

# 平台搭建流程

## 项目背景/架构

这次实训，主要开发一个安全的产品。

产品的核心功能： 威胁感知

涉及到的核心技术栈：

Java17+SpringBoot+MyBatis+MySQL8+Redis+RabbitMQ+Python(3.10+)

这次产品： 平台端 + Agent

平台----指令---->Agent

平台-----RabbitMQ(必须部署到外网)-----Agent

平台端： Java+SpringBoot+(SpringSecurity)+MyBatis+MySQL+Redis

解耦： RabbitMQ

Agent： Python

## 环境的要求/搭建

java : 17

MySQL: 8.0 +

Python: 3.10+

准备一个虚拟机：Centos 7

编辑器：

Java的IDE: IDEA 2023.3

Python的IDE: Pycharm 2023.3

AI: AI插件必须利用起来

阿里: 通义灵码 (TONGYILINGMA)

## SpringBoot的基本使用

### 框架介绍

SpringBoot不能算是单独的框架： Spring+SpringMVC+自动配置的思想

Spring框架： Java里面使用最广泛的一个框架

Spring框架：

#### 1. 控制反转(IOC)

主要用来管理对象

日常程序员创建对象： new Object(), new申请内存的时候比较消耗资源

Spring的处理方式：

项目启动的时候， 准备一个容器，然后把所有需要的对象，一次性的初始化放在容器里面（键值对的形式存在）

当需要用这个对象的时候，直接根据名字去容器里面取，用完之后又丢回去

## 2. 依赖注入(DI)

对于Spring框架的体现，主要体现在，参数的注入

```
public void add(String name, Integer age){  
    // 方法体  
}
```

add("jack", 18);

// MVC: 设计模式

C: 控制层, 调度

M: 模型, 操作数据库

V: 视图, 负责渲染数据给用户

// 在Spring的控制层里面: Controller里面

在正常的开发里面, 离不开控制器的, 控制器主要用来接收用户的传参

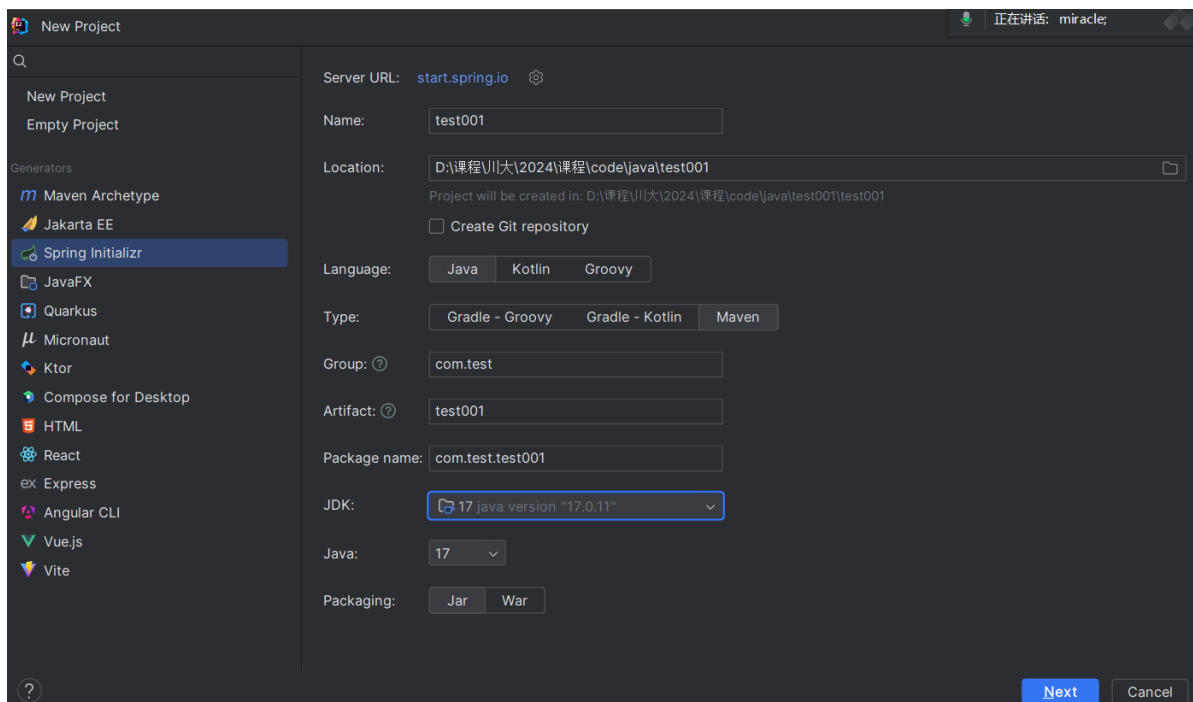
浏览器页面-----http协议----->Tomcat----->Java的程序, 接收到这个参数之后, 怎么将参数传递给控制器呢?

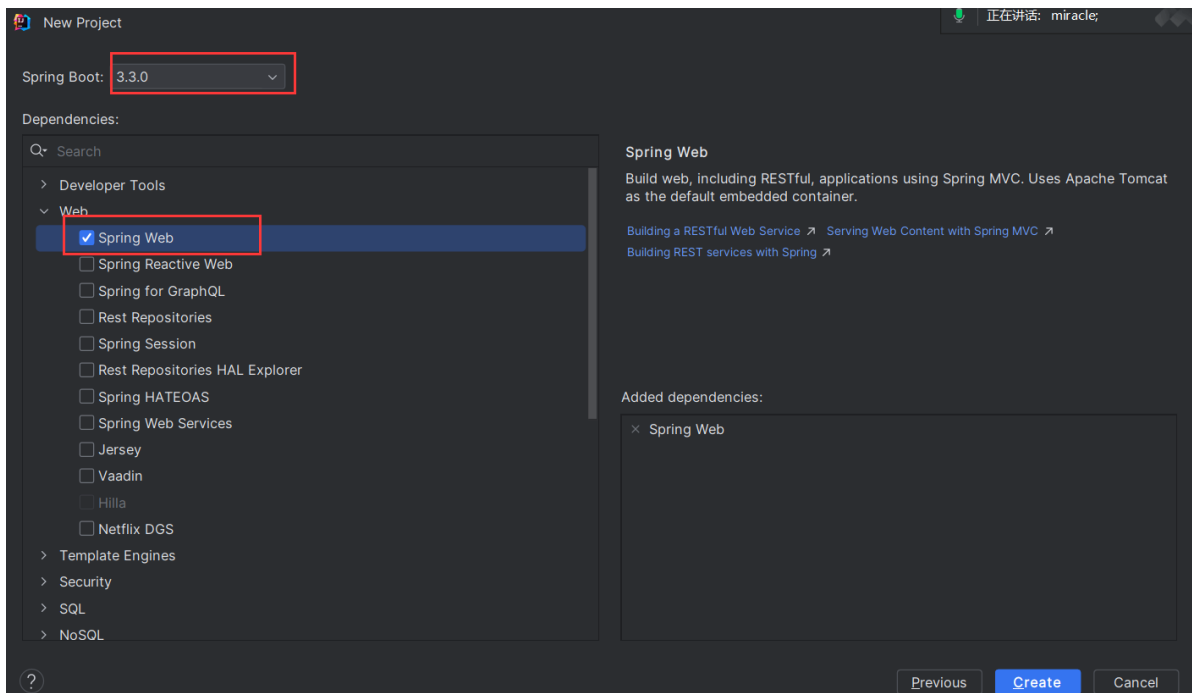
Java反射机制-----> DI -----> 动态的将参数直接注入的方法

SpringBoot的出现让死气沉沉的Java生态一下容光焕发。

## 框架的安装

安装很简单, 直接IDEA里面直接安装。

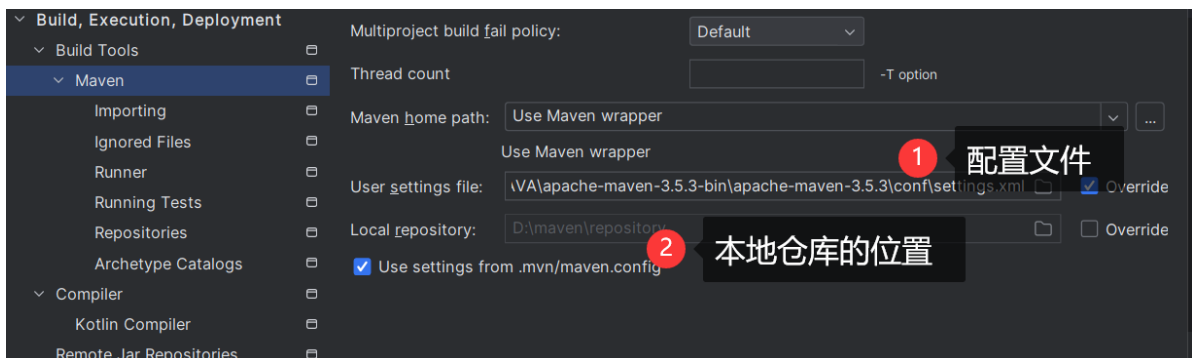




## 解决一下maven依赖的问题

由于网络问题，导致maven的依赖包下载不下来，解决办法：

1. 我将我的maven配置文件发给你们
2. 我将我的maven的本地仓库发给你们



## 框架的目录解析



## 框架的基本使用

这个地方主要掌握两个点：

### 1. 如何返回json数据

先添加一个依赖.pom.xml种

```
<!-- 添加fastJson的依赖-->
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.2.62</version>
</dependency>
```

注意需要刷新pom.xml

然后在Java代码里面直接返回对象即可： 重点注意@RestController

```
package com.test.test001.controller;

import com.test.test001.pojo.User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
// 在spring里面所有的对象都叫： bean
@RestController
public class TestController {
```

```

// 创建一个测试页面
@GetMapping("/test")
public String test(){
    return "Hello World!!!";
}

@GetMapping("/get/user")
public User getUser(){
    User user = new User();
    user.setUserName("jack");
    user.setUserPwd("123123");
    user.setAge(20);
    return user;
}
}

```

## 2. 如何接收JSON数据

```

@PostMapping("/add/user")
public User addUser(@RequestBody User user){
    user.setUserName("rose");
    return user;
}

```

## 3. 如何接收普通的数据

只需要接收一个id数据

```

@GetMapping("/get/id")
public Integer getId(Integer id){
    return id;
}

```

## 4. 如何接收对象

```

@RequestMapping("/add/user2")
public User addUser2(User user){
    user.setUserName(user.getUserName()+"_1001");
    return user;
}

```

# mybatis的使用

MyBatis是属于Java生态里面一个很重要的ORM框架(对象关系映射====> 数据库里面的一张表====>Java的一个实体类, Java的一个对象===》对应数据库里面的一行数据)

**Hibernate:** 完全的ORM框架, 有一个特点, 不用写SQL语句就能操作数据库, 复杂的查询, 复杂操作, 就会将代码写的很复杂, 可读性极低, 不利于维护

**MyBatis:** 半ORM框架, 有一个明显的特点, 程序员可以完全自定义SQL语句, 符合中国程序员的编码习惯

MyBatis-plus

spring-data-jpa

jpa

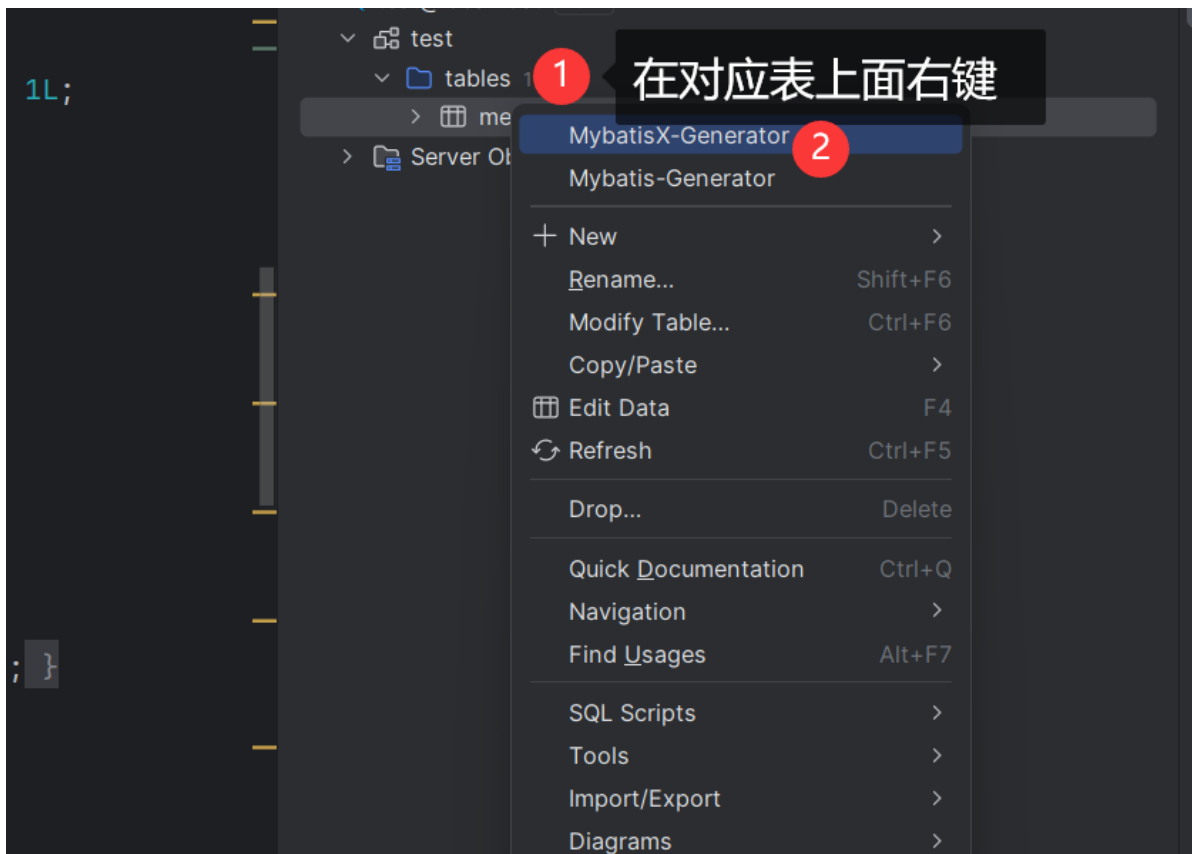
依赖:

```
<!--mybatis-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>3.0.3</version>
</dependency>
<!--数据库驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

配置:

```
# 配置MySQL
spring.datasource.url=jdbc:mysql://localhost:3306/threat_perception?
useUnicode=true&characterEncoding=utf-
8&useSSL=false&serverTimezone=Asia/Shanghai
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root
mybatis.type-aliases-package=com.tpp.threat_perception_platform.pojo
# mapper.xml的位置
mybatis.mapper-locations=classpath:/mapper/*.xml
# 输出日志
mybatis.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
# 自动转驼峰
mybatis.configuration.map-underscore-to-camel-case=true
```

利用mybatis-x这个插件, 帮我们生成了几个文件:



注意：生成之后需要改一下相关的包

pojo/Member.java ： 实体类， 承载数据

dao/MemberMapper.java： 这是一个接口， 专门用来操作数据库的接口

resources/mapper/MemberMapper.xml： 存放操作数据库的SQL语句

MyBatis特点： SQL语句和Java代码完全解耦

接下来，做一下CURD：

在Java的规范里面，MVC不能完全满足实际的需求，需要额外添加一层Service【处理业务逻辑】

Views<-----Controller----->**Service**----->**Model【dao/mapper】** ----->database

Views----->Controller----->**Service**----->**Model【dao/mapper】** ----->database

=====

=====

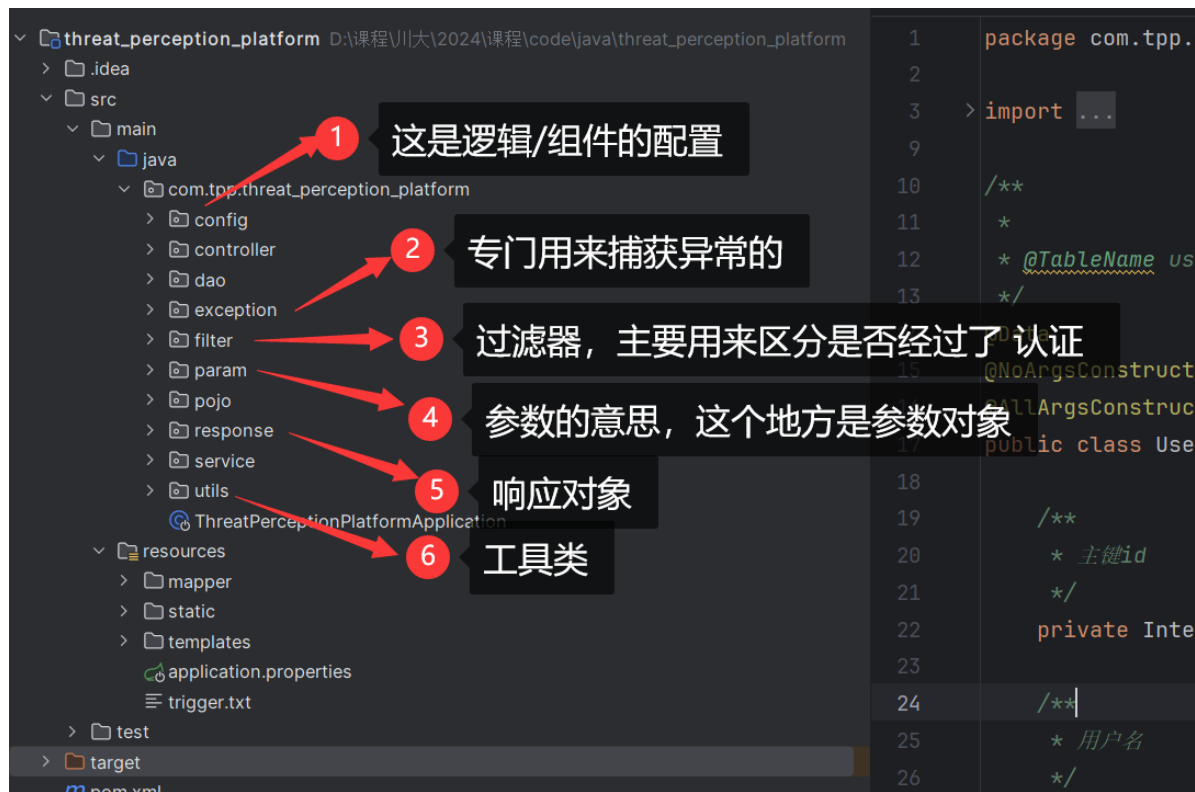
## 平台脚手架的使用

脚手架：

利用框架搭建的一个基础的架构，里面已经包含了需要用到的常用的类。

剩下的事情只需要去实现相关的业务逻辑即可。

脚手架的目录结构：



## 角色管理

增删改成

从后向前

先设计数据库，然后设计DAO，然后设计Service，在设计Controller，最后视图

## 设计角色表

表与表之间的关系：

用户表

角色表

标准的用户和角色之间是 多对多 简化一点： 多对一

```
CREATE TABLE `threat_perception2`.`role` (  
  `role_id` int UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '主键ID',  
  `role_name` varchar(20) NULL COMMENT '角色名字',  
  `role_desc` varchar(255) NULL COMMENT '角色描述',  
  PRIMARY KEY (`role_id`)  
);
```



## 创建pojo/mapper/mapper.xml

直接插件生成，注意修改包。

## 创建Service

service的规范： 一个接口， 多个实现类

接口： RoleService

add(Role role)

delete(Integer roleId)

update(Role role)

list(xxx)

实现类： RoleServiceImpl

@Service

```
public class RoleServiceImpl implements RoleService {
```

@Autowired

```
private RoleMapper roleMapper;
```

/\*\*

\* 角色列表

\* @return

\*/

@Override

```
public ResponseResult list() {
```

// 业务逻辑

```
List<Role> datas = roleMapper.findAll();
```

```
Long count = 5L;
```

```
return new ResponseResult(count, datas);
```

```
}
```

```
}
```

## 创建controller

@RestController

```
public class RoleController {
```

@Autowired

```
private RoleService roleService;
```

/\*\*

\* 角色列表

\* @return

\*/

```
@PostMapping("/role/list")
```

```
public ResponseResult list()
```

```
{
```

```
return roleService.list();
```

```
}
```

```
}
```

## 处理视图

分成两个部分：

### 1. 渲染页面

1. 先配置路由
2. 添加html页面

### 2. 渲染数据

```
table.render({
  elem: '#LAY-user-manage'
  ,method: 'post'
  ,url: '/role/list' //后端接口
  ,headers: {
    'Authorization': localStorage.getItem('token')
  }
  ,width: 'auto'// 自适应宽度
  ,cellMinWidth: 80 //全局定义常规单元格的最小宽度
  ,title: '系统用户数据表'
  ,cols: [
    [
      {type: 'checkbox', fixed: 'left'}
      ,{field: 'roleId', title: 'ID', width: 80}
      ,{field: 'roleName', title: '角色', width: 100}
      ,{field: 'roleDesc', title: '角色描述', width: 200}
      ,{fixed: 'right', title: '操作', toolbar: '#table-useradmin-webuser', width: 180}
    ]
  ]
  ,page: {
```

## 分页的问题

分页的核心作用是什么？减轻内存压力

```
# 分页的原理
# 分页
select * from role;
# 假设 2条一页
# 第一页
# limit的用法
# limit n : 限制条数
# limit index, len: 限制从index开始，然后查询n条
# 下标从 0 开始
select * from role limit 0,2;
# 第二页
select * from role limit 2,2;
# 第三页
select * from role limit 4,2;
# 假设 当前的页码 page
# 当前的页容量 pageSize
# page = 1; pageSize = 5
# 如何转换成SQL语句
# select * from role limit (page-1)*pageSize, pageSize;
```

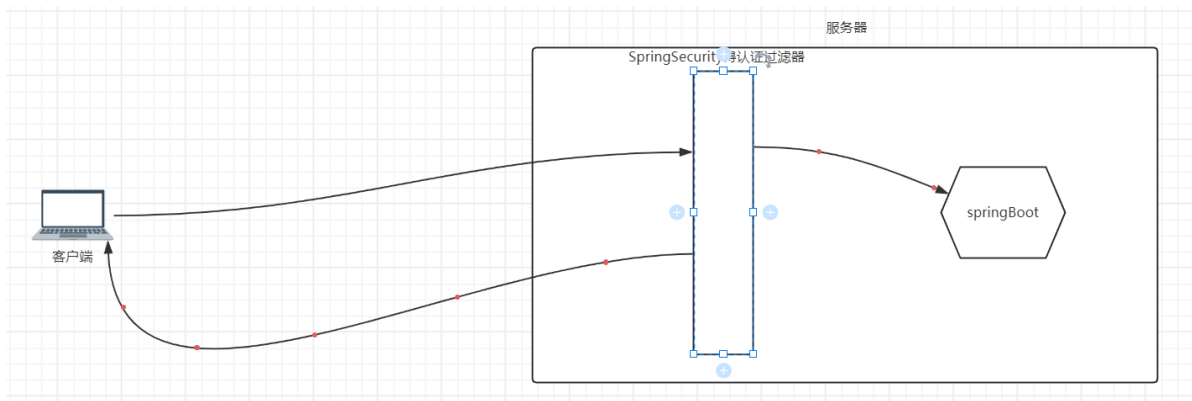
## 关于数据库ID自动增长

通常MySQL的每个表，都会有一个自增的主键ID  
我们添加数据的时候，没有指定这个id。

创建一个role表，有一个自增id:role\_id  
然后MySQL会自动的去创建一个对应这个role\_id的自增序列  
比如我们添加了一条数据，自增序列: 5, 这个role\_id: 5  
将5删掉，role表里面没有5，序列里面的5还在  
当我们添加新的数据的时候，序列的值变成了6

## springSecurity的认证流程

在Java生态里面，有两大权限框架: Apache shiro / SpringSecurity  
权限框架有两大功能: 1. 认证(登录) 2. 授权(是否有权限去操作某个功能)  
不用权限框架容易出现的问题: 越权(高危)  
所以项目上都得用权限框架去规避这个问题，又会产生一个问题，这个框架本身有问题(反序列化)



## redis的引入

redis的安装

```
$ wget https://download.redis.io/releases/redis-5.0.5.tar.gz
$ tar -zxvf redis-5.0.5.tar.gz
$ cd redis-5.0.5
$ make
# 如果出现找不到make, 执行: apt/yum install -y make gcc g++

# 编译完成之后重点进入src目录下面: 观察是否存在 redis-server/redis-cli

# 启动redis服务需要修改配置文件
# 1. # bind 127.0.0.1
# 2. protected-mode no
# 3. requirepass 123456

# 运行redis服务端
nohup ./src/redis-server ./redis.conf &
```

```
# 运行客户端
./src/redis-cli -a 密码

# 简单的指令测试
[root@192 redis-5.0.5]# ./src/redis-cli -a 123456
warning: Using a password with '-a' or '-u' option on the command line interface
may not be safe.
127.0.0.1:6379> set name jack
OK
127.0.0.1:6379> get name
"jack"
```

## 管理员数据表结构分析

## 登录的完整流程

用户输入用户名密码----->SpringSecurity得认证中心----->获取用户传递得用户名和密码----->利用这个信息去数据库里面查询----->将查询到数据和提交得数据进行比对----->不匹配----->认证失败  
----->匹配----->将用户得认证信息存放到redis，并且生成一个JWT，返回给客户端  
----->后续每次请求，前端都必须携带jwt，才能访问到资源。

redis的作用？

web应用的性能瓶颈是啥？----->数据库IO

内存级别的数据库----->nosql----->redis

JWT是什么？

JSON web token

核心目的是用来解决前后端分离项目的状态问题

也能解决很多安全问题(CSRF/XSS。。。。)

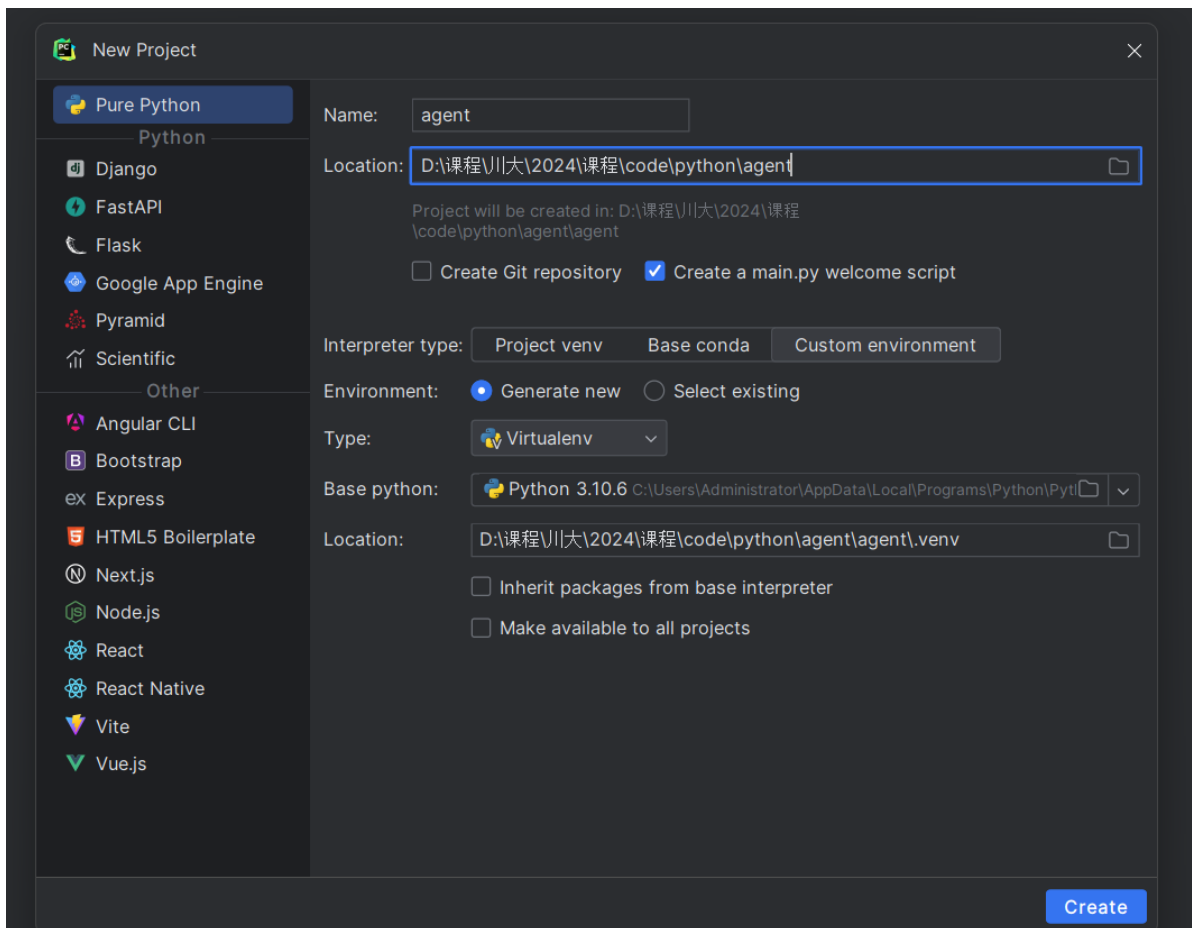
## Agent的开发

## python的安装和使用

版本： 3.10+

开发的IDE： Pycharm

python创建项目：



## 资产发现流程解析

Agent运行之后---->采集主机信息---(数据是以什么形式存在? JSON)-->将主机信息回传给平台  
信息如何回传给平台呢?

传统做法:

1. 现在平台端写一个接口
2. python去访问这个接口

相对稳妥的做法:

必须解耦, 解耦就必须用到中间件, =》消息队列rabbitmq

生成者(Python)====RabbitMQ====>消费者(Java)

## python读取主机信息

### 获取主机名

```
def __get_hostname(self):  
    """  
        获取主机名  
    :return:  
    """  
  
    self.__hostname = socket.gethostname()  
    self.__sys_info['hostname'] = self.__hostname
```

## 获取IP地址

```
def __get_ip(self):
    """
    获取IP地址
    :return:
    """

    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.connect(("114.114.114.114", 80))
    ip_address = s.getsockname()[0]
    s.close()
    self.__ip = ip_address
    self.__sys_info['ip'] = self.__ip
```

## 获取MAC地址

```
def __get_mac(self):
    """
    获取MAC地址：这个是逻辑MAC，但也是唯一的，只是做一个标识
    :return:
    """

    mac = ':'.join("%012X" % uuid.getnode())[i:i + 2] for i in range(0, 12, 2))
    self.__mac = mac
    self.__sys_info['mac'] = self.__mac
```

## 获取操作系统相关

```
def __get_os_type(self):
    """
    获取操作系统相关信息
    :return:
    """

    self.__os_type = platform.system()
    # 这个地方获取的可能和操作系统里面显示的不一样，显示的是出厂的操作系统
    # self.__os_name = platform.platform()
    # 这个可以获取实际的操作系统版本，但是不跨平台
    self.__os_name = self.__get_win_os_name()

    self.__os_version = platform.version()
    self.__os_bit = platform.architecture()[0]
    self.__sys_info['os_type'] = self.__os_type
    self.__sys_info['os_name'] = self.__os_name
    self.__sys_info['os_version'] = self.__os_version
    self.__sys_info['os_bit'] = self.__os_bit

def __get_win_os_name(self):
    """
    获取win的实际操作系统版本
    :return:
    """

    # 执行wmic命令获取操作系统名称
```

```

        process = subprocess.Popen(['wmic', 'os', 'get', 'Caption'],
stdout=subprocess.PIPE)
        output, _ = process.communicate()

        # 解析输出结果, 获取操作系统名称
        return output.strip().decode('GBK').split('\n')[1]

```

## 获取CPU相关信息

```

def __get_cpu_ram_info(self):
    """
    获取cpu,ram相关信息
    :return:
    """
    self.__cpu_type = self.__get_win_cpu_name()
    self.__ram = math.ceil(psutil.virtual_memory().total / 1024 / 1024 / 1024)
    self.__sys_info['cpu_type'] = self.__cpu_type
    self.__sys_info['ram'] = self.__ram

def __get_win_cpu_name(self):
    """
    获取win cpu 的名字
    :return:
    """
    process = subprocess.Popen(['wmic', 'cpu', 'get', 'name'],
stdout=subprocess.PIPE)
    output, _ = process.communicate()
    return output.strip().decode('utf-8').split('\n')[1]

```

## RabbitMQ的使用

安装:

```

安装docker:
# 安装
yum install -y docker
# 启动
systemctl start docker

```

下载带有管理界面的MQ版本:

```

docker pull docker.io/rabbitmq:3.7.17-management
# 检测是否有对应的镜像
docker images | grep mq

```

启动:

```

docker run -d --name scdx_mq -p 4568:5672 -p 15334:15672 -v
/var/rabbitmq/data2:/var/lib/rabbitmq --hostname myRabbit -e
RABBITMQ_DEFAULT_VHOST=my_vhost -e RABBITMQ_DEFAULT_USER=admin -e
RABBITMQ_DEFAULT_PASS=20240606 docker.io/rabbitmq:3.7.17-management

```

解释:

```

-p 4568:5672 # rabbitMQ 端口

```

```
-p 15334:15672 # rabbitMQ web端口
```

如果拉不下来镜像：需要修改镜像源---> 选择阿里镜像源

主机信息同步队列：

```
mq消息的发送：
python ----->mq
python ----->exchange---标识: sysinfo--->queue
```

```
exchange: sysinfo_exchange type: direct
queue: sysinfo_queue
binding_key: sysinfo
```

## Python操作MQ

```
class RabbitMQ:
    def __init__(self):
        self.__host = 'ip地址'
        self.__port = 端口
        self.__user = '用户'
        self.__password = '密码'
        self.__virtual_host = '虚拟机'
        self.__connection = ''
        self.__channel = ''

        # 初始化连接
        self.__get_connection()

    def __get_connection(self):
        """
        获取连接对象
        :return:
        """
        # 获取认证对象
        credentials = pika.PlainCredentials(self.__user, self.__password)
        # 获取连接
        self.__connection = pika.BlockingConnection(
            pika.ConnectionParameters(host=self.__host, port=self.__port,
            virtual_host=self.__virtual_host,
                                   credentials=credentials))

        # 获取通道
        self.__channel = self.__connection.channel()

    def __my_producer(self, exchange, routing_key, data):
        """
        生产者
        :param routing_key: 路由键
        """
```



```

        :param exchange: 交换机
        :param data: 数据
        :return:
        """
        self.__channel.basic_publish(exchange=exchange, routing_key=routing_key,
        body=data)

    def produce_sysinfo(self, data):
        """
        生产系统信息
        :return:
        """
        exchange = 'sysinfo_exchange'
        routing_key = 'sysinfo'
        # 发送数据
        self.__my_producer(exchange, routing_key, data)

```

## Java操作MQ

引入依赖:

```

<!--rabbitMQ-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>

```

配置:

```

# 配置RabbitMQ
spring.rabbitmq.host=ip
spring.rabbitmq.port=端口
spring.rabbitmq.username=用户名
spring.rabbitmq.password=密码
spring.rabbitmq.virtual-host=虚拟机
spring.rabbitmq.connection-timeout=15000
# 消费者配置
# 并发数
spring.rabbitmq.listener.simple.concurrency=5
spring.rabbitmq.listener.simple.max-concurrency=10
# 签收方式
spring.rabbitmq.listener.simple.acknowledge-mode>manual
# 限流: 每次一条
spring.rabbitmq.listener.simple.prefetch=1

```

template:

```

package com.tpp.threat_perception_platform.config;

import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.context.annotation.Bean;

```

```

import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMQConfig {

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory factory){
        return new RabbitTemplate(factory);
    }
}

```

consumer:

```

package com.tpp.threat_perception_platform.consumer;

import com.alibaba.fastjson.JSON;
import com.rabbitmq.client.Channel;
import com.tpp.threat_perception_platform.pojo.Host;
import com.tpp.threat_perception_platform.service.HostService;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.amqp.support.AmqpHeaders;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.Headers;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.Map;

@Component
public class RabbitSysInfoConsumer {

    @Autowired
    private HostService hostService;

    @RabbitListener(queues = "sysinfo_queue")

    public void receive(String message, @Headers Map<String, Object> headers,
        Channel channel) throws IOException {
        System.out.println("Received message: " + message);
        // 将数据存储到数据库
        Host host = JSON.parseObject(message, Host.class);
        // 存储到数据库
        int res = hostService.save(host);

        if (res > 0){
            // 手动 ACK, 先获取 deliveryTag
            Long deliveryTag = (Long)headers.get(AmqpHeaders.DELIVERY_TAG);
            // ACK
            channel.basicAck(deliveryTag, false);
        }
    }
}

```

# 主机信息入库

## 主机信息数据可视化

## 主机状态处理

主机的状态对于平台是什么意思？

主机处于开机状态/Agent是处于运行状态

这样平台发送的指令，Agent才能消费

思考如何维持主机状态的更新？

平台怎么认定这个主机处于上线状态？

数据库有一个字段：**status** 1 上线 0 下线

判断状态的模型：心跳检测机制

有两种模式：

1. 平台发送心跳检测，看客户端是否做出回应
2. 客户端主动发起心跳更新，平台接收到消息之后，更新状态和时间

心跳检测的思路：

**Agent**每隔3秒向平台发送一条指令，告诉平台自己还处于存活状态

平台接收到心跳指令之后，需要更新哪些数据？**status**，**update\_time**

平台判断客户端是否存活的最终依据是：**update\_time**是否超过3秒，超过3秒，客户端就死了，小于等于3秒处于存活状态

MQ队列：

**exchange:** sysinfo\_exchange

**queue:** status\_queue

**binding\_key:** status

## 资产探测

由平台发送探测指令到队列 **Agent**去消费队列

这个地方就需要每个**Agent**对应一个队列

队列在主机上线之后就自动创建，并且绑定交换机，**bing\_key**为**mac**，这样**routing\_key**也为**mac**，就能进行消息传递了

这里单独创建一个交换机：**agent\_exchange** **type:** direct

所有客户端的队列:           agent\_mac地址\_queue  
路由键:                   mac

平台发送的指令形式:

```
{"account":1,"service":1,"hostname":"miracle","mac":"82:30:49:99:96:73","process":1,"type":"assets","app":1}
```

account: 表示去探测账号

service: 表示去探测服务

process: 表示去探测进程

app: 表示去探测软件

1 表示探测 0 表示不需要探测

assets: 表示资产探测, 主要是为了和其他的探测做区分

## Java创建队列和发送数据到队列

```
package com.tpp.threat_perception_platform.service.impl;

import com.tpp.threat_perception_platform.service.RabbitService;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class RabbitServiceImpl implements RabbitService {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    /**
     * 创建队列
     *
     * @param exchange : 交换机名字
     * @param queue : 队列名字
     * @param routingKey : 路由键
     */
    @Override
    public void createAgentQueue(String exchange, String queue, String routingKey) {
        // 创建并绑定队列
        rabbitTemplate.execute(channel -> {
            // 创建队列
            channel.queueDeclare(queue, true, false, false, null);
            // 绑定交换机
            channel.queueBind(queue, exchange, routingKey);
            return null;
        });
    }

    /**
     * 发送消息到rabbitMQ
     * @param exchange : 交换机
     * @param routingKey : 路由键
     * @param message : 发送的消息
     */
}
```

```

@Override
public void sendMessage(String exchange, String routingKey, String message)
{
    // 发送消息到队列
    rabbitTemplate.convertAndSend(exchange, routingKey, message);
}
}

```

## Java将消息发送到队列

## python消费消息

```

def consume_queue(self, queue_name):
    """
    消费队列数据
    :param queue_name:
    :return:
    """
    # 消费队列
    self.__channel.basic_consume(queue=queue_name,
on_message_callback=self.__process_message, auto_ack=True)
    # 开始监听
    self.__channel.start_consuming()

def __process_message(self, ch, method, properties, message):
    """
    处理消息
    :param ch:
    :param properties:
    :param message:
    :return:
    """
    # JSON字符串转换成字典
    data = json.loads(message)
    # 判断类型
    if data['type'] == 'assets':
        # 资产探测
        assetsDetect = AssetsDetect(self, data)
        assetsDetect.detect()

```

## 探测账号

```
def __account_detect(self):
    """
    探测账号
    :return:
    """
    # 创建wmi客户端对象
    # 初始化
    pythoncom.CoInitialize()
    c = wmi.WMI()
    account_list = []
    # 获取所有用户
    for user in c.Win32_UserAccount():
        user_dict = {
            "mac": self.__data['mac'],
            "name": user.Name,
            "full_name": user.FullName,
            "sid": user.SID,
            "sid_type": user.SIDType,
            "status": user.Status,
            "disabled": user.Disabled,
            "lockout": user.Lockout,
            "password_changeable": user.PasswordChangeable,
            "password_expires": user.PasswordExpires,
            "password_required": user.PasswordRequired,
        }
        account_list.append(user_dict)
    # 去初始化
    pythoncom.CoUninitialize()
    # 转换成JSON字符串
    account_data = json.dumps(account_list)
    # 发送给队列
    self.__mq.produce_account_data(account_data)
    print("探测账号数据结束！")
```

队列信息:

交换机: sysinfo\_exchange  
队列: account\_queue  
路由键: account

## 探测服务

```
def __service_detect(self):
    """
    探测服务
    :return:
    """

    print('开始探测服务数据.....!')
    # 创建一个扫描仪对象
```

```

nm = nmap.PortScanner()
# 扫描目标主机
nm.scan(hosts='127.0.0.1', arguments='-sTV') # 指定扫描端口范围

# 获取扫描结果
state = nm.all_hosts()
# 装最终结果的
res_list = []
if state:
    for host in nm.all_hosts():
        for proto in nm[host].all_protocols():
            lport = nm[host][proto].keys()
            for port in lport:
                # 接收nmap扫描结果
                nmap_res = {
                    'mac': self.__data['mac'],
                    'protocol': proto,
                    'port': port,
                    'state': nm[host][proto][port]['state'],
                    'name': nm[host][proto][port]['name'],
                    'product': nm[host][proto][port]['product'],
                    'version': nm[host][proto][port]['version'],
                    'extrainfo': nm[host][proto][port]['extrainfo']}
                res_list.append(nmap_res)

# 转换成JSON字符串
res_json = json.dumps(res_list)
# 发送到队列
self.__mq.produce_service_data(res_json)
print("服务数据探测结束！")

```

队列信息：

```

交换机： sysinfo_exchange
队列： service_queue
路由键： service

```

## 探测进程

```

def __process_detect(self):
    # 获取进程信息
    # 初始化
    print('开始探测进程数据.....!')
    pythoncom.CoInitialize()
    c = wmi.WMI()
    process_list = []
    for process in c.Win32_Process():
        process_info = {
            'mac': self.__data['mac'],
            'pid': process.ProcessId,
            'ppid': process.ParentProcessId,

```

```

        'name': process.Name,
        'cmd': process.CommandLine,
        'priority': process.Priority,
        'description': process.Description,
    }
    process_list.append(process_info)
# 去初始化
pythoncom.CoUninitialize()
# 转换成JSON
process_data = json.dumps(process_list)
# 发送到队列
self.__mq.produce_process_data(process_data)
print("进程数据探测结束！")

```

队列信息：

```

交换机： sysinfo_exchange
队列： process_queue
路由键： process

```

## 探测安装的软件

```

def __app_detect(self):
    # 从注册表获取软件信息
    print('开始探测app数据.....!')
    registry_key = winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE,
    r'SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall')
    software_list = []
    # 获取软件数量
    number = winreg.QueryInfoKey(registry_key)[0]
    for i in range(number):
        try:
            sub_key_name = winreg.EnumKey(registry_key, i)
            sub_key = winreg.OpenKey(registry_key, sub_key_name)

            software = {}
            try:
                software['mac'] = self.__data['mac']
                software['display_name'] = winreg.QueryValueEx(sub_key,
                'DisplayName')[0]
                software['install_location'] = winreg.QueryValueEx(sub_key,
                'InstallLocation')[0]
                software['uninstall_string'] = winreg.QueryValueEx(sub_key,
                'UninstallString')[0]
                software_list.append(software)
            except windowsError:
                continue
        except windowsError:
            break
    # 转换成JSON字符串

```



```

app_data = json.dumps(software_list)
# 发送到队列
self.__mq.produce_app_data(app_data)

print("app数据探测结束! ")

```

队列信息:

```

交换机: sysinfo_exchange
队列: app_queue
路由键: app

```

## 风险发现

### 补丁安全

思路:

- 核心就是判断主机上面安装的hotfix是否安全/是否有漏洞?
1. 先在平台端构建一个CVE<==>补丁的数据库
  2. 再查询主机安装的补丁列表
  3. 根据补丁查询可能存在的问题

### 构建CVE补丁数据库

表:

```

CREATE TABLE `win_cve_db` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'id',
  `cve` char(128) NOT NULL COMMENT 'CVE id',
  `score` char(10) DEFAULT NULL COMMENT '各产品的CVSS Score平均值',
  `product_id_list` text COMMENT '漏洞影响范围, 以productid集表示',
  `kb_list` varchar(1024) DEFAULT NULL COMMENT '漏洞对应补丁号合集',
  `cvrf_id` char(128) DEFAULT NULL COMMENT 'cvrf id',
  `dt` char(128) NOT NULL COMMENT '数据日期',
  PRIMARY KEY (`id`,`dt`)
) ENGINE=MyISAM AUTO_INCREMENT=386 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci COMMENT='windows漏洞与KB补丁关系库'

```

```

CREATE TABLE `win_product_name` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT COMMENT 'id',
  `product_id` char(128) NOT NULL COMMENT 'product_id, 不需要比较大小, 所以set为字符串的格式',
  `product_name` varchar(1024) NOT NULL COMMENT 'product_id对应产品名',
  `dt` char(128) NOT NULL COMMENT '数据日期',
  PRIMARY KEY (`id`,`dt`)
) ENGINE=MyISAM AUTO_INCREMENT=205 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci COMMENT='product_id对应产品名'

```

对应py的代码:

```

# coding=utf-8
# Author: HSJ
# 2024/6/6 22:08

import requests
import datetime
import json
import calendar
import re
import pymysql

YEAR = str(datetime.datetime.now().year)
TODAY = datetime.datetime.now().strftime('%Y-%m-%d')
MONTH_IN_SHORT_EN = calendar.month_abbr[datetime.datetime.now().month]
# THIS_MONTH_ID = YEAR + "-" + MONTH_IN_SHORT_EN
THIS_MONTH_ID = YEAR + "-Mar"

print(THIS_MONTH_ID)
print(YEAR)

base_url = "https://api.msrc.microsoft.com/"
# windows api key
api_key = ""

def get_cvrf_json():
    db = pymysql.connect(host="localhost", port=3306, user="root",
password="root", db="threat_perception")
    cur = db.cursor()
    url = f"{base_url}cvrf/{THIS_MONTH_ID}?api-Version={YEAR}"
    headers = {'api-key': api_key, 'Accept': 'application/json'}
    response = requests.get(url, headers=headers)
    data = json.loads(response.content)

    for each_product in data["ProductTree"]["FullProductName"]:
        productid = each_product['ProductID']
        product_name = each_product['Value']

        search_productid_sql = f"SELECT * FROM win_product_name WHERE
product_id='{productid}'"
        cur.execute(search_productid_sql)
        if cur.rowcount == 0:
            insert_product_sql = f"INSERT INTO win_product_name
VALUES(null,'{productid}','{product_name}','{TODAY}')"
            cur.execute(insert_product_sql)
            db.commit()

    print('-----')
    for each_cve in data["Vulnerability"]:
        cve = each_cve["CVE"]
        if re.search('ADV', cve):
            pass
        else:
            cve = each_cve["CVE"]
            kblist = []

```

```

scorelist = []
product_id_list = str(each_cve["ProductStatuses"][0]
["ProductID"]).replace("'", "")
product_id_list = product_id_list.replace("[", "")
product_id_list = product_id_list.replace("]", "")
product_id_list = product_id_list.replace(" ", "")

for each_kb in each_cve["Remediations"]:
    try:
        kb_num = each_kb["Description"]["Value"]
        if re.search('Click to Run', kb_num) or re.search('Release
Notes', kb_num):
            pass
        else:
            kblist.append('KB{}'.format(kb_num))
    except Exception as e:
        print("kb", e)
kblist = list(set(kblist))
kblist = str(kblist).replace("'", "")
kblist = kblist.replace("[", "")
kblist = kblist.replace("]", "")
kblist = kblist.replace(" ", "")

for each_score in each_cve["CVSSScoreSets"]:
    try:
        scorelist.append(each_score["BaseScore"])
    except Exception as e:
        print("score", e)

try:
    score_mean = format(sum(scorelist) / len(scorelist), '.1f')
except Exception as e:
    print("score_mean", e)
score_mean = 0

insert_cve_sql = f"INSERT INTO win_cve_db
VALUES(null,'{cve}','{score_mean}','{product_id_list}','{kblist}','{THIS_MONTH_I
D}','{TODAY}')"
print(insert_cve_sql)
cur.execute(insert_cve_sql)
db.commit()

db.close()

if __name__ == '__main__':
    get_cvrif_json()

```

队列信息

```

exchange: sysinfo_exchange
queue: hotfix_queue
routing_key: hotfix

```

## 补丁发现

```
def __hotfix_discovery(self):
    """
    补丁发现
    :return:
    """

    print("开始补丁安全发现.....!")
    # 初始化
    pythoncom.CoInitialize()
    # 创建WMI客户端
    c = wmi.WMI()

    # 获取补丁信息
    hotfixes = c.query("SELECT HotFixID FROM Win32_QuickFixEngineering")

    # 组装补丁ID
    hotfix_list = []
    for hotfix in hotfixes:
        data = {
            'mac': self.__data['mac'],
            'hotfixId': hotfix.HotFixID
        }
        hotfix_list.append(data)

    # 去初始化
    pythoncom.CoUninitialize()
    # 转换成JSON数据
    data = json.dumps(hotfix_list)
    # 发送到队列
    self.__mq.produce_hotfix_data(data)
    print("补丁安全发现结束")
```

剩下的就是做数据可视化！

## 漏洞检测

思路：

1. 平台搭建漏洞库(漏洞名字，介绍，漏洞等级，请求类型，漏洞路径，payload)
2. 如果进行漏洞探测就会将漏洞库里面的漏洞都在主机上去进行尝试[可以进行筛选探测]
3. 客户端接收到指令之后进行漏洞探测，并将结果进行返回
4. 平台做可视化处理

## 构建漏洞库

```
#漏洞库的积累是需要时间，人力，所以这个地方仅仅演示原理即可
#推荐参考： exploit-db/国内厂商的漏洞平台(CNVD)/乌云漏洞库
# 需要创建一个表： vulnerability
CREATE TABLE `vulnerability` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '自增ID',
  `vul_name` varchar(100) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '漏洞名字',
  `vul_desc` varchar(200) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '漏洞描述',
```

```

`vul_level` int(10) unsigned DEFAULT NULL COMMENT '漏洞等级: 1 高危 2中危 3低危',
`vul_request_type` varchar(10) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT
'请求类型: GET/POST/PUT/DELETE',
`vul_type` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '漏洞类
型: SQL注入/反序列化',
`vul_path` varchar(200) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '漏洞产
生的路径',
`vul_payload` text COLLATE utf8mb4_general_ci COMMENT '漏洞攻击载荷',
`vul_flag` varchar(100) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '漏洞的
验证标记',
PRIMARY KEY (`id`)
) ENGINE=MyISAM AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_general_ci;

```

## Agent的漏扫模块

流程:

1. 平台将漏洞库发送给客户端
2. 客户端接收到漏洞规则之后，循环的去探测
3. 将探测结果返给Java平台
4. java做可视化(关于漏洞，平台更关注的是这个漏洞的影响范围--这个漏洞影响了多少台主机)

发起请求的模块:

```

# coding=utf-8
# Author: HSY
# 2024/6/17 14:39
import requests

class DoRequest:
    """
    专门用于发起请求的类: GET/POST
    """

    def __init__(self, path, method, data):
        # 请求的路径
        self.__path = path
        # 请求的方法
        self.__method = method
        # 请求的数据
        self.__data = data

    def do_request(self):
        """
        发起的方法
        :return:
        """
        if self.__method == 'GET':
            return self.__do_get()
        elif self.__method == 'POST':
            return self.__do_post()
        else:
            return None

```

```

def __do_get(self):
    """
    发起GET请求的方法
    :return:
    """

    # 准备请求的URL地址
    # http://www.baidu.com?name=xx&age=18
    url = self.__path + self.__data
    try:
        # 发起请求
        response = requests.get(url)
        # 获取响应结果并解码
        result = response.content.decode('utf-8')
        # 返回响应结果
        return result
    except Exception as e:
        print(e)
        return None

def __do_post(self):
    """
    发起POST请求的方法
    :return:
    """

    try:
        # 发起请求
        response = requests.post(url=self.__path, data=self.__data)
        # 获取响应结果并解码
        result = response.content.decode('utf-8')
        # 返回响应结果
        return result
    except Exception as e:
        print(e)
        return None

```

## 弱密码检测

思路:

1. 有两个点去探测弱密码
2. 系统账号弱密码
3. 系统服务弱密码(MySQL, FTP...)
4. Agent可以内置弱密码字典/也可以平台下发弱密码字典
5. Agent接收到平台探测弱密码的指令后, 执行探测, 并返回探测结果
6. 平台进行可视化

windows系统smb登录的方法:

```

from impacket.examples.utils import parse_target
from impacket.smbconnection import SMBConnection

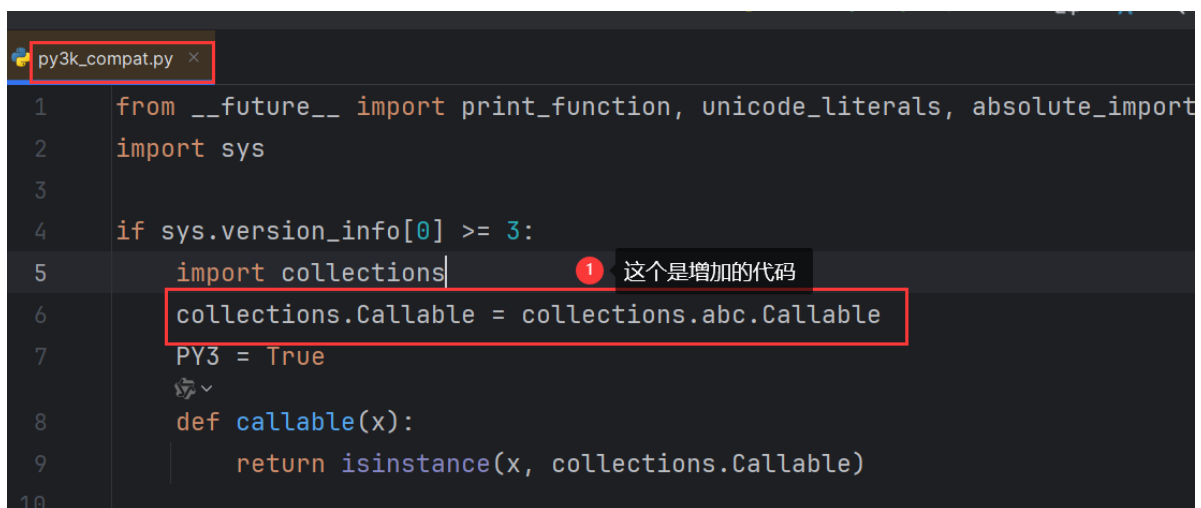
def smb_login():
    # [[domain/]username[:password]@]<targetName or address>
    target = 'test:123124@127.0.0.1'
    domain, username, password, address = parse_target(target)
    target_ip = address
    domain = ''
    lmhash = ''
    nthash = ''

    try:
        smbClient = SMBConnection(address, target_ip, sess_port=int(445))
        smbClient.login(username, password, domain, lmhash, nthash)
        print("登录成功!")
    except Exception as e:
        print(e)
        print("登录失败!")

if __name__ == "__main__":
    smb_login()

```

需要额外安装: impacket (杀毒软件需要信任), pyreadline包  
 以及需要修改: 当前虚拟环境中: py3k\_compat.py里面的代码  
 collections.Callable = collections.abc.Callable



```

py3k_compat.py x
1 from __future__ import print_function, unicode_literals, absolute_import
2 import sys
3
4 if sys.version_info[0] >= 3:
5     import collections 1 这个是增加的代码
6     collections.Callable = collections.abc.Callable
7     PY3 = True
8     def callable(x):
9         return isinstance(x, collections.Callable)
10

```

## 应用风险

应用风险，指的是系统搭建的服务(中间件居多)，服务有风险思路同漏洞检测一致，需要构建应用风险库，利用应用风险库去检测比如：

1. **tomcat**运行权限检查
  2. **ftp**用户权限可跨目录
  3. **tomcat** 404错误页面重定向配置检查
- 等等....

## 系统风险

思路同漏洞检测一致，需要构建系统风险库，利用系统风险库去检测  
比如：

1. 服务器时间校验
2. 路由转发功能开启
3. 网卡处于混杂模式
- 等等....

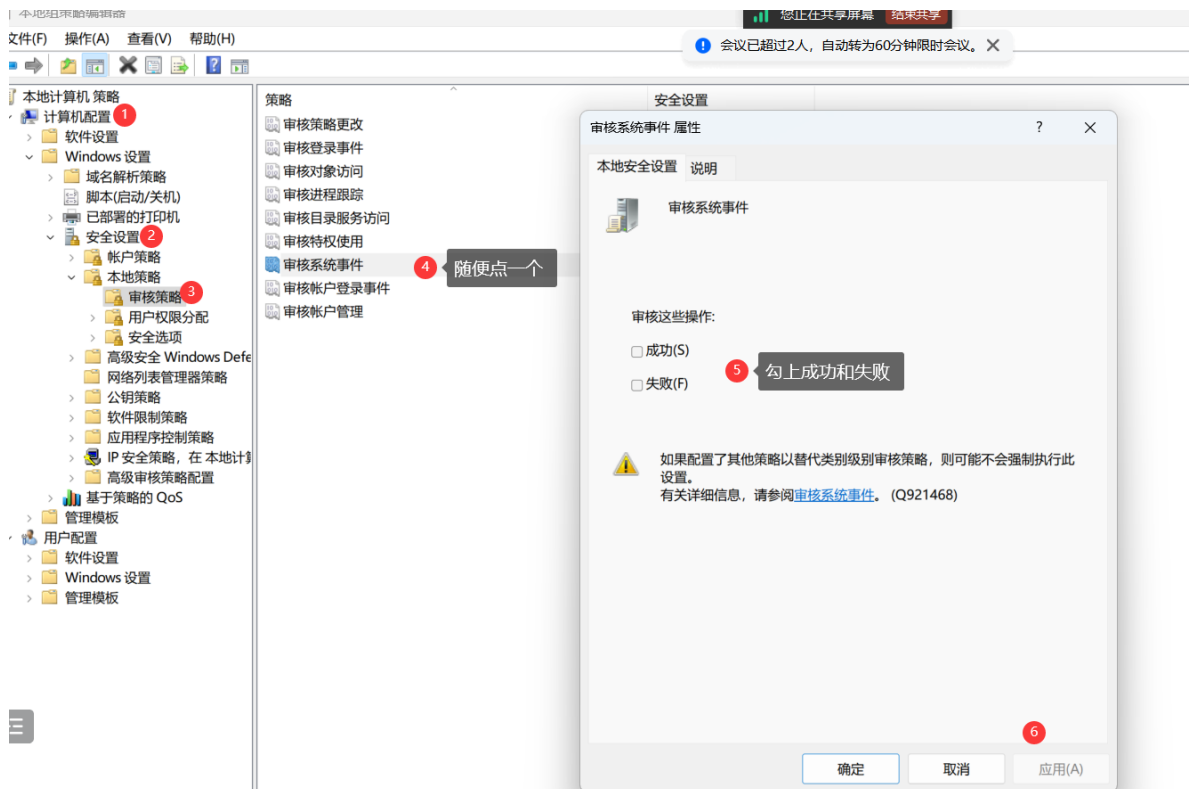
## 安全日志

## 日志记录对于安全来说：不可或缺

日志主要是用来记录使用者在这个操作系统中做了什么事情

## 开启日志审计功能

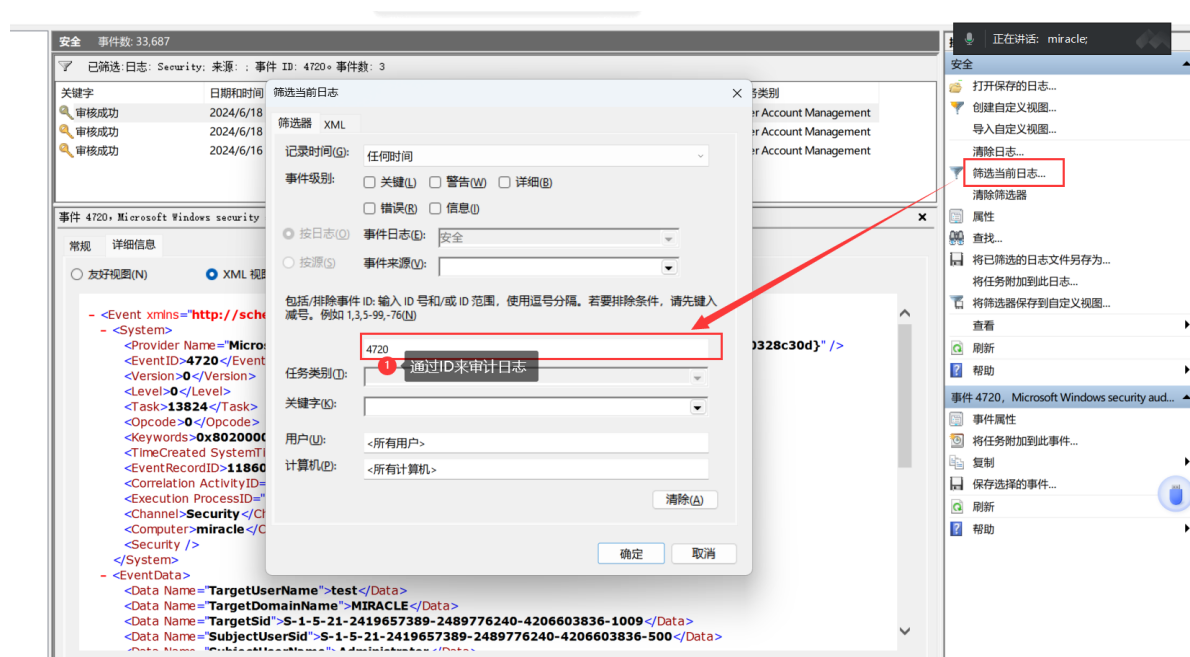
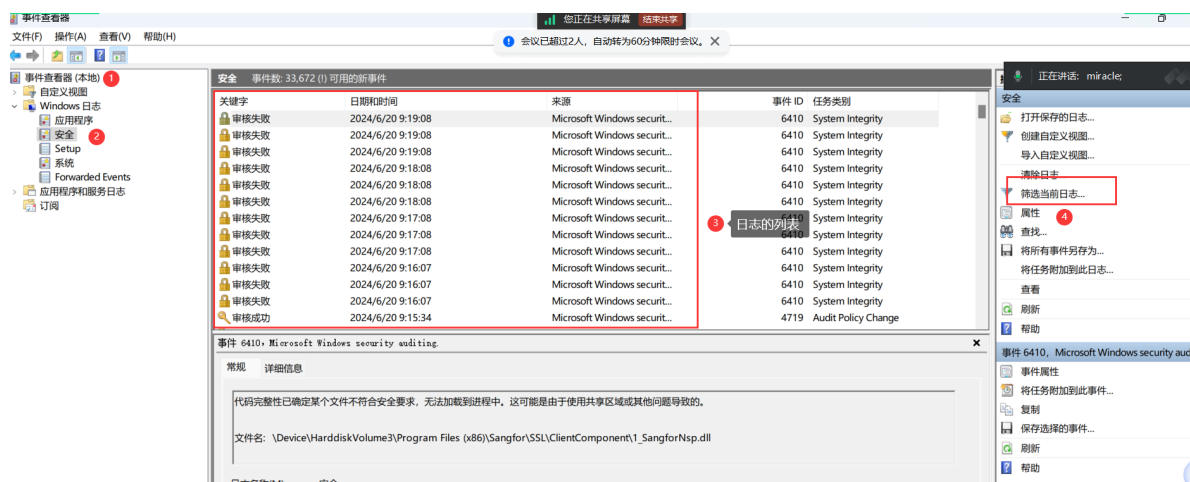
gpedit.msc 打开组策略编辑器





# 手工审计windows日志

eventvwr.msc 事件查看器



审计日志的思路:

1. 如果发现创建了一个新的用户 test
2. 再去看test什么时候登录过? --- 登录时间
3. 再去看test什么时候注销的? --- 注销时间
4. 接下来的精力就放在登录时间和注销时间之间去通过不同的事件ID去做审计

## windows常见事件ID

常见的安全事件ID:

- 4624 - 成功登录事件, 表示用户成功登录系统。
- 4625 - 登录失败事件, 表示用户尝试但未能成功登录系统。
- 4634 - 注销事件, 表示用户注销系统。
- 4647 - 用户注销事件, 表示用户通过重新启动或关闭计算机来注销系统。
- 4720 - 创建用户事件, 表示新用户帐户已创建。
- 4722 - 启用用户帐户事件, 表示禁用的用户帐户已被启用。

- 4723 - 更改用户密码事件，表示用户密码已更改。
- 4724 - 创建安全组事件，表示新安全组已创建。
- 4728 - 成功授权事件，表示用户获得了指定对象的权限。
- 4738 - 设置用户密码事件，表示用户密码已更改或重置。
- 4726 - 删除用户

常见的系统事件ID:

- 1074 - 通过这个事件ID查看计算机的开机、关机、重启的时间以及原因和注释。
- 6005 - 表示计算机日志服务已启动，如果出现了事件ID为6005，则表示这天正常启动了系统。
- 6006 - 系统关闭
- 6008 - 非正常关机
- 6009 - 系统已经重新启动
- 6013 - 系统已经重新启动，原因是操作系统版本升级
- 7036 - 服务状态更改
- 7040 - 启动或停止监视者通知
- 7045 - 安装服务

日志清除事件:

- 1102 - 这个事件ID记录所有审计日志清除事件，当有日志被清除时，出现此事件ID。

## windows日志文件位置

安全事件日志:

C:\Windows\system32\winevt\Logs\Security.evtx

系统事件日志:

C:\Windows\system32\winevt\Logs\System.evtx

## Python获取windows日志的方法

```
import html
from xml.dom import minidom
import Evtx.Evtx as evt

def get_log_record(event_path, event_id):
    """
    获取对应事件id的日志内容
    :param event_path: 事件日志路径
    :param event_id: 事件id
    :return:
    """
    event_res = []
    with evt.Evtx(event_path) as log:
        for record in log.records():
            timestamp = record.timestamp().timestamp()
            r = {}
            xml_doc = minidom.parseString(record.xml())
            # 事件ID 例如 4624登录成功, 4625登录失败
            id_ = xml_doc.getElementsByTagName('EventID')[0].childNodes[0].data
            if id_ != event_id:
                continue
            r['id'] = id_
            r['timestamp'] = timestamp
            data = xml_doc.getElementsByTagName('Data')
            for d in data:
                try:
                    name = d.getAttribute('Name')
```

```

        value = html.unescape(d.childNodes[0].data)
        r[name] = value
    except Exception as e:
        pass

    event_res.append(r)
return event_res

if __name__ == '__main__':
    path = r"C:\Windows\system32\winevt\Logs\Security.evtx"
    event_id = '4720'
    event_res = get_log_record(path, event_id)
    for log in event_res:
        print(log)

```

```

# coding=utf-8
# Author: HSJ
# 2024/6/20 14:03

from Evtx import PyEvtxParser
import re
import html
from xml.dom import minidom

def get_log_info(event_path, event_id_to_filter=None):
    """
    过滤自己想要的日志
    :param event_path: 日志的路径
    :param event_id_to_filter: 过滤的日志ID, 如果为None, 则返回所有日志
    :return:
    """

    # 创建一个EvtxParser对象来读取文件
    parser = PyEvtxParser(event_path)
    pattern = re.compile(r'<EventID>(\d+)</EventID>')
    # 给一个容器, 装最终的事件
    event_list = []
    # 遍历日志条目, 筛选出特定事件ID的日志
    for record in parser.records():
        # print(record)
        xml_data = record['data']
        res = re.findall(pattern, xml_data)
        event_id = int(res[0])

        if event_id_to_filter is not None and event_id == event_id_to_filter:
            r = {}
            r['event_id'] = event_id
            r['timestamp'] = record['timestamp']
            xml_doc = minidom.parseString(xml_data)
            data = xml_doc.getElementsByTagName('Data')
            for d in data:
                try:
                    name = d.getAttribute('Name')
                    value = html.unescape(d.childNodes[0].data)
                    r[name] = value
                except Exception as e:

```

```

        pass
    event_list.append(r)
elif event_id_to_filter is None:
    # 返回所有的日志
    r = {}
    r['event_id'] = event_id
    r['timestamp'] = record['timestamp']
    xml_doc = minidom.parseString(xml_data)
    data = xml_doc.getElementsByTagName('Data')
    for d in data:
        try:
            name = d.getAttribute('Name')
            value = html.unescape(d.childNodes[0].data)
            r[name] = value
        except Exception as e:
            pass
    event_list.append(r)
return event_list

if __name__ == '__main__':
    # 指定EVTX文件路径
    path = r"C:\Windows\system32\winevt\Logs\Security.evtx"
    event_list = get_log_info(path, 4726)
    for event in event_list:
        print(event)

```

思考日志怎么去同步？

思考一个策略：  
 读取日志，肯定是要消耗性能的  
 思考：是否一直同步日志呢？  
 所以思考在平台端设置一个开关，这个开关决定是否去同步日志

## 审计日志

将windows的日志转移到平台进行可视化展示

## 登录日志

专门审计登录日志

## 账号变更日志

专门审计账号变更相关日志

# 合规基线

## 基线的含义

基线： 安全基线

安全的最低的底线，安全的最低要求 ===》 操作系统基线 =====》操作系统最低的安全要求=====》各项安全相关的配置

操作系统=====》文件系统=====》各种配置都在一个文件里面

基线核查===》利用脚本/人工 去检查相关的安全配置是否满足要求

思路：

基线核查/加固不能随时去跑，通常都是在业务不繁忙的时候去跑。

所以我们得规定一个时间去做基线核查，这个地方就得用到定时器

1. 设置基线任务
2. 定时得发送任务给Agent
3. Agent按照内置得基线规则或平台下发得基线规则进行基线核查
4. Agent将核查得数据返回给平台
5. 平台做可视化

## Java开启定时任务

入口文件加入对应得注解：

```
7
8 @SpringBootApplication
9 @MapperScan("com.thpp.threat_perception_platform.dao")
10 @EnableScheduling
11 public class ThreatPerceptionPlatformApplication {
12
13     public static void main(String[] args) { SpringApplication.run(ThreatPercep
14
15
16
17
18 }
```

对应任务得Java代码：

```
package com.thpp.threat_perception_platform.task;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;

/**
 * 定时任务类
 */
@Component
public class TimeTask {

    /**
     * 每隔5秒进行一次动作
     */
}
```

```

    @Scheduled(cron = "*/5 * * * *")
    public void task01(){
        // 每隔5秒执行一次动作
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
        System.out.println("每隔5秒执行一次动作: " +
simpleDateFormat.format(System.currentTimeMillis()));
    }
}

```

## 基线任务功能

基线任务功能就是平台添加的一个定时执行基线检查的一个功能

1. 任务---规定一个执行的时间
2. 任务---绑定需要做基线核查的主机
3. 任务---执行的时候，就会将基线的指令发送给需要做基线核查的主机

先创建任务表：

表名： base\_line\_task  
 字段：  
 id, task\_name, task\_time, task\_status, task\_hosts

```

CREATE TABLE `base_line_task` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键自增ID',
  `task_name` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '任务
名',
  `task_time` datetime DEFAULT NULL COMMENT '任务执行时间',
  `task_status` int(10) unsigned DEFAULT NULL,
  `task_hosts` text COLLATE utf8mb4_general_ci COMMENT '需要执行任务的主机mac地址',
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;

```

## Python进行基线核查

可以在github上面去找一个基线核查脚本，这个不用自己写，没有必要重复造轮子！

windows的基线核查脚本通常是powershell脚本，所以这个地方要python去执行powershell脚本。

```
import subprocess

# 定义PowerShell命令
ps_command = 'powershell -ExecutionPolicy bypass -File ./ps/windows.ps1'

# 使用subprocess.run来运行PowerShell命令
result = subprocess.run(['powershell', '-Command', ps_command],
stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

# 打印输出和错误信息
print(result.stdout) # 输出信息
print(result.stderr) # 错误信息
```

这个脚本运行之后会产生一个文件，可以利用python去处理这个结果文件，处理之后再返回给平台即可。