

SpringBoot2-Shiro接入统一身份认证CAS-Session/JWT两种方案-多Realm认证

作者:陈护航 时间: 2022年3月19日 单位: USTC-NSRL

单点登录

单点登录, 即Single Sign On(简称**SSO**), 是指在多系统应用群中登录一个系统, 便可在其他所有系统中得到授权而无需再次登录。举例: 淘宝、百度等产品, 用户只需要在一个子产品中登录账户, 其他同公司的产品或平台可自动登录或仅进行简单授权即可登录使用该产品, 用户登录了淘宝就不需要登录天猫; 用户登录了百度网盘就不需要登录百度贴吧。

统一身份认证

为了实现单点登录功能, 那么必定会提及**统一身份认证**, 即Central Authentication Service(简称**CAS**)。

原理详解

注意: 这里你必须要了解cookie、session、token以及http协议等知识。

系统简要介绍

子系统MyServer(也代表我们自己的项目):

http://<IP_MyServer>:<Port>

统一身份认证中心服务CAS Server的服务接口

http://<Domain_CAS_Server>/cas

详细步骤

1. 用户向子系统MyServer发起请求:

GET/POST http://<IP_MyServer>:<Port>/<requestUrl>

2. 子系统过滤器拦截requestUrl的请求

判断请求是否带有**登录凭证**, 此处登录凭证常见有两种形式, 一种是JSESSIONID, 另一种是Token, 对应两种认证方案: 即Session与JSON WEB TOKEN简称JWT或者Token, 后面会分别描述实现方法。如果有该凭证且凭证有效, 放行请求, 否则执行下列步骤。

接下来有两种方案:

2.1 后端直接重定向到

http://<Domain_CAS_Server>/cas/login?service=http://<IP_MyServer>:<Port>/casCallback

2.2 后端重定向到一个前端登录界面, 前端登录界面有控件链接到地址:

http://<Domain_CAS_Server>/cas/login?service=http://<IP_MyServer>:<Port>/casCallback

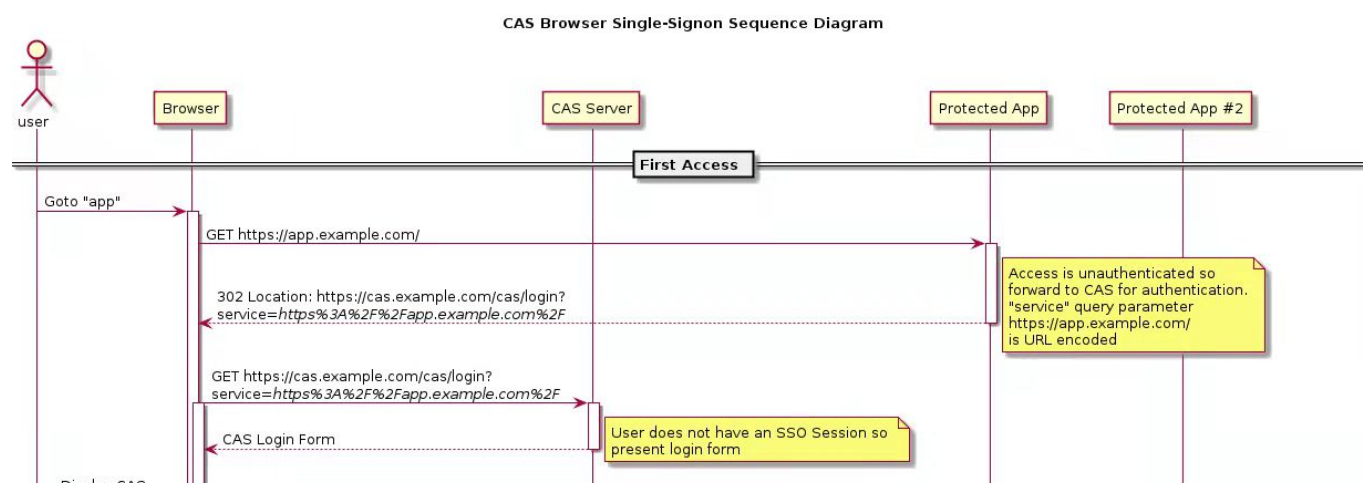
注意: 此处的/casCallback可以自定义, 它代表回调接口, 后面详细介绍它的作用。

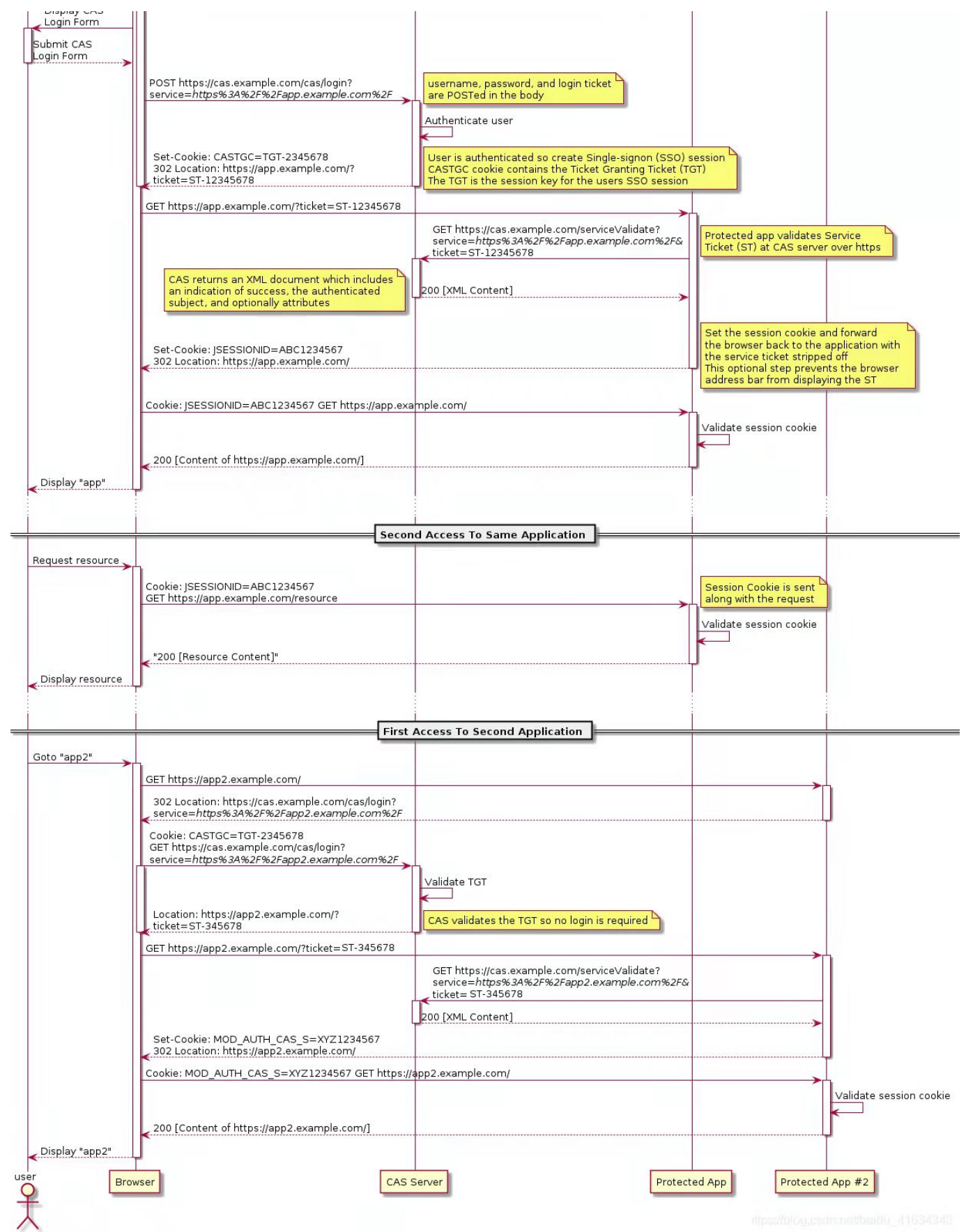
3. 由于上一步重定向, CAS Server会展示统一身份认证登录界面。
4. 用户输入统一身份认证的账号密码并提交给CAS Server。

5. CAS Server读取数据库校验用户密码, 校验成功进行下一步, 否则返回失败信息展示给用户。
6. CAS Server对密码校验成功, 然后为用户签发登录票据即TGT(Ticket Granting Ticket), TGT封装了Cookie值以及此Cookie值对应的用户信息, 保存在CAS Server的缓存中, 进行下一步。并且用户浏览器会存放该cookie, 叫TGC(Ticket-granting cookie), 下一次当用户再次请求到CAS Server时, 如果带有该cookie且未过期, 并且该cookie里面的key值在缓存中查询到TGT, 证明登录过, 不需要重复步骤3、4的登录流程, 进行下一步, 否则需要回到步骤3重新登录。
7. CAS Server会在生成一个票据ST(Service Ticket)并且重定向到子系统的回调接口或地址, 这个接口也就是步骤2里面我们填写的参数service的内容。/casCallback的作用:当用户在CAS Server中登录认证成功后, CAS Server会带着票据ST并重定向到子系统上的/casCallback接口或者针对/casCallback的拦截器上:
`http://<IP_MyServer>:<Port>/casCallback?ticket=<Ticket>`
其中 <Ticket>一般形式为ST-123456xxx。
8. 子系统的/casCallback接口或者专门的拦截器会从上述地址中解析出ticket参数, 然后会拿着这个票据向CAS Server发起验证票据请求:
`GET http://<Domain_CAS_Server>/cas/serviceValidate?service=http://<IP_MyServer>:<Port>/casCallback?ticket=<Ticket>`
CAS Service验证票据有效性, 如果有效则返回用户的信息到子系统, 否则子系统收到验证失败信息。一般这个过程不需要我们手动去实现, 早有前辈为我们封装成TicketValidator, 完成这一步。
9. 子系统得到用户信息后, 进行后续操作:
根据我们的需求进行后续操作。
可以为本地数据库新添加用户信息, 默认分配角色和权限(非必须),
提取用户的角色、权限等信息(非必须),
上述信息保存在session中(非必须), 采用session机制
根据上述信息以及密钥生成token(非必须), 采用token机制
其他操作.....
10. 子系统根据用户信息为用户返回步骤2提及的**登录凭证**, 并重定向到登录界面。
采用session机制: cookie形式保存JSESSIONID, 在子系统后端缓存内对应保存用户信息
重定向到用户登录后界面, 加载该界面时, 前端可以通过返回参数或请求后端拿到用户信息
采用token机制: 前端可以通过返回内容或者重定向的地址参数上获得token, 前端将token保存在Storage里面, 后面的请求都需要在header的Authorization字段内加入token。

具体还可以参见下面两个图:

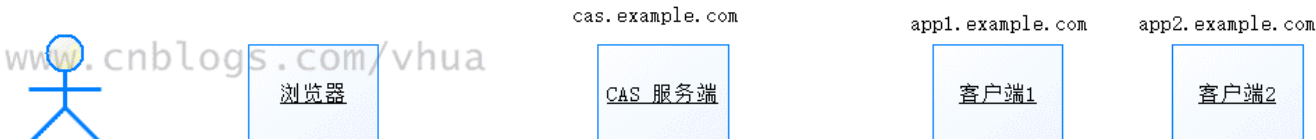
图1

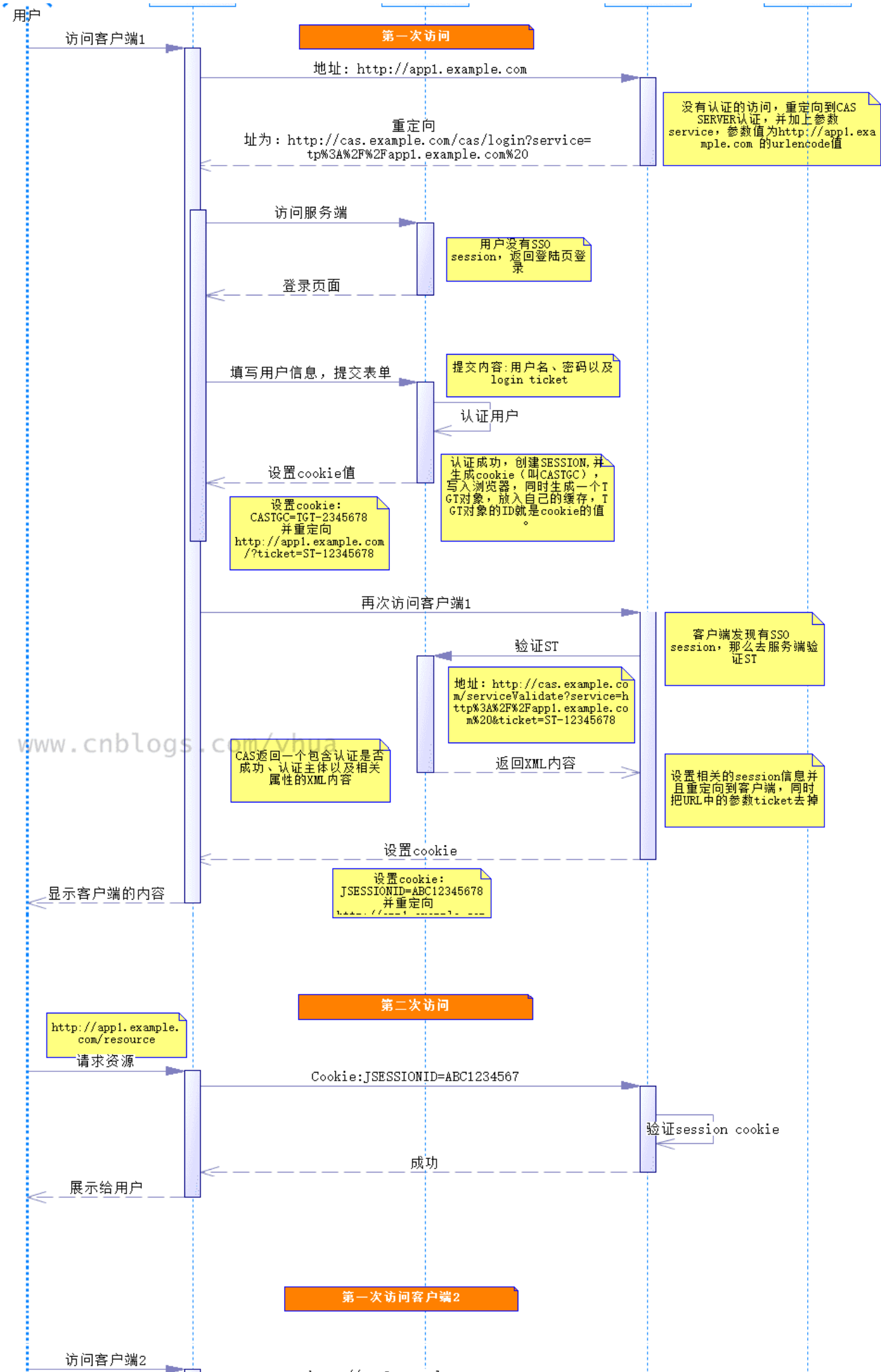


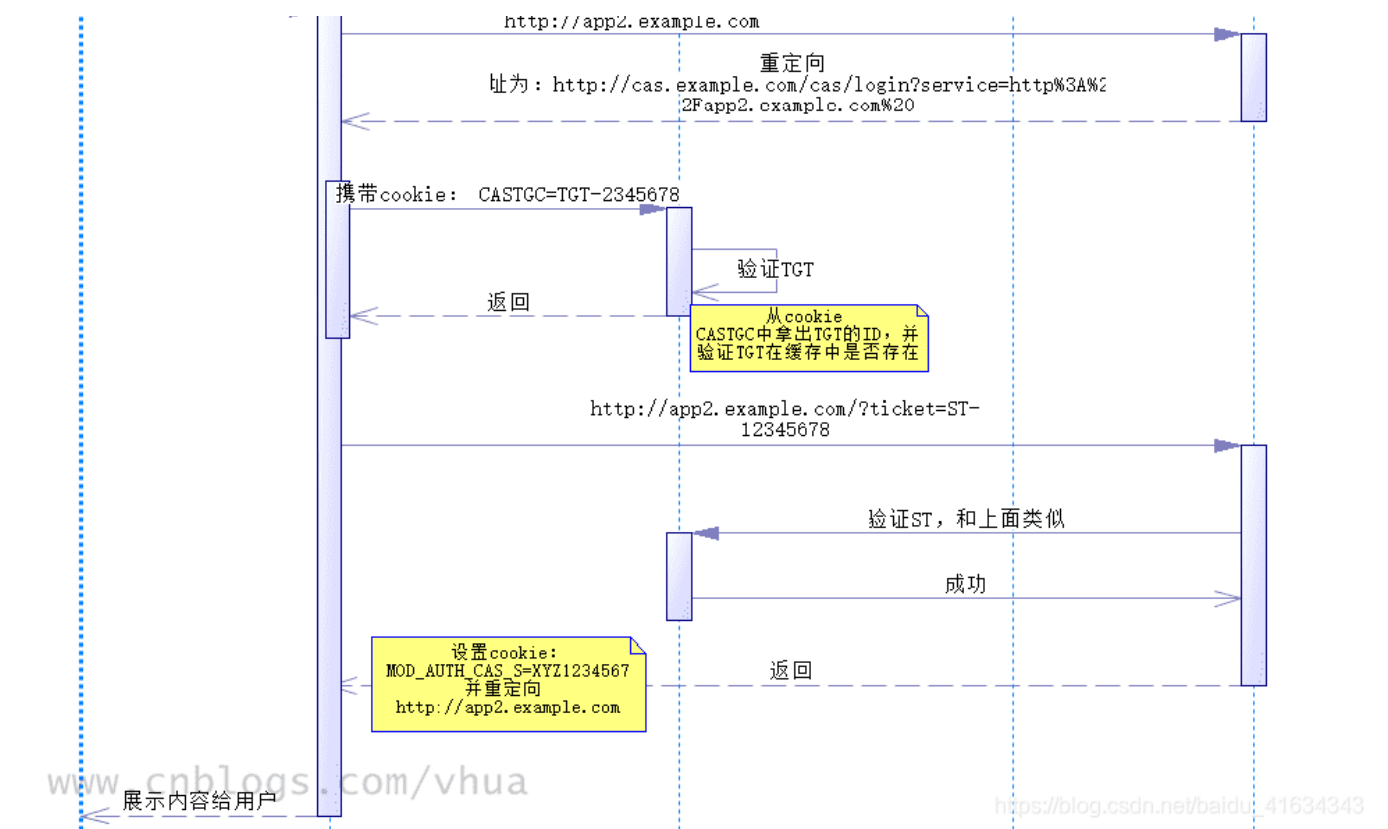


分割线

图2



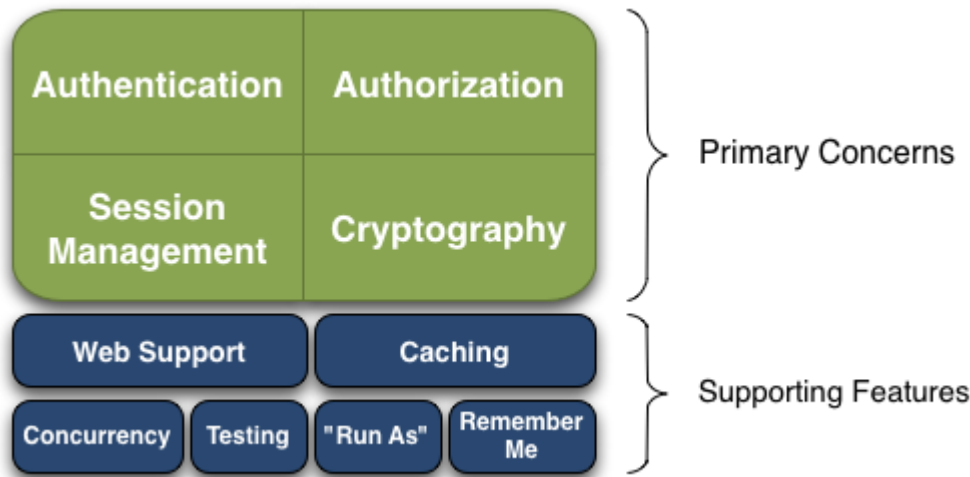




结合Shiro框架实现

Shiro基本介绍

Shiro介绍



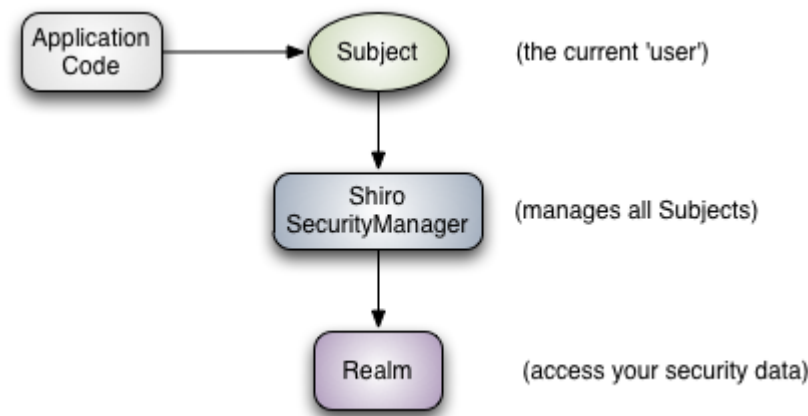
Shiro四个主要功能

- Authentication：身份认证/登录，验证用户账号密码或是否拥有身份；
- Authorization：用户授权，权限验证，判断某个已经认证过的用户是否拥有某些权限访问某些资源，一般授权会有角色授权和权限授权；
- SessionManager：会话管理功能，即用户登录后就是一次会话，在没有退出之前(关闭浏览器之前或会话超时之前)，它的所有信息都在会话中；会话可以是普通JavaSE环境的，也可以是如Web环境的，web环境中作用是和 HttpSession 是一样的；
- Cryptography：加密功能，保护数据的安全性，例如密码加盐取散列值，加密保存到数据库内，而不是明文保存。关于Shiro的详细介绍可以参见：<https://shiro.apache.org/introduction.html>

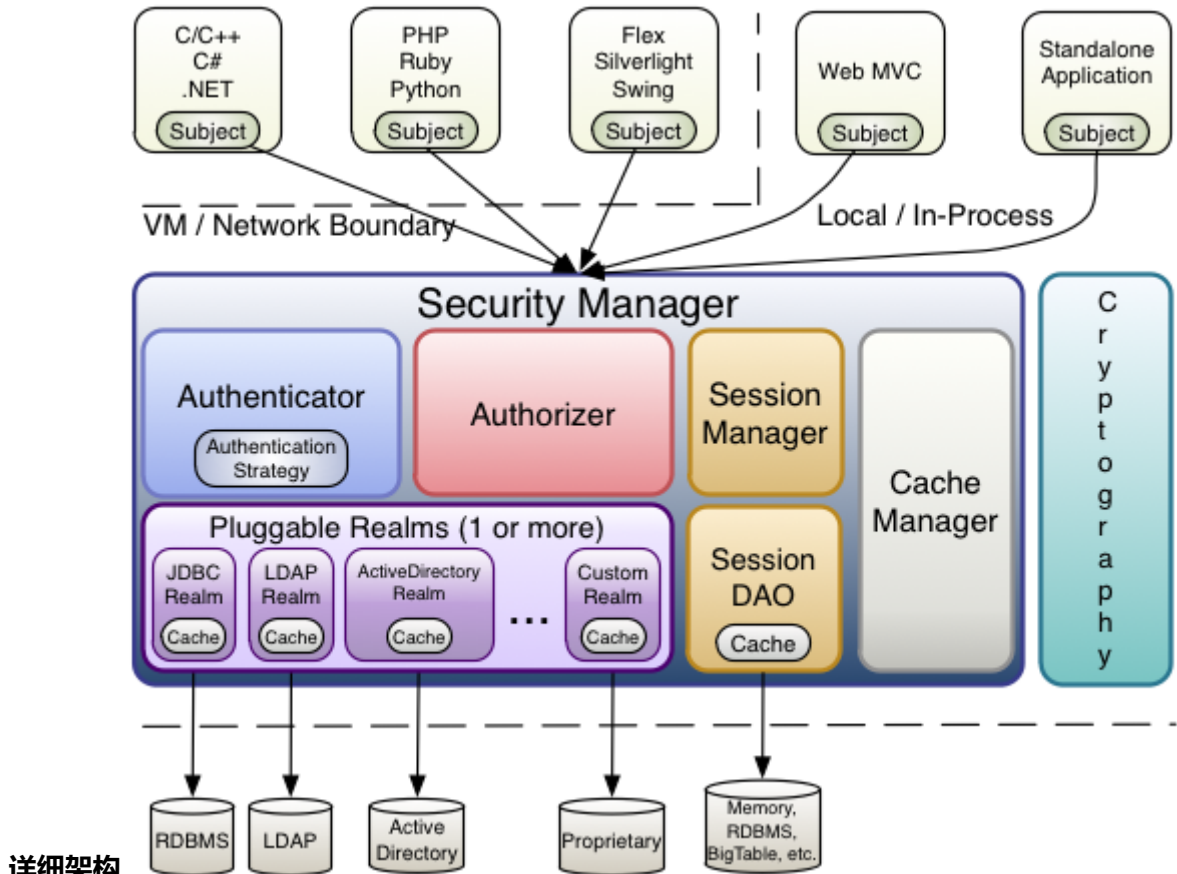
Shiro的基本架构

如果需要充分学习和使用Shiro，其架构是十分重要的内容。

基本架构



- Subject: 访问实体，这个概念比较抽象。如果是用户访问，那么这个实体可以代表用户，但实际上它可以代表任何与软件交互的“东西”。
- SecurityManager: 安全管理器，这是Shiro的核心组件，实际上是由更多更小的组件组成的，这些小组件各个完成不同的功能，例如：CacheManager缓存管理器、SessionManager会话管理器、SubjectFactory实体工厂、Authenticator认证器、Authorizer授权器等组件。醉胡这些组件形成一套安全方案即构成安全管理器。
- Realms: 领域，其实这个翻译并不好。正如官方对它的介绍：本质上就是特定于安全的DAO，充当Shiro与安全数据(用户数据库或者认证授权数据库)的连接桥梁，并且可以针对不同的数据形式不同的访问认证类型有不同的数据获取方案。例如用户输入账户密码登录和统一身份认证登录和短信登录三者登录就通过不同的Realm进行数据读写。



详细架构

此图较为完整描述Shiro架构。额外介绍两个组件：

- Authenticator: 认证器, 是负责执行和响应用户身份验证(登录)尝试的组件。
- - AuthenticationStrategy: 认证策略, 假如有多个不同的Realm, 就应该有不同的认证策略。例如: 手动输入账户密码登录与统一身份认证登录的策略肯定是不同的, 一个是通过密码校对另一个是根据CAS返回的票据和用户信息。
- Authorizer: 授权器, 是负责确定用户在应用程序中的访问控制的组件。它是最终决定用户是否被允许做某事的机制。像 一样, 也知道如何与多个后端数据源协调以访问角色和权限信息。

基本准备

1. pom.xml导入依赖, 版本任意

```

<!--shiro, spring核心依赖-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.4.0</version>
</dependency>
<!--shiro, springboot核心依赖-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring-boot-web-starter</artifactId>
    <version>1.4.0</version>
</dependency>
<!--shiro自带的缓存依赖(可以不加)-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-ehcache</artifactId>
    <version>1.4.0</version>
</dependency>
<!--shiro, 接入Cas统一身份认证的依赖, 后面的版本已被弃用, 该部分转移到Pac4j框架(可以不加)-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-cas</artifactId>
    <version>1.4.0</version>
</dependency>

```

2. 确定CAS服务的相关参数

- CAS服务的服务地址/登录地址/登出地址以及端口号, 一般验证票据的地址都是固定的, 不需要单独写, 推荐使用yml/yaml格式, 这里是使用的默认的properties

```

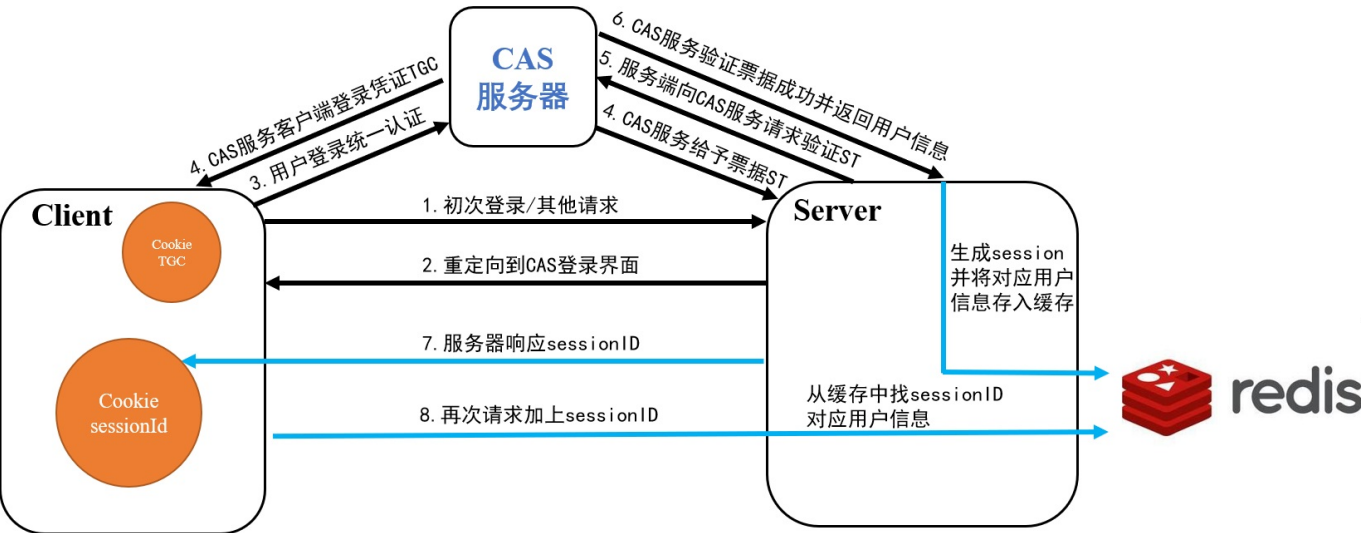
cas.casServer.urlPrefix=https://nsrloa.ustc.edu.cn/cas
cas.casServer.loginUrl=${cas.casServer.urlPrefix}/login
cas.casServer.logoutUrl=https://nsrloa.ustc.edu.cn/sso/logout
cas.client.urlPrefix=http://localhost:8080
cas.client.casCallback=/casCallback
cas.casServer.loginRequestUrl=${cas.casServer.loginUrl}?service=
{cas.client.urlPrefix}{cas.client.casCallback}

```

```
cas.client.successUrl=/user
cas.client.unauthorizedUrl=/unauthorized
cas.client.needLogin=/needLogin
```

方案一——Session认证实现并使用Redis做Session共享

这种方案原理图如下：

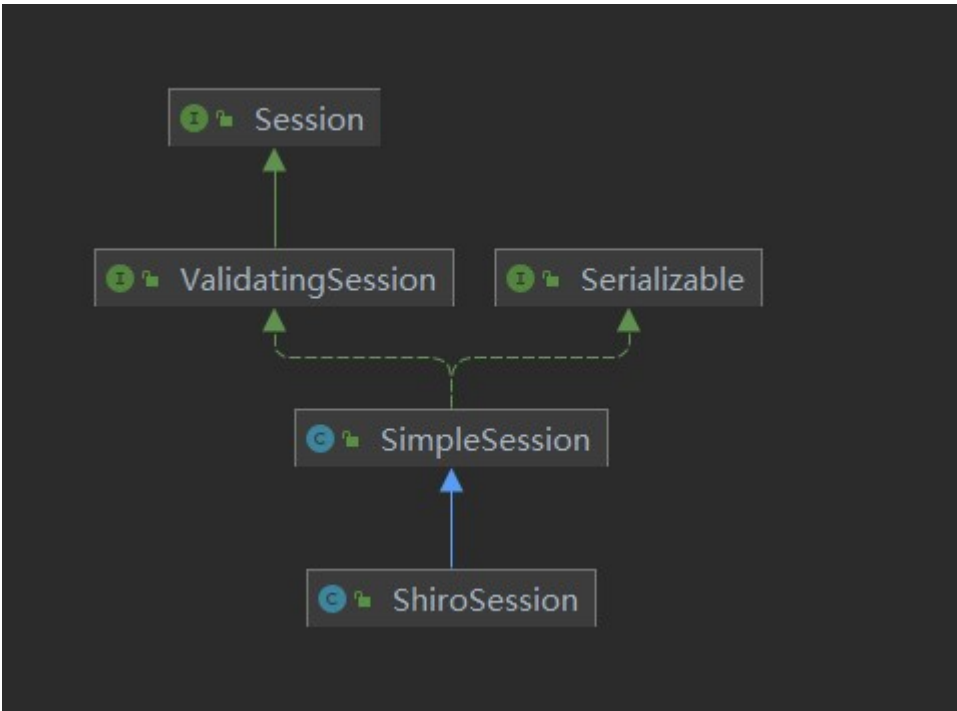


**Redis作用： **存放用户的权限数据；存放Session数据，用于分布式布局时不同服务器之间Session共享。

第一步：将Session存入Redis中，实现Session共享

此处需要创建以下四个类，其中ShiroSession是我们自定义的Session类，实现用户操作后更新redis中session功能；RedisSessionDao、ShiroSessionListener和ShiroSessionManager都是自定义的类，其中RedisSessionDao、ShiroSessionListener是ShiroSessionManager的两个组件。

- 1. 创建自定义的ShiroSession，继承SimpleSession(org.apache.shiro.session.mgt.SimpleSession)，相较于父类添加一个field：




```
public class ShiroSession extends SimpleSession {
    // 除lastAccessTime以外其他字段发生改变时为true
    private boolean isChanged;

    public ShiroSession() {
        super();
        this.setChanged(true);
    }

    public ShiroSession(String host) {
        super(host);
        this.setChanged(true);
    }

    @Override
    public void setId(Serializable id) {
        super.setId(id);
        this.setChanged(true);
    }

    @Override
    public void setStopTimestamp(Date stopTimestamp) {
        super.setStopTimestamp(stopTimestamp);
        this.setChanged(true);
    }

    @Override
    public void setExpired(boolean expired) {
        super.setExpired(expired);
        this.setChanged(true);
    }

    @Override
    public void setTimeout(long timeout) {
        super.setTimeout(timeout);
        this.setChanged(true);
    }

    @Override
    public void setHost(String host) {
        super.setHost(host);
        this.setChanged(true);
    }

    @Override
    public void setAttributes(Map<Object, Object> attributes) {
        super.setAttributes(attributes);
        this.setChanged(true);
    }

    @Override
    public void setAttribute(Object key, Object value) {
```

```
        super.setAttribute(key, value);
        this.setChanged(true);
    }

    @Override
    public Object removeAttribute(Object key) {
        this.setChanged(true);
        return super.removeAttribute(key);
    }

    /**
     * 停止
     */
    @Override
    public void stop() {
        super.stop();
        this.setChanged(true);
    }

    /**
     * 设置过期
     */
    @Override
    protected void expire() {
        this.stop();
        this.setExpired(true);
    }

    public boolean isChanged() {
        return isChanged;
    }

    public void setChanged(boolean isChanged) {
        this.isChanged = isChanged;
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }

    @Override
    protected boolean onEquals(SimpleSession ss) {
        return super.onEquals(ss);
    }

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```

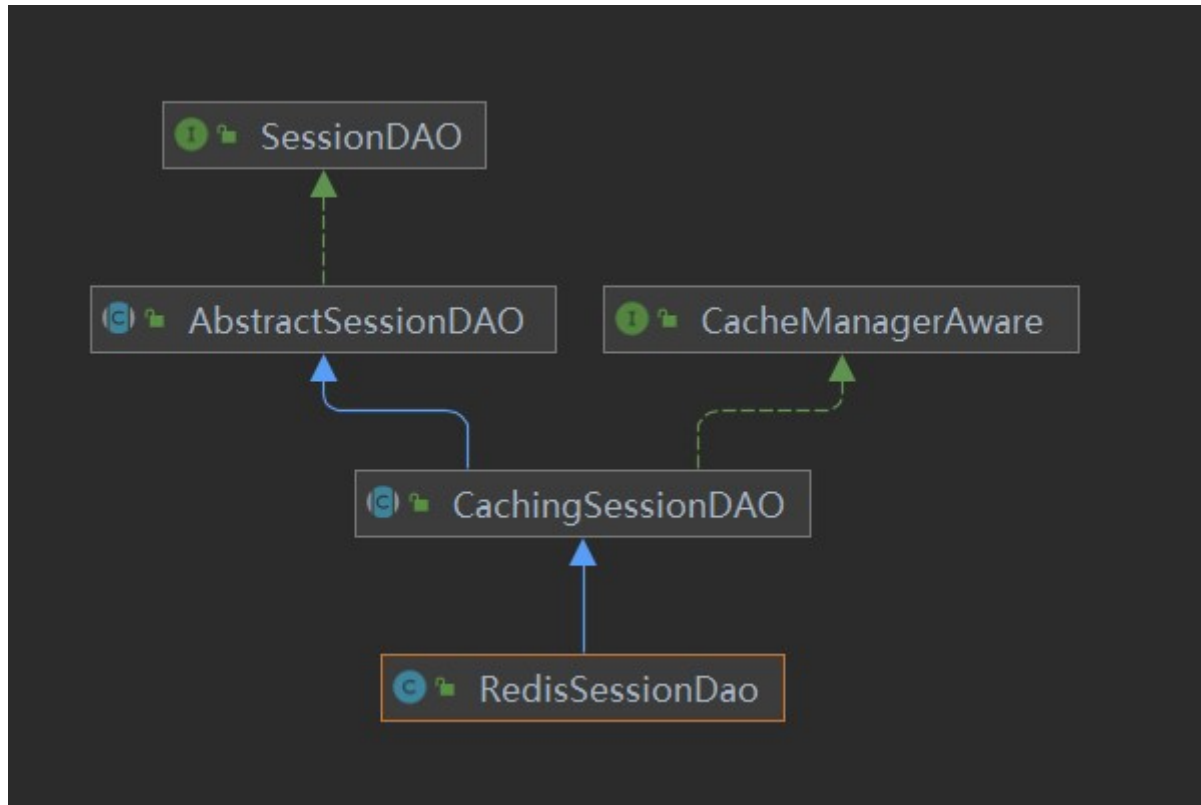
```

    }
}

```

2. 创建一个RedisSessionDao, 继承

CachingSessionDao(org.apache.shiro.session.mgt.eis.CachingSessionDAO)抽象类, 重写创建(doCreate)、读取(readSession和doReadSession)、更新(doUpdate)、删除(doDelete)、获取活跃会话(getActiveSessions)方法, 注入Redis工具的RedisTemplate对象, 实现增删改查。



```

public class RedisSessionDao extends CachingSessionDAO {
    private static final Logger LOGGER =
LoggerFactory.getLogger(RedisSessionDao.class);
    // 设置key的前缀
    private String prefix = "sessionId:";
    // 设置会话的过期时间
    private int seconds = 100;
    @Autowired
    private StringRedisTemplate redisTemplate;

    @Override
    protected Serializable doCreate(Session session) {
        // 创建一个Id并设置给Session
        Serializable sessionId = this.generateSessionId(session);
        assignSessionId(session, sessionId);
        try {
            session.setTimeout(seconds);
            redisTemplate.opsForValue().set(prefix + sessionId,
SerializeUtils.serializeToString((ShiroSession)session),seconds,
TimeUnit.MINUTES);
            LOGGER.info("sessionId {} name {} 被创建", sessionId,
session.getClass().getName());

```

```
        } catch (Exception e) {
            LOGGER.warn("创建Session失败", e);
        }
        return sessionId;
    }

    @Override
    public Session readSession(Serializable sessionId) throws
        UnknownSessionException {
        Session session = getCacheSession(sessionId);
        if (session == null
            ||
            session.getAttribute(DefaultSubjectContext.PRINCIPALS_SESSION_KEY) == null) {
            session = this.doReadSession(sessionId);
            if (session == null) {
                return null;
                //throw new UnknownSessionException("There is no session with id
[" + sessionId + "]");
            } else {
                // 缓存
                cache(session, session.getId());
            }
        }
        return session;
    }

    @Override
    protected Session doReadSession(Serializable sessionId) {
        Session session = null;
        try {
            String key = prefix + sessionId;
            String value = redisTemplate.opsForValue().get(key);
            if (value != null) {
                redisTemplate.opsForValue().set(key, value, seconds,
                    TimeUnit.MINUTES);
                session = SerializeUtils.deserializeFromString(value);
                LOGGER.info("sessionId {} name {} 被读取", sessionId,
                    session.getClass().getName());
            }
        } catch (Exception e) {
            LOGGER.warn("读取Session失败", e);
        }
        return session;
    }

    public Session doReadSessionWithoutExpire(Serializable sessionId) {
        Session session = null;
        try {
            String key = prefix + sessionId;
            String value = redisTemplate.opsForValue().get(key);
            if (StringUtils.isNotBlank(value)) {
                session = SerializeUtils.deserializeFromString(value);
            }
        } catch (Exception e) {
```

```
        LOGGER.warn("读取Session失败", e);
    }
    return session;
}

@Override
protected void doUpdate(Session session) {
    //如果会话过期/停止 没必要再更新了
    try {
        if (session instanceof ValidatingSession && !((ValidatingSession)
session).isValid()) {
            return;
        }
    } catch (Exception e) {
        LOGGER.error("ValidatingSession error");
    }

    try {
        if (session instanceof ShiroSession) {
            // 如果没有主要字段(除lastAccessTime以外其他字段)发生改变
            ShiroSession ss = (ShiroSession) session;
            if (!ss.isChanged()) {
                return;
            }
            try {
                ss.setChanged(false);
                redisTemplate.opsForValue().set(prefix + session.getId(),
SerializeUtils.serializeToString(ss), seconds, TimeUnit.MINUTES);
                // redisTemplate.opsForValue().set(prefix + session.getId(),
JsonCovertUtils.objToJson(ss), seconds, TimeUnit.MINUTES);
                LOGGER.info("sessionId {} name {} 被更新", session.getId(),
session.getClass().getName());
                // 执行事务
            } catch (Exception e) {
                throw e;
            }
        } else if (session instanceof Serializable) {
            redisTemplate.opsForValue().set(prefix + session.getId(),
JsonCovertUtils.objToJson((Serializable) session), seconds, TimeUnit.MINUTES);
            LOGGER.info("sessionId {} name {} 作为非ShiroSession对象被更新, ",
session.getId(), session.getClass().getName());
        } else {
            LOGGER.warn("sessionId {} name {} 不能被序列化 更新失败",
session.getId(), session.getClass().getName());
        }
    } catch (Exception e) {
        LOGGER.warn("更新Session失败", e);
    }
}

@Override
protected void doDelete(Session session) {
```

```

        try {
            redisTemplate.delete(prefix + session.getId());
            LOGGER.debug("Session {} 被删除", session.getId());
        } catch (Exception e) {
            LOGGER.warn("修改Session失败", e);
        }
    }

    /**
     * 删除cache中缓存的Session
     */
    public void uncache(Serializable sessionId) {
        Session session = this.readSession(sessionId);
        super.uncache(session);
        LOGGER.info("取消session {} 的缓存", sessionId);
    }

    /**
     * 获取当前所有活跃用户，如果用户量多此方法影响性能
     */
    @Override
    public Collection<Session> getActiveSessions() {
        try {
            Set<String> keys = redisTemplate.keys(prefix + "*");
            if (CollectionUtils.isEmpty(keys)) {
                return null;
            }
            List<String> valueList = redisTemplate.opsForValue().multiGet(keys);
            // return SerializeUtils.deserializeFromStringController(valueList);
            // return JsonCovertUtils.jsonToObj(valueList, ArrayList.class);
        } catch (Exception e) {
            LOGGER.warn("统计Session信息失败", e);
        }
        return null;
    }

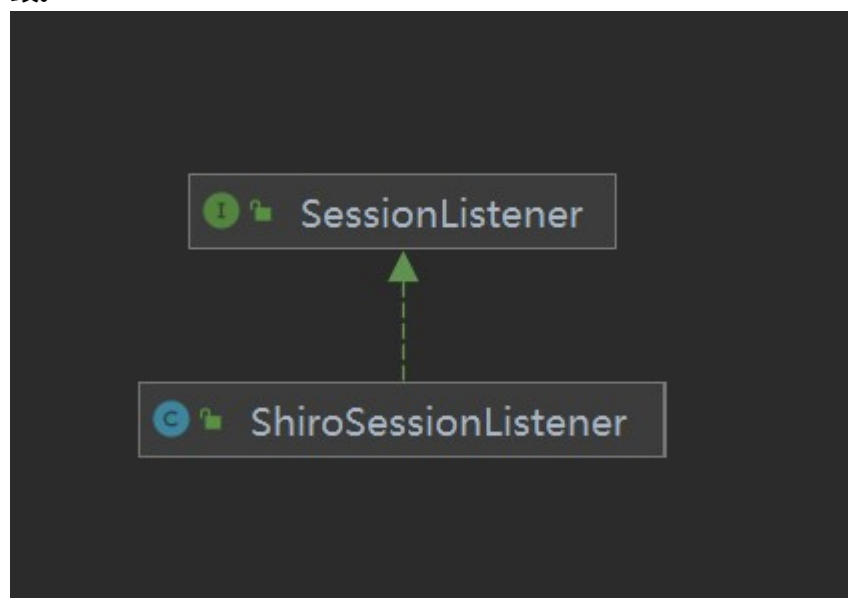
    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSeconds(int seconds) {
        this.seconds = seconds;
    }
}

```

3. 创建ShiroSessionListener，实现SessionListener(org.apache.shiro.session.SessionListener)接口，实现会话创建(onStart)、停止(onStop)、过期(onExpiration)方法：注入自定义SessionDao进行缓存创建、销

毁。



```
public class ShiroSessionListener implements SessionListener {
    private static final Logger logger =
        LoggerFactory.getLogger(ShiroSessionListener.class);

    @Autowired
    private RedisSessionDao sessionDao;

    @Override
    public void onStart(Session session) {
        // 会话创建时触发
        logger.info("ShiroSessionListener session {} 被创建", session.getId());
    }

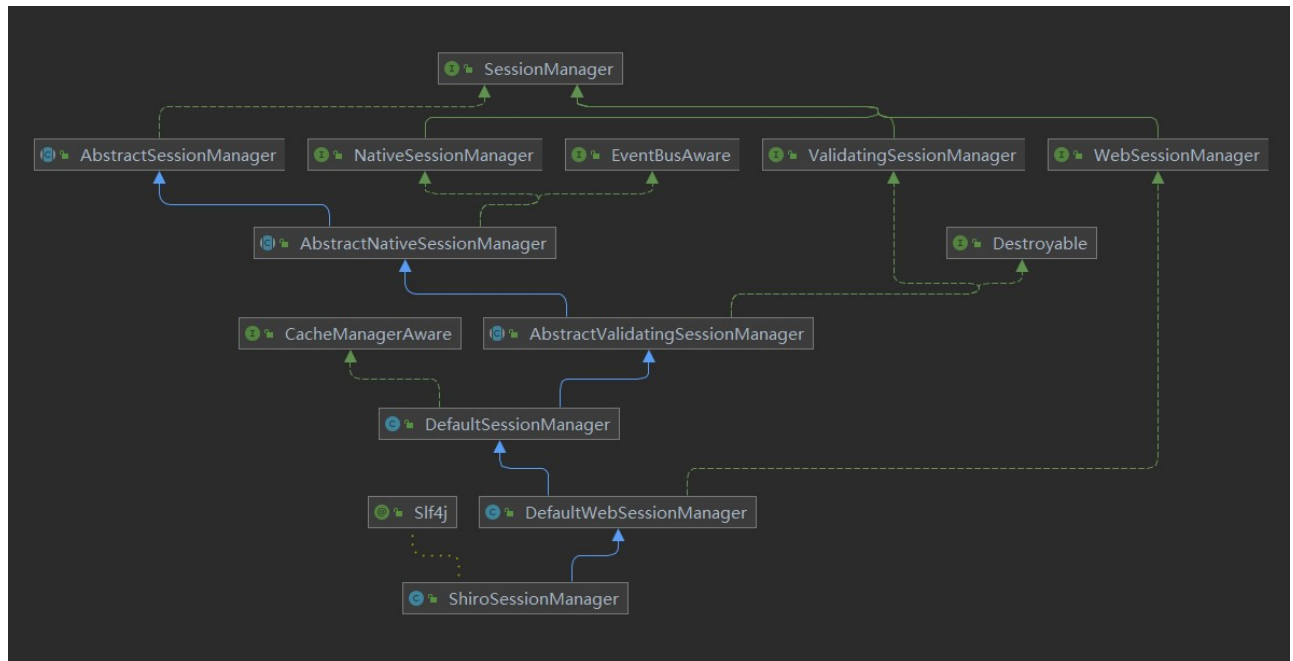
    @Override
    public void onStop(Session session) {
        sessionDao.delete(session);
        // 会话被停止时触发
        logger.info("ShiroSessionListener session {} 被销毁", session.getId());
    }

    @Override
    public void onExpiration(Session session) {
        sessionDao.delete(session);
        // 会话过期时触发
        logger.info("ShiroSessionListener session {} 过期", session.getId());
    }
}
```

4. 创建ShiroSessionManager，继承

DefaultWebSessionManager(org.apache.shiro.web.session.mgt.DefaultWebSessionManager)，重写

retrieveSession方法从缓存中获取session并解决多次访问redis问题。



```

@Slf4j
public class ShiroSessionManager extends DefaultWebSessionManager {
    /**
     * 获取session
     * 优化单次请求需要多次访问redis的问题
     * @param sessionKey
     * @return
     * @throws UnknownSessionException
     */
    @Override
    protected Session retrieveSession(SessionKey sessionKey) throws
    UnknownSessionException {
        Serializable sessionId = getSessionId(sessionKey);

        ServletRequest request = null;
        if (sessionKey instanceof WebSessionKey) {
            request = ((WebSessionKey) sessionKey).getServletRequest();
        }

        if (request != null && null != sessionId) {
            Object sessionObj = request.getAttribute(sessionId.toString());
            if (sessionObj != null) {
                return (Session) sessionObj;
            }
        }

        //下面这个父类的方法我们之间提出来单独来实现
        Session session = super.retrieveSession(sessionKey);

        if (sessionId == null) {
            log.debug("Unable to resolve session ID from SessionKey [{}].
            Returning null to indicate a " +

```

```

        "session could not be found.", sessionKey);
        return null;
    }

    //        //去缓存里面搜索session
    //        Session session = retrieveSessionFromDataSource(sessionId);
    //        if (session == null) {
    //            //session ID was provided, meaning one is expected to be found, but
    //            we couldn't find one:
    //            //因为种种原因session找不到了: 比如手动删除了或者过期了
    //            String msg = "Could not find session with ID [" + sessionId + "]";
    //            log.warn(msg);
    //            Response302(msg, ((WebSessionKey) sessionKey).getServletResponse());
    //            throw new UnknownSessionException(msg);
    //        }

    //=====单独实现结束=====
    if (request != null) {
        request.setAttribute(sessionId.toString(), session);
    }
    return session;
}

private void Response302(String msg, ServletResponse response){
    HttpServletResponse httpResponse = (HttpServletResponse) response;
    httpResponse.setStatus(302);
    httpResponse.setCharacterEncoding("utf-8");
    httpResponse.setContentType("text/html");
    try {
        PrintWriter out = httpResponse.getWriter();
        out.append(msg);
    } catch (Exception e){
        log.error("直接返回Response信息异常"+ e.getMessage());
        e.printStackTrace();
    }
}
}

```

5. 同时需要增加一个序列化工具类SerializeUtils, 实现序列化和反序列化, 具体实现看代码:

```

public class SerializeUtils extends SerializationUtils{

    public static String serializeToString(Serializable obj) {
        try {
            byte[] value = serialize(obj);
            return Base64.encodeToString(value);
        } catch (Exception e) {
            throw new RuntimeException("serialize session error", e);
        }
    }

    public static Session deserializeFromString(String base64) {
        try {

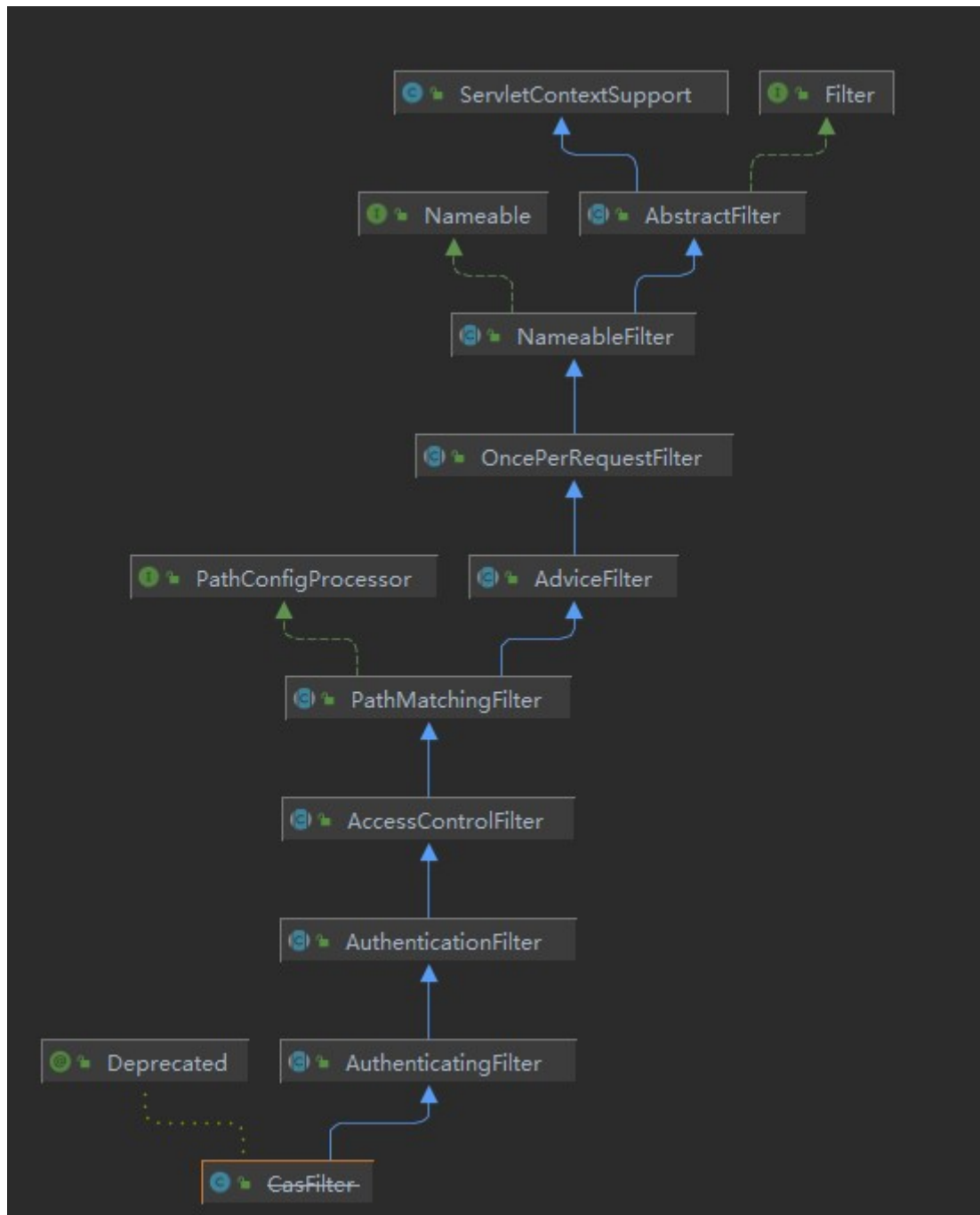
```

```
        byte[] objectData = Base64.decode(base64);
        return deserialize(objectData);
    } catch (Exception e) {
        throw new RuntimeException("deserialize session error", e);
    }
}

public static <T> Collection<T>
deserializeFromStringController(Collection<String> base64s) {
    try {
        List<T> list = new LinkedList<>();
        for (String base64 : base64s) {
            byte[] objectData = Base64.decode(base64);
            T t = deserialize(objectData);
            list.add(t);
        }
        return list;
    } catch (Exception e) {
        throw new RuntimeException("deserialize session error", e);
    }
}
```

第二步：利用shiro-cas依赖实现统一身份认证

1. 至此，我们已经实现了将session存入redis的功能。接下来需要介绍接入CAS统一身份认证，这里我们采用shiro内集成的cas功能来实现，所以这里要导入前面shiro-cas的依赖。首先我们看一下自带的CasFilter(org.apache.shiro.cas.CasFilter)，这个类已经被弃用了，但是仍然可以使用。这里并没有添加任何代码，仅描述一下工作原理和一些事项。



从图中可以看出，CasFilter继承自AuthenticatingFilter，具体代码看如下源代码：

```

/**
 * This filter validates the CAS service ticket to authenticate the user. It must
 * be configured on the URL recognized
 * by the CAS server. For example, in {@code shiro.ini}:
 * <pre>
 * [main]
 * casFilter = org.apache.shiro.cas.CasFilter
 * ...
 *
 * [urls]
 * /shiro-cas = casFilter
 * ...
 * </pre>
 * (example : http://host:port/mycontextpath/shiro-cas)

```

```
*
* @since 1.2
* @see <a href="https://github.com/bujiio/buji-pac4j">buji-pac4j</a>
* @deprecated replaced with Shiro integration in <a
href="https://github.com/bujiio/buji-pac4j">buji-pac4j</a>.
*/
@Deprecated
public class CasFilter extends AuthenticatingFilter {

    private static Logger logger = LoggerFactory.getLogger(CasFilter.class);

    // the name of the parameter service ticket in url
    private static final String TICKET_PARAMETER = "ticket";

    // the url where the application is redirected if the CAS service ticket
    validation failed (example : /mycontextpatch/cas_error.jsp)
    private String failureUrl;

    @Override
    protected AuthenticationToken createToken(ServletRequest request,
    ServletResponse response) throws Exception {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String ticket = httpRequest.getParameter(TICKET_PARAMETER);
        return new CasToken(ticket);
    }

    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse
    response) throws Exception {
        return executeLogin(request, response);
    }

    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse
    response, Object mappedValue) {
        return false;
    }

    @Override
    protected boolean onLoginSuccess(AuthenticationToken token, Subject subject,
    ServletRequest request,
                                ServletResponse response) throws Exception {
        issueSuccessRedirect(request, response);
        return false;
    }

    @Override
    protected boolean onLoginFailure(AuthenticationToken token,
    AuthenticationException ae, ServletRequest request,
                                ServletResponse response) {
        if (logger.isDebugEnabled()) {
            logger.debug( "Authentication exception", ae );
        }
        // is user authenticated or in remember me mode ?
    }
}
```



```

        Subject subject = getSubject(request, response);
        if (subject.isAuthenticated() || subject.isRemembered()) {
            try {
                issueSuccessRedirect(request, response);
            } catch (Exception e) {
                logger.error("Cannot redirect to the default success url", e);
            }
        } else {
            try {
                WebUtils.issueRedirect(request, response, failureUrl);
            } catch (IOException e) {
                logger.error("Cannot redirect to failure url : {}", failureUrl,
e);
            }
        }
        return false;
    }

    public void setFailureUrl(String failureUrl) {
        this.failureUrl = failureUrl;
    }
}

```

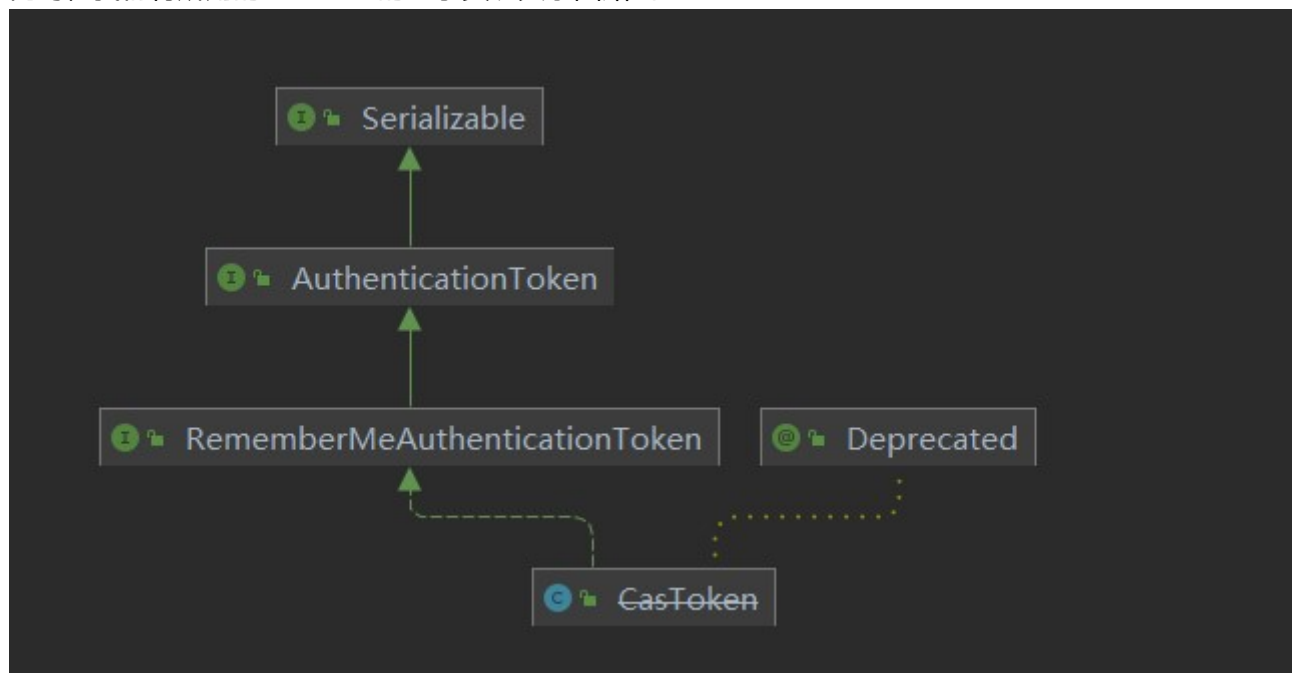
为了描述这个过程，我们把父类AuthenticatingFilter的executeLogin()方法的代码也贴出：

```

    protected boolean executeLogin(ServletRequest request, ServletResponse
response) throws Exception {
        AuthenticationToken token = this.createToken(request, response);
        if (token == null) {
            String msg = "createToken method implementation returned null. A valid
non-null AuthenticationToken must be created in order to execute a login
attempt.";
            throw new IllegalStateException(msg);
        } else {
            try {
                Subject subject = this.getSubject(request, response);
                subject.login(token);
                return this.onLoginSuccess(token, subject, request, response);
            } catch (AuthenticationException var5) {
                return this.onLoginFailure(token, var5, request, response);
            }
        }
    }
}

```

同时，我们将所用的CasToken的继承实现关系图贴出：



这里简要描述以下它的执行流程：

- 符合该过滤器处理的请求进来后首先进入isAccessAllowed()方法，判断能否允许进入，这里return false; 代表一律拦截。
- 然后进入访问拒绝的onAccessDenied()方法，然后执行父类AuthenticatingFilter的执行登录executeLogin()的方法。
- 在第一行执行子类CasFilter的createToken()方法返回AuthenticationToken的对象(注意：这里所说的token与我们前面提到的JWT token不要混淆，这里指的是Shiro内部里面的身份认证凭证，两者不是一种东西。)，该方法从请求参数中获取票据，然后封装成CasToken对象返回。
- 回到AuthenticatingFilter中executeLogin方法。接下来执行Subject.login()方法并把刚刚拿到的CasToken作为参数传入。
- 此后还有一系列操作，包括选择进入合适的realm等流程，这里你不用去自行了解这个流程，你只需要知道Subject.login()方法最终会到我们指定的realm里面去执行认证方法doGetAuthenticationInfo()(后面会提到)。
- 上一步成功后，进入CasFilter的onLoginSuccess()方法(Why? 明明是父类AuthenticatingFilter调用的this.onLoginSuccess()方法为什么是子类运行，因为子类重写了父类的方法，且实例化了子类)，调用issueSuccessRedirect()方法，实际上是父类的父类AuthenticationFilter的issueSuccessRedirect()方法内WebUtils.redirectToSavedRequest()方法，即登录成功后重定向到原本请求的地址，流程结束。如果上一步失败，会进入CasFilter的onLoginFailure()方法，判断是不是已经认证过或者设置了“记住我”功能，如果是重定向访问，否则重定向到failureUrl，流程结束。

所以结合前面统一身份认证的流程，这个CasFilter应该是用于Cas重定向我们的服务时候回调拦截的，因为回调的时候会在请求参数里面带上票据ticket。并且这个类里面只有一个field：

```
private String failureUrl;
```

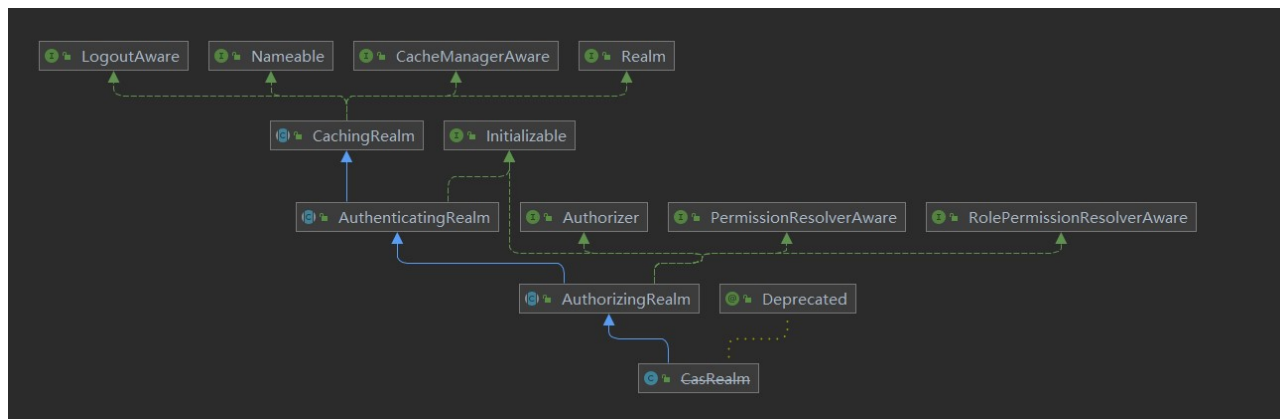
并没有关于CAS服务参数的相关信息field，所以更加印证它的作用仅用于回调拦截并把ticket封装成CasToken交付给realm的认证方法。根据统一身份认证流程，我们该如何验证这个ticket并得到用户信息呢？我们可以从前面得出结论：**我们需要在realm的doGetAuthenticationInfo()中去实现验证ticket并得到用户信息！**

还有一个问题：请问这个拦截器只作用于回调的地址还是所有都要用此拦截？

答：只作用于回调地址，因为该拦截器并没有实现跳转到登录界面的代码，那么用户没登陆访问资源地址怎么拦截并跳转到统一身份认证登录界面呢？三个方案：

- 自己新增一个拦截器，判断subject有无认证，没有就重定向。
- 上面方案其实可以通过设置ShiroFilterFactoryBean的LoginUrl值为跳转地址，但是需要解决跨域问题。
- 设置ShiroFilterFactoryBean的LoginUrl值为本地接口假如为/needLogin，然后在对应Controller里面返回前端一个状态码和信息，前端收到状态码后执行跳转任务。(推荐)

2. 创建统一身份认证的MyShiroCasRealm，继承CasRealm(org.apache.shiro.cas.CasRealm)，并在里面实现验证票据操作。



```

public class MyShiroCasRealm extends CasRealm {
    private static final Logger logger =
    LoggerFactory.getLogger(MyShiroCasRealm.class);
    {
        //设置realm的名字，用于多realm认证区分
        super.setName("user");
    }

    @Autowired
    private UserService userService;

    /**
     * 此处就是前文提到的认证方法(关键!)
     */
    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
        // 从Subject.login(token)这个地方拿到AuthenticationToken强制转化为CasToken
        CasToken casToken = (CasToken)token;
        if (token == null) {
            return null;
        } else {
            // 在CasToken中Credentials就是票据ticket
            String ticket = (String)casToken.getCredentials();
            if (!StringUtils.hasText(ticket)) {
                return null;
            } else {
                // 创建TicketValidator对象
            }
        }
    }
}
  
```

```

        TicketValidator ticketValidator = this.ensureTicketValidator();
        try {
            // 校验票据, 参数有票据本身, 还有父类field: casService, 代表cas服
            // 务地址, 有这个参数TicketValidator才知道去哪验票
            Assertion casAssertion = ticketValidator.validate(ticket,
this.getCasService());
            // 验票成功, 获取principal身份信息
            AttributePrincipal casPrincipal = casAssertion.getPrincipal();
            // 获取用户名
            String userId = casPrincipal.getName();
            logger.debug("Validate ticket : {} in CAS server : {} to
retrieve user : {}", new Object[]{ticket, this.getCasServerUrlPrefix(), userId});
            // 获取更多用户信息, 具体有什么信息需要看CAS服务的相关信息
            Map<String, Object> attributes = casPrincipal.getAttributes();
            casToken.setUserId(userId);

            //查询数据库
            User user = userService.getUserByUserName(userId);

            // 判断数据库有无该用户
            if(ObjectUtils.isEmpty(user)){
                // 如果没有, 进行操作
                // TODO
            }
            // 进行后续操作, 例如将用户信息放入session, 总之都要将用户相关信息
            // 封装成SimpleAuthenticationInfo信息
            // TODO

            SecurityUtils.getSubject().getSession().setAttribute(Constant.SESSION_USER_INFO,
JsonCovertUtils.objToJson(user));
            List<Object> principals = CollectionUtils.asList(new Object[]
{userId, attributes});
            PrincipalCollection principalCollection = new
SimplePrincipalCollection(principals, this.getName());
            return new SimpleAuthenticationInfo(principalCollection,
ticket);
        } catch (TicketValidationException | JsonProcessingException
var14) {
            throw new CasAuthenticationException("Unable to validate
ticket [" + ticket + "]", var14);
        }
    }
}

/**
 * 授权(非关键)
 * 注意: 连续请求同一个url, 该方法不会执行多次, 因为有一个时间间隔, 这个参数可以在
shiro 配置内设置
 */
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
    logger.info("=====进入授权领域, Realm:" + this.getName() +

```

```
"=====");
    String loginName =
(String)super.getAvailablePrincipal(principalCollection);
    // 到数据库查查询用户身份或权限，实际项目中可以将授权信息保存在Redis缓存内提高性能，避免重复操作数据库
    // TODO

    //如果返回null，则会重定向到unAuthorization页面
    return simpleAuthorizationInfo;
}
}
```

3. 至此我们已经拥有了基本组件，现在只需要将这些组件全部配置到ShiroConfig

```
@Configuration
public class ShiroConfiguration {
    /** Cas服务前缀*/
    @Value("${cas.casServer.urlPrefix}")
    private String casServerUrlPrefix;

    /** Cas登录页面地址 */
    @Value("${cas.casServer.loginUrl}")
    private String casLoginUrl;

    /** Cas登出页面地址 */
    @Value("${cas.casServer.logoutUrl}")
    private String casLogoutUrl;

    /** 当前工程对外提供的服务地址前缀 */
    @Value("${cas.client.urlPrefix}")
    private String myServerUrlPrefix;

    /** 当前项目的回调接口 */
    @Value("${cas.client.casCallback}")
    private String casFilterUrlPattern;

    /** 登录的具体地址 */
    @Value("${cas.casServer.loginRequestUrl}")
    private String loginUrl;

    /** 成功地址 */
    @Value("${cas.client.successUrl}")
    private String successUrl;

    /** 未授权地址 */
    @Value("${cas.client.unauthorizedUrl}")
    private String unauthorizedUrl;

    /** 本地通知需要登录接口*/
    @Value("${cas.client.needLogin}")
    private String needLogin;
```

```
/** 注入Redis缓存管理器对象：后续实现多Realm时讲解该组件使用*/
@Bean("redisCacheManager")
public RedisCacheManager redisCacheManager(){
    return new RedisCacheManager();
}

/** 注入MyShiroCasRealm对象*/
@Bean(name = "myShiroCasRealm")
public MyShiroCasRealm myShiroCasRealm(RedisCacheManager redisCacheManager) {
    MyShiroCasRealm realm = new MyShiroCasRealm();
    // 设置Redis缓存
    realm.setCacheManager(redisCacheManager);
    // 开启缓存
    realm.setCachingEnabled(true);
    // 不打开认证缓存，否则出错
    realm.setAuthenticationCachingEnabled(false);
    // 开启授权缓存，提高性能
    realm.setAuthorizationCachingEnabled(true);
    // 设置授权缓存名称
    realm.setAuthorizationCacheName("CasAuthorizationCache");
    // 设置CAS服务前缀
    realm.setCasServerUrlPrefix(casServerUrlPrefix);
    // 设置CAS服务回调地址
    realm.setCasService(myServerUrlPrefix + casFilterUrlPattern);
    return realm;
}

/** 注入账号密码登录的realm对象，构成多realm，不需要的可以去掉*/
@Bean(name = "userRealm")
public UserRealm userRealm(RedisCacheManager redisCacheManager){
    UserRealm userRealm = new UserRealm();
    //创建hash校验匹配器
    HashedCredentialsMatcher credentialsMatcher = new
HashedCredentialsMatcher();
    //设置散列算法
    credentialsMatcher.setHashAlgorithmName("MD5");
    //设置散列次数
    credentialsMatcher.setHashIterations(64);
    //设置凭证匹配器
    userRealm.setCredentialsMatcher(credentialsMatcher);

    // 设置Redis缓存管理器
    userRealm.setCacheManager(redisCacheManager);
    // 开启缓存
    userRealm.setCachingEnabled(true);
    // 开启认证缓存
    userRealm.setAuthenticationCachingEnabled(true);
    // 开启授权缓存
    userRealm.setAuthorizationCachingEnabled(true);
    // 设置认证缓存名称
    userRealm.setAuthenticationCacheName("authenticationCache");
    // 设置授权缓存名称
    userRealm.setAuthorizationCacheName("authorizationCache");
    return userRealm;
}
```



```
}

/**
 * CAS过滤器
 */
@Bean(name = "casFilter")
public CasFilter getCasFilter() {
    CasFilter casFilter = new CasFilter();
    casFilter.setName("casFilter");
    casFilter.setEnabled(true);
    //casFilter.setSuccessUrl(successUrl);
    // 登录失败后跳转的URL, 也就是 Shiro 执行 CasRealm 的
doGetAuthenticationInfo 方法向CasServer验证ticket
    casFilter.setFailureUrl(loginUrl);// 我们选择认证失败后再打开登录页面
    return casFilter;
}

/**
 * 通过FilterRegistrationBean实例注册
 */
@Bean
public FilterRegistrationBean filterRegistrationBean() {
    FilterRegistrationBean filterRegistration = new FilterRegistrationBean();
    filterRegistration.setFilter(new
DelegatingFilterProxy("shiroFilterFactoryBean"));
    // 该值缺省为false,表示生命周期由SpringApplicationContext管理,设置为true则表示由ServletContainer管理
    filterRegistration.addInitParameter("targetFilterLifecycle", "true");
    filterRegistration.setEnabled(true);
    filterRegistration.addUrlPatterns("/");
    return filterRegistration;
}

/**
 * 安全管理器, Shiro的核心组件, 如果不用设置多realm, 参数少一个realm, 然后设置一个
realm就行, 多realm的代码可以去掉
 */
@Bean(name = "securityManager")
public DefaultWebSecurityManager getDefaultWebSecurityManager(MyShiroCasRealm
myShiroCasRealm, UserRealm userRealm, RedisSessionDao redisSessionDao) {
    DefaultWebSecurityManager dwsm = new DefaultWebSecurityManager();
    // 设置 SubjectFactory 指定为CasSubjectFactory, 不需要额外进行设置
    dwsm.setSubjectFactory(new CasSubjectFactory());
    //注入记住我管理器, 该管理器是给userRealm使用的
    dwsm.setRememberMeManager(rememberMeManager());
    // 设置session管理器, 将我们的组件导入
    dwsm.setSessionManager(sessionManager(sessionListener(), redisSessionDao));

    // 设置一个realm
    // dwsm.setRealm(myShiroCasRealm)

    // 设置多realm
    List<Realm> realms = new ArrayList<>();
```

```

        realms.add(myShiroCasRealm);
        realms.add(userRealm);

        // 设置多realm相关：关于MyModularRealmAuthorizer与
        MyModularRealmAuthenticator后面会给出实现
        MyModularRealmAuthorizer authorizer = new MyModularRealmAuthorizer();
        authorizer.setRealms(realms);
        dwsm.setAuthorizer(authorizer);
        MyModularRealmAuthenticator authenticator = new
        MyModularRealmAuthenticator();
        authenticator.setRealms(realms);
        dwsm.setAuthenticator(authenticator);

        return dwsm;
    }

    /**
     * 加载shiroFilter权限控制规则（从数据库读取然后配置）
     */
    private void loadShiroFilterChain(ShiroFilterFactoryBean
    shiroFilterFactoryBean){
        Map<String, String> filterChainDefinitionMap = new LinkedHashMap<String,
        String>();
        // shiro集成cas后，首先添加该规则，将/casCallback接口与注入的casFilter进行匹配
        filterChainDefinitionMap.put(casFilterUrlPattern, "casFilter");

        // authc：该过滤器下的页面必须验证后才能访问，它是Shiro内置的一个拦截器
        org.apache.shiro.web.filter.authc.FormAuthenticationFilter
        // anon：它对应的过滤器里面是空的，什么都没做

        log.info("#####从数据库读取权限规则，加载到shiroFilter中
        #####");
        //TODO 把角色权限规则从数据库读取

        filterChainDefinitionMap.put("/public/**", "anon");
        filterChainDefinitionMap.put("/**", "anon");

        shiroFilterFactoryBean.setFilterChainDefinitionMap(filterChainDefinitionMap);
    }

    /**
     * 注入，shiroFilterFactoryBean，主要配置登入登出以及需要登录接口，未授权地址等信息
     以及加载权限，也就是loadShiroFilterChain函数
     */
    @Bean(name = "shiroFilterFactoryBean")
    public ShiroFilterFactoryBean
    getShiroFilterFactoryBean(DefaultWebSecurityManager securityManager, CasFilter
    casFilter) {
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
        ShiroFilterFactoryBean();

```

```
// 必须设置 SecurityManager
shiroFilterFactoryBean.setSecurityManager(securityManager);
// 如果不设置默认会自动寻找Web工程根目录下的"/login.jsp"页面
shiroFilterFactoryBean.setLoginUrl(loginUrl);
// 登录成功后要跳转的连接
shiroFilterFactoryBean.setSuccessUrl(successUrl);
shiroFilterFactoryBean.setUnauthorizedUrl(unauthorizedUrl);
//设置登出拦截器
LogoutFilter logout = new LogoutFilter();
// 添加casFilter到shiroFilter中
Map<String, Filter> filters = new HashMap<>();
filters.put("logout", logout);
shiroFilterFactoryBean.setFilters(filters);
loadShiroFilterChain(shiroFilterFactoryBean);
return shiroFilterFactoryBean;
}

/**
 * “记住我”功能的cookie对象;
 */
@Bean
public SimpleCookie rememberMeCookie(){
    //这个参数是cookie的名称，对应前端的checkbox的name = rememberMe
    SimpleCookie simpleCookie = new SimpleCookie("rememberMe");
    //记住我cookie生效时间30天 ,单位秒
    simpleCookie.setMaxAge(2592000);
    return simpleCookie;
}

/**
 * “记住我”功能cookie管理对象;
 */
@Bean
public CookieRememberMeManager rememberMeManager(){
    CookieRememberMeManager cookieRememberMeManager = new
CookieRememberMeManager();
    cookieRememberMeManager.setCookie(rememberMeCookie());
    return cookieRememberMeManager;
}

/**
 * session管理器，设置session有效期，鉴别周期等信息
 */
@Bean
public SessionManager sessionManager(SessionListener sessionListener,
RedisSessionDao redisSessionDao){
    ShiroSessionManager sessionManager = new ShiroSessionManager();
    sessionManager.setGlobalSessionTimeout(1800*1000);
    sessionManager.setDeleteInvalidSessions(false);
    sessionManager.setSessionValidationSchedulerEnabled(false);
    sessionManager.setSessionValidationInterval(5000);
    sessionManager.setSessionFactory(new ShiroSessionFactory());
    sessionManager.setSessionDAO(redisSessionDao);
}
```

```
        List<SessionListener> sessionListeners = new ArrayList<>();
        sessionListeners.add(sessionListener);
        sessionManager.setSessionListeners(sessionListeners);
        return sessionManager;
    }

    /**
     * 注入组件sessionLisener
     */
    @Bean("sessionLisener")
    public SessionListener sessionListener(){
        return new ShiroSessionListener();
    }

    /**
     * 注入组件redisSessionDao
     */
    @Bean("redisSessionDao")
    public RedisSessionDao redisSessionDao(){
        return new RedisSessionDao();
    }

    /**
     * 下面的代码是添加注解支持，必须添加，否则会引起@Bean注入为null
     */
    @Bean
    @DependsOn("lifecycleBeanPostProcessor")
    public DefaultAdvisorAutoProxyCreator defaultAdvisorAutoProxyCreator() {
        DefaultAdvisorAutoProxyCreator defaultAdvisorAutoProxyCreator = new
DefaultAdvisorAutoProxyCreator();
        // 强制使用cglib，防止重复代理和可能引起代理出错的问题
        // https://zhuanlan.zhihu.com/p/29161098
        defaultAdvisorAutoProxyCreator.setProxyTargetClass(true);
        return defaultAdvisorAutoProxyCreator;
    }

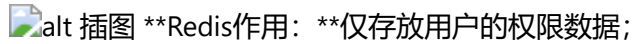
    @Bean
    public static LifecycleBeanPostProcessor lifecycleBeanPostProcessor() {
        return new LifecycleBeanPostProcessor();
    }

    @Bean
    public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(DefaultWebSecurityManager securityManager) {
        AuthorizationAttributeSourceAdvisor advisor = new
AuthorizationAttributeSourceAdvisor();
        advisor.setSecurityManager(securityManager);
        return advisor;
    }
}
```

4. 运行测试，关于RedisCacheManager、MyModularRealmAuthorizer与MyModularRealmAuthenticator、Constant的代码在介绍第二种解决方案时统一给出。

方案二——JWT Token认证实现

这种方案原理图如下：

alt 插图 **Redis作用： **仅存放用户的权限数据；

第一步：添加一个JWT工具类

这个工具类主要用于生成JWT，并且校验JWT提取JWT内的字段。具体代码：

```
@Component
public class JwtTokenUtils {

    // 设置加密用的secret
    @Value("${jwt.token.secret}")
    private String secret;

    // 设置有效期
    @Value("${jwt.token.expiration}")
    private Long expiration;

    // 设置发布者
    @Value("${jwt.token.issuer}")
    private String issuer;

    public String getSecret() {
        return secret;
    }

    public void setSecret(String secret) {
        this.secret = secret;
    }

    public Long getExpiration() {
        return expiration;
    }

    public void setExpiration(Long expiration) {
        this.expiration = expiration;
    }

    public String getIssuer() {
        return issuer;
    }

    public void setIssuer(String issuer) {
        this.issuer = issuer;
    }
}
```

```
/**
 * 生成token
 * @param username 用户名
 * @param
 * @return
 */
public String generateToken(String username, String IPAddress, String
department) {
    String token = null;
    try {
        Algorithm algorithm = Algorithm.HMAC256(secret);
        Date date = new Date();
        token = JWT.create()
            .withIssuer(issuer)
            .withIssuedAt(date)
            .withExpiresAt(DateUtils.addMilliseconds(date,
expiration.intValue()))
            .withClaim("username",username)
            .withClaim("IPAddress",IPAddress)
            .withClaim("department",department)
            .sign(algorithm);
    }catch (JWTCreationException e){
        e.printStackTrace();
    }
    return token;
}

/**
 * 根据token获取用户名
 * @param token
 * @return
 */
public String getUsernameFromToken(String token) {
    String username = null;
    DecodedJWT jwt = verifyToken(token);
    if(ObjectUtils.isEmpty(jwt)){
        return null;
    }
    username = jwt.getClaim("username").asString();
    return username;
}

/**
 * 根据token获取IP
 * @param token
 * @return
 */
public String getIPAdressToken(String token) {
    String IPAddress = null;
    DecodedJWT jwt = verifyToken(token);
    if(ObjectUtils.isEmpty(jwt)){
        return null;
    }
}
```



```
        IPAddress = jwt.getClaim("IPAddress").asString();
        return IPAddress;
    }

    /**
     * 根据token获取可选字段, 自己想设置的字段, 非必须, 这里是部门
     * @param token
     * @return
     */
    public String getDepartmentFromToken(String token) {
        String department = null;
        DecodedJWT jwt = verifyToken(token);
        if(ObjectUtils.isEmpty(jwt)){
            return null;
        }
        department = jwt.getClaim("department").asString();
        return department ;
    }

    /**
     * verify token返回一个verifier
     * @param token
     * @return
     */
    public DecodedJWT verifyToken(String token){
        DecodedJWT jwt = null;
        try {
            Algorithm algorithm = Algorithm.HMAC256(secret);
            JWTVerifier verifier = JWT.require(algorithm)
                .withIssuer(issuer)
                .build();
            jwt = verifier.verify(token);
        } catch (JWTDecodeException e) {
            e.printStackTrace();
        }
        return jwt;
    }

    /**
     * 判断token失效时间是否到了
     * @param token
     * @return
     */
    public Boolean isTokenExpired(String token) {
        try {
            DecodedJWT jwt = verifyToken(token);
        } catch (TokenExpiredException e) {
            return true;
        }
        return false;
    }

    /**
     * 获取设置的token失效时间
```

```
    * @param token
    * @return
    */
    public Date getExpirationDateFromToken(String token) {
        Date expiration;
        try {
            if(isTokenExpired(token)){
                return null;
            }
            DecodedJWT jwt = verifyToken(token);
            expiration = jwt.getExpiresAt();
        } catch (TokenExpiredException e) {
            return null;
        }
        return expiration;
    }

    public String refreshToken(String token) {
        DecodedJWT jwt = verifyToken(token);
        String username = jwt.getClaim("username").asString();
        String IPAddress = jwt.getClaim("IPAddress").asString();
        String department = jwt.getClaim("department").asString();
        return generateToken(username, IPAddress, department);
    }
}
```

这个工具类实现其实并不优雅，代码值得优化，getUsernameFromToken、getExpirationDateFromToken、getIPAdressToken、getDepartmentFromToken都可以统一成一个函数，它们都会去调用verifyToken检验token，并且需要统一一下异常抛出以便后续使用。

同时给出配置properties：

```
# 设置secret
jwt.token.secret=12345678909876543210102030405060708090
# 设置有效期7*24*60*60*1000
jwt.token.expiration=604800000
# 设置发布者
jwt.token.issuer=daku
```

第二步：自定义AuthenticationToken

我们假如我们的拦截器是即可以用于CAS回调拦截验证票据，还能拦截所有请求校验JWT的多功能拦截器，那在这两种情形下，我们要对不同的请求进行区分，如果请求参数带了ticket，那么我们就对应去验证票据，否则就要验证请求中可能携带的JWT。那为了在认证过程中存放两个数据，我们就要求AuthenticationToken能够存放两种不同的数据，并且还能标记类型。

1. 自定义一个枚举类，代表不同的两种请求类型：带有ticket回调需要验证票据的请求和可能带有JWT的普通用户访问请求(用户没登陆就没有JWT)。

```
// 该枚举类是区别CasJwtToken的不同应用场景
public enum CasJwtTokenMode {
    // CAS登录回调
    CasLogin,

    // 用户访问资源
    UserAccess
}
```

2. 有了这个枚举类，我们就可以自定义能同时处理两种情况的AuthenticationToken。创建一个CasJwtToken实现AuthenticationToken。设置username、ticket、token、IPAddress、mode字段，分别代表用户名(用户唯一标识)、票据、JWT、IP地址(用于校验发放JWT时IP地址与实际请求IP地址是否一致，可以在一定程度上避免攻击者盗用用户JWT实现了异地登录，如果有必要还可以加上机器码)、请求模式(也就是上面枚举类型)。

```
public class CasJwtToken implements AuthenticationToken {
    private String username;
    private String ticket;
    private String token;
    private String IPAddress;
    private CasJwtTokenMode mode;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getTicket() {
        return ticket;
    }

    public void setTicket(String ticket) {
        this.ticket = ticket;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }

    public String getIPAddress() {
        return IPAddress;
    }
}
```

```
public void setIPAddress(String IPAddress) {
    this.IPAddress = IPAddress;
}

public CasJwtToken() {
}

public CasJwtTokenMode getMode() {
    return mode;
}

public void setMode(CasJwtTokenMode mode) {
    this.mode = mode;
}

public CasJwtToken(String token) {
    this.token = token;
}

public CasJwtToken(String ticket, String token) {
    this.ticket = ticket;
    this.token = token;
}

public CasJwtToken(String ticket, String token, String IPAddress) {
    this.ticket = ticket;
    this.token = token;
    this.IPAddress = IPAddress;
}

public CasJwtToken(String username, String ticket, String token, String
IPAddress) {
    this.username = username;
    this.ticket = ticket;
    this.token = token;
    this.IPAddress = IPAddress;
}

@Override
public Object getPrincipal() {
    return this.username;
}

// 获取凭证时候通过mode判断决定返回ticket还是JWT
@Override
public Object getCredentials() {
    if(this.mode.equals(CasJwtTokenMode.CasLogin)){
        return this.ticket;
    }else{
        return this.token;
    }
}
}
```

第三步：自定义一个拦截器

我们可以参考上面CasFilter的思路，自己来定义一个拦截器。但是不同的是，CasFilter是继承了AuthenticationFilter，因为第一种方案是有状态的认证方案，所以继认证拦截器。

具体代码：

```
@Slf4j
public class CustomCallbackFilter extends BasicHttpAuthenticationFilter {

    // 一律拒绝访问
    @Override
    protected boolean isAccessAllowed(ServletRequest request, ServletResponse
response, Object mappedValue) {
        return false;
    }

    // 拒绝后执行登录
    @Override
    protected boolean onAccessDenied(ServletRequest request, ServletResponse
response) throws Exception{
        try {
            this.executeLogin(request, response);
            //((HttpServletResponse) response).setStatus(HttpStatus.OK.value());
            return true;
        } catch (Exception e) {
            String msg = e.getMessage();
            Throwable throwable = e.getCause();
            if(throwable instanceof JWTDecodeException){
                responseError(request, response, "JWT不合法或解码错误!", 403);
            }else if(throwable instanceof TokenExpiredException){
                responseError(request, response, "JWT过期失效!", 403);
            }else if(e instanceof LoginIPException) {
                responseError(request, response, "IP校验异常!", 403);
            }else if(e instanceof AuthenticationException){
                responseError(request, response, e.getMessage(), 403);
            }else {
                if(throwable != null){
                    responseError(request, response, e.getMessage(), 403);
                }
            }
            return false;
            //e.printStackTrace();
        }
    }

    // 创建自定义AuthenticationToken
    @Override
    protected AuthenticationToken createToken(ServletRequest request,
ServletResponse response) {
        // 转化为HttpServletRequest
```

```
HttpServletRequest httpRequest = (HttpServletRequest)request;
// 获取请求里面可能存在的JWT
String token = getRequestToken(httpRequest);
// 获取参数里面可能存在的ticket
String ticket = httpRequest.getParameter(Constant.TICKET_NAME);
//创建CasJwtToken对象
CasJwtToken casJwtToken = new CasJwtToken();
//设置JWT、ticket、以及请求IP地址
casJwtToken.setTicket(ticket);
casJwtToken.setToken(token);
casJwtToken.setIPAddress(getIPAddress(httpRequest));
//判断请求是不是带有ticket参数, 从而决定请求类型
if(isTicketRequest(httpRequest)){
    casJwtToken.setMode(CasJwtTokenMode.CasLogin);
}else{
    casJwtToken.setMode(CasJwtTokenMode.UserAccess);
}
return casJwtToken;
}

// 执行登录
@Override
protected boolean executeLogin(ServletRequest request, ServletResponse
response) throws Exception{
    CasJwtToken token = (CasJwtToken) this.createToken(request, response);
    Subject subject = this.getSubject(request, response);
    subject.login(token);
    return true;
}

private boolean isTicketRequest(HttpServletRequest httpRequest) {
    return
StringUtils.isEmpty(httpRequest.getParameter(Constant.TICKET_NAME));
}

// 该方法用于从header,params,cookie中搜索token
protected String getRequestToken(HttpServletRequest request){
    String token = request.getHeader(Constant.TOKEN);
    if (com.example.dakudemo.util.StringUtils.isBlank(token)){
        token = request.getHeader(Constant.AUTHORIZATION_HEADER);
    }
    if(com.example.dakudemo.util.StringUtils.isBlank(token)){
        token = request.getParameter(Constant.TOKEN);
    }
    if (com.example.dakudemo.util.StringUtils.isBlank(token)) {
        token = request.getParameter("sx_sso_sessionid");
    }
    if(com.example.dakudemo.util.StringUtils.isBlank(token)){
        Cookie[] cookies = request.getCookies();
        if(cookies != null && cookies.length > 0){
            for(Cookie cookie : cookies){
                if(Constant.TOKEN.equals(cookie.getName())){
```

```
        token = cookie.getValue();
        break;
    }
}
}
}
return token;
}

// 获取请求的ip地址
public static String getIPAddress(HttpServletRequest request) {
    String ipAddress = request.getHeader("x-forwarded-for");
    if (ipAddress == null || ipAddress.length() == 0 ||
"unknown".equalsIgnoreCase(ipAddress)) {
        ipAddress = request.getHeader("Proxy-Client-IP");
    }
    if (ipAddress == null || ipAddress.length() == 0 ||
"unknown".equalsIgnoreCase(ipAddress)) {
        ipAddress = request.getHeader("WL-Proxy-Client-IP");
    }
    if (ipAddress == null || ipAddress.length() == 0 ||
"unknown".equalsIgnoreCase(ipAddress)) {
        ipAddress = request.getRemoteAddr();
        String localIp = "127.0.0.1";
        String localIpv6 = "0:0:0:0:0:0:0:1";
        if (ipAddress.equals(localIp) || ipAddress.equals(localIpv6)) {
            // 根据网卡取本机配置的IP
            InetAddress inet = null;
            try {
                inet = InetAddress.getLocalHost();
                ipAddress = inet.getHostAddress();
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
        }
    }
    // 对于通过多个代理的情况，第一个IP为客户端真实IP,多个IP按照','分割
    String ipSeparate = ",";
    int ipLength = 15;
    if (ipAddress != null && ipAddress.length() > ipLength) {
        if (ipAddress.indexOf(ipSeparate) > 0) {
            ipAddress = ipAddress.substring(0, ipAddress.indexOf(ipSeparate));
        }
    }
    return ipAddress;
}

/**
 * 响应相应错误
 */
private void responseError(ServletRequest request, ServletResponse response,
String msg, Integer status){
    HttpServletResponse httpResponse = (HttpServletResponse) response;
    httpResponse.setStatus(status);
}
```

```

        httpServletResponse.setCharacterEncoding("utf-8");
        httpServletResponse.setContentType("application/text; charset=utf-8");
        try(PrintWriter out = httpServletResponse.getWriter()){
            out.append("无法访问: ").append(msg);
        }catch (IOException e){
            log.error("responseError返回信息错误"+e.getMessage());
        }
    }

    /**
     * 预处理, 解决跨域问题, 不跨域可去掉
     */
    @Override
    protected boolean preHandle(ServletRequest request, ServletResponse
response)throws Exception{
        HttpServletRequest httpServletRequest = (HttpServletRequest) request;
        HttpServletResponse httpServletResponse = (HttpServletResponse) response;
        httpServletResponse.setHeader("Access-control-Allow-Origin",
httpServletRequest.getHeader("Origin"));
        httpServletResponse.setHeader("Access-Control-Allow-Methods",
"GET,POST,OPTIONS,PUT,DELETE");
        httpServletResponse.setHeader("Access-Control-Allow-Headers",
httpServletRequest.getHeader("Access-Control-Request-Headers"));
        // 跨域时会首先发送一个OPTIONS请求, 这里我们给OPTIONS请求直接返回正常状态
        if (httpServletRequest.getMethod().equals(RequestMethod.OPTIONS.name())) {
            httpServletResponse.setStatus(HttpStatus.OK.value());
            return false;
        }
        return super.preHandle(request, response);
    }
}

```

如果你有认真读过前面对于CasFilter源码的解读, 那这个拦截器实现理解起来也就非常简单。此处还是简述一下它的工作原理:

- 任意一个请求进来后, 进入isAccessAllowed()方法, return false代表所有请求一律拒绝。
- 被拒绝后, 进入onAccessDenied()方法, 执行该类的executeLogin()方法。如果executeLogin()内部出现错误, 则捕获异常并使用responseError()方法回馈错误。状态码403可以根据自己需要修改。
- 进入executeLogin()方法, 调用这个类的createToken()方法并将结果强制转化为我们自定义的CasJwtToken对象。
- 进入createToken()方法, 从请求中获取可能存在的JWT, ticket, 并且获取请求的IP地址, 并且判断请求类型封装成CasJwtToken对象返回。
- 返回executeLogin()方法, 执行Subjetc.login(), 并且将上一步返回的CasJwtToken传入, 此后会进入相应的realm执行doGetAuthenticationInfo()方法。至此, 该拦截器的作用介绍完毕。那么我们可以知道, 验证ticket或者JWT的过程还是交给realm去完成。

第三步: 自定义Realm完成ticket和JWT的验证

在上一步我们把封装好的CasJwtToken对象传入Subjetc.login()方法, 那在realm的doGetAuthenticationInfo()中我们就可以拿到这个对象, 我们就可以提取里面的ticket去验证票据, 提取token(JWT)去校验合法性, 提取IP地

址去校验JWT是否被盗用。同样参考CasRealm。

```
// JWT模式专属的realm,改写自CasRealm
public class CasJwtUserRealm extends AuthorizingRealm{
    public static final String DEFAULT_REMEMBER_ME_ATTRIBUTE_NAME =
"longTermAuthenticationRequestTokenUsed";
    public static final String DEFAULT_VALIDATION_PROTOCOL = "CAS";
    private static final Logger logger =
LoggerFactory.getLogger(CasJwtUserRealm.class);
    private String casServerUrlPrefix;
    private String casService;
    private String validationProtocol = "CAS";
    private TicketValidator ticketValidator;
    private String defaultRoles;
    private String defaultPermissions;

    @Autowired
    private UserService userService;

    // 注入JWT工具
    @Autowired
    private JwtTokenUtils jwtTokenUtil;

    @Override
    public boolean supports(AuthenticationToken token) {
        //必须重写该方法才能支持我们自定义的CasJwtToken
        return token instanceof CasJwtToken;
    }

    {
        super.setName("user");//设置realm的名字, 非常重要
    }

    protected void onInit() {
        super.onInit();
        this.ensureTicketValidator();
    }

    protected TicketValidator ensureTicketValidator() {
        if (this.ticketValidator == null) {
            this.ticketValidator = this.createTicketValidator();
        }

        return this.ticketValidator;
    }

    protected TicketValidator createTicketValidator() {
        String urlPrefix = this.getCasServerUrlPrefix();
        return (TicketValidator)
("saml".equalsIgnoreCase(this.getValidationProtocol()) ? new
Saml11TicketValidator(urlPrefix) : new Cas20ServiceTicketValidator(urlPrefix));
    }
}
```

```
public String getCasServerUrlPrefix() {
    return this.casServerUrlPrefix;
}

public void setCasServerUrlPrefix(String casServerUrlPrefix) {
    this.casServerUrlPrefix = casServerUrlPrefix;
}

public String getCasService() {
    return this.casService;
}

public void setCasService(String casService) {
    this.casService = casService;
}

public String getValidationProtocol() {
    return this.validationProtocol;
}

public void setValidationProtocol(String validationProtocol) {
    this.validationProtocol = validationProtocol;
}

public String getDefaultRoles() {
    return this.defaultRoles;
}

public void setDefaultRoles(String defaultRoles) {
    this.defaultRoles = defaultRoles;
}

public String getDefaultPermissions() {
    return this.defaultPermissions;
}

public void setDefaultPermissions(String defaultPermissions) {
    this.defaultPermissions = defaultPermissions;
}

@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
    //获取当前登录输入的用户名，等价于(String)
principalCollection.fromRealm(getName()).iterator().next();
    logger.info("=====进入授权领域, Realm:" + this.getName() +
"=====");
    String loginName =
(String)super.getAvailablePrincipal(principalCollection);
    //到数据库查是否有此对象
    User user=userService.getRolesByUsername(loginName);// 实际项目中，这里可以
根据实际情况做缓存，如果不做，Shiro自己也是有时间间隔机制，2分钟内不会重复执行该方法
```

```

        System.out.println(user.getRoles());
        if(!CollectionUtils.isEmpty(user.getRoles())){
            // TODO 加载权限
            return simpleAuthorizationInfo;
        }
        // 返回null的话, 就会导致任何用户访问被拦截的请求时, 都会自动跳转到
        unauthorizedUrl指定的地址
        return null;
    }

    @Override
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
        if(ObjectUtils.isEmpty(token)){
            throw new AuthenticationException("Authentication token is null!");
        }
        //
        return null;

        // 读取casJwtToken中数据
        CasJwtToken casJwtToken = (CasJwtToken) token;
        String ticket = casJwtToken.getTicket();
        String jwt = casJwtToken.getToken();
        String IPAddress = casJwtToken.getIPAddress();
        CasJwtTokenMode mode = casJwtToken.getMode();

        Map<String, Object> attributes_last = new HashMap<>();//此处暂时置空
        // 注意: 如果ticket为空, token不为空说明是访问资源;如果ticket不为空, 而token为空
        那说明是CAS登录; 否则出错
        if(ObjectUtils.isEmpty(mode)){
            throw new AuthenticationException("jwt mode is null!");
        }
        //
        }

        // 判断请求类型
        if(mode.equals(CasJwtTokenMode.CasLogin)){
            // CAS回调类型 验证票据
            if(StringUtils.isEmpty(ticket) || StringUtils.isBlank(ticket)){
                throw new AuthenticationException("ticket is null or blank!");
            }
            //
            return null;
        }
        // 进入Cas登录回调后的步骤: 鉴别ticket、判断数据库是否存在接入用户、生成
        token

        TicketValidator ticketValidator = this.ensureTicketValidator();
        try{
            // 鉴别ticket
            Assertion casAssertion = ticketValidator.validate(ticket,
this.getCasService());
            // 鉴别成功获取CAS返回的各种信息
            AttributePrincipal casPrincipal = casAssertion.getPrincipal();
            username = casPrincipal.getName();
            logger.debug("Validate ticket : {} in CAS server : {} to retrieve
user : {}", ticket, this.getCasServerUrlPrefix(), username);
            Map<String, Object> attributes = casPrincipal.getAttributes();
            casJwtToken.setUsername(username);

```

```
        department = (String) attributes.get(Constant.DEPARTMENT);
        displayName = (String) attributes.get(Constant.DISPLAY_NAME);

        /* 为用户创建一个JWT */
        // 从CAS 返回信息中获取用户登录IP
        String clientIpAddress = (String)
attributes.get(Constant.CLIENT_IP_ADDRESS);
        if(StringUtils.isEmpty(clientIpAddress) ||
StringUtils.isBlank(clientIpAddress)){
            throw new AuthenticationException("CAS获取IP地址为null! ");
        }
        String[] strArray = clientIpAddress.split(":");
        clientIpAddress = strArray[0];
        // 生成JWT
        jwt = jwtTokenUtil.generateToken(username, clientIpAddress,
department);
        //jwt = jwtTokenUtil.generateToken(username, IPAddress,
department);
        casJwtToken.setToken(jwt);

    }catch (TicketValidationException e){
        // 鉴别异常
        throw new AuthenticationException(e.getMessage());
    }
}else if(mode.equals(CasJwtTokenMode.UserAccess)){
    // 用户普通访问请求
    if(StringUtils.isEmpty(jwt) || StringUtils.isBlank(jwt)){
        throw new AuthenticationException("jwt is null!");
        //return null;
    }
    // 校验JWTToken
    try {
        // 获取JWT里面保存的IP, 顺便校验JWT
        String IPAddressJwt = jwtTokenUtil.getIpAddressToken(jwt);
        // 获取用户名
        username = jwtTokenUtil.getUsernameFromToken(jwt);
        // 获取可选字段(非必须)
        department = jwtTokenUtil.getDepartmentFromToken(jwt);

        // 校验IP
        if(! IPAddressJwt.equals(IPAddress)){
            throw new LoginIPException("IP地址检查异常! ");
        }
    }catch (JWTDecodeException e){
        throw new JWTDecodeException("JWT解码错误! ");
    }catch (TokenExpiredException e){
        throw new TokenExpiredException("JWT已经过期失效! ");
    }catch (Exception e){
        throw new AuthenticationException(e.getMessage());
    }
}else {
    // 运行错误
    throw new AuthenticationException("Authentication running error!");
}
//
return null;
```

```
    }
    // 判断数据库有无此用户
    User user = userService.getUserByUserName(username);
    if(ObjectUtils.isEmpty(user)){
        // TODO 自定义操作, 新增用户默认分配权限等
    }
    attributes_last.put(Constant.DEPARTMENT, department);
    attributes_last.put(Constant.TOKEN, jwt);
    user.setPassword(null);
    user.setRoles(userService.getRolesByUserName(username).getRoles());
    List<Object> principals = CollectionUtils.asList(username,
attributes_last);
    PrincipalCollection principalCollection = new
SimplePrincipalCollection(principals, this.getName());
    String credentials = null;

    // 根据类型设置凭证
    if(casJwtToken.getMode().equals(CasJwtTokenMode.CasLogin)){
        credentials = ticket;
    }else {
        credentials = jwt;
    }
    return new SimpleAuthenticationInfo(principalCollection, credentials);
}
}
```

同时给出自定义的一个异常代码:

```
public class LoginIPException extends Exception{
    public LoginIPException() {
        super();
    }

    public LoginIPException(String message) {
        super(message);
    }
}
```

第四步: 实现无状态的StatelessDefaultSubjectFactory

由于JWT认证是无状态的, 不需要创建session。创建StatelessDefaultSubjectFactory去继承DefaultWebSubjectFactory并重写createSubject()方法。

```
public class StatelessDefaultSubjectFactory extends DefaultWebSubjectFactory {
    @Override
    public Subject createSubject(SubjectContext context) {
        // 不创建session
        context.setSessionCreationEnabled(false);
        return super.createSubject(context);
    }
}
```

```
}  
}
```

第五步：完成ShiroConfig

已经完成前面的工作后，可以实现Shiro的配置了。

```
@Configuration  
public class JwtShiroConfig {  
  
    /** Cas服务前缀*/  
    @Value("${cas.casServer.urlPrefix}")  
    private String casServerUrlPrefix;  
  
    /** Cas登录页面地址 */  
    @Value("${cas.casServer.loginUrl}")  
    private String casLoginUrl;  
  
    /** Cas登出页面地址 */  
    @Value("${cas.casServer.logoutUrl}")  
    private String casLogoutUrl;  
  
    /** 当前工程对外提供的服务地址 */  
    @Value("${cas.client.urlPrefix}")  
    private String shiroServerUrlPrefix;  
    /** Cas回调地址 */  
    @Value("${cas.client.casCallback}")  
    private String casCallback;  
    /** 登录的具体地址 */  
    @Value("${cas.casServer.loginRequestUrl}")  
    private String loginUrl;  
    /** 成功地址 */  
    @Value("${cas.client.successUrl}")  
    private String successUrl;  
    /** 未授权地址 */  
    @Value("${cas.client.unauthorizedUrl}")  
    private String unauthorizedUrl;  
  
    @Autowired  
    private JwtTokenUtils jwtTokenUtil;  
  
    @Bean(name = "casUserRealm")  
    public CasJwtUserRealm casUserRealm() {  
        CasJwtUserRealm userRealm = new CasJwtUserRealm();  
        userRealm.setCachingEnabled(true);  
        userRealm.setCachingEnabled(true);  
        userRealm.setAuthenticationCachingEnabled(false);  
        userRealm.setAuthorizationCachingEnabled(true);  
        userRealm.setAuthorizationCacheName("CasAuthorizationCache");  
        userRealm.setCasServerUrlPrefix(casServerUrlPrefix);  
    }  
}
```

```
        userRealm.setCasService(shiroServerUrlPrefix + casCallback);
        return userRealm;
    }

    @Bean(name= "redisCacheManager")
    public RedisCacheManager redisCacheManager(){
        return new RedisCacheManager();
    }

    @Bean(name = "userRealm")
    public UserRealm userRealm(RedisCacheManager redisCacheManager){
        UserRealm userRealm = new UserRealm();
        //创建hash校验匹配器
        HashedCredentialsMatcher credentialsMatcher = new
HashedCredentialsMatcher();
        //设置散列算法
        credentialsMatcher.setHashAlgorithmName("MD5");
        //设置散列次数
        credentialsMatcher.setHashIterations(64);
        userRealm.setCredentialsMatcher(credentialsMatcher);
        userRealm.setCacheManager(redisCacheManager);
        userRealm.setCachingEnabled(false);
        userRealm.setAuthenticationCachingEnabled(false);
        userRealm.setAuthorizationCachingEnabled(true);
        userRealm.setAuthorizationCacheName("authorizationCache");
        return userRealm;
    }

    @Bean
    public FilterRegistrationBean delegatingFilterProxy() {
        FilterRegistrationBean filterRegistrationBean = new
FilterRegistrationBean();
        DelegatingFilterProxy proxy = new DelegatingFilterProxy();
        proxy.setTargetFilterLifecycle(true);
        proxy.setTargetBeanName("shiroFilterFactoryBean");
        filterRegistrationBean.setFilter(proxy);
        return filterRegistrationBean;
    }

    @Bean(name = "shiroFilterFactoryBean")
    public ShiroFilterFactoryBean shiroFilter(DefaultWebSecurityManager
securityManager) {
        ShiroFilterFactoryBean shiroFilter = new ShiroFilterFactoryBean();
        //Shiro的核心安全接口,这个属性是必须的
        shiroFilter.setLoginUrl(loginUrl);
        shiroFilter.setSecurityManager(securityManager);
        Map<String, Filter> filters = new LinkedHashMap<>();
        //创建自定义拦截器对象
        CustomCallbackFilter casCallbackFilter = new CustomCallbackFilter();
        casCallbackFilter.setSuccessUrl(successUrl);
        shiroFilter.setUnauthorizedUrl(unauthorizedUrl);
        //将所有认证与casCallbackFilter关联
        filters.put("authc", casCallbackFilter);
        //设置登出拦截器
```

```

        LogoutFilter logout = new LogoutFilter();
        logout.setRedirectUrl(casLogoutUrl);
        filters.put("logout", logout);
        shiroFilter.setFilters(filters);
        Map<String, String> filterMap = new LinkedHashMap<>();
        /* 过滤链定义, 从上向下顺序执行, 一般将 / ** 放在最为下边;
           authc:所有url都必须认证通过才可以访问; anon:所有url都可以匿名访问 */
        // TODO 从数据库加载规则

        /* 注意: 这里必须要自己设定一个回调controller接口*/
        filterMap.put("/daku/test/cas/callback", "authc");
        filterMap.put("daku/logout", "logout");
        filterMap.put("daku/public/logout", "logout");
        shiroFilter.setFilterChainDefinitionMap(filterMap);
        return shiroFilter;
    }

    @Bean
    public DefaultWebSubjectFactory subjectFactory(){
        // 设置无状态subjectFactory
        DefaultWebSubjectFactory subjectFactory = new
        StatelessDefaultSubjectFactory();
        return subjectFactory;
    }

    /**
     * 安全管理器, Shiro的核心组件, 如果不用设置多realm, 参数少一个realm, 然后设置一个
     * realm就行, 多realm的代码可以去掉
     */
    @Bean
    public DefaultWebSecurityManager securityManager(DefaultWebSubjectFactory
    subjectFactory, DefaultSessionManager sessionManager, CasJwtUserRealm
    casUserRealm, UserRealm userRealm) {
        DefaultWebSecurityManager securityManager = new
        DefaultWebSecurityManager();
        // 设置单个realm
        // securityManager.setRealm(casUserRealm);
        securityManager.setCacheManager(new RedisCacheManager());
        // 替换默认的DefaultSubjectFactory, 用于关闭session功能
        securityManager.setSubjectFactory(subjectFactory);
        securityManager.setSessionManager(sessionManager);
        // 关闭session存储, 禁用Session作为存储策略的实现, 但它没有完全地禁用Session所
        // 以需要配合SubjectFactory中的context.setSessionCreationEnabled(false)
        ((DefaultSessionStorageEvaluator)
        ((DefaultSubjectDAO)securityManager.getSubjectDAO()).getSessionStorageEvaluator())
        .setSessionStorageEnabled(false);
        SecurityUtils.setSecurityManager(securityManager);

        //设置多realm
        List<Realm> realms = new ArrayList<>();
        realms.add(casUserRealm);
        realms.add(userRealm);
        // 设置多realm相关: 关于MyModularRealmAuthorizer与
        MyModularRealmAuthenticator后面会给出实现
    }

```



```

        MyModularRealmAuthorizer authorizer = new MyModularRealmAuthorizer();
        authorizer.setRealms(realms);
        securityManager.setAuthorizer(authorizer);

        MyModularRealmAuthenticator authenticator = new
MyModularRealmAuthenticator();
        authenticator.setRealms(realms);
        securityManager.setAuthenticator(authenticator);
        return securityManager;
    }

    @Bean
    public DefaultSessionManager sessionManager(){
        DefaultSessionManager sessionManager =new DefaultSessionManager();
        // 关闭session定时检查, 通过setSessionValidationSchedulerEnabled禁用掉会话调
度器
        sessionManager.setSessionValidationSchedulerEnabled(false);
        return sessionManager;
    }

    /**
     * 下面的代码是添加注解支持, 必须添加, 否则会引起@Bean注入为null
     */
    @Bean
    public DefaultAdvisorAutoProxyCreator advisorAutoProxyCreator() {
        DefaultAdvisorAutoProxyCreator advisorAutoProxyCreator = new
DefaultAdvisorAutoProxyCreator();
        advisorAutoProxyCreator.setProxyTargetClass(true);
        return advisorAutoProxyCreator;
    }

    @Bean
    public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(DefaultWebSecurityManager securityManager) {
        AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor =
new AuthorizationAttributeSourceAdvisor();
        authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);
        return authorizationAttributeSourceAdvisor;
    }
}

```

第六步：在Controller中添加回调接口

为什么？ 因为这种实现，该拦截器是完全被动的，回调验证票据后并不会自己重定向，需要通过cotroller设置一个接口，回调验票后自动调用这个接口将生成的JWT回馈到前端，特别是前后端分离项目中必须使用。请对应到上一步shiroFilter中的设置。

```

@Controller
@RequestMapping("/daku/test")
public class TestController {

```

```

@Value("${cas.client.successUrl}")
private String successUrl;

@Autowired
private UserService userService;

@ApiOperation(value = "回调")
@GetMapping("/cas/callback")
@CrossOrigin(allowCredentials = "true")
public String callback(HttpServletRequest request, HttpServletResponse
response) throws Exception {
    String username = (String) SecurityUtils.getSubject().getPrincipal();
    HashMap<String, String> map = (HashMap<String, String>)
SecurityUtils.getSubject().getPrincipals().asList().get(1);
    //JWT存放位置
    //System.out.println(map.get(Constant.TOKEN));
    response.setHeader("content-type", "text/html;charset=utf-8");
    response.setHeader(Constant.TOKEN, map.get(Constant.TOKEN));
    //将JWT放在跳转地址参数
    response.sendRedirect(successUrl + "?token=" + map.get(Constant.TOKEN));
    return username;
}
}

```

至此，JWT认证实现的基本代码已经展示完毕。下面我们补充一下前面没有添加的代码。

多Realm认证实现

实现原理：MyModularRealmAuthenticator继承ModularRealmAuthenticator重写doAuthenticate()方法；

MyModularRealmAuthorizer继承ModularRealmAuthorizer重写isPermitted()、hasRole()方法。

MyModularRealmAuthenticator具体代码：

```

public class MyModularRealmAuthenticator extends ModularRealmAuthenticator {
    @Override
    protected AuthenticationInfo doAuthenticate(AuthenticationToken
authenticationToken)
        throws AuthenticationException {
        // 判断getRealms()是否返回为空
        assertRealmsConfigured();
        String loginType = null;
        //判断token类型
        //这里展示的是方案二、被注释掉的代码是方案一下面同理
        // if(authenticationToken instanceof CasToken) {
        //     loginType = "user";
        // }
        if (authenticationToken instanceof CasJwtToken){
            loginType = "user";
        }else if(authenticationToken instanceof UsernamePasswordToken){
            loginType = "admin";
        }else{
            log.error(this.getClass().getName() + " does not support this class of
authenticationToken:" + authenticationToken.getClass().toString());

```

```

        throw new ClassCastException(this.getClass().getName() + " does not
support this class of
authenticationToken:"+authenticationToken.getClass().toString());
    }

    // 所有Realm
    Collection<Realm> realms = getRealms();
    // 登录类型对应的所有Realm
    Collection<Realm> typeRealms = new ArrayList<>();
    for (Realm realm : realms) {
        if (realm.getName().equals(loginType))
            typeRealms.add(realm);
    }

    // 判断是单Realm还是多Realm
    if (typeRealms.size() == 1)
        return doSingleRealmAuthentication(typeRealms.iterator().next(),
authenticationToken);
    else
        return doMultiRealmAuthentication(typeRealms, authenticationToken);
    }
}

```

MyModularRealmAuthorizer具体代码:

```

public class MyModularRealmAuthorizer extends ModularRealmAuthorizer {
    @Override
    public boolean isPermitted(PrincipalCollection principals, String permission)
    {
        assertRealmsConfigured();
        Set<String> realmNames = principals.getRealmNames();
        //获取realm的名字
        String realmName = realmNames.iterator().next();
        for (Realm realm : getRealms()) {
            if (!(realm instanceof Authorizer)) continue;
            //匹配名字
            if(realmName.equals("admin")) {
                if (realm instanceof UserRealm) {
                    return ((UserRealm) realm).isPermitted(principals,
permission);
                }
            }
            if(realmName.equals("user")) {
                //
                if (realm instanceof MyShiroCasRealm) {
                    //
                    return ((MyShiroCasRealm) realm).isPermitted(principals,
permission);
                }
                if (realm instanceof CasJwtUserRealm) {
                    return ((CasJwtUserRealm) realm).isPermitted(principals,
permission);
                }
            }
        }
    }
}

```

```

    }
    }
    return false;
}

@Override
public boolean isPermitted(PrincipalCollection principals, Permission
permission) {
    assertRealmsConfigured();
    Set<String> realmNames = principals.getRealmNames();
    //获取realm的名字
    String realmName = realmNames.iterator().next();
    for (Realm realm : getRealms()) {
        if (!(realm instanceof Authorizer)) continue;
        //匹配名字
        if(realmName.equals("admin")) {
            if (realm instanceof UserRealm) {
                return ((UserRealm) realm).isPermitted(principals,
permission);
            }
        }
        //匹配名字
        if(realmName.equals("user")) {
            // if (realm instanceof MyShiroCasRealm) {
            // return ((MyShiroCasRealm) realm).isPermitted(principals,
permission);
            // }
            if (realm instanceof CasJwtUserRealm) {
                return ((CasJwtUserRealm) realm).isPermitted(principals,
permission);
            }
        }
    }
    return false;
}

@Override
public boolean hasRole(PrincipalCollection principals, String roleIdentifier)
{
    assertRealmsConfigured();
    Set<String> realmNames = principals.getRealmNames();
    //获取realm的名字
    String realmName = realmNames.iterator().next();
    for (Realm realm : getRealms()) {
        if (!(realm instanceof Authorizer)) continue;
        //匹配名字
        if(realmName.equals("admin")) {
            if (realm instanceof UserRealm) {
                return ((UserRealm) realm).hasRole(principals,
roleIdentifier);
            }
        }
        //匹配名字
        if(realmName.equals("user")) {
            // if (realm instanceof MyShiroCasRealm) {

```

```

//                return ((MyShiroCasRealm) realm).hasRole(principals,
roleIdIdentifier);
//            }
            if (realm instanceof CasJwtUserRealm) {
                return ((CasJwtUserRealm) realm).hasRole(principals,
roleIdIdentifier);
            }
        }
        return false;
    }
}

```

关于ShiroConfig内配置，方案一、二都在给出securityManager给出，不再赘述。

redis缓存管理器实现

使用redis替代默认的shiro内的ehcache。原理：RedisCacheManager实现CacheManager的getCache()方法，

```

/**
 * @author chh
 * @date 2022/2/19 20:06
 * 自定义Redis缓存管理器实现CacheManager
 */
public class RedisCacheManager implements CacheManager {
    /*此处参数为CacheName*/
    @Override
    public <K, V> Cache<K, V> getCache(String cacheName) throws CacheException {
        return new RedisCache<K, V>(cacheName);
    }
}

```

同时：创建RedisCache<k,v>实现Cache<k,v>。

```

/**
 * @author chh
 * @date 2022/2/19 20:10
 * 自定义RedisCache实现Cache
 */
@Slf4j
public class RedisCache<k,v> implements Cache<k,v> {
    private String cacheName;

    public RedisCache() {
    }

    public RedisCache(String cacheName) {
        this.cacheName = cacheName;
    }
}

```

```
}

@Override
public v get(k k) throws CacheException {
    String key = null;
    try {
        key = getKey(k);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    System.out.println("get:" + key);
    return (v) getRedisTemplate().opsForHash().get(this.cacheName, key);
}

@Override
public v put(k k, v v) throws CacheException {
    String key = null;
    try {
        key = getKey(k);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    System.out.println("put key:" + k);
    System.out.println("put value:" + v);
    getRedisTemplate().opsForHash().put(this.cacheName, key, v);
    return null;
}

@Override
public v remove(k k) throws CacheException {
    String key = null;
    try {
        key = getKey(k);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return (v) getRedisTemplate().opsForHash().delete(this.cacheName, key);
}

@Override
public void clear() throws CacheException {
    getRedisTemplate().delete(this.cacheName);
}

@Override
public int size() {
    return getRedisTemplate().opsForHash().size(this.cacheName).intValue();
}

@Override
public Set<k> keys() {
    return (Set<k>) getRedisTemplate().opsForHash().keys(this.cacheName);
}
```

```
@Override
public Collection<v> values() {
    return (Collection<v>)
getRedisTemplate().opsForHash().values(this.cacheName);
}

private RedisTemplate getRedisTemplate(){
    RedisTemplate redisTemplate = (RedisTemplate)
ApplicationContextUtils.getBean("redisTemplate");
    redisTemplate.setKeySerializer(new StringRedisSerializer());
    redisTemplate.setHashKeySerializer(new StringRedisSerializer());
    return redisTemplate;
}

private String getKey(k k) throws ClassNotFoundException {
    String key = null;
    if(k instanceof SimplePrincipalCollection){
        SimplePrincipalCollection collection = (SimplePrincipalCollection) k;
        key = (String) collection.getPrimaryPrincipal();
    }else if(k instanceof String){
        key = (String) k;
    }else {
        log.error(this.getClass().getName() + " does not support this class of
key:"+k.getClass().toString());
        throw new ClassNotFoundException(this.getClass().getName() + " does
not support this class of key:"+k.getClass().toString());
    }
    return key;
}
}
```

代码说明:

由于Cache<k,v>的键值k, v使用泛型, 底层代码在调用时并不预知它的类型, 更何况我们使用的时多realm认证, 我们在对待k时, 先传入getKey()方法中去使用instanceof判断类型(有可能是SimplePrincipalCollection有可能是String), 从里面获取真正我们需要的key然乎返回。此外, 我们在redis里面使用opsForHash()即hashmap形式存放, 顶层是cacheName, 然后按key存放, 可以自己读一下代码。

关于

联系方式: chenhuhang@mail.ustc.edu.cn