.xml 中：

```
<spi name="themeSelector">
    <provider name="myThemeSelector" enabled="true">
        <properties>
            <property name="theme" value="my-theme"/>
        </properties>
    </provider>
</spi>
```

然后您可以在 ProviderFactory init 方法中检索配置：

```
public void init(Config.Scope config) {
    String themeName = config.get("theme");
}
```

如果需要，您的供应商也可以查找其他提供程序。例如：

```
public class MyThemeSelectorProvider implements ThemeSelectorProvider {

    private KeycloakSession session;

    public MyThemeSelectorProvider(KeycloakSession session) {
        this.session = session;
    }

    @Override
    public String getThemeName(Theme.Type type) {
        return session.getContext().getRealm().getLoginTheme();
    }
}
```

6.1.1. 在管理控制台中显示您的 SPI 实施的信息

有时，对于向 Red Hat Single Sign-On 管理员显示您的提供程序的其他信息会很有用。您可以显示供应商构建时间信息（例如，当前安装的自定义提供程序版本）、提供商的当前配置（例如，您的供应商与供应商的 url）或一些操作信息（从远程系统通信到的发送时间）。Red Hat Single Sign-On Admin Console 提供 Server Info 页面来显示此类信息。

要显示供应商中的信息，足以在您的 ProviderFactory 中实现 org.keycloak.provider.ServerInfoAware ProviderFactory 接口。

上例中的 MyThemeSelectorProviderFactory 的实现示例：

```
package org.acme.provider;

import ...

public class MyThemeSelectorProviderFactory implements ThemeSelectorProviderFactory,
ServerInfoAwareProviderFactory {
    ...

    @Override
    public Map<String, String> getOperationalInfo() {
        Map<String, String> ret = new LinkedHashMap<>();
```

```
        ret.put("theme-name", "my-theme");
        return ret;
    }
}
```

6.2. 使用可用供应商

在提供程序实施中，您可以使用 Red Hat Single Sign-On 中提供的其他提供程序。现有提供程序通常使用 KeycloakSession 来检索，该提供程序可供您的供应商使用，如 实施 SPI 一节中所述。

Red Hat Single Sign-On 有两个供应商类型：

- 单 实施供应商类型 - 红帽单点登录 运行时只能实施特定提供程序类型的单一主动实施。

  例如 HostnameProvider 指定 Red Hat Single Sign-On 使用的主机名，并可为整个 Red Hat Single Sign-On 服务器共享该主机名。因此，这个提供程序对于 Red Hat Single Sign-On 服务器只能有一个有效实施。如果有多个提供程序实施可供服务器运行时使用，则需要将其中一个提供程序实施指定为默认的实施。

例如：

```
<spi name="hostname">
    <default-provider>default</default-provider>
    ...
</spi>
```

默认值用作 default -provider 的值，必须与特定 provider factory.getId （）的 ProviderFactory.getId （） 返回的 ID 匹配。在代码中，您可以获取提供程序，如 keycloakSession.getProvider (HostnameProvider.class)

- 多种实施供应商类型 - Those 是供应商类型，允许多个实施可用并在 Red Hat Single Sign-On 运行时中协同工作。

  例如，EventListener 供应商允许有多个可用的实施并注册，这意味着特定的事件可以发送到所有监听程序(jboss-logging、sysout 等)。在代码中，您可以获取提供程序的指定实例，如 session.getProvider (EventListener.class, "jboss-logging")。您需要指定提供程序的 provider_id 作为第二个参数，因为这里可能会有多个此提供程序类型的实例，如下所述。

  提供程序 ID 必须与特定供应商工厂实施的 ProviderFactory.getId （） 返回的 ID。某些提供程序类型可以通过使用 ComponentModel 作为第二个参数来检索，一些（如 Authenticator）

甚至需要通过 KeycloakSessionFactory 的使用来检索。不建议以这种方式实施自己的供应商，因为将来可能会被弃用。

6.3. 注册供应商实现

注册供应商实施的方法有两种。在大多数情况下，最简单的方法是使用 Red Hat Single Sign-On 部署器方法，因为这可为您处理多个依赖项。它还支持热部署和重新部署。

另一种方法是将 作为模块部署。

如果您要创建自定义 SPI，则需要将其部署为模块，否则我们推荐使用 Red Hat Single Sign-On deployer 方法。

6.3.1. 使用 Red Hat Single Sign-On deployer

如果您将供应商 jar 复制到 Red Hat Single Sign-On standalone/deployments/ 目录中，则您的供应商将自动部署。热部署也可以正常工作。此外，您的供应商 jar 与在 JBoss EAP 环境中部署的其他组件类似，它们可以使用 jboss-deployment-structure.xml 文件等功能。您可以通过此文件来设置对其他组件的依赖项，并加载第三方 jars 和模块。

供应商 jar 也可以包含在其他可部署单元中，如 EAR 和 WAR。使用 EAR 进行部署实际上使得使用第三方 jar 非常容易，因为您可以将这些库放在 EAR 的 lib/ 目录中。

6.3.2. 使用模块注册供应商

流程

1. 使用 jboss-cli 脚本或手动创建文件夹创建一个模块。

   a. 例如，要使用 jboss-cli 脚本添加事件监听程序 sysout 示例供应商，请执行：

   ```
   KEYCLOAK_HOME/bin/jboss-cli.sh --command="module add --name=org.acme.provider
   --resources=target/provider.jar --dependencies=org.keycloak.keycloak-
   core,org.keycloak.keycloak-server-spi"
   ```

   b. 或者，您可以在 KEYCLOAK_HOME/modules 中手动创建模块，并添加您的 jar 和 module.xml。

例如，创建 **KEYCLOAK_HOME/modules/org/acme/provider/main** 文件夹。然后，将 **provider.jar** 复制到此文件夹，并使用以下内容创建 **module.xml** ：

```
<?xml version="1.0" encoding="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" name="org.acme.provider">
    <resources>
        <resource-root path="provider.jar"/>
    </resources>
    <dependencies>
        <module name="org.keycloak.keycloak-core"/>
        <module name="org.keycloak.keycloak-server-spi"/>
    </dependencies>
</module>
```

2. 使用 Red Hat Single Sign-On 注册此模块，方法是编辑 **standalone.xml**、**standalone-ha.xml** 或 **domain.xml** 的 **keycloak-server** 子系统部分，并将它添加到提供程序：

```
<subsystem xmlns="urn:jboss:domain:keycloak-server:1.1">
    <web-context>auth</web-context>
    <providers>
        <provider>module:org.keycloak.examples.event-sysout</provider>
    </providers>
    ...
```

6.3.3. 禁用供应商

您可以通过在 **standalone.xml**、**standalone-ha.xml** 或 **domain.xml** 中将 **provider** 的 **enabled** 属性设置为 **false** 来禁用供应商。例如，禁用 Infinispan 用户缓存提供程序添加：

```
<spi name="userCache">
    <provider name="infinispan" enabled="false"/>
</spi>
```

6.4. 使用 JAKARTA EE

服务供应商只要您正确设置了 **META-INF/services** 文件以指向您的提供商，即可将其打包在任意 Jakarta EE 组件中。例如，如果您的供应商需要使用第三方库，可以在 ear 中打包您的供应商，并将这些第三方库存储在 ear 的 **lib/** 目录中。另请注意，提供程序 jars 可以使用 **jboss-deployment-structure.xml** 文件，该文件 EJB、WARS 和 EARs 可在 JBoss EAP 环境中使用。有关此文件的详情，请参阅 JBoss EAP 文档。它允许您在其他精细操作中拉取外部依赖项。

提供商工厂 实施需要是普通 java 对象。但是，我们目前还支持将提供程序类实施为有状态 EJB。这是实现它的方式：

```
@Stateful
@Local(EjbExampleUserStorageProvider.class)
public class EjbExampleUserStorageProvider implements UserStorageProvider,
        UserLookupProvider,
        UserRegistrationProvider,
        UserQueryProvider,
        CredentialInputUpdater,
        CredentialInputValidator,
        OnUserCache
{
    @PersistenceContext
    protected EntityManager em;

    protected ComponentModel model;
    protected KeycloakSession session;

    public void setModel(ComponentModel model) {
        this.model = model;
    }

    public void setSession(KeycloakSession session) {
        this.session = session;
    }


    @Remove
    @Override
    public void close() {
    }
...
}
```

您可以定义 @Local 注释并指定您的提供程序类。如果没有这样做，EJB 将不会正确代理供应商实例，您的供应商将无法正常工作。

您可以在供应商的 close （） 方法上放置 @Remove 注释。如果没有，则有状态的 bean 不会被清理，您最终可能会看到错误消息。

提供商工厂 的 Ixmplementations 需要是普通 java 对象。您工厂类在其 create （） 方法中对 Stateful EJB 执行 JNDI 查找。

```
public class EjbExampleUserStorageProviderFactory
        implements UserStorageProviderFactory<EjbExampleUserStorageProvider> {

    @Override
    public EjbExampleUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        try {
            InitialContext ctx = new InitialContext();
            EjbExampleUserStorageProvider provider =
```

```
(EjbExampleUserStorageProvider)ctx.lookup(
                "java:global/user-storage-jpa-example/" +
EjbExampleUserStorageProvider.class.getSimpleName());
         provider.setModel(model);
         provider.setSession(session);
         return provider;
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
   }
```

6.5. JAVASCRIPT 供应商

Red Hat Single Sign-On 能够在运行时执行脚本的功能，以便管理员能够自定义特定的功能：

- authenticator
- JavaScript Policy
- OpenID Connect 协议映射程序
- SAML 协议映射程序

6.5.1. authenticator

身份验证脚本必须至少提供以下功能之一：身份验证 (..) 来自 Authenticator#authenticate (AuthenticationFlowContext) action (..)，它从 Authenticator#action (AuthenticationFlowContext)操作中调用。

自定义 授权器 至少应提供 authenticate (..) 函数。您可以使用代码中的 javax.script.Bindings 脚本。

script

用于访问脚本元数据的 ScriptModel

realm

RealmModel

user

当前 UserModel

会话

active KeycloakSession

authenticationSession

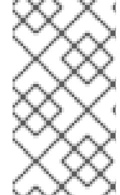
当前 AuthenticationSessionModel

httpRequest

当前 org.jboss.resteasy.spi.HttpRequest

LOG

org.jboss.logging.Logger 限定于 基于脚本的Authenticator

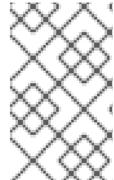


注意

您可以从传递给 验证(context) 操作(context) 功能的上下文参数中提取其他上下文信息。

```
AuthenticationFlowError = Java.type("org.keycloak.authentication.AuthenticationFlowError");

function authenticate(context) {

  LOG.info(script.name + " --> trace auth for: " + user.username);

  if (   user.username === "tester"
      && user.getAttribute("someAttribute")
      && user.getAttribute("someAttribute").contains("someValue")) {

      context.failure(AuthenticationFlowError.INVALID_USER);
      return;
  }

  context.success();
}
```

6.5.2. 使用脚本创建 JAR 以进行部署



注意

JAR 文件是具有 .jar 扩展名的常规 ZIP 文件。

为了使您的脚本可用于 Red Hat Single Sign-On，您需要将其部署到服务器。因此，您应该创建带有以下结构的 JAR 文件：

```
META-INF/keycloak-scripts.json

my-script-authenticator.js
my-script-policy.js
my-script-mapper.js
```

META-INF/keycloak-scripts.json 是一个文件描述符，提供关于您要部署的脚本的元数据信息。这是一个带有以下结构的 JSON 文件：

```
{
  "authenticators": [
    {
      "name": "My Authenticator",
      "fileName": "my-script-authenticator.js",
      "description": "My Authenticator from a JS file"
    }
  ],
  "policies": [
    {
      "name": "My Policy",
      "fileName": "my-script-policy.js",
      "description": "My Policy from a JS file"
    }
  ],
  "mappers": [
    {
      "name": "My Mapper",
      "fileName": "my-script-mapper.js",
      "description": "My Mapper from a JS file"
    }
  ],
  "saml-mappers": [
    {
      "name": "My Mapper",
      "fileName": "my-script-mapper.js",
      "description": "My Mapper from a JS file"
    }
  ]
}
```

此文件应引用您要部署的不同脚本供应商：

- **authenticators**

  对于 OpenID Connect 脚本编写器。您可以在同一 JAR 文件中有一个或多个验证器

- **policies**

  对于使用 Red Hat Single Sign-On Authorization 服务时的 JavaScript 策略。您可以在同一 JAR 文件中拥有一个或多个策略

- **映射程序**

  对于 OpenID Connect 脚本映射程序。您可以在同一 JAR 文件中有一个或多个映射程序

- **saml-mappers**

  对于 SAML 脚本协议映射程序。您可以在同一 JAR 文件中有一个或多个映射程序

对于 JAR 文件中的每个脚本文件，您需要在 META-INF/keycloak-scripts.json 中对应条目，将您的脚本文件映射到特定的提供程序类型。为此，您应该为每个条目提供以下属性：

- **name**

  用于通过 Red Hat Single Sign-On 管理控制台显示脚本的友好名称。如果没有提供，则会改为使用脚本文件的名称

- **description**

  最好描述脚本文件的可选文本

- **fileName**

脚本文件的名称。这个属性 是必需的，应映射到 JAR 中的文件。

6.5.3. 部署脚本 JAR

拥有带有描述符和要部署的脚本的 JAR 文件后，您只需将 JAR 复制到 Red Hat Single Sign-On standalone/deployments/ 目录。

6.5.3.1. 在 Java 15 及更高版本上部署脚本引擎

为了运行脚本，Java 应用中需要使用 JavaScript 引擎。Java 14 及较低版本包括 Nashorn JavaScript Engine。它作为 Java 本身和 JavaScript 提供程序的一部分自动提供，默认可以使用此脚本引擎。但是，对于 Java 15 或更高版本，脚本引擎不是 Java 本身的一部分。需要将其添加至您的服务器，因为 Red Hat Single Sign-On 默认没有任何脚本引擎。部署脚本提供者时，Java 15 及更高版本需要额外步骤 - 将您选择的脚本引擎添加到您的发行版中。

您可以使用任何脚本引擎。但是，我们仅使用 Nashorn JavaScript Engine 进行测试。以下步骤假设使用了这个引擎：

您可以通过在 Red Hat Single Sign-On 中添加新模块 nashorn-core 来安装脚本引擎。服务器启动后，您可以在 KEYCLOAK_HOME/bin 目录中运行类似如下的命令：

```
export NASHORN_VERSION=15.3
wget https://repo1.maven.org/maven2/org/openjdk/nashorn/nashorn-core/$NASHORN_VERSION/nashorn-core-$NASHORN_VERSION.jar
./jboss-cli.sh -c --command="module add --module-root-dir=../modules/system/layers/keycloak/ --name=org.openjdk.nashorn.nashorn-core --resources=./nashorn-core-$NASHORN_VERSION.jar --dependencies=asm.asm,jdk.dynalink"
rm nashorn-core-$NASHORN_VERSION.jar
```

如果要将供应商安装到不同的模块中，您可以使用默认脚本供应商配置属性 script-engine-module。例如，您可以使用 KEYCLOAK_HOME/standalone/configuration/standalone-*.xml 文件等内容：

```xml
<spi name="scripting">
   <provider name="default" enabled="true">
      <properties>
         <property name="script-engine-module" value="org.graalvm.js.js-scriptengine"/>
      </properties>
   </provider>
</spi>
```

6.6. 可用的 SPI

如果要在运行时查看所有可用的 SPI 列表，您可以在管理门户中检查 Server Info 页面，如 Admin Console 部分所述。

第 7 章 USER STORAGE SPI

您可以使用 User Storage SPI 编写 Red Hat Single Sign-On 的扩展来连接外部用户数据库和凭证存储。内置的 LDAP 和 ActiveDirectory 支持是此操作中此 SPI 的实现。开箱即用，Red Hat Single Sign-On 使用其本地数据库来创建、更新和查找用户并验证凭证。然而，组织拥有现有的外部专有用户数据库，他们无法迁移到红帽单点登录的数据模型。对于这样的情况，应用程序开发人员可编写用户存储 SPI 的实施，以桥接 Red Hat Single Sign-On 用来登录用户和管理它们的内部用户对象模型。

当红帽单点登录运行时需要查找用户，如登录用户时，它会执行几个步骤来定位用户。首先，它会查看用户是否在用户缓存中；如果用户发现它使用内存中表示。然后，它将在 Red Hat Single Sign-On 本地数据库中查找该用户。如果没有找到用户，则通过 User Storage SPI 供应商实施循环来执行用户查询，直到其中之一返回用户查找运行时。提供程序查询用户的外部用户存储，并将用户的外部数据表示映射到 Red Hat Single Sign-On 的用户 metamodel。

用户存储 SPI 供应商实施还可以执行复杂的条件查询，对用户执行 CRUD 操作，验证和管理凭证，或者一次对许多用户执行批量更新。它取决于外部存储的功能。

用户存储 SPI 提供程序实施与 （通常）Jkarta EE 组件相似打包并部署。默认情况下，它们不会被启用，但必须在管理控制台的 User Federation 选项卡下启用和配置。

警告

如果您的用户供应商实施正在使用某些用户属性作为元数据属性来链接/提交用户身份，请确保用户无法编辑属性，并且对应的属性是只读的。示例是 LDAP_ID 属性，内置 Red Hat Single Sign-On LDAP 供应商用于存储 LDAP 服务器端用户的 ID。请参阅 Threat 模型缓解章节中 的详细信息。

7.1. 供应商接口

在构建 User Storage SPI 实施时，您必须定义供应商类和供应商工厂。提供商类实例按照供应商工厂为每个事务创建。提供程序类执行用户查询和其他用户操作的繁重处理。它们必须实施 org.keycloak.storage.UserStorageProvider 接口。

```
package org.keycloak.storage;

public interface UserStorageProvider extends Provider {
```

```
    /**
     * Callback when a realm is removed.  Implement this if, for example, you want to do some
     * cleanup in your user storage when a realm is removed
     *
     * @param realm
     */
    default
    void preRemove(RealmModel realm) {

    }

    /**
     * Callback when a group is removed.  Allows you to do things like remove a user
     * group mapping in your external store if appropriate
     *
     * @param realm
     * @param group
     */
    default
    void preRemove(RealmModel realm, GroupModel group) {

    }

    /**
     * Callback when a role is removed.  Allows you to do things like remove a user
     * role mapping in your external store if appropriate

     * @param realm
     * @param role
     */
    default
    void preRemove(RealmModel realm, RoleModel role) {

    }

}
```

您可能会考虑 UserStorageProvider 接口不是稀疏的？本章稍后将看到，您的供应商课程可能会实施的其他混合接口来支持用户集成。

UserStorageProvider 实例为每个事务创建一次。事务完成后，调用 UserStorageProvider.close（） 方法，然后实例便被垃圾回收。实例由供应商工厂创建。提供程序工厂实施 org.keycloak.storage.UserStorageProviderFactory 接口。

```
package org.keycloak.storage;

/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public interface UserStorageProviderFactory<T extends UserStorageProvider> extends
ComponentFactory<T, UserStorageProvider> {
```

```
    /**
     * This is the name of the provider and will be shown in the admin console as an option.
     *
     * @return
     */
    @Override
    String getId();

    /**
     * called per Keycloak transaction.
     *
     * @param session
     * @param model
     * @return
     */
    T create(KeycloakSession session, ComponentModel model);
...
}
```

在实施 UserStorageProviderFactory 时，提供者类必须将 concrete provider 类指定为模板参数。这是运行时必须内省此类，以扫描其功能（其实施的其他接口）。例如，如果您的供应商类命名为 FileProvider，则工厂类应如下所示：

```
public class FileProviderFactory implements UserStorageProviderFactory<FileProvider> {

    public String getId() { return "file-provider"; }

    public FileProvider create(KeycloakSession session, ComponentModel model) {
       ...
    }
```

getId （）方法返回 User Storage 供应商的名称。当您想要为特定域启用提供程序时，admin 控制台的 User Federation 页面中会显示此 ID。

create （）方法负责分配提供程序类的实例。它取 org.keycloak.models.KeycloakSession 参数。此对象可用于查找其他信息和元数据，以及提供对运行时内各种其他组件的访问。ComponentModel 参数代表供应商如何在特定域中启用和配置。它包含启用的供应商的实例 ID，以及在通过 admin 控制台启用时为它指定的任何配置。

UserStorageProviderFactory 还具有其他功能，本章稍后会用到它。

7.2. 供应商功能接口

如果您仔细检查 UserStorageProvider 接口，可能会发现它不会为查找或管理用户定义任何方法。这些方法实际在其他功能 *接口* 中定义，具体取决于外部用户存储能够提供并执行的功能范围。例如，一些

外部存储是只读的，只能进行简单的查询和凭证验证。您只需为您能够的功能实施 *功能接口*。您可以实施这些接口：

SPI	描述
org.keycloak.storage.user.UserLookupProvider	如果要能够使用此外部存储的用户登录，则需要这个接口。大多数（全部）提供商实施此界面。
org.keycloak.storage.user.UserQueryProvider	定义用于查找一个或多个用户的复杂查询。如果要从管理控制台查看和管理用户，则必须实施此接口。
org.keycloak.storage.user.UserRegistrationProvider	如果您的供应商支持添加和删除用户，则实施此接口。
org.keycloak.storage.user.UserBulkUpdateProvider	如果您的供应商支持一组用户的批量更新，实施此界面。
org.keycloak.credential.CredentialInputValidator	如果您的供应商可以验证一个或多个不同的凭证类型（例如，如果您的供应商可以验证密码），实施此界面。
org.keycloak.credential.CredentialInputUpdater	如果您的供应商支持更新一个或多个不同的凭证类型，则实施此接口。

7.3. 型号接口

能力 接口 中定义的大多数方法都返回或传递用户表示。这些表示由 org.keycloak.models.UserModel 接口定义。应用开发人员必须实现此界面。它提供了一个映射，外部用户存储和 Red Hat Single Sign-On 使用的用户 metamodel。

```
package org.keycloak.models;

public interface UserModel extends RoleMapperModel {
    String getId();

    String getUsername();
    void setUsername(String username);

    String getFirstName();
    void setFirstName(String firstName);

    String getLastName();
    void setLastName(String lastName);

    String getEmail();
    void setEmail(String email);
...
}
```

UserModel 实施提供对用户的读取和更新元数据的访问权限，包括用户名、名称、电子邮件、角色和组映射等内容，以及其他任意属性。

org.keycloak.models 软件包中包含其他模型类，它们代表 Red Hat Single Sign-On metamodel: RealmModel、RoleModel、GroupModel、clientModel、client Model 等等。

7.3.1. 存储 Ids

UserModel 的一个重要方法是 getId（） 方法。实施 UserModel 开发人员时，必须了解用户 ID 格式。格式必须是：

```
"f:" + component id + ":" + external id
```

Red Hat Single Sign-On 运行时通常必须在其用户 id 中查找用户。用户 ID 包含充足的信息，以便运行时不必查询系统中每一个 UserStorageProvider 来查找用户。

组件 ID 是从 ComponentModel.getId（） 返回的 id。在创建提供程序类时，ComponentModel 作为参数传递，因此您可以从那里获取它。外部 ID 是您的提供程序类需要在外部存储中查找用户的信息。这通常是用户名或 uid。例如，它可能类似如下：

```
f:332a234e31234:wburke
```

当运行时通过 id 查找时，将解析 id 来获取组件 ID。组件 ID 用于查找最初用于加载用户的 UserStorageProvider。然后，该提供程序会传递 id。该提供程序再次解析 id 来获取外部 ID，它将用于在外部用户存储中查找用户。

7.4. 打包和部署

为了让红帽单点登录识别提供程序，您需要向 JAR 添加文件： META-INF/services/org.keycloak.storage.UserStorageProviderFactory。此文件必须包含 UserStorageProviderFactory 实施的完全限定类名列表：

```
org.keycloak.examples.federation.properties.ClasspathPropertiesStorageFactory
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

要部署此 jar，只需将其复制到 standalone/deployments/ 目录中。=== Simple read-only, lookup 示例

为了说明实施用户存储 SPI 的基础知识，让我们来浏览一个简单示例。在本章中，您将看到实施简单的 UserStorageProvider，它将在简单属性文件中查找用户。属性文件包含用户名和密码定义，并硬编码到 classpath 上的特定位置。此提供程序将能够通过 ID 和用户名查找用户，并可验证密码。源自此提供程序的用户是只读的。

7.4.1. 供应商类

首先要了解的是 UserStorageProvider 类。

```
public class PropertyFileUserStorageProvider implements
        UserStorageProvider,
        UserLookupProvider,
        CredentialInputValidator,
        CredentialInputUpdater
{
...
}
```

我们的供应商类别 PropertyFileUserStorageProvider 实现了许多接口。它实施 UserStorageProvider，因为这是 SPI 的基础要求。它实施 UserLookupProvider 接口，因为我们希望能使用此提供程序存储的用户登录。它实施 CredentialInputValidator 接口，因为我们希望使用登录屏幕验证输入的密码。我们的属性文件是只读的。我们实施 CredentialInputUpdater，因为我们希望在用户尝试更新其密码时发布错误条件。

```
    protected KeycloakSession session;
    protected Properties properties;
    protected ComponentModel model;
    // map of loaded users in this transaction
    protected Map<String, UserModel> loadedUsers = new HashMap<>();

    public PropertyFileUserStorageProvider(KeycloakSession session, ComponentModel
model, Properties properties) {
        this.session = session;
        this.model = model;
        this.properties = properties;
    }
```

此提供程序类的构造器将存储对 KeycloakSession、CoonModel 和 属性文件的引用。稍后我们将使用所有这些产品。另请注意，有加载的用户映射。每当我们找到某个用户时，我们将将其存储在此地图中，这样我们都避免在同一交易中再次重新命名。这是为了遵守许多提供商，需要达到此目的（即，任何与 JPA 集成的供应商）的良好做法。请记住，每个事务一次创建提供程序类实例，并在事务完成后关闭。

7.4.1.1. UserLookupProvider 实现

```
    @Override
```

```
public UserModel getUserByUsername(String username, RealmModel realm) {
    UserModel adapter = loadedUsers.get(username);
    if (adapter == null) {
        String password = properties.getProperty(username);
        if (password != null) {
            adapter = createAdapter(realm, username);
            loadedUsers.put(username, adapter);
        }
    }
    return adapter;
}

protected UserModel createAdapter(RealmModel realm, String username) {
    return new AbstractUserAdapter(session, realm, model) {
        @Override
        public String getUsername() {
            return username;
        }
    };
}

@Override
public UserModel getUserById(String id, RealmModel realm) {
    StorageId storageId = new StorageId(id);
    String username = storageId.getExternalId();
    return getUserByUsername(username, realm);
}

@Override
public UserModel getUserByEmail(String email, RealmModel realm) {
    return null;
}
```

当用户登录时，Red Hat Single Sign-On 登录页面会调用 getUserByUsername （） 方法。在我们的实施中，我们首先检查 加载的Users 映射，以查看该用户是否已在此事务中载入。如果尚未加载，我们查看了用户名的属性文件。如果存在，我们创建了 UserModel 实施，将其存储在 loadUsers 中以供以后参考，然后返回此实例。

createAdapter （） 方法使用 helper 类 org.keycloak.storage.adapter.AbstractUserAdapter。这为 UserModel 提供了一个基本实现。它使用用户的用户名作为外部 ID，根据所需的存储 ID 格式自动生成用户 id。

```
"f:" + component id + ":" + username
```

每个 get method of AbstractUserAdapter 都会返回 null 或空集合。但是，返回角色和组映射的方法会返回为每个用户为域配置的默认角色和组。AbstractUserAdapter 的每个设置方法都会抛出一个 org.keycloak.storage.ReadOnlyException。因此，如果您试图修改 Admin 控制台中的用户，则会出现错误。

getUserById （） 方法使用 org.keycloak.storage.StorageId helper 类解析 id 参数。调用 StorageId.getExternalId （） 方法来获取嵌入在 id 参数中的用户名。然后，方法会将 委派为 getUserByUsername （） 。

不会存储电子邮件，因此 getUserByEmail （） 方法返回 null。

7.4.1.2. CredentialInputValidator 实现

接下来，让我们查看 CredentialInputValidator 的方法实施。

```
@Override
public boolean isConfiguredFor(RealmModel realm, UserModel user, String credentialType)
{
    String password = properties.getProperty(user.getUsername());
    return credentialType.equals(PasswordCredentialModel.TYPE) && password != null;
}

@Override
public boolean supportsCredentialType(String credentialType) {
    return credentialType.equals(PasswordCredentialModel.TYPE);
}

@Override
public boolean isValid(RealmModel realm, UserModel user, CredentialInput input) {
    if (!supportsCredentialType(input.getType())) return false;

    String password = properties.getProperty(user.getUsername());
    if (password == null) return false;
    return password.equals(input.getChallengeResponse());
}
```

运行时调用 isConfiguredFor （） 方法来确定是否为用户配置了特定的凭证类型。此方法检查为该用户设置了密码。

supportCredentialType （） 方法返回是否支持特定凭证类型的验证。我们检查该凭证类型是否为密码。

isValid （） 方法负责验证密码。CredentialInput 参数只是所有凭证类型的抽象接口。我们确保支持凭证类型，同时也是 UserCredentialModel 的实例。当用户通过登录页面登录时，密码输入的纯文本会被放入一个 UserCredentialModel 实例中。isValid （） 方法根据属性文件中存储的纯文本密码检查这个值。返回值为 true 表示密码有效。

7.4.1.3. CredentialInputUpdater 实现

如前所述，本例中我们实现 CredentialInputUpdater 接口的唯一原因是，对用户密码的修改是强制修改。我们必须这样做的原因是，由于该运行时可在 Red Hat Single Sign-On 本地存储中覆盖密码。本章稍后会对此进行讨论。

```
    @Override
    public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input)
{
        if (input.getType().equals(PasswordCredentialModel.TYPE)) throw new
ReadOnlyException("user is read only for this update");

        return false;
    }

    @Override
    public void disableCredentialType(RealmModel realm, UserModel user, String
credentialType) {

    }

    @Override
    public Set<String> getDisableableCredentialTypes(RealmModel realm, UserModel user) {
        return Collections.EMPTY_SET;
    }
```

updateCredential （） 方法只检查凭证类型是否为密码。如果是，则引发 ReadOnlyException。

7.4.2. 供应商工厂实施

现在，该提供商课程已经完成，现在我们把注意力转向了提供商的工厂级。

```
public class PropertyFileUserStorageProviderFactory
                implements UserStorageProviderFactory<PropertyFileUserStorageProvider> {

    public static final String PROVIDER_NAME = "readonly-property-file";

    @Override
    public String getId() {
        return PROVIDER_NAME;
    }
```

首先需要注意的是，在实施 UserStorageProviderFactory 类时，您必须作为模板参数传递 concrete 提供程序类实施。此处我们指定我们在之前定义的提供程序类： PropertyFileUserStorageProvider。

警告

如果没有指定模板参数，您的供应商将无法正常工作。运行时通过类内省来确定提供程序实施 *的功能接口。*

getId （） 方法标识运行时中的 factory，当您想要为域启用用户存储供应商时，admin 控制台也会显示字符串。

7.4.2.1. 初始化

```
    private static final Logger logger =
Logger.getLogger(PropertyFileUserStorageProviderFactory.class);
    protected Properties properties = new Properties();

    @Override
    public void init(Config.Scope config) {
        InputStream is = getClass().getClassLoader().getResourceAsStream("/users.properties");

        if (is == null) {
            logger.warn("Could not find users.properties in classpath");
        } else {
            try {
                properties.load(is);
            } catch (IOException ex) {
                logger.error("Failed to load users.properties file", ex);
            }
        }
    }

    @Override
    public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        return new PropertyFileUserStorageProvider(session, model, properties);
    }
```

UserStorageProviderFactory 接口具有可实施的可选 init （） 方法。当 Red Hat Single Sign-On 引导时，每个提供程序工厂仅创建一个实例。另外，init （） 方法会在每次工厂实例上调用。您还可以实施 postInit （） 方法。调用每个工厂的 init （） 方法后，会调用其 postInit （） 方法。

在 init （） 方法实施中，我们找到含有 classpath 中用户声明的属性文件。然后，我们使用在其中存储的用户名和密码载入 properties 字段。

Config.Scope 参数是通过服务器配置配置的工厂配置。

例如，将以下内容添加到 standalone.xml 中：

```
<spi name="storage">
   <provider name="readonly-property-file" enabled="true">
      <properties>
         <property name="path" value="/other-users.properties"/>
      </properties>
   </provider>
</spi>
```

我们可以指定用户属性文件的类路径，而不是硬编码。然后，您可以检索 PropertyFileUserStorageProviderFactory.init （） 中的配置：

```
public void init(Config.Scope config) {
   String path = config.get("path");
   InputStream is = getClass().getClassLoader().getResourceAsStream(path);

   ...
}
```

7.4.2.2. 创建方法

创建提供程序工厂的最后一步是 create （） 方法。

```
   @Override
   public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
      return new PropertyFileUserStorageProvider(session, model, properties);
   }
```

我们简单地分配 PropertyFileUserStorageProvider 类。这种创建方法将根据事务调用一次。

7.4.3. 打包和部署

我们提供程序实施的类文件应放在 jar 中。您还必须在 META-INF/services/org.keycloak.storage.UserStorageProviderFactory 文件中声明 provider factory 类。

```
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

要部署此 jar，只需将它复制到 standalone/deployments/ 目录中。

7.4.4. 在 Admin 控制台中启用供应商

您可以在 Admin Console 的 User Federation 页面中启用用户存储供应商。

流程

1. 从列表中选择刚才创建的供应商： readonly-property-file。

   将显示我们供应商的配置页面。

2. 单击 Save，因为我们没有任何配置。

3. 返回到主 User Federation 页面

   您现在可以看到您的供应商列出。

现在，您将能够在 users.properties 文件中声明的用户登录。此用户只能在登录后查看帐户页面。

7.5. 配置技术

我们的 PropertyFileUserStorageProvider 示例是一对一的。它被硬编码为嵌入到提供程序 jar 中的属性文件，此文件并并不有用。我们希望使此文件在提供商的实例可配置位置。换句话说，我们可能希望在多个不同域中多次重复使用此提供程序，并指向完全不同的用户属性文件。我们还要在管理控制台 UI 中执行此配置。

UserStorageProviderFactory 有额外的方法，您可以实施这些供应商配置。您描述您要为每个提供程序配置的变量，而 Admin Console 会自动呈现一个通用输入页面来收集此配置。实施时，回调方法也会在保存配置前验证配置，首次创建提供程序，以及更新时间。UserStorageProviderFactory 从 org.keycloak.component.ComponentFactory 接口继承这些方法。

```
List<ProviderConfigProperty> getConfigProperties();

default
void validateConfiguration(KeycloakSession session, RealmModel realm, ComponentModel
model)
```

```
        throws ComponentValidationException
    {

    }

    default
    void onCreate(KeycloakSession session, RealmModel realm, ComponentModel model) {

    }

    default
    void onUpdate(KeycloakSession session, RealmModel realm, ComponentModel model) {

    }
```

ComponentFactory.getConfigProperties （）方法返回 org.keycloak.provider.ProviderConfigProperty 实例列表。这些实例声明了呈现和存储提供程序配置变量所需的元数据。

7.5.1. 配置示例

让我们展开 PropertyFileUserStorageProviderFactory 示例，以便将提供商实例指向磁盘上的特定文件。

PropertyFileUserStorageProviderFactory

```
public class PropertyFileUserStorageProviderFactory
                  implements UserStorageProviderFactory<PropertyFileUserStorageProvider> {

    protected static final List<ProviderConfigProperty> configMetadata;

    static {
        configMetadata = ProviderConfigurationBuilder.create()
                .property().name("path")
                .type(ProviderConfigProperty.STRING_TYPE)
                .label("Path")
                .defaultValue("${jboss.server.config.dir}/example-users.properties")
                .helpText("File path to properties file")
                .add().build();
    }

    @Override
    public List<ProviderConfigProperty> getConfigProperties() {
        return configMetadata;
    }
```

ProviderConfigurationBuilder 类是一个很好的帮助程序类，用于创建配置属性列表。这里我们指定一个名为 path 的变量，它是 String 类型。在此提供程序的 Admin Console 配置页面上，此配置变量被标记为 Path，默认值为 ${jboss.server.config.dir}/example-users.properties。当您将鼠标指针悬停在这个配置选项提示中时，它会显示帮助文本、属性文件 的文件路径。

下一个要做的事情是验证该文件是否在磁盘上。我们不想在 realm 中启用此提供程序的实例，除非它指向有效的用户属性文件。为此，我们实施 validateConfiguration （） 方法。

```
    @Override
    public void validateConfiguration(KeycloakSession session, RealmModel realm,
ComponentModel config)
                   throws ComponentValidationException {
        String fp = config.getConfig().getFirst("path");
        if (fp == null) throw new ComponentValidationException("user property file does not
exist");
        fp = EnvUtil.replace(fp);
        File file = new File(fp);
        if (!file.exists()) {
            throw new ComponentValidationException("user property file does not exist");
        }
    }
```

在 validateConfiguration （） 方法中，我们从 ComponentModel 获取配置变量，并检查是否磁盘上是否存在该文件。请注意，我们使用 org.keycloak.common.util.EnvUtil.replace （） 方法。使用此方法，任何具有 ${} 的字符串都将替换为一个系统属性值。${jboss.server.config.dir} 字符串对应于 服务器的配置/ 目录，在本例中非常有用。

接下来，我们要做的是删除旧的 init （） 方法。我们这样做，因为用户属性文件将根据提供商实例唯一。我们将此逻辑移到 create （） 方法。

```
    @Override
    public PropertyFileUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        String path = model.getConfig().getFirst("path");

        Properties props = new Properties();
        try {
            InputStream is = new FileInputStream(path);
            props.load(is);
            is.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
```

```
        return new PropertyFileUserStorageProvider(session, model, props);
    }
```

当然，这个逻辑是效率，因为每个事务都从磁盘中读取整个用户属性文件，但希望以简单的方式表明这一点，如何在配置变量中使用 hook。

7.5.2. 在 Admin 控制台中配置供应商

现在，当在 Admin 控制台中配置供应商时，可以设置 路径 变量。

7.6. 添加/删除用户并查询功能接口

通过我们的示例，我们尚未完成一件事情是，允许它添加和删除用户或更改密码。示例中定义的用户还无法在管理控制台中查询或查看。要添加这些增强功能，我们的示例供应商必须实施 UserQueryProvider 和 UserRegistrationProvider 接口。

7.6.1. 实施用户RegistrationProvider

使用这个步骤实施从特定存储中添加和删除用户，我们首先需要将此属性文件保存到磁盘中。

PropertyFileUserStorageProvider

```
    public void save() {
        String path = model.getConfig().getFirst("path");
        path = EnvUtil.replace(path);
        try {
            FileOutputStream fos = new FileOutputStream(path);
            properties.store(fos, "");
            fos.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

然后，addUser（）和 removeUser（）方法的实现会变得简单。

PropertyFileUserStorageProvider

```
public static final String UNSET_PASSWORD="#$!-UNSET-PASSWORD";

@Override
public UserModel addUser(RealmModel realm, String username) {
   synchronized (properties) {
      properties.setProperty(username, UNSET_PASSWORD);
      save();
   }
   return createAdapter(realm, username);
}

@Override
public boolean removeUser(RealmModel realm, UserModel user) {
   synchronized (properties) {
      if (properties.remove(user.getUsername()) == null) return false;
      save();
      return true;
   }
}
```

请注意，在添加用户时，我们将属性映射的 password 值设置为 UNSET_PASSWORD。我们这样做，因为我们不能对属性值中的属性具有 null 值。我们还必须修改 CredentialInputValidator 方法以反映这一点。

如果提供程序实施 UserRegistrationProvider 接口，则将调用 addUser （） 方法。如果您的供应商有配置开关来关闭添加用户，则从此方法返回 null 将跳过该提供程序并调用下一个用户。

PropertyFileUserStorageProvider

```
@Override
public boolean isValid(RealmModel realm, UserModel user, CredentialInput input) {
   if (!supportsCredentialType(input.getType()) || !(input instanceof UserCredentialModel))
return false;

   UserCredentialModel cred = (UserCredentialModel)input;
   String password = properties.getProperty(user.getUsername());
   if (password == null || UNSET_PASSWORD.equals(password)) return false;
   return password.equals(cred.getValue());
}
```

由于现在我们可以保存我们的属性文件，因此允许密码更新也有意义。

PropertyFileUserStorageProvider

```
@Override
public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input)
{
    if (!(input instanceof UserCredentialModel)) return false;
    if (!input.getType().equals(CredentialModel.PASSWORD)) return false;
    UserCredentialModel cred = (UserCredentialModel)input;
    synchronized (properties) {
        properties.setProperty(user.getUsername(), cred.getValue());
        save();
    }
    return true;
}
```

现在，我们可以实施禁用密码。

PropertyFileUserStorageProvider

```
@Override
public void disableCredentialType(RealmModel realm, UserModel user, String
credentialType) {
    if (!credentialType.equals(CredentialModel.PASSWORD)) return;
    synchronized (properties) {
        properties.setProperty(user.getUsername(), UNSET_PASSWORD);
        save();
    }

}

private static final Set<String> disableableTypes = new HashSet<>();

static {
    disableableTypes.add(CredentialModel.PASSWORD);
}
```

```
@Override
public Set<String> getDisableableCredentialTypes(RealmModel realm, UserModel user) {

    return disableableTypes;
}
```

通过实施这些方法，您现在可以在 Admin 控制台中更改和禁用用户的密码。

7.6.2. 实施 UserQueryProvider

如果不实施 UserQueryProvider，管理控制台将无法查看和管理我们的示例提供者加载的用户。我们来看一下实施此界面。

PropertyFileUserStorageProvider

```
@Override
public int getUsersCount(RealmModel realm) {
    return properties.size();
}

@Override
public List<UserModel> getUsers(RealmModel realm) {
    return getUsers(realm, 0, Integer.MAX_VALUE);
}

@Override
public List<UserModel> getUsers(RealmModel realm, int firstResult, int maxResults) {
    List<UserModel> users = new LinkedList<>();
    int i = 0;
    for (Object obj : properties.keySet()) {
        if (i++ < firstResult) continue;
        String username = (String)obj;
        UserModel user = getUserByUsername(username, realm);
        users.add(user);
        if (users.size() >= maxResults) break;
    }
    return users;
}
```

getUsers （） 方法迭代属性文件的密钥集，委派为 getUserByUsername （） 来加载用户。请注

意，我们根据 第一个 结果和 maxResults 参数索引了这个调用。如果您的外部存储不支持分页，则您必须执行类似的逻辑。

PropertyFileUserStorageProvider

```
@Override
public List<UserModel> searchForUser(String search, RealmModel realm) {
   return searchForUser(search, realm, 0, Integer.MAX_VALUE);
}

@Override
public List<UserModel> searchForUser(String search, RealmModel realm, int firstResult, int
maxResults) {
    List<UserModel> users = new LinkedList<>();
    int i = 0;
    for (Object obj : properties.keySet()) {
       String username = (String)obj;
       if (!username.contains(search)) continue;
       if (i++ < firstResult) continue;
       UserModel user = getUserByUsername(username, realm);
       users.add(user);
       if (users.size() >= maxResults) break;
    }
    return users;
}
```

searchForUser （） 的第一个声明采用 String 参数。这应该是字符串，用于搜索用户名和电子邮件属性来查找用户。这个字符串可以是子字符串，这也是我们在进行搜索时使用 String.contains （） 方法的原因。

PropertyFileUserStorageProvider

```
@Override
public List<UserModel> searchForUser(Map<String, String> params, RealmModel realm) {
   return searchForUser(params, realm, 0, Integer.MAX_VALUE);
}

@Override
public List<UserModel> searchForUser(Map<String, String> params, RealmModel realm, int
firstResult, int maxResults) {
    // only support searching by username
    String usernameSearchString = params.get("username");
```

```
        if (usernameSearchString == null) return Collections.EMPTY_LIST;
        return searchForUser(usernameSearchString, realm, firstResult, maxResults);
    }
```

采用 Map 参数的 searchForUser （） 方法可以根据第一个、姓、用户名和电子邮件搜索用户。我们只存储用户名，因此我们只根据用户名进行搜索。我们委派给 搜索ForUser （）。

PropertyFileUserStorageProvider

```
    @Override
    public List<UserModel> getGroupMembers(RealmModel realm, GroupModel group, int
firstResult, int maxResults) {
        return Collections.EMPTY_LIST;
    }

    @Override
    public List<UserModel> getGroupMembers(RealmModel realm, GroupModel group) {
        return Collections.EMPTY_LIST;
    }

    @Override
    public List<UserModel> searchForUserByUserAttribute(String attrName, String attrValue,
RealmModel realm) {
        return Collections.EMPTY_LIST;
    }
```

我们不存储组或属性，因此其他方法会返回一个空列表。

7.7. 增强外部存储

PropertyFileUserStorageProvider 示例实际上有限。虽然我们可以使用存储在属性文件中的用户登录，但我们将无法做其他工作。如果此提供程序加载的用户需要特殊的角色或组映射来完全访问特定应用程序，我们就无法向这些用户添加其他角色映射。您也可以修改或添加其他重要属性，如电子邮件、名字和姓氏。

对于这些类型的问题，Red Hat Single Sign-On 允许您通过在 Red Hat Single Sign-On 的数据库中存储额外信息来增强外部存储。这称为联合用户存储，并封装在

org.keycloak.storage.federated.UserFederatedStorageProvider 类中。

UserFederatedStorageProvider

```
package org.keycloak.storage.federated;

public interface UserFederatedStorageProvider extends Provider {

    Set<GroupModel> getGroups(RealmModel realm, String userId);
    void joinGroup(RealmModel realm, String userId, GroupModel group);
    void leaveGroup(RealmModel realm, String userId, GroupModel group);
    List<String> getMembership(RealmModel realm, GroupModel group, int firstResult, int max);

...
```

UserFederatedStorageProvider 实例可在 KeycloakSession.userFederatedStorage（）方法中找到。它具有存储属性、组和角色映射、不同凭据类型和所需操作的所有不同类型的方法。如果您的外部存储的数据model 无法支持完整的 Red Hat Single Sign-On 功能集，则该服务可能会填补空白。

Red Hat Single Sign-On 附带了帮助类 org.keycloak.storage.adapter.AbstractUserAdapterFederatedStorage，除了 get/set of username to user federated storage 之外。覆盖您需要覆盖的方法，以委派给外部存储表示法。强烈建议您阅读本课程的 javadoc，因为它有较小的受保护的方法。具体围绕组成员资格和角色映射。

7.7.1. 8 月示例

在我们的 PropertyFileUserStorageProvider 示例中，我们需要一个简单的更改，以便我们的提供程序使用 AbstractUserAdapterFederatedStorage。

PropertyFileUserStorageProvider

```
    protected UserModel createAdapter(RealmModel realm, String username) {
        return new AbstractUserAdapterFederatedStorage(session, realm, model) {
            @Override
            public String getUsername() {
                return username;
            }

            @Override
```

```
            public void setUsername(String username) {
                String pw = (String)properties.remove(username);
                if (pw != null) {
                    properties.put(username, pw);
                    save();
                }
            }
        };
    }
```

我们改为定义 AbstractUserAdapterFederatedStorage 的匿名类实施。setUsername （） 方法更改属性文件并保存它。

7.8. 导入实施策略

在实施用户存储供应商时，可以采取的另一策略。您可以在 Red Hat Single Sign-On 内置用户数据库中本地创建用户，并将属性从外部存储中复制到此本地副本中。这种方法有很多优点。

- 红帽单点登录基本上成为外部存储的持久性用户缓存。导入用户后，您将不再通过外部存储来减少负载。

- 如果您要作为您的官方用户商店使用 Red Hat Single Sign-On 并弃用旧的外部存储，您可以慢慢地迁移应用程序以使用 Red Hat Single Sign-On。迁移完所有应用程序后，取消链接导入的用户，并停用旧的传统外部存储。

使用导入策略有一些明显的缺点：

- 第一次查找用户时，需要多个更新红帽单点登录数据库。这可能会给负载带来巨大性能损失，并给红帽单点登录数据库带来大量负担。用户联合存储方法将仅根据需要存储额外的数据，且永远不会根据外部存储的功能使用。

- 通过导入方法，您必须保持本地红帽单点登录存储和外部存储同步。User Storage SPI 有功能接口来支持同步，但这可能会很快变得困难和我。

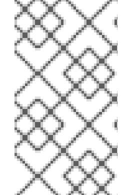
要实施导入策略，只需检查第一个用户是否在本地导入该策略。如果返回本地用户，如果没有在本地创建用户并从外部存储导入数据。您还可以代理本地用户，以便自动同步大多数更改。

这将是一个位表述，但我们扩展了 PropertyFileUserStorageProvider 来采用这种方法。首先，我们首先修改 createAdapter （） 方法。

PropertyFileUserStorageProvider

```
protected UserModel createAdapter(RealmModel realm, String username) {
    UserModel local = session.userLocalStorage().getUserByUsername(username, realm);
    if (local == null) {
        local = session.userLocalStorage().addUser(realm, username);
        local.setFederationLink(model.getId());
    }
    return new UserModelDelegate(local) {
        @Override
        public void setUsername(String username) {
            String pw = (String)properties.remove(username);
            if (pw != null) {
                properties.put(username, pw);
                save();
            }
            super.setUsername(username);
        }
    };
}
```

在此方法中，我们调用 KeycloakSession.userLocalStorage （） 方法来获取对本地红帽单点登录用户存储的引用。我们可以看到用户是否存储在本地，我们是否在本地添加它。不要设置本地用户的 id。让 Red Hat Single Sign-On 会自动生成 id。另请注意，我们称之为 UserModel.setFederationLink （），并传递我们供应商的 ComponentModel 的 ID。这会在供应商和导入的用户之间设置链接。



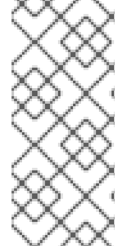
注意

删除用户存储供应商时，其导入的任何用户也将被删除。这是调用 UserModel.setFederationLink （） 的目的之一。

要注意的一点是，如果本地用户链接，您的存储供应商仍将委派给它从 CredentialInputValidator 和 CredentialInputUpdater 接口实施的方法。从验证或更新返回 false 只会使 Red Hat Single Sign-On 观察到是否使用本地存储进行验证或更新。

另请注意，我们使用 org.keycloak.models.utils.UserModelDelegate 类代理本地用户。这个类是

UserModel 的实现。每个方法都只委托给 UserModel，使用.我们将覆盖此委派类的 setUsername （）方法，以便自动与属性文件同步。对于您的供应商，您可以使用此方法 *截获* 本地 UserModel 上的其他方法，以便与外部存储执行同步。例如，get 方法可确保本地存储处于同步状态。设置方法使外部存储与本地存储保持同步。需要注意的是，getId （） 方法应始终返回您在本地创建用户时自动生成的 id。您不应该返回联合 ID，如其它非导入示例所示。



注意

如果您的供应商实施 UserRegistrationProvider 接口，则 removeUser （） 方法不需要从本地存储中删除该用户。运行时将自动执行此操作。另请注意，removeUser （） 将在从本地存储中删除之前调用。

7.8.1. ImportedUserValidation 接口

如果您记得在本章前面部分，我们讨论如何查询用户。如果用户已找到，则会首先查询本地存储，然后查询结束。因为我们需要代理本地用户 Model，所以我们可能会使用户名保持同步。每当从本地数据库加载链接的本地用户时，用户存储 SPI 具有回调。

```
package org.keycloak.storage.user;
public interface ImportedUserValidation {
    /**
     * If this method returns null, then the user in local storage will be removed
     *
     * @param realm
     * @param user
     * @return null if user no longer valid
     */
    UserModel validate(RealmModel realm, UserModel user);
}
```

加载链接的本地用户时，如果用户存储提供程序类实现了这个接口，则将调用 validate （） 方法。此处，您可以代理 作为参数传递的本地用户，并返回它。将使用新的 UserModel。您还可以选择检查以查看用户是否仍然在外部存储中。if validate （） 返回 null，则本地用户将从数据库中删除。

7.8.2. ImportSynchronization 接口

通过导入策略，您可以看到本地用户副本可以和外部存储同步。例如，一个用户可能已从外部存储中删除。User Storage SPI 有另外一个界面，您可以实现这个界面，以便处理这个接口 org.keycloak.storage.user.ImportSynchronization:

```
package org.keycloak.storage.user;

public interface ImportSynchronization {
    SynchronizationResult sync(KeycloakSessionFactory sessionFactory, String realmId,
UserStorageProviderModel model);
```

```
    SynchronizationResult syncSince(Date lastSync, KeycloakSessionFactory sessionFactory,
String realmId, UserStorageProviderModel model);
}
```

此接口由供应商工厂实施。旦此界面由提供程序工厂实施，供应商的管理控制台管理页面会显示附加选项。您可以点击按钮手动强制同步。这会调用 ImportSynchronization.sync （） 方法。另外还会显示额外的配置选项，供您自动调度同步。自动同步调用 syncSince （） 方法。

7.9. 用户缓存

当用户对象由 ID、用户名或电子邮件查询被缓存时。当用户对象被缓存时，它会迭代整个 UserModel 接口，并将这些信息拉取到一个本地只读缓存。在集群中，这个缓存仍然是本地的，但它会变为无效的缓存。修改用户对象时，它将被驱除。这个驱除事件会传播到整个集群，以便其他节点的用户缓存也无效。

7.9.1. 管理用户缓存

您可以通过调用 KeycloakSession.userCache （） 来访问用户缓存。

```
/**
 * All these methods effect an entire cluster of Keycloak instances.
 *
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
public interface UserCache extends UserProvider {
    /**
     * Evict user from cache.
     *
     * @param user
     */
    void evict(RealmModel realm, UserModel user);

    /**
     * Evict users of a specific realm
     *
     * @param realm
     */
    void evict(RealmModel realm);

    /**
     * Clear cache entirely.
     *
     */
    void clear();
}
```

有方法可以驱除特定用户、特定域或整个缓存中包含的用户。

7.9.2. OnUserCache 回调接口

您可能需要缓存特定于供应商的实现的额外信息。每当缓存用户时，User Storage SPI 具有回调：org.keycloak.models.cache.OnUserCache。

```
public interface OnUserCache {
    void onCache(RealmModel realm, CachedUserModel user, UserModel delegate);
}
```

如果您的供应商类应该实现这个接口（如果它想要这个回调）。UserModel delegate 参数是您的供应商返回的 UserModel 实例。CachedUserModel 是一个展开的 UserModel 接口。这是在本地本地存储中缓存的实例。

```
public interface CachedUserModel extends UserModel {

    /**
     * Invalidates the cache for this user and returns a delegate that represents the actual data
provider
     *
     * @return
     */
    UserModel getDelegateForUpdate();

    boolean isMarkedForEviction();

    /**
     * Invalidate the cache for this model
     *
     */
    void invalidate();

    /**
     * When was the model was loaded from database.
     *
     * @return
     */
    long getCacheTimestamp();

    /**
     * Returns a map that contains custom things that are cached along with this model.  You
can write to this map.
     *
     * @return
     */
    ConcurrentHashMap getCachedWith();
}
```

这个 CachedUserModel 接口允许您从缓存中驱除用户并获取提供程序 UserModel 实例。getCachedWith （） 方法返回一个映射，允许您缓存与用户相关的其他信息。例如，凭证不是 UserModel 接口的一部分。如果要在内存中缓存凭据，您要使用 getCachedWith （） 方法实施 OnUserCache 和缓存用户凭证。

7.9.3. 缓存策略

在用户存储供应商的管理控制台管理页面中，您可以指定唯一的缓存策略。

7.10. 使用 JAKARTA EE

如果您正确设置了 META-INF/services 文件，则用户存储提供程序可以封装在任何 Jakarta EE 组件中。例如，如果您的供应商需要使用第三方库，您可以将供应商打包在 EAR 中，并将这些第三方库存储在 EAR/ 目录的 lib/ 目录中。另请注意，提供程序 JAR 可以使用 jboss-deployment-structure.xml 文件，该文件 EJB、WARS 和 EARs 可在 JBoss EAP 环境中使用。有关此文件的详情，请查看 JBoss EAP 文档。它允许您在其他精细操作中拉取外部依赖项。

提供程序实施需要是纯 java 对象。但我们目前还支持将 UserStorageProvider 类实施为有状态 EJB。如果要使用 JPA 连接到关系存储，这尤其有用。这是实现它的方式：

```
@Stateful
@Local(EjbExampleUserStorageProvider.class)
public class EjbExampleUserStorageProvider implements UserStorageProvider,
        UserLookupProvider,
        UserRegistrationProvider,
        UserQueryProvider,
        CredentialInputUpdater,
        CredentialInputValidator,
        OnUserCache
{
    @PersistenceContext
    protected EntityManager em;

    protected ComponentModel model;
    protected KeycloakSession session;

    public void setModel(ComponentModel model) {
        this.model = model;
    }

    public void setSession(KeycloakSession session) {
        this.session = session;
    }


    @Remove
    @Override
```

```
    public void close() {
    }
...
}
```

您必须定义 @Local 注释并指定您的提供程序类。如果没有这样做，EJB 将不会正确代理用户，您的供应商将无法正常工作。

您必须将 @Remove 注释放在提供程序的 close （） 方法上。如果没有，则有状态的 bean 将不会被清理，您最终可能会看到错误消息。

对于 UserStorageProvider，需要是普通 Java 对象。您工厂类在其 create （）方法中对 Stateful EJB 执行 JNDI 查找。

```
public class EjbExampleUserStorageProviderFactory
        implements UserStorageProviderFactory<EjbExampleUserStorageProvider> {

    @Override
    public EjbExampleUserStorageProvider create(KeycloakSession session, ComponentModel
model) {
        try {
            InitialContext ctx = new InitialContext();
            EjbExampleUserStorageProvider provider =
(EjbExampleUserStorageProvider)ctx.lookup(
                    "java:global/user-storage-jpa-example/" +
EjbExampleUserStorageProvider.class.getSimpleName());
            provider.setModel(model);
            provider.setSession(session);
            return provider;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

本例还假定您已在与提供程序相同的 JAR 中定义 JPA 部署。这意味着 persistent .xml 文件以及任何 JPA @Entity 类。

警告

使用 JPA 时，任何其他数据源必须是 XA 数据源。Red Hat Single Sign-On 数据源不是 XA 数据源。如果您在同一事务中与两个或多个非XA 数据源交互，服务器会返回错误消息。单个事务中只允许一个非XA 资源。有关部署 XA 数据源的详情，请参阅 JBoss EAP 手册。

不支持 CDI。

7.11. REST 管理 API

您可以通过管理员 REST API 创建、删除和更新用户存储供应商部署。User Storage SPI 基于通用组件接口构建，因此您将使用该通用 API 来管理您的供应商。

REST 组件 API 在您的域管理资源下提供。

```
/admin/realms/{realm-name}/components
```

我们将只显示与 Java 客户端交互的 REST API。希望您可以从这个 API 中提取如何执行此操作。

```
public interface ComponentsResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent,
@QueryParam("type") String type);

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query(@QueryParam("parent") String parent,
                                               @QueryParam("type") String type,
                                               @QueryParam("name") String name);
```

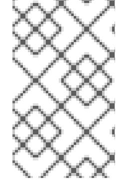
```
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    Response add(ComponentRepresentation rep);

    @Path("{id}")
    ComponentResource component(@PathParam("id") String id);
}

public interface ComponentResource {
    @GET
    public ComponentRepresentation toRepresentation();

    @PUT
    @Consumes(MediaType.APPLICATION_JSON)
    public void update(ComponentRepresentation rep);

    @DELETE
    public void remove();
}
```

要创建用户存储供应商，您必须指定提供程序 ID、字符串 provider 类型 org.keycloak.storage.UserStorageProvider，以及配置。

```
import org.keycloak.admin.client.Keycloak;
import org.keycloak.representations.idm.RealmRepresentation;
...

Keycloak keycloak = Keycloak.getInstance(
    "http://localhost:8080/auth",
    "master",
    "admin",
    "password",
    "admin-cli");
RealmResource realmResource = keycloak.realm("master");
RealmRepresentation realm = realmResource.toRepresentation();

ComponentRepresentation component = new ComponentRepresentation();
component.setName("home");
component.setProviderId("readonly-property-file");
component.setProviderType("org.keycloak.storage.UserStorageProvider");
component.setParentId(realm.getId());
component.setConfig(new MultivaluedHashMap());
component.getConfig().putSingle("path", "~/users.properties");

realmResource.components().add(component);

// retrieve a component

List<ComponentRepresentation> components =
realmResource.components().query(realm.getId(),
                                    "org.keycloak.storage.UserStorageProvider",
                                    "home");
component = components.get(0);
```

```
// Update a component

component.getConfig().putSingle("path", "~/my-users.properties");
realmResource.components().component(component.getId()).update(component);

// Remove a component

realmREsource.components().component(component.getId()).remove();
```

7.12. 从较早的用户联合 SPI 迁移



注意

本章仅在您已使用先前（及现已删除）用户 Federation SPI 实施的供应商时才适用。

在 Keycloak 版本 2.4.0 及更高版本中有一个用户 Federation SPI。虽然不被支持，但 Red Hat Single Sign-On 版本 7.0 也具有之前的 SPI 可用。这个较早的用户 Federation SPI 已从 Keycloak 版本 2.5.0 和 Red Hat Single Sign-On 版本 7.1 中删除。但是，如果您已经编写了这个早期的 SPI 的供应商，本章将讨论一些可以使用的策略来移植它。

7.12.1. 导入和非导入

较早的用户 Federation SPI 需要您在 Red Hat Single Sign-On 的数据库中创建用户的本地副本，并将信息从外部存储导入到本地副本。然而，这不再是一个要求。您仍然可以将之前的供应商的端口，但应考虑一个非导入策略是否为更好的方法。

导入策略的优点：

- 红帽单点登录基本上成为外部存储的持久性用户缓存。导入用户后，您将不再点击外部存储，从而减少负载。
- 如果您要作为您的官方用户商店使用 Red Hat Single Sign-On 并弃用较早的外部存储，您可以慢慢地迁移应用程序以使用 Red Hat Single Sign-On。迁移完所有应用程序后，取消链接导入的用户，并停用之前的传统外部存储。

使用导入策略有一些明显的缺点：

- 第一次查找用户时，需要多个更新红帽单点登录数据库。这可能会给负载带来巨大性能损失，并给红帽单点登录数据库带来大量负担。用户联合存储方法仅根据需要存储额外的数据，且

可能永远不会根据外部存储的功能使用。

- 通过导入方法，您必须保持本地红帽单点登录存储和外部存储同步。User Storage SPI 有功能接口来支持同步，但这可能会很快变得困难和我。

7.12.2. UserFederationProvider 与 UserStorageProvider

需要注意的是，UserFederationProvider 是一个完整的界面。您在这个接口中实施了每种方法。但是，UserStorageProvider 已将此接口划分为多个您根据需要实施的功能接口。

UserFederationProvider.getUserByUsername （） 和 getUserByEmail （） 在新的 SPI 中具有确切的等效点。两者之间的差别在于您如何导入。如果您要继续使用导入策略，您不再调用 KeycloakSession.userStorage （）.addUser （） 在本地创建用户。而是调用 KeycloakSession.userLocalStorage （）.addUser （）。userStorage （） 方法不再存在。

UserFederationProvider.validateAndProxy （） 方法已移至可选的功能接口 ImportedUserValidation。如果您正移植之前的供应商，您需要实施此界面。另请注意，在之前的 SPI 中，每次访问用户时都会调用此方法，即使本地用户位于缓存中。在后面的 SPI 中，只有从本地存储加载本地用户时，才会调用此方法。如果本地用户被缓存，则 ImportedUserValidation.validate （） 方法不会被调用。

之后的 SPI 中不再存在 UserFederationProvider.isValid （） 方法。

UserFederationProvider 方法 synchronizeRegistrations （）、registerUser （） 和 removeUser （） 已移至 UserRegistrationProvider 能力接口。这个新界面是可选的，因此如果您的供应商不支持创建和删除用户，则不必实施它。如果您的供应商有切换支持注册新用户，则在新的 SPI 中支持此操作，如果供应商不支持添加用户，则返回 null from UserRegistrationProvider.addUser （）。

现在，在 CredentialInputValidator 和 CredentialInputUpdater 接口中封装了相关的 UserFederationProvider 方法，这也是可选的，具体取决于您支持验证或更新凭证。用于 UserModel 方法中的凭证管理。它们也被移到 CredentialInputValidator 和 CredentialInputUpdater 接口。请注意，如果没有实现 CredentialInputUpdater 接口，则您的供应商提供的任何凭证都可以在 Red Hat Single Sign-On 存储中本地覆盖。因此，如果您希望您的凭据为只读，实施 CredentialInputUpdater.updateCredential （） 方法并返回 ReadOnlyException。

UserFederationProvider 查询方法，如 searchByAttributes （） 和 getGroupMembers （） 现在封装在一个可选接口 UserQueryProvider 中。如果不实施此接口，则在管理控制台中无法查看用户。您仍然能够登录。

7.12.3. UserFederationProvidery 与 UserStorageProviderFactory 相比

之前 SPI 中的同步方法现在封装在一个可选的 ImportSynchronization 接口中。如果您实施了同步逻辑，则您的新 UserStorageProviderFactory 实施 ImportSynchronization 接口。

7.12.4. 升级到新模型

User Storage SPI 实例存储在一组不同的关系表中。Red Hat Single Sign-On 会自动运行迁移脚本。如果为某个域部署任何较早的用户 Federation 提供程序，则它们将转换为更新的存储模型，包括数据的 ID。只有用户存储供应商 ID （例如，"ldap"，"kerberos"）作为先前的用户 Federation 提供程序存在时，才会进行此迁移。

因此，了解这种方法可以采用不同的方法。

1. 您可以在之前的 Red Hat Single Sign-On 部署中删除之前的供应商。这将删除您导入的所有用户的本地链接副本。然后，当您升级 Red Hat Single Sign-On 时，只需为域部署和配置新提供程序。

2. 第二个选项用于编写您的新提供程序，确保它具有相同的提供程序 ID：UserStorageProviderFactory.getId （）。确保此提供程序已部署到服务器。引导服务器，并内置迁移脚本从以前的数据模型转换到更新的数据模型。在这种情况下，所有之前导入的用户都可以正常工作。

如果您决定解决导入策略并重写您的 User Storage 提供程序，我们建议您在升级 Red Hat Single Sign-On 前删除之前的供应商。这将删除您导入的任何用户的本地导入副本。

7.13. 基于流的接口

Red Hat Single Sign-On 中的许多用户存储接口包含可能会返回大量对象集的查询方法，这可能会给内存消耗和处理时间造成显著影响。当查询方法的逻辑中使用对象内部状态的一个小子集时，这尤其如此。

为了为开发人员提供这些查询方法中处理大型数据集的效率，在用户存储界面中添加了 Streams 子接口。这些流子接口将 super-interfaces 中的原始基于集合的方法替换为基于流的变体，从而使基于集合的方法默认。基于集合的查询方法的默认实现调用其 Stream counterpart，并将结果收集到正确的集合类型中。

Streams 子接口允许实施基于流处理数据集的数据，并从该方法的潜在内存和性能优化中受益。提供要

实施的 Streams 子接口的接口包括几个 *功能接口*、org.keycloak.storage.fe deated 软件包中的所有接口，以及可能根据自定义存储实施范围而实现的其它接口。

请参阅此界面列表，这些接口为开发人员提供流子接口。

软件包	类
org.keycloak.credential	**CredentialInputUpdater**(*)
org.keycloak.models	**GroupModel**,**RoleMapperModel**,**UserModel**
org.keycloak.storage.federated	所有接口
org.kecyloak.storage.user	**UserQueryProvider**(*)

7000 表示接口是一个 *功能接口*

从流方法中受益的自定义用户存储实施应只是实施 Streams 子接口，而不是原始接口。例如，以下代码使用 UserQueryProvider 接口的 Streams 变体：

```
public class CustomQueryProvider extends UserQueryProvider.Streams {
...
   @Override
   Stream<UserModel> getUsersStream(RealmModel realm, Integer firstResult, Integer maxResults) {
       // custom logic here
   }

   @Override
   Stream<UserModel> searchForUserStream(String search, RealmModel realm) {
       // custom logic here
   }
...
}
```

第 8 章 VAULT SPI

8.1. VAULT 供应商

您可以使用 org.keycloak.vault 软件包中的 vault SPI 编写 Red Hat Single Sign-On 的自定义扩展来连接到任意 vault 实施。

内置文件说明文本 提供程序是实施此 SPI 的示例。通常适用以下规则：

- 要防止机密在域间泄漏，您可能需要隔离或限制可由域检索的机密。在这种情况下，您的供应商在查找 secret 时应该考虑域名，例如，为带有 realm 名称的条目添加前缀。例如，表达式 ${vault.key} 将按照在域 *A* 还是 realm *B* 中使用的，则通常会评估为不同的条目名称。为了区分域，域需要传递到创建的 VaultProvider 实例（从 KeycloakSession 参数获取的 VaultProviderFactory.create （） 方法）。

- vault 提供程序需要实施单一方法 get Secret，该方法为给定的 secret 名称返回 VaultRawSecret。该类包含 secret 的表示，可以是 byte[] 或 ByteBuffer，它应该在两个需求之间转换。请注意，此缓冲区将在使用后丢弃，具体如下方所述。

有关如何打包和部署自定义提供程序的详情，请参考 服务提供商接口 章节。

8.2. 从 VAULT 消耗值

库包含敏感数据，红帽单点登录会相应地对待 secret。在访问机密时，该 secret 仅从 JVM 内存中获取并保留在 JVM 内存中。然后，所有可能都尝试从 JVM 内存丢弃其内容。这可以通过在 try-with-resources 语句中使用 vault secret 来实现，如下所示：

```
char[] c;
try (VaultCharSecret cSecret = session.vault().getCharSecret(SECRET_NAME)) {
    // ... use cSecret
    c = cSecret.getAsArray().orElse(null);
    // if c != null, it now contains password
}

// if c != null, it now contains garbage
```

这个示例使用 KeycloakSession.vault （） 作为用于访问 secret 的入口点。直接使用 VaultProvider.obtainSecret 方法也可以直接使用。但是 vault （） 方法具有能够解释原始 secret （通常是字节阵列）的功能，作为字符数组(via vault （).getCharSecret （））或一个 String (via vault （).getStringSecret （）），除了获取原始未中断的值（通过 vault （）.getRawSecret （） 方法）。

请注意，由于 String 对象是不可变的，因此无法通过使用随机垃圾回收来覆盖其内容丢弃。虽然在默认的 VaultStringSecret 实现中已采取措施以防止内部使用字符串，但 String 对象中存储的 secret 至少将变为下一个 GC 轮流。因此，最好使用纯字节和字符数组和缓冲区。