Red Hat build of Keycloak 24.0

服务器开发人员指南

Last Updated: 2024-06-28

法律通告

摘要

本指南包含开发人员用来自定义红帽 Keycloak 24.0 的信息。

目录

第 1 章 前言

在某些示例列表中，在一行中显示什么内容不适合可用的页面宽度内。这些行已损坏。行末尾的 '\' 表示在页面中已被引入一个中断，以下几行缩进。因此：

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

实际上：

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

第 2 章 ADMIN REST API

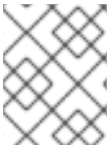
红帽构建的 Keycloak 附带了一个功能齐全的 Admin REST API，以及管理控制台提供的所有功能。

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 中描述了所需的权限。

您可以使用红帽构建的 Keycloak 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



注意

以下示例假设您使用 **master** 域中的 **密码** 创建了 **admin** 用户，如 Getting Started Guide 指南 所示。

流程

1. 获取 realm **master** 中具有用户名 **admin 和密码** 的访问令牌：

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

注意

默认情况下，此令牌在 1 分钟内过期

结果将是 JSON 文档。

2. 通过提取 **access_token** 属性的值来调用您需要的 API。

3. 通过在 API 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/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. 确保客户端配置如下：

- **client_id 是** 属于 realm **master**的机密客户端
- **client_id** 启用 **Service Accounts Enabled** 选项
- **client_id** 有一个自定义"Audience"映射器
  - 包含的客户端受众： **security-admin-console**

2. 检查 **client_id** 是否在"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/realms/master/protocol/openid-connect/token"
```

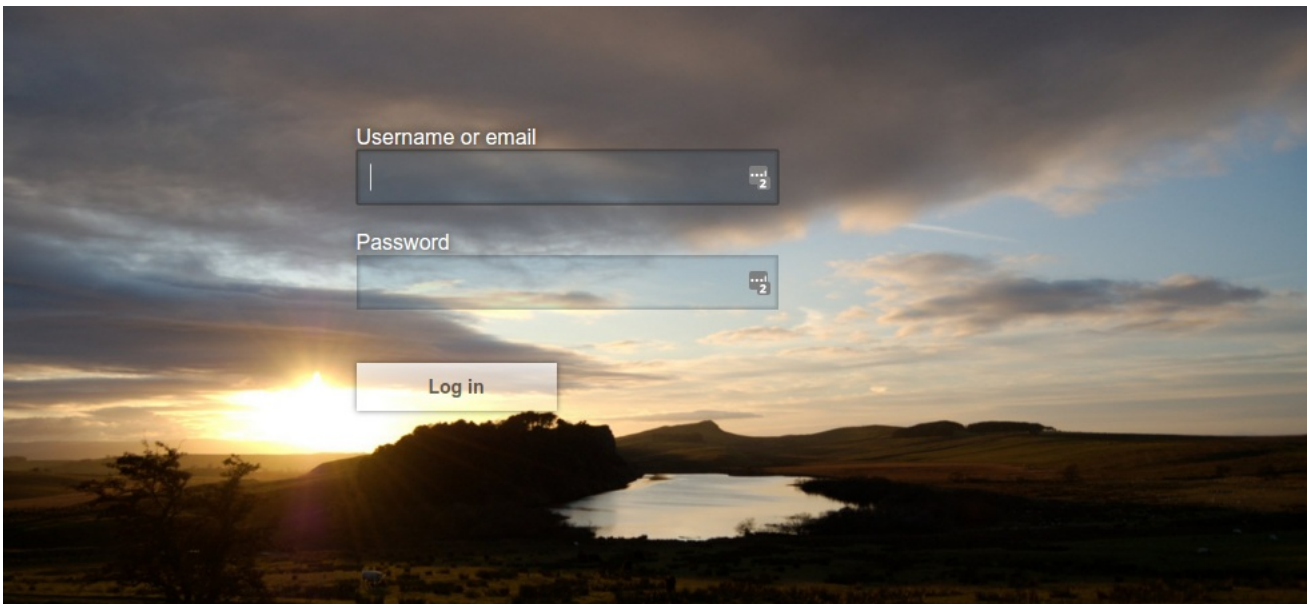
2.2. 其他资源

- 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

红帽构建的 Keycloak 为网页和电子邮件提供主题支持。这允许自定义面向最终用户的页面的外观和感觉，以便可以与您的应用程序集成。

图 3.1. 带有 sunrise 示例主题的登录页面



3.1. THEME 类型

主题可以提供一个或多个类型来自定义红帽构建的 Keycloak 的不同方面。可用的类型有：

- 帐户 - 帐户控制台
- Admin - 管理控制台
- 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，所有主题类型都通过 Admin 控制台进行配置。

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，单击 **Realm Settings**。
4. 点 **Themes** 选项卡。

注意

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

5. 要查看对 Admin Console 的更改，请刷新页面。
6. 使用 **spi-theme-welcome-theme** 选项更改 welcome 主题。
7. 例如：

```
bin/kc.[sh|bat] start --spi-theme-welcome-theme=custom-theme
```

3.3. 默认主题

红帽构建的 Keycloak 将与默认主题捆绑在服务器分发中的 JAR 文件 **keycloak-themes-24.0.0.redhat-00001.jar** 中。默认情况下，服务器的根目录会包含任何主题，但它包含一个 README 文件，其中包含一些有关默认主题的详情。**要简化升级，请不要直接编辑捆绑它们。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**
- **theme 属性**

除非计划替换每个单个页面，否则您应该扩展另一个主题。很可能您要扩展一些现有的主题。或者，如果您打算提供自己的管理员或帐户控制台实施，请考虑扩展 基础 主题。基本 主题由消息捆绑包组成，因

此此类实施需要从头开始，包括主 index.ftl Freemarker 模板的实现，但它可以利用消息捆绑包中的现有转换。

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

虽然创建一个主题，最好不要从 themes 目录中编辑主题资源，而无需重启红帽构建的 Keycloak。

流程

1. 使用以下选项运行 Keycloak：

```
bin/kc.[sh|bat] start --spi-theme-static-max-age=-1 --spi-theme-cache-themes=false --spi-theme-cache-templates=false
```

2. 在 themes 目录中创建目录。

   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。

3. 在主题目录中，为主题将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 主题，请创建目录 themes/mytheme/login。

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

   例如，若要配置主题 themes/mytheme/login 来扩展 基础 主题并导入一些通用资源，请创建文件 themes/mytheme/login/theme.properties，其内容如下：

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

   您现在已创建了支持登录类型的主题。

5. 登录 Admin 控制台以签出您的新主题

6. 选择您的域

7. 从菜单中，单击 Realm Settings。

8. 点 Themes 选项卡。

9. 对于 Login Theme select mytheme，点 Save。

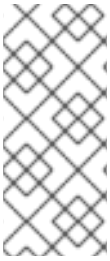
10. 打开域的登录页面。

    您可以通过应用程序登录或打开帐户控制台(/realms/{realm name}/account)来完成此操作。

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

注意

务必在生产环境中重新启用缓存，因为它会对性能有严重影响。



注意

如果要手动删除主题缓存的内容，您可以通过删除服务器发行版本的 data/tmp/kc-gzip-cache 目录来实现。如果您重新部署了自定义提供程序或自定义它们而不在之前的服务器执行中禁用它们缓存，则它很有用。

3.4.1. theme 属性

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

- Parent 主题(Parent theme to extend)
- Import - 从另一个主题导入资源

- common - 覆盖通用资源路径。如果没有指定，默认值为 common/keycloak。这个值将用作 ${url.resourcesCommonPath} 的后缀值，它通常用于 freemarker 模板(${url.resoucesCommonPath} 的值为主题 root uri)。

- 样式 - 要包含的以空格分隔的样式列表

- Locale - 支持的区域列表

有一些属性列表可用于更改用于特定元素类型的 css 类。有关这些属性的列表，请查看 keycloak 主题(sees/keycloak/<THEME TYPE>/theme.properties)的相应类型中的 theme.properties文件。

您还可以添加自己的自定义属性，并从自定义模板中使用它们。

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在 ${foo:defaultValue} 未找到时，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，并以模板中的格式结束。

下面是一个可能的例子：

```
javaVersion=${java.version}

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

3.4.2. 在主题中添加风格表

您可以在主题中添加一个或多个风格表。

流程

1. 在主题的 &lt ;THEME TYPE>/resources/css 目录中创建一个文件。

2. 将此文件添加到 theme.properties 中的 styles 属性中。

   例如，要将 styles.css 添加到 mytheme，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 themes/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。

   您会注意到应用的唯一样式是来自您的自定义风格表。

5. 要包括父主题中的样式，加载该主题的样式。编辑 themes/mytheme/login/theme.properties，并将 风格 改为：

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

   注意

   要覆盖父风格表中的样式，请确保您的风格表最后列出。

3.4.3. 在主题中添加脚本

您可以在主题中添加一个或多个脚本。

流程

1. 在主题的 &lt ;THEME TYPE>/resources/js 目录中创建一个文件。
2. 将此文件添加到 theme.properties 中的 scripts 属性中。

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

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

   然后，编辑 themes/mytheme/login/theme.properties 并添加：

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

3.4.4. 将镜像添加到主题

要使镜像可用于主题，请将它们添加到主题的 &lt ;THEME TYPE>/resources/img 目录中。它们可用于风格的表格，也可直接在 HTML 模板中使用。

例如，要将镜像添加到 mytheme，将镜像复制到 themes/mytheme/login/resources/img/image.jpg。

然后，您可以在自定义风格的表格中使用此镜像：

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

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

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

3.4.5. 将镜像添加到电子邮件主题

要使镜像可用于主题，请将它们添加到主题的 &lt ;THEME TYPE>/email/resources/img 目录中。它们可以直接在 HTML 模板中使用。

例如，要将镜像添加到 mytheme，将镜像复制到 themes/mytheme/email/resources/img/logo.jpg。

要直接在 HTML 模板中使用，请将以下内容添加到自定义 HTML 模板中：

```
<img src="${url.resourcesUrl}/img/image.jpg" alt="My image description">
```

3.4.6. 消息

模板中的文本从消息捆绑包加载。扩展另一个主题的主题将从父消息捆绑包中继承所有消息，您可以通过向主题中添加 < THEME TYPE>/messages/messages_en.properties 来覆盖单个消息。

例如，将登录表单上的 Username 替换为 mytheme 的 Username，创建 themes/mytheme/login/messages/messages_en.properties 文件，其内容如下：

```
usernameOrEmail=Your Username
```

在使用消息时，消息值（如 {0} 和 consumption）会被替换为参数。例如，在登录 {0} 中，{0} 被替换为域的名称。

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

3.4.7. 在域中添加语言

先决条件

- 要为域启用国际化，请参阅 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 属性中。要使用户可用于域 登录、帐户和 电子邮件 的语言，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 翻译添加到 mytheme 主题，请创建文件 themes/mytheme/login/messages/messages_no.properties，其内容如下：

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

   如果省略了消息的翻译，它们将使用 English。

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

   ```
   locales=en,no
   ```

4. 为帐户添加相同的 帐户 和电子邮件 主题类型。为此，请创建 themes/mytheme/account/messages/messages_no.properties，以及 themes/mytheme/email/messages/messages_no.properties。将这些文件留空将导致使用英文消息。

5. 将 themes/mytheme/login/theme.properties 复制到 themes/mytheme/account/theme.properties，将 themes/mytheme/email/theme.properties 复制到 themes/mytheme/email/theme.properties。

6. 为语言选择器添加翻译。这可以通过在英语中添加消息来完成。要做到这一点，请在 themes/mytheme/account/messages/messages/messages_en.properties 和 themes/mytheme/login/messages/messages_en.properties 中添加以下内容：

   ```
   locale_no=Norsk
   ```

默认情况下，消息属性文件应使用 UTF-8 进行编码。如果无法读取内容为 UTF-8，则 Keycloak 会回退到 ISO-8859-1 处理。Unicode 字符可以被转义，如 Java 的 PropertyResourceBundle 文档 中所述。之前的 Keycloak 版本支持使用类似 # encoding: UTF-8 的注释在第一行中指定编码，该注释不再被支持。

其他资源

- 如需了解如何选择当前区域设置的详细信息，请参阅 Locale Selector。

3.4.8. 添加自定义身份提供程序图标

红帽构建的 Keycloak 支持为自定义身份提供程序添加图标，这些图标显示在登录屏幕中。

流程

1. 在登录 主题.properties 文件中定义图标类（例如，es /mytheme/login/theme.properties），使用密钥模式 kcLogoIdP-<alias>。

2. 对于别名为 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
```

所有图标都在 PatternFly4 的官方网站上提供。社交供应商的图标已在 基础 登录主题属性 (es/keycloak/login/theme.properties)中定义，您可以在其中激发自己。

3.4.9. 创建自定义 HTML 模板

Red Hat build of Keycloak 使用 Apache Freemarker 模板来生成 HTML 和 render 页面。

虽然可以创建自定义模板以完全改变页面的呈现方式，但建议尽可能利用内置模板。原因包括：

- 在升级过程中，您可能会强制更新自定义模板，以从更新的版本获取最新的更新
- 通过为主题 配置 CSS 风格，您可以调整 UI 与 UI 设计标准和准则相匹配。
- 用户配置文件 允许您支持自定义用户属性并配置如何呈现它们。

在大多数情况下，您不需要更改模板，将红帽构建的 Keycloak 适应您的需求，但您可以通过创建 < THEME TYPE>/<TEMPLATE>.ftl 来覆盖您自己的主题中的单个模板。admin 和 account 控制台使用单

个模板 index.ftl 来渲染应用程序。

有关其他主题类型中的模板列表，请查看 JAR 文件中的 主题/基础/<THEME_TYPE > 目录，位于 $KEYCLOAK_HOME/lib/lib/main/org.keycloak.keycloak-themes-<VERSION>.jar。

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 主题创建一个自定义登录表单，请将 themes/base/login/login.ftl 复制到 themes/mytheme/login，并在编辑器中打开它。

   在第一行(<#import …>)后，添加 & lt;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 Keycloak 时，您可能需要更新您的自定义模板，以便在适用时对原始模板应用更改。

其他资源

- 有关如何编辑模板的详情，请查看 FreeMarker Manual。

3.4.10. 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消息。一个用于主题，一个用于纯文本正文，一个用于 html 正文。

要查看所有可用的电子邮件，请查看 themes/base/email/messages/messages_en.properties。

例如，要更改 mytheme 主题的密码恢复电子邮件，请创建 themes/mytheme/email/email/messages_en.properties，其内容如下：

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

3.5. 部署主题

主题目录可以部署到 Keycloak 的红帽构建中，方法是将主题目录复制到 主题 目录，也可以部署为存档。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。通过存档，可以更轻松地显示主题的版本控制副本，特别是当您有多个红帽构建的 Keycloak 实例（例如用于集群）时。

流程

1. 要将主题部署为存档，请使用主题资源创建一个 JAR 存档。

2. 将文件 META-INF/keycloak-themes.json 添加到存档中，其中列出了存档中的可用主题，以及每个主题提供的类型。

   例如，对于 mytheme theme create mytheme.jar，其内容如下：

   - 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

在这种情况下，META-INF/keycloak-themes.json 的内容将是：

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

单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。

要将存档部署到红帽构建的 Keycloak 中，请将其添加到红帽构建的 Keycloak 的 providers/ 目录中，并在服务器已在运行时重新启动服务器。

3.6. 主题的其他资源

- 有关创新，请参阅 红帽构建的 Keycloak 中捆绑的默认主题。

- Red Hat build of Keycloak Quickstarts Repository - Quickstart 仓库的目录 扩展 包含一些主题示例，也可以用作鼓励。=== Theme selector

默认情况下，使用为域配置的主题，但客户端能够覆盖登录主题除外。可以通过 Theme Selector SPI 来更改此行为。

例如，这可用于为桌面和移动设备选择不同的主题，例如：

要创建自定义主题选择器，您需要实现 ThemeSelectorProviderFactory 和 ThemeSelectorProvider。

3.7. THEME 资源

在 Red Hat build of Keycloak 中实施自定义供应商时，可能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/templates 资源中创建一个 JAR，并在me - resources/ messages 中的消息捆绑包中。

如果您需要更灵活地加载通过 ThemeResourceSPI 实现的模板和资源。通过实施 ThemeResourceProviderFactory 和 ThemeResourceProvider，您可以精确决定如何加载模板和资源。

3.8. LOCALE 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，英语是默认的语言。

启用国际化后，区域设置会根据 服务器管理指南 中描述的逻辑来解决。

通过实施 LocaleSelectorProvider 和 LocaleSelectorProviderFactory，可以通过 LocaleSelectorSPI 更改此行为。

LocaleSelectorProvider 接口具有单一方法 resolveLocale，它必须返回带有 RealmModel 和 nullable UserModel 的区域设置。实际请求可从 KeycloakSession#getContext 方法获得。

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，自定义实施可能会扩展默认提供程序，覆盖其 getAcceptLanguageHeaderLocale，并返回 null 值。因此，区域设置选择将回退到 realm 的默认语言。

3.9. LOCALE 选择器的其他资源

- 有关创建和部署自定义供应商的更多详细信息，请参阅 服务提供商接口。

第 4 章 身份代理 API

红帽构建的 Keycloak 可将身份验证委托给父 IDP 以进行登录。其中一个典型的例子是您希望用户能够通过 Facebook 或 Google 等社交提供商登录。您还可以将现有帐户链接到代理 IDP。本节论述了应用程序可以使用的一些 API 与身份代理相关。

4.1. 检索外部 IDP 令牌

红帽构建的 Keycloak 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，您可以在 IDP 的设置页面中使用 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 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，您需要发送请求，如下所示：

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

应用程序必须通过红帽构建的 Keycloak 进行身份验证，并收到访问令牌。此访问令牌需要设置 代理客户端级别的角色 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具有该角色。在这种情况下，由于您在红帽构建的 Keycloak 中访问受保护的服务，您需要在用户身份验证过程中发送由红帽构建 Keycloak 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 开关来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，也可以使用客户端发起的帐户链接 API 来重新建立。

4.2. 客户端启动的帐户链接

有些应用程序希望与 Facebook 等社交供应商集成，但不想通过这些社交提供商登录。红帽构建的 Keycloak 提供了一个基于浏览器的 API，应用程序可以使用它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。帐户链接只能由 OIDC 应用程序启动。

其工作方式在于，应用程序将用户的浏览器转发到红帽构建的 Keycloak 服务器上的 URL，要求它希望将用户帐户链接到特定的外部提供程序（例如，Facebook）。服务器启动具有外部提供程序的登录。外部提供程序的浏览器日志，并重定向到服务器。服务器建立链接，并通过确认重新重定向到应用程序。

客户端应用程序必须满足一些前提条件，然后才能启动此协议：

- 必须在管理控制台中为用户域配置和启用所需的身份提供程序。
- 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。
- 应用必须被授予其访问令牌中这些角色的范围
- 应用必须有权访问其访问令牌，因为它需要信息来生成重定向 URL。

若要启动登录，应用必须结构一个 URL，并将用户的浏览器重定向到此 URL。URL 类似如下：

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

以下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。当您在管理控制台中将应用程序注册为客户端时，必须指定此客户端 ID。

redirect_uri

这是您要在帐户链接建立后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，它必须与您在管理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

这是以 Base64 URL 编码的哈希。此哈希由 Base64 URL 编码为 nonce + token.getSessionState （） + token.getIssuedFor （） + 供应商 生成。令牌变量从 OIDC 访问令

牌获取。基本上，您是随机的非ce、用户会话 ID、客户端 ID 和您要访问的身份提供程序别名。

以下是生成 URL 以建立帐户链接的 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("/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？这样做是为了保证 auth 服务器知道客户端应用程序启动请求，而其他恶意应用程序不会随机要求用户帐户链接到特定提供程序。auth 服务器首先通过检查登录时设置的 SSO cookie 来检查用户是否已登录。然后，它将尝试根据当前登录重新生成哈希，并将其与应用程序发送的哈希匹配。

在帐户被链接后，身份验证服务器将重新重定向到 redirect_uri。如果为链接请求提供服务，则身份验证服务器可能会或不重新重定向到 redirect_uri。浏览器可能只位于错误页面，而不是重新重定向到应用。如果存在错误条件，并且 auth 服务器足够安全地重定向到客户端应用程序，则会将额外的 错误 查询参数附加到 redirect_uri。

警告

虽然此 API 保证应用程序启动了请求，但它不会阻止 CSRF 对此操作进行攻击。应用程序仍然负责保护其自身对 CSRF 攻击目标。

4.2.1. 刷新外部令牌

如果您使用通过登录到提供程序（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 5 章 服务供应商接口(SPI)

Red Hat build of Keycloak 旨在覆盖大多数用例，而无需自定义代码，但我们也希望它可以被自定义。为了实现此红帽构建的 Keycloak，有许多服务提供商接口(SPI)，您可以自行实施供应商。

5.1. 实施 SPI

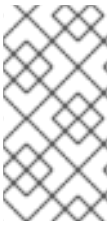
要实施 SPI，您需要实施其 ProviderFactory 和 Provider 接口。您还需要创建服务配置文件。

例如，要实现 Theme Selector SPI，您需要实现 ThemeSelectorProviderFactory 和 ThemeSelectorProvider，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";
    }
}
```

建议您的供应商工厂实现通过方法 getId （） 返回唯一 id。但是，在 Overriding providers 部分所述，这个规则可能存在一些例外。



注意

红帽构建的 Keycloak 创建一个供应商工厂实例，以便可以存储多个请求的状态。通过调用每个请求的工厂上的 create 来创建供应商实例，因此这些实例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
```

要配置您的提供程序，请参阅配置提供程序 章节。

例如，要配置供应商，您可以设置选项，如下所示：

```
bin/kc.[sh|bat] --spi-theme-selector-my-theme-selector-enabled=true --spi-theme-selector-my-theme-selector-theme=my-theme
```

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供应商。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();
    }
}
```

5.1.1. 覆盖内置供应商

如前文所述，建议您使用 ProviderFactory 实现的唯一 ID。但是，同时，覆盖红帽构建的 Keycloak 内置供应商会很有用。建议的做法是使用唯一 ID 的 ProviderFactory 实施，然后实例用于设置 配置提供程序一章中所述的默认提供程序。另一方面，这可能无法始终实现。

例如，当您需要对默认 OpenID Connect 协议进行一些自定义，并希望覆盖默认的 OIDCLoginProtocolFactory 实现的 Keycloak 实现时，您需要保留相同的 providerId。例如，OIDC 协议众所周知的端点，各种因素依赖于开放连接协议工厂的 ID。

在这种情况下，强烈建议您实施自定义实施的方法 order （），并确保其顺序高于内置实施。

```
public class CustomOIDCLoginProtocolFactory extends OIDCLoginProtocolFactory {

    // Some customizations here

    @Override
    public int order() {
        return 1;
    }
}
```

如果有多个具有相同供应商 ID 的实现，则红帽构建 Keycloak 运行时只会使用具有最高顺序的那一个。

5.1.2. 在控制台中显示您的 SPI 实施的信息

有时，向红帽构建的 Keycloak 管理员显示有关供应商的额外信息会很有用。您可以显示供应商构建时间信息（例如，当前安装的自定义供应商版本）、提供商的当前配置（例如您的供应商与之通信的远程系统的 url）或一些操作信息（来自您的供应商的远程系统平均响应时间）。红帽构建的 Keycloak 管理控制台提供了 Server Info 页面来显示此类信息。

若要显示您的供应商的信息，最好在 ProviderFactory 中实施 org.keycloak.provider.ServerInfoAware ProviderFactory 接口。

来自上例的 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;
    }
}
```

5.2. 使用可用的供应商

在供应商实现中，您可以使用 Red Hat build of Keycloak 中提供的其他供应商。现有提供程序通常使用 KeycloakSession 来检索，该会话可供您的供应商使用，如 实施 SPI 部分所述。

红帽构建的 Keycloak 有两个供应商类型：

- 单实施供应商类型 - 红帽构建的 Keycloak 运行时中只能有一个活跃的特定供应商类型实现。

  例如 HostnameProvider 指定红帽构建的 Keycloak 使用的主机名，并为整个红帽构建的 Keycloak 服务器共享。因此，对于红帽构建的 Keycloak 服务器，只能有一个有效的供应商实现。如果服务器运行时有多个提供程序实现，则需要将它们指定为默认提供程序。

例如：

```
bin/kc.[sh|bat] build --spi-hostname-provider=default
```

用作 default -provider 值的值必须与特定供应商工厂实施的 ProviderFactory.getId （） 返回的 ID 匹配。在代码中，您可以获取 keycloakSession.getProvider (HostnameProvider.class)等提供程序

- 多个实现供应商类型 - Those 是供应商类型，它允许在红帽构建的 Keycloak 运行时中有多个实施并协同工作。

  例如，EventListener 供应商允许有多个可用的实现并注册，这意味着特定事件可以发送到所有监听程序(jboss-logging、sysout 等)。在代码中，您可以获取指定提供程序实例，例如 session.getProvider (EventListener.class, "jboss-logging")。您需要将 provider_id 指定为第二个参数，因为此供应商类型有多个实例，如上所述。

  供应商 ID 必须与特定供应商工厂实施的 ProviderFactory.getId （） 返回的 ID 匹配。某些供应商类型可以使用 ComponentModel 作为第二个参数以及一些参数（如 Authenticator）来检索，即使使用 KeycloakSessionFactory 即可检索。不建议以这种方式实施自己的提供程序，因为它可能会在以后被弃用。

5.3. 注册供应商实现

提供程序通过将 JAR 文件复制到 providers 目录中来注册到服务器。

如果您的供应商需要尚未由 Keycloak 提供的其他依赖项，请将它们复制到 供应商 目录中。

在注册新供应商或依赖项 Keycloak 后，需要使用非优化的启动或 kc.[sh|bat] build 命令重新构建。

注意

提供程序 JAR 不会在隔离的类加载器中加载，因此不要在提供程序 JAR 中包含与内置资源或类冲突的资源或类。特别是包含 application.properties 文件或覆盖 commons-lang3 依赖项，如果删除了供应商 JAR，则会导致 auto-build 失败。如果您包含冲突的类，您可能会在服务器开始日志中看到分割软件包警告。不幸的是，并非所有内置的 lib jar 都由分割软件包警告逻辑检查，因此您需要在捆绑或包含传输依赖项前检查 lib 目录 JAR。如果存在冲突，可以通过删除或重新打包现有类来解决。

如果您有冲突的资源文件，则不会发出警告。您应该确保 JAR 的资源文件具有包含该提供程序唯一内容的路径名称，或者您可以检查 "install root"/lib/lib/main 目录下的 JAR 内容中是否存在 some.file，如下所示：

```
find . -type f -name "*.jar" -exec unzip -l {} \; | grep some.file
```

如果发现服务器因为与已删除供应商 JAR 相关的 NoSuchFileException 错误而启动，则运行：

```
./kc.sh -Dquarkus.launch.rebuild=true
```

这将强制 Quarkus 重建类加载相关索引文件。从那里，您应能够执行非优化的启动或构建，而无需例外。

5.3.1. 禁用供应商

您可以通过将 provider 的 enabled 属性设置为 false 来禁用供应商。例如，要禁用 Infinispan 用户缓存提供程序，请使用：

```
bin/kc.[sh|bat] build --spi-user-cache-infinispan-enabled=false
```

5.4. JAVASCRIPT 供应商

红帽构建的 Keycloak 能够在运行时执行脚本，以便管理员能够自定义特定功能：

- 身份验证器
- JavaScript 策略

- OpenID Connect Protocol Mapper
- SAML 协议映射程序

5.4.1. 身份验证器

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

自定义 身份验证器 应至少提供身份验证 (..) 函数。您可以使用代码中的 javax.script.Bindings 脚本。

script

用于访问脚本 元数据的脚本模型

realm

RealmModel

user

当前 UserModel。请注意，如果在身份验证流中配置了脚本验证器时，用户就可以在另一个验证器成功下触发，并将该用户设置为身份验证会话。

会话

活跃的 KeycloakSession

authenticationSession

当前的 身份验证SessionModel

httpRequest

当前 org.jboss.resteasy.spi.HttpRequest

LOG

org.jboss.logging.Logger 范围为 ScriptBasedAuthenticator

注意

您可以从传递给 authentication (context) 操作(context) 函数的上下文参数中提取额外的上下文信息。

```
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();
}
```

5.4.1.1. 在什么位置添加脚本验证器

可以使用脚本验证器是在身份验证结束时进行一些检查。请注意，如果您希望脚本验证器总是被触发（即使在使用身份 Cookie 的 SSO 重新身份验证过程中为实例），您可能需要将它添加为身份验证流末尾的 REQUIRED，并将现有的验证器封装到单独的 REQUIRED 身份验证子流中。这需要是因为 REQUIRED 和 ALTERNATIVE 执行不应处于相同的级别。例如，身份验证流配置应如下所示：

```
- User-authentication-subflow REQUIRED
-- Cookie ALTERNATIVE
-- Identity-provider-redirect ALTERNATIVE
...
- Your-Script-Authenticator REQUIRED
```

5.4.2. OpenID Connect Protocol Mapper

OpenID Connect 协议映射程序脚本是 javascript 脚本，允许您更改 ID 令牌和/或访问令牌的内容。

您可以使用代码中的 javax.script.Bindings 脚本。

user

当前 UserModel

realm

RealmModel

token

当前 IDToken。只有在为 ID 令牌配置了映射程序时，它才可用。

tokenResponse

当前 AccessTokenResponse。只有在为 Access 令牌配置了映射程序时，它才可用。

userSession

活跃的 UserSessionModel

keycloakSession

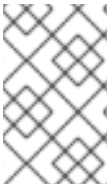
活跃的 KeycloakSession

脚本的导出将用作令牌声明的值。

```
// prints can be used to log information for debug purpose.
print("STARTING CUSTOM MAPPER");

var inputRequest = keycloakSession.getContext().getHttpRequest();
var params = inputRequest.getDecodedFormParameters();
var output = params.getFirst("user_input");
exports = output;
```

以上脚本允许从授权请求中检索 user_input。这可用于映射程序中配置的 Token Claim Name。

5.4.3. 使用要部署的脚本创建一个 JAR



注意

JAR 文件是带有 .jar 扩展名的常规 ZIP 文件。

为了使您的脚本可供红帽构建的 Keycloak 使用，您需要将它们部署到服务器。为此，您应该创建一个具有以下结构的 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，提供有关您要部署脚本的元数据信息。它是具有以下结构的 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"
    }
  ]
}
```

此文件应该引用您要部署的不同类型的脚本供应商：

- **身份验证器**

  对于 **OpenID Connect 脚本身份验证器。同一 JAR 文件中可以有一个或多个验证器**

- **policies**

对于使用红帽构建的 Keycloak 授权服务时的 JavaScript 策略。同一 JAR 文件中可以有一个或多个策略

- **Mappers**

  对于 OpenID Connect 脚本协议映射程序。同一 JAR 文件中可以有一个或多个映射程序

- **saml-mappers**

  对于 SAML 脚本协议映射程序。同一 JAR 文件中可以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 中有一个对应的条目，将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**

  用于通过红帽构建的 Keycloak 管理控制台显示脚本的友好名称。如果没有提供，则使用脚本文件的名称

- **description**

  更好的可选文本，它描述了脚本文件的预期

- **fileName**

  脚本文件的名称。此属性 是必需的，应映射到 JAR 中的文件。

5.4.4. 部署脚本 JAR

您有一个带有描述符和您要部署的脚本的 JAR 文件后，您只需要将 JAR 复制到 Keycloak 提供程序/目录的红帽构建中，然后运行 bin/ kc.[sh|bat] 构建。请注意，您还需要启用 脚本功能。

5.5. 可用的 SPI

如果要在运行时查看所有可用 SPI 列表，您可以检查管理控制台中的 Server Info 页面，如 Admin Console 部分所述。

第 6 章 USER STORAGE SPI

您可以使用 User Storage SPI 为红帽构建的 Keycloak 编写扩展，以连接到外部用户数据库和凭证存储。内置的 LDAP 和 ActiveDirectory 支持是此 SPI 的实施。红帽构建的 Keycloak 使用其本地数据库创建、更新和查找用户并验证凭证。虽然组织通常具有现有的外部专有用户数据库，它们无法迁移到红帽构建的 Keycloak 数据模型。对于这些情况，应用程序开发人员可编写 User Storage SPI 的实现，以桥接外部用户存储，以及红帽构建的 Keycloak 的内部用户对象模型，以管理它们。

当红帽构建的 Keycloak 运行时需要查找用户（如用户登录时），它会执行多个步骤来定位用户。首先查看用户是否位于用户缓存中；如果用户找到了用户，则使用内存表示法。然后，它会在红帽构建的 Keycloak 本地数据库中查找用户。如果没有找到用户，它将通过 User Storage SPI 供应商实现循环来执行用户查询，直到其中一个用户返回用户运行时正在查找。供应商查询外部用户存储，并将用户的外部数据表示映射到红帽构建的 Keycloak 用户 metamodel。

用户存储 SPI 供应商实现也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，或者一次性执行许多用户的批量更新。它取决于外部存储的功能。

用户存储 SPI 提供程序实现以类似（通常）Jakarta EE 组件打包和部署。默认情况下不启用它们，而是必须在管理控制台的 User Federation 选项卡下为每个域启用和配置。

警告

如果您的用户供应商实现使用一些用户属性作为链接/建立用户身份的元数据属性，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，内置的红帽构建的 Keycloak LDAP 供应商用于将用户的 ID 存储在 LDAP 服务器端。请参阅 Threat 模型缓解章节中 的详细信息。

红帽构建的 Keycloak Quickstarts 仓库有两个示例项目。每个快速入门都有一个 README 文件，其中包含有关如何构建、部署和测试示例项目的说明。下表提供了可用用户存储 SPI 快速入门的简短描述：

表 6.1. User Storage SPI Quickstarts

Name	描述
user-storage-jpa	演示使用 EJB 和 JPA 实施用户存储提供程序。

Name	描述
user-storage-simple	演示使用包含用户名/密码密钥对的简单属性文件实施用户存储提供程序。

6.1. 供应商接口

在构建用户存储 SPI 的实现时，您必须定义供应商类和供应商工厂。供应商类实例由供应商工厂为每个事务创建。供应商类执行用户查找和其他用户操作的所有繁重。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) {
```

```
    }

}
```

您或许正在认为 UserStorageProvider 接口是相当稀疏的？您将在本章的后续部分中看到，您的供应商类可以实施其他混合接口来支持用户集成的机位。

每个事务都会创建 UserStorageProvider 实例。事务完成后，会调用 UserStorageProvider.close() 方法，然后收集实例。实例由供应商工厂创建。provider factories 实现 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，供应商工厂类必须将 concrete 供应商类指定为模板参数。这是必须因为运行时将内省此类来扫描其功能（它实施的其他接口）。例如，如果您的供应商类命名为 FileProvider，则工厂类应类似如下：

```
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。当您要为特定域启用供应商时，此 id 将显示在管理控制台的 User Federation 页面中。

create （） 方法负责分配供应商类的实例。它采用 org.keycloak.models.KeycloakSession 参数。此对象可用于查找其他信息和元数据，并提供运行时内各种其他组件的访问。ComponentModel 参数代表如何在特定域中启用和配置供应商。它包含已启用提供程序的实例 ID，以及通过管理控制台启用时您可以为它指定的任何配置。

UserStorageProviderFactory 还有其他功能，我们将在本章的后续部分中接管。

6.2. 供应商功能接口

如果您仔细检查了 UserStorageProvider 接口，您可能会注意到没有定义任何用于查找或管理用户的方法。这些方法实际上在其他功能 *接口上* 定义，具体取决于外部用户存储可以提供和执行的功能范围。例如，一些外部存储是只读的，只能执行简单的查询和凭证验证。您只需要为能够的功能实施 *功能接口*。您可以实现这些接口：

SPI	描述
org.keycloak.storage.user.UserLookupProvider	如果您希望能够使用来自此外部存储的用户登录，则需要这个接口。大多数（所有？）提供程序都实施此接口。
org.keycloak.storage.user.UserQueryMethodsProvider	定义用于定位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，您必须实现此接口。
org.keycloak.storage.user.UserCountMethodsProvider	如果您的供应商支持计数查询，请实施此接口。
org.keycloak.storage.user.UserQueryProvider	这个接口是 UserQueryMethodsProvider 和 UserCountMethodsProvider 的组合功能。
org.keycloak.storage.user.UserRegistrationProvider	如果您的供应商支持添加和删除用户，请实施此接口。
org.keycloak.storage.user.UserBulkUpdateProvider	如果您的供应商支持一组用户的批量更新，则实施此接口。

SPI	描述
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，如果您的供应商能够验证密码，则实施此接口）。
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，请实施此接口。

6.3. 模型接口

功能 接口 中定义的大多数方法都返回或以用户的表示形式传递。这些表示由 org.keycloak.models.UserModel 接口定义。需要应用程序开发人员才能实施此接口。它提供了外部用户存储和红帽构建的 Keycloak 使用的用户 metamodel 之间的映射。

```
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 实现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。

org.keycloak.models 软件包中还有其他模型类，它们代表红帽构建的 Keycloak metamodel: RealmModel、RoleModel、GroupModel 和 ClientModel 的其他部分。

6.3.1. 存储 Ids

UserModel 的一个重要方法是 getId （） 方法。在实施 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
```

红帽构建的 Keycloak 运行时通常必须通过用户 ID 查找用户。用户 id 包含足够信息，因此运行时不必查询系统中的每个 UserStorageProvider 来查找用户。

组件 ID 是从 ComponentModel.getId （） 返回的 id。组件Model 在创建供应商类时作为参数传递，以便您可以从那里获取它。外部 id 是您的供应商类需要在外部存储中查找用户的信息。这通常是一个用户名或 uid。例如，它可能类似如下：

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

当运行时按 id 进行查找时，id 会被解析，以获取组件 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。提供商再次解析 id 来获取外部 ID，它将用来在外部用户存储中查找用户。

6.4. 打包和部署

为了让红帽构建的 Keycloak 识别提供程序，您需要向 JAR 添加一个文件： META-INF/services/org.keycloak.storage.UserStorageProviderFactory。此文件必须包含 UserStorageProviderFactory 实施的完全限定类名称列表：

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

要部署此 jar，将其复制到 provider/ 目录中，然后运行 bin/kc.[sh|bat] build。

6.5. 简单只读查找示例

为了说明实施用户存储 SPI 的基础知识，我们来浏览一个简单的示例。在本章中，您将看到一个简单 UserStorageProvider 的实现，它会在一个简单的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。该提供程序将能够按 ID 和用户名查找用户，并能够验证密码。源自此提供程序的用户将只读。

6.5.1. 供应商类

首先，我们将进入的事情是 UserStorageProvider 类。

```
public class PropertyFileUserStorageProvider implements
      UserStorageProvider,
```

```
        UserLookupProvider,
        CredentialInputValidator,
        CredentialInputUpdater
{
...
}
```

我们的供应商类 PropertyFileUserStorageProvider 实现了许多接口。它实现 UserStorageProvider，因为这是 SPI 的基本要求。它实现了 UserLookupProvider 接口，因为我们希望能够使用此提供程序存储的用户登录。它实现了 CredentialInputValidator 接口，因为我们希望能够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，因为当用户尝试更新其密码时，我们希望发出错误条件。

```
    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、ComponentModel 和 属性文件的引用。我们稍后将用到所有这些产品。另请注意，有加载的用户映射。每当我们发现某个用户，我们将将其存储在此地图中，以便我们避免在同一事务中再次重新创建。这是遵循许多提供商需要执行此操作的良好做法（即，与 JPA 集成的任何提供商）。请记住，每个事务都会创建提供程序类实例，并在事务完成后关闭。

6.5.1.1. UserLookupProvider 实现

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

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

当用户登录时，Red Hat build of Keycloak 登录页会调用 getUserByUsername （） 方法。在我们的实施中，我们首先检查 loadedUsers 映射，以查看用户是否已在此事务中载入了用户。如果没有加载，我们将查看用户名的属性文件。如果存在，我们创建一个 UserModel 的实现，将其存储在 loadUsers 中，以备将来参考，并返回此实例。

createAdapter （） 方法使用帮助程序类 org.keycloak.storage.adapter.AbstractUserAdapter。这为 UserModel 提供了基础实施。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 方法 AbstractUserAdapter 都会返回 null 或空集合。但是，返回角色和组映射的方法将返回为每个用户配置域的默认角色和组。每个组 AbstractUserAdapter 都会抛出 org.keycloak.storage.ReadOnlyException。因此，如果您试图在管理控制台中修改用户，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，以获取嵌入在 id 参数中的用户名。然后，方法将委托为 getUserByUsername （）。

电子邮件不会被存储，因此 getUserByEmail （） 方法返回 null。

6.5.1.2. CredentialInputValidator 实现

接下来，让我们来看看 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，以确定是否为用户配置了特定的凭证类型。这个方法检查是否为用户设置了密码。

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

isValid （） 方法负责验证密码。CredentialInput 参数实际上是所有凭证类型的一个抽象接口。我们确保支持凭证类型，并且也是 UserCredentialModel 的实例。当用户通过登录页面登录时，密码输入的纯文本将放入 UserCredentialModel 的实例。isValid （） 方法针对存储在属性文件中的纯文本密码检查这个值。返回值 true 表示密码有效。

6.5.1.3. CredentialInputUpdater 实现

如前所述，我们实现 CredentialInputUpdater 接口的唯一原因是，禁止修改用户密码。我们必须这样做的原因是，否则运行时允许在红帽构建的 Keycloak 本地存储中覆盖密码。本章稍后将对此进行更多讨论。

```
    @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 Stream<String> getDisableableCredentialTypesStream(RealmModel realm,
UserModel user) {
        return Stream.empty();
    }
```

updateCredential （） 方法只检查凭证类型是否为 password。如果是，则会抛出 ReadOnlyException。

6.5.2. 供应商工厂实施

现在，供应商类已完成，我们现在将注意供应商工厂类。

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

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

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

首先要注意的是，在实施 UserStorageProviderFactory 类时，您必须传递 concrete provider 类实现作为模板参数。此处我们指定之前定义的供应商类： PropertyFileUserStorageProvider。

警告

如果没有指定 template 参数，您的供应商将无法正常工作。运行时执行类内省，以确定提供程序实施 *的能力接口*。

getId （） 方法标识运行时中的工厂，当您要为域启用用户存储供应商时，也会是 admin 控制台中显示的字符串。

6.5.2.1. 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 接口具有可实现的可选 init （） 方法。当红帽构建的 Keycloak 引导时，只会为每个供应商工厂创建一个实例。另外，引导时都会调用 init （） 方法。还有一个 postInit （） 方法，也可以实施。在调用每个工厂的 init （） 方法后，会调用其 postInit （） 方法。

在我们的 init （） 方法实现中，我们从 classpath 中找到包含我们用户声明的属性文件。然后，使用存储了用户名和密码组合来加载 properties 字段。

Config.Scope 参数是通过服务器配置的工厂配置。

例如，使用以下参数运行服务器：

```
kc.[sh|bat] start --spi-storage-readonly-property-file-path=/other-users.properties
```

我们可以指定用户属性文件的 classpath，而不是对其进行硬编码。然后，您可以在 PropertyFileUserStorageProviderFactory.init （） 中检索配置：

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

```
    ...
}
```

6.5.2.2. 创建方法

创建供应商工厂的最后一步是 create （） 方法。

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

我们只需分配 PropertyFileUserStorageProvider 类。此创建方法将为每个事务调用一次。

6.5.3. 打包和部署

用于供应商实现的类文件应放在 jar 中。您还必须在 META-INF/services/org.keycloak.storage.UserProviderFactory 文件中声明供应商工厂类。

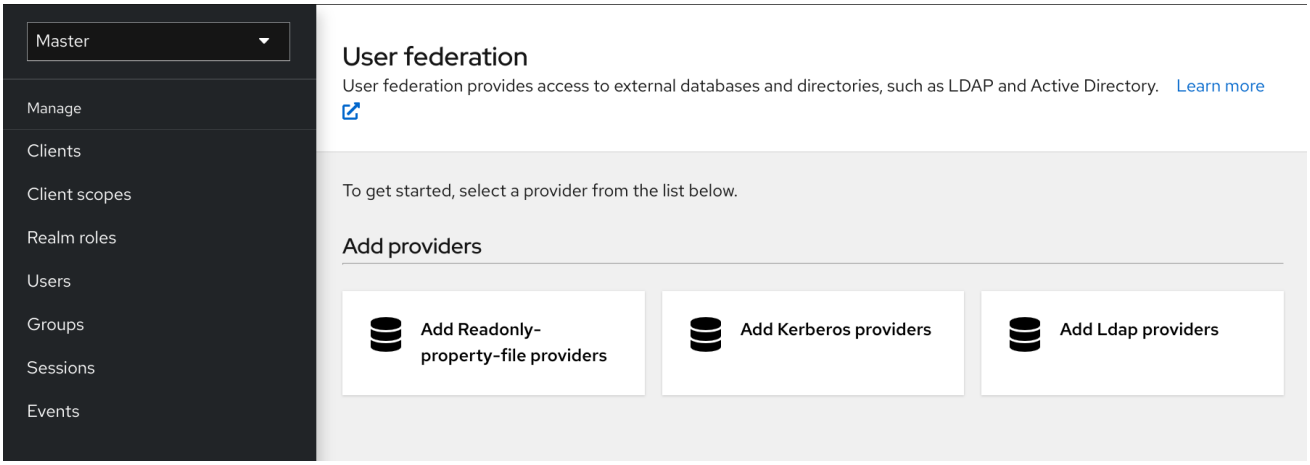
```
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

要部署此 jar，将其复制到 provider/ 目录中，然后运行 bin/kc.[sh|bat] build。

6.5.4. 在管理门户中启用提供程序

您可以在管理控制台的 User Federation 页面中为每个域启用用户存储供应商。

用户 Federation



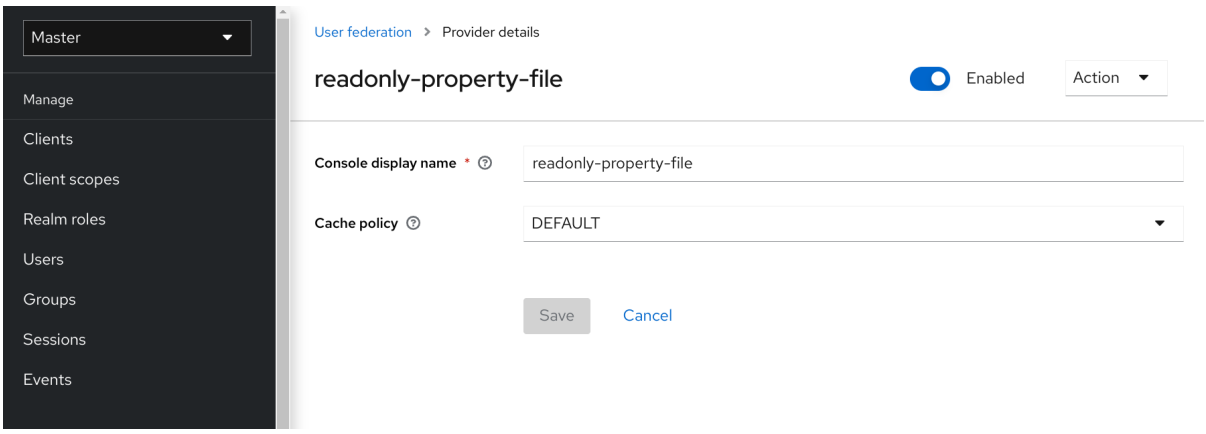
流程

1. **从列表中选择刚才创建的提供程序： readonly-property-file。**

   这时将显示我们供应商的配置页面。

2. 单击 **Save，因为我们无需配置。**

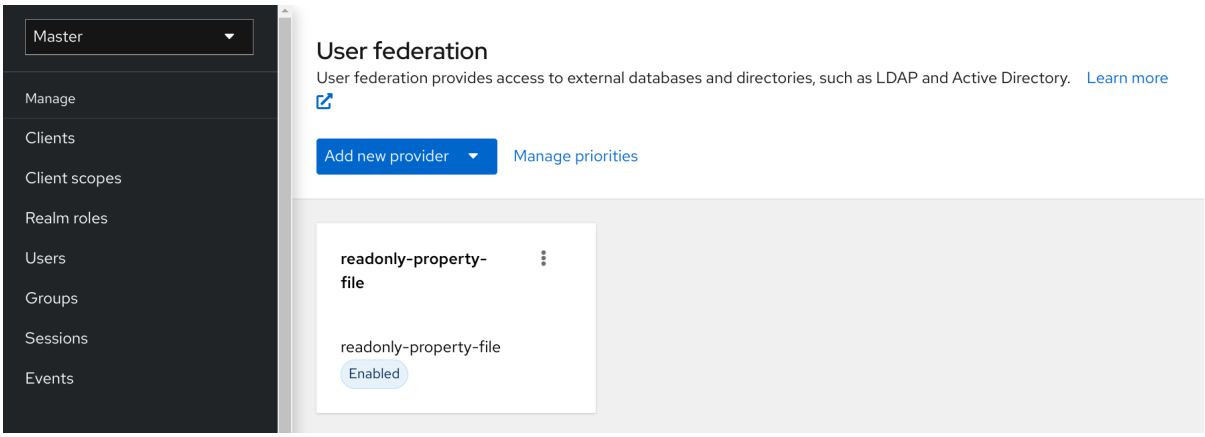
   配置的供应商



3. **返回到主 User Federation 页面**

   现在，您会看到列出的供应商。

   用户 Federation



现在，您将能够使用 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。

6.6. 配置技术

我们的 PropertyFileUserStorageProvider 示例是一个点子。它被硬编码为嵌入在提供程序的 jar 中的属性文件，它不特别有用。我们希望使此文件的位置可以为每个提供程序的实例进行配置。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。我们还希望在管理控制台 UI 中执行此配置。

UserStorageProviderFactory 具有处理供应商配置的其他方法。您可以描述您要为每个供应商配置的变量，管理控制台会自动呈现通用输入页面来收集此配置。实施时，回调方法也会在保存前验证配置，当首次创建提供程序并在更新时。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) {

    }
```

component Factory.getConfigProperties （） 方法返回 org.keycloak.provider.ProviderConfigProperty 实例列表。这些实例声明所需的元数据，以呈现和存储提供程序的每个配置变量。

6.6.1. 配置示例

我们扩展了 PropertyFileUserStorageProviderFactory 示例，允许您将供应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 类是用于创建配置属性列表的绝佳帮助程序类。在这里，我们指定了一个名为 path 的变量，它是一个 String 类型。在此提供程序的 Admin Console 配置页面中，此配置变量被标记为 Path，默认值为 ${jboss.server.config.dir}/example-users.properties。当您将鼠标悬停在此配置选项的工具提示上时，它会显示帮助文本、属性文件 的路径。

接下来我们要做的是验证该文件是否存在于磁盘上。我们不想在域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");
        }
    }
```

validateConfiguration （） 方法提供 组件Model 中的配置变量，以验证磁盘上是否存在该文件。请注意，使用 org.keycloak.common.util.EnvUtil.replace （） 方法。使用这个方法，包含 ${} 的任何字符串都将用系统属性值替换该值。${jboss.server.config.dir} 字符串对应于服务器的 conf/ 目录，这对于本例来说非常有用。

接下来我们必须做的是删除旧的 init （） 方法。我们这样做，因为用户属性文件会为每个提供程序实例是唯一的。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，这种逻辑效率低下，因为每个事务从磁盘读取整个用户属性文件，但希望以简单的方式进行演示，如何在配置变量中 hook。

6.6.2. 在管理控制台中配置提供程序

现在，启用了配置，您可以在管理控制台中配置供应商时设置 path 变量。

6.7. 添加/删除用户和查询功能接口

我们还没有通过我们的例子完成某个操作，即允许其添加和删除用户或更改密码。在我们的示例中定义的用户在管理控制台中不可查询或查看。要添加这些增强功能，我们的示例供应商必须实施 UserQueryMethodsProvider （或 UserQueryProvider）和 UserRegistrationProvider 接口。

6.7.1. 实施 UserRegistrationProvider

使用此流程实施添加和从特定存储中删除用户，我们必须首先将属性文件保存到磁盘。

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，我们将属性映射的密码值设置为 UNSET_PASSWORD。我们这样做，因为我们在属性值中没有属性的 null 值。我们还必须修改 CredentialInputValidator 方法来反映这一点。

如果供应商实现了 UserRegistrationProvider 接口，则会调用 addUser （） 方法。如果您的供应商有一个配置开关来关闭添加用户，从此方法返回 null 将跳过该提供程序并调用下一个提供程序。

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());
    }
```

由于我们现在可以保存我们的属性文件，因此允许密码更新也很有意义。

PropertyFileUserStorageProvider

```
    @Override
    public boolean updateCredential(RealmModel realm, UserModel user, CredentialInput input)
{
        if (!(input instanceof UserCredentialModel)) return false;
```

```
        if (!input.getType().equals(PasswordCredentialModel.TYPE)) 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(PasswordCredentialModel.TYPE)) return;
        synchronized (properties) {
            properties.setProperty(user.getUsername(), UNSET_PASSWORD);
            save();
        }

    }

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

    static {
        disableableTypes.add(PasswordCredentialModel.TYPE);
    }

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

        return disableableTypes.stream();
    }
```

实施这些方法后，您现在可以在管理控制台中更改和禁用用户的密码。

6.7.2. 实施 UserQueryProvider

UserQueryProvider 是 UserQueryMethodsProvider 和 UserCountMethodsProvider 的组合。如果没有实现 UserQueryMethodsProvider，则管理控制台将无法查看和管理我们示例供应商加载的用户。让我们来看看这个接口。

PropertyFileUserStorageProvider

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

    @Override
    public Stream<UserModel> searchForUserStream(RealmModel realm, String search, Integer
firstResult, Integer maxResults) {
        Predicate<String> predicate = "*".equals(search) ? username -> true : username ->
username.contains(search);
        return properties.keySet().stream()
                .map(String.class::cast)
                .filter(predicate)
                .skip(firstResult)
                .map(username -> getUserByUsername(realm, username))
                .limit(maxResults);
    }
```

searchForUserStream （） 的第一个声明采用 String 参数。在本例中，参数代表您要搜索的用户名。此字符串可以是子字符串，它解释了执行搜索时 String.contains （） 方法的选择。请注意，使用 * 表示请求所有用户的列表。该方法迭代属性文件的关键集合，委派为 getUserByUsername （） 来加载用户。请注意，我们根据 firstResult 和 maxResults 参数索引此调用。如果您的外部存储不支持分页，则必须执行类似的逻辑。

PropertyFileUserStorageProvider

```
    @Override
    public Stream<UserModel> searchForUserStream(RealmModel realm, Map<String, String>
params, Integer firstResult, Integer maxResults) {
        // only support searching by username
        String usernameSearchString = params.get("username");
        if (usernameSearchString != null)
            return searchForUserStream(realm, usernameSearchString, firstResult, maxResults);
```

```
        // if we are not searching by username, return all users
        return searchForUserStream(realm, "*", firstResult, maxResults);
    }
```

使用 Map 参数的 searchForUserStream （） 方法可以根据名字、姓氏、用户名和电子邮件搜索用户。仅存储用户名，因此搜索仅基于用户名，除了 Map 参数不包含 username 属性时除外。在这种情况下，所有用户都会被返回。在这种情况下，使用 searchForUserStream (realm, search, firstResult, maxResults)。

PropertyFileUserStorageProvider

```
    @Override
    public Stream<UserModel> getGroupMembersStream(RealmModel realm, GroupModel
group, Integer firstResult, Integer maxResults) {
        return Stream.empty();
    }

    @Override
    public Stream<UserModel> searchForUserByUserAttributeStream(RealmModel realm, String
attrName, String attrValue) {
        return Stream.empty();
    }
```

组或属性不会被存储，因此其他方法会返回空流。

6.8. 增加外部存储

PropertyFileUserStorageProvider 示例实际上有限。虽然我们将能够使用存储在属性文件中的用户登录，但我们无法执行许多其他操作。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，则无法向这些用户添加额外的角色映射。您还可以修改或添加其他重要属性，如电子邮件、名字和姓氏。

对于这些类型的情况，红帽构建的 Keycloak 允许您通过在红帽构建的 Keycloak 数据库中存储额外信息来增加外部存储。这称为联邦用户存储，并封装在 org.keycloak.storage.federated.UserFederatedStorageProvider 类中。

UserFederatedStorageProvider

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

public interface UserFederatedStorageProvider extends Provider,
        UserAttributeFederatedStorage,
        UserBrokerLinkFederatedStorage,
        UserConsentFederatedStorage,
        UserNotBeforeFederatedStorage,
        UserGroupMembershipFederatedStorage,
        UserRequiredActionsFederatedStorage,
        UserRoleMappingsFederatedStorage,
        UserFederatedUserCredentialStore {
    ...

}
```

UserFederatedStorageProvider 实例在 UserStorageUtil.userFederatedStorage (KeycloakSession) 方法上提供。它有各种不同的方法来存储属性、组和角色映射、不同的凭证类型和所需操作。如果您的外部存储的 datamodel 无法支持完整的红帽构建的 Keycloak 功能集，则该服务可能会填补差距。

红帽构建的 Keycloak 附带了帮助程序类 org.keycloak.storage.adapter.AbstractUserAdapterFederatedStorage，它将把每个单个 UserModel 方法委托给用户联邦存储。覆盖您需要覆盖的方法，以委托给外部存储表示形式。强烈建议您阅读此类的 javadoc，因为它有较小的保护方法，您可能希望覆盖。特别是组成员资格和角色映射。

6.8.1. 8 月示例

在我们的 PropertyFileUserStorageProvider 示例中，我们只需要对供应商进行简单的更改才能使用 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();
            }
        }
    };
}
```

相反，我们定义 AbstractUserAdapterFederatedStorage 的匿名类实现。setUsername （） 方法更改属性文件并保存。

6.9. 导入实现策略

在实施用户存储供应商时，您可以采用另一个策略。除了使用用户联邦存储外，您可以在红帽构建的 Keycloak 内置用户数据库中在本地创建用户，并将外部存储的属性复制到这个本地副本。此方法有很多优点。

- 红帽构建的 Keycloak 基本上成为外部存储的持久性用户缓存。导入用户后，您将不再到达外部存储，从而关闭它。
- 如果您要以您的官方用户存储的形式迁移到红帽 Keycloak，并弃用旧的外部存储，您可以缓慢迁移应用程序以使用红帽构建的 Keycloak。当所有应用程序都迁移后，取消链接导入的用户，然后停用旧的传统外部存储。

使用导入策略有一些明显的缺点：

- 第一次查找用户将需要对红帽构建的 Keycloak 数据库进行多次更新。这可能会因为负载造成大量性能损失，并在红帽构建的 Keycloak 数据库上造成大量压力。用户联邦存储方法仅根据需要存储额外的数据，并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构建的 Keycloak 存储和外部存储同步。User Storage SPI 具有可以实施以支持同步的功能接口，但这可能会很快变得困难且混乱。

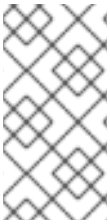
要实现导入策略，只需首先检查用户是否已在本地导入。如果返回本地用户，如果没有在本地创建用户并从外部存储导入数据。您还可以代理本地用户，以便大多数更改会自动同步。

这将是一个点子，但我们可以扩展我们的 PropertyFileUserStorageProvider 以采用这种方法。首先修改 createAdapter （） 方法。

PropertyFileUserStorageProvider

```
    protected UserModel createAdapter(RealmModel realm, String username) {
        UserModel local =
UserStoragePrivateUtil.userLocalStorage(session).getUserByUsername(realm, username);
        if (local == null) {
            local = UserStoragePrivateUtil.userLocalStorage(session).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);
            }
        };
    }
```

在这个方法中，我们调用 UserStoragePrivateUtil.userLocalStorage (session) 方法来获取对本地红帽构建的 Keycloak 用户存储的引用。我们会发现用户是否存储在本地（如果不是），我们将其添加到本地。不要设置本地用户的 id。让红帽构建的 Keycloak 会自动生成 id。另请注意，我们调用 UserModel.setFederationLink （），并传递供应商 组件 模型的 ID。这将设置提供程序和导入用户之间的链接。



注意

删除用户存储提供程序时，由它导入的任何用户也会被删除。这是调用 UserModel.setFederationLink （） 的一种目的。

要注意的是，如果本地用户链接，您的存储提供程序仍然被委派给，它从 CredentialInputValidator 和

CredentialInputUpdater 接口实现的方法。从验证或更新中返回 false 会导致红帽构建的 Keycloak 查看是否使用本地存储验证或更新。

另请注意，我们使用 org.keycloak.models.utils.UserModelDelegate 类代理本地用户。此类是 UserModel 的实现。每个方法只是委托给它所实例化的 UserModel。我们覆盖此委派类的 setUsername （） 方法，以自动与属性文件同步。对于您的供应商，您可以使用它来 *截获* 本地 UserModel 上的其他方法，以与外部存储执行同步。例如，get 方法可以确保本地存储保持同步。设置方法会使外部存储与本地存储保持同步。要注意的一件事是 getId （） 方法应始终返回您在本地创建用户时自动生成的 id。您不应该返回联邦 id，如其他非导入示例所示。

注意

如果您的供应商正在实现 UserRegistrationProvider 接口，则您的 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，在从本地存储中删除之前，将调用 removeUser （）。

6.9.1. ImportedUserValidation 接口

如果您在本章前面文中记住，我们将讨论查询用户如何工作。如果用户找到，则首先查询本地存储，则查询结束。因为我们需要代理本地用户 Model，因此我们可以使用户名保持同步。当链接的本地用户从本地数据库加载时，User Storage SPI 有一个回调。

```
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);
}
```

每当加载链接的本地用户时，如果用户存储供应商类实现了这个接口，则调用 validate （） 方法。在这里，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。您还可以选择进行检查来查看用户是否仍然存在于外部存储中。如果 validate （） 返回 null，则将从数据库中删除本地用户。

6.9.2. ImportSynchronization 接口

通过导入策略，您可以看到本地用户副本可以与外部存储不同步。例如，用户可能已从外部存储中删除。User Storage SPI 有一个额外的接口来处理这

个、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);
}
```

这个接口由供应商工厂实现。此接口由供应商工厂实现后，提供程序的管理控制台管理页面会显示附加选项。您可以通过单击按钮手动强制进行同步。这会调用 ImportSynchronization.sync （） 方法。另外，还会显示其他配置选项，供您自动调度同步。自动同步调用 syncSince （） 方法。

6.10. 用户缓存

当用户对象通过 ID、用户名或电子邮件查询加载时，会缓存它。当用户对象被缓存时，它会迭代整个 UserModel 接口，并将此信息拉取到本地内存缓存中。在集群中，此缓存仍然是本地的，但它成为无效的缓存。修改用户对象时，它会被驱除。此驱除事件传播到整个集群，以便其他节点的用户缓存也无效。

6.10.1. 管理用户缓存

您可以通过调用 KeycloakSession.getProvider (UserCache.class) 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、特定域中的用户或整个缓存。

6.10.2. OnUserCache 回调接口

您可能需要缓存特定于您的供应商实现的额外信息。每当用户被缓存时，User Storage SPI 有一个回调： org.keycloak.models.cache.OnUserCache。

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

如果需要此回调，您的供应商类应实施此接口。UserModel delegate 参数是您的供应商返回的 UserModel 实例。CachedUserModel 是一个扩展的 UserModel 接口。这是在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();
}
```

此 CachedUserModel 接口允许您从缓存中驱除用户，并获取供应商 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证，则需要实施 OnUserCache，并使用 getCachedWith （） 方法缓存用户的凭证。

6.10.3. 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。

6.11. 利用 JAKARTA EE

从版本 20 开始，Keycloak 仅适用于 Quarkus。与 WildFly 不同，Quarkus 不是应用服务器。详情请查看 https://www.keycloak.org/migration/migrating-to-quarkus#_quarkus_is_not_an_application_server。

因此，用户存储提供程序无法打包在任何 Jakarta EE 组件中，或者像 Keycloak 在之前的版本中通过 WildFly 运行时一样。

供应商实现必须是实现合适的 User Storage SPI 接口的普通 java 对象，如上一节中所述。如本迁移指南中所述，必须打包和部署它们：

- https://www.keycloak.org/migration/migrating-to-quarkus#_migrating_custom_providers

您仍然可以实现自定义 UserStorageProvider 类，该类可通过 JPA Entity Manager 集成外部数据库，如下例所示：

- https://github.com/redhat-developer/rhbk-quickstarts/tree/24.x/extension/user-storage-jpa

不支持 CDI。

6.12. 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，因此您将使用通用 API 管理您的供应商。

REST 组件 API 在您的域 admin 资源下存在。

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

我们将仅显示与 Java 客户端进行这个 REST API 交互。希望您可以从此 API 从 curl 中提取如何执行此操作。

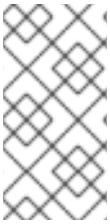
```
public interface ComponentsResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<ComponentRepresentation> query();

    @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，您必须指定供应商 ID，即字符串 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",
    "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();
```

6.13. 从以前的用户联邦 SPI 迁移



注意

本章只有在您使用之前（及现在已被删除）用户 Federation SPI 实施提供程序时才适用。

在 Keycloak 版本 2.4.0 及更早版本中，有一个 User Federation SPI。Red Hat Single Sign-On 版本 7.0 （但不支持）也提供了这个早期的 SPI。之前的用户 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，如果您用这个早期的 SPI 编写了提供程序，本章将讨论一些可用于端口的策略。

6.13.1. 导入而不是非导入

之前的用户 Federation SPI 要求您在红帽构建的 Keycloak 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。但是，这不再是必需的。您仍然可以将之前的提供程序移植为原来，但您应该考虑一个非重要策略是否是一个更好的方法。

导入策略的优点：

- 红帽构建的 Keycloak 基本上成为外部存储的持久性用户缓存。导入用户后，您将不再到达外部存储，从而关闭它。
- 如果您要以您的官方用户存储和弃用早期的外部存储迁移到 Red Hat build of Keycloak，您可以缓慢迁移应用程序以使用红帽构建的 Keycloak。当所有应用程序都迁移后，取消链接导入的用户，然后停用早期的传统外部存储。

使用导入策略有一些明显的缺点：

- 第一次查找用户将需要对红帽构建的 Keycloak 数据库进行多次更新。这可能会因为负载造成大量性能损失，并在红帽构建的 Keycloak 数据库上造成大量压力。用户联邦存储方法仅根据需要存储额外的数据，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构建的 Keycloak 存储和外部存储同步。User Storage SPI 具有可以实施以支持同步的功能接口，但这可能会很快变得困难且混乱。

6.13.2. UserFederationProvider versus UserStorageProvider

首先要注意的是 UserFederationProvider 是一个完整的接口。您在此界面中实施每个方法。但是，UserStorageProvider 已将这个接口分成多个您需要实现的功能接口。

UserFederationProvider.getUserByUsername （） 和 getUserByEmail （） 在新的 SPI 中有准确的等效项。两者之间的区别在于如何导入。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。相反，您将调用 KeycloakSession.userLocalStorage （）.addUser （）。userStorage （） 方法不再存在。

UserFederationProvider.validateAndProxy （） 方法已移到可选功能接口 ImportedUserValidation 中。如果您要以原样移植您的早期供应商，则需要实施此接口。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。在后续的 SPI 中，只有在本地用户从本地存储加载时才调用此方法。如果缓存了本地用户，则完全不调用 ImportedUserValidation.validate （） 方法。

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

UserFederationProvider 方法 同步Registrations （）、registerUser （） 和 removeUser （） 已被移到 UserRegistrationProvider 功能接口。这个新接口是可选的，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。如果您的以前的供应商有切换支持来注册新用户，在新的 SPI 中支持，如果供应商不支持添加用户，从 UserRegistrationProvider.addUser （） 返回 null。

以前的 用户FederationProvider 方法围绕凭证中心被封装在 CredentialInputValidator 和 CredentialInputUpdater 界面中，它们也是可选的，具体取决于您支持验证或更新凭证。UserModel 方法中用于存在的凭证管理。它们也被移到 CredentialInputValidator 和 CredentialInputUpdater 接口。请注意，如果您没有实现 CredentialInputUpdater 接口，则您的供应商提供的任何凭证都可以在红帽构建的 Keycloak 存储的本地覆盖。因此，如果您的凭据是只读的，请实施 CredentialInputUpdater.updateCredential （） 方法并返回 ReadOnlyException。

UserFederationProvider 查询方法（如 searchByAttributes （） 和 getGroupMembers （） ）现在被封装在可选接口 UserQueryProvider 中。如果您不实现此接口，则在管理控制台中将无法查看用户。您仍然能够登录。

6.13.3. UserFederationProviderFactory 和 UserStorageProviderFactory

之前 SPI 中的同步方法现在封装在可选的 ImportSynchronization 界面中。如果您实施了同步逻辑，则让新的 UserStorageProviderFactory 实现 ImportSynchronization 接口。

6.13.4. 升级到新模型

User Storage SPI 实例存储在不同的相关表中。红帽构建的 Keycloak 会自动运行一个迁移脚本。如

果为域部署了任何早期的用户 Federation 供应商，则它们会按照原样转换为后续存储模型，包括数据的 id。只有存在与之前的用户联邦提供程序 ID （如 "ldap", "kerberos"）相同的用户存储供应商时，才会进行此迁移。

因此，了解您可以采用的不同方法。

1. 您可以在以前的红帽构建的 Keycloak 部署中删除早期的供应商。这将删除您导入的所有用户的本地链接副本。然后，当您升级红帽构建的 Keycloak 时，只需为您的域部署和配置新供应商。

2. 第二个选项是编写新提供程序，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，并使内置迁移脚本从早期的数据模型转换为更新的数据模型。在这种情况下，您之前导入的所有用户都将正常工作，且相同。

如果您已决定导入策略并重写您的用户存储供应商，建议您在升级红帽构建的 Keycloak 前删除之前的供应商。这将删除您导入的任何用户的链接本地导入副本。

6.14. 基于流的接口

红帽构建的 Keycloak 中的许多用户存储接口包含可返回大量对象的查询方法，这可能会导致内存消耗和处理时间出现显著影响。当在查询方法的逻辑中使用对象内部状态的一小部分时，这尤其如此。

为开发人员提供在这些查询方法中处理大型数据集的更有效的替代方案，在用户存储接口中添加了流子接口。这些流 子接口将 super-interfaces 中的基于集合的方法替换为基于流的变体，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 对应部分，并将结果收集到正确的集合类型。

Streams 子接口允许实施专注于处理数据集合的基于流的方法，并从该方法的潜在内存和性能优化中受益。提供要实现的流子接口的接口包括几个 *功能接口*，org.keycloak.storage.federated 软件包中的所有接口，以及一些可能根据自定义存储实现的范围实施的接口。

请参阅向开发人员提供流子 接口 的接口列表。

软件包	类
org.keycloak.credential	**CredentialInputUpdater**(*)

org.keycloak.models	GroupModel,RoleMapperModel,UserModel
org.keycloak.storage.federated	所有接口
org.keycloak.storage.user	UserQueryProvider(*)

packagemanifests 表示接口是一个 *功能接口*

希望从流方法中获益的自定义用户存储实现应该只实现 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 7 章 VAULT SPI

7.1. VAULT 供应商

您可以使用来自 org.keycloak.vault 软件包的 vault SPI 为红帽构建的 Keycloak 编写自定义扩展，以连接到任意 vault 实现。

内置文件纯文本 提供程序是此 SPI 的实现示例。通常应用以下规则：

- 要防止机密在域间泄漏，您可能需要隔离或限制域可以检索的 secret。在这种情况下，您的供应商在查找 secret 时应考虑 realm 名称，例如，使用 realm name 前缀。例如，一个表达式 ${vault.key} 会根据域 *A* 或 realm *B* 中使用的不同条目名称进行评估。要区分域，需要将域从 VaultProvider Factory.create （） 方法传递给创建的 VaultProvider 实例，其中它可从 KeycloakSession 参数获取。

- vault 供应商需要实施单一方法 get Secret，该方法为给定 secret 名称返回 VaultRawSecret。该类以 byte[] 或 ByteBuffer 形式包含 secret 的表示，并期望根据需要在两者之间进行转换。请注意，在使用后将丢弃此缓冲区，如下所述。

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。

7.2. 从 VAULT 消耗值

库包含敏感数据，红帽构建的 Keycloak 会相应地处理 secret。在访问 secret 时，secret 会从密码库获取，并仅在所需时间保留在 JVM 内存中。然后，所有可能尝试丢弃来自 JVM 内存的内容。这可以通过仅在 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 方法也很有可能。但是 vault （） 方法具有将原始 secret （通常是字节阵列）解释为字符数组（通过 vault （）.getCharSecret （））或 String (via vault （).getStringSecret （））的功能，除了获取原始的非解释值（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覆盖来丢弃其内容。虽然在默认的 VaultStringSecret 实现中采取了措施以防止内部化 String，但存储在 String 对象中的 secret 至少会存在于下一个 GC 循环中。因此，最好使用普通字节和字符数组和缓冲区。