

用户级别令牌是与特定用户相关联的令牌。它们通常用于对用户的资源进行访问控制，例如对用户的个人资料、账户信息或其他与用户相关的数据进行访问。用户级别令牌通常是长期有效的，并且可以在多个应用程序之间共享。

应用级别令牌是与特定应用程序相关联的令牌。它们通常用于对应用程序的资源进行访问控制，例如对应用程序的数据、API 或其他与应用程序相关的功能进行访问。应用级别令牌通常是短期有效的，并且只能在单个应用程序中使用。

在 Keycloak 中，获取用户级别和应用级别令牌的方式略有不同。以下是它们之间的主要区别：

1. 令牌类型：用户级别令牌通常是 JWT (JSON Web Token) 令牌，而应用级别令牌通常是 Keycloak 自己的令牌格式。
2. 令牌生命周期：用户级别令牌通常是长期有效的，而应用级别令牌通常是短期有效的。
3. 令牌用途：用户级别令牌通常用于对用户的资源进行访问控制，而应用级别令牌通常用于对应用程序的资源进行访问控制。
4. 令牌共享：用户级别令牌可以在多个应用程序之间共享，而应用级别令牌只能在单个应用程序中使用。

ID token 和 access token 是在 OAuth 2.0 授权框架中常用的两种令牌。它们的主要区别在于用途和生命周期：

1. 用途：
 - ID token：用于验证用户身份，通常包含用户的身份信息，如用户 ID、用户名、用户角色等。
 - Access token：用于访问受保护的资源，通常包含访问资源所需的权限信息。
2. 生命周期：
 - ID token：通常是一次性的，生命周期较短，通常用于单次登录或会话期间。
 - Access token：生命周期较长，可以在多个请求中使用，通常用于长期访问受保护的资源。

总的来说，ID token 用于验证用户身份，而 access token 用于访问受保护的资源。ID token 通常生命周期较短，而 access token 生命周期较长，可以在多个请求中使用。在实际应用中，您可以根据具体需求选择使用 ID token 或 access token。

何时只需要id token

在 OAuth 2.0 授权框架中，当您只需要验证用户身份而不需要访问受保护的资源时，您只需要使用 ID token。例如，当您需要在客户端应用程序中显示用户信息时，您可以使用 ID token 来验证用户的身份，并获取用户的身份信息，如用户名、用户角色等。

此外，当您使用 OpenID Connect 协议时，ID token 是默认的令牌类型，它包含了用户的身份信息和授权信息，可以用于在客户端应用程序中验证用户的身份和获取用户的授权信息。

总之，当您只需要验证用户身份而不需要访问受保护的资源时，或者当您使用 OpenID Connect 协议时，您只需要使用 ID token。

意思是 openid connect协议默认会返回 id token , 同时也会返回 access token

是的, OpenID Connect 协议默认会返回 ID token 和 access token。

ID token 是 OpenID Connect 协议中的一种令牌, 它包含了用户的身份信息和授权信息, 可以用于在客户端应用程序中验证用户的身份和获取用户的授权信息。

Access token 是 OAuth 2.0 授权框架中的一种令牌, 它用于访问受保护的资源。在 OpenID Connect 协议中, access token 通常用于访问资源服务器上的受保护资源, 例如读取用户的个人资料或执行其他操作。

当您使用 OpenID Connect 协议进行身份验证时, 服务器会返回一个 ID token 和一个 access token。您可以使用 ID token 来验证用户的身份, 并获取用户的授权信息, 例如用户角色、权限等。您可以使用 access token 来访问资源服务器上的受保护资源。

需要注意的是, 具体的实现方式可能因服务器而异, 有些服务器可能只返回 ID token, 而有些服务器可能同时返回 ID token 和 access token。

OAuth2.0 中的 auth 是不是既包含了 authentication 意思 也包含了 authorization

是的, "OAuth 2.0" 中的 "auth" 一词既包含了 "authentication" (认证) 的意思, 也包含了 "authorization" (授权) 的意思。

"Authentication" 指的是验证用户的身份, 以确保用户是合法的。在 OAuth 2.0 中, 认证通常通过用户名和密码等方式进行。

"Authorization" 指的是授予用户访问特定资源的权限。在 OAuth 2.0 中, 授权通常通过令牌 (token) 的方式进行, 令牌代表了用户对特定资源的访问权限。

因此, "OAuth 2.0" 中的 "auth" 一词同时包含了 "authentication" 和 "authorization" 的意思, 它代表了 OAuth 2.0 授权框架的核心功能, 即认证和授权。

还有吗

以下是关于 OAuth 2.0 中认证流程的更多内容:

在 OAuth 2.0 中, 客户端向资源服务器请求资源时, 需要先向授权服务器请求授权。授权服务器会验证客户端的身份, 并根据客户端的请求和用户的授权信息, 决定是否颁发授权令牌。

在认证流程中, 客户端需要向授权服务器提供客户端标识 (Client Identifier) 和客户端密钥 (Client Secret), 以证明自己的身份。授权服务器会根据客户端标识和客户端密钥, 验证客户端的身份是否合法。

如果客户端的身份验证通过, 授权服务器会向客户端颁发一个授权令牌。授权令牌是一个字符串, 包含了授权服务器颁发的授权信息。客户端可以使用授权令牌向资源服务器请求资源, 资源服务器会验证授权令牌的有效性, 并根据令牌中的授权信息授予或拒绝客户端对资源的访问权限。

??

需要注意的是, 授权令牌是有时效的, 客户端需要在令牌失效前更新令牌的有效期。此外, OAuth 2.0 还提供了多种授权类型, 例如密码授权、客户端证书授权等, 以满足不同的应用场景。

总之, OAuth 2.0 认证流程的核心思想是通过客户端向授权服务器请求授权, 获得授权令牌, 然后使用授权令牌向资源服务器请求资源。这种认证方式可以保护用户的隐私和安全, 同时也为客户端应用程序提供了灵活的授权机制。

授权服务器怎么知道需要给客户端授予哪些资源的权限呢

在 OAuth 2.0 中，授权服务器通过客户端向其发送的授权请求（Authorization Request）来了解客户端需要哪些资源的权限。

授权请求中包含了客户端的标识、请求的资源、授权类型以及其他授权选项等信息。授权服务器会根据这些信息，决定是否授予客户端所需的资源权限。

具体来说，授权服务器的决策过程如下：

1. 授权服务器首先会验证客户端的身份，以确保请求是由合法的客户端发出的。
2. 授权服务器会根据客户端请求的资源，检查用户是否已经对这些资源进行了授权。如果用户已经授权，授权服务器会向客户端颁发授权令牌。
3. 如果用户没有对所需的资源进行授权，授权服务器会返回一个错误响应，告知客户端用户没有授权。
4. 如果用户对所需的资源进行了部分授权，授权服务器会根据授权选项，决定是否授予客户端所需的全部权限。
5. 授权服务器还可以根据客户端的请求，对授权进行限制，例如限制授权的有效期、限制授权的范围等。

总之，授权服务器通过授权请求来了解客户端需要哪些资源的权限，并根据用户的授权信息和授权选项，决定是否授予客户端所需的权限。

总之，授权服务器通过授权请求来了解客户端需要哪些资源的权限，并根据用户的授权信息和授权选项，决定是否授予客户端所需的权限。

授权选项指的是什么

在 OAuth 2.0 中，授权选项指的是在授权请求中包含的一些可选参数，用于指定授权的范围、有效期、权限等信息。这些选项可以帮助授权服务器更好地理解客户端的请求，并根据用户的授权信息和应用场景，授予适当的权限。

常见的授权选项包括：

1. scope：指定客户端需要的资源权限范围，例如读取用户的个人信息、修改用户的个人资料等。
2. response_type：指定授权服务器返回的授权类型，例如授权令牌（Authorization Token）、刷新令牌（Refresh Token）等。
3. client_id：客户端的标识，用于标识客户端的身份。
4. redirect_uri：指定客户端在授权成功后重定向的地址，用于接收授权服务器返回的授权信息。
5. state：用于防止 CSRF（跨站请求伪造）攻击的随机字符串，用于验证请求的合法性。
6. max_age：指定授权令牌的有效期，单位为秒。
7. authorization_code：用于获取授权码的选项，授权码可以用于获取授权令牌。
8. refresh_token：用于获取刷新令牌的选项，刷新令牌可以用于延长授权令牌的有效期。

这些授权选项可以根据实际应用场景进行配置和修改，以满足不同的授权需求。授权服务器会根据这些选项，决定是否授予客户端所需的权限，并返回适当的授权信息。

资源服务器怎么验证授权令牌的有效性的？

在 OAuth 2.0 授权框架中，资源服务器可以使用多种方法来验证授权令牌的有效性。以下是常用的几种方法：

1. 使用令牌签名算法（Token Signature Algorithm）验证：授权令牌通常包含一个签名，该签名是使用授权服务器的私钥对令牌的一些重要信息进行签名得到的。资源服务器可以使用相同的签名算法和公钥来验证令牌的签名是否有效，从而确认令牌的来源和有效性。
2. 使用令牌有效期（Token Expiration Date）验证：授权令牌通常包含一个有效期，资源服务器可以在处理请求时检查令牌的有效期是否已经过期。如果令牌已经过期，资源服务器可以拒绝请求。
3. 使用令牌的颁发者（Token Issuer）验证：授权令牌通常包含一个颁发者字段，该字段表示令牌是由哪个授权服务器颁发的。资源服务器可以验证令牌的颁发者是否是可信的授权服务器，从而确认令牌的来源和有效性。
4. 使用客户端的身份验证：资源服务器可以要求客户端在发送请求时提供其身份验证信息，例如客户端 ID 和客户端密钥。资源服务器可以使用这些信息来验证客户端的身份，并确认客户端是否有权使用授权令牌。

以上方法可以结合使用，以提高授权令牌的安全性和有效性。资源服务器可以根据具体的应用场景和安全需求选择合适的验证方法。

资源服务器如何得知授权服务器使用的的签名算法，如何知道授权服务器的公钥？

在 OAuth 2.0 授权框架中，资源服务器需要知道授权服务器使用的签名算法和公钥，以便验证令牌的签名是否有效。

授权服务器会在颁发令牌时将签名算法和公钥一起发送给客户端，客户端可以将公钥存储在本地，并在向资源服务器发送令牌时一起发送。资源服务器可以从客户端接收到令牌和公钥，然后使用令牌签名算法和公钥来验证令牌的签名是否有效。

另外，授权服务器还可以将公钥发布到一个可公开访问的证书权威机构（CA），资源服务器可以从 CA 获取公钥，并使用该公钥来验证令牌的签名是否有效。

需要注意的是，为了保证令牌的安全性和可靠性，授权服务器应该使用安全的方式来发布公钥，例如使用数字证书或者其他安全机制来保护公钥的机密性和完整性。同时，资源服务器和客户端也需要采取措施来保护公钥的安全性，例如将公钥存储在安全的位置，并对其进行加密保护。

还有吗

除了之前提到的方法，还有以下几种方式可以让资源服务器得知授权服务器使用的签名算法和公钥：

1. **通过预配置**：在部署 OAuth 2.0 授权服务器和资源服务器时，可以事先在它们之间进行预配置，指定授权服务器使用的签名算法和公钥。这种方式适用于固定的 OAuth 2.0 部署场景，不需要每次验证令牌时都进行交互。
2. **通过 API 调用**：资源服务器可以通过调用授权服务器的 API，获取令牌的签名算法和公钥。这种方式需要授权服务器提供相应的 API 接口，并且需要在网络上进行交互，因此安全性需要特别注意。
3. **通过 JWT 头信息**：如果授权服务器颁发的令牌是 JWT 令牌，那么令牌本身就包含了签名算法和公钥信息。资源服务器可以解析 JWT 令牌的头信息，获取签名算法和公钥。

需要根据实际情况选择合适的方式来获取签名算法和公钥，并采取相应的安全措施来保护它们的机密性和完整性。

如果授权服务器使用证书来存放公钥，资源服务器验证token有效性时候，怎么才能拿到证书，进而又如何从证书里面拿到公钥呢？

如果授权服务器使用证书来存放公钥，资源服务器可以通过以下步骤来验证令牌的有效性并获取公钥：

1. 证书获取：资源服务器可以通过网络请求或其他方式从授权服务器获取到包含公钥的证书。
2. 证书验证：资源服务器需要验证证书的有效性，以确保证书是由合法的证书颁发机构颁发的，并且没有过期或被吊销。可以使用证书链或证书吊销列表等方式来验证证书的有效性。
3. 公钥提取：一旦证书被验证为有效，资源服务器可以使用证书中的公钥来验证令牌的签名。可以使用 PKI (Public Key Infrastructure) 标准中的 API 来提取证书中的公钥。
4. 令牌验证：资源服务器使用提取出来的公钥来验证令牌的签名，以确保令牌是由授权服务器颁发的，并且没有被篡改。如果令牌验证通过，资源服务器可以认为令牌是有效的，并使用令牌中的授权信息来访问受保护的资源。

需要注意的是，在实际应用中，为了保证安全性，证书的获取和验证应该在安全的网络环境中进行，并且需要采取相应的安全措施来保护证书和公钥的机密性和完整性。同时，为了避免证书被篡改或替换，资源服务器也应该对证书进行定期的验证和更新。

如何获取证书

证书可以通过多种方式获取，以下是一些常见的方法：

1. 在线申请：许多证书颁发机构（CA）提供在线证书申请服务。您可以在 CA 的网站上填写申请表，提供必要的信息，如域名、公司名称、个人信息等，然后支付证书费用并等待证书颁发。
2. 命令行工具：一些操作系统提供了命令行工具，可以用来生成和管理证书。例如，在 Windows 上，可以使用 makecert 和 certmgr.msc 命令来生成和管理证书。
3. 证书管理软件：许多证书管理软件可以帮助您生成、管理和部署证书。这些软件通常提供了图形化界面，方便您管理证书。
4. 自签名证书：如果您是开发人员，并且只需要在自己的设备上使用证书，可以使用自签名证书。自签名证书是由您自己生成的，不需要通过 CA 进行签名。但是，自签名证书在安全性上不如通过 CA 签名的证书。

无论您使用哪种方法获取证书，都需要确保生成的证书符合您的需求，并且证书的私钥是安全的。此外，您还需要根据证书的用途和使用场景，选择合适的证书类型和算法。

使用公钥验证令牌的步骤如下：

1. 获得公钥：首先需要获得公钥，可以从公钥证书、PEM 文件或其他可信来源获取公钥。
2. 读取令牌：读取要验证的令牌，可以是 JSON Web Token (JWT)、OAuth 2.0 Token 等。
3. 提取令牌中的负载：从令牌中提取负载，即包含授权信息的部分。
4. 计算令牌的散列值：使用提取出的负载计算散列值，可以使用 HMAC、SHA-256 等算法。
5. 使用公钥验证签名：使用公钥验证令牌中的签名，签名是使用令牌的私钥计算得到的，只有令牌的创建者拥有私钥。

使用公钥验证签名时需要哪些信息

使用公钥验证签名时需要以下信息：

1. 公钥：用于验证签名的公钥。该公钥必须是由可信的第三方机构（如证书颁发机构）颁发的，并且与签名者的身份相关联。
2. 待验证的签名：需要验证的签名数据。该签名是使用签名者的私钥计算得到的，用于证明签名者对数据的所有权和完整性。
3. 待验证的数据：签名所保护的数据。该数据可以是任何形式的数据，如文件、文本、图片等。

具体的验证过程如下：

1. 使用公钥加密待验证的数据：使用公钥加密待验证的数据，得到加密后的数据。
2. 计算待验证的签名：使用签名者的私钥计算待验证的数据的签名，得到签名数据。
3. 比较加密后的数据和签名数据：将加密后的数据和签名数据进行比较，如果相同，则说明签名是有效的，否则说明签名无效。

需要注意的是，公钥必须是可信的，并且与签名者的身份相关联。在实际应用中，公钥通常来自于可信的证书颁发机构，并且可以通过证书链验证其有效性。同时，为了提高安全性，可以使用数字签名算法（如 RSA、ECDSA 等）来计算签名，并使用加密算法（如 AES、RSA 等）来加密待验证的数据。

openid connect 与 spring authorization server 有啥区别

OpenID Connect (OIDC) 和 Spring Authorization Server 是两种不同的技术，用于实现身份验证和授权。

OIDC 是一种开放标准，用于在 Web 应用程序和 API 之间进行身份验证和授权。它基于 OAuth 2.0 协议，并为其增加了身份验证和用户属性的功能。OIDC 提供了一种标准的方式来获取用户的身份信息和授权令牌，并可以与其他应用程序和服务进行集成。

Spring Authorization Server 是 Spring Security 框架的一部分，用于实现 OAuth 2.0 和 OpenID Connect 协议。它提供了一个完整的授权服务器实现，可以生成令牌，管理用户权限，以及与其他应用程序和服务进行集成。

因此，可以说 OIDC 和 Spring Authorization Server 都是用于实现身份验证和授权的技术，但它们的实现方式和功能有所不同。OIDC 是一种开放标准，提供了一种标准的方式来获取用户的身份信息和授权令牌；而 Spring Authorization Server 则是一个完整的授权服务器实现，可以生成令牌，管理用户权限，以及与其他应用程序和服务进行集成。

Spring Authorization Server

Spring Authorization Server 是一个基于 Spring Security 的授权服务器实现，它提供了一个完整的 OAuth 2.0 授权框架，可以生成令牌，管理用户权限，以及与其他应用程序和服务进行集成。它支持多种 OAuth 2.0 协议，包括 Client Credentials Grant、Authorization Code Grant 和 Implicit Grant，并提供了丰富的配置和管理功能。

??? 场景与用途

openid connect , oAuth2 , Spring Authorization Server 这三者之间的关系到底该如何梳理，如何理解

OpenID Connect (OIDC) 和 OAuth 2.0 是两种用于身份验证和授权的开放标准。它们都提供了一种通过令牌 (token) 来保护资源访问的机制，令牌可以在客户端和资源服务器之间传递，以验证客户端的身份和授权其访问资源。

Spring Authorization Server 是一个用于实现 OAuth 2.0 和 OIDC 的框架。它提供了一个完整的授权服务器实现，可以生成令牌，管理用户权限，以及与其他应用程序和服务进行集成。

总之，可以将 Spring Authorization Server 视为 OAuth 2.0 和 OIDC 的一个具体实现。它提供了一个完整的授权服务器框架，可以帮助开发人员轻松实现 OAuth 2.0 和 OIDC 协议，并与其他应用程序和服务进行集成。

openidconnect 只是一种协议吗？不是具体的产品实现？

是的，OpenID Connect (OIDC) 是一种开放标准协议，而不是具体的产品实现。

OIDC 是基于 OAuth 2.0 协议的一种扩展，主要用于在分布式应用程序中进行身份验证和授权。它提供了一种标准的方式来获取用户的身份信息（如用户名、邮箱地址等）和令牌（如 access token、refresh token 等），并可以与其他应用程序和服务进行集成。

虽然 OIDC 不是具体的产品实现，但有许多开源和商业的 OIDC 实现可供选择，例如：

- Spring Authorization Server：一个基于 Spring Security 的授权服务器实现，提供了 OIDC 授权框架。
- Keycloak：一个开源的身份和访问管理 (IAM) 平台，提供了 OIDC 支持。
- Auth0：一个商业的身份验证和授权服务提供商，提供了 OIDC 实现。

这些产品都实现了 OIDC 协议，并提供了不同的功能和特点，以满足不同的需求。因此，在实际应用中，开发人员可以根据具体需求选择合适的 OIDC 实现。

如何使用spring authorization server 来保护 一个或者多个 spring boot Rest API 应用

使用 Spring Authorization Server 来保护一个或多个 Spring Boot REST API 应用的步骤如下：

1. 安装 Spring Authorization Server：首先，你需要安装 Spring Authorization Server。它是一个基于 Spring Security 的授权服务器实现，可以生成令牌，管理用户权限，以及与其他应用程序和服务进行集成。
2. 配置授权服务器：在安装完成后，你需要配置授权服务器。这包括设置 OIDC 配置，如 Issuer URL、JWKS URL、Audience URL 等，以及用户管理和权限管理配置。
3. 创建客户端应用程序：在你的 Spring Boot REST API 应用中，你需要创建一个客户端应用程序，以与授权服务器进行交互。这包括配置 OIDC 客户端，如配置 Callback URL、获取 access token 等。
4. 配置 API 授权策略：在你的 Spring Boot REST API 应用中，你需要配置 API 授权策略。这包括设置哪些资源需要进行授权，以及哪些用户或角色具有访问这些资源的权限。
5. 使用令牌进行访问控制：在你的 Spring Boot REST API 应用中，你需要使用令牌进行访问控制。当你的客户端应用程序向 API 发送请求时，它需要携带 access token。API 应用程序将使用该 access token 验证客户端的身份和授权，并决定是否允许访问资源。

通过以上步骤，你可以使用 Spring Authorization Server 来保护你的 Spring Boot REST API 应用。它提供了一个完整的授权服务器框架，可以帮助你轻松实现 OAuth 2.0 和 OIDC 协议，并与其他应用程序和服务进行集成。

是的，Issuer URL、JWKS URL 和 Audience URL 这些配置通常是在授权服务器安装好之后就已经存在的。

Issuer URL 是指授权服务器的身份验证 URL，客户端应用程序可以通过该 URL 向授权服务器进行身份验证。

JWKS URL 是指授权服务器的 JSON Web Key Set (JWKS) URL，它包含了授权服务器的公钥信息，客户端应用程序可以通过该 URL 获取授权服务器的公钥，用于验证令牌的签名。

Audience URL 是指客户端应用程序的身份验证目标 URL，授权服务器会根据该 URL 来验证客户端应用程序是否具有访问资源的权限。

这些配置通常是在授权服务器的配置文件或代码中定义的，并且可以根据实际需求进行修改。如果你使用的是开源授权服务器，通常可以在其文档或配置文件中找到这些配置的详细信息。如果你使用的是自定义授权服务器，则需要根据你的实现方式来确定如何配置这些参数。

Issuer URL 是 OpenID Connect (OIDC) 授权协议中的一个重要概念。它是指授权服务器的身份验证 URL，客户端应用程序可以通过该 URL 向授权服务器进行身份验证。

对于 Issuer URL，有以下几点要求：

1. 必须是有效的 URL：Issuer URL 必须是一个有效的 URL，能够被客户端应用程序访问到。
2. 必须是唯一的：每个授权服务器必须有一个唯一的 Issuer URL，以便客户端应用程序能够正确地向授权服务器进行身份验证。
3. 必须可验证：客户端应用程序可以通过向 Issuer URL 发送 HTTP GET 请求来获取授权服务器的 JSON Web Key Set (JWKS)，并验证 Issuer URL 是否与 JWKS 中的信息匹配。
4. 必须可解析：Issuer URL 必须能够被解析为一个有效的域名或主机名，以便客户端应用程序能够确定其所在的域。

因此，Issuer URL 并不是一个可以任意定义的 URL 地址。它必须符合上述要求，并且必须是唯一的，以便客户端应用程序能够正确地向授权服务器进行身份验证。在实际应用中，Issuer URL 通常是一个具体的 URL，或者是一个域名或主机名的别名。

JWKS 如何配置，及配置要求

JWKS (JSON Web Key Set) 是 OpenID Connect (OIDC) 授权协议中的一种用于存储公钥和私钥的标准格式。在配置 OIDC 授权服务器时，你需要配置 JWKS，以便客户端应用程序可以获取授权服务器的公钥，用于验证令牌的签名。

以下是配置 JWKS 的一般步骤：

1. 生成 JWKS：首先，你需要生成 JWKS。这可以通过使用授权服务器的密钥管理系统或其他工具来完成。
2. 确定 JWKS 的存储位置：你需要确定 JWKS 的存储位置。这可以是一个文件，一个数据库或其他存储方式。
3. 在配置文件中设置 JWKS 的 URL：在授权服务器的配置文件中，你需要设置 JWKS 的 URL。这通常是一个指向 JWKS 存储位置的 URL。
4. 验证 JWKS 的有效性：在设置完 JWKS 的 URL 后，你需要验证它是否有效。你可以使用 OIDC 客户端应用程序向该 URL 发送 HTTP GET 请求，并检查响应是否正确。
5. 更新 JWKS：如果你更新了授权服务器的密钥，你需要相应地更新 JWKS。这可以通过更新 JWKS 的存储位置或使用其他方式来完成。

对于 JWKS 的配置，有以下几点要求：

1. JWKS 必须是有效的 JSON 格式：JWKS 必须是有效的 JSON 格式，并且符合 OIDC 协议的标准格式。
2. JWKS 必须包含公钥和私钥：JWKS 必须包含授权服务器的公钥和私钥，以便客户端应用程序可以使用公钥来验证令牌的签名。
3. JWKS 必须是可访问的：JWKS 的 URL 必须是可访问的，并且客户端应用程序可以通过该 URL 获取 JWKS。
4. JWKS 必须定期更新：如果你更新了授权服务器的密钥，你需要相应地更新 JWKS。这可以确保客户端应用程序使用最新的公钥来验证令牌的签名。

java 中如何获取到 某个有证书的网站的证书

在 Java 中，你可以使用 KeyStore 和 TrustStore 来获取某个网站的证书。

KeyStore 是一个存储私钥和公钥的容器，而 TrustStore 是一个存储信任证书的容器。当你访问某个网站时，浏览器会使用网站的证书来验证网站的身份。同样，在 Java 中，你也可以使用 KeyStore 和 TrustStore 来获取网站的证书，并进行身份验证。

以下是获取网站证书的一般步骤：

1. 创建 KeyStore 和 TrustStore：首先，你需要创建一个 KeyStore 和一个 TrustStore。KeyStore 用于存储你的私钥和公钥，而 TrustStore 用于存储你信任的证书。
2. 加载网站的证书：你可以使用 KeyStore 和 TrustStore 的 load() 方法来加载网站的证书。你需要提供证书的路径和密码，以便 KeyStore 和 TrustStore 能够加载证书。
3. 验证网站的身份：在加载完网站的证书后，你可以使用 KeyStore 和 TrustStore 的 getCertificate() 方法来获取证书，并使用证书的公钥来验证网站的身份。
4. 设置 KeyStore 和 TrustStore：最后，你需要设置 KeyStore 和 TrustStore，以便 Java 应用程序能够使用它们来进行身份验证。你可以使用 SecurityManager 或其他方法来设置 KeyStore 和 TrustStore。

请注意，获取网站证书需要一定的网络编程知识和安全知识。如果你不熟悉这些知识，建议你先学习相关的教程和文档，或者请教专业的安全工程师。

非常抱歉，我之前提提供的示例代码存在缺失。以下是一个完整的示例代码，演示如何使用 `SSLContext` 类获取证书：

```
java

import java.io.File;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;

import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;

public class SSLContextExample {
    public static void main(String[] args) {
        // 创建 KeyStore
        KeyStore keyStore = KeyStore.getInstance("JKS");
        try {
            keyStore.load(new File("keystore.jks"), "password".toCharArray());
        } catch (KeyStoreException | CertificateException | NoSuchAlgorithmException e) {
            e.printStackTrace();
        }

        // 创建 TrustManager
        TrustManager[] trustManagers = new TrustManager[] { new MyX509TrustManager() };

        // 创建 SSLContext
        SSLContext sslContext = SSLContext.getInstance("TLS");
        try {
            sslContext.init(keyStore, "password".toCharArray(), trustManagers);
        } catch (KeyStoreException | CertificateException | NoSuchAlgorithmException | KeyMan
            e.printStackTrace();
        }

        // 获取证书
        X509Certificate certificate = (X509Certificate)sslContext.getCertificates().nextEleme
```

```
// 获取证书
X509Certificate certificate = (X509Certificate)sslContext.getCertificates().nextElement();
System.out.println(certificate.getSubjectDN());
System.out.println(certificate.getIssuerDN());
}

private static class MyX509TrustManager implements X509TrustManager {
    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {}

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {}
}
```

接着写

以下是示例代码的续写：

```
java
@Override
public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[0];
}
}
```

在上述代码中，我们定义了一个名为 `MyX509TrustManager` 的类，它实现了 `X509TrustManager` 接口。在这

在上述代码中，我们定义了一个名为 `MyX509TrustManager` 的类，它实现了 `X509TrustManager` 接口。在这个类中，我们实现了 `checkClientTrusted` 和 `checkServerTrusted` 方法，它们都被注释掉了，因为我们不需要在这个示例中进行客户端或服务端的证书验证。

我们还实现了 `getAcceptedIssuers` 方法，它返回一个空的 `X509Certificate` 数组，表示我们接受所有的证书。在实际应用中，你可以根据需要添加证书验证逻辑。

最后，在 `SSLContextExample` 类的 `main` 方法中，我们创建了一个 `SSLContext` 对象，并将我们定义的 `MyX509TrustManager` 作为信任管理器传递给 `init` 方法。然后，我们使用 `getCertificates` 方法获取证书，并打印出证书的主题和颁发者信息。

请注意，上述代码中的证书路径和密码需要根据你实际的证书和密钥库进行修改。

如何根据 https url 获取证书信息

要获取 HTTPS (Hypertext Transfer Protocol Secure) 协议的 URL 对应的证书信息，可以按照以下步骤进行操作：

1. 使用适当的网络库（例如 `HttpClient`）发起 HTTPS 请求，并获取响应的内容。
2. 使用响应内容中的 `getInputStream` 方法获取输入流，该流包含了证书信息。
3. 使用 Java 中的 `CertificateFactory` 类创建一个 `X509Certificate` 对象，该对象表示证书的结构和内容。
4. 使用 `X509Certificate` 对象的方法获取证书的详细信息，例如主题、颁发者、有效日期等。

下面是一个简单的示例代码，演示如何使用上述步骤获取证书信息：

```
import javax.net.ssl.X509TrustManager;

public class SSLCertificateInfo {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://www.example.com");
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();

            // 信任所有证书
            SSLContext context = SSLContext.getInstance("TLS");
            context.init(null, new TrustManager[] { new X509TrustManager() {
                @Override
                public void checkClientTrusted(X509Certificate[] chain, String authType) thro
            }

            @Override
            public void checkServerTrusted(X509Certificate[] chain, String authType) thro
            }

            @Override
            public X509Certificate[] getAcceptedIssuers() {
                return new X509Certificate[0];
            }
        } }, null);
        conn.setSSLSocketFactory(context.getSocket
```

接着写

以下是接着上面代码的内容：

```
java
        .getSocketFactory());

        conn.setRequestMethod("GET");
        conn.setDoOutput(true);

        // 发送 HTTP 请求并获取响应
        int responseCode = conn.getResponseCode();
        BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStr
        String line;
        StringBuffer response = new StringBuffer();
        while ((line = reader.readLine()) != null) {
            response.append(line);
        }
        reader.close();

        // 打印证书信息
        X509Certificate certificate = conn.getServerCertificate();
        if (certificate != null) {
            System.out.println("Subject DN: " + certificate.getSubjectDN());
            System.out.println("IssuerDN: " + certificate.getIssuerDN());
            System.out.println("Validity: " + certificate.getNotBefore() + " - " + certif
        } else {
            System.out.println("Certificate is null.");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

访问一个https请求时候，https服务端会自动将证书信息发送到请求的客户端吗？

是的，当客户端向 HTTPS 服务器发出请求时，服务器会自动将其证书信息发送到客户端。

在 HTTPS 通信中，服务器使用 SSL/TLS 协议来保护通信内容的机密性和完整性。在建立 SSL/TLS 连接时，服务器会向客户端发送其证书，其中包含了服务器的公钥和其他信息。客户端可以使用证书中的公钥来验证服务器的身份，并确保通信是安全的。

因此，当您访问一个 HTTPS 请求时，服务器会自动将其证书信息发送到客户端，以便客户端进行验证和建立安全的连接。

在Java中要访问Https链接时，会用到一个关键类`HttpsURLConnection`；参见如下实现代码：

```
1. // 创建URL对象
2. URL myURL = new URL("https://www.sun.com");
3.
4. // 创建HttpsURLConnection对象，并设置其SSLContextFactory对象
5. HttpsURLConnection httpsConn = (HttpsURLConnection) myURL
6.     .openConnection();
7.
8. // 取得该连接的输入流，以读取响应内容
9. InputStreamReader insr = new InputStreamReader(httpsConn
10.     .getInputStream());
11.
12. // 读取服务器的响应内容并显示
13. int respInt = insr.read();
14. while (respInt != -1) {
15.     System.out.print((char) respInt);
16.     respInt = insr.read();
17. }
```

在取得connection的时候和正常浏览器访问一样，仍然会验证服务端的证书是否被信任（权威机构发行或者被权威机构签名）；如果服务端证书不被信任，则默认的实现就会有异常，一般来说，用`SunJSSE`会抛出如下异常信息：

`javax.net.ssl.SSLHandshakeException: sun.security.validator.ValidatorException: PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target`

上面提到`SunJSSE`，`JSSE`（Java Secure Socket Extension）是实现Internet安全通信的一系列包的集合。它是一个SSL和TLS的纯Java实现，可以透明地提供数据加密、服务器认证、信息完整性等功能，可以使我们像使用普通的套接字一样使用`JSSE`建立的安全套接字。`JSSE`是一个开放的标准，不只是Sun公司才能实现一个`SunJSSE`，事实上其他公司也有自己实现的`JSSE`，然后通过JCA就可以在JVM中使用。

关于`JSSE`的详细信息参考官网Reference：<http://java.sun.com/j2se/1.5.0/docs/guide/security/jsse/JSSERefGuide.html>；

以及Java Security Guide：<http://java.sun.com/j2se/1.5.0/docs/guide/security/>；

在深入了解`JSSE`之前，需要了解一个有Java安全的概念：客户端的`TrustStore`文件。客户端的`TrustStore`文件中保存着被客户端所信任的服务器的证书信息。客户端在进行SSL连接时，`JSSE`将根据这个文件中的证书决定是否信任被连接的证书。在`SunJSSE`中，有一个信任管理器类负责决定是否信任连接的证书，这个类有如下的处理规则：

1. 若系统属性`javax.net.ssl.trustStore`指定了`TrustStore`文件，那么信任管理器就去Jre安装路径下的`lib/security/`目录中寻找并使用这个文件来检查证书。
2. 若该系统属性没有指定`TrustStore`文件，它就会去Jre安装路径下寻找默认的`TrustStore`文件，这个文件的相对路径为：`lib/security/jssecacerts`。
3. 若`jssecacerts`不存在，但是`cacerts`存在（它随J2SDK一起发行，含有数量有限的可信的基本证书），那么这个默认的`TrustStore`文件就是`lib/security/cacerts`。

那遇到这种情况，怎么处理呢？有以下两种方案：

1. 按照以上信任管理器的规则，将服务端的公钥导入到`jssecacerts`，或者是在系统属性中设置要加载的`trustStore`文件的路径；证书导入可以用如下命令：`keytool -import -file src_cer_file -keystore dest_cer_store`；至于证书可以通过浏览器导出获得；
2. 实现自己的证书信任管理器，比如`MyX509TrustManager`，该类必须实现`X509TrustManager`接口中的三个method；然后在`HttpsURLConnection`中加载自定义的类，可以参见如下两个代码片段，其一为自定义证书信任管理器，其二为connect时的代码：

```
/* 忽略HTTPS请求的SSL证书，必须在openConnection之前调用
 * @throws Exception
 */
public static void ignoreSsl() throws Exception{
    HostnameVerifier hv = new HostnameVerifier() {
        public boolean verify(String urlHostName, SSLSession session) {
            return true;
        }
    };
    trustAllHttpsCertificates();
    HttpsURLConnection.setDefaultHostnameVerifier(hv);
}
```


java PKI 如何从证书中获取公钥

在 Java 中，您可以使用 `X509Certificate` 类来获取证书中的公钥。以下是一个示例代码，演示了如何从证书中获取公钥：

```
java

import java.security.cert.X509Certificate;

public class GetPublicKeyFromCertificate {
    public static void main(String[] args) {
        // 假设您已经有了一个 X509Certificate 实例
        X509Certificate certificate = /* 获取证书的方式 */;

        // 获取公钥
        PublicKey publicKey = certificate.getPublicKey();

        // 使用公钥进行操作
        /* 例如，加密或验证数据 */
    }
}
```

在上面的代码中，您可以使用任何方法获取一个 `X509Certificate` 实例，例如从文件、数据库或网络连接中读取证书。然后，使用 `getPublicKey()` 方法获取证书中的公钥，并对其进行操作。

请注意，证书中的公钥是一个 `java.security. PublicKey` 实例，您可以使用其提供的方法进行各种操作，例如加密或验证数据。

HostnameVerifier

在 Java 中，`HostnameVerifier` 是一个接口，用于定义验证主机名的逻辑。当您使用 SSL/TLS 进行网络通信时，您可以配置 `HostnameVerifier` 来验证服务器的主机名是否与证书中的主机名匹配。

`HostnameVerifier` 的工作方式如下：

1. 当客户端尝试连接服务器时，它会检查服务器的主机名。
2. 如果客户端配置了 `HostnameVerifier`，则它会将服务器的主机名传递给 `HostnameVerifier` 的 `verify()` 方法。
3. `HostnameVerifier` 的 `verify()` 方法会根据您配置的逻辑验证服务器的主机名是否有效。
4. 如果 `verify()` 方法返回 `true`，则表示服务器的主机名有效，客户端可以继续连接。
5. 如果 `verify()` 方法返回 `false`，则表示服务器的主机名无效，客户端应该拒绝连接。

自定义了HostnameVerifier

在握手期间，如果 URL 的主机名和服务器的标识主机名不匹配，则验证机制可以回调此接口的实现程序来确定是否应该允许此连接。如果回调内实现不恰当，默认接受所有域名，则有安全风险。代码示例。

```
1 HostnameVerifier hnv = new HostnameVerifier() {
2     @Override
3     public boolean verify(String hostname, SSLSession session) {
4         // Always return true, 接受任意域名服务器
5         return true;
6     }
7 };
8 HttpURLConnection.setDefaultHostnameVerifier(hnv);
9
```

信任所有主机名

```
1 SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
2 sf.setHostnameVerifier(SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER);
3
```

方案1

不论是权威机构颁发的证书还是自签名的，打包一份到 app 内部，比如存放在 asset 里。通过这份内置的证书初始化一个KeyStore，然后用这个KeyStore去引导生成的TrustManager来提供验证，具体代码如下：

```
1 try {
2     CertificateFactory cf = CertificateFactory.getInstance("X.509");
3     // uwca.crt 打包在 asset 中，该证书可以从https://itconnect.uw.edu/security/securing-computer/i
4     InputStream caInput = new BufferedInputStream(getAssets().open("uwca.crt"));
5     Certificate ca;
6     try {
7         ca = cf.generateCertificate(caInput);
8         Log.i("Longer", "ca=" + ((X509Certificate) ca).getSubjectDN());
9         Log.i("Longer", "key=" + ((X509Certificate) ca).getPublicKey());
10    } finally {
11        caInput.close();
12    }
13
14    // Create a KeyStore containing our trusted CAs
15    String keyStoreType = KeyStore.getDefaultType();
16    KeyStore keyStore = KeyStore.getInstance(keyStoreType);
17    keyStore.load(null, null);
18    keyStore.setCertificateEntry("ca", ca);
19
20    // Create a TrustManager that trusts the CAs in our KeyStore
21    String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
22    TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
23    tmf.init(keyStore);
24
25    // Create an SSLContext that uses our TrustManager
26    SSLContext context = SSLContext.getInstance("TLSv1", "AndroidOpenSSL");
27    context.init(null, tmf.getTrustManagers(), null);
28
29    URL url = new URL("https://certs.cac.washington.edu/CAtest/");
30    HttpsURLConnection urlConnection =
31        (HttpsURLConnection)url.openConnection();
32    urlConnection.setSSLSocketFactory(context.getSocketFactory());
33    InputStream in = urlConnection.getInputStream();
34    copyInputStreamToOutputStream(in, System.out);
35 } catch (CertificateException e) {
36     e.printStackTrace();
37 } catch (IOException e) {
38     e.printStackTrace();
39 } catch (NoSuchAlgorithmException e) {
40     e.printStackTrace();
41 } catch (KeyStoreException e) {
42     e.printStackTrace();
43 }
```

方案2

同方案1，打包一份到证书到 app 内部，但不通过KeyStore去引导生成的TrustManager，而是干脆直接自定义一个TrustManager，自己实现校验逻辑；校验逻辑主要包括：

- 服务器证书是否过期
- 证书签名是否合法

```
1 try {
2     CertificateFactory cf = CertificateFactory.getInstance("X.509");
3     // uwca.crt 打包在 asset 中，该证书可以从https://itconnect.uw.edu/security/securing-computer/i
4     InputStream caInput = new BufferedInputStream(getAssets().open("uwca.crt"));
5     final Certificate ca;
6     try {
7         ca = cf.generateCertificate(caInput);
8         Log.i("Longer", "ca=" + ((X509Certificate) ca).getSubjectDN());
9         Log.i("Longer", "key=" + ((X509Certificate) ca).getPublicKey());
10    } finally {
11        caInput.close();
12    }
13    // Create an SSLContext that uses our TrustManager
14    SSLContext context = SSLContext.getInstance("TLSv1", "AndroidOpenSSL");
15    context.init(null, new TrustManager[]{
16        new X509TrustManager() {
17            @Override
18            public void checkClientTrusted(X509Certificate[] chain,
19                String authType)
20                throws CertificateException {
21
22            }
23
24            @Override
25            public void checkServerTrusted(X509Certificate[] chain,
26                String authType)
27                throws CertificateException {
28                for (X509Certificate cert : chain) {
29
30                    // Make sure that it hasn't expired.
31                    cert.checkValidity();
32
33                    // Verify the certificate's public key chain.
34                    try {
35                        cert.verify(((X509Certificate) ca).getPublicKey());
36                    } catch (NoSuchAlgorithmException e) {
37                        e.printStackTrace();
38                    } catch (InvalidKeyException e) {
39                        e.printStackTrace();
40                    } catch (NoSuchProviderException e) {
41                        e.printStackTrace();
42                    }
43                }
44            }
45        }
46    }, new KeyManager[] {});
47    HttpsURLConnection.setDefaultSSLSocketFactory(context.getSocketFactory());
48}
```

```

        e.printStackTrace();
    } catch (NoSuchProviderException e) {
        e.printStackTrace();
    } catch (SignatureException e) {
        e.printStackTrace();
    }
    }
}

@Override
public X509Certificate[] getAcceptedIssuers() {
    return new X509Certificate[0];
}
}
}, null);

URL url = new URL("https://certs.cac.washington.edu/CAtest/");
URLConnection urlConnection =
    (URLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(context.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
} catch (CertificateException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
} catch (KeyManagementException e) {
    e.printStackTrace();
} catch (NoSuchProviderException e) {
    e.printStackTrace();
}
}

```


同样上述代码只能访问 certs.cac.washington.edu 相关域名地址，如果访问 <https://www.taobao.com/> 或者 <https://www.baidu.com/>，则会在 `cert.verify(((X509Certificate)ca).getPublicKey());` 处抛异常，导致连接失败。

•自定义 `HostnameVerifier`，简单的话就是根据域名进行字符串匹配校验；业务复杂的话，还可以结合配置中心、白名单、黑名单、正则匹配等多级别动态校验；总体来说逻辑还是比较简单的，反正只要正确地实现那个方法。

```
1  HostnameVerifier hnv = new HostnameVerifier() {
2      @Override
3      public boolean verify(String hostname, SSLSession session) {
4          //示例
5          if("yourhostname".equals(hostname)){
6              return true;
7          } else {
8              HostnameVerifier hv =
9                  HttpURLConnection.getDefaultHostnameVerifier();
10             return hv.verify(hostname, session);
11         }
12     }
13 };
14
```

•主机名验证策略改成严格模式

```
1  SSLSocketFactory sf = new MySSLSocketFactory(trustStore);
2  sf.setHostnameVerifier(SSLSocketFactory.STRICT_HOSTNAME_VERIFIER);
```

java中，有哪些主要的类参与该过程

在 Java 中，参与 SSL/TLS 通过程的主要类包括：

1. `SSLContext`：表示一个 SSL/TLS 上下文，用于管理证书、密钥和加密算法等信息。
2. `SSLSocket`：表示一个支持 SSL/TLS 的套接字，用于建立和管理 SSL/TLS 连接。
3. `SSLSocketFactory`：表示一个 SSL/TLS 套接字工厂，用于创建 `SSLSocket` 对象。
4. `KeyManagerFactory`：用于创建密钥管理器，用于管理证书和密钥。
5. `TrustManagerFactory`：用于创建信任管理器，用于管理证书的信任关系。
6. `HostnameVerifier`：用于验证服务器的主机名是否与证书中的主机名匹配。
7. `X509TrustManager`：用于管理证书的信任关系，并决定是否信任服务器的证书。

这些类协同工作，共同实现了 SSL/TLS 通信的建立和管理。通过使用这些类，开发人员可以轻松地在 Java 应用程序中实现 SSL/TLS 通信，保证数据的安全性。

如何给客户端配置其所需的权限

在 Keycloak 中，您可以通过以下步骤为客户端配置其所需的权限：

1. 登录到 Keycloak 管理界面。
2. 在左侧菜单中选择 "Clients"。
3. 找到您要配置权限的客户端，并点击其名称。
4. 在 "Client详情" 页面中，找到 "Scopes" 部分。
5. 点击 "Add Scope" 按钮，添加一个新的权限范围。
6. 在 "New Scope" 页面中，输入权限范围的名称，并选择要**授予该范围的权限**。
7. 点击 "Save" 按钮保存权限范围。
8. 重复步骤 5-7，为客户端添加所需的所有权限范围。
9. 在 "Client详情" 页面中，找到 "Authorization" 部分。
10. 选择 "GrantedScopes" 选项，并选择您为客户端配置的权限范围。
11. 点击 "Save" 按钮保存客户端的权限配置。



这样，客户端就可以使用配置的权限范围访问受保护的资源。

这些权限跟服务端的springboot是如何协作的

Keycloak 和 Spring Boot 可以通过多种方式进行协作，具体取决于您的架构和需求。

一种常见的方式是使用 Keycloak 的 Spring Security Adapter，它可以将 Keycloak 的身份验证和授权功能集成到 Spring Boot 应用程序中。通过这种方式，您可以使用 Keycloak 管理用户和角色，并将其映射到 Spring Boot 中的 Spring Security 认证和授权框架。

另一种方式是直接使用 Keycloak 的 API 进行身份验证和授权。您可以在 Spring Boot 应用程序中编写代码来获取令牌，并在需要时使用令牌进行授权。这种方式更灵活，但需要更多的开发工作。

无论您选择哪种方式，Keycloak 和 Spring Boot 都可以很好地协作，以提供安全的身份验证和授权功能。

要实现使用 Keycloak 作为授权服务器，资源服务器（如 Spring Boot 应用）需要进行以下配置：

1. 引入 Keycloak Spring Security Adapter 依赖：在 pom.xml 文件中添加以下依赖：

<!-->

```
plaintext
<dependency>
  <groupId>org.keycloak</groupId>
  <artifactId>keycloak-spring-security-adapter</artifactId>
  <version>${keycloak.version}</version>
</dependency>
```

2. 配置 Keycloak Spring Security Adapter：在 Spring Boot 应用的配置文件中添加以下配置：

<!-->

```
plaintext
keycloak:
  realm: your-realm
  url: http://localhost:8080/auth
  client:
    id: your-client
    secret: your-secret
```

3. 配置 SecurityFilterChain：在 Spring Security 配置文件中添加以下配置：

<!-->

3. 配置 SecurityFilterChain：在 Spring Security 配置文件中添加以下配置：

<!-->

```
plaintext
http.antMatcher("/api/**").authorizeHttpRequests().keycloak()
```

在上面的配置中，`keycloak()` 方法会将所有匹配 `/api/**` 路径的请求委托给 Keycloak 进行认证和授权。

4. 编写 API 控制器：在 Spring Boot 应用中编写 API 控制器，确保在每个请求中都检查 Keycloak 颁发的令牌是否有效。例如：

<!-->

```
plaintext
@RestController
public class ExampleController {

  @KeycloakSecurityContext
  private SecurityContext securityContext;

  @GetMapping("/api")
  public String getExample() {
    if (securityContext.getToken() == null || securityContext.getToken().isExpired()) {
      throw new AccessDeniedException("Invalid or expired Token");
    }
    return "Example Response";
  }
}
```

发消息...



<!-->

plaintext

```
@RestController
public class ExampleController {

    @KeycloakSecurityContext
    private SecurityContext securityContext;

    @GetMapping("/api")
    public String getExample() {
        if (securityContext.getToken() == null || securityContext.getToken().isExpired()) {
            throw new AccessDeniedException("Invalid or expired Token");
        }
        return "Example Response";
    }
}
```

在上面的代码中，我们使用了 `@Keycloak SecurityContext` 注解来获取当前请求的 Keycloak 安全上下文。然后，我们在控制器方法中检查令牌是否存在并且是否过期，如果令牌无效或过期，我们将抛出一个 `AccessDeniedException` 异常。

请注意，上面的配置和

接续

上面的配置和代码示例仅提供了一个基本的实现，你可能需要根据实际需求进行调整。例如，你可能需要配置 Keycloak 以验证客户端的身份，配置 API 控制器以处理不同的授权请求等。

此外，为了方便测试，你可以在本地启动 Keycloak 服务器，并使用 Postman 或其他 API 测试工具模拟客户端应用程序的请求。在请求中，你需要提供有效的令牌，否则资源服务器将拒绝访问。

发消息...



如果你想在 Keycloak 中获取 SecurityContext，可以使用 Keycloak 提供的 SecurityContext API。例如，你可以使用 `KeycloakPrincipal` 和 `KeycloakSecurityContext` 来获取当前登录用户的信息和 SecurityContext。

以下是一个示例代码，演示如何使用 SecurityContext API 来获取 Keycloak SecurityContext：

plaintext

```
import org.keycloak.KeycloakPrincipal;
import org.keycloak.KeycloakSecurityContext;

@RestController
public class ExampleController {

    @GetMapping("/api")
    public String getExample() {
        KeycloakPrincipal principal = (KeycloakPrincipal) SecurityContext.getAuthentication().get
        KeycloakSecurityContext securityContext = principal.getKeycloakSecurityContext();

        // 获取 SecurityContext 中的令牌和用户信息
        String token = securityContext.getToken().getTokenString();
        String username = securityContext.getToken().getUsername();
    }
}
```

spring security 具体转发给了 keycloak 的那个类，

在 Spring Security 中，请求是通过 FilterChainProxy 转发给 Keycloak 的。FilterChainProxy 是 Spring Security 的核心组件之一，它负责管理请求的过滤和转发。

当请求到达 FilterChainProxy 时，FilterChainProxy 会检查请求是否需要认证或授权，如果需要，它会将请求转发给 Keycloak。在转发请求之前，FilterChainProxy 会将请求头中的用户信息（如用户名和密码）传递给 Keycloak，以便 Keycloak 能够进行认证。

具体来说，FilterChainProxy 会调用 Keycloak 的 AuthenticationManager 类来进行认证。

AuthenticationManager 是 Keycloak 提供的一个接口，用于处理认证请求。FilterChainProxy 会将请求和用户信息传递给 AuthenticationManager，然后等待 AuthenticationManager 返回认证结果。

如果认证成功，FilterChainProxy 会将请求继续转发到应用程序的其他过滤器进行处理；如果认证失败，FilterChainProxy 会返回一个 401 响应，指示用户认证失败。

需要注意的是，具体的转发方式和 Keycloak 的实现可能会有所不同，但 FilterChainProxy 是 Spring Security 中处理请求过滤和转发的核心组件。

这个默认的处理过程，都用到了哪些具体的类

在 Spring Security 的默认处理过程中，使用了多个具体的类来实现安全控制。以下是一些主要的类：

1. FilterChainProxy：这是一个过滤器链代理，它负责将请求传递给不同的过滤器进行处理。它是 Spring Security 的核心组件，用于实现请求的过滤和转发。
2. FilterInvocationSecurityMetadataSource：这是一个安全元数据源，它提供了请求的安全信息，例如请求的目标资源、请求的方法等。
3. FilterInvocationSecurityMetadataAdapters：这是一个适配器类，它将不同的安全元数据源转换为 FilterInvocationSecurityMetadataSource 可以使用的格式。
4. FilterSecurityInterceptor：这是一个过滤器，它负责在请求到达目标资源之前进行安全检查。它使用 SecurityMetadataSource 提供的安全信息来确定请求是否合法，并根据配置的安全规则进行授权决策。
5. SecurityContextHolder：这是一个上下文容器，它存储了当前登录的用户信息和其他安全相关的信息。它可以用于在整个请求处理过程中访问当前登录的用户信息。
6. UserDetailsService：这是一个接口，它提供了获取用户详细信息的方法，例如用户名、密码、权限等。Spring Security 使用这个接口来获取用户认证信息。

这些类共同协作，实现了 Spring Security 的安全控制功能。通过配置和扩展这些类，开发人员可以根据自己的需求实现更加复杂的安全控制。


```
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer

If you have multiple classes that should be added you can use "," to separate the values. For example:
org.springframework.security.config.annotation.web.configurers.
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer, sample.
OtherThatExtendsAbstractHttpConfigurer

Deprecated Use a org.springframework.security.web.SecurityFilterChain Bean to configure HttpSecurity or a WebSecurityCustomizer Bean to configure WebSecurity.
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authz ->
            authz.anyRequest().authenticated()
        );
    // ...
    return http.build();
}

@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
    return (web) -> web.ignoring().antMatchers("/resources/**");
}

See the Spring Security without WebSecurityConfigurerAdapter for more details.
Author: Rob Winch

113 @Order(100)
114 @Deprecated
115 public abstract class WebSecurityConfigurerAdapter implements WebSecurityConfigurer<WebSecurity> {
116
117     private final Log logger = LoggerFactory.getLogger(WebSecurityConfigurerAdapter.class);
118
119     private ApplicationConfig config;
```

```
org.springframework.security.config.annotation.web.configurers.
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer

If you have multiple classes that should be added you can use "," to separate the values. For example:
org.springframework.security.config.annotation.web.configurers.
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer, sample.
OtherThatExtendsAbstractHttpConfigurer

Deprecated Use a org.springframework.security.web.SecurityFilterChain Bean to configure HttpSecurity or a WebSecurityCustomizer Bean to configure WebSecurity.
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authz ->
            authz.anyRequest().authenticated()
        );
    // ...
    return http.build();
}

@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
    return (web) -> web.ignoring().antMatchers("/resources/**");
}

See the Spring Security without WebSecurityConfigurerAdapter for more details.
Author: Rob Winch

113 @Order(100)
114 @Deprecated
115 public abstract class WebSecurityConfigurerAdapter implements WebSecurityConfigurer<WebSecurity> {
116
117     private final Log logger = LoggerFactory.getLogger(WebSecurityConfigurerAdapter.class);
118
119     private ApplicationConfig config;
```

```
org.springframework.security.config.annotation.web.configurers.
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer

If you have multiple classes that should be added you can use "," to separate the values. For example:
org.springframework.security.config.annotation.web.configurers.
AbstractHttpConfigurer = sample.MyClassThatExtendsAbstractHttpConfigurer, sample.
OtherThatExtendsAbstractHttpConfigurer

Deprecated Use a org.springframework.security.web.SecurityFilterChain Bean to configure HttpSecurity or a WebSecurityCustomizer Bean to configure WebSecurity.
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authz ->
            authz.anyRequest().authenticated()
        );
    // ...
    return http.build();
}

@Bean
public WebSecurityCustomizer webSecurityCustomizer() {
    return (web) -> web.ignoring().antMatchers("/resources/**");
}

See the Spring Security without WebSecurityConfigurerAdapter for more details.
Author: Rob Winch

113 @Order(100)
114 @Deprecated
115 public abstract class WebSecurityConfigurerAdapter implements WebSecurityConfigurer<WebSecurity> {
116
117     private final Log logger = LoggerFactory.getLogger(WebSecurityConfigurerAdapter.class);
118
119     private ApplicationConfig config;
```

resources/META-INF/spring.factories 是springboot starter 的工作原理

resources/META-INF/spring.factories 文件是 Spring Boot Starter 的核心。该文件包含一个或多个键值对，其中键是 Starter 的名称，值是一个包含 Starter 相关配置类路径。

当 Spring Boot 启动时，它会加载 META-INF/spring.factories 文件，并解析其中的键值对。对于每个 Starter，它会加载对应的配置类，并将其配置应用于应用程序上下文。

具体来说，META-INF/spring.factories 文件中的每个键值对都对应于一个 org.springframework.boot.autoconfigure.EnableAutoConfiguration 属性。该属性的值是一个包含自动配置类的类路径，这些自动配置类用于在应用程序上下文中注册和配置各种 bean。

例如，如果您在 META-INF/spring.factories 文件中添加了一个名为 your-starter 的键值对，其值为 com.example.YourStarterAutoConfiguration，那么当您在 Spring Boot 项目中添加了 your-starter Starter 依赖时，Spring Boot 将自动加载 com.example.YourStarterAutoConfiguration 类，并将其配置应用于应用程序上下文。

通过这种方式，Spring Boot Starter 可以方便地将相关的依赖和配置打包在一起，并在应用程序启动时自动应用这些配置，从而简化了 Spring Boot 应用程序的开发和配置过程。