第5章 权限管理与Shiro入门

学习目标：

- 理解前端权限控制思路
- 理解有状态服务和无状态服务
- 通过拦截器实现JWT鉴权
- 能够理解shiro以及shiro的认证和授权

1 前端权限控制

1.1 需求分析

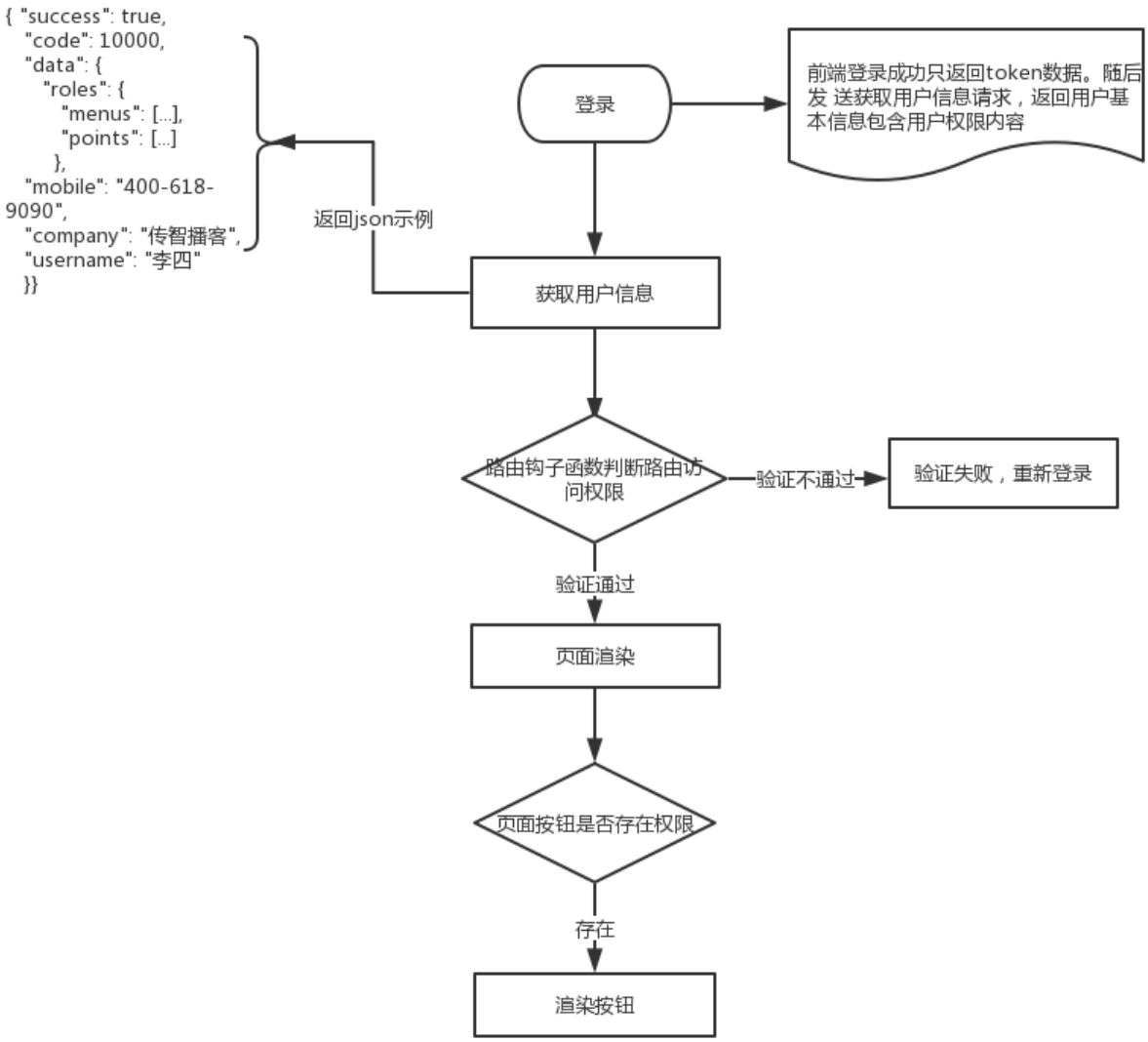
1.1.1 需求说明

基于前后端分离的开发模式中，权限控制分为前端页面可见性权限与后端API接口可访问行权限。前端的权限控制主要围绕在菜单是否可见，以及菜单中按钮是否可见两方面展开的。

1.1.2 实现思路

在vue工程中，菜单可以简单的理解为vue中的路由，只需要根据登录用户的权限信息动态的加载路由列表就可以动态的构造出访问菜单。

1. 登录成功后获取用户信息，包含权限列表（菜单权限，按钮权限）
2. 根据用户菜单权限列表，动态构造路由（根据路由名称和权限标识比较）
3. 页面按钮权限通过自定义方法控制可见性



1.2 服务端代码实现

对系统微服务的FrameController的profile方法（获取用户信息接口）进行修改，添加权限信息

```
    /**
     * 获取个人信息
     */
    @RequestMapping(value = "/profile", method = RequestMethod.POST)
    public Result profile(HttpServletRequest request) throws Exception {
        //请求中获取key为Authorization的头信息
        String authorization = request.getHeader("Authorization");
        if(StringUtils.isEmpty(authorization)) {
            throw new CommonException(ResultCode.UNAUTHENTICATED);
        }
        //前后端约定头信息内容以  Bearer+空格+token 形式组成
        String token = authorization.replace("Bearer ", "");
        //比较并获取claims
        Claims claims = jwtUtil.parseJWT(token);
        if(claims == null) {
```

```
            throw new CommonException(ResultCode.UNAUTHENTICATED);
        }
        //查询用户
        User user = userService.findById(userId);
        ProfileResult result = null;

        if("user".equals(user.getLevel())) {
            result = new ProfileResult(user);
        }else {
            Map map = new HashMap();
            if("coAdmin".equals(user.getLevel())) {
                map.put("enVisible","1");
            }
            List<Permission> list = permissionService.findAll(map);
            result = new ProfileResult(user,list);
        }
        return new Result(ResultCode.SUCCESS,result);
    }
```

1.3 前端代码实现

1.3.1 路由钩子函数

vue路由提供的钩子函数（beforeEach）主要用来在加载之前拦截导航，让它完成跳转或取消。可以在路由钩子函数中进行校验是否对某个路由具有访问权限

```
router.beforeEach((to, from, next) => {
  NProgress.start() // start progress bar
  if (getToken()) {
    // determine if there has token
    /* has token */
    if (to.path === '/login') {
      next({path: '/'})
      NProgress.done() // if current page is dashboard will not trigger afterEach hook,
so manually handle it
    } else {
      if (store.getters.roles.length === 0) {
        // 判断当前用户是否已拉取完user_info信息
        store
          .dispatch('GetUserInfo')
          .then(res => {
            // 拉取user_info
            const roles = res.data.data.roles // note: roles must be a array! such as:
['editor','develop']
            store.dispatch('GenerateRoutes', {roles}).then(() => {
              // 根据roles权限生成可访问的路由表
              router.addRoutes(store.getters.addRouters) // 动态添加可访问路由表
              next({...to, replace: true}) // hack方法 确保addRoutes已完成 ,set the
replace: true so the navigation will not leave a history record
            })
          })
          .catch(() => {
```

```
            store.dispatch('FedLogOut').then(() => {
              Message.error('验证失败，请重新登录')
              next({path: '/login'})
            })
          })
      } else {
        next()
      }
    }
  } else {
    /* has no token */
    if (whiteList.indexOf(to.path) !== -1) {
      // 在免登录白名单，直接进入
      next()
    } else {
      next('/login') // 否则全部重定向到登录页
      NProgress.done() // if current page is login will not trigger afterEach hook, so
manually handle it
    }
  }
})
```

1.3.2 配置菜单权限

在 `\src\module-dashboard\store\permission.js` 下进行修改，开启路由配置

```
  actions: {
    GenerateRoutes({ commit }, data) {
      return new Promise(resolve => {
        const { roles } = data
        //动态构造权限列表
        let accessedRouters = filterAsyncRouter(asyncRouterMap, roles)
        commit('SET_ROUTERS', accessedRouters)
        //commit('SET_ROUTERS', asyncRouterMap) // 调试开启全部路由
        resolve()
      })
    }
  }
```

1.3.3 配置验证权限的方法

找到 `\src\utils\permission.js` 配置验证是否具有权限的验证方法

```
import store from '@/store'

// 检查是否有权限
export function hasPermission(roles, route) {
  if (roles.menus && route.name) {
    return roles.menus.some(role => {
      return route.name.toLowerCase() === role.toLowerCase()
    })
  } else {
```

```
    return false
  }
}

// 检查是否有权限点
export function hasPermissionPoint(point) {
  let points = store.getters.roles.points
  if (points) {
    return points.some(it => it.toLowerCase() === point.toLowerCase())
  } else {
    return false
  }
}
```

1.3.4 修改登录和获取信息的请求接口

（1）关闭模拟测试接口

`\mock\index.js` 中不加载登录（login）以及（profile）的模拟测试

```
import Mock from 'mockjs'
import TableAPI from './table'
import ProfileAPI from './profile'
import LoginAPI from './login'

Mock.setup({
  //timeout: '1000'
})

Mock.mock(/\/table\/list\.*/, 'get', TableAPI.list)
//Mock.mock(/\/frame\/profile/, 'post', ProfileAPI.profile)
//Mock.mock(/\/frame\/login/, 'post', LoginAPI.login)
```

1.4 权限测试

（1）菜单测试

分配好权限之后，重新登录前端页面，左侧菜单已经发生了变化。

（2）对需要进行权限控制的（权限点）验证测试

页面添加校验方法

```
  methods: {
    checkPoint(point){
      return hasPermissionPoint(point);
    }
  }
```

使用v-if验证权限是否存在，其中参数为配置的权限点标识

```
<el-button  type="primary" v-if="checkPoint('POINT-USER-ADD')" size="mini" icon="el-icon-plus" @click="handlAdd">新增员工</el-button>
```

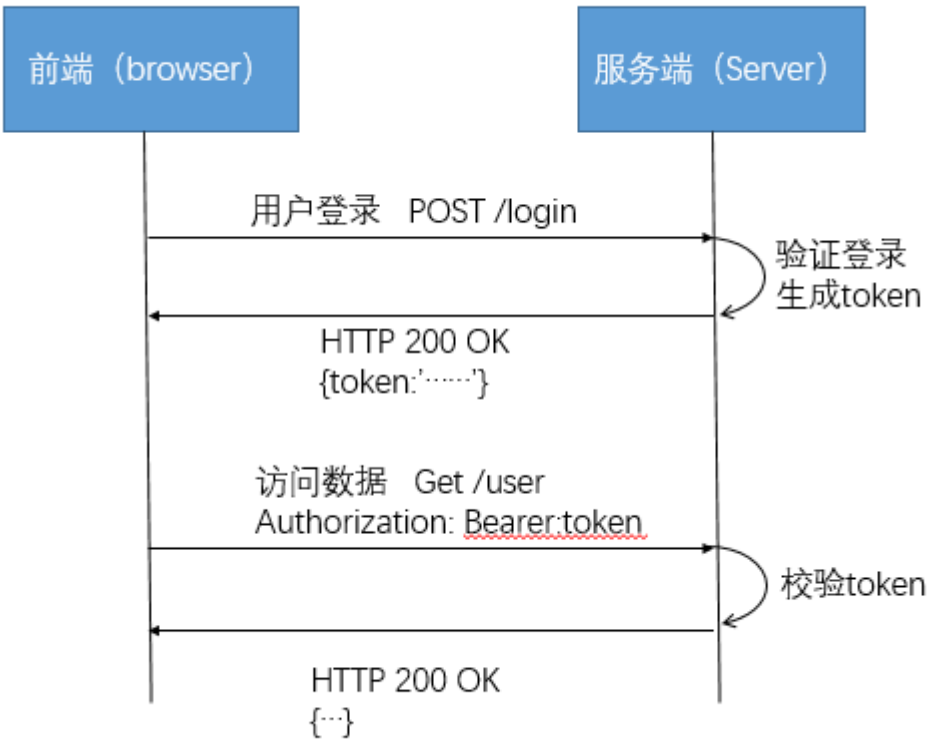
2 有状态服务和无状态服务

2.1 什么是服务中的状态

有状态和无状态服务是两种不同的服务架构，两者的不同之处在于对于服务状态的处理。服务状态是服务请求所需的数据，它可以是一个变量或者一个数据结构。无状态服务不会记录服务状态，不同请求之间也是没有任何关系；而有状态服务则反之。对服务器程序来说，究竟是有状态服务，还是无状态服务，其判断依据——两个来自相同发起者的请求在服务器端是否具备上下文关系。

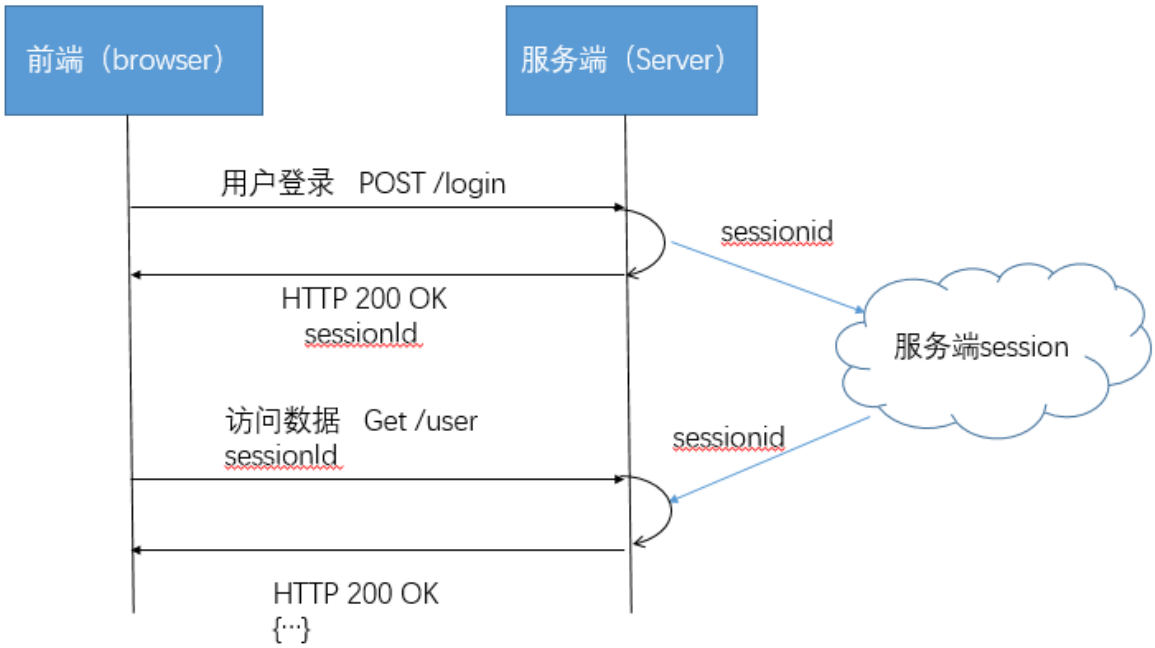
2.2 无状态服务

无状态请求，服务器端所能够处理的数据全部来自于请求所携带的信息，无状态服务对于客户端的单次请求的处理，不依赖于其他请求，处理一次请求的信息都包含在该请求里。最典型的就是通过cookie保存token的方式传输请求数据。也可以理解为Cookie是通过客户端保持状态的解决方案。



2.3 有状态服务

有状态服务则相反，服务会存储请求上下文相关的数据信息，先后的请求是可以有关联的。例如，在Web 应用中，经常会使用Session 来维系登录用户的上下文信息。虽然http 协议是无状态的，但是借助Session，可以使 http 服务转换为有状态服务



3 基于JWT的API鉴权

3.1 基于拦截器的token与鉴权

如果我们每个方法都去写一段代码，冗余度太高，不利于维护，那如何做使我们的代码看起来更清爽呢？我们可以将这段代码放入拦截器去实现

3.1.1 Spring中的拦截器

Spring为我们提供了org.springframework.web.servlet.handler.HandlerInterceptorAdapter这个适配器，继承此类，可以非常方便的实现自己的拦截器。他有三个方法：分别实现预处理、后处理（调用了Service并返回ModelAndView，但未进行页面渲染）、返回处理（已经渲染了页面）

> 1.在preHandle中，可以进行编码、安全控制等处理； 2.在postHandle中，有机会修改ModelAndView； 3.在afterCompletion中，可以根据ex是否为null判断是否发生了异常，进行日志记录。

3.2 签发用户API权限

在系统微服务的`com.ihrm.system.controller.UserController`修改签发token的登录服务添加API权限

```
    /**
     * 用户登录
     *  1.通过service根据mobile查询用户
     *  2.比较password
     *  3.生成jwt信息
     *
     */
    @RequestMapping(value="/login",method = RequestMethod.POST)
    public Result login(@RequestBody Map<String,String> loginMap) {
```

```
        String mobile = loginMap.get("mobile");
        String password = loginMap.get("password");
        User user = userService.findByMobile(mobile);
        //登录失败
        if(user == null || !user.getPassword().equals(password)) {
            return new Result(ResultCode.MOBILEORPASSWORDERROR);
        }else {
            //登录成功
            //api权限字符串
            StringBuilder sb = new StringBuilder();
            //获取到所有的可访问API权限
            for (Role role : user.getRoles()) {
                for (Permission perm : role.getPermissions()) {
                    if(perm.getType() == PermissionConstants.PERMISSION_API) {
                        sb.append(perm.getCode()).append(",");
                    }
                }
            }
            Map<String,Object> map = new HashMap<>();
            map.put("apis",sb.toString());//可访问的api权限字符串
            map.put("companyId",user.getCompanyId());
            map.put("companyName",user.getCompanyName());
            String token = jwtUtils.createJwt(user.getId(), user.getUsername(), map);
            return new Result(ResultCode.SUCCESS,token);
        }
    }
```

3.3 拦截器中鉴权

（1）在ihrm-common下添加拦截器 JwtInterceptor

```
@Component
public class JwtInterceptor extends HandlerInterceptorAdapter {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        // 1.通过request获取请求token信息
        String authorization = request.getHeader("Authorization");
        //判断请求头信息是否为空，或者是否已Bearer开头
        if(!StringUtils.isEmpty(authorization) && authorization.startsWith("Bearer")) {
            //获取token数据
            String token = authorization.replace("Bearer ","");
            //解析token获取claims
            Claims claims = jwtUtils.parseJwt(token);
            if(claims != null) {
                //通过claims获取到当前用户的可访问API权限字符串
                String apis = (String) claims.get("apis");  //api-user-delete,api-user-
update
```

```
                //通过handler
                HandlerMethod h = (HandlerMethod) handler;
                //获取接口上的reqeustmapping注解
                RequestMapping annotation =
h.getMethodAnnotation(RequestMapping.class);
                //获取当前请求接口中的name属性
                String name = annotation.name();
                //判断当前用户是否具有响应的请求权限
                if(apis.contains(name)) {
                    request.setAttribute("user_claims",claims);
                    return true;
                }else {
                    throw new CommonException(ResultCode.UNAUTHORISE);
                }
            }
        }
        throw new CommonException(ResultCode.UNAUTHENTICATED);
    }
}
```

（2）修改UserController的profile方法

```
    /**
     * 用户登录成功之后，获取用户信息
     *      1.获取用户id
     *      2.根据用户id查询用户
     *      3.构建返回值对象
     *      4.响应
     */
    @RequestMapping(value="/profile",method = RequestMethod.POST)
    public Result profile(HttpServletRequest request) throws Exception {

        String userid = claims.getId();
        //获取用户信息
        User user = userService.findById(userid);
        //根据不同的用户级别获取用户权限

        ProfileResult result = null;

        if("user".equals(user.getLevel())) {
            result = new ProfileResult(user);
        }else {
            Map map = new HashMap();
            if("coAdmin".equals(user.getLevel())) {
                map.put("enVisible","1");
            }
            List<Permission> list = permissionService.findAll(map);
            result = new ProfileResult(user,list);
        }
        return new Result(ResultCode.SUCCESS,result);
    }
```

（3）配置拦截器类,创建com.ihrm.system.SystemConfig

```
package com.ihrm.system;

import com.ihrm.common.interceptor.JwtInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;

@Configuration
public class SystemConfig extends WebMvcConfigurationSupport {

    @Autowired
    private JwtInterceptor jwtInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(jwtInterceptor).
        addPathPatterns("/**").
        excludePathPatterns("/frame/login","/frame/register/**"); //设置不拦截的请求地址
    }
}
```

4 Shiro安全框架

4.1 什么是Shiro

4.1.1 什么是Shiro

Apache Shiro是一个强大且易用的Java安全框架,执行**身份验证、授权、密码和会话管理**。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

Apache Shiro 的首要目标是易于使用和理解。安全有时候是很复杂的，甚至是痛苦的，但它没有必要这样。框架应该尽可能掩盖复杂的地方，露出一个干净而直观的 API，来简化开发人员在使他们的应用程序安全上的努力。以下是你可以用 Apache Shiro 所做的事情：

- 验证用户来核实他们的身份
- 对用户执行访问控制，如：

  判断用户是否被分配了一个确定的安全角色

  判断用户是否被允许做某事
- 在任何环境下使用 Session API，即使没有 Web 或 EJB 容器。
- 在身份验证，访问控制期间或在会话的生命周期，对事件作出反应。
- 聚集一个或多个用户安全数据的数据源，并作为一个单一的复合用户“视图”。
- 启用单点登录（SSO）功能。
- 为没有关联到登录的用户启用"Remember Me"服务

4.1.2 与Spring Security的对比

Shiro：

Shiro较之 Spring Security，Shiro在保持强大功能的同时，还在简单性和灵活性方面拥有巨大优势。

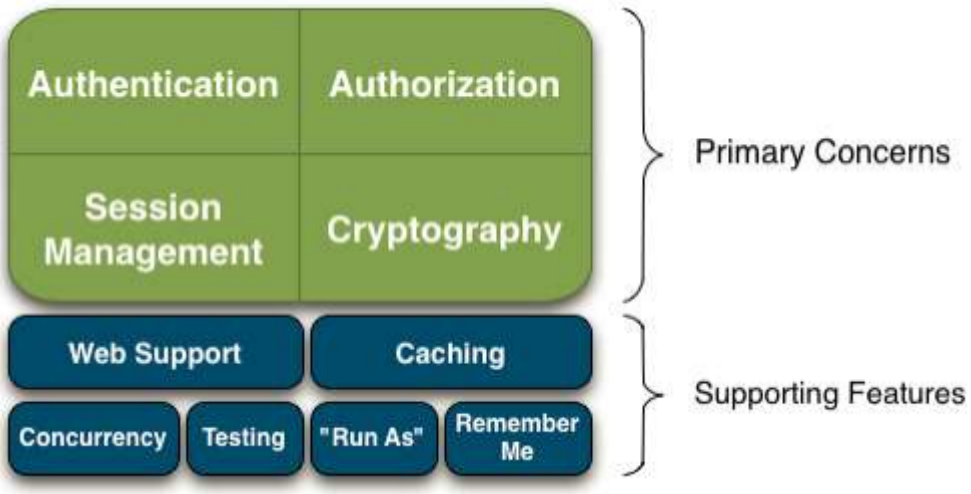
1. 易于理解的 Java Security API；
2. 简单的身份认证（登录），支持多种数据源（LDAP，JDBC，Kerberos，ActiveDirectory 等）；
3. 对角色的简单的签权（访问控制），支持细粒度的签权；
4. 支持一级缓存，以提升应用程序的性能；
5. 内置的基于 POJO 企业会话管理，适用于 Web 以及非 Web 的环境；
6. 异构客户端会话访问；
7. 非常简单的加密 API；
8. 不跟任何的框架或者容器捆绑，可以独立运行

Spring Security：

除了不能脱离Spring，shiro的功能它都有。而且Spring Security对Oauth、OpenID也有支持,Shiro则需要自己手动实现。Spring Security的权限细粒度更高。

4.1.3 Shiro的功能模块

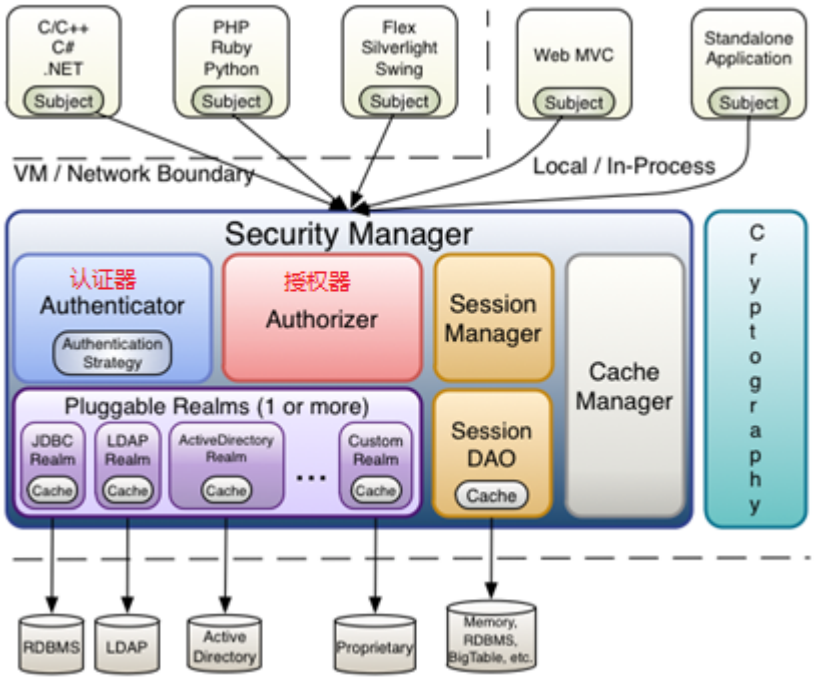
Shiro可以非常容易的开发出足够好的应用，其不仅可以用在JavaSE环境，也可以用在JavaEE环境。Shiro可以帮助我们完成：认证、授权、加密、会话管理、与Web集成、缓存等。这不就是我们想要的嘛，而且Shiro的API也是非常简单；其基本功能点如下图所示：



- Authentication：身份认证/登录，验证用户是不是拥有相应的身份。
- Authorization：授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情。
- Session Management：会话管理，即用户登录后就是一次会话，在没有退出之前，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。
- Cryptography：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储。
- Web Support：Shiro 的 web 支持的 API 能够轻松地帮助保护 Web 应用程序。
- Caching：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率。
- Concurrency：Apache Shiro 利用它的并发特性来支持多线程应用程序。
- Testing：测试支持的存在来帮助你编写单元测试和集成测试，并确保你的能够如预期的一样安全。
- "Run As"：一个允许用户假设为另一个用户身份（如果允许）的功能，有时候在管理脚本很有用。

- "Remember Me"：记住我。

4.2 Shiro的内部结构



Subject：主体，可以看到主体可以是任何可以与应用交互的“用户”；

SecurityManager：相当于SpringMVC中的DispatcherServlet或者Struts2中的FilterDispatcher；是Shiro的心脏；所有具体的交互都通过SecurityManager进行控制；它管理着所有Subject、且负责进行认证和授权、及会话、缓存的管理。

Authenticator：认证器，负责主体认证的，这是一个扩展点，如果用户觉得Shiro默认的不好，可以自定义实现；其需要认证策略（Authentication Strategy），即什么情况下算用户认证通过了；

Authrizer：授权器，或者访问控制器，用来决定主体是否有权限进行相应的操作；即控制着用户能访问应用中的哪些功能；

Realm：可以有1个或多个Realm，可以认为是安全实体数据源，即用于获取安全实体的；可以是JDBC实现，也可以是LDAP实现，或者内存实现等等；由用户提供；注意：Shiro不知道你的用户/权限存储在哪及以何种格式存储；所以我们一般在应用中都需要实现自己的Realm；

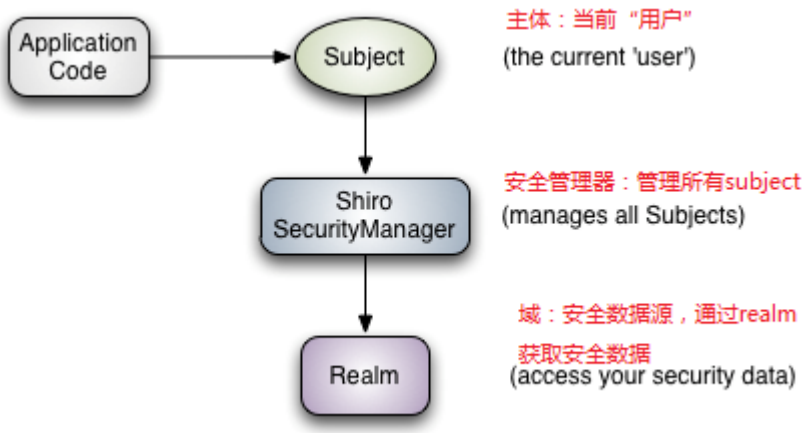
SessionManager：如果写过Servlet就应该知道Session的概念，Session呢需要有人去管理它的生命周期，这个组件就是SessionManager；而Shiro并不仅仅可以用在Web环境，也可以用在如普通的JavaSE环境、EJB等环境；所有呢，Shiro就抽象了一个自己的Session来管理主体与应用之间交互的数据；

SessionDAO：DAO大家都用过，数据访问对象，用于会话的CRUD，比如我们想把Session保存到数据库，那么可以实现自己的SessionDAO，通过如JDBC写到数据库；比如想把Session放到Memcached中，可以实现自己的Memcached SessionDAO；另外SessionDAO中可以使用Cache进行缓存，以提高性能；

CacheManager：缓存控制器，来管理如用户、角色、权限等的缓存的；因为这些数据基本上很少去改变，放到缓存中后可以提高访问的性能

Cryptography：密码模块，Shiro提高了一些常见的加密组件用于如密码加密/解密的。

4.3 应用程序使用Shiro



也就是说对于我们而言，最简单的一个Shiro应用：

1、应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；

2、我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro不提供维护用户/权限，而是通过Realm让开发人员自己注入。

4.4 Shiro的入门

4.4.1 搭建基于ini的运行环境

（1）创建工程导入shiro坐标

```
    <dependencies>
        <dependency>
            <groupId>org.apache.shiro</groupId>
            <artifactId>shiro-core</artifactId>
            <version>1.3.2</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
```

4.4.1 用户认证

认证：身份认证/登录，验证用户是不是拥有相应的身份。基于shiro的认证，是通过subject的login方法完成用户认证工作的

（1）在resource目录下创建shiro的ini配置文件构造模拟数据（shiro-auth.ini）

```
[users]
#模拟从数据库查询的用户
#数据格式    用户名=密码
zhangsan=123456
lisi=654321
```

（2）测试用户认证

```
    @Test
    public void testLogin() throws Exception{
        //1.加载ini配置文件创建SecurityManager
        Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro.ini");
        //2.获取securityManager
        SecurityManager securityManager = factory.getInstance();
        //3.将securityManager绑定到当前运行环境
        SecurityUtils.setSecurityManager(securityManager);
        //4.创建主体(此时的主体还为经过认证)
        Subject subject = SecurityUtils.getSubject();
        /**
         * 模拟登录，和传统等不同的是需要使用主体进行登录
         */
        //5.构造主体登录的凭证（即用户名/密码）
        //第一个参数：登录用户名，第二个参数：登录密码
        UsernamePasswordToken upToken = new UsernamePasswordToken("zhangsan","123456");
        //6.主体登录
        subject.login(upToken);
        //7.验证是否登录成功
        System.out.println("用户登录成功="+subject.isAuthenticated());
        //8.登录成功获取数据
        //getPrincipal 获取登录成功的安全数据
        System.out.println(subject.getPrincipal());
    }
```

4.4.2 用户授权

授权，即权限验证，验证某个已认证的用户是否拥有某个权限；即判断用户是否能做事情，常见的如：验证某个用户是否拥有某个角色。或者细粒度的验证某个用户对某个资源是否具有某个权限

（1）在resource目录下创建shiro的ini配置文件构造模拟数据（shiro-prem.ini）

```
[users]
#模拟从数据库查询的用户
#数据格式    用户名=密码,角色1,角色2..
zhangsan=123456,role1,role2
lisi=654321,role2
[roles]
#模拟从数据库查询的角色和权限列表
#数据格式    角色名=权限1，权限2
role1=user:save,user:update
role2=user:update,user.delete
role3=user.find
```

（2）完成用户授权

```
package cn.itcast.shiro;

import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.junit.Test;

public class ShiroTest1 {

    @Test
    public void testLogin() throws Exception{
        //1.加载ini配置文件创建SecurityManager
        Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro.ini");
        //2.获取securityManager
        SecurityManager securityManager = factory.getInstance();
        //3.将securityManager绑定到当前运行环境
        SecurityUtils.setSecurityManager(securityManager);
        //4.创建主体(此时的主体还为经过认证)
        Subject subject = SecurityUtils.getSubject();
        /**
         * 模拟登录，和传统等不同的是需要使用主体进行登录
         */
        //5.构造主体登录的凭证（即用户名/密码）
        //第一个参数：登录用户名，第二个参数：登录密码
        UsernamePasswordToken upToken = new UsernamePasswordToken("lisi","654321");
        //6.主体登录
        subject.login(upToken);

        //7.用户认证成功之后才可以完成授权工作
        boolean hasPerm = subject.isPermitted("user:save");

        System.out.println("用户是否具有save权限="+hasPerm);
```

```
    }
}
```

4.4.3 自定义Realm

Realm域：Shiro从Realm获取安全数据（如用户、角色、权限），就是说SecurityManager要验证用户身份，那么它需要从Realm获取相应的用户进行比较以确定用户身份是否合法；也需要从Realm得到用户相应的角色/权限进行验证用户是否能进行操作；可以把Realm看成DataSource，即安全数据源

（1）自定义Realm

```
package cn.itcast.shiro;

import org.apache.shiro.authc.*;
import org.apache.shiro.authz.AuthorizationInfo;
import org.apache.shiro.authz.SimpleAuthorizationInfo;
import org.apache.shiro.realm.AuthorizingRealm;
import org.apache.shiro.subject.PrincipalCollection;

import java.util.ArrayList;
import java.util.List;

/**
 * 自定义realm，需要继承AuthorizingRealm父类
 *      重写父类中的两个方法
 *          doGetAuthorizationInfo      :授权
 *          doGetAuthenticationInfo     :认证
 */
public class PermissionRealm extends AuthorizingRealm {

    @Override
    public void setName(String name) {
        super.setName("permissionRealm");
    }

    /**
     * 授权：授权的主要目的就是查询数据库获取用户的所有角色和权限信息
     */
    protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
        // 1.从principals获取已认证用户的信息
        String username = (String) principalCollection.getPrimaryPrincipal();

        /**
         * 正式系统：应该从数据库中根据用户名或者id查询
         *          这里为了方便演示，手动构造
         */
        // 2.模拟从数据库中查询的用户所有权限
        List<String> permissions = new ArrayList<String>();
        permissions.add("user:save");// 用户的创建
        permissions.add("user:update");// 商品添加权限
```

```
        // 3.模拟从数据库中查询的用户所有角色
        List<String> roles = new ArrayList<String>();
        roles.add("role1");
        roles.add("role2");

        // 4.构造权限数据
        SimpleAuthorizationInfo simpleAuthorizationInfo = new
SimpleAuthorizationInfo();
        // 5.将查询的权限数据保存到simpleAuthorizationInfo
        simpleAuthorizationInfo.addStringPermissions(permissions);
        // 6.将查询的角色数据保存到simpleAuthorizationInfo
        simpleAuthorizationInfo.addRoles(roles);

        return simpleAuthorizationInfo;
    }

    /**
     * 认证：认证的主要目的，比较用户输入的用户名密码是否和数据库中的一致
     */
    protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
        //1.获取登录的upToken
        UsernamePasswordToken upToken = (UsernamePasswordToken)authenticationToken;
        //2.获取输入的用户名密码
        String username = upToken.getUsername();
        String password = new String(upToken.getPassword());

        /**
         * 3.验证用户名密码是否正确
         * 正式系统：应该从数据库中查询用户并比较密码是否一致
         *          为了测试，只要输入的密码为123456则登录成功
         */
        if(!password.equals("123456")) {
            throw  new RuntimeException("用户名或密码错误");//抛出异常表示认证失败
        }else{
            SimpleAuthenticationInfo info = new SimpleAuthenticationInfo(username,
password,
                    this.getName());
            return info;
        }
    }
}
```

（2）配置shiro的ini配置文件（shiro-realm.ini）

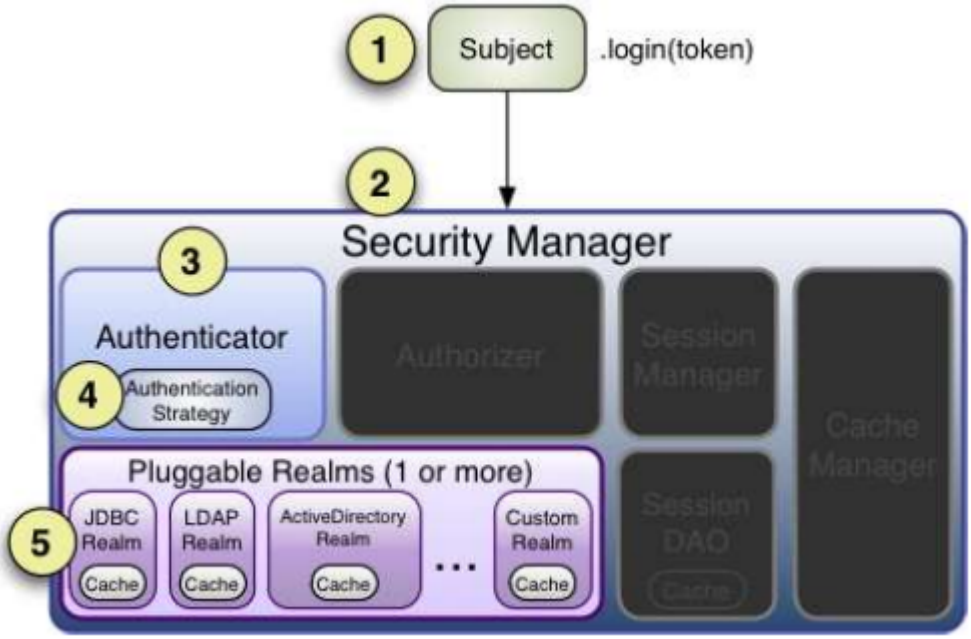
```
[main]
#声明realm
permReam=cn.itcast.shiro.PermissionRealm
#注册realm到securityManager中
securityManager.realms=$permReam
```

（3）验证

```
package cn.itcast.shiro;

import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.UsernamePasswordToken;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.junit.Before;
import org.junit.Test;

public class ShiroTest2 {

    private SecurityManager securityManager;

    @Before
    public void init() throws Exception{
        //1.加载ini配置文件创建SecurityManager
        Factory<SecurityManager> factory = new
IniSecurityManagerFactory("classpath:shiro-realm.ini");
        //2.获取securityManager
        SecurityManager securityManager = factory.getInstance();
        //13.将securityManager绑定到当前运行环境
        SecurityUtils.setSecurityManager(securityManager);
    }

    @Test
    public void testLogin() throws Exception{
        //1.创建主体(此时的主体还为经过认证)
        Subject subject = SecurityUtils.getSubject();
        //2.构造主体登录的凭证（即用户名/密码）
        UsernamePasswordToken upToken = new UsernamePasswordToken("lisi","123456");
        //3.主体登录
        subject.login(upToken);
        //登录成功验证是否具有role1角色
        //System.out.println("当前用户具有role1="+subject.hasRole("role3"));
        //登录成功验证是否具有某些权限
        System.out.println("当前用户具有user:save权限="+subject.isPermitted("user:save"));
    }
}
```

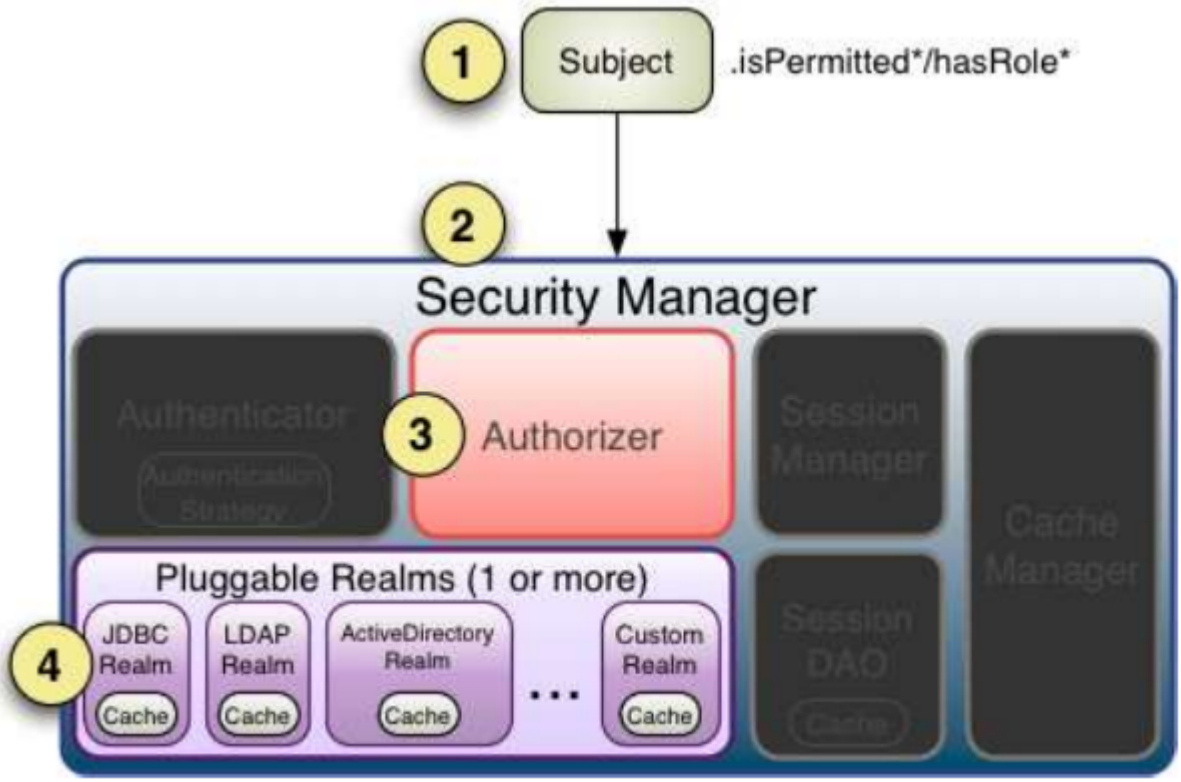
4.4.4 认证与授权的执行流程分析

（1）认证流程



1. 首先调用Subject.login(token)进行登录，其会自动委托给Security Manager，调用之前必须通过SecurityUtils. setSecurityManager()设置；
2. SecurityManager负责真正的身份验证逻辑；它会委托给Authenticator进行身份验证；
3. Authenticator才是真正的身份验证者，Shiro API中核心的身份认证入口点，此处可以自定义插入自己的实现；
4. Authenticator可能会委托给相应的AuthenticationStrategy进行多Realm身份验证，默认ModularRealmAuthenticator会调用AuthenticationStrategy进行多Realm身份验证；
5. Authenticator会把相应的token传入Realm，从Realm获取身份验证信息，如果没有返回/抛出异常表示身份验证失败了。此处可以配置多个Realm，将按照相应的顺序及策略进行访问。

（2）授权流程



1. 首先调用Subject.isPermitted/*hasRole*接口，其会委托给SecurityManager，而SecurityManager接着会委托给Authorizer；
2. Authorizer是真正的授权者，如果我们调用如isPermitted(“user:view”)，其首先会通过PermissionResolver把字符串转换成相应的Permission实例；
3. 在进行授权之前，其会调用相应的Realm获取Subject相应的角色/权限用于匹配传入的角色/权限；
4. Authorizer会判断Realm的角色/权限是否和传入的匹配，如果有多个Realm，会委托给ModularRealmAuthorizer进行循环判断，如果匹配如isPermitted/*hasRole*会返回true，否则返回false表示授权失败。