Keycloak中：common/integration/admin-client/src/main/jaba/org

Keycloak的协议代码：server-spi/src/main/java/sessions

Services/src/main/java/org/keycloak/authentication/authenticators/client

Keycloak入口地址： quarkus/runtime/src/main/java/org/keycloak/quarkus/runtime/keycloakMain.java

在 KeycloakMain.java 中，重载后的 main(String[] args, Picocli picocli) 方法是 Keycloak Quarkus 版本启动流程的核心命令行参数解析与调度入口。它的主要职责是解析命令行参数、处理特殊启动模式（如 dry-run、fast-start）、并最终调用 Picocli 执行对应命令

**KeycloakMain -> picocli（解析命令，在picocli中的parseAndRun中执行 cmd.execute(argArray)**

在 Picocli 类中，命令解析后会在 parseAndRun(List<String> cliArgs) 方法里执行命令。具体流程如下：

先用 CommandLine cmd = createCommandLine(...) 创建命令行对象并解析参数。

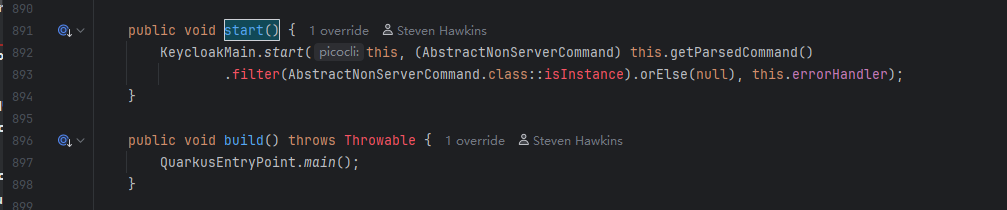
解析后，调用 cmd.execute(argArray) 执行命令，返回 exit code。

执行完毕后，调用 exit(exitCode) 退出进程。

核心执行点就是 cmd.execute(argArray)，它会根据解析到的命令和参数，自动调用对应命令类（如 Start）的 call() 或 run() 方法，完成命令的实际执行。

Cmd.execute是在CommandLine中定义的

keyCloak入口：picocli.java中



quarkus/runtime/src/main/java/org/keycloak/quarkus/runtime/cli/command 定义了一些cli命令

AbstractCommand: 是 Keycloak CLI 命令体系的抽象基类，定义了所有命令的通用结构和行为

quarkus/runtime/src/main/java/org/keycloak/quarkus/runtime/cli/command/BootstrapAdminService.java

`BootstrapAdminService.java` 是 Keycloak CLI 中用于添加临时管理员服务账号的命令实现类，主要内容和实现逻辑如下：

1. \*\*类定义与注解\*\*

- 使用 `@Command` 注解，声明命令名为 `service`，并设置命令头部和描述。

- 继承自 `AbstractNonServerCommand`，表示该命令不启动 HTTP 服务，仅做配置和数据操作。

2. \*\*参数定义\*\*

- 内部类 `ClientIdOptions` 通过 `@Option` 注解定义了 `--client-id` 和 `--client-id:env` 参数，分别用于直接指定 client id 或通过环境变量获取。

- `@ArgGroup` 实现参数互斥（只能二选一）。

- `--client-secret:env` 用于通过环境变量获取 client secret。

- 还定义了 `clientSecret` 和 `clientId` 字段用于存储最终值。

3. \*\*命令名称\*\*

- 重写 `getName()`，返回命令名 `service`。

4. \*\*命令执行前逻辑（doBeforeRun）\*\*

- 处理 client id 和 client secret 的获取逻辑：

- 优先从参数或环境变量获取。

- 如果未指定且允许交互，则通过控制台提示用户输入。

- client secret 需要二次确认且不能为空，否则抛出 `PropertyException`。

5. \*\*环境变量读取\*\*

- `getFromEnv(String envVar)` 方法用于从系统环境变量获取值，未找到时报错。

6. \*\*命令启动后逻辑（onStart）\*\*

- 在 Keycloak 应用启动后，调用 `createTemporaryMasterRealmAdminService` 创建临时管理员服务账号。

7. \*\*选项分类过滤\*\*

- 重写 `excludedCategories()`，排除导入、导出和自身（BOOTSTRAP\_ADMIN）相关的选项。

\*\*总结\*\*：

该类实现了 Keycloak CLI 的 `service` 子命令，主要用于通过命令行或环境变量交互式地创建临时管理员服务账号，适合自动化或初始化场景。参数校验和交互逻辑较为完善，确保安全性和易用性。

quarkus/runtime/src/main/java/org/keycloak/quarkus/runtime/cli/command/BootstrapAdminUser.java

`BootstrapAdminUser.java` 是 Keycloak CLI 中用于添加临时管理员用户的命令实现类，主要实现内容如下：

1. \*\*类定义与注解\*\*

- 使用 `@Command` 注解，声明命令名为 `user`，并设置命令头部和描述。

- 继承自 `AbstractNonServerCommand`，表示该命令只做配置和数据操作，不启动 HTTP 服务。

2. \*\*参数定义\*\*

- 内部类 `UsernameOptions` 通过 `@Option` 注解定义了 `--username`（直接指定用户名）和 `--username:env`（通过环境变量获取用户名），二者互斥。

- `@ArgGroup` 实现参数互斥逻辑。

- `--password:env` 用于通过环境变量获取管理员密码。

- 字段 `username` 和 `password` 用于存储最终用户名和密码。

3. \*\*命令名称\*\*

- 重写 `getName()`，返回命令名 `user`。

4. \*\*命令执行前逻辑（doBeforeRun）\*\*

- 处理用户名和密码的获取逻辑：

- 用户名优先从参数或环境变量获取，否则在允许交互时通过控制台输入。

- 密码优先从环境变量获取，否则在允许交互时通过控制台输入并要求二次确认，且不能为空，否则抛出 `PropertyException`。

5. \*\*环境变量读取\*\*

- `getFromEnv(String envVar)` 方法用于从系统环境变量获取值，未找到时报错。

6. \*\*命令启动后逻辑（onStart）\*\*

- 在 Keycloak 应用启动后，调用 `createTemporaryMasterRealmAdminUser` 创建临时管理员用户。

7. \*\*选项分类过滤\*\*

- 重写 `excludedCategories()`，排除导入、导出和自身（BOOTSTRAP\_ADMIN）相关的选项。

\*\*总结\*\*：

该类实现了 Keycloak CLI 的 `user` 子命令，主要用于通过命令行或环境变量交互式地创建临时管理员用户，适合自动化或初始化场景。参数校验和交互逻辑完善，确保安全性和易用性。

**在 Keycloak 项目中，OAuth 核心部分的代码主要集中在以下几个包和文件夹中：**

1. `org.keycloak.protocol.oidc`

这是实现 OAuth 2.0 和 OpenID Connect（OIDC）协议的核心包。包括认证、授权、Token 颁发、校验等主要逻辑。

2. `services/src/main/java/org/keycloak/protocol/oidc`

这里包含了 OIDC 相关的端点（如 `/token`、`/authorize`、`/userinfo`）、协议处理器、Token 生成与校验等实现。

3. `core/src/main/java/org/keycloak/protocol`

协议抽象层，定义了协议 SPI（Service Provider Interface），便于扩展和自定义协议实现。

4. `core/src/main/java/org/keycloak/representations/oidc`

包含 OIDC/OAuth2 相关的数据结构（如 Token、Claims、认证请求/响应等）。

5. `services/src/main/java/org/keycloak/services/resources`

这里有 OAuth2/OIDC 相关的 REST 资源类，如 `TokenEndpoint`、`AuthorizationEndpoint` 等。

\*\*总结\*\*：

如果你要查找 OAuth2/OIDC 的核心实现，重点关注 `org.keycloak.protocol.oidc` 及其在 `services` 和 `core` 模块下的相关实现。

services/src/main/java/org/keycloak/protocol/oidc/AccessTokenIntrospectionProvider.java

主要实现了 Keycloak OIDC 协议下的 Token Introspection（令牌自省）功能。它负责校验、解析、转换访问令牌（AccessToken），并生成符合 OAuth2 Introspection 标准的响应。核心实现内容如下：

services/src/main/java/org/keycloak/protocol/oidc/AccessTokenIntrospectionProviderFactory.java

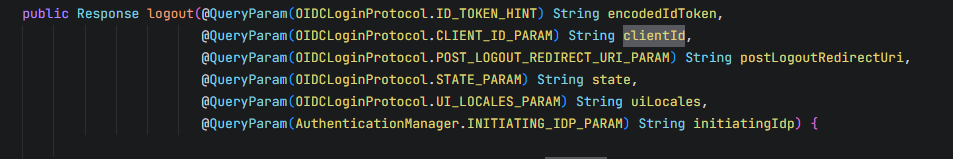
是 Keycloak OIDC 协议中 Token Introspection（令牌自省）功能的工厂类，主要作用是为访问令牌（Access Token）自省端点提供具体的 Provider 实例。其实现内容如下：

services/src/main/java/org/keycloak/protocol/oidc/BackchannelLogoutResponse.java

主要用于描述 OIDC Backchannel Logout（后端通道登出）流程中的响应结果，便于服务端记录和处理登出操作的执行情况。

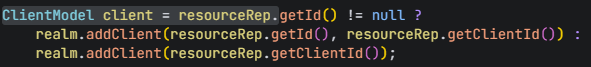
Client\_id绑定Code错误

Client\_id是String类型



分配Client\_id的语句

keycloak\server-spi-private\src\main\java\org\keycloak\models\utils\RepresentationToModel.java line334



Client\_id定义的语句

Keycloak\keycloak\core\src\main\java\org\keycloak\representations\idm\ClientRepresentation.java line31 此文件定义了客户端的一些基本属性和访问接口 其中还包括重定向的URL

public class ClientRepresentation {

    protected String id;

    protected String clientId;

    protected String name;

可知client\_id是string类型，一般String进行匹配时使用的是

接下来去找将client\_id与code绑定时的代码

 // Standard or hybrid flow

        String code = null;

        if (responseType.hasResponseType(OIDCResponseType.CODE)) {

            OAuth2Code codeData = new OAuth2Code(UUID.randomUUID().toString(),

                Time.currentTime() + userSession.getRealm().getAccessCodeLifespan(),

                nonce,

                authSession.getClientNote(OAuth2Constants.SCOPE),

                authSession.getClientNote(OIDCLoginProtocol.REDIRECT\_URI\_PARAM),

                authSession.getClientNote(OIDCLoginProtocol.CODE\_CHALLENGE\_PARAM),

                authSession.getClientNote(OIDCLoginProtocol.CODE\_CHALLENGE\_METHOD\_PARAM),

                authSession.getClientNote(OIDCLoginProtocol.DPOP\_JKT),

                userSession.getId());

            code = OAuth2CodeParser.persistCode(session, clientSession, codeData);

            redirectUri.addParam(OAuth2Constants.CODE, code);

        }

生成一个code对象，调用Oauth2CodeParser.persistCode绑定住clientSession，ClientSession中包含

public static String persistCode(KeycloakSession session, AuthenticatedClientSessionModel clientSession, OAuth2Code codeData) {

        SingleUseObjectProvider codeStore = session.singleUseObjects();

        String key = codeData.getId();

        if (key == null) {

            throw new IllegalStateException("ID not present in the data");

        }

        Map<String, String> serialized = codeData.serializeCode();

        codeStore.put(key, clientSession.getUserSession().getRealm().getAccessCodeLifespan(), serialized);

        return key + "." + clientSession.getUserSession().getId() + "." + clientSession.getClient().getId();

    }

codeData.getId() 授权码的唯一标识符（随机生成）

clientSession.getUserSession().getId() 用户会话的唯一标识符

clientSession.getClient().getId() Client\_id

**验证客户端身份匹配的代码在OAuth2CodeParser类的parseCode中**

String[] parsed = DOT.split(code, 3);

// ...

String userSessionId = parsed[1];

String clientUUID = parsed[2];

// ...

result.clientSession = userSession.getAuthenticatedClientSessionByClient(clientUUID);

// ...

String persistedUserSessionId = result.codeData.getUserSessionId();

if (!userSessionId.equals(persistedUserSessionId)) {

    logger.warnf("Code '%s' is bound to a different session", codeUUID);

    return result.illegalCode();

}

这部分代码通过以下步骤验证客户端身份匹配： equals()

将授权码分割为三部分： codeUUID 、 userSessionId 和 clientUUID

使用 userSession.getAuthenticatedClientSessionByClient(clientUUID) 获取客户端会话

验证解析出的 userSessionId 与存储在授权码数据中的 persistedUserSessionId 是否匹配

客户端会话对象（ClientModel）初始化时会绑定client\_id和（通过在初始化对象时传入ClientModel类型的对象client进行绑定，）

public AuthenticatedClientSessionAdapter(KeycloakSession kcSession,

                                             AuthenticatedClientSessionEntity entity, ClientModel client, UserSessionModel userSession,

                                             SessionsChangelogBasedTransaction<UUID, AuthenticatedClientSessionEntity> clientSessionUpdateTx, boolean offline) {

        if (userSession == null) {

            throw new NullPointerException("userSession must not be null");

        }

        this.kcSession = kcSession;

        this.entity = entity;

        this.userSession = userSession;

        this.client = client;

        this.clientSessionUpdateTx = clientSessionUpdateTx;

        this.offline = offline;

    }

ClientModel接口中定义了一个方法getClientId()用于获取Client\_id

该方法在ClientAdapter.java中实现，从客户端模型中的缓存适配器中读取

public String getClientId() {

        if (isUpdated()) return updated.getClientId();

        return cached.getClientId();

    }

Cache是从数据库中提取client\_id

ClientResource类

传入客户端的注册请求rep，构造ClientModel对象，rep中包含client\_id

ClientModel clientModel = ClientManager.createClient(session, realm, rep);

RepresentationToModel类中 使用resourceRep中的client\_id构造客户端对象

 ClientModel client = resourceRep.getId() != null ? realm.addClient(resourceRep.getId(), resourceRep.getClientId()) : realm.addClient(resourceRep.getClientId());

JpaRealmProvider类 将客户端信息存储到数据库中

public ClientModel addClient(RealmModel realm, String id, String clientId) {

        ClientModel resource;

        if (id == null) {

            id = KeycloakModelUtils.generateId();

        } else if (id.length() > ClientEntity.ID\_MAX\_LENGTH){

            throw new ModelValidationException("Client ID must not exceed 36 characters");

        }

        if (clientId == null) {

            clientId = id;

        }

        logger.tracef("addClient(%s, %s, %s)%s", realm, id, clientId, getShortStackTrace());

        ClientEntity entity = new ClientEntity();

        entity.setId(id);

        entity.setClientId(clientId);

        entity.setEnabled(true);

        entity.setStandardFlowEnabled(true);

        entity.setRealmId(realm.getId());

        em.persist(entity); //写入数据库

        resource = toClientModel(realm, entity);

        session.getKeycloakSessionFactory().publish((ClientModel.ClientCreationEvent) () -> resource);

        return resource;

    }

最终结论：不存在路径混淆类似的攻击能成功，因为是每一步传递client\_id的过程都是通过传值方式传的，并且中间有一步会验证client\_id在系统中的唯一性

授权码单次使用

AuthorizationCodeGrantType类中

isIllegalCode()判断是否是非法Code

OAuth2CodeParser.ParseResult parseResult = OAuth2CodeParser.parseCode(session, code, realm, event);

        if (parseResult.isIllegalCode()) {

            AuthenticatedClientSessionModel clientSession = parseResult.getClientSession();

            // Attempt to use same code twice should invalidate existing clientSession

            if (clientSession != null) {

                clientSession.detachFromUserSession();

            }

            event.error(Errors.INVALID\_CODE);

            throw new CorsErrorResponseException(cors, OAuthErrorException.INVALID\_GRANT, "Code not valid", Response.Status.BAD\_REQUEST);

        }

找到isIllegalCode()定义 属于OAuth2CodeParser类

public boolean isIllegalCode() {

            return isIllegalCode;

        }

将Code设置成非法状态 调用其即可

private ParseResult illegalCode() {

            this.isIllegalCode = true;

            return this;

        }

检查授权码是否已经使用 若已经使用则调用illegalCode（）

SingleUseObjectProvider codeStore = session.singleUseObjects();

        Map<String, String> codeData = codeStore.remove(codeUUID);

        // Either code not available or was already used

        if (codeData == null) {

            logger.warnf("Code '%s' already used for userSession '%s' and client '%s'.", codeUUID, userSessionId, clientUUID);

            return result.illegalCode();

        }

SingleUseObjectProvider接口

是 Keycloak 中用于管理单次使用对象的核心接口，定义了安全存储和检索只能使用一次的敏感数据的标准方法

重定向URL

ClientRepresentation类存放了客户端登记时的URL redirectUris

然后通过ClientManager.createClient方法调用RepresentationToModel.createClient创建客户端

在 RepresentationToModel.updateClientProperties 方法中，URL 被设置到 ClientModel 对象

Builder

思考过程

为了回答重定向URL如何从客户端传递到服务器的问题，我需要梳理Keycloak中授权码流程中重定向URL的处理路径。从之前的分析中，我已经知道在``AuthorizationCodeGrantType.java`` 中会验证重定向URL，但我需要了解完整的传递流程：

1. 客户端如何设置重定向URL

2. 重定向URL如何在授权请求中传递

3. 服务器如何接收、存储和验证重定向URL

让我搜索与授权请求处理相关的代码，特别是处理重定向URL的部分。我可以使用``search\_by\_regex`` 工具搜索"redirectUri"或"redirect\_uri"相关的代码，重点关注授权端点和客户端注册相关的实现。

为了梳理重定向URL从客户端到服务器的传递流程，我需要查找授权请求处理、重定向URL存储和验证相关的代码。让我搜索与redirectUri相关的关键实现。

在工作区搜索 'redirectUri|redirect\_uri'

搜索结果显示了许多与重定向URL相关的消息文件和测试工具类，但我需要找到处理重定向URL传递流程的核心代码。让我搜索更具体的关键词，如授权端点实现、redirect\_uri参数处理和验证逻辑。

在工作区搜索 'AuthorizationEndpoint|validateRedirectUri|redirect\_uri\s\*=|redirectUri\s\*\('

搜索结果显示了许多与重定向URL相关的测试文件和工具类，但仍未找到核心业务逻辑。我注意到测试代码频繁引用AuthorizationEndpointResponse类，这表明授权端点可能是处理重定向URL的关键位置。让我搜索AuthorizationEndpoint的具体实现。

在工作区搜索 'class\s+AuthorizationEndpoint\s\*[{]|public\s+class\s+AuthorizationEndpoint'

找到了AuthorizationEndpoint类的实现，这是处理OAuth 2.0授权端点的核心类，很可能包含重定向URL的处理逻辑。让我查看该类的代码以了解重定向URL的传递流程。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\endpoints\AuthorizationEndpoint.java 1 - 500

文件大小超过限制，我将分部分查看AuthorizationEndpoint.java代码。先查看前200行了解基本结构和主要方法。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\endpoints\AuthorizationEndpoint.java 1 - 200

从已查看的代码可以看出，AuthorizationEndpoint类是处理OAuth 2.0授权端点的核心实现，其中process方法处理授权请求。代码中使用AuthorizationEndpointChecker来检查并重定向URI，并将其存储在redirectUri变量中。我需要继续查看该类的剩余部分，以了解重定向URL的完整处理流程。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\endpoints\AuthorizationEndpoint.java 201 - 400

继续查看代码，我注意到在updateAuthenticationSession方法中有关键代码：authenticationSession.setRedirectUri(redirectUri)，这表明重定向URL被存储在认证会话中。同时还设置了客户端备注OIDCLoginProtocol.REDIRECT\_URI\_PARAM。为了完整了解重定向URL的传递流程，我需要查看剩余的代码部分。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\endpoints\AuthorizationEndpoint.java 401 - 446

现在我已经查看了AuthorizationEndpoint类的完整代码，发现重定向URL的处理主要在process方法和updateAuthenticationSession方法中。为了更深入了解重定向URL的验证和传递机制，我需要查看AuthorizationEndpointChecker类的实现，特别是其中的checkRedirectUri方法。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\endpoints\AuthorizationEndpointChecker.java 1 - 300

在AuthorizationEndpointChecker类中，checkRedirectUri方法调用了RedirectUtils.verifyRedirectUri来验证重定向URI的有效性。为了更深入了解重定向URL的验证机制，我需要查看RedirectUtils类的实现。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\utils\RedirectUtils.java 1 - 300

重定向URL从客户端到服务器的传递流程如下：

1. 1.

客户端发起请求 ：客户端在OAuth 2.0/OpenID Connect授权请求中通过 redirect\_uri 参数传递重定向URL。

2. 2.

请求到达授权端点 ：请求被路由到 AuthorizationEndpoint 类的 process 方法处理。

3. 3.

重定向URL验证 ：

- AuthorizationEndpoint 调用 AuthorizationEndpointChecker.checkRedirectUri() 方法

- 该方法进一步调用 RedirectUtils.verifyRedirectUri() 进行实际验证

- RedirectUtils.resolveValidRedirects() 将客户端注册的相对路径重定向URI转换为绝对路径

- RedirectUtils.matchesRedirects() 检查请求中的 redirect\_uri 是否匹配客户端注册的有效URI

4. 4.

验证通过后存储 ：

- 验证通过的重定向URL被存储在认证会话中： authenticationSession.setRedirectUri(redirectUri)

- 同时设置客户端备注： authenticationSession.setClientNote(OIDCLoginProtocol.REDIRECT\_URI\_PARAM, request.getRedirectUriParam())

5. 5.

后续使用 ：在授权流程完成后，重定向URL会被用于构建授权响应，将用户重定向回客户端应用。

这个流程确保了重定向URL的有效性和安全性，防止了未授权的URL被用于重定向，符合OAuth 2.0和OpenID Connect的安全规范。

相关核心类和方法：

- `AuthorizationEndpoint.java`

- `AuthorizationEndpointChecker.java`

- `RedirectUtils.java`

客户端发起 OAuth请求 在URL参数中携带redirect\_uri

AuthorizationEndpoint类的process方法接受请求，在其中调用AuthorizationEndpointChecker类中的checkRedirectUri方法检查重定向uri

request = AuthorizationEndpointRequestParserProcessor.parseRequest(event, session, client, params, AuthorizationEndpointRequestParserProcessor.EndpointType.OIDC\_AUTH\_ENDPOINT);

        AuthorizationEndpointChecker checker = new AuthorizationEndpointChecker()

                .event(event)

                .client(client)

                .realm(realm)

                .request(request)

                .session(session)

                .params(params);

        try {

            checker.checkRedirectUri();

            this.redirectUri = checker.getRedirectUri();

        } catch (AuthorizationEndpointChecker.AuthorizationCheckException ex) {

            ex.throwAsErrorPageException(authenticationSession);

        }

checkRedirectUri**方法实现** AuthorizationEndpoint

public void checkRedirectUri() throws AuthorizationCheckException {

        String redirectUriParam = request.getRedirectUriParam();

        boolean isOIDCRequest = TokenUtil.isOIDCRequest(request.getScope());

        event.detail(Details.REDIRECT\_URI, redirectUriParam);

        // redirect\_uri parameter is required per OpenID Connect, but optional per OAuth2

        this.redirectUri = RedirectUtils.verifyRedirectUri(session, redirectUriParam, client, isOIDCRequest); //此处核心验证

        if (redirectUri == null) {

            event.error(Errors.INVALID\_REDIRECT\_URI);

            throw new AuthorizationCheckException(Response.Status.BAD\_REQUEST, Messages.INVALID\_PARAMETER, OIDCLoginProtocol.REDIRECT\_URI\_PARAM);

        }

    }

RedirectUtils.verifyRedirectUri源码

public static String verifyRedirectUri(KeycloakSession session, String rootUrl, String redirectUri, Set<String> validRedirects, boolean requireRedirectUri) {

        KeycloakUriInfo uriInfo = session.getContext().getUri();

        RealmModel realm = session.getContext().getRealm();

        if (redirectUri == null) {

            if (!requireRedirectUri) {

                redirectUri = getSingleValidRedirectUri(validRedirects);

            }

            if (redirectUri == null) {

                logger.debug("No Redirect URI parameter specified");

                return null;

            }

        } else if (validRedirects.isEmpty()) {

            logger.debug("No Redirect URIs supplied");

            redirectUri = null;

        } else {

            URI originalRedirect = toUri(redirectUri);

            if (originalRedirect == null) {

                // invalid URI passed as redirectUri

                return null;

            }

            // check if the passed URI allows wildcards

            boolean allowWildcards = areWildcardsAllowed(originalRedirect);

            String r = redirectUri;

            Set<String> resolveValidRedirects = resolveValidRedirects(session, rootUrl, validRedirects);

            String valid = matchesRedirects(resolveValidRedirects, r, allowWildcards);

            if (valid == null && "http".equals(originalRedirect.getScheme()) && LOOPBACK\_INTERFACES.contains(originalRedirect.getHost())) {

                String redirectWithDefaultPort = KeycloakUriBuilder.fromUri(originalRedirect).port(80).buildAsString();

                valid = matchesRedirects(resolveValidRedirects, redirectWithDefaultPort, allowWildcards);

            }

            if (valid != null && !originalRedirect.isAbsolute()) {

                // return absolute if the original URI is relative

                if (!redirectUri.startsWith("/")) {

                    redirectUri = "/" + redirectUri;

                }

                redirectUri = relativeToAbsoluteURI(session, rootUrl, redirectUri);

            }

            String scheme = originalRedirect.getScheme();

            if (valid != null && scheme != null) {

                // check the scheme is valid, it should be http(s) or explicitly allowed by the validation

                if (!valid.startsWith(scheme + ":") && !"http".equalsIgnoreCase(scheme) && !"https".equalsIgnoreCase(scheme)) {

                    logger.debugf("Invalid URI because scheme is not allowed: %s", redirectUri);

                    valid = null;

                }

            }

            redirectUri = valid != null ? redirectUri : null;

        }

        if (Constants.INSTALLED\_APP\_URN.equals(redirectUri)) {

            return Urls.realmInstalledAppUrnCallback(uriInfo.getBaseUri(), realm.getName()).toString();

        } else {

            return redirectUri;

        }

    }

    private static URI toUri(String redirectUri) {

        URI uri = null;

        if (redirectUri != null) {

            try {

                uri = URI.create(redirectUri);

            } catch (IllegalArgumentException cause) {

                logger.debugf(cause, "Invalid redirect uri %s", redirectUri);

            } catch (Exception cause) {

                logger.debugf(cause, "Unexpected error when parsing redirect uri %s", redirectUri);

            }

        }

        return uri;

    }

如果对重定向URI进行匹配，有三种匹配模式，这三种匹配模式都是安全的吗

1. - 精确匹配：完全相等的 URI
2. - 前缀匹配：以 \* 结尾的 URI（如 https://example.com/\* ）
3. - 全局通配符：单个 \* 匹配所有 http/https URI

通配符检查

- 1.

处理空重定向 URI

- 如果 redirectUri 为 null 且 requireRedirectUri 为 false，则尝试从 validRedirects 获取单个有效重定向 URI

- 如果无法获取有效 URI，则返回 null

- 2.

处理空有效重定向集合

- 如果 validRedirects 为空，则设置 redirectUri 为 null

- 3.

URI 格式验证

- 尝试将 redirectUri 解析为 URI 对象，若解析失败则返回 null

- 4.

通配符权限检查

- 通过 areWildcardsAllowed 方法检查 URI 是否允许使用通配符（无用户信息且路径不包含不安全模式）

- 5.

解析相对路径

- 调用 resolveValidRedirects 方法将相对路径的有效重定向 URI 转换为绝对路径

- 6.

重定向 URI 匹配

- 调用 matchesRedirects 方法实现三种匹配模式：

- 精确匹配：完全相等的 URI

- 前缀匹配：以 \* 结尾的 URI（如 https://example.com/\* ）

- 全局通配符：单个 \* 匹配所有 http/https URI

- 7.

回环接口特殊处理

- 对于使用 http 协议的回环接口（localhost、127.0.0.1、[::1]），尝试使用默认端口 80 进行匹配

- 8.

相对 URI 处理

- 如果原始 URI 是相对路径，则转换为绝对路径

- 9.

协议验证

- 确保重定向 URI 使用的协议是 http 或 https

- 10.

已安装应用特殊处理

- 如果 redirectUri 是 Constants.INSTALLED\_APP\_URN ，则返回特定格式的回调 URI

keycloak\services\src\main\java\org\keycloak\protocol\oidc\utils\OAuth2Code.java

OAuth2Code类为授权码Code相关定义

OAuth2Code.java 是Keycloak中表示OAuth 2.0授权码的数据模型类，用于存储授权码相关的所有必要信息。该类位于 org.keycloak.protocol.oidc.utils 包下，实现了授权码的序列化、反序列化和属性访问功能。

核心功能概述

该类主要职责是：

- 封装授权码相关的所有数据（如ID、过期时间、nonce等）

- 提供序列化和反序列化方法，方便在存储和传输中使用

- 提供属性访问方法，供其他组件获取授权码信息

keycloak\services\src\main\java\org\keycloak\protocol\oidc\OIDCLoginProtocol.java

OIDCLoginProtocol.java 是Keycloak中实现OpenID Connect (OIDC) 登录协议的核心类，实现了 LoginProtocol 接口，负责处理OIDC认证流程中的各种请求和响应。

核心功能概述

该类主要处理：

- 用户认证成功后的响应生成（授权码或令牌）

- 认证错误处理

- 前端/后端通道登出

- 重定向URI构建

- 认证要求判断等OIDC协议相关操作

keycloak\services\src\main\java\org\keycloak\protocol\oidc\utils\OAuth2CodeParser.java

OAuth2CodeParser

OAuth2CodeParser.java是Keycloak中处理OAuth 2.0授权码的核心解析器类，位于org.keycloak.protocol.oidc.utils包下。以下是其主要实现内容和方法说明：

### 核心功能

该类负责授权码的生成持久化和解析验证，确保授权码与客户端ID正确绑定，防止未授权使用。

keycloak\services\src\main\java\org\keycloak\protocol\oidc\grants\AuthorizationCodeGrantType.java AuthorizationCodeGrantType

- 1. 实现 OAuth 2.0 授权码授权流程 (Authorization Code Grant)

- 2. 验证授权码的存在性、合法性和有效性

- 3. 验证用户会话和客户端会话的有效性

- 4. 校验重定向 URI 的一致性

- 5. 验证客户端身份和授权流程权限

- 6. 支持 PKCE (Proof Key for Code Exchange) 安全验证

- 7. 支持 DPoP (Demonstration of Proof of Possession) 令牌安全机制

- 8. 触发客户端策略事件

- 9. 更新客户端会话和用户会话状态

- 10. 验证用户对客户端请求作用域的持续同意

- 11. 生成并返回包含访问令牌的响应

keycloak\server-spi\src\main\java\org\keycloak\models\KeycloakSession.java

KeycloakSession.java是Keycloak框架中定义会话管理的核心接口文件，位于 org.keycloak.models 包下。

AuthenticatedClientSessionAdapter类

该类主要负责客户端会话的持久化管理，将会话数据存储到持久化介质（如数据库或缓存）中，并在需要时恢复这些数据。

userSessionModel类

UserSessionModel 是 Keycloak 中表示用户会话的核心接口，定义了用户会话的基本属性、状态管理和客户端会话关联等关键功能。它是用户认证后在系统中存在的会话表示，负责维护用户与客户端之间的会话关系

ClientModel类

ClientModel 是 Keycloak 中表示客户端应用的核心接口，定义了客户端的基本属性、配置和行为。它位于 org.keycloak.models 包中，继承了多个接口以实现丰富的功能。

- getId() 获取客户端内部UUID

- getClientId() / setClientId() : 获取/设置用户定义的客户端ID

- getName() / setName() : 获取/设置客户端名称

- getDescription() / setDescription() : 获取/设置客户端描述

- getType() / setType() : 获取/设置客户端类型

SingleUseObjectProvider接口 定义了管理单次使用对象的核心方法

Remove（String key）方法是一个 原子操作 ，它会同时完成两个动作：

从存储中查找并返回与 codeUUID 关联的数据

将该数据从存储中删除

1. UUID ( getId() )

- 定义 ：客户端的内部唯一标识符，由系统自动生成的 UUID 格式字符串。

- 来源 ：系统在创建客户端时自动生成，无需用户干预。

- 用途 ：主要用于系统内部识别和引用客户端，如数据库存储、内部API调用等。

- 特性 ：

- 全局唯一，确保在整个系统中不会重复

- 不可修改，一旦生成后就固定不变

- 对用户通常不可见，主要用于系统内部操作

- 获取方式 ：通过 getId() 方法获取 2. 客户端 ID ( getClientId() )

- 定义 ：用户定义的客户端标识符，用于在应用程序中标识客户端。

- 来源 ：由用户在创建客户端时手动指定，或通过客户端注册流程生成。

- 用途 ：主要用于客户端与 Keycloak 服务器之间的交互，如 OAuth2/OpenID Connect 认证流程中标识客户端。

- 特性 ：

- 在同一领域（Realm）内唯一

- 可以修改（通过 setClientId() 方法）

- 对用户可见，通常在客户端配置中显示

- 通常是有意义的字符串，便于识别客户端用途

- 获取方式 ：通过 getClientId() 方法获取