一、功能说明

伴随部门与技术中心的技术融合演进, 以及自身技术迭代的需要, 注册中心的多样性逐步展现

为了保证系统间的交互安全, 一般采用签名方案, 于是补全该文档, 该工具具备能力如下:

- HTTP交互
- 调用方主动签名
- 被调方主动验签

二、工具使用

- 添加依赖: pom.xml

```
<dependency>
    <groupId>cn.hll.tools</groupId>
    <artifactId>tools-base</artifactId>
    <version>4.2.6</version>
</dependency>
```

- 调用方签名: (如果你是调用方)

> Sign可以被继承/重写, 扩展一些自定义的签名字段、业务字段, 比如: 令牌、订单号等 (扩展字段均有无法篡改效果)

```
// 创建签名参数对象
Sign sign = new Sign(appId);
// 补全签名字段 (sign参数也可以是个Map)
SignUtil.sign(sign, appKey);
// 生成HTTP请求所需要的请求头
Kv<String, String>[] kvs = SignUtil.getHeaderKvs(sign);
// 发起请求
HttpResult httpResult = HttpUtil.post(url, param, kvs);
// 返回结果
return httpResult.jsonResponseBody2bean(reference);
```

- 被调方验签: (如果你是被调方)

> 只需要将需要验签的接口置于继承了SignController的控制器中即可, 接口可专注与业务开发, 无需关注验签事宜

```
@RequestMapping("api/user")
public class UserApi extends SignController {
    /**
    * 案例接口: 注册用户, 凡是进入该方法的请求皆已通过验签
    */
    @PostMapping("register")
    public JsonResult<User> register() {
        // 专注业务代码无需关注签名..
    }
}
```

> 到这里已经完成, 除非密钥保存不是在redis中, 或扩展了签名字段, 那么你需要继续阅读

- 高阶验签用法
  - 自定义密钥存储

```
@RequestMapping("api/user")
public class UserApi extends SignController {

    @Resource
    private SysService sysService;

    @Override
    public String getSecretKey(String appId) {
        // 从数据库中获取密钥
        this.sys = super.sysService.getById(appId);
        return this.sys.getAppKey();
    }
}
```

```
# 默认redis密钥存储位置如下(可更改)
tools.webapp.sign.appKeyPrefix = REDIS:SIGN:APP_KEY_
```

  - 自定义验签方式: (如果需要)

> 如果SignController不好用/不满足要求, 可以自己代码验签, TOOLS也提供了快捷工具

```
@RequestMapping("api/user")
public class UserApi  {

    /**
    * 案例接口: 注册用户, 代码验签或者开发拦截器/过滤器验签
    * (此处采用代码案例演示)
    */
    @PostMapping("register")
    public JsonResult<User> register() {
        HttpServletRequest request = WebUtil.getRequest();
        boolean result = SignUtil.check(request, "123456");
        // 一般会断言验签结果, 但也可以开发者自由处理..
    }
}
```

  - 自定义扩展字段

> 当若业务系统扩展了自己的签名字段, 那么需要开发者取出请求中的扩展字段加入到验签中

```
@RequestMapping("api/user")
public class UserApi  {

    /**
    * 案例接口: 注册用户, 扩展签名字段.
    */
    @PostMapping("register")
    public JsonResult<User> register(User user) {
```

```
        // 获取签名请求头
        Map<String, Object> signMap =
SignUtil.getHeaderMap(WebUtil.getRequest());
        signMap.addAll(ClassUtil.generateMap(user));
        boolean result = SignUtil.check(request, "123456");
        // 一般会断言验签结果，但也可以开发者自由处理..
    }
}
```

- 解锁更多用法...

三、工具原理

逻辑

- 该工具通过密钥保证非法请求不可模仿/恶意请求
- 调用方按照与被调方的约定进行参数签名(非对称加密)

> 将所有不希望被篡改的参数与签名字段(appid,nonce,timestamp)进行加密并将加密结果(signature)传输给被调方

- 被调方则也按照约定进行参数签名, 并比对调用方的签名是否一致

> 不一致则被篡改过, 属于非法请求, 一致则是拥有密钥以及知晓签名算法的合法用户

代码

1. 通过SignUtil进行参数签名
2. 通过HttpUtil工具发起接口请求并将签名内容以请求头的方式传递

```
HttpResult httpResult = HttpUtil.post(url, param, kvs);
```

3. 继承SignController对所在控制器的接口要求签名并验签

```
@RequestMapping("api/user")
public class UserApi extends SignController {
}
```

数据

- 默认将密钥存储在redis中, 直到废弃才清理~

四、使用示例

- 日志样本

  ```
  待补
  ```

- 日志说明

  ```
  待补
  ```